Implementing Synchronization Primitives

Arvind Krishnamurthy
Fall 2004

Role of Synchronization

- “A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”

- Types of Synchronization
  - Mutual Exclusion
  - Event synchronization
    - point-to-point
    - group
    - global (barriers)
**Hardware-Software Interplay**

- Synchronization: critical component of concurrency where one sees the interplay between hardware and software

- Dijkstra (1965) and Knuth (1966) demonstrated that it is possible to provide mutual exclusion with just reads/writes
  - Assumes sequentially consistent execution

- Most real systems rely on hardware support for synchronization
  - Key requirement: an instruction that performs more than one operation (read/write)

- How much hardware support?
  - high level operations?
  - atomic instructions?
  - specialized interconnect?

---

**History of Instruction Sets**

- IBM 370 provided a compare-and-swap instruction
  - Takes three operands: memory location, two registers
  - Compares memory value with first register; if equal, stores second register in the specified memory location
  - All done atomically!

- x86 architectures allowed a prefix lock modifier for all instructions
  - Can make any instruction atomic

- High-level language advocates wanted hardware locks/barriers
  - but it goes against the “RISC” flow, and has other problems

- SPARC: atomic register-memory ops (swap, compare&swap)

- MIPS, IBM Power: no atomic operations but pair of instructions
  - load-locked, store-conditional
  - later used by PowerPC and DEC Alpha too
Components of a synchronization event

- Acquire method
  - Acquire right to the synchronization
    - enter critical section, go past event
- Waiting algorithm
  - Wait for synchronization to become available when it isn't
  - busy-waiting, blocking, or hybrid
- Release method
  - Enable other processors to acquire right to the synchronization

- Waiting algorithm is independent of type of synchronization
  - makes no sense to put in hardware

Hardware Setting

![Diagram of hardware setting](image)
Atomic Instructions

- Specifies a location, register, & atomic operation
  - Value in location read into a register
  - Another value (function of value read or not) stored into location
- Many variants
  - Varying degrees of flexibility in second part
- Simple example: test&set
  - Value in location read into a specified register
  - Constant 1 stored into location
    - Location = 0 ➞ lock is free; Location = 1 ➞ locked
    - Successful if value loaded into register is 0

Simple Test&Set Lock

```assembly
lock:    t&s  register, location
  bnz  lock   /* if not 0, try again */
  ret    /* return control to caller */
unlock:  st   location, #0  /* write 0 to location */
         ret    /* return control to caller */
```

- Other read-modify-write primitives
  - Swap
  - Fetch&op (fetch-and-increment, fetch-and-decrement)
  - Compare&swap
    - Three operands: location, register to compare with, register to swap with
    - Not commonly supported by RISC instruction sets
**Performance Criteria**

- Latency (time per op)
  - especially when light contention
- Bandwidth (ops per sec)
  - especially under high contention
- Traffic
  - load on critical resources
  - especially on failures under contention
- Storage
- Fairness

**Test&Set on SGI Challenge**

- Question: Why does performance degrade?
**Enhancements to Simple Lock**

- Reduce frequency of issuing test&sets while waiting
  - *Test&set lock with backoff*
  - Don’t back off too much or will be backed off when lock becomes free
  - Exponential backoff works quite well empirically: $t^i$ time = $k^i \cdot c^i$
- Busy-wait with read operations rather than test&set
  - *Test-and-test&set lock*
  - Keep testing with ordinary load
    - cached lock variable will be invalidated when release occurs
  - When value changes (to 0), try to obtain lock with test&set

```
lock: if (!location)
if (test-and-set(location))
goto lock;
/* have lock now */
```

**Improved Hardware Primitives: LL-SC**

- Goals:
  - Test with reads
  - Failed read-modify-write attempts don’t generate invalidations
  - Nice if single primitive can implement range of r-m-w operations
- *Load-Locked* (or -linked), *Store-Conditional*
  - LL reads variable into register
  - Follow with arbitrary instructions to manipulate its value
  - SC tries to store back to location
  - succeed if and only if no other write to the variable since this processor’s LL
    - indicated by condition codes;
  - If SC succeeds, all steps happened atomically
  - If it fails, doesn’t write or generate invalidations
    - must retry acquire
**Simple Lock with LL-SC**

```assembly
lock:   li    reg1, location    /* LL location to reg1 */
bnez   reg1, lock
sc     location, reg2       /* SC reg2 into location*/
beqz   lock          /* if failed, start again */
ret
unlock: st    location, #0    /* write 0 to location */
         ret
```

- Can do more fancy atomic ops by changing what's between LL & SC
  - But keep it small so SC likely to succeed
  - Don't include instructions that would need to be undone (e.g. stores)
- SC can fail (without putting transaction on bus) if:
  - Detects intervening write even before trying to get bus
  - Tries to get bus but another processor's SC gets bus first
- LL, SC are not lock, unlock respectively
  - Only guarantee no conflicting write to lock variable between them
  - But can use directly to implement simple operations on shared variables

---

**Trade-offs So Far**

- Latency?
- Bandwidth? Traffic?
- Storage?
- Fairness?

- What happens when several processors spinning on lock and it is released?
  - traffic per P lock operations?
Ticket Lock

- Only one r-m-w per acquire
- Two counters per lock (next_ticket, now_serving)
  - Acquire: `fetch&inc next_ticket, my_ticket`  
    
    `wait for now_serving == my_ticket`
  - atomic op when arrive at lock, not when it’s free (so less contention)
  - Release: increment now_serving
- Performance
  - Question: evaluate the approach with respect to the different criteria mentioned before.

- Question: how can we improve on this approach in general?

---

Array-based Queuing Locks

- Waiting processes poll on different locations in an array of size $p$
  - Acquire
    - fetch&inc to obtain address on which to spin (next array element)
    - ensure that these addresses are in different cache lines or memories
  - Release
    - set next location in array, thus waking up process spinning on it
- $O(1)$ traffic per acquire with coherent caches
- FIFO ordering, as in ticket lock, but, $O(p)$ space per lock
- Not so great for non-cache-coherent machines with distributed memory
Array Locks

- Distribute the shared value, and do directed “unlocks”

```
Lock:
    my_slot = fetch_and_increment(next_slot);
    if (my_slot % numProcs == 0)
        fetch_and_add(next_slot, -numProcs);
    my_slot = my_slot % numProcs;
    while (slots[my_slot] == must_wait);
    slots[my_slot] = must_wait;

Unlock:
    slots[(my_slot + 1) % numProcs] = has_lock;
```

MCS Algorithm

- Uses only as much storage as required (“entry” record)
- Maintains a list of requesting processors

```
Lock:
    entry = new Entry;
    entry->next = nil;
    pred = fetch_and_store(Ltail, entry);
    if (pred != nil) {
        entry->blocked = true;
        pred->next = entry;
        while (entry->blocked);
    }

Unlock:
    if (entry->next == nil) {
        if (cmp_and_swap(Ltail, entry, nil))
            return;
        while (entry->next == nil);
    }
    entry->next->blocked = false;
```
Lock Performance on SGI Challenge

Loop:  
  lock;
  delay(c);
  unlock;
  delay(d);

(a) Null (c = 0, d = 0)  
(b) Critical-section (c = 3.64 µs, d = 0)  
(c) Delay (c = 3.64 µs, d = 1.29 µs)