Synchronization

CS 740
November 14, 2003

Based on TCM/C&S

Topics
• Locks
• Barriers
• Hardware primitives

Types of Synchronization

Mutual Exclusion
• Locks

Event Synchronization
• Global or group-based (barriers)
• Point-to-point

Busy Waiting vs. Blocking

Busy-waiting is preferable when:
• scheduling overhead is larger than expected wait time
• processor resources are not needed for other tasks
• schedule-based blocking is inappropriate (e.g., in OS kernel)

A Simple Lock

lock: ld register, location
cmp register, #0
bnz lock
st location, #1
ret

unlock: st location, #0
ret
Need Atomic Primitive!

Test&Set
Swap
Fetch&Op
  • Fetch&Incr, Fetch&Decr
Compare&Swap

Test&Set based lock

lock: t&s register, location
bnz lock
ret
unlock: st location, #0
ret

T&S Lock Performance

Code: lock; delay(c); unlock;
Same total no. of lock calls as \( p \) increases; measure time per transfer

Test and Test and Set

A: while (lock != free)
    if (test&set(lock) == free) {
        critical section;
    }
else goto A;

(+): spinning happens in cache
(-): can still generate a lot of traffic when many processors go to do test&set
**Test and Set with Backoff**

Upon failure, delay for a while before retrying
- either constant delay or exponential backoff

Tradeoffs:
(+ ) much less network traffic
(- ) exponential backoff can cause starvation for high-contention locks
  - new requestors back off for shorter times

But exponential found to work best in practice

**Test and Set with Update**

Test and Set sends updates to processors that cache the lock

Tradeoffs:
(+ ) good for bus-based machines
(- ) still lots of traffic on distributed networks

Main problem with test&set-based schemes is that a lock release causes all waiters to try to get the lock, using a test&set to try to get it.

**Ticket Lock (fetch&incr based)**

Two counters:
- next_ticket (number of requestors)
- now_serving (number of releases that have happened)

Algorithm:
- First do a fetch&incr on next_ticket (not test&set)
- When release happens, poll the value of now_serving
  - if my_ticket, then I win

Use delay; but how much?

**Ticket Lock Tradeoffs**

(+ ) guaranteed FIFO order; no starvation possible
(+ ) latency can be low if fetch&incr is cacheable
(+ ) traffic can be quite low
(- ) but traffic is not guaranteed to be O(1) per lock acquire
Lock Performance on SGI Challenge

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

- Simple LL-SC lock does best at small p due to unfairness
  - Not so with delay between unlock and next lock
  - Need to be careful with backoff
- Ticket lock with proportional backoff scales well, as does array lock
  - Methodologically challenging, and need to look at real workloads

Array-Based Queueing Locks

Every process spins on a unique location, rather than on a single now_serving counter
- fetch&incr gives a process the address on which to spin

Tradeoffs:
- (+) guarantees FIFO order (like ticket lock)
- (+) O(1) traffic with coherence caches (unlike ticket lock)
- (-) requires space per lock proportional to P

List-Base Queueing Locks (MCS)

All other good things + O(1) traffic even without coherent caches (spin locally)
- Uses compare&swap to build linked lists in software
- Locally-allocated flag per list node to spin on
- Can work with fetch&store, but loses FIFO guarantee

Tradeoffs:
- (+) less storage than array-based locks
- (+) O(1) traffic even without coherent caches
- (-) compare&swap not easy to implement

Implementing Fetch&Op

Load Linked/Store Conditional

lock: ll reg1, location /* LL location to reg1 */
bnz reg1, lock /* check if location locked*/
sc location, reg2 /* SC reg2 into location*/
beqz reg2, lock /* if failed, start again */
ret
unlock:
  st location, #0 /* write 0 to location */
  ret
**Barriers**

We will discuss five barriers:

- centralized
- software combining tree
- dissemination barrier
- tournament barrier
- MCS tree-based barrier

**A Working Centralized Barrier**

Consecutively entering the same barrier doesn’t work

- Must prevent process from entering until all have left previous instance
- Could use another counter, but increases latency and contention

Sense reversal: wait for flag to take different value consecutive times

```plaintext
BARRIER (bar_name, p) {
    local_sense = !local_sense; /* toggle private sense var */
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;
    /* mycount is private */
    if (bar_name.counter == p)
        UNLOCK(bar_name.lock);
        bar_name.counter = 0; /* reset for next */
        bar_name.flag = local_sense; /* release waiters */
    else
        { UNLOCK(bar_name.lock);
            while (bar_name.flag != local_sense) {};
        }
}
```

**Centralized Barrier**

Basic idea:

- notify a single shared counter when you arrive
- poll that shared location until all have arrived

Simple implementation require polling/spinning twice:

- first to ensure that all procs have left previous barrier
- second to ensure that all procs have arrived at current barrier

Solution to get one spin: *sense reversal*

**Software Combining Tree Barrier**

Writes into one tree for barrier arrival

Reads from another tree to allow procs to continue

Sense reversal to distinguish consecutive barriers
**Tournament Barrier**

Binary combining tree

Representative processor at a node is statically chosen

- no fetch\&op needed

In round $k$, proc $i = 2^k$ sets a flag for proc $j = i - 2^k$

- $i$ then drops out of tournament and $j$ proceeds in next round
- $i$ waits for global flag signalling completion of barrier to be set

- could use combining wakeup tree

---

**Barrier Performance**

- Centralized does quite well
- Helpful hardware support: piggybacking of reads misses on bus
- Also for spinning on highly contended locks

---

**MCS Software Barrier**

Modifies tournament barrier to allow static allocation in wakeup tree, and to use sense reversal

Every processor is a node in two P-node trees:

- has pointers to its parent building a fanin-4 arrival tree
- has pointers to its children to build a fanout-2 wakeup tree

---

**Barrier Recommendations**

Criteria:

- length of critical path
- number of network transactions
- space requirements
- atomic operation requirements
**Space Requirements**

- **Centralized:**
  - constant

- **MCS, combining tree:**
  - $O(P)$

- **Dissemination, Tournament:**
  - $O(P\log P)$

**Network Transactions**

- **Centralized, combining tree:**
  - $O(P)$ if broadcast and coherent caches;
  - unbounded otherwise

- **Dissemination:**
  - $O(P\log P)$

- **Tournament, MCS:**
  - $O(P)$

**Critical Path Length**

- **If independent parallel network paths available:**
  - all are $O(\log P)$ except centralized, which is $O(P)$

- **Otherwise (e.g., shared bus):**
  - linear factors dominate

**Primitives Needed**

- **Centralized and combining tree:**
  - atomic increment
  - atomic decrement

- **Others:**
  - atomic read
  - atomic write
 Barrier Recommendations

Without broadcast on distributed memory:
- **Dissemination**
  - MCS is good, only critical path length is about 1.5X longer
  - MCS has somewhat better network load and space requirements

Cache coherence with broadcast (e.g., a bus):
- **MCS with flag wakeup**
  - centralized is best for modest numbers of processors

Big advantage of *centralized* barrier:
- adapts to changing number of processors across barrier calls