Lecture 12:

Memory Consistency

Parallel Computer Architecture and Programming

CMU / 清华大学, Summer 2017

What you should know from this lecture

- Understand the motivation for relaxed consistency models
- Understand the implications of relaxing W → R ordering

Who should care about today's topic

- Students who:
  - Want to implement a synchronization library
  - Will ever work a job in kernel (or driver) development
  - Seeks to implement lock-free data structures *
  - Does any of the above on ARM processors

* Topic of a later lecture

Memory coherence vs. memory consistency

- **Memory 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
  - In other words: it is possible to put all operations involving X on a timeline such that the observations of all processors are consistent with that timeline

- **Memory consistency** defines the behavior of reads and writes to different locations (as observed by other processors)
  - 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

Coherence vs. consistency

(said again, perhaps more intuitively this time)

- The goal of cache coherence is to ensure that the memory system in a parallel computer behaves as if the caches were not there
  - Just like how the memory system in a single-processor system behaves as if the cache was not there
- A system without caches would have no need for cache coherence
- Memory consistency defines the allowed behavior of loads and stores to different addresses in a parallel system
  - The allowed behavior of memory should be specified whether or not caches are present (and that's what a memory consistency model does)

Memory operation ordering

- 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 *
  - 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

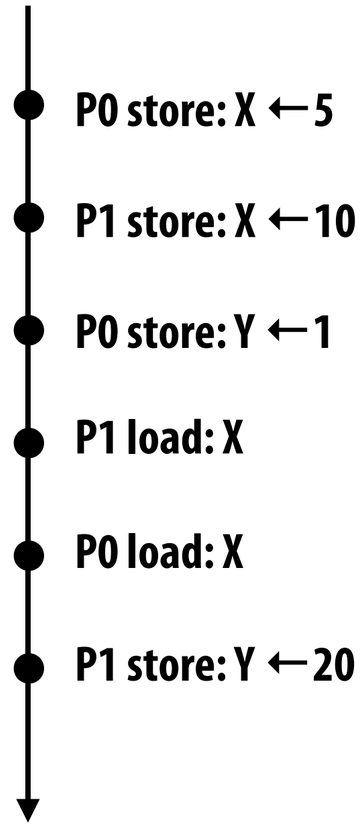
- A sequentially consistent memory system maintains all four memory operation orderings

* To clarify: "write must commit before subsequent read" means:
When a write comes before a read in program order, the write must commit (its results are visible) by the time the read occurs.

Sequential consistency

- A memory system is sequentially consistent if the result of any parallel execution is the same as if all the memory operations were executed in some sequential order, and the memory operations of any one processor are executed in program order.

There is a timeline of all memory operations that is consistent with observed values



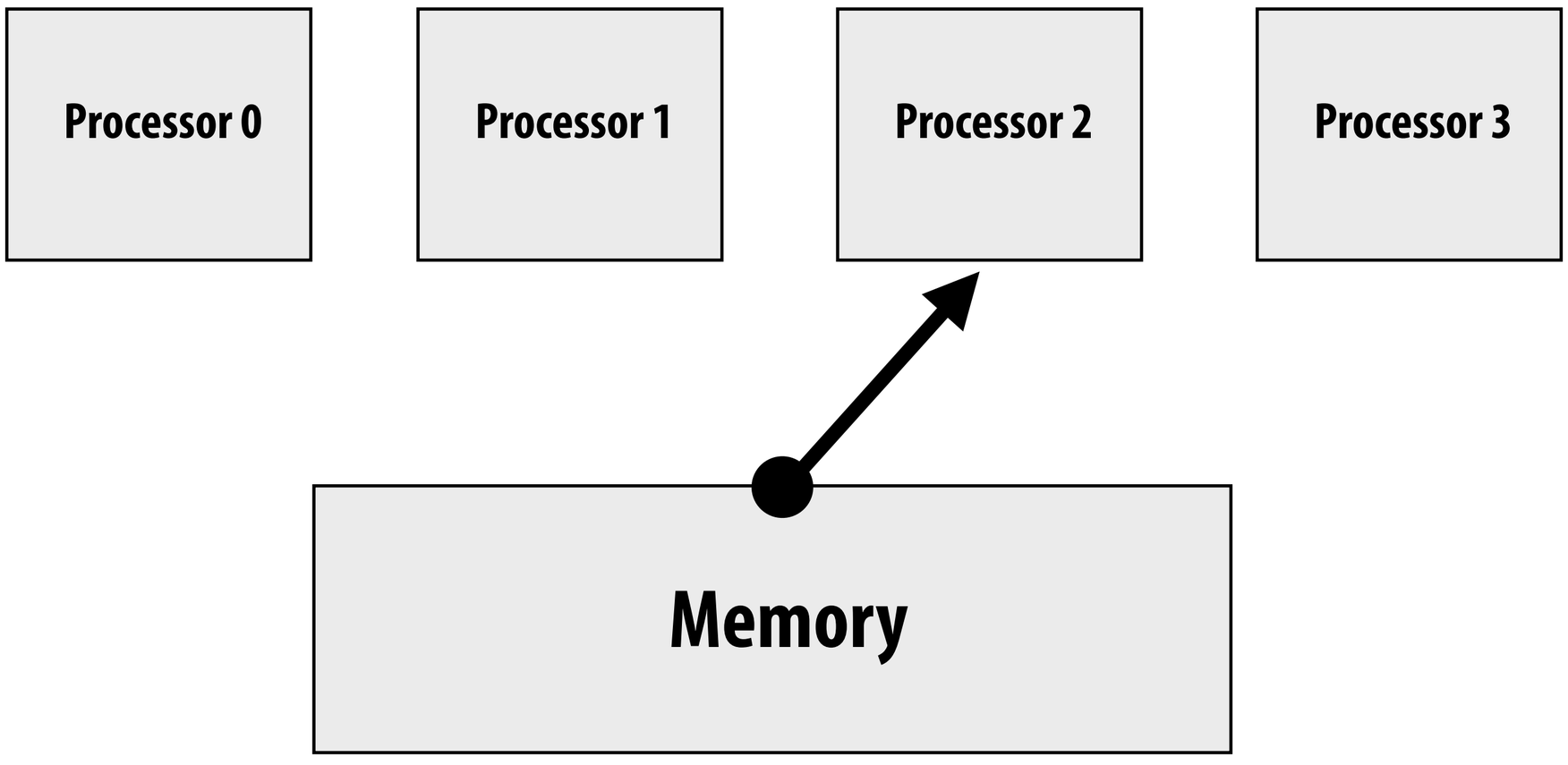
Note, now timeline lists operations to addresses X and Y

(in coherence lecture all operations were only to address X)

[Adve and Gharachorloo 95]

Sequential consistency (switch metaphor)

- All processors issue loads and stores in program order
- Memory chooses a processor, performs a memory operation to completion, then chooses another processor, . . .



Quick example

Thread 1 (on P1)

```
A = 1;
if (B == 0)
  print “Hello”;
```

Thread 2 (on P2)

```
B = 1;
if (A == 0)
  print “World”;
```

Assume A and B are initialized to 0.

Question: Imagine threads 1 and 2 are being run simultaneously on a two processor system. What will get printed?

Answer: assuming writes propagate immediately (e.g., P1 won’t continue to ‘if’ statement until P2 observes the write to A), then code will either print “hello” or “world”, but not both.

Consider various interleavings…

Thread 1 (on P1)

```
A = 1;
if (B == 0)
  print “Hello”;
```

Thread 2 (on P2)

```
B = 1;
if (A == 0)
  print “World”;
```

```
P1: Write 1 to A
P1: Read B  (it’s 0, print “Hello”)
P2: Write 1 to B
P2: Read A  (it’s 1)
```

```
P1: Write 1 to A
P2: Write 1 to B
P1: Read B  (it’s 1)
P2: Read A  (it’s 1)
```

```
P2: Write 1 to B
P2: Read A  (it’s 0, print “World”)
P1: Write 1 to A
P1: Read B  (it’s 1)
```

```
P1: Write 1 to A
P2: Write 1 to B
P2: Read A  (it’s 1)
P1: Read B  (it’s 1)
```

Answer: assuming writes propagate immediately (e.g., P1 won’t continue to ‘if’ statement until P2 observes the write to A), then code will either print “hello” or “world”, but not both.

Relaxing memory operation ordering

- A <u>sequentially consistent</u> memory system maintains all four memory operation orderings (W→R, R→R, R→W, W→W)

- Relaxed memory consistency models allow certain orderings to be violated

Back to the quick example

Thread 1 (on P1)

```
A = 1;
if (B == 0)
print “Hello”;
```

From the processor’s perspective, these are independent instructions in each thread.

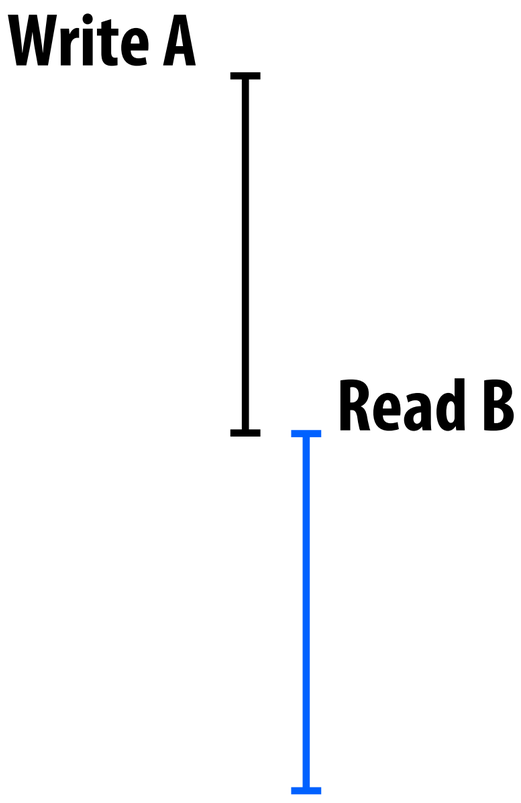
Thread 2 (on P2)

```
B = 1;
if (A == 0)
print “World”;
```

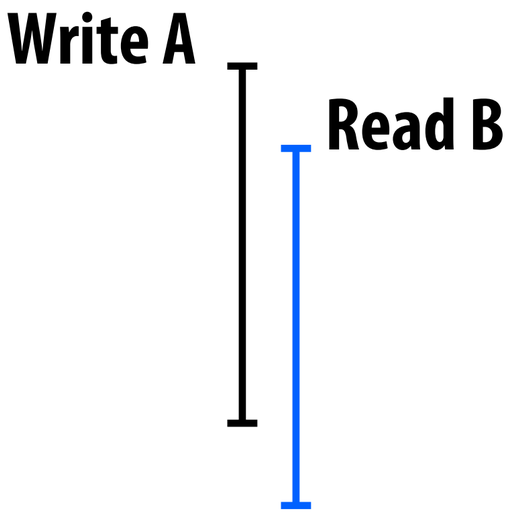
(If this was a sequential program, it would not violate program correctness if a processor chose to reorder the instructions within a single thread… e.g., execute them concurrently)

Motivation for relaxed consistency: hiding latency

- **Why are we interested in relaxing ordering requirements?**
  - To gain performance
  - Specifically, hiding memory latency: overlap memory access operations with other operations when they are independent
  - Remember, memory access in a cache coherent system may entail much more work then simply reading bits from memory (finding data, sending invalidations, etc.)



vs.



Another way of thinking about relaxed ordering

Program order

(dependencies in red: required for sequential consistency)

Thread 1 (on P1)

```
A = 1;
B = 1;
unlock(L);
```

Thread 2 (on P2)

```
lock(L);
x = A;
y = B;
```

"Sufficient" order for correctness

(logical dependencies in red)

Thread 1 (on P1)

```
A = 1;
B = 1;
unlock(L);
```

Thread 2 (on P2)

```
lock(L);
x = A;
y = B;
```

An intuitive notion of correct = execution produces the same results as a sequentially consistent system

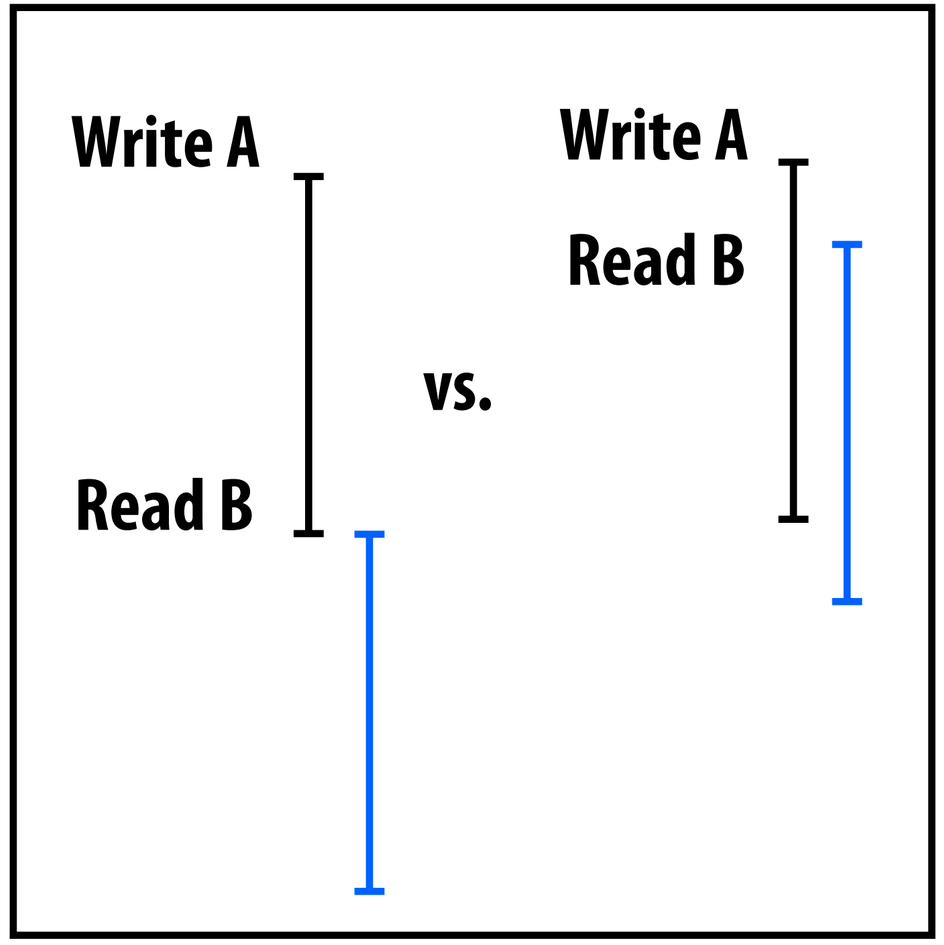
Allowing reads to move ahead of writes

- Four types of memory operation orderings
  - ~~W→R: write must complete before subsequent read~~
  - R→R: read must complete before subsequent read
  - R→W: read must complete before subsequent write
  - W→W: write must complete before subsequent write

← Relaxing W->R ordering allows a read to begin before an earlier write has committed

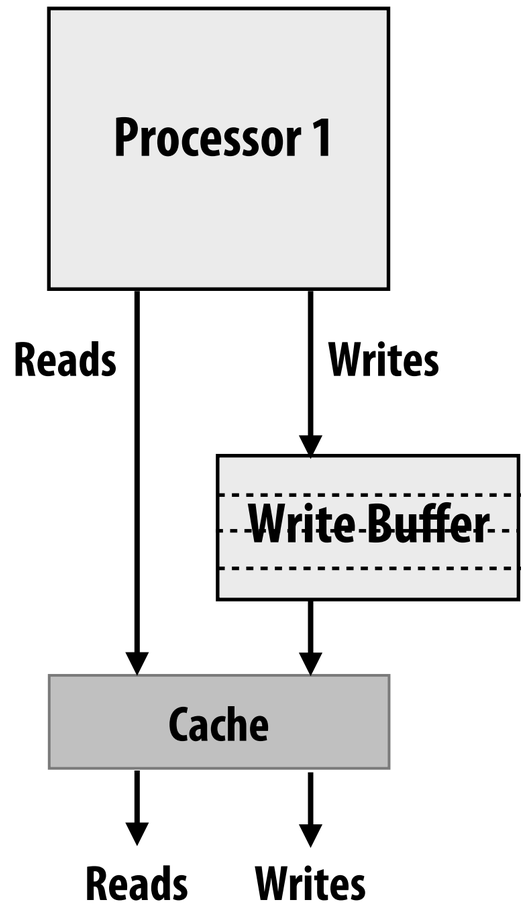
- Reason: allow processor to hide latency of writes when later read is independent

- Examples:
  - Total Store Ordering (TSO)
  - Processor Consistency (PC)



Write buffering example

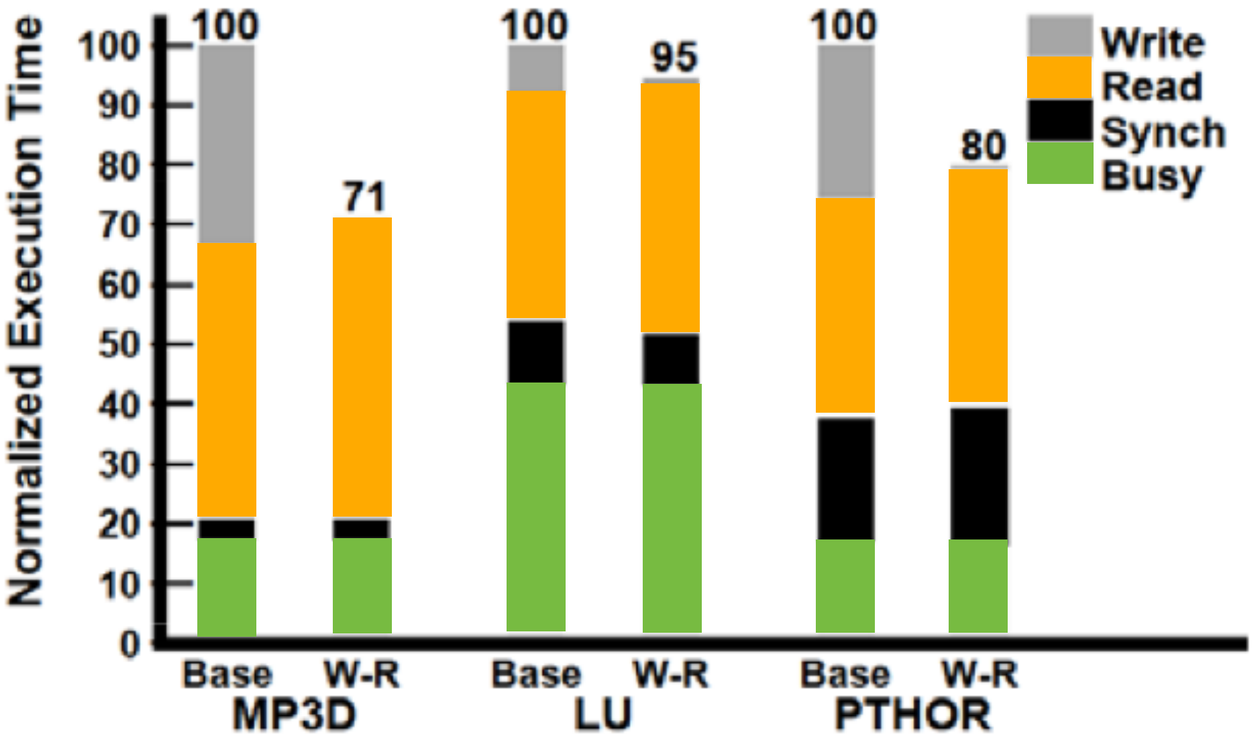
- **Write buffering is a common processor optimization that allows reads to occur before earlier writes have committed**
  - When store is issued, processor "buffers" the store operation in the write buffer (assume store is to address X)
  - Processor immediately begins executing subsequent loads, provided they are not accessing address X (exploits ILP in program)
  - Further writes can also be added to write buffer (write buffer is processed in order, there is no W→W reordering)

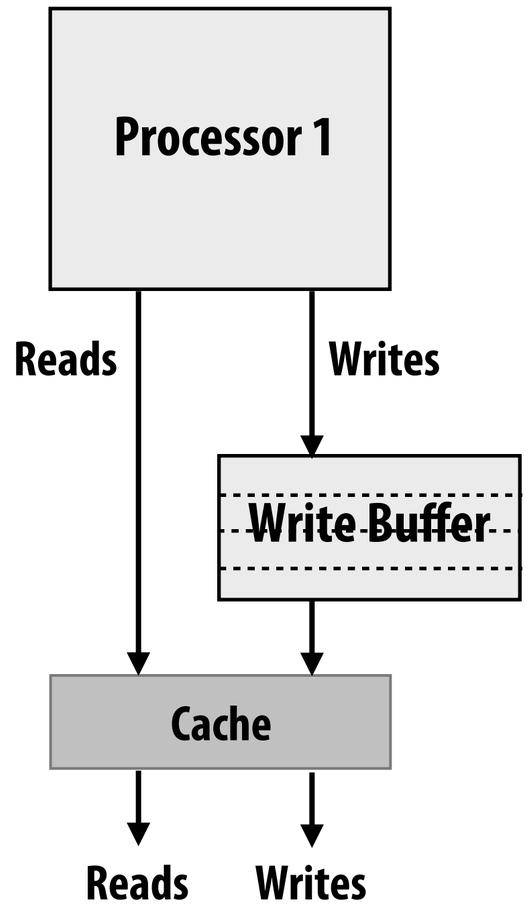


- **Write buffering relaxes W→R ordering**

* Do not confuse a write buffer (shown here) with a cache's write-back buffer (discussed last lecture). Both buffers exist to hide the latency of memory operations. However, the write buffer holds writes that have been issued by the processor, but not yet committed in the system. The write-back buffer holds dirty cache lines that must be flushed to memory so memory stays up to date. The lines are dirty because there was some write to them completed by the processor a long time ago. (This is a good distinction to discuss in comments.)

Relaxed consistency performance





Base: Sequentially consistent execution. Processor issues one memory operation at a time, stalls until completion

W-R: relaxed W→R ordering constraint (notice write latency is almost fully hidden)

Allowing reads to move ahead of writes

- Total store ordering (TSO)
  - Processor P can read B before its early write to A is seen by all processors (processor can move its own reads in front of its own writes)
  - Reads by other processors will not return the new value of A until the write to A is observed by all processors

- Processor consistency (PC)
  - Any processor can read new value of A before the write is observed by all processors (but no guarantee that all processors have seen the write of A commit)

- In TSO and PC, 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)

Four example programs

Assume A and B are initialized to 0
Assume prints are loads

1

Thread 1 (on P1)	Thread 2 (on P2)
`A = 1;`	`while (flag == 0);`
`flag = 1;`	`print A;`

2

Thread 1 (on P1)	Thread 2 (on P2)
`A = 1;`	`print B;`
`B = 1;`	`print A;`

3

Thread 1 (on P1)	Thread 2 (on P2)	Thread 3 (on P3)
`A = 1;`	`while (A == 0);`	`while (B == 0);`
	`B = 1;`	`print A;`

4

Thread 1 (on P1)	Thread 2 (on P2)
`A = 1;`	`B = 1;`
`print B;`	`print A;`

Do results of execution match that of sequential consistency (SC)

	1	2	3	4
Total Store Ordering (TSO)	✔	✔	✔	✗
Processor Consistency (PC)	✔	✔	✗	✗

Clarification

- The cache coherency problem exists because of the optimization of duplicating data in multiple processor caches. The copies of the data must be kept coherent.

- Relaxed memory consistency issues arise from the optimization of reordering memory operations. (Consistency is unrelated to whether or not caches exist in the system.)

Allowing writes to be reordered

- **Four types of memory operation orderings**
  - ~~**W→R: write must complete before subsequent read**~~
  - **R→R: read must complete before subsequent read**
  - **R→W: read must complete before subsequent write**
  - ~~**W→W: write must complete before subsequent write**~~

- **Partial Store Ordering (PSO)**
  - **Execution may not match sequential consistency on program 1 (P2 may observe change to `flag` before change to `A`)**

Thread 1 (on P1)	Thread 2 (on P2)
`A = 1;`	`while (flag == 0);`
`flag = 1;`	`print A;`

Why might it be useful to allow more aggressive memory operation reorderings?

- W→W: processor might reorder write operations in a write buffer (e.g., one is a cache miss while the other is a hit)

- R→W, R→R: processor might reorder independent instructions in an instruction stream (out-of-order execution)

- Keep in mind these are all valid optimizations if a program consists of a single instruction stream

Allowing all reorderings

- Four types of memory operation orderings
  - ~~W⟶R: write must complete before subsequent read~~
  - ~~R⟶R: read must complete before subsequent read~~
  - ~~R⟶W: read must complete before subsequent write~~
  - ~~W⟶W: write must complete before subsequent write~~
- Examples:
  - Weak ordering (WO)
  - 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
```

Example: expressing synchronization in relaxed models

- Intel x86/x64 ~ total store ordering
  - Provides sync instructions if software requires a specific instruction ordering not guaranteed by the consistency model
    - `mm_lfence` ("load fence": wait for all loads to complete)
    - `mm_sfence` ("store fence": wait for all stores to complete)
    - `mm_mfence` ("mem fence": wait for all me operations to complete)

- ARM processors: very relaxed consistency model

A cool post on the role of memory fences in x86:
http://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/

ARM has some great examples in their programmer's reference:
http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf

A great list:
http://www.cl.cam.ac.uk/~pes20/weakmemory/

Acquire/release semantics

- **Operation X with acquire semantics: prevent reordering of X with any load/store after X in program order**
  - **Other processors see X's effect before the effect of all subsequent operations**
  - **Example: taking a lock must have acquire semantics**

these loads can stores can be moved below X

operation X (with acquire semantics)

---

loads and stores that cannot be moved above X

- **Operation X with release semantics: prevent reordering of X with any load/store before X in program order**
  - **Other processors see effects of all prior operations before seeing effect of X.**
  - **Example: releasing a lock must have release semantics**

loads and stores that cannot be moved below X

---

operation X (with release semantics)

loads and stores that can be moved above X

C++ 11 atomic<T>

- **Provides atomic read, write, read-modify-write of entire objects**
  - Atomicity may be implemented by mutex or efficiently by processor-supported atomic instructions (if T is a basic type)
  - More on this after spring break

- **Provides memory ordering semantics for operations before and after atomic operations**
  - By default: sequential consistency
  - See std::memory_order or more detail

Thread 1 (on P1)

```
atomic<int> flag;
int foo;

foo = 1;
flag.store(1);
```

Thread 2 (on P2)

```
// other code…
while (flag.load()==0);
// use foo here…
```

C++ atomic ensures sequentially consistent behavior by default, so compiler must emit appropriate fences on x86

C++ 11 atomic<T>

- **Provides atomic read, write, read-modify-write of entire objects**
  - Atomicity may be implemented by mutex or efficiently by processor-supported atomic instructions (if T is a basic type)
  - More on this after spring break

- **Provides memory ordering semantics for operations before and after atomic operations**
  - By default: sequential consistency
  - See std::memory_order or more detail

Thread 1 (on P1)

```
atomic<int> flag;
int foo;

foo = 1;
flag.store(1, memory_order_release);
```

Thread 2 (on P2)

```
// other code…
while (flag.load(memory_order_acquire)==0);
// use foo here…
```

No fence required on x86

Conflicting data accesses

- 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)

Synchronized programs

- Synchronized programs yield SC results on non-SC systems
  - Synchronized programs are data-race-free

- In practice, most programs you encounter will be synchronized (via locks, barriers, etc. implemented in synchronization libraries)
  - Rather than via ad-hoc reads/writes to shared variables like in the earlier "four example programs" slide

Summary: relaxed consistency

- Motivation: obtain higher performance by allowing recording of memory operations (reordering is not allowed by sequential consistency)

- One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific operation orderings when needed
  - But in practice complexities encapsulated in libraries that provide intuitive primitives like lock/unlock, barrier (or lower level primitives like fence)
  - 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

- Relaxed consistency models differ in which memory ordering constraints they ignore

Eventual consistency in distributed systems

- For many of you, relaxed memory consistency will be a key factor in writing web-scale programs in distributed environments

- "Eventual consistency"
  - Say machine A writes to an object X in a shared distributed database
  - Many copies of database exist for performance scaling and redundancy
  - Eventual consistency guarantees that if there are no other updates to X, A's update will <u>eventually</u> be observed by <u>all</u> other nodes in the system (note: no guarantees on when, so updates to objects X and Y might propagate to different clients differently)