CPEN 411: Computer Architecture

Slide Set #14: Memory Consistency

Instructor: Mieszko Lis

Original Slides: Professor Tor Aamodt

1

Introduction to Slide Set 14

- Writing correct software for a shared memory computer requires some understanding of how updates to memory made by one thread can be seen by other threads.

- X86 hardware behaves "mostly" as you would expect, but there are some "corner cases" that might be surprising.

- Other popular architectures such as ARM and PowerPC can provide even more surprising results to naive programmers.

Learning Objectives

By the time we finish talking about this slide set in lectures you should be able to:

– Define sequential consistency.

– Evaluate memory orderings allowed by the TSO/x86 memory model.

– Explain how TSO or more relaxed consistency models differs from sequential consistency.

– Explain what a fence operation is and why it is necessary.

– Define what is meant by a data race

– Define sequential consistency for data race free programs

Example #1

```
Core C1                 Core C2                           Comments

                                                          // Initially data=0,
S1: Store data=NEW                                        // flag != SET
S2: Store flag=SET      L1: Load r1=flag                  // L1 and B1 may
                        B1: if( r1!=SET ) goto L1;        // iterate many times
                        L2: Load r2=data
```

Can r2 get value other than NEW?

4

Example #1

```
Core C1                 Core C2                        Comments

                                                       // Initially data=0,
S1: Store data=NEW                                     // flag != SET
S2: Store flag=SET      L1: Load r1=flag                // L1 and B1 may
                        B1: if( r1!=SET ) goto L1;     // iterate many times
                        L2: Load r2=data
```

Can r2 get value other than NEW?

A: Yes
B: No
C: It depends
D: Not sure

Example #1

```
Core C1                   Core C2                              Comments

                                                               // Initially data=0,
S1: Store data=NEW                                             // flag != SET
S2: Store flag=SET        L1: Load r1=flag                     // L1 and B1 may
                          B1: if( r1!=SET ) goto L1;           // iterate many times
                          L2: Load r2=data
```

Can `r2` get value other than NEW?

A: Yes
B: No
C: It depends ✔
D: Not sure

4

Example #1

```
Core C1              Core C2                          Comments

                                                      // Initially data=0,
S1: Store data=NEW                                    // flag != SET
S2: Store flag=SET   L1: Load r1=flag                 // L1 and B1 may
                     B1: if( r1!=SET ) goto L1;       // iterate many times
                     L2: Load r2=data
```

Can r2 get value other than NEW?

Potential scenarios:
1. C1 and C2 have non-FIFO store buffer, S1 misses and S2 hits
2. L2 executed before L1 (out-of-order execution with speculation)

Example #2

```
Core C1            Core C2            Comments

S1: x=NEW          S2: y=NEW          // Initially x=y=0

L1: r1=y           L2: r2=x
```

Example #2

```
Core C1            Core C2              Comments

S1: x=NEW          S2: y=NEW            // Initially x=y=0

L1: r1=y           L2: r2=x
```

What are "legal" outcomes after above code executes?

Example #2

```
Core C1          Core C2          Comments

S1: x=NEW        S2: y=NEW        // Initially x=y=0

L1: r1=y         L2: r2=x
```

What are "legal" outcomes after above code executes?

A: (r1,r2) = (0,NEW) or (NEW,0)
B: (r1,r2) = (NEW,NEW)
C: (r1,r2) = (0,0)
D: All of the above
E: A or B (but not C)

Example #2

```
Core C1            Core C2              Comments

S1: x=NEW          S2: y=NEW            // Initially x=y=0

L1: r1=y           L2: r2=x
```

What are "legal" outcomes after above code executes?
(r1, r2) = (0, NEW) for execution S1, L1, S2, then L2

Example #2

```
Core C1           Core C2              Comments

S1: x=NEW         S2: y=NEW            // Initially x=y=0

L1: r1=y          L2: r2=x
```

What are "legal" outcomes after above code executes?
(r1, r2) = (0, NEW) for execution S1, L1, S2, then L2
(r1, r2) = (NEW, 0) for S2, L2, S1, and L1

Example #2

```
Core C1          Core C2              Comments

S1: x=NEW        S2: y=NEW            // Initially x=y=0

L1: r1=y         L2: r2=x
```

What are "legal" outcomes after above code executes?
(r1, r2) = (0, NEW) for execution S1, L1, S2, then L2
(r1, r2) = (NEW, 0) for S2, L2, S1, and L1
(r1, r2) = (NEW, NEW), e.g., for S1, S2, L1, and L2

Example #2

```
Core C1              Core C2              Comments

S1: x=NEW            S2: y=NEW            // Initially x=y=0

L1: r1=y             L2: r2=x
```

What are "legal" outcomes after above code executes?
(r1, r2) = (0, NEW) for execution S1, L1, S2, then L2
(r1, r2) = (NEW, 0) for S2, L2, S1, and L1
(r1, r2) = (NEW, NEW), e.g., for S1, S2, L1, and L2

How about following? Is this a legal outcome?
(r1, r2) = (0, 0)

Example #2

Core C1	Core C2	Comments
S1: **x**=NEW	S2: **y**=NEW	// Initially x=y=0
L1: r1=**y**	L2: r2=**x**	

What are "legal" outcomes after above code executes?
(r1, r2) = (0, NEW) for execution S1, L1, S2, then L2
(r1, r2) = (NEW, 0) for S2, L2, S1, and L1
(r1, r2) = (NEW, NEW), e.g., for S1, S2, L1, and L2

How about following? Is this a legal outcome?
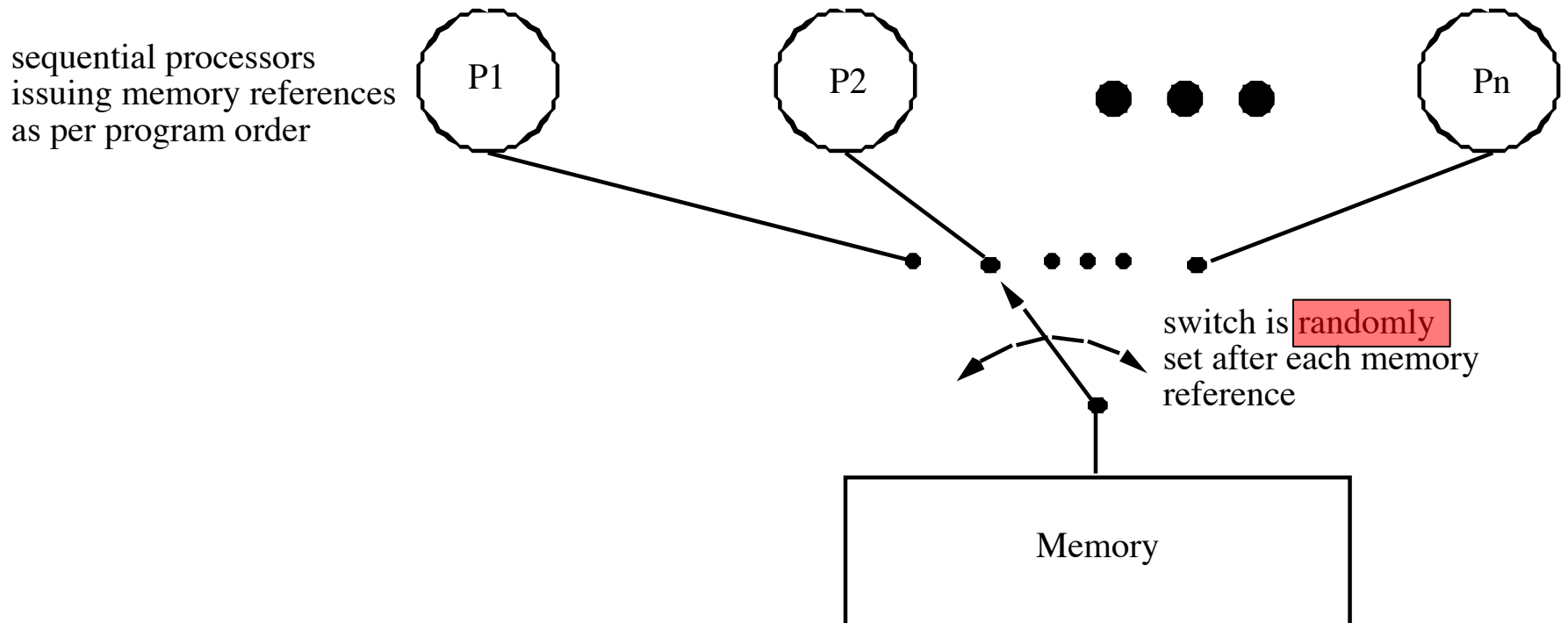(r1, r2) = (0, 0)

This last result can happen on x86 processors from Intel and AMD due to their use of write buffering to enhance performance.

Consistency Model

- Memory consistency model specifies **allowable** orderings of loads and stores to **different locations** (e.g., x and y last slide)

- The number of <u>allowable</u> execution orderings generally far greater than one.

- Ordering of operations from different processors is non-deterministic. Software must use synchronization (mutexes, semaphores, etc…) to provide determinism.

Sequential Consistency

- Sequential consistency is basically a "naïve" programmer's intuition of allowable orderings:

sequential processors
issuing memory references
as per program order

P1

P2

● ● ●

Pn

● ● ●

switch is randomly
set after each memory
reference

Memory

Sequential Consistency

- A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequence, and the operations of each individual processor appear in this sequence in the order specified by its program (Lamport)

- Note: Parallelism still possible (many interleaving orders are equivalent).

Memory Order

- Single writer multiple reader (SWMR) restriction implies all memory operations can be placed into a global "memory order"

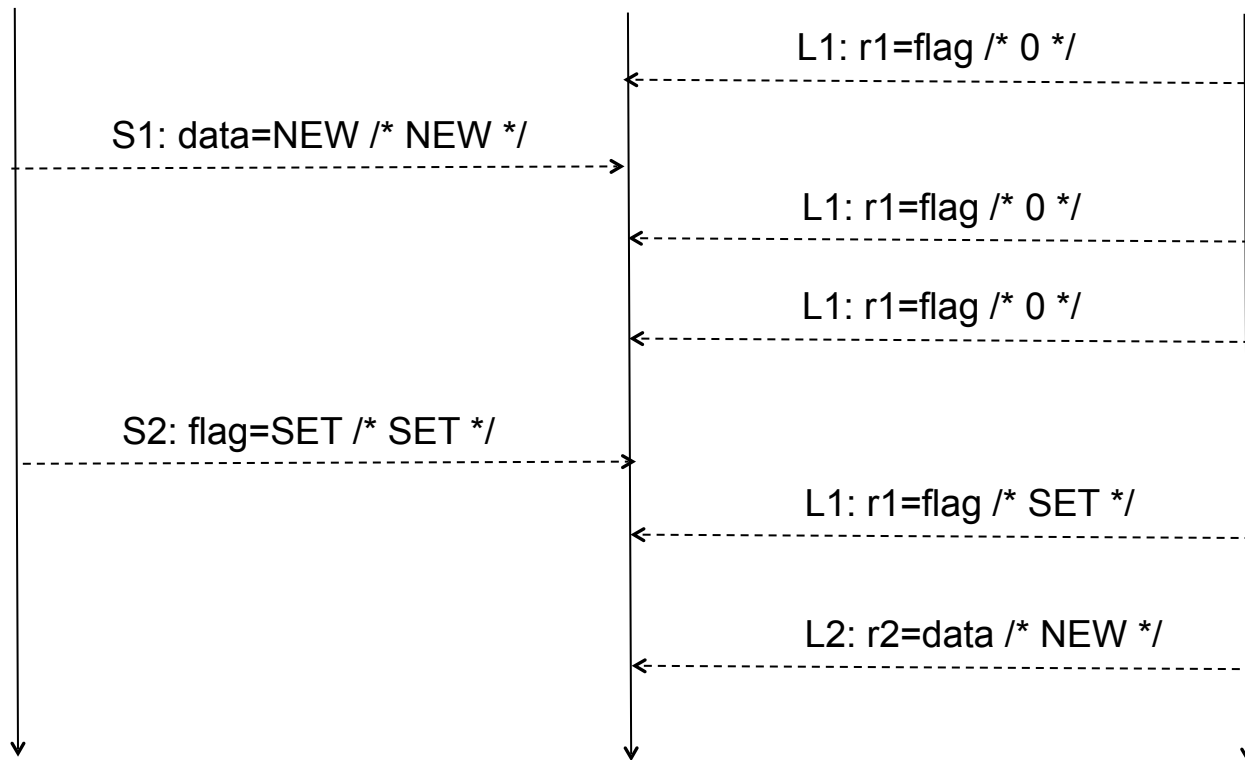- Consistency can then be defined in terms of relationship of "program order" to "memory order"

Example #1, Revisited

Sequential Consistent memory ordering

Program order of core C1   Memory order   Program order of core C2

L1: r1=flag /* 0 */

S1: data=NEW /* NEW */

L1: r1=flag /* 0 */

L1: r1=flag /* 0 */

S2: flag=SET /* SET */
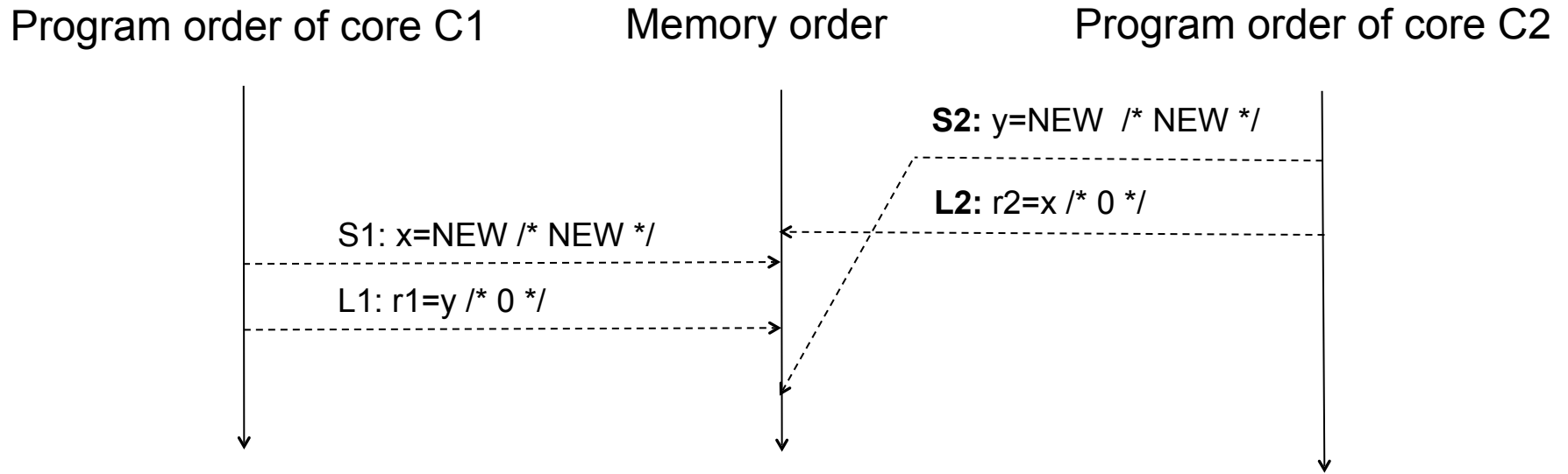
L1: r1=flag /* SET */

L2: r2=data /* NEW */

10

SC Ordering Rules

Operations from a single thread must respect following orderings when placed into memory order:

- Load -> Load
- Load -> Store
- Store -> Store
- Store -> Load

Example #2, Revisited

Example of Non-SC execution:

Program order of core C1   Memory order   Program order of core C2

S2: y=NEW /* NEW */

L2: r2=x /* 0 */

S1: x=NEW /* NEW */

L1: r1=y /* 0 */

S2 and L2 violate "Store -> Load" ordering rule.

Example #1, Revisited

```
Core C1                  Core C2                        Comments

                                                        // Initially data=0,
S1: Store data=NEW                                      // flag != SET
S2: Store flag=SET       L1: Load r1=flag               // L1 and B1 may
                         B1: if( r1!=SET ) goto L1;     // iterate many times
                         L2: Load r2=data
```

Example #1, Revisited

```
Core C1              Core C2                  Comments

                                              // Initially data=0,
S1: Store data=NEW                            // flag != SET
S2: Store flag=SET   L1: Load r1=flag         // L1 and B1 may
                     B1: if( r1!=SET ) goto L1;  // iterate many times
                     L2: Load r2=data
```

Allowable outcomes if memory
system is sequentially consistent?

A: (r1,r2) = (SET,0)
B: (r1,r2) = (SET,NEW)
C: Both A and B
D: Other
E: Not sure

13

Example #1, Revisited

```
Core C1              Core C2                     Comments
                                                 // Initially data=0,
S1: Store data=NEW                               // flag != SET
S2: Store flag=SET   L1: Load r1=flag            // L1 and B1 may
                     B1: if( r1!=SET ) goto L1;  // iterate many times
                     L2: Load r2=data
```

Example #1, Revisited

```
Core C1                  Core C2                      Comments

                                                      // Initially data=0,
S1: Store data=NEW                                    // flag != SET
S2: Store flag=SET       L1: Load r1=flag             // L1 and B1 may
                         B1: if( r1!=SET ) goto L1;   // iterate many times
                         L2: Load r2=data
```

Allowable outcomes if memory system is sequentially consistent?

A: (r1,r2) = (SET,0)
B: (r1,r2) = (SET,NEW) ✔
C: Both A and B
D: Other
E: Not sure

Example #1, Revisited

```
Core C1              Core C2                       Comments
                                                   // Initially data=0,
S1: Store data=NEW                                 // flag != SET
S2: Store flag=SET   L1: Load r1=flag              // L1 and B1 may
                     B1: if( r1!=SET ) goto L1;    // iterate many times
                     L2: Load r2=data
```

Example #2, Revisited

Core C1	Core C2	Comments
S1: **x**=NEW	S2: **y**=NEW	// Initially x=y=0
L1: r1=**y**	L2: r2=**x**	

Example #2, Revisited

```
Core C1            Core C2            Comments

S1: x=NEW          S2: y=NEW          // Initially x=y=0

L1: r1=y           L2: r2=x
```

Allowable outcomes if memory system
is sequentially consistent?

A: (r1,r2) = (0,NEW) or (NEW,0)
B: (r1,r2) = (NEW,NEW)
C: (r1,r2) = (0,0)
D: All of the above
E: A or B (but not C)

Example #2, Revisited

```
Core C1              Core C2              Comments

S1: x=NEW            S2: y=NEW            // Initially x=y=0

L1: r1=y            L2: r2=x
```

Allowable outcomes if memory system is sequentially consistent?

A: (r1,r2) = (0,NEW) or (NEW,0)
B: (r1,r2) = (NEW,NEW)
C: (r1,r2) = (0,0)
D: All of the above
E: A or B (but not C) ✔

Example #2, Revisited

```
Core C1              Core C2              Comments

S1:  x=NEW           S2:  y=NEW           // Initially x=y=0

L1:  r1=y            L2:  r2=x
```
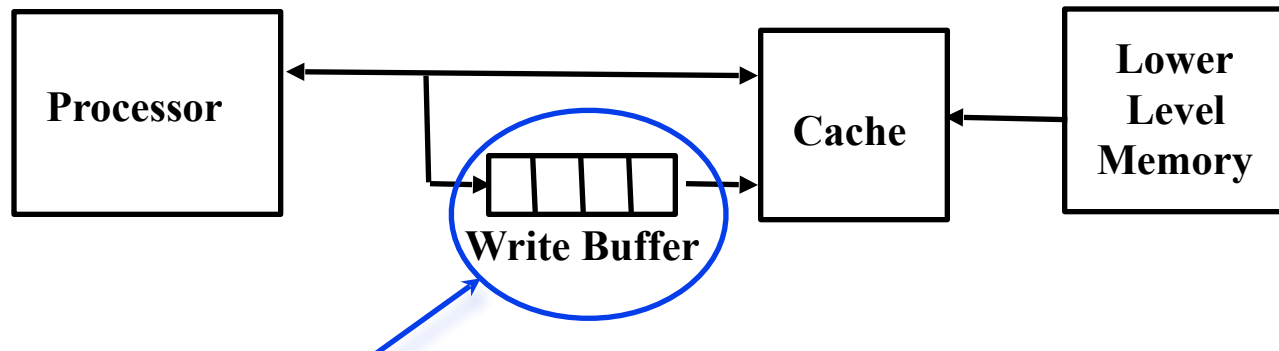
14

- Recall: Stores update memory at commit stage for Tomasulo with reorder buffer.

- Problem: Store miss stalls commit.

- Solution: Avoid stall by storing into a FIFO write buffer like we saw for write through cache.

- Impact on consistency model?

Write Buffer, Revisited

Processor → Cache → Lower Level Memory

Write Buffer

Holds data awaiting write to coherent cache

Q. Why a write buffer ?

A. So CPU doesn't stall

Q. Why a buffer, why not just one register ?

A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?

A. Yes! Drain buffer before next read, or send read after 1st checking write buffers (stall or bypass value if match found)

Total Store Order (TSO/x86) Memory Model

Use of write (store) buffer considered very important by Intel and AMD for x86.

Leads to total store order memory model supported by x86.

In general, memory model on multicore processors is not sequential consistency.

TSO/x86 Ordering Rules

Operations from a single thread must respect following orderings when placed into memory order:

- Load -> Load
- Load -> Store
- Store -> Store
- Store -> ~~Load~~ : TSO/x86 allow loads to appear before prior store in memory order to enable write buffer in each core

Example #1, Revisited

```
Core C1              Core C2                       Comments

                                                   // Initially data=0,
S1: Store data=NEW                                 // flag != SET
S2: Store flag=SET   L1: Load r1=flag              // L1 and B1 may
                     B1: if( r1!=SET ) goto L1;    // iterate many times
                     L2: Load r2=data
```

Example #1, Revisited

```
Core C1                Core C2                   Comments
                                                 // Initially data=0,
S1: Store data=NEW                               // flag != SET
S2: Store flag=SET     L1: Load r1=flag          // L1 and B1 may
                       B1: if( r1!=SET ) goto L1; // iterate many times
                       L2: Load r2=data
```

Allowable outcomes if memory system supports TSO?

A: (r1,r2) = (SET,0)
B: (r1,r2) = (SET,NEW)
C: Both A and B
D: Other
E: Not sure

Example #1, Revisited

```
Core C1                Core C2                       Comments
                                                     // Initially data=0,
S1: Store data=NEW                                   // flag != SET
S2: Store flag=SET     L1: Load r1=flag              // L1 and B1 may
                       B1: if( r1!=SET ) goto L1;    // iterate many times
                       L2: Load r2=data
```

Example #1, Revisited

```
Core C1                  Core C2                      Comments

                                                      // Initially data=0,
S1: Store data=NEW                                    // flag != SET
S2: Store flag=SET       L1: Load r1=flag             // L1 and B1 may
                         B1: if( r1!=SET ) goto L1;   // iterate many times
                         L2: Load r2=data
```

Allowable outcomes if memory
system supports TSO?

A: (r1,r2) = (SET,0)
B: (r1,r2) = (SET,NEW) ✔
C: Both A and B
D: Other
E: Not sure

Example #1, Revisited

```
Core C1              Core C2                      Comments
                                                  // Initially data=0,
S1: Store data=NEW                                // flag != SET
S2: Store flag=SET   L1: Load r1=flag             // L1 and B1 may
                     B1: if( r1!=SET ) goto L1;   // iterate many times
                     L2: Load r2=data
```

Example #2, Revisited

Core C1

S1: **x**=NEW

L1: r1=**y**

Core C2

S2: **y**=NEW

L2: r2=**x**

Comments

// Initially x=y=0

Example #2, Revisited

```
Core C1              Core C2              Comments

S1: x=NEW            S2: y=NEW            // Initially x=y=0

L1: r1=y             L2: r2=x
```

Allowable outcomes if memory system supports TSO?

A: (r1,r2) = (0,NEW) or (NEW,0)
B: (r1,r2) = (NEW,NEW)
C: (r1,r2) = (0,0)
D: All of the above
E: A or B (but not C)

Example #2, Revisited

```
Core C1            Core C2            Comments

S1: x=NEW          S2: y=NEW          // Initially x=y=0

L1: r1=y           L2: r2=x
```

Allowable outcomes if memory system supports TSO?

A: (r1,r2) = (0,NEW) or (NEW,0)
B: (r1,r2) = (NEW,NEW)
C: (r1,r2) = (0,0)
D: All of the above ✔
E: A or B (but not C)

Example #2, Revisited

Core C1

S1: **x**=NEW

L1: r1=**y**
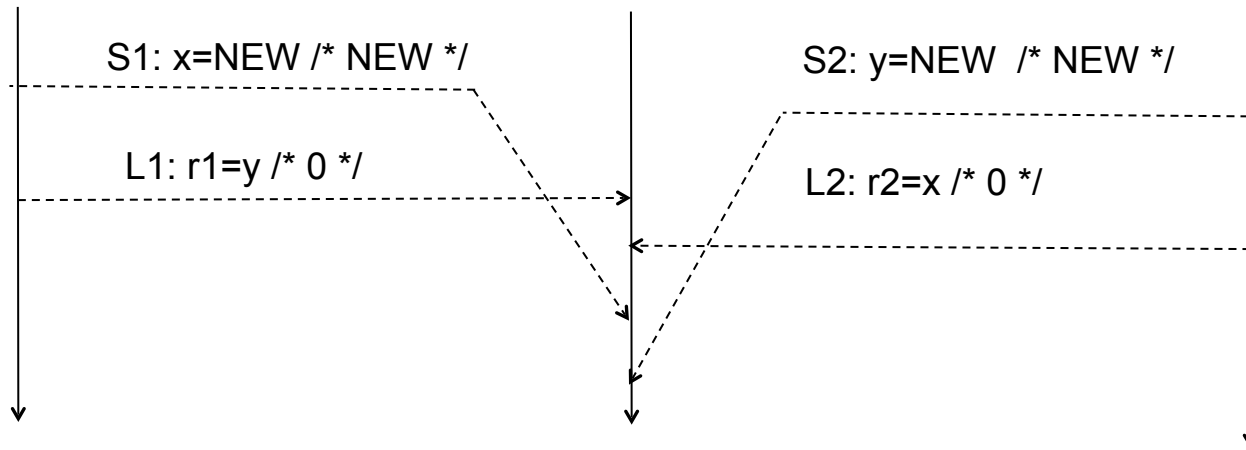
Core C2

S2: **y**=NEW

L2: r2=**x**

Comments

// Initially x=y=0

Example #2, TSO/x86 ordering

Program order of core C1   Memory order   Program order of core C2

S1: x=NEW /* NEW */   S2: y=NEW /* NEW */

L1: r1=y /* 0 */   L2: r2=x /* 0 */

(r1, r2) = (0, 0) is legal outcome under TSO/x86 (!)

Fence Operations

- Hardware provides ISA support to enable intended execution of code in Example #2.

- Fence instruction ensures prior memory operations inserted into memory order before subsequent memory operations. E.g., for TSO require:

      Store -> Fence

      Fence -> Load

Example #2, Revisited

```
Core C1          Core C2              Comments

S1:  x=NEW       S2:  y=NEW           // Initially x=y=0

F1:  Fence       F2:  Fence

L1:  r1=y        L2:  r2=x
```

Example #2, Revisited

```
Core C1          Core C2              Comments

S1: x=NEW        S2: y=NEW            // Initially x=y=0

F1: Fence        F2: Fence

L1: r1=y         L2: r2=x
```

Allowable outcomes if memory system supports TSO?

A: (r1,r2) = (0,NEW) or (NEW,0)
B: (r1,r2) = (NEW,NEW)
C: (r1,r2) = (0,0)
D: All of the above
E: A or B (but not C)

Example #2, Revisited

```
Core C1              Core C2              Comments

S1:  x=NEW           S2:  y=NEW           // Initially x=y=0

F1:  Fence           F2:  Fence

L1:  r1=y            L2:  r2=x
```

Allowable outcomes if memory system supports TSO?

A: (r1,r2) = (0,NEW) or (NEW,0)
B: (r1,r2) = (NEW,NEW)
C: (r1,r2) = (0,0)
D: All of the above
E: A or B (but not C) ✔

Example #2, Revisited

```
Core C1            Core C2              Comments

S1: x=NEW          S2: y=NEW            // Initially x=y=0

F1: Fence          F2: Fence

L1: r1=y           L2: r2=x
```

Relaxed Memory Consistency

- PowerPC, ARM increase performance by enforcing memory ordering ONLY when programmer asks for it explicitly

- Require fence operations to enforce most ordering

Related: Write atomicity

- When performing a store does new value become visible to all other threads at same time?

- Yes: called write atomicity (or store atomicity)

- PowerPC & ARM: Do not enforce write atomicity (breaks SWMR assumption)

Example 3: Causality

(message delay for A)

/* Assume initial value of A and B are 0 */
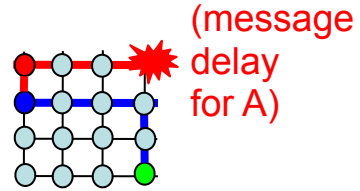
| P₁ | P₂ | P₃ |

```
A = 1;  ⟶  while (A == 0);/*spin wait */
              B = 1;  ⟶  while (B == 0);
                           Print A;
```

- Causality: "If I see it and tell you about it, then you see it too."

- Programmer's intuition about causality ("Print A" displays "1") may be violated on a system lacking write atomicity

When do we need fences?

- Typically, programmer uses "synchronization" to protect accesses to shared data.

- E.g., critical section

  aquire(mutex) // synchronization
  modify shared data
  release(mutex) // synchronization

Data Races

- Data race: Two or more threads accessing same data with one thread writing and no synchronization between accesses

SC for DRF Programs

- Fact: All memory consistency models support sequential consistency for programs free of data races.

- Avoid needing to worry about hardware consistency model if you use correct synchronization (e.g., mutex)