Practice: Small Systems

Part 2, Chapter 3

Roger Wattenhofer

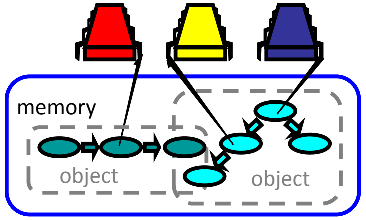
ETH Zurich – Distributed Computing – www.disco.ethz.ch

Overview

- Introduction
- Spin Locks
  - Test-and-Set & Test-and-Test-and-Set
  - Backoff lock
  - Queue locks
- Concurrent Linked List
  - Fine-grained synchronization
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization
- Hashing
  - Fine-grained locking
  - Recursive split ordering

Concurrent Computation

- We started with…
- Multiple threads
  - Sometimes called processes
- Single shared memory
- Objects live in memory
- Unpredictable asynchronous delays

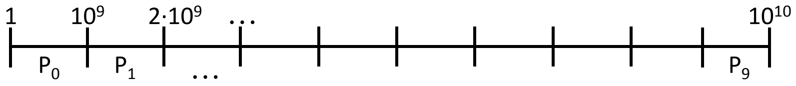


- In the previous chapters, we focused on fault-tolerance
  - We discussed theoretical results
  - We discussed practical solutions with a focus on efficiency
- In this chapter, we focus on efficient concurrent computation!
  - Focus on asynchrony and not on explicit failures

Example: Parallel Primality Testing

- Challenge
  - Print all primes from 1 to 10^{10}
- Given
  - Ten-core multiprocessor
  - One thread per processor
- Goal
  - Get ten-fold speedup (or close)
- Naïve Approach
  - Split the work evenly
  - Each thread tests range of 10^9

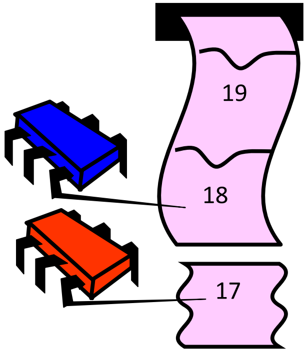
Problems with this approach?



Issues

- Higher ranges have fewer primes
- Yet larger numbers are harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need dynamic load balancing

- Better approach
  - Shared counter!
  - Each thread takes a number



Procedure Executed at each Thread

```
Counter counter = new Counter(1);

void primePrint() {
    long j = 0;
    while(j < 10^10) {
        j = counter.getAndIncrement();
        if(isPrime(j))
            print(j);
    }
}
```

Shared counter object

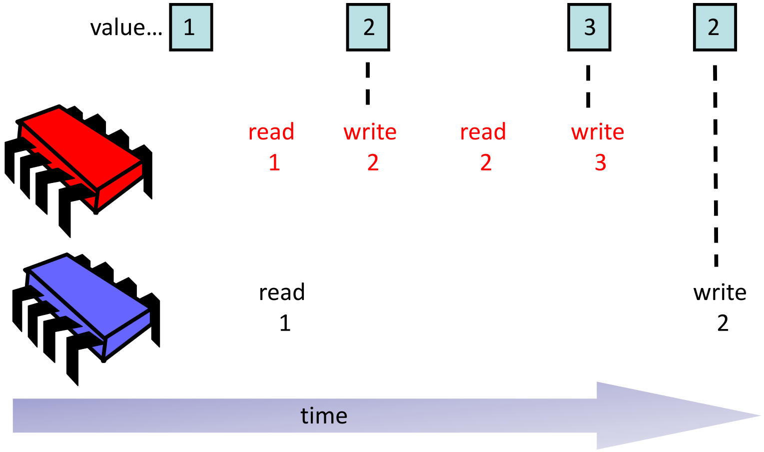
Increment counter & test if return value is prime

Counter Implementation

```
public class Counter {

    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
```

What's the problem with this implementation?

Problem



Counter Implementation

```
public class Counter {

    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}
```

These steps must be atomic!

Recall: We can use **Read-Modify-Write (RMW)** instructions!

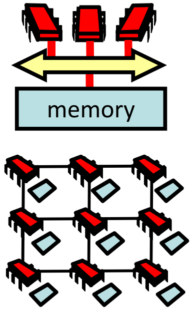
We have to guarantee **mutual exclusion**

Model

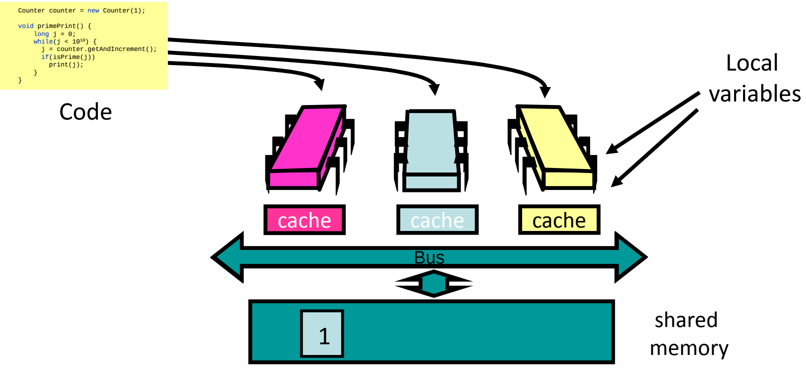
- The model in this part is slightly more complicated
  - However, we still focus on principles

I.e., multiprocessors

- What remains the same?
  - Multiple instruction multiple data (MIMD) architecture
  - Each thread/process has its own code and local variables
  - There is a shared memory that all threads can access
- What is new?
  - Typically, communication runs over a shared bus (alternatively, there may be several channels)
  - Communication contention
  - Communication latency
  - Each thread has a local cache



Model: Where Things Reside



```
Counter counter = new Counter(1);

void primePrint() {
    long j = 0;
    while(j < 10¹⁰) {
      j = counter.getAndIncrement();
      if(isPrime(j))
        print(j);
    }
}
```

E.g., the shared counter is here

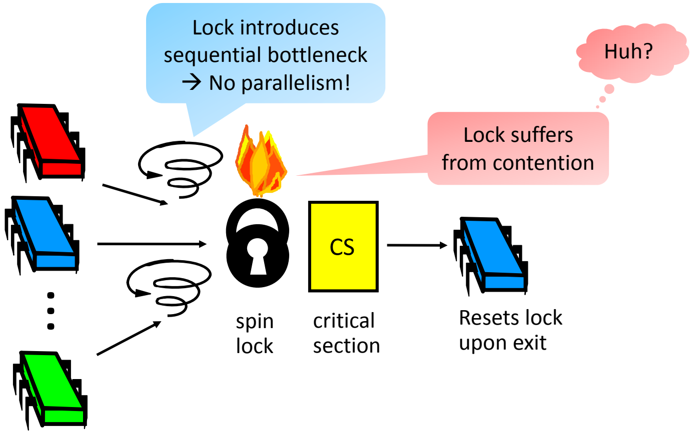
Revisiting Mutual Exclusion

- We need mutual exclusion for our counter
- We are now going to study mutual exclusion from a different angle
  - Focus on performance, not just correctness and progress
- We will 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
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

} Our focus

Basic Spin-Lock



Lock introduces sequential bottleneck
→ No parallelism!

Huh?

Lock suffers from contention

CS

spin lock

critical section

Resets lock upon exit

Reminder: Test&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**
- Also known as “getAndSet”

Reminder: Test&Set

```
public class AtomicBoolean {
  private boolean value;

  public synchronized boolean getAndSet() {
    boolean prior = this.value;
    this.value = true;
    return prior;
  }

}
```

java.util.concurrent.atomic

Get current value and set value to true

Test&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&Set Lock

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

  public void lock() {
    while (state.getAndSet()) {}
  }

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

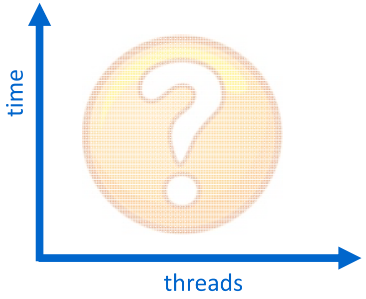
Lock state is AtomicBoolean

Keep trying until lock acquired

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?



Test&Test&Set Locks

- How can we improve TAS?
- A crazy idea: Test before you test and set!

- Lurking stage
  - Wait until lock “looks” free
  - Spin while read returns true (i.e., the lock is taken)
- Pouncing state
  - As soon as lock “looks” available
  - Read returns false (i.e., the lock is free)
  - Call TAS to acquire the lock
  - If TAS loses, go back to lurking

Test&Test&Set Lock

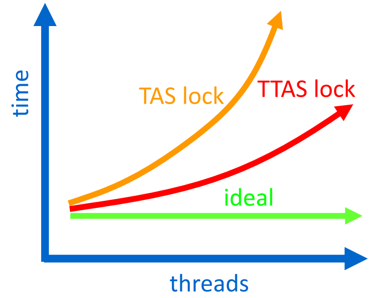
```
public class TTASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);

  public void lock() {
    while (true) {
      while(state.get()) {}
      if(!state.getAndSet())
        return;
    }
  }

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

Wait until lock looks free

Then try to acquire it

Performance

- Both TAS and TTAS do the same thing (in our old model)
- So, we would expect basically the same results

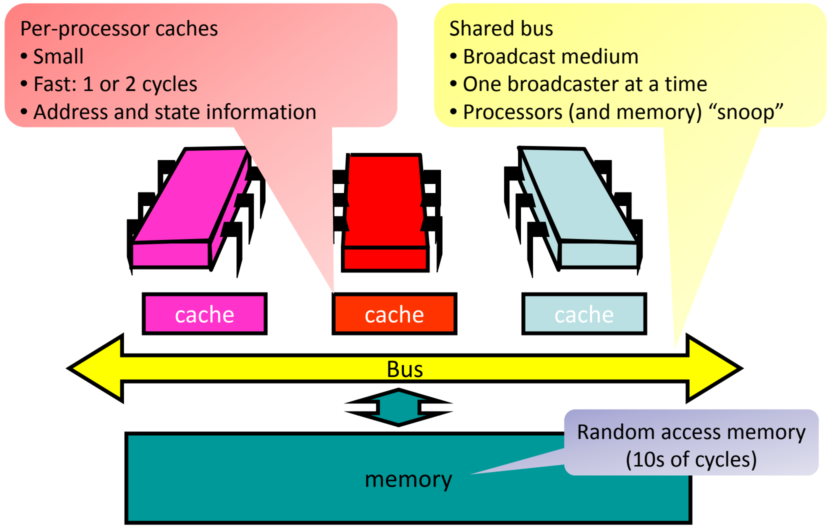


- Why is TTAS so much better than TAS? Why are both far from ideal?

Opinion

- TAS & TTAS locks
  - are provably the same (in our old model)
  - except they aren't (in field tests)
- Obviously, it must have something to do with the model…
- Let's take a closer look at our new model and try to find a reasonable explanation!

Bus-Based Architectures



Per-processor caches
- Small
- Fast: 1 or 2 cycles
- Address and state information

Shared bus
- Broadcast medium
- One broadcaster at a time
- Processors (and memory) "snoop"

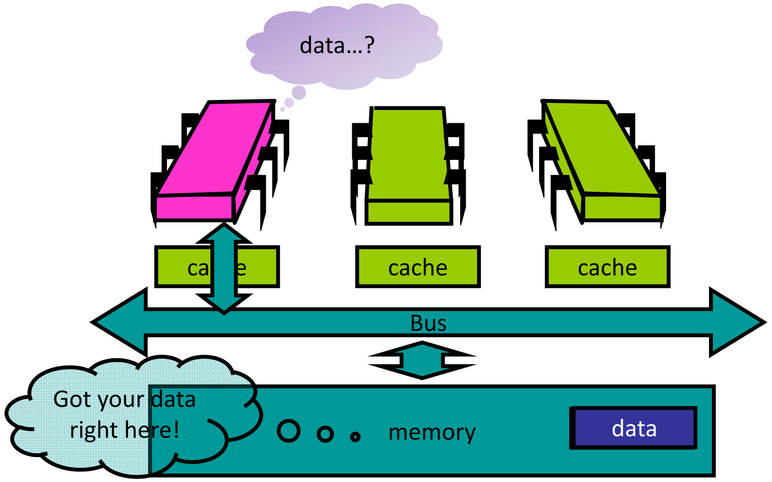
Random access memory (10s of cycles)

Jargon Watch

- Load request
  - When a thread wants to access data, it issues a load request
- Cache hit
  - The thread found the data in its own cache
- Cache miss
  - The data is not found in the cache
  - The thread has to get the data from memory

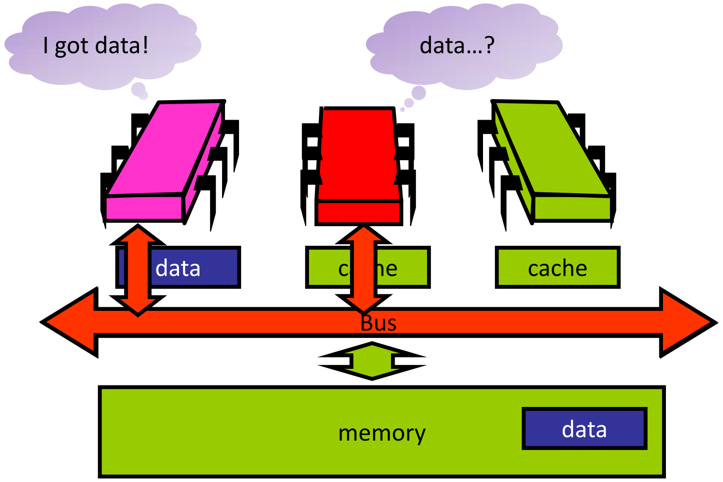
Load Request

- Thread issues load request and memory responds



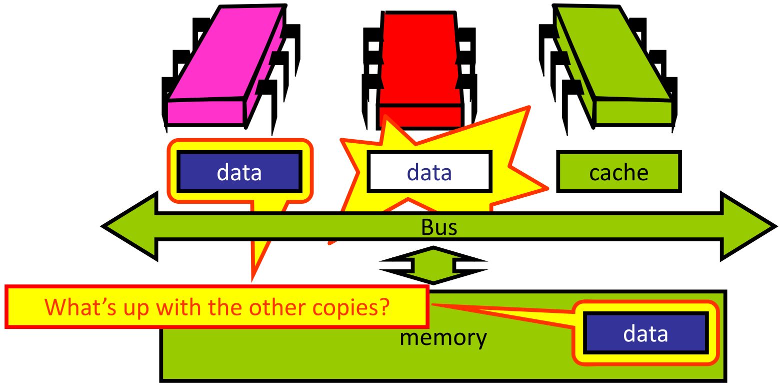
Another Load Request

- Another thread wants to access the same data. Get a copy from the cache!



Modify Cached Data

- Both threads now have the data in their cache
- What happens if the red thread now modifies the 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?

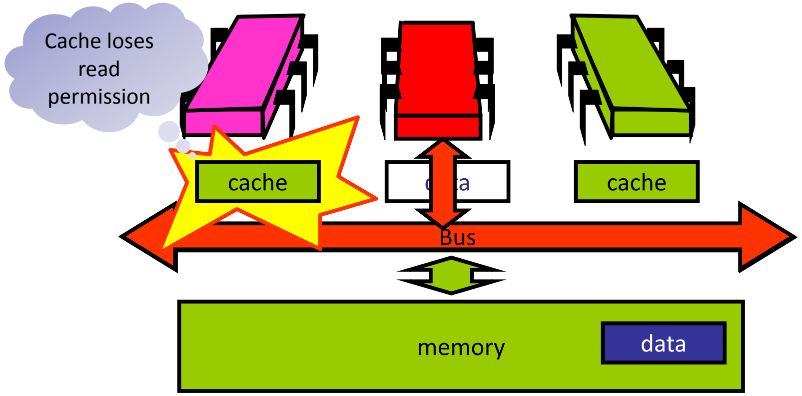
Write-Back Caches

- 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 …

- Cache entry has three states:
  - Invalid: contains raw bits
  - Valid: I can read but I can’t write
  - Dirty: Data has been modified
    - Intercept other load requests
    - Write back to memory before reusing cache

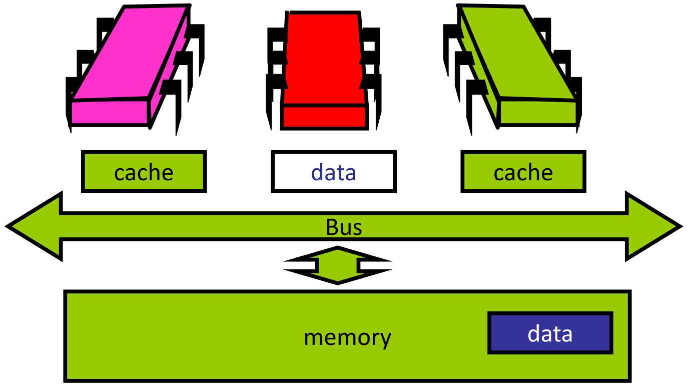
Invalidate

- Let’s rewind back to the moment when the red processor updates its cached data
- It broadcasts an invalidation message → Other processor invalidates its cache!



Invalidate

- Memory provides data only if not present in any cache, so there is no need to change it now (this is an expensive operation!)
- Reading is not a problem → The threads get the data from the red process



Mutual Exclusion

- What do we want to optimize?
  1. Minimize the bus bandwidth that the spinning threads use
  2. Minimize the lock acquire/release latency
  3. Minimize the latency to acquire the lock if the lock is idle

TAS vs. TTAS

- TAS invalidates cache lines
- Spinners
  - Always go to bus
- Thread wants to release lock
  - delayed behind spinners!!!

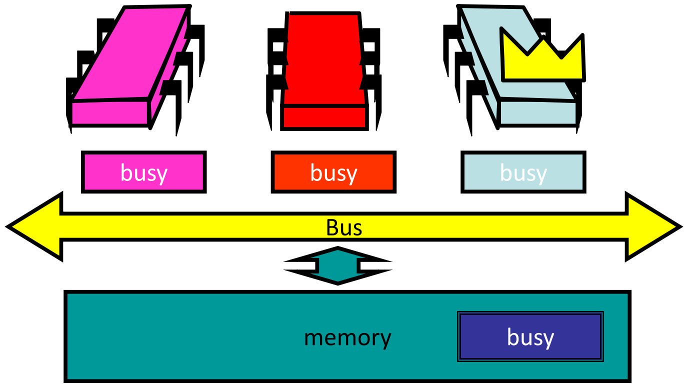
This is why TAS performs so poorly…

- TTAS waits until lock “looks” free
  - Spin on local cache
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm …

Huh?

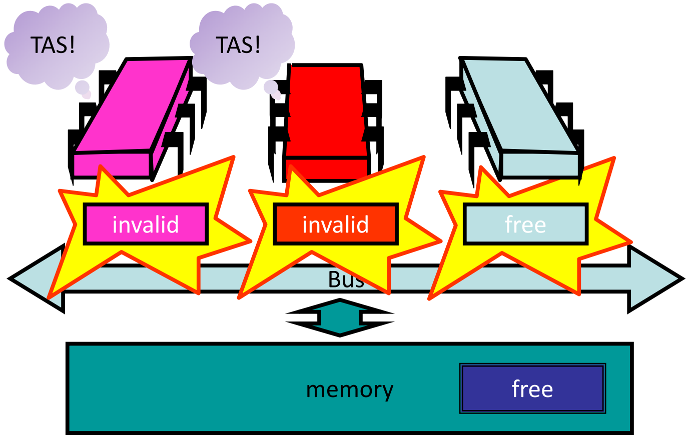
Local Spinning while Lock is Busy

- While the lock is held, all contenders spin in their caches, rereading cached data without causing any bus traffic

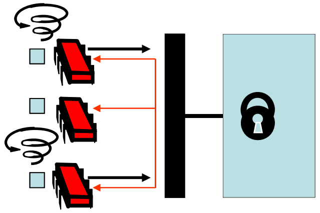


On Release

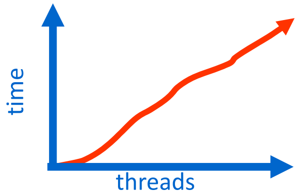
- The lock is released. All spinners take a cache miss and call Test&Set!



Time to Quiescence

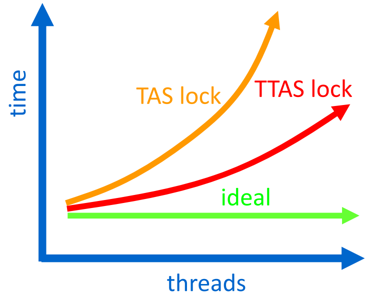


- Every process experiences a cache miss
  - All state.get() satisfied sequentially
- Every process does TAS
  - Caches of other processes are invalidated
- Eventual quiescence (“silence”) after acquiring the lock
- The time to quiescence increases linearly with the number of processors for a bus architecture!



Mystery Explained

- Now we understand why the TTAS lock performs much better than the TAS lock, but still much worse than an ideal lock!

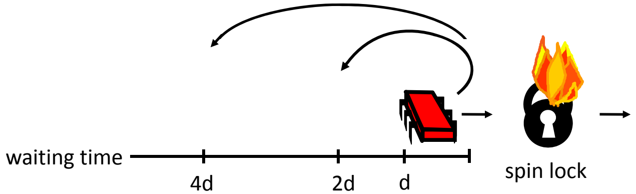


- How can we do better?

Introduce Delay

- If the lock looks free, but I fail to get it, there must be lots of contention
- It's better to back off than to collide again!

- Example: Exponential Backoff
- Each subsequent failure doubles expected waiting time



Exponential Backoff Lock

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

  public void lock() {
    int delay = MIN_DELAY;
    while (true) {
      while(state.get()) {}
      if (!lock.getAndSet())
        return;
      sleep(random() % delay);
      if (delay < MAX_DELAY)
        delay = 2 * delay;
    }
  }

  // unlock() remains the same

}
```

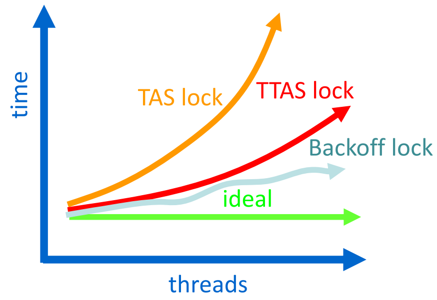
Fix minimum delay

Back off for random duration

Double maximum delay until an upper bound is reached

Backoff Lock: Performance

- The backoff lock outperforms the TTAS lock!
- But it is still not ideal…



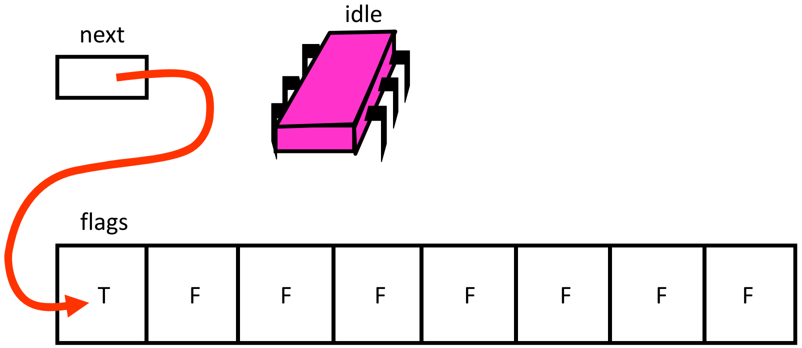
Backoff Lock: Evaluation

- Good
  - Easy to implement
  - Beats TTAS lock
- Bad
  - Must choose parameters carefully
  - Not portable across platforms

- How can we do better?
- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others

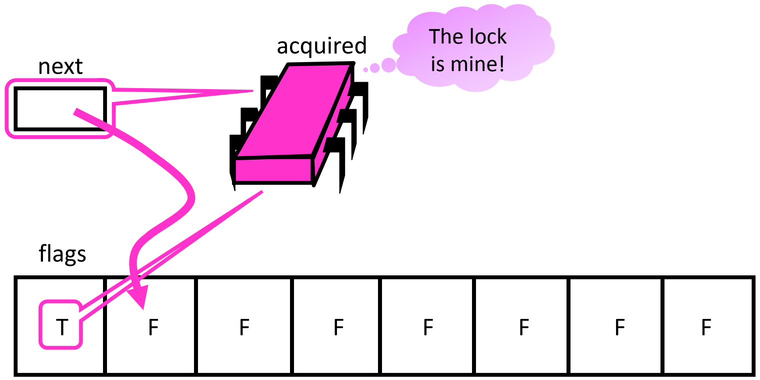
ALock: Initially

- The Anderson queue lock (ALock) is an array-based queue lock
- Threads share an atomic tail field (called next)



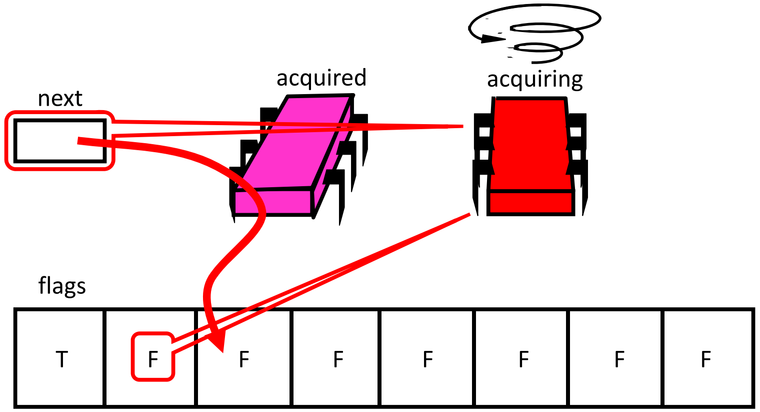
ALock: Acquiring the Lock

- To acquire the lock, each thread atomically increments the tail field
- If the flag is true, the lock is acquired
- Otherwise, spin until the flag is true



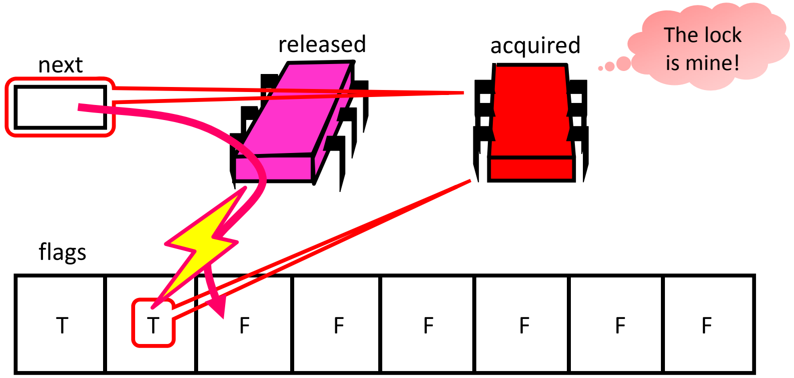
ALock: Contention

- If another thread wants to acquire the lock, it applies get&increment
- The thread spins because the flag is false



ALock: Releasing the Lock

- The first thread releases the lock by setting the next slot to true
- The second thread notices the change and gets the lock



ALock

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

  public void lock() {
    mySlot = next.getAndIncrement();
    while (!flags[mySlot % n]) {}
    flags[mySlot % n] = false;
  }

  public void unlock() {
    flags[(mySlot+1) % n] = true;
  }
}
```

One flag per thread

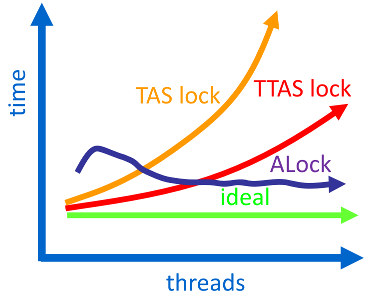
Thread-local variable

Take the next slot

Tell next thread to go

ALock: Performance

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

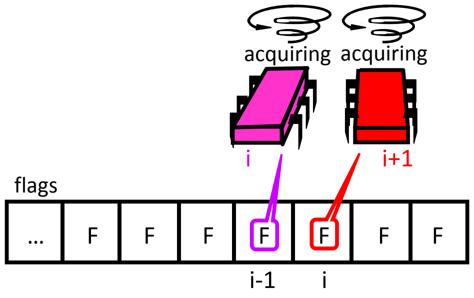


ALock: Evaluation

- Good
  - First truly scalable lock
  - Simple, easy to implement
- Bad
  - One bit per thread
  - Unknown number of threads?

ALock: Alternative Technique

- The threads could update own flag and spin on their predecessor's flag



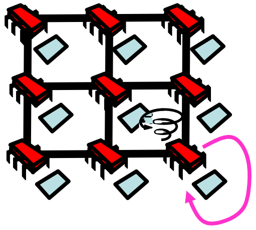
- This is basically what the CLH lock does, but using a linked list instead of an array
- Is this a good idea?

Not discussed in this lecture

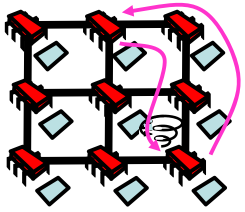
NUMA Architectures

- **N**on-**U**niform **M**emory **A**rchitecture
- Illusion
  - Flat shared memory
- Truth
  - No caches (sometimes)
  - Some memory regions faster than others

Spinning on local memory is fast:



Spinning on remote memory is slow:

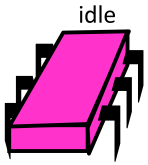


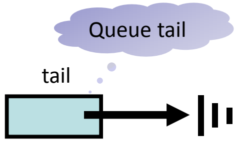
MCS Lock

- Idea
  - Use a linked list instead of an array
    → Small, constant-sized space
  - Spin on own flag, just like the Anderson queue lock
- The space usage
  - L = number of locks
  - N = number of threads
- of the Anderson lock is O(LN)
- of the MCS lock is O(L+N)

MCS Lock: Initially

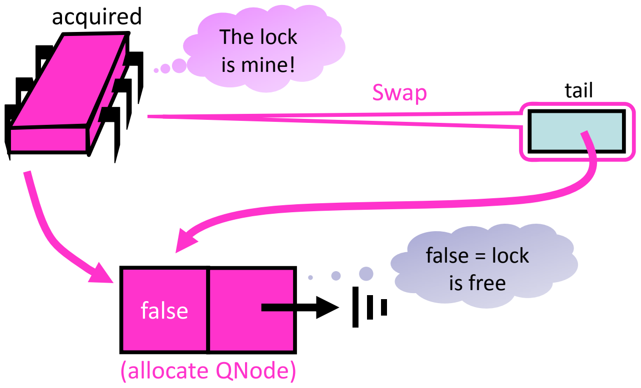
- The lock is represented as a linked list of QNodes, one per thread
- The tail of the queue is shared among all threads





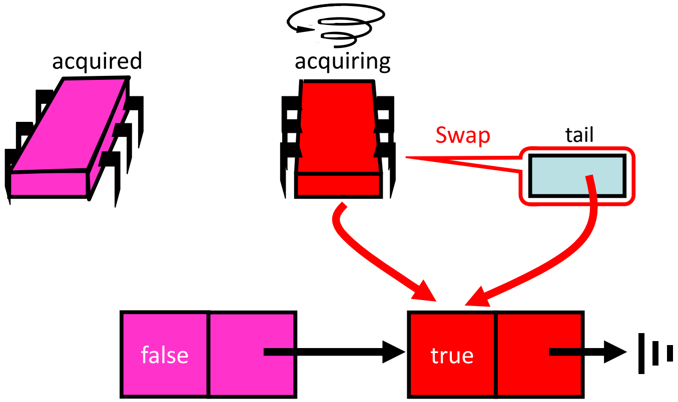
MCS Lock: Acquiring the Lock

- To acquire the lock, the thread places its QNode at the tail of the list by swapping the tail to its QNode
- If there is no predecessor, the thread acquires the lock



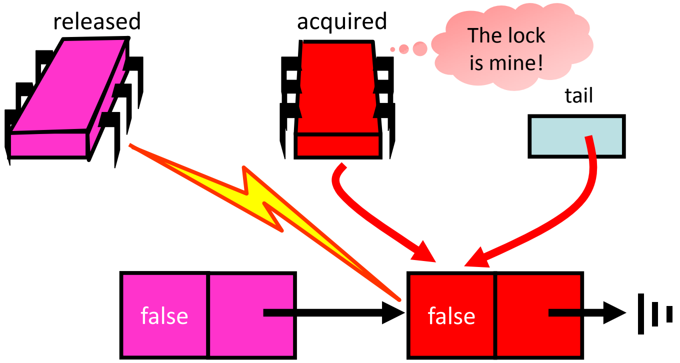
MCS Lock: Contention

- If another thread wants to acquire the lock, it again applies swap
- The thread spins on its own QNode because there is a predecessor



MCS Lock: Releasing the Lock

- The first thread releases the lock by setting its successor’s QNode to false



MCS Queue Lock

```
public class QNode {
  boolean locked = false;
  QNode next = null;
}
```

MCS Queue Lock

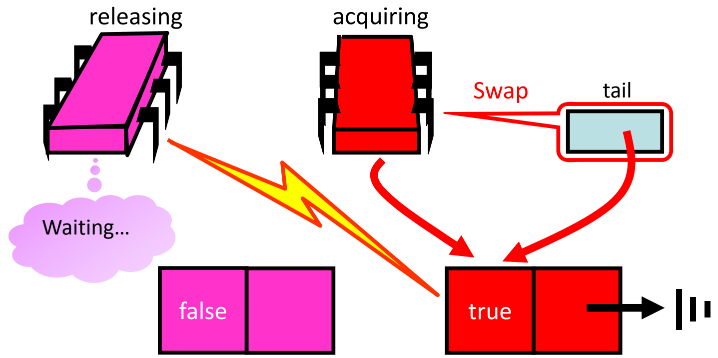
```
public class MCSLock implements Lock {
  AtomicReference tail;

  public void lock() {
    QNode qnode = new QNode();
    QNode pred = tail.getAndSet(qnode);
    if (pred != null) {
      qnode.locked = true;
      pred.next = qnode;
      while (qnode.locked) {}
    }
  }

  ...
```

Add my node to the tail

Fix if queue was non-empty

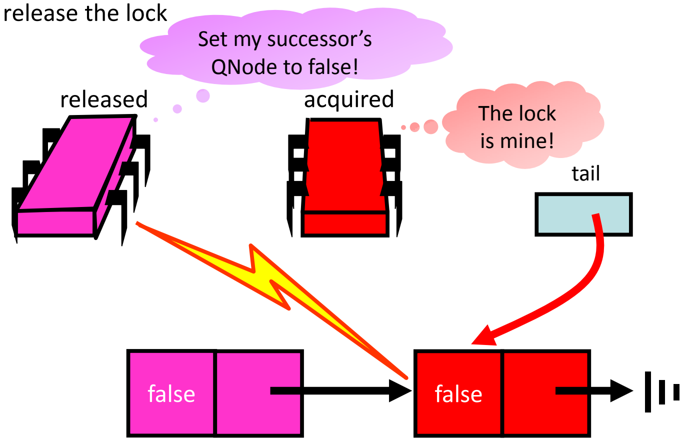
MCS Lock: Unlocking

- If there is a successor, unlock it. But, be cautious!
- Even though a QNode does not have a successor, the purple thread knows that another thread is active because tail does not point to its QNode!



MCS Lock: Unlocking Explained

- As soon as the pointer to the successor is set, the purple thread can release the lock



MCS Queue Lock

```
  ...

  public void unlock() {
    if (qnode.next == null) {
      if (tail.CAS(qnode, null)
        return;
      while (qnode.next == null) {}
    }
    qnode.next.locked = false;
  }
}
```

Missing successor?

If really no successor, return

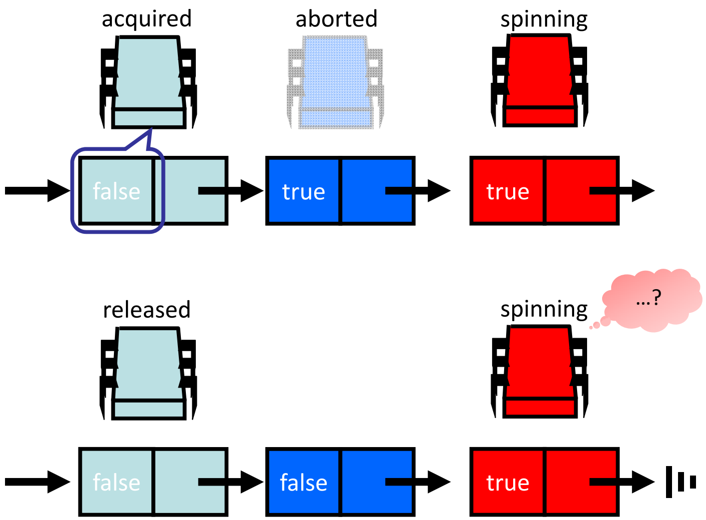
Otherwise, wait for successor to catch up

Pass lock to successor

Abortable Locks

- What if you want to give up waiting for a lock?
- For example
  - Time-out
  - Database transaction aborted by user
- Back-off Lock
  - Aborting is trivial: Just return from lock() call!
  - Extra benefit: No cleaning up, wait-free, immediate return
- Queue Locks
  - Can't just quit: Thread in line behind will starve
  - Need a graceful way out...

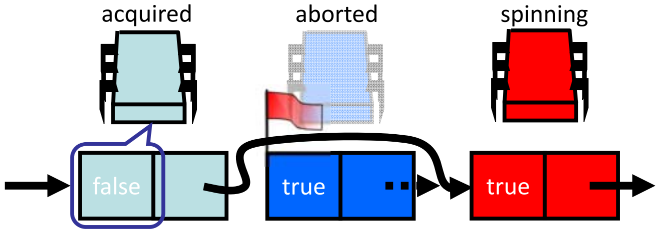
Problem with Queue Locks



Abortable MCS Lock

- A mechanism is required to recognize and remove aborted threads
  - A thread can set a flag indicating that it aborted
  - The predecessor can test if the flag is set
  - If the flag is set, its new successor is the successor's successor
  - How can we handle concurrent aborts? This is not discussed in this lecture

Spinning on remote object...?!

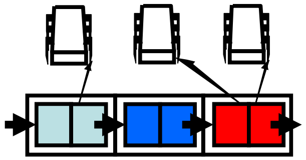


Composite Locks

- Queue locks have many advantages
  - FIFO fairness, fast lock release, low contention

  but require non-trivial protocols to handle aborts (and recycling of nodes)
- Backoff locks support trivial time-out protocols

  but are not scalable and may have slow lock release times
- A composite lock combines the best of both approaches!
- Short fixed-sized array of lock nodes
- Threads randomly pick a node and try to acquire it
- Use backoff mechanism to acquire a node
- Nodes build a queue
- Use a queue lock mechanism to acquire the lock



One Lock To Rule Them All?

- TTAS+Backoff, MCS, Abortable MCS…
- Each better than others in some way
- There is not a single best solution
- Lock we pick really depends on
  - the application
  - the hardware
  - which properties are important

Handling Multiple Threads

- Adding threads should not lower the throughput
  - Contention effects can mostly be fixed by Queue locks

- Adding threads should increase throughput
  - Not possible if the code is inherently sequential
  - Surprising things are parallelizable!

- How can we guarantee consistency if there are many threads?

Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using queue locks
  - Mostly easy to reason about
  - This is the standard Java model (**synchronized** blocks and methods)

- Problem: Sequential bottleneck
  - Threads “stand in line”
  - Adding more threads does not improve throughput
  - We even struggle to keep it from getting worse…

- So why do we even use a multiprocessor?
  - Well, some applications are inherently parallel…
  - We focus on exploiting non-trivial parallelism

Exploiting Parallelism

- We will now talk about four “patterns”
  - Bag of tricks …
  - Methods that work more than once …

- The goal of these patterns are
  - Allow concurrent access
  - If there are more threads, the throughput increases!

Pattern #1: Fine-Grained Synchronization

- Instead of using a single lock split the concurrent object into independently-synchronized components
- Methods conflict when they access
  - The same component
  - At the same time

Pattern #2: Optimistic Synchronization

- Assume that nobody else wants to access your part of the concurrent object
- Search for the specific part that you want to lock without locking any other part on the way
- If you find it, try to lock it and perform your operations
  - If you don't get the lock, start over!
- Advantage
  - Usually cheaper than always assuming that there may be a conflict due to a concurrent access

---

Pattern #3: Lazy Synchronization

- Postpone hard work!
- Removing components is tricky
  - Either remove the object physically
  - Or logically: Only mark component to be deleted

Pattern #4: Lock-Free Synchronization

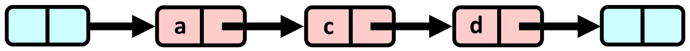
- Don't use locks at all!
  - Use compareAndSet() & other RMW operations!
- Advantages
  - No scheduler assumptions/support
- Disadvantages
  - Complex
  - Sometimes high overhead

Illustration of Patterns

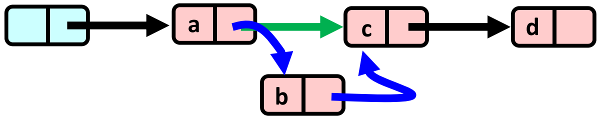
- In the following, we will illustrate these patterns using a list-based set
  - Common application
  - Building block for other apps
- A set is an collection of items
  - No duplicates
- The operations that we want to allow on the set are
  - `add(x)` puts x into the set
  - `remove(x)` takes x out of the set
  - `contains(x)` tests if x is in the set

The List-Based Set

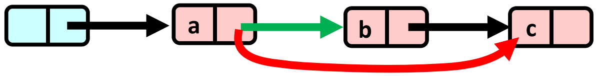
- We assume that there are sentinel nodes at the beginning and end of the linked list



- Add node b:

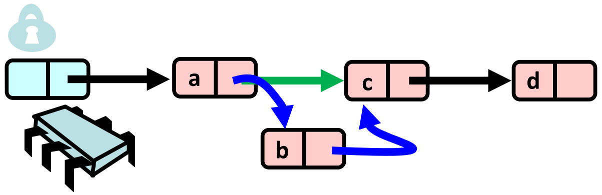


- Remove node b:



Coarse-Grained Locking

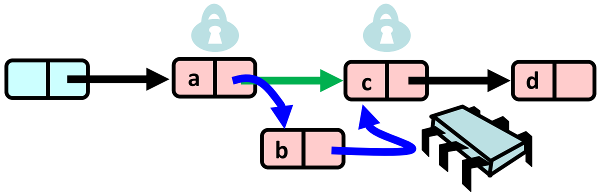
- A simple solution is to lock the entire list for each operation
  - E.g., by locking the first sentinel



- Simple and clearly correct!
- Works poorly with contention…

Fine-Grained Locking

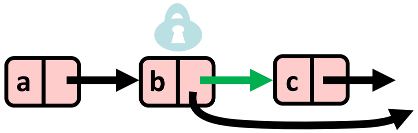
- Split object (list) into pieces (nodes)
  - Each piece (each node in the list) has its own lock
  - Methods that work on disjoint pieces need not exclude each other



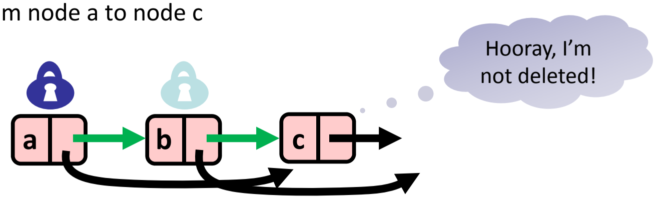
- Hand-over-hand locking: Use two locks when traversing the list
  - Why two locks?

Problem with One Lock

- Assume that we want to delete node c
- We lock node b and set its next pointer to the node after c



- Another thread may concurrently delete node b by setting the next pointer from node a to node c

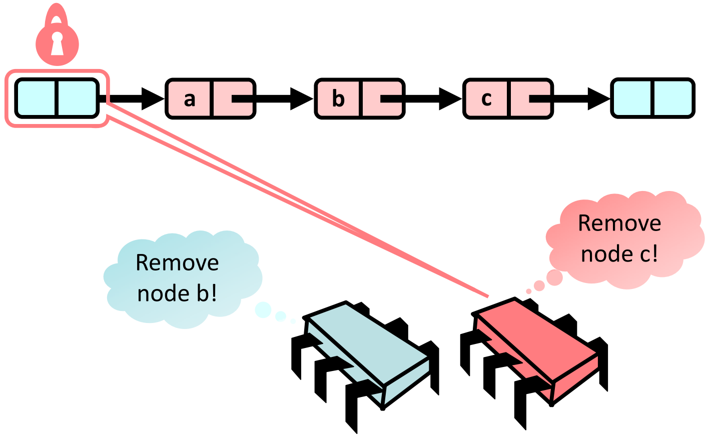


Insight

- If a node is locked, no one can delete the node's *successor*
- If a thread locks
  - the node to be deleted
  - and also its predecessor
- then it works!
- That's why we (have to) use two locks!

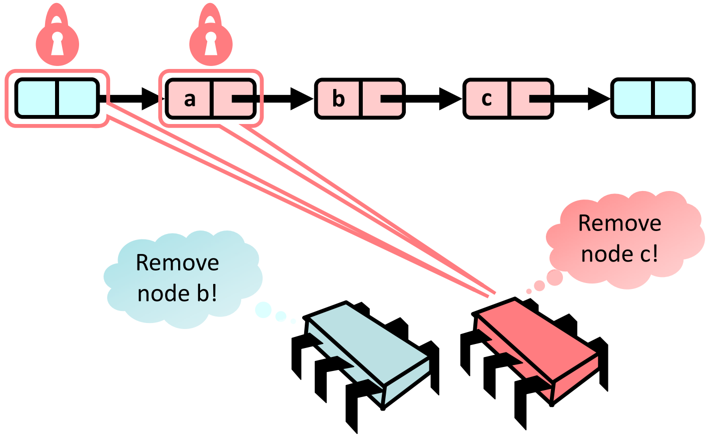
Hand-Over-Hand Locking: Removing Nodes

- Assume that two threads want to remove the nodes b and c
- One thread acquires the lock to the sentinel, the other has to wait



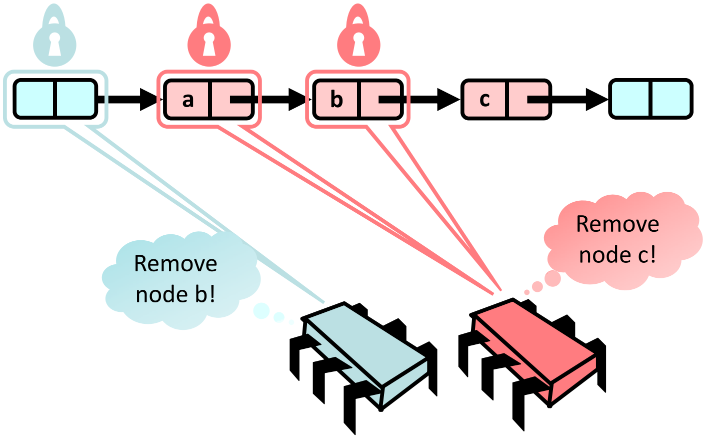
Hand-Over-Hand Locking: Removing Nodes

- The same thread that acquired the sentinel lock can then lock the next node



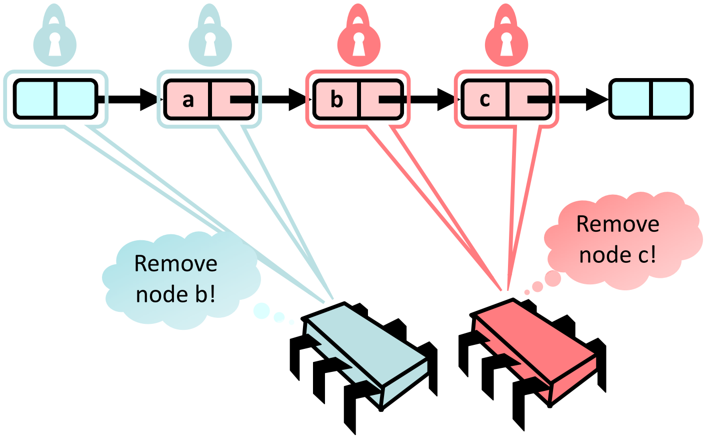
Hand-Over-Hand Locking: Removing Nodes

- Before locking node b, the sentinel lock is released
- The other thread can now acquire the sentinel lock



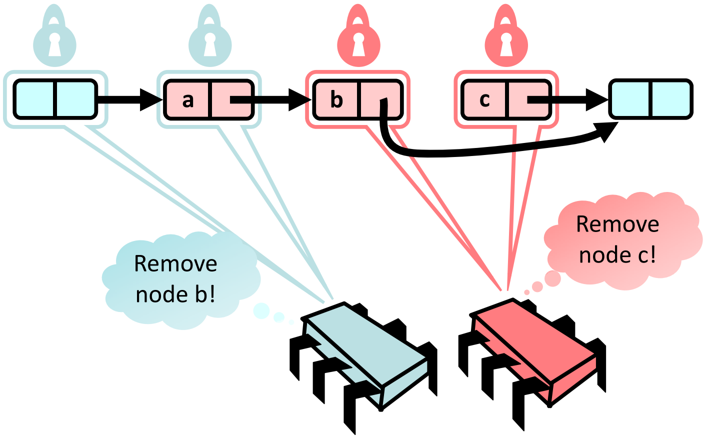
Hand-Over-Hand Locking: Removing Nodes

- Before locking node c, the lock of node a is released
- The other thread can now lock node a



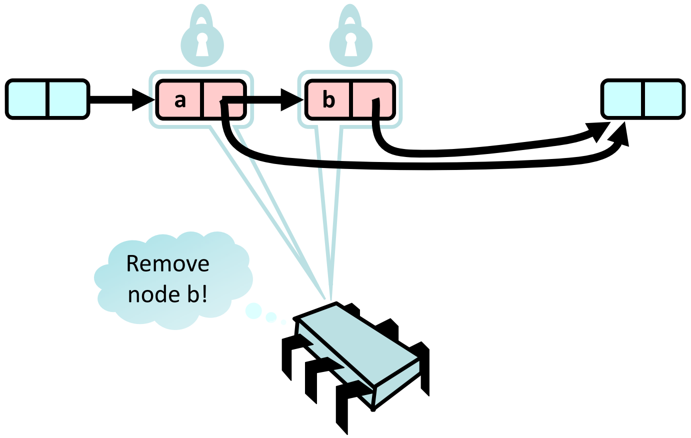
Hand-Over-Hand Locking: Removing Nodes

- Node c can now be removed
- Afterwards, the two locks are released



Hand-Over-Hand Locking: Removing Nodes

- The other thread can now lock node b and remove it



List Node

```
public class Node {
  public T item;
  public int key;
  public Node next;
}
```

Item of interest

Usually a hash code

Reference to next node

Remove Method

```
public boolean remove(Item item) {
  int key = item.hashCode();
  Node pred, curr;
  try {
    pred = this.head;
    pred.lock();
    curr = pred.next;
    curr.lock();

    ...

  } finally {
    curr.unlock();
    pred.unlock();
  }
}
```

Start at the head and lock it

Lock the current node

Traverse the list and remove the item

On the next slide!

Make sure that the locks are released

Remove Method

```
    while (curr.key <= key) {
      if (item == curr.item) {
        pred.next = curr.next;
          return true;
      }
      pred.unlock();
      pred = curr;
      curr = curr.next;
      curr.lock();
    }
    return false;
```

Search key range

If item found, remove the node

Unlock pred and lock the next node

Return false if the element is not present

Why does this work?

- To remove node e
  - Node e must be locked
  - Node e’s predecessor must be locked
- Therefore, if you lock a node
  - It can’t be removed
  - And neither can its successor

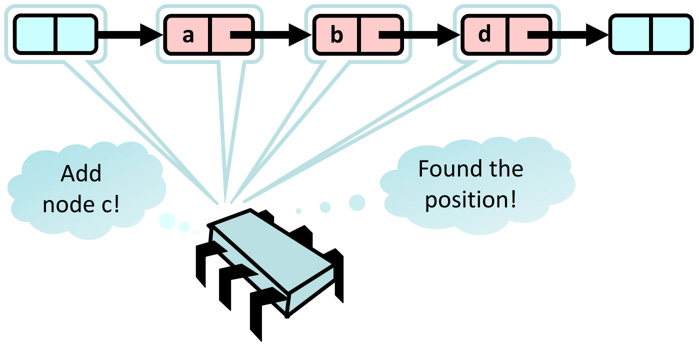
- To add node e
  - Must lock predecessor
  - Must lock successor
- Neither can be deleted
  - Is the successor lock actually required?

Drawbacks

- Hand-over-hand locking is sometimes better than coarse-grained lock
  - Threads can traverse in parallel
  - Sometimes, it’s worse!
- However, it’s certainly not ideal
  - Inefficient because many locks must be acquired and released
- How can we do better?

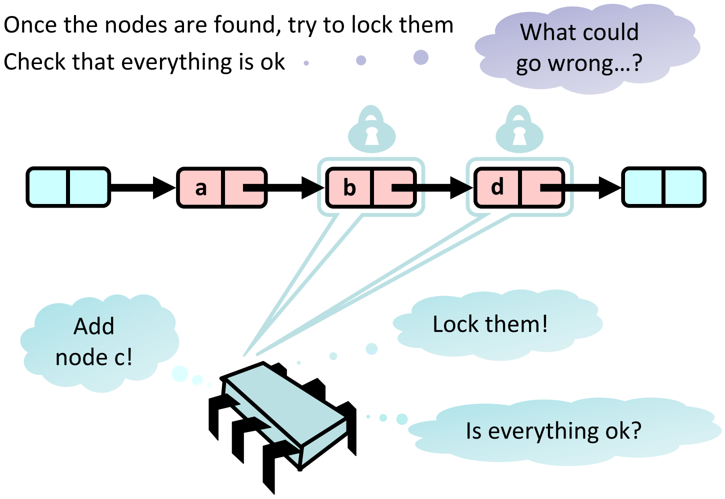
Optimistic Synchronization

- Traverse the list without locking!



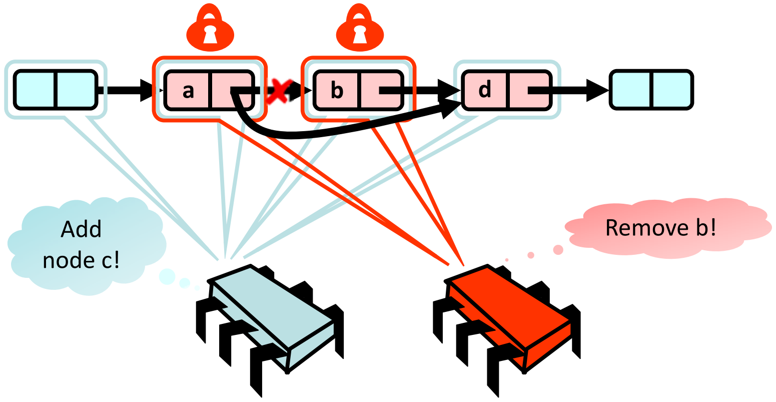
Optimistic Synchronization: Traverse without Locking

- Once the nodes are found, try to lock them
- Check that everything is ok



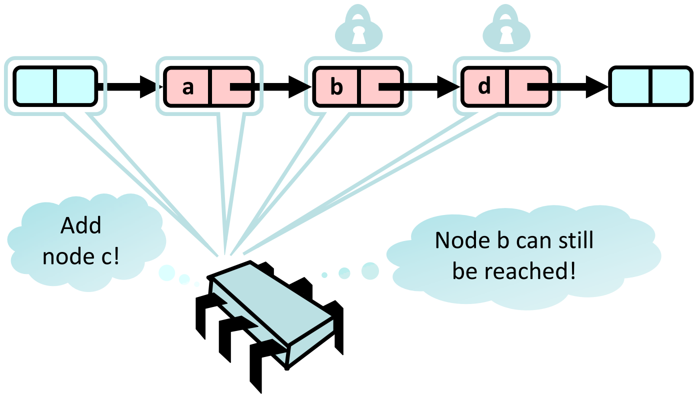
Optimistic Synchronization: What Could Go Wrong?

- Another thread may lock nodes a and b and remove b before node c is added → If the pointer from node b is set to node c, then node c is not added to the list!



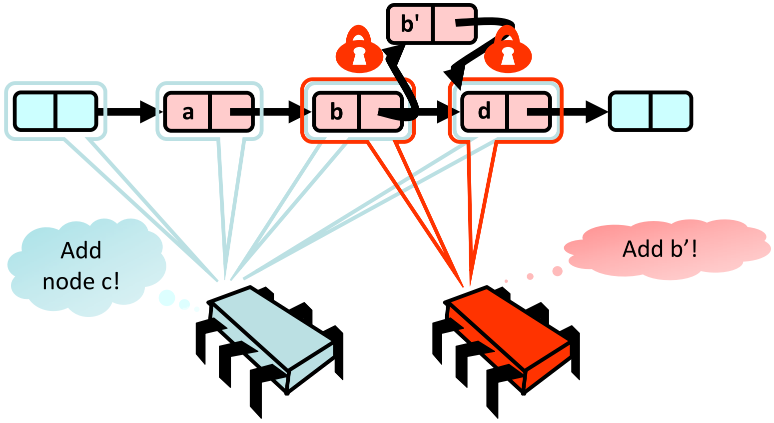
Optimistic Synchronization: Validation #1

- How can this be fixed?
- After locking node b and node d, traverse the list again to verify that b is still reachable



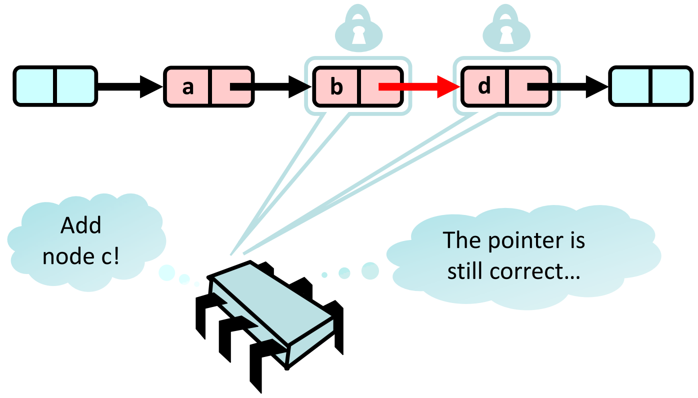
Optimistic Synchronization: What Else Could Go Wrong?

- Another thread may lock nodes b and d and add a node b’ before node c is added → By adding node c, the addition of node b’ is undone!



Optimistic Synchronization: Validation #2

- How can this be fixed?
- After locking node b and node d, also check that node b still points to node d!



Optimistic Synchronization: Validation

```
private boolean validate(Node pred,Node curr) {
  Node node = head;
  while (node.key <= pred.key) {
    if (node == pred)
      return pred.next == curr;
    node = node.next;
  }
  return false;
}
```

If pred is reached, test if the successor is curr

Predecessor not reachable

Optimistic Synchronization: Remove

```
private boolean remove(Item item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = pred.next;
    while (curr.key <= key) {
      if (item == curr.item)
        break;
      pred = curr;
      curr = curr.next;
    }
    ...
```

Retry on synchronization conflict

Stop if we find the item

Optimistic Synchronization: Remove

```
      ...
      try {
        pred.lock(); curr.lock();
        if (validate(pred,curr)) {
          if (curr.item == item) {
            pred.next = curr.next;
            return true;
          } else {
            return false;
          }
        }
      } finally {
        pred.unlock();
        curr.unlock();
      }
    }
  }
```

Lock both nodes

Check for synchronization conflicts

Remove node if target found

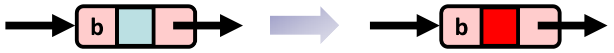
Always unlock the nodes

Optimistic Synchronization

- Why is this correct?
  - If nodes b and c are both locked, node b still accessible, and node c still the successor of node b, then neither b nor c will be deleted by another thread
  - This means that it’s ok to delete node c!
- Why is it good to use optimistic synchronization?
  - Limited hot-spots: no contention on traversals
  - Less lock acquisitions and releases
- When is it good to use optimistic synchronization?
  - When the cost of scanning twice without locks is less than the cost of scanning once with locks
- Can we do better?
  - It would be better to traverse the list only once…

Lazy Synchronization

- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”
- How can we remove nodes “lazily”?
  - First perform a logical delete: Mark current node as removed (new!)



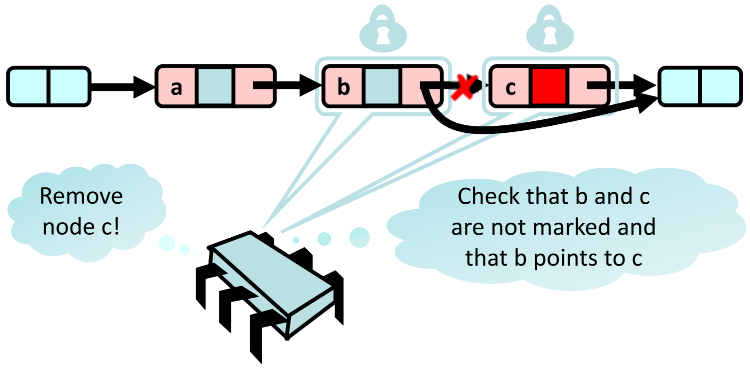
  - Then perform a physical delete: Redirect predecessor’s next (as before)

Lazy Synchronization

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn’t slow down other method calls…
- Note that we must still lock pred and curr nodes!
- How does validation work?
  - Check that neither pred nor curr are marked
  - Check that pred points to curr

Lazy Synchronization

- Traverse the list and then try to lock the two nodes
- Validate!
- Then, mark node c and change the predecessor’s next pointer



Remove node c!

Check that b and c are not marked and that b points to c

Lazy Synchronization: Validation

```
private boolean validate(Node pred,Node curr) {
 return !pred.marked && !curr.marked &&
 pred.next == curr);
}
```

Nodes are not logically removed

Predecessor still points to current

Lazy Synchronization: Remove

```
public boolean remove(Item item) {
 int key = item.hashCode();
 while (true) {
  Node pred = this.head;
  Node curr = pred.next;
  while (curr.key <= key) {
   if (item == curr.item)
    break;
   pred = curr;
   curr = curr.next;
  }
  ...
```

This is the same as before!

Lazy Synchronization: Remove

```
  ...
  try {
   pred.lock(); curr.lock();
   if (validate(pred,curr)) {
    if (curr.item == item) {
     curr.marked = true;
     pred.next = curr.next;
     return true;
    } else {
     return false;
    }
   }
  } finally {
   pred.unlock();
   curr.unlock();
  }
 }
}
```

Check for synchronization conflicts

If the target is found, mark the node and remove it

Lazy Synchronization: Contains

```
public boolean contains(Item item) {
 int key = item.hashCode();
 Node curr = this.head;
 while (curr.key < key) {
  curr = curr.next
 }
 return curr.key == key && !curr.marked;
```

Traverse without locking (nodes may have been removed)

Is the element present and not marked?

Evaluation

- Good
  - The list is traversed only once without locking
  - Note that contains() doesn't lock at all!
  - This is nice because typically contains() is called much more often than add() or remove()
  - Uncontended calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays

- Traffic jam?
  - If one thread gets the lock and experiences a cache miss/page fault, every other thread that needs the lock is stuck!
  - We need to trust the scheduler….

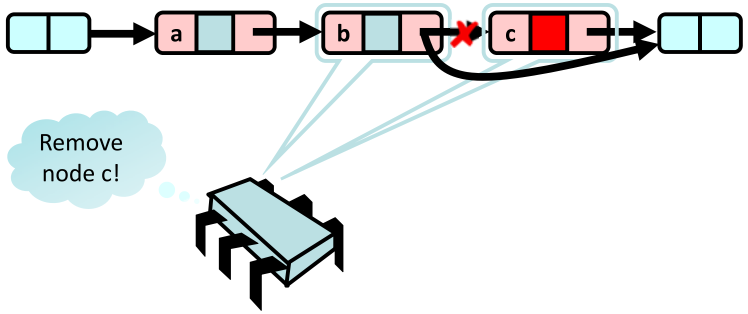
Reminder: Lock-Free Data Structures

- If we want to guarantee that some thread will eventually complete a method call, even if other threads may halt at malicious times, then the implementation cannot use locks!

- Next logical step: Eliminate locking entirely!
- Obviously, we must use some sort of RMW method
- Let's use compareAndSet() (CAS)!

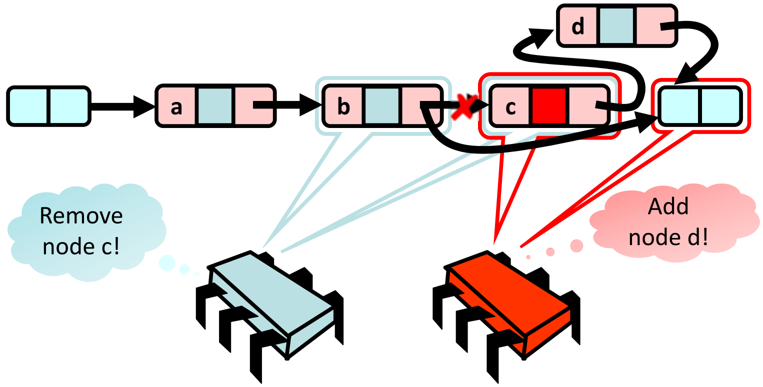
Remove Using CAS

- First, remove the node logically (i.e., mark it)
- Then, use CAS to change the next pointer
- Does this work…?



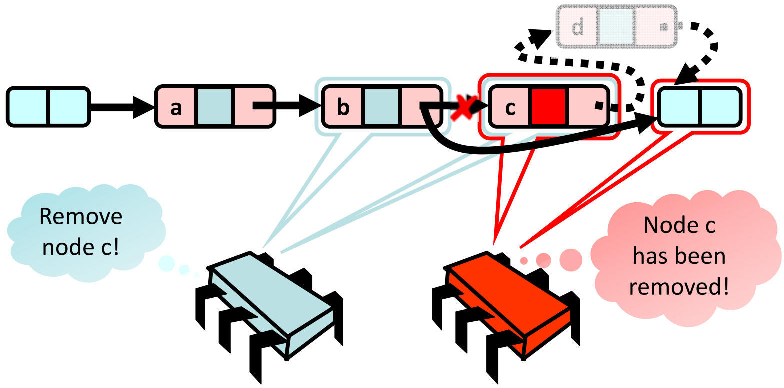
Remove Using CAS: Problem

- Unfortunately, this doesn’t work!
- Another node d may be added before node c is physically removed
- As a result, node d is not added to the list…



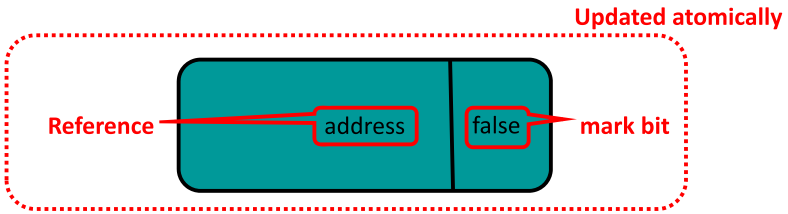
Solution

- Mark bit and next pointer are “CASed together”
- This atomic operation ensures that no node can cause a conflict by adding (or removing) a node at the same position in the list



Solution

- Such an operation is called an atomic markable reference
  - Atomically update the mark bit and redirect the predecessor’s next pointer
- In Java, there’s an AtomicMarkableReference class
  - In the package Java.util.concurrent.atomic package



Changing State

```
private Object ref;
private boolean mark;

public synchronized boolean compareAndSet(
Object expectedRef, Object updateRef,
boolean expectedMark, boolean updateMark) {

  if (ref == expectedRef && mark == expectedMark){
    ref = updateRef;
    mark = updateMark;
  }
}
```

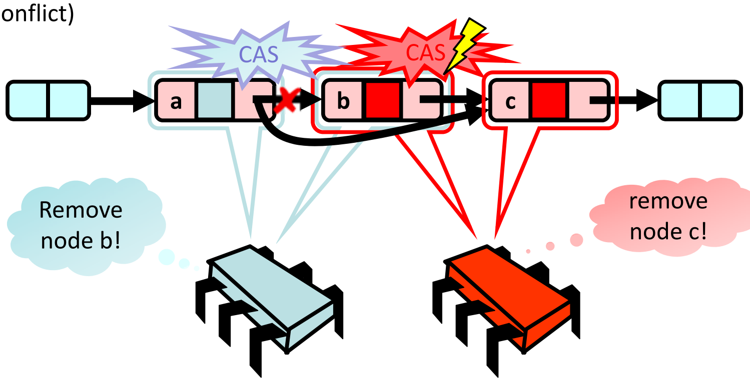
The reference to the next Object and the mark bit

If the reference and the mark are as expected, update them atomically

---

Removing a Node

- If two threads want to delete the nodes b and c, both b and c are marked
- The CAS of the red thread fails because node b is marked!
- (If node b is yet not marked, then b is removed first and there is no conflict)

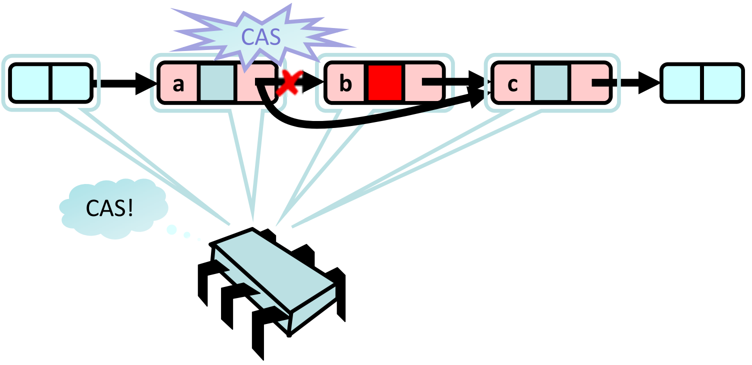


Traversing the List

- Question: What do you do when you find a “logically” deleted node in your path when you’re traversing the list?

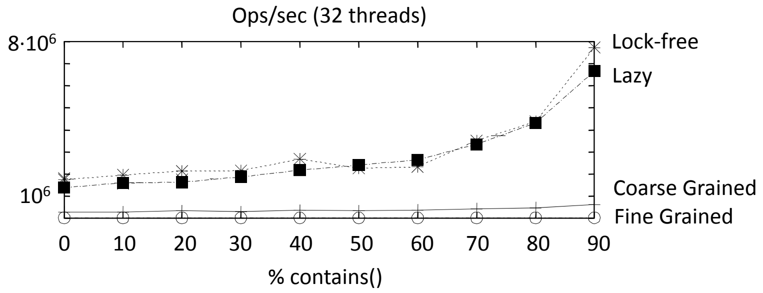
Lock-Free Traversal

- If a logically deleted node is encountered, CAS the predecessor’s next field and proceed (repeat as needed)



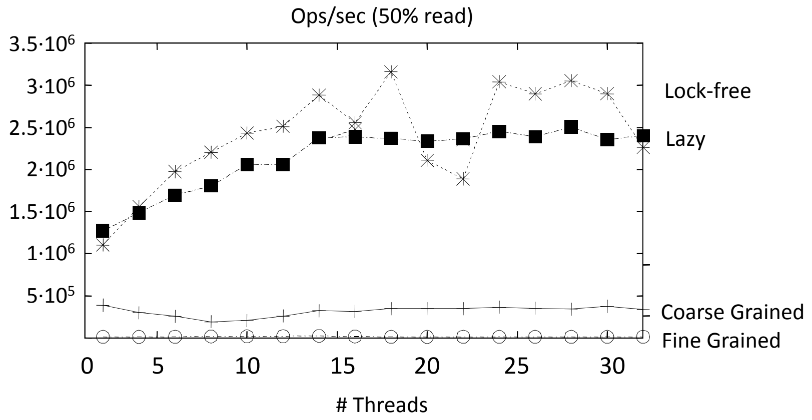
Performance

- The throughput of the presented techniques has been measured for a varying percentage of contains() method calls
  - Using a benchmark on a 16 node shared memory machine



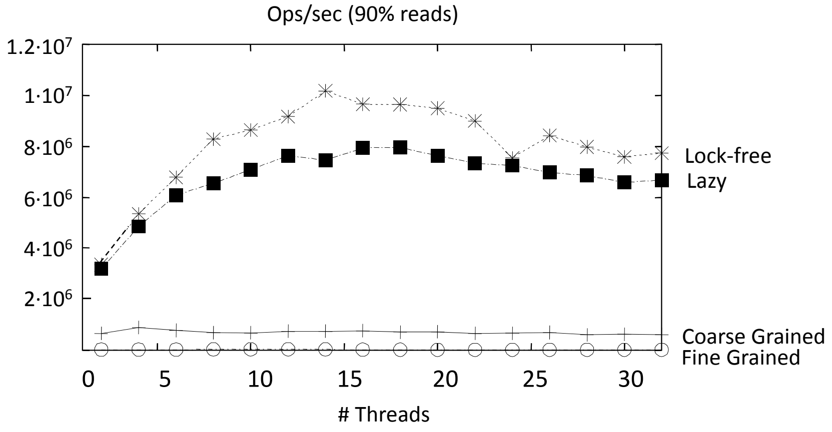
Low Ratio of contains()

- If the ratio of contains() is low, the lock-free linked list and the linked list with lazy synchronization perform well even if there are many threads



High Ratio of contains()

- If the ratio of contains() is high, again both the lock-free linked list and the linked list with lazy synchronization perform well even if there are many threads



“To Lock or Not to Lock”

- Locking vs. non-blocking: Extremist views on both sides
- It is nobler to compromise by combining locking and non-blocking techniques
  - Example: Linked list with lazy synchronization combines blocking add() and remove() and a non-blocking contains()
  - Blocking/non-blocking is a property of a method

Linear-Time Set Methods

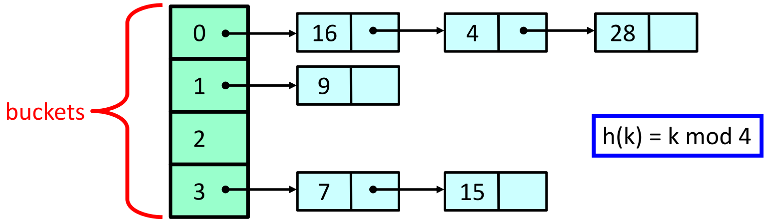
- We looked at a number of ways to make highly-concurrent list-based sets
  - Fine-grained locks
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization
- What's not so great?
  - add(), remove(), contains() take time linear in the set size
- We want constant-time methods! How…?
  - At least on average…

Hashing

- A hash function maps the items to integers
  - h: items → integers
- Uniformly distributed
  - Different items "most likely" have different hash values
- In Java there is a hashCode() method

Sequential Hash Map

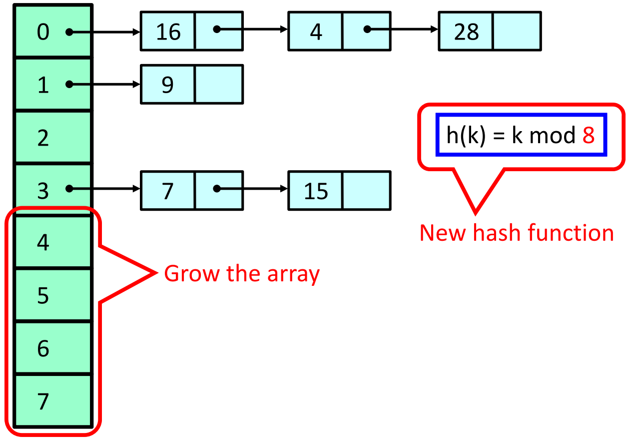
- The hash table is implemented as an array of buckets, each pointing to a list of items



- Problem: If many items are added, the lists get long → Inefficient lookups!
- Solution: Resize!

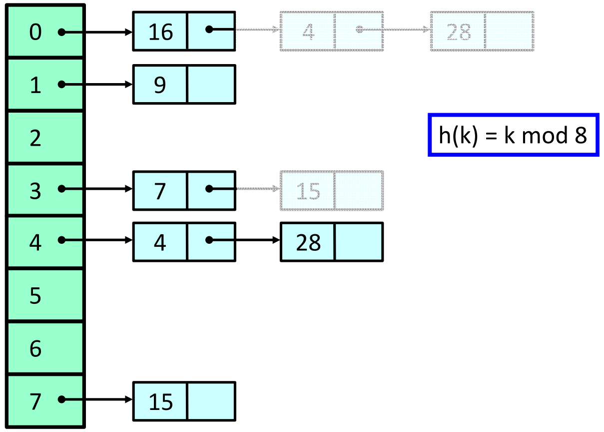
Resizing

- The array size is doubled and the hash function adjusted



Resizing

- Some items have to be moved to different buckets!



Hash Sets

- A hash set implements a set object
  - Collection of items, no duplicates
  - add(), remove(), contains() methods
- More coding ahead!

Simple Hash Set

```
public class SimpleHashSet {
  protected LockFreeList[] table;

  public SimpleHashSet(int capacity) {
    table = new LockFreeList[capacity];
    for (int i = 0; i < capacity; i++)
      table[i] = new LockFreeList();
    }

  public boolean add(Object key) {
    int hash = key.hashCode() % table.length;
    return table[hash].add(key);
```

Array of lock-free lists

Initial size

Initialization

Use hash of object to pick a bucket and call bucket's add() method

---

Simple Hash Set: Evaluation

- We just saw a
  - Simple
  - Lock-free
  - Concurrent

  hash-based set implementation
- But we don't know how to resize…
- Is Resizing really necessary?
  - Yes, since constant-time method calls require constant-length buckets and a table size proportional to the set size
  - As the set grows, we must be able to resize

Set Method Mix

- Typical load
  - 90% contains()
  - 9% add ()
  - 1% remove()
- Growing is important, shrinking not so much

- When do we resize?
- There are many reasonable policies, e.g., pick a threshold on the number of items in a bucket
- Global threshold
  - When, e.g., ≥ ¼ buckets exceed this value
- Bucket threshold
  - When any bucket exceeds this value

Coarse-Grained Locking

- If there are concurrent accesses, how can we safely resize the array?

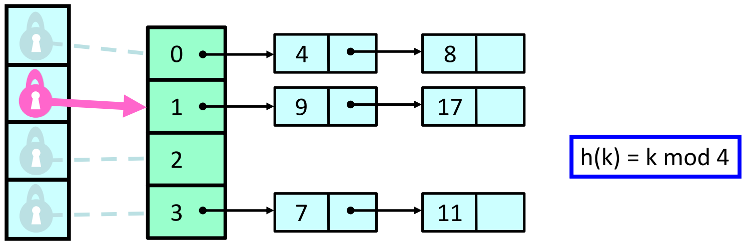
- As with the linked list, a straightforward solution is to use coarse-grained locking: lock the entire array!

- This is very simple and correct
- However, we again get a sequential bottleneck…

- How about fine-grained locking?

Fine-Grained Locking

- Each lock is associated with one bucket



- After acquiring the lock of the list, insert the item in the list!

Fine-Grained Locking: Resizing

- Acquire all locks in ascending order and make sure that the table reference didn't change between resize decision and lock acquisition!

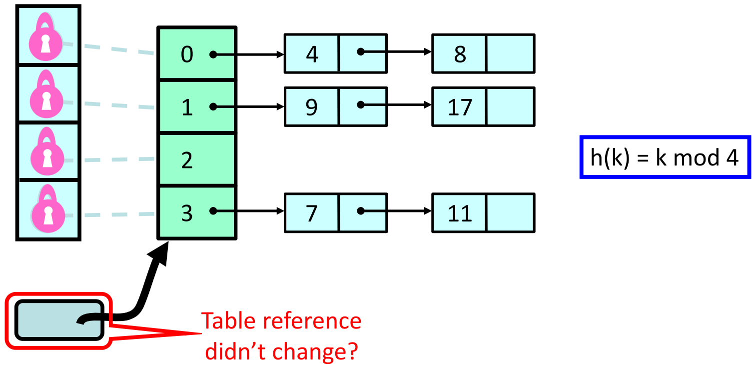
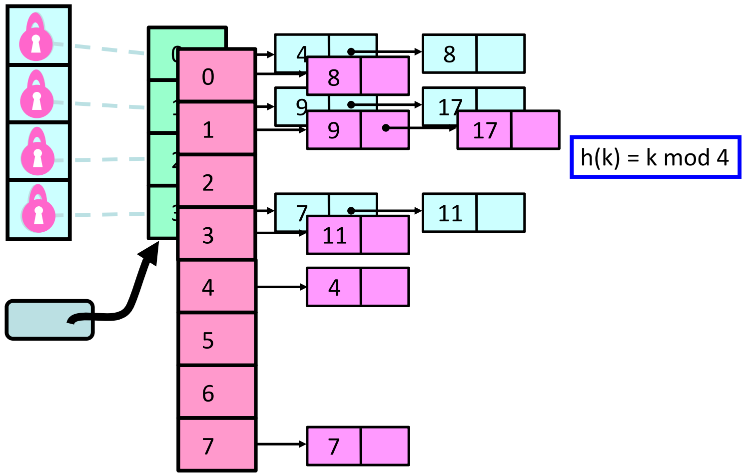


Table reference didn't change?

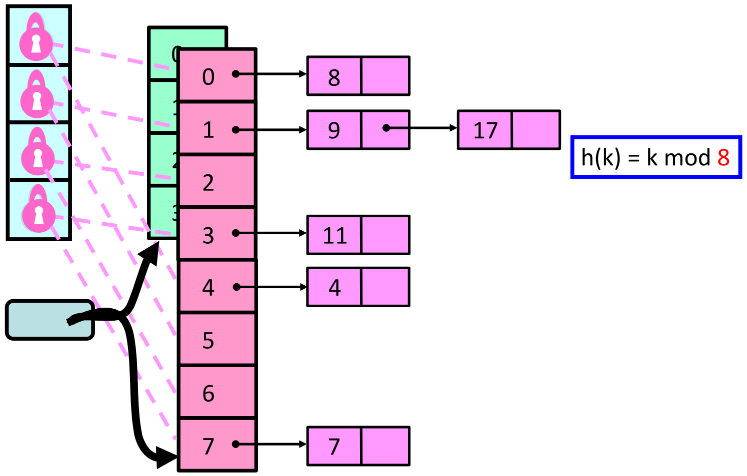
Fine-Grained Locking: Resizing

- Allocate a new table and copy all elements



Fine-Grained Locking: Resizing

- Stripe the locks: Each lock is now associated with two buckets
- Update the hash function and the table reference



Observations

- We grow the table, but we don't increase the number of locks
  - Resizing the lock array is tricky …
- We use sequential lists (coarse-grained locking)
  - No lock-free list
  - If we're locking anyway, why pay?

Fine-Grained Hash Set

```
public class FGHashSet {
  protected RangeLock[] lock;
  protected List[] table;

  public FGHashSet(int capacity) {
    table = new List[capacity];
    lock = new RangeLock[capacity];
    for (int i = 0; i < capacity; i++)
      lock[i] = new RangeLock();
    table[i] = new LinkedList();
    }
  }
```

Array of locks

Array of buckets

Initially the same number of locks and buckets

Fine-Grained Hash Set: Add Method

```
public boolean add(Object key) {
 int keyHash = key.hashCode() % lock.length;
 synchronized(lock[keyHash]) {
  int tableHash = key.hashCode() % table.length;
  return table[tableHash].add(key);
 }
}
```

Acquire the right lock

Call the add() method of the right bucket

Fine-Grained Hash Set: Resize Method

```
 public void resize(int depth, List[] oldTable) {
  synchronized (lock[depth]) {
   if (oldTable == this.table) {
    int next = depth + 1;
    if (next < lock.length)
     resize(next, oldTable);
    else
     sequentialResize();
   }
  }
 }
}
```

Resize() calls resize(0,this.table)

Acquire the next lock and check that no one else has resized

Recursively acquire the next lock

Once the locks are acquired, do the work

Fine-Grained Locks: Evaluation

- We can resize the table, but not the locks
- It is debatable whether method calls are constant-time in presence of contention …

- Insight: The contains() method does not modify any fields
  - Why should concurrent contains() calls conflict?

Read/Write Locks

```
public interface ReadWriteLock {
 Lock readLock();
 Lock writeLock();
}
```

Return the associated read lock

Return the associated write lock

Lock Safety Properties

- No thread may acquire the write lock
  - while any thread holds the write lock
  - or the read lock
- No thread may acquire the read lock
  - while any thread holds the write lock
- Concurrent read locks OK

- This satisfies the following safety properties
  - If readers > 0 then writer == false
  - If writer = true then readers == 0

Read/Write Lock: Liveness

- How do we guarantee liveness?
  - If there are lots of readers, the writers may be locked out!

- Solution: FIFO Read/Write lock
  - As soon as a writer requests a lock, no more readers are accepted
  - Current readers “drain” from lock and the writers acquire it eventually

---

Optimistic Synchronization

- What if the contains() method scans without locking…?

- If it finds the key
  - It is ok to return true!
  - Actually requires a proof…

We won’t discuss this in this lecture

- What if it doesn’t find the key?
  - It may be a victim of resizing…
  - Get a read lock and try again!
  - This makes sense if is expected (?) that the key is there and resizes are rare…

Stop The World Resizing

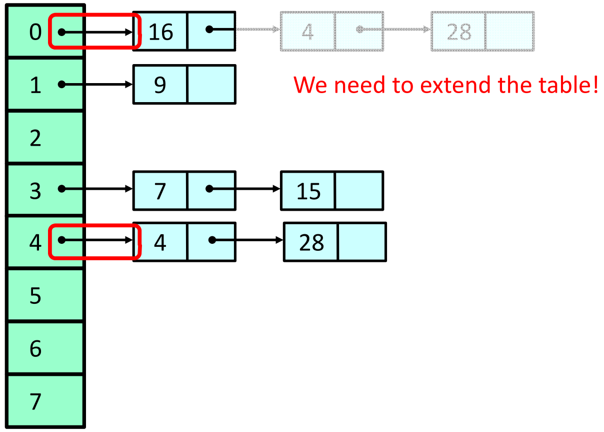
- The resizing we have seen up till now stops all concurrent operations
- Can we design a resize operation that will be incremental?
- We need to avoid locking the table…

- We want a lock-free table with incremental resizing!

How…?

Lock-Free Resizing Problem

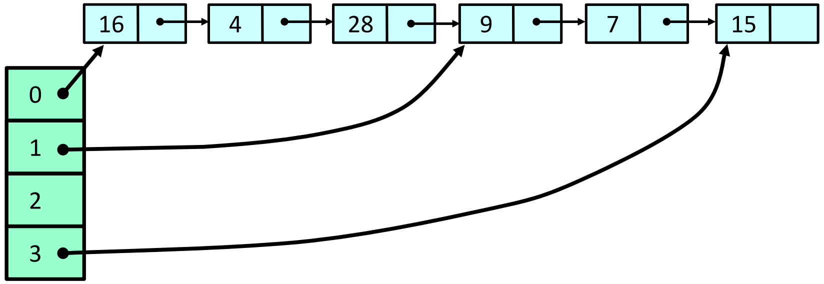
- In order to remove and then add even a single item, “single location CAS’ is not enough…



We need to extend the table!

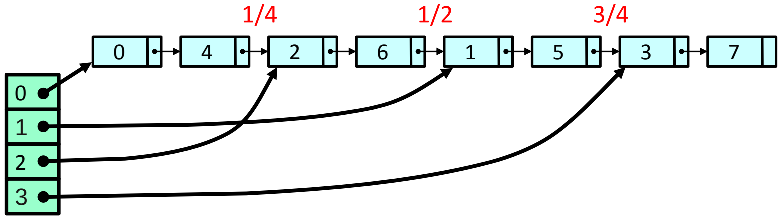
Idea: Don’t Move the Items

- Move the buckets instead of the items!
- Keep all items in a single lock-free list
- Buckets become “shortcut pointers” into the list



Recursive Split Ordering

- Example: The items 0 to 7 need to be hashed into the table
- Recursively split the list the buckets in half:

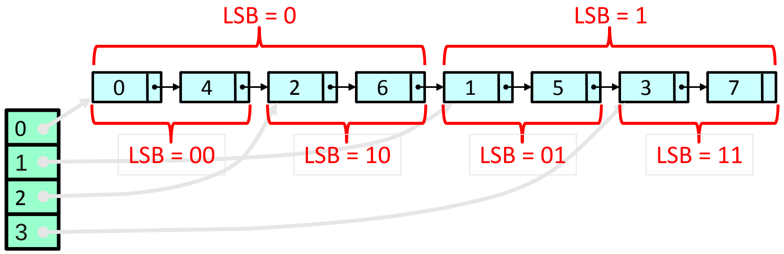


- The list entries are sorted in an order that allows recursive splitting

How…?

Recursive Split Ordering

- Note that the least significant bit (LSB) is 0 in the first half and 1 in the other half! The second LSB determines the next pointers etc.

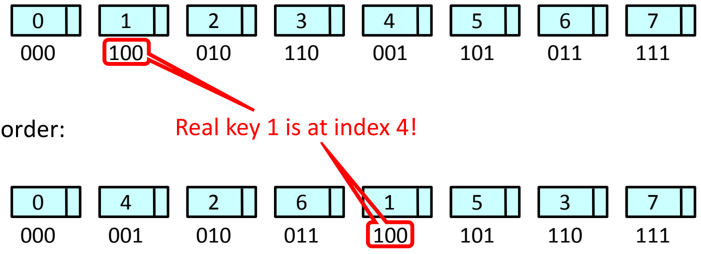


Split-Order

- If the table size is 2^i:
  - Bucket b contains keys k = b mod 2^i
  - The bucket index consists of the key's i least significant bits
- When the table splits:
  - Some keys stay (b = k mod 2^{i+1})
  - Some keys move (b+2^i = k mod2^{i+1})
- If a key moves is determined by the (i+1)st bit
  - counting backwards

A Bit of Magic

- We need to map the real keys to the split-order
- Look at the binary representation of the keys and the indices
- The real keys:



0	1	2	3	4	5	6	7
000	100	010	110	001	101	011	111

- Split-order:

0	4	2	6	1	5	3	7
000	001	010	011	100	101	110	111

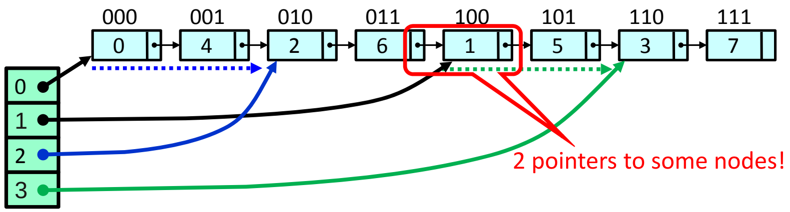
Real key 1 is at index 4!

- Just reverse the order of the key bits!

Split Ordered Hashing

- After a resize, the new pointers are found by searching for the right index

Order according to reversed bits

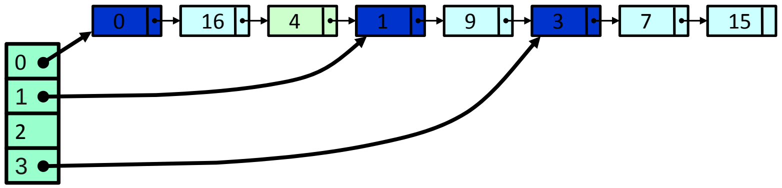


2 pointers to some nodes!

- A problem remains: How can we remove a node by means of a CAS if two sources point to it?

Sentinel Nodes

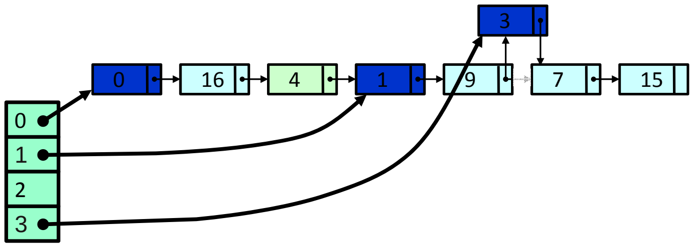
- Solution: Use a sentinel node for each bucket



- We want a sentinel key for i ordered
  - before all keys that hash to bucket i
  - after all keys that hash to bucket (i-1)

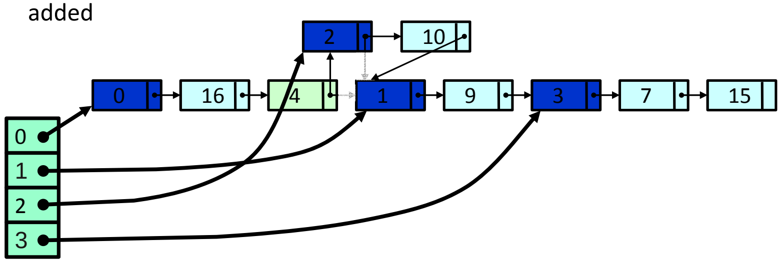
Initialization of Buckets

- We can now split a bucket in a lock-free manner using two CAS() calls
- Example: We need to initialize bucket 3 to split bucket 1!



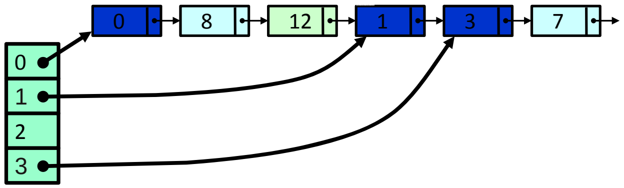
Adding Nodes

- Example: Node 10 is added
- First, bucket 2 (= 10 mod 4) must be initialized, then the new node is added



Recursive Initialization

- It is possible that buckets must be initialized recursively
- Example: When node 7 is added, bucket 3 (= 7 mod 4) is initialized and then bucket 1 (= 3 mod 2) is also initialized



n = number of nodes

- Note that ≈ log *n* empty buckets may be initialized if one node is added, but the expected depth is constant!

Lock-Free List

```
private int makeRegularKey(int key) {
  return reverse(key | 0x80000000);
}

private int makeSentinelKey(int key) {
  return reverse(key);
}
```

Set high-order bit to 1 and reverse

Simply reverse (high-order bit is 0)

Split-Ordered Set

```
public class SOSet{
  protected LockFreeList[] table;
  protected AtomicInteger tableSize;
  protected AtomicInteger setSize;

  public SOSet(int capacity) {
    table = new LockFreeList[capacity];
    table[0] = new LockFreeList();
    tableSize = new AtomicInteger(2);
    setSize = new AtomicInteger(0);
}
```

This is the lock-free list (slides 108-116) with minor modifications

Track how much of table is used and the set size so we know when to resize

Initially use 1 bucket and the size is zero

Split-Ordered Set: Add

```
public boolean add(Object object) {
  int hash = object.hashCode();
  int bucket = hash % tableSize.get();
  int key = makeRegularKey(hash);
  LockFreeList list = getBucketList(bucket);
  if (!list.add(object,key))
    return false;
  resizeCheck();
  return true;
}
```

Pick a bucket

Non-sentinel split-ordered key

Get pointer to bucket's sentinel, initializing if necessary

Try to add with reversed key

Resize if necessary

Recall: Resizing & Initializing Buckets

- Resizing
  - Divide the set size by the total number of buckets
  - If the quotient exceeds a threshold, double the tableSize field up to a fixed limit
- Initializing Buckets
  - Buckets are originally null
  - If you encounter a null bucket, initialize it
  - Go to bucket's parent (earlier nearby bucket) and recursively initialize if necessary
  - Constant expected work!

Split-Ordered Set: Initialize Bucket

```
public void initializeBucket(int bucket) {
  int parent = getParent(bucket);
  if (table[parent] == null)
    initializeBucket(parent);
  int key = makeSentinelKey(bucket);
  LockFreeList list = new
    LockFreeList(table[parent],key);
}
```

Find parent, recursively initialize if needed

Prepare key for new sentinel

Insert sentinel if not present and return reference to rest of list

Correctness

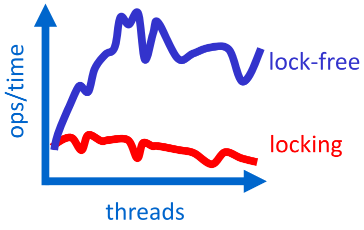
- Split-ordered set is a correct, linearizable, concurrent set implementation

- Constant-time operations!
  - It takes no more than O(1) items between two dummy nodes on average
  - Lazy initialization causes at most O(1) expected recursion depth in initializeBucket()

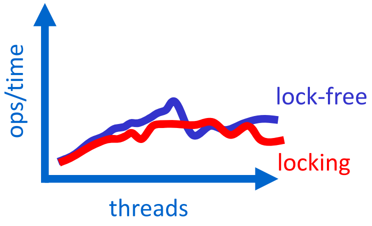
Empirical Evaluation

- Evaluation has been performed on a 30-processor Sun Enterprise 3000
- Lock-Free vs. fine-grained (Lea) optimistic locking
- In a non-multiprogrammed environment
- 10^6 operations: 88% contains(), 10% add(), 2% remove()

No work:

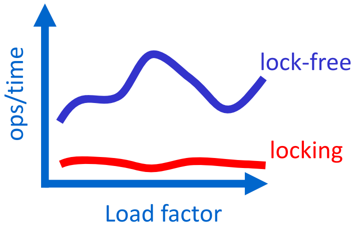


Busy:

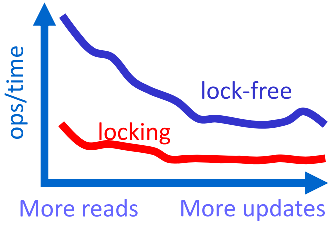


Empirical Evaluation

- Expected bucket length
  - The load factor is the capacity of the individual buckets



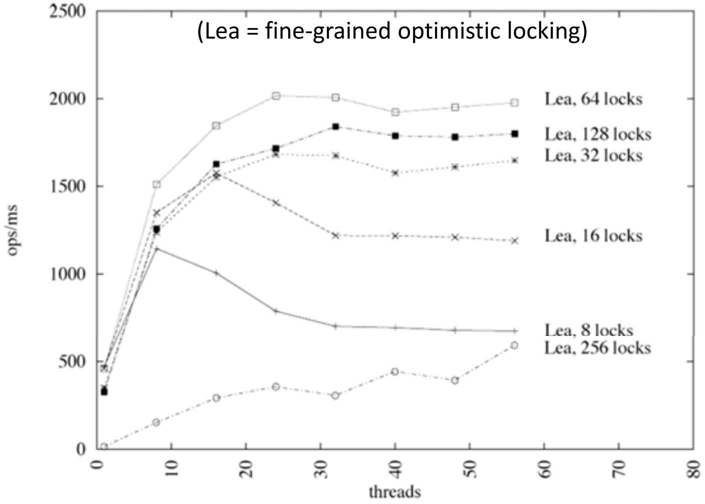
- Varying The Mix
  - Increasing the number of updates



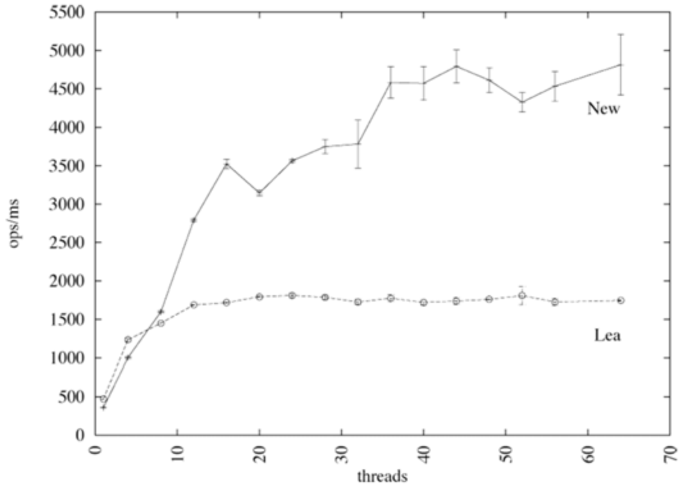
Additional Performance

- Additionally, the following parameters have been analyzed:
  - The effects of the choice of locking granularity
  - The effects of the bucket size

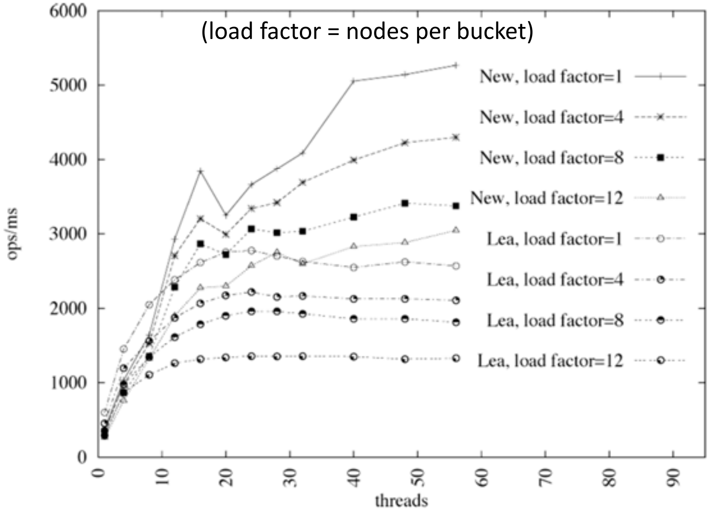
Number of Fine-Grain Locks


(Lea = fine-grained optimistic locking)

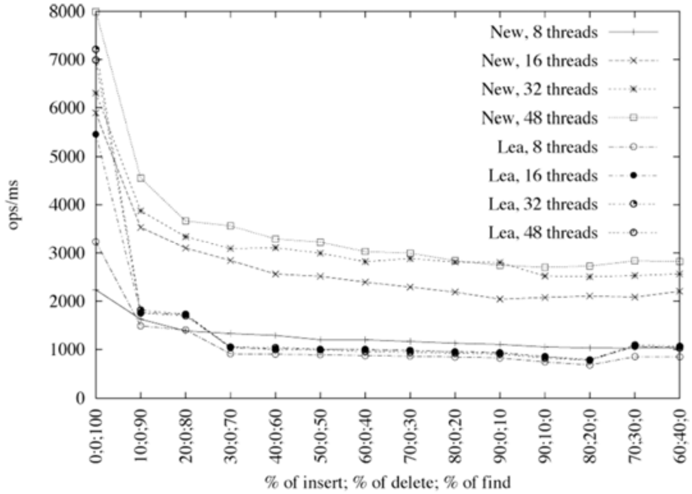
Lock-free vs. Locks



Hash Table Load Factor


(load factor = nodes per bucket)

Varying Operations



Conclusion

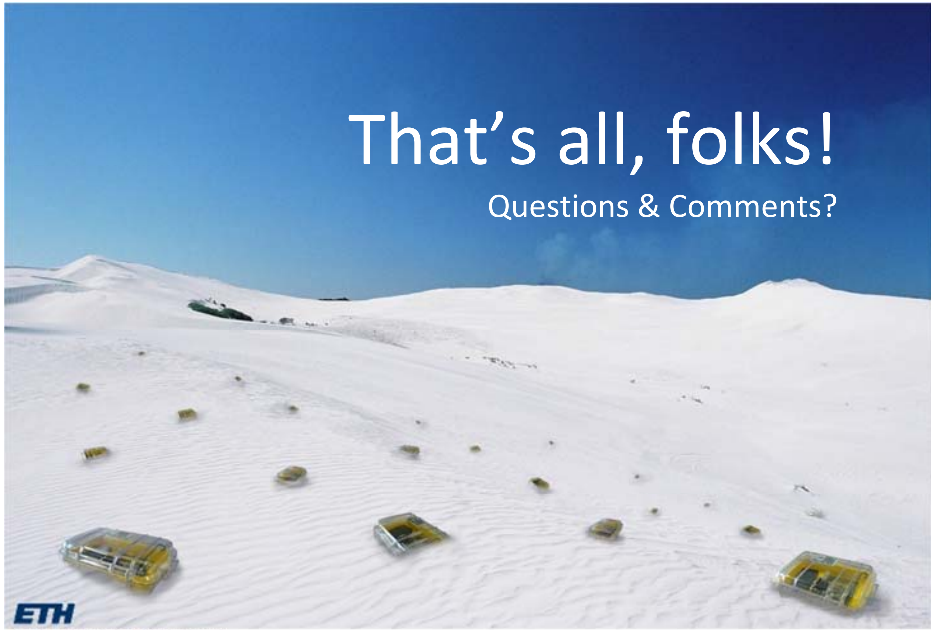
- Concurrent resizing is tricky
- Lock-based
    - Fine-grained
    - Read/write locks
    - Optimistic
- Lock-free
    - Builds on lock-free list

Summary

- We talked about several locking mechanisms
- In particular we have seen
    - TAS & TTAS
    - Alock & backoff lock
    - MCS lock & abortable MCS lock
- We also talked about techniques to deal with concurrency in linked lists
    - Hand-over-hand locking
    - Optimistic synchronization
    - Lazy synchronization
    - Lock-free synchronization
- Finally, we talked about hashing
    - Fine-grained locking
    - Recursive split ordering

Credits

- The TTAS lock is due to Kruskal, Rudolph, and Snir, 1988.
- Tom Anderson invented the ALock, 1990.
- The MCS lock is due to Mellor-Crummey and Scott, 1991.
- The first lock-free list algorithms are credited to John Valois, 1995.
- The lock-free list algorithm discussed in this lecture is a variation of algorithms proposed by Harris, 2001, and Michael, 2002.
- The lock-free hash set based on split-ordering is by Shalev and Shavit, 2006.



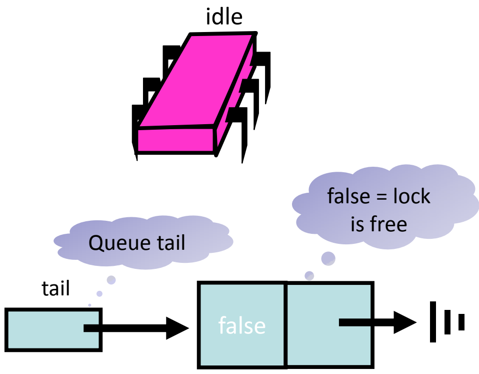
That's all, folks!

Questions & Comments?

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

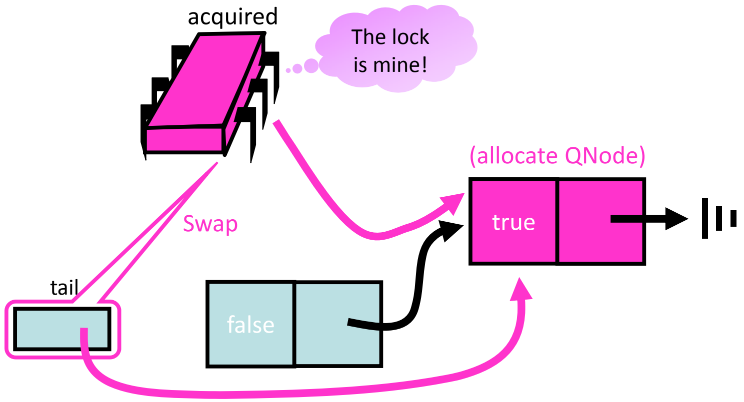
MCS Lock: Initially

- The lock is again represented as a linked list of QNodes
- Unlike the CLH lock the list is explicit



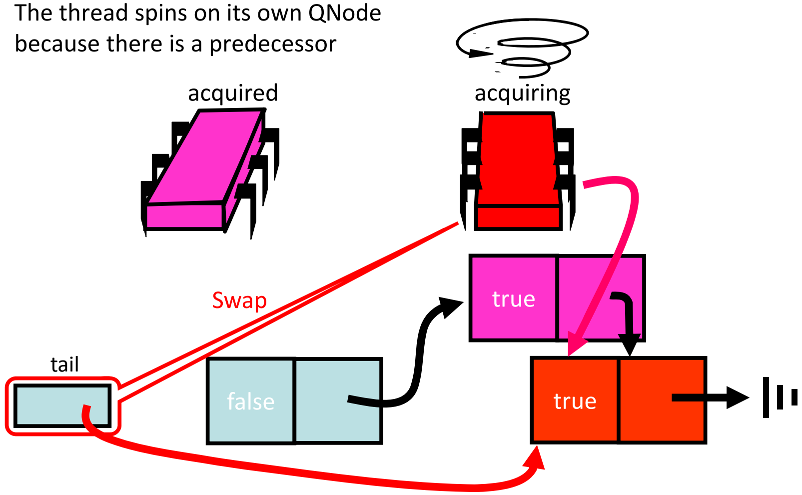
MCS Lock: Acquiring the Lock

- To acquire the lock, the thread places its QNode at the tail of the list
- The thread then swaps in a reference to its own Qnode
- There is no predecessor, so the thread acquires the lock



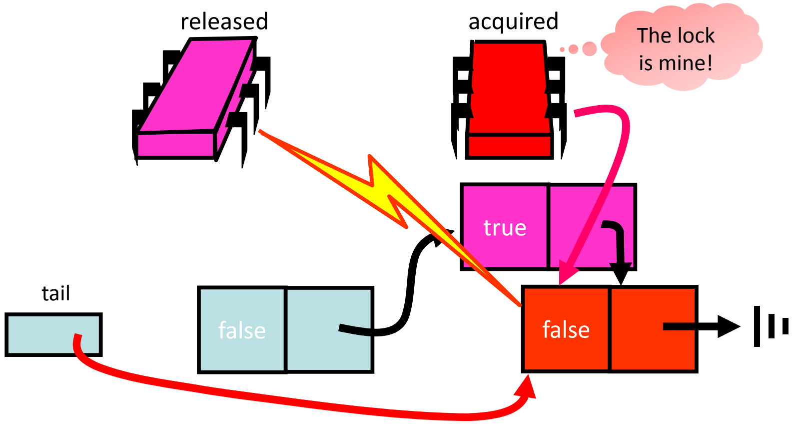
MCS Lock: Contention

- If another thread wants to acquire the lock, it again applies swap
- The thread spins on its own QNode because there is a predecessor



MCS Lock: Release the Lock

- The first thread releases the lock by setting its successor's QNode to false



MCS Queue Lock

```
public class QNode {
  boolean locked = false;
  QNode next = null;
}
```

MCS Queue Lock

```
public 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) {}
    }
  }

  ...
```

Add my node to the tail

Fix if queue was non-empty

MCS Queue Lock

```
    ...

    public void unlock() {
      if (qnode.next == null) {
        if (tail.CAS(qnode, null)
          return;
        while (qnode.next == null) {}
      }
      qnode.next.locked = false;
    }
}
```

Missing successor?

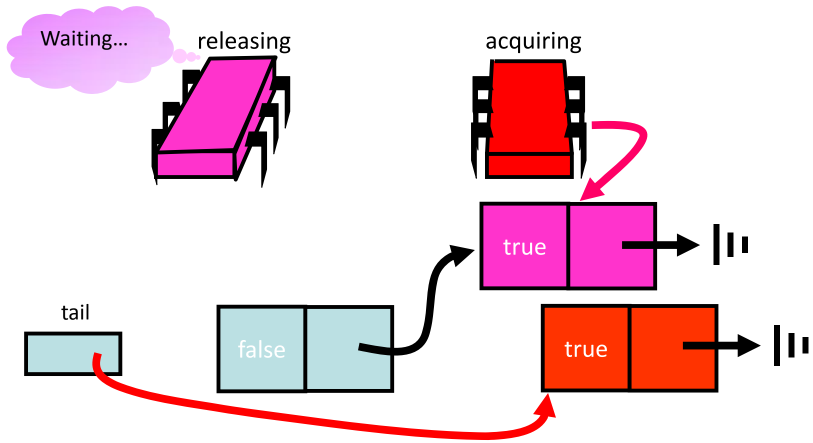
If really no successor, return

Otherwise, wait for successor to catch up

Pass lock to successor

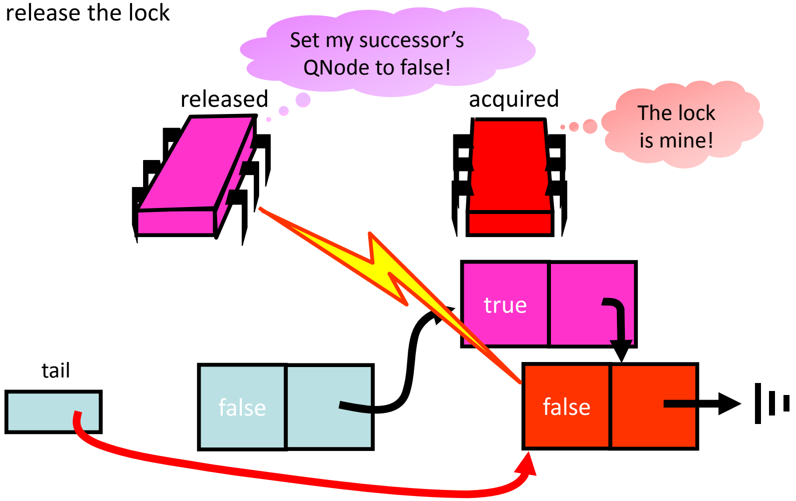
MCS Lock: Unlocking Explained

- The purple thread sees that another thread is active because its QNode does not have a successor, but tail does not point to its QNode!



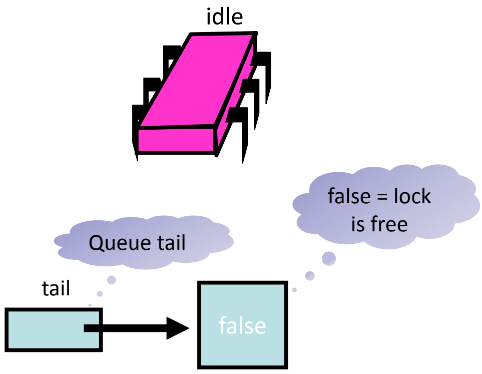
MCS Lock: Unlocking Explained

- As soon as the pointer to the successor is set, the purple thread can release the lock



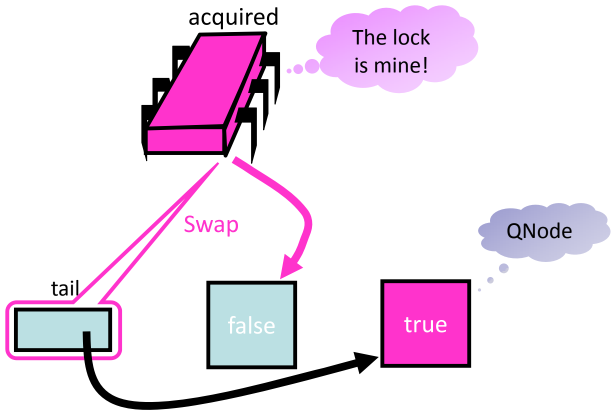
CLH Lock: Initially

- Each thread's status is recorded in a QNode object with a Boolean locked field: if the field is true, the thread has acquired the lock or is waiting for it



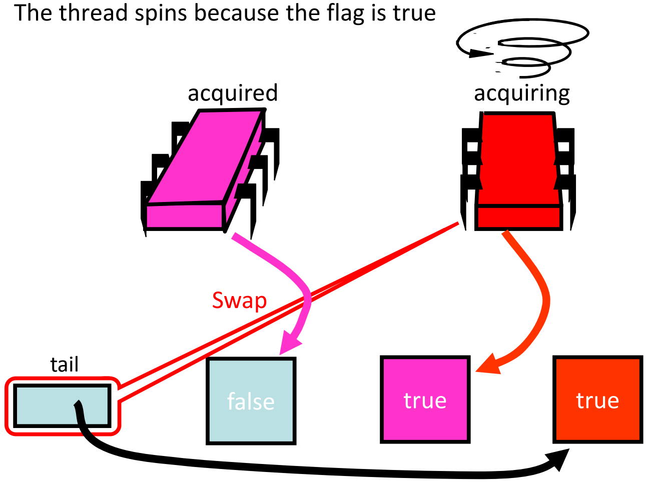
CLH Lock: Acquiring the Lock

- The thread sets the locked field of its Qnode to true
- The thread applies swap to the tail → Its own node is now the tail
- Simultaneously, it acquires a reference to the predecessor's QNode



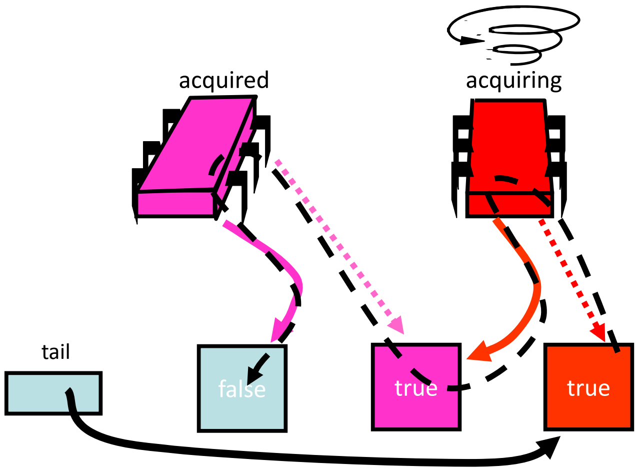
CLH Lock: Contention

- If another thread wants to acquire the lock, it applies swap
- The thread spins because the flag is true



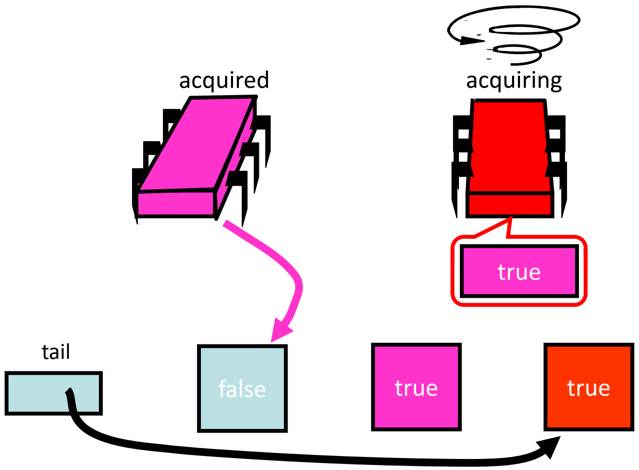
CLH Lock: Implicit Linked List

- Note that the list is ordered implicitly!



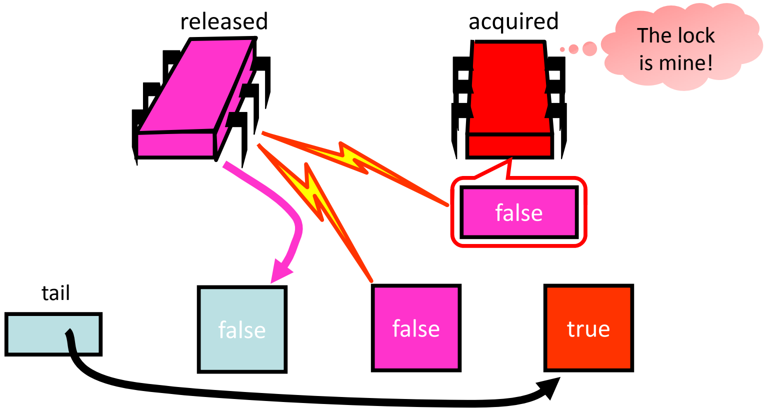
CLH Lock: Spinning on Cache

- Note that the red thread actually spins on a cached copy



CLH Lock: Release Lock

- The first thread releases the lock by setting its QNode to false
- The second thread notices the change and gets the lock
- The red thread can use its predecessor’s QNode for future lock accesses



CLH Queue Lock

```
public class QNode {
  AtomicBoolean locked = new Atomic Boolean(true);
}
```

CLH Queue Lock

```
public class CLHLock implements Lock {
  AtomicReference<QNode> tail;
  ThreadLocal<QNode> myNode = new Qnode();

  public void lock() {
    Qnode pred = tail.getAndSet(myNode);
    while(pred.locked) {}
  }

  public void unlock() {
    myNode.locked.set(false);
    myNode = pred;
  }
}
```

Tail of the queue

Thread-local QNode

Swap in my node

Recycle predecessor's node

CLH Lock: Evaluation

- Space usage
  - L = number of locks
  - N = number of threads
- ALock
  - O(LN)
- CLH lock
  - O(L+N)

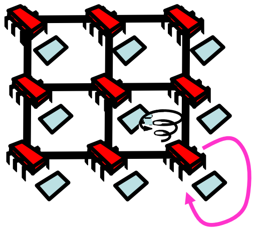
- Good
  - Lock release affects predecessor only
  - Small, constant-sized space
- Bad
  - Doesn't work for uncached NUMA architectures

???

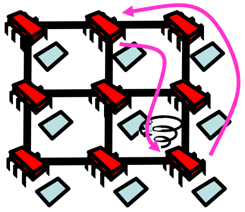
NUMA Architecturs

- **N**on-**U**niform **M**emory **A**rchitecture
- Illusion
  - Flat shared memory
- Truth
  - No caches (sometimes)
  - Some memory regions faster than others

Spinning on local memory is fast:



Spinning on remote memory is slow:



CLH Lock: Problem

- Each thread spin's on predecessor's memory
- The predecessor could be far away …

- What we want is that
  - each thread spins on local memory only
  - and the overhead is still small (constant size)