Multiprocessor Synchronization

- Material in this lecture in Henessey and Patterson, Chapter 8
  - pgs. 694-708
- Some material from David Patterson's slides for CS 252 at Berkeley

Multiprogramming and Multiprocessing Imply Synchronization

- Locking
  - Critical sections
  - Mutual exclusion
  - Used for exclusive access to shared resource or shared data for some period of time
  - Efficient update of a shared (work) queue
- Barriers
  - Process synchronization -- All processes must reach the barrier before any one can proceed (e.g., end of a parallel loop).
- Why doesn't coherency solve these problems?

Locking

- Typical use of a lock:

```
while (!acquire (lock))    /*spin*/
    ;
/* some computation on shared data (critical section) */
release (lock)
```

- Acquire based on primitive: Read-Modify-Write
  - Basic principle: “Atomic exchange”
  - Test-and-set
  - Fetch-and-increment

Synchronization

Issues for Synchronization:

- Uninterruptable instruction to fetch and update memory (atomic operation);
- User level synchronization operation using this primitive;
- For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

(slide from Patterson CS 252)

Uninterruptable Instruction to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory

  0 => synchronization variable is free

  1 => synchronization variable is locked and unavailable

  – Set register to 1 & swap
  – New value in register determines success in getting lock
    0 if you succeeded in setting the lock (you were first)
    1 if other processor had already claimed access
  – Key is that exchange operation is indivisible

  – Release the lock simply by writing a 0
  – Note that every execution requires a read and a write

Uninterruptable Instruction to Fetch and Update Memory

- Test-and-set: tests a value and sets it if the value passes the test
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  – 0 => synchronization variable is free

Load linked & store conditional

- Hard to have read & write in 1 instruction (needed for atomic exchange and others)
  - Potential pipeline difficulties from needing 2 memory operations
  - Makes coherence more difficult, since hardware cannot allow any operations between the read and write, and yet must not deadlock
- So, use 2 instructions instead…
- Load linked (or load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise
- How could you implement this instruction?

(slide from Patterson CS 252)

Load linked & store conditional - Example

- Example doing atomic exchange with LL & SC:

```
try:  mov   R3,R4       ; move exchange value
      ll    R2,0(R1)    ; load linked
      sc    R3,0(R1)    ; store conditional
      beqz  R3,try      ; branch if store fails (R3 = 0)
      mov   R4,R2       ; put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try:  ll    R2,0(R1)    ; load linked
      addi  R2,R2,#1    ; increment (OK if reg–reg)
      sc    R2,0(R1)    ; store conditional
      beqz  R2,try      ; branch if store fails (R2 = 0)
```

Note that these code sequences only do a single atomic swap or a fetch & increment – they do not implement a full lock acquire function (more on this later).

(slide from Patterson CS 252)

Spin Locks

- Processor continuously tries to acquire, spinning around a loop trying to get the lock

```
            li     R2,#1
lockit:     exch   R2,0(R1)      ;atomic exchange
            bnez   R2,lockit     ;already locked?
```

- Spin lock in a cache coherent environment (invalidation-based):
  - Bus utilized during the whole read-modify-write cycle
  - Since atomic-exchange writes a location in memory, need to send an invalidate (even if the lock is not acquired)
  - In general loop to test the lock is short, so lots of bus contention

Improved Spin Locks ("test and test&set")

- Be smarter about spinning:
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables, if we can avoid invalidates
- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:        li     R2,#1
lockit:     lw     R3,0(R1)      ;load var
            bnez   R3,lockit     ;not free=>spin
            exch   R2,0(R1)      ;atomic exchange
            bnez   R2,try        ;already locked?
```

- Any remaining performance problems?

(slide from Patterson CS 252)

Problems with Improved Spin Lock

- Have a race condition for acquiring a lock that has just been released.
  - All waiting processors will suffer read and write miss
  - $O(n^2)$ bus transactions for n contending processes.
- Potential improvements
  - Queuing Locks (software or hardware)
  - Exponential backoff – after each failed attempt to grab the lock, wait an exponentially increasing amount of time before trying again (similar to Ethernet collision handling)

Queuing Locks

- Basic idea: a queue of waiting processors is maintained in shared-memory for each lock (best for bus-based machines)
  - Each processor performs an atomic operation to obtain a memory location (element of an array) on which to spin
  - Upon a release, the lock can be directly handed off to the next waiting processor

Barriers

- All processes have to wait at a synchronization point
  - End of parallel do loops
- Processes don't progress until they all reach the barrier
- Low-performance implementation: use a counter initialized with the number of processes
  - When a process reaches the barrier, it decrements the counter (atomically -- fetch-and-add (-1)) and busy waits
  - When the counter is zero, all processes are allowed to progress (broadcast)
- Lots of possible optimizations (tree, butterfly etc.)
  - Is it important? Barriers do not occur that often (Amdahl's law….)