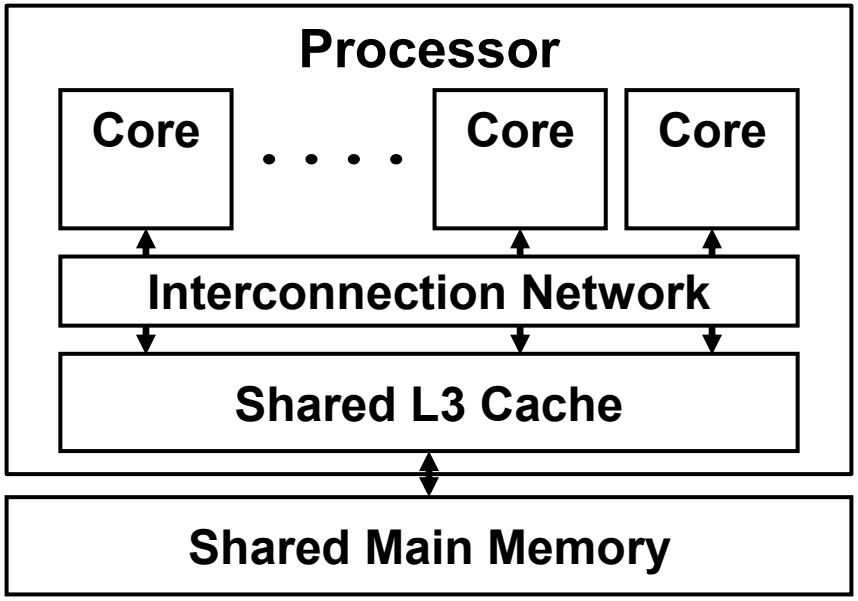
Lecture 21: Multicore – Consistency

Hunjun Lee
hunjunlee@hanyang.ac.kr

Memory Ordering Problem

- One of the major problems in multicore structure is about the order of memory operations!

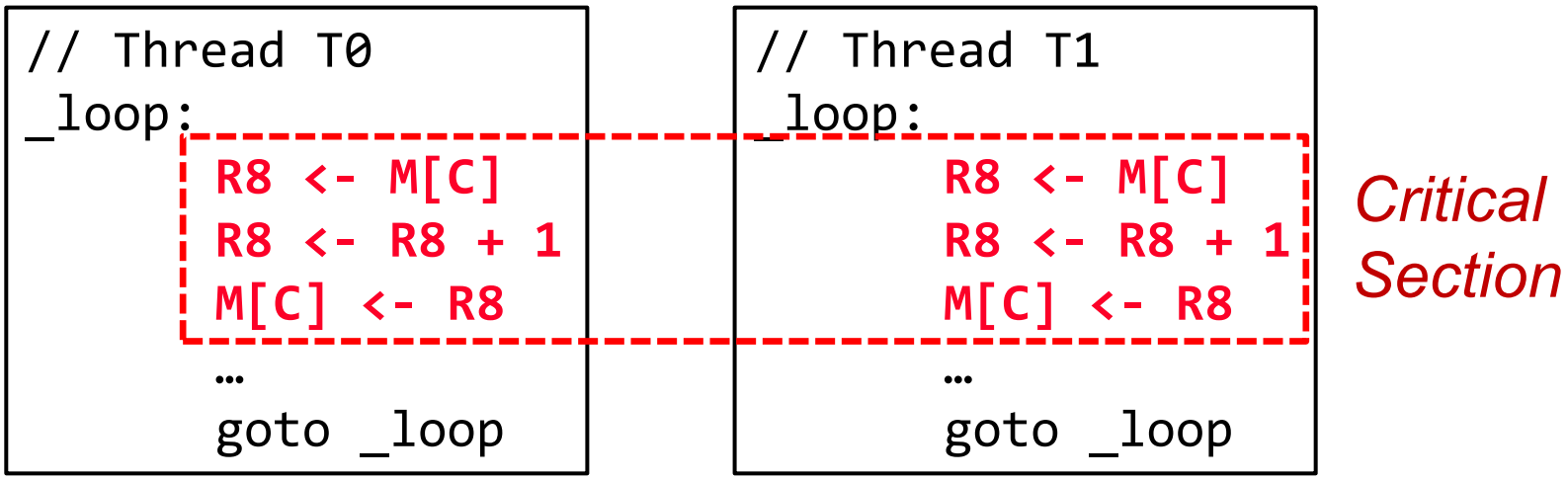
- The computation is distributed across cores: can we assume Von Neumann Architecture?



Core 여러개, Memory 하나
→ Memory 영역에서는 하나의 데이터가 여러 core가 언제 실행하여 데이터는 언제인지가 명확하지 않다
→ Memory Sharing

Protecting Shared Data

- The major correctness problem is related to the shared memory accesses!
- How to ensure functional correctness? We need to protect the data using synchronization or critical sections!
  - **Mutual Exclusion:** Make only a single thread execute a critical section at a time
- This type of synchronization should be supported! ➔ To enforce a specific order!



What do we need? Lock!

- The programmer should have a mechanism (instruction) to ensure mutual exclusion! ➔ For a functional correctness
- A representative example: Lock
  - Acquire(L)
    - If L = 0, advance to the next line and set L to 1 // LOCK
    - If L = 1, stop and wait until L becomes 0 // WAIT
  - Release(L)
    - Set L to 0 // UNLOCK

그럼 Acquire, Release는 어떻게 구현하는가?

Programmer가 → Lock을 잘못한 경우

```
// Thread T0
_loop:
        Acquire(L)
        R8 <- M[C]
        R8 <- R8 + 1
        M[C] <- R8
        Release(L)
        …
        goto _loop
```

```
// Thread T1
_loop:
        Acquire(L)
        R8 <- M[C]
        R8 <- R8 + 1
        M[C] <- R8
        Release(L)
        …
        goto _loop
```

Critical Section

Spin Lock (Not an Answer)

Acdure, Release를 구현한 가장 쉬운 방법

- A thread modifies the lock value, and the other thread spins until the lock is released

- The following implementation may lead to both threads executing the critical section simultaneously

```
// T0
Acquire:
        R1 <- M[L]
        if (R1 == 1) goto Acquire
        M[L] <- 1

Release:
        M[L] <- 0
```

```
// T1
Acquire:
        R1 <- M[L]
        if (R1 == 1) goto Acquire
        M[L] <- 1

Release:
        M[L] <- 0
```

Concurrent Execution

What we really need? RMW instructions!

- How to implement a correct Acquire(L)?
- **A special class of memory instructions (=atomic instruction)** is needed for synchronization (between concurrent or time-multiplexed threads)

- Atomic RMW instruction (e.g., Test and Set)
  - Read from a location
  - Perform a simple computation
  - Write something back to the same memory location
- Obviously, such instructions (=atomic instructions) are very expensive to implement and to perform in HW

Representative Atomic Operations

- There are four (actually more …) representative atomic RMW operations
- There can be no exception or interrupts in between an atomic operation

```
Test&Set(addr, r):
    r <- M[addr];
    M[addr] <- 1;
```

```
Swap(addr, r):
    temp <- M[addr];
    M[addr] <- r;
    r <- temp;
```

```
Fetch&Add(addr, r):
    r <- M[addr];
    M[addr] <- r+1;
```

```
Compare&Swap(addr, r1, r2):
    if (r1 == M[addr])
        Swap(M[addr], r2)
```

Implementing a Lock

- Using **Test & Set**, L initially 0

```
Acquire(L):
  do {
      Test&Set(L, r_temp)
  } while (r_temp !=0)
```

```
Release(L):
  M[L] ← 0
```

- Using **Swap**, L initially 0

```
Acquire(L):
  do {
      r_temp ← 1
      Swap(L, r_temp)
  } while (r_temp !=0)
```

```
Release(L):
  M[L] ← 0
```

Implementing a Lock

- Using **Fetch&Add**, L initially 0

```
Acquire(L):
  do {
      Fetch&Add(L, r_temp)
  } while (r_temp !=0 )
```

```
Release(L):
  M[L] ← 0
```

- Using **Compare&Swap**, L initially 0

```
Acquire(L):
  do {
      r_comp ← 0; r_swap ← 1
      Compare&Swap(L, r_comp, r_swap)
  } while (r_swap !=0 )
```

```
Release(L):
  M[L] ← 0
```

Advanced Lock - 1

- What about Test & Test & Set
  - Perform atomic operations less!

```
Acquire(L):
  while (true) {
      r_original <- M[L]
      if (r_original == 0) {
          Test&Set(L, r_temp)
          if (r_temp == 0) break
      }
  }

Release(L):
  M[L] ← 0
```

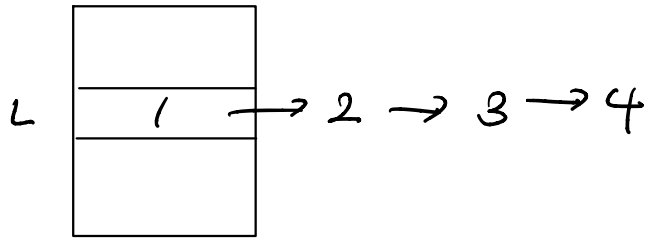
Advanced Lock - 2

- You can implement your own lock for various purposes

```
next_ticket = 0
serving = 0

Acquire:
  my_ticket = fetch_and_add(&next_ticket, 1)
  while (serving != my_ticket) {}

Release:
  serving++
```



Specialized Load & Store Support

- Atomic RMW can be “mimicked” by a pair of load and store instructions with “extra” semantics
  - This is highly effective for RISC
  - Each core has < flag, addr> storage, and store-cond. status bit

```
(LL) load-locked (r, addr):    // a.k.a  load-linked
        r <- M[addr]
        <flag, address> <- <1, addr>
```

-- HW Invalidates flag to 0 if other processors update the addr --

```
(SC) store-conditional (r, addr):
        if <flag, address> = <1, addr> then
                M[addr] <- r
                set status to succeed // store done
        else
                set status to fail    // store cancelled
```

Atomic Ops Using LL/SC

- Basic strategy when mimicking an atomic RWM operation ➔ keep retrying until you know you have succeeded without being interrupted)

```
Swap(addr,r):
loop:  load-locked(r_temp, addr)
       store-conditional(r,addr)
       if status=failed then go to loop
       r=r_temp
```

```
Fetch&Add(m,v):
loop:  load-locked(r_temp, addr)
       r_temp= r_temp+ 1
       store-conditional(r_temp, addr)
       if status=failed then go to loop
```

Case: Pthread Synchronization

```
#include <pthread.h>
pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;

void *do_threading(void *ptr) {
        printf("%s is live!\n", (char*) ptr);
        ……

        pthread_mutex_lock(&my_mutex);

        // CRITICAL SECTION
        (e.g., updating a shared variable)

        pthread_mutex_unlock(&my_mutex);
        ……

        pthread_exit()
}
```

Consistency!

Memory Ordering Problem

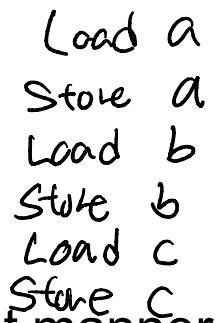
- Let's say there are four memory operations: A, B, C, D
  - How does this operate in a single-core architecture? ➔ Of course, **it should execute in-order (A → B → C → D)**
  - This is defined in the ISA!

- Does this assumption hold in multicore architecture?
  - Yes: the programmer's life would be easy!
  - No: the multicore processing would be faster!

- We need a new ISA that defines the memory ordering in the multicore architecture
  - It may be different from that of the single-core architecture
  - But, it should be **"consistently"** applied in various cases!

Sequential Ordering

- The memory ordering is specified by **von Neumann model**
- The hardware executes the load and store operations in **sequential order**
  - Remember? There are out-of-order execution, but the operation commits in a sequential order! (ROB)



- Advantage: extremely straightforward
  - The architectural state is precise!
  - The architectural state changes in a consistent manner (across program runs) ➔ Easy for debugging
- Disadvantage: hardware complexity
  - There are various mechanisms to support sequential order … (remember?)

Memory Ordering in Multicore?

- **Processors' perspective:** each processor loads and stores data in a sequential order
- **Memory's perspective:** OMG … there are multiple processors … what should be the order of memory operations from different cores?
- But Why?? (Can't we just make is arbitrary?)
  - Debugging purpose (repeatability)
  - Each processor should see the same global order (correctness)
    - ➔ When the two cores share a data!
    - **Core A:** `L 0x108 -> S 0x108 -> L 0x108`
    - **Core B:** `S 0x108`

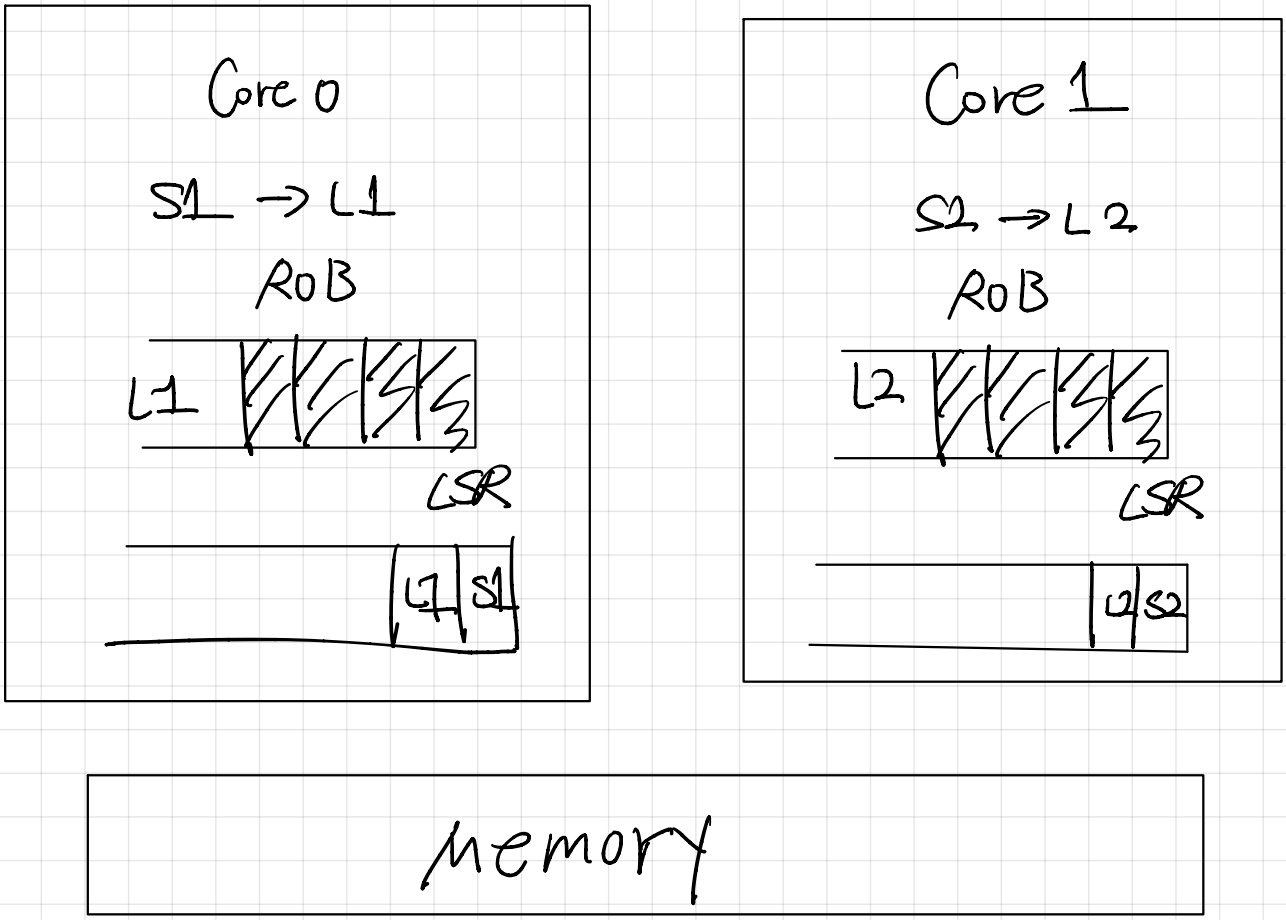
Ensuring Correct Execution Order

- You can use two variables to enable locks! ➔ Dekker's algorithm!
  - Dijkstra, "Cooperating Sequential Processes" **(1965)**

```
// Thread A
// M[f1] = 0 and M[f2] = 0
_loop:
S1:     M[f1] <- SET
L1:     r1 <- M[f2]
        if (r1 == 0)
                <Critical Section>
        …
```

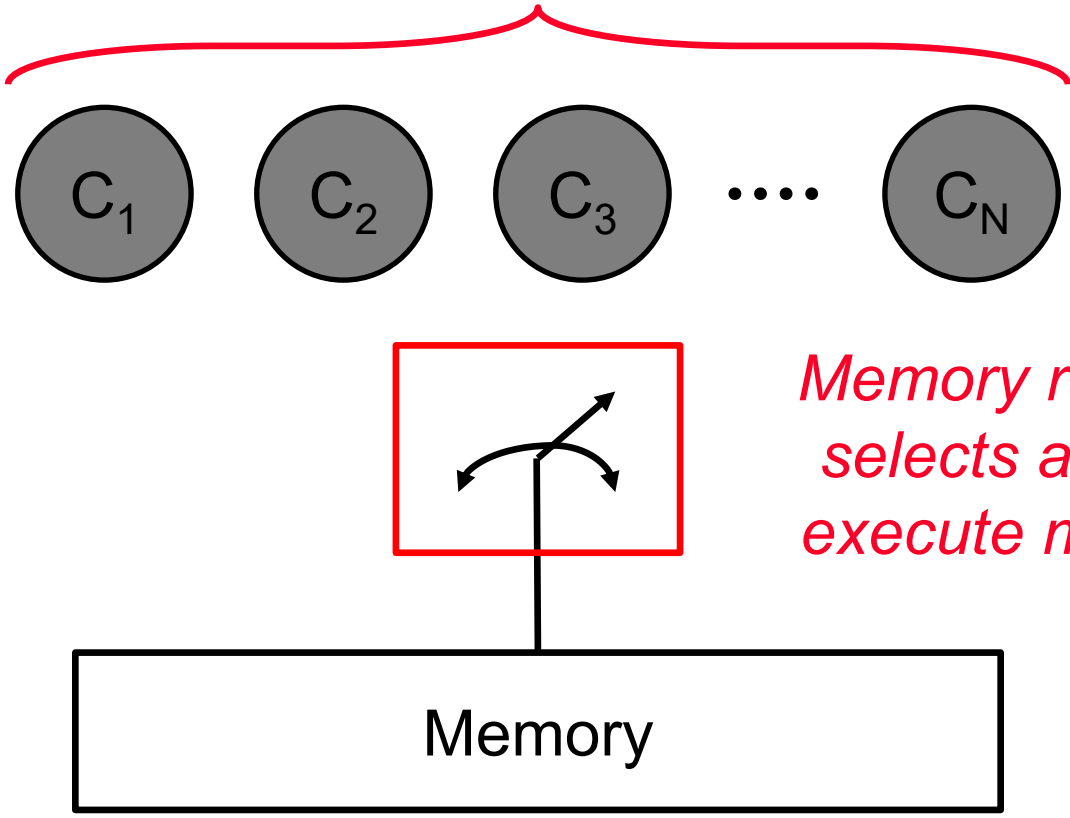
```
// Thread B
// M[f1] = 0 and M[f2] = 0
_loop:
S2:     M[f2] <- SET
L2:     r2 <- M[f1]
        if (r2 == 0)
                <Critical Section>
        …
```

- There can be multiple possibilities
  - S1 -> L1 -> S2 -> L2 ➔ (r1, r2) = (0, SET)
  - S2 -> L2 -> S1 -> L1 ➔ (r1, r2) = (SET, 0)
  - S1 -> S2 -> L1 -> L2 ➔ (r1, r2) = (SET, SET)
  - What about? **S1 -> L2 -> L1 -> S2 ➔ (r1, r2) = (0, 0)**


Core 0
S1 → L1
ROB
L1
LSR
L1
S1
Core 1
S2 → L2
ROB
L2
LSR
L2
S2
memory

Reason Behind This … Sequential Consistency (SC)

Each core issues request in program order

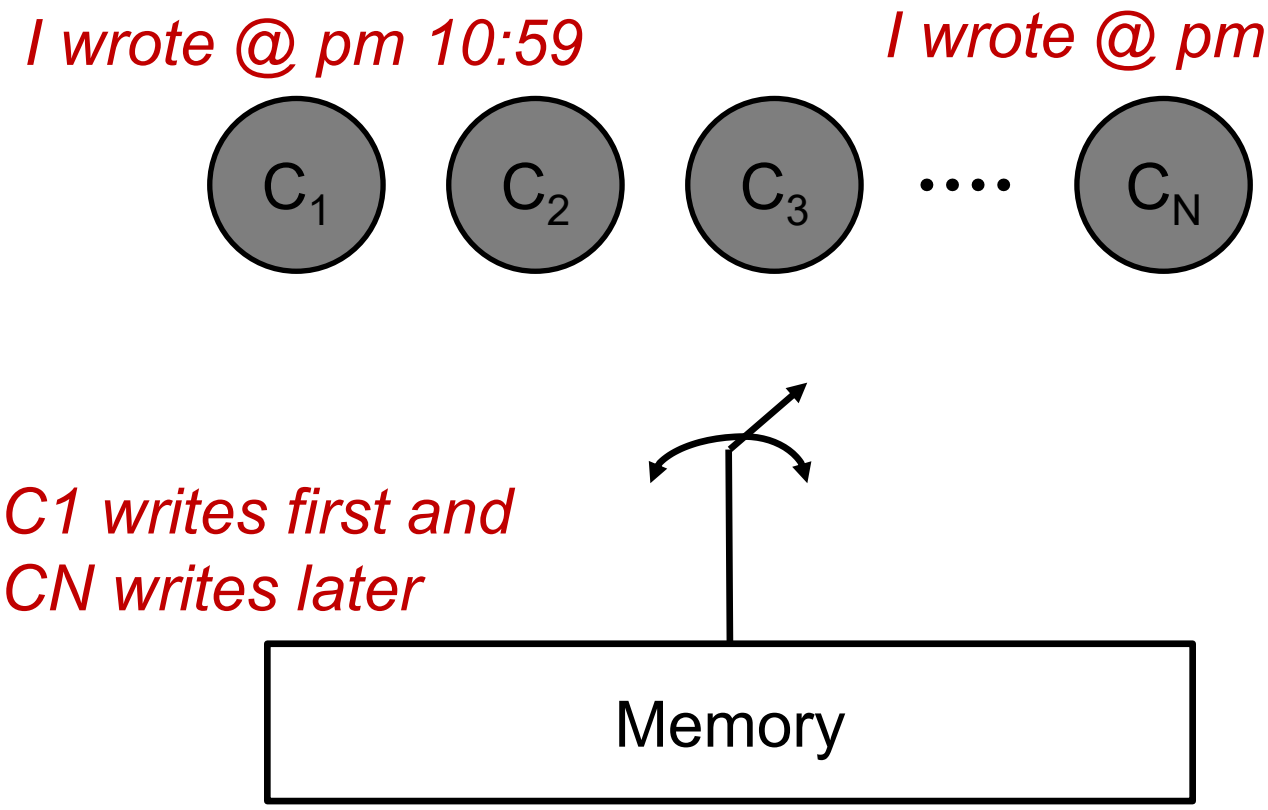


Memory randomly selects a core to execute mem ops

There is a single global ordering determined by the memory

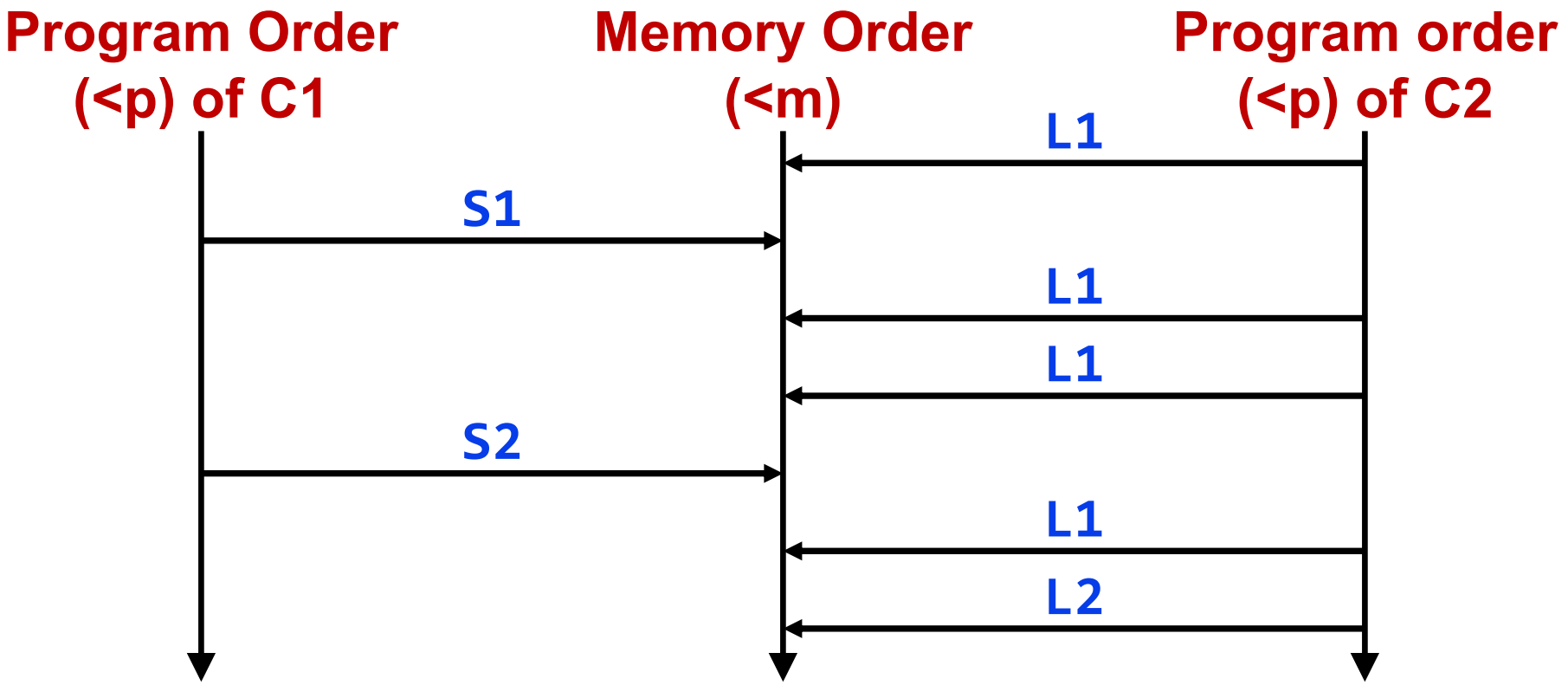
This is about “relative timing”, not the exact timing

- The global ordering of S1 -> S2 -> L1 -> L2 does not necessarily mean that, the memory operations has been issued at the given timing order



Main Assumptions Behind SC

- **Program order:** order of memory operations per core
- **Memory order:** the global ordering of all memory operations from all cores

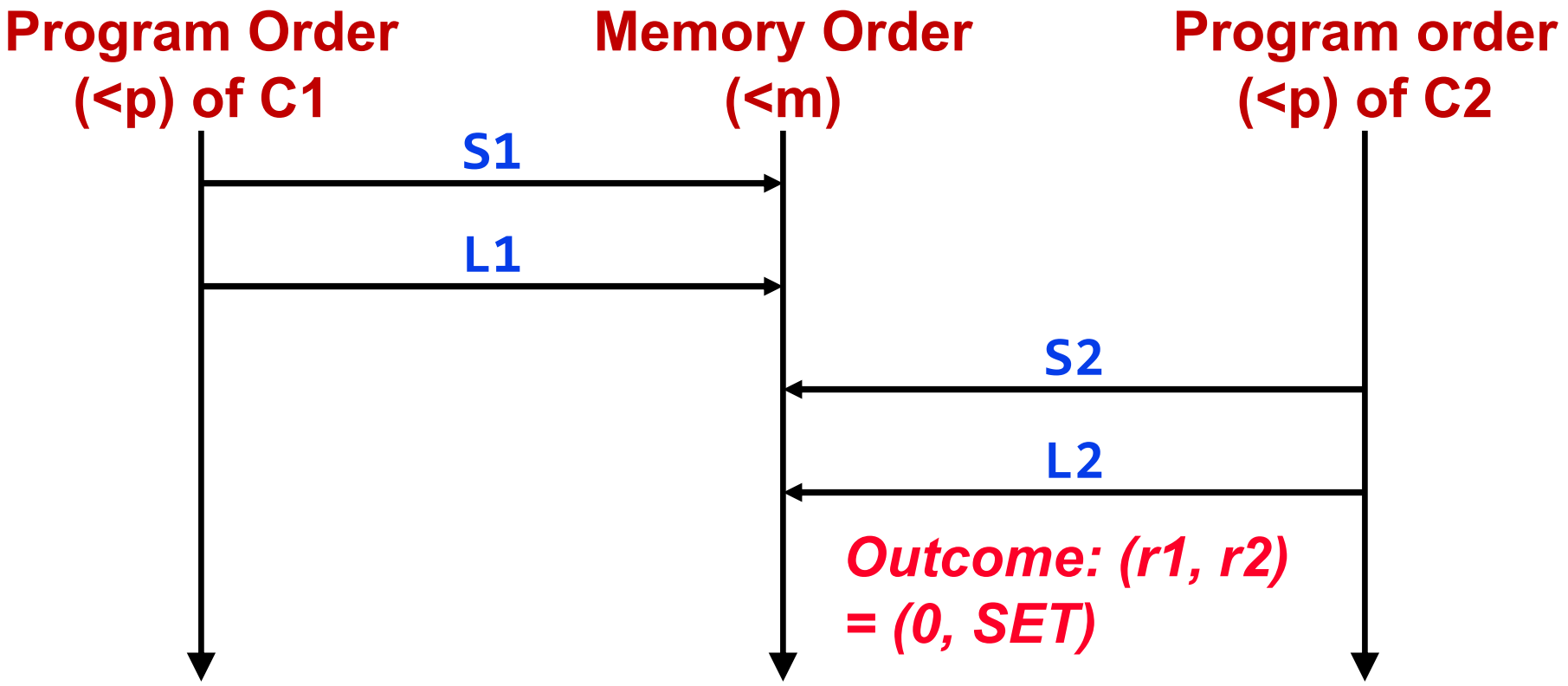


In SC, memory order respects each core's program order

Sequential Consistency

```
// Thread A
// M[f1] = 0 and M[f2] = 0
_loop:
S1:     M[f1] <- SET
L1:     r1 <- M[f2]
```

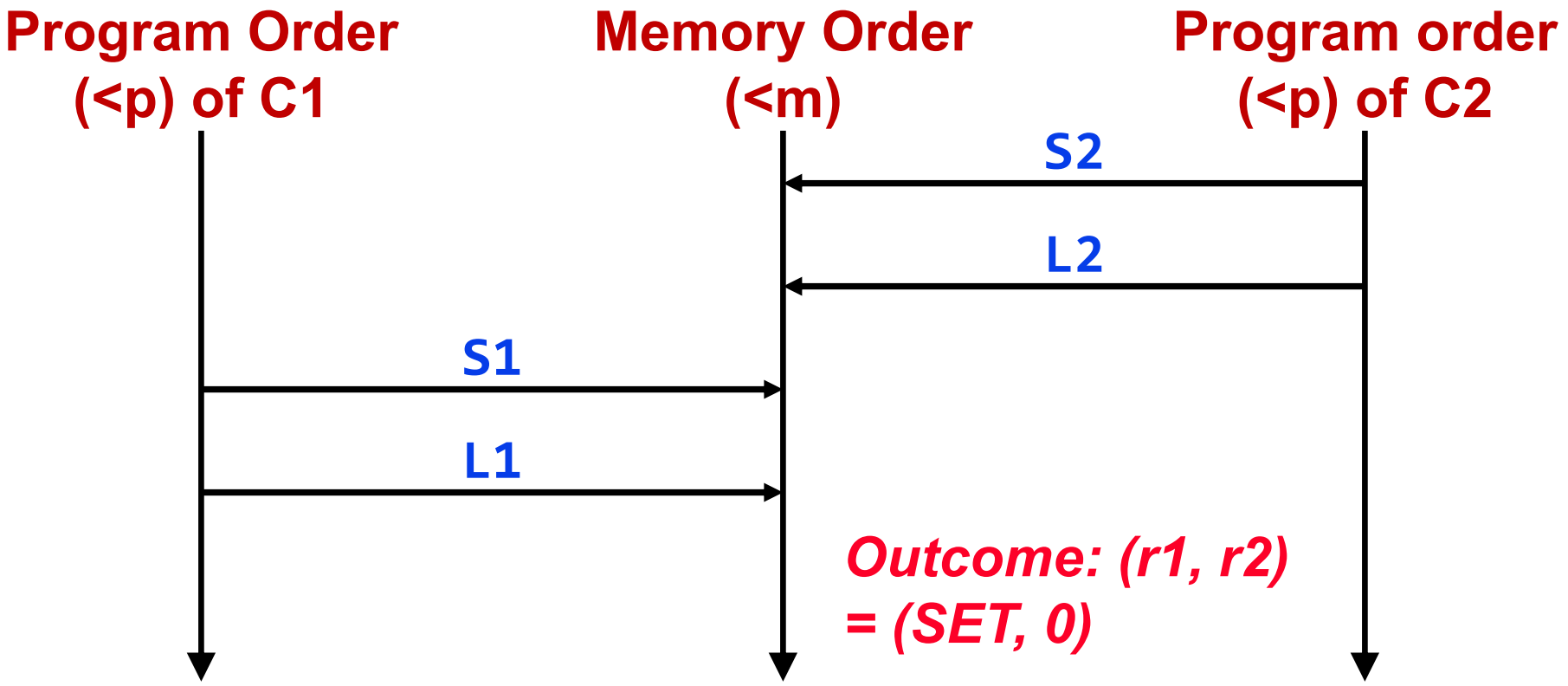
```
// Thread B
// M[f1] = 0 and M[f2] = 0
_loop:
S2:     M[f2] <- SET
L2:     r2 <- M[f1]
```



Sequential Consistency

```
// Thread A
// M[f1] = 0 and M[f2] = 0
_loop:
S1:     M[f1] <- SET
L1:     r1 <- M[f2]
```

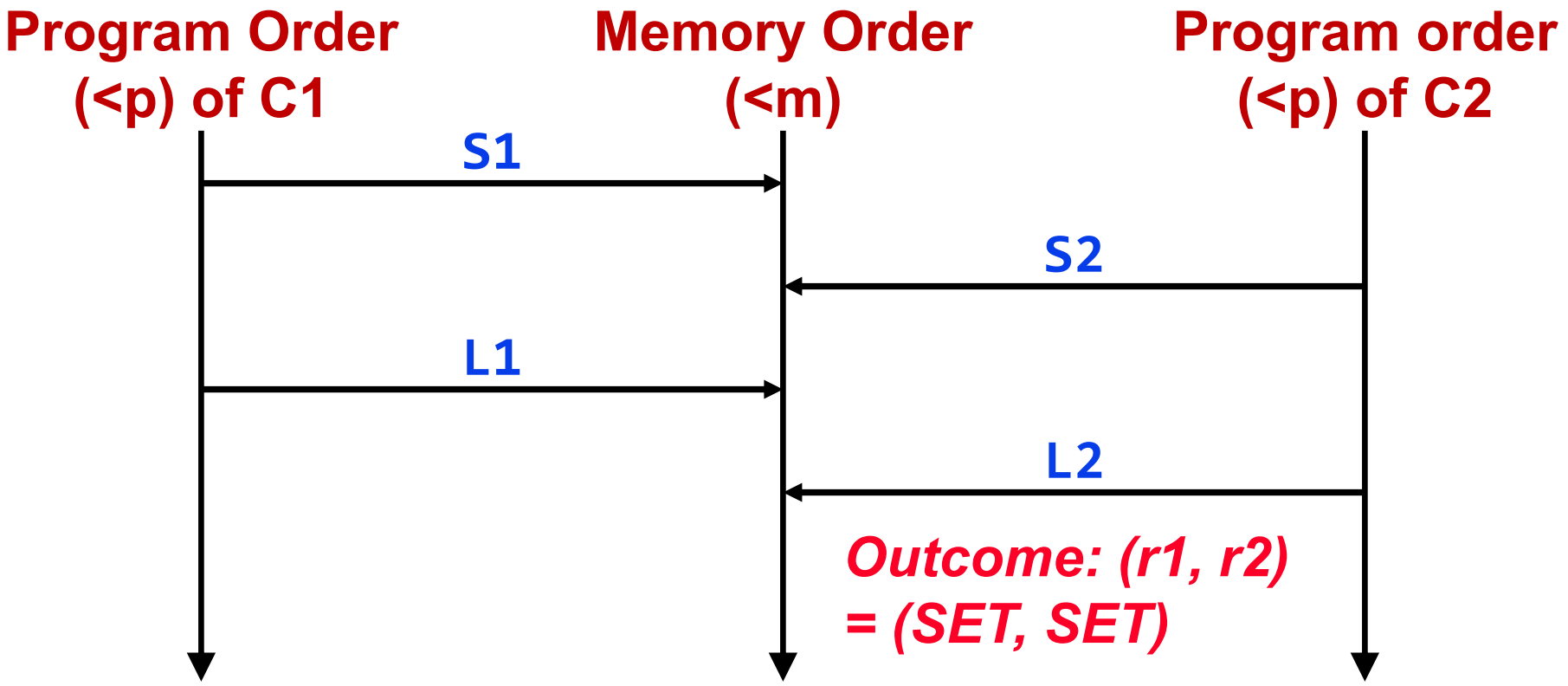
```
// Thread B
// M[f1] = 0 and M[f2] = 0
_loop:
S2:     M[f2] <- SET
L2:     r2 <- M[f1]
```



Sequential Consistency

```
// Thread A
// M[f1] = 0 and M[f2] = 0
_loop:
S1:     M[f1] <- SET
L1:     r1 <- M[f2]
```

```
// Thread B
// M[f1] = 0 and M[f2] = 0
_loop:
S2:     M[f2] <- SET
L2:     r2 <- M[f1]
```



Problem

- r3 value at Core3 in sequential consistency?

```
// Initially M[x] = M[y] = 0
```

```
// Core 1
S1:     M[x] <- 1
```

```
// Core 2
S2:     r1 <- M[x]
        if (r1 == 1)
S3:             M[y] <- 1
```

```
// Core 3
S2:     r2 <- M[y]
        if (r2 == 1)
L3:             r3 <- M[x]
```

- Can r3 become 0?

Formulate SC Ordering

- SC Execution requires the following

```
// a and b may or may not be the same
If L(a) <p L(b) ➔ L(a) <m L(b) // Load -> Load
If L(a) <p S(b) ➔ L(a) <m S(b) // Load -> Store
If S(a) <p S(b) ➔ S(a) <m S(b) // Store -> Store
If S(a) <p L(b) ➔ S(a) <m L(b) // Store -> Load
```

  The program order is **completely reflected** in the memory order

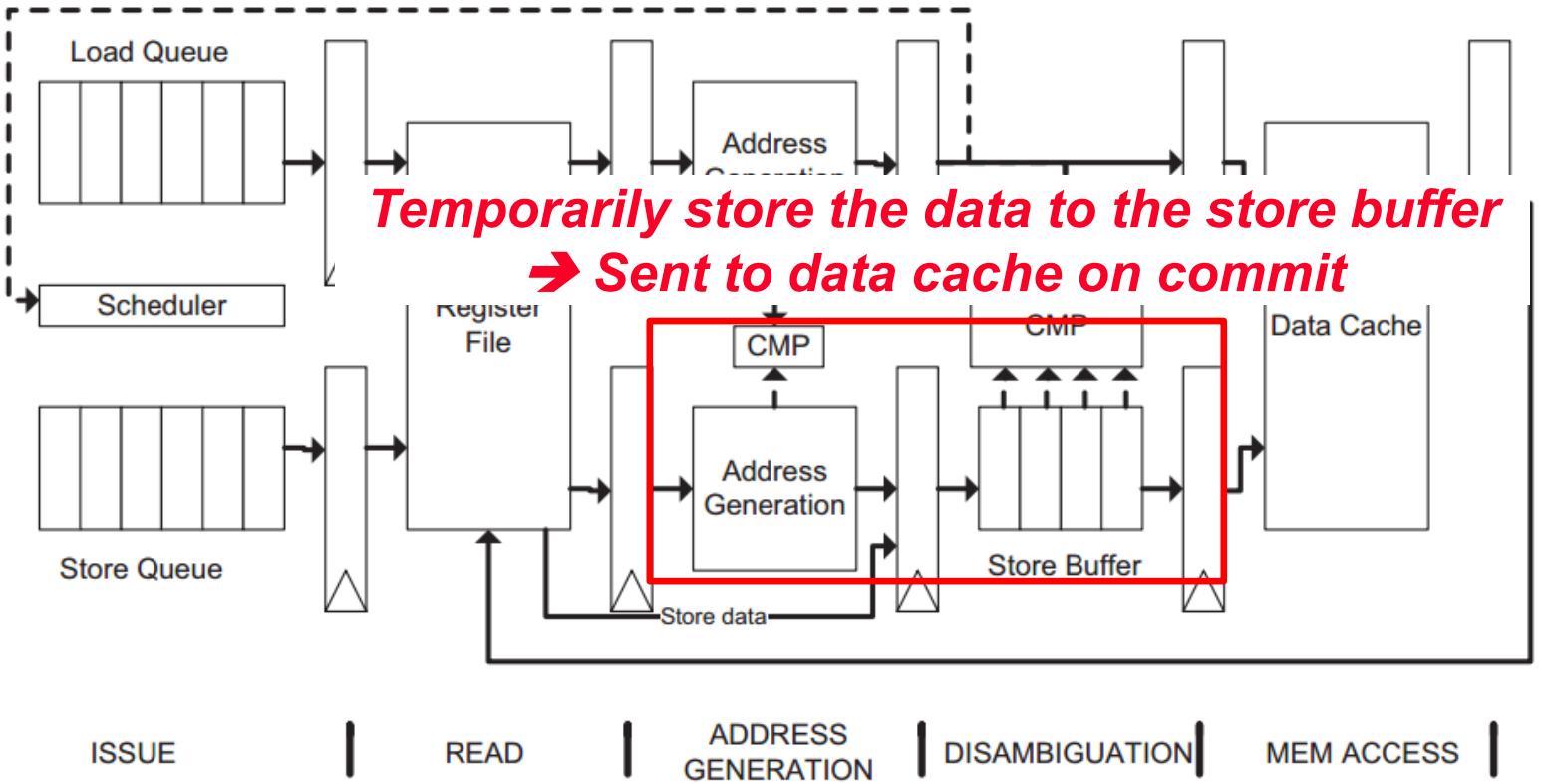
- So,,, every load gets its value from the last store before it (in global order) to the same address:

```
L(a) = Value of MAX <m {S(a) | S(a) <m L(a)}
```

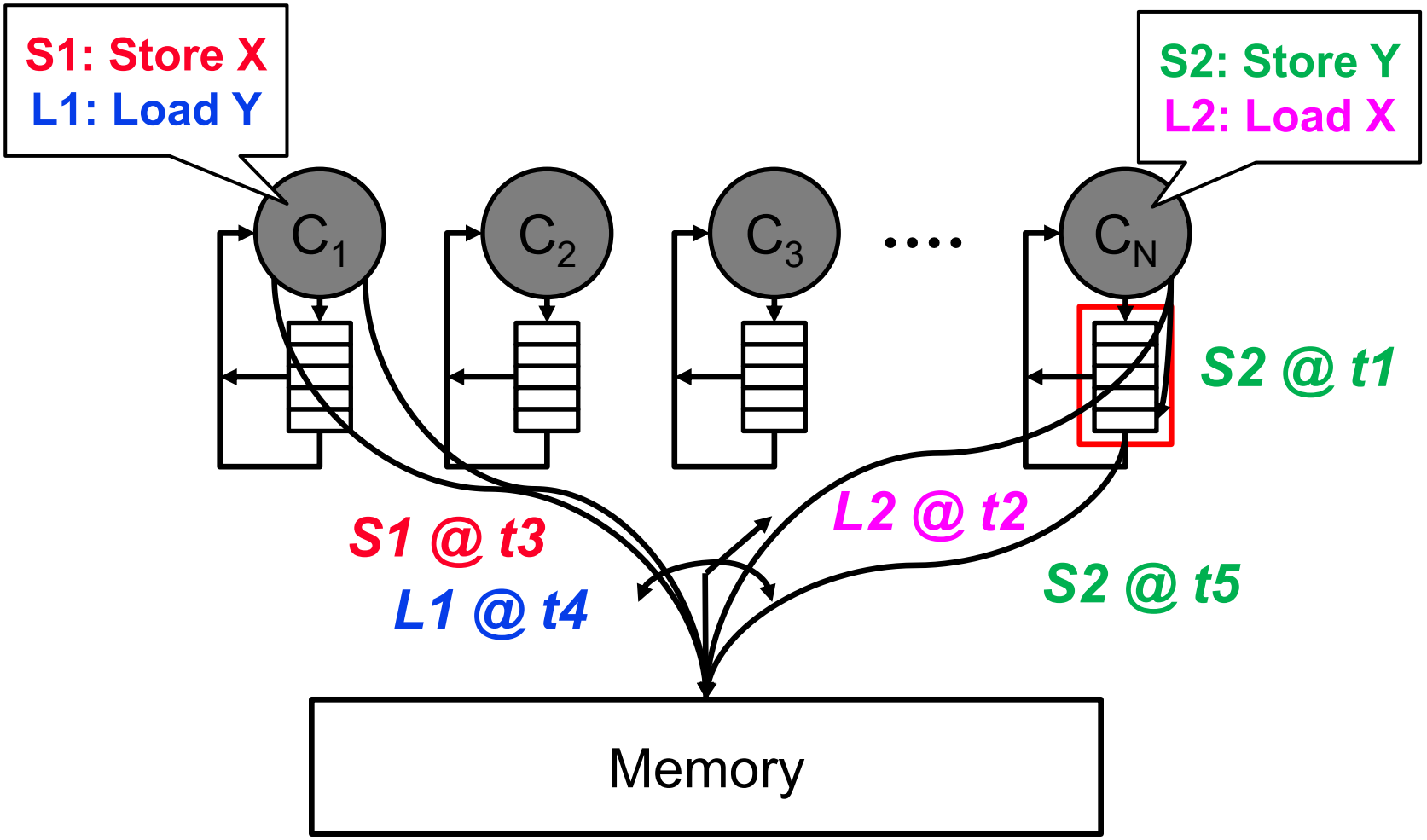
Is SC Ideal?

- Difficult to implement efficiently in hardware **(overly conservative …)**
  - To realize an overly conservative model:
    - No concurrent memory accesses
    - Strict ordering of memory access at each node
    - Essentially precludes out-of-order CPUs
- If we do that, **unnecessarily restrictive**
  - Most parallel programs won't notice out-of-order accesses (may not matter at all …)
- That would conflict with various latency-hiding techniques

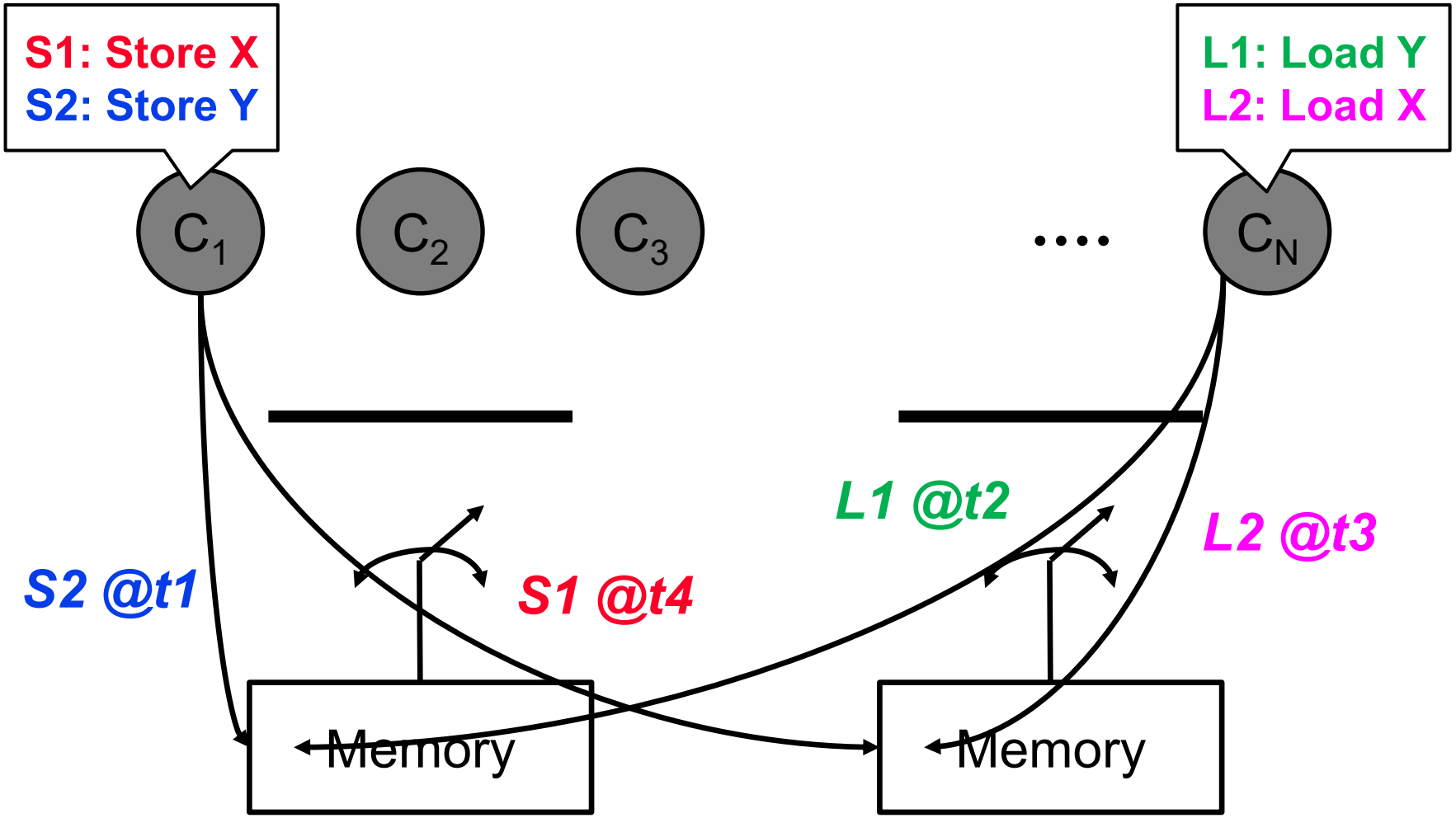
In Reality …



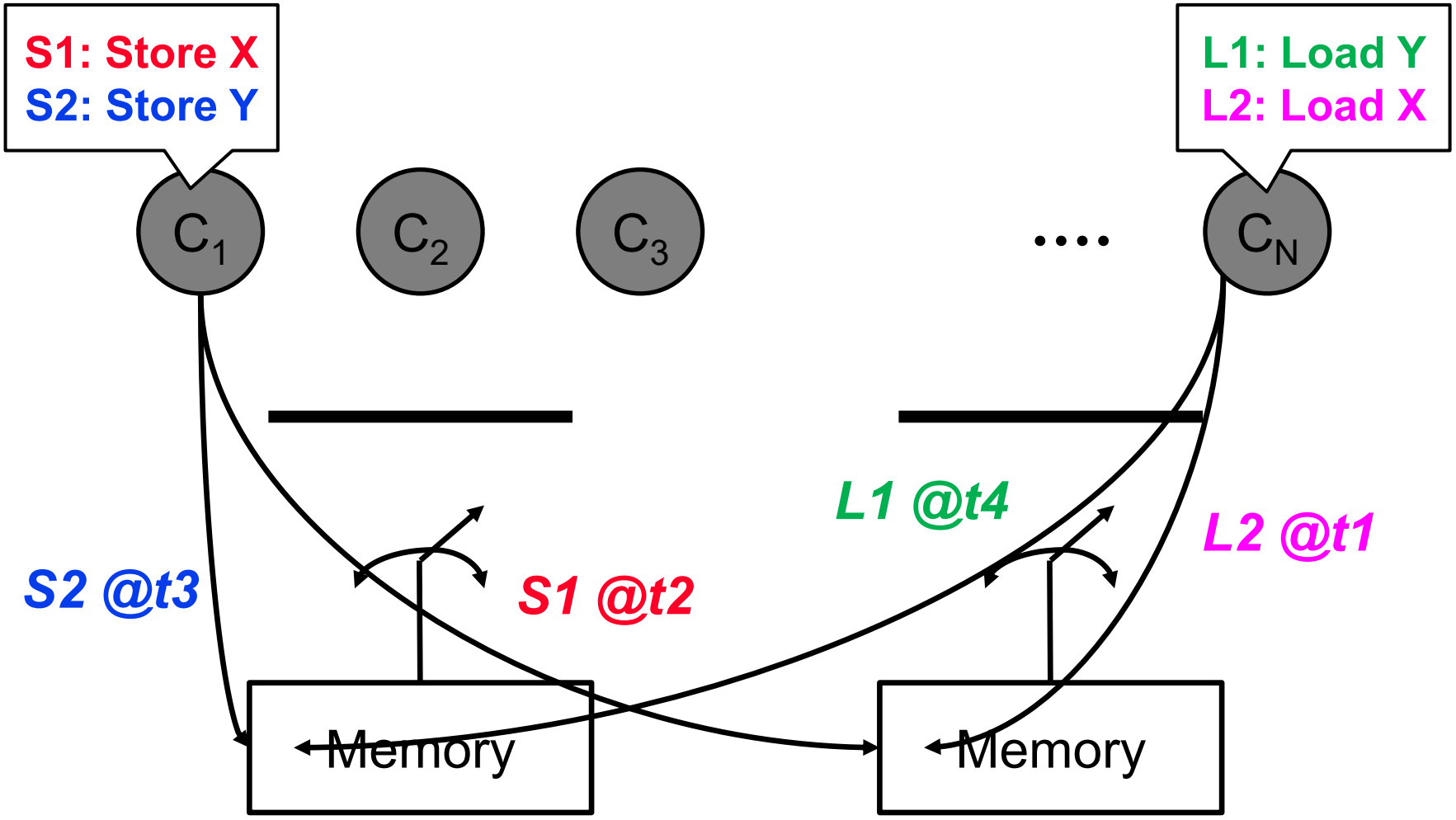
When used in a multiprocessor context



What about this …?
Overlapping writes @ NUMA

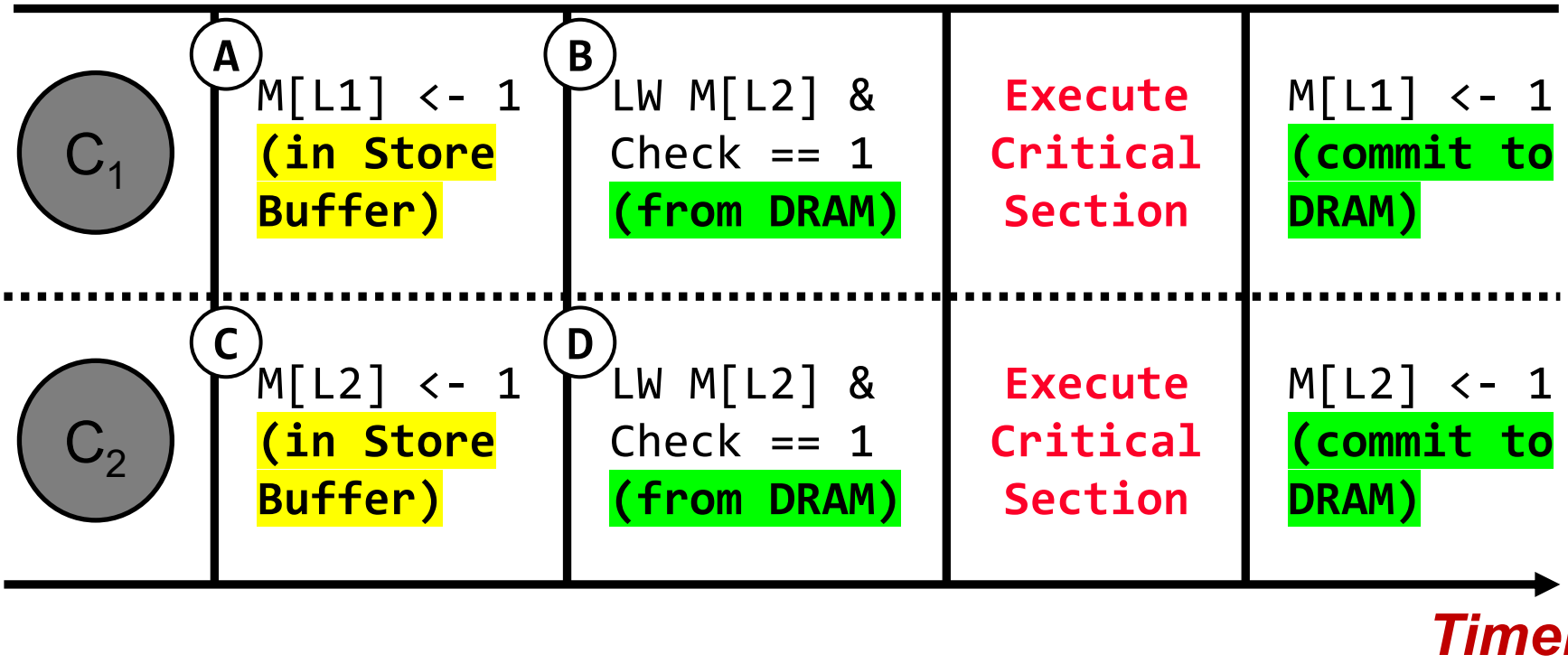


What about this …?
Non-blocking Reads @ NUMA



Dekker's Algorithm w/ Store Buffer

- We cannot assume a sequential consistency w/ out-of-order execution model
  - **In C1's View:** `A -> B -> C  // A is before C`
  - **In C2's View:** `C -> D -> A  // C is before A`



How to resolve this issue?

- **Option #1:** Do not allow all the optimizations (e.g., out-of-order execution …)

- **Option #2:** Use a weaker consistency model instead of sequential consistency
  - The programmer should **expect relaxed ordering** (not all the processors see the same global order)
  - **Solution #1:** Enforce **global ordering for only stores**
    - Processor Consistency (PC)
    - Total store order (TSO) → implemented in x86 processors
  - **Solution #2:** Enforce global order only at the boundary of synchronization
    - Relaxed memory models
    - Acquire-release consistency models

Relaxing Consistency Model

- Multiple relaxation knobs exit (can apply multiple of them)
  - **Option #1:** Relax write to read program order
    - Allowed to read the data before write completes
  - **Option #2:** Relax write to write program order
    - Allowed to write before older write completes
  - **Option #3:** Relax read to read and read to write program order
    - Allowed to read before older read/write completes
  - **Option #4:** Read own write early
    - A processor can read its own write! (Consider load forwarding …)
  - **Option #5:** Read others' write early
    - A processor can read another processors' write before it is "globally" visible ➔ (In some cases, a write is not atomic!)

Relaxing W → R Order

Relaxation	W → R	W → W	R → R/W	Read Own Write Early	Read Others' Write Early
IBM 370	V				
TSO	V			V	
PC	V			V	V

Write is not atomic!

Formulate IBM 370

- TSO Execution requires the following

```
// a and b may or may not be the same
If L(a) <p L(b) ➔ L(a) <m L(b) // Load -> Load
If L(a) <p S(b) ➔ L(a) <m S(b) // Load -> Store
If S(a) <p S(b) ➔ S(a) <m S(b) // Store -> Store
~~If S(a) <p L(b) ➔ S(a) <m L(b) // Store -> Load~~
```

  The memory order may not reflect the store to load program order

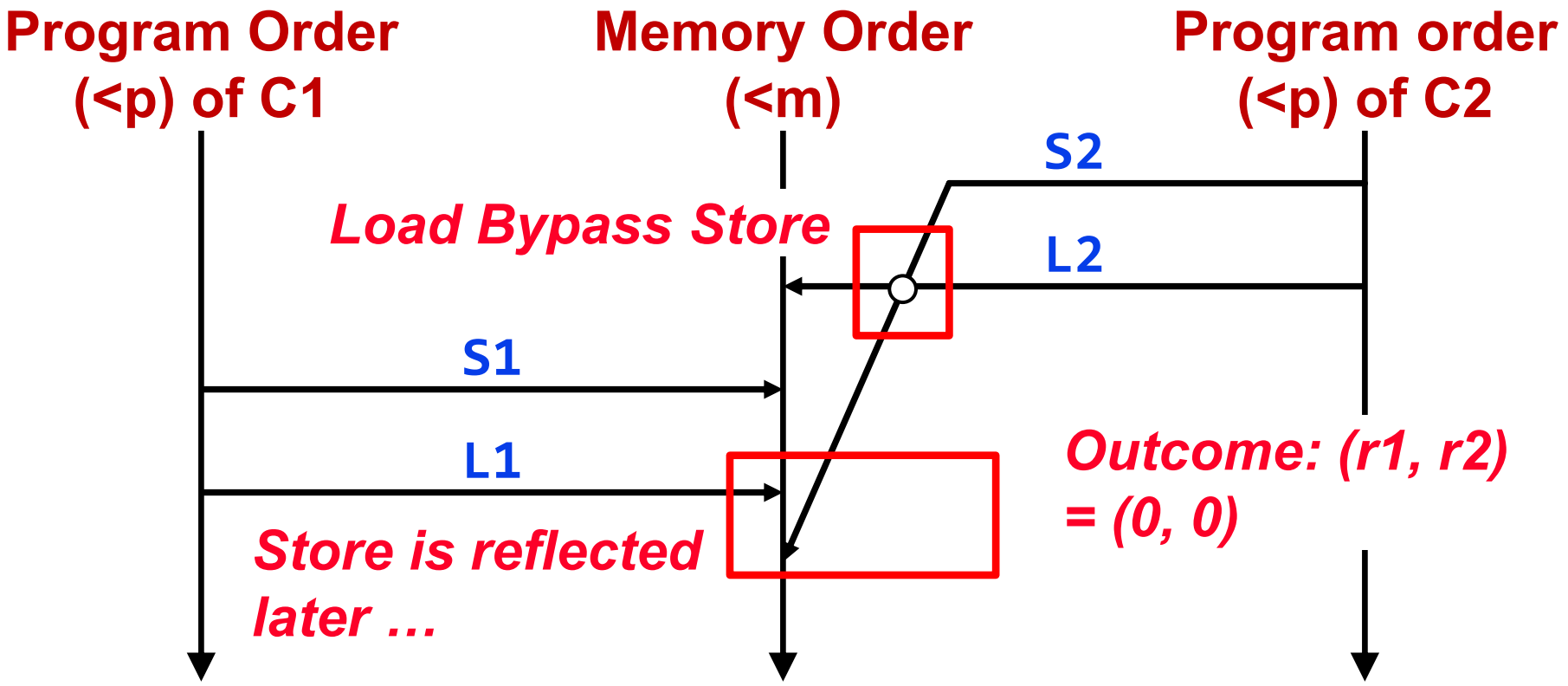
- So,,, every load gets its value from the last store before it (in global order) to the same address:

~~L(a) = Value of MAX <m {S(a) | S(a) <m L(a)}~~

IBM 370

```
// Thread A
// M[f1] = 0 and M[f2] = 0
_loop:
S1:     M[f1] <- SET
L1:     r1 <- M[f2]
```

```
// Thread B
// M[f1] = 0 and M[f2] = 0
_loop:
S2:     M[f2] <- SET
L2:     r2 <- M[f1]
```



IBM 370 vs. Total Store Ordering (TSO)

- TSO allows read its own write early
  - Think about load forwarding

```
// Initially
// M[f1] = 0, M[f2] = 0, M[d] = 0
```

```
// P1
S1p1:   M[f1] <- 1
S2p1:   M[d] <- 1
L1p1:   r1 <- M[d]
L2p1:   r2 <- M[f2]
```

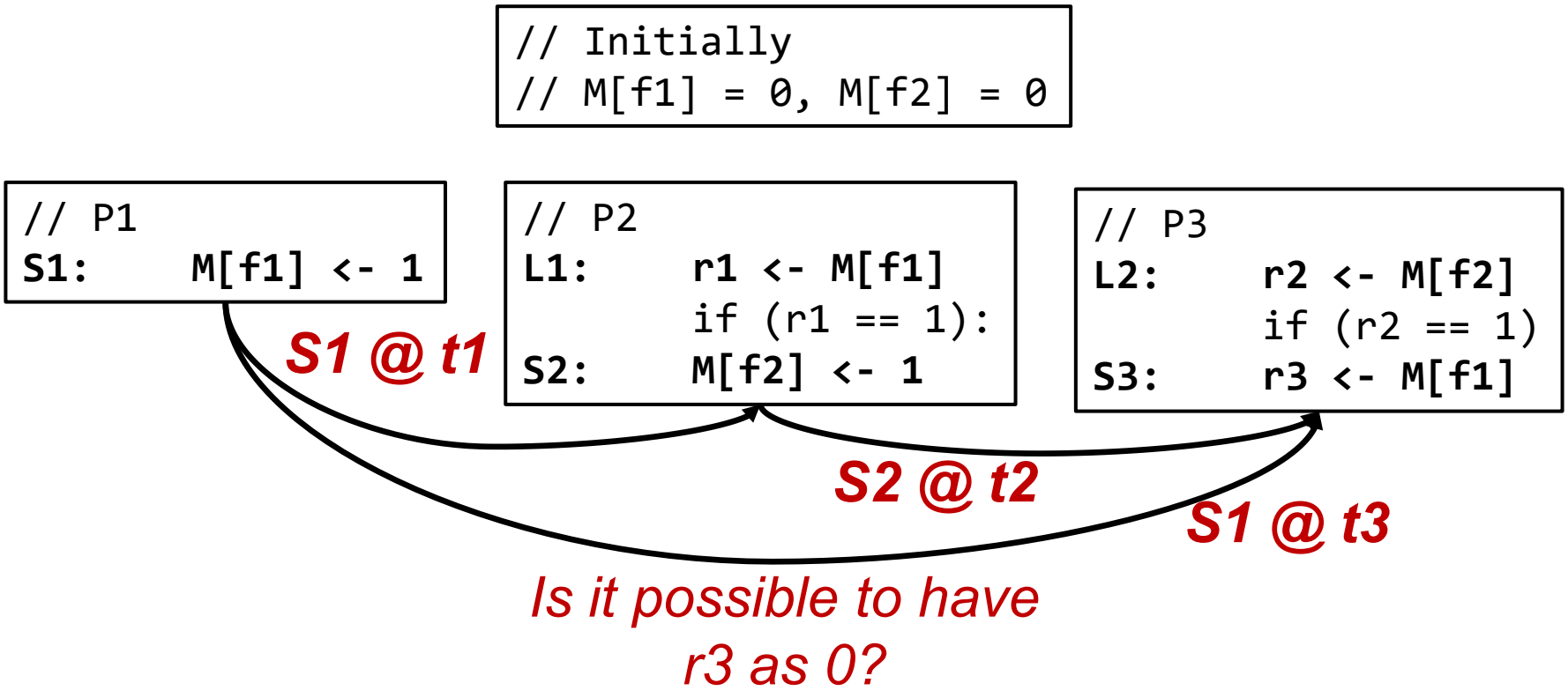
```
// P2
S1p2:   M[f2] <- 1
S2p2:   M[d] <- 2
L1p2:   r3 <- M[d]
L2p2:   r4 <- M[f1]
```

Is it possible to have
r1 = 1, r2 = 0, r3 = 2, r4 = 0?

In IBM 370, L1 must wait until S2 flushes ➔ (force S1 → S2 → L2)
In TSO, L1 can forward data from S2 ➔ (may … L2 → S1 → S2)
➔ Can we have (S1p2 →) L2p2 → S1p1 and (S1p1 →) L2p1 → S1p2?

TSO vs. Processor Consistency (PC)

- PC allows read others' write early



In TSO, S1 should "finish" before L1
In PC, S1 finish (for P2) but not for P3

How to ensure ordering when needed

- Utilizing RMW operations and fences
- RMW cannot be reordered with earlier stores or loads

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	X	X	X	X
	Store	B	X	X	X
	RMW	X	X	X	X
	FENCE	X	X	X	X

Fence Operation

- What if we want to enforce sequential consistency?
- Fence: an explicit ISA that enforces **"memory operations before the fence"** are placed in memory order before **"memory operations after the fence"**

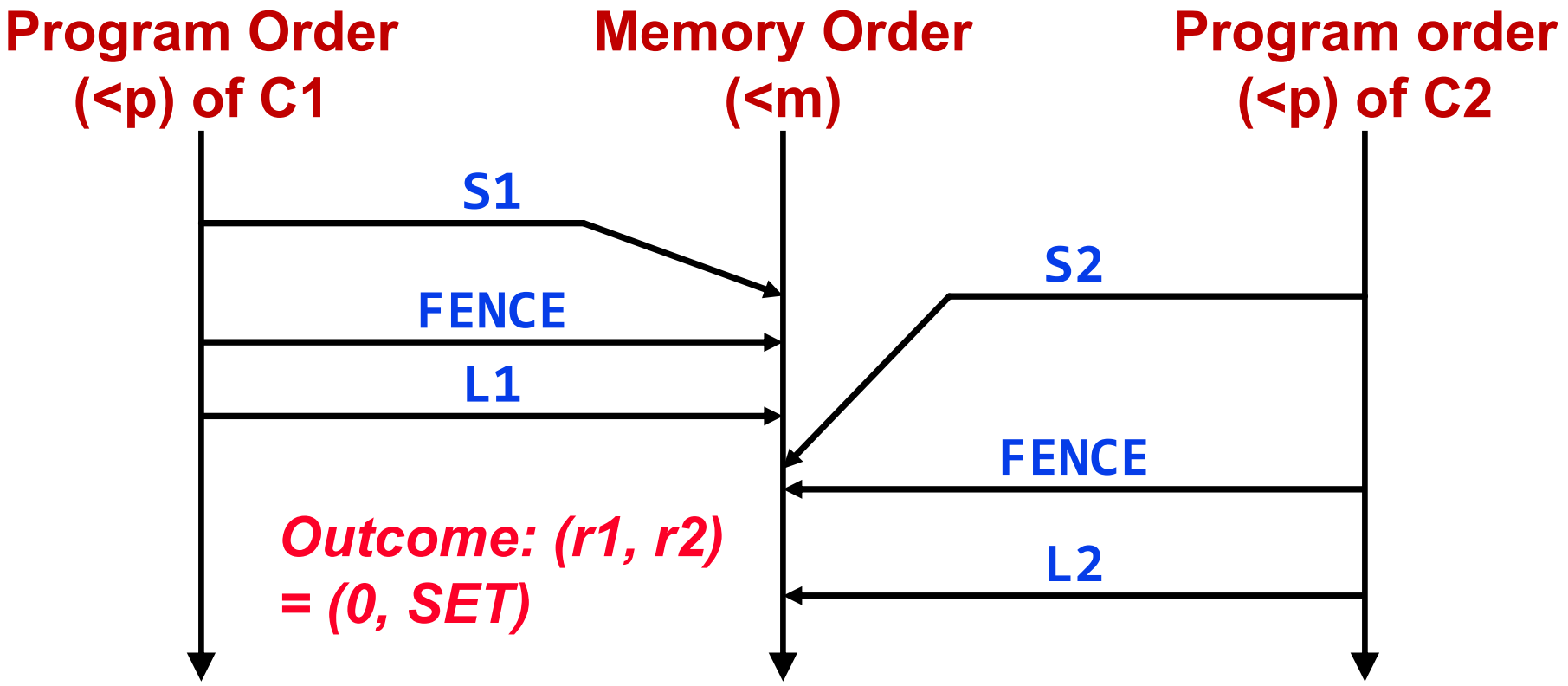
```
// Thread A
// M[x] = 0 and M[y] = 0
_loop:
S1:     M[x] <- SET
        FENCE
L1:     r1 <- M[y]
```

```
// Thread B
// M[x] = 0 and M[y] = 0
_loop:
S2:     M[y] <- SET
        FENCE
L2:     r2 <- M[x]
```

Forcefully drain the store buffer

Fence Operation

- What if we want to enforce sequential consistency in a TSO architecture?
- Fence: an explicit ISA that enforces **"memory operations before the fence"** are placed in memory order before **"memory operations after the fence"**



Formulate IBM 370 Ordering (w/ Fence)

- Additional rules for TSO w/ Fence
  - `If L(a) <p FENCE ➔ L(a) <m FENCE // Load -> FENCE`
  - `If S(a) <p FENCE ➔ S(a) <m FENCE // Store -> FENCE`
  - `If FENCE <p FENCE ➔ FENCE <m FENCE // FENCE -> FENCE`
  - `If FENCE <p L(a) ➔ FENCE <m L(a) // FENCE -> Load`
  - `If FENCE <p S(a) ➔ FENCE <m S(a) // FENCE -> Store`

➔ This is a highly conservative definition of the fence

In fact, Fence is to correct store -> load ordering:

Therefore, we can alternatively define FENCE as

- `If S(a) <p FENCE ➔ S(a) <m FENCE // Store -> FENCE`
- `If FENCE <p L(a) ➔ FENCE <m L(a) // FENCE -> Load`

Is TSO, PC, … Sufficient?

Relaxing W → W Order

Relaxation	W → R	W → W	R → R/W	Read Own Write Early	Read Others' Write Early
IBM 370	V				
TSO	V			V	
PC	V			V	V
PSO	V	V		V	

Partial Store Ordering (PSO)

- TSO Execution requires the following

```
// a and b may or may not be the same
If L(a) <p L(b) ➔ L(a) <m L(b) // Load -> Load
If L(a) <p S(b) ➔ L(a) <m S(b) // Load -> Store
~~If S(a) <p S(b) ➔ S(a) <m S(b) // Store -> Store~~
~~If S(a) <p L(b) ➔ S(a) <m L(b) // Store -> Load~~
```

  The memory order may not reflect the store to load program order

- So,,, every load gets its value from the last store before it (in global order) to the same address:

~~L(a) = Value of MAX <m {S(a) | S(a) <m L(a)}~~

Need for More Relaxed Model

- Assume two threads execute different tasks in a sequential manner

```
// P0
// Initially
// M[A] = 0 & M[F] = 0
S1: M[A] <- NEW
S2: M[F] <- SET
```

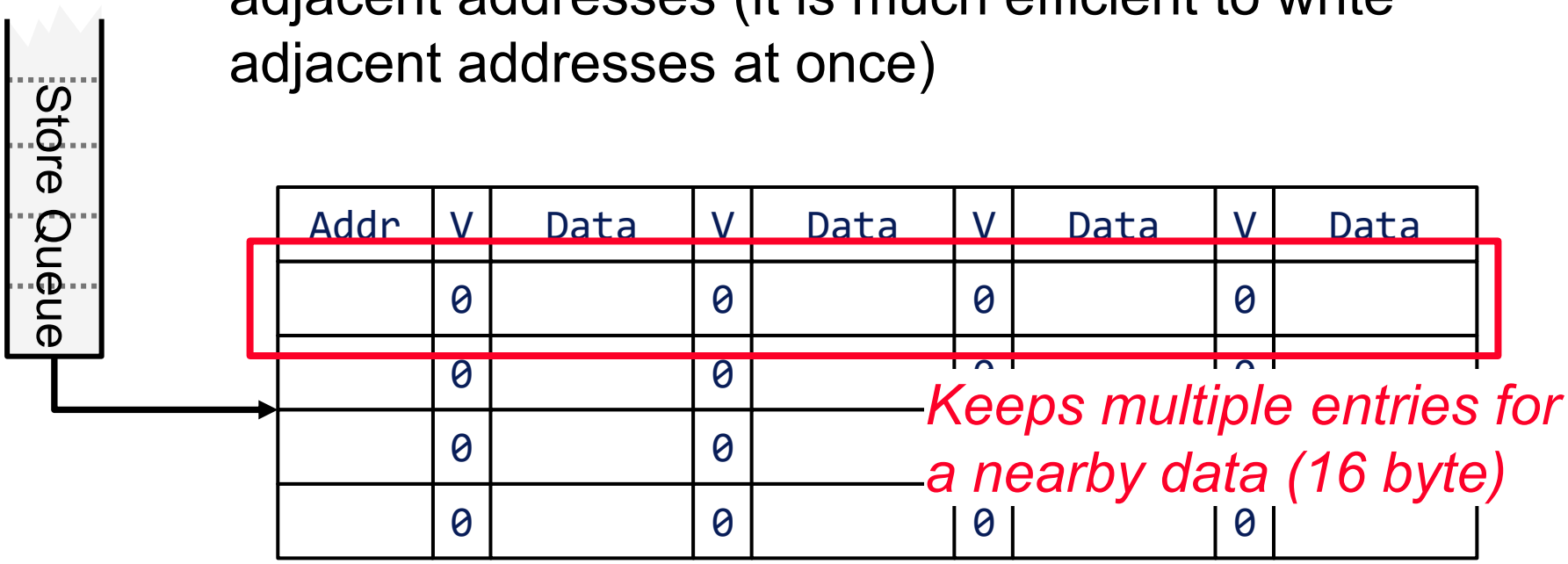
```
// P1
// M[A] = 0 & M[F] = 0
L1: while(M[F] == SET) {}
L2: r2 <- M[A]
```

- There are also multiple possibilities
  - S1 -> S2 -> L1 -> L2
  - S1 -> L1 -> S2 -> L1 -> L2
  - L1 -> L1 -> L1 -> S1 -> S2 -> L1 -> L2
  - What about? **S2 -> L1 -> L2 -> S1 ➔ (r1, r2) = (SET, 0)**

Need for More Relaxed Model
➔ Remember? Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)

- Write buffer may merge multiple entries in the adjacent addresses (it is much efficient to write adjacent addresses at once)

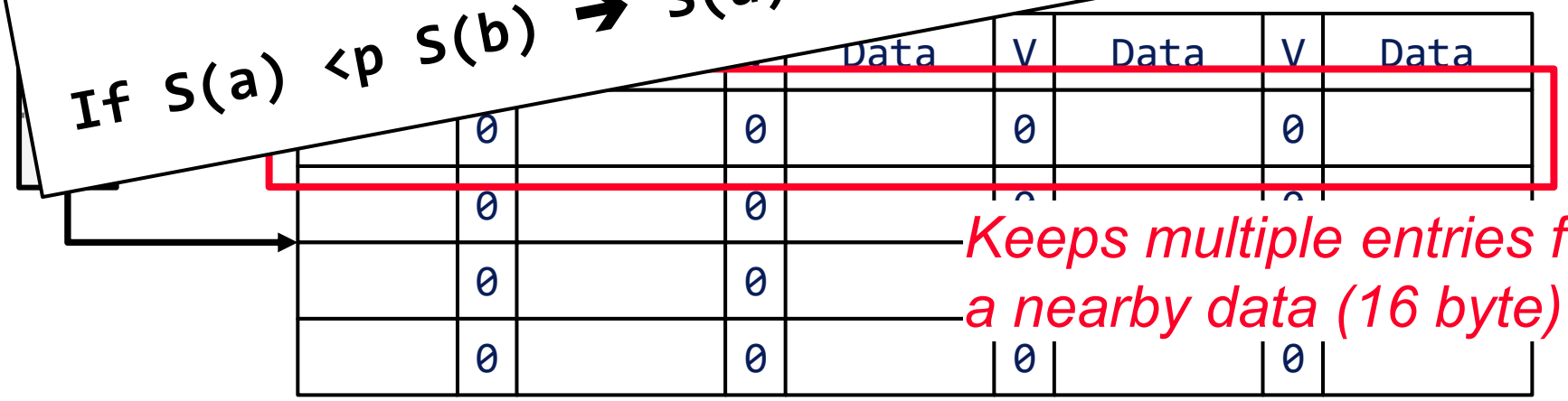


Need for More Relaxed Model
➔ Remember? Merging Write Buffer

- You can adopt a write buffer before writing the data to the memory (or cache as you will learn later on …)
- Write buffer may merge adjacent

Merging Write Buffer Changes The Store Order
Does not Ensure:

If S(a) <p S(b) ➔ S(a) <m S(b) // Store -> Store



Keeps multiple entries for a nearby data (16 byte)

Is PSO Sufficient?

Relaxing R → R/W Order

Relaxation	W → R	W → W	R → R/W	Read Own Write Early	Read Others' Write Early
IBM 370	√				
TSO	√			√	
PC	√			√	√
PSO	√	√		√	
WO	√	√	√	√	
RC	√	√	√	√	√

Weak Consistency (WC)

- WC Execution requires the following

  // a and b may or may not be the same

  ~~If L(a) <p L(b) ➔ L(a) <m L(b) // Load -> Load~~

  ~~If L(a) <p S(b) ➔ L(a) <m S(b) // Load -> Store~~

  ~~If S(a) <p S(b) ➔ S(a) <m S(b) // Store -> Store~~

  ~~If S(a) <p L(b) ➔ S(a) <m L(b) // Store -> Load~~

  - (Of course, does not obey load / store to the same address bypass load / store to the same address)

- Only meet the fences

  If L(a)/S(a) <p FENCE ➔ L(a)/S(a) <m FENCE // Load / Store -> FENCE

  If FENCE <p L(a)/S(a) ➔ FENCE <m L(a)/S(a) // FENCEH / Load / Store

  If FENCE1 <p FENCE2 ➔ FENCE1 <m FENCE2 // FENCE -> FENCE

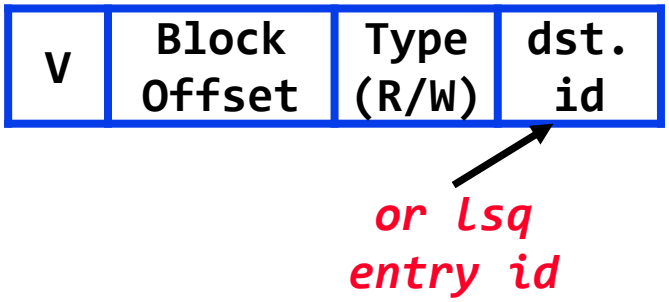
Table 5.5: XC ordering rules. An "X" denotes an enforced ordering. A "B" denotes that bypassing is required if the operations are to the same address. An "A" denotes an ordering that is enforced only if the operations are to the same address. Entries different from TSO are shaded and shown in bold.

		Operation 2			
		Load	Store	RMW	FENCE
Operation 1	Load	**A**	**A**	**A**	X
	Store	B	**A**	**A**	X
	RMW	**A**	**A**	**A**	X
	FENCE	X	X	X	X

Need for further relaxation?
➔ Non-blocking read

- **Remember MSHR?**

V	Block Addr	Slot0	Slot1	Slot2	Slot3
1	0x1200	pend0			
0	-				
1	0x100	pend0	pend1	pend2	**pend3**
1	0x600	pend0	pend1		



- We were able to read the data while another load / store is pending! (resolving cache misses)

Summary

- The consistency model defines the ordering of memory operations (the programmer should take for granted)
  - How does the other processors (core) see the memory operation order of one processor
- SC is the strictest ordering requirement, while there are relaxed consistency models (e.g., TSO, WC, …)
- The consistency model determines to which extent you can apply the memory optimization schemes
  - Store buffer: the load may bypass store
  - Merging write buffer: the store may bypass store
  - MSHR: a younger miss may be handled faster than another

Question?

Announcements:

Reading: *read P&H Ch.6*

Handouts: *none*