ECE 552 / CPS 550 Advanced Computer Architecture I

Lecture 18 Multiprocessors

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall12.html

ECE552 Administrivia

13 November – Homework #4 Due

Project Status

- Plan on having preliminary data or infrastructure

8 November – Class Discussion

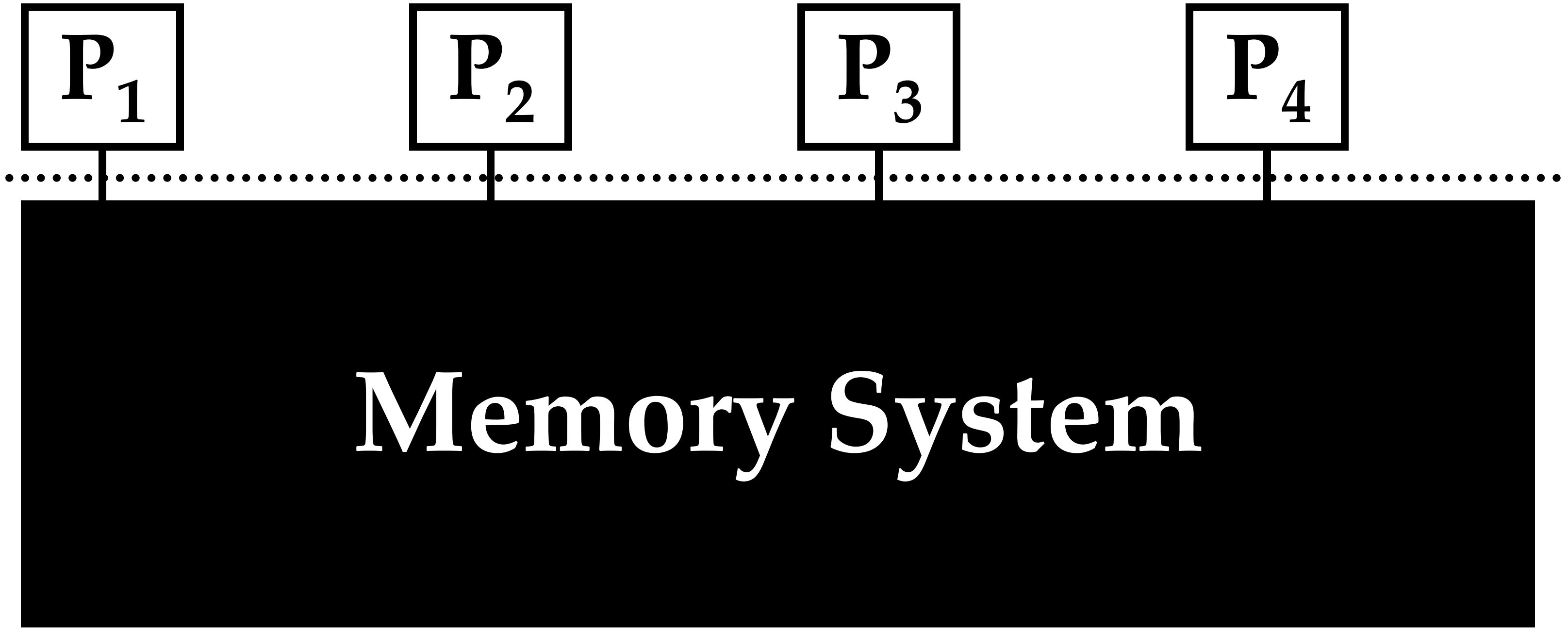
Roughly one reading per class. Do not wait until the day before!

1. Mudge, “Power: A first-class architectural design constraint”
2. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs”
3. Lenoski et al. “The Stanford DASH Multiprocessor”
4. Tullsen et al. “Simultaneous multithreading: Maximizing on-chip parallelism”

Multiprocessors

Shared-memory Multiprocessors

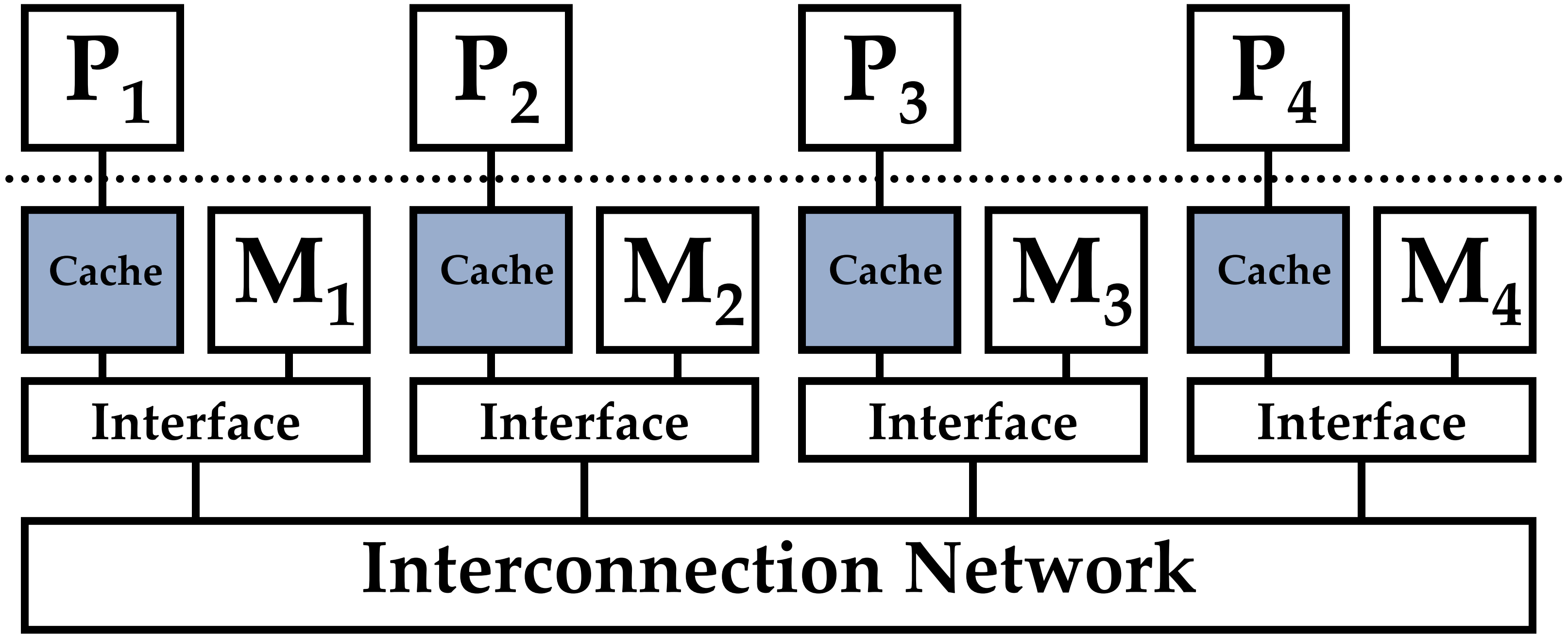
- Provide a shared-memory abstraction
- Enables familiar and efficient programmer interface



Multiprocessors

Shared-memory Multiprocessors

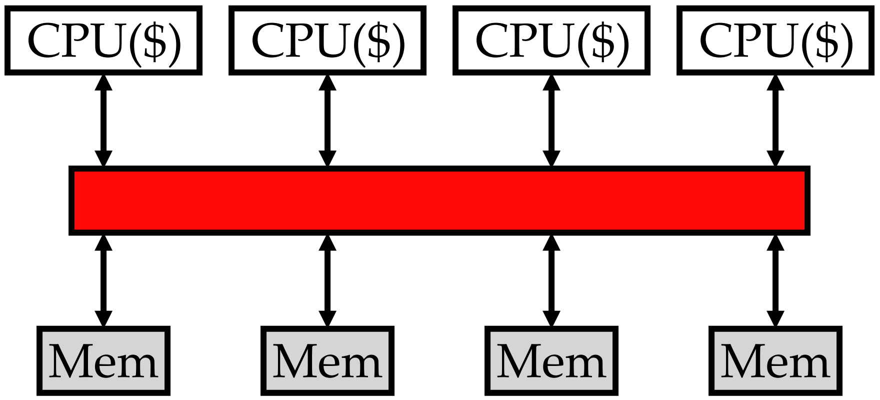
- Provide a shared-memory abstraction
- Enables familiar and efficient programmer interface



Processors and Memory – UMA

Uniform Memory Access (UMA)

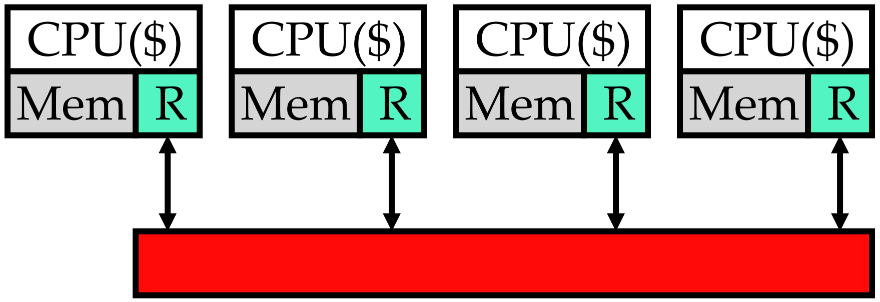
- Access all memory locations with same latency
- Pros: Simplifies software. Data placement does not matter
- Cons: Lowers peak performance. Latency defined by worst case
- Implementation: Bus-based UMA for symmetric multiprocessor (SMP)



Processors and Memory – NUMA

Non-Uniform Memory Access (NUMA)

- Access local memory locations faster
- Pros: Increases peak performance.
- Cons: Increases software complexity, data placement.
- Implementation: Network-based NUMA with various network topologies, which require routers (R).



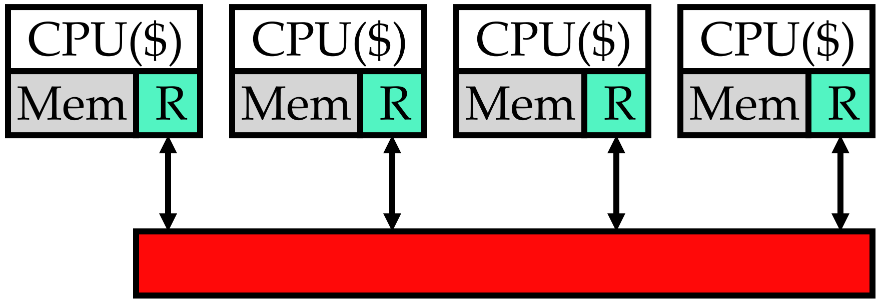
Networks and Topologies

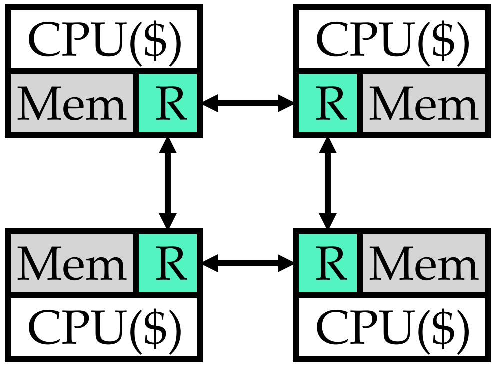
Shared Networks

- Every CPU can communicate with every other CPU via bus or crossbar
- Pros: lower latency
- Cons: lower bandwidth and more difficult to scale with processor count (e.g., 16)

Point-to-Point Networks

- Every CPU can talk to specific neighbors (depending on topology).
- Pros: higher bandwidth and easier to scale with processor count (e.g., 100s)
- Cons: higher multi-hop latencies





Topology 1 – Bus

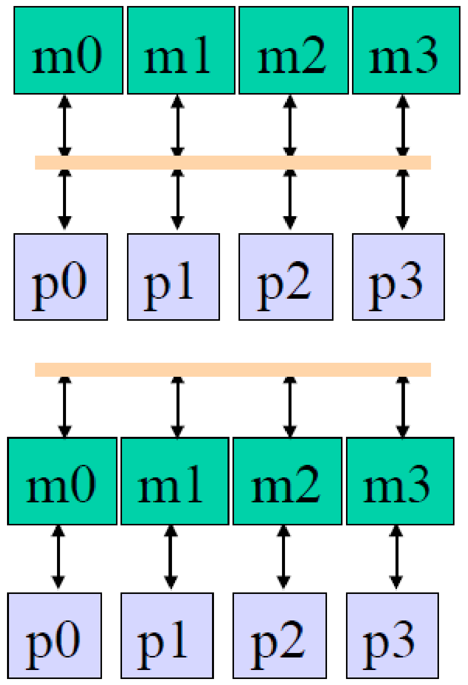
Network Topology

- Defines organization of network nodes
- Topologies differ in connectivity, latency, bandwidth, and cost.
- Notation: f(1) denotes constant independent of p, f(p) denotes linearly increasing cost with p, etc…

Bus

- Direct interconnect style

- Latency: f(1) wire delay
- Bandwidth: f(1/p) and not scalable (p<=4)
- Cost: f(1) wire cost
- Supports <u>ordered broadcast</u> only



Topology 2 – Crossbar Switch

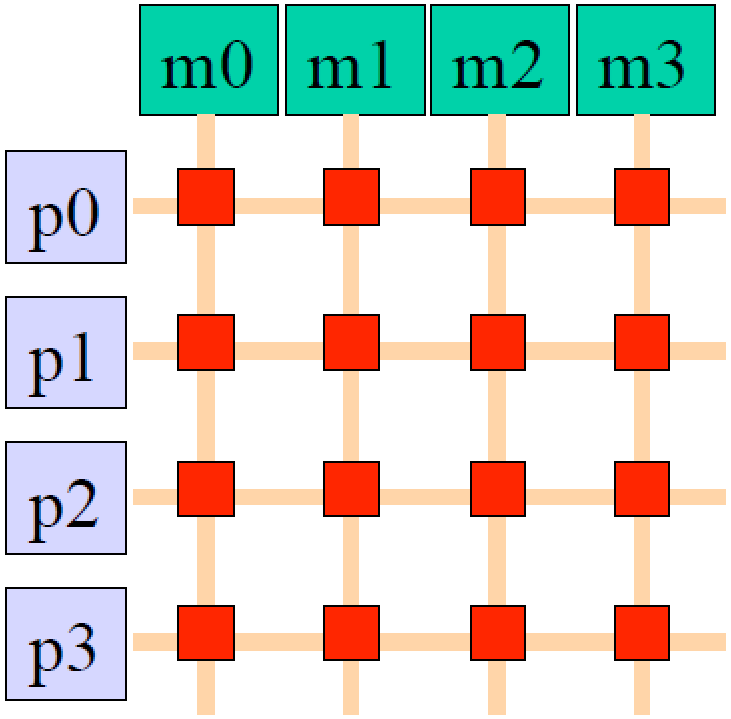
Network Topology

- Defines organization of network nodes
- Topologies differ in connectivity, latency, bandwidth, and cost.
- Notation: f(1) denotes constant independent of p, f(p) denotes linearly increasing cost with p, etc…

Crossbar Switch

- Indirect interconnect.
- Switches implemented as big multiplexors

- Latency: f(1) constant latency
- Bandwidth: f(1)
- Cost: f(2P) wires, f(P^2) switches



Topology 3 – Multistage Network

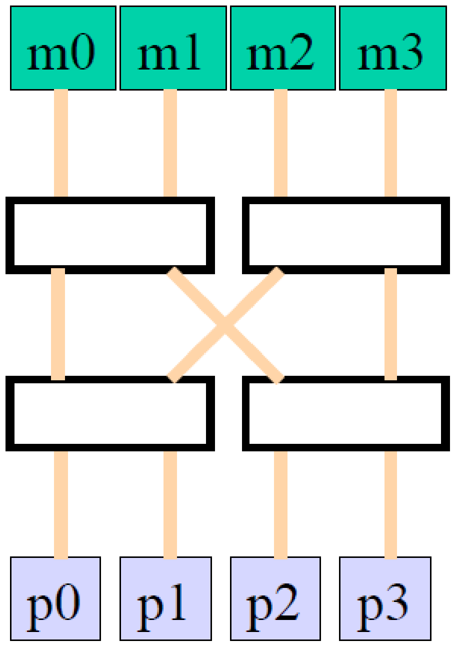
Network Topology

- Defines organization of network nodes
- Topologies differ in connectivity, latency, bandwidth, and cost.
- Notation: f(1) denotes constant independent of p, f(p) denotes linearly increasing cost with p, etc…

Crossbar Switch

- Indirect interconnect.
- Routing done by address decoding
- k: switch arity (#inputs or #outputs)
- d: number of network stages = $\log_k P$

- Latency: f(d)
- Bandwidth: f(1)
- Cost: f(d*P/k) switches, f(P*d) wires
- Commonly used in large UMA systems



Topology 4 – 2D Torus

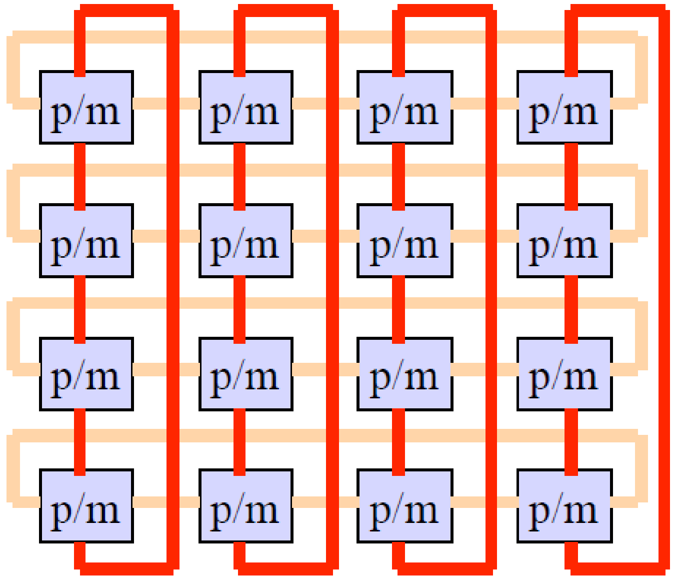
Network Topology

- Defines organization of network nodes
- Topologies differ in connectivity, latency, bandwidth, and cost.
- Notation: f(1) denotes constant independent of p, f(p) denotes linearly increasing cost with p, etc…

2D Torus

- Direct interconnect

- Latency: $f(P^{1/2})$
- Bandwidth: f(1)
- Cost: f(2P) wires
- Scalable and widely used.
- Variants: 1D torus, 2D mesh, 3D torus



Challenges in Shared Memory

Cache Coherence

- **“Common Sense”**

- **P1-Read[X] → P1-Write[X] → P1-Read[X]**	**Read returns X**
- **P1-Write[X] → P2-Read[X]**	**Read returns value written by P1**
- **P1-Write[X] → P2-Write[X]**	**Writes serialized**
	All P’s see writes in same order

Synchronization

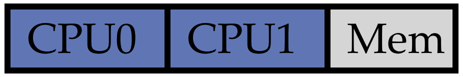
- Atomic read/write operations

Memory Consistency

- What behavior should programmers expect from shared memory?
- Provide a formal definition of memory behavior to programmer
- Example: When will a written value be seen?
- Example: P1-Write[X] <<10ps>> P2-Read[X]. What happens?

Example Execution

```
Processor 0                        Processor 1
0:  addi   r1, accts, r3
1:  ld     0(r3), r4
2:  blt    r4, r2, 6
3:  sub    r4, r2, r4
4:  st     r4, 0 (r3)
5:  call   give-cash               0: addi    r1, accts, r3       # get addr for account
                                   1: ld      0(r3), r4           # load balance into r4
                                   2: blt     r4, r2, 6           # check for sufficient funds
                                   3: sub     r4, r2, r4          # withdraw
                                   4: st      r4, 0(r3)           #store new balance
                                   5: call    give-cash
```



Two withdrawals from one account. Two ATMs

- Withdraw value: r2 (e.g., $100)
- Account memory address: accts+r1
- Account balance: r4

Scenario 1 – No Caches

```
Processor 0                        Processor 1                         Mem
0:  addi   r1, accts, r3                                               500
1:  ld     0(r3), r4                                                   500
2:  blt    r4, r2, 6
3:  sub    r4, r2, r4
4:  st     r4, 0 (r3)                                                  400
5:  call   give-cash               0: addi   r1, accts, r3
                                   1: ld     0(r3), r4                 400
                                   2: blt    r4, r2, 6
                                   3: sub    r4, r2, r4
                                   4: st     r4, 0(r3)                 300
                                   5: call   give-cash
```

Processors have no caches

- Withdrawals update balance without a problem

Scenario 2a – Cache Incoherence

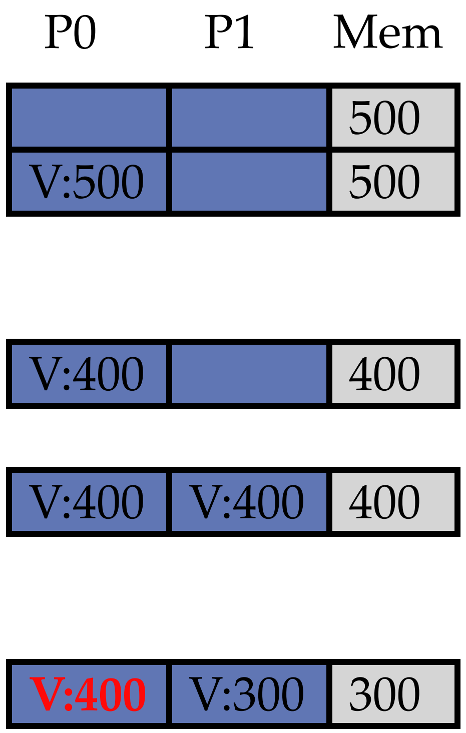
Processor 0		Processor 1		P0	P1	Mem
0: addi	r1, accts, r3					500
1: ld	0(r3), r4			V:500		500
2: blt	r4, r2, 6					
3: sub	r4, r2, r4					
4: st	r4, 0 (r3)			D:400		500
5: call	give-cash	0: addi	r1, accts, r3			
		1: ld	0(r3), r4	D:400	V:500	500
		2: blt	r4, r2, 6			
		3: sub	r4, r2, r4			
		4: st	r4, 0(r3)	D:400	D:400	500
		5: call	give-cash			

Processors have write-back caches

- Processor 0 updates balance in cache, but does not write-back to memory
- Multiple copies of memory location [accts+r1]
- Copies may get inconsistent

Scenario 2b – Cache Incoherence

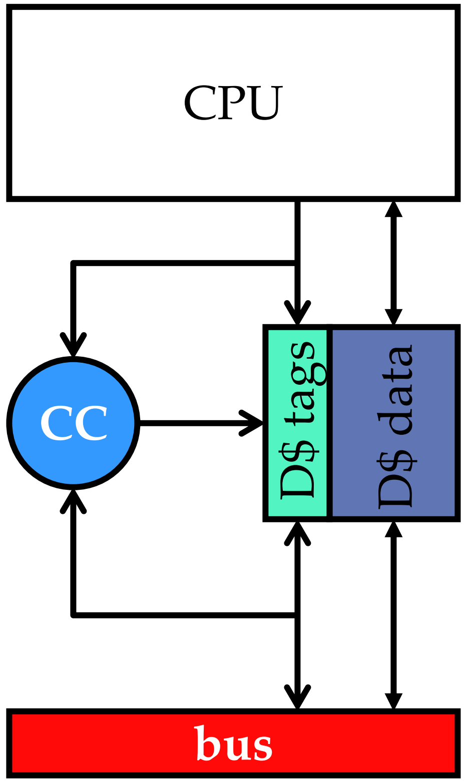
```
Processor 0                     Processor 1
0:  addi   r1, accts, r3
1:  ld     0(r3), r4
2:  blt    r4, r2, 6
3:  sub    r4, r2, r4
4:  st     r4, 0 (r3)
5:  call   give-cash        0: addi    r1, accts, r3
                            1: ld      0(r3), r4
                            2: blt     r4, r2, 6
                            3: sub     r4, r2, r4
                            4: st      r4, 0(r3)
                            5: call    give-cash
```



Processors have write-through caches

- What happens if processor 0 performs another withdrawal?

Hardware Coherence Protocols



Absolute Coherence

- All cached copies have same data at same time. Slow and hard to implement

Relative Coherence

- Temporary incoherence is ok (e.g., write-back caches) as long as no load reads incoherent data.

Coherence Protocol

Finite state machine that runs for every cache line

(1) Define states per cache line

(2) Define state transitions based on bus activity

(3) Requires coherence controller to examine bus traffic (address, data)

(4) Invalidates, updates cache lines

Protocol 1 – Write Invalidate

Mechanics

- Process P performs write, broadcasts address on bus
- !P snoop the bus. If address is locally cached, !P invalidates local copy

- Process P performs read, broadcasts address on bus
- !P snoop the bus. If address is locally cached, !P writes back local copy

Example

Processor-Activity	Bus-Activity	Data in Cache-A	Data in Cache-B	Data in Mem[X]
				0
CPU-A reads X	Cache miss for X	0		0
CPU-B reads X	Cache miss for X	0	0	0
CPU-A writes 1 to X	Invalidation for X	1		0
CPU-B reads X	Cache miss for X	1	1	1

Protocol 2 – Write Update

Mechanics – processor P performs write

- Do not invalidate !P cache line.
- Instead update !P cache line and memory
- Pro: !P gets data faster
- Con: Requires significant bandwidth

Processor-Activity	Bus-Activity	Data in Cache-A	Data in Cache-B	Data in Mem[X]
				0
CPU-A reads X	Cache miss for X	0		0
CPU-B reads X	Cache miss for X	0	0	0
CPU-A writes 1 to X	**Write Broadcast X**	**1**	**1**	**1**
CPU-B reads X	Cache hit for X	1	1	1

Cache Coherent Systems

Provide Coherence Protocol

- States
- State transition diagram
- Actions

Implement Coherence Protocol

(0) Determine when to invoke coherence protocol
(1) Find state of cache line to determine action
(2) Locate other cached copies
(3) Communicate with other cached copies (invalidate, update)

Implementation Variants

(0) is done in the same way for all systems. Maintain additional state per cache line. Invoke protocol based on state

(1-3) have different approaches

Implementation 1 – Snooping

Bus-based Snooping

- All cache/coherence controllers observe/react to all bus events.
- Protocol relies on globally visible events
  - i.e., all processors see all events
- Protocol relies on globally ordered events
  - i.e., all processors see all events in same sequence

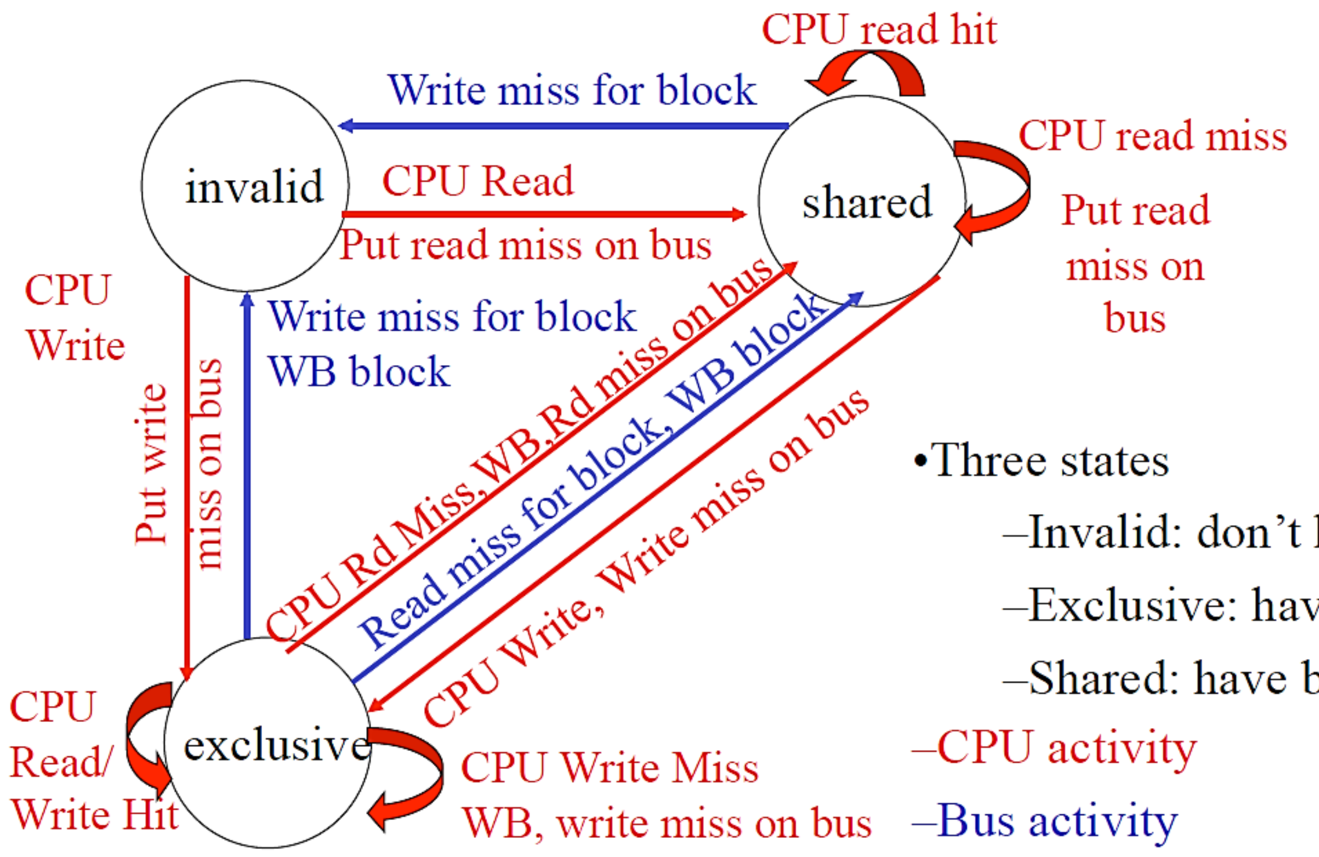
Bus Events

- Processor (events initiated by own processor P)
  - read (R), write (W), write-back (WB)
- Bus (events initiated by other processors !P)
  - bus read (BR), bus write (BW)

Three-State Invalidate Protocol

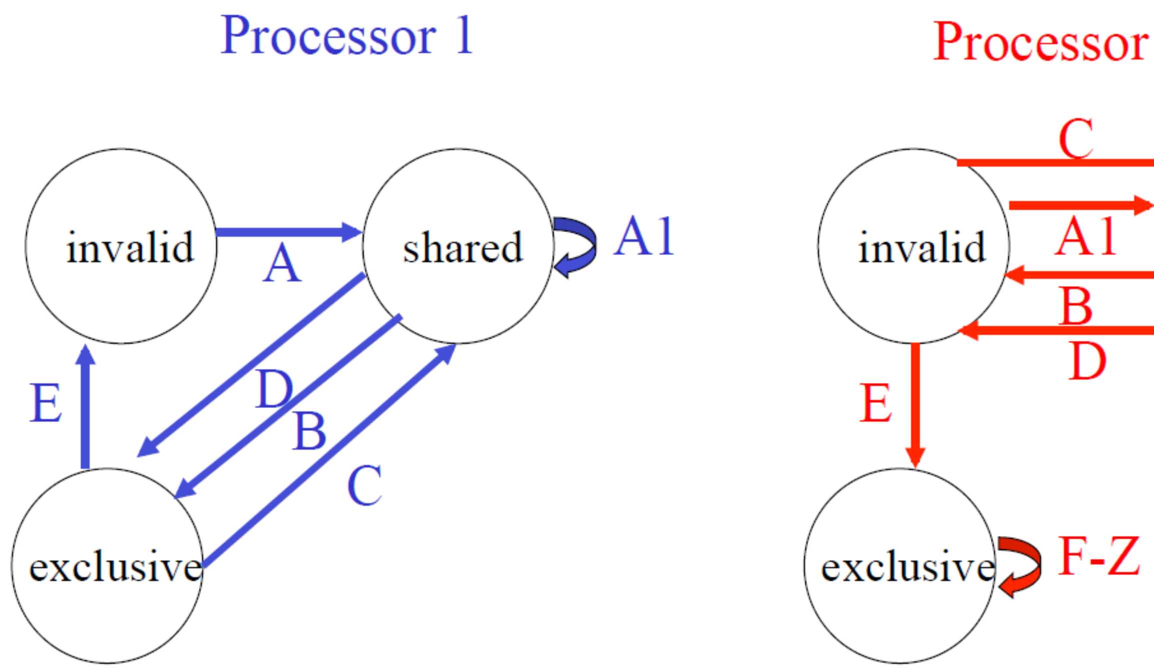
Implement protocol for every cache line.

Add state bits to every cache to indicate (1) invalid, (2) shared, (3) exclusive



- Three states
  - Invalid: don’t have block
  - Exclusive: have block and wrote it
  - Shared: have block but only read it
- CPU activity
- Bus activity

Example



P1 read	(A)
P2 read	(A1)
P1 write	(B)
P2 read	(C)
P1 write	(D)
P2 write	(E)
P2 write	(F-Z)

Implementation 2 – Directory

Bus-based Snooping – Limitations

- Snooping scalability is limited
- Bus has insufficient data bandwidth for coherence traffic
- Processor has insufficient snooping bandwidth for coherence traffic

Directory-based Coherence – Scalable Alternative

- Directory contains state for every cache line
- Directory identifies processors with cached copies and their states
- In contrast to snoopy protocols, processors observe/act only on relevant memory events. Directory determines whether a processor is involved.

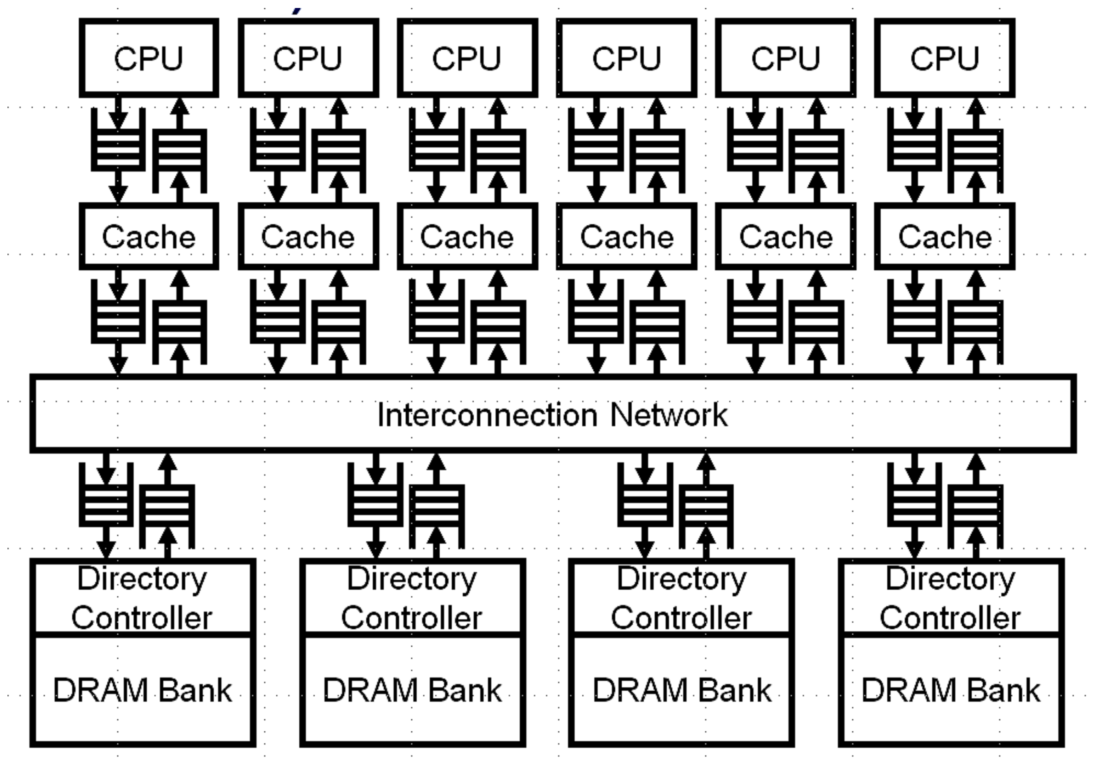
Directory Communication

Processor sends coherence events to directory

(1) Find directory entry

(2) Identify processors with copies

(3) Communicate with processors, if necessary



Challenges in Shared Memory

Cache Coherence

- "Common Sense"	
- P1-Read[X] → P1-Write[X] → P1-Read[X]	Read returns X
- P1-Write[X] → P2-Read[X]	Read returns value written by P1
- P1-Write[X] → P2-Write[X]	Writes serialized
	All P's see writes in same order

Synchronization

- **Atomic read/write operations**

Memory Consistency

- What behavior should programmers expect from shared memory?
- Provide a formal definition of memory behavior to programmer
- Example: When will a written value be seen?
- Example: P1-Write[X] <<10ps>> P2-Read[X]. What happens?

Synchronization

Regulate access to data shared by processors

- Synchronization primitive is a lock
- Critical section is a code segment that accesses shared data
- Processor must acquire lock before entering critical section.
- Processor should release lock when exiting critical section

Spin Locks – Broken Implementation

```
acquire (lock)          # if lock=0, then set lock = 1, else spin
critical section
release (lock)          # lock = 0

Inst-0: ldw    R1, lock        # load lock into R1
Inst-1: bnez   R1, Inst-0      # check lock, if lock!=0, go back to Inst-0
Inst-2: stw    1, lock         # acquire lock, set to 1
<< critical section>>>         # access shared data
Inst-n: stw    0, lock         # release lock, set to 0
```

Implementing Spin Locks

Processor 0	Processor 1	
Inst-0: ldw R1, lock		
Inst-1: bnez R1,Inst-0		# P0 sees lock is free
	Inst-0: ldw R1, lock	
	Inst-1: bnez R1, Inst-0	# P1 sees lock is free
Inst-2: stw 1, lock		# P0 acquires lock
	Inst-2: stw 1, lock	# P1 acquires lock
…..		
	….	# P0/P1 in critical section
….		# at the same time
Inst-n: stw 0, lock		

Problem: Lock acquire not atomic

- A set of <u>atomic</u> operations either all complete or all fail. During a set of atomic operations, no other processor can interject.

- Spin lock requires atomic <u>load-test-store</u> sequence

Implementing Spin Locks

Solution: Test-and-set instruction

- Add single instruction for load-test-store (t&s R1, lock)
- Test-and-set atomically executes

```
ld R1, lock;          # load previous lock value
st 1, lock;           # store 1 to set/acquire
```

- If lock initially free (0), t&s acquires lock (sets to 1)
- If lock initially busy (1), t&s does not change it
- Instruction is un-interruptible/atomic by definition

```
Inst-0        t&s R1, lock       # atomically load, check, and set lock=1
Inst-1        bnez R1            # if previous value of R1 not 0,
....                             acquire unsuccessful
Inst-n        stw R1, 0          # atomically release lock
```

Test-and-Set Inefficiency

Test-and-set works…

Processor 0	Processor 1	
Inst-0: t&s R1, lock		
Inst-1: bnez R1,Inst-0	Inst-0: t&s R1, lock	# P0 sees lock is free
	Inst-1: bnez R1, Inst-0	# P1 does not acquire

…but performs poorly

- Suppose Processor 2 (not shown) has the lock
- Processors 0/1 must…
  - Execute a loop of t&s instructions
  - Issue multiple store instructions
  - Generate useless interconnection traffic

Test-and-Test-and-Set Locks

Solution: Test-and-test-and-set

```
Inst-0        ld R1, lock        # test with a load, see if lock changed
Inst-1        bnez R1, Inst-0    # if lock=1, spin
Inst-2        t&s R1, lock       # if lock=1, test-and-set
Inst-4        bnez R1, Inst-0    # if can not acquire, spin
```

Advantages

- Spins locally without stores
- Reduces interconnect traffic
- Not a new instruction, simply new software (lock implementation)

Semaphores

Semaphore (semaphore S, integer N)

- Allows N parallel threads to access shared variable
- If N = 1, equivalent to lock
- Requires atomic fetch-and-add

```
Function Init (semaphore S, integer N) {
        s = N;
}

Function P (semaphore S) {                   # "Proberen" to test
        while (S == 0) { };
        s = s -1 ;
}

Function V (semaphore S) {                   # "Verhogen" to increment
        s = s + 1;
}
```

Challenges in Shared Memory

Cache Coherence

- “Common Sense”

- P1-Read[X] → P1-Write[X] → P1-Read[X]	Read returns X
- P1-Write[X] → P2-Read[X]	Read returns value written by P1
- P1-Write[X] → P2-Write[X]	Writes serialized
	All P’s see writes in same order

Synchronization

- Atomic read/write operations

Memory Consistency

- **What behavior should programmers expect from shared memory?**
- **Provide a formal definition of memory behavior to programmer**
- **Example: When will a written value be seen?**
- **Example: P1-Write[X] <<10ps>> P2-Read[X]. What happens?**

Memory Consistency

Execution Example

A = Flag = 0

Processor 0	Processor 1
A = 1	while (!Flag)
Flag = 1	print A

Intuition – P1 should print A=1

Coherence – Makes no guarantees!

Consistency and Caches

Execution Example

A = Flag = 0

Processor 0	Processor 1
A = 1	while (!Flag)
Flag = 1	print A

Caching Scenario

1. P0 writes A=1. Misses in cache. Puts write into a store buffer.
2. P0 continues execution.
3. P0 writes Flag=1. Hits in cache. Completes write (with coherence)
4. P1 reads Flag=1.
5. P1 exits spin loop.
6. P1 prints A=0

Caches, buffering, and other performance mechanisms can cause strange behavior.

Sequential Consistency (SC)

Definition of Sequential Consistency

Formal definition of programmers' expected view of memory

(1) Each processor P sees its own loads/stores in program order

(2) Each processor P sees !P loads/stores in program order

(3) All processors see same global load/store ordering.

P and !P loads/stores may be interleaved into some order.

But all processors see the same interleaving/ordering.

Definition of Multiprocessor Ordering [Lamport]

Multi-processor ordering corresponds to some sequential interleaving of uni-processor orderings. Multiprocessor ordering should be indistinguishable from multi-programmed uni-purocessor

Enforcing SC

Consistency and Coherence

- SC Definition: loads/stores globally ordered
- SC Implications: coherence events of all load/stores globally ordered

Implementing Sequential Consistency

- All loads/stores commit in-order

- Delay completion of memory access until all invalidations that are caused by access are complete

- Delay a memory access until previous memory access is complete

- Delay memory read until previous write completes. Cannot place writes in a buffer and continue with reads.

- Simple for programmer but constraints HW/SW performance optimizations

Weaker Consistency Models

Assume programs are synchronized

- SC required only for lock variables
- Other variables are either (1) in critical section and cannot be accessed in parallel or (2) not shared

Use fences to restrict re-ordering

- Increases opportunity for HW optimization but increases programmer effort
- <u>Memory fences</u> stall execution until write buffers empty
- Allows load/store reordering in critical section.
- Slows lock acquire, release

```
acquire
memory fence
critical section
memory fence     # ensures all writes from critical section
release          # are cleared from buffer
```

Summary

Shared Memory Multiprocessors

- Provides efficient and familiar abstraction to programmer
- Much, much more in ECE259

Cache Coherence

- Coordinate accesses to shared, writeable data
- Coherence protocol defines cache line states, state transitions, actions
- Snooping implementation – bus and broadcast
- Directory implementation – directory and

Synchronization

- Locks and ISA support for atomicity

Memory Consistency

- Defines programmers' expected view of memory
- Sequential consistency imposes ordering on loads/stores