Thread-level Parallelism

(continued)

Professor Hugh C. Lauer
Professor Thérèse M. Smith

CS-4515, System Programming Concepts

Programming example

- **Thread A needs to update kernel table**
  - Grabs spinlock()
- **Processor A takes an interrupt or page fault**

  Time passes!

  Much time passes!

- **Thread B also needs to update kernel table**
  - Tries to grab spinlock
  - Spins!
  - … and spins
  - … and spins
  - … and spins
  - … and spins

How does Linux kernel cope with this possibility?

Programming example 2

- **Thread A needs to add an item to application linked list**
  - Calls mutex_lock()
  - Acquires lock
  - Takes page fault while updating list

    Time passes
  - Eventually finishes, calls mutex_unlock()
- **Thread B needs to add another item to same linked list**
  - Calls mutex_lock()
  - Put into wait state
  - Proceeds

Example 2

- **Thread A needs to add an item to application linked list**
  - Calls mutex_lock()
  - Acquires lock
- **Thread B needs to add another item to same linked list**
  - Calls mutex_lock()
  - Put into wait state

Two problems:–

- **Calls to *mutex_lock()*, *mutex_unlock()* involve expensive calls to the operating system**
  - **Orders of magnitude more expensive than the data and operations that they protect**
- **A thread can be forced to wait for long periods of time through no fault of its own**

Goal

- **Provide instructions or other facilities in processor to allow threads to synchronize with each other *without***
  - Expensive trips to the OS kernel
  - Lengthy waits far out of proportion to the task
- **Such operations/facilities are also essential inside of an OS …**
  - Just to manage the bookkeeping for lots of concurrent activity

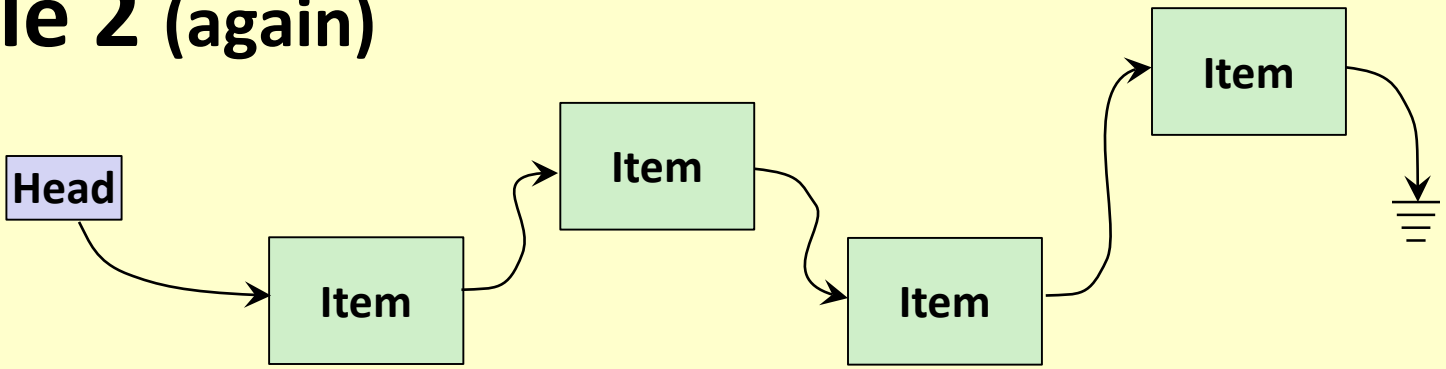
"Atomic" operations

- **Most processors include one or more *atomic* operations in their instruction sets**
  - Test_and_set (IBM 360/370)
    - Set variable to new value *and* return old value — all in a single operation
    - Atomic increment — update variable so no other thread or processor can change it in mid-operation.
    - …
- **It turns out that none of these are "rich" enough to solve the general case of earlier slide!**
    - General proof in Herlihy's book — next slide

Reality

- **Highly multi-threaded applications on systems with many processors need faster, more efficient ways of synchronization**
- **Keep the OS out of critical paths**
- **Provide *wait-free* synchronization methods at application level**
- **Maurice Herlihy, *The Art of Multi-Processor Programming*,** Revised First Edition, Morgan Kaufmann, 2012
  - Original 1st edition is too far out of date to be useful!

Example 2 (again)



Thread A

```
//Add new item to head of list
Addit: LEA  R1, newItem // &newItem
       LL   R2, head
       ST   R2, next(R1) //new -> next
       SC   R1, head
       BNEZ Addit
```

Thread B

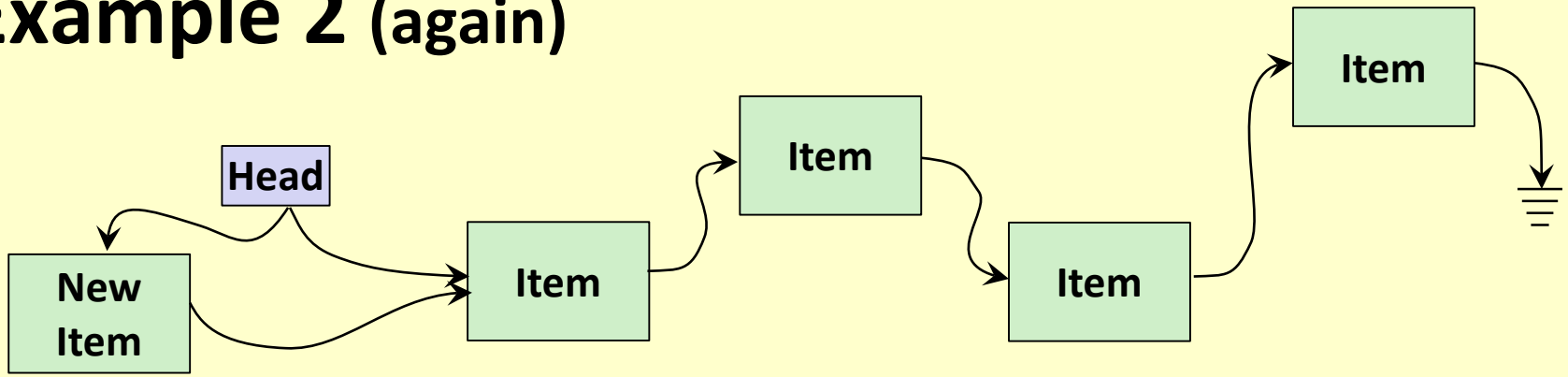
```
//Add new item to head of list
Addit: LEA  R1, newItem // &newItem
       LL   R2, head
       ST   R2, next(R1) //new -> next
       SC   R1, head
       BNEZ Addit
```

Synchronizing operations— `LL` and `SC`

- **`LL` – Load Linked**
  - Load from memory location to a register
  - Hardware keeps track of memory location (somehow)

- **`SC` – Store Conditional**
  - Store into *same* memory location of most recent **`LL`** instruction
  - Return success or failure
  - *Fail* if memory location was changed since **`LL`**
    - e.g., by a another thread (or I/O device!)
  - *Success* if *not* changed since **`LL`**!

Example 2 (again)



Thread A

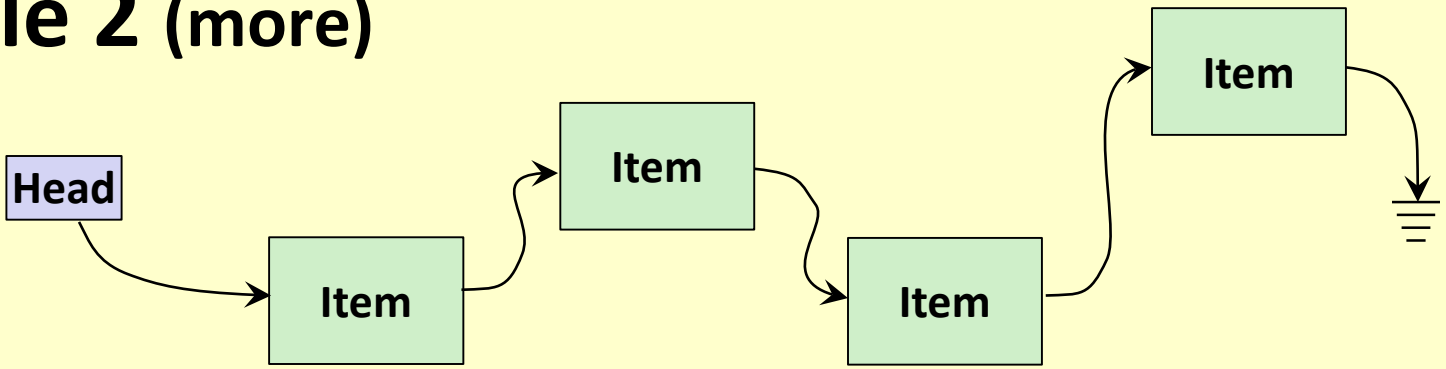
```
//Add new item to head of list
Addit: LEA  R1, newItem // &newItem
       LL   R2, head
       ST   R2, next(R1) //new -> next
       SC   R1, head
       BNEZ Addit
```

Thread B

```
//Add new item to head of list
Addit: LEA  R1, newItem // &newItem
       LL   R2, head
       ST   R2, next(R1) //new -> next
       SC   R1, head
       BNEZ Addit
```

Does this work?
Did we have to invoke OS to do it?

Example 2 (more)



Thread A

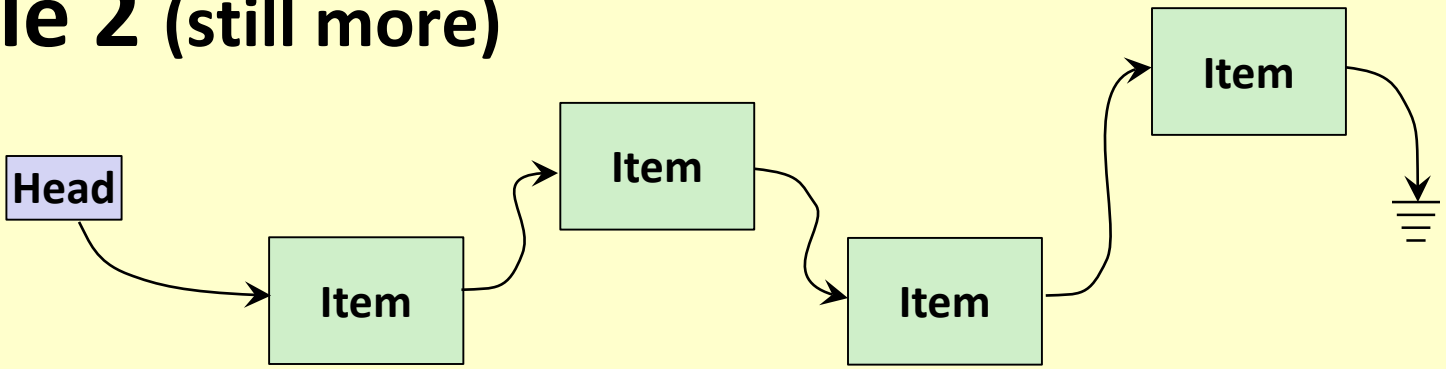
```
//Remove 1st item from list
Del:  LL   R1, head // &firstItem
      LD   R2, next(R1) //head -> next
      SC   R2, head
      BNEZ Del
      // R1 contains &removedItem
```

Thread B

```
//Remove 1st item from list
Del:  LL   R1, head // & firstItem
      LD   R2, next(R1) //head -> next
      SC   R2, head
      BNEZ Del
      // R1 points to removed Item
```

Does this work?
Did we have to invoke OS to do it?

Example 2 (still more)



Thread A

```
//Add new item to head of list
Addit: LEA  R1, newItem // &newItem
       LL   R2, head
       ST   R2, next(R1) //new -> next
       SC   R1, head
       BNEZ Addit
```

Thread B

```
//Remove 1st item from list
Del:   LL   R1, head // & firstItem
       LD   R2, next(R1) //head -> next
       SC   R2, head
       BNEZ Del
       // R1 points to removed Item
```

Does this work?
Did we have to invoke OS to do it?

Definition

- ***Wait-free***
  - Implementation of a “concurrent” object that guarantees that any process/thread can complete an operation in a finite number of steps
  - No process/thread can be prevented from completing the operation by undetected halting or failures of another process
  - No process/thread can be prevented from completing the operation by arbitrary variations in speed

Wait-free (continued)

- **Is Test-and-Set wait-free?**
  - **`Test-and-Set(int *lock)`** :– Atomically fetch the variable pointed to by *lock* and set it to 1 (in memory)
- **What about Fetch-and-Add?**
  - **`Fetch-and-Add(int *lock)`** :– Atomically fetch the variable pointed to by *lock* and increment it by 1 (in memory)
- **What about Exchange?**
  - **`Exchange(int *val, int newVal)`** :– Atomically fetch the variable pointed to by *val* and store the value *newVal* in its place
  - See EXCH instruction, p. 415

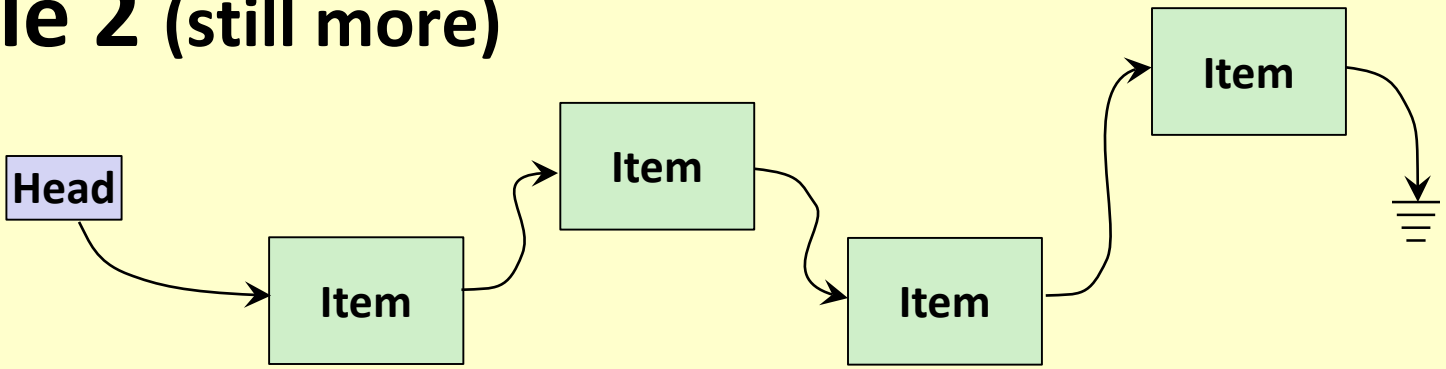
Wait-Free (continued)

- **General theory developed by Maurice Herlihy in 1991**
- **Compare-and-Swap is most general wait-free primitive**
  - `Compare-and-Swap(*var, oldVal, newVal)`:– Atomically fetch the variable pointed to by *var* and replace it by *newVal*, but only if its existing value equals *oldVal*
- **Supports wait-free synchronization among *n* threads**

Compare-and-swap()

- **Equivalent to *LL* + *SC***
  - I.e., Compare-and-swap() can be implemented by LL + SC
  - LL + SC can be implemented by Compare-and-swap()
- **Included in Intel architectures as *CMPEXCH***
- **Included in all IBM architectures since System 370**
  - Early 1970s
- **(*LL* + *SC*) & *CMPEXCH* depend crucially on cache coherency mechanisms**

Example 2 (still more)



Thread A

```
//Add new item to head of list
Addit: LEA  R1, newItem // &newItem
       LL   R2, head
       ST   R2, next(R1) //new -> next
       SC   R1, head
       BNEZ Addit
```

Thread B

```
//Remove 1st item from list
Del:   LL   R1, head // & firstItem
       LD   R2, next(R1) //head -> next
       SC   R2, head
       BNEZ Del
       // R1 points to removed Item
```

Does this solution extend to *n* threads?

Uses of *LL* + *SC*

- **Atomically updating a variable by f(variable)**
- **Atomically linking or unlinking**
- **Easy to do at the head of the list**
  - Only one item needs to be updated (i.e., head)
- **Possible to do in the middle or the tail of the list**
  - Two memory items need to be updated atomically
  - Requires clever approaches — transactions
  - … or transactional memory (after textbook was published!)

Uses of *LL* + *SC* (continued)

- Any other management of shared data structures
- Any other management of *concurrent objects*
- Equivalent to *Compare & Swap*

Implementation of *LL* & *SC*

Questions?

Two classes of cache coherence

Snooping

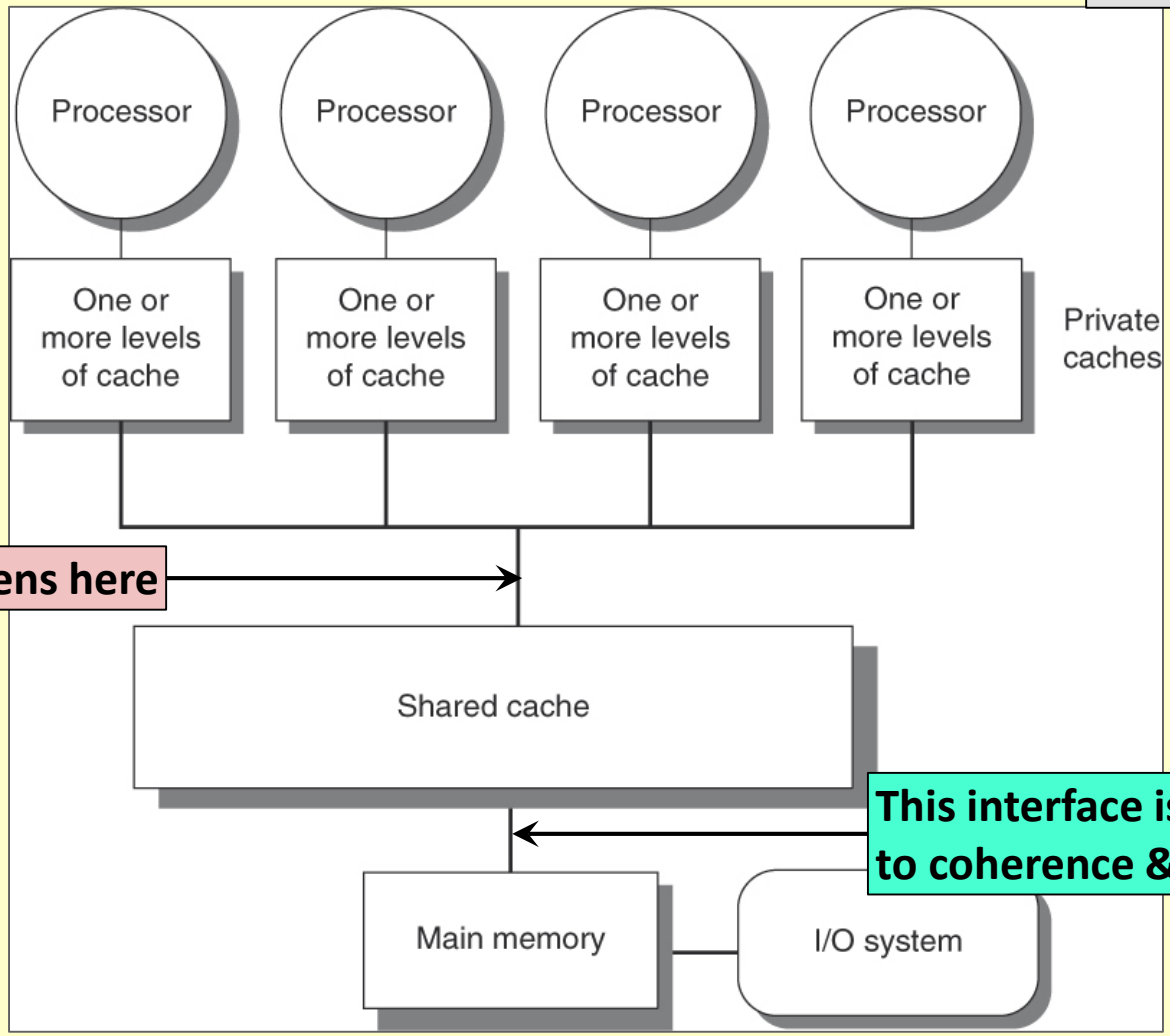
- ***Snooping*** **— Every cache keeps copy of *sharing status* of each block**
  - All memory access via a broadcast medium
  - Cache controllers *snoop* all transactions to update own status of cached blocks
  - *No centralized state*

Directory

- **Sharing status of a block of physical memory is kept in just one location — i.e., the *directory***
  - Any processor needing to share a block *must* query its directory
  - Directory maintains info about who has *every* block

Symmetric Multi-processors

Most single-chip microprocessors use this approach

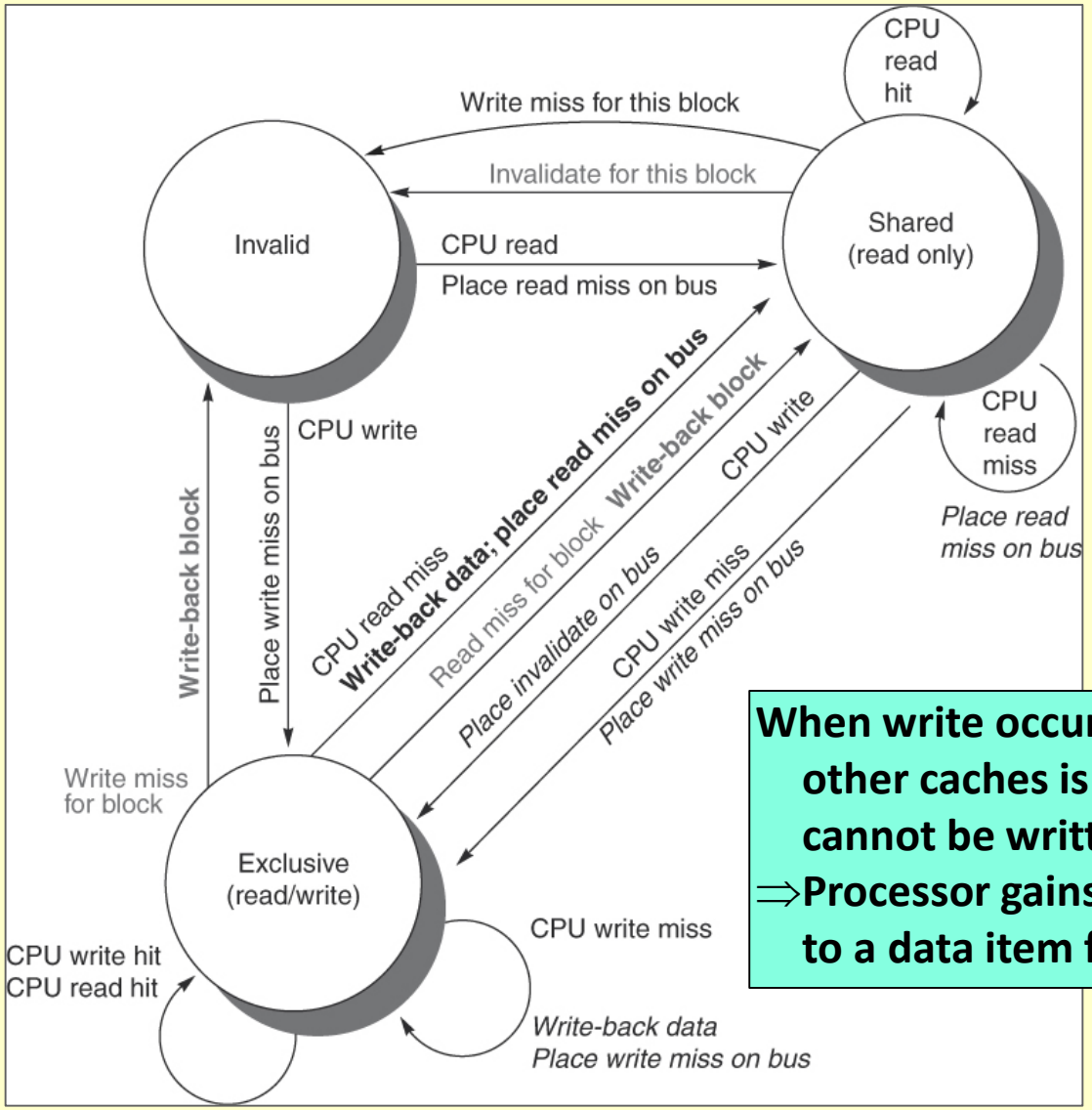


Snooping happens here

This interface is oblivious to coherence & sharing

Three states of cache block

Modified (exclusive)
Shared (readonly)
Invalid



When write occurs, same item in other caches is invalidated — cannot be written
⇒Processor gains exclusive access to a data item for write

Fig 5.7

Principle of write-invalidation

- **Designed for write-back caches**
- **Listens to both processor and bus**
  - Bus ensures serialization
- **Each block in cache has three possible states**
  - *Invalid* – i.e., a cache miss
  - *Shared* – block is potentially shared
    - Blocks in this state guaranteed to be up to date in memory
  - *Modified* – implies that the block is exclusive
    - Not necessarily up to date in memory
- **Assumes that all actions are atomic**
  - At the hardware level

Two components of previous diagram

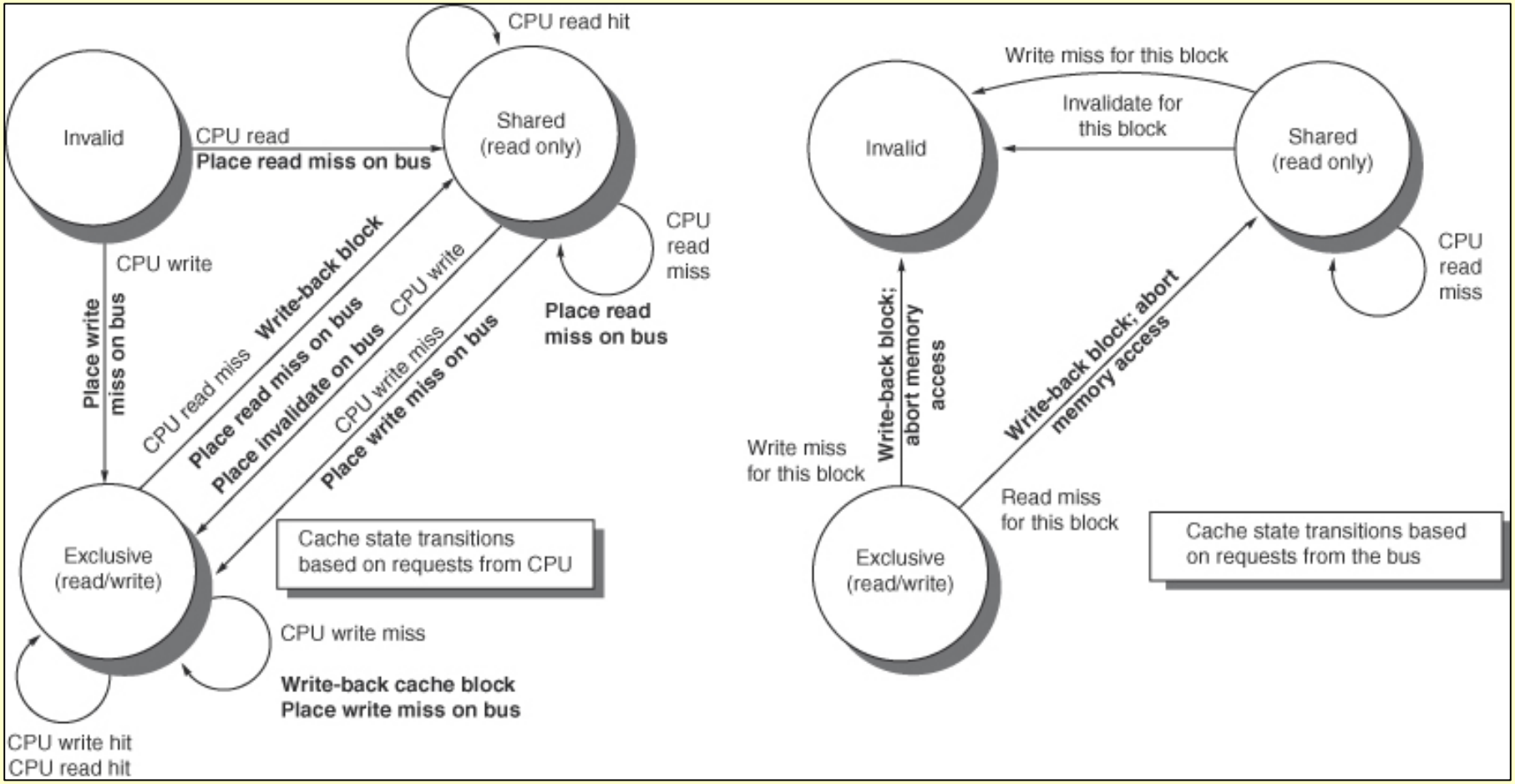


Fig 5.6

Extensions

- **MSI**
  - Modified (i.e., exclusive)
  - Shared
  - Invalid
- **MESI**
  - Add Exclusive but clean
    - Avoids some misses of clean data
- **MOESI**
  - Add owned
    - I.e., modified and dirty

Exclusivity Flag

- May be used to improve snooping
- CPU tracks whether only it has a copy of data
- If this flag is set, other snooping CPUs don't have to check their caches for copies
- Caches must now watch to see if other processors are requesting an exclusive sector
- A write-through of a block invalidates all copies and makes it exclusive again

Architectural Building Blocks

- **Cache block state transition diagram**
  - FSM specifying how disposition of block changes
    - invalid, valid, exclusive
- **Broadcast Medium Transactions (e.g., bus)**
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/addr, data

  ⇒ Every device observes every transaction

- **…**

Architectural Building Blocks (continued)

- …
- **Broadcast medium enforces serialization of read or write accesses ⇒ Write serialization**
  - 1st processor to get medium invalidates others copies
  - Implies cannot complete write until it obtains bus
  - All coherence schemes require serializing accesses to same cache block
- **Also need to find up-to-date copy of cache block**

The "Bus" (or other medium)

- **Shared communication path among *all* processors or their caches**

- **Assumes an arbitration mechanism**
  - Only one node can place transaction on bus at a time
  - If multiple nodes attempt at "same time," one wins, others wait

Limitations on Busses

- ***Bandwidth* – a function of**
  - Physical size
  - Lengths of wires
  - Types of stubs or connections
- ***Latency***
  - Long enough to drive signal from one end to other
  - Long enough to accommodate *arbitration* protocol
- **Alternative:– *switch* with point-to-point connections**
  - Like your Ethernet hub
  - All nodes can see all transactions

Summary — Snooping

- Enough mechanism to implement cache coherence
- Enough mechanism to ensure memory consistency
- Underlying mechanism for implementing *LL* + *SC*
- Not highly scalable to many processors or multi-chip systems

Questions?

Cache Coherence and Synchronization

- **Spin lock**
  - If no coherence:

Spins in L1 cache
Breaks out after Invalidate

```
            DADDUI   R2,R0,#1
lockit:     EXCH     R2,0(R1)    ;atomic exchange
            BNEZ     R2,lockit   ;already locked?
```

  - If coherence:

```
lockit:     LD       R2,0(R1)    ;load of lock
            BNEZ     R2,lockit   ;not available-spin
            DADDUI   R2,R0,#1    ;load locked value
            EXCH     R2,0(R1)    ;swap
            BNEZ     R2,lockit   ;branch if not 0
```

Cache Coherence & Synch (continued)

- **LL + SC**
  - Hypothetical model

```
Addit: LEA   R1, newItem
       LL    R2, head        // Cache block → M or E or O
       ST    R2, next(R1)
       SC    R1, head        // Succeeds only if Cache block still M, E, or O!
       BNEZ  Addit
```

Implemented entirely in L1!
Depends on Cache Coherence mechanism

Inclusion property

- ***Inclusion*** $\Rightarrow$
  - If item in L_i cache, then
    must also be in L_{i+1} cache
  - MSI, MESI, MOESI state must be reflected in all cache levels

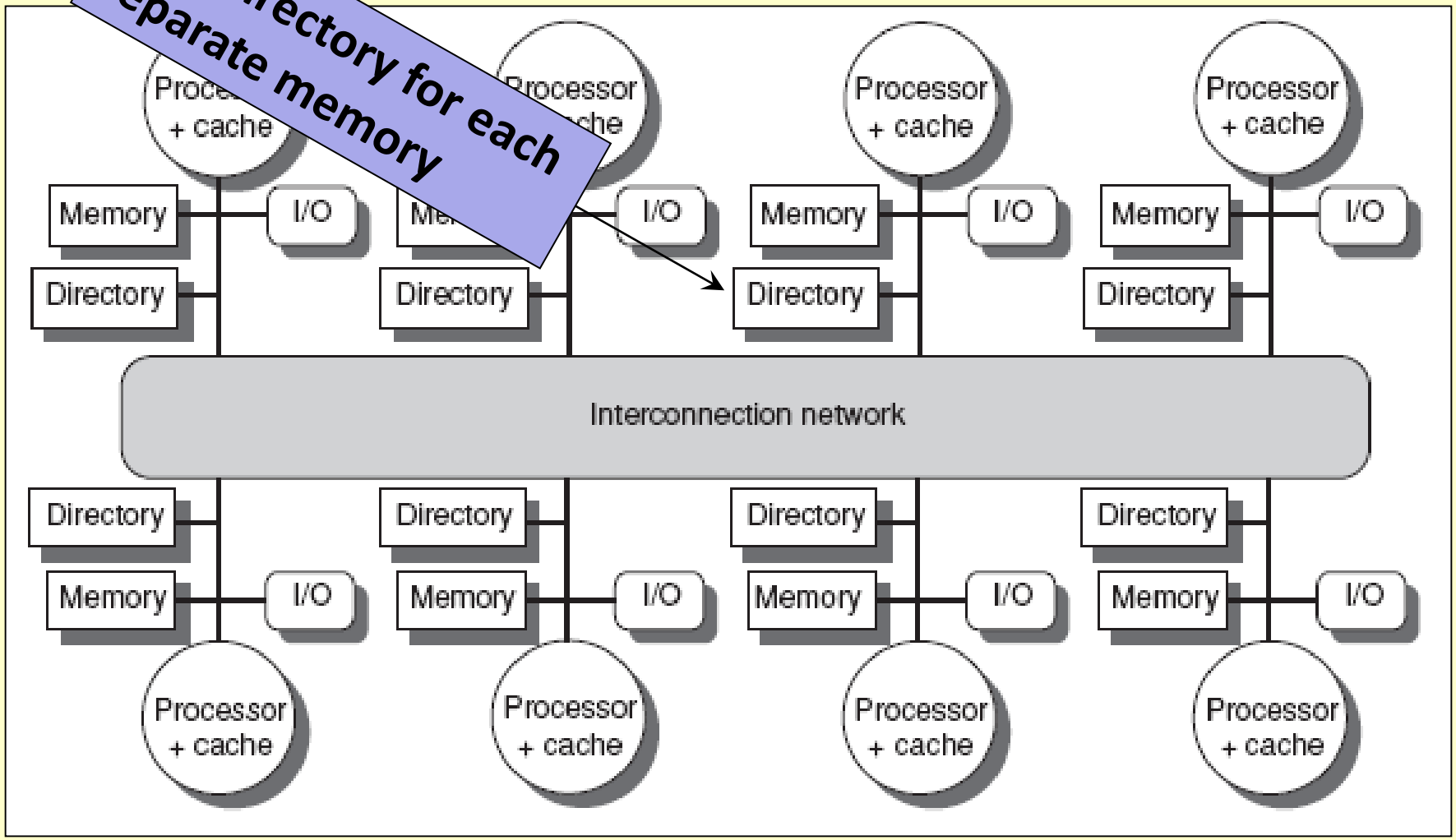
- **LL + SC can operate locally in L1 cache if no contention**

Questions?

Scalable Approach:– Directories

- ***Every* memory block has associated directory information**
  - I.e., a hardware data structure
  - Tracks copies of cached blocks and their states
  - Looks up directory entry on miss, communicates only with the nodes that have copies as needed
  - In scalable networks, communication with directory and copies is via network transactions
- **Many alternatives for organizing directory information**

Directory-based Approach

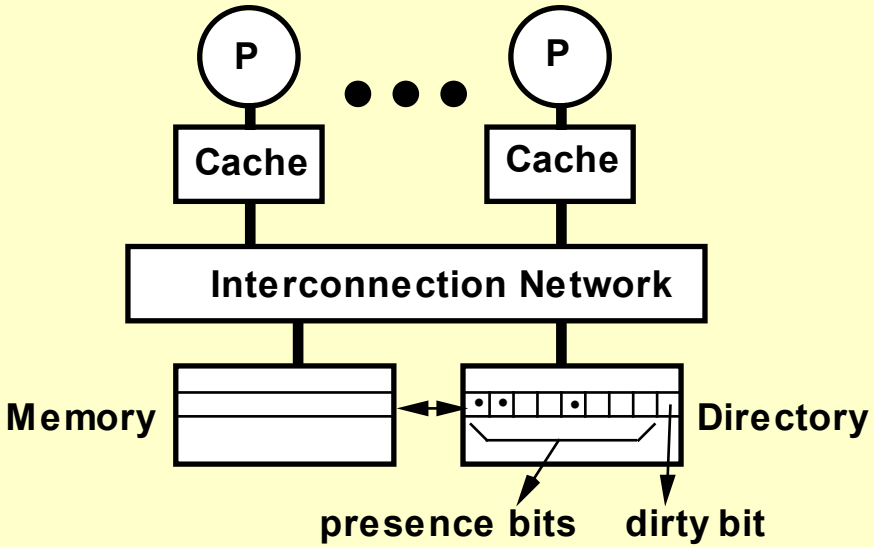


<figure 5.20 p. 380>

Advantages

- **Cache misses go only to directory**
  - Not to all processors/caches
- **Directory "knows" who has what blocks**
  - Can keep status up-to-date
- **Much better use of bandwidth of interconnection network**
  - Multiple paths
  - Point-to-point networks
  - Etc.

Basic Operation of Directory



- **k processors.**
- **With each cache-block in memory: k presence-bits, 1 dirty-bit**
- **With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit**

- **Read from main memory by processor *i*:**
  - If dirty-bit OFF then { read from main memory; turn *p*[*i*] ON; }
  - if dirty-bit ON then { recall line from *owner*; update memory; turn dirty-bit OFF; turn *p*[*i*] ON; supply recalled data to *i*;}
- **Write to main memory by processor *i*:**
  - If dirty-bit OFF then { supply data to *i*; send *invalidations* to all caches that have the block; turn dirty-bit ON; turn *p*[*i*] ON; ... }

Directory Protocol

- **Similar to Snoopy Protocol: Three states**
  - *Uncached* (no processor has it; not valid in any cache)
  - *Shared*: ≥ 1 processors have data, memory up-to-date
  - *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; bit = 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

Note

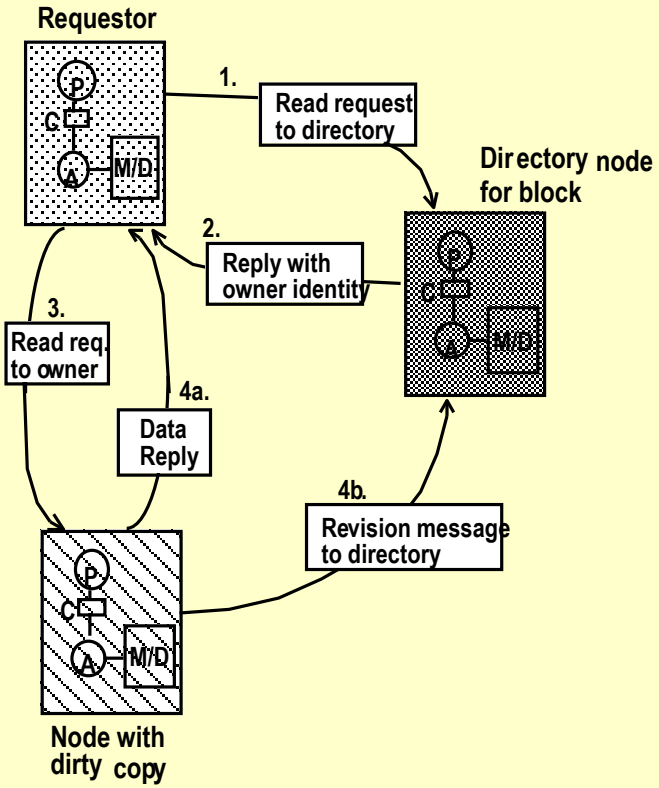
- **Each directory must be able to arbitrate among simultaneous, competing requests**
- **Requires arbitration circuitry in directory interface**
  - Non-trivial

Implementing a Directory

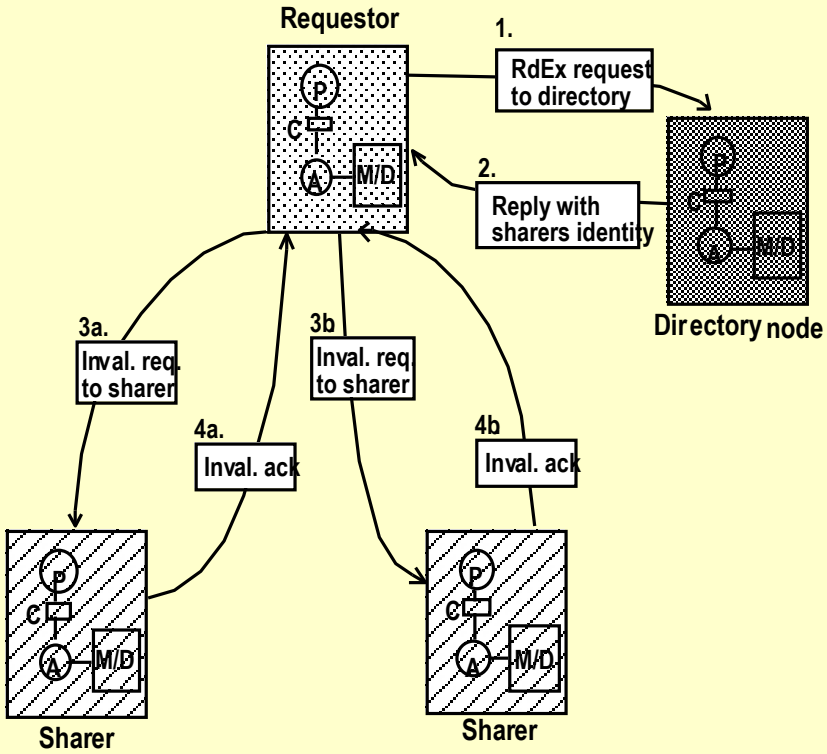
- **Would like to assume operations atomic**
- **… but they are not;**
  - reality is much harder;
  - must avoid deadlock when running out of buffers in network (see Appendix)

- **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

Basic Directory Transactions



(a) Read miss to a block in dirty state



(b) Write miss to a block with two sharers

See also Fig 5.22, 5.23

Questions?

Transactional Memory

- Herlihy & Moss, 1993
- Appeared in Intel & AMD processes since textbook
- More general than SC, LL

Transactional Memory (continued)

- **Like atomic transactions**
  - From database & distributed systems
- ***A C I D***
  - *Atomicity* – to outside world, transaction happens indivisibly
  - *Consistency* – transaction preserves system invariants
  - *Isolated* – transactions do not interfere with each other
  - *Durable* – once a transaction "commits," the changes are permanent

Transactional Memory

- **AMD announcement of ASF**
  - 2006
  - Not known in AMD products
- **IBM BlueGene/Q**
  - 2011
- **TSX instructions in Haswell**
  - Implementation details not published

More info to come?

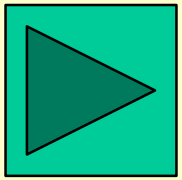
Implementing LL & SC

- **LL & SC can be implemented *entirely* in L1 cache!**
  1. LL leaves cache block containing lock in "read-shared" state, *or*
  2. … in write-exclusive state (if already in that state)

- **Any *change* to cache block by another processor or core causes state to change to *invalid* …**
- **… thereby invalidating the LL operation**

- **Likewise, any write to same address by another thread in *same* core …**
- **… can be detected by L1 cache to invalidate the LL operation**

Questions?



CS-4515
Computer Architecture

Professor Hugh C. Lauer
Professor Thérèse M. Smith

CS-4515, System Programming Concepts