CS758: Multicore Programming

Shared Memory & Concurrency

Prof. David A. Wood
University of Wisconsin-Madison

Today's Outline: Shared Memory Review

- Introduction to Shared Memory
  - Thread-Level Parallelism
  - Shared Memory
- Cache Coherence
- Synchronization
- Memory Consistency

Thread-Level Parallelism

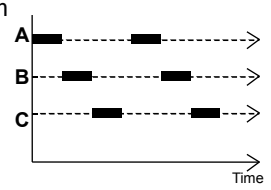
```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id,amt;
if (accts[id].bal >= amt)
{
   accts[id].bal -= amt;
   spew_cash();
}
```

```
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
```

- **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 even if it were scalar
  - `id` and `amt` are private variables, register allocated to `r1` & `r2`

Concurrency v. Parallelism

- Concurrency is not (just) parallelism
- Concurrency
  - Logically simultaneous processing
  - Interleaved execution (perhaps on UP)



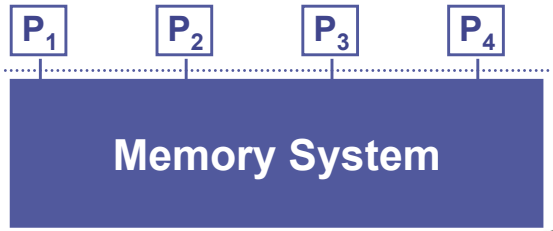
- Parallelism
  - Physically simultaneous processing
  - Requires multiprocessor

Shared Memory

- **Shared memory**
  - Multiple execution contexts sharing a single address space
    - Multiple programs (MIMD)
    - Or more frequently: multiple copies of one program (SPMD)
  - Implicit (automatic) communication via loads and stores
  + Simple software
    - No need for messages, communication happens naturally
      – Maybe too naturally
    - Supports irregular, dynamic communication patterns
      - Both DLP and **TLP**
  – Complex hardware
    - Must create a uniform view of memory
      - Several aspects to this as we will see

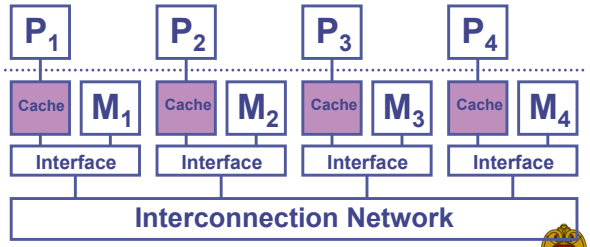
Shared-Memory Multiprocessors

- **Provide a shared-memory abstraction**
  - Familiar and efficient for programmers



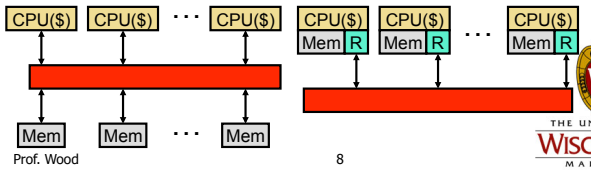
Shared-Memory Multiprocessors

- **Provide a shared-memory abstraction**
  - Familiar and efficient for programmers



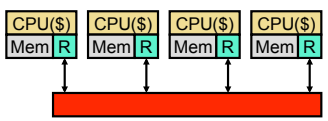
Paired vs. Separate Processor/Memory?

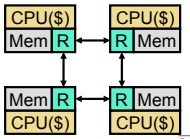
- **Separate processor/memory**
  - **Uniform memory access (UMA):** equal latency to all memory
  + Simple software, doesn't matter where you put data
  – Lower peak performance
  - Bus-based UMAs common: **symmetric multi-processors (SMP)**
- **Paired processor/memory**
  - **Non-uniform memory access (NUMA)**: faster to local memory
  – More complex software: where you put data matters
  + Higher peak performance: assuming proper data placement



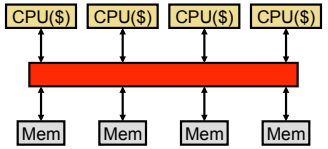
Shared vs. Point-to-Point Networks

- **Shared network**: e.g., bus (left)
  - + Low latency
  - – Low bandwidth: doesn't scale beyond ~16 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



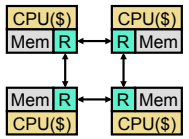


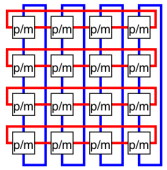
Implementation #1: Snooping Bus MP



- Two basic implementations
- Bus-based systems
  - Typically small: 2–8 (maybe 16) processors
  - Typically processors split from memories (UMA)
    - **Multiple processors on single chip (CMP)**
    - **Symmetric multiprocessors (SMPs)**
    - Common, most chips have 2-4 cores today

Implementation #2: Scalable MP





- General point-to-point network-based systems
  - Typically processor/memory/router blocks (NUMA)
    - **Glueless MP**: no need for additional "glue" chips
  - Can be arbitrarily large: 1000's of processors
    - **Massively parallel processors (MPPs)**
  - Increasingly used for small systems
    - Eliminates need for buses, enables point-to-point wires
    - **Coherent Hypertransport (AMD Opteron)**
    - **Intel QuickPath (Core 2)**

Today's Outline: Shared Memory Review

- Motivation for Four-Lecture Course
- Introduction to Shared Memory
- Cache Coherence
  - Problem
  - Snooping
  - Directories
- Synchronization
- Memory Consistency

An Example Execution

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

CPU0	CPU1	Mem

- 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 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        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
```

500
500
400
400
300

- Scenario I: processors have no caches
  - No problem

Cache Incoherence

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

		500
V:500		500
D:400		500
D:400	V:500	500
D:400	D:400	500

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

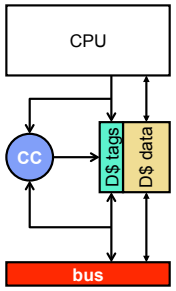
Write-Thru Alone Doesn't Help

```
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 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
```

		500
V:500		500
V:400		400
V:400	V:400	400
V:400	V:300	300

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

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

Bus-Based Coherence Protocols

- Bus-based coherence protocols
  - Also called **snooping** or **broadcast**
  - **ALL controllers see ALL transactions IN SAME ORDER**
    - Bus is the **ordering point**
    - Protocol relies on all processors seeing a total order of requests
  - Three processor-side events
    - **R**: read
    - **W**: write
    - **WB**: write-back (select block for replacement)
  - Three bus-side events
    - **BR**: bus-read, read miss on another processor
    - **BW**: bus-write, write miss or write-back on another processor
    - **CB**: copy-back, send block back to memory or other processor
- Point-to-point network protocols also exist
  - Typical solution is a **directory protocol**

VI (MI) Coherence Protocol

- **VI (valid-invalid) protocol**: aka MI
  - Two states (per block)
    - **V (valid)**: have block
      - aka **M (modified)** when block written
    - **I (invalid)**: don't have block
- Protocol summary
  - If anyone wants to read/write block
    - Give it up: transition to **I** state
  - copy-back on replacement or other request
  - Miss gets latest copy (memory or processor)
- This is an **invalidate protocol**
- **Update protocol**
  - copy data, don't invalidate
  - Sounds good, but wastes bandwidth

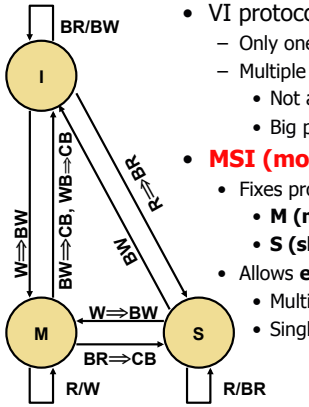
VI Protocol (Write-Back Cache)

Processor 0	Processor 1			
`0: addi r1,accts,r3`				500
`1: ld 0(r3),r4`		V:500		500
`2: blt r4,r2,6`				
`3: sub r4,r2,r4`				
`4: st r4,0(r3)`		V:400		500
`5: call spew_cash`	`0: addi r1,&accts,r3`			
	`1: ld 0(r3),r4`	I:WB	V:400	400
	`2: blt r4,r2,6`			
	`3: sub r4,r2,r4`			
	`4: st r4,0(r3)`		V:300	400
	`5: call spew_cash`			

`ld` by processor 1 generates a BR

- processor 0 responds by WB 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)

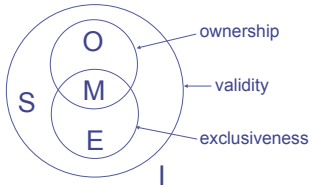
MSI Protocol (Write-Back Cache)

Processor 0	Processor 1			
				500
0: addi r1,accts,r3				
1: ld 0(r3),r4		S:500		500
2: blt r4,r2,6				
3: sub r4,r2,r4				
4: st r4,0(r3)				
5: call spew_cash		M:400		500
	0: addi r1,accts,r3	S:400	S:400	400
	1: ld 0(r3),r4			
	2: blt r4,r2,6			
	3: sub r4,r2,r4			
	4: st r4,0(r3)	I:	M:300	400
	5: call spew_cash			

- `ld` by processor 1 generates a BR
  - processor 0 responds by WB its dirty copy, transitioning to **S**
- `st` by processor 1 generates a BW
  - processor 0 responds by transitioning to **I**

More Generally: MOESI

- [Sweazey & Smith ISCA86]
- M - Modified (dirty)
- O - Owned (dirty but shared) WHY?
- E - Exclusive (clean unshared) only copy, not dirty
- S - Shared
- I - Invalid



- Variants
  - MSI
  - MESI
  - MOSI
  - MOESI

Qualitative Sharing Patterns

- [Weber & Gupta, ASPLOS3]
- Read-Only
- Mostly Read
  - More processors imply more invalidations, but writes are rare
- Migratory Objects
  - Manipulated by one processor at a time
  - Often protected by a lock
  - Usually a write causes only a single invalidation
- Synchronization Objects
  - Often more processors imply more invalidations
- Frequently Read/Written
  - More processors imply more invalidations

False Sharing

- Two (or more) logically separate data share a cache block
  - Modern caches have block sizes of 32-256 bytes (64 bytes typ.)
  - Compilers may co-allocate independent fields
  - Updates may cause block to "ping-pong"
- Example

```
int a[16];
foreach thread i {
   a[i]++;
}
```

- Solution
  - Pad data structure

```
struct {int d; int pad[7]} a[16];
foreach thread i {
     a[i].d++;
}
```

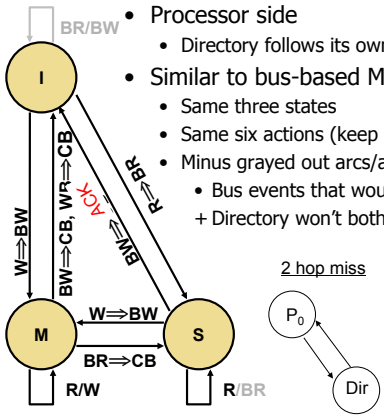

Scalable Cache Coherence

- **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
    - For loosely shared data, only one processor probably has it
  - Replace non-scalable broadcast protocol (spam everyone)…
  - …with scalable **directory protocol** (only spam processors that care)

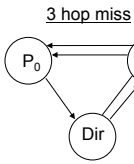
Directory Coherence Protocols

- 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 forwards events to processors
    - Processors only receive events they care about

MSI Directory Protocol



- Processor side
  - Directory follows its own protocol (obvious in principle)
- Similar to bus-based MSI
  - Same three states
  - Same six 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



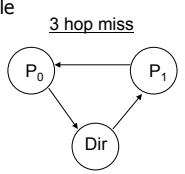
Directory MSI Protocol

Processor 0	Processor 1	P0	P1	Directory
0: addi r1,accts,r3				–:–:500
1: ld 0(r3),r4		S:500		S:0:500
2: blt r4,r2,6				
3: sub r4,r2,r4				
4: st r4,0(r3)		M:400		M:0:500 (stale)
5: call spew_cash	0: addi r1,accts,r3			
	1: ld 0(r3),r4	S:400	S:400	S:0,1:400
	2: blt r4,r2,6			
	3: sub r4,r2,r4			
	4: st r4,0(r3)		M:300	M:1:400
	5: call spew_cash			

- `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 block: get data from memory
    - Bus: 2 hops (P0→memory→P0)
    - Directory: 2 hops (P0→memory→P0)
  - Shared or exclusive block: get data from other processor (P1)
    - Assume cache-to-cache transfer optimization
      - Bus: 2 hops (P0→P1→P0)
      - – Directory: **3 hops** (P0→memory→P1→P0)
    - Occurs frequently with many processors
      - high probability one other processor has it

Today's Outline: Shared Memory Review

- Introduction to Shared Memory
- Cache Coherence
- Synchronization
  - Problem
  - Test-and-set, etc.
- Memory Consistency

The Need for Synchronization

Processor 0	Processor 1			
0: addi r1,accts,r3				500
1: ld 0(r3),r4		S:500		500
2: blt r4,r2,6	0: addi r1,accts,r3	S:500	S:500	500
3: sub r4,r2,r4	1: ld 0(r3),r4	M:400	I:	400
4: st r4,0(r3)	2: blt r4,r2,6	I:	M:400	400
5: call spew_cash	3: sub r4,r2,r4			
	4: st r4,0(r3)			
	5: call spew_cash			

- We're not done, consider the following execution
  - Write-back caches (doesn't matter, though), MSI protocol
- What happened?
  - We got it wrong…
    and coherence had nothing to do with it

The Need for Synchronization

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

		500
S:500		500
S:500	S:500	500
M:400	I:	400
I:	M:400	400

- What really happened?
  - Access to `accts[241].bal` should conceptually be **atomic**
    - Transactions should not be "interleaved"
    - But that's exactly what happened
    - Same thing can happen on a multiprogrammed uniprocessor!
- Solution: **synchronize** access to `accts[241].bal`

Synchronization

- **Synchronization**: second issue for shared memory
  - Regulate access to shared data
  - Software constructs: semaphore, monitor
  - Hardware primitive: **lock**
    - Operations: `acquire(lock)` and `release(lock)`
    - Region between `acquire` and `release` is a **critical section**
    - Must interleave `acquire` and `release`
    - Second consecutive `acquire` will fail (actually it 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) {
   accts[id].bal -= amt;          // critical section
   spew_cash(); }
release(lock);
```

Working Spinlock: Test-And-Set

- ISA provides an atomic lock acquisition instruction
  - Example: **test-and-set**

    `t&s r1,0(&lock)`
    - Atomically executes

      ```
      mov r1,r2
      ld r1,0(&lock)
      st r2,0(&lock)
      ```
    - If lock was initially free (0), acquires it (sets it to 1)
    - If lock was initially busy (1), doesn't change it
  - New acquire sequence

    ```
    A0: t&s r1,0(&lock)
    A1: bnez r1,A0
    ```
  - Similar instructions: **swap**, **compare&swap**, **exchange**, and **fetch-and-add**

Test-and-Set Lock Correctness

```
Processor 0              Processor 1
A0: t&s r1,0(&lock)
A1: bnez r1,#A0          A0: t&s r1,0(&lock)
CRITICAL_SECTION         A1: bnez r1,#A0
                         A0: t&s r1,0(&lock)
                         A1: bnez r1,#A0
```

+ Test-and-set lock actually works
  - Processor 1 keeps spinning

Test-and-Set Lock Performance

```
Processor 1              Processor 2
A0: t&s r1,0(&lock)
A1: bnez r1,#A0          A0: t&s r1,0(&lock)
A0: t&s r1,0(&lock)      A1: bnez r1,#A0
A1: bnez r1,#A0          A0: t&s r1,0(&lock)
                         A1: bnez r1,#A0
```

M:1	I:	1
I:	M:1	1
M:1	I:	1
I:	M:1	1
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

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

```
Processor 1              Processor 2
A0: ld r1,0(&lock)
A1: bnez r1,A0           A0: ld r1,0(&lock)
A0: ld r1,0(&lock)       A1: bnez r1,A0
      // lock released by processor 0
A0: ld r1,0(&lock)       A1: bnez r1,A0
A1: bnez r1,A0           A0: ld r1,0(&lock)
A2: addi r1,1,r1         A1: bnez r1,A0
A3: t&s r1,(&lock)       A2: addi r1,1,r1
A4: bnez r1,A0           A3: t&s r1,(&lock)
CRITICAL_SECTION         A4: bnez r1,A0
                         A0: ld r1,0(&lock)
                         A1: bnez r1,A0
```

S:1	I:	1
S:1	S:1	1
S:1	S:1	1
I:	**I:**	**0**
S:0	I:	0
S:0	S:0	0
S:0	S:0	0
M:1	I:	1
I:	M:1	1
I:	M:1	1
I:	M:1	1
I:	M:1	1

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

A Final Word on Locking

- A single lock for the whole array may restrict parallelism
  - Will force updates to different accounts to proceed serially
  - Solution: one lock per account
  - **Locking granularity**: how much data does a lock lock?
  - A software issue, but one you need to be aware of
    - Also, recall deadlock example…

```
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;
   spew_cash(); }
release(accts[id].lock);
```

Today's Outline: Shared Memory Review

- Motivation for Four-Lecture Course
- Introduction to Shared Memory
- Cache Coherence
- Synchronization
- Memory Consistency

Memory Consistency

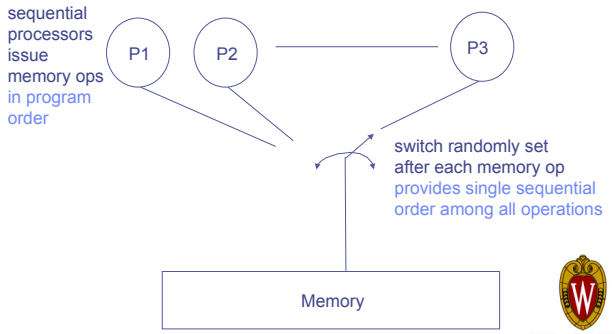
- **Memory coherence**
  - Creates globally uniform (consistent) view…
  - Of **a single memory location** (in other words: cache line)
  - Not enough
    - Cache lines A and B can be individually consistent…
    - But inconsistent with respect to each other
- **Memory consistency**
  - Creates globally 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
    - Maybe P0 has a coalescing write buffer that reorders writes
- Imagine trying to figure out why this code sometimes "works" and sometimes doesn't
- **(some) Real systems** act in this strange manner

Sequential Consistency (SC)



Sufficient Conditions for SC

- Every processor issues memory ops in program order
- Processor must wait for store to complete before issuing next memory operation
- After load, issuing proc waits for load to complete, and store that produced value to complete before issuing next op
- Easily implemented with shared bus.

Relaxed Memory Models

- Motivation 1: Directory Protocols
  - Misses have longer latency
  - Collecting acknowledgements can take even longer
- Motivation 2: (Simpler) Out-of-order processors
  - do cache hits to get to next miss
- 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
- (Most) Relaxed Models
  - PC: Relax ordering from writes to (other proc's) reads
  - RC: Relax all read/write orderings (but add "fences")

Relax Write to Read Order

/* initial A = B = 0 */

P1	P2
A = 1;	B = 1
r1 = B;	r2 = A;

Violates SC, OK in PC

Processor Consistent (PC) Models

Allow r1==r2==0 (precluded by SC)

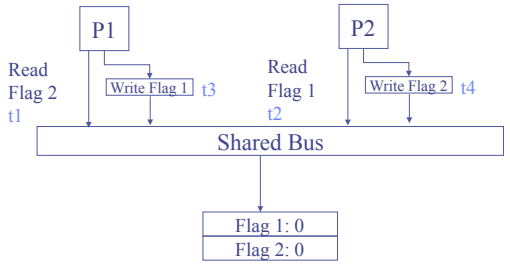
Examples: IBM 370, Sun TSO, & Intel IA-32

Why do this?

Allows FIFO write buffers

Does not astonish programmers (too much)

Write Buffers w/ Read Bypass




```
P1                          P2
Flag 1 = 1                  Flag 2 = 1
if (Flag 2 == 0)            if (Flag 1 == 0)
   critical section            critical section
```

Also Want "Causality"

```
        /* initially all 0 */
  P1          P2                    P3
A = 1;    while (flag1==0) {};   while (flag2==0) {};
flag1 = 1; flag2 = 1;            r3 = A;
```

All commercial models guarantee causality

Why Not Relax All Order?

```
        /* initially all 0 */
  P1                  P2
A = 1;          while (flag == 0); /* spin */
B = 1;                r1 = A;
flag = 1;             r2 = B;
```

Reorder of "A = 1"/"B = 1" or "r1 = A"/"r2 = B"

Via OOO processor, non-FIFO write buffers, delayed directory acknowledgements, etc.

But programmers expect the following order:

"A = 1"/"B = 1" before "flag =1"

"flag != 0" before "r1 = A"/"r2 = B"

Order with "Synch" Operations

```
        /* initially all 0 */
  P1                  P2
A = 1;          while (SYNCH  flag == 0);
B = 1;                r1 = A;
SYNCH flag = 1;     r2 = B;
```

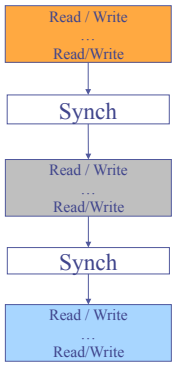
Called "weak ordering" or "weak consistency" (WC)

Alternatively, relaxed consistency (RC) specializes

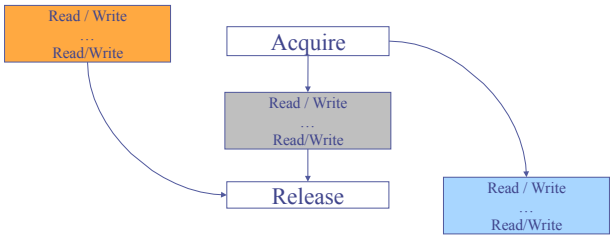
Acquires: force subsequent reads/writes after

Releases: force previous reads/writes before

Weak Ordering Example



Release Consistency Example



Commercial Models use "Fences"

```
/* initially all 0 */
  P1                  P2
A = 1;              while (flag == 0);
B = 1;                  FENCE;
FENCE;                  r1 = A;
flag = 1;               r2 = B;
```

Examples: Compaq Alpha, IBM PowerPC, & Sun RMO

Can specialize fences (e.g., Alpha and RMO)