Spin Locks and Contention

Adapted from the Companion slides for The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit
Spin Locks and Contention
Focus so far: Correctness and Progress

• Models
  – Accurate
  – 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 (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

Our space
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
Basic Spin-Lock

...lock introduces sequential bottleneck

Spin lock critical section resets lock upon exit.
Basic Spin-Lock

...lock suffers from contention

![Diagram of Basic Spin-Lock](image-url)

Art of Multiprocessor Programming
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 $\rightarrow$ ???
Java Lock Interface

```java
Lock mutex = new LockImpl(...);
mutex.lock();
try {
    // ...
} finally { mutex.unlock(); }
```
class Peterson
    implements Lock
{
    private boolean[] flag =
        new boolean[2];
    private int victim;
The Peterson Lock

```java
public void lock()
{
    int i = ThreadID.get();
    int j = 1 - i;
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {} 
}
```
Issues

• Mutual exclusion using reads and writes requires space linear in n.
• Wrong proofs or wrong assumptions?
  – Reads/writes are atomic
  – Memory is sequentially consistent
Issues

• Modern multiprocessors typically do not provide sequentially consistent memory.

• They do not guarantee program-order among reads-writes by a thread.
  – Compiler reorder instruction for efficiency
  – Multiprocessor hardware (write buffer)
Memory Fences

• Memory barrier instructions (memory fence) forces outstanding operations to take effect.
  – Programmers responsibility (for Peterson lock?)
  – Expensive (as compareAndSet and getAndSet, both include memory barrier)

• Use compareAndSet and getAndSet instead?
Review: Test-and-Set

- Operate on single word memory that holds a 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”
public class AtomicBoolean {
    boolean value;

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

Review: Test-and-Set
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

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
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
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

```java
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

time

threads

ideal
Mystery #1

What is going on?

TAS lock

Ideal

threads

time

Art of Multiprocessor Programming
Reminder: 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-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()) {}
         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;
            }
        }
    }
}
Mystery #2

- TAS lock
- TTAS lock
- Ideal

Graph showing time on the y-axis and threads on the x-axis.
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

Art of Multiprocessor Programming
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
Terms

• **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

![Diagram showing processor issues with load request to memory with cache involvement.](image-url)
Processor Issues Load Request

Gimme data

Cache

Cache

Cache

Bus

Memory data

Art of Multiprocessor Programming
Memory Responds

Got your data right here
Processor Issues Load Request

Gimme data

Memory

Data

Cache

Cache

Bus
Processor Issues Load Request

Gimme data

memory

data

cache

cache

Bus
Processor Issues Load Request

I got data

memory

data

cache

cache

Bus
Other Processor Responds

I got data

memory

data
Other Processor Responds

Art of Multiprocessor Programming
Modify Cached Data

![Diagram showing data transfer between memory, cache, and bus]
Modify Cached Data
Modify Cached Data
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

memory data

data

cache

Bus

data
Invalidate

Mine, all mine!
Invalidate

Uh,oh

cache data cache

Bus

memory data
Invalidate

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

[Diagram showing the interaction between cache, memory, and data components in a multiprocessor system.]
Owner Responds

Here it is!

cache  data  cache

Bus

memory  data
End of the Day …

- Data
- Data
- Cache

Reading OK, no writing
Simple TASLock

The following observations can be made:

1. Memory access is expensive
2. Processors communicate over a bus
3. Cache mitigates cost of memory accesses
Mutual Exclusion

• What do we want to optimize in the spin lock?
  – 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-set Lock

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

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

    void unlock() {
        state.set(false);
    }
}
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 (writing false to lock variable)
  - Invalidation storm …
- Local spinning.
Test-and-test-and-set Lock

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

    void lock() {
        while (true) {
            while (state.get()) {}  // infinite loop
            if (!state.getAndSet(true))
                return;
        }
    }
}
Local Spinning while Lock is Busy

![Diagram showing local spinning while lock is busy](image-url)
On Release

invalid  invalid  free

memory  free

Bus
On Release

Everyone misses, rereads

miss miss free

memory free
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

Increses linearly with the number of processors for bus architecture

Art of Multiprocessor Programming
Mutual Exclusion

• What do we want to optimize in the spin lock?
  – 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-set Lock

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

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

    void unlock() {
        state.set(false);
    }
}
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 (writing false to lock variable)
  - Invalidation storm …
- Local 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;
        }
    }
}
Local Spinning while Lock is Busy
On Release

Invalid

Invalid

Free

Bus

Memory

Free
On Release

Everyone misses, rereads

miss miss free

Bus

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

Fix minimum 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;
        }
    }
}
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;
        }
    }
}
```

If we win, 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;
        }
```
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;
    }
  }
}
```

Double max delay, within reason.
Spin-Waiting Overhead

![Graph showing Spin-Waiting Overhead with time on the y-axis and threads on the x-axis. The graph compares TTAS Lock and Backoff lock overheads over time.]
Backoff: Other Issues

• **Good**
  – Easy to implement
  – Beats TTAS lock

• **Bad**
  – Must choose parameters carefully
  – Not portable across platforms
Problems with BackoffLock

• Cache-coherence traffic: All threads spin on the same shared location causing cache-coherence traffic on every successful lock access (less than in case of TASLock)

• Critical Section Underutilization: Threads delay longer than necessary, causing the CS to be underutilized.
Idea

• Avoid useless invalidations
  – By keeping a queue of threads
• Each thread
  – Notifies next in line
  – Without bothering the others
Anderson Queue Lock
(Array Based Queue Lock)

flags

next

idle

T  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
Anderson Queue Lock

acquiring

getAndIncrement

flags

next
Anderson Queue Lock

next

flags

acquired

Mine!
Anderson Queue Lock

---

flags

acquired

acquiring

next

---

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

- **acquired**
- **acquiring**

**Flags**:
- T
- F
- F
- F
- F
- F
- F
- F
- F

**GetAndIncrement**
Anderson Queue Lock

flags

acquired

acquiring

next

getAndIncrement

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

acquired

acquiring

next

flags

T F F F F F F F F

Art of Multiprocessor Programming 122
Anderson Queue Lock

released

acquired

next

flags

T
F
F
F
F
F
F
F

Art of Multiprocessor Programming 123
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 \times 2)] = 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

def lock()
    mySlot = next.getAndIncrement()
    while (!flags[mySlot % n]) {
    }
    flags[mySlot % n] = false

def unlock()
    flags[(mySlot+1) % n] = True
Good thing

- Thread-local variables (`mySlot`) are not shared, requires no synchronization and do not generate coherence traffic (accessed by one thread: `get()` and `set()`)
- `flag[]` array is shared. But contention is minimized since each thread spins on its locally cached copy of one array location, reducing invalidation traffic
- Let's see!
Local Spinning

Unfortunately many bits share cache line

next

flags

released

acquired

Spin on my bit

Art of Multiprocessor Programming
False Sharing

Line 1
T F F F F

next

released

acquired

Result: contention

Spin on my bit

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

Line 2

Art of Multiprocessor Programming
The Solution: Padding

flags

next

released

acquired

Spin on my line

Line 1

Line 2
Example

Array

Cache
Example

Array

Cache
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 $\Rightarrow$ one cache line per thread
  • What if unknown number of threads?
  • What if small number of actual contenders?
CLH Lock

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

idle

tail

false
Initially

Queue tail

Art of Multiprocessor Programming
Initially

`idle`

`false`

Lock is free
Initially

idle

false
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

tail

false
true
Red Wants the Lock

acquired

acquiring

tail

false

true

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

false

true

tail

acquiring

true

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

Implicit Linked list

false

true

true

tail
Red Wants the Lock

acquired

acquiring

tail

false

ture

true

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

true

Actually, it spins on cached copy
Purple Releases

release

acquiring

false

false

false

true

Bingo!
Purple Releases

released

acquired

tail

true
Space Usage

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

• ALock
  – \( O(LN) \)

• 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
class CLHLock implements Lock {
  AtomicReference<Qnode> tail;
  ThreadLocal<Qnode> myNode = new Qnode();
  public void lock() {
    Qnode pred = tail.getAndSet(myNode);
    while (pred.locked) {} // 1
  }
}
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;
    }
}
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
  – Does not need to know number of threads
  – Provides fairness

• **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 …
CLH Lock

• Good
  – Lock release affects predecessor only
  – Small, constant-sized space
  – Does not need to know number of threads
  – Provides fairness

• Bad
  – Doesn’t work for uncached NUMA architectures
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

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

idle

false

tail
Acquiring

(allocate Qnode)

tail

false

true
Acquiring
Acquiring
Acquired
Acquiring
Acquiring

acquired

acquiring

false

true

tail
Acquiring

acquired

acquiring

false

tail

true

Art of Multiprocessor Programming
Acquiring

acquired

false

tail

true

Art of Multiprocessor Programming
Acquiring

acquired

tail

false
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) {} }
    }}}
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 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)
                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
class MCSLock implements Lock {
    AtomicReference tail;

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

public void unlock() {
    if (qnode.next == null) {
        if (tail.CAS(qnode, null)) {
            return;
        }
    }
    while (qnode.next == null) {}
}
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;
    }
}
Purple Release

releasing

swap

false

false

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

false
Purple Release

releasing

Acquired lock

false

false

false

true

false

false

true
MCS Lock

• Good
  – Lock release affects predecessor only
  – Small, constant-sized space
  – Does not need to know number of threads
  – Provides fairness
  – Suits cache-less NUMA architecture
  – $O(L + n)$
MCS Lock

• Bad
  – Releasing lock requires spinning
  – Requires more reads, writes and compareAndSet() calls than CLHLock algorithm.
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
Queue Locks

locked

spinning

spinning

false

true

true

Art of Multiprocessor Programming
Queue Locks

locked

spinning

false → true

Art of Multiprocessor Programming
Queue Locks
Queue Locks

spinning

true

spinning

true

spinning

true
Queue Locks

spinning

true

true

true

spinning
Queue Locks

locked

spinning

false

true

true
Queue Locks

false → true

spinning
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
Acquiring

Null predecessor means lock not released or aborted
Acquiring

Swap

A
Acquiring
Reference to AVAILABLE means lock is free.
Automatic Case

locked  spinning  spinning

Normal Case

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
Time-out Lock

```java
public class TOLock implements Lock {
    static Qnode AVAILABLE = new Qnode();
    AtomicReference<Qnode> tail;
    ThreadLocal<Qnode> myNode;
}
```
Time-out Lock

```java
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;
    }
}

Create & initialize node
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;
    }
    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
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;
    }
}
```

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;
    }
}
...
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;
}
}
Time-out Lock

... 

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

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

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

If CAS succeeds: no successor, simply return false
Time-Out Unlock

```java
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
Time-out Lock

• Good
  – Local spinning on cached location
  – Small, constant-sized space
  – Wait-free time out (as also in BackoffLock)

• Bad
  – New node per lock access
  – Going up chain of timed-out node to access critical section.
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