Programming With Locks Is Tricky

- Multicore processors are the way of the foreseeable future
  - thread-level parallelism anointed as parallelism model of choice
  - Just one problem…
- Writing lock-based multi-threaded programs is tricky!
- 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*
    - And that's the whole point, unfortunately
  - Selecting the "right" kind of lock for performance
    - Spin lock, queue lock, ticket lock, read/writer lock, *etc.*
  - **Locking granularity issues**

Goldibear and the 3 Locks

- **Coarse-grain locks**: **correct, but slow**
  - one lock for entire database
  - \+ Easy to make correct: no chance for unintended interference
  - – Limits parallelism: no two critical sections can proceed in parallel
- **Fine-grain locks**: **parallel, but difficult**
  - multiple locks, one per record
  - \+ Fast: critical sections (to different records) can proceed in parallel
  - – Difficult to make correct: easy to make mistakes
- **Multiple locks**: **just right?** (sorry, no fairytale ending)
  - acct-to-acct transfer: must acquire both `id_from`, `id_to` locks
  - Simultaneous transfers 241 → 37 and 37 → 241
  - **Deadlock**: circular wait for shared resources
  - **Solution:** Always acquire multiple locks in same order
    - Just another thing to keep in mind when programming

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…

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 (or was a few years ago)
- No fewer than eight research projects:
  - Brown, Stanford, MIT, Wisconsin, Texas, Rochester, Intel, Penn

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
- **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`

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();
```

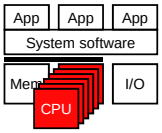

Transactional Memory: End

`end_transaction`

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

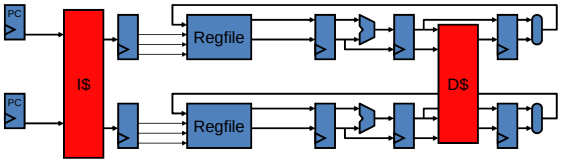
```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

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


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

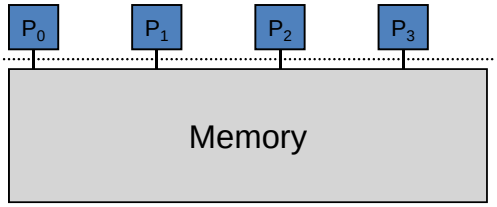


- What if we don't want to share the L1 caches?
  - Bandwidth and latency issue
- Solution: use per-processor ("private") caches
  - Coordinate them with a ***Cache Coherence Protocol***

Shared-Memory Multiprocessors

Conceptual model

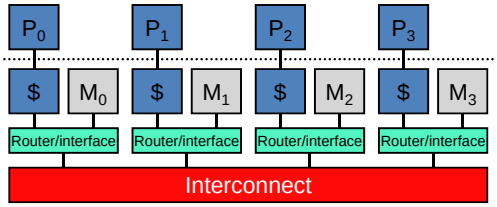
- The shared-memory abstraction
- Familiar and feels natural to programmers
- Life would be easy if systems actually looked like this…



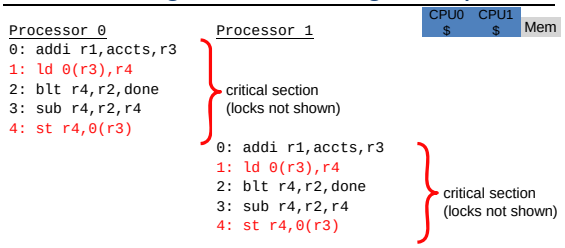
Shared-Memory Multiprocessors

…but systems actually look more like this

- Processors have caches
- Memory may be physically distributed
- Arbitrary interconnect

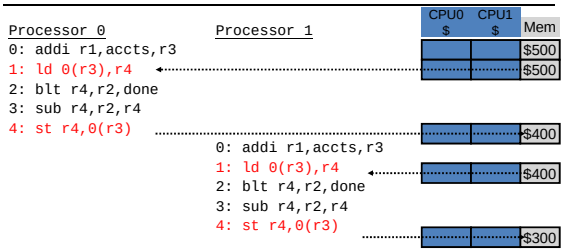


Revisiting Our Motivating Example



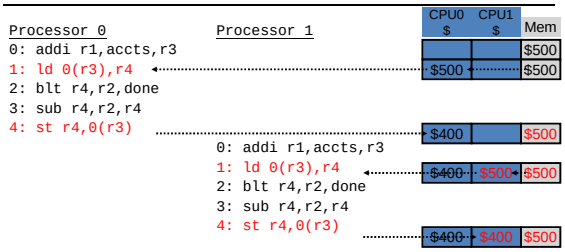
- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `$r3`)

No-Cache, No-Problem



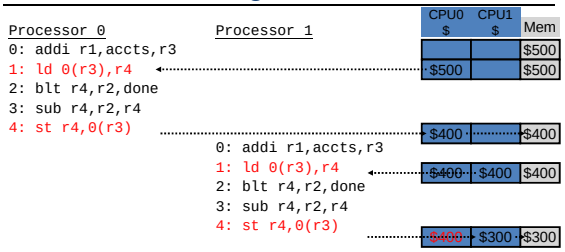
- Scenario I: processors have no caches
  - No problem

Cache Incoherence



- 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



- 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"
- 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

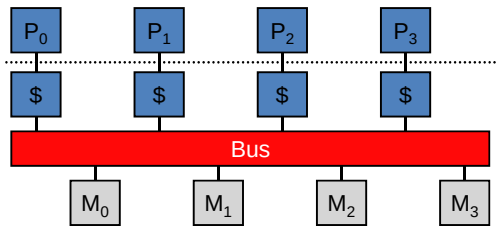
Bus-based Multiprocessor

Simple multiprocessors use a bus

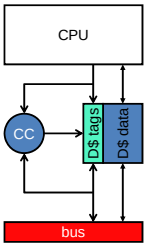
- **All** processors see **all requests** at the **same time**, same order

Memory

- Single memory module, **-or-**
- Banked memory module

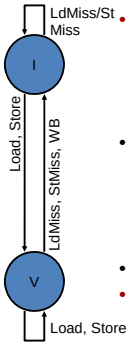


Hardware Cache Coherence



- **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
  - **Ld**: load
  - **St**: store
  - **WB**: write-back
- Two remote-initiated events
  - **LdMiss**: read miss from ***another*** processor
  - **StMiss**: 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)
  - If *you* load/store a block: transition to **V**
  - If anyone *else* 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

* M=modified, comes later

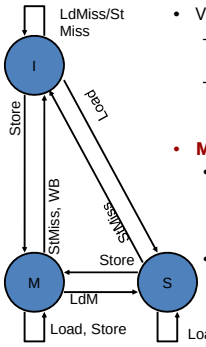
VI Protocol (Write-Back Cache)

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

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

`ld` by processor 1 generates an "other load miss" event (LdMiss)
- 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 r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)
                           0: addi r1,accts,r3
                           1: ld 0(r3),r4
                           2: blt r4,r2,done
                           3: sub r4,r2,r4
                           4: st r4,0(r3)
```

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

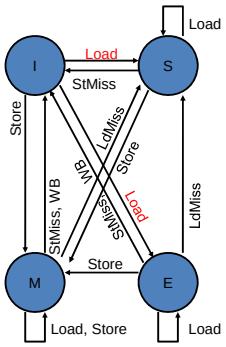
`ld` by processor 1 generates a "other load miss" event (LdMiss)
- Processor 0 responds by sending its dirty copy, transitioning to **S**

`st` by processor 1 generates a "other store miss" event (StMiss)
- Processor 0 responds by transitioning to **I**

Cache Coherence and Cache Misses

- Coherence introduces two new kinds of cache misses
  - **Upgrade miss**
    - On stores to read-only blocks
    - Delay to acquire write permission to read-only block
  - **Coherence miss**
    - Miss to a block evicted by another processor's requests
- Making the cache larger…
  - Doesn't reduce these type of misses
  - As cache grows large, these sorts of misses dominate
- **False sharing**
  - Two or more processors sharing parts of the same block
  - But *not* the same bytes within that block (no actual sharing)
  - Creates pathological "ping-pong" behavior
  - Careful data placement may help, but is difficult

MSI → MESI: *Exclusive Clean* Protocol



Most modern protocols also include **E (exclusive)** state

- "I have the only cached copy, and it's a **clean** copy"
- Why is this state useful?

Load transitions to E if no other processors is caching the block, otherwise S

Exclusive Clean Protocol Optimization

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

```
Processor 1
```

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

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