



departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Concurrency and Parallelism

(Concorrência e Paralelismo – CP 11158)

Lecture 6

— Memory Consistency Models —

Slides based in material from:

www.cs.utexas.edu/users/pingali/CS378/2008sp/lectures/consistency.ppt

<https://www.kernel.org/doc/Documentation/memory-barriers.txt>

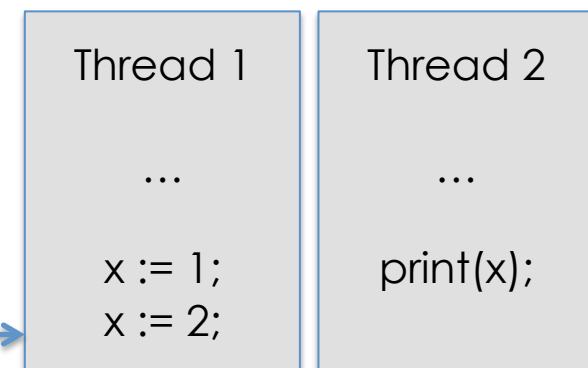
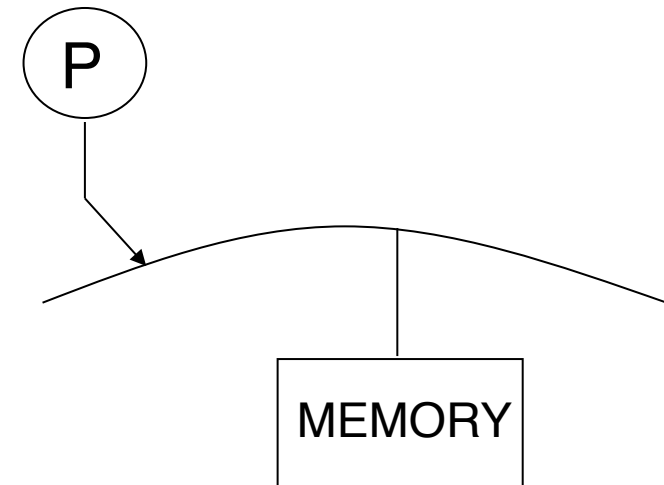
<http://preshing.com/20120710/memory-barriers-are-like-source-control-operations/>

Outline

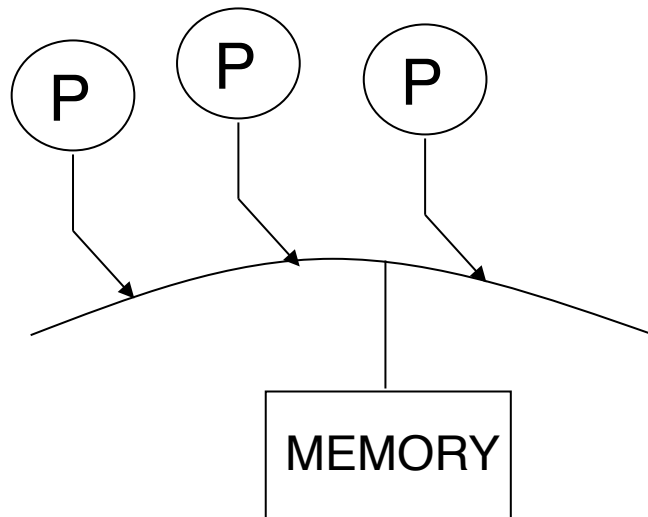
- Review of multi-threaded program execution on uniprocessor
- Need for memory consistency models
- Sequential consistency model
- Relaxed memory models
 - weak consistency model
 - release consistency model
- Conclusions

Multi-threaded programs on uniprocessor

- Processor executes all the threads of the program
 - unspecified scheduling policy
- Operations in each thread are executed in order
- Use of atomic operations: lock/unlock etc. for synchronization between threads
- Result is as if instructions from different threads were interleaved in some order
- Non-determinacy: program may produce different outputs depending on scheduling of threads (e.g.)



Multi-threaded programs on multiprocessor



- Each processor executes one thread
 - let's keep it simple
- Operations in each thread are executed in order
- One processor at a time can access global memory to perform load/store/atomic operations
- No caching of global data
- One can show that running multi-threaded program on multiple processors does not change possible output(s) of program from uniprocessor case

More realistic architecture

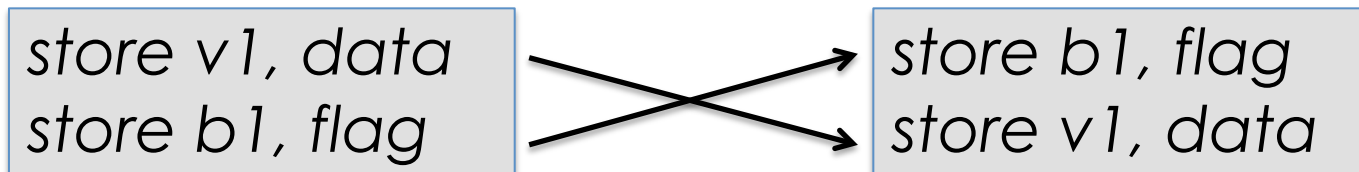
- Two key assumptions so far:
 1. **processors do not cache global data**
 - improving execution efficiency:
 - allow processors to cache global data
 - » leads to cache coherence problem, which can be solved using coherent caches as explained before
 2. **instructions within each thread are executed in order**
 - improving execution efficiency:
 - allow processors to execute instructions out of order subject to data/control dependences
 - » surprisingly, this can change the semantics of the program
 - » preventing this requires attention to the *memory consistency model of processor*

Recall: uniprocessor execution

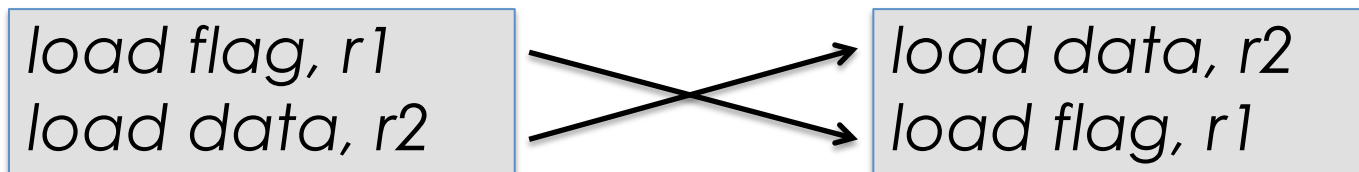
- Processors reorder operations to improve performance
- Constraint on reordering: must respect dependences
 - data dependences must be respected: in particular, loads/stores to a given memory address must be executed in program order
 - control dependences must be respected
- Reorderings can be performed either by compiler or processor

Permitted memory-op reorderings

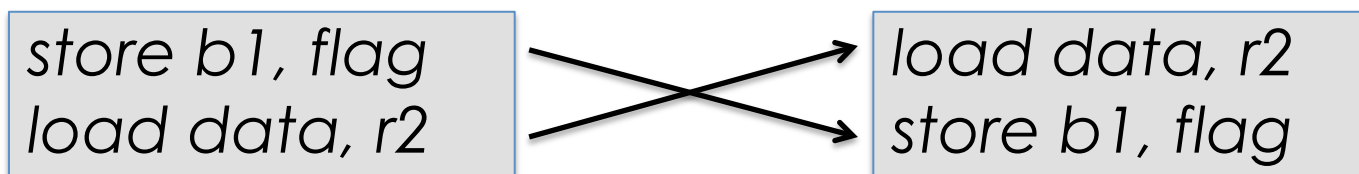
- **Stores to different memory locations**
can be performed out of program order



- **Loads from different memory locations**
can be performed out of program order



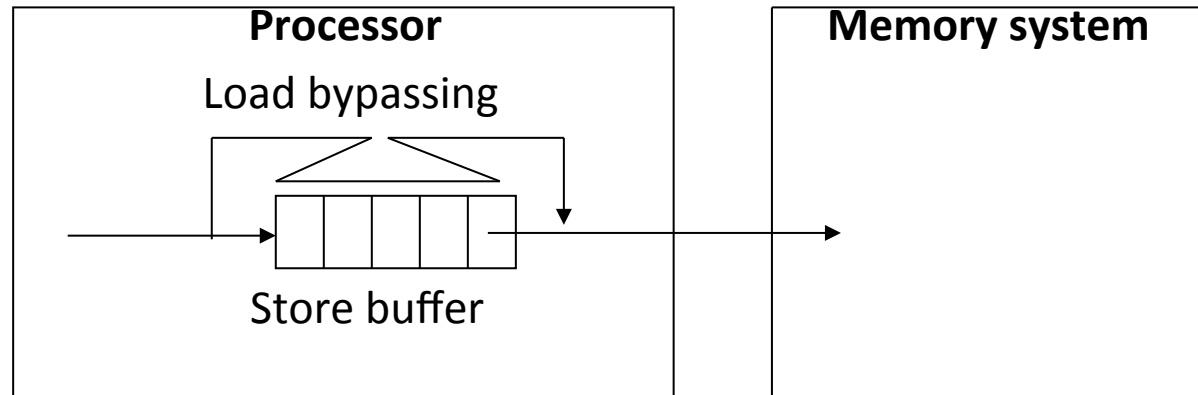
- **Load and store to different memory locations**
can also be performed out of program order



Optimizations Enabled

- $W \rightarrow R$: takes writes off the critical path
- $W \rightarrow W$: memory parallelism
(bandwidth utilization)
- $R \rightarrow WR$: non-blocking caches, overlaps
other useful work with a read miss

Example of hardware reordering



- Store buffer holds store operations that need to be sent to memory
- Loads are higher priority operations than stores since their results are needed to keep processor busy, so they bypass the store buffer
- Load address is checked against addresses in store buffer, so store buffer satisfies load if there is an address match
- Result: load can bypass stores to other addresses

Problem in multiprocessor context

- Canonical model
 - operations from given processor are executed in program order
 - memory operations from different processors appear to be interleaved in some order at the memory
- Question:
 - If a processor is allowed to reorder independent operations **in its own** instruction stream, will the execution always produce the same results as the canonical model?
 - Answer: no. Let us look at some examples.

Example (I)

Code:

Initially A = Flag = 0

P1

```
A = 23;  
Flag = 1;
```

P2

```
while (Flag != 1) {;  
    ... = A;
```

Idea:

- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits until Flag is set and then reads data from A.

Execution Sequence for (I)

Code:

Initially $A = \text{Flag} = 0$

P1

$A = 23;$
 $\text{Flag} = 1;$

P2

$\text{while} (\text{Flag} \neq 1) \{;$
 $\dots = A;$

Possible execution sequence on each processor:

P1

Write A 23
Write Flag 1

P2

Read Flag //get 0
.....
Read Flag //get 1
Read A //what do you get?

Problem: If the two writes on processor P1 can be reordered, it is possible for processor P2 to read 0 from variable A.
Can happen on most modern processors.

Example (II)

Code: (like Dekker's algorithm)

Initially Flag1 = Flag2 = 0

P1

Flag1 = 1;

If (Flag2 == 0)

critical section;

P2

Flag2 = 1;

If (Flag1 == 0)

critical section;

Possible execution sequence on each processor:

P1

Write Flag1, 1

Read Flag2 //get 0

P2

Write Flag2, 1

Read Flag1 //what do you get?

Execution sequence for (II)

Code: (like Dekker's algorithm)

Initially Flag1 = Flag2 = 0

P1

```
Flag1 = 1;  
If (Flag2 == 0)  
    critical section;
```

P2

```
Flag2 = 1;  
If (Flag1 == 0)  
    critical section;
```

Possible execution sequence on each processor:

P1

```
Write Flag1, 1  
Read Flag2 //get 0
```

P2

```
Write Flag2, 1  
Read Flag1 //what do you get?
```

- Most people would say that P2 will read 1 as the value of Flag1.
 - Since P1 reads 0 as the value of Flag2, P1's read of Flag2 must happen before P2 writes to Flag2.
 - Intuitively, we would expect P1's write of Flag1 to happen before P2's read of Flag1.
- However, this is true only if reads and writes on the same processor to different locations are not reordered by the compiler or the hardware.
 - Unfortunately, this is very common on most processors (store-buffers with load-bypassing).

Lessons

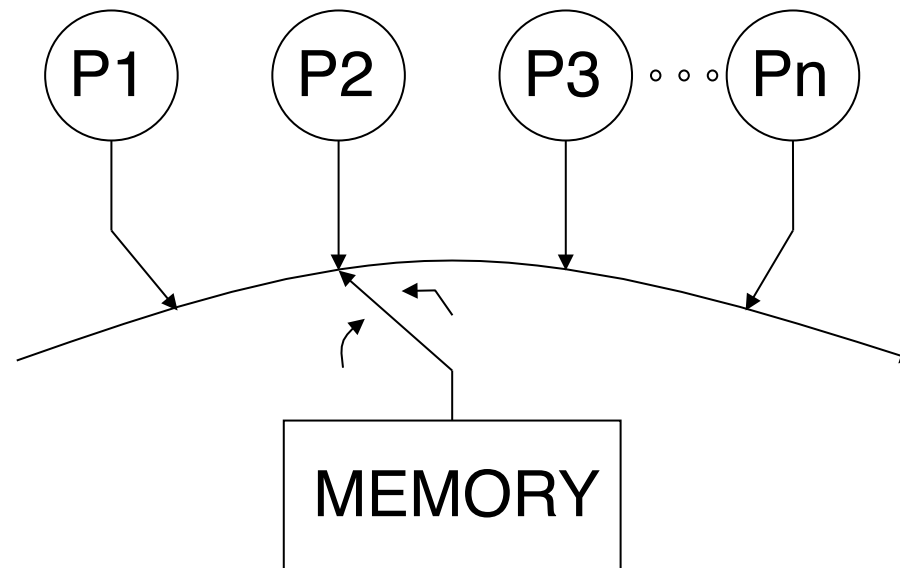
- Uniprocessors can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory context
 - simple parallel programs may produce counter-intuitive results
- Question: what constraints must we put on uniprocessor instruction reordering so that
 - shared-memory programming is intuitive
 - but we do not lose uniprocessor performance?
- Many answers to this question
 - answer is called **the memory consistency model** supported by the processor

Consistency Models

- Consistency models are **not** about memory operations **from different processors**.
- Consistency models are **not** about **dependent memory operations in a single processor's** instruction stream (these are respected even by processors that reorder instructions).
- Consistency models are all about ordering constraints on **independent memory operations in a single processor's** instruction stream that have **some high-level dependence** (such as flags guarding data) that should be respected to obtain intuitively reasonable results.

Sequential consistency [Lamport]

- Is the simplest Memory Consistency Model
 - our canonical model: processor is not allowed to reorder reads and writes to global memory

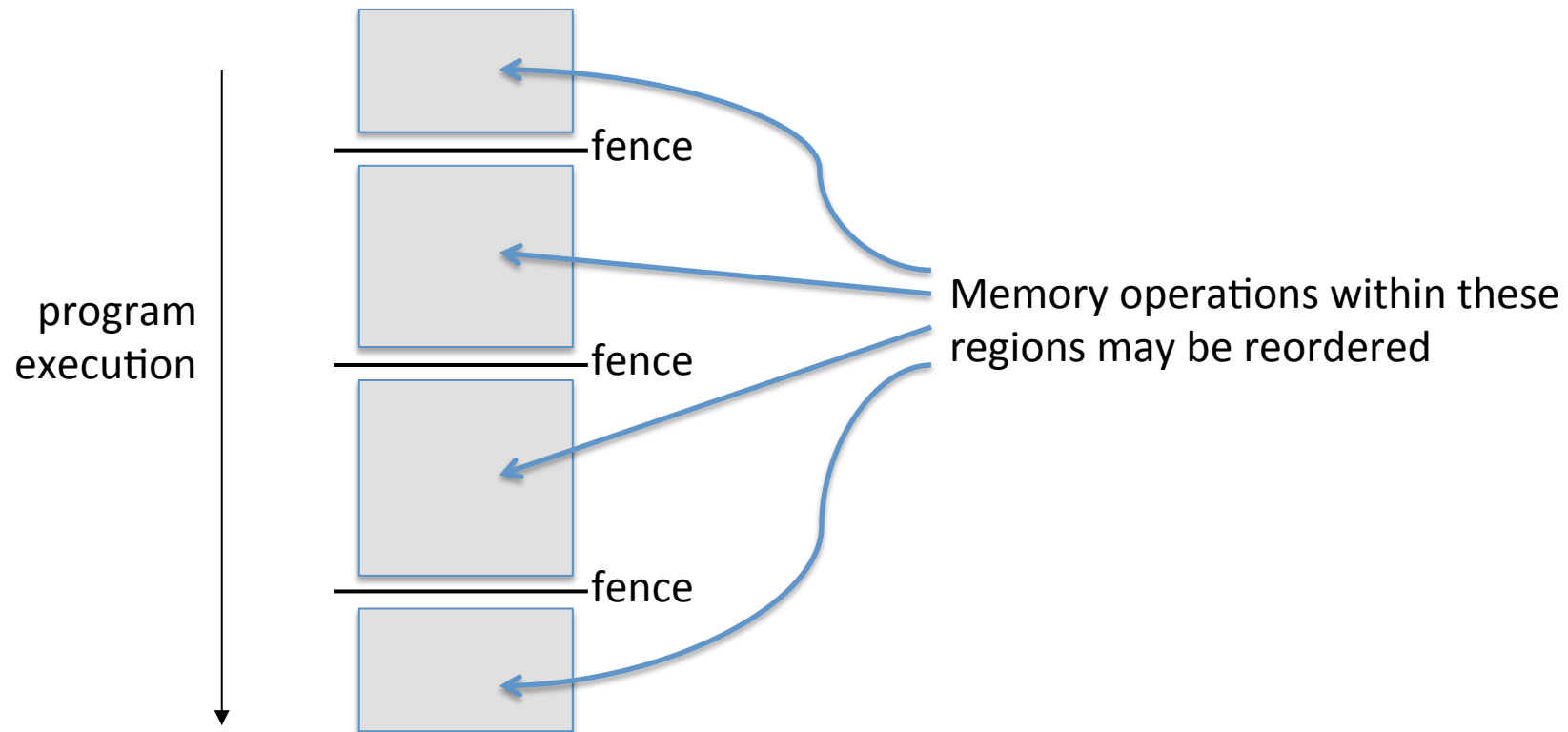


Sequential Consistency

- SC constrains all memory operations:
 - Write \rightarrow Read \rightarrow “is not reordered after”
 - Write \rightarrow Write
 - Read \rightarrow Read, Write
- Simple model for reasoning about parallel programs
 - You can verify that the examples considered earlier work correctly under sequential consistency.
- However, this simplicity comes at the cost of uniprocessor performance.
- **Question: how do we reconcile sequential consistency model with the demands of performance?**

Weak ordering picture

- Programmer specifies regions within which global memory operations can be reordered



Relaxed consistency model: Weak consistency

- Programmer specifies regions within which global memory operations can be reordered
- Processor has **fence** instruction:
 - all data operations **before** fence in program order must complete before fence is executed
 - all data operations **after** fence in program order must wait for fence to complete
 - fences 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
- Examples:
 - PowerPC has **SYNC** instruction, x8 has **mfence**
 - Language constructs: OpenMP has **flush**
- All synchronization operations like lock and unlock act like a fence

Example (I) revisited

Code:

Initially A = Flag = 0

P1

```
A = 23;  
flush;  
Flag = 1;
```

P2

```
while (Flag != 1) {;}  
... = A;
```

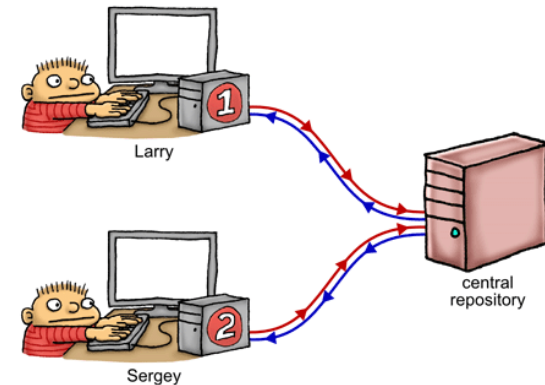
Does P2 need a flush between the two statements?

Execution:

- P1 writes data into A
- Flush waits till write to A is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of A even if memory operations in P1 before flush and memory operations after flush are reordered by the hardware or compiler.

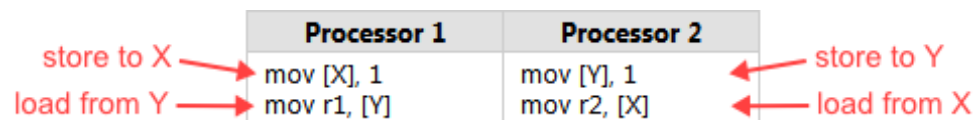
The Control Version Analogy (1)

- The source control strategy is very strange.
 - As Larry and Sergey modify their working copies of the repository, their modifications are constantly leaking in the background, to and from the central repository, at totally random times.
- Once Larry edits the file X, his change will leak to the central repository, but there's no guarantee about when it will happen.
 - It might happen immediately, or it might happen much, much later.
- He might go on to edit other files, say Y and Z, and those modifications might leak into the repository before X gets leaked.
 - In this manner, stores are effectively reordered on their way to the repository.
- Similarly, on Sergey's machine, there's no guarantee about the timing or the order in which those changes leak back from the repository into his working copy.
 - In this manner, loads are effectively reordered on their way out of the repository.

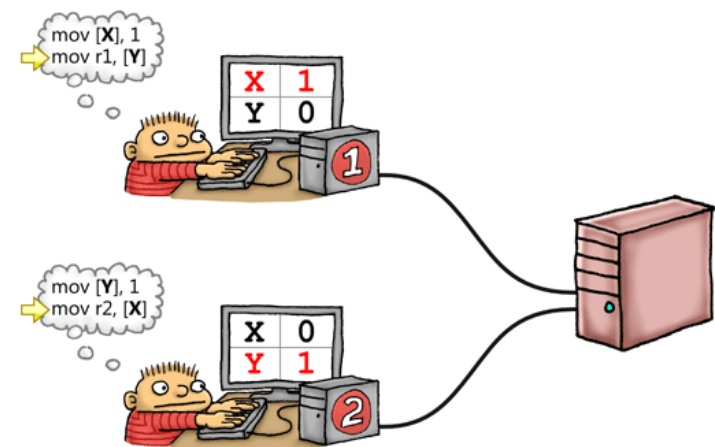


The Control Version Analogy (2)

- If Larry and always Sergey work on different files
 - No problem will arise and the model follows the canonical model
- If they work on the same parts of the repository




- Ends with “r1 = r2 = 0”



Memory Barriers

- In abstract we can define 4 types of memory barriers

#LoadLoad	#LoadStore
 #StoreLoad	#StoreStore

- Processors (and some languages) implement a subset or variants of these barriers

#LoadLoad

- Prevents reordering of loads performed before the barrier with loads performed after the barrier.



- Equivalent to “git pull” or “svn update”
 - There’s no guarantee that #LoadLoad will pull the latest, or head, revision of the entire repository!
 - It may pull an older revision than the head, as long as that revision is *at least as new as the newest value which leaked from the central repository into his local machine.*

#LoadLoad

```
if (IsPublished)           // Load and check shared flag
{
    LOADLOAD_FENCE();       // Prevent reordering of loads
    return Value;           // Load published value
}
```

- This example depends on having the *IsPublished* flag leak into Sergey's working copy by itself.
 - It doesn't matter exactly when that happens
- Once the leaked flag has been observed, he issues a #LoadLoad fence to prevent reading some value of Value which is older than the flag itself.

#StoreStore

- Prevents reordering of stores performed before the barrier with stores performed after the barrier.



- Performed in a delayed, asynchronous manner.
 - So, even though Larry executes a #StoreStore, we can't make any assumptions about when all his previous stores finally become visible in the central repository.

#StoreStore

```
Value = x;           // Publish some data
STORESTORE_FENCE();
IsPublished = 1;     // Set shared flag to indicate availability of data
```

- We are counting on the value of *IsPublished* to leak from Larry's working copy over to Sergey's, all by itself.
- Once Sergey detects that, he can be confident he'll see the correct value of *Value*.
 - What's interesting is that, for this pattern to work, *Value* does not even need to be an atomic type; it could just as well be a huge structure with lots of elements..

#LoadStore

- Imagine Larry has a set of instructions to follow.
 - Some instructions make him load data from his private working copy into a register, and some make him store data from a register back into the working copy.
- Larry has the ability to juggle instructions, but only in specific cases.
 - Whenever he encounters a load, he looks ahead at any stores that are coming up after that; if the stores are completely unrelated to the current load, then he's allowed to skip ahead, do the stores first, then come back afterwards to finish up the load.
 - In such cases, the cardinal rule of memory ordering – never modify the behavior of a single-threaded program – is still followed.



#LoadStore

- It is valid for Larry to perform this kind of LoadStore reordering even when there is a #LoadLoad or #StoreStore barrier between the load and the store.
 - However, on a real CPU, instructions that act as a #LoadStore barrier typically act as at least one of those other two barrier types.
- On a real CPU, such instruction reordering might happen on certain processors if, say, there is a cache miss on the load followed by a cache hit on the store.



#StoreLoad

- Could be achieved by pushing all local changes to the central repository, waiting for that operation to complete, then pulling the absolute latest head revision of the repository.
- Different from “#StoreStore” + “#LoadLoad”
 - The push operation may be delayed for an arbitrary number of instructions.
 - The pull operation might not pull from the head revision.



#StoreLoad

- Ensures that...
 - all stores performed before the barrier are visible to other processors; and
 - all loads performed after the barrier receive the latest value that is visible at the time of the barrier.
- It effectively prevents reordering of all stores before the barrier against all loads after the barrier
 - Respecting the way a sequentially consistent multiprocessor would perform those operations.
 - Is the only type of memory barrier that will prevent the result “ $r1 = r2 = 0$ ” from the example before

#StoreLoad

- On most processors, instructions that act as a #StoreLoad barrier tend to be more expensive than instructions acting as the other barrier types.
- Followed by a “#LoadStore” barrier makes a full memory fence.

Example (weak consistency)

A == 1; B == 2

CPU 1

A = 3;
B = 4;

CPU 2

X = A;
Y = B;

Accesses as seen by the memory

STORE A=3	STORE B=4	x=LOAD A->3	y=LOAD B->4
STORE A=3	STORE B=4	y=LOAD B->4	x=LOAD A->3
STORE A=3	x=LOAD A->3	STORE B=4	y=LOAD B->4
STORE A=3	x=LOAD A->3	y=LOAD B->2	STORE B=4
STORE A=3	y=LOAD B->2	STORE B=4	x=LOAD A->3
STORE A=3	y=LOAD B->2	x=LOAD A->3	STORE B=4
STORE B=4	STORE A=3	x=LOAD A->3	y=LOAD B->4
STORE B=4	...		
...			

Possible results!

X = 1	Y = 2
X = 1	Y = 4
x = 3	Y = 2
X = 3	Y = 4

Example (Reordering)

- Imagine an ethernet card with a set of internal registers that are accessed through an address port register (A) and a data port register (D).
- To read internal register 5, the following code might be used:

CPU 1
*A = 5;
x = *D;

Reordering...

STORE *A = 5	x = LOAD *D
x = LOAD *D	STORE *A = 5

Guarantees (1)

- **On any given CPU, dependent memory accesses will be issued in order, with respect to itself.** This means that for:

$$Q = P; \quad D = *Q;$$

the CPU will issue the following memory operations:

$$Q = \text{LOAD } P, \quad D = \text{LOAD } *Q$$

and always in that order.

Guarantees (2)

- **Overlapping loads and stores within a particular CPU will appear to be ordered within that CPU.** This means that for:

$a = *X; \quad *X = b;$

the CPU will only issue the following sequence of memory operations:

$a = \text{LOAD } *X, \quad \text{STORE } *X = b$

and for:

$*X = c; \quad d = *X;$

the CPU will only issue:

$\text{STORE } *X = c, \quad d = \text{LOAD } *X$

(Loads and stores overlap if they are targeted at overlapping pieces of memory).

Must not be assumed

- Independent loads and stores will be issued in the order given. This means that for:

$X = *A; \quad Y = *B; \quad *D = Z;$

we may get any of the following sequences:

X = LOAD *A	Y = LOAD *B	STORE *D = Z
X = LOAD *A	STORE *D = Z	Y = LOAD *B
Y = LOAD *B	X = LOAD *A	STORE *D = Z
Y = LOAD *B	STORE *D = Z	X = LOAD *A
STORE *D = Z	X = LOAD *A	Y = LOAD *B
STORE *D = Z	Y = LOAD *B	X = LOAD *A

Must be assumed

- Overlapping memory accesses may be merged or discarded.

For this:

a) $X = *A;$ $Y = *(A + 4);$

we may get any of this:

a)

$X = \text{LOAD } *A$	$Y = \text{LOAD } *(A + 4)$
$Y = \text{LOAD } *(A + 4)$	$X = \text{LOAD } *A$
$\{X, Y\} = \text{LOAD } \{ *A, *(A + 4) \}$	

b) $*A = X;$ $*(A + 4) = Y;$

b)

$\text{STORE } *A = X$	$\text{STORE } *(A + 4) = Y$
$\text{STORE } *(A + 4) = Y$	$\text{STORE } *A = X$
$\text{STORE } \{ *A, *(A + 4) \} = \{X, Y\}$	

Memory barriers (1)

- Write (or store) memory barrier [wfence@x86]
 - Gives a guarantee that all the STORE operations specified before the barrier will appear to happen *before* all the STORE operations specified after the barrier with respect to the other components of the system.
 - Imposes a partial ordering on stores only; it is not required to have any effect on loads.
 - A CPU can be viewed as committing a sequence of store operations to the memory system as time progresses. All stores before a write barrier will occur in the sequence *before* all the stores after the write barrier.
 - [!] *Note that write barriers should normally be paired with read barriers.*

Memory barriers (2)

- Read (or load) memory barriers [lfence@x86]
 - Gives a guarantee that all the LOAD operations specified before the barrier will appear to happen before all the LOAD operations specified after the barrier with respect to the other components of the system.
 - Imposes a partial ordering on loads only; it is not required to have any effect on stores.
 - This makes program state exposed from other CPUs visible to this CPU before making further progress.
 - A CPU can be viewed as updating a sequence of values from the memory system as time progresses. All loads before a load barrier will occur in the sequence *before* all the loads after the write barrier.
 - *[!]* Note that read barriers should normally be paired with write barriers.

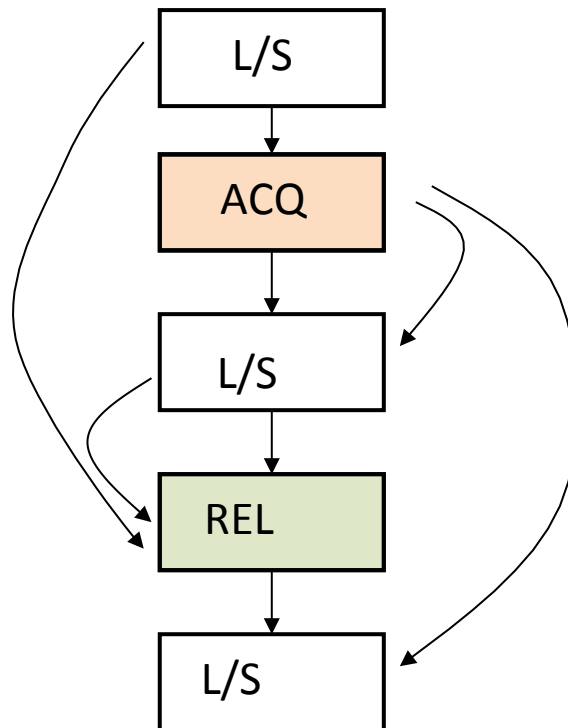
Memory barriers (3)

- General memory barrier [mfence@x86]
 - Gives a guarantee that all the LOAD and STORE operations specified before the barrier will appear to happen before all the LOAD and STORE operations specified after the barrier with respect to the other components of the system.
 - Imposes a partial ordering over both loads and stores.
 - Imply both read and write memory barriers, and so can substitute for either (but is slower).

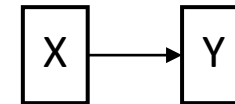
Another relaxed model: release consistency

- 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
- However,
 - Acquire does not wait for accesses preceding it
 - 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

Example



Which operations can be overlapped?



"X" must precede "Y"
"Y" must succeed "X"

Semantics of acquire:

*Acquire must complete before all
following memory accesses*

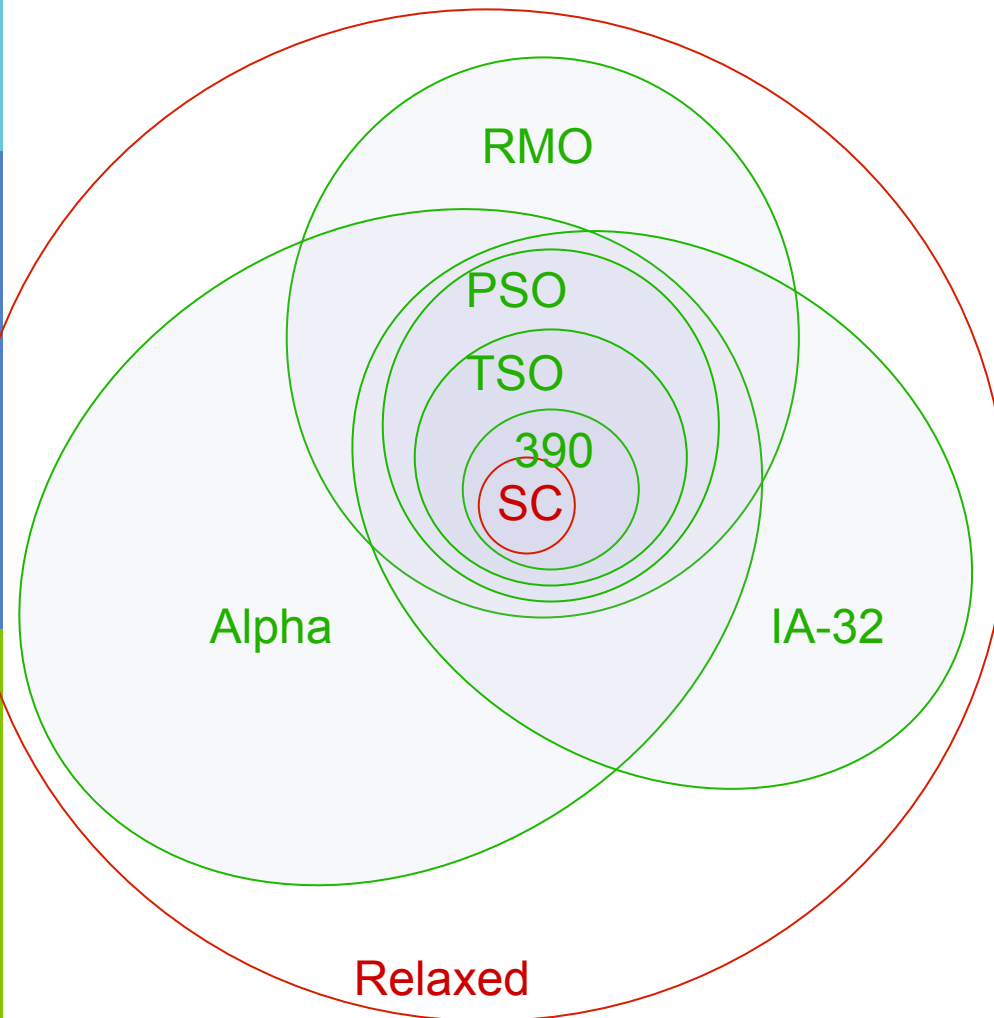
Semantics of release:

*All memory operations before release
are complete*

Some Current System-Centric Models

Relaxation	different mem. loc.				same mem. loc.		Safety Net
	W → R Order	W → W Order	R → R Order	R → W Order	Read Others' Write Early	Read Own Write Early	
IBM 370	✓						Serialization of instructions
Sparc TSO	✓					✓	RMW
Sparc PSO	✓	✓				✓	RMW, STBAR
Sparc RMO	✓	✓	✓	✓		✓	various MEMBARs
IA-32 (x86)	✓		✓		?	?	rfence, wfence, mfence
IA-64	✓	✓	✓	✓	?	?	rfence, wfence, mfence
AMD64	✓		✓		?	?	rfence, wfence, mfence
Alpha	✓	✓	✓	✓		✓	MB, WMB
PowerPC	✓	✓	✓	✓	✓	✓	SYNC

Which Memory Model?



- Memory models are platform dependent
- We may use a conservative approximation “Relaxed” to capture common effects
- Once code is correct for “Relaxed”, it is correct for many models

Comments

- In the literature, there are a large number of other consistency models
 - processor consistency
 - total store order (TSO)
 -
- It is important to remember that these are concerned with reordering of independent memory operations **within** a processor.
- Easy to come up with shared-memory programs that behave differently for each consistency model.
- Emerging consensus that weak/release consistency is adequate.

Summary

- Two problems: memory consistency and memory coherence
- Memory consistency model
 - what instructions is compiler or hardware allowed to reorder?
 - nothing really to do with memory operations from different processors/threads
 - sequential consistency: perform global memory operations in program order
 - relaxed consistency models: all of them rely on some notion of a fence operation that demarcates regions within which reordering is permissible
- Memory coherence
 - preserve the illusion that there is a single logical memory location corresponding to each program variable even though there may be lots of physical memory locations where the variable is stored

The End
