Spin Locks and Contention

Acknowledgement:
Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming"
by Maurice Herlihy and Nir Shavit
What We'll Cover Today

Chapter 7 of:

Digital copy can be obtained via WUSTL library:
http://catalog.wustl.edu/search/
Today: Revisit Mutual Exclusion

- Performance, not just correctness
- Proper use of multiprocessor architectures
- A collection of locking algorithms that are:
  - Elegant (in their fashion)
  - Important (why else would we pay attention)
  - And realistic (your mileage may vary)
What Should you do if you can’t get a lock?

• Keep trying
  – “spin” or “busy-wait”
  – Good if delays are short
  – Does not make sense on uniprocessor

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

our focus
Basic Spin-Lock

Resets lock upon exit.
Basic Spin-Lock

...lock introduces sequential bottleneck
Basic Spin-Lock

...lock introduces sequential bottleneck
...lock suffers from contention

Notice: these are distinct phenomena
Test-and-Set Locks
Test-and-Test-and-Set Locks
Primitive: Test-and-Set

• Boolean value
• Test-and-set (TAS)
  – Atomically swap true with current value
  – Return value tells if prior value was true or false
• Can reset just by writing false
• In Java: TAS aka “getAndSet”
public class AtomicBoolean {
    boolean value;

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

```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 Using 
AtomicBoolean

```java
public class AtomicBoolean {
    boolean value;

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

Atomically swap old and new values
(This is not how getAndSet is actually implemented.)
Test-and-Set Using
AtomicBoolean

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

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

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

Swapping in true is called “test-and-set” or TAS
An Aside, Recall the Real World

- To prove correctness of locking protocols, we typically assume sequential consistency.
- No modern-day processor implements sequential consistency.
- The compiler can reorder the instructions, too (and it usually does)!
- Use of memory fences is sometimes necessary.
- High-level languages tend to provide additional language features for implementing concurrent protocols that would prevent instruction reordering.
Atomics and Volatile Variables in Java

- In Java, can ask compiler to keep a variable up-to-date by declaring it `volatile`:
  - Inhibits certain reordering, removing from loops, & other “compiler optimizations.”
- Java provides a set of Atomic variables as standard library, which provides a set of atomic RMW (read-modify-write) methods, setter, and getter.
  - Accessing an atomic variable is akin to accessing volatile variable.
- The actual rules are more complicates; we will see more about this in a later lecture.
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)) {}  
  }

  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)) {} // Lock
    }

    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 (read-modify-write) 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
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;
    }
  }
}
class TTASlock {
  AtomicBoolean state =
      new AtomicBoolean(false);

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

- Experiment
  - $n$ threads
  - Increment a shared counter 1 million times (in aggregate across all threads)

- How long should it take?
- How long does it take?
Graph

- no speedup because of sequential bottleneck

Axes:
- Time (vertical)
- Threads (horizontal)

Ideal performance line is shown.

Graph shows that with increasing threads, time does not decrease as expected, indicating a sequential bottleneck.
Mystery #1

What is going on?

TAS lock

Ideal

What is going on?
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)
• When talking about performance, need to think about underlying architecture.
Basic MIMD Architectures

Factors that affect performance:
• Memory Contention
• Communication Contention
• Communication Latency
Bus-Based Architectures
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"

![Diagram of bus-based architectures with shared bus and cache blocks connected to memory via a bus](image-url)
Bus-Based Architectures

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

Cache 1
Cache 2
Cache 3

Bus
Memory
Processor Issues Load Request

load x

Cache

Cache

Cache

Bus

Memory

Data
Memory Responds

Got it!
Processor Issues Load Request

Load x
Other Processor Responds

Got it
Modify Cached Data
Invalidate
Problems with Simple TASLock

• Each 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
On Release

memory free

invalid

invalid

free

Bus
On Release

Everyone misses, rereads

miss miss free

memory free

(1)
On Release

Everyone tries TAS
Problems with Test-and-Test-and-Set

• Upon release everyone incurs misses
• Everyone does TAS
  – Exclusive ownership satisfied sequentially
  – Invalidates others’ caches
• Eventually quiesces after lock acquired
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() == null) {
            }
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}
public class Backoff implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {}
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}
public class Backoff implements Lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {} // 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 (!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;
        }
    }
}
```

Back off for random duration
Exponential Backoff Lock

```java
public class Backoff implements Lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get() == false) {
                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 the relationship between time, threads, TTAS Lock, and Backoff lock](image-url)
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
Anderson Queue Lock

next

flags

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

flags

next

acquiring

getAndIncrement

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

flags

next

acquired

Mine!

T

F

F

F

F

F

F

F

F

F

F

F
Anderson Queue Lock

next

flags

acquired
acquiring

T 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
Anderson Queue Lock

next

acquired
acquiring

flags

getAndIncrement

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

flags

next

acquired

acquiring

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

next

released
acquired

flags

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

flags: T T F F 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
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
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;
}
```

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
Anderson Queue Lock

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

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

Compiler can actually optimize this away
Properly, Anderson Queue Lock

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

    Declare the flags array volatile*, so the loop involves a volatile read.

    *This just means that the array itself is volatile, but not each array element.
Local Spinning

next

Flags

released

acquired

Spin on my bit

Unfortunately many bits share cache line
False Sharing

Flags

next

released

acquired

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

Spin on my bit

T F F F F F

Line 1

Line 2
The Solution: Padding

flags

released

acquired

next

Spin on my line

Line 1

Line 2
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
(by Travis Craig, Erik Hagersten, and Anders Landin)
CLH Lock

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

idle

tail

false
Initially

Queue tail
Initially

Lock is free
Initially
Purple Wants the Lock

acquiring

tail

false
Purple Wants the Lock

acquiring

tail

false

true
Purple Wants the Lock

acquiring

tail

Swap

false

true
Purple Has the Lock

acquired

false

true
Red Wants the Lock

acquired

acquiring

tail

false

true

true
Red Wants the Lock

acquired

acquiring

Swap

tail

false

tail

true

true
Red Wants the Lock

acquired

acquiring

tail

false

true

true
Red Wants the Lock
Red Wants the Lock

Implicit Linked list
Red Wants the Lock

tail

true

false

acquired

acquiring
Red Wants the Lock

Actually, it spins on cached copy
Purple Releases

release

acquiring

false

true

false

false

tail

Bingo!
class Qnode {
    AtomicBoolean locked =
        new AtomicBoolean(true);
}
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();
    ThreadLocal<Qnode> myPred = null;
    public void lock() {
        Qnode pred = tail.getAndSet(myNode);
        myPred.set(pred);
        while (pred.locked) {}
    }
}
class CLHLock implements Lock {
  AtomicReference<Qnode> tail;
  ThreadLocal<Qnode> myNode = new Qnode();
  ThreadLocal<Qnode> pred = null;
  public void lock() {
    Qnode pred = tail.getAndSet(myNode);
    myPred.set(pred);
    while (pred.locked) {} 
  }
}
class CLHLock implements Lock {
    AtomicReference<Qnode> tail;

    ThreadLocal<Qnode> myNode = new Qnode();
    ThreadLocal<Qnode> pred = null;

    public void lock() {
        Qnode pred = tail.getAndSet(myNode);
        myPred.set(pred);
        while (pred.locked) {} 
    }
}
class CLHLock implements Lock {
    AtomicReference<Qnode> tail;
    ThreadLocal<Qnode> myNode = new Qnode();
    ThreadLocal<Qnode> pred = null;
    public void lock() {
        Qnode pred = tail.getAndSet(myNode);
        myPred.set(pred);
        while (pred.locked) {}
    }
}
class CLHLock implements Lock {
    AtomicReference<Qnode> tail;
    ThreadLocal<Qnode> myNode = new Qnode();
    ThreadLocal<Qnode> pred = null;
    public void lock() {
        Qnode pred = tail.getAndSet(myNode);
        myPred.set(pred);
        while (pred.locked) {} // Spin until predecessor releases lock
    }
}
Class CLHLock implements Lock {

... 

public void unlock() {
    Qnode mine = myNode.get();
    mine.locked = false;
    myNode.set(myPred.get());
}
}
Class `CLHLock` implements `Lock` {

...  

public void `unlock()` {
    `Qnode` mine = myNode.get();
    mine.locked = false;
    myNode.set(myPred.get());
}
}
CLH Queue Lock

Class CLHLock implements Lock {
...
public void unlock() {
    Qnode mine = myNode.get();
    mine.locked = false;
    myNode.set(myPred.get());
}
}
Space Usage

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

• ALock
  – \( O(LN) \)

• CLH lock
  – \( O(L+N) \)
CLH Lock

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

• Bad
  – Doesn’t work well for 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
(by John Mellor-Crummey and Michael Scott)
MCS Lock

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

idle

tail
Acquiring

(allocate Qnode)

false

Acquiring

tail
Acquiring
Acquiring

acquired

acquiring

false

false

swap
tail
Acquiring
Acquiring

tail

acquired

acquiring

false

true
Acquiring

acquired

acquiring

false

tail

true
Acquiring

released

acquired

Yes!

true

false

tail
class Qnode {
  volatile boolean locked = false;
  volatile Qnode next = null;
}

Must be volatile
class MCSLock implements Lock {
    AtomicReference tail; //null initially
    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 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 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 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 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 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

gfalse
I don’t see a successor. But by looking at the queue tail, I see another thread is active.
Purple Release

I don’t see a successor. But by looking at the queue tail, 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
Purple Release

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

Missing successor?
class MCSLock implements Lock {

  AtomicReference<qnode> tail;

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

If really no successor, return
MCS Queue Lock

Otherwise wait for successor to catch up

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;
  }
}
MCS Queue Lock

• Good
  – Works better for NUMA architecture
  – Qnodes can be recycled to have the space complexity as CLH locks

• Bad
  – Require spinning (sometimes) to release a lock
  – Requires more CAS than CLH locks
References


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.