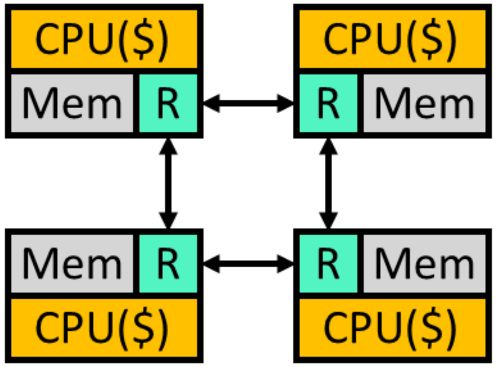
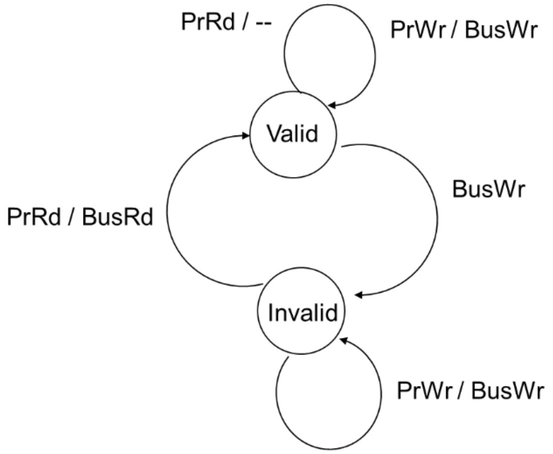
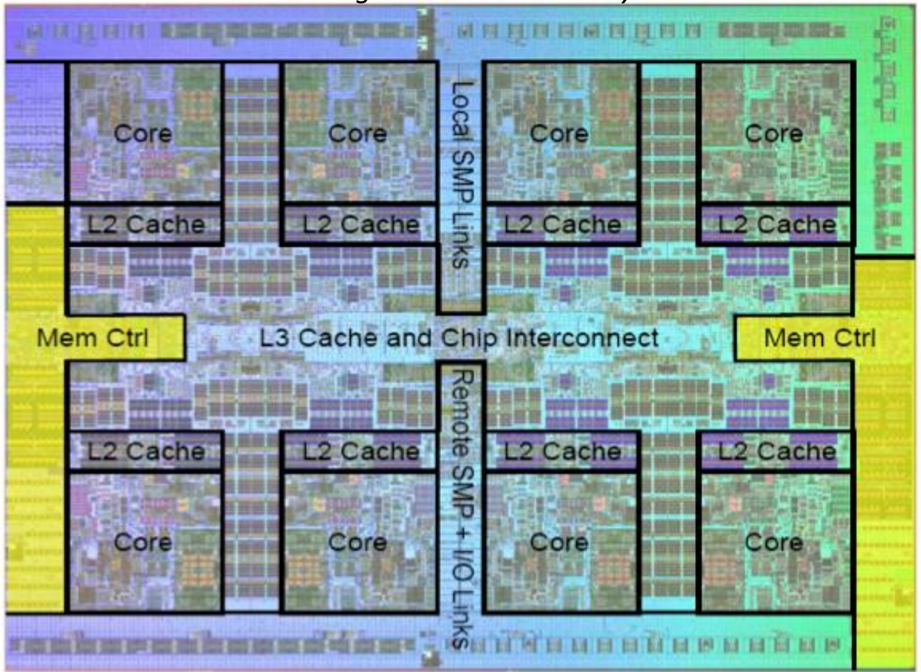


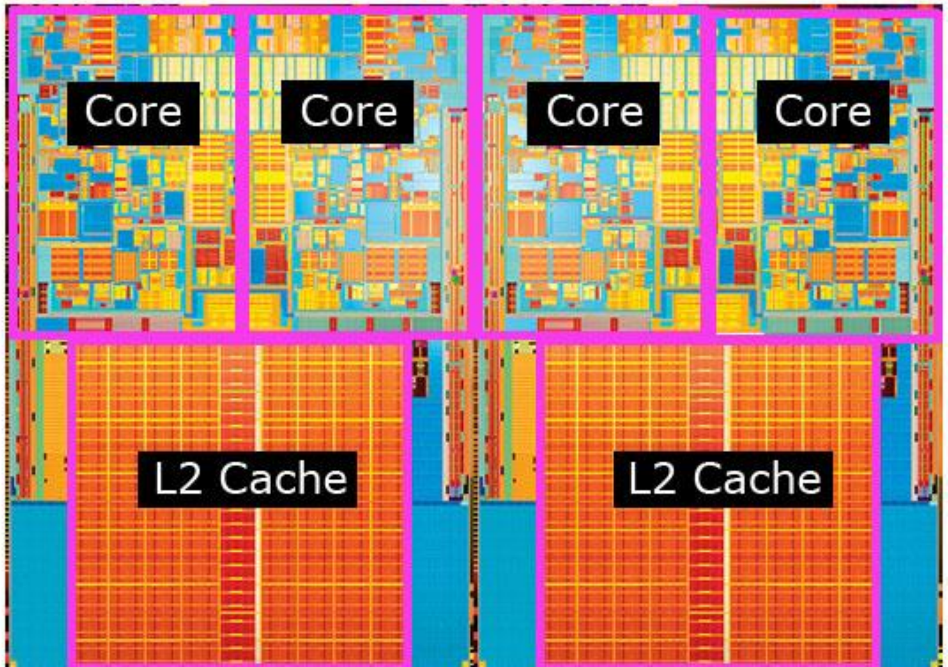


Multiprocessors continued

IBM's Power7 with eight cores and 32 Mbytes eDRAM



Quad-core Kentsfield package



Quick overview

- Why do we have multi-processors?
- What type of programs will work well? What type work poorly?

Overview of today's lecture

- *Performance expectations*
  - Amdahl's law.
- *Review*
  - Shared memory vs. message passing
  - Interconnection networks
  - Thread-level parallelism
- *Context*
  - Bandwidth
- *Coherence*
  - Directory-based consistency protocols
- *Consistency*
  - Sequential (strong) consistency, weak consistency

Review

Let's (re)start at the top: Performance Expectations

- ***Amdahl's Law:***
  - If a fraction P of your program can be sped up by a factor of S, then your performance is:

$$\frac{1}{(1-P)+\frac{P}{S}}$$

  - So if P=0.3 and S=2 we get (1/(.7+.15)) 1/.85 which is about 1.17.
- Question:
  - If you have a program in which 10% can't be done in parallel (i.e. must be serial) and you've got 64 processors, what's the best speed up you could hope for?

Review: Thread-Level Parallelism

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)
{
   accts[id].bal -= amt;
   spew_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 spew_cash
6: ... ... ...
```

- **Thread-level parallelism (TLP)**
  - *Collection of asynchronous tasks*: not started and stopped together
  - Data shared loosely, dynamically
- Example: database/web server (each query is a thread)
  - **`accts`** is **shared**, *can't register allocate*[1] even if it were scalar
  - **`id`** and **`amt`** are private variables, register allocated to **`r1`**, **`r2`**

[1]Keyword in C is "*volatile*"

Thread level parallelism (TLP)

- We exploited parallelism at the instruction level (ILP)
  - Hardware can’t find TLP.
    - Complier/programmer needs to find it (we lack the window size)
    - Compilers getting better at this.
  - But hardware can take advantage of TLP
    - Run on multiple cores
    - If data is shared, provide rules to the software about what it can assume about shared data.

Review: Why Shared Memory?

Pluses:

- For applications looks like multitasking uniprocessor
- For OS only evolutionary extensions required
- Easy to do communication without OS
- Software can worry about correctness first then performance

Minuses:

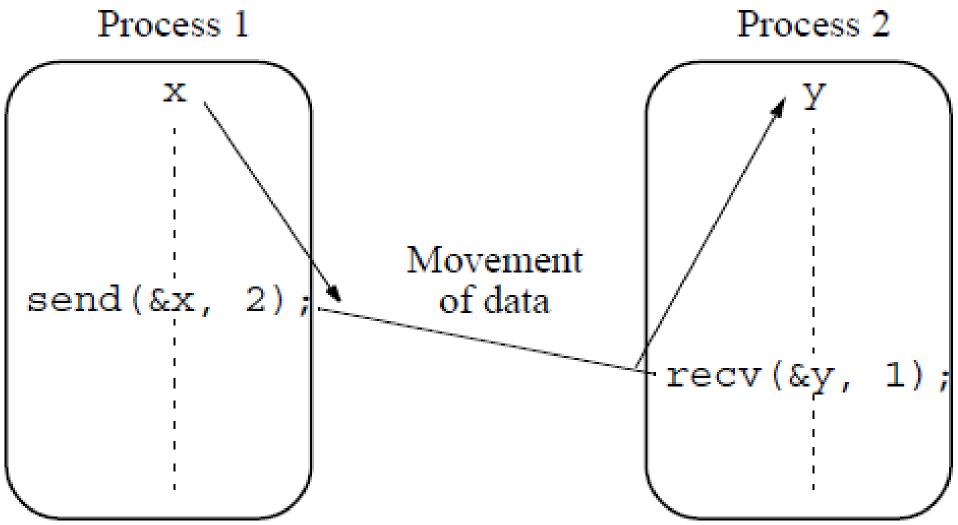
- Proper synchronization is complex
- Communication is implicit so harder to optimize
- Hardware designers must implement

Result:

- Traditionally bus-based Symmetric Multiprocessors (SMPs), and now the CMPs are the most success parallel machines ever
- And the first with multi-billion-dollar markets

Alternative to shared memory?

- Explicit message passing
  - Rather difficult to code
- We'll not be doing anything with it
  - But while not a 470 topic, it is important.
    - Graduate level parallel programming class covers this in detail.



But shared memory systems have their own problems

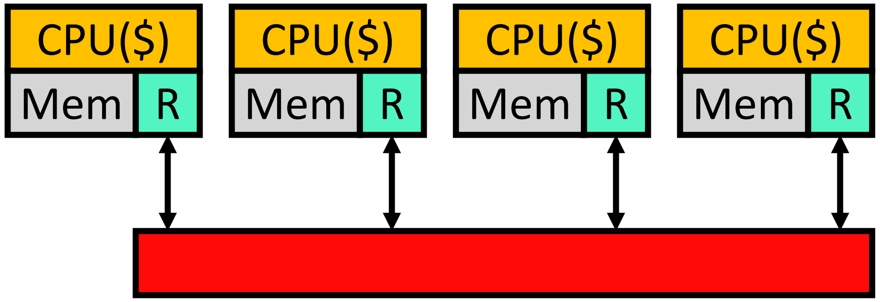
- Two in particular
  - **Cache coherence**
  - Memory consistency model
- Different solutions depending on interconnect
  - So let's do interconnect now, and jump back to these two later.

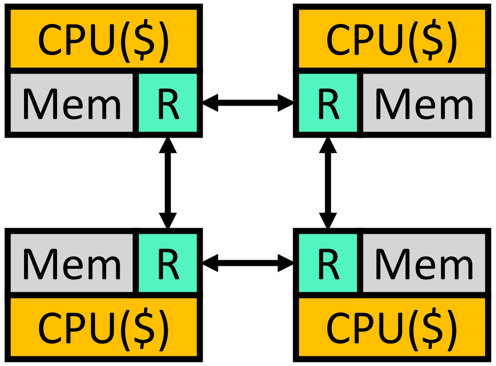
Review:
Interconnect

- There are many ways to connect processors.
  - Shared network
    - Bus-based
    - Crossbar
  - Point-to-point
    - More topologies than I want to think about
      - Mesh (in any amount of dimensionality)
      - Torus (in any amount of dimensionality)
      - nCube
      - Tree
      - etc. etc. etc. etc.

Shared vs. Point-to-Point Networks

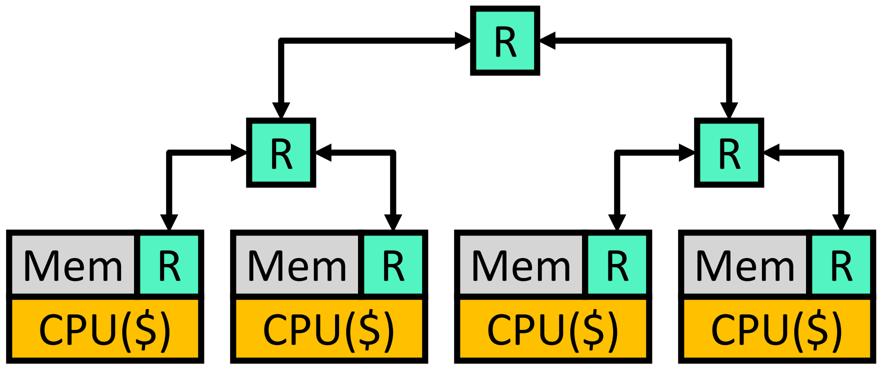
- **Shared network**: e.g., bus (left)
  - + Low latency
  - – Low bandwidth: doesn’t scale beyond ~8 processors
  - + Shared property simplifies cache coherence protocols (later)
- **Point-to-point network**: e.g., mesh or ring (right)
  - – Longer latency: may need multiple “hops” to communicate
  - + Higher bandwidth: scales to 1000s of processors
  - – Cache coherence protocols are complex

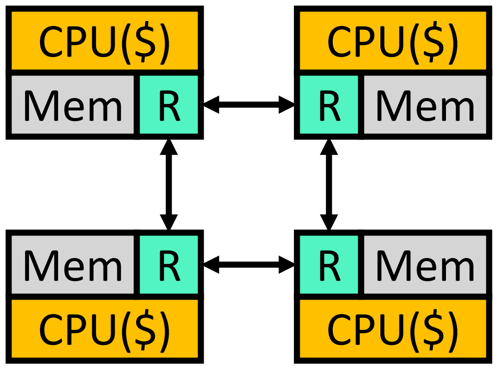




Organizing Point-To-Point Networks

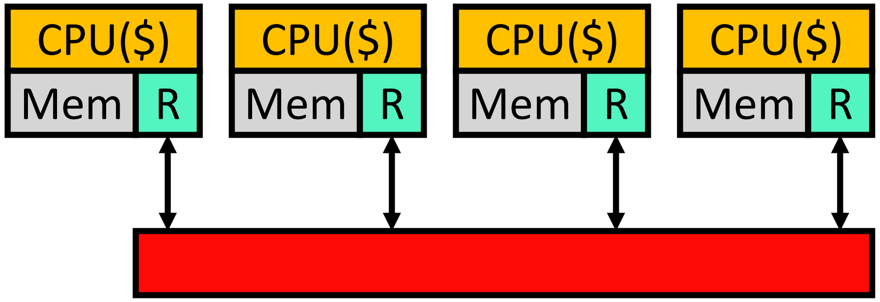
- **Network topology**: organization of network
  - Tradeoff performance (connectivity, latency, bandwidth) ↔ cost
- Router chips
  - Networks that require separate router chips are **indirect**
  - Networks that use processor/memory/router packages are **direct**
    - + Fewer components, "Glueless MP"
- Point-to-point network examples
  - Indirect tree (left)
  - Direct mesh or ring (right)



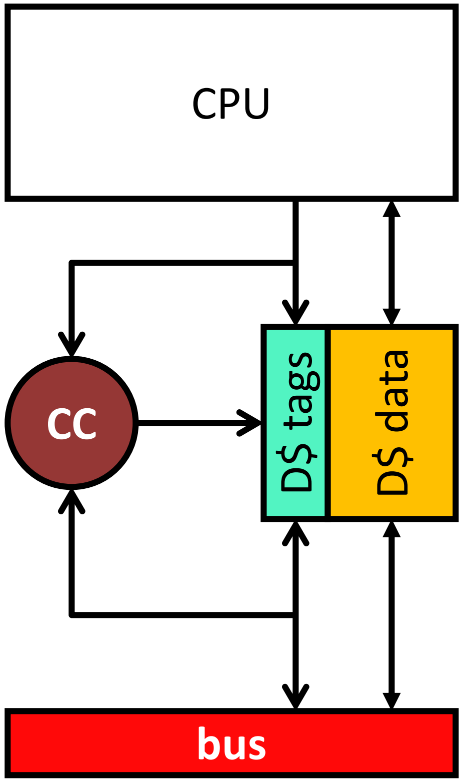


Context:
Bandwidth

- As in the uniprocessor we need caches
  - To reduce average memory latency.
- But recall caches are also important for reducing bandwidth.
  - And now we've got (say) 4x as many requests
    - And the ***total*** amount of bandwidth is shrinking (why?)
  - We've probably got a bandwidth problem.
    - It's important that caches work well!



Hardware Cache Coherence



- **Coherence controller**:
  - Examines bus traffic (addresses and data)
  - Executes **coherence protocol**
    - What to do with local copy when you see different things happening on bus

Scalability problems of Snoopy Coherence

- Prohibitive **bus bandwidth**
  - Required bandwidth grows with # CPUS…
  - … but available BW per bus is fixed
  - Adding busses makes serialization/ordering hard
- Prohibitive **processor snooping bandwidth**
  - All caches do tag lookup when ANY processor accesses memory
  - Inclusion limits this to L2, but still lots of lookups

- **Upshot**: bus-based coherence doesn’t scale beyond 8–16 CPUs

Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

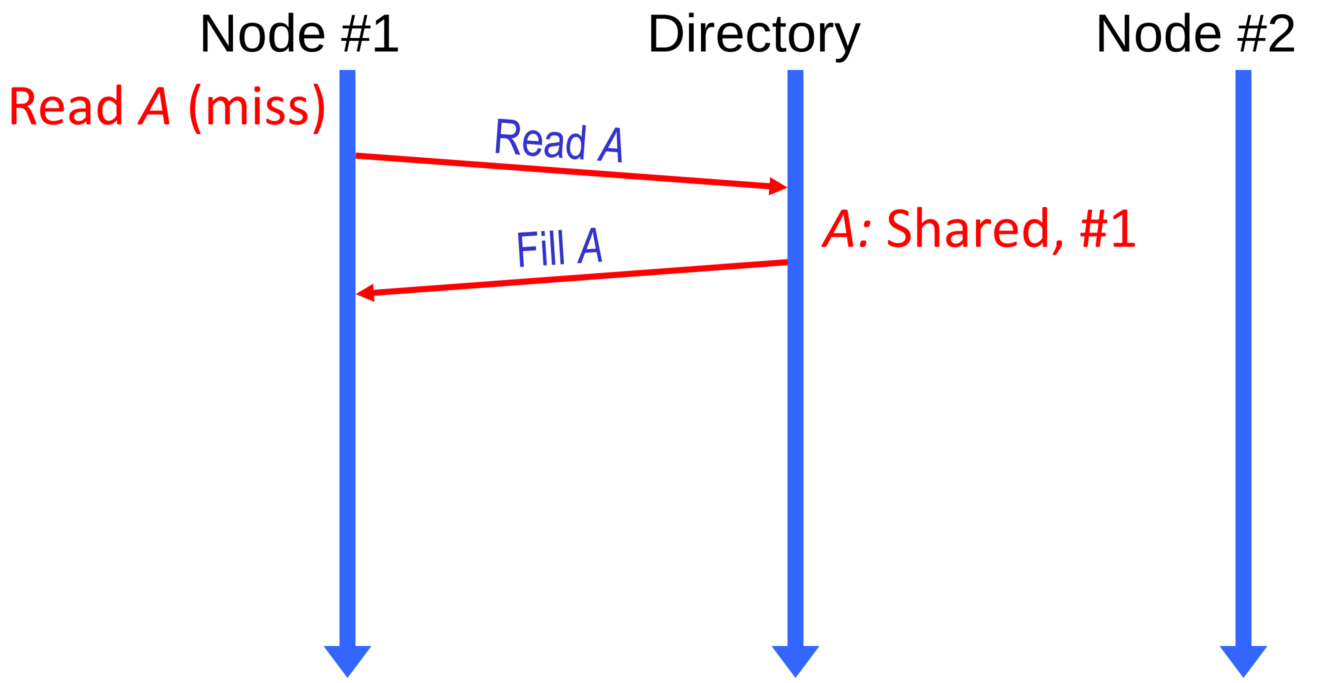
- Part I: **bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)…
  - …with scalable bandwidth one (point-to-point network, e.g., mesh)

- Part II: **processor snooping bandwidth**
  - Interesting: most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)…
  - …with scalable **directory protocol** (only spam processors that care)

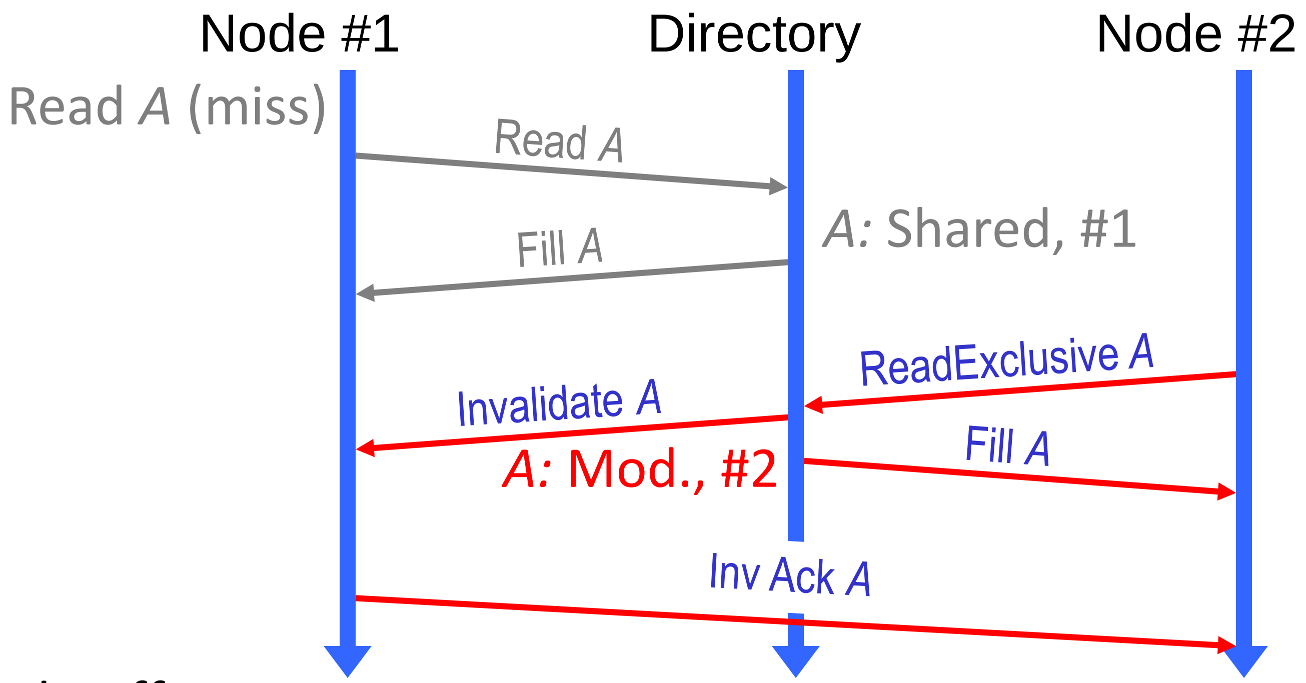
Directory Coherence Protocols

- Observe: physical address space statically partitioned
  - + Can easily determine which memory module holds a given line
    - That memory module sometimes called "**home**"
  - – Can't easily determine which processors have line in their caches
  - Bus-based protocol: broadcast events to all processors/caches
    - ± Simple and fast, but non-scalable
- **Directories**: non-broadcast coherence protocol
  - Extend memory to track caching information
  - For each physical cache line whose home this is, track:
    - **Owner**: which processor has a dirty copy (I.e., M state)
    - **Sharers**: which processors have clean copies (I.e., S state)
  - Processor sends coherence event to home directory
    - Home directory only sends events to processors that care

Read Processing



Write Processing



Trade-offs:

- Longer accesses (3-hop between Processor, directory, other Processor)
- Lower bandwidth → no snoops necessary

Makes sense either for CMPs (lots of L1 miss traffic)
  or large-scale servers (shared-memory MP > 32 nodes)

Serialization and Ordering

Let A and flag be initially 0

<u>P1</u>	<u>P2</u>
A += 5	while (flag == 0)
flag = 1	print A

Assume A and flag are in different cache blocks

What happens?

How do you implement it correctly?

Coherence vs. Consistency

Intuition says loads should return latest value

- what is latest?

Coherence concerns only one memory location

Consistency concerns apparent ordering for all locations

A Memory System is Coherent if

- can serialize all operations to that location such that,
- operations performed by any processor appear in program order
  - program order = order defined program text or assembly code
- value returned by a read is value written by last store to that location

Why Coherence != Consistency

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

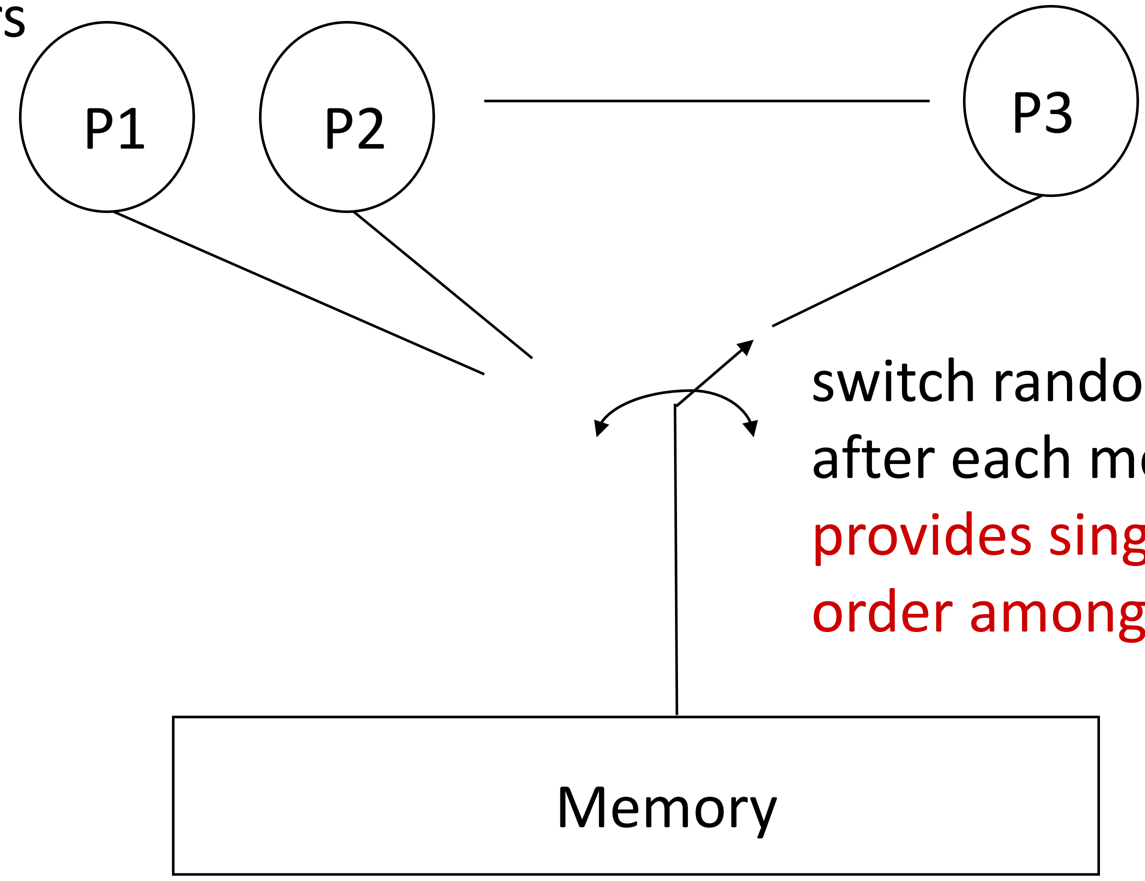
Intuition says printed A = B = 1

Coherence doesn't say anything. Why?

Your uniprocessor ordering mechanisms (ld/st queue) hurts here.
Why?

Sequential Consistency (SC)

processors issue memory ops in program order



switch randomly set after each memory op provides single sequential order among all operations

Sufficient Conditions for SC

Every proc. issues memory ops in program order

Memory ops happen (start and end) atomically

- must wait for store to complete before issuing next memory op
- after load, issuing proc waits for load to complete, before issuing next op

Easily implemented with a shared bus

Relaxed Memory Models

Motivation with Directory Protocols

- Misses have longer latency (do cache hits to get to next miss)
- Collecting acknowledgements can take even longer

Recall SC has

- Each processor generates at total order of its reads and writes (R-->R, R-->W, W-->W, & W-->R)
- That are interleaved into a global total order

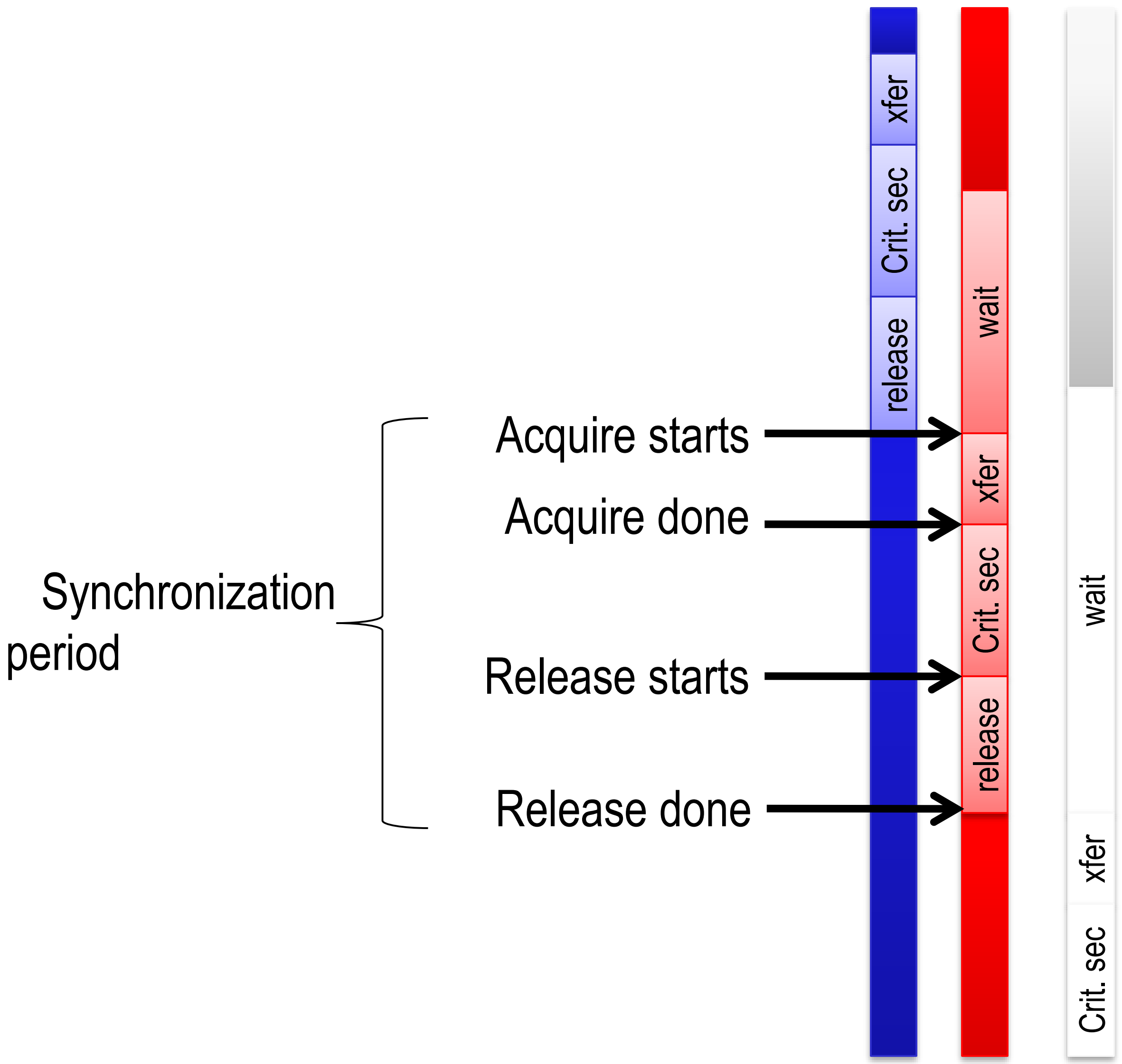
Example Relaxed Models

PC: Relax ordering from writes to (other proc's) reads

RC: Relax all read/write orderings (but add "fences")

Locks

Lock-based Mutual Exclusion



No contention:

- Want low latency

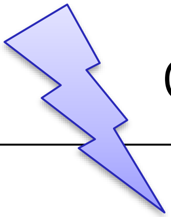
Contention:

- Want low period
- Low traffic
- Fairness

How Not to Implement Locks

- **LOCK**

```
while (lock_variable == 1);
```



```
lock_variable = 1
```

- **UNLOCK**

```
lock_variable = 0;
```

Solution: Atomic Read-Modify-Write

- Test&Set(r,x)

  ```
  {r=m[x]; m[x]=1;}
  ```

- r is register
- m[x] is memory location x

- Fetch&Op(r1,r2,x,op)

  ```
  {r1=m[x]; m[x]=op(r1,r2);}
  ```

- Swap(r,x)

  ```
  {temp=m[x]; m[x]=r; r=temp;}
  ```

- Compare&Swap(r1,r2,x)

  ```
  {temp=r2; r2=m[x]; if r1==r2 then m[x]=temp;}
  ```

Write a lock and unlock with test-and-set

- Lock:

- unlock:

Implementing RMWs

- Bus-based systems:
  - Hold bus and issue load/store operations without any intervening accesses by other processors

- Scalable systems
  - Acquire exclusive ownership via cache coherence
  - Perform load/store operations without allowing external coherence requests

Test-and-Set Spin Lock (T&S)

- Lock is "acquire", Unlock is "release"
- acquire(lock_ptr):

```
    while (true):
        // Perform "test-and-set"
        old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
        if (old == UNLOCKED):
            break    // lock acquired!
        // keep spinning, back to top of while loop
```

- release(lock_ptr):

```
    store[lock_ptr] <- UNLOCKED
```

- Performance problem
  - CAS is both a read and write; spinning causes lots of invalidations

Load Locked / Store Conditional (LL/SC)

- Sometimes having an atomic read/modify/write operation isn't viable.
  - Maybe your memory system just doesn't support it
  - Maybe it involves a very expensive "lock out" that prevents any other memory operation until the lock finishes.
- Still possible to manage with a two step process
  - Do a special load called a "load lock" or load "link".
    - Works like a normal load but it tells the hardware to watch for a store to that same address.
  - Do a "store conditional" to the same address
    - If there was a store to that location since the load lock, the store "fails" and that failure is communicated to the programmer (store has a destination register)
      - Failed store doesn't write to memory
  - Now we know if the load and store was effectively atomic.

Load-Locked Store-Conditional

- Load-locked
  - Issues a normal load…
  - …and sets a flag and address field
- Store-conditional
  - Checks that flag is set and address matches
    - If so, performs store and sets store register to 1.
    - Otherwise sets store register to 0.
- Flag is cleared by
  - Invalidation
  - Cache eviction
  - Context switch

```
lock:  lda r2, #1
       ll r1, lock_variable
       sc lock_variable, r2
       beqz r2, lock
       neqz r1, lock                // branch not equal to zero

unlock:st lock_variable, #0
```

Next up: Static Optimizations

- Why might we want to start loads early?
  - Moving loads to happen earlier is called "hoisting"
- What three things limits a complier's ability to hoist a load?