Spin Locks and Contention
CS 475, Spring 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming
# Sequential Consistency vs Linearizability

<table>
<thead>
<tr>
<th></th>
<th>t=0</th>
<th>t=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td>W(X) 1</td>
<td>R(Y) 1</td>
</tr>
<tr>
<td>CPU1</td>
<td>W(Y) 1</td>
<td>R(X) 1</td>
</tr>
</tbody>
</table>

**Linearizable**

**Sequential Consistency**

<table>
<thead>
<tr>
<th></th>
<th>t=0</th>
<th>t=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU0</td>
<td>W(X) 1</td>
<td>R(Y) 1</td>
</tr>
<tr>
<td>CPU1</td>
<td>W(Y) 1</td>
<td>R(X) 0</td>
</tr>
</tbody>
</table>
Sequential Consistency vs Linearizability

• Linearizability can be composed:
  • If p’s execution and q’s execution are both linearizable, then the combination must also be linearizable

• Sequential consistency can not be composed:
  • If p’s execution and q’s execution are both sequential, then the combination MAY also be sequential (but not guaranteed!)

• Why use sequential consistency?
  • Does not require global clock
Memory Operations are Slow

[Diagram showing memory operations with CPU 1 and CPU 2 accessing Main Memory. Times indicated include 7ns for CPU 1 Cache, 100ns for Main Memory, and 7ns for CPU 2 Cache.]
Volatile Keyword

CPU 1

thread0() \[7\text{ns}\] CPU 1 Cache \[X\] 100ns

Main Memory

CPU 2

thread1() \[\text{CPU 2 Cache}\] \[X\]
Today

• Wrap up on conditions and read-write locks

• Reading: H&S 7.1-7.4
• Note: HW2 now out!
• Note: Midterm is now before spring break (hooray?)
Conditions

• When a thread is waiting for something to happen, it might want to release the lock and be notified when that thing has happened
• Ex: while queue is full, release the lock to let someone else empty it
• This is what a condition does
• Key methods: await, signal

```java
Condition condition = lock.newCondition();
lock.lock();
try{
    while(!property)
        condition.await();
} catch(InterruptedException e){
    //Application-dependent response, property may still be false
}
//at this point, property must be true and we have the lock again
condition.signal(); //wake up one thread await'ing
condition.signalAll(); //wake up all threads await'ing
```
Monitors

- Java’s **synchronized** keyword creates a *monitor*, which is both a lock and a condition variable.
- Three relevant methods that have been there all along, and you may not have known what for:
  - `object.notify();`
  - `object.notifyAll();`
  - `object.wait();`
wait and notify()

• Two mechanisms to enable coordination between multiple threads using the same monitor (target of synchronized)
• While holding a monitor on an object, a thread can wait on that monitor, which will temporarily release it, and put that thread to sleep
• Another thread can then acquire the monitor, and can notify a waiting thread to resume and re-acquire the monitor
wait and notify() example

```java
public class BlockingQueue<T> {
    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(T element) throws InterruptedException {
        while (queue.size() == capacity) {
            wait();
        }
        queue.add(element);
        notify(); // notifyAll() for multiple producer/consumer threads
    }

    public synchronized T take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T item = queue.remove();
        notify(); // notifyAll() for multiple producer/consumer threads
        return item;
    }
}
```

Only one thread can be in put or take of the same queue.
Non-mutual exclusion locks

• The strict mutual exclusion property of locks is often relaxed. Three examples are:
  • Readers-writers lock: Allows concurrent readers, while a writer disallows concurrent readers and writers.
  • Reentrant lock: Allows a thread to acquire the same lock multiple times, to avoid deadlock (we have mostly used reentrant locks)
  • Semaphore: Allows at most c concurrent threads in their critical section, for some given capacity c.
Readers and Writers Problem

• Commonly called a read-write lock: data can be read by an unlimited number of threads at a time, written by at most one
• If a thread wants to write, nobody else can be reading
• High level approach:
  • Build on top of mutual exclusion locks with condition variables
Readers and Writers Lock

class ReadLock implements Lock{
    public void lock()
    {
        lock.lock();
        try{
            while(writer){
                condition.await();
            }
        } finally {
            lock.unlock();
        }
    }

    @Override
    public void unlock() {
        lock.lock();
        try{
            readers--;
            if(readers == 0)
                condition.signalAll();
        } finally{
            lock.unlock();
        }
    }
}

class WriteLock implements Lock{
    public void lock()
    {
        lock.lock();
        try{
            while(readers > 0)
                condition.await();
            writer = true;
        } finally{
            lock.unlock();
        }
    }

    @Override
    public void unlock() {
        writer = false;
        condition.signalAll();
    }
}
Readers and Writers Lock

- Note: not fair - many readers can prevent a single writer from getting in
- A fair solution is outlined in the textbook
- For our purposes though, we’ll understand how to implement the basic version, and in practice, use Java’s ReentrantReadWriteLock
Focus so far: Correctness and Progress

- Models
  - Accurate (we never lied to you)
  - But idealized (so we forgot to mention a few things)
- 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)
Today: 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…
What Should you do if you can’t get a lock?

• Keep trying
  – “spin” or “busy-wait”
  – Good if delays are short
    (consider cost of switching threads)

• 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
    (consider cost of switching threads)

• Give up the processor
  - Good if delays are long
  - Always good on uniprocessor
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}
public void unlock() {
    flag[i] = false;
}
Basic Spin-Lock

Resets lock upon exit

spin lock critical section

J. Bell
Basic Spin-Lock

...lock introduces sequential bottleneck

Seq Bottleneck -> no parallelism
Basic Spin-Lock

...lock suffers from contention

Note: Contention and bottlenecking are separate phenomena
Basic Spin-Lock

...lock suffers from contention

spin lock critical section

Resets lock upon exit

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;
    }
}
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

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)
```
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
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)) {} // Keep trying until lock acquired
    }

    void unlock() {
        state.set(false);
    }
}
Test-and-set Lock

```java
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 $O(n)$ Peterson/Bakery
- How did we overcome the $\Omega(n)$ lower bound?
- We used an atomic Read/Modify/Write operation
  - Under the covers uses some special hardware features, not just the synchronized keyword as in the slides (if so, still $O(n)$?)
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 (but no slower)

ideal

threads

time
Mystery #1

Adding MORE threads makes it SLOWER!
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;
        }
    }
}
class TTASLock {
    AtomicBoolean state =
        new AtomicBoolean(false);

    void lock() {
        while (true) {
            while (state.get()) {} // Then try to acquire it
            if (!state.getAndSet(true))
                return;
        }
    }
}
Mystery #2

- TAS lock
- TTAS lock
- Ideal

(time) (threads)
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 …
CAUTION: LEAKY ABSTRACTIONS

CPU Architectures
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.

Most modern desktop/laptop CPUs
MIMD 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"
Bus-Based Architectures

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™

• Cache miss
  – “I had to shlep all the way to memory for that data”
  – Bad Thing™
Processor Issues Load Request

Diagram showing the load request process involving cache, memory, and data transfer via a bus.
Processor Issues Load Request

Gimme data

Cache

Bus

Memory

Data
Memory Responds

Got your data right here

Art of Multiprocessor Programming
Processor Issues Load Request

Gimme data

data cache cache

Bus

memory data
Processor Issues Load Request

Gimme data

Bus

data

cache

cache

memory

data
Processor Issues Load Request

I got data

Bus

data

cache

memory

data
Other Processor Responds

I got data

Bus

memory

data

cache

data

cache

data
Other Processor Responds
Modify Cached Data
Modify Cached Data
Modify Cached Data

![Diagram showing the process of modifying cached data]

1. Data is read from memory.
2. The data is stored in the cache.
3. The cache data is modified.
4. The modified cache data is sent back to memory.
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
Invalidate

Mine, all mine!
Invalidate

Uh, oh

cache data cache

memory data

Bus
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
Owner Responds

Here it is!

cache → data → cache

Bus

memory → data
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

![Diagram of local spinning while lock is busy]

- Busy
- Memory
- Busy

Bus
On Release

![Diagram showing memory status changes on release]

- **Invalid**
- **Free**

Memory changes from invalid to free upon release.
On Release

Everyone misses, rereads

miss miss

memory free

free

Bus

(1)
On Release

Everyone tries TAS

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

Bus memory free

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

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

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**

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
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
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

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

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

- TAS Lock
- TTAS Lock
- Backoff lock
- Ideal
Backoff: Other Issues

- Good
  - Easy to implement
  - Beats TTAS lock

- Bad
  - Must choose parameters carefully
  - Not portable across platforms
Moral of the story?

• EVEN IF we do pretty good with parallelizing most parts our application, we can still see slowdown from **contention** for locks
• For resources infrequently contended, spin locks can be fast because no context switch is necessary
• But, contention in spin locks can have repercussions due to hardware architectures
This work is licensed under a Creative Commons Attribution-ShareAlike license

• This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/

• You are free to:
  • Share — copy and redistribute the material in any medium or format
  • Adapt — remix, transform, and build upon the material
  • for any purpose, even commercially.

• Under the following terms:
  • Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  • ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  • No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.