CSEP 548: Computer Systems Architecture

Memory Consistency Models and Synchronization

Luis Ceze, Spring 2017

(based on slides lifted from friends at UPenn, UIUC, UW, MIT, CMU)

Coherence vs. Consistency

```
                A=0  flag=0
Processor 0                 Processor 1
A=1;                        while (!flag); // spin
flag=1;                     print A;
```

- **Intuition says?**

Coherence vs. Consistency

```
                A=0  flag=0
Processor 0                 Processor 1
A=1;                        while (!flag); // spin
flag=1;                     print A;
```

- **Intuition says?** P1 prints A=1
- **Coherence says?**

Coherence vs. Consistency

```
                A=0   flag=0
Processor 0                 Processor 1
A=1;                        while (!flag); // spin
flag=1;                     print A;
```

- **Intuition says**: P1 prints A=1
- **Coherence says**: absolutely nothing
  - P1 can see P0's write of **`flag`** before write of **`A`**!!!
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **Real systems** are allowed to act in this strange manner
  - What is allowed? defined as part of the ISA and/or language

– OMG, Why???

What is Going On?

- Reordering of memory operations to **different** addresses!

- **In the compiler**
  - Compiler is generally allowed to re-order memory operations to different addresses
  - Many other compiler optimizations also cause problems

- **In the hardware**
  - To tolerate write latency
    - Processes don't wait for writes to complete
    - Write coalescing, etc..
    - And why should they? No reason on a uniprocessors
  - To simplify out-of-order execution

Memory Consistency

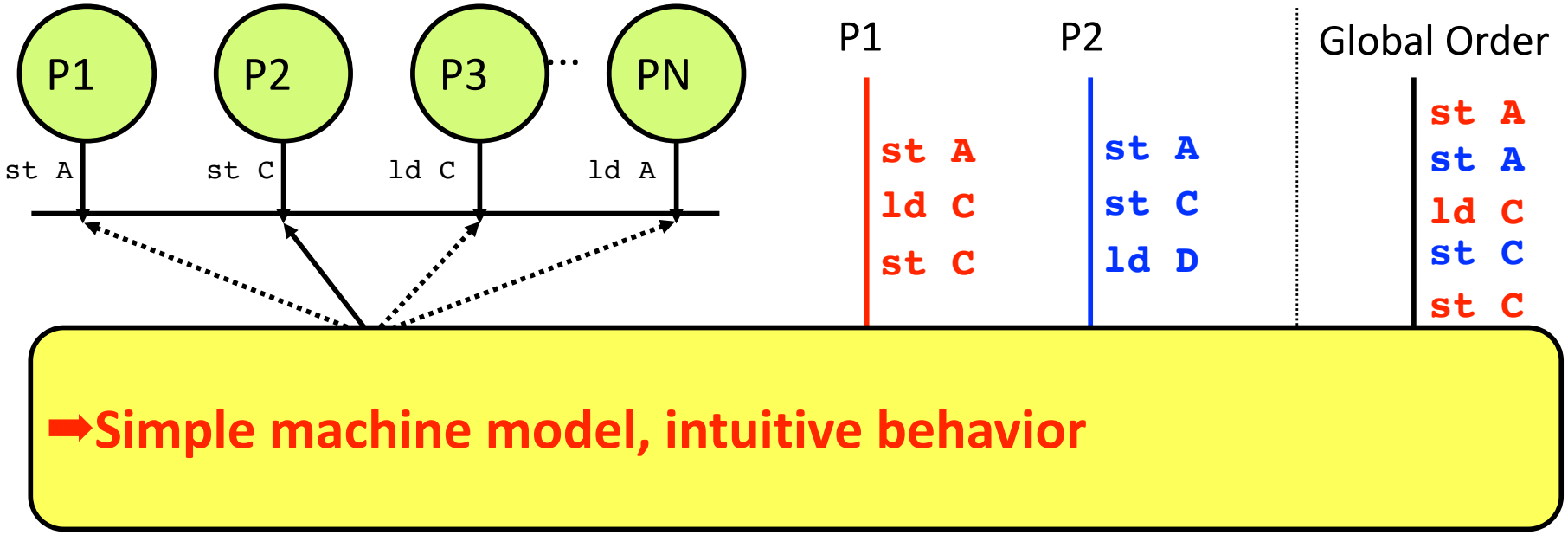
- **Memory coherence**
  - Creates globally uniform (consistent) view…
  - Of **a single memory location** (in other words: cache blocks)
  - Not enough
    - Cache blocks A and B can be individually consistent…
    - But inconsistent with respect to each other

- **Memory consistency**
  - Creates globally uniform (consistent) view…
  - Of **all memory locations relative to each other**

- Who cares? Programmers/Compiler writers/HW Designers
  - Globally inconsistent memory creates mystifying behavior
  - Reordering is key to performance

What is Sequential Consistency?

Sequential Consistency (SC)



Per-processor program order: memory operations from individual processors maintain program order

Single sequential order: the memory operations from all processors maintain a single sequential order

[Lamport'79]

Pop Quiz! (assume SC)

- Answer the following questions:
  - Initially: all variables zero (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
`ld x` `ld y`	`st 1 → y` `st 1 → x`

  - What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
`st 1 → y` `ld x`	`st 1 → x` `ld y`

  - What value can be read by the load A?

thread 1	thread 2
`st 1 → A` `st 1 → flag`	`while(flag == 0) { }` `ld A`

Pop Quiz Again! (assume SC)

- Answer the following questions:
  - Initially: all variables zero (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
`ld x` `ld y`	`st 1 → y` `st 1 → x`

How about (1,0)?

  - What value pairs can be read by the two loads? (x, y) pairs:

thread 1	thread 2
`st 1 → y` `ld x`	`st 1 → x` `ld y`

How about (0,0)?

  - What value can be read by the load A?

thread 1	thread 2
`st 1 → A` `st 1 → flag`	`while(flag == 0) { }` `ld A`

Online ”game”

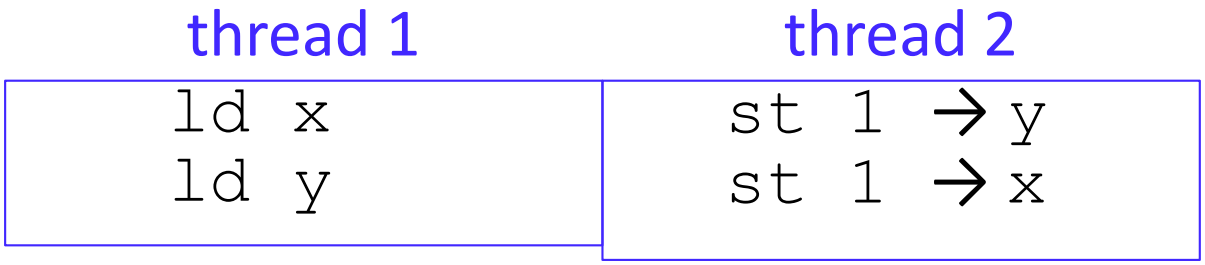
What about this test?

```
=====================
Initial: A=0 /\ B=0
=====================
P0         | P1
---------------------
r1 <- [A]  | [B] <- 1
r2 <- [B]  | [A] <- 1
=====================
Outcome: r1=1 /\ r2=0
```

Allowed | Forbidden | Start Again

Detecting Violations of SC

x = y = 0



x = 1, y = 0?

Is Sequential Consistency Practical?

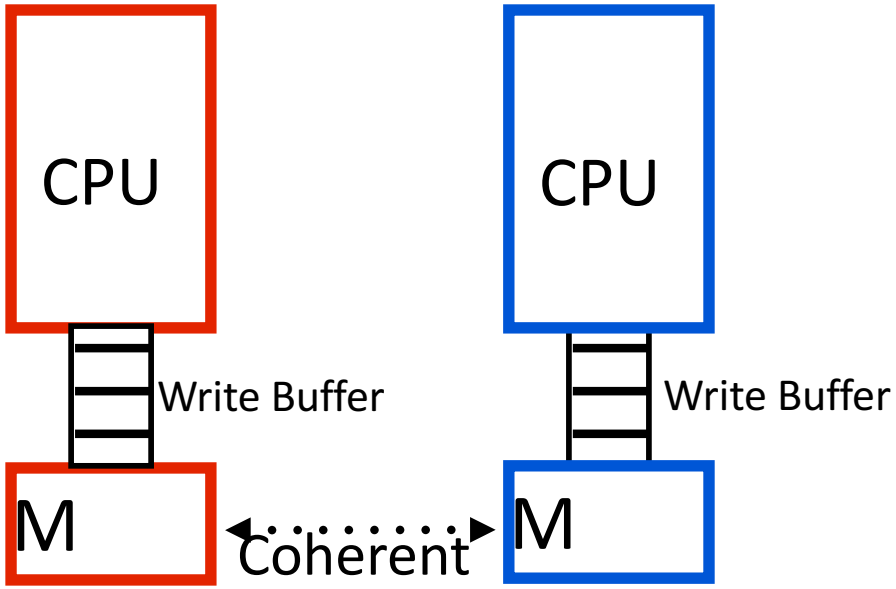
- Why? Why not?

Is Sequential Consistency Practical?

- Well…
- SC constrains all memory operations:
    - Write –> Read, Write
    - Read –> Read, Write

- But: Modern microprocessors reorder operations all the time to obtain performance (write buffers, overlapped writes,non-blocking reads…).

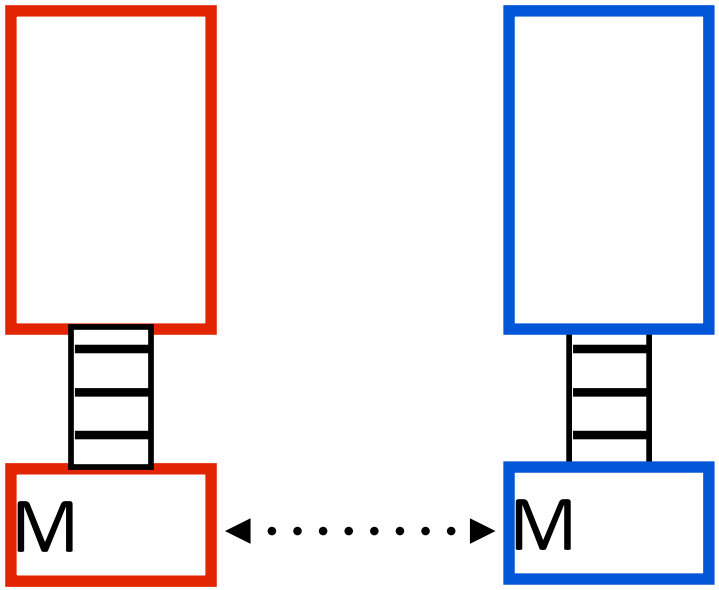
Reordering #1: Write Buffers



CPU can read its write buffer, but not others'

Buffered writes eventually end up in coherent shared memory

Reordering #1: Write Buffers



Program

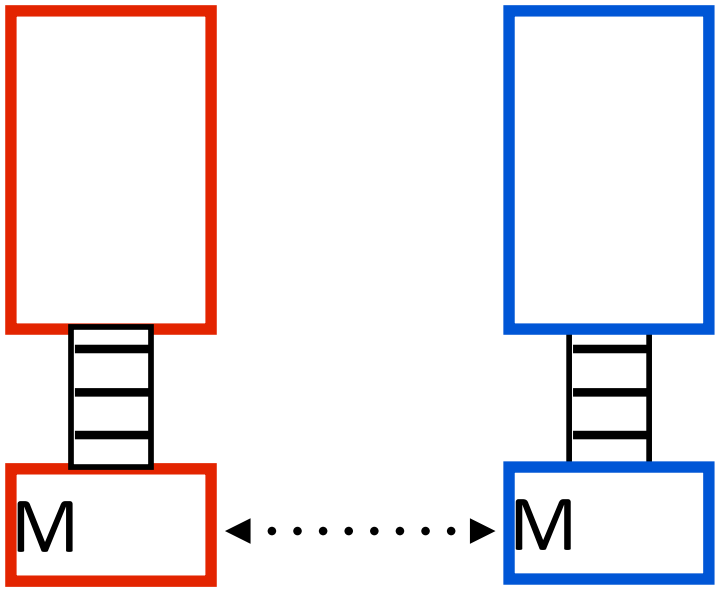
Initially X == Y == 0

X=1 Y=1

r1=Y r2=X

Is r1==r2==0
a valid result?

Reordering #1: Write Buffers



Program

Initially X == Y == 0

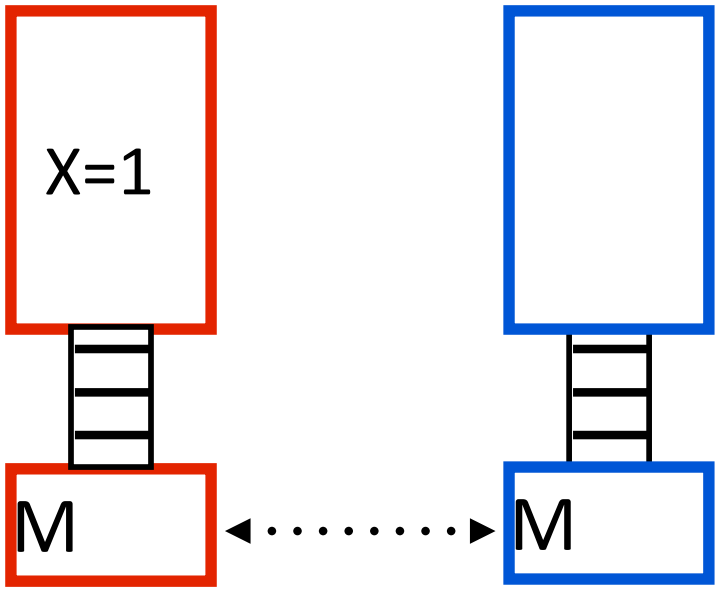
X=1 Y=1

r1=Y r2=X

Is r1==r2==0
a valid result?

r1 == r2 == 0 is **not** SC, but it can happen with write buffers

Reordering #1: Write Buffers



Program

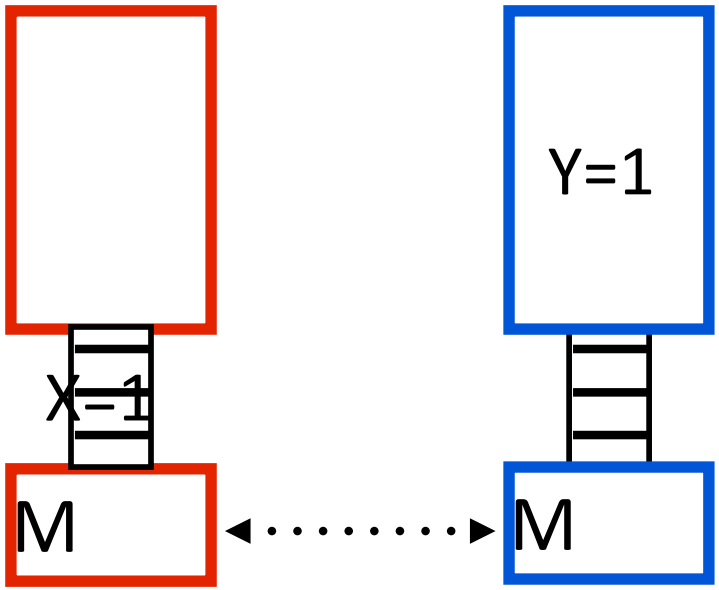
Initially X == Y == 0

Y=1

r1=Y r2=X

Execution

Reordering #1: Write Buffers



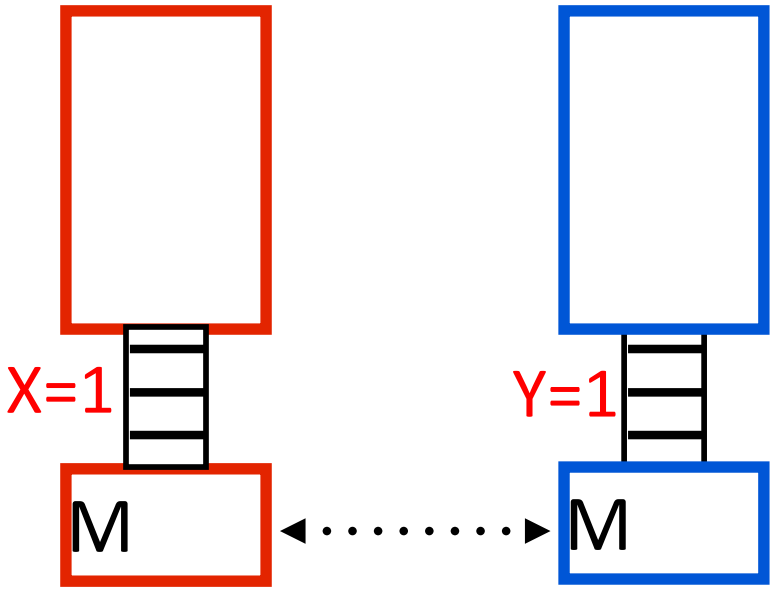
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



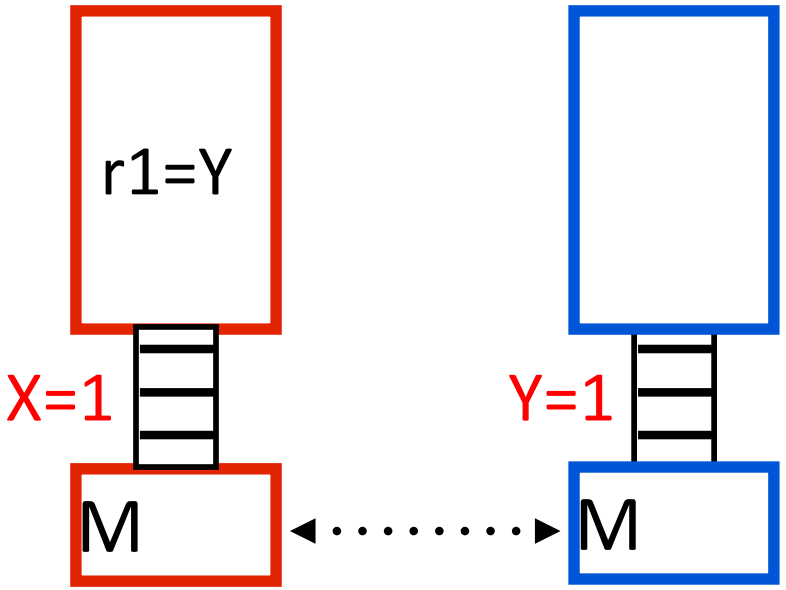
Program

Initially X == Y == 0

r1=Y r2=X

Execution

Reordering #1: Write Buffers



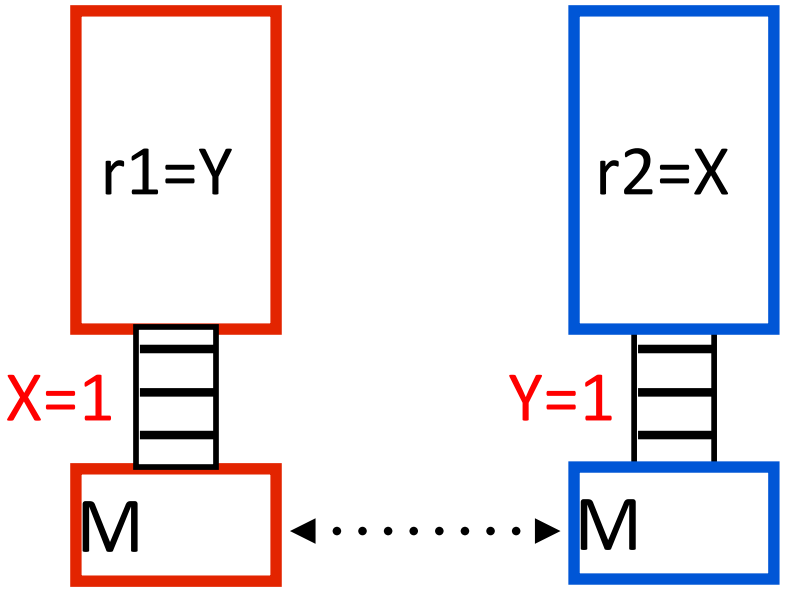
Program

Initially X == Y == 0

r2=X

Execution

Reordering #1: Write Buffers

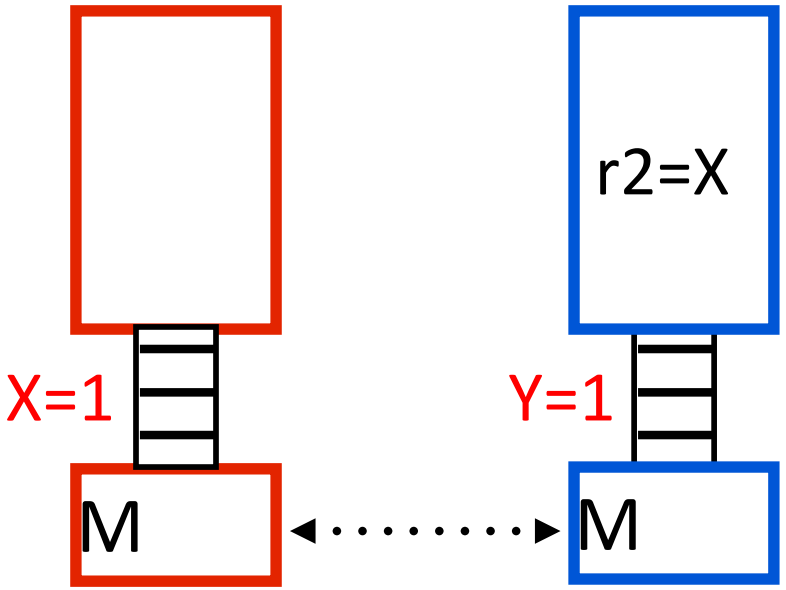


Program

Initially X == Y == 0

Execution

Reordering #1: Write Buffers



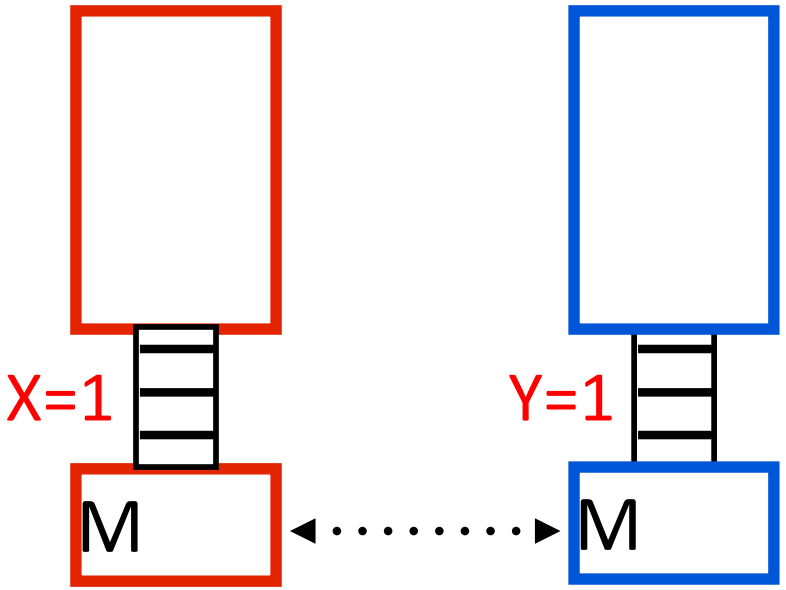
Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

Reordering #1: Write Buffers



Program

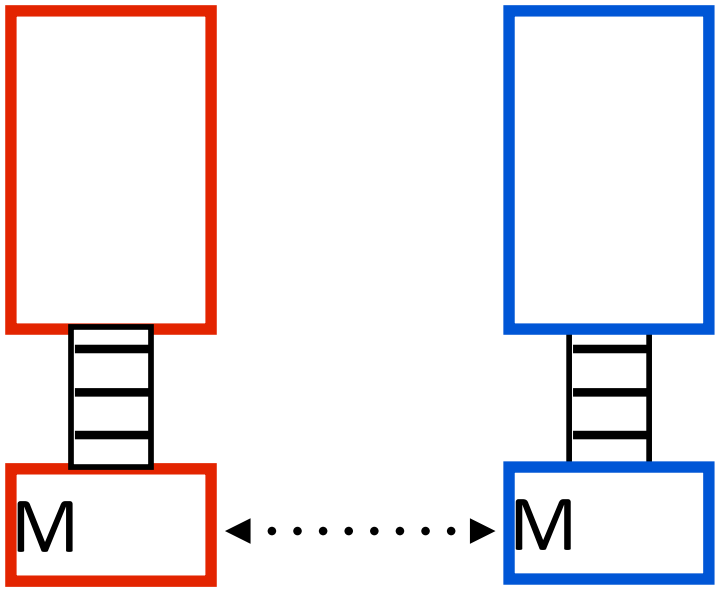
Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

Reordering #1: Write Buffers



WBs let reads finish before older writes

Program

Initially X == Y == 0

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

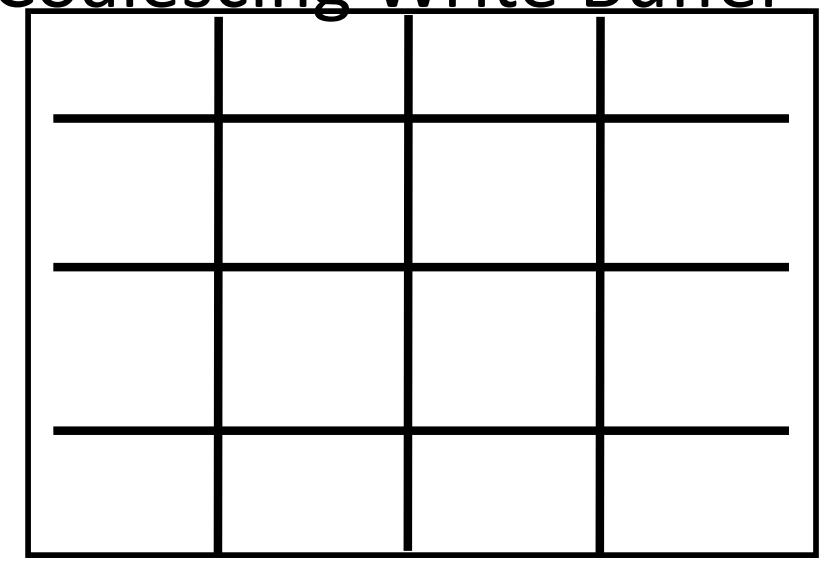
X=1

Y=1

(Not SC!)

Reordering #2: Write Combining

Coalescing Write Buffer



4 word cache line

Program

X,Z in same $ line

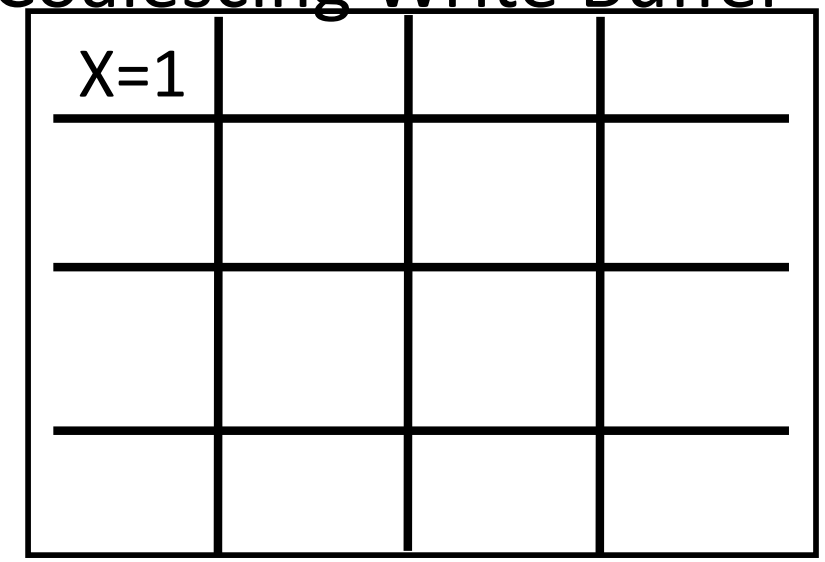
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

X,Z in same $ line

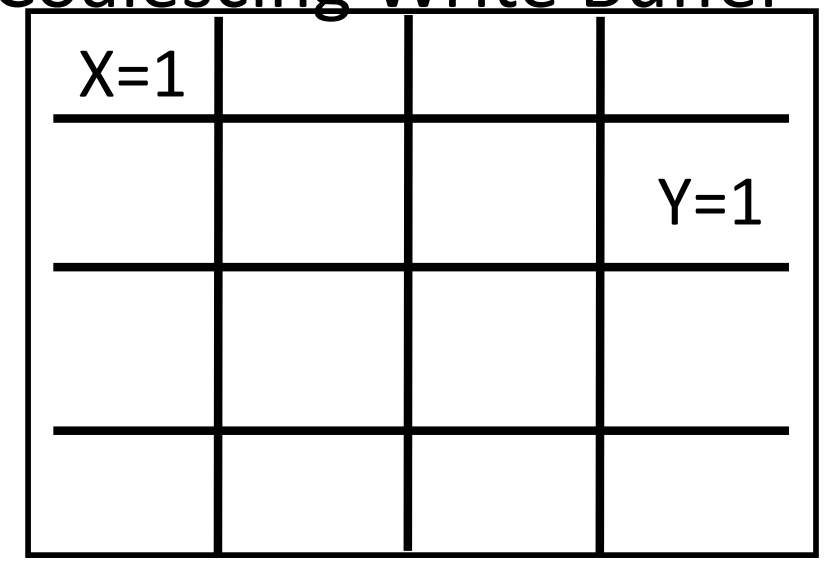
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

X,Z in same $ line

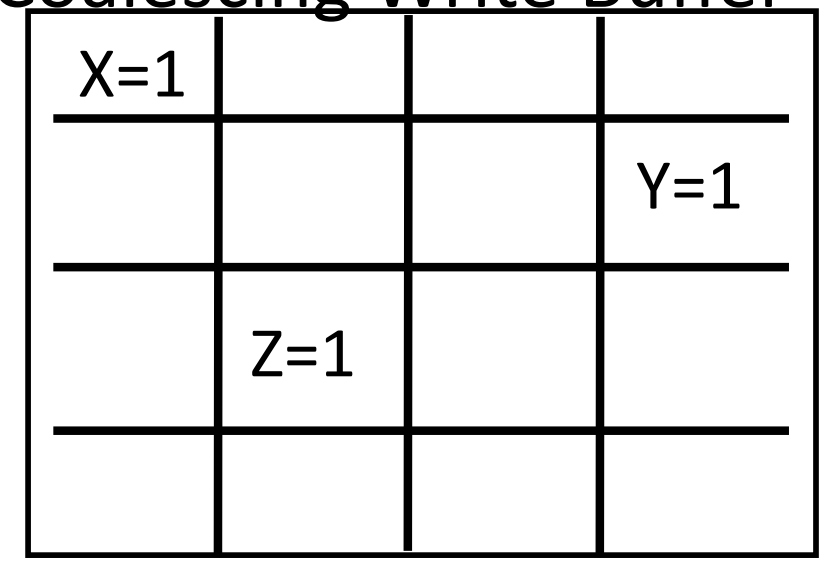
X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer



Program

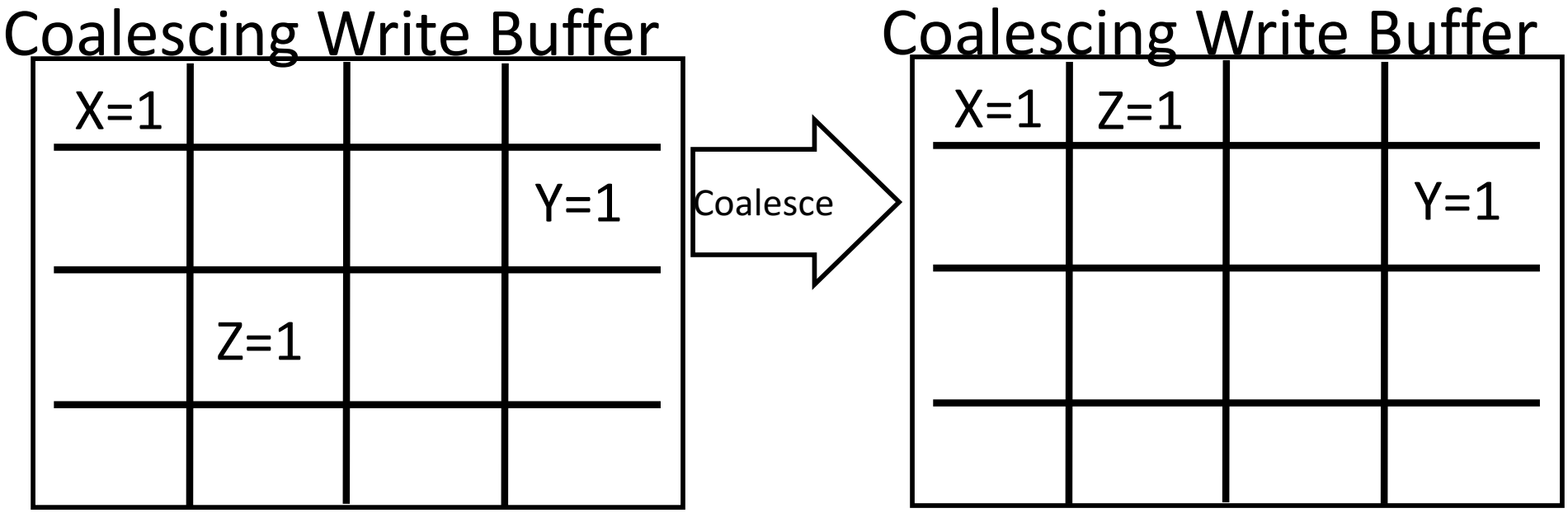
X,Z in same $ line

X=1

Y=1

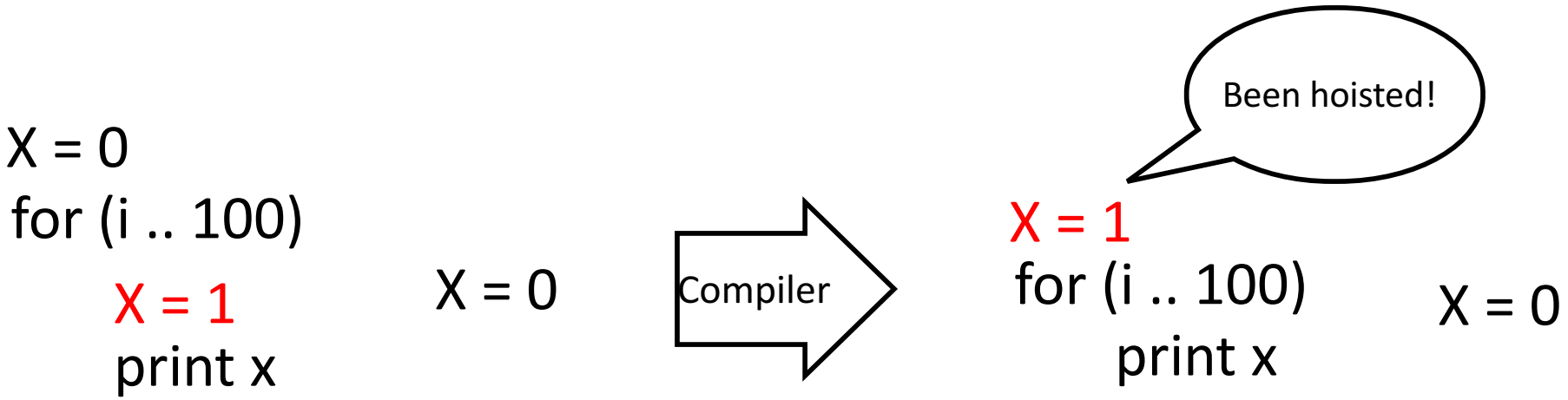
Z=1

Reordering #2: Write Combining



Combining the write to X & Z saves bandwidth, but **reorders** Z=1 and Y=1

Reordering #3: Compilers



The compiler hoists the write out of the loop, permitting new (non-SC) results (e.g., “1 0 0 0 0 0 0 0…”)

What else may break SC?

- Instruction scheduling?
- DCE?
- Common sub-expression elimination?
- Load scheduling?
- <anything else?>

Enforcing SC

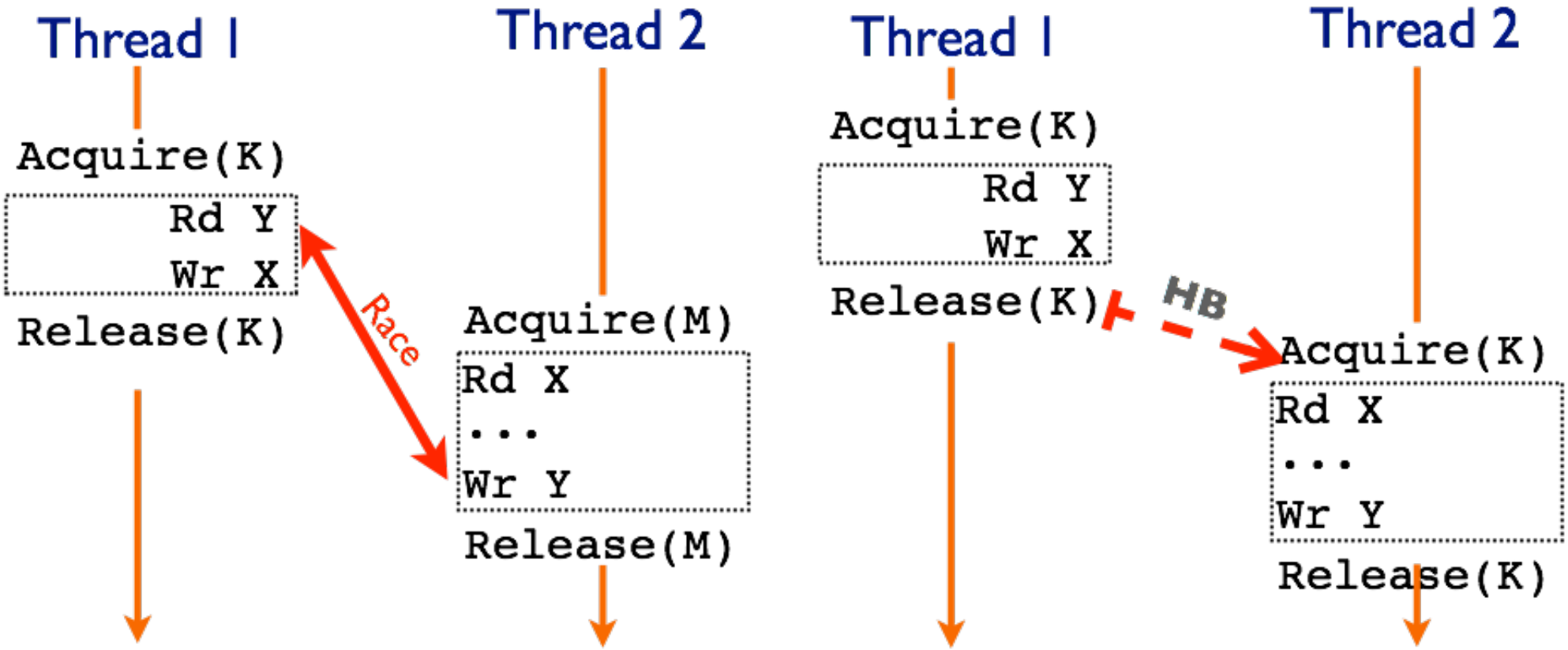
- What does it take to enforce SC?
  - Definition: all loads/stores globally ordered
  - Use ordering of coherence events to order all loads/stores
- **When do coherence events happen naturally?**
  - On cache access
  - For stores: retirement –> in-order –> good
    - No write buffer? Yikes, but OK with write-back D$
  - For loads: execution –> out-of-order –> bad
    - No out-of-order execution? Double yikes
- How about make multiprocessors inorder?
  - That would be really bad
  - Out-of-order is needed to hide cache miss latency
  - And multi-processors not only have more misses…
  - … but miss handling takes longer (coherence actions)

I want SC, and I want it to be fast! Hmmm..

- When do we really care that operations are done in order?
  - *Always?*
- What if we had more information about the program?

What is a data-race?

Many intuitive definitions, but *one t*echnical definition for memory model purposes: two accesses from different threads; at least one a write; accessing the same **location**; *without explicit happens-before ordering via synchronization.*



Relaxing Program Orders

- Divide memory operations into data operations and synchronization operations
- Synchronization operations act like a fence:
  - All data operations before synch in program order must complete before synch is executed
  - All data operations after synch in program order must wait for synch to complete
  - Synchs are performed in program order
- Implementation of fence: processor has counter that is incremented when data op is issued, and decremented when data op is completed
  - A fence is a effectively a local ***passive*** operation
- Major implications on language-level memory models (more later)

Fences aka Memory Barriers

- **Fences (memory barriers)**: special insns
  - Ensure that loads/stores don't cross acquire release boundaries
  - Very roughly

```
acquire
fence
critical section
fence
release
```

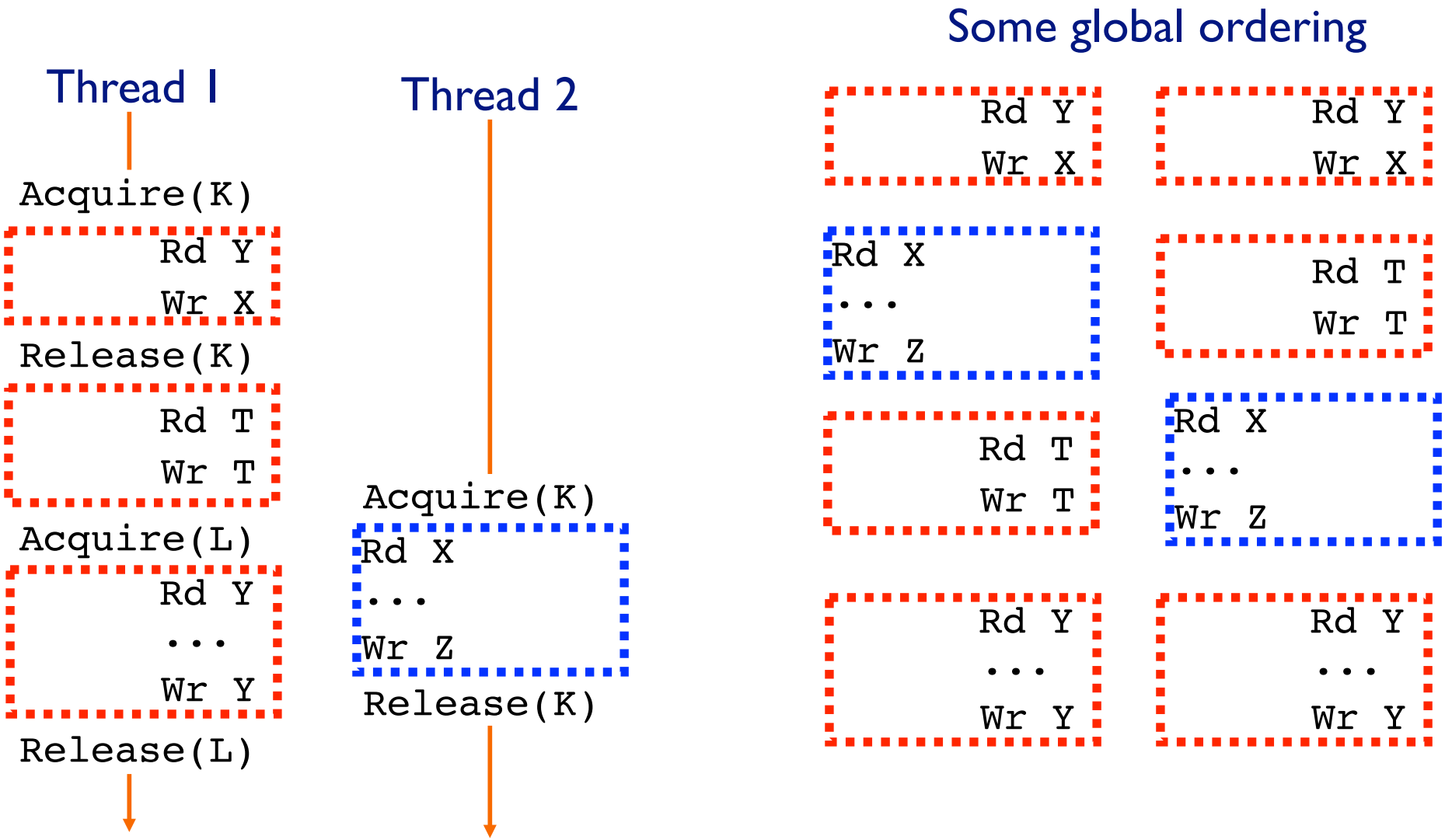
- How do they work?
  - `fence` insn must commit before any younger insn dispatches
    - This also means write buffer is emptied
  - Makes lock acquisition and release slow(er)
- **Use synchronization library, don't write your own**

Weak Ordering

- SC for data-race-free (properly synchronized) programs…
- …only **`acquires/releases`** must be strictly ordered
- Why? **`acquire-release`** pairs define **critical sections**
  - Between critical-sections: data is private
    - Globally unordered access OK
  - Within critical-section: access to shared data is exclusive
    - Globally unordered access also OK
  - Implication: compiler or dynamic scheduling is OK
    - As long as re-orderings do not cross synchronization points

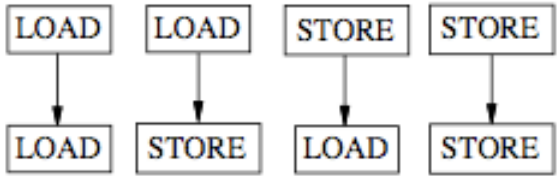
- **Weak Ordering (WO)**: Alpha, Itanium, ARM, PowerPC
  - ISA provides fence **fence** to indicate scheduling barriers
  - Proper use of fences is somewhat subtle
  - **Use synchronization library, don't write your own**

Sequential Consistency for DRF Example

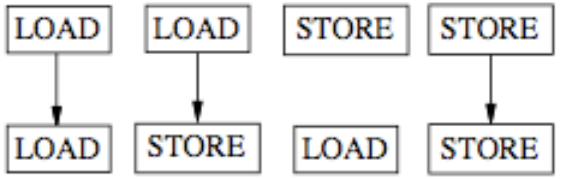


Another model: Release consistency

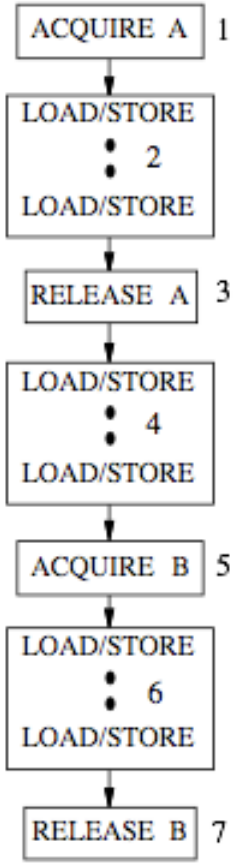
- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - Acquires: operations like lock
  - Release: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - all memory operations before release are complete
  - but accesses after release in program order do not have to wait for release
  - operations which follow release and which need to wait must be protected by an acquire



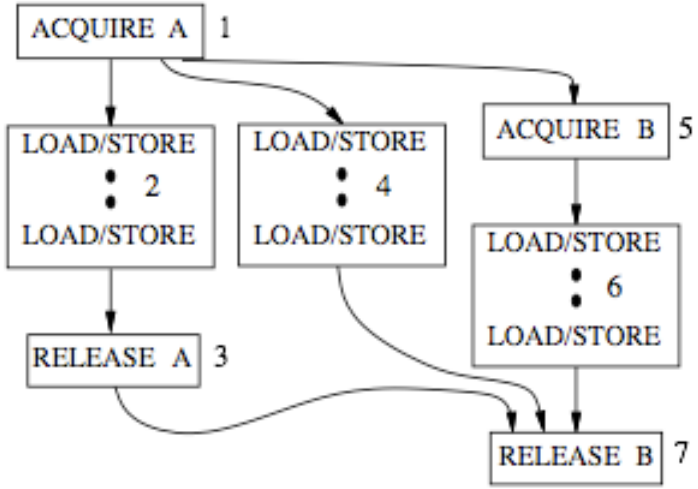
Sequential Consistency (SC)



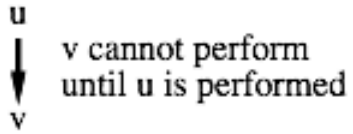
Processor Consistency (PC)



Weak Consistency (WC)



Release Consistency (RC)

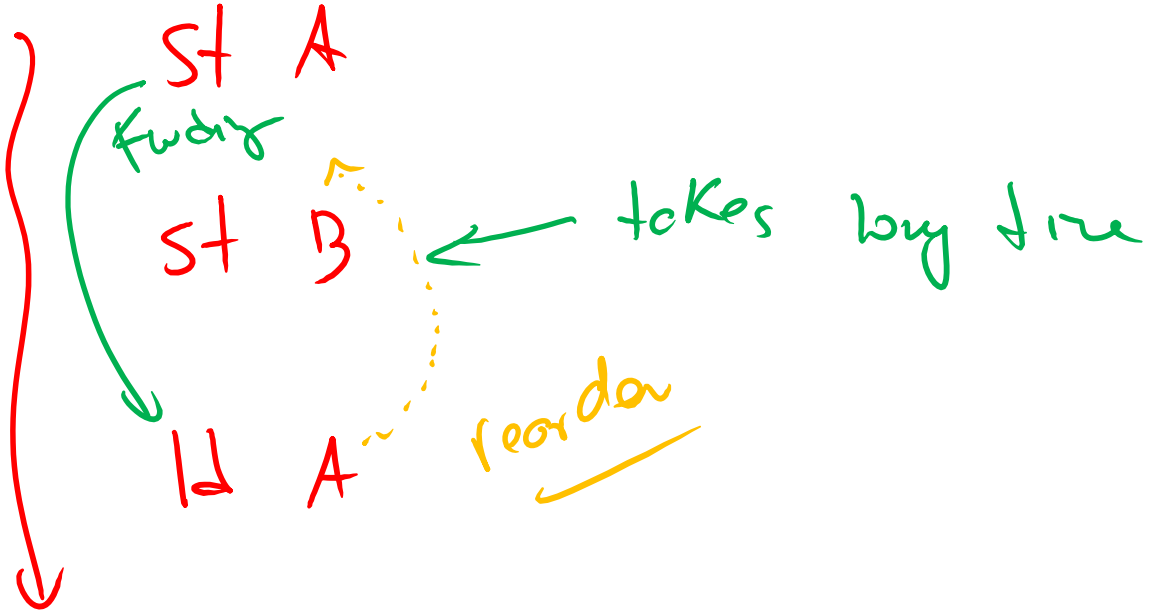


v cannot perform
until u is performed

LOADs and STOREs can perform in any order as long as local data and control dependences are observed

Processor Consistency

- What does st->ld relaxation buy?



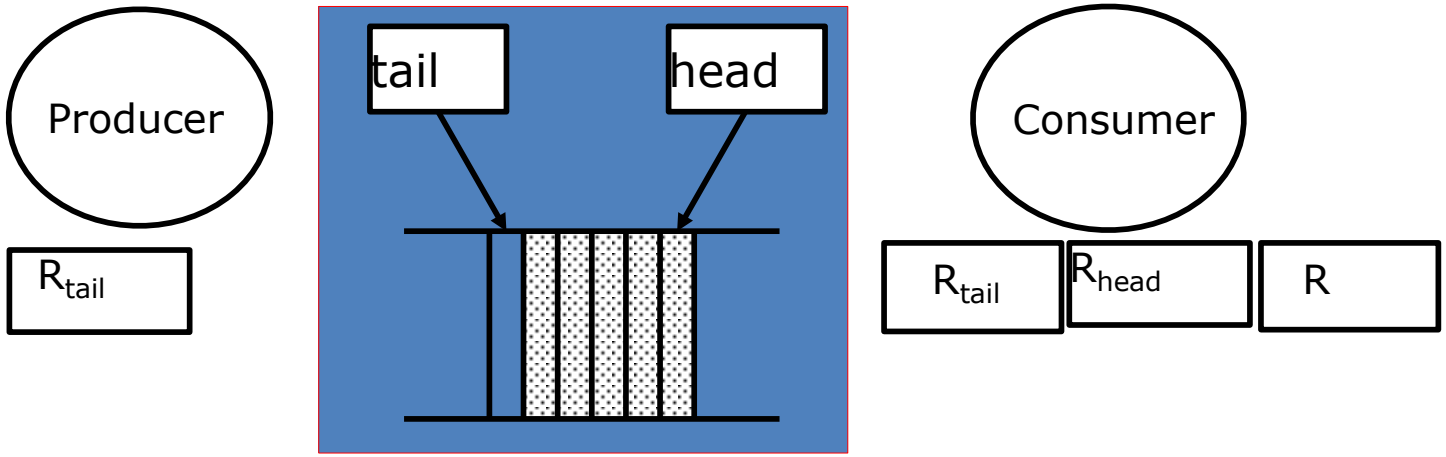
Why are fences expensive?

- How can you speed up fences?
  - (or other delays caused by the enforcement of a consistency model)

Tolerating Consistency Delays

- Prefetching
  - Make consistency-delayed accesses faster
- Speculation
  - overlap accesses speculatively
  - checkpoint state in program order
  - rollback in case processors interacted (race?)
  - Bottomline: if no one saw it, it never happened
- Current machinery in out-of-order processors?

Where should the fences go?



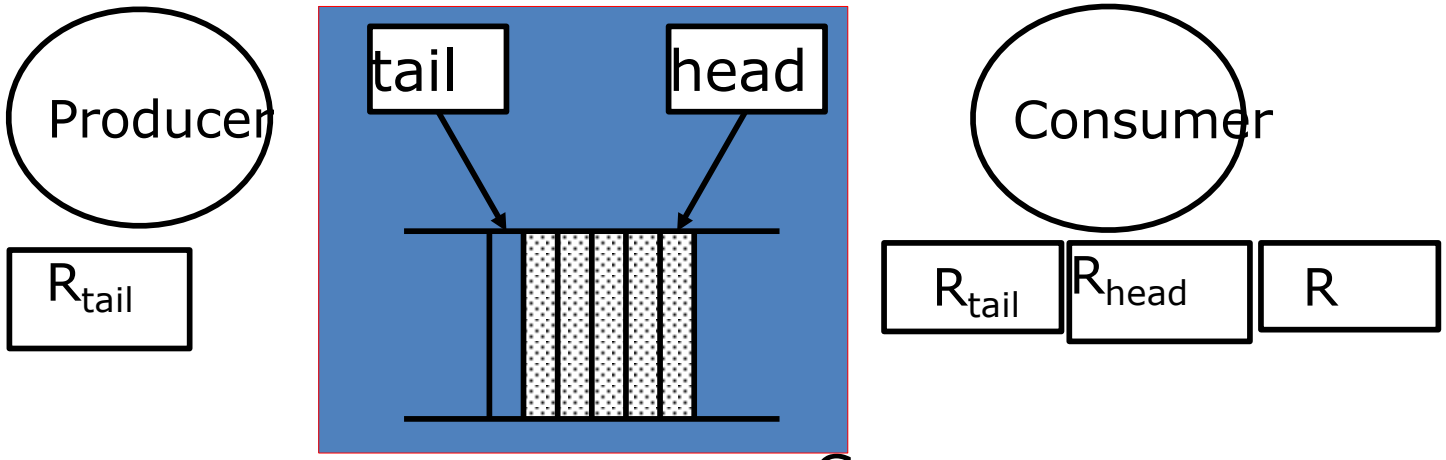
Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
R_{tail}=R_{tail}+1
Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)
spin: Load R_{tail}, (tail)
if R_{head}==R_{tail} goto spin
Load R, (R_{head})
R_{head}=R_{head}+1
Store (head), R_{head}
process(R)

Using Memory Fences



Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
fence
$R_{tail}=R_{tail}+1$
Store (tail), R_{tail}

ensures that tail ptr is not updated before x has been stored

Consumer:

Load R_{head}, (head)
spin: Load R_{tail}, (tail)
if $R_{head}==R_{tail}$ goto spin
fence
Load R, (R_{head})
$R_{head}=R_{head}+1$
Store (head), R_{head}
process(R)

ensures that R is not loaded before x has been stored

C/C++ Standard on Memory Model

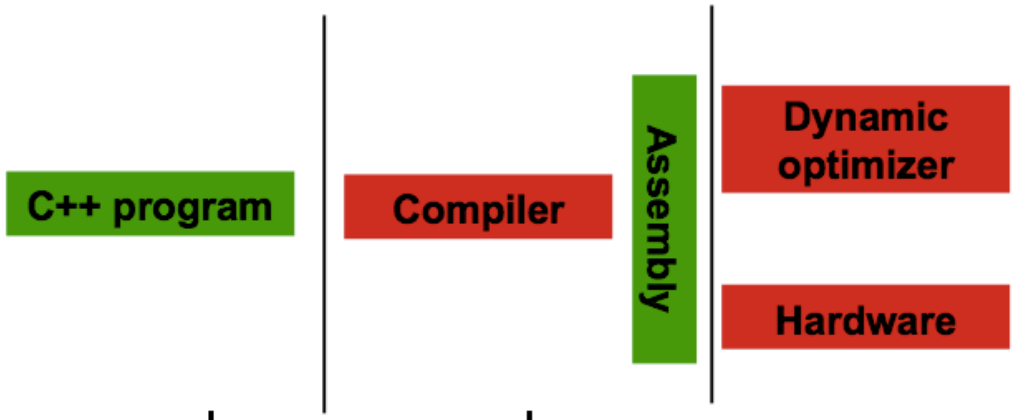
- Sequential Consistency...

- for **Data-Race Free** programs

C/C++ Standard on Memory Model

- What does that buy?
  - A *lot* of freedom to compiler and hardware
    - e.g., HW buffers, loop-inv code motion, CSE, etc.
  - Pretty much can do whatever reordering as long as it does not cross synchronization
- Key is to determine if there is a race...
  - **very** hard to do statically

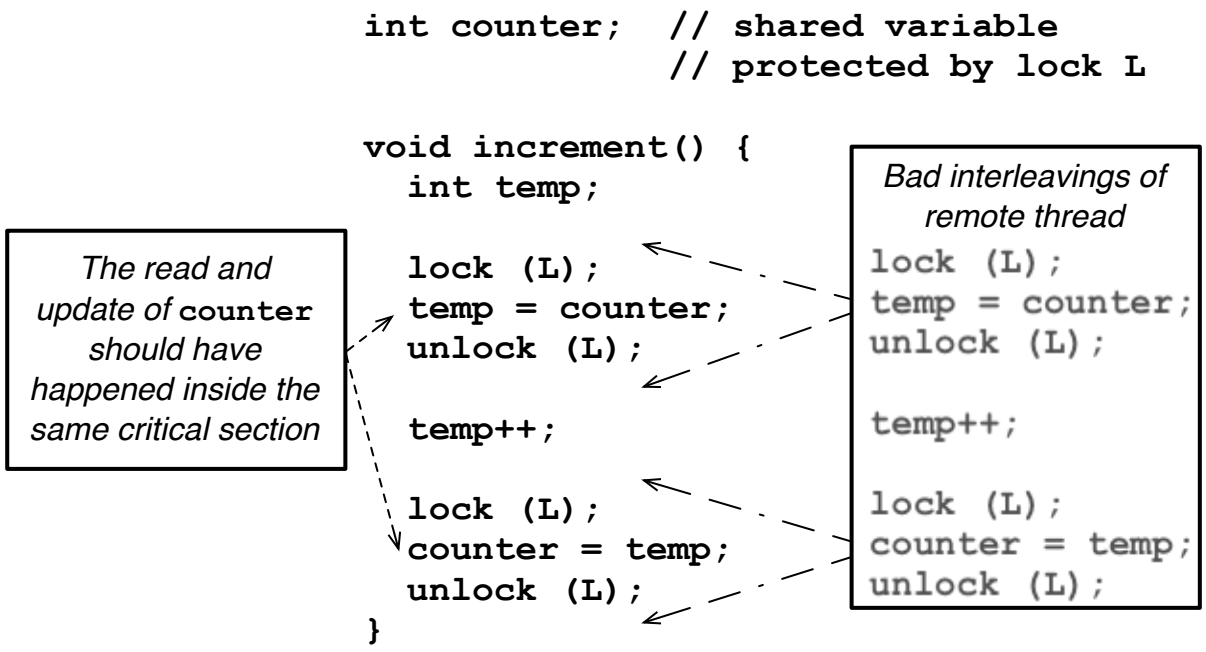
It is all about the interfaces



	C++ program	Compiler	Assembly	Dynamic optimizer
Language	*Strong* -reordering has to obey language model -not much room for reordering	*Strong* -reordering has to obey language model *-needs to insert fences to map language model to weaker ISA model* -not much room for reordering	*Weak* -room for reordering *-no worries about the HW reordering*	*Weak* -room for reordering
Compiler	*Strong* -reordering has to obey ISA or not be visible to SW	*Weak* -reordering may be visible to SW -room for reordering	*Strong* -reordering has to obey ISA or not be visible to SW	*Weak* -reordering may be visible to SW -room for reordering
Hardware				

Quick aside: General concurrency errors…

- Does free of data-races mean "correct" concurrency-wise?



» *Atomicity violations* aren't races necessarily

Memory Consistency in Real Systems

- **Processor consistency (PC)** (**x86**, SPARC)
  - Allows a in-order (FIFO) store buffer
    - Stores can be deferred, but must be put into the cache in order
- **Release consistency (RC)** (**ARM**, Itanium, **PowerPC**)
  - Allows an un-ordered coalescing store buffer
    - Stores can be put into cache in any order
  - Loads re-ordered, too.

Memory Consistency in GPUs/CPU+GPU

- Main issue
  - fences involve getting ACKs from whole system
  - GPUs are "scoped" to avoid excessing control communication
- What makes sense in this context?
  - Often explicit kernels operating on large data sets/regions
  - Explicit data partitioning/communication
  - Sync barriers are frequent
- Goal of a memory model in this context:
  - Provide reasonable semantics for system software
  - Enable optimizations
  - Avoid excessive hardware complexity
- Recent proposal from AMD: HRF, up next week

Implementing a Lock

- Shared counter/sum update example
  - Use a mutex variable for mutual exclusion
  - Only one processor can own the mutex
    - Many processors may call lock(), but only one will succeed (others block)
    - The winner updates the shared sum, then calls unlock() to release the mutex
    - Now one of the others gets it, etc.
  - But how do we implement a mutex?
    - As a shared variable (1 – owned, 0 –free)
- *How would you implement it?*

```
1. while (lock_var != 0);
2. lock_var = 1;
```

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - What invariant do we need?

```
1. while (lock_var != 0);
2. lock_var = 1;
```

Thread 1	Thread 2
Line1: lock_var == 0	
… descheduled …	Line 1: lock_var == 0
	Line 2: Sets lock_var = 1 (Thinks it has the lock.)
Line 2: Sets lock_var = 1 (Thinks it has the lock)	… descheduled …

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - Need a way to ***atomically*** see that the mutex is 0 ***and*** set it to 1
  - *How?*

Atomic Read-Update Instructions

- Atomic exchange instruction
  - E.g., EXCH R1,78(R2) will swap content of register R1 and mem location at address 78+R2
  - To acquire a mutex, 1 in R1 and EXCH
    - Then look at R1 and see whether mutex acquired
    - If R1 is 1, mutex was owned by somebody else and we will need to try again later
    - If R1 is 0, mutex was free and we set it to 1, which means we have acquired the mutex
- Other atomic read-and-update instructions
  - E.g., Test-and-Set

src = 1

```
xchg lock_var, src
```

If src == 0, you got the lock.

RISC Test-And-Set

- **`swap`**: a load and store in one insn is not very "RISC"
  - Broken up into micro-ops, but then how is it made atomic?
- "Load-link" / "store-conditional" pairs
  - Atomic load/store pair

```
label:
  load-link r1,0(&lock)
  // potentially other insns
  store-conditional r2,0(&lock)
  branch-not-zero label   // check for failure
```

  - On **`load-link`**, processor remembers address…
    - …And looks for writes by other processors
    - If write is detected, next **`store-conditional`** will fail
      - Sets failure condition
- Used by ARM, PowerPC, MIPS, Itanium

Using LL & SC

Swap R4 w/ 0(R1)

Atomic Exchange

```
swap:  mov R3, R4
       ll  R2,0(R1)
       sc  R3,0(R1)
       beqz   R3,swap
       mov R4,R2
```

Test if 0(R1) is zero, set to one

Atomic Test&Set

```
t&s:   mov R3,1
       ll  R2,0(R1)
       sc  R3,0(R1)
       bnez   R2,t&s
       beqz   R3,t&s
```

Atomic Add to Shared Variable

```
upd:   ll  R2,0(R1)
       add R3,R2,R4
       sc  R3,0(R1)
       beqz   R3,upd
```

Implementing Locks

- A simple swap (or test-and-set) works
  - But causes a lot of invalidations
    - Every write sends an invalidation
    - Most writes redundant (swap 1 with 1)
- More efficient: test-and-swap (or test-and-test-and-set ☺)
  - Read, do swap only if 0
    - Read of 0 does not guarantee success (not atomic)
    - But if 1 we have little chance of success
  - Write only when good chance we will succeed
- *Would either scale? What can we do?*

Large-Scale Systems: Locks

- Contention even with test-and-test-and-set
  - Every write goes to many, many spinning procs
  - Making everybody test less often reduces contention for high-contention locks but hurts for low-contention locks
  - Solution: exponential back-off
    - If we have waited for a long time, lock is probably high-contention
    - Every time we check and fail, double the time between checks
      - Fast low-contention locks (checks frequent at first)
      - Scalable high-contention locks (checks infrequent in long waits)
  - Special hardware support
- Queuing locks

Queue Locks

- Test-and-test-and-set locks can still perform poorly
  - If lock is contended for by many processors
  - Lock release by one processor, creates "free-for-all" by others
  - Interconnect gets swamped with `swap` requests
- **Software queue lock**
  - Each waiting processor spins on a different location (a queue)
  - When lock is released by one processor...
    - Only the next processors sees its location go "unlocked"
    - Others continue spinning locally, unaware lock was released
  - Effectively, passes lock from one processor to the next, in order
  + Greatly reduced network traffic (no mad rush for the lock)
  + Fairness (lock acquired in FIFO order)
  - Higher overhead in case of no contention (more instructions)
  - Poor performance if one thread is descheduled by O.S.