Advanced Computer Architecture
高级计算机体系结构

第11讲：Thread-Level Parallelism (3)

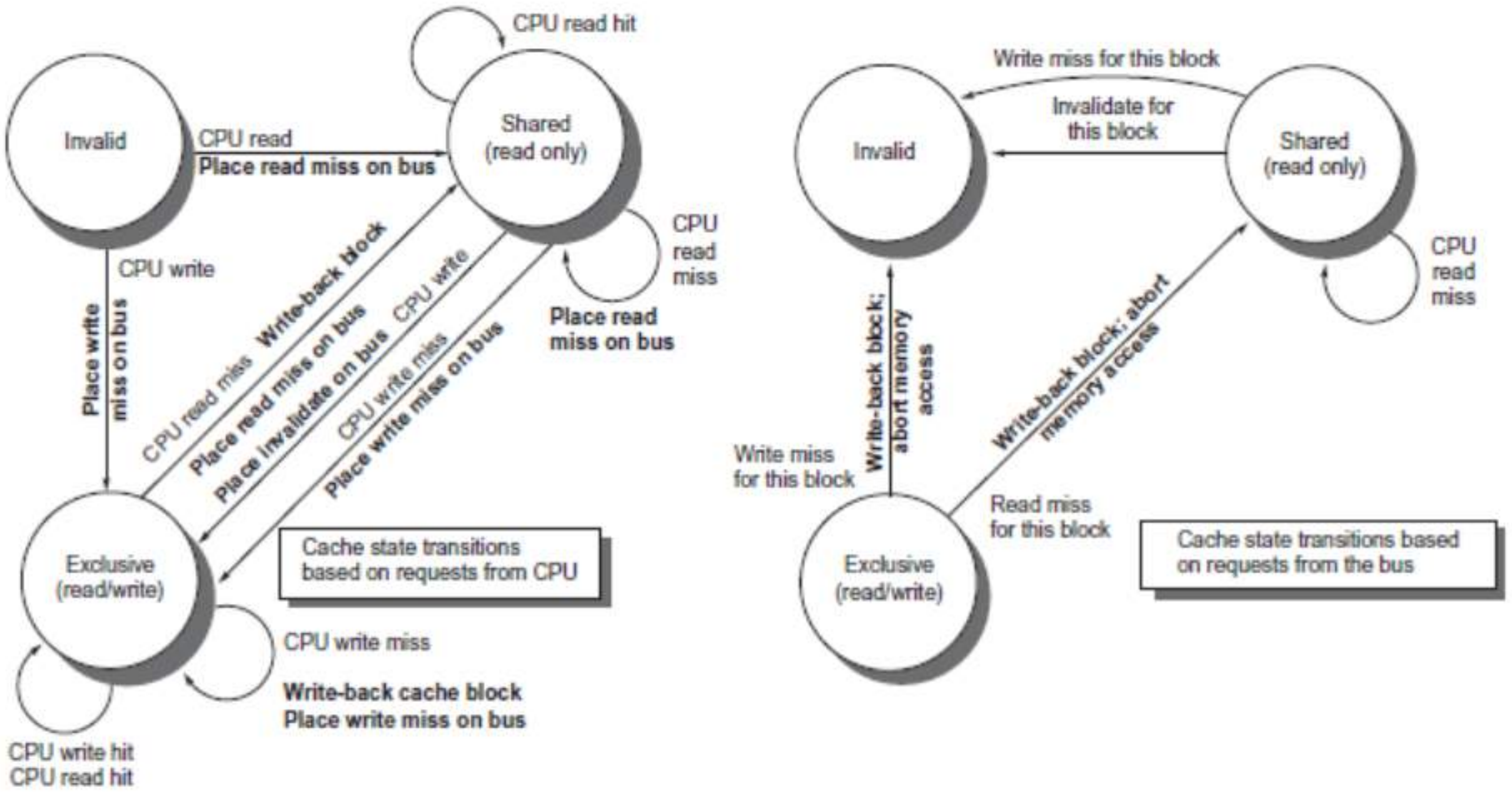
张献伟

xianweiz.github.io

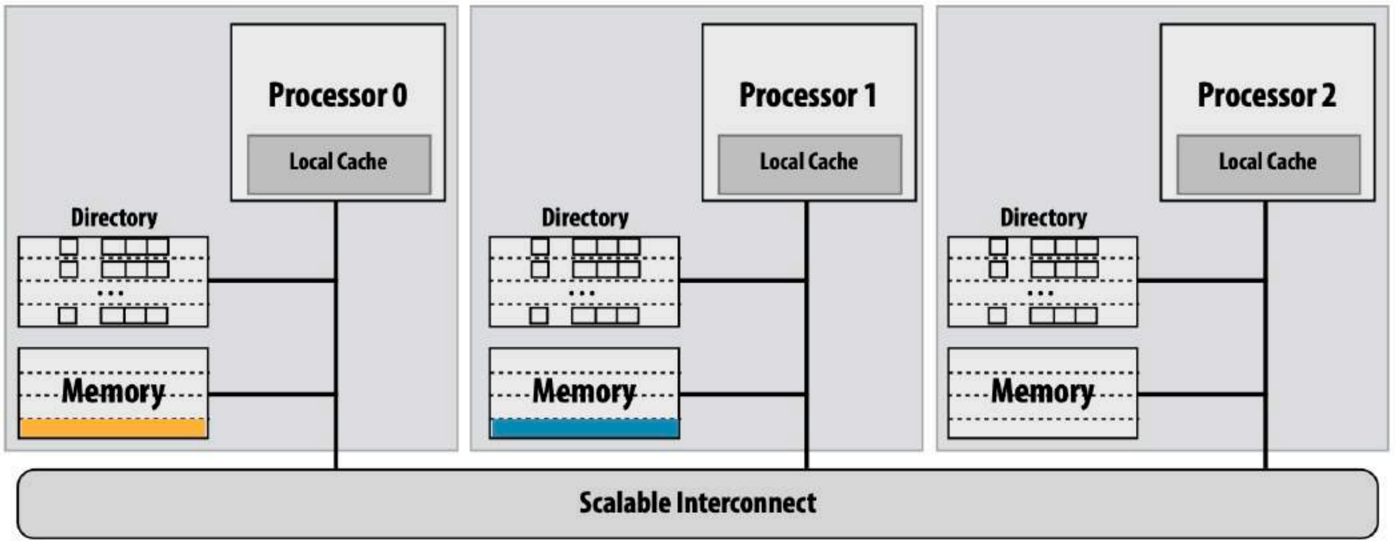
DCS5367, 12/14/2021

Review: Formal Specification

- Finite state transition diagram for a single private cache block[状态转换图]
  - Transitions based on processor and bus requests, respectively



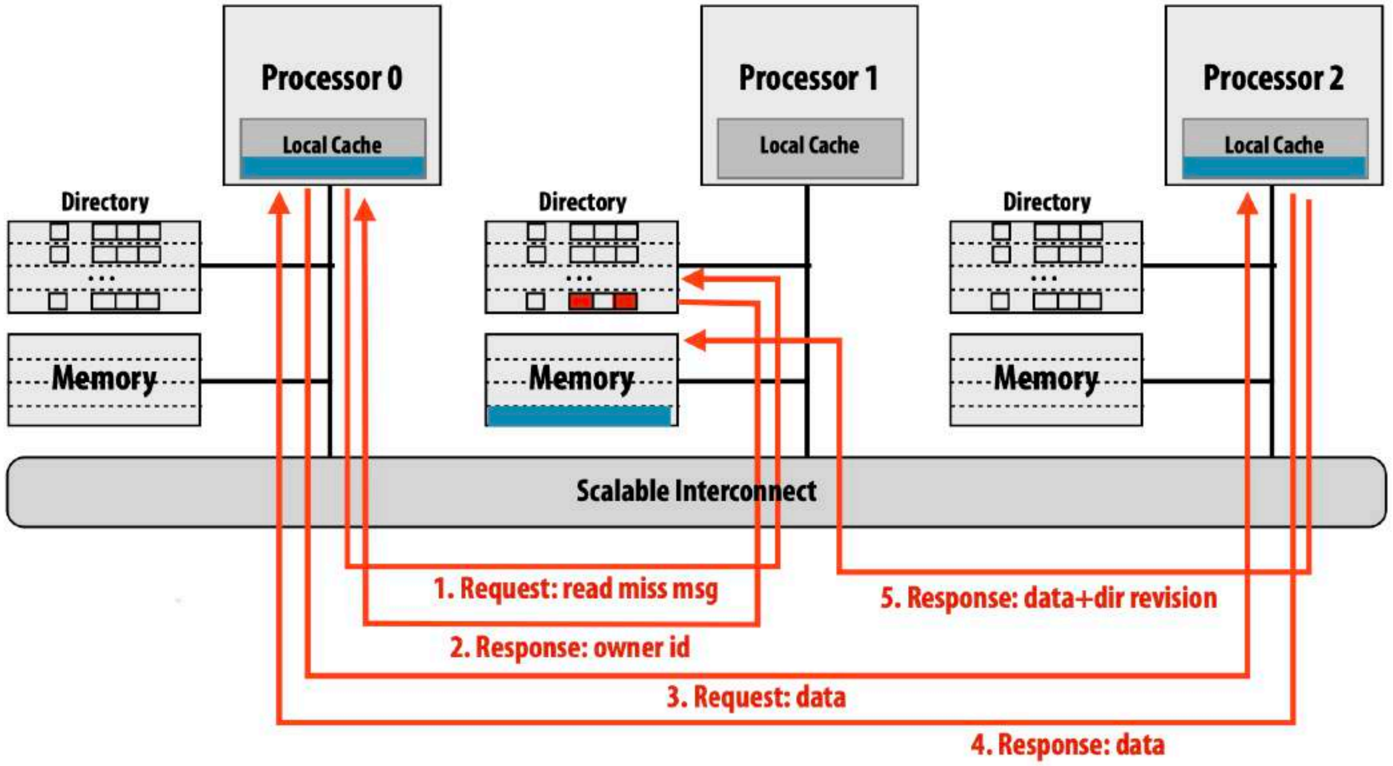
Review: Distributed Directory



- Directory partition is co-located with memory it describes
- "Home node" of a line: node with memory holding the corresponding data for the line
  - For example: node 0 is the home node of orange line, node 1 is the home node of blue node
- "Requesting node": node containing processor requesting line

Review: read miss to dirty line

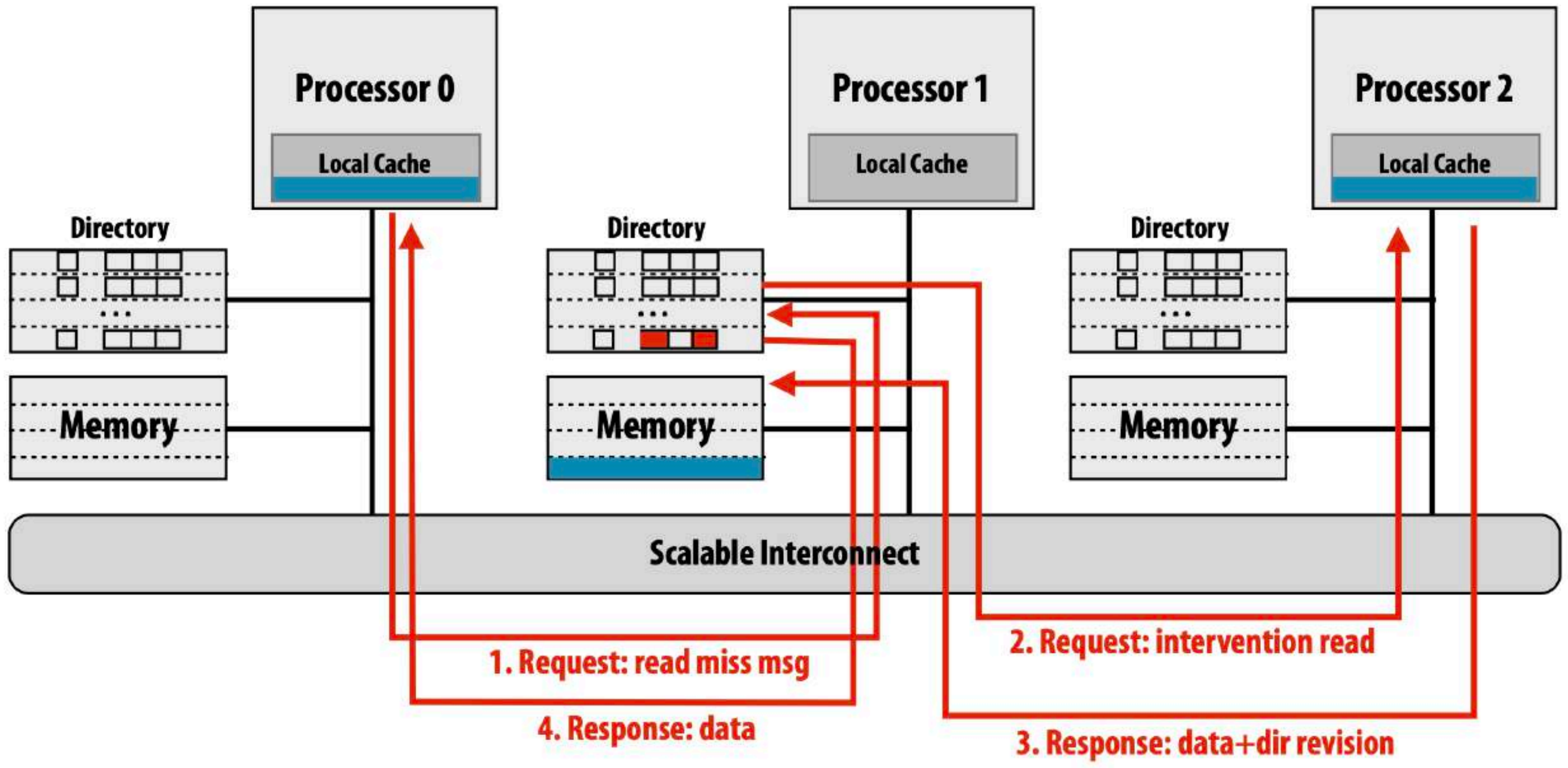
- Read from main memory by processor 0 of blue line
  - Dirty and its content is in P2's cache



http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

Review: Intervention Forwarding

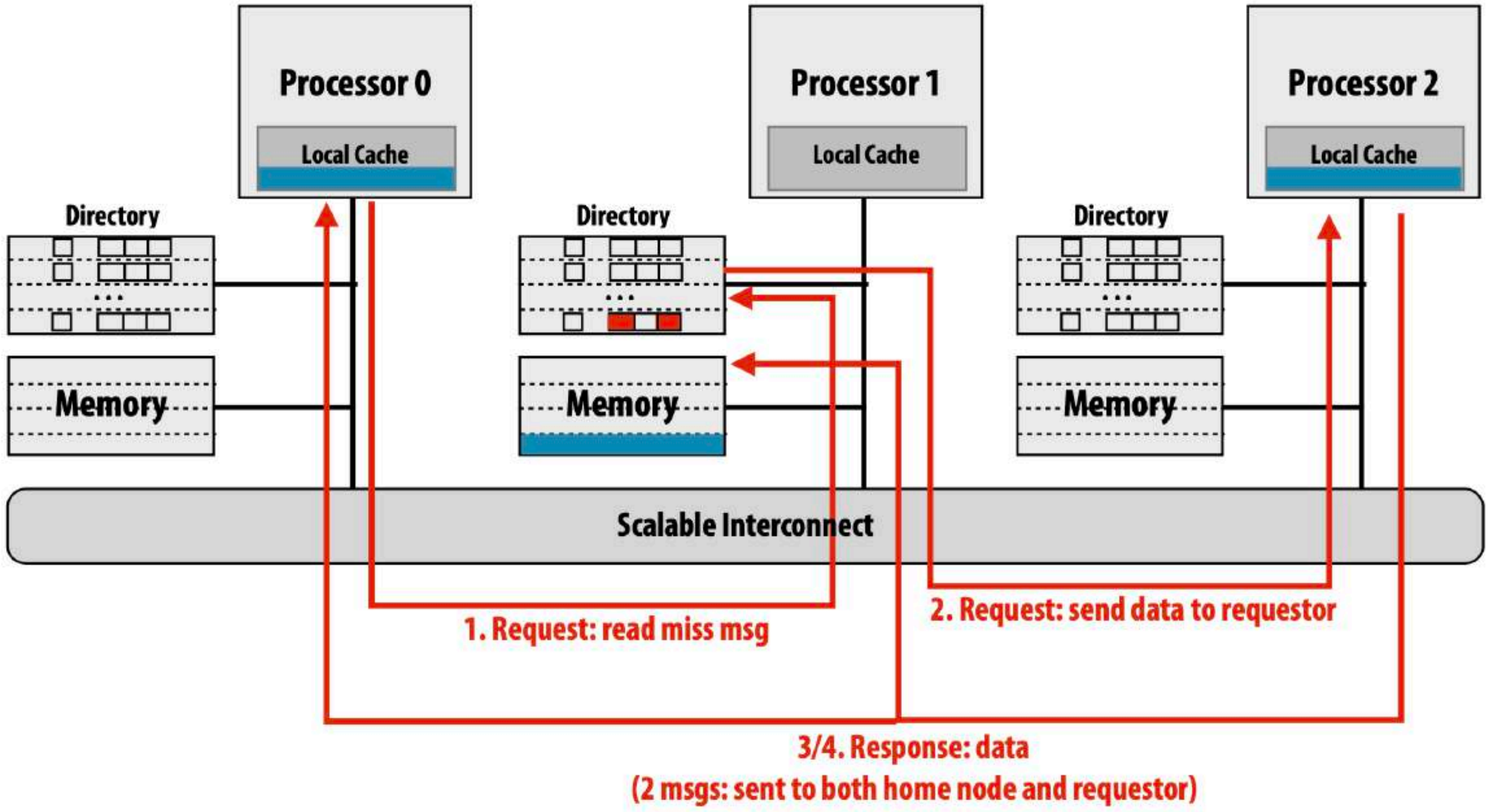
Read from main memory by P0 of the blue line: line is dirty (contained in P2's cache)



http://15418.courses.cs.cmu.edu/spring2017/lecture/directorycoherence

Review: Request Forwarding

Read from main memory by P0 of the blue line: line is dirty (contained in P2's cache)



Summary of Directory-base Coherence

- Primary observation: broadcast doesn’t scale, but we don’t need to broadcast to ensure coherence because often the number of caches containing a copy of a line is small
- Instead of snooping, just store the list of sharers in a directory and check the list when necessary
- One challenge
  - Use hierarchies of processors or larger cache size
  - Limited pointer schemes: exploit fact that most processors not sharing line
  - Sparse directory schemes: exploit fact that most lines not in line
- Another challenge
  - Reduce messages sent (traffic) and parallelize transactions (latency)

Example

- Assume that
  - Processes *P1* and *P2* are running on two different processors
  - Locations *A* and *B* are originally cached by both processors with the initial value of *0*
- If writes always take immediate effect and are immediately seen by other processors
  - Then impossible for both *IF* to be true

    Reaching the IF means that either A or B must have been assigned the value 1
- If write invalidate can be delayed, and the processor is allowed to continue during this delay
  - Then possible to that *P1* and *P2* haven't seen the invalidations before they attempt to read the values

```
    P1

    A = 1;
L1: if (B == 0) … …
```

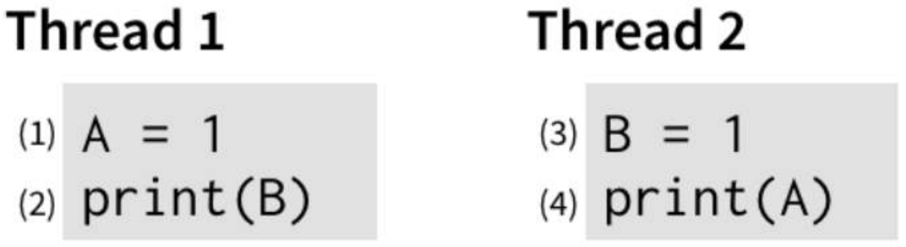
```
    P2

    B = 1;
L2: if (A == 0) … …
```

Coherence vs. Consistency[对比]

- **Cache coherence** defines requirements for the observed behavior of reads and writes to the same memory location
  - All processors must agree on the order of reads/writes to X
  - Goal: to ensure that the memory system in a parallel computer behaves as if the caches were not there
    - A system without caches would have no need for cache coherence
- **Memory consistency** defines the behavior of reads and writes to different locations
  - The allowed behavior of memory should be specified whether or not caches are present
  - Coherence only guarantees that writes to address X will eventually propagate to other processors
  - Consistency deals with when writes to X propagate to other processors, relative to reads and writes to other addresses

Memory Consistency[内存一致性]

- Memory consistency specifies the ordering behaviors
  - What ordering behavior should be allowed?
  - Under what conditions?
- Example: a program running two threads, where A and B are initially both 0. What this program can output?
  - 01: (1)(2)(3)(4) or (3)(4)(1)(2)
  - 11: (1)(3)(2)(4) or (1)(3)(4)(2)
  - 00: intuitively, it shouldn't be possible



Thread 1

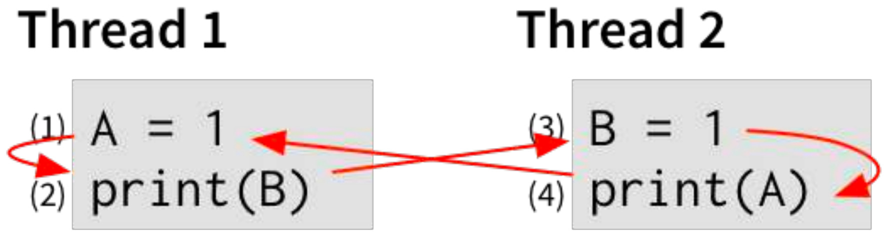
```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```

Example

- x → y: x must happen before y
  - (2) to print 0: (2) → (3)
  - (4) to print 0: (4) → (1)
  - If each thread's events happen in order
    - (1) → (2)
    - (3) → (4)
- Start from (1), follow the edges
  - (1) → (2) → (3) → (4) → (1)
  - (1) must happen before itself???



```
A = 1
print(B)
```

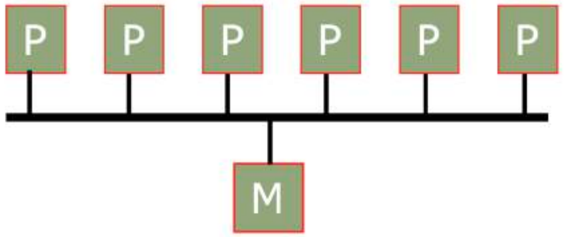
```
B = 1
print(A)
```

Memory Operation Ordering[访存先后]

- A program defines a sequence of loads and stores (this is the "program order" of the loads and stores)
- Four types of memory operation orderings
  - W→R: write to X must commit before subsequent read from Y
    - When a write comes before a read in program order, the write must commit (its results are visible) by the time the read occurs
  - R→R: read from X must commit before subsequent read from Y
  - R→W: read to X must commit before subsequent write to Y
  - W→W: write to X must commit before subsequent write to Y
- A <u>sequentially consistent</u> memory system maintains all four memory operation orderings
- Certain orderings can be violated ???

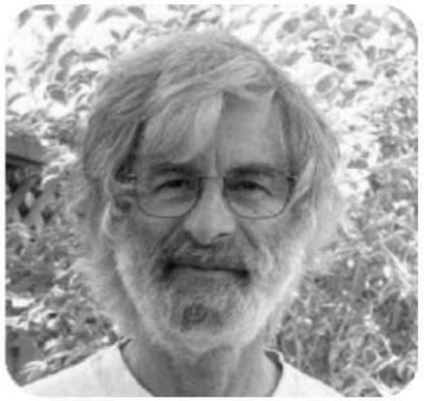
Sequential Consistency[顺序一致性]

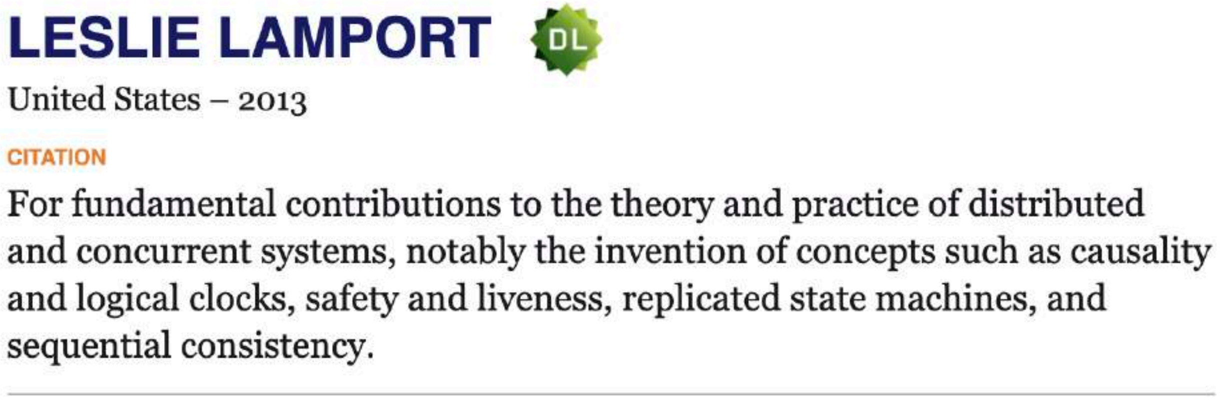
- The most straightforward model for memory consistency
  - Intuitive idea: multiple threads running in parallel are manipulating a single main memory, and so everything must happen in order
    - But what order?
  - Intuitive order: the events in a single thread happen in the order in which they were written
    - Intuitive to programmers
- **Sequential consistency** requires that the result of any execution be the same as though
  - Memory accesses executed by each proc. were kept in order
  - The accesses among different processors were arbitrarily interleaved



Sequential Consistency (cont.)

- Sequential consistency (SC)
  - Formalized by Leslie Lamport in 1979
  - "A system is **sequentially consistent** if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"
  - Defining SC is one of the many achievements that earned Lamport the Turing award in 2013
-





LESLIE LAMPORT

United States – 2013

CITATION

For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.

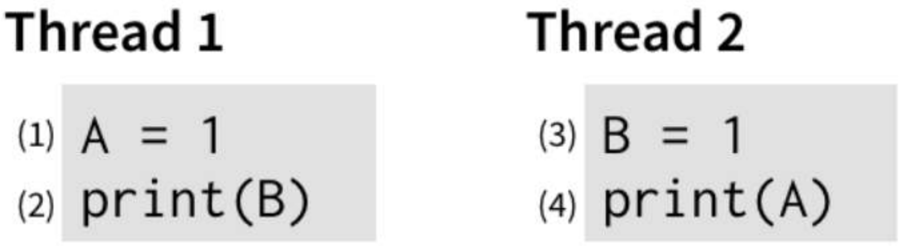
http://csg.csail.mit.edu/6.823S14/lectures/L22.pdf

The Examples

- With SC,
- Example-1:
  - Must delay the read of A or B (A == 0 or B == 0) until the previous write has completed (B = 1 or A = 1)
  - Cannot simply place the write in a buffer and continue with the read
- Example-2:
  - Print(B)/Print(A) cannot happen before A = 1/B = 1
    - 00 cannot be printed

```
    P1

    A = 1;
L1: If (B == 0) … …
```

```
    P2

    B = 1;
L2: If (A == 0) … …
```



Thread 1

```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```

Memory Consistency Model[一致性模型]

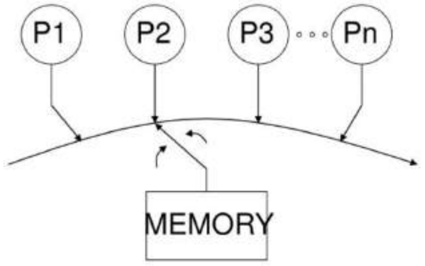
- Memory consistency model (or just “memory model”) defines the allowed orderings of multiple threads on a multiprocessor
  - E.g., orderings that print 01/11 are allowed, but not 00

- A memory consistency model is a contract between the hw and sw
  - The hw promises to only reorder operations in ways allowed by the model
  - In return, the sw acknowledges that all such reorderings are possible and that is needs to account for them

https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Issues of SC[问题]

- SC: just like a switch to select thread to run, and runs its next event completely
  - Events happen in program order



- SC presents a simple programming paradigm
- But, SC reduces potential performance
  - Especially in a multiprocessor with a large number of processors or long interconnect delays
- Simplest way to implement SC
  - A processor delays the completion of any memory access until all the invalidations caused by that access are completed
  - Example: for a write miss, four processors share a block
    - 170 cycles for write: 50 cycles to establish ownership, then 10 cycles to issue each invalidate, and 80 cycles for an invalidate to complete and be acknowledged (50 + 40 + 80)

Optimizations[优化]

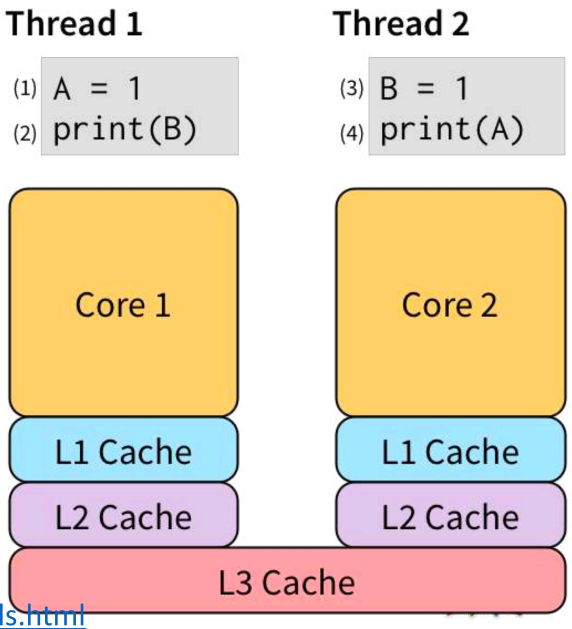
- **Goal**: develop a model that is simple to explain and yet allows a high performance implementation

- **Solution-1**: develop ambitious implementations that preserve SC but use latency-hiding techniques to reduce the penalty

- **Solution-2**: develop less restrictive memory consistency models that allow for faster hw
  - Such models can affect how the programmer sees the multiprocessor

The Example

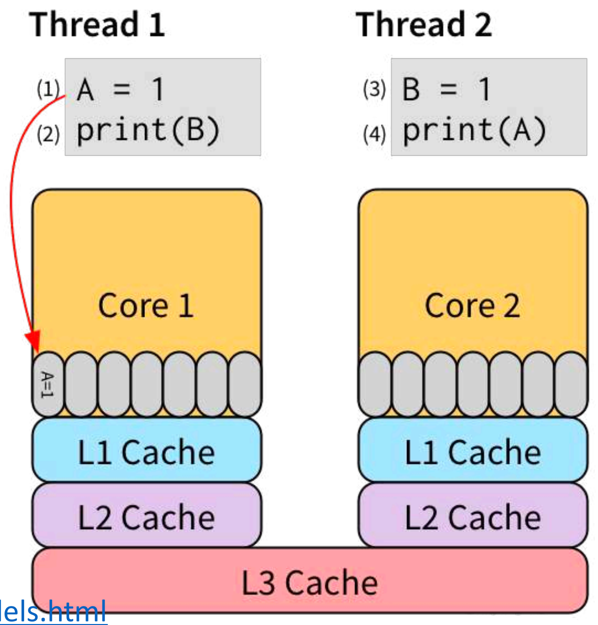
- SC maintains a single view of memory
  - Cannot run (2) until (1) has become visible to every other thread
- No reason why (2) needs to wait until (1) completes
  - (2): a read from B, (1): a write to A
  - They don't interfere with each other at all
    - So should be allowed to run in parallel
  - Note that event (1) is very slow
    - A very high overhead
- SC greatly hurts performance
  - The model should be relaxed!!!
    - Event(2) should not wait for (1)



https://www.cs.utexas.edu/~bornholt/post/memory-models.html

The Example (cont.)

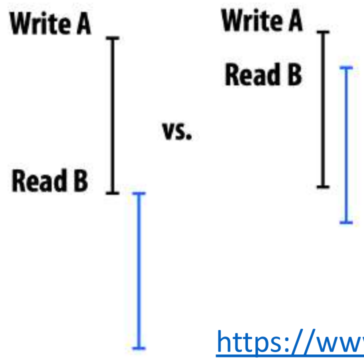
- Place write(1) into a store buffer, rather than waiting for it to become visible
  - Then (2) could start immediately, rather than waiting for (1) to reach the L3
  - The store buffer is on core: very fast to access
  - At some time in the future, the cache hierarchy will pull the write from the store buffer and propagate it through the L3 so that it becomes visible to other threads
- The buffer helps hide the write latency
- Preserves single-threaded behavior
  - Access: store buffer → memory

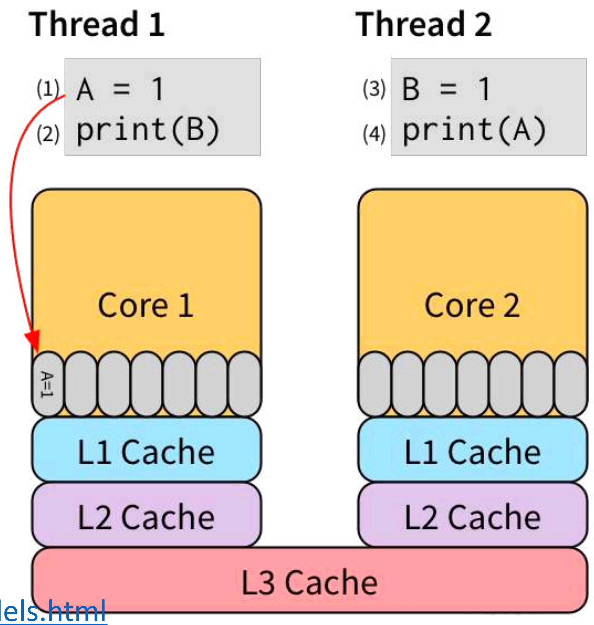


https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Total Store Ordering[TSO一致性]

- TSO mostly preserves the same guarantees as SC, except that it allows the use of store buffers
  - There buffers hide write latency, making execution significantly faster
- Retains ordering among writes (that's why called 'total store ordering')
  - Relaxed only the W→R ordering
- Performance gain
  - Allow processor to hide latency of writes when later read is independent

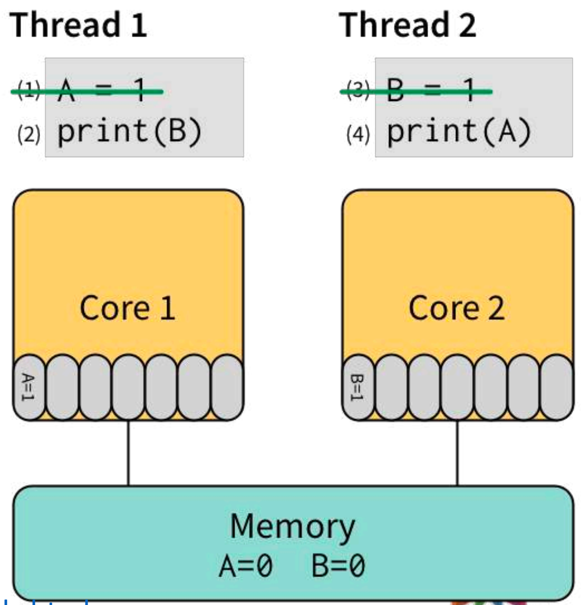




https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Total Store Ordering (cont.)

- While boosting performance, TSO allows behaviors that SC does not
  - I.e., programs running on TSO hw can exhibit behavior that programmers would find suprising
- The example: both threads first check their local store buffer, but fails to locate and then fetches from memory
  - This program can print 00
  - TSO cannot put into practices???
- Actually, every modern architecture includes a store buffer, and so has a memory model at least as weak as TSO



https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Partial Store Ordering[PSO一致性]

- In TSO, only W→R order is relaxed
  - The W→W constraint still exists. Writes by the same thread are not reordered (they occur in program order)
- In partial store ordering (PSO), W → W is also relaxed

- Example: A and flag are initially 0s
  - SC: print ’1’
  - TSO: print ‘1’
  - PSO: may print ‘0’

Thread 1 (on P1)	Thread 2 (on P2)

A = 1;
flag = 1;
	while (flag == 0);
	print A;

Aggressive Memory Ordering???

- SC maintains all four memory operation orderings
- Certain orderings can be violated ???
  - W→R: store buffer to allow read execute earlier
  - W→W: reorder writes in the store buffer
    - Earlier write is a cache miss, later is a hit
  - R→W, R→R: processor may reorder independent instructions
    - Out-of-order execution
  - Note that all are valid optimizations if a program consists of a single instruction stream

- What if we allow all four memory orderings?
  - Still a memory consistency model (Release Consistency)

http://15418.courses.cs.cmu.edu/tsinghua2017content/lectures/12_consistency/12_consistency_slides.pdf

Release Consistency[RC一致性]

- Release Consistency (RC)
  - Processors support special synchronization operations
  - Memory accesses before memory fence instruction must complete before the fence issues
  - Memory accesses after fence cannot begin until fence instruction is complete

```
reorderable reads and writes here
...
MEMORY FENCE
...
reorderable reads and writes here
...
MEMORY FENCE
```

Express Synchronization[同步]

- ’00’ is not allowed in SC (the example)
- All modern architectures include **synchronization** operations to bring their relaxed memory models under control when necessary
  - Most common operation: barrier (or fence)
- A barrier inst forces all memory operations before it to complete before any memory operation after it can begin
  - I.e., a barrier inst effectively reinstates SC at a particular point in program execution

Thread 1

```
(1) A = 1
(2) print(B)
```

Thread 2

```
(3) B = 1
(4) print(A)
```

```
S1: Store x = NEW;  | S2: Store y = NEW;
FENCE               | FENCE
L1: Load r1 = y;    | L2: Load r2 = x;
```

Summary: Relaxed Consistency

- Motivation: obtain higher performance by allowing recording of memory operations (reordering is not allowed by sequential consistency)
  - Relaxed consistency models differ in which memory ordering constraints they ignore
- One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific operation orderings when needed
  - Optimize for the common case: most memory accesses are not conflicting, so don't design a system that pays the cost as if they are
  - But in practice complexities encapsulated in libraries that provide intuitive primitives like lock/unlock, barrier (or lower level primitives like fence)

Synchronized Programs[同步程序]

- Two memory accesses by different processors conflict if
  - They access the same memory location
  - At least one is a write
- Unsynchronized program
  - Conflicting accesses not ordered by synchronization (e.g., a fence, operation with release/acquire semantics, barrier, etc.)
  - Unsynchronized programs contain data races: the output of the program depends on relative speed of processors (non-deterministic program results)
- In practice, most programs are synchronized (via locks, barriers, etc. implemented in synchronization libraries)
- Synchronized programs yield SC results on non-SC systems
  - Synchronized programs are data-race-free

Locks[锁]

- A lock surrounding the data/code ensures that only one program can be in a critical section at a time
  - Lock algorithms assume an underlying cache coherence mechanism – when a process updates a lock, other processes will eventually see the update
- The hardware must provide some basic primitives that allow us to construct locks with different properties

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Hardware Primitives[硬件元语]

- Basic hardware primitives: a set of hw primitives with the ability to atomically read and modify a memory location
  - The basic building blocks that are used to build a wide variety of user-level sync operations, including things such as locks and barriers
  - The primitives will be used by system programmers to build a sync library, a process that is often complex and tricky

- One typical operation for building synchronization operations is the atomic exchange
  - Interchanges a value in a register for a value in memory

Atomic Exchange[原子交换]

- To build a simple lock (0: lock free, 1: lock unavailable)
  - A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock
- Exchange inst returns value:
  - 1: if some other processor had ow already claimed access
  - 0: no one has claimed. But it is also changed to 1, preventing any competing exchange from also retrieving a 0
- Key to using the exchange primitive to implement sync is that the operation is atomic
  - The exchange is indivisible, and two simultaneous exchanges will be ordered

Other Primitives[其他]

- Other atomic primitives
  - Test-and-set: tests a value and sets it if the value passes test
    - Example: test for 0 and set the value to 1 (similar to atomic exchange)
  - Fetch-and-increment: fetches the value of a memory location and atomically increments it
    - Example: using value 0 to indicate that the sync variable is unclaimed (just as exchange)

- Key property of all atomic primitives
  - Read and update a memory value in such a manner that we can tell whether the two operations executed atomically

Atomic Primitives[原子性]

- How to guarantee the atomic primitive?
  - Option-1: implement a single atomic memory operation
  - Option-2: have a pair of instructions where the 2nd inst returns a value from which it can be deduced whether the pair of insts was executed as though the insts were atomic
- Option-1: single
  - Requires both a memory read and write in uninterruptable instruction
    - Complicates the coherence: hw cannot allow any other operations between the read and the write, and yet must not deadlock
- Option-2: pair
  - Effectively atomic if it appears as through all other operations executed by any processor occurred before or after the pair
    - No other processor can change the value between the inst pair

Atomic Pair[原子对]

- Used in the MIPS processor and RISC-V
- RISC-V: load reserved/store conditional
  - Load reserved (lr): loads the contents of memory given by rs1 into rd and creates a reservation on that memory address
    - Also called load linked or load locked
  - Store conditional (sc): stores the value in rs2 into the memory address given by rs1
- Store conditional fails (writes a non-zero) if,
  - The reservation of the load is broken by a write to the same memory location
  - The processor does a context switch between the instructions

Atomic Pair (cont.)

- Instructions lr/sc are used in sequence
  - lr returns the initial value
  - sc returns 0 only if it succeeds
- Example: use lr/sc to implement an atomic exchange on the memory location specified by the contents of x1 with the value in x4
  - Anytime a processor intervenes and modifies the value in memory between the lr and sc, the sc returns a non-zero, causing the code sequence to try again

```
try: mov x3,x4       ;mov exchange value
     lr x2,x1        ;load reserved from
     sc x3,0(x1)     ;store conditional
     bnez x3,try     ;branch store fails
     mov x4,x2       ;put load value in x4?
```

Atomic exchange

```
try: lr x2,x1        ;load reserved 0(x1)
     addi x3,x2,1    ;increment
     sc x3,0(x1)     ;store conditional
     bnez x3,try     ;branch store fails
```

Atomic fetch-and-increment

Implementing Locks[实现锁]

- Once we have an atomic operation, we can use the coherence mechanism of a multiprocessor to implement spin locks
  - Locks that a processor continuously tries to acquire, spinning around a loop until it succeeds
- Simplest implementation of keeping the lock variables in memory (there were no cache coherence)
- Cache the locks using the coherence mechanism to maintain the lock value coherently

```
        addi x2,R0,#1
lockit: EXCH x2,0(x1)     ;atomic exchange
        bnez x2,locket    ;already locked?
```

```
lockit: ld x2,0(x1)       ;load of lock
        bnez x2,locket    ;not available-spin
        addi x2,R0,#1     ;load locked value
        EXCH x2,0(x1)     ;swap
        bnez x2,locket    ;branch if lock wasn't 0
```

Coherence-based Locks[基于一致性]

- Spins by doing reads on a local copy of the lock until it successfully sees that the lock is available
- Then, attempts to acquire the lock by doing a swap operation
  - All processes use a swap inst that reads the old value and stores 1 into the lock variable
  - The single winner will see the 0, and the losers will see a 1 that was placed there by the winner
  - The winning processor executes the code after the lock and, when finished, stores a 0 into the lock variable to release the lock

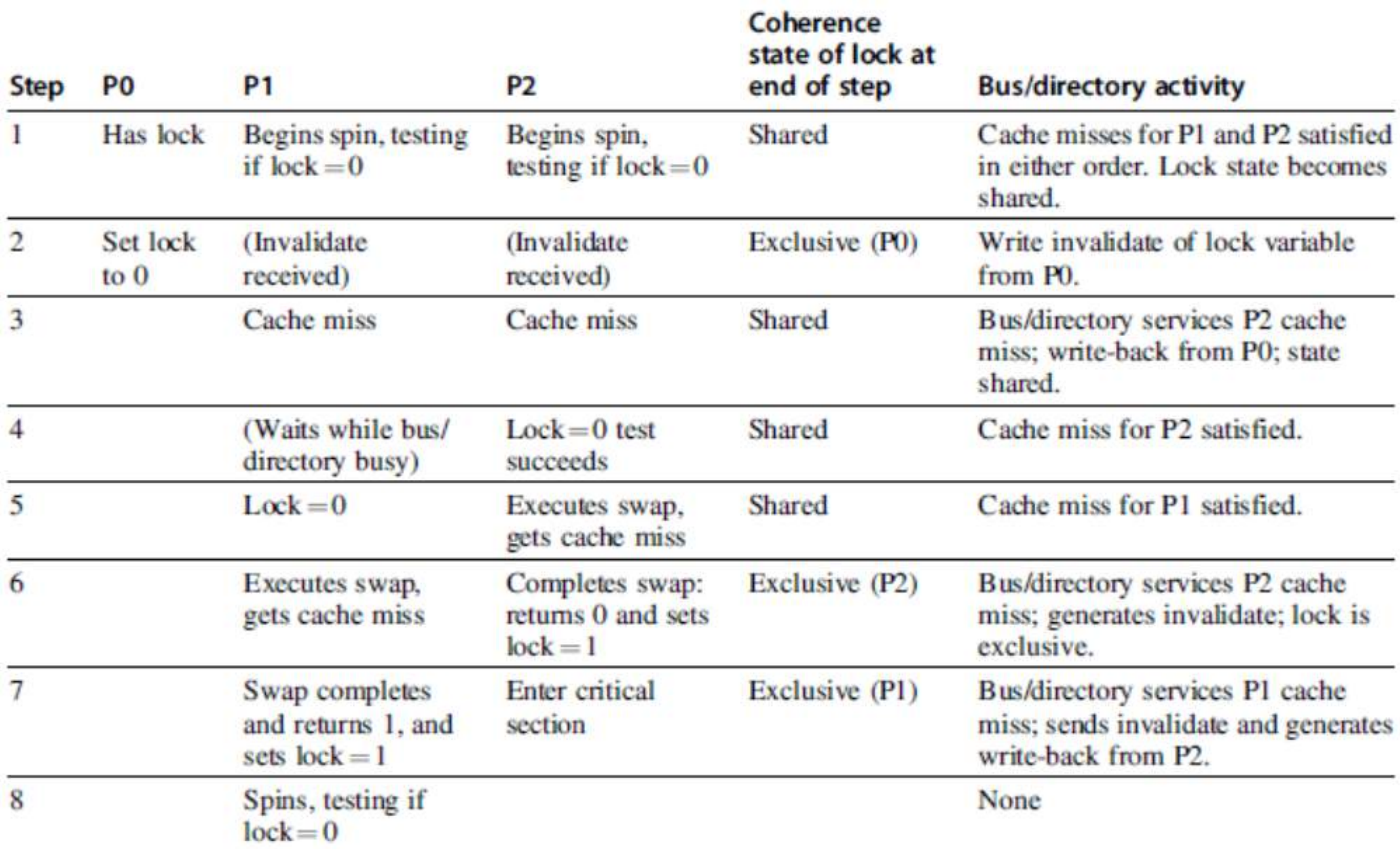
```
lockit: ld x2,0(x1)         ;load of lock
        bnez x2,locket      ;not available-spin
        addi x2,R0,#1       ;load locked value
        EXCH x2,0(x1)       ;swap
        bnez x2,locket      ;branch if lock wasn't 0
```

Coherence-based Locks (cont.)

- To lock a variable using an atomic swap
  - Once the processor with the lock stores a 0 into the lock (i.e., lock released), all other caches are invalidated and must fetch the new value to update their copy of the lock[锁释放->竞争]
  - One such cache gets the copy of the unlocked value (0) first and performs the swap[赢得竞争]
  - When the cache miss of other processors is satisfied, they find that the variable is already locked, so they must return to testing and spinning[继续]

```
lockit: ld x2,0(x1)         ;load of lock
        bnez x2,locket      ;not available-spin
        addi x2,R0,#1       ;load locked value
        EXCH x2,0(x1)       ;swap
        bnez x2,locket      ;branch if lock wasn't 0
```

Coherence-based Locks (cont.)



Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied.
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied.
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Languages' Memory Models[语言内存模型]

- Besides hardware, compilers can also reorder memory operations
  - Example: the program always prints a string of 100 '1's
  - Possible to optimize the code?
    - Loop-invariant code motion: move the write outside the loop
    - Dead store elimination: remove X = 0
  - These two programs are totally equivalent
    - Produce the same output

```
X = 0
for i in range(100):
   X = 1
   print X
```

```
X = 1
for i in range(100):
   print X
```

Languages' Memory Models (cont.)

- Now suppose there's another thread running in parallel with the program, and it performs a single write to X
  - The first program
    - It can print strings like 11101111 …, so long as there's only one single zero (because it will reset X = 1 on the next iteration)
  - The second program
    - It can print strings like 1110000 …, where once it starts printing 0s it never goes back to 1s
  - The first can never print 1110000…; the second cannot print 11011111…

With parallelism, the compiler optimization no longer produces an equivalent program.

```
X = 0
for i in range(100):
  X = 1
  print X
```

```
X = 1
for i in range(100):
  print X
```

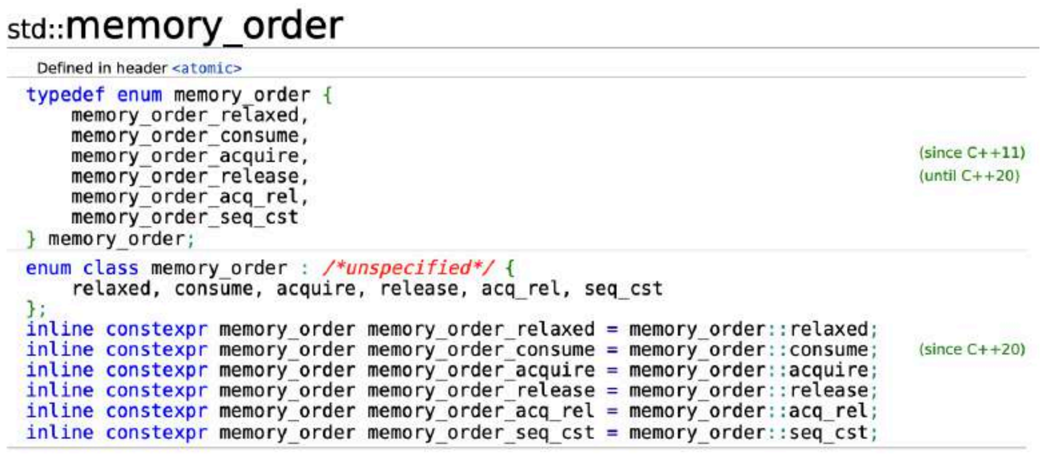
```
X = 0
```

```
X = 0
```

https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Languages' Memory Models (cont.)

- Memory consistency at the program level
- The compiler optimization is effectively reordering
  - It's rearranging (and removing some) memory accesses in ways that may or may not be visible to programmers
- To preserve intuitive behavior, programming languages need memory models of their own,
  - To provide a contract to programmers about how their memory operations will be reordered



std::memory_order

Defined in header <atomic>

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,                                   (since C++11)
    memory_order_release,                                   (until C++20)
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
enum class memory_order : /*unspecified*/ {
    relaxed, consume, acquire, release, acq_rel, seq_cst
};
inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
inline constexpr memory_order memory_order_consume = memory_order::consume;     (since C++20)
inline constexpr memory_order memory_order_acquire = memory_order::acquire;
inline constexpr memory_order memory_order_release = memory_order::release;
inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
```

https://www.cs.utexas.edu/~bornholt/post/memory-models.html

Summary

- Multiprocessors with thread-level parallelism
  - Sharing memory, having private caches
- Cache coherence
  - Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary
  - Directory-based: A single location (directory) keeps track of the sharing status of a block of memory
- Memory consistency
  - Sequential consistency: maintains all four memory operation orderings (W→R, R→R, R→W, W→W)
  - Relaxed consistency: allows certain orderings to be violated
    - TSO, PSO, RC