Mutual Exclusion:
Classical Algorithms for Locks

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Motivation

• Problem: ensure that a data structure is maintained consistently
  — avoid conflicting accesses to shared data (data races)
    – read/write conflicts
    – write/write conflicts

• Locks guarantee consistency by providing exclusion
  — acquire lock before manipulating the shared data
  — release lock when finished manipulating the shared data
Problems with Locks

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

- **Semantic**
  - deadlock
  - priority inversion

- **Performance**
  - intolerance of page faults and preemption
Alternatives to Locks

- **Transactional memory (TM)**
  - support arbitrary atomic actions on multi-word shared data
  ```c
  atomic (entries > 0) {
    node *first = head; head = head->next;
    entries--; return first;
  }
  ```
  - transactions that don’t conflict run uninterrupted in parallel
  - transactions that conflict abort and retry
    - benefit: no need for programmer to worry about deadlock!
    - cost: repeated aborts can waste resources and hurt performance
  + easy to use, well-understood metaphor
  - high overhead in software; HTM on Blue Gene/Q, Intel Haswell, IBM Power8
  ± subject of much active research

- **Ad hoc non-blocking synchronization (NBS)**
  + thread failure/delay cannot prevent progress
  + can be faster than locks (stacks, queues)
  - difficult to write: every new algorithm is a publishable result
  + can be “canned” in libraries (e.g. java.util.concurrent’s ConcurrentLinkedQueue)
Synchronization Landscape

Programmer Effort

System Performance

Figure credit: William Scherer
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 a 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
- Tree lock for n threads
- Lamport’s bakery lock for n threads
- Performance evaluation
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 lock algorithms?
  - understand the principles underlying synchronization
    - ubiquitous in parallel programs
  - appreciate their subtlety
  - understand the motivation for atomic operations in hardware
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

假设初始时两个标志都为假

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

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

线程 0
- `CS_0`
- `wait`
- `CS_1`
- `flag[1] = false`
Lock1 Provides Mutual Exclusion

Proof

• Suppose not. Then \( \exists j, k \in \text{integers} \)
  \[ CS_0^j \not\rightarrow CS_1^k \quad \text{and} \quad CS_1^k \not\rightarrow CS_0^j \]

• Consider each thread’s acquire before its \( j^{th} \) (\( k^{th} \)) critical section
  \[
  \begin{align*}
  \text{write}_0(\text{flag}[0] = \text{true}) & \rightarrow \text{read}_0(\text{flag}[1] == \text{false}) \rightarrow CS_0 \quad (1) \\
  \text{write}_1(\text{flag}[1] = \text{true}) & \rightarrow \text{read}_1(\text{flag}[0] == \text{false}) \rightarrow CS_1 \quad (2)
  \end{align*}
  \]

• However, once \( \text{flag}[1] == \text{true} \), it remains \( \text{true} \) while thread 1 in \( CS_1 \)

• So (1) could not hold unless
  \[
  \begin{align*}
  \text{read}_0(\text{flag}[1] == \text{false}) & \rightarrow \text{write}_1(\text{flag}[1] = \text{true}) \quad (3)
  \end{align*}
  \]

• From (1), (2), and (3)
  \[
  \begin{align*}
  \text{write}_0(\text{flag}[0] = \text{true}) & \rightarrow \text{read}_0(\text{flag}[1] == \text{false}) \rightarrow \\
  \text{write}_1(\text{flag}[1] = \text{true}) & \rightarrow \text{read}_1(\text{flag}[0] == \text{false}) \\
  \end{align*}
  \]

• By (4) \( \text{write}_0(\text{flag}[0] = \text{true}) \rightarrow \text{read}_1(\text{flag}[0] == \text{false}) \): a contradiction
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

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

deadlock!

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

wait
Summary of Lock1 Properties

- Lock1 guarantees mutual exclusion
- Works if one thread completes its acquire before the other
- Deadlock if both threads write flags before either reads
- Since it admits deadlock, Lock1 is inadequate
Lock2

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);

wait

victim = 0
while(victim == 0);

wait

thread 1

victim = 1
while(victim == 1);

wait
Lock2 Provides Mutual Exclusion

Proof

- Suppose not. Then \( \exists j, k \in \text{integers} \)
  \[ CS_0^j \leftrightarrow CS_1^k \quad \text{and} \quad CS_1^k \leftrightarrow CS_0^j \]

- Consider each thread’s acquire before its \( j^{\text{th}} \) \( (k^{\text{th}}) \) critical section
  \[
  \begin{align*}
  \text{write}_0(\text{victim} = 0) & \rightarrow \text{read}_0(\text{victim} \neq 0) \rightarrow CS_0 \quad (1) \\
  \text{write}_1(\text{victim} = 1) & \rightarrow \text{read}_1(\text{victim} \neq 1) \rightarrow CS_1 \quad (2)
  \end{align*}
  \]

- For thread 0 to enter the critical section, thread 1 must assign \( \text{victim} = 1 \)
  \[
  \begin{align*}
  \text{write}_0(\text{victim} = 0) & \rightarrow \text{write}_1(\text{victim} = 1) \rightarrow \text{read}_0(\text{victim} \neq 0) \rightarrow CS_0 \quad (3)
  \end{align*}
  \]

- Once \( \text{write}_1(\text{victim} = 1) \) occurs, \( \text{victim} \) does not change

- Therefore, thread 1 cannot \( \text{read}_1(\text{victim} \neq 1) \) and enter \( CS_1 \)

- Contradiction!

```c
void acquire() {
    victim = self_threadid;
    while (victim == self_threadid); // busy wait
}
```
Lock2 protocol
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);

wait

deadlock!
Summary of Lock2 Properties

• Guarantees mutual exclusion
• If two threads run concurrently: acquire succeeds for one
• Deadlock if one thread runs before the other
• Since it admits deadlock, Lock2 is inadequate
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

```
flag[0] = true
victim = 0
while(flag[1] == true && victim == 0);
flag[0] = false
CS_0

flag[0] = false

flag[1] = true
victim = 1
while(flag[0] == true && victim == 1);
flag[1] = false
CS_1

wait
```

thread 0

thread 1
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

CS₀
flag[0] = false

CS₁
flag[1] = false
Peterson’s Algorithm Provides Mutual Exclusion

- Suppose not. Then $\exists j, k \in \text{integers}$
  \[ CS_0^j \not\leftrightarrow CS_1^k \text{ and } CS_1^k \not\leftrightarrow CS_0^j \]
- Consider each thread’s lock op before its $j^{th}$ ($k^{th}$) critical section
  \[
  \text{write}_0(\text{flag}[0] = \text{true}) \rightarrow \text{write}_0(\text{victim} = 0) \rightarrow \\
  \text{read}_0(\text{flag}[1] == \text{false}) \text{ or } \text{read}_0(\text{victim} != 0) \rightarrow CS_0 \tag{1}
  \]
  \[
  \text{write}_1(\text{flag}[1] = \text{true}) \rightarrow \text{write}_1(\text{victim} = 1) \rightarrow \\
  \text{read}_1(\text{flag}[0] == \text{false}) \text{ or } \text{read}_1(\text{victim} != 1) \rightarrow CS_1 \tag{2}
  \]
- Without loss of generality, assume thread 0 was the last to write victim
  \[
  \text{write}_1(\text{victim} = 1) \rightarrow \text{write}_0(\text{victim} = 0) \tag{3}
  \]
- From (1), (2), and (3), thread 0 must read $\text{victim} == 0$ in (1)
- Since thread 0 nevertheless enters its CS, it must have read $\text{flag}[1]==\text{false}$
- From (1), it must be the case that $\text{write}_0(\text{victim} = 0) \rightarrow \text{read}_0(\text{flag}[1] == \text{false})$
- From (1), (2), and (3) and transitivity,
  \[
  \text{write}_1(\text{flag}[1] = \text{true}) \rightarrow \text{write}_1(\text{victim} = 1) \rightarrow \\
  \text{write}_0(\text{victim} = 0) \rightarrow \text{read}_0(\text{flag}[1] == \text{false}) \tag{4}
  \]
- From (4), it follows that $\text{write}_1(\text{flag}[1] = \text{true}) \rightarrow \text{read}_0(\text{flag}[1] == \text{false})$
- Contradiction!
Peterson’s Algorithm is Starvation-Free

• Suppose not: WLG, suppose that thread 0 waits forever in acquire
  — it must be executing the while statement
    – waiting until flag[1] == false or victim != 0

• What is thread 1 doing while thread 0 fails to make progress?
  — perhaps outside the critical section
    – flag[1] == true only if thread 1 is awaiting or in the critical section
      contradiction!
  — perhaps entering and leaving the critical section
    – if so, thread 1 will set victim to 1 when it tries to re-enter the CS
    – once it is set to 1, it will not change
    – thus, thread 0 must eventually return from acquire
      contradiction!
  — waiting in acquire as well
    – waiting for flag[0] == false or victim == 0
    – victim cannot be both 1 and 0, thus both threads cannot wait
      contradiction!

• Corollary: Peterson’s lock is deadlock-free as well
Dekker’s Algorithm

```plaintext
"begin integer c1, c2, turn;
    c1:= 1; c2:= 1; turn:= 1;
parbegin
  process 1: begin A1: c1:= 0;
      L1: if c2 = 0 then
          begin if turn = 1 then goto L1;
              c1:= 1;
          end;
      B1: if turn = 2 then goto B1;
          goto A1
      end;
    critical section 1;
    turn:= 2; c1:= 1;
    remainder of cycle 1; goto A1
  end;
  process 2: begin A2: c2:= 0;
      L2: if c1 = 0 then
          begin if turn = 2 then goto L2;
              c2:= 1;
          end;
      B2: if turn = 1 then goto B2;
          goto A2
      end;
    critical section 2;
    turn:= 1; c2:= 1;
    remainder of cycle 2; goto A2
  end
end
end"
```

Dijkstra, Edsger W. *Cooperating sequential processes (EWD-123)* (PDF). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (transcription) (September 1965)
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, $N > 2$?
- Several strategies that are generalizations of Peterson’s lock
An N-way Lock as a Tree of Peterson Locks

- For a lock involving N threads, construct a balanced binary tree with N/2 leaves. Assume N = \(2^k\)
- Each thread uses Peterson’s lock to compete against another thread in a leaf node of the tree
- When a thread acquires a lock, it moves up the tree to compete for the parent lock
- When a thread acquires the root lock, it may enter the critical section
- When a thread exits the critical section, it releases locks along the path from the root to its leaf
Properties of Tree of Peterson Locks

- $O(N)$ space
  - if $N = 2^k$, there are $2^{k-1}$ leaves and $N-1$ nodes in total
- $\lg N$ steps to acquire or release the lock
class LamportBakery: public Lock {
    private:
        volatile bool flag[N]; volatile Label label[N];
    public:
        void acquire() {
            int i = self_threadid;
            flag[i] = true;
            label[i] = max(label[0], ..., label[N-1]) + 1;
            while (exists k != i such that
                flag[k] && <label[k],k> <L <label[i],i> );
        }
        void release() {
            flag[self_threadid] = 0;
        }
    }

lexicographic ordering of <label, thread_id> tuples;
thread id is used in tuple to break labeling ties
Bakery Algorithm Intuition

• Data structure components
  —flag[A] = Boolean that indicate whether A wants to enter the CS
  —label[A] = integer that indicates the thread’s turn to enter the bakery

• Protocol operation
  —when a thread tries to acquire the lock, it generates a new label
    – reads all other thread labels in some arbitrary order
    – generates a label greater than the largest it read
    – notes:
      if 2 threads select labels concurrently, they may get the same
  —algorithm uses lexicographical order on pairs of (label, thread_id)
    – (label[j], j) <L (label[k],k)
      iff (label[j] < label[k]) || ((label[j] == label[k]) && j < k)
  —in the waiting phase
    – a thread repeatedly rereads the labels
    – waits until
      no thread with its flag set has a smaller (label, thread_id) pair

• Proofs: See Herlihy and Shavit manuscript (deadlock-free, FIFO, ME)
Observations

• Bakery algorithm is concise, elegant and fair

• Why is it not practical?
  —must read N distinct locations; N could be very large
  —threads must be assigned unique ids between 0 and n-1
    – awkward for dynamic threads
  —value of a label is monotonically increasing & unbounded

• Are locking algorithms based on load/store commonly used?
  —no.
  —minimum space O(N)
  —uncontended acquisition latency is O(lg N)

• Atomic primitives enable locks with
  —constant space
  —constant time acquisition in the uncontended case
  —maximum number of threads need not be known in advance
Spin Lock Performance: Maximal Contention

- Peterson-Buhr is a tree of Peterson’s 2-party locks using load/store
- Spinlock uses test-and-set and exponential backoff
- MCS lock uses SWAP and CAS

Figure credit: Peter A. Buhr, David Dice and Wim H. Hesselink. High-performance N-thread software solutions for mutual exclusion. Concurrency and Computation: Practice and Experience, 2014.
Spin Lock Performance: Minimal Contention

- Peterson-Buhr is a tree of Peterson’s 2-party locks using load/store
- Spinlock uses test-and-set and exponential backoff
- MCS lock uses SWAP and CAS

Figure credit: Peter A. Buhr, David Dice and Wim H. Hesselink. High-performance N-thread software solutions for mutual exclusion. Concurrency and Computation: Practice and Experience, 2014.
References


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 expressed interest 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 book