Modern High-Performance Locking

Nir Shavit

Slides based in part on
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

acquire lock

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

acquire lock

release lock

Mutual Exclusion Properties

Mutual Exclusion

- At most one thread holds the lock (has completed lock() and not completed unlock()) at any time

Mutual Exclusion Properties

Freedom from Deadlock

- If a thread calls lock() or unlock() and never returns, then other threads must complete invocations of lock() and unlock() infinitely often.

Mutual Exclusion Properties

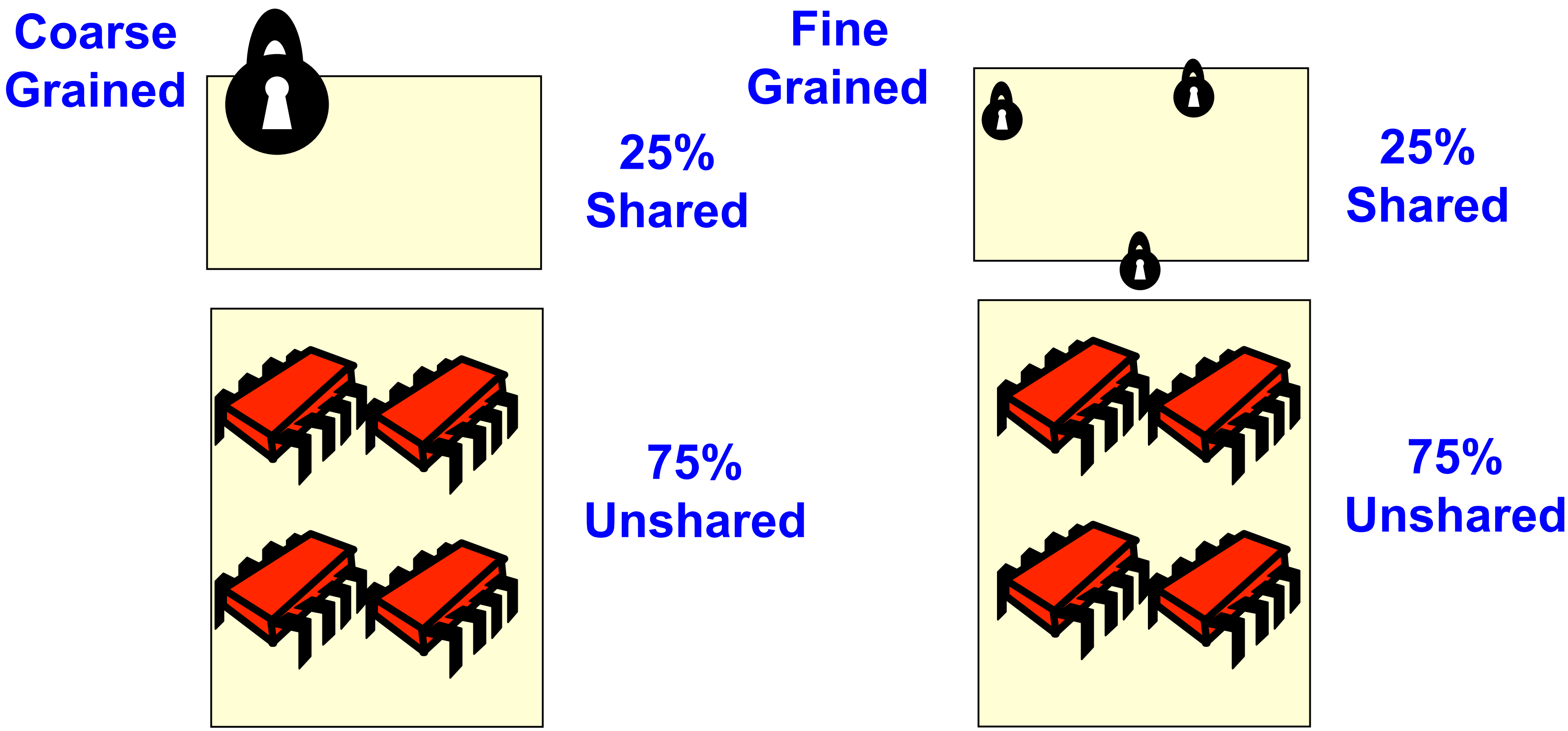
Freedom from Starvation

- Every call to lock() or unlock() eventually returns.

Locking and Amdahl's Law

$$\text{Speedup} = \frac{1}{1 - p + \frac{\boxed{p}}{n}}$$

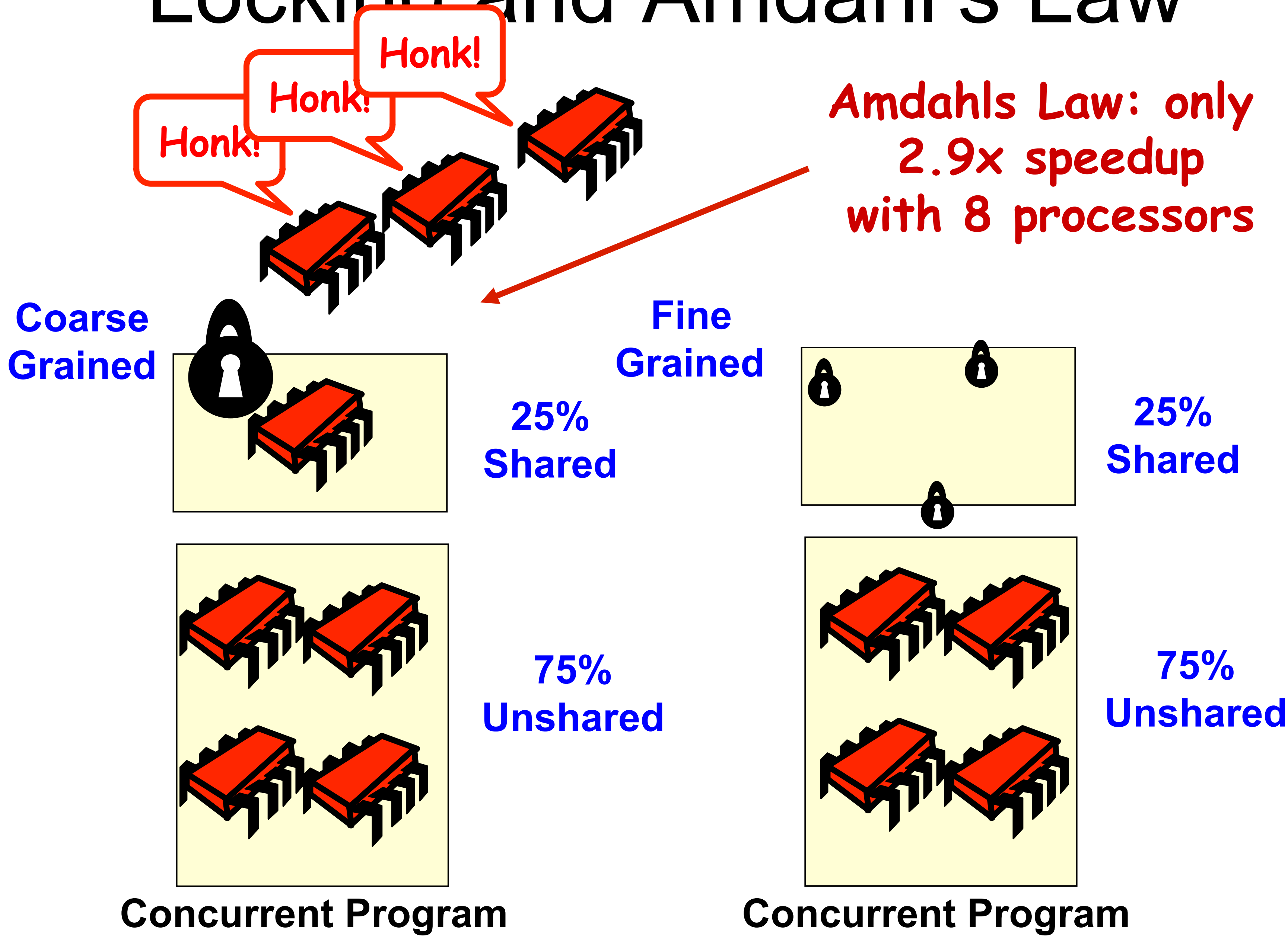
Parallel fraction



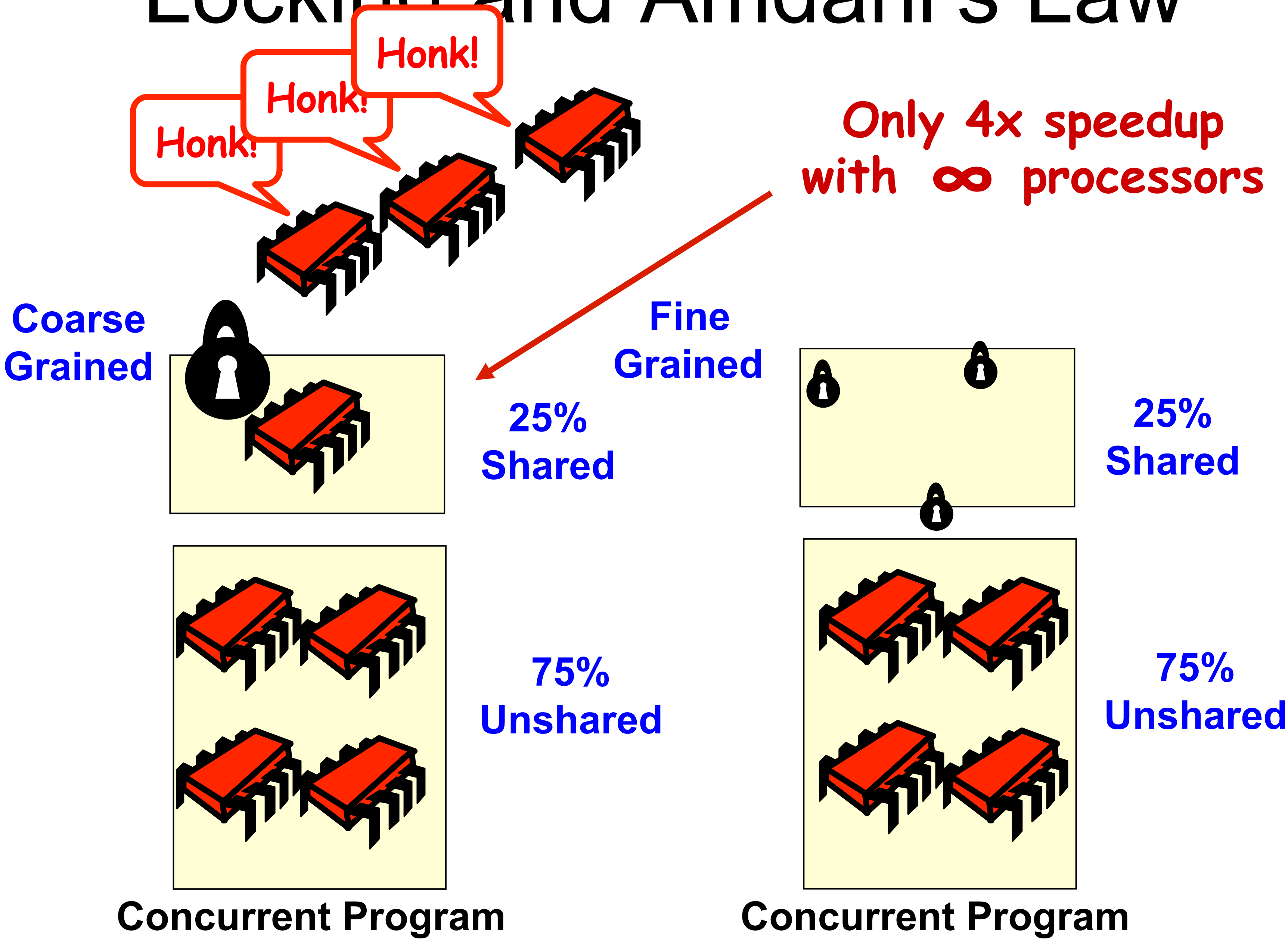
Concurrent Program

Concurrent Program

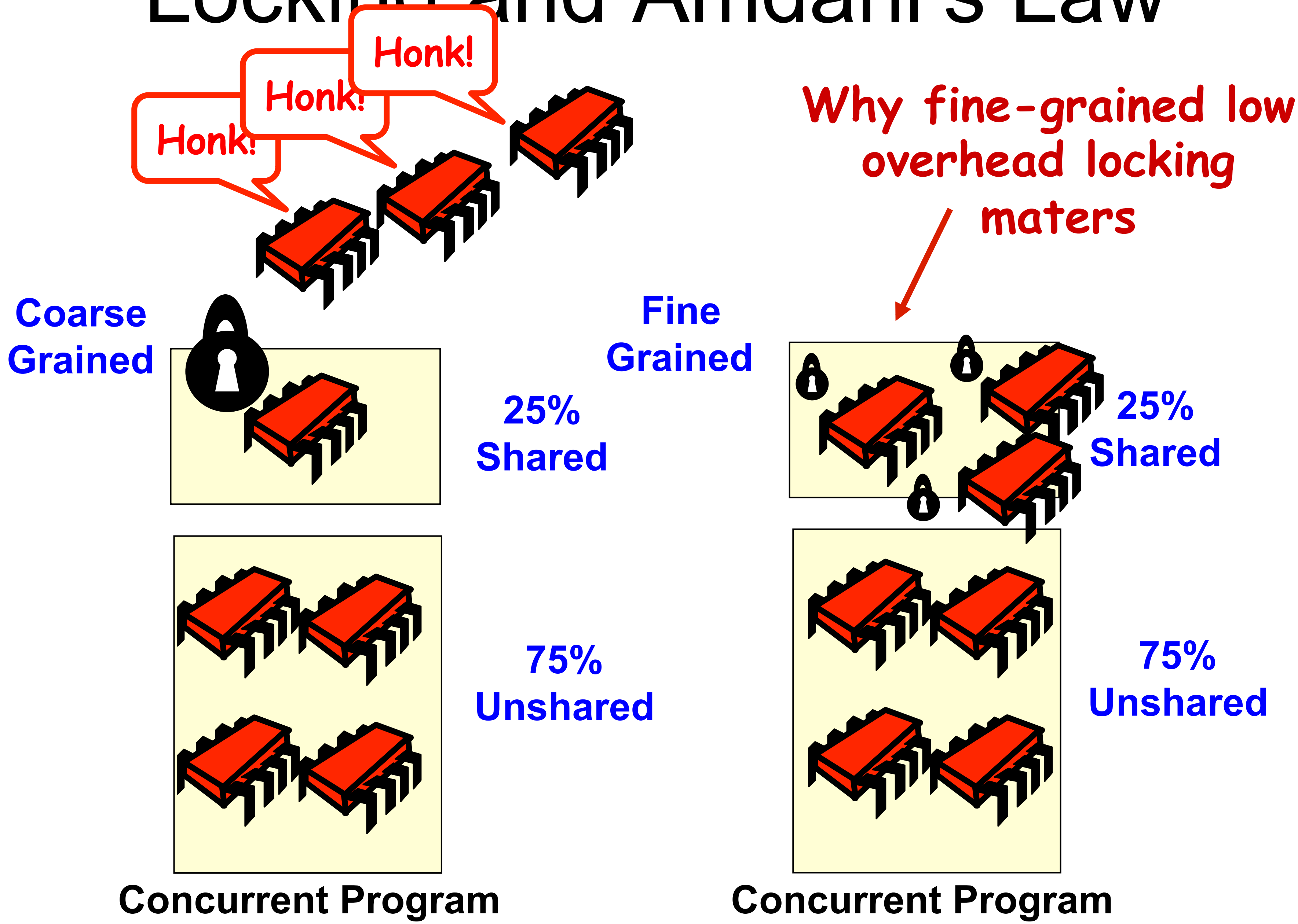
Locking and Amdahl's Law



Locking and Amdahl's Law



Locking and Amdahl's Law



What Should you do if you can’t get a lock?

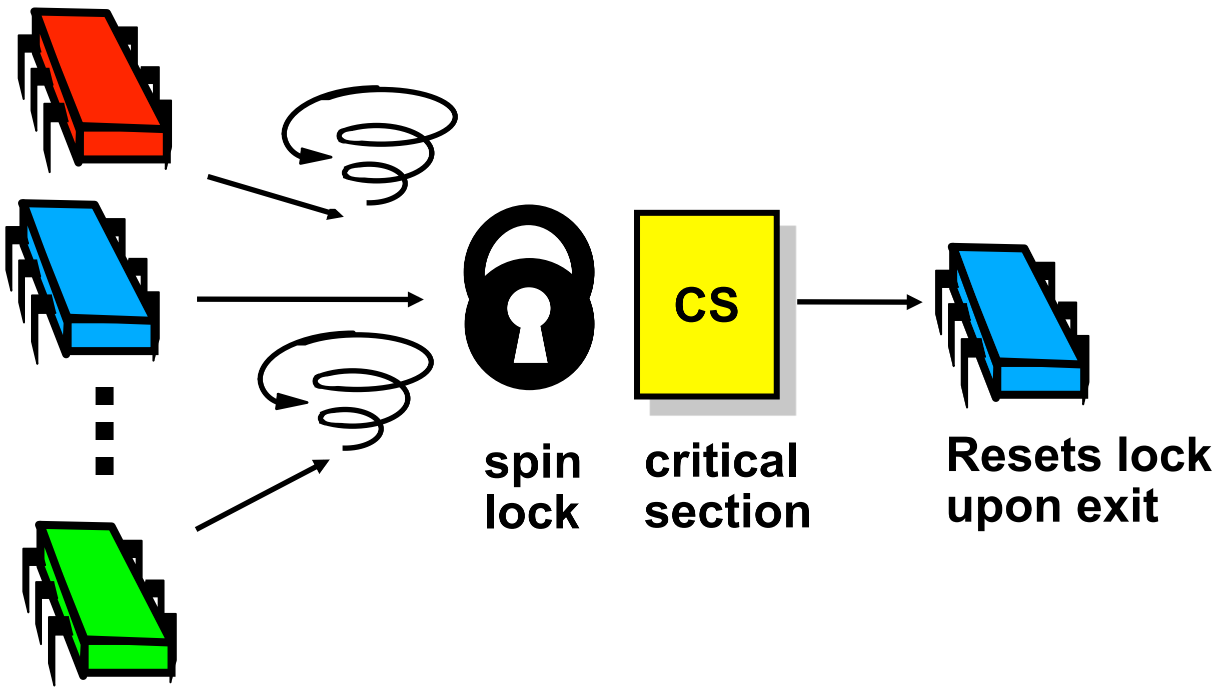
- Keep trying
  - “spin” or “busy-wait”
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

What Should you do if you can't get a lock?

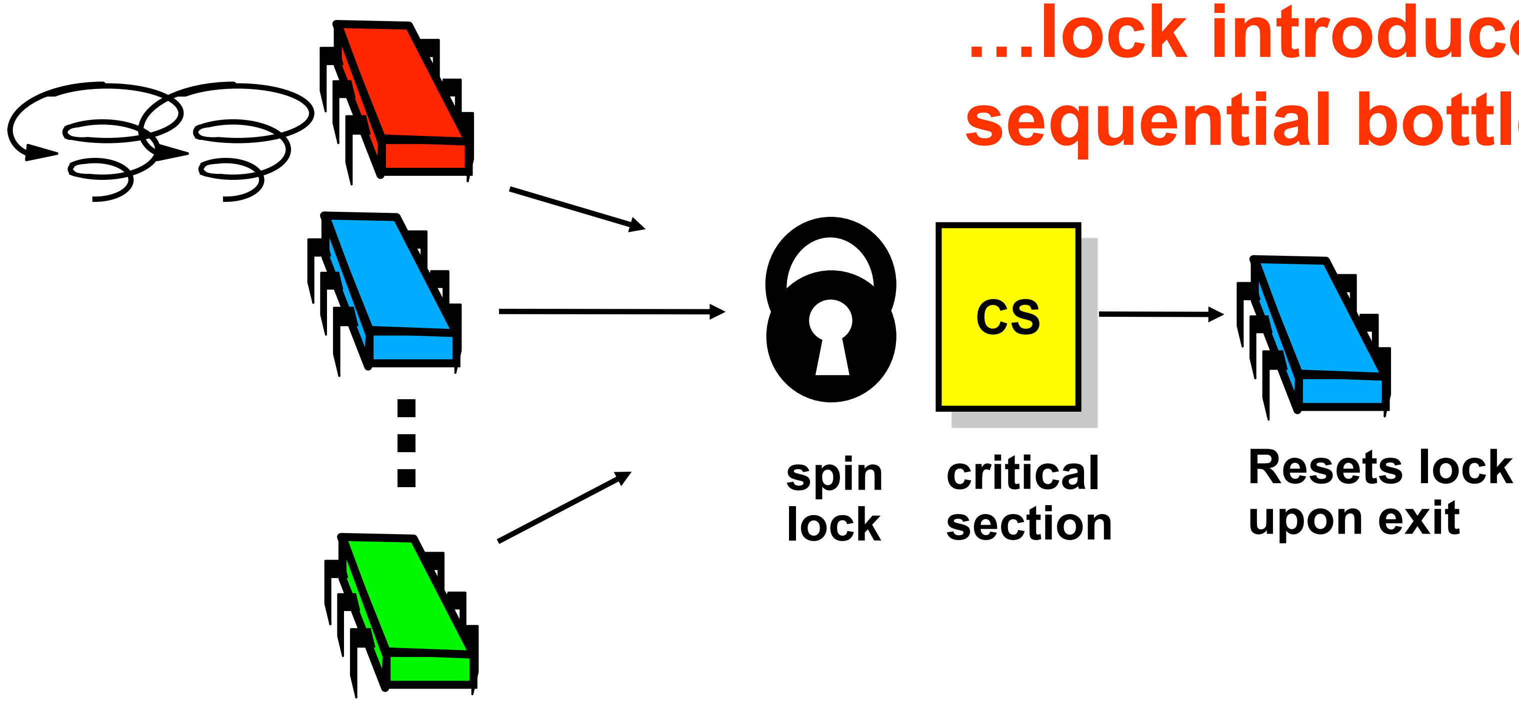
- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

our focus

Basic Spin-Lock

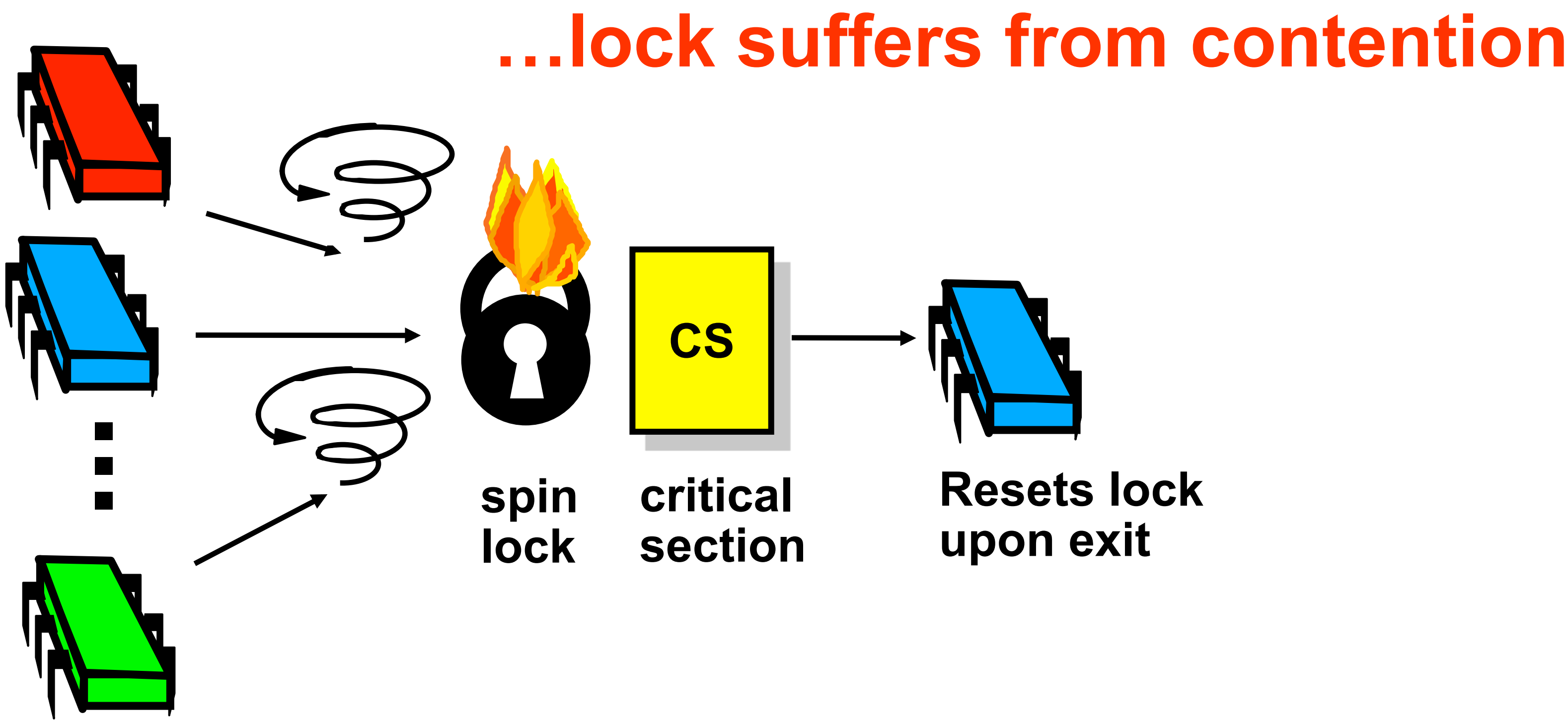


Basic Spin-Lock

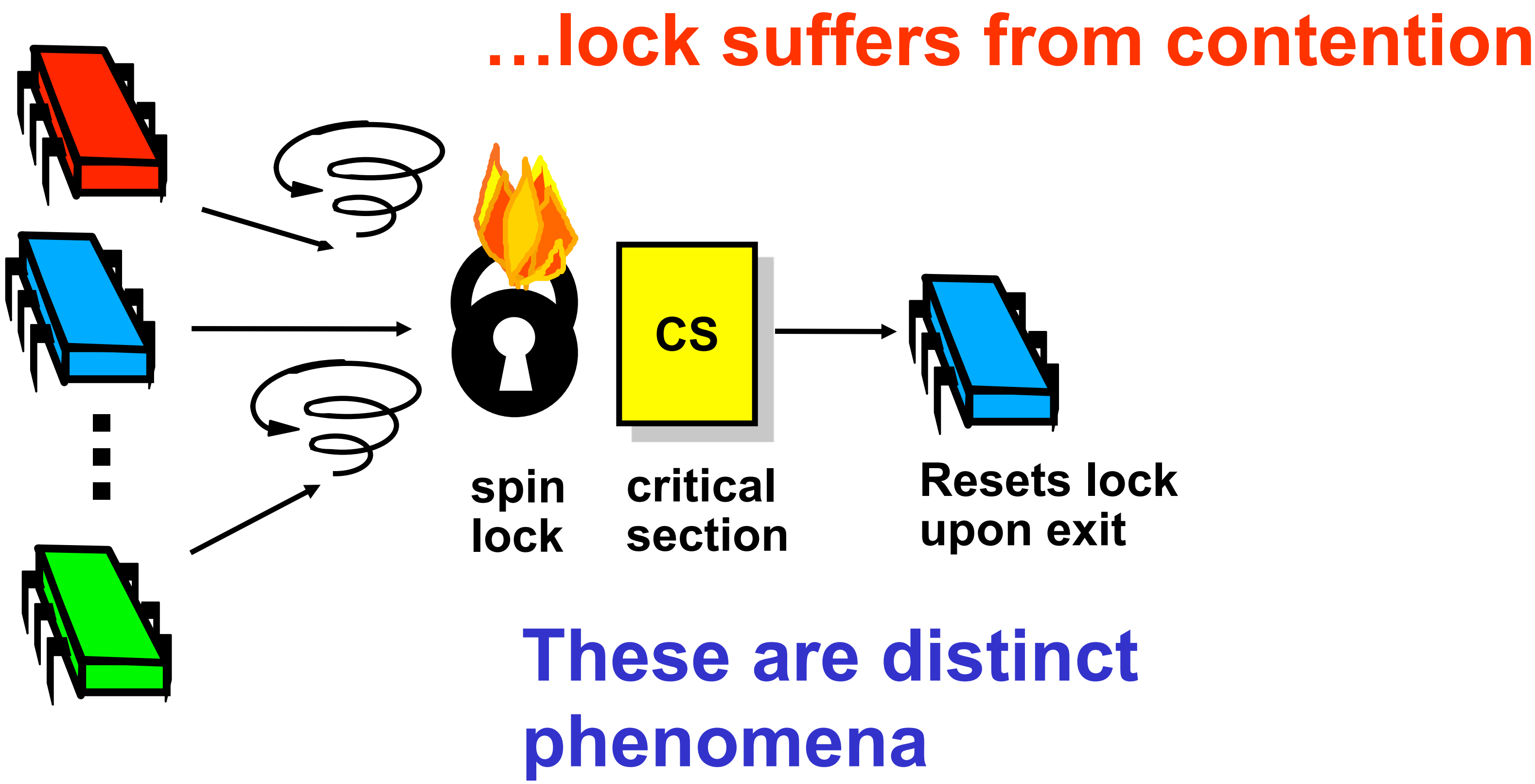


...lock introduces sequential bottleneck

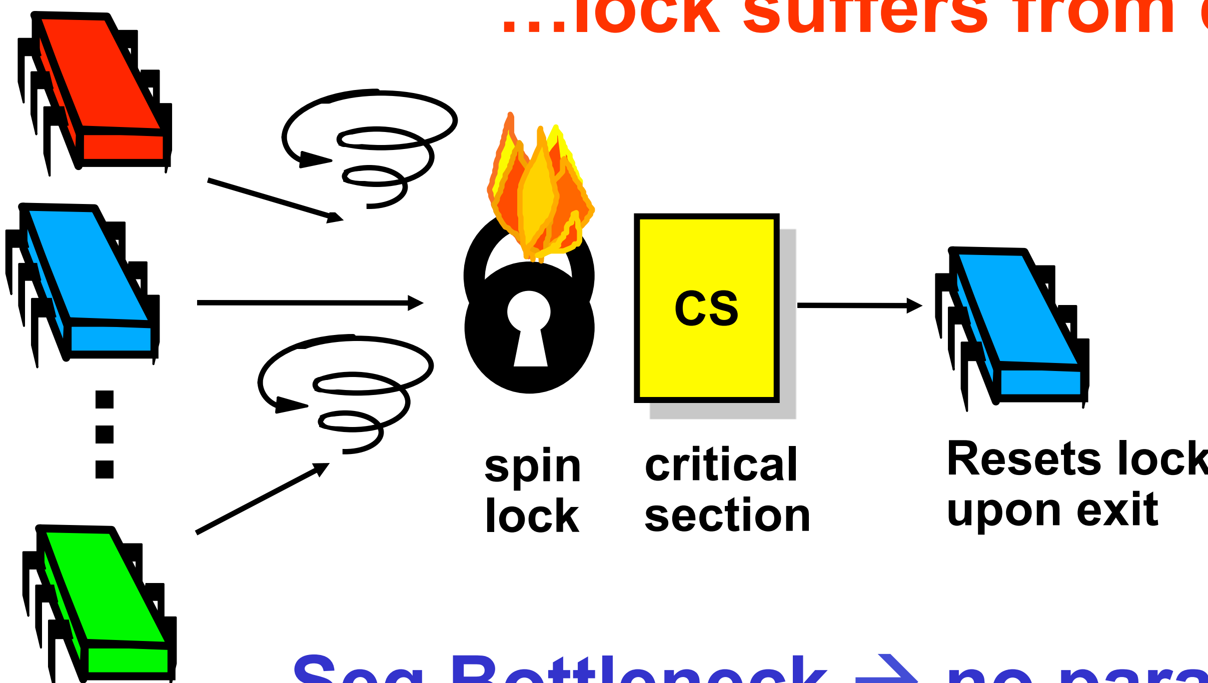
Basic Spin-Lock



Basic Spin-Lock



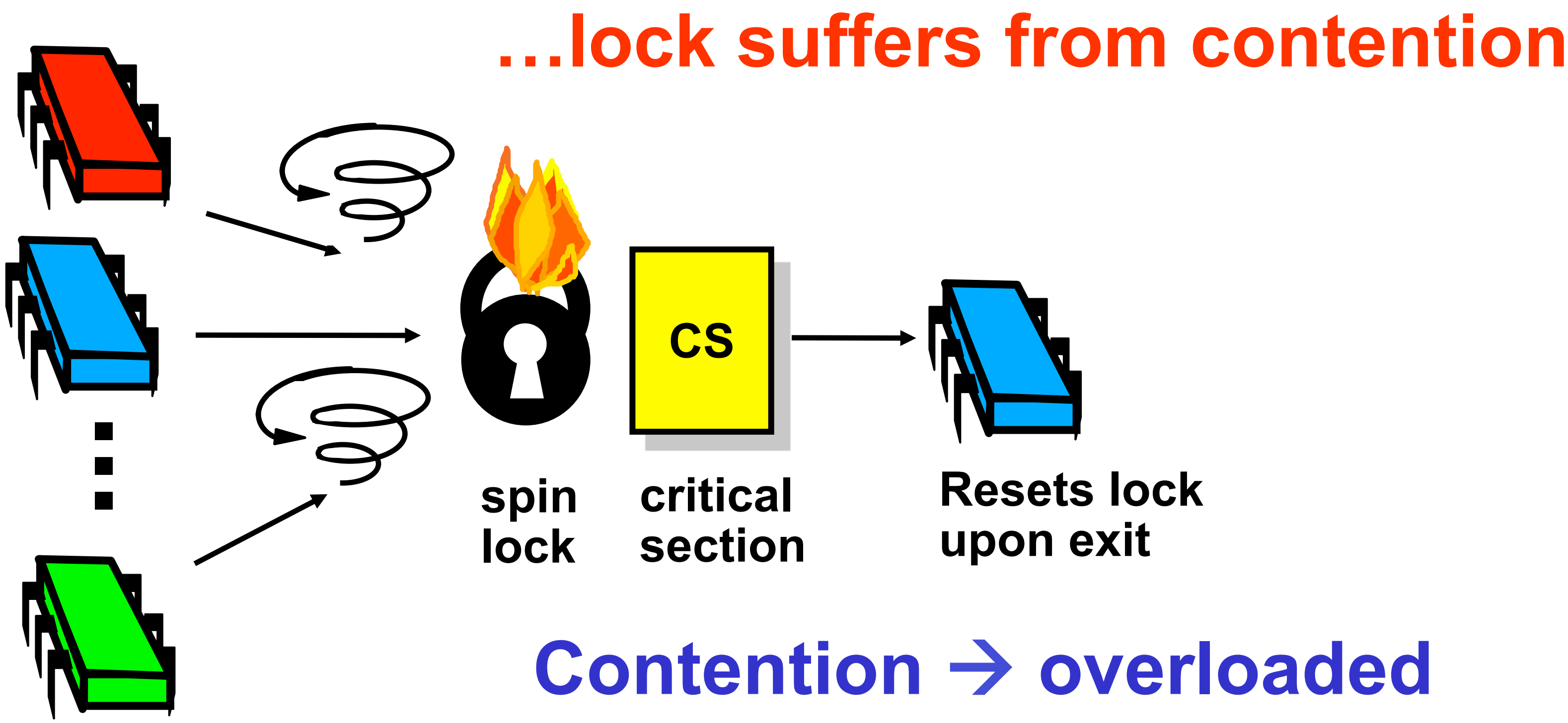
Basic Spin-Lock



...lock suffers from contention

Seq Bottleneck → no parallelism

Basic Spin-Lock



Contention → overloaded communication medium

Mutual Exclusion

- What do we want to optimize?
  - Bus bandwidth used by spinning threads
  - Release/Acquire latency
  - Acquire latency for idle lock

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

 public synchronized boolean
  getAndSet(boolean newValue) {
   boolean prior = value;
   value = newValue;
   return prior;
 }
}
```

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

 public synchronized boolean
  getAndSet(boolean newValue) {
   boolean prior = value;
   value = newValue;
   return prior;
 }
}
```

Package java.util.concurrent.atomic

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

 public synchronized boolean
  getAndSet(boolean newValue) {
   boolean prior = value;
   value = newValue;
   return prior;
  }
}
```

Swap old and new values.

Review: Test-and-Set

```
AtomicBoolean lock
 = new AtomicBoolean(false)
…
boolean prior = lock.getAndSet(true)
```

Review: Test-and-Set

```
AtomicBoolean lock
 = new AtomicBoolean(false)
…
boolean prior = lock.getAndSet(true)
```

Swapping in true is called "test-and-set" or TAS. Both "Swap" and "TAS" available in hardware.

Test-and-Set Locks

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

Simple TASLock

- TAS invalidates cache lines
- Spinners
  - Miss in cache
  - Go to bus
- Thread wants to release lock
  - delayed behind spinners

Test-and-Test-and-Set Locks

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

Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

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

Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

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

Wait until lock looks free

Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

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

Then try to acquire it

Test-and-test-and-set

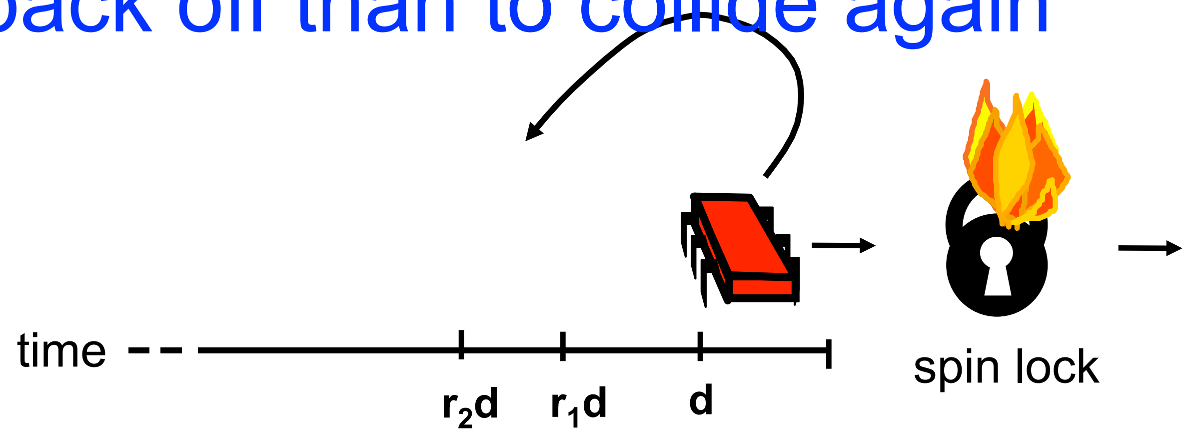
- Wait until lock “looks” free
  – Spin on local cache
  – No bus use while lock busy
- Problem: when lock is released
  – Invalidation storm …

Problems

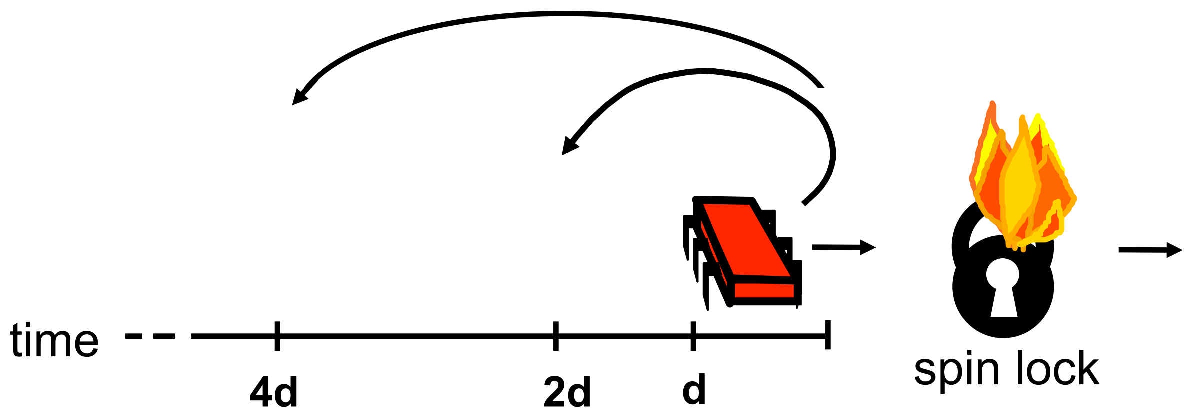
- Everyone misses
  – Reads satisfied sequentially
- Everyone does TAS
  – Invalidates others' caches
- Eventually quiesces after lock acquired
  – Quiescence time often linear in number of cores

Solution: Introduce Delay

- If the lock looks free
  - but I fail to get it
- There must be contention
  - better to back off than to collide again



Dynamic Example: Exponential Backoff



If I fail to get lock

- Wait random duration before retry
- Each subsequent failure doubles expected wait

Exponential Backoff Lock

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

Exponential Backoff Lock

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

Fix minimum delay

Exponential Backoff Lock

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

Wait until lock looks free

Exponential Backoff Lock

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

If we win, return

Exponential Backoff Lock

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

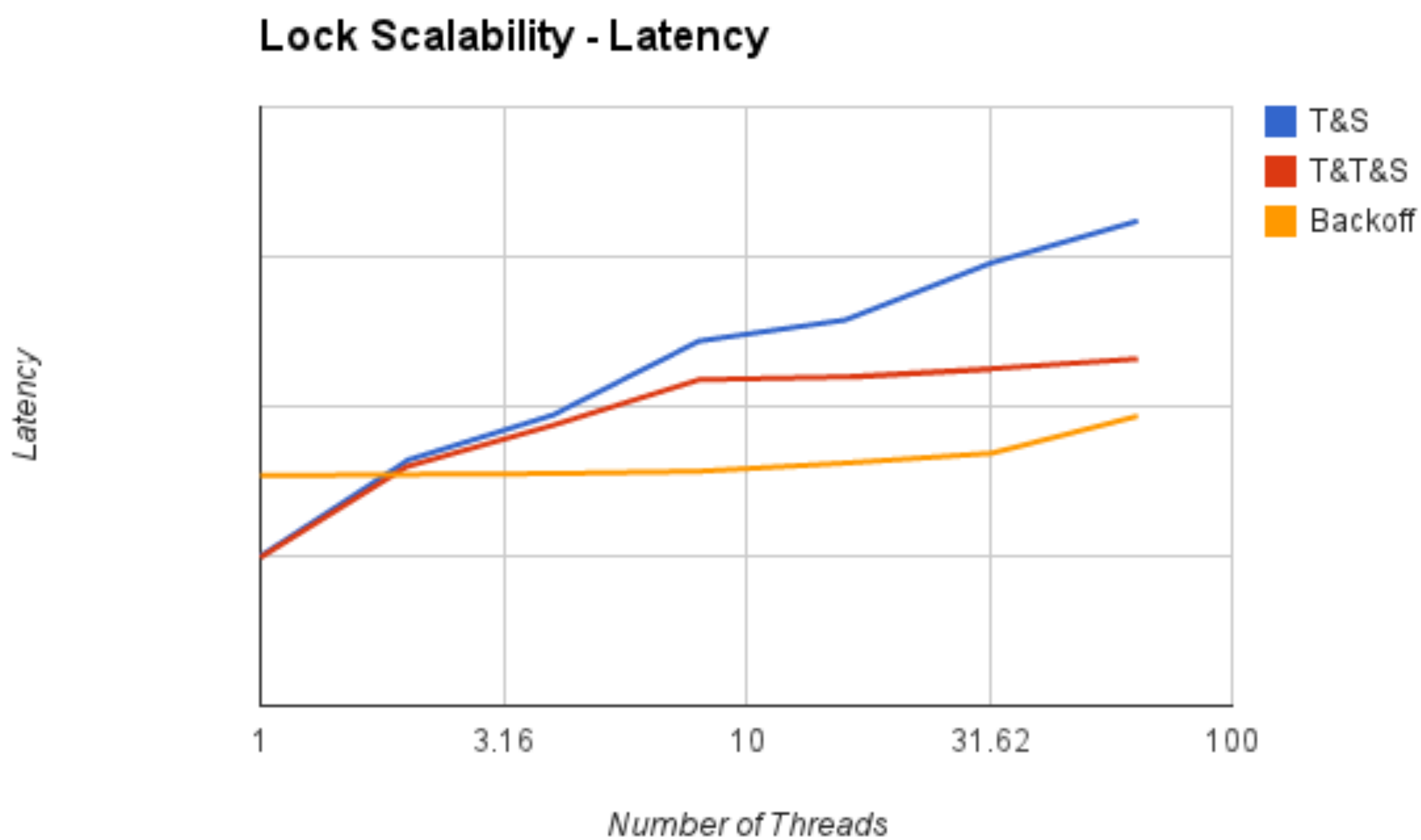
Back off for random duration

Exponential Backoff Lock

Double max delay, within reason

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

Actual Data on 40-Core Machine



Backoff: Other Issues

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

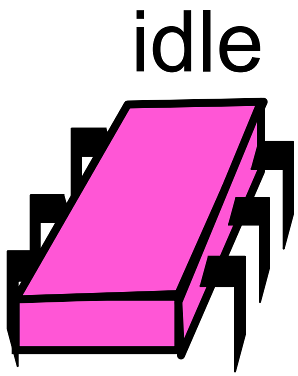
Idea

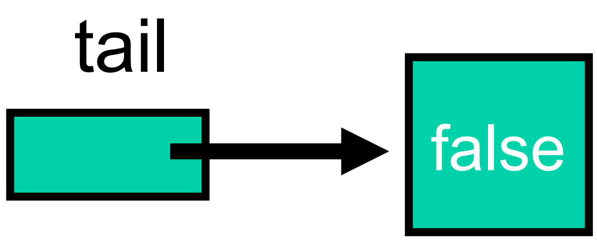
- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others

CLH Lock

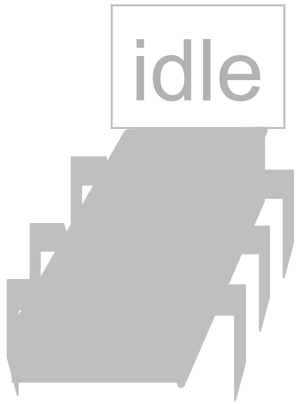
- First Come First Served order
- Small, constant-size overhead per thread

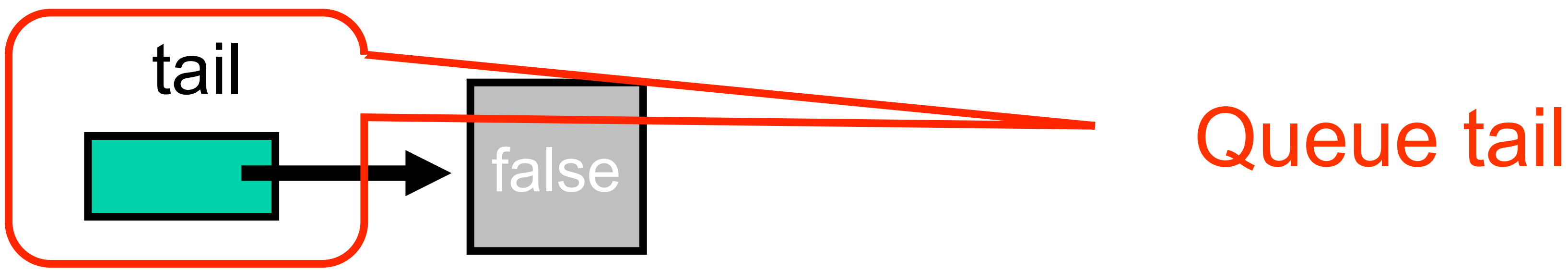
Initially



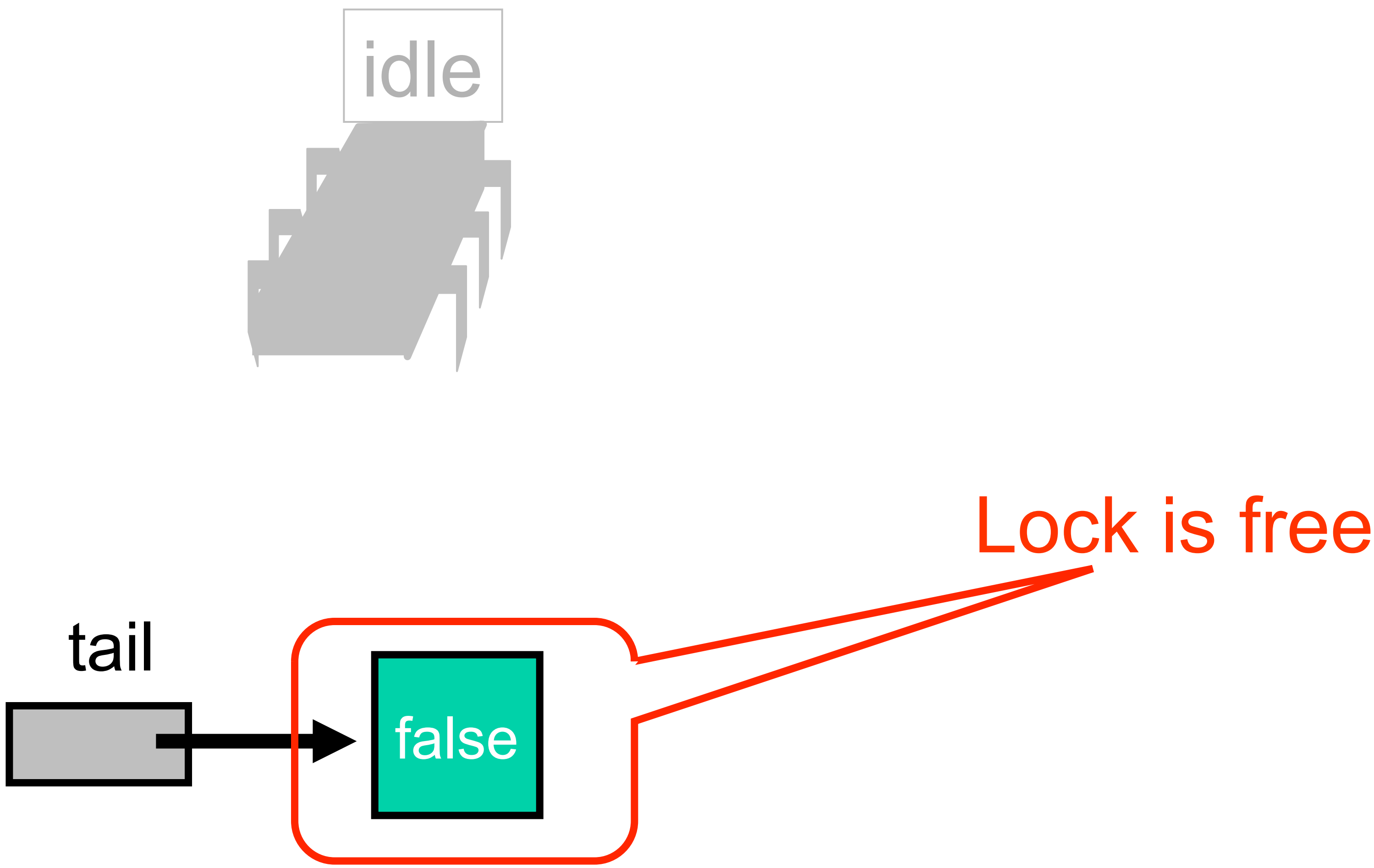


Initially

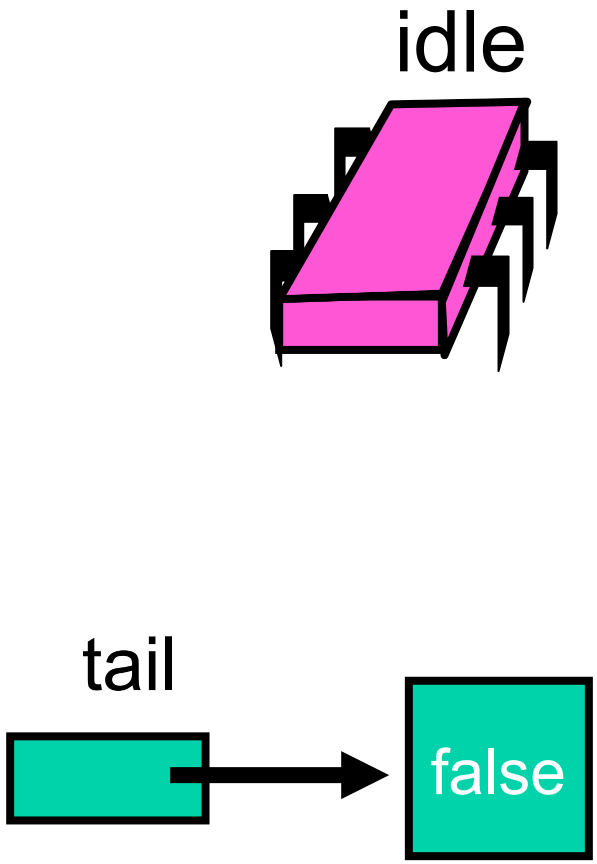




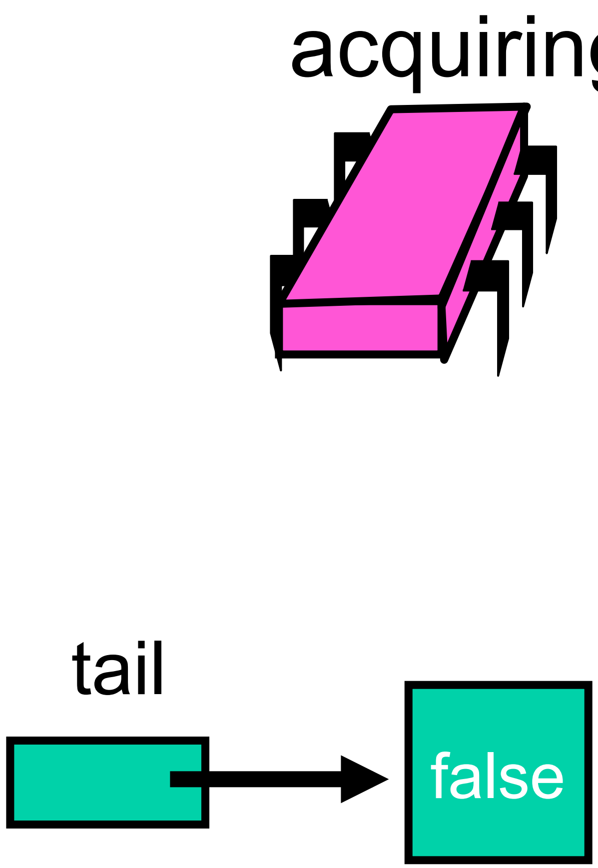
Initially



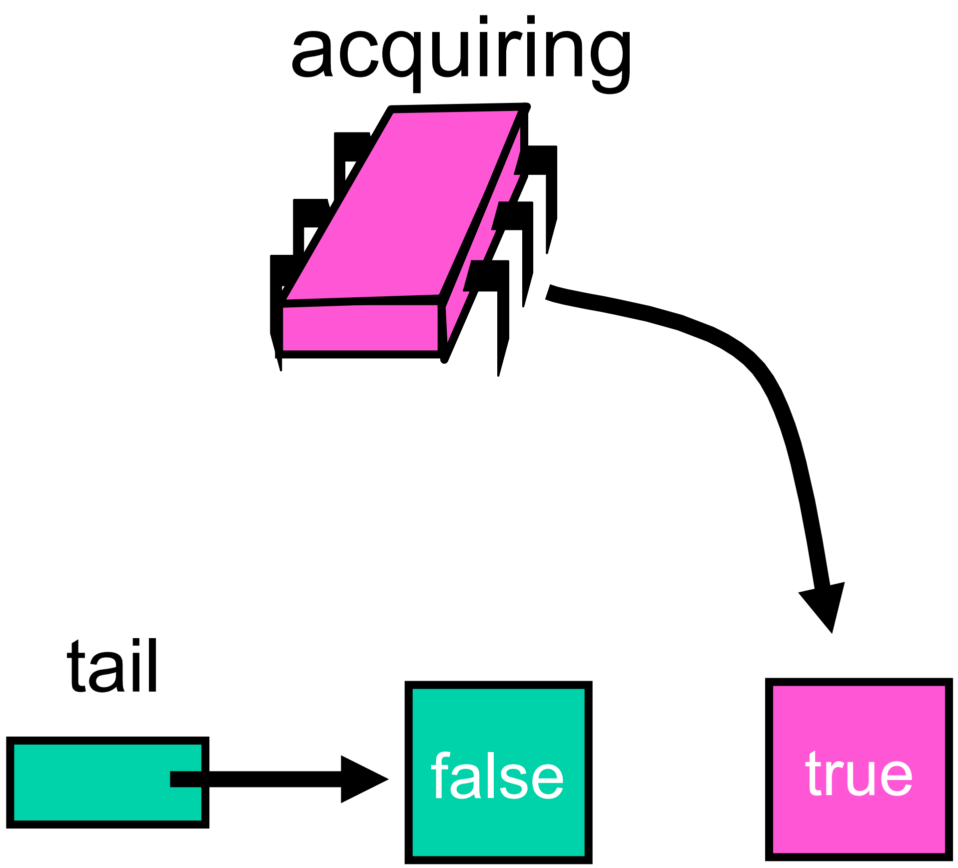
Initially



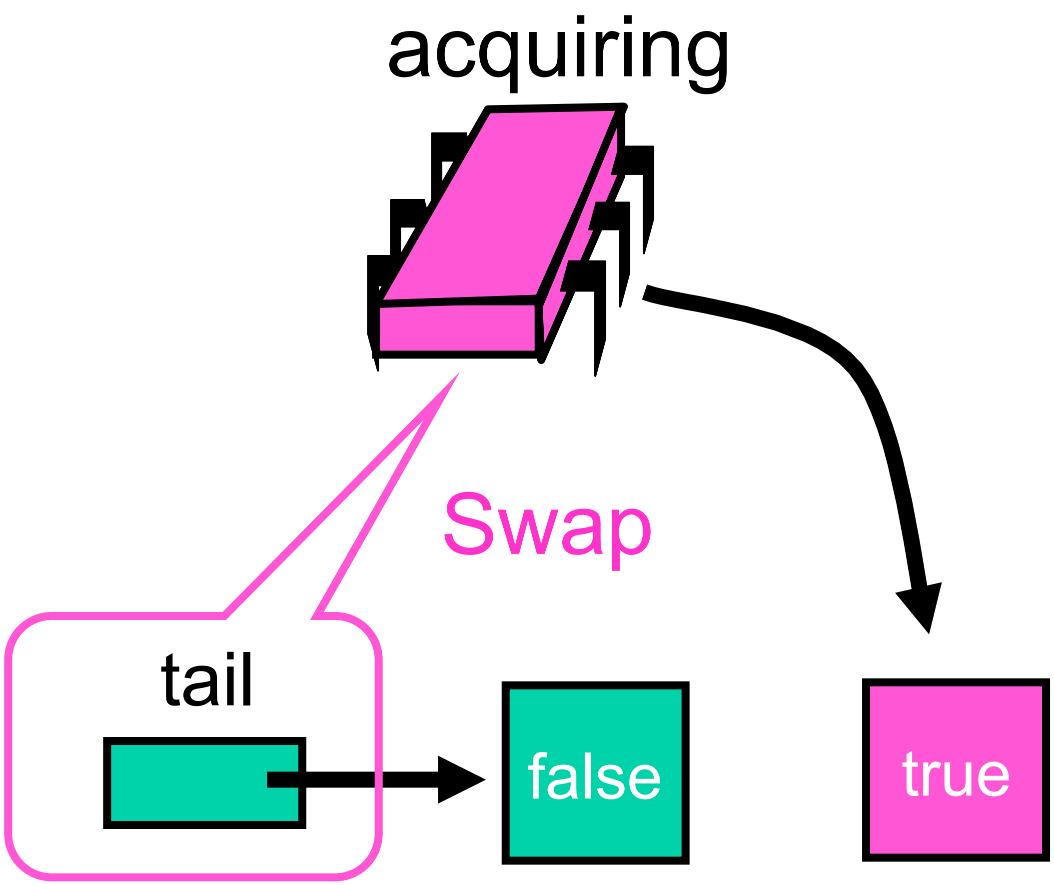
Purple Wants the Lock



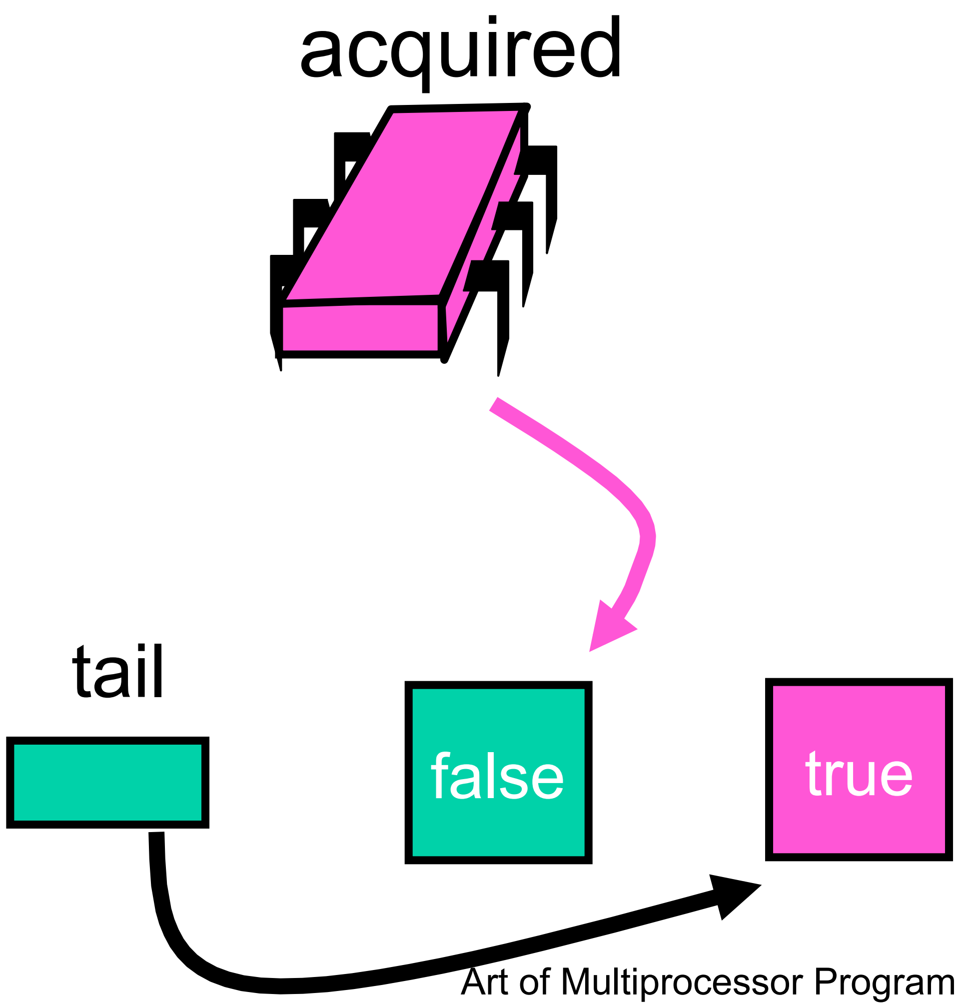
Purple Wants the Lock



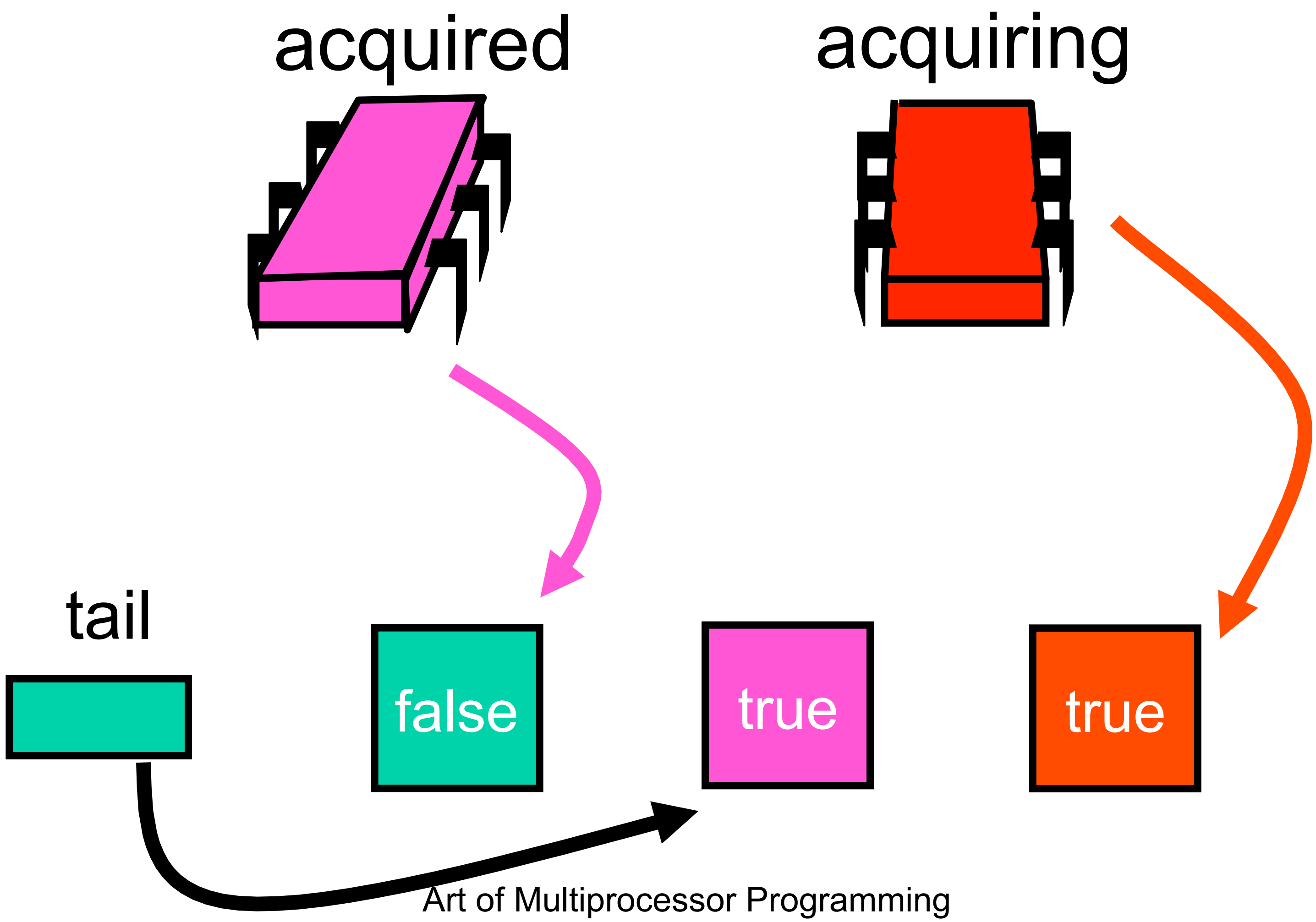
Purple Wants the Lock



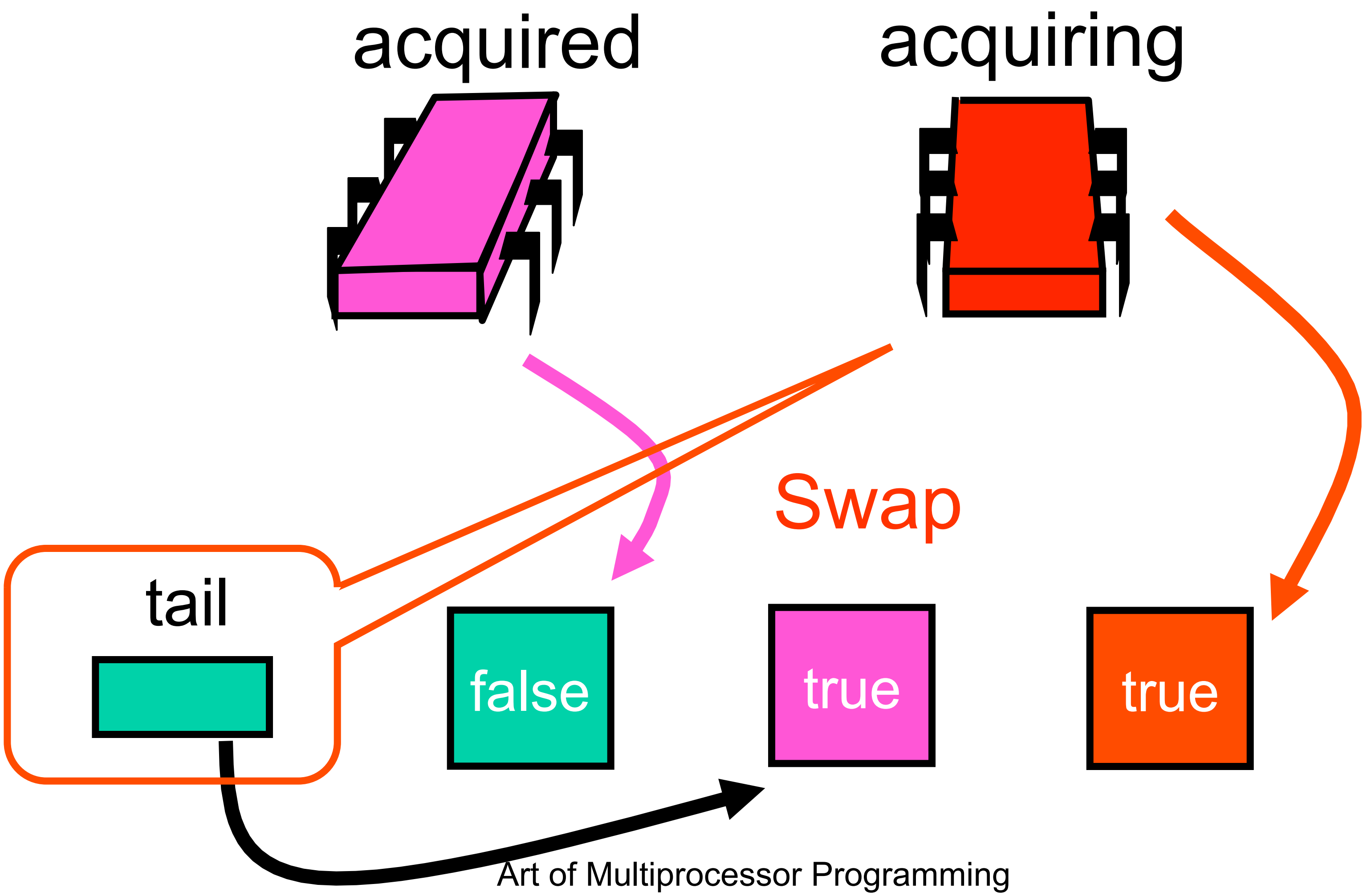
Purple Has the Lock



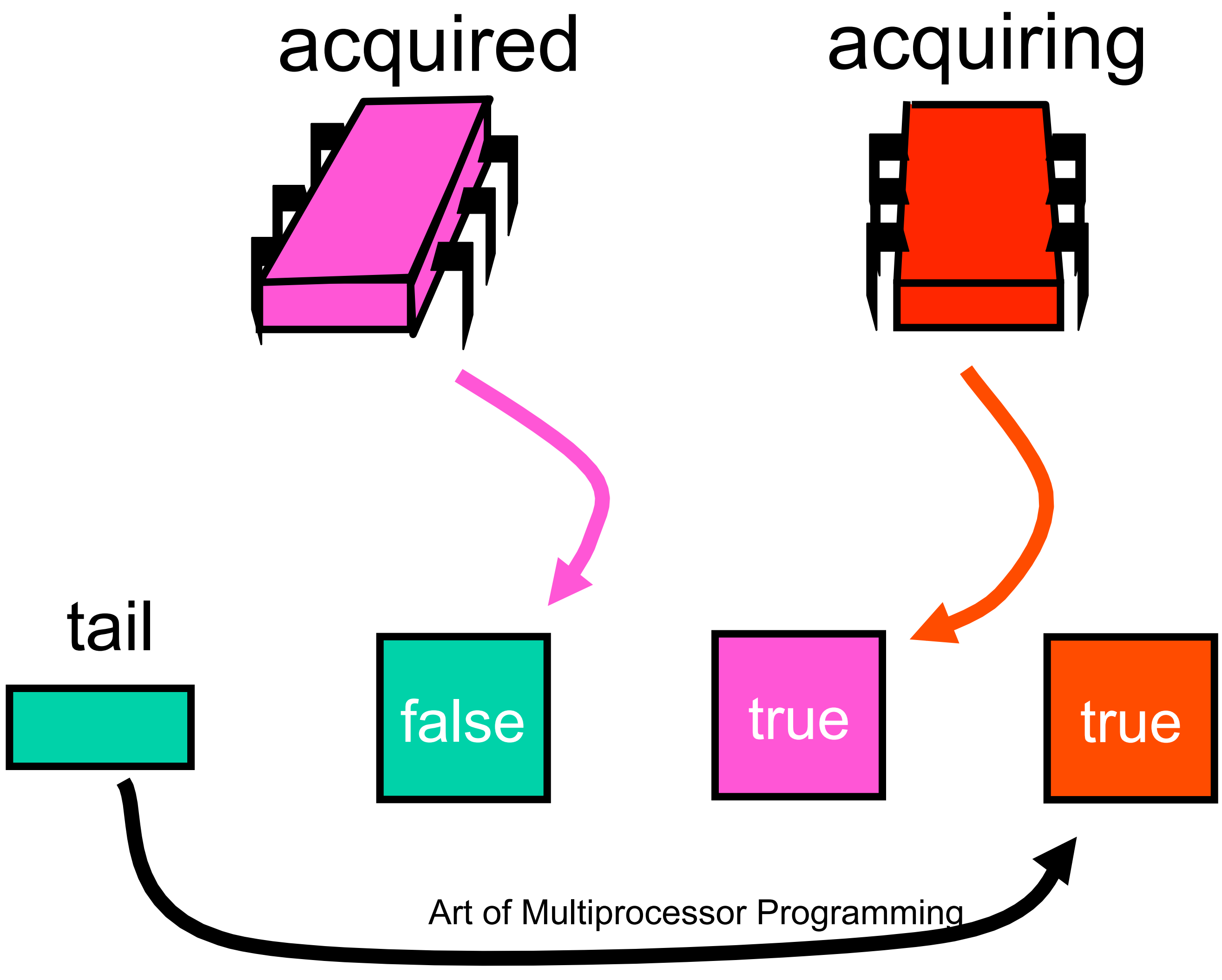
Red Wants the Lock



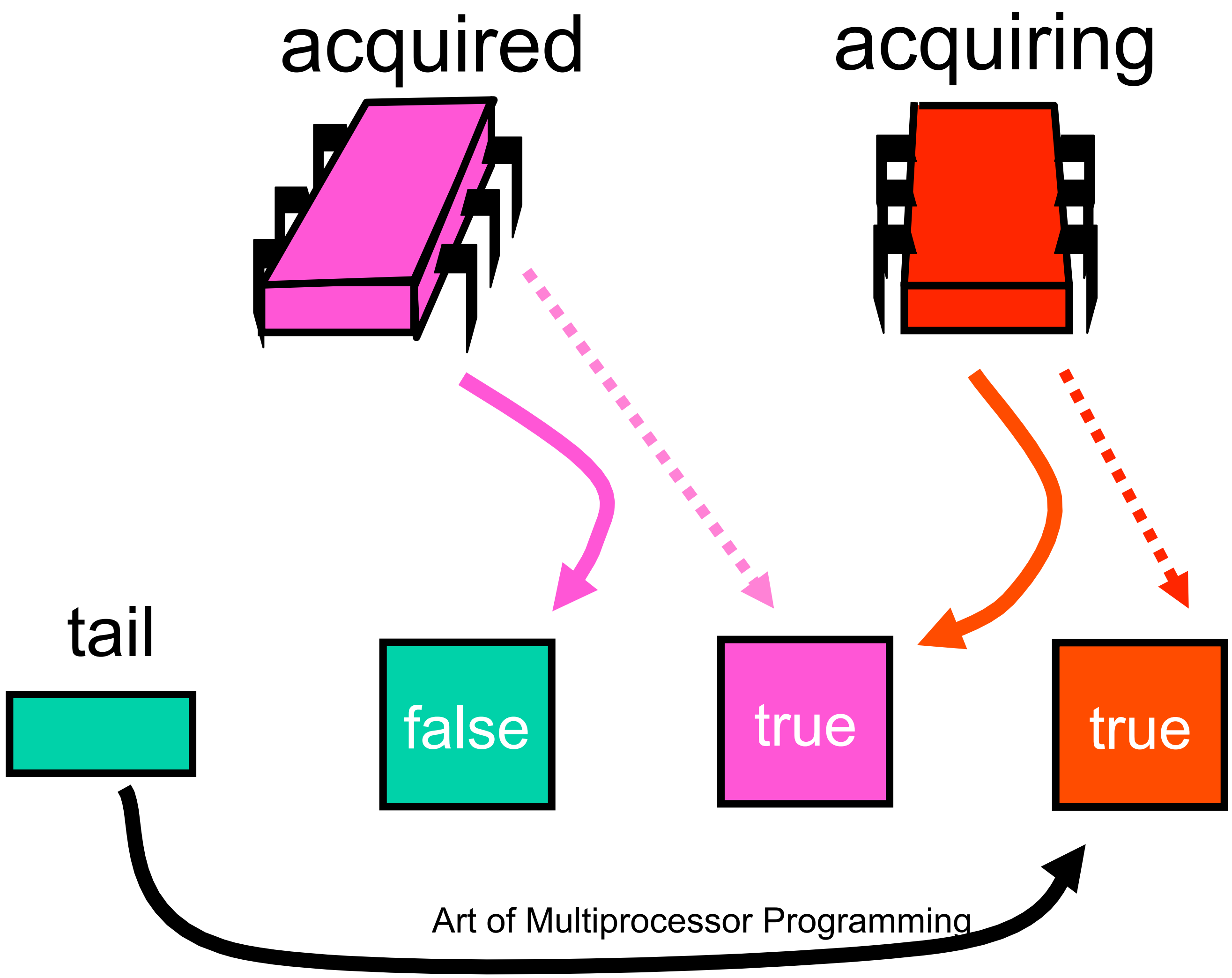
Red Wants the Lock



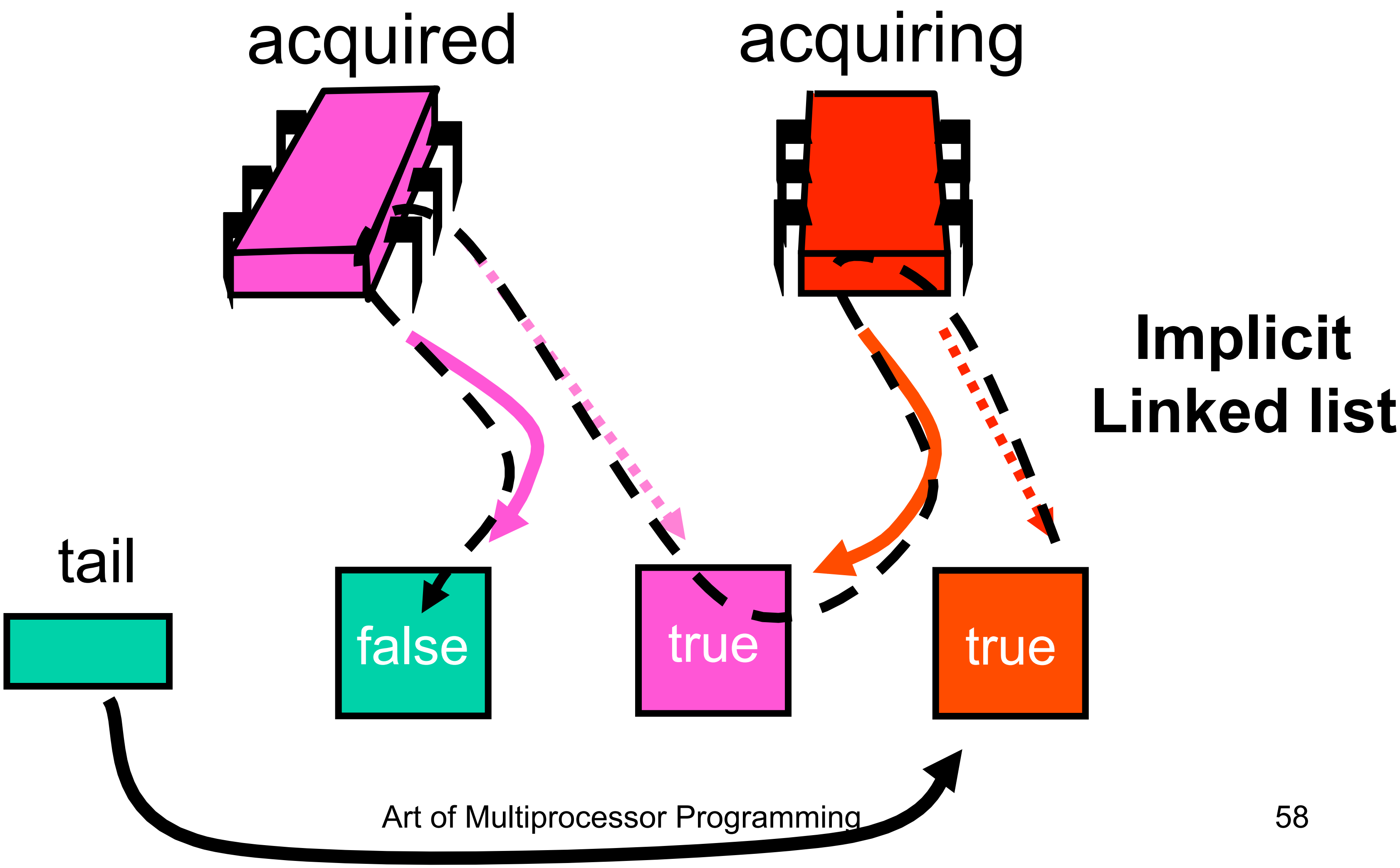
Red Wants the Lock



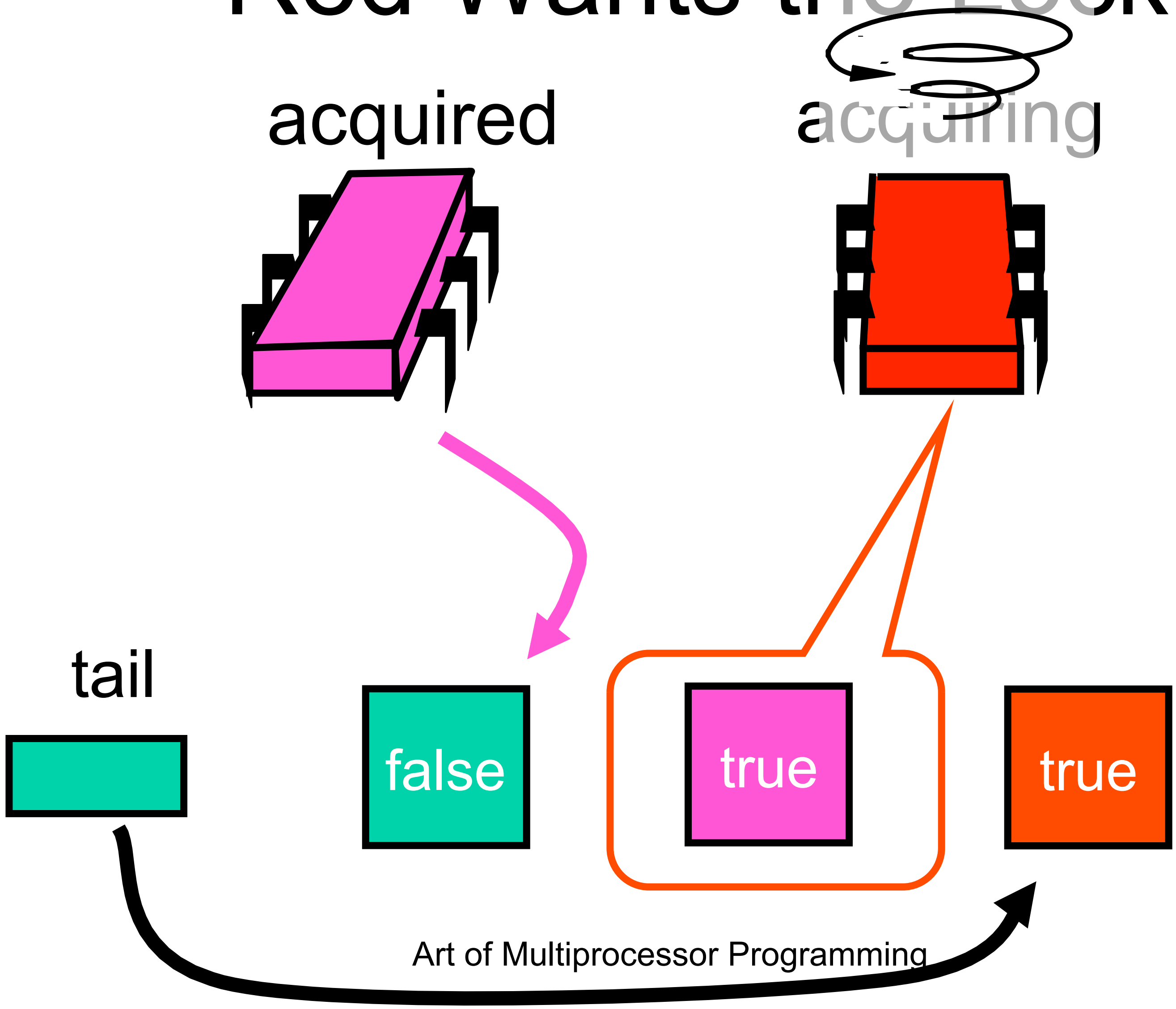
Red Wants the Lock



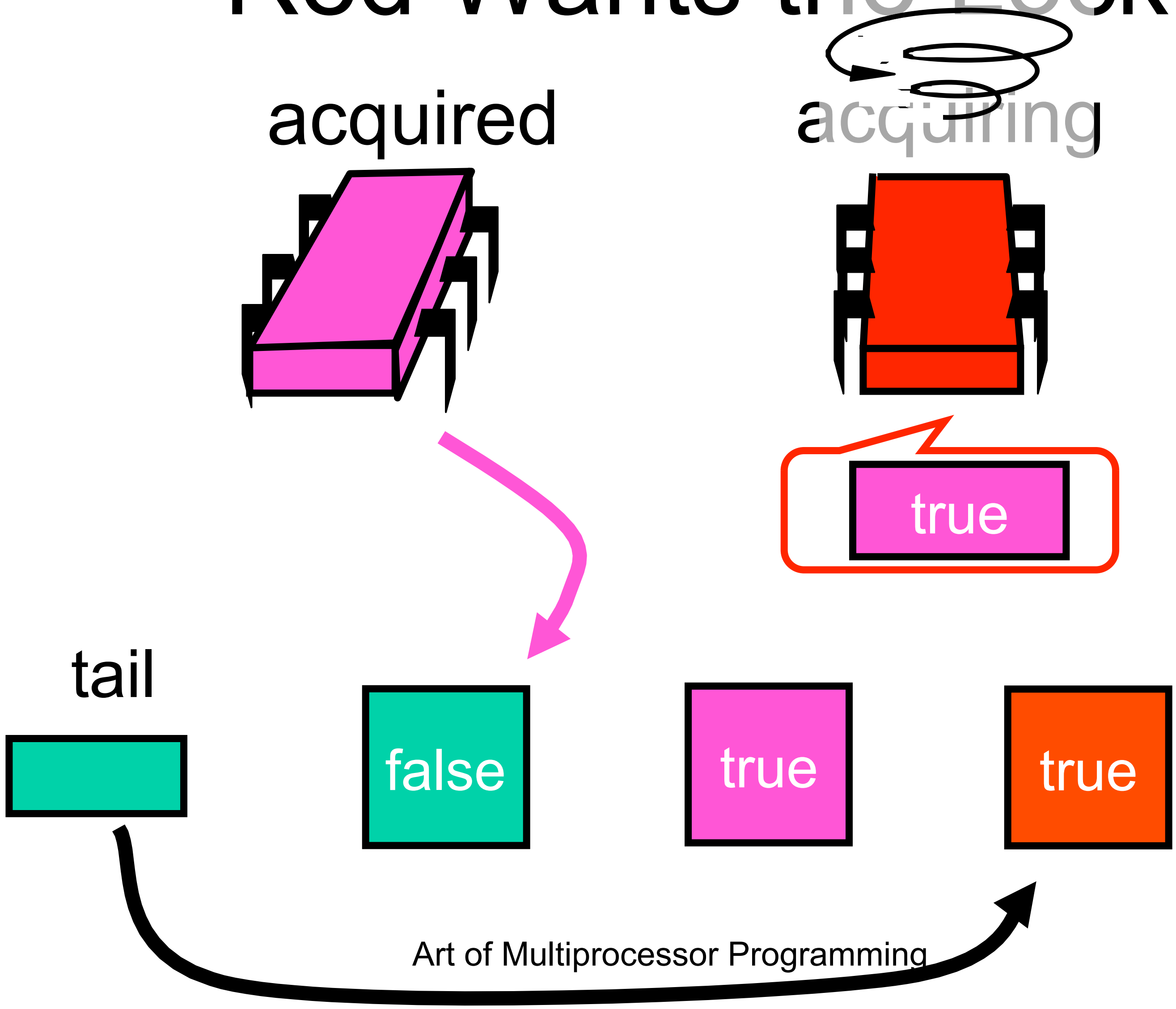
Red Wants the Lock



Red Wants the Lock

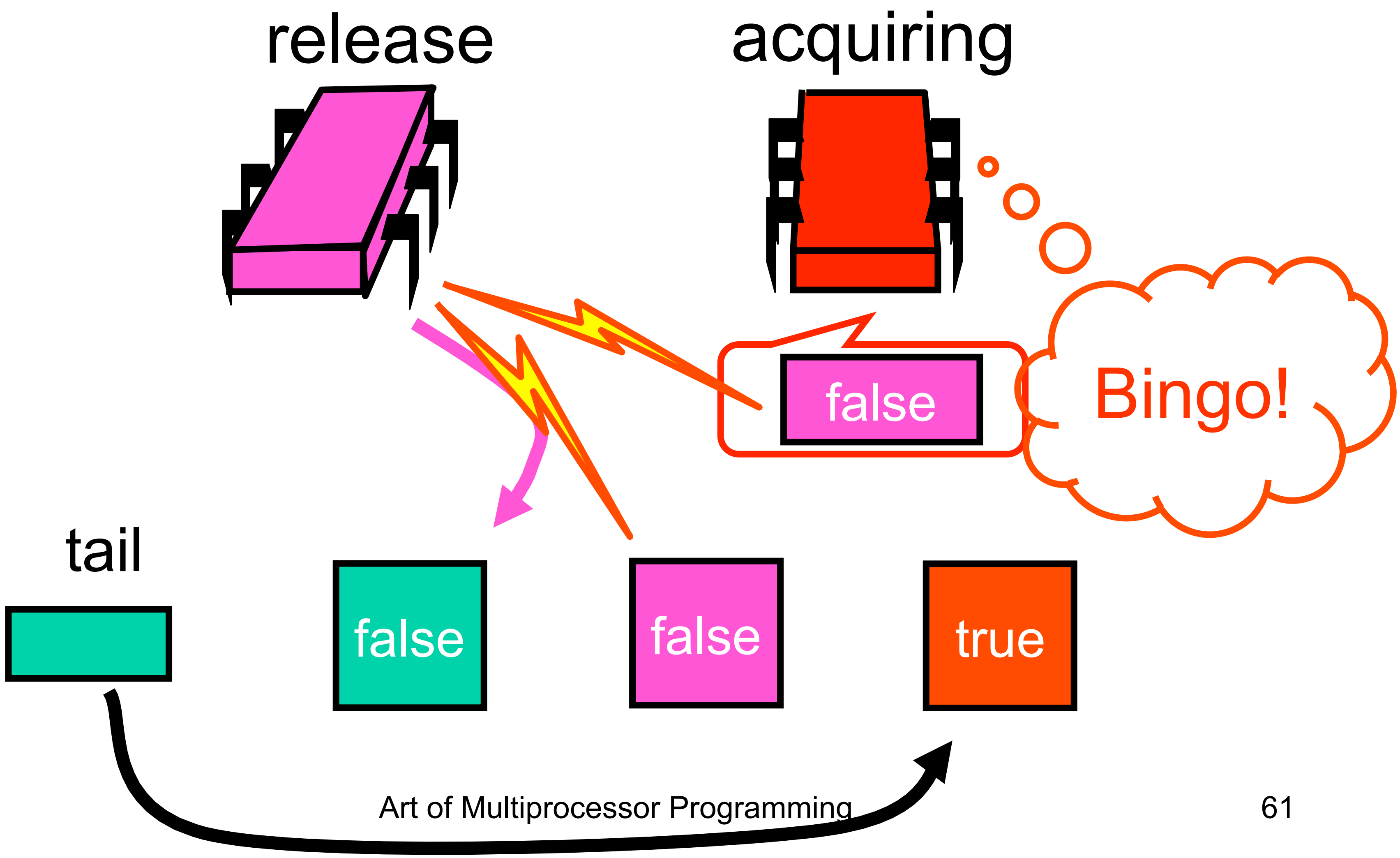


Red Wants the Lock

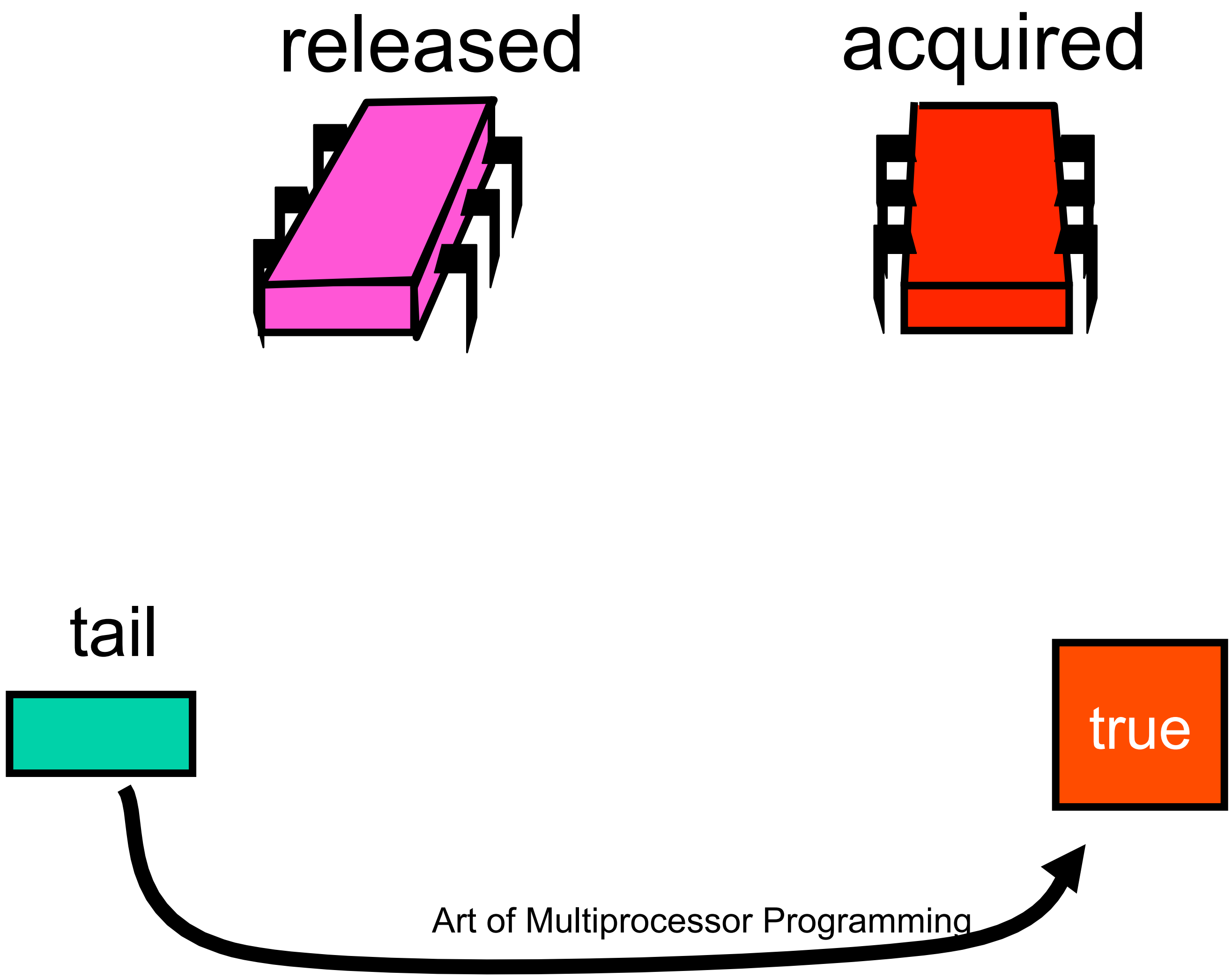


Actually, it spins on cached copy

Purple Releases



Purple Releases



CLH Queue Lock

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

CLH Queue Lock

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

Not released yet

CLH Queue Lock

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

CLH Queue Lock

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

Queue tail

CLH Queue Lock

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

Thread-local Qnode

CLH Queue Lock

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

Swap in my node

CLH Queue Lock

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

Spin until predecessor releases lock

CLH Queue Lock

```
Class CLHLock implements Lock {
 …
 public void unlock() {
  myNode.locked.set(false);
  myNode = pred;
 }
}
```

CLH Queue Lock

```
Class CLHLock implements Lock {
 …
 public void unlock() {
  myNode.locked.set(false);
  myNode = pred;
 }
}
```

Notify successor

CLH Queue Lock

```
Class CLHLock implements Lock {
 …
 public void unlock() {
  myNode.locked.set(false);
  myNode = pred;
 }
}
```

Recycle predecessor's node

CLH Queue Lock

```
Class CLHLock implements Lock {
 …
 public void unlock() {
  myNode.locked.set(false);
  myNode = pred;
 }
}
```

(Here we don't actually reuse myNode. Can see how it's done in Art of Multiprocessor Programming book)

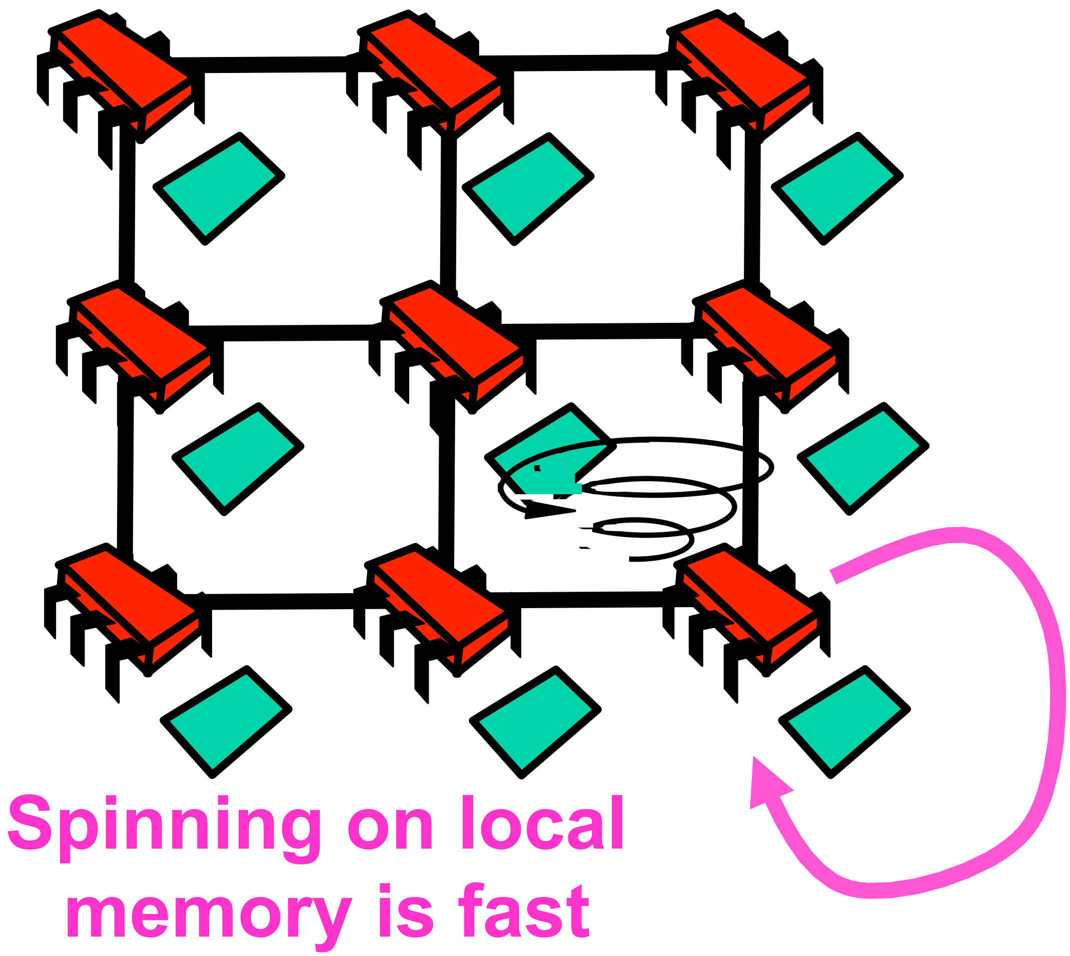
CLH Lock

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

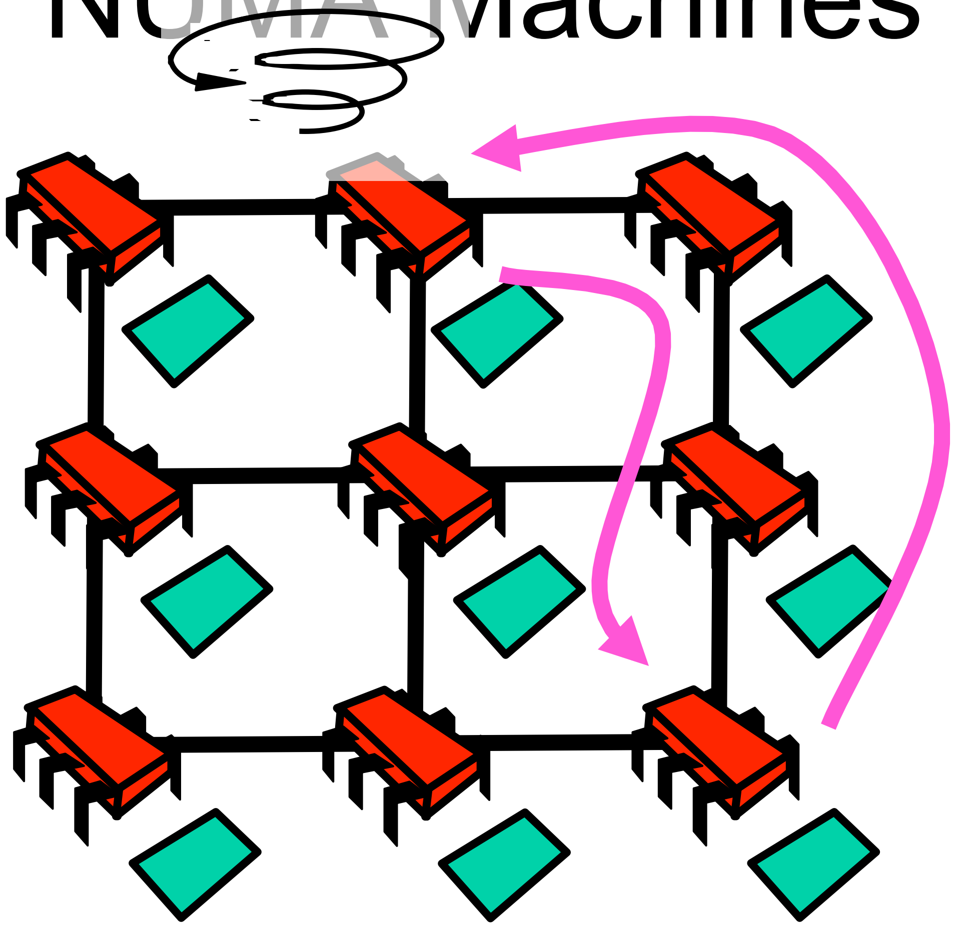
NUMA and cc-NUMA Architectures

- Acronym:
  - **N**on-**U**niform **M**emory **A**rchitecture
  - ccNUMA = cache coherent NUMA
- Illusion:
  - Flat shared memory
- Truth:
  - No caches (sometimes)
  - Some memory regions faster than others

NUMA Machines



NUMA Machines



Spinning on remote memory is slow

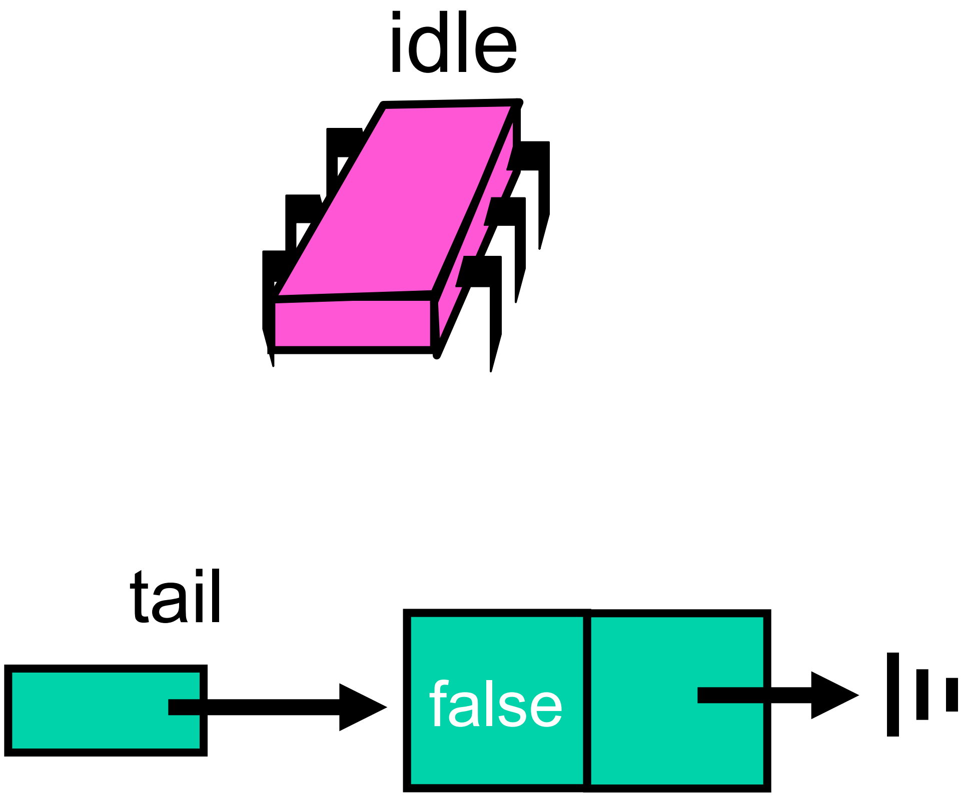
CLH Lock

- Each thread spins on predecessor's memory
- Could be far away …

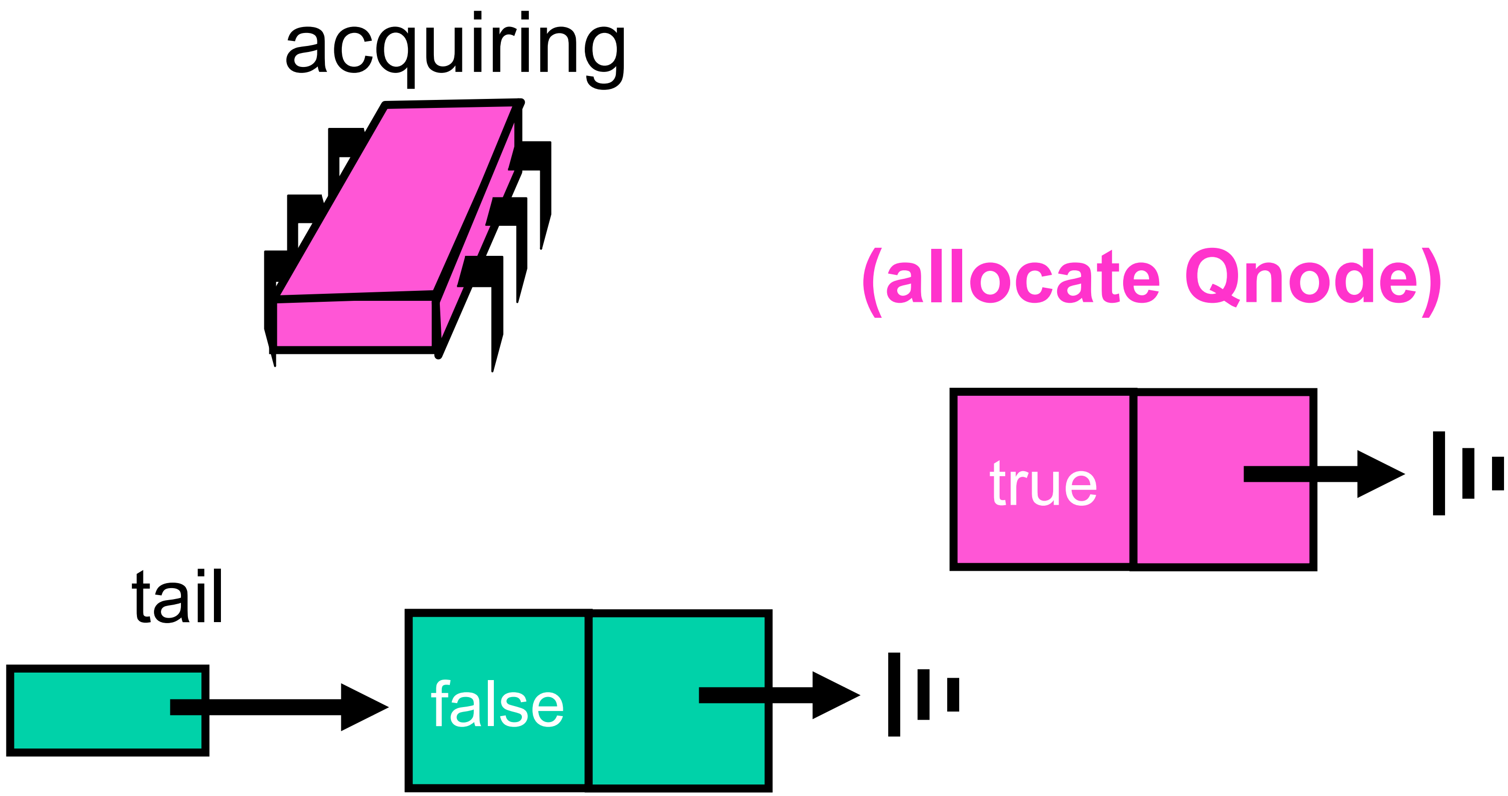
MCS Lock

- FCFS order
- Spin on local memory only
- Small, Constant-size overhead

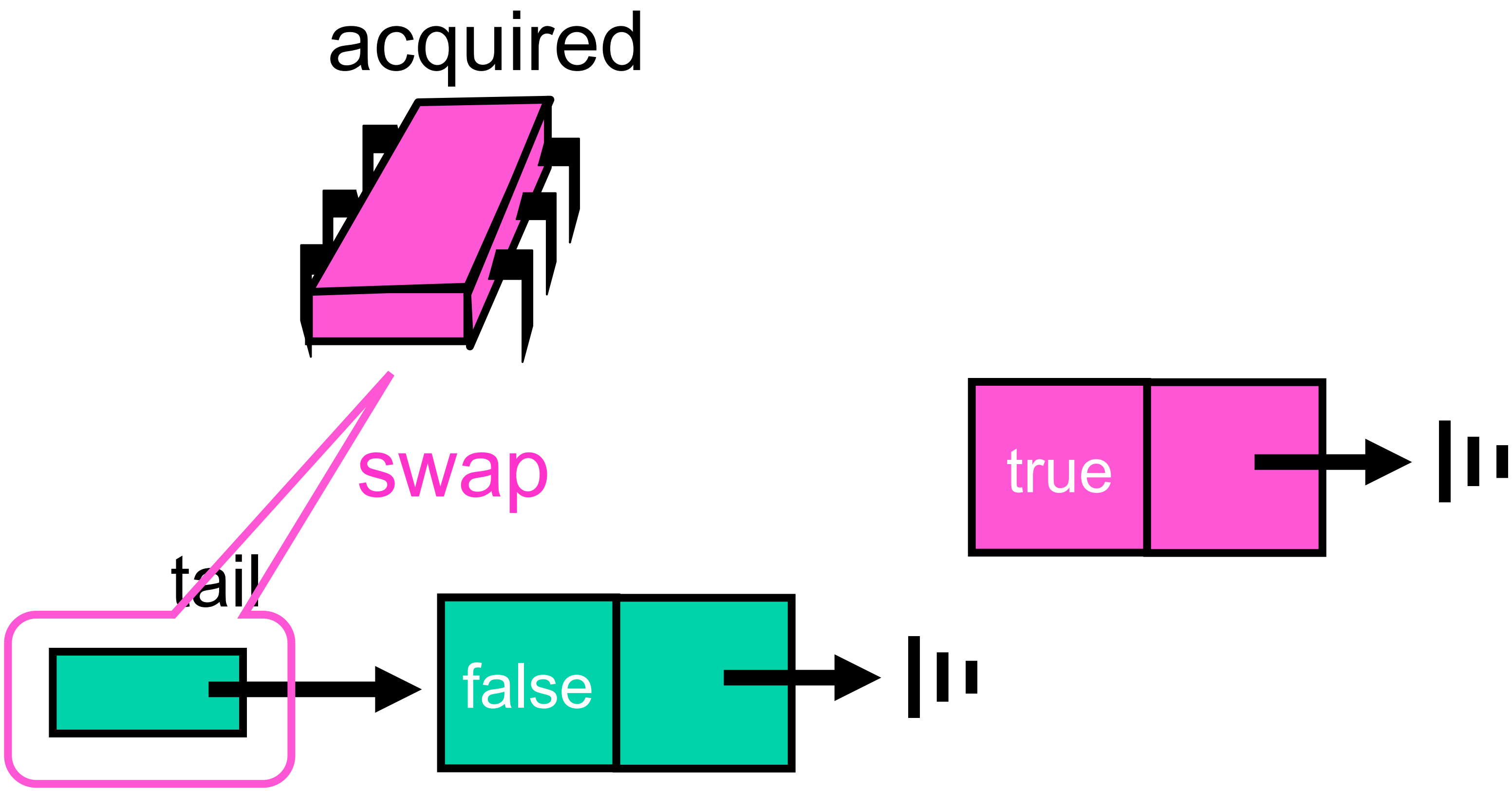
Initially



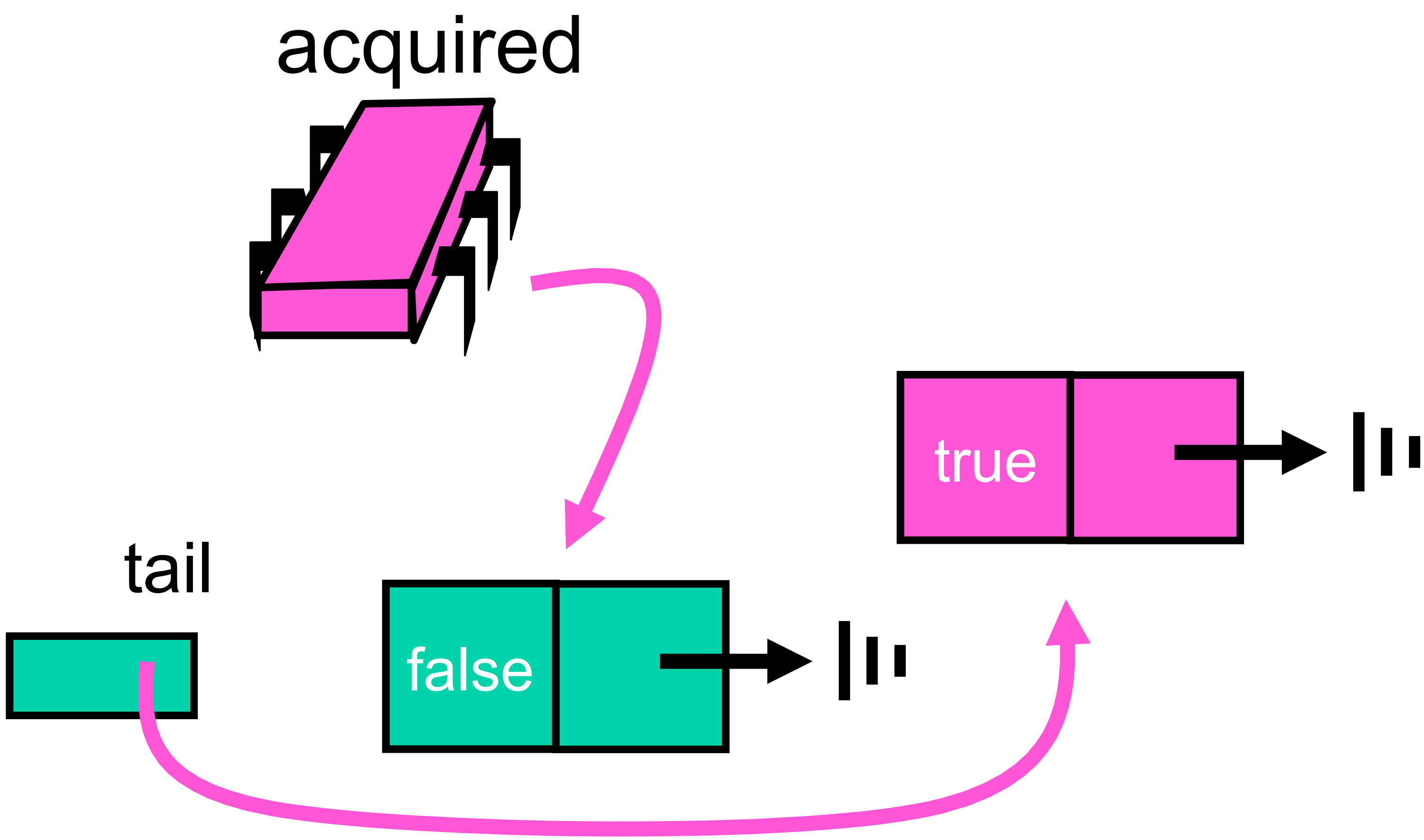
Acquiring



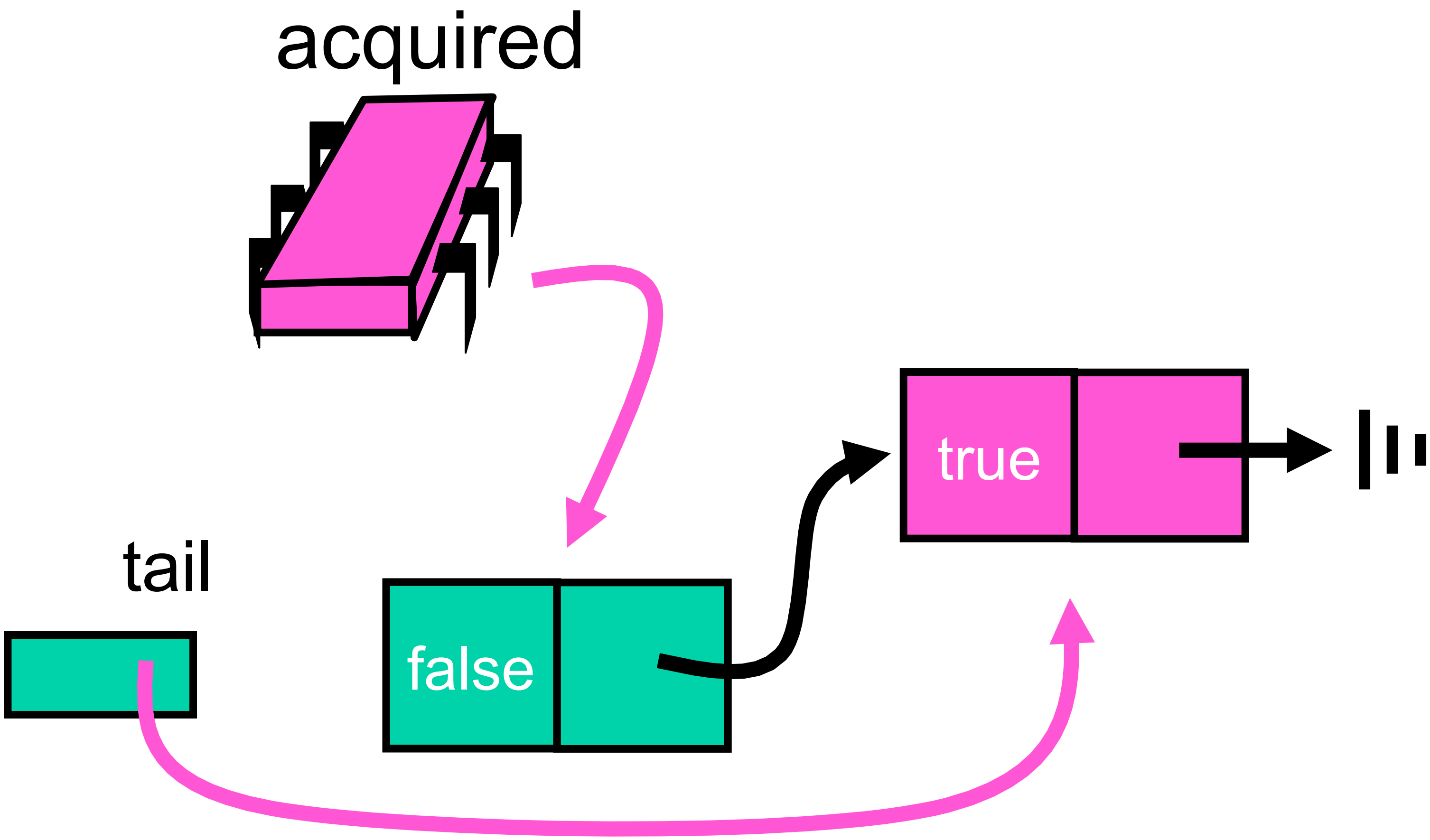
Acquiring



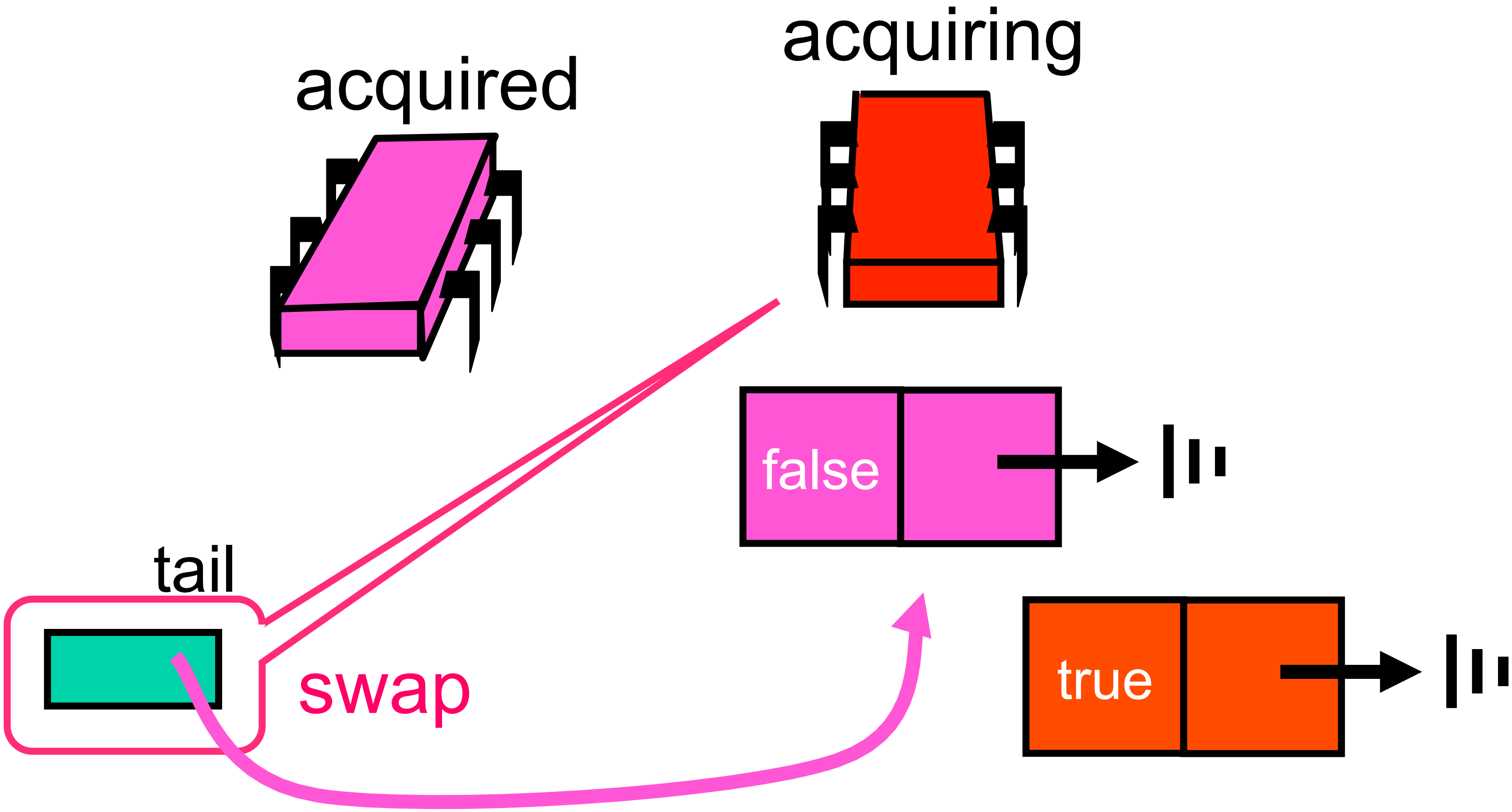
Acquiring



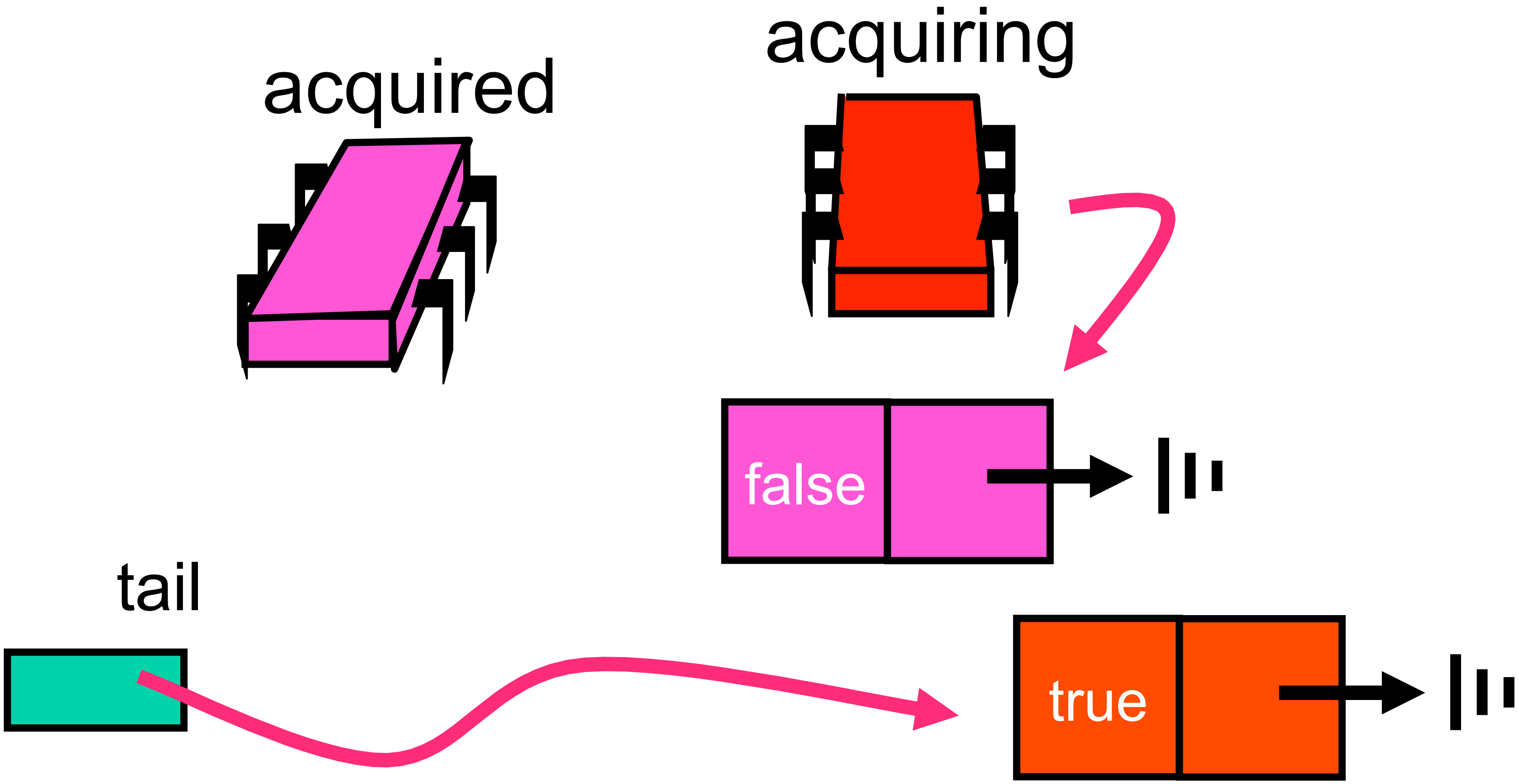
Acquired



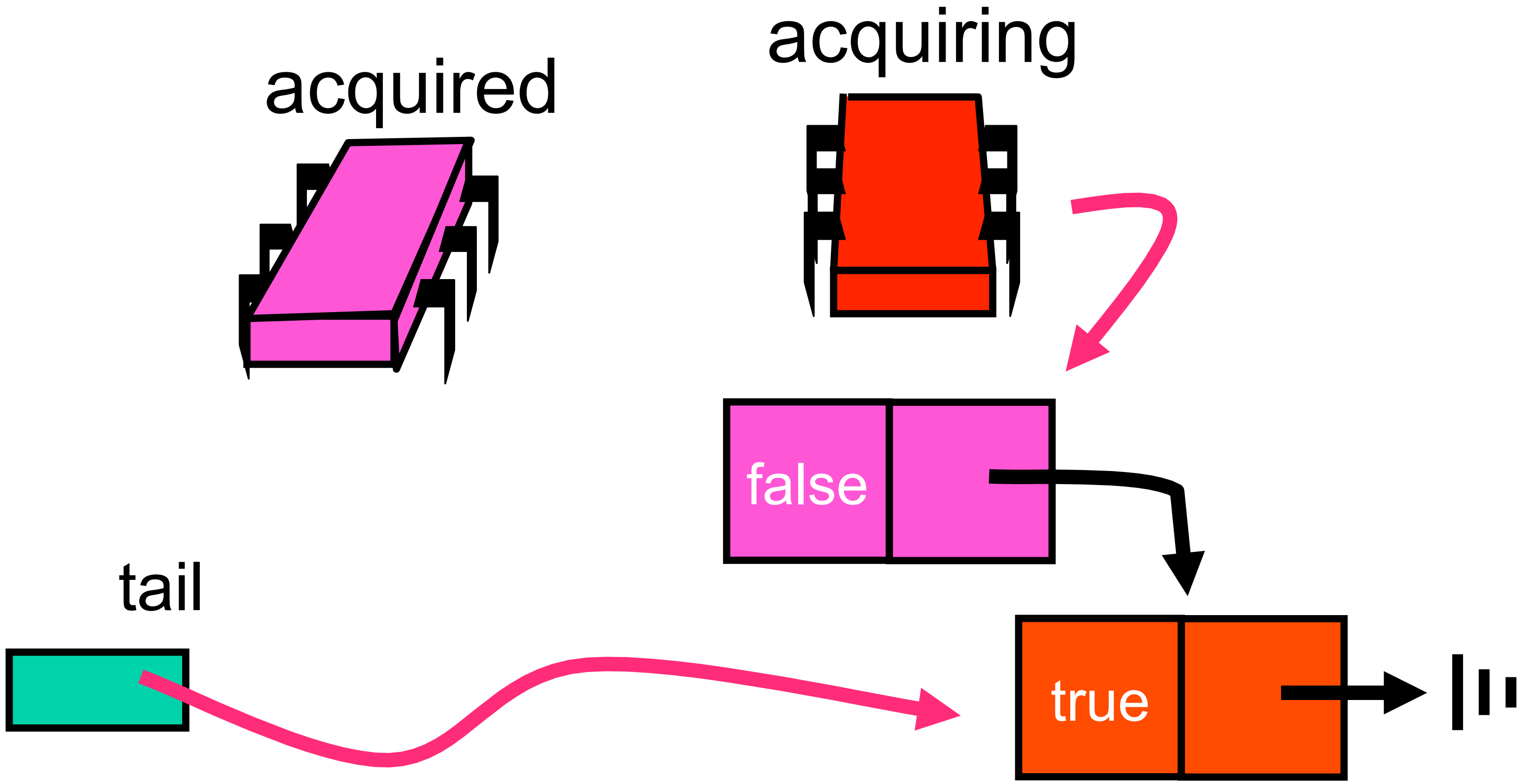
Acquiring



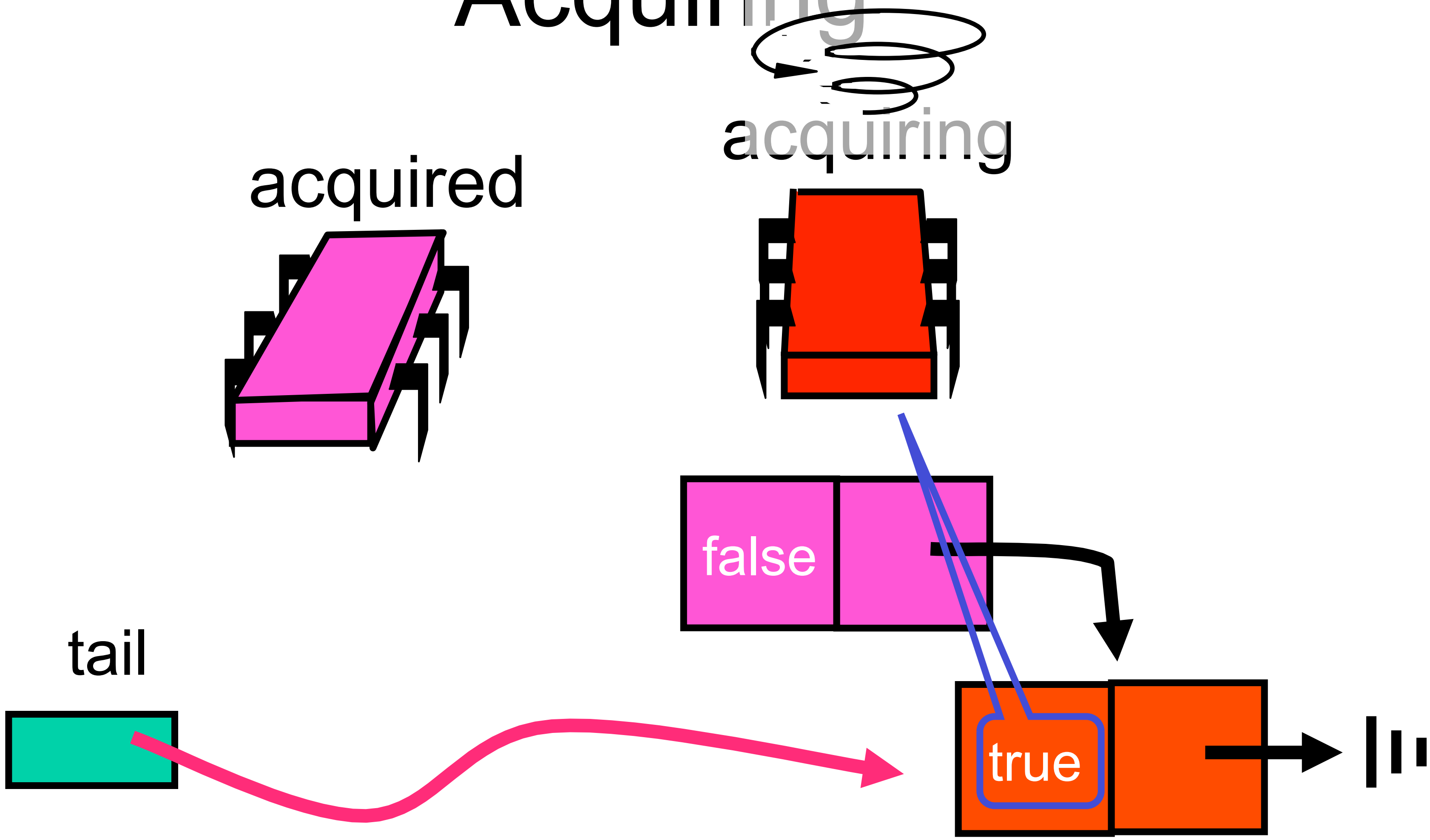
Acquiring



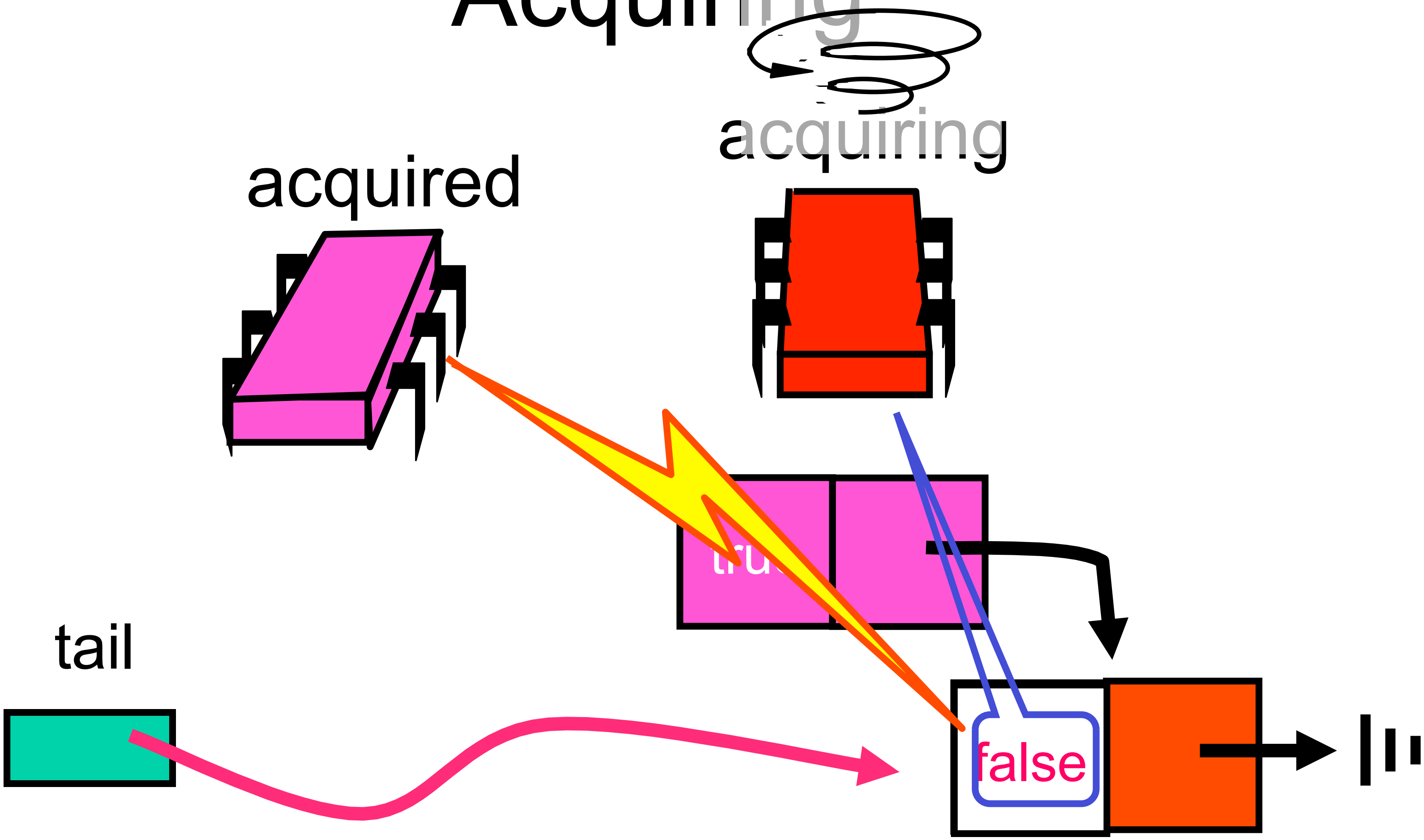
Acquiring



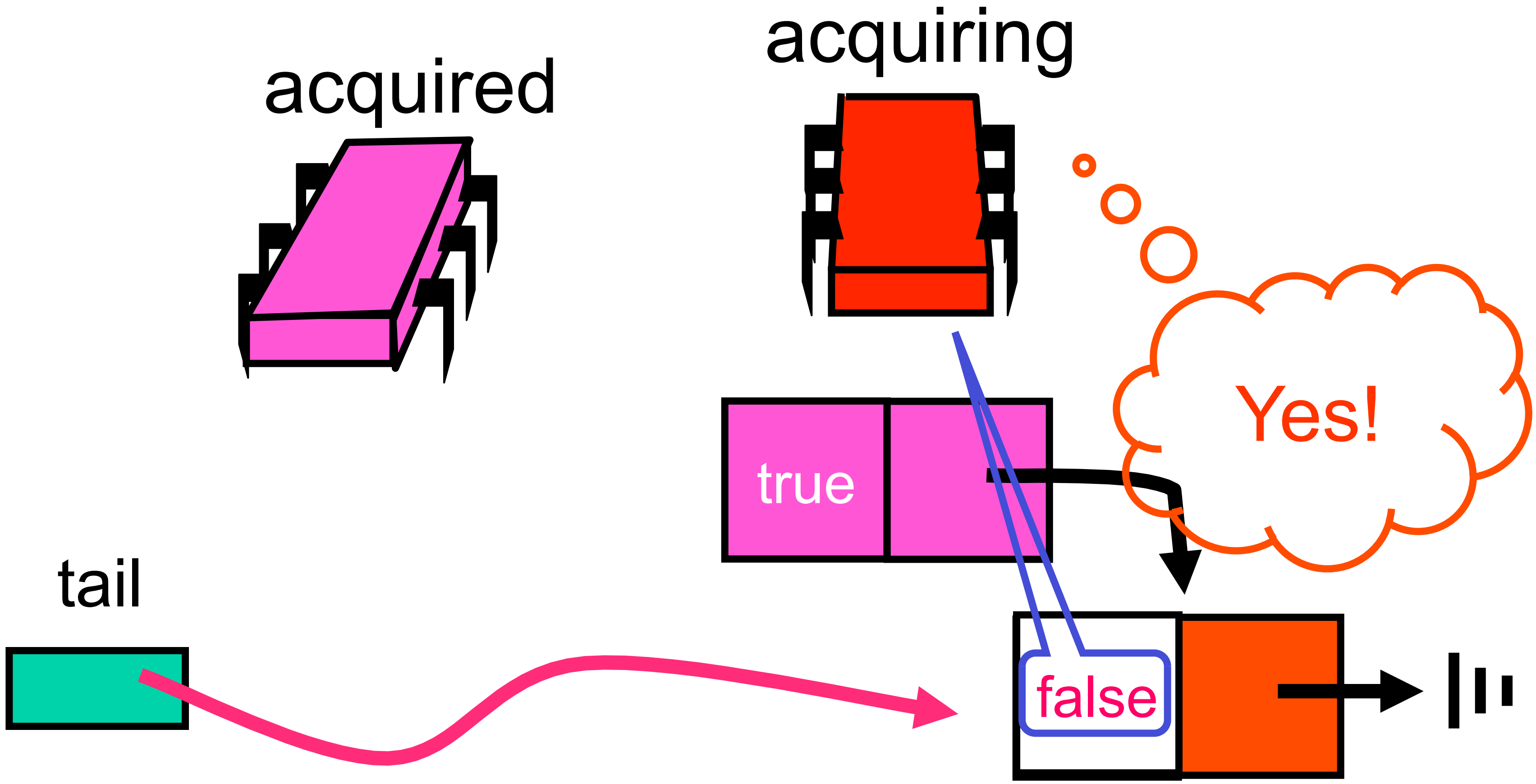
Acquiring



Acquiring



Acquiring



MCS Queue Lock

```
class Qnode {
 volatile boolean locked = false;
 volatile qnode   next   = null;
}
```

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

Make a QNode

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

add my Node to the tail of queue

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

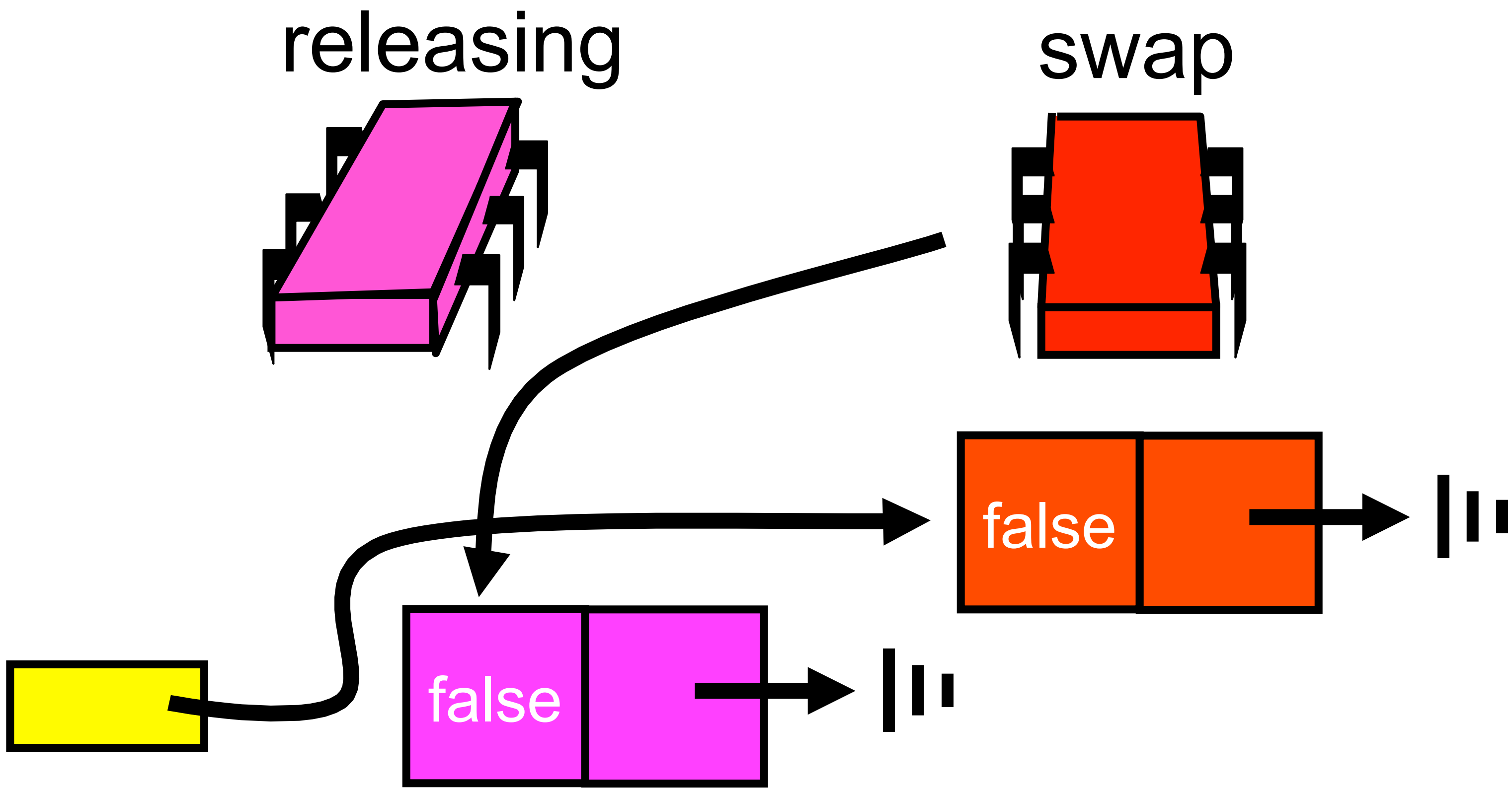
Fix if queue was non-empty

MCS Queue Lock

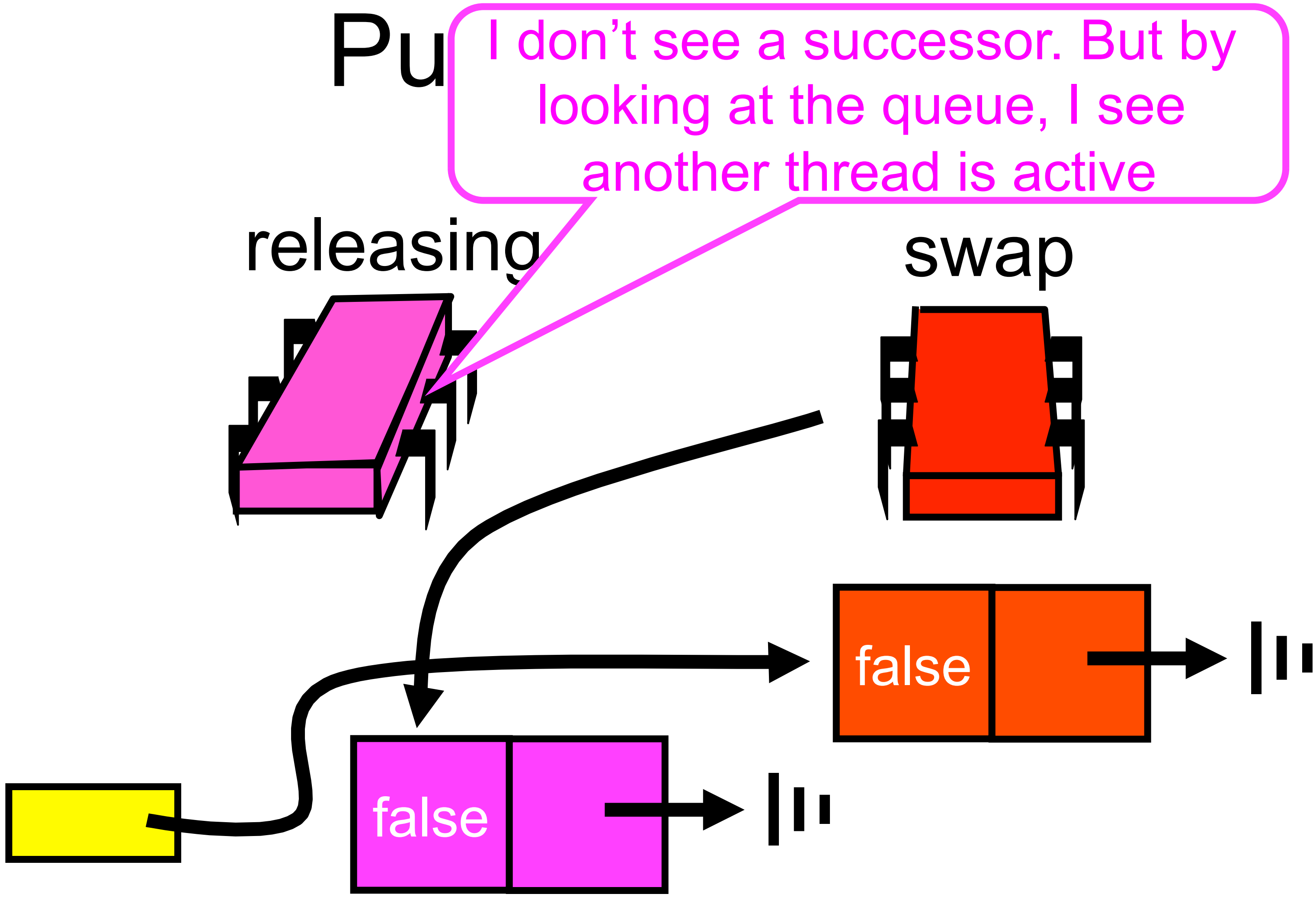
```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

Wait until unlocked

Purple Release

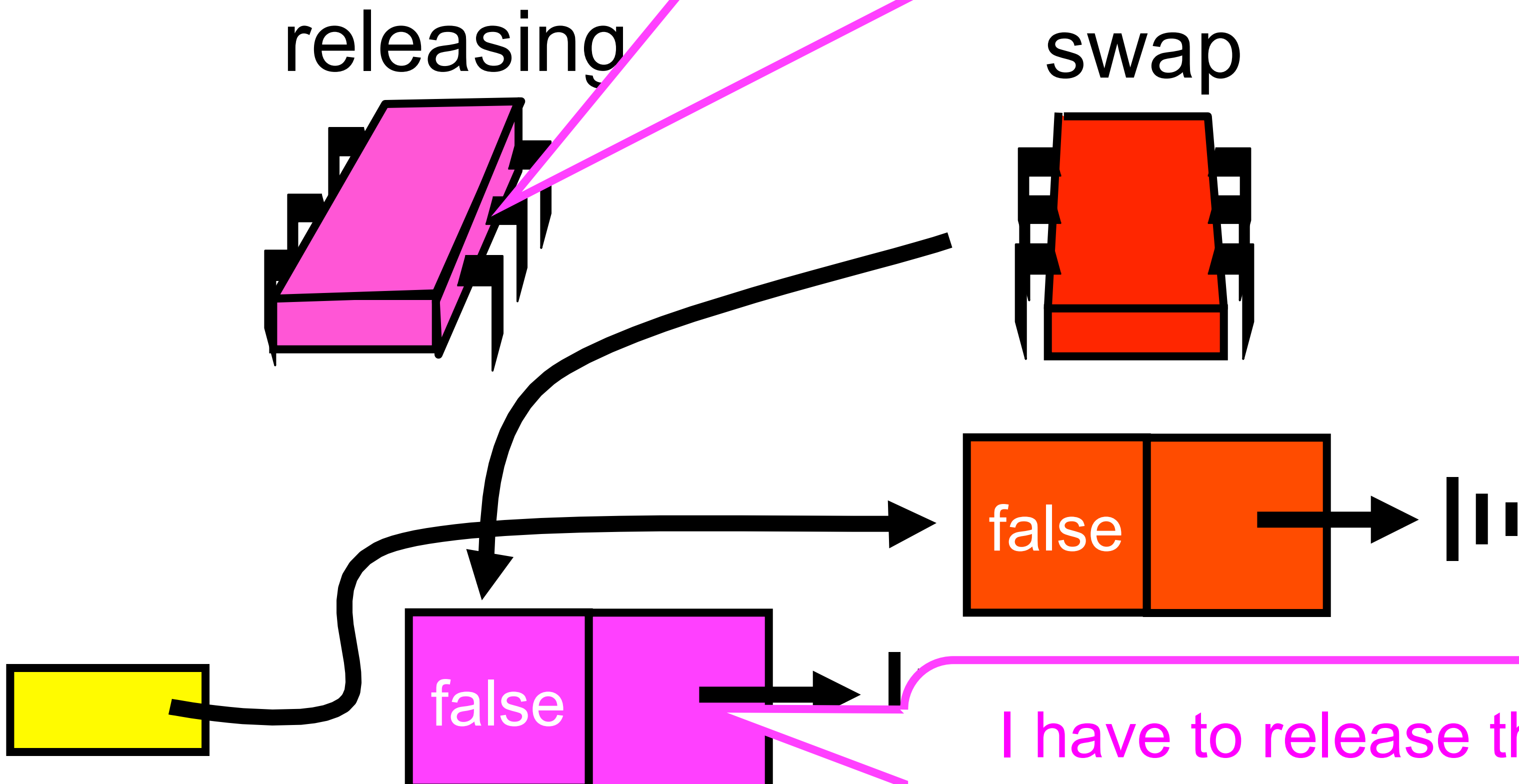


Pu


I don't see a successor. But by looking at the queue, I see another thread is active
releasing
swap
false
false

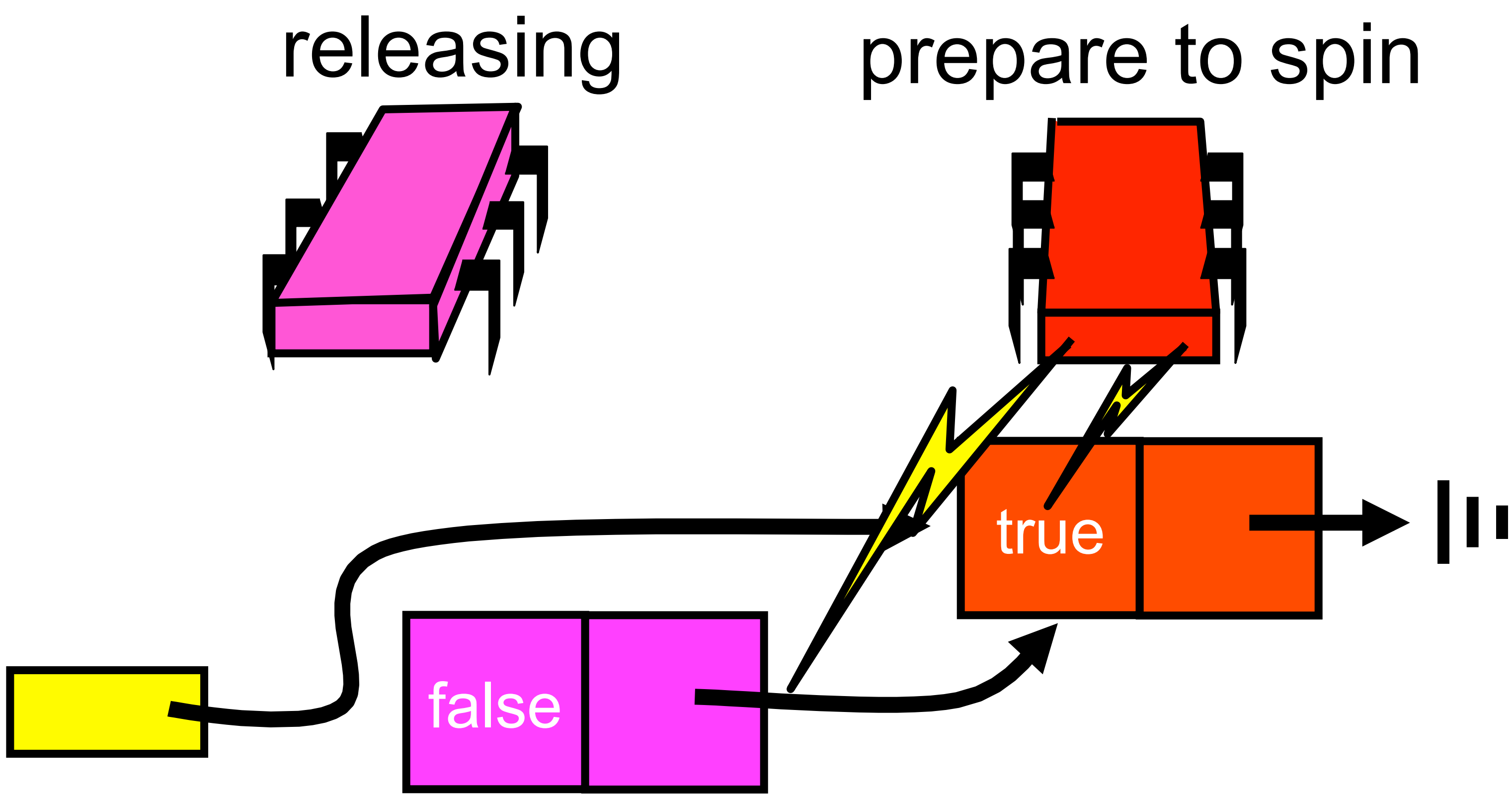
Pu

I don’t see a successor. But by looking at the queue, I see another thread is active

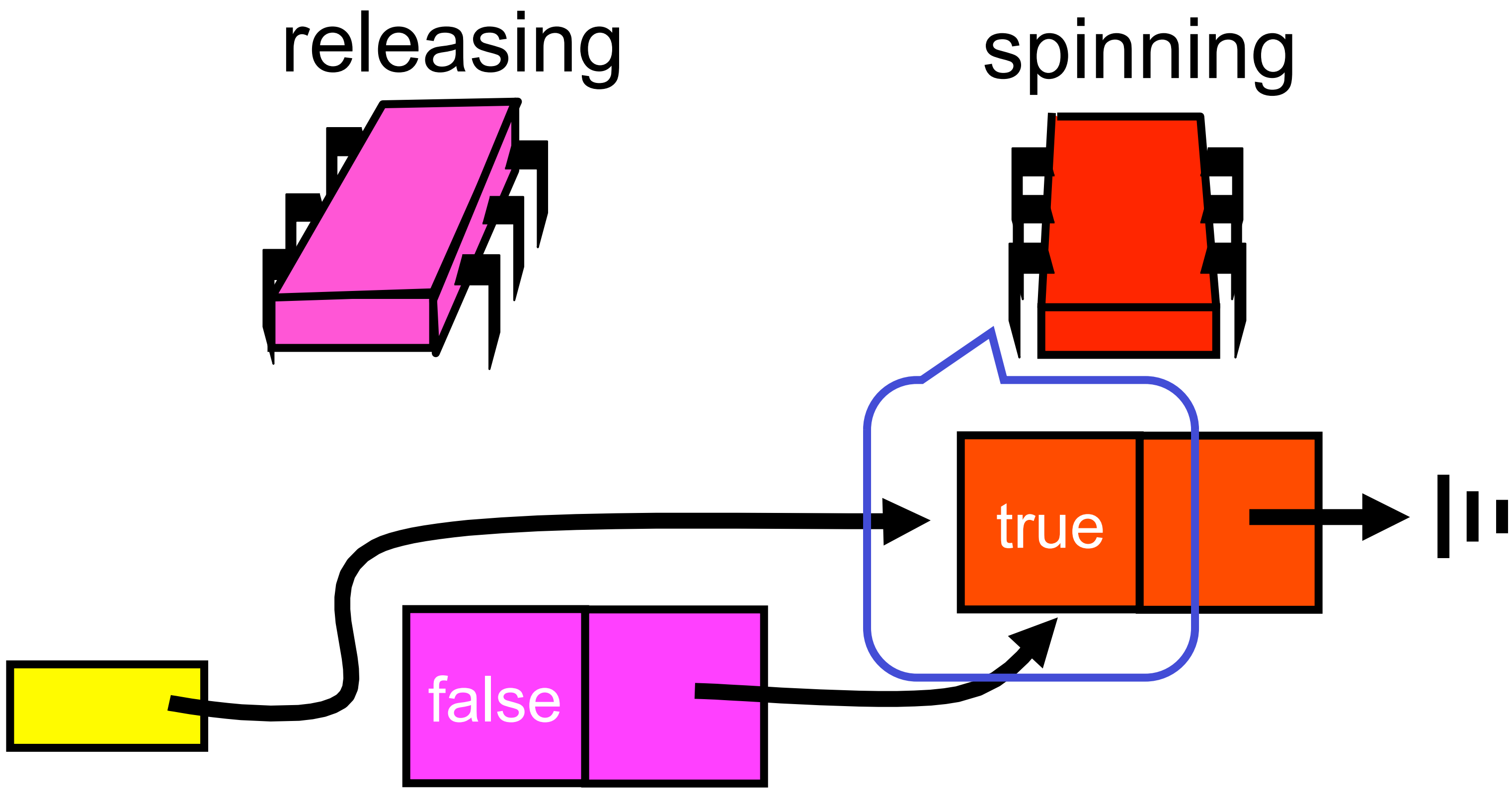


I have to release that thread so must wait for it to identify its node

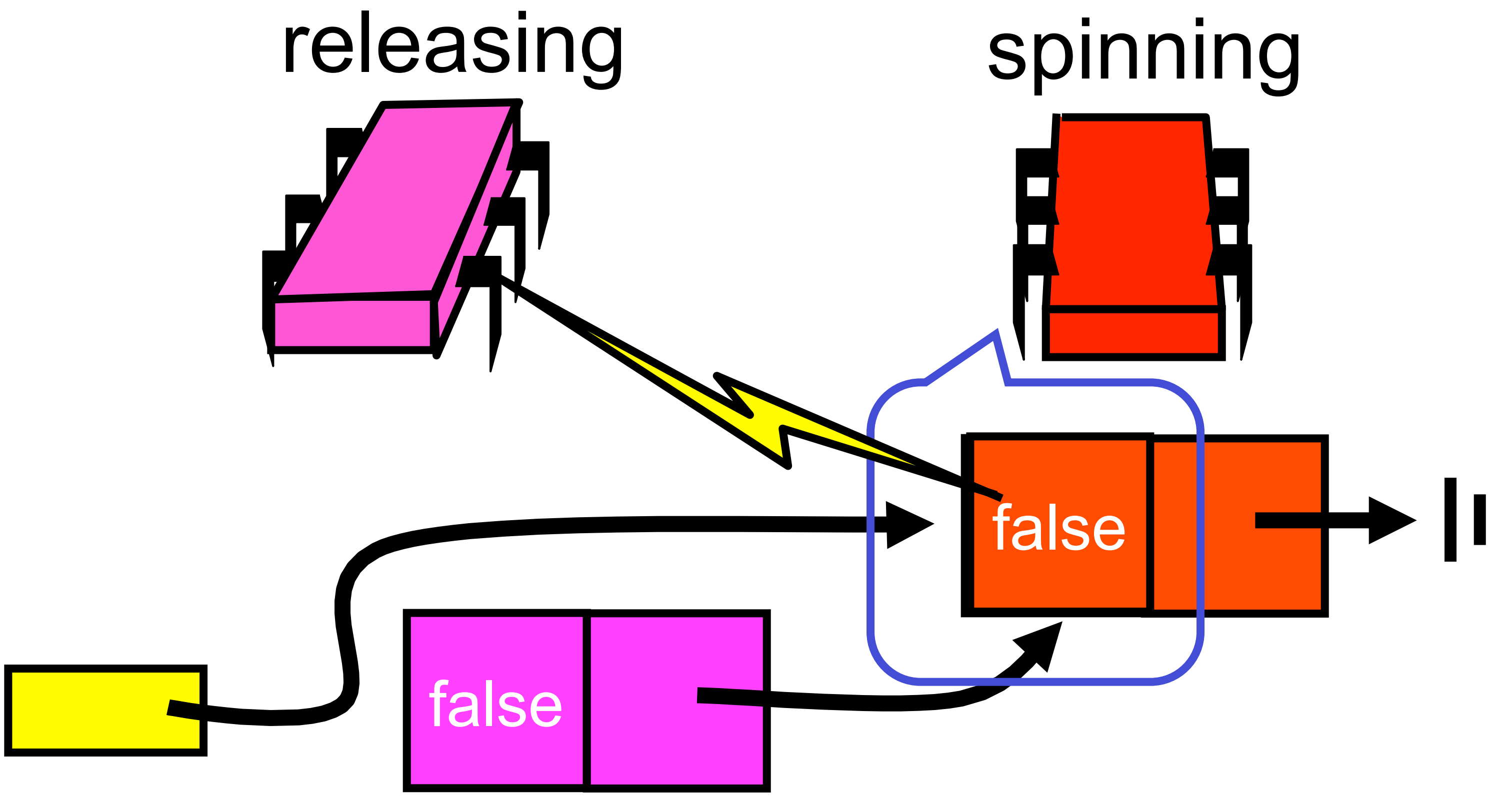
Purple Release



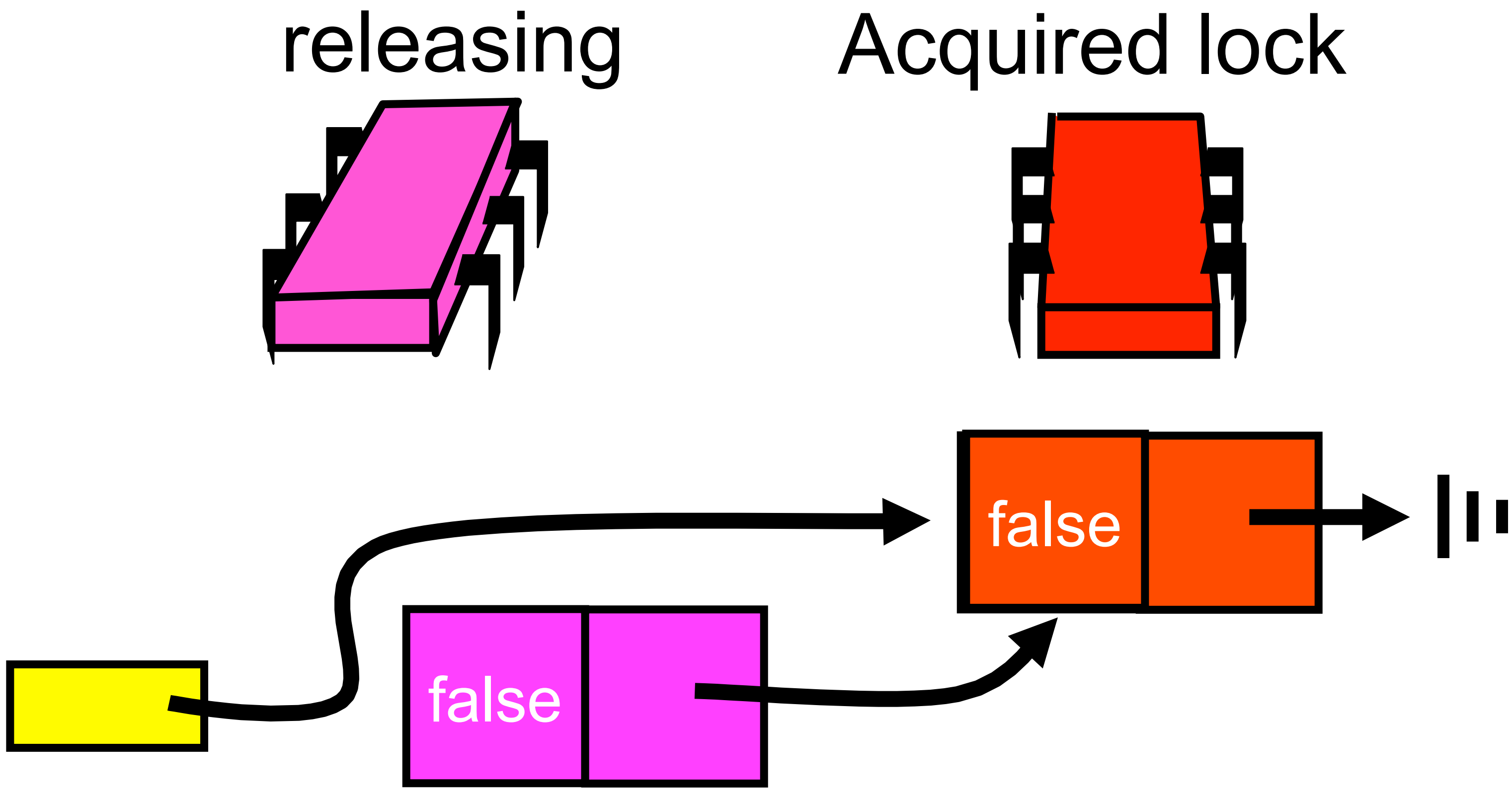
Purple Release



Purple Release



Purple Release



MCS Queue Unlock

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

MCS Queue Lock

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

Missing successor?

MCS Queue Lock

If really no successor, return

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

MCS Queue Lock

Otherwise wait for successor to catch up

```
                              :k {

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

MCS Queue Lock

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

Pass lock to successor

Abortable Locks

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

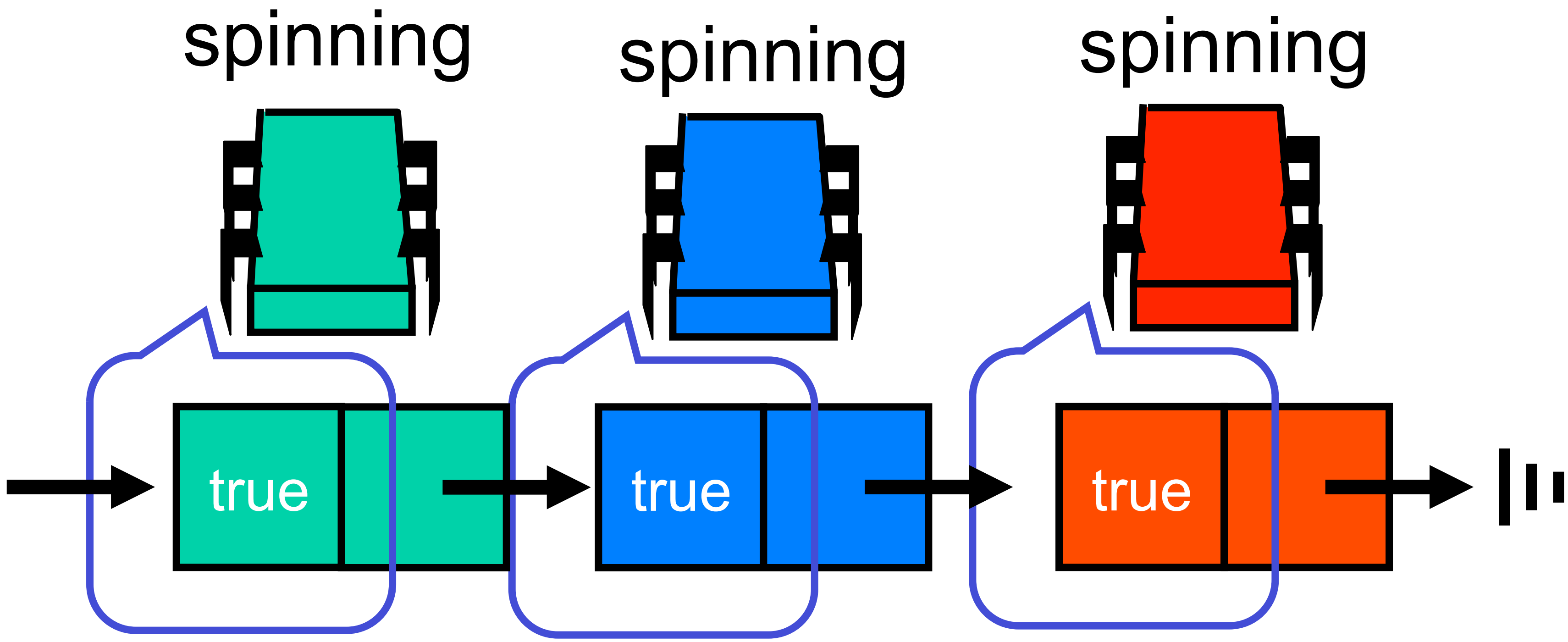
Back-off Lock

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

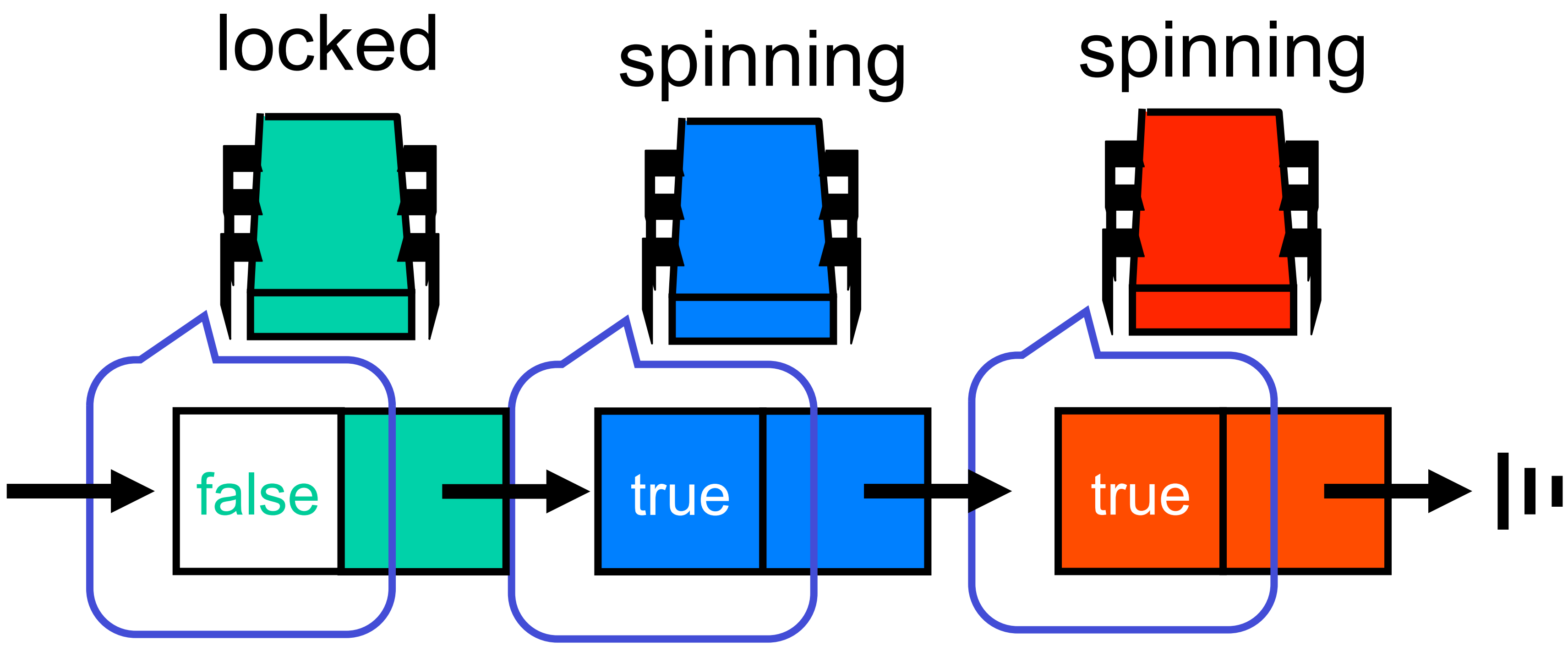
Queue Locks

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

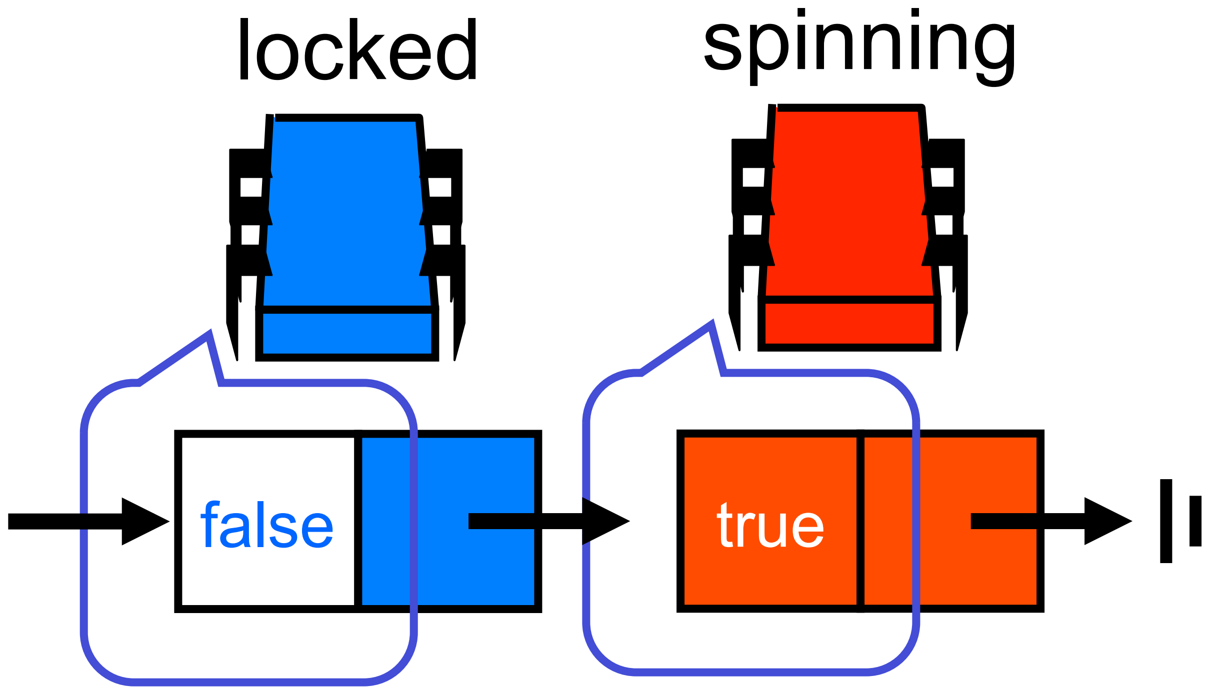
Queue Locks



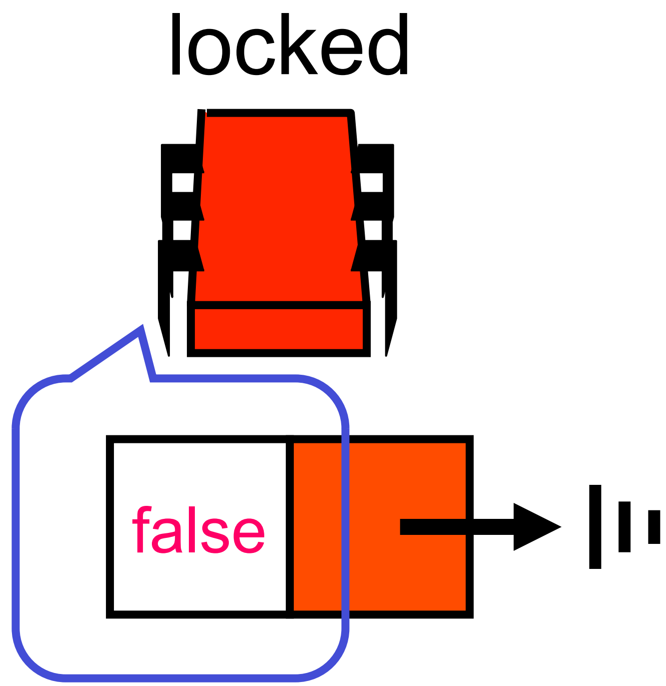
Queue Locks



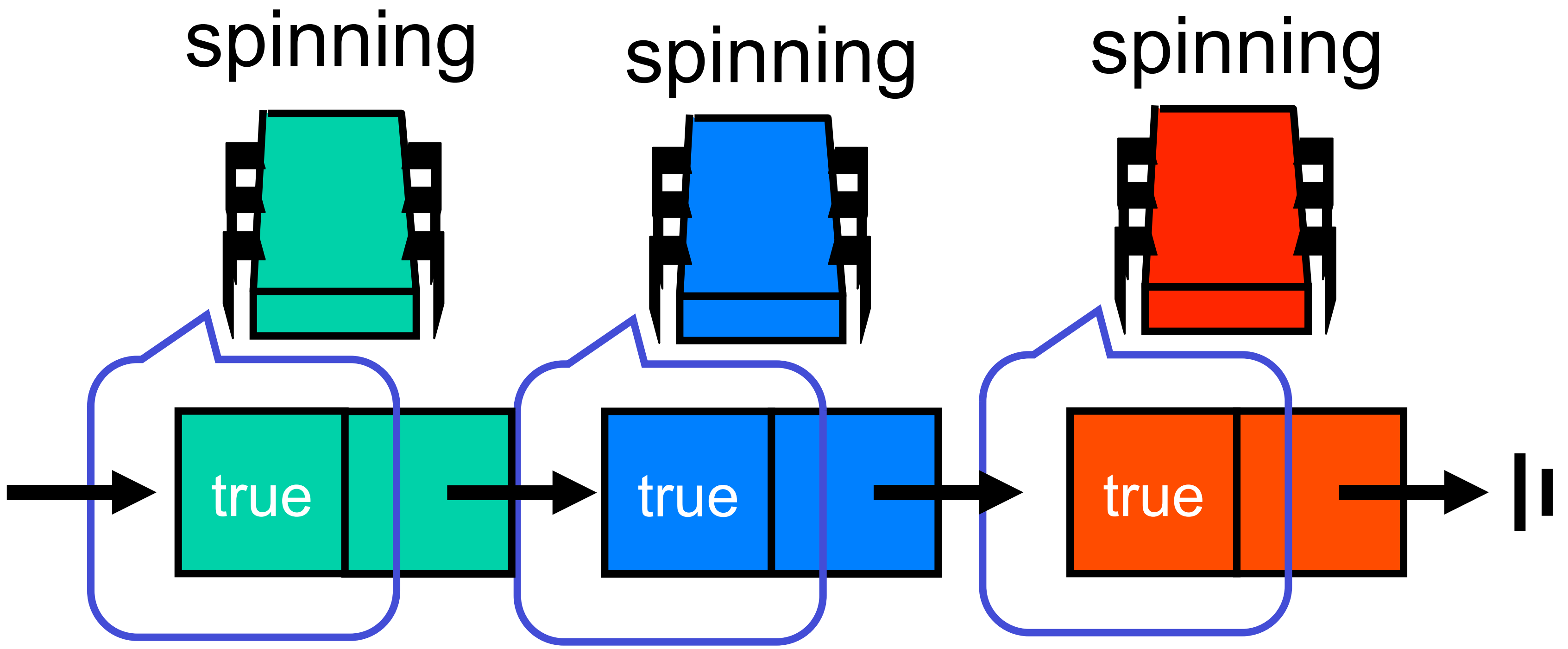
Queue Locks



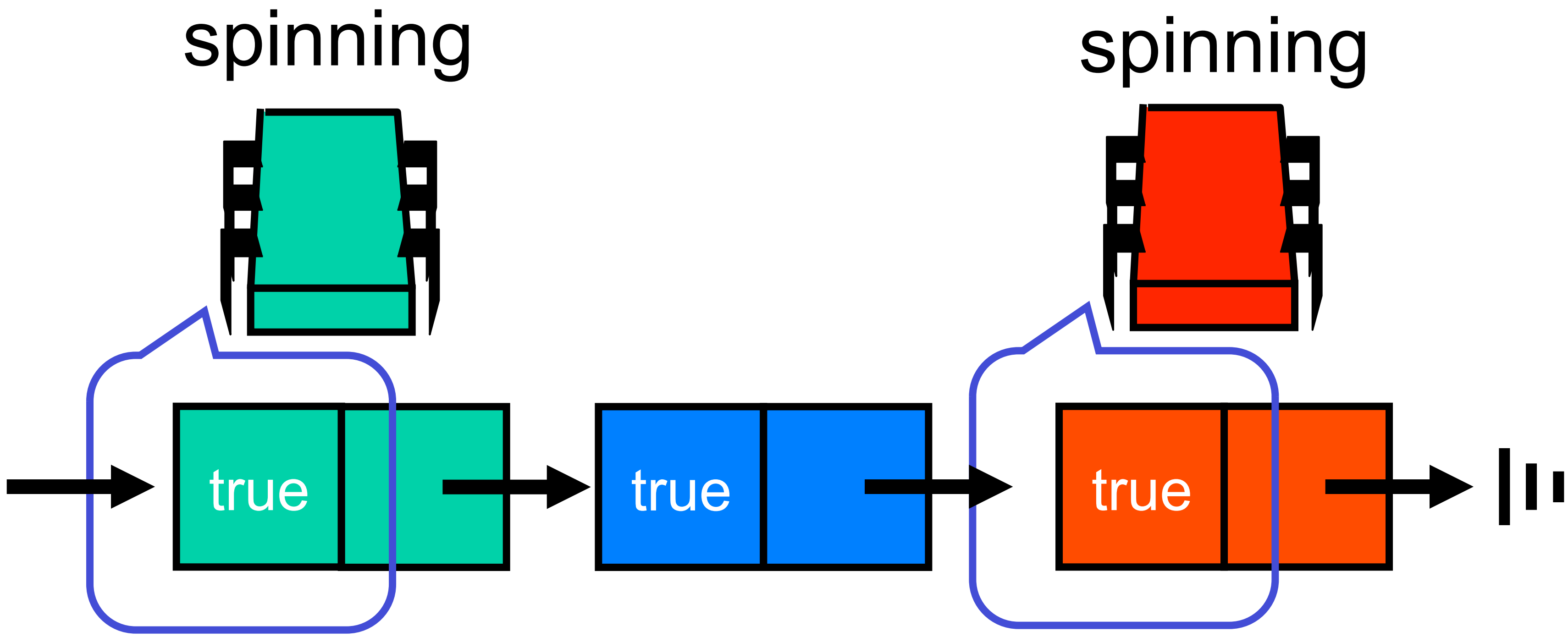
Queue Locks



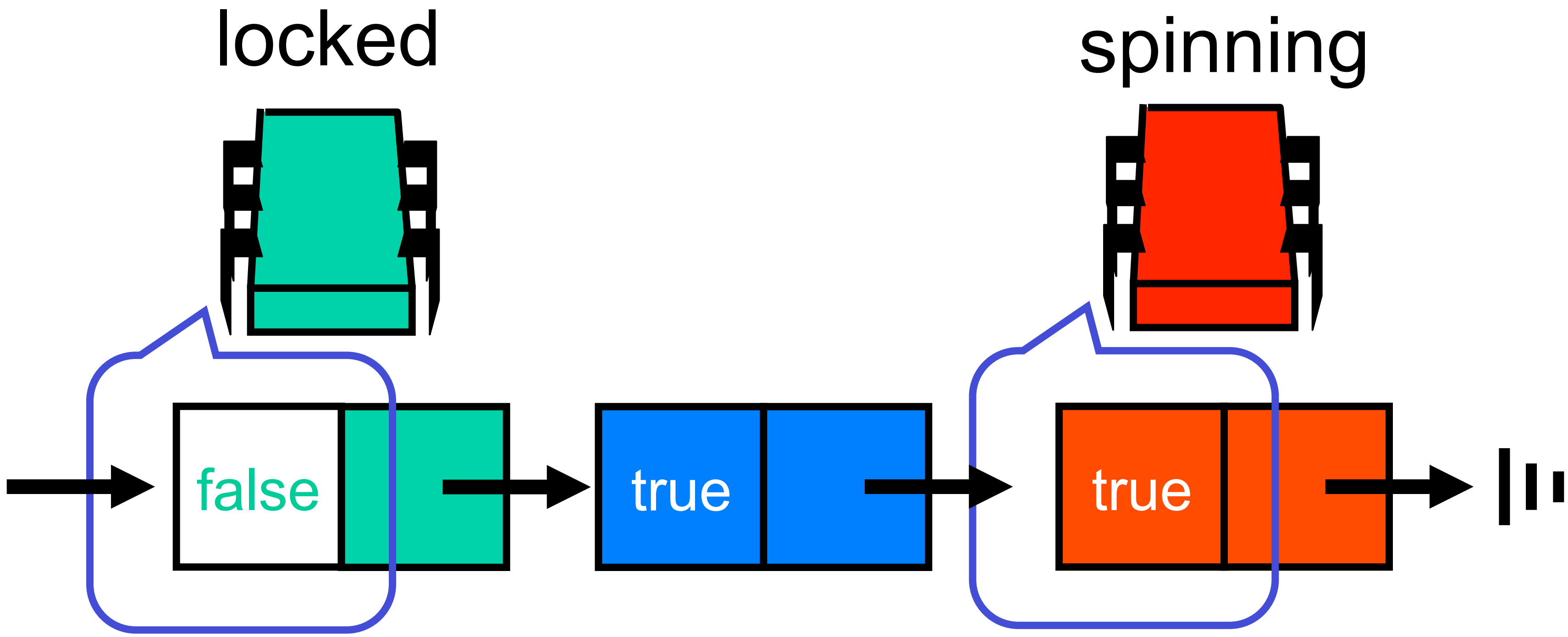
Queue Locks



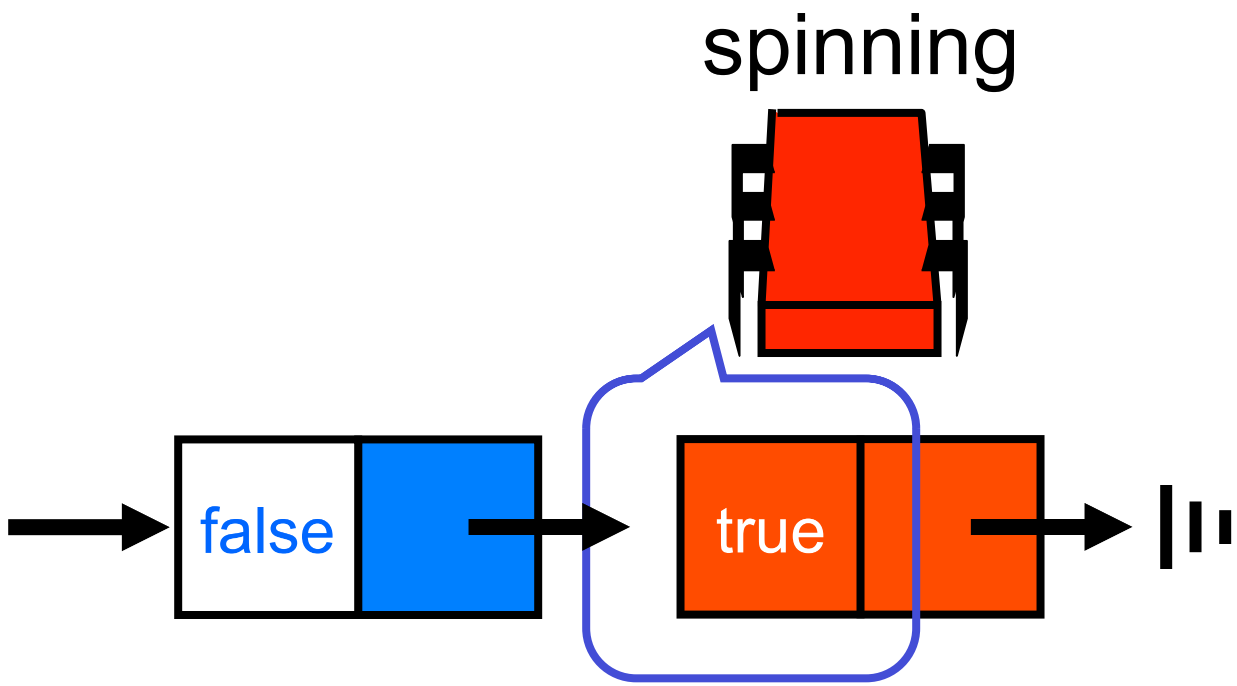
Queue Locks



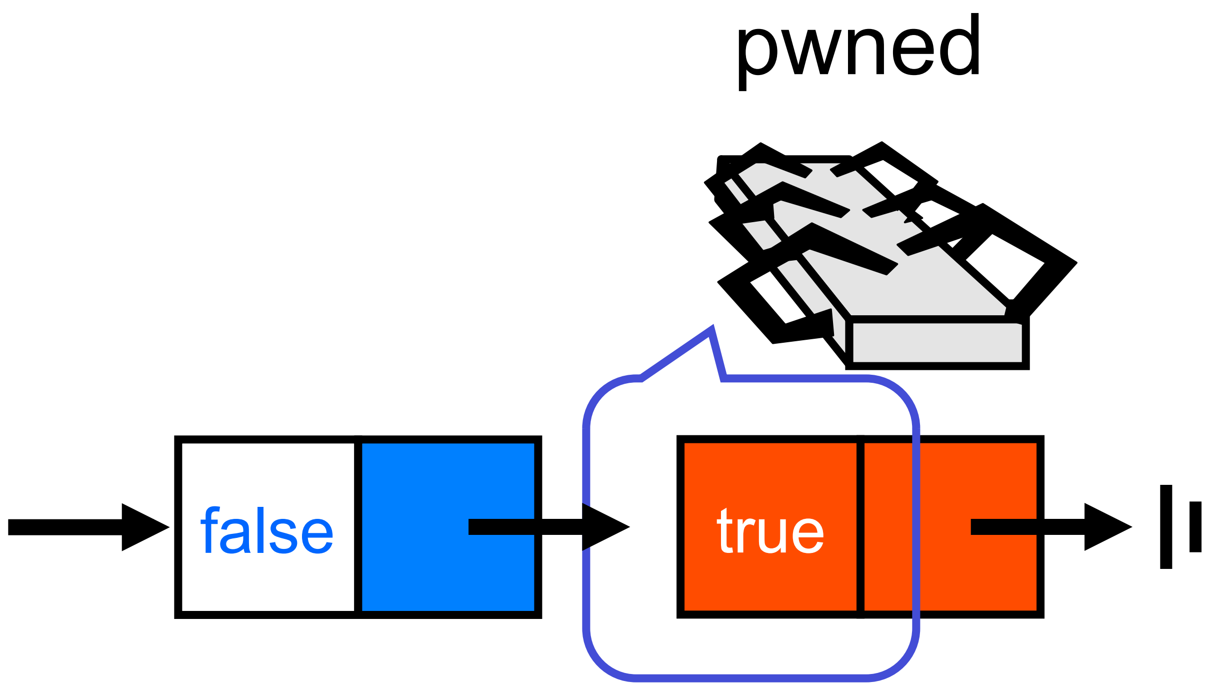
Queue Locks



Queue Locks



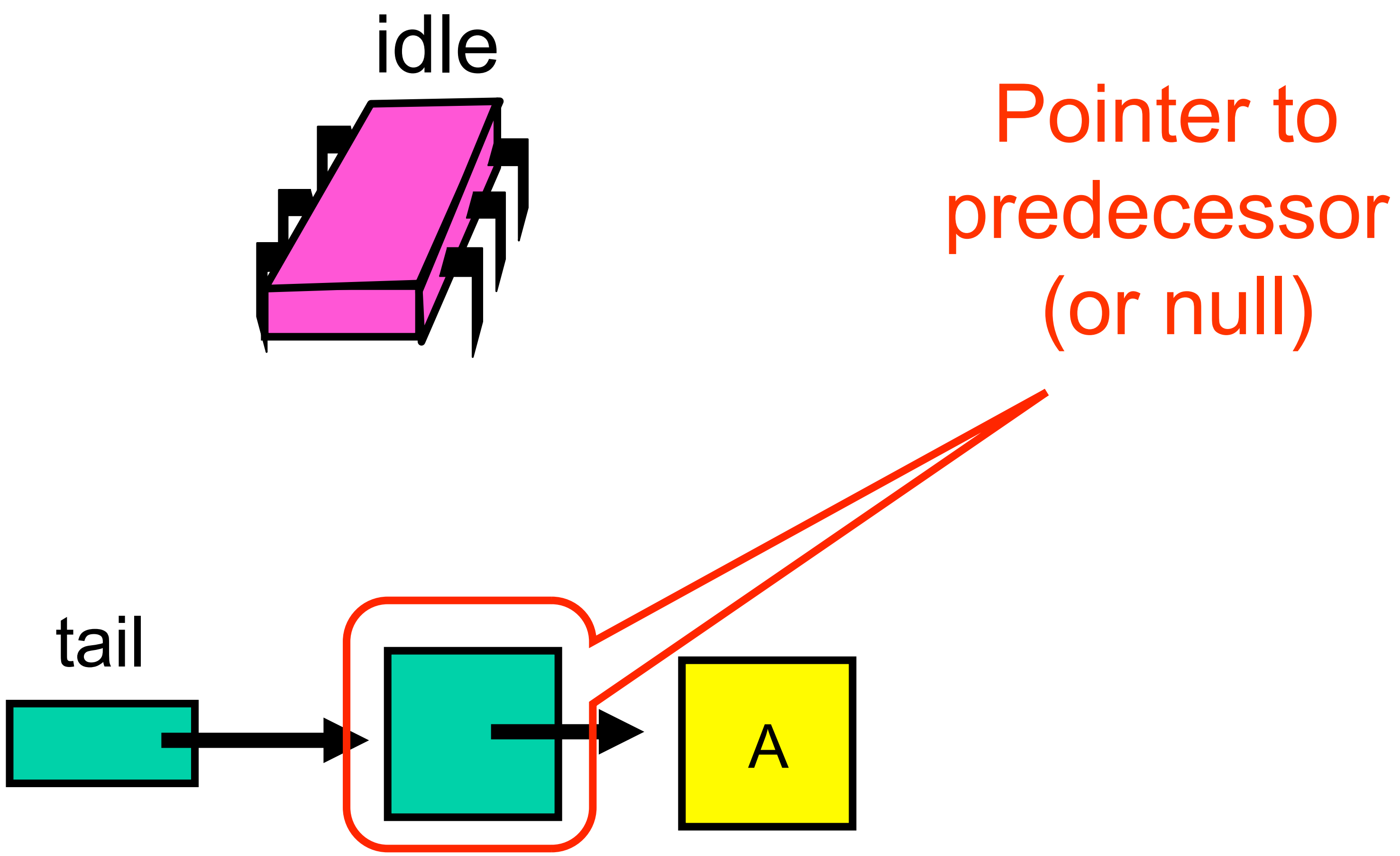
Queue Locks



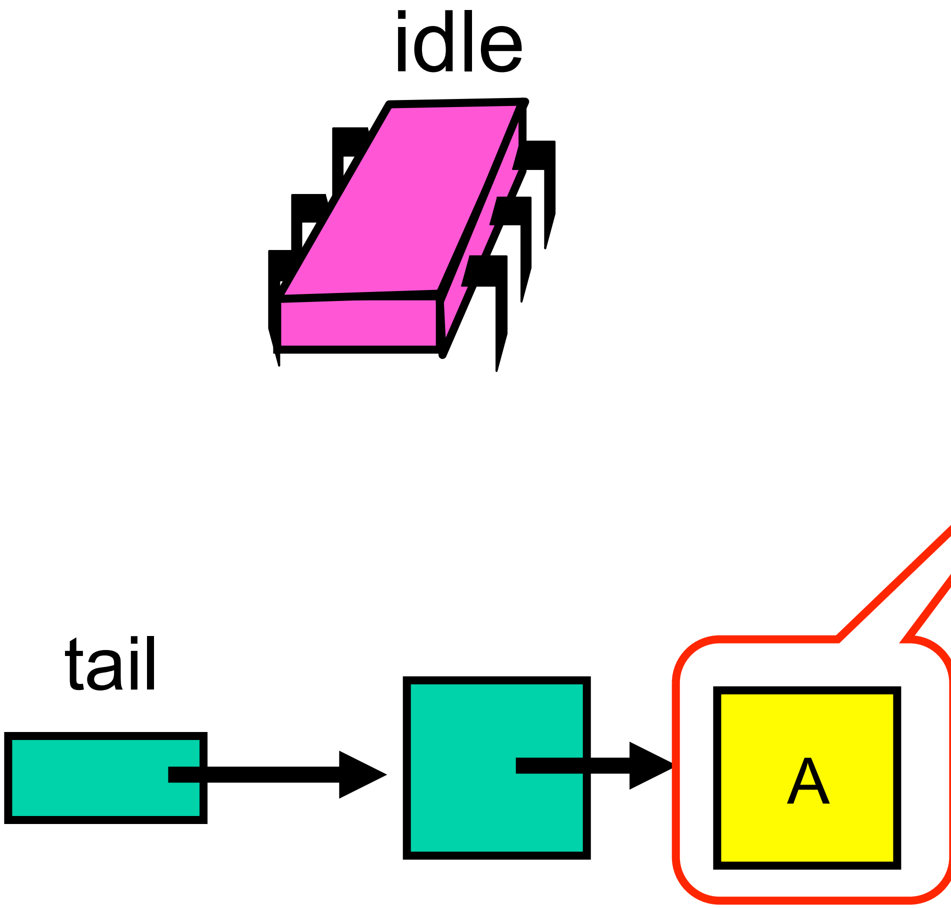
Abortable CLH Lock

- When a thread gives up
  - Removing node in a wait-free way is hard
- Idea:
  - let successor deal with it.

Initially

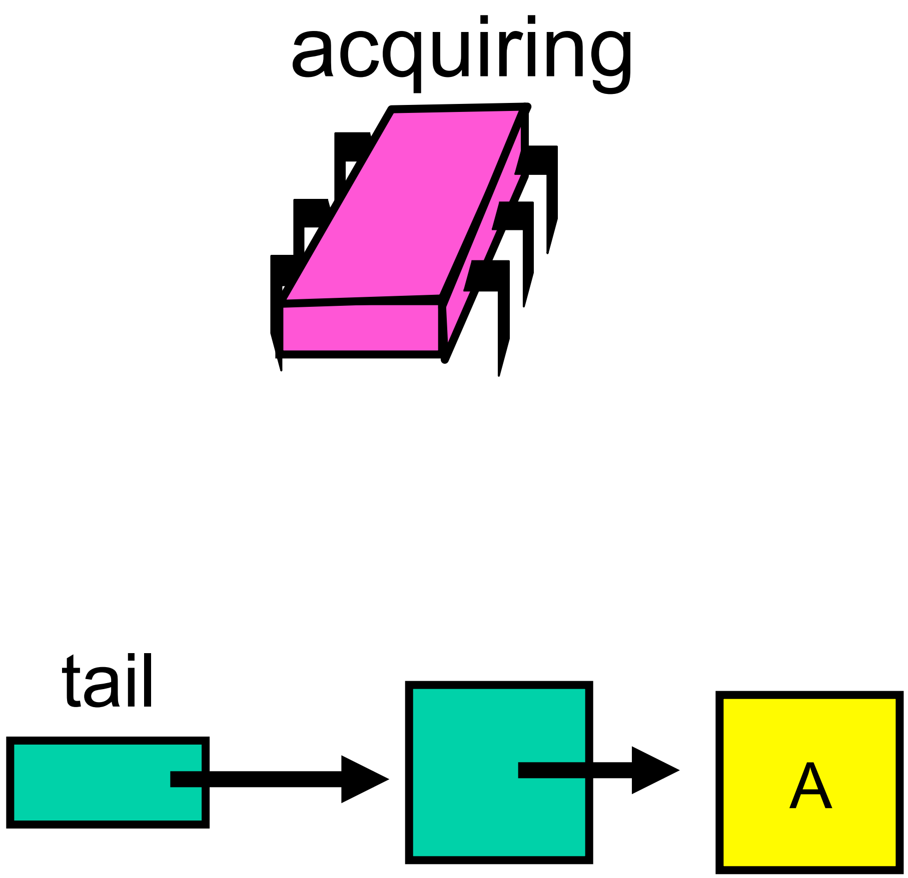


Initially

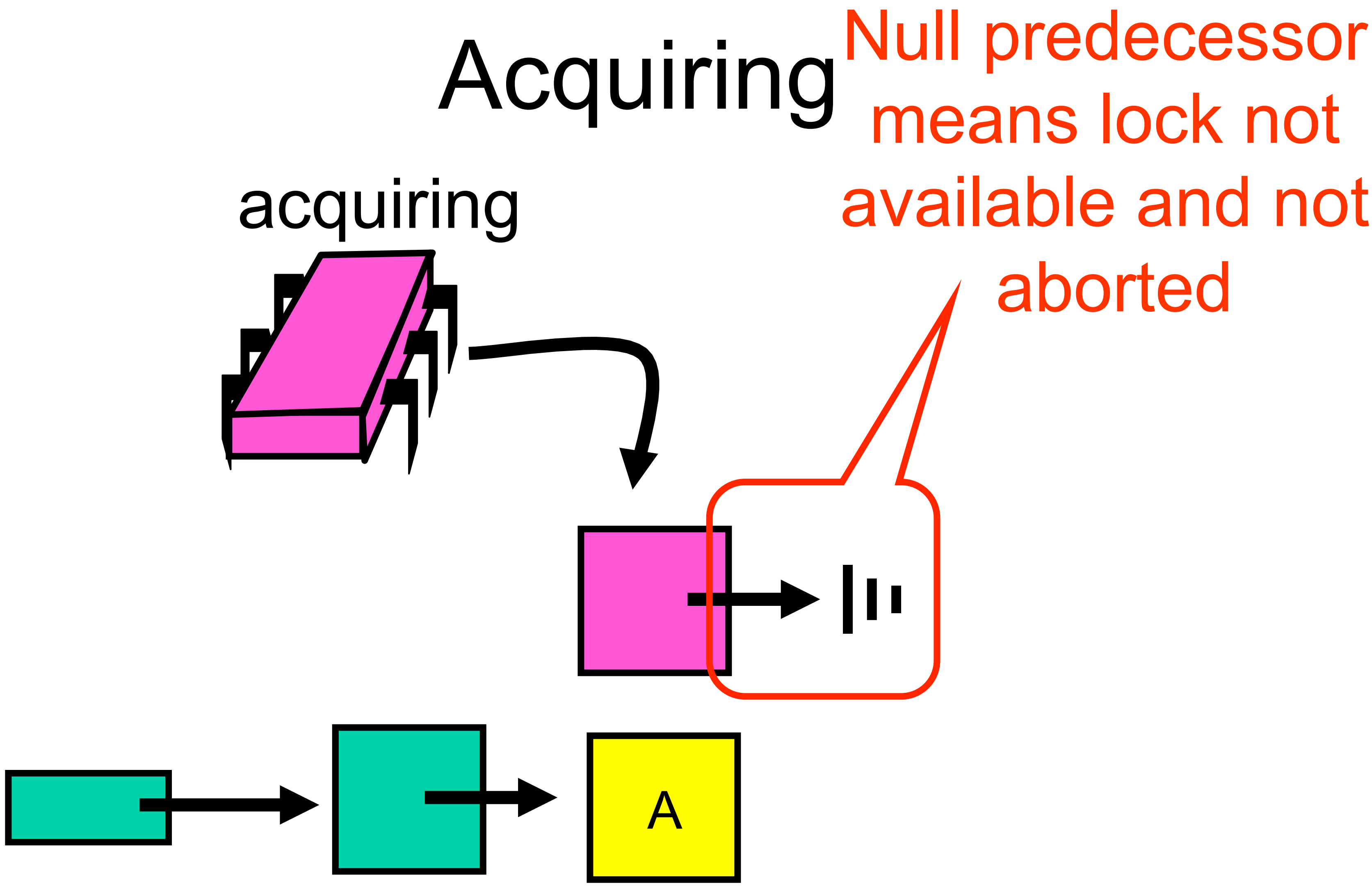


Distinguished available node means lock is free

Acquiring

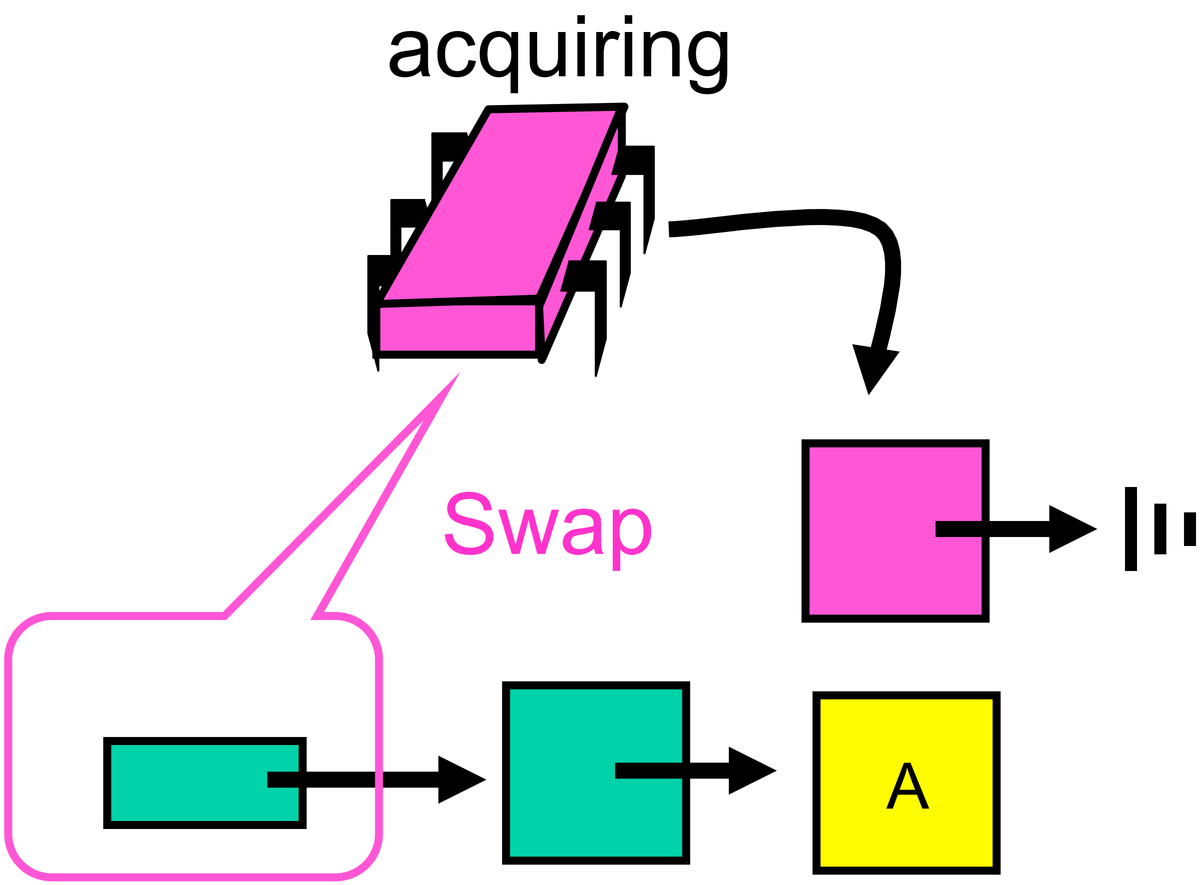


Acquiring

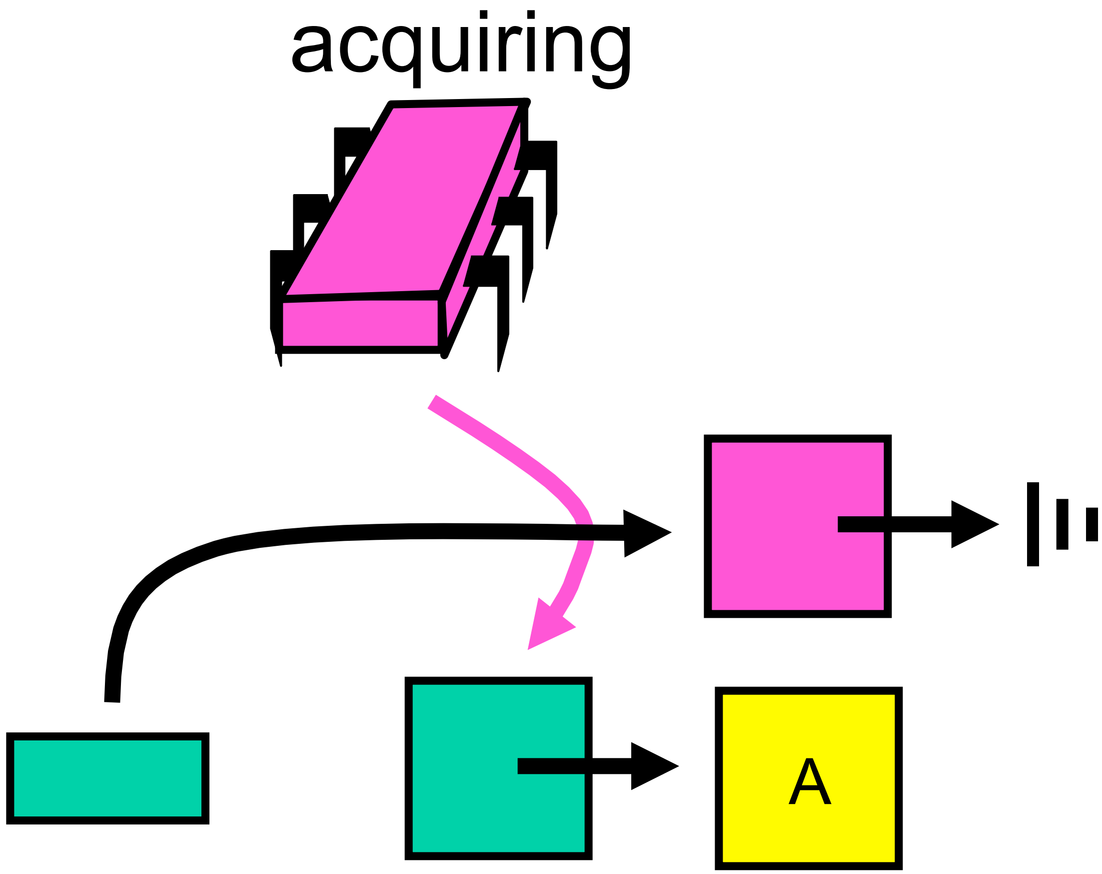


acquiring
Null predecessor means lock not available and not aborted
A

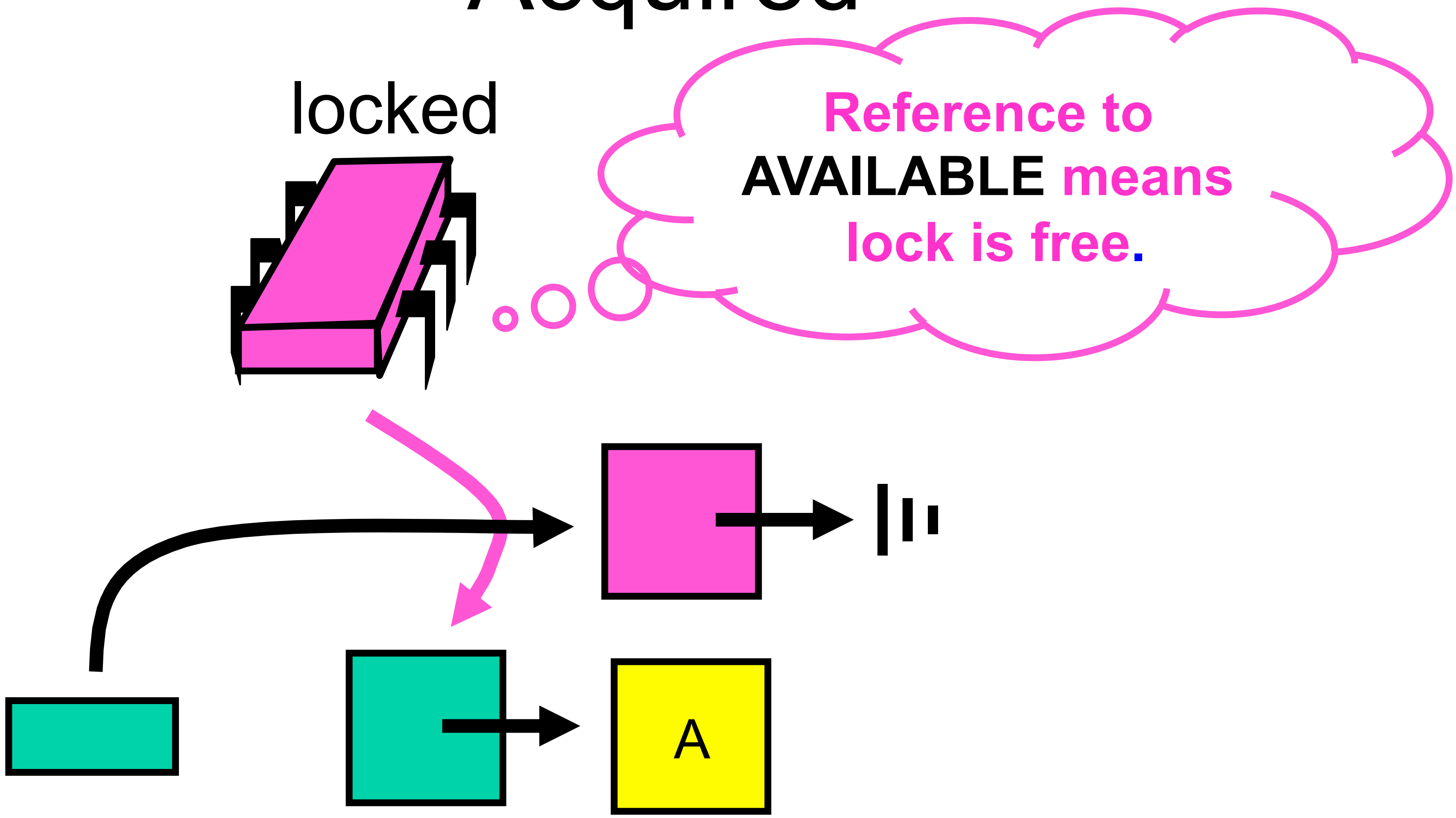
Acquiring



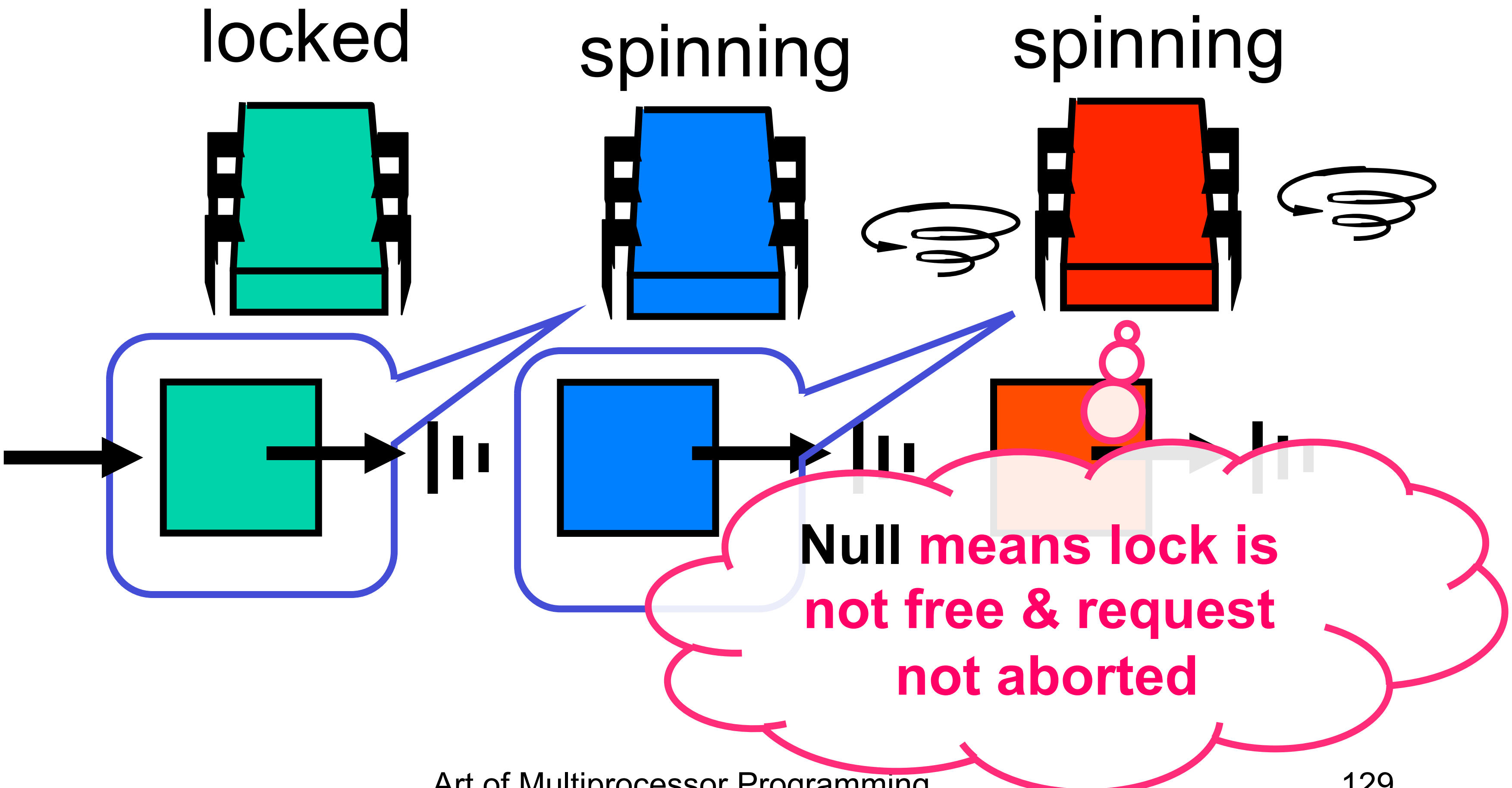
Acquiring



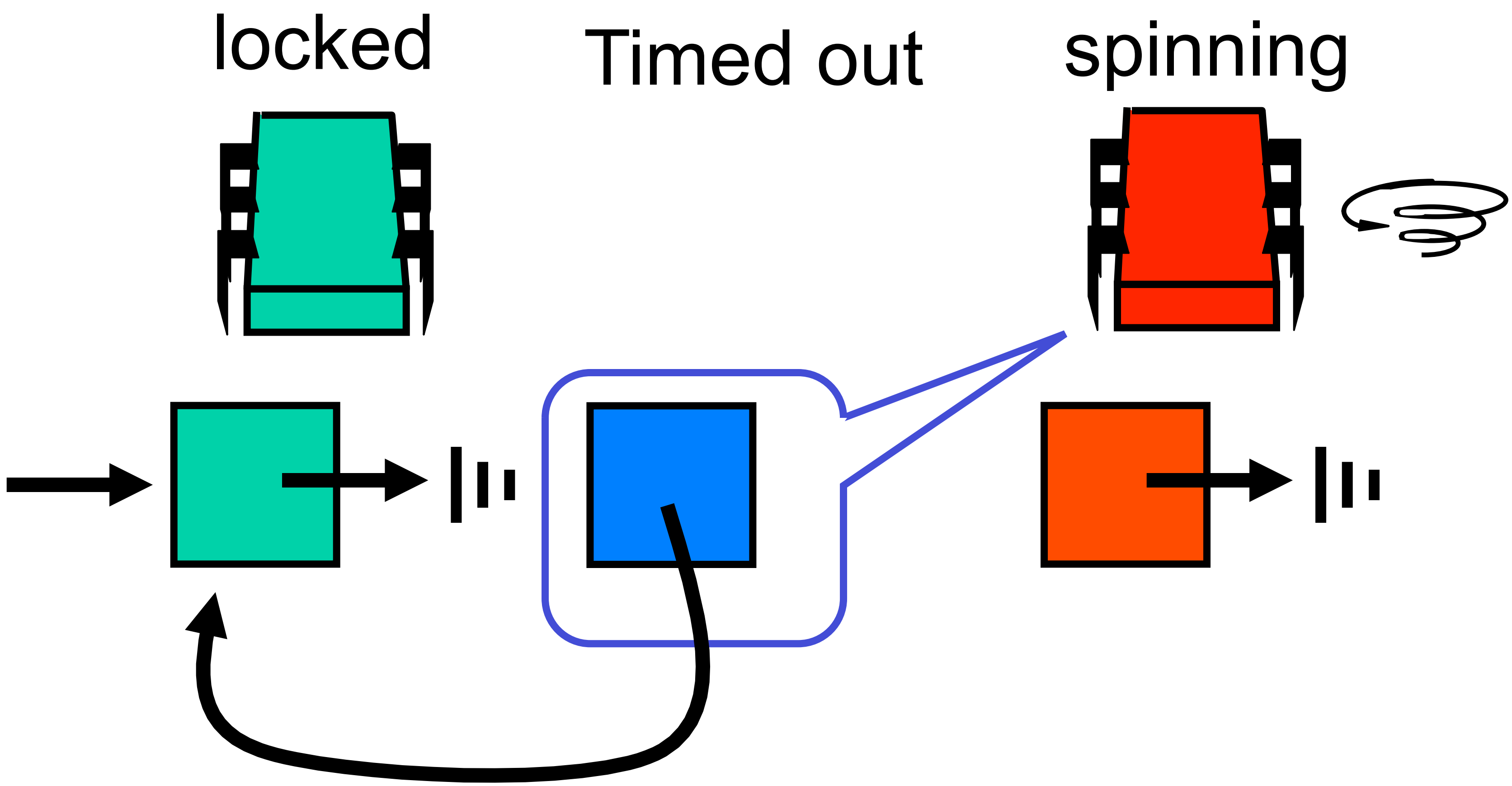
Acquired



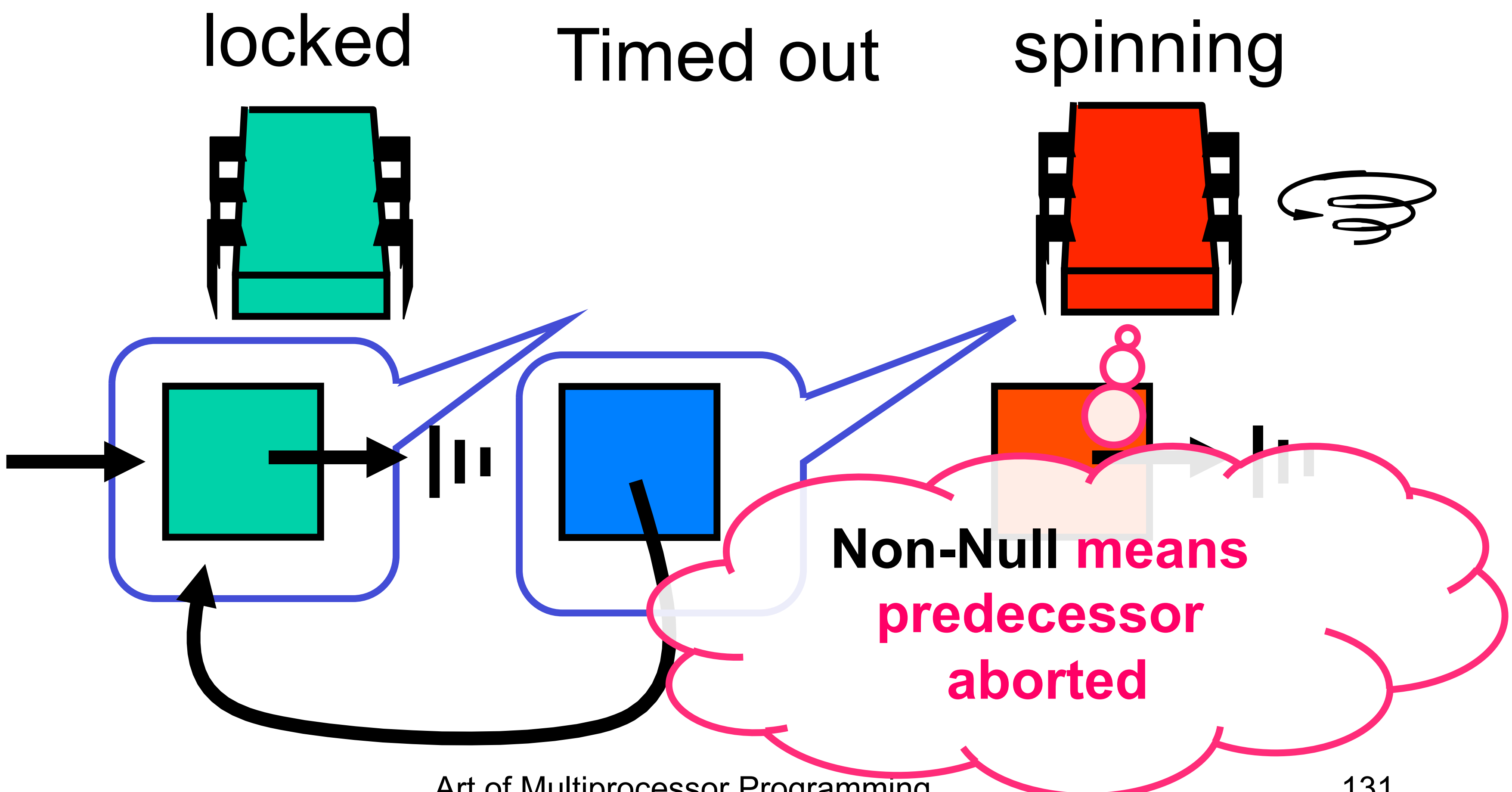
Normal Case


locked
spinning
spinning
Null means lock is not free & request not aborted

One Thread Aborts

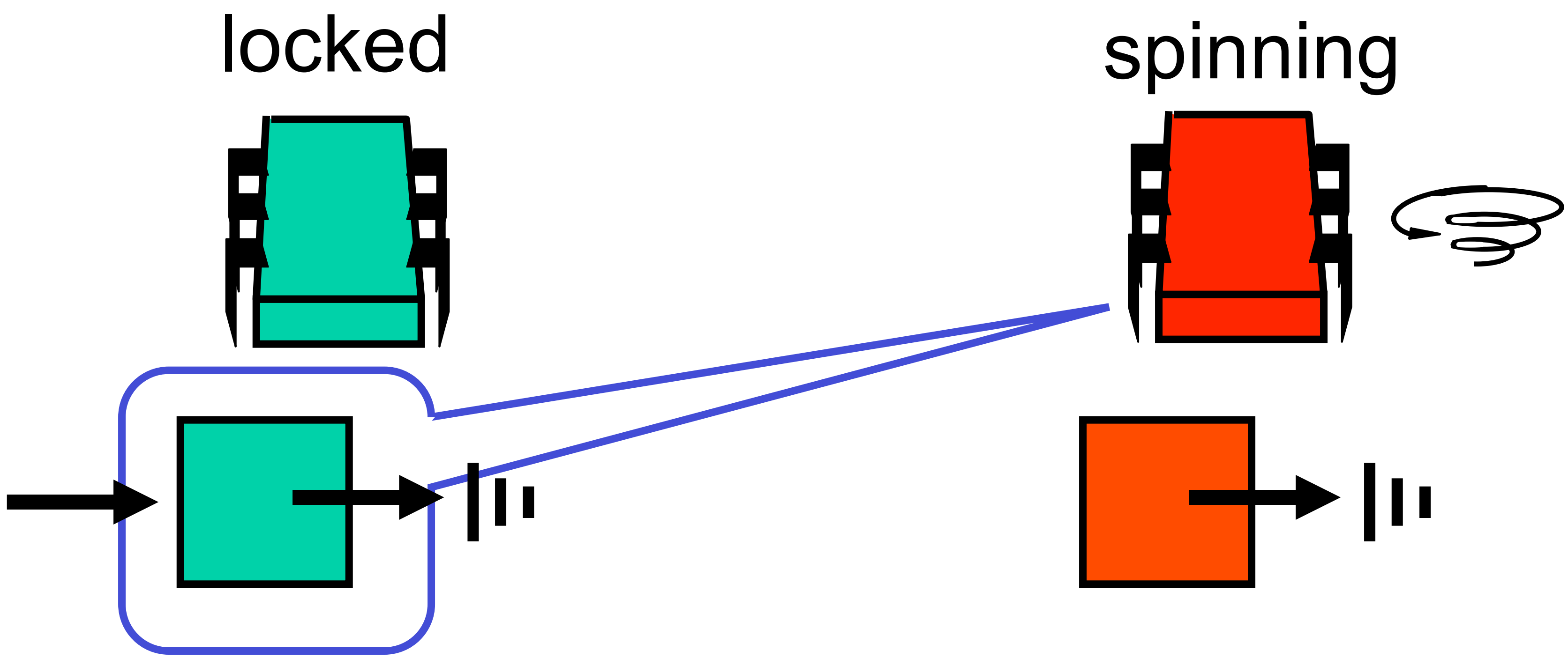


Successor Notices

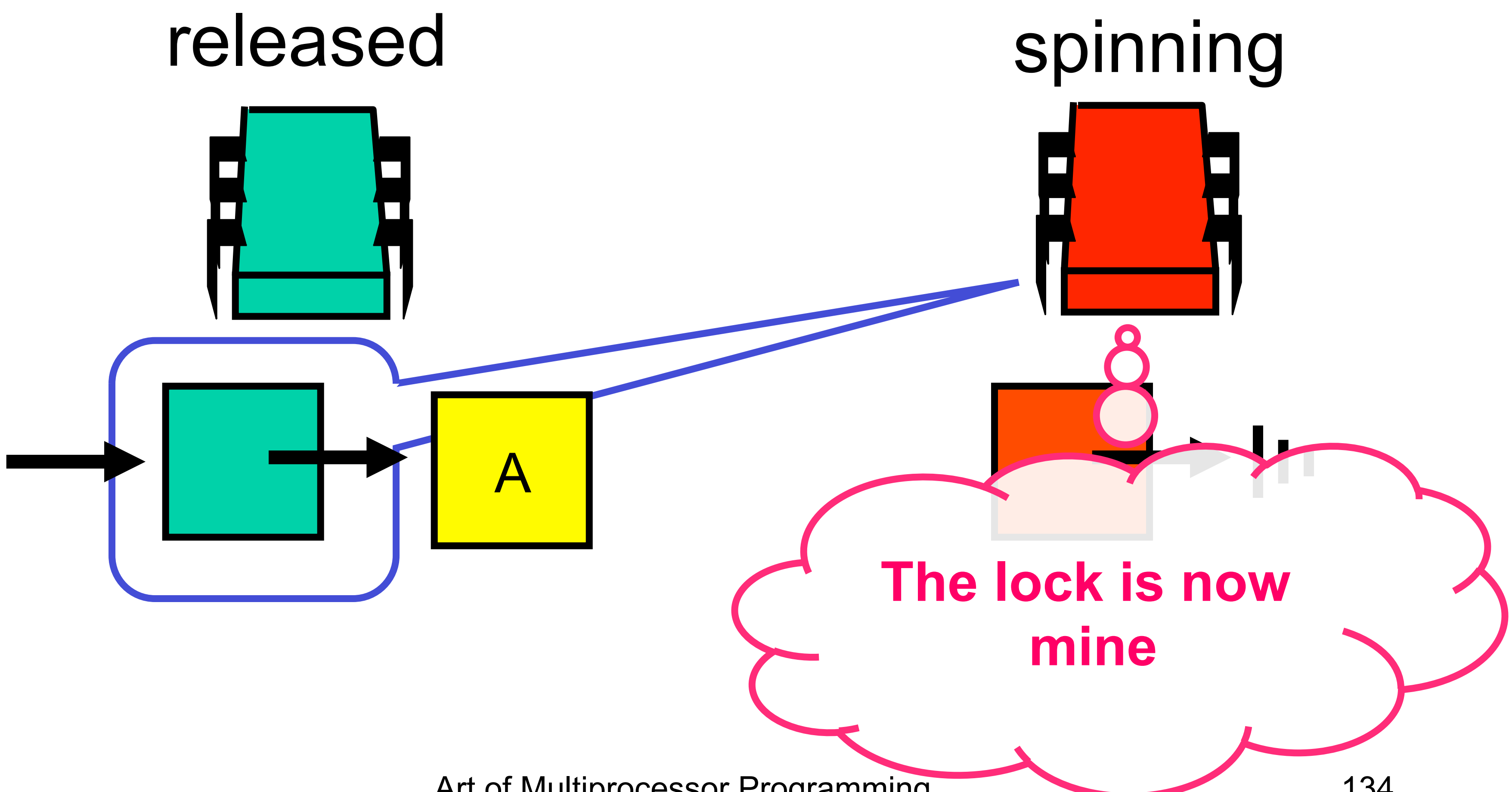


Recycle Predecessor's Node

Spin on Earlier Node



Spin on Earlier Node



Time-out Lock

```
public class TOLock implements Lock {
  static Qnode AVAILABLE
    = new Qnode();
  AtomicReference<Qnode> tail;
  ThreadLocal<Qnode> myNode;
```

Time-out Lock

```
public class TOLock implements Lock {
  static Qnode AVAILABLE
    = new Qnode();
  AtomicReference<Qnode> tail;
  ThreadLocal<Qnode> myNode;
```

AVAILABLE node signifies free lock

Time-out Lock

```
public class TOLock implements Lock {
  static Qnode AVAILABLE
    = new Qnode();
  AtomicReference<Qnode> tail;
  ThreadLocal<Qnode> myNode;
```

Tail of the queue

Time-out Lock

```
public class TOLock implements Lock {
  static Qnode AVAILABLE
    = new Qnode();
  AtomicReference<Qnode> tail;
  ThreadLocal<Qnode> myNode;
```

Remember my node …

Time-out Lock

```
public boolean lock(long timeout) {
  Qnode qnode = new Qnode();
  myNode.set(qnode);
  qnode.prev = null;
  Qnode myPred = tail.getAndSet(qnode);
  if (myPred== null
      || myPred.prev == AVAILABLE) {
      return true;
   }
…
```

Time-out Lock

```
public boolean lock(long timeout) {
  Qnode qnode = new Qnode();
  myNode.set(qnode);
  qnode.prev = null;
  Qnode myPred = tail.getAndSet(qnode);
  if (myPred == null
      || myPred.prev == AVAILABLE) {
      return true;
   }
```

Create & initialize node

Time-out Lock

```
public boolean lock(long timeout) {
  Qnode qnode = new Qnode();
  myNode.set(qnode);
  qnode.prev = null;
  Qnode myPred = tail.getAndSet(qnode);
  if (myPred == null
      || myPred.prev == AVAILABLE) {
      return true;
   }
```

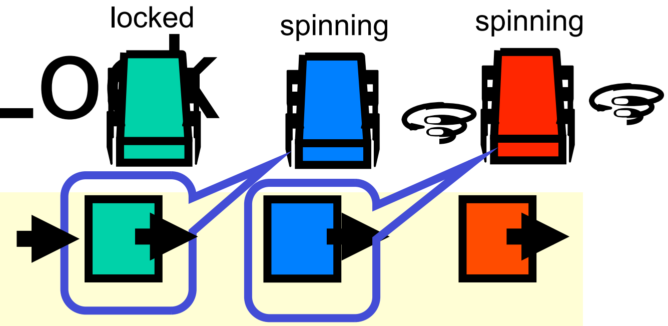
Swap with tail

Time-out Lock

```
public boolean lock(long timeout) {
  Qnode qnode = new Qnode();
  myNode.set(qnode);
  qnode.prev = null;
  Qnode myPred = tail.getAndSet(qnode);
  if (myPred == null
      || myPred.prev == AVAILABLE) {
      return true;
  }
  ...
```

If predecessor absent or released, we are done

Time-out Lock




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

Time-out Lock

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

Keep trying for a while

...

Time-out Lock

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

Spin on predecessor's prev field

Time-out Lock

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

Predecessor released lock

Time-out Lock

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

Predecessor aborted, advance one

Time-out Lock

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

What do I do when I time out?

Time-out Lock

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

Do I have a successor? If CAS succeeds: no successor, tail just set to my pred, simply return false

Time-out Lock

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

If CAS fails, I do have a successor.
Tell it about myPred

Time-Out Unlock

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

Time-out Unlock

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

If CAS failed: successor exists, notify it can enter

Timing-out Lock

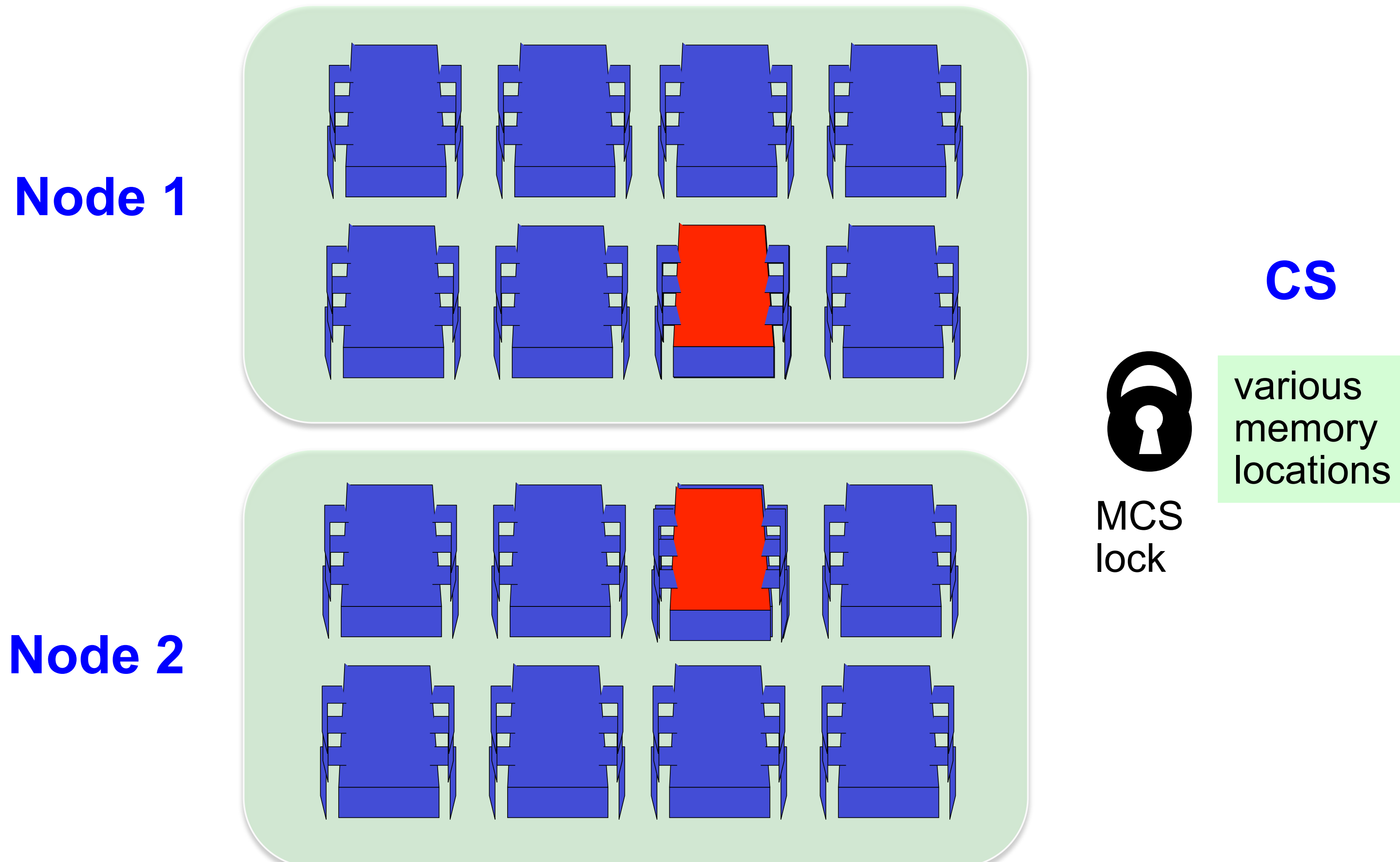
```
public void unlock() {
  Qnode qnode = myNode.get();
  if (!tail.compareAndSet(qnode, null))
    qnode.prev = AVAILABLE;
}
```

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

Fairness and NUMA Locks

- MCS lock mechanics are aware of NUMA
- Lock Fairness is FCFS
- Is this a good fit with NUMA and Cache-Coherent NUMA machines?

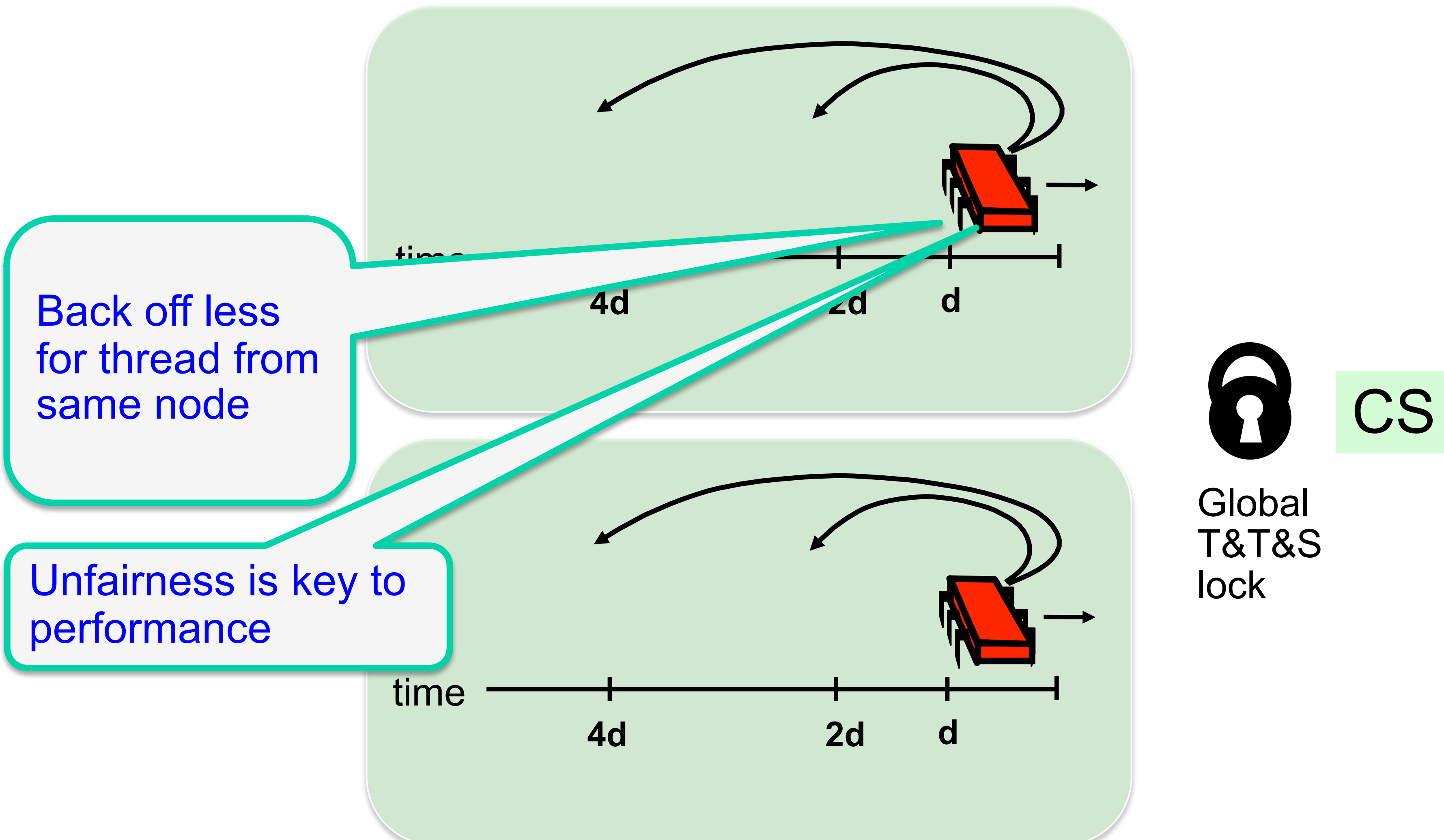
Lock Data Access in NUMA Machine



“Who’s the Unfairest of Them All?”

- locality crucial to NUMA performance
- Big gains if threads from same node/ cluster obtain lock consecutively
- Unfairness pays

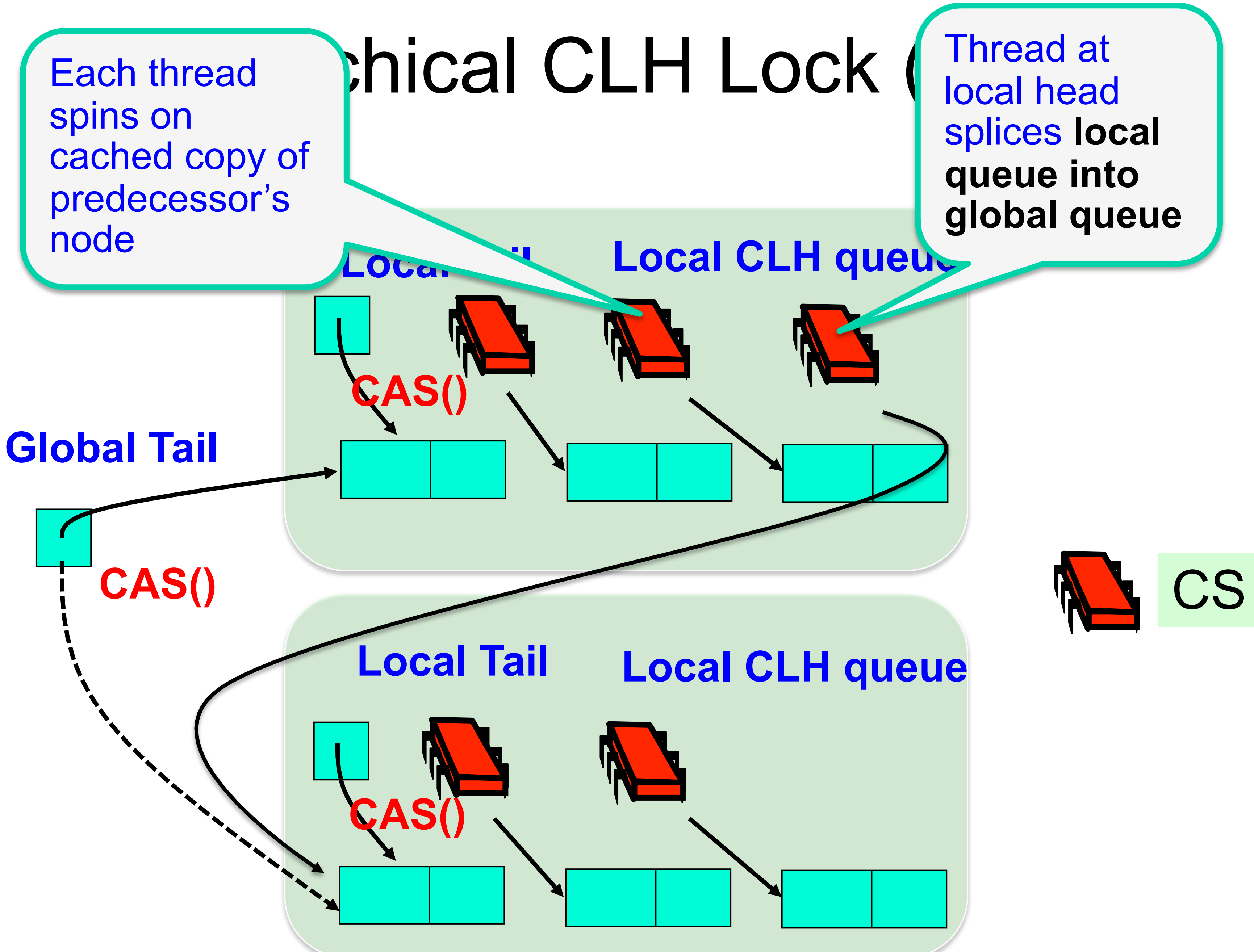
Hierarchical Backoff Lock (HBO)



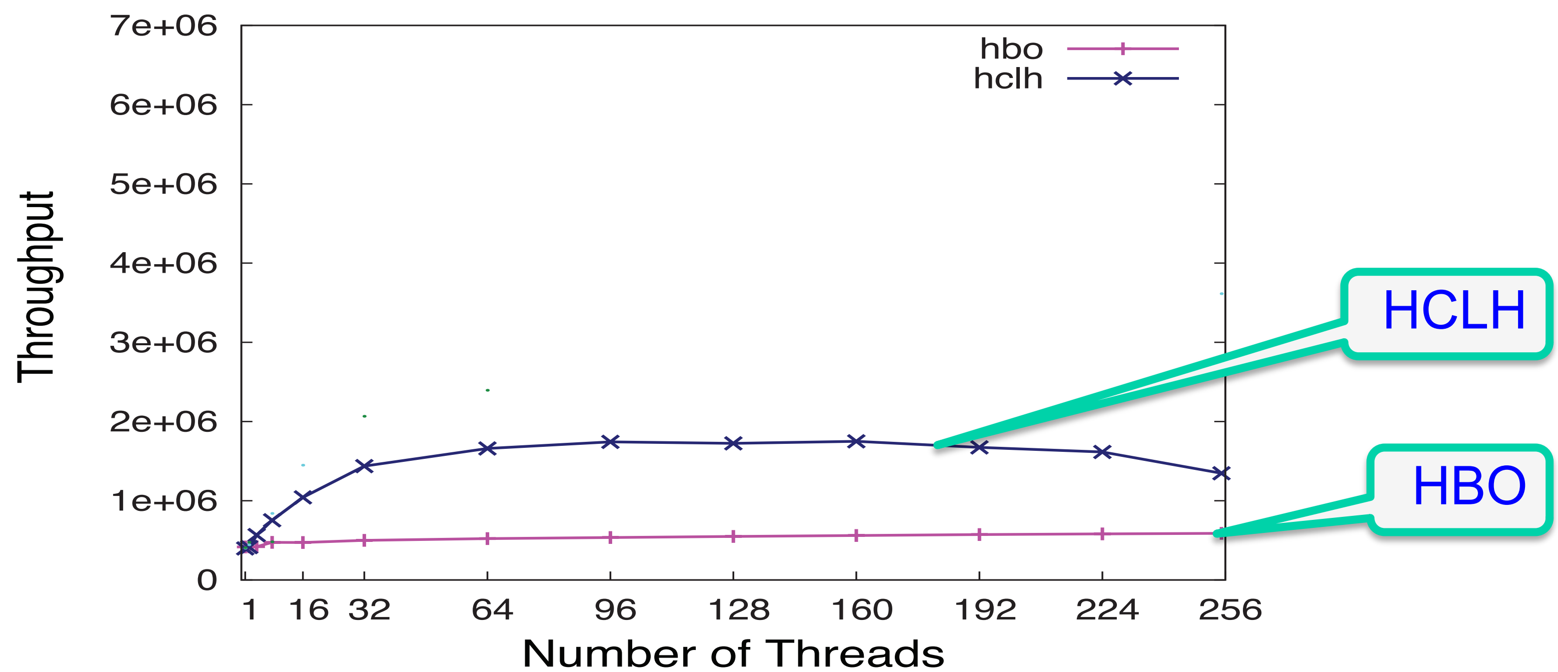
Hierarchical Backoff Lock (HBO)

- Advantages:
  - Simple, improves locality
- Disadvantages:
  - Requires platform specific tuning
  - Unstable
  - Unfair
  - Continuous invalidations on shared global lock word

chical CLH Lock (



Hierarchical CLH Lock (HCLH)



Threads access 4 cache lines in CS

Hierarchical CLH Lock (HCLH)

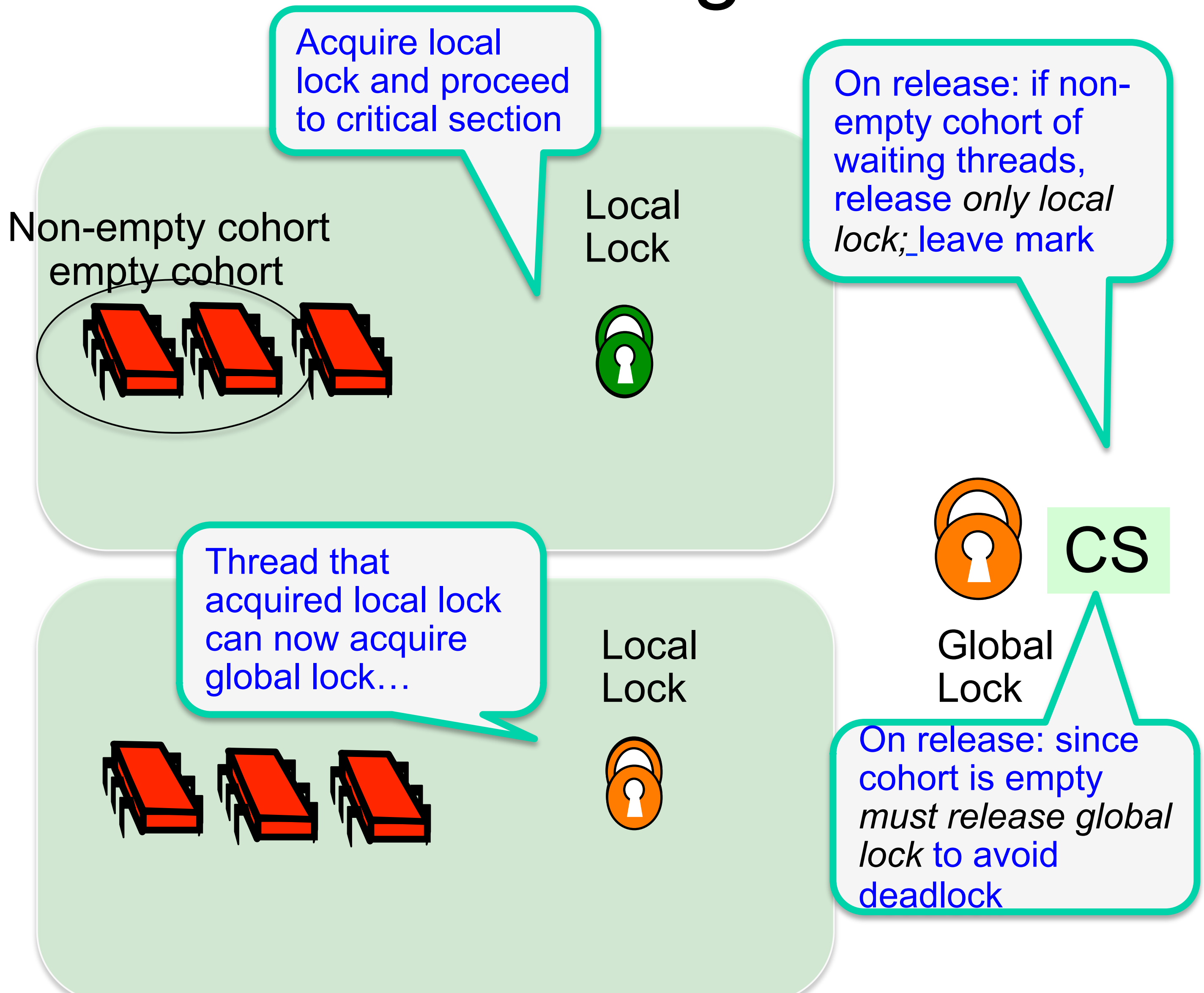
- Advantages:
  - Improved locality
  - Local spinning
  - Fair
- Disadvantages:
  - Complex code implies long common path
  - Splicing into both local and global requires CAS
  - Hard to get long local sequences

"Nothing yet. ... How about you, Newton?"

Lock Cohorting

- General technique for converting almost any lock into a NUMA lock
- Allows combining different lock types
- But need these locks to have certain properties (will discuss shortly)

Lock Cohorting



Thread Obliviousness

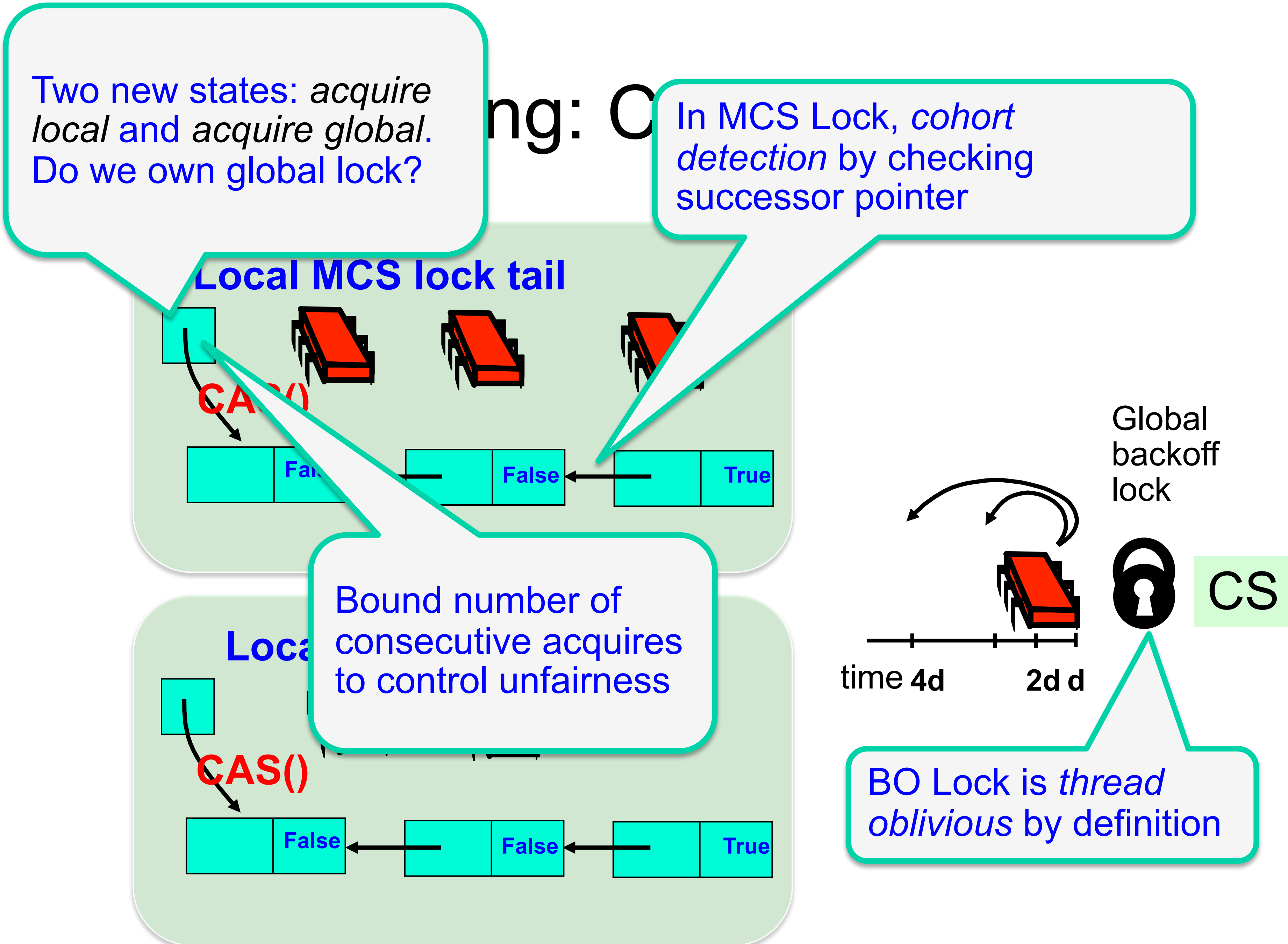
- A lock is ***thread-oblivious*** if
  - After being acquired by one thread,
  - Can be released by another

Cohort Detection

- A lock provides ***cohort detection*** if
  - It can tell whether any thread is trying to acquire it

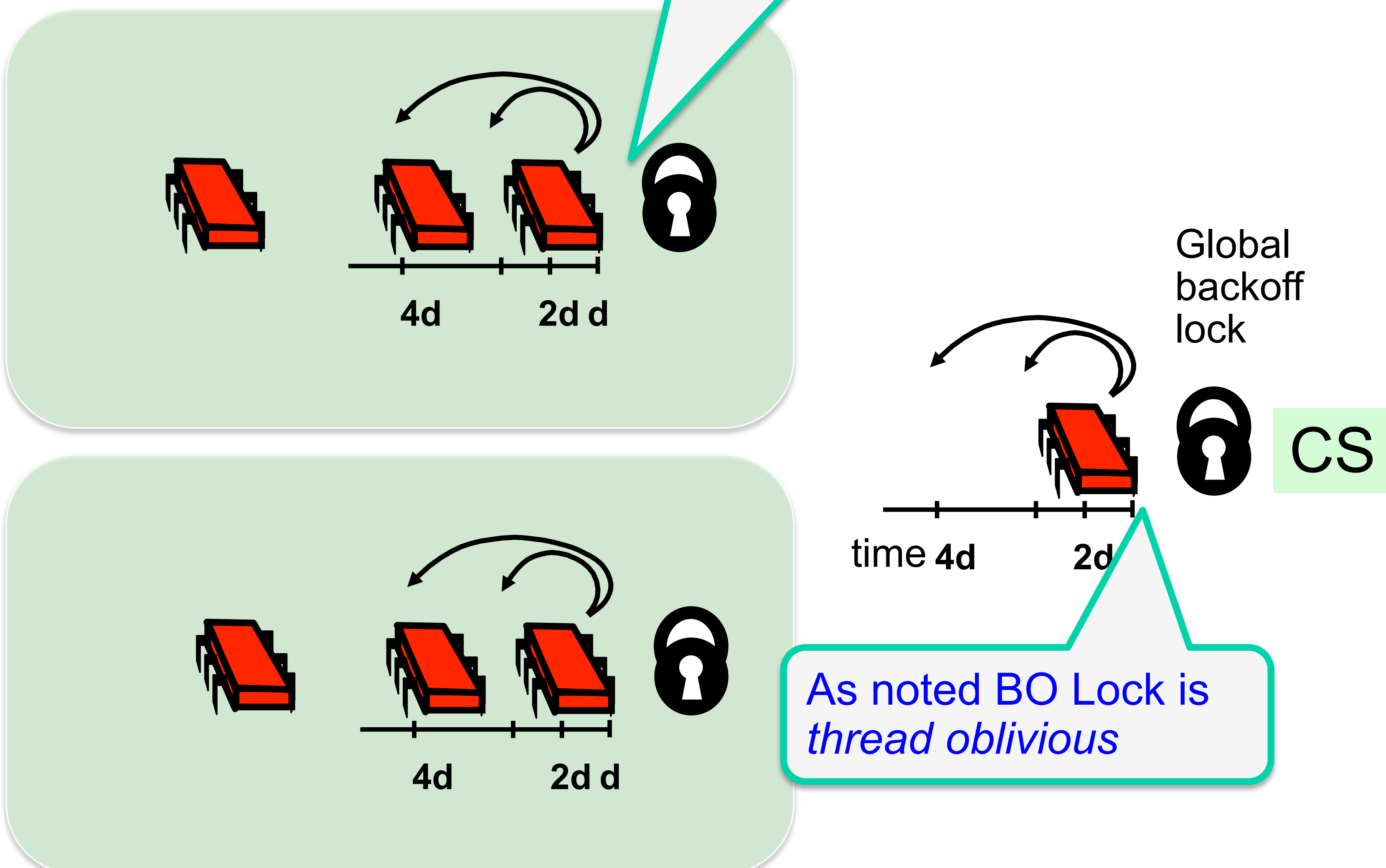
Lock Cohorting

- Two levels of locking
- **Global lock**: thread oblivious
  - Thread acquiring the lock can be different than one releasing it
- **Local lock**: cohort detection
  - Thread releasing can detect if some thread is waiting to acquire it


ng: C
Two new states: *acquire local* and *acquire global*. Do we own global lock?
In MCS Lock, *cohort detection* by checking successor pointer
Local MCS lock tail
CAS()
False
True
Bound number of consecutive acquires to control unfairness
False
False
True
Global backoff lock
CS
time 4d
2d d
BO Lock is *thread oblivious* by definition

Lock Co O Lock

How to add *cohort detection property* to BO lock?



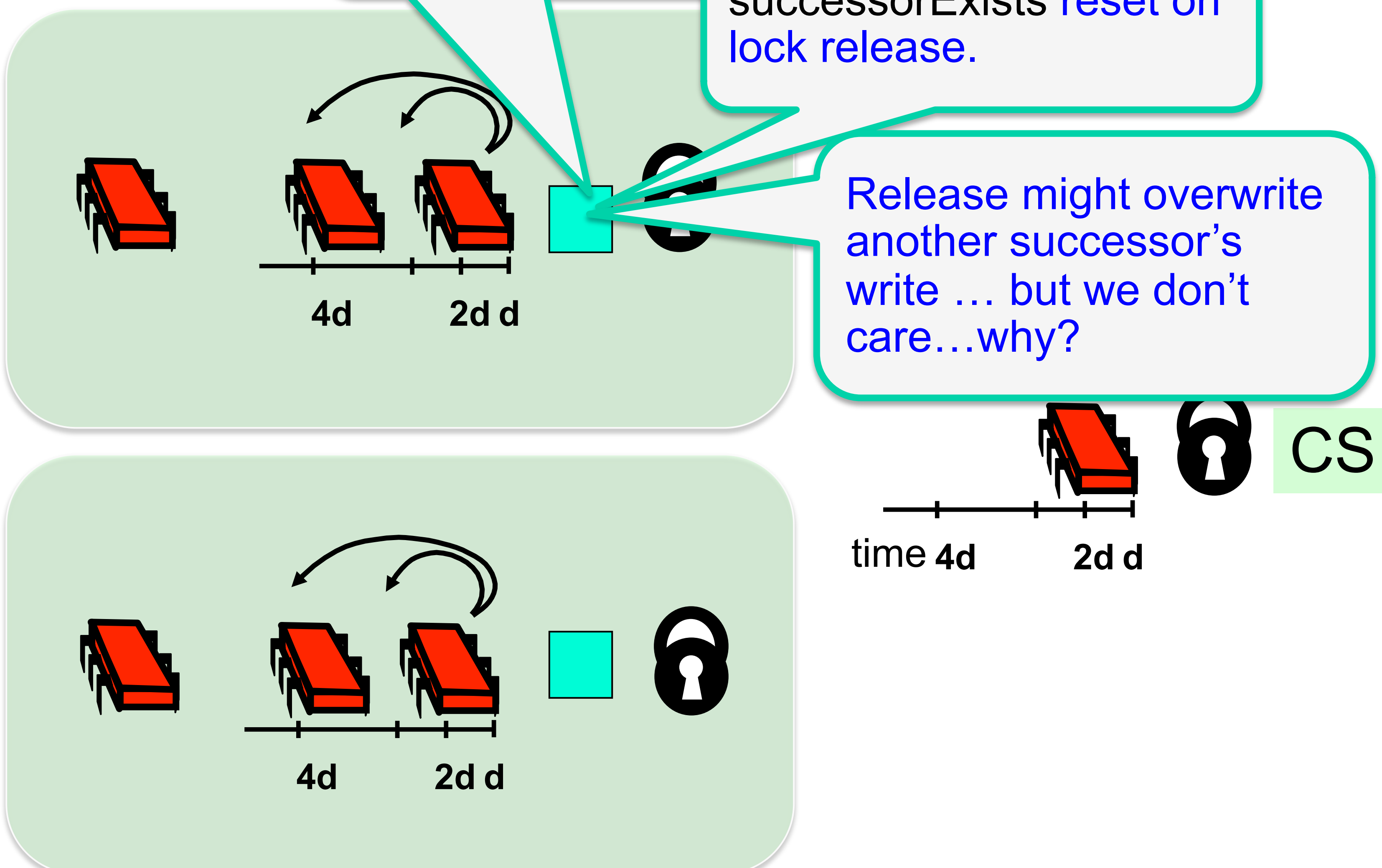
As noted BO Lock is *thread oblivious*

Lock Co -BO Lock

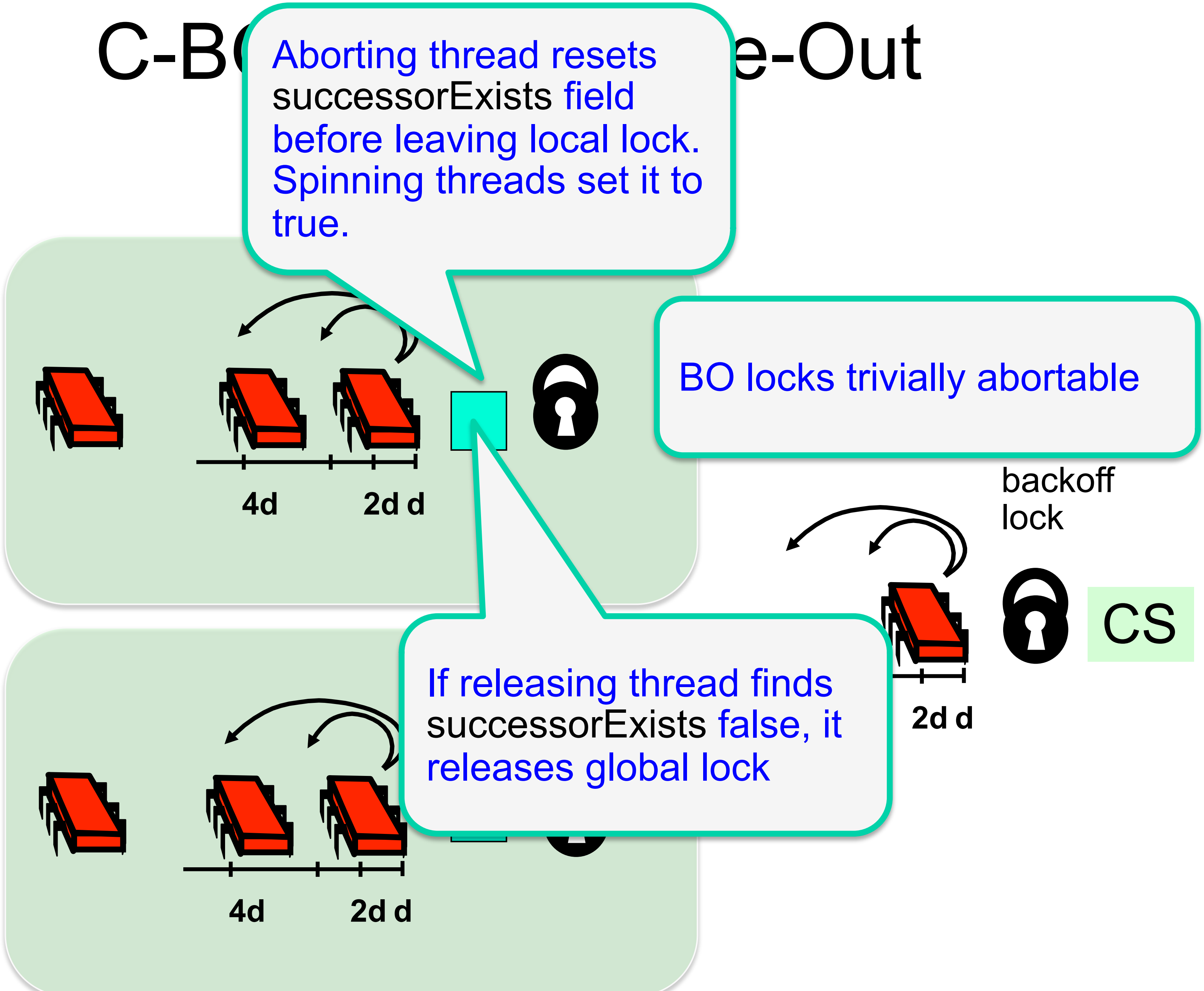
Write successorExists field before attempting to acquire local lock.

successorExists reset on lock release.

Release might overwrite another successor's write … but we don't care…why?



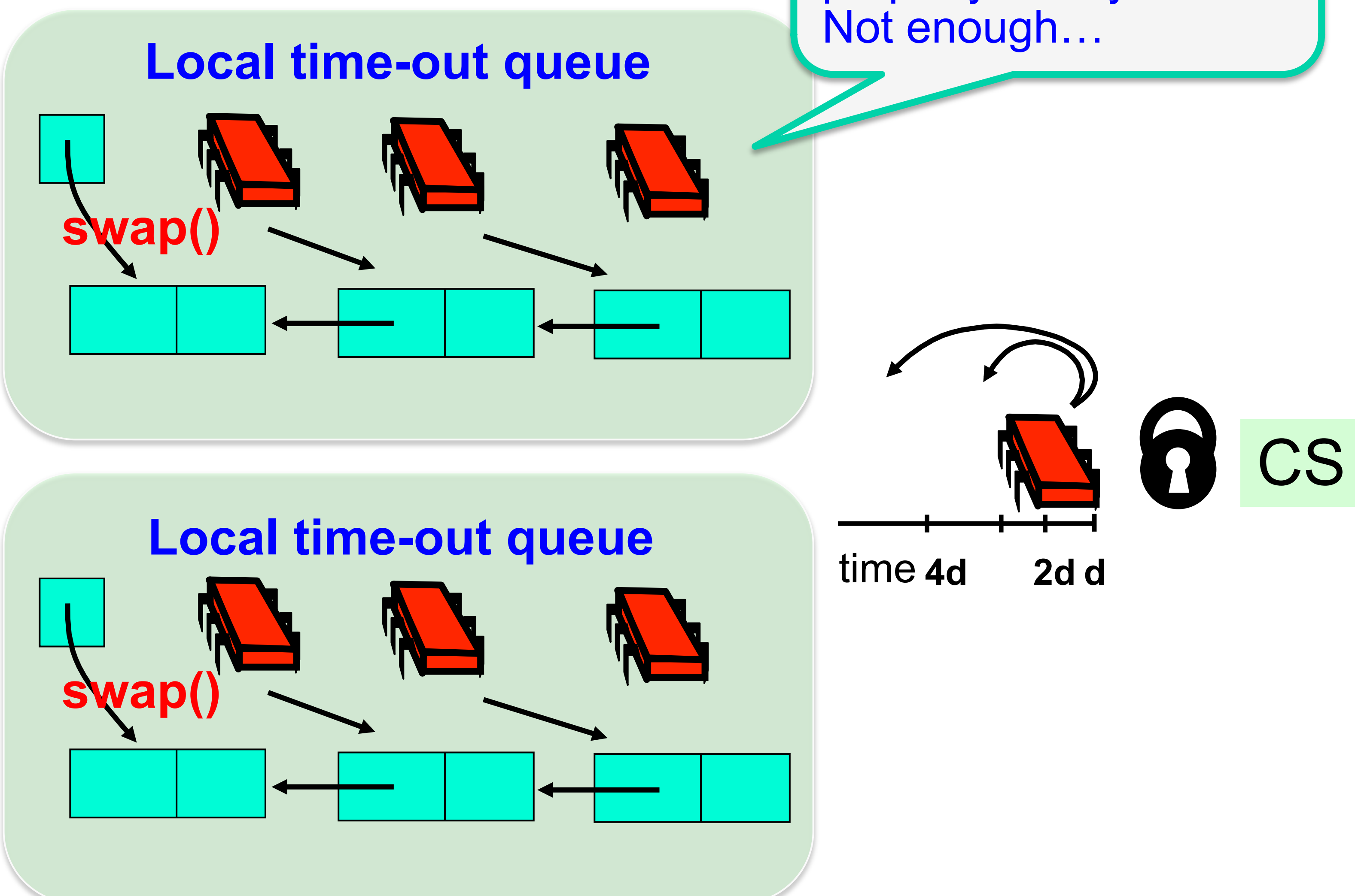
C-B e-Out



Aborting thread resets successorExists field before leaving local lock. Spinning threads set it to true.
BO locks trivially abortable
4d
2d d
backoff lock
CS
2d d
If releasing thread finds successorExists false, it releases global lock
4d
2d d

Time-Out NUMA Lock

Local Time-Out locks
have cohort detection
property …why?
Not enough…

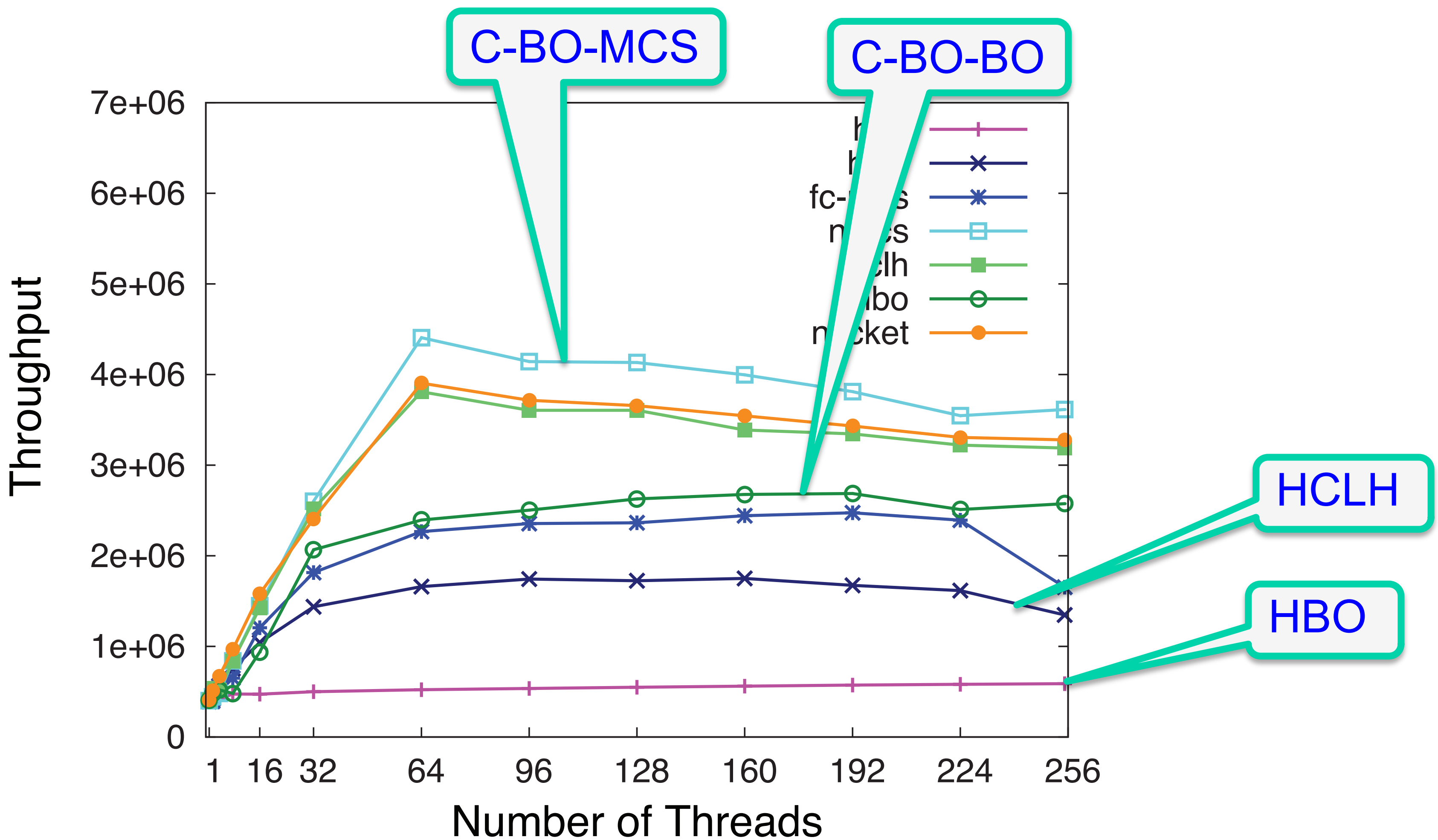


Lock Cohorting

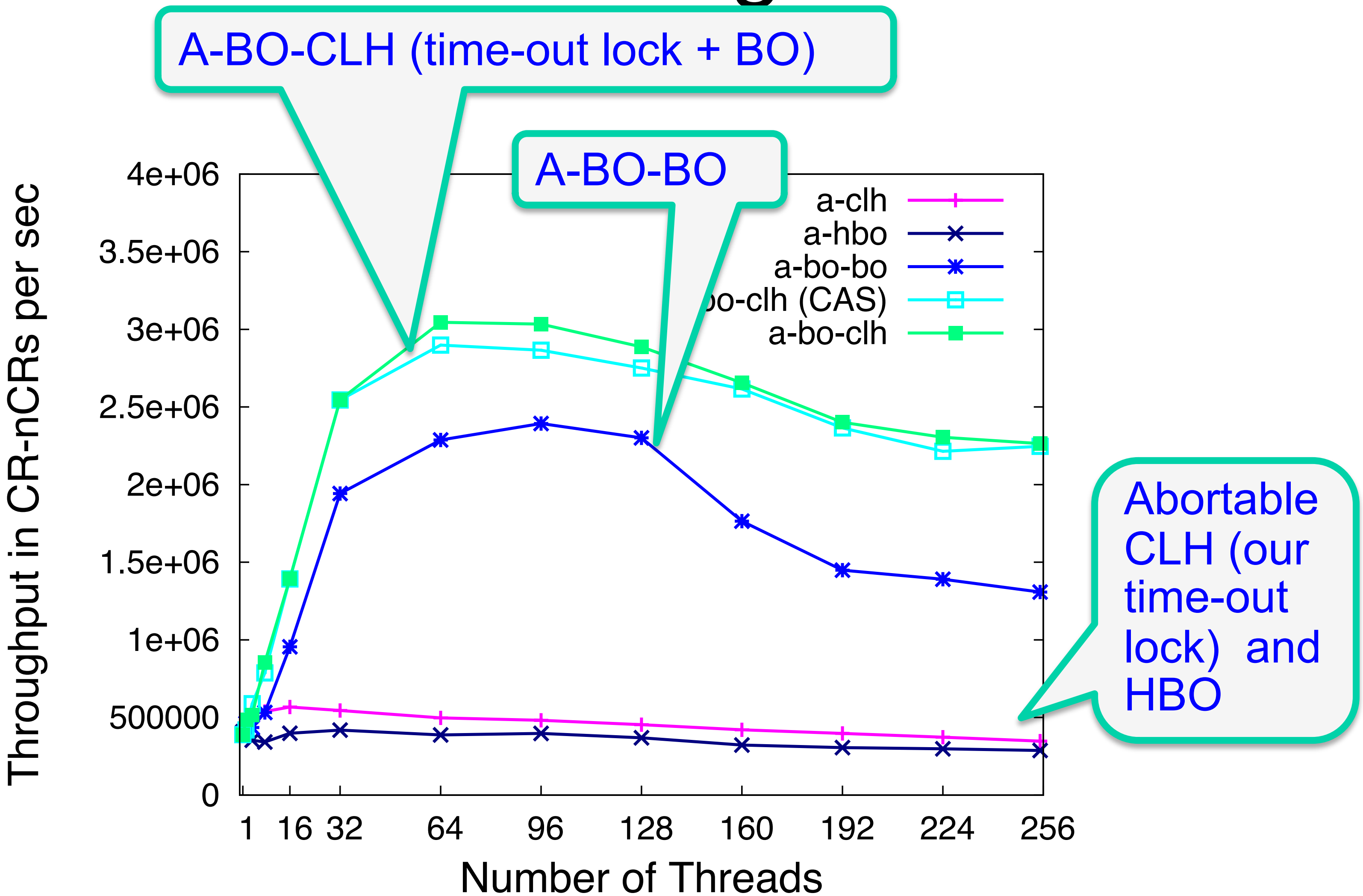
- Advantages:
  - Great locality
  - Low contention on shared lock
  - Practically no tuning
  - Has whatever properties you want:
    - Can be more or less fair, abortable…

    just choose the appropriate type of locks…
- Disadvantages:
  - Must tune fairness parameters

Lock Cohorting



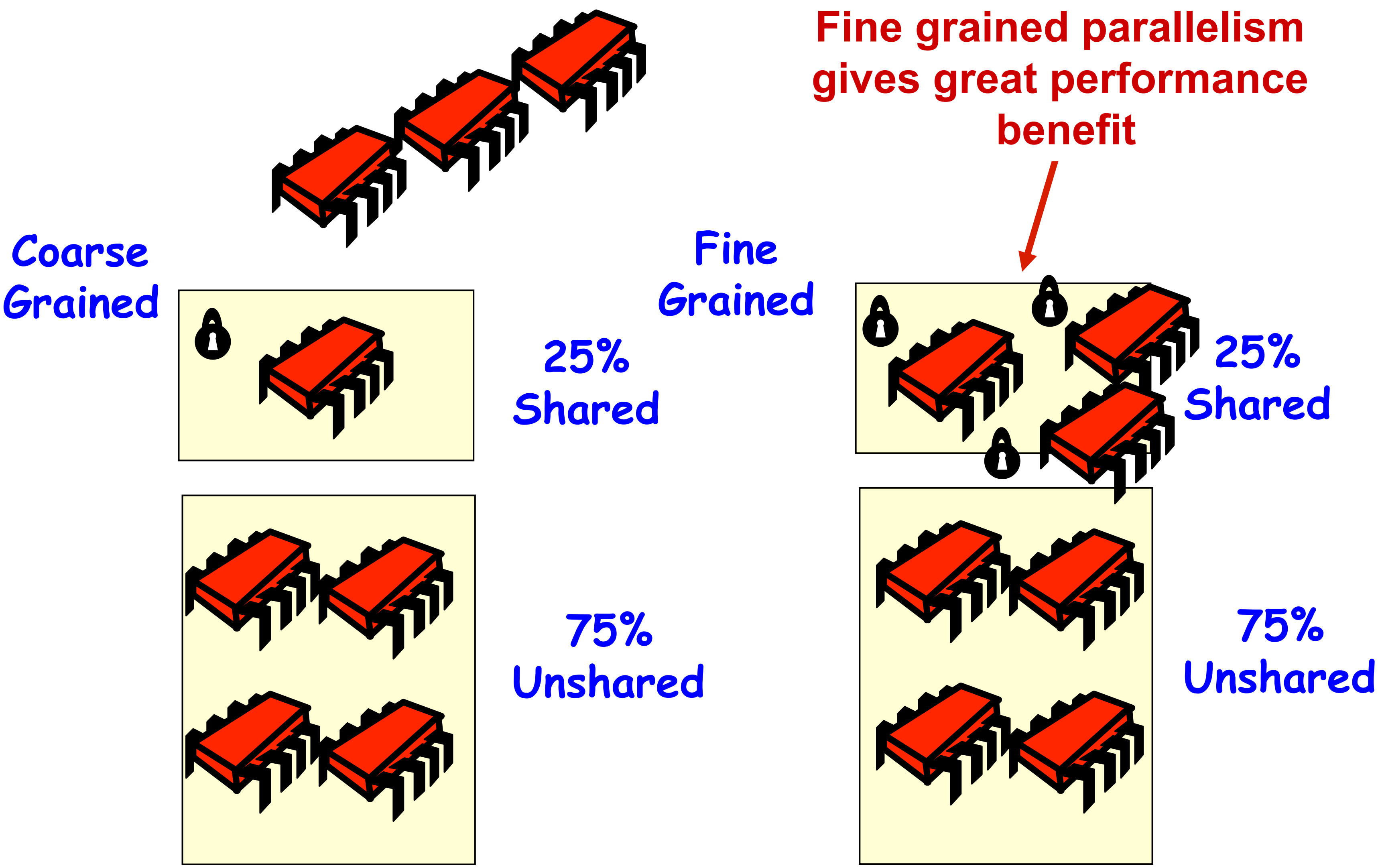
Time-Out (Abortable) Lock Cohorting



One Lock To Rule Them All?

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

Yeahy! Amdahl's Law Works

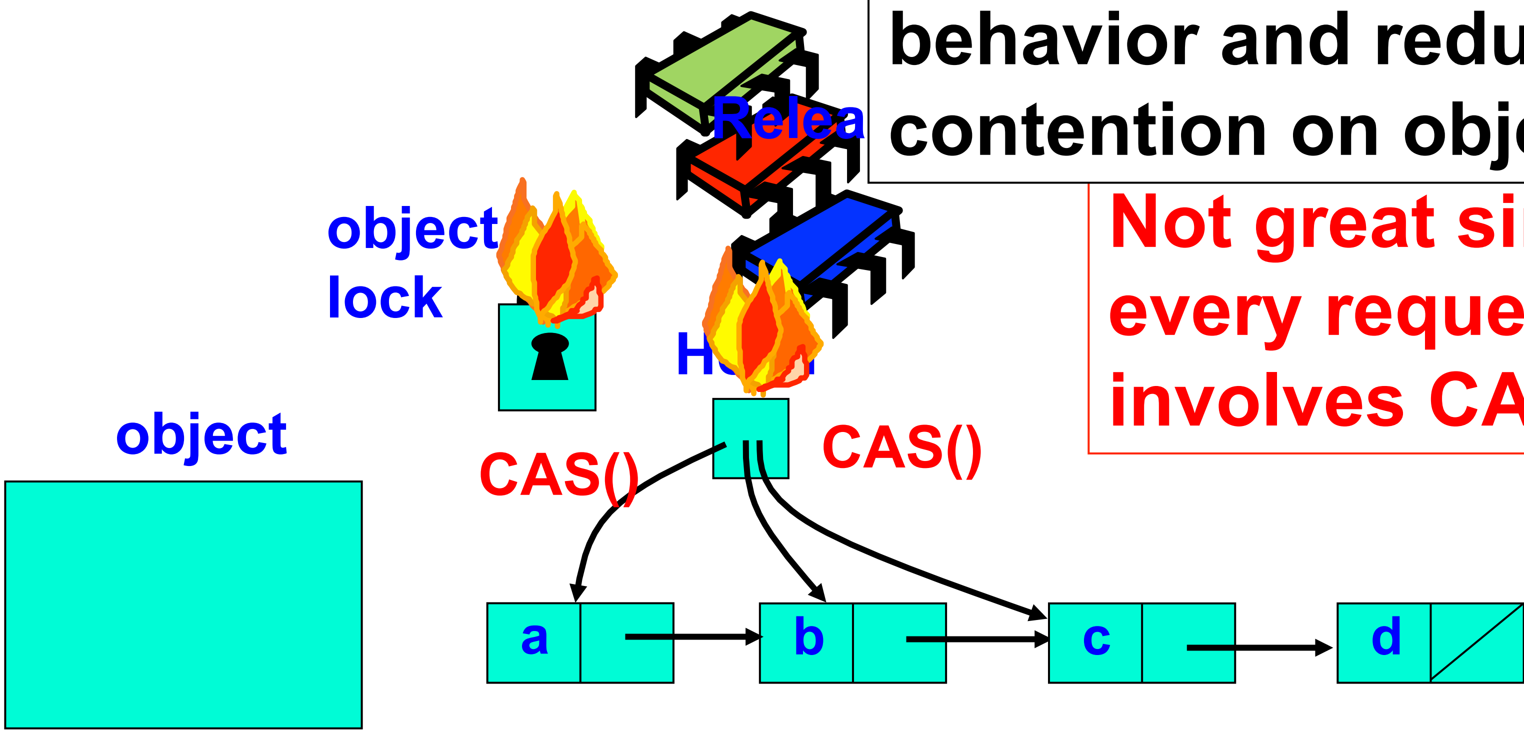


But…

- Can we always draw the right conclusions from Amdahl's law?
- Claim: sometimes the overhead of fine-grained synchronization is so high…that it is better to have a single thread do all the work sequentially in order to avoid it

Oyama et. al Mutex

Improves cache behavior and reduces contention on object



Not great since every request involves CAS

Apply a,b,c, and d to object

return responses

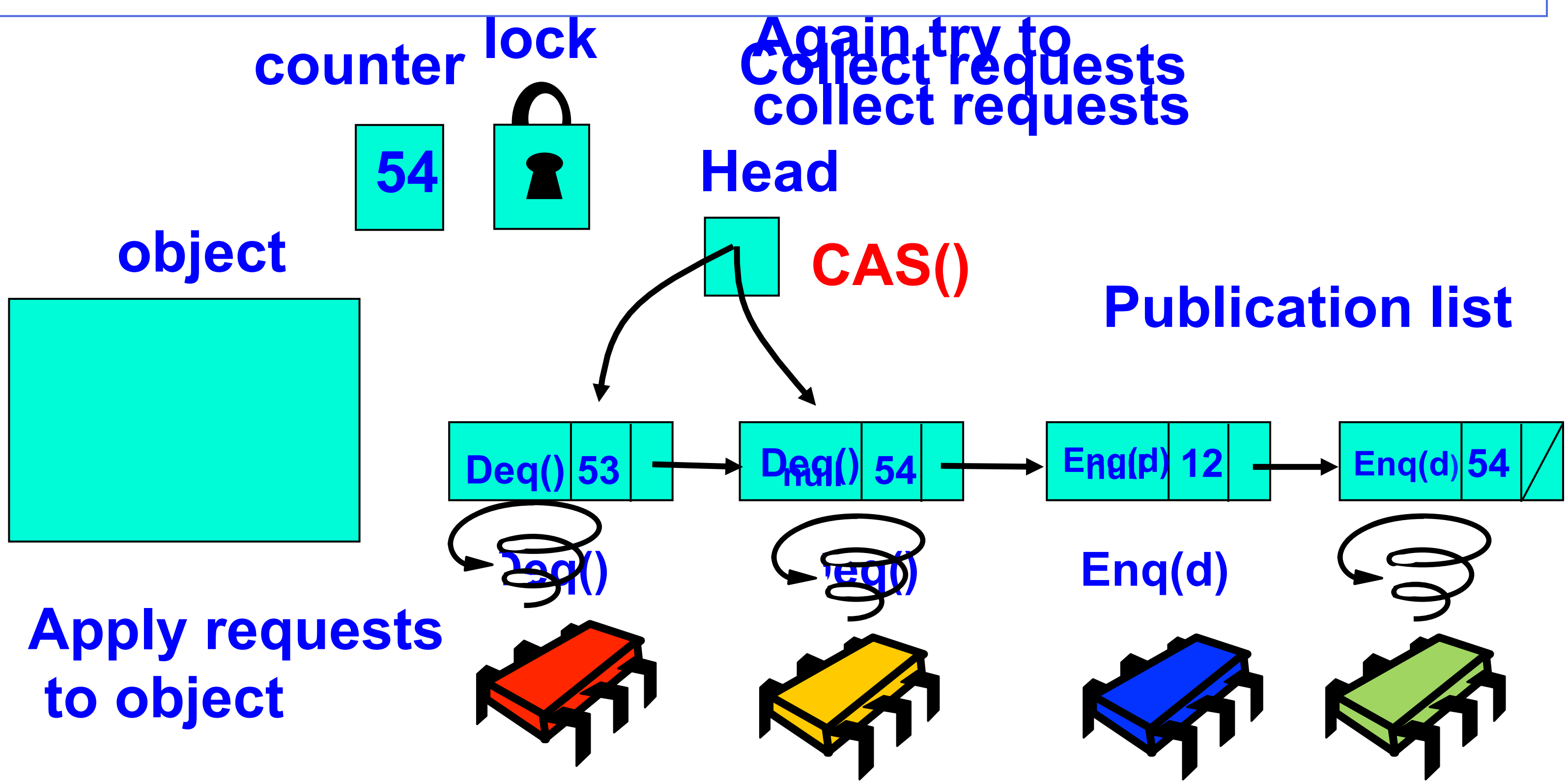
Combine lock requests

Flat Combining

- Have single lock holder collect and perform requests of all others
  - Without using CAS operations to coordinate requests
  - With (non-naïve) combining of requests (if cost of k batched operations is less than that of k operations in sequence → we win)

Flat-Combining

Most requests do not involve a CAS, in fact, not even a memory barrier



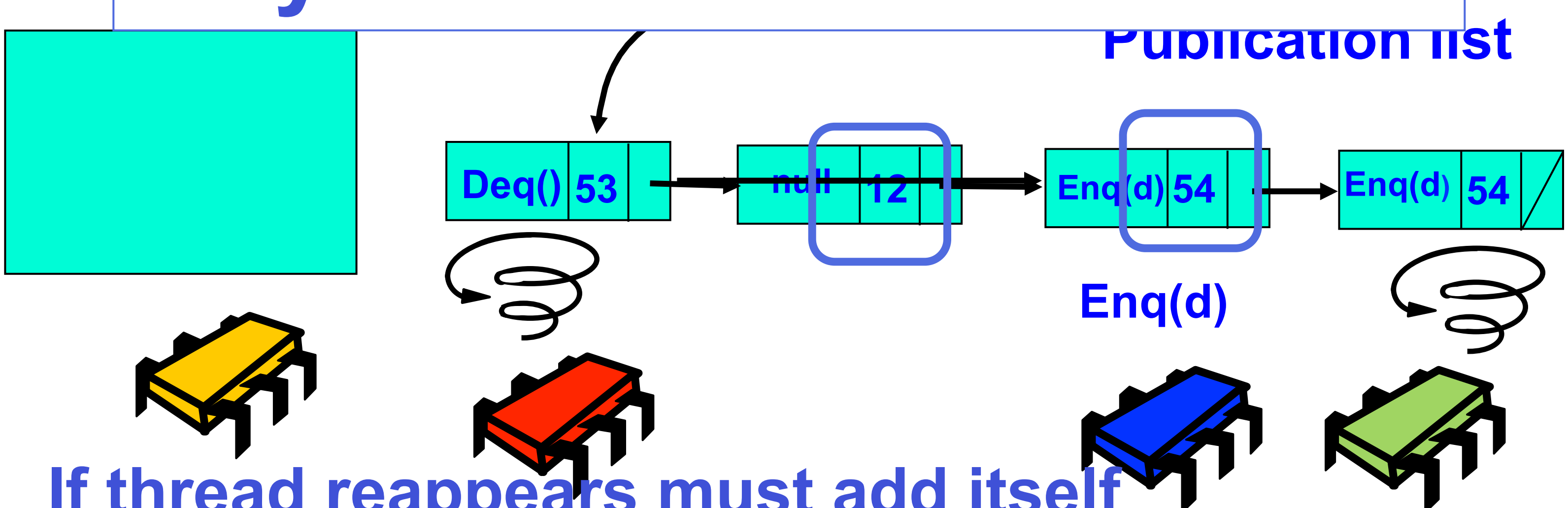
Flat-Combining Pub-List Cleanup

Every combiner increments counter and updates record's time stamp when returning response

Traverse and remove from

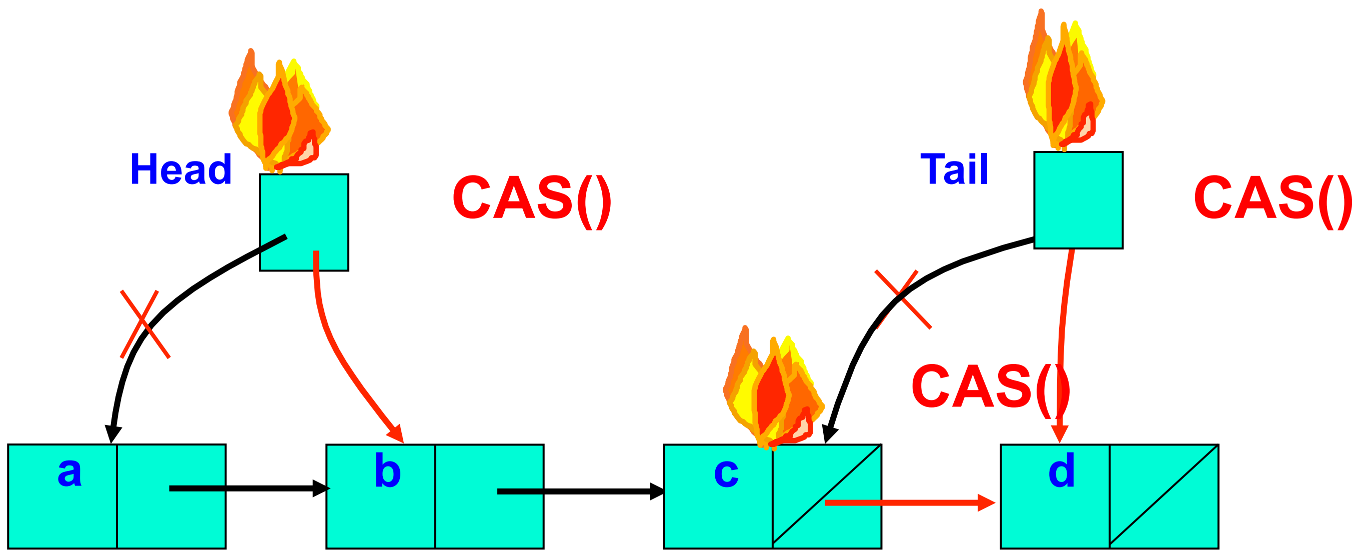
ime

Cleanup requires no CAS, only reads and writes



If thread reappears must add itself to pub list

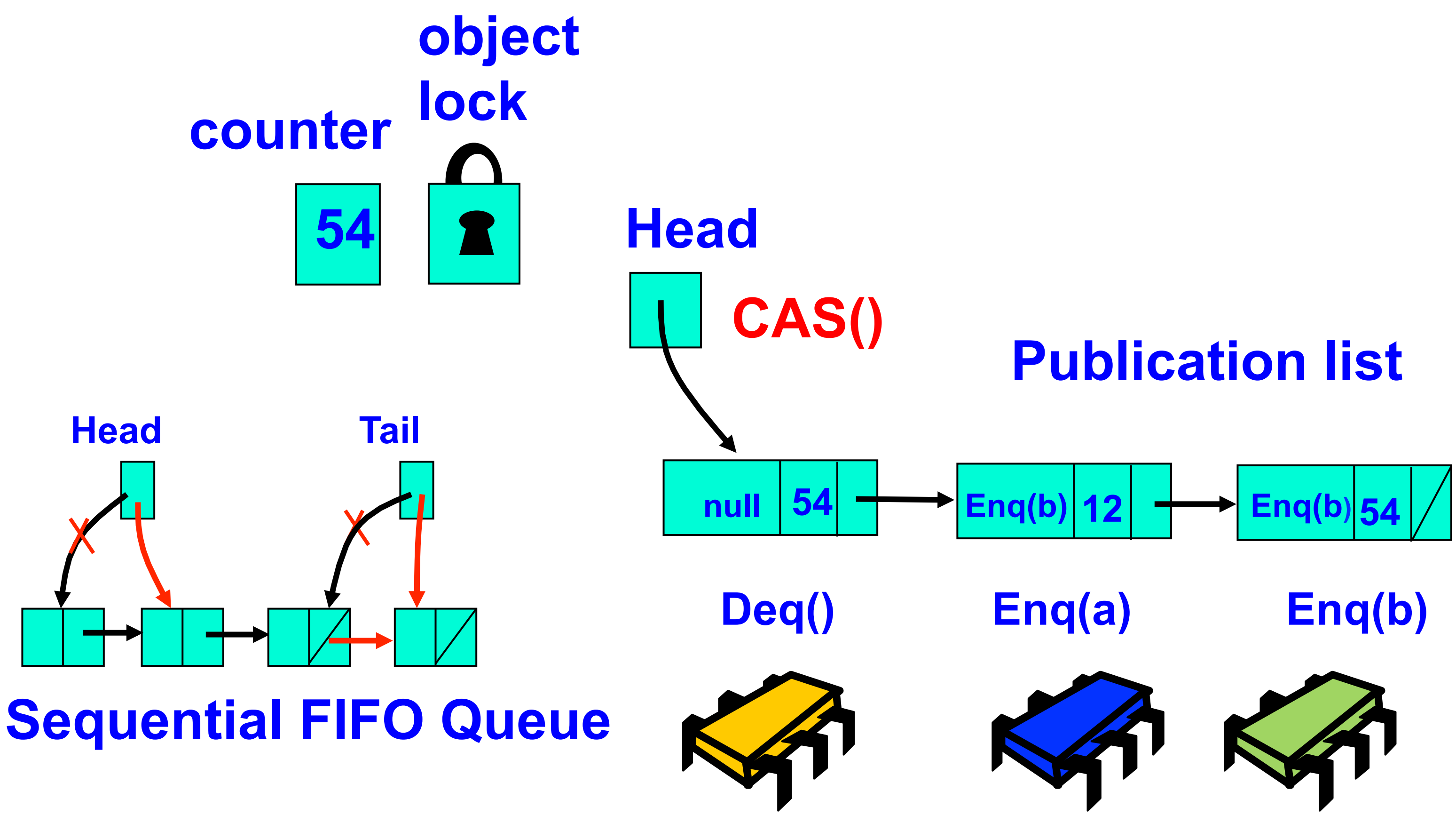
Fine-Grained Lock-free FIFO Queue



P: Dequeue() => a

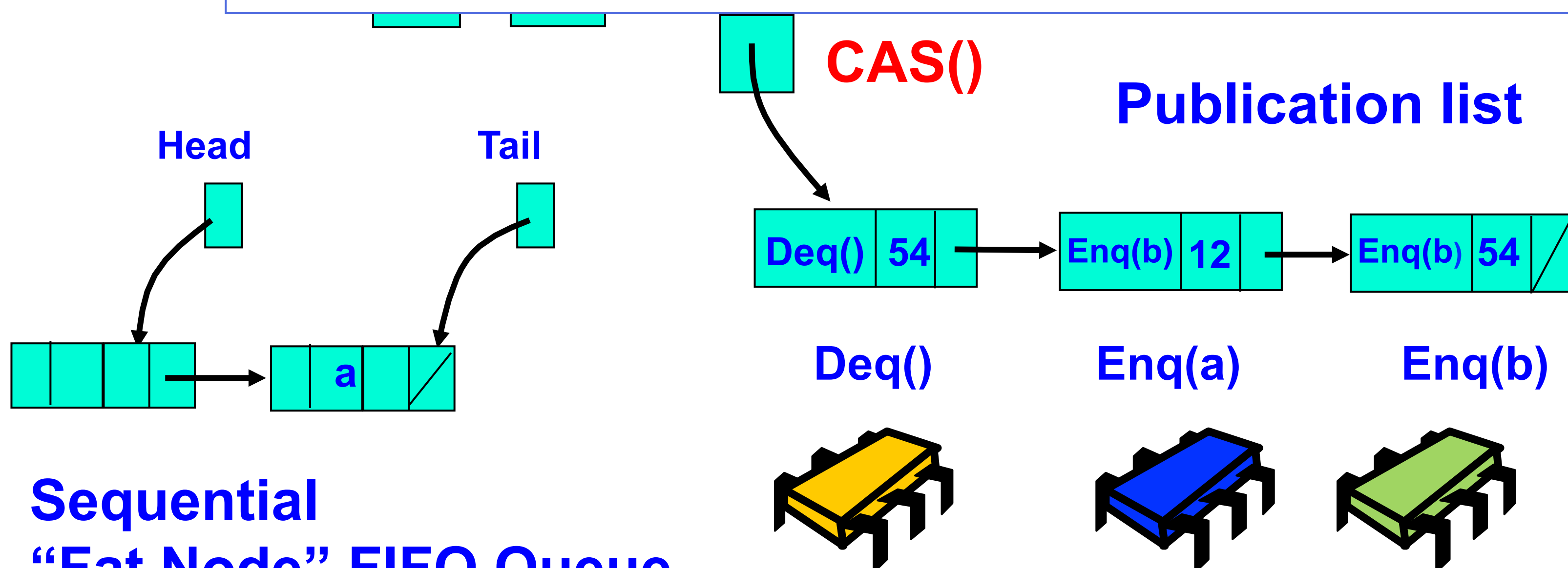
Q: Enqueue(d)

Flat-Combining FIFO Queue



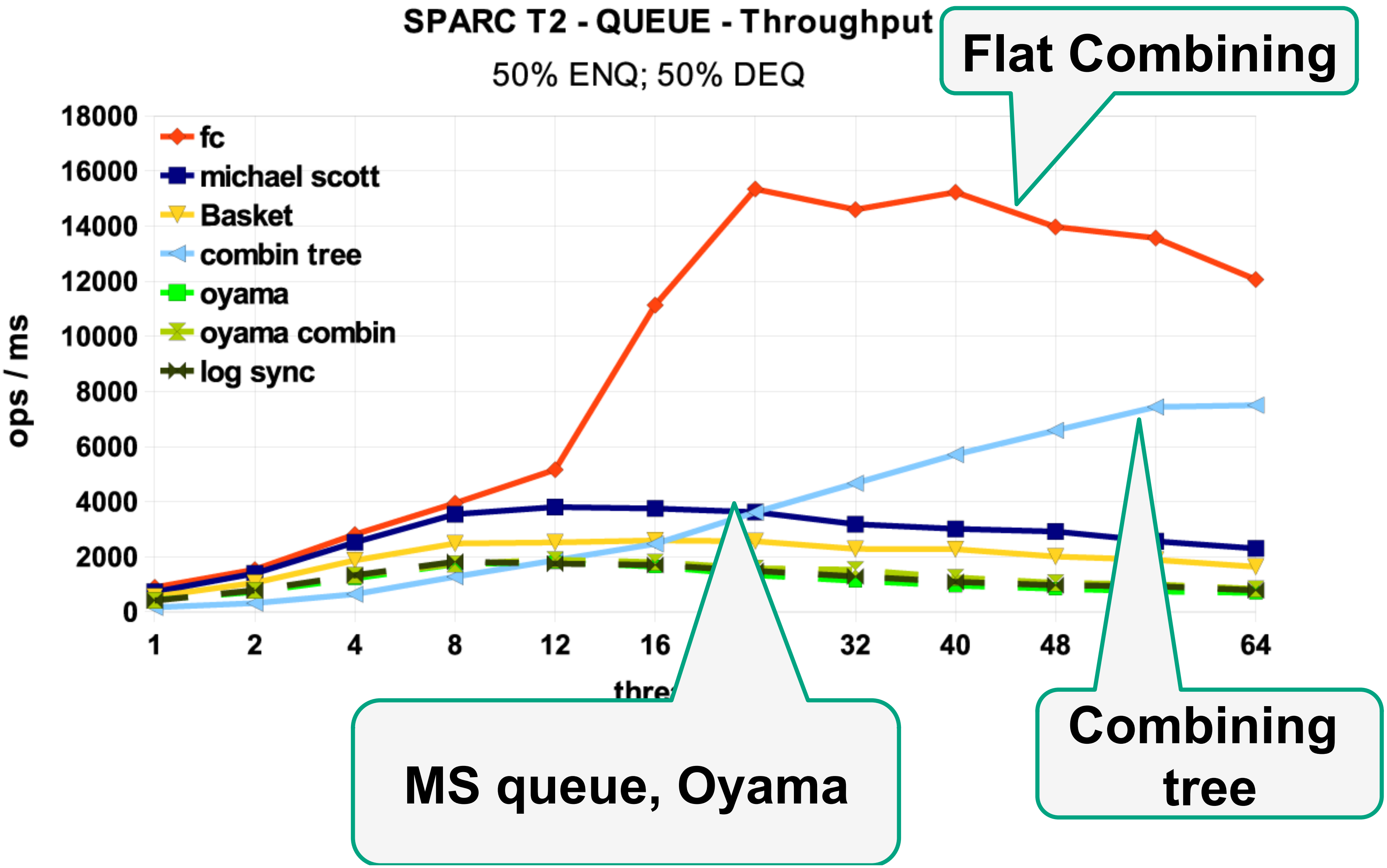
OK, but can do better…combining: collect all items into a “fat node”, enque…

“Fat Node” easy sequentially but cannot be done in concurrent alg without CAS

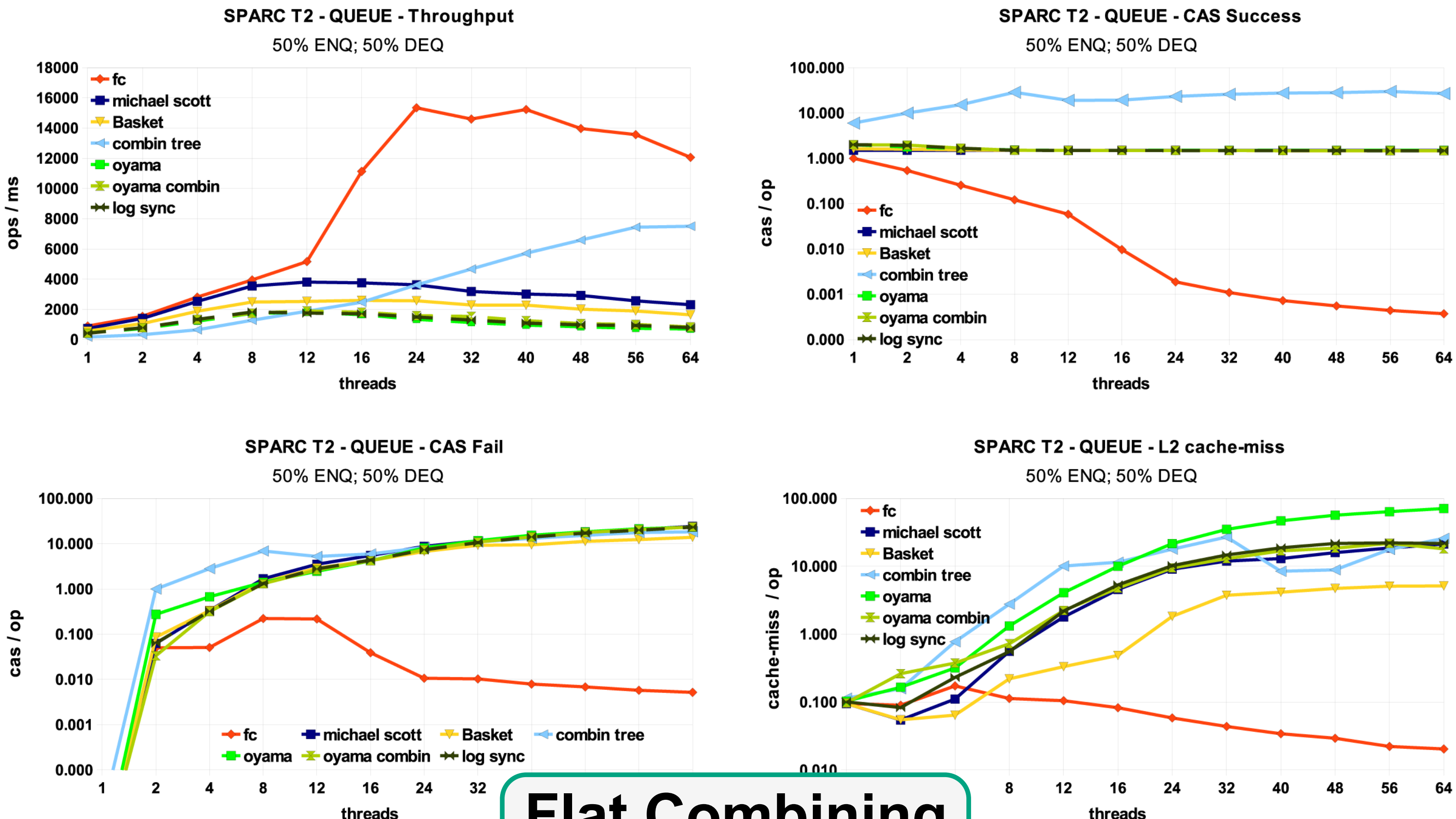


Sequential
“Fat Node” FIFO Queue

Linearizable FIFO Queue

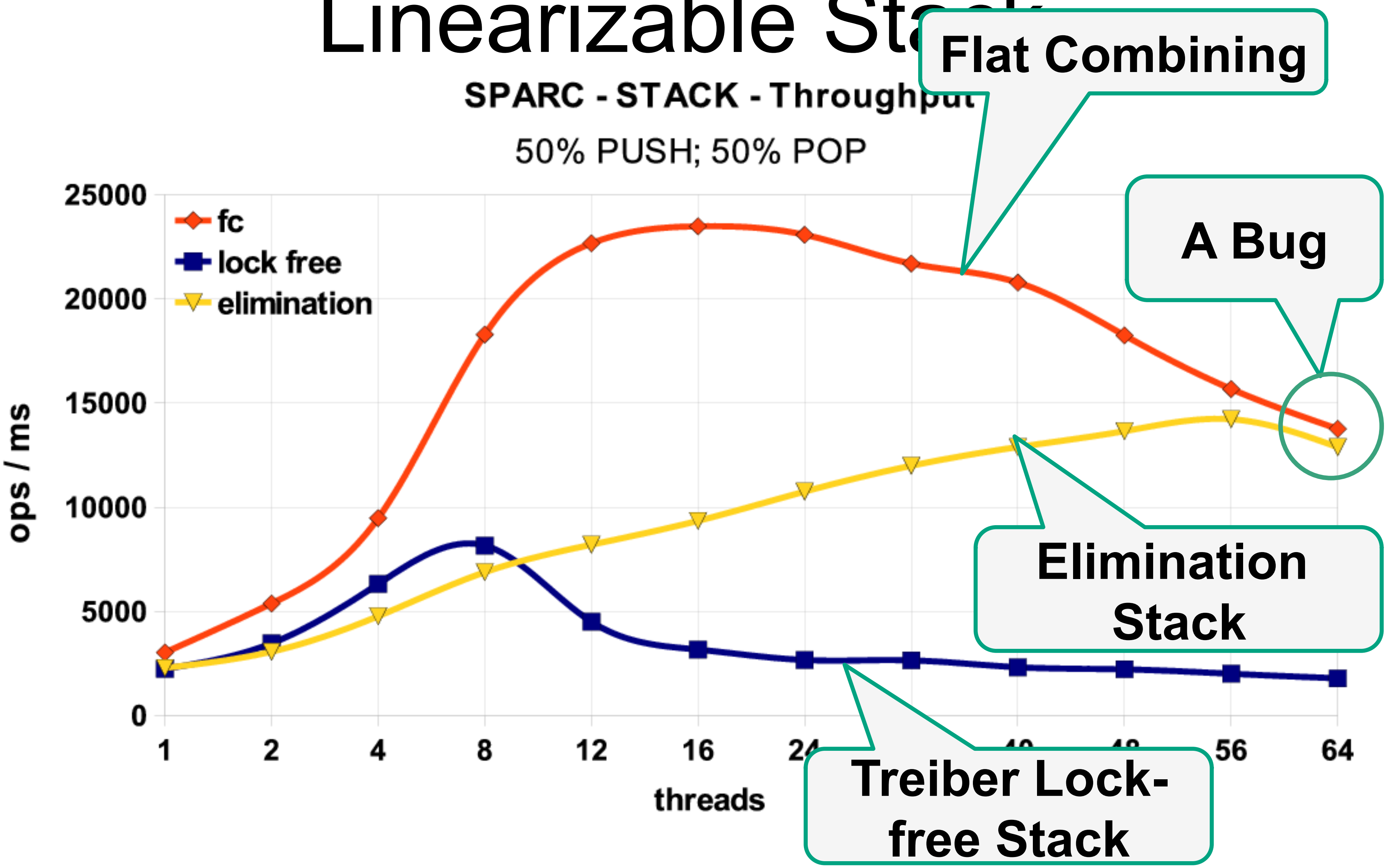


Benefit's of Flat Combining



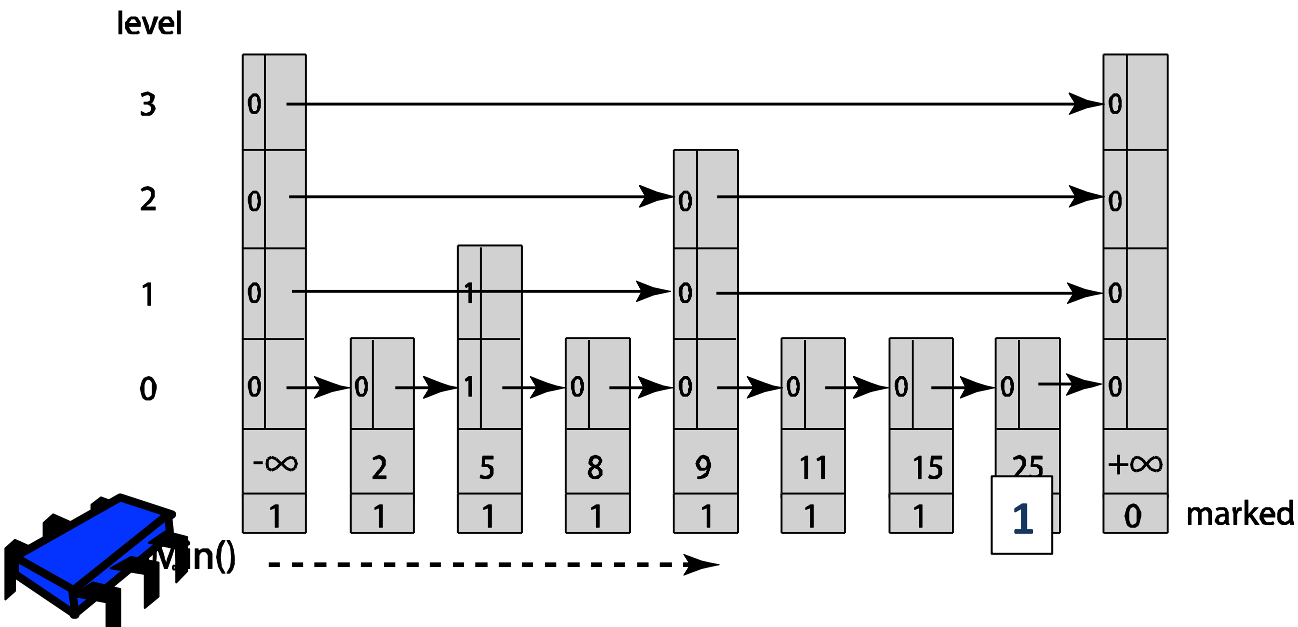
Flat Combining in Red

Linearizable Stack



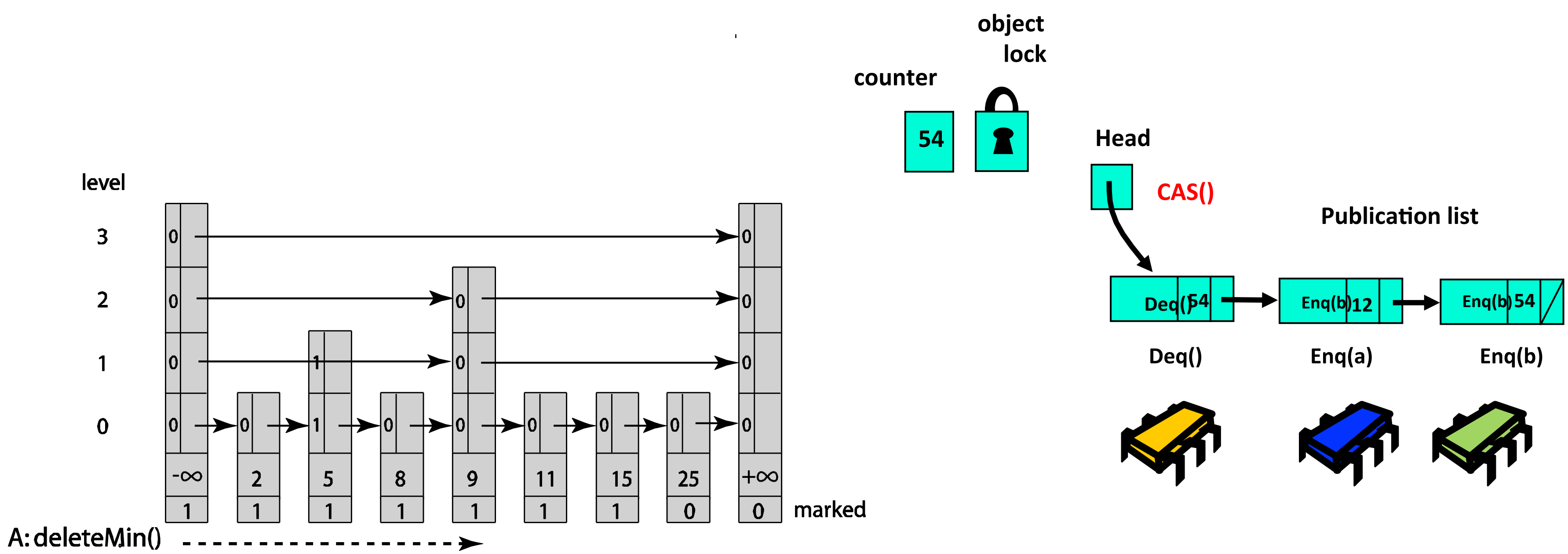
Concurrent Priority Queue (Chapter 15)

k deleteMin operations take O(k log n)



deleteMin() traverses CASing until you manage to mark a node, then use skiplist remove your marked node

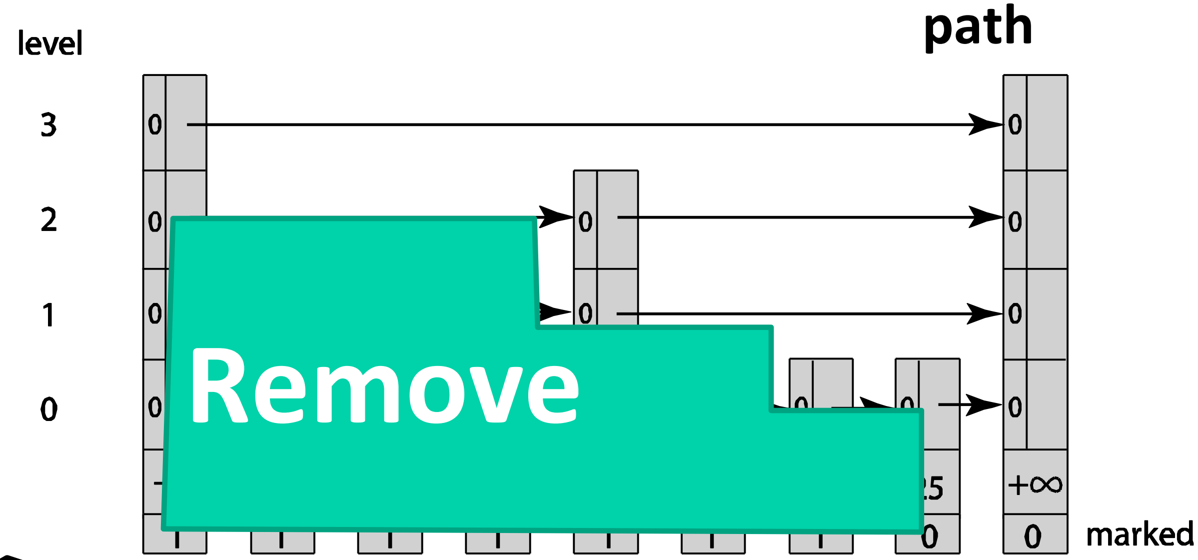
Flat-Combining Priority Queue

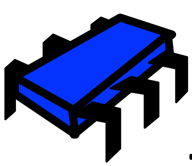


Flat Combining Priority Queue

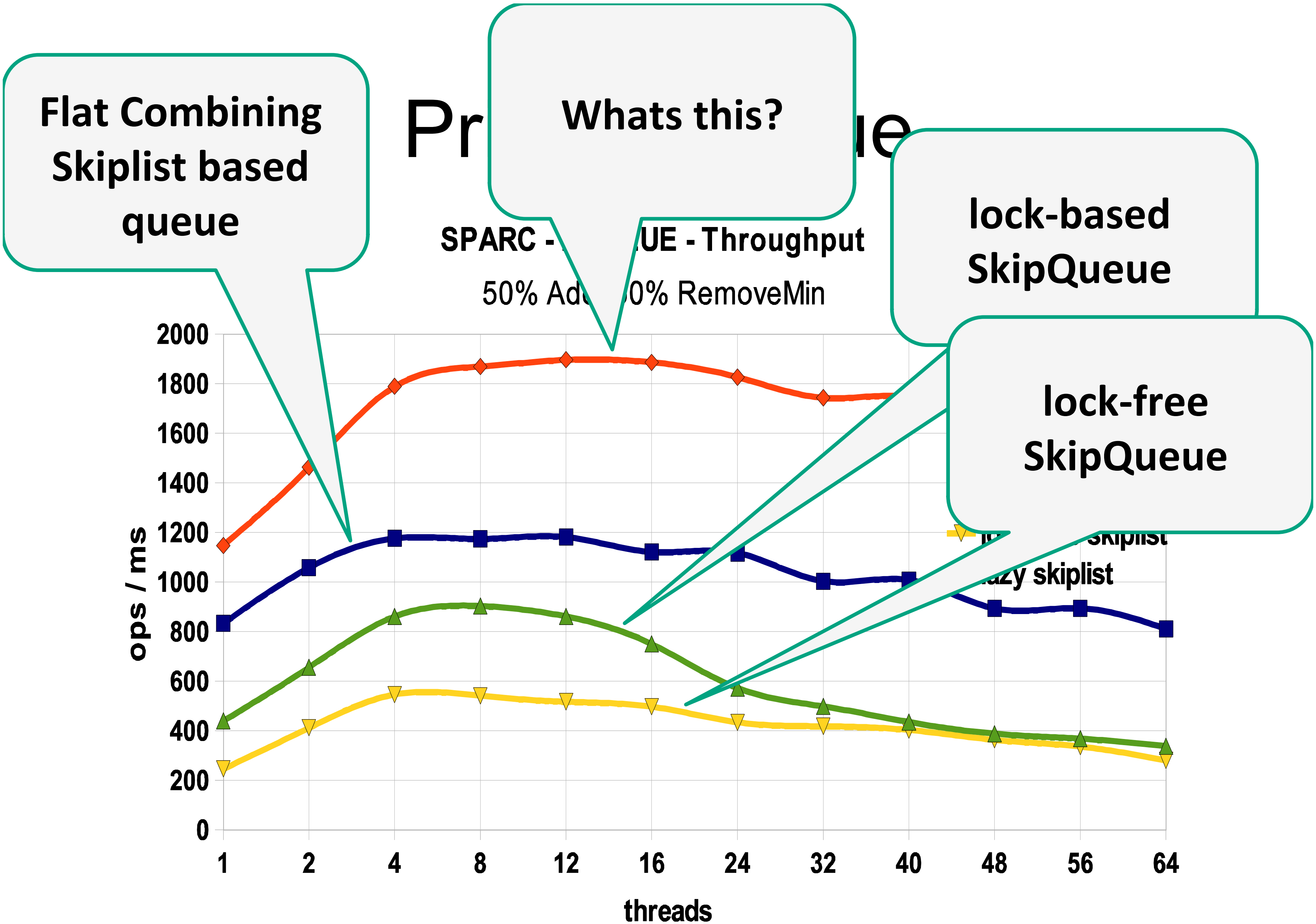
k deleteMin operations take O(k+log n)

removing all nodes below your path



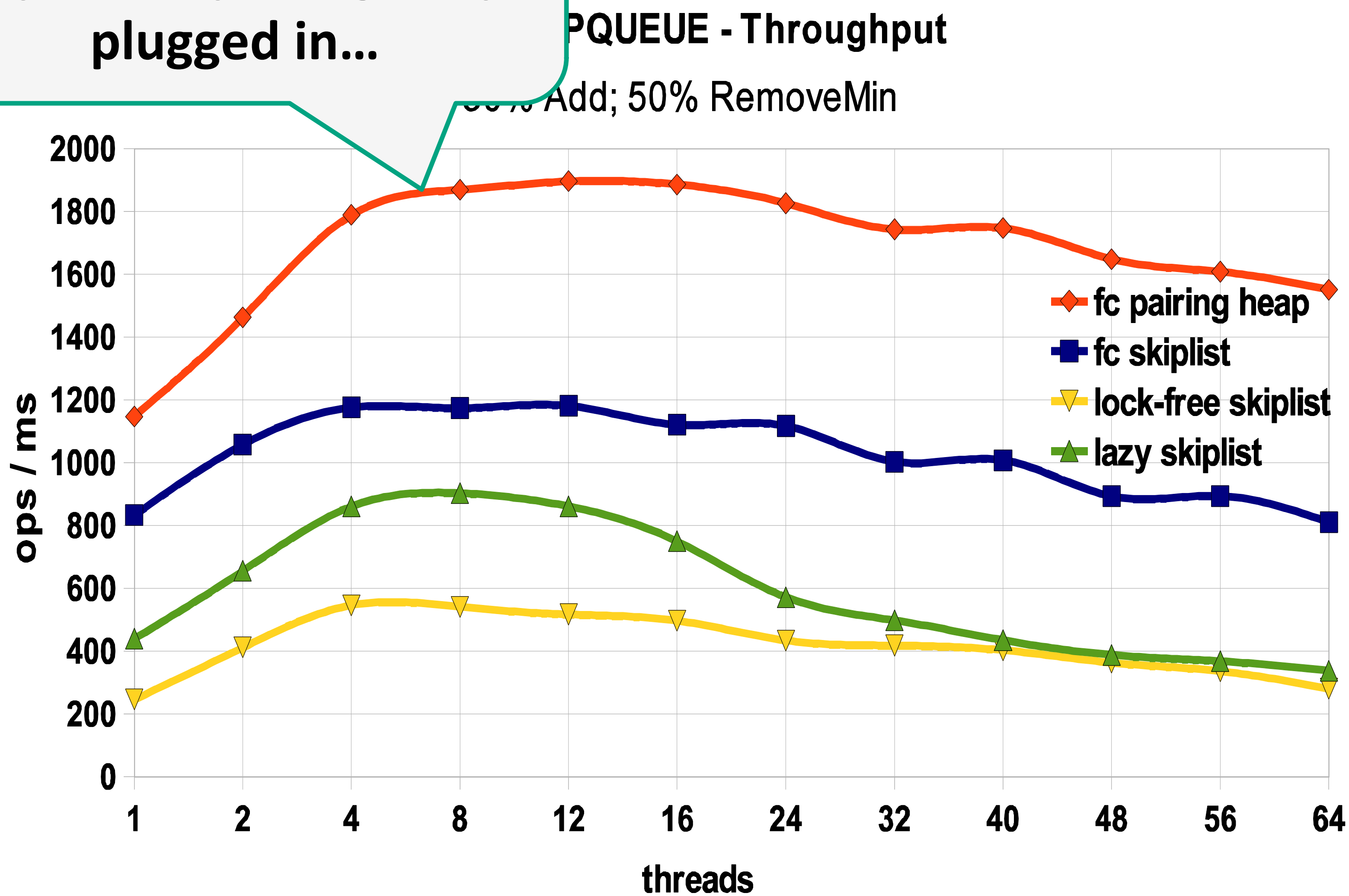


traverse to find kth key, collect values to be returned

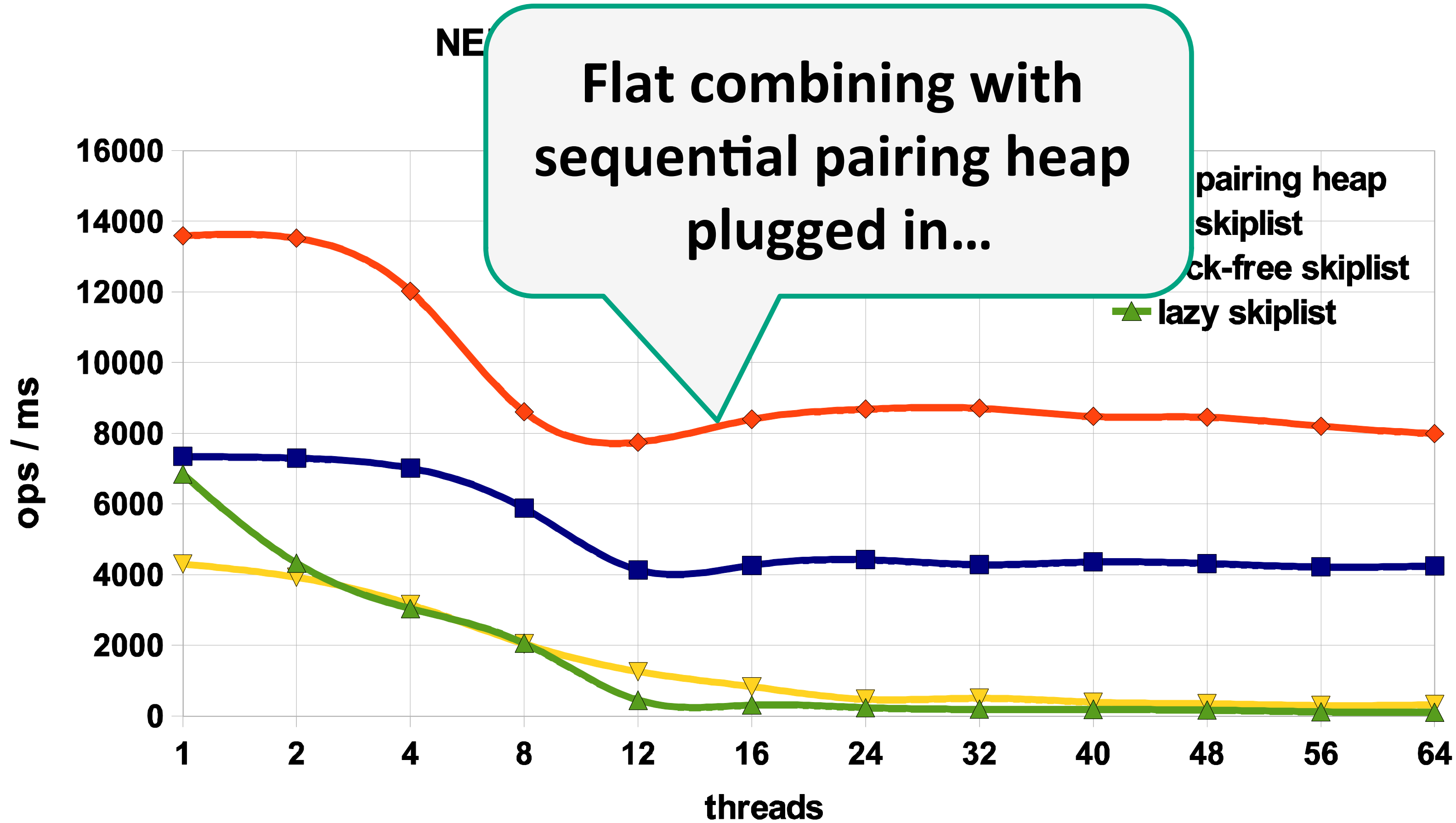

Flat Combining Skiplist based queue
Whats this?
lock-based SkipQueue
lock-free SkipQueue
Pr
ue
SPARC -
UE - Throughput
50% Ad
0% RemoveMin
ops / ms
2000
1800
1600
1400
1200
1000
800
600
400
200
0
1
2
4
8
12
16
24
32
40
48
56
64
threads
skiplist
azy skiplist

Flat combining with sequential pairing heap plugged in…

ritv Queue



Priority Queue on Intel



Don’t be Afraid of the Big Bad Lock

- Fine grained parallelism comes with an overhead…not always worth the effort.
- Sometimes using a single global lock is a win.

Thanks!