Concurrent programming

Spin Locks and Contention

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy, Nir Shavit, Victor Luchangco,
and Michael Spear

Modified by Piotr Witkowski

Focus so far: Correctness and Progress

- Models
  - Accurate (we never lied to you)
  - But idealized (so we forgot to mention a few things)
- Protocols
  - Elegant
  - Important
  - But naïve

New Focus: Performance

- Models
  - More complicated (not the same as complex!)
  - Still focus on principles (not soon obsolete)
- Protocols
  - Elegant (in their fashion)
  - Important (why else would we pay attention)
  - And realistic (your mileage may vary)

Kinds of Architectures

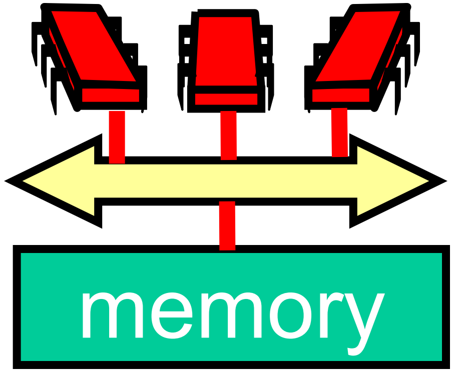
- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

Kinds of Architectures

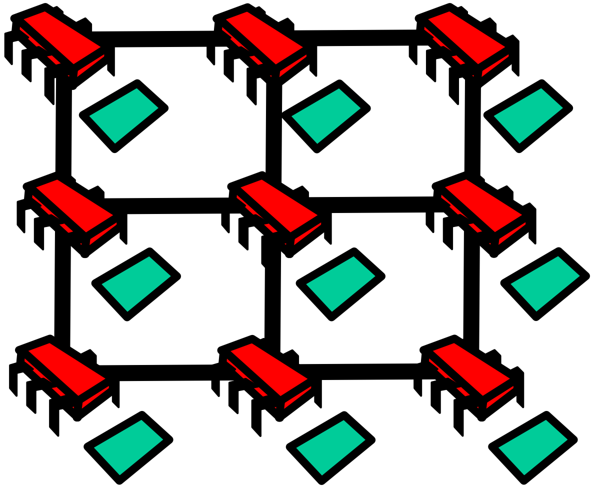
- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

Our space

MIMD Architectures



Shared Bus



Distributed

- Memory Contention
- Communication Contention
- Communication Latency

Today: Revisit Mutual Exclusion

- Performance, not just correctness
- Proper use of multiprocessor architectures
- A collection of locking algorithms…

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

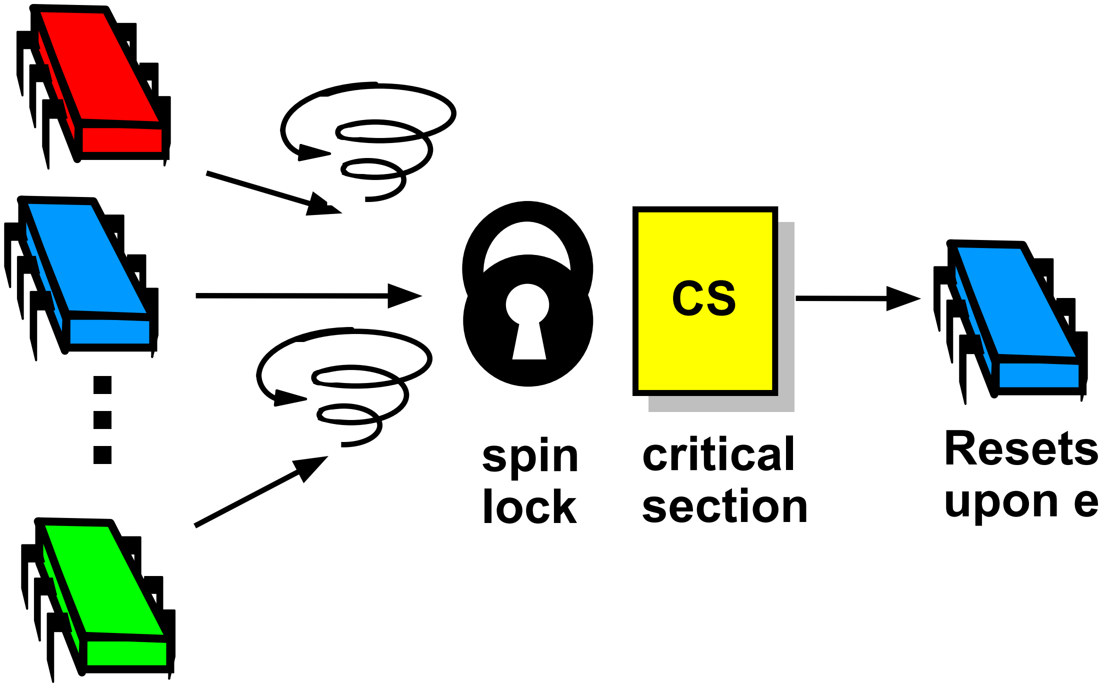
- Keep trying
  - “spin” or “busy-wait”
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

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



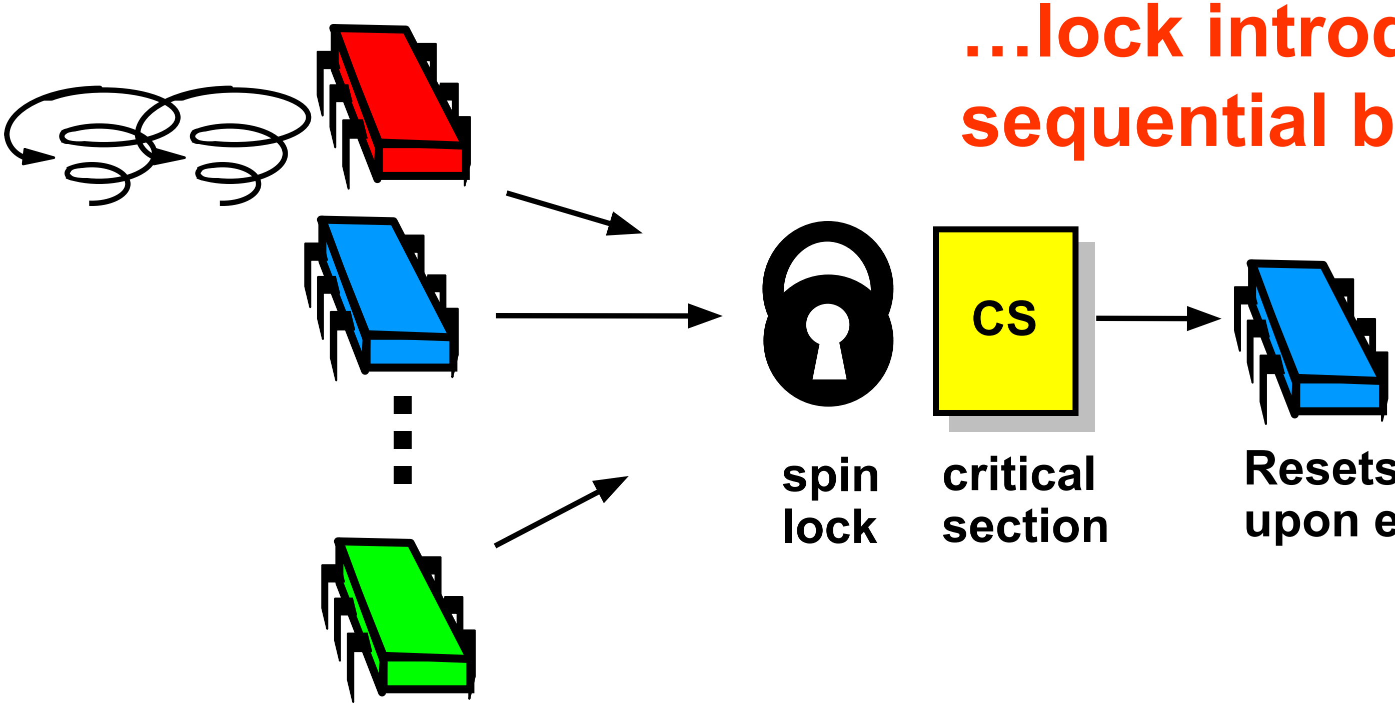
spin lock

critical section

Resets lock upon exit

Basic Spin-Lock

…lock introduces sequential bottleneck



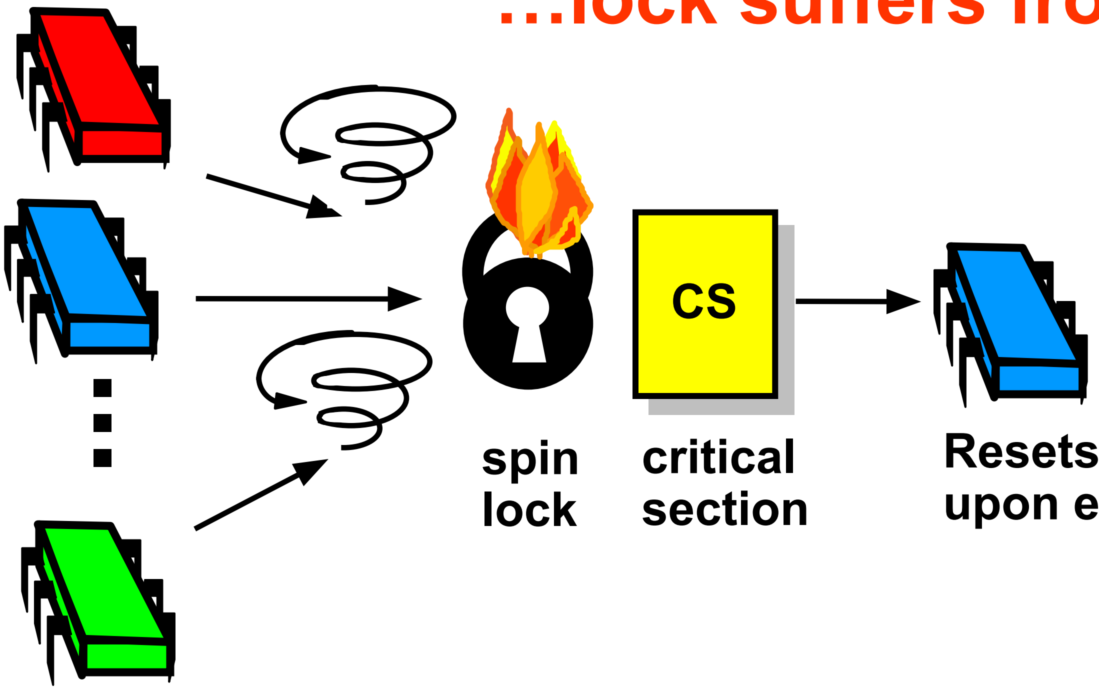
spin lock

critical section

Resets lock upon exit

Basic Spin-Lock

…lock suffers from contention



CS

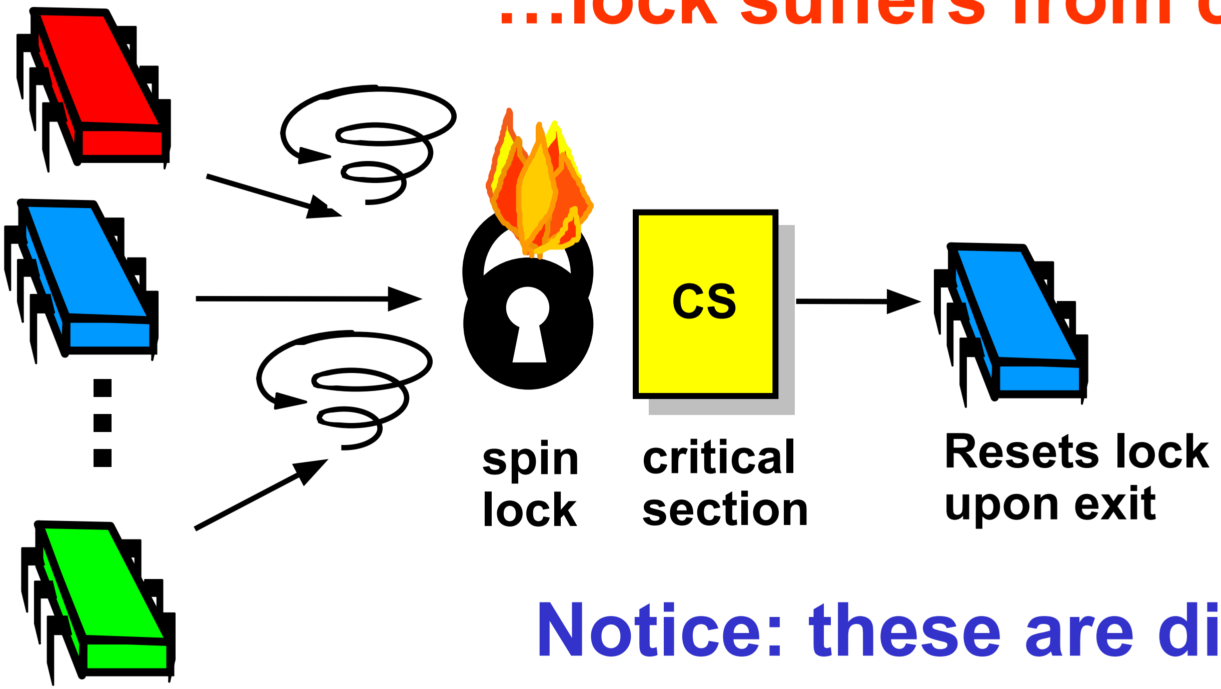
spin
lock

critical
section

Resets lock
upon exit

Basic Spin-Lock

…lock suffers from contention



CS

spin lock

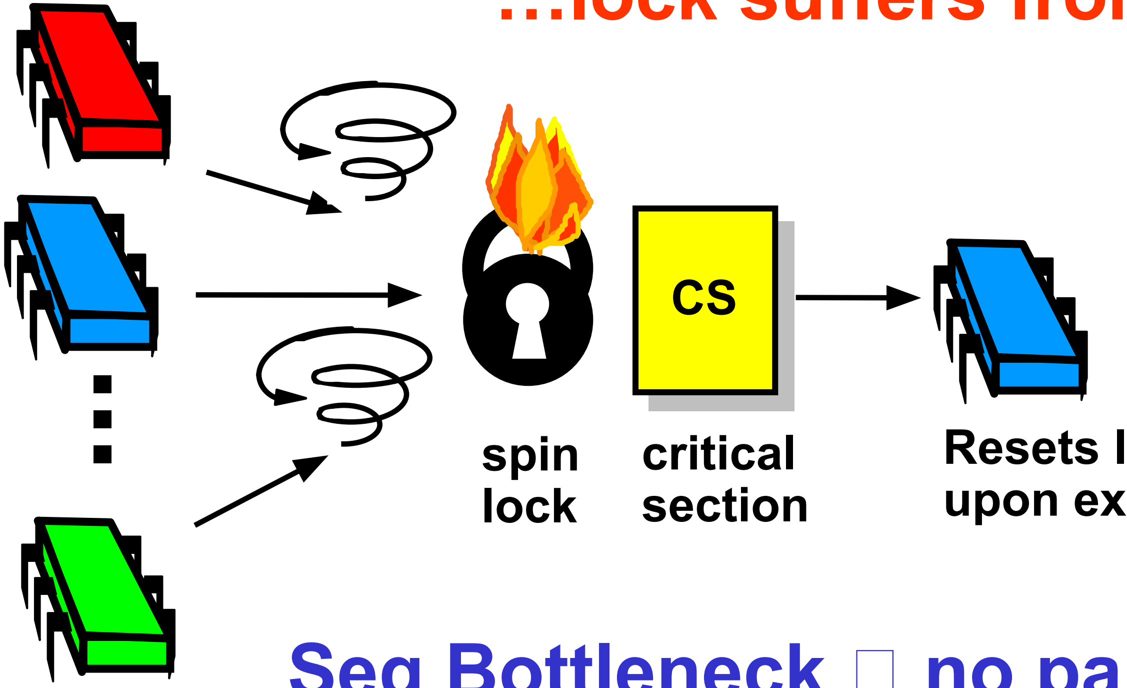
critical section

Resets lock upon exit

Notice: these are distinct phenomena

Basic Spin-Lock

...lock suffers from contention



CS

spin lock

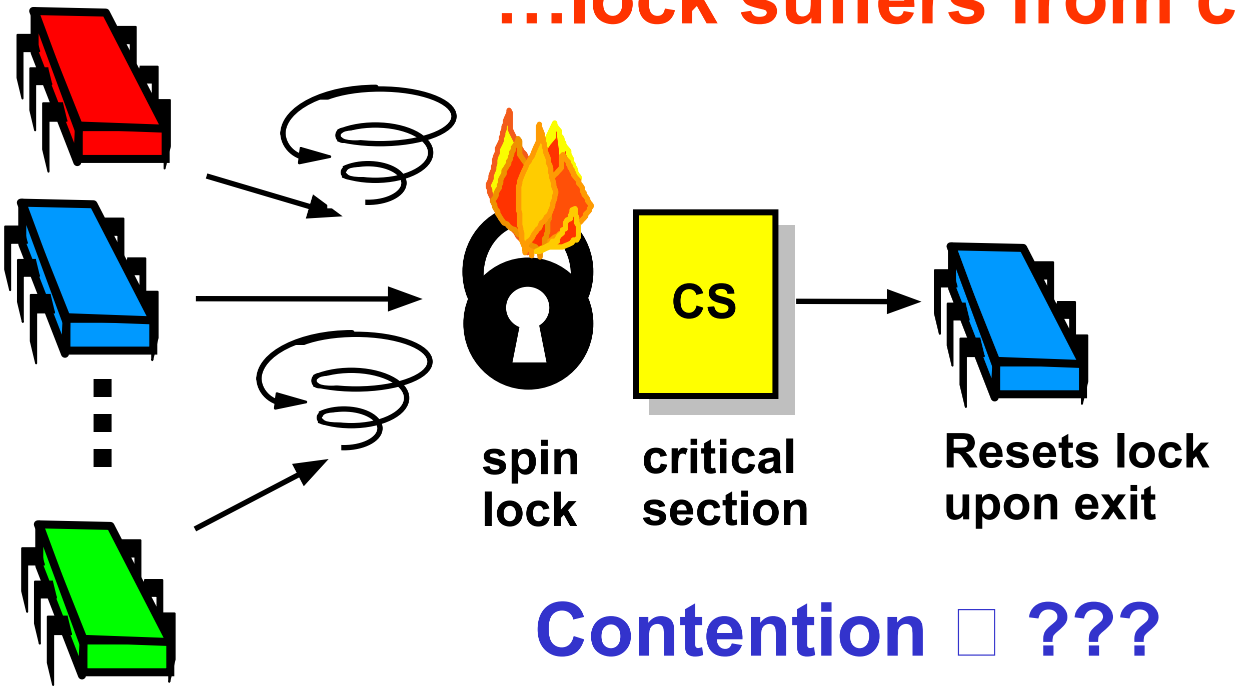
critical section

Resets lock upon exit

Seq Bottleneck □ no parallelism

Basic Spin-Lock

…lock suffers from contention



spin
lock

critical
section

Resets lock
upon exit

Contention □ ???

Review: Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap **true** with current value
  - Return value tells if prior value was **true** or **false**
- Can reset just by writing **false**
- TAS aka “getAndSet”

Review: Test-and-Set

```
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

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

Test-and-set Lock

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

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

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

Test-and-set Lock

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

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

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

Lock state is AtomicBoolean

Test-and-set Lock

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

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

 void unlock() {
  sta
 }}
```

Keep trying until lock acquired

Test-and-set Lock

```
class TASlock {
 AtomicB
  new At

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

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

Release lock by resetting state to false

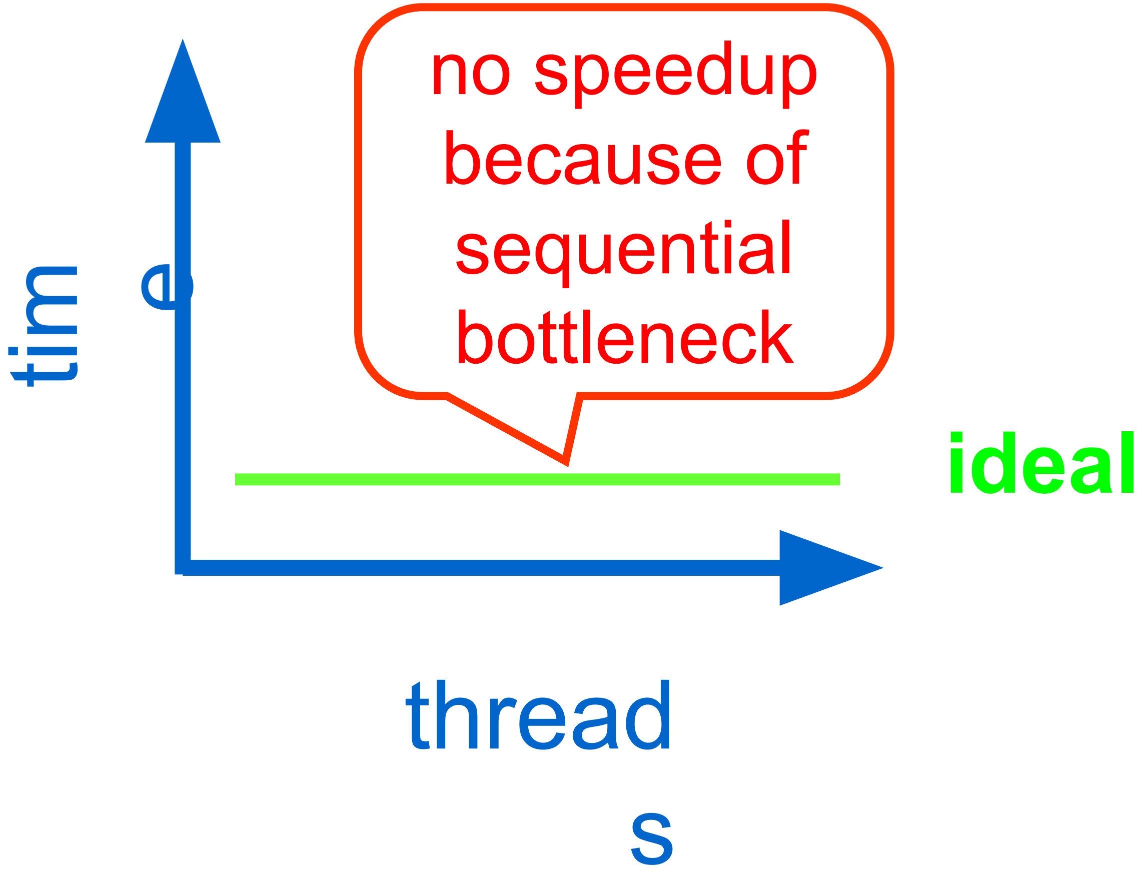
Space Complexity

- TAS spin-lock has small “footprint”
- N thread spin-lock uses O(1) space
- As opposed to O(n) Peterson/Bakery
- How did we overcome the $\Omega(n)$ lower bound?
- We used a RMW operation…

Performance

- Experiment
  - *n* threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?

Graph



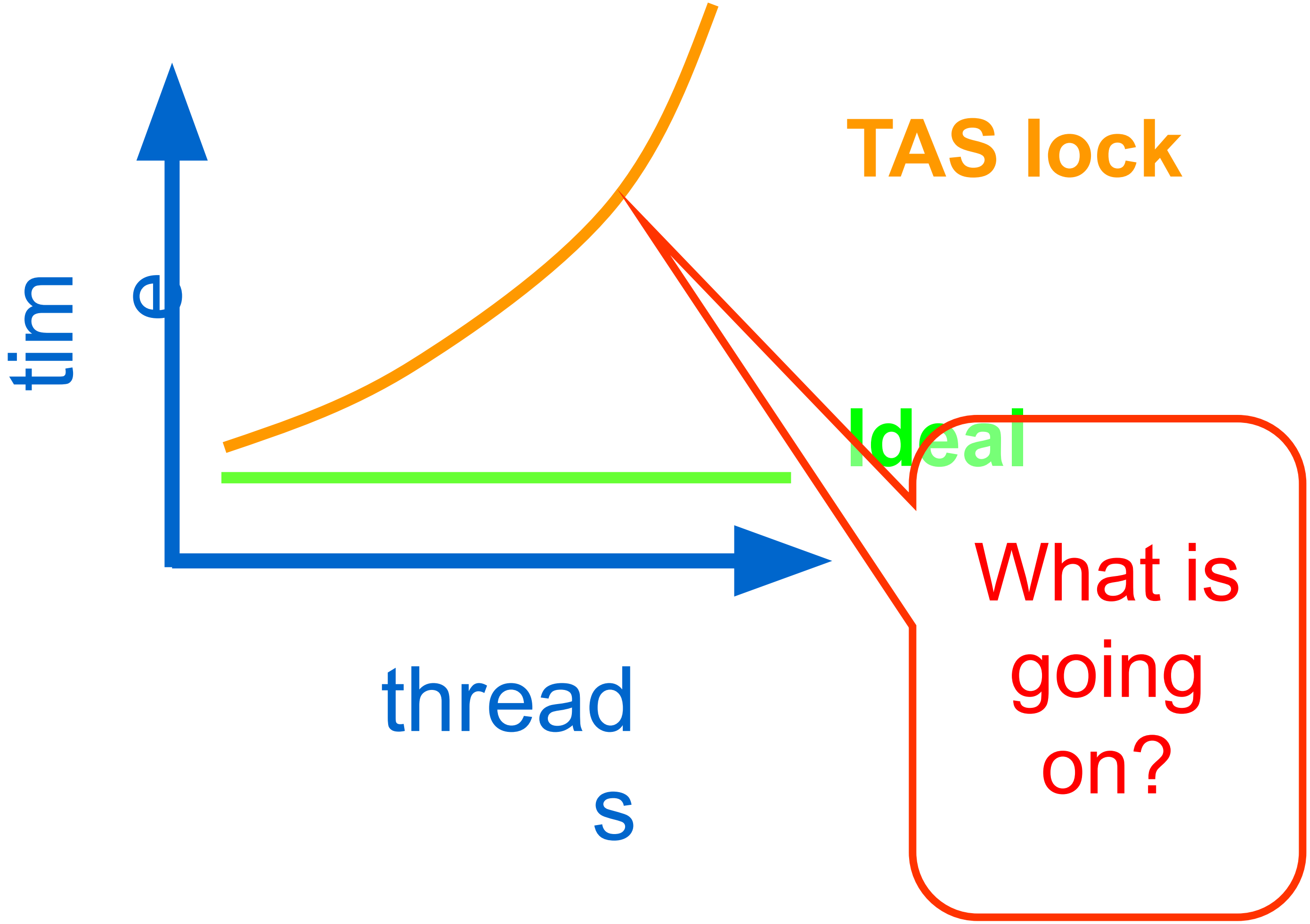
no speedup because of sequential bottleneck

ideal

time

threads

Mystery #1



What is going on?

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

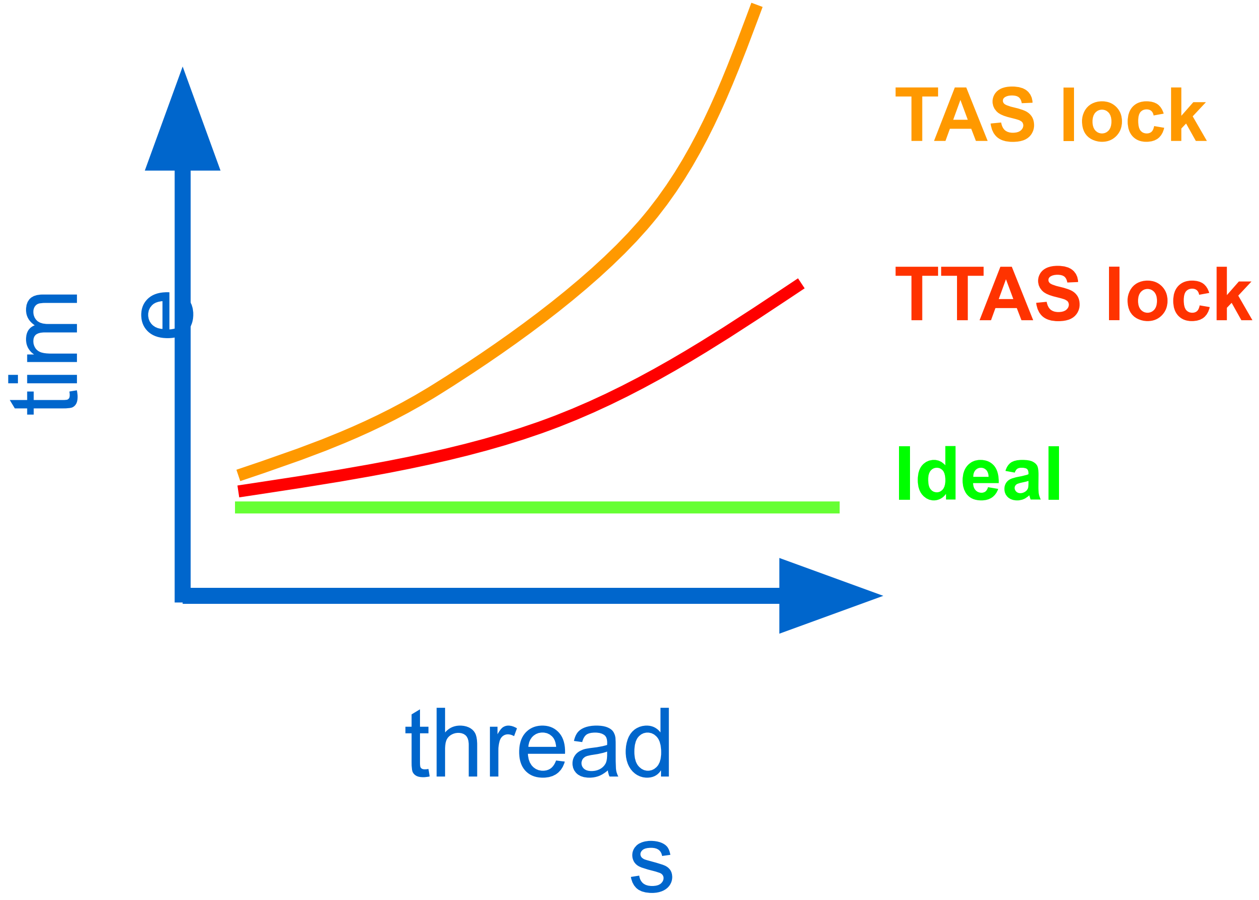
Test-and-test-and-set Lock

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

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

Then try to acquire it

Mystery #2



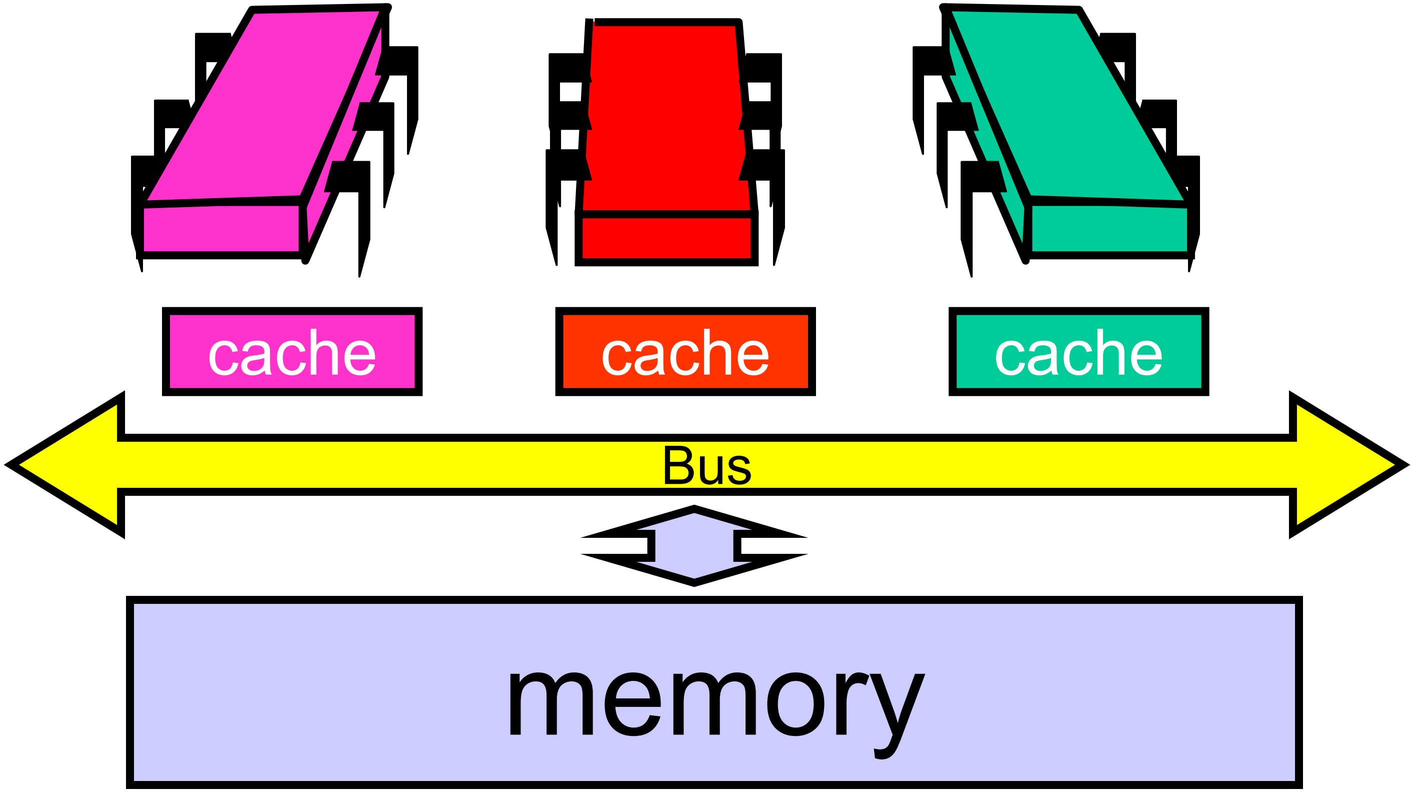
Mystery

- Both
  - TAS and TTAS
  - Do the same thing (in our model)
- Except that
  - TTAS performs much better than TAS
  - Neither approaches ideal

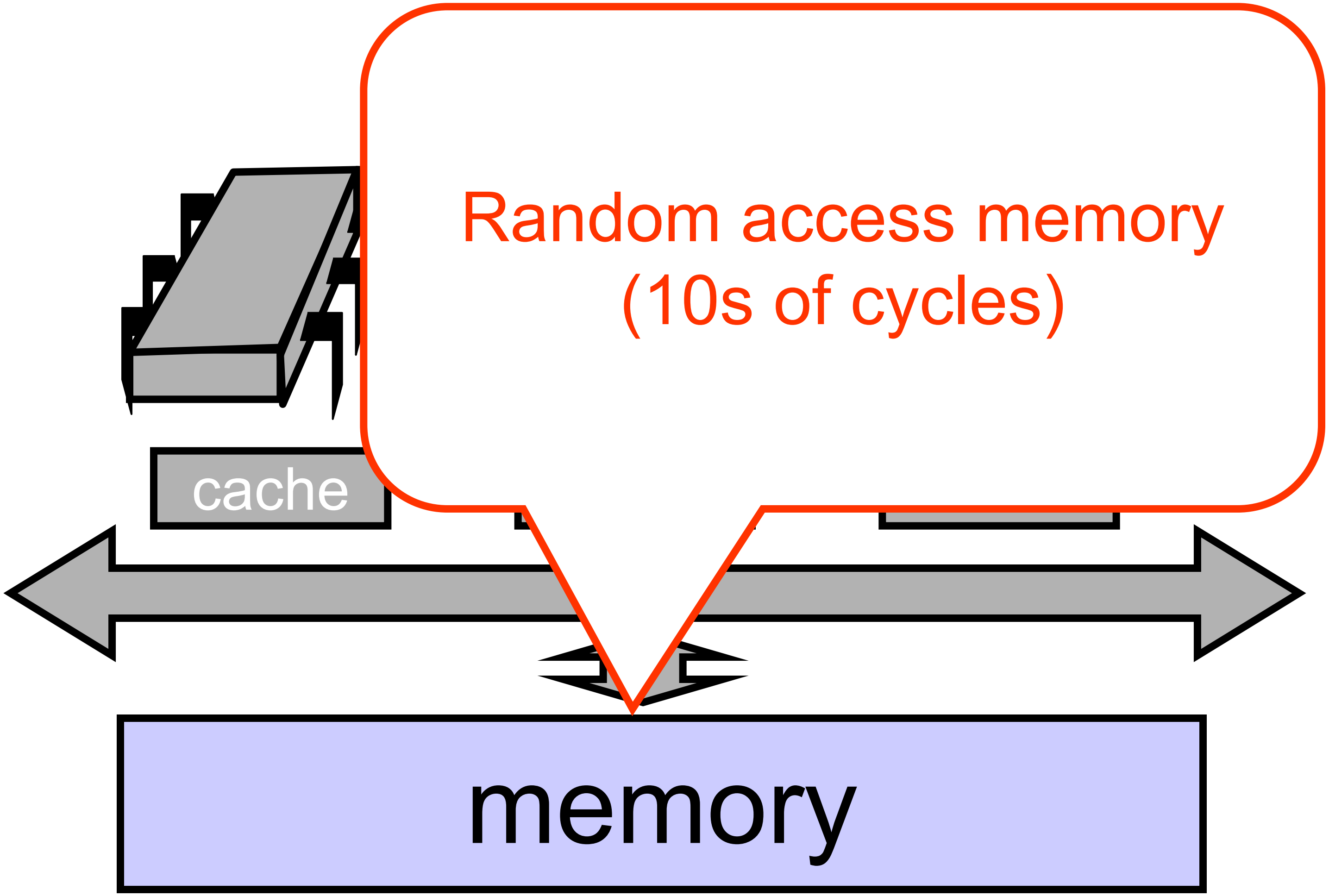
Opinion

- Our memory abstraction is broken
- TAS & TTAS methods
  - Are provably the same (in our model)
  - Except they aren’t (in field tests)
- Need a more detailed model …

Bus-Based Architectures



Bus-Based Architectures

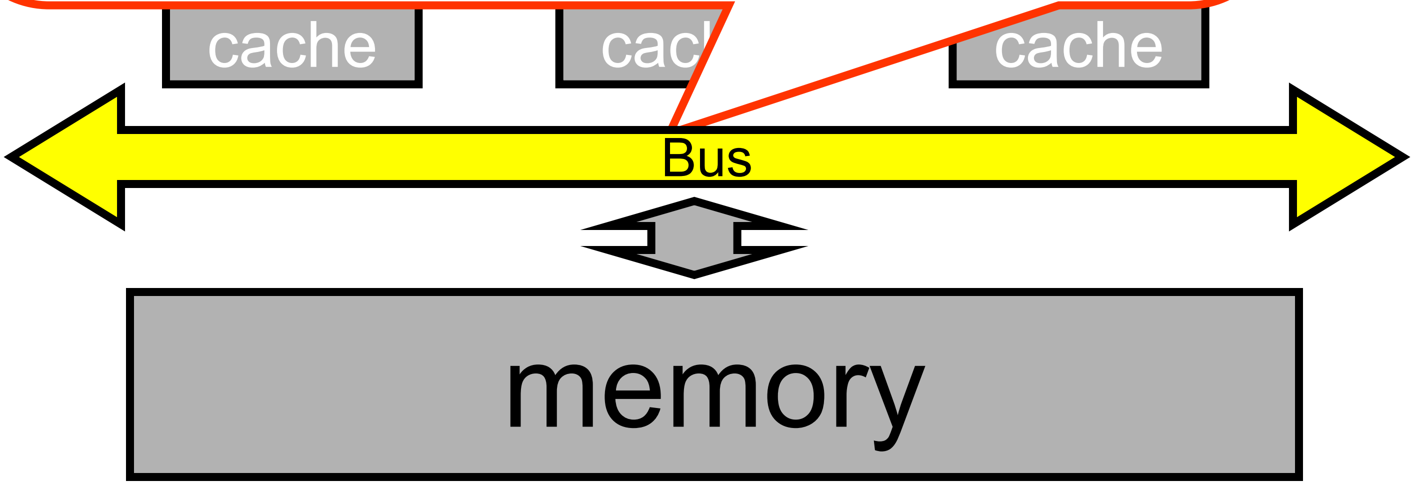


Random access memory
(10s of cycles)

Bus-Based Architectures

Shared Bus

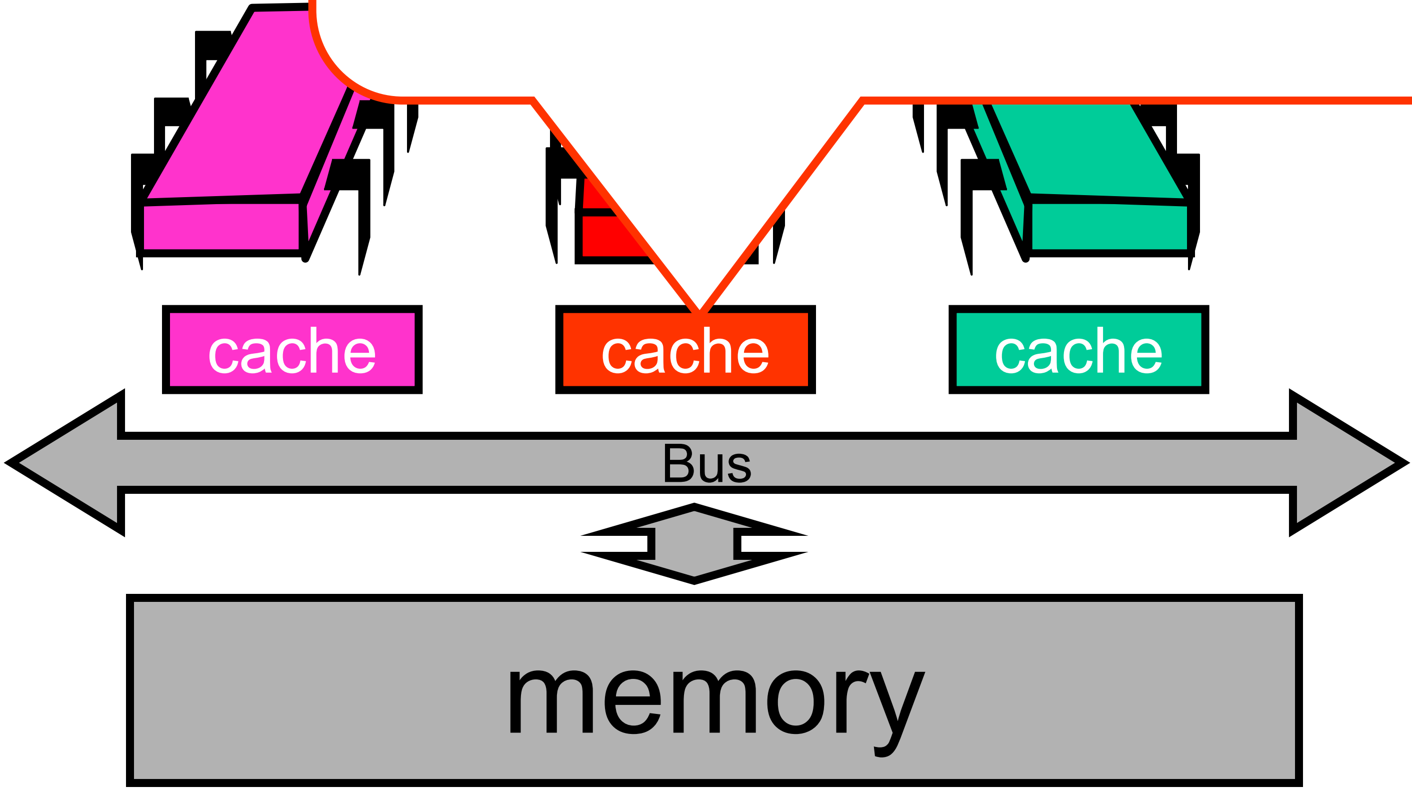
- Broadcast medium
- One broadcaster at a time
- Processors and memory all "snoop"



Bus-

Per-Processor Caches

- Small
- Fast: 1 or 2 cycles
- Address & state information



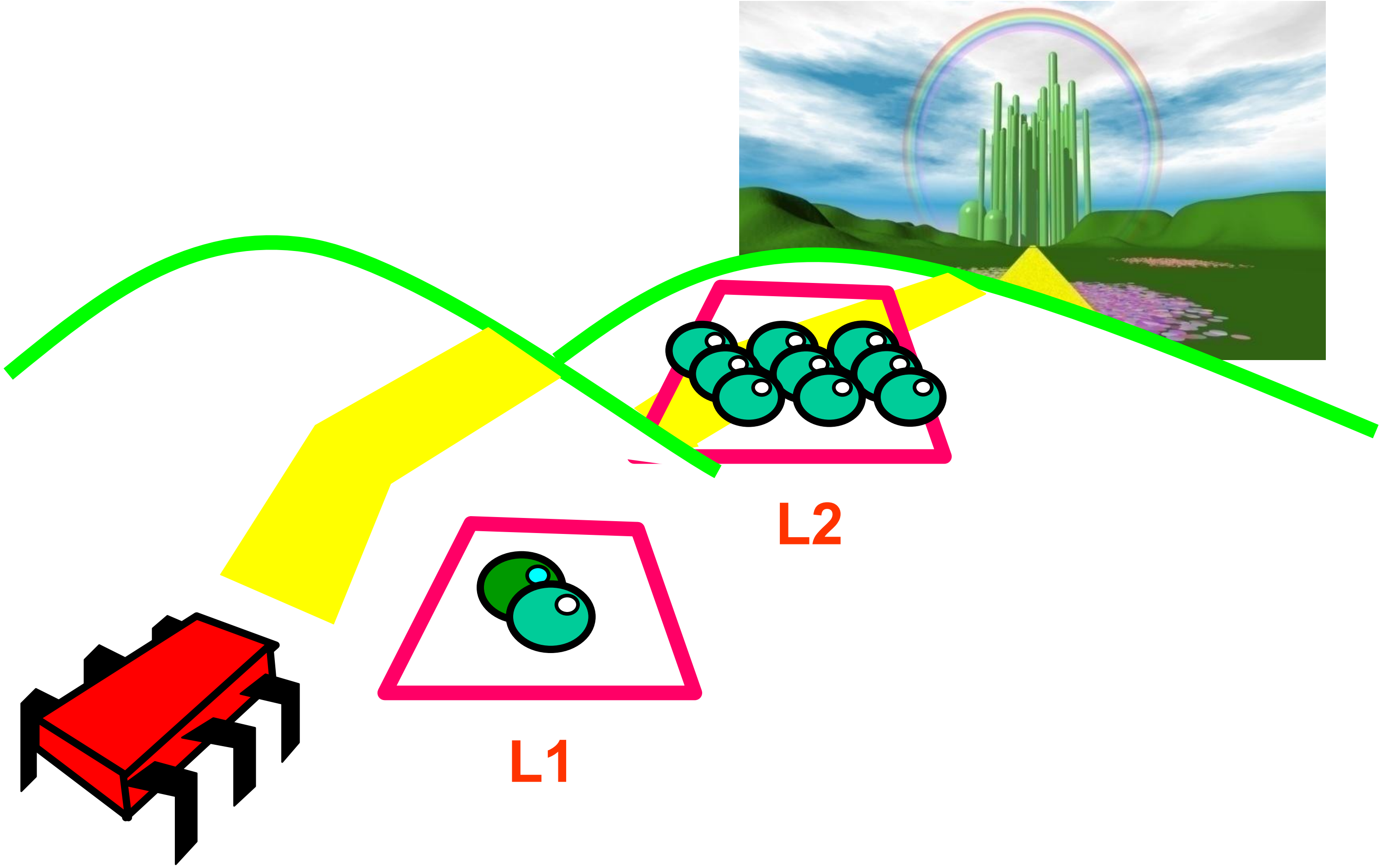
Granularity

- Caches operate at a larger granularity than a word
- Cache line: fixed-size block containing the address (today 64 or 128 bytes)

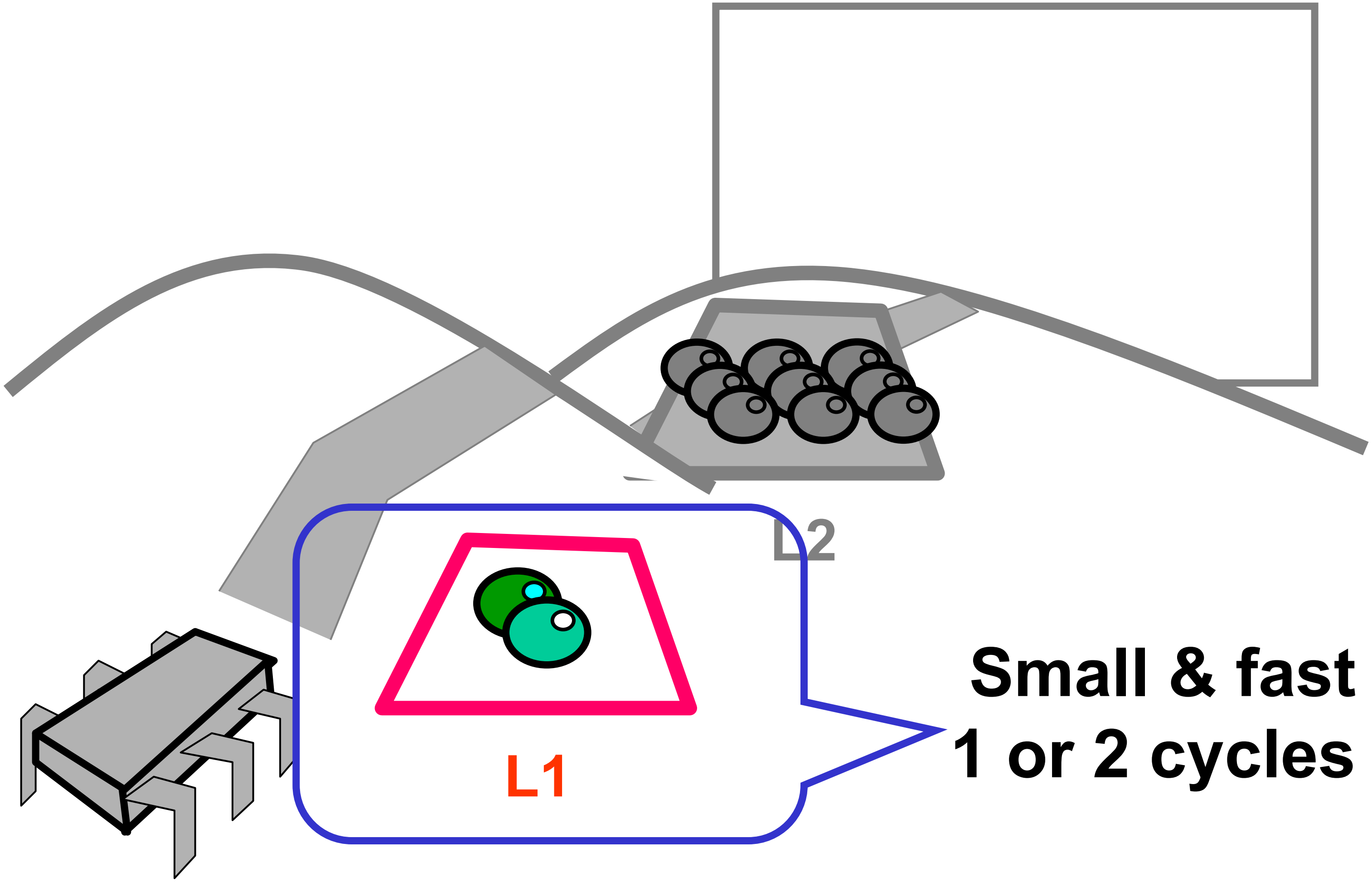
Locality

- If you use an address now, you will probably use it again soon
  - Fetch from cache, not memory
- If you use an address now, you will probably use a nearby address soon
  - In the same cache line

L1 and L2 Caches



L1 and L2 Caches



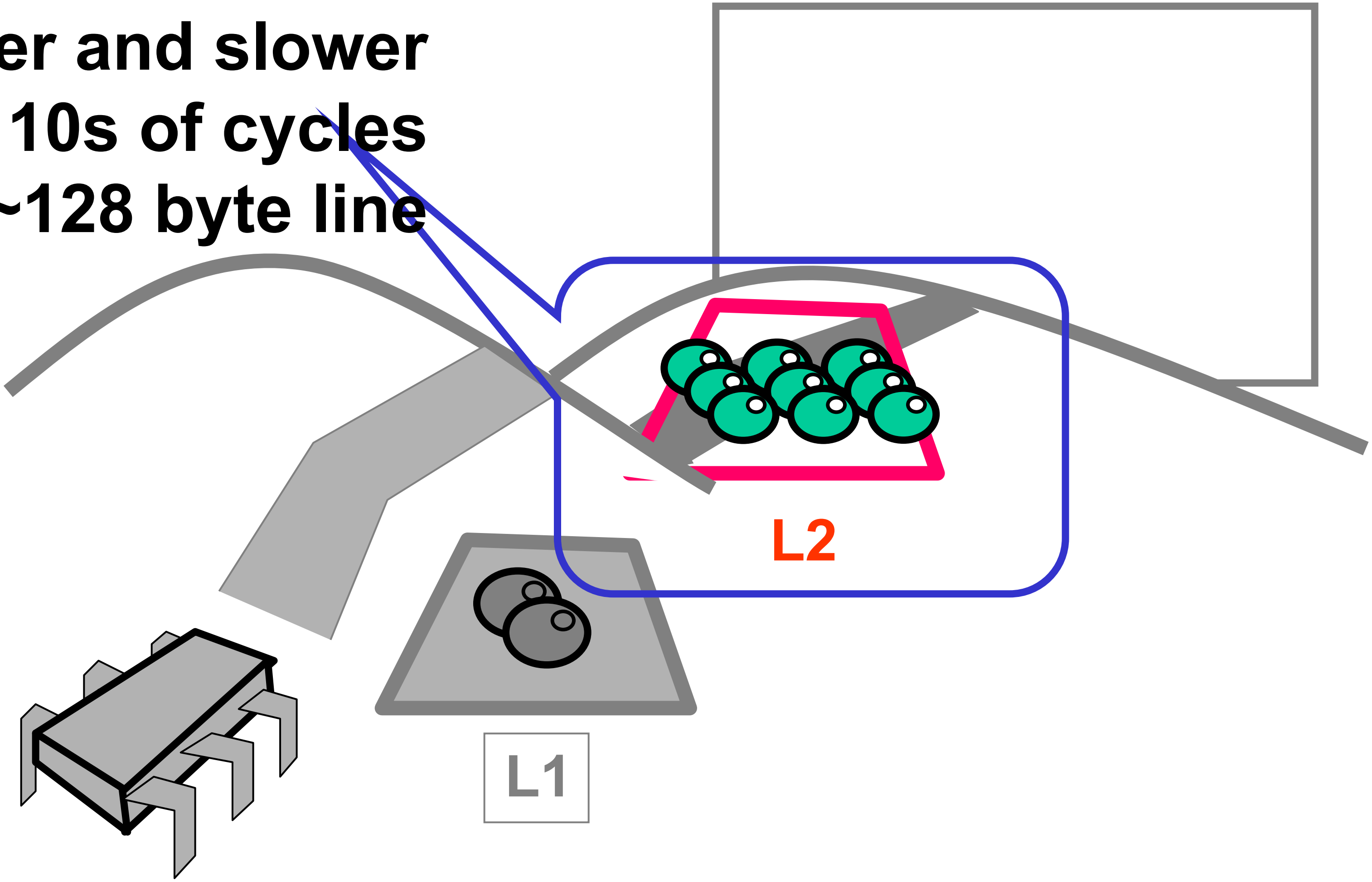
Small & fast
1 or 2 cycles

L1 and L2 Caches

Larger and slower

10s of cycles

~128 byte line



Jargon Watch

- Cache hit
  - “I found what I wanted in my cache”
  - Good Thing™

Jargon Watch

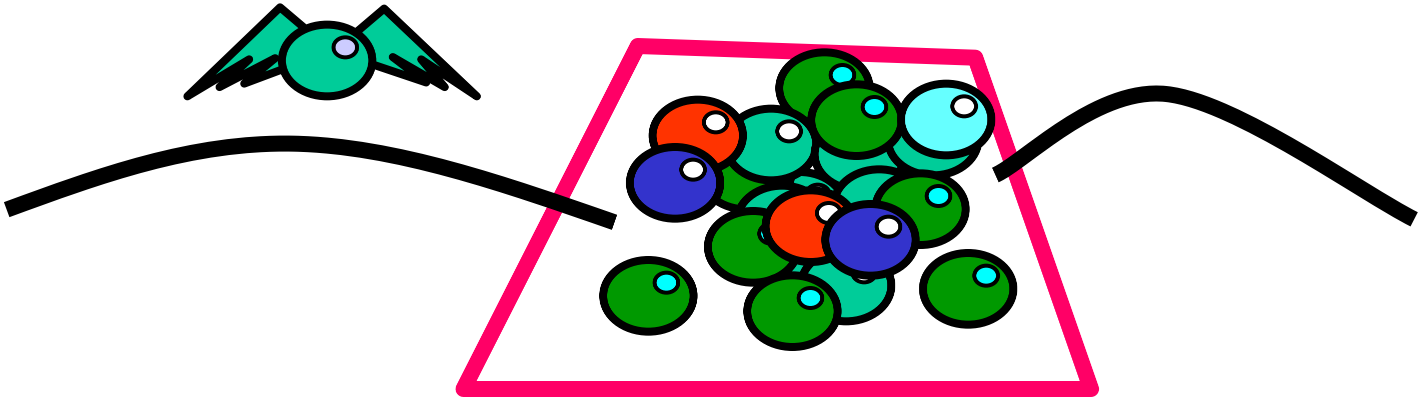
- Cache hit
  - “I found what I wanted in my cache”
  - Good Thing™
- Cache miss
  - “I had to shlep all the way to memory for that data”
  - Bad Thing™

Cave Canem

- This model is still a simplification
  - But not in any essential way
  - Illustrates basic principles
- Will discuss complexities later

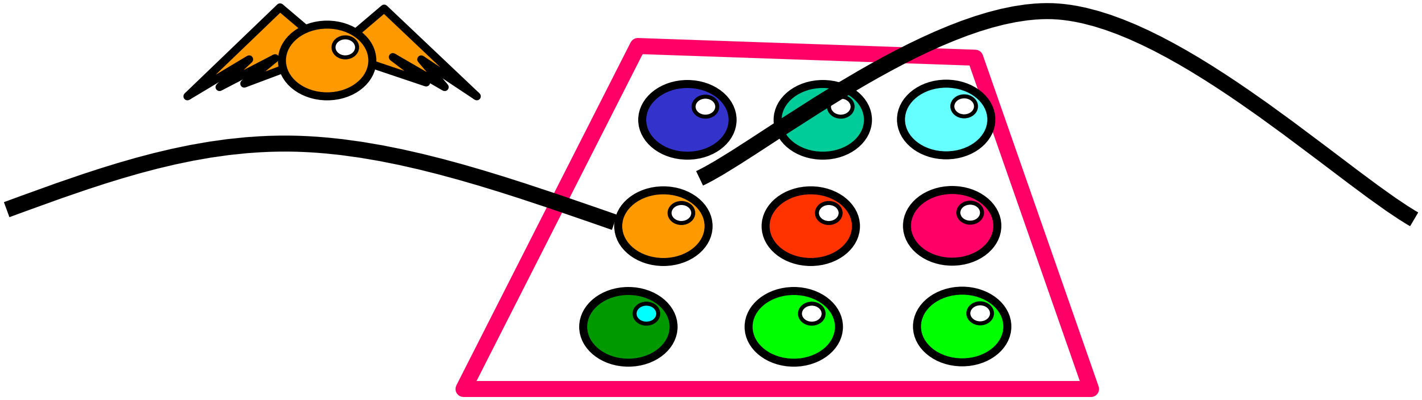
Fully Associative Cache

- Any line can be anywhere in the cache
  - Advantage: can replace any line
  - Disadvantage: hard to find lines



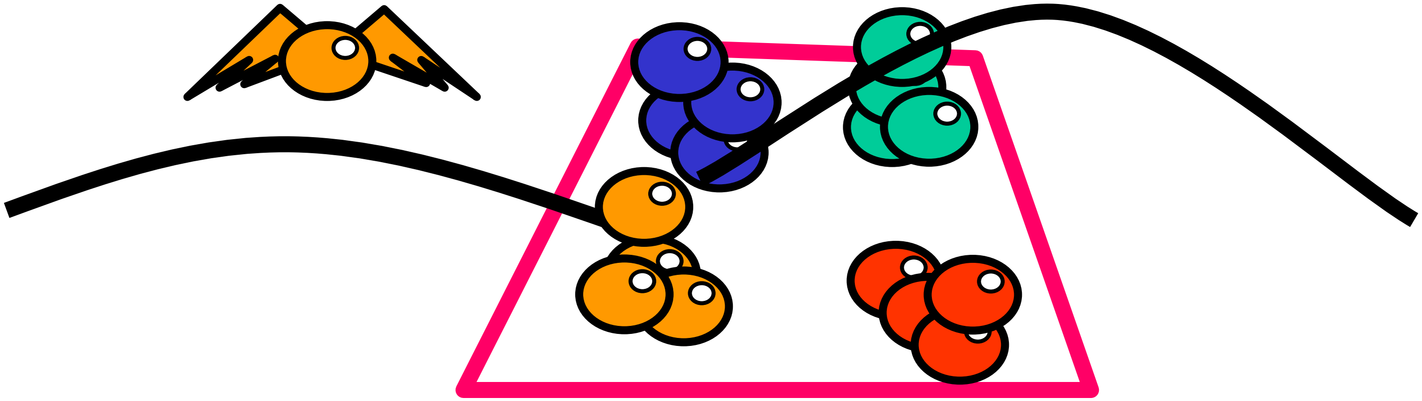
Direct Mapped Cache

- Every address has exactly 1 slot
  - – Advantage: easy to find a line
  - – Disadvantage: must replace fixed line



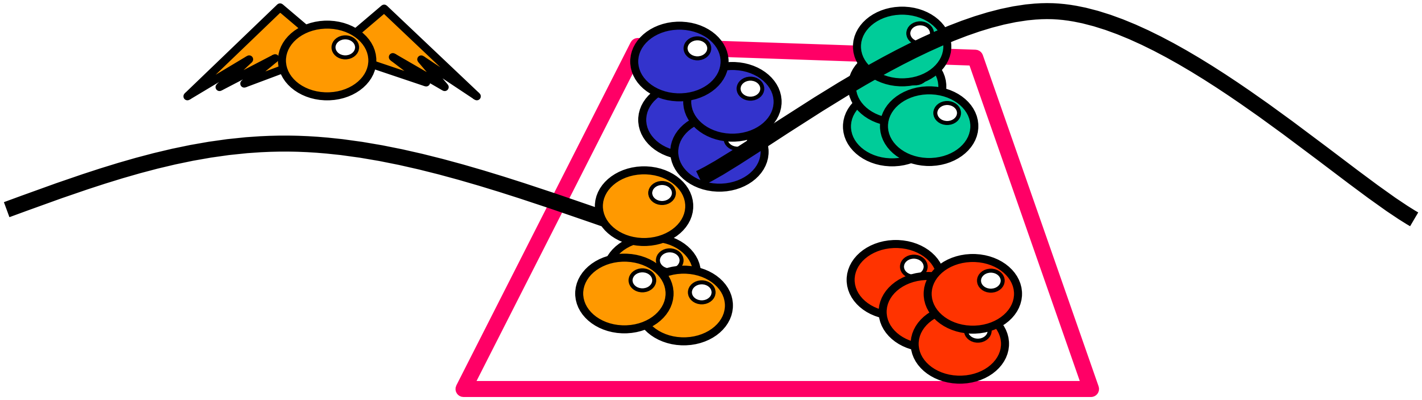
K-way Set Associative Cache

- Each slot holds k lines
  - Advantage: pretty easy to find a line
  - Advantage: some choice in replacing line



Multicore Set Associativity

- k is 8 or even 16 and growing…
  - Why? Because cores share sets
  - Threads cut effective size if accessing different data



Cache Coherence

- A and B both cache address x
- A writes to x
  - Updates cache
- How does B find out?
- Many cache coherence protocols in literature

MESI

- Modified
  - Have modified cached data, must write back to memory

MESI

- Modified
  - Have modified cached data, must write back to memory
- Exclusive
  - Not modified, I have only copy

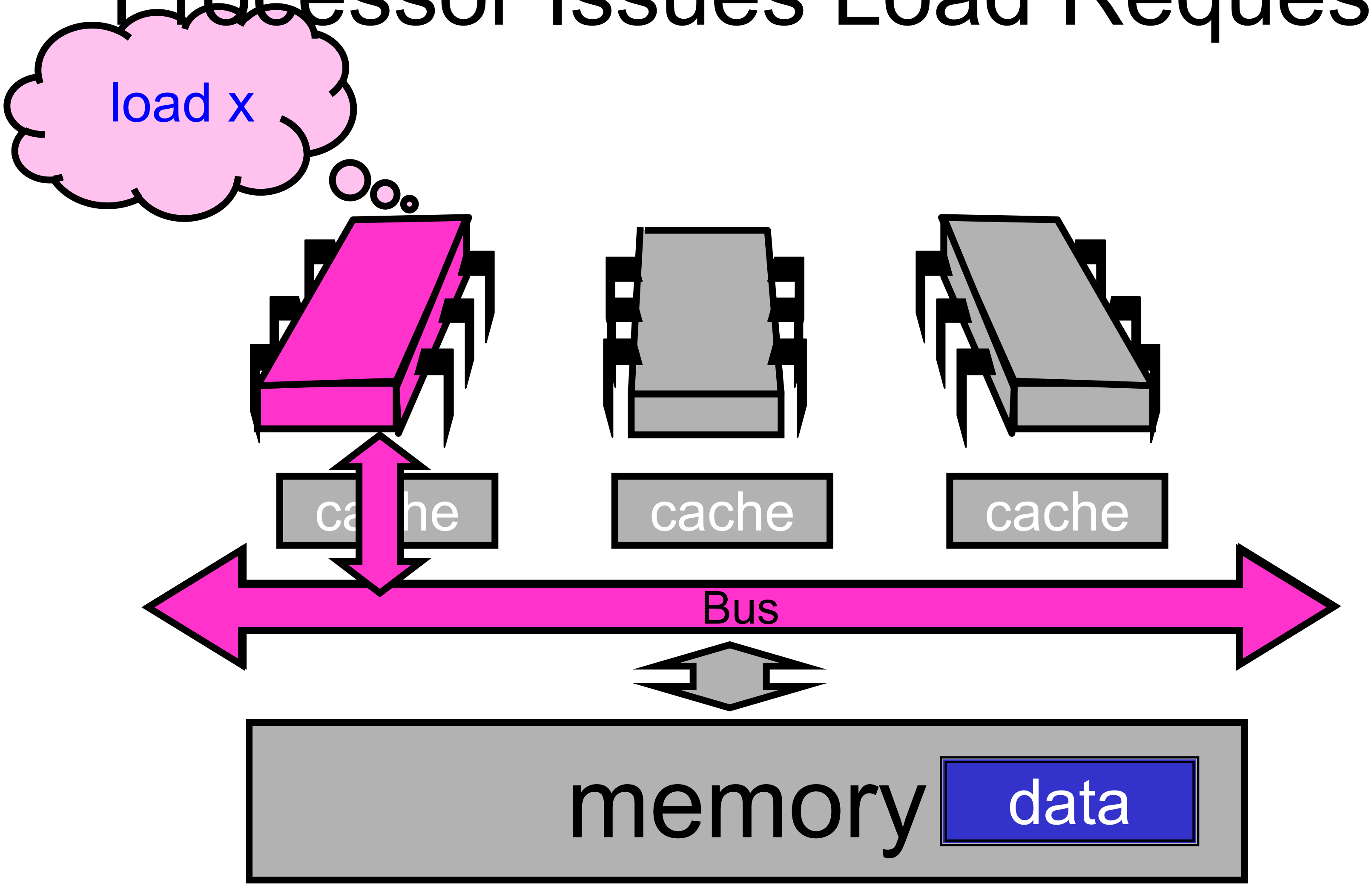
MESI

- Modified
  - Have modified cached data, must write back to memory
- Exclusive
  - Not modified, I have only copy
- Shared
  - Not modified, may be cached elsewhere

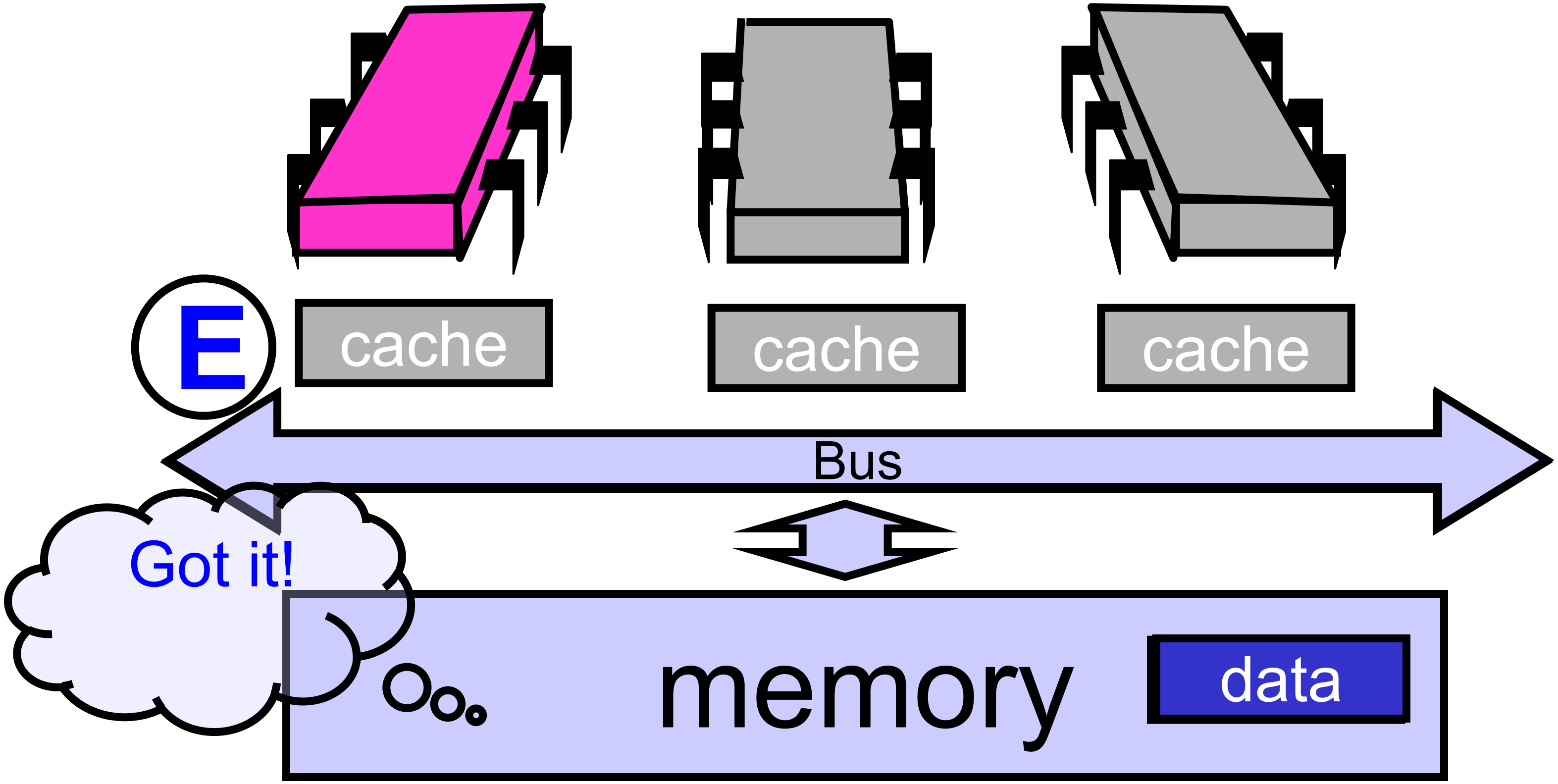
MESI

- Modified
  - Have modified cached data, must write back to memory
- Exclusive
  - Not modified, I have only copy
- Shared
  - Not modified, may be cached elsewhere
- Invalid
  - Cache contents not meaningful

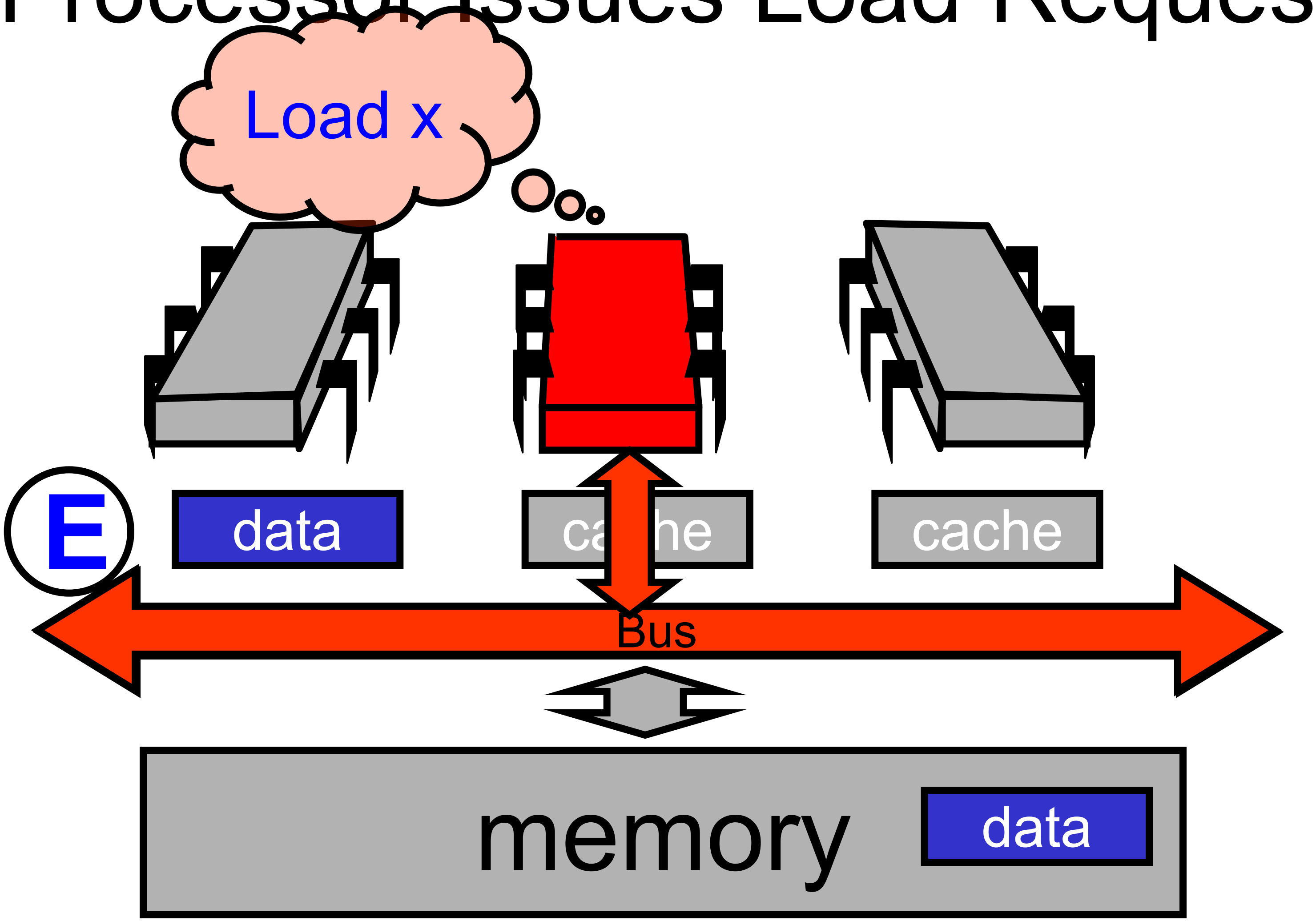
Processor Issues Load Request



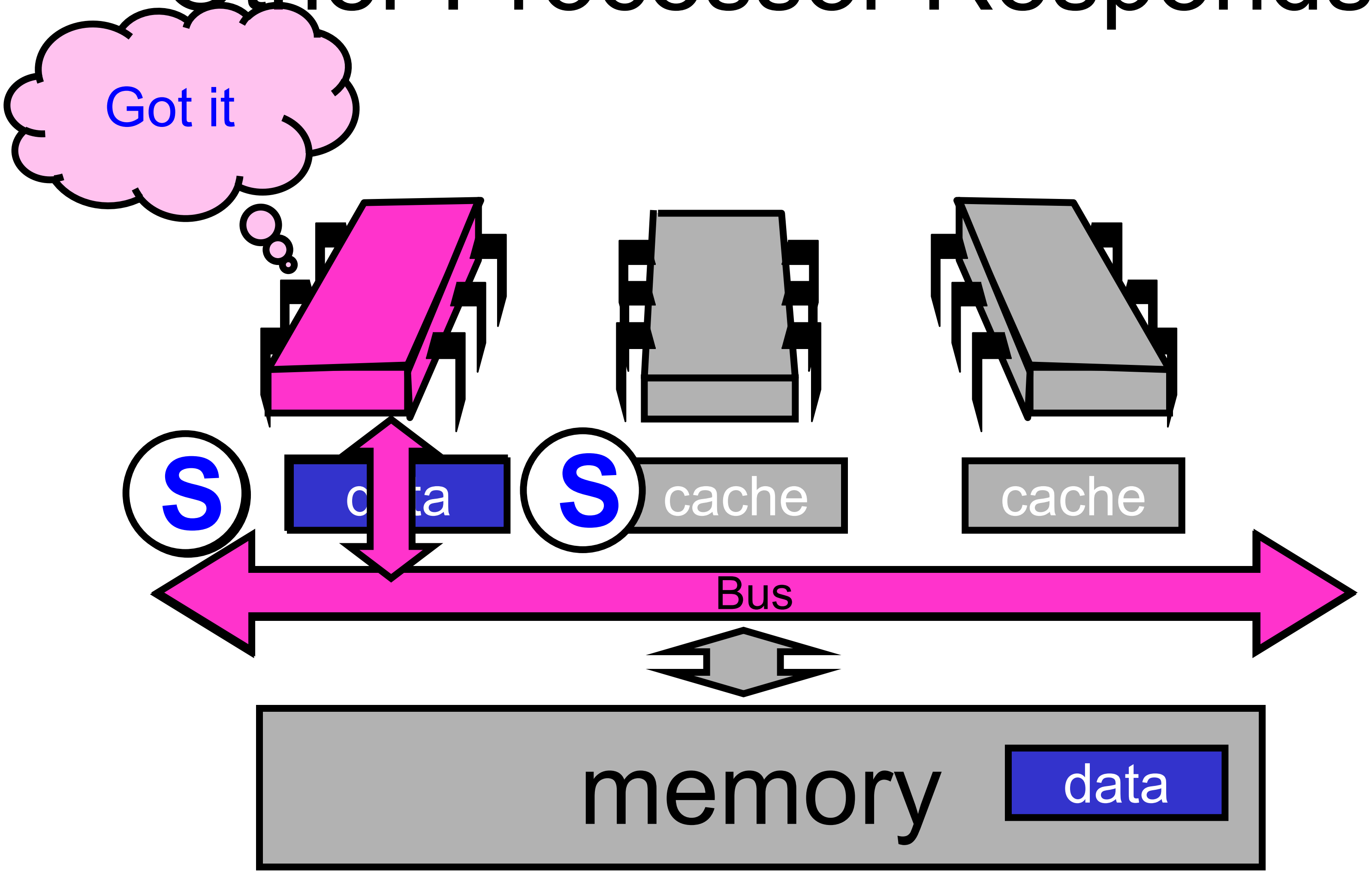
Memory Responds



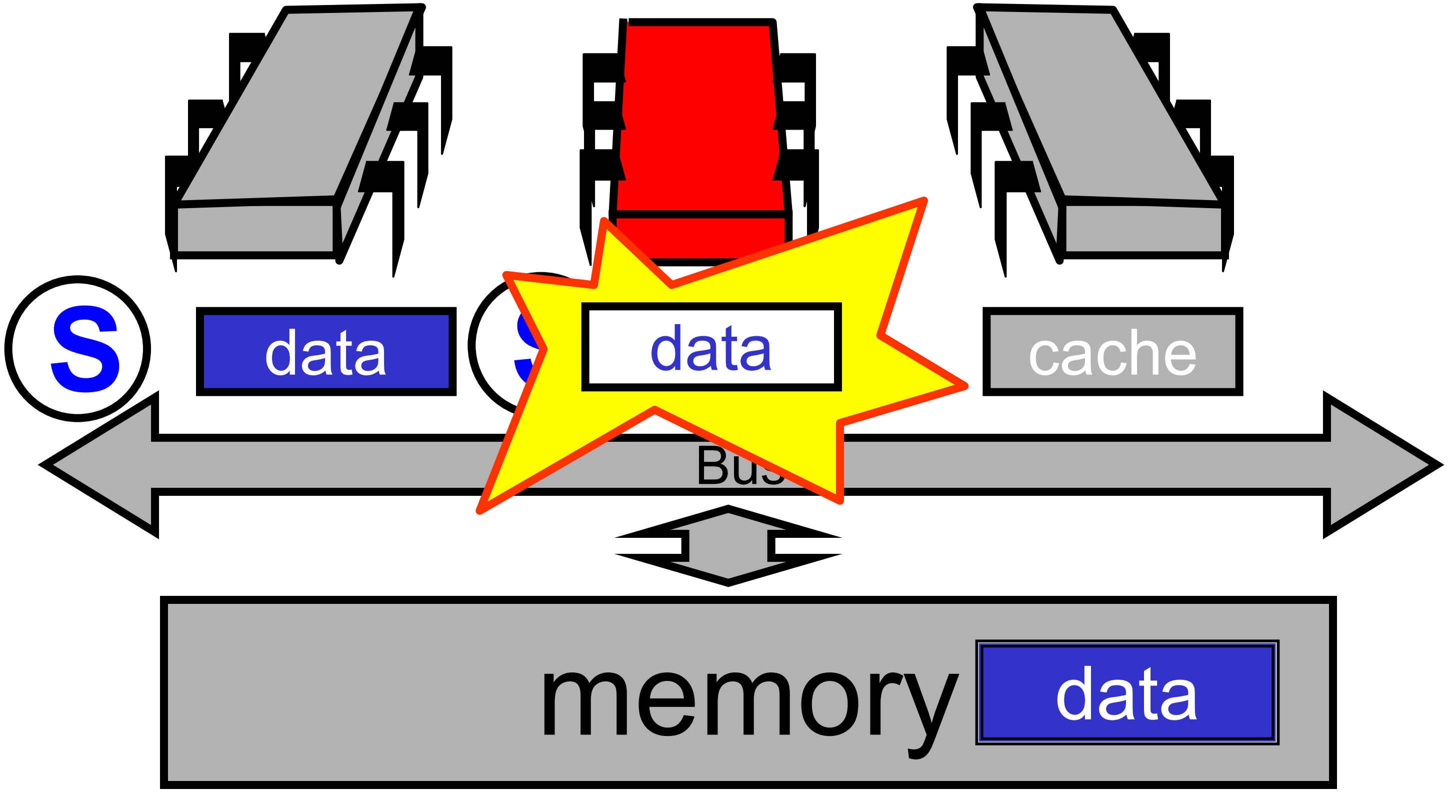
Processor Issues Load Request



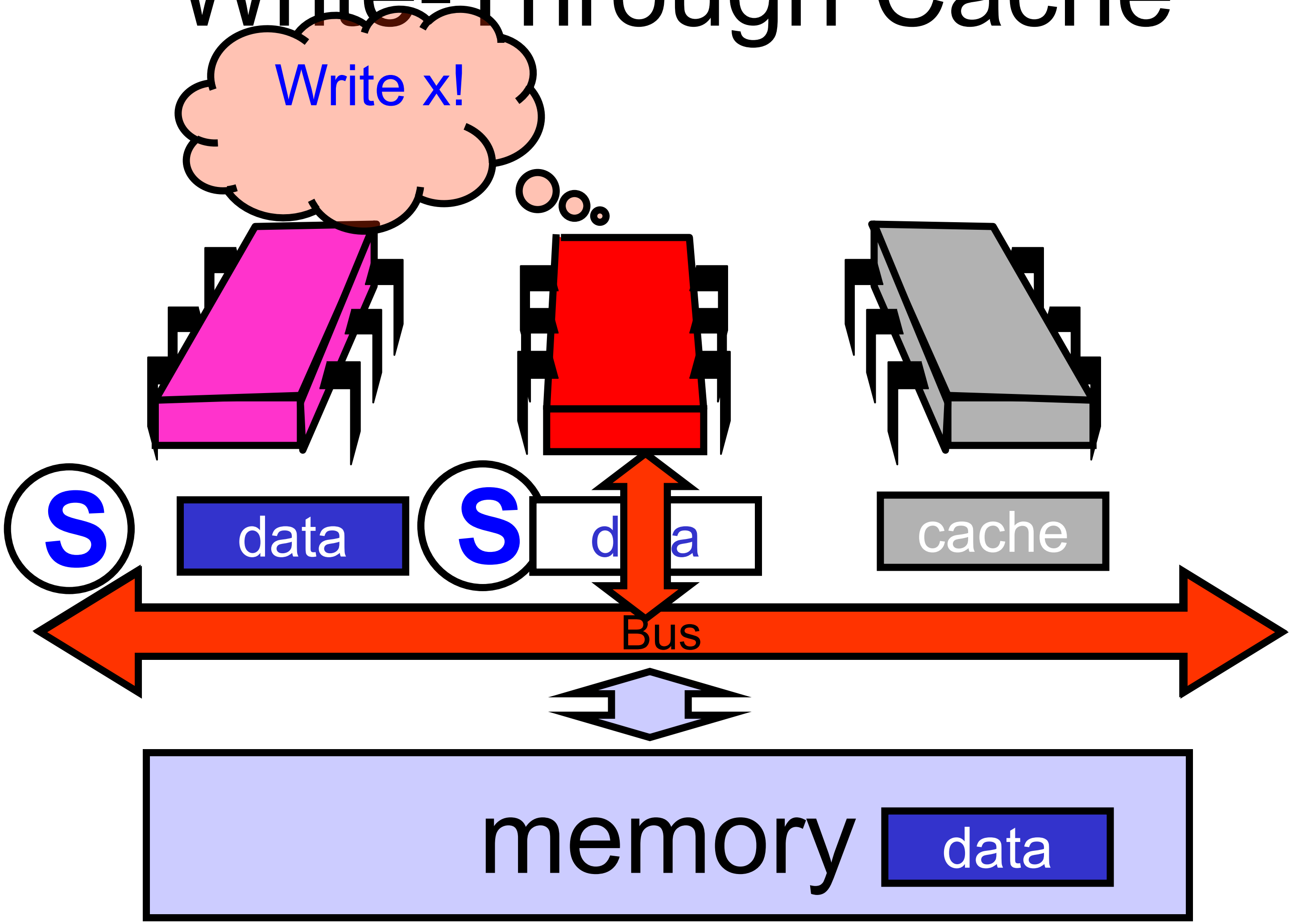
Other Processor Responds



Modify Cached Data



Write-Through Cache



Write-Through Caches

- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes …

Write-Through Caches

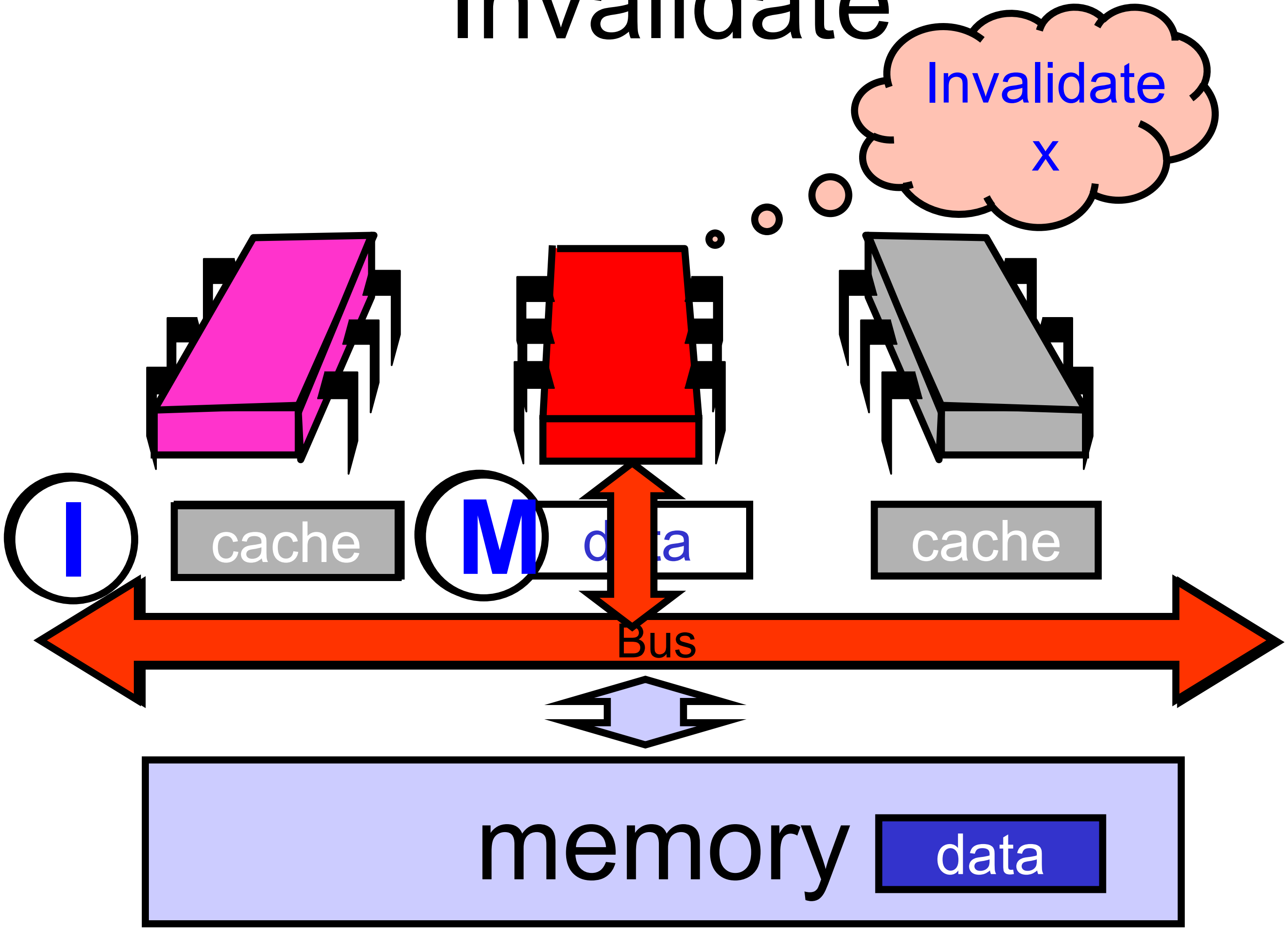
- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes …

“show stoppers”

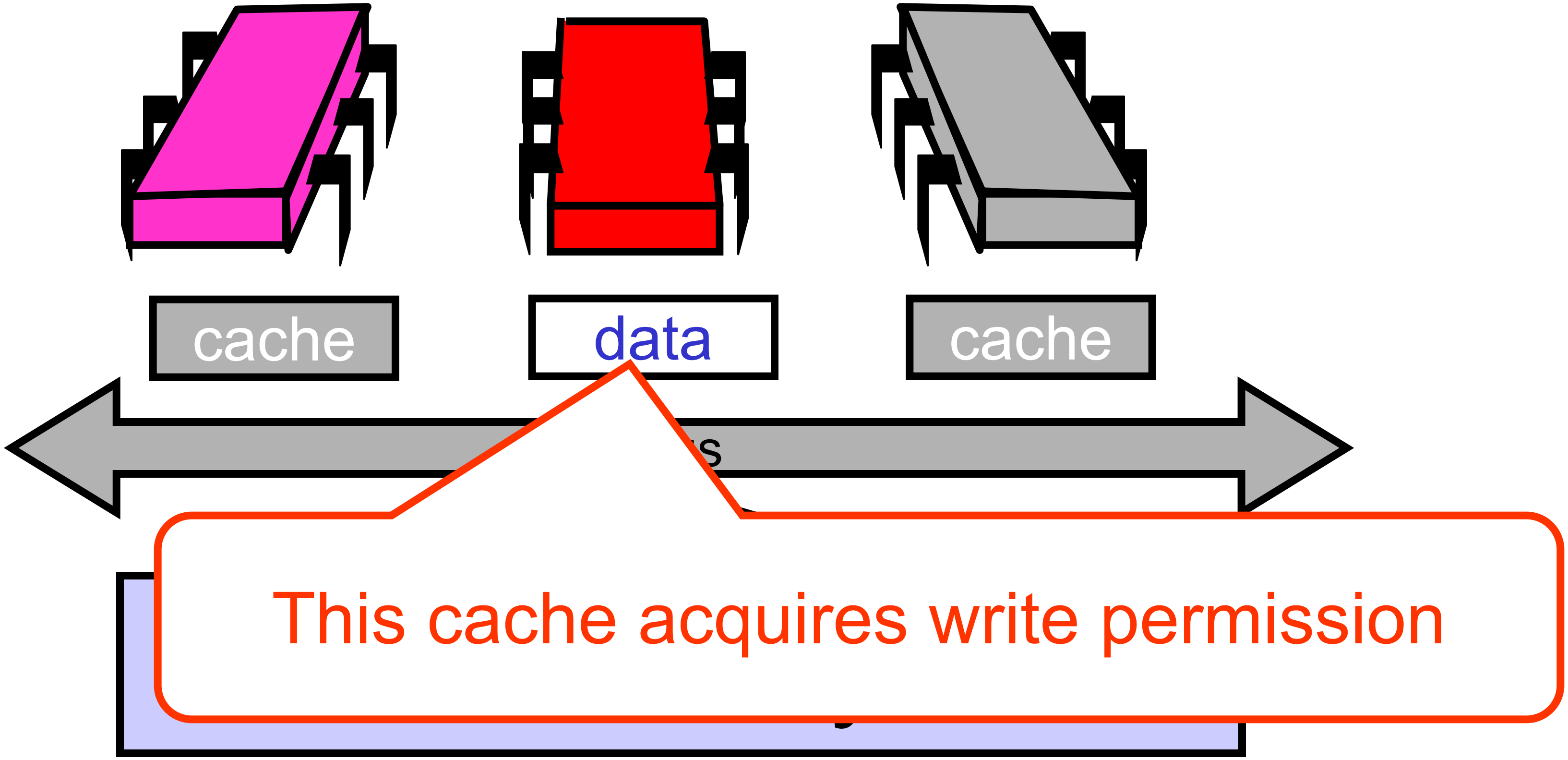
Write-Back Caches

- Accumulate changes in cache
- Write back when line evicted
  - Need the cache for something else
  - Another processor wants it

Invalidate

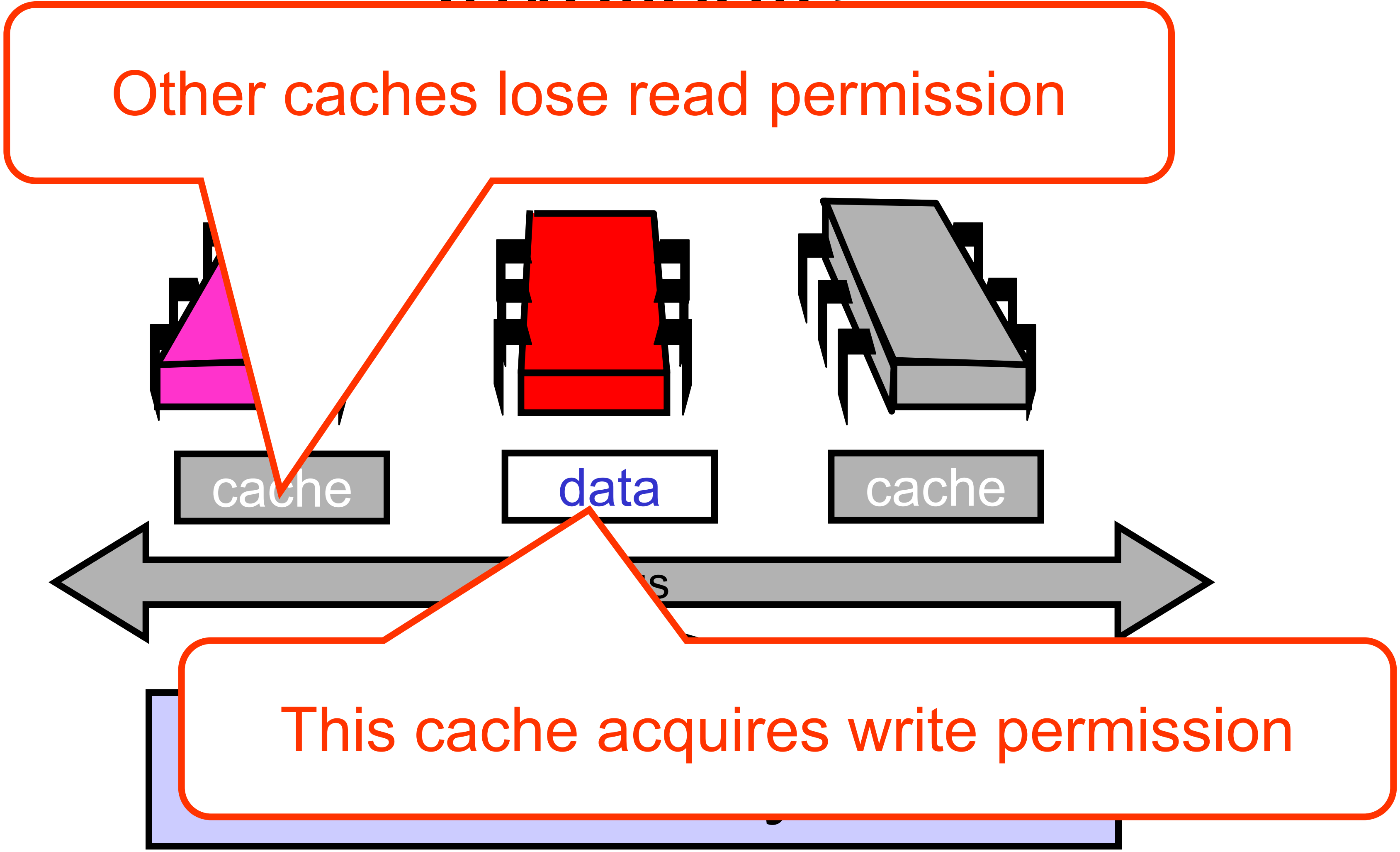


Invalidate



This cache acquires write permission

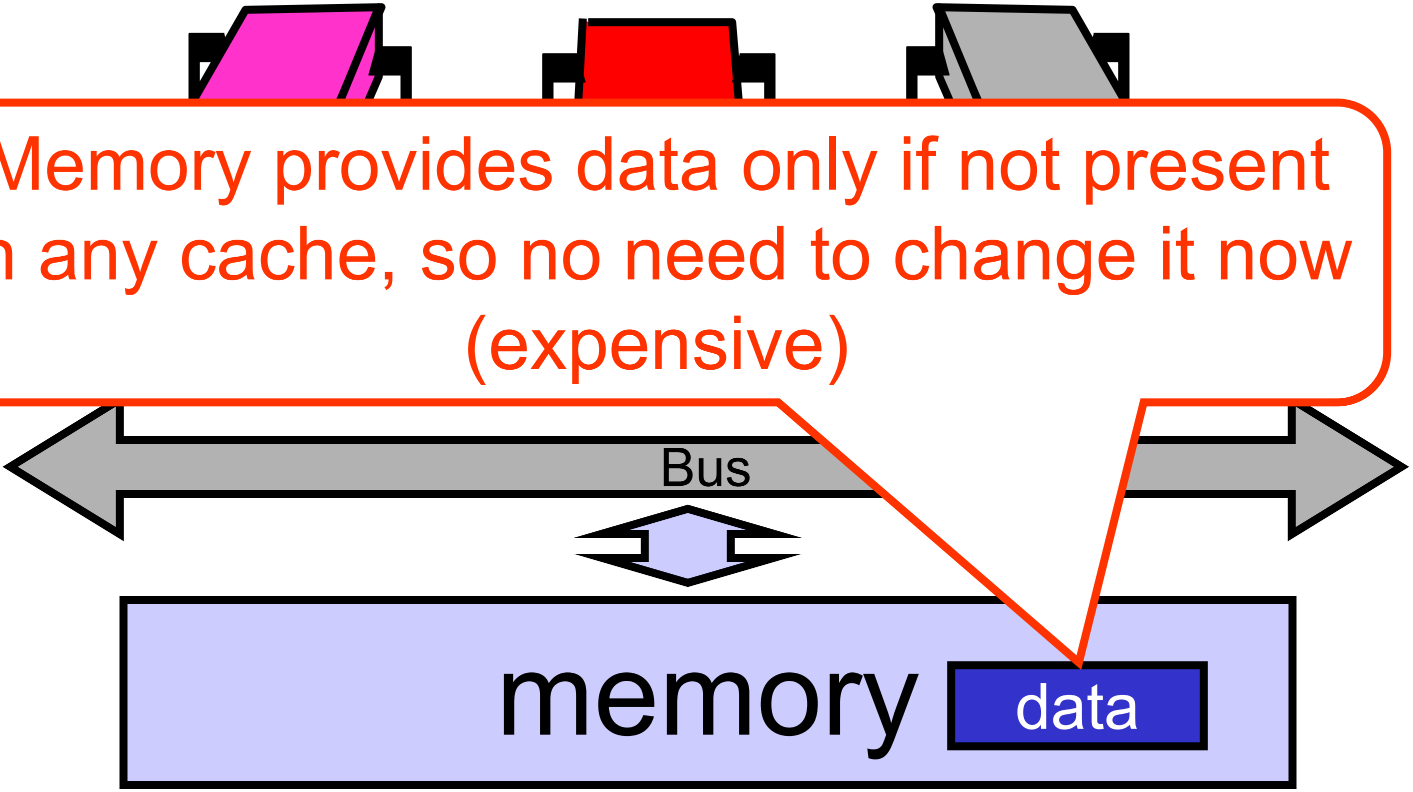
Invalidate



Other caches lose read permission

This cache acquires write permission

Invalidate



Memory provides data only if not present in any cache, so no need to change it now (expensive)

Mutual Exclusion

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

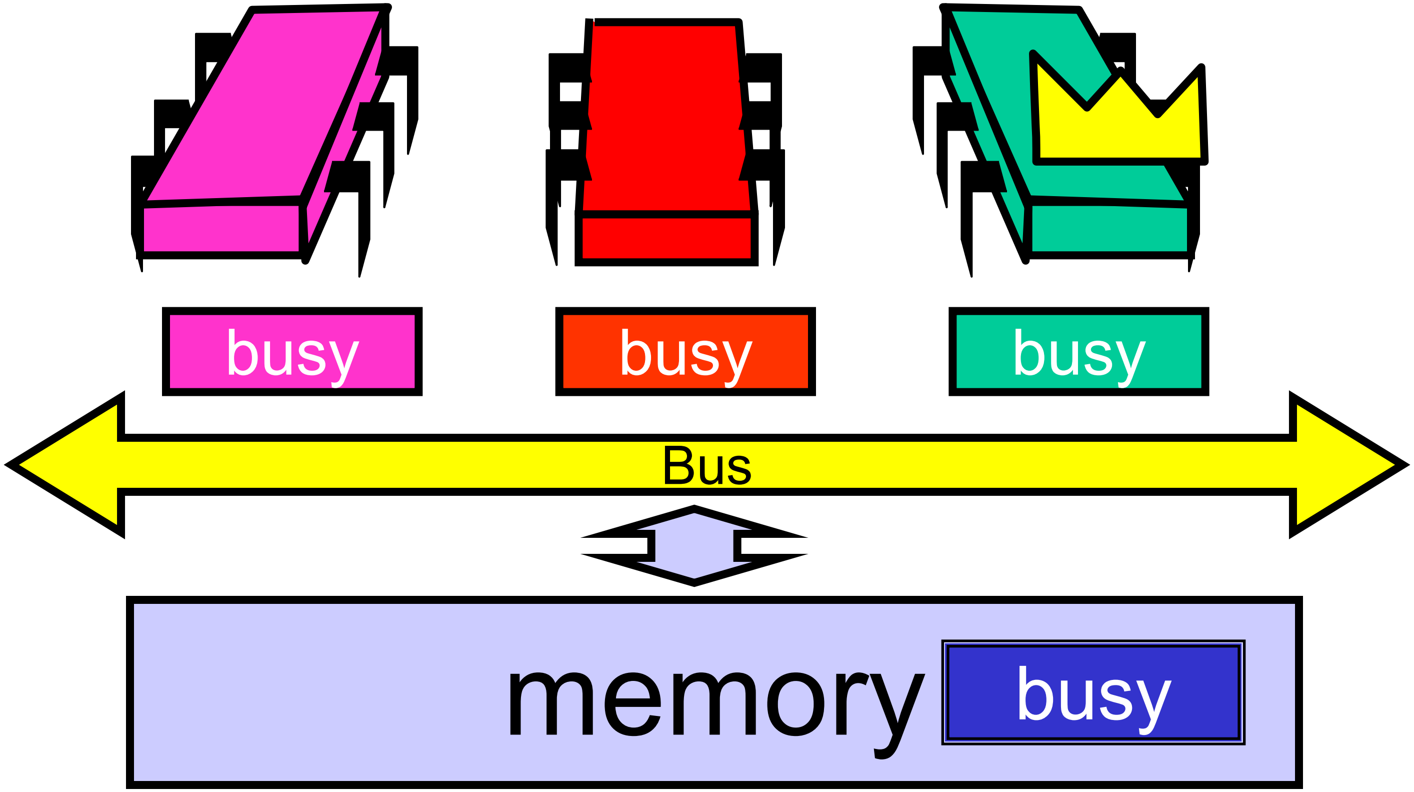
Simple TASLock

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

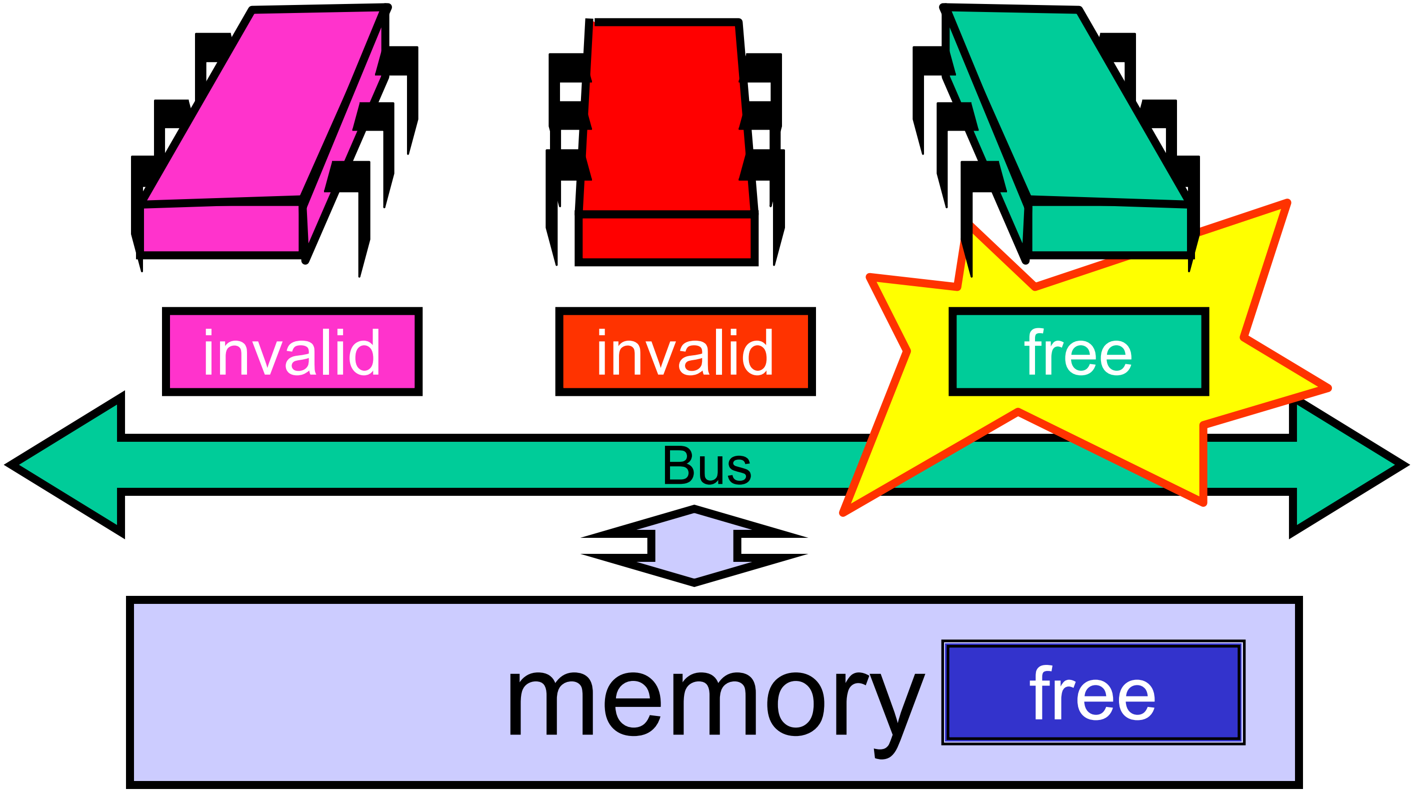
Test-and-test-and-set

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

Local Spinning while Lock is Busy

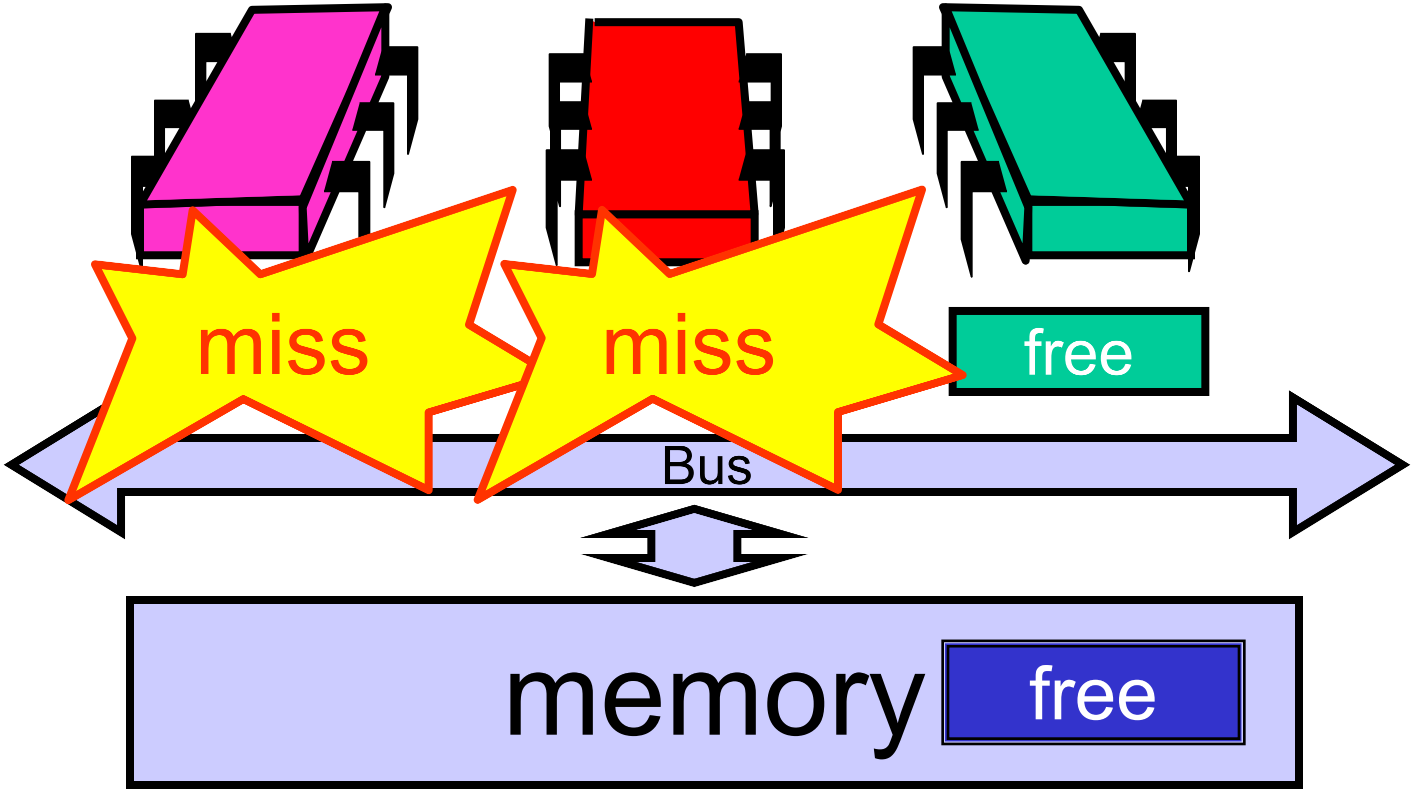


On Release



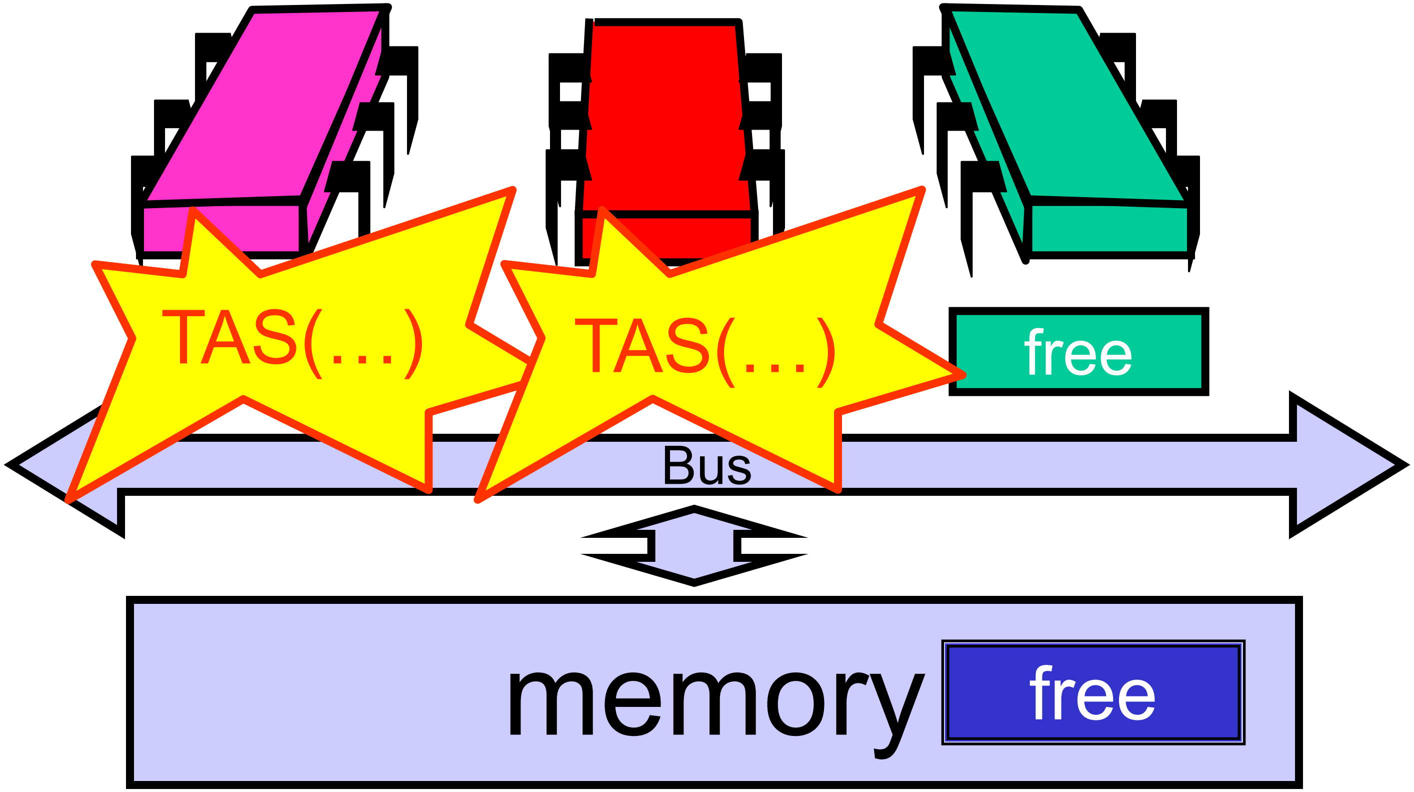
On Release

Everyone misses, rereads



On Release

Everyone tries TAS

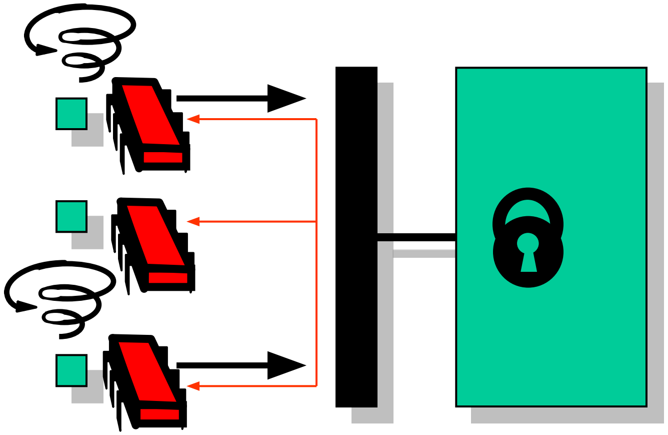


Problems

- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?

Measuring Quiescence Time

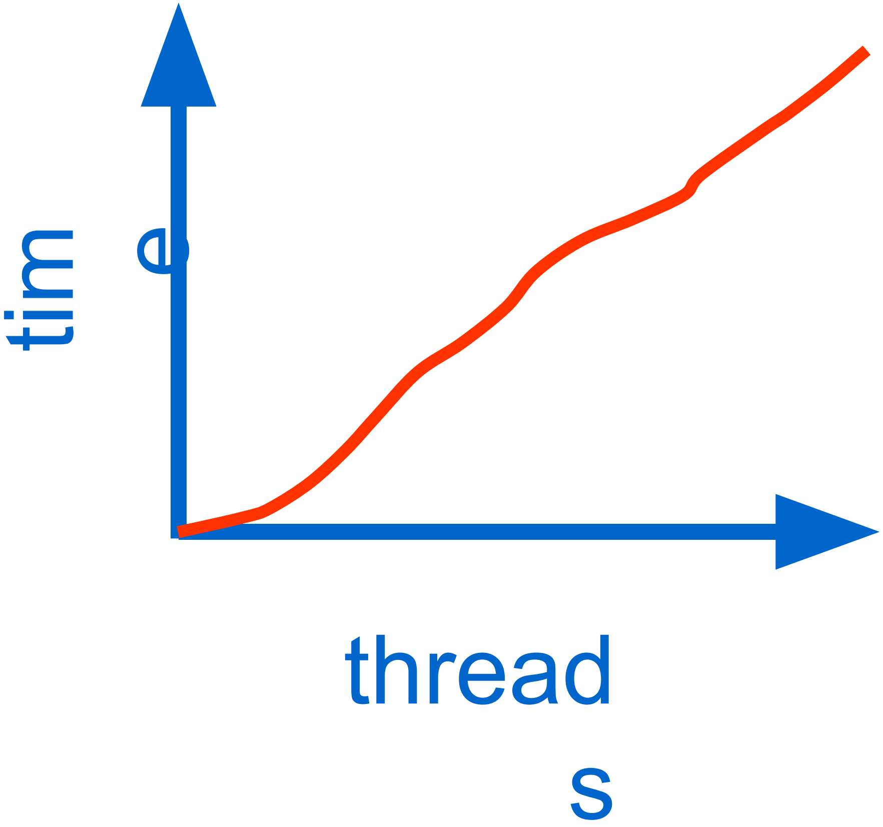
- Acquire lock
- Pause without using bus
- Use bus heavily



If pause > quiescence time,
critical section duration independent of number of threads

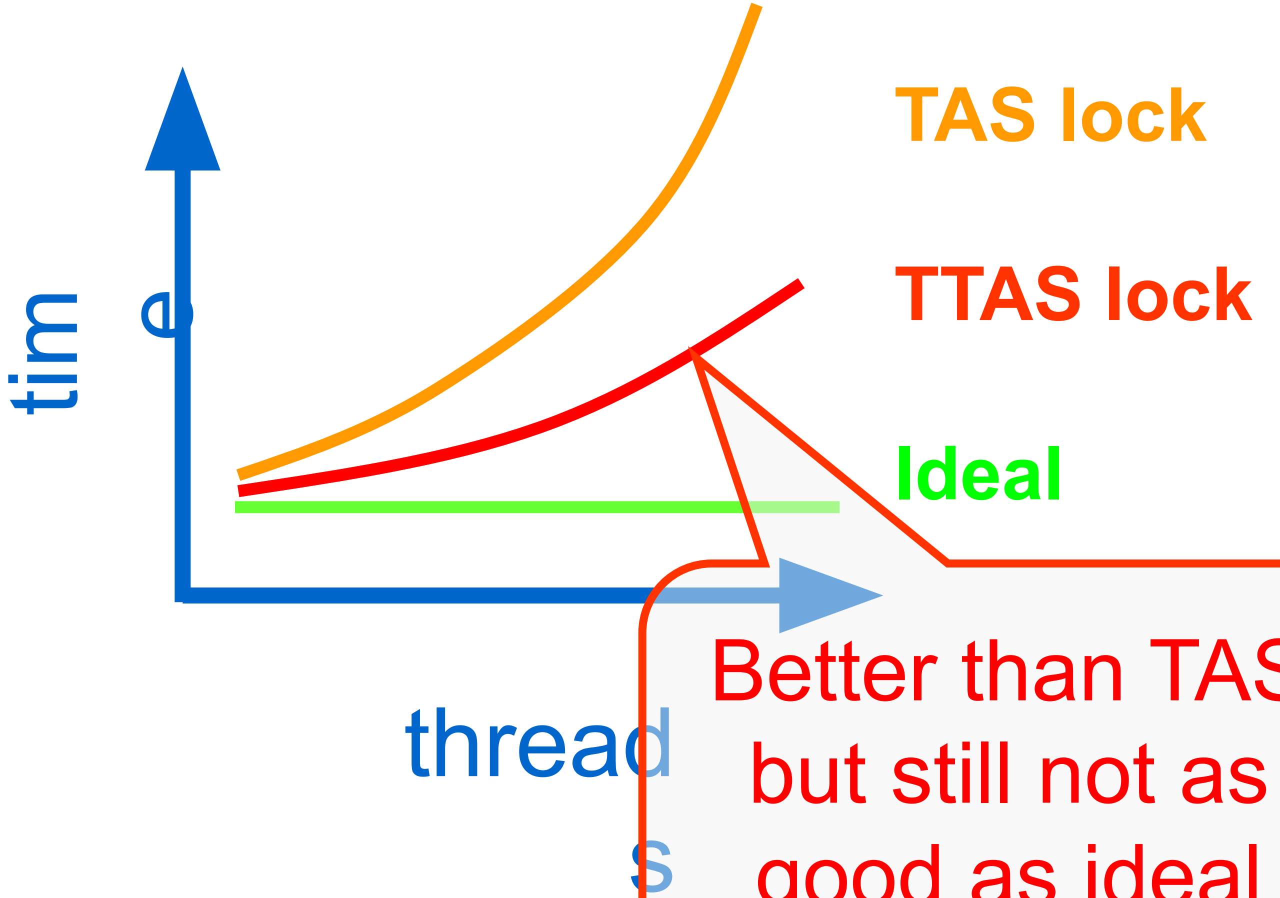
If pause < quiescence time,
critical section duration slower with more threads

Quiescence Time



Increses linearly with the number of processors for bus architecture

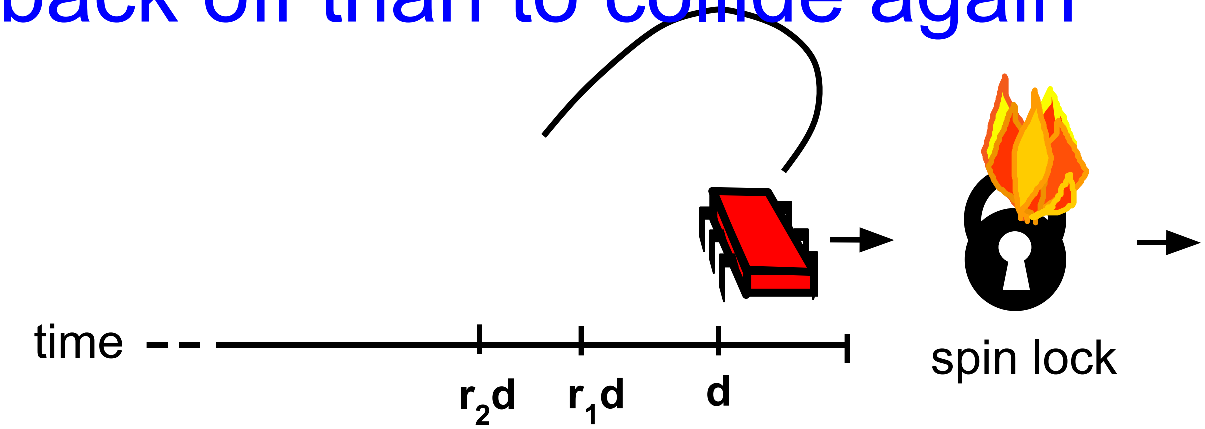
Mystery Explained



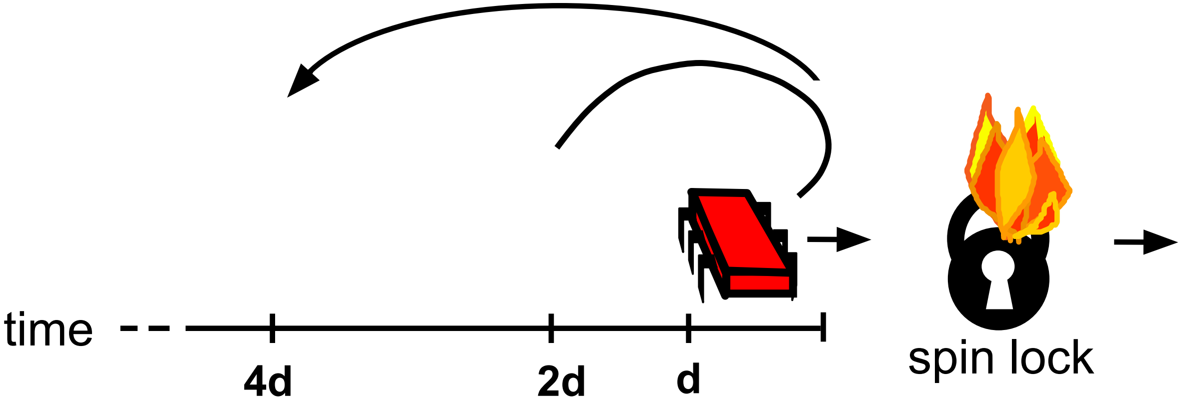
Better than TAS but still not as good as ideal

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 (!lock.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 (!lock.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 (!lock.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 (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}
```

If we win, return

Exponential Backoff Lock

Back off for random duration

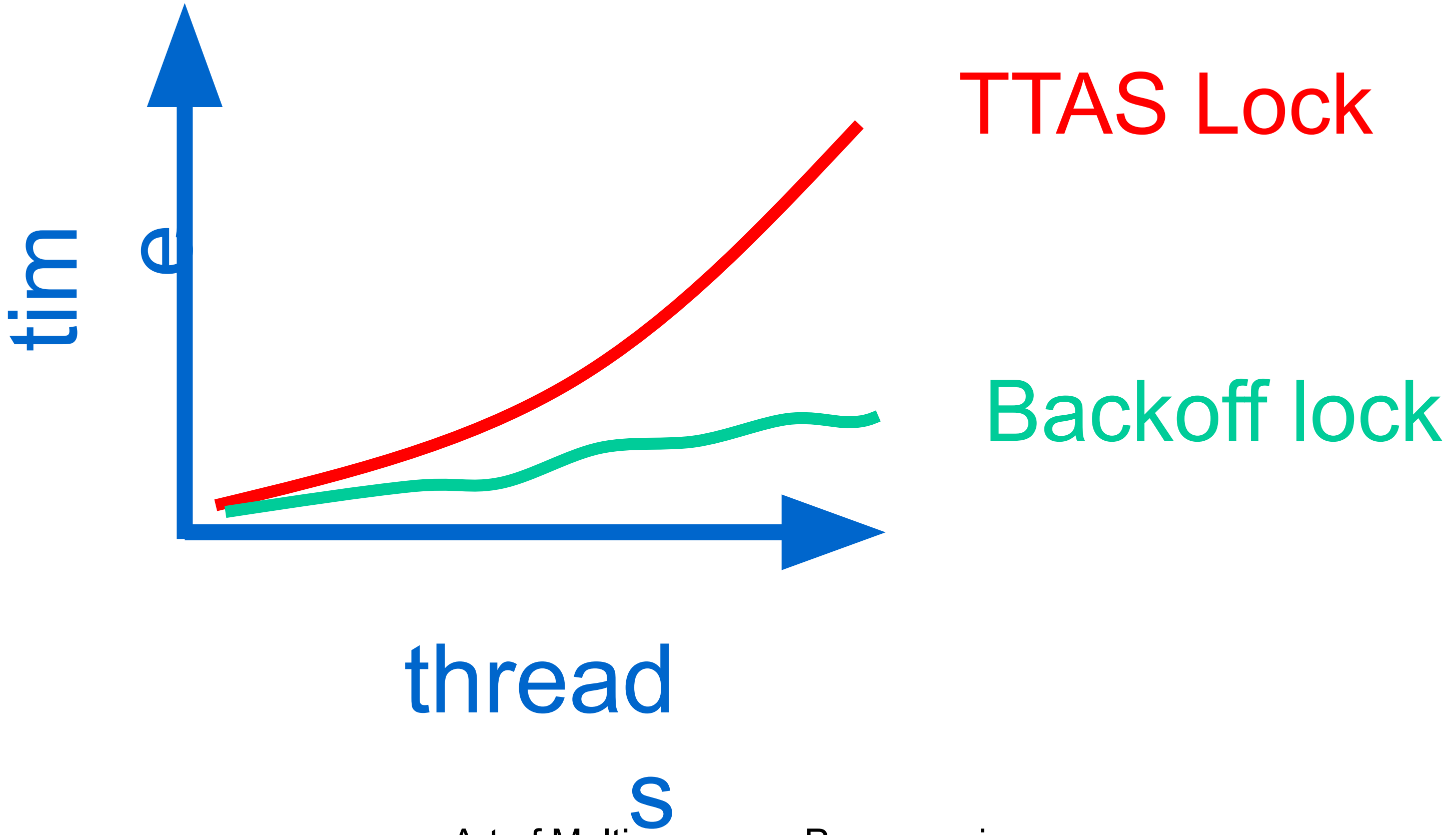
```
public class Backoff implements lock {
 publi
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}
```

Exponential Backoff Lock

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

Double max delay, within reason

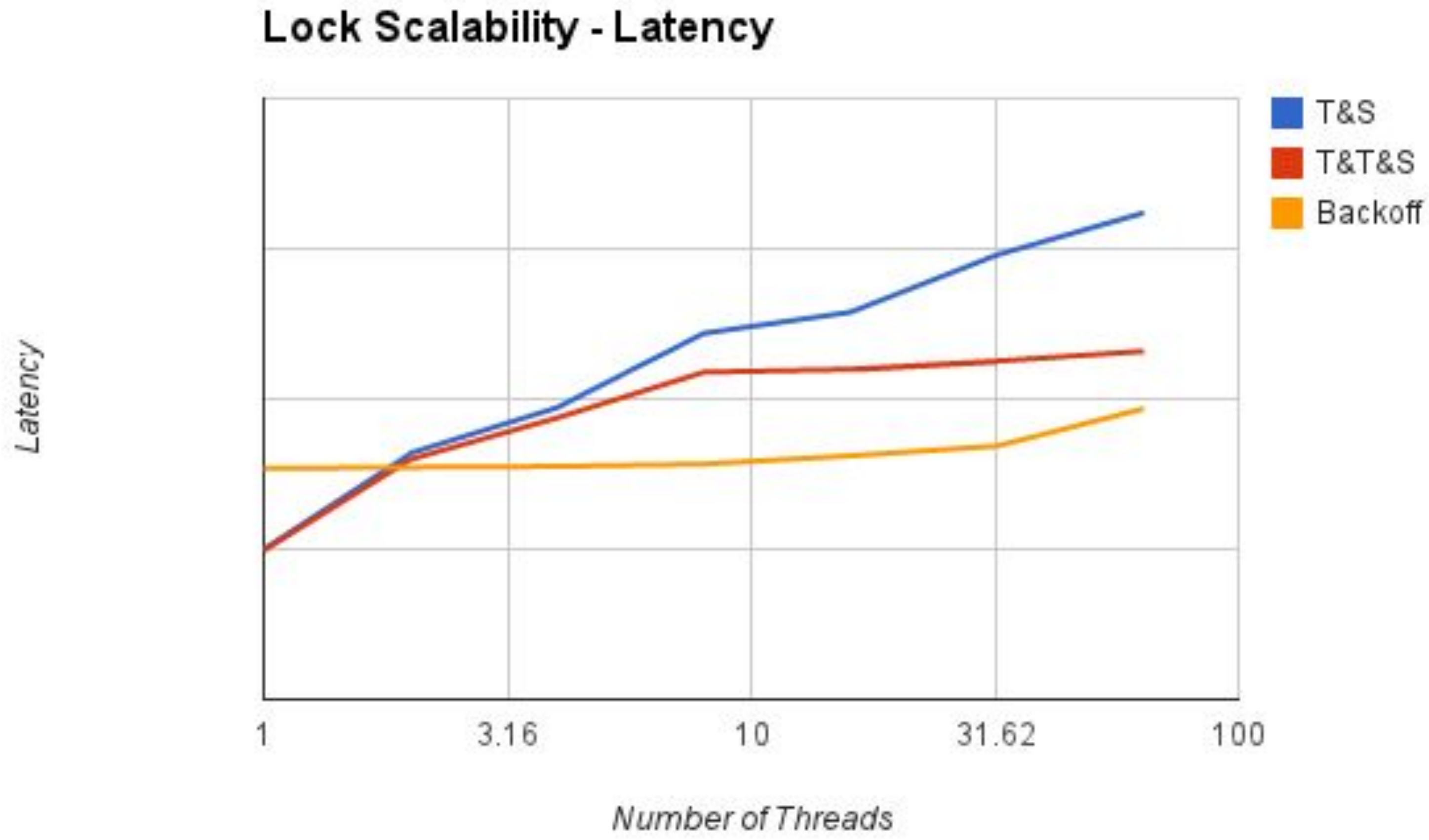
Spin-Waiting Overhead



Backoff: Other Issues

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

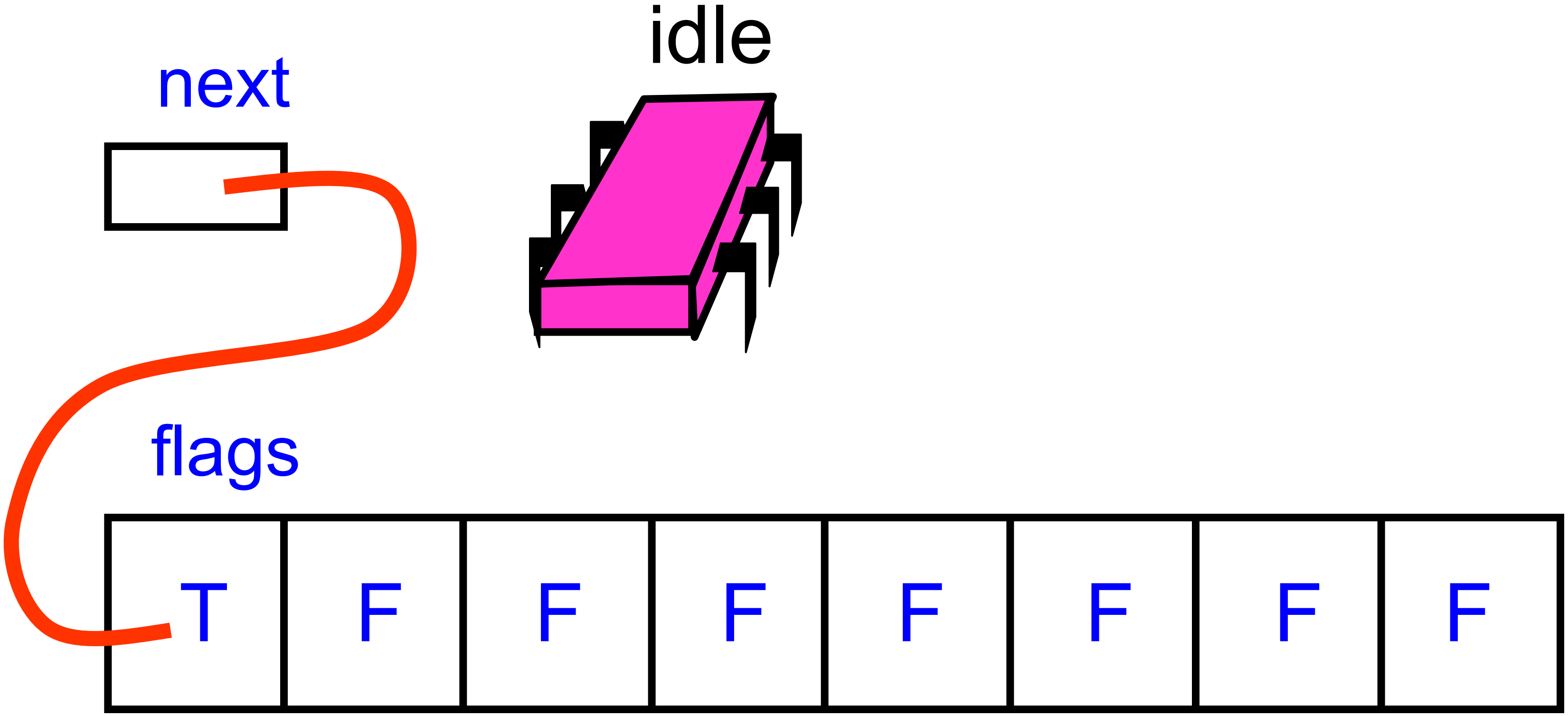
Actual Data on 40-Core Machine



Idea

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

Anderson Queue Lock



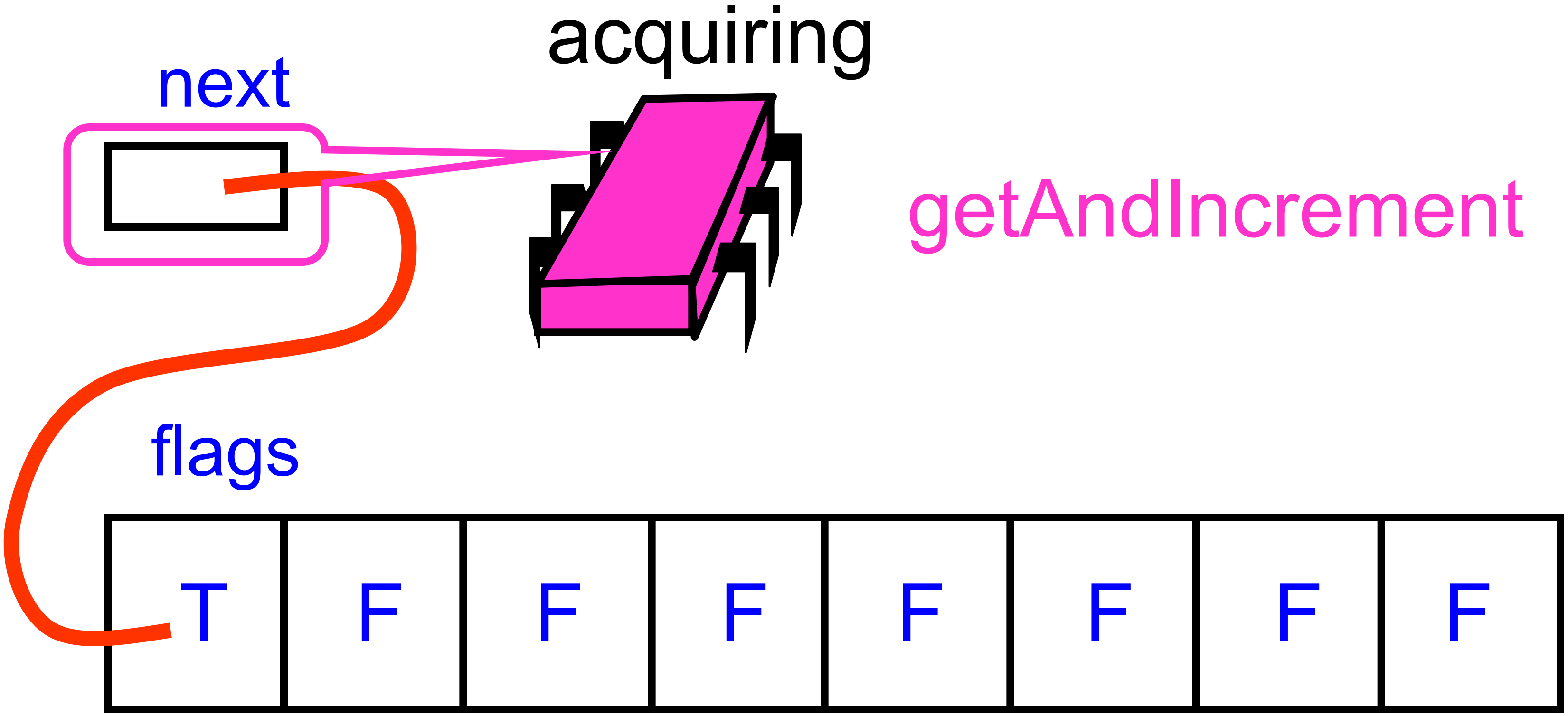
idle

next

flags

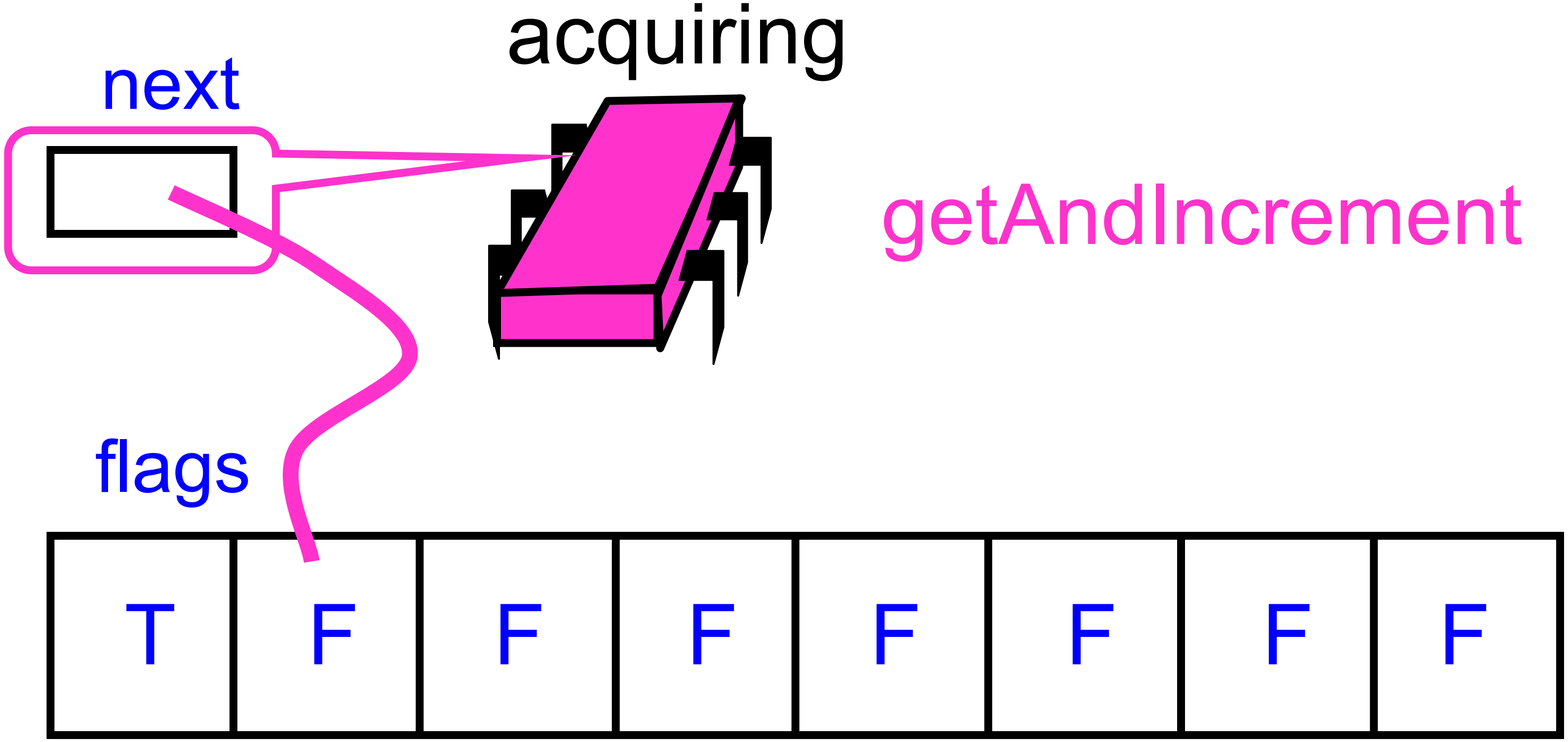
T	F	F	F	F	F	F	F

Anderson Queue Lock



T	F	F	F	F	F	F	F

Anderson Queue Lock



acquiring

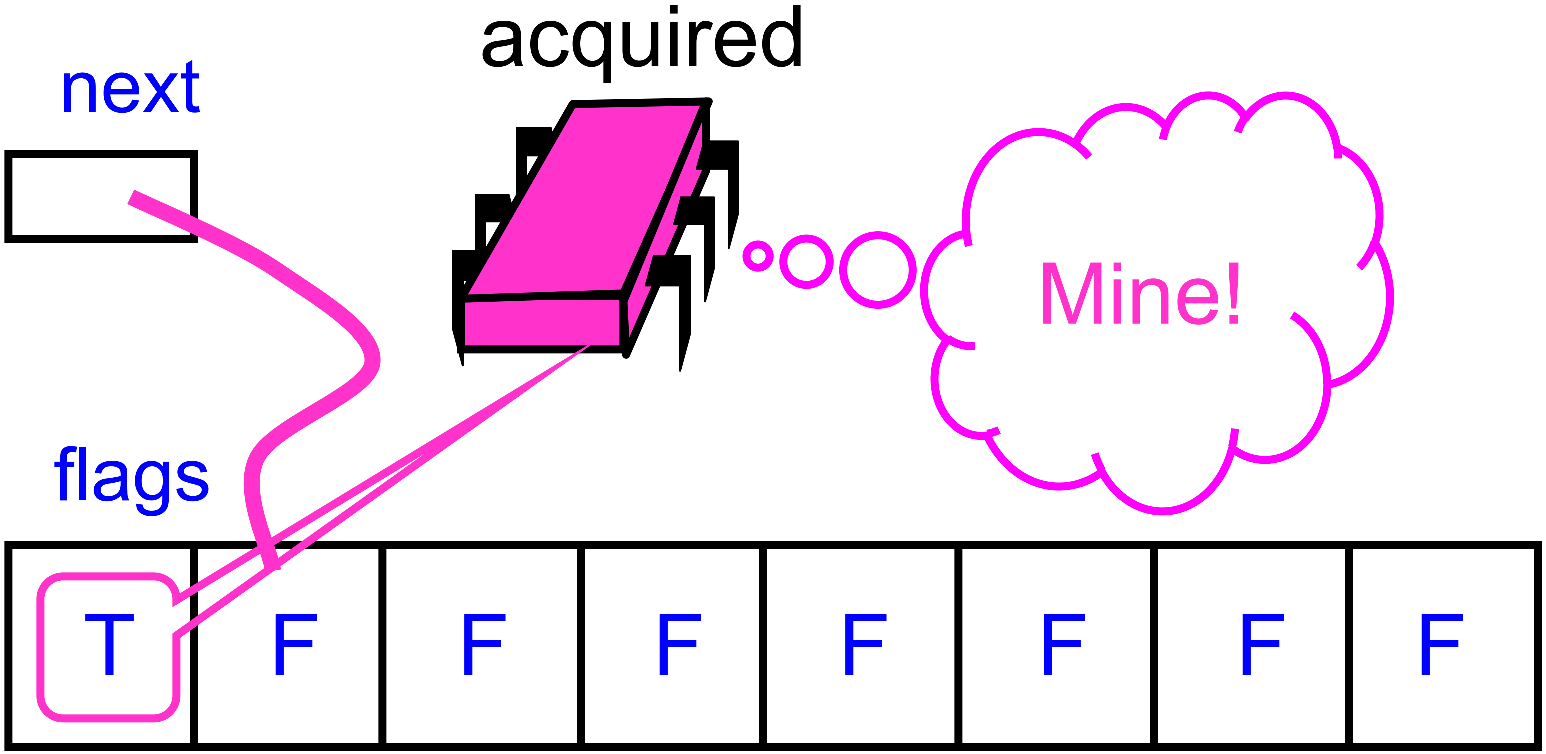
next

getAndIncrement

flags

T	F	F	F	F	F	F	F

Anderson Queue Lock



acquired

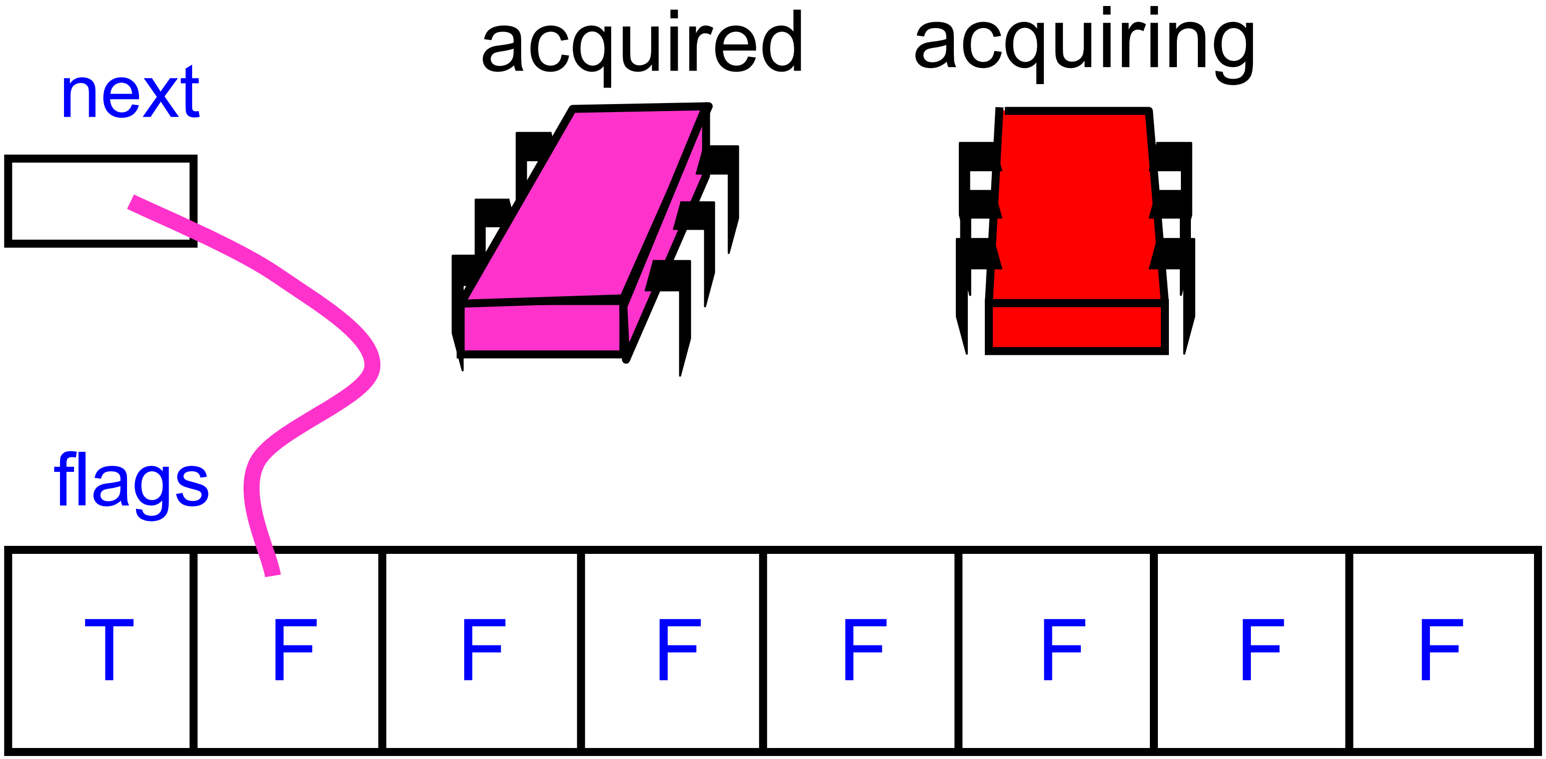
next

Mine!

flags

T	F	F	F	F	F	F	F

Anderson Queue Lock



next

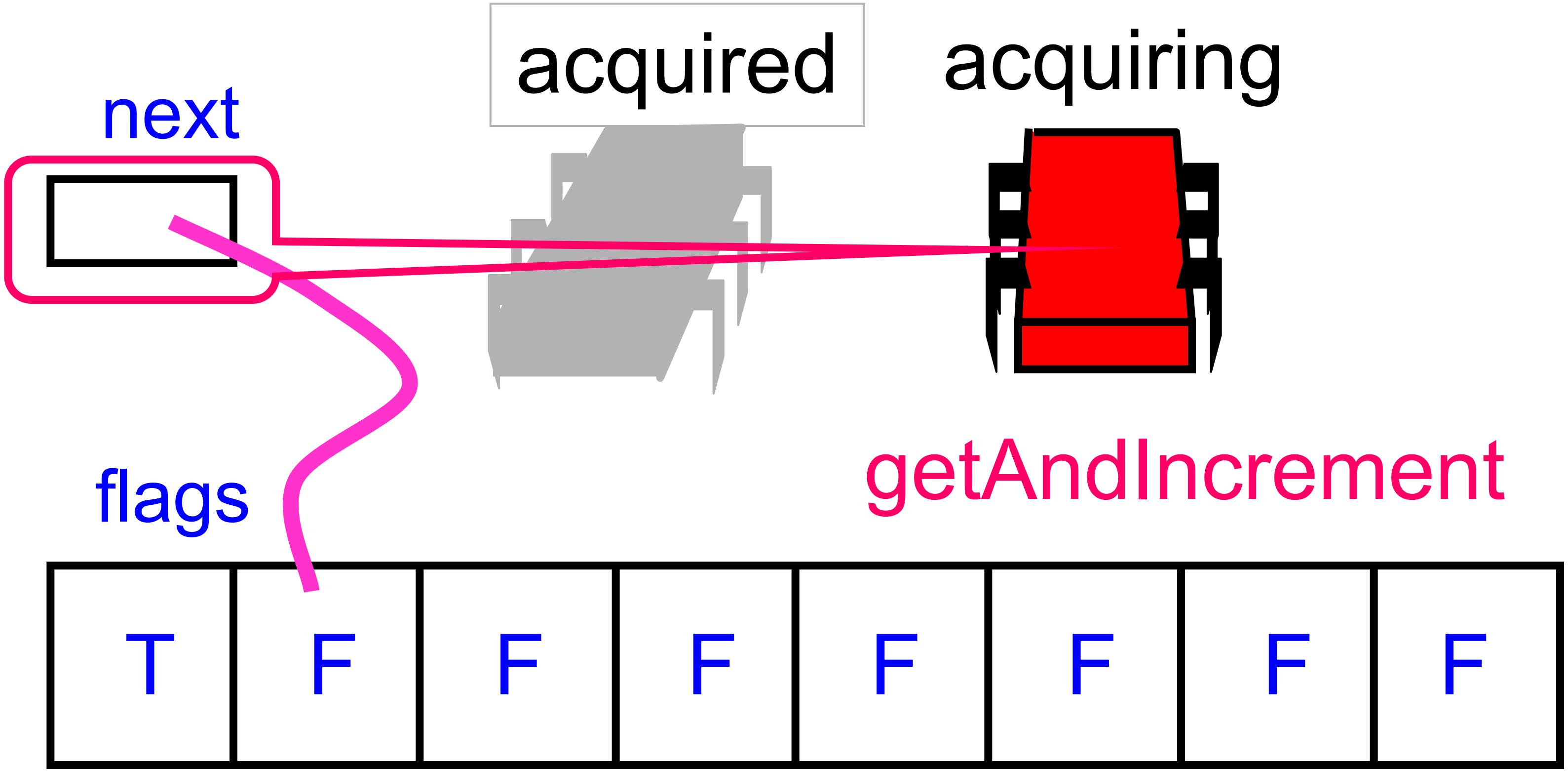
acquired

acquiring

flags

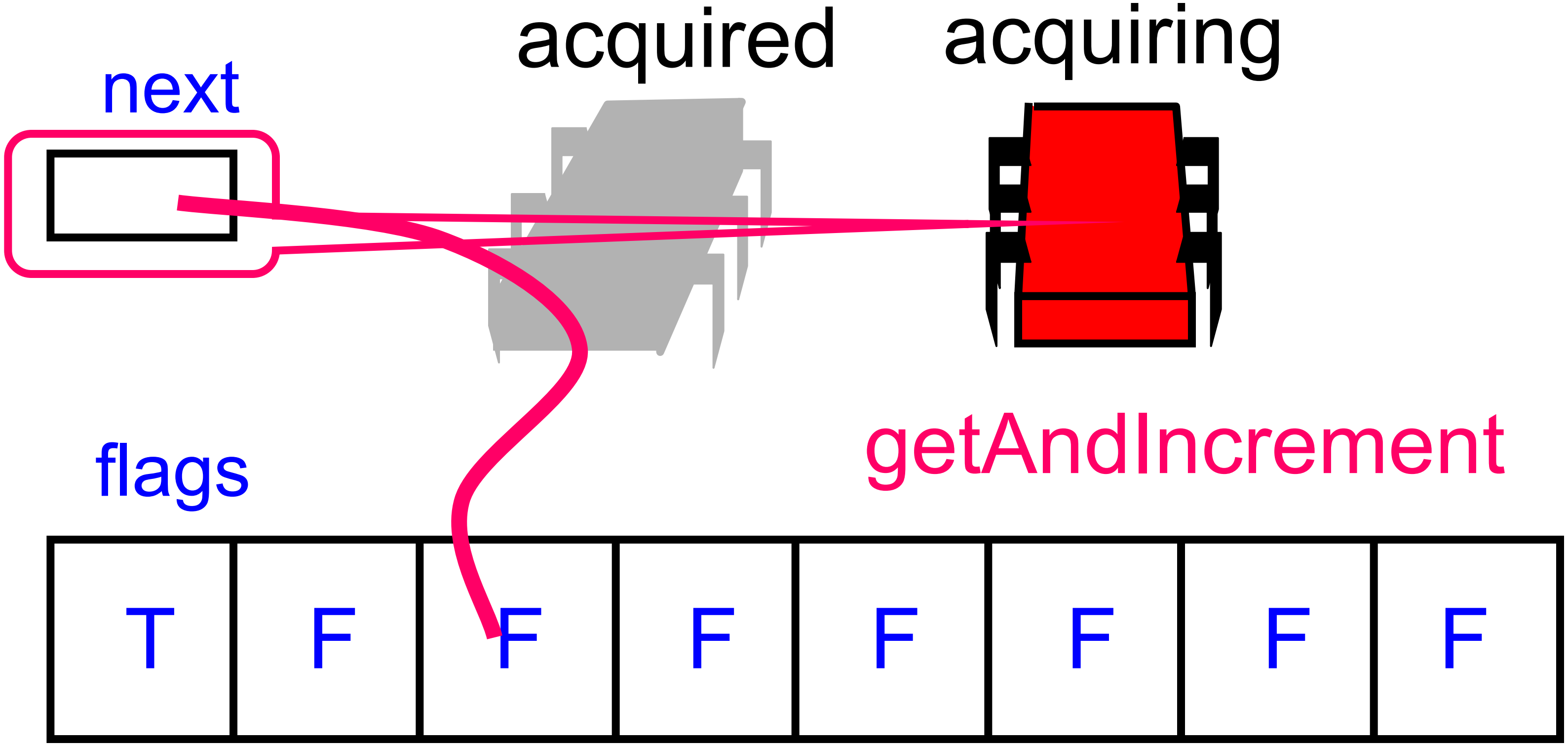
T	F	F	F	F	F	F	F

Anderson Queue Lock

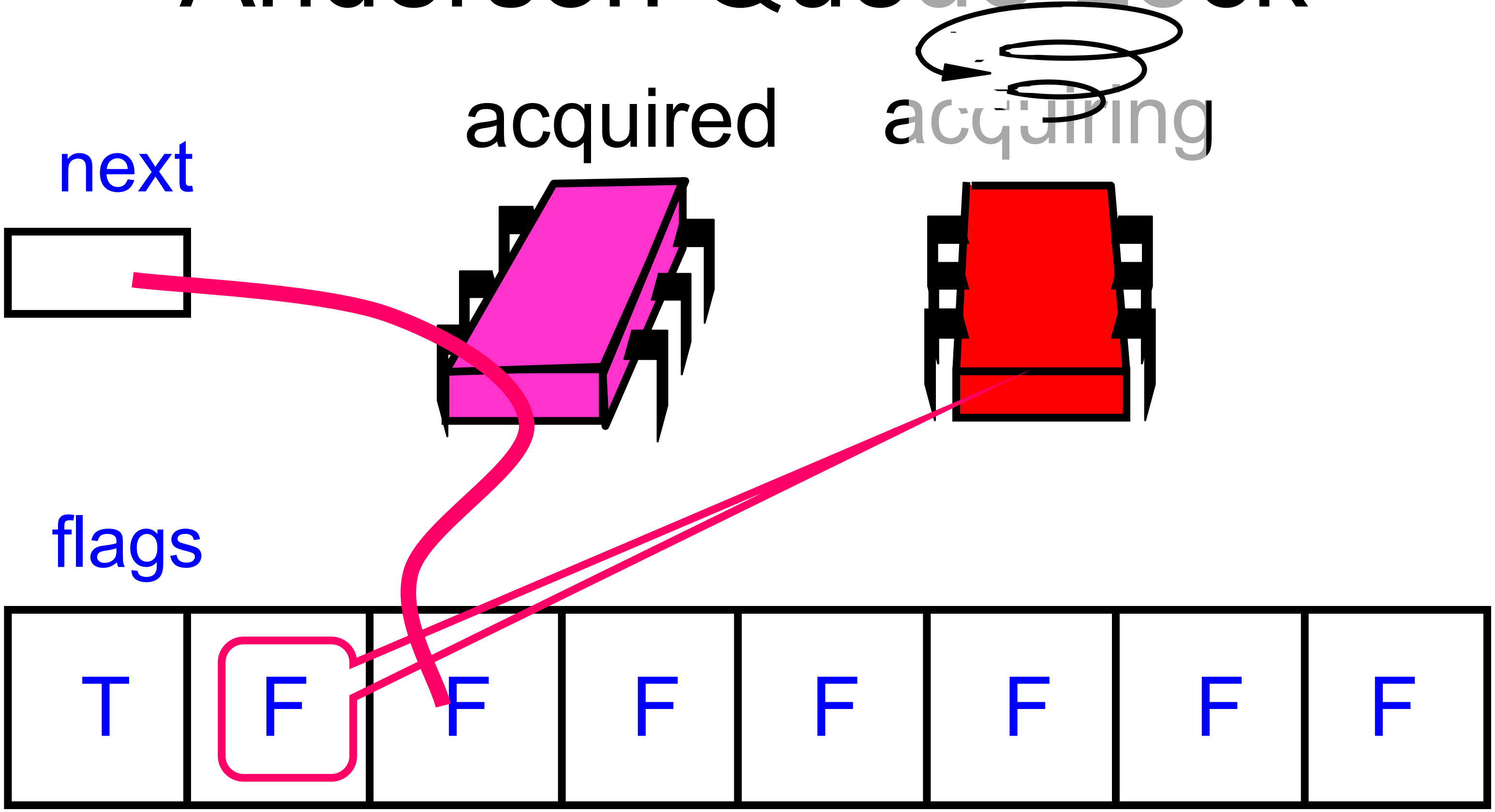


T	F	F	F	F	F	F	F

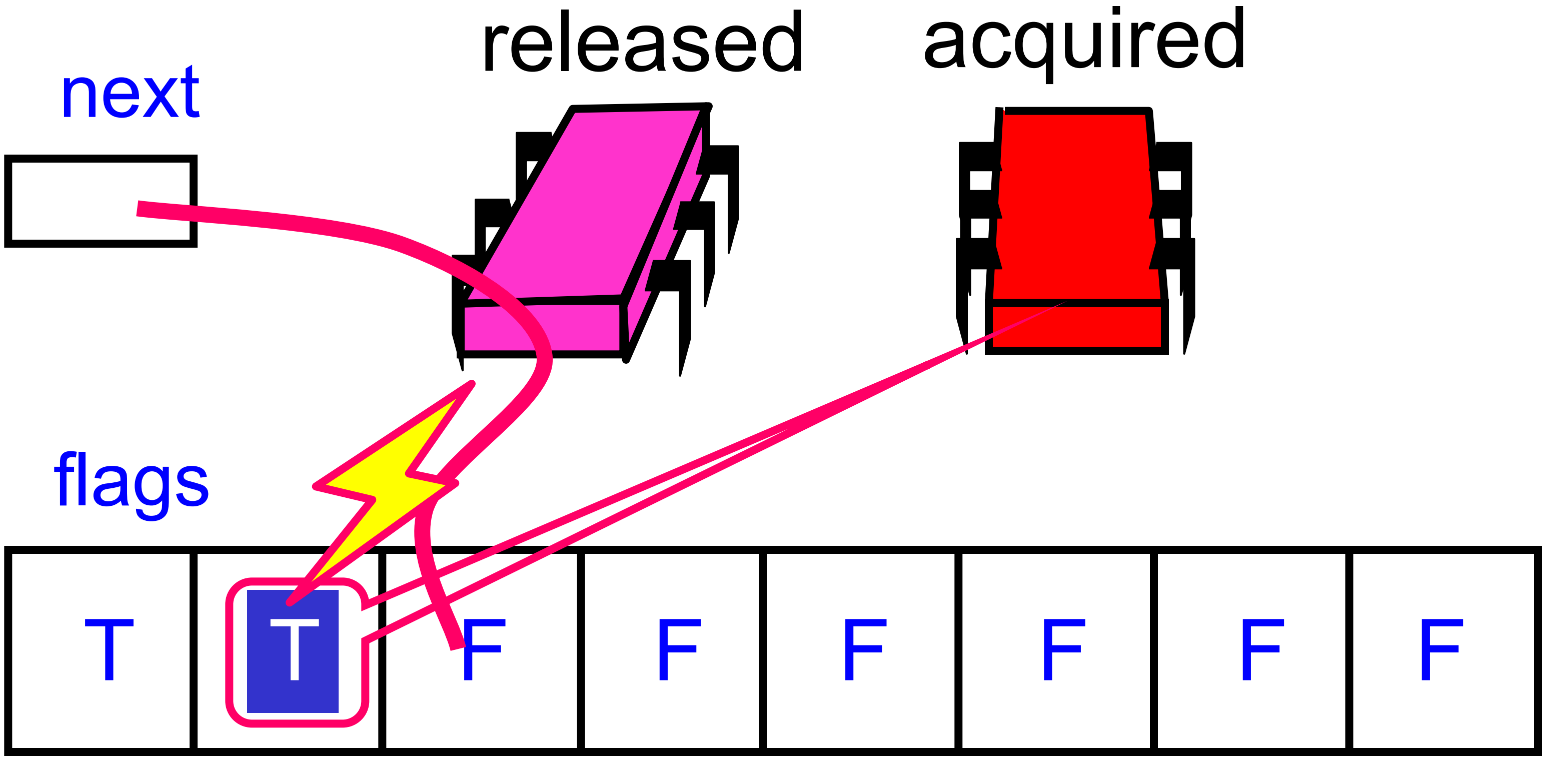
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



released

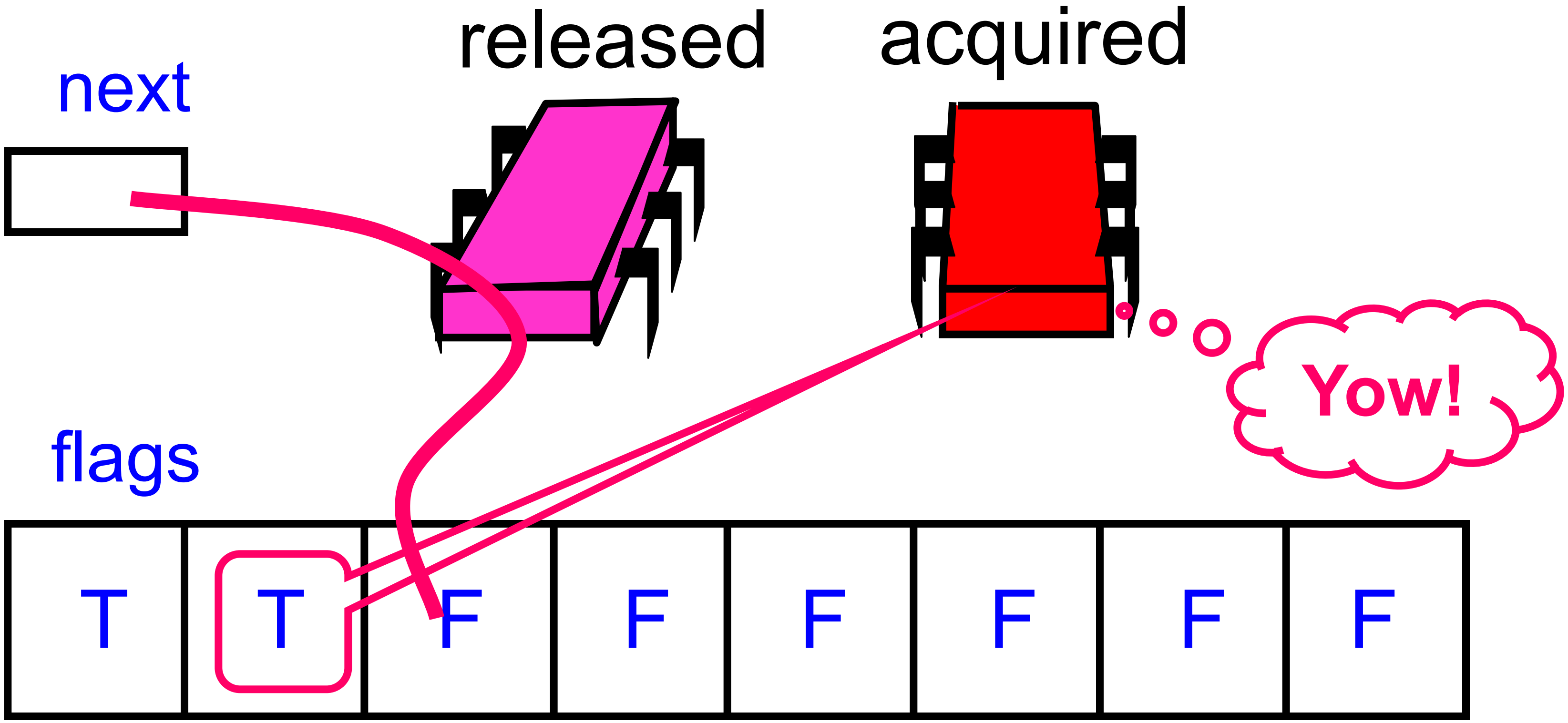
acquired

next

flags

T	T	F	F	F	F	F	F

Anderson Queue Lock



released

acquired

next

Yow!

flags

T	T	F	F	F	F	F	F

Anderson Queue Lock

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

Anderson Queue Lock

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

One flag per thread

Anderson Queue Lock

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

Next flag to use

Anderson Queue Lock

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

Thread-local variable

Anderson Queue Lock

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

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

Anderson Queue Lock

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

public unlock() {
 flags[(mySlot+1) % n]
}
```

Take next slot

Anderson Queue Lock

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

public unlock() {
 flags[(mySlot+1
}
```

Spin until told to go

Anderson Queue Lock

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

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

Prepare slot for re-use

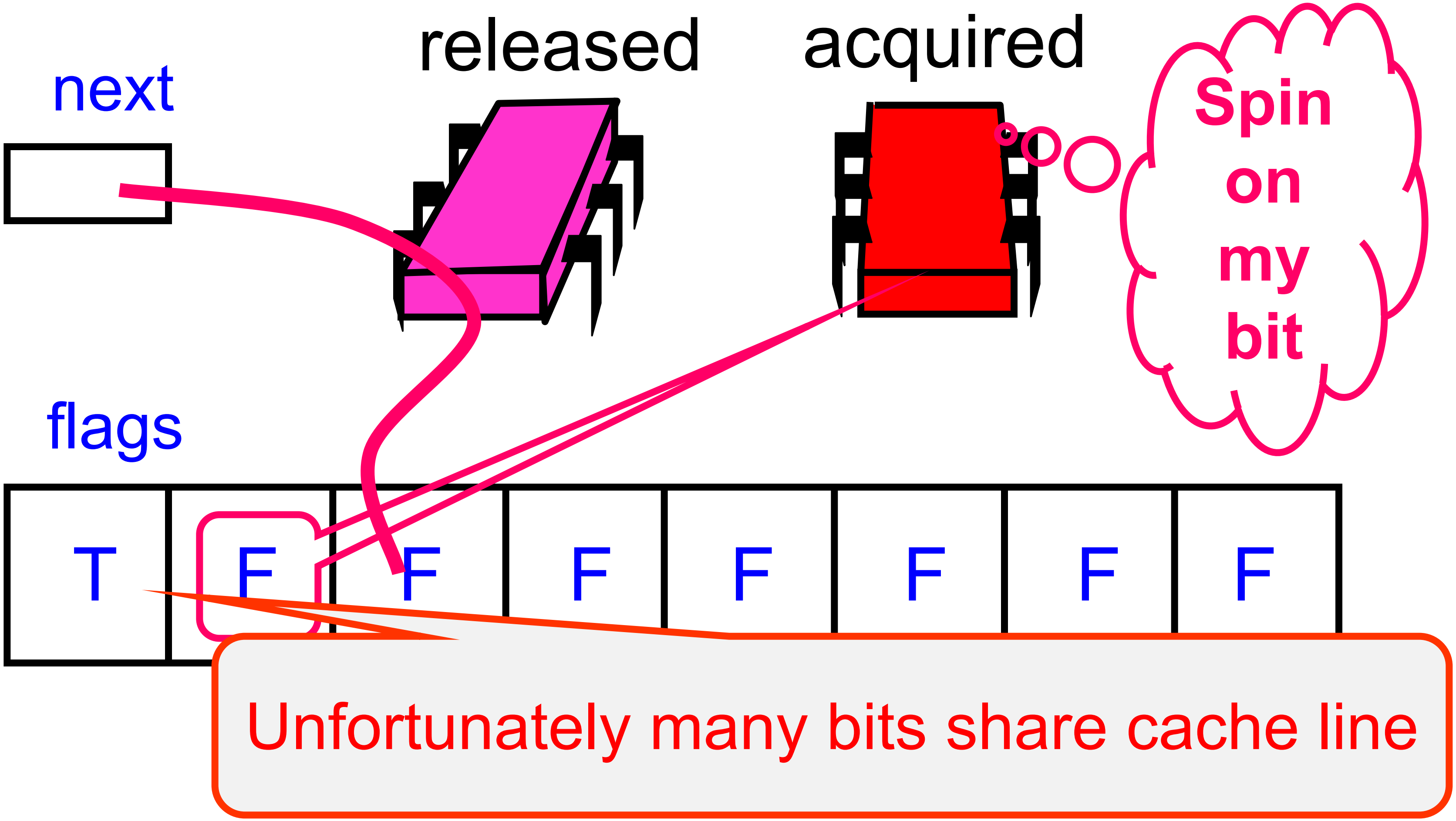
Anderson Queue Lock

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

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

Tell next thread to go

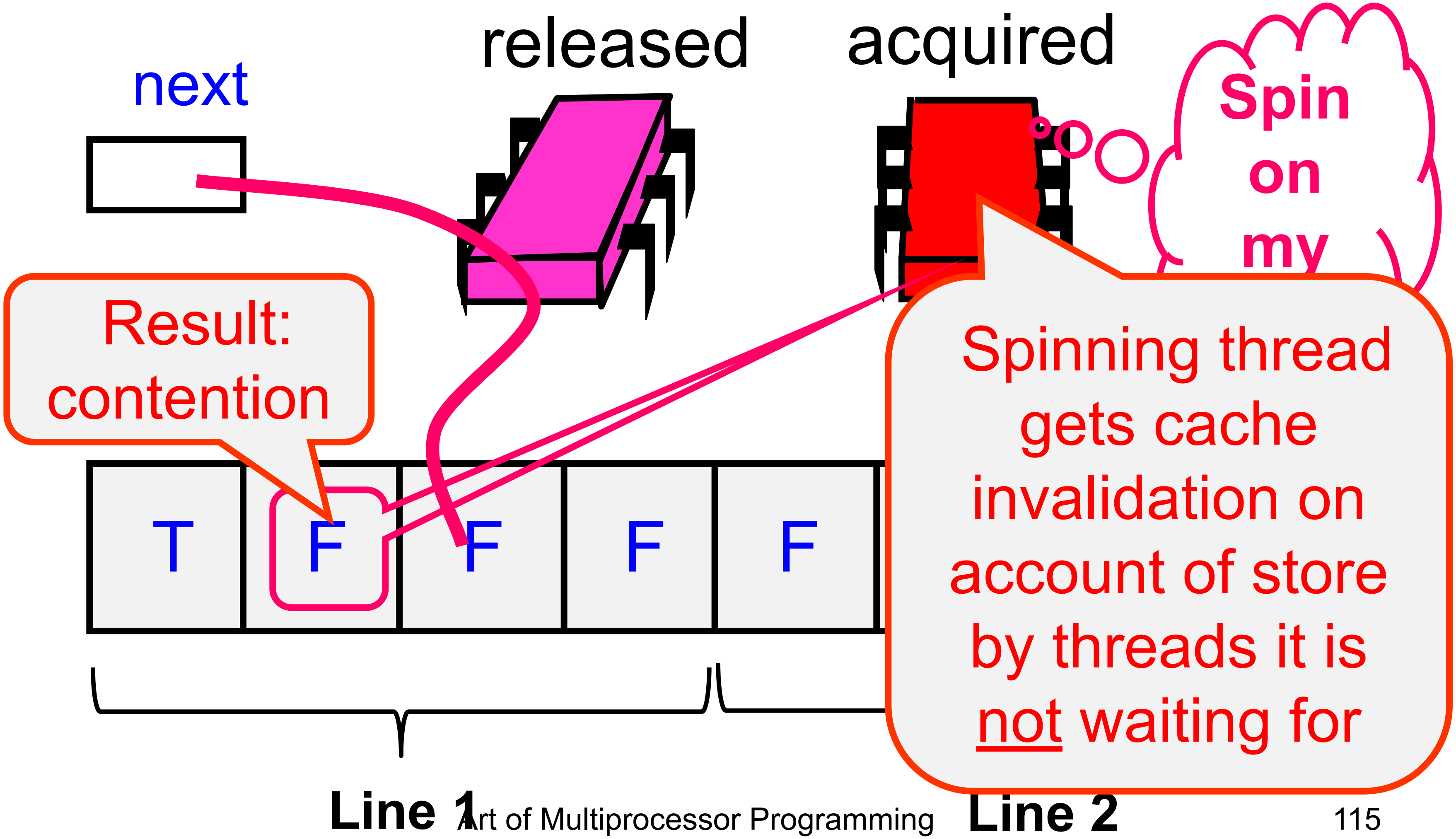
Local Spinning



T	F	F	F	F	F	F	F

Unfortunately many bits share cache line

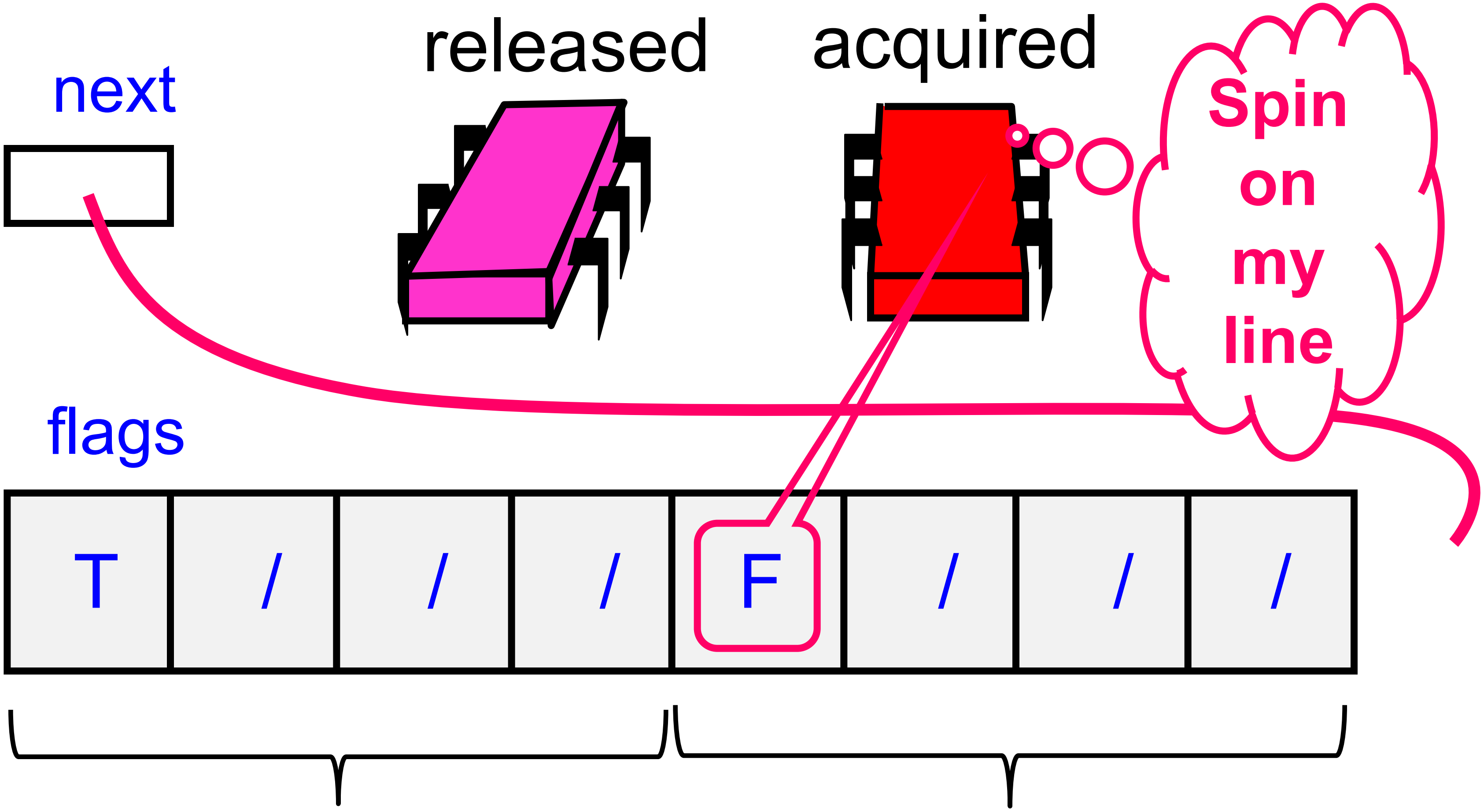
False Sharing



Result: contention

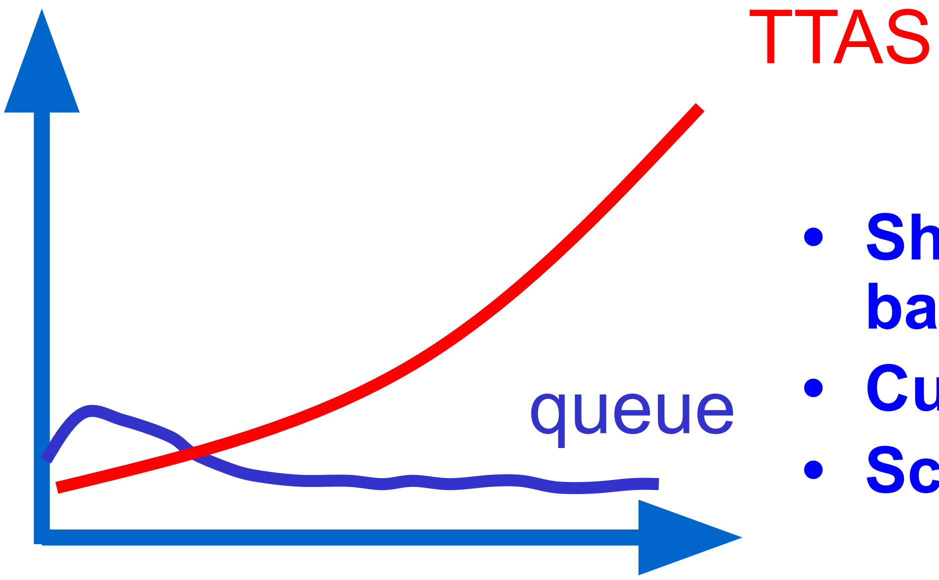
Spinning thread gets cache invalidation on account of store by threads it is not waiting for

The Solution: Padding



Line 1 Line 2

Performance



- **Shorter handover than backoff**
- **Curve is practically flat**
- **Scalable performance**

Anderson Queue Lock

Good

- First truly scalable lock
- Simple, easy to implement
- Back to FCFS order (like Bakery)

Anderson Queue Lock

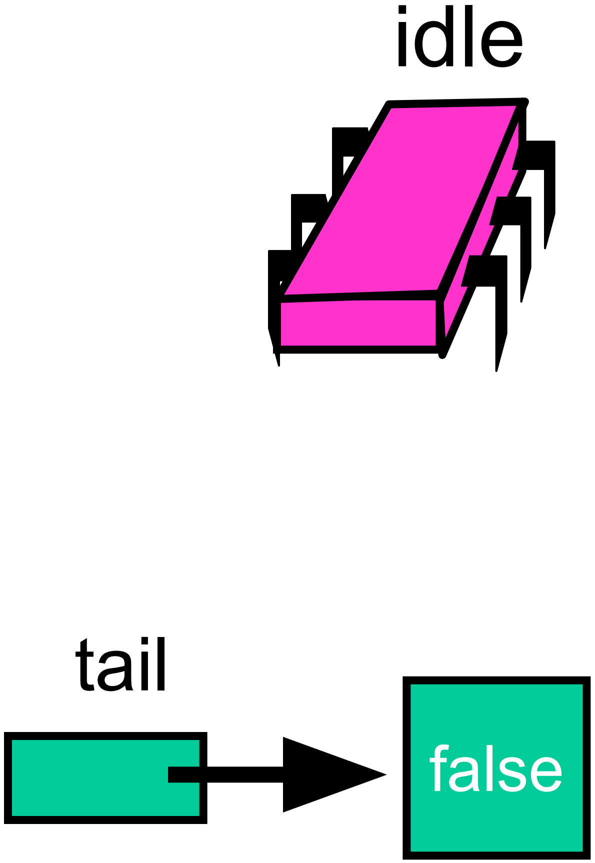
Bad

- Space hog…
- One bit per thread □ one cache line per thread
  - What if unknown number of threads?
  - What if small number of actual contenders?

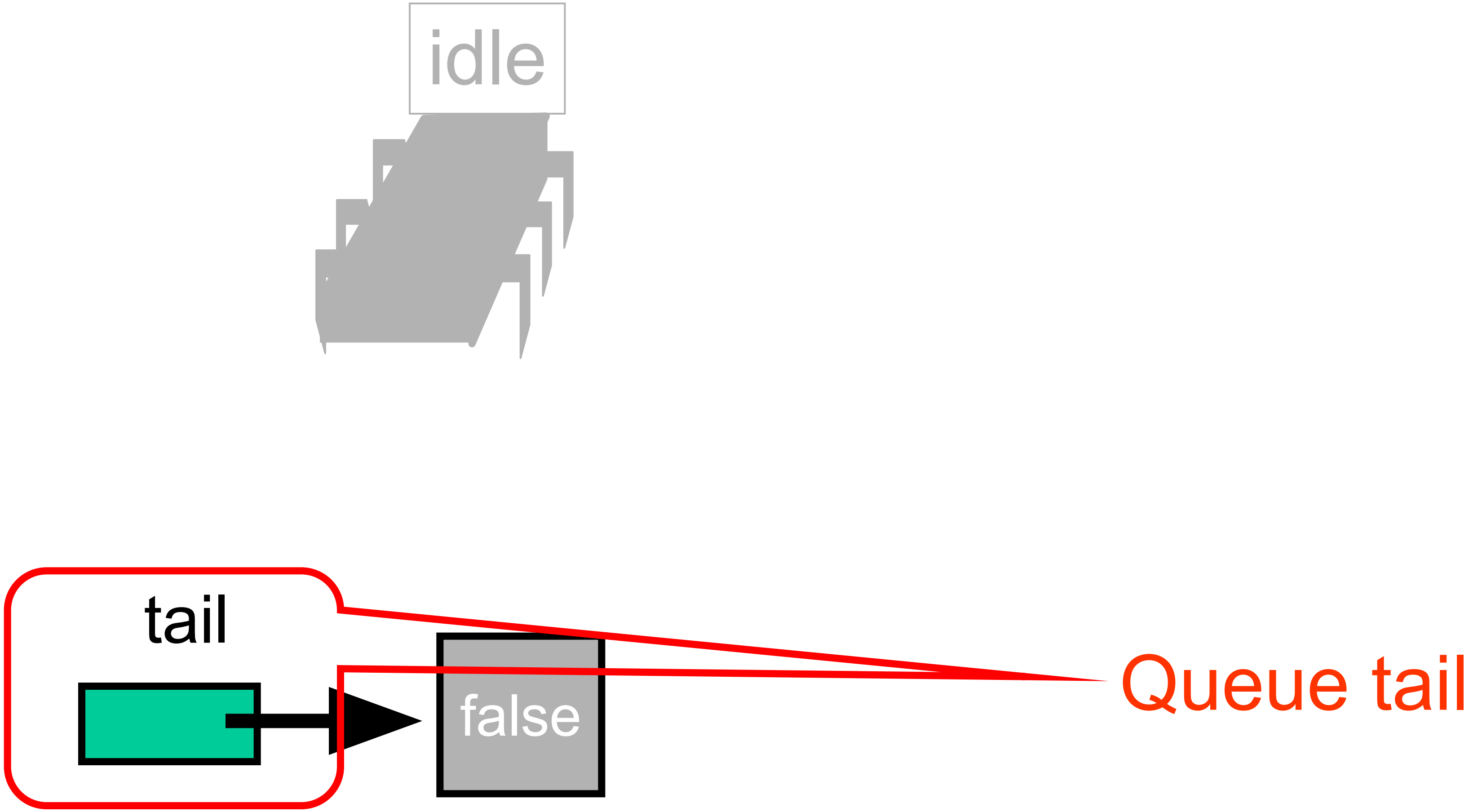
CLH Lock

- FCFS order
- Small, constant-size overhead per thread

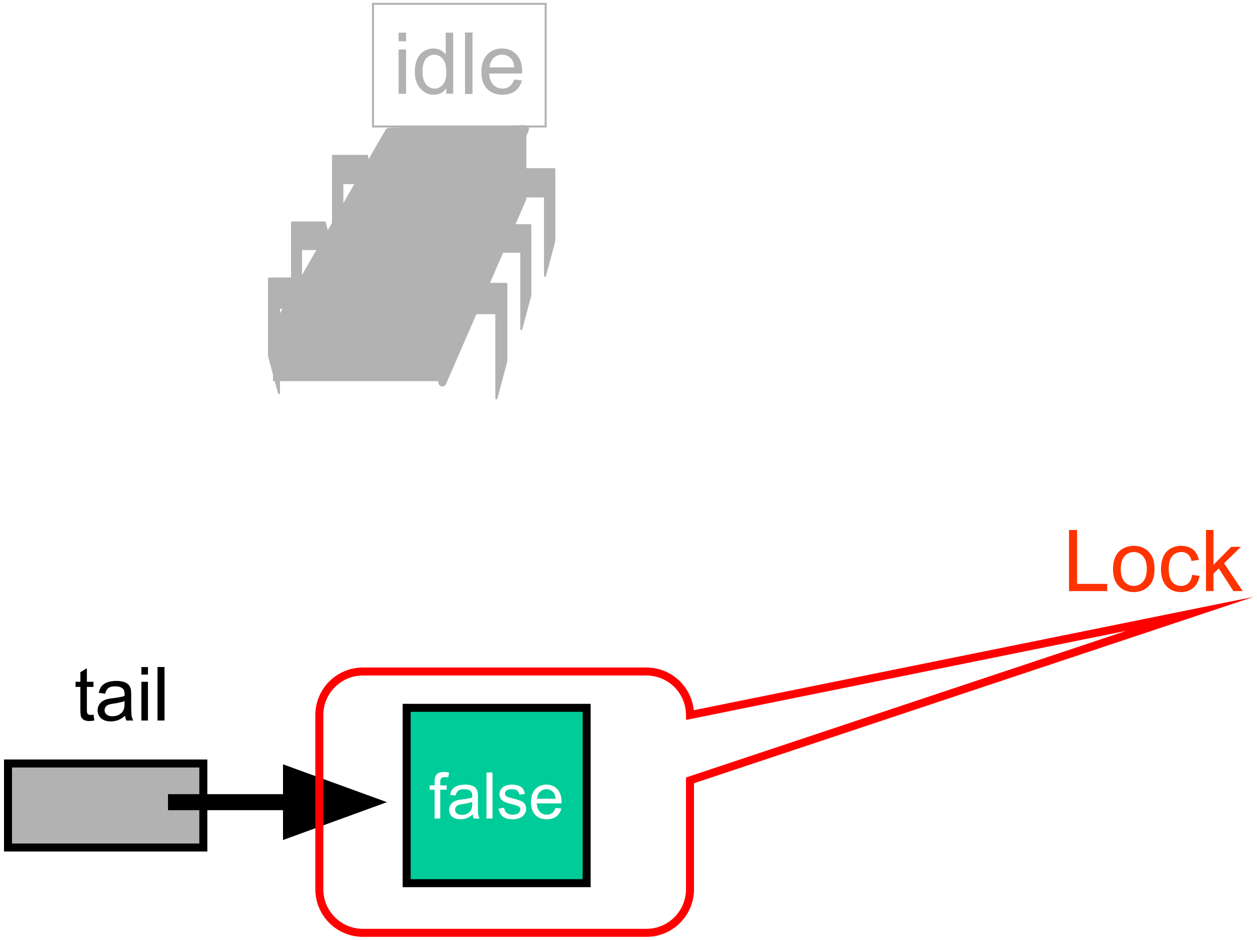
Initially



Initially



Initially



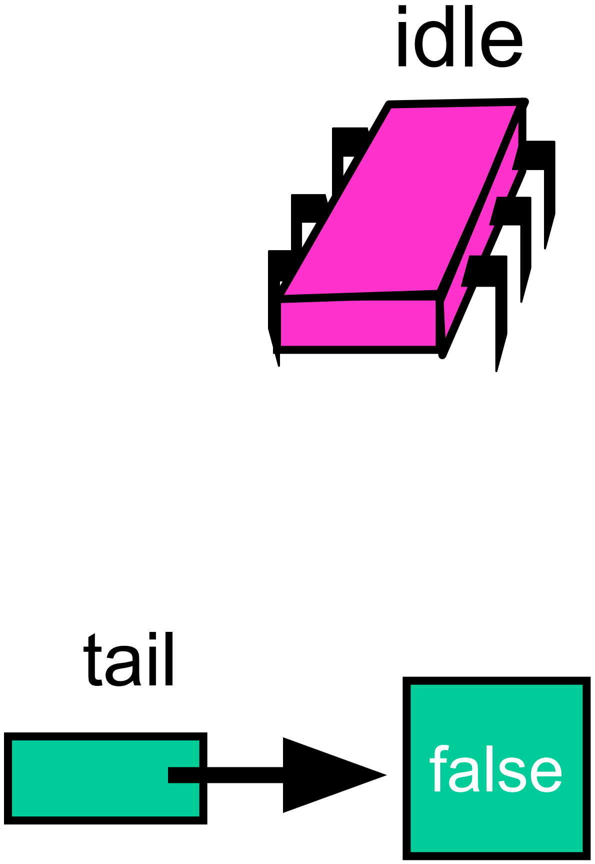
idle

tail

false

Lock is free

Initially

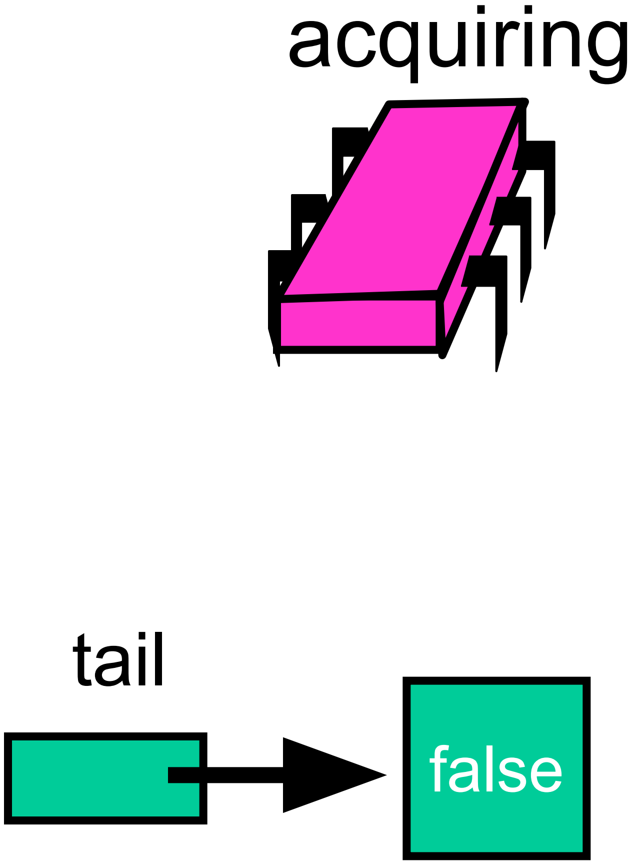


idle

tail

false

Purple Wants the Lock

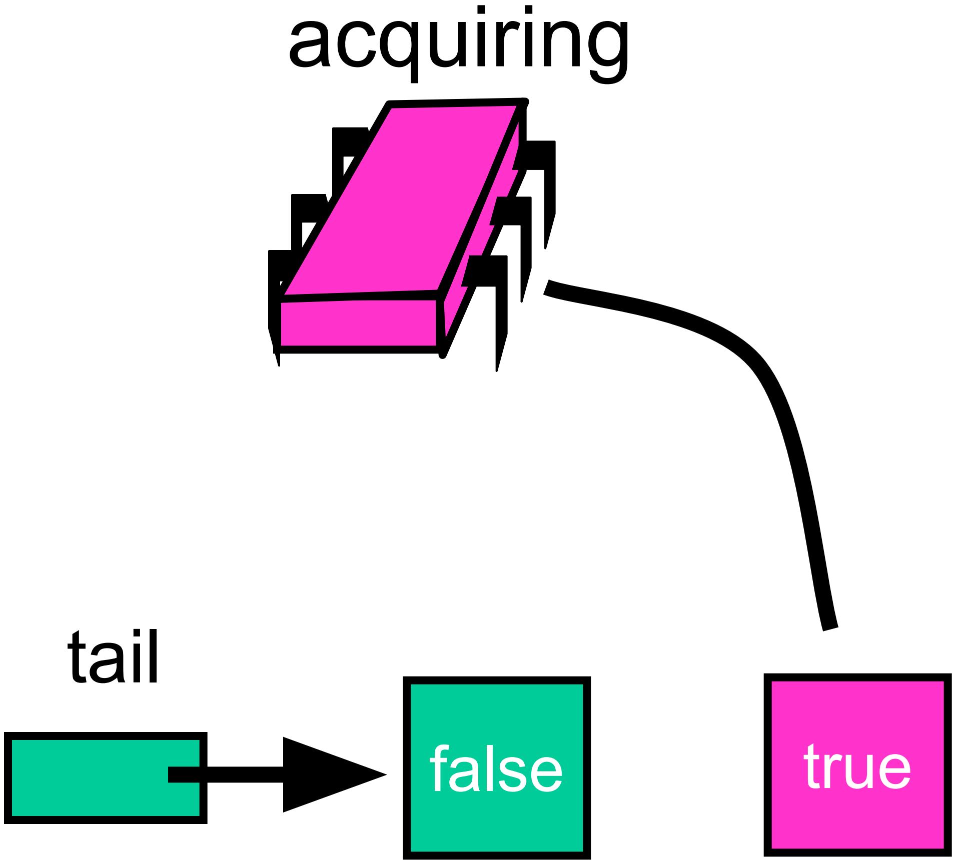


acquiring

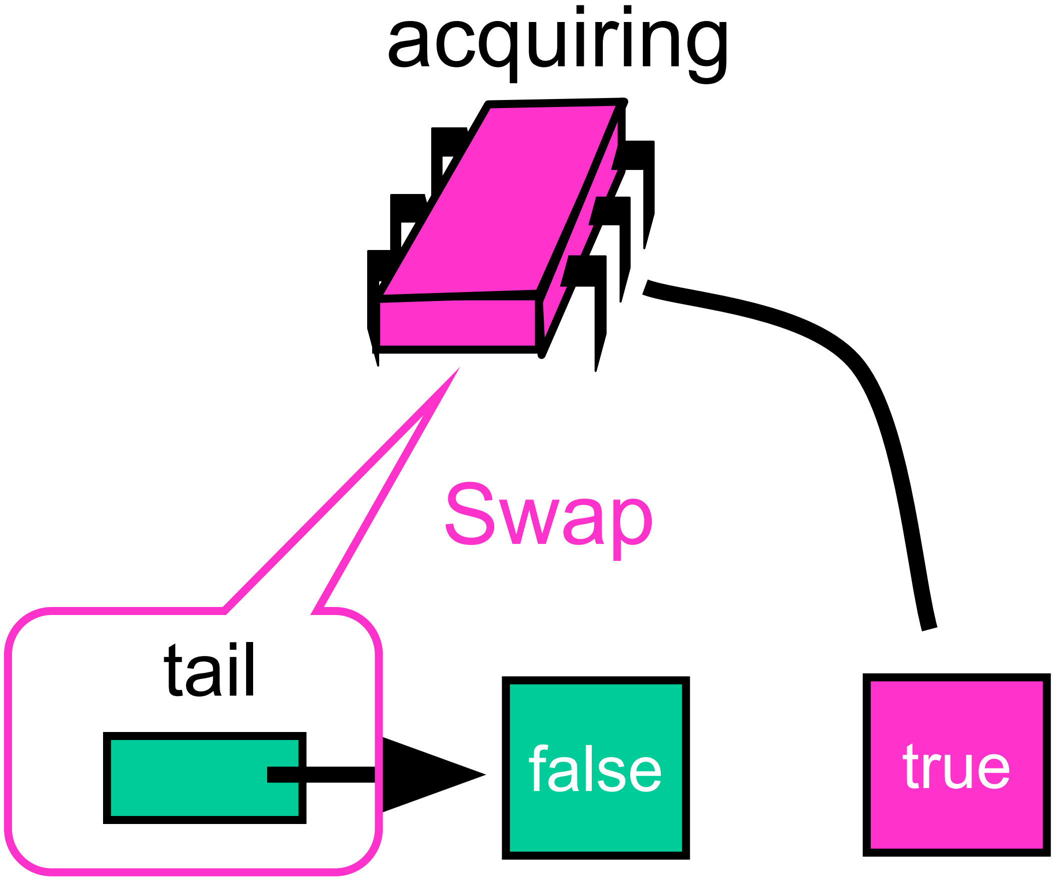
tail

false

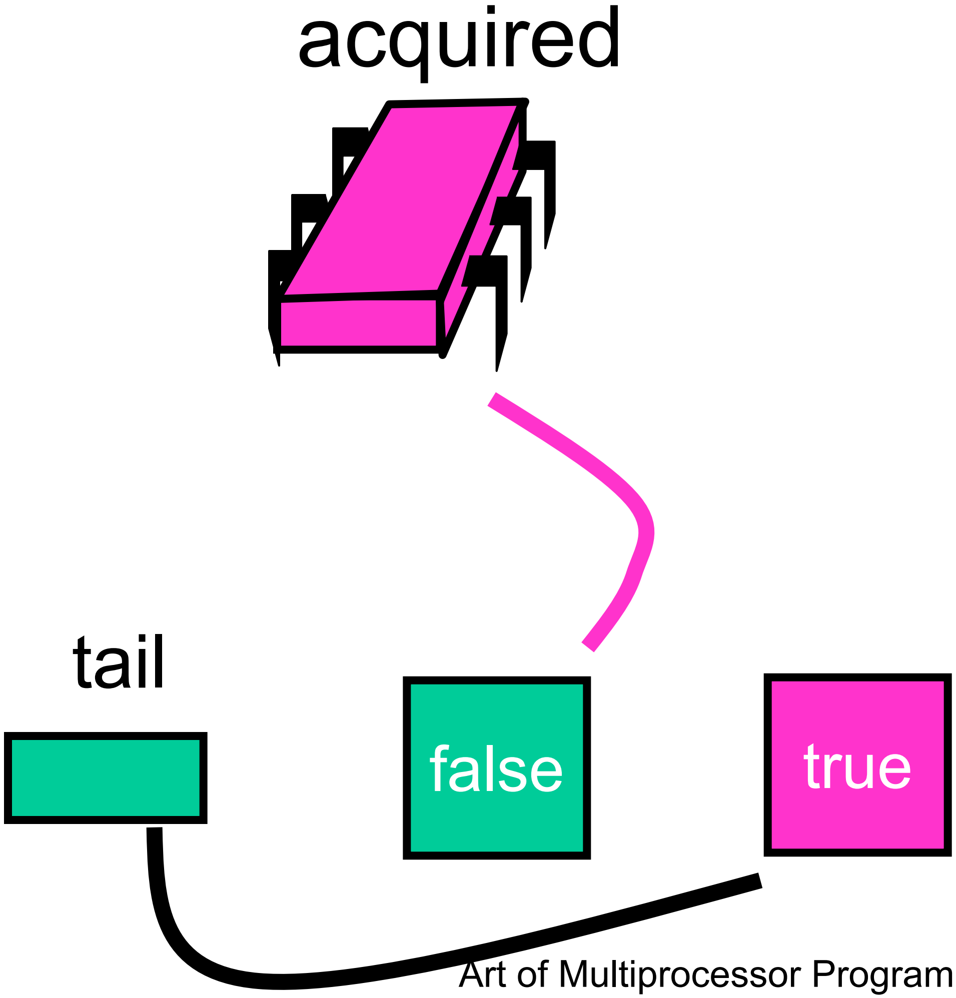
Purple Wants the Lock



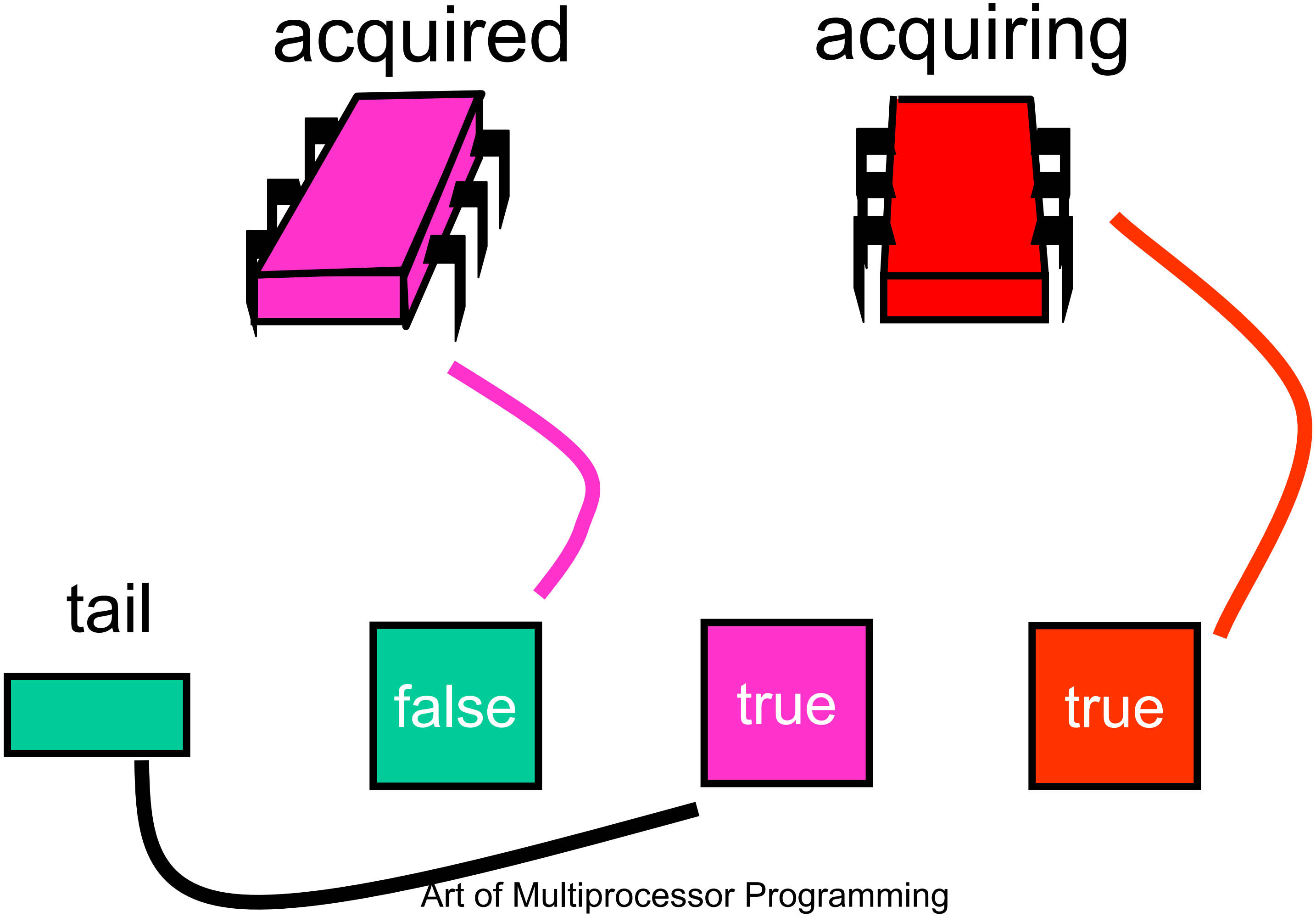
Purple Wants the Lock



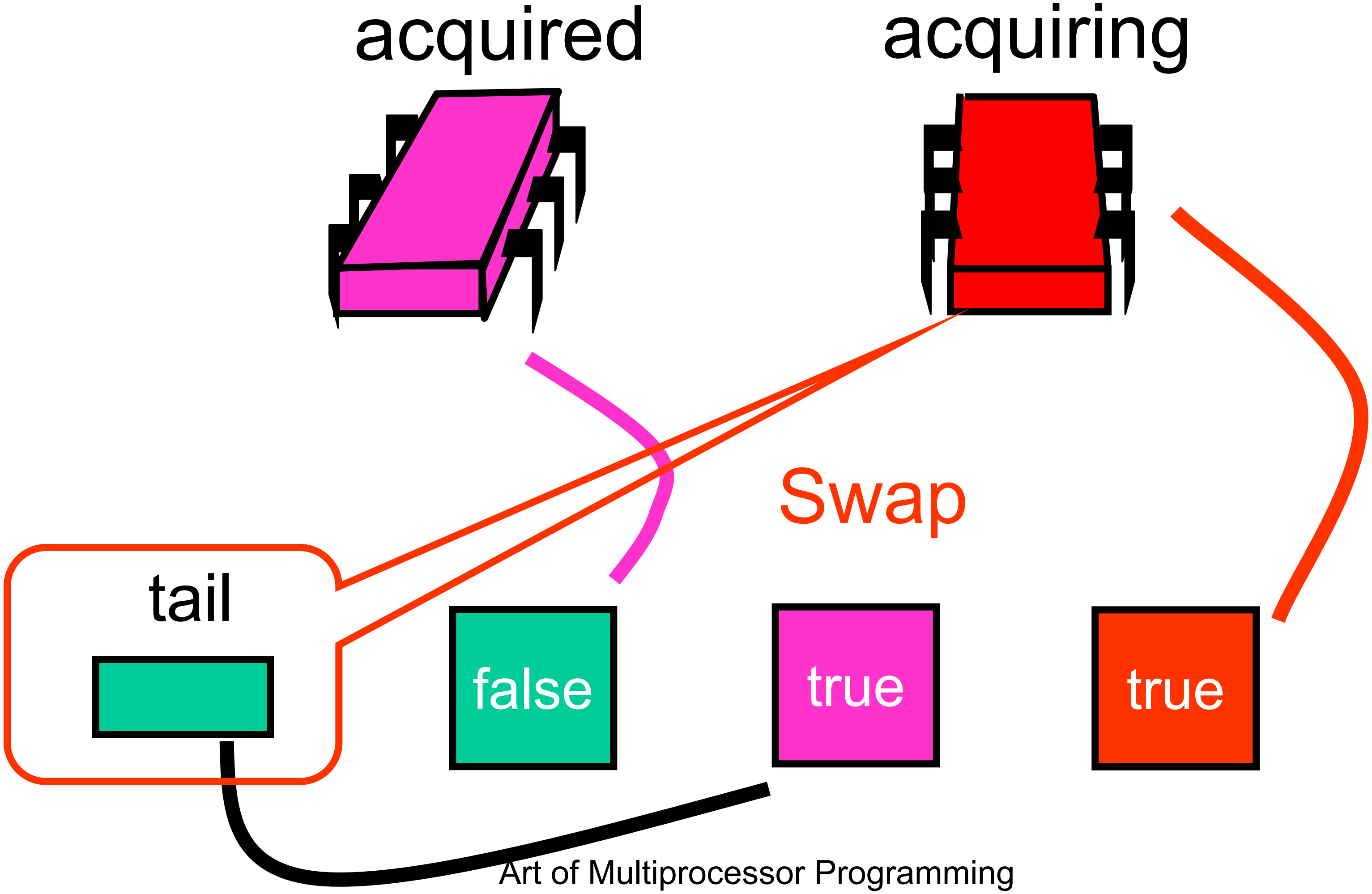
Purple Has the Lock



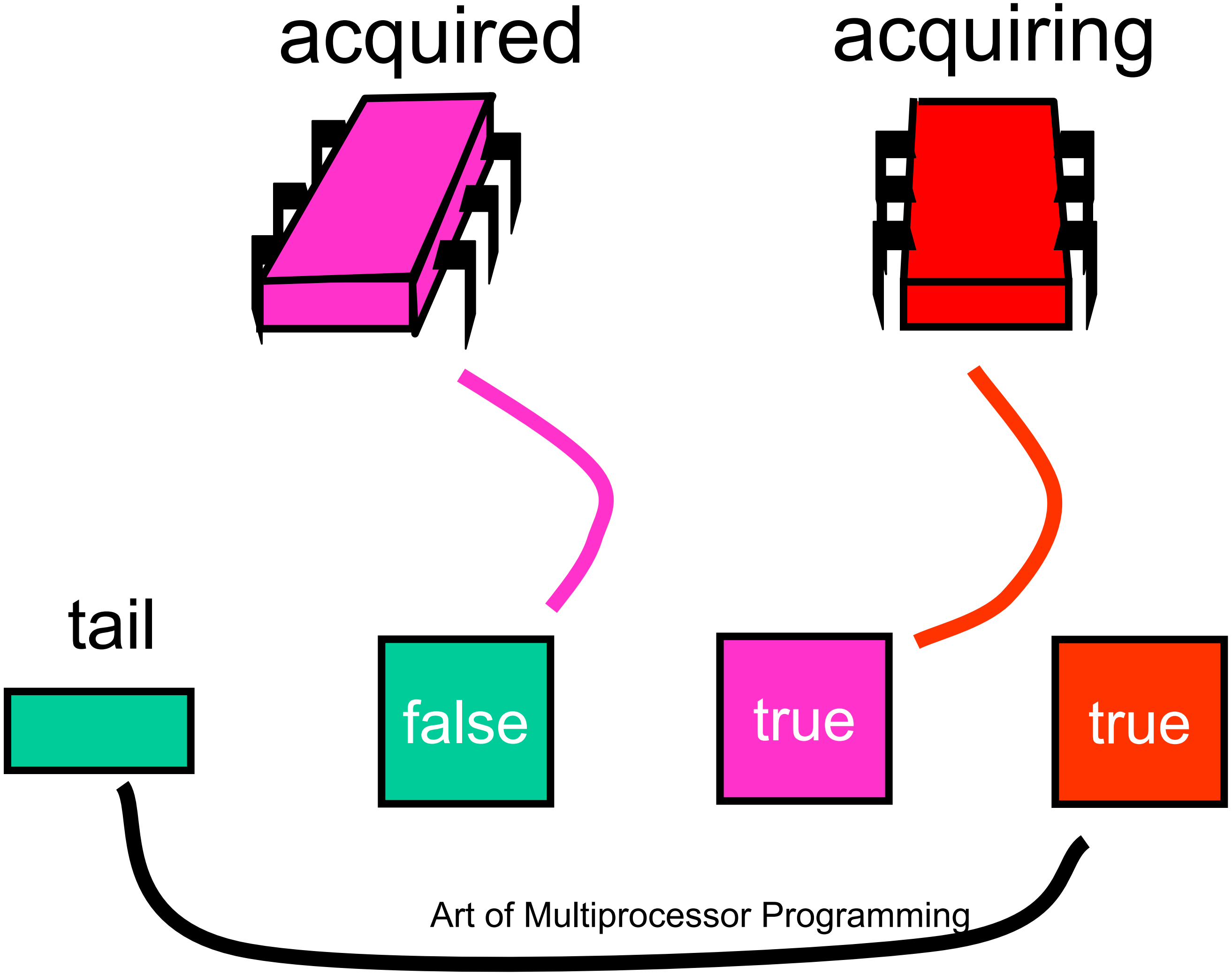
Red Wants the Lock



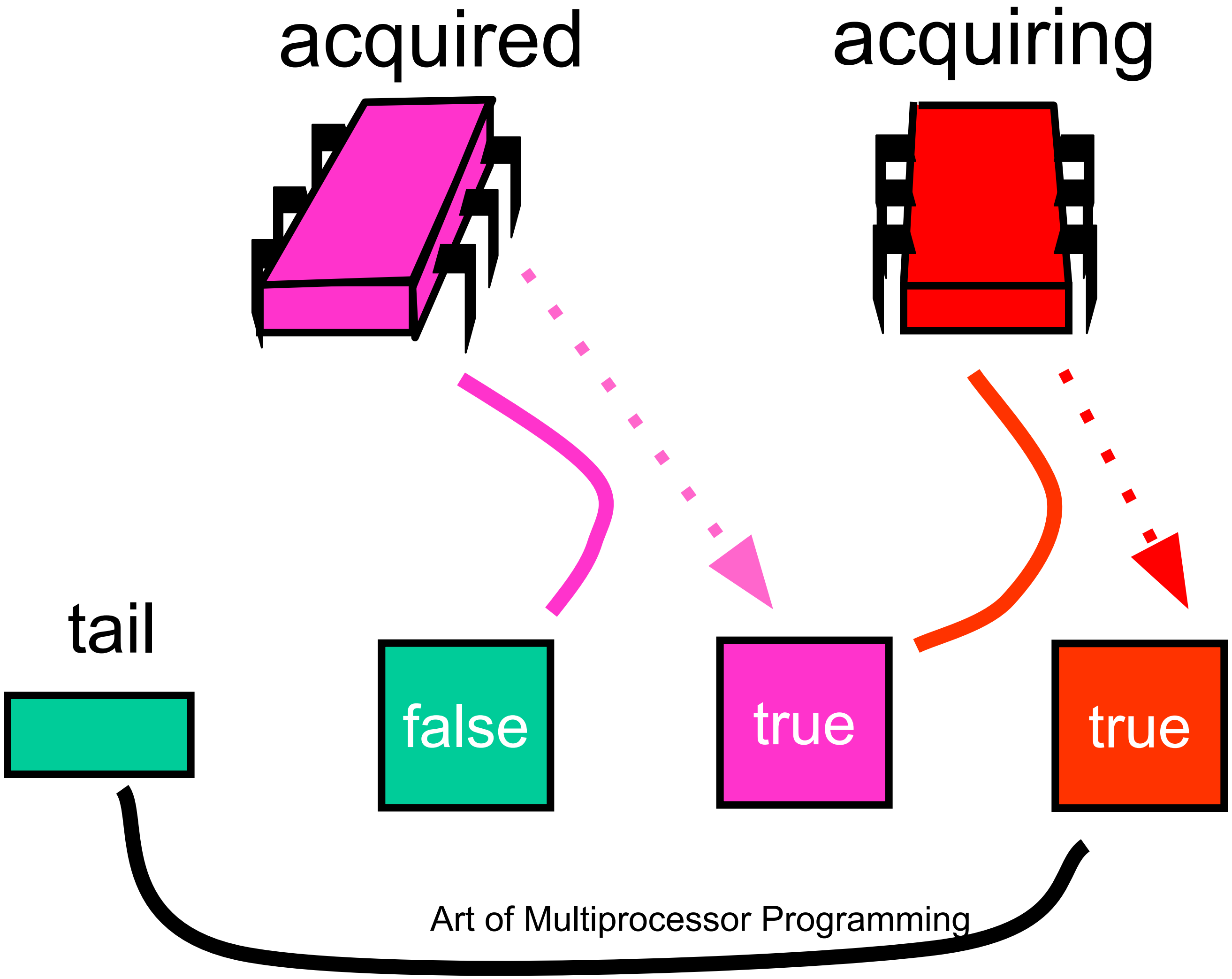
Red Wants the Lock



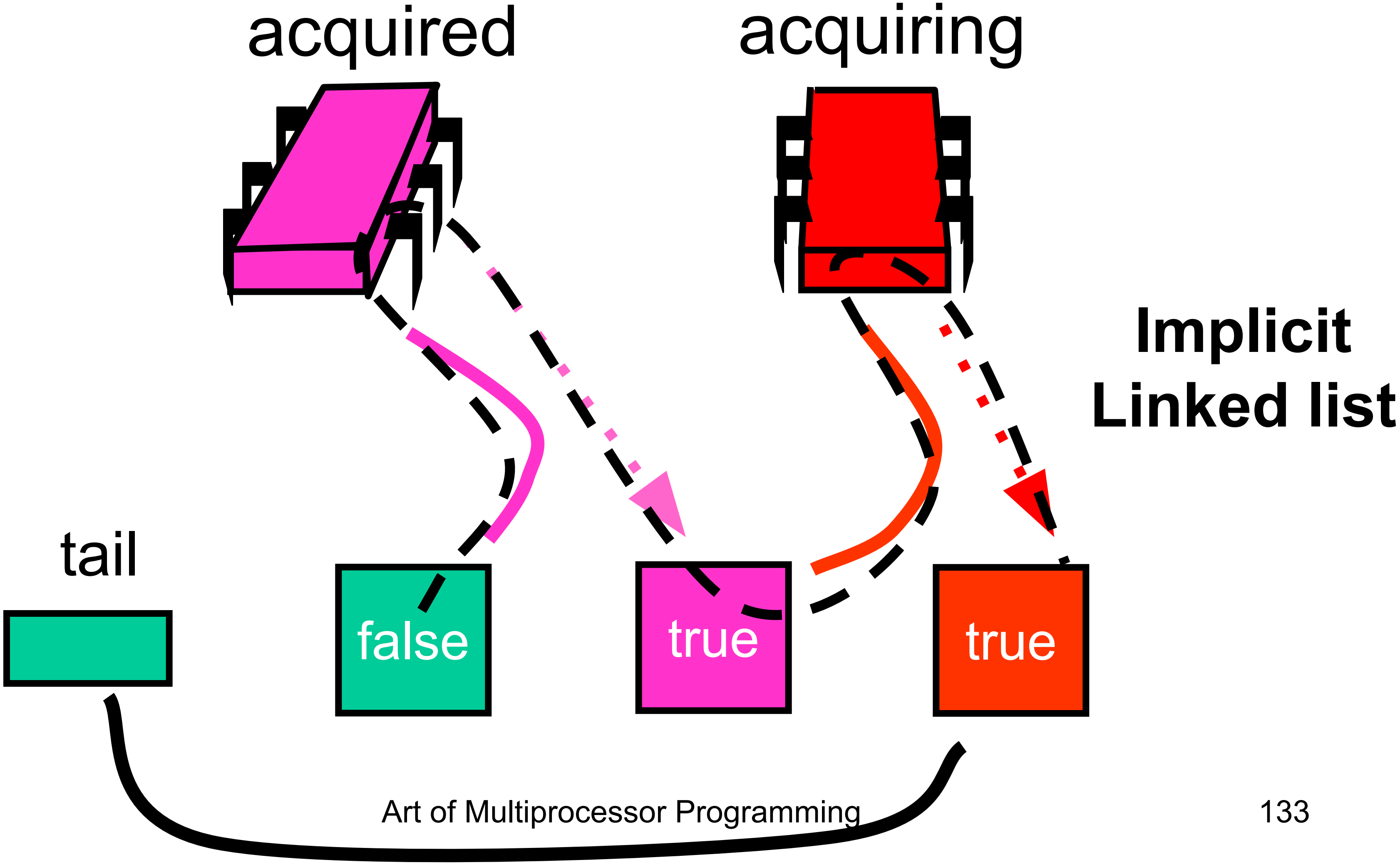
Red Wants the Lock



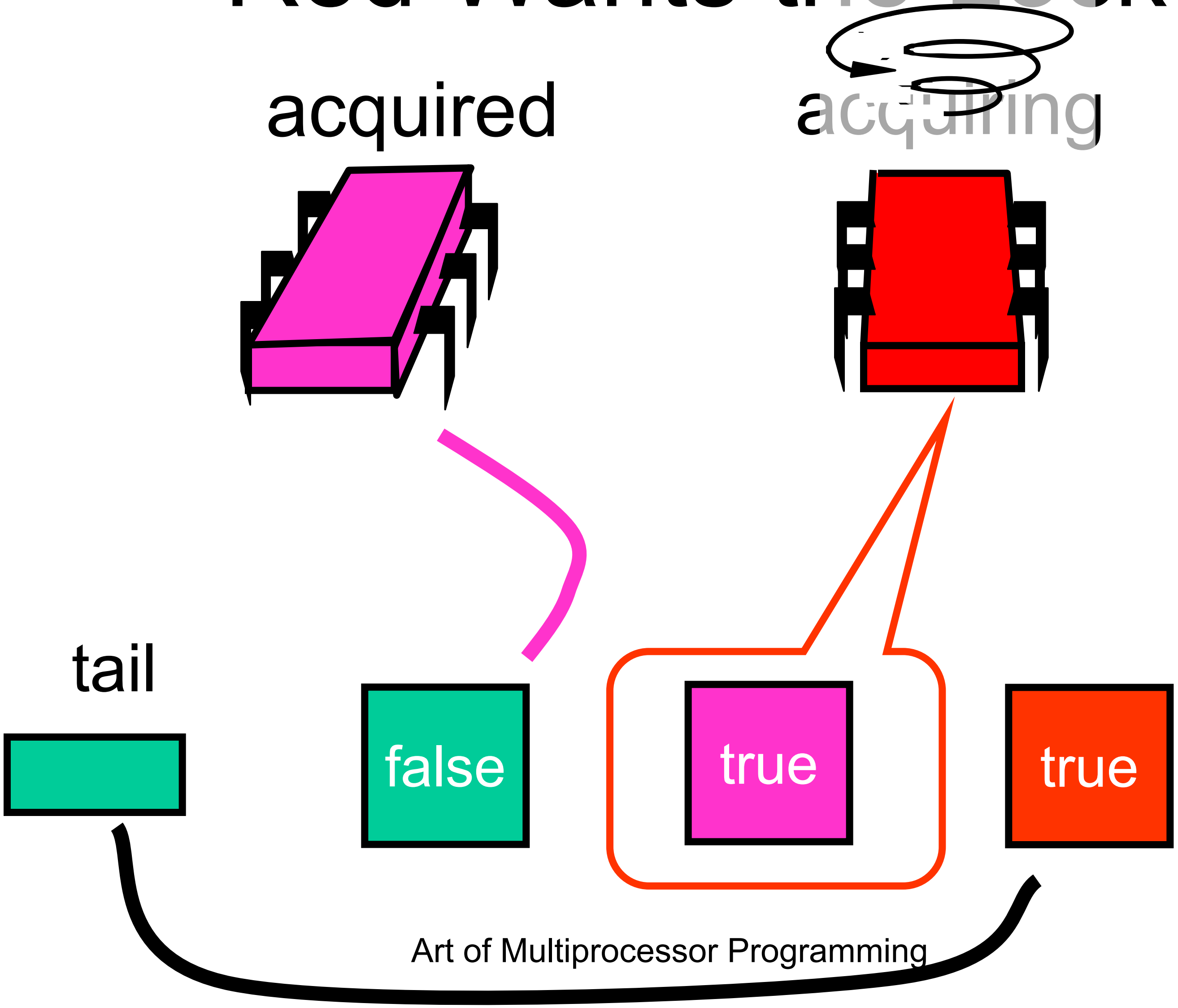
Red Wants the Lock



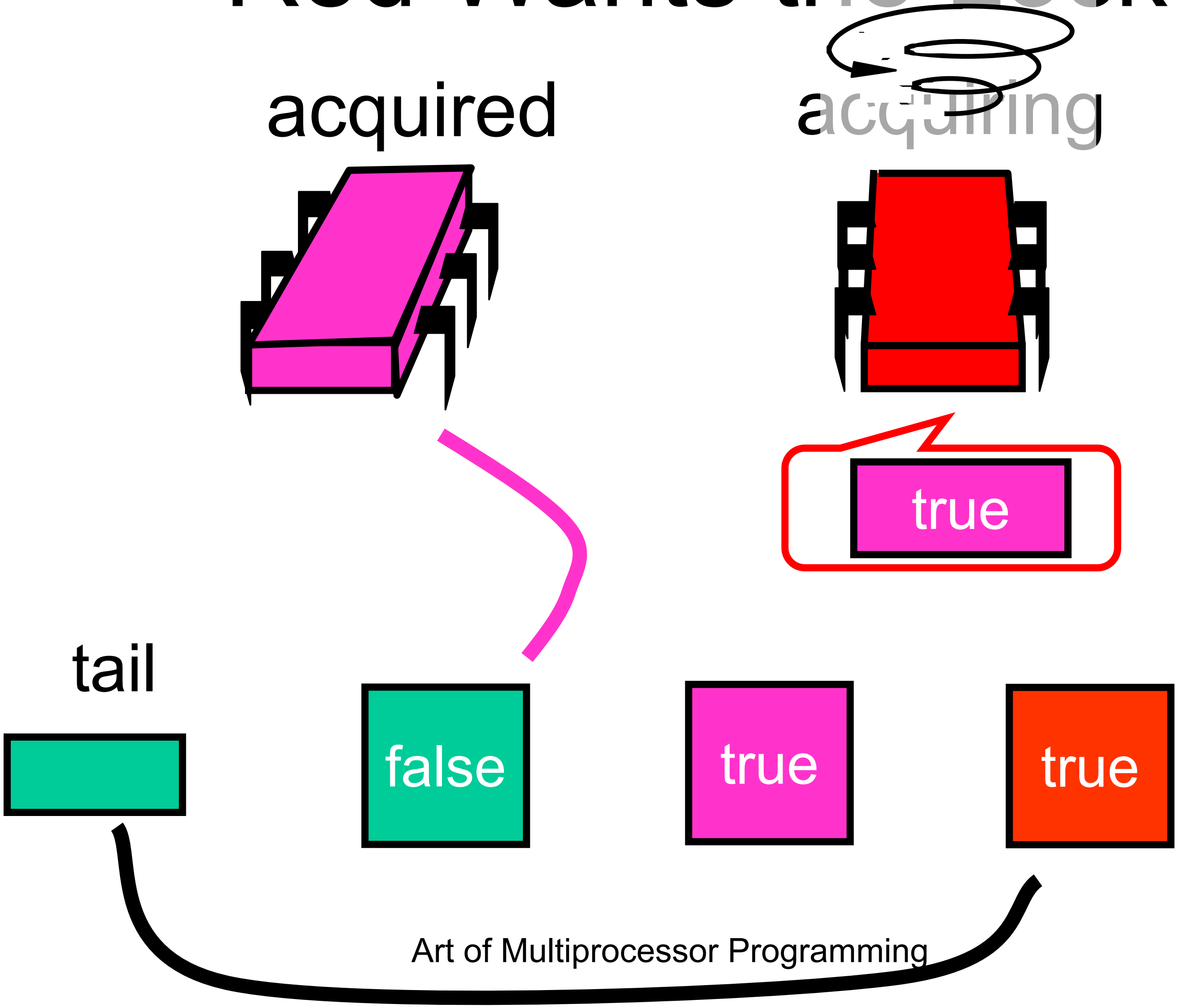
Red Wants the Lock



Red Wants the Lock

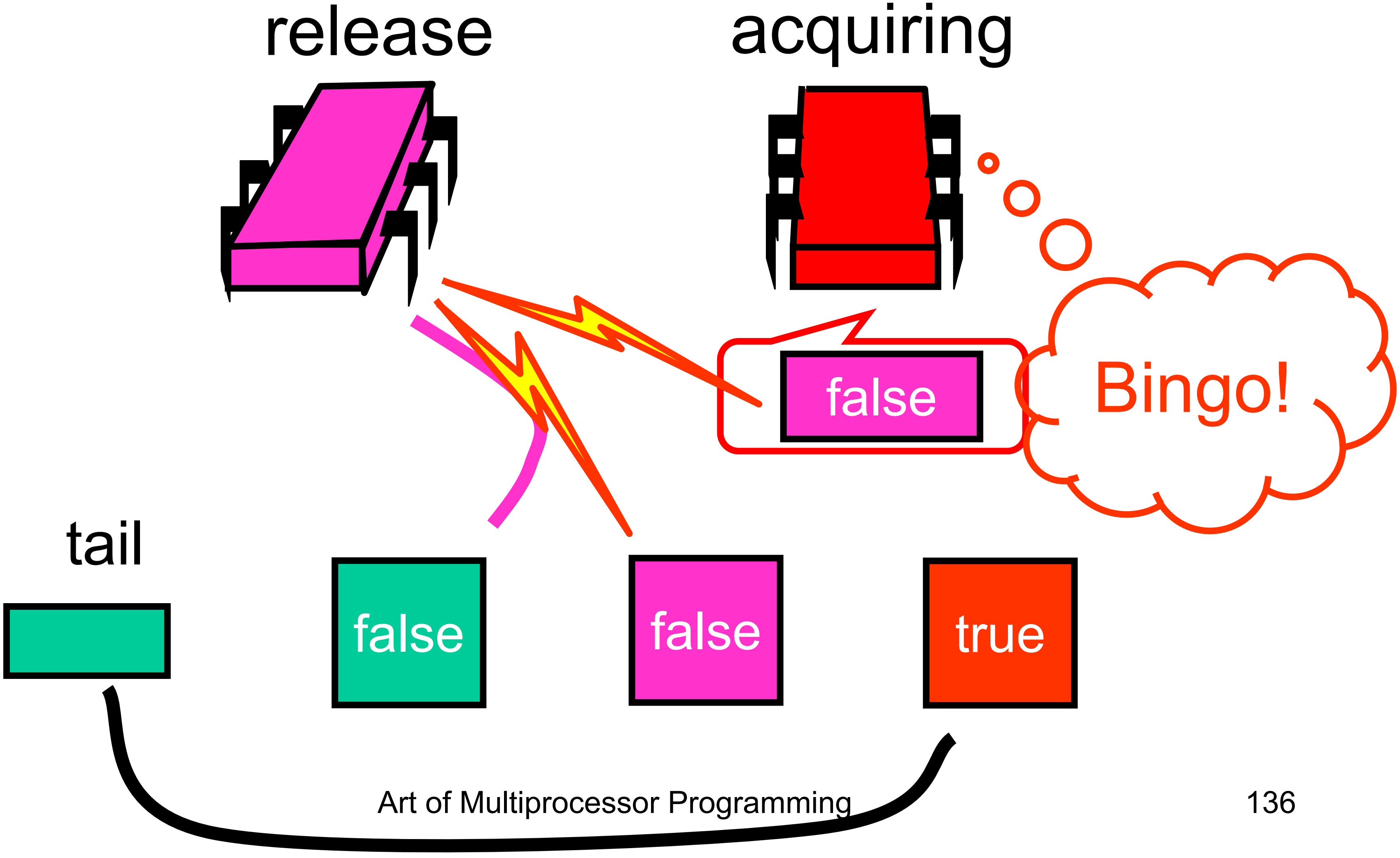


Red Wants the Lock

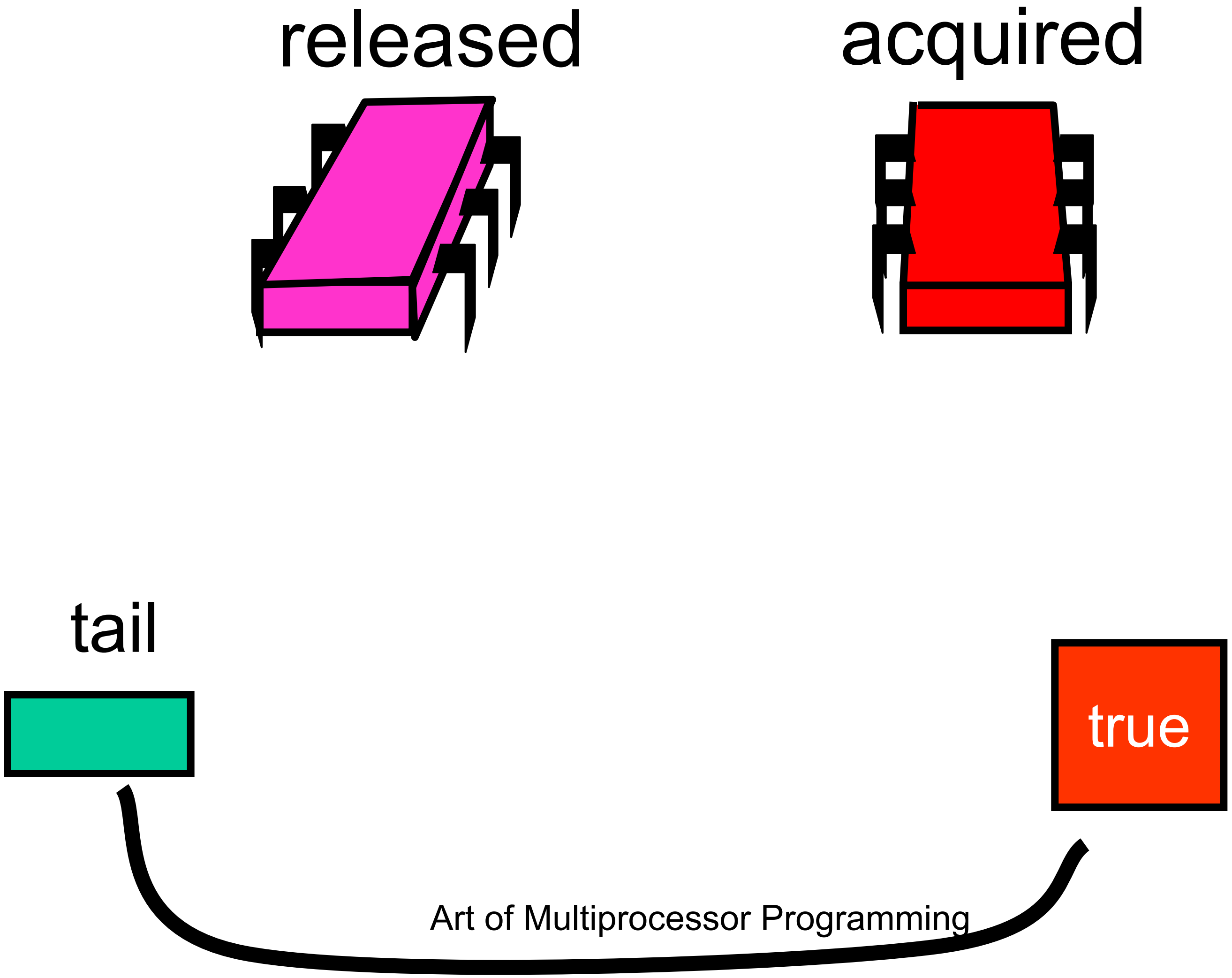


Actually, it spins on cached copy

Purple Releases



Purple Releases



Space Usage

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

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;
 }
}
```

(we don’t actually reuse myNode. Code in book shows how it’s done.)

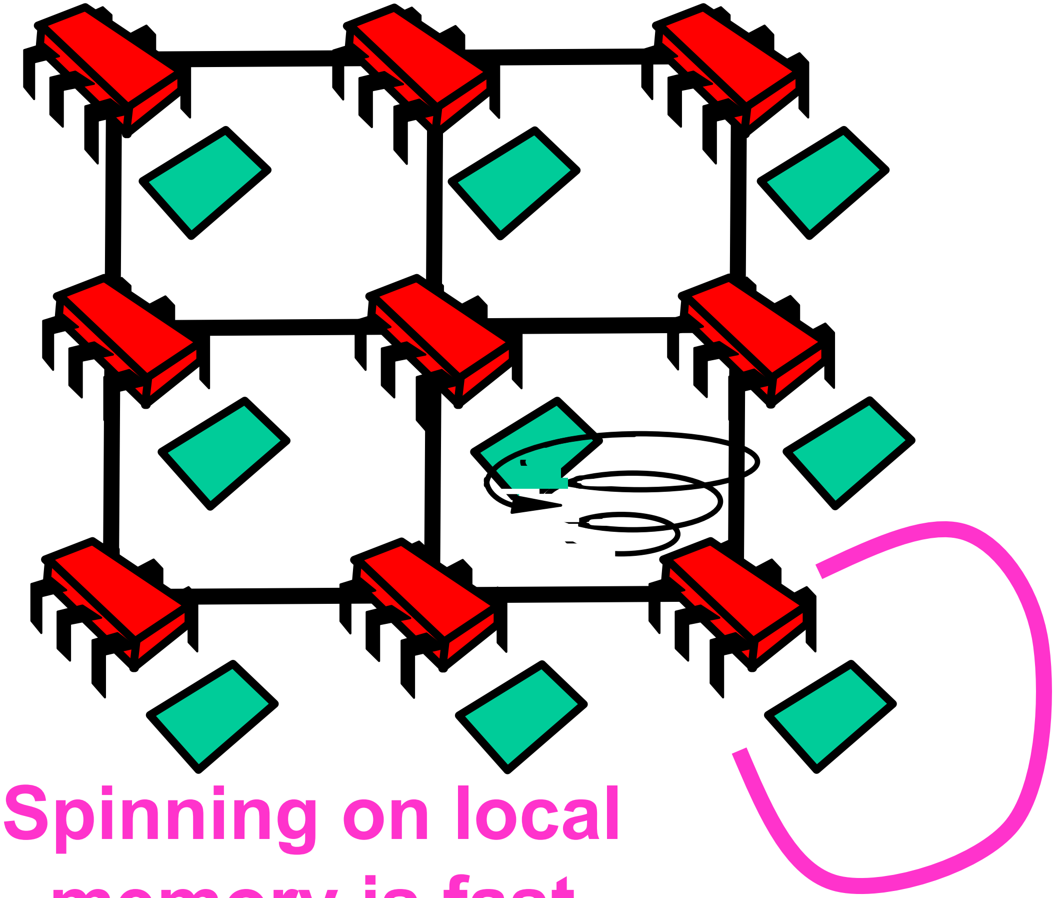
CLH Lock

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

NUMA and cc-NUMA Architectures

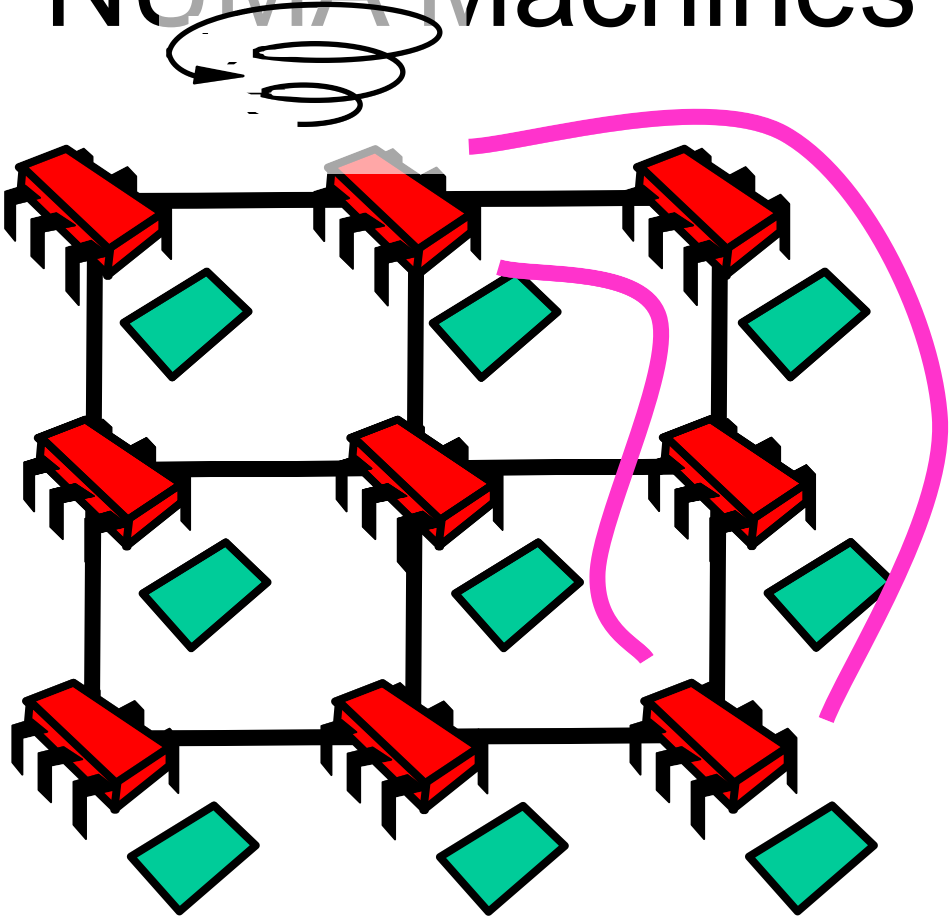
- Acronym:
  - **N**on-**U**niform **M**emory **A**rchitecture
  - ccNUMA = cache coherent NUMA
- Illusion:
  - Flat shared memory
- Truth:
  - No caches (sometimes)
  - Some memory regions faster than others

NUMA Machines



Spinning on local memory is fast

NUMA Machines



Spinning on remote memory is slow

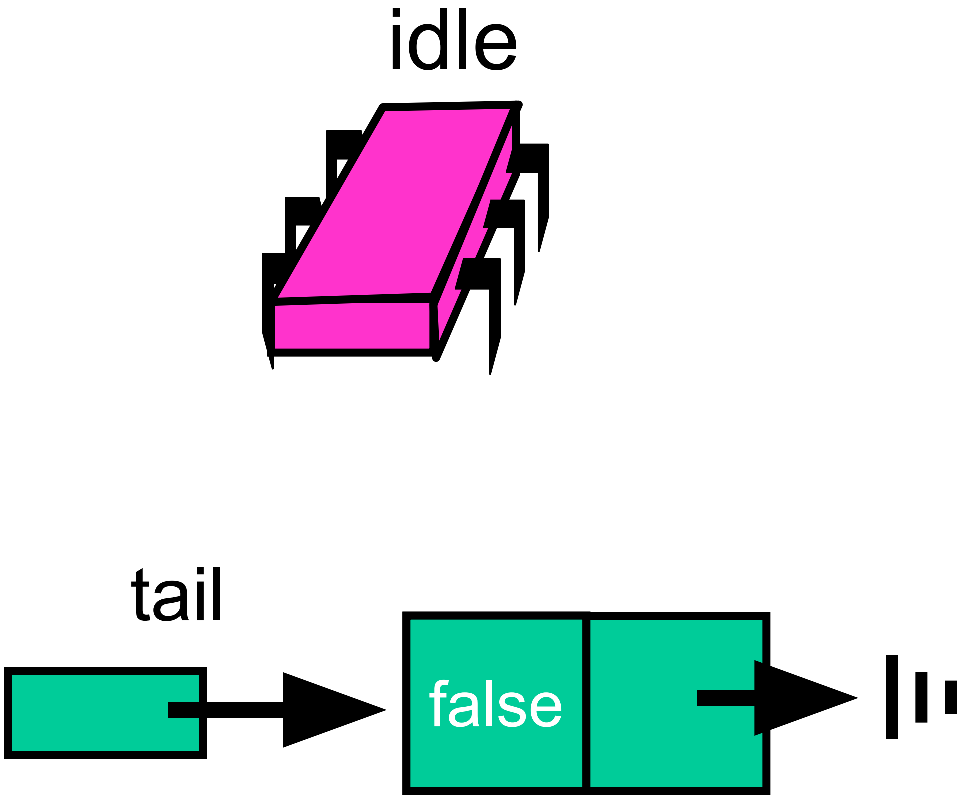
CLH Lock

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

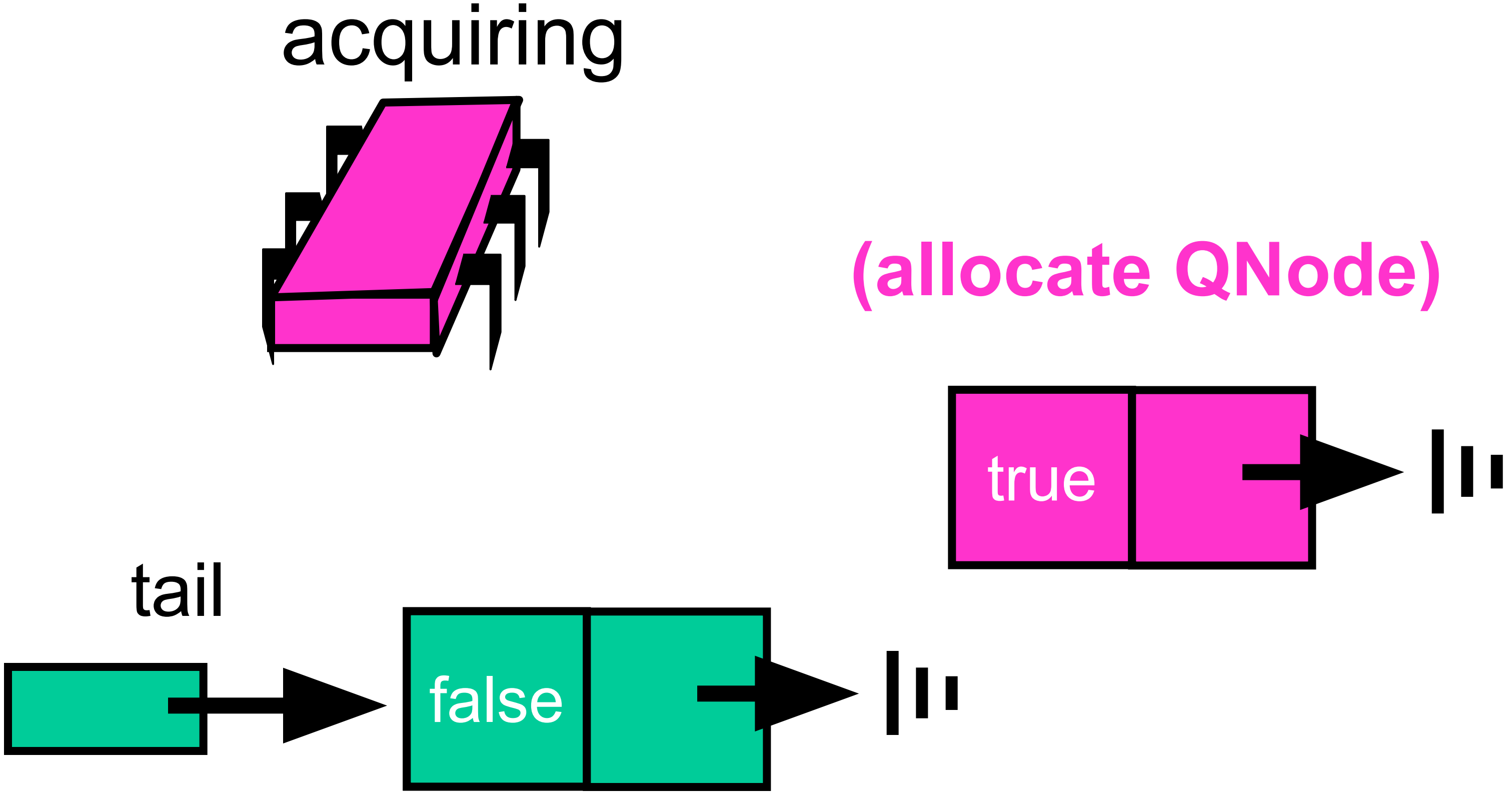
MCS Lock

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

Initially



Acquiring



acquiring

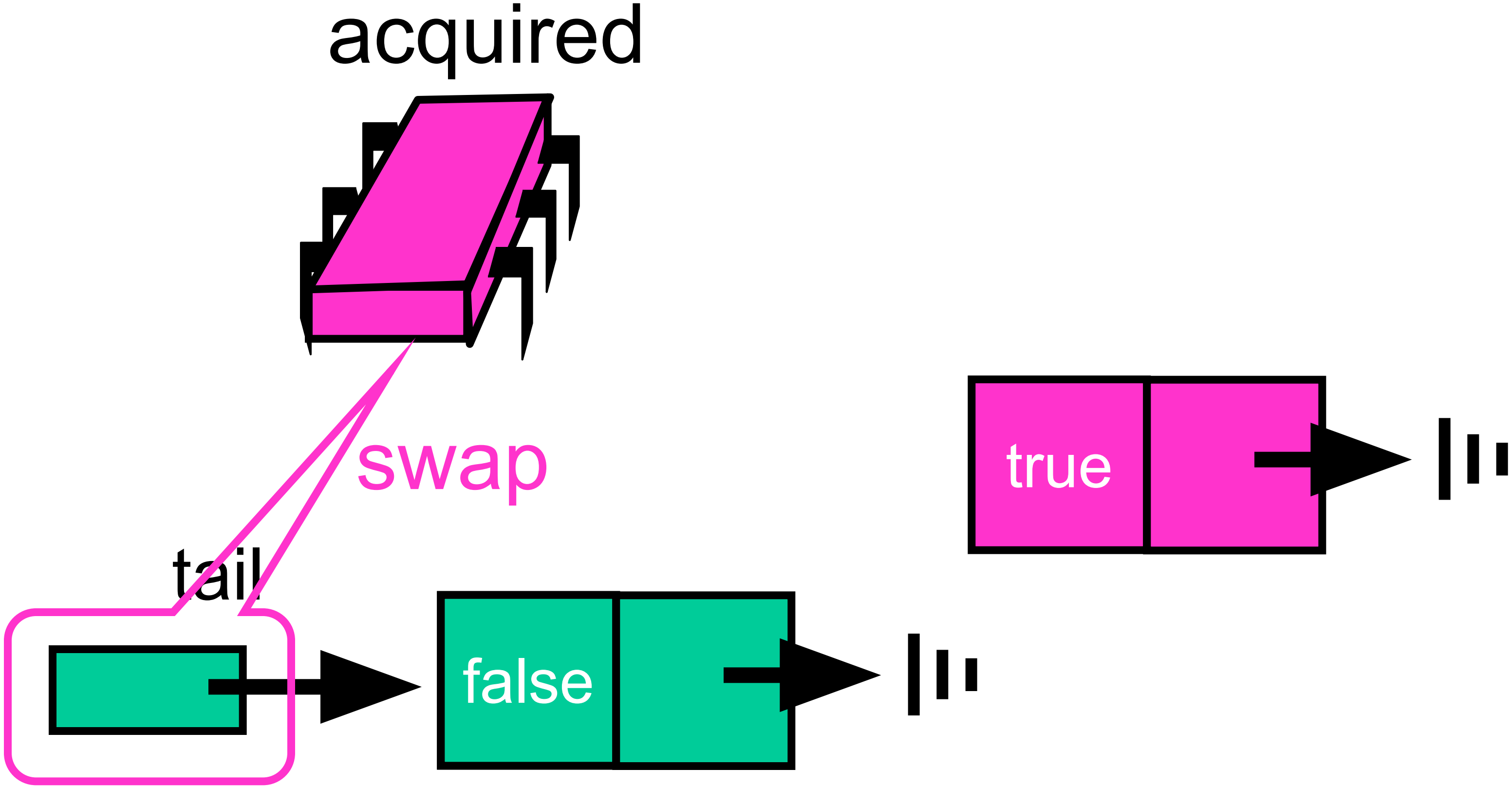
(allocate QNode)

true

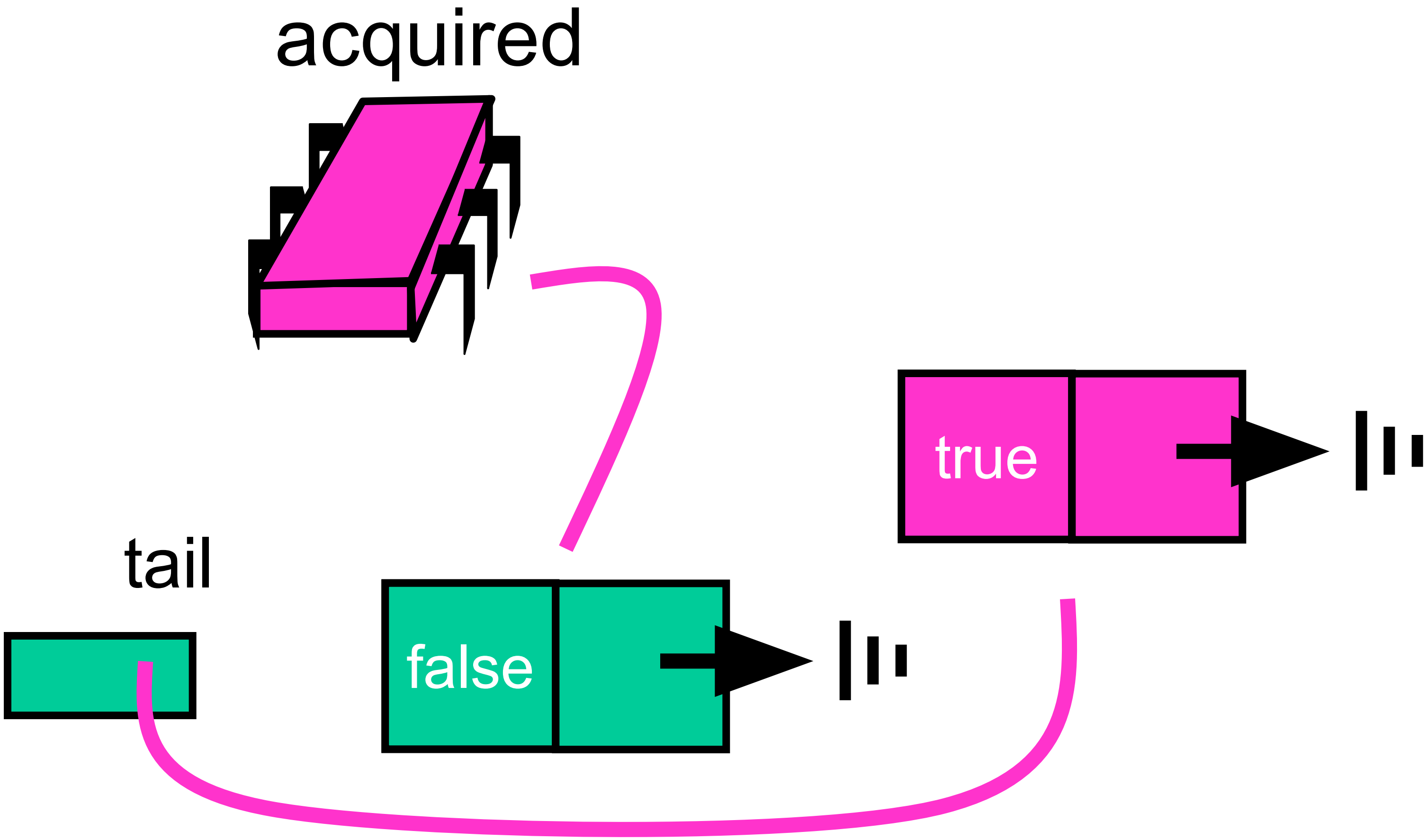
tail

false

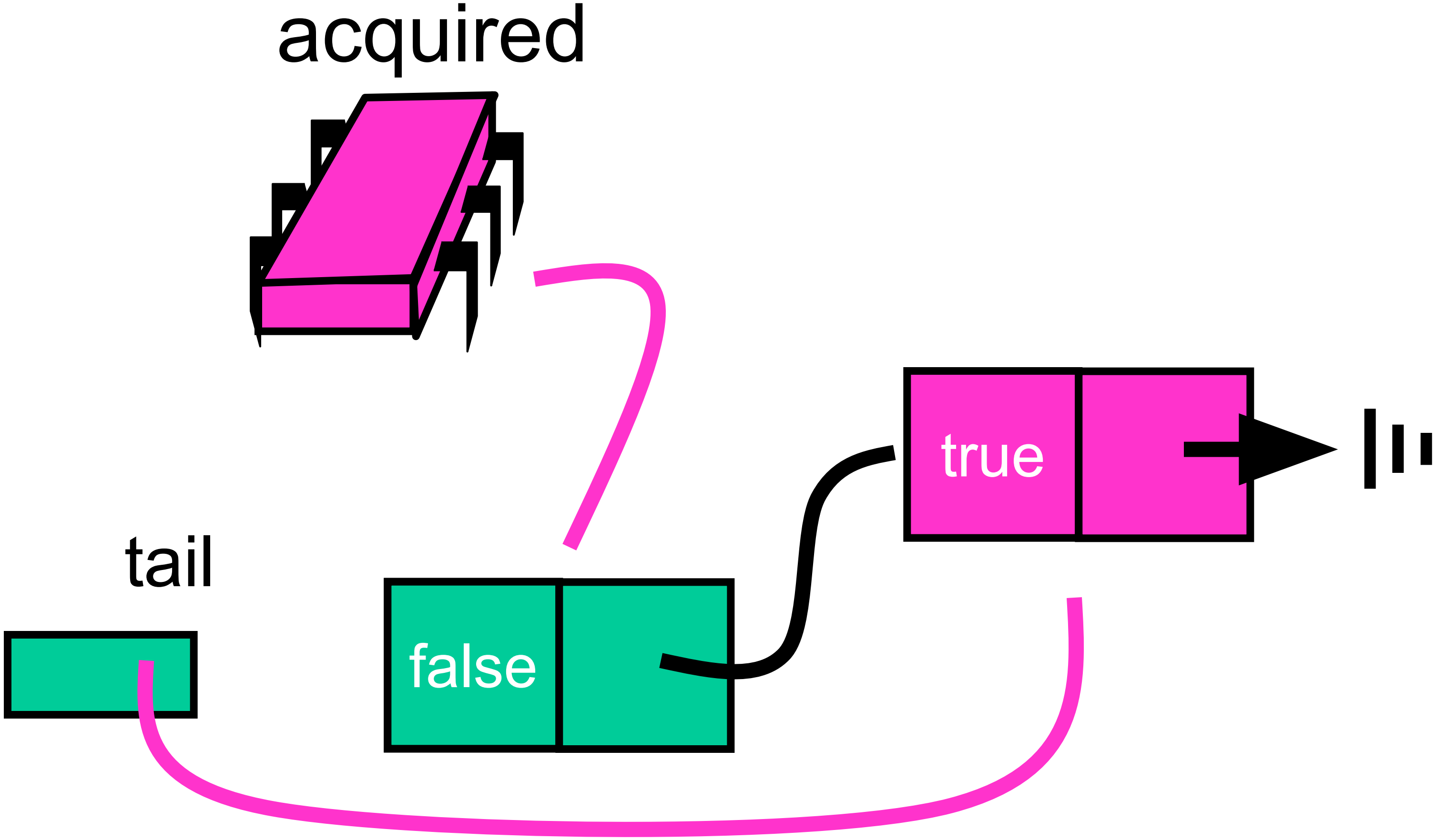
Acquiring



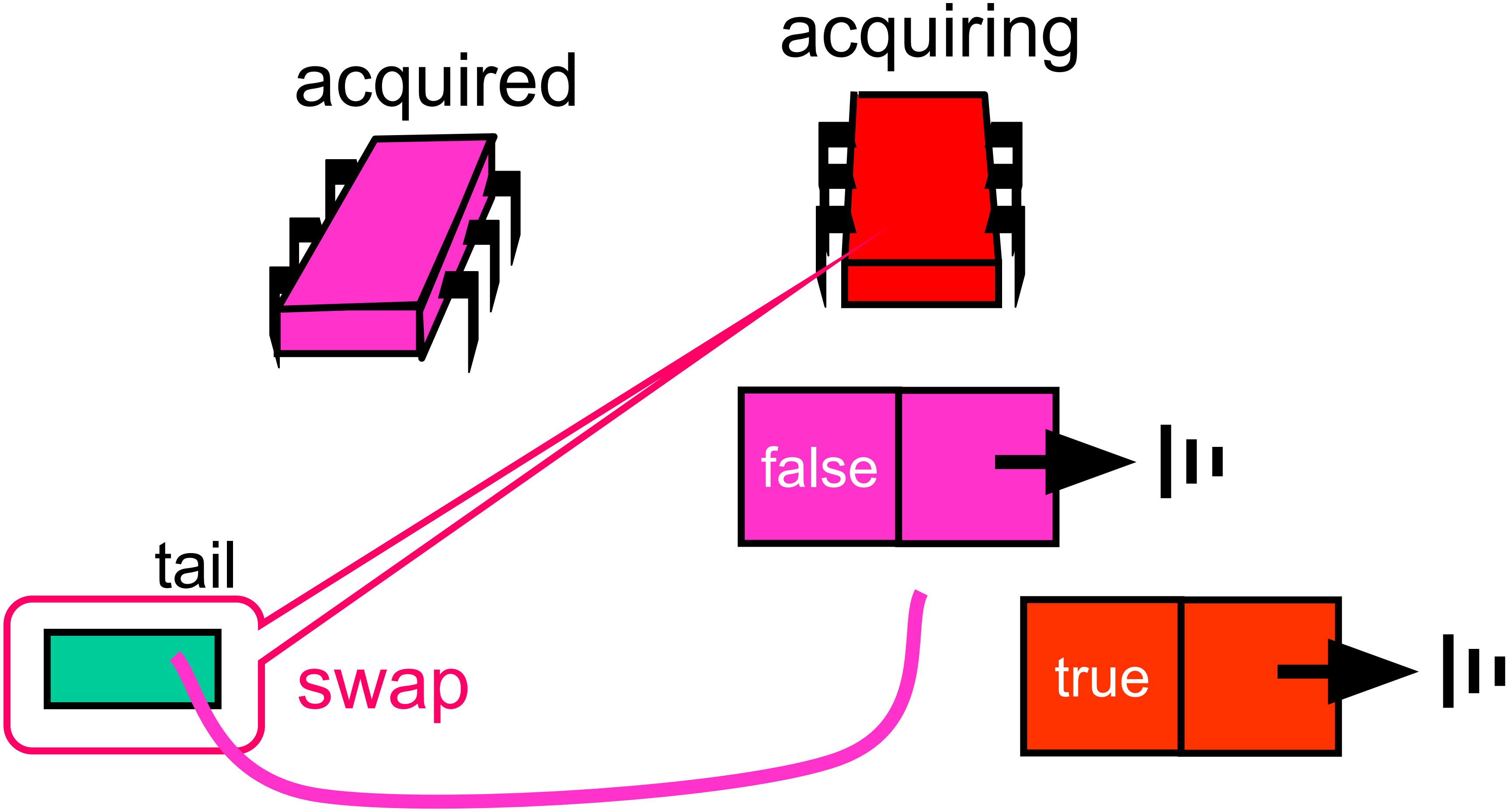
Acquiring



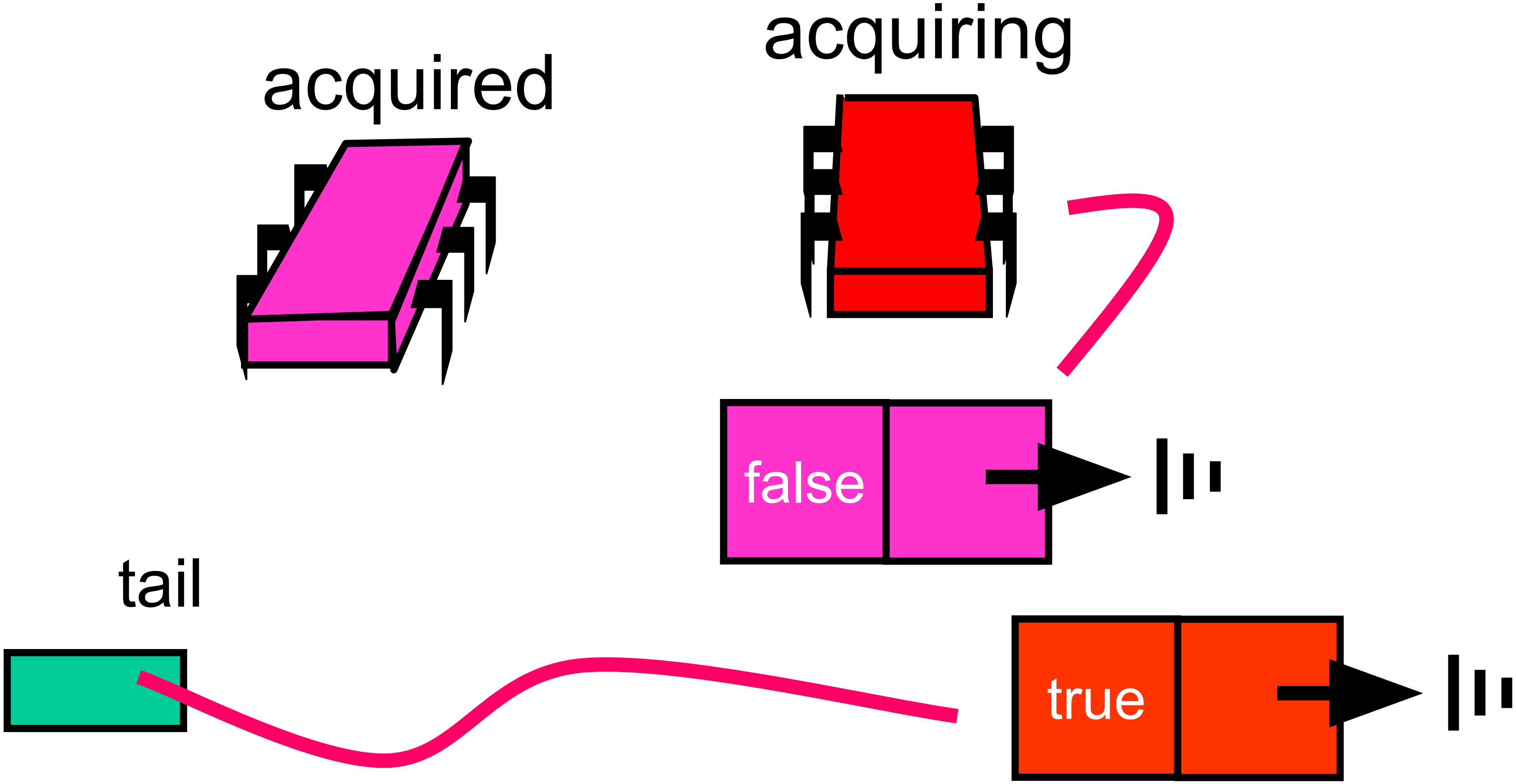
Acquired



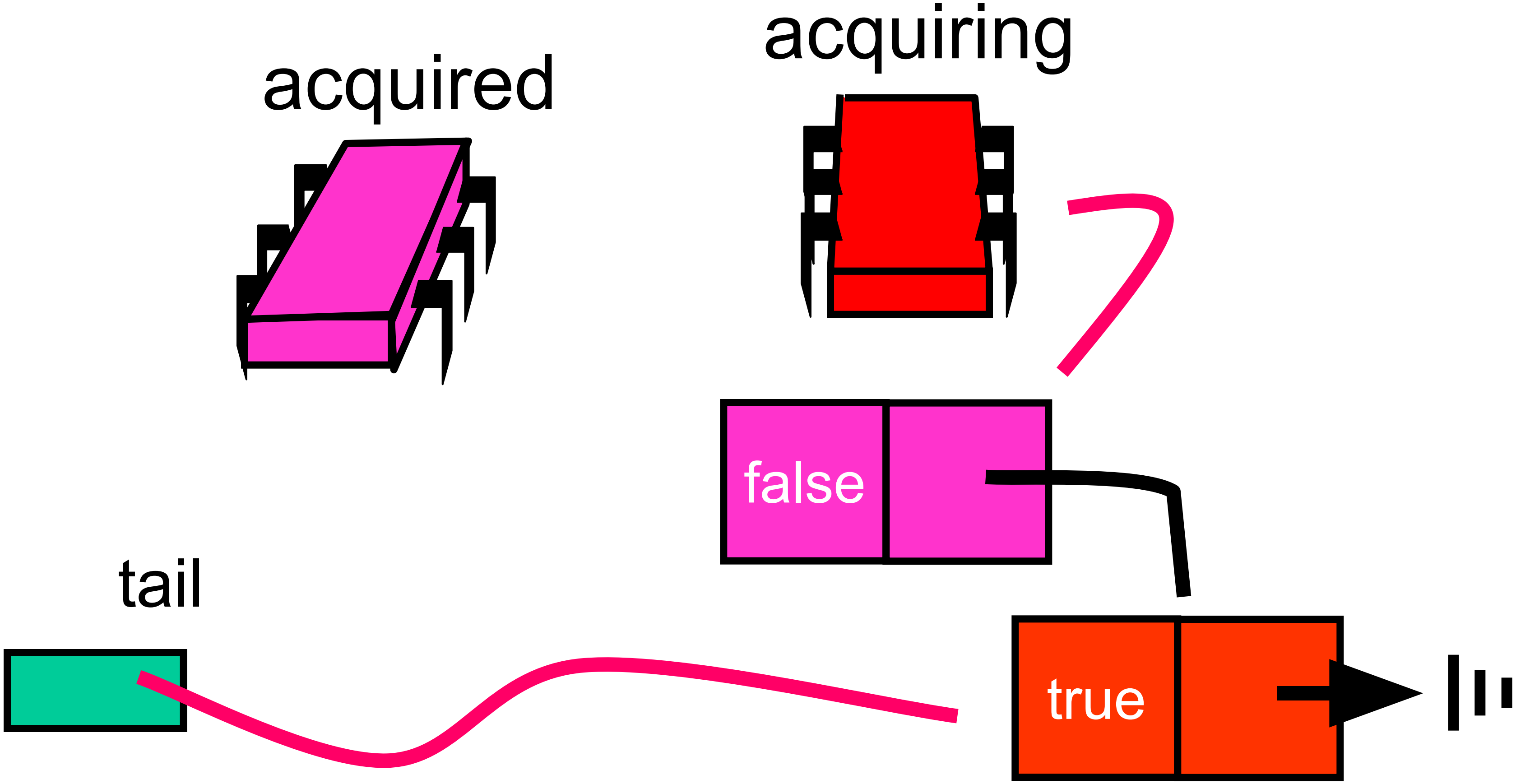
Acquiring



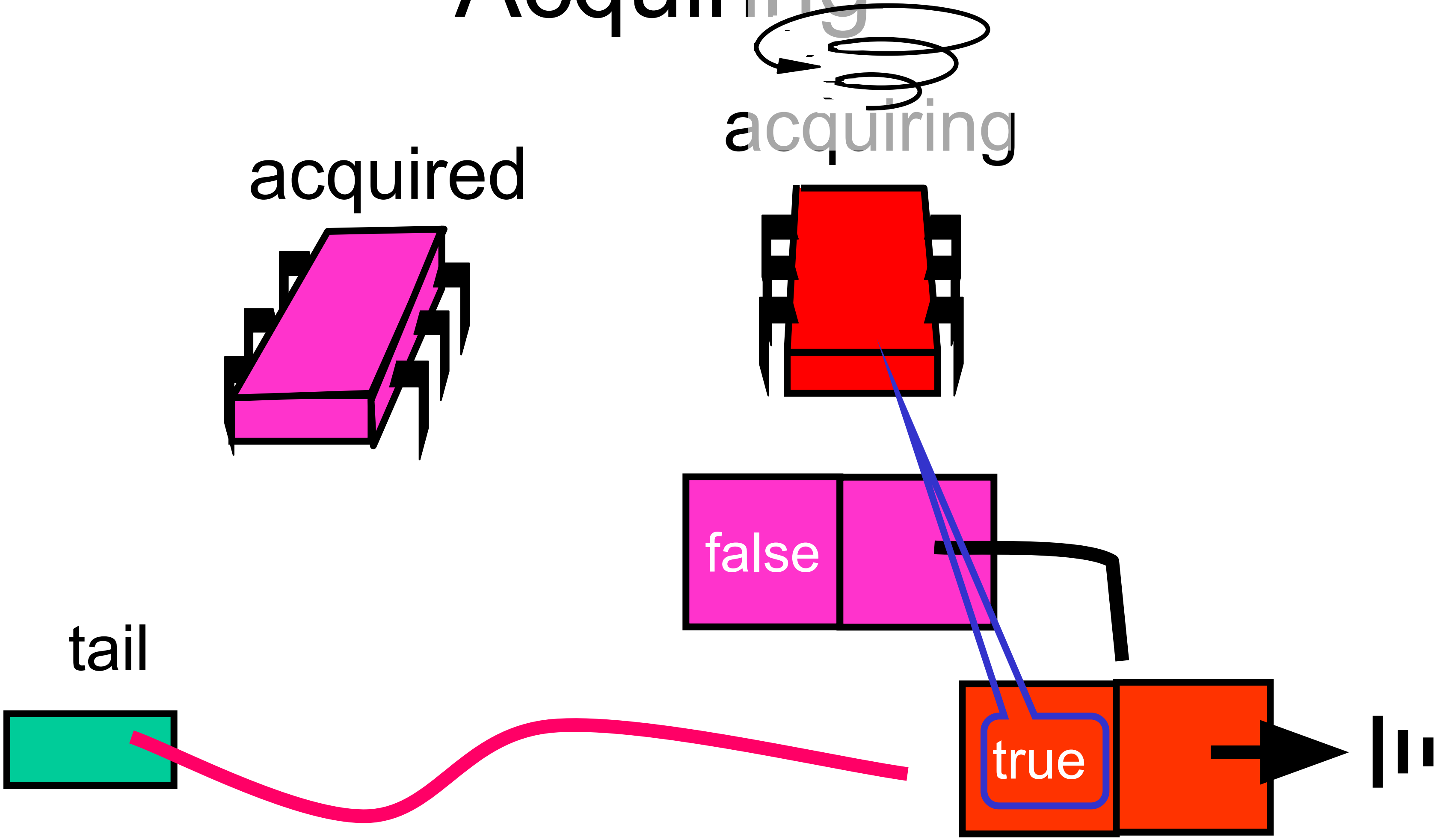
Acquiring



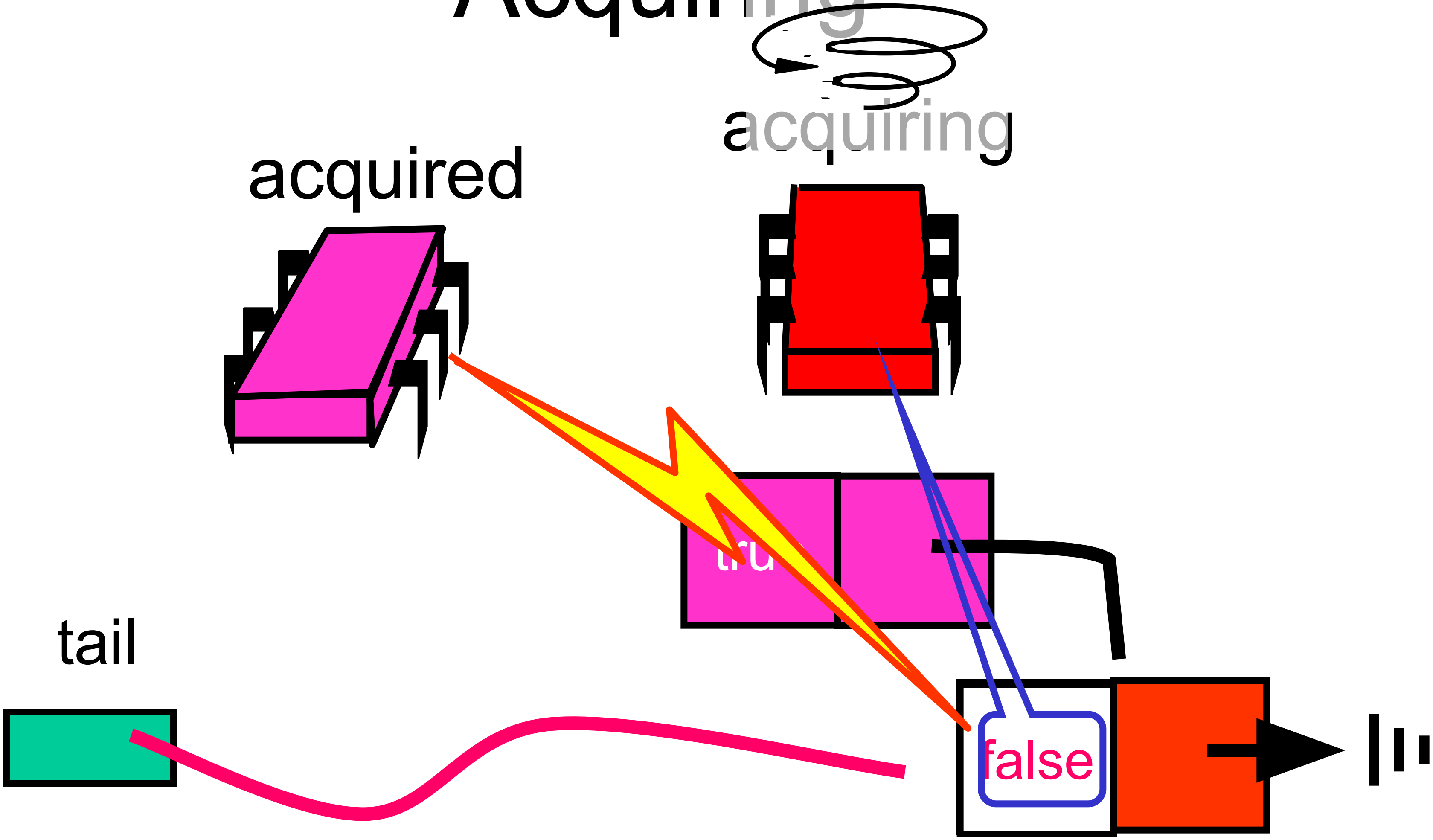
Acquiring



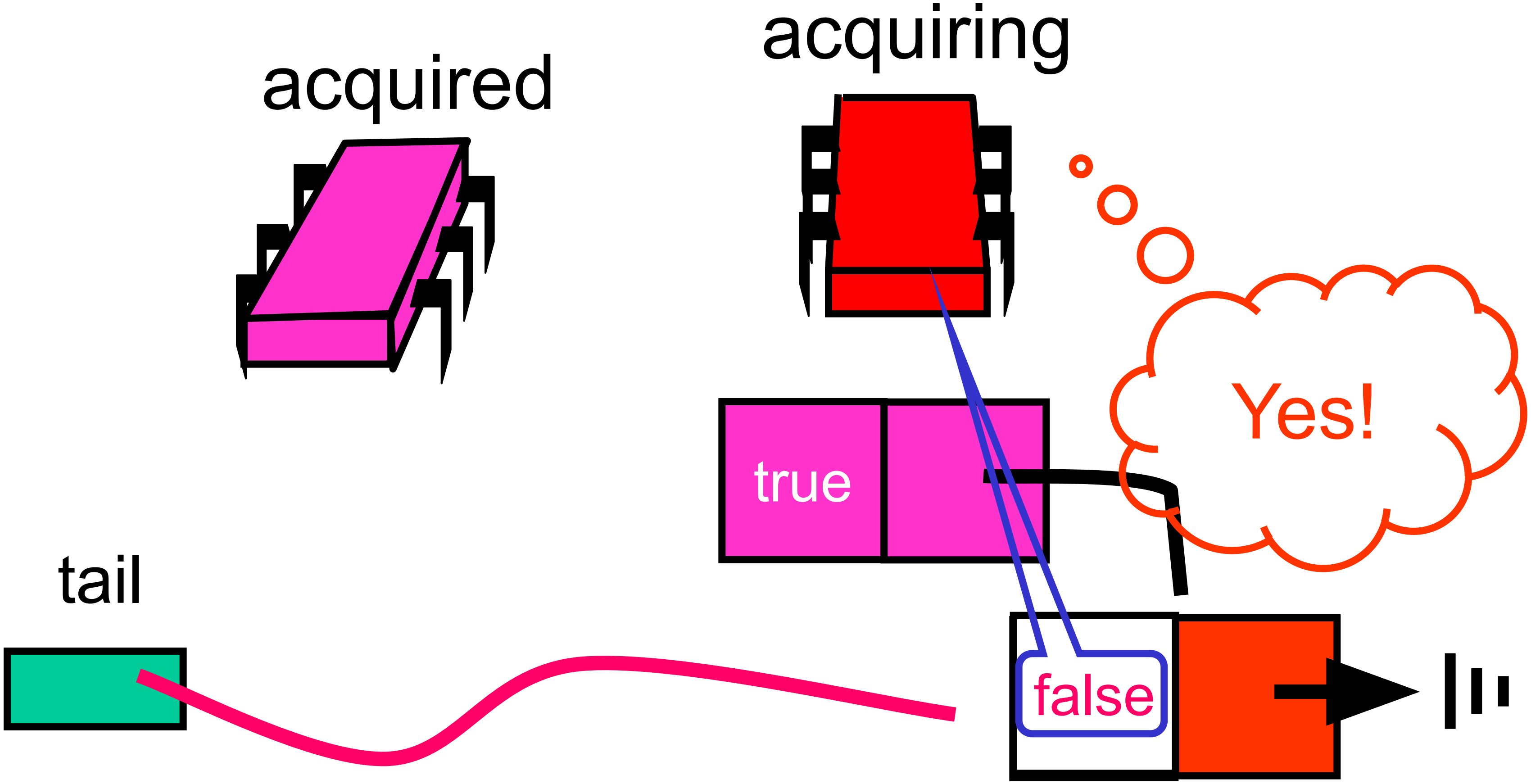
Acquiring



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) {}
  }}}
```

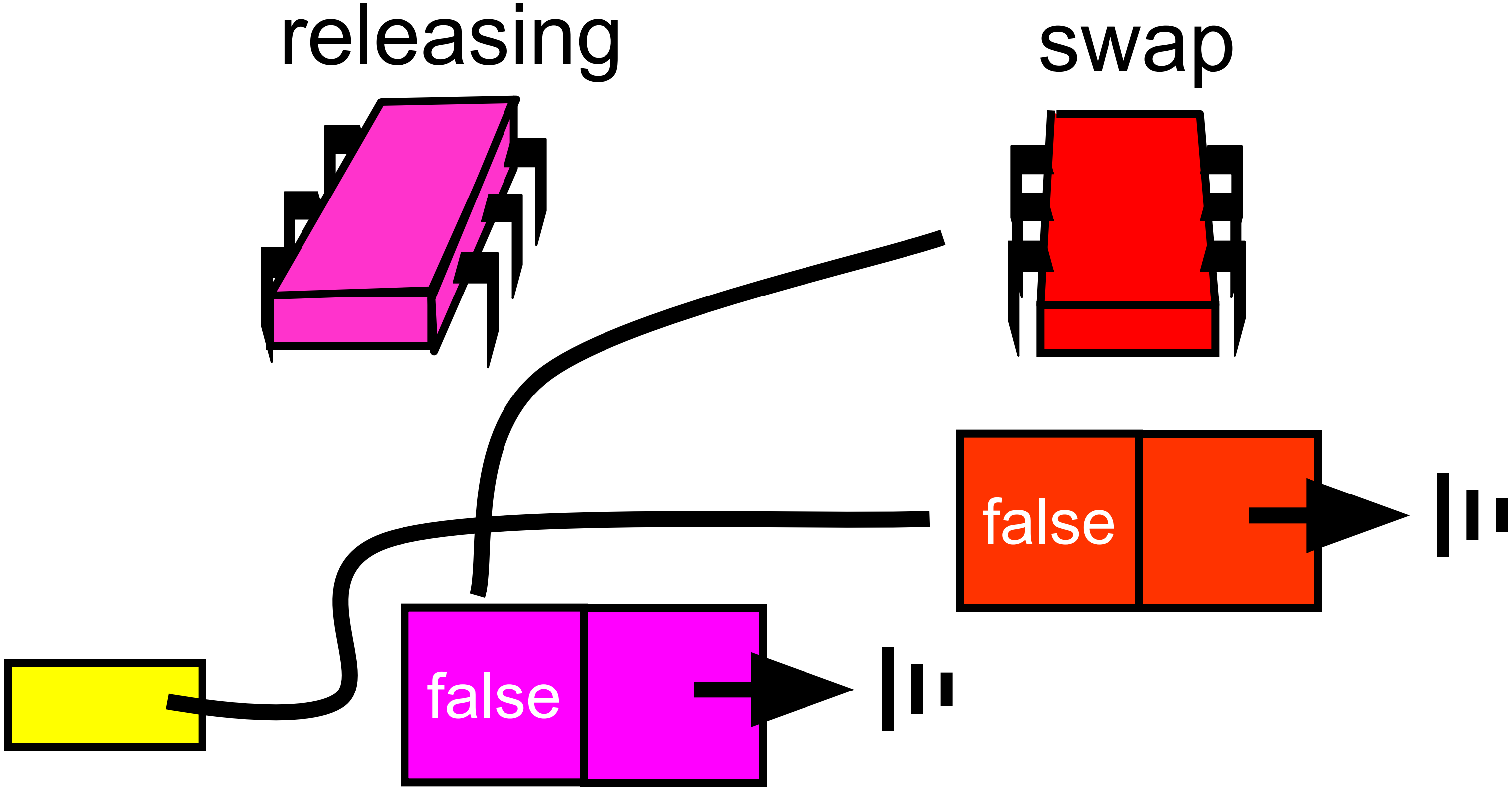
Fix if queue was non-empty

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) {}
  }}}
```

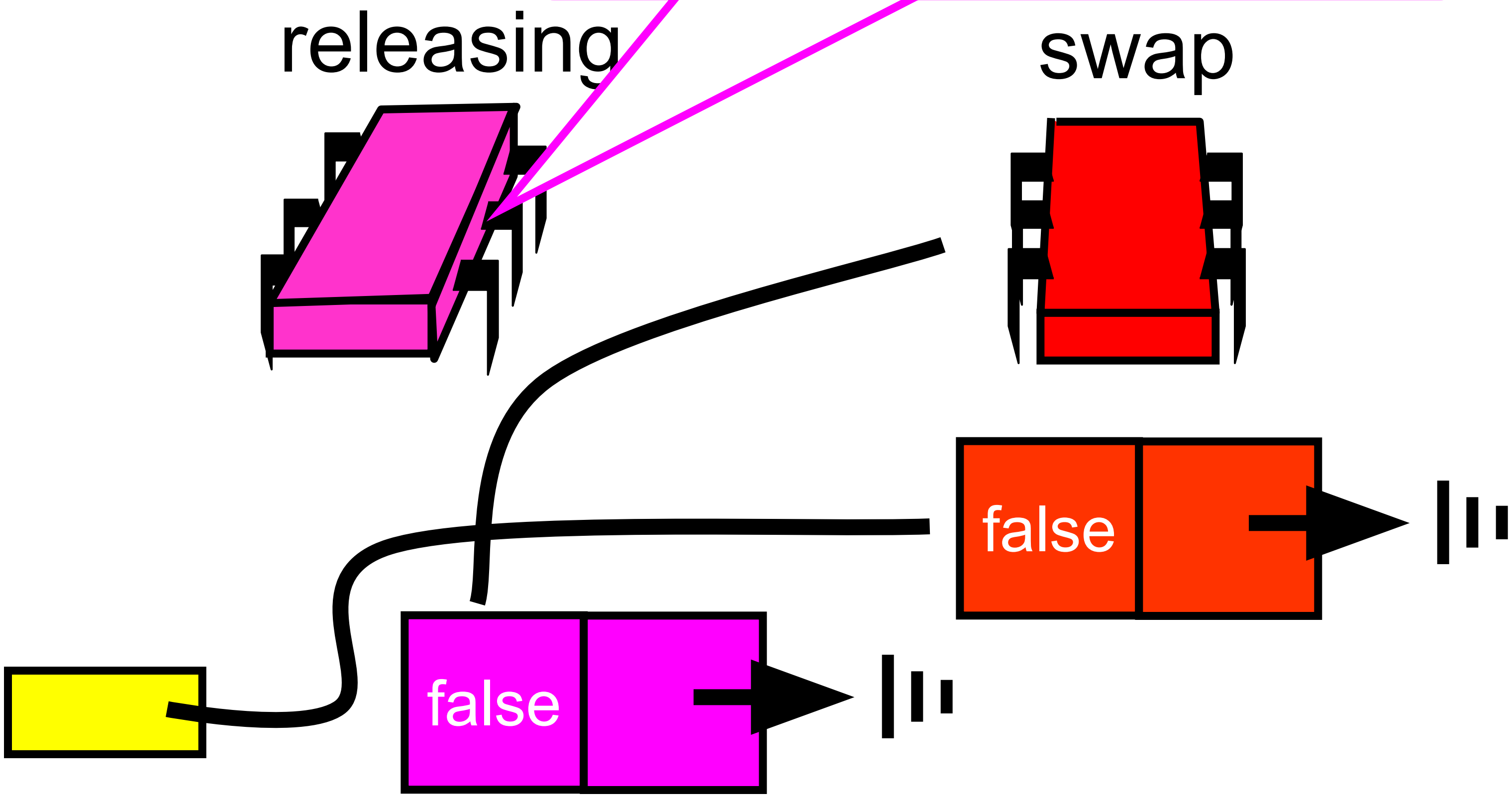
Wait until unlocked

Purple Release



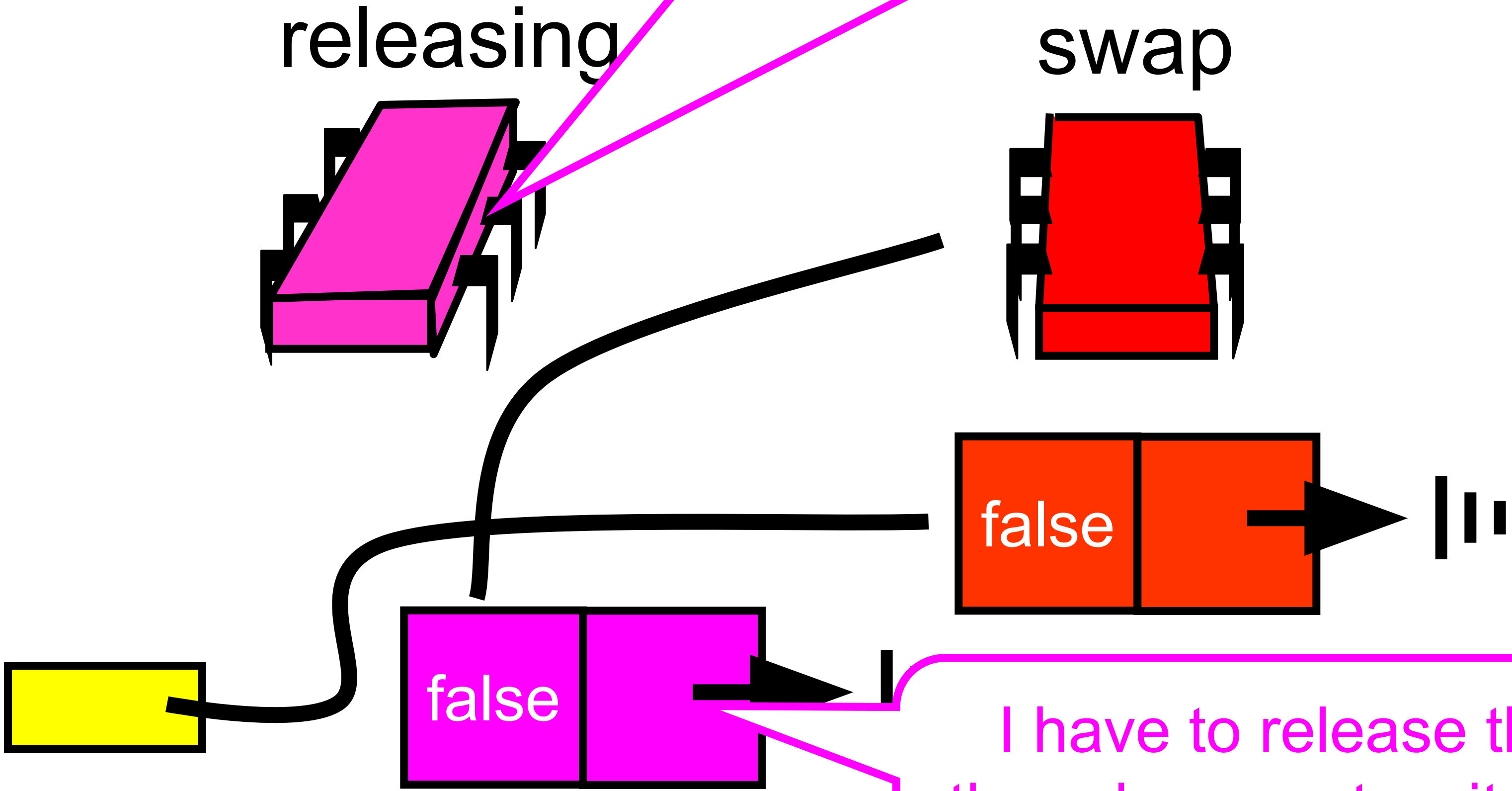
Pu

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



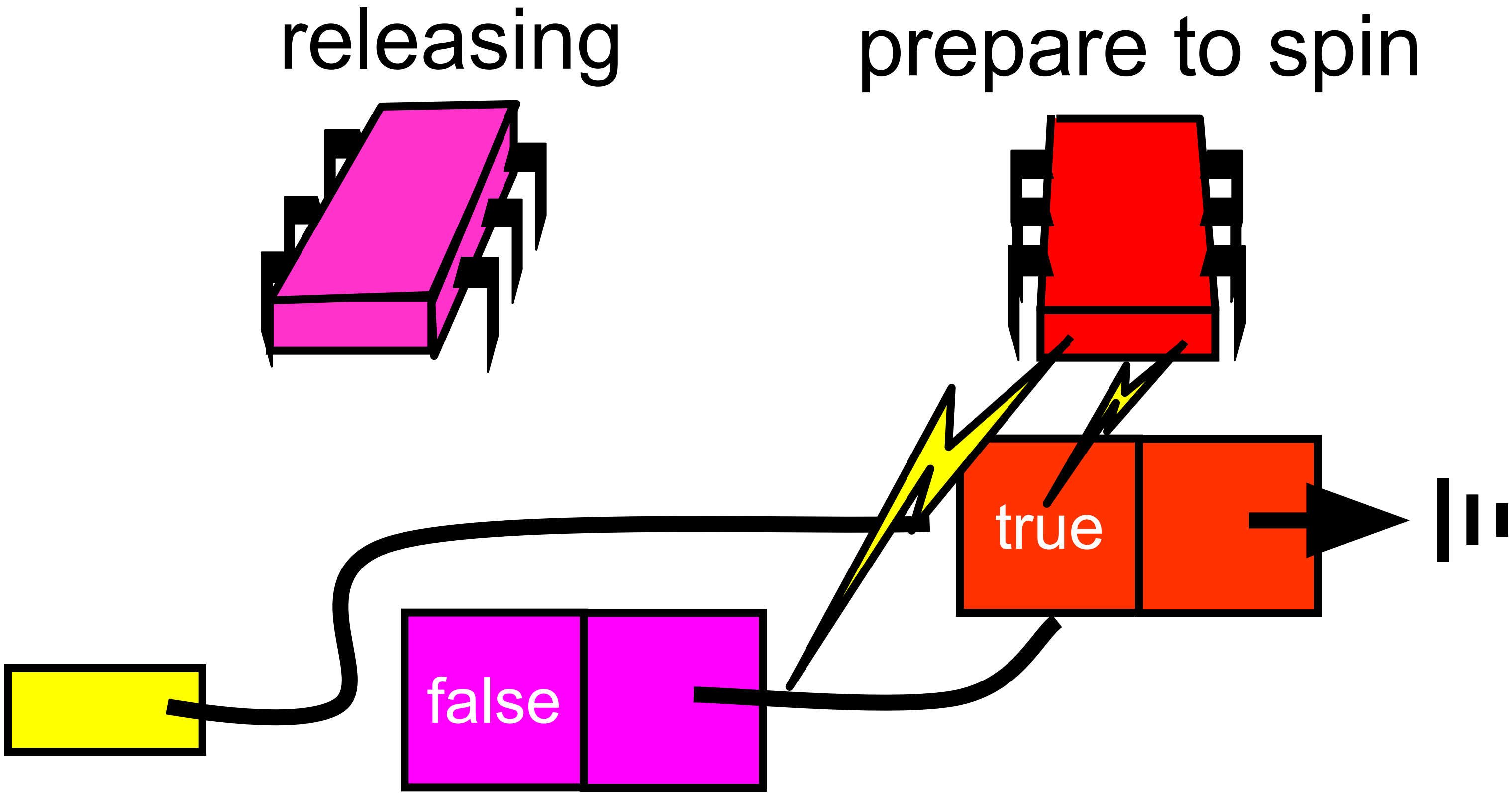
Pu

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



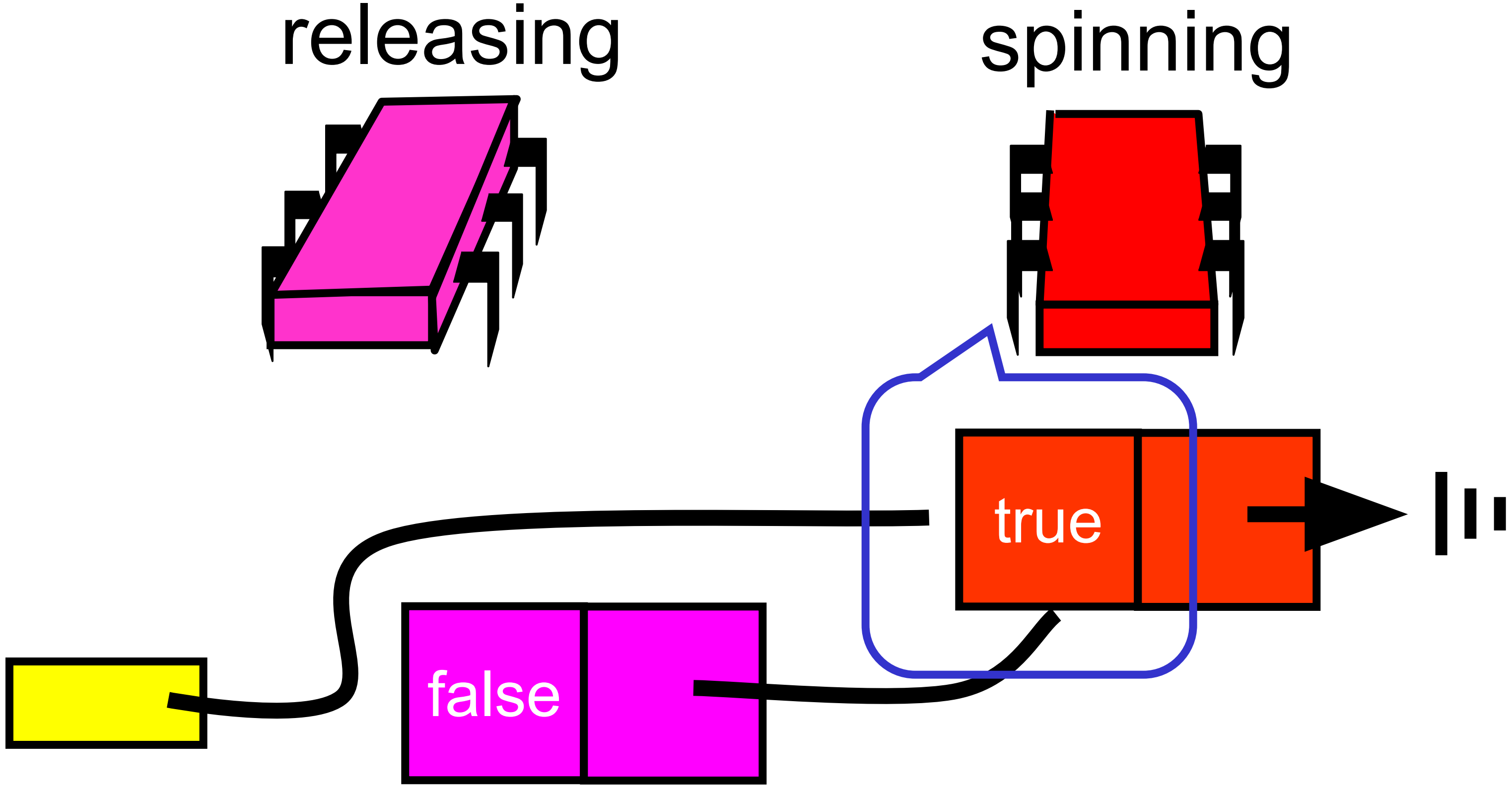
I have to release that thread so must wait for it to identify its node

Purple Release

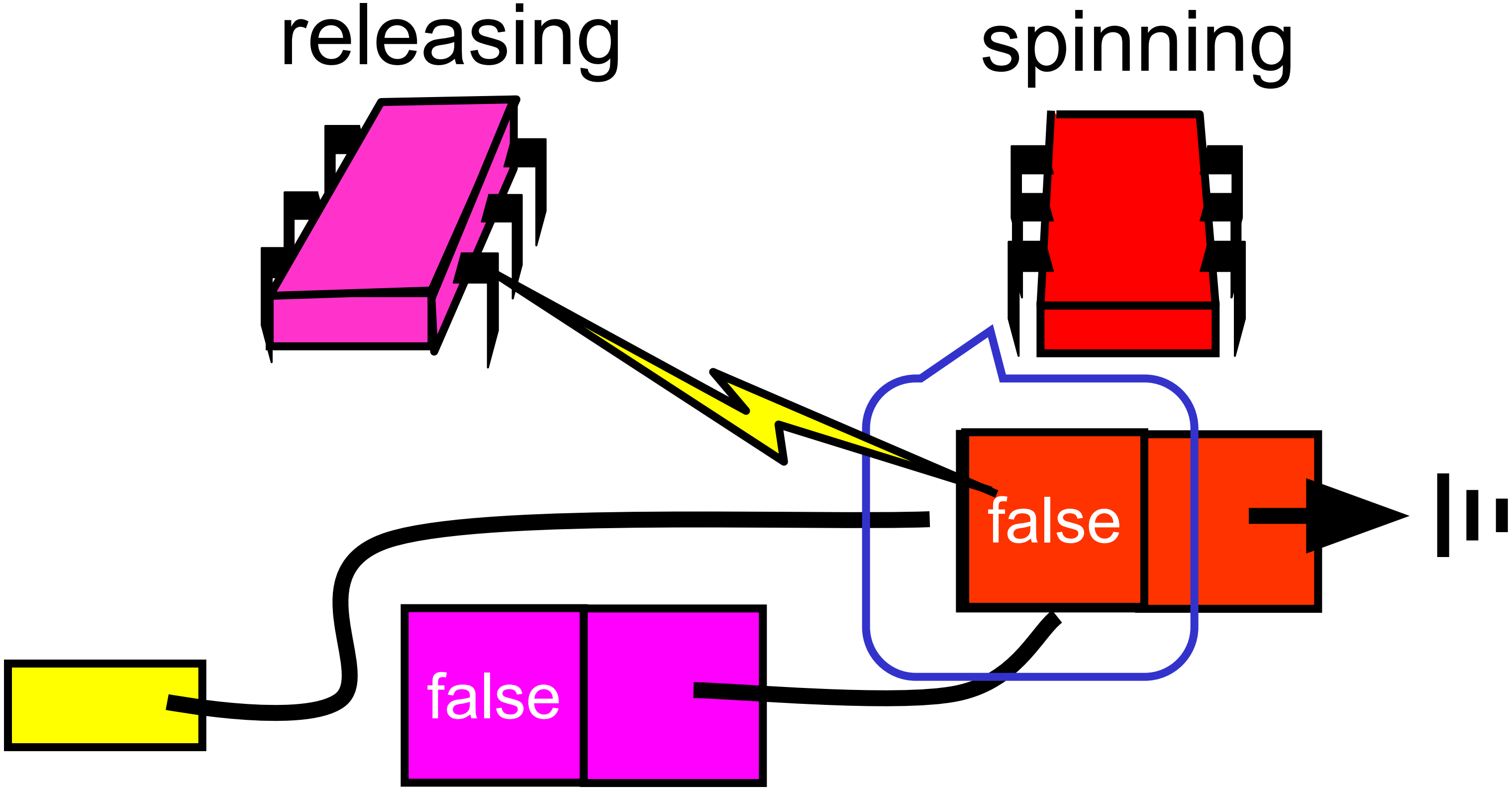

releasing

prepare to spin

Purple Release



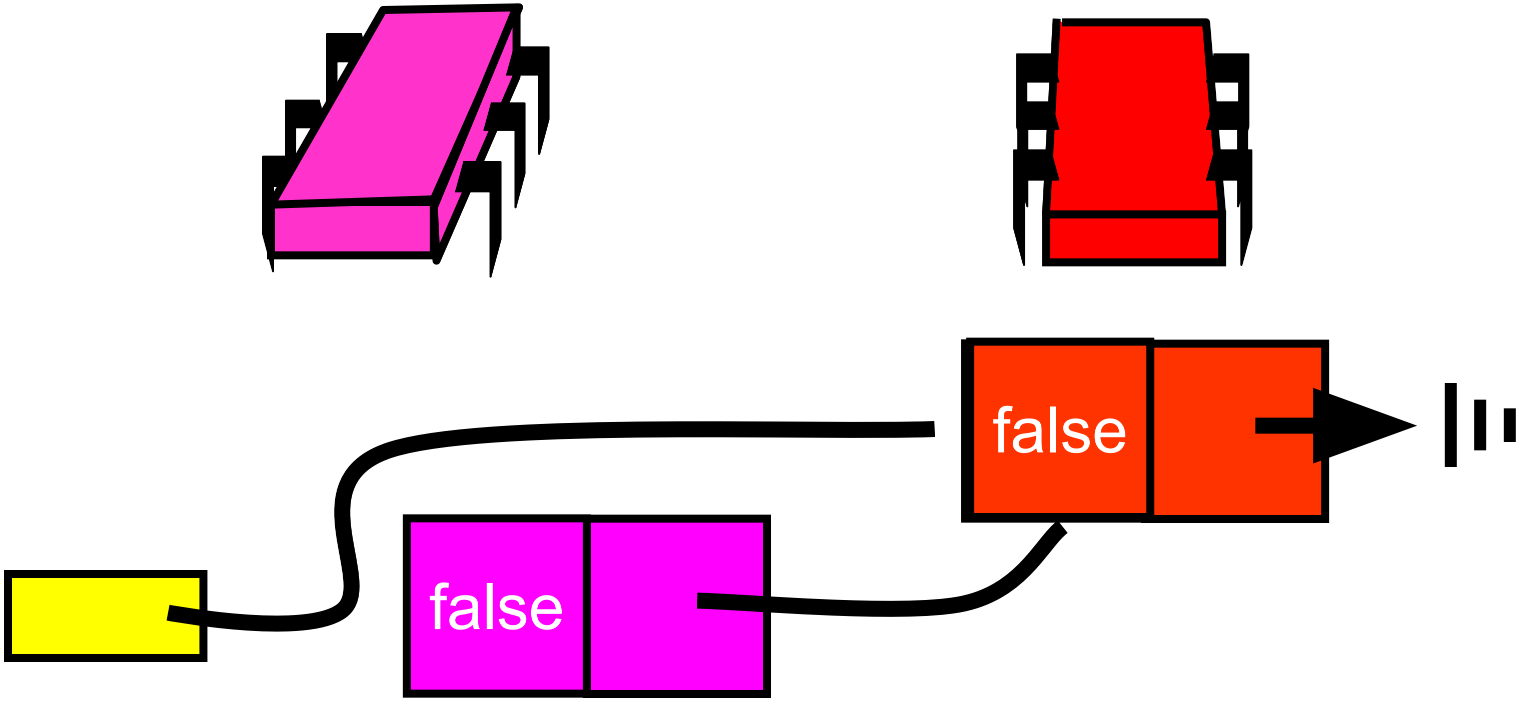
Purple Release



Purple Release

releasing

Acquired lock



false

false

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

Pass lock to successor

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

Abortable Locks

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

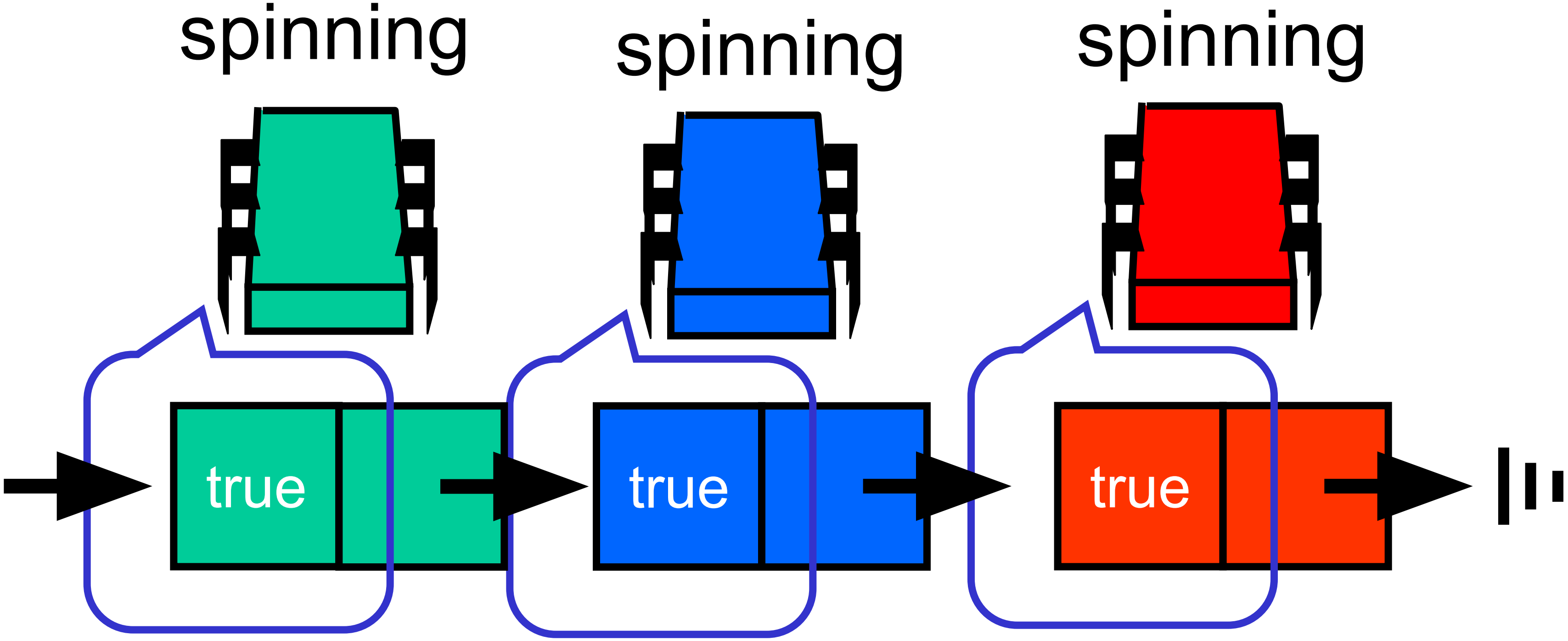
Back-off Lock

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

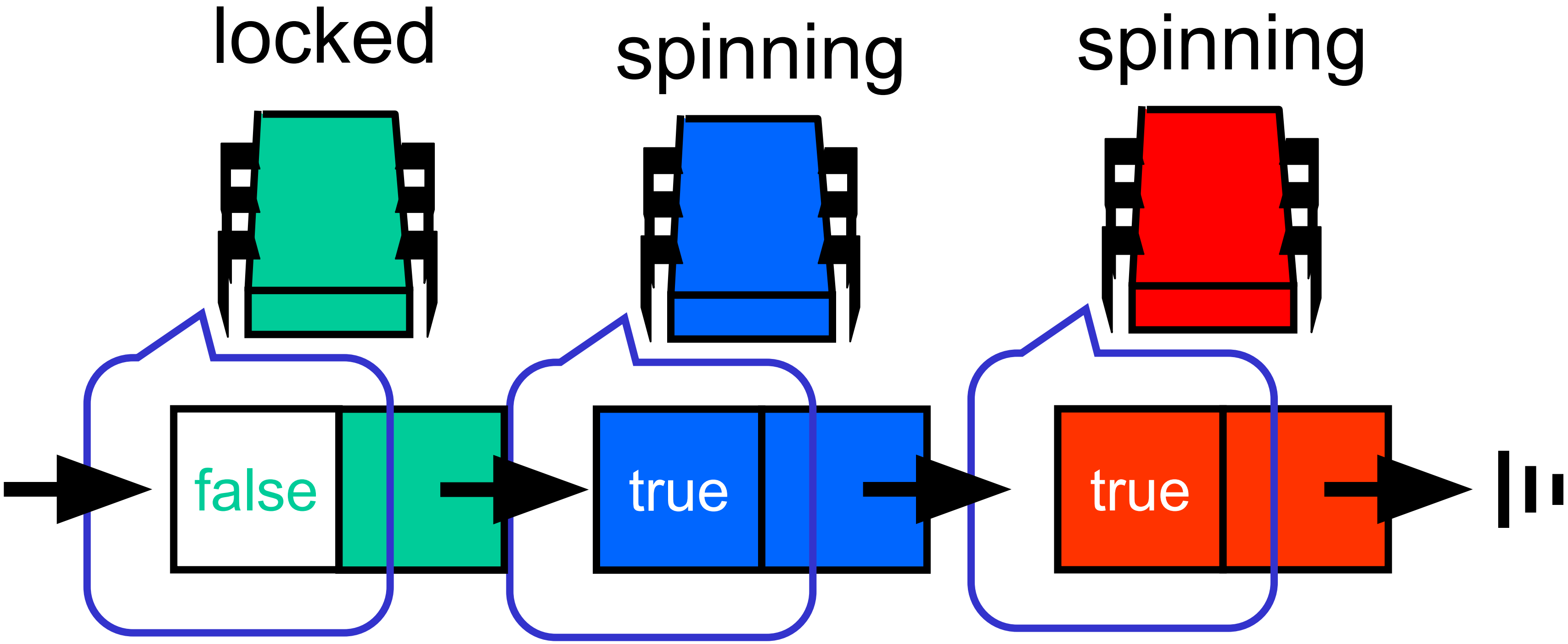
Queue Locks

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

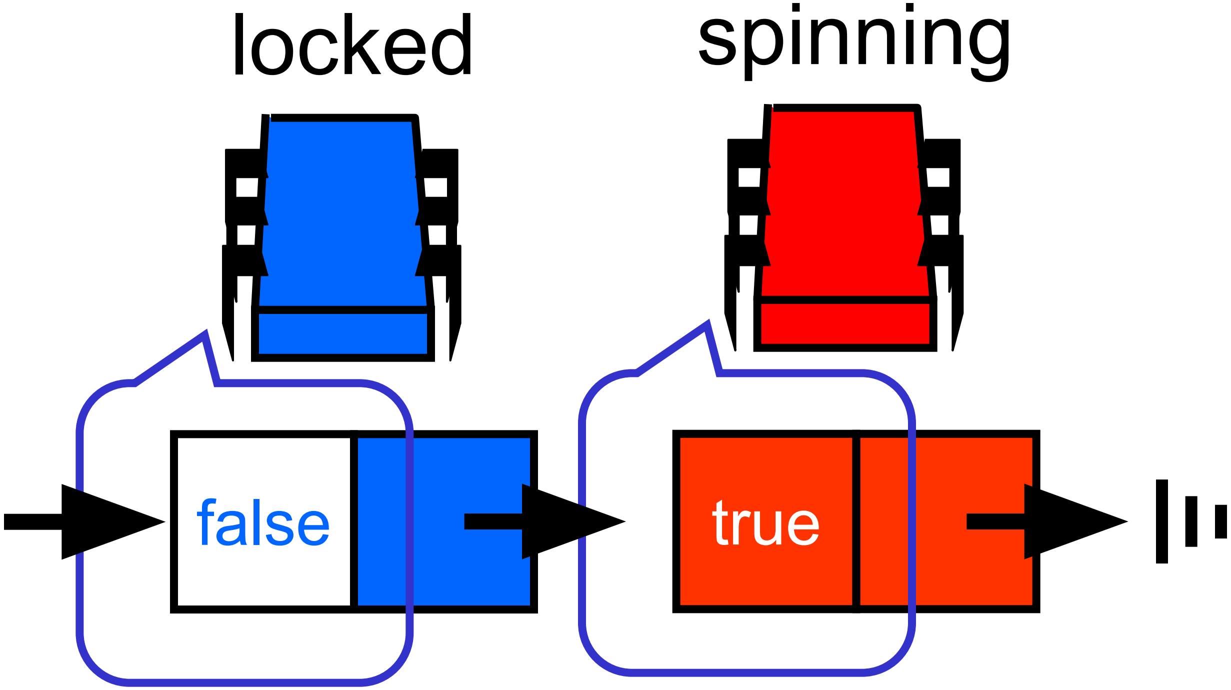
Queue Locks



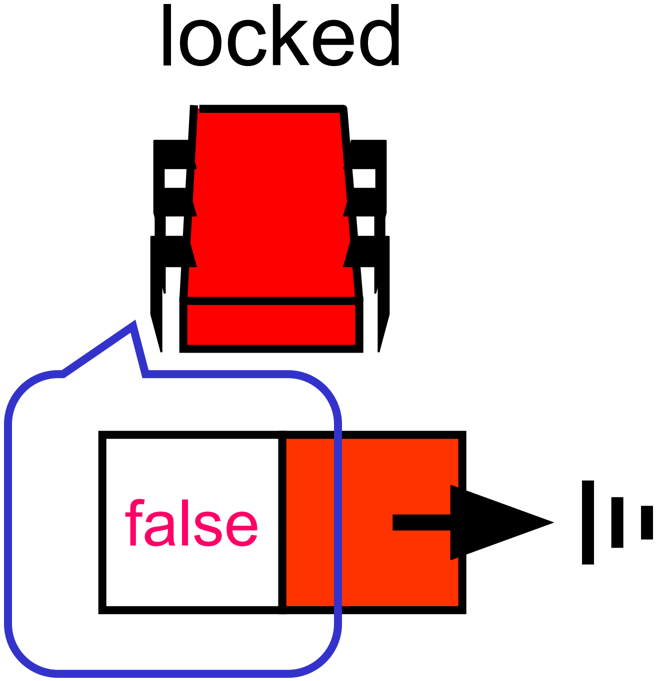
Queue Locks



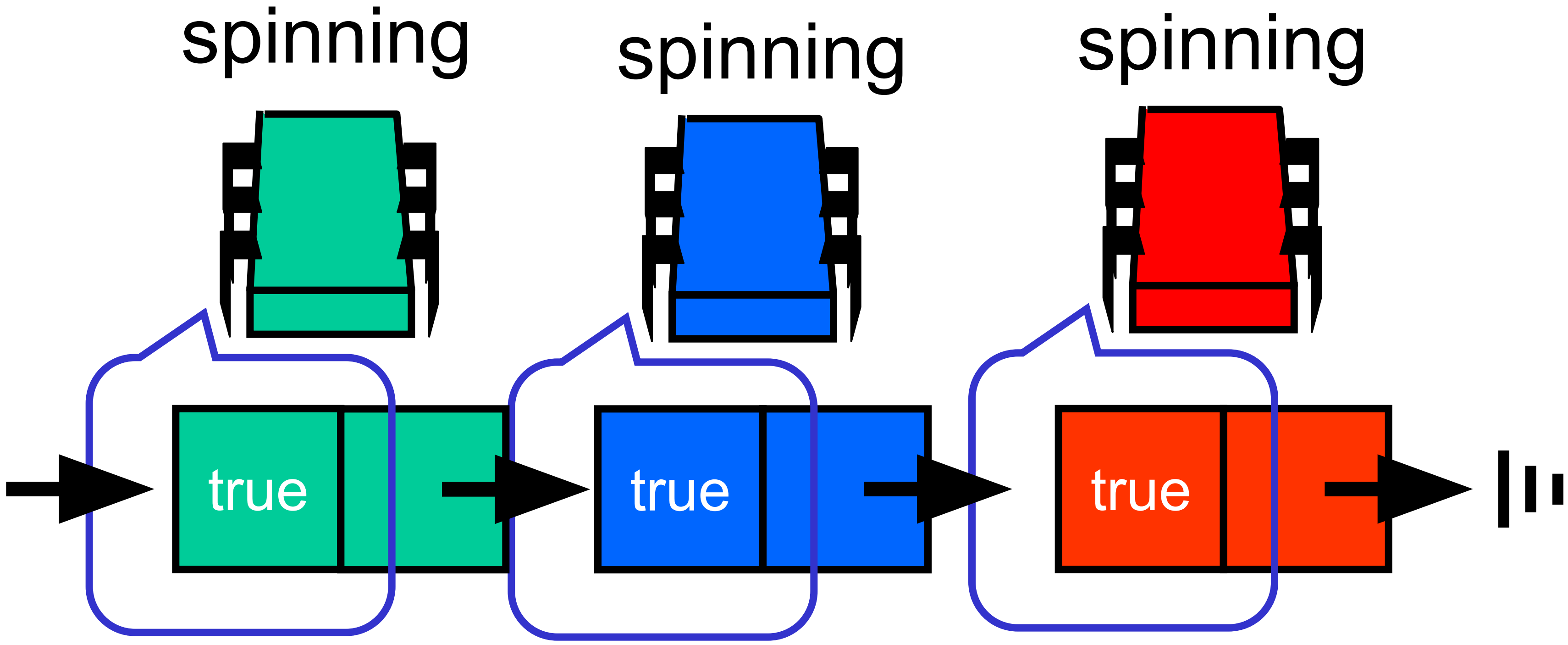
Queue Locks



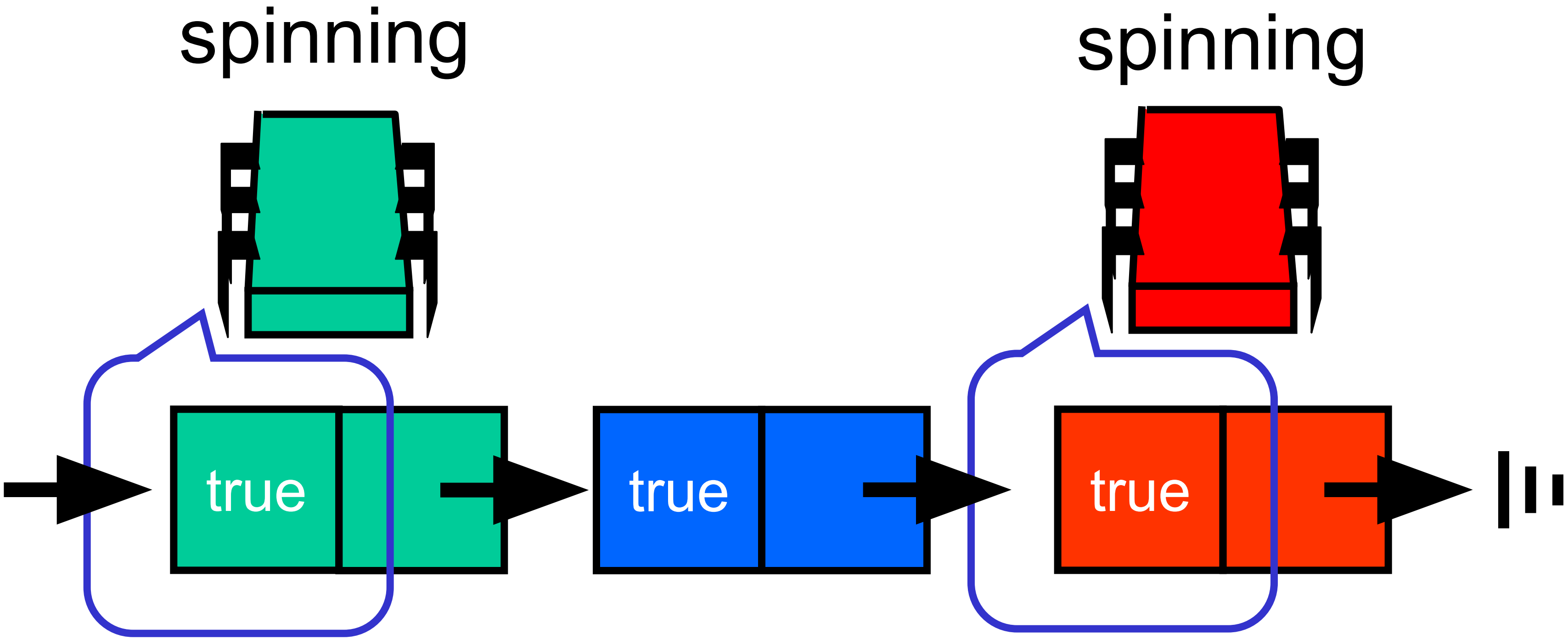
Queue Locks



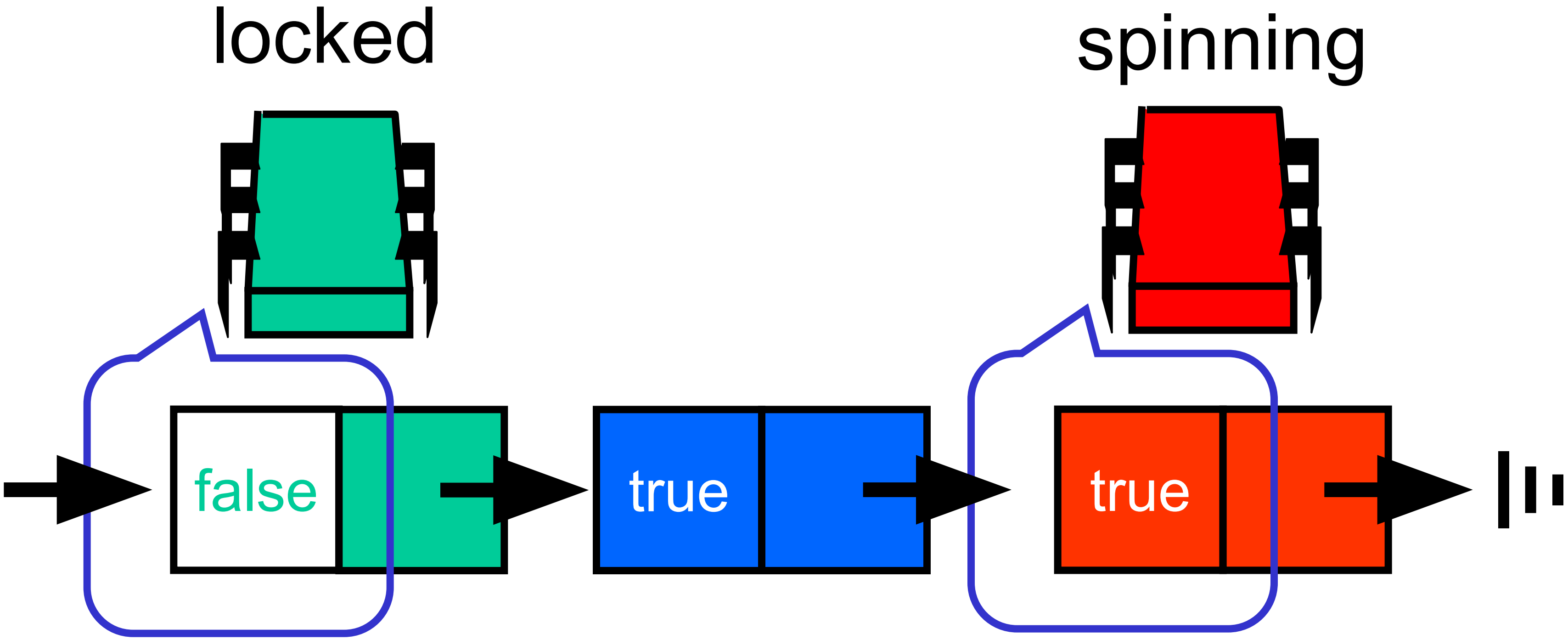
Queue Locks



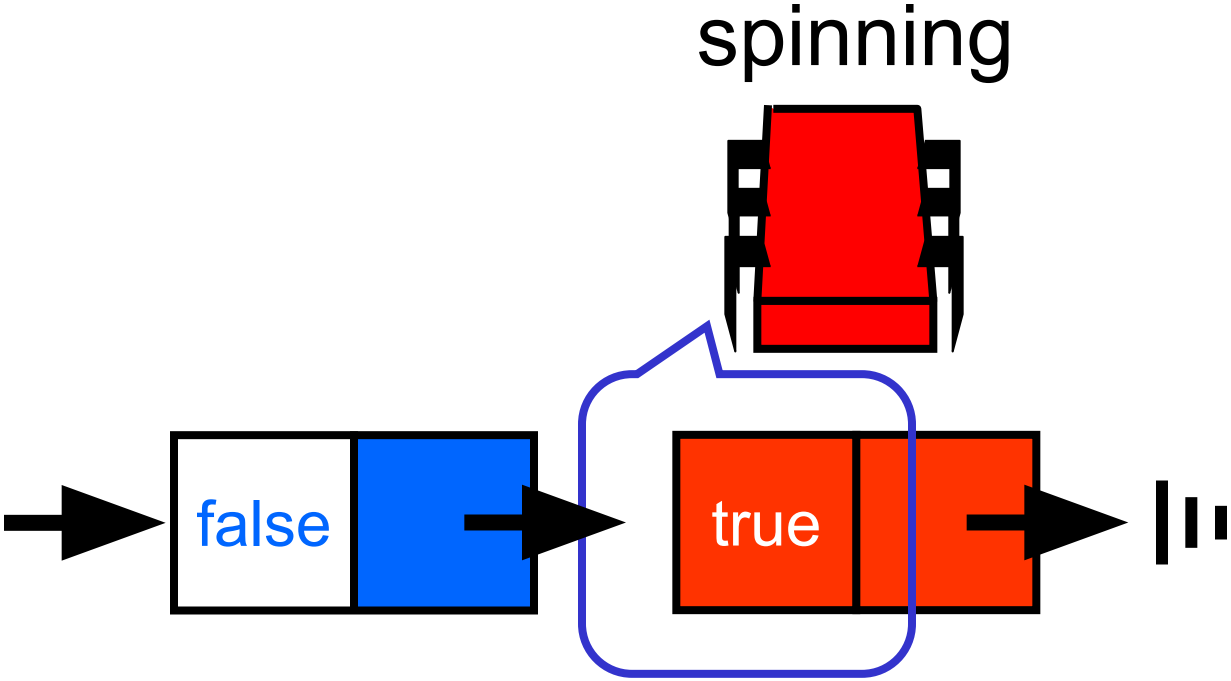
Queue Locks



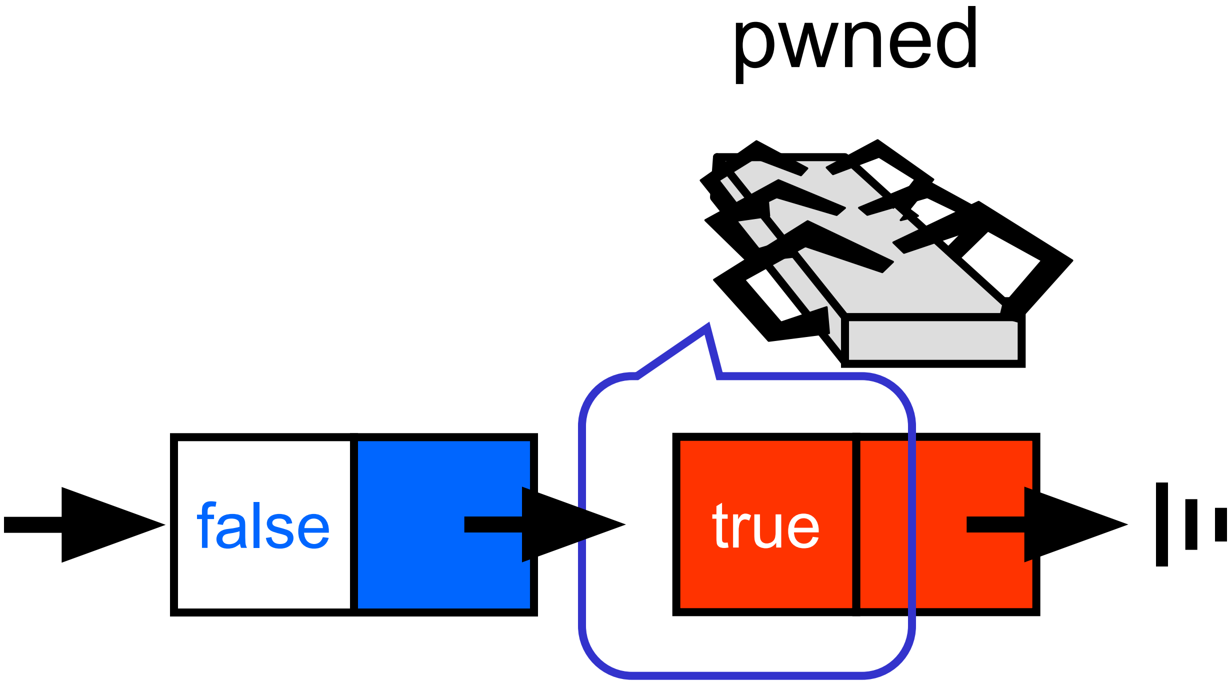
Queue Locks



Queue Locks



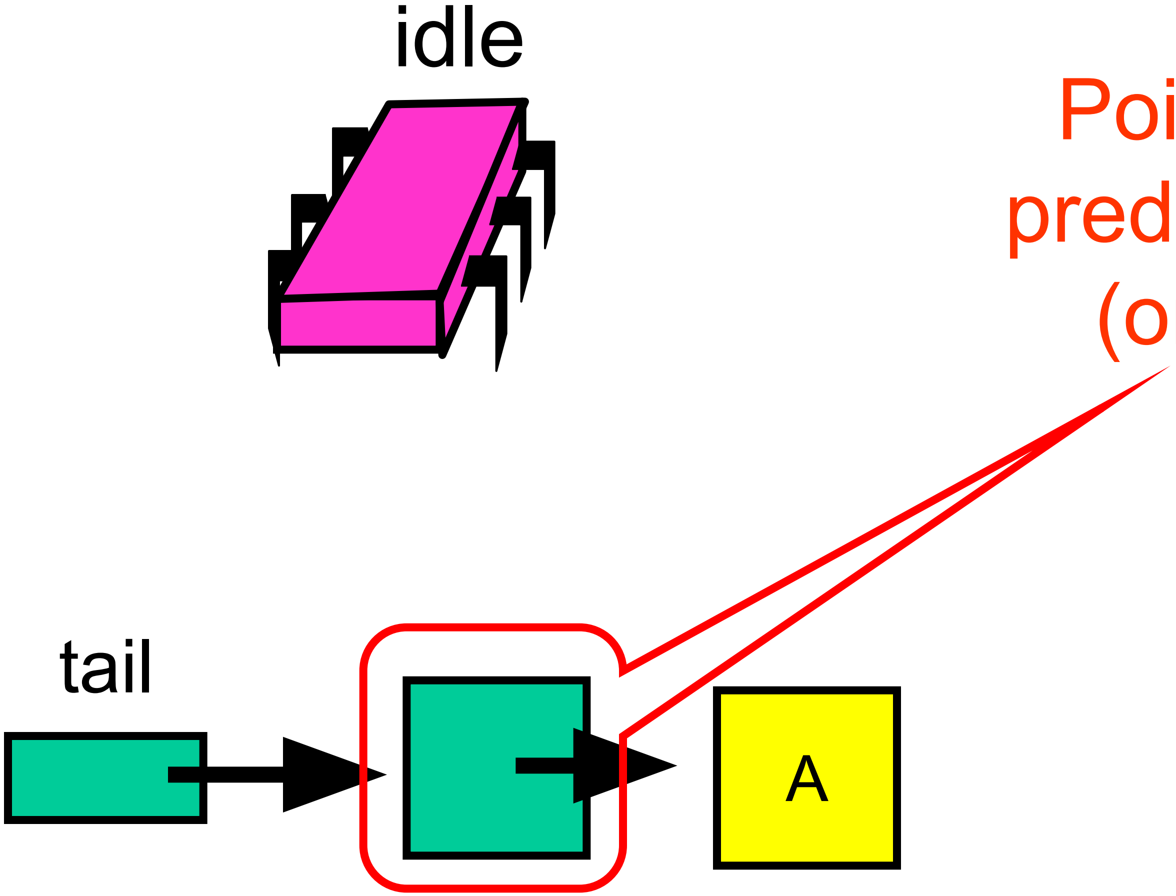
Queue Locks



Abortable CLH Lock

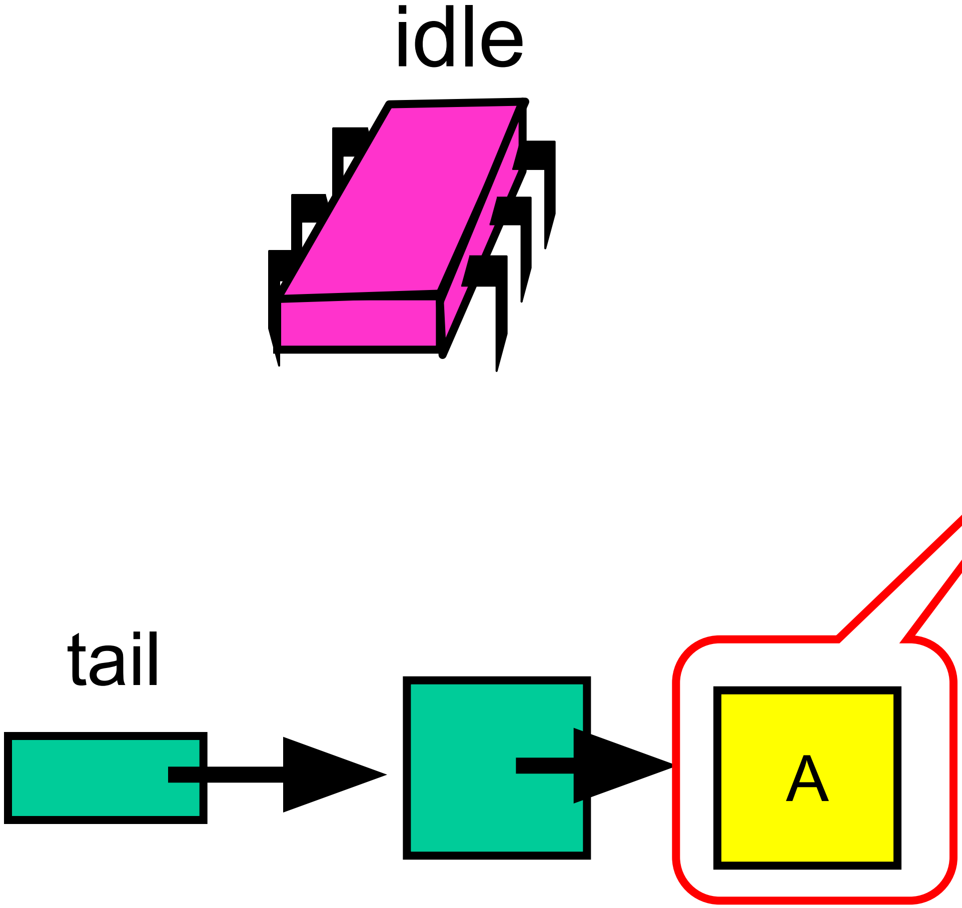
- When a thread gives up
  - Removing node in a wait-free way is hard
- Idea:
  - let successor deal with it.

Initially



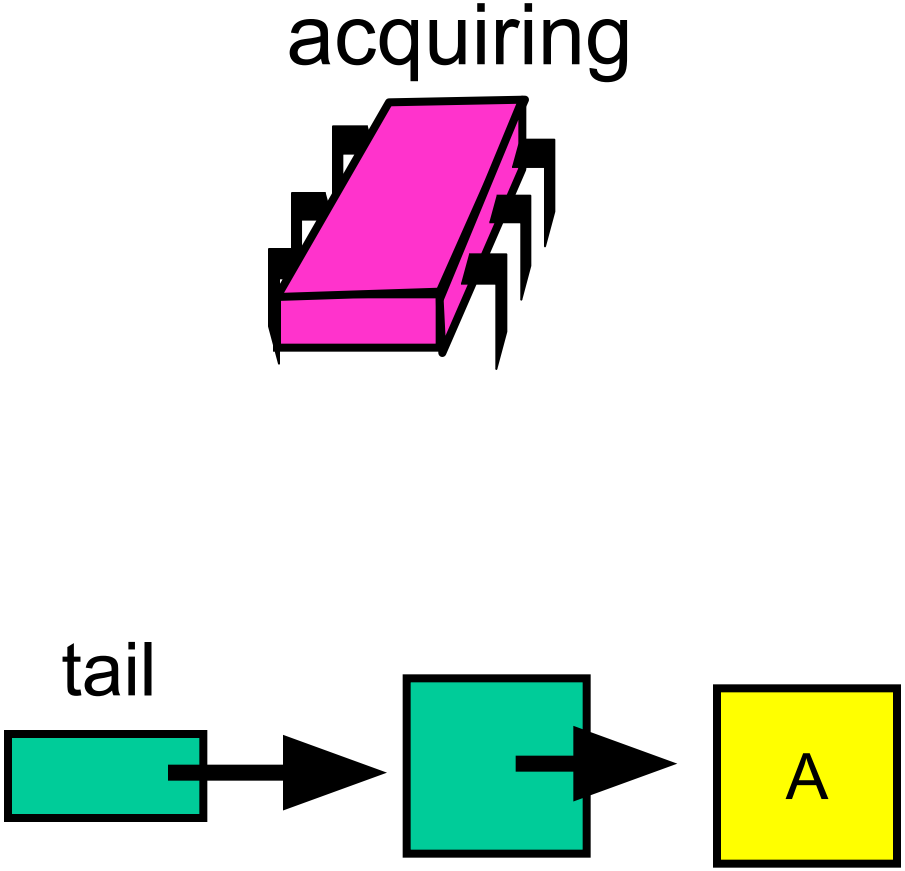
Pointer to predecessor (or null)

Initially



Distinguished available node means lock is free

Acquiring

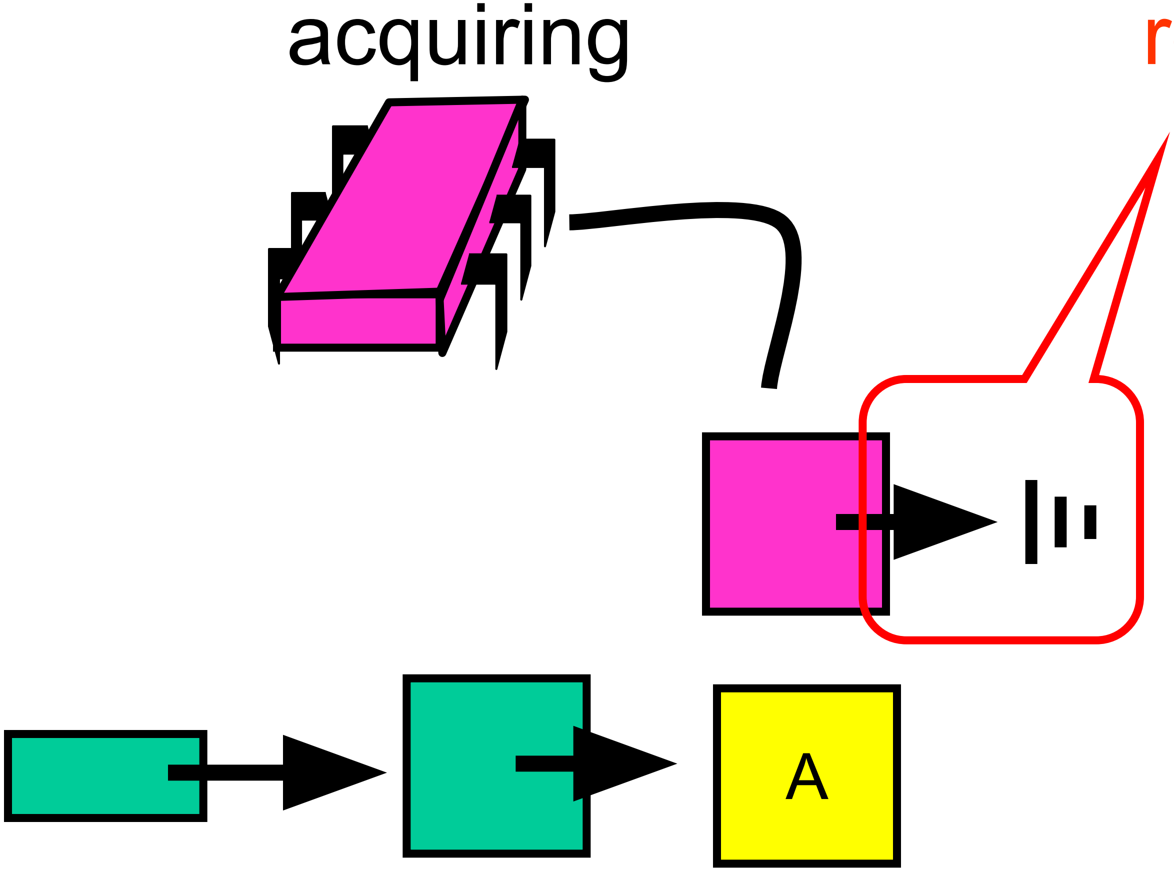


acquiring

tail

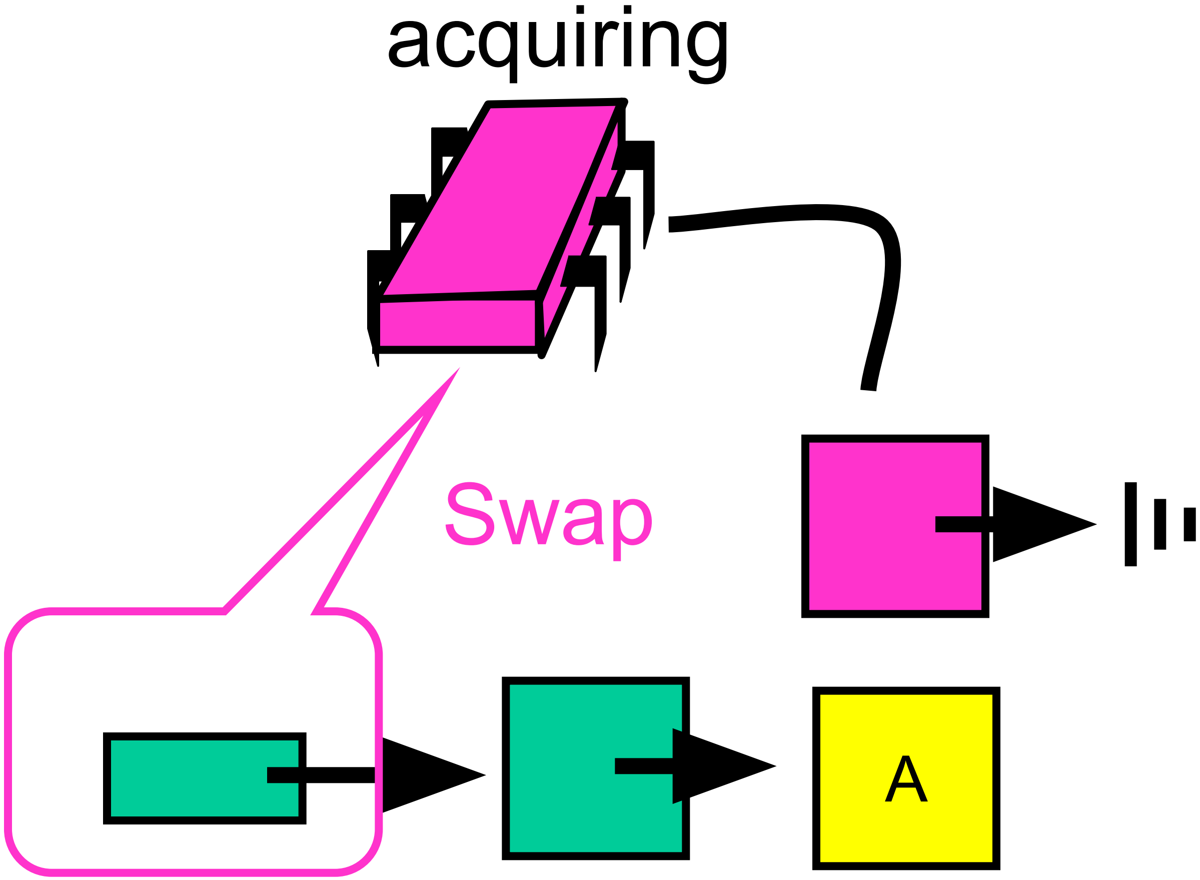
A

Acquiring

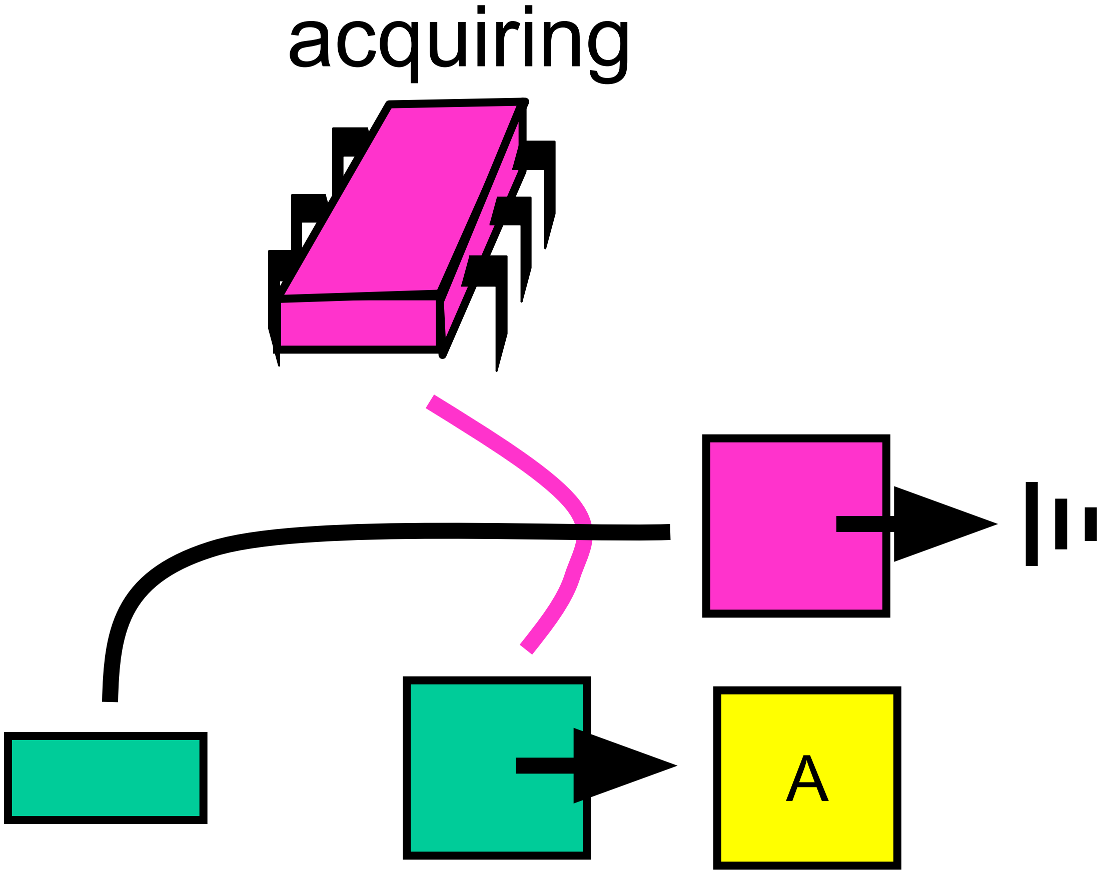


Null predecessor means lock not released or aborted

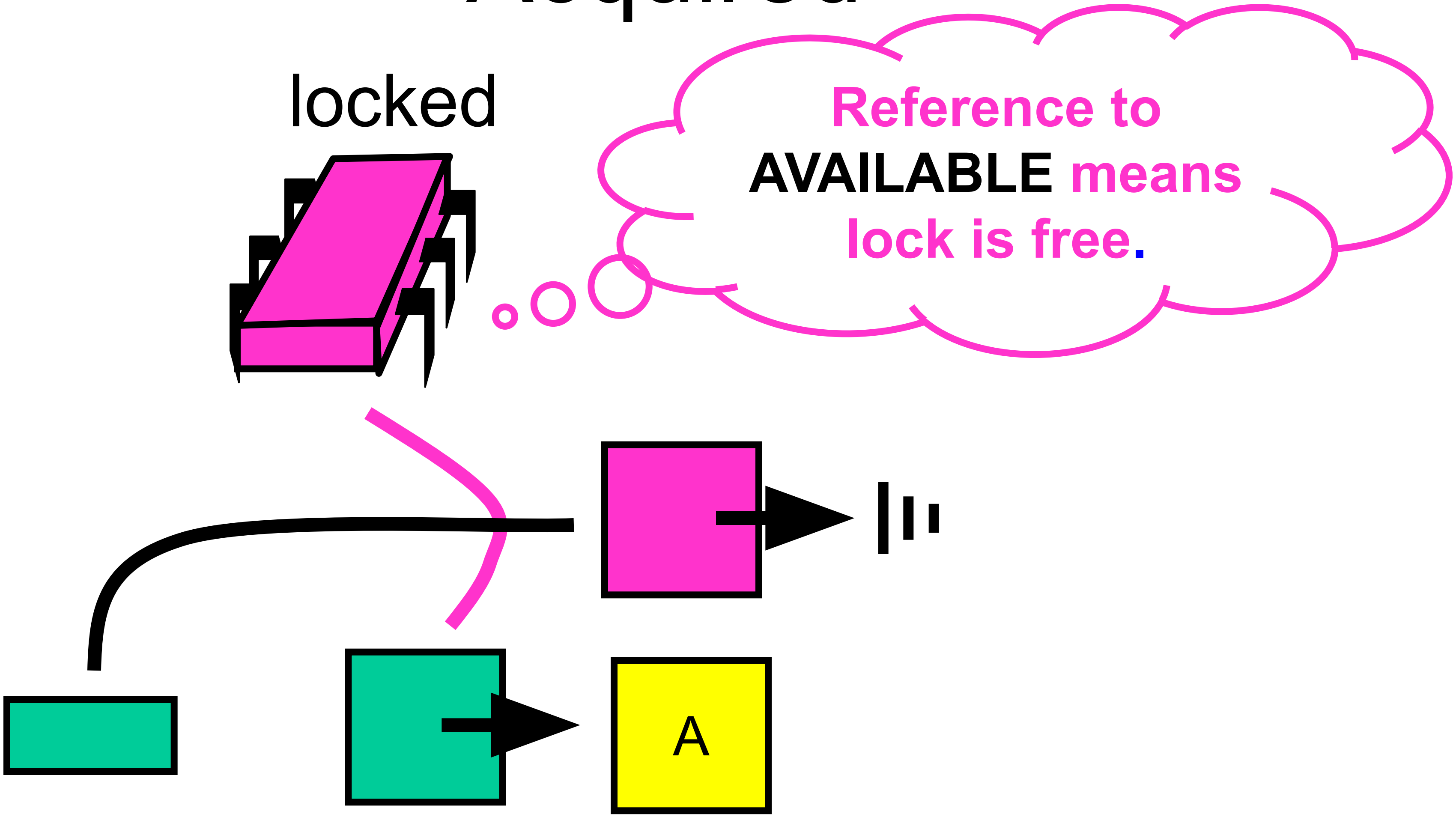
Acquiring



Acquiring

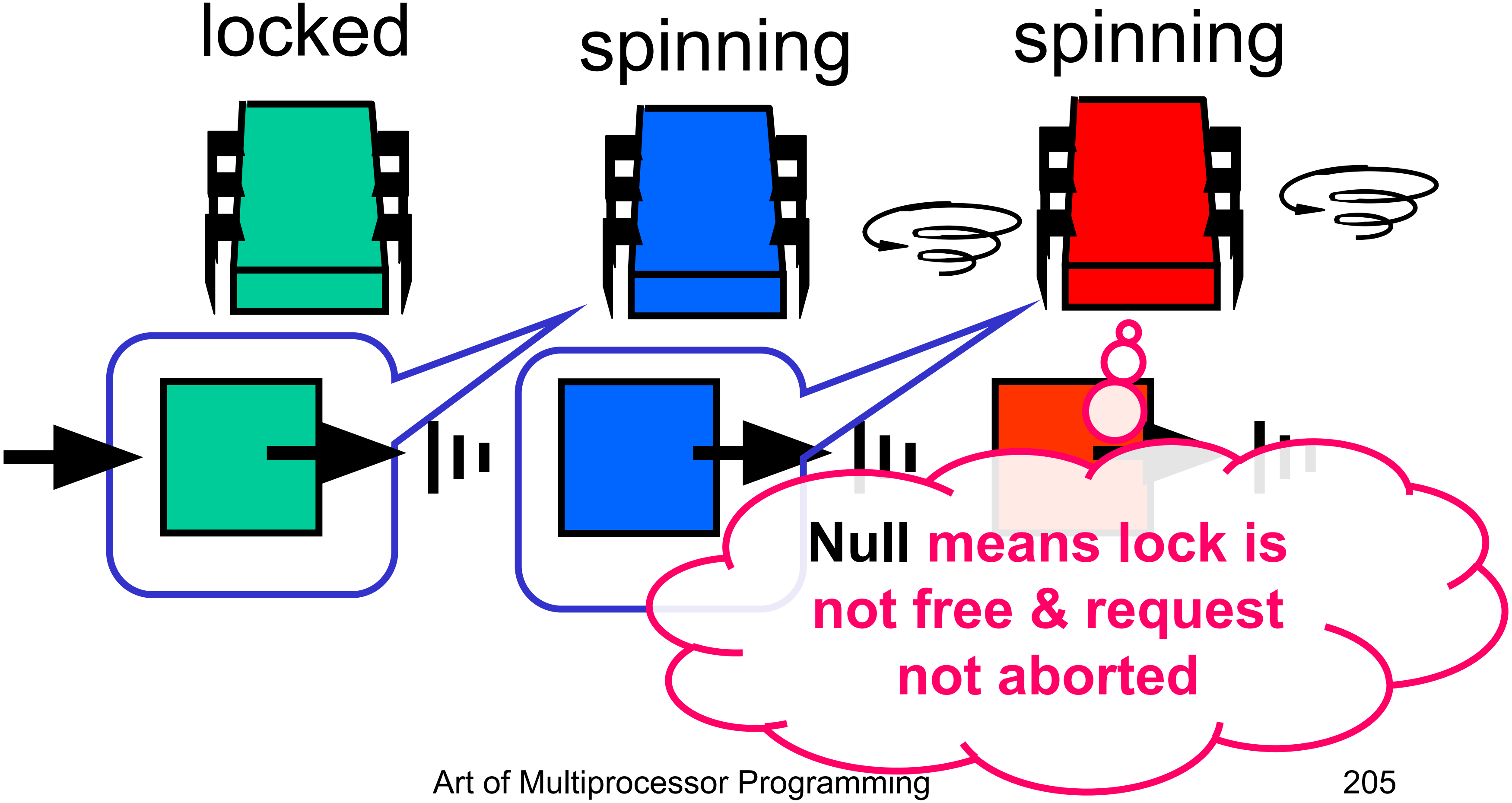


Acquired



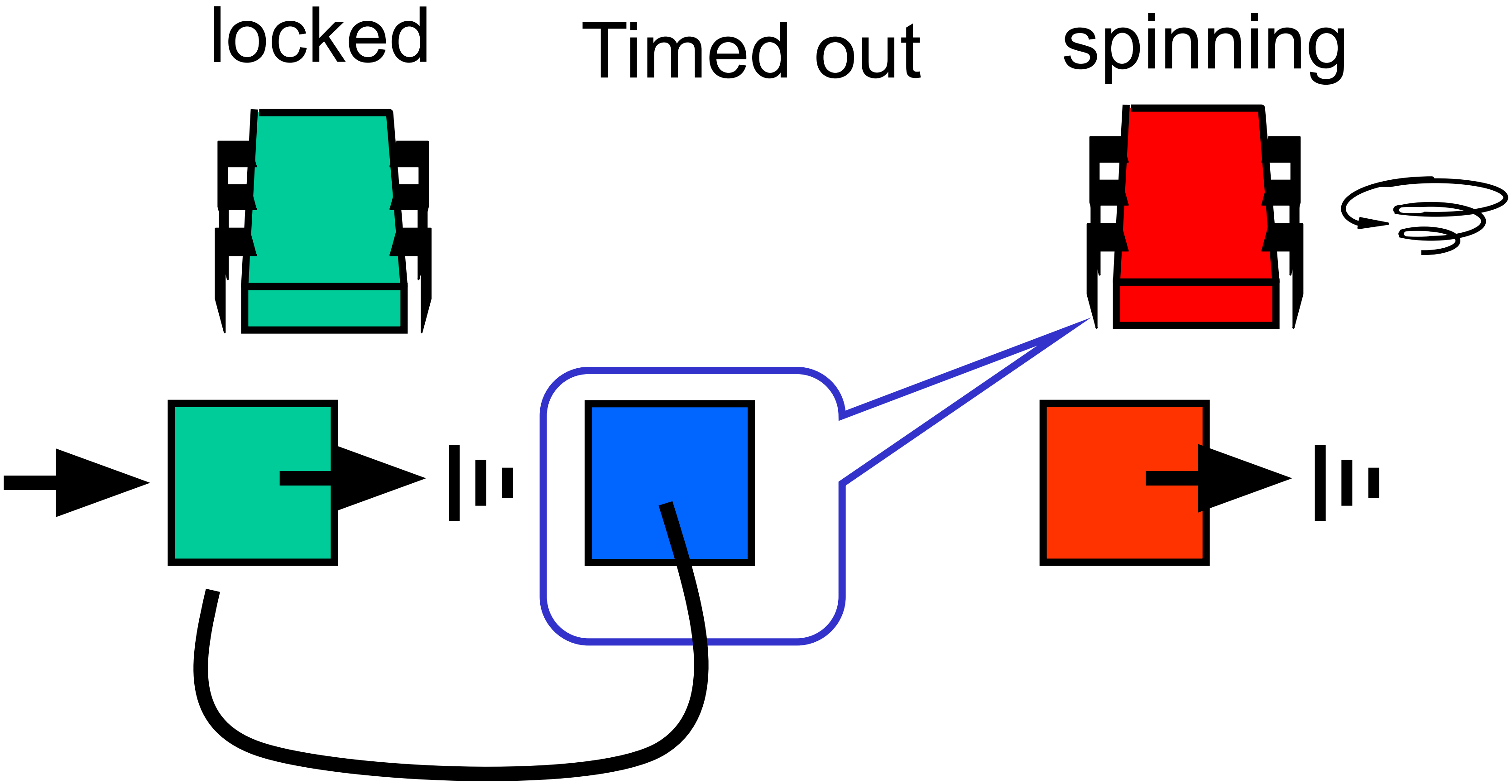
Reference to **AVAILABLE** means lock is free.

Normal Case

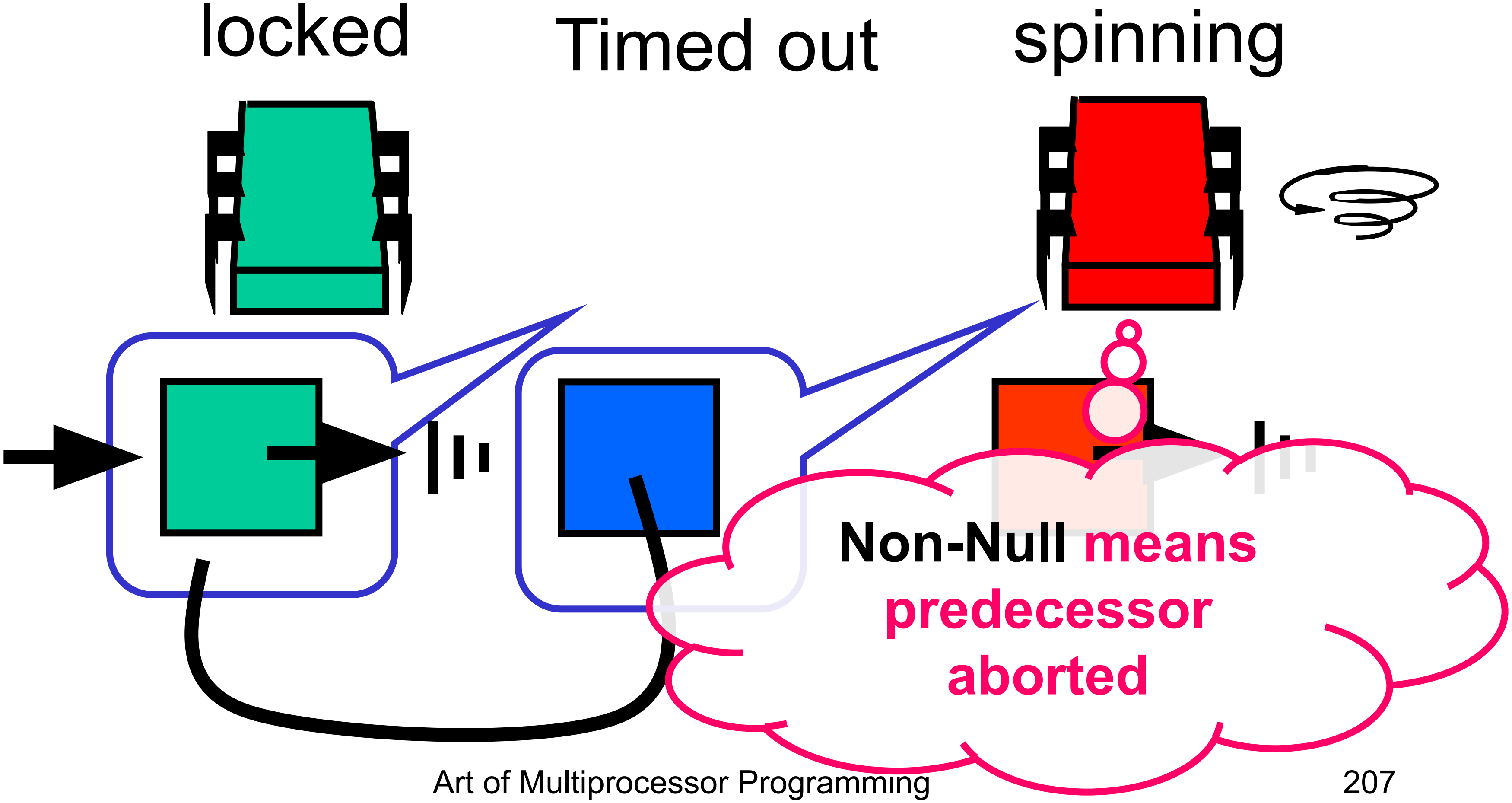


Null **means lock is not free & request not aborted**

One Thread Aborts

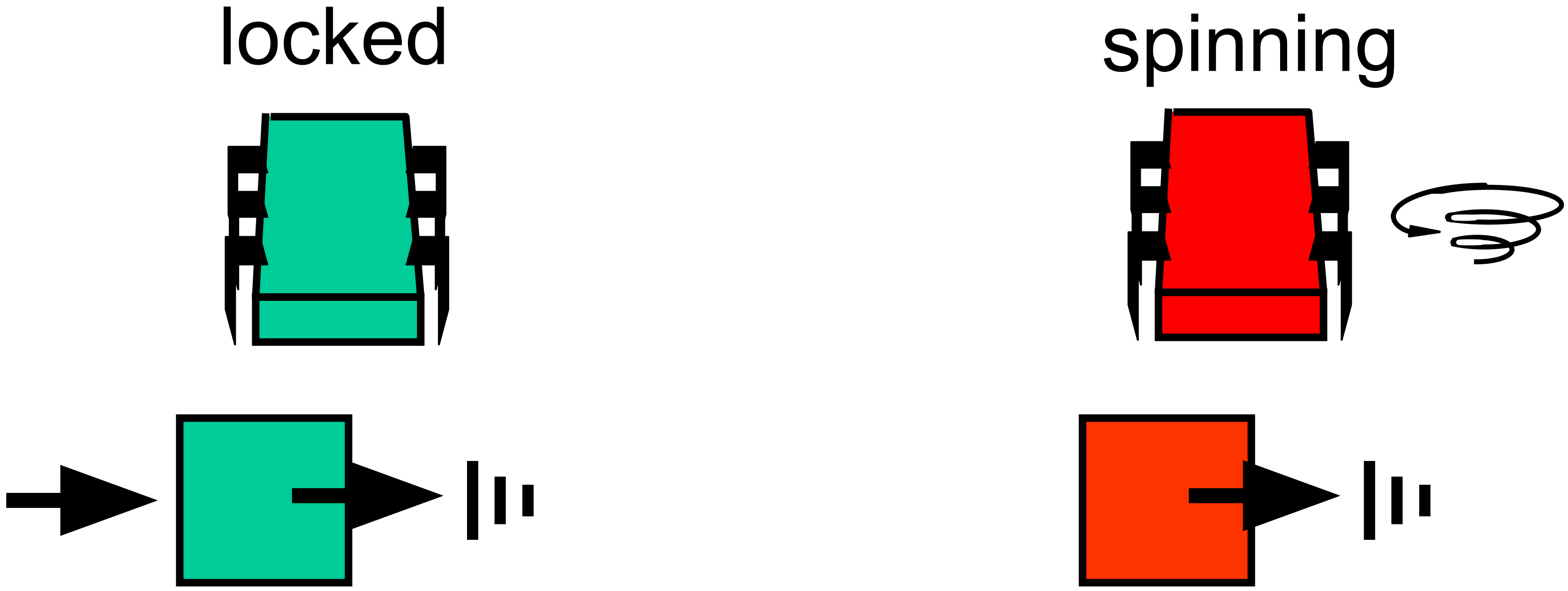


Successor Notices



Non-Null means predecessor aborted

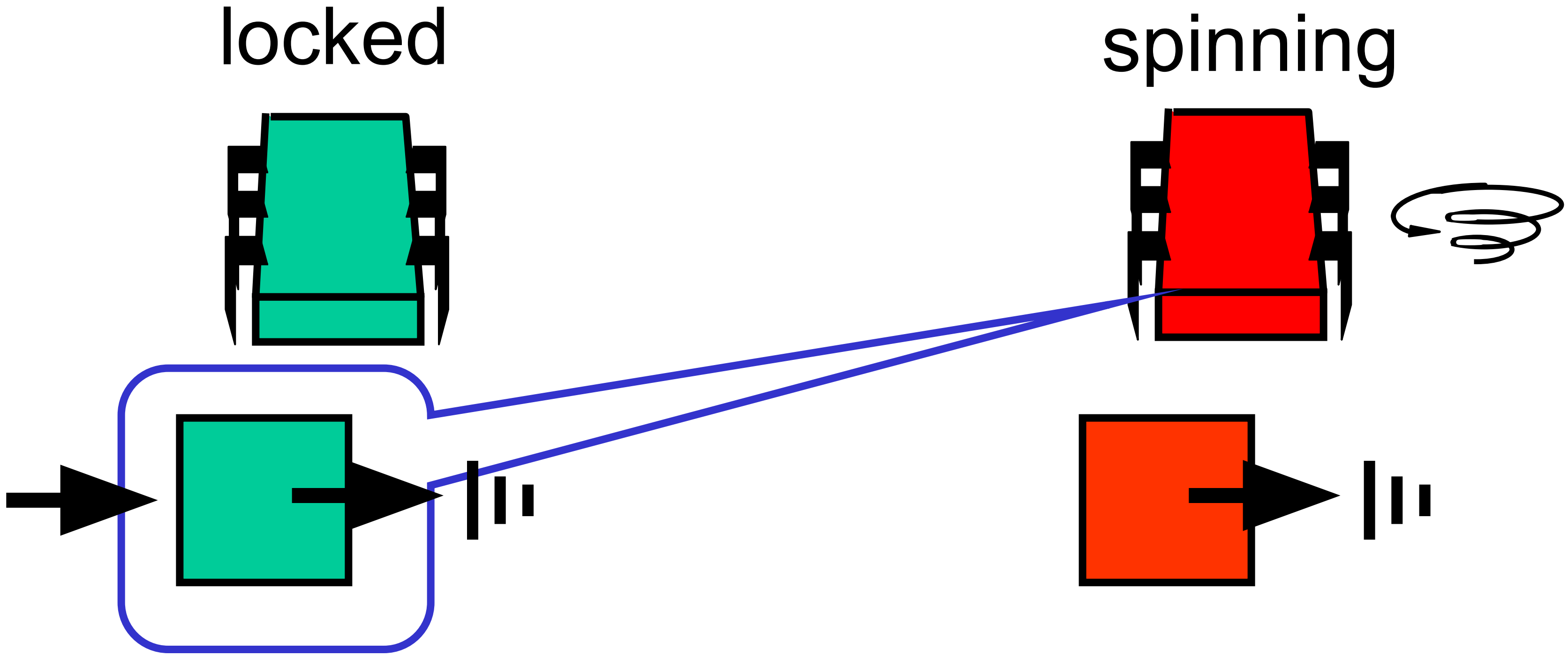
Recycle Predecessor’s Node



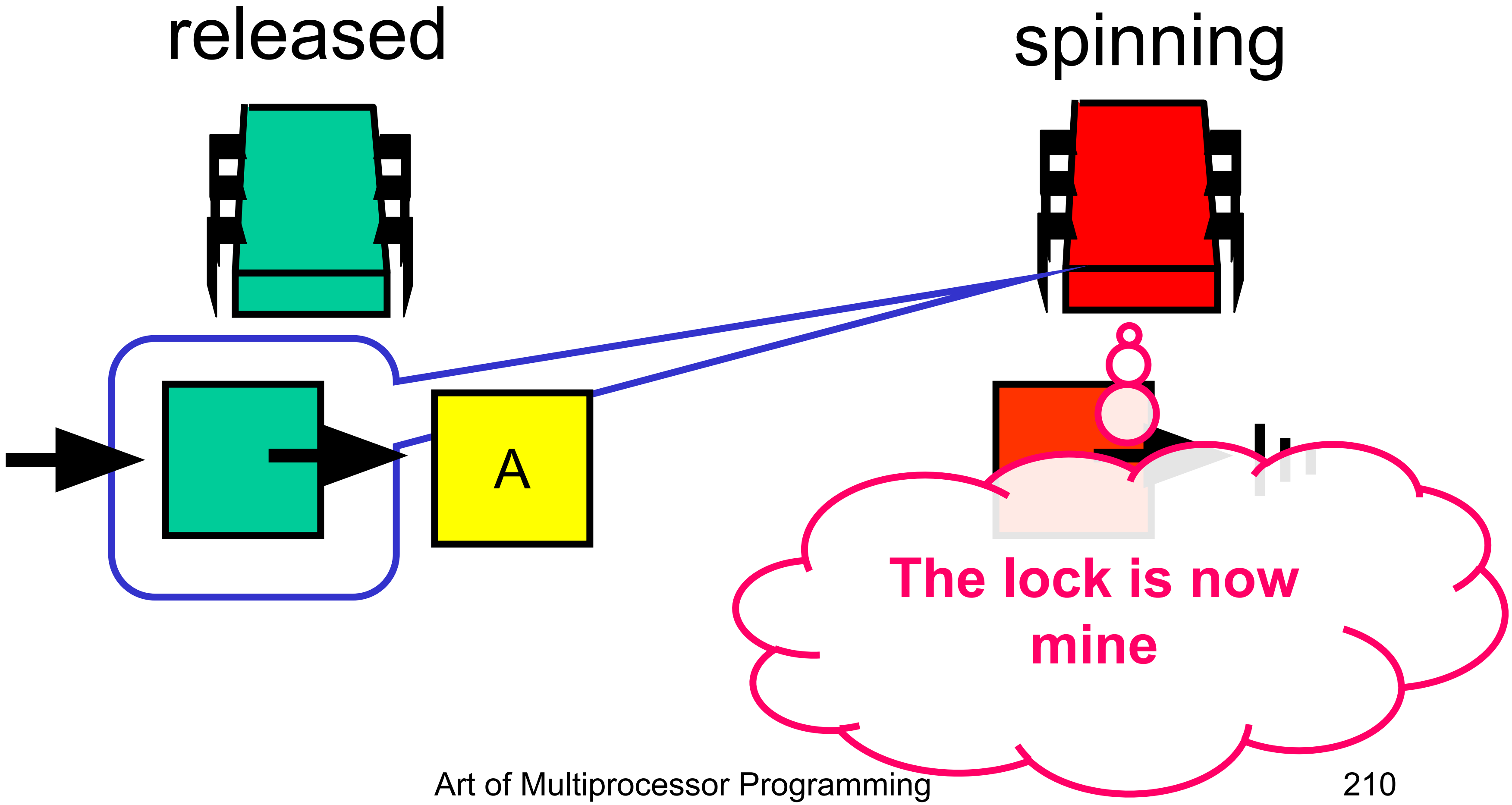
locked

spinning

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;
   }
```

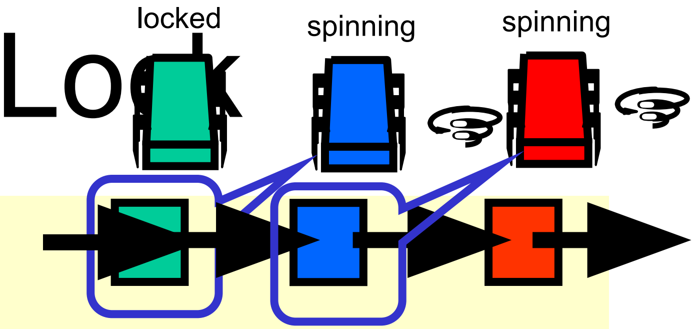
Swap with tail

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 fails, I do.
Tell it about myPred

Time-out Lock

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

If CAS succeeds: no successor, simply return false

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

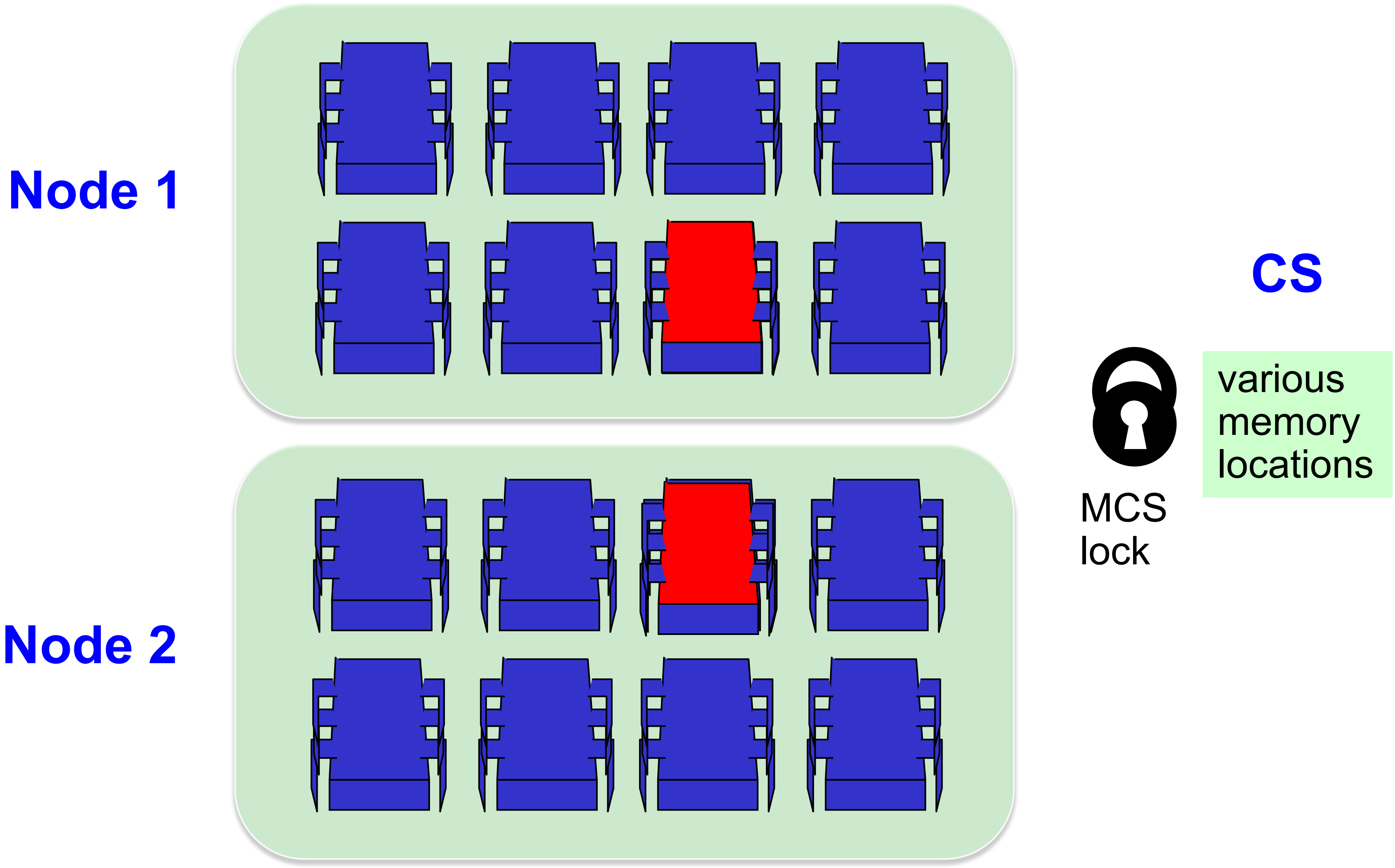
```
public void unlock() {
  QNode qnode = myNode.get();
  if (!tail.compareAndSet(qnode, null))
    qnode.prev = AVAILABLE;
}
```

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



Node 1

Node 2

CS

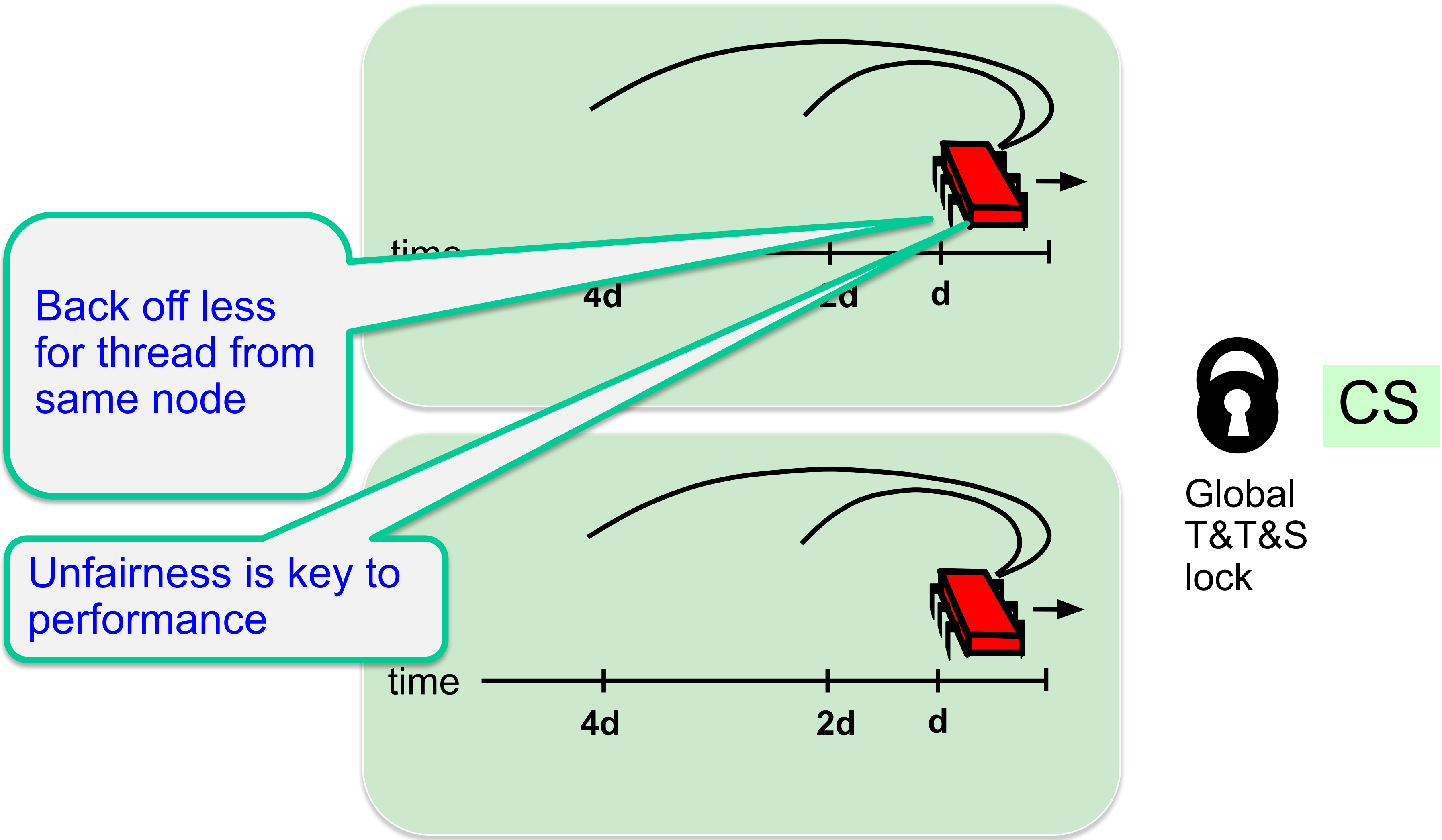
various memory locations

MCS lock

“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)



Back off less for thread from same node

Unfairness is key to performance

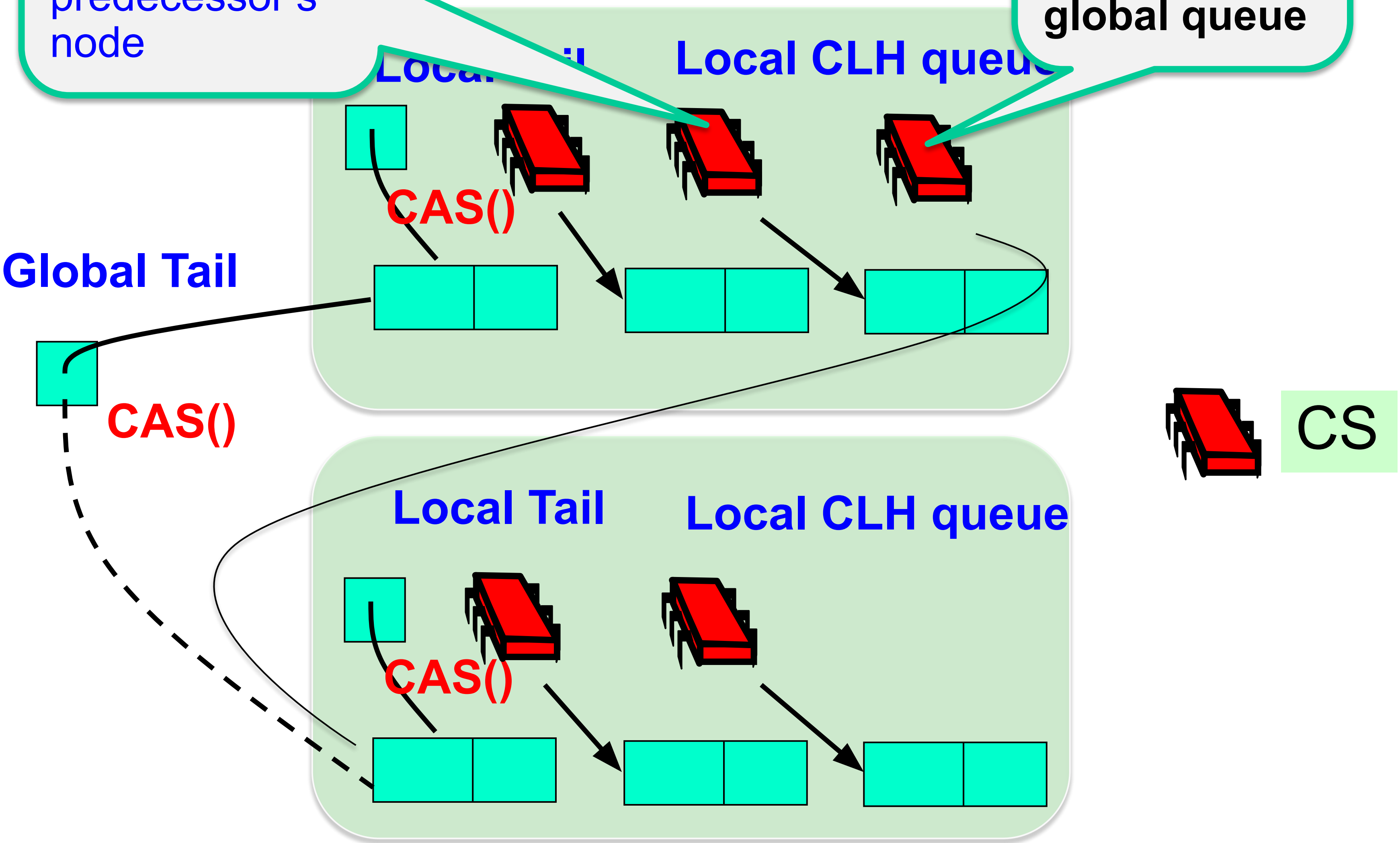
Hierarchical Backoff Lock (HBO)

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

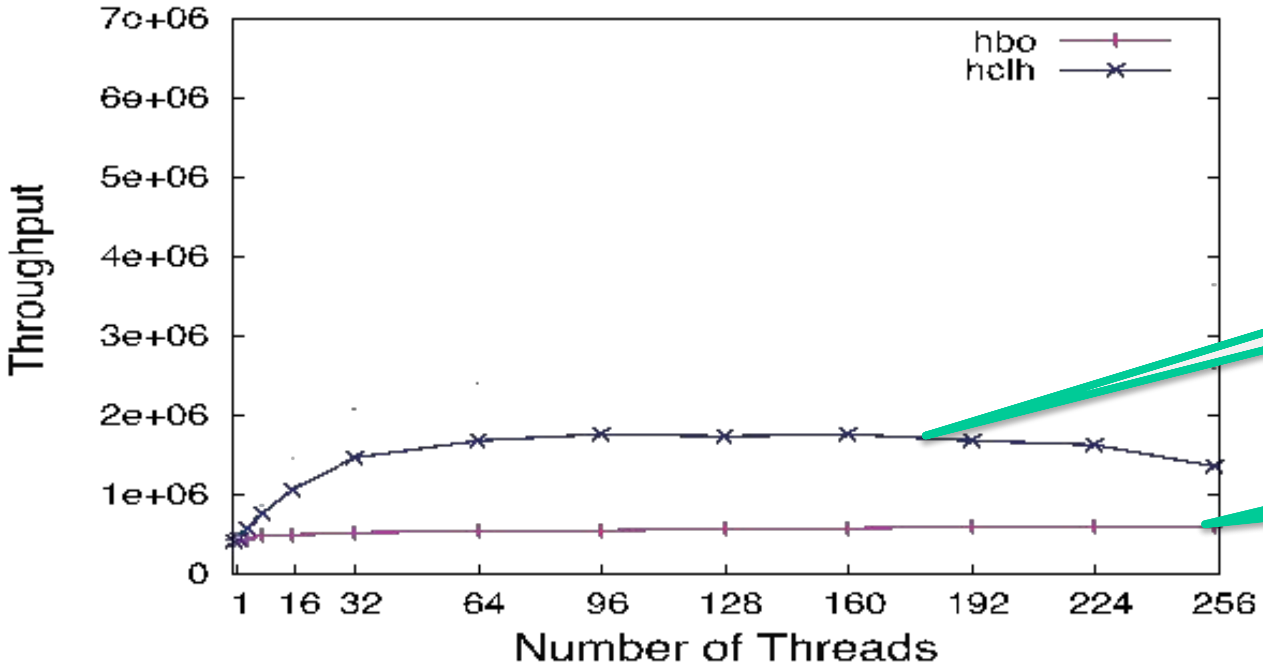
chical CLH Lock (

Each thread spins on cached copy of predecessor’s node

Thread at local head splices **local queue into global queue**



Hierarchical CLH Lock (HCLH)



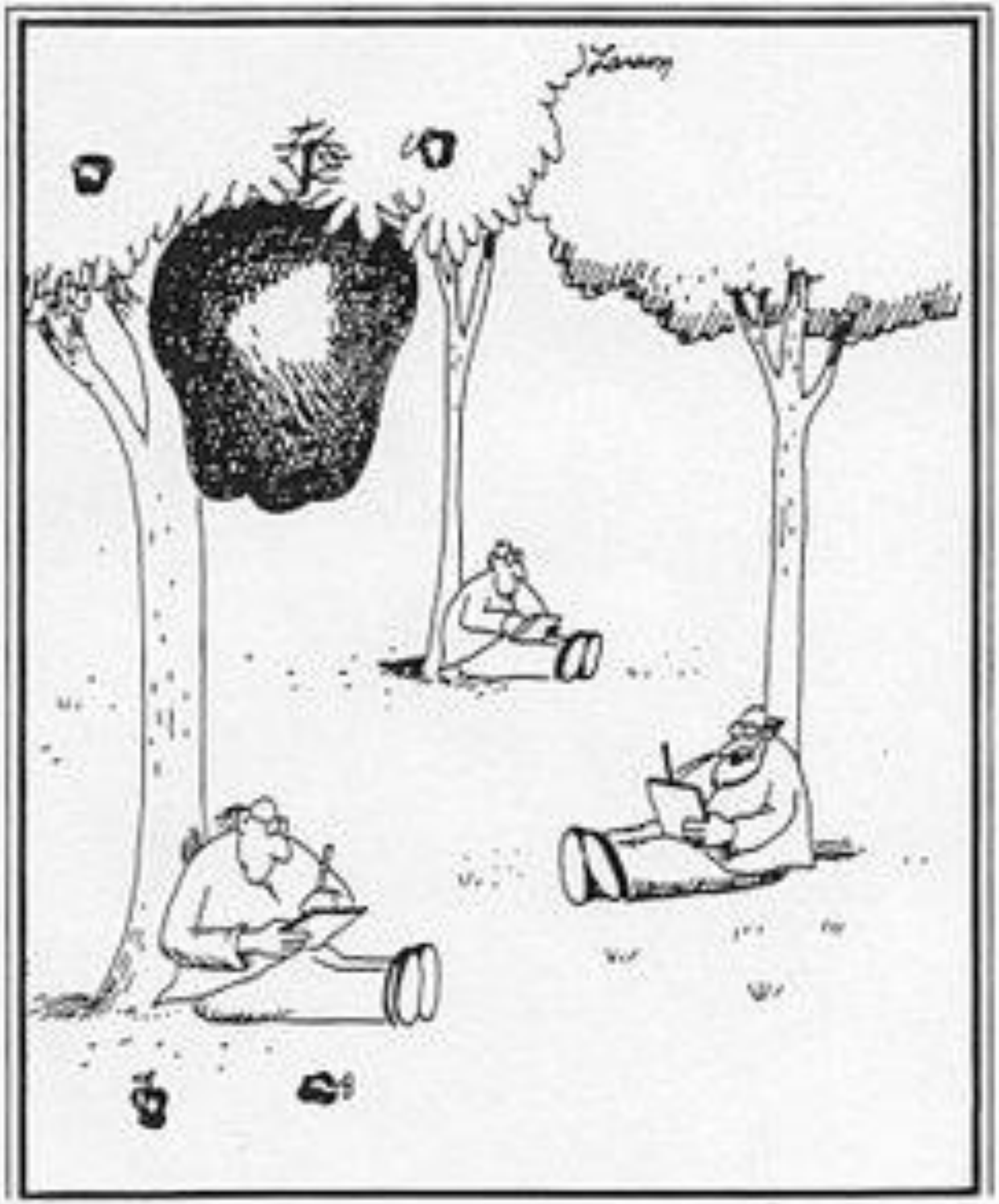
HCLH

HBO

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

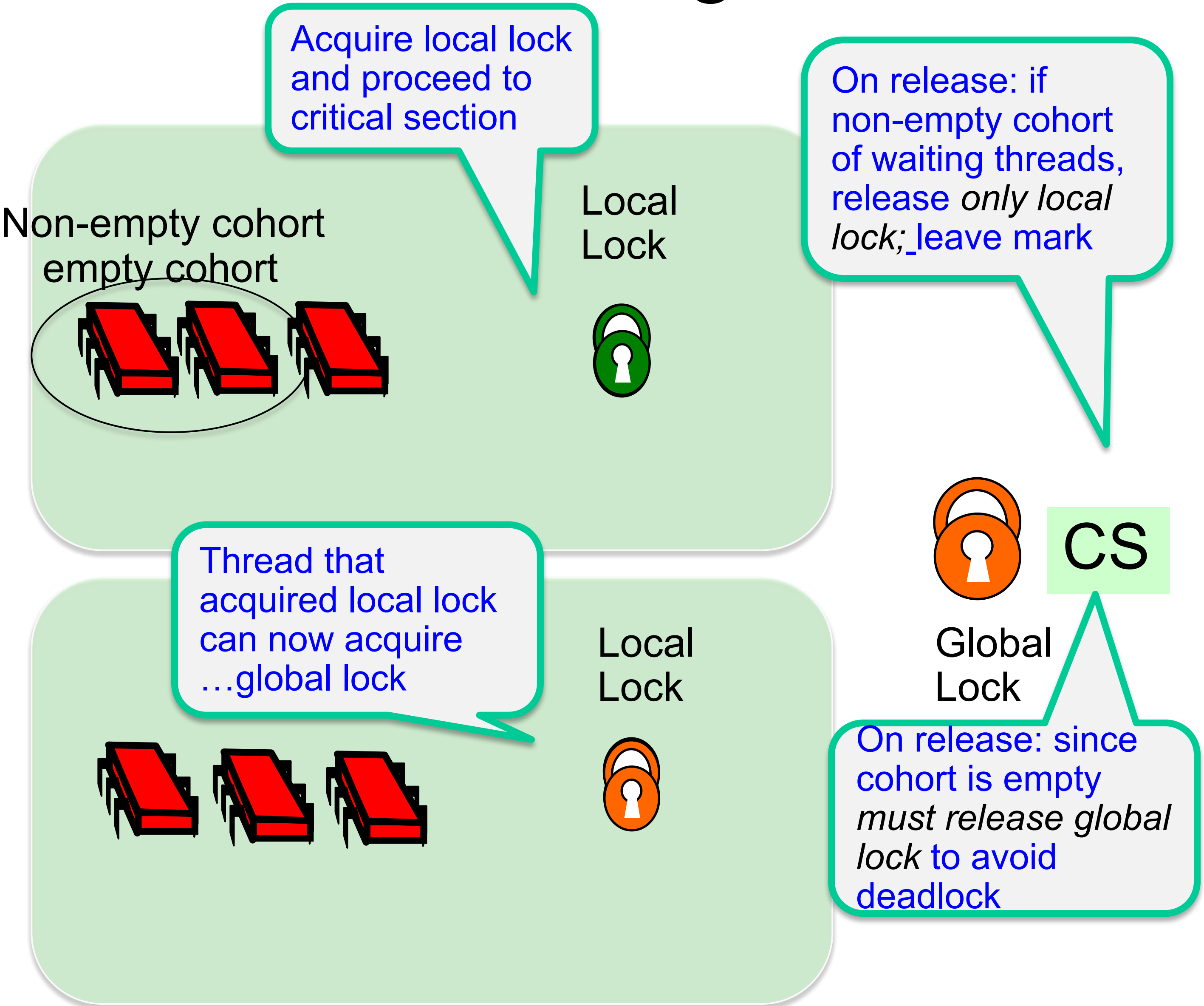


"Nothing yet. . . . How about you, Newton?"

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



Acquire local lock and proceed to critical section

On release: if non-empty cohort of waiting threads, release *only local lock;*_leave mark

Thread that acquired local lock can now acquire …global lock

On release: since cohort is empty *must release global lock* to avoid deadlock

Thread Obliviousness

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

Cohort Detection

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

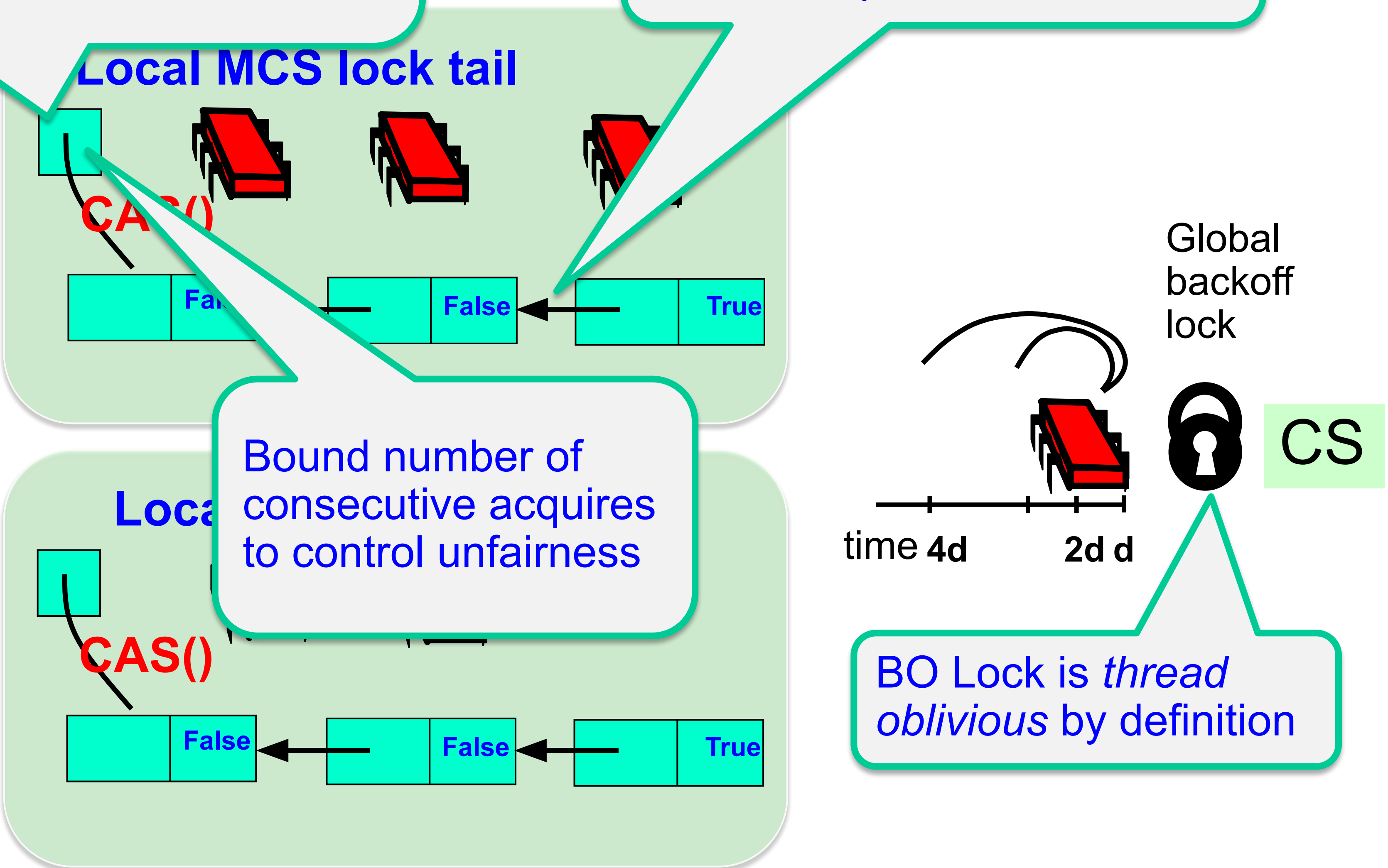
Lock Cohorting

- 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

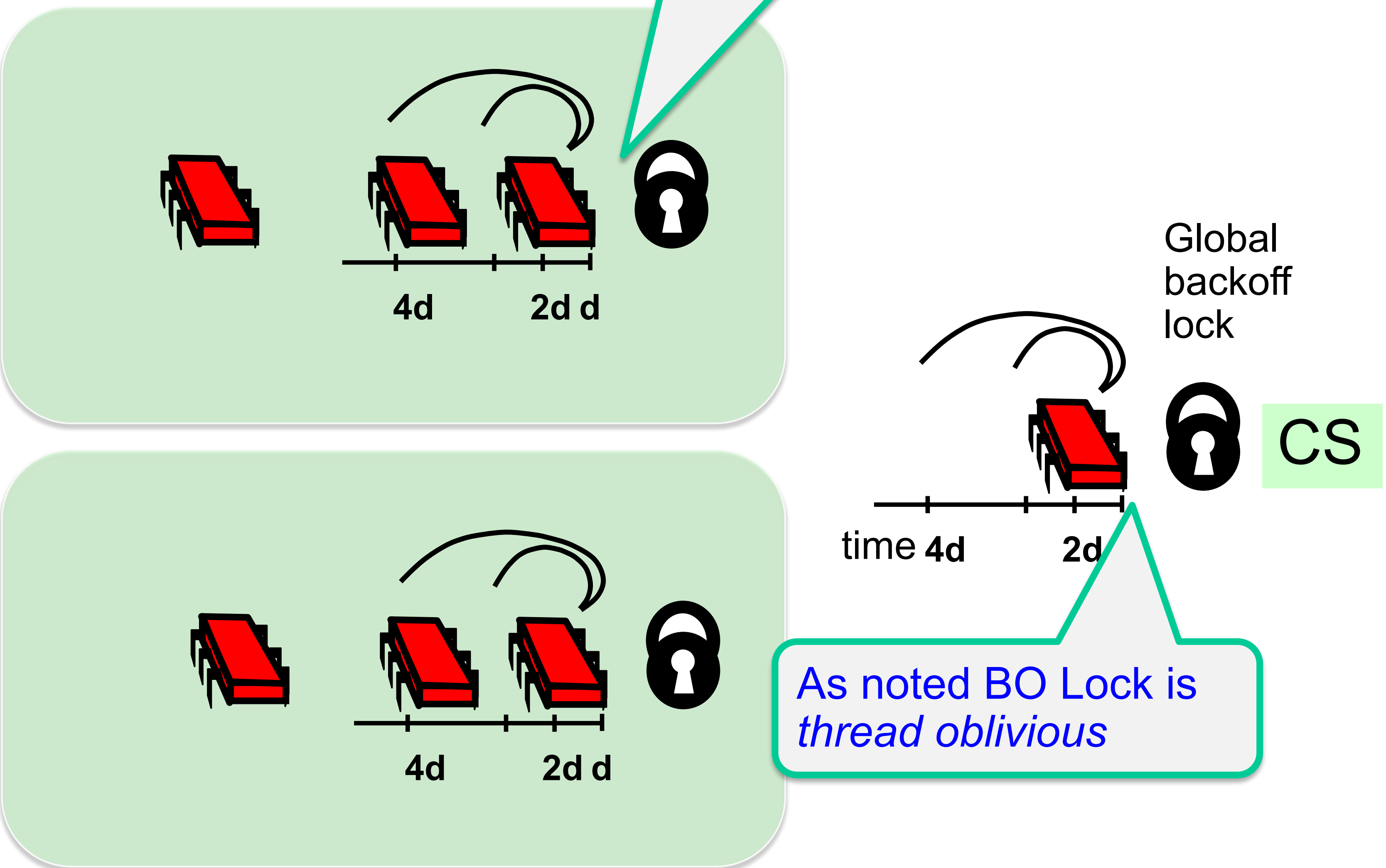


Bound number of consecutive acquires to control unfairness

BO Lock is *thread oblivious* by definition

Lock Co BO Lock

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



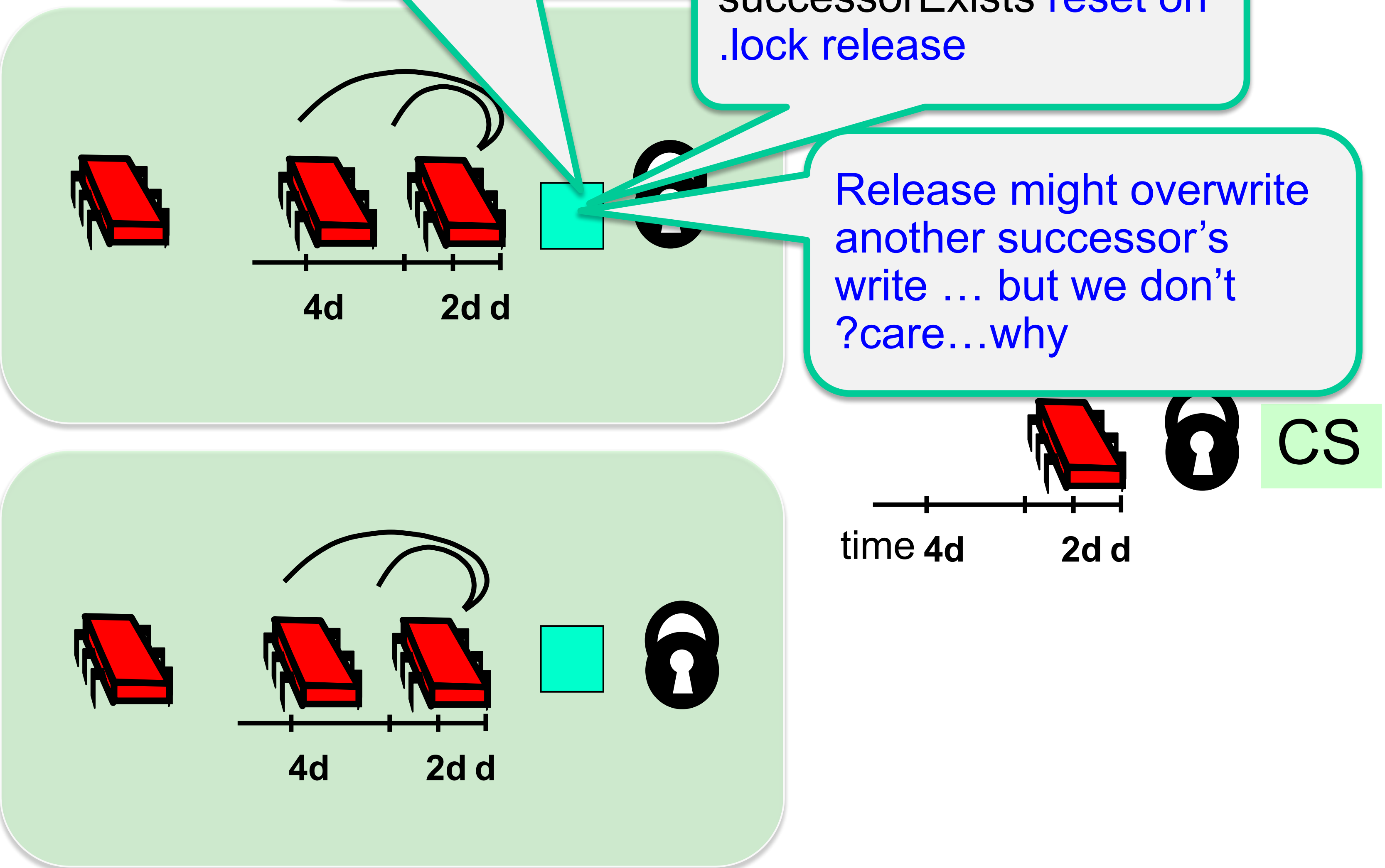
As noted BO Lock is *thread oblivious*

Lock Co -BO Lock

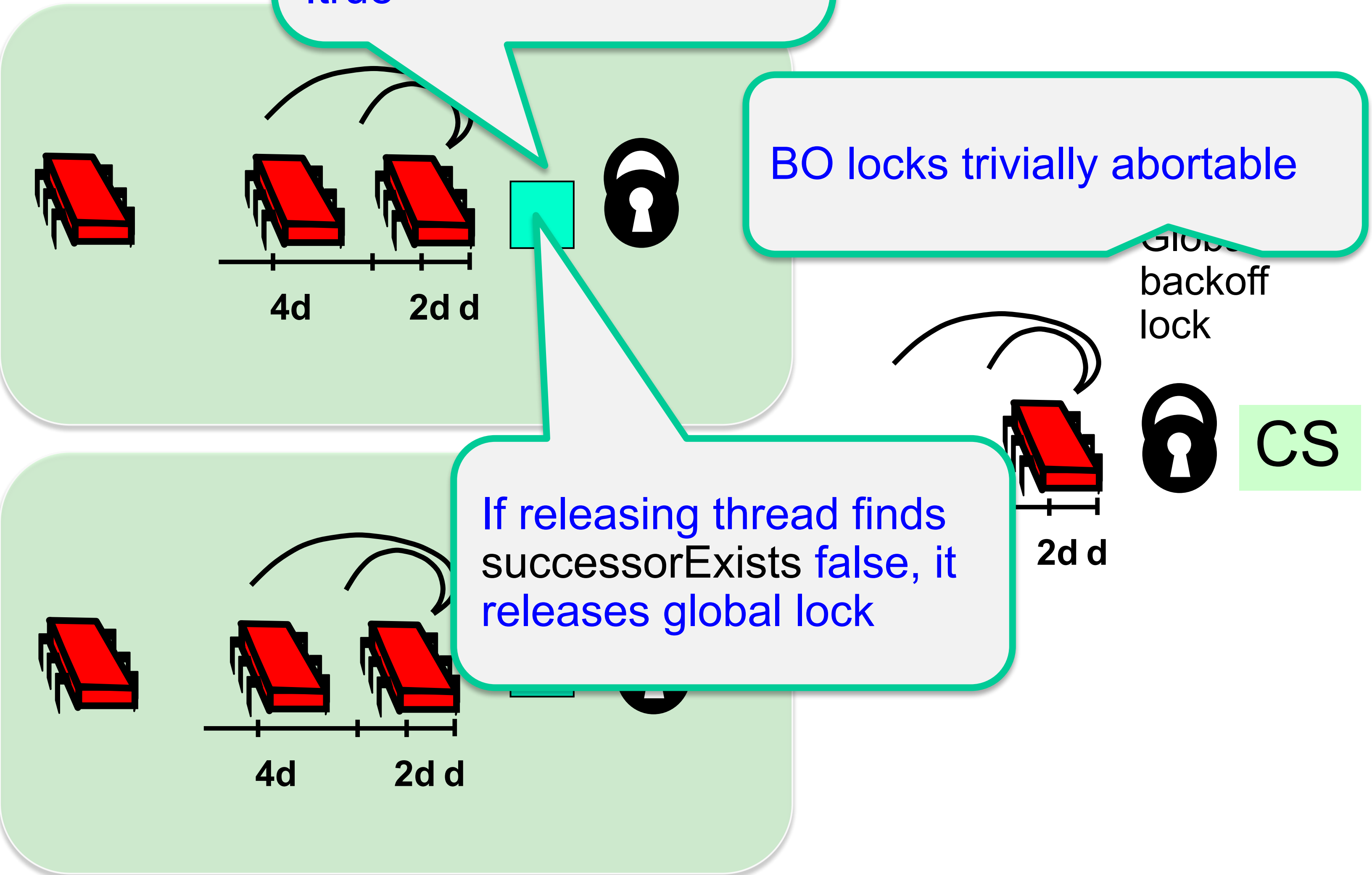
Add successorExists field before attempting to .acquire local lock

successorExists reset on .lock release

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



C-B e-Out



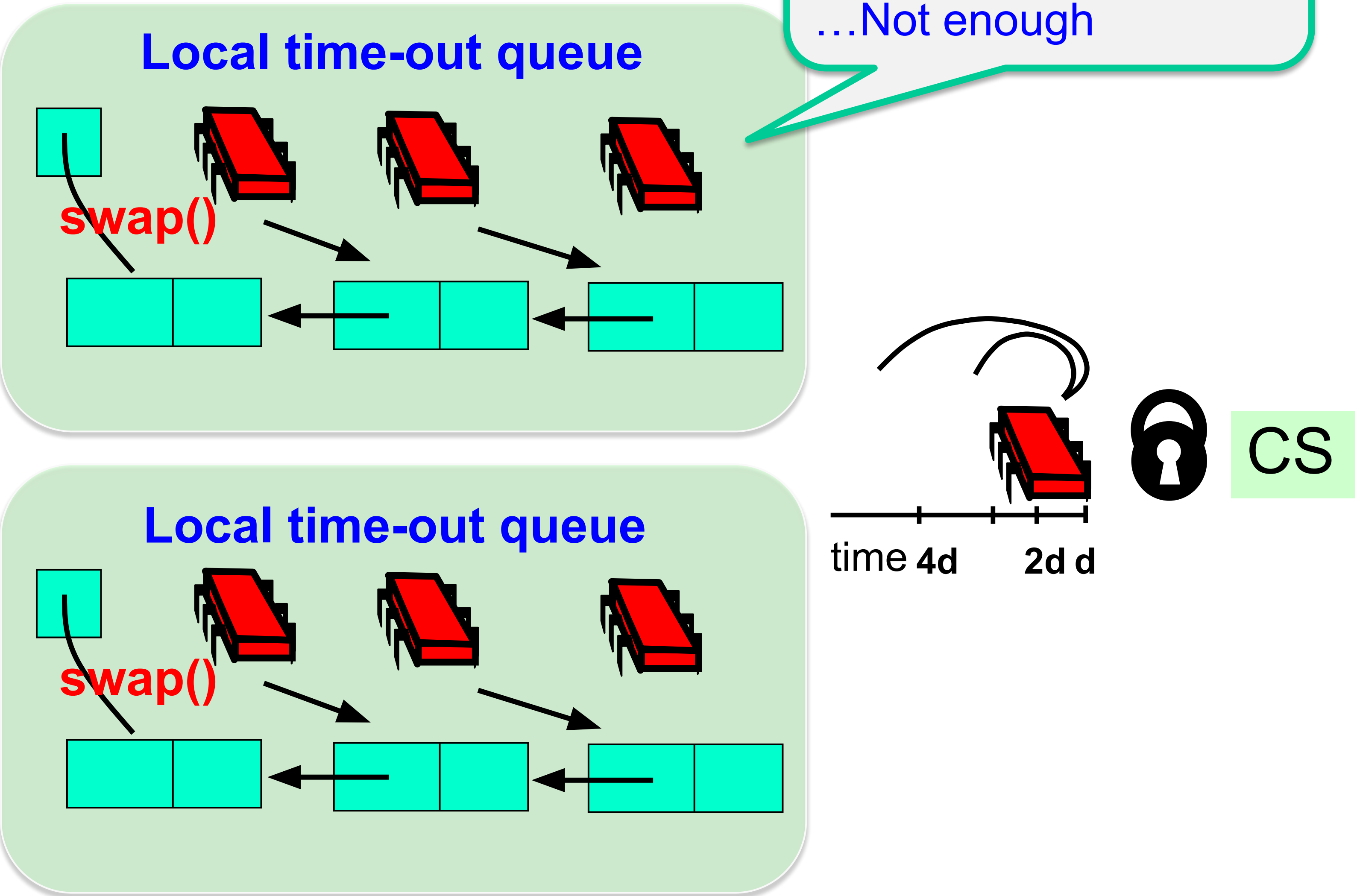
Aborting thread resets successorExists field before leaving local lock. Spinning threads set it to .true

BO locks trivially abortable

If releasing thread finds successorExists false, it releases global lock

Time-Out NUMA Lock

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


Local time-out queue

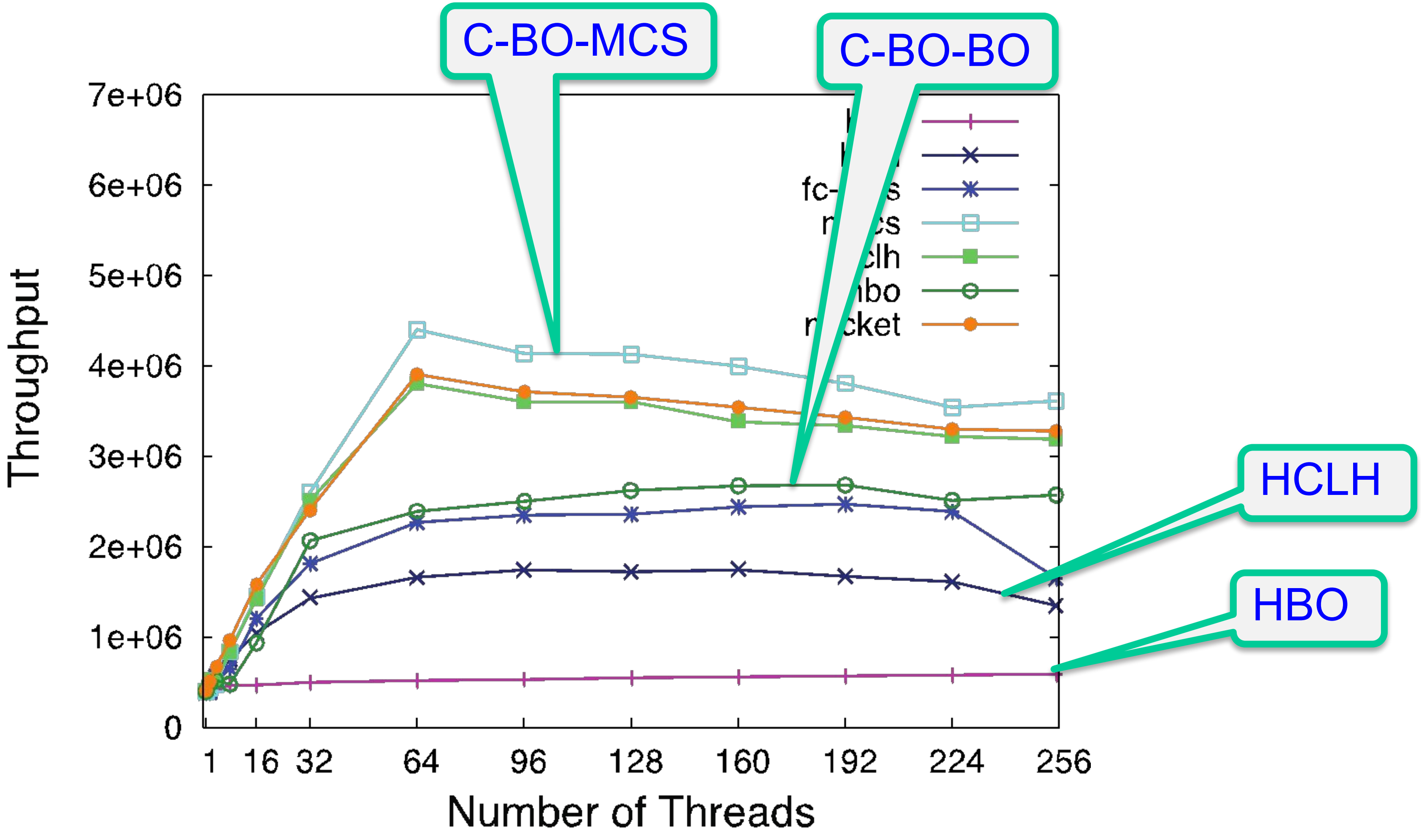
Local time-out queue

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



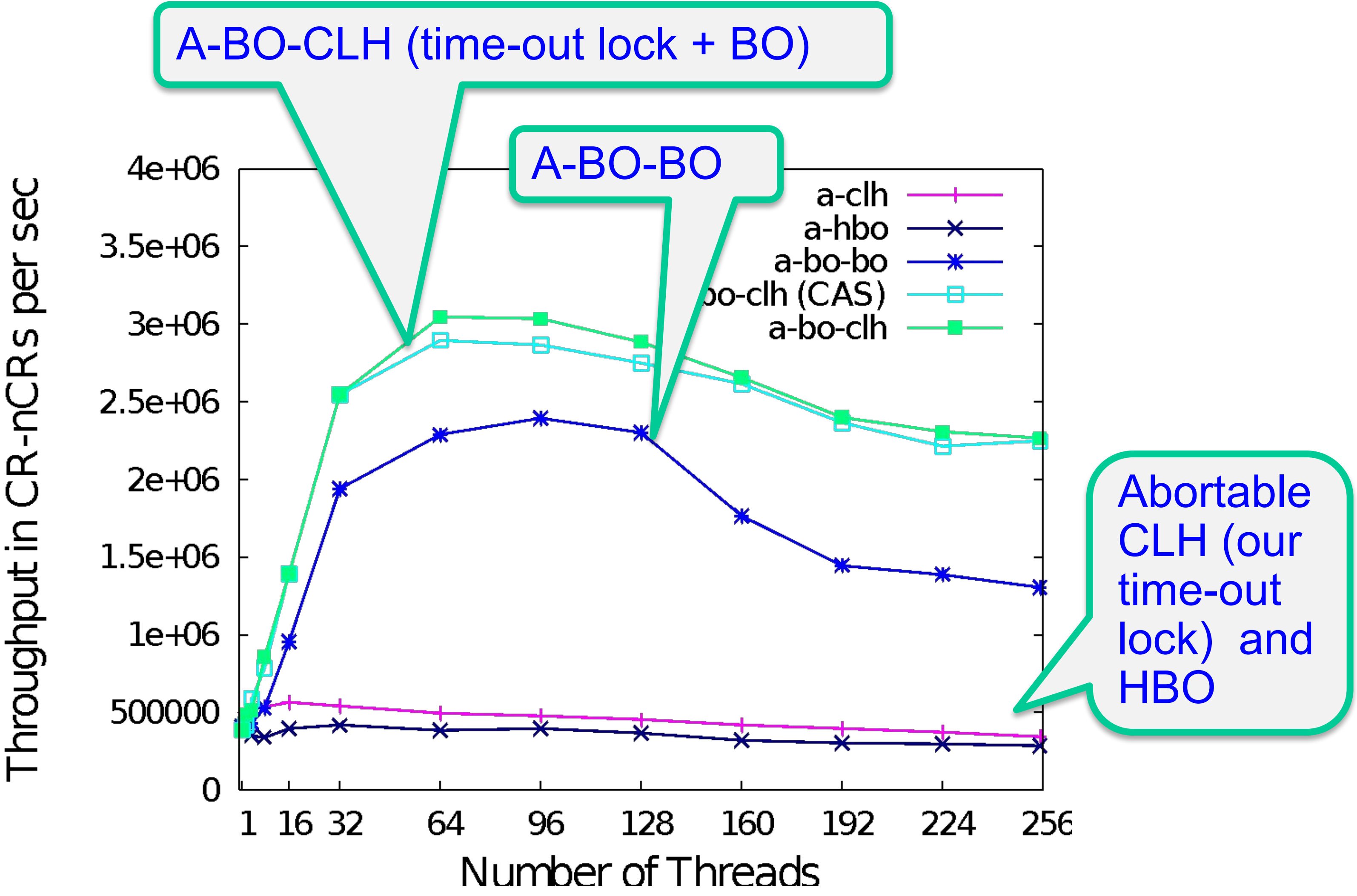
C-BO-MCS

C-BO-BO

HCLH

HBO

Time-Out (Abortable) Lock Cohorting



A-BO-CLH (time-out lock + BO)

A-BO-BO

Abortable CLH (our time-out lock) and HBO

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