Memory Consistency

Jingke Li

Portland State University

Memory Consistency Problems

On a multiprocessor system, there are two memory-related consistency problems:

1. ▸ *The "Cache Coherence" Problem* —
   Consistency of multiple copies of the same data.

2. ▸ *The "Memory Consistency" Problem* —
   Consistency of views of memory operations among processors.

Both of these problems are aimed at clarifying the "correctness" of shared memory operations.

Two variables can be linked for some reason

Much broader case than "1"

Can't just read add write to memory anymore. Need to keep track of any update in the system.

Nowdays cache coherence is supported by hardware

"2" Could be solved ~~only~~ by hardware only.

Not a problem on single CPU systems.

The Cache Coherence Problem

In modern computer architectures, memory hierarchy (main memory plus multi-level of caches) is used to overcome the memory access latency problem.

A single data item may have multiple copies reside in different levels of a memory hierarchy, these copies may not always be identical.

- On an uniprocessor, disagreement may happen between the cache and the memory. But that is not a problem, because the cache copy is always accessed first.
- On a share-memory multiprocessor system, multiple caches are connected to the same memory. Disagreement can happen not only between a cache and the memory, but also among different caches themselves.

Solutions

Cache Coherence Problem Example

Two processors, P_0 and P_1, access the same shared data x.

With *Write-Through* Caches: Write to main memory immediately

1. Processors P_0 and P_1 each reads x from main memory, bringing a copy to their cache.
2. P_0 changes x's value, the new value will be copied to main memory.
3. Processor P_1 reads x's value again — it gets the old value from its cache! → Not correct.

Cache Coherence Problem Example (cont.)

Delay update of main memory → instead update local cache
Only write to memory when multiple updates accumulated.

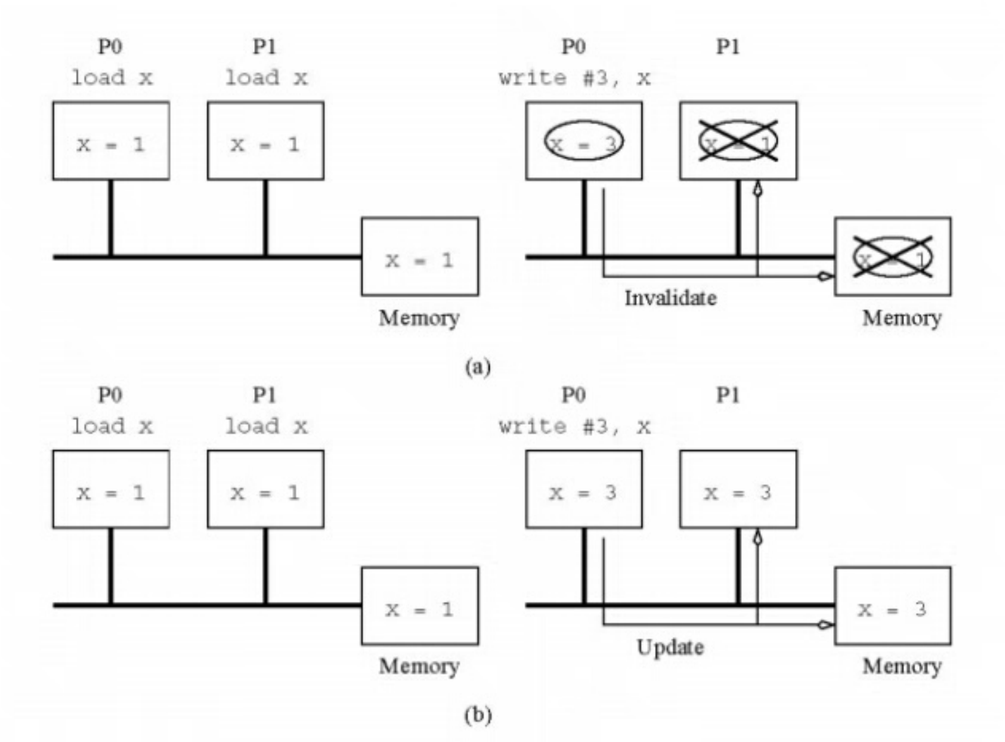
With *Write-Back* Caches:

1. Processors P_0 and P_1 each reads x from main memory, bringing a copy to their cache.
2. P_0 changes x's value, the new value stays in P_0's cache.
3. P_1 reads x, gets the stale value from its cache. (Any other processor reading x, will also get the stale value from memory.) → Unacceptable

In addition, if multiple processors with distinct values of x to write back, the final value of x in main memory will be determined by the order of the cache lines arrival at the destination, which may not have anything to do with the order of the writes to x.

What to do when ver B updated?.

Solutions: Invalid or Update

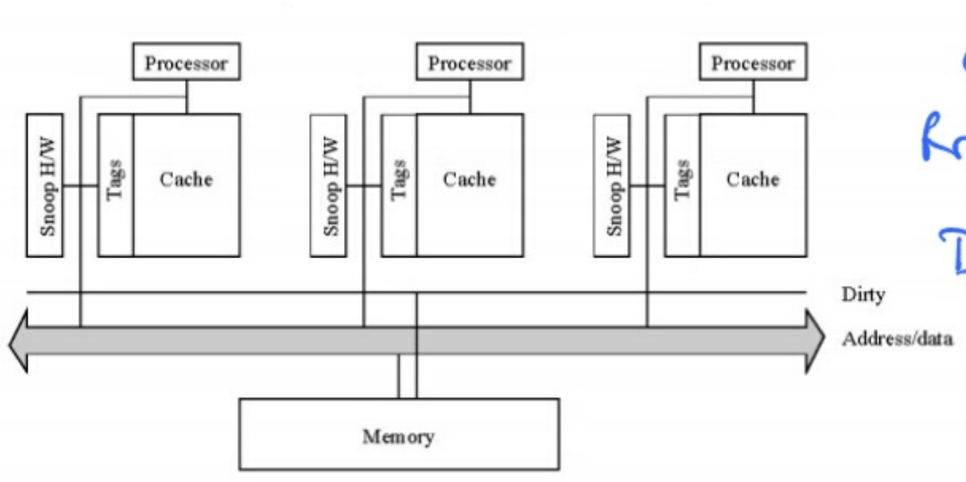


- *Invalidate* — whenever a data item is written, all other copies in the memory system are invalidated. → Keep local, invalidate anywhere
- *Update* — whenever a data item is written, all other copies are updated. → update everywhere.

Requires knowledge of location of all other copies.

Cache-Coherence through Bus Snoopy

Assume multiprocessors with private caches are placed on a shared bus.



Every CPU gets updates from the Bus (like hub)

Info to send to the bus received by all CPUs.

- Each processor's cache controller continuously snoops on the bus watching for relevant transaction (i.e. that involves cache lines of which it has a copy in its cache)
- Once such a transaction is caught, it takes either of the following actions: invalidate the copy in the cache or update the copy in the cache

Cache Coherence is SOLVED at the time at cost!

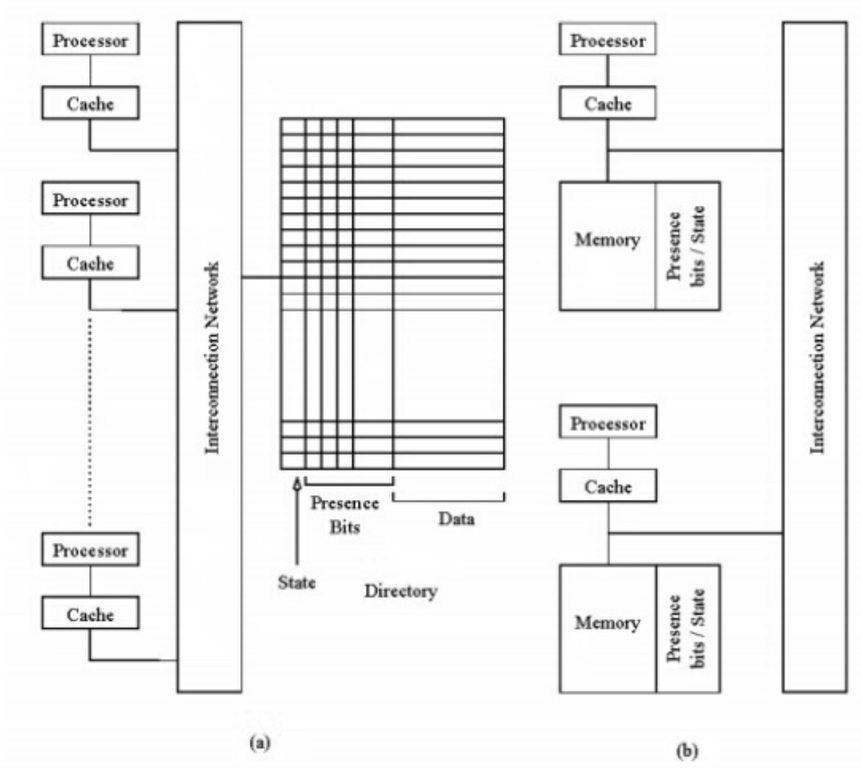
Hardware issue.

Due to increased scale old topology no longer work (bandwidth limit)

Directory-Based Cache Coherency

Use a cache directory to record the locations and states of all cached lines

- A directory entry contains locations of all remote copies of the same line and status info
- Main advantage is scalability; works also on machines with physically distributed memory



No longer instantaneous/free (some delay is introduced)

Broadcast is simple, fast but expensive

In MIS approach we send messages directly only to the nodes that want it.

→ Directory usually resides at a "master" CPU memory.
↳ new bottleneck, still more effective than broadcast.

Broader Scope almost anything else falls here.

1. Step -> Run Compiler -> produce target code (executable)
   Source program. // Compiler preserves semantics of the source.

2. Step -> Execute target code // Modern hardware doesn't guarantee to execute code as given!

The Memory Consistency Problem

Example: During each step semantics might be altered

Source program:

```
c = a + c;
d = a + b;
```

Target code:

```
movl a, %edx
movl c, %eax
addl %edx, %eax
movl %eax, c
movl a, %edx
movl b, %eax
addl %edx, %eax
movl %eax, d
```

Execution:

```
Read(a)
Read(c)
..
Write(c)
Read(b)
..
Write(d)
```

1. Read a
2. Read c
3 Update C

Three memory operation orders:

- *Program Order* — Operation order defined by the source program.
- *Code Order* — Operation order defined by the executable code compiled from the source program.
- *Commit Order* — Memory operation order committed to the memory. The actual order

REMEMBER THOSE NAMES

Jingke Li (Portland State University) CS 415/515 Memory Consistency 9 / 28

Code order ≠ Program order (due to Compiles optimizations) it could read either a or c first.

Observations About Memory Operation Orders

Due to compiler and hardware optimizations, the following may happen:

code order ≠ program order

and/or

commit order ≠ code order // Similar to code order.

Taken as granted (could force the order though)

On a uniprocessor:

- No matter what these orders are, the program's semantics is preserved by the compiler and hardware.

  Conclusion: There is no memory consistency issue.

// Order is not an issue

On a multiprocessor: -> Order can cause issues (how to prove it?)

- For a multi-threaded program, there are multiple program orders.
- There are multiple views of the memory due to local caches.

  Question: How do we address memory consistency? (when have multiple CPUs?)

Jingke Li (Portland State University) CS 415/515 Memory Consistency 10 / 28

Motivating Examples

P_1 and P_2 are two separate programs.

Example 1: Assume initially `A=0` and `B=0`.

P_1:

```
A = 1;
B = 2;
```

P_2:

```
print B;
print A;
```

Question: If P_2 prints `B`'s value as 2, what value of `A` would you expect it to print?

Answ: Not clear

Motivating Examples (cont.)

Example 2: Assume initially `A=0` and `flag=0`.

Programmers intention: We want A to be 1. So we wait for the flag.

P_1:

```
A = 1;
flag = 1;
```

P_2:

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

Question: What value of `A` would you expect P_2 to print? Answ: Not Clear, but user intention is clear.

Example 3: Assume initially `A=0` and `flag=0`.

basic - sync routine

P_1:

```
A = 1;
barrier(b);
```

P_2:

```
barrier(b);
print A;
```

Question: Again, what value of `A` would you expect P_2 to print? Answ: Should force the compiler. look more like sync, so we expect 1.

Example 3 is trying to forge program order, compiler should recognize word barrier and force the order.

Defining model that helps to solve memory consistency problem.
Still should be supported by the compiler.

Sequential Consistency (SC) Force reads and writes to a sequence

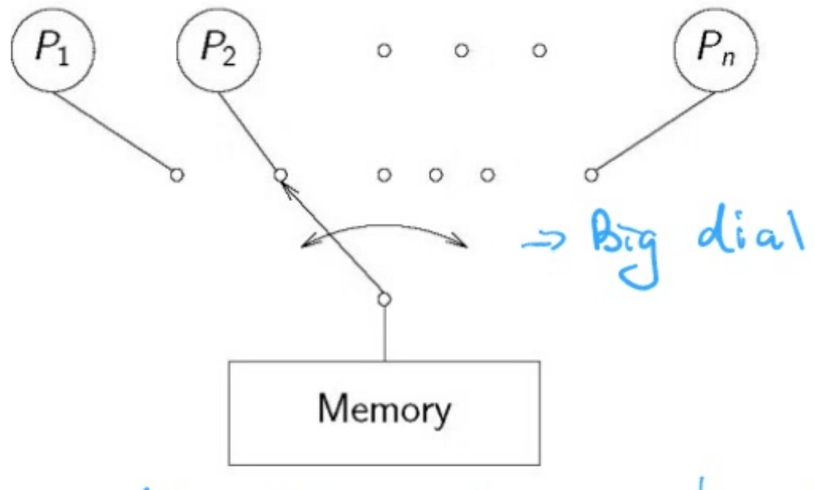
A multiprocessor is *sequentially consistent* if the reads and writes by all processors appear to execute serially in a single global memory order that *conforms to the program orders* in individual processors.

In other words,

Other way to look at it

- all memory operations form a single global commit order; and
- the commit order conforms to the program order of each processor.

Sequential consistency can be viewed as if the instructions from all processors are executed on a single processor in an interleaved way:



→ Big dial

While executing keep changing the dial.

SC Applied to Example 1

Assume initially A=0 and flag=0.

P_1:

```
A = 1;
B = 2;
```

P_2:

```
print B;
print A;
```

Valid Scenarios: total 3×6 → 29

W_1(A=1)	W_1(A=1)	W_1(A=1)	R_2(B=0)	R_2(B=0)	R_2(B=0)
W_1(B=2)	R_2(B=0)	R_2(B=0)	W_1(A=1)	W_1(A=1)	R_2(A=0)
R_2(B=2)	W_1(B=2)	R_2(A=1)	W_1(B=2)	R_2(A=0)	W_1(A=1)
R_2(A=1)	R_2(A=1)	W_1(B=2)	R_2(A=1)	W_1(B=2)	W_1(B=2)

Invalid Scenarios: Order of each P must be followed, but not both

Inconsistent

W_1(B=2)	W_1(A=1)	W_1(B=2)
W_1(A=1)	W_1(B=2)	W_1(A=1)
R_2(B=2)	R_2(A=1)	R_2(A=1)
R_2(A=1)	R_2(B=2)	R_2(B=2)

...

SC Applied to Example 2

Assume initially A=0 and flag=0.

P_1:

```
A = 1;
flag = 1;
```

P_2:

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

Valid Scenarios:

W_1(A=1)
W_1(flag=1)
R_2(flag=1)
R_2(A=1)

W_1(A=1)
R_2(flag=0)
...
R_2(flag=0)
W_1(flag=1)
R_2(flag=1)
R_2(A=1)

R_2(flag=0)
...
R_2(flag=0)
W_1(A=1)
W_1(flag=1)
R_2(flag=1)
R_2(A=1)

R_2(flag=0)
...
R_2(flag=0)
W_1(A=1)
R_2(flag=0)
...
R_2(flag=0)
W_1(flag=1)
R_2(flag=1)
R_2(A=1)

If we do have sequential consistency we should be happy.

A Subtle Point

Still some room for inconsistency -> doesn't force inter-process order (important for shared memory).

Sequential consistency insists on having a single global commit order.
However, this commit order *does not* need to be identical to every processor's local view of the global memory.

Example: Assume initially A=0 and B=0.

P_1:

```
B = 1;
C = A;
```

P_2:

```
C = B;
```

A possible scenario: 1. Don't forget about local copy.

Global Order:	*P_1's View:*	*P_2's View:*
W_1(B=1)	W_1(B=1)	W_1(B=1)
R_1(A=0)	R_1(A=0)	R_2(B=1)
R_2(B=1)	R_2(B=1)	R_1(A=0)
W_1(C=0)	W_1(C=0)	W_1(C=0)
W_2(C=1)	W_2(C=1)	W_2(C=1)

-> This is allowed by SC! Will cause problems.

Two orders might happen at the same time. Order depends on the value of local copy, or on the time when it updates.

Cost of the model?
↓
all readers must match.

Sequential Consistency Requirements

Implementing SC requires that the system preserve two intuitive constraints:

1. ▶ *Program order requirement* —

   Memory operations of a processor must appear to become visible, to itself and others, in program order, *i.e.*

   Commit order = Code order = Program order → this can be hard.

   for all processors.

2. ▶ *Write atomicity requirement* — ① is not enough to guarantee SC.

   It should appear that a write operation is completed with respect to all processors before the next read or write operation in the total order is issued (regardless of which processor issues it), *i.e.*

   While not all local views of memory operation orders are required to be identical, all local views regarding write operations' ordering with respect to other operations are. (Every time you do write → it must visible to all participating processors.

② Explanation:

Relay:

The Importance of Write Atomicity

We can rely on the wait since SC is used!

P_1:	P_2:	P_3:
`A = 1;`	`while (A==0);` `B = 1;`	`while (B==0);` `print A;`

Intuition expects P_3 to print 1. It might take longer for P_1 to update P_3 cache, so it might still print out 0

However, if P_2 is allowed to go on pass the read of A, and write B before confirming that the new A value is seen by *all* other processes, then P_3 may read the new value of B but read the old value of A.

Global Order:	*P_2's View:*	*P_3's View*
W_1(A=1)	W_1(A=1)	R_2(A=1)
R_2(A=1)	R_2(A=1)	W_2(B=1)
W_2(B=1)	W_2(B=1)	R_3(B=1)
R_3(B=1)	R_3(B=1)	r_3(A=0)
R_3(A=1)	R_3(A=1)	W_1(A=1)
		(Not allowed!)

Write atomicity enforces for the update to be propagated before any other operation happens.

And solves this problem.

Sufficient Conditions for Preserving SC

1. Every process issues memory operations in program order.
2. After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.
3. After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing the next operation.

Write atom → 2 and 3 are not enough on their own

Combination must be used to satisfy SC

The third condition is to ensure write atomicity.

Implications of SC: // More harm?

1. ▸ The compiler cannot change the order of memory operations.
2. ▸ The processor cannot use any out-of-order processing technique.

(2) → turn of all optimizations

Common Compiler Optimizations that Violate SC

All these needs to be turned off (usually runned by default?)

- Constant propagation
- Common subexpression elimination
- Loop transformations
- Instruction scheduling
- Register allocation → LOOK IT UP

Any of these might violate SC

Example: Register allocation

P_1:	P_2:	P_1:	P_2:

`B = 0;`
`A = 1;`
`u = B;`
	`A = 0;`
	`B = 1;`
	`v = A;`
		`r1 = 0;`
		`A = 1;`
		`u = r1;`
		`B = r1;`
			`r2 = 0;`
			`B = 1;`
			`v = r2;`
			`A = r2;`

already out of order
u is assigned first

Relaxing Program Order Requirements

Nice but too restrictive! → that is the reason SC not supported by hardware. Sometimes can be implemented by software

There are four types of program orders between memory operations, *write-to-read*, *write-to-write*, *read-to-write*, and *read-to-read*. What would happen if we relax some of these orders? These operations not necessarily need to be done to the same variable

Relaxed Memory Consistency Models:

One step down

- *Total Store Ordering (TSO)*
  — Relaxing write-to-read program order
- *Processor Consistency (PC)* (IBM uses this model)
  — Similar to TSO, but does not guarantee write atomicity
- *Partial Store Ordering (PSO)* (Intel)
  — Relaxing write-to-read and write-to-write program orders

Effects on real progcams.

Relaxing Program Order Examples

Would it work under TSO? Can we expect 1 to print out in both cases

Assume all variables start out with value 0.

No write-read so SC is ok!

P_1:

```
A = 1; write
B = 1; write
```

P_2:

```
print B; read
print A; read
```

P_1:

```
A = 1;  write
flag = 1; write
```

P_2:

```
while (flag==0);   read
print A;           read
```

- Relaxing write-to-read program order is OK with these examples.

Relaxing Program Order Examples (cont.)

Works under TSO but not under PC

P_1:

```
A = 1;   write
```

P_2:

```
while (A==0);   read
B = 1;          write
```

P_3:

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

- Relaxing write-to-read program order is still OK if write-atomicity is guaranteed.

P_1:

```
A = 1;       write
print B;     read
```

P_2:

```
B = 1;       write
print A;     read
```

two write-reads
no guarantee at all by TSO

- Under SC, printed values of A and B cannot be both 0. However, relaxing write-to-read program order would allow it to happen.

Two ways to relax → hardware software

Additional Relaxed Memory Consistency Models

The previous relaxed memory consistency models do not work for all cases the way we'd like. Let's try a slightly different way of thinking.

P_1:

```
1  A = 1;   Done some work
   B = 1;
2  flag = 1;//sync
```

P_2:

```
while (flag == 0); // wait until work is done
u = A;
v = B;
```

Question: Is program order involving *every* statement really necessary for the (intuitive) correctness of this program? → yes

- *Weak Ordering (WO)* — Relaxing all program orders on regular operations; requiring SC on synchronization operations

Separating two types of variables
1. Sync variables
2. All others.

One step further

- *Release Consistency (RC)* — Similar to WO, except that synch operations are further divided into acquires (reads) and releases (writes). //Signal and wait. → next page example

WO allows ① to happen in any order. Which already achieves a lot.

Weak Ordering (Weak Consistency)

- Relaxes all program orders within *regular* operations; maintaining sequential consistency between regular ops and *synchronization* ops, and between sync ops.

P_1:

```
A = 1;
B = 1;
flag = 1;
```

P_2:

```
while (flag == 0);
u = A;
v = B;
```


Weak Ordering (cont.)

Programmers order of relaxation is less strict than hardware.
(Easier to change, allows more control).

- The programmer can control the level of relaxation by labeling less or more operations as "synch ops".

P_1:

```
TOP: while (flag2==0);
     A = 1;
     u = B;
     v = C;
     D = B * C;
     flag2 = 0;
     flag1 = 1;
     goto TOP;
```

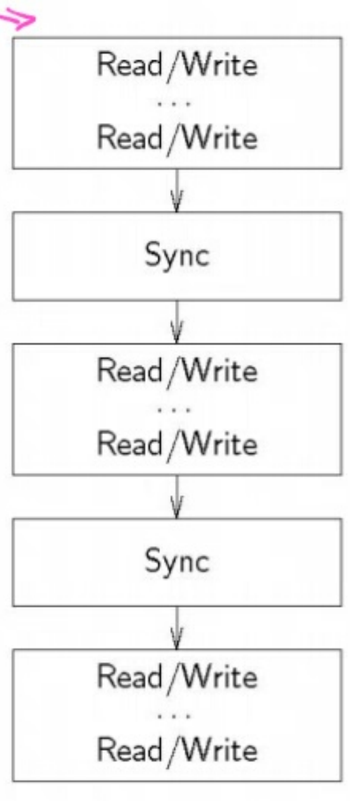
P_2:

```
TOP: while (flag1==0);
     x = A;
     y = D;
     B = 3;
     C = D / B;
     flag1 = 0;
     flag2 = 1;
     goto TOP;
```

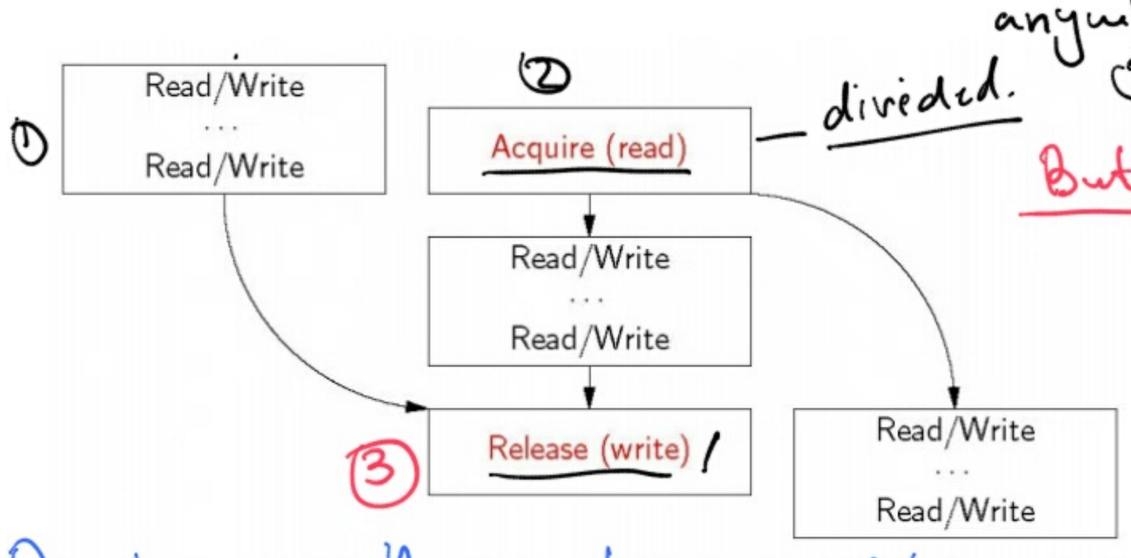
Release Consistency

Further relaxes program orders between regular ops and synch ops. Divides synch ops into *acquires* (reads or read-modify-writes) and *releases* (writes). Some regular ops only need to be ordered with acquires; others only need to be ordered with releases.

WO →



Release consisten model



① can execute anywhere → before ② or after

— divided.

But not after ③

People use these two models, we didn't see it because pThreads hide it (do it explicitely),

→ why this never mentioned? C didn't have it, Java was first to introduce it (but incorrectly).

Memory Consistency and Programming Languages

- In the past, most high-level programming languages did not specify a memory (consistency) model.
  - There was no need, since most programs were single-threaded.
- As more languages support multi-threaded programming, it becomes necessary to define a clear memory model.
  - Java did it multiple times.
  - C++ recently added a memory model (2008).
  - New languages, such as Chapel, mostly include a memory model.
  - OpenMP also has a memory model.

Implication: Programmers must pay attention to a language's memory model. // → important

Nowdays all ~~four~~ new languages have memory model.

OpenMP → not language, set of compiler directives.

strongly need a memory model!