Parallel Computers

- Definition: “A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”

  Almasi and Gottlieb, *Highly Parallel Computing* ,1989

- Questions about parallel computers:
  - How large a collection?
  - How powerful are processing elements?
  - How do they cooperate and communicate?
  - How are data transmitted?
  - What type of interconnection?
  - What are HW and SW primitives for programmer?
  - Does it translate into performance?

Parallel Processors "Religion"

- The dream of computer architects since 1950s: replicate processors to add performance vs. design a faster processor
- Led to innovative organizations tied to particular programming models since "uniprocessors can't keep going"
  - e.g., uniprocessors must stop getting faster due to limit of speed of light: 1972, … , 1989
  - Borders religious fervor: you must believe!
  - Fervor damped some when 1990s companies went out of business: Thinking Machines, Kendall Square, ...
- Argument instead is the "pull" of opportunity of scalable performance, not the "push" of uniprocessor performance plateau?

What level Parallelism?

- Bit level parallelism: 1970 to ~1985
  - 4 bits, 8 bit, 16 bit, 32 bit microprocessors
- Instruction level parallelism (ILP): ~1985 through today
  - Pipelining
  - Superscalar
  - VLIW
  - Out-of-Order execution
  - Limits to benefits of ILP?
- Process Level or Thread level parallelism; mainstream for general purpose computing?
  - Servers are parallel
  - High-end Desktop dual processor PC soon??

Why Multiprocessors?

1. Microprocessors as the fastest CPUs
   - Collecting several much easier than redesigning 1
2. Complexity of current microprocessors
   - Do we have enough ideas to sustain 1.5X/yr?
   - Can we deliver such complexity on schedule?
3. Slow (but steady) improvement in parallel software (scientific apps, databases, OS)
4. Emergence of embedded and server markets driving microprocessors in addition to desktops
   - Embedded functional parallelism, producer/consumer model
   - Server figure of merit is tasks per hour vs. latency

Popular Flynn Categories

- SISD (Single Instruction Single Data)
  - Uniprocessors
- MISD (Multiple Instruction Single Data)
  - ???; multiple processors on a single data stream
- SIMD (Single Instruction Multiple Data)
  - Examples: Illiac-IV, CM-2
    - » Simple programming model
    - » Low overhead
    - » Flexibility
    - » All custom integrated circuits
  - (Phrase reused by Intel marketing for media instructions ~ vector)
- MIMD (Multiple Instruction Multiple Data)
  - Examples: Sun Enterprise 5000, Cray T3D, SGI Origin
    - » Flexible
    - » Use off-the-shelf micros
- MIMD current winner: Concentrate on major design emphasis <= 128 processor MIMD machines

Major MIMD Styles

1. Centralized shared memory ("Uniform Memory Access" time or "Shared Memory Processor")
2. Decentralized memory (memory module with CPU)
   - get more memory bandwidth, lower memory latency
   - Drawback: Longer communication latency
   - Drawback: Software model more complex

Decentralized Memory versions

1. Shared Memory with "Non Uniform Memory Access" time (NUMA)
2. Message passing "multicomputer" with separate address space per processor
   - Can invoke software with Remote Procedure Call (RPC)
   - Often via library, such as MPI: Message Passing Interface
   - Also called "Synchronous communication" since communication causes synchronization between 2 processes

Performance Metrics: Latency and Bandwidth

1. Communication Bandwidth
   - Need high bandwidth in communication
   - Match limits in network, memory, and processor
   - Challenge is link speed of network interface vs. bisection bandwidth of network
2. Communication Latency
   - Affects performance, since processor may have to wait
   - Affects ease of programming, since requires more thought to overlap communication and computation
   - Overhead to communicate is a problem in many machines
3. Communication Latency Hiding
   - How can a mechanism help hide latency?
   - Increases programming system burden
   - Examples: overlap message send with computation, prefetch data, switch to other tasks

Parallel Architecture

- Parallel Architecture extends traditional computer architecture with a communication architecture
  - abstractions (HW/SW interface)
  - organizational structure to realize abstraction efficiently

Parallel Framework

- Layers:
  - Programming Model:
    - Multiprogramming : lots of jobs, no communication
    - Shared address space: communicate via memory
    - Message passing: send and receive messages
    - Data Parallel: several agents operate on several data sets simultaneously and then exchange information globally and simultaneously (shared or message passing)
  - Communication Abstraction:
    - Shared address space: e.g., load, store, atomic swap
    - Message passing: e.g., send, receive library calls

Shared Address Model Summary

- Each processor can name every physical location in the machine
- Each process can name all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Uses virtual memory to map virtual to local or remote physical
- Memory hierarchy model applies: now communication moves data to local processor cache (as load moves data from memory to cache)
  - Latency, BW, scalability when communicate?

Shared Address/Memory Multiprocessor Model

- Communicate via Load and Store
  - Oldest and most popular model
- Based on timesharing: processes on multiple processors vs. sharing single processor
- process: a virtual address space and ~ 1 thread of control
  - Multiple processes can overlap (share), but ALL threads share a process address space
- Writes to shared address space by one thread are visible to reads of other threads
  - Usual model: share code, private stack, some shared heap, some private heap

SMP Interconnect

- Processors to Memory AND to I/O
- Bus based: all memory locations equal access time so SMP = “Symmetric MP”
  - Sharing limited BW as add processors, I/O

Message Passing Model

- Whole computers (CPU, memory, I/O devices) communicate as explicit I/O operations
  - Essentially NUMA but integrated at I/O devices vs. memory system
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
  - Usually send includes process tag
    and receive has rule on tag: match 1, match any
  - Synch: when send completes, when buffer free, when request accepted, receive wait for send
- Send+receive => memory-memory copy, where each each supplies local address,
  AND does pair wise synchronization!

Data Parallel Model

- Operations can be performed in parallel on each element of a large regular data structure, such as an array
- 1 Control Processor broadcast to many PEs
  - When computers were large, could amortize the control portion of many replicated PEs
- Condition flag per PE so that can skip
- Data distributed in each memory
- Early 1980s VLSI => SIMD rebirth: 32 1-bit PEs + memory on a chip was the PE
- Data parallel programming languages lay out data to processor

Data Parallel Model

- Vector processors have similar ISAs, but no data placement restriction
- SIMD led to Data Parallel Programming languages
- Advancing VLSI led to single chip FPUs and whole fast μProcs (SIMD less attractive)
- SIMD programming model led to <u>Single Program Multiple Data (SPMD)</u> model
  - All processors execute identical program
- Data parallel programming languages still useful, do communication all at once: "<u>Bulk Synchronous</u>" phases in which all communicate after a global barrier

Advantages shared-memory communication model

- Compatibility with SMP hardware
- Ease of programming when communication patterns are complex or vary dynamically during execution
- Ability to develop applications using familiar SMP model, attention only on performance critical accesses
- Lower communication overhead, better use of BW for small items, due to implicit communication and memory mapping to implement protection in hardware, rather than through I/O system
- HW-controlled caching to reduce remote communication by caching of all data, both shared and private.

Advantages message-passing communication model

- The hardware can be simpler (esp. vs. NUMA)
- Communication explicit => simpler to understand; in shared memory it can be hard to know when communicating and when not, and how costly it is
- Explicit communication focuses attention on costly aspect of parallel computation, sometimes leading to improved structure in multiprocessor program
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization
- Easier to use sender-initiated communication, which may have some advantages in performance

Communication Models

- Shared Memory
  - Processors communicate with shared address space
  - Easy on small-scale machines
  - Advantages:
    - » Model of choice for uniprocessors, small-scale MPs
    - » Ease of programming
    - » Lower latency
    - » Easier to use hardware controlled caching
- Message passing
  - Processors have private memories, communicate via messages
  - Advantages:
    - » Less hardware, easier to design
    - » Focuses attention on costly non-local operations
- Can support either SW model on either HW base

3 Parallel Applications

- Commercial Workload
- Multiprogramming and OS Workload
- Scientific/Technical Applications

Parallel App: Commercial Workload

- Online transaction processing workload (OLTP) (like TPC-B or -C)
- Decision support system (DSS) (like TPC-D)
- Web index search (Altavista)

Benchmark	% Time User Mode	% Time Kernel	% Time I/O time (CPU Idle)
OLTP	71%	18%	11%
DSS (range)	82-94%	3-5%	4-13%
DSS (avg)	87%	4%	9%
Altavista	> 98%	< 1%	<1%

Parallel App: Multiprogramming and OS

- 2 independent copies of the compile phase of the Andrew benchmark (parallel make 8 CPUs)
- 3 phases: compiling the benchmarks; installing the object files in a library; removing the object files (I/O little, lot, almost all)

	User	Kernel	Synch. wait	I/O (CPU Idle)
% instructions	27%	3%	1%	69%
% time	27%	7%	2%	64%

Parallel App: Scientific/Technical

- FFT Kernel: 1D complex number FFT
  - 2 matrix transpose phases => all-to-all communication
  - Sequential time for n data points: O(n log n)
  - Example is 1 million point data set
- LU Kernel: dense matrix factorization
  - Blocking helps cache miss rate, 16x16
  - Sequential time for nxn matrix: $O(n^3)$
  - Example is 512 x 512 matrix

Parallel App: Scientific/Technical

- Barnes App: Barnes-Hut n-body algorithm solving a problem in galaxy evolution
  - n-body algs rely on forces drop off with distance; if far enough away, can ignore (e.g., gravity is $1/d^2$)
  - Sequential time for n data points: O(n log n)
  - Example is 16,384 bodies
- Ocean App: Gauss-Seidel multigrid technique to solve a set of elliptical partial differential eq.s'
  - red-black Gauss-Seidel colors points in grid to consistently update points based on previous values of adjacent neighbors
  - Multigrid solve finite diff. eq. by iteration using hierarch. Grid
  - Communication when boundary accessed by adjacent subgrid
  - Sequential time for nxn grid: $O(n^2)$
  - Input: 130 x 130 grid points, 5 iterations

Parallel Scientific App: Scaling

- p is # processors
- n is + data size
- Computation scales up with n by O( ), scales down linearly as p is increased
- Communication
  - FFT all-to-all so n
  - LU, Ocean at boundary, so $n^{1/2}$
  - Barnes complex: $n^{1/2}$ greater distance, x log n to maintain bodies relationships
  - All scale down $1/p^{1/2}$

App	Scaling computation	Scaling communication	Scaling comp-to-comm
FFT	$n \log n / p$	n / p	$\log n$
LU	n / p	$n^{1/2} / p^{1/2}$	$n^{1/2} / p^{1/2}$
Barnes	$n \log n / p$	$n^{1/2} \log n / p^{1/2}$	$n^{1/2} / p^{1/2}$
Ocean	n / p	$n^{1/2} / p^{1/2}$	$n^{1/2} / p^{1/2}$

- **Keep n same, but inc. p?**
- **Inc. n to keep comm. same w. +p?**

Amdahl's Law and Parallel Computers

- Amdahl's Law (FracX: original % to be speeded up)
  Speedup = 1 / [(FracX/SpeedupX + (1-FracX)]
- A portion is sequential => limits parallel speedup
  - Speedup <= 1/ (1-FracX)
- Ex. What fraction should be sequential to get 80X speedup from 100 processors? Assume either 1 processor or 100 fully used

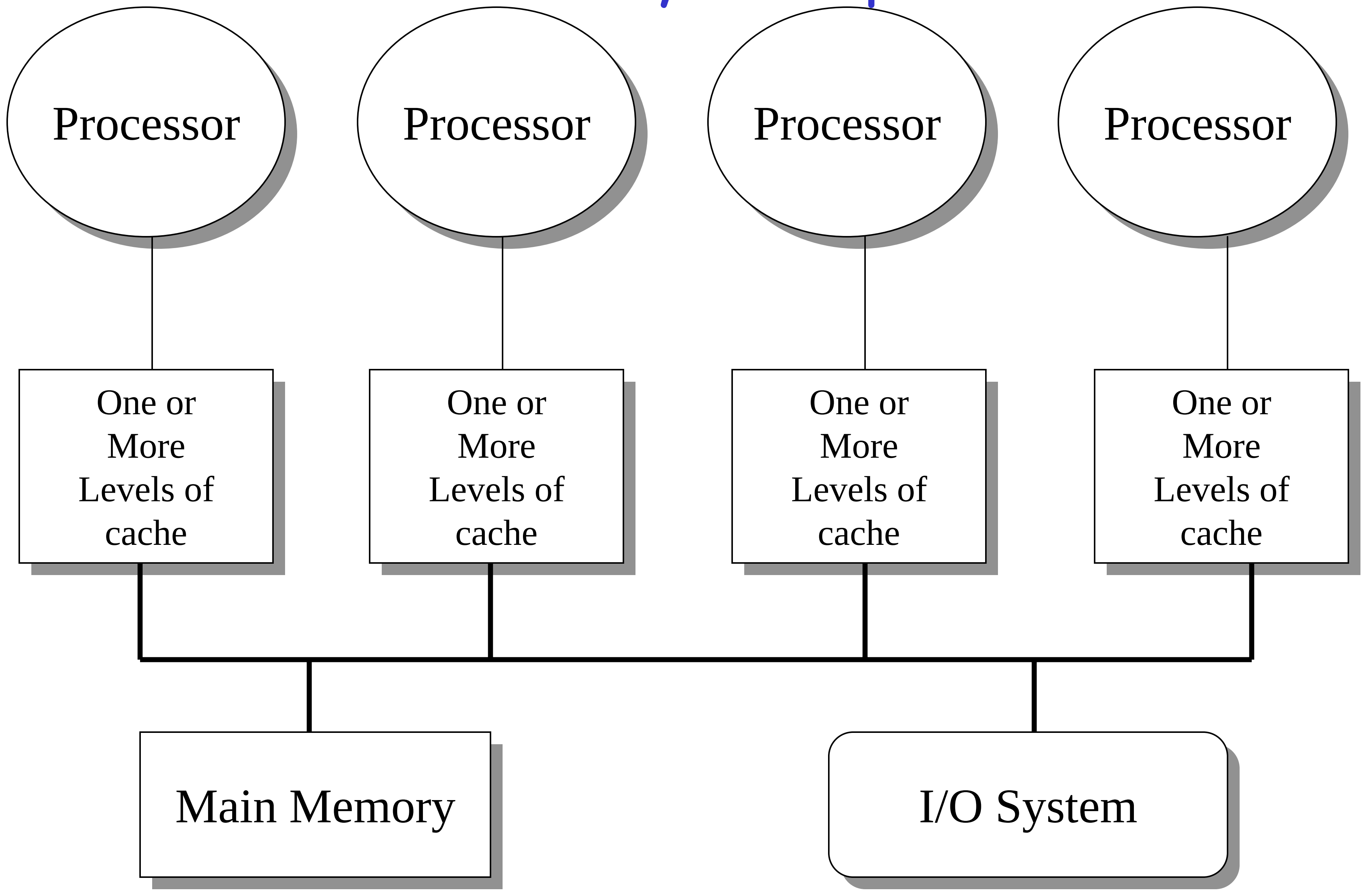
80 = 1 / [(FracX/100 + (1-FracX)]

0.8*FracX + 80*(1-FracX) = 80 - 79.2*FracX = 1

FracX = (80-1)/79.2 = 0.9975

- Only 0.25% should be sequential!

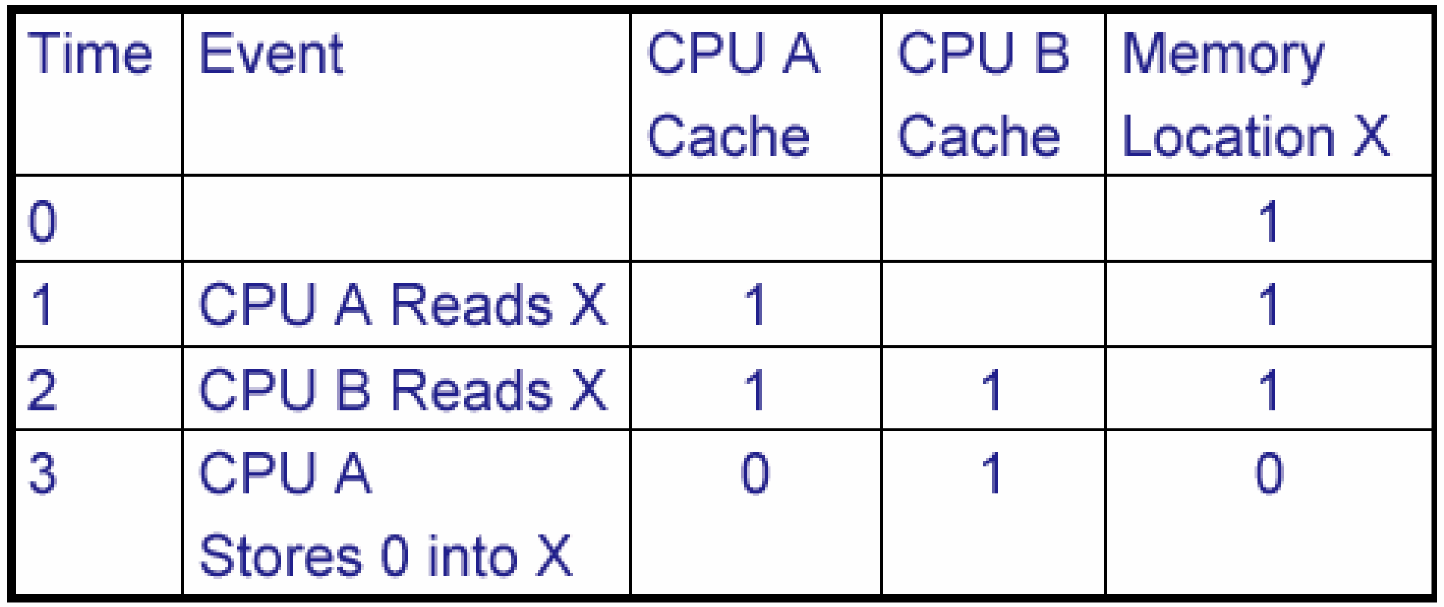
Basic Structure of a Symmetric Shared Memory Multiprocessor


Processor
Processor
Processor
Processor
One or More Levels of cache
One or More Levels of cache
One or More Levels of cache
One or More Levels of cache
Main Memory
I/O System

Symmetric Shared-Memory Architectures - Use of Caches

- Caches serve to:
  - Increase bandwidth versus bus/memory
  - Reduce latency of access
  - Valuable for both private data and shared data
- What about cache consistency?

What is Multiprocessor Cache Coherence?



Time	Event	CPU A Cache	CPU B Cache	Memory Location X
0				1
1	CPU A Reads X	1		1
2	CPU B Reads X	1	1	1
3	CPU A Stores 0 into X	0	1	0

What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write"
  - Too strict and too difficult to implement
- Better:
  - "Any write must eventually be seen by a read"
  - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
  - "If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"
  - Writes to a single location are serialized: seen in one order
    - » Latest write will be seen
    - » Otherwise could see writes in illogical order (could see older value after a newer value)

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
  - 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

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

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

Snooping Cache Variations

Basic Protocol	Berkeley Protocol	Illinois Protocol	MESI Protocol
	Owned Exclusive	Private Dirty	**M**odfied (private,!=Memory)
Exclusive	Owned Shared	Private Clean	e**X**clusive (private,=Memory)
Shared	Shared	Shared	**S**hared (shared,=Memory)
Invalid	Invalid	Invalid	**I**nvalid

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

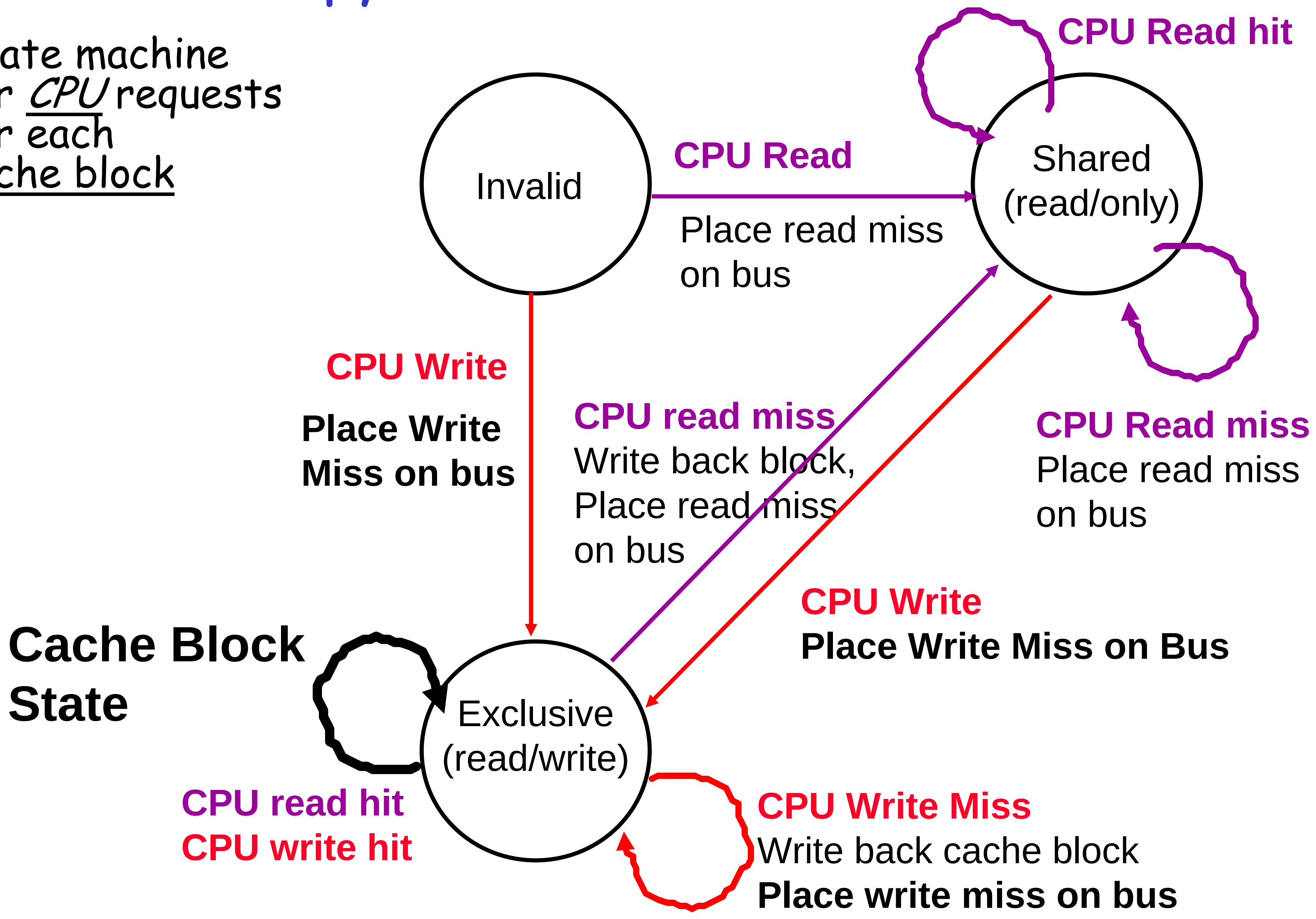
If read sourced from memory, then Private Clean
if read sourced from other cache, then Shared
Can write in cache if held private clean or dirty

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 (<u>Shared</u>)
  - OR Dirty in exactly one cache (<u>Exclusive</u>)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - <u>Shared</u> : block can be read
  - OR <u>Exclusive</u> : cache has only copy, its writeable, and dirty
  - OR <u>Invalid</u> : block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses

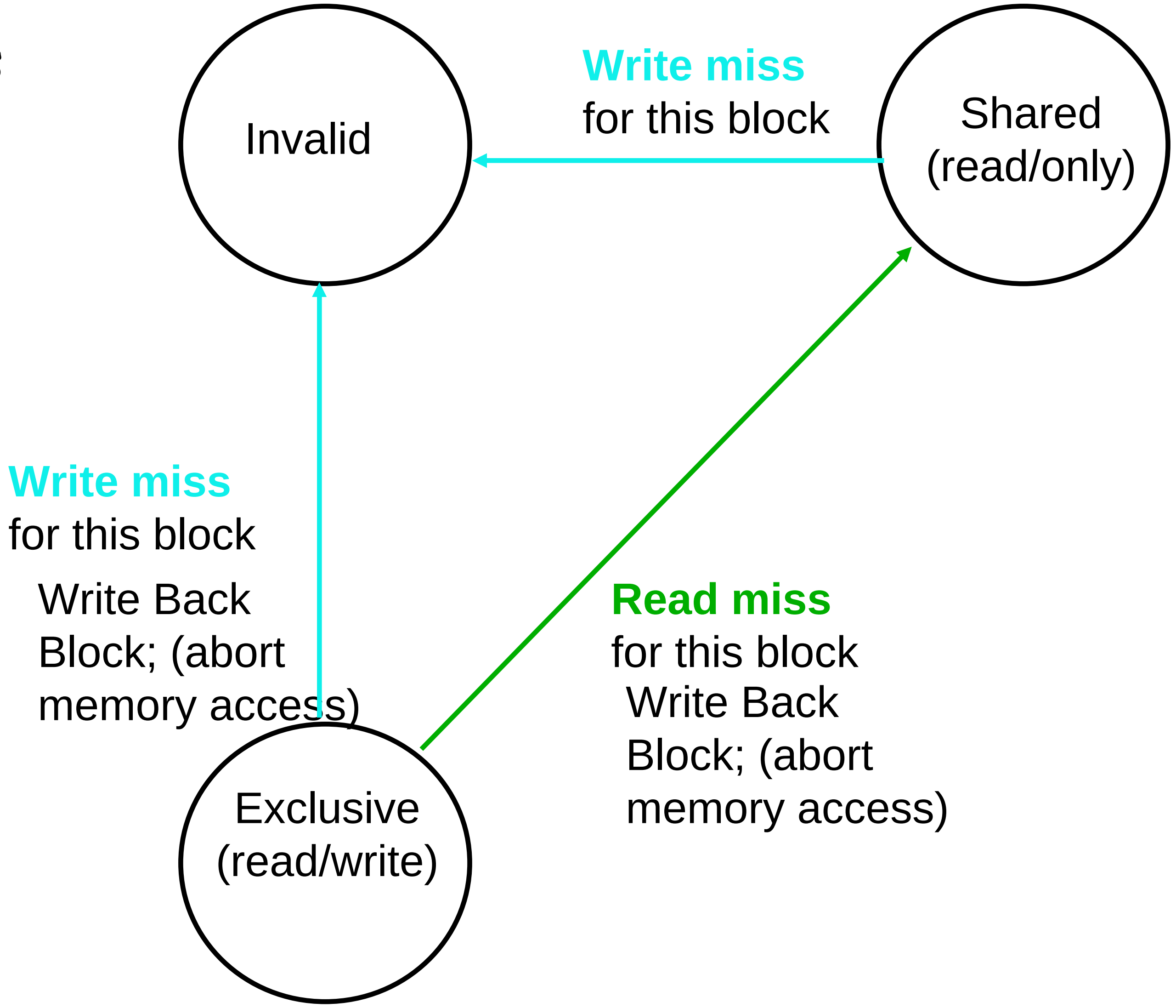
Snoopy-Cache State Machine-I

- State machine for *CPU* requests for each cache block



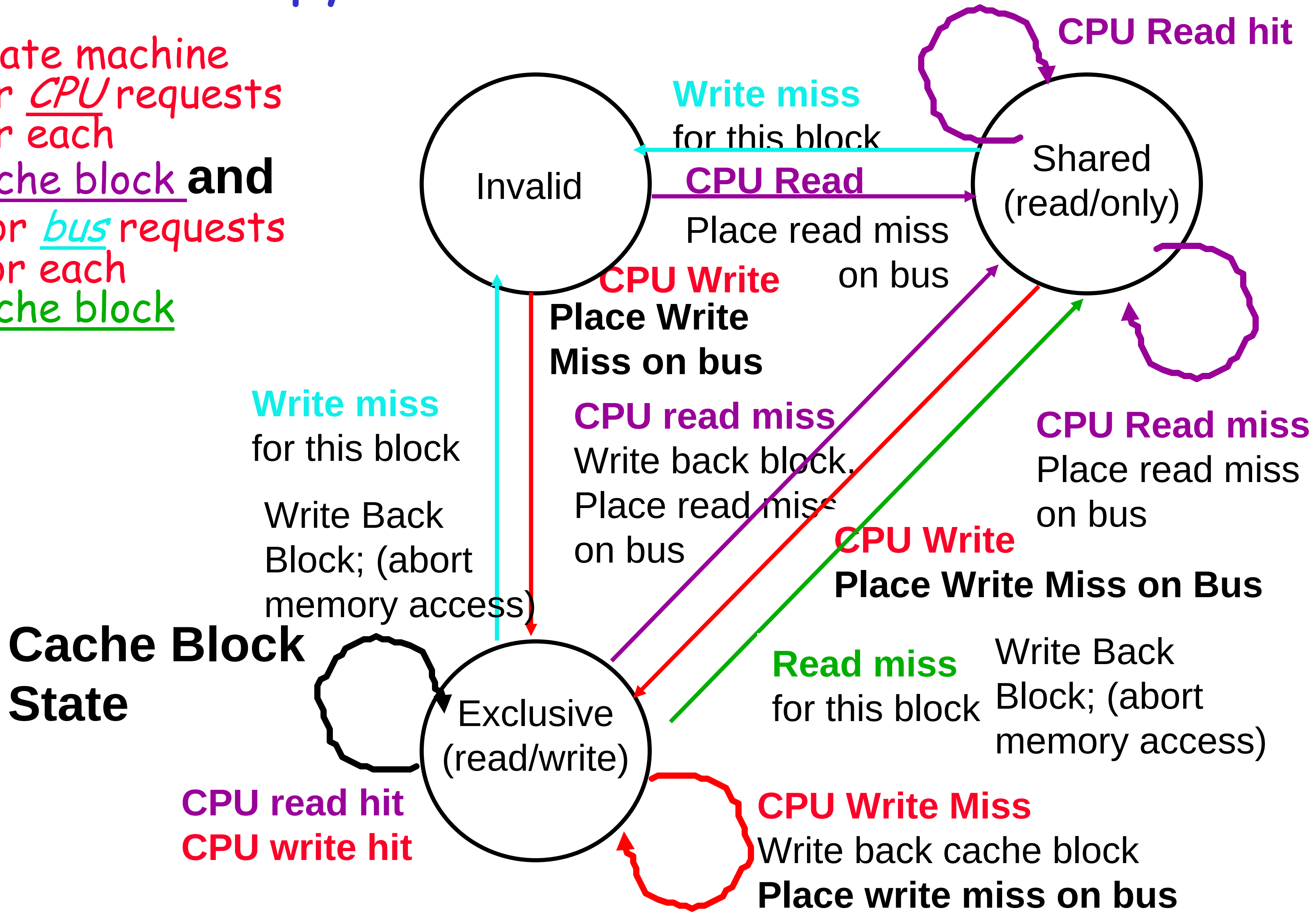
Snoopy-Cache State Machine-II

- State machine for *bus* requests for each cache block


Invalid
Shared (read/only)
Exclusive (read/write)
Write miss for this block
Write miss for this block
Write Back Block; (abort memory access)
Read miss for this block
Write Back Block; (abort memory access)

Snoopy-Cache State Machine-III

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



Example

	P1			P2			Bus				Memory	
step	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 A1 and A2 map to same cache block,
initial cache state is invalid

Example

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

Assumes A1 and A2 map to same cache block

Example

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

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			*P2*			*Bus*				*Memory*	
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*Value*
P1 Write 10 to A1	*Excl.*	*A1*	*10*				*WrMs*	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				*Shar.*	*A1*		*RdMs*	P2	A1			
	Shar.	A1	10				*WrBk*	P1	A1	10	A1	*10*
				Shar.	A1	*10*	*RdDa*	P2	A1	10	A1	10
P2: Write 20 to A1	*Inv.*			*Excl.*	A1	*20*	*WrMs*	P2	A1		A1	10
P2: Write 40 to A2							*WrMs*	P2	A2		A1	10
				Excl.	*A2*	*40*	*WrBk*	P2	A1	20	A1	*20*

Assumes A1 and A2 map to same cache block,
but A1 != A2

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

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 just to check:
  - 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

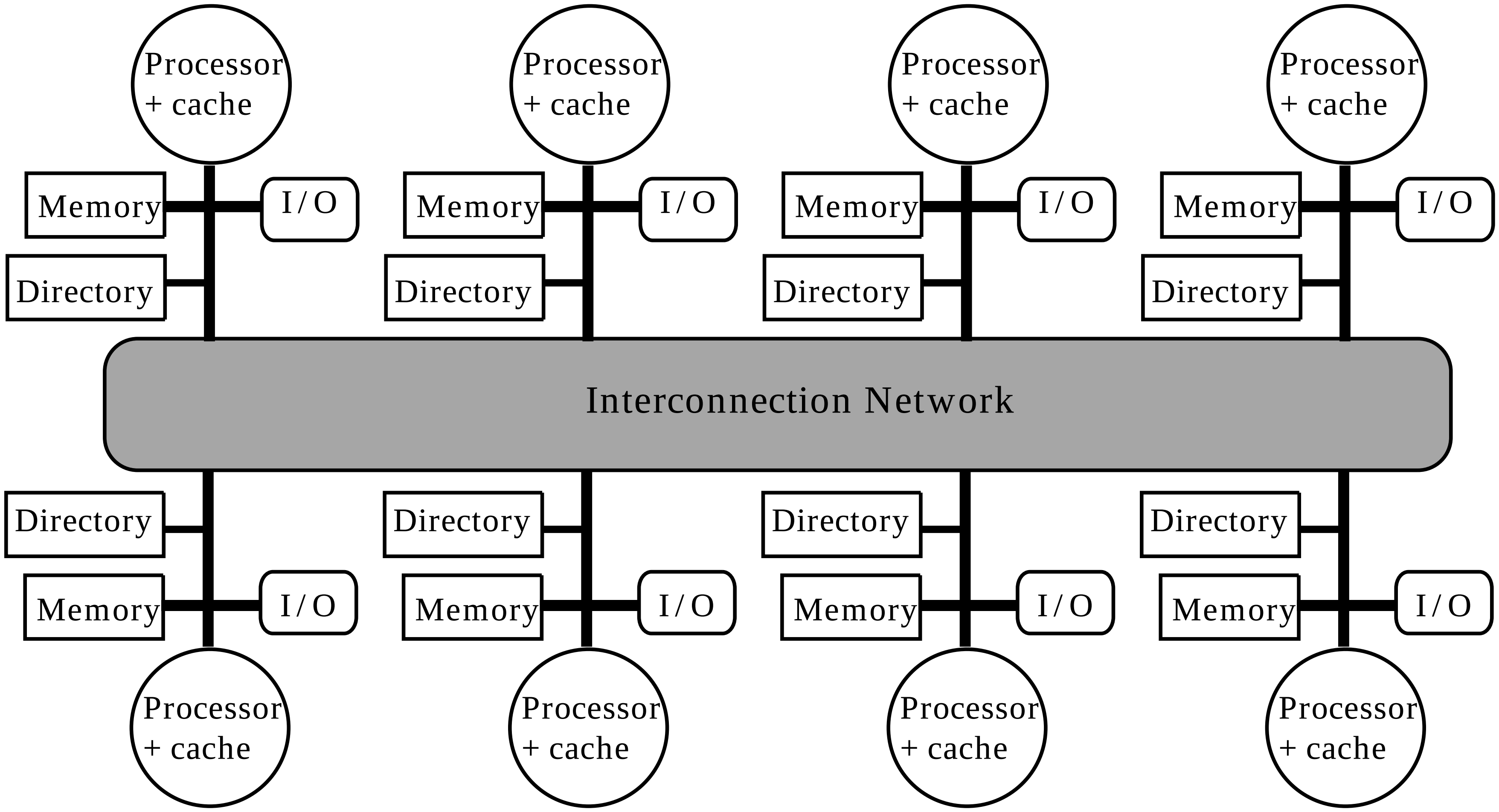
Implementing Snooping Caches

- 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)

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*(memory size) vs. *f*(cache size)
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which Procs have copies of their blocks

Distributed Directory MPs



Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor hasit; 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

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

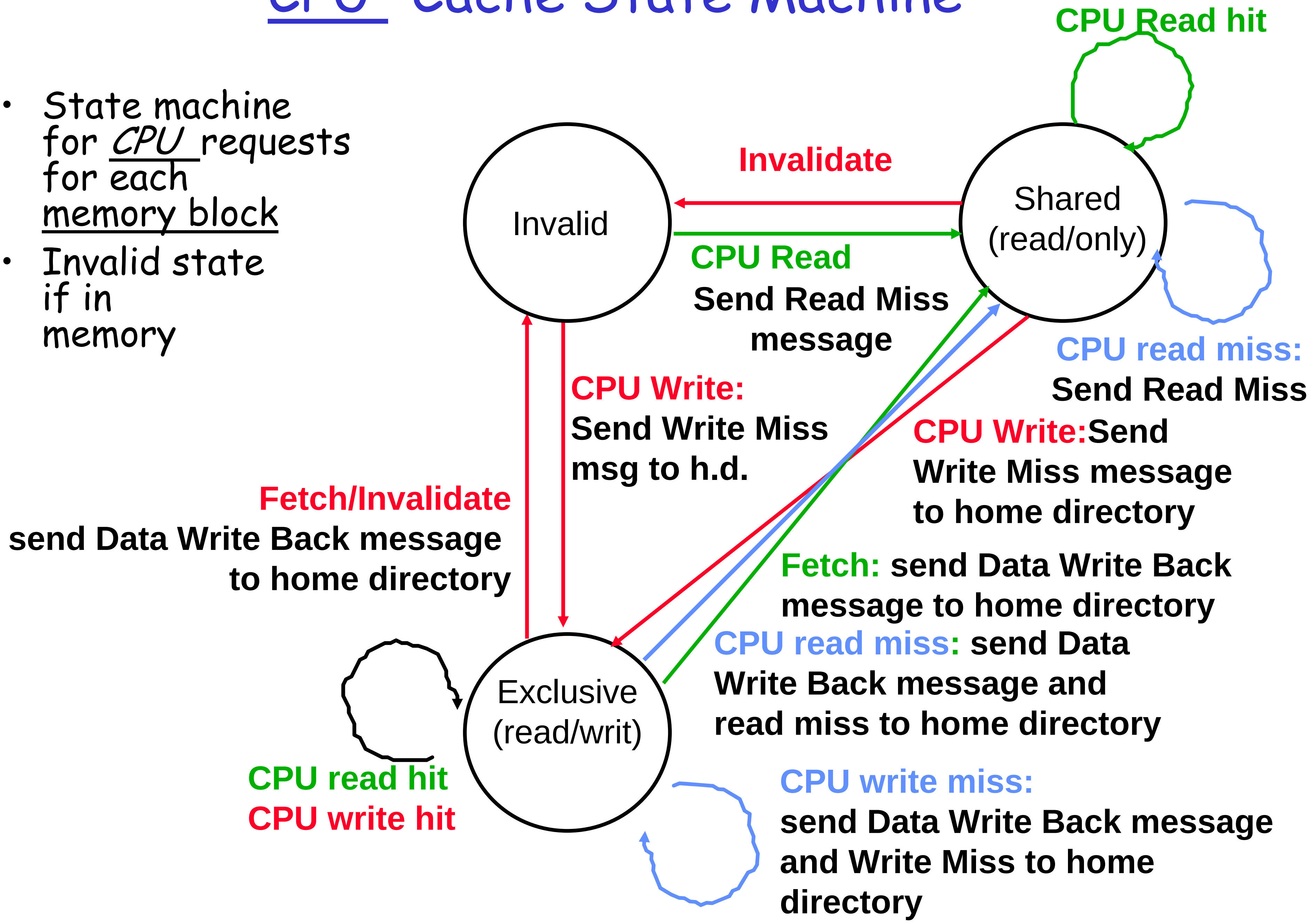
Message type	*Source*	*Destination*	*Msg Content*
Read miss	Local cache	Home directory	P, A
– *Processor P reads data at address A; make P a read sharer and arrange to send data back*			
Write miss	Local cache	Home directory	P, A
– *Processor P writes data at address A; make P the exclusive owner and 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)*			

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

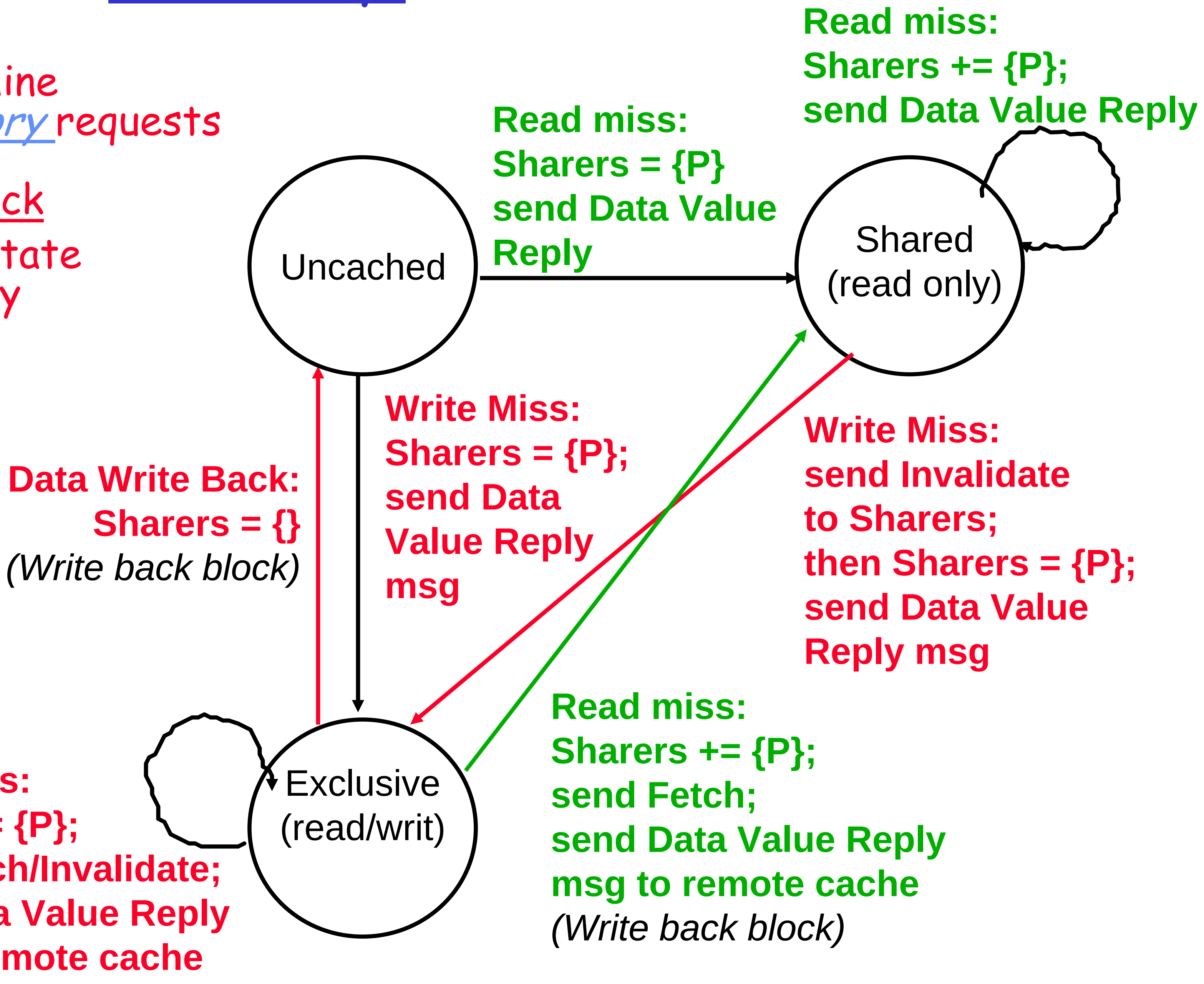

CPU Read hit
Invalid
Shared (read/only)
Invalidate
CPU Read
Send Read Miss message
CPU read miss: Send Read Miss
CPU Write: Send Write Miss msg to h.d.
CPU Write:Send Write Miss message to home directory
Fetch/Invalidate send Data Write Back message to home directory
Fetch: send Data Write Back message to home directory
CPU read miss: send Data Write Back message and read miss to home directory
Exclusive (read/writ)
CPU read hit
CPU write hit
CPU write miss: send Data Write Back message and Write Miss to home directory

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 statisfy 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



Uncached
Shared (read only)
Exclusive (read/writ)
Read miss: Sharers = {P} send Data Value Reply
Read miss: Sharers += {P}; send Data Value Reply
Write Miss: Sharers = {P}; send Data Value Reply msg
Data Write Back: Sharers = {} (Write back block)
Write Miss: send Invalidate to Sharers; then Sharers = {P}; send Data Value Reply msg
Read miss: Sharers += {P}; send Fetch; send Data Value Reply msg to remote cache (Write back block)
Write Miss: Sharers = {P}; send Fetch/Invalidate; send Data Value Reply msg to remote cache

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 <u>only</u> 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.

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.

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			*P2*			*Bus*				*Directory*			*Memory*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*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

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			*P2*			*Bus*				*Directory*			*Memory*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1							*WrMs*	P1	A1		*A1*	*Ex*	*{P1}*	
	Excl.	*A1*	*10*				*DaRp*	P1	A1	0				
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

Example

Processor 1 **Processor 2** **Interconnect** **Directory** **Memory**

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
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

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1														
P2: Write 40 to A2														

Write Back

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			*P2*			*Bus*				*Directory*			*Memory*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1							*WrMs*	P1	A1		*A1*	*Ex*	*{P1}*	
	Excl.	*A1*	*10*				*DaRp*	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				*Shar.*	*A1*		*RdMs*	P2	A1					
	Shar.	A1	10				*Ftch*	P1	A1	10	*A1*			*10*
				Shar.	A1	*10*	*DaRp*	P2	A1	10	A1	*Shar.*	*P1,P2}*	10
P2: Write 20 to A1				Excl.	A1	*20*	*WrMs*	P2	A1					10
	Inv.						*Inval.*	P1	A1		A1	*Excl.*	*{P2}*	10
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			*P2*			*Bus*				*Directory*			*Memory*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1							*WrMs*	P1	A1		*A1*	*Ex*	*{P1}*	
	Excl.	*A1*	*10*				*DaRp*	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				*Shar.*	*A1*		*RdMs*	P2	A1					
	Shar.	A1	10				*Ftch*	P1	A1	10	*A1*			*10*
				Shar.	A1	*10*	*DaRp*	P2	A1	10	A1	*Shar.*	*P1,P2}*	10
P2: Write 20 to A1				Excl.	A1	*20*	*WrMs*	P2	A1					10
	Inv.						*Inval.*	P1	A1		A1	*Excl.*	*{P2}*	10
P2: Write 40 to A2							*WrMs*	P2	A2		*A2*	*Excl.*	*{P2}*	0
							WrBk	P2	A1	20	*A1*	*Unca.*	*{}*	*20*
				Excl.	*A2*	*40*	*DaRp*	P2	A2	0	A2	Excl.	{P2}	0

A1 and A2 map to the same cache block

Implementing a Directory

- We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of bufffers 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
- Message passing is implicit coordination with transmission or arrival of data
- Shared address
  => additional operations to explicitly coordinate:
  e.g., write a flag, awaken a thread, interrupt a processor
- Issues for Synchronization:
  - Uninterruptible 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

Uninterruptible 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

Uninterruptible 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)
```

User Level Synchronization—Operation Using this Primitive

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

```
                li      R2,#1
lockit: 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:            li      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?
```

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

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

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