Synchronization

Todd C. Mowry
CS 740
November 24, 1998

Topics
• Locks
• Barriers
Types of Synchronization

Mutual Exclusion

• Locks

Event Synchronization

• Global or group-based (barriers)
• Point-to-point
  – tightly coupled with data (full-empty bits, futures)
  – separate from data (flags)
Blocking vs. Busy-Waiting

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)

But typical busy-waiting produces lots of network traffic
  • hot-spots
Reducing Busy-Waiting Traffic

Trend was toward increasing hardware support

- sophisticated atomic operations
- combining networks
- multistage networks with special sync variables in switches
- special-purpose cache hardware to maintain queues of waiters

But (Mellor-Crummey and Scott),

*Appropriate software synchronization algorithms can eliminate most busy-waiting traffic.*
Software Synchronization

Hardware support required
- simple fetch-and-op support
- ability to “cache” shared variables

Implications
- efficient software synch can be designed for large machines
- special-purpose hardware has only small additional benefits
  - small constant factor for locks
  - best case factor of log P for barriers
Mutual Exclusion

Discuss four sets of primitives:
- test-and-set lock
- ticket lock
- array-based queueing lock
- list-based queueing lock
Test and Set Lock

Cacheable vs. non-cacheable

Cacheable:
- good if locality on lock access
  - reduces both latency and lock duration
  - particularly if lock acquired in exclusive state
- not good for high-contention locks

Non-cacheable:
- read-modify-write at main memory
- easier to implement
- high latency
- better for high-contention locks
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)

Ticket lock has just one proc do a test&set when a lock is released.

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?
- exponential is bad
- tough to find a delay that makes network traffic constant for a single lock
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
Array-Based Queueing Locks

Every process spins on a unique location, rather than on a single no_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
Lock Performance

*(See attached sheets.)*
Implementing Fetch&Op Primitives

One possibility for implementation in caches: LL/SC

- “Load Linked” (LL) loads the lock and sets a bit
- When “atomic” operation is finished, “Store Conditional” (SC) succeeds only if bit was not reset in interim
- Fits within the “load/store” (aka RISC) architecture paradigm
  - I.e. no instruction performs more than one memory operation
  - Good for pipelining and clock rates

Good for bus-based machines: SC result known on bus

More complex for directory-based machines:

- wait for SC to go to directory and get ownership (long latency)
- have LL load in exclusive mode, so SC succeeds immediately if still in exclusive mode
Bottom Line for Locks

Lots of options

Using simple hardware primitives, we can build very successful algorithms in software
- LL/SC works well if there is locality of synch accesses
- Otherwise, in-memory fetch&ops are good for high contention
Lock Recommendations

Criteria:

- scalability
- one-processor latency
- space requirements
- fairness
- atomic operation requirements
Scalability

MCS locks most scalable in general
  • array-based queueing locks also good with coherent caches

Test&set and Ticket locks scale well with backoff
  • but add network traffic
Single-Processor Latency

Test&set and Ticket locks are best
MCS and Array-based queueing locks also good when good fetch&op primitives are available
  • array-based bad in terms of space if too many locks needed
    – in OS kernel, for example
    – when more processes than processors
Fairness

Ticket lock, array-based lock and MCS all guarantee FIFO
  • MCS needs compare&swap

Fairness can waste CPU resources in case of preemption
  • can be fixed by coscheduling
Primitives Required:

Test&Set:
  • test&set

Ticket:
  • fetch&incr

MCS:
  • fetch&store
  • compare&swap

Queue-based:
  • fetch&store
Lock Recommendations

For scalability and minimizing network traffic:
  • *MCS*
    – one-proc latency good if efficient fetch&op provided

If one-proc latency is critical, or fetch&op not available:
  • *Ticket with proportional backoff*

If preemption possible while spinning, or if neither fetch&store nor fetch&incr provided:
  • *Test&Set with exponential backoff*
Barriers

We will discuss five barriers:

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

Tradeoffs:
(+) simple
(-) high traffic, unless machines are cache-coherent with broadcast
Software Combining Tree Barrier

Represent shared barrier variable as a tree of variables
  • each node is in a separate memory module
  • leaf is a group of processors, one of which plays “representative”

Writes into one tree for barrier arrival

Reads from another tree to allow procs to continue

Sense reversal to distinguish consecutive barriers

Disadvantage:
  • except on cache-coherent machines, assignment of spin flags to processors is hard, so may spin remotely
Dissemination Barrier

$\log P$ rounds of synchronization

In round $k$, proc $i$ synchronizes with proc $(i+2^k) \mod P$

Advantage:
- Can statically allocate flags to avoid remote spinning
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

Disadvantage:
  • without coherent caches and broadcast, suffers from either:
    – traffic due to single flag, or
    – same problem as combining trees (for wakeup tree)
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

Advantages:
  • spins on local flag variables
  • requires O(P) space for P processors
  • achieves theoretical minimum number of network transactions: 
    − (2P - 2)
  • O(logP) network transactions on critical path
Barrier Performance

(See attached sheets.)
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