Spin Locks and Contention

(Part 2)

Companion slides for The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit
Basic Spin-Lock

...lock suffers from contention

Art of Multiprocessor Programming
MIMD Architectures

Shared Bus

• Memory Contention
• Communication Contention
• Communication Latency

Distributed
Test-and-set Lock

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

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

    void unlock() {
        state.set(false); 
    }
}
Mystery #2

- TAS lock
- TTAS lock
- Ideal
Bus-Based Architectures

![Diagram showing bus-based architecture with cache and memory modules connected by a bus.](image-url)
Solution: Introduce Delay

- If the lock looks free
- But I fail to get it
- There must be lots of contention
- Better to back off than to collide again
Anderson Queue Lock

released acquired

next

flags

T T F F F F F F
Anderson Queue Lock

• Good
  – First truly scalable lock
  – Simple, easy to implement

• Bad
  – Space hog
  – One bit per thread
    • Unknown number of threads?
    • Small number of actual contenders?
CLH Lock

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

idle

tail

false
Initially

idle

tail
false

Queue tail
Initially

Lock is free

Art of Multiprocessor Programming
Initially

tail

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
Red Wants the Lock

acquired

acquiring

Swap

tail
false
true
true

Art of Multiprocessor Programming
Red Wants the Lock

acquired

false

tail

acquiring

true

true

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

tail

false

true

true

Art of Multiprocessor Programming
Red Wants the Lock

- Acquired
- Acquiring

Implicitely
Linked list

tail

false
true
true

Art of Multiprocessor Programming
Red Wants the Lock

acquired

acquiring

tail

false

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

• ALock
  – \( O(LN) \)

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

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

Art of Multiprocessor Programming
CLH Queue Lock

class CLHLock implements Lock {
    AtomicReference<Qnode> tail =
        new AtomicReference<>(new Qnode());
    ThreadLocal<Qnode> myNode = new Qnode();
    public void lock() {
        Qnode me = myNode.get();
        me.set(true);
        Qnode pred = tail.getAndSet(me);
        while (pred.locked.get()) {}
    }
}
CLH Queue Lock

Class `CLHLock` implements `Lock` {

...  
public void unlock() {
  myNode.get().locked.set(false);
  myNode.remove();
}

}  

Special reset method for ThreadLocals.
It does NOT reset the content of myNode in other Threads
CLH Lock

• Good
  – Lock release affects predecessor only
  – Small, constant-sized space
• Bad
  – Doesn’t work for uncached NUMA architectures
NUMA Architecture

• 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

Art of Multiprocessor Programming
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
Acquiring

(queue)

(acquiring)

(allocate Qnode)

false
Acquiring
Acquired
Acquiring

acquired

acquiring

tail

false

tail

true

Art of Multiprocessor Programming
Acquiring
Acquiring

acquired

false

true

tail

Art of Multiprocessor Programming
Acquiring

acquired

acquiring

false

Yes!

tail

false
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 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;
    }
}
Purple Release

releasing

swap

false

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

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

I have to wait for that thread to finish

Art of Multiprocessor Programming
Purple Release

releasing

prepare to spin

false

ture
Purple Release

releasing

spinning

false

true
Purple Release

releasing

spinning

false

false

false
Purple Release

releasing

Acquired lock

false

false

false
Properties

+ Space: $O(L+N)$
+ Local spinning (in the NUMA sense)
- Spinning on unlock
- needs more atomic operations (including CAS)
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
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
Queue Locks

locked

spinning

false → true → true

true → true → true
Queue Locks

Queue Locks

spinning

false

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

idle

tail

Distinguished available node means lock is free
Acquiring

tail

Art of Multiprocessor Programming 73
Null predecessor means lock not released or aborted.
Acquiring
Acquiring
A Pointer 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

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

...
Time-out Lock

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

What do I do when I time out?
public void unlock() {
    Qnode qnode = myNode.get();
    if (!tail.compareAndSet(qnode, null))
        qnode.prev = AVAILABLE;
    myNode.remove();
}
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/](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.