Memory Consistency

Jingke Li

Portland State University

Memory Consistency Problems

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

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

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.

Cache Coherence Problem Example

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

With *Write-Through* 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 will be copied to main memory.
3. Processor P_1 reads x's value again — it gets the old value from its cache!

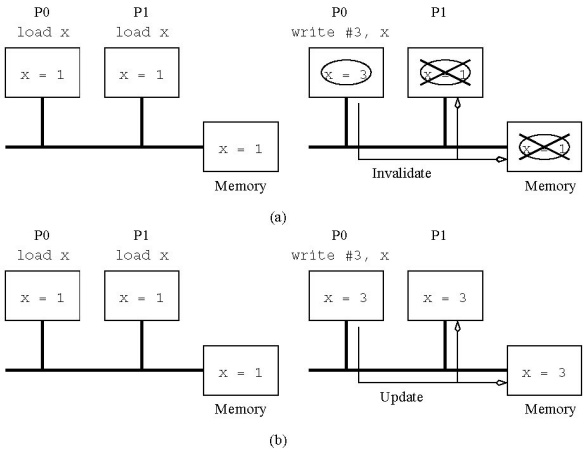
Cache Coherence Problem Example (cont.)

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.)

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.

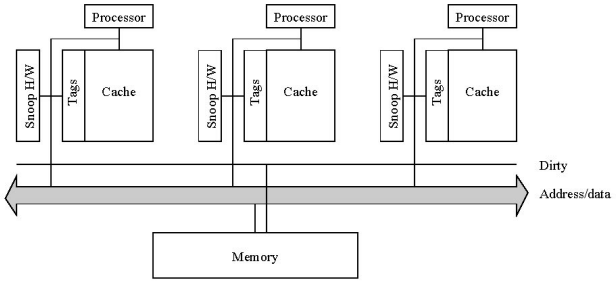
Solutions: Invalid or Update



- *Invalidate* — whenever a data item is written, all other copies in the memory system are invalidated.
- *Update* — whenever a data item is written, all other copies are updated.

Cache-Coherence through Bus Snoopy

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

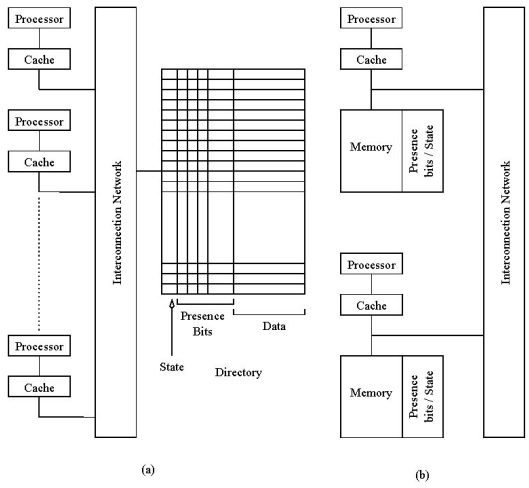


- Each processor's cache controller continuously snoops on the bus watching for relevant transaction (i.e. that involves cache lines of which it has a copy in its cache)
- Once such a transaction is caught, it takes either of the following actions: invalidate the copy in the cache or update the copy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

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



The Memory Consistency Problem

Example:

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)
```

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.

Observations About Memory Operation Orders

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

code order $\neq$ program order

and/or

commit order $\neq$ code order

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.

On a multiprocessor:

- 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?

Motivating Examples

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?

Motivating Examples (cont.)

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

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?

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

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?

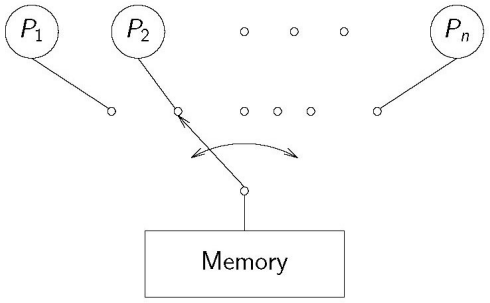
Sequential Consistency (SC)

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,

- 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:



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:

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:

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

A Subtle Point

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:

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)

Sequential Consistency Requirements

Implementing SC requires that the system preserve two intuitive constraints:

- *Program order requirement* —

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

  $$\boxed{\text{Commit order} = \text{Code order} = \text{Program order}}$$

  for all processors.

- *Write atomicity requirement* —

  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.

The Importance of Write Atomicity

P_1:

```
A = 1;
```

P_2:

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

P_3:

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

Intuition expects P_3 to print 1.

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)	$\mathbf{r}_3$(A=0)
R_3(A=1)	R_3(A=1)	W_1(A=1)
		(Not allowed!)

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.

The third condition is to ensure write atomicity.

Implications of SC:

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

Common Compiler Optimizations that Violate SC

- Constant propagation
- Common subexpression elimination
- Loop transformations
- Instruction scheduling
- Register allocation

Example: Register allocation

P_1:

```
B = 0;
A = 1;
u = B;
```

P_2:

```
A = 0;
B = 1;
v = A;
```

P_1:

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

P_2:

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

Relaxing Program Order Requirements

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?

Relaxed Memory Consistency Models:

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

Relaxing Program Order Examples

Assume all variables start out with value 0.

P_1:

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

P_2:

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

P_1:

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

P_2:

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

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

Relaxing Program Order Examples (cont.)

P_1:

```
A = 1;
```

P_2:

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

P_3:

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

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

P_1:

```
A = 1;
print B;
```

P_2:

```
B = 1;
print A;
```

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

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:

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

P_2:

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

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

- *Weak Ordering (WO)* — Relaxing all program orders on regular operations; requiring SC on synchronization operations
- *Release Consistency (RC)* — Similar to WO, except that synch operations are further divided into acquires (reads) and releases (writes).

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.)

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

P_1:

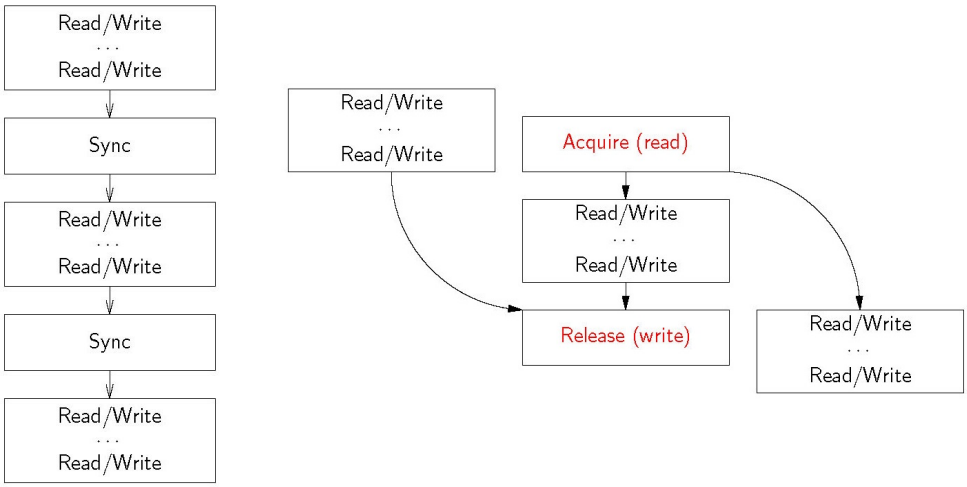
```
TOP: while (flag2==0);
     A = 1;
     u = B;
     v = C;
     D = B * C;
     flag2 = 0;
     flag1 = 1;
     goto TOP;
```

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.



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.