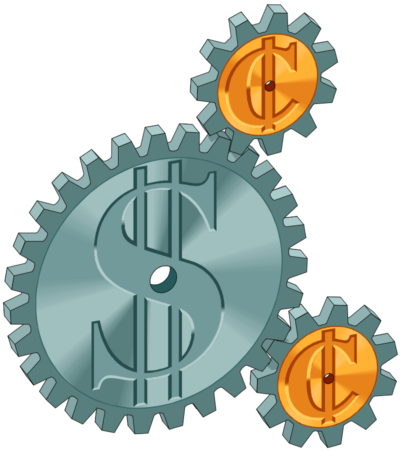
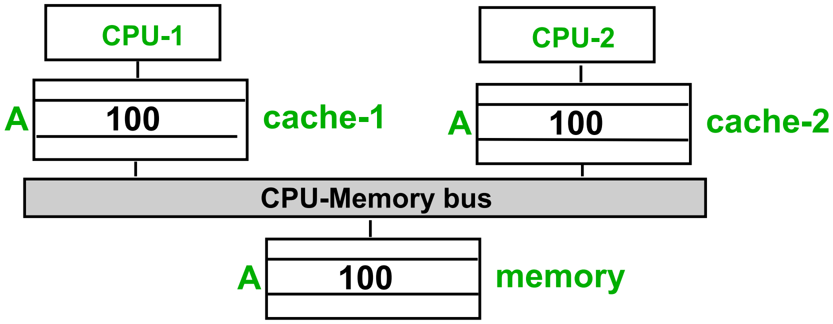
Cache Coherence



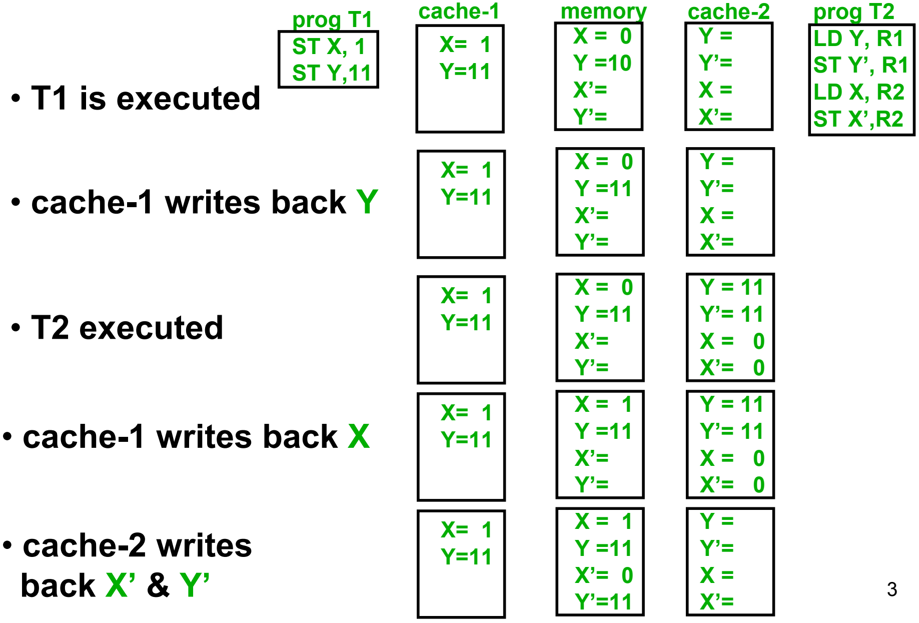
Memory Consistency in SMPs



Suppose CPU-1 updates A to 200.
write-back: memory and cache-2 have stale values
write-through: cache-2 has a stale value

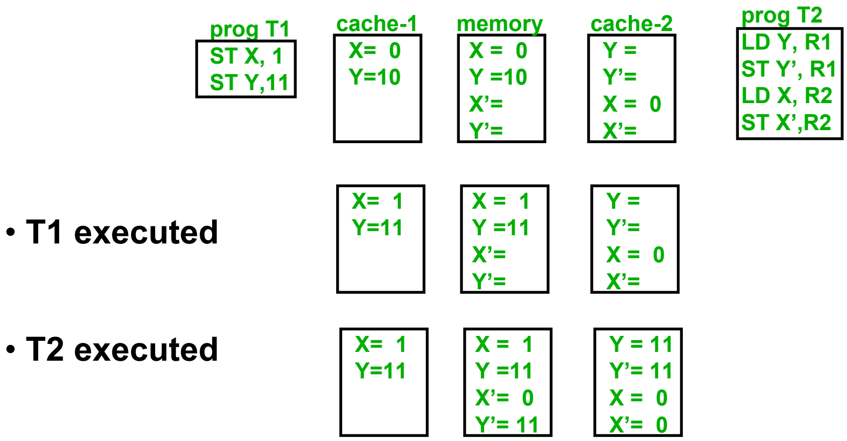
Do these stale values matter?
What is the view of shared memory for programming?

Write-back Caches & SC



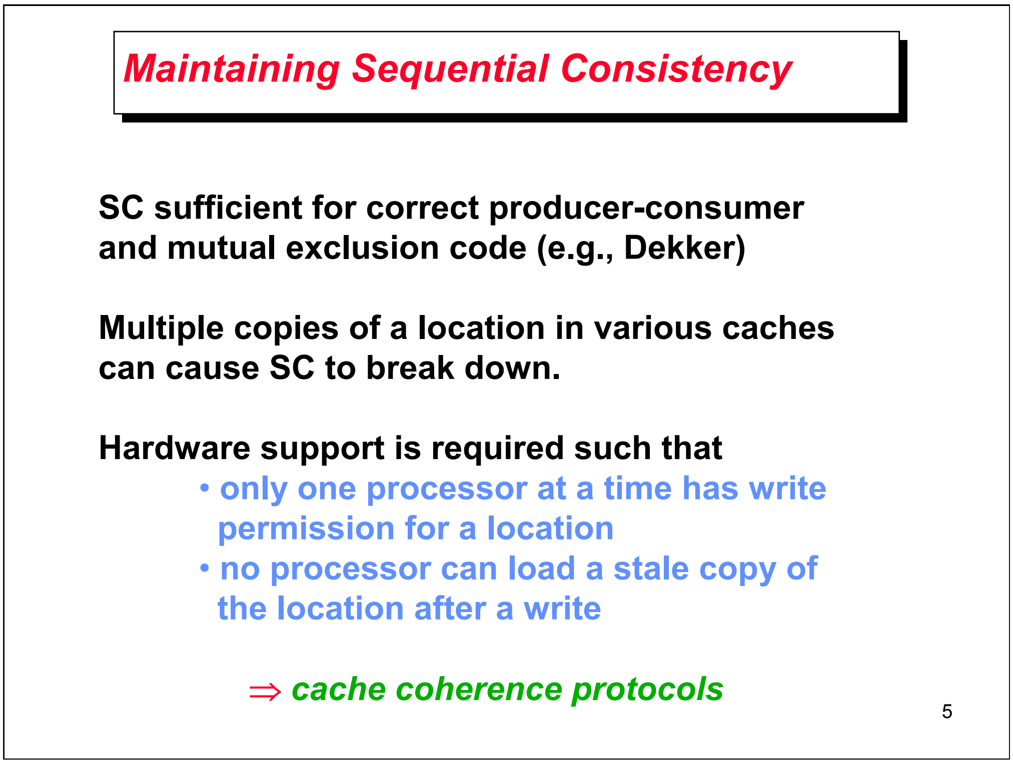
	prog T1	cache-1	memory	cache-2	prog T2
• T1 is executed	ST X, 1 ST Y,11	X= 1 Y=11	X = 0 Y =10 X’= Y’=	Y = Y’= X = X’=	LD Y, R1 ST Y’, R1 LD X, R2 ST X’,R2
• cache-1 writes back Y		X= 1 Y=11	X = 0 Y =11 X’= Y’=	Y = Y’= X = X’=	
• T2 executed		X= 1 Y=11	X = 0 Y =11 X’= Y’=	Y = 11 Y’= 11 X = 0 X’= 0	
• cache-1 writes back X		X= 1 Y=11	X = 1 Y =11 X’= Y’=	Y = 11 Y’= 11 X = 0 X’= 0	
• cache-2 writes back X’ & Y’		X= 1 Y=11	X = 1 Y =11 X’= 0 Y’=11	Y = Y’= X = X’=	

Write-through Caches & SC



	prog T1	cache-1	memory	cache-2	prog T2
	ST X, 1 ST Y,11	X= 0 Y=10	X = 0 Y =10 X'= Y'=	Y = Y'= X = 0 X'=	LD Y, R1 ST Y', R1 LD X, R2 ST X',R2
• T1 executed		X= 1 Y=11	X = 1 Y =11 X'= Y'=	Y = Y'= X = 0 X'=	
• T2 executed		X= 1 Y=11	X = 1 Y =11 X'= 0 Y'= 11	Y = 11 Y'= 11 X = 0 X'= 0	

Write-through caches don't preserve sequential consistency either



Maintaining Sequential Consistency

SC sufficient for correct producer-consumer and mutual exclusion code (e.g., Dekker)

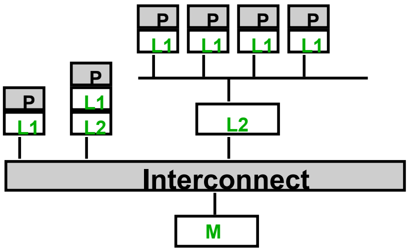
Multiple copies of a location in various caches can cause SC to break down.

Hardware support is required such that

- **only one processor at a time has write permission for a location**
- **no processor can load a stale copy of the location after a write**

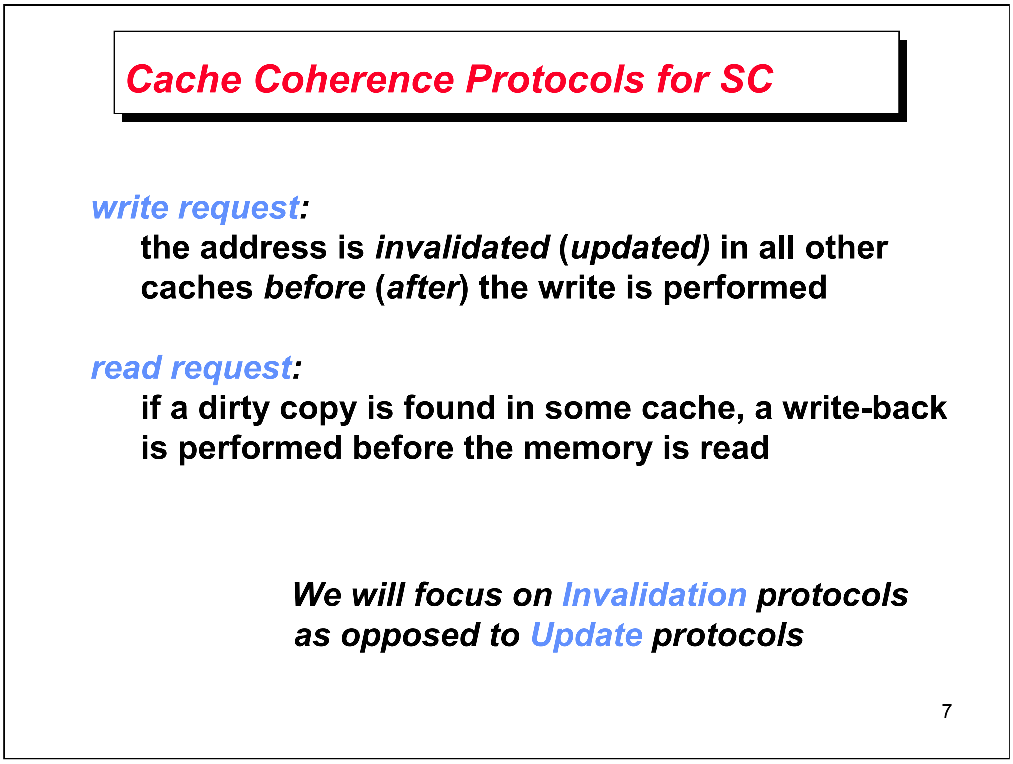
⇒ ***cache coherence protocols***

A System with Multiple Caches



- Modern systems often have hierarchical caches
- Each cache has exactly one parent but can have zero or more children
- Only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,

$$a \in L_i \quad \Rightarrow \quad a \in L_{i+1}$$



Cache Coherence Protocols for SC

***write request*:**

the address is *invalidated* (*updated)* in all other caches *before* (*after*) the write is performed

***read request*:**

if a dirty copy is found in some cache, a write-back is performed before the memory is read

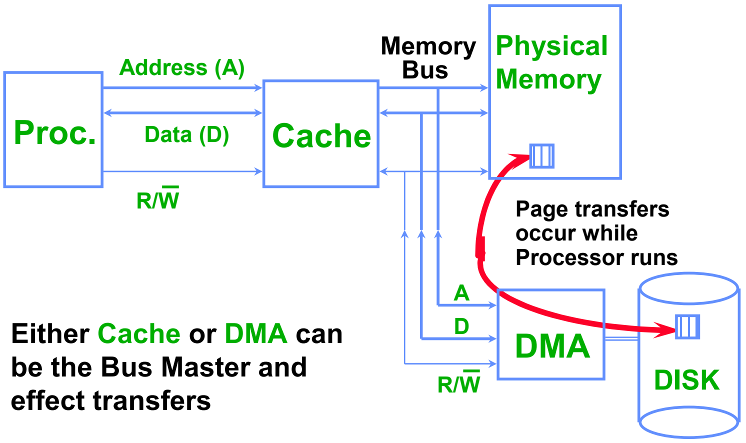
We will focus on Invalidation protocols as opposed to Update protocols

Update protocols, or write broadcast. Latency between writing a word in one processor

and reading it in another is usually smaller in a write update scheme.

But since bandwidth is more precious, most multiprocessors use a write invalidate scheme.

Warmup: Parallel I/O

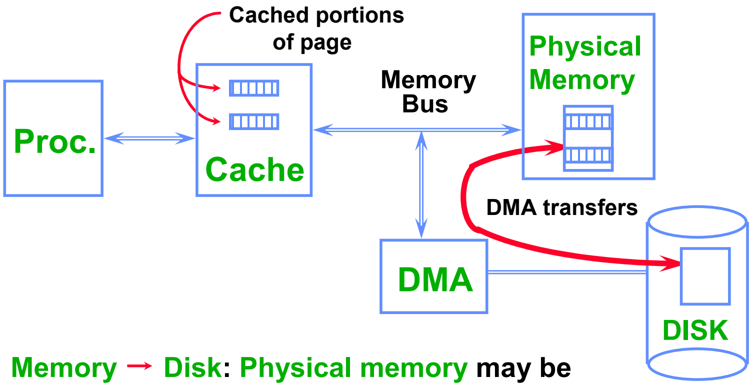


Page transfers occur while Processor runs

Either Cache or DMA can be the Bus Master and effect transfers

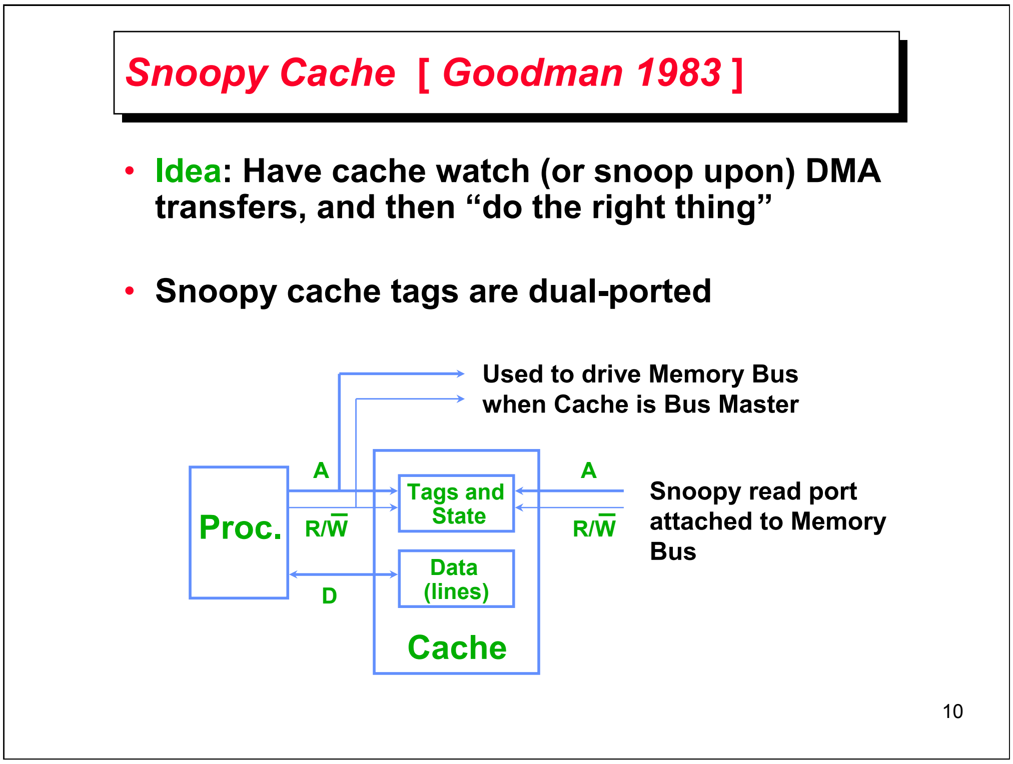
DMA stands for Direct Memory Access

Problems with Parallel I/O



Memory → Disk: Physical memory may be stale if Cache is _________.

Disk → Memory: Cache may have data corresponding to ________.



Snoopy Cache [Goodman 1983]

- Idea: Have cache watch (or snoop upon) DMA transfers, and then “do the right thing”
- Snoopy cache tags are dual-ported

Used to drive Memory Bus when Cache is Bus Master

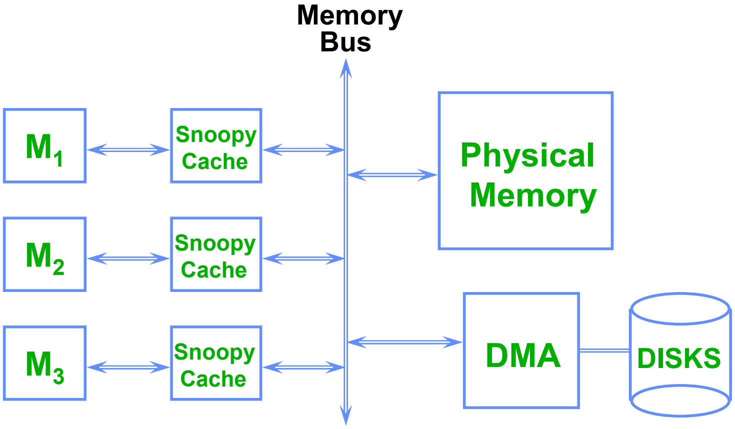
Snoopy read port attached to Memory Bus

A snoopy cache works in analogy to your snoopy next door neighbor, who is always watching to see what you're doing, and interfering with your life. In the case of the snoopy cache, the caches are all watching the bus for transactions that affect blocks that are in the cache at the moment. The analogy breaks down here; the snoopy cache only does something if your actions actually affect it, while the snoopy neighbor is *always* interested in what you're up to.

Snoopy Cache Actions

Observed Bus Cycle	Cache State	Cache Action
Read Cycle Memory → Disk	Address not cached	
	Cached, unmodified	
	Cached, modified	
Write Cycle Disk → Memory	Address not cached	
	Cached, unmodified	
	Cached, modified	

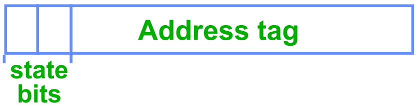
Shared Memory Multiprocessor



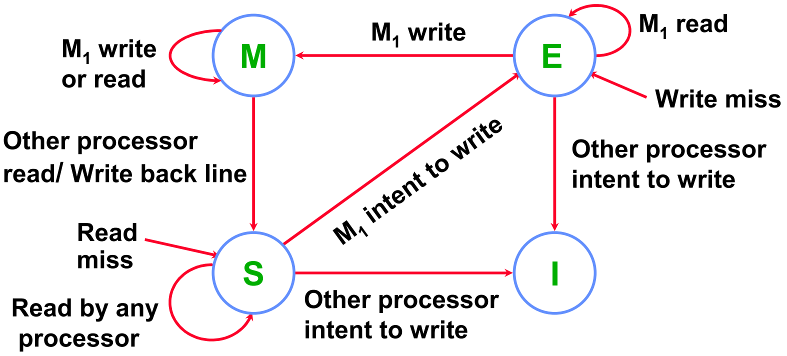
Use snoopy mechanism to keep all processors' view of memory coherent

Cache State Transition Diagram

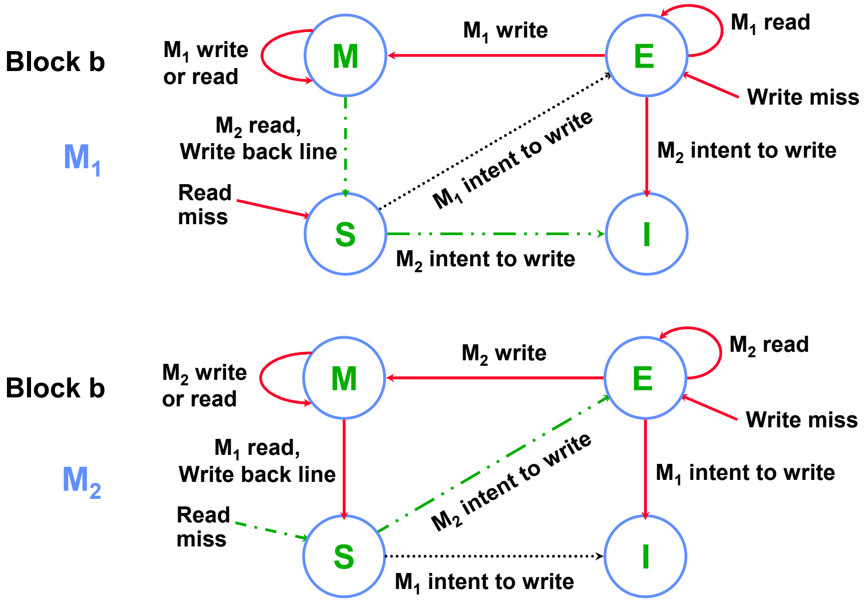
Each cache line tag



M: Modified Exclusive
E: Exclusive, unmodified
S: Shared
I: Invalid



2 Processor Example



Block b

M_1

M_1 write or read

M

M_1 write

E

M_1 read

Write miss

M_2 read, Write back line

M_1 intent to write

M_2 intent to write

Read miss

S

I

M_2 intent to write

Block b

M_2

M_2 write or read

M

M_2 write

E

M_2 read

Write miss

M_1 read, Write back line

M_2 intent to write

M_1 intent to write

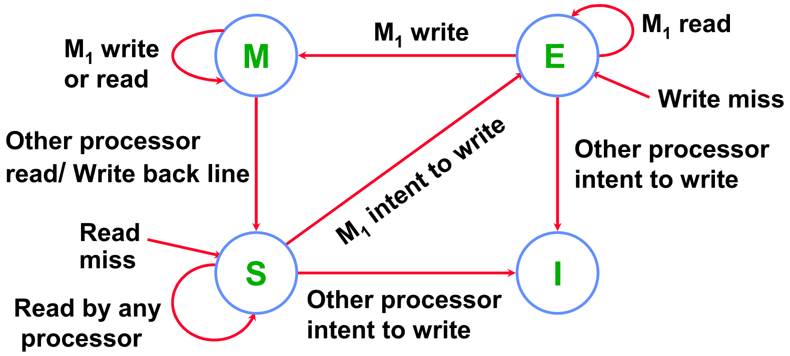
Read miss

S

I

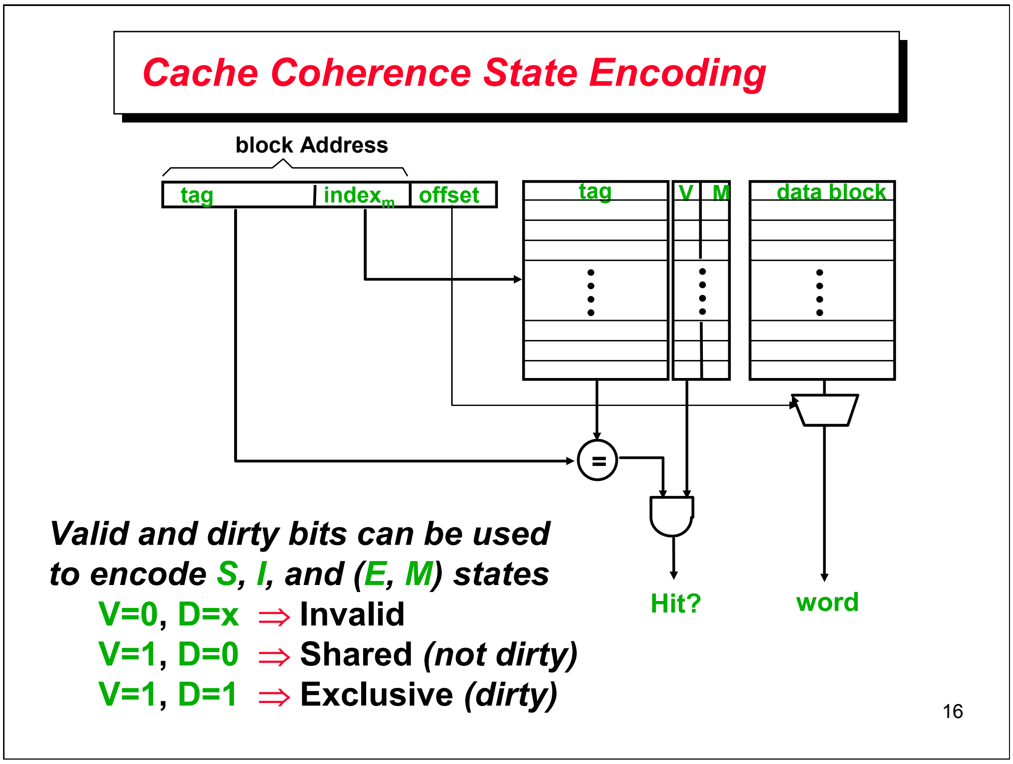
M_1 intent to write

Observation



If a line is in M or E, then no other cache has a valid copy of the line!

- Memory stays coherent, multiple differing copies cannot exist

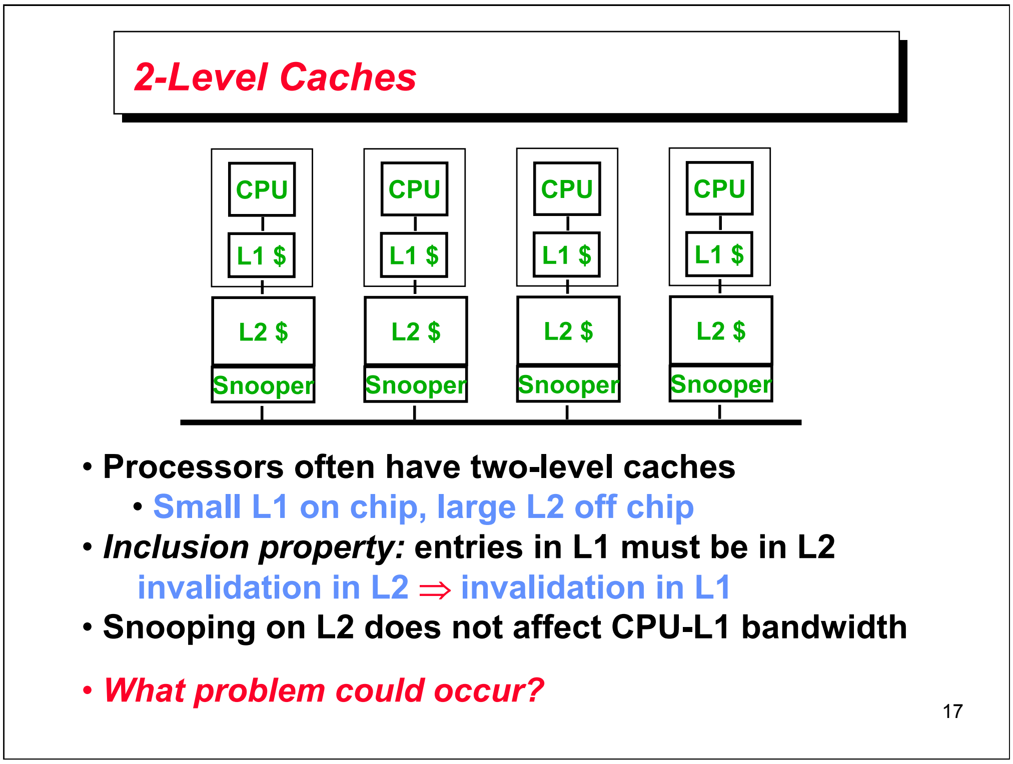


Cache Coherence State Encoding

Valid and dirty bits can be used to encode S, I, and (E, M) states

- **V=0, D=x ⇒ Invalid**
- **V=1, D=0 ⇒ Shared *(not dirty)***
- **V=1, D=1 ⇒ Exclusive *(dirty)***

What does it mean to merge E, M states?



2-Level Caches

CPU
L1 $
L2 $
Snooper
CPU
L1 $
L2 $
Snooper
CPU
L1 $
L2 $
Snooper
CPU
L1 $
L2 $
Snooper

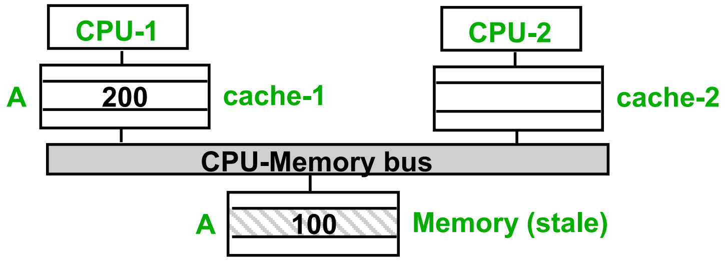
- Processors often have two-level caches
  - Small L1 on chip, large L2 off chip
- ***Inclusion property:*** **entries in L1 must be in L2**

  invalidation in L2 ⇒ invalidation in L1
- **Snooping on L2 does not affect CPU-L1 bandwidth**

- ***What problem could occur?***

Interlocks are required when both CPU-L1 and L2-Bus interactions involve the same address.

Intervention

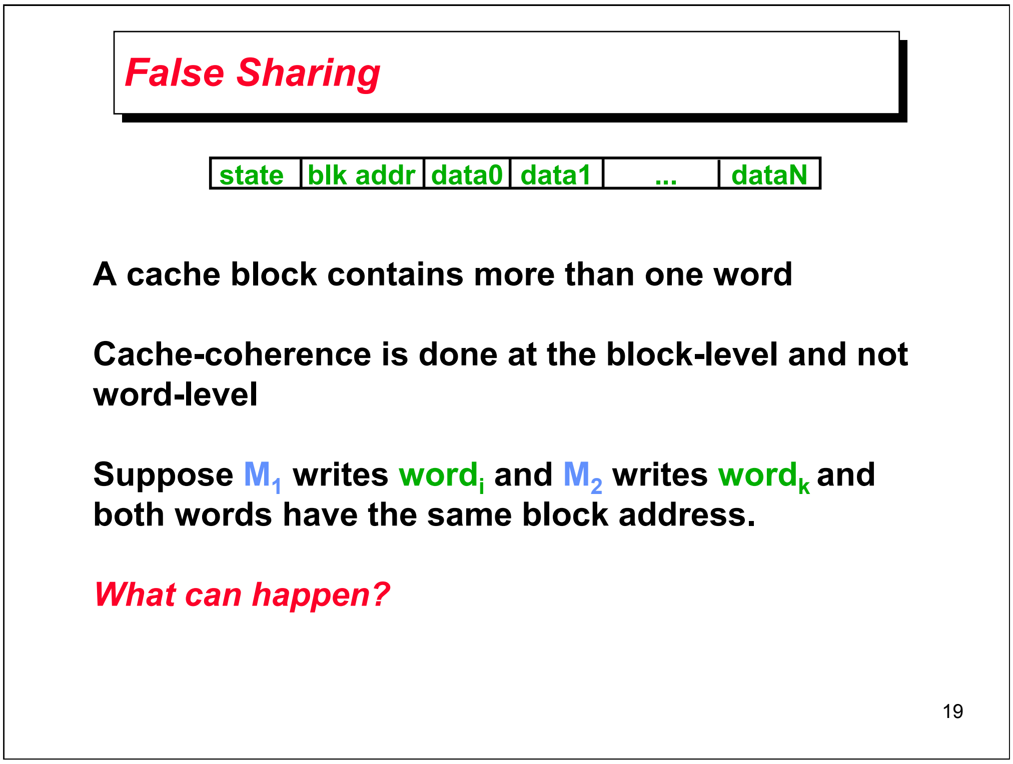


When a read-miss for A occurs in cache-2, a read request for A is placed on the bus

- Cache-1 needs to supply & change its state to shared
- The memory may respond to the request also!

does memory know it has stale data?

Cache-1 needs to intervene through memory controller to supply correct data to cache-2



False Sharing

state	blk addr	data0	data1	...	dataN

A cache block contains more than one word

Cache-coherence is done at the block-level and not word-level

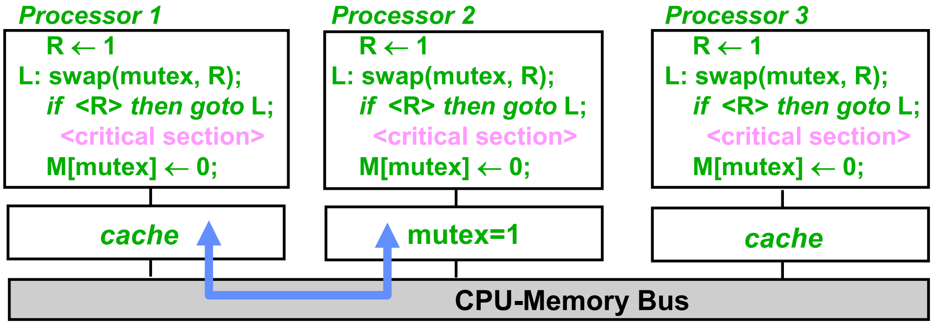
Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same block address.

What can happen?

The block may be invalidated many times unnecessarily because the addresses share a common block.

Synchronization and Caches:

Performance Issues



Processor 1

```
   R ← 1
L: swap(mutex, R);
   if <R> then goto L;
     <critical section>
   M[mutex] ← 0;
```

Processor 2

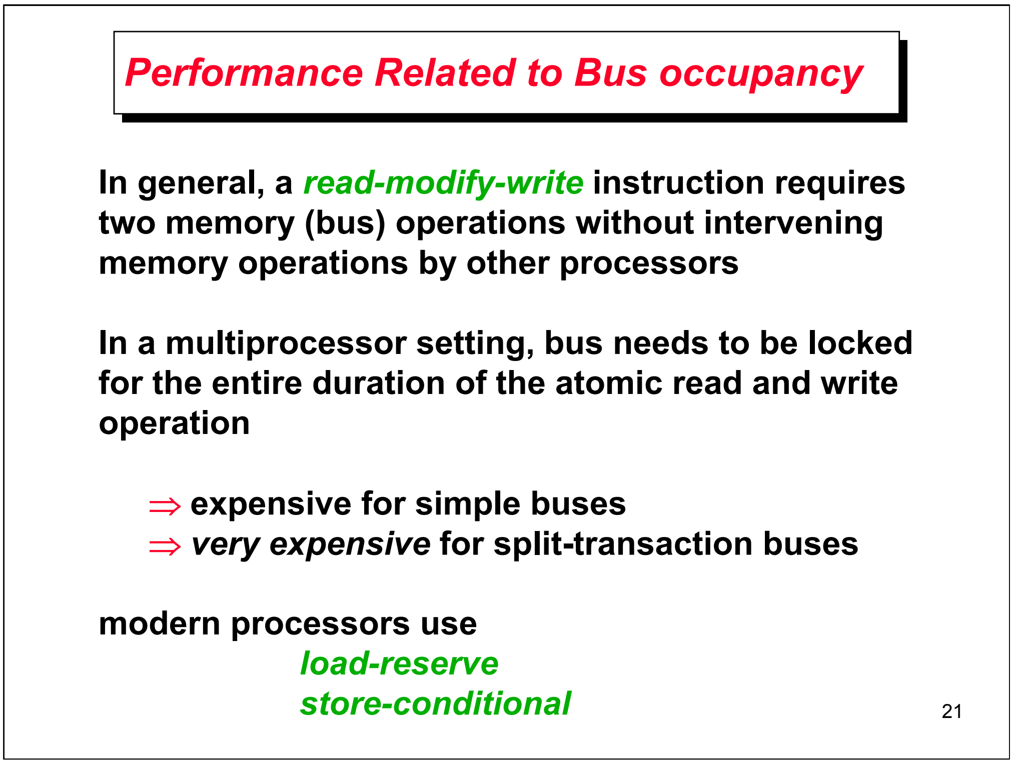
```
   R ← 1
L: swap(mutex, R);
   if <R> then goto L;
     <critical section>
   M[mutex] ← 0;
```

Processor 3

```
   R ← 1
L: swap(mutex, R);
   if <R> then goto L;
     <critical section>
   M[mutex] ← 0;
```

Cache-coherence protocols will cause mutex to *ping-pong* between P1’s and P2’s caches.

Ping-ponging can be reduced by first reading the mutex location *(non-atomically)* and executing a swap only if it is found to be zero.



Performance Related to Bus occupancy

In general, a *read-modify-write* instruction requires two memory (bus) operations without intervening memory operations by other processors

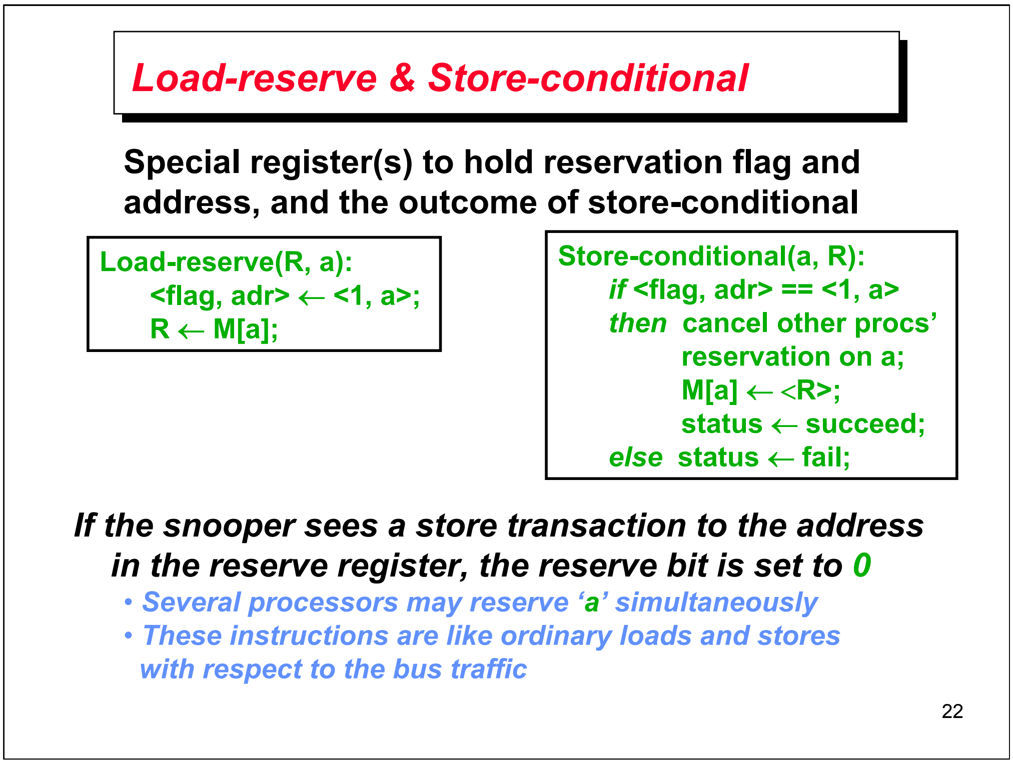
In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation

- ⇒ **expensive for simple buses**
- ⇒ ***very expensive* for split-transaction buses**

modern processors use

load-reserve
store-conditional

Split transaction bus has a read-request transaction followed by a Memory-reply transaction that contains the data.
Split transactions make the bus available for other masters While the memory reads the words of the requested address.
It also normally means that the CPU must arbitrate for the bus To request the data and memory must arbitrate for the bus to Return the data. Each transaction must be tagged. Split Transaction buses have higher bandwidth and higher latency.



Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve(R, a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional(a, R):
    if <flag, adr> == <1, a>
    then  cancel other procs'
          reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

If the snooper sees a store transaction to the address in the reserve register, the reserve bit is set to 0

- *Several processors may reserve '**a**' simultaneously*
- *These instructions are like ordinary loads and stores with respect to the bus traffic*

Performance:

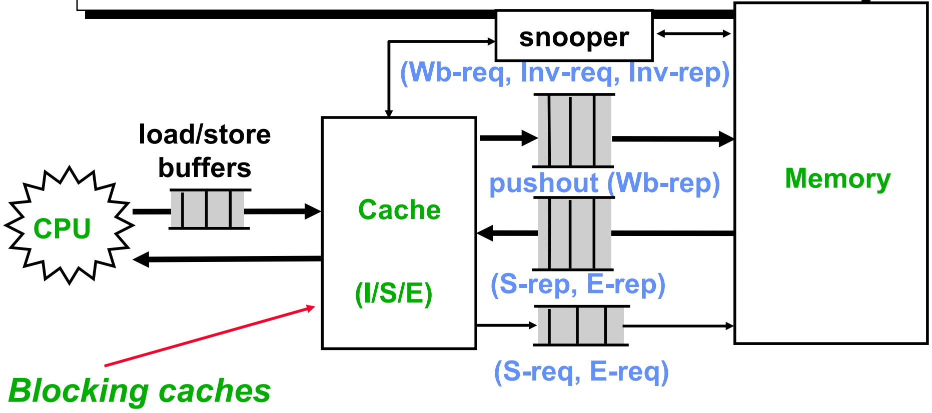
Load-reserve & Store-conditional

The total number of memory (bus) transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- ***increases bus utilization*** **(and reduces processor stall time), especially in split-transaction buses**

- ***reduces cache ping-pong effect*** **because processors trying to acquire a semaphore do not have to perform a store each time**

Out-of-Order Loads/Stores & CC



Blocking caches

One request at a time + CC ⇒ SC

Non-blocking caches

Multiple requests (different addresses) concurrently + CC

⇒ Relaxed memory models

CC ensures that all processors observe the same order of loads and stores to an address