Review

- Last Class:
  - Lecture record: Formal loop transformation.
- Today's class:
  - Synchronization in multi-core.
- Announcement and reminder
  - HW3 dues tonight

Shared Memory multiProcessor (SMP)

- Q1 – Single memory address space shared by all cores
- Q2 – Cores coordinate/communicate through shared variables in memory (via loads and stores)
  1. Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one core at a time (used to implement critical sections)
  2. Caches must be kept coherent (cores have the same shared data)
- SMPs come in two styles
  1. Uniform memory access (UMA) multiprocessors
  2. Nonuniform memory access (NUMA) multiprocessors
- Programming NUMAs is the harder of the two
- But, NUMAs can scale to larger sizes and have lower latency to local memory

Synchronization

- Need to be able to coordinate processes working on a common task; sharing data must be coordinated carefully
  - Lock variables (semaphores) are used to coordinate or synchronize processes
    - mutual exclusion – restrict data access to one core at a time
    - sequential ordering – must complete the first operation before the second operation can begin

1. Need an architecture-supported arbitration mechanism to decide which core gets access to the lock variable
   - Single bus provides an arbitration mechanism, since the bus is the only path to memory – the core that gets the bus wins
2. Need an architecture-supported operation that locks the variable
   - Locking can be done via an atomic swap operation which allows a core to both read a location and set it to the locked state – test-and-set – in the same bus operation

Synchronization

- Synchronization mechanisms are typically built with user-level software routines that rely on hardware-supplied synchronization instructions
- Synchronization can be a *performance bottleneck*
- One typical hardware support is atomic exchange
  - Interchanges a value in a register for a value in memory
  - Assume we want to build a simple lock where **0** is used to indicate the lock is free and **1** is used to indicate the lock is unavailable
  - A processor performs an exchange of 1, which is in a register, with the memory address corresponding to the lock
  - The value read (after the exchange)
    - 1: the lock is unavailable (some other processor is currently using it)
    - 0: we got the lock

- Key property: the operation is atomic!

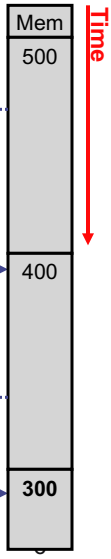
Synchronization:

- Regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: "semaphore")
  - 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;
void debit(int id, int amt):
  acquire(lock);                          critical section
  if (accts[id].bal >= amt) {
     accts[id].bal -= amt;
  }
  release(lock);
```


A Synchronized Execution

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



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

Strawman Lock (Incorrect)

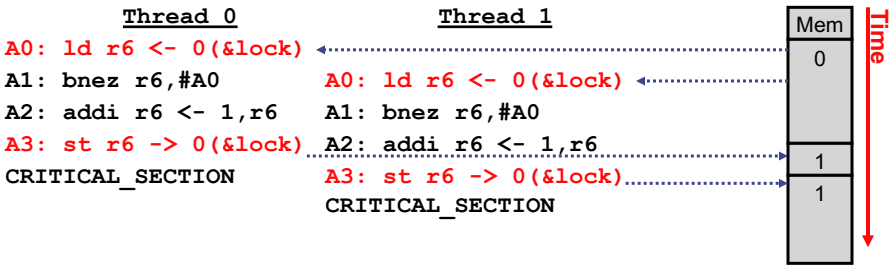
- **Spin lock**: software lock implementation
  - `acquire(lock): while (lock != 0) {} lock = 1;`
    - "Spin" while lock is 1, wait for it to turn 0

```
A0:  ld r6 <- 0(&lock)
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
```


Incorrect Lock Implementation



- 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!**

Correct Spin Lock: Compare and Swap

- ISA provides an atomic lock acquisition instruction
  - Example: **atomic compare-and-swap (CAS)**

```
cas r3 <- r1,r2,0(&lock)
```

  - Atomically executes:

```
ld r3 <- 0(&lock)
if r3 == r2:
  st r1 -> 0(&lock)
```

- New acquire sequence

```
A0: cas r3 <- 1,0,0(&lock)
A1: bnez r3,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**

CAS Implementation

- How is CAS implemented?
  - Need to ensure no intervening memory operations
  - Requires blocking access by other threads temporarily

- How to pipeline it?
  - Both a load and a store
  - Not very RISC-like
- `CAS`: a load+branch+store in one insn is not very "RISC"
  - Broken up into micro-ops, but then how is it made atomic?

Synchronization

- An alternative (as in MIPS) is to have a pair of instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instructions were atomic
- Load Linked and Store Conditional
  - Used in sequence
  - Load-link returns the current value of a memory location
  - Subsequent store-conditional to the same memory location will store a new value only if **no updates have occurred** to that location since the load-link.
  - Load Linked returns the initial value
  - Store Conditional returns 1 only if it succeeds

Atomic Exchange Support

- Atomic exchange (atomic swap) – interchanges a value in a register for a value in memory atomically, i.e., as one operation (instruction)
  - Implementing an atomic exchange would require **both** a memory read and a memory write in a single, uninterruptable instruction
- MIPS provides `ll/sc`: load-linked / store-conditional
  - Atomic load/store pair

```
ll r2,0(&lock)       #load linked
// potentially other load/store's by other cores
sc r1,0(&lock)       #store conditional (returns a value)
```

  - On `ll`, the SRAM cache controller (or DRAM memory controller) remembers the core id and the load address (in a link register)
    - And watches for **stores** by other cores (or for any exceptions)
  - If a store by another core to the same address is detected or a context switch occurs, the `sc` fails (i.e., the store is not performed)
    - the `sc` returns a 1 to the core if the store was successful, 0 on fail
  - Store also fails if a context switch occurs between the two instr's

Atomic Exchange with `ll` and `sc`

- If the contents of the memory location specified by the `ll` are changed (by some other core) before the `sc` to the same address executes, the `sc` fails (returns a zero)

```
lock: add $t0, $zero, $s4      #$t0=$s4 (exchange value)
      ll  $t1, 0($s1)          #load memory value to $t1
       // potentially other load/store instr's executed by other cores
      sc  $t0, 0($s1)          #try to store the exchange
                               #value to memory, if fail
                               #$t0 will be 0, if succeed
                               #$t0 will be 1
      beq $t0, $zero, lock     #try again on failure
      add $s4, $zero, $t1      #put exchange value in $s4
```

If the `sc` fails (does not succeed in storing the exchange value into memory) and returns a 0 in `$t0`, the code sequence tries again

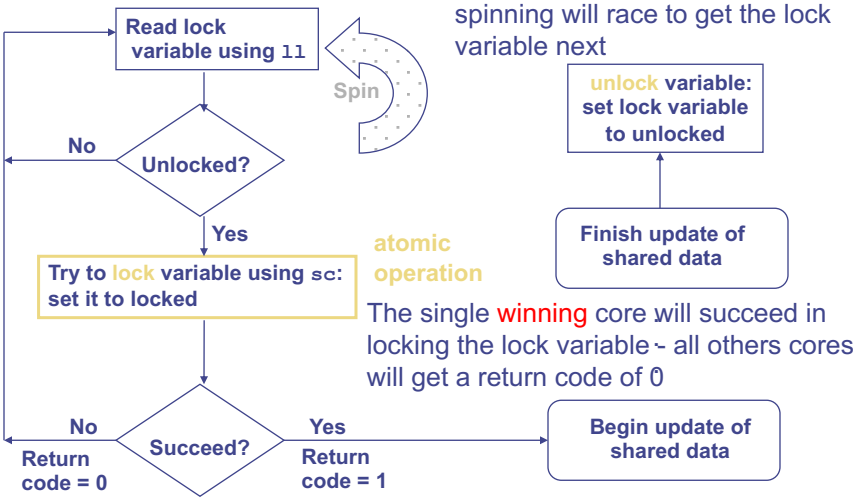
On success, the contents of `$s4` and the memory location specified by `$s1` have been atomically exchanged

Atomic Exchange Example

- Assume all loads and stores (including `ll` and `sc`) hit in the L1 cache

Core0	Core1	Cycle	Core0		Mem ($s1)	Core1	
			$t0	$t1		$t0	$t1
		0	30	40	55	10	11
	`ll $t1,0($s1)`	1					**55**
`ll $t1,0($s1)`		2		**55**			
	`sc $t0,0($s1)`	3			**10**	**1**	
`sc $t0,0($s1)`		4	**0**		**10**		

Spin Lock Synchronization



On unlock, all the cores spinning will race to get the lock variable next

The single winning core will succeed in locking the lock variable - all others cores will get a return code of 0

Lock Correctness

Thread 0	Thread 1
A0: cas r1 <- 1,0,0(&lock)	
A1: bnez r1,#A0	A0: cas r1 <- 1,0,0(&lock)
CRITICAL_SECTION	A1: bnez r1,#A0
	A0: cas r1 <- 1,0,0(&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

"Test-and-Set" Lock Performance

```
Thread 0                        Thread 1
A0: cas r1 <- 1,0,0(&lock)
A1: bnez r1,#A0                 A0: cas r1 <- 1,0,0(&lock)
A0: cas r1 <- 1,0,0(&lock)      A1: bnez r1,#A0
A1: bnez r1,#A0                 A0: cas r1 <- 1,0,0(&lock)
                                A1: bnez r1,#A0
```

- …but performs poorly
  - Consider 3 processors rather than 2
  - Processor 2 (not shown) has the lock and is in the critical section
  - But what are processors 0 and 1 doing in the meantime?
    - Loops of `cas`, each of which includes a `st`
    - Repeated stores by multiple processors costly
    - Generating a ton of useless interconnect 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: cas r1 <- r1,0,0(&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

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 `cas` requests
- **Software queue lock**
  - Each waiting processor spins on a different location (a queue)
  - When lock is released by one processor...
    - Only the next processors 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 is descheduled by O.S.

Summing 100,000 Numbers on 100 Core SMP

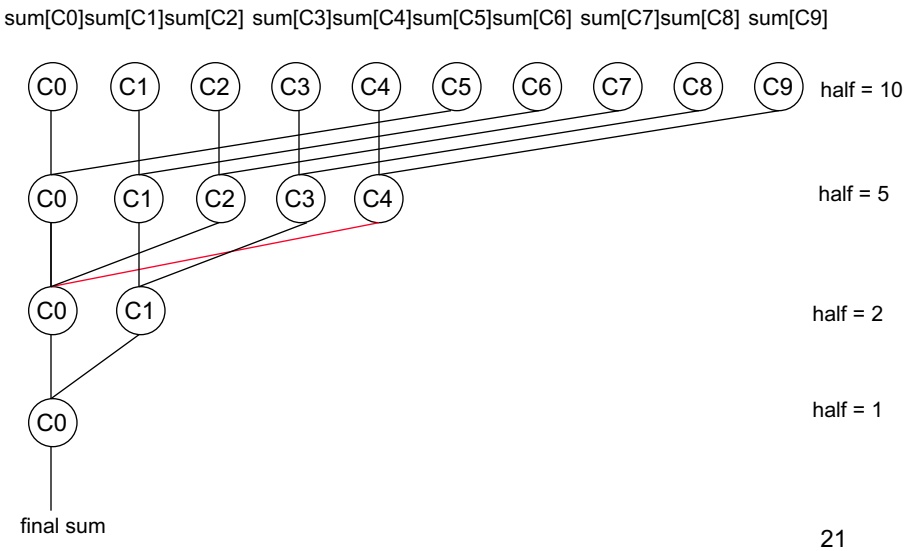
- Cores start by running a thread loop that sums their subset of vector `A` numbers (vectors `A` and `sum` are shared variables, `Cn` is the core's number, `i` is a private variable)

```
sum[Cn] = 0;
for (i = 1000*Cn; i< 1000*(Cn+1); i = i + 1)
  sum[Cn] = sum[Cn] + A[i];
```

- The cores' threads then coordinate in adding together the partial sums (`half` is a private variable initialized to `100` (the number of cores)) – reduction

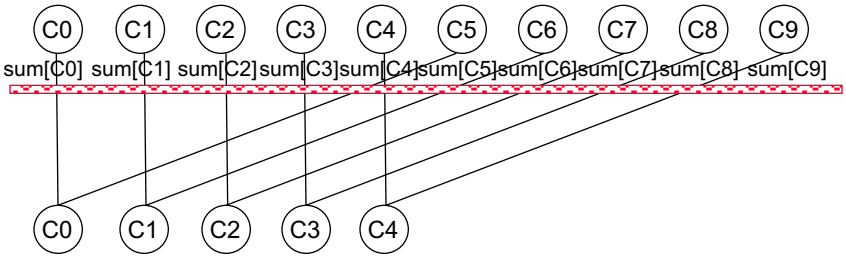
```
repeat
  synch();                /*synchronize first
  if (half%2 != 0 && Cn == 0)
      sum[0] = sum[0] + sum[half-1];
  half = half/2
  if (Cn<half) sum[Cn] = sum[Cn] + sum[Cn+half]
until (half == 1);        /*final sum in sum[0]
```

An Example with 10 Cores (10 Threads)



An Example with 10 Cores (10 Threads)

- `synch()`: Cores (theads) must synchronize before the "consumer" core (thread) tries to read the results from the memory location written by the "producer" core (thread)
  - Barrier synchronization – cores wait at the barrier, not proceeding until every core has reached it



Barrier Synchronization

- A barrier synchronization between N threads can be implemented using a shared variable initialized to N.
- When a processor reaches the barrier, it decrements the shared variable by 1 and waits (in a busy wait loop) until the value of the variable is equal to zero before it leaves the barrier.
- Need locks???
- What if there is no shared variables (distributed memory machines)?

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 or not?
  - Writing programs that are highly parallel is not easy
  - **Writing programs that are correct and parallel is even harder**
  - Selecting the "right" kind of lock for performance
    - Spin lock, queue lock, ticket lock, read/writer lock, etc.
  - **Locking granularity issues**
    - False sharing?

Coarse-Grain Locks: Correct but Slow

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

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


Fine-Grain Locks: Parallel But Difficult

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

```
struct acct_t { int bal, Lock_t lock; …  };
shared struct acct_t  accts[MAX_ACCT];

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

- What about critical sections that require two locks?

Multiple Locks

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

```
struct acct_t { int bal, Lock_t lock; …};
shared struct acct_t  accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
  acquire(accts[id_from].lock);
  acquire(accts[id_to].lock);
  if (accts[id_from].bal >= amt) {
     accts[id_from].bal -= amt;
     accts[id_to].bal += amt;
  }
  release(accts[id_to].lock);
  release(accts[id_from].lock);
}
```


Multiple Locks And Deadlock

Thread 0	Thread 1
`id_from = 241;`	`id_from = 37;`
`id_to = 37;`	`id_to = 241;`
`acquire(accts[241].lock);`	`acquire(accts[37].lock);`
`// wait to acquire lock 37`	`// wait to acquire lock 241`
`// waiting…`	`// waiting…`
`// still waiting…`	`// …`

Multiple Locks And Deadlock

```
Thread 0                          Thread 1

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

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

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

Coffman Conditions for Deadlock

- 4 necessary conditions
  - mutual exclusion
  - hold+wait
  - no preemption
  - circular waiting
- break any **one** of these conditions to get deadlock freedom

Correct Multiple Lock Program

- **Always acquire multiple locks in same order**
  - Yet another thing to keep in mind when programming

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

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


Correct Multiple Lock Execution

Thread 0	Thread 1
`id_from = 241;`	`id_from = 37;`
`id_to = 37;`	`id_to = 241;`
`id_first = min(241,37)=37;`	`id_first = min(37,241)=37;`
`id_second = max(37,241)=241;`	`id_second = max(37,241)=241;`
`acquire(accts[37].lock);`	`// wait to acquire lock 37`
`acquire(accts[241].lock);`	`// waiting…`
`// do stuff`	`// …`
`release(accts[241].lock);`	`// …`
`release(accts[37].lock);`	`// …`
	`acquire(accts[37].lock);`

More Lock Madness

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

- **So lock-based programming is difficult…**

And To Make It Worse…

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

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

- Fixing these problem is an area of active research
  - One proposed solution "Transactional Memory"
  - Programmer uses construct: "atomic { … code … }"
    - Hardware, compiler & runtime executes the code "atomically"
    - Uses **speculation**, rolls back on conflicting accesses
- Synchronization in GPUs, stay tuned.