

CSE 560

Computer Systems Architecture

Multicores

(Shared Memory Multiprocessors)

1

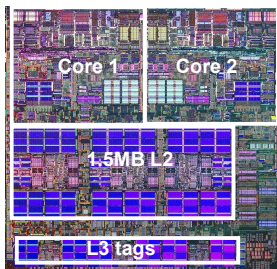
Multiplying Performance

- 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
 - \$\$\$



2

Multicore: Mainstream Multiprocessors



Why multicore? What else would you do with 1 billion transistors?

- Multicore chips**
 - IBM Power5**
 - Two 2+GHz PowerPC cores
 - Shared 1.5 MB L2, L3 tags
 - AMD Quad Phenom
 - Four 2+ GHz cores
 - Per-core 512KB L2 cache
 - Shared 2MB L3 cache
 - Intel Core i7 Quad
 - Four cores, private L2s
 - Shared 6 MB L3
 - Sun Niagara
 - 8 cores, each 4-way threaded
 - Shared 2MB L2, shared FP
 - For servers, not desktop

3

3

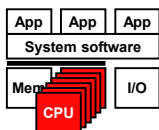
Application Domains for Multiprocessors

- 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**

4

4

This Unit: Shared Memory Multiprocessors



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

5

5

Identifying Parallelism

Consider

```
for (I = 0; I < 10000; I++)
  C[I] = A[I] * B[I];
```

or

```
struct acct_t { int balance; };
struct acct_t accounts[MAX_ACCT]; // current balances

struct trans_t { int id; int amount; };
struct trans_t transactions[MAX_TRANS]; // debit amounts

for (i = 0; i < MAX_TRANS; i++) {
  int id = transactions[i].id;
  int amount = transactions[i].amount;
  if (accounts[id].balance >= amount)
  {
    accounts[id].balance -= amount;
  }
}
```

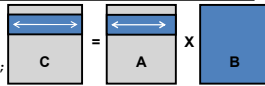
Can we do these in parallel?

6

6

Example: Parallelizing Matrix Multiply

```
for (I = 0; I < 100; I++)
  for (J = 0; J < 100; J++)
    for (K = 0; K < 100; K++)
      C[I][J] += A[I][K] * B[K][J];
```



How to parallelize matrix multiply?

- Replace outer “for” loop with “**parallel_for**”
- Support by many parallel programming environments

Implementation: give each of N processors loop iterations

```
int start = (100/N) * my_id();
for (I = start; I < start + 100/N; I++)
  for (J = 0; J < 100; J++)
    for (K = 0; K < 100; K++)
      C[I][J] += A[I][K] * B[K][J];
```

Each processor runs copy of loop above

- Library provides **my_id()** function

7

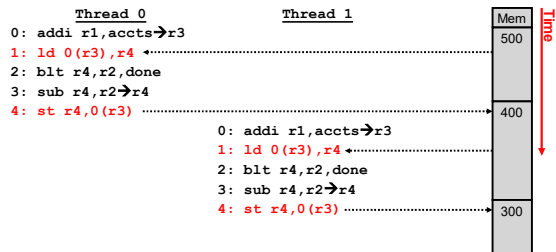
Example: Bank Accounts

```
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id, amt;
if (accts[id].bal >= amt)
{
  accts[id].bal -= amt;
}
r1 = id
r2 = amt
r3 = entry addr
r4 = bal
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)
```

- Example of **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

8

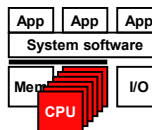
An Example Execution



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

9

Roadmap Checkpoint



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

10

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 OS) 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 an operating systems course

11

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)

12

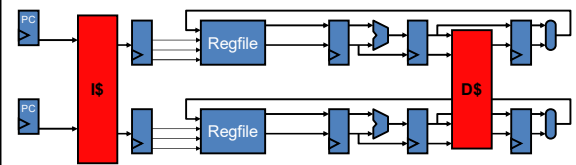
Shared Memory Implementations

- **Multiplexed uniprocessor**
 - Runtime system and/or OS occasionally pre-empt & swap threads
 - Interleaved, but no parallelism
- **Hardware multithreading** (previous unit)
 - 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 support the shared memory programming model**

13

13

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?

14

14

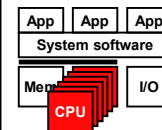
Shared Memory Issues

- Three in particular, not unrelated to each other
- Synchronization
 - How to regulate access to shared data?
 - How to implement critical sections?
- Cache coherence
 - How to make writes to one cache "show up" in others?
- Memory consistency model
 - How to keep programmer sane & let hw optimize?
 - How to reconcile shared memory with store buffers?

15

15

Roadmap Checkpoint

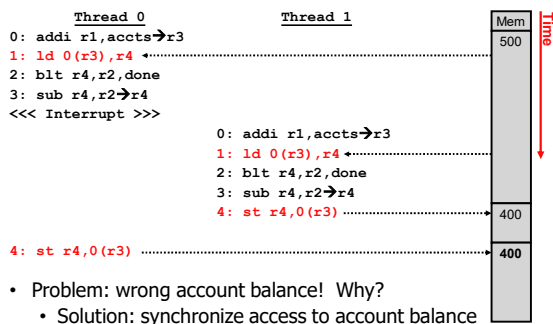


- Thread-level parallelism (TLP)
- Shared memory model
 - Multiplexed uniprocessor
 - Hardware multithreading
 - Multiprocessing
- Synchronization
 - Lock implementation
 - Locking gotchas
- Cache coherence
 - Bus-based protocols
 - Directory protocols
- Memory consistency models

16

16

A Problem Execution



17

17

Synchronization:

- **Synchronization**: a key issue for shared memory
 - Regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: "semaphore" or "mutex")
 - Operations: **acquire(lock)** and **release(lock)**
 - Region between **acquire** and **release** is a **critical section**
 - Must interleave **acquire** and **release**
 - Interfering **acquire** will block
- Another option: **Barrier synchronization**
 - Blocks until all threads reach barrier, used at end of "parallel_for"

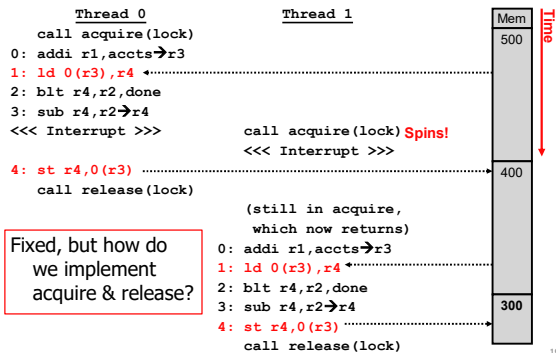
```

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

18

18

A Synchronized Execution



19

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)

```

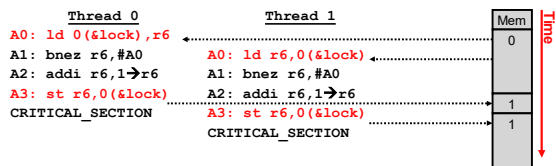
```

• release(lock): lock = 0;
R0: st r0,0(&lock) // r0 holds 0

```

20

Strawman Lock (Incorrect)



- 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

21

A Correct Implementation: SYSCALL Lock

```

ACQUIRE_LOCK:
A1: disable_interrupts
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 hardware multithreading or a multiprocessor...

22

Better Spin Lock: Use Atomic Swap

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

```

swap r1,0(&lock)

```

 - Atomically executes:

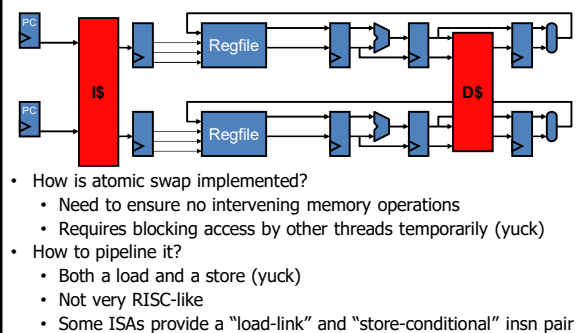

```

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
- Ensures lock held by **at most one thread**
 - Other variants: **exchange**, **compare-and-swap**, **test-and-set (t&s)**, or **fetch-and-add**

23

Atomic Update/Swap Implementation



24

RISC Test-And-Set

- **swap**: a load and store in one insn is not very "RISC"
 - Broken up into micro-ops, but then how is it made atomic?
- **ll/sc**: load-locked / store-conditional
 - Atomic load/store pair


```
ll r1,0(&lock)
// potentially other insns
sc r2,0(&lock)
```
 - On **ll**, processor remembers address...
 - ...And looks for writes by other processors
 - If write is detected, next **sc** to same address is annulled
 - Sets failure condition

25

25

Lock Correctness

```

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

+ Lock actually works...

- Thread 1 keeps spinning
- Sometimes called a "test-and-set lock"
 - Named after the common "test-and-set" atomic instruction

26

26

"Test-and-Set" Lock Performance

```

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

- ...but performs poorly
- Consider 3 processors rather than 2
- P2 (not shown) has the lock and is in the critical section
- But what are P0 and P1 doing in the meantime?
 - Loops of **swap**, each of which includes a **st**
 - Repeated stores by multiple processors costly (more in a bit)
 - Generating a ton of useless interconnect traffic

27

27

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: swap r1,0(&lock)
A4: bnez r1,A0
```
- Within each loop iteration, before doing a **swap**
 - Spin doing a simple test (**ld**) to see if lock value has changed
 - Only do a **swap (st)** if lock is actually free
- Processors can spin on a busy lock locally (in their own cache)
 - + Less unnecessary interconnect traffic
- Note: test-and-test-and-set is *not* a new instruction!
 - Just different software

28

28

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
 - Interconnect gets swamped with **t&s** requests
 - **Software queue lock**
 - Each waiting processor spins on a different location (a queue)
 - When lock is released by one processor...
 - Only the next processor sees its location go "unlocked"
 - Others continue spinning locally, unaware lock was released
 - Effectively, passes lock from one processor to the next, in order
- + Greatly reduced network traffic (no mad rush for the lock)
- + Fairness (lock acquired in FIFO order)
- Higher overhead in case of no contention (more instructions)
- Poor performance if one thread gets swapped out

29

29

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**

30

30

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

31

31

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

32

32

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"

33

33

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

34

34

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`

35

35

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();
```

36

36

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();
```

37