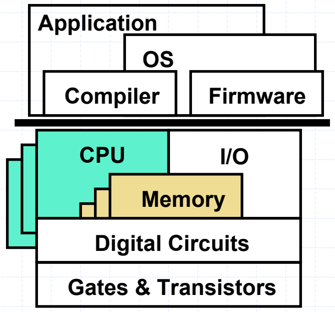
CIS 501
Introduction To Computer Architecture

Unit 11: Shared-Memory Multiprocessors

This Unit: Shared Memory Multiprocessors



- Three issues
  - Cache coherence
  - Synchronization
  - Memory consistency
- Two cache coherence approaches
  - "Snooping" (SMPs): < 16 processors
  - "Directory"/Scalable: lots of processors

Readings

- H+P
  - Chapter 6

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
```

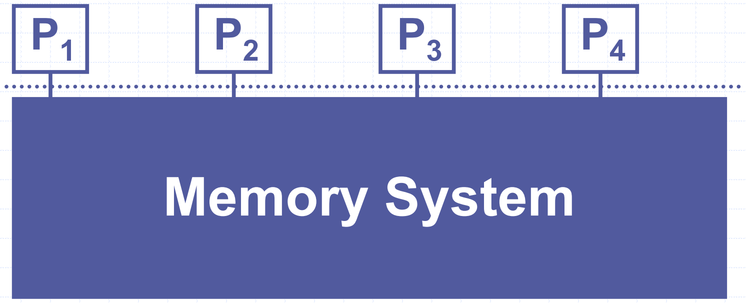
- **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared loosely, dynamically
- Example: database/web server (each query is a thread)
  - `accts` is **shared**, can't register allocate even if it were scalar
  - `id` and `amt` are private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

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

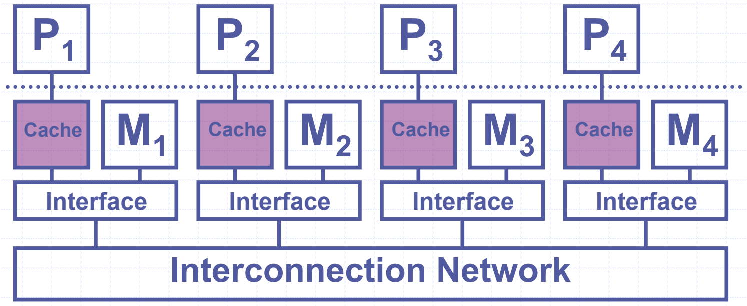
Shared-Memory Multiprocessors

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



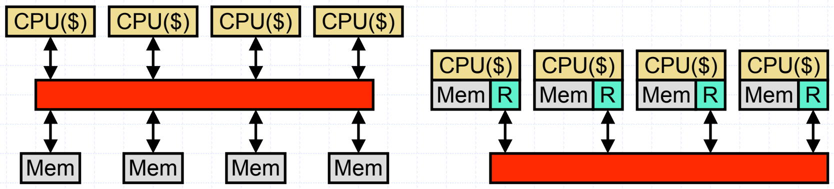
Shared-Memory Multiprocessors

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



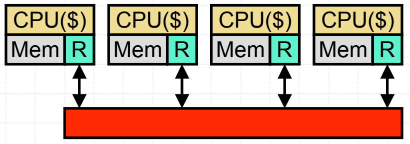
Paired vs. Separate Processor/Memory?

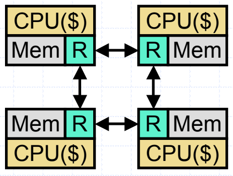
- **Separate processor/memory**
  - **Uniform memory access (UMA):** equal latency to all memory
  - + Simple software, doesn't matter where you put data
  - – Lower peak performance
  - Bus-based UMAs common: **symmetric multi-processors (SMP)**
- **Paired processor/memory**
  - **Non-uniform memory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

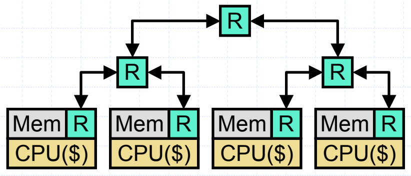
- **Shared network**: e.g., bus (left)
  + Low latency
  – Low bandwidth: doesn't scale beyond ~16 processors
  + Shared property simplifies cache coherence protocols (later)
- **Point-to-point network**: e.g., mesh or ring (right)
  – Longer latency: may need multiple "hops" to communicate
  + Higher bandwidth: scales to 1000s of processors
  – Cache coherence protocols are complex

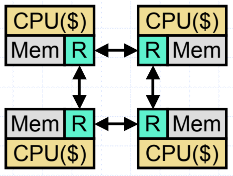




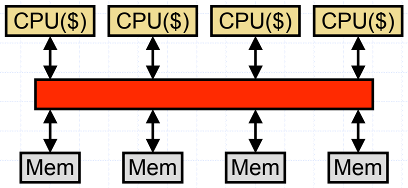
Organizing Point-To-Point Networks

- **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)





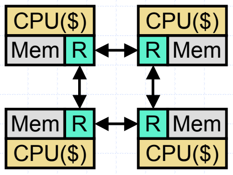
Implementation #1: Snooping Bus MP

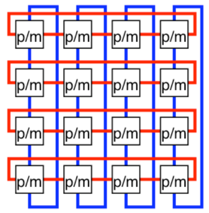


- Two basic implementations

- Bus-based systems
  - Typically small: 2–8 (maybe 16) processors
  - Typically processors split from memories (UMA)
    - Sometimes **multiple processors on single chip (CMP)**
    - **Symmetric multiprocessors (SMPs)**
    - Common, I use one everyday

Implementation #1: 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)**
  - In reality only government (DoD) has MPPs…
    - Companies have much smaller systems: 32–64 processors
    - **Scalable multi-processors**

Issues for Shared Memory Systems

- Three in particular
  - **Cache coherence**
  - Synchronization
  - Memory consistency model
  - Not unrelated to each other
- Different solutions for SMPs and MPPs

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

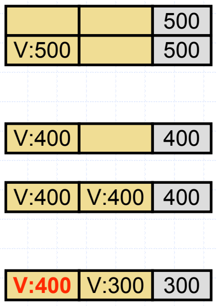
```
Processor 0               Processor 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash         0: addi r1,accts,r3
                          1: ld 0(r3),r4
                          2: blt r4,r2,6
                          3: sub r4,r2,r4
       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

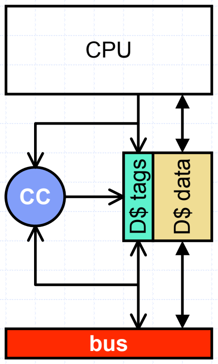


- Scenario II: processors have write-thru caches
  - This time only 2 (different) copies of **accts[241].bal**
  - No problem? What if another withdrawal happens on processor 0?

What To Do?

- Have already seen this problem before with DMA
  - DMA controller acts as another processor
  - Changes cached versions behind processor's back
- Possible solutions
  - No caches? Slow
  - Make shared data uncachable? Still slow
  - Timely flushes and wholesale invalidations? Slow, but OK for DMA
  - **Hardware cache coherence**
    - + Minimal flushing, maximum caching → best performance

Hardware Cache Coherence

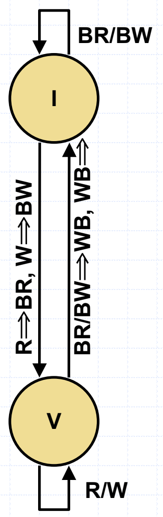


- Absolute coherence
  - All copies have same data at all times
  - – Hard to implement and slow
  - + Not strictly necessary
- **Relative coherence**
  - Temporary incoherence OK (e.g., write-back)
  - As long as all loads get right values
- **Coherence controller**:
  - Examines bus traffic (addresses 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**
  - Protocol relies on this
  - Three processor-side events
    - **R**: read
    - **W**: write
    - **WB**: write-back
  - Two bus-side events
    - **BR**: bus-read, read miss on another processor
    - **BW**: bus-write, write miss or write-back on another processor
- Point-to-point network protocols also exist
  - Called **directories**

VI (MI) Coherence Protocol



- **VI (valid-invalid) protocol**: aka MI
  - Two states (per block)
    - **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

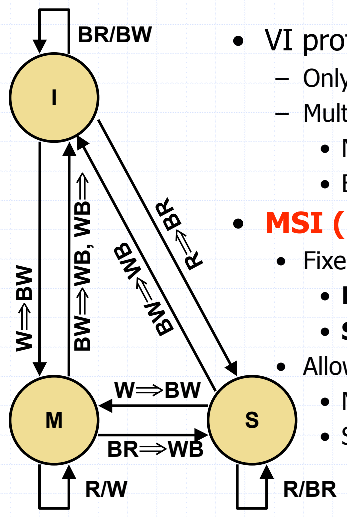
VI Protocol (Write-Back Cache)

```
Processor 0                 Processor 1
0: addi 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		500
I:WB	V:400	400
	V:300	400

- `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)

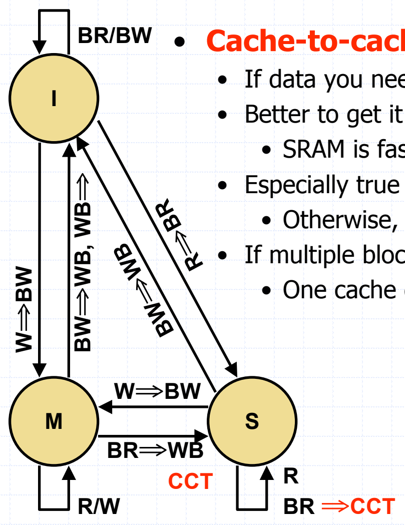
MSI Protocol (Write-Back Cache)

```
Processor 0                 Processor 1
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
S:500		500
M:400		500
S:400	S:400	400
I:	M:300	400

- `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**

A Protocol Optimization



- **Cache-to-cache transfers (CCT)**
  - If data you need is in both memory and other cache…
  - Better to get it from the other cache
    - SRAM is faster than DRAM
  - Especially true if cache block is dirty
    - Otherwise, writeback followed by memory read
  - If multiple blocks have copies, who does CCT?
    - One cache designated as "owner"

Another Protocol Optimization

- Most modern protocols also include **E (exclusive)** state
  - Interpretation: can write to this block, but haven't yet
  - Why is this state useful?

Cache Coherence and Cache Misses

- A coherence protocol can effect a cache's miss rate (%$_{\text{miss}}$)
  - 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 16B cache, 4B blocks, nibble notation

Cache contents (state:address)	Event	Outcome
S:0000, M:0010, S:0020, S:0030		
S:0000, M:0010, S:0020, **M**:0030	Wr:0030	Upgrade Miss
S:0000, M:0010, S:0020, M:0030	BusRd:0000	Nothing
S:0000, M:0010, **I**:0020, M:0030	BusWr:0020	S➔I Invalidation
S:0000, M:0010, I:0020, **S:3030**	Rd:3030	Compulsory Miss
S:0000, M:0010, **S:0020**, S:3030	Rd:0020	**Coherence Miss**
S:0000, M:0010, S:0020, **S:0030**	Rd:0030	Conflict Miss

Cache Coherence and Cache Misses

- 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

- Number of processors also affects coherence misses
  – More processors: more coherence misses

Coherence Bandwidth Requirements

- How much address bus bandwidth does snooping need?
  - Well, coherence events generated on…
    - Misses (only in L2, not so bad)
    - Dirty replacements
- Some parameters
  - 2 GHz CPUs, 2 IPC, 33% memory operations,
  - 2% of which miss in the L2, 50% of evictions are dirty
  - (0.33 * 0.02) + (0.33 * 0.02 * 0.50)) = 0.01 events/insn
  - 0.01 events/insn * 2 insn/cycle * 2 cycle/ns = 0.04 events/ns
  - Request: 0.04 events/ns * 4 B/event = 0.16 GB/s = 160 MB/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!!
  - Yes, you can use multiple buses… but that hinders global ordering

More Coherence Bandwidth

- Bus bandwidth is not the only problem
- Also **processor snooping bandwidth**
  - Recall: snoop implies matching address against current cache tags
    - Just a tag lookup, not data
  - 0.01 events/insn * 2 insn/cycle = 0.01 events/cycle per processor
  - With 16 processors, each would do 0.16 tag lookups per cycle
    - ± Add a port to the cache tags … OK
  - With 128 processors, each would do 1.28 tag lookups per cycle
    - If caches implement **inclusion** (L1 is strict subset of L2)
    - – Additional snooping ports only needed on L2, still bad though

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

Scalable Cache Coherence

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

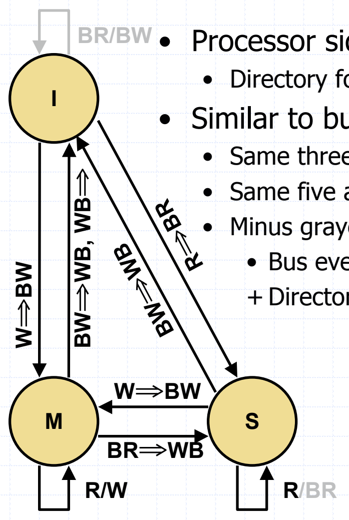
- Part I: **bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)…
  - …with scalable bandwidth one (point-to-point network, e.g., mesh)

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

Directory Coherence Protocols

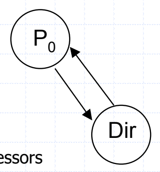
- Observe: physical address space statically partitioned
  - + Can easily determine which memory module holds a given line
    - That memory module sometimes called "**home**"
  - – Can't easily determine which processors have line in their caches
  - Bus-based protocol: broadcast events to all processors/caches
    - ± 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

MSI Directory Protocol

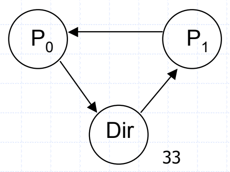


- Processor side
  - Directory follows its own protocol (obvious in principle)
- Similar to bus-based MSI
  - Same three states
  - Same five actions (keep BR/BW names)
  - Minus grayed out arcs/actions
    - Bus events that would not trigger action anyway
    - + Directory won't bother you unless you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

2 hop miss



3 hop miss



Directory MSI Protocol

Processor 0
```
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
```

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
```

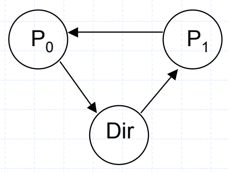
P0	P1	Directory
		–:–:500
S:500		S:0:500
M:400		M:0:500 (stale)
S:400	S:400	S:0,1:400
	M:300	M:1:400

- `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

3 hop miss



- 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)
    - 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 line must appear atomic
  - Bus: all processors see all requests in same order
    - Atomicity automatic
  - Point-to-point network: requests may arrive in different orders
    - Directory has to enforce atomicity 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
  - – Extremely complicated in practice

Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
  - Can be easily combined to form SMPs
  - E.g., Intel Pentium4 Xeon
- Larger scale (directory) systems built from smaller SMPs
  - 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 2-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 … **data race**

- **Token Coherence (TC)**
  - An unordered broadcast snooping protocol … without data races
  - Interesting, but won't talk about here

One Down, Two To Go

- Coherence only one part of the equation
  - Synchronization
  - Consistency

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

- 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) {            // critical section
   accts[id].bal -= amt;
   spew_cash(); }
release(lock);
```

Spin Lock Strawman (Does not work)

- **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
```

Spin Lock Strawman (Does not work)

```
Processor 0              Processor 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
  - Definition of "squeezing toothpaste"
  - Note, `release` is trivially atomic

Better Implementation: SYSCALL Lock

```
ACQUIRE_LOCK:
A0: enable_interrupts
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: jr $r31
```

- Implement lock in a SYSCALL
  - Kernel can control interleaving by disabling interrupts
  - + Works…
  - – But only in a multi-programmed uni-processor
  - – Hugely expensive in the common case, lock is free

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
    ```
  - Also known as **swap**, **exchange**, or **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)`		S:1	I:	1
`A1: bnez r1,A0`	`A0: ld r1,0(&lock)`	S:1	S:1	1
`A0: ld r1,0(&lock)`	`A1: bnez r1,A0`	S:1	S:1	1
`// lock released by processor 0`		I:	I:	0
`A0: ld r1,0(&lock)`	`A1: bnez r1,A0`	S:0	I:	0
`A1: bnez r1,A0`	`A0: ld r1,0(&lock)`	S:0	S:0	0
`A2: addi r1,1,r1`	`A1: bnez r1,A0`	S:0	S:0	0
`A3: t&s r1,(&lock)`	`A2: addi r1,1,r1`	M:1	I:	1
`A4: bnez r1,A0`	`A3: t&s r1,(&lock)`	I:	M:1	1
`CRITICAL_SECTION`	`A4: bnez r1,A0`	I:	M:1	1
	`A0: ld r1,0(&lock)`	I:	M:1	1
	`A1: bnez r1,A0`	I:	M:1	1

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

Queue Locks

- Test-and-test-and-set locks can still perform poorly
  - If lock is contended for by many processors
  - Lock release by one processor, creates "free-for-all" by others
  - – Network gets swamped with `t&s` requests
- **Queue lock**
  - When lock is released by one processor...
  - Directory doesn't notify (by invalidations) **all** waiting processors
  - Instead, chooses one and sends invalidation only to it
    - Others continue spinning locally, unaware lock was released
  - Effectively, directory passes lock from one processor to the next
  - + Greatly reduced network traffic

Queue Lock Performance

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

- Processor 0 releases lock, **informs only processor 1**

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

```
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);
```

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
- **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**

Enforcing SC

- What does it take to enforce SC?
  - Definition: all loads/stores globally ordered
  - Translation: coherence events of all loads/stores globally ordered
- **When do coherence events happen naturally?**
  - On cache access
  - For stores: retirement → in-order → good
    - No write buffer? Yikes, but OK with write-back D$
  - For loads: execution → out-of-order → bad
    - – No out-of-order execution? Double yikes
- Is it true that multi-prcessors cannot be out-of-order?
  - – That would be really bad
  - Out-of-order is needed to hide cache miss latency
  - And multi-processors not only have more misses…
  - … but miss handling takes longer (coherence actions)

SC + OOO

- Recall: opportunistic load scheduling in a uni-processor
  - **Loads issue speculatively relative to older stores**
  - Stores scan for younger loads to same address have issued
  - Find one? Ordering violation → flush and restart
  - In-flight loads effectively "snoop" older stores from same process
- SC + OOO can be reconciled using **same technique**
  - Write bus requests from other processors snoop in-flight loads
  - Think of MOB as extension of the cache hierarchy
  - MIPS R10K does this
- SC implementable, but overheads still remain:
  - Write buffer issues
  - Complicated ld/st logic

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 IA-32, 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
    - **Use synchronization library, don't write your own**

SC + OOO vs. WO

- Big debate these days
  - Is SC + OOO equal to WO performance wise?
  - And if so, which is preferred?
- Another hot button issue
  - Can OOO be used to effectively speculate around locks?
  - Short answer: yes

Multiprocessors Are Here To Stay

- Moore's law is making the multiprocessor a commodity part
  - 500M transistors on a chip, what to do with all of them?
  - Not enough ILP to justify a huge uniprocessor
  - Really big caches? t_{hit} increases, diminishing $\%_{miss}$ returns
- **Chip multiprocessors (CMPs)**
  - Multiple full processors on a single chip
  - Example: IBM POWER4: two 1GHz processors, 1MB L2, L3 tags
- Multiprocessors a huge part of computer architecture
  - Multiprocessor equivalent of H+P is 50% bigger than H+P
  - Another entire cores on multiprocessor architecture
    - Hopefully every other year or so (not this year)

Multiprocessing & Power Consumption

- Multiprocessing can be very power efficient
- Recall: dynamic voltage and frequency scaling
  - Performance vs power is NOT linear
  - Example: Intel's Xscale
    - 1 GHz → 200 MHz reduces energy used by 30x
- Impact of parallel execution
  - What if we used 5 Xscales at 200Mhz?
  - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
    - 5 cores * 1/30th = 1/6th
- Assumes parallel speedup (a difficult task)
  - Remember Ahmdal's law

Shared Memory Summary

- Shared-memory multiprocessors
  - + Simple software: easy data sharing, handles both DLP and TLP
  - – Complex hardware: must provide illusion of global address space
- Two basic implementations
  - **Symmetric (UMA) multi-processors (SMPs)**
    - Underlying communication network: bus (ordered)
    - + Low-latency, simple protocols that rely on global order
    - – Low-bandwidth, poor scalability
  - **Scalable (NUMA) multi-processors (MPPs)**
    - Underlying communication network: point-to-point (unordered)
    - + Scalable bandwidth
    - – Higher-latency, complex protocols

Shared Memory Summary

- Three aspects to global memory space illusion
  - **Coherence**: consistent view of individual cache lines
    - Absolute coherence not needed, relative coherence OK
    - VI and MSI protocols, cache-to-cache transfer optimization
    - Implementation? SMP: snooping, MPP: directories
  - **Synchronization**: regulated access to shared data
    - Key feature: atomic lock acquisition operation (e.g., `t&s`)
    - Performance optimizations: test-and-test-and-set, queue locks
  - **Consistency**: consistent global view of all memory locations
    - Programmers intuitively expect sequential consistency (SC)
      - Global interleaving of individual processor access streams
      - – Not naturally provided by coherence, needs extra stuff
    - Weaker ordering: consistency only for synchronization points