Parallel Computer Architecture
Spring 2019

Memory Consistency

Nikos Bellas

Electrical and Computer Engineering Department
University of Thessaly

Coherence vs Consistency

- *Cache coherence* ensures that all processors see a consistent view of the memory
- A variable X stored in some cache(s) or in memory will have the same value for every processor
- *Consistency* is concerned with "**when**" must a processor see a value that was updated by another processor
- A *memory consistency model* is a formal specification of how the memory system will appear to the programmer
  - A model that programmers can use to reason about possible results and correctness of the program

Memory Consistency Example

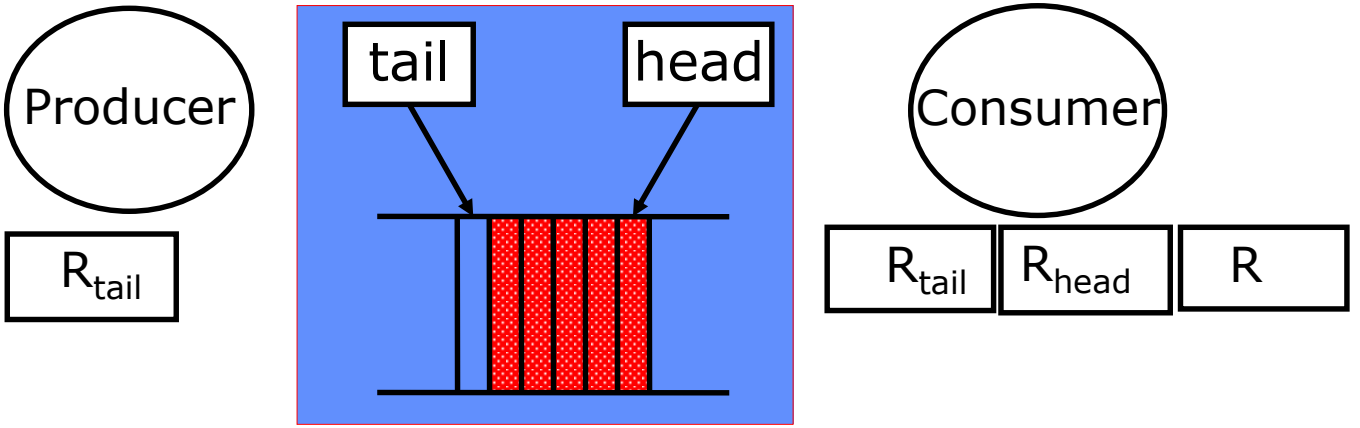
Initially A=B=0 in both P1 and P2 caches

```
P1:                                P2:

    ....                           ....
  A = 1;                         B = 1;
  if (B==0)  X=1;                if (A==0)  X=3;
```

- If writes takes always immediate effects, the two processors can never have BOTH if-statements evaluate to TRUE
- BUT, if write invalidates are delayed, P1 does not see “B=1” and/or P2 does not see “A=1”. Both if-statements can evaluate to TRUE
- The problem is not solved by cache coherence, which only requires that the new value of A (and B) become available to all processors after some time.

A Producer-Consumer Example



Producer posting Item x:

Load R_{tail}, (tail)
Store (R_{tail}), x
R_{tail}=R_{tail}+1
Store (tail), R_{tail}

Consumer:

Load R_{head}, (head)
empty: Load R_{tail}, (tail)
if R_{head}==R_{tail} goto empty
Load R, (R_{head})
R_{head}=R_{head}+1
Store (head), R_{head}
process(R)

The program is written assuming instructions are executed in order. Assume infinite-size buffer

Problem is that Consumer has to access R_{tail} which is controlled by Producer

A Producer-Consumer Example

continued

Producer posting Item x:

```
    Load R_tail, (tail)
1   Store (R_tail), x
    R_tail=R_tail+1
2   Store (tail), R_tail
```

Consumer:

```
    Load R_head, (head)
spin: Load R_tail, (tail)      3
    if R_head==R_tail goto spin
    Load R, (R_head)           4
    R_head=R_head+1
    Store (head), R_head
    process(R)
```

Can the tail pointer get updated before the item x is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences are:

2, 3, 4, 1 // Instruction 4 does not load X

4, 1, 2, 3

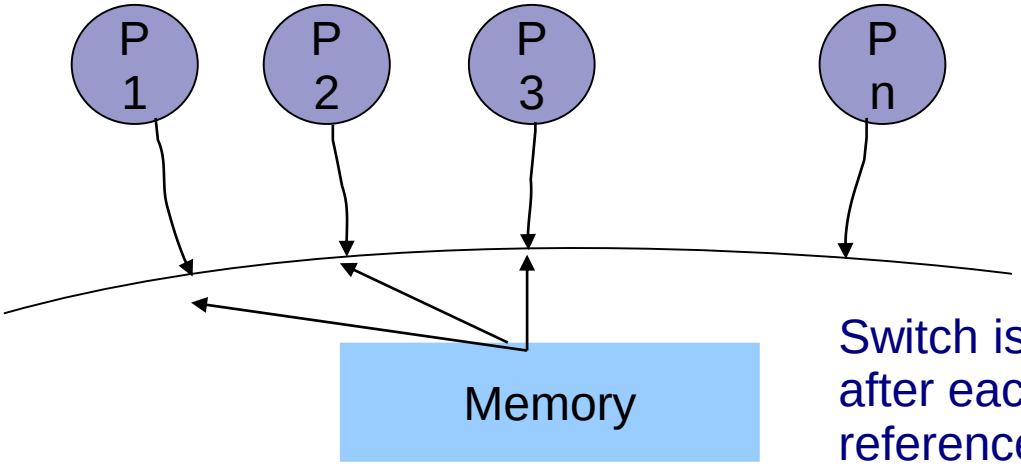
Sequential Consistency

- A memory consistency model specification is ***required*** for every interface between the programmer and the machine
- It affects hardware design, compiler design, and high level programming language design
- *Sequential consistency* is the most widely assumed model for shared memory multiprocessors
  - Easy to understand
  - Restricting in terms of performance

Sequential Consistency

- Definition: A multiprocessor system is *sequentially consistent* if:
  - the operations of each individual processor appear [to all processors] in sequence in the order specified of the program, and
  - The result of any execution is the same as if the operations of all processors were executed in **some** sequential order

Programmer's view of sequential consistency



Switch is randomly set after each memory reference

Sequential Consistency Requirements
Hardware

- According to the definition of SC, hardware implementations need to satisfy two conditions:

1) A processor needs to ensure that its previous memory operation is complete, before proceeding with the next one (1:*program order requirement*)

2a) Writes to the same location should be serialized, i.e. made visible in the same order to all processors, AND

2b) The value of a write to a location should only be returned by a subsequent read to the same location when all invalidates of the write have been acknowledged. In other words, when the effect of the write has been made visible to all processors.

(2a,2b: *write atomicity*)

Reasoning with Sequential Consistency

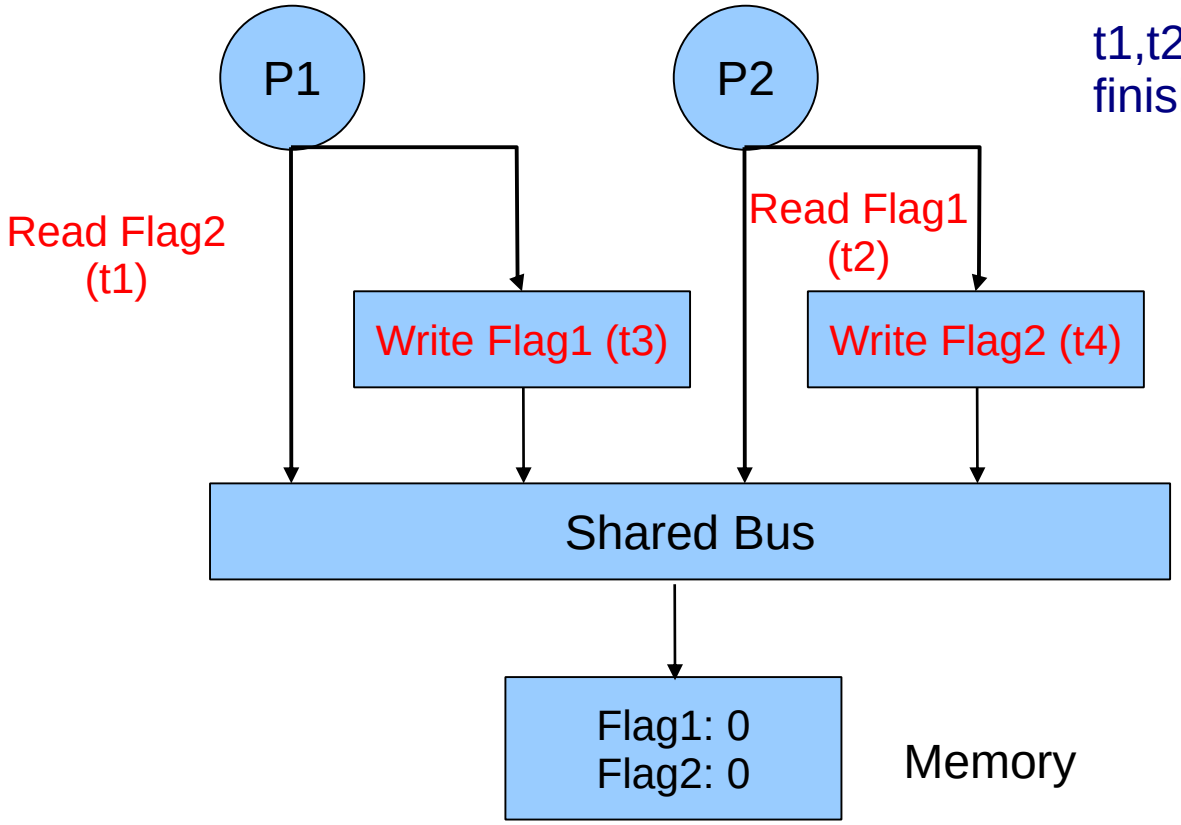
initial: A, flag, x, y == 0

p1	p2
(a) A = 1;	(c) x = flag;
(b) flag = 1;	(d) y = A

- **program order: (a) → (b) and (c) → (d)**
- **claim: (x,y) == (1,0) cannot occur**
  - **x == 1 => (b) → (c)**
  - **y == 0 => (d) → (a)**
  - **thus, (a) → (b) → (c) → (d) → (a)**
  - **Wrong!**

Sequential Inconsistency Examples

Importance of sequential ordering (property 1)



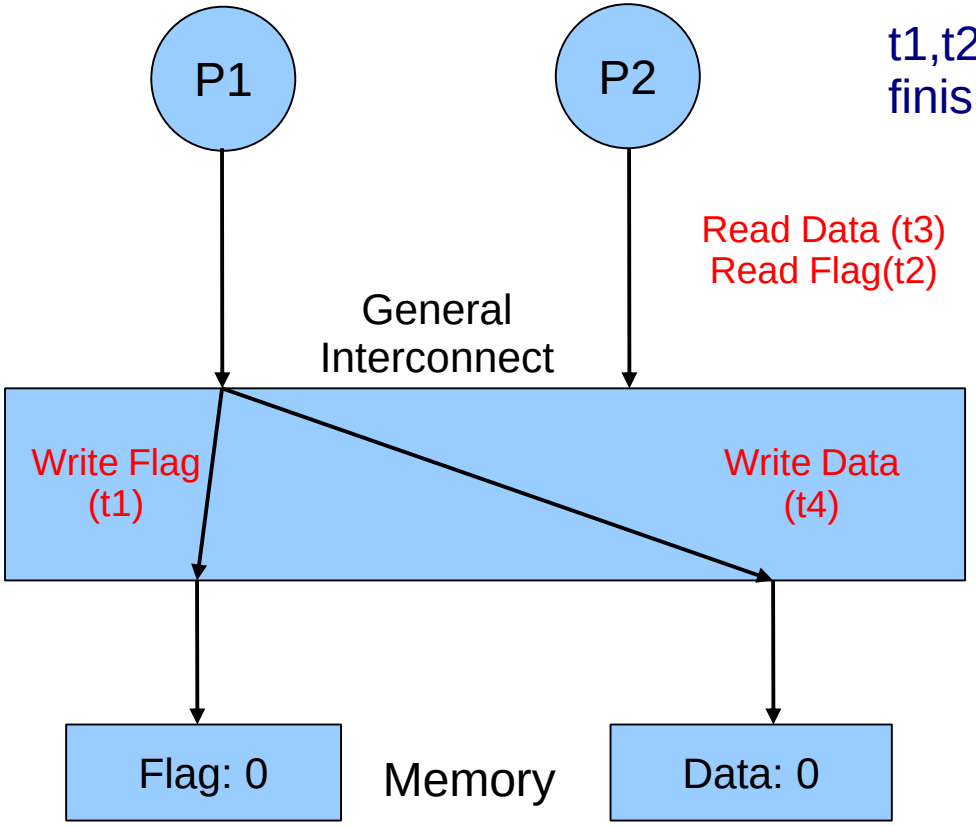
t1,t2,t3,t4 sequence of finishing events

P1	P2
Flag1=1	Flag2=1
if (Flag2==0)	if (Flag1==0)
<critical code>	<critical code>

Effects of delayed writes (Write Buffers) :
P1 and P2 enter the critical section simultaneously
This is also known as relaxation of the W --> R program order (for the same processor)
Total Store Ordering (TSO)

Sequential Inconsistency Examples

Importance of sequential ordering (property 1)



t1,t2,t3,t4 sequence of finishing events

P1	P2
Data=20	while (Flag==0) {;}
Flag=1	X = Data

Effects of reordered writes
X becomes 0, instead of 20
This is also known as relaxation of the W --> W program order

Partial Store Ordering Model (PSO) relaxes W->W and W->R

Sequential Inconsistency Examples

Importance of write atomicity (property 2)

Initially A=B=C=0

P1	**P2**	**P3**	**P4**
A=1	A=2	While (B!=1) {;}	While (B!=1) {;}
B=1	C=1	While (C!=1) {;}	While (C!=1) {;}
		Register1 = A	Register2=A

Assume that **program order requirement** is satisfied in each processor.
Let us assume execution sequence in P1/P2 : *A=1, B=1, A=2, C=1*
The write invalidation of statement *A=2* to all copies of A is delayed in the interconnection network. P3 reads A before it receives write invalidation of A, and therefore reads the older value of *A=1*.
P3 sees A=1 --> Register1 = 1
P4 sees A=2 --> Register2 = 2
Inconsistent! (Register1 == Register2 in a consistent system)

Memory Consistency Model

- The MC model specifies constraints on the order in which memory operations (to the same or different locations) can appear to execute with respect to one another,
- enabling programmers to reason about the behavior and correctness of their programs.
  - fewer possible reorderings => more intuitive
  - more possible reorderings => allows for more performance optimization
    - ‘fast but wrong’ ?

Do we really need SC?

- Programmer needs a model to reason with
  - not a different model for each machine

=> Define “correct” as same results as sequential consistency

- Many programs execute correctly even without “strong” ordering

initial: A, flag, x, y == 0

p1	**p2**
A=1;	
B =3.1415	
barrier;	barrier;
	... = A;
	... = B;

- explicit synch operations order key accesses

Does SC eliminate synchronization?

- No, still need critical sections, barriers, events
  - insert element into a doubly-linked list
  - generation of independent portions of an array
- only ensures interleaving semantics of individual memory operations
- It says nothing about logical correctness of the program

Properly Synchronized Programs

- All synchronization operations explicitly identified
- All data accesses ordered though synchronizations
  - no data races!

=> Compiler generated programs from structured high-level parallel languages

=> Structured programming in explicit thread code