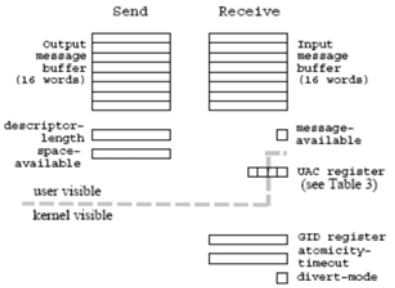
CS252
Graduate Computer Architecture
Lecture 18
April 2nd, 2012

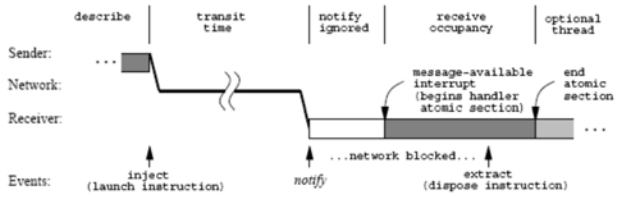
Memory Consistency Models and Snoopy Bus Protocols

Prof John D. Kubiatowicz
http://www.cs.berkeley.edu/~kubitron/cs252

Review: Alewife Messaging

- Send message
  - write words to special network interface registers
  - Execute atomic launch instruction
- Receive
  - Generate interrupt/launch user-leve thread context
  - Examine message by reading from special network interface registers
  - Execute dispose message
  - Exit atomic section



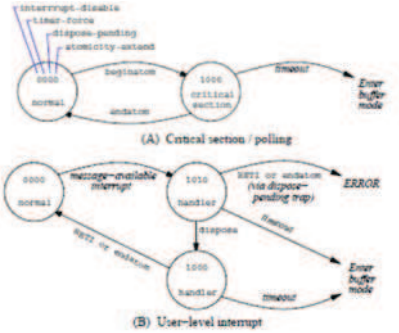


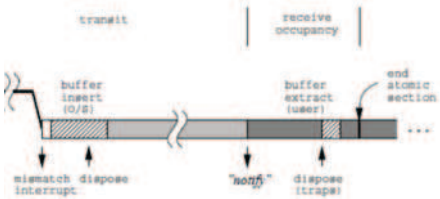
Review: Sharing of Network Interface

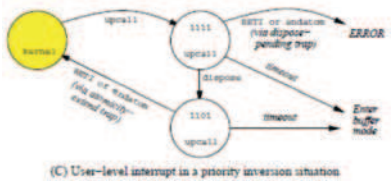
- What if user in middle of constructing message and must context switch???
  - Need *Atomic Send operation!*
    - » Message either completely in network or not at all
    - » Can save/restore user's work if necessary (think about single set of network interface registers
  - J-Machine mistake: after start sending message must let sender finish
    - » Flits start entering network with first *SEND* instruction
    - » Only a SENDE instruction constructs tail of message
- Receive Atomicity
  - If want to allow user-level interrupts or polling, must give user control over network reception
    - » Closer user is to network, easier it is for him/her to screw it up: Refuse to empty network, etc
    - » However, must allow *atomicity:* way for good user to select when their message handlers get interrupted
  - Polling: ultimate receive atomicity – never interrupted
    - » Fine as long as user keeps absorbing messages

Review: User-level event mechanism

- Disable during polling:
  - Allowed as long as user code properly removing messages
- Disable as atomicity for user-level interrupt
  - Allowed as long as user removes message quickly
- Emulation of hardware delivery in software:

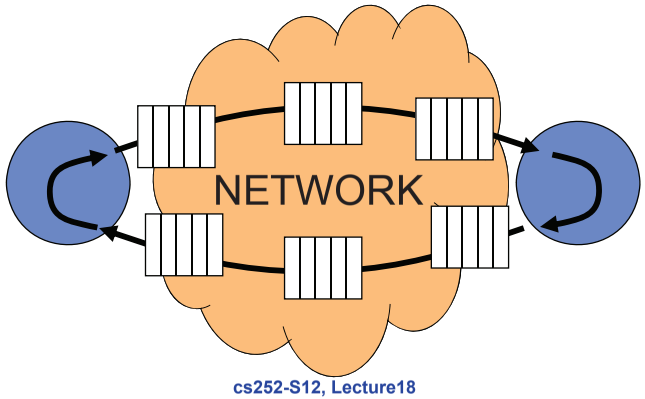






The Fetch Deadlock Problem

- Even if a node cannot issue a request, it must sink network transactions!
  - Incoming transaction may be request ⇒ generate a response.
  - Closed system (finite buffering)
- Deadlock occurs even if network deadlock free!

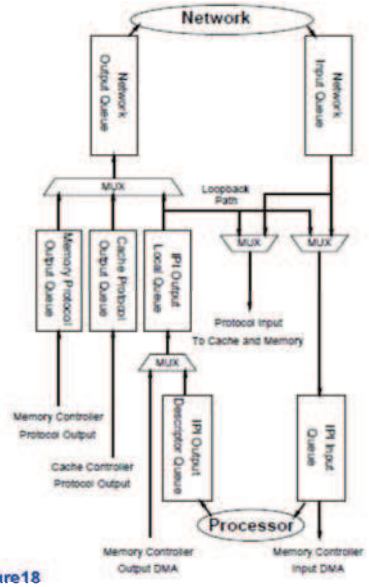


Review: Solutions to Fetch Deadlock?

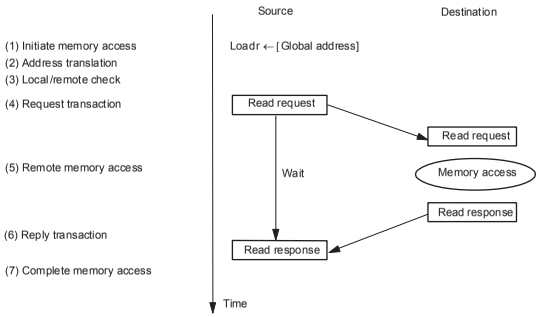
- logically independent request/reply networks
  - physical networks
  - virtual channels with separate input/output queues
- bound requests and reserve input buffer space
  - K(P-1) requests + K responses per node
  - service discipline to avoid fetch deadlock?
- NACK on input buffer full
  - NACK delivery?
- Alewife Solution:
  - Dynamically increase buffer space to memory when necessary
  - Argument: this is an uncommon case, so use software to fix

Example Queue Topology: Alewife

- Message-Passing and Shared-Memory both need messages
  - Thus, can provide both!
- When deadlock detected, start storing messages to memory (out of hardware)
  - Remove deadlock by increasing available queue space
- When network starts flowing again, relaunch queued messages
  - They take loopback path to be handled by local hardware

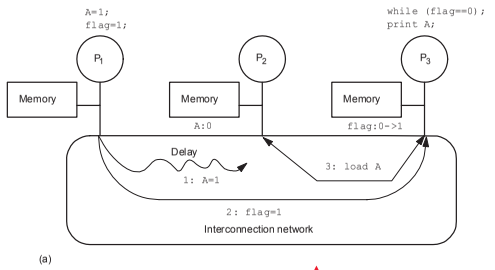


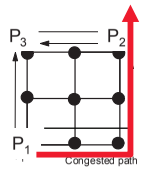
Shared Address Space Abstraction



- Fundamentally a two-way request/response protocol
  - writes have an acknowledgement
- Issues
  - fixed or variable length (bulk) transfers
  - remote virtual or physical address, where is action performed?
  - deadlock avoidance and input buffer full
- coherent? consistent?

Example of need for control of ordering





- "Natural ordering" violated even without caching!
  - No way to enforce serialization
- Solution? Acknowledge write of A before writing Flag…

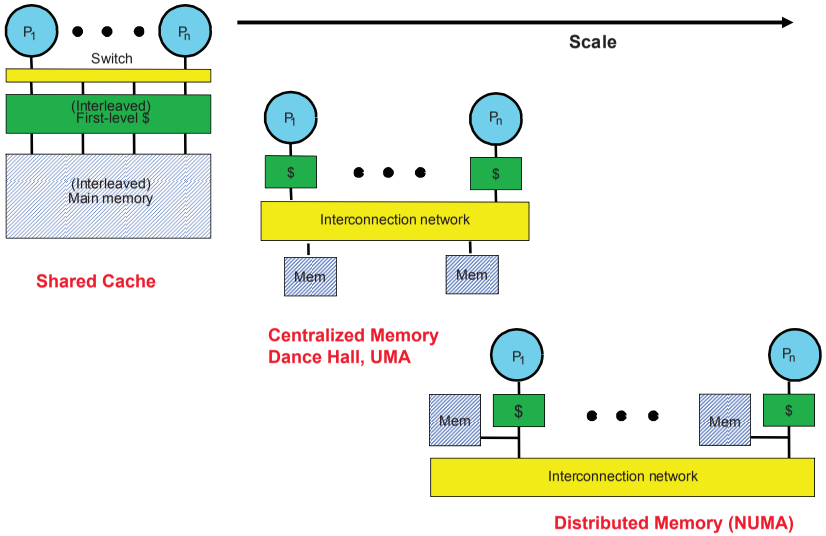
Properties of Shared Address Abstraction

- Source and destination data addresses are specified by the source of the request
  - a degree of logical coupling and trust
- no storage logically "outside the address space"
  - may employ temporary buffers for transport
- Operations are fundamentally request/response
- Remote operation can be performed on remote memory
  - logically does not require intervention of the remote processor

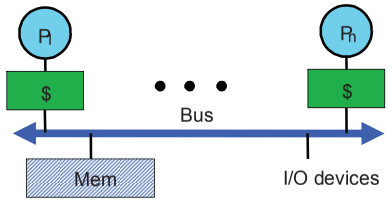
Administrative

- Midterm I: Still grading:
  - I hope to have exams graded soon
  - Sorry about this – was traveling last week
  - Will try to get solutions posted soon(er) as well
- Should be working full blast on project by now!
  - I'm going to want you to submit a (written) update on Wednesday
  - We will meet shortly after that

Natural Extensions of Memory System



Bus-Based Symmetric Shared Memory



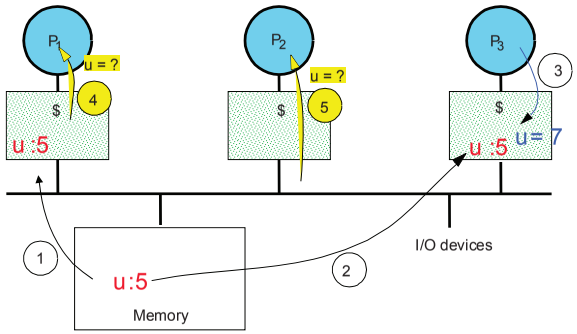
- Still an important architecture – even on chip (until very recently)
  - Building blocks for larger systems; arriving to desktop
- Attractive as throughput servers and for parallel programs
  - Fine-grain resource sharing
  - Uniform access via loads/stores
  - Automatic data movement and coherent replication in caches
  - Cheap and powerful extension
- Normal uniprocessor mechanisms to access data
  - Key is extension of memory hierarchy to support multiple processors

Caches and Cache Coherence

- Caches play key role in all cases
  - Reduce average data access time
  - Reduce bandwidth demands placed on shared interconnect
- private processor caches create a problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
    - » They'll keep accessing stale value in their caches

  ⇒ *Cache coherence* problem
- What do we do about it?
  - Organize the mem hierarchy to make it go away
  - Detect and take actions to eliminate the problem

Example Cache Coherence Problem



Things to note:

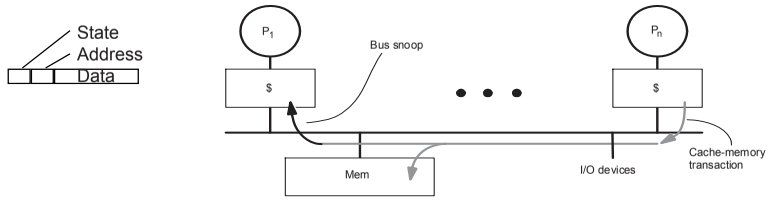
Processors see different values for u after event 3

With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when

Processes accessing main memory may see very stale value

Unacceptable to programs, and frequent!

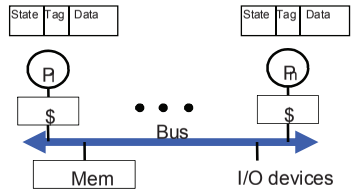
Snoopy Cache-Coherence Protocols

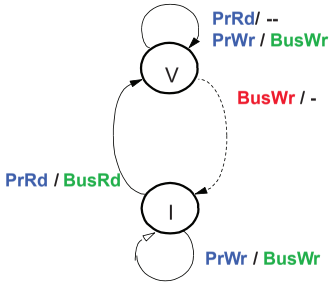


- Works because bus is a broadcast medium & Caches know what they have
- Cache Controller "snoops" all transactions on the shared bus
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - » invalidate, update, or supply value
  - depends on state of the block and the protocol

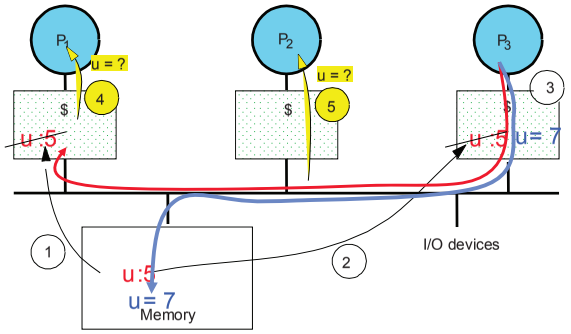
Write-through Invalidate Protocol

- Basic Bus-Based Protocol
  - Each processor has cache, state
  - All transactions over bus snooped
- Writes invalidate all other caches
  - can have multiple simultaneous readers of block,but write invalidates them
- Two states per block in each cache
  - as in uniprocessor
  - state of a block is a *p*-vector of states
  - Hardware state bits associated with blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache





Example: Write-thru Invalidate



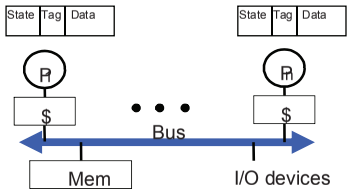
Write-through vs. Write-back

- Write-through protocol is simple
  - every write is observable
- Every write goes on the bus
  - ⇒ Only one write can take place at a time in any processor
- Uses a lot of bandwidth!

Example: 200 MHz dual issue, CPI = 1,
15% stores of 8 bytes

- 30 M stores per second per processor
- 240 MB/s per processor

1GB/s bus can support only about 4 processors without saturating



Invalidate vs. Update

- Basic question of program behavior:
  - Is a block written by one processor later read by others before it is overwritten?
- Invalidate.
  - yes: readers will take a miss
  - no: multiple writes without additional traffic
    - » also clears out copies that will never be used again
- Update.
  - yes: avoids misses on later references
  - no: multiple useless updates
    - » even to nodes that have dropped value from cache!

⇒Need to look at program reference patterns and hardware complexity

⇒Can we tune this automatically????

but first - correctness

Coherence?

- Caches are supposed to be transparent
- What would happen if there were no caches
- Every memory operation would go "to the memory location"
  - may have multiple memory banks
  - all operations on a particular location would be serialized
    - » all would see THE order
- Interleaving among accesses from different processors
  - within individual processor => program order
  - across processors => only constrained by explicit synchronization
- Processor only observes state of memory system by issuing memory operations!

Definitions

- Memory operation
  - load, store, read-modify-write
- Issues
  - leaves processor's internal environment and is presented to the memory subsystem (caches, buffers, busses,dram, etc)
- Performed with respect to a processor
  - write: subsequent reads return the value
  - read: subsequent writes cannot affect the value
- Coherent Memory System
  - there exists a serial order of mem operations on each location s.t.
    - » operations issued by a process appear in order issued
    - » value returned by each read is that written by previous write in the serial order

=> write propagation + write serialization

Is 2-state Protocol Coherent?

- Assume bus transactions and memory operations are atomic, one-level cache
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory op to complete before issuing next
  - with one-level cache, assume invalidations applied during bus xaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)

  ⇒ invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order

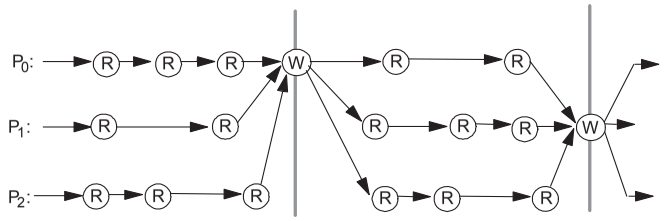
Ordering Reads

- Read misses
  - appear on bus, and will "see" last write in bus order
- Read hits: do not appear on bus
  - But value read was placed in cache by either
    - » most recent write by this processor, or
    - » most recent read miss by this processor
  - Both these transactions appeared on the bus
  - So reads hits also see values as produced bus order

Determining Orders More Generally

- Define a partial ordering on all memory operations ("Happens Before")
  - Written as: M1→M2
  - Loosely equivalent to "time"
- On single processor, M1→M2 from program order:
  - Crucial assumption: processor doesn't reorder operations!
- write W → read R if
  - read generates bus xaction that follows that for W.
- read or write M → write W if
  - M generates bus xaction and the xaction for W follows that for M.
- read R → write W if
  - read R does not generate a bus xaction and
  - is not already separated from write W by another bus xaction.

Ordering



- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order

Setup for Mem. Consistency

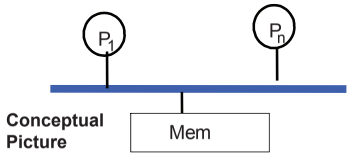
- Coherence ⇒ Writes to a location become visible to all in the same order
- But when does a write become visible?

- How do we establish orders between a write and a read by different procs?
  - use event synchronization
- Typically use more than one location!

Example

P_1	P_2
/*Assume initial value of A and ag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location



Another Example of Ordering?

P_1	P_2
/*Assume initial values of A and B are 0 */	
(1a) A = 1;	(2a) print B;
(1b) B = 2;	(2b) print A;

- What's the intuition?
  - Whatever it is, we need an ordering model for clear semantics
    - » across different locations as well
    - » so programmers can reason about what results are possible
  - This is the memory consistency model

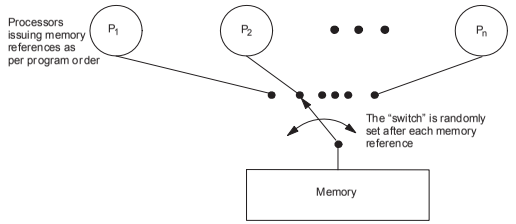
Memory Consistency Model

- Specifies constraints on the order in which memory operations (from any process) can appear to execute with respect to one another
  - What orders are preserved?
  - Given a load, constrains the possible values returned by it
- Without it, can't tell much about an SAS program's execution
- Implications for both programmer and system designer
  - Programmer uses to reason about correctness and possible results
  - System designer can use to constrain how much accesses can be reordered by compiler or hardware
- Contract between programmer and system

Sequential Consistency

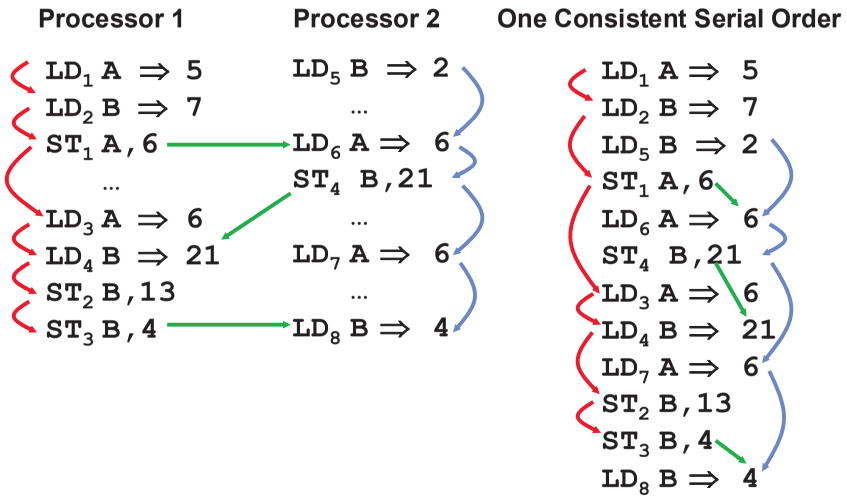
- Memory operations from a proc become visible (to itself and others) in program order
- There exists a total order, consistent with this partial order - i.e., an interleaving
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors
- Said another way:
  - For any possible individual run of a program on multiple processors
  - Should be able to come up with a serial interleaving of all operations that respects
    - » Program Order
    - » Read-after-write orderings (locally and through network)
    - » Also Write-after-read, write-after-write

Sequential Consistency

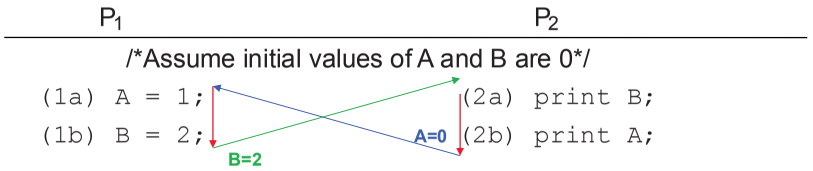


- Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory
- "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

Sequential Consistency Example



SC Example

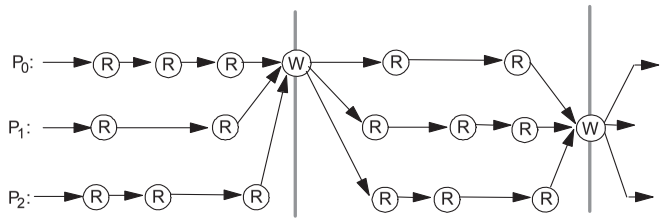


- What matters is order in which operations *appear to execute*, not the chronological order of events
- Possible outcomes for (A,B): (0,0), (1,0), (1,2)
- What about (0,2) ?
  - program order ⇒ 1a->1b and 2a->2b
  - A = 0 implies 2b->1a, which implies 2a->1b
  - B = 2 implies 1b->2a, which leads to a contradiction (cycle!)
- Since there is a cycle⇒no sequential order that is consistent!

Implementing SC

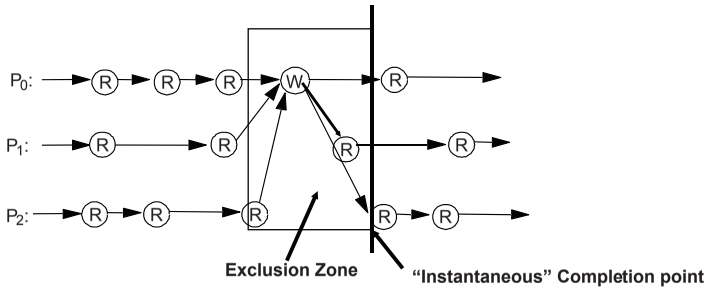
- Two kinds of requirements
  - Program order
    - » memory operations issued by a process must appear to execute (become visible to others and itself) in program order
  - Atomicity
    - » in the overall hypothetical total order, one memory operation should appear to complete with respect to all processes before the next one is issued
    - » guarantees that total order is consistent across processes
  - tricky part is making writes atomic
- How can compilers violate SC?
  - Architectural enhancements?

Happens Before: arrows are time



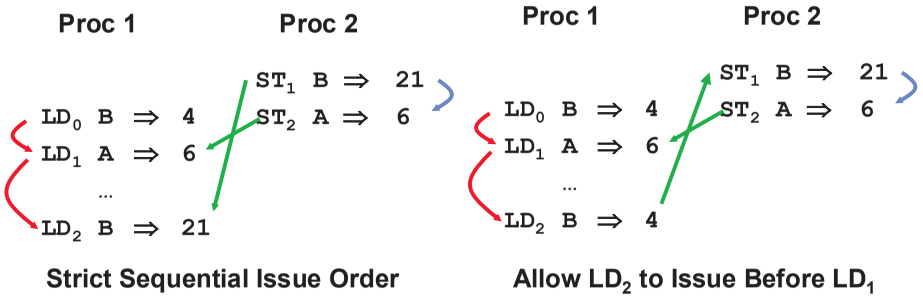
- Tricky part is relationship between nodes with respect to single location
  - Program order adds relationship between locations
- Easy topological sort comes up with sequential ordering assuming:
  - All happens-before relationships are time
  - Then – can't have time cycles (at least not inside classical machine in normal spacetime ☺).
- Unfortunately, writes are not instantaneous
  - What do we do?

Ordering: Scheurich and Dubois



- Sufficient Conditions
  - every process issues mem operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing next memory operation
  - after a read is issued, the issuing process waits for the read to complete and for the write whose value is being returned to complete (gloabaly) before issuing its next operation

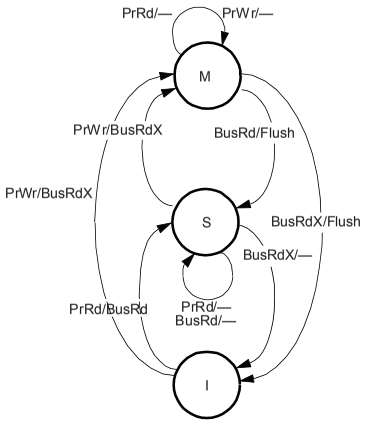
What about reordering of accesses?



- Can LD_2 issue before LD_1?
  - Danger of getting CYCLE! (i.e. not sequentially consistent)
- What can we do?
  - Go ahead and issue ld early, but watch cache
  - If value invalidated from cache early:
    - » Must squash LD_2 and any instructions that have used its value
- Reordering of Stores
  - Must be even more careful

MSI Invalidate Protocol

- Three States:
  - "M": "Modified"
  - "S": "Shared"
  - "I": "Invalid"
- Read obtains block in "shared"
  - even if only cache copy
- Obtain exclusive ownership before writing
  - BusRdx causes others to invalidate (demote)
  - If M in another cache, will flush
  - BusRdx even if hit in S
    - » promote to M (upgrade)
- What about replacement?
  - S->I, M->I as before



Write Serialization for Coherence

- Correctness
  - When is write miss performed?
    - » How does writer "observe" write?
    - » How is it "made visible" to others?
    - » How do they "observe" the write?
  - When is write hit made visible to others?
  - When does a write hit complete globally?
- Writes that appear on the bus (BusRdX) are ordered by bus
  - performed in writer's cache before other transactions, so ordered same w.r.t. all processors (incl. writer)
  - Read misses also ordered wrt these
- Write that don't appear on the bus:
  - P issues BusRdX B.
  - further mem operations on B until next transaction are from P
    - » read and write hits
    - » these are in program order
  - for read or write from another processor
    - » separated by intervening bus transaction
- Reads hits?

Sequential Consistency

- Bus imposes total order on xactions for all locations
- Between xactions, procs perform reads/writes (locally) in program order
- So any execution defines a natural partial order
  - M_j subsequent to M_i if
    - » (i) M_j follows M_i in program order on same processor,
    - » (ii) M_j generates bus xaction that follows the memory operation for M_i
- In segment between two bus transactions, any interleaving of local program orders leads to consistent total order
- Within segment writes observed by proc P serialized as:
  - Writes from other processors by the previous bus xaction P issued
  - Writes from P by program order
  - Insight: only one cache may have value in "M" state at a time…

Sufficient conditions

- Sufficient Conditions
  - issued in program order
  - after write issues, the issuing process waits for the write to complete before issuing next memory operation
  - after read is issues, the issuing process waits for the read to complete and for the write whose value is being returned to complete (globally) before issuing its next operation
- Write completion
  - can detect when write appears on bus (flush) appears
- Write atomicity:
  - if a read returns the value of a write, that write has become visible to all others already
    - » Either: it is being read by the processor that wrote it and no other processor has a copy (thus any read by any other processor will get new value via a flush
    - » Or: it has already been flushed back to memory and all processors will have the value

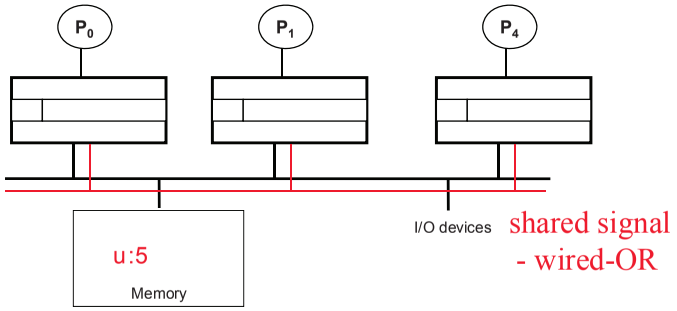
Lower-level Protocol Choices

- BusRd observed in M state: what transition to make?
  - M ----> I
  - M ----> S
  - Depends on expectations of access patterns
- Problem: How does memory know whether or not to supply data on BusRd?
  - Must abort memory transaction somehow – or make sure that the cache with a dirty line gets to respond first
- Problem: Read/Write is 2 bus xactions, even if no sharing
  - » BusRd (I->S) followed by BusRdX or BusUpgr (S->M)
  - » What happens on sequential programs?

MESI (4-state) Invalidation Protocol

- Four States:
  - "M": "Modified"
  - "E": "Exclusive"
  - "S": "Shared"
  - "I": "Invalid"
- Add *exclusive* state
  - distinguish exclusive (writable) and owned (written)
  - Main memory is up to date, so cache not necessarily owner
  - can be written locally
- States
  - invalid
  - exclusive or *exclusive-clean* (only this cache has copy, but not modified)
  - shared (two or more caches may have copies)
  - modified (dirty)
- I -> E on PrRd if no cache has copy
  - => How can you tell?

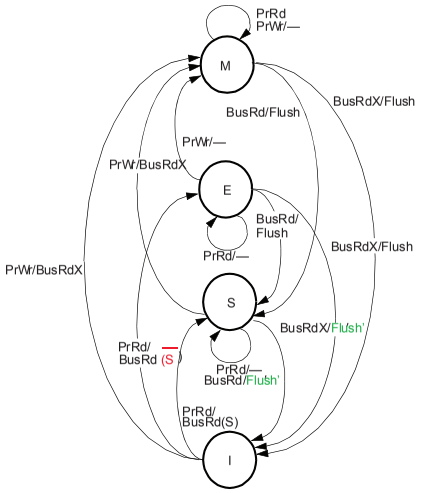
Hardware Support for MESI



- All cache controllers snoop on BusRd
- Assert 'shared' if present (S? E? M?)
- Issuer chooses between S and E
  - how does it know when all have voted?

MESI State Transition Diagram

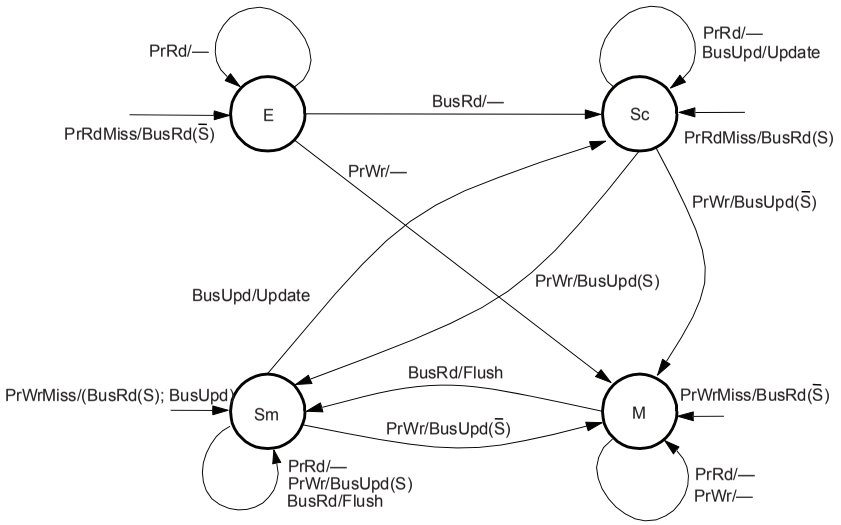
- BusRd(S) means shared line asserted on BusRd transaction
- Flush': if cache-to-cache xfers
  - only one cache flushes data
- Replacement:
  - S→I can happen without telling other caches
  - E→I, M→I
- MOESI protocol: Owned state: exclusive but memory not valid



Dragon Write-back Update Protocol

- 4 states
  - Exclusive-clean or exclusive (E): I and memory have it
  - Shared clean (Sc): I, others, and maybe memory, but I'm not owner
  - Shared modified (Sm): I and others but not memory, and I'm the owner
    - » Sm and Sc can coexist in different caches, with only one Sm
  - Modified or dirty (D): I and, noone else
- No invalid state
  - If in cache, cannot be invalid
  - If not present in cache, view as being in not-present or invalid state
- New processor events: PrRdMiss, PrWrMiss
  - Introduced to specify actions when block not present in cache
- New bus transaction: BusUpd
  - Broadcasts single word written on bus; updates other relevant caches

Dragon State Transition Diagram



Summary

- Shared-memory machine
  - All communication is implicit, through loads and stores
  - Parallelism introduces a bunch of overheads over uniprocessor
- Cache Coherence Problem
  - Local Caches ⇒ Copies of data ⇒ Potential inconsistencies
- Memory Coherence:
  - Writes to a given location eventually propagated
  - Writes to a given location seen in same order by everyone
- Memory Consistency:
  - Constraints on ordering between processors and locations
- Sequential Consistency:
  - For every parallel execution, there exists a serial interleaving
- Snoopy Bus Protocols
  - Make use of broadcast to ensure coherence
  - Various tradeoffs:
    - » Write Through vs Write Back
    - » Invalidate vs Update