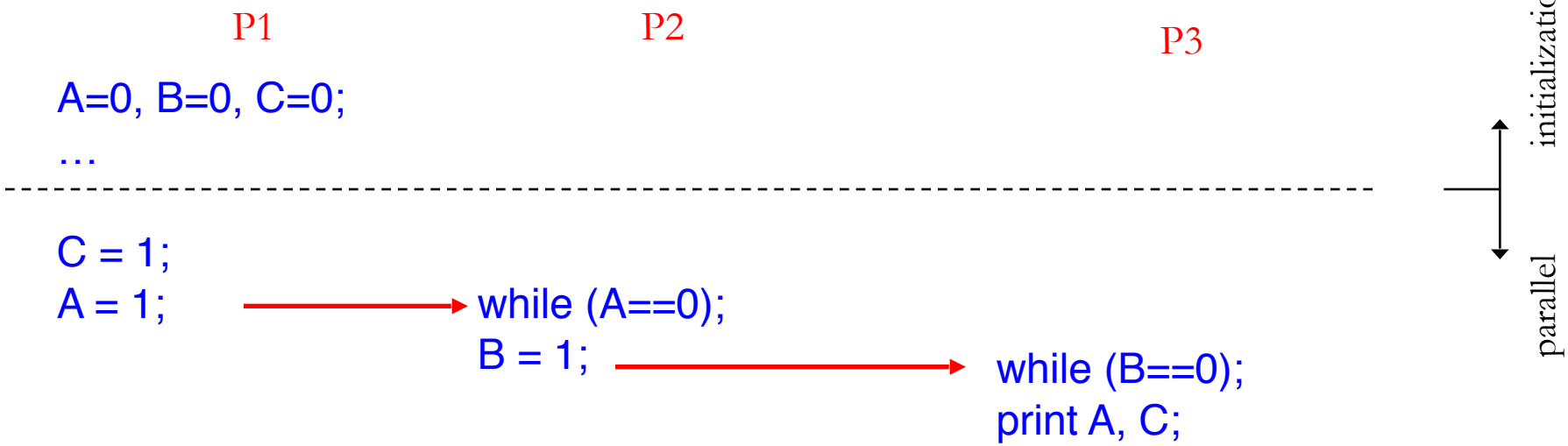
Lect. 7: Memory Consistency

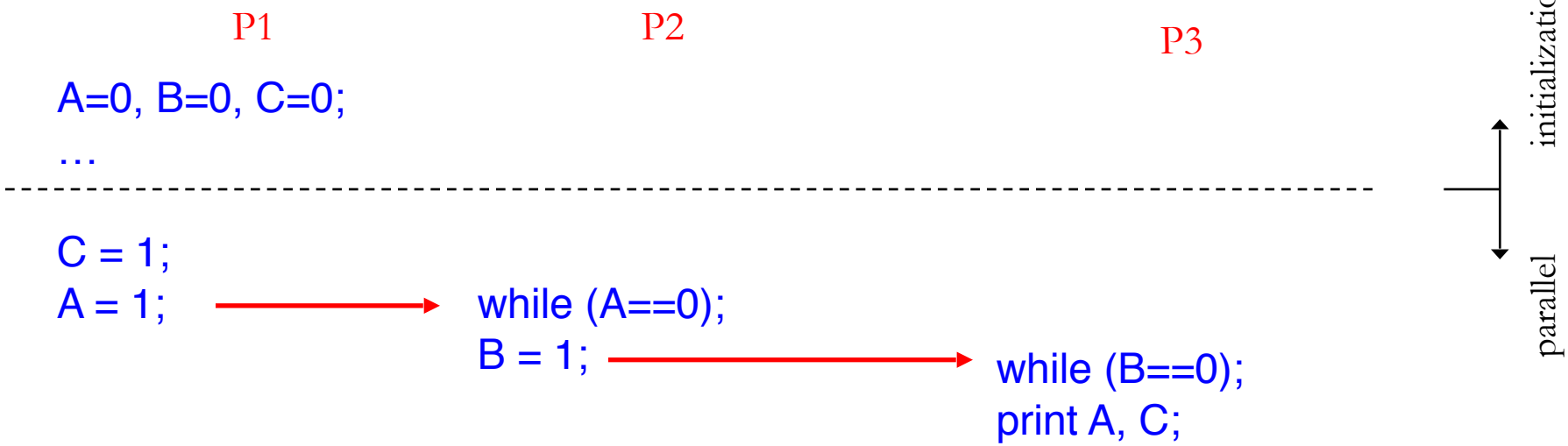
- Consider the following code:



  - What are the possible outcomes?

Lect. 7: Memory Consistency

- Consider the following code:



  - What are the possible outcomes?

A==1, C==1? → Yes. This is what one would expect.

A==0, C==1? → Yes. If st to B overtakes the st to A on the interconnect toward P3.

A==0, C==0?

A==1, C==0? → Yes. If the st to A overtakes the st to C from the same processor.

Memory Consistency

- Cache coherence:
  - Guarantees eventual write propagation
  - Guarantees a single order of all writes to same location

No guarantees on when writes propagate.

For the same memory location.
No guarantee on write-atomcity

- Memory consistency:
  - Specifies the ordering of loads and stores to different memory locations
  - Defined in so called Memory Consistency Models
  - This is really a "contract" between the hardware, the compiler, and the programmer
    - i.e., hardware and compiler will not violate the ordering specified
    - i.e., the programmer will not assume a stricter order than that of the model
  - Hardware/Compiler provide "safety net" mechanisms so the user can enforce a stricter order than that provided by the model

Write-Serialization vs Write-Atomicity

// Initially all values are 0.

P1	P2	P3	P4
X= 1			X=2
	=X(1)	=X(2)	
	=X(2)	=X(1)	

Violation of Write-serialization

P1	P2	P3	P4
X= 1			Y=1
	=X(1)	=Y(1)	
	=Y(0)	=X(0)	

Violation of Write-atomicity

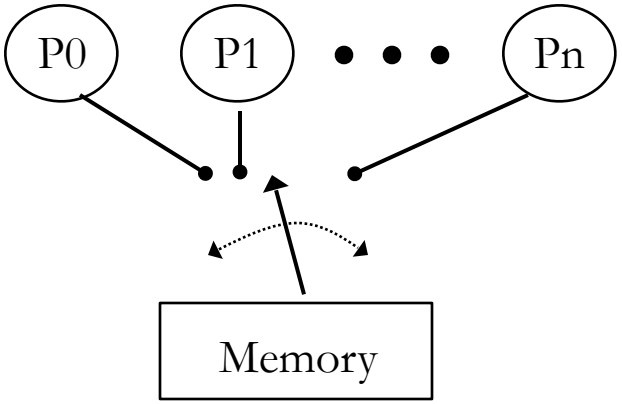
Sequential Consistency (SC)

A multiprocessor system 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 78]

Sequential Consistency (SC)

- Key ideas:
  - The behaviour should be the same as in a time-shared multiprocessor
  - Two aspects:
    - Program order: memory ordering has to follow the individual order in each thread (R→R, R→W, W→W, W→R)
    - Write-atomicity: there can be any interleaving of such sequential segments - but a single total order of all memory operations

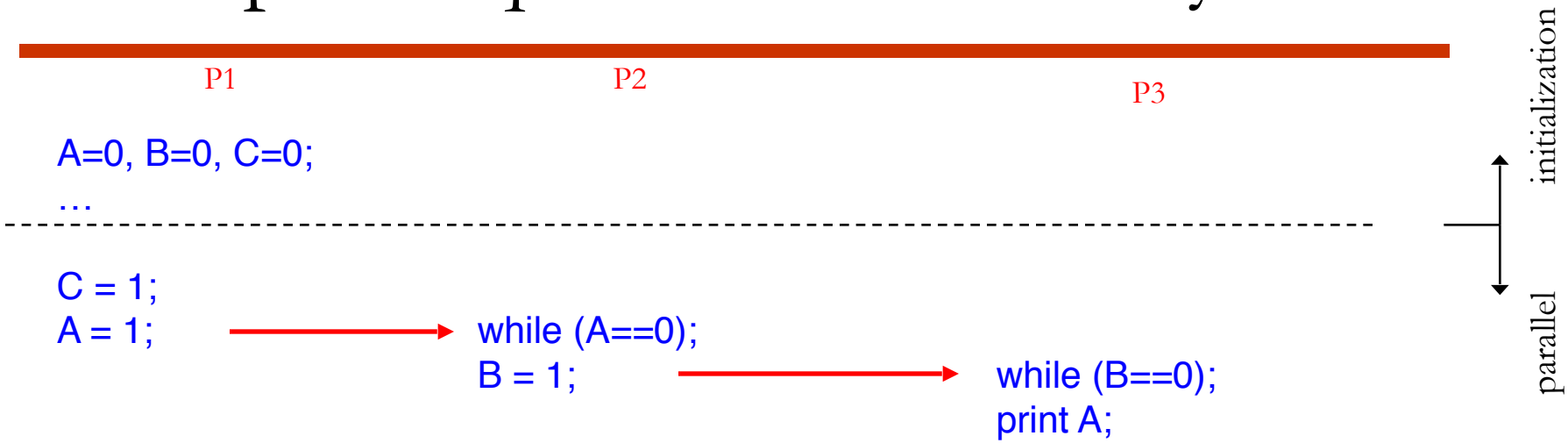


  - Notice that in practice many orderings are still valid

Terminology

- Issue: memory operation leaves the processor and becomes visible to the memory subsystem
- Performed: memory operation appears to have taken place
  - Performed w.r.t. processor X: as far as processor X can tell
    - E.g., a store S by processor Y to variable A is performed w.r.t. processor X if a subsequent load by X to A returns the value of S (or the value of a store later than S, but never a value older than that of S)
    - E.g., a load L is performed w.r.t. processor X if all subsequent stores by any processor cannot affect the value returned by L to X
  - Globally performed or complete: performed w.r.t. to all processors
    - E.g., a store S by processor Y to variable A is globally performed if any subsequent load by any processor to A returns the value of S
- X consistent execution: result of <u>any</u> execution that matches <u>one of</u> the possible total orders (interleavings) as defined by model X
  - "Result of an execution": Values returned by the reads

Example: Sequential Consistency



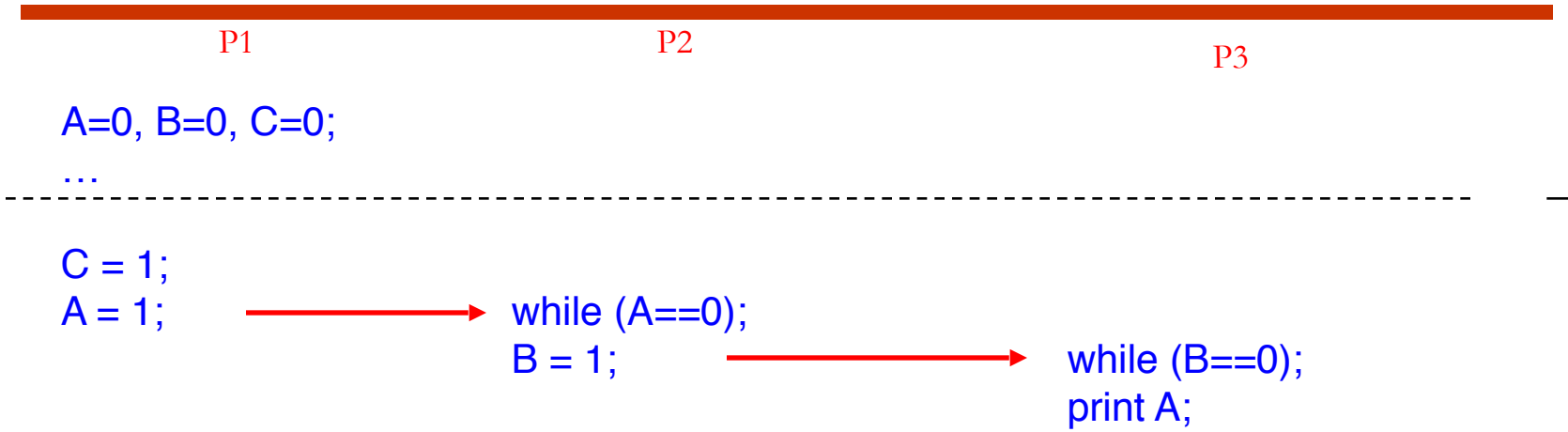
- Some valid SC orderings:

```
P1: st C # C=1
P1: st A # A=1
P2: ld A # while
P2: st B # B=1
P3: ld B # while
P3: ld A # print
```

```
P1: st C # C=1
P2: ld A # while
…
P1: st A # A=1
P2: ld A # while
P2: st B # B=1
P3: ld B # while
P3: ld A # print
```

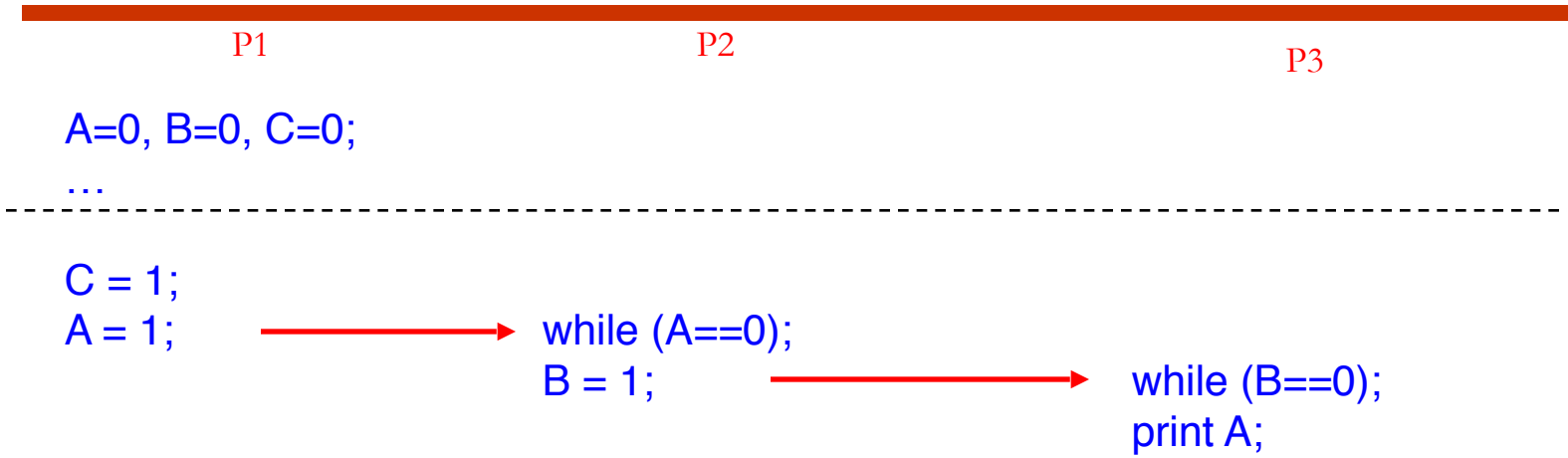
```
P1: st C # C=1
P2: ld A # while
…
P1: st A # A=1
P2: ld A # while
P3: ld B # while
…
P2: st B # B=1
P3: ld B # while
P3: ld A # print
```

Is this ordering SC?



P1: st A # A=1
P1: st C # C=1
P2: ld A # while
P2: st B # B=1
P3: ld B # while
P3: ld A # print

Is this ordering SC? Yes!



The values returned by the reads of the execution on the left matches that of the SC execution on the right!

P1: st A # A=1
P1: st C # C=1
P2: ld A # while
P2: st B # B=1
P3: ld B # while
P3: ld A # print

P1: st C # C=1
P1: st A # A=1
P2: ld A # while
P2: st B # B=1
P3: ld B # while
P3: ld A # print

Sequential Consistency (SC)

- Sufficient conditions
  1. Threads issue memory operations in program order
  2. Before issuing next memory operation threads wait until last issued memory operation completes (i.e., performs w.r.t. all other processors)
  3. A read is allowed to complete only if the matching write (i.e., the one whose value is returned to the read) also completes
- Notes:
  - Condition 2 guarantees program ordering
  - Condition 3 guarantees write atomicity
  - These conditions are easily violated in real hardware and compilers (e.g., write buffers in hdw. and ld-st scheduling in compiler)
  - In practice necessary conditions may be more relaxed

How SC can be violated

Initially flag1 = flag2 = 0

P1

```
flag1 = 1

if(flag2 == 0)
{
   /* critical*/
}
```

P2

```
flag2 = 1

if(flag1 == 0)
{
   /* critical*/
}
```

If write-buffering is used each processor can buffer the writes (to flag1 and flag2 resp.) and go ahead with the reads. This will cause both reads (flag1 and flag2) to return 0, causing both P1 and P2 to enter critical section.

Efficient SC: In-window Speculation

- Using speculation (for optimizing W→R, R→R)
  - The later read can be issued earlier
  - But memory operations complete in program order
  - If an invalidate is received, speculation squashed and replayed starting from the (later) load

- Write-prefetching (for optimizing W→W)
  - Obtain read-exclusive out-of-order (or in parallel)
  - Complete writes in program order.

Post-retirement speculation

- Inwindow speculation in practice good for R→R and W→W but not W→R
- Write takes long time to perform: instruction window (ROB) can get full.
- Post-retirement speculation: speculation beyond instruction window

SC via Conflict Ordering

- Efficient SC without aggressive speculation
- Memory operations may be reordered as long as reordering is invisible to other processors

Conflict Ordering

Initially flag1 = flag2 = 0

P1

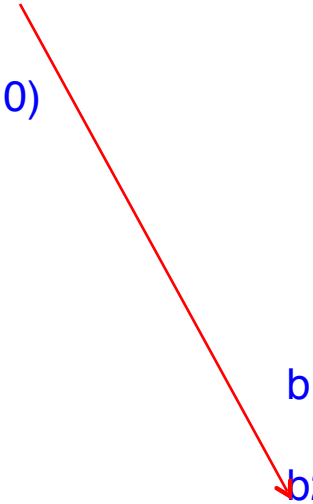
```
a1:flag1 = 1

a2:if(flag2 == 0)
{
   /* critical*/
}
```

P2

```
b1:flag2 = 1

b2:if(flag1 == 0)
{
   /* critical*/
}
```



In this example, does not matter if (a1,a2) and (b1,b2) perform out-of-order; as long as b2 sees a1, both processors can't enter critical section at the same time

Conflict Ordering

Initially flag1 = flag2 = 0

P1

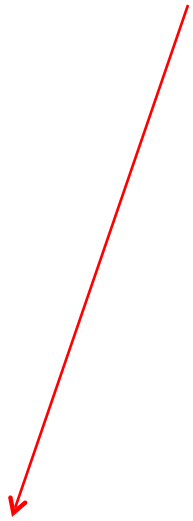
```
a1:flag1 = 1

a2:if(flag2 == 0)
{
   /* critical*/
}
```

P2

```
b1:flag2 = 1

b2:if(flag1 == 0)
{
   /* critical*/
}
```



In this example, does not matter if (a1,a2) and (b1,b2) perform out-of-order; as long as a2 sees b1, both processors can't enter critical section at the same time

Relaxed Memory Consistency Models

- At a high level they relax ordering constraints between pairs of reads, writes, and read-write (e.g., reads are allowed to bypass writes, writes are allowed to bypass each other)
- Some of the models also don't guarantee write atomicity
- Some models make synchronization explicit and different from normal loads and stores
- Many models have been proposed and implemented
  - Total Store Ordering (TSO) (e.g., Sparc, intel): relaxes W—>R ordering
  - Partial Store Ordering (PSO) (e.g., Sparc): relaxes W—>R and W —>W
  - Relaxed Memory Ordering (RMO) (e.g., Sparc): relaxes all 4 memory orders
  - Release Consistency (RC) (e.g Itanium): relaxes all 4 memory orders but provides *release store* and *acquire load* (ARM v8 has a similar model).
  - IBM Power: relaxes all 4 memory orderings; also relaxes write atomicity; provides 2 types of barriers.
    - lwsync: ensures R—>R , R—>W, and W—>W; write-atomicity still not ensured
    - sync: ensures all 4 memory ordering; write atomicity also ensured.

Relaxed Memory Consistency Models

- Note that control flow and data flow dependences within a thread must still be honoured regardless of the consistency model
  - E.g.,

```
A=0, B=0, C=0;
…
-----------------------------------------------------------------------
C = 1;
A = 1;                    while (A==0);
                          B = 1;
```

ld to A cannot perform before ld to B

```
while (B==0);
print A;
```

  - E.g.,

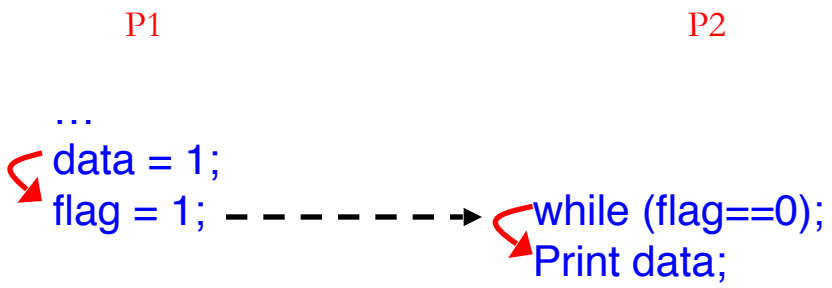
st to B cannot perform before ld to A

```
A = 1;
…                         Second st to A cannot perform earlier than st to A
A = 2;
…                         ld to A cannot perform before earlier st to A
B = A;
```

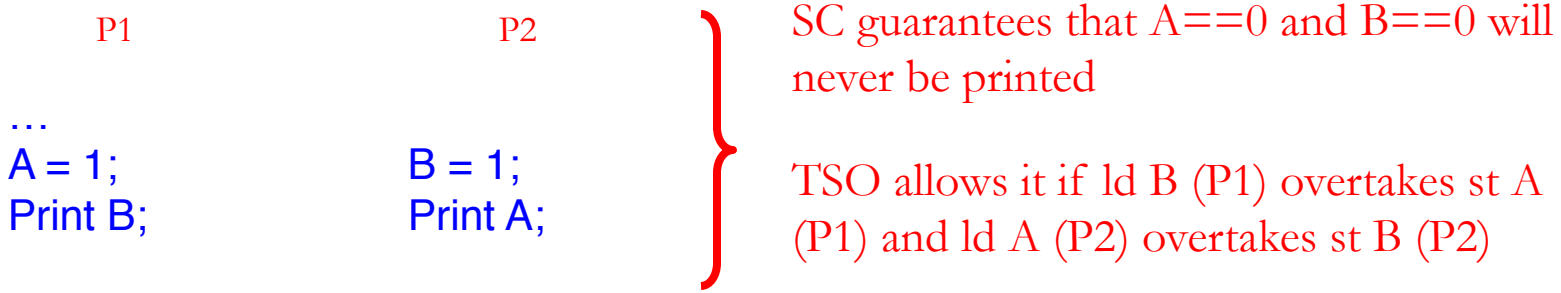
Example: Total Store Ordering (TSO)

- Reads are allowed to bypass writes (can hide write latency)

- Still makes prior example work as expected,

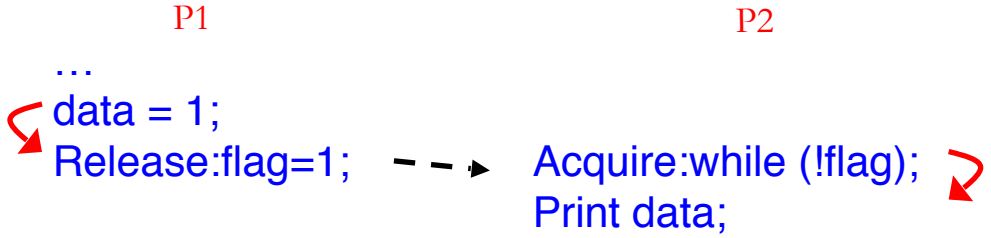


but breaks some intuitive assumptions,



Example: Release Consistency (RC)

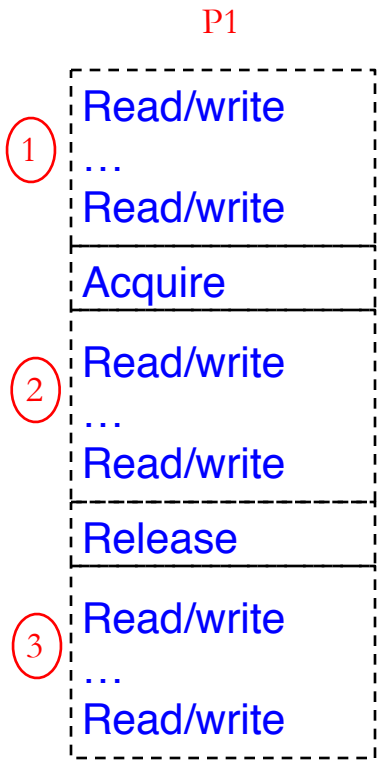
- Reads and writes are allowed to bypass both reads and writes (i.e., any order that satisfies control flow and data flow is allowed)
- Assumes explicit synchronization operations: acquire and release So, for correct operation, our example must become:



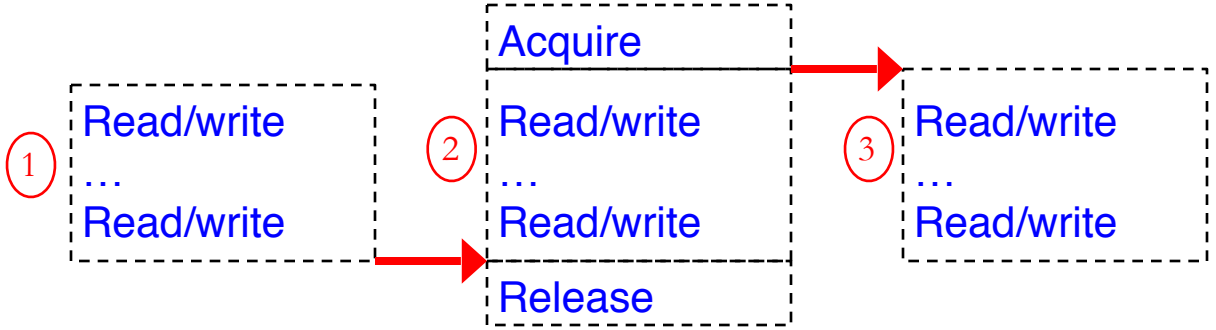
- Constraints
  - All previous writes (and previous reads) must complete before a release can complete
  - No subsequent reads (and subsequent writes) can complete before a previous acquire completes
  - All synchronization operations must be sequentially consistent (i.e., follow the rules for SC, where an acquire is equivalent to a read and a release is equivalent to a write)

Example: Release Consistency (RC)

- Example: original program order



- Allowable overlaps



  - Reads and writes from block 1 can appear after the acquire
  - Reads and writes from block 3 can appear before the release
  - Between acquire and release any order is valid in block 2 (and also 1 and 3)
- Note that despite the many reorderings, this still matches our intuition of critical sections

Implementing RC

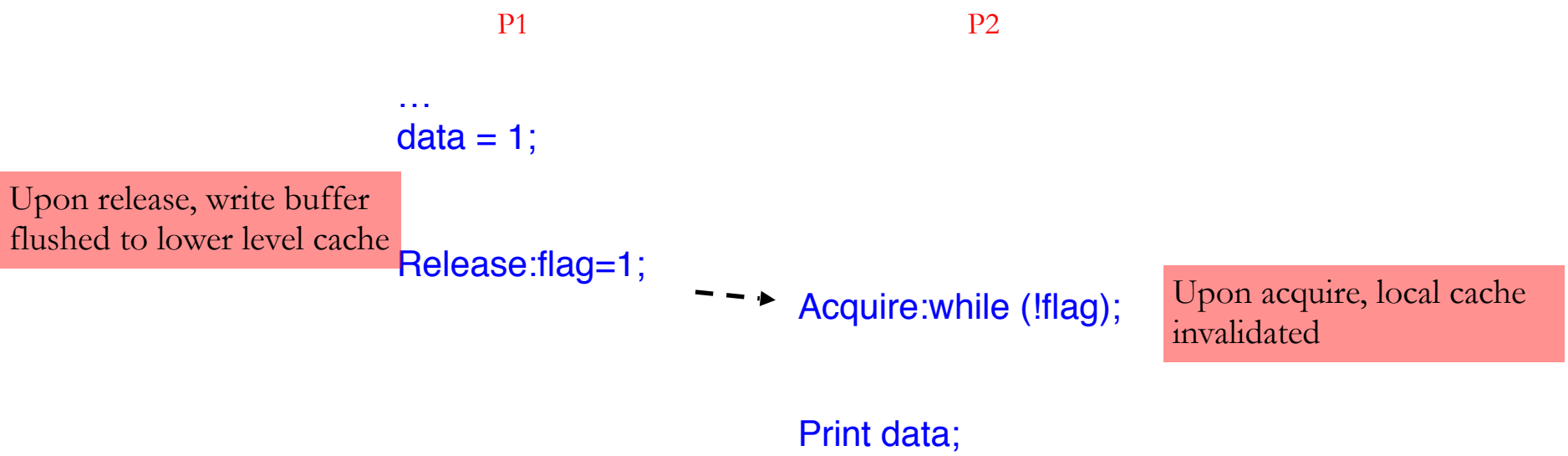
- Eager RC
  - Writes retire into write buffer (and could complete out of order)
  - Reads maybe be issued out-of-order (and may complete out of order, although in practice they processors commit in order.)
  - Note: Invalidates need not snoop LSQ
  - A release drains the write-buffer
  - All loads following an acquire are issued after the acquire.

- Question: *Must* writes *eagerly* invalidate shared copies?

Implementing RC

- Lazy RC
  - Writes retire into write buffer
    - (and complete out of order)
    - But only written to the local cache – no eager invalidates.
  - A release writes all dirty blocks (including the release write) in the local cache into coherent lower-level cache (memory)
  - Reads maybe be issued out-of-order (but completed in inorder)
  - Upon an acquire the local cache is self-invalidated (and the acquire load also forced to miss the local cache)

Implementing RC: Lazy RC

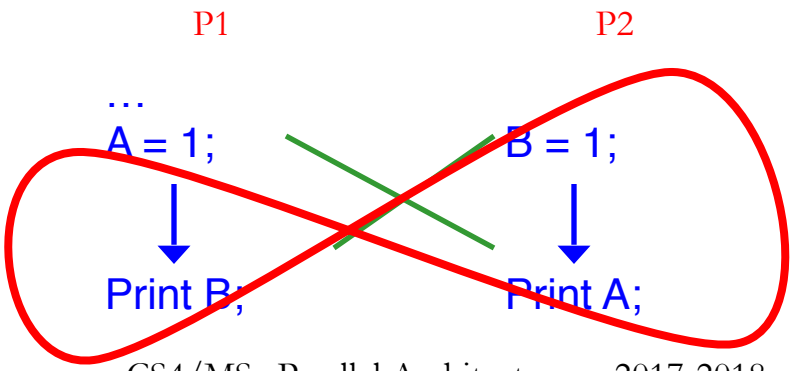


Lazy RC

- Does not require a sharer vector
- Would still benefit from local/shared tracking
  - Local data need not be written to the lower-level upon a release
  - Local data need not be self-invalidated upon an acquire
  - Could use the directory to track this or even the TLB. (How)?
- Aside: Is the cache still coherent?

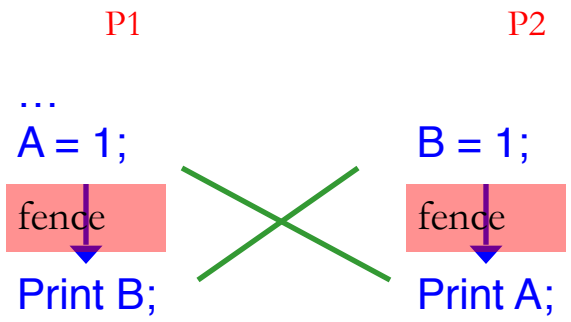
SC Behaviour on Relaxed Models

- Delay-set Analysis
  - Technique to identify pairs of memory accesses that need to be ordered for SC
  - Mark all memory references in both threads and create arcs between them
    - Directed intra-thread arcs that follow program order (the blue ones below)
    - Undirected inter-thread arcs that follow cross-thread data dependences (the green ones below, recall that the print implicitly contains a read)
  - Cycles following the arcs indicate the problematic memory references



SC Behaviour on Relaxed Models

- How can ordering be enforced on memory accesses?
- Memory Fences:
  - New instructions in the IS
  - Specify that previously issued memory operations must complete before processor is allowed to proceed past the fence
    - Read fence : all previous reads must complete before the next read (or write) can be issued
    - Write fence: all previous writes must complete before the next write (or read) can be issued
    - Full fence : all previous reads and writes must complete before the next memory operation can be issued; also guarantees write atomicity.



Final Notes

- Many processors/systems support more than one consistency model, usually set at boot time
  - Sparc supports TSO, PSO, RMO

- It is possible to decouple consistency model presented to programmer from that of the hardware/compiler
  - E.g., hardware may implement a relaxed model but compiler guarantees SC via memory fences
  - Language Level memory model (C, C++, Java etc. have memory models)
    - Variants of DRF: Data-race-free memory models.
    - SC guaranteed but only for well synchronised programs
    - Well synchronised: All potentially conflicting (racing) operations are labelled by the programmer.

References and Further Reading

- Original definition of sequential consistency:

  "How to Make a Multiprocessor Computer that Correctly Execute Multiprocess Programs", L. Lamport, IEEE Trans. on. Computers, September 1979.

- In-window speculation

  K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models., ISCA '91, pages 355--364, 1991.

- Post-retirement speculation

  Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: performance-transparent memory ordering in conventional multiprocessors. SIGARCH Comput. Archit. News37, 3 (June 2009)

- Conflict ordering

  Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. 2012. Efficient sequential consistency via conflict ordering. ASPLOS 2012.

- A very good tutorial on memory consistency models:

  "Shared Memory Consistency Models: A Tutorial", S. Adve and K. Gharachorloo, IEEE Computer, December 1996.

References and Further Reading

- Lazy RC

Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. 1992. Lazy release consistency for software distributed shared memory. SIGARCH Comput. Archit. News 20, 2 (April 1992), 13-21.

Ashby, T.J.; Díaz, P.; Cintra, M.; , "Software-Based Cache Coherence with Hardware-Assisted Selective Self-Invalidations Using Bloom Filters,"Computers, IEEE Transactions on , vol.60, no.4, pp.472-483, April 2011

- Delay set analysis:

"Efficient and Correct Execution of Parallel Programs that Share Memory", D. Shasha and M. Snir, ACM Trans. on. Programming Languages and Operating Systems, February 1988.

- Compiler support for SC on non-SC hardware:

"Hiding Relaxed Memory Consistency with Compilers", J. Lee and D. Padua, Intl. Conf. on Parallel Architectures and Compilation Techniques, October 2000