Spin Locks and Contention

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Modified for Software1 students
by Lior Wolf and Mati Shomrat
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
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
  [ask another thread to run expensive since switching is pricey]
  – 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
Basic Spin-Lock

...lock suffers from contention

spin lock  critical section  Resets lock upon exit
Basic Spin-Lock

...lock suffers from contention

Notice: these are distinct phenomena

Art of Multiprocessor Programming 10
Basic Spin-Lock

...lock suffers from contention

Seq Bottleneck $\rightarrow$ no parallelism
Basic Spin-Lock

...lock suffers from contention

Contention → ???
Review: Test-and-Set

- **Boolean value**
- **Test-and-set (TAS)**
  - Swap *true* with current value
  - Return value tells if prior value was *true* or *false*
- **Can reset just by writing *false***
- **TAS aka “getAndSet”**
public class AtomicBoolean {
    boolean value;

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

```java
class AtomicBoolean {
    boolean value;

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

Package `java.util.concurrent.atomic`
Review: Test-and-Set

```java
public class AtomicBoolean {
    boolean value;

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

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

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

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

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

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

- **Locking**
  - Lock is free: value is false
  - Lock is taken: value is true
- **Acquire lock by calling TAS**
  - If result is false, you win
  - If result is true, you lose
- **Release lock by writing false**
Test-and-set Lock

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

    void lock() {
        while (state.getAndSet(true)) {} // using TST
    }

    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

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

    void lock() {
        while (state.getAndSet(true)) {} // Release lock by resetting state to false
    }

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

- TAS spin-lock has small “footprint”
- N thread spin-lock uses $O(1)$ space
- As opposed to $O(n)$ in solutions that keep record of who else is interested (we’ll see later)
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

Art of Multiprocessor Programming
Mystery #1

What is going on?

TAS lock

Ideal

threads

time

Art of Multiprocessor Programming
Test-and-Test-and-Set Locks

Main idea:

Split the following lock line to two
while (state.getAndSet(true)) {}
Test-and-Test-and-Set Locks

• Lurking stage
  - Wait until lock “looks” free
  - Spin while read returns \texttt{true} (lock taken)

• Pouncing state
  - As soon as lock “looks” available
  - Read returns \texttt{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

time

threads

TAS lock

TTAS lock

Ideal

Art of Multiprocessor Programming
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

Bus-cache-memory

cache

Bus

memory

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”
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
Processor Issues Load Request

![Diagram showing processor issues with load request]
Processor Issues Load Request

Gimme data

memory data

Bus

cache cache cache
Memory Responds

Got your data right here

Art of Multiprocessor Programming
Processor Issues Load Request

Gimme data

Data
Cache
Cache

Bus
Memory
Data

Art of Multiprocessor Programming
Processor Issues Load Request

Gimme data

data

cache

cache

Bus

memory

data

Art of Multiprocessor Programming

47
Processor Issues Load Request

I got data

Art of Multiprocessor Programming
Other Processor Responds

I got data

Bus

data cache cache

memory data
Other Processor Responds

Art of Multiprocessor Programming
Modify Cached Data

![Diagram showing the modification of cached data.]
ModifyCachedData

Art of Multiprocessor Programming
Modify Cached Data

Art of Multiprocessor Programming
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: meaningless content
  - Valid: I can read but I can’t write (may be cached elsewhere)
  - Dirty: Data has been modified
    • Intercept other load requests
    • Write back to memory before using cache
Invalidate

![Diagram showing memory, cache, and data]

Art of Multiprocessor Programming
Invalidate

Mine, all mine!

data

Bus

memory data

data

data

cache
Invalidate

Uh, oh

Cache
Data
Cache

Memory
Data

Art of Multiprocessor Programming
Invalidate

Other caches lose read permission

Cache  | Data  | Cache
---|---|---
Memory  | Data

Art of Multiprocessor Programming
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!

Cache data

Bus

Memory data

Art of Multiprocessor Programming
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-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

invalid  invalid  free

memory  free
On Release

Everyone misses, rereads

miss miss free

memory free
On Release

Everyone tries TAS
Problems

- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others’ caches
- Eventually quiesces after lock acquired
  - How long does this take?
    Linearly with the number of processors
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 lots of 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;
        }
    }
```
public class Backoff implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {} // Fix minimum delay
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}
Exponential Backoff Lock

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

Wait until lock looks free
Exponential Backoff Lock

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

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

```java
public class Backoff implements Lock {
    private 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]

- TTAS Lock
- Backoff lock

Art of Multiprocessor Programming
Backoff: Other Issues

- **Good**
  - Easy to implement
  - Beats TTAS lock

- **Bad**
  - Must choose parameters carefully
  - Not portable across platforms
Idea

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

flags

next

idle

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

acquiring

getAndIncrement

flags

next

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

acquired

Mine!

flags

t
F 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  F  F
Anderson Queue Lock

flags

next

acquired

acquiring

getAndIncrement

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

flags

acquired

acquiring

getAndIncrement

next

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 F
Anderson Queue Lock

released  acquired

next

flags

T T F F F F F F
Anderson Queue Lock

released acquired

flags

next

T T F F F F F F F

Yow!
Anderson Queue Lock

class ALock implements Lock {
    boolean[] flags={true,false,...,false};
    AtomicInteger next
        = new AtomicInteger(0);
    int[] slot = new int[n];
Anderson Queue Lock

```java
class ALock implements Lock {
    boolean[] flags={true, false, ..., false};
    AtomicInteger next = new AtomicInteger(0);
    int[] slot = new int[n];
}
```

One flag per thread
Anderson Queue Lock

class ALock implements Lock {
    boolean[] flags={true,false,...,false};
    AtomicInteger next = new AtomicInteger(0);
    int[][] slot = new int[n];

    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
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;
}
```
Performance

- Shorter handover than backoff
- Curve is practically flat
- Scalable performance
- FIFO fairness

TTAS
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?
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.