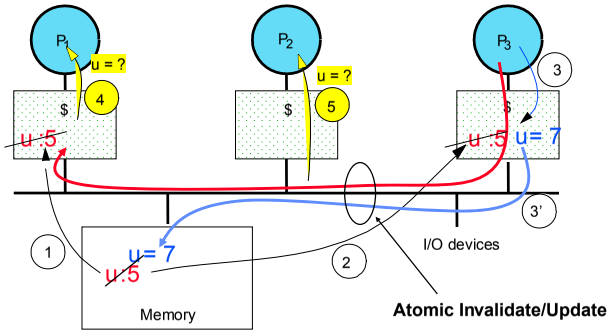
CS 258
Parallel Computer Architecture
Lecture 14

Shared Memory Multiprocessors

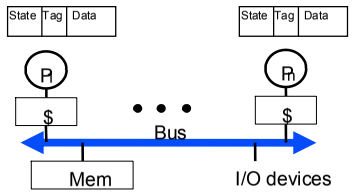
March 12, 2008
Prof John D. Kubiatowicz
http://www.cs.berkeley.edu/~kubitron/cs258

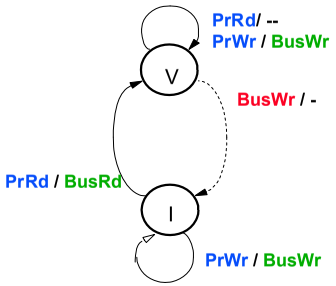
Recall: Write-thru Invalidate



Recall: 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





Recall: 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

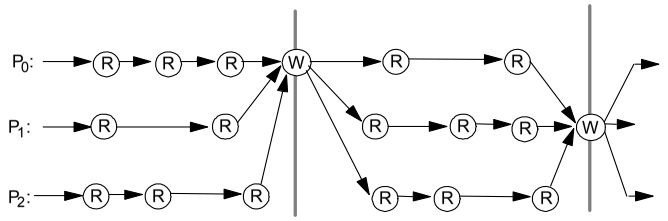
  => write propagation + write serialization

Recall: 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 operation 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
- 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

Ordering Within single Processor

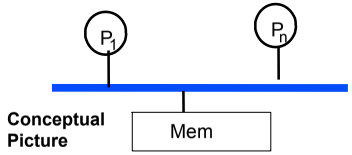


- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order
- Writes to bus represent Instantaneous points in time ⇒ *graph is acylic ⇒ serial order exists*

Recall: Multiple 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



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 a single address space (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

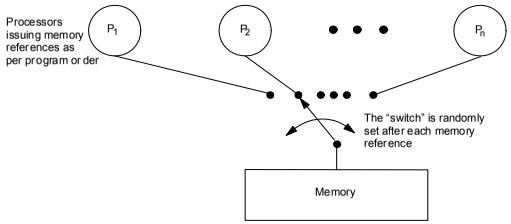
What Really is Program Order?

- Intuitively, order in which operations appear in source code
  - Straightforward translation of source code to assembly
  - At most one memory operation per instruction
- But not the same as order presented to hardware by compiler
- So which is program order?
  - Depends on which layer, and who's doing the reasoning
  - *We assume order as seen by programmer*

Sequential Consistency

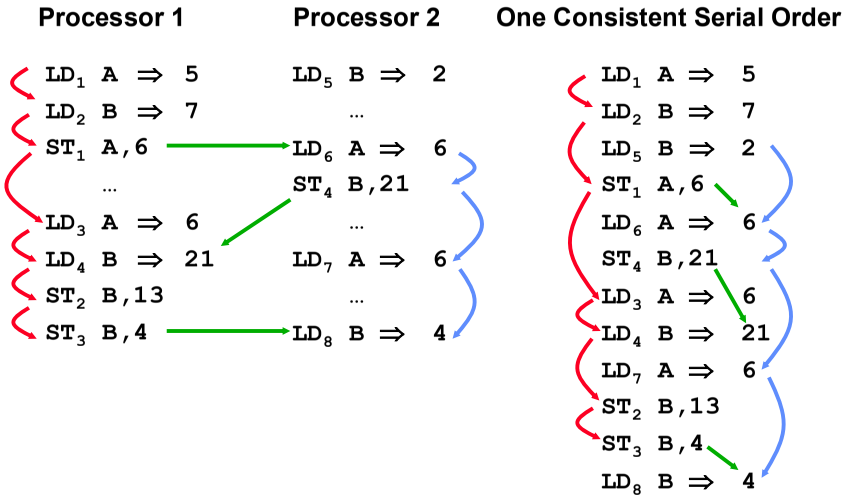
- Memory operations from a proc become visible (to itself and others) in program order
- There exists a total order, consistent with this partial order - i.e., an interleaving
  - the position at which a write occurs in the hypothetical total order should be the same with respect to all processors
- 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



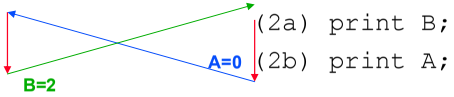
- ∃ Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - as if there were no caches, and a single memory
- "A multiprocessor is sequentially 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

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;

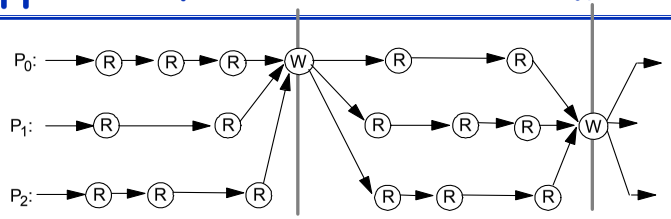


- What matters is order in which operations *appear to execute*, not the chronological order of events
- Possible outcomes for (A,B): (0,0), (1,0), (1,2)
- What about (0,2) ?
  - program order ⇒ 1a->1b and 2a->2b
  - A = 0 implies 2b->1a, which implies 2a->1b
  - B = 2 implies 1b->2a, which leads to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

Implementing SC

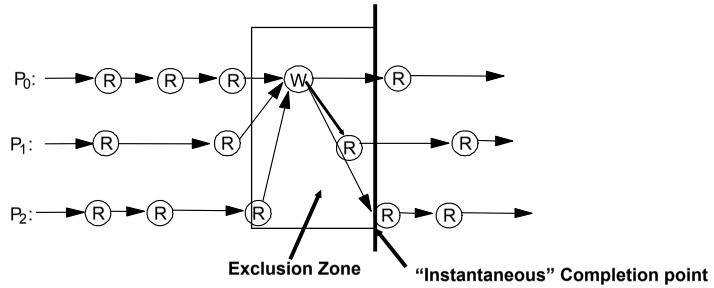
- Two kinds of requirements
  - Program order
    - » memory operations issued by a process must appear to execute (become visible to others and itself) in program order
  - Atomicity
    - » in the overall hypothetical total order, one memory operation should appear to complete with respect to all processes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

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?

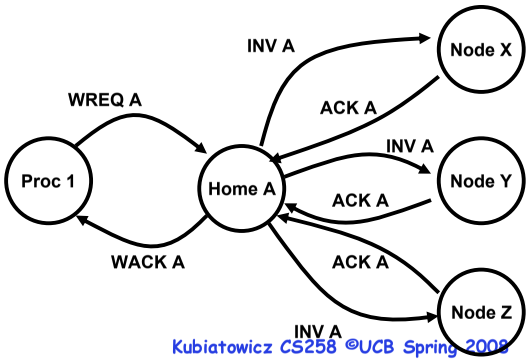
Ordering: Scheurich and Dubois



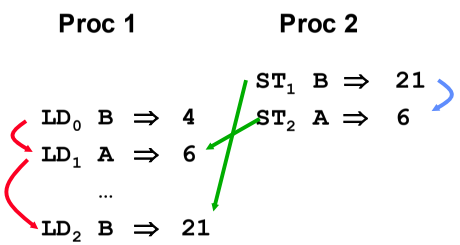
- Sufficient Conditions for Sequential Consistency
  - every process issues mem operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing next memory operation
  - after a read is issued, the issuing process waits for the read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

Implications for Implementation

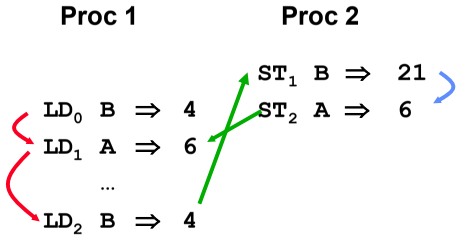
- Serious restrictions on when you can issue requests
- Processor held up on store until:
  - Every processor cache invalidated!
  - On bus, may not be big deal, otherwise, must send INV and wait for ACK
- Consider Distributed Protocol (more later):



What about reordering of accesses?



Strict Sequential Issue Order



Allow LD_2 to Issue Before LD_1

- 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

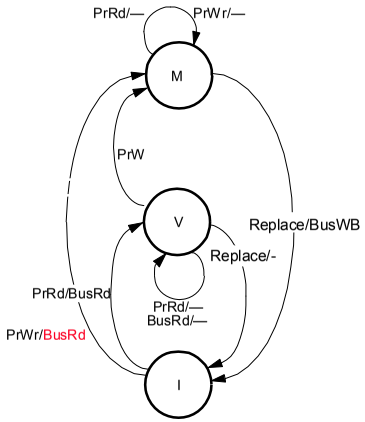
Write-back Caches (Uniprocessor)

- 2 processor operations
  - PrRd, PrWr
- 3 states
  - invalid, valid (clean), modified (dirty)
  - ownership: who supplies block
- 2 bus transactions:
  - read (BusRd), write-back (BusWB)
  - only cache-block transfers

⇒ treat Valid as "shared" and Modified as "exclusive"

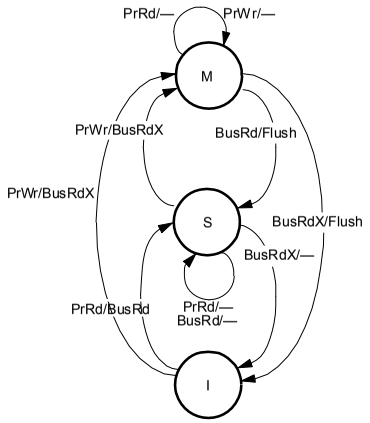
⇒ introduce one new bus transaction

  - read-exclusive: read for purpose of modifying (read-to-own)

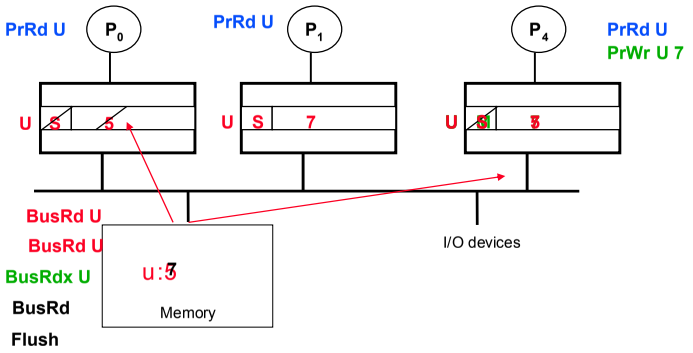


MSI Invalidate Protocol

- 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



Example: Write-Back Protocol



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?

Write Serialization for Coherence

- 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 bus 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) follows 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
- w/i 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

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
- Write atomicity:
  - if a read returns the value of a write, that write has already become visible to all others already

Summary

- Shared-memory machine
  - All communication is implicit, through loads and stores
  - Parallelism introduces a bunch of overheads over uniprocessor
- 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