Coherence vs. Consistency

- Intuition says loads should return latest value
  - what is latest?
- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all locations
- A Memory System is Coherent if
  - can serialize all operations to that location such that,
  - operations performed by any processor appear in program order
    - program order = order defined by program text or assembly code
  - value returned a read is value written by last store to that location
Why Coherence != Consistency

/* initial A = B = flag = 0 */

P1
A = 1;
B = 1;
flag = 1;

P2
while (flag == 0); /* spin */
print A;
print B;

Intuition says printed A = B = 1
Coherence doesn’t say anything, why?
Consider coalescing write buffer

Sequential Consistency

• Lamport 1979

“A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”
The Memory Model

Sequential processors issue memory ops in program order

\[ \text{P1} \quad \text{P2} \quad \cdots \quad \text{Pn} \]

Switch randomly set after each memory op

Memory

Definitions and Sufficient Conditions

- **Sequentially Consistent Execution**
  - result is same as one of the possible interleavings on uniprocessor
- **Sequentially Consistent System**
  - any possible execution corresponds to some possible total order
Definitions

- **Memory operation**
  - execution of load, store, atomic read-modify-write access to mem location

- **Issue**
  - operation is issued when it leaves processor and is presented to memory system (cache, write-buffer, local and remote memories)

- **Perform**
  - store is performed wrt to a processor p when a load by p returns value produced by that store or a later store
  - A load is performed wrt to a processor when subsequent stores cannot affect value returned by that load

- **Complete**
  - memory operation is performed wrt all processors.

- **Program Execution**
  - Memory operations for specific run only (ignore non memory-referencing instructions)

Sufficient Conditions for Sequential Consistency

- Every processor issues memory ops in program order
- Processor must wait for store to complete before issuing next memory operation
- After load, issuing proc waits for load to complete, and store that produced value to complete before issuing next op
- Easily implemented with shared bus.
Synchronization

- Mutual Exclusion (critical sections)
  - Lock & Unlock
- Event Notification
  - point-to-point (producer-consumer, flags)
  - global (barrier)
- LOCK, BARRIER
  - How are these implemented?

Anatomy of A Synchronization Operation

- Acquire Method
  - method for trying to obtain the lock, or proceed past barrier
- Waiting Algorithm
  - Spin or busy wait
  - Block (suspend)
- Release Method
  - method to allow other processes to proceed 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**
  - The Need for Speed (it’s fast)
- **Software**
  - Flexible
- **Want**
  - low latency
  - low traffic
  - Scalability
  - low storage overhead
  - fairness

---

**How Not To Implement Locks**

- **LOCK**
  ```
  while(lock_variable == 1);
  lock_variable = 1;
  ```
- **UNLOCK**
  ```
  lock_variable = 0;
  ```
- **Implementation requires Mutual Exclusion!**
  - Can have two processes successfully acquire the lock
Atomic Read-Modify-Write Operations

- **Test&Set**(r,x)
  
  \[ r = m[x] \]
  
  \[ m[x] = 1 \]

- **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)
  
  \[ r = m[x], m[x] = \text{op}(m[x]) \]

Load-Locked Store-Conditional

- **Pair of Instructions**
- **Load-Locked** sets flag and address
- **Store-Conditional** fails if flag clear
- **Flag is cleared on**
  - invalidation
  - replacement
  - context switch

  \[
  \text{lock: ll r1, location} \\
  \text{sc location, ll} \\
  \text{beginc, lock} \\
  \text{ret} \\
  \text{unlock: st location, ret}
  \]
Alpha test & set Bit in 32-bit word

\[
\begin{array}{ll}
t\_and\_set: & \# \text{ arg reg a0 holds bit# to test} \\
1: & \text{ldl}_1 t0, (a1) \# \text{get word to test} \\
    & \text{bis} \text{ zero, 1, t1} \# \text{logical OR set t1=1} \\
    & \text{sli} \text{ t1, a0, t2} \# \text{create bitmask} \\
    & \text{and} \text{ t0, t2, t3} \# \text{isolate bit} \\
    & \text{bne} \text{ t3, 2f} \# \text{branch if already set} \\
    & \text{bis} \text{ t0, t2, v0} \# \text{or in bit to be set} \\
    & \text{stl} \_ q \text{ v0, (a1)} \# \text{conditional store} \\
    & \text{beq} \text{ v0, 3f} \# \text{branch if store failed} \\
    & \text{mb} \# \text{memory barrier} \\
    & \text{ret} \text{ zero, (ra)} \# \text{ret 1 (v0) if set success} \\
2: & \text{bis} \text{ zero, zero, v0} \# \text{ret 0 if bit already set} \\
    & \text{ret} \text{ zero, (ra)} \\
3: & \text{br} \text{ zero, 1b} \# \text{retry interlocked update}
\end{array}
\]

Performance of Test & Set

\text{LOCK}
\begin{verbatim}
    while (test&set(x) == 1);
\end{verbatim}

\text{UNLOCK}
\begin{verbatim}
    x = 0;
\end{verbatim}

- High contention (many processes want lock)
- Remember the CACHE!
- Each test&set is a read miss and a write miss
  - Not fair
- Problem is?
- Waiting Algorithm!
Better Lock Implementations

• Two choices:
  – Don’t execute test&set so much
  – Spin without generating bus traffic

• Test&Set with Backoff
  – Insert delay between test&set operations (not too long)
  – Exponential seems good (k*c')
  – 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 get invalidated...
  – Still contention after invalidate
  – Still not fair

Fetch&Inc Based Locks

• Ticket Lock
  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
Software Queue

- Linked list points to next in line

(a) ![Diagram](a)

(b) ![Diagram](b)

(c) ![Diagram](c)

(d) ![Diagram](d)

(e) ![Diagram](e)

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 procs (< 64)
- Low storage overhead
- Fairness
- Blocking/Non-blocking
- Are spin locks fair?
Performance of Locks

- Contested vs. Uncontested
- 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

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
  - we’ll exploit this fact later in the semester
Point-to-Point Event Synchronization

- Often use normal variables as flags
  \[ \text{a} = \text{f}(x); \quad \text{while} \ (\text{flag} == 0); \]
  \[ \text{flag} = 1; \quad \text{b} = \text{g}(a); \]

- If we know a before hand
  \[ \text{a} = \text{f}(x) \quad \text{while} \ (\text{a} == 0); \]
  \[ \text{b} = \text{g}(a); \]

- Assumes Sequential Consistency!!

- Full/Empty Bits
  - 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 */
}
```

- Does this work?
Barrier With Sense Reversal

```c
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 */
}
```

Synchronization Algorithms

- Tournament Locks, SW Combining Tree

![Diagram of tournament locks and combining tree](image)

(C) 2001 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhart, & Singh
Outline

• Motivation
• Coherence
• Coherence Tradeoffs
• Memory Consistency
• Synchronization