Multiprocessors and Multithreading

- why multiprocessors and/or multithreading?
  - what workloads do they run well
- types of MT/MP
  - Flynn's taxonomy
  - message passing vs. shared memory
  - newer stuff: CMP, SMT
- interconnection networks
- why caching shared memory is challenging
  - cache coherence, synchronization, and consistency

just the tip of the iceberg — take ECE 259/CPS 221 for more!

Readings

H+P

- chapter 6
  - 6.1–6.4, 6.7–6.9

Recent Research Papers

- SMT (Simultaneous Multithreading)
- Multiscalar

Threads, Processes, Processors, etc.

some terminology to keep straight

- process
- thread
- processor (we should know what this is!)
- thread context
- multithreaded processor
- multiprocessor

many issues are the same for MT and MP

- will discuss in terms of MPs, but will point out MT diffs

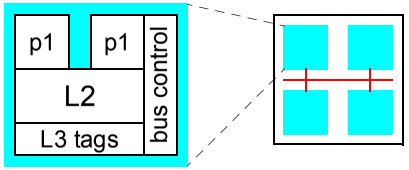
Why Parallel Processing?

- multiple processors working together, why?
- performance: break physical limits of uniprocessing
  - ILP (branch prediction, RAW dependences, etc.)
- cost and cost effectiveness
  - build big systems from *commodity parts* (ordinary uniprocessors)
  - enough transistors to make MT processors
  - enough transistors to make MP on a single chip (CMP)!
- other
  - smooth upgrade path (keep adding processors)
  - fault tolerance (one processor fails, still have P-1 working)
  - power-effective (remember Mudge's paper on power)

Chip Multiprocessors

trend today: multiprocessors on a single chip (CMPs)

- can't spend all of the transistors on just one processor
  - with limited ILP, single processor would not exploit it
- e.g., IBM POWER4
  - 1 chip contains: 2 1GHz processors, L2, L3 tags, interconnect
  - can connect 4 chips on 1 MCM to create 8 processor system
  - targets threaded server workloads



Multithreaded Processors

another trend: multithreaded processors

- processor utilization: IPC / processor width
  - decreases as processor width increases (~50% on 4 wide)
  - why? cache misses, branch mis-predictions, RAW dependences
- idea: two (or more) processes (threads) share one pipeline
- replicate process (thread) state
  - PC, register file, bpred history, page table pointer, etc.
- one copy of stateless (or naturally tagged) structures
  - caches, functional units, buses, etc.
- hardware thread switch must be fast
  - multiple on-chip contexts ⇒ no need to load from memory

Two Multithreading Paradigms

- coarse-grained
  - in-order processor with short pipeline
  - switch threads on long stalls (e.g., L2 cache misses)
  - instructions from one thread in stage per cycle
  - \+ threads don't interfere with each other much
  - – can't improve utilization on L1 misses, or branch mispredictions
  - e.g., IBM Northstar/Pulsar (2 threads)
- fine-grained: simultaneous multithreading (SMT)
  - out-of-order processor with deep pipeline
  - instructions from multiple threads in stage at same time
  - \+ improves utilization in all scenarios
  - – individual thread performances suffer due to interference
  - e.g., Pentium4 = 2 threads, Alpha 21464 (R.I.P.) = 4 threads

Why Parallel Processing Is Hard

in a word: *software*

- difficult to parallelize applications
  - – compiler parallelization hard
  - – by-hand parallelization maybe harder (very error prone, not fun)
- difficult to make parallel applications run fast
  - – communication very expensive (must be aware of it)
  - – synchronization very complicated

IT'S THE SOFTWARE, STUPID!

Amdahl's Law Revisited

speedup = 1/ [$frac_{parallel}$/$speedup_{parallel}$ + 1 – $frac_{parallel}$]

- example
  - achieve speedup of 80 using 100 processors
  - $\Rightarrow$ 80 = 1 / [$frac_{parallel}$/100 + 1 – $frac_{parallel}$]
  - $\Rightarrow$ $frac_{parallel}$ = 0.9975 $\Rightarrow$ only 0.25% work can be serial!
- good application domains for parallel processing
  - problems where parallel parts scale faster than serial parts
  - e.g., O(N^2) parallel vs. O(N) serial
  - interesting programs require communication between parallel parts
  - problems where computation scales faster than communication

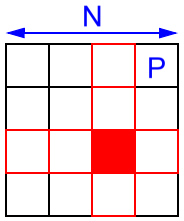
Application Domain 1: Parallel Programs

- true parallelism in one job
  - regular loop structures
  - data usually tightly shared
  - automatic parallelization
  - called "*data-level parallelism*"
  - can often exploit vectors as well

```
for (i=0;i<1000;i++){
    A[i] = B[i]*C[i];
}
```

- workloads
  - scientific simulation codes (e.g., FFT, weather, fluid dynamics, etc.)
  - was the dominant market segment of 10–15 years ago

Parallel Program Example: Matrix Multiply



- parameters
  - N = size of matrix (N*N)
  - P = number of processors
- growth functions
  - computation grows as f(N^3)
  - computation per processor grows as f(N^3/P)
  - data size grows as f(N^2)
  - data size per processor grows as f(N^2/P)
  - *communication* grows as f($N^2/P^{1/2}$)
  - computation/communication = f($N/P^{1/2}$)

Application Domain 2: Parallel Tasks

- parallel independent-but-similar tasks
  - irregular control structures
  - loosely shared data locked at different granularities
  - programmer defines & fine-tunes parallelism
  - cannot exploit vectors
  - called "*thread-level parallelism*" or "*throughput-oriented parallelism*"
- workload
  - transaction processing, OS, databases, web-servers
  - e.g., assign a thread to handle each request to server
  - dominant MP/MT market segment today (by far)

Parallel Task Example: Bank Database

- parameters
  - D = number of accounts
  - P = number of processors in central server
  - N = number of ATMs (parallel transactions)
- growth functions
  - computation: f(N)
  - computation per processor: f(N/P)
  - what is communication? have to *lock* records while changing them
  - communication: f(N)
  - – computation/communication: f(1)
  - + but no serial parts!

Taxonomy of Processors

Flynn Taxonomy [1966]

- not universal, but simple
- dimensions
  - instruction streams: single (SI) or multiple (MI)
  - data streams: single (SD) or multiple (MD)
- cross-product
  - SISD: uniprocessor (been there)
  - SIMD: vectors (skimmed that - refresher on next slide)
  - MISD: no practical examples (won't do that)
  - *MIMD*: multiprocessors + multithreading (doing it now)

SIMD Vector Architectures

different type of ISA that has vector instructions

- addv $v1, $v2, $v3 # v1 = v2 + v3
- vector registers are like V N-bit scalar registers
  - for example, 128 32-bit values
- can operate on all entries of vector at once
- many Cray systems were vector architectures

useful for "vector code"

for i=1 to 128 { sum [i] = addend1 [i] + addend2 [i]; }

SIMD vs. MIMD

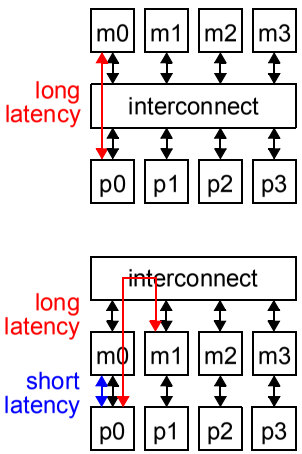
why are MPs (much) more common than vector processors?

- programming model flexibility
  - can simulate vectors with an MP, but not the other way around
  - dominant market segment cannot exploit vectors
- cost effectiveness
  - *commodity part*: high volume (translation: cheap) component
  - MPs made up of commodity parts (i.e., microprocessors)
  - can match size of MP to your budget
  - can't do this for a vector processor
- footnote: vectors are making a comeback
  - for graphics/multimedia applications (MMX, SSE, Tarantula)
  - NEC's EarthSimulator is an MP of vector processors

Taxonomy of Parallel (MIMD) Processors

- again, two dimensions
  - focuses on organization of main memory (shared vs. distributed)
- dimension I: appearance of memory to *hardware*
  - Q: is access to all memory uniform in latency?
  - *shared (UMA)*: yes ⇒ where you put data doesn't matter
  - *distributed (NUMA)*: no ⇒ where you put data <u>really</u> matters
- dimension II: appearance of memory to *software*
  - Q: can processors communicate via memory directly?
  - *shared (shared memory)*: yes ⇒ communicate via loads/stores
  - *distributed (message passing)*: no ⇒ communicate via messages
- dimensions are orthogonal
  - e.g., DSM: (physically) distributed (logically) shared memory

UMA vs. NUMA

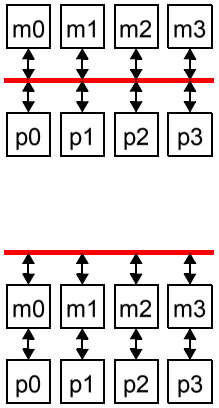


- *UMA: uniform memory access*
  - from p0, same latency to m0 as to m3
  - \+ data placement unimportant (software is easier)
  - – latency long, gets worse as system grows
  - – interconnect contention restricts bandwidth
  - typically used in small multiprocessors only
- *NUMA: non-uniform memory accesss*
  - from p0 faster to m0 (local) than m3 (non-local)
  - \+ low latency to local memory helps performance
  - – data placement important (software is harder)
  - \+ less contention (non-local only) ⇒ more scalable
  - typically used in larger multiprocessors

Interlude: What Is "Interconnect"?

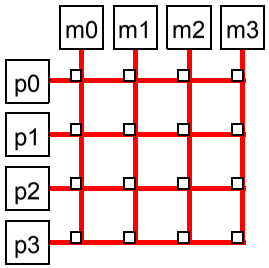
- connects processors/memories to each other
  - *direct*: endpoints (i.e., procs, mems) connected directly (e.g., mesh)
  - *indirect*: endpoints connected via switches/routers (e.g., tree)
- interconnect issues
  - *latency*: average latency most important (locality optimizations?)
  - *bandwidth*: per processor (also, bisection bandwidth)
  - *cost*: # wires, # switches, # ports per switch
  - *scalability*: how latency, bandwidth, cost grow with # processors (P)
- we're mainly concerned with *interconnect topology*
- can have separate interconnects for addresses and data

Interconnect 1: Bus



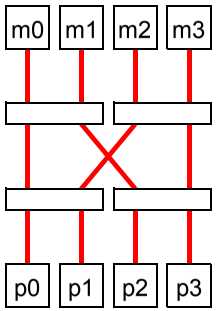
- direct interconnect
- \+ cost
  - f(1) wires
- \+ latency: f(1)
  - no neighbor/locality optimization
- – bandwidth: *not scalable at all*, f(1/P)
  - only used in small systems (P <= 8)
- \+ capable of *ordered broadcast*
  - incapable of anything else
- new: logical buses w/point-to-point links
  - tree = logical bus, if all messages go to root
  - e.g., Sun UltraEnterprise E10000

Interconnect 2: Crossbar Switch



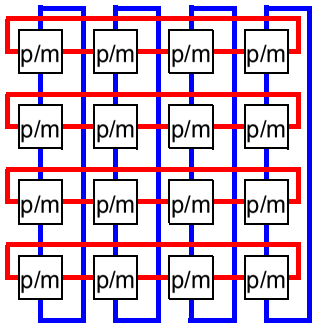
- indirect interconnect
- \+ latency: f(1)
  - no locality/neighbor optimizations
- \+ bandwidth: f(1)
- – cost
  - f(2P) wires
  - f(P^2) switches
  - 4 wires per switch

Interconnect 3: Multistage Network



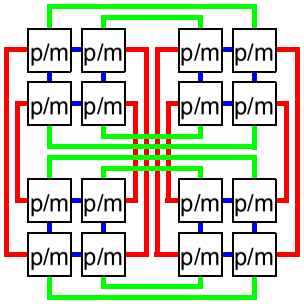
- indirect interconnect
  - routing done by address bit decoding
  - *k*: switch arity (# inputs and outputs per switch)
  - *d*: number of network stages = $\log_k P$
- \+ cost
  - f(d*P/k) switches
  - f(P*d) wires
  - f(k) wires per switch
- \+ latency: f(d)
- \+ bandwidth: f(1)
- commonly used in large UMA systems
  - a.k.a. butterfly, banyan, omega

Interconnect 4: 2D Torus



- direct interconnect
  - no dedicated switches
- \+ latency: f($P^{1/2}$)
  - locality/neighbor optimization
- \+ bandwidth: f(1), scales with P
- \+ cost
  - f(2P) wires
  - 4 wires per switch
- good scalability ⇒ widely used
  - variants: 3D, mesh (no "wraparound")
- e.g., Alpha 21364-based MPs

Interconnect 5: Hypercube



- direct interconnect
  - *k*: arity (# nodes per dimension)
  - *d*: dimension = $\log_k P$
  - in figure: P = 16, k = 2, d = 4
- \+ latency: f(k/d)
  - locality/neighbor optimized
- \+ bandwidth: f((k–1)*d)
- – cost
  - f((k–1)*d*P) wires
  - f((k–1)*d) wires per switch
- good scalability, expensive switches
  - – switch changes as nodes are added

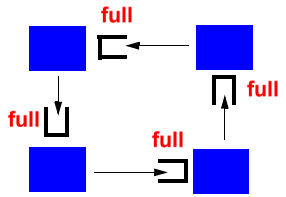
Interconnect Routing

- *store-and-forward* routing
  - switch buffers entire message before passing it on
  - latency = [(message length / bandwidth) + fixed overhead] * # hops
- *wormhole* routing
  - pipeline message through interconnect
  - switch passes message on before completely arrives
  - latency = (message length / bandwidth) + (fixed overhead * # hops)
  - + no buffering needed at switch
  - + latency (relatively) independent of number of intermediate hops

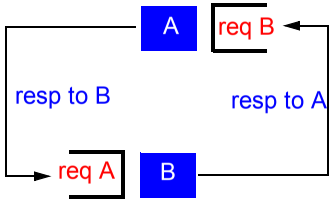
Avoiding Deadlock in Interconnect

two types of deadlock

- routing deadlock
  - circular dependence on buffers
- solutions
  - routing restrictions (turn model)
  - virtual channels



- request/response deadlock
  - circular dependence on messages
- solutions
  - separate networks
  - virtual networks



Shared Memory vs. Message Passing

MIMD dimension II: appearance of address space to software

- *message passing* (multicomputers, clusters)
  - each processor has its own address space (and unique processor #)
  - processors send (receive) messages to (from) each other
  - communication pattern explicit and precise (only way)
  - used for scientific codes (explicit communication patterns)
  - message passing systems: PVM, MPI
  - + simple hardware
  - – difficult programming model (in general)

Shared Memory vs. Message Passing

- *shared memory* (multiprocessors)
  - one shared address space
  - processors use conventional loads/stores to access shared data
  - communication can be complex/dynamic
  - + simpler programming model (compatible with uniprocessors)
  - – but with its own nasties (e.g., synchronization)
  - – more complex hardware... (we'll see soon)
  - + but more room for hardware optimization
- aside: software shared virtual memory (SVM) exists

(Not Too) Recent Parallel Systems

machine	communication	interconnect	#cpus	remote latency (us)
SPARCcenter	shared memory	bus	<=20	1
SGI Challenge	shared memory	bus	<= 32	1
Cray T3D	shared memory (nc)	3D torus	64-1024	1
Convex SPP	shared memory	X-bar/ring	8-64	2
KSR-1	shared memory	bus/ring	32	2-6
TMC CM-5	messages	fat tree	64-1024	10
Intel Paragon	messages	2-d mesh	32-2048	10-30
IBM SP-2	messages	multistage	32-256	30-100

we will concentrate on shared memory systems

- more hardware oriented
- market is going this way
  - speaking of which...

Multiprocessor Industry Trends

- shared memory
  - easier, more dynamic program model (it IS the software, stupid!)
  - can do more to optimize the hardware
- small-to-medium size UMA systems (2–8 processors)
  - processors + memory + switch on single board (e.g., quad Pentium)
  - same thing on a single chip (e.g., IBM POWER4)
  - commodity part of the future (present?)
  - *glueless MP*: slap these together and MP just works! e.g., Opteron
- larger NUMA systems built from smaller (N)UMA systems
  - exploit commodity nature of small systems
  - use commodity interconnect (e.g., gigabit Ethernet, Myrinet)
  - called NUMA clusters

Caching Shared Memory

- three issues
  - cache coherence
  - synchronization
  - memory consistency model
- not completely unrelated to each other
- not issues for message passing machines
  - why not?

Cache (In)Coherence

- most common cause: sharing of writeable data
  - example

```
processor 0  processor 1  correct value of A is in..
-----------  -----------  --------------------------
                          memory
read A                    memory, p0 cache
             read A       memory, p0 cache, p1 cache
write A                   p0 cache, memory (if wthru)
             read A       p1 gets stale value on hit
```

- other causes
  - process migration (even if jobs are independent)
  - I/O (can be fixed by OS cache flushes)

Solutions to Coherence Problem

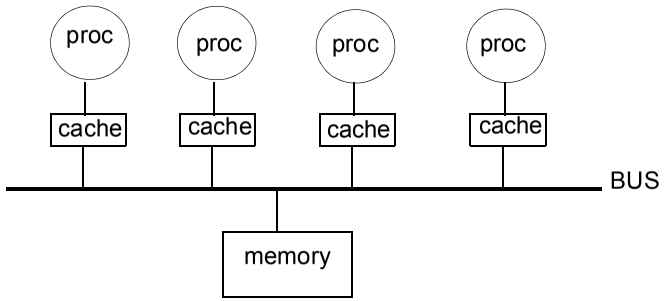
- no caches
  - not a good solution - caches are important!
- make shared-data non-cacheable
  - + simplest software solution
  - – low performance if a lot of data is shared
- software flush at strategic times: e.g., after critical sections
  - + relatively simple
  - – low performance if synchronization is frequent
- hardware *cache coherence*
  - make memory and caches coherent (consistent) with each other
  - in other words: let memory and other processors see writes
  - invisible to software

Cache Coherence Protocols

- absolute coherence
  - all copies of each block have same data at all times
  - not necessary
- what is required is *appearance of absolute coherence*
  - temporary incoherence is OK (e.g., write-back cache)
  - as long as all loads get "correct" values
- *cache coherence protocol*: FSM that runs at every cache
  - and usually a FSM at every memory, too
- two kinds of protocols: depends on how writes handled
  - *invalidate protocol*: invalidate copies in other caches
  - *update protocol*: update copies in other caches

Bus-Based Protocols (Snooping)

- bus-based cache coherence protocol (snooping)
  - ALL caches/memories see and react to ALL bus events
  - protocol relies on global visibility of requests (ordered broadcast)
  - owner (either proc or mem) responds to request with data

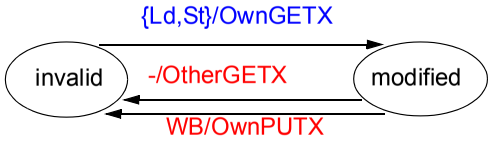


Snooping Protocol Events

- requests from proc/cache to cache coherence controller
  - load (Ld)
  - store (St)
  - writeback (WB)
- bus events (= cache coherence transactions)
  - GetShared (GETS) - broadcast request for read-only data
  - GetExclusive (GETX) - broadcast request for read-write data
  - PutExclusive (PUTX) - broadcast request to write data back to mem
- coherence transactions on bus can be from self or others
- we'll assume atomic bus transactions
  - thus, we have atomic cache coherence transactions

Two-State (MI) Invalidate Protocol

- two states
  - *invalid*: either don't have block or have it but not allowed to use it
  - *modified*: have block with read-write access
- problem
  - – block can be in only one cache at a time
  - – not efficient, especially if data is only being read



notation "a/b":

a = proc request

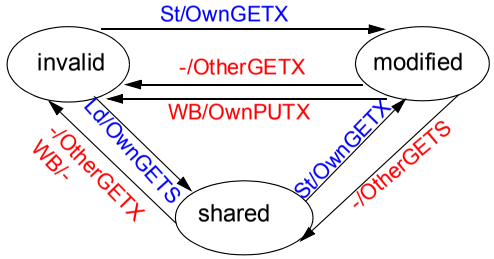
b = coherence transaction

blue = upgrade

red = downgrade

Three-State (MSI) Invalidate Protocol

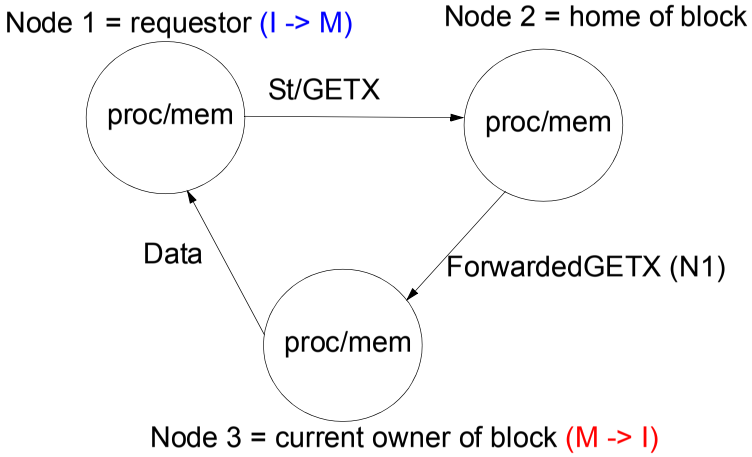
- three states
  - *idea: add new "read-only" state (shared) - allows multiple readers!*
  - *invalid*
  - *modified*: have block with read-write access
  - *shared*: have block with read-only access



Scalable Coherence Protocols: Directories

- – bus-based protocols (i.e., broadcast) are not scalable!
  - not enough bus b/w for everyone's coherence traffic
  - not enough processor snooping b/w to handle everyone's traffic
- *directories*: scalable cache coherence for large MPs
  - each memory entry (cache line) has a bit vector (1 bit per processor)
  - bit vector tracks which processors have cached copies of line
  - send all requests to directory at home memory
  - if no other cached copies, memory is owner and returns data
  - otherwise, memory forwards request to current owner processor
  - + low b/w consumption (communicate only with processors that care)
  - + works with general interconnect (bus not needed)
  - – longer latency (3-hop transactions: p0 ⇒ directory ⇒ p1 ⇒ p0)

Directory Protocol in Action (MI)



Coherence Protocols: Performance

- 3C miss model ⇒ 4C miss model
  - capacity, compulsory, conflict
  - *coherence*: additional misses due to coherence protocol
  - complicates uniprocessor cache analysis
- as processors are added
  - – coherence misses increase (more communication)
- as cache size is increased
  - + capacity misses decrease
  - – coherence misses increase (more shared data is cached)
- as block size is increased
  - – coherence misses increase (false sharing)
  - *false sharing*: sharing of different data in same cache line

Synchronization

- *synchronization*: important issue for shared memory
  - regulates access to shared data
  - e.g., semaphore, monitor, critical section (s/w constructs)
  - synchronization primitive: *lock*

```
acquire(lock);      // while (lock != 0); lock = 1;
critical section;
release(lock);      // lock = 0;


0: ldw r1, lock     // wait for lock to be free
1: bnez r1, #0
2: stw #1, lock     // acquire lock
...                 // critical section
9: stw #0, lock     // release lock
```


Implementing Locks

- lock implementation from previous slide
  - called “spin lock”
  - – doesn’t actually work in all situations (=incorrect!)

```
processor 0     processor 1
0: ldw r1,lock
1: bnez r1,#0                   // p0 sees lock free
                0: ldw r1,lock
                1: bnez r1,#0   // p1 sees lock free
2: stw #1,lock                  // p0 acquires lock
                2: stw #1,lock  // p1 acquires lock
...                             // p0 AND p1 in
                ...             // critical section
...                             // TOGETHER
9: stw #0,lock
```


Implementing Locks

problem: acquire sequence (load-test-store) is *not atomic*

- option I: implement sequence in kernel
  - kernel can control interleaving by suppressing interrupts
  - + implementation works
  - – hugely expensive for common case (lock is free)

```
ACQUIRE_LOCK:                   0: syscall ACQUIRE_LOCK
10: enable interrupts           1: ...
11: disable interrupts          2: stw #0, lock
12: ldw r1,lock
13: bnez r1,#10
14: stw #1,lock
15: enable interrupts
16: ret
```

Implementing Locks

- option II: ISA provides an *atomic lock-acquire* operation
  - load+check+store in one instruction (uninterruptible by definition)
  - e.g., test&set instruction (t&s) (aka fetch&add, swap)

```
t&s r1,lock     // ldw r1,lock; stw #1,lock;

0: t&s r1,lock
1: bnez r1, #0
2: ...
3: stw r1, #0
```

  - BTW, lock-release is already atomic

a lot of work has gone into making synchronization fast + cheap

Correctness: Memory Ordering

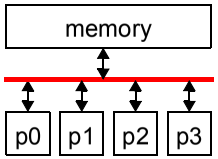
memory updates may become re-ordered by the memory system

- example

```
processor 0          processor 1
A = 0                B = 0
A = 1                B = 1
L1: if (B == 0)      L2: if (A == 0)
critical section     critical section
```

  - intuitively impossible for both processors to be in critical section
  - BUT can happen if memory operations are reordered
  - coherence: A's must be same, B's must be same EVENTUALLY
  - says nothing about relative timing of A's and B's coherence
  - this is specified by the *memory consistency model*

Memory Ordering: Sequential Consistency



"system is *sequentially consistent* if the result of ANY execution is the same as if the operations of all processors were executed in SOME sequential order and the operations of each individual processor appear in this sequence in program order" [Lamport]

- *sequential consistency (SC)*
  - all loads and stores in order
  - \+ simple for programmer
  - – not much room for hardware (or software) optimization
  - example works if system obeys *sequential consistency (SC)*

Weak(er) Consistency Models

- observation: SC needed *only* for lock variables
  - other variables?
  - either in critical section (no parallel access)
  - or not shared
- weaker consistency: can delay/reorder loads and stores
  - \+ more room for hardware optimization
  - – somewhat trickier programming model?
  - e.g., Intel IA-32: processor consistency (PC)
  - e.g., Sun: total store order (TSO) is very similar to PC
  - e.g., Alpha: weak ordering (WO)
  - e.g., Intel IA-64: release consistency (RC)

Summary

- multiprocessors and multithreaded processors
  - workloads: parallel programs and parallel tasks
  - UMA vs. NUMA
  - trends: multithreaded processors, CMP
- interconnect
  - direct vs. indirect, store-and-forward vs. wormhole, topologies
- interprocess communication
  - message passing vs. shared memory
- shared memory
  - cache coherence: bus-based (2-state vs. 3-state), directory-based
  - synchronization
  - memory consistency