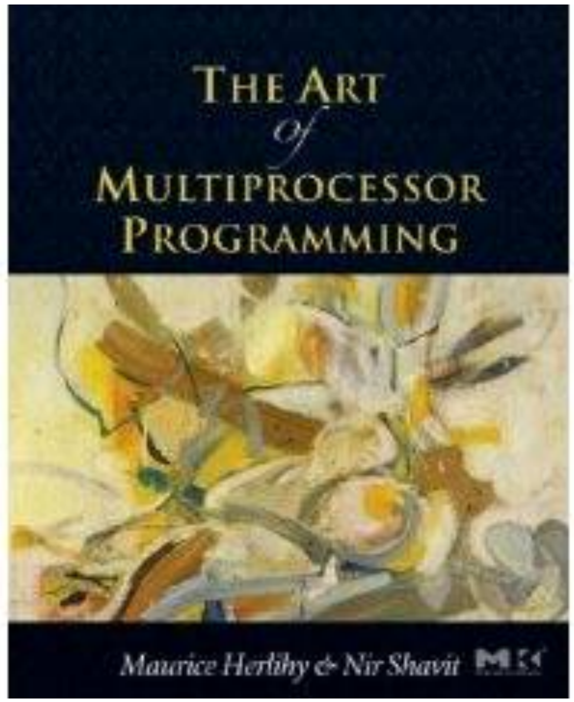
Spin Locks and Contention

Acknowledgement:
Slides adopted from the companion slides for the book
"The Art of Multiprocessor Programming"
by Maurice Herlihy and Nir Shavit

What We'll Cover Today

Chapter 7 of:



Digital copy can be obtained via WUSTL library:

http://catalog.wustl.edu/search/

Today: Revisit Mutual Exclusion

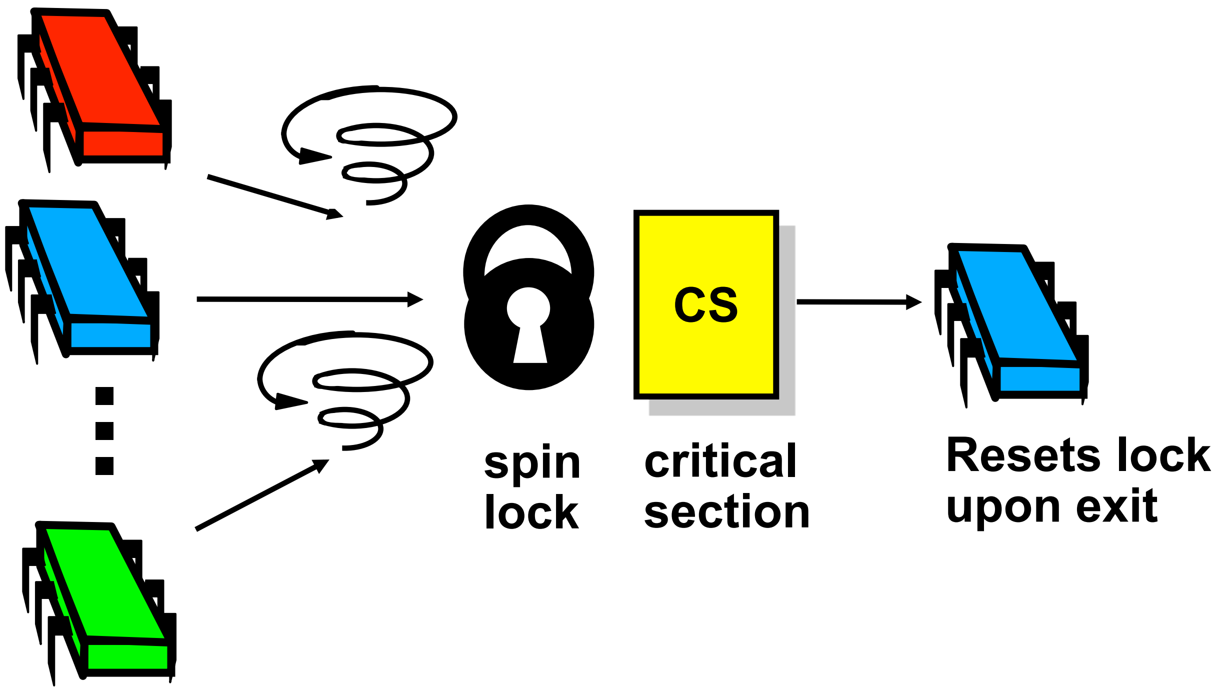
- Performance, not just correctness
- Proper use of multiprocessor architectures
- A collection of locking algorithms that are:
  - Elegant (in their fashion)
  - Important (why else would we pay attention)
  - And realistic (your mileage may vary)

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

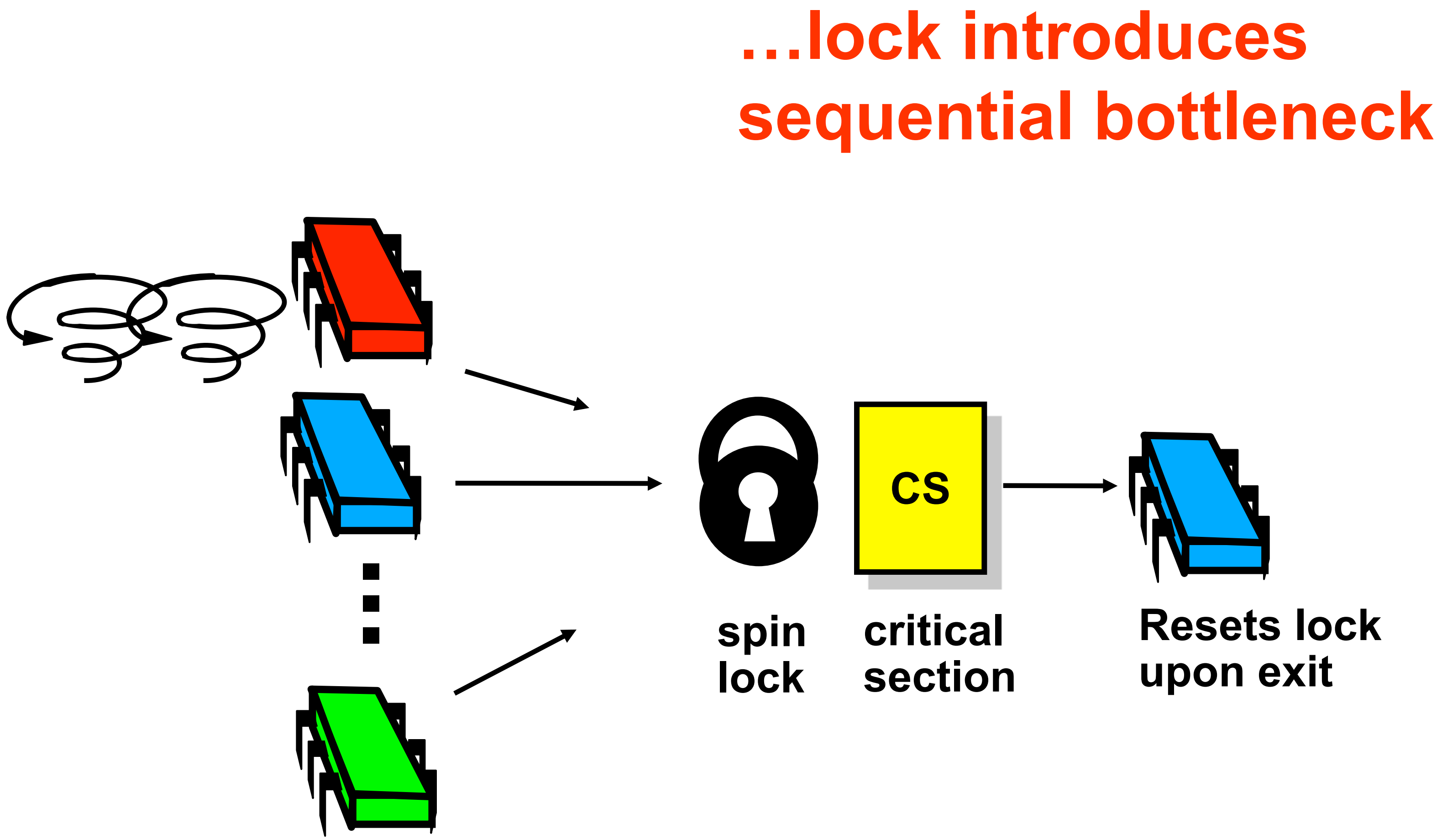
- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
  - Does not make sense on uniprocessor
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

our focus

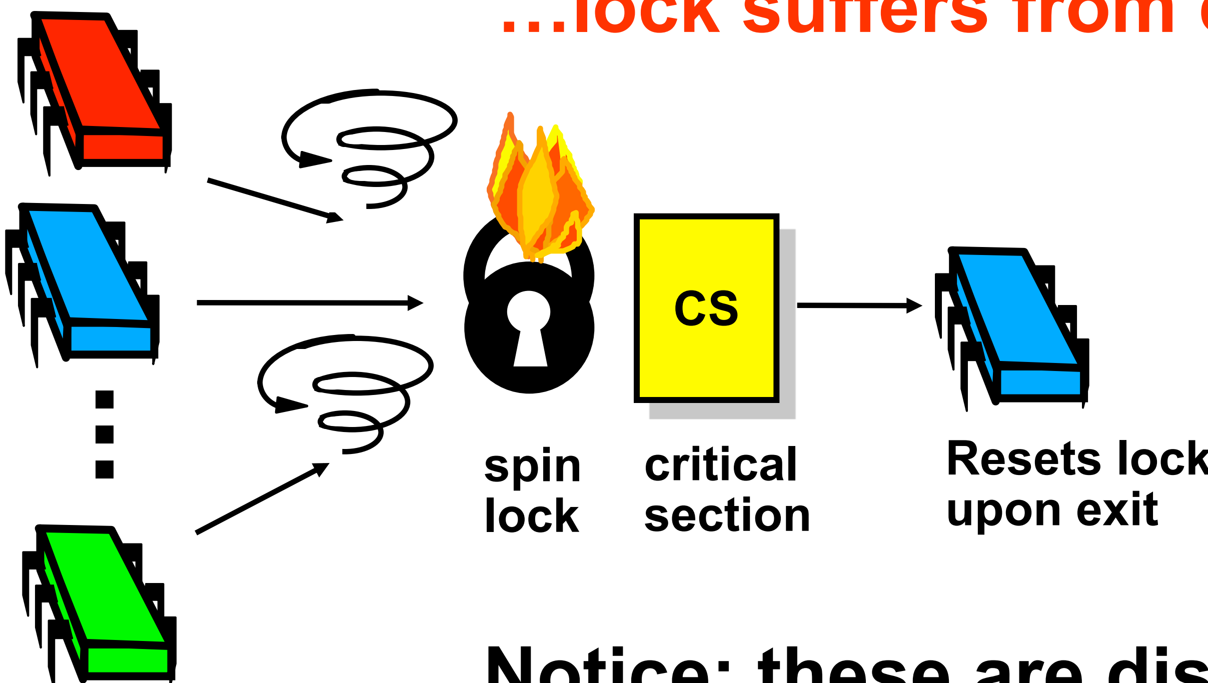
Basic Spin-Lock



Basic Spin-Lock



Basic Spin-Lock



…lock introduces sequential bottleneck

…lock suffers from contention

Notice: these are distinct phenomena

Test-and-Set Locks
Test-and-Test-and-Set Locks

Primitive: Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Atomically swap **true** with current value
  - Return value tells if prior value was **true** or **false**
- Can reset just by writing **false**
- In Java: TAS aka "getAndSet"

Test-and-Set Using AtomicBoolean

```
public class AtomicBoolean {
 boolean value;

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

Test-and-Set Using AtomicBoolean

```
public class AtomicBoolean {
 boolean value;

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

Package java.util.concurrent.atomic

Test-and-Set Using AtomicBoolean

```
public class AtomicBoolean {
 boolean value;

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

Atomically swap old and new values
(This is not how getAndSet is actually implemented.)

Test-and-Set Using AtomicBoolean

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

Test-and-Set Using AtomicBoolean

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

Swapping in true is called "test-and-set" or TAS

An Aside, Recall the Real World

- To prove correctness of locking protocols, we typically assume sequential consistency.
- No modern-day processor implements sequential consistency.
- The compiler can reorder the instructions, too (and it usually does)!
- Use of memory fences is sometimes necessary.
- High-level languages tend to provide additional language features for implementing concurrent protocols that would prevent instruction reordering.

Atomics and Volatile Variables in Java

- In Java, can ask compiler to keep a variable up-to-date by declaring it ***volatile***:
  - Inhibits certain reordering, removing from loops, & other "compiler optimizations."
- Java provides a set of Atomic variables as standard library, which provides a set of atomic RMW (read-modify-write) methods, setter, and getter.
  - Accessing an atomic variable is akin to accessing volatile variable.
- The actual rules are more complicates; we will see more about this in a later lecture.

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
 }}
```

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 TA
 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 Ω(n) lower bound?
- We used a RMW (read-modify-write) operation…

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

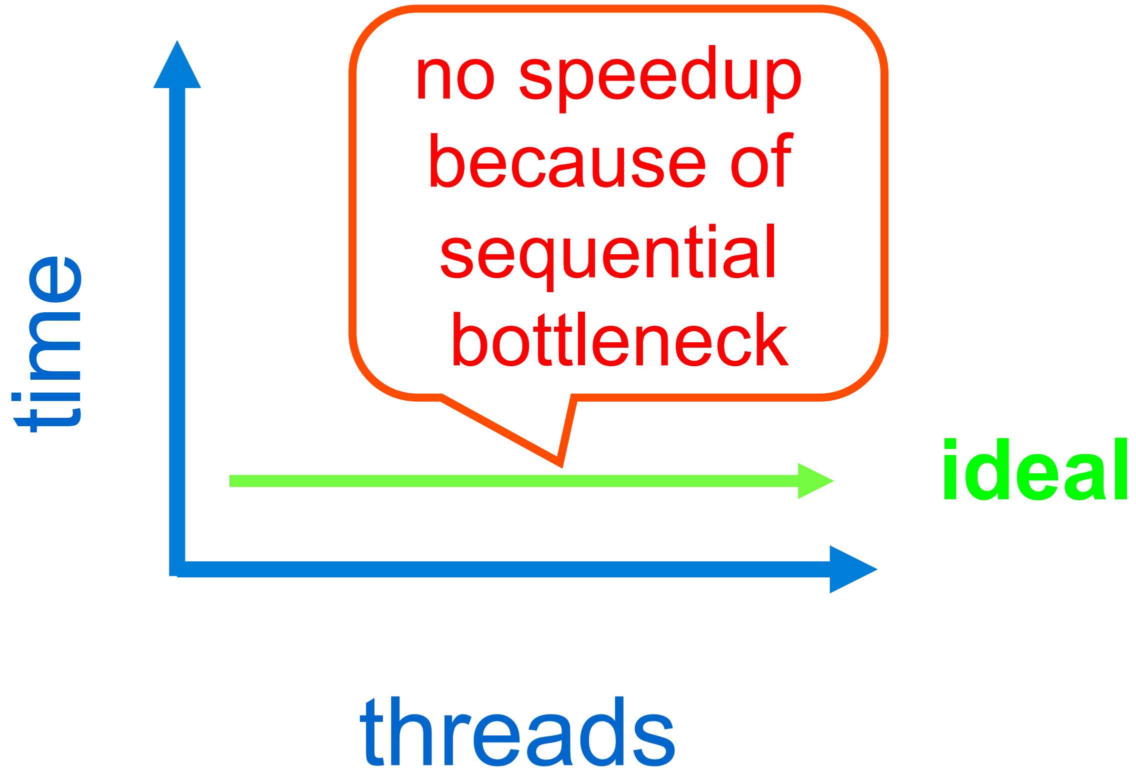
```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

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

Then try to acquire it

Performance

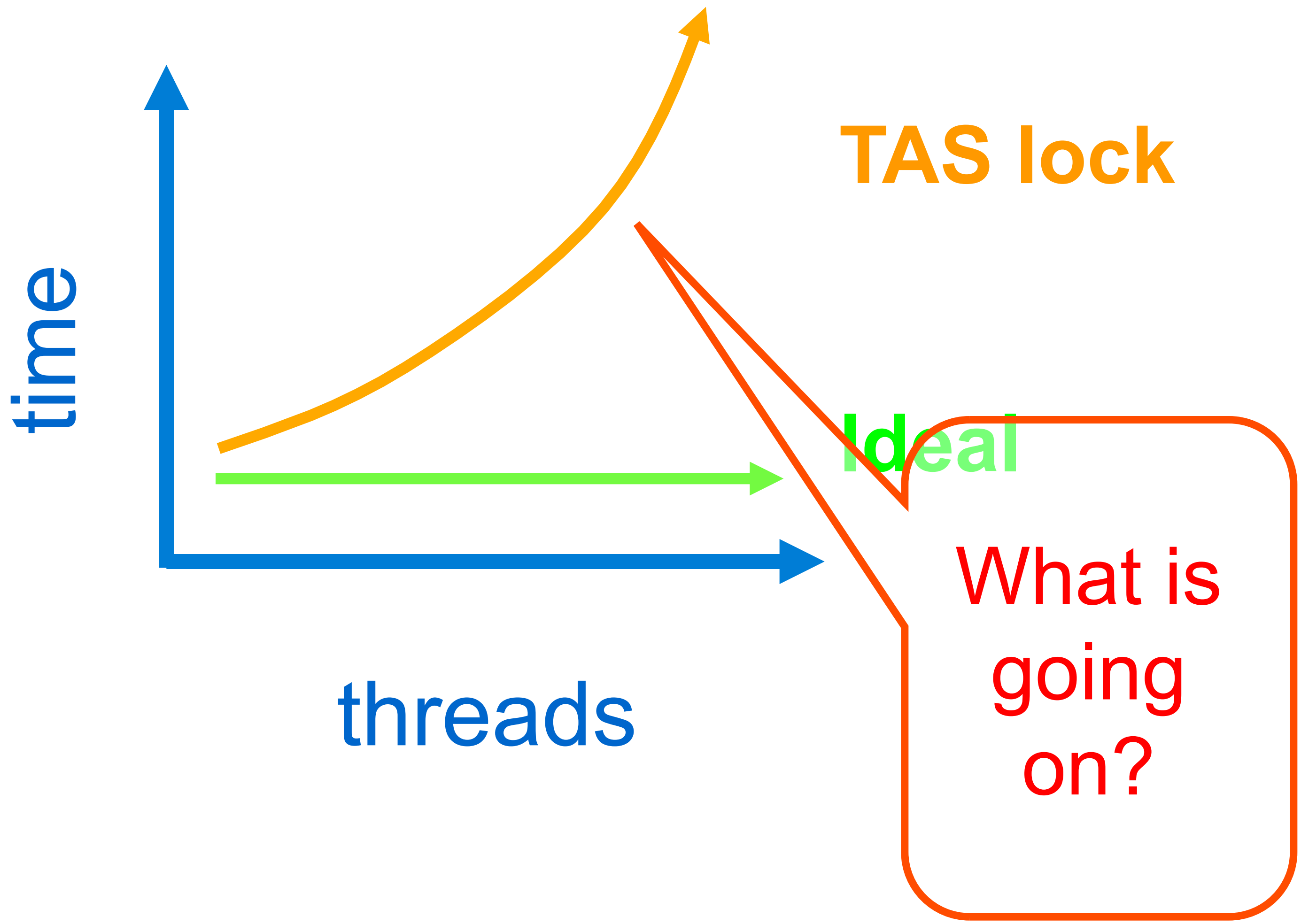
- Experiment
  - *n* threads
  - Increment a shared counter 1 million times (in aggregate across all threads)
- How long should it take?
- How long does it take?

Graph

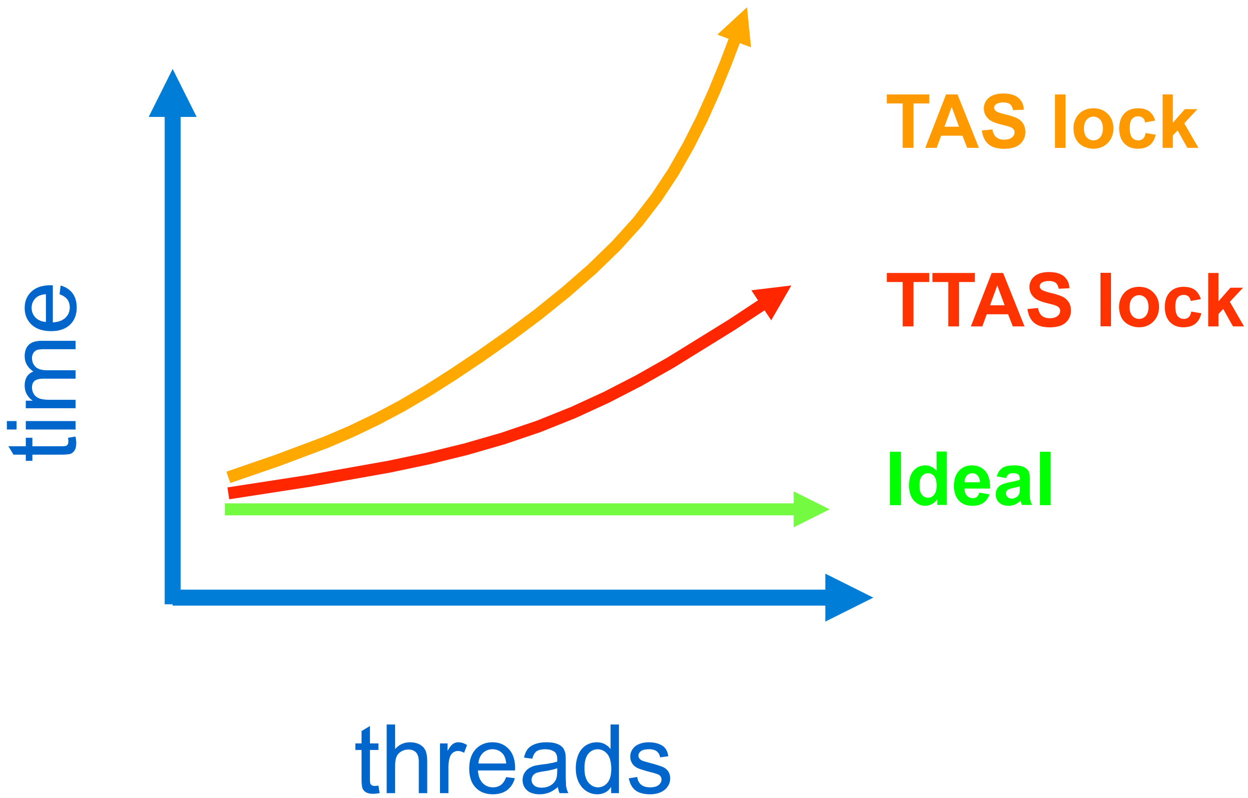


no speedup because of sequential bottleneck
ideal
time
threads

Mystery #1



Mystery #2



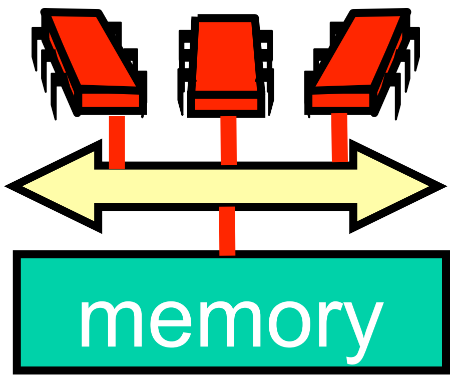
Mystery

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

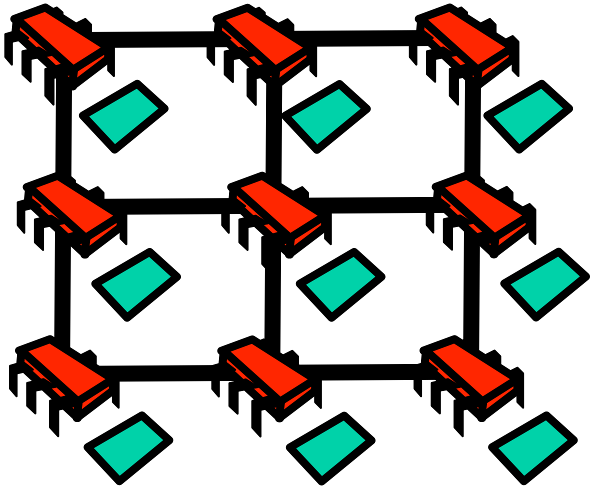
Opinion

- Our memory abstraction is broken
- TAS & TTAS methods
  - Are provably the same (in our model)
  - Except they aren't (in field tests)
- When talking about performance, need to think about underlying architecture.

Basic MIMD Architectures



Shared Bus

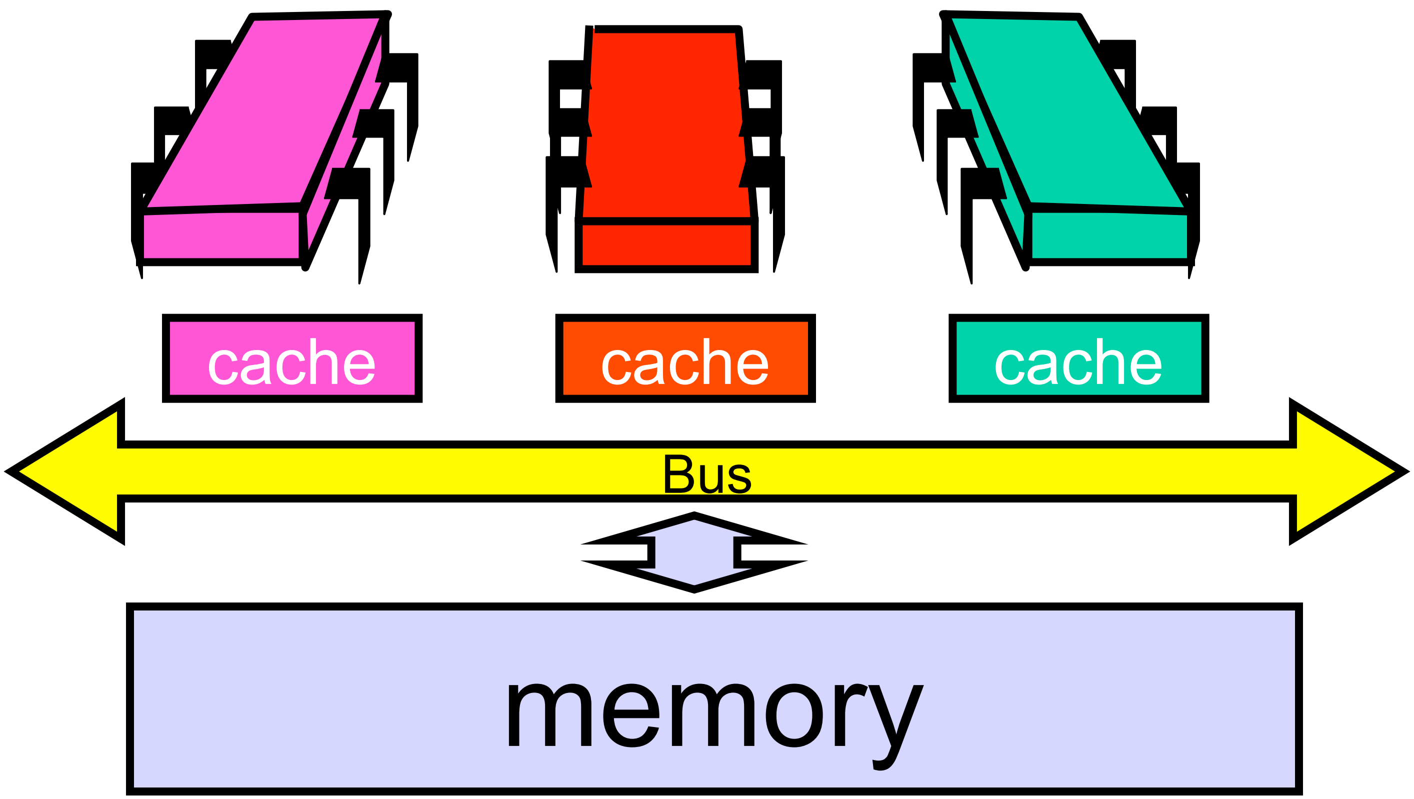


Distributed

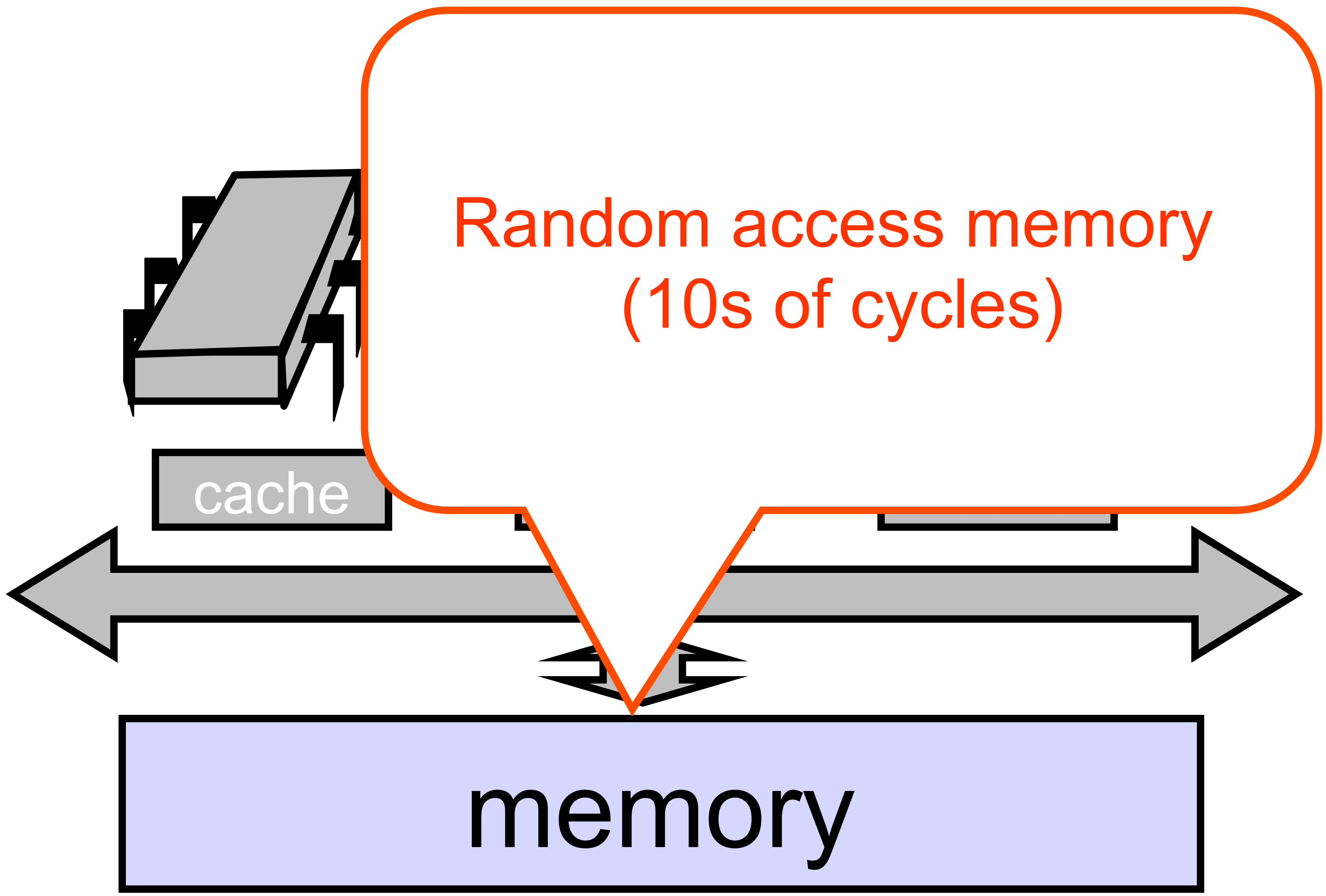
Factors that affect performance:

- Memory Contention
- Communication Contention
- Communication Latency

Bus-Based Architectures



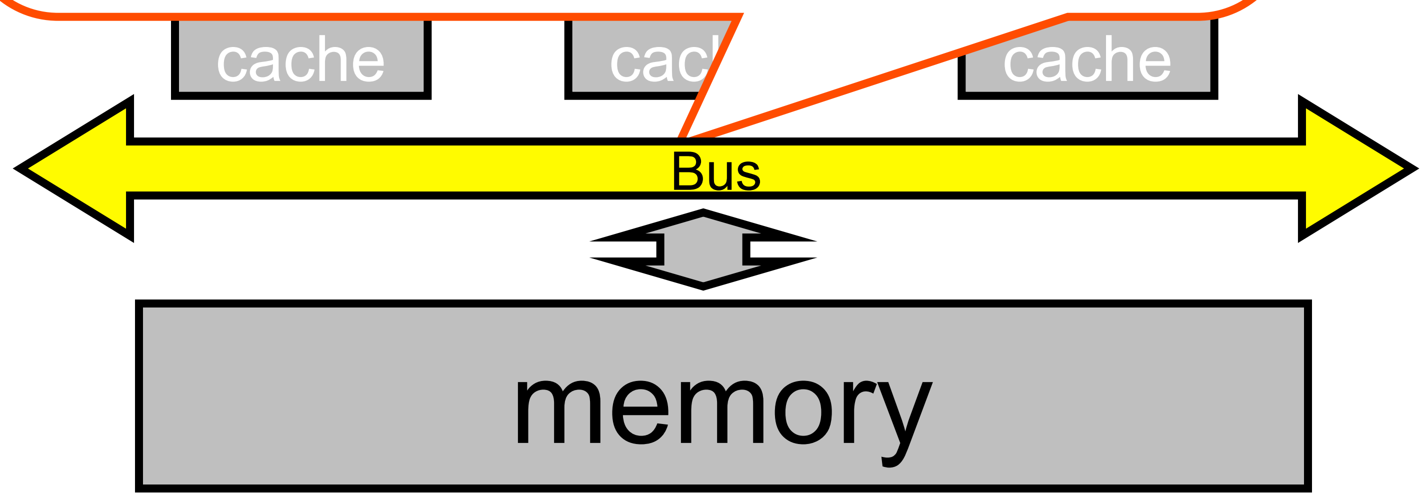
Bus-Based Architectures



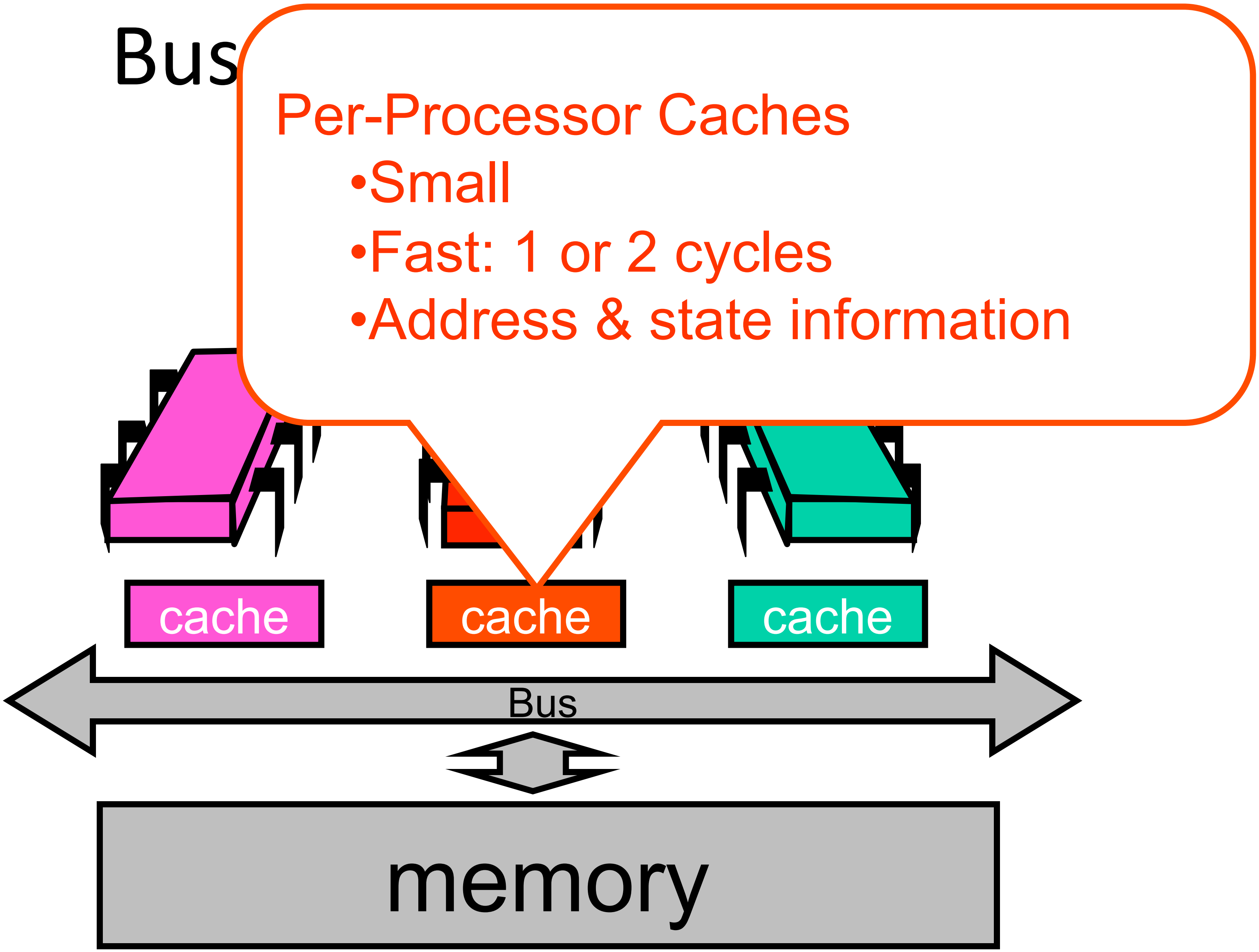
Bus-Based Architectures

Shared Bus

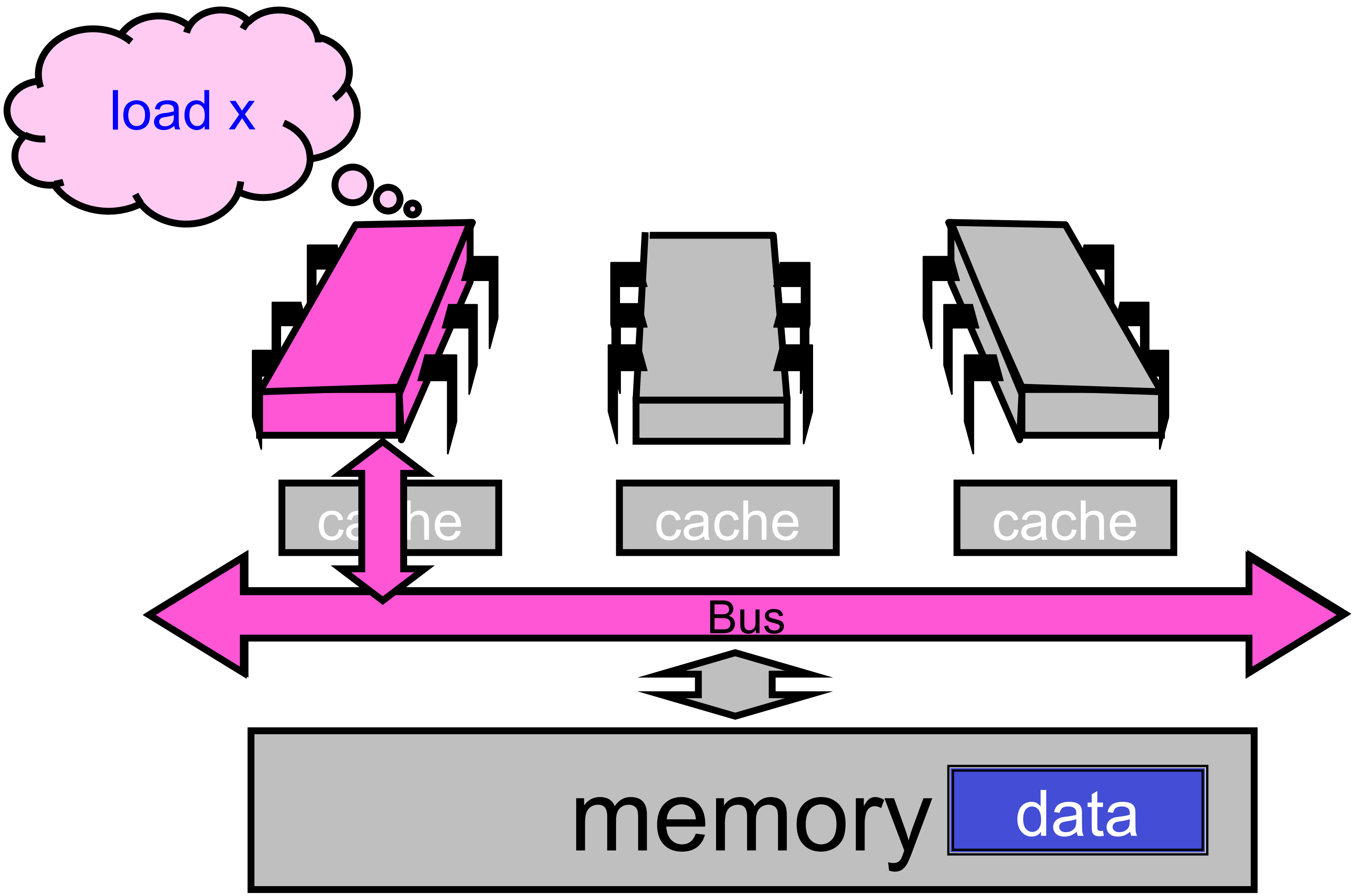
- Broadcast medium
- One broadcaster at a time
- Processors and memory all "snoop"



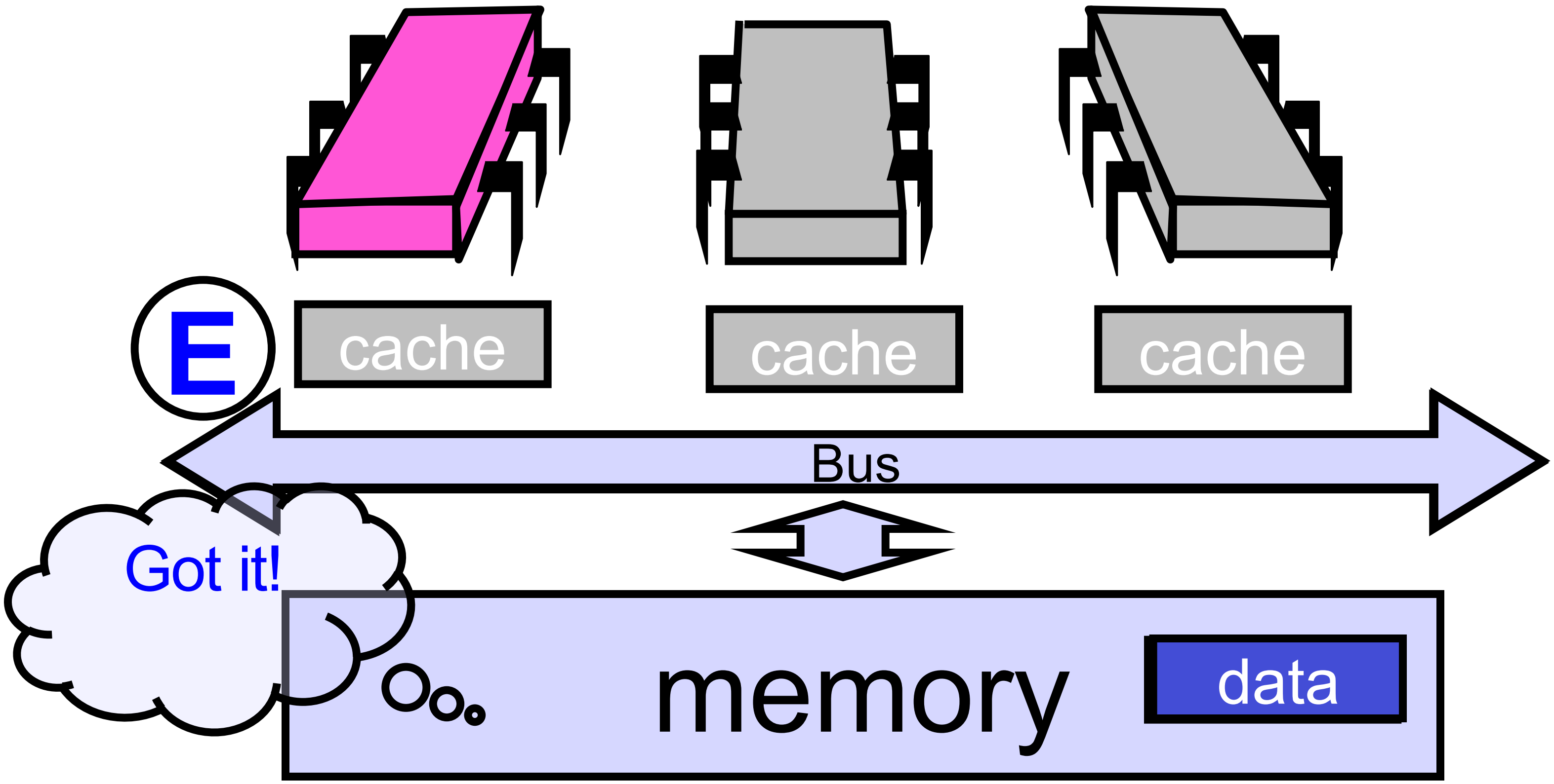
Bus


Per-Processor Caches
•Small
•Fast: 1 or 2 cycles
•Address & state information
cache
cache
cache
Bus
memory

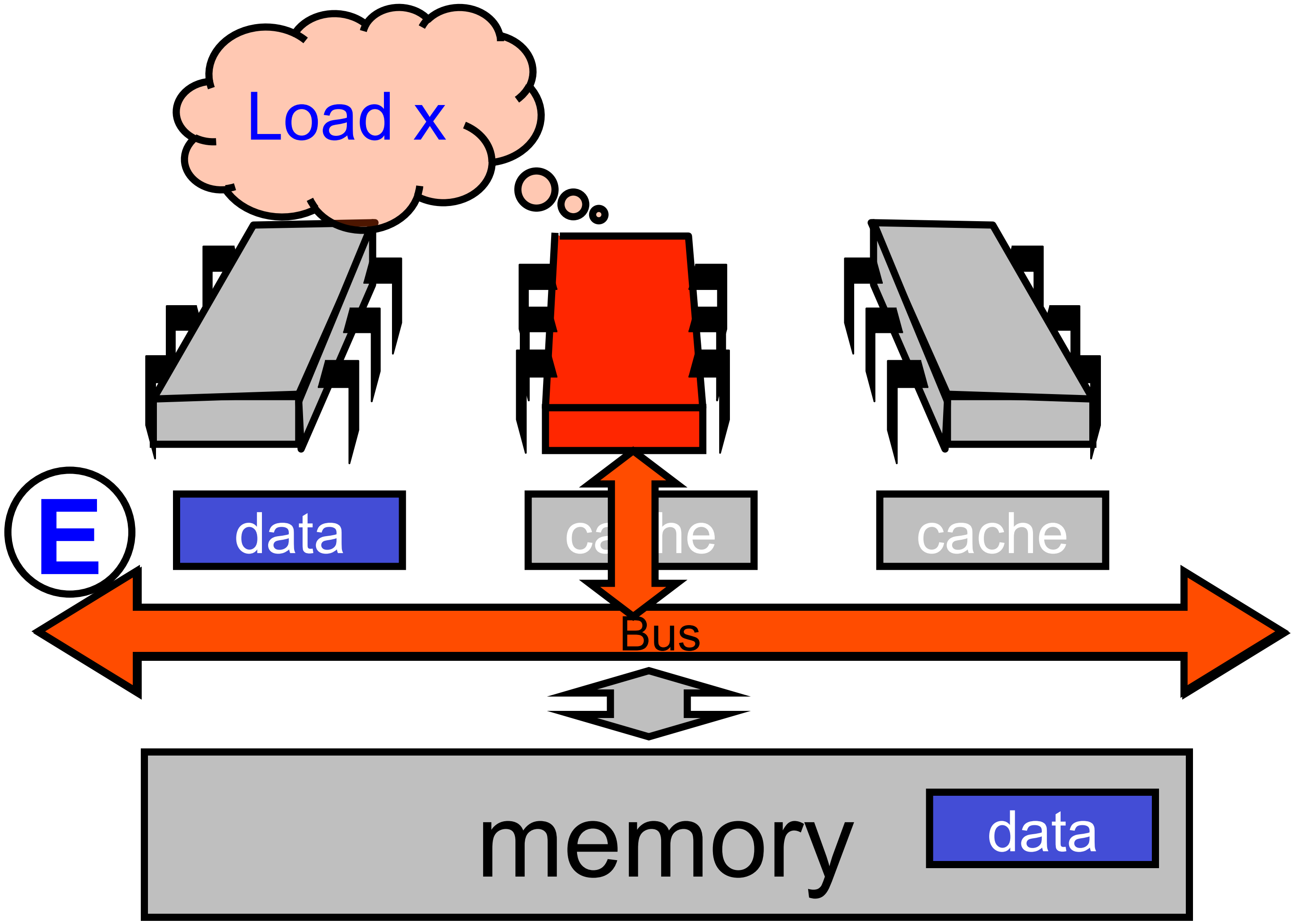
Processor Issues Load Request



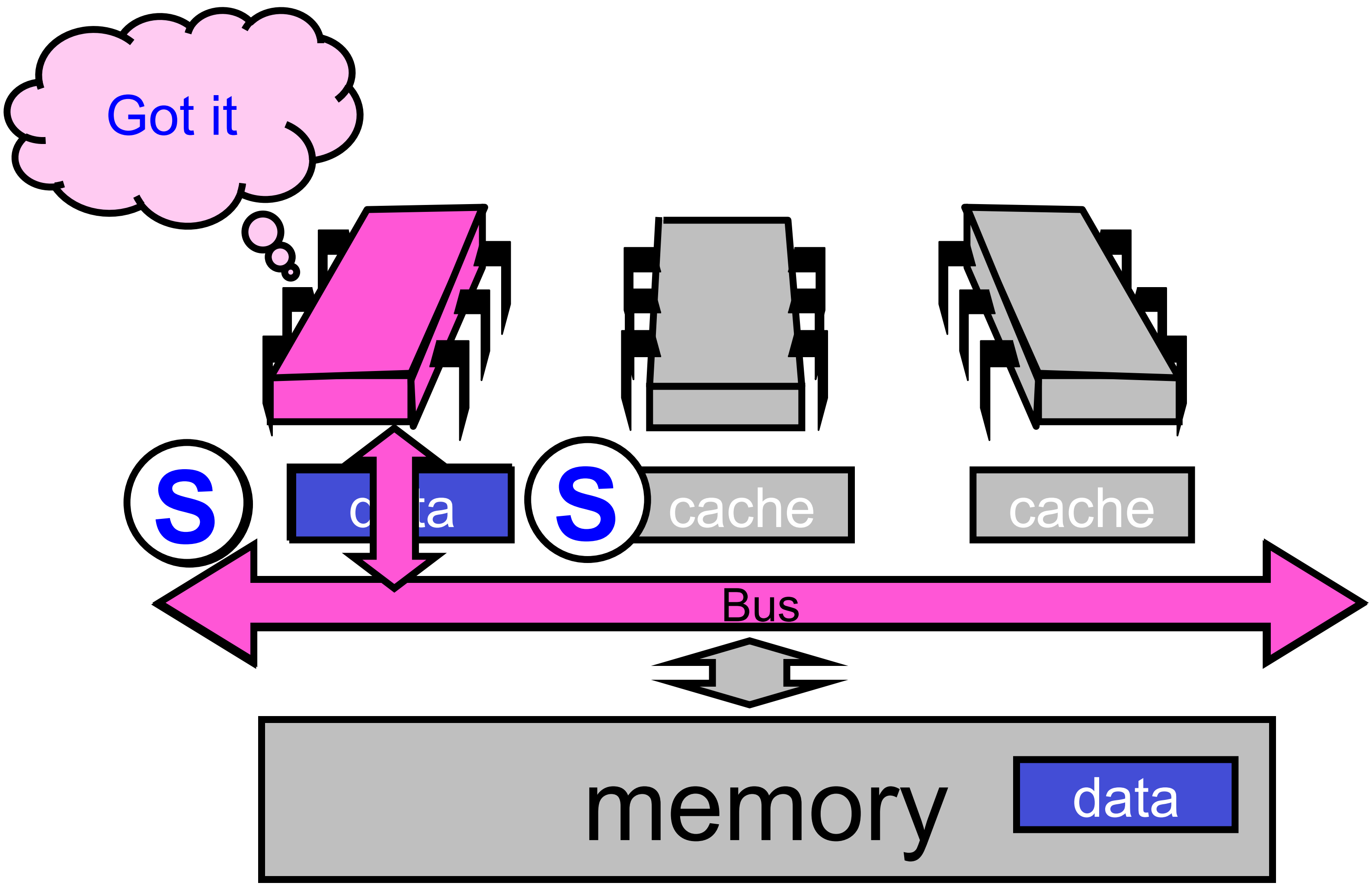
Memory Responds



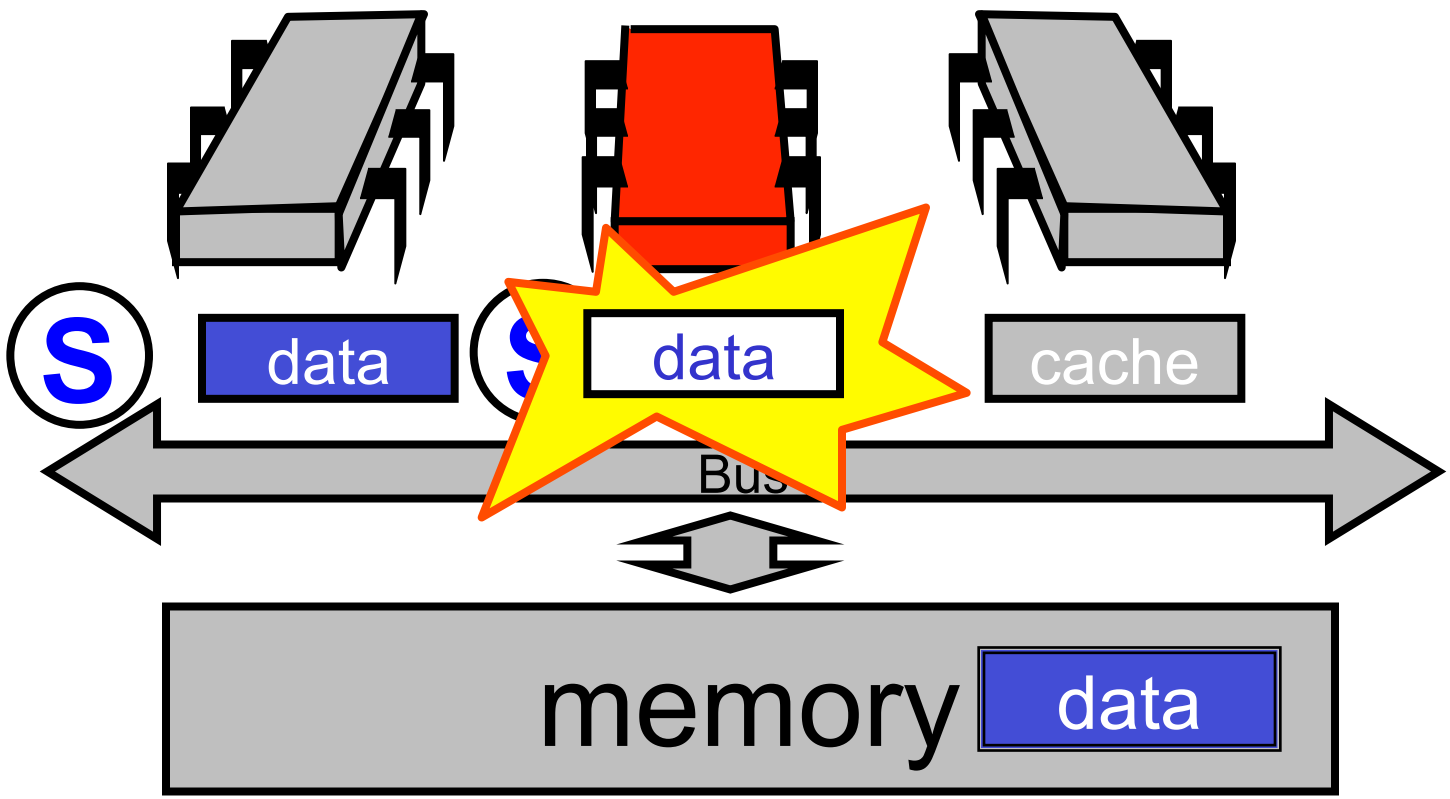
Processor Issues Load Request



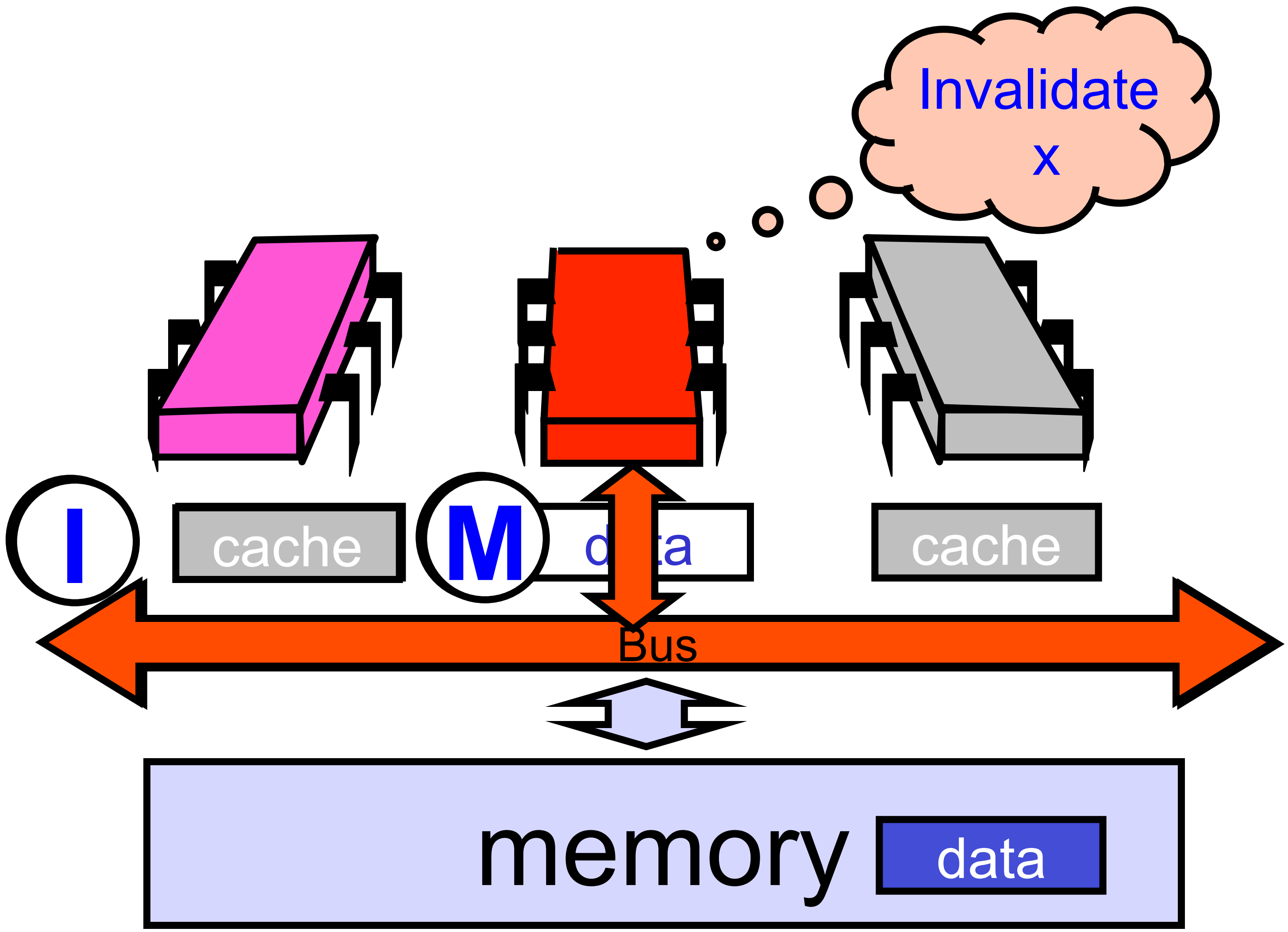
Other Processor Responds



Modify Cached Data



Invalidate



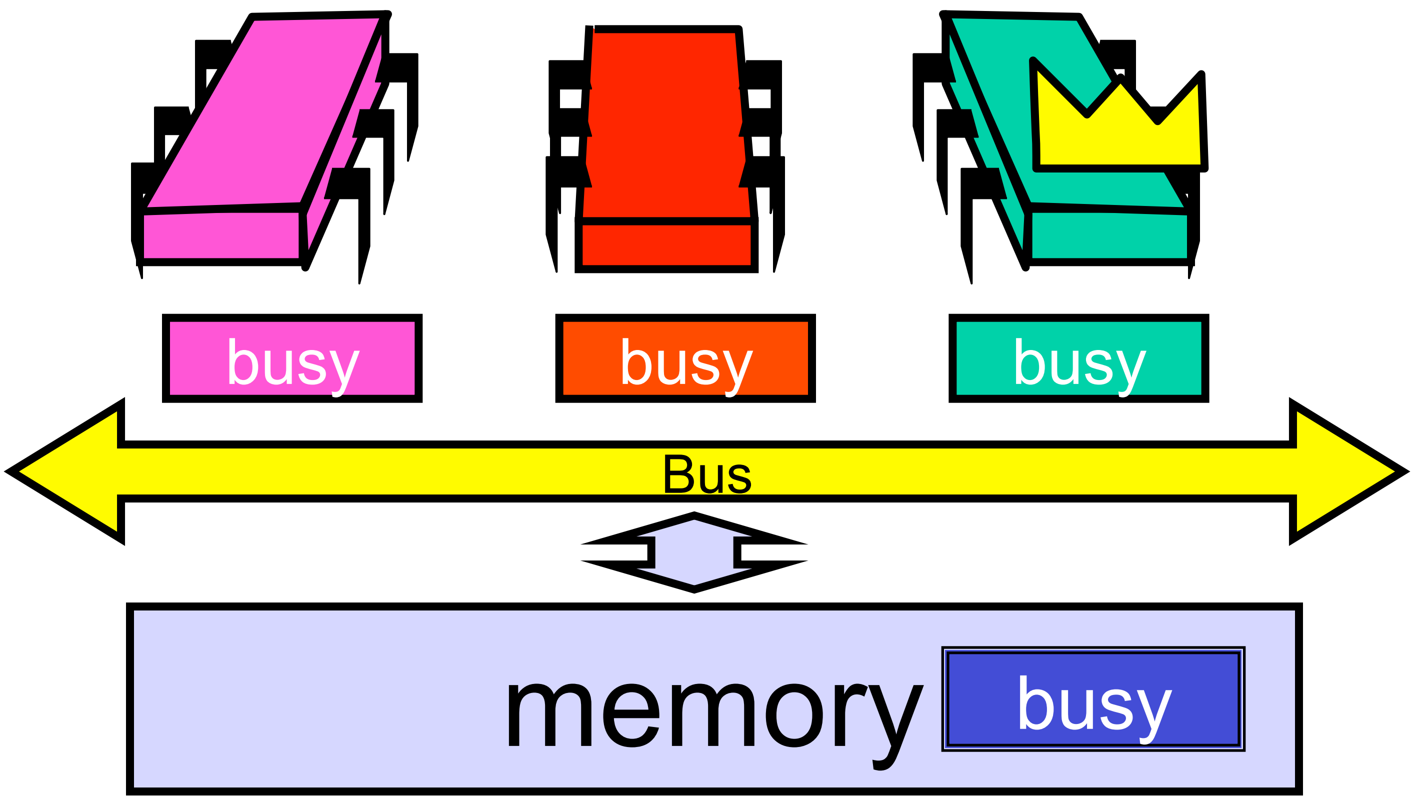
Problems with Simple TASLock

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

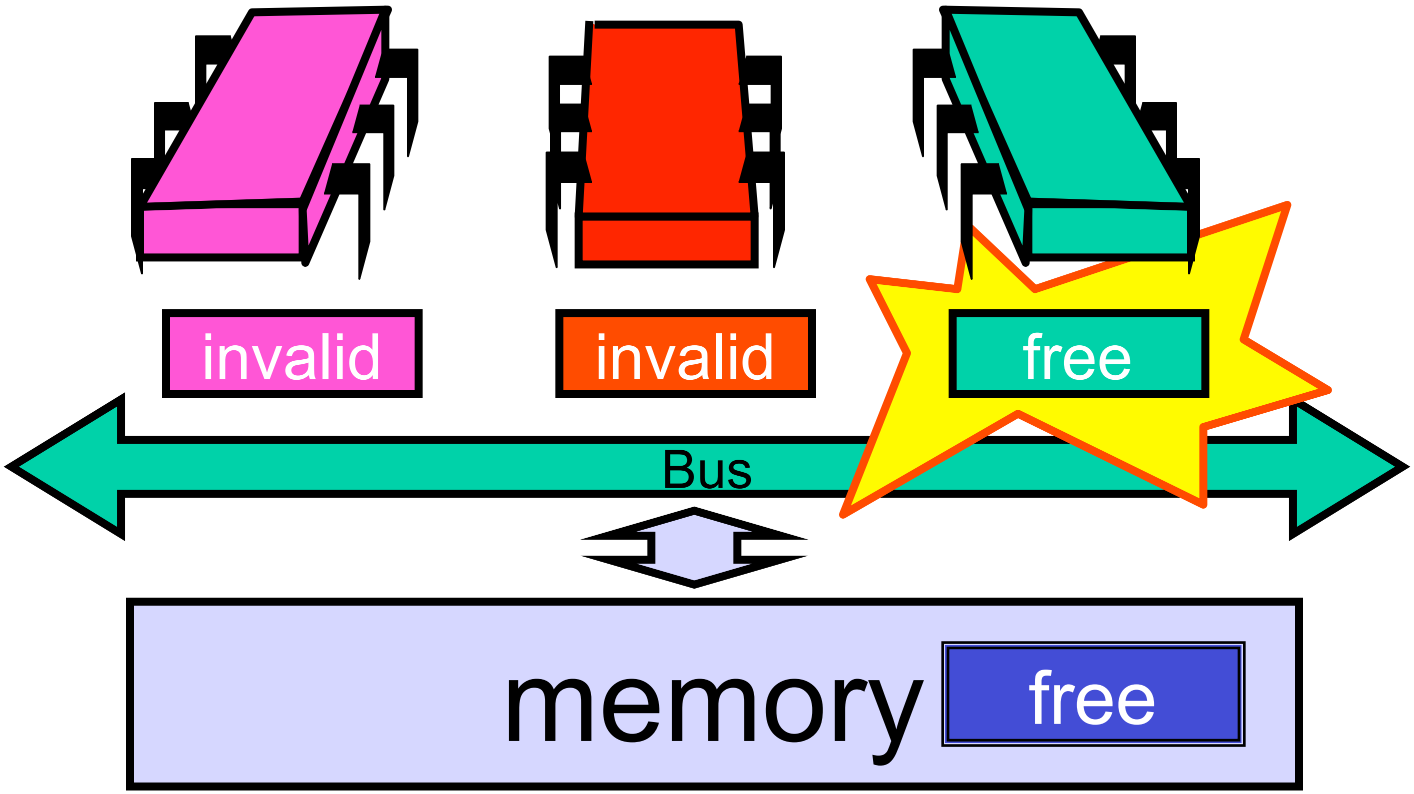
Test-and-Test-and-Set

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

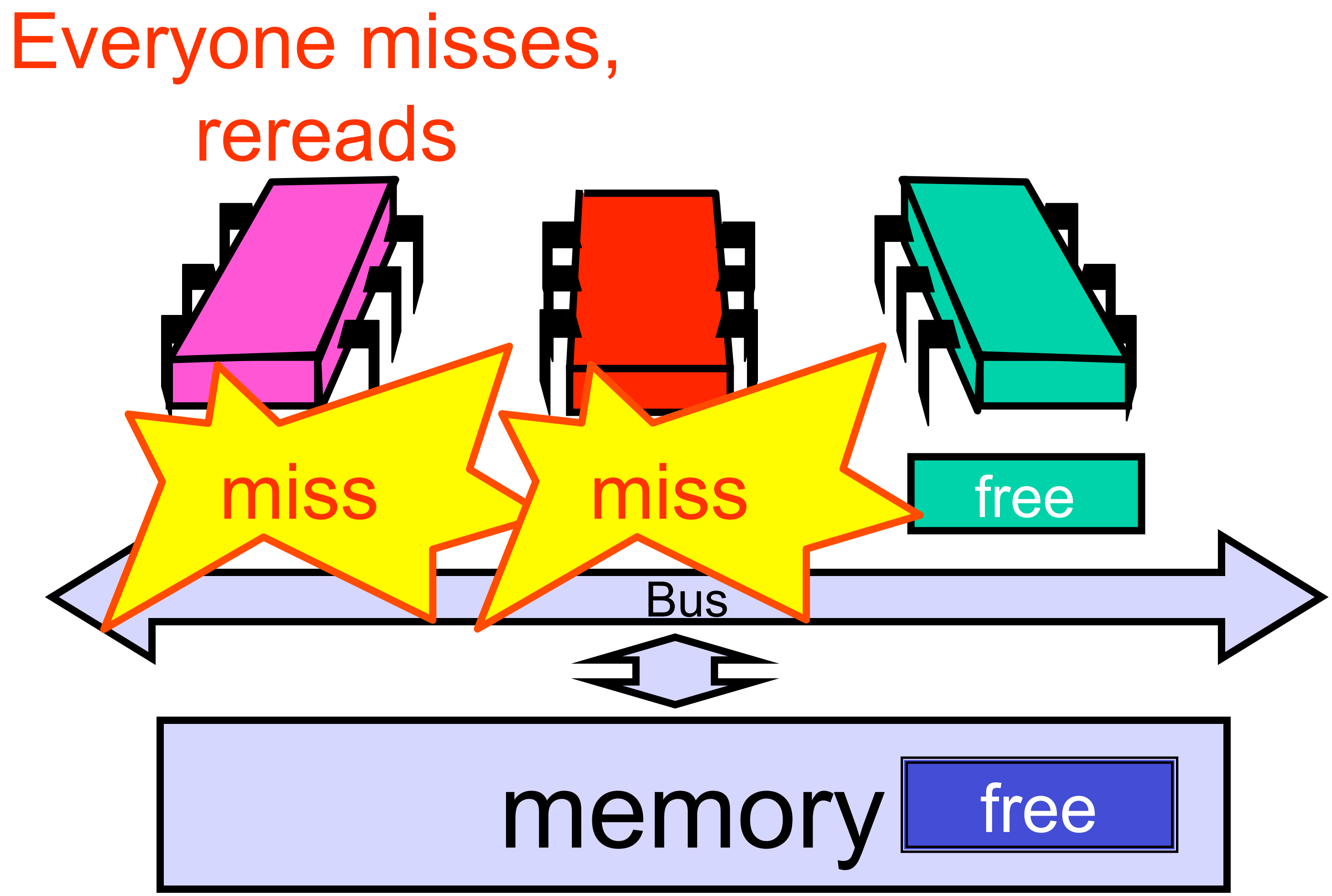
Local Spinning while Lock is Busy



On Release



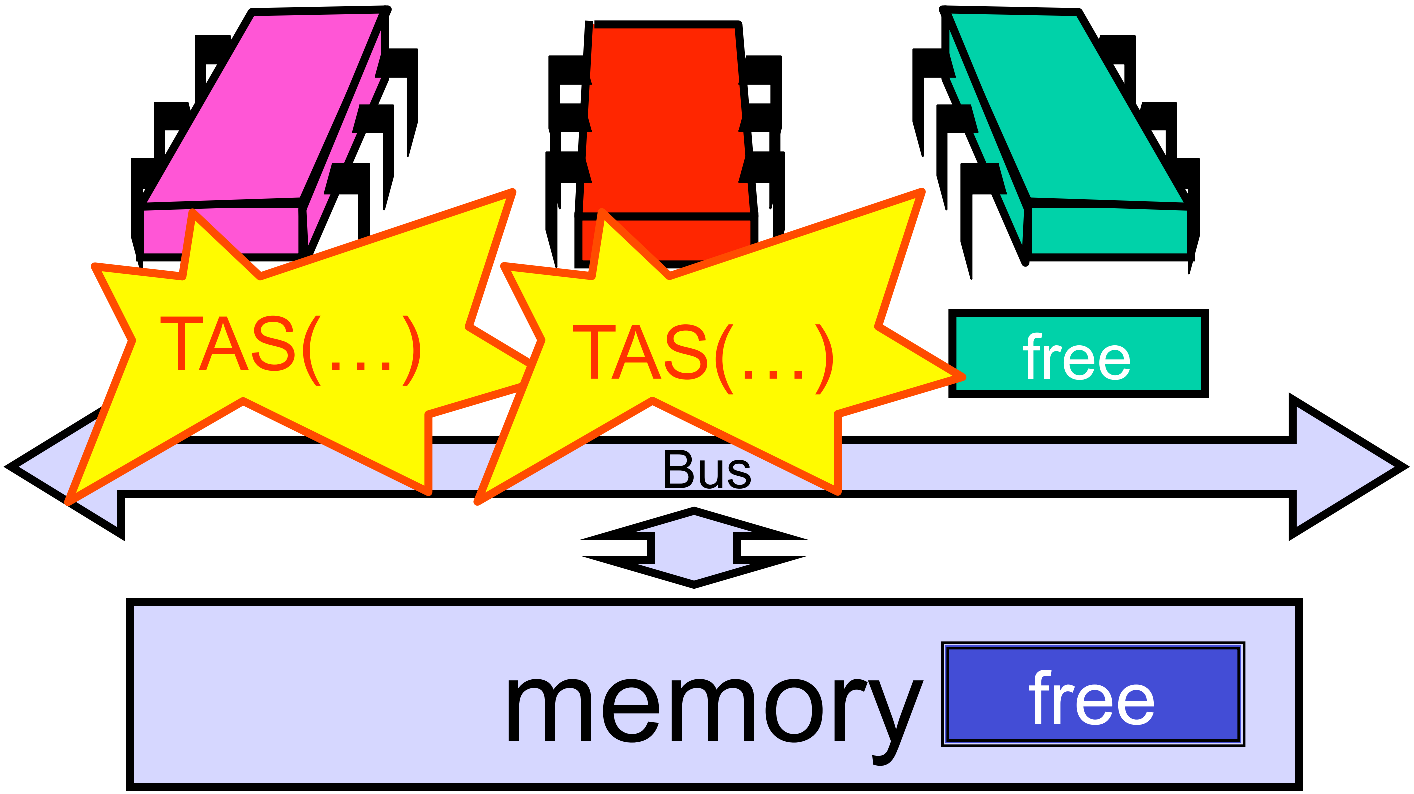
On Release



(1)

On Release

Everyone tries TAS

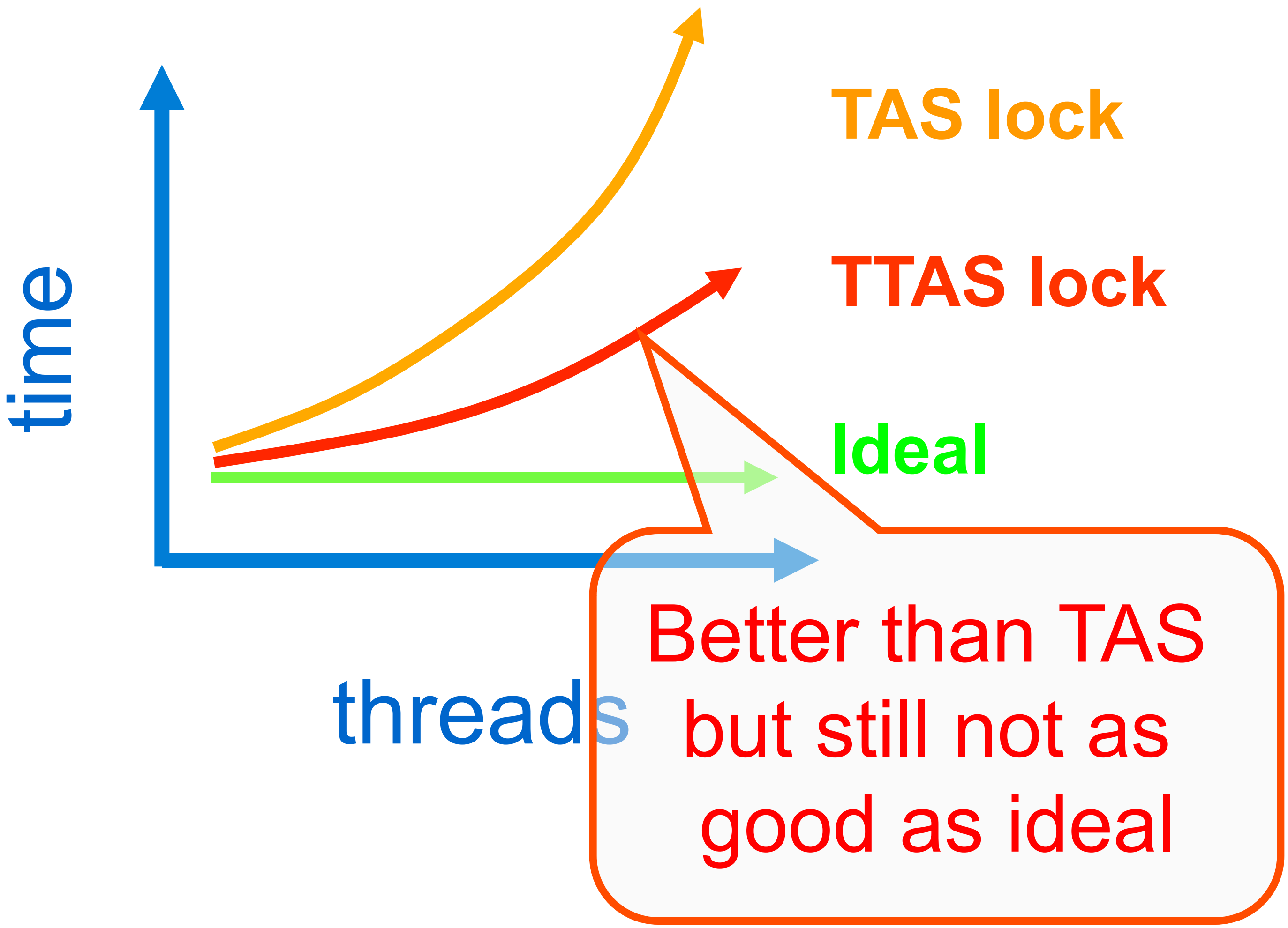


(1)

Problems with Test-and-Test-and-Set

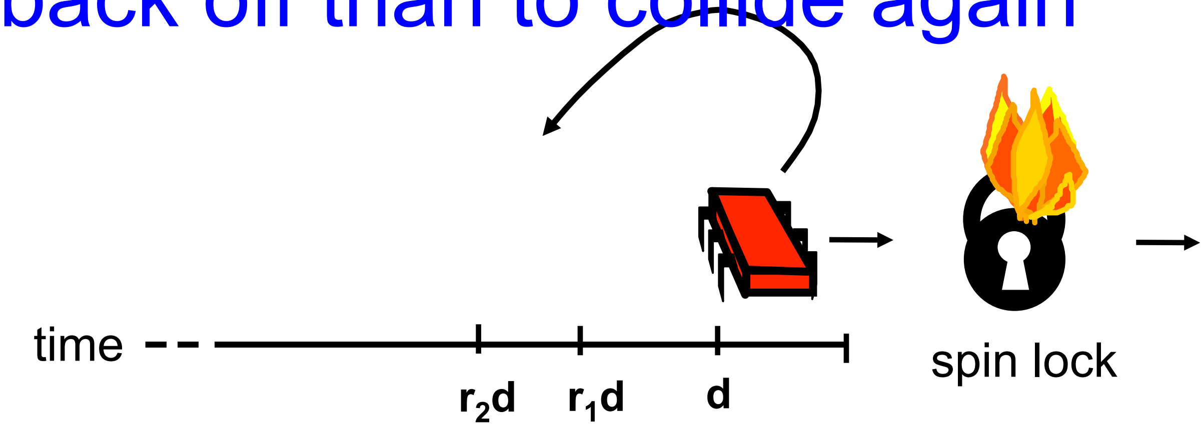
- Upon release everyone incurs misses
- Everyone does TAS
  - Exclusive ownership satisfied sequentially
  - Invalidates others' caches
- Eventually quiesces after lock acquired

Mystery Explained

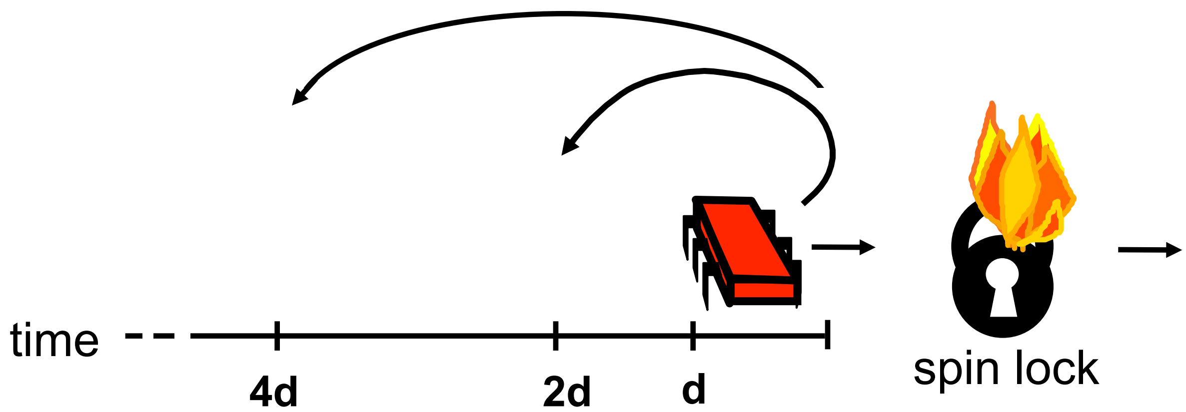


Solution: Introduce Delay

- 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

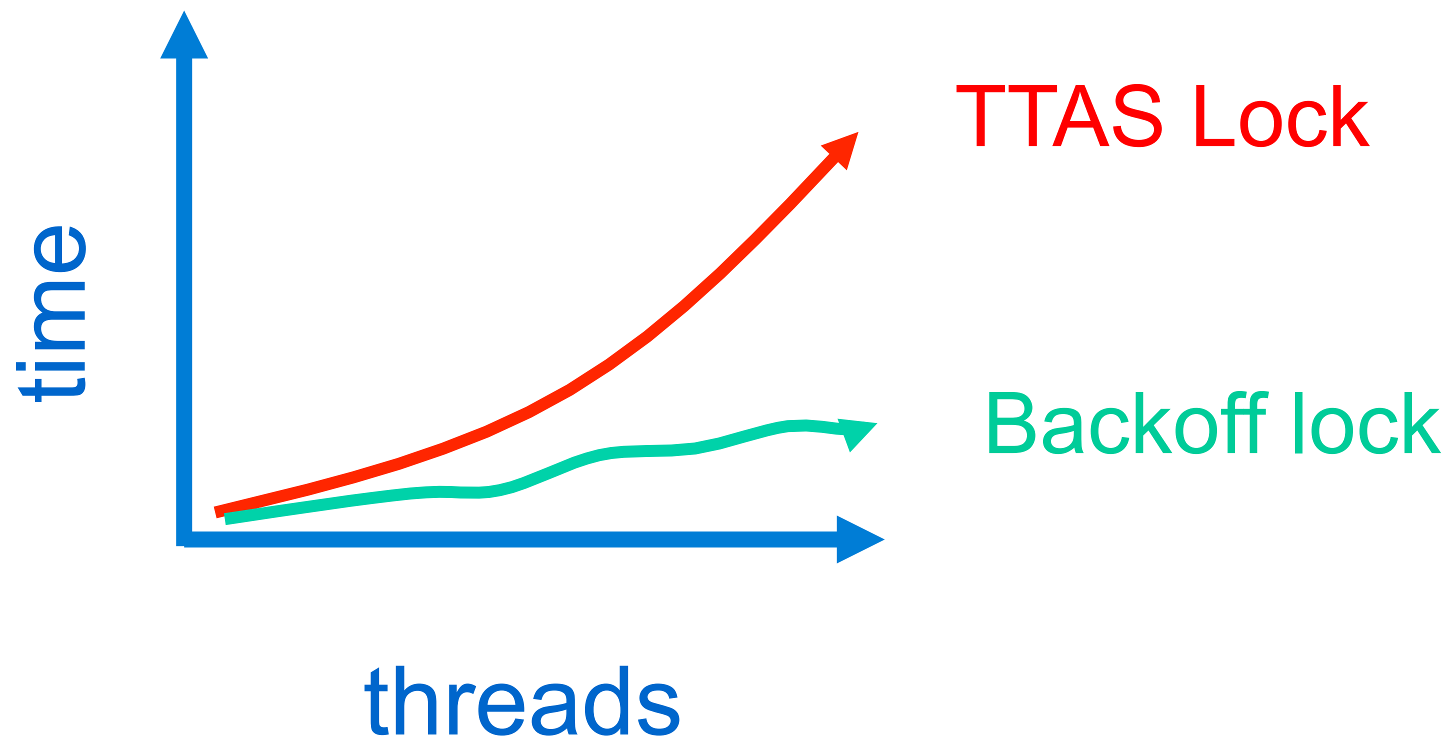
```
public
 publi
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }}}
```

Exponential Backoff Lock

Double max delay, within reason

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

Spin-Waiting Overhead



Backoff: Other Issues

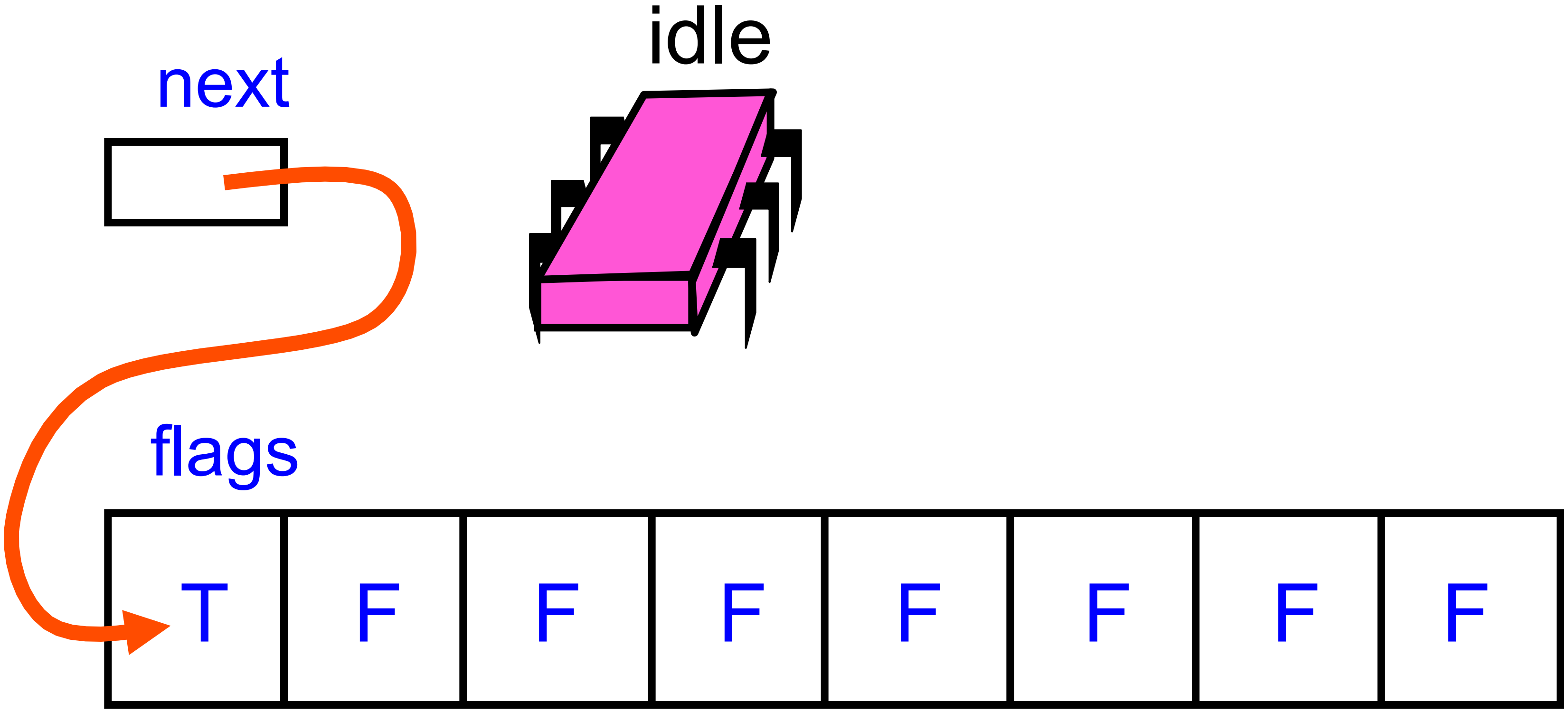
- Good
  - Easy to implement
  - Beats TTAS lock
- Bad
  - Must choose parameters carefully
  - Not portable across platforms

Idea

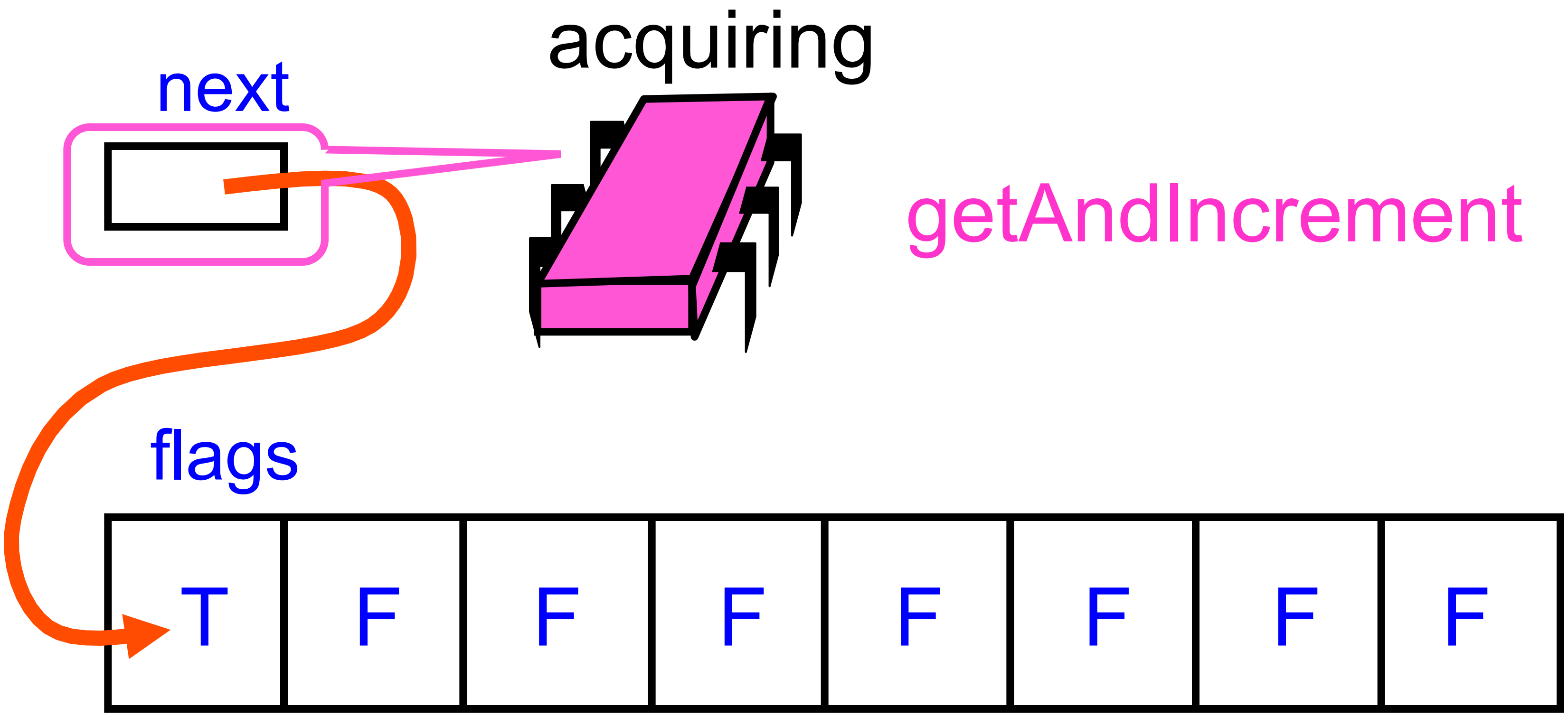
- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others

Anderson Queue Lock

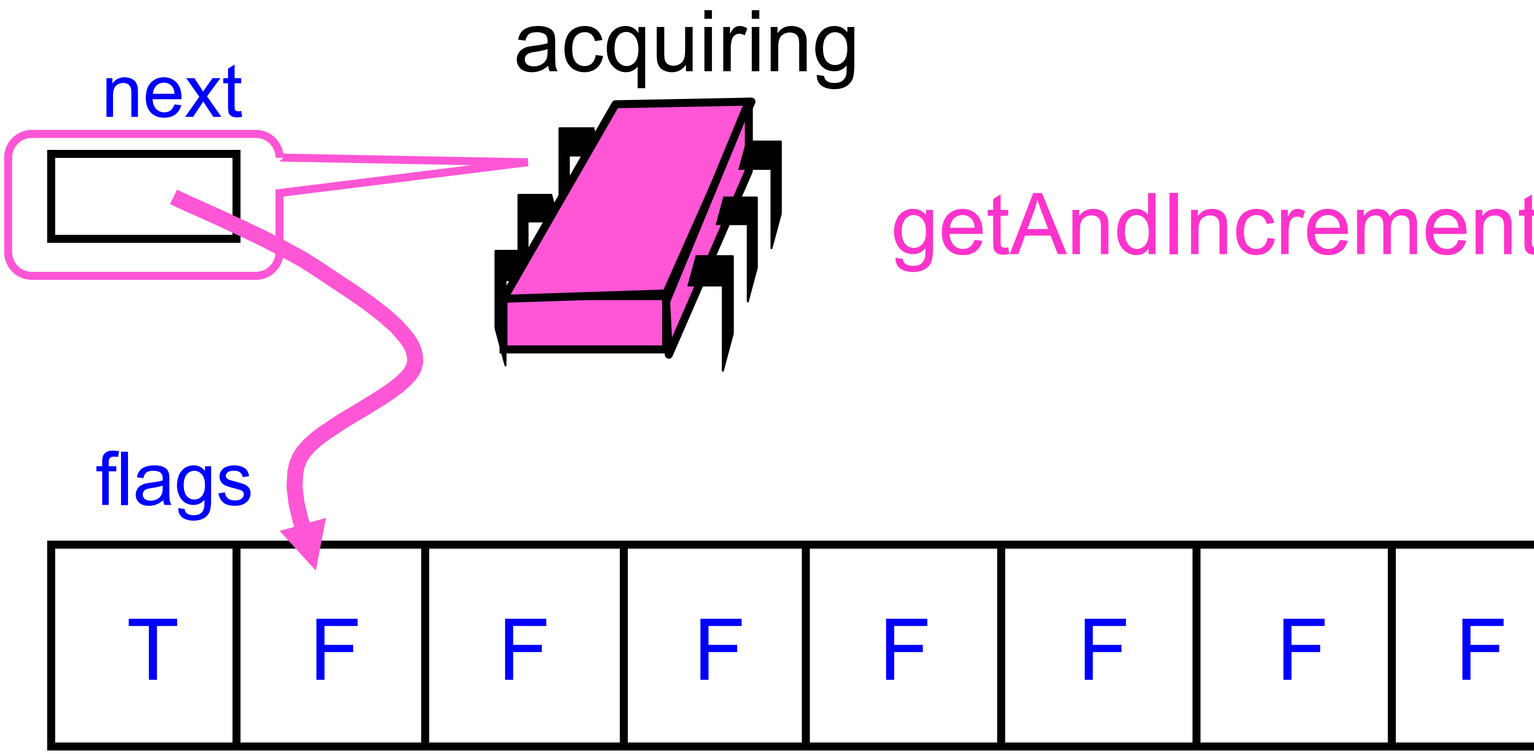
Anderson Queue Lock



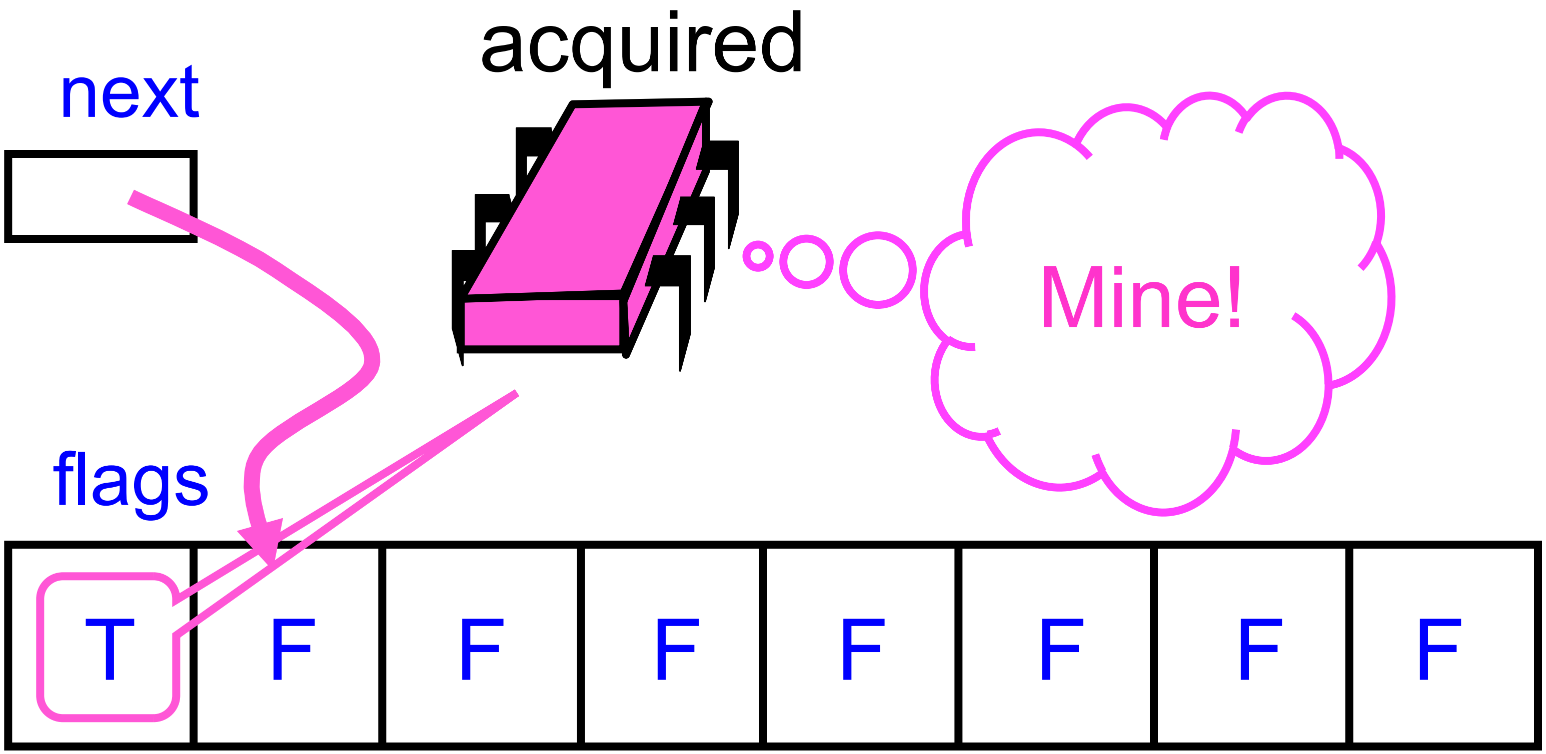
Anderson Queue Lock



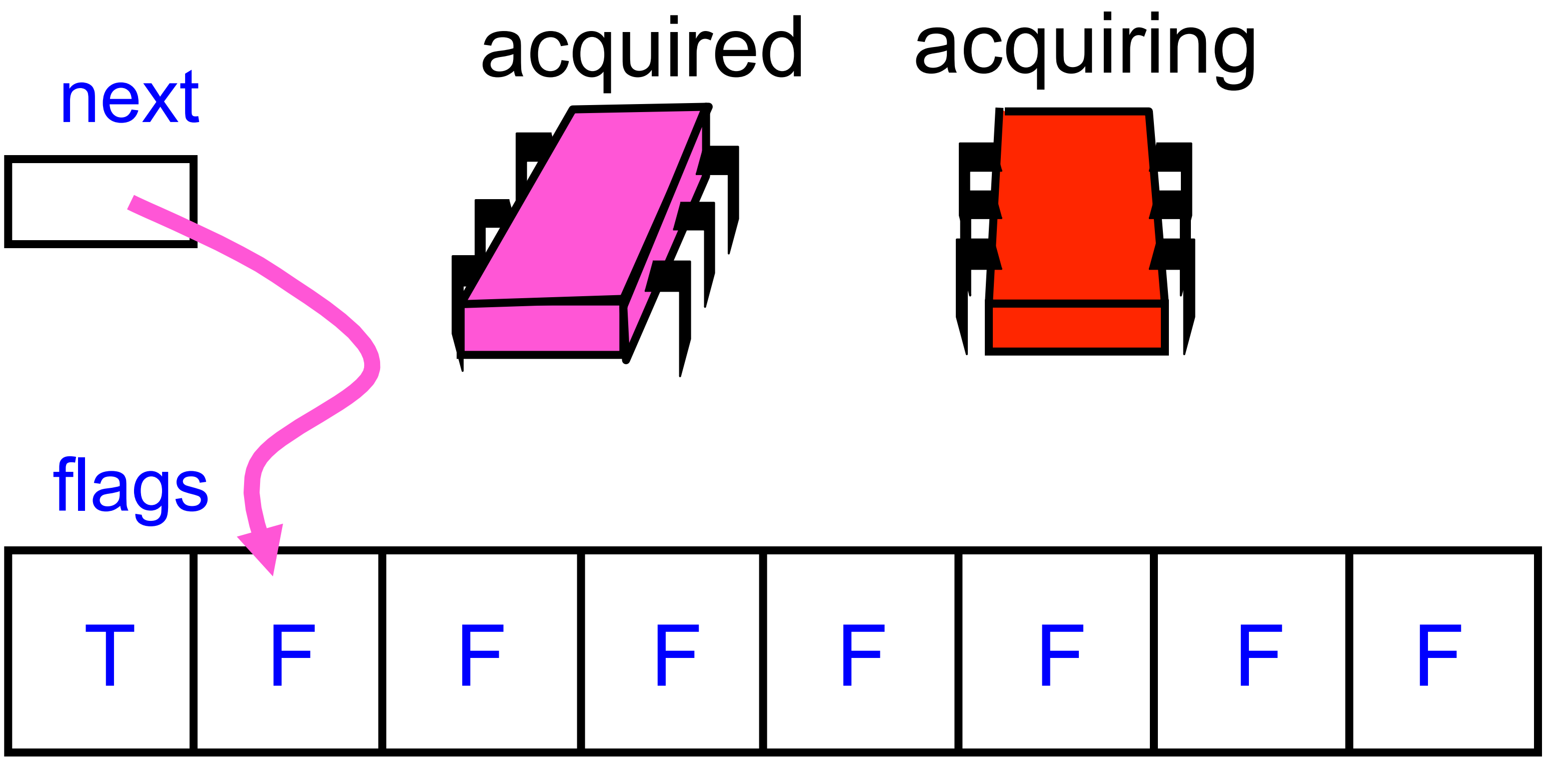
Anderson Queue Lock



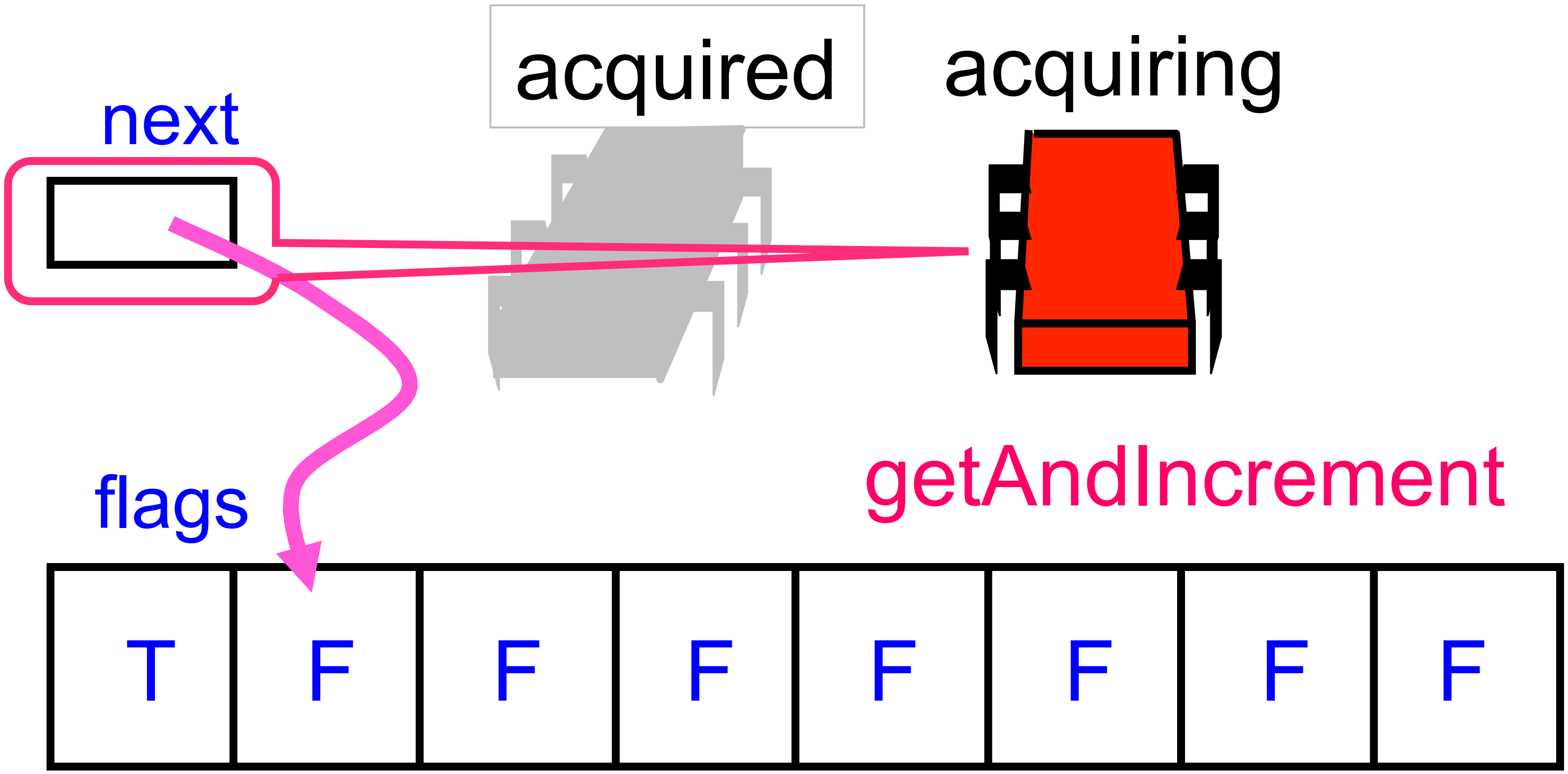
Anderson Queue Lock



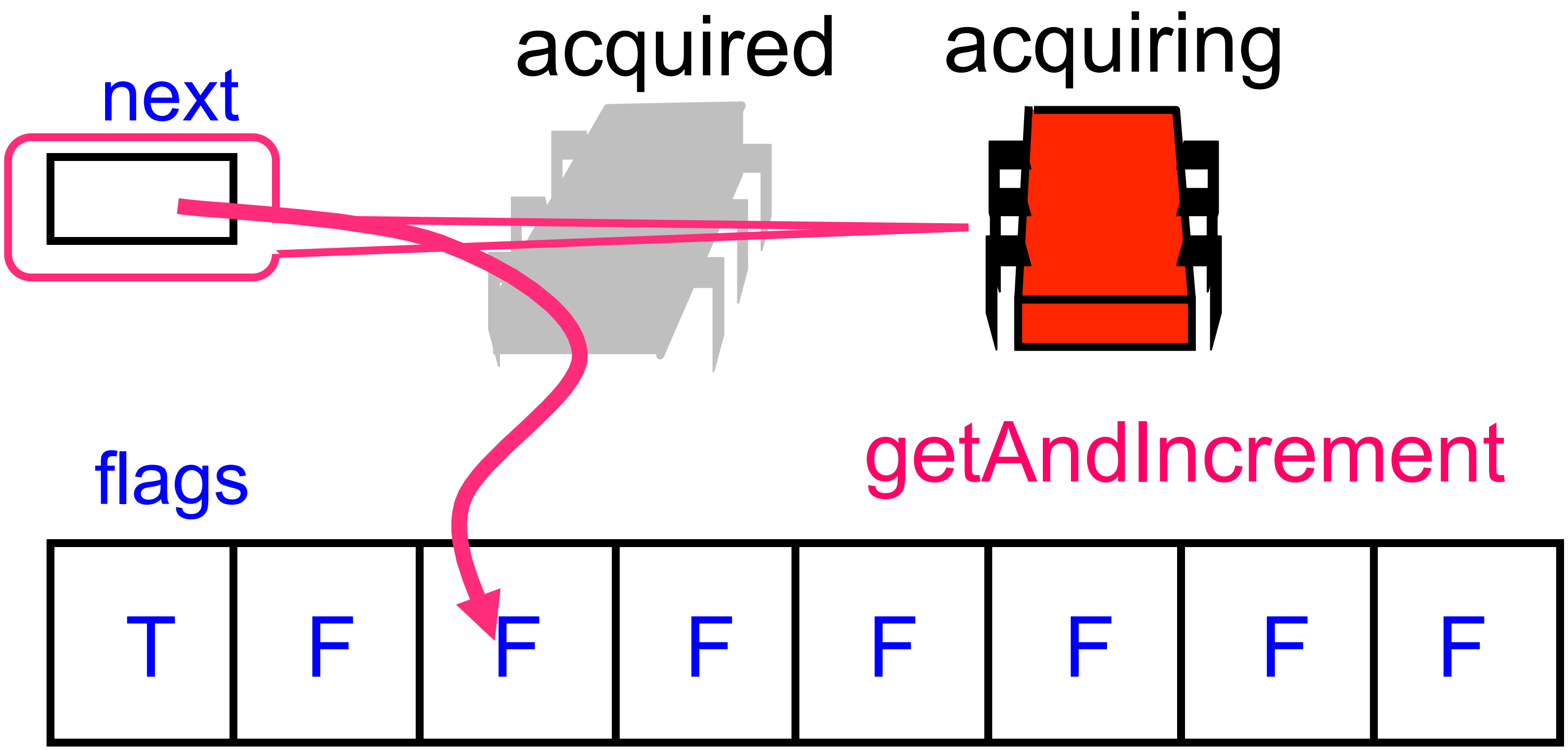
Anderson Queue Lock



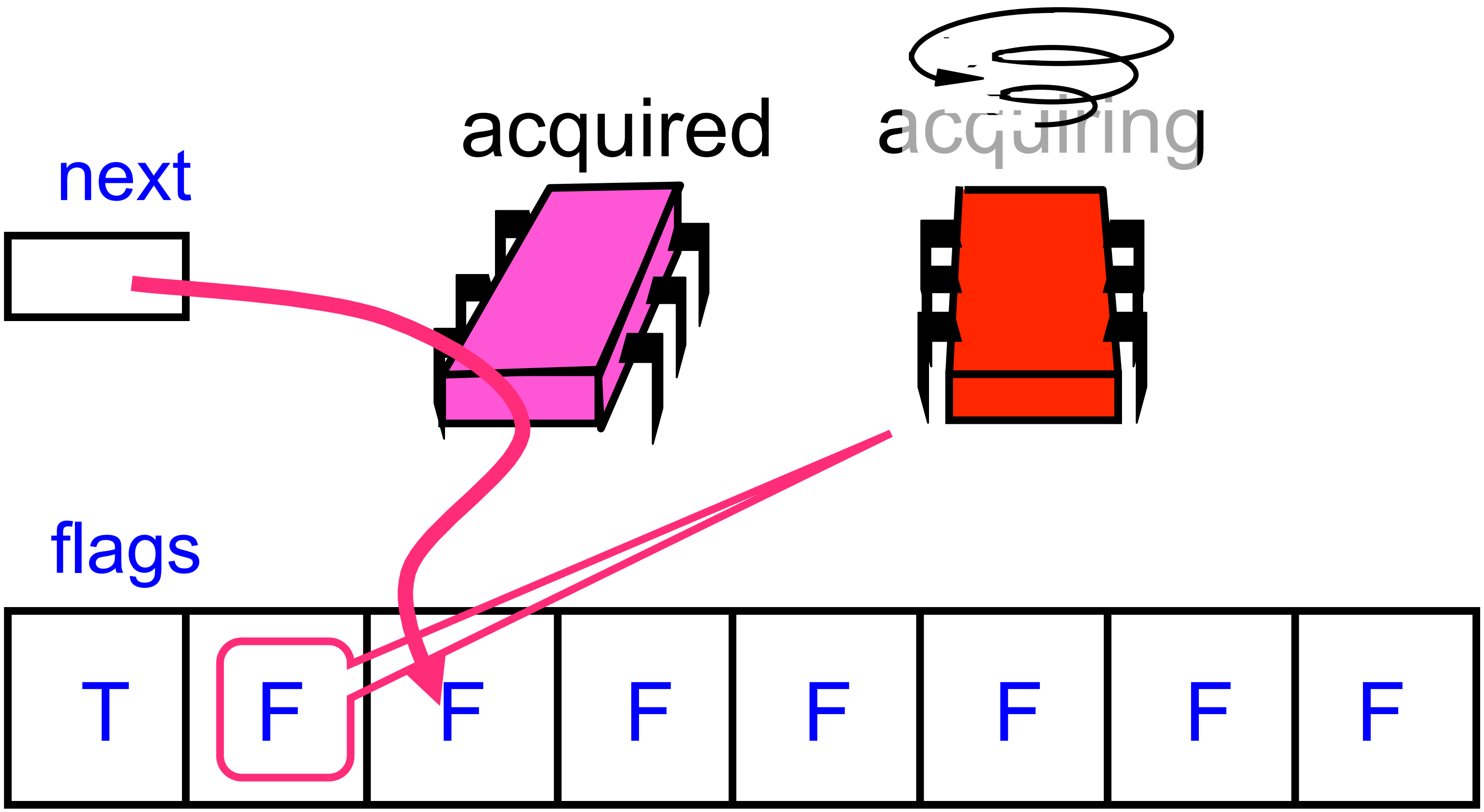
Anderson Queue Lock



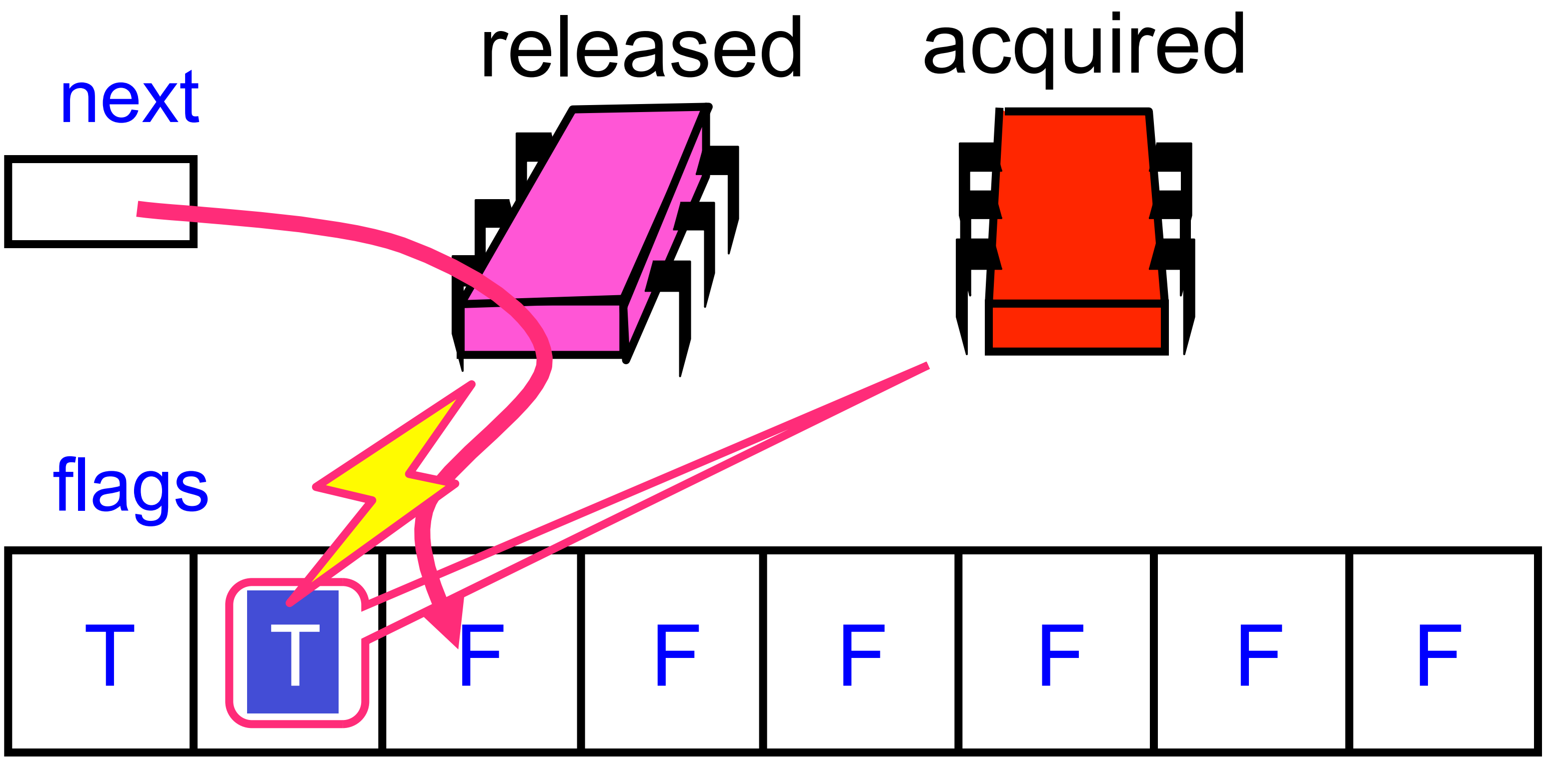
Anderson Queue Lock



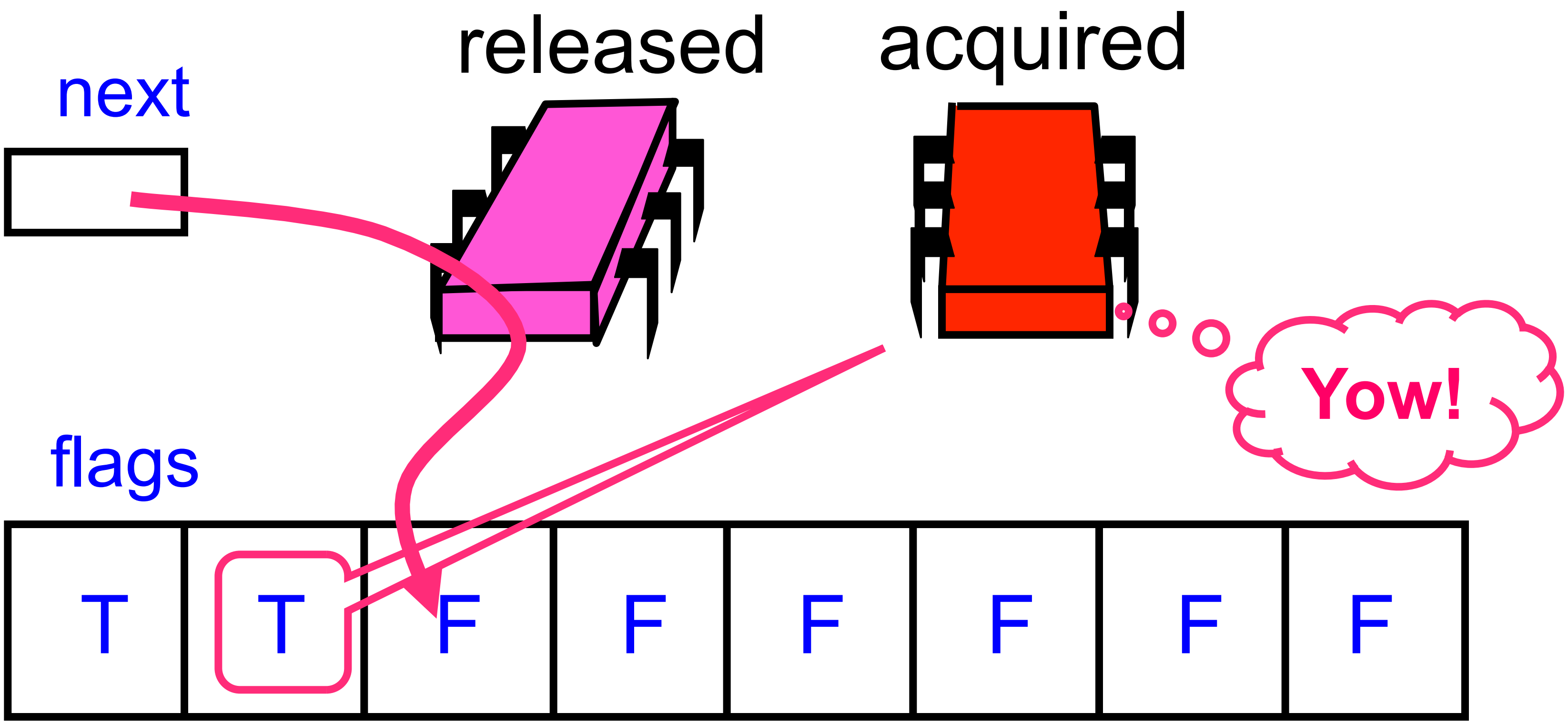
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 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

Tell next thread to go

```
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] = true;
}
```

Compiler can actually optimize this away

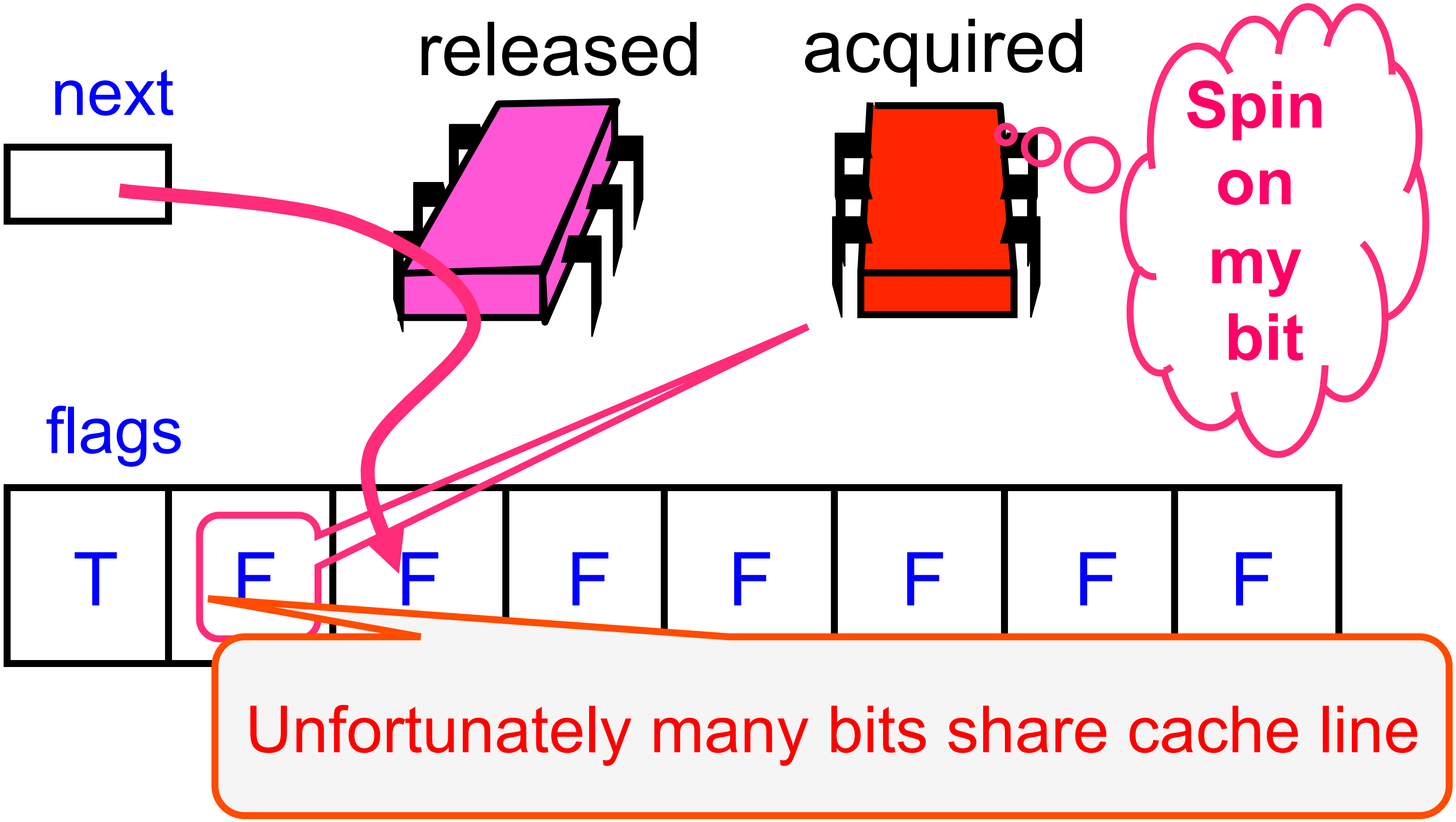
Properly, Anderson Queue Lock

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

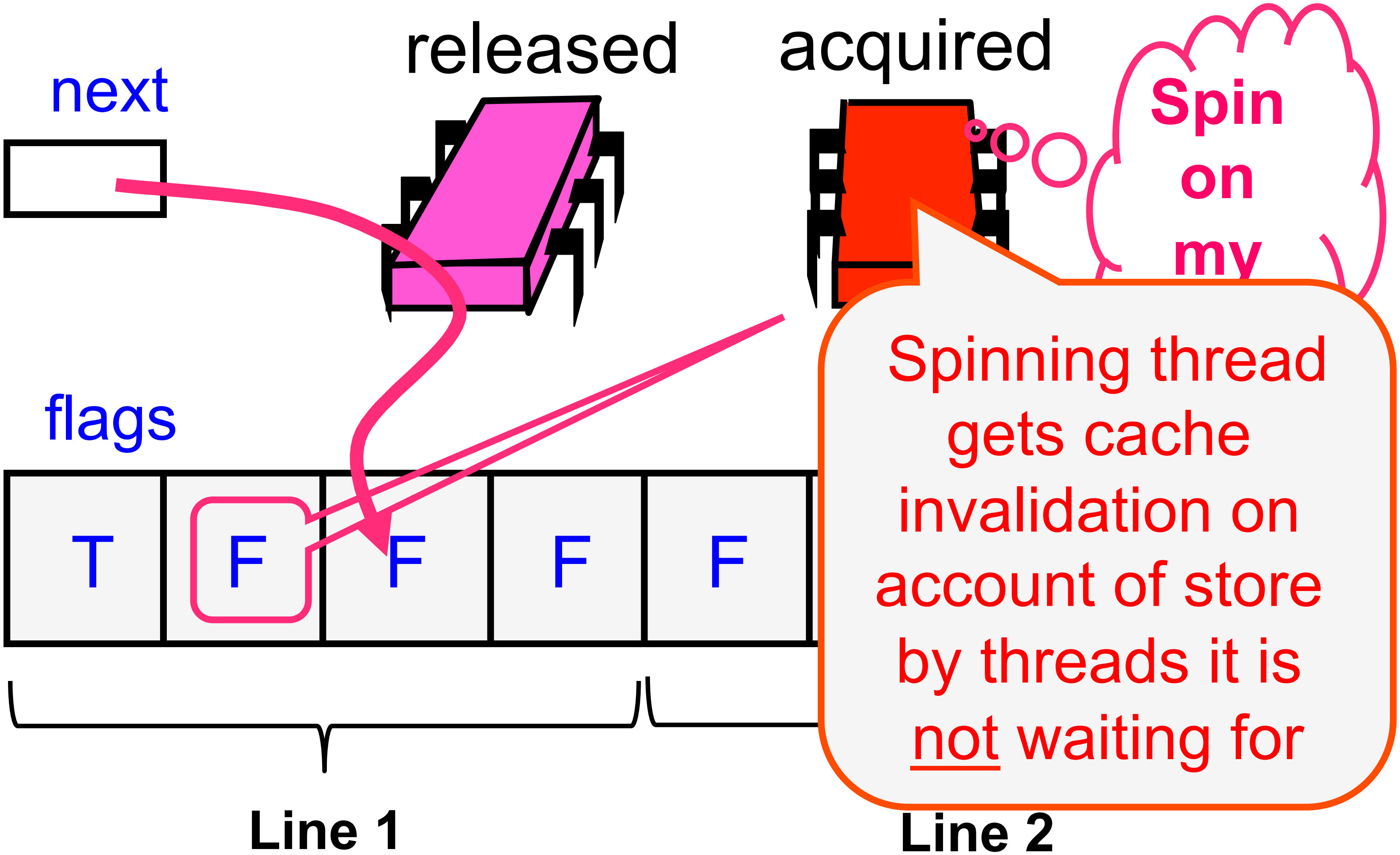
Declare the flags array volatile*, so the loop involves a volatile read.

*This just means that the array itself is volatile, but not each array element.

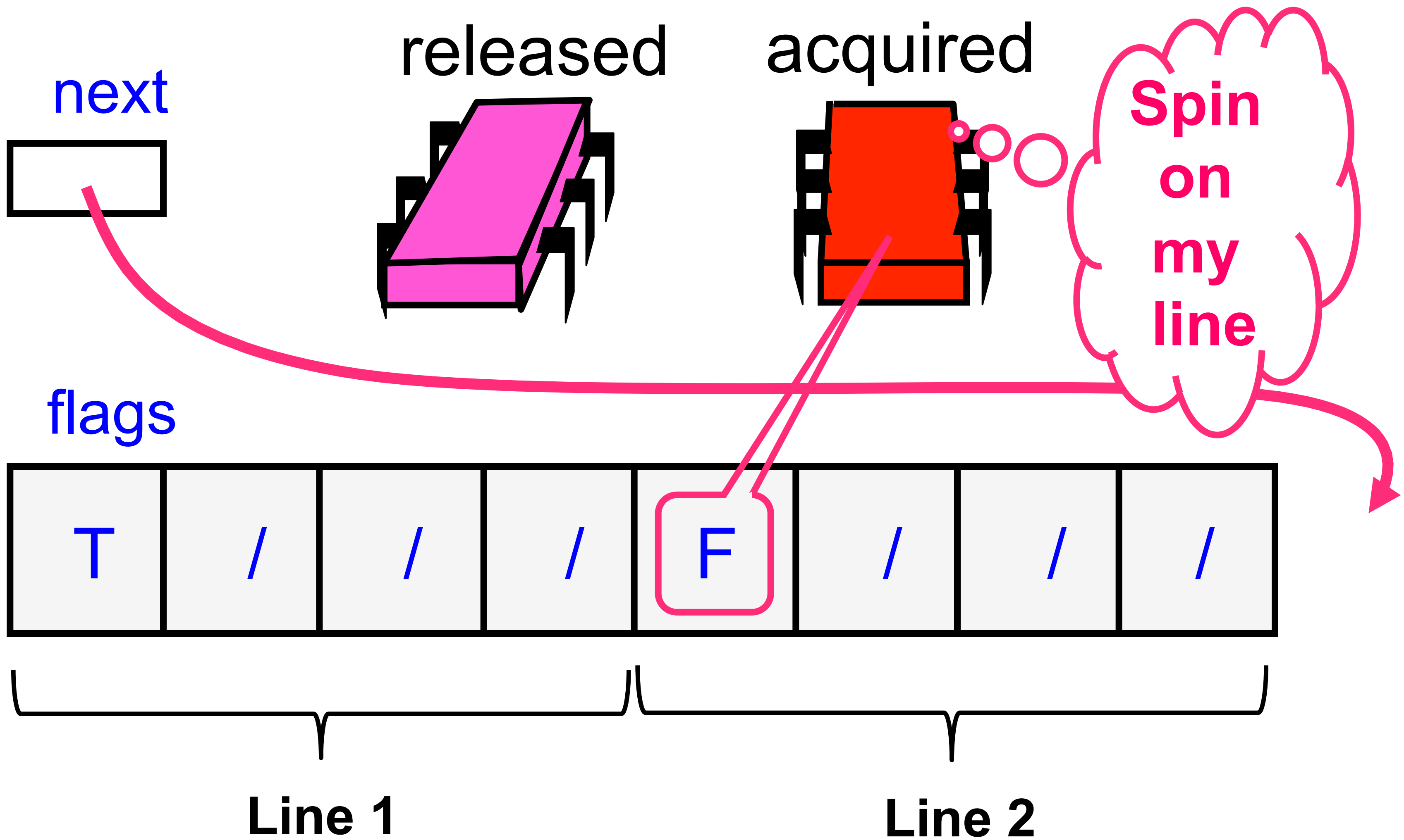
Local Spinning



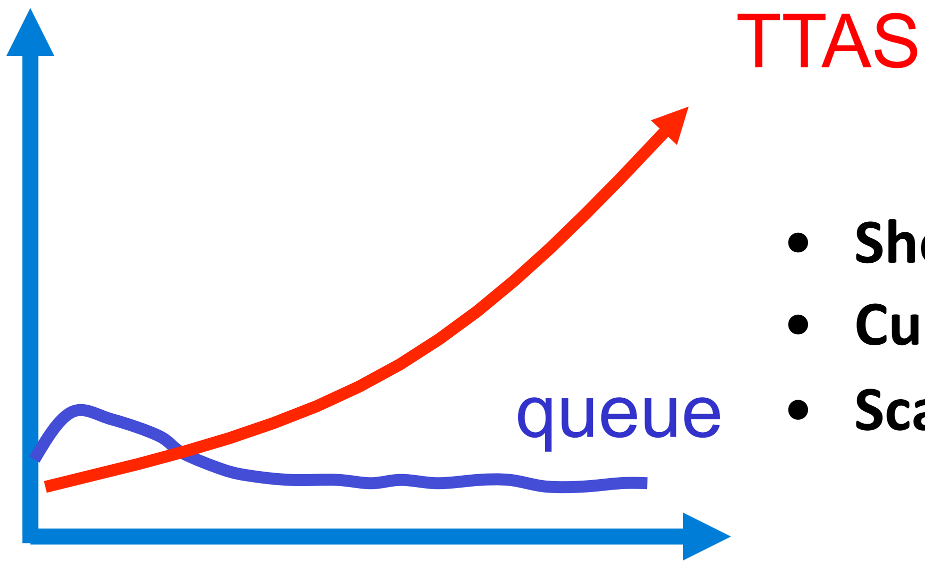
False Sharing



The Solution: Padding



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?

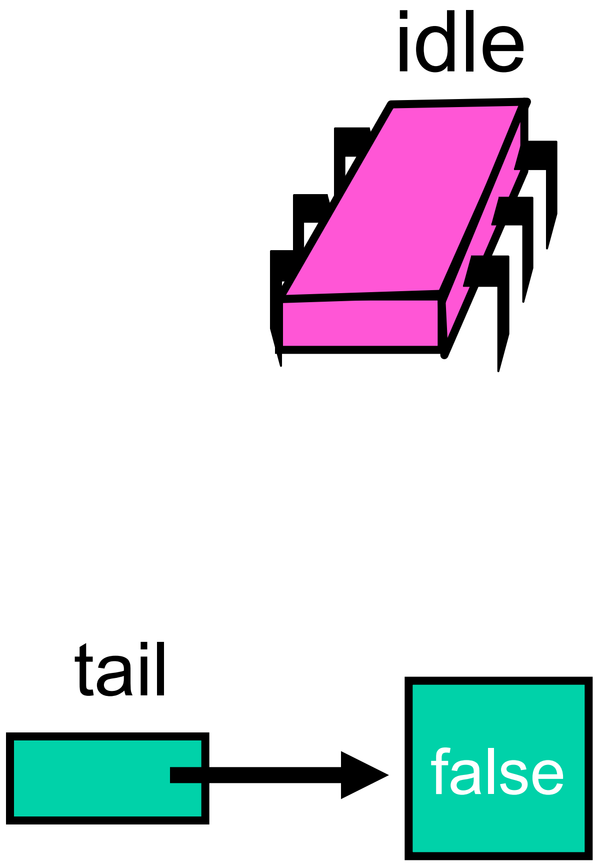
CLH Lock

(by Travis Craig, Erik Hagersten, and Anders Landin)

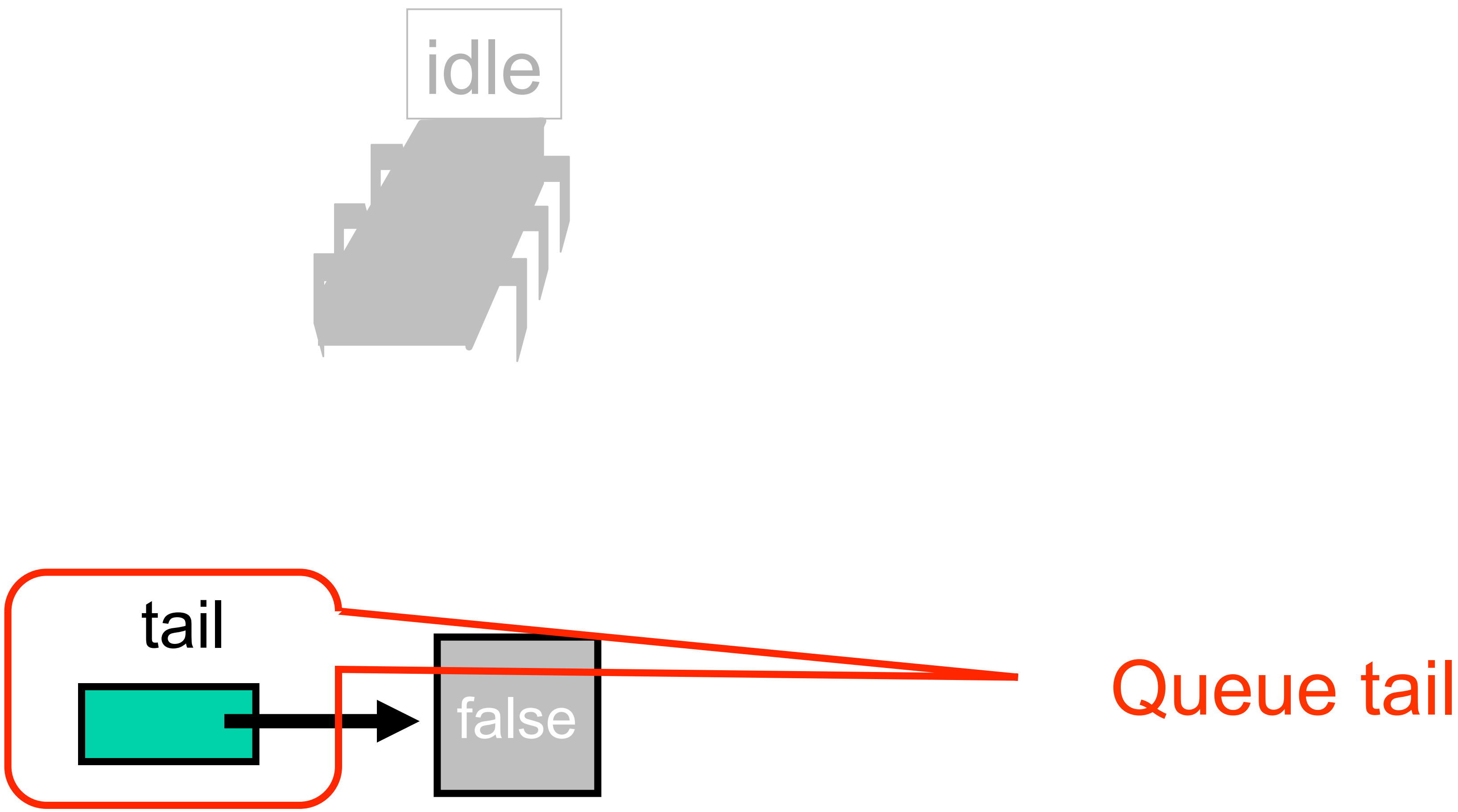
CLH Lock

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

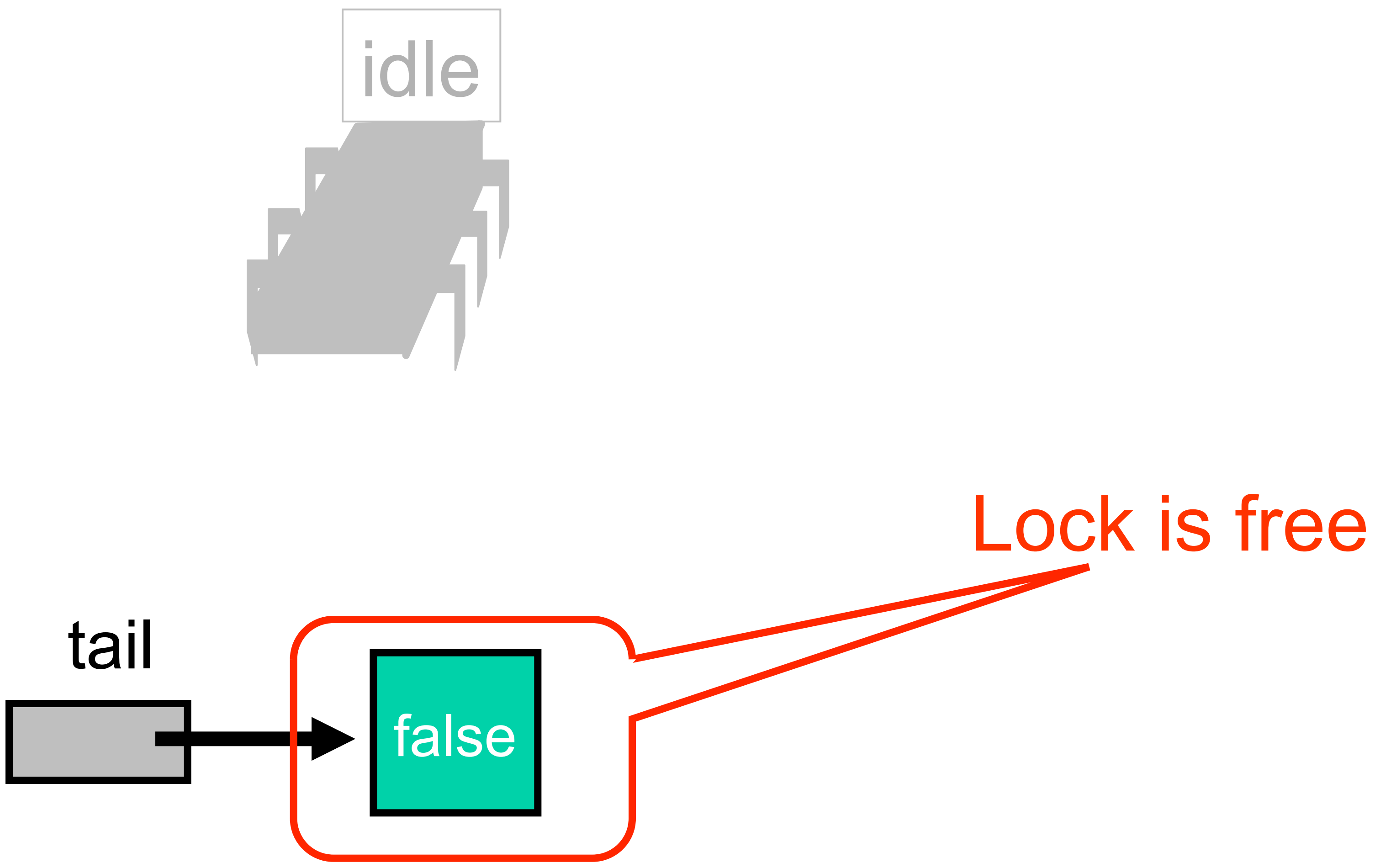
Initially



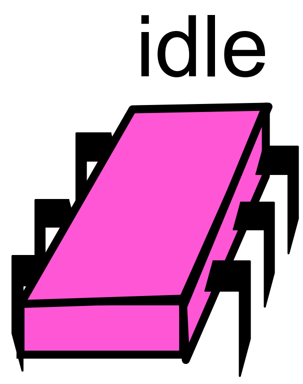
Initially

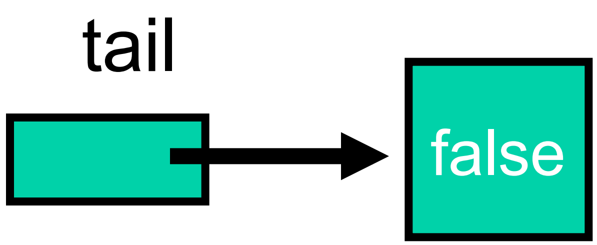


Initially

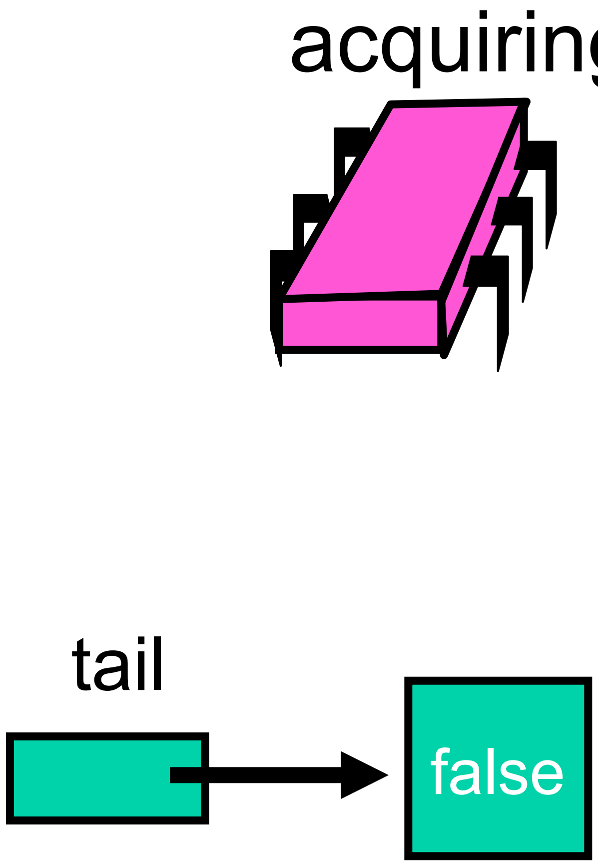


Initially

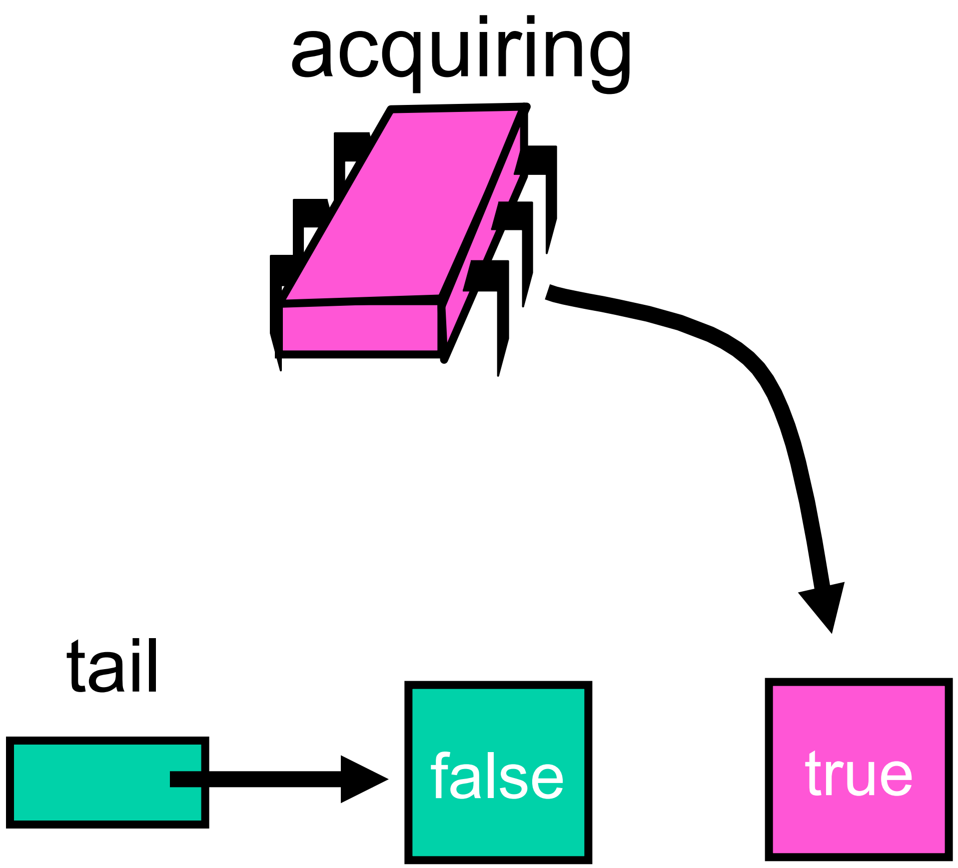




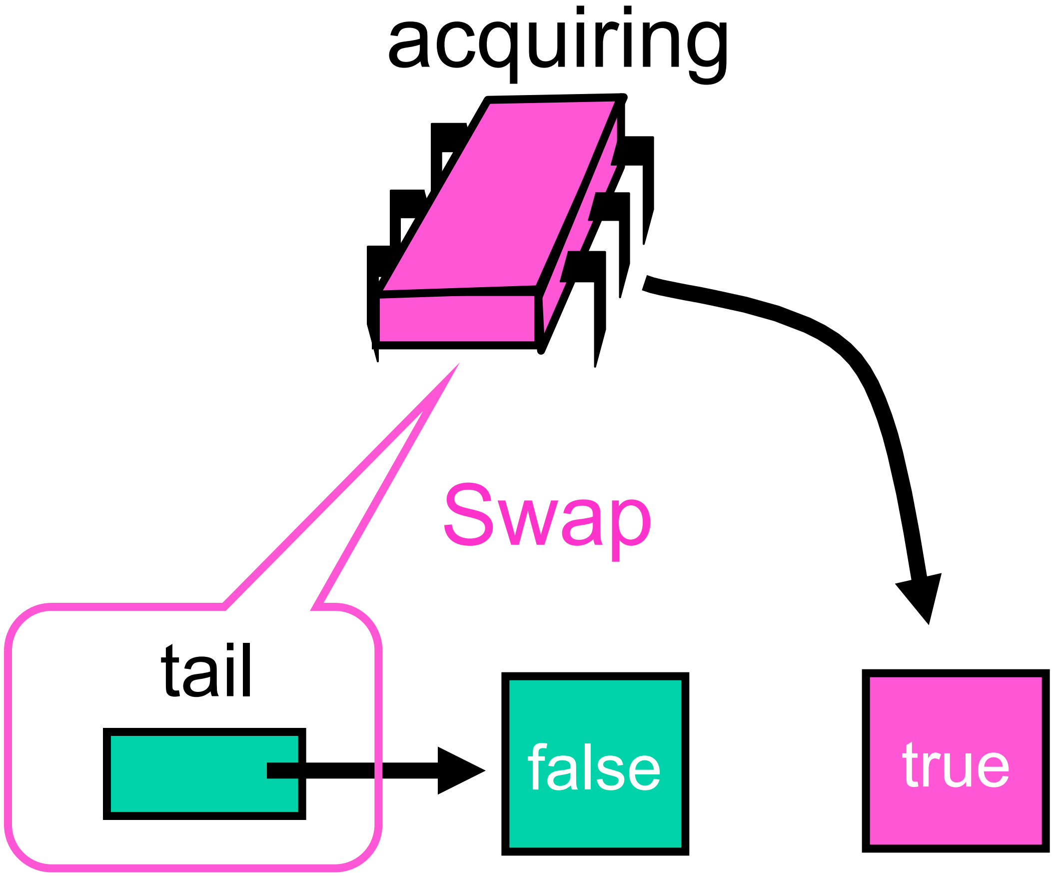
Purple Wants the Lock



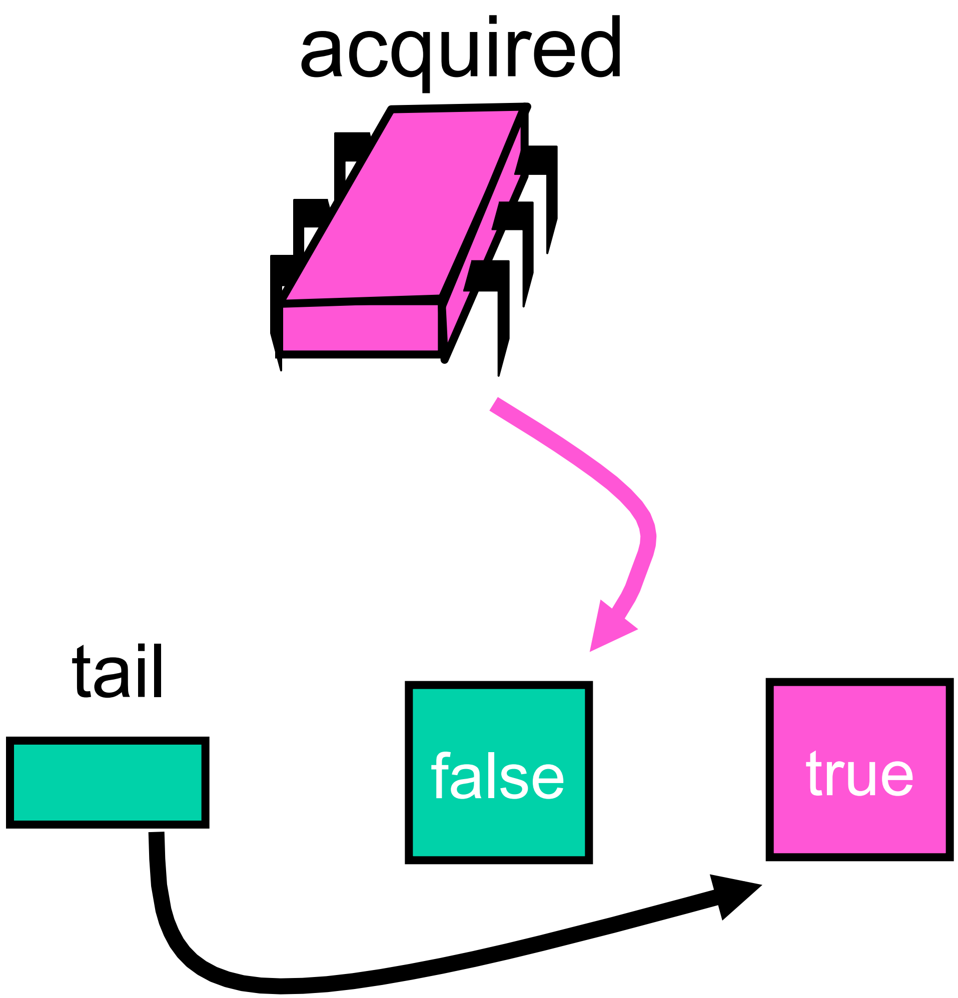
Purple Wants the Lock



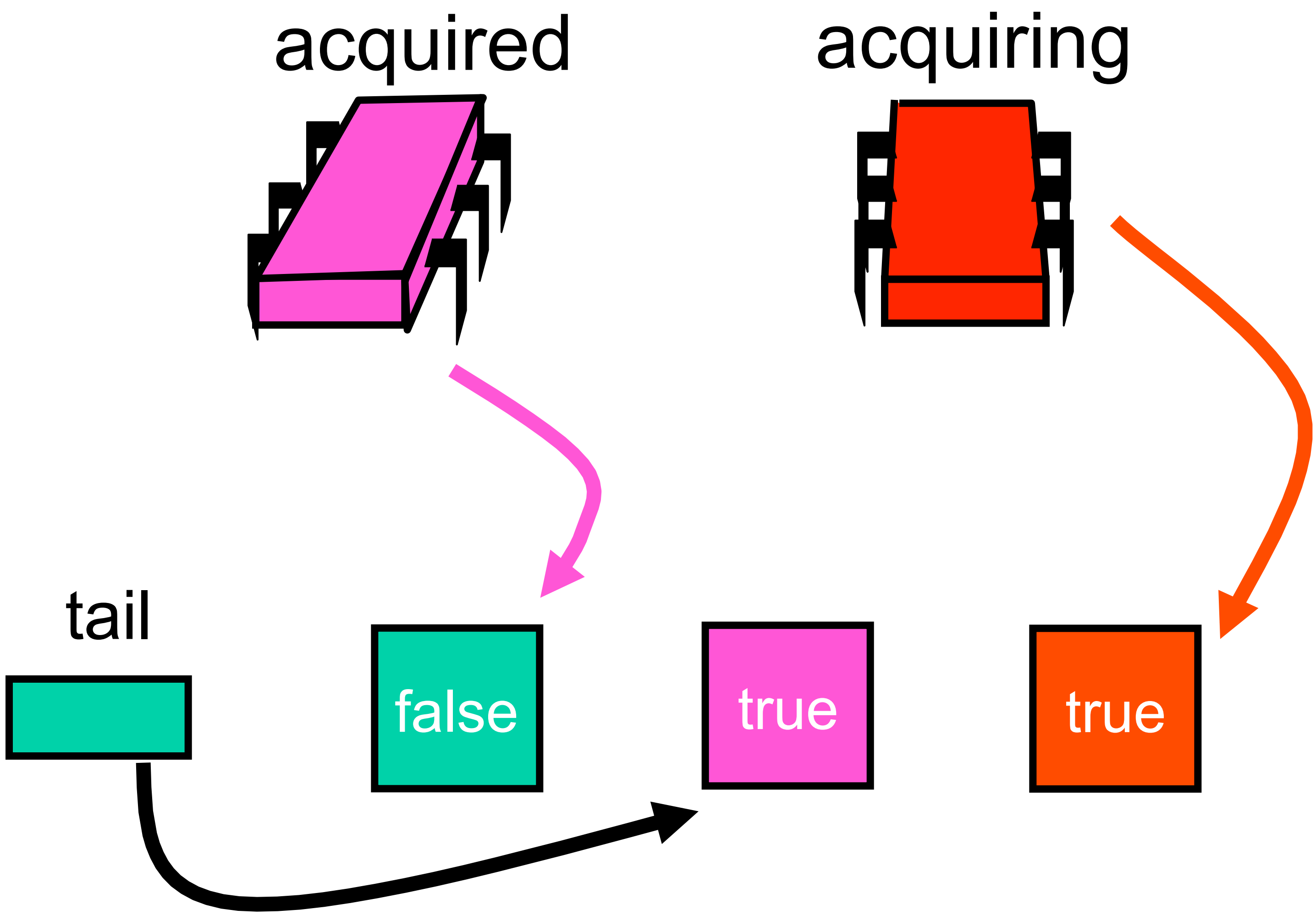
Purple Wants the Lock



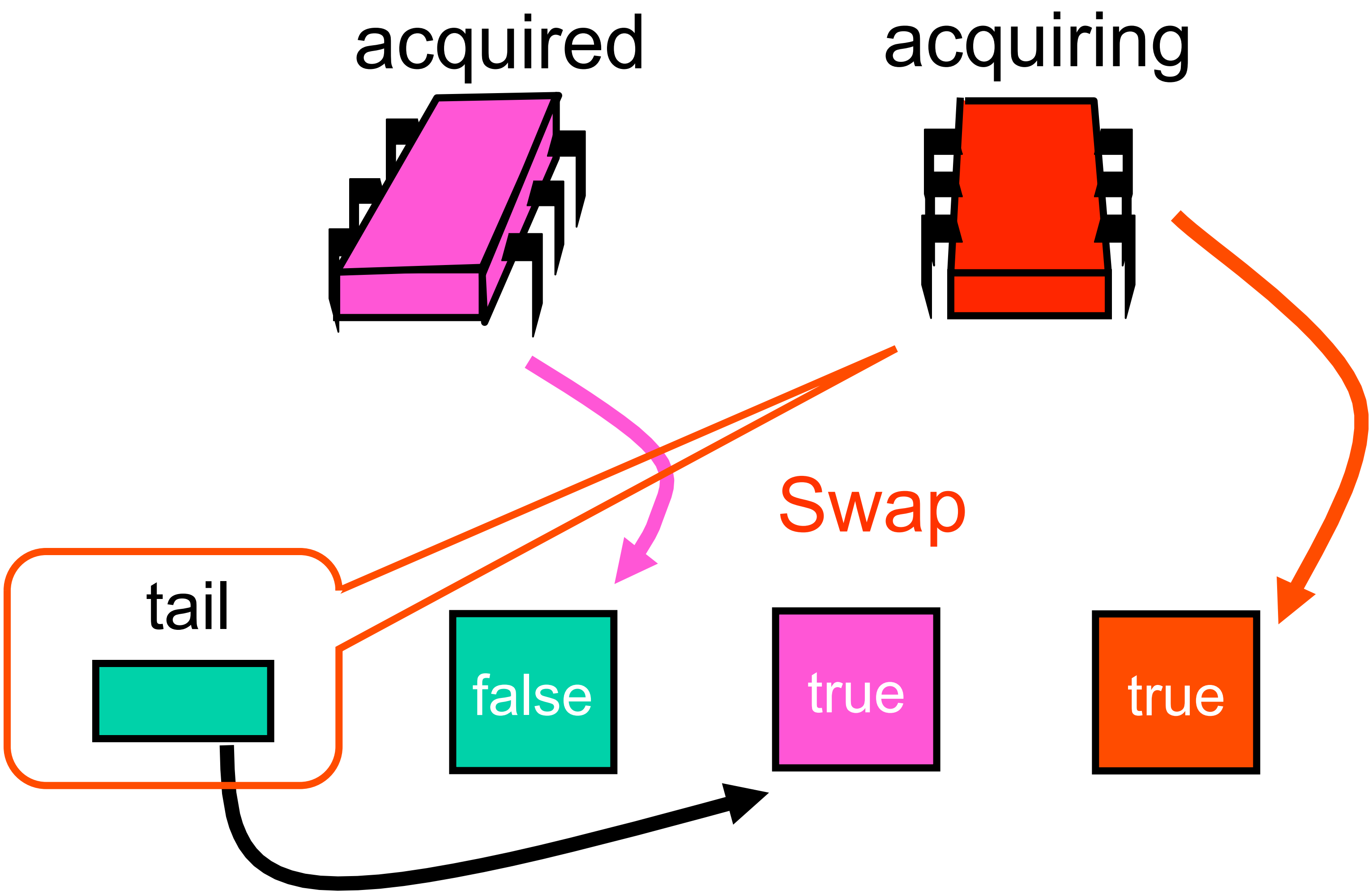
Purple Has the Lock



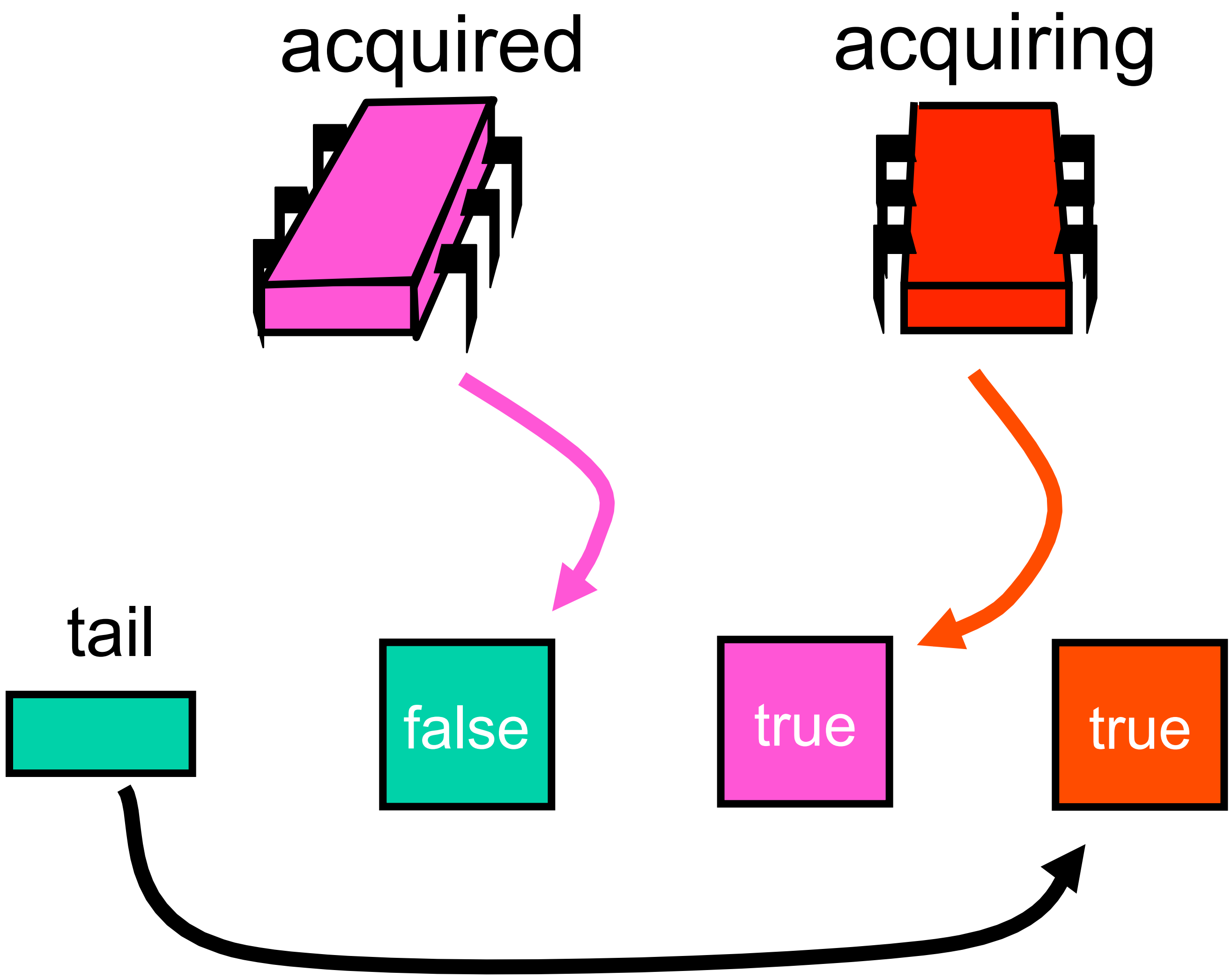
Red Wants the Lock



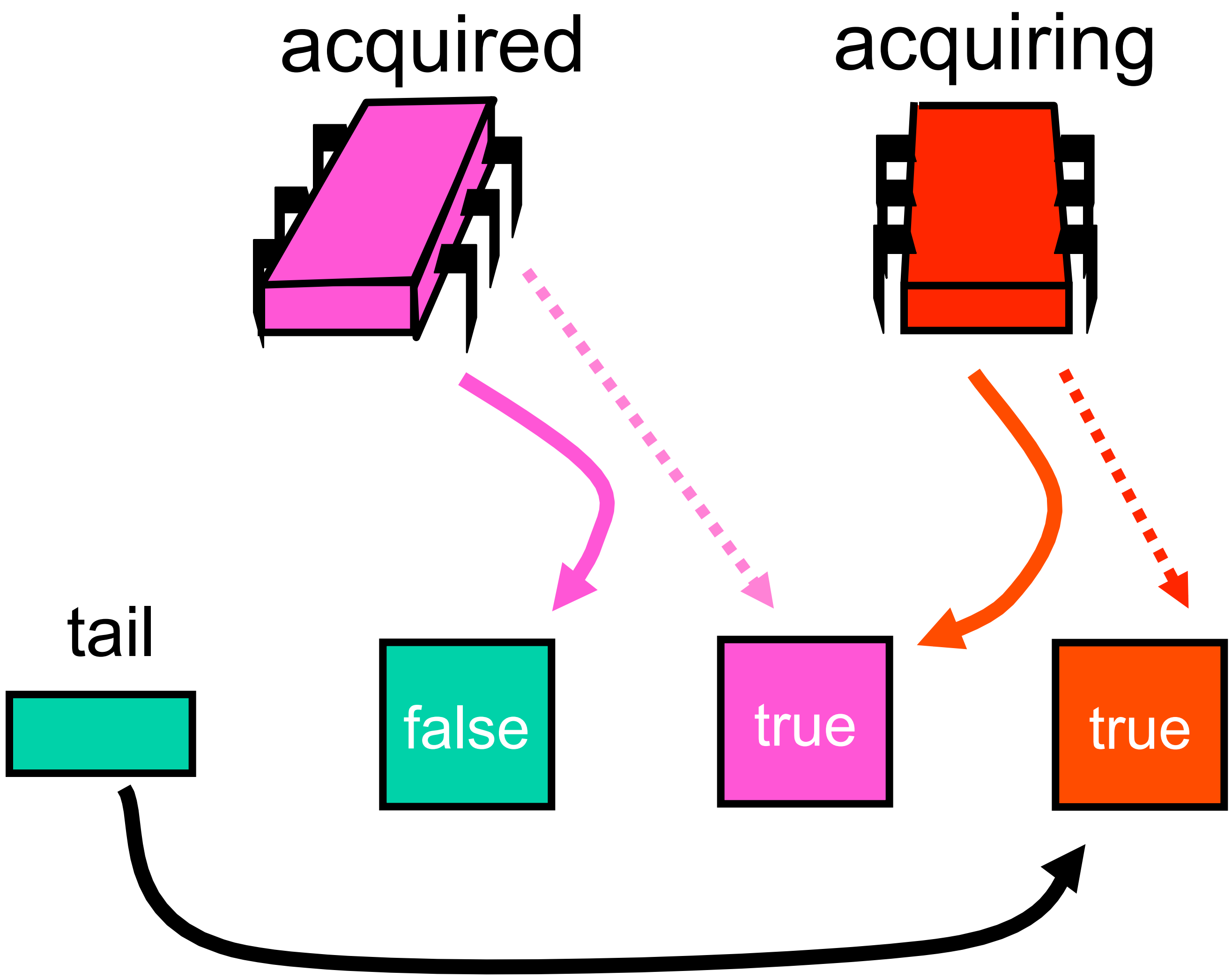
Red Wants the Lock



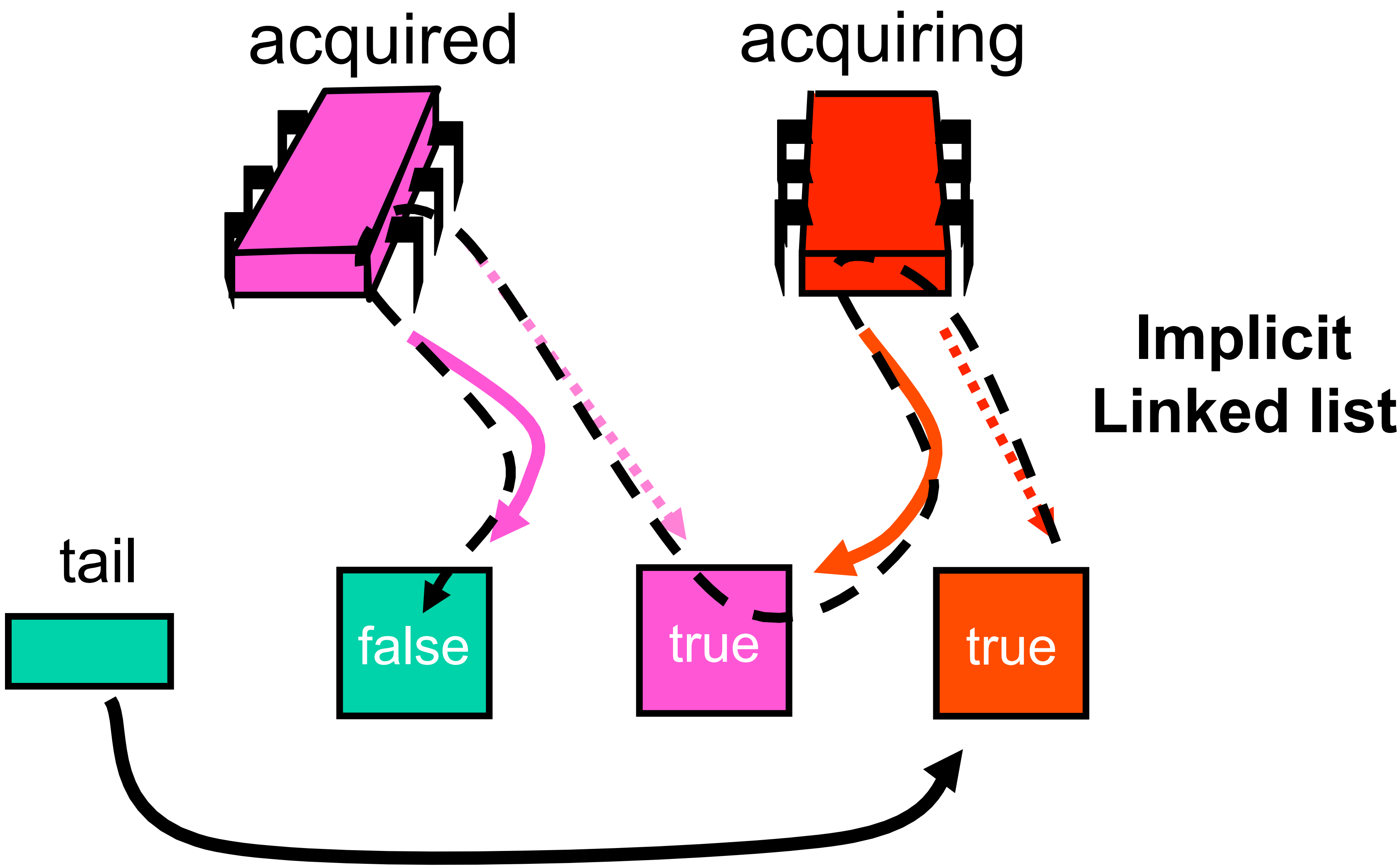
Red Wants the Lock



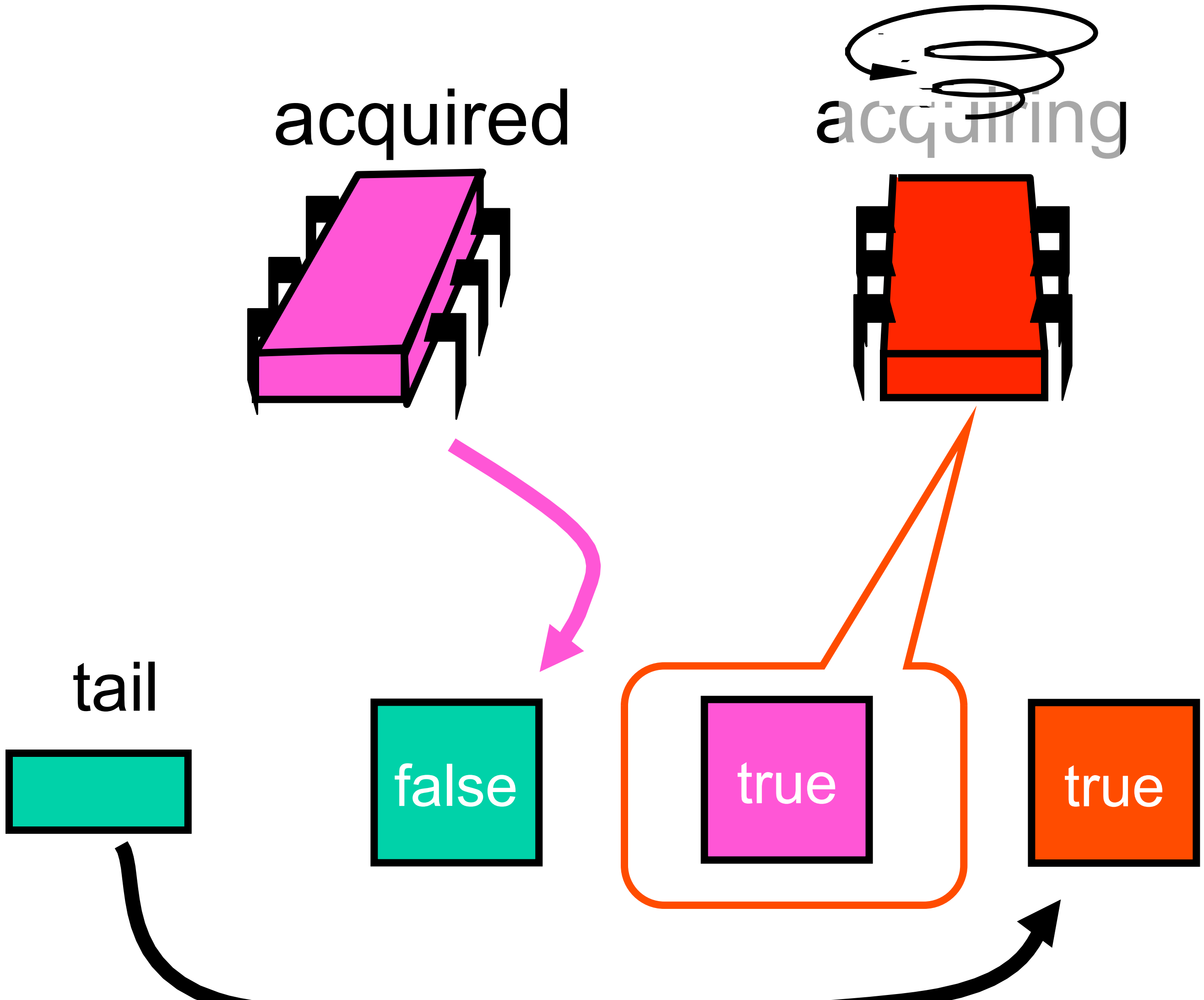
Red Wants the Lock



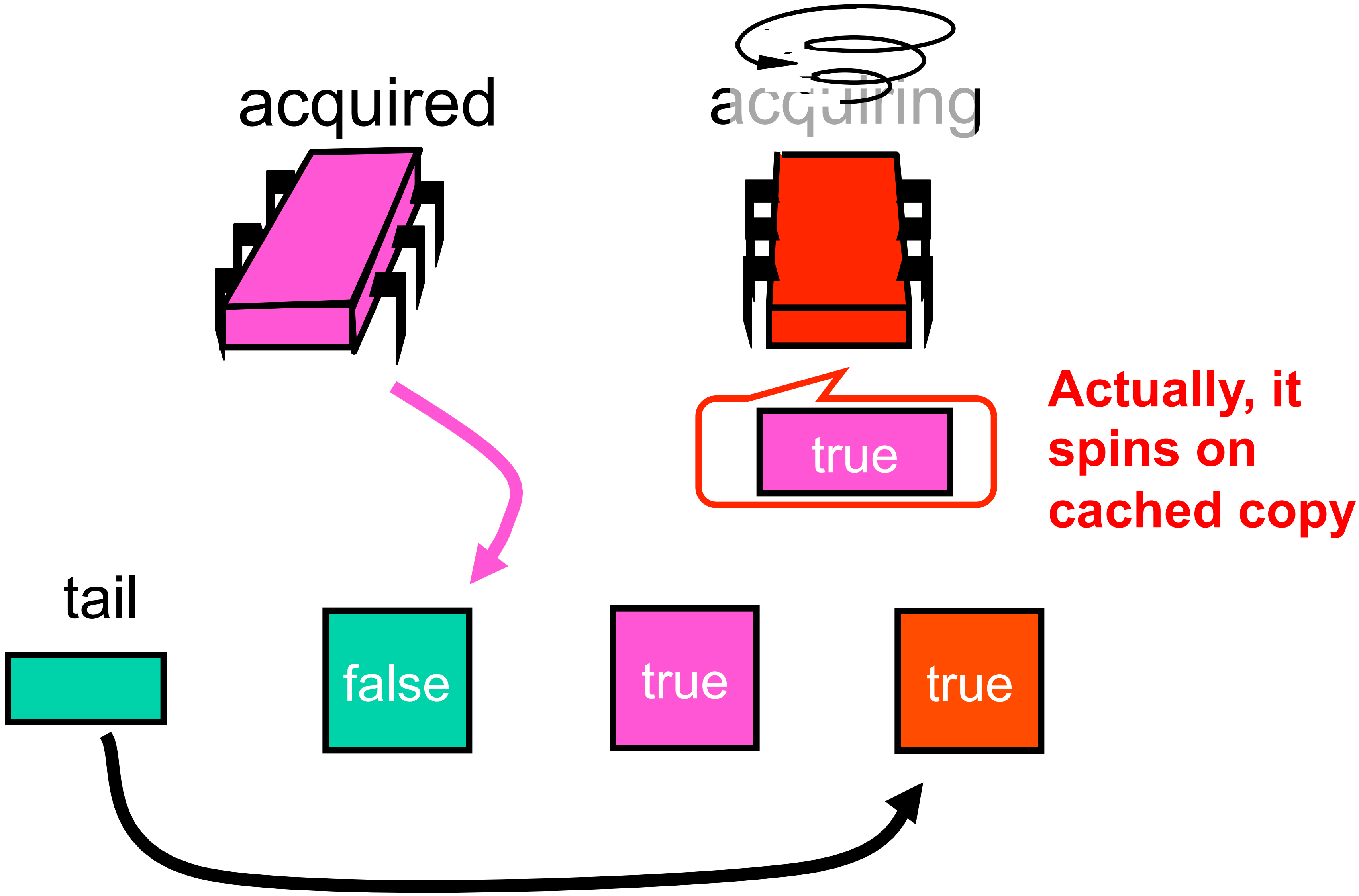
Red Wants the Lock



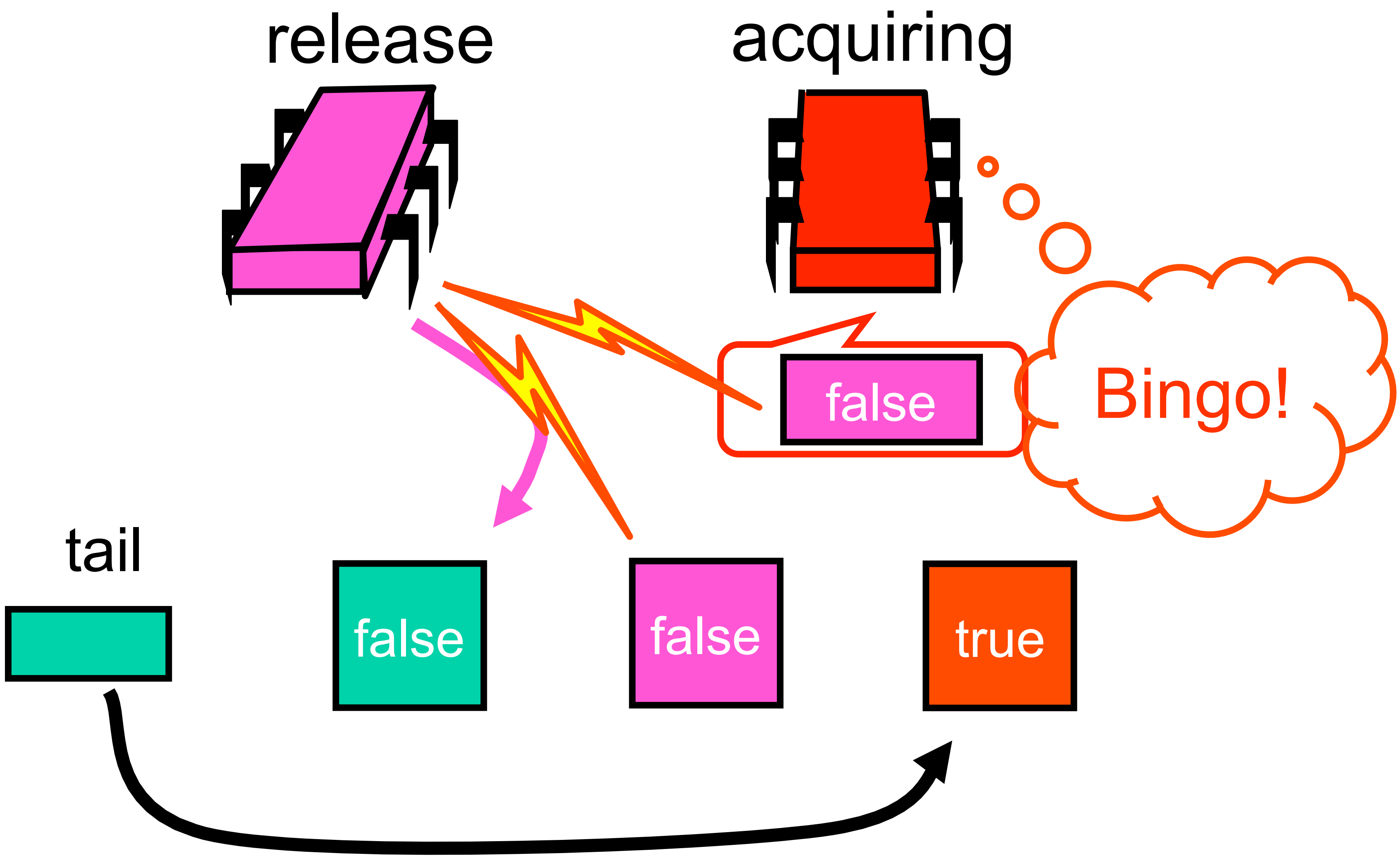
Red Wants the Lock



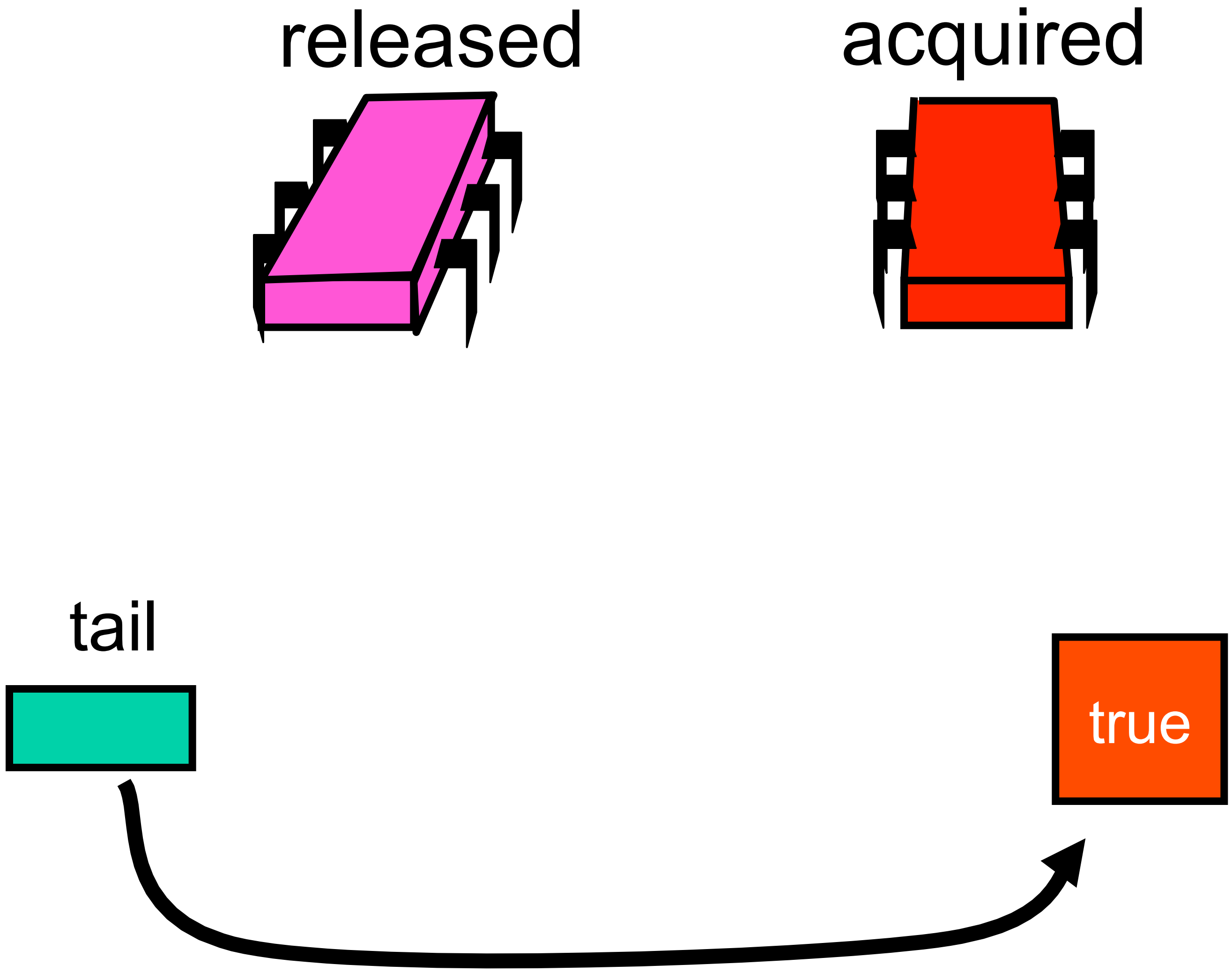
Red Wants the Lock



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();
 ThreadLocal<Qnode> myPred = null;
 public void lock() {
  Qnode pred = tail.getAndSet(myNode);
  myPred.set(pred);
  while (pred.locked) {}
 }}
```

CLH Queue Lock

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

Queue tail

CLH Queue Lock

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

Thread-local Qnode

CLH Queue Lock

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

Swap in my node

CLH Queue Lock

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

Spin until predecessor releases lock

CLH Queue Lock

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

CLH Queue Lock

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

Notify successor

CLH Queue Lock

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

Recycle predecessor’s node

Space Usage

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

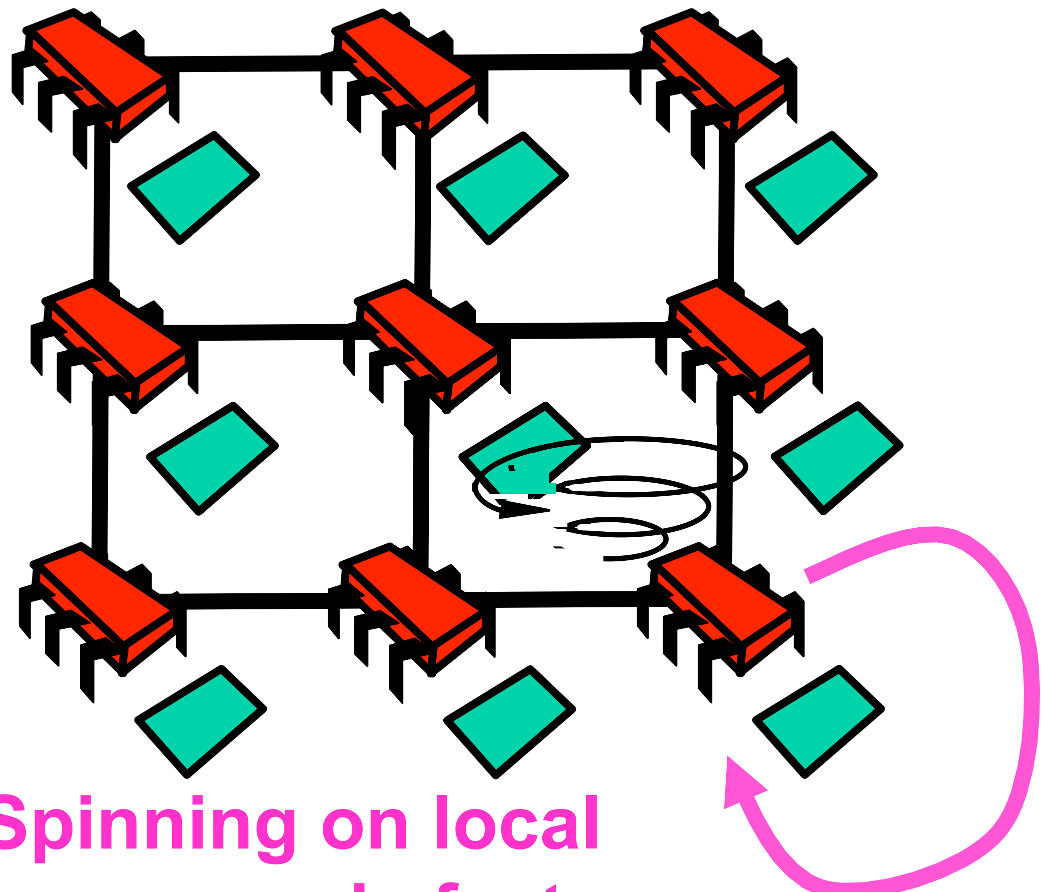
CLH Lock

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

NUMA and cc-NUMA Architectures

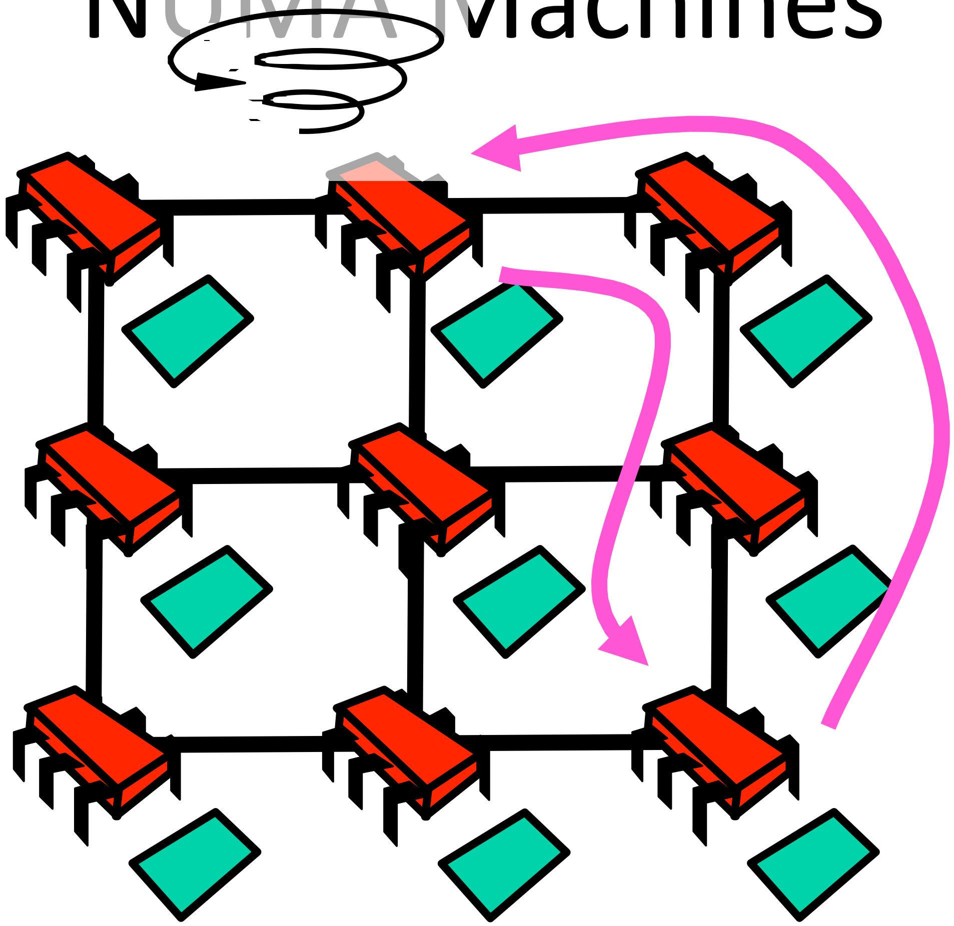
- Acronym:
  - **N**on-**U**niform **M**emory **A**rchitecture
  - ccNUMA = cache coherent NUMA
- Illusion:
  - Flat shared memory
- Truth:
  - No caches (sometimes)
  - Some memory regions faster than others

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 …

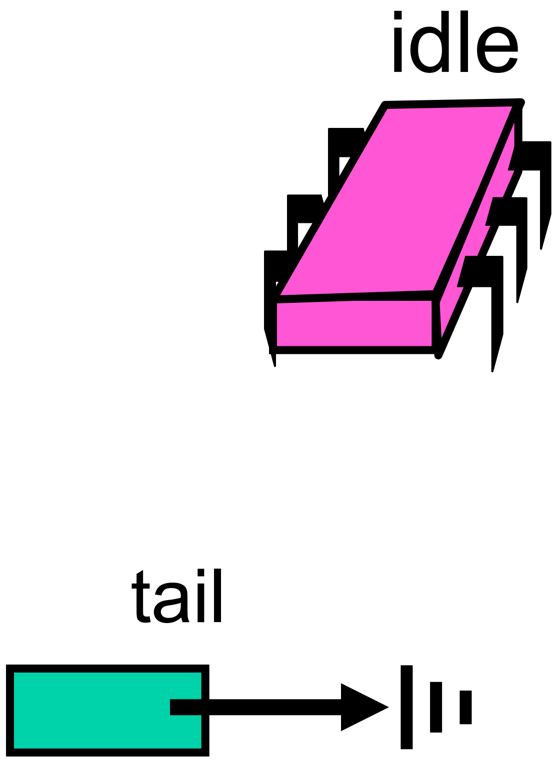
MCS Lock

(by John Mellor-Crummey and Michael Scott)

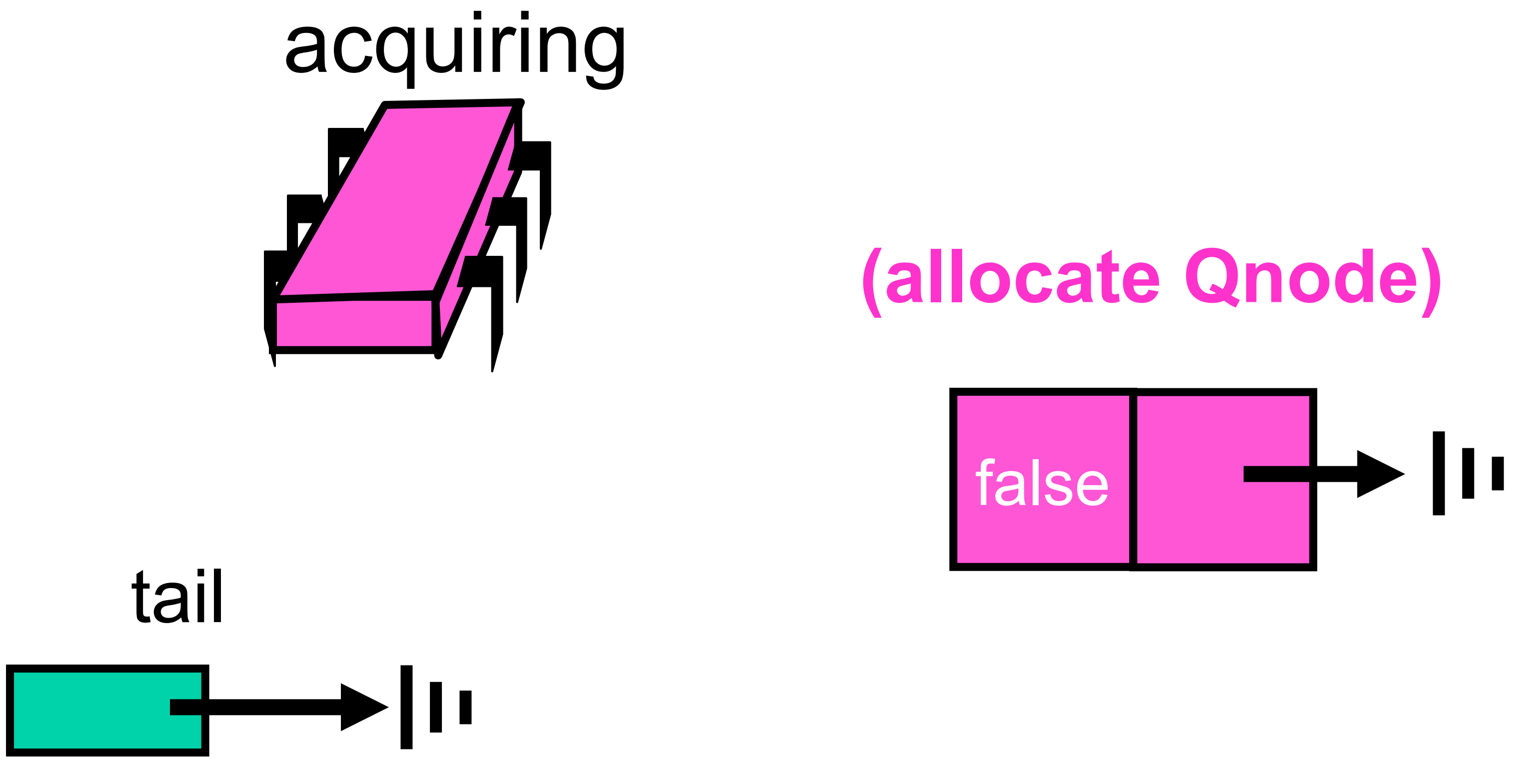
MCS Lock

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

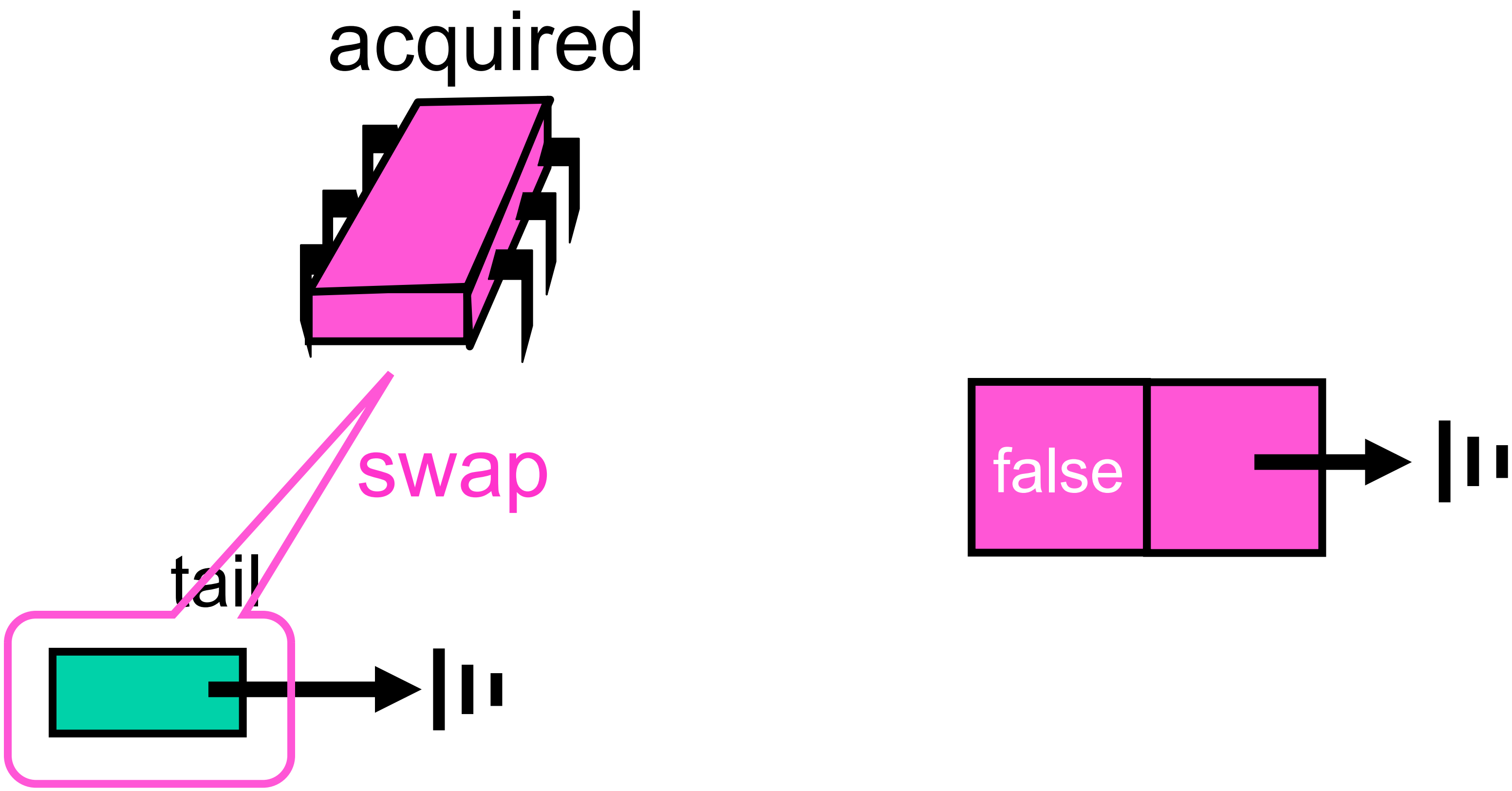
Initially



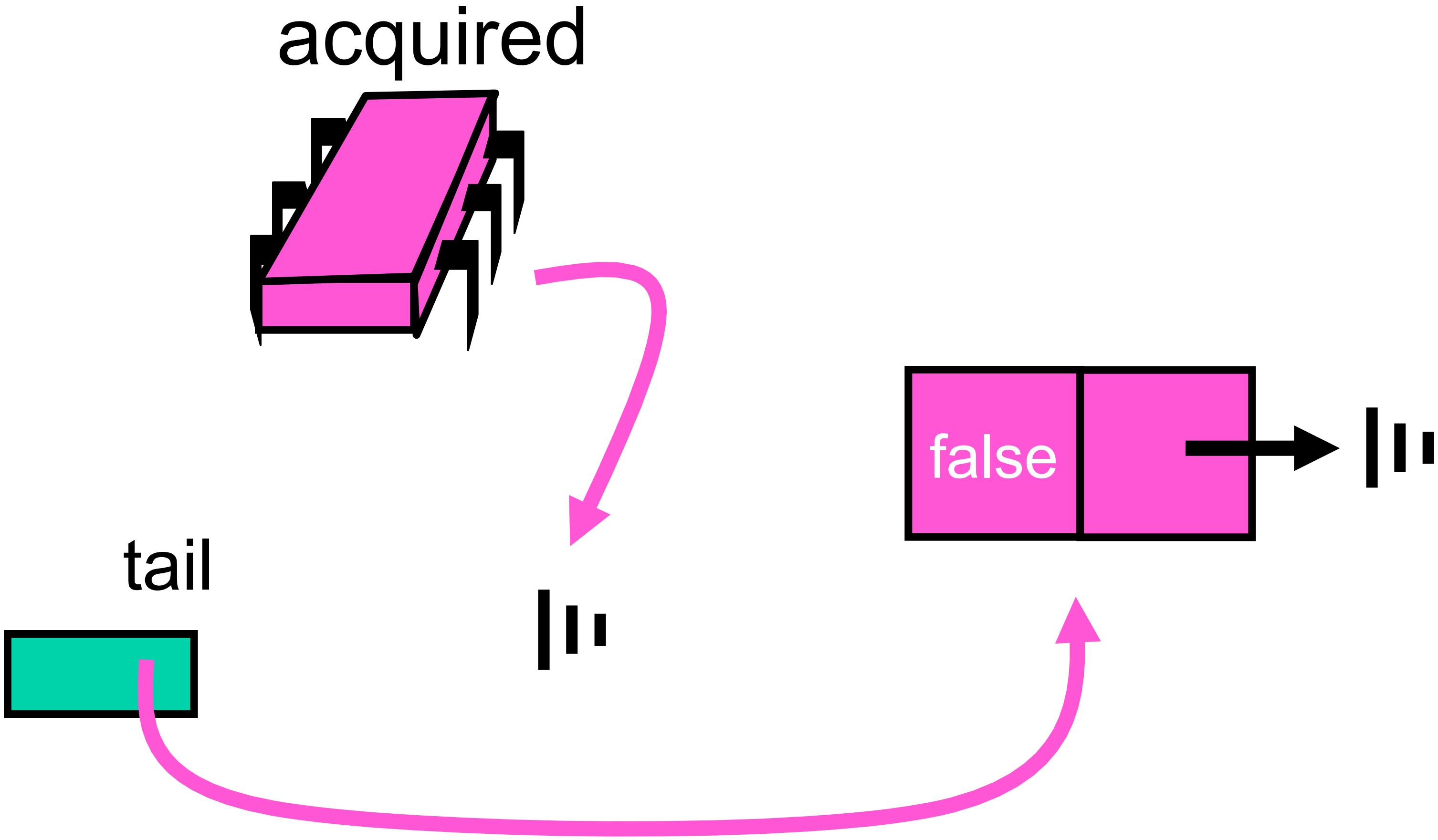
Acquiring



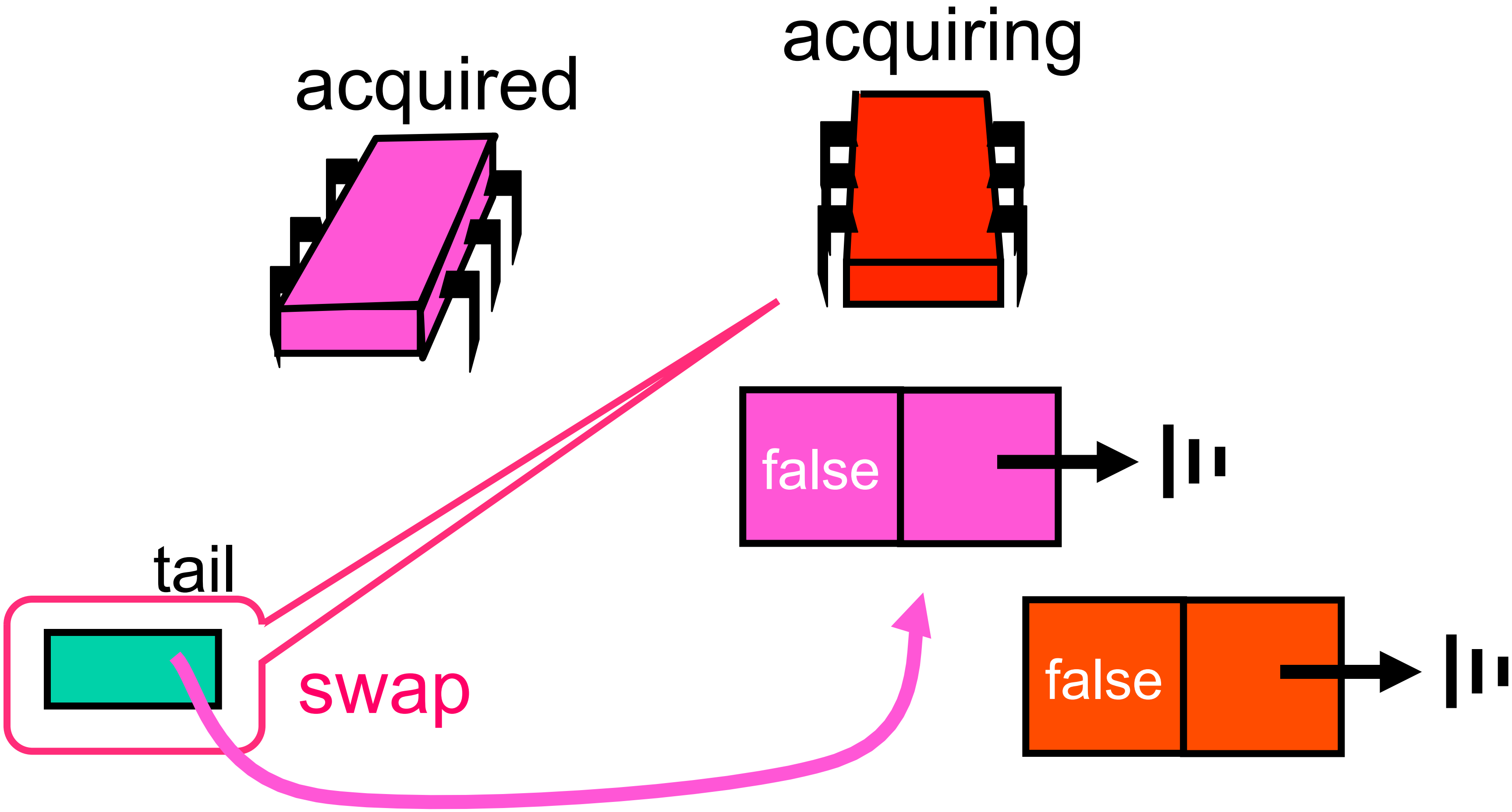
Acquiring



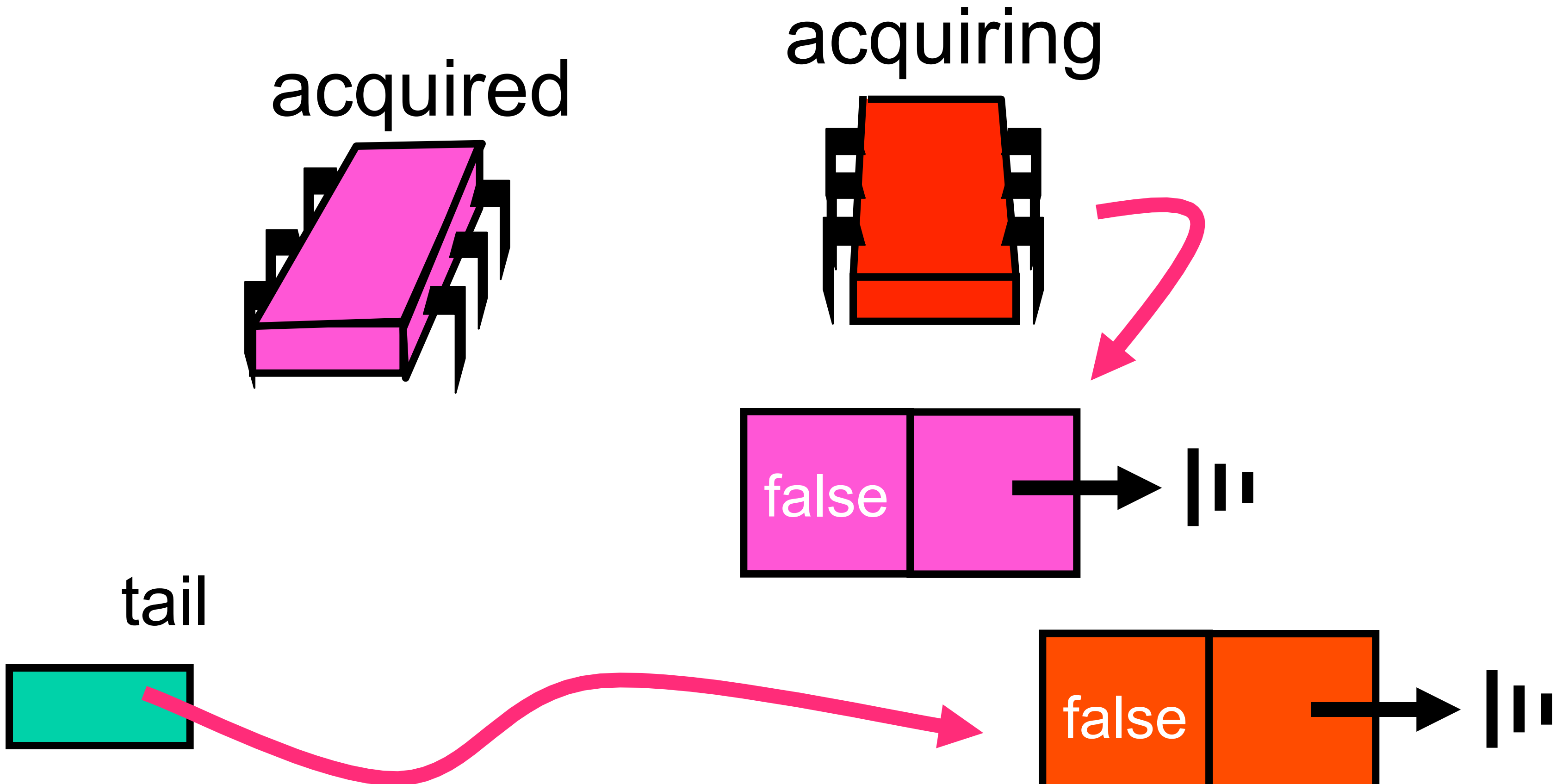
Acquired



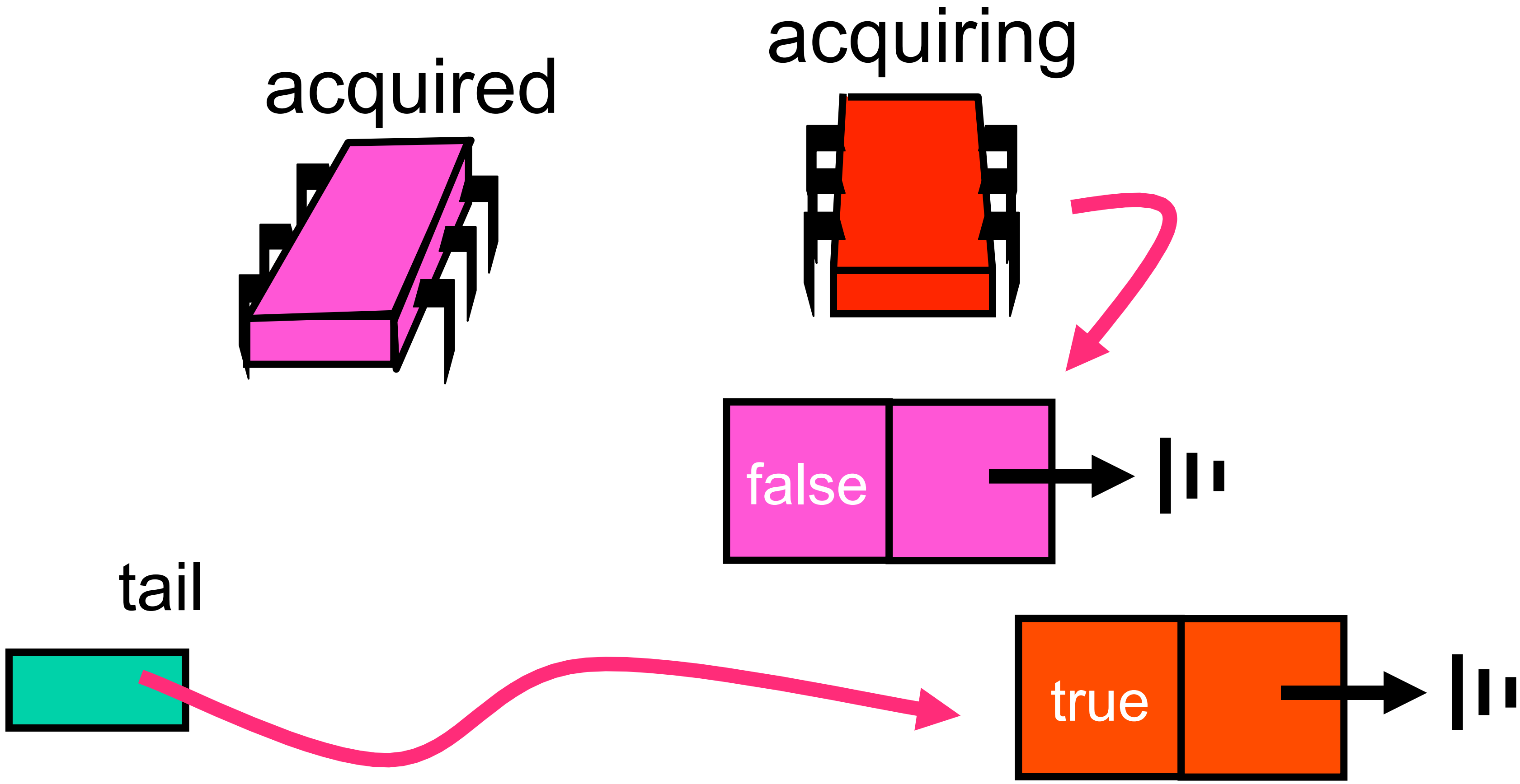
Acquiring



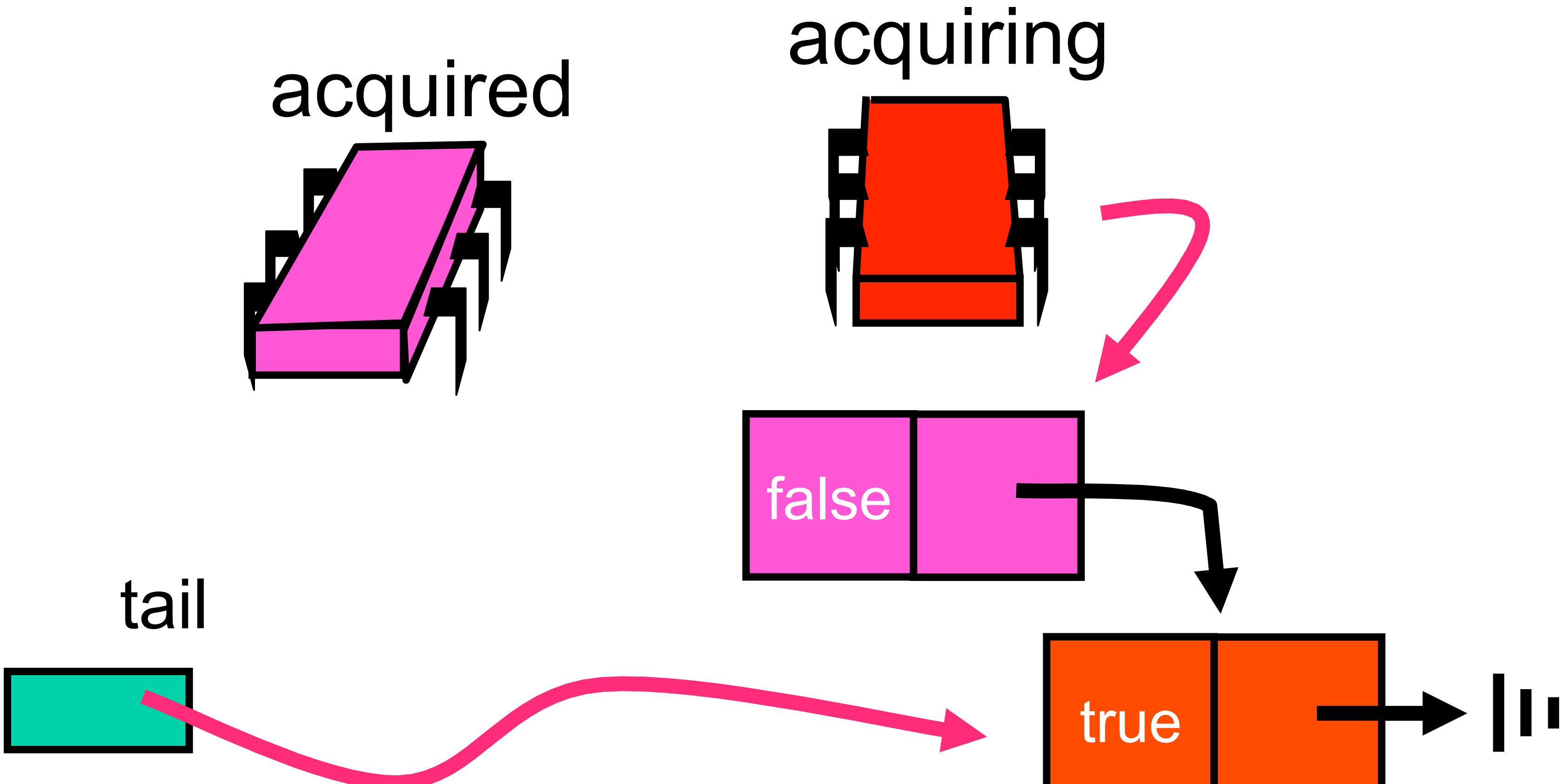
Acquiring



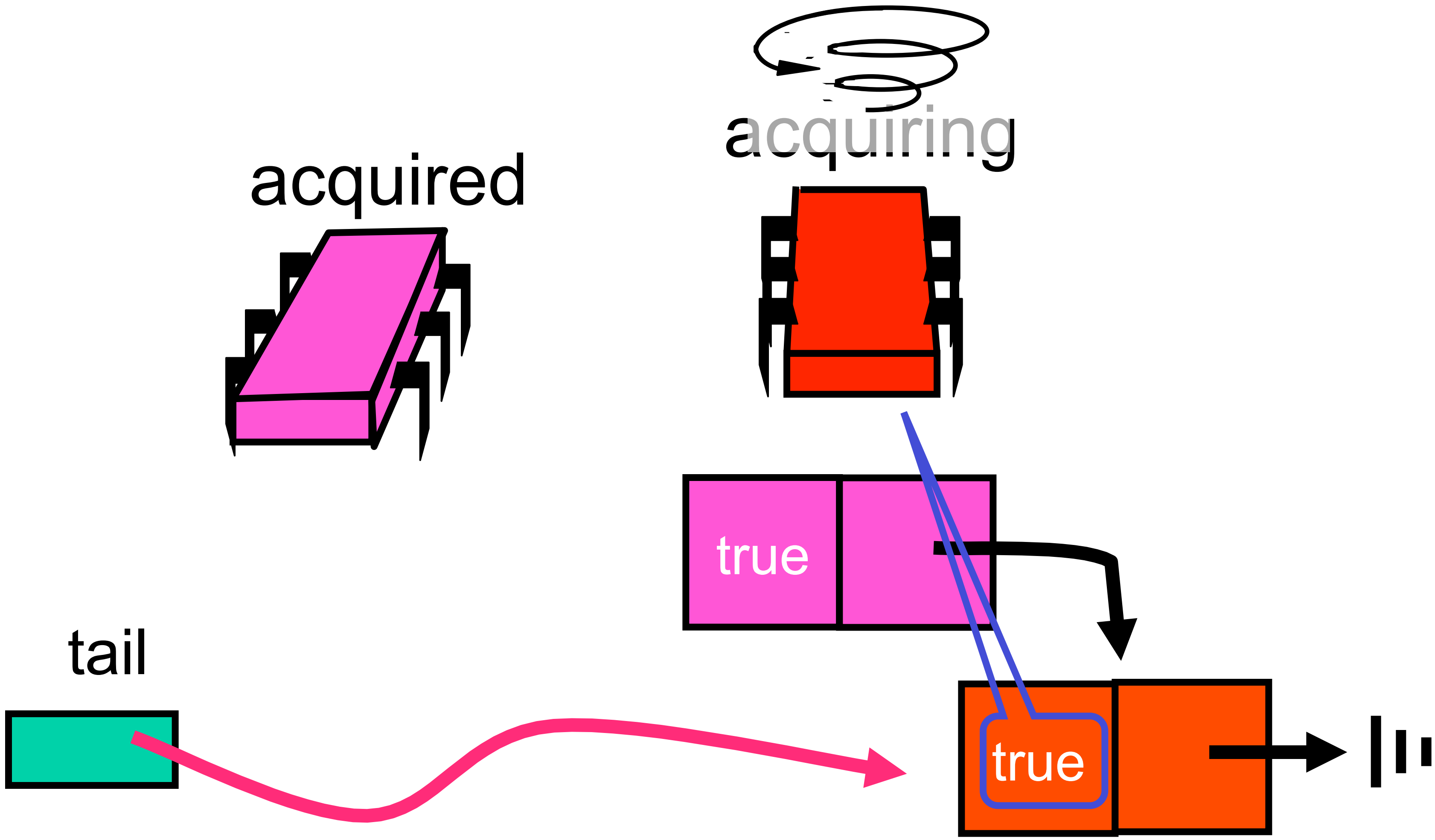
Acquiring



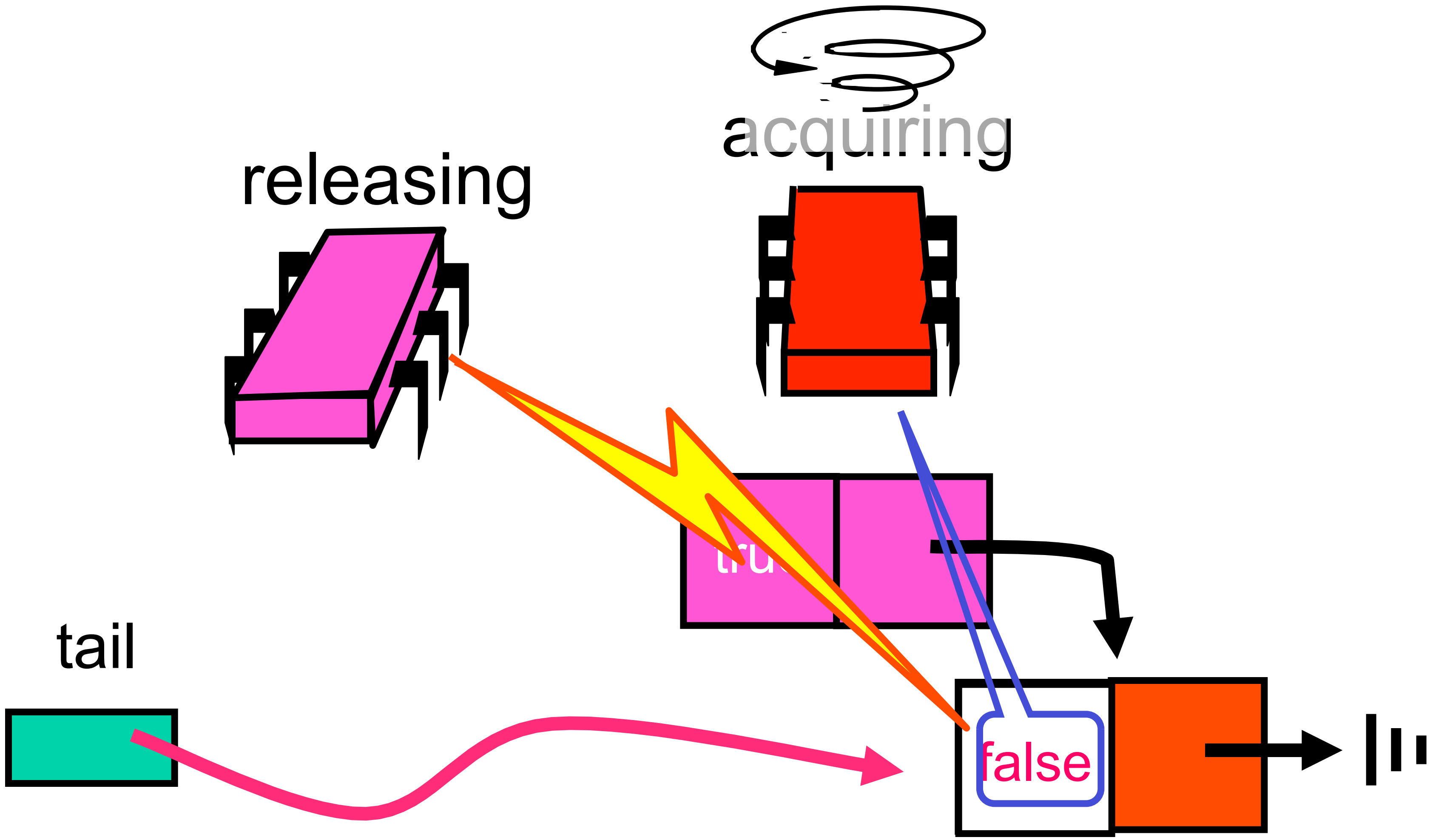
Acquiring



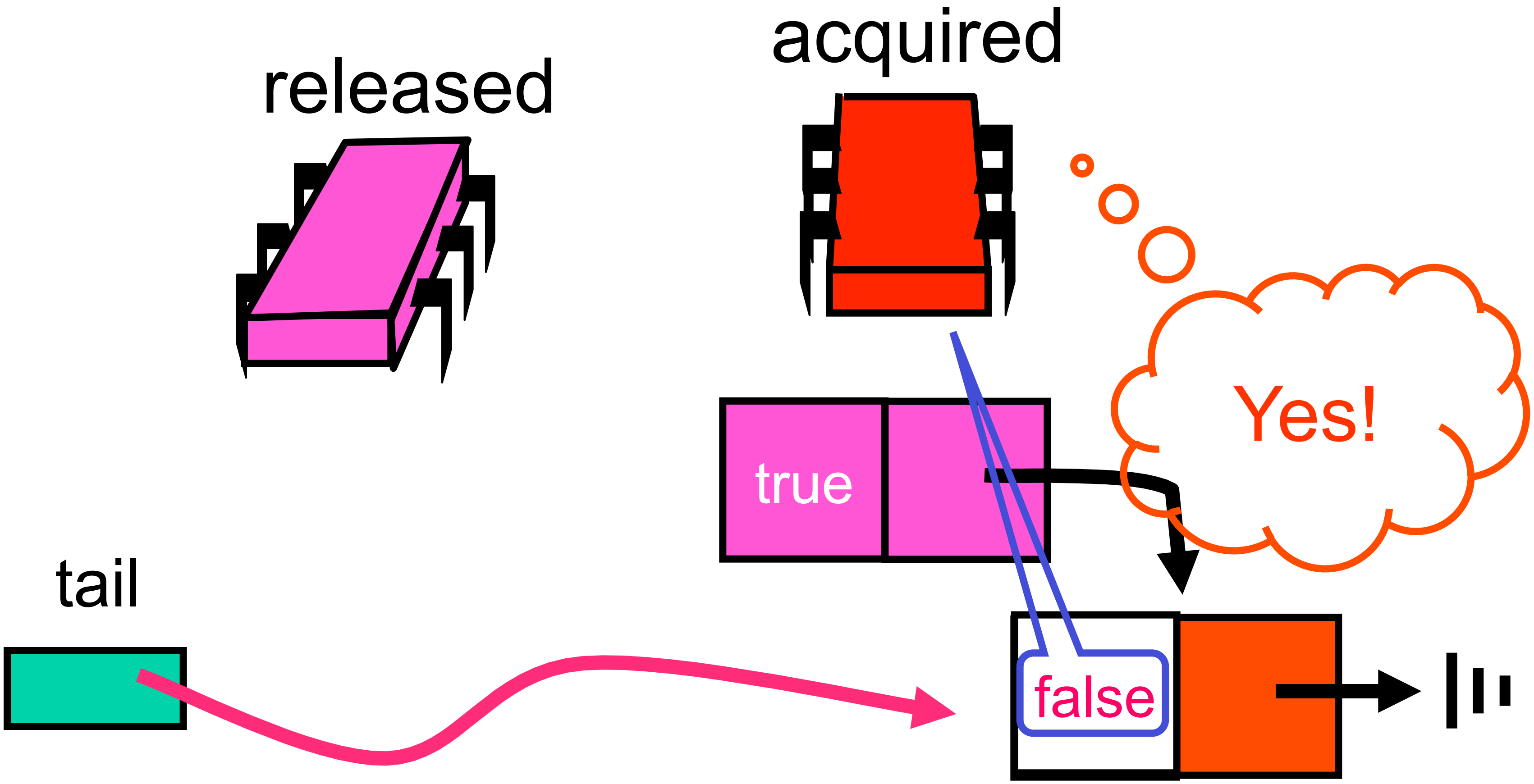
Acquiring



Acquiring



Acquiring



MCS Queue Lock

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

Must be volatile

MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail; //null initially
 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) {}
  }}}
```

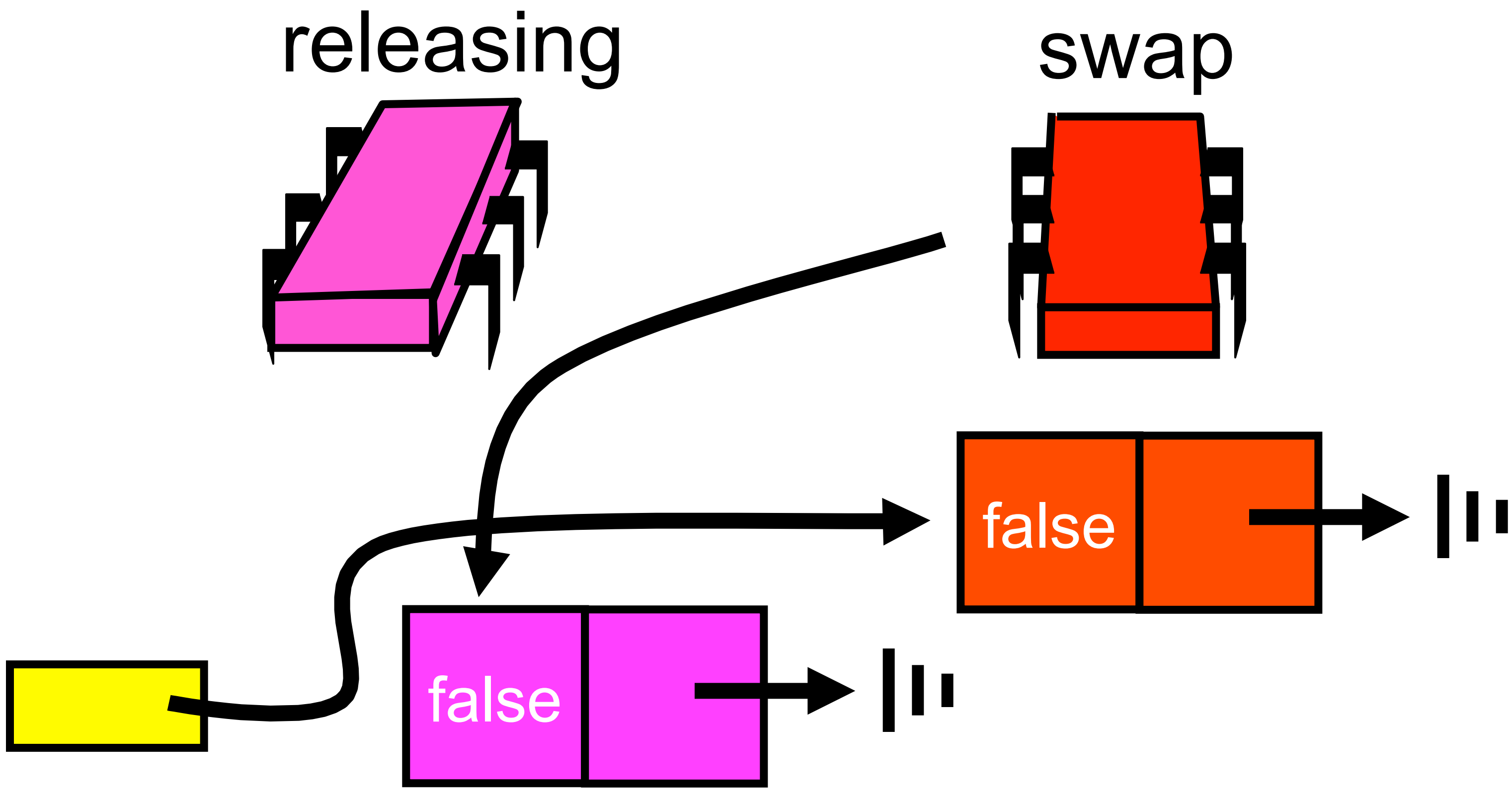
Wait until unlocked

MCS Queue Lock

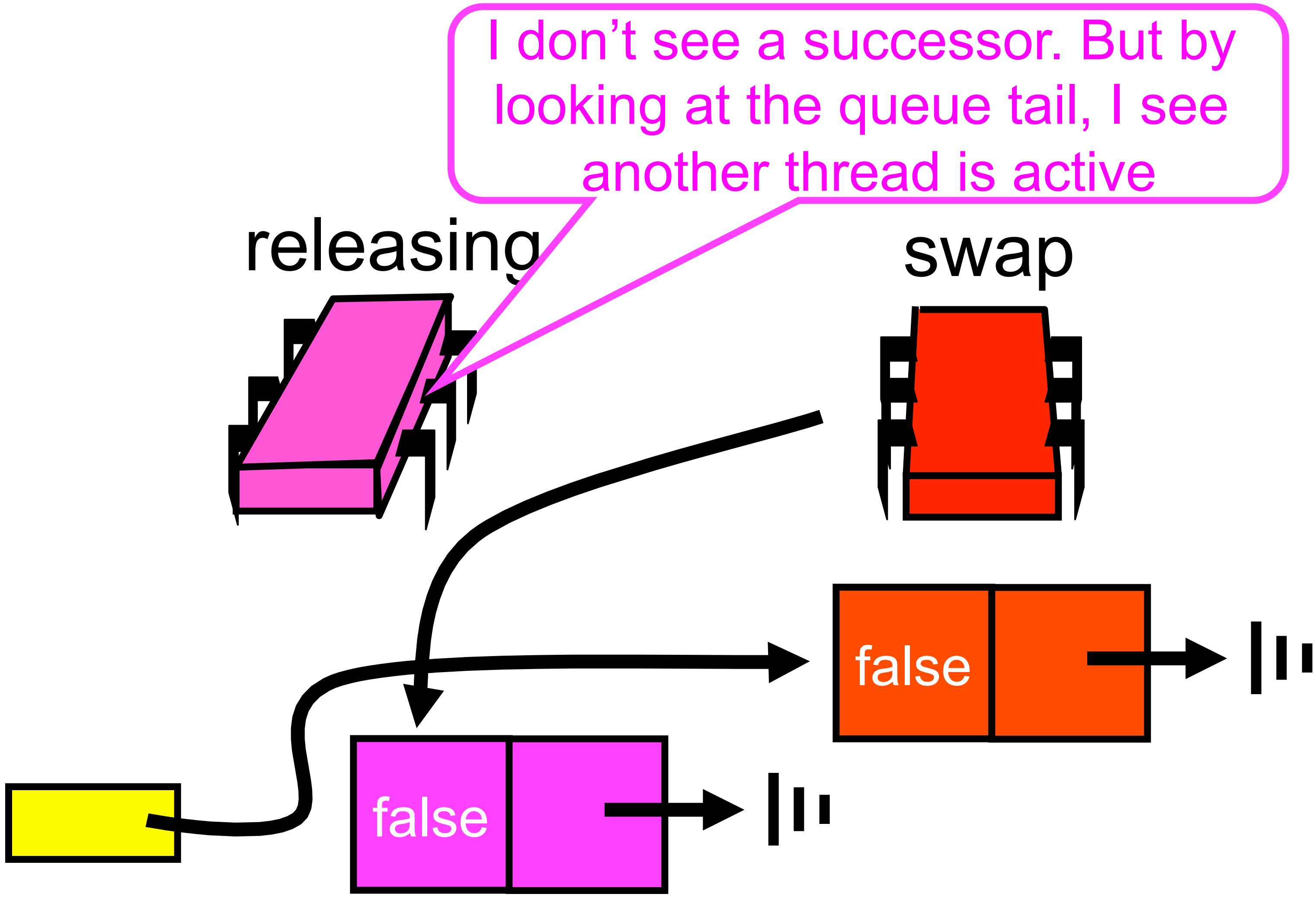
```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

Not Atomic

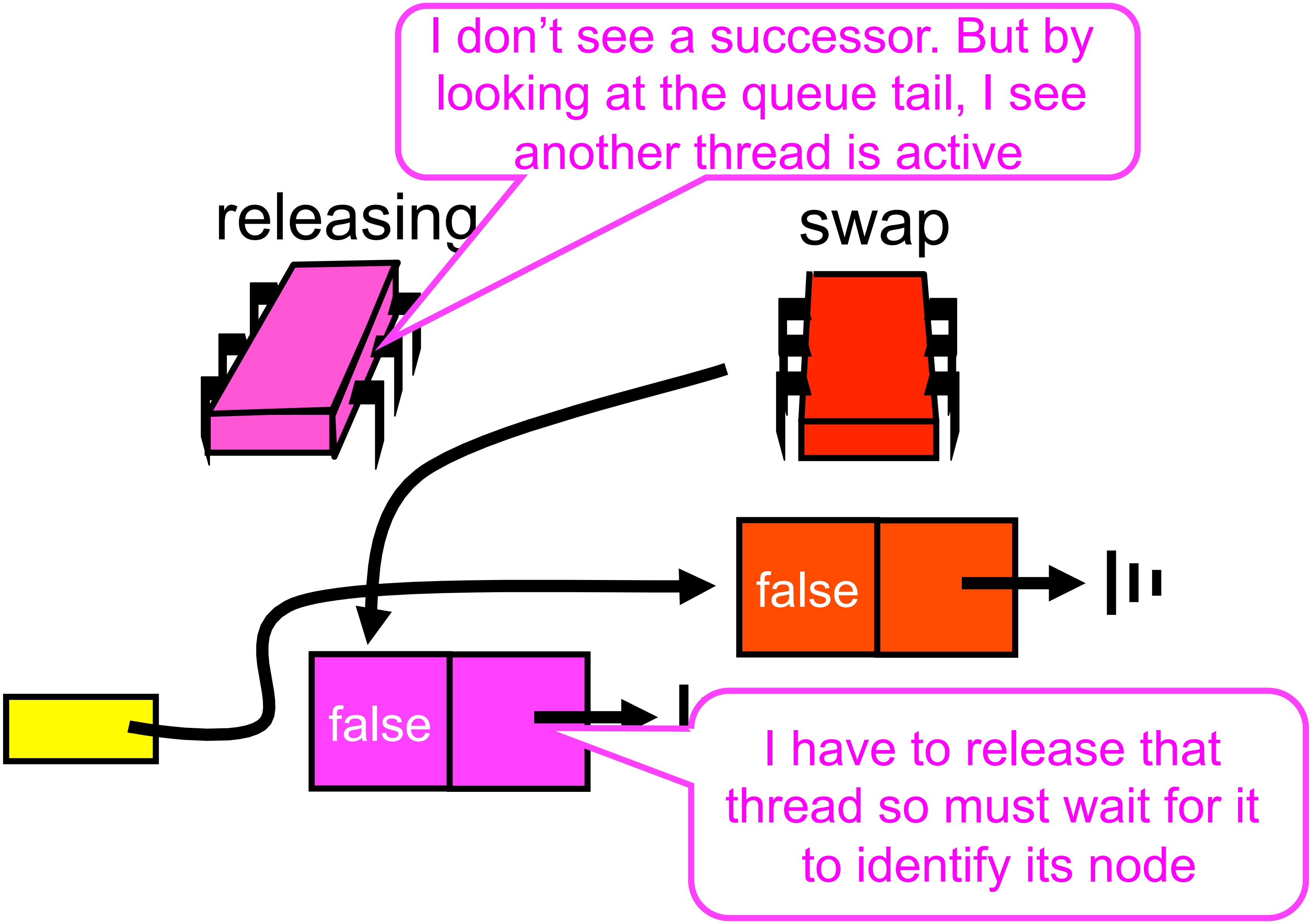
Purple Release



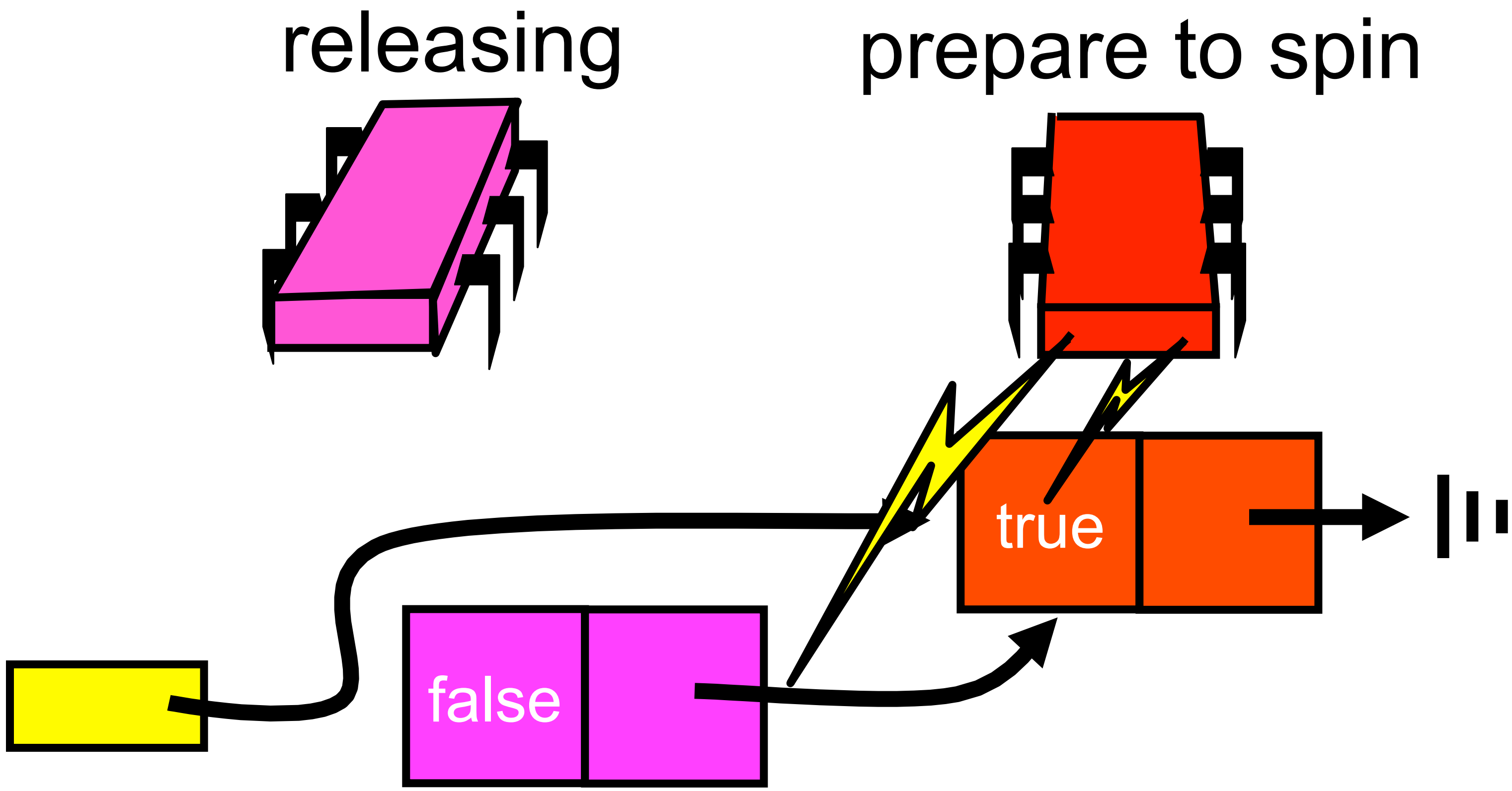
Purple Release



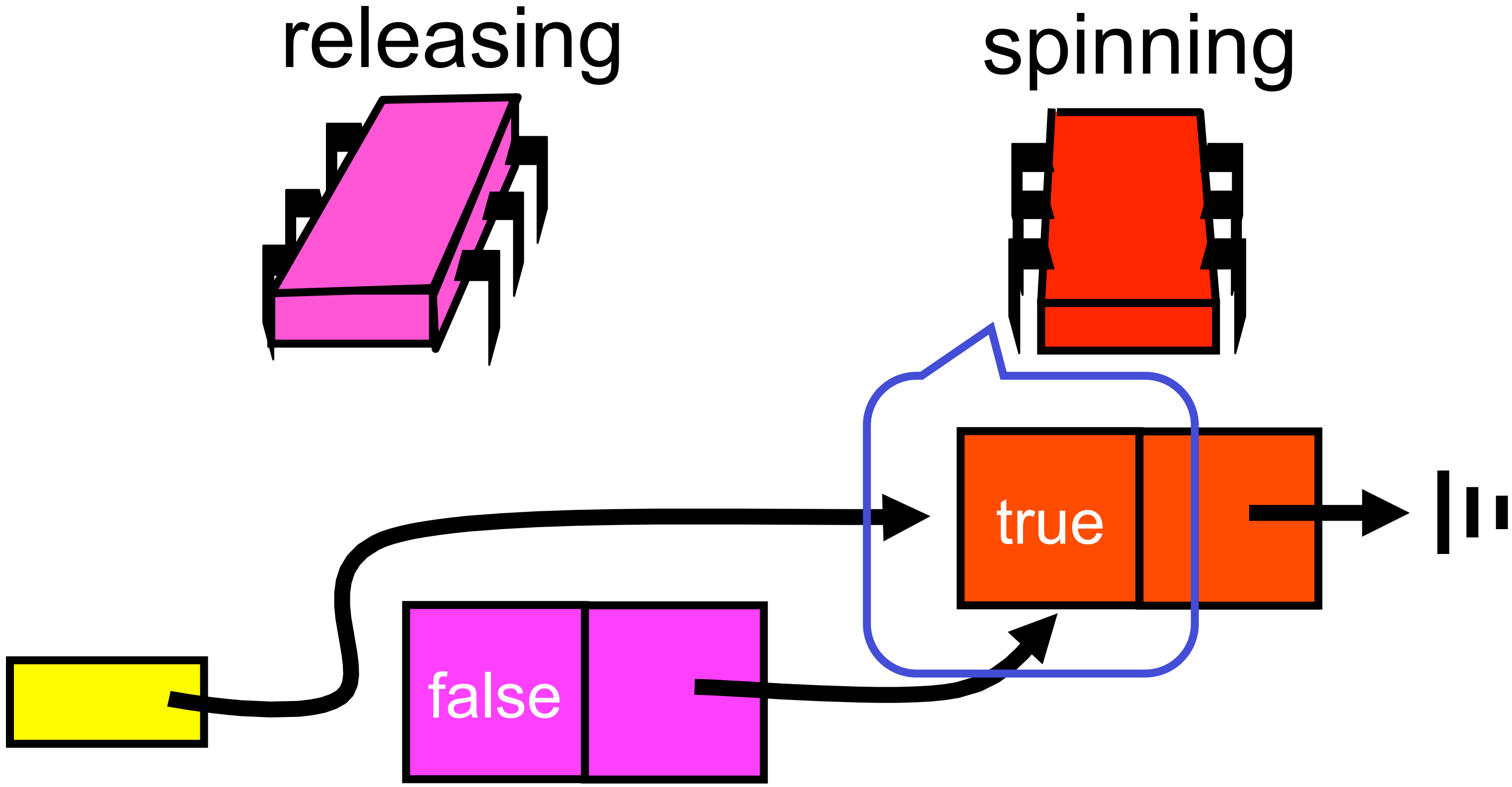
Purple Release



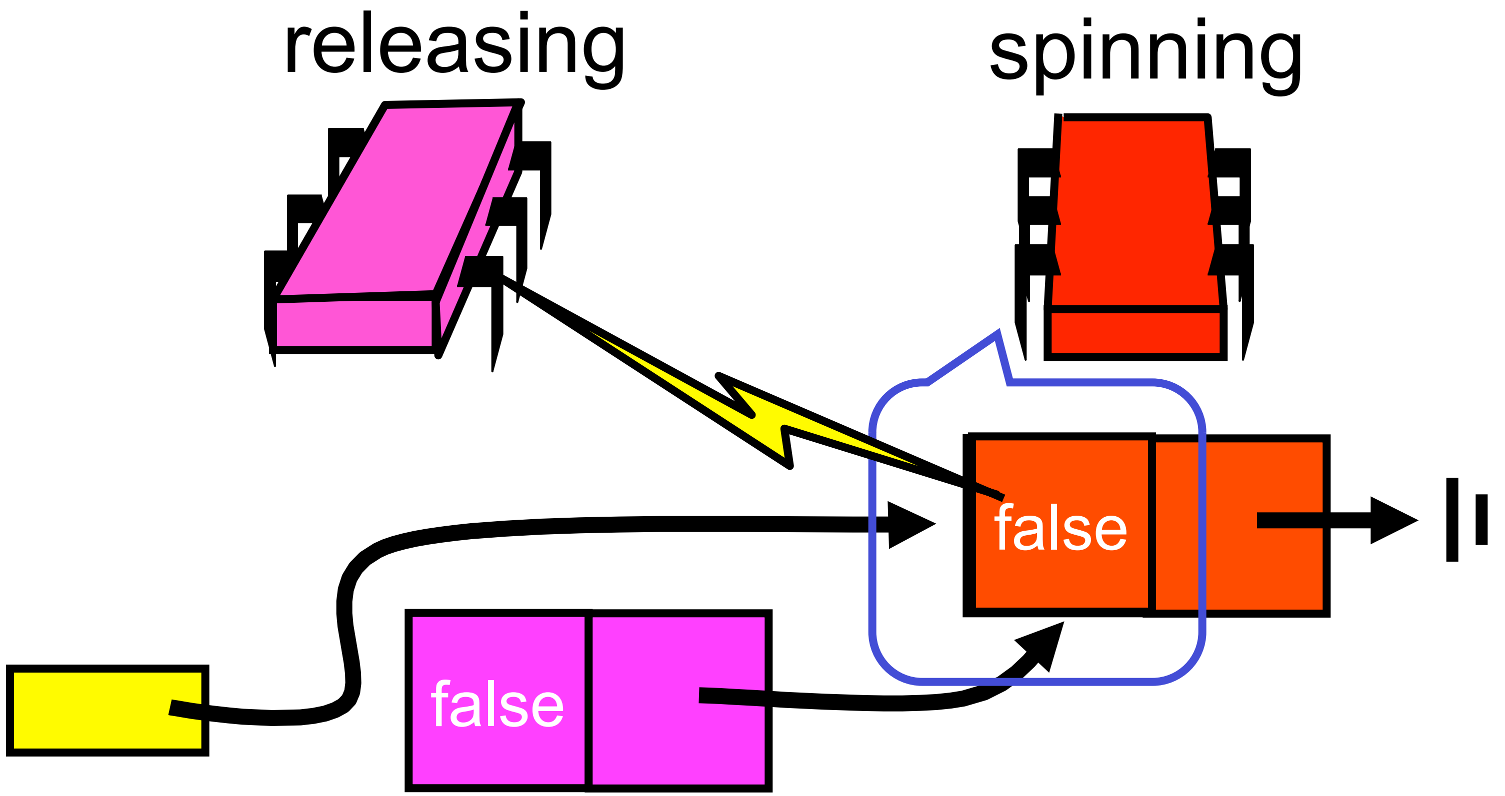
Purple Release



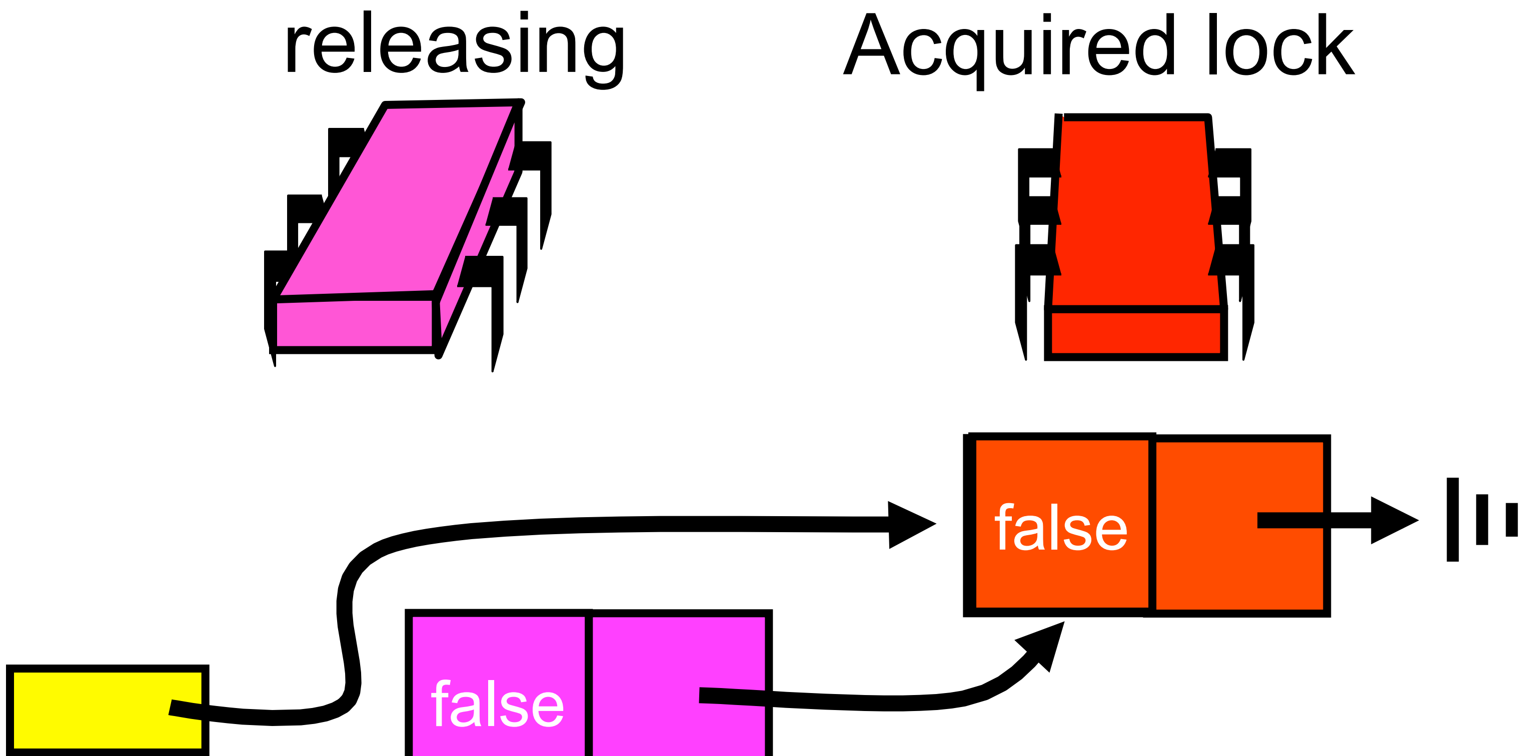
Purple Release



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

MCS Queue Lock

- Good
  - Works better for NUMA architecture
  - Qnodes can be recycled to have the space complexity as CLH locks
- Bad
  - Require spinning (sometimes) to release a lock
  - Requires more CAS than CLH locks

References

TTASLock: C. P. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. ACM Transactions on Programming Languages and Systems (TOPLAS), 10(4):579–601, 1988.

Anderson's Lock: T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems. 1990;1(1):6–16.

CLH Lock: T. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington, Department of Computer Science, February 1993.

P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In Proc. of the 8th International Symposium on Parallel Processing (IPPS), pp. 165–171, April 1994.

MCS Lock: J. Mellor-Crummey, M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems. 1991;9(1): 21–65.