ECE565: Computer Architecture

Ch 5

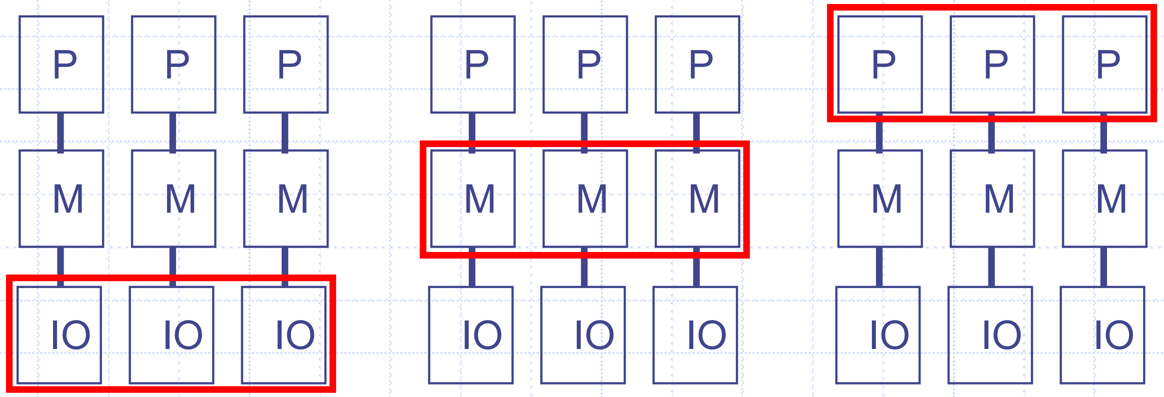
Outline

- Multiprocessors
  - Why?
  - Power changes everything
  - What about performance
    - Two processors of speed X instead of one processor of speed 2X
    - Challenges in harnessing parallelism
  - Is there a single vendor with uniprocessor products? (non-embedded)
- Key concepts
  - Programming models
  - Cache coherence
  - Synchronization
  - Consistency

Why Multicores

- In the past, we got faster clock with every microprocessor generation
- But faster clock led to higher power which cannot be sustained
- Power = CV^2f where C is circuit capacitance, V is voltage, f is clock frequency
- V is linear in f, so power is cubic in f
- 2x clock => 8x power but 2x cores at same clock => 2x power
- We have been lowering voltage but that causes other problems (leakage, reliability)

Various Programming Views



Join At:	I/O (Network)	Memory	Processor
Program With:	Message Passing	Shared Memory	(Dataflow/Systolic), Single-Instruction Multiple-Data (SIMD) ==> Data Parallel

- Our focus: Shared Address Space
  - Most representative design point for multicores

Various Programming Views

for most interesting parallel applications

- parallel processes (threads) need to communicate
- NO assumptions about threads' relative speeds

communication method leads to another division

- message passing
- shared memory
- In message passing and shared memory – programmer writes explicitly programs
  - Shared memory is most general
- In SIMD either parallelized by compiler or written by programmer (eg CUDA)

Shared memory

shared memory - all processors see one global memory

- programs use loads/stores to access data
- + conceptually compatible with uniprocessors
- + ease of programming even if communication complex/dynamic
- + lower latency communicating small data items
  - Real programs tend to communicate small fine-grain data
- + hardware controlled sharing allows automatic data motion

- The pluses/minuses on this and next slide are from programmer’s point of view

Message passing

message passing – each processor sees only its private memory

- programs use sends/receives to access data
- + very simple hardware (almost nothing extra)
- + communication pattern explicit
  - but they MUST be explicit
- + shared memory can emulate message passing easily
- - (BIG MINUS) biggest programming burden: managing communication artifacts
  - Not bad for fixed simple communication patterns
  - Very hard for arbitrary, dynamic communication patterns found in real programs – databases, middleware, ...

Message Passing

- distribute data carefully to threads
  - no automatic data motion
- partition data if possible, replicate if not
  - replicate in s/w not automatic (so extra work to ensure ok)
- coalesce small mesgs into large, gather separate data into one mesg to send and scatter received mesg into separate data

Shared memory

```
• Thread1                              Thread2
• ....                                   .....
• compute (data)                      synchronize
• store( A,B, C, D, ...)         load  (A, B, C, D, ....)
• synchronize                          compute
• ......                                .....
```

- A B C D SAME in both threads- SINGLE shared memory

Message Passing

- Thread1 Thread2
- ....
- compute (data) receive (mesg)
- store (A,B, C, D ..) scatter (mesg to A B C D ..)
- gather (A B C D into mesg) load (A, B, C, D, ...)
- send (mesg) compute
- ......
- .....
- A B C D are DIFFERENT in each thread -- PRIVATE memory

Example

- Ocean - Compute temperature of a part of an ocean
- Very simple example so you can see
- Real parallel programs have much more complex communication patterns – databases, online transaction processing (reservations), decision support (inventory control), web servers, Java middleware
    - – in other words all the real programs that make the world go round!

Sequential Ocean

```
main()
begin
  read(n);
  A = malloc(n * n);
  initialize(A);
  Solve(A);
end main
```

```
Solve(float **A)
begin
  while (!done)
    diff = 0;
    for i=1 to n do
      for j = 1 to n do
        temp = A(i,j);
          A(i,j)=0.2*(A(i,j)+A(i,j-1)+
                      A(i,j+1)+A(i+1,j)+
                      A(i-1,j));
        diff += abs(A(i,j) - temp);
      end for
    end for
    if (diff / (n*n) < TOL) then done = 1;
  end while
  end Solve
```

Shared Memory Ocean

```
main()
begin

  p = NUM_PROCS();

  read(n);
  A = G_MALLOC(n * n);
  initialize(A);
  CREATE(p-1);
  Solve(A);
  WAIT_FOR_END(p-1);
end main
```

```
Solve(float **A)
begin
  pid = MY_PROC();
  start_row = 1 + (pid * n/p);
  end_row = start_row + n/p -1;
  while (!done)
    mydiff = diff = 0;
    BARRIER();
    for i=start_row to end_row do
      for j = 1 to n do
          temp = A(i,j);
          A(i,j)=0.2*(A(i,j)+A(i,j-1)+ A(i+1,j)+
                              A(i-1,j));
           mydiff += abs(A(i,j) - temp);
      end for
    end for
    LOCK(dlock); diff += mydiff; UNLOCK(dlock);
    BARRIER();
    if (diff / (n*n) < TOL) then done = 1;
  end while
end Solve
```

Message Passing Ocean

```
main()
begin
  p = NUM_PROCS();
  read(n);
  CREATE(p-1, solve);
  solve(A)
  WAIT_FOR_END(p-1);
end main
```

```
Solve(float **A)
begin
  myA = malloc(n/p + 2 by n array);
  while (!done)
    mydiff = 0;
    send (myA  1 row); send (myA n/p row);
    receive (myA 0 row); receive (myA n/p+1 row);
    for i= 1 to mymax row do
      for j = 1 to n do
        temp = myA(i,j);
        myA(i,j)=0.2*(myA(i,j)+myA(i,j-1)+myA(i-1,j)
                   +myA(i,j+1) + myA(I+1, j));
        mydiff += abs(myA(i,j) - temp);
      end for
    end for
```

Message Passing Ocean

```
If (pid != 0)
  send (mydiff)
  receive (done);
else
   for (p-1 times)
      receive (tempdiff);
      mydiff += tempdiff;
   end for
   if (mydiff/n*n < TOL) done = 1;
    for (p-1 procs)
      send (done);
    end for
  end if
end while
```

Ocean

- Think of how each core’s cache would look in the shared memory case
- Think of how each core’s messages would look in message passing case

Common Parallelization Strategies

- Think of parallelism at the algorithm level
- Possibly different
  - Best parallel algorithm
  - Best parallelization of best sequential algorithm
- Think of partitioning
  - Tasks? (do webpage serving and data-base lookup for different requests in parallel.)
  - Data? (Do similar operations on different data in parallel.)

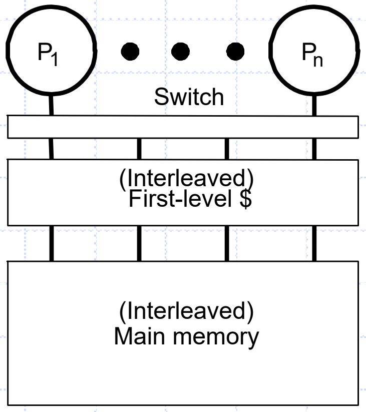
Programming models

- Pthreads – Shared memory processors **

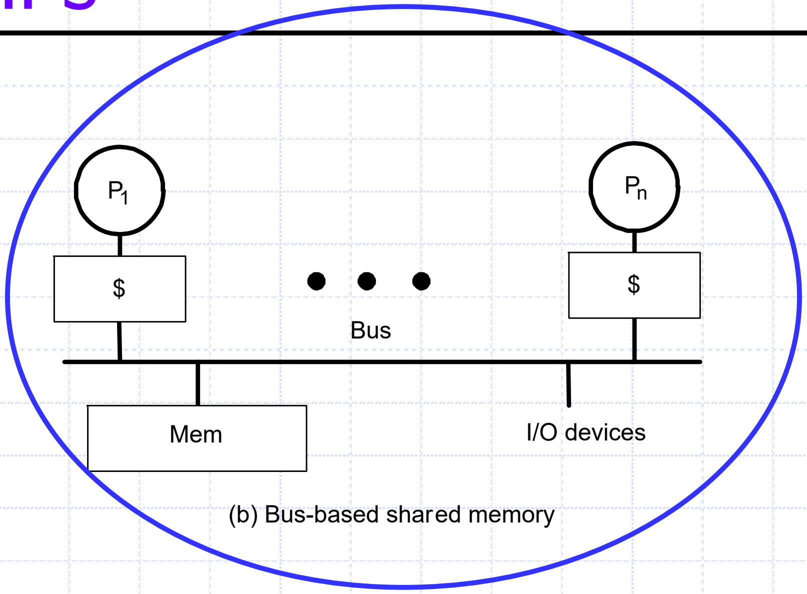
- MPI (mesg passing interface) – clusters **
- OpenMP – Shared memory
  - Marking loops as parallel loops

- Higher level primitives
  - Streaming, MapReduce, Parallel Recursion (Cilk C)
  - Customized to architecture
    - CUDA for nVidia GPGPUs

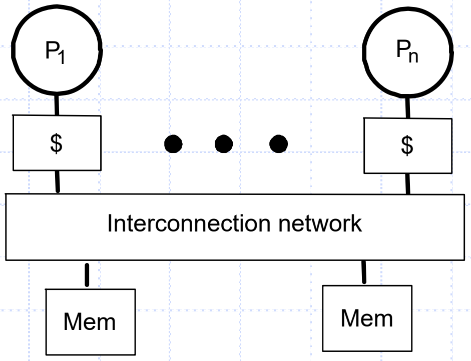
Shared address space MPs



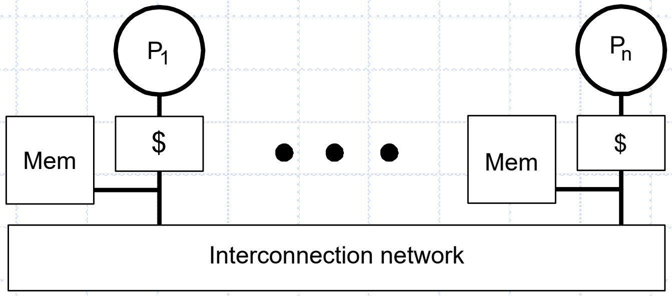
(a) Shared cache



(b) Bus-based shared memory



(c) Dancehall



(d) Distributed-memory

Terminology: Process vs. Thread

- "Heavyweight" processes
  - Different Thread of control
    - PC, registers, stack
  - Different Address space (page table, heap)
- "Lightweight" processes, a.k.a. "threads"
  - Different PC, register values, and stack allocation
    - "Context"
  - Same address space (thus same page table, same heap)
- Shared regions across heavyweight processes possible.

How do threads communicate?

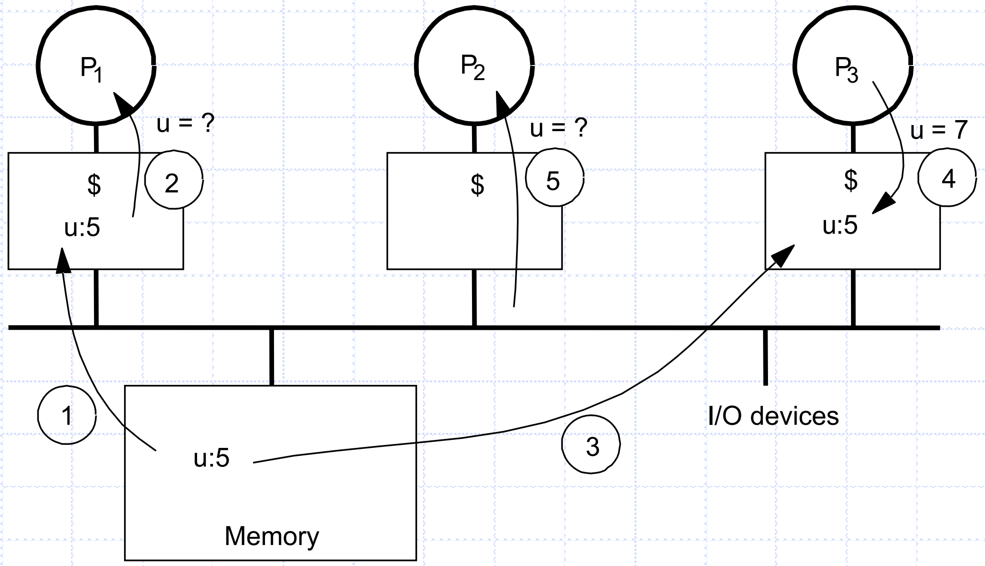
- Our focus:
  - Shared memory, bus-based systems
    - Shared address space
  - Communication via loads and stores

Several Interesting Problems

- Does everything work intuitively***
  - No.
  - Reason about coherence/consistency
- Sequencing
  - Synchronization

- Today: Cache Coherence

Example Cache Coherence Problem



- Is there a problem?
  - Replication + writes = coherence problem

Cache coherence

- Previous slide makes intuitive sense but EXACTLY what should coherence guarantee?
- Is the previous slide the only way to do business? Are there many options in what coherence can and cannot do?
- What are the rules of the game?

Coherence

- A = 0 (initial)

Thread 1	thread 2	thread 3
…	…	..
A = 10	…	..
..	A = 5	…
..	…	read A

- What should thread3 read?
  - No relative speeds assumed
- Does this mean ‘anything goes’ in shared memory?

Coherence

- A = 0 (initial)
- Thread 1 thread 2 thread 3
- … … ..
- A = 10 ..
- .. Read A wait thread2
- .. Signal thread 3 read A

- What should thread2 read?
- What should thread3 read if thread2 got 10?
  - If thread2 reads 0 then thread3 can read 0 or 10
- Do you still think ‘anything goes’ in shared memory?

Coherence

- Hard to distinguish between previous two slides
- Hard to track arbitrary synchronization among arbitrary threads in a large distributed system
  - Previous slide threads 2 and 3 synchronization affects thread1's write – hard for thread1 to track threads 2&3
- → the sanest option is that whenever a write is done it is done for ALL threads
- Whenever thread1's write to A is done it should be done for ALL threads
  - Ie the previous value is not visible to anyone
  - Called write atomicity and is provided by cache coherence

Coherence

- A = 0 (initial)
- Thread 1 thread 2 thread 3
- … … ..
-
- A = 10 … ..
- .. A = 5 …
- .. … read A

Given write atomicity, does this mean thread3 will always return 5?
(A) Yes
(B) No

Coherence

- A = 0 (initial)

Thread 1	thread 2	thread 3
…	…	..
A = 10	..	
..	Read A	wait thread2
..	Signal thread 3	read A

So in this example, what does write atomicity actually say?

Given write atomicity, does this mean thread3 will always read 10?
(A) Yes
(B) No

Given write atomicity, does this mean thread2 will always read 10?
(A) Yes
(B) No

Coherence

- Fuzzy idea
  - Necessary for correctness
- Precise definition
- For any memory location X
  - [within one thread – easy] Operations issued by any one thread occur in the order in which they issued
  - [across threads - hard] Value returned by read is the value written by last write (but “last” is relative!)
- Derivative properties
  - Write propagation : writes become visible to everyone (when?)
  - Write serialization/atomicity: writes to a single location seen in same order by every processor OR writes occur for everyone or no one (ie indivisibly)
- If you understand the above you will understand why coherence does what it does

Coherence

- cache coherence suggests absolute time scale
  - not necessary
- what is required is appearance of coherence
  - not instantaneous coherence
- Bart Simpson’s famous words -
  - “nobody saw me do it so I didn’t do it!”
- e.g. temporary incoherence between
  - writeback cache and memory ok

- Remember that no assumptions about threads’ relative speed

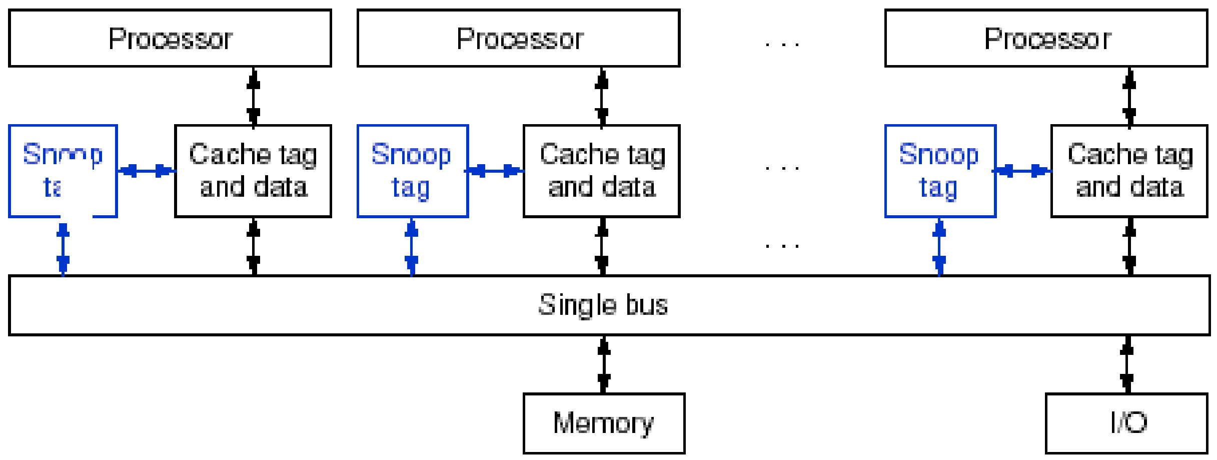
Coherence

- The ONLY thing coherence provides is write atomicity
  - ie when a write happens, it happens for all so nobody can see the previous value
  - even this suggests instantaneous and global update of all copies but cannot be implemented that way
  - writes take a non-zero time window to complete (ie not instantaneous) and after this window nobody can see the old value and during this window any access to the new value is blocked so nobody can see the new value WHILE the old value is not all gone
    - But during the window the old value may be visible because the write reaches different cores at different times

How to enforce coherence

- Correct actions:
  - On write, invalidate other copies
  - On write, update other copies
- Snooping:
  - Hits are guaranteed "not stale" → misses have to get latest data
  - Every node can "see" the bus
  - If all misses from all cores can be observed
    - Each can maintain coherence

Snoopy coherence



- Hits are guaranteed "not stale" → misses have to get latest data
- ALL cache misses go on the bus which snoops into ALL the other caches (ie look at tags)
- Cache controller updates state of blocks in response to processor and snoop events and generates bus actions

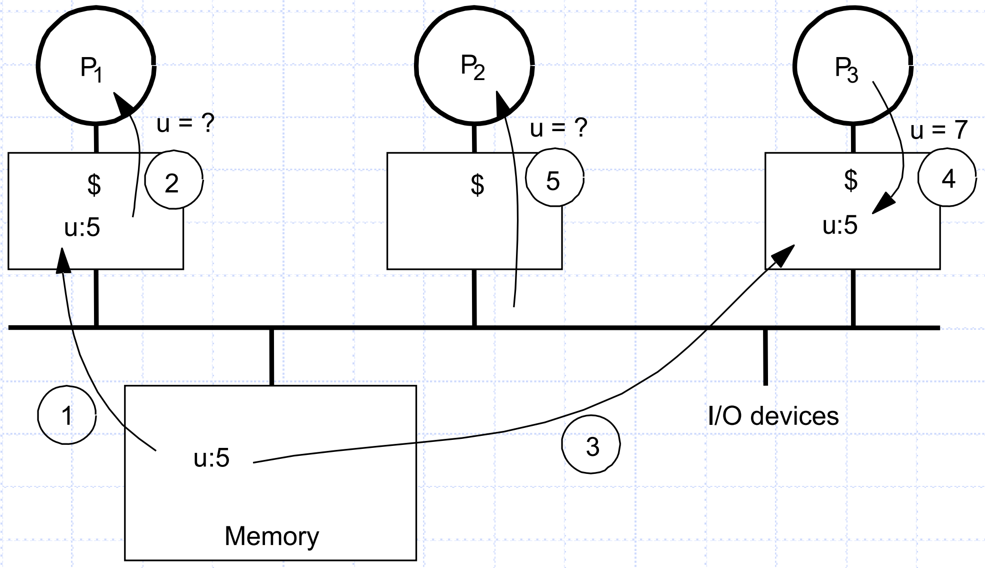
Snoopy Design Choices

- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - write-through vs. write-back
  - invalidate vs. update

A 3-State Write-Back Invalidation Protocol

- 3-State Protocol (MSI)
  - Modified
    - one cache has valid/latest copy
    - memory is stale
  - Shared
    - one or more caches have valid copy
  - Invalid
- Must invalidate ALL other copies before entering modified state
- Requires bus transaction (order to invalidate)

Example Cache Coherence Problem

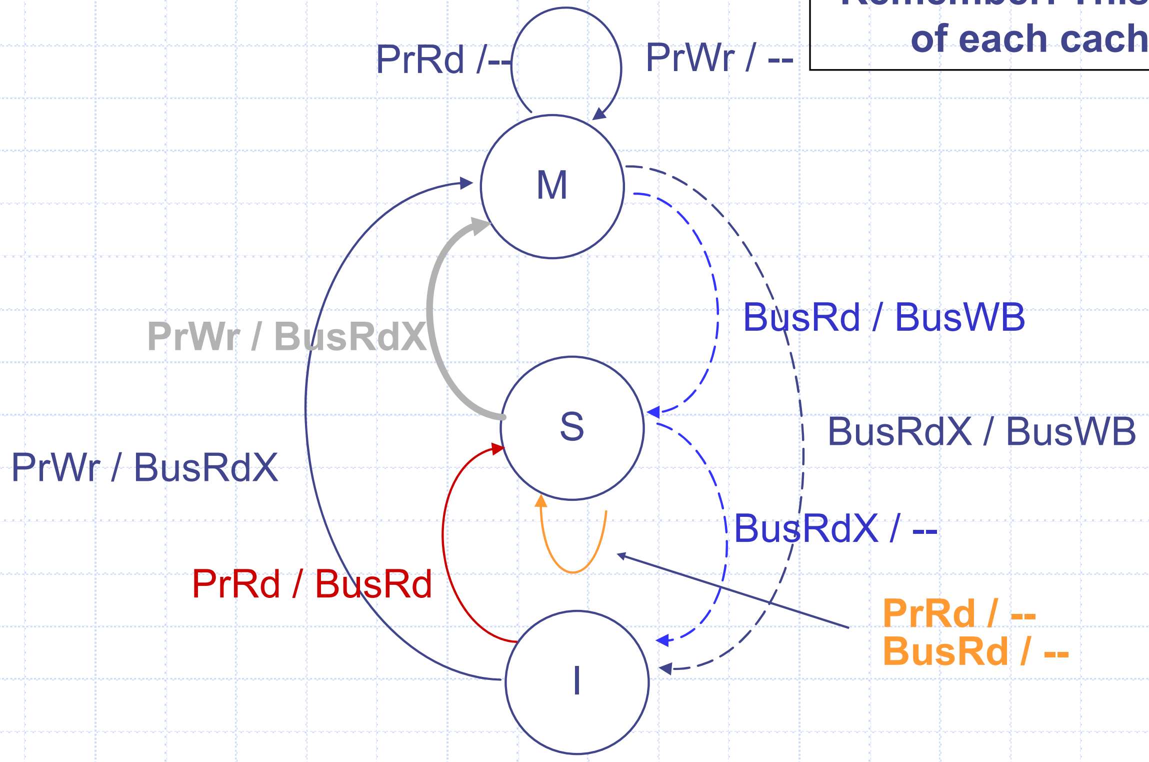


MSI Processor and Bus Actions

- Processor:
  - PrRd
  - PrWr
  - Writeback on replacement of modified block
- Bus
  - PrRD miss → Bus Read (BusRd) Read without intent to modify, data could come from memory or another cache
  - PrWr miss → Bus Read-Exclusive (BusRdX) Read with intent to modify, must invalidate all other caches copies
  - Writeback (BusWB) cache controller puts contents on bus and memory is updated (no atomicity issues)
  - Definition: cache-to-cache transfer occurs when another cache satisfies BusRd or BusRdX request
- Let's draw it!

MSI State Diagram

Remember: This is the state of each cache block.



A/B means action A causes transaction B on the bus

An example

Proc Action	P1 State	P2 state	P3 state	Bus Act	Data from
1. P1 read u	S	--	--	BusRd	Memory
2. P3 read u	S	--	S	BusRd	Memory
3. P3 write u	I	--	M	BusRdX	Memory (?)
4. P1 read u	S	--	S	BusRd	P3's cache
5. P2 read u	S	S	S	BusRd	Memory

- **Single writer, multiple reader protocol**
- **Why Modified to Shared?**
  - Why not to Invalid?
    - Give up as little as possible. Retain option to read.
  - Why throw away write permission?
    - Ownership
- **What if not in any cache?**
  - Read, followed by Write produces 2 bus transactions!

MSI is a distributed FSM

- Though the slide shows one state machine, on a bus snoop, MULTIPLE caches with the same block transit from one state to another
  - Eg upon busrdx, ALL copies other than the requestor transit to I and requestor goes to M
- The slide shows one state machine but really there is a state machine per block per cache
  - Of course the state machines in all the caches are identical (but the state may not be identical)
    - States of different blocks obviously different
    - But state of same block in different caches may or may not be identical

BusWB

- BusWB is for writing back replaced M blocks
  - BusWB can also occur as a reply to BusRD or BusRDX
- There is no snooping
- WBs are not writes – there is no atomicity requirement – why?
- You just write back the block to memory

Writes and invalidations

- So much talk about invalidations, but invalidation means many cache misses
- Then why bother with coherence or even caches if you are going to miss all the time
- Whoa!
- Think about shared memory Ocean.

Shared Memory Ocean

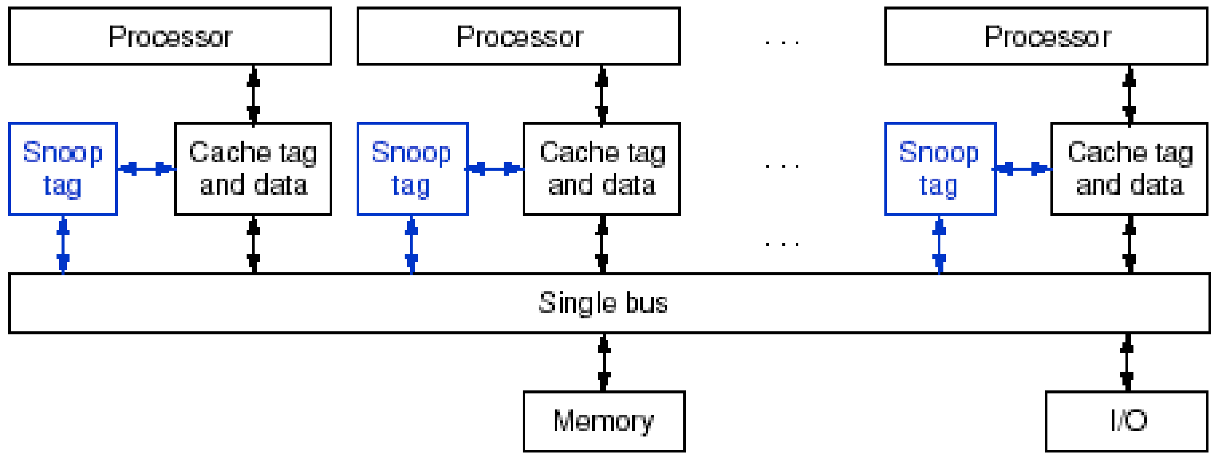
```
main()
begin

  p = NUM_PROCS();

  read(n);
  A = G_MALLOC(n * n);
  initialize(A);
  CREATE(p-1);
  Solve(A);
  WAIT_FOR_END(p-1);
end main
```

```
Solve(float **A)
begin
   pid = MY_PROC();
   start_row = 1 + (pid * n/p);
   end_row = start_row + n/p -1;
   while (!done)
      mydiff = diff = 0;
      BARRIER();
      for i=start_row to end_row do
        for j = 1 to n do
              temp = A(i,j);
              A(i,j)=0.2*(A(i,j)+A(i,j-1)+ A(i+1,j)+
                                          A(i-1,j));
               mydiff += abs(A(i,j) - temp);
        end for
      end for
      LOCK(dlock); diff += mydiff; UNLOCK(dlock);
      BARRIER();
      if (diff / (n*n) < TOL) then done = 1;
      BARRIER();
   end while
end Solve
```

Snoopy coherence



- If only one tag array then both processor and bus snoop contend for it – structural hazard
- → use a duplicate tag array so one tag array serves the processor and other serves snoops
  - Only tag array copies but not data array – why not?

Snoopy coherence

- But the tag copies have to be updated together
  - any state transitions due to processor AND bus have to update both tag copies
    - Eg processor reads (I $\rightarrow$ s) or writes (I$\rightarrow$ M or S $\rightarrow$ M)
    - Bus invalidations
  - Wait a minute – if both copies have to be updated then won't we have structural hazards?

Connect to previous egs

- Write atomicity
- Thread 1 thread2 thread3
  - Wr A rd A rd A
- Coherence does not mean instantaneous – wr A will take time to reach thread2 &3 and may not reach both threads at the same time even for snoopy bus coherence

Connect to previous egs

- The write will reach thread2 and 3 at different times (may be thread3 tag is busy) so thread3 could read old A for some time even if thread2 is invalidated (of course thread2 cannot read new A until write is done when all invalidations completes – the "non-zero time window" in previous slide)

Snoopy bus

- Coherence makes writes atomic
  - Not instantaneous but atomic
  - Hits guaranteed to have correct data
  - Misses go thru snoopy bus
  - But if two misses overlap write atomicity can be violated
    - Eg 2 write misses
  - Make the bus atomic – ie only one miss at a time
    - Each miss has to be complete before next miss starts
    - → if any memory access then bus is blocked for whole access (request to response)
    - Bus becomes bottleneck
    - Bus arbiter ensures one at a time - first come first served
    - What order are writes seen by everyone (everyone sees the same order)
    - In the order in which they get to the bus

Snoopy bus

- Real systems don’t use atomic buses
  - Either use a “split transaction bus” which splits a request and its response → other requests/responses can come in between
  - write atomicity complicated
    - Many small-scale multiprocessors – Sun Wildfire
    - Recent multicores – Intel Sandy Bridge, Ivy Bridge AMD Bulldozer
  - OR use no buses at all – called distributed shared memory where write atomicity is quite challenging
    - Large-scale shared memory multiprocessors
    - SGI Origin
    - ASCI Red, White, Blue machines from Intel, IBM, Cray
    - Take 666 next semester

Coherence: Performance Issues

- Access Pattern specific optimizations
- Update vs. Invalidate
- Cache Block size
  - Remember 3C classification
  - 4th C: " Coherence miss"
  - True vs. False Sharing
- Non-Atomic Bus

Why Synchronize?

```
lw $r1, ac-balance
add $r1, $r1, 100
sw $r1, ac-balance
```

```
lw $r1, ac-balance
add $r1, $r1, -50
sw $r1, ac-balance
```

- Two concurrent operations on the same bank account
  - Deposit $100 check
  - Withdraw $50 cash
- Correct result
  - ac-balance(new) = ac-balance(old)+50
- What can go wrong?

Synchronization

- Solution: Locks/Mutex
  - "critical section"
  - Only one person can have a lock
- Other synchronization primitives
- Barriers
  - Wait for everyone before proceeding
  - Can we just use a counter?
- Event notification
  - Producer consumer
  - Just use flags?
  - Consistency issues

```
LOCK ac-lock
lw $r1, ac-balance
add $r1, $r1, 100
sw $r1, ac-balance
UNLOCK ac-lock
```

```
LOCK ac-lock
lw $r1, ac-balance
add $r1, $r1, -50
sw $r1, ac-balance
UNLOCK ac-lock
```

How Not To Implement Locks

- LOCK
  while(**lock_variable** == 1); /* locked so wait */
    lock_variable = 1;
- UNLOCK
  lock_variable = 0;
- Implementation requires atomicity!
  - read and write of lock_variable must be atomic
  - Else two processes can acquire lock at same time
  - Another thread can come between read &write
  - Unlock is easy

Atomic Read-Modify-Write Operations

- Test&Set(r,x) -- r gets previous lock value

  ```
  r = m[x]
  m[x] = 1
  ```

  - r is register, m[x] is memory location x
  - if r is 1 then already locked else accessor gets the lock
- Swap(r,x) --- r gets previous lock value

  ```
  r = m[x], m[x] = r
  ```

- Compare&Swap(r1,r2,x) -- r2 gets previous lock value

  ```
  if (r1 == m[x]) then
      r2 = m[x], m[x] = r2
  ```

- Fetch&Op(r,x,op) -- r gets previous lock value

  ```
  r = m[x], m[x] = op(m[x])
  ```

- In ALL cases, the operations have to be ATOMIC in hardware

Correct lock implementation

LOCK

```
while (test&set(x) == 1);
```

UNLOCK

```
x = 0;
```

LL-SC based locks

- All primitives on previous slides need atomic Read-Modify-Write support in h/w
- Cannot easily do atomic Read-modify-write in h/w
  - R-M-W is reading followed by writing
  - one way would be to read (load) and then “negative acknowledge (nack)” invalidations from others so their write cannot complete while you try to write to complete your R-M-W
  - But what happens if two threads load at the same time?
  - They will both read and then permanently nack each other
  - Called a DEADLOCK!
  - Chances may be low but bad if it happens
  - Rare case has to be correct and it is not so rare if lock is heavily contended
  - Harder to do in a pipelined bus/distributed system

LL-SC based locks

- Different approach
  - Instead of forcing a core’s R-M-W to be atomic, detect non-atomicity between this core’s R and W (of lock)
  - if detected, retry R-M-W else see if got the lock
    - If did not get the lock, retry getting lock
  - How to detect non-atomicity
    - i.e., if some other core got the lock BETWEEN this core’s R and W of lock
    - lock = 0 if unlocked and 1 if locked ➔ to get the lock, a core MUST write to lock
    - Writes invalidate in coherence
    - So if another core invalidates lock between this core’s R and W of lock → non-atomicity

LL-SC based locks

- Invalidations already go to the cache but we want to know if the lock is invalidated between its R and W
- So need special instrs to catch such invalidations
- Instead of normal ld for R, use load-linked (**ll**) and instead of normal st for W, use store-conditional (**sc**)
- **ll** reads the lock as usual but also puts its address in a "link register" (next to cache – NOT CPU register)
- And **sc** first checks if the link register has not been invalidated (ie zeroed) and equals **sc**'s address
- if so means no invalidation between R and W and **sc** simply stores; else atomicity violation between this core's R and W, so **sc** does NOT store (ie fails) and instead sets its source (CPU) register to 0 to signal failure (ie non-atomicity)

LL-SC based locks

- Link register is "implicit" in **ll** and **sc**, so not specified

```
Test&set:  move $s2, 1   // 1 means locked
           ll $s1, X     // $s1 ← M[X] & linkreg ← X
           sc X,$s2      // $s2 originally contains 1
                            to signify locked
                         // if linkreg == X then
                                                M[X]← $s2
                         // else sc fails and $s2 ← 0
           beq $s2, $0 T&S     // retry if atomicity
                                                failure
           jr $31          //return $s1 the previous
                                          lock value
```

LL-SC based locks

- The returned previous lock value may already be locked in which case retry test&set in lock's "while loop" – this retry is not due to atomicity failure but due to not getting the lock

```
Unlock:  sw $0, X    // HUMBLE sw -  not fancy sc
         jr $31
```

LL-SC based locks

- Test&set does a strange thing: it ALWAYS writes a 1 whether a thread gets the lock or not
- Getting the lock is determined by the test&set return value which is the PREVIOUS value of the lock
    - If already locked t&s writes a 1 on top of 1 (previous value 1 which is returned)
    - If not t&s writes a 1 on top of 0 (previous value 0 which is returned)

Performance of Test & Set

LOCK

```
    while (test&set(x) == 1);
```

UNLOCK

```
    x = 0;
```

- High contention (many threads want lock while one thread holds it)
- All non-holders keep retrying T&S but each retry writes a 1 on top of 1 (**sc** within T&S is a write)
- Remember the CACHE!
- Each test&set writes and invalidates others – LOTS of cache misses while non-holders wait
- Several Optimizations

Test-and-Set Lock Performance

Processor 1	Processor 2	lock state		
		P1	P2	Latest State
`A0: t&s r1,0(&lock)`		M:1	I:	1
`A1: bnez r1,#A0`	`A0: t&s r1,0(&lock)`	I:	M:1	1
`A0: t&s r1,0(&lock)`	`A1: bnez r1,#A0`	M:1	I:	1
`A1: bnez r1,#A0`	`A0: t&s r1,0(&lock)`	I:	M:1	1
	`A1: bnez r1,#A0`	M:1	I:	1

- But performs poorly in doing so
  - Consider 3 processors rather than 2
  - Processor 0 (not shown) has the lock and is in the critical section
  - But what are processors 1 and 2 doing in the meantime?
    - Loops of `t&s`, each of which includes a `st`
    - Taking turns invalidating each others cache lines
    - Generating a ton of useless bus (network) traffic

Better Lock Implementations

- Two choices:
  - Don’t execute test&set so much
  - Spin without generating bus traffic
- Test&Set with Backoff
  - Insert delay between test&set operations (not too long)
  - Exponential seems good ($k*c^i$)
  - Not fair (a later contender may get lock before earlier one)
- Test-and-Test&Set
  - Intuition: No point in setting the lock with a 1 on top of 1 until we know that it’s unlocked
  - First keep testing (ie repeated NORMAL loads) until test succeeds (ie unlocked) then try test&set (for atomicity)
  - Testing (loading) gets cache hits without invalidations/misses until lock is unlocked (a store) which will invalidate all contenders and then one test$set will succeed
  - So no cache misses while non-holders wait
  - Still contention at unlock
  - Still not fair

Test-and-Test-and-Set Locks

- Solution: **test-and-test-and-set locks**
  - New acquire sequence

```
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1,r1
A3: t&s r1,0(&lock)
A4: bnez r1,A0
```

  - Within each loop iteration, before doing a `t&s`
    - Spin doing a simple test (`ld`) to see if lock value has changed
    - Only do a `t&s` (`st`) if lock is actually free
  - Processors can spin on a busy lock locally (in their own cache)
  - Less unnecessary bus traffic

Test-and-Test-and-Set Lock Performance

		lock state		
Processor 1	**Processor 2**	**P1**	**P2**	**Latest State**
`A0: ld r1,0(&lock)`		S:1	I:	1
`A1: bnez r1,A0`	`A0: ld r1,0(&lock)`	S:1	S:1	1
`A0: ld r1,0(&lock)`	`A1: bnez r1,A0`	S:1	S:1	1
`// lock released by processor 0`		I:	I:	0
`A0: ld r1,0(&lock)`	`A1: bnez r1,A0`	S:0	I:	0
`A1: bnez r1,A0`	`A0: ld r1,0(&lock)`	S:0	S:0	0
`A2: addi r1,1,r1`	`A1: bnez r1,A0`	S:0	S:0	0
`A3: t&s r1,(&lock)`	`A2: addi r1,1,r1`	M:1	I:	1
`A4: bnez r1,A0`	`A3: t&s r1,(&lock)`	I:	M:1	1
`CRITICAL_SECTION`	`A4: bnez r1,A0`	I:	M:1	1
	`A0: ld r1,0(&lock)`	I:	M:1	1
	`A1: bnez r1,A0`	I:	M:1	1

- Processor 0 releases lock, invalidates processors 1 and 2
- Processors 1 and 2 race to acquire, processor 1 wins

Queue Locks

- Test-and-test-and-set locks can still perform poorly
  - If lock is contended for by many processors
  - Lock release by one processor, creates "free-for-all" by others
  - – Network gets swamped with `t&s` requests

- **Queue lock**
  - **use two locks: a global master lock + queue of waiting threads and a private lock per waiting thread**
  - The first locker locks master lock
  - Later lockers join the queue and spin on their private lock
  - A lock releaser unlocks the next thread's private lock while later threads' private lock is undisturbed so they keep waiting
  - Greatly reduced bus traffic

Synchronization: Performance Issues

- Granularity
  - Lock single element vs. Lock whole array
  - Locking overhead vs. loss of concurrency
- Contention
  - TAS good at low contention
- Fairness
  - TAS offers no guarantees of fairness

Synchronization

- Deadlocks: Huge Correctness Issue
  - Previous deadlock was hardware deadlock here it is software deadlocks
- Races
- May not be visible to the programmer
  - Hard to debug (irreproducible)
- "Locks don't compose with themselves"
  - Other synchronization mechanisms have other problems
- We live with this problem
  - Transactional memory is a recent way to raise the abstraction level (666)

Memory Consistency

- **Cache coherence**
  - Creates globally uniform (consistent) view…
  - Of **a single memory location** (in other words: cache line)
  - Not enough
    - Cache lines A and B can be individually consistent…
    - But inconsistent with respect to each other

- **Memory consistency**
  - Creates globally uniform (consistent) view…
  - Of **ALL memory locations relative to each other**

- Who cares? Programmers
  - Globally inconsistent memory creates mystifying behavior

Why Coherence != Consistency

```
/* initial A = flag = 0 */
          P1                        P2
        A = 1;                 while (flag == 0); /*wait*/
         flag = 1;             print A;
```

- Intuition says print A = 1
- Coherence doesn't say anything, why?
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **Real systems** act in this strange manner

Why Coherence != Consistency

```
/* initial A = flag = 0 */
        P1                          P2
       A = 1;                 while (flag == 0); /*wait*/
       flag = 1;              print A;
```

Consider coalescing write buffer

- If multiple writes to same location, the buffer coalesces so only one write goes to memory
- A and flag written atomically – coherence correctly invalidates other copies
- BUT flag = 1 coalesced with previous write to flag so flag and A reordered in buffer so flag updates memory before A → print A = 0!

Why Coherence != Consistency

- Coherence concerns ONE location from all threads
- Consistency concerns ordering of ALL locations from one thread

- Coherence says WHAT action has to happen
- Consistency says WHEN action has to be complete
- Related through implementation but not the same

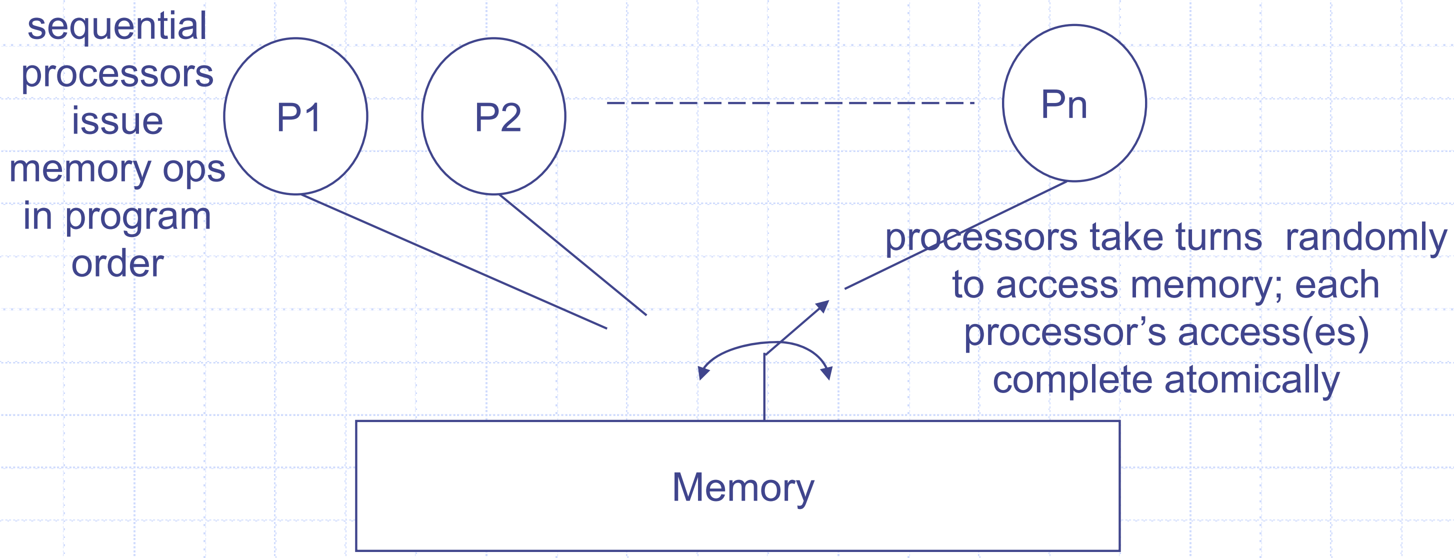
Consistency Model

- Is a contract between the system and the programmer
  - The system includes hardware,compiler, runtime, OS, etc

- Consistency models says what behavior is to be expected
- Programmer programs based on that behavior

- Sequential consistency is the simplest/most intuitive such behavior/model

Sequential Consistency Memory Model


sequential processors issue memory ops in program order
P1
P2
Pn
processors take turns randomly to access memory; each processor's access(es) complete atomically
Memory

- (1) Accesses from ONE thread in program order and (2) done atomically one at a time and multiple threads interleave randomly
- Each access is complete before next starts
- Intuition assumes this – hence print A = 1

Sequential Consistency (SC)

```
            A=flag=0;
Processor 0          Processor 1
A=1;                 while (!flag); // spin
flag=1;              print A;
```

- **Sequential consistency (SC)**
  - **Formal definition of memory view programmers expect**
  - Processors see their own loads and stores in program order
    - + Provided naturally, even with out-of-order execution
  - But also: processors see others' loads and stores in program order
  - And finally: all processors see same global load/store ordering
    - – Last two conditions not naturally enforced by coherence
- **Lamport definition**: multiprocessor ordering…
  - Corresponds to some sequential interleaving of uniprocessor orders
  - **I.e., indistinguishable from multi-programmed uni-processor**

Enforcing SC

- What does it take to enforce SC?
  - Definition: all loads/stores globally ordered
  - Translation: coherence events of all loads/stores globally ordered

- **When do coherence events happen naturally?**
  - On cache access
  - For stores: retirement → in-order → good
    - No write buffer? Yikes, but OK with write-back D$
  - For loads: execution → out-of-order → bad (in processor 1, ld A before ld flag!)
    - No out-of-order execution? Double yikes
- Is it true that multi-processors cannot be out-of-order?
  - No, come to 666

Sequential Consistency

- Programmers think in SC
  - i.e. preserve memory order
- Performance maximized when memory order is flexible
  - i.e., do not preserve memory order
- Architect's solution: Don't preserve order
  - Our problem is now the programmer's problem
  - Done today but causes programmer grief
- Can we get intuitive programming, correctness and fast performance?
- Several interesting approaches proposed
  - Parallel programming remains harder than sequential programming for this and other reasons