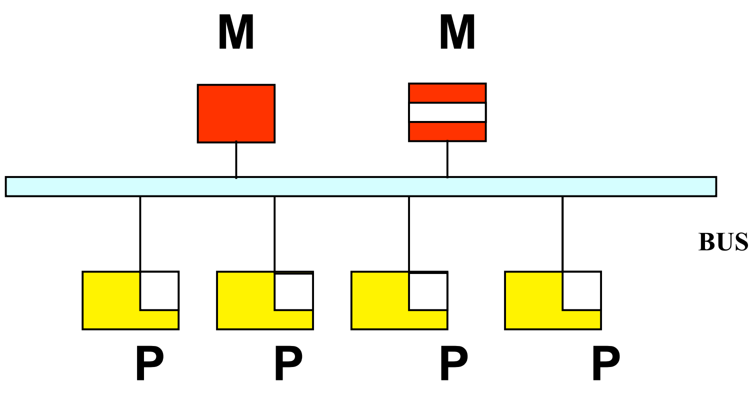
Multiprocessor Cache Coherence



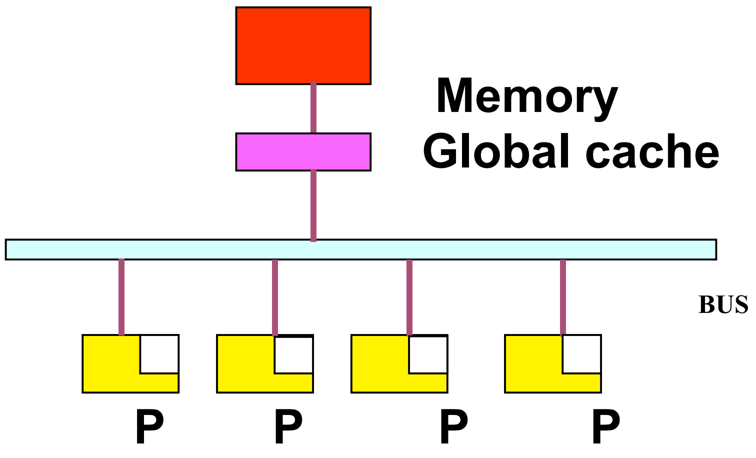
The goal is to make sure that READ(X) returns the most recent value of the shared variable X, i.e. all valid copies of a shared variable are identical.

1. Software solutions
2. Hardware solutions

   - Snooping Cache Protocol (for bus-based machines)
   - Directory Based Solutions (for NUMA machines using a scalable switch)

<u>Software Solutions</u>

Compiler tags data as cacheable and non-cacheable. Only read-only data is considered cachable and put in private cache. All other data are non-cachable, and can be put in a global cache, if available.



Hardware Solution: Snooping Cache

Widely used in bus-based multiprocessors.

The cache controller constantly watches the bus.

Write Invalidate

When a processor writes into C*, all copies of it in other processors are invalidated*. These processors have to read a valid copy either from M, or from the processor that modified the variable.

Write Broadcast

Instead of invalidating, why not broadcast the updated value to the other processors sharing that copy?

This will act as write through for shared data, and write back for private data.

Write broadcast **consumes more bus bandwidth compared** ***to write invalidate*****. Why?**

MESI Protocol (Papamarcos & Patel 1984)

It is a version of the snooping cache protocol.

Each cache block can be in one of four states:

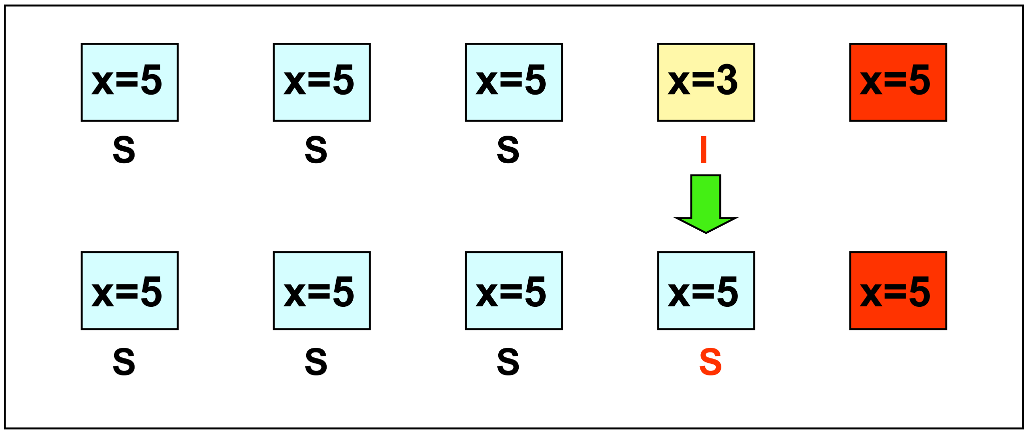
INVALID	Not valid
SHARED	Multiple caches may hold valid copies.
EXCLUSIVE	No other cache has this block, M-block is valid
MODIFIED	Valid block, but copy in M-block is not valid.

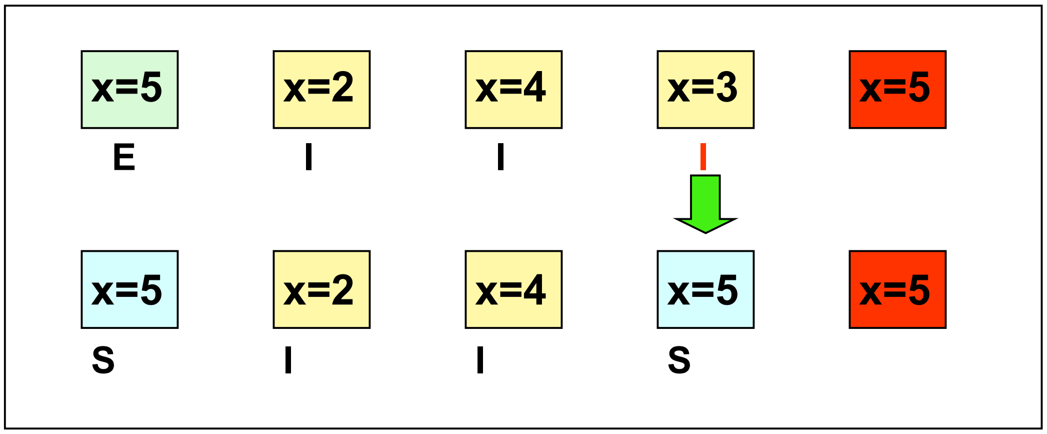
Event	*Local*	*Remote*
Read hit	Use local copy	No action
Read miss	I to S, or I to E	(S,E,M) to S
Write hit	(S,E) to M	(S,E,M) to I
Write miss	I to M	(S,E,M) to I

When a cache block changes its status from M, it first updates the main memory.

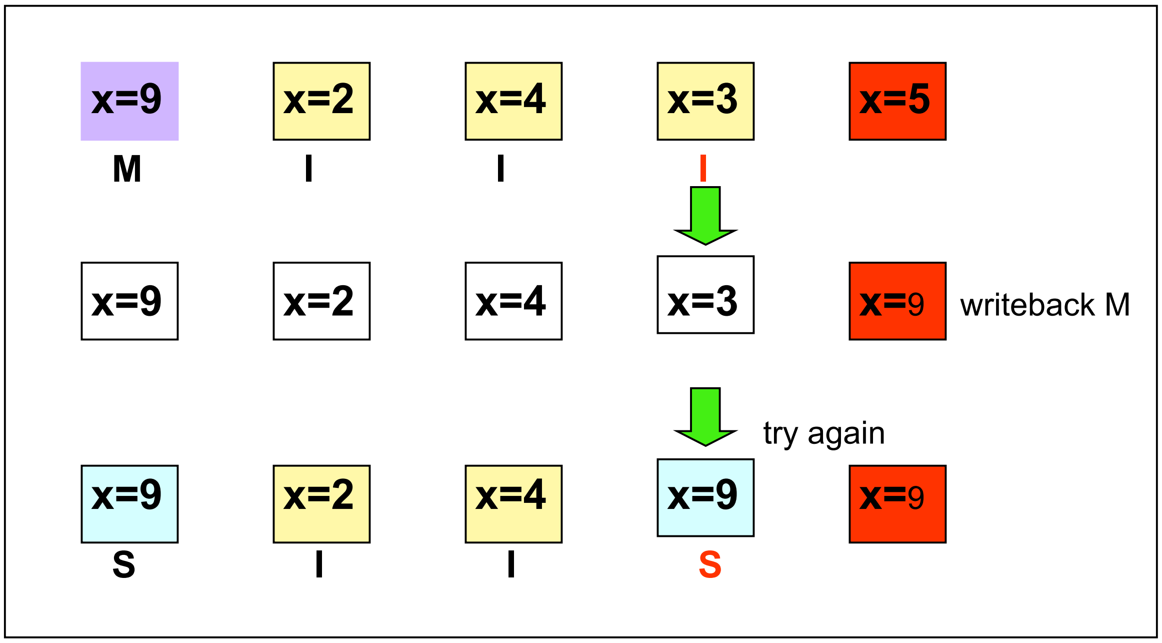
<u>Examples of state transitions under MESI protocol</u>

A. Read Miss



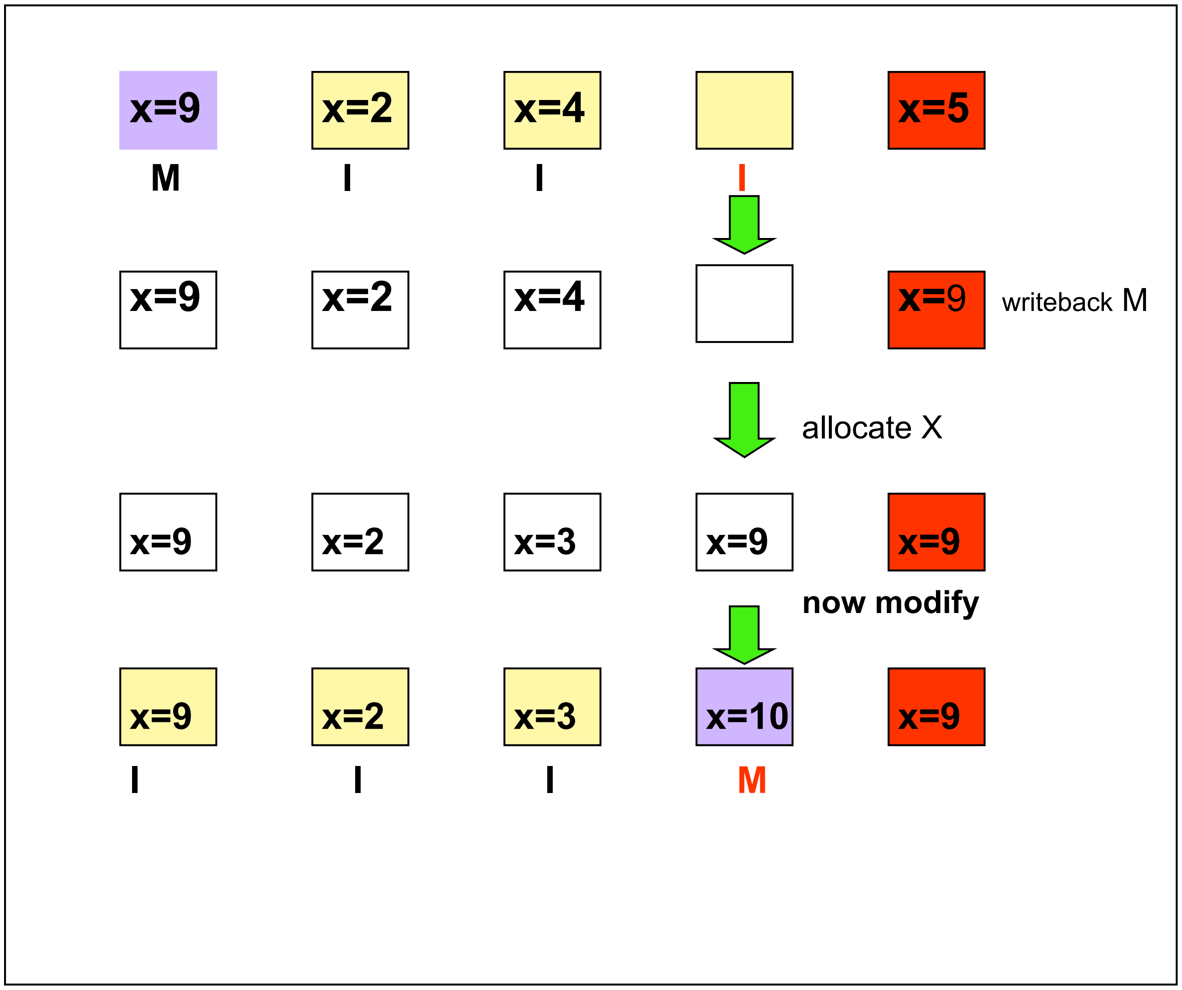


B. More Read Miss



Following the read miss, the holder of the modified copy signals the initiator to try again. Meanwhile, it seizes the bus, and write the updated copy into the main memory.

C. Write Miss


x=9
M
x=2
I
x=4
I
I
x=5
x=9
x=2
x=4
x=9
writeback M
allocate X
x=9
x=2
x=3
x=9
x=9
now modify
x=9
x=2
x=3
x=10
x=9
I
I
I
M

Directory-based cache coherence

The snooping cache protocol does not work if there is no bus. Large-scale shared memory multiprocessors may connect processors with memories through switches.

A directory has to beep track of the states of the shared variables, and oversee that they are modified in a consistent way. Maintenance of the directory in a distributed environment is another issue.

Naïve solutions may lead to deadlock. Consider this:

P1 has a read miss for x2 (local to P2)
P2 has a read miss for x1 (local to P1)

Each will block and expect the other process to send the correct value of x: deadlock (!)

Memory Consistency Models

Multiple copies of a shared variable can be found at different locations, and any write operation must update *all of them*.

Coherence vs. consistency

Cache coherence protocols guarantee that *eventually* all copies are updated. Depending on how and when these updates are performed, a read operation may sometimes return unexpected values.

Consistency deals with *what values can be returned to the user* by a read operation (may return unexpected values if the update is not complete). **Consistency model** is a contract that defines what a programmer can expect from the machine.

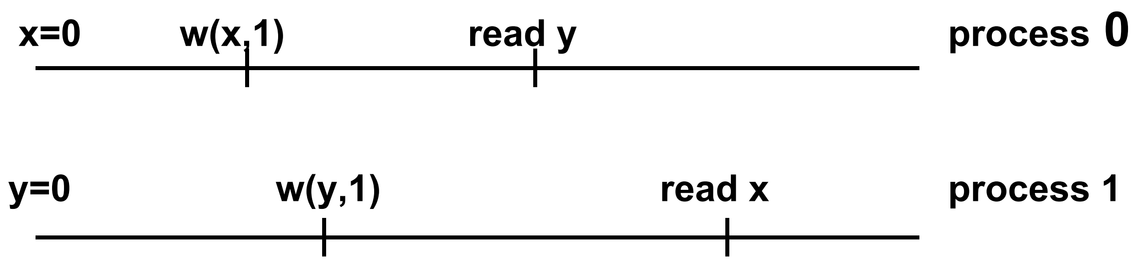
Sequential Consistency

Program 1.

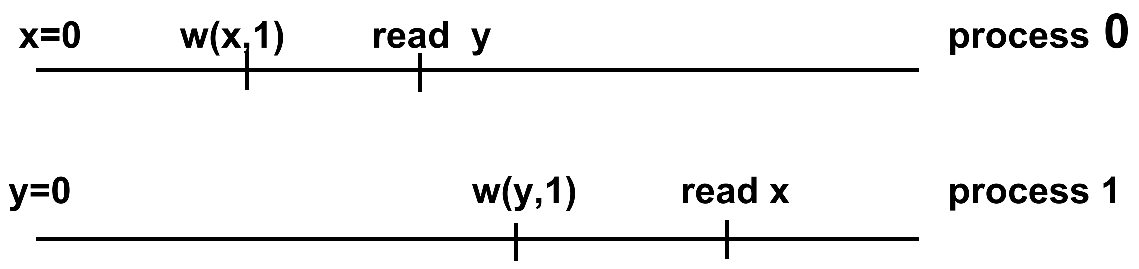
```
process 0                          process 1

{initially,x=0 and y=0}
x:=1;                              y:=1;
if (y=0) then x:=2;                if (x=0) then y:=2;
print  x;                          print  y;
```

If both processes run concurrently, then can we see a printout (x=2, y=2)?

A key question is: Did process 0 read y before process 1 modified it? One possible scenario is:



Here, the final values are: (x=1, y=1)



Here, the final values are: (x=2, y=1)

Properties of SC.

SC1. All operations in a single process are executed in program order.

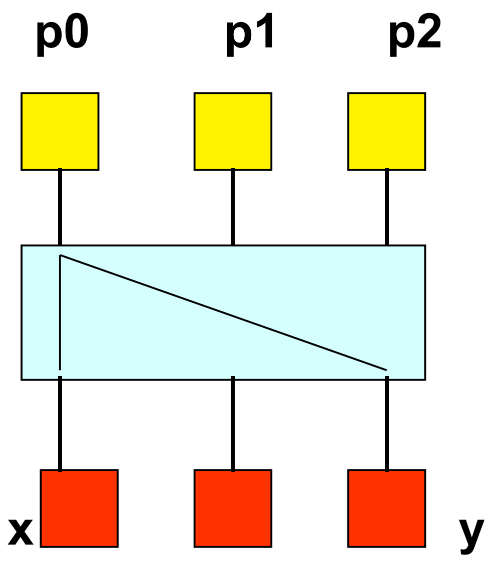
SC2. The result of any execution is the same as if a single sequential order has been maintained among all operations.

Implementing SC

Case 1

Consider a switch-based multiprocessor.

Assume there is no cache.



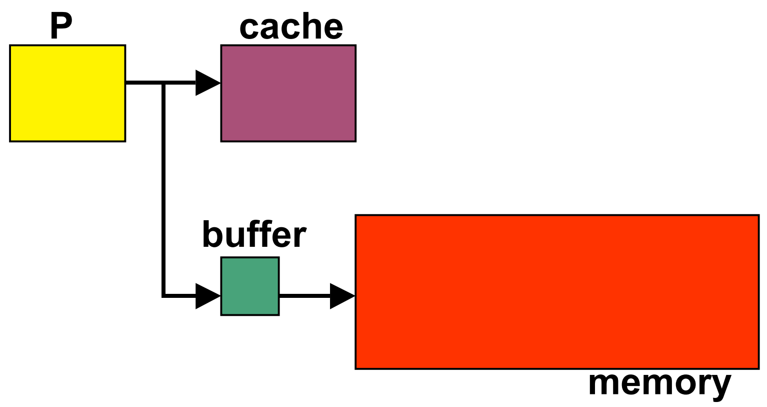
Process 0 executes: (x:=1; y:=2)

To prevent p1 or p2 from seeing these in a different order, p0 must receive an acknowledgement after every write operation.

Case 2

In a multiprocessor where processors have private cache, all invalidate signals must be acknowledged.

Write-buffers and New problems



Go back to the first program now.

Let both x:=1 and y:=1 be written into the write buffers, but before the memory is updated, let the two if statements be evaluated.

Both can be true, and (x:=2, y:= 2) are possible!

***This violates sequential consistency*.**