Algorithms for Scalable Lock Synchronization on Shared-memory Multiprocessors

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu
Summary from Last Week

Locks using only load and store

- \(O(n)\) words for one lock for mutual exclusion among \(n\) threads
- \(O(n)\) operations required to acquire lock in uncontended case
- Need more hardware support for better protocols
- Important issues for lock design
  - \(-\) space
  - \(-\) time
  - \(-\) properties
    - provide
      - mutual exclusion
      - fairness
    - avoid
      - deadlock
      - starvation
Atomic Primitives for Synchronization

Atomic read-modify-write primitives

- **test_and_set(Word &M)**
  - writes a 1 into M
  - returns M’s previous value

- **swap(Word &M, Word V)**
  - replaces the contents of M with V
  - returns M’s previous value

- **fetch_and_Φ(Word &M, Word V)**
  - Φ can be ADD, OR, XOR, ...
  - replaces the value of M with Φ(old value, V)
  - returns M’s previous value

- **compare_and_swap(Word &M, Word oldV, Word newV)**
  - if (M == oldV) M ← newV
  - returns TRUE if store was performed
  - universal primitive
Load-Linked & Store Conditional

- **load_linked(Word &M)**
  - sets a mark bit in M’s cache line
  - returns M’s value

- **store_conditional(Word &M, Word V)**
  - if mark bit is set for M’s cache line, store V into M, otherwise fail
  - condition code indicates success or failure
  - may spuriously fail if
    - context switch, another load-link, cache line eviction

- **Arbitrary read-modify-write operations with LL / SC**

  loop forever
  - load linked on M returns V
  - $V' = f(V, ...)$  
    // $V'$ = arbitrary function of V and other values
  - store conditional of $V'$ into M
  - if store conditional succeeded exit loop

- **Supported on Alpha, PowerPC, MIPS, and ARM**
Test & Set Lock

define type Lock = (unlocked, locked)

procedure acquire_lock(Lock *L)
  loop
    // NOTE: test and set returns old value
    if test_and_set(L) = unlocked
      return

procedure release_lock(Lock *L)
  *L := unlocked
Test & Set Lock Notes

• Space: \( n \) words for \( n \) locks and \( p \) processes

• Lock acquire properties
  — spin waits using atomic read-modify-write

• Starvation theoretically possible; unlikely in practice

• Poor scalability
  — continual updates to a lock cause heavy network traffic
    — on cache-coherent machines, each update causes an invalidation
Test & Set Lock with Exponential Backoff

type Lock = (unlocked, locked)

procedure acquire_lock(Lock *L)
    delay : integer := 1

    // NOTE: test and set returns old value
    while test_and_set(L) = locked
        pause(delay) // wait this many units of time
        delay := delay * 2 // double delay each time

procedure release_lock(Lock *L)
    *L := unlocked

Tom Anderson, IEEE TPDS, January 1990
Test & Set Lock with Exp. Backoff Notes

- Grants requests in unpredictable order
- Starvation is theoretically possible, but unlikely in practice
- Spins (with backoff) on remote locations
- Atomic primitives: test_and_set

- Pragmatics: need to cap probe delay to some maximum
Ticket Lock with Proportional Backoff

**Type definition:**

```plaintext
type Lock = record
    unsigned int next_ticket := 0
    unsigned int now_serving := 0
end
```

**Acquire lock procedure:**

```plaintext
procedure acquire_lock (Lock *L)
// NOTES: fetch_and_increment returns old value
// arithmetic overflow is harmless here by design
    unsigned int my_ticket :=
        fetch_and_increment(&L->next_ticket)
    loop
        // delay proportional to # customers ahead of me
        // NOTE: on most machines, subtraction works correctly despite overflow
        pause(my_ticket - L->now_serving)
        if (L->now_serving = my_ticket) return
```

**Release lock procedure:**

```plaintext
procedure release_lock (Lock *L)
    L->now_serving := L->now_serving + 1
```
Ticket Lock Notes

- Grants requests in FIFO order
- Spins (with backoff) on remote locations
- Atomic primitives: fetch_and_increment
Anderson’s Array-based Queue Lock

type Lock = record
    slots: array [0..numprocs -1] of (has_lock, must_wait)
        := (has_lock, must_wait, must_wait, ..., must_wait)
    // each element of slots should lie in a different memory module or cache line
    int next_slot := 0

    // parameter my_place, below, points to a private variable in an enclosing scope
procedure acquire_lock (Lock *L, int *my_place)
    *my_place := fetch_and_increment (&L->next_slot)
    if *my_place mod numprocs = 0
        // decrement to avoid problems with overflow; ignore return value
        atomic_add(&L->next_slot,-numprocs)
    *my_place := *my_place mod numprocs
    repeat while L->slots[*my_place] = must_wait // spin
        L->slots[*my_place] := must_wait // init for next time

procedure release_lock (Lock *L, int *my_place)
    L->slots[(*my_place + 1) mod numprocs] := has_lock
Anderson’s Lock

The diagram illustrates a lock mechanism with processes and their states.

- L: A lock with the value 5.
- p3: A process that attempts to acquire the lock.
- has_lock: A state indicating that the lock is available.
- must_wait: States indicating that processes must wait.
Anderson’s Lock Notes

• Grants requests in FIFO order
• Space: $O(pn)$ space for $p$ processes and $n$ locks
• Spins only on local locations on a cache-coherent machine
• Atomic primitives: fetch_and_increment and atomic_add

IEEE TPDS, January 1990
The MCS List-based Queue Lock

type qnode = record
    qnode *next
    bool locked

procedure acquire_lock (Lock *L, qnode *I)
    I->next := nil
    qnode *predecessor := fetch_and_store (L, I)
    if predecessor != nil // queue was non-empty
        I->locked := true
        predecessor->next := I
        repeat while I->locked // spin

procedure release_lock (Lock *L, qnode *I)
    if I->next = nil // no known successor
        if compare_and_swap (L, I, nil) return
        // compare_and_swap returns true iff it stored
        repeat while I->next = nil // spin
MCS Lock In Action - I

Process 4 arrives, attempting to acquire lock
MCS Lock In Action - II

- Process 4 swaps self into tail pointer
- Acquires pointer to predecessor (3) from swap on tail
- 3 can’t leave without noticing that one or more successors will link in behind it because the tail no longer points to 3
MCS Lock In Action - III

4 links behind predecessor (3)
MCS Lock In Action - IV

4 links now spins until 3 signals that the lock is available by setting a flag in 4’s lock record
**MCS Lock In Action - V**

- **Process 1 prepares to release lock**
  - If it’s next field is set, signal successor directly
  - Suppose 1’s next pointer is still null
    - Attempt a compare_and_swap on the tail pointer
    - Finds that tail no longer points to self
    - Waits until successor pointer is valid (already points to 2 in diagram)
    - Signal successor (process 2)
procedure release_lock (Lock *L, qnode *I)
    if I->next = nil    // no known successor
        qnode *old_tail := fetch_and_store(L, nil)
        if old_tail = I    // I really had no successor
            return
        // we accidentally removed requestor(s) from the queue; we must put them back
        usurper := fetch_and_store(L, old_tail)
        repeat while I->next = nil    // wait for pointer to victim list
            if usurper != nil
                // somebody got into the queue ahead of our victims
                usurper->next := I->next    // link victims after the last usurper
            else
                I->next->locked := false
            else
                I->next->locked := false
**MCS Release Lock without CAS - I**

- 1 is running
- 2 and 3 begin to execute acquire protocol
  - 2 swaps tail to point to self; acquires pointer to 1
  - 3 swings tail to self, links behind 2 and begins to spin
  - 2 prepares to complete arrival bookkeeping by linking behind 1
MCS Release Lock without CAS - II

- 1 begins release lock protocol before 2 links behind
- finds successor null; executes swap on tail pointer to set it to null
1 finds that tail did not point to 1
—there are one or more others out there who have initiated an acquire on the lock (e.g. 2 and 3)
MCS Release Lock without CAS - IV

- 4 arrives, finds tail null and acquires the lock
- 5 arrives and queues behind 4
- 2 finishes linking behind 1 and starts to spin
- 2 and 3 disengaged from the lock queue
1 swaps 3 into tail and acquires a pointer to 5, indicating that others have acquired the lock since 1 cleared the tail pointer.

5 will not be able to finish until someone links behind him since he is no longer at the tail.
MCS Release Lock without CAS - VI

- 1 finishes by linking 2 behind 5
MCS Lock Notes

• Grants requests in FIFO order
• Space: \(2p + n\) words of space for \(p\) processes and \(n\) locks
• Requires a local "queue node" to be passed in as a parameter
  — alternatively, additional code can allocate these dynamically in acquire_lock, and look them up in a table in release_lock
• Spins only on local locations
  — cache-coherent and non-cache-coherent machines
• Atomic primitives
  — fetch_and_store and (ideally) compare_and_swap

ASPLOS, April 1991
ACM TOCS, February 1991
Impact of the MCS Lock

- Local spinning technique bounds remote memory traffic
  — influenced virtually all practical synchronization algorithms since
- Shifted the debate regarding hardware support
  — synchronization was identified as causing tree saturation
  — hardware support for avoiding contention was no longer essential
    - e.g. combining networks in NYU Ultracomputer, IBM RP3
  — hardware support became about reducing constant factor of overhead
- Widely used
  — e.g., monitor locks used in Java VMs are variants of MCS
type qnode = record
    qnode *prev
    Boolean succ_must_wait

type qnode *Lock // initialized to point to an unowned qnode

procedure acquire_lock(Lock *L, qnode *I)
    I->succ_must_wait := true
    qnode *pred := I->prev := fetch_and_store(L, I)
    repeat while pred->succ_must_wait

procedure release_lock(qnode **I)
    qnode *pred := *I->prev
    *I->succ_must_wait := false
    *I := pred // take pred's qnode
run spin spin spin

CLH
CLH Queue Lock Notes

• Discovered twice, independently
  — Travis Craig (University of Washington)
    – TR 93-02-02, February 1993
  — Anders Landin and Eric Hagersten (Swedish Institute of CS)
    – IPPS, 1994

• Space: $2p + 3n$ words of space for $p$ processes and $n$ locks
  — MCS lock requires $2p + n$ words

• Requires a local "queue node" to be passed in as a parameter

• Spins only on local locations on a cache-coherent machine

• Local-only spinning possible when lacking coherent cache
  — can modify implementation to use an extra level of indirection
    (local spinning variant not shown)

• Atomic primitives: fetch_and_store
Case Study:
Evaluating Lock Implementations for the BBN Butterfly and Sequent Symmetry

Sequent Symmetry

- 16 MHz Intel 80386
- Up to 30 CPUs
- 64KB 2-way set associative cache
- Snoopy coherence
- Various logical and arithmetic ops
  — no return values, condition codes only
Lock Comparison (Selected Locks Only)

Sequent Symmetry: shared-bus, coherent caches

![Diagram showing the performance of different lock algorithms across varying numbers of processors. The y-axis represents time in microseconds, and the x-axis represents the number of processors. The legend includes lines for test & test & set, anderson, test & set, exp. backoff, and mcs.](image-url)
Lock Comparison (Selected Locks Only)

Sequent Symmetry: shared-bus, coherent caches

![Diagram showing performance comparison of different lock algorithms across varying numbers of processors.](image)
BBN Butterfly

- 8 MHz MC68000
- 24-bit virtual address space
- 1-4 MB memory per PE
- $\log_4$ depth switching network
- Packet switched, non-blocking
- Remote reference
  - 4us (no contention)
  - 5x local reference
- Collisions in network
  - 1 reference succeeds
  - others aborted and retried later
- 16-bit atomic operations
  - fetch_clear_then_add
  - fetch_clear_then_xor
Lock Comparison

BBN Butterfly: distributed memory, no coherent caches

empty critical section
Lock Comparison (Selected Locks Only)

BBN Butterfly: distributed memory, no coherent caches
Case Study:
Evaluating Lock Implementations for the IBM Cyclops C64

IBM Cyclops-64 Node

- 80 processors: 2 thread units (TU) each, 1 floating point (FP)
- No data cache; 32K instruction cache per 5 PEs
- Scratchpad memory (SP) + global memory (GM)
Lock Implementations on the C64

- Test-and-set: threads spin on values in global memory
  - plain (TS) and exponential backoff (TS-exp)
  - threads spin on values in global memory
- Ticket: threads spin on values in global memory
- MCS: threads spin locally in scratch pad memory
- MCS with sleep/wakeup (MCS-SW)
  - thread suspends after adding itself to queue
  - constant # instructions per acquire/release pair
    - independent of # threads contending for lock
Evaluation Strategy

- Each thread performs 1K acquires and releases
- Evaluation benchmarks
  - lock-null: no delays
  - lock-delay: fixed delays
    - delay inside critical section = 3 x delay outside critical section
- Evaluation metrics
  - overhead

\[
\text{Overhead} = \begin{cases} 
\frac{\text{Execution Time}}{\text{No. Lock Acquires}} - D_i - D_o, & P = 1 \\
\frac{\text{Execution Time}}{\text{No. Lock Acquires}} - D_i, & P > 1 
\end{cases}
\]

- contention
  - measured with software simulator
  - two or more threads compete for same resource in same cycle
  - contention counter incremented
  - important because it affects execution as well as lock acquisition
C64 lock-null Overhead

- MCS and MCS-SW have lowest overhead
- MCS and MCS-SW have perfect scalability
C64 lock-delay Overhead

- MCS and MCS-SW have lowest overhead
- MCS and MCS-SW have perfect scalability
C64 lock-null Contention

Contention
—2 or more threads compete for same resource in same cycle
—normalized by the number of threads

MCS and MCS-SW have lowest contention
C64 lock-delay Contention

- **Contestation**
  - 2 or more threads compete for same resource in same cycle
  - normalized by the number of threads

- MCS and MCS-SW have lowest contention
C64 Lock Evaluation Summary

• MCS and MCS-SW are superior

• MCS-SW is preferable
  — for > 1 thread, MCS-SW overhead < MCS overhead by few cycles
  — sleeping instead of spin waiting reduces power consumption
References


