ADVANCED PARALLEL COMPUTING
LECTURE 09 - MEMORY CONSISTENCY

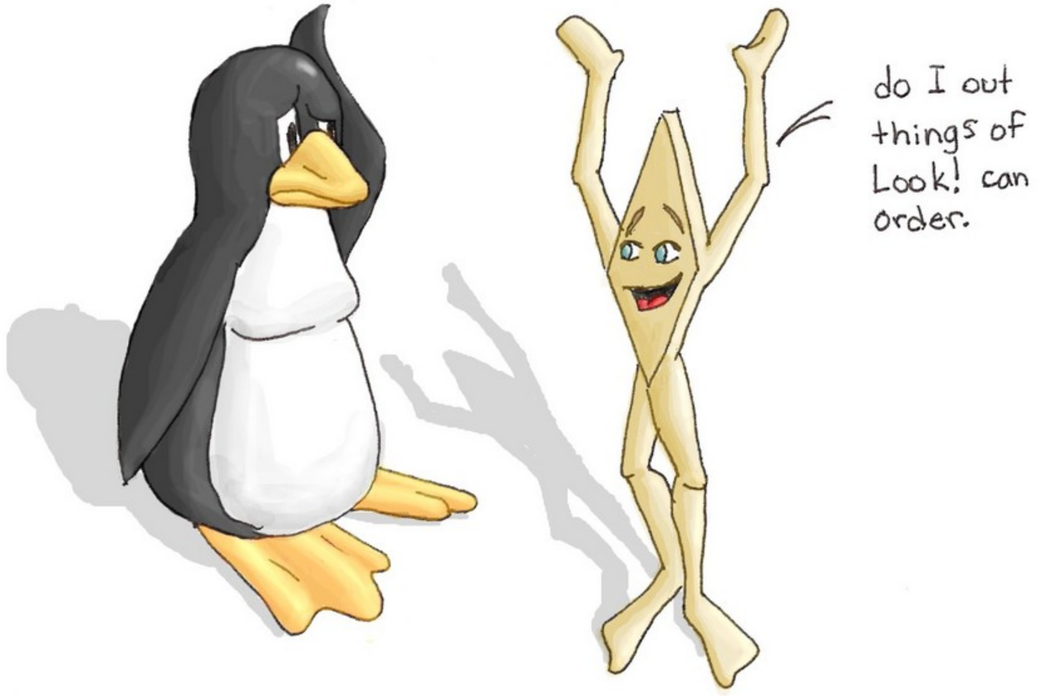
Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

FINAL COHERENCE NOTES

- Main coherence problem: overhead and scalability
  - Bandwidth for probing / latency increase
  - No caches, no coherence problem!
- Intuition says load should return latest value
  - What is latest?
- View from above the clouds (only):
  - Coherence is about reads, consistency is about writes
  - Coherence concerns only one memory location
  - Consistency concerns apparent ordering for all locations

RELAXED MEMORY CONSISTENCY

- Consistency model defines correct behavior
  - Contract between system and programmer (similar to ISA)
  - Restricts ordering of loads/stores
- Software-visible
  - Coherence is not visible to software



http://www.linuxjournal.com

WHY COHERENCE != CONSISTENCY

```
P0                          P1

A = 1;                      while (!flag);
B = 1;                      print A;
flag = 1;                   print B;
```

Intuition says: A=B=1

However, not for all consistency models

Coherence doesn‘t say anything about that

- Why?
- What does it guarantee in this example?

WHY ARE MEMORY ACCESSES RELAXED?

Memory accesses are the slowest operations of a processor (processor/memory gap)

- Huge caches are mandatory to address this disparity
- Caches are partitioned into (almost) independent banks
- E.g.: odd-numbered addresses into cache bank #0, even-numbered into #1
- Idle/busy banks change the visibility of writes

Huge overhead to maintain ordering of memory accesses

=> Performance degradation for strict consistency

=> Only reason for relaxed consistency: performance

STORE ATOMICITY

Again, caches:

```
P0                          P1                          P2

A = 1;

                            while (!A);
                            B = 1;

                                                        while (!B);
                                                        print A;
```

Intuition says: P2 prints A=1

Key issue here: store atomicity

Does a new value reach all nodes at the same time?

Many (past) commercial systems allow P2 to print A=0

ONE PROGRAM - VARIOUS ORDERINGS

What the user wants: memory references specified in high-level language

- C, Java, etc.

Program order: what the compiler makes out of it

- Compilers increasingly reorder memory accesses depending on the dependencies they see (see keyword volatile)

Execution order: how the processor executes these instructions

- Modern CPUs are microcode-based, out-of-order, superscalar, multi-core architectures

Perceived order: how the processor(s) observe memory operations

- Own and others' memory operations are reordered due to caching, interconnect and memory-system optimizations.
- Different processors can even have a different view of memory operations

SOME THINGS YOU CAN COUNT ON

1. One processor always sees its own memory operations in program order

    Thus, reordering issues arise only between processors

2. An operation is reordered with a store only for different addresses
3. Aligned simple loads and stores are atomic
4. Any relaxation comes with a safety net

    I.e., revert to sequential consistency as needed

STRICT CONSISTENCY MODELS

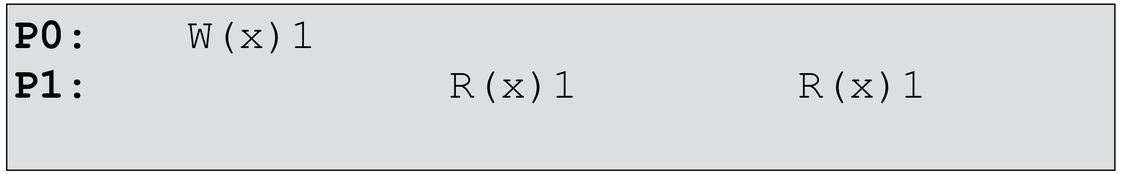
STRICT CONSISTENCY

Strict (or atomic) consistency: "A read returns the most recent value written to a location"

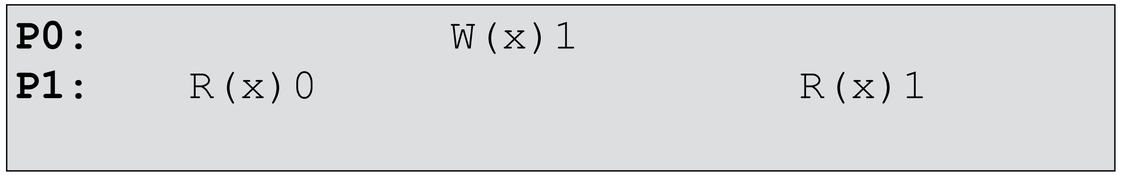
Simplest and strictest consistency model

Relies on a global clock to serialize all memory operations

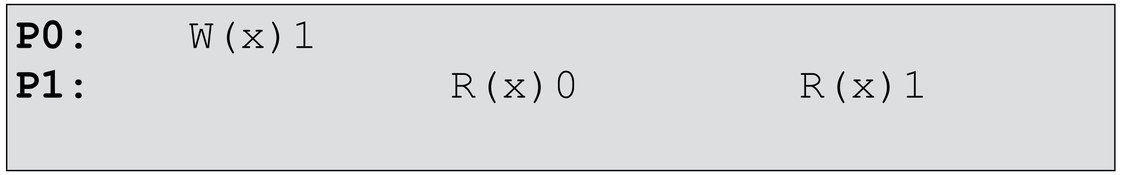
Only used for uniprocessors



Valid



Valid

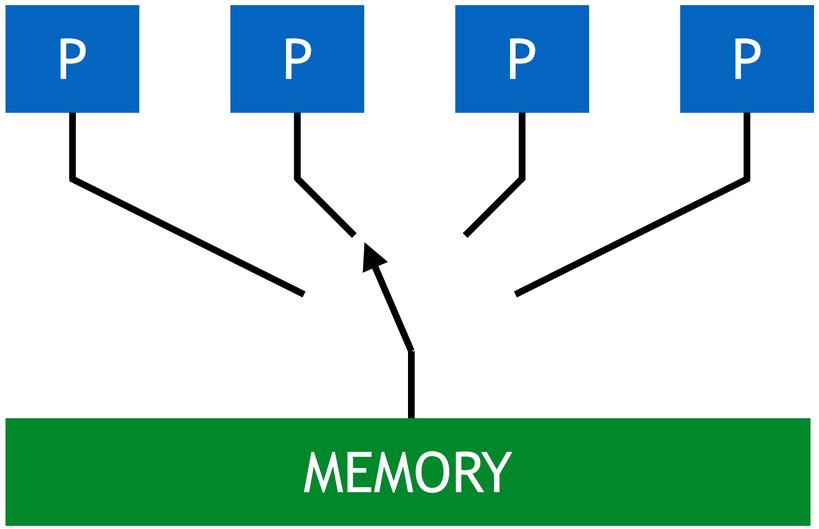


Invalid

SEQUENTIAL CONSISTENCY (SC)

Sufficient condition for SC:

"*A multiprocessor 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 this sequence in the order specified by its program*" – Lamport, 1979



SEQUENTIAL CONSISTENCY

1. Every processor issues memory requests in program order

2. Memory operations happen (start and end) atomically

3. Must wait for a store to complete before issuing next operation

4. After a load, issuing processor waits for load to complete, before issuing next operation

```
P0:     W(x)1
P1:                 R(x)0           R(x)1
```
Valid

```
P0:     W(x)1
P1:                 R(x)1           R(x)2
P2:                 R(x)1           R(x)2
P3:     W(x)2
```
Valid

```
P0:     W(x)1
P1:                 R(x)1           R(x)2
P2:                 R(x)2           R(x)1
P3:     W(x)2
```
Invalid

=> Easily implemented with a shared bus: synchronization point, serializing all accesses

SEQUENTIAL CONSISTENCY

Example code

P0:	**P1:**	**P2:**
$W_0(x,1)$	$W_1(y,2)$	$R_2(y,2)$
	$R_1(a,0)$	$R_2(x,0)$
		$R_2(x,1)$

Perceived order

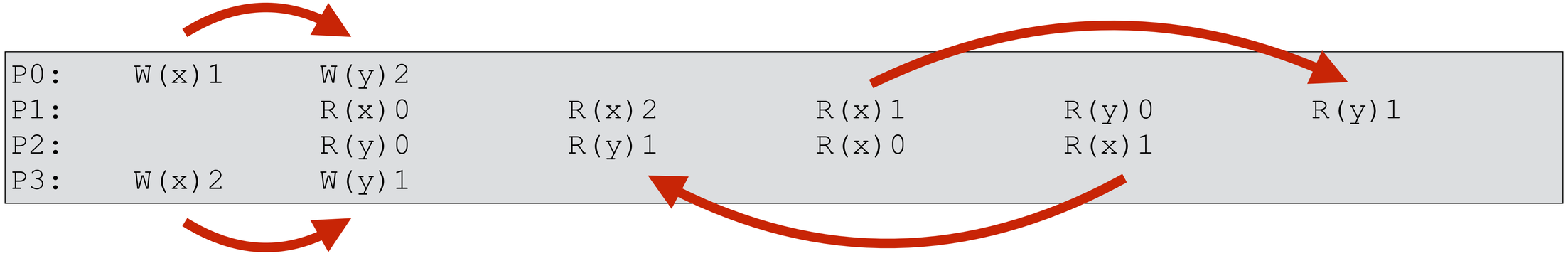
A:	$W_1(y,2)$	<	$R_2(y,2)$	<	$R_2(x,0)$	<	$W_0(x,1)$	<	$R_2(x,1)$	< $R_1(a,0)$
B:	$W_1(y,2)$	<	$R_2(y,2)$	<	$R_2(x,0)$	<	$R_1(a,0)$	<	$W_0(x,1)$	< $R_2(x,1)$
C:	$W_1(y,2)$	<	**$R_2(x,0)$**	<	$W_0(x,1)$	<	$R_2(x,1)$	<	$R_1(a,0)$	< **$R_2(y,2)$**
D:	$W_1(y,2)$	<	**$W_0(x,1)$**	<	$R_2(y,2)$	<	**$R_2(x,0)$**	<	$R_2(x,1)$	< $R_1(a,0)$

Sequentially consistent: A, B

Sequentially inconsistent: C, D

SEQUENTIAL CONSISTENCY: SUFFICIENT CONDITIONS

Sequentially consistent != cache coherent



Sufficient conditions

1. Processors issue memory ops in program order
2. Processors must wait for stores to complete before issuing next memory ops
3. After load, issuing processor waits for load to complete, before issuing next memory operation

Easily implemented with shared (physical) bus

Sufficient, but more than necessary

Example: SGI Origin is SC (MIPS R10000 is dynamically scheduled)

RELAXED CONSISTENCY MODELS

RELAXED CONSISTENCY

Do we really need such a strong model?

Buggy, because of a data race

- Unpredictable behavior without appropriate synchronization
- So, why to maintain strong consistency anyway?
- Compilers neither see multiple processors nor the consistency model

```
P0                  P1
x = x + 1           x = x + 2
```

```
P0                              P1
A = 1;                          B = 1;
if (B != 0) goto retry;         if (A != 0) goto retry;
// enter critical section       // enter critical section
```

PROCESSOR CONSISTENCY

Processor consistency (PC): "*writes done by a single processor are received by all other processors in the order in which they were issued, but writes from different processors may be seen in a different order by different processors*"

- Relax write-to-read order (W2R)
- Allow FIFO store buffers, allowing reads to bypass incomplete writes
- W2W, R2R, R2W still ordered

Formal requirement [Goodman 1989]:

- Before load is preformed, all previous loads must have been performed
- Before store is performed, all previous memory operations must have been performed

Does not require store atomicity!

Examples: x86, VAX, roughly IBM-370, SUN's Total-Store-Order (includes store atomicity)

See also: PRAM (Pipelined Random Access Memory) consistency

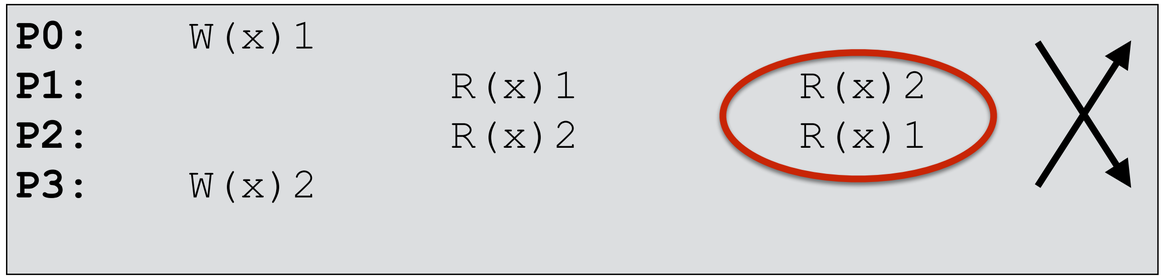
- Also called FIFO consistency
- (Not Parallel Random Access Machine from computability theory)

PROCESSOR CONSISTENCY

Example 1: sequentially inconsistent example

Valid under processor consistency!

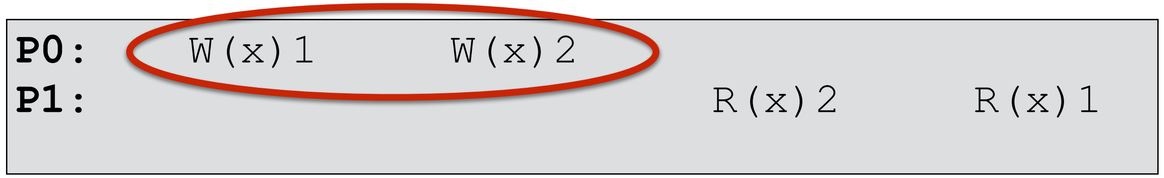
Imagine Ps in a chain



Example 2:

Invalid (W2W is preserved)!

Example 3: previous mutual exclusion code?



Unsafe (W2R not preserved)!

```
P0                                  P1
A = 1;                              B = 1;
if (B != 0) goto retry;             if (A != 0) goto retry;
// enter critical section           // enter critical section
```

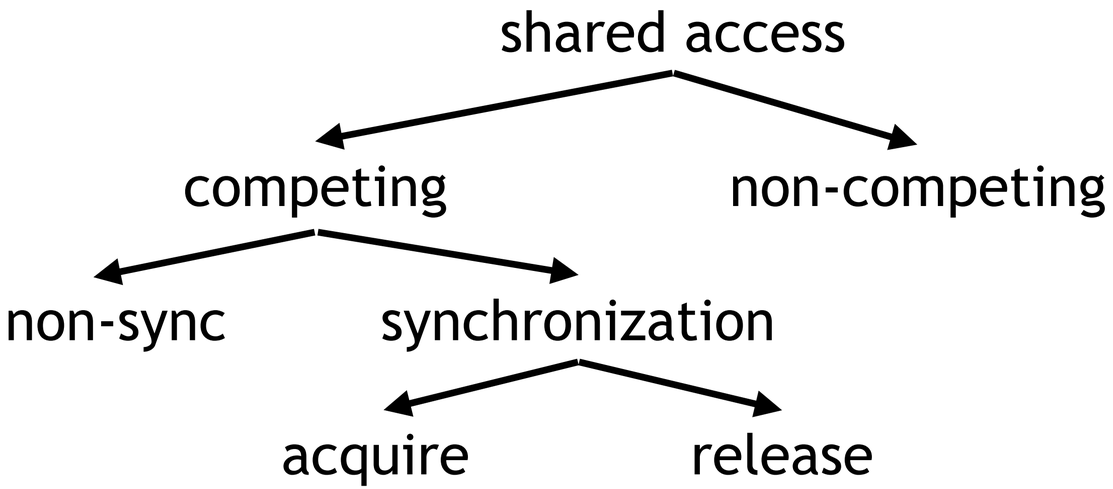
RELAXED CONSISTENCY: SAFETY NET

Each relaxed consistency model comes with a safety net

- Allowing to revert to sequential consistency
- Barriers, syncs, atomics
- Puh!

```
P0                  P1
x = x + 1
S                   S
                    x = x + 2
```

Synchronization accesses vs. ordinary accesses



Competing: multiple accesses from different processors to the same address, at least one write

PARTIAL STORE ORDER

PC: relax W2R order

SUN’s partial store order (PSO): relax W2R and W2W order

- Safety net: RMW, memory barriers

Improves store buffer: unordered, coalescing post-retirement store buffer

- Much more efficient

WEAK CONSISTENCY

Weak consistency (WC): relax everything (W2W, W2R, R2W, R2R), only separate accesses into sync and non-sync

Properties:

1. Synchronization accesses are sequentially consistent
2. No synchronization access is allowed until all previous writes have completed
3. No access is allowed until all previous synchronization accesses have completed

Key advantage: no need to propagate changes until synchronization occurs

But then, update all memory

Dramatic network traffic reduction

RELEASE CONSISTENCY

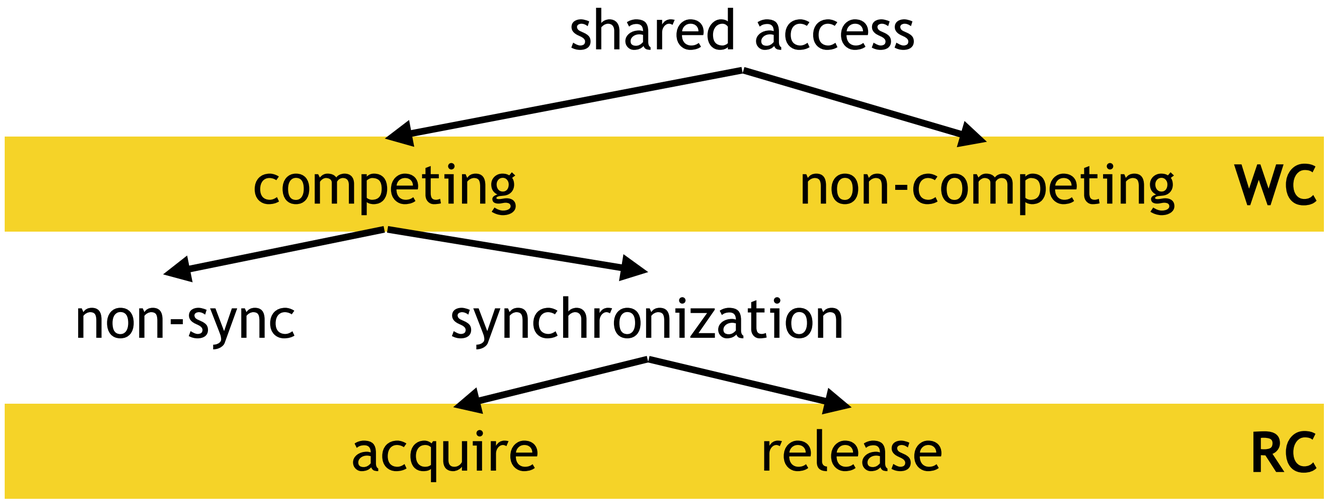
Instead of globally updating memory, release consistency (RC) considers locks on areas of memory, and propagates only the locked memory as needed

Definition:

1. Before a non-sync access is preformed, all previous acquires by the process must have completed
2. Before a release is performed, all previous reads/writes must have completed
3. Acquire/release is sequentially consistent (RCsc)

Eager: actions for releases

Lazy: for subsequent acquires



RELAXING ALL ORDER – APPROACH 1

Approach 1: using explicit “fence” (memory barrier)

SUN’s Relaxed Memory Order (RMO), Alpha, PowerPC, ARM

x86: “Loads may be reordered with older stores to different locations”

MFENCE, SFENCE, LFENCE

```
P0:    W(x)1          W(x)2                                  FENCE
P1:                                R(x)0          R(x)2      FENCE      R(x)2
P2:                                               R(x)1      FENCE      R(x)2
```

RELAXING ALL ORDER – APPROACH 2

Approach 2: annotate loads/stores that do synchronization

Weak consistency (WC): loads/stores can be labeled as “sync”

- No reordering allowed across sync operations

Release consistency (RC): specialize loads and stores as acquires and releases

- Loads and stores form critical sections, allows reordering into critical sections, but not out
- IA64: RCpc (release/acquire obey PC)

Similar to Data-Race-Free-0 (DRF0)

- Foundation of C++/Java memory model

P0:	W(x)1	W(x)2			~~FENCE~~	
P1:			R(x)0	R(x)2	~~FENCE~~	SR(x)2
P2:				R(x)1	~~FENCE~~	SR(x)2

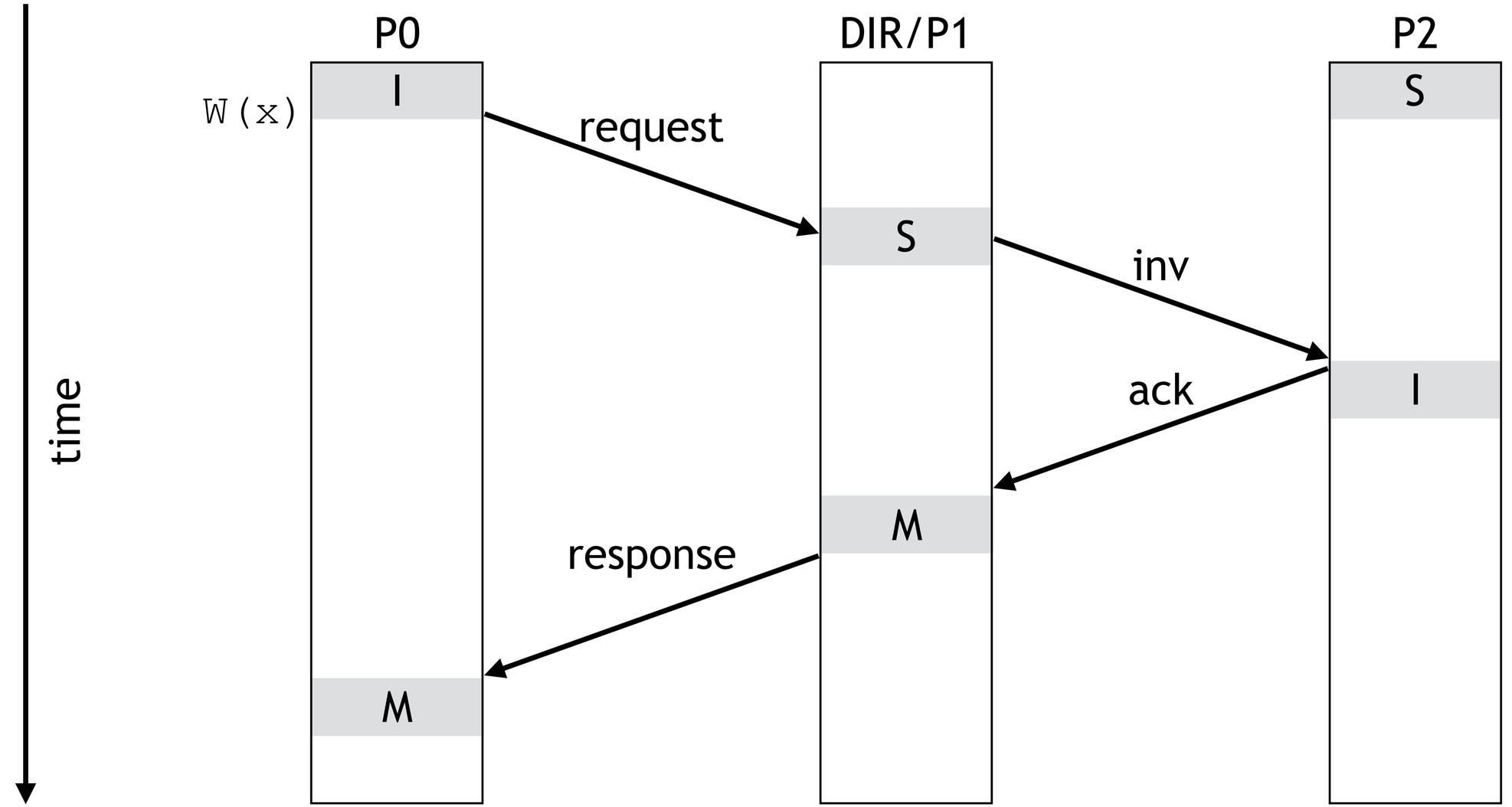
RELAXING ALL ORDER - EXAMPLE

```
P0                          P1
A = 1;                      while (!flag);
B = 1;                      print A;
flag = 1;                   print B;
```

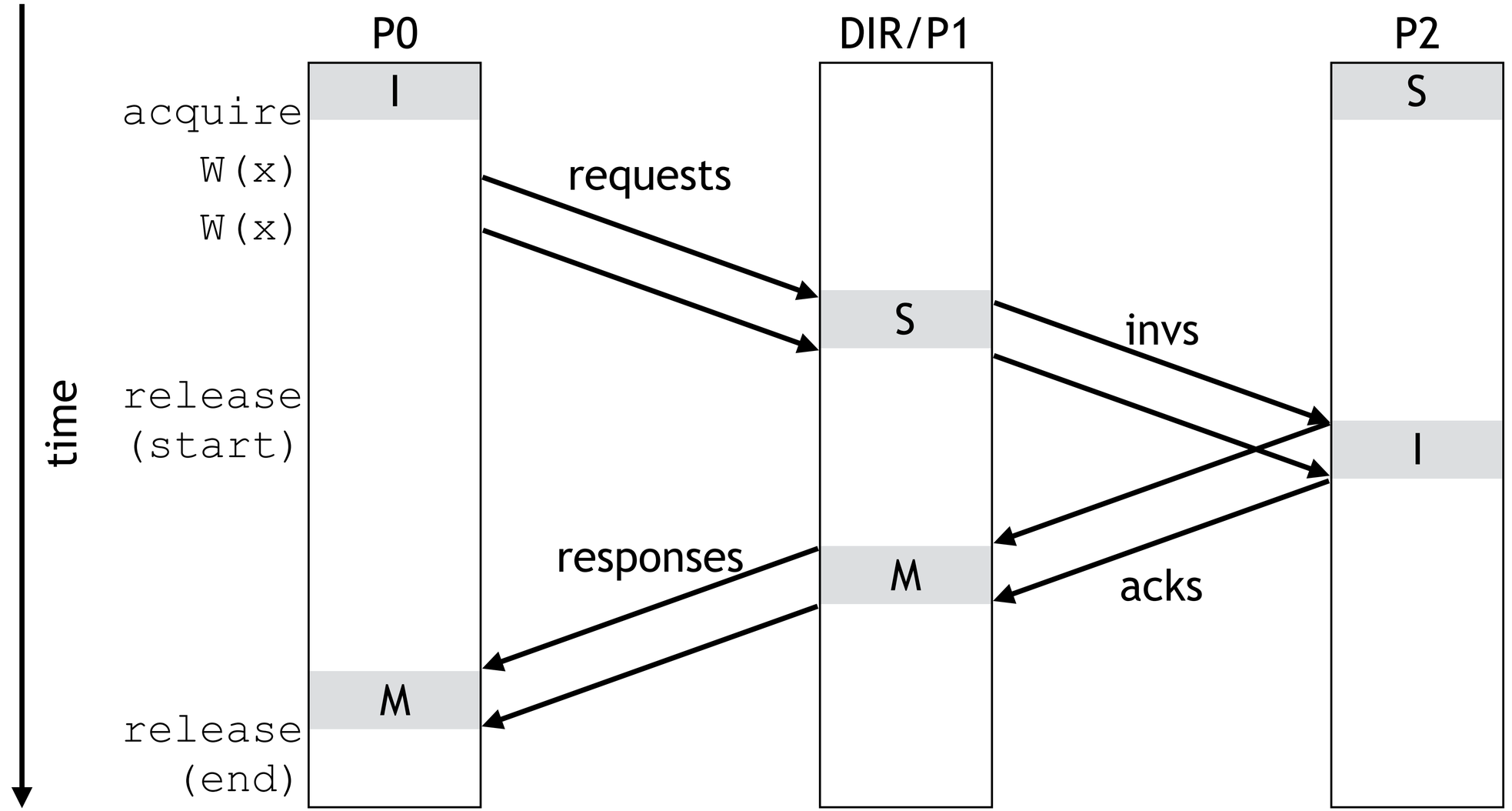
```
P0                          P1
A = 1;                      while (SYNCH !flag);
B = 1;                      print A;
SYNCH flag = 1;             print B;
```

COMMENTS

DIRECTORY EXAMPLE WITH SEQUENTIAL CONSISTENCY



DIRECTORY EXAMPLE WITH RELEASE CONSISTENCY



RELAXING ALL ORDER

Compiler still needs to be careful when optimizing:

```
unsigned int i = x;
// x is shared variable

if (i < 2) { // opt: don't copy x, use by ref

    foo: ... // suppose x changes here…

    switch (i) { // opt: implement as jump table
        case 0: ... break;
        case 1: ... break;
        default: ... break;          // opt: range inference
                                     // tells compiler this
                                     // case is impossible,
                                     // drop the check
        }
    }
```

SOME COMMENTS

Advanced programming models: Partitioned Global Address Space

- UPC: Strict and relaxed operations defined as part of the language

Large-scale DBMS: eventually consistent

- Amazon, Google, Facebook, etc

MEMSCALE's On-Demand Consistency Model

- Custom, but like weak consistency
- Synchronization operations handled in software

	Programmer	Compiler	Hardware	Comment
Strict Consistency	What most/novice programmer expects	Complete disaster!	Global ordering / clock required! No OOO, latency hiding difficult!	Only for uniprocessors
Sequential Consistency	Least astonishing Typically assumed for cache coherence	Disaster! Almost all optimizations are illegal, no reordering!	Disaster! Only one outstanding request! No OOO!	Overkill, most programmers rely on synchronization intrinsics!
Processor Consistency	Sometimes unexpected behavior (membar); however, locks (RMWs) work	May now reorder loads across stores, potential left	Allows for FIFO store buffers & multiple outstanding requests	Typical today x86
Relaxed Consistency (WC,RC,EC)	Very hard (membars where needed)	Sweet, sweet freedom!	Allows for unordered, coalescing SBs & OOO CPUs	

	Programmer	Compiler	Hardware	Comment
Strict Consistency	What most/novice programmer expects	Complete disaster!	Global ordering / clock required! No OOO, latency hiding difficult!	Only for uniprocessors
Sequential Consistency	Least astonishing Typically assumed for cache coherence	Disaster! Almost all optimizations are illegal, no reordering!	Disaster! Only one outstanding request! No OOO!	Overkill, most programmers rely on synchronization intrinsics!
Processor Consistency	Sometimes unexpected behavior (membar); however, locks (RMWs) work	May now reorder loads across stores, potential left	Allows for FIFO store buffers & multiple outstanding requests	Typical today x86
Relaxed Consistency (WC,RC,EC)	Very hard (membars where needed)	Sweet, sweet freedom!	Allows for unordered, coalescing SBs & OOO CPUs