

Lecture 12: Multiprocessor 2: Snooping Protocol, Directory Protocol, Synchronization, Consistency

February 28, 2001
Prof. David A. Patterson
Guest Lecturer: Yujia Jin
Computer Science 252
Spring 2001

2/28/01

CS252/Patterson
Lec 12.1

2/28/01

CS252/Patterson
Lec 12.2

Review: Multiprocessor

- Basic issues and terminology
- Communication: share memory, message passing
- Parallel Application:
 - Commercial workload: OLTP, DSS, Web index search
 - Multiprogramming and OS
 - Scientific/Technical

App	Scaling compu- tation	Scaling communi- cation	Scaling comp- to-comm
FFT	$n \log n/p$	n/p	$\log n$
LU	n^2/p	$n^{1/2} p^{1/2}$	$n^{1/2} p^{1/2}$
Barne s	$n \log n/p$	$n^{1/2} \log n$	$n^{1/2} p^{1/2}$
		$p^{1/2}$	
Ocean	n/p	$n^{1/2} p^{1/2}$	$n^{1/2} p^{1/2}$

- Amdahl's Law: Speedup $\leq 1 / \text{Sequential_Frac}$
- Cache Coherence: serialization

Potential HW Coherency Solutions

- Snooping Solution (Snoopy Bus):
 - Send all requests for data to all processors
 - Processors snoop to see if they have a copy and respond accordingly
 - Requires broadcast, since caching information is at processors
 - Works well with bus (natural broadcast medium)
 - Dominates for small scale machines (most of the market)
- Directory-Based Schemes (discuss later)
 - Keep track of what is being shared in 1 centralized place (logically)
 - Distributed memory => distributed directory for scalability (avoids bottlenecks)
 - Send point-to-point requests to processors via network
 - Scales better than Snooping
 - Actually existed BEFORE Snooping-based schemes

2/28/01

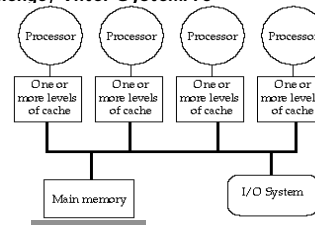
CS252/Patterson
Lec 12.3

2/28/01

CS252/Patterson
Lec 12.4

Bus Snooping Topology

- Memory: centralized with uniform access time ("uma") and bus interconnect
- Examples: Sun Enterprise 5000, SGI Challenge, Intel SystemPro



Basic Snoopy Protocols

- Write **Invalidate** Protocol:
 - Multiple readers, single writer
 - Write to shared data: an invalidate is sent to all caches which snoop and **invalidate** any copies
 - Read Miss:
 - » Write-through: memory is always up-to-date
 - » Write-back: snoop in caches to find most recent copy
- Write **Broadcast** Protocol (typically write through):
 - Write to shared data: broadcast on bus, processors snoop, and **update** any copies
 - Read miss: memory is always up-to-date
- **Write serialization**: bus serializes requests!
 - Bus is single point of arbitration

2/28/01

CS252/Patterson
Lec 12.5

2/28/01

CS252/Patterson
Lec 12.6

Basic Snoopy Protocols

- Write Invalidate versus Broadcast:
 - Invalidate requires one transaction per write-run
 - Invalidate uses spatial locality: one transaction per block
 - Broadcast has lower latency between write and read

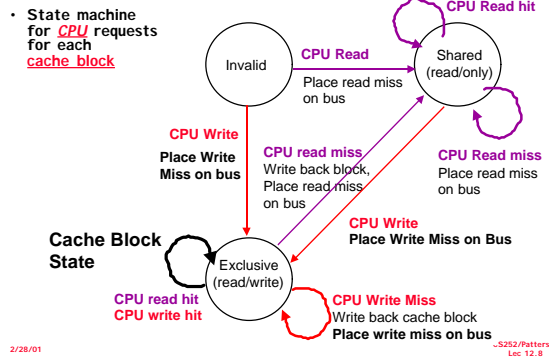
An Example Snoopy Protocol

- Invalidation protocol, write-back cache
- Each block of memory is in one state:
 - Clean in all caches and up-to-date in memory (**Shared**)
 - OR Dirty in exactly one cache (**Exclusive**)
 - OR Not in any caches
- Each cache block is in one state (track these):
 - **Shared** : block can be read
 - OR **Exclusive** : cache has only copy, its writeable, and dirty
 - OR **Invalid** : block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses

2/28/01

CS252/Patterson Lec 12.7

Snoopy-Cache State Machine-I

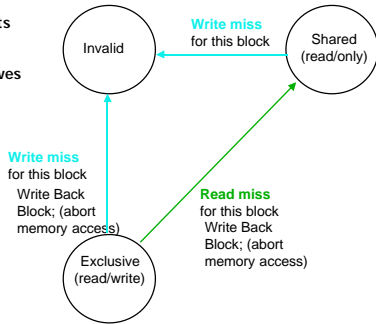


2/28/01

CS252/Patterson Lec 12.8

Snoopy-Cache State Machine-II

- State machine for bus requests for each cache block
- Appendix E? gives details of bus requests

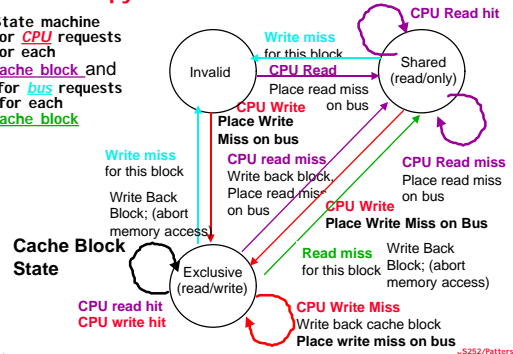


2/28/01

CS252/Patterson Lec 12.9

Snoopy-Cache State Machine-III

- State machine for CPU requests for each cache block and for bus requests for each cache block



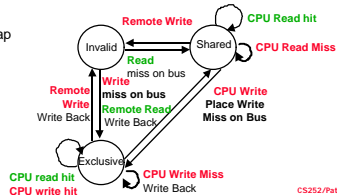
2/28/01

CS252/Patterson Lec 12.10

Example

step	Processor 1			Processor 2			Bus			Memory		
	P1 State	Addr	Value	P2 State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



2/28/01

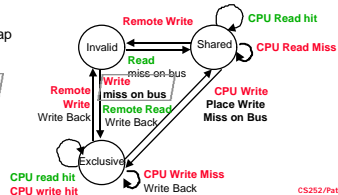
CS252/Patterson Lec 12.11

Example: Step 1

step	P1			P2			Bus			Memory		
	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1: Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2

Active arrow =



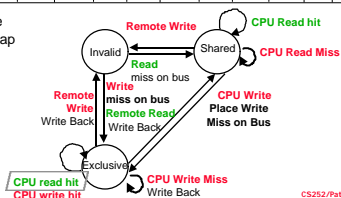
2/28/01

CS252/Patterson Lec 12.12

Example: Step 2

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Proc.	Addr	Value	Memory Addr	Value
P1: Write 10 to A1	Excl	A1	10				WtMs	P1	A1			
P1: Read A1	Excl	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



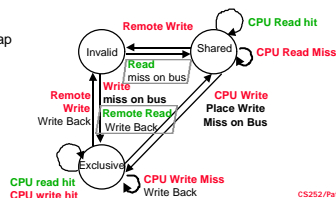
2/28/01

CS252/Patterson Lec 12.13

Example: Step 3

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Proc.	Addr	Value	Memory Addr	Value
P1: Write 10 to A1	Excl	A1	10				WtMs	P1	A1			
P1: Read A1	Excl	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.



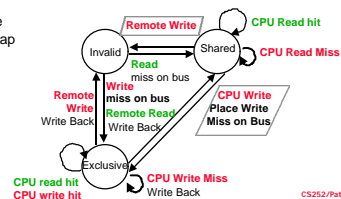
2/28/01

CS252/Patterson Lec 12.14

Example: Step 4

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Proc.	Addr	Value	Memory Addr	Value
P1: Write 10 to A1	Excl	A1	10				WtMs	P1	A1			
P1: Read A1	Excl	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



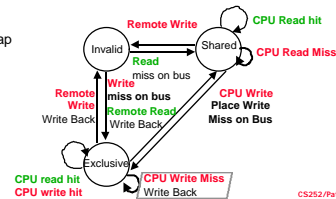
2/28/01

CS252/Patterson Lec 12.15

Example: Step 5

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Proc.	Addr	Value	Memory Addr	Value
P1: Write 10 to A1	Excl	A1	10				WtMs	P1	A1			
P1: Read A1	Excl	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2



2/28/01

CS252/Patterson Lec 12.16

Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
Owned Exclusive	Private Dirty	Modified (private != Memory)	Modified (private != Memory)
Owned Shared	Private Clean	Exclusive (private = Memory)	Exclusive (private = Memory)
Shared	Shared Invalid	Shared (shared = Memory)	Shared (shared = Memory)
Invalid	Invalid	Invalid	Invalid

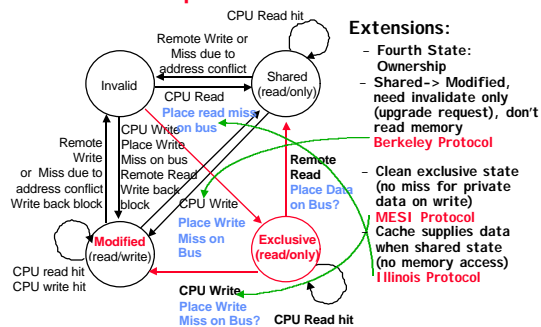
Owner can update via bus invalidate operation
Owner must write back when replaced in cache

If read sourced from memory, then Private Clean
if read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

2/28/01

CS252/Patterson Lec 12.17

Snoop Cache Extensions



Extensions:

- Fourth State: Ownership
- Shared -> Modified, need invalidate only (upgrade request), don't read memory
- Berkeley Protocol
- Clean exclusive state (no miss for private data on write)
- MESI Protocol Cache supplies data when shared state (no memory access)
- Illinois Protocol

2/28/01

CS252/Patterson Lec 12.18

Implementation Complications

- Write Races:
 - Cannot update cache until bus is obtained
 - » Otherwise, another processor may get bus first, and then write the same cache block!
 - Two step process:
 - » Arbitrate for bus
 - » Place miss on bus and complete operation
 - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
 - Split transaction bus:
 - » Bus transaction is not atomic: can have multiple outstanding transactions for a block
 - » Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
 - » Must track and prevent multiple misses for one block
- Must support interventions and invalidations

2/28/01

CS252/Patterson
Lec 12.19

2/28/01

CS252/Patterson
Lec 12.20

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
 - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU cache access:
 - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
 - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
 - » block size, associativity of L2 affects L1

Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and its dirty, so must reply
- Add extra state bit to cache to determine shared or not
- Add 4th state (MESI)

2/28/01

CS252/Patterson
Lec 12.21

2/28/01

CS252/Patterson
Lec 12.22

CS 252 Administrivia

- Chapter 8 survey
- Quiz #1 Wed March 7 5:30-8:30 306 Soda
- La Val's afterward quiz: free food and drink

Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- 1 Cache Coherency solution: non-cached pages
- Alternative: **directory** per cache that tracks state of every block in every cache
 - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
 - PLUS: In memory => simpler protocol (centralized/one location)
 - MINUS: In memory => directory is $f(\text{memory size})$ vs. $f(\text{cache size})$
- Prevent directory as bottleneck?
 - distribute directory entries with memory, each keeping track of which Procs have copies of their blocks

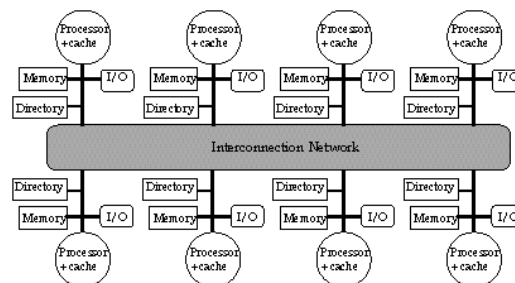
2/28/01

CS252/Patterson
Lec 12.23

2/28/01

CS252/Patterson
Lec 12.24

Distributed Directory MPs



Directory Protocol

- Similar to Snoopy Protocol: Three states
 - **Shared:** " 1 processors have data, memory up-to-date
 - **Uncached** (no processor has it; not valid in any cache)
 - **Exclusive:** 1 processor (**owner**) has data; memory out-of-date
- In addition to cache state, must track **which processors** have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
 - Writes to non-exclusive data => write miss
 - Processor blocks until access completes
 - Assume messages received and acted upon in order sent

2/28/01

CS252/Patterson Lec 12.25

2/28/01

CS252/Patterson Lec 12.26

Directory Protocol

- No bus and don't want to broadcast:
 - interconnect no longer single arbitration point
 - all messages have explicit responses
- Terms: typically 3 processors involved
 - **Local node** where a request originates
 - **Home node** where the memory location of an address resides
 - **Remote node** has a copy of a cache block, whether exclusive or shared
- Example messages on next slide:
P = processor number, A = address

Directory Protocol Messages

Message type	Source	Destination	Msg Content
Read miss	Local cache	Home directory	P, A
- Processor P reads data at address A; make P a read sharer and arrange to send data back			
Write miss	Local cache	Home directory	P, A
- Processor P writes data at address A; make P the exclusive owner and arrange to send data back			
Invalidate	Home directory	Remote caches	A
- Invalidate a shared copy at address A.			
Fetch	Home directory	Remote cache	A
- Fetch the block at address A and send it to its home directory			
Fetch/Invalidate	Home directory	Remote cache	A
- Fetch the block at address A and send it to its home directory; invalidate the block in the cache			
Data value reply	Home directory	Local cache	Data
- Return a data value from the home memory (read miss response)			
Data write-back	Remote cache	Home directory	A, Data
- Write-back a data value for address A (invalidate response)			

2/28/01

CS252/Patterson Lec 12.27

2/28/01

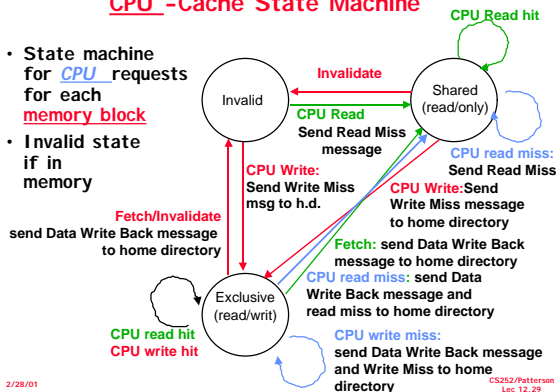
CS252/Patterson Lec 12.28

State Transition Diagram for an Individual Cache Block in a Directory Based System

- States identical to snoopy case; transactions very similar.
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss msg to home directory.
- Write misses that were broadcast on the bus for snooping => explicit invalidate & data fetch requests.
- Note: on a write, a cache block is bigger, so need to read the full cache block

CPU -Cache State Machine

- State machine for **CPU** requests for each **memory block**
- Invalid state if in memory



2/28/01

CS252/Patterson Lec 12.29

2/28/01

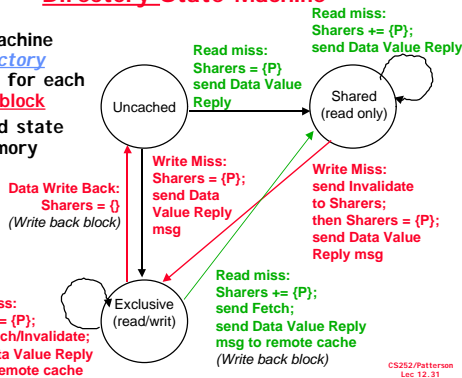
CS252/Patterson Lec 12.30

State Transition Diagram for the Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send msgs to satisfy requests
- Tracks all copies of memory block.
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message.

Directory State Machine

- State machine for **Directory** requests for each **memory block**
- Uncached state if in memory



2/28/01 CS252/Patterson Lec 12.31

Example Directory Protocol

- Message sent to directory causes two actions:
 - Update the directory
 - More messages to satisfy request
- Block is in **Uncached** state: the copy in memory is the current value; only possible requests for that block are:
 - Read miss:** requesting processor sent data from memory & requestor made only sharing node; state of block made Shared.
 - Write miss:** requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
- Block is **Shared** => the memory value is up-to-date:
 - Read miss:** requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
 - Write miss:** requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

2/28/01 CS252/Patterson Lec 12.32

Example Directory Protocol

- Block is **Exclusive**: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) => three possible directory requests:
 - Read miss:** owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor. Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
 - Data write-back:** owner processor is replacing the block and hence must write it back, making memory copy up-to-date (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
 - Write miss:** block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

2/28/01 CS252/Patterson Lec 12.33

Example

	P1	P2	Bus	Directory	Memory									
step	State	Addr	Value	State	Addr	Value	Action	Proc	Addr	Value	Addr	State	Proc	Value
P1: Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

2/28/01 CS252/Patterson Lec 12.34

Example

	P1	P2	Bus	Directory	Memory									
step	State	Addr	Value	State	Addr	Value	Action	Proc	Addr	Value	Addr	State	Proc	Value
P1: Write 10 to A1														
P1: Read A1	Excl	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

2/28/01 CS252/Patterson Lec 12.35

Example

	P1	P2	Bus	Directory	Memory									
step	State	Addr	Value	State	Addr	Value	Action	Proc	Addr	Value	Addr	State	Proc	Value
P1: Write 10 to A1														
P1: Read A1	Excl	A1	10											
P2: Read A1	Excl	A1	10											
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

2/28/01 CS252/Patterson Lec 12.36

Example

	Processor 1		Processor 2		Interconnect		Directory		Memory	
step	P1 State	P1 Addr	P2 State	P2 Addr	Bus Action	Proc Addr	Value	Dir State	Proc1 Value	Mem Value
P1: Write 10 to A1	Excl	A1 10			WbM	P1 A1	0	AI Ex	[P1]	
P1: Read A1	Excl	A1 10			DrRd	P1 A1	0			
P2: Read A1			Shar	A1	RdM	P2 A1				
P2: Write 20 to A1			Shar	A1 10	WbM	P2 A1	10	AI Shar	[P1,P2]	10
P2: Write 40 to A2										

Write Back

A1 and A2 map to the same cache block

2/28/01

CS252/Patterson
Lec 12.37

Example

	Processor 1		Processor 2		Interconnect		Directory		Memory	
step	P1 State	P1 Addr	P2 State	P2 Addr	Bus Action	Proc Addr	Value	Dir State	Proc1 Value	Mem Value
P1: Write 10 to A1	Excl	A1 10			WbM	P1 A1	0	AI Ex	[P1]	
P1: Read A1	Excl	A1 10			DrRd	P1 A1	0			
P2: Read A1			Shar	A1	RdM	P2 A1				
P2: Write 20 to A1			Shar	A1 10	WbM	P2 A1	10	AI Shar	[P1,P2]	10
P2: Write 40 to A2					Inv			Inval	[P1]	10

A1 and A2 map to the same cache block

2/28/01

CS252/Patterson
Lec 12.38

Example

	Processor 1		Processor 2		Interconnect		Directory		Memory	
step	P1 State	P1 Addr	P2 State	P2 Addr	Bus Action	Proc Addr	Value	Dir State	Proc1 Value	Mem Value
P1: Write 10 to A1	Excl	A1 10			WbM	P1 A1	0	AI Ex	[P1]	
P1: Read A1	Excl	A1 10			DrRd	P1 A1	0			
P2: Read A1			Shar	A1	RdM	P2 A1				
P2: Write 20 to A1			Shar	A1 10	WbM	P2 A1	10	AI Shar	[P1,P2]	10
P2: Write 40 to A2					Inv			Inval	[P2]	10

A1 and A2 map to the same cache block

2/28/01

CS252/Patterson
Lec 12.39

2/28/01

CS252/Patterson
Lec 12.40

Implementing a Directory

- We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network (see Appendix E)
- Optimizations:
 - read miss or write miss in Exclusive: send data directly to requestor from owner vs. 1st to memory and then from memory to requestor

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data
- Issues for Synchronization:
 - Uninterruptable instruction to fetch and update memory (atomic operation);
 - User level synchronization operation using this primitive;
 - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

2/28/01

CS252/Patterson
Lec 12.41

2/28/01

CS252/Patterson
Lec 12.42

Uninterruptable Instruction to Fetch and Update Memory

- Atomic exchange:** interchange a value in a register for a value in memory
 - 0 => synchronization variable is free
 - 1 => synchronization variable is locked and unavailable
 - Set register to 1 & swap
 - New value in register determines success in getting lock
 - 0 if you succeeded in setting the lock (you were first)
 - 1 if other processor had already claimed access
 - Key is that exchange operation is indivisible
- Test-and-set:** tests a value and sets it if the value passes the test
- Fetch-and-increment:** it returns the value of a memory location and atomically increments it
 - 0 => synchronization variable is free

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- **Load linked (or load locked) + store conditional**
 - Load linked returns the initial value
 - Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise
- **Example doing atomic swap with LL & SC:**

```
try: mov R3,R4 ; mov exchange value
ll R2,0(R1) ; load linked
sc R3,0(R1) ; store conditional
beqz R3,try ; branch store fails (R3 = 0)
mov R4,R2 ; put load value in R4
```
- **Example doing fetch & increment with LL & SC:**

```
try: ll R2,0(R1) ; load linked
addi R2,R2,#1 ; increment (OK if reg-reg)
sc R2,0(R1) ; store conditional
beqz R2,try ; branch store fails (R2 = 0)
```

2/28/01

CS252/Patterson
Lec 12.43

User Level Synchronization—Operation Using this Primitive

- **Spin locks:** processor continuously tries to acquire, spinning around a loop trying to get the lock


```
lockit: ll R2,#1
        exch R2,0(R1) ;atomic exchange
        bnez R2,lockit ;already locked?
```
- **What about MP with cache coherency?**
 - Want to spin on cache copy to avoid full memory latency
 - Likely to get cache hits for such variables
- **Problem:** exchange includes a write, which invalidates all other copies; this generates considerable bus traffic
- **Solution:** start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):


```
try: ll R2,#1
lockit: lw R3,0(R1) ;load var
        bnez R3,lockit ;not free->spin
        exch R2,0(R1) ;atomic exchange
        bnez R2,try ;already locked?
```

2/28/01

CS252/Patterson
Lec 12.44

Another MP Issue: Memory Consistency Models

- What is consistency? **When** must a processor see the new value? e.g., seems that


```
P1: A = 0;          P2: B = 0;
      .....
      A = 1;          B = 1;
L1: if (B == 0) ... L2: if (A == 0) ...
```
- Impossible for both if statements L1 & L2 to be true?
 - What if write invalidate is delayed & processor continues?
- Memory consistency models: what are the rules for such cases?
 - **Sequential consistency:** result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
 - SC: delay all memory accesses until all invalidates done

2/28/01

CS252/Patterson
Lec 12.45

Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not really an issue for most programs; they are **synchronized**
 - A program is synchronized if all access to shared data are ordered by synchronization operations


```
write (x)
...
release (s) (unlock)
...
acquire (s) (lock)
...
read(x)
```
- Only those programs willing to be nondeterministic are not synchronized: "**data race**": outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

2/28/01

CS252/Patterson
Lec 12.46

Summary

- Caches contain all information on state of cached memory blocks
- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)
- Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory => scalable shared address multiprocessor
 - => Cache coherent, Non uniform memory access

2/28/01

CS252/Patterson
Lec 12.47