Spin Locks and Contention

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Focus so far: Correctness and Progress

• Models
  – Accurate (we never lied to you)
  – But idealized (so we forgot to mention a few things)

• 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 (Vector)**
  - Single instruction
  - Multiple data

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

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

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

- **MIMD (Multiprocessors)**
  - Multiple instruction
  - Multiple data.
MIMD Architectures

- 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

• Give up the processor
  – Good if delays are long
  – Always good on uniprocessor
What Should you do if you can’t get a lock?

• Keep trying
  – “spin” or “busy-wait”
  – Good if delays are short

• Give up the processor
  – Good if delays are long
  – Always good on uniprocessor

our focus
Basic Spin-Lock

spin lock  critical section  Resets lock upon exit

- Lock is acquired by spinning until the lock is released.
- Critical section executes while the lock is held.
- Lock is reset upon exiting the critical section.
Basic Spin-Lock

lock introduces sequential bottleneck
Basic Spin-Lock

...lock suffers from contention
Basic Spin-Lock

...lock suffers from contention

Notice: these are distinct phenomena
Basic Spin-Lock

...lock suffers from contention

Seq Bottleneck ➞ no parallelism
Basic Spin-Lock

...lock suffers from contention

Contention $\rightarrow$ ????
Review: Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap true with current value
  - Return value tells if prior value was true or false
- Can reset just by writing false
- TAS aka “getAndSet”
Review: Test-and-Set

```java
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

```java
public class AtomicBoolean {
    boolean value;

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

Swap old and new values
Review: Test-and-Set

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

```java
AtomicBoolean lock = new AtomicBoolean(false)

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

Swapping in true is called “test-and-set” or TAS
Test-and-Set Locks

• Locking
  – Lock is free: value is false
  – Lock is taken: value is true
• Acquire lock by calling TAS
  – If result is false, you win
  – If result is true, you lose
• 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(true);
    }
}

Lock state is AtomicBoolean
Test-and-set Lock

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

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

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

Keep trying until lock acquired
Test-and-set Lock

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

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

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

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
• How did we overcome the $\Omega(n)$ lower bound?
• We used a RMW operation…
Performance

• Experiment
  – $n$ threads
  – Increment shared counter 1 million times

• How long should it take?

• How long does it take?
Graph

no speedup
because of
sequential bottleneck

ideal

threads

time
Mystery #1

What is going on?

TAS lock

Ideal

threads

time
Test-and-Test-and-and-Set Locks

• Lurking stage
  – Wait until lock “looks” free
  – Spin while read returns true (lock taken)

• Pouncing state
  – As soon as lock “looks” available
  – Read returns false (lock free)
  – Call TAS to acquire lock
  – If TAS loses, back to lurking
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()) {}

            if (!state.getAndSet(true))
                return;
        }
    }
}

Wait until lock looks free
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;
        }
    }
}
Mystery #2

- TAS lock
- TTAS lock
- Ideal
Mystery

• Both
  – TAS and TTAS
  – Do the same thing (in our model)

• 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 more detailed model …
Bus-Based Architectures

memory

Bus

cache

cache

cache
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
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
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™
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 for that data”
  – Bad Thing™
Cave Canem

- This model is still a simplification
  - But not in any essential way
  - Illustrates basic principles
- Will discuss complexities later
When a Cache Becomes Full…

• Need to make room for new entry
• By evicting an existing entry
• Need a replacement policy
  – Usually some kind of least recently used heuristic
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
Multicore Set Associativity

- \( k \) is 8 or even 16 and growing…
  - Why? Because cores share sets
  - Threads cut effective size if accessing different data
Cache Coherence

• A and B both cache address x
• A writes to x
  – Updates cache
• How does B find out?
• Many cache coherence protocols in literature
MESI

• Modified
  – Have modified cached data, must write back to memory
MESI

• Modified
  – Have modified cached data, must write back to memory

• Exclusive
  – Not modified, I have only copy
MESI

• Modified
  – Have modified cached data, must write back to memory

• Exclusive
  – Not modified, I have only copy

• Shared
  – Not modified, may be cached elsewhere
MESI

- **Modified**
  - Have modified cached data, 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

memory

cache

cache

cache

Bus
Memory Responds

got it!

cache cache cache

E

Bus

memory

data
Processor Issues Load Request

Load x

E

data

cache

cache

Bus

memory

data
Other Processor Responds

Got it
Modify Cached Data
Write-Through Cache

Write x!

memory data

Bus

data

cache

S

data

S

data

Art of Multiprocessor Programming 65
Write-Through Caches

• Immediately broadcast changes
• 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-Through Caches

• Immediately broadcast changes
• Good
  – Memory, caches always agree
  – More read hits, maybe
• Bad
  – Bus traffic on all writes
  – Most writes to unshared data
  – For example, loop indexes …

“show stoppers”
Write-Back Caches

- Accumulate changes in cache
- Write back when line evicted
  - Need the cache for something else
  - Another processor wants it
Invalidate

Bus

Invalidate x

I cache M data cache

memory data
Invalidate

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)
Mutual Exclusion

- What do we want to optimize?
  - Bus bandwidth used by spinning threads
  - Release/Acquire latency
  - Acquire latency for idle lock
Simple TASLock

• TAS invalidates cache lines
• Spinners
  – Miss in cache
  – Go to bus
• Thread wants 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

memory busy

busy

busy

busy
On Release

invalid  invalid  free

memory  free

Bus
On Release

Everyone misses, rereads

miss miss free

memory free
On Release

Everyone tries TAS
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

Increses linearly with the number of processors for bus architecture
Mystery Explained

Better than TAS but still not as good as 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

```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;
        }
    }
}
```
Exponential Backoff Lock

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

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

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

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

```java
public class Backoff implements Lock {

    Double max delay, within reason

    int delay = MIN_DELAY;
    while (true) {
        while (state.get()) {}
        if (!lock.getAndSet(true))
            return;
        sleep(random() % delay);
        if (delay < MAX_DELAY)
            delay = 2 * delay;
    }
```
Spin-Waiting Overhead

- TTAS Lock
- Backoff lock

Time

Threads
Backoff: Other Issues

- **Good**
  - Easy to implement
  - Beats TTAS lock
- **Bad**
  - Must choose parameters carefully
  - Not portable across platforms
Idea

- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others
Anderson Queue Lock

flags

next

idle

T  F  F  F  F  F  F  F  F  F  F
Anderson Queue Lock

acquiring

getAndIncrement

next

flags

T  F  F  F  F  F  F  F  F  F  F  F
Anderson Queue Lock

acquiring

next

getAndIncrement

flags

T  F  F  F  F  F  F  F  F  F  F
Anderson Queue Lock

T F F F F F F F F F F

next

flags

acquired

Mine!
Anderson Queue Lock

flags

acquired

acquiring

next
Anderson Queue Lock

next

flags

acquired

acquiring

getAndIncrement

T  F  F  F  F  F  F  F  F  F  F
Anderson Queue Lock

flags

next

acquired

acquiring

getAndIncrement

T  F  F  F  F  F  F  F  F
Anderson Queue Lock

next

acquired

acquiring

flags

T  F  F  F  F  F  F  F  F
Anderson Queue Lock

next

released

acquired

flags

T T F F F F F F
Anderson Queue Lock

released
acquired

flags

next

T T F F F F F F

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
Anderson Queue Lock

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

Next flag to use
Anderson Queue Lock

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;
}

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

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

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

Take next slot
Anderson Queue Lock

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

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

Spin until told to go
Anderson Queue Lock

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

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

Prepare slot for re-use
Anderson Queue Lock

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

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

Tell next thread to go
Local Spinning

released

next

acquired

flags

T
F
F
F
F
F
F
F
F
F
F

Spin on my bit

Unfortunately many bits share cache line
False Sharing

Result: contention

Spin on my

Spinning thread gets cache invalidation on account of store by threads it is not waiting for
The Solution: Padding

flags

released

acquired

next

Spin on my line

T / / / / F / / /
Performance

- Shorter handover than backoff
- Curve is practically flat
- Scalable performance
Anderson Queue Lock

Good

– First truly scalable lock
– Simple, easy to implement
– Back to FCFS order (like Bakery)
Anderson Queue Lock

Bad

– Space hog...
– 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

tail

false
Initially

Queue tail

tail

idle

false
Initially

tail

idle

false

Lock is free
Initially

idle

false

tail
Purple Wants the Lock

acquiring

tail

false
Purple Wants the Lock

acquiring

tail

false → true
Purple Wants the Lock

```
acquiring

Swap

tail

false

true
```
Purple Has the Lock

acquired

false → true

tail
Red Wants the Lock

- Red: acquiring
- Tail: false
- Acquired: true

Art of Multiprocessor Programming
Red Wants the Lock

- acquired
- acquiring

Swap

- tail
  - false
  - true
  - true

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

tail

false

true

true
Red Wants the Lock

acquired

acquiring

tail

false

true

true
Red Wants the Lock

acquired

acquiring

Implicit Linked list

tail

false

true

true

Art of Multiprocessor Programming
Red Wants the Lock

acquired

false

tail

true

true

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

Actually, it spins on cached copy
Purple Releases

release

acquiring

false

false

false

true

Bingo!
Purple Releases

released

acquired

tail

true
Space Usage

• Let
  – \( L = \text{number of locks} \)
  – \( N = \text{number of threads} \)

• ALock
  – \( O(LN) \)

• CLH lock
  – \( O(L+N) \)
CLH Queue Lock

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

Art of Multiprocessor Programming
CLH Queue Lock

```java
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) {}  
    }
}
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) {}
    }
}

Queue tail
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) {} 
    }
}
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) {}}
}
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) {}}
CLH Queue Lock

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

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

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

Class CLHLock implements Lock {
...
    public void unlock() {
        myNode.locked.set(false);
        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
  – 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

false

tail

false -> | |
Acquiring

allocating QNode

tail

false

true
Acquiring

acquired

swap

tail

false

true
Acquiring

acquired

tail

false

true
Acquired

tail

false

true

acquired
Acquiring

acquired

acquiring

false

true

tail

swap
Acquiring

green rectangle: tail

acquired

acquiring

false

true
Acquiring

acquired

acquiring

false

tail

true
Acquiring

acquired

acquiring

tail

false

true

Art of Multiprocessor Programming
Acquiring

acquired

acquiring

true

false

Yes!

tail
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) {}
        }
    }
}
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) {}
        }
    }
}
Purple Release

releasing

swap

false

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

I have to release that thread so must wait for it to identify its node.
Purple Release

releasing  prepare to spin

false  true
Purple Release

releasing

spinning

false

true
Purple Release

releasing

spinning

false

false

false
Purple Release

releasing

Acquired lock

false
false
false
MCS Queue Unlock

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;
    }
}
MCS Queue Lock

If really no successor, return

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

Otherwise wait for successor to catch up

```java
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 queue;
    public void unlock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null))
                return;
            while (qnode.next == null) {}
        }
        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 benefit:
  - No cleaning up
  - Wait-free
  - Immediate return
Queue Locks

• Can’t just quit
  – Thread in line behind will starve

• Need a graceful way out
Queue Locks

spinning

true

spinning

true

spinning

true

Art of Multiprocessor Programming
Queue Locks

locked

spinning

spinning

false

true

true

|||
Queue Locks
Queue Locks

Locked

false
Queue Locks

spinning

true

true

true

| | |
Queue Locks

true

true

true
Queue Locks

Locked

Spinning
Queue Locks

![Diagram of queue locks with 'spinning' and 'false' and 'true']
Queue Locks

false

true

pwned
Abortable CLH Lock

• When a thread gives up
  – Removing node in a wait-free way is hard

• Idea:
  – let successor deal with it.
Initially

Pointer to predecessor (or null)
Initially

Distinguished available node means lock is free
Acquiring
Acquiring
Null predecessor means lock not released or aborted.

Art of Multiprocessor Programming
Acquiring
Acquiring
Acquired

locked

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
Spin on Earlier Node

locked

spinning
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 signifies free lock
public class TOLock implements Lock {
    static QNode AVAILABLE = new QNode();
    AtomicReference<QNode> tail;
    ThreadLocal<QNode> myNode;

    Tail of the queue
Time-out Lock

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;
    }

    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
Time-out Lock

```java
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
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

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

Keep trying for a while

...
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;
  }
}

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
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 aborted, advance one
Time-out Lock

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

What do I do when I time out?
Time-out Lock

Do I have a successor? If CAS fails, I do. Tell it about myPred

```java
if (!tail.compareAndSet(qnode, myPred)) {
    qnode.prev = myPred;
    return false;
}
```
Time-out Lock

... if (!tail.compareAndSet(qnode, myPred))
    qnode.prev = myPred;
    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;
}
Timing-out Lock

```java
public void unlock() {
    QNode 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

various memory locations

MCS
lock

Lock data access in a NUMA machine, showing how different nodes (1 and 2) have access to 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

Thread at local head splices local queue into global queue

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

• General technique for converting almost any lock into a NUMA lock
• Allows combining different lock types
• But need these locks to have certain properties (will discuss shortly)
Lock Cohorting

- **Non-empty cohort**
  - Empty cohort
  - Acquire local lock and proceed to critical section

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

- **Global Lock**
  - Thread that acquired local lock can now acquire global lock...
  - On release: since cohort is empty, must release global lock to avoid deadlock

- **CS** (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 \textit{cohort detection} if
  - It can tell whether any thread is trying to acquire it
Lock Cohortung

• 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
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

Global backoff 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
Lock Cohorting: BO Lock

How to add cohort detection property to BO lock?

As noted BO Lock is thread oblivious.
Lock Cohorting - 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?
C-B-O is a Time-Out NUMA Lock

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

BO locks trivially abortable

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

• **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 schemes. The graph compares C-BO-MCS, C-BO-BO, HCLH, and HBO. The x-axis represents the number of threads ranging from 1 to 256, and the y-axis represents throughput ranging from 0 to 7.0e+06.]
Time-Out (Abortable) Lock Cohorting

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

A-BO-BO

Abortable CLH (our time-out lock) and HBO
One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS, ToLock…
- Each better than others in some way
- There is no one solution
- Lock we pick really depends on:
  - the application
  - the hardware
  - which properties are important
This work is licensed under a 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/.
- 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.