

CS5460/6460: Operating Systems

Lecture 13: Memory barriers

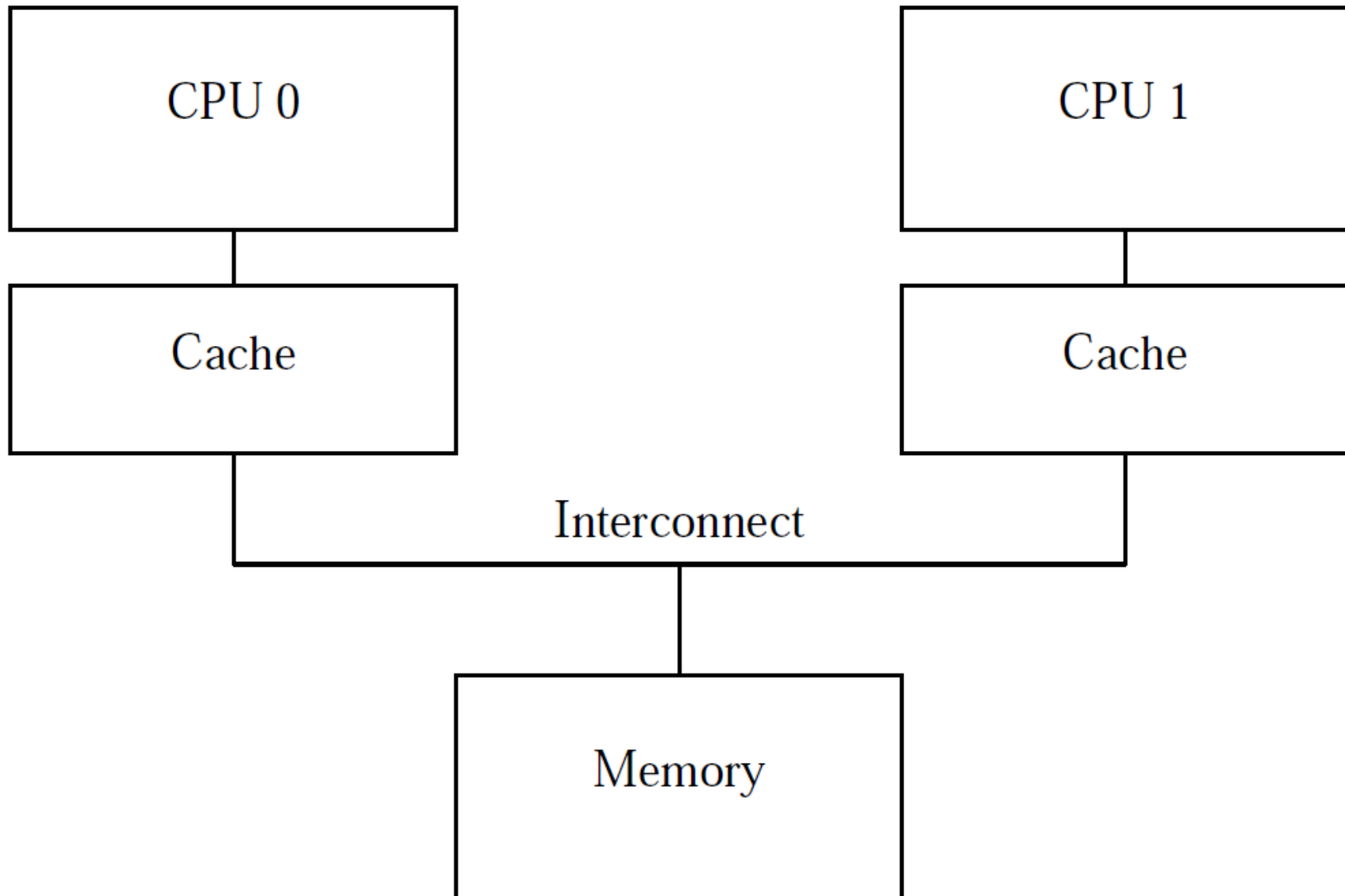
Anton Burtsev
February, 2014

Recap from last time


- Two synchronization paradigm
 - Critical sections
 - Implemented as spinlocks
 - Doesn't scale (atomic operations don't scale)
 - Read Copy Update
 - Lock-free
 - Scales well for a large number of readers
 - Updates must be infrequent

Cache coherence

CPU cache



CPU cache

- Effectively a hash 
 - Contains sequential blocks of memory
 - Called “cache lines”, 16 – 256 bytes each
 - Simple hashing function
 - E.g. 16-set cache means, 16 buckets
 - Limited chain length to resolve conflicts
 - E.g. 2-way cache means chain of length 2

Cache hierarchy

- Hierarchy of caches: L1, L2, L3
- Grow in size but get slower
- Ivy Bridge-EP
 - L1 cache 32 KB per core
 - L2 cache 256 KB per core
 - L3 cache 10 MB to 30 MB shared

Some latency numbers

- L1 cache reference 0.5 ns
 - On a 2GHz CPU, 1 ns == 2 cycles
- L2 cache reference 7 ns
- L3 cache reference 15 ns
- Main memory reference 100 ns
- Send 1K bytes over 1 Gbps network 10,000 ns
- Read 4K randomly from SSD* 150,000 ns
- Round trip within same datacenter 500,000 ns
- Disk seek 10,000,000 ns

Example: 16 set, 2-way, 256 byte line

| | Way 0 | Way 1 |
|-----|------------|------------|
| 0x0 | 0x12345000 | |
| 0x1 | 0x12345100 | |
| 0x2 | 0x12345200 | |
| 0x3 | 0x12345300 | |
| 0x4 | 0x12345400 | |
| 0x5 | 0x12345500 | |
| 0x6 | 0x12345600 | |
| 0x7 | 0x12345700 | |
| 0x8 | 0x12345800 | |
| 0x9 | 0x12345900 | |
| 0xA | 0x12345A00 | |
| 0xB | 0x12345B00 | |
| 0xC | 0x12345C00 | |
| 0xD | 0x12345D00 | |
| 0xE | 0x12345E00 | 0x43210E00 |
| 0xF | | |

Example: 16 set, 2-way, 256 byte line

| | Way 0 | Way 1 |
|-----|------------|------------|
| 0x0 | 0x12345000 | |
| 0x1 | 0x12345100 | |
| 0x2 | 0x12345200 | |
| 0x3 | 0x12345300 | |
| 0x4 | 0x12345400 | |
| 0x5 | 0x12345500 | |
| 0x6 | 0x12345600 | |
| 0x7 | 0x12345700 | |
| 0x8 | 0x12345800 | |
| 0x9 | 0x12345900 | |
| 0xA | 0x12345A00 | |
| 0xB | 0x12345B00 | |
| 0xC | 0x12345C00 | |
| 0xD | 0x12345D00 | |
| 0xE | 0x12345E00 | 0x43210E00 |
| 0xF | | |

Cache coherency protocol

- MESI: 4 states:
 - Modified
 - Exclusive
 - Shared
 - Invalid

Protocol messages

- Read
 - The physical address of the cache line to be read
- Read response
 - The data requested by an earlier “read”
 - Might be supplied either by memory or by one of the caches
 - If one of the caches has the desired data in “modified” state, that cache must supply the “read response” message.

Protocol messages (2)

- Invalidate
 - The physical address of the cache line to be invalidated
- Invalidate acknowledge
 - A CPU receiving an “invalidate” message must respond with an “invalidate acknowledge” message after removing the specified data from its cache.

Protocol messages (3)

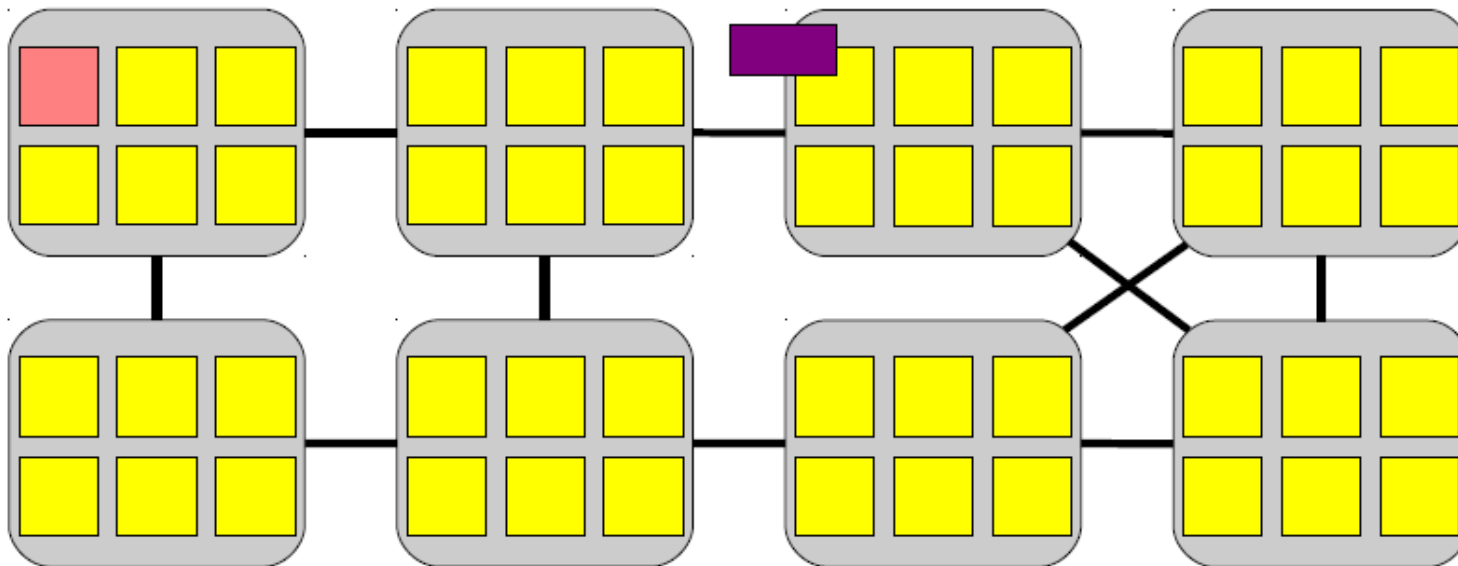
- Read Invalidate
 - The physical address of the cache line to be read, while at the same time directing other caches to remove the data
 - A combination of a “read” and an “invalidate”
 - Requires
 - “read response” and
 - a set of “invalidate acknowledge”
- Writeback
 - The address and the data to be written back to memory
 - And possibly other caches
 - The way to eject lines in the “modified” state as needed to make room for other data

Atomic increment next ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

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



Example

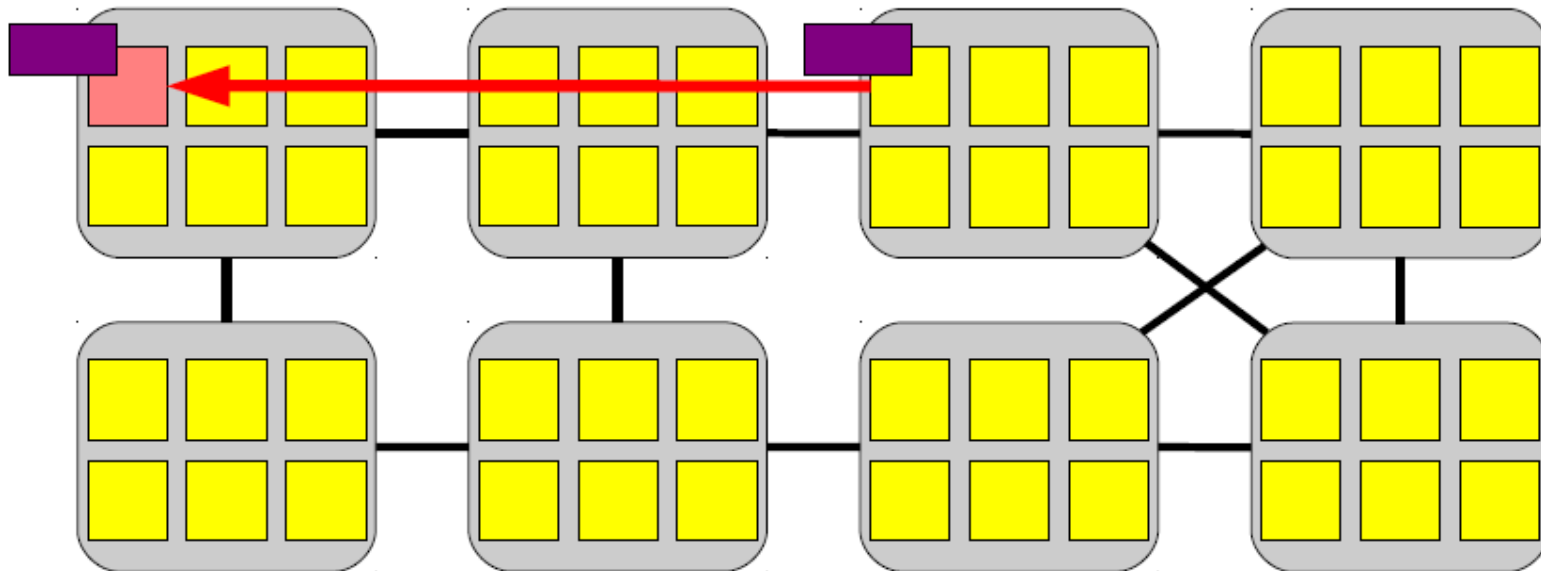
- Invalid → modify
 - An atomic read-modify-write operation on a data item that was not present in its cache
 - Transmits a “read invalidate”, receiving the data via a “read response”
 - The CPU can complete the transition once it has also received a full set of “invalidate acknowledge” responses

Read current ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

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



Example

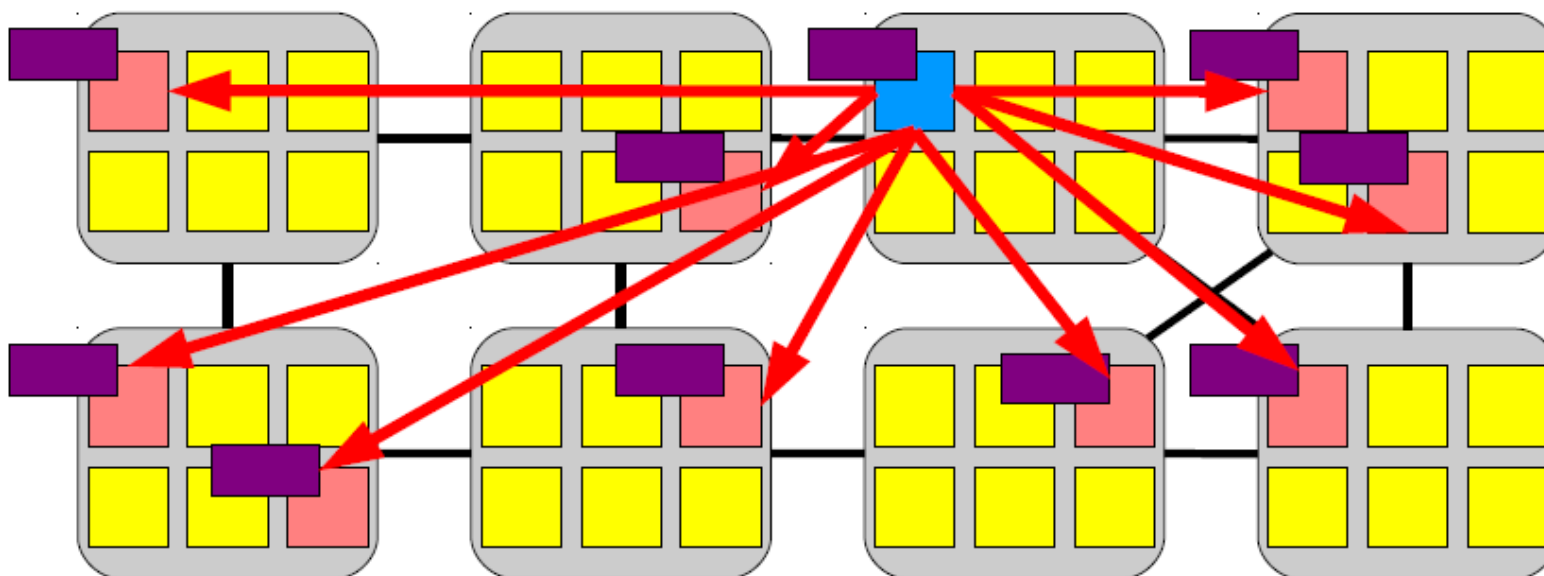
- Invalid → shared
 - Load data that is not in the cache
 - “Read” message, wait for “read response”

Update current ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

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



Example

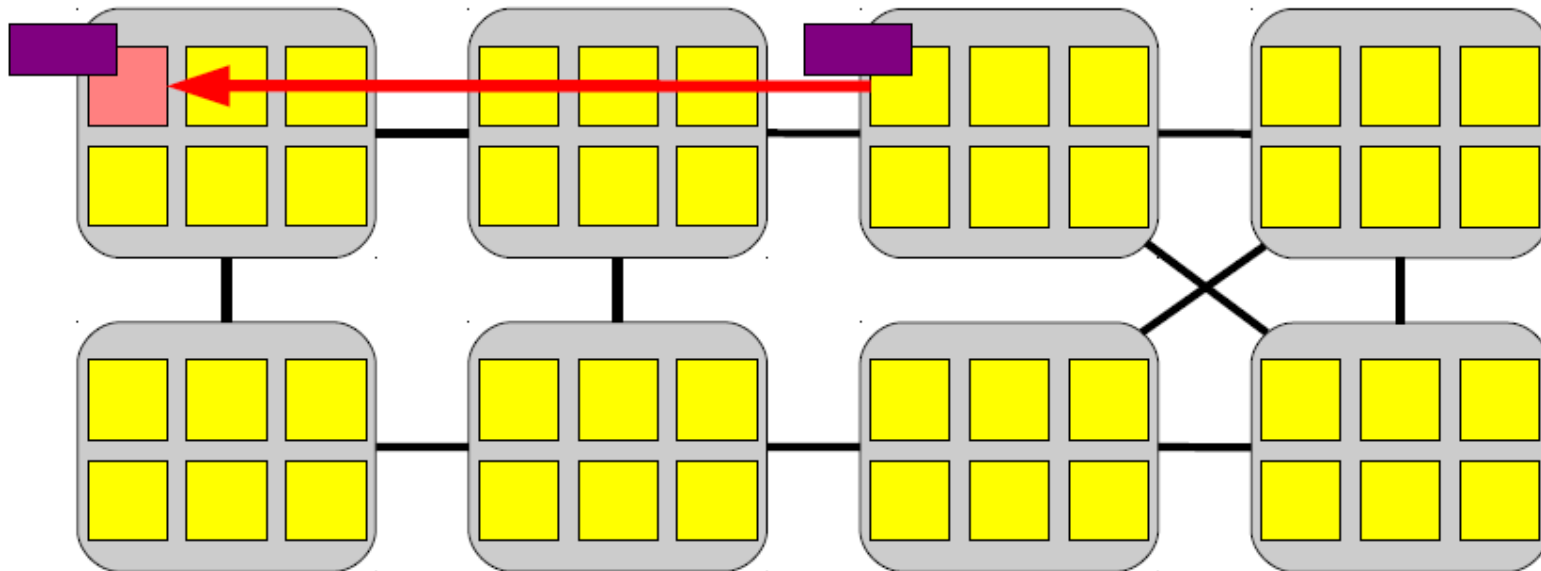
- Shared → exclusive
 - This CPU realizes that it will soon need to write to some data item in this cache line, and thus transmits an “invalidate” message.
 - The CPU cannot complete the transition until it receives a full set of “invalidate acknowledge” responses
- Exclusive → modified
 - The CPU writes to the cache line that it already had exclusive access to.
 - This transition does not require any messages to be sent or received.

Re-read current-ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

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

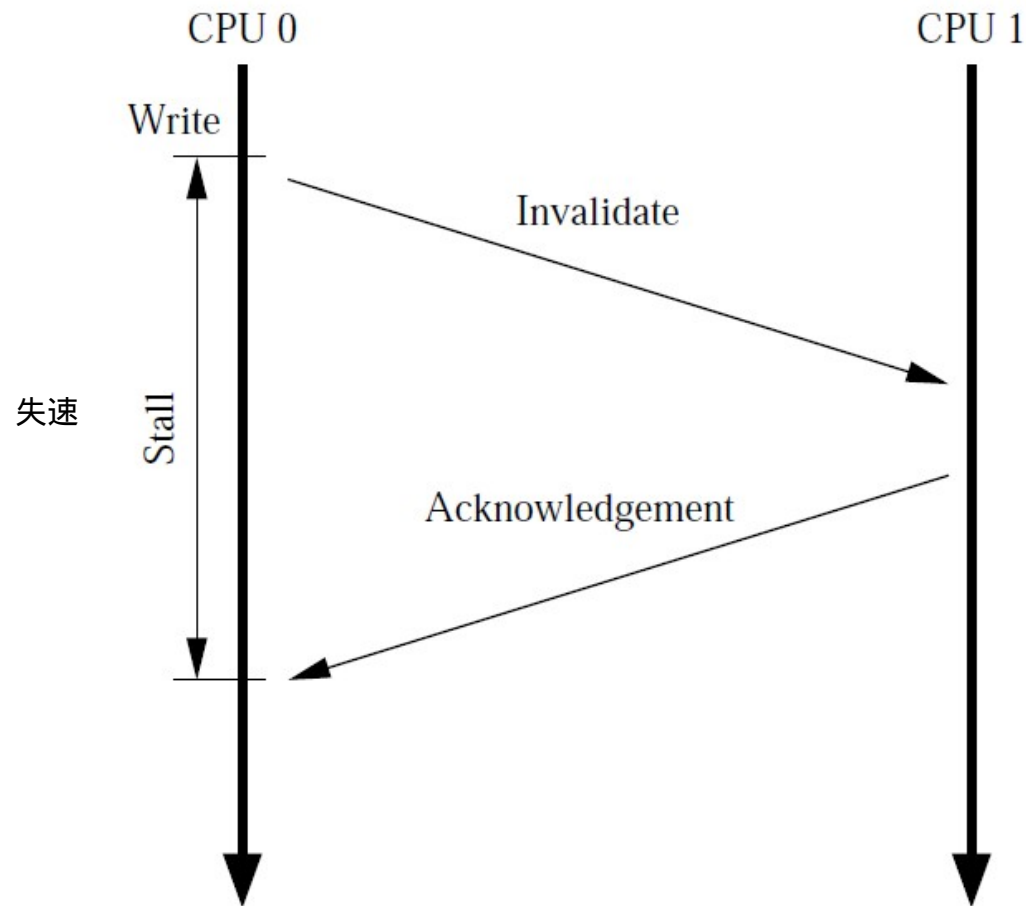


Example

- Modified → shared
 - Some other CPU reads the cache line, and it is supplied from this CPU's cache, which retains a read-only copy, possibly also writing it back to memory.
 - This transition is initiated by the reception of a “read” message, and this CPU responds with a “read response” message containing the requested data.

Memory ordering

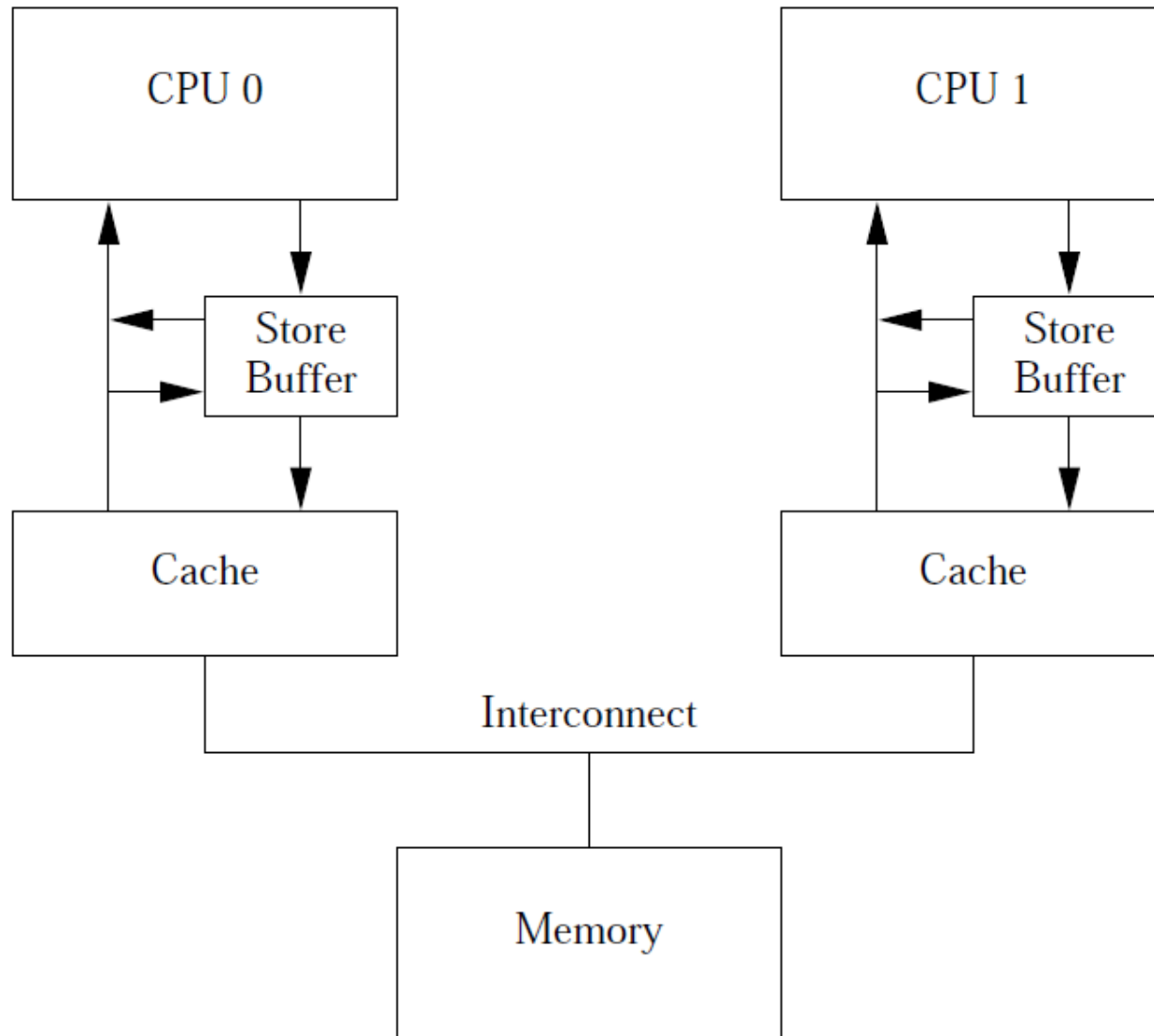
Writes become slow



Store buffers

- Idea:
 - Record a store in a CPU buffer
 - CPU can proceed immediately
- Complete store when invalidate message is received
 - Move a cache line from the store buffer to the cache

Store buffers



Consistency problem

```
1 void foo(void)
2 {
3     a = 1;
4     b = 1;
5 }
6
7 void bar(void)
8 {
9     while (b == 0) continue;
10    assert(a == 1);
11 }
```

- CPU0: foo()
- CPU1: bar()
- Can assert fail?

```
a = [invalid], b = 0 [owned]
a = 1;
// save a in store buffer
// send invalidate(a) message to
// finish write
```

```
b = 1;
// b = [owned], update it in cache
// receive read(b), b → [shared]
// send read_reply(b, 1)
```

```
a = 0 [shared], b = [invalid]

while (b == 0)
// read(b)
```

```
// receive read_reply (b, 1)
assert (a == 1) // fails
// receive invalidate(a)
```

```
a = [invalid], b = 0 [owned]
a = 1;
// save a in store buffer
// send invalidate(a) message to
// finish write

b = 1;
// b = [owned], update it in cache
// DO NOT UPDATE CACHE UNTILL STORE
// BUFFER IS DRAINED
// receive read(b), b → [shared]
// send read_reply(b, 1)
```

```
a = 0 [shared], b = [invalid]

while (b == 0)
// read(b)

// receive read_reply (b, 1)
assert (a == 1) // fails
// receive invalidate(a)
```

Write memory barrier

- Memory barrier `smp_wmb()`
 - *Cause the CPU to flush its store buffer before applying subsequent stores to their cache lines*
 - The CPU could either simply stall until the store buffer was empty before proceeding,
 - Or it could use the store buffer to hold subsequent stores until all of the prior entries in the store buffer had been applied

Consistency fixed

```
1 void foo(void)
2 {
3     a = 1;
4     smp_wmb();
5     b = 1;
6
7 void bar(void)
8 {
9     while (b == 0) continue;
10    assert(a == 1);
11 }
```

```
a = [invalid], b = 0 [owned]
a = 1;
// save a in store buffer
// send invalidate(a) message
smp_wmb()
// mark store buffer

b = 1;
// b = [owned], but there are marked
// entries in the store buffer
// put b = 1 on the store buffer, but
// do not update cache

// receive read(b), b → [shared]
// send read_reply(b, 0)

// receive invalidate(a)
// flush the store buffer
```

```
a = 0 [shared], b = [invalid]

while (b == 0)
// read(b)

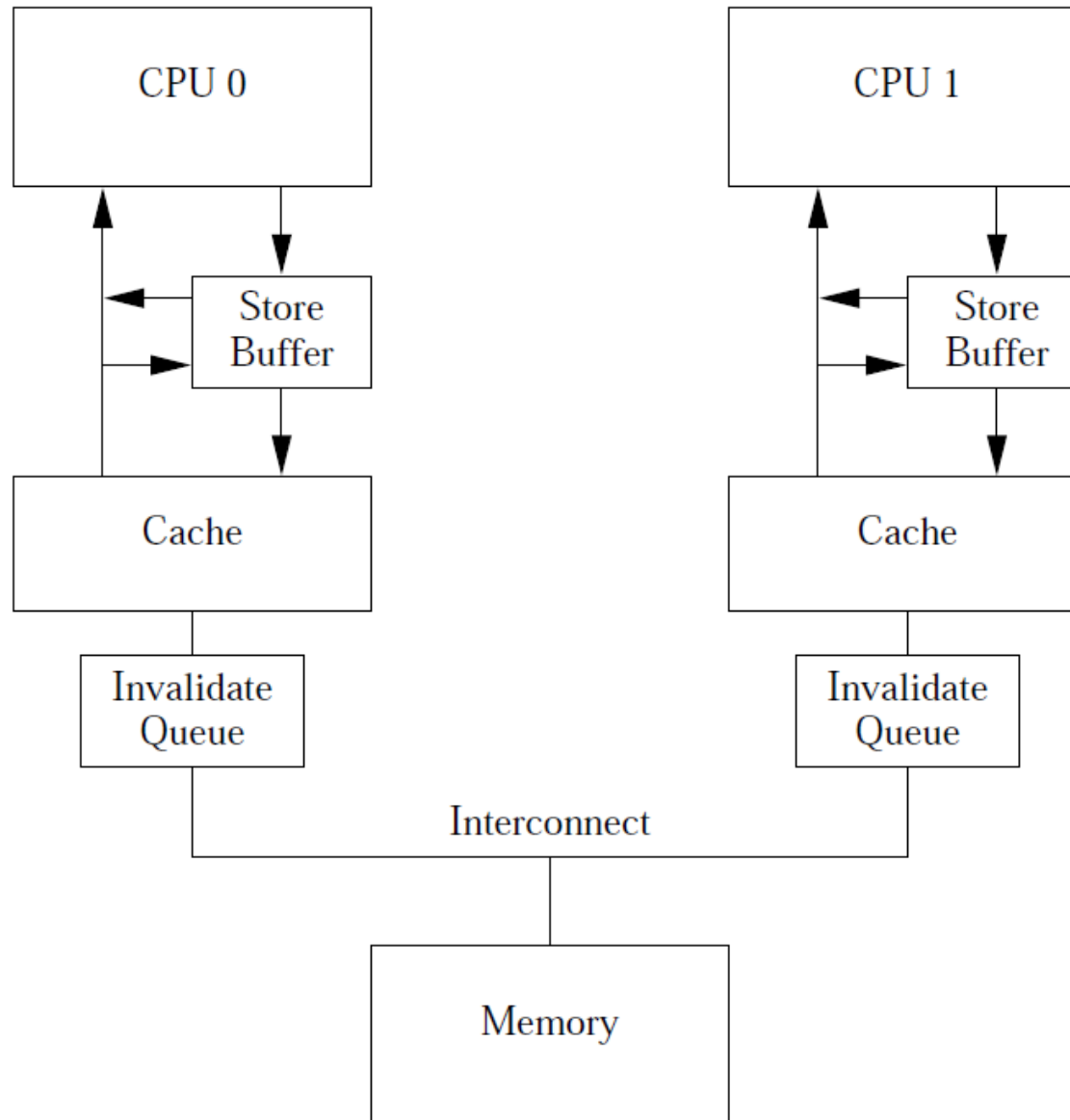
// receive read_reply (b, 0)
// loop

while (b == 0)
// b is [invalid], read(b) again
assert (a == 1) // succeed
// receive invalidate(a)
```

Invalidate queues

- Invalidate messages can be slow
 - Caches can be overloaded
- While waiting for invalidate acknowledgements
 - Run out of space in the store buffer
- Idea: Why wait for cache?
 - Store invalidate request in a queue
 - Acknowledge it right away
 - Apply later

Invalidate queues



```
a = [invalid], b = 0 [owned]
a = 1;
// save a in store buffer
// send invalidate(a) message
smp_wmb()
// mark store buffer

// receive invalidate(a)
// flush the store buffer

b = 1;
// b = [owned], update cache

// receive read(b), b → [shared]
// send read_reply(b, 1)
```

```
a = 0 [shared], b = [invalid]

while (b == 0)
// read(b)
// receive invalidate(a), queue it and
// reply right away

// receive read_reply (b, 1)
assert (a == 1) // fail
// receive invalidate(a)
```

```
a = [invalid], b = 0 [owned]
a = 1;
// save a in store buffer
// send invalidate(a) message
smp_wmb()
// mark store buffer

// receive invalidate(a)
// flush the store buffer

b = 1;
// b = [owned], update cache

// receive read(b), b → [shared]
// send read_reply(b, 1)
```

```
a = 0 [shared], b = [invalid]

while (b == 0)
// read(b)
// receive invalidate(a), queue it and
// reply right away

// receive read_reply (b, 1)
// MAKE SURE INVALIDATE QUEUE IS DRAINED
assert (a == 1) // fail
// receive invalidate(a)
```

Read memory barrier

- Read barrier `smp_rmb()`
 - *Marks all the entries currently in its invalidate queue, and forces any subsequent load to wait until all marked entries have been applied to the CPU's cache.*

Consistency fixed

```
1 void foo(void)
2 {
3     a = 1;
4     smp_wmb();
5     b = 1;
6
7 void bar(void)
8 {
9     while (b == 0) continue;
10    smp_rmb();
11    assert(a == 1);
12 }
```

Trying to execute

```
while (b == 0)
```

- CPU 1 sends read (b) message, receives the cache line containing “b” and installs it in its cache
- CPU 1 can now finish executing while(b==0) continue, and since it finds that the value of “b” is 1, it proceeds to the next statement, which is now a memory barrier
- CPU 1 must now stall until it processes all preexisting messages in its invalidation queue
- CPU 1 now processes the queued “invalidate” message, and invalidates the cache line containing “a” from its own cache
- CPU 1 executes the assert(a==1), and, since the cache line containing “a” is no longer in CPU 1’s cache, it transmits a “read” message
- CPU 0 responds to this “read” message with the cache line containing the new value of “a”
- CPU 1 receives this cache line, which contains a value of 1 for “a”, so that the assertion does not trigger

Conclusion

- Memory barriers are required to ensure correct order of cross-CPU memory updates
 - E.g. update two memory locations a, and b
- Two memory barriers are common
 - Write
 - Read

Thank you!