Design of Parallel and High-Performance Computing
Fall 2016
Lecture: Languages and Locks

Motivational video: https://www.youtube.com/watch?v=1o4YViBAGU0

Instructor: Torsten Hoefler & Markus Püschel
TAs: Salvatore Di Girolamo

Administrivia
- Final project presentation: Monday 12/18 during last lecture
  - Report will be due in January!
  - Still, starting to write early is very helpful — write — rewrite — rewrite (no joke!)
- Some more ideas what to talk about:
  - What tools/programming language/parallelization scheme do you use?
  - Which architecture? (we only offer access to Xeon Phi, you may use different)
  - How to verify correctness of the parallelization?
  - How to argue about performance (bounds, what to compare to?)
  - (Somewhat) realistic use-cases and input sets?
  - What are the key concepts employed?
  - What are the main obstacles?

Review of last lecture
- Locked Queue
  - Correctness
  - Lock-free two-thread queue
- Linearizability
  - Combine object pre- and postconditions with serializability
  - Additional (semantic) constraints!
- Histories
  - Analyze given histories
  - Projections, Sequential/Concurrent, Completeness, Equivalence, Well-formed, Linearizability (formal)

Peer Quiz
- Instructions:
  - Pick some partners (locally) and discuss each question for 2 minutes
  - We then select a random student (team) to answer the question
- How can histories be used to prove a parallel code correct?
- How do histories relate to the source code?
- Can proofing be automated?
- What are the practical limits of linearizability?
  - Can it always be applied?
  - Is there a performance tradeoff? Always? Sometimes? Never?

DPHPC Overview

Goals of this lecture
- Languages and Memory Models
  - Java/C++ definition
- Recap sequential consistency
  - Races (now in practice)
- Mutual exclusion
- Locks
  - Two-thread
  - Peterson
  - N-thread
  - Many different locks, strengths and weaknesses
  - Lock options and parameters
- Problems and outline to next class
Language Memory Models

- Which transformations/reorderings can be applied to a program
- Affects platform/system
  - Compiler, (VM), hardware
- Affects programmer
  - What are possible semantics/output
  - Which communication between threads is legal?
- Without memory model
  - Impossible to even define “legal” or “semantics” when data is accessed concurrently
- A memory model is a contract
  - Between platform and programmer

History of Memory Models

- Java’s original memory model was broken [1]
  - Difficult to understand => widely violated
  - Did not allow reorderings as implemented in standard VMs
  - Final fields could appear to change value without synchronization
  - Volatile writes could be reordered with normal reads and writes => counter-intuitive for most developers
- Java memory model was revised [2]
  - Java 1.5 (JSR-133)
  - Still some issues (operational semantics definition [3])
- C++ didn’t even have a memory model until much later
  - Not able to make any statement about threaded semantics!
  - Introduced in C++11 and C11
  - Based on experience from Java, more conservative

Everybody wants to optimize

- Language constructs for synchronization
  - Java: volatile, synchronized, ...
  - C++: atomic, [NOT volatile], mutex, ...
- Without synchronization (defined language-specific)
  - Compiler, (VM), architecture
  - Reorder and appear to reorder memory operations
  - Maintain sequential semantics per thread
  - Other threads may observe any order (have seen examples before)

Java and C++ High-level overview

- Relaxed memory model
  - No global visibility ordering of operations
  - Allows for standard compiler optimizations
- But
  - Program order for each thread (sequential semantics)
  - Partial order on memory operations (with respect to synchronizations)
  - Visibility function defined
- Correctly synchronized programs
  - Guarantee sequential consistency
- Incorrectly synchronized programs
  - Java: maintain safety and security guarantees
    Type safety etc. (require behavior bounded by causality)
  - C++: undefined behavior
    No safety (anything can happen/change)

Communication between threads: Intuition

- Not guaranteed unless by:
  - Synchronization
  - Volatile/atomic variables
  - Specialized functions/classes (e.g., java.util.concurrent, ...)

Flag is a synchronization variable (atomic in C++, volatile in Java), i.e., all memory written by T1 must be visible to T2 after it reads the value true for flag!

Recap: Memory Model (Intuition)

- Abstract relation between threads and memory
  - Local thread view!
- Does not talk about classes, objects, methods, ...
  - Linearizability is a higher-level concept!
Lock synchronization

- **Java**
  - Synchronized methods as syntactic sugar

- **C++ (RAII)**
  - Many flexible variants

<table>
<thead>
<tr>
<th>Semantics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutual exclusion</td>
</tr>
<tr>
<td>at most one thread may hold a lock at a time</td>
</tr>
<tr>
<td>a thread B trying to acquire a lock held by thread A blocks until thread A releases the lock</td>
</tr>
<tr>
<td>note: threads may wait forever (no progress guarantee!)</td>
</tr>
</tbody>
</table>

Memory semantics

- Similar to synchronization variables
  - All memory accesses before an unlock ...
  - are ordered before and are visible to ...
  - any memory access after a matching lock!

Synchronization variables

- Variables can be declared volatile (Java) or atomic (C++)
- Reads and writes to synchronization variables
  - Are totally ordered with respect to all threads
  - Must not be reordered with normal reads and writes
- Compiler
  - Must not allocate synchronization variables in registers
  - Must not swap variables with synchronization variables
  - May need to issue memory fences/barriers
  - ...

Intuitive memory model rules

- Java/C++: Correctly synchronized programs will execute sequentially consistent
- Correctly synchronized = data-race free
  - if all sequentially consistent executions are free of data races
- Two accesses to a shared memory location form a data race in the execution of a program if
  - The two accesses are from different threads
  - At least one access is a write and
  - The accesses are not synchronized

Case Study: Locks - Lecture Goals

- Among the simplest concurrency constructs
  - Yet, complex enough to illustrate many optimization principles
- **Goal 1:** You understand locks in detail
  - Requirements / guarantees
  - Correctness / validation
  - Performance / scalability
- **Goal 2:** Acquire the ability to design your own locks
  - Understand techniques and weaknesses/traps
  - Extend to other concurrent algorithms
  - Issues are very much the same
- **Goal 3:** Feel the complexity of shared memory!
Preliminary Comments

- All code examples are in C/C++ style
  - Neither C nor C++ <11 have a clear memory model
  - C++ is one of the languages of choice in HPC
  - Consider source as exemplary (and pay attention to the memory model)
    In fact, you'll most likely not need those algorithms, but the principles will be useful!
- x86 is really only used because it is common
  - This does not mean that we consider the ISA or memory model elegant!
  - We assume atomic memory (or registers)!
    Usually given on x86 (easy to enforce)
- Number of threads/processes is p, tid is the thread id

Recap Concurrent Updates

- Multi-threaded execution!
  - Value of a for p=1?
  - Value of a for p>1?
    Why? Isn't it a single instruction?

\[
\begin{align*}
\text{const int n}=1000; \\
\text{volatile int a}=0; \\
\text{for \( i=0; i<n; ++i \)} & \\
\text{\( a++; \)}
\end{align*}
\]

\[
\begin{align*}
\text{const int n}=1000; \\
\text{std::atomic<int> a; a}=0; \\
\text{for \( i=0; i<n; ++i \)} & \\
\text{\( a++; \)}
\end{align*}
\]

Some Statistics

- Nondeterministic execution
  - Result depends on timing (probably not desired)
- What do you think are the most significant results?
  - Running two threads on Core i5 dual core

Some Statistics

- (recap) two memory accesses conflict if they can happen at the same time
  (in happens-before) and one of them is a write (store)
- Such a code is said to have a “race condition”
  - Also data-race
  - Trivia around races:
    The Therac-25 killed three people due to a race
    A data-race lead to a large blackout in 2003, leaving 55 million people without power causing $1bn damage
  - Can be avoided by critical regions
    - Mutually exclusive access to a set of operations

Conflicting Accesses

- Few instructions less! Performance!?
  - run with larger n (10^8)
  - Compiler: gcc version 4.9.2 (enabled experimental c++11 support, -O3)
  - Single-threaded execution only!
Mutual Exclusion

- Control access to a critical region
  - Memory accesses of all processes happen in program order (a partial order, many interleavings)
  - An execution history defines a total order of memory accesses
  - Some subsets of memory accesses (issued by the same process) need to happen atomically (thread a's memory accesses may not be interleaved with other thread's accesses)
  - To achieve linearizability!
  - To need to restrict the valid executions

- Requires synchronization of some sort
  - Many possible techniques (e.g., TM, CAS, T&S, …)
  - We first discuss locks which have wait semantics

- What must the functions lock and unlock guarantee?
  - #1: prevent two threads from simultaneously entering CR (i.e., accesses to CR must be mutually exclusive!)
  - #2: ensure consistent memory
  - i.e., stores must be globally visible before new lock is granted!
  - Any performance guesses (remember, 0.23s \( \rightarrow \) 0.78s for atomics)
    - 2.26s

Fixing it with locks

```
const int n=1000;
volatile int a=0;
omp_lock_t lck;
for (int i=0; i<n; ++i) {
    // lock
    omp_set_lock(&lck);
    a++;
    // leave
    omp_unset_lock(&lck);
}
```

- What must the functions lock and unlock guarantee?
  - #1: prevent two threads from simultaneously entering CR
  - i.e., accesses to CR must be mutually exclusive!
  - #2: ensure consistent memory
  - i.e., stores must be globally visible before new lock is granted!

- Desired Lock Properties
  - Mutual exclusion
    - Only one thread is on the critical region
  - Consistency
    - Memory operations are visible when critical region is left
  - Progress
    - If any thread a is not in the critical region, it cannot prevent another thread b from entering
  - Starvation-freedom (implies deadlock-freedom)
    - If a thread is requesting access to a critical region, then it will eventually be granted access
  - Fairness
    - A thread a requested access to a critical region before thread b. Did is also granted access to this region before b?
  - Performance
    - Scaling to large numbers of contending threads

Simplified Notation (cf. Histories)

- Time defined by precedence (a total order on events)
  - Events are instantaneous (linearizable)
  - Threads produce sequences of events \( a_0, a_1, a_2, \ldots \)
  - Program statements may be repeated, denote i-th instance of a as \( a^i \)
  - Event a occurs before event b: \( a \rightarrow b \)
  - An interval \( (a,b) \) is the duration between events \( a \rightarrow b \)
  - Interval \( I_j = (a,b) \) precedes interval \( I_{j'}=(c,d) \) if \( b \rightarrow c \)

- Critical regions
  - A critical region CR is an interval \( (a,b) \), where a is the first operation in the CR and b the last

- Mutual exclusion
  - Critical regions CR\(_a\) and CR\(_b\) are mutually exclusive if:
    - Either CR\(_a\) \( \rightarrow \) CR\(_b\) or CR\(_b\) \( \rightarrow \) CR\(_a\) for all valid executions!

- Assume atomic registers (for now)

Simple Two-Thread Locks

- A first simple spinlock

```c
volatile int flag=0;
void lock() {
    flag = 1;
}
void unlock() {
    flag = 0;
}
```

- Busy-wait to acquire lock (spinning)

- Why does this not guarantee mutual exclusion?
**Proof Intuition**
- Construct a sequentially consistent history that permits both processes to enter the CR

**Simple Two-Thread Locks**
- Another two-thread spin-lock: LockOne

```c
volatile int flag[2];
void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}
void unlock() {
    flag[tid] = false;
}
```

**Correctness Proof**
- In sequential consistency!
- Intuitions:
  - Situation: both threads are ready to enter
  - Show that situation that allows both to enter leads to a schedule violating sequential consistency
    
    Using transitivity of program and synchronization orders

**Simple Two-Thread Locks**
- A third attempt at two-thread locking: LockTwo

```c
volatile int victim;
void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}
void unlock() {}
```

**Correctness Proof**
- Intuition:
  - Victim is only written once per lock()
  - A can only enter after B wrote
  - B cannot enter in any sequentially consistent schedule

**Simple Two-Thread Locks**
- Another two-thread spin-lock: LockOne

```c
volatile int flag[2];
void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}
void unlock() {
    flag[tid] = false;
}
```

**Correctness Proof**
- Intuition:
  - Victim is only written once per lock()
  - A can only enter after B wrote
  - B cannot enter in any sequentially consistent schedule

**Simple Two-Thread Locks**
- Another two-thread spin-lock: LockOne

```c
volatile int flag[2];
void lock() {
    int j = 1 - tid;
    flag[tid] = true;
    while (flag[j]) {} // wait
}
void unlock() {
    flag[tid] = false;
}
```
Simple Two-Thread Locks

A third attempt at two-thread locking: LockTwo

```c
volatile int victim;
void lock() {
    victim = tid; // grant access
    while (victim == tid) {} // wait
}
void unlock() {
}
```

Does this guarantee mutual exclusion?
Does it work in practice?

Peterson’s Two-Thread Lock (1981)

Combines the first lock (request access) with the second lock (grant access)

```c
volatile int flag[2];
volatile int victim;
void lock() {
    int j = 1 - tid;
    flag[tid] = 1;  // I'm interested
    victim = tid;  // other goes first
    while (flag[j] && victim == tid) {} // wait
}
void unlock() {
    flag[tid] = 0;  // I'm not interested
}
```

Proof Correctness

Intuition:
- Victim is written once
- Pick thread that wrote victim last
- Show thread must have read flag==0
- Show that no sequentially consistent schedule permits that

Starvation Freedom

(recap) definition: Every thread that calls lock() eventually gets the lock.
- Implies deadlock-freedom!

Is Peterson’s lock starvation-free?

```c
volatile int flag[2];
volatile int victim;
void lock() {
    int j = 1 - tid;
    flag[tid] = 1;  // I'm interested
    victim = tid;  // other goes first
    while (flag[j] && victim == tid) {} // wait
}
void unlock() {
    flag[tid] = 0;  // I'm not interested
}
```

Proof Starvation Freedom

Intuition:
- Threads can only wait/starve in while()
- Until flag==0 or victim==other
- Other thread enters lock() \(\Rightarrow\) sets victim to other
- Will definitely "unstuck" first thread
- So other thread can only be stuck in lock()
  - Will wait for victim==other, victim cannot block both threads \(\Rightarrow\) one must leave!
Implement and run our little counter on x86

100000 iterations
- 1.6 \times 10^{-6}\% errors
- What is the problem?

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1;  // I'm interested
  victim = tid;  // other goes first
  while (flag[j] && victim == tid) {} // wait
}

void unlock() {
  flag[tid] = 0;  // I'm not interested
}
```

Correct Peterson Lock on x86

- Unoptimized (naïve sprinkling of mfences)
- Performance:
  - No mfence: 375ns
  - mfence in lock: 379ns
  - mfence in unlock: 404ns
  - Two mfence: 427ns (+14%)
Filter Lock - Correctness

- Lemma: For 0<j<n-1, there are at most n-j threads at level j!

  - Intuition:
    - Recursive proof (induction on j)
    - By contradiction, assume n-j+1 threads at level j-1 and j
    - Assume last thread to write victim
    - Any other thread writes level before victim
    - Last thread will stop at spin due to other thread’s write

  - j=n-1 is critical region

Filter Lock Starvation Freedom

- Intuition:
  - Inductive argument over j (levels)
  - Base-case: level n-1 has one thread (not stuck)
  - Level j: assume thread is stuck
    - Eventually, higher levels will drain (induction)
    - Last entering thread is victim, it will wait
    - Thus, only one thread can be stuck at each level
    - Victim can only have one value → older threads will advance!

Locking for N threads

- Simple generalization of Peterson’s lock, assume n levels l = 0…n-1
  - Is it starvation-free?

```
volatile int level[n] = {0,0,…,0}; // indicates highest level a thread tries to enter
volatile int victim[n]; // the victim thread, excluded from next level

void lock() {
  for (int i = 1; i < n; i++) { // attempt level
    level[tid] = i;
    victim[i] = tid;
    // spin while conflicts exist
    while ((∃k != tid) (level[k] >= i && victim[i] == tid)) {};
  }
}

void unlock() {
  level[tid] = 0;
}
```

Filter Lock

- What are the disadvantages of this lock?

  - Starvation freedom provides no guarantee on how long a thread waits or if it is “passed”!
  - To reason about fairness, we define two sections of each lock algorithm:
    - Doorway D (bounded # of steps)
    - Waiting W (unbounded # of steps)

    - FIFO locks:
      - If Tj finishes its doorway before Tk the CRj → CRk
      - Implies fairness

```
void lock() {
  int j = 1 - tid;
  flag[tid] = true; // I'm interested
  victim = tid;      // other goes first
  while (flag[j] && victim == tid) {};
}
```

Lock Fairness

- Starvation freedom provides no guarantee on how long a thread waits or if it is “passed”!
- To reason about fairness, we define two sections of each lock algorithm:
  - Doorway D (bounded # of steps)
  - Waiting W (unbounded # of steps)

  - FIFO locks:
    - If Tj finishes its doorway before Tk the CRj → CRk
    - Implies fairness

Lamport’s Bakery Algorithm (1974)

- Is a FIFO lock (and thus fair)
- Each thread takes a number in the doorway and threads enter in the order of their number!

```
volatile int flag[n] = {0,0,...,0};
volatile int label[n] = {0,0,...,0};

void lock() {
  flag[tid] = 1; // request
  label[tid] = max(label[0], ..., label[n-1]) + 1; // take ticket
  while ((∃k != tid)(flag[k] && label[k] < label[tid])) {};
  public void unlock() {
    flag[tid] = 0;
  }
```
Lamport’s Bakery Algorithm (1974)

Advantages:
- Elegant and correct solution
- Starvation free, even FIFO fairness

Not used in practice!
- Why?
- Needs to read/write N memory locations for synchronizing N threads
- Can we do better?
  Using only atomic registers/memory

A Lower Bound to Memory Complexity

Theorem 5.1 in [1]: "If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress (deadlock-freedom), then S must have at least as many variables as processes"

So we’re doomed! Optimal locks are available and they’re fundamentally non-scalable. Or not?


Hardware Support?

Hardware atomic operations:
- Test&Set
  Write const to memory while returning the old value
- Atomic swap
  Atomically exchange memory and register
- Fetch&Op
  Get value and apply operation to memory location
- Compare&Swap
  Compare two values and swap memory with register if equal
- Load-linked/Store-Conditional LL/SC
  Loads value from memory, allows operations, commits only if no other updates committed \(\rightarrow\) mini-TM
- Intel TSX (transactional synchronization extensions)
  Hardware-TM (roll your own atomic operations)

Relative Power of Synchronization

Design-Problem I: Multi-core Processor
- Which atomic operations are useful?

Design-Problem II: Complex Application
- What atomic should I use?

Concept of “consensus number” \(C\) if a primitive can be used to solve the “consensus problem” in a finite number of steps (even if threads stop)
- atomic registers have \(C=1\) (thus locks have \(C=1\!\))
- TAS, Swap, Fetch&Op have \(C=2\)
- CAS, LL/SC, TM have \(C=\infty\)

Test-and-Set Locks

Test-and-Set semantics
- Memoize old value
- Set fixed value TASval (true)
- Return old value

After execution:
- Post-condition is a fixed (constant) value!

bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = true;
    return old;
} // all atomic!

Test-and-Set Locks

Assume TASval indicates “locked”
Write something else to indicate “unlocked”

TAS until return value is \(!=\) TASval (1 in this example)

When will the lock be granted?
Does this work well in practice?

volatile int lck = 0;
void lock() {
    while (TestAndSet(&lck) == 1);
}
void unlock() {
    lck = 0;
}
Contention

- On x86, the XCHG instruction is used to implement TAS
  - x86 lock is implicit in xchg!

- Cacheline is read and written
  - Ends up in exclusive state, invalidates other copies
  - High load on memory subsystem
  - x86 lock is essentially a full memory barrier 🔄

Test-and-Test-and-Set (TATAS) Locks

- Spinning in TAS is not a good idea
- Spin on cache line in shared state
  - All threads at the same time, no cache coherency/memory traffic

Danger!
- Efficient but use with great care!
- Generalizations are very dangerous

```c
volatile int lck = 0;
void lock() {
  do {
    while (lck == 1);
  } while (TestAndSet(&lck) == 1);
}
void unlock() {
lck = 0;
}
```

Warning: Even Experts get it wrong!

- Example: Double-Checked Locking

```
Problem: Memory ordering leads to race-conditions!
```

Contention?

- Do TATAS locks still have contention?
- When lock is released, k threads fight for cache line ownership
  - One gets the lock, all get the CL exclusively (serially)!
  - What would be a good solution? (think "collision avoidance")

TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically
  - Locks granted in unpredictable order
  - Starvation possible but unlikely

```
volatile int lck = 0;
void lock() {
  do {
    while (TestAndSet(&lck) == 1) {
      wait(time);
    }
    time = min(time * 2, maxtime);
  } while (TestAndSet(&lck) == 1);
}
void unlock() {
  lck = 0;
}
```


```
TAS Lock with Exponential Backoff
- Exponential backoff eliminates contention statistically
  - Locks granted in unpredictable order
  - Starvation possible but unlikely
  - Maximum waiting time makes it less likely

volatile int lck = 0;
const int maxtime=1000;
void lock() {
  do {
    while (TestAndSet(&lck) == 1) {
      wait(time);
      time = min(time * 2, maxtime);
    }
  } while (TestAndSet(&lck) == 1);
}
void unlock() {
  lck = 0;
}
```

Comparison of TAS Locks

![Graph comparing TAS Locks]  

<table>
<thead>
<tr>
<th></th>
<th>TAS</th>
<th>TAS+Backoff</th>
<th>TAS-Backoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time per Update (ns)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

Number of Threads

---

Improvements?

- Are TAS locks perfect?
  - What are the two biggest issues?
  - Cache coherency traffic (contending on same location with expensive atomics)
  - Critical section underutilization (waiting for backoff times will delay entry to CR)
- What would be a fix for that?
  - How is this solved at airports and shops (often at least)?
- Queue locks -- Threads enqueue
  - Learn from predecessor if it's their turn
  - Each thread spins at a different location
  - FIFO fairness

---

Array Queue Lock

- Array to implement queue
  - Tail-pointer shows next free queue position
  - Each thread spins on own location
  - SU padding!
  - index[] array can be put in TLS
- So are we done now?
  - What’s wrong?
  - Synchronizing M objects requires Θ(NM) storage
  - What do we do now?

```c
volatile int array[n] = {1,0,...,0};
volatile int index[tid] = {0,0,...,0};
volatile int tail = 0;

void lock() {  
index[tid] = GetAndInc(tail) % n;  
while (!array[index[tid]]); // wait to receive lock
}

void unlock() {  
array[index[tid]] = 0; // I release my lock  
array[(index[tid] + 1) % n] = 1; // next one
}
```

---

CLH Lock (1993)

- List-based (same queue principle)
- Discovered twice by Craig, Landin, Hagersten 1993/94
- 2N+3M words
  - N threads, M locks
- Requires thread-local qnode pointer
  - Can be hidden!

```c
typedef struct qnode {
    struct qnode *prev;
    int succ_blocked;
    } qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
    qn->prev = FetchAndSet(lck, qn);
    while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
    qnode *pred = (*qn)->prev;
    (*qn)->prev = pred;
}
```

---

MCS Lock (1991)

- Make queue explicit
  - Acquire lock by appending to queue
  - Spin on own node until locked is reset
- Similar advantages as CLH but
  - Only 2N + M words
  - Spinning position is fixed!  
    Benefits cache-less NUMA
- What are the issues?
  - Releasing lock spins
  - More atomics!

```c
typedef struct qnode {
    struct qnode *next;
    int succ_blocked;
    } qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
    qn->next = NULL;
    qnode *pred = FetchAndSet(lck, qn);
    if(pred == NULL) {  
        qn->locked = 1;
        pred->next = qn;
        while(qn->locked);
    }
}

void unlock(qnode * lck, qnode *qn) {
    if(qn->next == NULL) { // if we’re the last waiter
        return;
    }
    while(qn->next == NULL); // wait for pred arrival
    qn->locked = 0; // free next waiter
    qn->next = NULL;
}
Lessons Learned!

- Key Lesson:
  - Reducing memory (coherency) traffic is most important!
  - Not always straight-forward (need to reason about CL states)

- MCS: 2006 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, fair, scalable → widely used, always compared against!

Time to Declare Victory?

- Down to memory complexity of 2N+M
  - Probably close to optimal

- Only local spinning
  - Several variants with low expected contention

- But: we assumed sequential consistency 😓
  - Reality causes trouble sometimes
  - Sprinkling memory fences may harm performance
  - Open research on minimally-synching algorithms!

  *Come and talk to me if you’re interested*