Spin Locks and Contention
The goal

The goal of this lecture is to understand how architecture affects performance, and how to exploit this knowledge to write efficient concurrent programs.
Basic Spin-Lock

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

lock suffers from contention

spin lock \rightarrow \text{critical section} \rightarrow \text{CS} \rightarrow \text{Resets lock upon exit}
Spin lock and blocking

- **Spin lock** - repeatedly testing the lock. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of **busy waiting**.

- **blocking** - suspend yourself and ask the operating system’s scheduler to schedule another thread on your processor. Blocking makes sense if you expect the lock delay to be long.
1  Lock mutex = new LockImpl(...); // lock implementation
2      ...
3  mutex.lock();
4  try {
5      ... // body
6  } finally {
7      mutex.unlock();
8  }
Peterson algorithm

Lock()
  flag[i]=true
  turn=1-i
  While (flag[1-i] and turn=1-i)
  Unlock()
Unlock()
  flag[i]=false
Not providing mutual exclusion in practice

- Running the Peterson algorithm in practice will provide a slightly off value than expected.
- It must be that both threads are occasionally in the critical section at the same time, even though we have proved that this cannot happen.

“How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?”
Why?

- compilers that reorder instructions to enhance performance - It is therefore possible that the order of writes of flag[i] and turn by thread i will be reversed by the compiler.

- writes to multiprocessor memory do not necessarily take effect when they are issued - it is therefore possible that the order of i’s write to turn is delayed and may arrive in memory only after i reads flag[j].
GetAndSet()

- Atomic instruction.
- `getAndSet(b)` that atomically replaces the current value with `b`, and returns the previous value.

- We will now look at two algorithm that uses this method.
Solution - Test-And-Set Locks

```java
public class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

    public void lock() {
        while (state.getAndSet(true)) {}  // TASLock algorithm
    }

    public void unlock() {
        state.set(false);
    }
}
```
public class TTASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);
    public void lock() {
        while (true) {
            while (state.get()) {};
            if (!state.getAndSet(true))
                return;
        }
    }
}
TTASLock un-lock algorithm

```java
8   public void unlock() {
9     state.set(false); }
```
Schematic performance of a TASLock, a TTASLock, and an ideal lock
TASLock vs. TTASLock

- Both algorithms are the same.
- Yet they have different performant - TTASLock perform better than TASLock.
- Both algorithms perform poorly vs. the ideal performance.
Bus architecture

- cache
- cache
- cache
- memory
TASLock vs. TTASLock

TASLock:
- Each test-and-set call go over the bus.
- Thread that want to release the lock will be delayed.
- Threads not waiting to lock can be delayed when access memory.

TTASLock:
- Set calls not use bus if value in cache.
- Thread that want to release the lock will not be delayed.

TTASLock is still far from ideal because when the lock is free all threads try test-and-set call
Solution - Backoff

- When lock is free: tries to lock it
  - If succeed: great
  - Else: do backoff

- We will like that the larger the unsuccessful tries the longer the backoff
public void backoff() throws InterruptedException {
    int delay = random.nextInt(limit);
    limit = Math.min(maxDelay, 2 * limit);
    Thread.sleep(delay);
}
public class BackoffLock implements Lock {
    private AtomicBoolean state = new AtomicBoolean(false);
    public void lock() {
        Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
        while (true) {
            while (state.get()) {};
            if (!state.getAndSet(true)) {
                return;
            } else {
                backoff.backoff();
            }
        }
    }
}
Backoff unlock

12 public void unlock() {
13    state.set(false);
14 }
Performance graph

- TTAS Lock
- Backoff lock

(time, threads)
Backoff algorithm

Pros:
- Better than TTASLock in contention (less contention).
- Easy to implement.

Cons:
- All threads spin on the same shared location causing cache-coherence traffic on every successful lock access.
- Threads delay longer than necessary, causing the critical section to be underutilized.
- Parameters need to be choose carefully.
Solution - queue lock

- each thread spin on a different cell in the queue.
- Thread is notified when it is its time to go to the critical section through the queue.
Queue Lock

next

idle

flags

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

next

acquiring

getAndIncrement

flags

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

Flags:

next

idle

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

- `next`
- `acquiring`
- `getAndIncrement`
- `flags`

```
T F F F F F F F F F
```
Queue Lock

acquiring

getAndIncrement

flags

next

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

flags

next

acquired

Mine!
Queue Lock

flags

next

acquired

acquiring

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

- flags
- next
- acquired
- acquiring
- getAndIncrement

```
T  F  F  F  F  F  F  F  F  F
```
Queue Lock

next

flags

acquired

acquiring

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

flags

next

released

acquired

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

next

released

acquired

flags

T T F F F F F F F

Yow!
Queue Locks

```java
public void lock() {
    int slot = tail.getAndIncrement() % size;
    mySlotIndex.set(slot);
    while (!flag[slot]) {
    }
}

public void unlock() {
    int slot = mySlotIndex.get();
    flag[slot] = false;
    flag[(slot + 1) % size] = true;
}
```
Performance graph

- time
- threads
- queue
- TTAS
Local Spinning

flags

next

released

acquired

Unfortunately many bits share cache line

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

Line 2
acquired
released
next

spin on my b**it**
Solution: Padding

next
released
acquired

flags

T / / / / F / / / /

Line 1

Line 2

Spin on my line
Queue Locks

Pros:
- first-come-first-served fairness.
- Easy to implement.
- Scalable performance.

Cons:
- Not space-efficient $o(n)$ per lock for $n$ threads.
- Specifically one cache line per thread.
Initially

idle

Lock is free

tail

false
Initially

idle

tail

false
Pink Wants the Lock

acquiring

tail

false
Pink Wants the Lock

acquiring

tail

false  true
Pink Wants the Lock

Diagram:
- Pink block labeled "acquiring" with arrows pointing to:
  - Blue box labeled "tail" with an arrow pointing to "false"
  - Blue box labeled "false" with an arrow pointing to "true"
  - Blue box labeled "true"
  - Blue box labeled "Swap"
Pink Has the Lock

acquired

tail

false

true
Red Wants the Lock

acquired

acquiring

false

true

true

tail
Red Wants the Lock

acquired

acquiring

tail

false

true

true

Swap
Red Wants the Lock

acquired

acquiring

tail

false

true

true
Red Wants the Lock

[Diagram showing a lock in two states: acquired and acquiring, with transitions between states labeled as true and false for tail]
Red Wants the Lock

acquired

acquiring

tail

false

true

true

Implicit Linked list
Red Wants the Lock

- acquired
- acquiring
- tail
- false
- true
- true
Red Wants the Lock

Actually, it spins on cached copy
Pink Releases

release

acquiring

false

false

true

Bingo!
Pink Releases

released

acquired

tail

true
public void lock() {
    QNode qnode = myNode.get();
    qnode.locked = true;
    QNode pred = tail.getAndSet(qnode);
    while (pred.locked) {}
CLH Lock

Pros:
- Constant size space.
- First-come-first-served fairness.
- Each thread spin on a distinct location.
- Does not require a knowledge of number of threads to come.

Cons:
- Doesn’t work for un-cached NUMA architectures.
NUMA Machines

Spinning on local memory is fast
NUMA Machines

Spinning on remote memory is slow
The thread spins on a (local) locked field in its own QNode waiting until its predecessor sets this field to \texttt{false}.

As a result each thread controls the location on which it spins.
Lock with timeout

- Up till now we studied first come first serve with some contention algorithm.
- In real time system sometimes thread need the ability to give up waiting for lock.
- In Backoff lock the thread can simply return from the lock() function.
- In queue lock algorithm if a thread just returns the threads queued up behind it will starve.
Queue Locks

spinning

true

spinning

true

spinning

true
Queue Locks

locked

spinning

spinning

false -> true -> true
Queue Locks

locked

false

spinning

true
Queue Locks
Queue Locks

spinning

true

spinning

true

spinning

true

||
Queue Locks

spinning

true → true → true

spinning
Queue Locks

locked

spinning

false → true → true
Queue Locks

- Spinning
- False
- True
Timeout in CLH lock

- The idea here is to let the successor know about the abandoned so it will deal with it.

- The thread successor in the queue, if there is one, notices that the node on which it is spinning has been abandoned, and starts spinning on the abandoned node’s predecessor.
Null means lock is not free & request not aborted
One Thread Aborts

locked

Timed out

spinning
Successor Notices

locked

Timed out

spinning

Non-Null means predecessor aborted
Recycle Predecessor’s Node

locked

spinning
Spin on Earlier Node

locked

spinning

Art of Multiprocessor Programming
Spin on Earlier Node

released

spinning

The lock is now mine
Timeout in CLH lock

1 public boolean tryLock(long time, TimeUnit unit) {
2     QNode qnode = new QNode();
3     myNode.set(qnode);
4     qnode.pred = null;
5     QNode myPred = tail.getAndSet(qnode);
6     if (myPred == null || myPred.pred == AVAILABLE) {
7         return true;
8     }
9 }

Timeout in CLH lock

8  long start=now()
9  while (now()- startTime < patience) {
10     QNode predPred = myPred.pred;
11     if (predPred == AVAILABLE) {
12         return true;
13     } else if (predPred != null) {
14         myPred = predPred; }
15  }
if (!tail.compareAndSet(qnode, myPred))
qnode.pred = myPred;
Return }

If my time is up!
Timeout in CLH un-lock

18 public void unlock() {
19 QNode qnode = myNode.get();
20 if (!tail.compareAndSet(qnode, null))
21 qnode.pred = AVAILABLE;
22 }

consider an advanced lock algorithm that combines the best of Queue locks and backoff lock.

The CompositeLock class keeps a short, fixed-size array of lock nodes.

Each thread that tries to acquire the lock selects a node in the array at random.

If: that node is in use then backoff.

Else: The thread spins on the preceding node, and when that node’s owner signals it is done, the thread enters the critical section.
Composite Lock

Pros:
- When threads back off, they access different locations, reducing contention.
- Abandoning a lock request is easy.
- For $L$ locks and $n$ threads, the CompositeLock class, requires only $O(L)$ space.

Cons:
- Complex code.
- No guarantee first-come-first-served.
Conclusion

- We have seen a variety of spin locks that vary in characteristics and performance.
- Each spin lock algorithm will work best in different situations.
- This depends on the application, hardware and which properties are important.
Thank you for listening!