Lecture 8: Locks

2/28/12

slides adapted from The Art of Multiprocessor Programming, Herlihy and Shavit
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.
MIMD Architectures

- Memory Contention
- Communication Contention
- Communication Latency
Revisit Mutual Exclusion

- Think of performance, not just correctness and progress
- Begin to understand how performance depends on our software properly utilizing the multiprocessor machine’s hardware
- And get to know a collection of locking algorithms…
Lock Contention

- **Keep trying**
  - “spin” or “busy-wait”
  - Good if delays are short
- **Give up the processor**
  - Good if delays are long
  - Always good on uniprocessor
Basic Spin-Lock

- Spin lock
- Critical section
- Resets lock upon exit
Basic Spin-Lock

...lock introduces sequential bottleneck
...and introduces contention

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
Basic Spin-Lock

...lock introduces sequential bottleneck

...and introduces contention

no parallelism
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

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);
    }
}
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

time

threads

ideal
Graph

- **time**
- **threads**
- **ideal**

no speedup because of sequential bottleneck
Mystery #1

What is going on?

TAS lock

Ideal

What is going on?

threads

time
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;
        }
    }
}
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
Compare and Swap

- Three operands:
  - a memory location (V)
  - an expected old value (A)
  - new value (B)

- Processor automatically updates location to new value if the value stored is the expected old value.

- Using this for synchronization:
  - read a value A from location V
  - perform some computation to derive new value B
  - use CAS to change the value of V from A to B
public class SimulatedCAS {
    private int value;

    public synchronized int getValue() { return value; }

    public synchronized int compareAndSwap(int expectedValue, int newValue) {
        int oldValue = value;
        if (value == expectedValue)
            value = newValue;
        return oldValue;
    }
}

Lock-free counter:

public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.getValue();
    }

    public int increment() {
        int oldValue = value.getValue();
        while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)
            oldValue = value.getValue();
        return oldValue + 1;
    }
}
Taxonomy

- An algorithm is said to be *wait-free* if every thread makes progress in the face of arbitrary delay (or even failure) of other threads.
- An algorithm is said to be *lock-free* if some thread always makes progress.
  - permits starvation
- An algorithm is said to be *obstruction-free* if at any point, a single thread executed in isolation for a bounded number of steps will complete.
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

CS390C: Principles of Concurrency and Parallelism
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"

Cache

Memory

Bus
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™
Processor Issues Load Request

CS390C: Principles of Concurrency and Parallelism
Processor Issues Load Request

Gimme data

Bus

cache

cache

cache

memory

data

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
Memory Responds

Got your data right here

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
Memory Responds

Got your data right here

CS390C: Principles of Concurrency and Parallelism
Processor Issues Load Request

CS390C: Principles of Concurrency and Parallelism
Processor Issues Load Request

Gimme data

Bus

data

cache
-memory
-data

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
Processor Issues Load Request

I got data

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
Other Processor Responds

I got data

CS390C: Principles of Concurrency and Parallelism
Other Processor Responds

CS390C: Principles of Concurrency and Parallelism
Other Processor Responds

CS390C: Principles of Concurrency and Parallelism
Modify Cached Data
Modify Cached Data
Modify Cached Data

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
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

CS390C: Principles of Concurrency and Parallelism
Invalidate

Mine, all mine!

CS390C: Principles of Concurrency and Parallelism
Invalidate

Uh, oh
Invalidate

Other caches lose read permission
Invalidate

Other caches lose read permission

This cache acquires write permission
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!
Owner Responds

Here it is!

CS390C: Principles of Concurrency and Parallelism
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

CS390C: Principles of Concurrency and Parallelism
On Release

invalid invalid free

memory free

Bus
On Release

Everyone misses, rereads

CS390C: Principles of Concurrency and Parallelism

(1)
On Release

Everyone tries TAS

TAS(...)  TAS(...)  free

Bus

memory  free

CS390C: Principles of Concurrency and Parallelism
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

\[ X = \text{time of ops that don't use the bus} \]
\[ Y = \text{time of ops that cause intensive bus traffic} \]

In critical section, run ops \( X \) then ops \( Y \). As long as Quiescence time is less than \( X \), no drop in performance.

By gradually varying \( X \), can determine the exact time to quiesce.
Quiescence Time

Increases linearly with the number of processors for bus architecture
Mystery Explained

- TAS lock
- TTAS lock
- Ideal

(time) ➔ (threads)

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
Mystery Explained

- TAS lock
- TTAS lock
- Ideal

Better than TAS but still not as good as ideal

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
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()) {
                // Do nothing
            }
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}
```
Spin-Waiting Overhead

- TTAS Lock
- Backoff lock

Plot showing time vs. threads with curves for TTAS Lock and Backoff lock.
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

next

acquiring

getAndIncrement

flags

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

flags

acquiring

getAndIncrement

| T | F | F | F | F | F | F | F | F | F |

next
Anderson Queue Lock

flags

next

acquired

Mine!

T

F

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
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 | F

CS390C: Principles of Concurrency and Parallelism
Anderson Queue Lock

Flags

next

released

acquired

T

F

F

F

F

F

F

F

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
Anderson Queue Lock

next

released

acquired

flags

T | T | F | F | F | F | F | F

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
Anderson Queue Lock

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

CS390C: Principles of Concurrency and Parallelism
Local Spinning

next

flags

released

acquired

Spin on my bit

Unfortunately many bits share cache line
False Sharing

Flags

next

released

acquired

Spin on my bit

Line 1

T | F | F | F | F | F | F | F

Line 2

CS390C: Principles of Concurrency and Parallelism
False Sharing

next released acquired

flags

T | F | F | F | F

Line 1

Line 2

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

Spin on my bit
False Sharing

Result: contention

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

Line 1

T

F

F

F

F

F

next

released

acquired

Line 2

Spin on my

Tuesday, February 28, 12
The Solution: Padding

flags

next

released

acquired

Spin on my line

Line 1

Line 2

CS390C: Principles of Concurrency and Parallelism

Tuesday, February 28, 12
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?
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.