Acknowledgement

Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

Concurrency issues

- Memory contention:
  - Not all processors can access the same memory at the same time and if they try they will have to queue
- Contention for communication medium:
  - If everyone wants to communicate at the same time, some of them will have to wait
- Communication latency:
  - If takes more time for a processor to communicate with memory or with another processor.

New goals

- Think of performance, not just correctness and progress
- Understand the underlying architecture
- Understand how the architecture affects performance
- Start with Mutual Exclusion

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
  - Suspend yourself and ask the schedule to create another thread on your processor
  - Good if delays are long
  - Always good on uniprocessor

Basic Spin-Lock

- Resets lock upon exit

Our focus
**Basic Spin-Lock**

- Lock suffers from contention
- Spin lock
- Critical section
- Resets lock upon exit

**Contention**

- Contention:
  - When multiple threads try to acquire a lock at the same time
- High contention:
  - There are many such threads
- Low contention:
  - The opposite

**Welcome to the real world**

- Java Lock interface
  - `java.util.concurrent.locks` package

```java
Lock mutex = new LockImpl (...);
...
mutex.lock();
try {
...
} finally {
    mutex.unlock();
}
```

**Why don’t we just use the Filter or Bakery Locks?**

- The principal drawback is the need to read and write \(n\) distinct locations where \(n\) is the number of concurrent threads
- This means that the locks require space linear in \(n\)

**What about the Peterson lock?**

```java
class Peterson implements Lock {
    private boolean[] flag = new boolean[2];
    private int victim;
    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) {};
    }
}
```

**Peterson lock?**

- It is not our logic that fails, but our assumptions about the real world
- We assumed that read and write operations are atomic
- Our proof relied on the assumption that any two memory accesses by the same thread, even to different variables, take place in program order
Why does it not take place in program order?

- Modern multiprocessors do not guarantee program order
- Due to:
  - Compilers: reorder instructions to enhance performance
  - Multiprocessor hardware itself: writes to multiprocessor memory do not necessarily take effect when they are issued
- Writes to shared memory are buffered and written to memory only when needed

What about the Peterson lock?

```java
class Peterson implements Lock {
    private boolean[] flag = new boolean[2];
    private int victim;
    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        victim = i;
        while (flag[j] && victim == i) ;
    }
}
```

Important that these steps take place in program order.

How can one fix this?

- Memory barriers (or memory fences) can be used to force outstanding operations to take effect
- It is the programmer’s responsibility to know when to insert a memory barrier
- However, memory barriers are expensive

Memory barriers

- Synchronization instructions such as getAndSet() or compareAndSet() often include memory barriers
- As do reads and writes to volatile fields

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"

Review: Test-and-Set

```java
public class AtomicBoolean {
    boolean value;
    public synchronized boolean getAndSet(boolean newValue) {
        boolean prior = value;
        value = newValue;
        return prior;
    }
}
```
Review: 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 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**

```java
AtomicBoolean lock = new AtomicBoolean(false);
boolean prior = lock.getAndSet(true);
```
Test-and-set Lock

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

    void lock() {
        while (state.getAndSet(true)) {} // Lock state is AtomicBoolean
    }

    void unlock() {
        state.set(false); // Release lock by resetting state to false
    }
}
```

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

Mystery #1

- TAS lock
- What is going on?
Questions

- Why is TAS so bad (so much worse than ideal)?

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"

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
- Will discuss complexities later

Processor Issues Load Request

Memory Responds

Processor Issues Load Request

Processor Issues Load Request
Processor Issues Load Request

Other Processor Responds

Modify Cached Data
Modify Cached Data

What's up with the other copies?

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 Cache Coherence Protocol

- Accumulate changes in cache
- Write back when needed
  - Need the cache for something else
  - Another processor wants it
- Write-back coherence protocol:
  - Invalidate other entries
  - Requires non-trivial protocol ...

Write-Back Caches

- Cache entry has three states
  - Invalid: contains raw seething bits (meaningless)
  - Valid: I can read but I can't write because it may be cached elsewhere
  - Dirty: Data has been modified
    - Intercept other load requests
    - Write back to memory before using cache

Invalidate
Invalidate

Mine, all mine!

Invalidate

Uh, oh

Invalidate

Other caches lose read permission

Invalidate

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
Test-and-set Lock

```java
Class TASLock {
    AtomicBoolean state = new AtomicBoolean(false);

    void lock() {
        while (state.getAndSet(true)) {} // Has to change to current value while spinning
    }

    void unlock() {
        state.set(false);
    }
}
```

Back to TASLocks

- How does a TASLock perform on a write-back shared-bus architecture?
  - Because it uses the bus, each `getAndSet()` call delays all the other threads
    - Even those not waiting for the lock
  - The `getAndSet()` call forces the other processors to discard their own cached copies – resulting in a cache miss every time
  - They must then use the bus to fetch the new, but unchanged value

TASLock

- When the thread wants to release the lock it may be delayed because the bus is being monopolized by the spinners

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;
        }
    }
}

**Wait until lock looks free**

Then try to acquire it

What about the TTASLock?

- Suppose thread A acquires the lock.
- The first time thread B reads the lock it takes a cache miss and has to use the bus to fetch the new value.
- As long as A holds the lock however, B repeatedly rereads the value – resulting in a cache hit every time.
- B thus produces no extra traffic.

What about the TTASLock?

- However when A releases the lock:
  - A writes false to the lock variable
  - The spinner’s cached copies are invalidated
  - Each one takes a cache miss
  - They all use the bus to read a new value
  - They all call getAndSet() to acquire the lock
  - The first one to acquire the lock invalidates the others who must then reread the value
  - Storm of traffic
Local spinning

- Threads repeatedly reread cached values instead of repeatedly using the bus

Exponential Backoff

- Recall that in the TTASLock, the thread first reads the lock and if it appears to be free it attempts to acquire the lock
- "If I see that the lock is free, but then another thread acquires it before I can, then there must be high contention for that lock"
- Better to back off and try again later

For how long should a thread back off?

- Rule of thumb:
  - The larger number of unsuccessful tries, the higher the contention, the longer the thread should back off.
- Approach:
  - Whenever the thread sees the lock has become free, but fails to acquire it, it backs off before retrying

What about lock-step?

- What happens if all the threads backs off the same amount of time?
- Instead the threads should back off for a random amount of time
- Each time the thread tries and fails to get the lock, it doubles the back-off time, up to a fixed maximum.

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;
        }
    }
}
```

Fix minimum delay
Exponential Backoff Lock

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

Important to note that a thread backs off only when it fails to acquire a lock that was just available, but is not available anymore.

Observing that a lock is held by another thread does not imply backoff.

Spin-Waiting Overhead

- TTAS Lock
- Backoff lock
Backoff: Other Issues

- **Good**
  - Easy to implement
  - Beats TTAS lock
- **Bad**
  - Must choose parameters carefully
  - Sensitive to choice of minimum and maximum delays
  - Sensitive to number of processors and their speed
  - Cannot have a general solution for all platforms and machines

BackoffLock drawbacks

- Cache-coherence Traffic:
  - All threads spin on the same location
- Critical Section Underutilization:
  - Threads delay longer than necessary

Idea

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

Queue Locks

- Cache-coherence traffic is reduced since each thread spins on a different location
- No need to guess when to attempt to access lock – increase critical section utilization
- First-come-first-served fairness

Anderson Queue Lock

```
<table>
<thead>
<tr>
<th>tail</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>
```

Anderson Queue Lock

```
<table>
<thead>
<tr>
<th>tail</th>
<th>acquiring</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>F</td>
</tr>
</tbody>
</table>
```
class ALock implements Lock {
    boolean[] flags={true,false,...,false};
    AtomicInteger tail = new AtomicInteger(0);
    ThreadLocal<Integer> mySlot;
}

Next flag to use
One flag per thread
Thread-local variable
Anderson Queue Lock

```java
public lock() {
    mySlot = tail.getAndIncrement();
    while (!flags[mySlot]) {};
}
public unlock() {
    flags[mySlot] = false;
    flags[(mySlot+1)] = true;
}
```

Although the flags [] array is shared, contention on the array locations are minimised since each thread spins on its own locally cached copy of a single array location.
Performance

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

Anderson Queue Lock

- Good
  - First truly scalable lock
  - Simple, easy to implement
- Bad
  - Not space efficient
  - One bit per thread
    - Unknown number of threads?
    - Small number of actual contenders?

CLH Queue Lock

- Virtual Linked List keeps track of the queue
- Each thread’s status is saved in its node:
  - True – has acquired the lock or wants to acquire the lock
  - False – is finished with the lock and has released it
- Each node keeps track of its predecessors status

Initially

- Initially
  - Idle
  - False
  - Tail

Queue tail

Locked field: Lock is free

Initially

- Idle
  - Tail
  - False

Locked field: Lock is free
Initially

Purple Wants the Lock

Purple Wants the Lock

Purple Wants the Lock

Purple Has the Lock

Red Wants the Lock
Red Wants the Lock

acquired acquiring

Add

tail false true true

Red Wants the Lock

myPred

myPred

tail false true true

Red Wants the Lock

Implicit
Linked list

Red Wants the Lock

Actually, it spins on cached copy
CLH Queue Lock

```java
class CLHLock implements Lock {
    AtomicReference<Qnode> tail;
    ThreadLocal<Qnode> myNode = new Qnode();
    public void lock() {
        myNode.locked = true;
        Qnode pred = tail.getAndSet(myNode);
        while (pred.locked) {}
    }
}
```

CLH Queue Lock

```java
class CLHLock implements Lock {
    AtomicReference<Qnode> tail;
    ThreadLocal<Qnode> myNode = new Qnode();
    public void lock() {
        myNode.locked = true;
        Qnode pred = tail.getAndSet(myNode);
        while (pred.locked) {}
    }
}
```

Thread-local Qnode

Add my node

Queue tail

Purple Releases

release

acquiring

tail

false

false

true

Bingo!

Purple Releases

released

acquired

tail

false

true

CLH Queue Lock

```java
class CLHLock implements Lock {
    AtomicReference<Qnode> tail;
    ThreadLocal<Qnode> myNode
        = new Qnode();
    public void lock() {
        myNode.locked = true;
        Qnode pred
            = tail.getAndSet(myNode);
        while (pred.locked) {} 
    }
}
```

Spin until predecessor releases lock

Notify successor

Recycle predecessor’s node

CLH Lock

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

MCS Lock

- FIFO order
- Spin on local memory only
- Small, Constant-size overhead
MCS Queue Lock

- Similar to CLHLock, but the linked list is explicit instead of implicit
- Each node in the Queue has a next field

Initially

Acquiring

Acquiring

Acquiring

Acquiring
MCS Queue Lock

class Qnode {
    boolean locked = false;
    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 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 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 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 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 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;
    }
}

MCS Queue Lock

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;
    }
}

MCS Queue Lock

If really no successor, ↩
public void unlock() {
    if (qnode.next == null) {
        if (tail.CompareAndSet(qnode, null))
            return;
        while (qnode.next == null) {}
    }
    qnode.next.locked = false;
}

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

Pass lock to successor
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
swapping
false
false
false
true
true
By looking at the queue, I see another thread is active. I have to wait for that thread to finish.
Lock with timeout

- Java interface includes a `tryLock()` method to specify a maximum duration the thread is willing to wait to acquire the lock
- Should the thread not acquire the lock in the designated time, the thread will timeout

Abortable Locks

- What if you want to give up waiting for a lock?
- For example
  - Timeout
  - Database transaction aborted by user

Lock with timeout

- What happens to other threads when you timeout?

Back-off Lock

- Aborting is trivial
  - Just return from lock() call
- Extra benefit:
  - No cleaning up
  - Wait-free
  - Immediate return

MCS Queue Locks

- `spinning`
- `locked`
- `true`
MCS Queue Locks

locked → spinning
false → true

MCS Queue Locks

locked → false

MCS Queue Locks

spinning → spinning → spinning
true → true → true

MCS Queue Locks

spinning → true → false

MCS Queue Locks

locked → spinning
false → true

MCS Queue Locks

spinning → spinning
false → true
MCS Queue Locks

CLH Queue Lock

Queue Locks
- Can’t just quit
  - Thread in line behind will starve
- Need a graceful way out

Abortable CLH Lock
- When a thread gives up
  - Removing node from queue in a wait-free way is hard
- Idea for lazy approach:
  - Let successor deal with it.
A queue lock with timeouts

- When a thread times out, it marks it node as abandoned
- The successor in the queue notices that the node has been abandoned
- Successor starts spinning on the abandoned node’s predecessors

Initially

- Pointer to predecessor (or null)

Acquiring

- Distinguished available node means lock is free

Acquiring

- Null predecessor means lock not released or aborted

Acquiring

- Pointer to AVAILABLE means lock is free.
public class TOLock implements Lock {
    static Qnode AVAILABLE = new Qnode();
    AtomicReference<Qnode> tail;
    ThreadLocal<Qnode> myNode;
}

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;
    }
    create & initialize node
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;
    }

    ...  // Keep trying for a while...

    Qnode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }

    ...  // Spin on predecessor's prev field

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

    ...  // Predecessor released lock

    // Keep trying for a while...

    if (predecessor absent or released, we are done

    ...
Time-out Lock

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

What do I do when I time out?

Predecessor aborted, advance one

Time-out Lock

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

Do I have a successor? If CAS fails: I do have a successor, tell it about myPred

If CAS succeeds: no successor, simply return false

Time-Out Unlock

```java
public void unlock() {
    Qnode qnode = myNode.get();
    if (!tail.compareAndSet(qnode, null))
        qnode.prev = AVAILABLE;
}
```

If CAS failed: exists successor, notify successor it can enter
Timing-out Lock

```java
public void unlock() {
    Qnode qnode = myNode.get();
    if (!tail.compareAndSet(qnode, null)) {
        qnode.prev = AVAILABLE;
    }
}
```

CAS successful: set tail to null, no clean up since no successor waiting