Synchronization

- *synchronization*: important issue for shared memory
  - regulates access to shared data
  - e.g., semaphore, monitor, critical section (s/w constructs)
  - synchronization primitive: *lock*

```
acquire(lock);      // while (lock != 0); lock = 1;
critical section;
release(lock);      // lock = 0;


0: ldw r1, lock     // wait for lock to be free
1: bnez r1, #0
2: stw #1, lock     // acquire lock
...                 // critical section
9: stw #0, lock     // release lock
```

Implementing Locks

- lock implementation from previous slide
  - called “spin lock”
  - – doesn’t actually work in all situations (=incorrect!)

```
processor 0      processor 1
0: ldw r1,lock
1: bnez r1,#0                    // p0 sees lock free
                 0: ldw r1,lock
                 1: bnez r1,#0   // p1 sees lock free
2: stw #1,lock                   // p0 acquires lock
                 2: stw #1,lock  // p1 acquires lock
...                              // p0 AND p1 in
                 ...             // critical section
...                              // TOGETHER
9: stw #0,lock
```

Implementing Locks

problem: acquire sequence (load-test-store) is *not atomic*

- option I: implement sequence in kernel
  - kernel can control interleaving by suppressing interrupts

  + implementation works

  – hugely expensive for common case (lock is free)

```
ACQUIRE_LOCK:
10: enable interrupts
11: disable interrupts
12: ldw r1,lock
13: bnez r1,#10
14: stw #1,lock
15: enable interrupts
16: ret
```

```
0: syscall ACQUIRE_LOCK
1: ...
2: stw #0, lock
```

Implementing Locks

- option II: ISA provides an *atomic lock-acquire* operation
  - load+check+store in one instruction (uninterruptible by definition)
  - e.g., test&set instruction (t&s) (aka fetch&add, swap)

```
t&s r1,lock     // ldw r1,lock; stw #1,lock;

0: t&s r1,lock
1: bnez r1, #0
2: ...
3: stw r1, #0
```

  - BTW, lock-release is already atomic

a lot of work has gone into making synchronization fast + cheap

Correctness: Memory Ordering

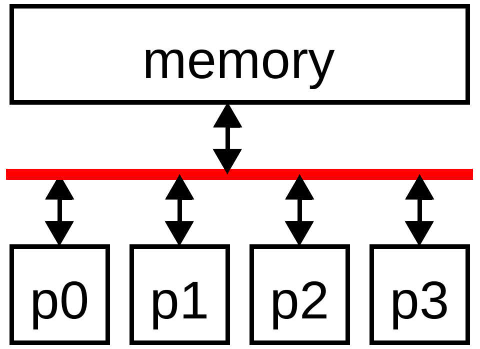
memory updates may become re-ordered by the memory system

- example

```
processor 0           processor 1
A = 0                 B = 0
A = 1                 B = 1
L1: if (B == 0)       L2: if (A == 0)
critical section      critical section
```

  - intuitively impossible for both processors to be in critical section
  - BUT can happen if memory operations are reordered
  - coherence: A’s must be same, B’s must be same EVENTUALLY
  - says nothing about relative timing of A’s and B’s coherence
  - this is specified by the *memory consistency model*

Memory Ordering: Sequential Consistency



"system is *sequentially consistent* if the result of ANY execution is the same as if the operations of all processors were executed in SOME sequential order and the operations of each individual processor appear in this sequence in program order" [Lamport]

- *sequential consistency (SC)*
  - all loads and stores in order

  + simple for programmer

  – not much room for hardware (or software) optimization
  - example works if system obeys *sequential consistency (SC)*

Weak(er) Consistency Models

- observation: SC needed *only* for lock variables
  - other variables?
  - either in critical section (no parallel access)
  - or not shared
- weaker consistency: can delay/reorder loads and stores
  - + more room for hardware optimization
  - – somewhat trickier programming model?
  - e.g., Intel IA-32: processor consistency (PC)
  - e.g., SPARC: total store order (TSO) is very similar to PC
  - e.g., Alpha: weak ordering (WO)
  - e.g., Intel IA-64: release consistency (RC)

Summary

- multiprocessors and multithreaded processors
  - workloads: parallel programs and parallel tasks
  - UMA vs. NUMA
  - trends: multithreaded processors, CMP
- interconnect
  - direct vs. indirect, store-and-forward vs. wormhole, topologies
- interprocess communication
  - message passing vs. shared memory
- shared memory
  - cache coherence: bus-based (2-state vs. 3-state), directory-based
  - synchronization
  - memory consistency