Mutual Exclusion: Classical Algorithms for Locks

Bill Scherer

Department of Computer Science
Rice University

scherer@cs.rice.edu
Motivation

Ensure that a block of code manipulating a data structure is executed by only one thread at a time

• Why? avoid conflicting accesses to shared data (data races)
  — read/write conflicts
  — write/write conflicts

• Approach: critical section

• Mechanism: lock
  — methods
    – acquire
    – release

• Usage
  — acquire lock to enter the critical section
  — release lock to leave the critical section
Problems with Locks

- **Conceptual**
  - coarse-grained: poor scalability
  - fine-grained: hard to write

- **Semantic**
  - deadlock
  - priority inversion

- **Performance**
  - convoying
  - intolerance of page faults and preemption
Lock Alternatives

• Transactional memory (TM)
  + Easy to use, well-understood metaphor
    – High overhead (so far)
  ± Subject of much active research

• Ad hoc nonblocking synchronization (NBS)
  + Thread failure/delay cannot prevent progress
  + Can be faster than locks (stacks, queues)
    – Notoriously difficult to write – every new algorithm is a publishable result
  + Can be “canned” in libraries (e.g. java.util)
Properties of Good Lock Algorithms

- Mutual exclusion (safety property)
  - critical sections of different threads do not overlap
    - cannot guarantee integrity of computation without this property

- No deadlock
  - if some thread attempts to acquire the lock, then some thread will acquire the lock

- No starvation
  - every thread that attempts to acquire the lock eventually succeeds
    - implies no deadlock

Notes

- Deadlock-free locks do not imply a deadlock-free program
  - e.g., can create circular wait involving a pair of “good” locks

- Starvation freedom is desirable, but not essential
  - practical locks: many permit starvation, although it is unlikely to occur

- Without a real-time guarantee, starvation freedom is weak property
Topics for Today

Classical locking algorithms using load and store

• Steps toward a two-thread solution
  —two partial solutions and their properties

• Peterson’s algorithm: a two-thread solution

• Filter lock: generalized Peterson
Classical Lock Algorithms

• Use atomic load and store only, no stronger atomic primitives

• Not used in practice
  — locks based on stronger atomic primitives are more efficient

• Why study classical algorithms?
  — understand the principles underlying synchronization
    – subtle
    – such issues are ubiquitous in parallel programs
Toward a Classical Lock for Two Threads

• First, consider two inadequate but interesting lock algorithms
  —use load and store only

• Assumptions
  —only two threads
  —each thread has a unique value of self_threadid ∈ {0,1}
class Lock1: public Lock {
private:
    volatile bool flag[2];
public:
    void acquire() {
        int other_threadid = 1 - self_threadid;
        flag[self_threadid] = true;
        while (flag[other_threadid] == true);
    }
    void release() {
        flag[self_threadid] = false;
    }
}
Using Lock1

assume that initially both flags are false

flag[0] = true
while(flag[1] == true);
flag[0] = false

flag[1] = true
while(flag[0] == true);
flag[1] = false

CS0

CS1

wait

thread 0

thread 1
Using Lock1

thread 0
flag[0] = true
while(flag[1] == true):
  wait

thread 1
flag[1] = true
while(flag[0] == true):
  wait

deadlock!
Summary of Lock1 Properties

• If one thread executes acquire before the other, works fine
  — Lock1 provides mutual exclusion

• However, Lock1 is inadequate
  — if both threads write flags before either reads → deadlock
class Lock2: public Lock {
    private:
        volatile int victim;
    public:
        void acquire() {
            victim = self_threadid;
            while (victim == self_threadid); // busy wait
        }
        void release() {
        }
}
Using Lock2

thread 0

victim = 0
while(victim == 0);

victim = 0
while(victim == 0);

wait

thread 1

victim = 1
while(victim == 1);

wait
Using Lock2

thread 0

victim = 0

while(victim == 0);

wait

deadlock!
Summary of Lock2 Properties

• If the two threads run concurrently, acquire succeeds for one
  —provides mutual exclusion
• However, Lock2 is inadequate
  —if one thread runs before the other, it will deadlock
Combining the Ideas

Lock1 and Lock2 complement each other

• Each succeeds under conditions that causes the other to fail
  —Lock1 succeeds when CS attempts do not overlap
  —Lock2 succeeds when CS attempts do overlap

• Design a lock protocol that leverages the strengths of both...
Peterson’s Algorithm: 2-way Mutual Exclusion

class Peterson: public Lock {
    private:
        volatile bool flag[2];
        volatile int victim;
    public:
        void acquire() {
            int other_threadid = 1 - self_threadid;
            flag[self_threadid] = true;  // I’m interested
            victim = self_threadid       // you go first
            while (flag[other_threadid] == true &&
                    victim == self_threadid);
        }
        void release() {
            flag[self_threadid] = false;
        }
}
Peterson’s Lock: Serialized Acquires

thread 0
flag[0] = true
victim = 0
while(flag[1] == true && victim == 0);
flag[0] = false

thread 1
flag[1] = true
victim = 1
while(flag[0] == true && victim == 1);
flag[1] = false

CS₀
wait

CS₁
Peterson’s Lock: Concurrent Acquires

**thread 0**

flag[0] = true
victim = 0

while(flag[1] == true && victim == 0);

flag[0] = false

**thread 1**

flag[1] = true
victim = 1

while(flag[0] == true && victim == 1);

flag[1] = false

wait
From 2-way to N-way Mutual Exclusion

• Peterson’s lock provides 2-way mutual exclusion
• How can we generalize to N-way mutual exclusion, N > 2?
• Filter lock: direct generalization of Peterson’s lock
class Filter: public Lock {
    private:
        volatile int level[N]; volatile int victim[N-1];
    public:
        void acquire() {
            for (int j = 1; j < N; j++) {
                level [self_threadid] = j;
                victim [j] = self_threadid;
                // wait while conflicts exist
                while (sameOrHigher(self_threadid,j) &&
                    victim[j] == self_threadid);
            }
        }
        bool sameOrHigher(int i, int j) {
            for(int k = 0; k < N; k++)
                if (k != i && level[k] >= j) return true;
            return false;
        }
        void release() {
            level[self_threadid] = 0;
        }
};
Understanding the Filter Lock

- Peterson’s lock used two-element Boolean flag array
- Filter lock generalization: an N-element integer level array
  - value of level[k] = highest level thread k is interested in entering
  - each thread must pass through N-1 levels of exclusion
- Each level has its own victim flag to filter out 1 thread, excluding it from the next level
  - natural generalization of victim variable in Peterson’s algorithm
- Properties of levels
  - at least one thread trying to enter level k succeeds
  - if more than one thread is trying to enter level k, then at least one is blocked
- For proofs, see Herlihy and Shavit’s manuscript
References


Lock Synchronization with Atomic Primitives

Bill Scherer
Department of Computer Science
Rice University
scherer@cs.rice.edu
Topics for Today

• Atomic primitives for synchronization

• Lock algorithms using atomic primitives
  — test-and-set lock
  — test-and-set with exponential backoff
  — Array-based queue locks
  — MCS list-based queue lock
  — CLH list-based queue lock

• Case study: performance of lock implementations
  — BBN Butterfly and Sequent Symmetry
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
  - execute sequence of instructions performing arbitrary computation on 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

type lock = (unlocked, locked)

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

procedure release_lock (L : ^lock)
  L^ := unlocked
Test & Test & Set (TATAS) Lock

type lock = (unlocked, locked)

procedure acquire_lock (L : ^lock)
  loop
    // NOTE: test and set returns old value
    if test_and_set (L) = unlocked
      return
    else
      loop
        until L^ <> locked
  end loop

procedure release_lock (L : ^lock)
  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
  — Fairness, however can be very uneven

• Poor scalability
  — continual updates to a lock cause heavy network traffic
    — on cache-coherent machines, each update causes an invalidation
  — Improved with TATAS variant, but still a big spike on each release of the lock, even on cache-coherent machines
Test & Set Lock with Exponential Backoff

type lock = (unlocked, locked)

procedure acquire_lock (L : ^lock)
  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 (L : ^lock)
  L^ := unlocked
Test & Set Lock with Exp. Backoff Notes

• Similar to code developed by Tom Anderson
• 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

IEEE TPDS, January 1990
Array-based Lock Notes

• Grants requests in FIFO order
• Space: $O(pn)$ space for $p$ processes and $n$ locks
The MCS List-based Queue Lock

type qnode = record
  next : ^qnode
  locked : Boolean
end

type lock = ^qnode  // initialized to nil

// parameter I, below, points to a qnode record allocated (in an enclosing scope) in
// shared memory locally-accessible to the invoking processor
procedure acquire_lock (L : ^lock, I : ^qnode)
  I->next := nil
  predecessor : ^qnode := 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 (L : ^lock, I: ^qnode)
  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
    I->next->locked := false
MCS Lock In Action - I

Process 4 arrives, attempting to acquire lock
• Process 4 swaps self into tail pointer
• Acquires pointer to predecessor (3) from swap on tail
• Note: 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

run  spin  spin  arriving

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)
MCS Lock In Action - VI

leaving  run  spin  spin  tail
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
Impact of the MCS Lock

• Key lesson: importance of reducing memory traffic in synchronization
  —local spinning technique influenced virtually all practical scalable synchronization algorithms since

• 2006 Edsger Dijkstra Prize in distributed computing
  —“an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade”
  —“probably the most influential practical mutual exclusion algorithm ever”
  —“vastly superior to all previous mutual exclusion algorithms”
  —fast, scalable, and fair in a wide variety of multiprocessor systems
  —avoids need to pre-allocate memory for a fixed, maximum # of threads
  —widely used: e.g., monitor locks used in Java VMs are variants of MCS
CLH List-based Queue Lock

type qnode = record
    prev : ^qnode
    succ_must_wait : Boolean

type lock = ^qnode  // initialized to point to an unowned qnode

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

procedure release_lock (ref I : ^qnode)
    pred : ^qnode := I->prev
    I->succ_must_wait := false
    I := pred  // take pred's qnode
CLH Lock In Action

run → spin → spin → spin → tail
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

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
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

BBN Butterfly: distributed memory, no coherent caches

empty critical section
Lock Comparison (Selected Locks Only)

BBN Butterfly: distributed memory, no coherent caches
Lock Comparison (Selected Locks Only)

Sequent Symmetry: shared-bus, coherent caches

![Graph showing Time (μs) vs Processors for different lock algorithms.](image)
Lock Comparison (Selected Locks Only)

Sequent Symmetry: shared-bus, coherent caches
References


• Travis Craig, Building FIFO and priority queuing spin locks from atomic swap. University of Washington, Dept. of Computer Science, TR 93-02-02, Feb. 1993.

Lemma: For $0 \leq j \leq n-1$, there are at most $n - j$ threads at level $j$

• Proof by induction on $j$.

• Base case: $j = 0$ is trivially true.

• Induction hypothesis: at most $n-j+1$ threads at level $j-1$

• Induction step:
  — show that at least one thread cannot progress to level $j$
  — argue by contradiction: assume there are $n-j+1$ threads at level $j$
    – let $A$ be the last thread at level $j$ to write to victim[$j$]
    – because $A$ is last, for any other $B$ at level $j$
      
      \[ \text{write}_B(\text{victim}[j] = B) \rightarrow \text{write}_A(\text{victim}[j] = A) \]
• **Evaluation criteria**
  — hardware support
  — performance: latency, throughput
  — fairness

• **Mutual exclusion**
  — load-store based protocols
  — test and set locks
  — ticket locks
  — queuing locks

• **Barriers**
  — centralized barriers: counters and flags
  — software combining trees
  — tournament barrier
  — dissemination barrier

• **Problems and solutions**
  — re-initialization via sense switching
  — handling counter overflow
Maintain the integrity of shared data structures

• Goal: avoid conflicting updates
  —read/write conflicts
  —write/write conflicts