Spin Locks & Contention

Maurice Herlihy
Brown University
Focus so far: Correctness and Progress

Models
- Accurate (we never lied to you)
- But idealized

Protocols
- Elegant
- Important
- But naïve
New Focus: Performance

Models

- More complicated (not the same as complex!)
- Still focus on principles (not soon obsolete)

Protocols

- Elegant (in their fashion)
- Important (why else would we pay attention)
- And realistic (your mileage may vary)
Kinds of Architectures

- **SISD (Uniprocessor)**
  - Single instruction stream
  - Single data stream

- **SIMD (GPU)**
  - Single instruction
  - Multiple data

- **MIMD (Multiprocessors)**
  - Multiple instruction
  - Multiple data
Kinds of Architectures

SISD (Uniprocessor)
- Single instruction stream
- Single data stream

SIMD (GPU)
- Single instruction
- Multiple data

MIMD (Multiprocessors)
- Multiple instruction
- Multiple data

Our space
MIMD Architectures

Shared Bus

Distributed

Memory Contention

Communication Contention

Communication Latency
Today: Revisit Mutual Exclusion

Performance, not just correctness

Proper use of multiprocessor architectures

A collection of locking algorithms…
What Should you do if you can’t get a lock?

- Keep trying
- “spin” or “busy-wait”
  - Good if delays are short
- Yield the processor
  - Good if delays are long
  - Always good on uniprocessor

our focus today
Basic Spin-Lock

lock means sequential bottleneck

Bottleneck means no parallelism

Art of Multiprocessor Programming
Basic Spin-Lock

Lock suffers from contention

Honk!

spin lock  critical section  CS  resets lock upon exit

Honk!
Test-and-Set Lock

Boolean state

Lock: Test-and-set (TAS)

Swap *true* with current value

Returns prior value

Unlock: write *false*

TAS sometimes called “getAndSet”
public class AtomicBoolean {
    boolean value;

    public synchronized boolean getAndSet(boolean newValue) {
        boolean prior = value;
        value = newValue;
        return prior;
    }
}
Review: Test-and-Set

```java
public class AtomicBoolean {
    boolean value;

    public synchronized boolean getAndSet(boolean newValue) {
        boolean prior = value;
        value = newValue;
        return prior;
    }
}
```

Package

`java.util.concurrent.atomic`
Review: Test-and-Set

public class AtomicBoolean {
    boolean value;

    public synchronized boolean getAndSet(boolean newValue) {
        boolean prior = value;
        value = newValue;
        return prior;
    }
}

Atomically swap old and new values
Review: Test-and-Set

AtomicBoolean lock
    = new AtomicBoolean(false)
...
boolean prior = lock.getAndSet(true)
Review: Test-and-Set

AtomicBoolean lock
    = new AtomicBoolean(false)

...boolean prior = lock.getAndSet(true)

Swapping in true is called “test-and-set” or TAS
(usually a HW machine instruction)
Test-and-Set Locks

- **Lock is free:** value is *false*
- **Lock is taken:** value is *true*
- **Spin:** repeatedly call TAS
  - When result is *false*, stop
  - While result is *true*, try again
- **Release lock by writing** *false*
Test-and-set Lock

class TASlock {
    AtomicBoolean state =
        new AtomicBoolean(false);

    void lock() {
        while (state.getAndSet(true)) {}  
    }

    void unlock() {
        state.set(false);
    }
}
Test-and-set Lock

class TASlock {
    AtomicBoolean state = new AtomicBoolean(false);

    void lock() {
        while (state.getAndSet(true)) {
        }
    }

    void unlock() {
        state.set(false);
    }
}

Lock state is AtomicBoolean
Test-and-set Lock

class TASlock {
    AtomicBoolean state =
        new AtomicBoolean(false);

    void lock() {
        while (state.getAndSet(true)) {} // Keep trying until lock acquired
    }

    void unlock() {
        state.set(false);
    }
}
class TASlock {
    AtomicBoolean state = new AtomicBoolean(false);

    void lock() {
        while (state.getAndSet(true)) {}
    }

    void unlock() {
        state.set(false);
    }
}

Test-and-set Lock

Release lock by resetting state to false
Space Complexity

TAS spin-lock has small “footprint”

$n$ thread spin-lock uses $O(1)$ space

As opposed to $O(n)$ Peterson/Bakery

What about the $\Omega(n)$ RW lower bound?

TAS is RMW, not RW …
Performance

Experiment

$n$ threads

Increment shared counter 1 million times

How long *should* it take?

How long *does* it take?
Graph

Sequential bottleneck means \textit{no speedup}
Mystery #1

- TAS lock
- Ideal
- Wait, what?

Graph:
- Vertical axis: time
- Horizontal axis: threads

The graph illustrates the relationship between time and threads, with TAS lock and Ideal points marked.
Test-and-Test-and-Set Locks

Spin until lock “looks” free

Continue while read returns *true* (lock taken)

As soon as read returns *false* (lock free) …

Attack!

Call TAS to acquire lock

If TAS fails, back to spinning
Test-and-test-and-set Lock

class TTASLock {
    AtomicBoolean state =
        new AtomicBoolean(false);

    void lock() {
        while (true) {
            while (state.get()) {}
            if (!state.getAndSet(true))
                return;
        }
    }
}
Test-and-test-and-set Lock

class TTASLock {
    AtomicBoolean state = new AtomicBoolean(false);

    void lock() {
        while (true) {
            while (state.get()) {} // Wait until lock looks free
            if (!state.getAndSet(true)) return;
        }
    }
}
Test-and-test-and-set Lock

class TTASlock {
    AtomicBoolean state =
        new AtomicBoolean(false);

    void lock() {
        while (true) {
            while (state.get()) {}
            if (!state.getAndSet(true))
                return;
        }
    }
}  

Then try to acquire it
Mystery #2

- TAS lock
- TTAS lock
- Ideal

Graph with time on the y-axis and threads on the x-axis.
Mystery

Both

TAS and TTAS

Do the same thing (logically)

Except that

TTAS performs much better than TAS

Neither approaches ideal
Opinion

Our memory abstraction is broken

TAS & TTAS methods

Are provably the same (in our model)

Except they aren’t (in field tests)

Need a better model …
Bus-Based Architectures

- cache
- cache
- cache

Bus

memory
Bus-Based Architectures

Random access memory (10s of cycles)
Bus-Based Architectures

Shared Bus
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”
Bus-Based Architectures

Per-Processor Caches
- Small
- Fast: 1 or 2 cycles
- Address & state information

Art of Multiprocessor Programming
Granularity

Caches operate at a larger granularity than a word

*Cache line*: fixed-size block containing the address (today 64 or 128 bytes)
Locality

If you use an address now, you will probably use it again soon

Fetch from cache, not memory

If you use an address now, you will probably use a nearby address soon

In the same cache line
L1 and L2 Caches

Memory (100s of cycles)

L1 (1 or 2 cycles)

L2 (10s of cycles)
L1 and L2 Caches

Small & fast 1 or 2 cycles
L1 and L2 Caches

Larger and slower
10s of cycles
\(~128\) byte line
Jargon Watch

Cache *hit*

“I found what I wanted in my cache”

Good Thing™

Cache *miss*

“I had to shlep all the way to memory …

Bad Thing™
Cave Canem

This model is still a simplification

But not in any essential way

Illustrates basic principles

There are many refinements …
When a Cache Becomes Full…

Need to make room for new entry

By evicting an existing entry

Need a replacement policy

Usually some variation on “least recently used”
Fully Associative Cache

Any line can be anywhere in the cache

*Advantage*: can replace any line

*Disadvantage*: hard to find lines
Direct Mapped Cache

Every address has exactly 1 slot

*Advantage*: easy to find a line

*Disadvantage*: must replace fixed line
K-way Set Associative Cache

Each slot holds k lines

*Advantage:* pretty easy to find a line

*Advantage:* some choice in replacing line
Cache Coherence

A and B both cache address x

A writes to x

A updates A’s own cache

How does B find out?

Many cache coherence protocols in literature
MESI

Modified
Data modified, must write back to memory

Exclusive
Not modified, I have only copy

Shared
Not modified, may be cached elsewhere

Invalid
Cache contents not meaningful
Processor Issues Load Request

load x

cache

cache

cache

Bus

memory data
Memory Responds

cache

Bus

cache

cache

memory

data

Got it!
Processor Issues Load Request

Load x

Bus

data

cache

cache

memory

data

Art of Multiprocessor Programming
Other Processor Responds

Got it
Modify Cached Data

- Bus
- Data
- Cache
- Memory
- Data

Art of Multiprocessor Programming
Write-Through Cache

Write x!

memory data

data

cache

S

data

S

data

Bus
Write-Through Caches

**Good**
- Memory, caches always agree
- More read hits, maybe…

**Bad**
- Bus traffic on all writes
- Most writes to unshared data
- For example, loop indexes
Write-Back Caches

Accumulate changes in cache

Write back when line evicted

Need the cache for something else

Another processor wants it
Invalidate

Invalidate x

Art of Multiprocessor Programming

58
This cache acquires write permission
Invalidate

Other caches lose read permission

This cache acquires write permission
Invalidate

Memory provides data only if not present in any cache, so no need to change it now (expensive)
Optimize What?

- Bus bandwidth used by spinning threads?
- Release/Acquire latency?
- Acquire latency for idle lock?
Simple TASLock

TAS invalidates cache lines

Spinning threads ...

Miss in cache

Go to bus

delayed behind spinners

When thread tries to release lock

delayed behind spinners
Test-and-test-and-set

Wait until lock “looks” free
Spin on local cache
No bus use while lock busy
Problem: when lock is released
Invalidation storm …
Local Spinning while Lock is Busy

Art of Multiprocessor Programming

65
On Release

invalid invalid free

memory free

Bus
On Release

Everyone misses, rereads

miss miss free

memory free

Bus
On Release

Everyone tries TAS

TAS(...)

TAS(...)

free

Bus

memory free
Problems

Everyone misses

Reads satisfied sequentially

Everyone does TAS

Invalidates others’ caches

Eventually quiesces after lock acquired

How long does this take?
Measuring Quiescence Time

- Acquire lock
- Pause without using bus
- Use bus heavily

If pause > quiescence time, critical section duration independent of number of threads.

If pause < quiescence time, critical section duration slower with more threads.
Quiescence Time

Increases linearly with the number of processors for bus architecture.
Mystery Explained

- TAS lock
- TTAS lock
- Ideal

Better than TAS but still way short of ideal
Solution: Introduce Delay

If the lock looks free …

But I fail to get it

There must be contention …

Better to back off than to collide again
Dynamic Example: Exponential Backoff

If I fail to get lock
Wait random duration before retry
Each subsequent failure doubles expected wait
Exponential Backoff Lock

class Backoff implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {}  
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}
public class Backoff implements lock {
        public void lock() {
            int delay = MIN_DELAY;
            while (true) {
                while (state.get()) {}
                if (!lock.getAndSet(true))
                    return;
                sleep(random() % delay);
                if (delay < MAX_DELAY)
                    delay = 2 * delay;
            }
        }
    }
public class Backoff implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {
            }
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}

Wait until lock looks free
public class Backoff implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {} // Check if state already locked
            if (!lock.getAndSet(true)) return; // If we acquire lock, return
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}

Exponential Backoff Lock

If we acquire lock, return
Exponential Backoff Lock

```java
public class Backoff implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {}
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}
```

Back off for random duration
public class Backoff implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {}
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}

Exponential Backoff Lock

Double max delay, within reason
Spin-Waiting Overhead

![Graph showing the relationship between time and threads for TTAS Lock and Backoff Lock]

- TTAS Lock
- Backoff Lock

Art of Multiprocessor Programming
Backoff: Other Issues

**Good**
- Easy to implement
- Beats TTAS lock

**Bad**
- Must choose parameters carefully
- Not portable across platforms
Actual Data on 40-Core Machine

Lock Scalability - Latency

Latency

Number of Threads

T&S
T&T&S
Backoff
Idea

Avoid useless invalidations

By keeping a *queue* of threads

Each thread

Notifies next in line

Without bothering the others
Anderson Queue Lock

next

idle

flags

FREE BUSY BUSY BUSY BUSY BUSY BUSY BUSY BUSY
Anderson Queue Lock

flags

next

idle

FREE BUSY BUSY BUSY BUSY BUSY BUSY BUSY BUSY
Anderson Queue Lock

flags

next

acquiring

getAndIncrement

FREE  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY
Anderson Queue Lock

flags

next

acquiring

cgetAndIncrement

FREE BUSY BUSY BUSY BUSY BUSY BUSY BUSY
Anderson Queue Lock

next

acquired

flags

FREE BUSY BUSY BUSY BUSY BUSY BUSY BUSY

Mine!
Anderson Queue Lock

next

flags

acquired

acquiring

FREE BUSY BUSY BUSY BUSY BUSY BUSY BUSY
Anderson Queue Lock

flags

next

acquired

acquiring

getAndIncrement

FREE | BUSY | BUSY | BUSY | BUSY | BUSY | BUSY | BUSY | BUSY

Art of Multiprocessor Programming
Anderson Queue Lock

next

flags

acquired

acquiring

getAndIncrement

FREE BUSY BUSY BUSY BUSY BUSY BUSY BUSY BUSY
Anderson Queue Lock

next

acquired

acquiring

flags

FREE  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY
Anderson Queue Lock

next

released

acquired

flags

FREE  FREE  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY
Anderson Queue Lock

flags

next

released

acquired

FREE  FREE  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY  BUSY

Yow!
Anderson Queue Lock

class ALock implements Lock {
    boolean[] flags={true, false, ..., false};
    AtomicInteger next
        = new AtomicInteger(0);
    ThreadLocal<Integer> mySlot;
}
Anderson Queue Lock

class ALock implements Lock {
    boolean[] flags={true, false, ..., false};
    AtomicInteger next = new AtomicInteger(0);
    ThreadLocal<Integer> mySlot;

    One flag per thread: false is BUSY, true is FREE
Anderson Queue Lock

class ALock implements Lock {
    boolean[] flags={true,false,...,false};
    AtomicInteger next = new AtomicInteger(0);
    ThreadLocal<Integer> mySlot;

    Next flag to use
class ALock implements Lock {
    boolean[] flags={true, false,..., false};
    AtomicInteger next
    = new AtomicInteger(0);
    ThreadLocal<Integer> mySlot;
}

Thread-local variable
Anderson Queue Lock

```java
public lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {};
    flags[mySlot % n] = false; // BUSY
}

public unlock() {
    flags[(mySlot+1) % n] = true; // FREE
}
```
Anderson Queue Lock

```java
public lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {};
    flags[mySlot % n] = false; // BUSY
}

public unlock() {
    flags[(mySlot+1) % n] = true; // FREE
}
```

Take next slot
Anderson Queue Lock

public lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {};
    flags[mySlot % n] = false;  // BUSY
}

public unlock() {
    flags[(mySlot+1) % n] = true;  // FREE
}

Spin until told to go
public lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {};
    flags[mySlot % n] = false; // BUSY
}

public unlock() {
    flags[(mySlot+1) % n] = true; // FREE
}
Anderson Queue Lock

```java
public lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {
        flags[mySlot % n] = false; // BUSY
    }
}

public unlock() {
    flags[(mySlot+1) % n] = true; // FREE
}
```

Tell next thread to go
Local Spinning

Flags

FREE BUSY BUSY BUSY BUSY BUSY BUSY BUSY BUSY

Unfortunately, many bits share cache line

next

released acquired
False Sharing

flags

next

released

acquired

spinning thread’s cache invalidated by unrelated store

FREE BUSY BUSY BUSY BUSY

Line 1

Line 2
The Solution: Padding

flags

next

released

acquired

on this line

FREE

BUSY

Line 1

Line 2

/ / / /

/ / / /

/ / / /
Performance

- TTAS
  - Shorter handover than backoff
  - Curve is practically flat
  - Scalable performance
Good Aspects

- First truly scalable lock
- Simple, easy to implement
- Back to FCFS order (like Bakery)
Bad Aspects

- Takes too much space
- One bit per thread ➔ one cache line per thread
- What if unknown number of threads?
- What if small number of actual contenders?
CLH Lock

FCFS order

Small, constant-size overhead per thread
Initially

idle

FREE
Initially

idle

tail

FREE

Queue tail
Initially

idle

Lock is free

FREE
Initially
Purple Wants the Lock

acquiring

tail  -->  FREE
Purple Wants the Lock

acquiring

tail

FREE

BUSY
Purple Wants the Lock

acquiring

Swap

tail

FREE

BUSY
Purple Has the Lock

acquired

tail

FREE

BUSY

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

tail

FREE

BUSY

BUSY

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

Swap

tail

FREE

BUSY

BUSY

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

tail

FREE

BUSY

BUSY
Red Wants the Lock

acquired

acquiring

tail

FREE

BUSY

BUSY

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

Implicit Linked list

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

tail

FREE

BUSY

BUSY

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

Actually, it spins on locally cached copy

FREE

BUSY

BUSY

tail

Art of Multiprocessor Programming
Purple Releases

release

acquiring

Bingo!

FREE

FREE

FREE

BUSY

tail

The Art of Multiprocessor Programming

127
Purple Releases

released

acquired

tail

BUSY

Art of Multiprocessor Programming
Space Usage

Let

\[ L = \text{number of locks} \]

\[ N = \text{number of threads} \]

\[ \text{ALock: } O(LN) \]

\[ \text{CLH lock: } O(L+N) \]
CLH Queue Lock

class QNode {
    AtomicBoolean locked =
        new AtomicBoolean(true);
}
CLH Queue Lock

class QNode {
    AtomicBoolean locked = new AtomicBoolean(true);
}

Not released yet
CLH Queue Lock

class CLHLock implements Lock {
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode
        = new QNode();
    public void lock() {
        QNode pred
            = tail.getAndSet(myNode);
        while (pred.locked) {} 
    }
}
class CLHLock implements Lock {
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode = new QNode();
    public void lock() {
        QNode pred = tail.getAndSet(myNode);
        while (pred.locked) {}
    }
}
class CLHLock implements Lock {
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode = new QNode();
    public void lock() {
        QNode pred = tail.getAndSet(myNode);
        while (pred.locked) {
            }
        }
    }
Thread-local QNode
class CLHLock implements Lock {
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode = new QNode();
    public void lock() {
        QNode pred = tail.getAndSet(myNode);
        while (pred.locked) {}  
    }
}
class CLHLock implements Lock {
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode
        = new QNode();
    public void lock() {
        QNode pred
            = tail.getAndSet(myNode);
        while (pred.locked) {}
    }
}
CLH Queue Lock

Class CLHLock implements Lock {
    ...
    public void unlock() {
        myNode.locked.set(false); // FREE
        myNode = pred;
    }
}
Class CLHLock implements Lock {
  ...
  public void unlock() {
    myNode.locked.set(false); // FREE
    myNode = pred;
  }
}
CLH Queue Lock

Class CLHLock implements Lock {
    ...
    public void unlock() {
        myNode.locked.set(false); // FREE
        myNode = pred;
    }
}

Recycle predecessor’s node
CLH Queue Lock

Class CLHLock implements Lock {

    ...  
    public void unlock() {
        myNode.locked.set(false); // FREE
        myNode = pred;
    }
}

(we don’t actually reuse myNode. Code in book shows how it’s done.)
CLH Lock

Good

- Lock release affects predecessor only
- Small, constant-sized space

Bad

- CLH lock: Doesn’t work for uncached NUMA architectures
NUMA and cc-NUMA Architectures

Acronym: Non-Uniform Memory Architecture

ccNUMA = cache coherent NUMA

Illusion: Flat shared memory

Truth: No caches (sometimes)

Some memory regions faster than others
NUMA Machines

Spinning on local memory is fast
NUMA Machines

Spinning on remote memory is slow
CLH Lock

Each thread spins on predecessor’s memory

Could be far away …
MCS Lock

- FCFS order
- Spin on local memory only
- Small, Constant-size overhead
Initially

idle

tail

FREE
Acquiring

(allocate QNode)

FREE

BUSY

tail

acquiring
Acquiring

acquired

swap

tail

FREE

BUSY
Acquiring

acquired

tail

FREE

BUSY
Acquired

acquired

tail

FREE

BUSY

Art of Multiprocessor Programming
Acquiring

acquired

acquiring

FREE

BUSY

tail

swap
Acquiring

acquired

tail

acquiring

FREE

BUSY
Acquiring

acquired

acquiring

tail

FREE

BUSY

Art of Multiprocessor Programming
Acquiring

acquired

tail

acquiring

FREE

BUSY

Art of Multiprocessor Programming
Acquiring

acquired

acquiring

tail

FREE
MCS Queue Lock

class QNode {
    volatile boolean locked = false;
    volatile QNode next = null;
}

MCS Queue Lock

class MCSLock implements Lock {
    AtomicReference tail;
    public void lock() {
        QNode qnode = new QNode();
        QNode pred = tail.getAndSet(qnode);
        if (pred != null) {
            qnode.locked = true;
            pred.next = qnode;
            while (qnode.locked) {}
        }
    }
}
MCS Queue Lock

class MCSLock implements Lock {
    AtomicReference tail;
    public void lock() {
        QNode qnode = new QNode();
        QNode pred = tail.getAndSet(qnode);
        if (pred != null) {
            qnode.locked = true;
            pred.next = qnode;
            while (qnode.locked)
        }
    }
}
MCS Queue Lock

class MCSLock implements Lock {
    AtomicReference tail;
    public void lock() {
        QNode qnode = new QNode();
        QNode pred = tail.getAndSet(qnode);
        if (pred != null) {
            qnode.locked = true;
            pred.next = qnode;
            while (qnode.locked) {}
        }
    }
}

add my Node to the tail of queue
class MCSLock implements Lock {
    AtomicReference tail;
    public void lock() {
        QNode qnode = new QNode();
        QNode pred = tail.getAndSet(qnode);
        if (pred != null) {
            qnode.locked = true;
            pred.next = qnode;
            while (qnode.locked) {}
        }
    }
}

Fix if queue was non-empty
class MCSLock implements Lock {
    AtomicReference tail;
    public void lock() {
        QNode qnode = new QNode();
        QNode pred = tail.getAndSet(qnode);
        if (pred != null) {
            qnode.locked = true;
            pred.next = qnode;
            while (qnode.locked) {}
        }
    }
}

Wait until unlocked
I don’t see a successor. But by looking at the queue, I see another thread is active.

Cannot release until that thread identifies its node.
Purple Release

releasing

prepare to spin

FREE

BUSY
Purple Release

spinning
Purple Release

releasing

Acquired lock

FREE

FREE

FREE
class MCSLock implements Lock {
    AtomicReference tail;

    public void unlock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null)) {
                return;
            }
        }
        while (qnode.next == null) {
        }
        qnode.next.locked = false;
    }
}
class MCSLock implements Lock {
    AtomicReference tail;
    public void unlock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null)
                return;
            while (qnode.next == null) {} }
        }
        qnode.next.locked = false;
    }
}
MCS Queue Lock

class MCSLock implements Lock {
    AtomicReference tail;
    public void unlock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null)
                return;
            while (qnode.next == null) {}}
        qnode.next.locked = false;
    }
}

If really no successor, return
class MCSLock implements Lock {

    AtomicReference tail;

    public void unlock()
    {

        if (qnode.next == null) {
            if (tail.CAS(qnode, null)
                return;

            while (qnode.next == null) {}}

        qnode.next.locked = false;
    }
}
class MCSLock implements Lock {
    AtomicReference qnode = null;

    public void unlock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null)) {
                return;
            }
        }
        while (qnode.next == null) {} // Here is the pass lock to successor
        qnode.next.locked = false;
    }
}
Abortable Locks

What if you want to give up waiting for a lock?

For example

Timeout

Database transaction aborted by user
Back-off Lock

Aborting is trivial
Just return from lock() call
Extra benefits:
No cleaning up
Wait-free
Immediate return
Queue Locks

- Can’t just quit
- Thread in line behind will starve
- Need a graceful off-ramp
Queue Locks

spinning

spinning

spinning

BUSY

BUSY

BUSY
Queue Locks

spinning

spinning

spinning

FREE

BUSY

BUSY
Queue Locks

locked

FREE

spinning

BUSY
Queue Locks
Queue Locks

spinning  spinning  spinning

BUSY  BUSY  BUSY
Queue Locks

spinning

BUSY

spinning

BUSY

BUSY

BUSY
Queue Locks

locked

spinning

FREE
BUSY
BUSY
Queue Locks

FREE

BUSY

spinning
Queue Locks

pwned

FREE

BUSY
Abortable CLH Lock

When a thread gives up

Removing node in a wait-free way is hard

Idea:

Let successor deal with it.
Initially

idle

tail

Pointer to predecessor (or null)
Initially

idle

Distinguished available node means lock is free

tail

A
Acquiring
Acquiring

Null predecessor means lock not released or aborted
Acquiring

acquiring

Swap

A
Acquiring
Acquired

Reference to AVAILABLE means lock is free.
Normal Case

Locked

Spinning

Spinning

Null means lock is not free & request not aborted
One Thread Aborts

locked  timed out  spinning
Successor Notices

locked

timed out

spinning

Non-Null means predecessor aborted
Recycle Predecessor's Node

locked

spinning

Art of Multiprocessor Programming 197
Spin on Earlier Node
Spin on Earlier Node

released

spinning

The lock is now mine
public class TOLock implements Lock {
    static QNode AVAILABLE = new QNode();
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode;
}
public class TOLock implements Lock {
    static QNode AVAILABLE = new QNode();
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode;
}

AVAILABLE node means lock is free
public class TOLock implements Lock {
    static QNode AVAILABLE = new QNode();
    AtomicReference&lt;QNode&gt; tail;
    ThreadLocal&lt;QNode&gt; myNode;

    Tail of the queue
public class TOLock implements Lock {
    static QNode AVAILABLE = new QNode();
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode;

    Remember my node ...
public boolean lock(long timeout) {
    QNode qnode = new QNode();
    myNode.set(qnode);
    qnode.prev = null;
    QNode myPred = tail.getAndSet(qnode);
    if (myPred== null || myPred.prev == AVAILABLE) {
        return true;
    }
}

...
public boolean lock(long timeout) {
    QNode qnode = new QNode();
    myNode.set(qnode);
    qnode.prev = null;
    QNode myPred = tail.getAndSet(qnode);
    if (myPred==null || myPred.prev == AVAILABLE) {
        return true;
    }
    ...
}

Create & initialize node
public boolean lock(long timeout) {
    QNode qnode = new QNode();
    myNode.set(qnode);
    qnode.prev = null;
    QNode myPred = tail.getAndSet(qnode);
    if (myPred == null || myPred.prev == AVAILABLE) {
        return true;
    }
}
...

Swap with tail
public boolean lock(long timeout) {
    QNode qnode = new QNode();
    myNode.set(qnode);
    qnode.prev = null;
    QNode myPred = tail.getAndSet(qnode);
    if (myPred== null || myPred.prev == AVAILABLE) {
        return true;
    }
    ...

    If predecessor absent or released, we are done
Time-out Lock

... long start = now();
while (now()- start < timeout) {
    QNode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }
}
...

Art of Multiprocessor Programming 208
long start = now();
while (now() - start < timeout) {
    QNode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    }
    else if (predPred != null) {
        myPred = predPred;
    }
}
long start = now();
while (now() - start < timeout) {
    QNode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }
}

Spin on predecessor’s prev field
Time-out Lock

... long start = now();
while (now() - start < timeout) {
    QNode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }
}
...

Predecessor released lock
long start = now();
while (now() - start < timeout) {
    QNode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }
}
Time-out Lock

... if (!tail.compareAndSet(qnode, myPred))
    qnode.prev = myPred;
    return false;
  }
}

What do I do when I time out?
if (!tail.compareAndSet(qnode, myPred))
    qnode.prev = myPred;
return false;

Do I have a successor?
If CAS fails, I do.
Tell it about myPred
If CAS succeeds: no successor, simply return false
public void unlock() {
    QNode qnode = myNode.get();
    if (!tail.compareAndSet(qnode, null))
        qnode.prev = AVAILABLE;
}
public void unlock() {
    QNode qnode = myNode.get();
    if (!tail.compareAndSet(qnode, null))
        qnode.prev = AVAILABLE;
}

If CAS failed: successor exists, notify that lock is free
Timing-out Lock

```java
public void unlock() {
    ONode qnode = myNode.get();
    if (!tail.compareAndSet(qnode, null))
        qnode.prev = AVAILABLE;
}
```

CAS successful: set tail to null, no clean up since no successor waiting
Fairness and NUMA Locks

MCS lock mechanics are aware of NUMA

Lock Fairness is FCFS

Is this a good fit with NUMA and Cache-coherent NUMA machines?
Lock Data Access in NUMA Machine

Node 1

Node 2

CS

MCS lock

various memory locations
“Who’s the Unfairest of Them All?”

locality crucial to NUMA performance

Big gains if threads from same
node/cluster obtain lock consecutively

Unfairness pays
Hierarchical Backoff Lock (HBO)

Back off less for thread from same node

Unfairness is key to performance
Hierarchical Backoff Lock (HBO)

Advantages:
- Simple, improves locality

Disadvantages:
- Requires platform specific tuning
- Unstable
- Unfair
- Continuous invalidations on shared global lock word
Hierarchical CLH Lock (HCLH)

Each thread spins on cached copy of predecessor’s node.

Thread at local head splices local queue into global queue.

Global Tail

CAS()

Local Tail

CAS()

Local CLH queue

Global CLH queue

Thread at local head splices local queue into global queue.

CAS()
Hierarchical CLH Lock (HCLH)

Threads access 4 cache lines in CS
Hierarchical CLH Lock (HCLH)

Advantages:
- Improved locality
- Local spinning
- Fair

Disadvantages:
- Complex code implies long common path
- Splicing into both local and global requires CAS
- Hard to get long local sequences
“Nothing yet. ... How about you, Newton?”
Lock Cohort ing

General technique for converting almost any lock into a NUMA lock

Allows combining different lock types

But these locks must have certain properties …
Lock Cohorting

Non-empty cohort
empty cohort

On release: if non-empty cohort of waiting threads, release *only local lock*; leave mark

Thread that acquired local lock can now acquire global lock...

On release: since cohort is empty *must release global lock* to avoid deadlock

Acquire local lock and proceed to critical section
Thread Obliviousness

A lock is *thread-oblivious* if

**After being acquired by one thread,**

**Can be released by another**
Cohort Detection

A lock $x$ provides *cohort detection* if it can tell whether a thread is trying to acquire it.
Two Levels of Locking

Global lock: thread oblivious
Thread acquiring the lock can be different than one releasing it

Local lock: cohort detection
Thread releasing can detect if some thread is waiting to acquire it
Lock Cohorting: C

- Local MCS lock tail
- Global backoff lock

Two new states: acquire local and acquire global. Do we own global lock?

In MCS Lock, cohort detection by checking successor pointer

Bound number of consecutive acquires to control unfairness

BO Lock is thread oblivious by definition
How to add cohort detection property to BO lock?

As noted, BO Lock is thread oblivious.
Lock Cohorting: BO-BO Lock

Add `successorExists` field before attempting to acquire local lock.

`successorExists` reset on lock release.

Release might overwrite another successor’s write … but we don’t care…why?

Add `successorExists` field before attempting to acquire local lock.

Release might overwrite another successor’s write … but we don’t care…why?
C-B-BO is a Time-Out NUMA Lock

Global backoff lock

BO locks trivially abortable

Aborting thread resets successorExists field before leaving local lock. Spinning threads set it to true.

If releasing thread finds successorExists false, it releases global lock.
Lock Cohortizing

Advantages:
- Great locality
- Low contention on shared lock
- Practically no tuning
- Has whatever properties you want: can be more or less fair, abortable...
  just choose the appropriate type of locks...

Disadvantages:
- Must tune fairness parameters
Lock Cohorting

![Graph showing throughput vs. number of threads for different lock cohorting methods: C-BO-MCS, C-BO-BO, HCLH, HBO.](image)
Time-Out (Abortable) Lock Cohorting

- A-BO-CLH (time-out lock + BO)
- A-BO-BO

Abortable CLH (our time-out lock) and HBO

Graph showing throughput in CR-nCRs per sec against number of threads.
One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS, ToLock…
- Each better than others in some way
- There is no one solution

Best pick depends on:
- the application
- the hardware architecture
- which properties are important
Creative Commons Attribution-ShareAlike 2.5 License

- **You are free:**
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
- **Under the following conditions:**
  - Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - [http://creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/).
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.
A lock x provides *cohort detection* if

It can tell whether any thread is trying to acquire it

Can be released by another

Database transaction aborted by user
TTAS+Backoff, CLH, MCS, ToLock…

Each better than others in some way

There is no one solution

Best pick depends on:

the application

the hardware architecture

which properties are important

Disadvantages:

Must tune fairness parameters