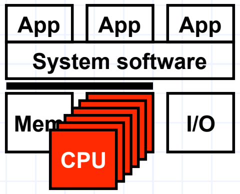
CIS 371
Computer Organization and Design

Unit 10: Shared Memory Multiprocessors

This Unit: Shared Memory Multiprocessors

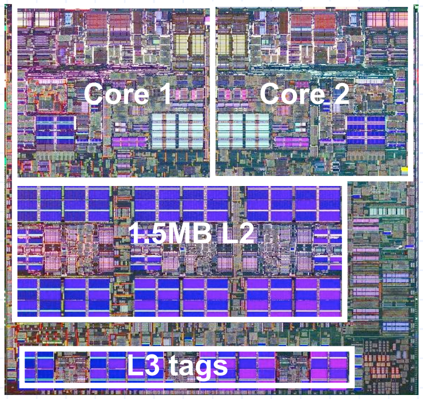


- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

Multiplying Performance

- A single processor can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism
  - Limited cache hierarchy
- What if we need even more computing power?
  - Use multiple processors!
  - But how?
- High-end example: Sun Ultra Enterprise 25k
  - 72 UltraSPARC IV+ processors, 1.5Ghz
  - 1024 GBs of memory
  - Niche: large database servers
  - $$$

Multicore: Mainstream Multiprocessors



Why multicore? What else would you do with 500 million transistors?

- **Multicore chips**
- **IBM Power5**
  - Two 2+GHz PowerPC cores
  - Shared 1.5 MB L2, L3 tags
- AMD Quad Phenom
  - Four 2.5-GHz cores
  - Per-core 512KB L2 cache
  - Shared 2MB L3 cache
- Intel Core 2 Quad
  - Four cores, shared 4 MB L2
  - Two 4MB L2 caches
- Sun Niagara
  - 8 cores, each 4-way threaded
  - Shared 2MB L2, shared FP
  - For servers, not desktop

Application Domains for Multiprocessors

- Scientific computing/supercomputing
  - Examples: weather simulation, aerodynamics, protein folding
  - Large grids, integrating changes over time
  - Each processor computes for a part of the grid
- Server workloads
  - Example: airline reservation database
  - Many concurrent updates, searches, lookups, queries
  - Processors handle different requests
- Media workloads
  - Processors compress/decompress different parts of image/frames
- Desktop workloads…
- Gaming workloads…

But software must be written to expose parallelism

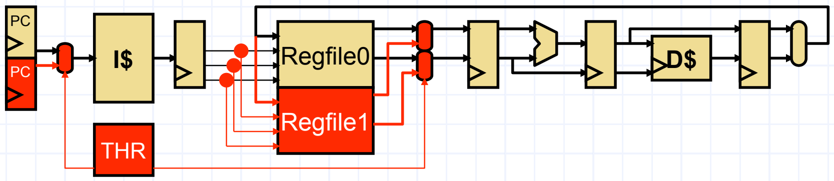
But First, Uniprocessor Concurrency

- Software "thread"
  - Independent flow of execution
  - Context state: PC, registers
  - Threads generally share the same memory space
  - "Process" like a thread, but different memory space
  - Java has thread support built in, C/C++ supports P-threads library
- Generally, system software (the O.S.) manages threads
  - "Thread scheduling", "context switching"
  - All threads share the one processor
    - Hardware timer interrupt occasionally triggers O.S.
    - Quickly swapping threads gives illusion of concurrent execution
  - Much more in CIS380

Multithreaded Programming Model

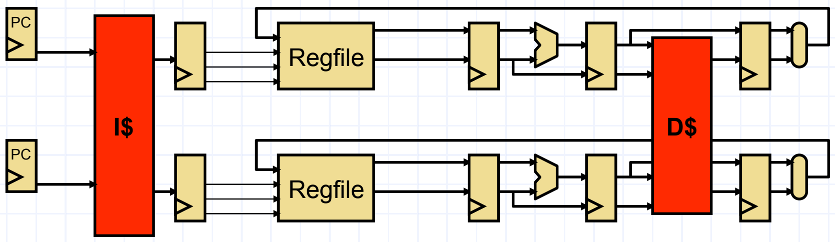
- Programmer explicitly creates multiple threads
- All loads & stores to a single **shared memory** space
  - Each thread has a private stack frame for local variables
- A "thread switch" can occur at any time
  - Pre-emptive multithreading by OS
- Common uses:
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send network message, wait for response)
  - Expressing parallel work via Thread-Level Parallelism (TLP)

Hardware Multithreading



- **Hardware Multithreading (MT)**
  - Multiple threads dynamically share a single pipeline (caches)
  - Replicate thread contexts: PC and register file
  - **Coarse-grain MT**: switch on L2 misses **Why?**
  - **Simultaneous MT**: no explicit switching, fine-grain interleaving
    - Pentium4 is 2-way hyper-threaded, leverages out-of-order core
  - + MT Improves utilization and throughput
    - Single programs utilize <50% of pipeline (branch, $ misses)
  - MT does not improve single-thread performance
    - Individual threads run as fast or even slower

Simplest Multiprocessor



- Replicate entire processor pipeline!
  - Instead of replicating just register file & PC
  - Exception: share caches (we'll address this bottleneck later)
- Same "shared memory" or "multithreaded" model
  - Loads and stores from two processors are interleaved
- Advantages/disadvantages over hardware multithreading?

Shared Memory Implementations

- **Multiplexed uniprocessor**
  - Runtime system and/or OS occasionally pre-empt & swap threads
  - Interleaved, but no parallelism
- **Hardware multithreading**
  - Tolerate pipeline latencies, higher efficiency
  - Same interleaved shared-memory model
- **Multiprocessing**
  - Multiply execution resources, higher peak performance
  - Same interleaved shared-memory model
  - Foreshadowing: allow private caches, further disentangle cores
- **All have same shared memory programming model**

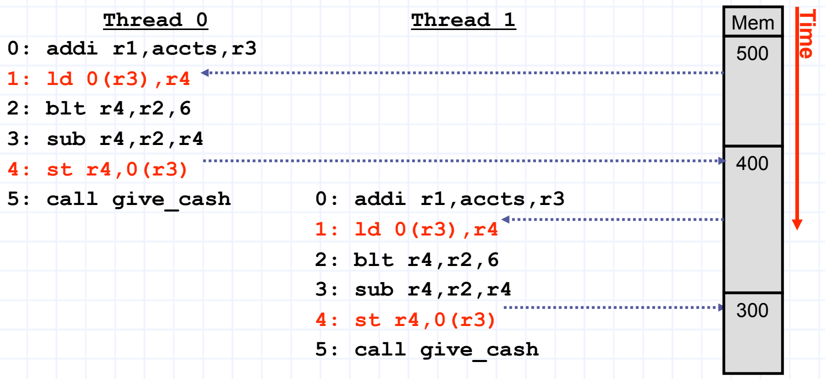
Thread-Level Parallelism Example

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

- **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared "loosely" (sometimes yes, mostly no), dynamically
- Example: database/web server (each query is a thread)
  - `accts` is **shared**, can't register allocate even if it were scalar
  - `id` and `amt` are private variables, register allocated to `r1`, `r2`
- Running example

An Example Execution

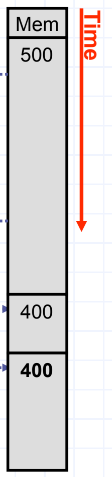


```
Thread 0                     Thread 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call give_cash            0: addi r1,accts,r3
                             1: ld 0(r3),r4
                             2: blt r4,r2,6
                             3: sub r4,r2,r4
                             4: st r4,0(r3)
                             5: call give_cash
```

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)

A **Problem** Execution

```
        Thread 0                  Thread 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
<<< Interrupt >>>
                               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
4: st r4,0(r3)
5: call give_cash
```



- Problem: wrong account balance! Why?
  - Solution: synchronize access to account balance

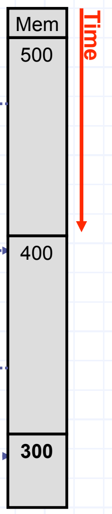
Synchronization

- **Synchronization**: a key issue for shared memory
  - Regulate access to shared data (mutual exclusion)
  - Software constructs: semaphore, monitor, mutex
  - Low-level primitive: **lock**
    - Operations: `acquire(lock)` and `release(lock)`
    - Region between `acquire` and `release` is a **critical section**
    - Must interleave `acquire` and `release`
    - Interfering `acquire` will block

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
shared int lock;
int id, amt;
acquire(lock);
if (accts[id].bal >= amt) {            // critical section
   accts[id].bal -= amt;
   give_cash(); }
release(lock);
```

A Synchronized Execution

```
        Thread 0                  Thread 1
   call acquire(lock)
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
<<< Interrupt >>>                 call acquire(lock) Spins!
                                  <<< Interrupt >>>
4: st r4,0(r3)
   call release(lock)
5: call give_cash                 (still in acquire)
                               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
```



- Fixed, but how do we implement acquire & release?

Strawman Lock (Incorrect)

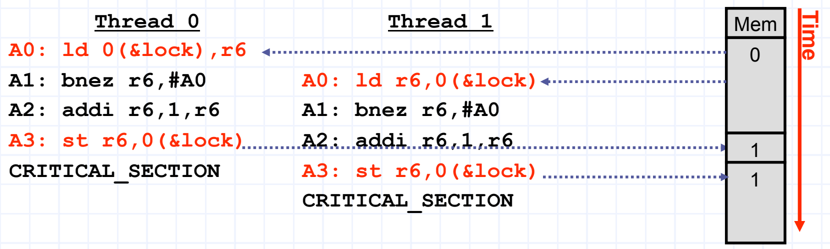
- **Spin lock**: software lock implementation
  - `acquire(lock): while (lock != 0); lock = 1;`
    - "Spin" while lock is 1, wait for it to turn 0

```
A0:  ld 0(&lock),r6
A1:  bnez r6,A0
A2:  addi r6,1,r6
A3:  st r6,0(&lock)
```

  - `release(lock): lock = 0;`

```
R0:  st r0,0(&lock)     // r0 holds 0
```

Strawman Lock (Incorrect)



```
      Thread 0                   Thread 1
A0: ld 0(&lock),r6
A1: bnez r6,#A0            A0: ld r6,0(&lock)
A2: addi r6,1,r6           A1: bnez r6,#A0
A3: st r6,0(&lock)         A2: addi r6,1,r6
CRITICAL_SECTION           A3: st r6,0(&lock)
                           CRITICAL_SECTION
```

- Spin lock makes intuitive sense, but doesn't actually work
  - Loads/stores of two `acquire` sequences can be interleaved
  - Lock `acquire` sequence also not atomic
  - **Same problem as before!**
- Note, `release` is trivially atomic

A Correct Implementation: SYSCALL Lock

```
ACQUIRE_LOCK:
A1: disable_interrupts          atomic
A2: ld r6,0(&lock)
A3: bnez r6,#A0
A4: addi r6,1,r6
A5: st r6,0(&lock)
A6: enable_interrupts
A7: return
```

- Implement lock in a SYSCALL
  - Only kernel can control interleaving by disabling interrupts
  + Works…
  – Large system call overhead
  – But not in a hardware multithreading or a multiprocessor…

Better Spin Lock: Use Atomic Swap

- ISA provides an atomic lock acquisition instruction
  - Example: **atomic swap**

    `swap r1,0(&lock)`
    - Atomically executes:

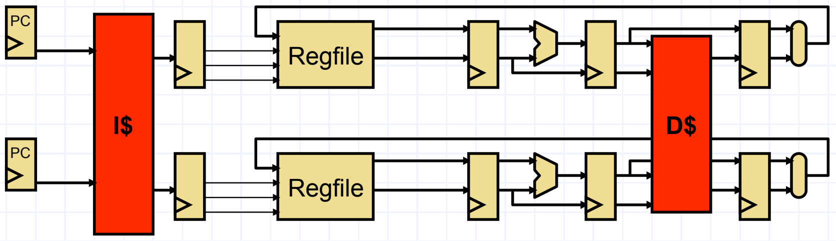
```
mov r1->r2
ld r1,0(&lock)
st r2,0(&lock)
```

- New acquire sequence

```
(value of r1 is 1)
A0: swap r1,0(&lock)
A1: bnez r1,A0
```

  - If lock was initially busy (1), doesn't change it, **keep looping**
  - If lock was initially free (0), acquires it (sets it to 1), break loop
- Insures lock held by **at most one thread**
  - Other variants: **exchange**, **compare-and-swap**, **test-and-set**, or **fetch-and-add**

Atomic Update/Swap Implementation



- How is atomic swap implemented?
  - Need to ensure no intervening memory operations
  - Requires blocking access by other threads temporarily (yuck)
- How to pipeline it?
  - Both a load and a store (yuck)
  - Not very RISC-like
  - Some ISAs provide a "load-link" and "store-conditional" insn. pair

Lock Correctness

```
        Thread 0              Thread 1
A0: swap r1,0(&lock)
A1: bnez r1,#A0       A0: swap r1,0(&lock)
CRITICAL_SECTION      A1: bnez r1,#A0
                      A0: swap r1,0(&lock)
                      A1: bnez r1,#A0
```

+ Test-and-set lock actually works…
    - Thread 1 keeps spinning

Programming With Locks Is Difficult

- Multicore processors are the way of the foreseeable future
    - TLP anointed as parallelism model of choice
    - Just one problem…
- Writing lock-based multi-threaded programs is difficult!
- More precisely:
    - Writing programs that are correct is "easy" (not really)
    - Writing programs that are highly parallel is "easy" (not really)
    - – **Writing programs that are both correct and parallel is difficult**
        - Very difficult (true)
        - Unfortunate goal (but that's the whole point after all)
    - Locking granularity issues

Coarse-Grain Locks: Correct but Slow

- **Coarse-grain locks**: e.g., one lock for entire database
    + Easy to make correct: no chance for unintended interference
    – No P in TLP: no two critical sections can proceed in parallel

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
shared int lock;

acquire(lock);
if (accts[id].bal >= amt) {
   accts[id].bal -= amt;
   give_cash(); }
release(lock);
```

Fine-Grain Locks: Parallel But Difficult

- **Fine-grain locks**: e.g., multiple locks, one per record
    + Fast: critical sections (to different records) can proceed in parallel
    – Difficult to make correct: easy to make mistakes
        - This particular example is easy
            - Requires only one lock per critical section
        - Consider critical section that requires two locks…

```
struct acct_t { int bal,lock; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;

acquire(accts[id].lock);
if (accts[id].bal >= amt) {
   accts[id].bal -= amt;
   give_cash(); }
release(accts[id].lock);
```

Multiple Locks

- **Multiple locks**: e.g., acct-to-acct transfer
  - Must acquire both `id_from`, `id_to` locks
  - Running example with accts 241 and 37
  - Simultaneous transfers 241 → 37 and 37 → 241
  - Contrived… but even contrived examples must work correctly too

```
struct acct_t { int bal,lock; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

acquire(accts[id_from].lock);
acquire(accts[id_to].lock);
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
release(accts[id_to].lock);
release(accts[id_from].lock);
```

Multiple Locks And Deadlock

```
Thread 0                          Thread 1

id_from = 241;                    id_from = 37;
id_to = 37;                       id_to = 241;

acquire(accts[241].lock);         acquire(accts[37].lock);
// wait to acquire lock           // wait to acquire lock 241
   37                             // waiting…
// waiting…                       // …
// still waiting…
```

- **Deadlock**: circular wait for shared resources
  - Thread 0 has lock 241 waits for lock 37
  - Thread 1 has lock 37 waits for lock 241
  - Obviously this is a problem
  - The solution is …

Correct Multiple Lock Program

- **Always acquire multiple locks in same order**
  - Just another thing to keep in mind when programming
    - Ho hum…

```
struct acct_t { int bal,lock; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;
int id_first = min(id_from, id_to);
int id_second = max(id_from, id_to);

acquire(accts[id_first].lock);
acquire(accts[id_second].lock);
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
release(accts[id_second].lock);
release(accts[id_first].lock);
```

Correct Multiple Lock Execution

```
Thread 0                          Thread 1

id_from = 241;                    id_from = 37;
id_to = 37;                       id_to = 241;
id_first = min(241,37)=37;        id_first = min(37,241)=37;
id_second = max(37,241)=241;      id_second = max(37,241)=241;

acquire(accts[37].lock);          // wait to acquire lock 37
acquire(accts[241].lock);         // waiting…
// do stuff                       // …
release(accts[241].lock);         // …
release(accts[37].lock);          // …
                                  acquire(accts[37].lock);
```

- Great, are we done? No

More Lock Madness

- What if…
  - Some actions (e.g., deposits, transfers) require 1 or 2 locks…
  - …and others (e.g., prepare statements) require all of them?
  - Can these proceed in parallel?
- What if…
  - There are locks for global variables (e.g., operation id counter)?
  - When should operations grab this lock?
- What if… what if… what if…

- **So lock-based programming is difficult…**
- **…wait, it gets worse**

And To Make It Worse…

- **Acquiring locks is expensive…**
  - By definition requires a slow atomic instructions
    - Specifically, acquiring write permissions to the lock
  - Ordering constraints (see soon) make it even slower

- **…and 99% of the time un-necessary**
  - Most concurrent actions don't actually share data
  – You paying to acquire the lock(s) for no reason

- Fixing these problem is an area of active research
  - One proposed solution "Transactional Memory"

Research: Transactional Memory (TM)

- **Transactional Memory**
  + Programming simplicity of coarse-grain locks
  + Higher concurrency (parallelism) of fine-grain locks
    - Critical sections only serialized if data is actually shared
  + No lock acquisition overhead
  - Hottest thing since sliced bread
  - No fewer than 9 research projects: Brown, Stanford, MIT, Intel…
    - Penn too

Transactional Memory: The Big Idea

- Big idea I: **no locks, just shared data**
  - Look ma, no locks
- Big idea II: **optimistic (speculative) concurrency**
  - Execute critical section speculatively, abort on conflicts
  - "Better to beg for forgiveness than to ask for permission"

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory: Read/Write Sets

- **Read set**: set of shared addresses critical section reads
  - Example: `accts[37].bal, accts[241].bal`
- **Write set**: set of shared addresses critical section writes
  - Example: `accts[37].bal, accts[241].bal`

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory: Begin

- `begin_transaction`
  - Take a local register checkpoint
  - Begin locally tracking read set (remember addresses you read)
    - See if anyone else is trying to write it
  - Locally buffer all of your writes (invisible to other processors)
  - + **Local actions only: no lock acquire**

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory: End

- `end_transaction`
  - Check read set: is all data you read still valid (i.e., no writes to any)
  - Yes? Commit transactions: commit writes
  - No? Abort transaction: restore checkpoint

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory Implementation

- How are read-set/write-set implemented?
  - Track locations accessed using bits in the cache
- Read-set: additional "transactional read" bit per block
  - Set on reads between begin_transaction and end_transaction
  - Any other write to block with set bit ➔ triggers abort
  - Flash cleared on transaction abort or commit
- Write-set: additional "transactional write" bit per block
  - Set on writes between begin_transaction and end_transaction
  - Flash cleared on transaction commit
  - On transaction abort: blocks with set bit are invalidated

Transactional Execution

```
Thread 0                          Thread 1

id_from = 241;                    id_from = 37;
id_to = 37;                       id_to = 241;

begin_transaction();              begin_transaction();
if(accts[241].bal > 100) {        if(accts[37].bal > 100) {
   …                                 accts[37].bal -= amt;
   // write accts[241].bal           acts[241].bal += amt;
   // abort                       }
                                  end_transaction();
                                  // no writes to accts[241].bal
                                  // no writes to accts[37].bal
                                  // commit
```

Transactional Execution II (More Likely)

```
Thread 0                          Thread 1

id_from = 241;                    id_from = 450;
id_to = 37;                       id_to = 118;

begin_transaction();              begin_transaction();
if(accts[241].bal > 100) {        if(accts[450].bal > 100) {
   accts[241].bal -= amt;            accts[450].bal -= amt;
   acts[37].bal += amt;              acts[118].bal += amt;
}                                 }
end_transaction();                end_transaction();
// no write to accts[240].bal     // no write to accts[450].bal
// no write to accts[37].bal      // no write to accts[118].bal
// commit                         // commit
```

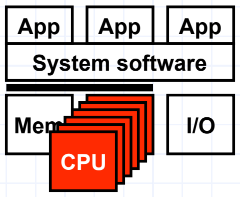
- Critical sections execute in parallel

So, Let's Just Do Transactions?

- What if…
  - Read-set or write-set bigger than cache?
  - Transaction gets swapped out in the middle?
  - Transaction wants to do I/O or SYSCALL (not-abortable)?
- How do we transactify existing lock based programs?
  - Replace **`acquire`** with **`begin_trans`** does not always work
- Several different kinds of transaction semantics
  - Which one do we want?

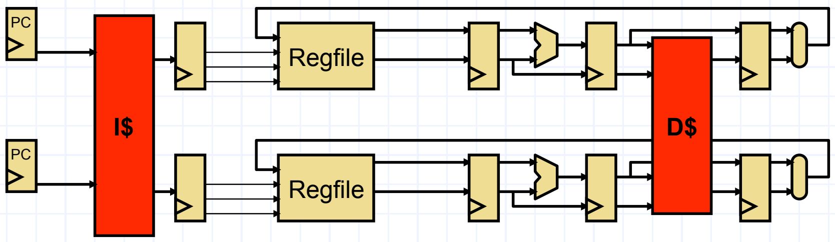
- That's what these research groups are looking at
- Industry adoption:
  - Sun's Rock processor has best-effort hardware TM
  - Speculative locking: Azul systems and Intel (rumor)

Roadmap Checkpoint



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

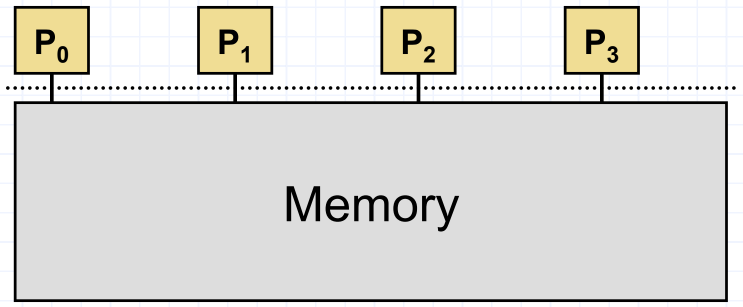
Recall: Simplest Multiprocessor



- What if we don't want to share the L1 caches?
  - Bandwidth and latency issue
- Solution: use per-processor ("private") caches
  - Coordinate them with a *Cache Coherence Protocol*

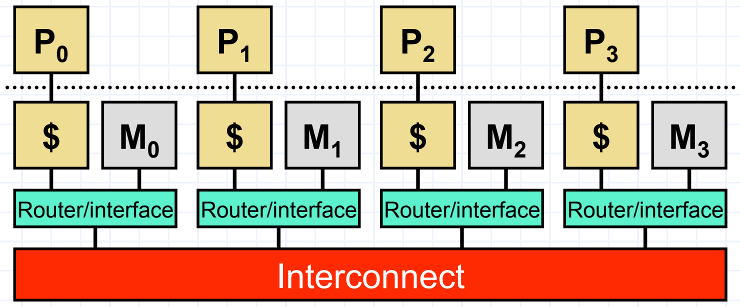
Shared-Memory Multiprocessors

- **Conceptual model**
  - The shared-memory abstraction
  - Familiar and feels natural to programmers
  - Life would be easy if systems actually looked like this…



Shared-Memory Multiprocessors

- …but systems actually look more like this
  - Processors have caches
  - Memory may be physically distributed
  - Arbitrary interconnect



Revisiting Our Motivating Example

CPU0	CPU1	Mem

```
Processor 0
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash
```

critical section (locks not shown)

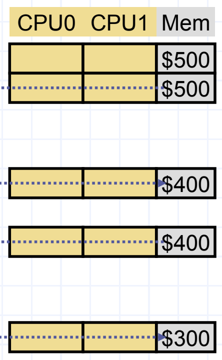
```
Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash
```

critical section (locks not shown)

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `$r3`)

No-Cache, No-Problem

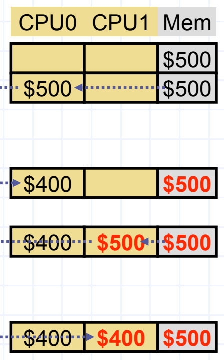
```
Processor 0                 Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash        0: addi $r3,$r1,&accts
                            1: lw $r4,0($r3)
                            2: blt $r4,$r2,6
                            3: sub $r4,$r4,$r2
                            4: sw $r4,0($r3)
                            5: jal dispense_cash
```



- Scenario I: processors have no caches
  - No problem

Cache Incoherence

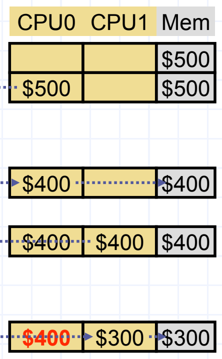
```
Processor 0                 Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash        0: addi $r3,$r1,&accts
                            1: lw $r4,0($r3)
                            2: blt $r4,$r2,6
                            3: sub $r4,$r4,$r2
                            4: sw $r4,0($r3)
                            5: jal dispense_cash
```



- Scenario II(a): processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
  - Can get incoherent (inconsistent)

Write-Through Doesn’t Fix It

```
Processor 0                 Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash        0: addi $r3,$r1,&accts
                            1: lw $r4,0($r3)
                            2: blt $r4,$r2,6
                            3: sub $r4,$r4,$r2
                            4: sw $r4,0($r3)
                            5: jal dispense_cash
```



- Scenario II(b): processors have write-through caches
  - This time only 2 (different) copies of `accts[241].bal`
  - No problem? What if another withdrawal happens on processor 0?

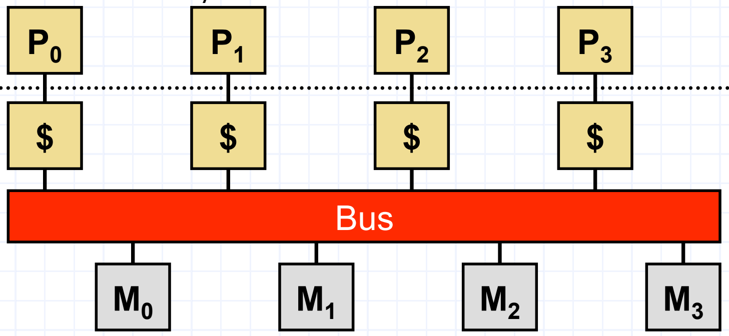
What To Do?

- No caches?
  - Slow
- Make shared data uncachable?
  - Faster, but still too slow
  - Entire `accts` database is technically “shared”
    - Definition of “loosely shared”
    - Data only really shared if two ATMs access same acct at once
- Flush all other caches on writes to shared data?
  - May as well not have caches

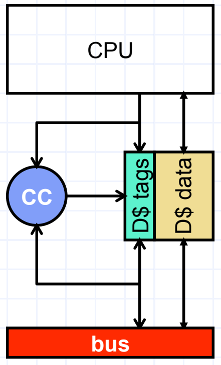
- **Hardware cache coherence**
  - Rough goal: all caches have same data at all times
  - + Minimal flushing, maximum caching → best performance

Bus-based Multiprocessor

- Simple multiprocessors use a bus
  - **All** processors see all requests at the **same time**, same order
- Memory
  - Single memory module, **-or-**
  - Banked memory module

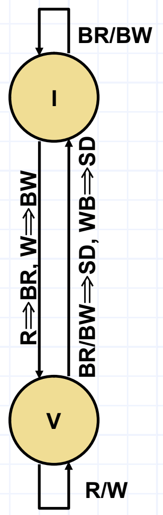


Hardware Cache Coherence



- **Coherence**
  - all copies have same data at all times
- **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
- Three processor-initiated events
  - **R**: read **W**: write **WB**: write-back
- One response event: **SD:** send data
- Two remote-initiated events
  - **BR**: bus-read, read miss from ***another*** processor
  - **BW**: bus-write, write miss from ***another*** processor

VI (MI) Coherence Protocol



- **VI (valid-invalid) protocol**: aka MI
  - Two states (per block in cache)
    - **V (valid)**: have block
    - **I (invalid)**: don't have block
    - + Can implement with valid bit
- Protocol diagram (left)
  - Convention: event⇒generated-event
  - Summary
    - If anyone wants to read/write block
    - Give it up: transition to **I** state
    - Write-back if your own copy is dirty
- This is an **invalidate protocol**
- **Update protocol**: copy data, don't invalidate
  - Sounds good, but wastes a lot of bandwidth

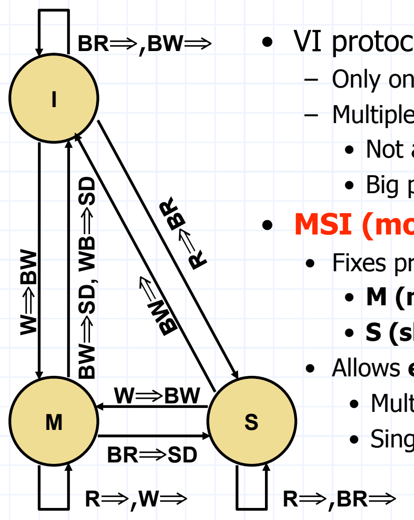
VI Protocol (Write-Back Cache)

```
Processor 0                   Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash          0: addi $r3,$r1,&accts
                              1: lw $r4,0($r3)
                              2: blt $r4,$r2,6
                              3: sub $r4,$r4,$r2
                              4: sw $r4,0($r3)
                              5: jal dispense_cash
```

CPU0	CPU1	Mem
		500
V:500		500
V:400		500
I:	V:400	400
	V:300	400

- `lw` by processor 1 generates a BR (bus read)
  - processor 0 responds by sending its dirty copy, transitioning to **I**

VI → MSI



- VI protocol is inefficient
  - Only one cached copy allowed in entire system
  - Multiple copies can't exist even if read-only
    - Not a problem in example
    - Big problem in reality
- **MSI (modified-shared-invalid)**
  - Fixes problem: splits "V" state into two states
    - **M (modified)**: local dirty copy
    - **S (shared)**: local clean copy
  - Allows **either**
    - Multiple read-only copies (S-state) **--OR--**
    - Single read/write copy (M-state)

MSI Protocol (Write-Back Cache)

```
Processor 0                    Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash           0: addi $r3,$r1,&accts
                               1: lw $r4,0($r3)
                               2: blt $r4,$r2,6
                               3: sub $r4,$r4,$r2
                               4: sw $r4,0($r3)
                               5: jal dispense_cash
```

CPU0	CPU1	Mem
		500
S:500		500
M:400		500
S:400	S:400	400
I:	M:300	400

- `lw` by processor 1 generates a BR
  - Processor 0 responds by sending its dirty copy, transitioning to **S**
- `sw` by processor 1 generates a BW
  - Processor 0 responds by transitioning to **I**

Exclusive Clean Protocol Optimization

```
Processor 0                    Processor 1
0: addi $r3,$r1,&accts
1: lw $r4,0($r3)
2: blt $r4,$r2,6
3: sub $r4,$r4,$r2
4: sw $r4,0($r3)
5: jal dispense_cash           0: addi $r3,$r1,&accts
                               1: lw $r4,0($r3)
                               2: blt $r4,$r2,6
                               3: sub $r4,$r4,$r2
                               4: sw $r4,0($r3)
                               5: jal dispense_cash
```

	CPU0	CPU1	Mem
			500
	E:500		500
(No miss)	M:400		500
	S:400	S:400	400
	I:	M:300	400

- Most modern protocols also include **E (exclusive)** state
  - Interpretation: "I have the only cached copy, and it's a **clean** copy"
  - Why would this state be useful?

Cache Coherence and Cache Misses

- A coherence protocol can effect a cache's miss rate ($\%_{miss}$)
  - Requests from other processors can invalidate (evict) local blocks
  - 4C miss model: compulsory, capacity, conflict, **coherence**
  - **Coherence miss**: miss to a block evicted by bus event
    - As opposed to a processor event
- Cache parameters interact with coherence misses
  - Larger capacity: more coherence misses
    - But offset by reduction in capacity misses
  - Increased block size: more coherence misses
    - **False sharing**: "sharing" a cache line without sharing data
    - Creates pathological "ping-pong" behavior
    - Careful data placement may help, but is difficult

Cache Coherence and Cache Misses

- A coherence protocol can effect a cache's miss rate ($\%_{miss}$)
  - Requests from other processors can invalidate (evict) local blocks
  - 4C miss model: compulsory, capacity, conflict, **coherence**
  - **Coherence miss**: miss to a block evicted by bus event
    - As opposed to a processor event
  - Example: direct-mapped 4B cache, 1B blocks, 4-bit memory

Cache contents (state:address)

Set00 Set01 Set10 Set11	Event	Outcome
S:0000, M:0001, S:0010, S:0011	Wr:0011	Upgrade Miss
S:0000, M:0001, S:0010, **M**:0011	BusRd:0000	Nothing
S:0000, M:0001, S:0010, M:0011	BusWr:0010	S→I Invalidation
S:0000, M:0001, **I**:0010, M:0011	Rd:1011	Compulsory Miss
S:0000, M:0001, I:0010, **S:1011**	Rd:0010	**Coherence Miss**
S:0000, M:0001, **S:0010**, S:1011		

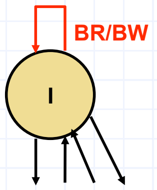
Snooping Bandwidth Requirements

- Coherence events generated on…
  - L2 misses (and writebacks)
- Some parameters
  - 2 GHz CPUs, 2 IPC, 33% memory operations,
  - 2% of which miss in the L2, 64B blocks, 50% dirty
  - (0.33 * 0.02 * 1.5) = 0.01 events/insn
  - 0.01 events/insn * 2 insn/cycle * 2 cycle/ns = 0.04 events/ns
  - Address request: 0.04 events/ns * 4 B/event = 0.16 GB/s
  - Data response: 0.04 events/ns * 64 B/event = 2.56 GB/s
- That's 2.5 GB/s … per processor
  - With 16 processors, that's 40 GB/s!
  - With 128 processors, that's 320 GB/s!!
  - You can use multiple buses… but that hinders global ordering

More Snooping Bandwidth Problems

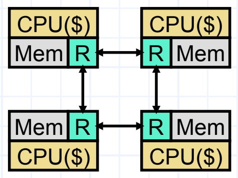
- Bus bandwidth is not the only problem
- Also **processor snooping bandwidth**
  - 0.01 events/insn * 2 insn/cycle = 0.02 events/cycle per processor
  - 16 processors: 0.32 bus-side tag lookups per cycle
    - Add 1 port to cache tags? Sure
    - Invalidate over upgrade: Tags smaller data, ports less expensive
  - 128 processors: 2.56 bus-side tag lookups per cycle!
    - Add 3 ports to cache tags? Oy vey!
  - Implementing **inclusion** (L1 is strict subset of L2) helps a little
    - 2 additional ports on L2 tags only
    - Processor doesn't use existing tag port most of the time
    - If L2 doesn't care (99% of the time), no need to bother L1
    - – Still kind of bad though
- **Upshot**: bus-based coherence doesn't scale well

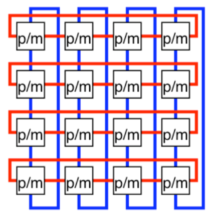
Scalable Cache Coherence



- Part I: **bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)…
  - …with scalable one (point-to-point network, e.g., mesh)
- Part II: **processor snooping bandwidth**
  - Most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)…
  - …with scalable **directory protocol** (only notify processors that care)

Scalable Cache Coherence



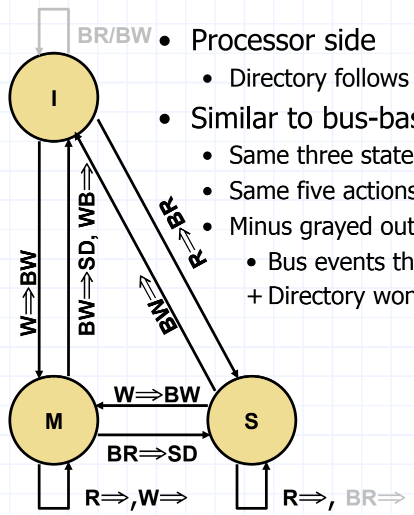


- Point-to-point interconnects
  - **Glueless MP**: no need for additional "glue" chips
  - + Can be arbitrarily large: 1000's of processors
    - **Massively parallel processors (MPPs)**
    - Only government (DoD) has MPPs…
  - Companies have much smaller systems: 32–64 processors
    - **Scalable multi-processors**
  - AMD Opteron/Phenom – point-to-point, glueless MP, uses broadcast

Directory Coherence Protocols

- Observe: 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

MSI Directory Protocol



- Processor side
  - Directory follows its own protocol (obvious in principle)
- Similar to bus-based MSI
  - Same three states
  - Same five actions (keep BR/BW names)
  - Minus grayed out arcs/actions
    - Bus events that would not trigger action anyway
    - + Directory won't bother you unless you need to act

Directory MSI Protocol

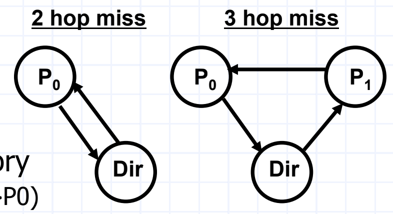
```
Processor 0                    Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call dispense_cash          0: addi r1,accts,r3
                               1: ld 0(r3),r4
                               2: blt r4,r2,6
                             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

P0	P1	Directory
		–:–:500
S:500		S:0:500
M:400		M:0:500 (stale)
S:400	S:400	S:0,1:400
	M:300	M:1:400

- `ld` by P1 sends BR to directory
  - Directory sends BR to P0, P0 sends P1 data, does WB, goes to **S**
- `st` by P1 sends BW to directory
  - Directory sends BW to P0, P0 goes to **I**

Directory Flip Side: Latency

- Directory protocols
  - + Lower bandwidth consumption → more scalable
  - – Longer latencies
- Two read miss situations



- Unshared: get data from memory
  - Snooping: 2 hops (P0→memory→P0)
  - Directory: 2 hops (P0→memory→P0)
- Shared or exclusive: get data from other processor (P1)
  - Assume cache-to-cache transfer optimization
  - Snooping: 2 hops (P0→P1→P0)
  - – Directory: **3 hops** (P0→memory→P1→P0)
  - Common, with many processors high probability someone has it

Directory Flip Side: Complexity

- Latency not only issue for directories
  - Subtle correctness issues as well
  - Stem from unordered nature of underlying inter-connect
- Individual requests to single cache must be ordered
  - Bus-based Snooping: all processors see all requests in same order
    - Ordering automatic
  - Point-to-point network: requests may arrive in different orders
    - Directory has to enforce ordering explicitly
    - Cannot initiate actions on request B…
    - Until all relevant processors have completed actions on request A
    - Requires directory to collect acks, queue requests, etc.
- Directory protocols
  - Obvious in principle
  - – Complicated in practice

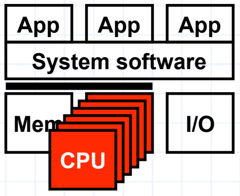
Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
  - Can be easily combined to form multi-processors
    - E.g., Intel Pentium4 Xeon
  - Multi-core
- Larger scale (directory) systems built from smaller MPs
  - E.g., Sun Wildfire, NUMA-Q, IBM Summit
- Some shared memory machines are **not cache coherent**
  - E.g., CRAY-T3D/E
  - Shared data is uncachable
  - If you want to cache shared data, copy it to private data section
  - Basically, cache coherence implemented in software
    - Have to really know what you are doing as a programmer

Best of Both Worlds?

- Ignore processor snooping bandwidth for a minute
- Can we combine best features of snooping and directories?
  - From snooping: fast two-hop cache-to-cache transfers
  - From directories: scalable point-to-point networks
  - In other words…
- Can we use broadcast on an unordered network?
  - Yes, and most of the time everything is fine
  - But sometimes it isn't … **protocol race**
- Research Proposal: **Token Coherence (TC)**
  - An unordered broadcast snooping protocol … without data races

Roadmap Checkpoint



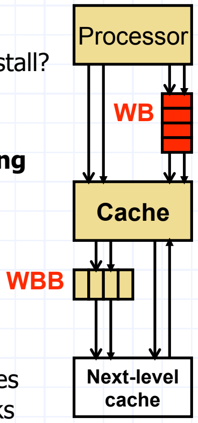
- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

Hiding Store Miss Latency

- Recall (back from caching unit)
  - Hiding store miss latency
  - How? Write buffer
- Said it would complicate multiprocessors
  - Yes. It does.

Recall: Write Misses and Write Buffers

- Read miss?
  - Load can’t go on without the data, it must stall
- Write miss?
  - Technically, no instruction is waiting for data, why stall?
- **Write buffer**: a small buffer
  - Stores put address/value to write buffer, **keep going**
  - Write buffer writes stores to D$ in the background
  - Loads must search write buffer (in addition to D$)
  - + Eliminates stalls on write misses (mostly)
  - – Creates some problems (later)
- Write buffer vs. writeback-buffer
  - Write buffer: “in front” of D$, for hiding store misses
  - Writeback buffer: “behind” D$, for hiding writebacks



Memory Consistency

- **Memory 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

Coherence vs. Consistency

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

- **Intuition says**: P1 prints A=1
- **Coherence says**: absolutely nothing
  - P1 can see P0's write of `flag` before write of `A`!!! How?
    - Maybe coherence event of `A` is delayed somewhere in network
    - **Or P0 has a coalescing write buffer that reorders writes**
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **Real systems** act in this strange manner

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**

SC Doesn't "Happen Naturally" Why?

- What is consistency concerned with?
  - P1 doesn't actually view P0's committed loads and stores
  - Views their **coherence events** instead
  - "Consistency model": how observed order of coherence events relates to order of committed insns
- What does SC say?
  - Coherence event order must match committed insn order
    - And be identical for all processors
  - Let's see what that implies

SC + Write Buffers

- Store misses are slow
  - Global acquisition of M state (write permission)
  - – Multiprocessors have more store misses than uniprocessors
    - **Upgrade miss**: I have block in S, require global upgrade to M
- Apparent solution: **write buffer**
  - Commit store to write buffer, let it absorb store miss latency
  - But a write buffer means…
  - I see my own stores commit before everyone else sees them

SC + Write Buffers

```
                      A=0; B=0;
Processor 0                      Processor 1
A=1;      // in-order to WB      B=1;      // in-order to WB
if(B==0)  // in-order commit     if(A==0)  // in-order commit
A=1;      // in-order to D$      B=1;      // in-order to D$
```

- Possible for both (**B==0**) and (**A==0**) to be true
- Because **B=1** and **A=1** are just sitting in the write buffers
  - Which is wrong
  - So does SC mean no write buffer?
  - – Yup, and that hurts

Is SC Really Necessary?

- SC
  - + Most closely matches programmer's intuition (don't under-estimate)
  - – Restricts optimization by compiler, CPU, memory system
  - Supported by MIPS, HP PA-RISC
- Is full-blown SC really necessary? What about…
  - All processors see others' loads/stores in program order
  - But not all processors have to see same global order
  - + Allows processors to have in-order write buffers
  - – Doesn't confuse programmers too much
    - Synchronized programs (e.g., our example) work as expected
  - **Processor Consistency (PC)**: e.g., Intel iA32, SPARC

Weak Memory Ordering

- For properly synchronized programs…
- …only **acquires/releases** must be strictly ordered
- Why? **acquire-release** pairs define **critical sections**
  - Between critical-sections: data is private
    - Globally unordered access OK
  - Within critical-section: access to shared data is exclusive
    - Globally unordered access also OK
  - Implication: compiler or dynamic scheduling is OK
    - As long as re-orderings do not cross synchronization points
- **Weak Ordering (WO)**: Alpha, IA-64, PowerPC
  - ISA provides fence insns to indicate scheduling barriers
  - Proper use of fences is somewhat subtle

Pop Quiz!

- Answer the following two questions:

Initially: **x==0, y==0**

thread 1	thread 2
ld x **ld y**	**st 1 → y** **st 1 → x**

- What value pairs can be read by the two loads?
  - (x, y) pairs:

Initially: **x==0, y==0**

thread 1	thread 2
st 1 → y **ld x**	**st 1 → x** **ld y**

- What value pairs can be read by the two loads?
  - (x, y) pairs:

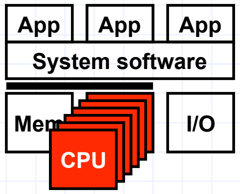
Fences aka Memory Barriers

- **Fences (memory barriers)**: special insns
  - Ensure that loads/stores don’t cross acquire release boundaries
  - Very roughly

```
acquire
fence
critical section
fence
release
```

- How do they work?
  - `fence` insn must commit before any younger insn dispatches
    - This also means write buffer is emptied
  - – Makes lock acquisition and release slow(er)
- **Use synchronization library, don’t write your own**

Summary



- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multihreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models