CS510 Concurrent Systems

Jonathan Walpole

Portland State
UNIVERSITY

Shared Memory Consistency Models: A Tutorial

Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion

Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Flag1 = 1

Flag2 = 0

Dekker's Algorithm

Process 1::

Flag1 = 1

If (Flag2 == 0)

critical section

Process 2::

Flag2 = 1

If (Flag1 == 0)

critical section

Flag1 = 1

Flag2 = 0

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Flag1 = 1

Flag2 = 0

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::

Flag1 = 1

If (Flag2 == 0)

critical section

Process 2::

Flag2 = 1

If (Flag1 == 0)

critical section

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Critical section is protected!

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Flag1 = 1

Flag2 = 0

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::

Flag1 = 1

If (Flag2 == 0)

critical section

Process 2::

Flag2 = 1

If (Flag1 == 0)

critical section

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Both processes can block, but the critical section is still protected!

Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion

Write Buffer With Bypass

SpeedUp:

- Write takes 100 cycles
- Buffering takes 1 cycle
- So Buffer and keep going!

Problem: Read from a location with a buffered write pending?

Dekker's Algorithm

Process 1::

Flag1 = 1

If (Flag2 == 0)

critical section

Process 2::

Flag2 = 1

If (Flag1 == 0)

critical section

Flag1 = 0

Flag2 = 0

Flag1 = 1

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Flag1 = 0

Flag2 = 0

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0

Flag2 = 0

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0

Flag2 = 0

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0

Flag2 = 0

Flag1 = 1

Flag2 = 1

Critical section is not protected!

Write Buffer With Bypass

Rule:

- If a write is issued, buffer it and keep executing

Unless: there is a read from the same location (subsequent writes don't matter), then wait for the write to complete

Dekker's Algorithm

Process 1::

Flag1 = 1

If (Flag2 == 0)

critical section

Process 2::

Flag2 = 1

If (Flag1 == 0)

critical section

Flag1 = 0

Flag2 = 0

Flag1 = 1

Dekker's Algorithm

Process 1::	Process 2::
Flag1 = 1	**Flag2 = 1**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	*critical section*

Flag1 = 0

Flag2 = 0

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Stall!

Flag1 = 0

Flag2 = 0

Flag1 = 1

Flag2 = 1

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1

Flag2 = 0

Flag2 = 1

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1

Flag2 = 1

Is This a General Solution ?

- If each CPU has a write buffer with bypass, and follows the rules, will the algorithm still work correctly?

Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 0

Flag2 = 0

Dekker's Algorithm

Process 1::

Flag1 = 1

If (Flag2 == 0)

critical section

Process 2::

Flag2 = 1

If (Flag1 == 0)

critical section

Flag1 = 1

Flag1 = 0

Flag2 = 0

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1

Flag1 = 0

Flag2 = 0

Flag2 = 1

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1

Flag1 = 0

Flag2 = 0

Flag2 = 1

Dekker's Algorithm

Process 1::
Flag1 = 1
If (Flag2 == 0)
critical section

Process 2::
Flag2 = 1
If (Flag1 == 0)
critical section

Flag1 = 1

Flag1 = 0

Flag2 = 0

Flag2 = 1

Its Broken!

How did that happen?

- write buffers are processor specific
- writes are not visible to other processors until they hit memory

Generalization of the Problem

Dekker's algorithm has the form:

WX	WY
RY	RX

- The write buffer delays the writes until after the reads!
- It reorders the reads and writes
- Both processes can read the value prior to the other's write!

1	WX	RY	WY	RX
2	WX	RY	RX	WY
3	WX	WY	RY	RX
4	WX	RX	RY	WY
5	WX	WY	RX	RY
6	WX	RX	WY	RY
7	RY	WX	WY	RX
8	RY	WX	RX	WY
9	WY	WX	RY	RX
10	RX	WX	RY	WY
11	WY	WX	RX	RY
12	RX	WX	WY	RY
13	RY	WY	WX	RX
14	RY	RX	WX	WY
15	WY	RY	WX	RX
16	RX	RY	WX	WY
17	WY	RX	WX	RY
18	RX	WY	WX	RY
19	RY	WY	RX	WX
20	RY	RX	WY	WX
21	WY	RY	RX	WX
22	RX	RY	WY	WX
23	WY	RX	RY	WX
24	RX	WY	RY	WX

There are 4! or 24 possible orderings.

If *either* WX<RX or WY<RY
Then the Critical Section is protected (Correct Behavior).

1	WX	RY	WY	RX
2	WX	RY	RX	WY
3	WX	WY	RY	RX
4	WX	RX	RY	WY
5	WX	WY	RX	RY
6	WX	RX	WY	RY
7	RY	WX	WY	RX
8	RY	WX	RX	WY
9	WY	WX	RY	RX
10	RX	WX	RY	WY
11	WY	WX	RX	RY
12	RX	WX	WY	RY
13	RY	WY	WX	RX
14	RY	RX	WX	WY
15	WY	RY	WX	RX
16	RX	RY	WX	WY
17	WY	RX	WX	RY
18	RX	WY	WX	RY
19	RY	WY	RX	WX
20	RY	RX	WY	WX
21	WY	RY	RX	WX
22	RX	RY	WY	WX
23	WY	RX	RY	WX
24	RX	WY	RY	WX

There are 4! or 24 possible orderings.

If *either* WX<RX or WY<RY
Then the Critical Section is protected (Correct Behavior).

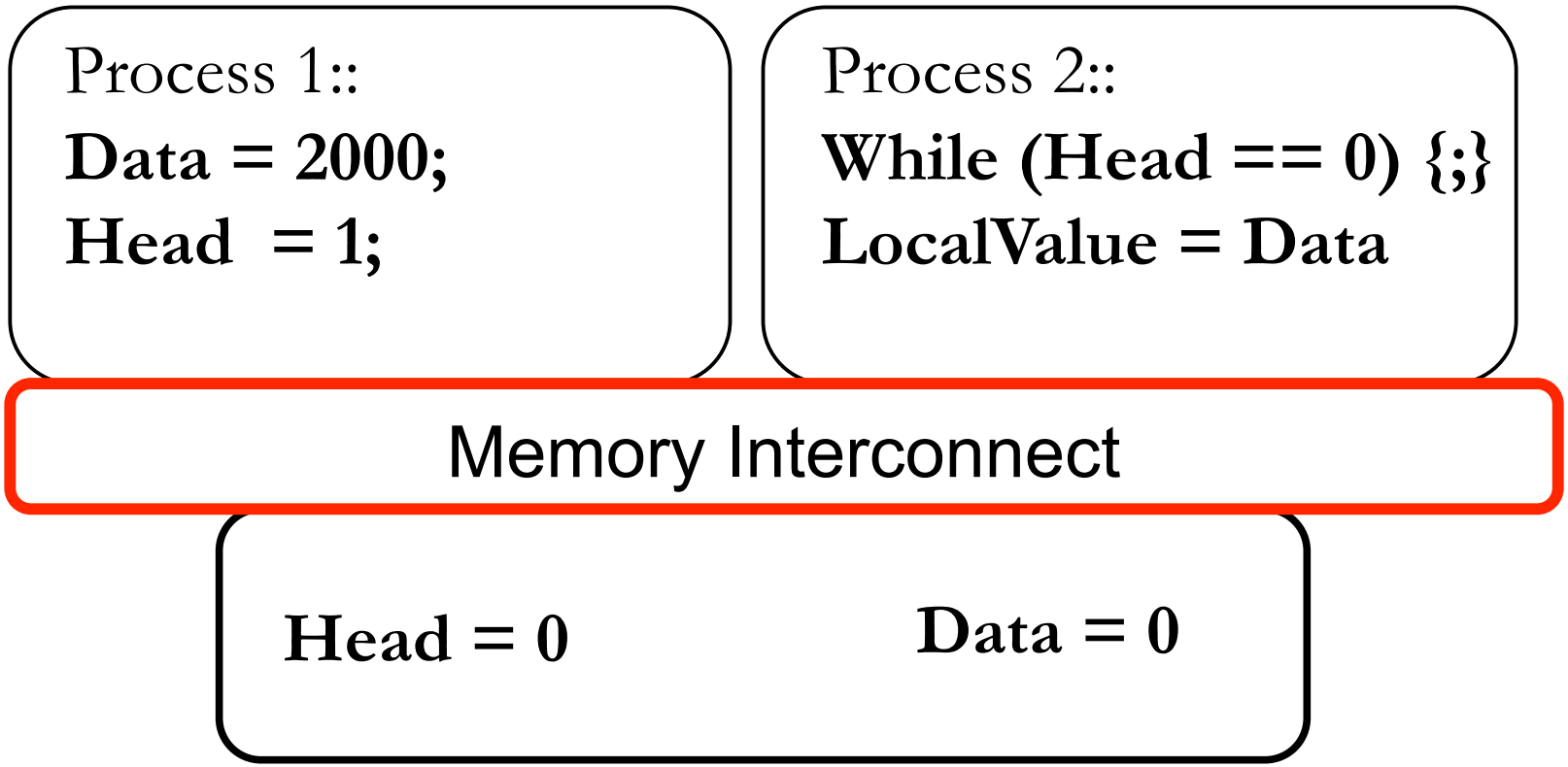
18 of the 24 orderings are OK.
But the other 6 are trouble!

Another Example

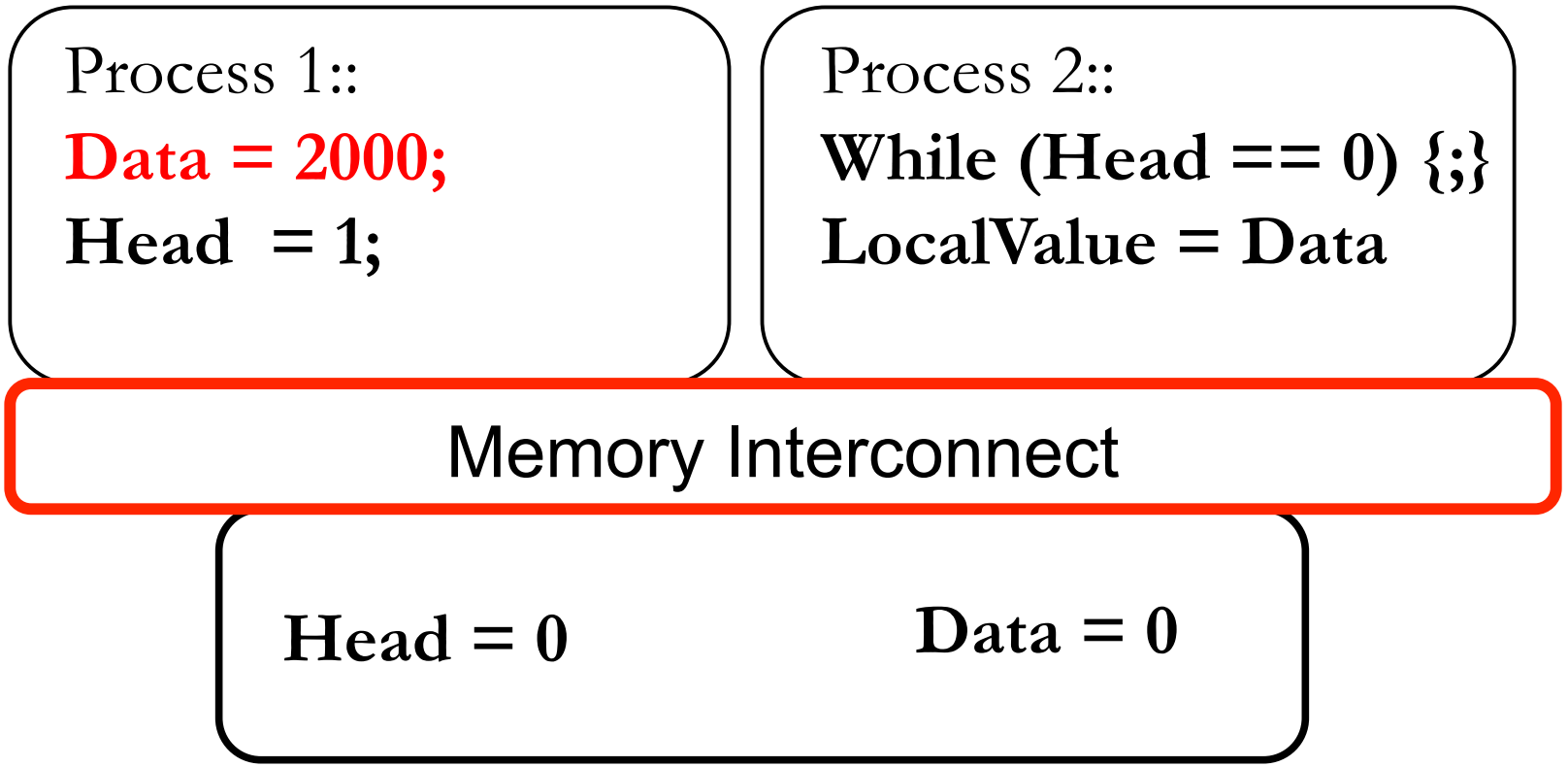
What happens if reads and writes can be delayed by the interconnect?

- non-uniform memory access time
- cache misses
- complex interconnects

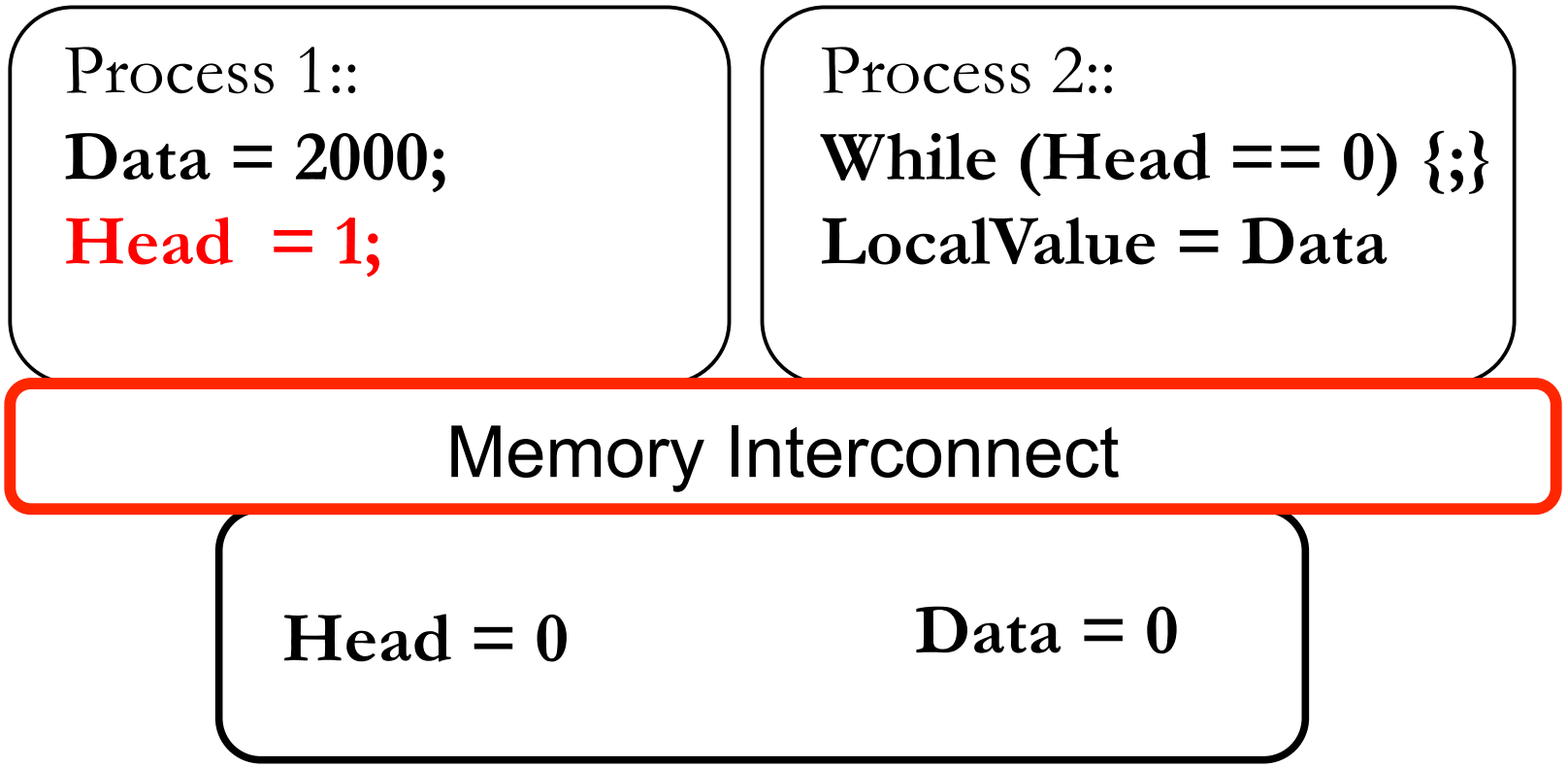
Non-Uniform Write Delays



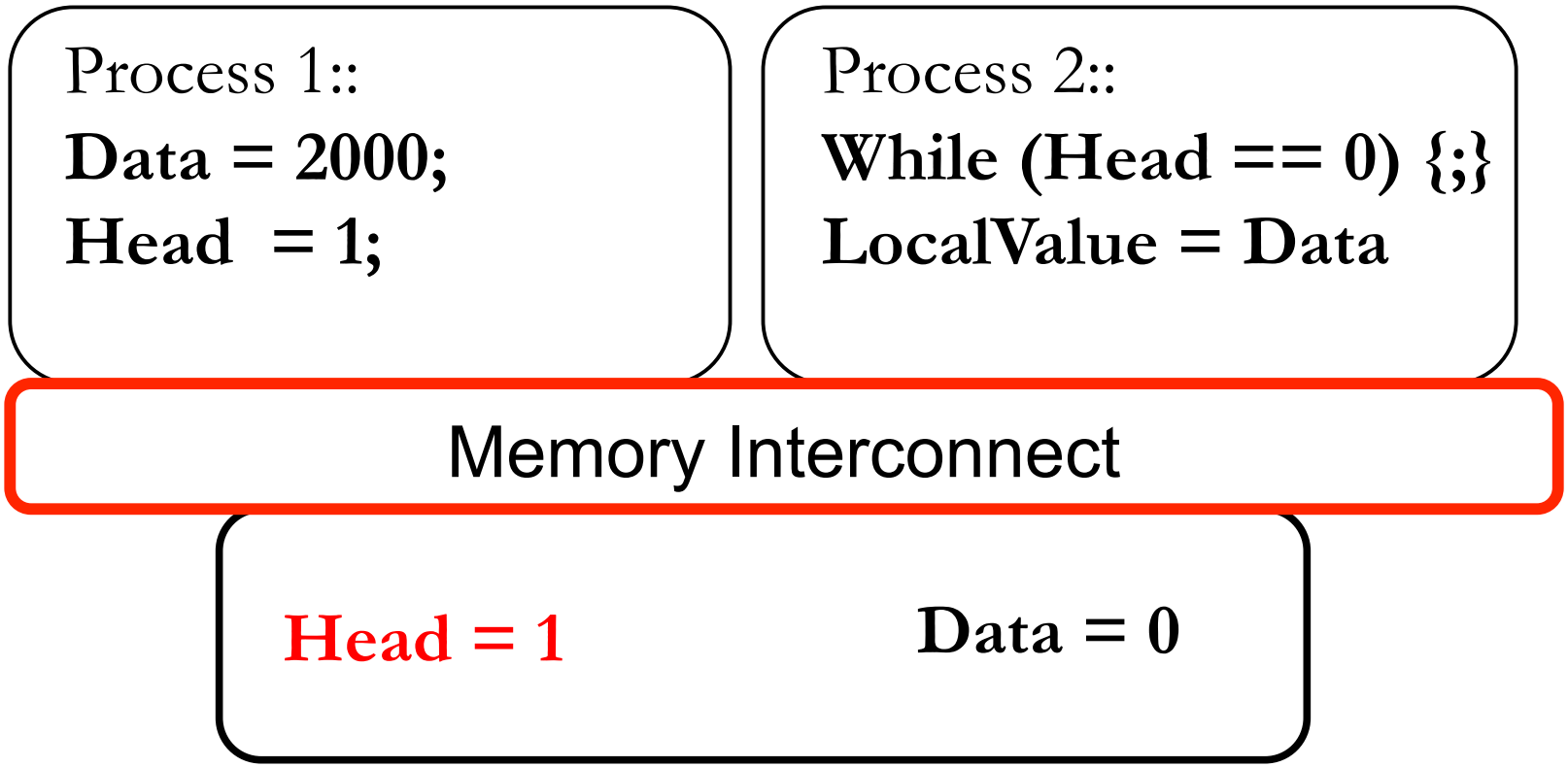
Non-Uniform Write Delays



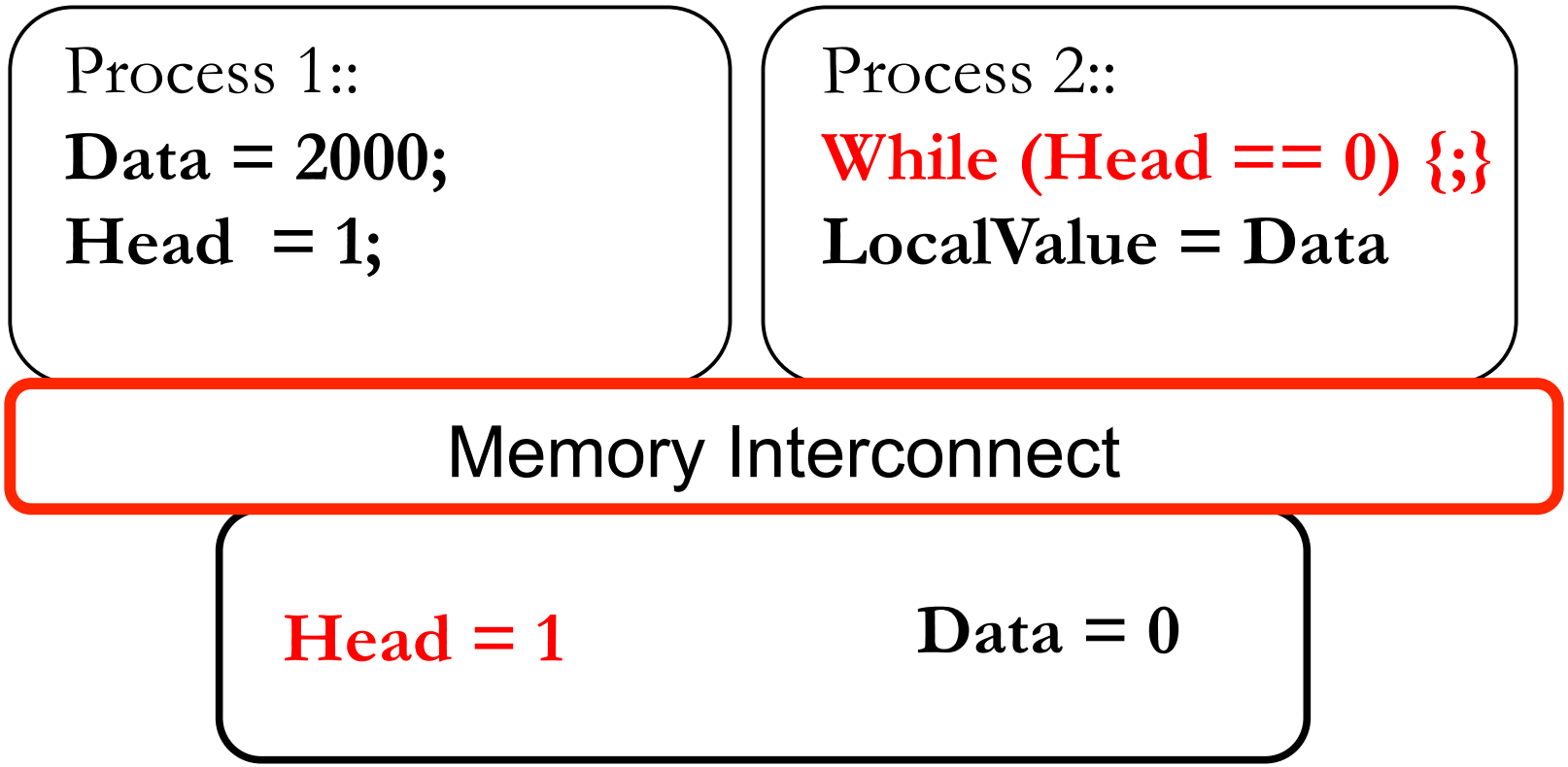
Non-Uniform Write Delays



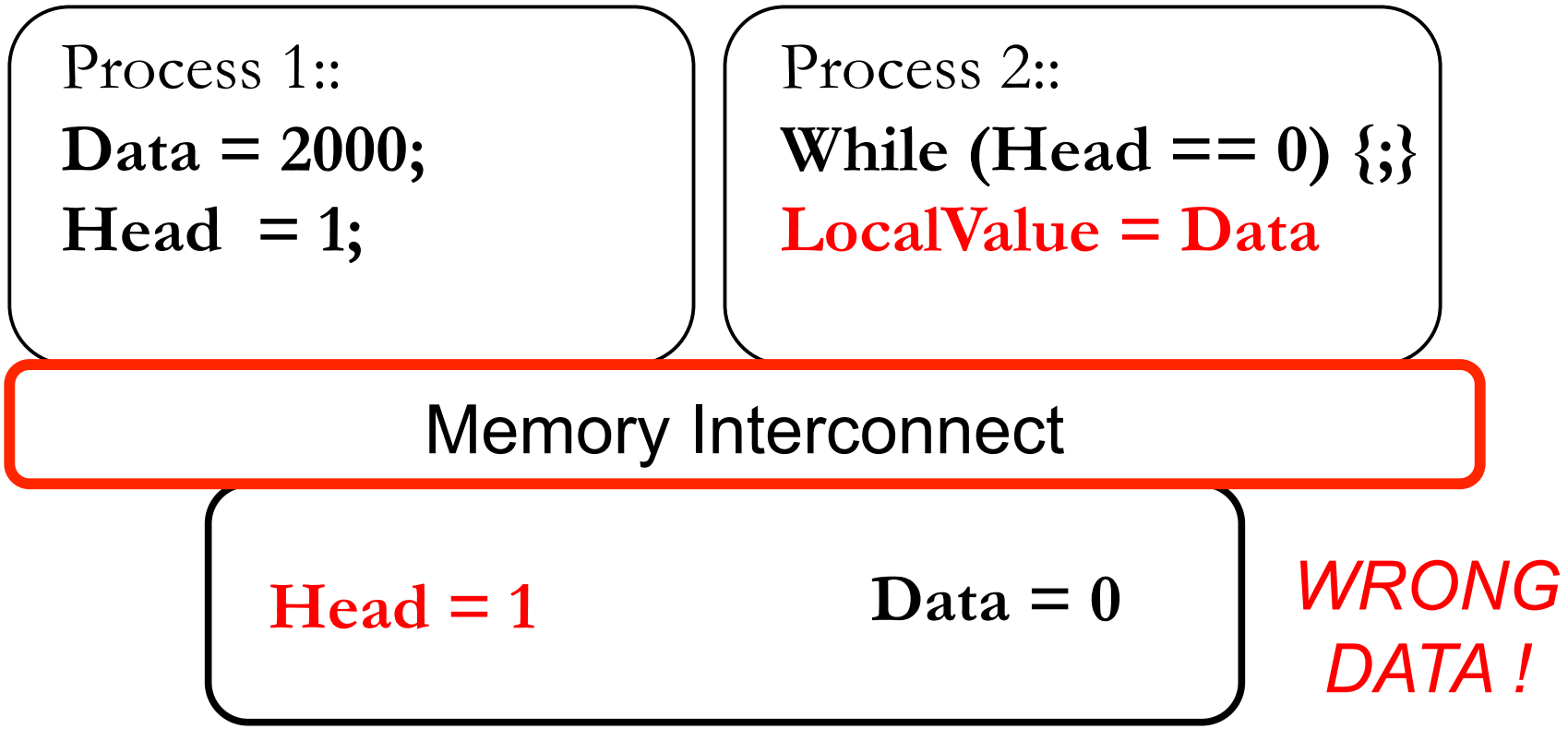
Non-Uniform Write Delays



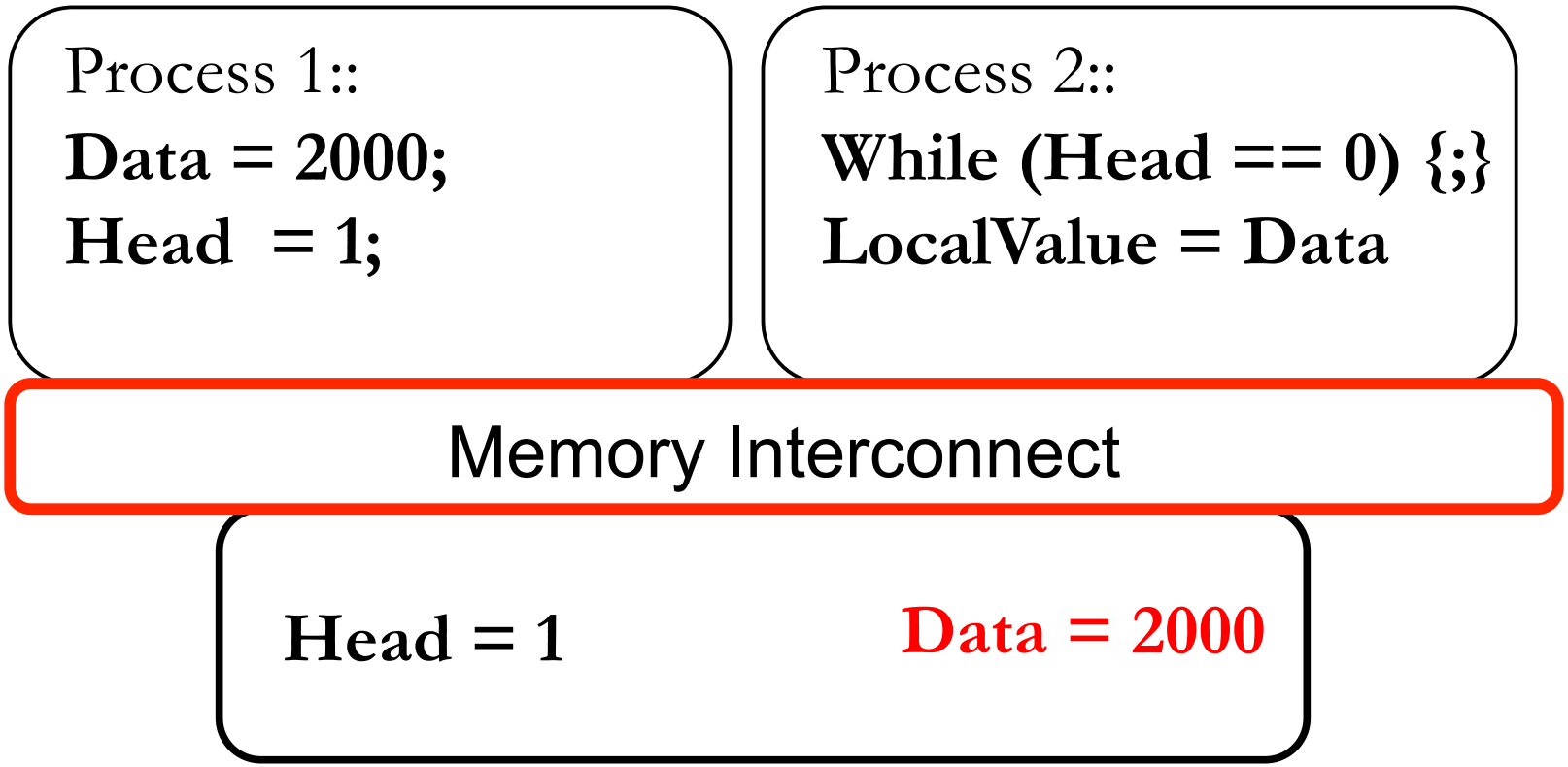
Non-Uniform Write Delays



Non-Uniform Write Delays



Non-Uniform Write Delays



What Went Wrong?

Maybe we need to acknowledge each write before proceeding to the next?

Write Acknowledgement?

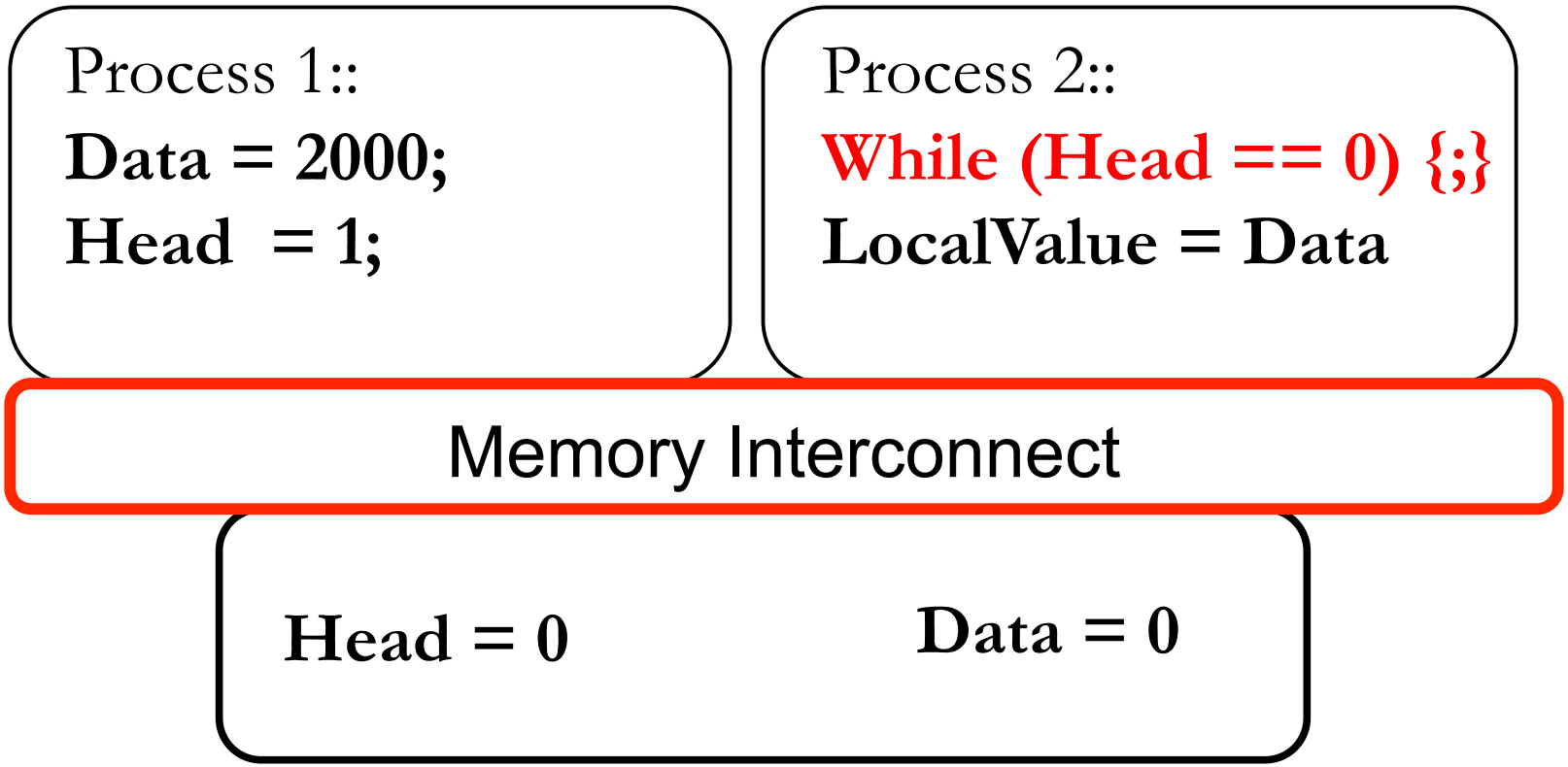
But what about reordering of reads?

- Non-Blocking Reads
- Lockup-free Caches
- Speculative execution
- Dynamic scheduling

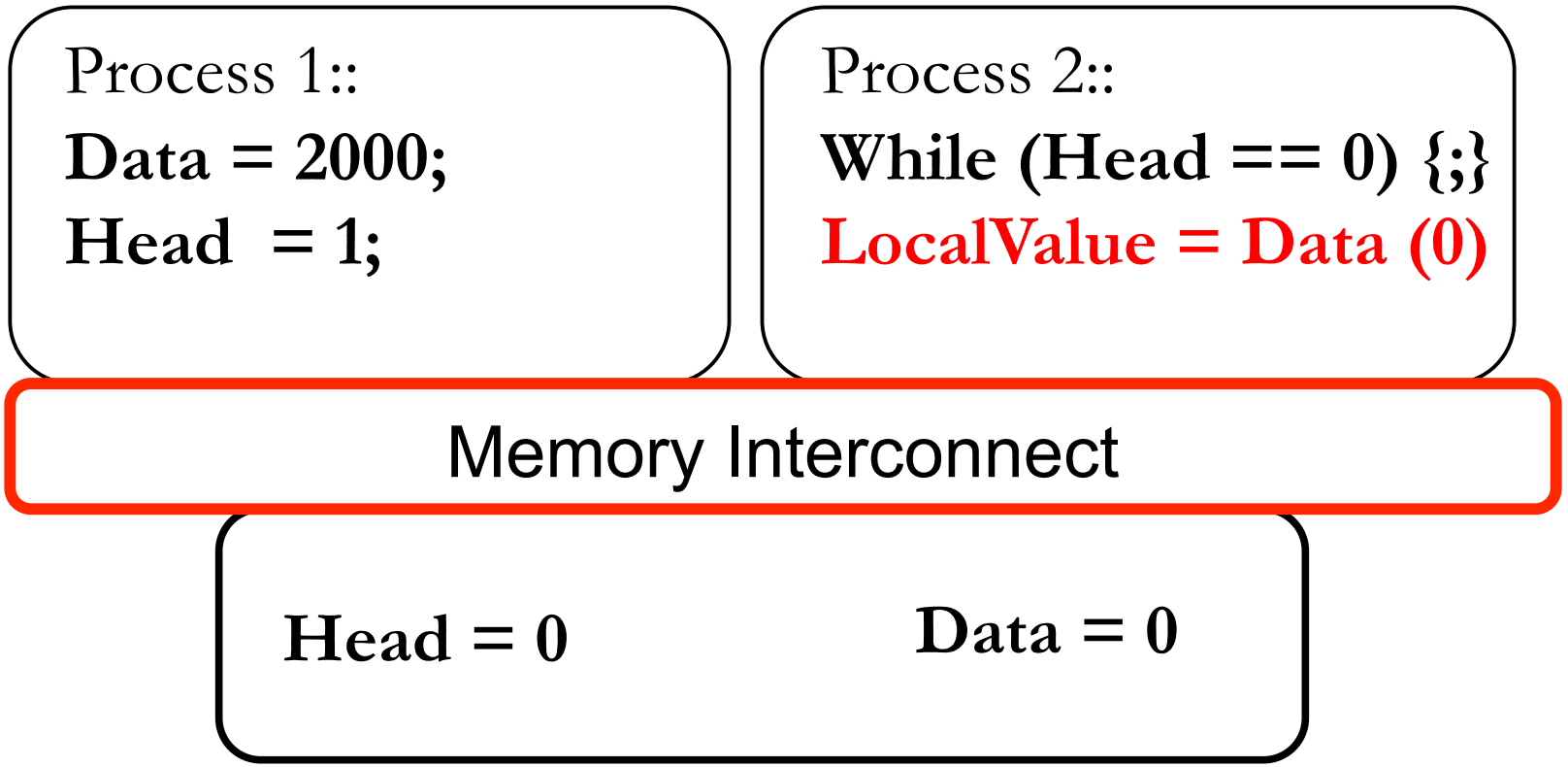
... all allow execution to proceed past a read

Acknowledging writes may not help!

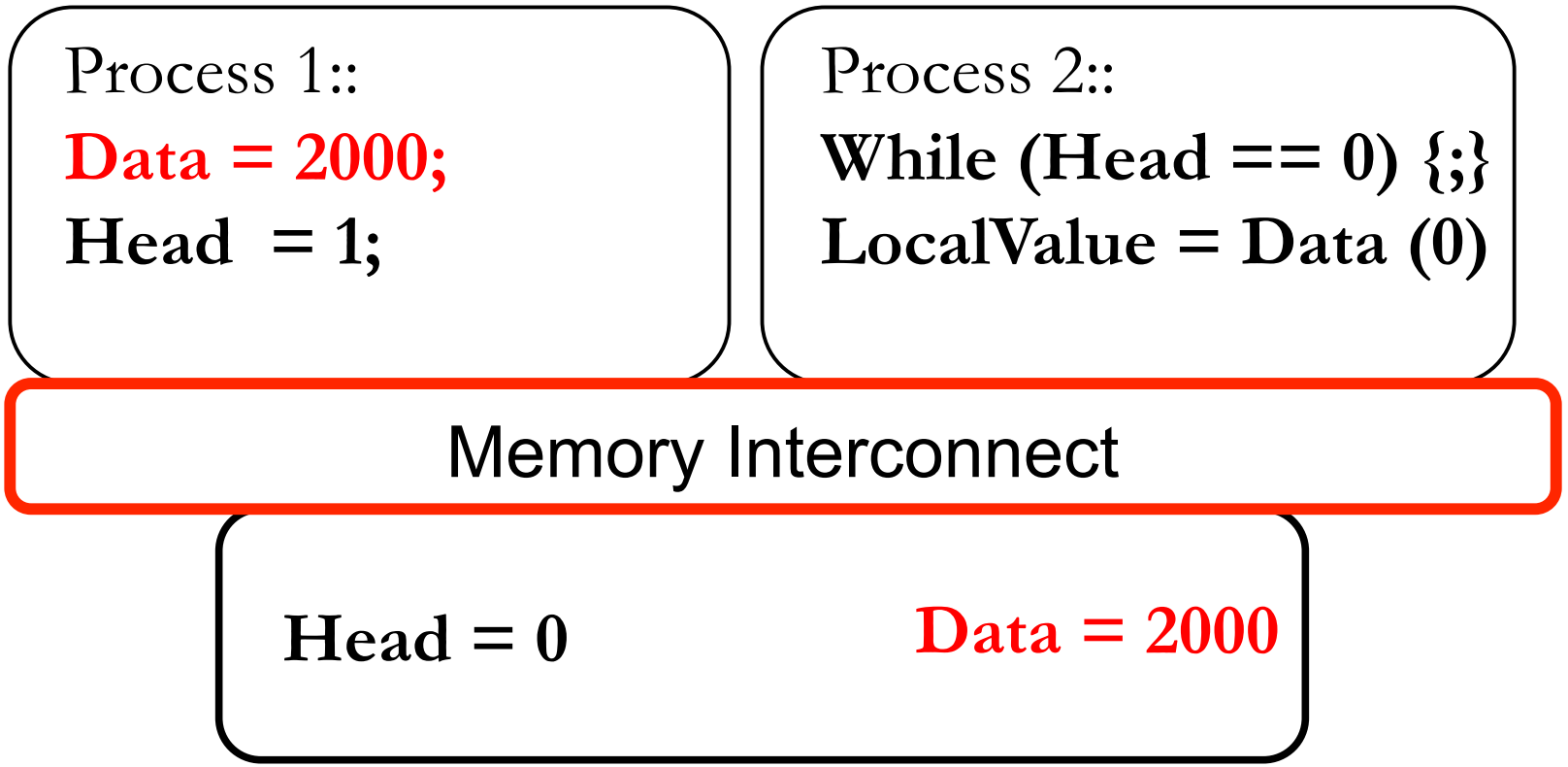
General Interconnect Delays



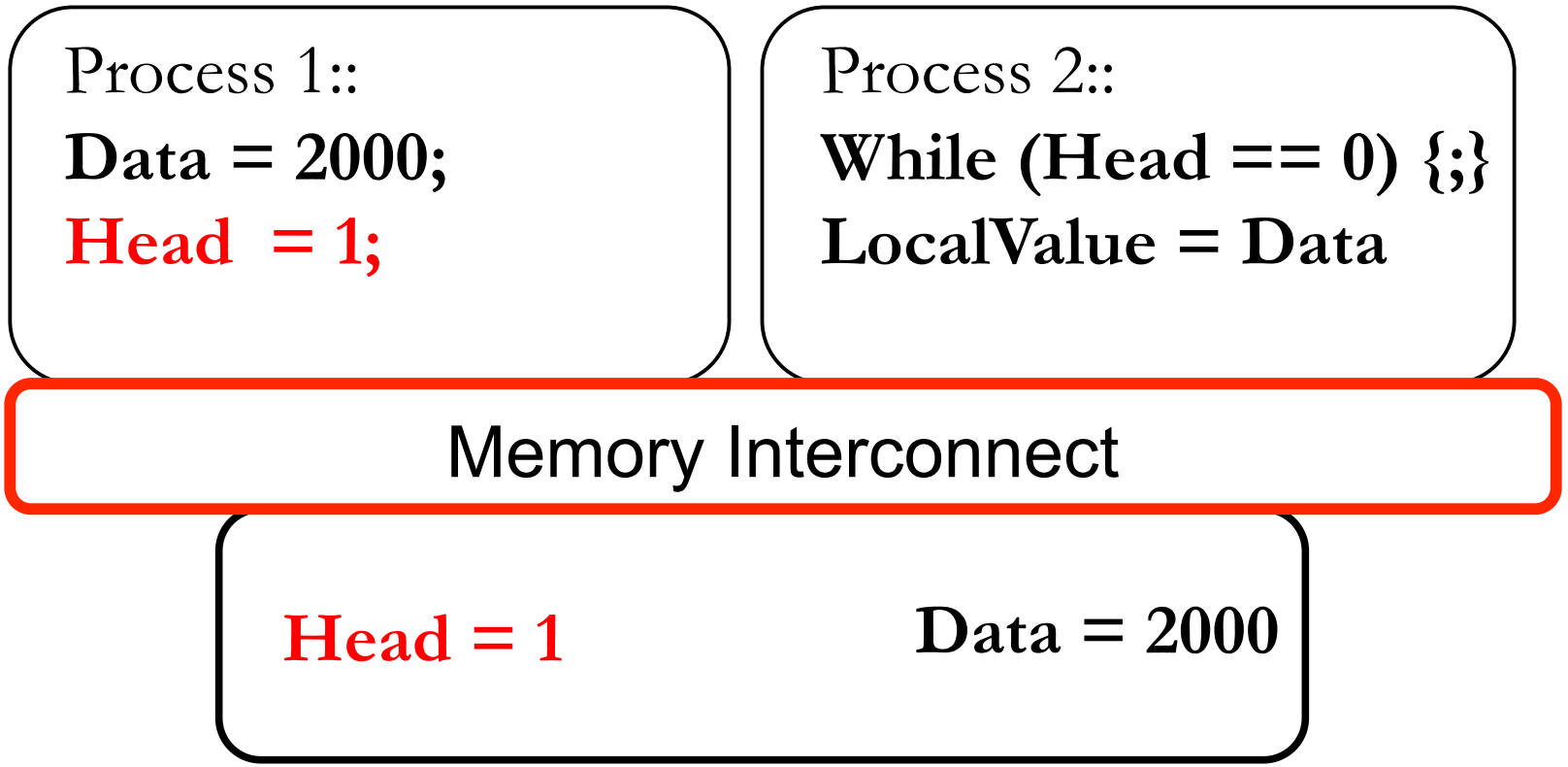
General Interconnect Delays



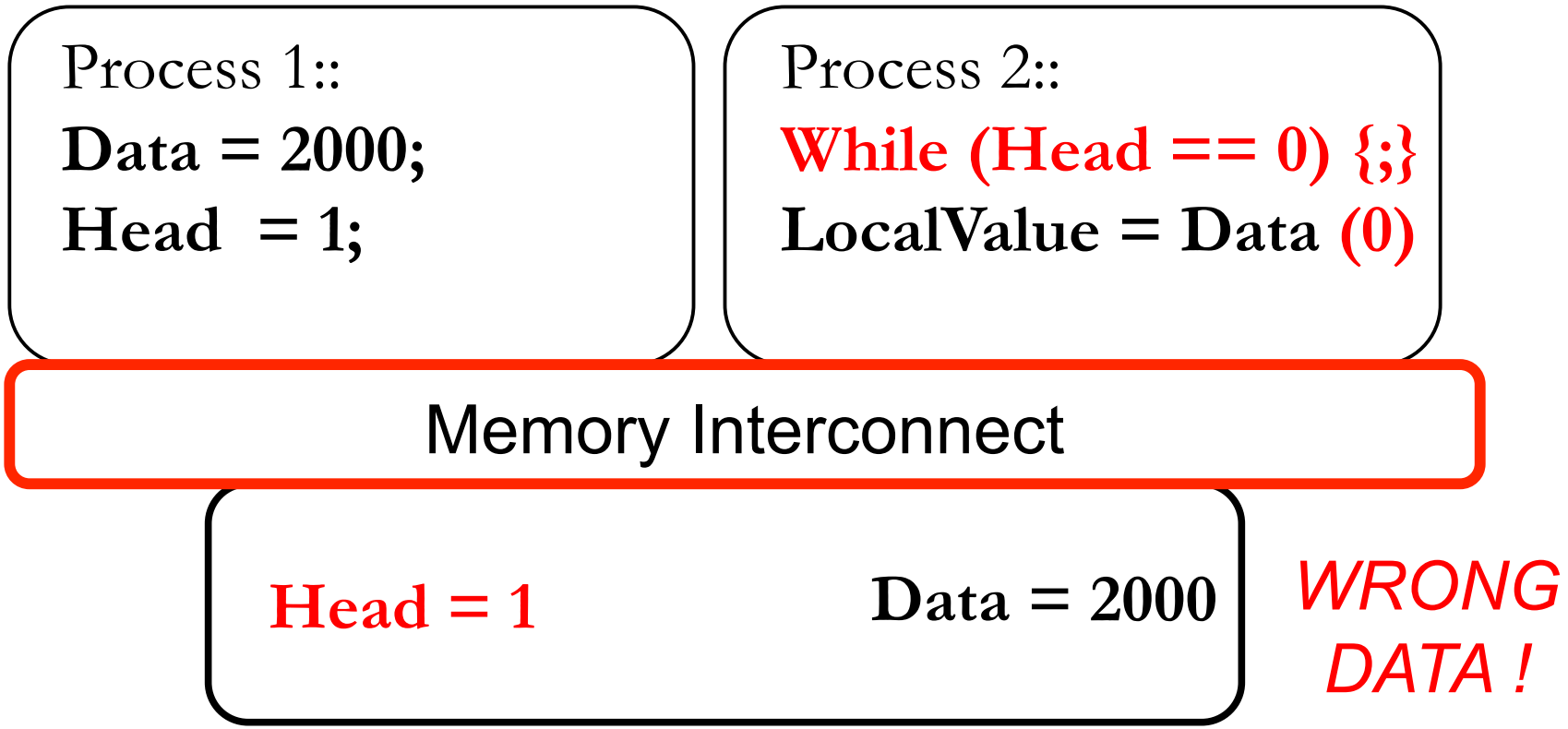
General Interconnect Delays



General Interconnect Delays



General Interconnect Delays



Generalization of the Problem

This algorithm has the form:

WX	RY
WY	RX

- The interconnect reorders reads and writes

1	WX	RY	WY	RX
2	WX	RY	RX	WY
3	WX	WY	RY	RX
4	WX	RX	RY	WY
5	WX	WY	RX	RY
6	WX	RX	WY	RY
7	RY	WX	WY	RX
8	RY	WX	RX	WY
9	WY	WX	RY	RX
10	RX	WX	RY	WY
11	WY	WX	RX	RY
12	RX	WX	WY	RY
13	RY	WY	WX	RX
14	RY	RX	WX	WY
15	WY	RY	WX	RX
16	RX	RY	WX	WY
17	WY	RX	WX	RY
18	RX	WY	WX	RY
19	RY	WY	RX	WX
20	RY	RX	WY	WX
21	WY	RY	RX	WX
22	RX	RY	WY	WX
23	WY	RX	RY	WX
24	RX	WY	RY	WX

Correct behavior requires WX<RX, WY<RY. Program requires WY<RX.
=> 6 correct orders out of 24.

1	WX	RY	WY	RX
2	WX	RY	RX	WY
3	WX	WY	RY	RX
4	WX	RX	RY	WY
5	WX	WY	RX	RY
6	WX	RX	WY	RY
7	RY	WX	WY	RX
8	RY	WX	RX	WY
9	WY	WX	RY	RX
10	RX	WX	RY	WY
11	WY	WX	RX	RY
12	RX	WX	WY	RY
13	RY	WY	WX	RX
14	RY	RX	WX	WY
15	WY	RY	WX	RX
16	RX	RY	WX	WY
17	WY	RX	WX	RY
18	RX	WY	WX	RY
19	RY	WY	RX	WX
20	RY	RX	WY	WX
21	WY	RY	RX	WX
22	RX	RY	WY	WX
23	WY	RX	RY	WX
24	RX	WY	RY	WX

Correct behavior requires WX<RX, WY<RY. Program requires WY<RX.
=> 6 correct orders out of 24.

Write Acknowledgment means WX < WY.
Does that Help?

Disallows only 12 out of 24.
9 still incorrect!

Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion

Sequential Consistency for MPs

Why is it surprising that these code examples break on a multi-processor?

What ordering property are we assuming (incorrectly!) that multiprocessors support?

We are assuming they are sequentially consistent!

Sequential Consistency

Sequential Consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and the accesses among different processors were interleaved arbitrarily.

...appears as if a memory operation executes atomically or instantaneously with respect to other memory operations

(Hennessy and Patterson, 4th ed.)

Understanding Ordering

Program Order

Compiled Order

Interleaving Order

Execution Order

Reordering

Writes reach memory, and reads see memory, in an order different than that in the program!

- Caused by Processor
- Caused by Multiprocessors (and Cache)
- Caused by Compilers

What Are the Choices?

If we want our results to be the same as those of a Sequentially Consistent Model. Do we:

- Enforce Sequential Consistency at the memory level?
- Use Coherent (Consistent) Cache ?
- Or what ?

Enforce Sequential Consistency?

Removes virtually all optimizations

Too slow!

What Are the Choices?

If we want our results to be the same as those of a Sequentially Consistent Model. Do we:

- Enforce Sequential Consistency at the memory level?
- Use Coherent (Consistent) Cache ?
- Or what ?

Cache Coherence

- Multiple processors have a consistent view of memory (i.e. MESI protocol)
- But this does not say ***when*** a processor must see a value updated by another processor.
- Cache coherency does not guarantee Sequential Consistency!
- Example: a write-through cache acts just like a write buffer with bypass.

What Are the Choices?

If we want our results to be the same as those of a Sequentially Consistent Model. Do we:

- Enforce Sequential Consistency at the memory level?
- Use Coherent (Consistent) Cache ?
- Or what ?

Involve the Programmer

Someone's got to tell your CPU about concurrency!

Use memory barrier / fence instructions when order really matters!

Memory Barrier Instructions

A way to prevent reordering

- Also known as a *safety net*
- Require previous instructions to complete before allowing further execution on that CPU

Not cheap, but perhaps not often needed?

- Must be placed by the programmer
- Memory consistency model for processor tells you what reordering is possible

Using Memory Barriers

Process 1::

Flag1 = 1

>>Mem_Bar<<

If (Flag2 == 0)

critical section

WX

>>Fence<<

RY

Fence: WX < RY

Process 2::

Flag2 = 1

>>Mem_Bar<<

If (Flag1 == 0)

critical section

WY

>>Fence<<

RX

Fence: WY < RX

1	WX	RY	WY	RX
2	WX	RY	RX	WY
3	WX	WY	RY	RX
4	WX	RX	RY	WY
5	WX	WY	RX	RY
6	WX	RX	WY	RY
7	RY	WX	WY	RX
8	RY	WX	RX	WY
9	WY	WX	RY	RX
10	RX	WX	RY	WY
11	WY	WX	RX	RY
12	RX	WX	WY	RY
13	RY	WY	WX	RX
14	RY	RX	WX	WY
15	WY	RY	WX	RX
16	RX	RY	WX	WY
17	WY	RX	WX	RY
18	RX	WY	WX	RY
19	RY	WY	RX	WX
20	RY	RX	WY	WX
21	WY	RY	RX	WX
22	RX	RY	WY	WX
23	WY	RX	RY	WX
24	RX	WY	RY	WX

There are 4! or 24 possible orderings.

If either WX<RX or WY<RY
Then the Critical Section is protected
(Correct Behavior)

18 of the 24 orderings are OK.
But the other 6 are trouble!

Enforce WX<RY and WY<RX.

Only 6 of the 18 good orderings are allowed OK.
But the 6 bad ones are still forbidden!

Example 2

Process 1::

Data = 2000;

>>Mem_Bar<<

Head = 1;

WX

>>Fence<<

WY

Fence: WX < WY

Process 2::

While (Head == 0) {;}

>>Mem_Bar<<

LocalValue = Data

RY

>>Fence<<

RX

Fence: RY < RX

1	WX	RY	WY	RX
2	WX	RY	RX	WY
3	WX	WY	RY	RX
4	WX	RX	RY	WY
5	WX	WY	RX	RY
6	WX	RX	WY	RY
7	RY	WX	WY	RX
8	RY	WX	RX	WY
9	WY	WX	RY	RX
10	RX	WX	RY	WY
11	WY	WX	RX	RY
12	RX	WX	WY	RY
13	RY	WY	WX	RX
14	RY	RX	WX	WY
15	WY	RY	WX	RX
16	RX	RY	WX	WY
17	WY	RX	WX	RY
18	RX	WY	WX	RY
19	RY	WY	RX	WX
20	RY	RX	WY	WX
21	WY	RY	RX	WX
22	RX	RY	WY	WX
23	WY	RX	RY	WX
24	RX	WY	RY	WX

Correct behavior requires WX<RX, WY<RY. Program requires WY<RX.
=> 6 correct orders out of 24.

We can require WX<WY and RY<RX. Is that enough?
Program requires **WY<RX**.
Thus, WX<WY<RY<RX; hence **WX<RX and WY<RY**.

Only 2 of the 6 good orderings are allowed -
But all 18 incorrect orderings are forbidden.

Memory Consistency Models

Every CPU architecture has one!

- It explains what reordering of memory operations that CPU can do

The CPUs instruction set contains memory barrier instructions of various kinds

- These can be used to constrain reordering where necessary
- The programmer must understand both the memory consistency model and the memory barrier instruction semantics!!

Memory Consistency Models

Alpha	AMD64	IA64	(PA–RISC)	PA–RISC CPUs	POWER	SPARC RMO	(SPARC PSO)	SPARC TSO	x86	(x86 OOStore)	zSeries	
Y	Y	Y	Y		Y	Y			Y	Y		Loads Reordered After Loads?
Y		Y	Y		Y	Y			Y	Y		Loads Reordered After Stores?
Y		Y	Y		Y	Y	Y			Y		Stores Reordered After Stores?
Y	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y	Stores Reordered After Loads?
Y		Y			Y	Y						Atomic Instructions Reordered With Loads?
Y		Y			Y	Y	Y					Atomic Instructions Reordered With Stores?
Y												Dependent Loads Reordered?
Y		Y			Y	Y	Y	Y	Y	Y	Y	Incoherent Instruction Cache/Pipeline?

Code Portability?

Linux provides a carefully chosen set of memory-barrier primitives, as follows:

- smp_mb(): "memory barrier" that orders both loads and stores. This means loads and stores preceding the memory barrier are committed to memory before any loads and stores following the memory barrier.
- smp_rmb(): "read memory barrier" that orders only loads.
- smp_wmb(): "write memory barrier" that orders only stores.

Words of Advice

- "The difficult problem is identifying the ordering constraints that are necessary for correctness."
- "...the programmer must still resort to reasoning with low level reordering optimizations to determine whether sufficient orders are enforced."
- "...deep knowledge of each CPU's memory-consistency model can be helpful when debugging, to say nothing of writing architecture-specific code or synchronization primitives."

Programmer's View

- What does a programmer need to do?
- How do they know when to do it?
- Compilers & Libraries can help, but still need to use primitives in truly concurrent programs
- Assuming the worst and synchronizing everything results in sequential consistency
  - Too slow, but may be a good way to start

Outline

- Concurrent programming on a uniprocessor
- The effect of optimizations on a uniprocessor
- The effect of the same optimizations on a multiprocessor
- Methods for restoring sequential consistency
- Conclusion

Conclusion

- Parallel programming on a multiprocessor that relaxes the sequentially consistent memory model presents new challenges
- Know the memory consistency models for the processors you use
- Use barrier (fence) instructions to allow optimizations while protecting your code
- Simple examples were used, there are others much more subtle.

References

- Shared Memory Consistency Models: A Tutorial By Sarita Adve & Kourosh Gharachorloo
- Memory Ordering in Modern Microprocessors, Part I, Paul E. McKenney, Linux Journel, June, 2005
- Computer Architecture, Hennessy and Patterson, 4th Ed., 2007