Replication and Consistency

08 Spin Locking and Contention

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Thank you!

These slides are based on companion material of the following books:

- **The Art of Multiprocessor Programming** by Maurice Herlihy and Nir Shavit
- **Synchronization Algorithms and Concurrent Programming** by Gadi Taubenfeld

Previously on Replication and Consistency

- Models
  - Accurate (we never lied to you)
  - But idealized (we forgot to mention a few things)
- Protocols
  - Elegant
  - Essential
  - But naive

New Focus: Performance in Real Systems

- Models
  - More complicated (more details)
  - Still focus on principles (not soon to become obsolete)
- Protocols
  - Elegant (in their fashion)
  - Important (why else would we discuss them)
  - And realistic (more optimizations will be possible, though)

Mutual Exclusion, revisited

- 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**

If a processor doesn't get a lock . . .

Question

What can the processor do?

If a processor doesn't get a lock . . .

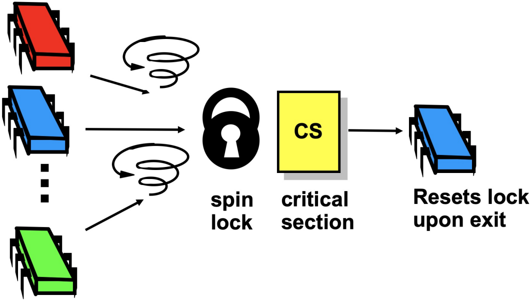
Question

What can the processor do?

- Keep trying
  - “spin” or “busy-wait” as with Filter and Bakery algorithm
  - Useful on multi-processors if expected delays are short
- Suspend and allow scheduler to schedule other processes
  - “blocking' ' as with Java's monitors
  - Good if delays are long
  - Always good on uniprocessors
- In practise, often mix of both strategies
  - Spin for a short time
  - Then, suspend

Basic Spin-Lock

- **Contention:** Multiple threads try to acquire lock at the same time
- Hoch can we avoid or alleviate contention?



Test-and-Set (TAS) revisited

- Machine-instruction on one word (*here:* for boolean values)
- Atomically, swap new value with prior value and return prior value
- Swapping in **`true`** is called Test-And-Set
- Aka `getAndSet()` in Java

```
\\ Package java.utitl.concurrent.atomic

public class AtomicBoolean {
  boolean value;

  // implemented as one hardware instruction
  public synchronized boolean getAndSet(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}
```

Task: Design a lock using Test-and-Set (TAS)!

```java
class TASLock implements Lock{

  // if false, lock is free
  // if true, lock is taken
  AtomicBoolean state = new AtomicBoolean(false);

  void lock() {
    // TODO
  }

  void unlock() {
    // TODO
  }
}
```

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

Question

How did we overcome the $\Omega(N)$ lower bound?

Space Complexity

- TAS spin-lock has small "footprint"
  - N thread spin-lock uses $O(1)$ space
  - As opposed to $O(N)$ Peterson/Bakery

Question

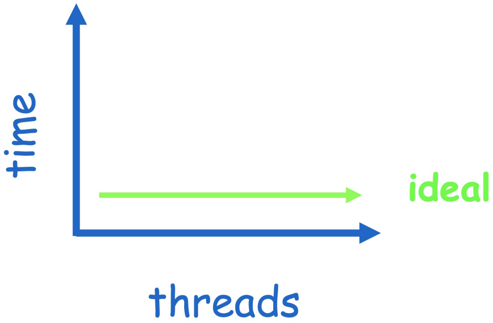
How did we overcome the $\Omega(N)$ lower bound?
⇒ Use an object with higher consensus number!

Performance Evaluation

- Experiment
  - Spawn N threads
  - Increment shared counter 1 million times
  - Work is split between the threads, i.e. each thread does $10^6/N$ increments
  - Each thread takes lock, increments a counter, releases lock
- How long should it take?
- How long does it take?

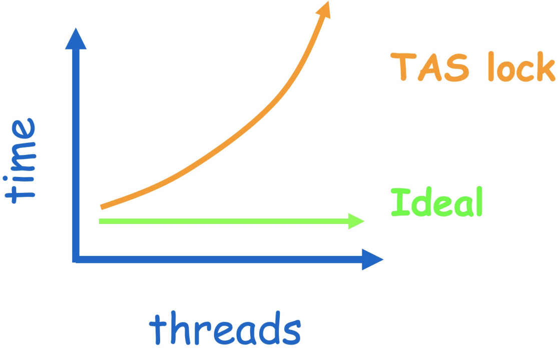
Hypothesis

- No speedup because lock is sequential bottleneck (Amadahl's law!)



Mystery 1

A typical evaluation looks like this:



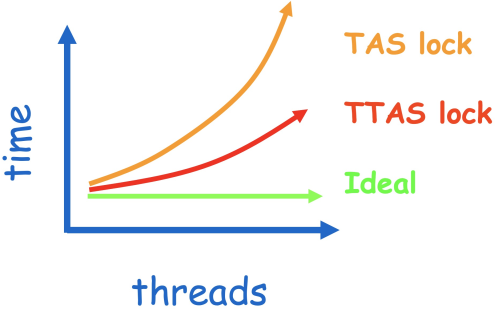
New approach: Test-and-Test-and-Set Locks

- Lurking stage
    - Wait until lock seems to be free
    - Spin while read returns true (lock taken)
- Pouncing state
    - As soon as lock seems to be available
    - Read returns false (lock free)
    - Call TAS to acquire lock
    - If TAS loses, back to lurking

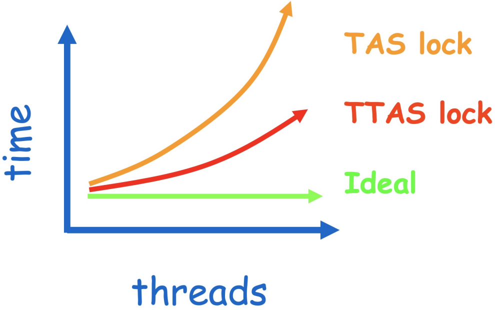
Test-and-Test-and-Set Locks

```
class TTASLock extends TASLock{
  void lock() {
    while (true) {
      while (state.get()) {} // Lurk
      if (!state.getAndSet(true)) // Pounce
        return;
  }
}
```

Mystery 2



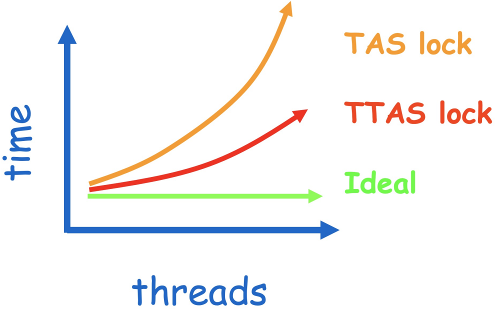
Mystery 2



- Both TAS and TTAS do the same thing in our model
- But TTAS performs much better in actual evaluations
- Neither approach is ideal

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

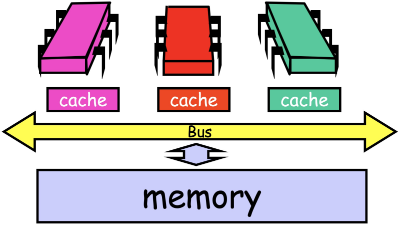
Mystery 2



- Both TAS and TTAS do the same thing in our model
- But TTAS performs much better in actual evaluations
- Neither approach is ideal

Our memory abstraction is broken! We need a more detailed model!

Bus-Based Architectures



- Random Access Memory (access time: 10s of cycles)
- Shared Bus as broadcast medium
  - One broadcaster at a time
  - Other processors and memory can passively listen
- Per-Processor Caches (access time: 1-2 cycles)

Cache Coherence

- We have lots of copies of data
  - Original copy in memory
  - Cached copies at processors
- If some processor modifies its own copy:
  - What do we do with the others?
  - How to avoid confusion about actual value?

Cache Coherence

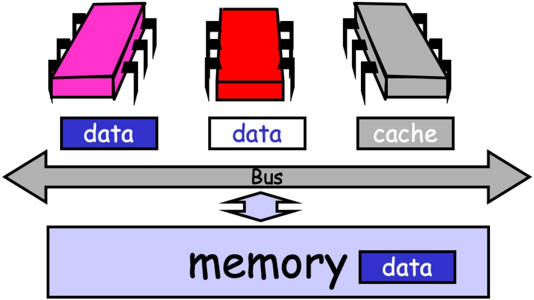
- We have lots of copies of data
  - Original copy in memory
  - Cached copies at processors
- If some processor modifies its own copy:
  - What do we do with the others?
  - How to avoid confusion about actual value?

Cache coherence protocol!

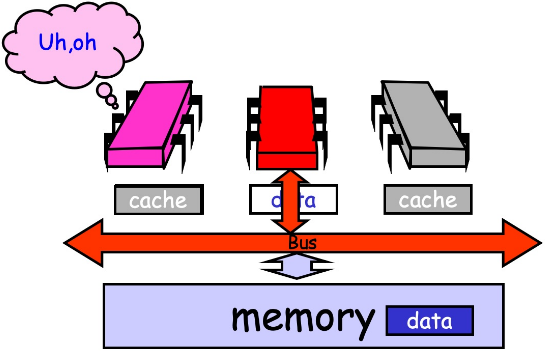
Write-Back Caches

- **Idea:** Accumulate changes in cache and write back when needed
  - Because we need cache for something else
  - Or because another processor wants to read the changed value
- On first modification, invalidate all other entries
- Cache entry can be marked as **dirty** (i.e. it must be eventually written back to main memory)

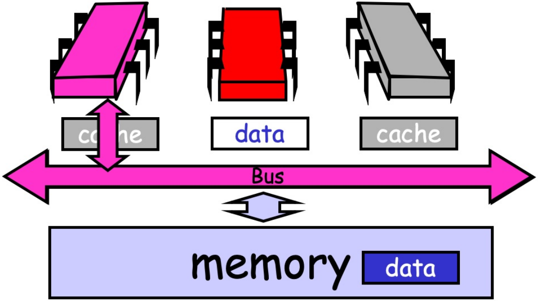
When a thread modifies its cache value, . . .



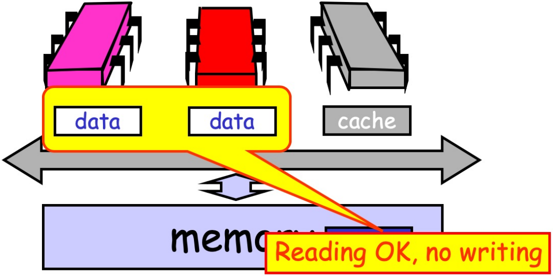
. . . it invalidates all other caches



When another thread want to read, . . .



. . . the owner responds



Mystery Explained!

TAS-Lock

- Spinning threads invalidate cache line with TAS, keeps bus busy
- Threads wanting to release lock is delayed behind spinners

TTAS-Lock

- Threads spin on local cache
- No bus use while lock is taken
- *Problem:* When lock is released, reads are satisfied sequentially on bus
- Eventually system **quiesces** after lock has been acquired

→ quiescence time linear in number of threads for bus architecture

Solution: Introduce Delay

“If the lock looks free, but I fail to get it, there must be lots of contention!”

$\Rightarrow$ Better to back off than to collide again

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

Example: Exponential Backoff

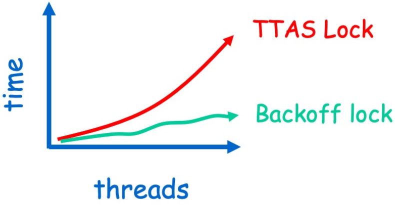
If I fail to get lock

- Wait random duration before retry
- Each subsequent failure doubles expected wait (up to fixed maximum)

Exponential Backoff Lock

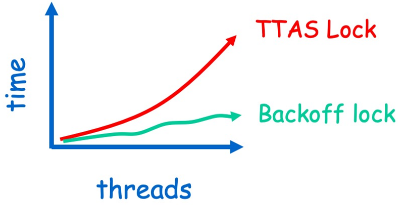
```
class Backoff extends TTASLock {

  void lock() {
    int delay = MIN_DELAY;
    while (true) {
        while (state.get()) {}
        if (!lock.getAndSet(true))
            return;
        // if not successful, we wait
        sleep(random() % delay);
        if (delay < MAX_DELAY)
            delay = 2 * delay;
    }
  }
}
```

Exponential Backoff Lock



- Easy to implement
- But must choose parameters carefully
- Not portable across platforms

Exponential Backoff Lock

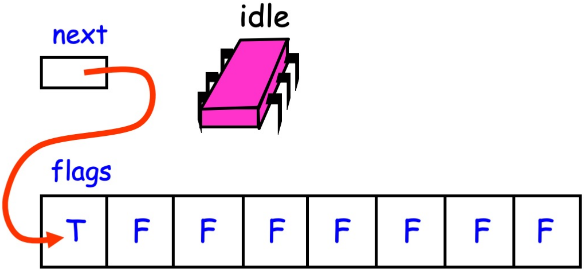


- Easy to implement
- But must choose parameters carefully
- Not portable across platforms

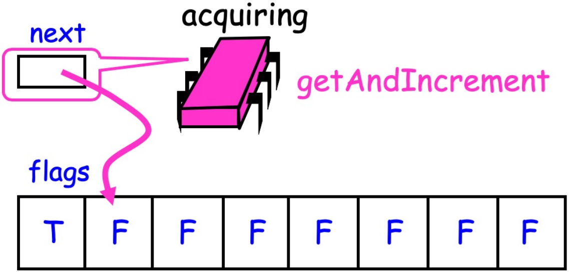
Idea

- Avoid useless invalidations by keeping a queue of threads
- Each thread notifies next in line without bothering the others

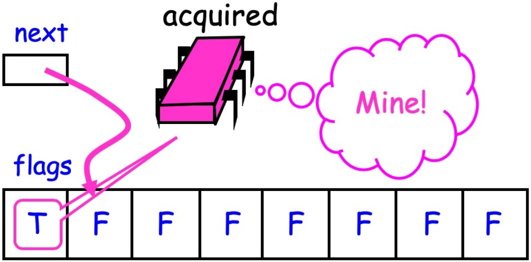
Anderson Queue Lock



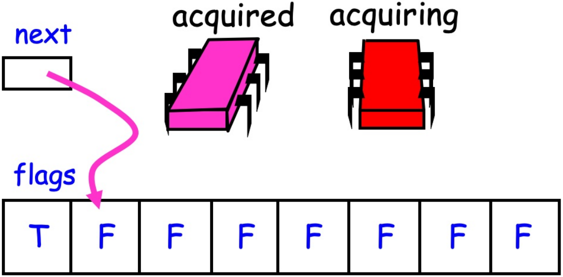
Anderson Queue Lock



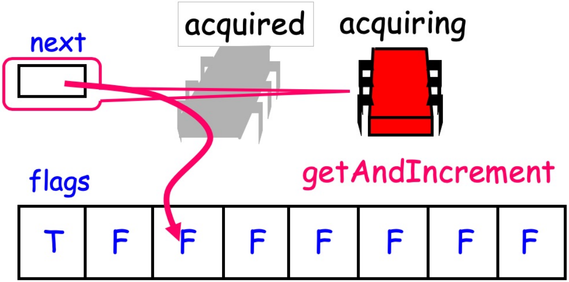
Anderson Queue Lock



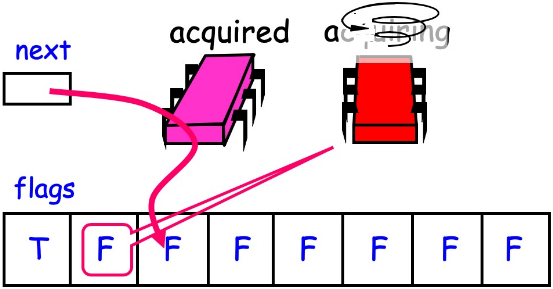
Anderson Queue Lock



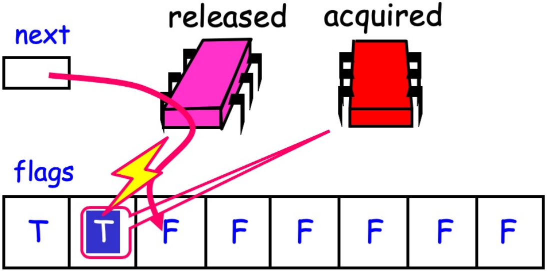
Anderson Queue Lock



Anderson Queue Lock



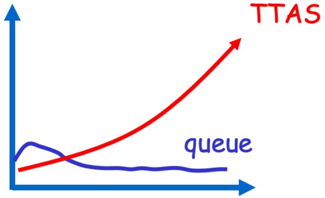
Anderson Queue Lock



Anderson Queue Lock

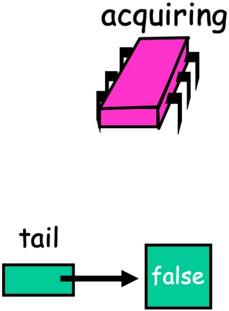
```
class ALock implements Lock {
  boolean[] flags = {true,false,...,false}; // one per thread
  AtomicInteger next = new AtomicInteger(0);
  ThreadLocal<Integer> mySlot;  // thread-local per thread

  void lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {}; //spin
    flags[mySlot % n] = false;  // prepare for re-use (wrong in
    Figure!)
  }

  void unlock() {
    flags[(mySlot+1) % n] = true; // tell next thread
  }
}
```

Anderson Lock

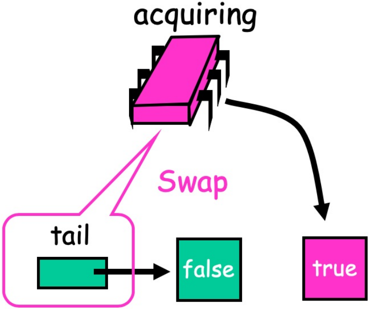


- FIFO fairness, no lockout
- Scalable performance
  - Threads spin on locally cached copy of single array location
  - But beware of *false sharing* of items on the same cache line!
  - Invalidations always per cache line
  - *Trick:* Use padding to avoid sharing
- Not space-efficient
- Requires knowledge about number of threads

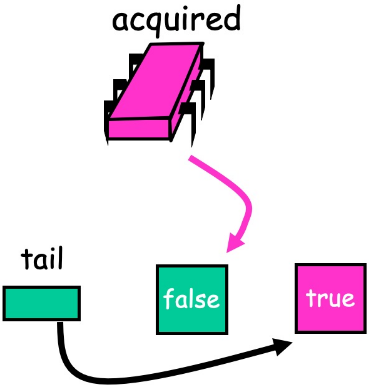
CLH Lock (by Craig, Landin, Hagersten)



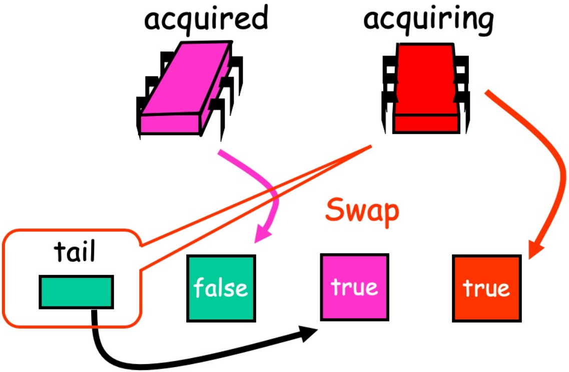
CLH Lock: Acquiring a lock



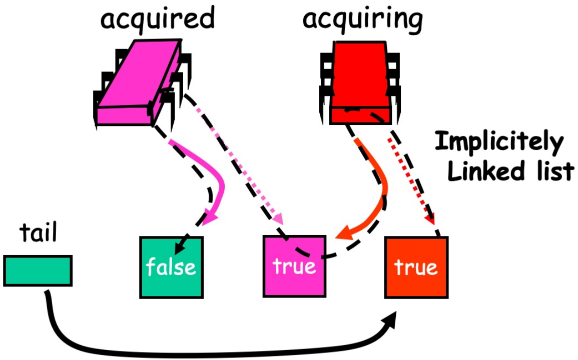
CLH Lock: Acquiring a lock



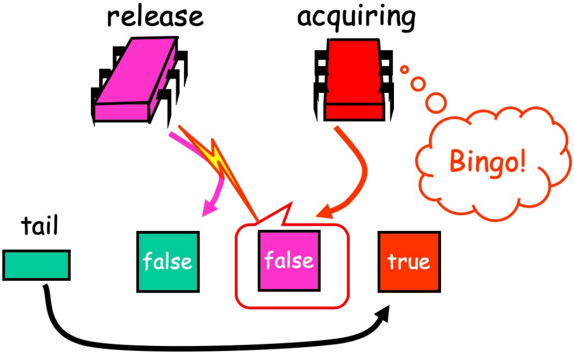
CLH Lock: It's a Queue!



CLH Lock: Releasing a lock



CLH Lock: Releasing a lock



Remarks

- Threads spin on cached copy (efficient)
- Lock can reuse predecessor's node for future lock accesses

CLH Lock

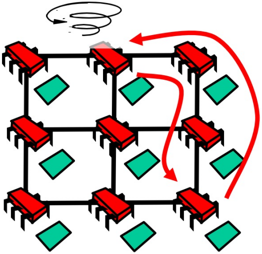
```
class Qnode {
  AtomicBoolean locked = new AtomicBoolean(true);
}

class CLHLock implements Lock {

  AtomicReference<Qnode> tail = new AtomicReference<Qnode>(null);
  ThreadLocal<Qnode> myNode = new Qnode(); // per thread

  void lock() {
    qnolde.locked = true;
    Qnode pred = tail.getAndSet(myNode); // swap my node into
    queue
    while (pred.locked) {} // spin
  }

  void unlock() {
    myNode.locked = false;
    myNode = pred; // "reuse" predecessor's qnode (see book)
  }
}
```

CLH Lock

- Lock release affects only successor
- Does not depend on prior knowledge about number of threads
- FIFO Fairness
- But doesn't work (efficiently) for uncached NUMA architectures

NUMA Architectures

- **N** on-**U** niform-**M** emomory-**A** rchitecture
- Model: Flat shared memory, no caches (in most variants)
- Some memory regions faster accessible than others
- Spinning on remote memory is slow



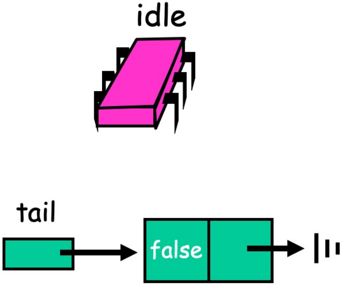
MCS Lock (by Mellor-Crummey and Scott)

- FIFO order
- Spin on local memory only
- Small, constant-size overhead

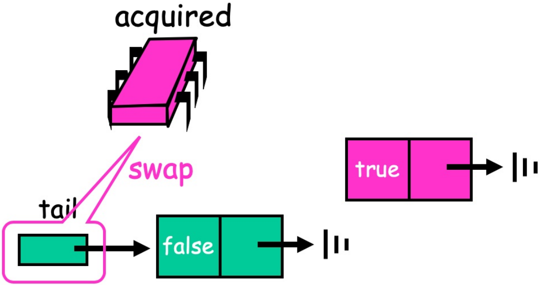
Idea:

- To acquire lock, place own Qnode at tail of list
- If it has a predecessor, modify predecessor's node to refer to own Qnode

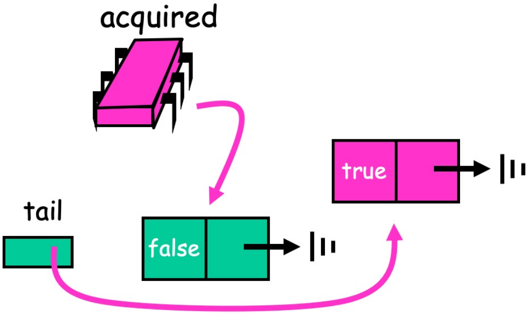
MCS Lock



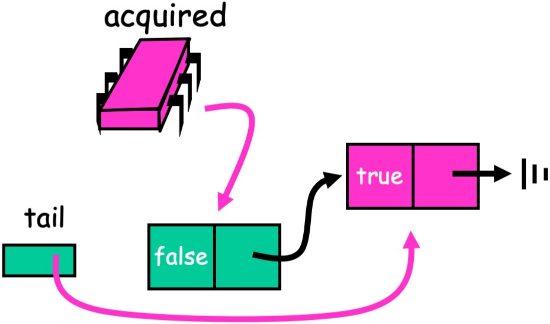
MCS Lock: Acquiring a lock



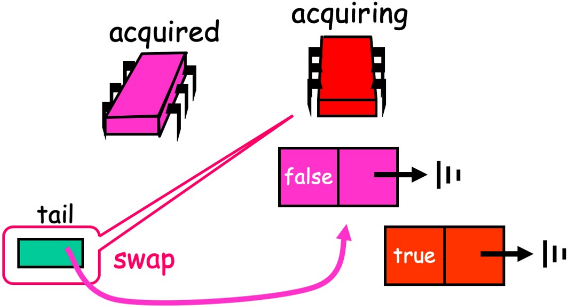
MCS Lock: Acquiring a lock



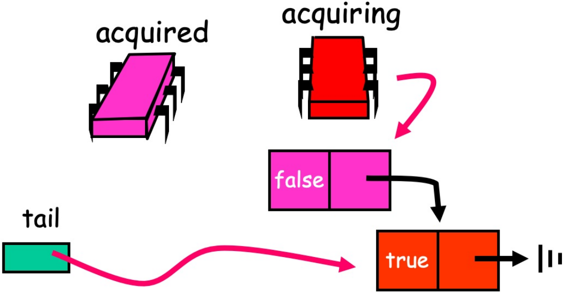
MCS Lock: Acquiring a lock



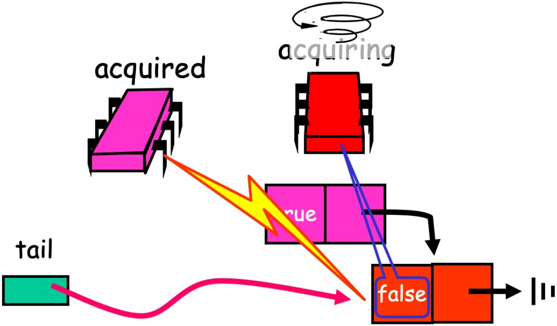
MCS Lock: Acquiring a lock



MCS Lock: Acquiring a lock



MCS Lock: Releasing a lock



MCS Lock

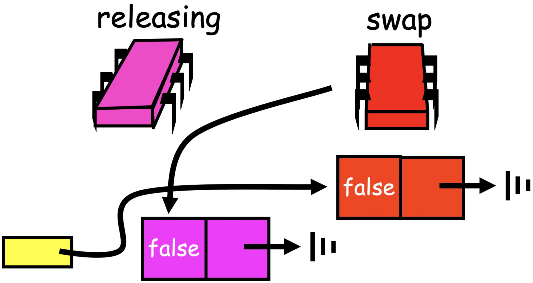
```
class Qnode {
  boolean locked = false;  // only reads/writes required
  Qnode   next   = null;
}
```

MCS Lock

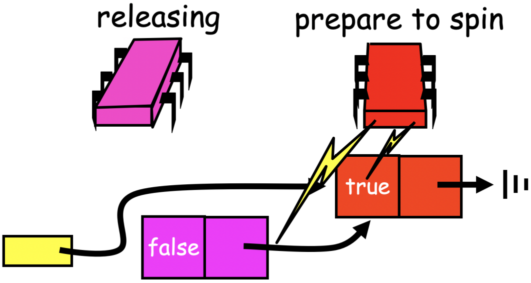
```
class MCSLock implements Lock {

  AtomicReference tail;
  ThreadLocal<Qnode> qnode = new Qnode();

  void lock() {
    // reset for reuse
    qnode.next = null;
    qnode.locked = false;

    // swap my node in
    Qnode pred = tail.getAndSet(qnode);

    if (pred != null) {
      // lock is taken, so set my status to wait
      qnode.locked = true;
      // tell predecessor where to find me
      pred.next = qnode;
      // spin on my node
      while (qnode.locked) {}
    }
  } ...
```

MCS Lock: Releasing

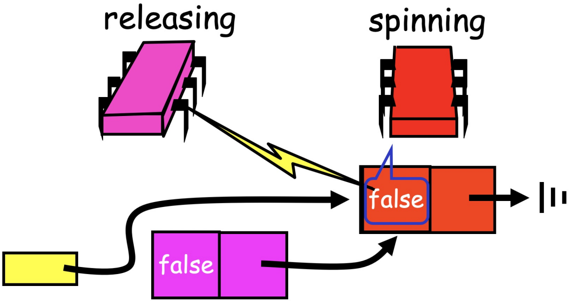
- Status of `qnode.next` indicates that other thread is active
- Need to wait for it to finish and start spinning



MCS Lock: Releasing



MCS Lock: Releasing



MCS Lock

```
  void unlock() {
    if (qnode.next == null) {
      // if really no thread waiting
      if (tail.compareAndSet(qnode, null)
        return;
      // otherwise, wait for successor to finish
      while (qnode.next == null) {}
    }
    // tell successor that it can start
    qnode.next.locked = false;
  }
}
```

Abortable Locks

- What if you want to give up waiting for a lock?
  - For example: timeout, transaction aborted by user, ...
- Simple for Backoff-Lock
  - Just return from `lock()` call
  - No cleanup, wait-free, immediate
- Problematic for Queue Locks
  - Can't just quit
  - Thread in line behind will starve

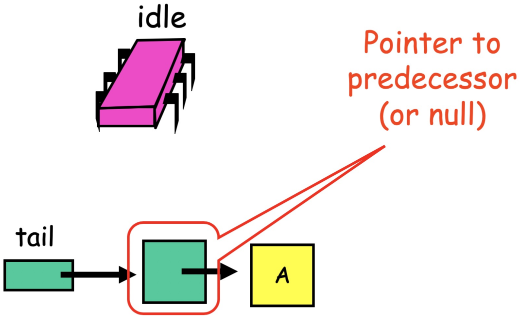
Abortable Locks

- What if you want to give up waiting for a lock?
  - For example: timeout, transaction aborted by user, . . .
- Simple for Backoff-Lock
  - Just return from `lock()` call
  - No cleanup, wait-free, immediate
- Problematic for Queue Locks
  - Can't just quit
  - Thread in line behind will starve

Idea: Let successor deal with the problem!

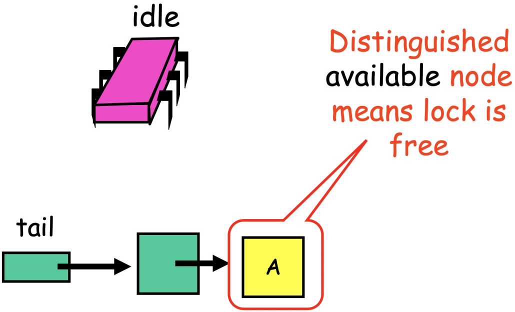
⇒ Abortable CLH Lock

Timeout Lock



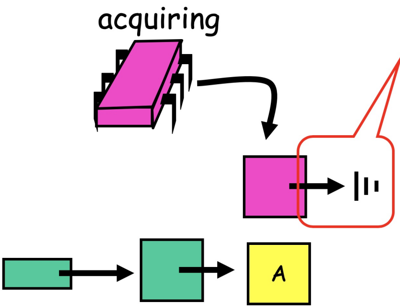
Pointer to predecessor (or null)

Timeout Lock: Acquire



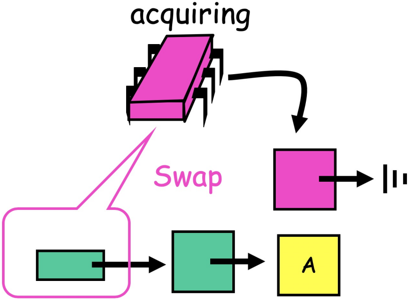
Distinguished available node means lock is free

Timeout Lock: Acquire

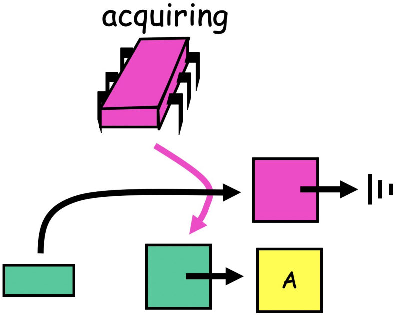


Null predecessor means lock not released or aborted

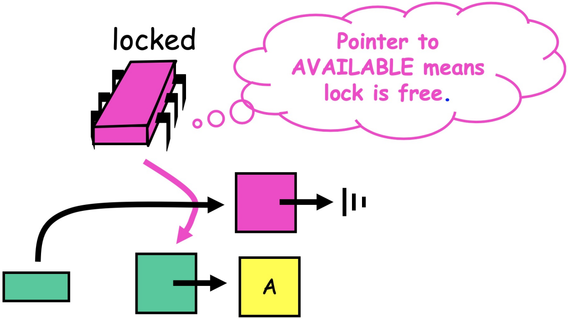
Timeout Lock: Acquire



Timeout Lock: Acquire



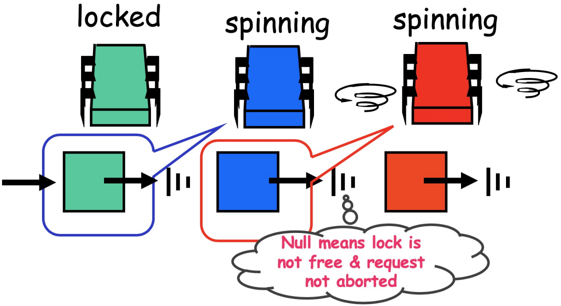
Timeout Lock: Acquire



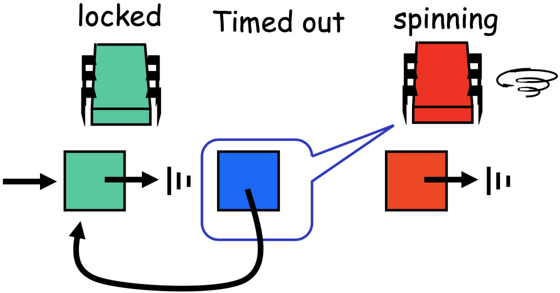
locked

Pointer to AVAILABLE means lock is free.

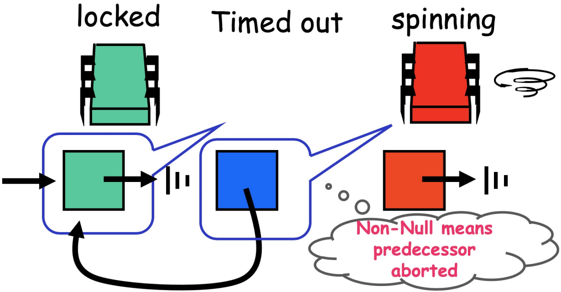
Timeout Lock: While waiting, ...



Timeout Lock: Thread times out



Timeout Lock: Thread times out



Timeout Locks: Implementation

```
class TOLock {
  static Qnode AVAILABLE = new Qnode(); // signifies free lock
  AtomicReference<Qnode> tail;
  ThreadLocal<Qnode> myNode; // per thread

  // Return value indicates success
  boolean lock(long timeout) {
    // Initialize node
    Qnode qnode = new Qnode();
    myNode = qnode;
    qnode.prev = null;

    // swap with tail
    Qnode myPred = tail.getAndSet(qnode);

    // if predecessor absent or released, we are done
    if (myPred == null || myPred.prev == AVAILABLE) {
      return true;
   }
...
```

Timeout Locks

```
...
  // Keep trying for a while
  long start = now();
  while (now()- start < timeout) {
    // Spin on predecessor's prev field
    Qnode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
      // predecessor released lock
      return true;
    } else if (predPred != null) {
      // predecessor aborted, we advance in queue
      myPred = predPred;
    }
  }
...
```

Timeout Locks

```
...
// In case timeout happened, we waited long enough
if (!tail.compareAndSet(qnode, myPred)){
  // If CAS fails, tell successor about my predecessor
    qnode.prev = myPred;
  }
  // If CAS succeeds, no successor, nothing to do
  return false;
}
```

Timeout Locks

```
void unlock() {
  Qnode qnode = myNode.get();
  if (!tail.compareAndSet(qnode, null)) {
    // If CAS failed: there is successor
    // Notify successor that it can enter
    qnode.prev = AVAILABLE;
  }
  // If CAS succeeds: no successor waiting
  // Set tail to null, no clean up
}
```

Summary: One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS, ToLock ...
- Each one better than others in some way
- There is no *one* solution
- Decision really depends on:
  - the application
  - the hardware
  - which properties are important