Outline

• Difference Between Coherence and Consistency

• Sequential Consistency

• Relaxed Memory Consistency Models

• Consistency Optimizations

• Synchronization Optimizations
  – Review of Synchronization
  – Synchronization Design Tradeoffs
  – Cool New Synchronization Schemes (e.g., SLE)
Synchronization

• Mutual exclusion (critical sections)
  – Lock & unlock

• Event notification
  – Point-to-point (producer-consumer, flags)
  – Global (barrier)

• Locks, Barriers, Flags, etc.
  – How are these implemented?
Anatomy of A Synchronization Operation

1) Acquire method
   • Method for trying to obtain lock or proceed past barrier

2) Waiting algorithm
   • Spin or busy wait
   • Block (suspend)

3) Release method
   • Method to allow other processes to get past synchronization event
HW/SW Implementation Tradeoffs

- User wants high level (ease of programming)
  - LOCK(lock_variable), UNLOCK(lock_variable)
  - BARRIER(barrier_variable, Num_Procs)

- Hardware’s advantage
  - The Need for Speed (it’s fast)

- Software’s advantage
  - Flexibility

- Goals
  - Low latency
  - Low traffic
  - Scalability
  - Low storage overhead
  - Fairness
How NOT To Implement Locks

• **LOCK**
  
  ```
  while(lock_variable == 1);  // spin
  lock_variable = 1;
  ```

• **UNLOCK**
  
  ```
  lock_variable = 0;
  ```

• Implementation requires Mutual Exclusion!
  – Can have two processes successfully acquire the lock

• Mutual exclusion requires atomic operations ...
Atomic Read-Modify-Write Operations

- **Test&Set**\((r,x)\)
  \[ r = m[x] \]
  \[ m[x] = 1 \]
  \[
  \text{• } r \text{ is a register}
  \text{• } m[x] \text{ is memory location } x
  \]

- **Swap**\((r,x)\)
  \[ r = m[x], m[x] = r \]

- **Compare&Swap**\((r1,r2,x)\)
  \[
  \text{if } (r1 == m[x]) \text{ then}
  \]
  \[ r2 = m[x], m[x] = r2 \]

- **Fetch&Op**\((r,x,op)\)
  \[ r = m[x], m[x] = op(m[x]) \]
Aside: Load-Locked Store-Conditional

- Pair of instructions that can implement lots of atomic operations
- Load-Locked loads address A into register r1
- We can then manipulate r1 for a while
- Store-Conditional writes r1 back to A if A hasn’t been modified
  - Invalidation
  - Replacement
  - Context switch

```
lock: .ll r1, memloc // load-lock memloc into r1
      sc memloc, #1 // conditionally store 1 into memloc
      beqz lock    // if sc fails, try lock again
      ret

unlock: st location, #0
        ret
```
Performance of Test & Set

LOCK
    while (test&set(x) == 1);
UNLOCK
    x = 0;

• Problem 1: Bad performance under contention
  – Test&Set causes GetExclusive coherence request

• Problem 2: Not fair
  – Processors don’t get lock in order in which they request it

• Both problems due to the waiting algorithm!
Better Lock Implementations

• Two choices:
  – Don’t execute test&set so much → test&set with back-off
  – Spin without generating bus traffic → test-and-test&set

• Test&Set with back-off
  – Insert delay between test&set operations (not too long)
  – Exponential seems good (k*c^i)
  – Not fair

• Test-and-Test&Set
  – Spin (test) on local cached copy until it gets invalidated, then issue test&set
  – Intuition: No point in trying to set the location until we know that it’s not set, which we can detect when it gets invalidated
  – Still contention after invalidation
  – Still not fair
Lock Implementation Details

• To Cache or Not to Cache, that is the question

Uncached
- Latency for one operation increases
+ Fast hand-off between processes

Cached
- Might generate a lot of traffic if lock moves around
+ Might reuse lock a lot (locality), then traffic would be reduced by caching

• Must keep ownership for entire read-modify-write cycle
  – Synchronization operation is visible to the memory system
Fetch&Increment Based Locks

• **Ticket Lock** (like at the bakery)

  **LOCK**
  – Obtain number via fetch&inc
  – Spin on *now-serving* counter

  **UNLOCK**
  – Increment *now-serving* counter

• **Array based Lock**
  – Obtain location to spin on rather than value
  – Fair
  – Slight increase in storage
  – Put locations in separate cache blocks, else same traffic as t&t&s
H/W Queue-Based Locks: QOLB

- Distributed directory (SCI)
- Lock bit in every cache line
- Maintains sharing list in a doubly-linked list
- Local spinning in the cache: cache hit while waiting
S/W Queue-Based Locks: MCS

- To emulate QOLB in software:
  - (master) lock resides in shared memory
  - Requesters allocate a block in shared memory to spin on
    - The flag the requester spins on is originally set to 1
  - Upon acquire, requester inserts its block in linked list
    - It reads the address of the last requester’s local block
    - It writes its local block address to the master
  - Upon release
    - Releaser removes itself from the linked list
    - Sets the next processor in line’s flag to 0
Mechanisms To Reduce Lock Overhead

- Optimizations used by locks:
  - Local spinning (on data in local cache)
  - Queue-based locking
  - Collocation of lock and the data it protects
  - Synchronous prefetch

<table>
<thead>
<tr>
<th></th>
<th>L-spin</th>
<th>Queue</th>
<th>Collocation</th>
<th>S-fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp;S</td>
<td>no</td>
<td>no</td>
<td>optional</td>
<td>no</td>
</tr>
<tr>
<td>T&amp;T&amp;S</td>
<td>yes</td>
<td>no</td>
<td>optional</td>
<td>no</td>
</tr>
<tr>
<td>MCS</td>
<td>yes</td>
<td>yes</td>
<td>partial</td>
<td>no</td>
</tr>
<tr>
<td>QOLB</td>
<td>yes</td>
<td>yes</td>
<td>optional</td>
<td>yes</td>
</tr>
</tbody>
</table>
Microbenchmark Analysis

Lock performance with increase in processors

Elapsed Time (M cycles)

Number of Processors

1 2 4 8 16 32 64

T&S
T&T&S
MCS
QOLB

(C) 2010 Daniel J. Sorin from Adve, Falsafi, Hill, Lebeck, Reinhardt, Singh

ECE 259 / CPS 221
What are the Performance Issues?

- **Low latency:**
  - Should be able to get a free lock quickly

- **Scalability:**
  - Should perform well beyond a small number of processors (< 64)

- Low storage overhead

- Fairness

- Blocking/Non-blocking
Performance of Locks

• Performance **depends** on many factors
  – Amount of contention
  – Number of processors
  – Snooping vs. directory
  – Hardware support for locks

• Test&set is good with no contention
• Array based (Queue) is best with high contention

• *Reactive Synchronization* by Lim & Agarwal
  – Choose lock implementation based on contention
Point-to-Point Event Synchronization

• Often use normal variables as flags
  
  ```
  a = f(x); while (flag == 0);
  flag = 1; b = g(a);
  ```

• If we know a=0 beforehand
  
  ```
  a = f(x) while (a == 0);
  b = g(a);
  ```

• Assumes Sequential Consistency!!

• Full/Empty bits are similar to flags
  – Set on Write
  – Cleared on Read
  – Can’t write if set, can’t read if clear
Implementing a Centralized Barrier

BARRIER(bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)
        bar_name.flag = 0;
    bar_name.counter++;
    UNLOCK(bar_name.lock);
    if (bar_name.counter == p) {
        bar_name.counter = 0;
        bar_name.flag = 1;
    } else {
        while(bar_name.flag == 0) {}; /* busy wait */
    }
}

But what if P1 doesn’t see flag set to 1 in first barrier before P2 re-enters barrier and resets flag to 0?
BARRIER(bar_name, p) {
    local_sense = !(local_sense); /* toggle private state */
    LOCK(bar_name.lock);
    bar_name.counter++;
    UNLOCK(bar_name.lock);
    if (bar_name.counter == p) {
        bar_name.counter = 0;
        bar_name.flag = local_sense;
    }
    else
        while(bar_name.flag != local_sense) {}; /* busy wait*/
}
SLE: Speculative Lock Elision

• PRESENTATION
Outline

• Difference Between Coherence and Consistency
• Sequential Consistency
• Relaxed Memory Consistency Models
• Consistency Optimizations
• Synchronization Optimizations