Cache Coherence and Memory Consistency

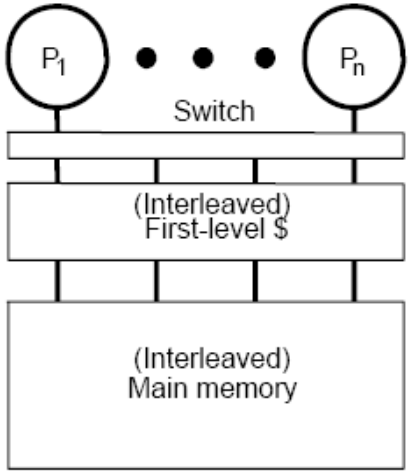
What is this?

- In shared-memory (uniform access and non-uniform access) and distributed-shared-memory systems, memory locations are “cached” locally by processors for rapid access.

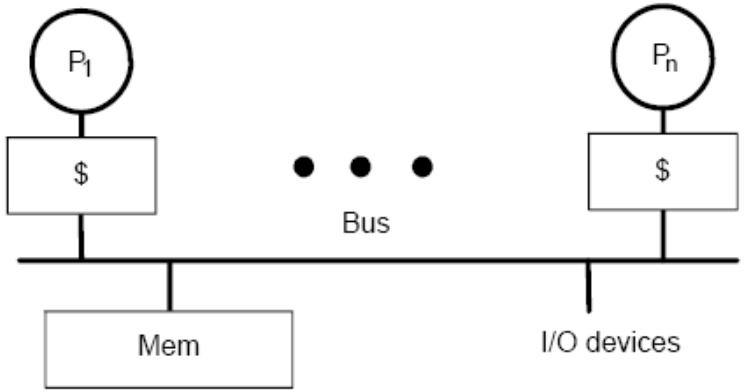
- Caching leads to the possibility that cached copies of memory information are not in synch with each other (are not **coherent**).

- A consistent shared memory state requires of eliminating this possible incoherence.

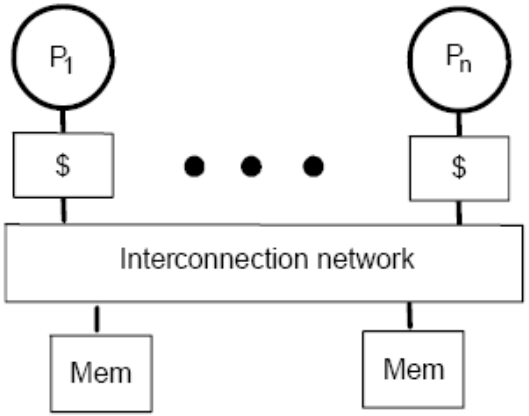
Shared Memory hardware organizations ($ = “cache”)



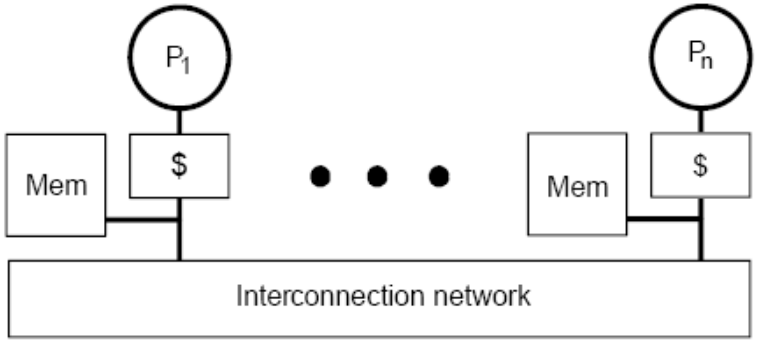
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall



(d) Distributed-memory

Example

- church.cs.hmc.edu is a ccNUMA.
- Uniform global address space.

- 4 x 12-core AMD Opteron 6168 (1.9GHz, 512KB/Core L2 Cache, 12MB L3 Cache)

- RAM: 64GB (16 x 4GB) Operating at 1333MHz Max (DDR3-1333 ECC Registered DIMMs)

Terminology

- “line” in a cache: a set of words that are cached as a unit. Usually these are words in adjacent memory locations.

- Fetching a word causes fetching the entire line.

My Terminology

- Two sets of words are “cache buddies” if they correspond to the same line in a cache.

Terminology

- “direct” cache: a cache in which each line corresponds to a fixed collection of addresses in main memory. Only one buddy can be in the cache at a time. When a different buddy is needed, the line is **replaced**.

- “associative” cache: caches in which lines do not correspond to fixed addresses. The purpose is to be able to defer replacing a line.

Terminology

- “set-associative” cache: is a hybrid of direct and associative. Each “set” of lines is treated associatively, but caching is direct **among** sets.

- In an “n-way” set associative cache, up to n buddies can be stored simultaneously.

Memory fetches

- When a processor needs to fetch a word at a specific address, the cache is first consulted to see if a line in the cache contains the word.
- If so, the word in the cache is used, because this takes less time to get.
- If not ("**read miss**") the word is fetched from memory, **and** also stored in the cache for future reference.

Terminology

- "**miss**": When the sought word is not in the cache.
- "**hit**": When the sought word is already in the cache.
- "**miss penality**": The ratio of time to fetch a word from memory vs. from the cache. Typically about 100-200.

Modification to memory

- If all memory were read-only, there would be no issue.

- If memory can be written:
  - What happens to cached copies of a word if one processor writes a word?
  - Those copies must immediately become **invalid**.

Terminology

- A “write miss” occurs when a processor wants to write to a location for which it does not have a cached copy.

Bus-Based Solutions

- Cache lines and the memory bus can carry extra bits to implement cache coherence.
- Each cache line carries a ***in*valid bit I** to indicate whether or not the line is valid.
- Writing to a word in one cache **invalidates** lines containing the word in other caches (called "**snooping**").
- "**snarfing**" option: When one processor writes a word, other processors caching the word **immediately update their copies**.

Write Protocols

- **Write-through** means that when a word in cache is written, it is at the same time written to memory.
- **Write-back** means the word is written to memory when the cache line is replaced (due to read or write of a buddy address), rather than when the word is first written in the cache.
- **No-write** means that caching only applies to reads.

More States

- In addition to the I (invalid) state, there can be:
- M (**modified**) state, meaning that this line has been written by this processor and has not yet been written to memory.
- S (**shared**) state, meaning that this line exists in some cache, but has not been written by any processor, thus **does not have to be written back** when it is replaced.

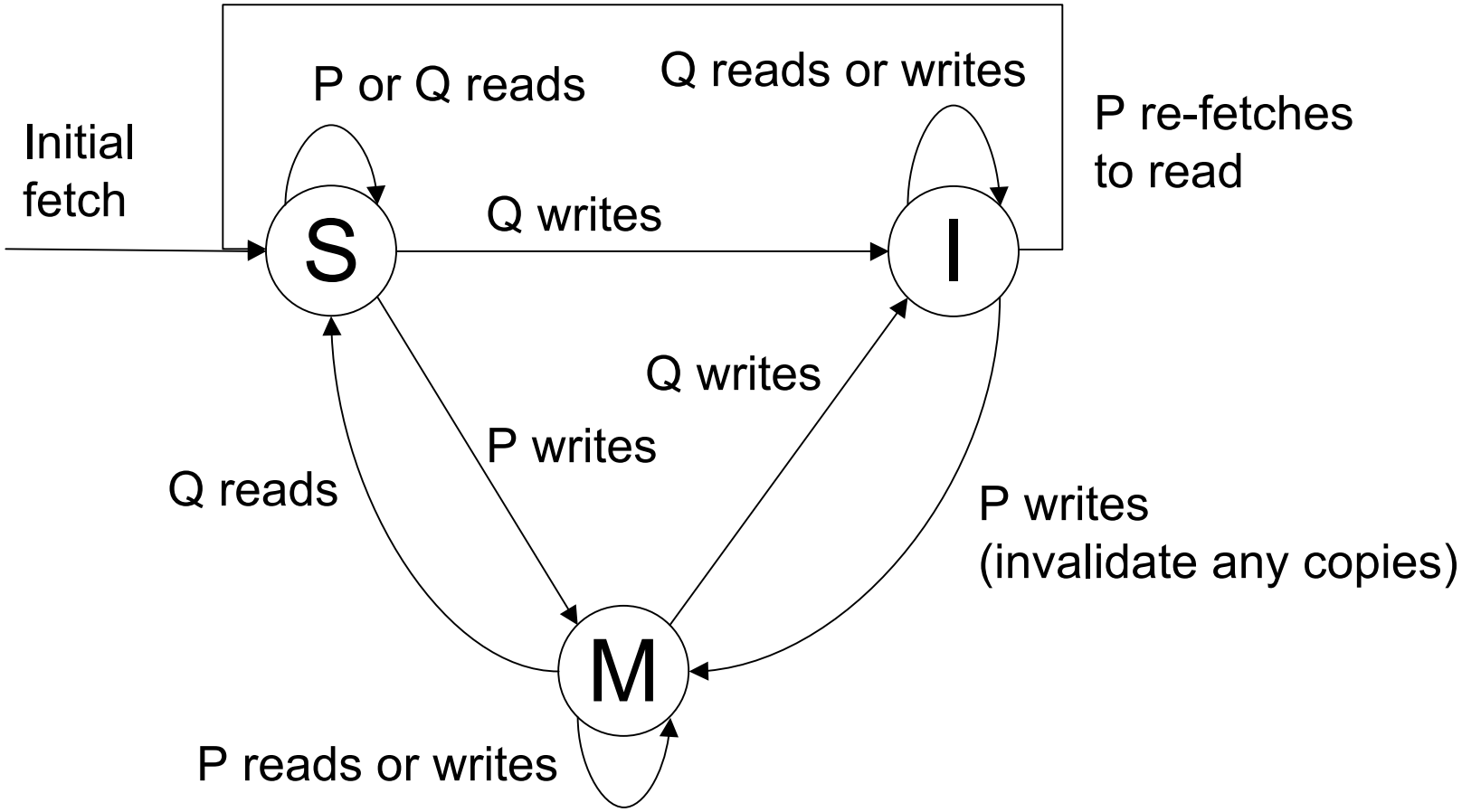
MSI Protocol

- Only one processor can have a line in state M. Other processors caching the line must have it in I, not S nor I.

- Any number of processors can have a line in state S, so long as no processor has it in state M.

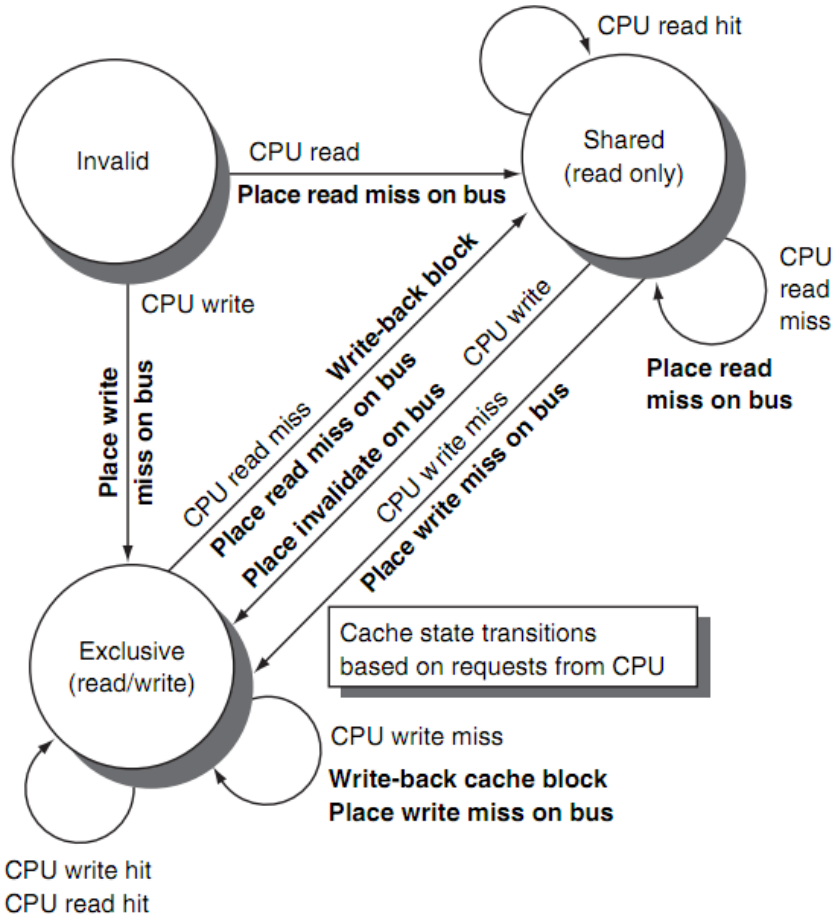
MSI Protocol

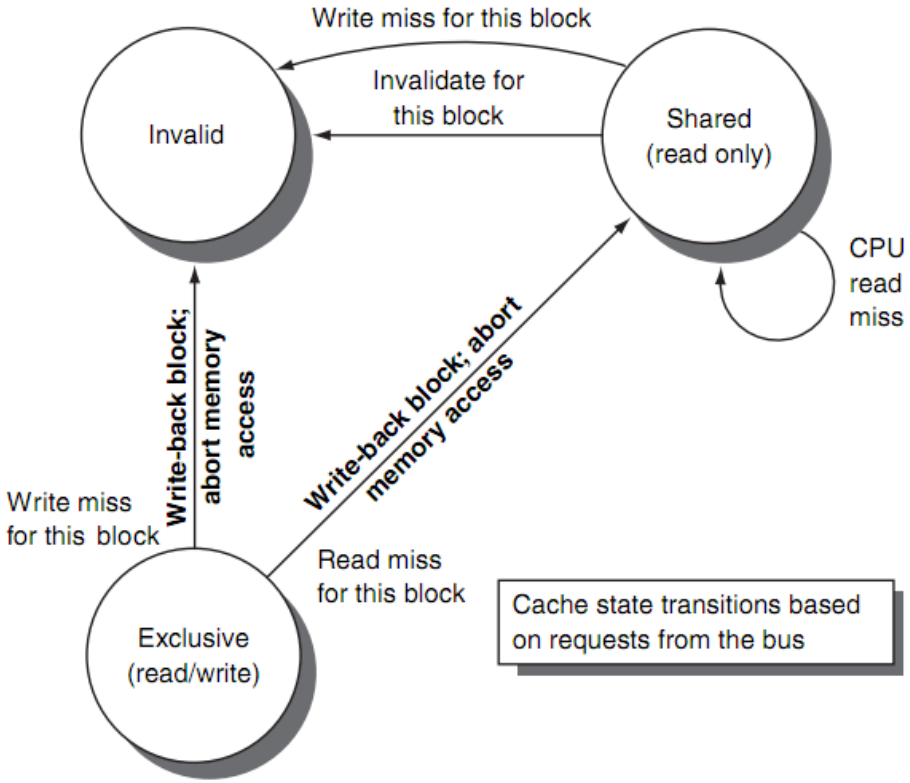
P = this processor
Q = some other processor



The M state is also called E for Exclusive.

Hennessey and Patterson version





Action Table from Hennessey and Patterson

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	processor	shared or modified	normal hit	Read data in cache.
Read miss	processor	invalid	normal miss	Place read miss on bus.
Read miss	processor	shared	replacement	Address conflict miss: place read miss on bus.
Read miss	processor	modified	replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	processor	modified	normal hit	Write data in cache.
Write hit	processor	shared	coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	processor	invalid	normal miss	Place write miss on bus.
Write miss	processor	shared	replacement	Address conflict miss: place write miss on bus.
Write miss	processor	modified	replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	bus	shared	no action	Allow memory to service read miss.
Read miss	bus	modified	coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	bus	shared	coherence	Attempt to write shared block; invalidate the block.
Write miss	bus	shared	coherence	Attempt to write block that is shared; invalidate the cache block.
Write miss	bus	modified	coherence	Attempt to write block that is exclusive elsewhere: write-back the cache block and make its state invalid.

Write Serialization

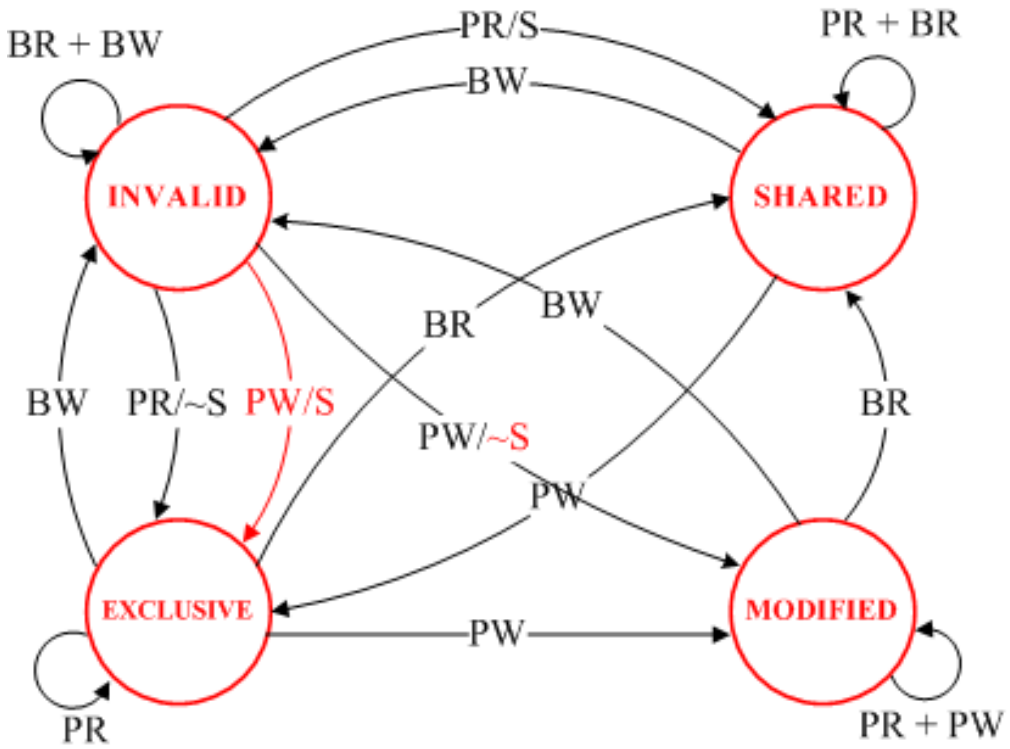
- When writing to a block that is shared, the writing processor must acquire bus access to broadcast its invalidation.
- If two processors attempt to write shared blocks at the same time, their attempts to broadcast an invalidate operation are **serialized** when they arbitrate for the bus.
- The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated.

MESI Protocol

- The M state of MSI is split into two states:
  - M: The line has been modified by this processor.
  - E exclusive: Meaning that this processor has the line, but it has **not been modified**

    (and thus does not have to be written back if some other processor invalidates it).
- A processor must acquire the line for reading before it can write.

MESI State Transitions



http://www.scss.tcd.ie/Jeremy.Jones/vivio/caches/MESI.png

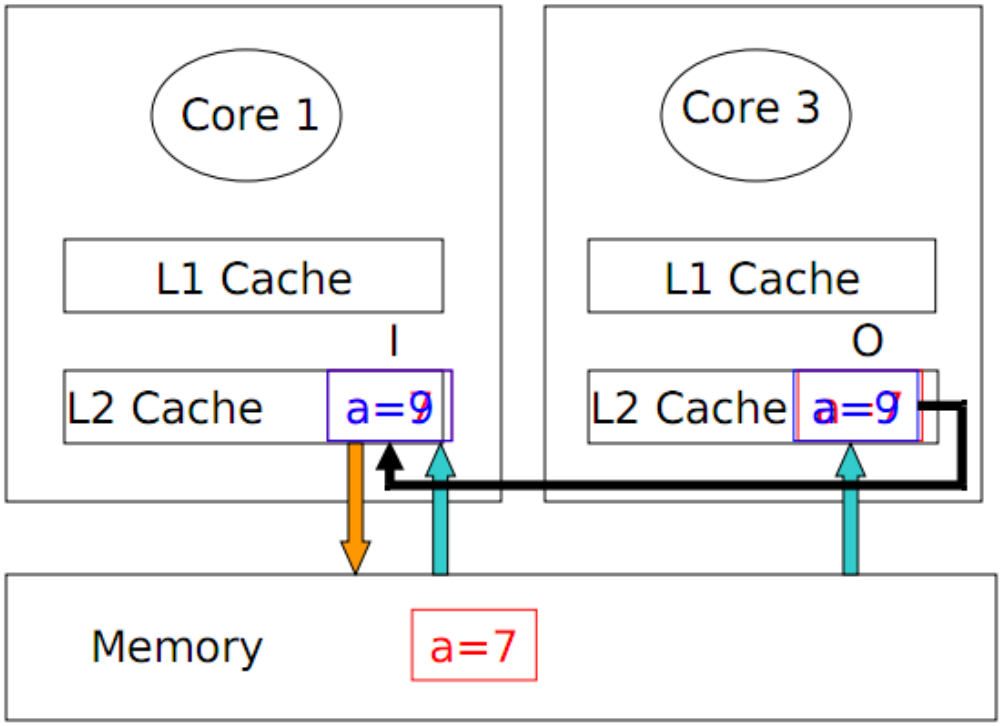
MOESI Protocol

- O stands for “Owner”
- One processor can transfer ownership of a line to another, without writing to memory.
- The other processor effectively “snarfs” the line from the current owner when the latter attempts to write the line to memory.
- The copy of the line in memory can be stale.

MOESI States

State	Correct in Cache	Correct in Memory	Copies in other caches	Write needed?
Exclusive	Y	Y	N	N
Modified	Y	N	N	Y
Owned	Y	Maybe	Maybe	Possibly
Shared	Y	Maybe	Maybe	N
Invalid	N	Maybe	Maybe	N

MOESI protocol used by AMD (e.g. church?)



http://usqcd.jlab.org/usqcd-docs/qmt/multicoretalk.pdf

Other Protocol States Possible

- F: Forward
  - split off of S
  - used in MESIF
  - F responds uniquely to requests
- R: Recent
  - split off of E
  - used in MERSI

- See

http://en.wikipedia.org/wiki/MESIF_protocol

http://en.wikipedia.org/wiki/MERSI_protocol

Scalability

- Scalability is a problem with bus-based systems.
- Distributed shared-memory systems use **directory** basis instead.
  - Directory tells where the item resides, not the item itself.

Memory Consistency

- Cache Coherence is a necessary, but **not sufficient** for semantically transparent multiprocessing.
- A **consistency model** sets down assumptions that can be made about the way **sequences** of memory requests interact.
- The model could be more conservative than the actual hardware.

Some Reasons Why Consistency is Non-Trivial

- Processors operate asynchronously with respect to each other. Memory reads and writes are not centrally coordinated. Thus there can be races between these operations.

- Compilers often reorder code, ignoring the possibility that the code might not be executed in isolation.

Strict Consistency

- The order in which requests are completed is exactly that in which they were issued.
- Problem: Processors operate asynchronously with respect to each other, without a central global clock.
- This model is thus **idealized and unrealistic**, as well as very restrictive.

Memory Operation Asynchrony

- Memory accesses (read or write) are not instantaneous.
- They may take 10’s of processor clock cycles.

Sequential Consistency

- The **result** of any execution is the same **as if** the operations of all the processes were executed in ***some*** [possibly interleaved] sequential order,

  and

- the operations of each individual process appear in this sequence in the order specified by its program ("program order"). [Lamport 1979 paraphrased]

IEEE TRANSACTIONS ON COMPUTERS

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programm

September 1979 (vol. 28 no. 9)

pp. 690-691

L. Lamport, Computer Science Laboratory, SRI International

Example

initially x = 0; y = 1

P1:

x = 2;

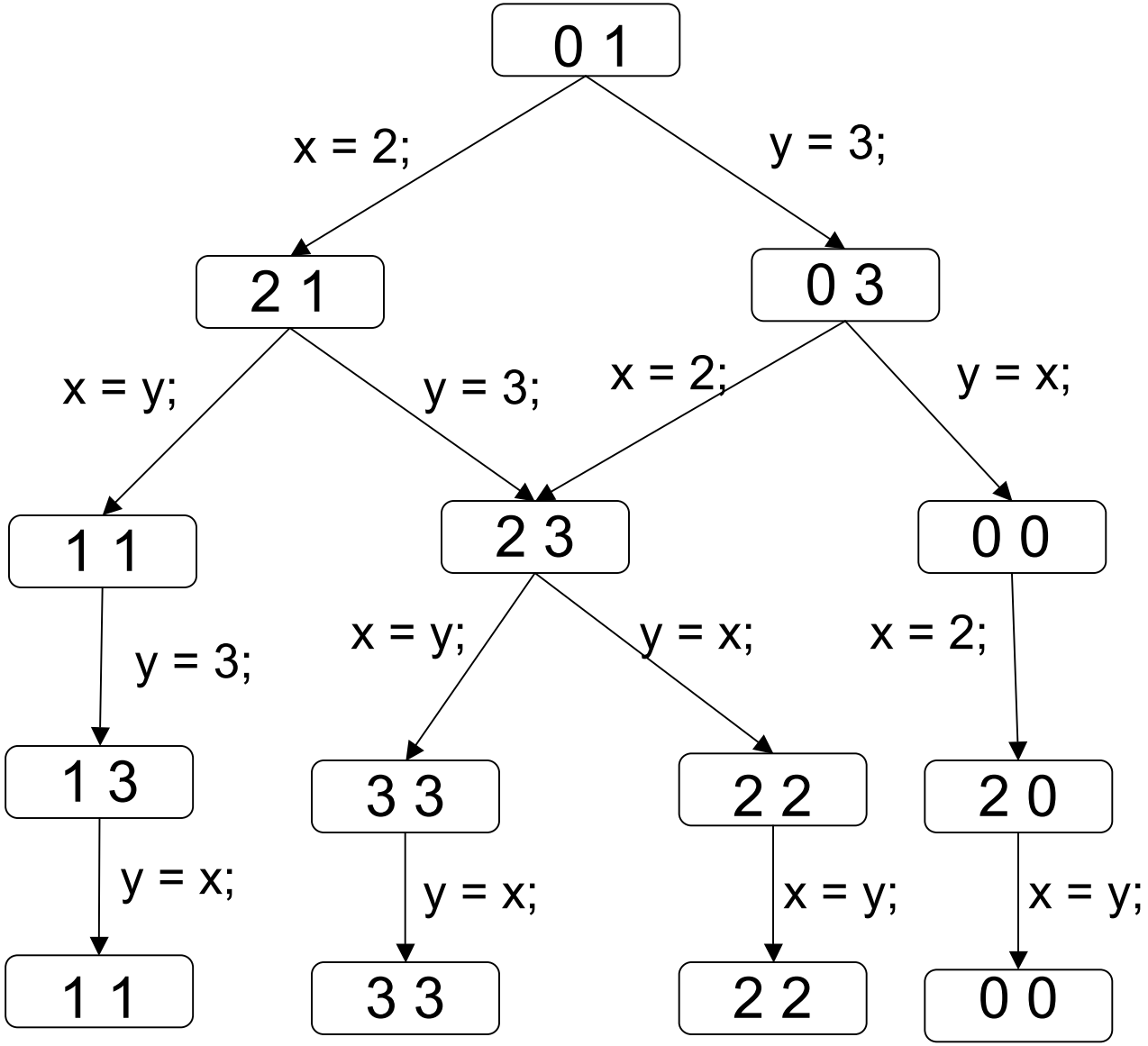
x = y;

P2:

y = 3;

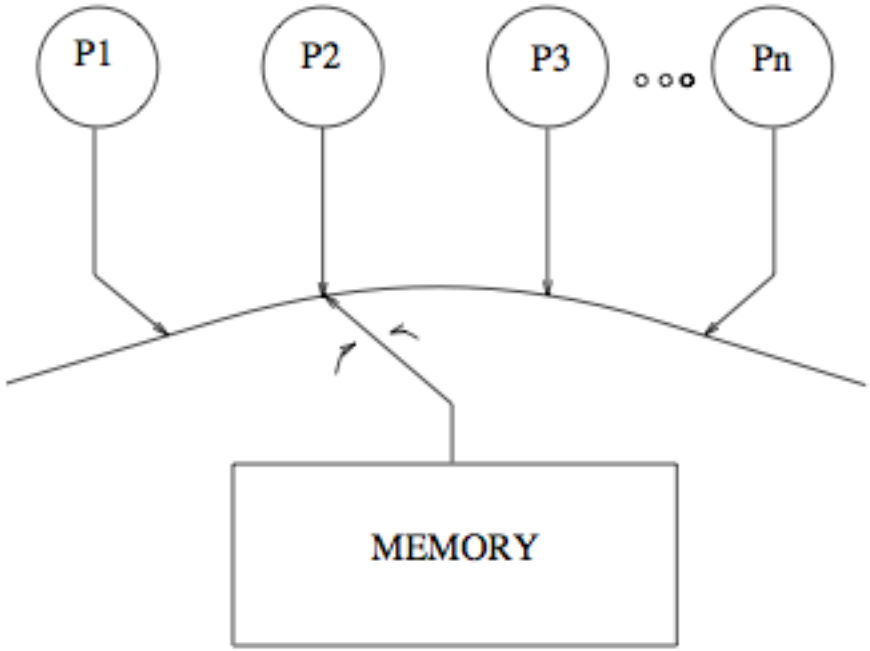
y = x;

finally x = ?, y = ?



Any of these final results is considered acceptable under SC.

Programmer's Abstraction of Sequential Consistency



The switch can be moved randomly between operations.

Sequential Consistency

- In database parlance,

  “equivalent to some sequential order”

  is called “**Serializabilty**” of transactions.

From “Java Concurrency in Practice”

- One convenient mental model for program execution is to imagine that there is a single order in which the operations happen in a program, regardless of what processor they execute on, and that each read of a variable will see the last write in the execution order to that variable by any processor.

- This happy, if unrealistic, model is called **sequential consistency**. Software developers often mistakenly **assume** sequential consistency, but **no modern multiprocessor offers sequential consistency and the Java memory model does not either**. The classic sequential computing model, the von Neumann model, is only a vague approximation of how modern multiprocessors behave.

From “Java Concurrency in Practice”

The bottom line is that modern shared memory multiprocessors (and compilers) can do some surprising things when data is shared across threads, unless you've told them not to through the use of memory barriers. Fortunately, Java programs need not specify the placement of memory barriers; they need only identify when shared state is being accessed,through the proper use of synchronization.

The Java Memory Model in 500 Words or Less

From “Java Concurrency in Practice”

The Java Memory Model is specified in terms of **actions**, which include reads and writes to variables, locks and unlocks of monitors, and starting and joining with threads.

The JMM defines a partial ordering called ***happens before*** on all actions within the program. To guarantee that the thread executing action B can see the results of action A (whether or not A and B occur in different threads), there must be a happens before relationship between A and B.

In the absence of a happens before ordering between two operations, the JVM is free to reorder them as it pleases.

Rules for “Happens-Before” from “Java Concurrency in Practice”

The rules for happens-before are:

- Program order rule. Each action in a thread happens-before every action in that thread that comes later in the program order.
- Monitor lock rule. An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock.[3]
- Volatile variable rule. A write to a volatile field happens-before every subsequent read of that same field.[4]
- Thread start rule. A call to `Thread.start` on a thread happens-before every action in the started thread.
- Thread termination rule. Any action in a thread happens-before any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning `false`.
- Interruption rule. A thread calling `interrupt` on another thread happens-before the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted` or `interrupted`).
- Finalizer rule. The end of a constructor for an object happens-before the start of the finalizer for that object.
- Transitivity. If A happens-before B, and B happens-before C, then A happens-before C.

Common Notation for Consistency Modeling

- {R,W}(*var*)*value*

- *e.g.*
  - W(x)5 means 5 is written to x
  - R(y)6 means y is read, giving 6

Sequential Consistency Example

x is 0 initially

P_1:	W(x) 1		
P_2:		R(x) 0	R(x) 1

- P_2 does not see P_1's write of x = 1 before its first read of x, so it happens to have an out-of-date value.

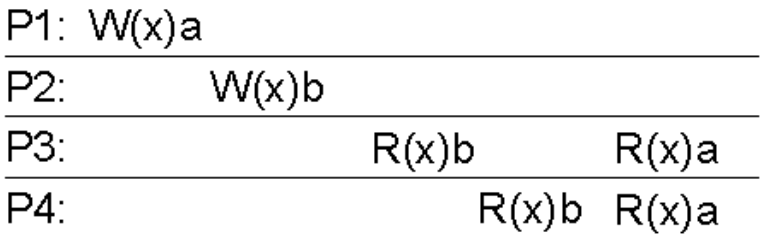
- However, the write propagates to P_2 before its second read of x.

- This is **allowed** under SC because:
  - Processors do not always have to see up-to-date values
  - Processors just need to see writes in the order they happen
  - There is ***an*** order that explains what P_2 sees.

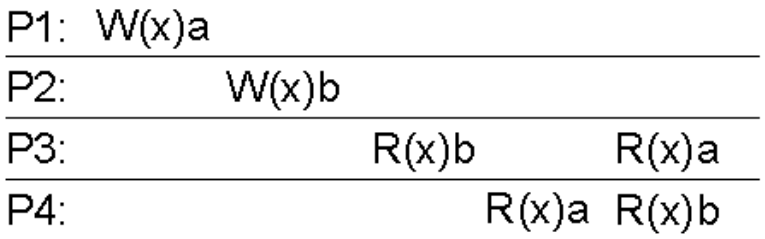
- Note: it would also have been legal under SC for W(x) 1 to propagate to P_2 before its first read of x or after its second read of x, i.e. the reads could have resulted in 0 0 or 1 1 as well as 0 1.

Which History is Sequentially Consistent (if either)?

x is c initially



(a)



(b)

Implications of SC

- The system behaves consistent with ***some*** serialization of reads and writes that ***respects program order***.
- Each read must therefore see the most recent value written in ***that one*** serialization.
- Hence it is impossible for two processes to read successively-written values from a given location in **opposite** orders.

Cache Coherence vs SC

- Coherence demands serialization on a ***per-location*** basis,
- but does not require a **single** serialization overall.
- Thus CC is a form of consistency, but one weaker than SC.

Cache Coherence Alone Does Not Imply SC

x = y = 0 initially

P_1:	W(x) 1	R(y) 0
P_2:	W(y) 1	R(x) 0

This history is possible with a coherent cache.
Each process initiates a write, then moves on to read a different variable.

Each process can legitmately read a “stale” value (0)
before the other’s write has finished propagating.

This history is **not SC**. There is no serialization consistent with it, as in any serialization x or y must get 1 before anything else happens.

Example

(from Rauber and Rünger)

Example We consider three processors P_1, P_2, P_3 which execute a parallel program with shared variables x_1, x_2, x_3. The three variables x_1, x_2, x_3 are assumed to be initialized to 0. The processors execute the following programs:

processor	P_1	P_2	P_3
program	(1) $x_1 = 1$;	(3) $x_2 = 1$;	(5) $x_3 = 1$;
	(2) print x_2, x_3;	(4) print x_1, x_3;	(6) print x_1, x_2;

What can be printed?
Are there any sequences of bits that can't be printed?
(Maybe start by stating what kind of consistency is assumed.)

How to Ensure SC?

In a parallel system, sequential consistency can be guaranteed by the following *sufficient conditions* [35, 45, 157]:

(1) Every processor issues its memory operations in program order. In particular, the compiler is not allowed to change the order of memory operations, and no out-of-order executions of memory operations are allowed.

(2) After a processor has issued a write operation, it waits until the write operation has been completed before it issues the next operation. This includes that for a write miss all cache blocks which contain the memory location written must be marked invalid (I) before the next memory operation starts.

(3) After a processor has issued a read operation, it waits until this read operation and the write operation whose value is returned by the read operation have been entirely completed. This includes that the value returned to the issuing processor becomes visible to all other processors before the issuing processor submits the next memory operation.

Rauber and Rünger - Parallel Programming

Serialization of Writes alone is insufficient

(Example from Rauber and Rünger)

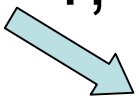
Initially: x = 0, y = 0		
P1	P2	P3
x = 1;	while(x == 0) {}; y = 1;	while(y == 0) {}; print(x);

P3 could print 0 if it didn't wait for P1's write of x = 1 to be completed.

This would show as a violation of causality.

Causality Violation

Initially: x = 0, y = 0		
P1	P2	P3
x = 1;	while(x == 0) {};	while(y == 0) {};
	y = 1;	print(x);



P2 could execute y = 1 before P3 sees the effect of x = 1. So P3 prints 0.
But the **causality chain** x = 1; y = 1; print(x) suggests P3 should print 1.

Relaxed Consistency Models

- These relax the conditions of SC, for purposes of achieving better performance, generally.

- For example, writes may be allowed to occur in a different order than in the program.

Causal Consistency (weaker than SC)

- Memory operations that potentially should **causally related** are seen by every node of the system in the same order.
- **Concurrent writes** (i.e. ones that are not causally related) may be seen in different order by different nodes.
- The JMM seems partly based on this idea.

Causal Relationships

- A **read** (from some variable) followed by a write (to **any** variable) from the **same process** is assumed to be causal, because the value written could have depended on the value read.

- A **write** to a variable, followed by a read or write of the **same variable** by a process is considered causal. But writes to different variables are not considered causal.

- The **transitive closure** of these relations is also considered causal.

Causally Consistent, but not SC

(David Mosberger, Memory Consistency Models, ACM SIGOPS Operating Systems Review, Volume 27 Issue 1, Jan. 1993)

P_1 :	$W(x)1$		$W(x)3$	
P_2 :	$R(x)1$	$W(x)2$		
P_3 :	$R(x)1$		$R(x)3$	$R(x)2$
P_4 :	$R(x)1$		$R(x)2$	$R(x)3$

x is 0 initially

Thus W(x)1 precedes R(x)1 and thus W(x)2, but W(x)2 and W(x)3 are unordered.

The two R(x)3, R(x)2 orders violate SC.

PRAM Consistency model

- PRAM = “Pipelined RAM”
- Do not confuse with the earlier PRAM (= “Parallel Random-Access Machine”)

- Local copies of each variable for each processor.

- Writes broadcast updates to all copies simultaneously.

- **Writes can be seen in different orders acrosss processors!**

PRAM Consistent, not CC

(David Mosberger, Memory Consistency Models, ACM SIGOPS Operating Systems Review, Volume 27 Issue 1, Jan. 1993)

The following execution history is legal under PRAM but not under CC:

P_1 :	$W(x)1$			
P_2 :		$R(x)1$	$W(x)2$	
P_3 :			$R(x)1$	$R(x)2$
P_4 :			$R(x)2$	$R(x)1$

P_3 and P_4 observe the writes by P_1 and P_2 in different orders, although $W(x)1$ and $W(x)2$ are potentially causally related. Thus, this would not be a legal history for CC.

Practical Aspects

DOI:10.1145/1785414.1785443

x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors

By Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen

JULY 2010 | VOL. 53 | NO. 7 | COMMUNICATIONS OF THE ACM 89

TSO = "Total Store Order"

x86-TSO Paper

- Laments the imprecision of consistency models provided in vendor documentation
- Observes that consistency behavior differs among processors of different vendors (e.g. AMD vs. Intel), or even the same vendor.
- Provides a set of test cases, some of which fail under specifications, but the failure of which is not always seen in practice.
- Describes a **formal** programmer abstraction model that suits existing processor implementations.

Example SB

- Violation is observable, as a consequence of FIFO memory-write buffering.

it is possible for both to read 0 *in the same execution*. It is easy to check that this result cannot arise from any interleaving of the reads and writes of the two processors; modern x86 multiprocessors do not have a sequentially consistent semantics.

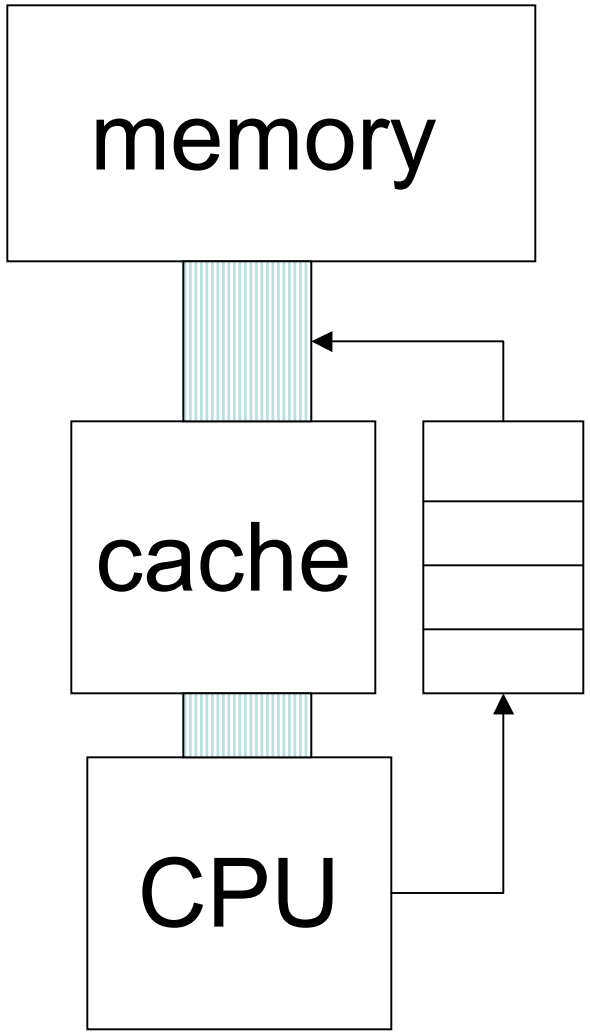
SB

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[y]	MOV [y]←1 MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

Microarchitecturally, one can view this particular example as a visible consequence of store buffering: if each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), then the reads from y and x could occur before the writes have propagated from the buffers to main memory.

write buffer

(write-through caching policy)



write buffer

(logically part of memory,
physically part of cache)

From:
Jacob, et al.
Memory Systems
Morgan Kaufmann, 2008

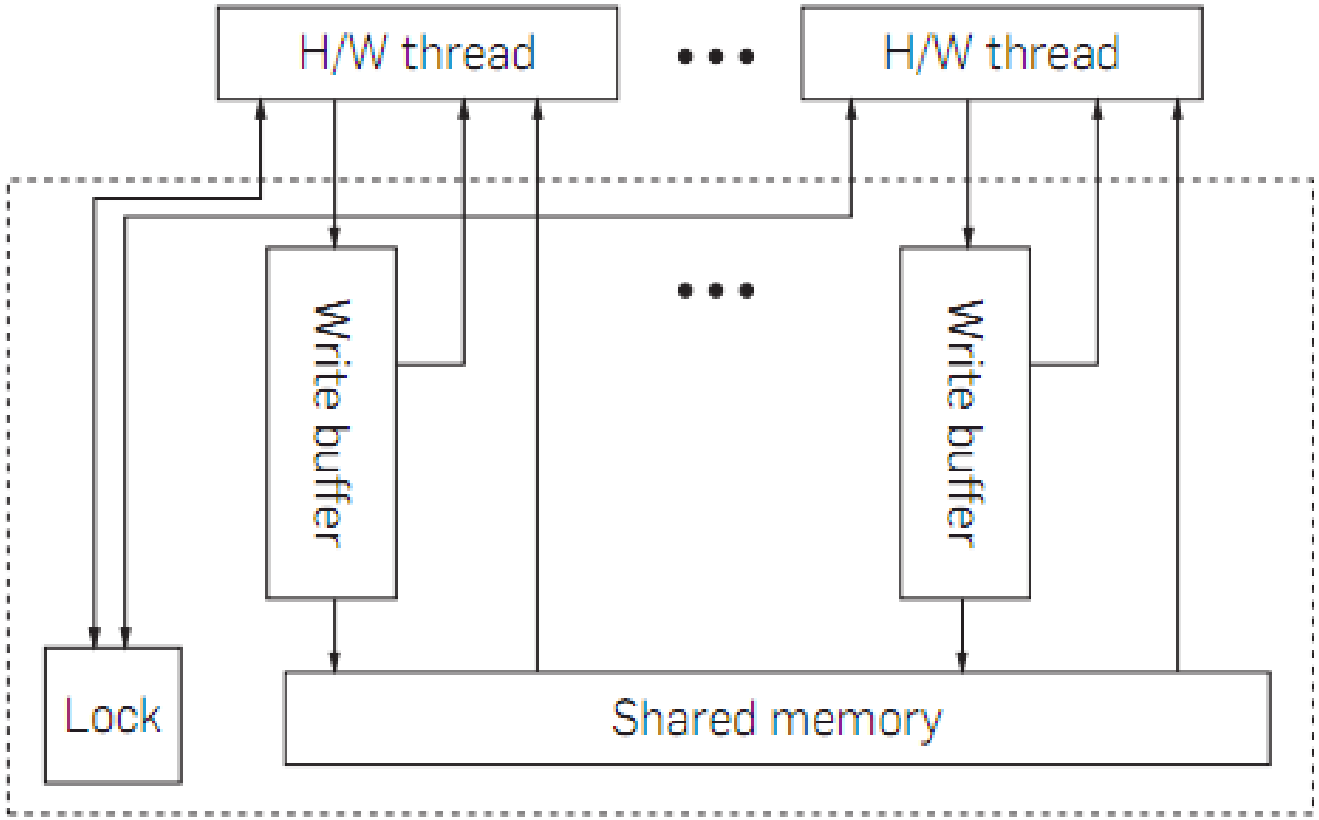
Example IRIW

- can arise from write-buffer shared by more than one processor

IRIW

Proc 0	Proc 1	Proc 2	Proc 3
MOV [x]←1	MOV [y]←1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Forbidden Final State: Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0			

x86-TSO Model Block Diagram



x86-TSO Abstract Machine

- The **store buffers** are FIFO and a reading thread must **read its most recent buffered write**, if there is one, to that address; otherwise reads are satisfied from shared memory.
- An MFENCE instruction **flushes the store buffer** of that thread.
- To execute a LOCK'd instruction, a thread must first **obtain the global lock**. At the end of the instruction, it flushes its store buffer and relinquishes the lock. **While the lock is held by one thread, no other thread can read.**
- A **buffered write** from a thread can propagate to the shared memory at any time except when some other thread holds the lock.

Events in the x86-TSO Model

- $W_p[a]=v$, for a write of value v to address a by thread p
- $R_p[a]=v$, for a read of v from a by thread p
- F_p, for an MFENCE memory barrier by thread p
- L_p, at the start of a LOCK'd instruction by thread p
- U_p, at the end of a LOCK'd instruction by thread p
- τ_p, for an internal action of the storage subsystem, propagating a write from p's store buffer to the shared memory

Behavior of the x86-TSO Model

1. $R_p[a]=v$: p can read v from memory at address a if p is not blocked, there are no writes to a in p's store buffer, and the memory does contain v at a.
2. $R_p[a]=v$: p can read v from its store buffer for address a if p is not blocked and has v as the newest write to a in its buffer.
3. $W_p[a]=v$: p can write v to its store buffer for address a at any time.
4. τ_p: if p is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread.
5. F_p: if p's store buffer is empty, it can execute an MFENCE (note that if a hardware thread encounters an MFENCE instruction when its store buffer is not empty, it can take one or more τ_p steps to empty the buffer and proceed, and similarly in 7 below).
6. L_p: if the lock is not held, it can begin a LOCK'd instruction.
7. U_p: if p holds the lock, and its store buffer is empty, it can end a LOCK'd instruction.

Formal Definition

- Labelled transition system.
- HOL4 (Higher-Order Logic) Model