Focus so far: Correctness

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

• Protocols
  – Elegant
  – Important
  – But naive
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

Resets lock upon exit.
Performance

• Experiment
  – $n$ threads
  – Increment shared counter 1 million times
• How long should it take?
• How long does it take?
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 $\rightarrow$ no parallelism
Basic Spin-Lock

...lock suffers from contention

Contension → ???
Graph

no speedup because of sequential bottleneck

ideal

threads

time
Mystery #1

What is going on?

Ideal

filter lock

threads

time
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”
Test-and-Set

```java
public class AtomicBoolean {
    boolean value;

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

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

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(false);
    }
}

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

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

    void unlock() {
    }
}
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 $O(n)$ Filter/Bakery
- How did we overcome the $\Omega(n)$ lower bound?
- We used a Read-Modify-Write (RMW) operation…
Test-and-Test-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()) {}  // Wait until the lock is not held.
            if (!state.getAndSet(true)) {  // Check if the lock can be acquired.
                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

(time) vs (threads)
Mystery

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

• Except that
  – TTAS performs much better than TAS
  – Neither approach is 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

cache

cache

cache

memory

Bus
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
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
Processor Issues Load Request

Bus

memory data
Processor Issues Load Request

Gimme data

memory data

Bus

cache
cache
cache
Memory Responds

Got your data right here

Bus

Memory

data
Processor Issues Load Request

Gimme data

data cache cache

memory data

Bus
Processor Issues Load Request

Gimme data
Processor Issues Load Request

I got data

memory

data

cache

cache

Bus
Other Processor Responds

I got data

memory

data

cache

cache

data

Bus
Other Processor Responds

![Diagram showing memory, cache, and data blocks connected by a bus.](image-url)
Modify Cached Data
Modify Cached Data
Modify Cached Data

memory
data

data

data

cache

Bus
Modify Cached Data

What’s up with the other copies?
Cache Coherence

• We have lots of copies of data
  – Original copy in memory
  – Cached copies at processors

• Some processor modifies its own copy
  – What do we do with the others?
  – How to avoid confusion?
Write-Back Caches

• Accumulate changes in cache
• Write back when needed
  – Need the cache for something else
  – Another processor wants it
• On first modification
  – Invalidate other entries
  – Requires non-trivial protocol …
Write-Back Caches

• Cache entry has three states
  – Invalid: contains raw seething bits
  – Valid: I can read but I can’t write
  – Dirty: Data has been modified
    • Intercept other load requests
    • Write back to memory before using cache
Invalidate
Invalidate

Mine, all mine!

memory data

data cache

Bus
Uh, oh

Invalidate

memory

cache

data

cache

Bus
Other caches lose read 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)
Another Processor Asks for Data
Owner Responds

Here it is!

memory

data
End of the Day …

Reading OK, no writing
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

(busy) (busy) (busy)
On Release

invalid  invalid  free

memory  free

Bus
On Release

Everyone misses, rereads

miss  miss  free

memory  free

Bus
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

TAS lock

TTAS lock

Ideal

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
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()) {}
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}
```
Exponential Backoff Lock

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;
        }
    }
}
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

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

Double max delay, within reason
Spin-Waiting Overhead

- TTAS Lock
- Backoff lock

Graph showing the relationship between time (vertical axis) and threads (horizontal axis) with two lines indicating different lock mechanisms.
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  F
Anderson Queue Lock

flags

next

acquiring

getAndIncrement

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

acquiring

getAndIncrement

next

flags
Anderson Queue Lock

acquired

flags

next

Mine!
Anderson Queue Lock

next

flags

acquired

acquiring

T  F  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 F

96
Anderson Queue Lock

flags

next

acquired

acquiring

getAndIncrement

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

next

flags

acquired

acquiring

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

text

flags

released  acquired

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

flags

T T F F F F F F

next

released

acquired

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
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

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;
}
```
Local Spinning

next

flags

released

acquired

Spin on my bit

Unfortunately many bits share cache line
False Sharing

Result: contention

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

next
flags
released
acquired

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 FIFO 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?
Craig-Landin-Hagersten Lock

- FIFO order
- Small, constant-size overhead per thread
Initially

idle

tail

false
Initially

idle

tail

false

Queue tail
Initially

```
idle
```

```
tail false
```

Lock is free
Initially

tail → false

idle
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

acquired

false

true

acquiring

true

tail
Red Wants the Lock

acquired

acquiring

Swap

tail

false

true

true
Red Wants the Lock

acquired

acquiring

false

true

tail

true
Red Wants the Lock

acquired

acquiring

tail

false

true

true
Red Wants the Lock

acquired

acquiring

Implicit Linked list

tail

false

true

true
Red Wants the Lock

acquired

acquiring

tail

color 1: false

color 2: true

color 3: true
Red Wants the Lock

Actually, it spins on cached copy
Purple Releases

release

acquiring

false

false

false

true

Bingo!

Art of Multiprocessor Programming
Purple Releases

released

acquired

tail

true
Space Usage

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

• ALock
  – $O(LN)$

• CLH lock
  – $O(L+N)$
class Qnode {
    AtomicBoolean locked =
        new AtomicBoolean(true);
}
CLH Queue Lock

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

Not released yet
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 {
    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 {
    AtomicReference<Qnode> tail;
    ThreadLocal<Qnode> myNode = new Qnode();
    public void lock() {
        Qnode pred = tail.getAndSet(myNode);
        while (pred.locked) {}
Class CLHLock implements Lock {

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

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

• Good
  – Lock release affects predecessor only
  – Small, constant-sized space

• Bad
  – Doesn’t work for uncached NUMA architectures
NUMA Architectures

• Acronym:
  – Non-Uniform Memory Architecture

• 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 …
Mellor-Crummey-Scott Lock

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

idle

tail false
Acquiring

(acquire Qnode)

tail

false

true

153
Acquiring

acquired

swap

tail

false

true
Acquired

tail

false

true

acquired
Acquiring

acquired

acquiring

false

true

swap
tail
Acquiring

acquired

acquiring

false

tail

true
Acquiring

acquired

acquiring

false

true

tail
Acquiring

tail

acquired

acquiring

true

false

Yes!
MCS Queue Lock

class Qnode {
    boolean locked = false;
    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) {}}
    }
}
class MCSQueue 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);
        add my Node to the tail of queue
        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) {} // Wait until unlocked
        }
    }
}
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;
    }
}
class MCSLock implements Lock {
    AtomicReference tail;
    public void unlock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null) == true) {
                return;
            }
            while (qnode.next == null) {} // Missing successor
        }
        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

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

Otherwise wait for successor to catch up

172
class MCSLock implements Lock {
    AtomicReference queue;
    public void unlock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null)) {
                if (tail.CAS(qnode, null)) {
                    return;
                }
                while (qnode.next == null) {
                    qnode.next.locked = false;
                }
            }
        }
    }
}
Purple Release

releasing

swap

false

false

false
Purple Release

By looking at the queue, I see another thread is active
Purple Release

releasing

By looking at the queue, I see another thread is active

I have to wait for that thread to finish
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
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
Queue Locks

locked

spinning

spinning

false

true

true
Queue Locks

stopped

true

locked

false

spinning

true
Queue Locks
Queue Locks

spinning

true

true

true
Queue Locks

spinning

true → true → true

spinning
Queue Locks

locked

false

true

spinning

true

| | |
Queue Locks

false → true → spinning

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

idle

Pointer to predecessor (or null)

tail

A
Initially

Distinguished available node means lock is free
Acquiring

acquiring

tail

A
Null predecessor means lock not released or aborted.
Acquiring

acquiring

Swap

A
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 or done
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
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;
    }
    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;
    }
    Swap with tail
    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;
    }

    ...

    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

```java
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
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;
    }
}

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
Time-out Lock

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

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;
}
Timing-out Lock

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
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.