Locks and
Mutual Exclusion

Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Definitions

- Synchronization: using atomic operations to coordinate multiple concurrent threads that are using shared state
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - *excludes* the other while doing its work
- Critical Section: piece of code that only one thread can execute at once. Only one thread gets in.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

Motivation: “Too much milk”

- Consider two roommates who need to coordinate to get milk if out of milk:

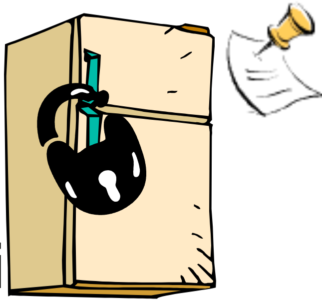
TOO MUCH MILK!

T		
3		
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering critical section
  - Unlock when leaving,
  - Wait if locked
    - Important idea: all synchronization involves waiting

- Example: Lock on the refrigerator
  - Lock it and take key if you are going to go buy mi

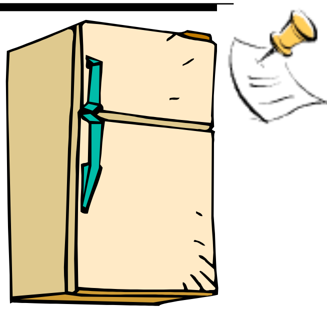


  - Too coarse-grained: refrigerator is unavailable
    - Roommate gets angry if he only wants OJ

Too Much Milk: Correctness Properties

- Correctness for “Too much milk” problem
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load (read) and store (write) operations
- Concurrent programs are non-deterministic due to many possible interleavings

Too Much Milk: Solution #1



- Use a note to avoid buying too much milk:
  - ➡ Leave a note before buying (kind of "lock")
  - ➡ Remove note after buying (kind of "unlock")
  - ➡ Don't buy if note (wait)

```
if (noMilk) {
        if (noNote) {
            leave Note;
            buy milk;
            remove note;
        }
}
```

Context-switch point

Result?

- ➡ Still too much milk but only occasionally!

Too Much Milk: Solution #1½

- Another try:

```
leave Note;
if (noMilk) {
       if (noNote) {
          leave Note;
          buy milk;
       }
}
remove note;
```

- What happens here?
  - "`leave Note; buy milk;`" will never run.
  - No one ever buys milk!

To Much Milk Solution #2

- How about labeled notes?

Thread A

```
leave note A;        ← Context-switch point
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

Thread B

```
leave note B;        ← Context-switch point
if (noNoteA) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

- Does this work? Still no
- Possible for neither thread to buy milk
  - Thread A leaves note A; Thread B leaves note B; each sees the other's note, thinking "*I'm* not getting milk, *You're* getting milk"
  - Each one thinks that the other is getting it.

Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) {  //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {  //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes.
- It is safe to buy, or Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
- At Y:
  - if no note A, safe for B to buy

Solution 3.5

- Note that the solution is asymmetric!
  - Quzz: does it work if Thread B also has a symmetric while loop?

Thread A — Context-switch point

```
leave note A;
while (note B) {
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B — Context-switch point

```
leave note B;
while (note A) {
   do nothing;
}
if (noMilk) {
    buy milk;
}
remove note B;
```

No. Each thread can leave a note, then go into infinite while loop.

Solution #3 Discussions

- Solution #3 works, but it’s really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A’s code is different from B’s – what if lots of threads?
    - Code would have to be slightly different for each thread
- There’s a better way
  - Have HW provide better (higher-level) primitives
  - Build even higher-level programming abstractions on this new hardware support

Too Much Milk: Solution #4

- We need to protect a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Solution:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

The correctness conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region
- Bounded waiting
  - A thread that enters the entry section enters the critical section within some bounded number of operations.
- Failure atomicity
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity

```
while(1) {
Acquire (Lock)
    Critical section
    Exit section
Release (Lock)
}
```

Where are we going with synchronization?

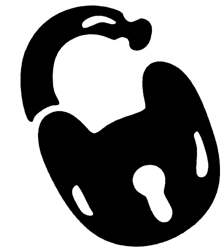
Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Summary

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available
- Shared data introduces challenges.
  - Programs must be properly synchronized
  - Without careful design, shared variables can become completely inconsistent
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - Construct various synchronization primitives

How to implement Locks?

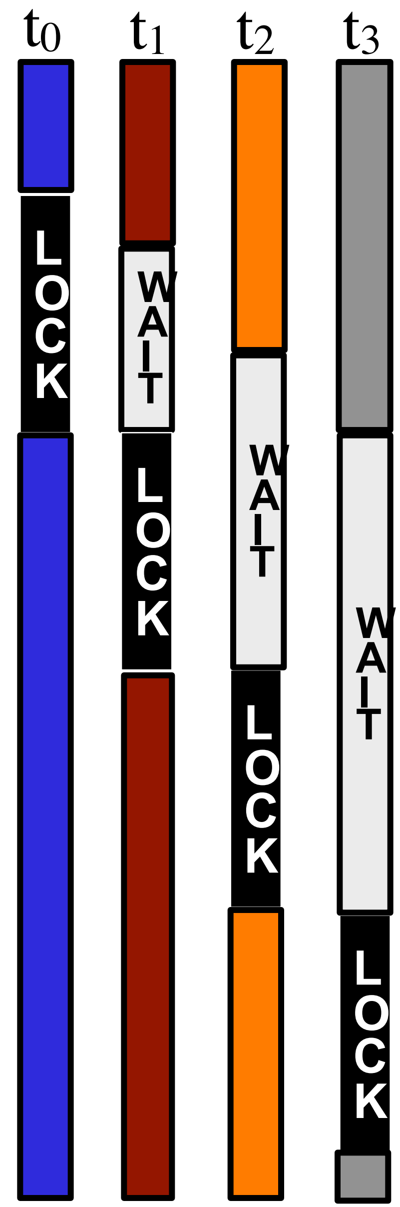
How to implement Locks? 

- Lock: prevents someone from doing something
  - Lock before entering critical section
  - Unlock when leaving, after accessing shared data
  - Wait if locked

- Hardware Lock instruction
  - Is this a good idea?
  - Complexity?
    - Done in the Intel 432
    - Each feature makes hardware more complex and slow
  - What about putting a task to sleep?
    - How do you handle the interface between the hardware and scheduler?

Lock-Based Mutual Exclusion

- Only one thread can hold a “lock” at a time
  - Used a provide serialized access to a data object
- If another threads tries to acquire a held lock
  - Must wait until other thread performs a release
- Performance implications
  - Lock contention limits parallelism
  - Lock acquire/release time adds overheads
- Correctness implications
  - Just one example:
    - Thread #1: Holds lock A, tries to acquire B
    - Thread #2: Holds lock B, tries to acquire A
    - Classic deadlock!



Read-Modify-Write (RMW)

- Implement locks using read-modify-write instructions
  - As an atomic and isolated action
    1. read a memory location into a register, **AND**
    2. write a new value to the location
  - Implementing RMW is tricky in multi-processors
    - Requires cache coherence hardware. Caches snoop the memory bus.

- Examples:
  - Test&set instructions (most architectures)
    - Reads a value from memory
    - Write “1” back to memory location
  - Compare & swap (68000)
    - Test the value against some constant
    - If the test returns true, set value in memory to different value
    - Report the result of the test in a flag
    - if [addr] == r1 then [addr] = r2;
  - Exchange, locked increment, locked decrement (x86)
  - Load linked/store conditional (PowerPC,Alpha, MIPS)

Simple Boolean Spin Locks

- Simplest lock:
  - Single variable, two states: **locked**, **unlocked**
  - When unlocked: atomically transition from unlocked to locked
  - When locked: keep checking (spin) until the lock is unlocked
- Busy waiting versus “blocking”
  - In a multicore, **busy wait** for other thread to release lock
    - Likely to happen soon, assuming critical sections are small
    - Likely nothing “better” for the processor to do anyway
  - In a single processor, if trying to acquire a held lock, **block**
    - The only sensible option is to tell the O.S. to context switch
    - O.S. knows not to reschedule thread until lock is released
  - Blocking has high overhead (O.S. call)
    - IMHO, rarely makes sense in multicore (parallel) programs

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

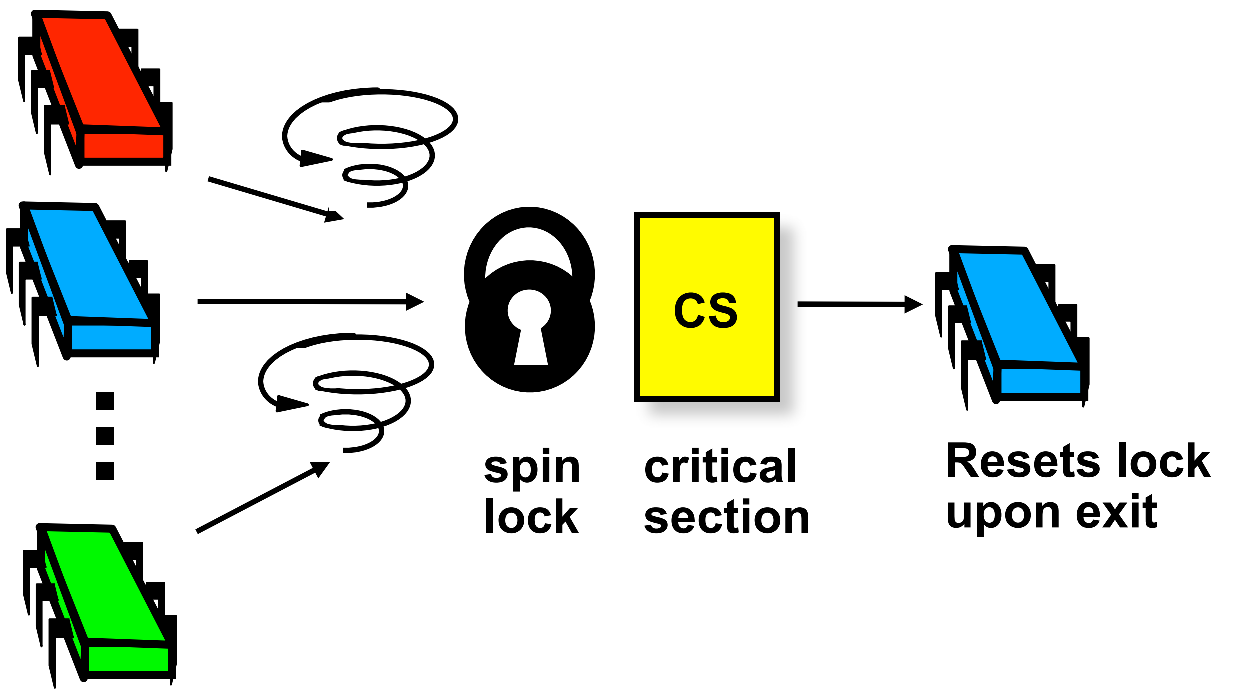
```
Lock::Acquire() {
while (test&set(lock) == 1)
  ; //spin
}
```

- If lock is free (lock_value == 0), then test&set reads 0 and sets value to 1 ➔ lock is set to busy and Acquire completes
- If lock is busy, the test&set reads 1 and sets value to 1 ➔ no change in lock's status and Acquire loops

```
Lock::Release() {
    *lock = 0;
}
```

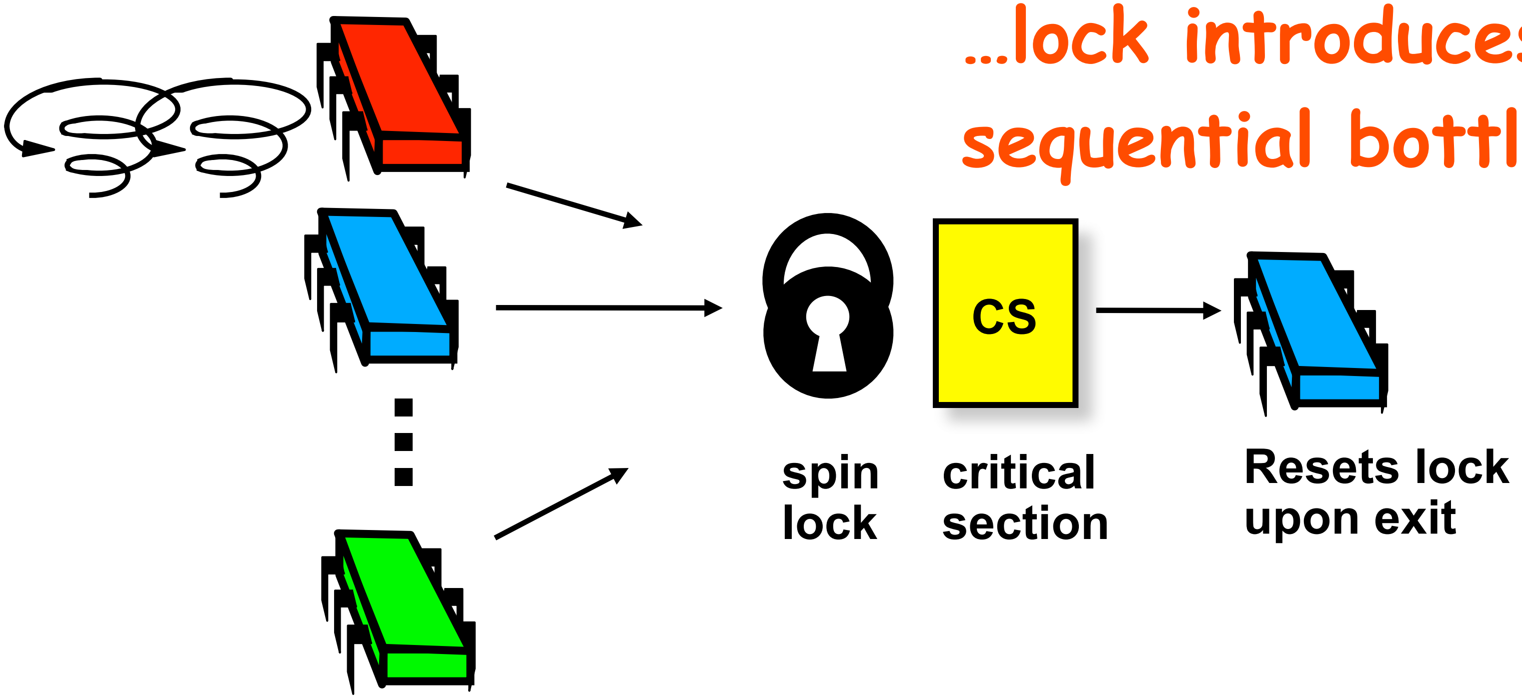
- Does this lock have bounded waiting?

Basic Spin-Lock

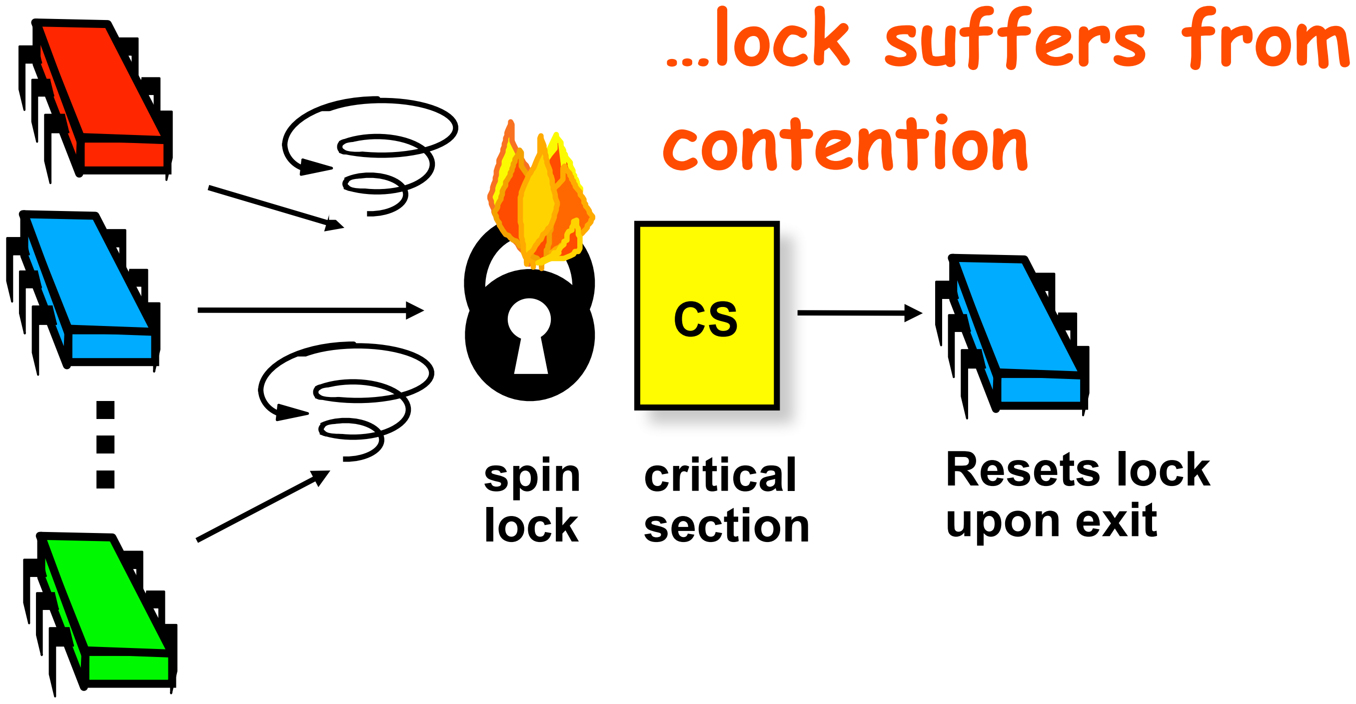


Basic Spin-Lock

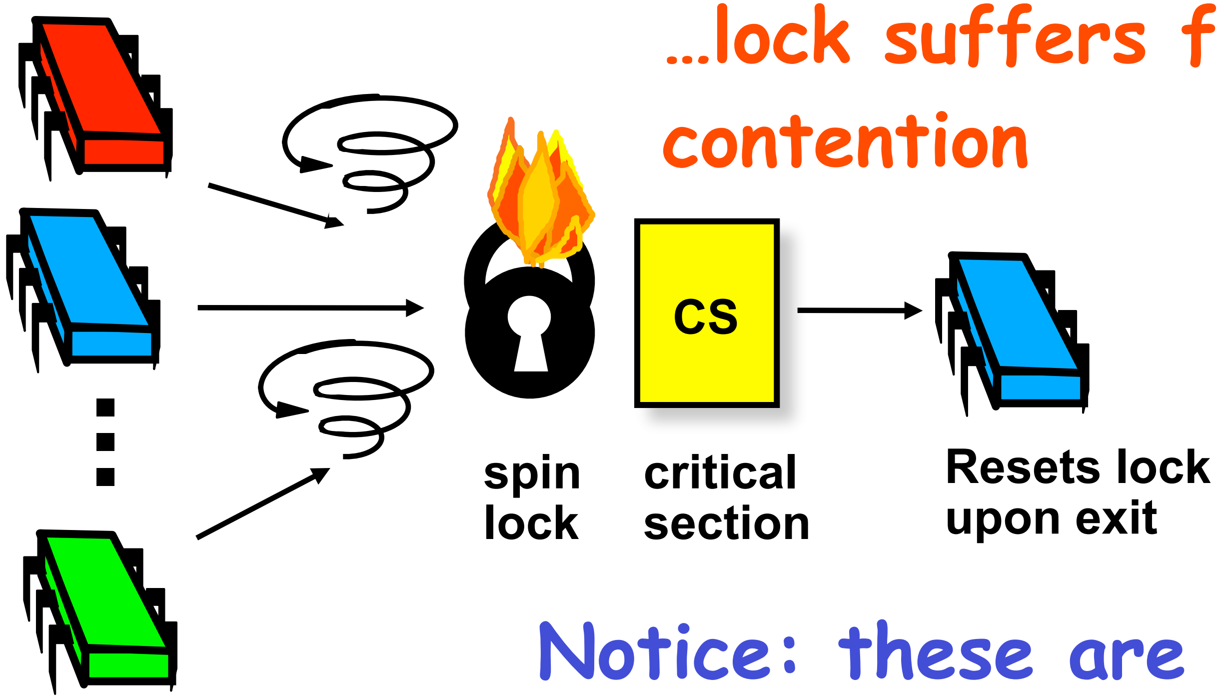
…lock introduces sequential bottleneck



Basic Spin-Lock



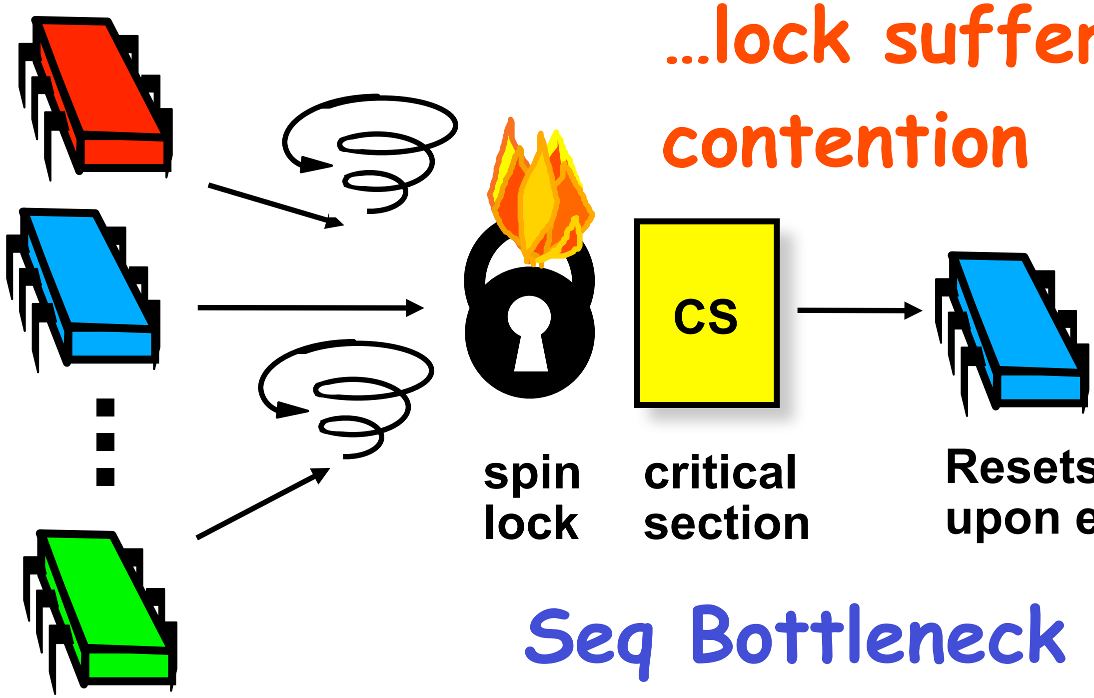
Basic Spin-Lock



…lock suffers from contention

Notice: these are distinct phenomena

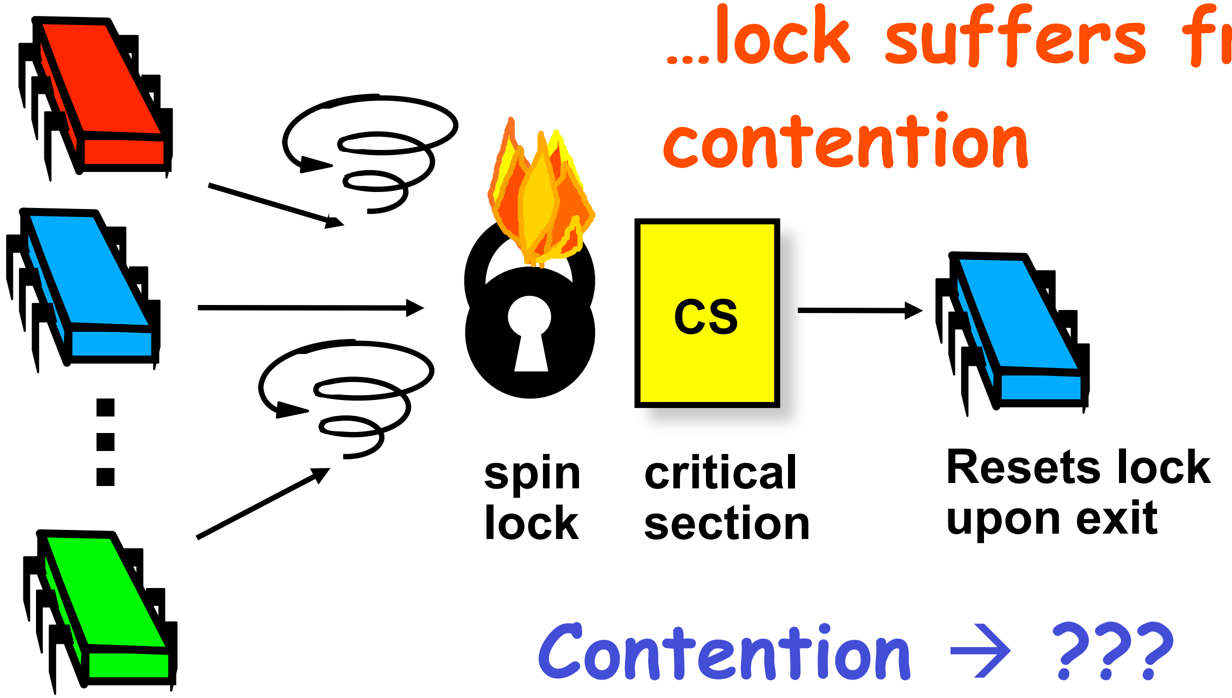
Basic Spin-Lock



...lock suffers from contention

Seq Bottleneck → no parallelism

Basic Spin-Lock



…lock suffers from contention

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;
 }
}
```

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

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

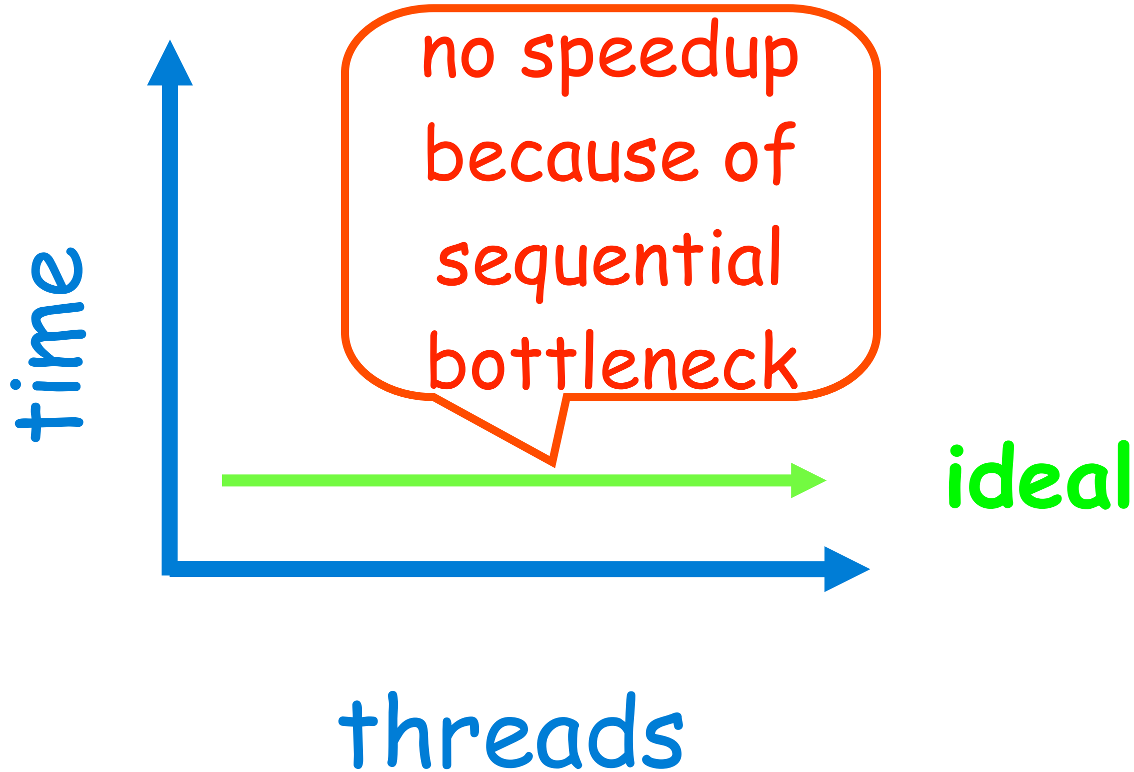
Space Complexity

- 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 $\Omega(n)$ lower bound?
- We used a RMW operation…

Performance

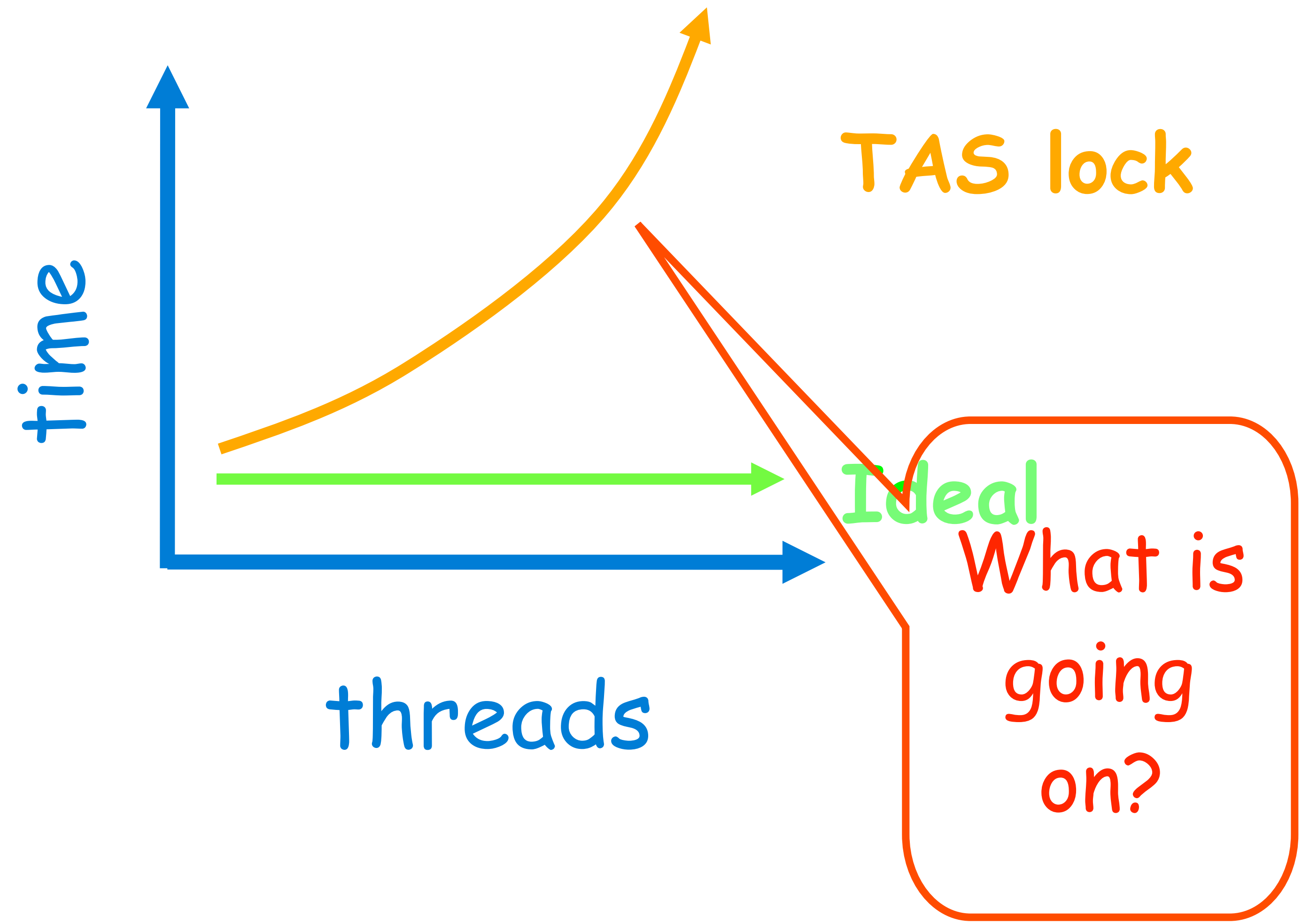
- Experiment
  - n threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?

Graph



no speedup because of sequential bottleneck
ideal
time
threads

Mystery #1



(1)

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

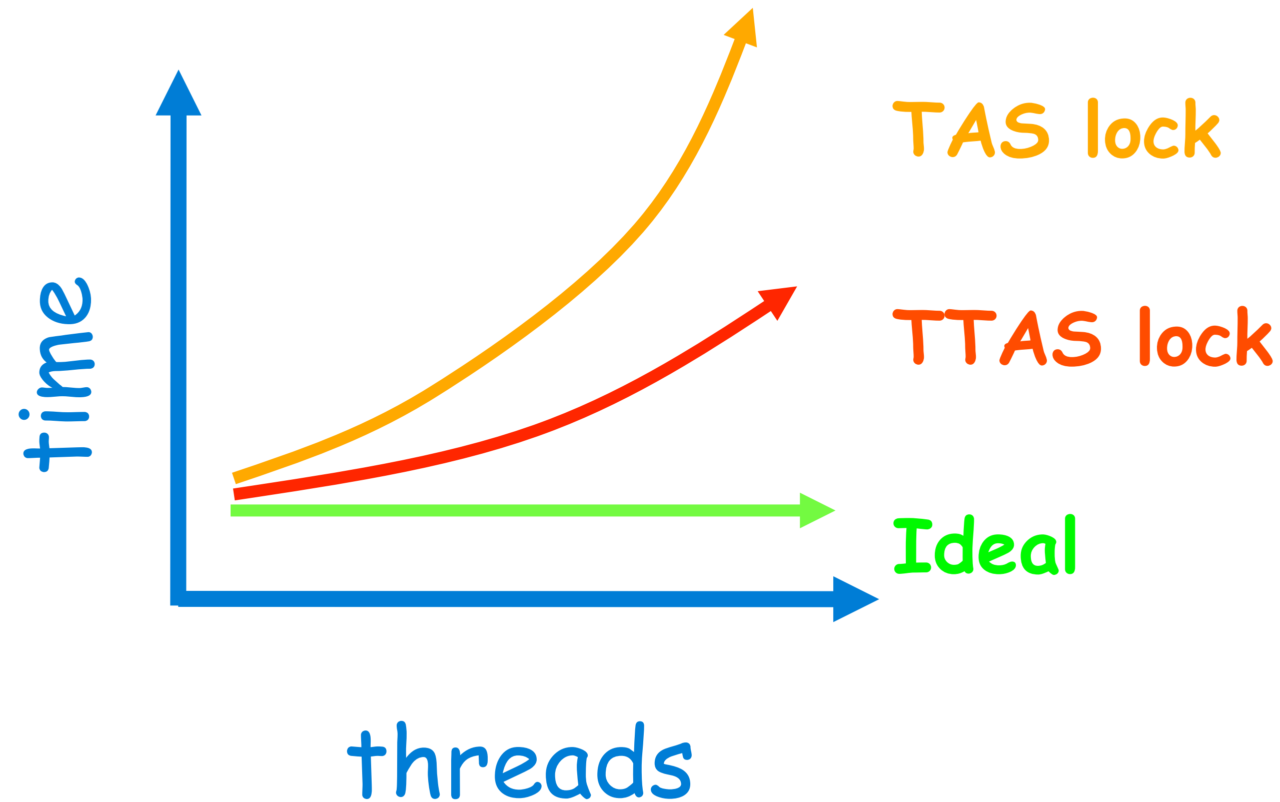
Test-and-test-and-set Lock

```
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



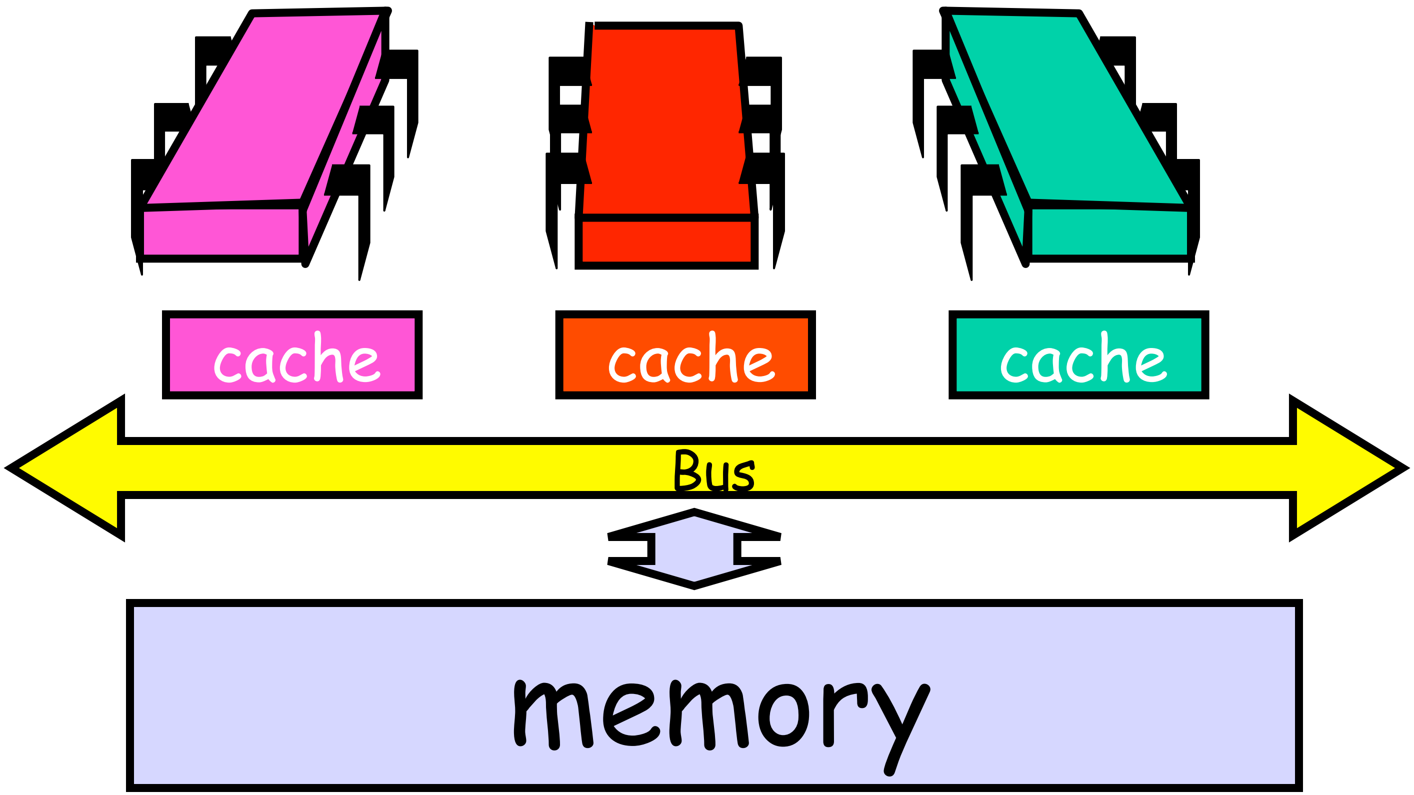
Mystery

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

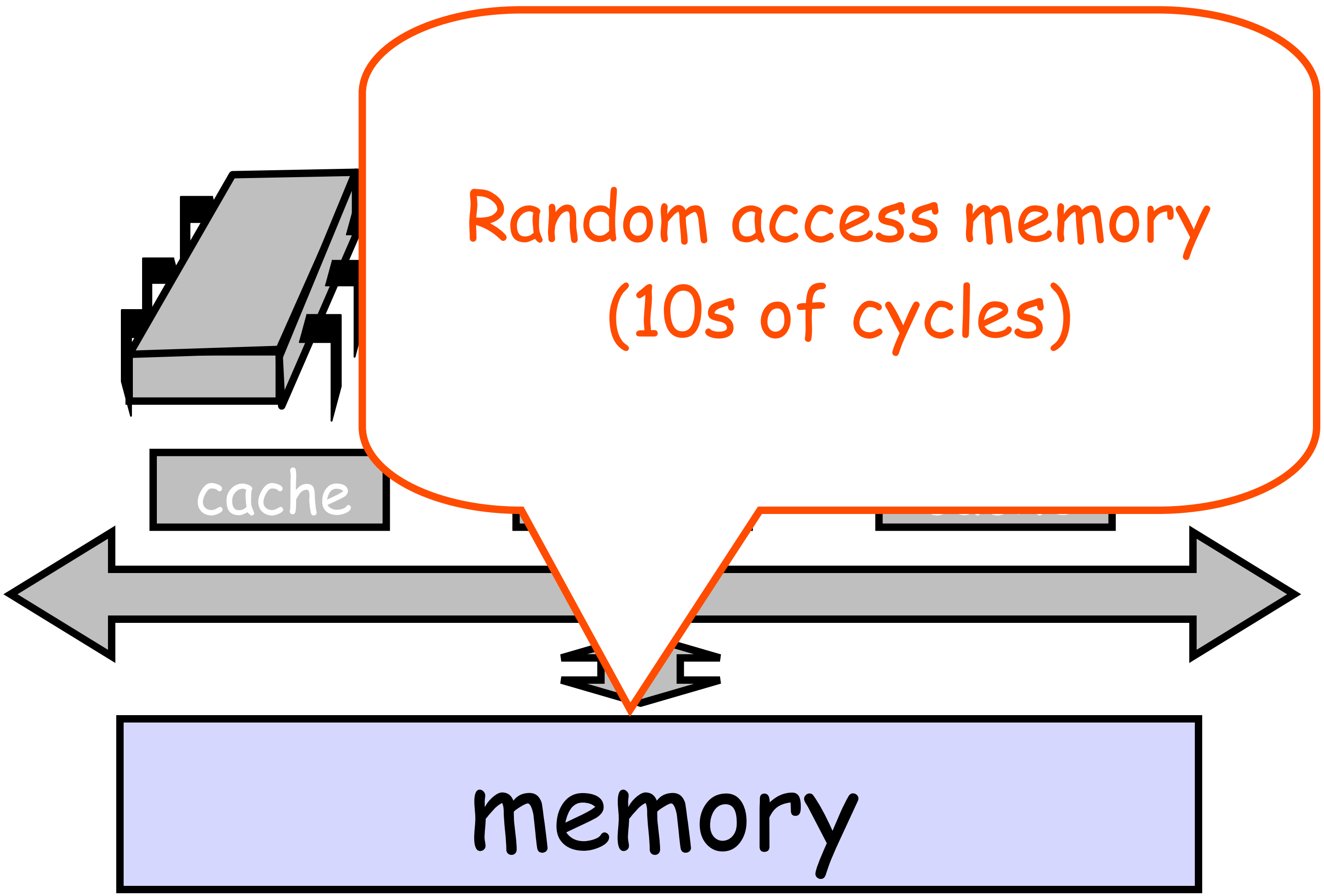
Opinion

- 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



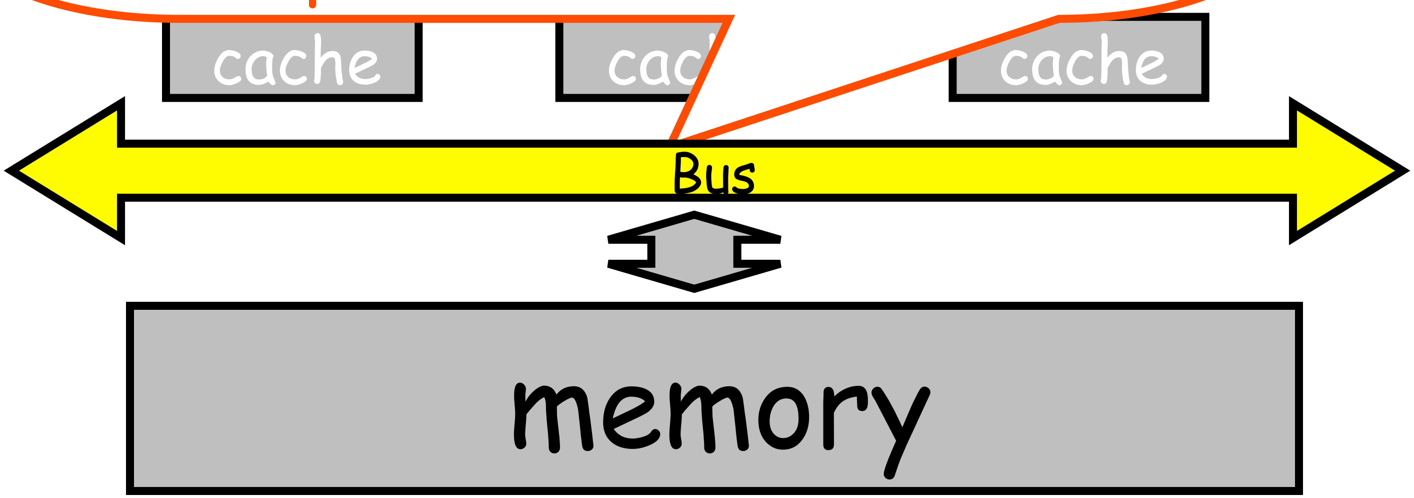
Bus-Based Architectures



Bus-Based Architectures

Shared Bus

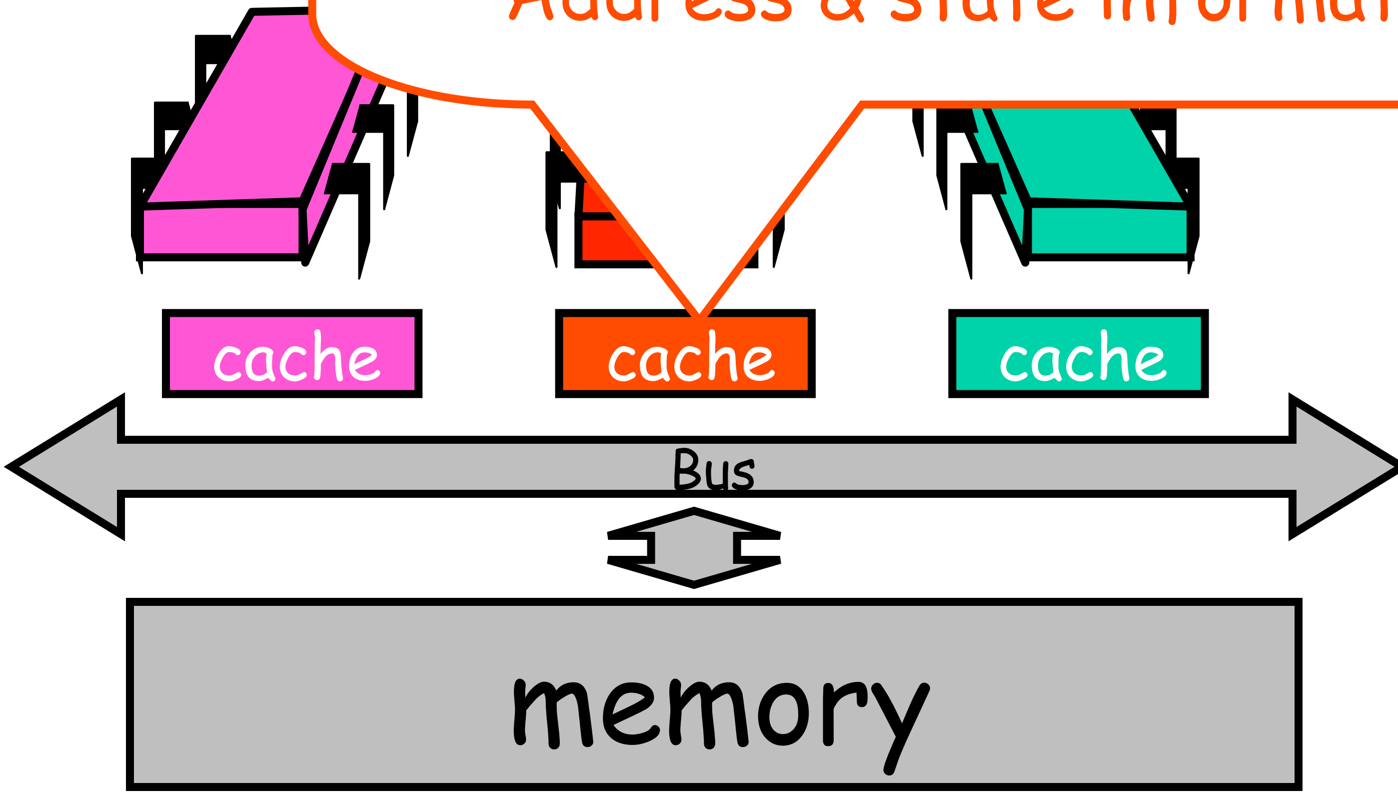
- broadcast medium
- One broadcaster at a time
- Processors and memory all "snoop"



Bus-B

Per-Processor Caches

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



Jargon Watch

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

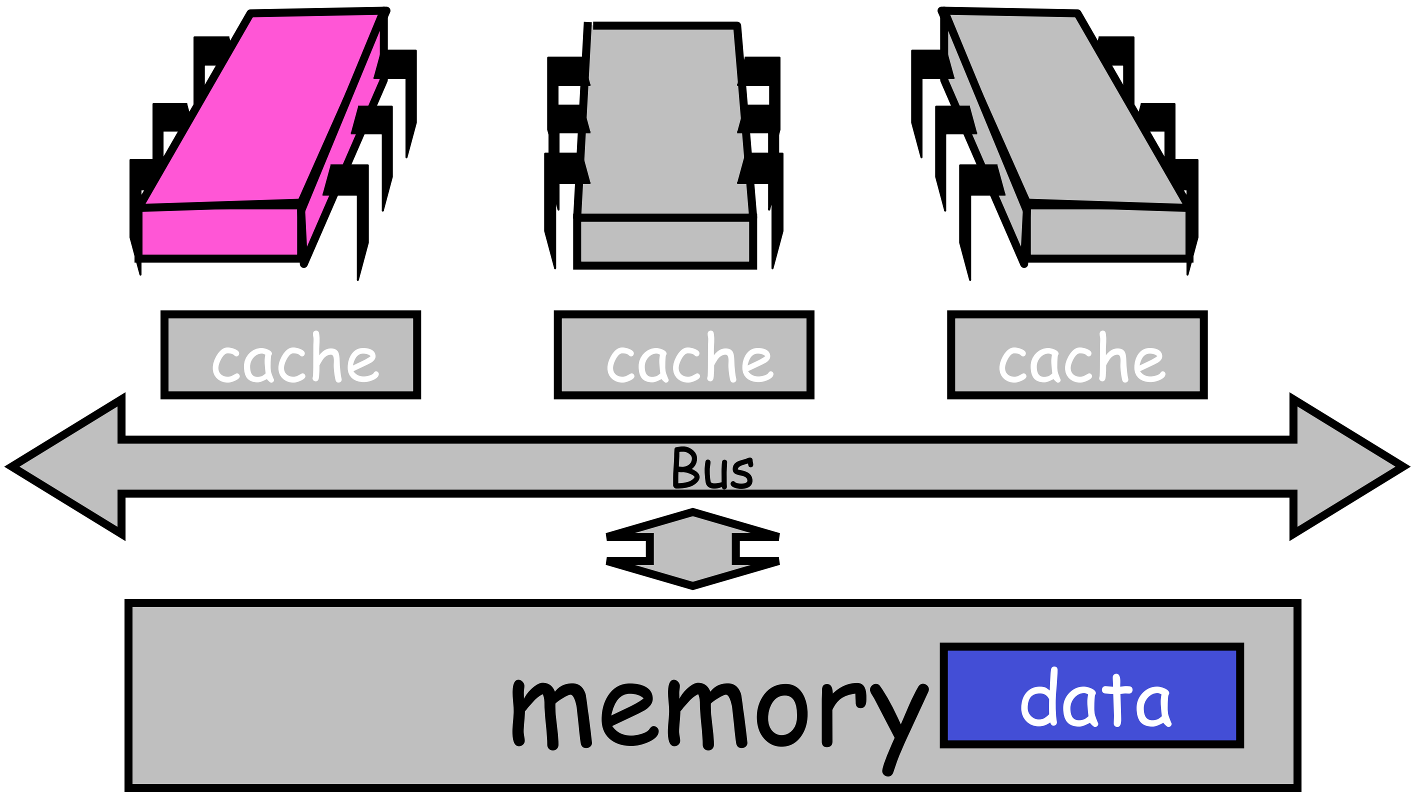
Jargon Watch

- 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™

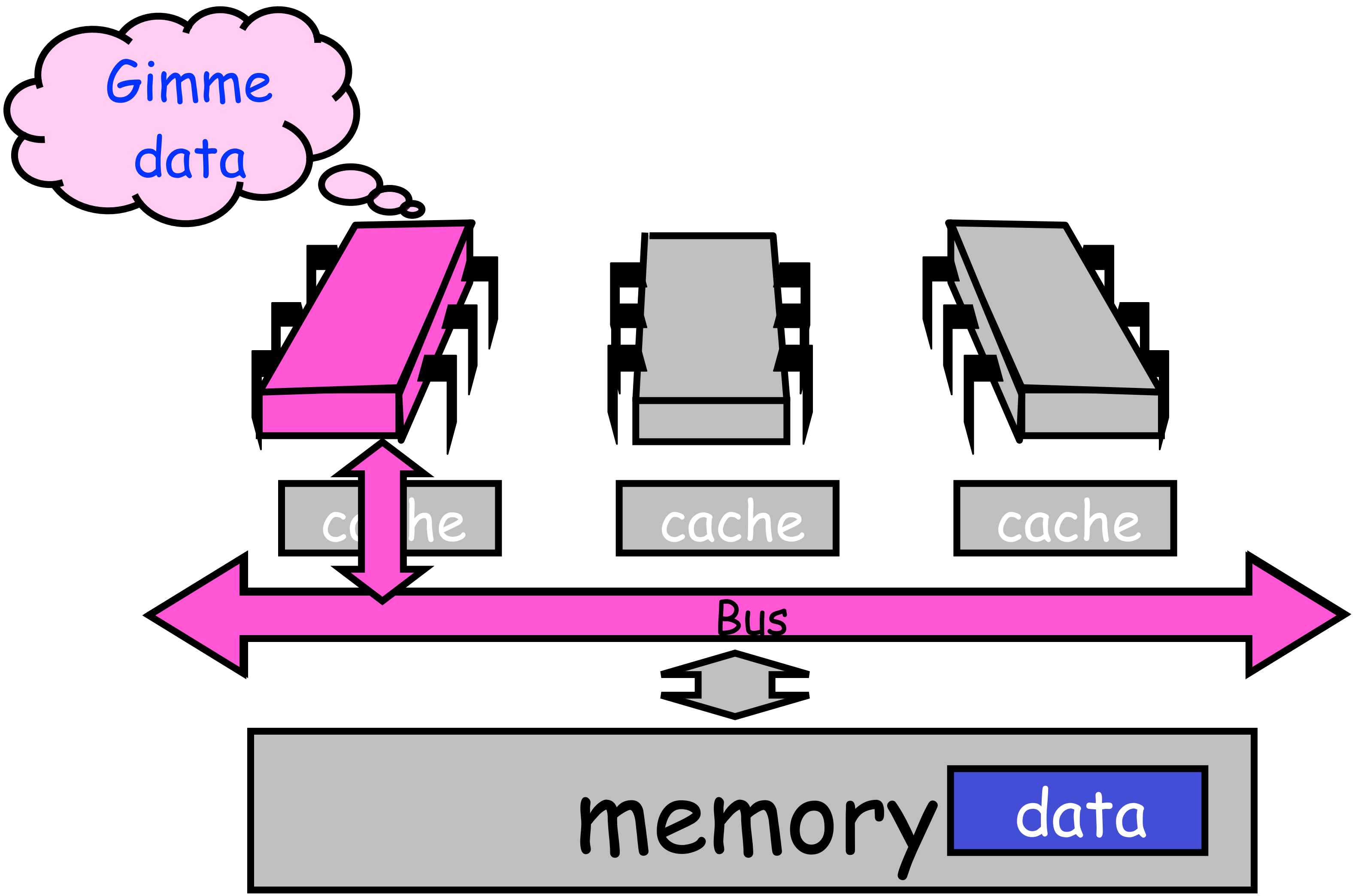
Cave Canem

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

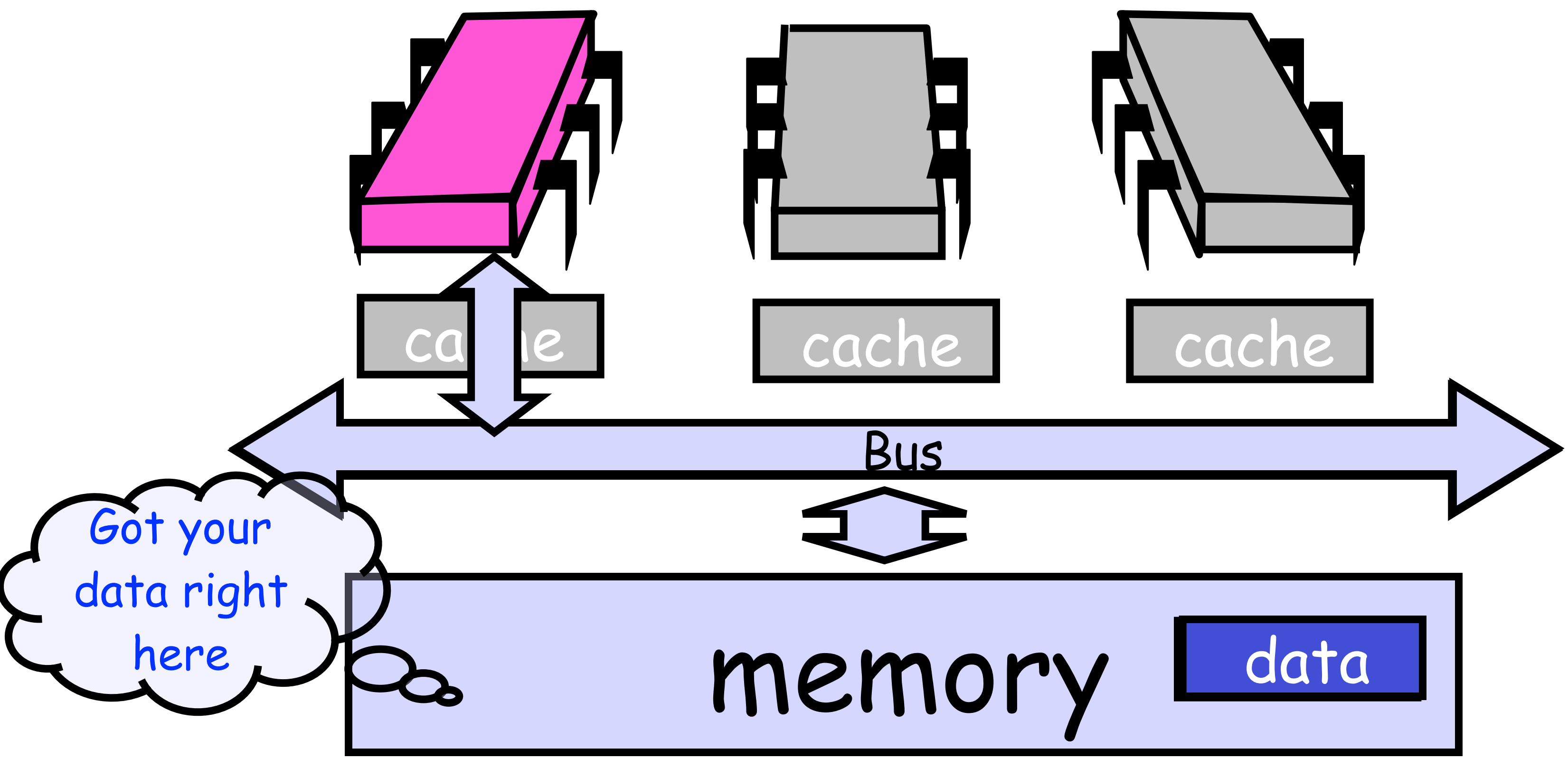
Processor Issues Load Request



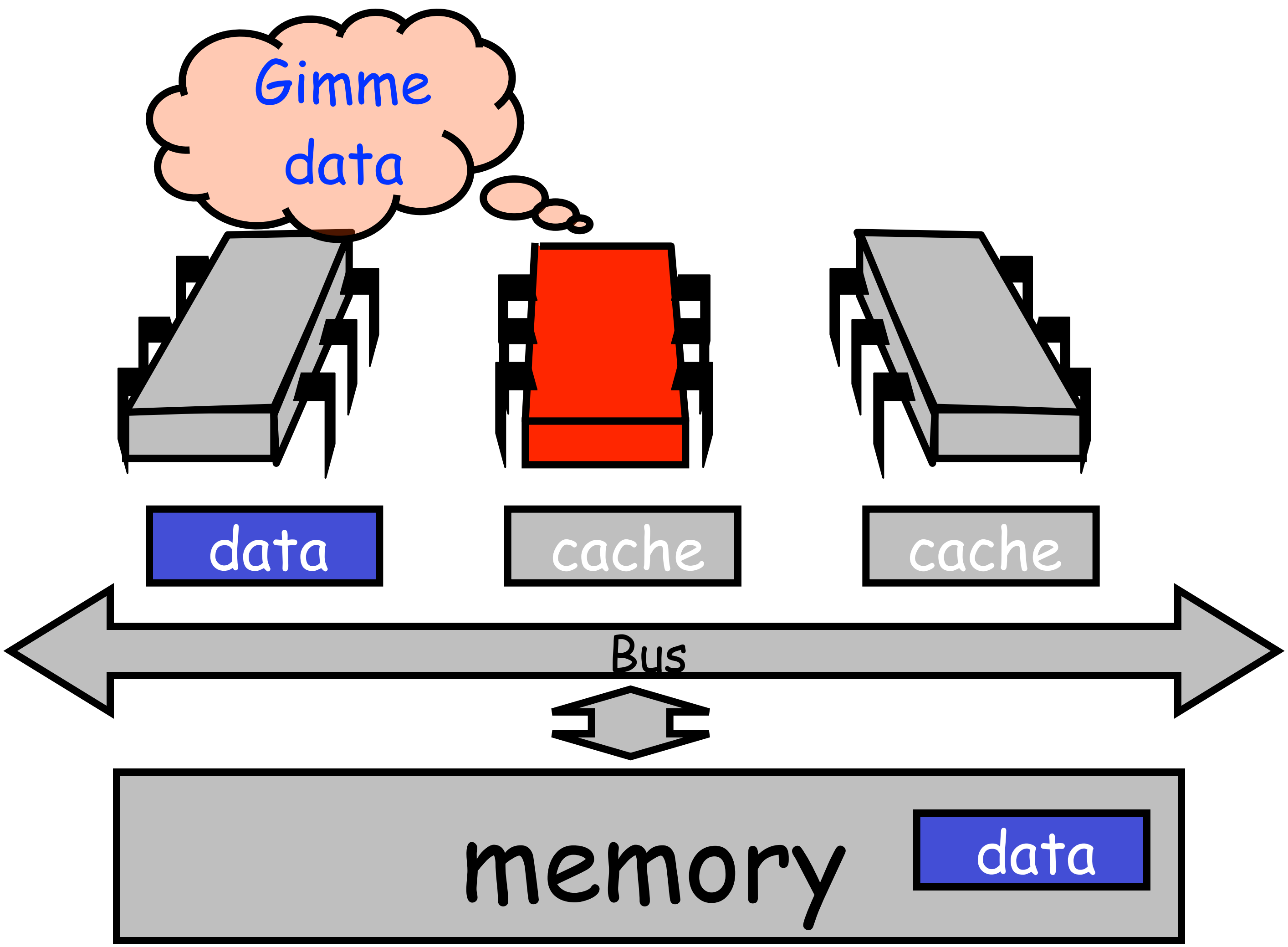
Processor Issues Load Request



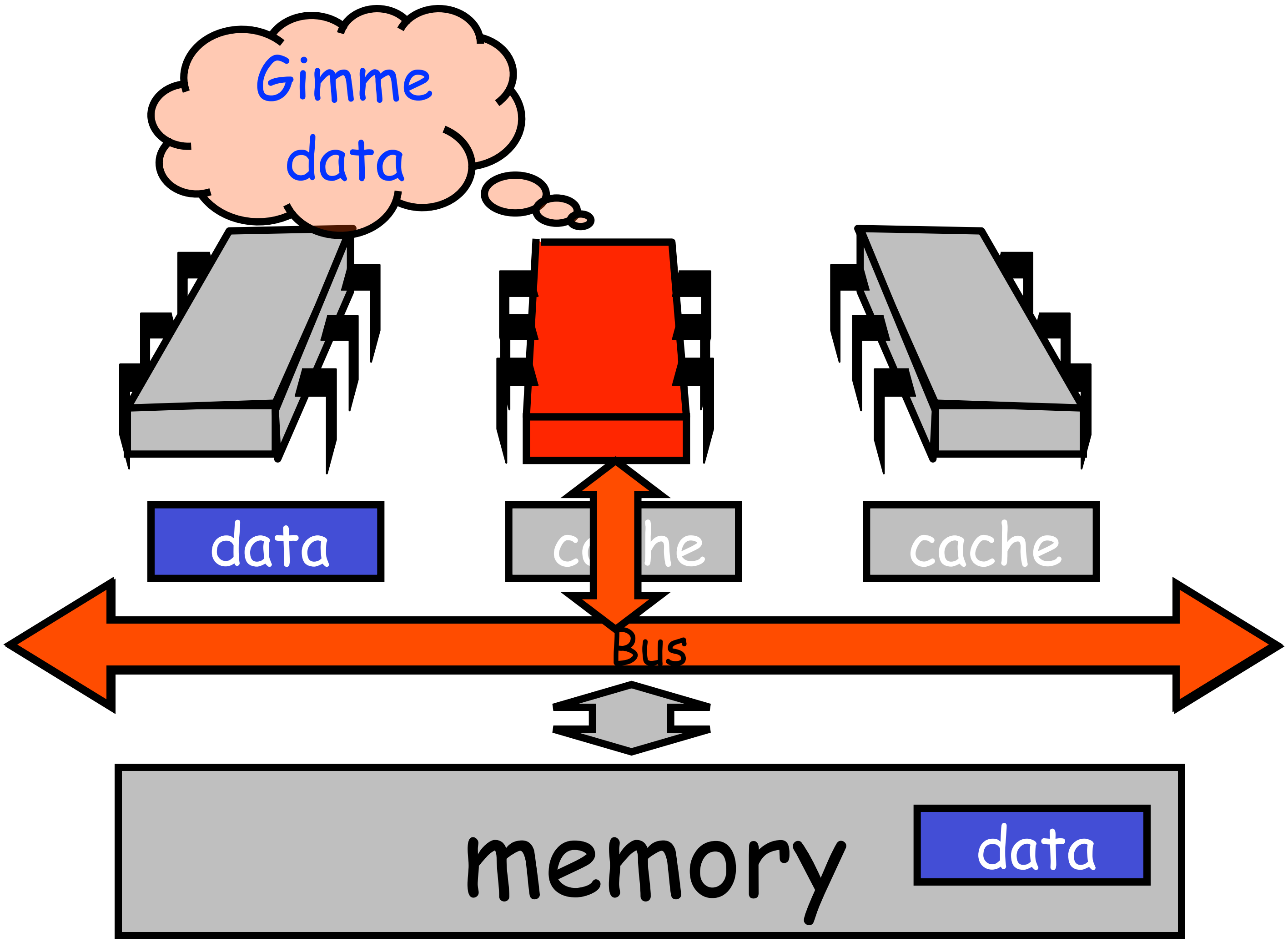
Memory Responds



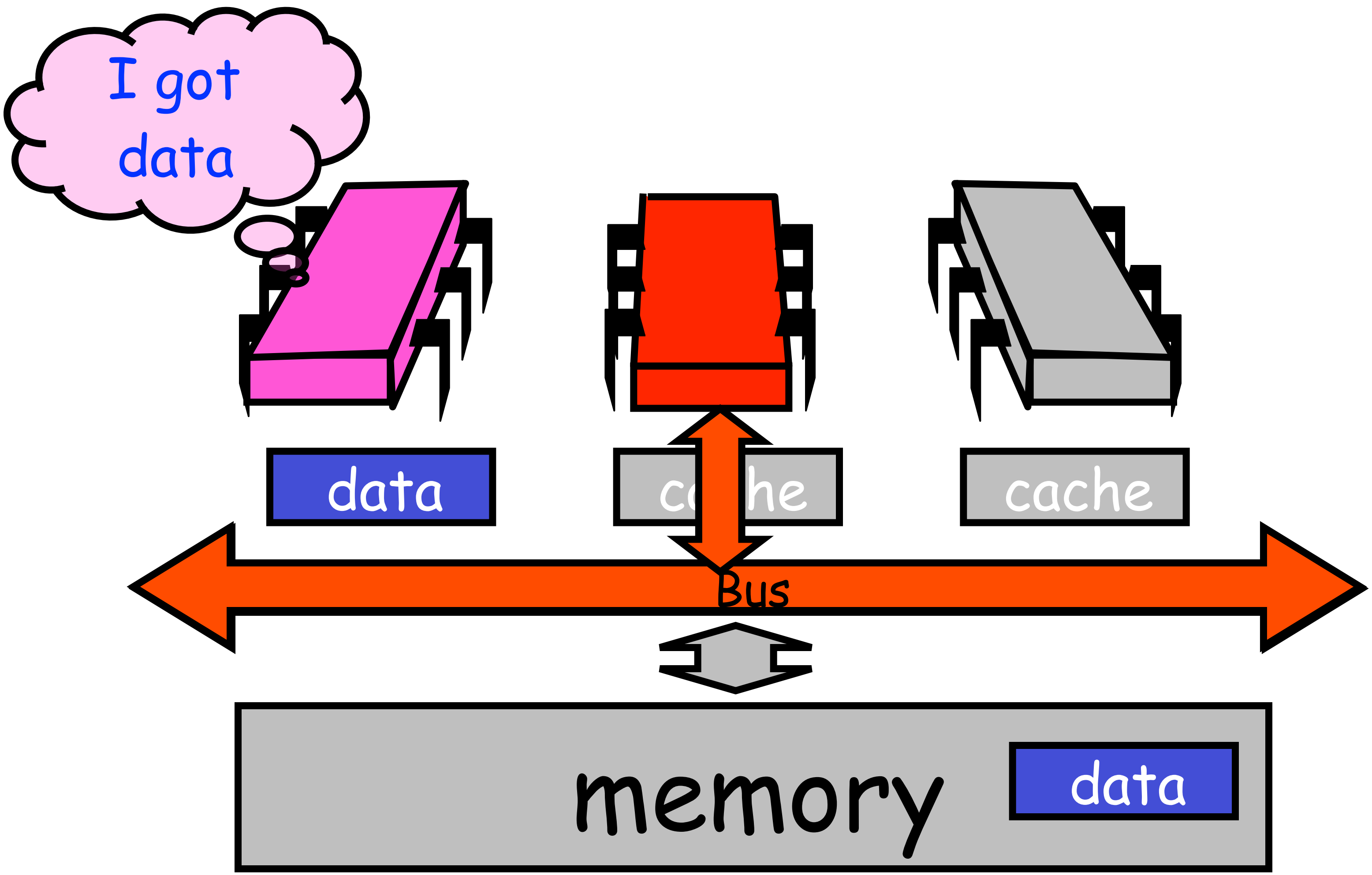
Processor Issues Load Request



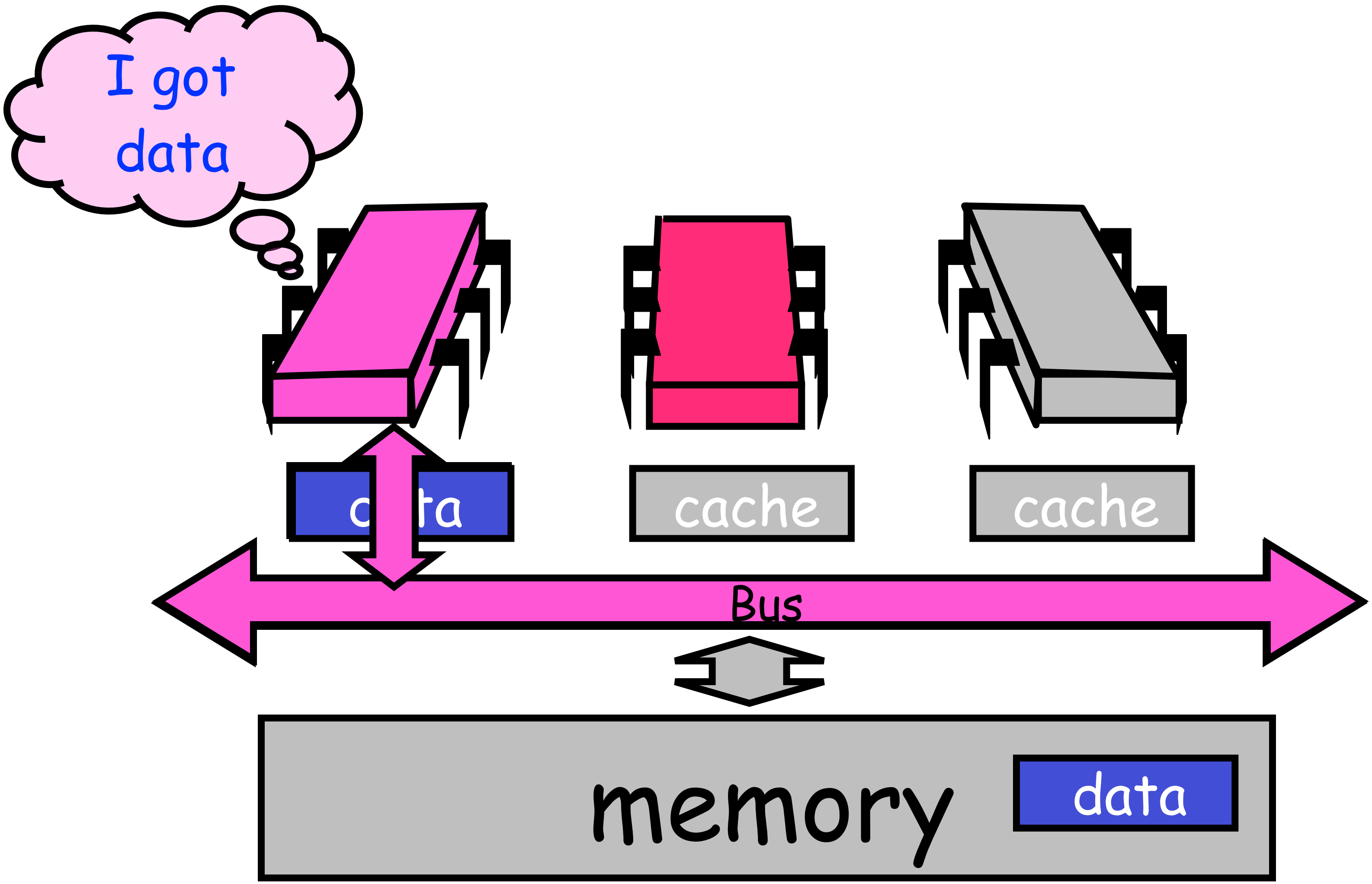
Processor Issues Load Request



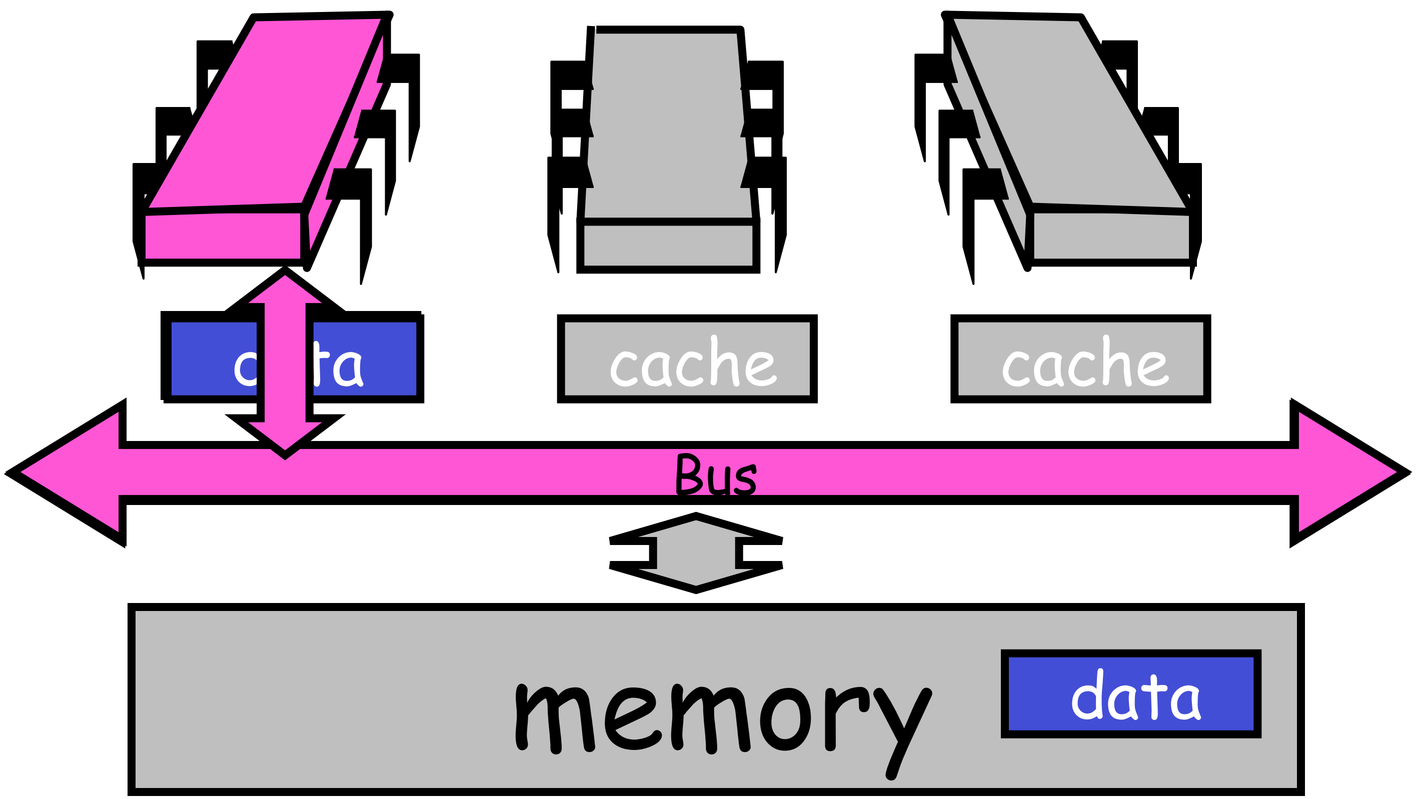
Processor Issues Load Request



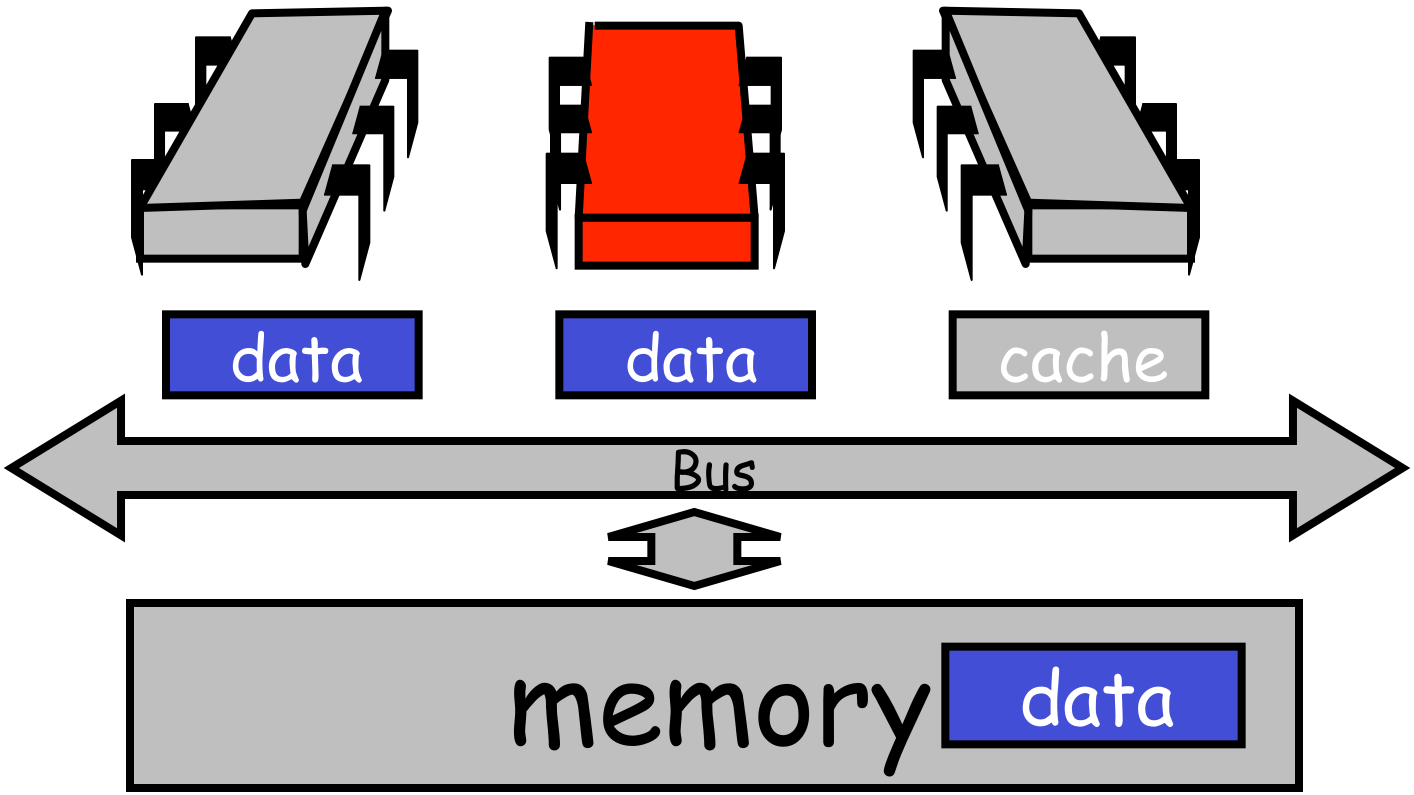
Other Processor Responds



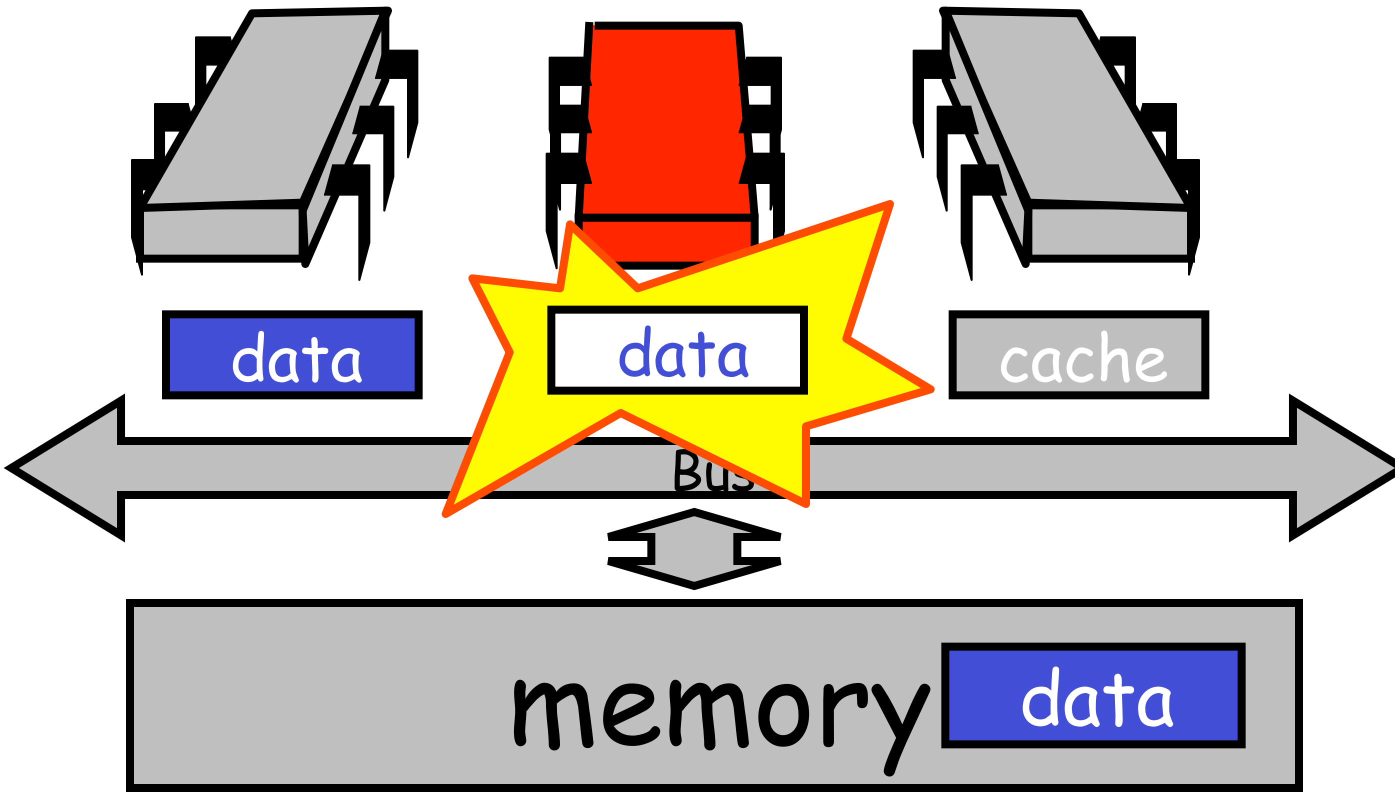
Other Processor Responds



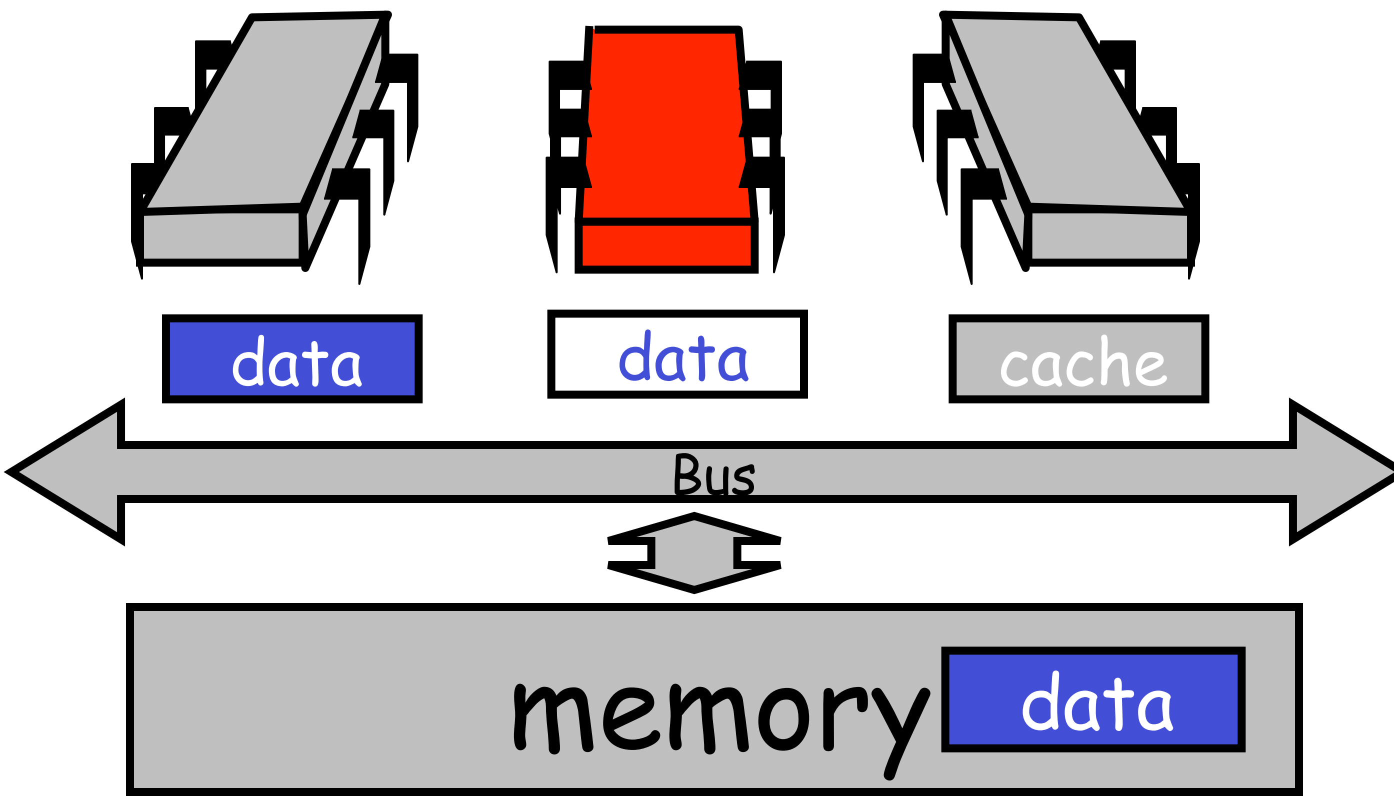
Modify Cached Data



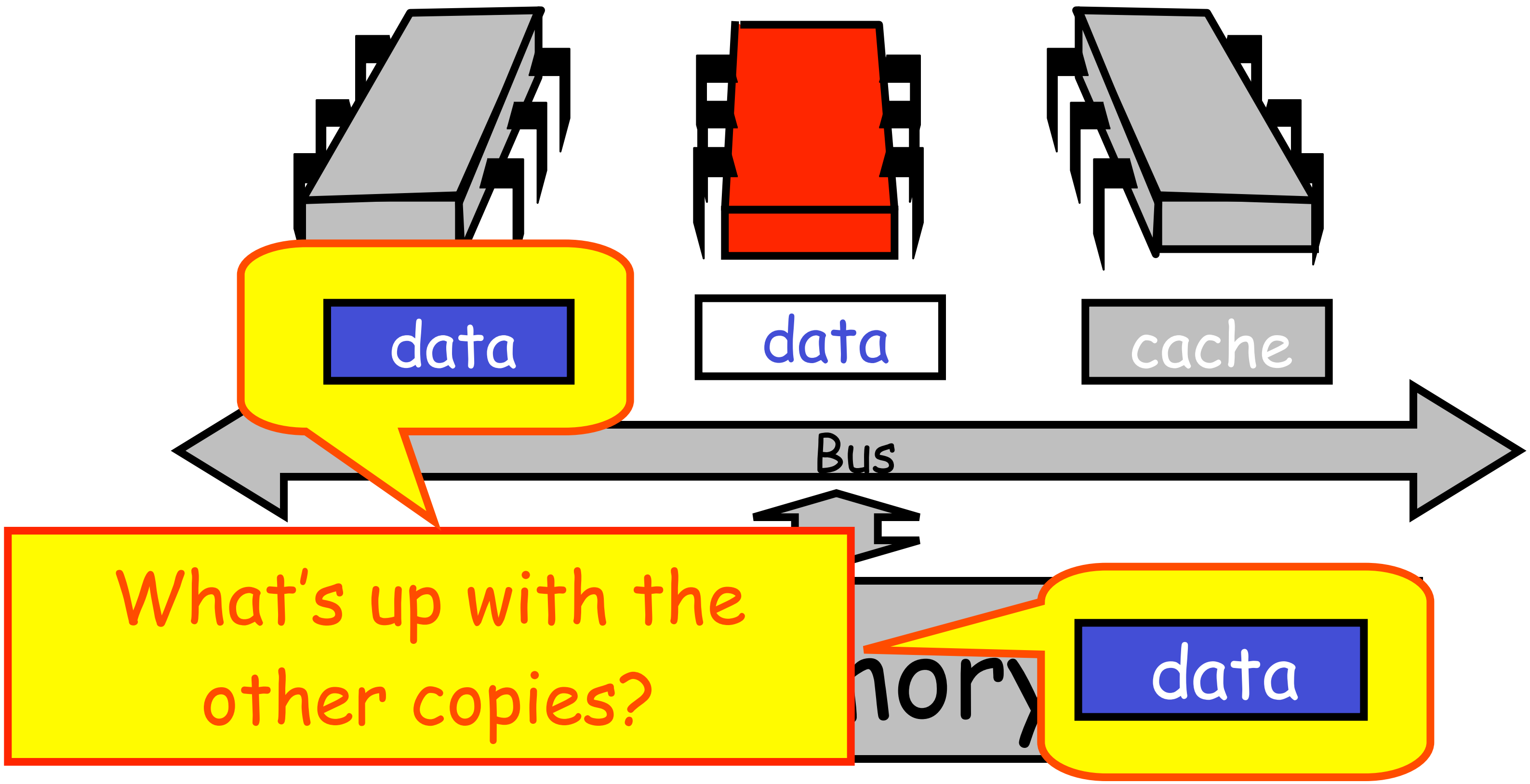
Modify Cached Data



Modify Cached Data



Modify Cached Data



Cache Coherence

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

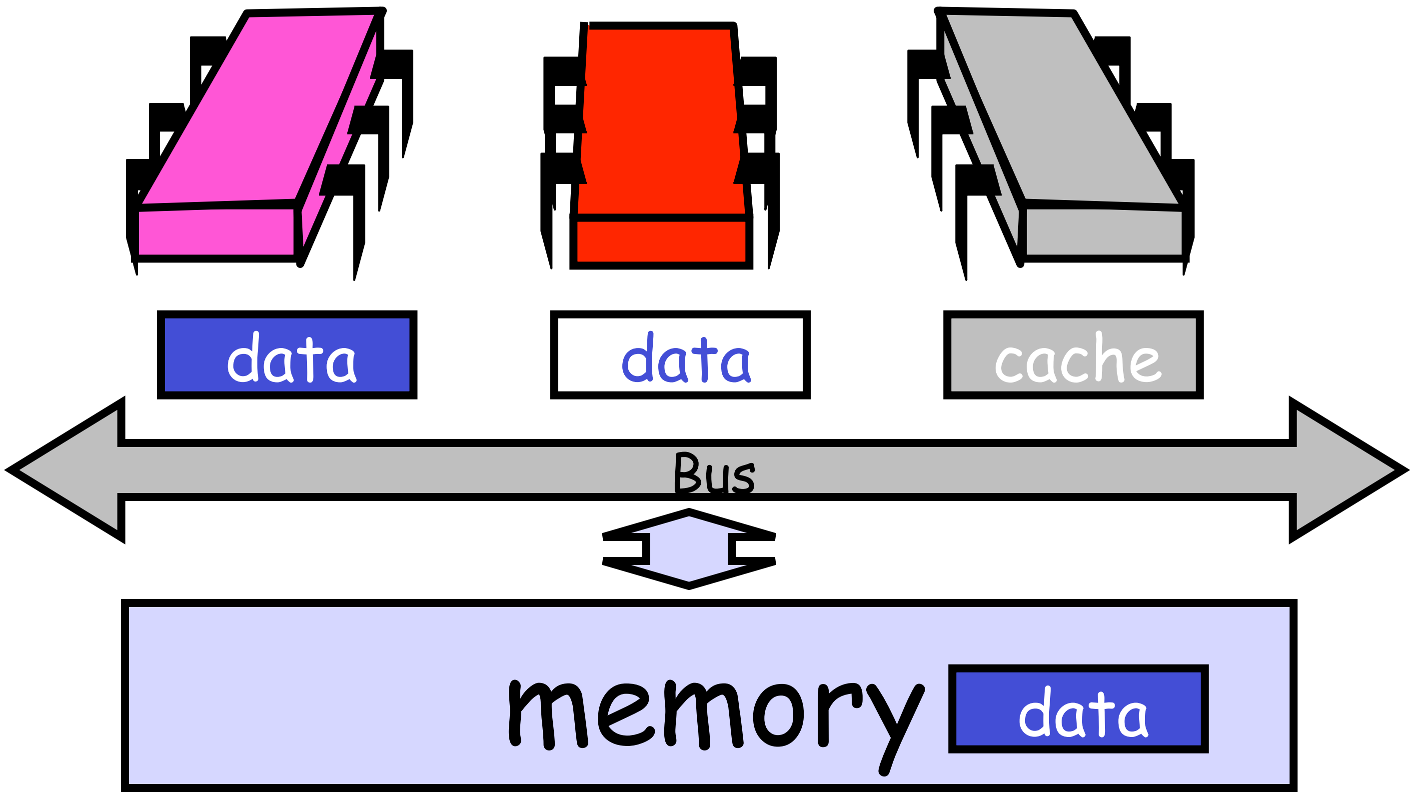
Write-Back Caches

- Accumulate changes in cache
- Write back when needed
  - Need the cache for something else
  - Another processor wants it
- On first modification
  - Invalidate other entries
  - Requires non-trivial protocol …

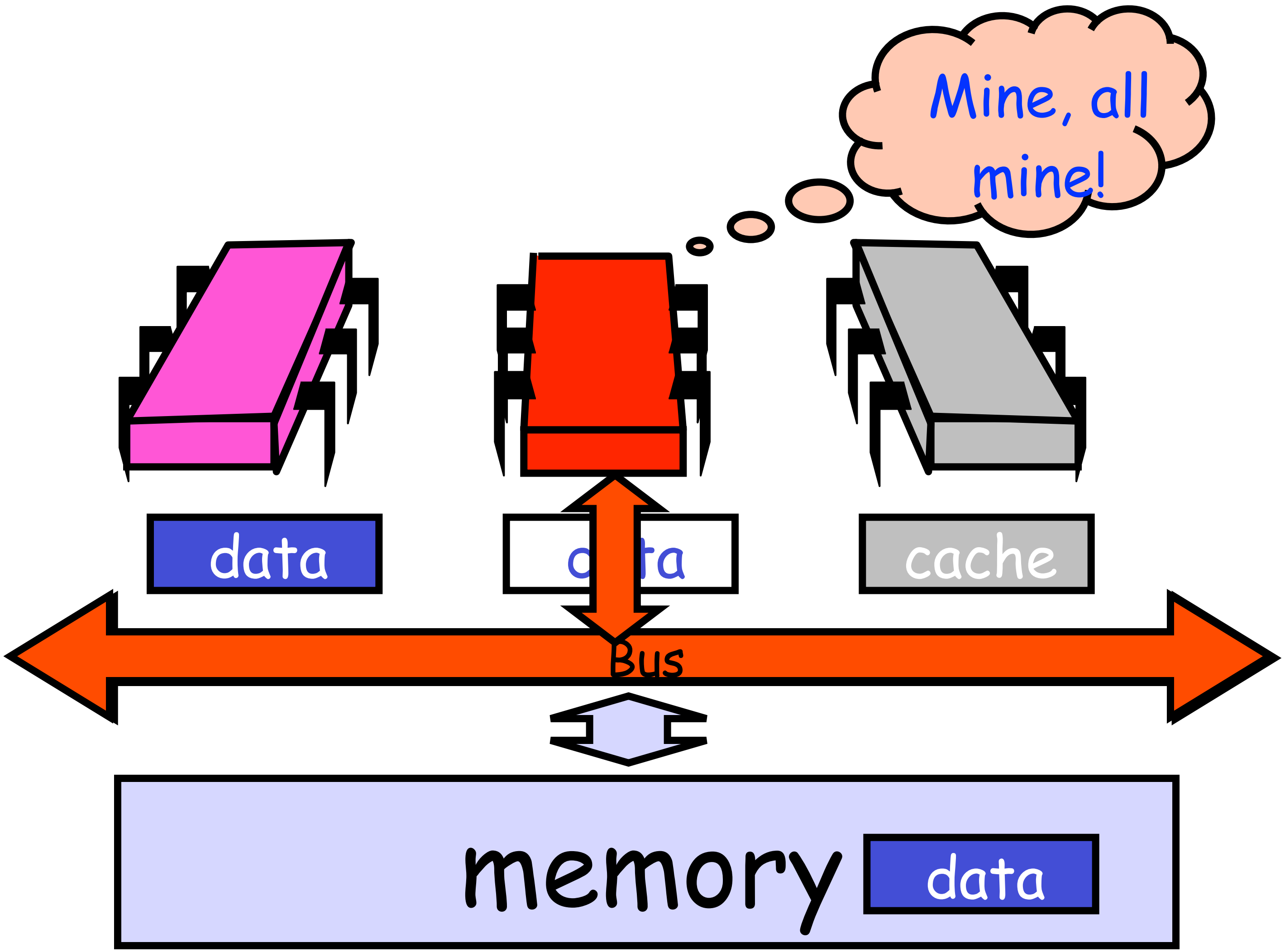
Write-Back Caches

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

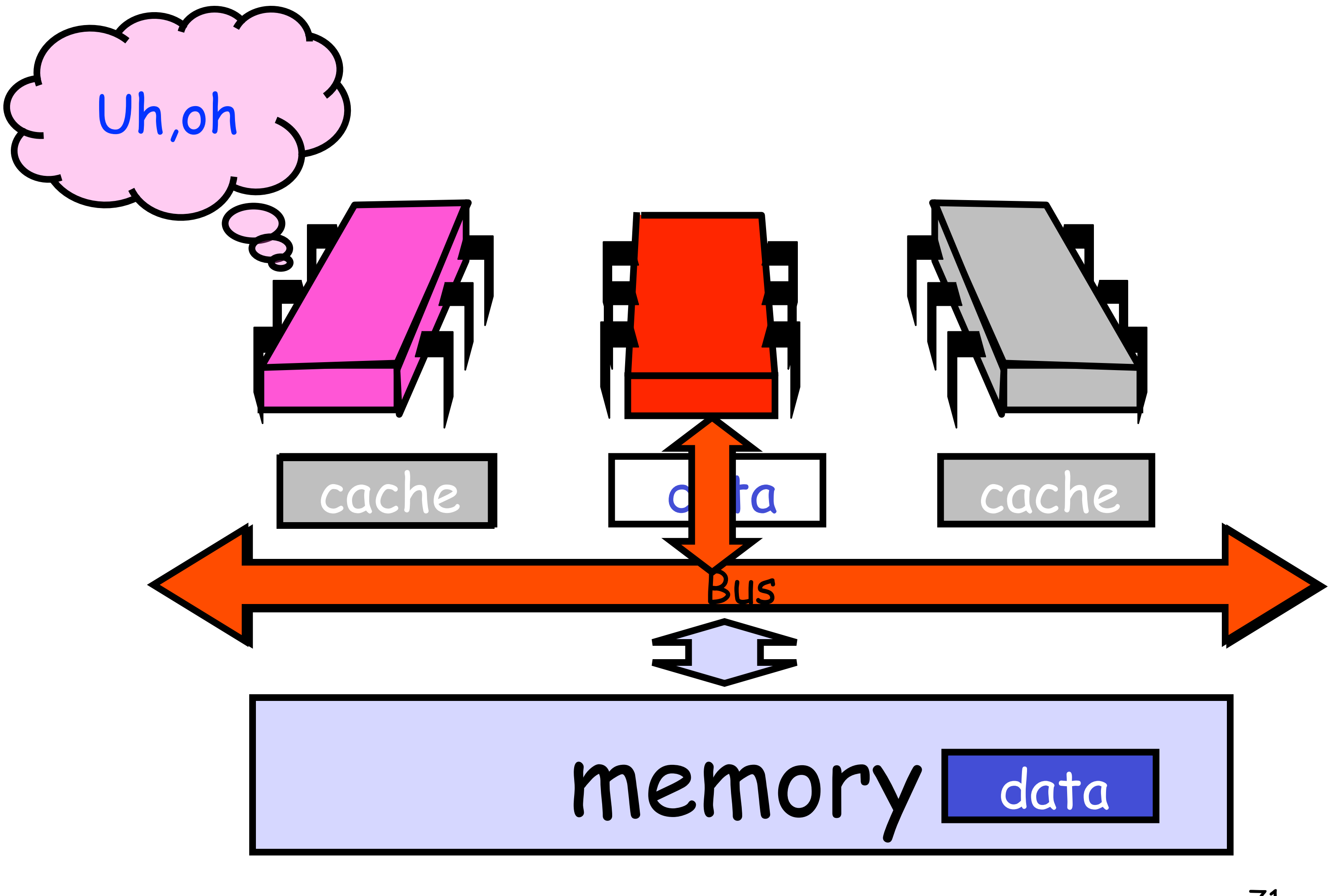
Invalidate



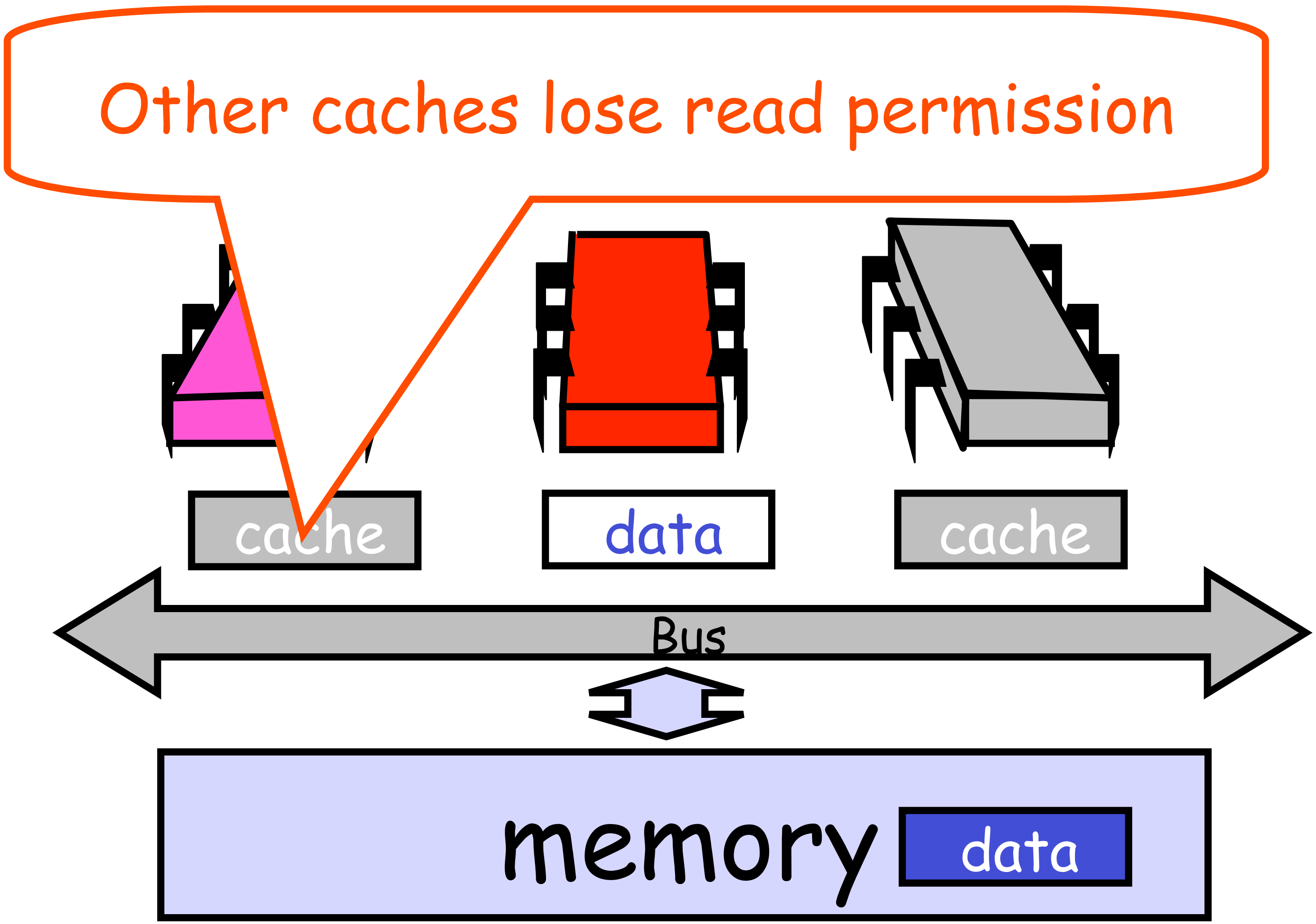
Invalidate



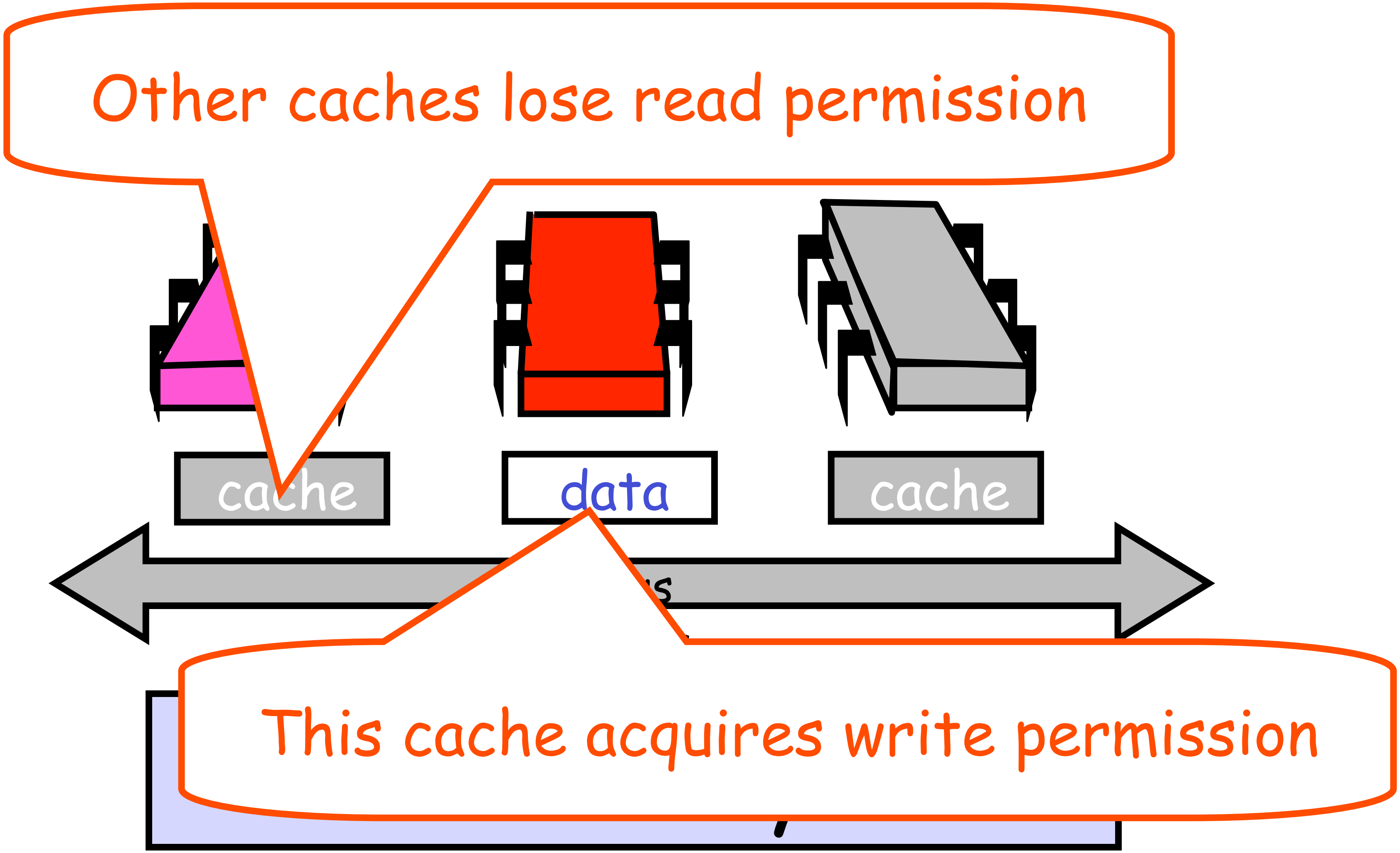
Invalidate



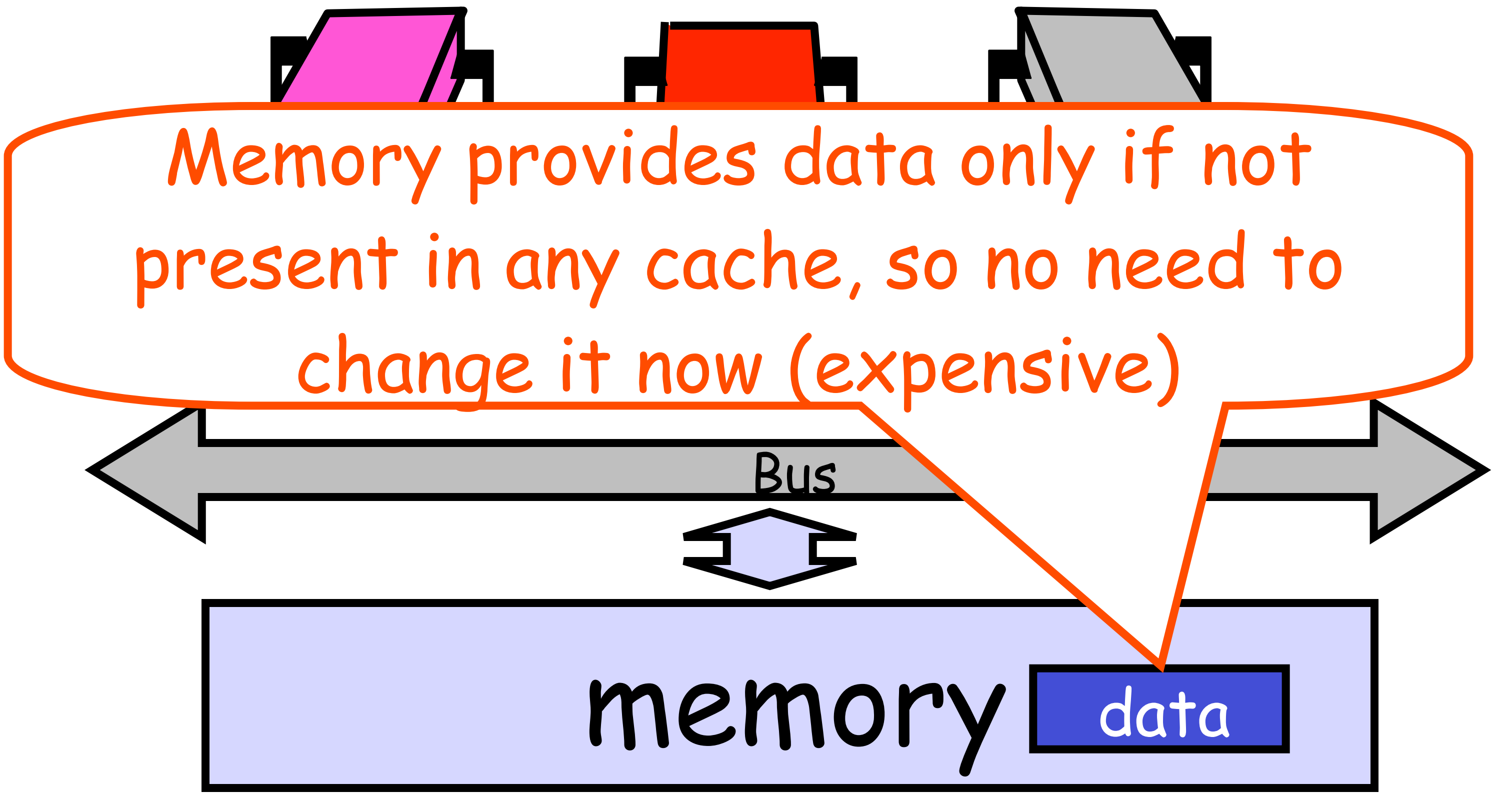
Invalidate


Other caches lose read permission
cache
data
cache
Bus
memory
data

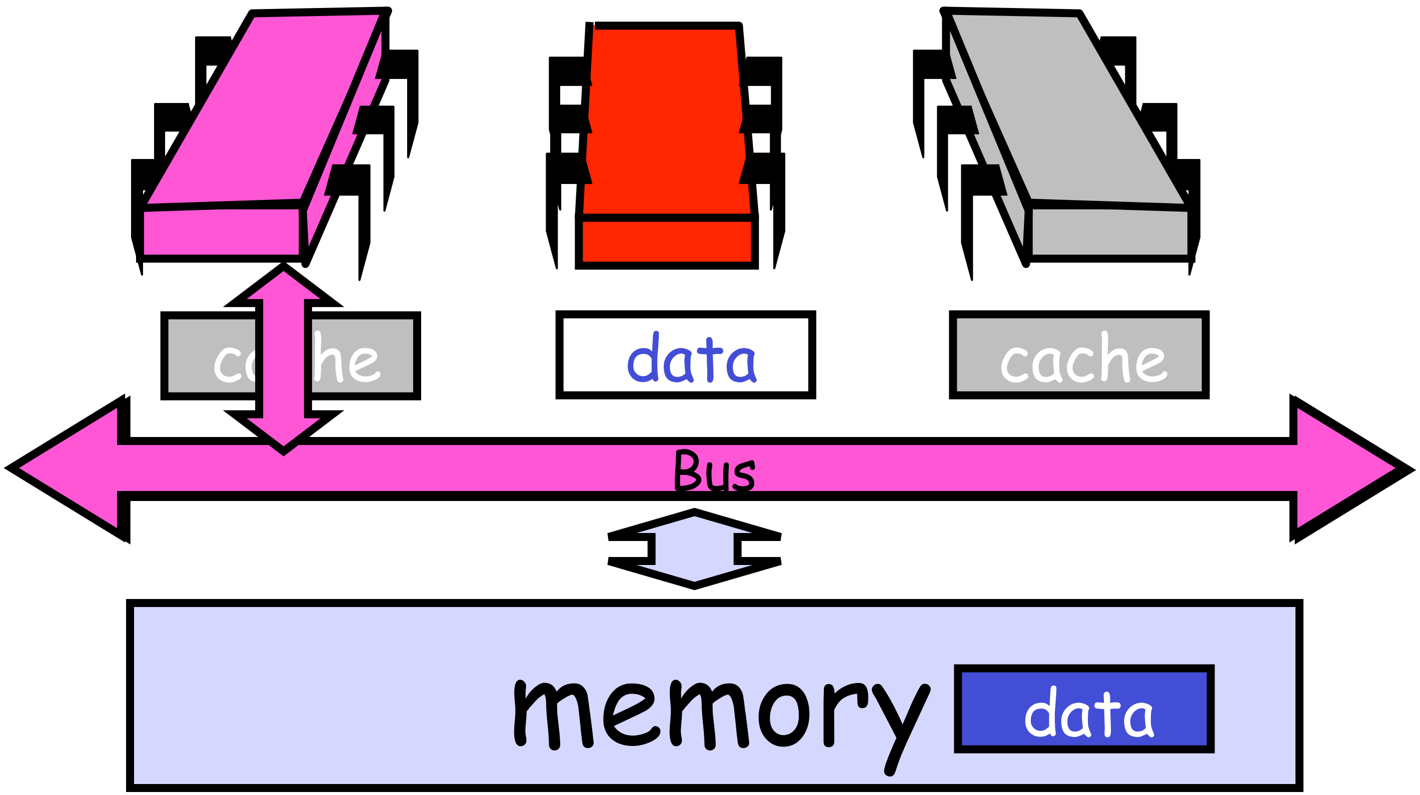
Invalidate


Other caches lose read permission
cache
data
cache
This cache acquires write permission

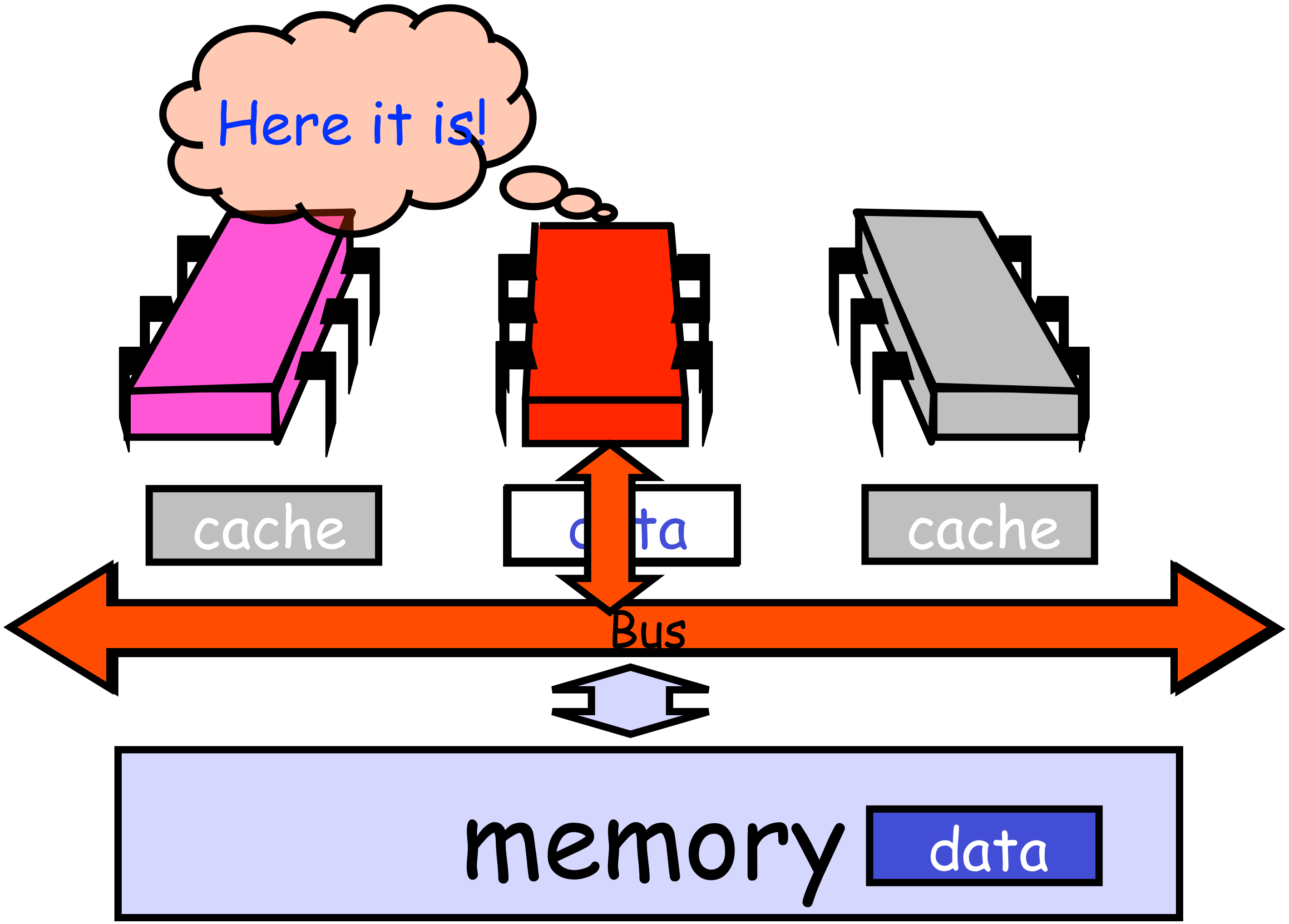
Invalidate


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

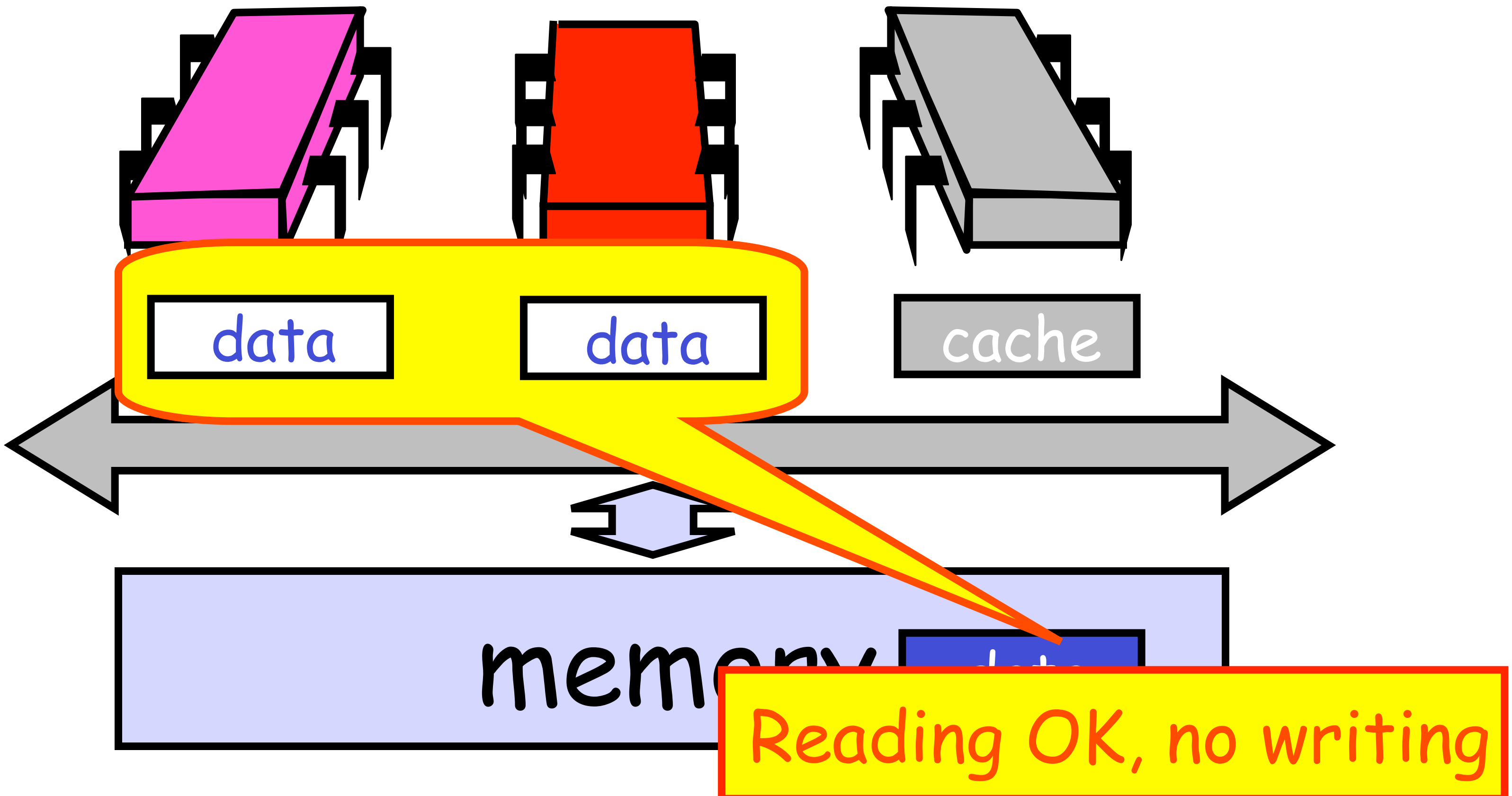
Another Processor Asks for



Owner Responds



End of the Day …



Mutual Exclusion

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

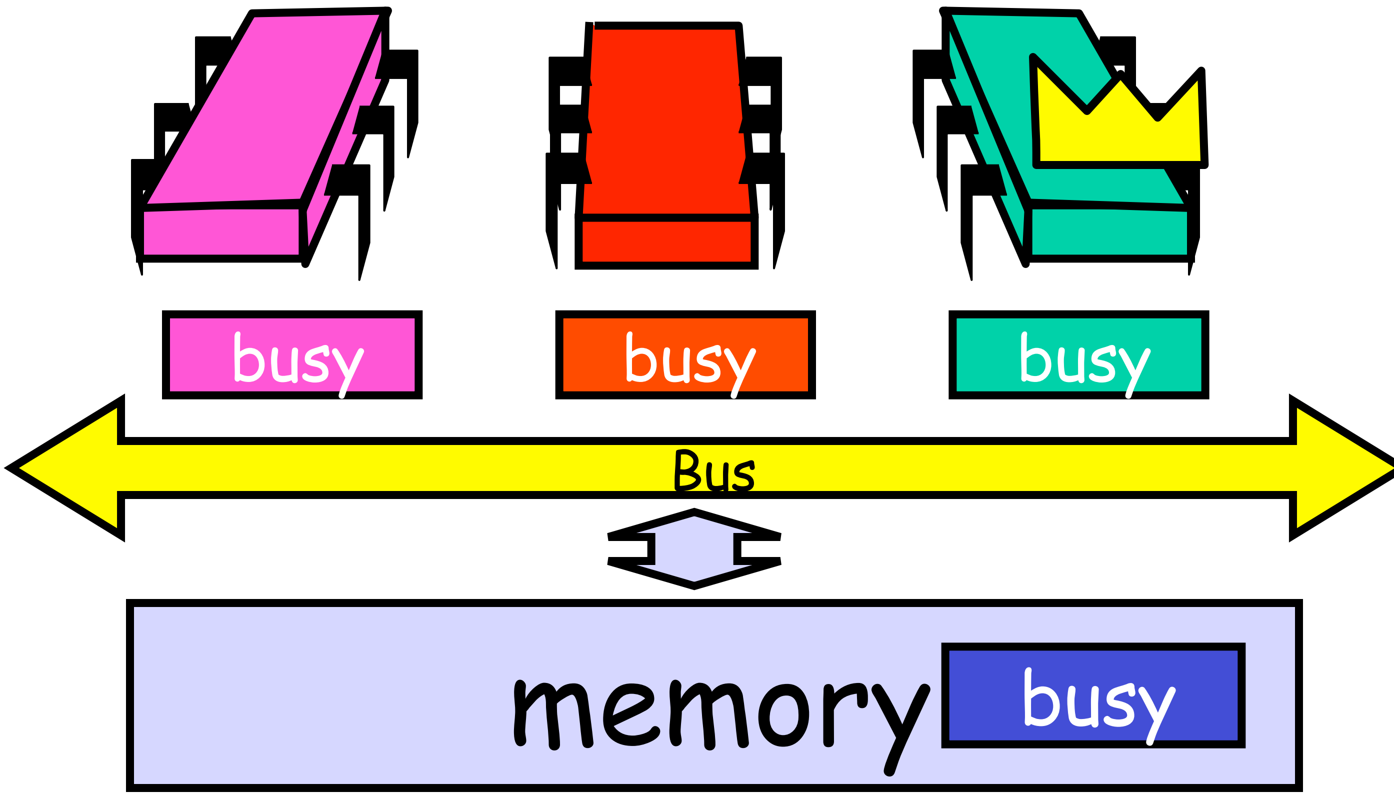
Simple TASLock

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

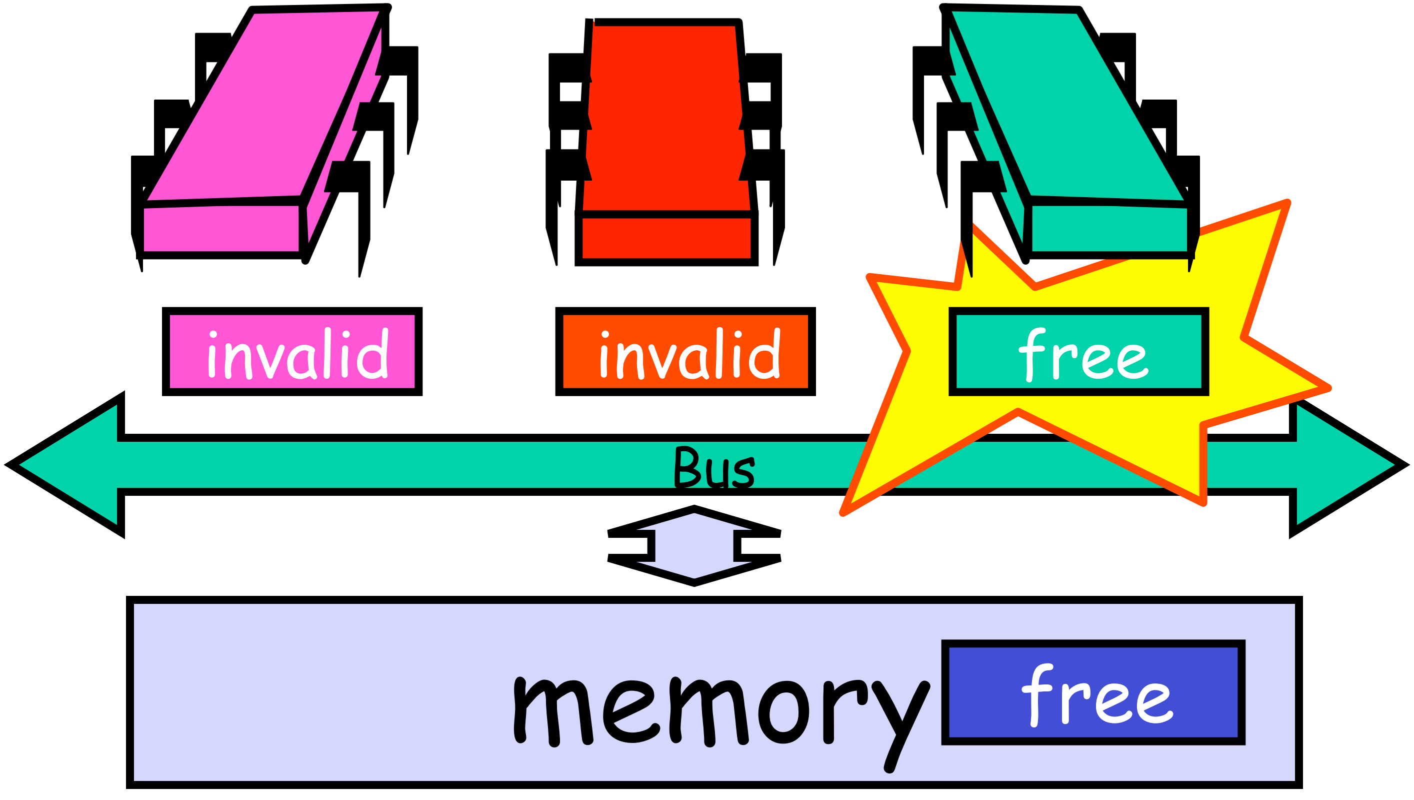
Test-and-test-and-set

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

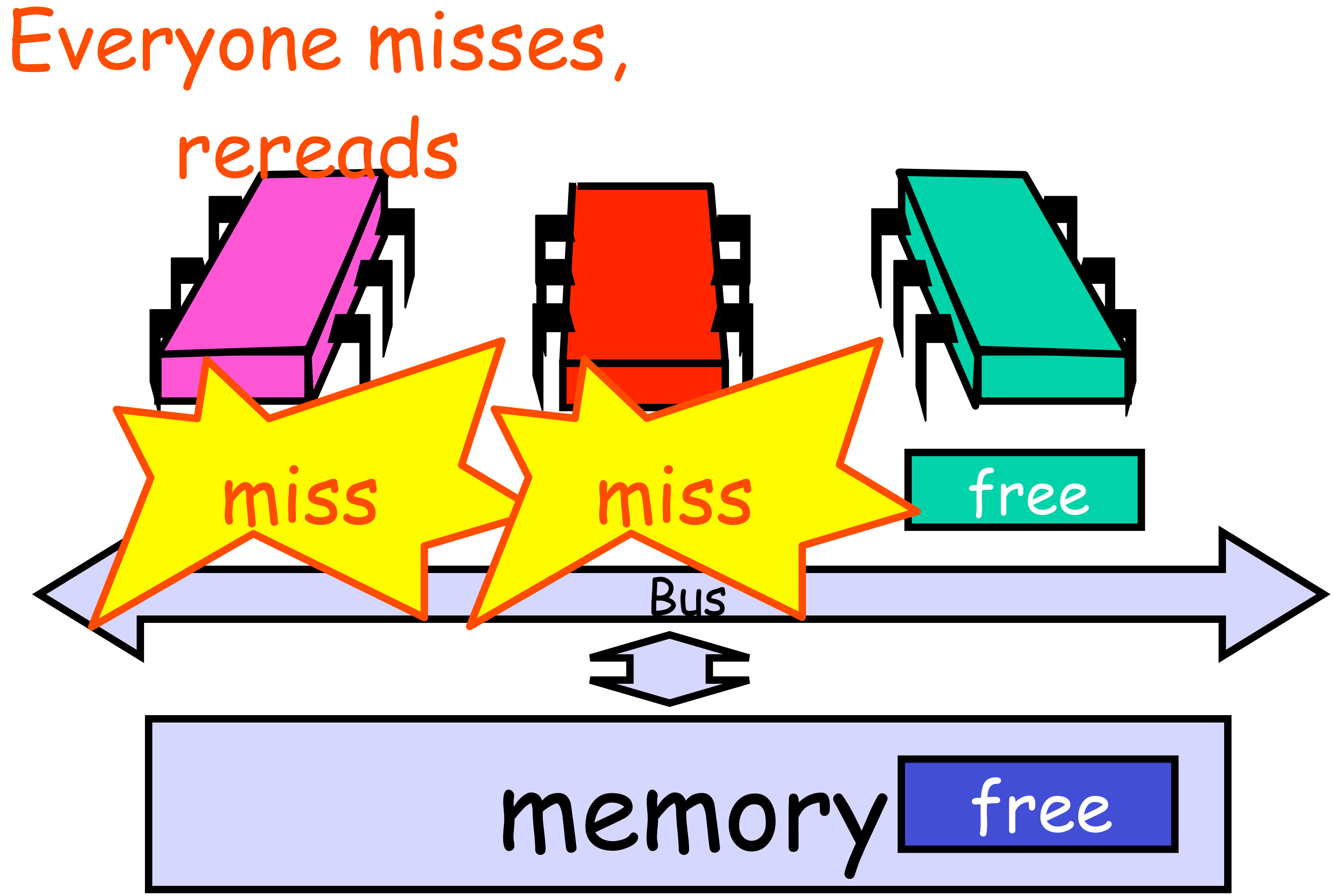
Local Spinning while Lock is



On Release

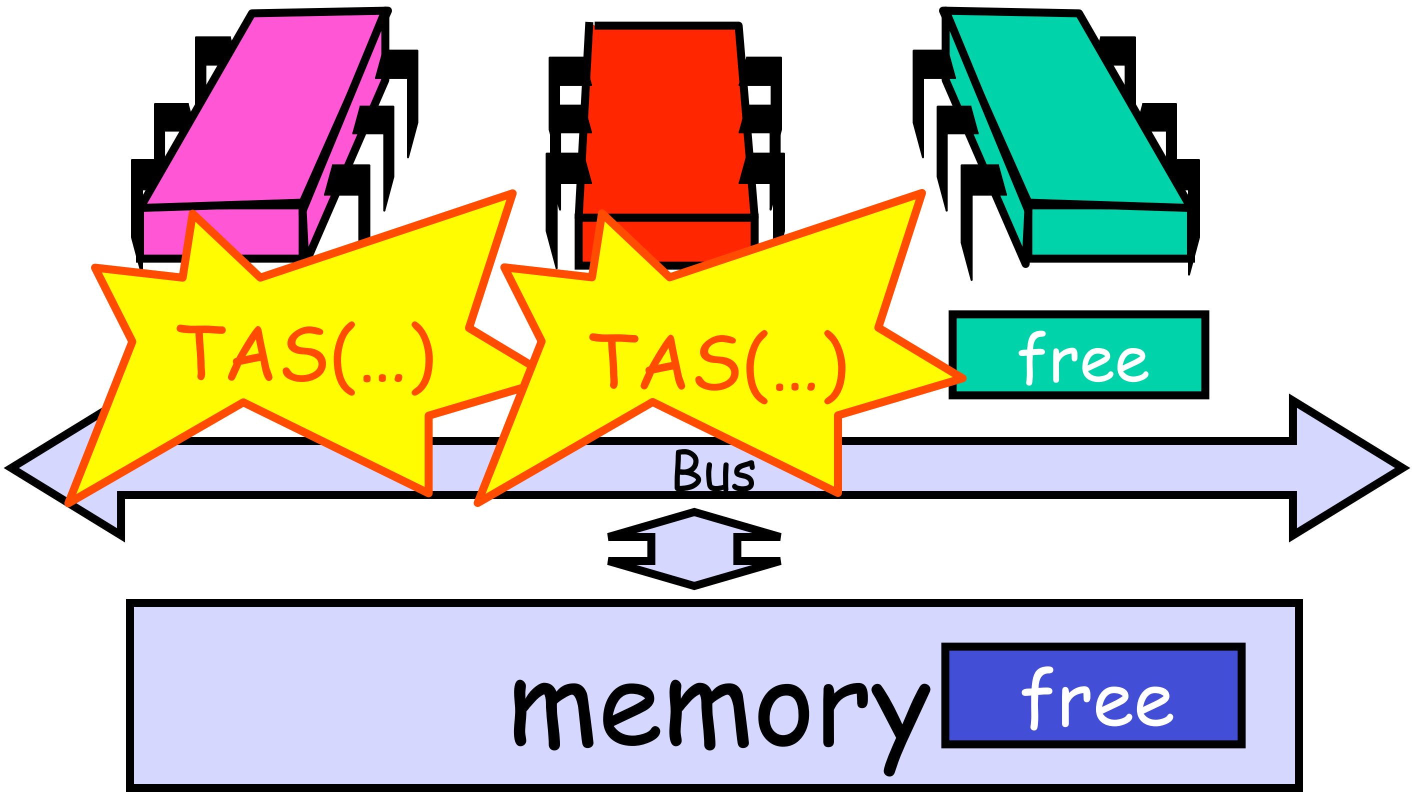


On Release



On Release

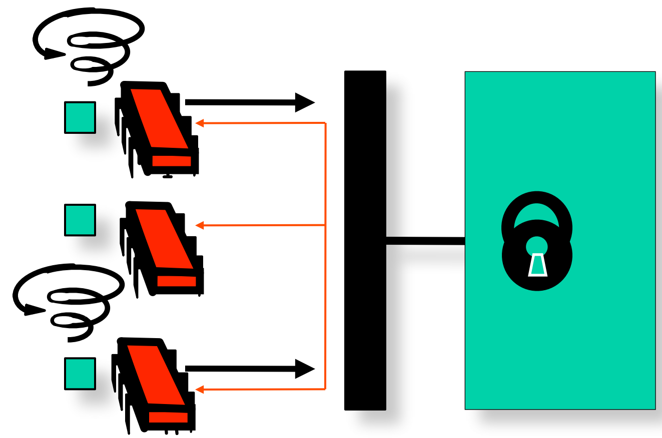
Everyone tries TAS



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



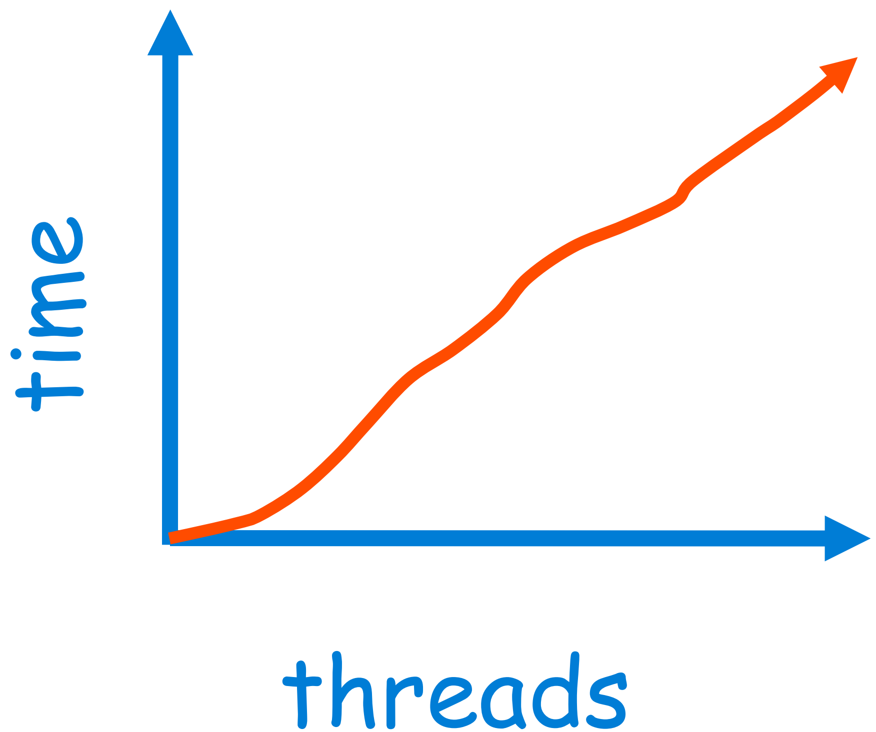
X = time of ops that don't use the bus

Y = time of ops that cause intensive bus traffic

In critical section, run ops X then ops Y. As long as Quiescence time is less than X, no drop in performance.

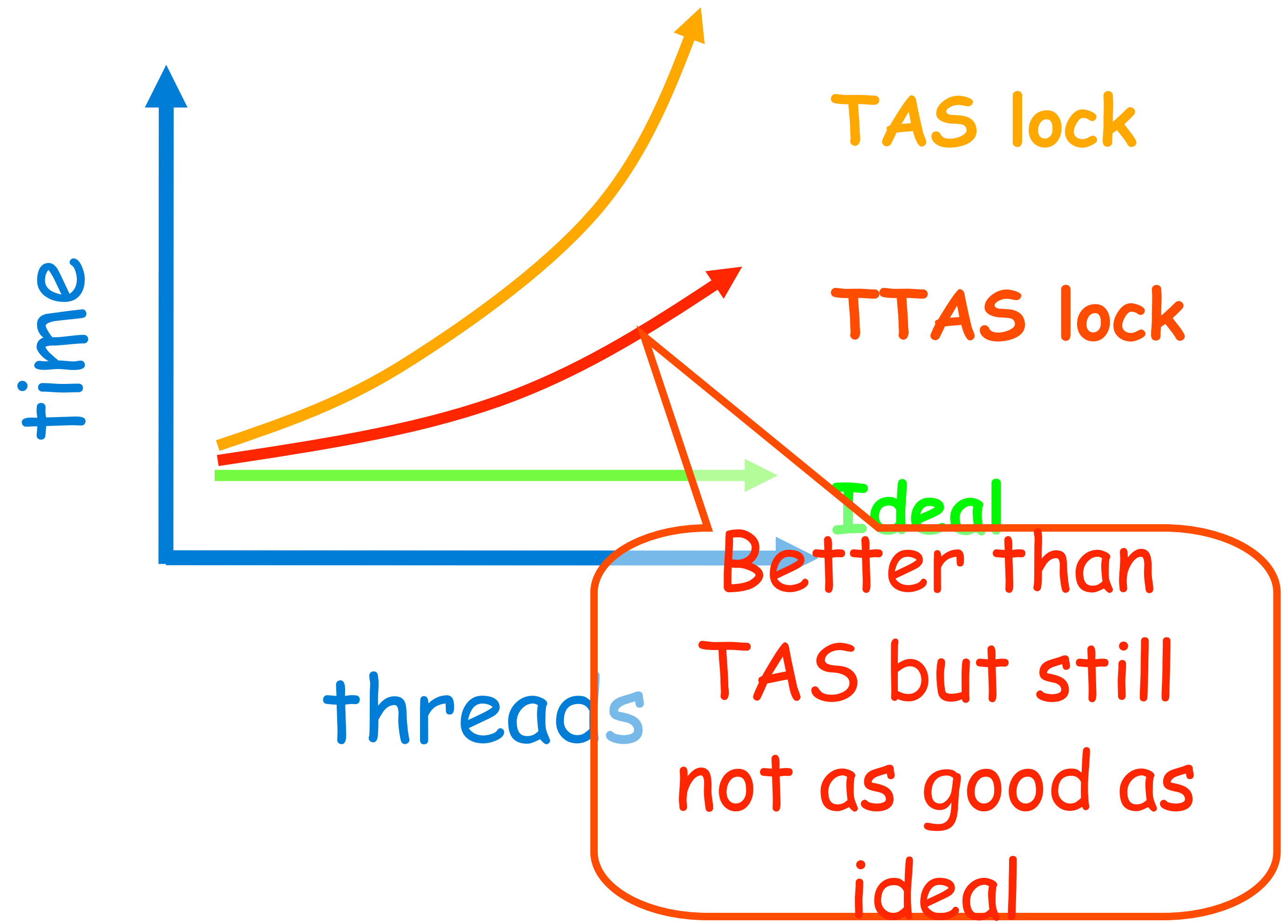
By gradually varying X, can determine the exact time to quiesce.

Quiescence Time



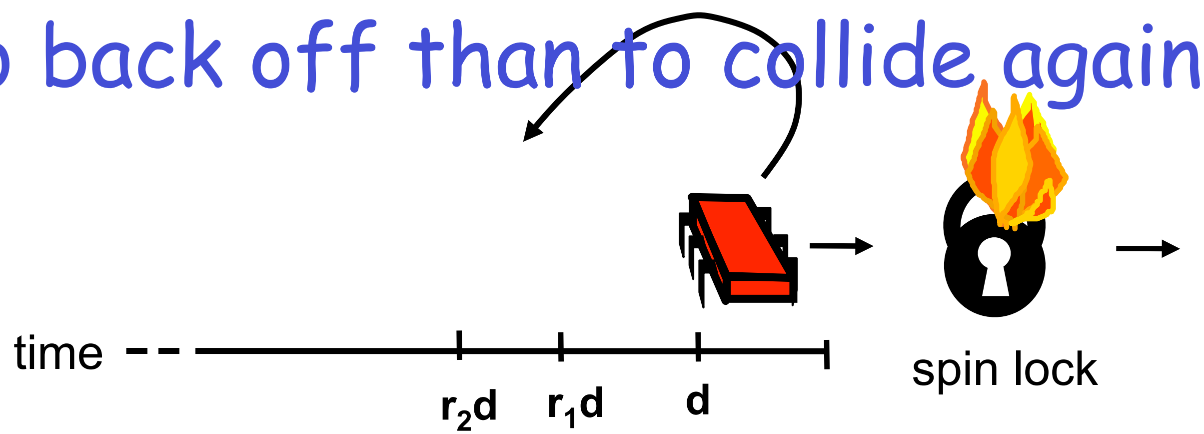
Increses linearly with the number of processors for bus architecture

Mystery Explained

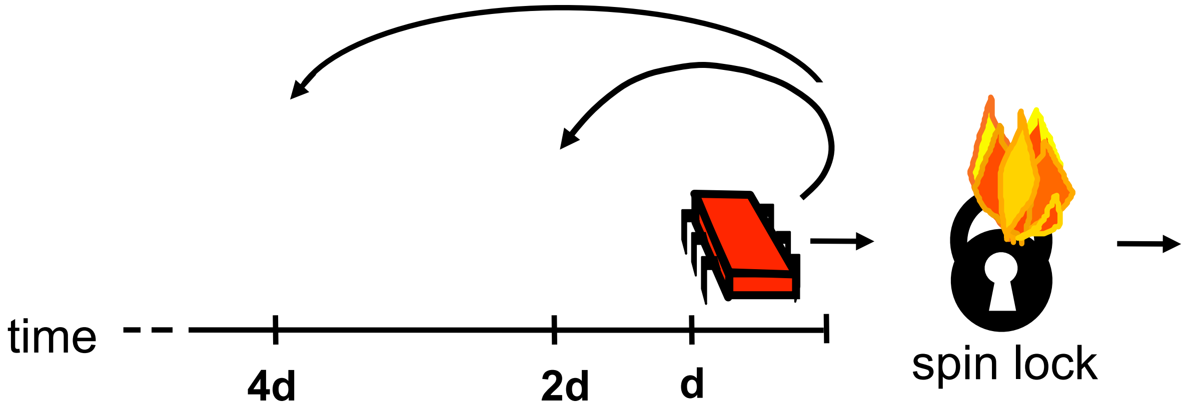


Solution: Introduce Delay

- If the lock looks free
  - But I fail to get it
- There must be lots of contention
  - Better to back off than to collide again



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

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

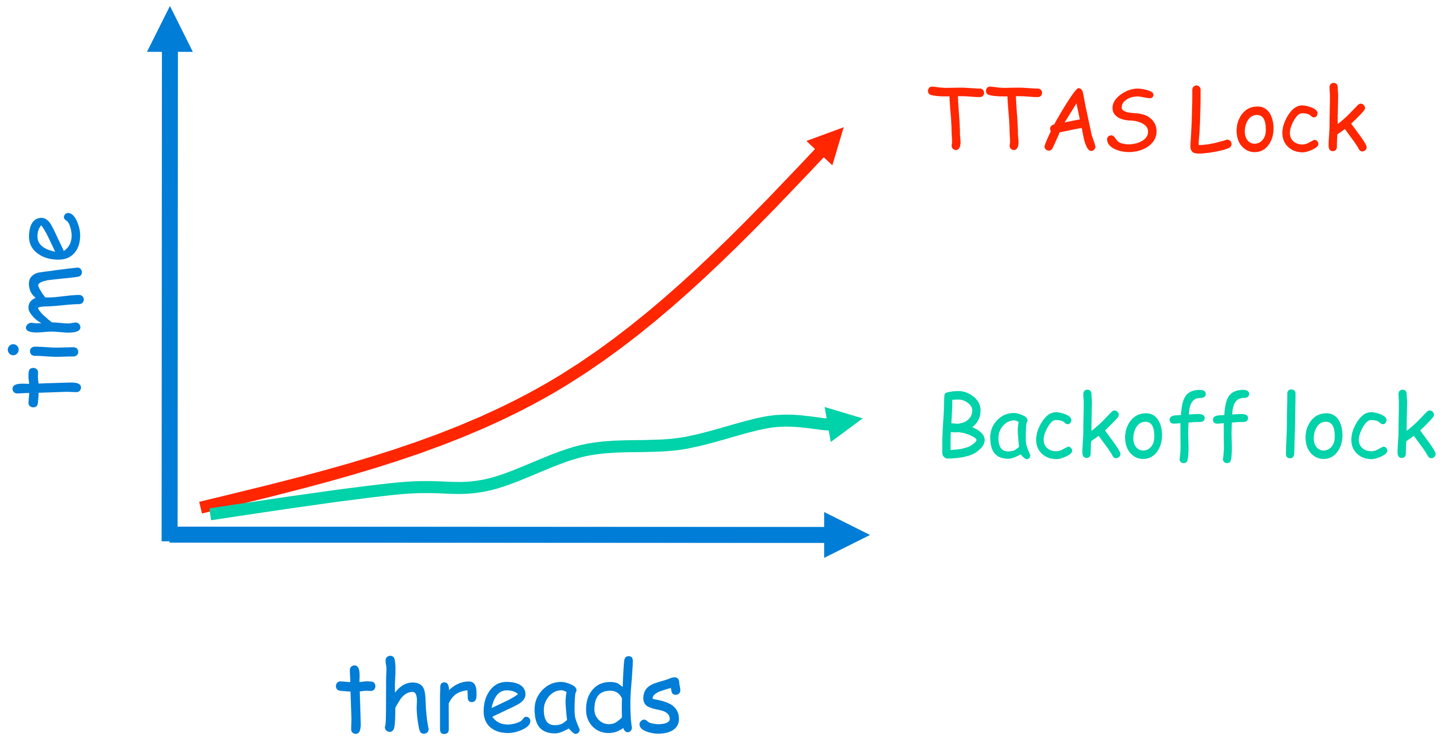
Back off for random duration

Exponential Backoff Lock

Double max delay, within reason

```
publi
 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;
 }}}
```

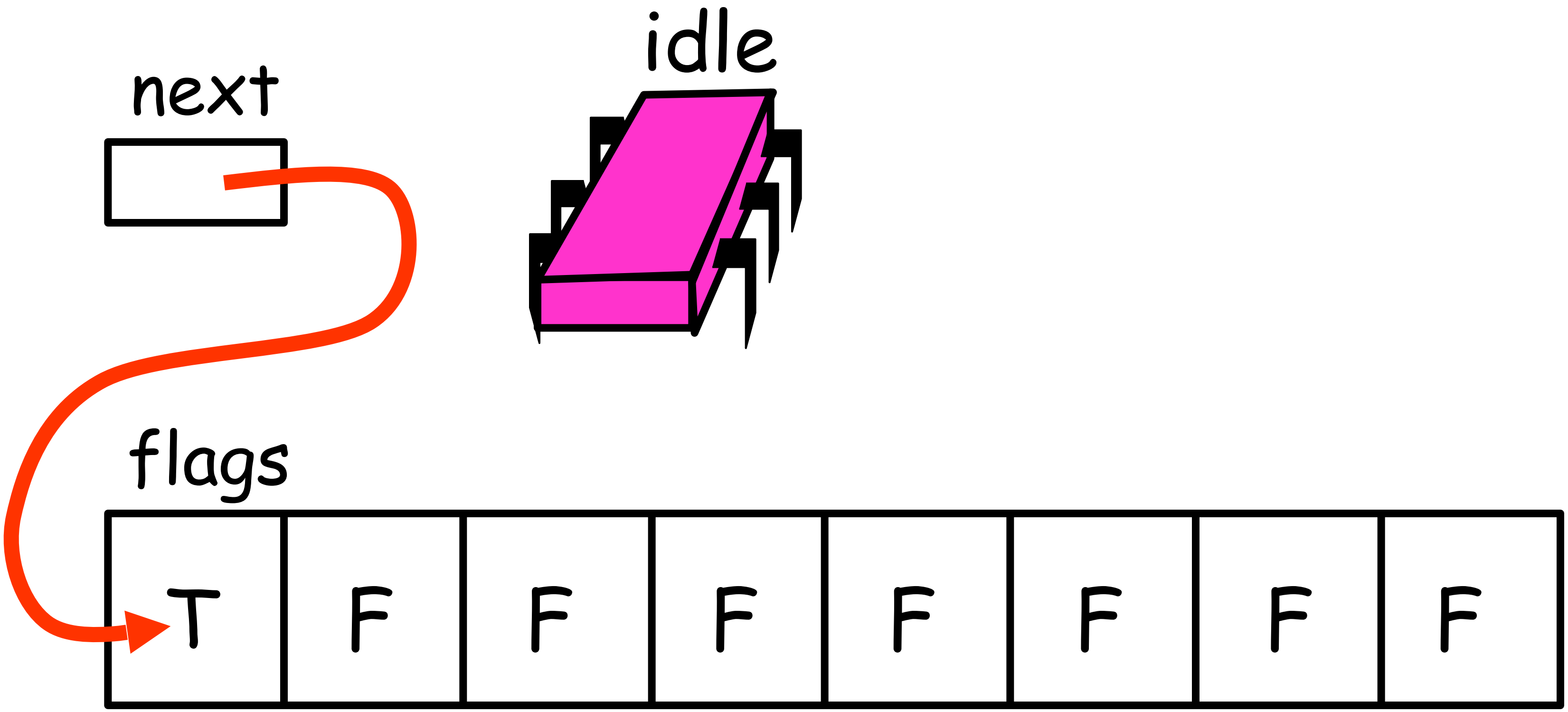
Spin-Waiting Overhead



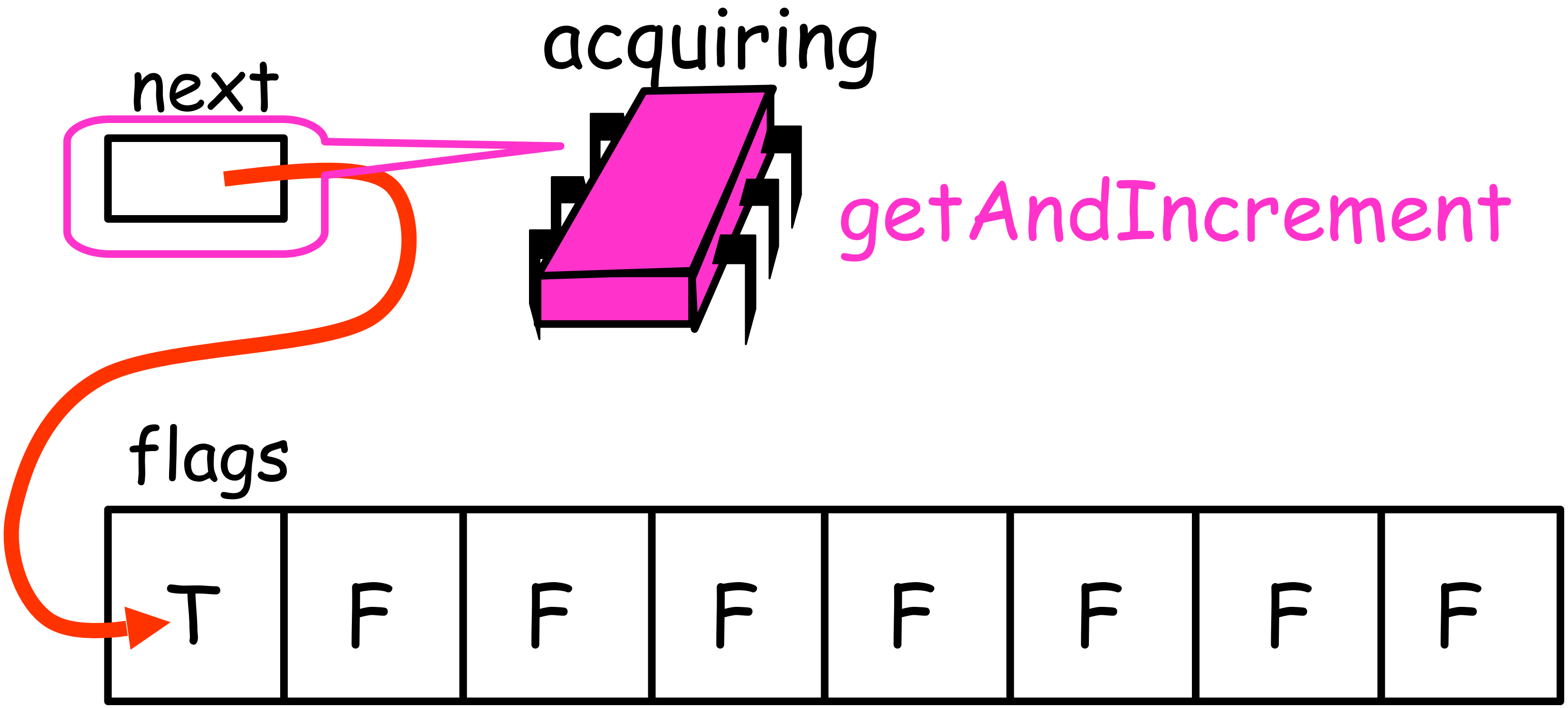
Idea

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

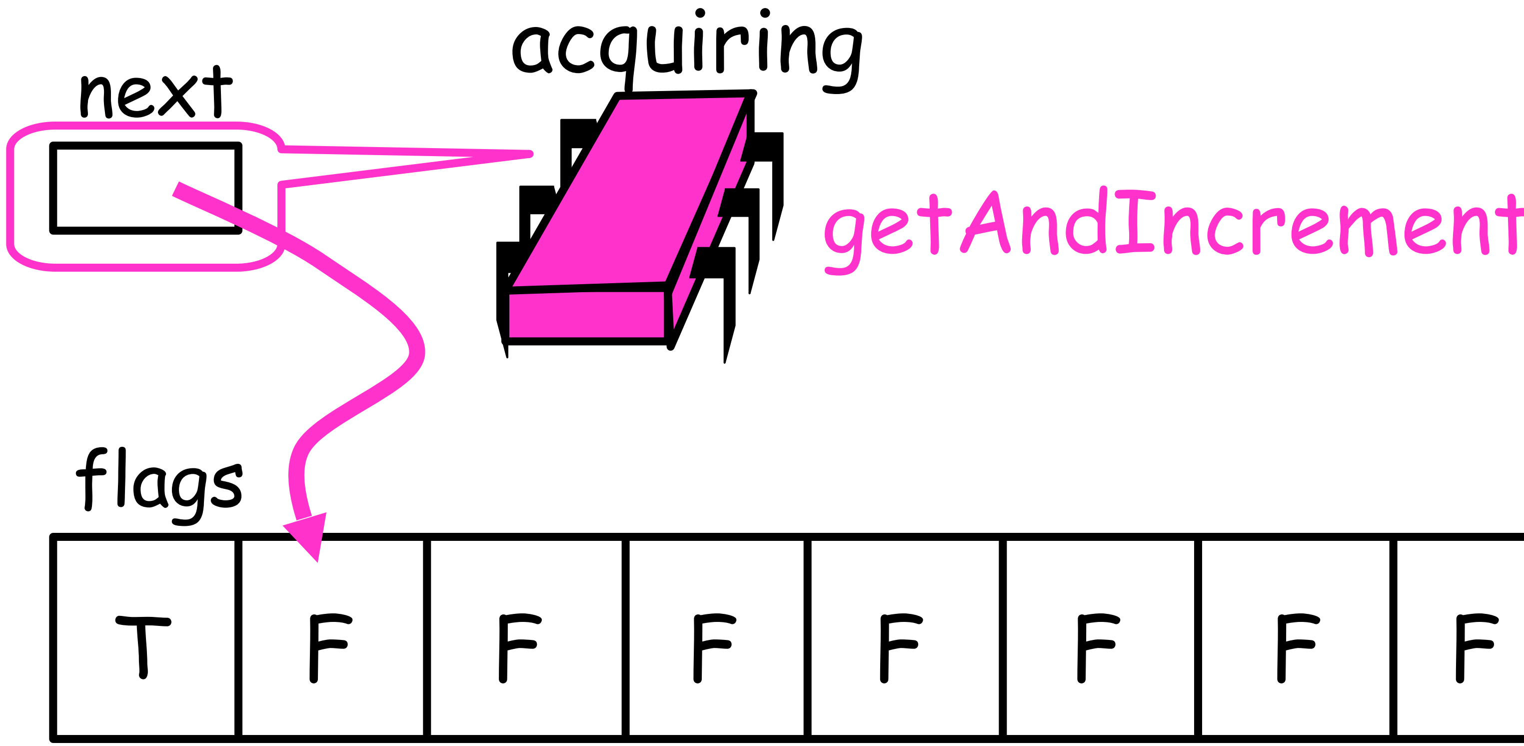
Anderson Queue Lock



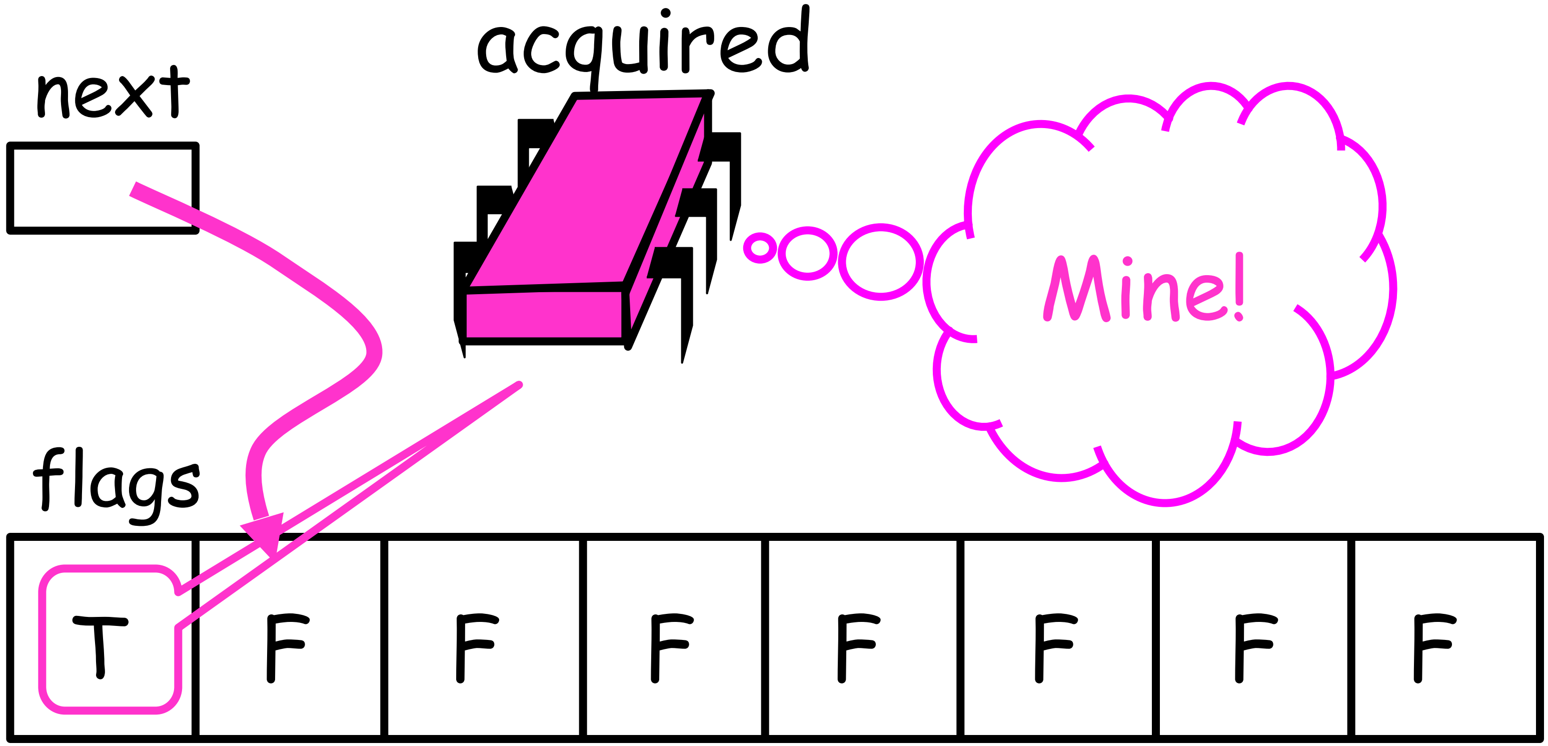
Anderson Queue Lock



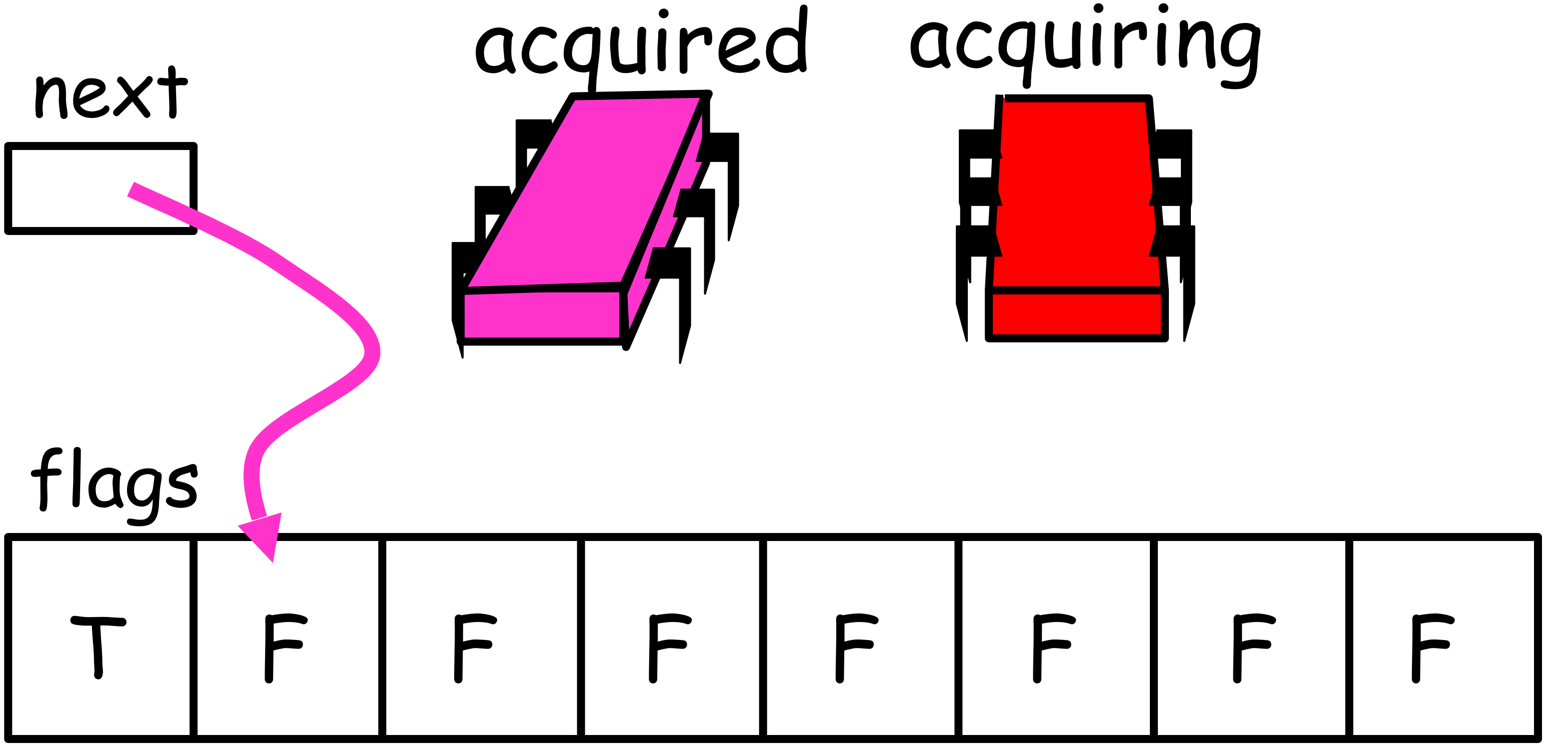
Anderson Queue Lock



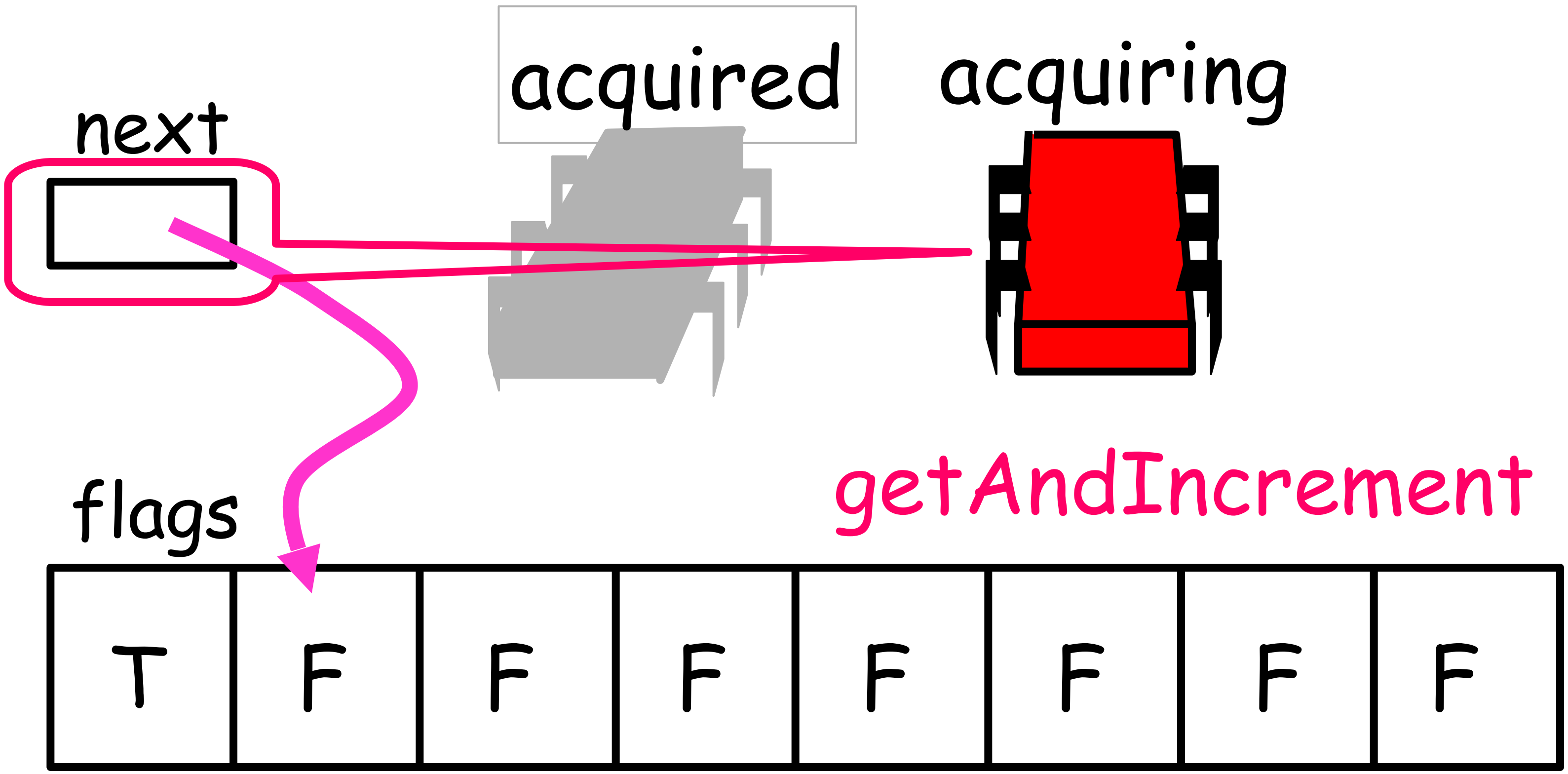
Anderson Queue Lock



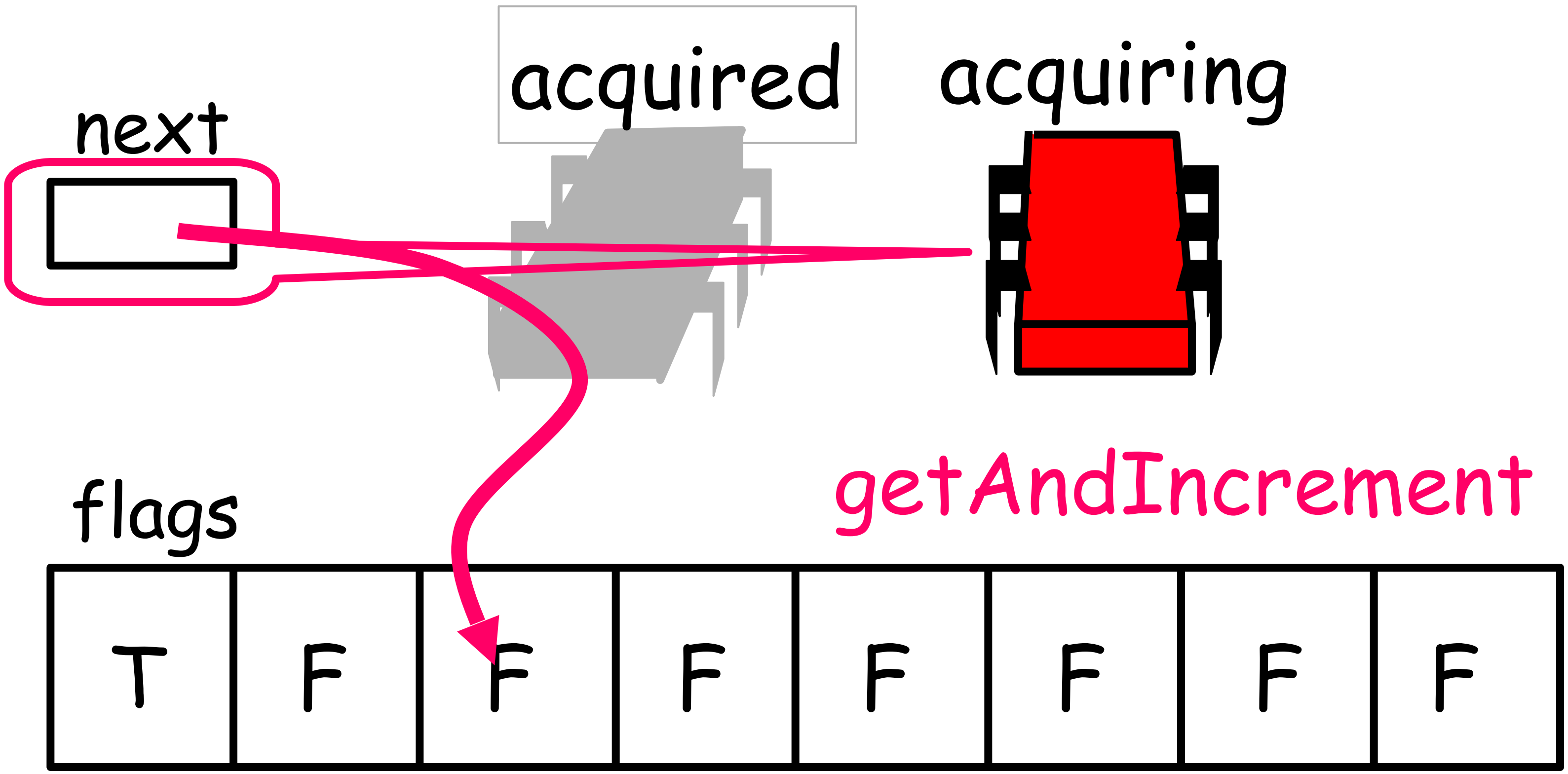
Anderson Queue Lock



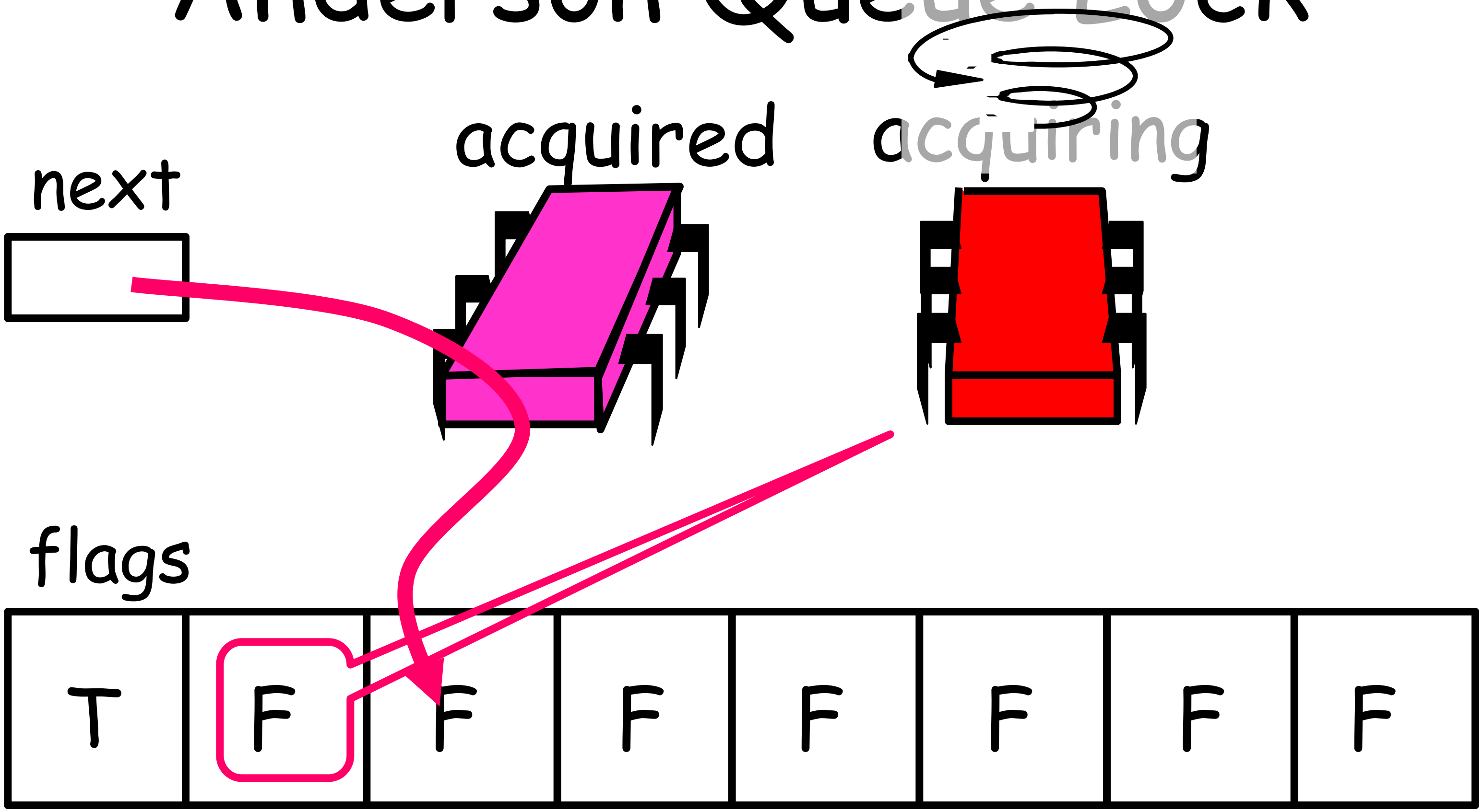
Anderson Queue Lock



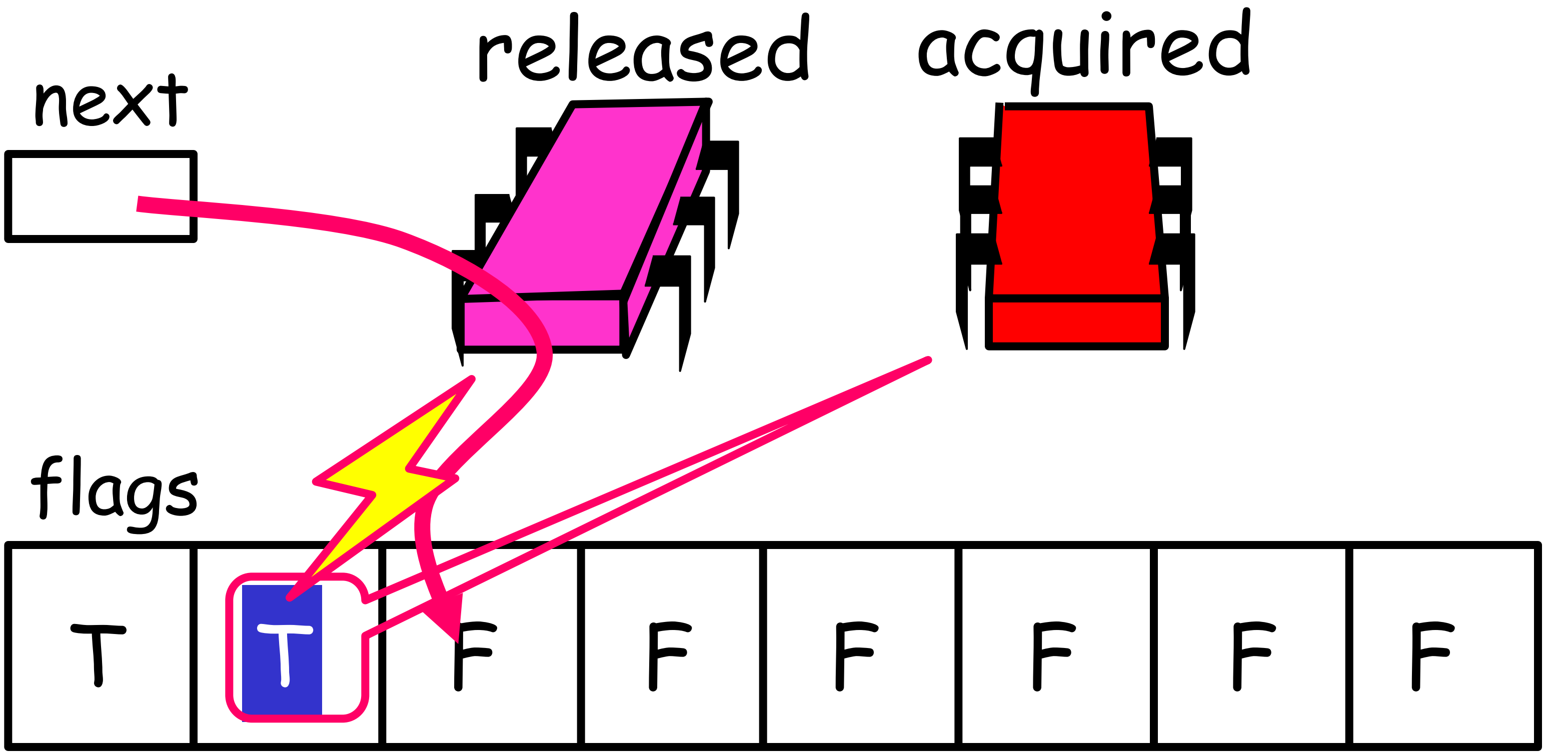
Anderson Queue Lock



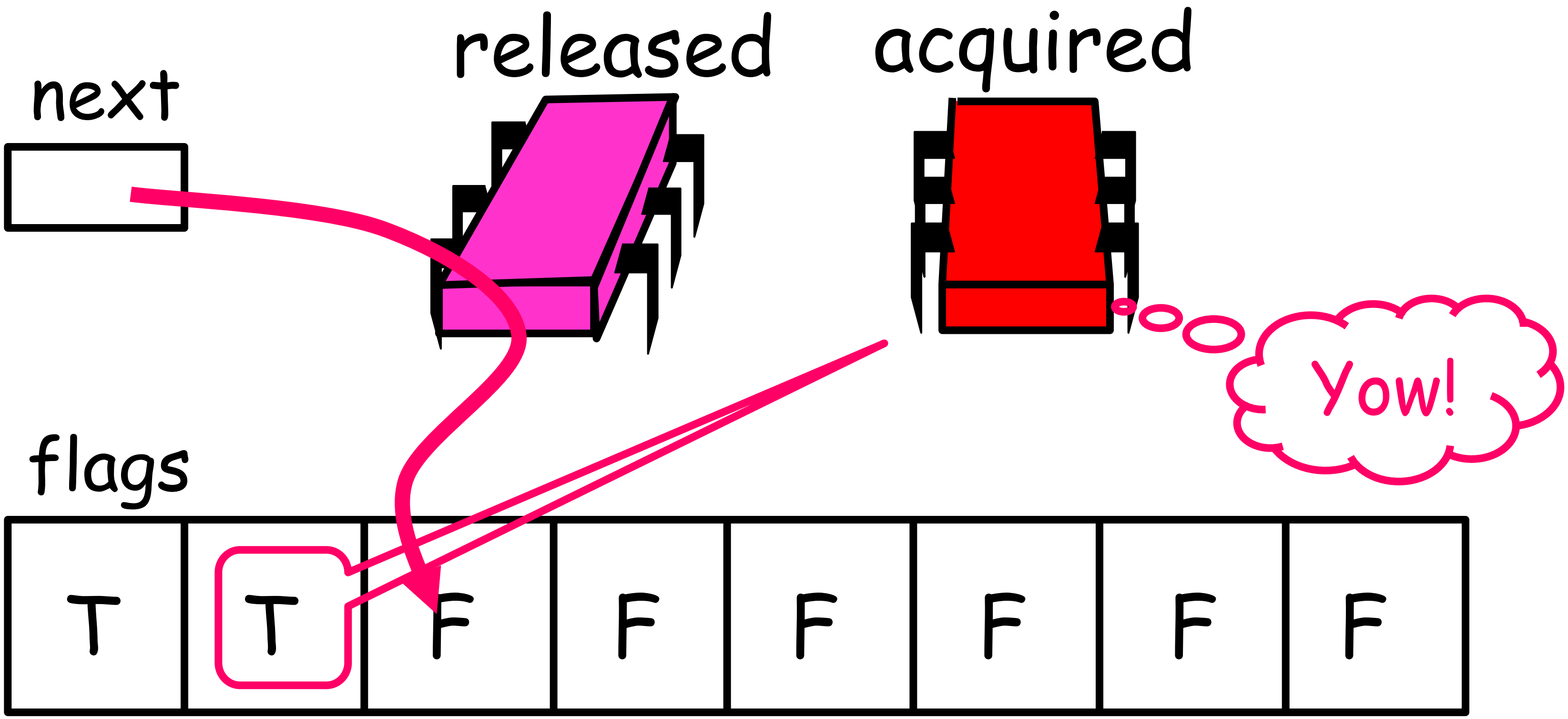
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```

Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```

One flag per thread

Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```

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] = true;
}
```

Take next slot

Anderson Queue Lock

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

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

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

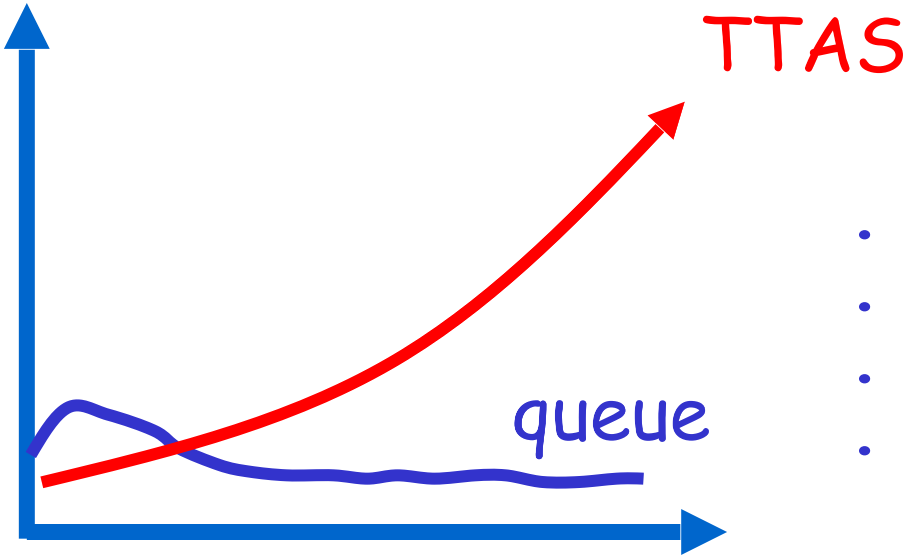
Anderson Queue Lock

Tell next thread to go

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

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

Performance



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

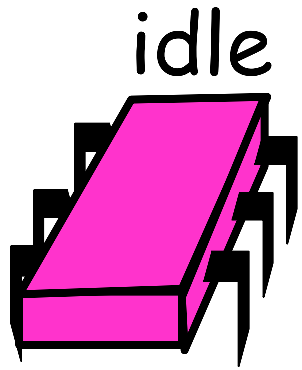
Anderson Queue Lock

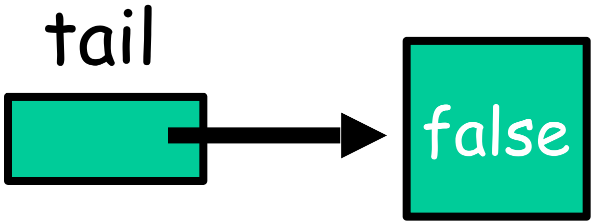
- Good
  - First truly scalable lock
  - Simple, easy to implement
- Bad
  - Space hog
  - One bit per thread
    - Unknown number of threads?
    - Small number of actual contenders?

CLH Lock

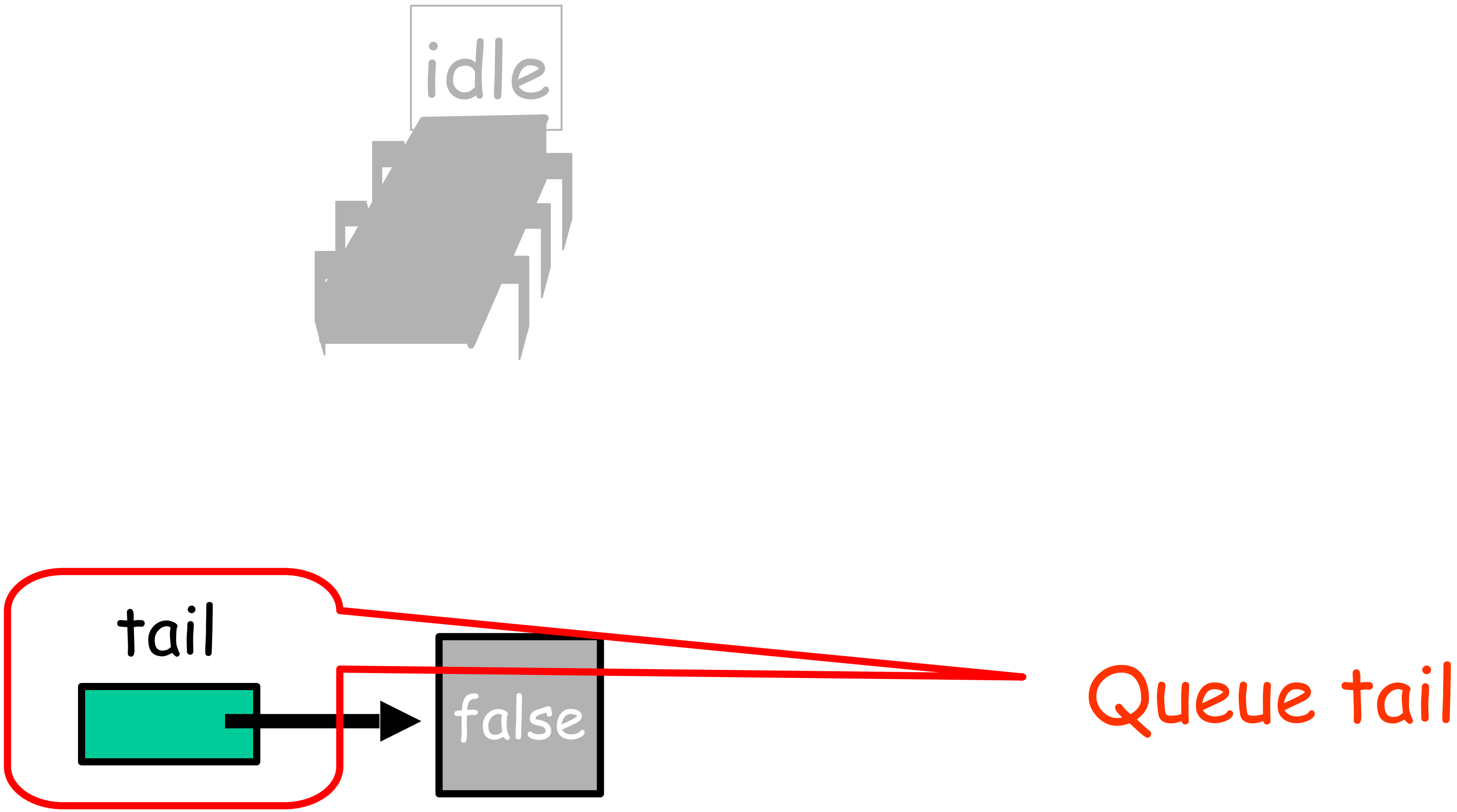
- FIFO order
- Small, constant-size overhead per thread

Initially

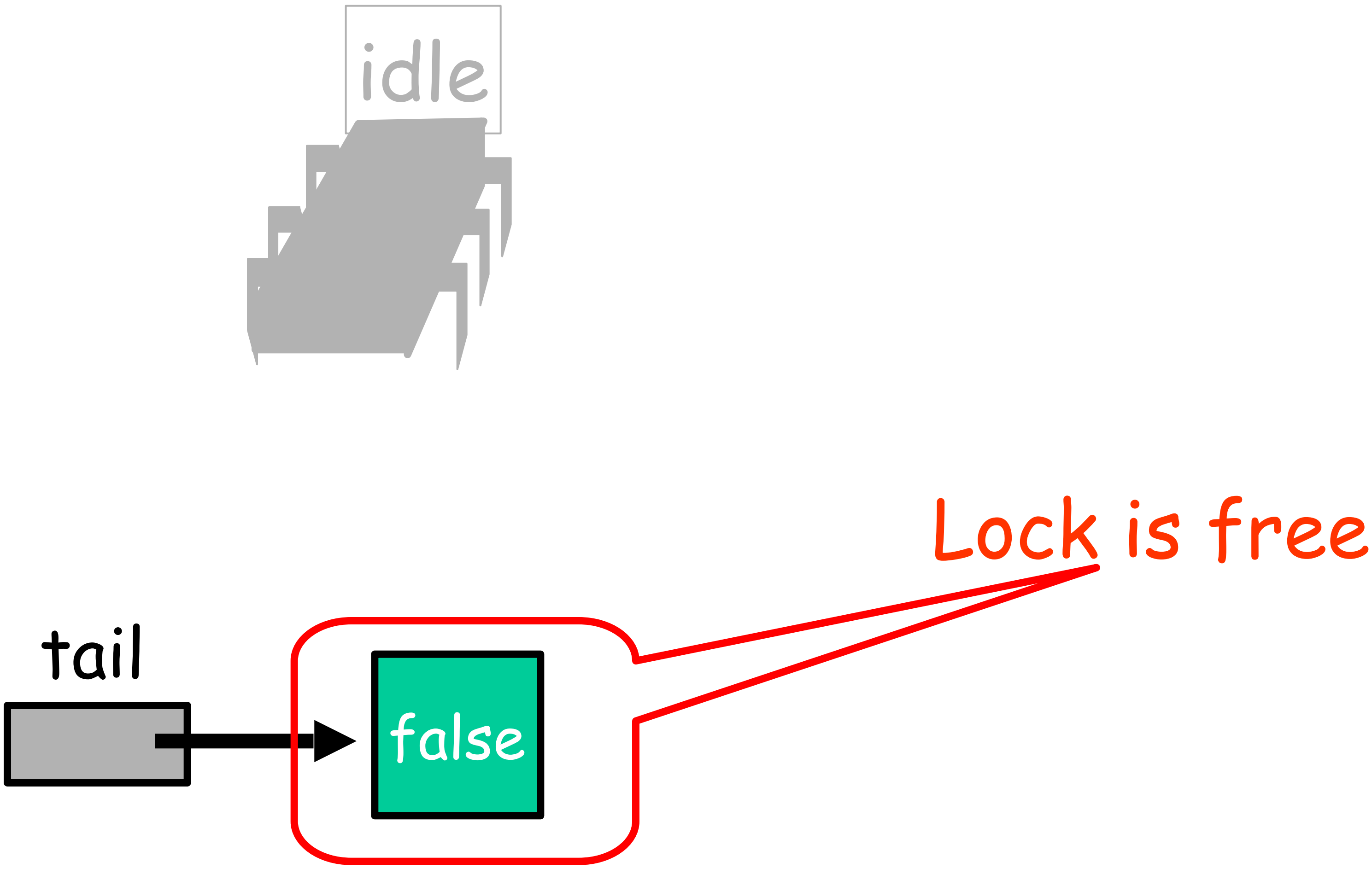




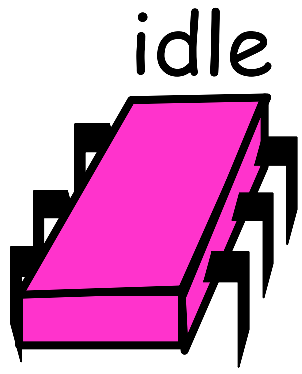
Initially

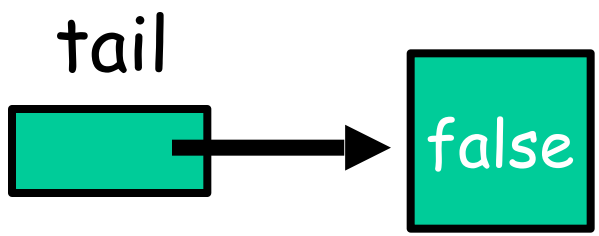


Initially

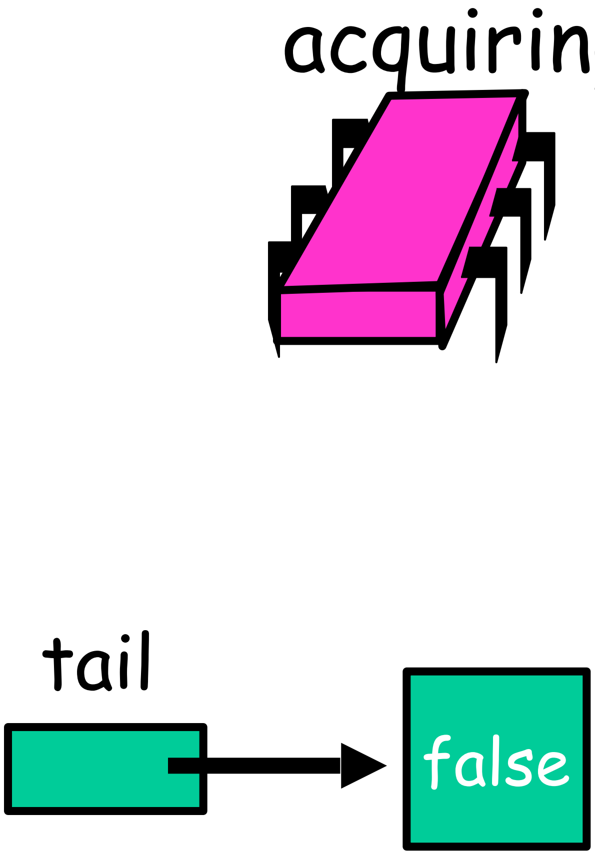


Initially

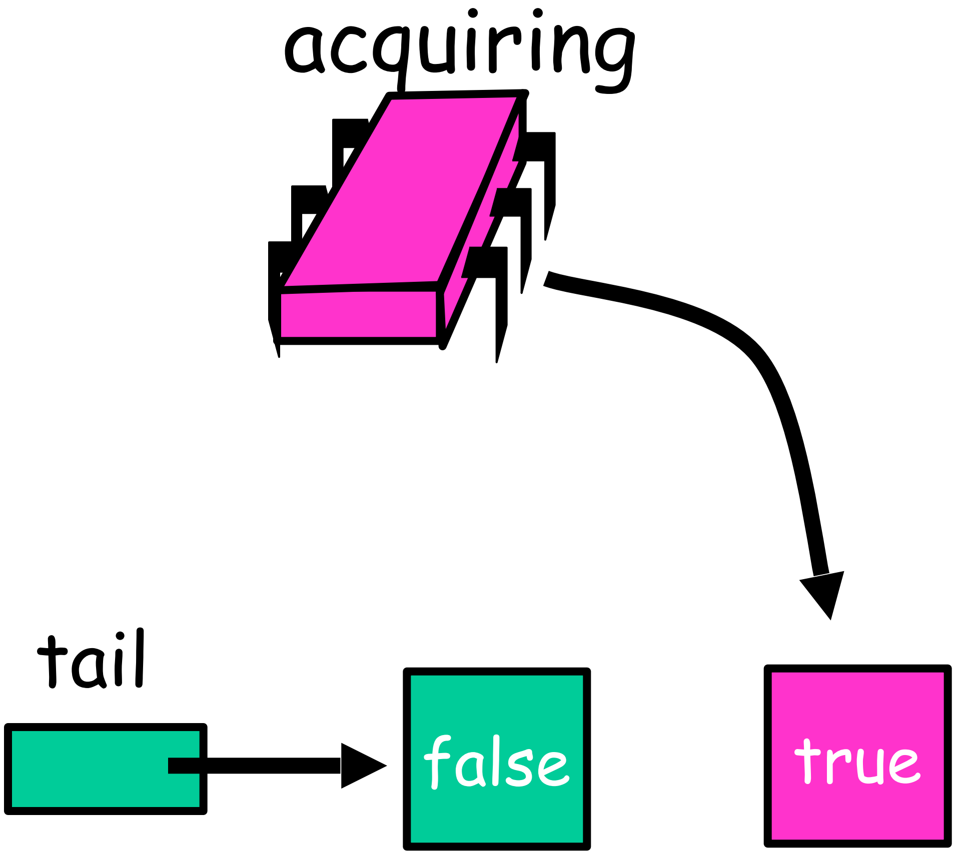




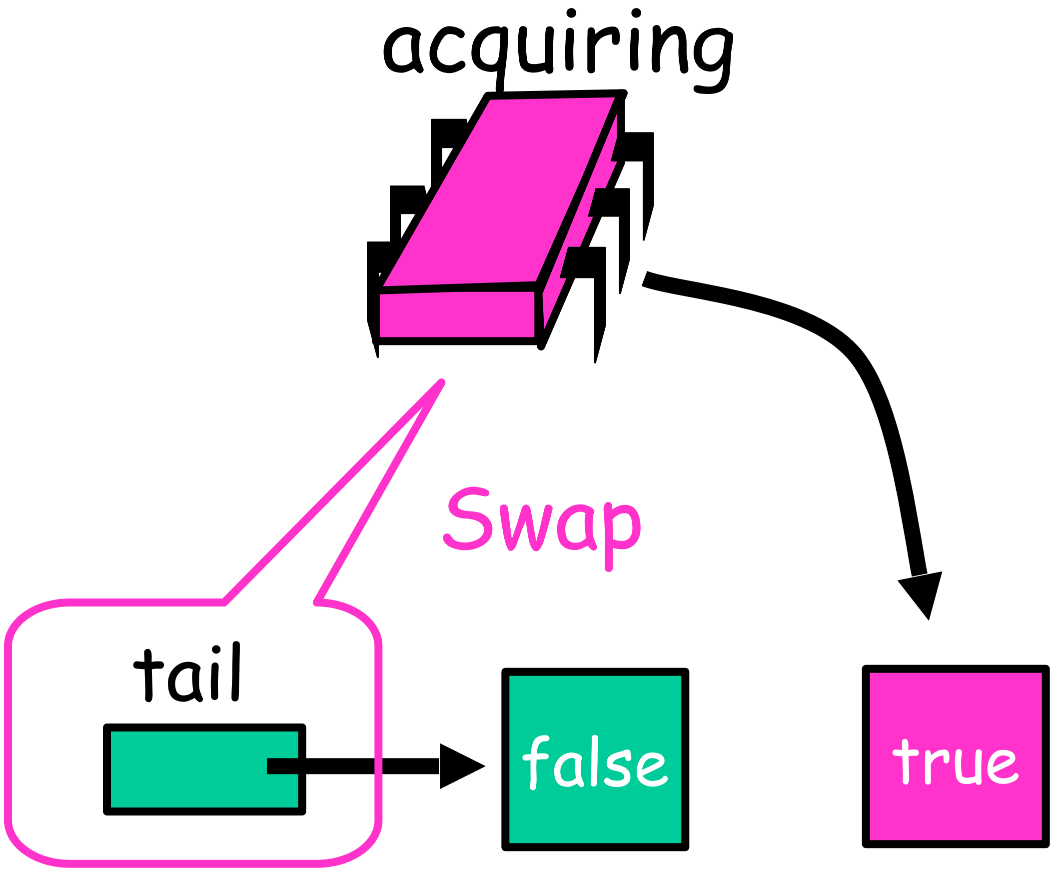
Purple Wants the Lock



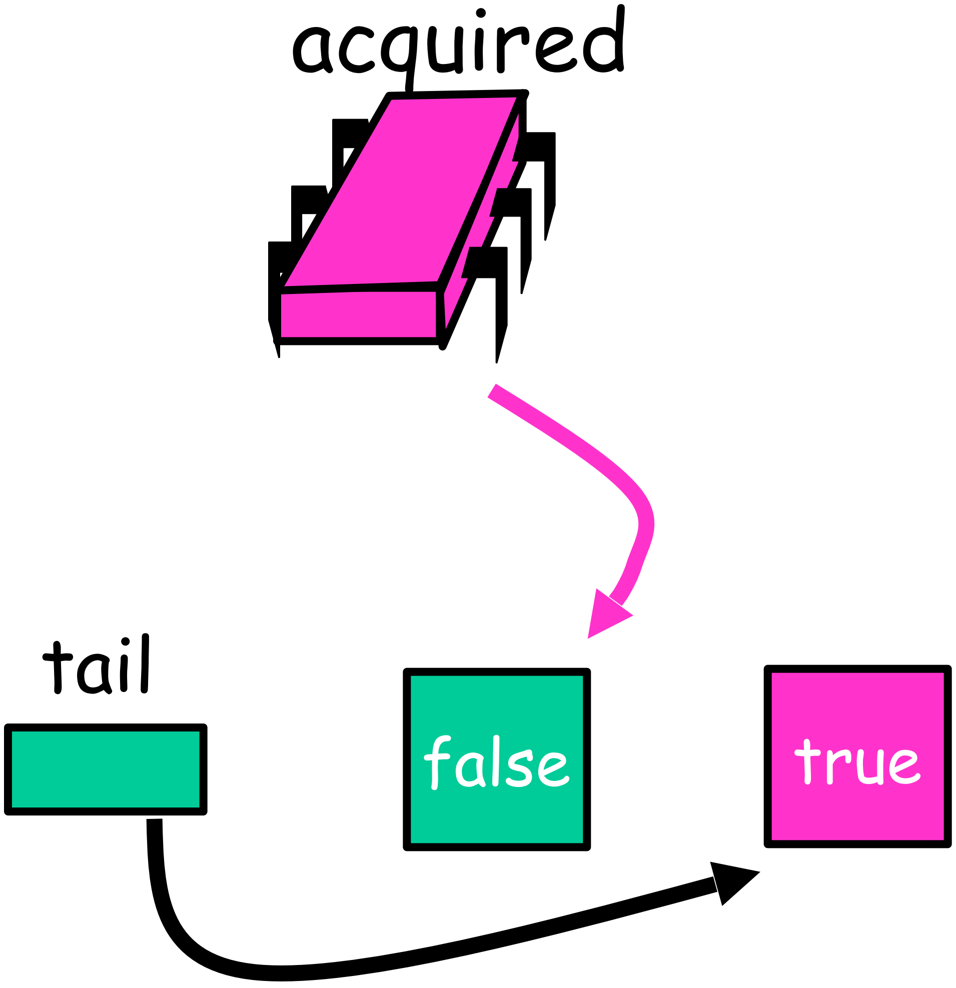
Purple Wants the Lock



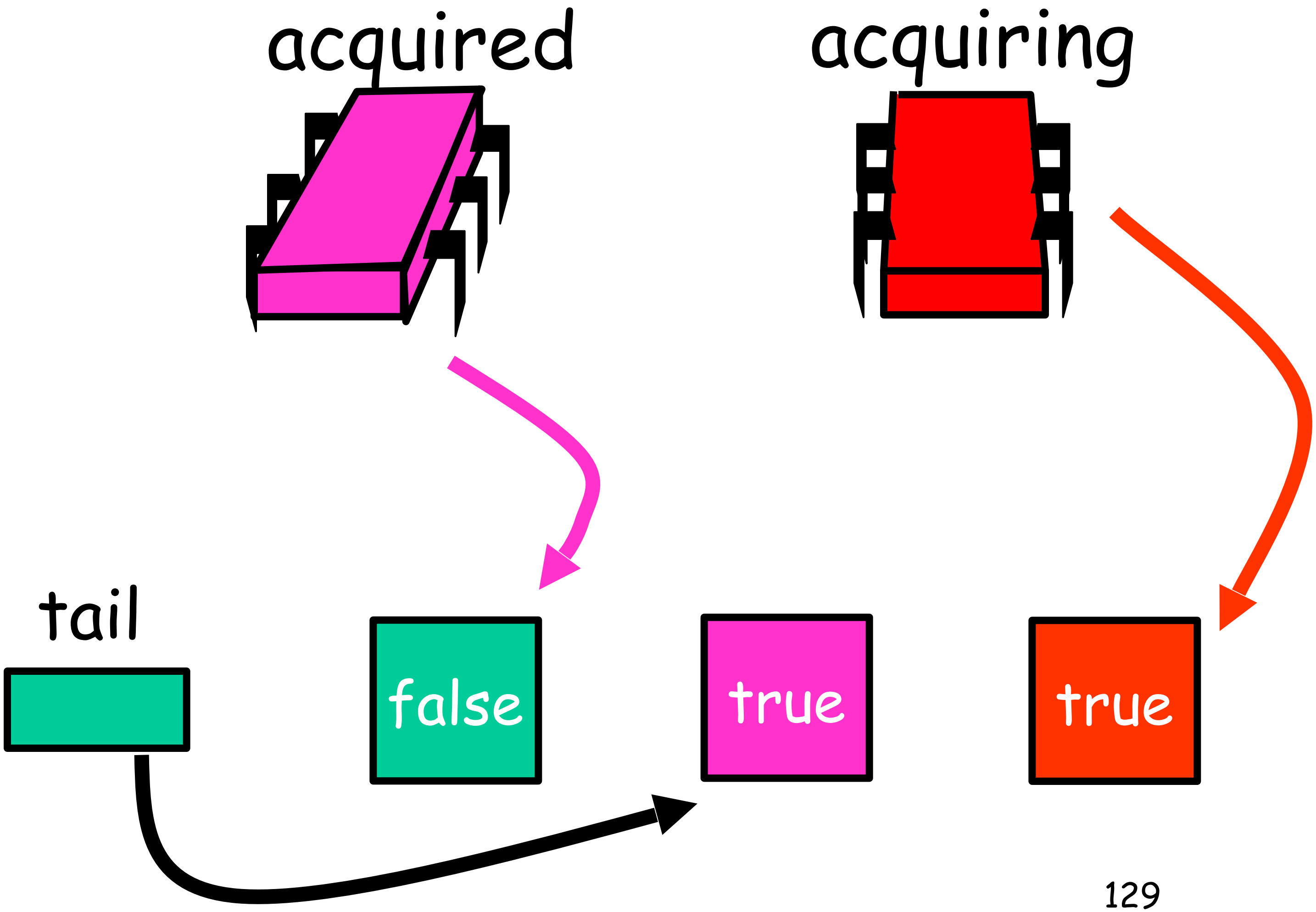
Purple Wants the Lock



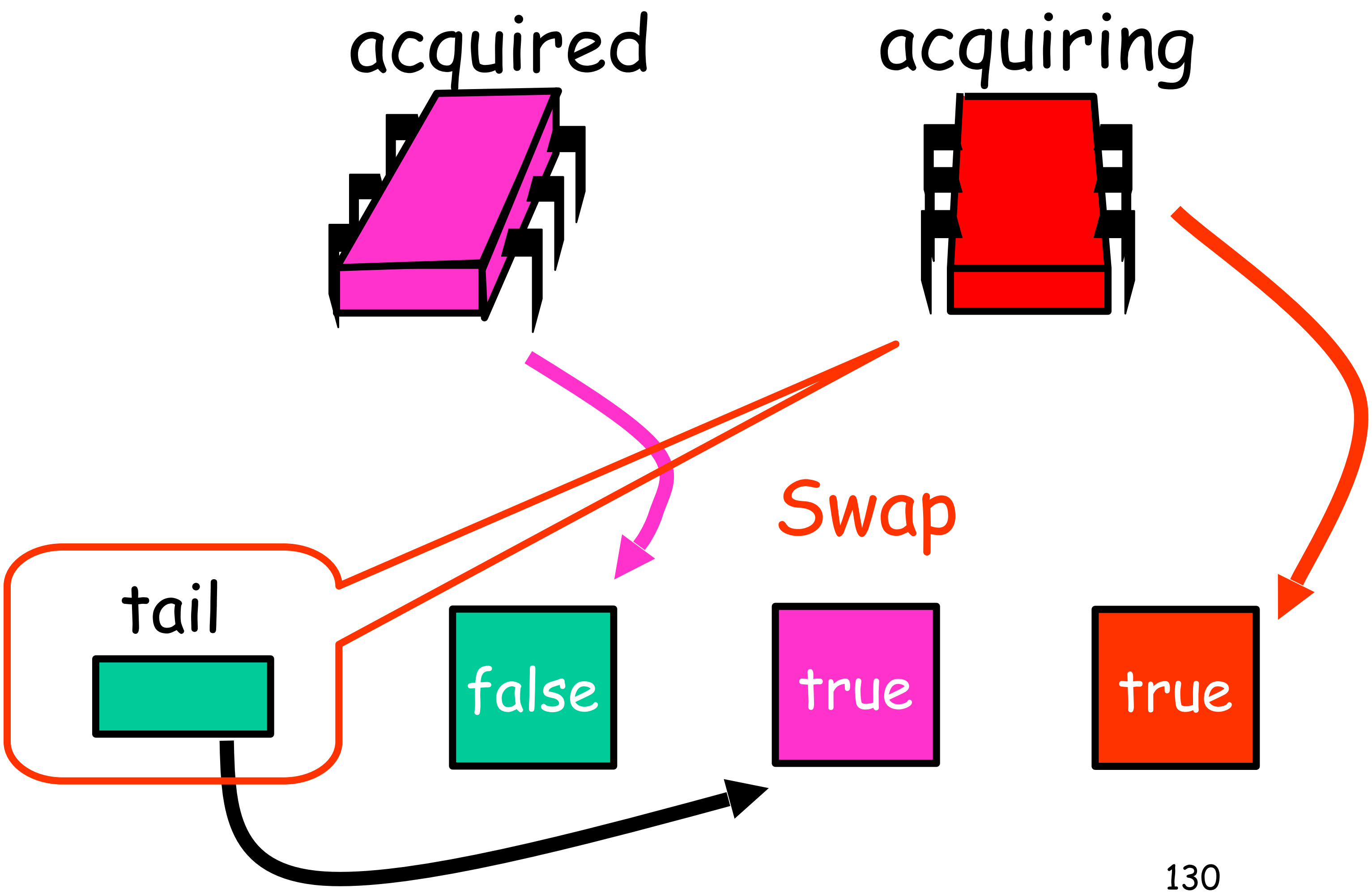
Purple Has the Lock



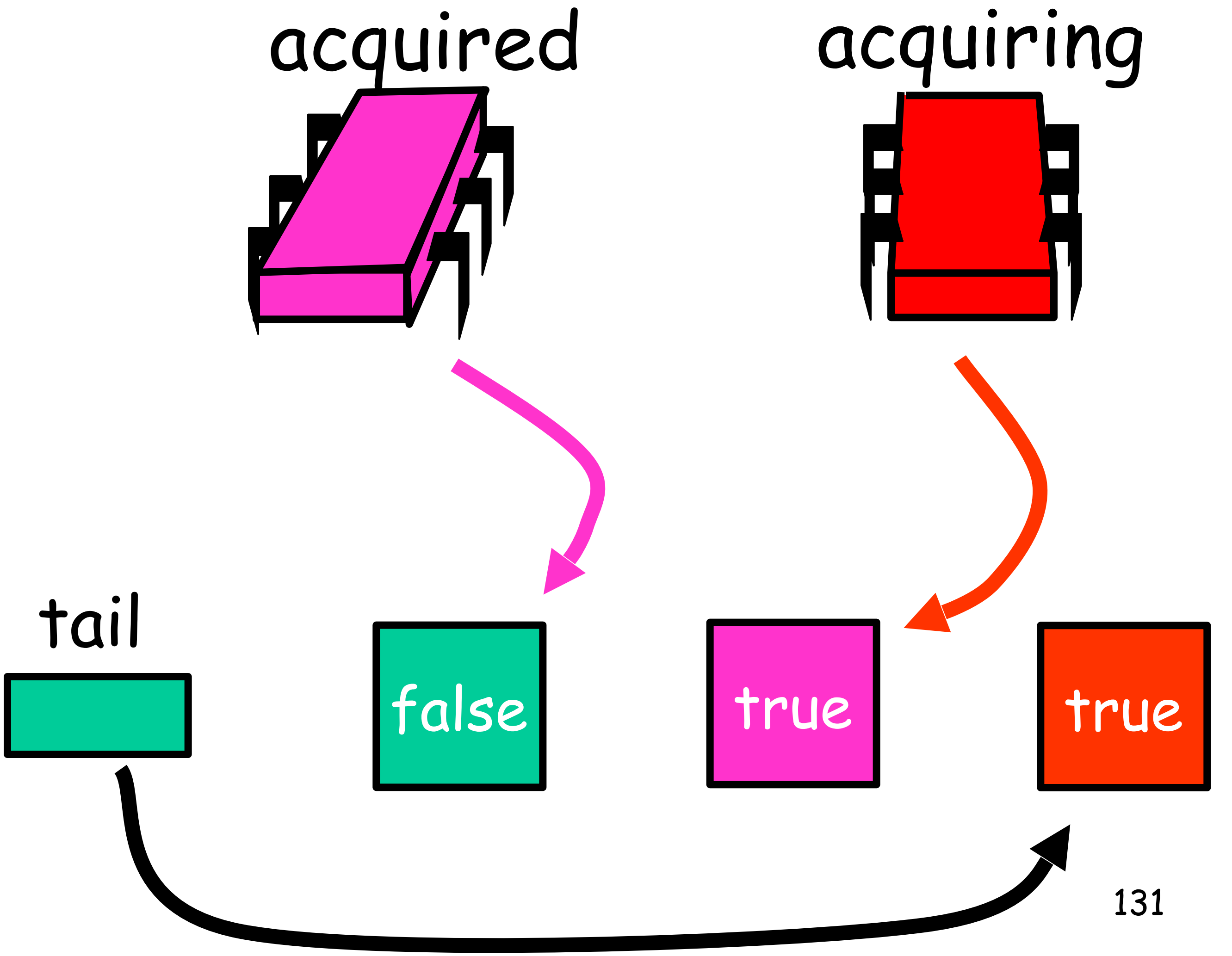
Red Wants the Lock



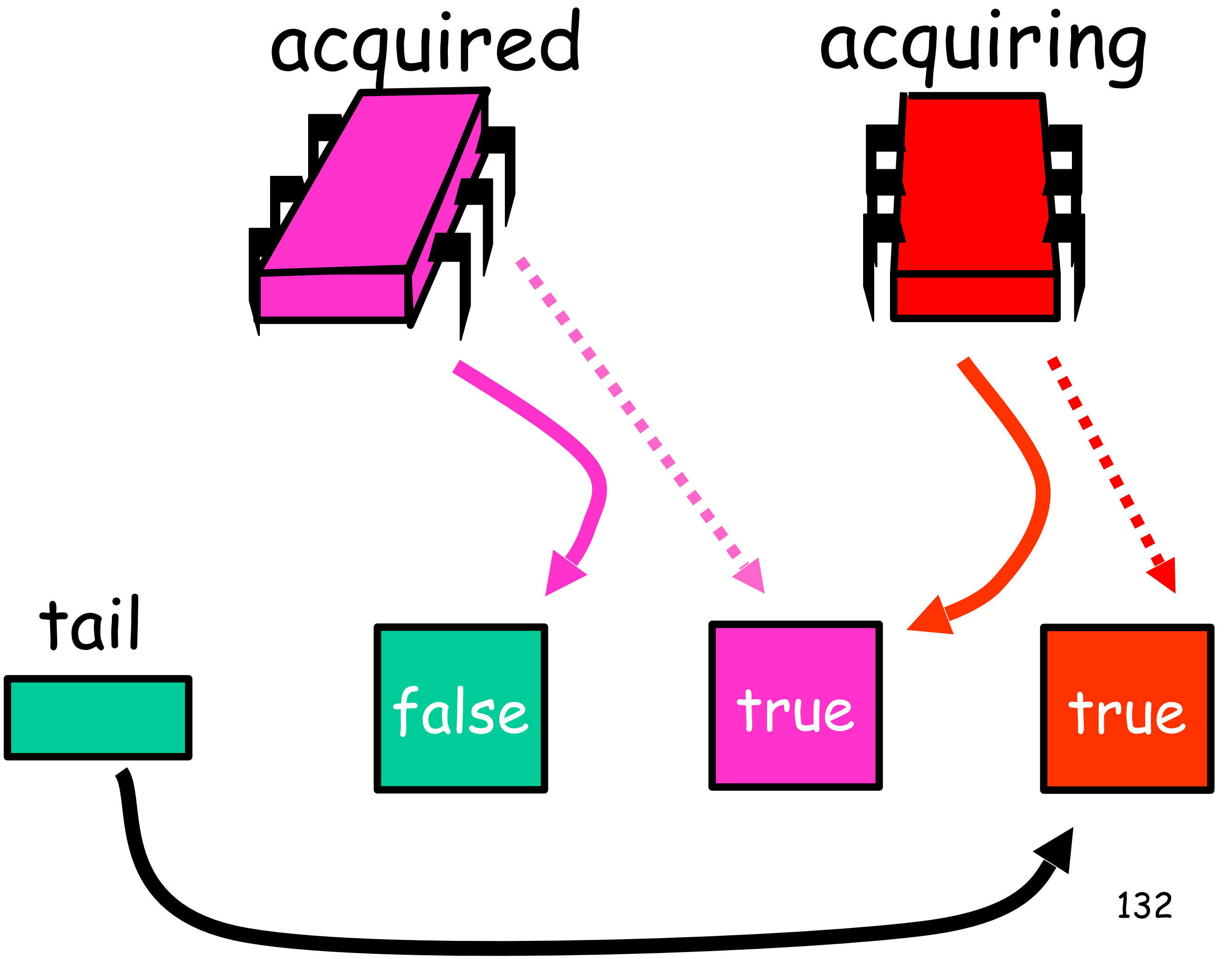
Red Wants the Lock



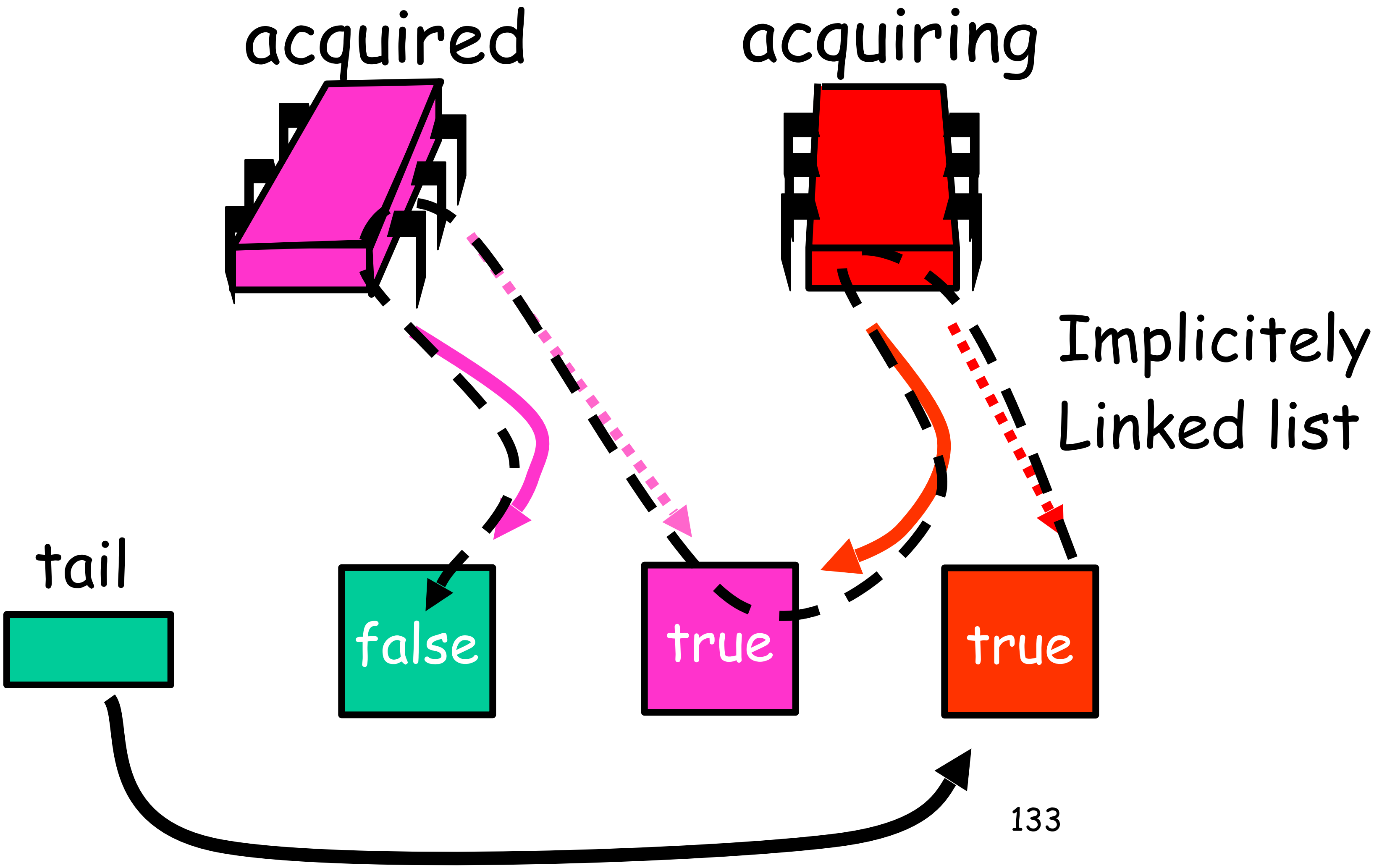
Red Wants the Lock



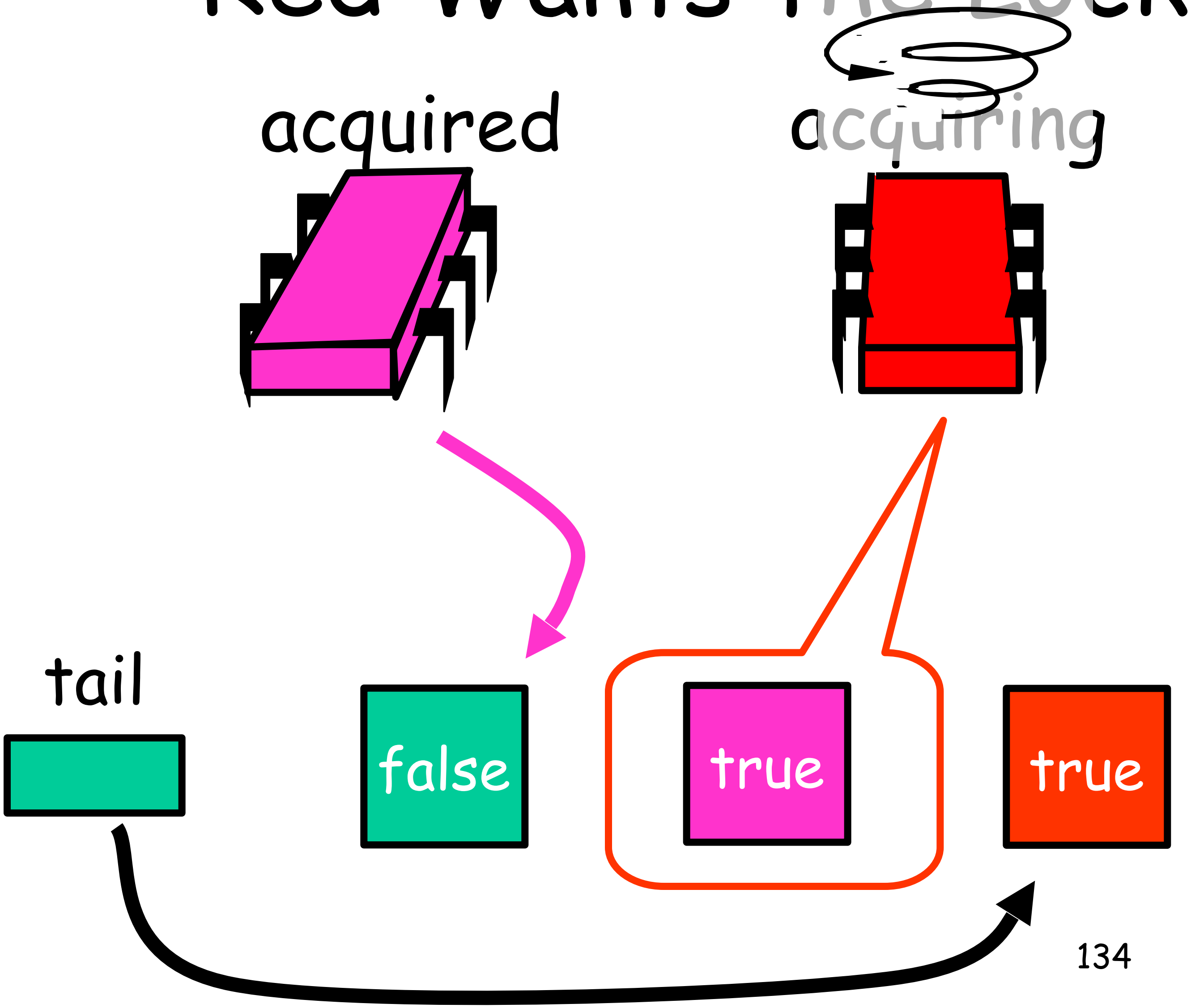
Red Wants the Lock



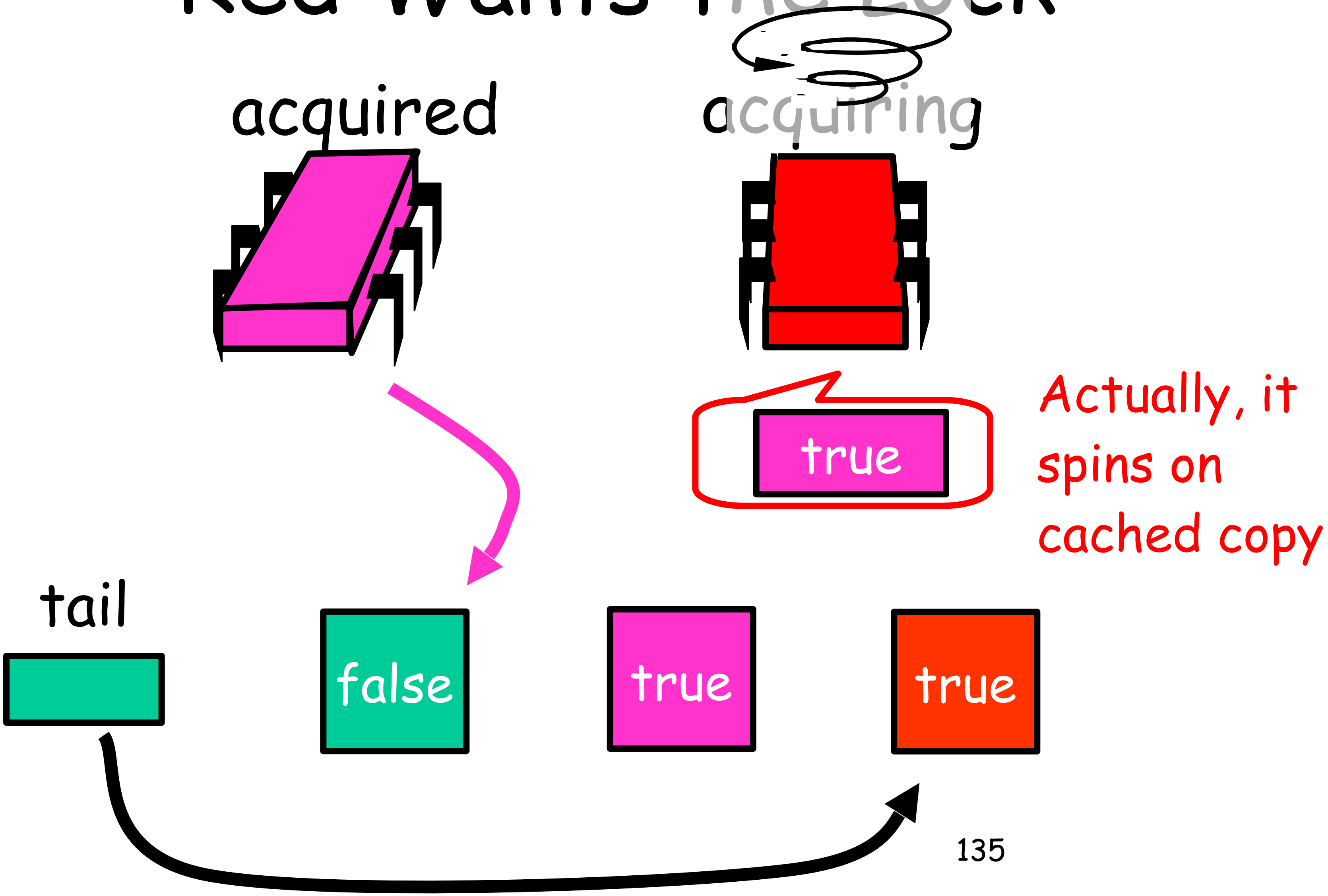
Red Wants the Lock



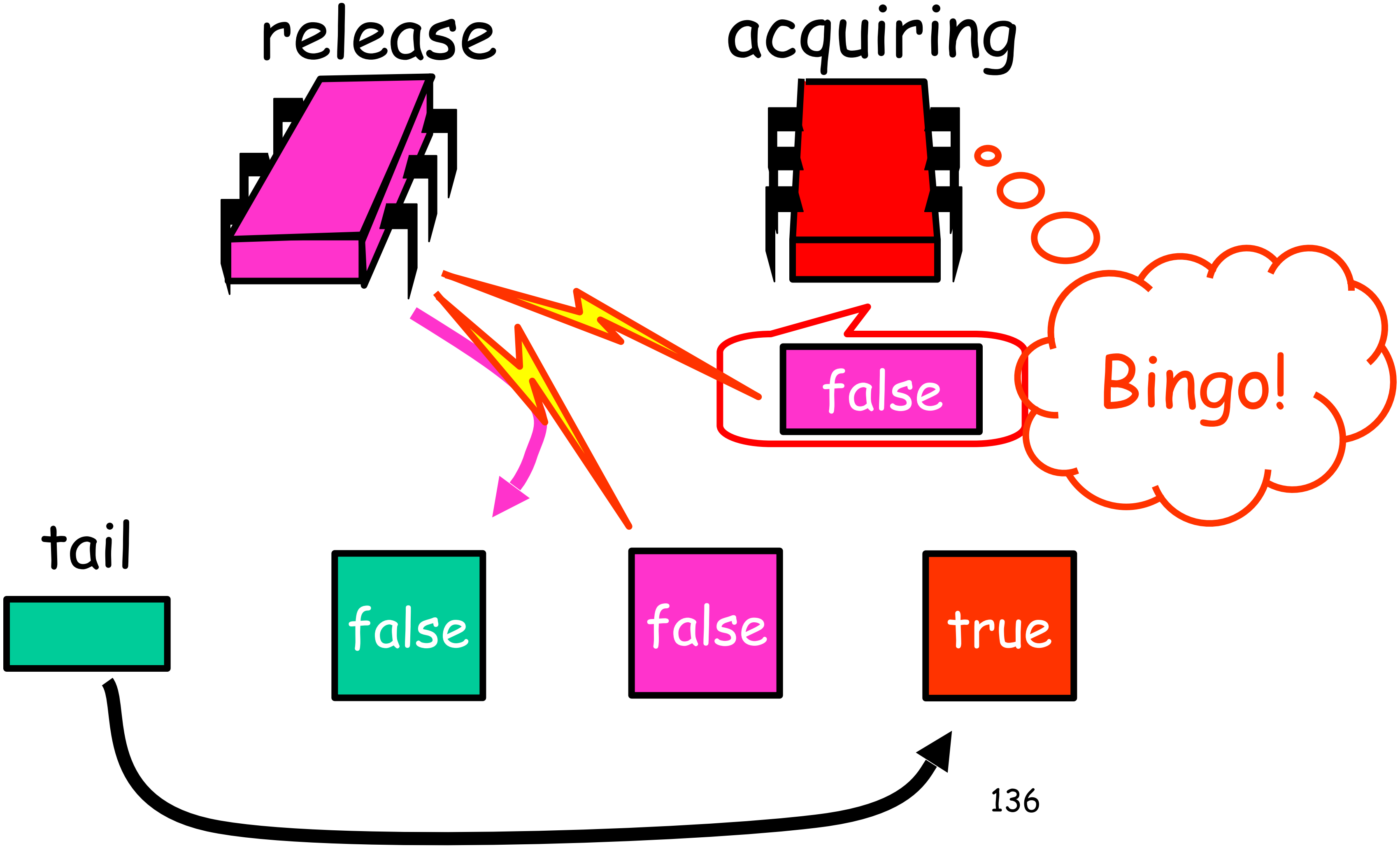
Red Wants the Lock



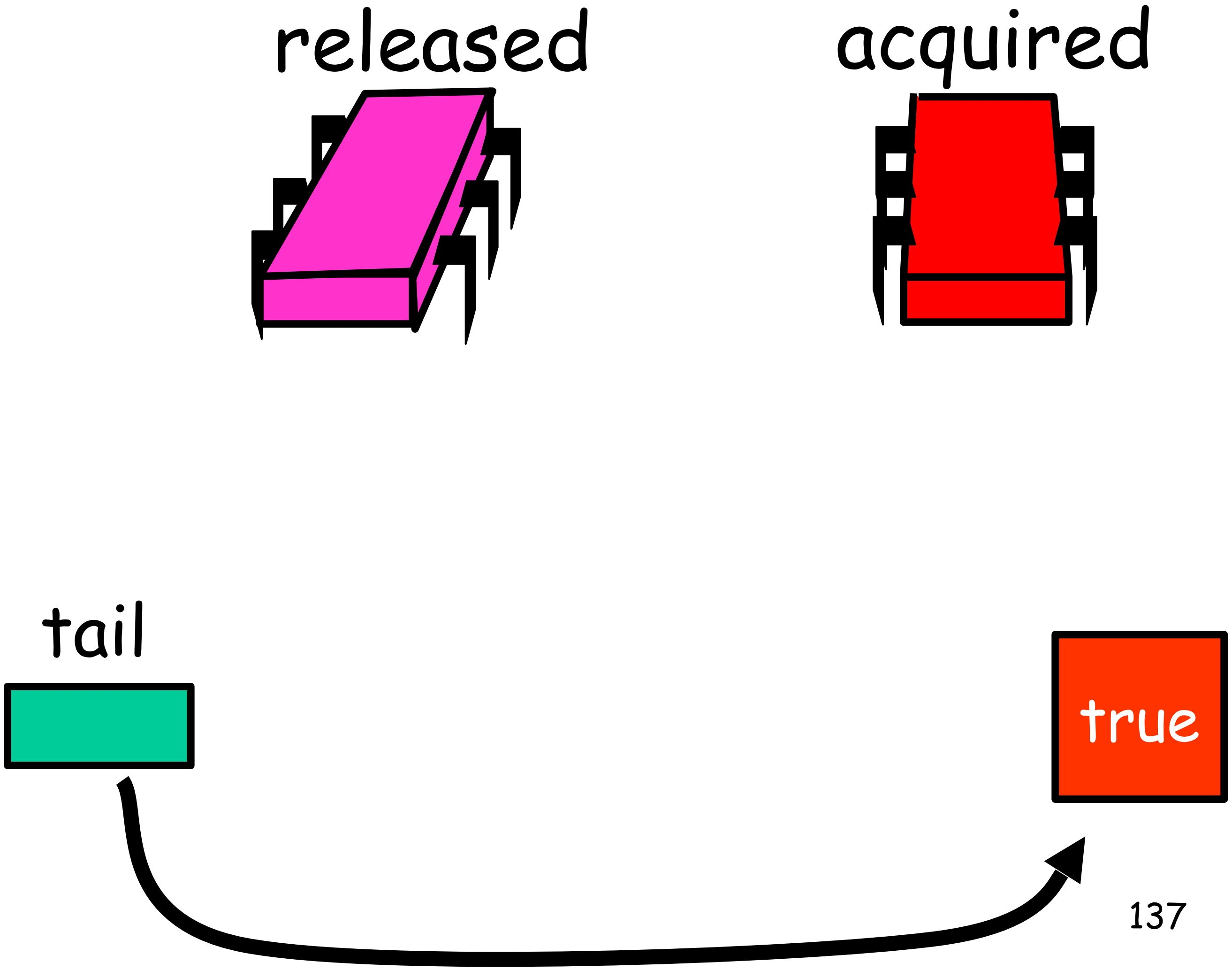
Red Wants the Lock



Purple Releases



Purple Releases



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) {}
 }}
```


(3)

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) {}
 }}
```

Tail of the queue

(3)

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

(3)

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

(3)

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

(3)

CLH Queue Lock

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


(3)

CLH Queue Lock

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

Notify successor

(3)

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;
 }
}
```

(notice that we actually don't reuse myNode. Code in book shows how its done.)

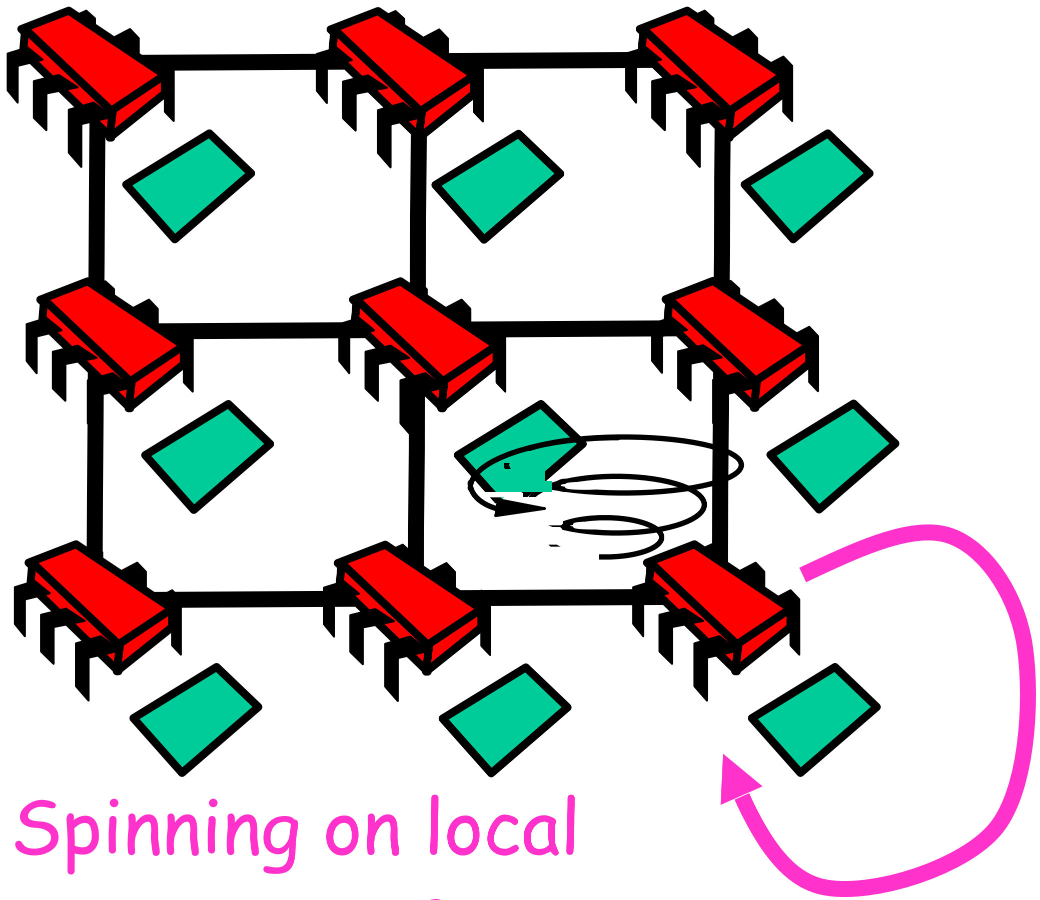
CLH Lock

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

NUMA Architecturs

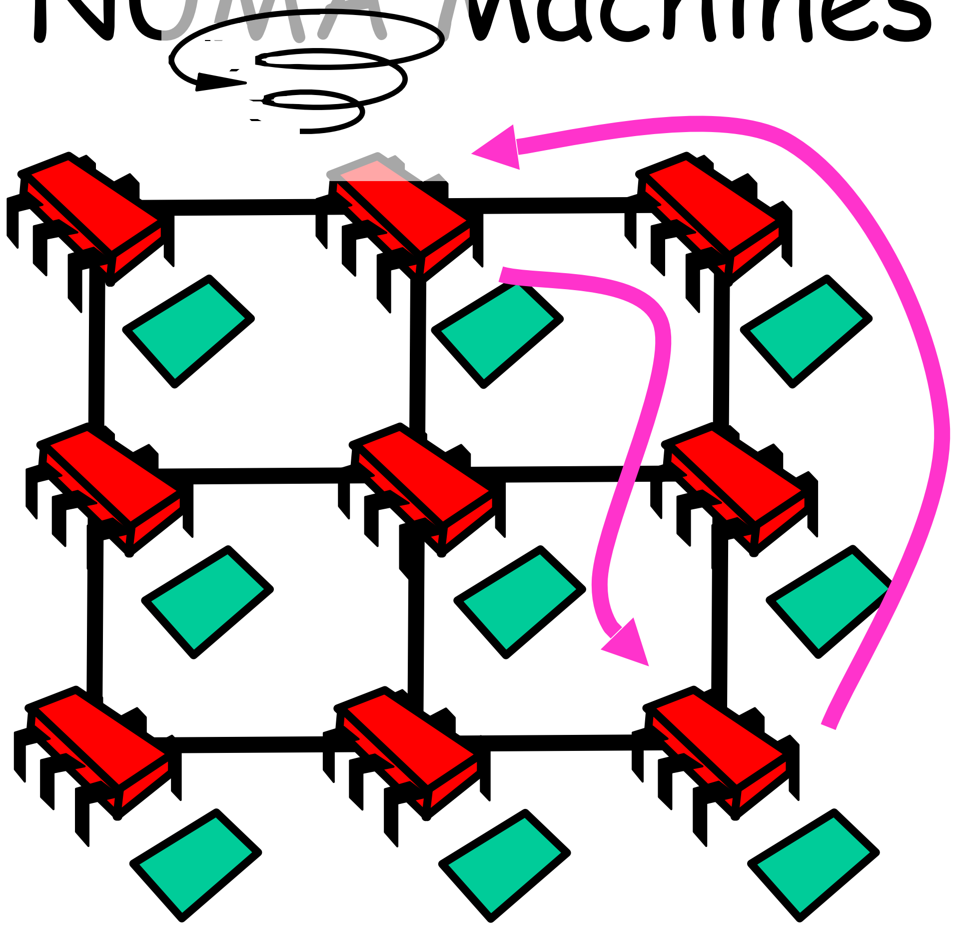
- Acronym:
  - **N**on-**U**niform **M**emory **A**rchitecture
- Illusion:
  - Flat shared memory
- Truth:
  - No caches (sometimes)
  - Some memory regions faster than others

NUMA Machines



Spinning on local memory is fast

NUMA Machines



Spinning on remote memory is slow

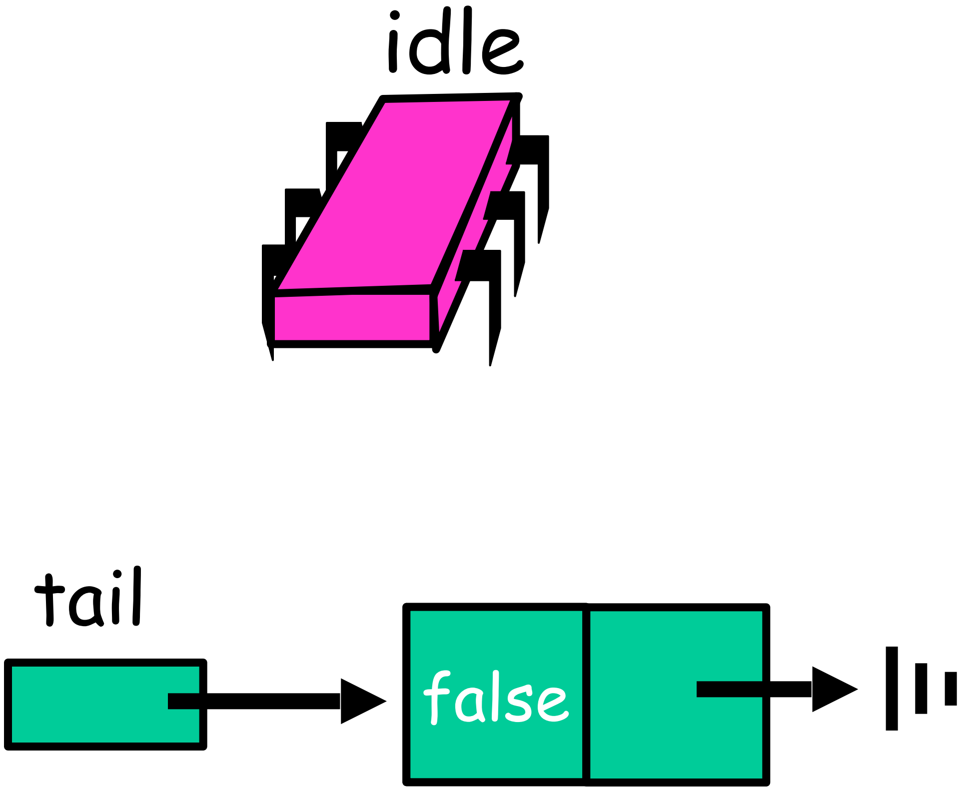
CLH Lock

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

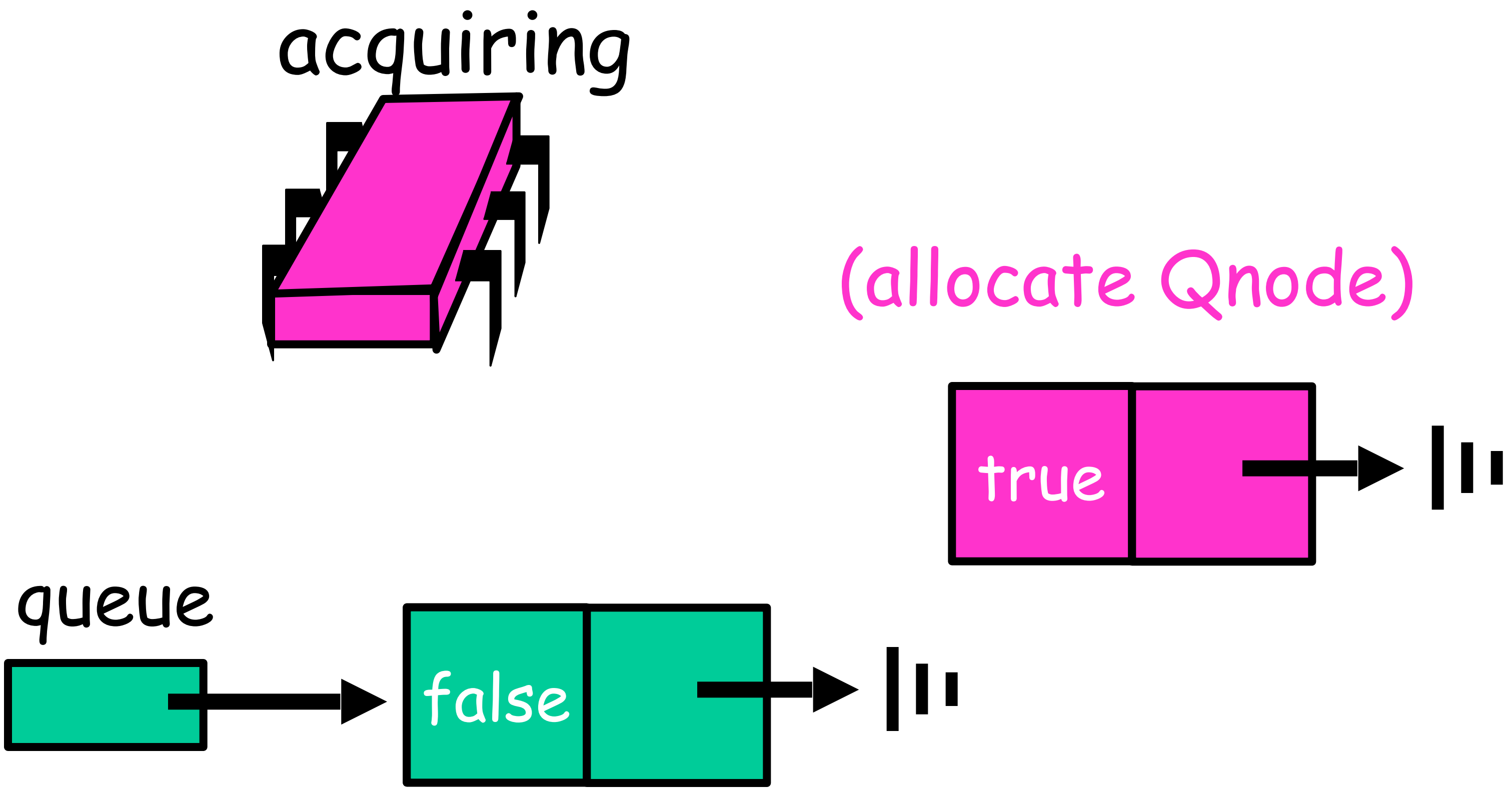
MCS Lock

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

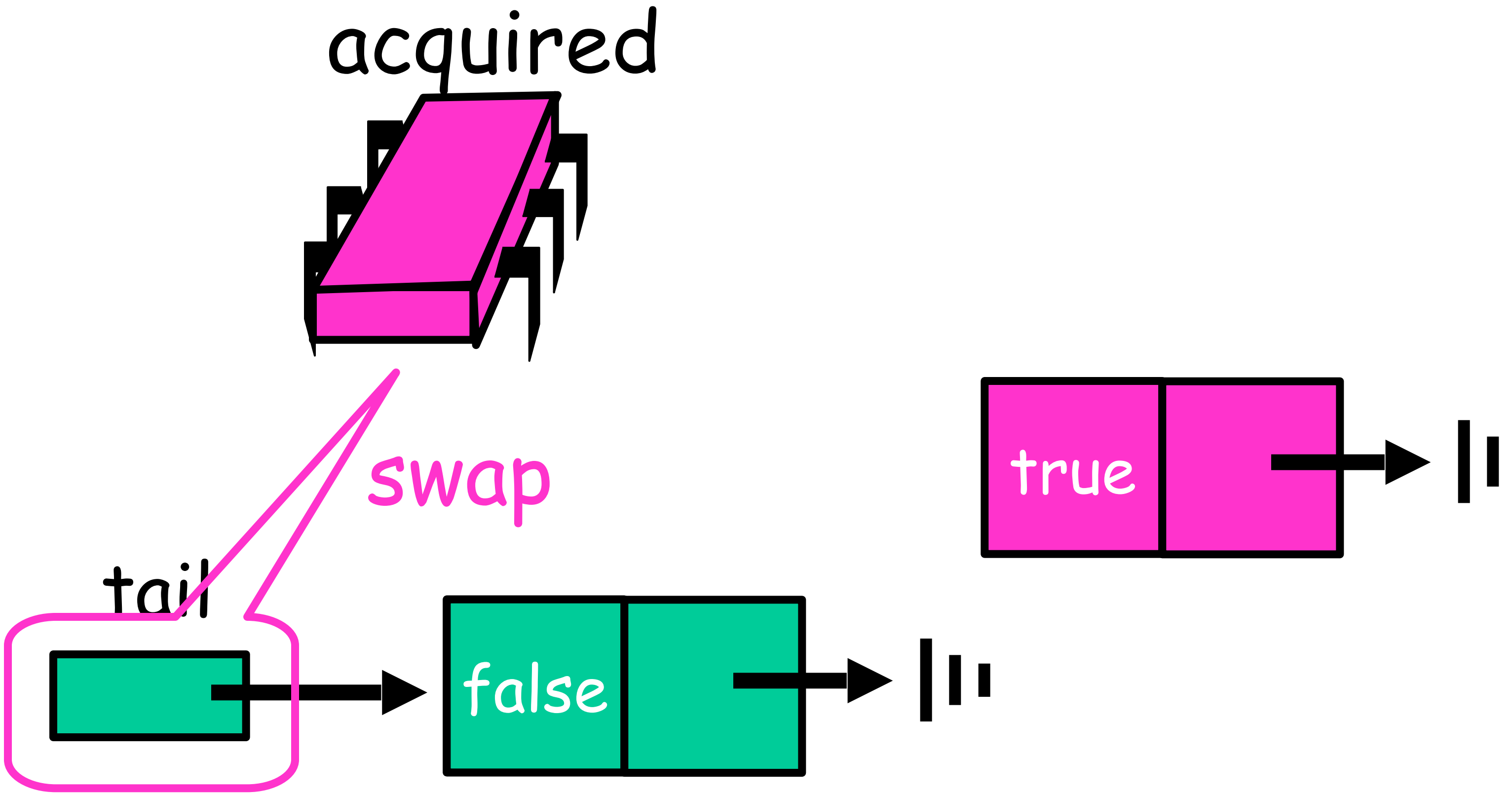
Initially



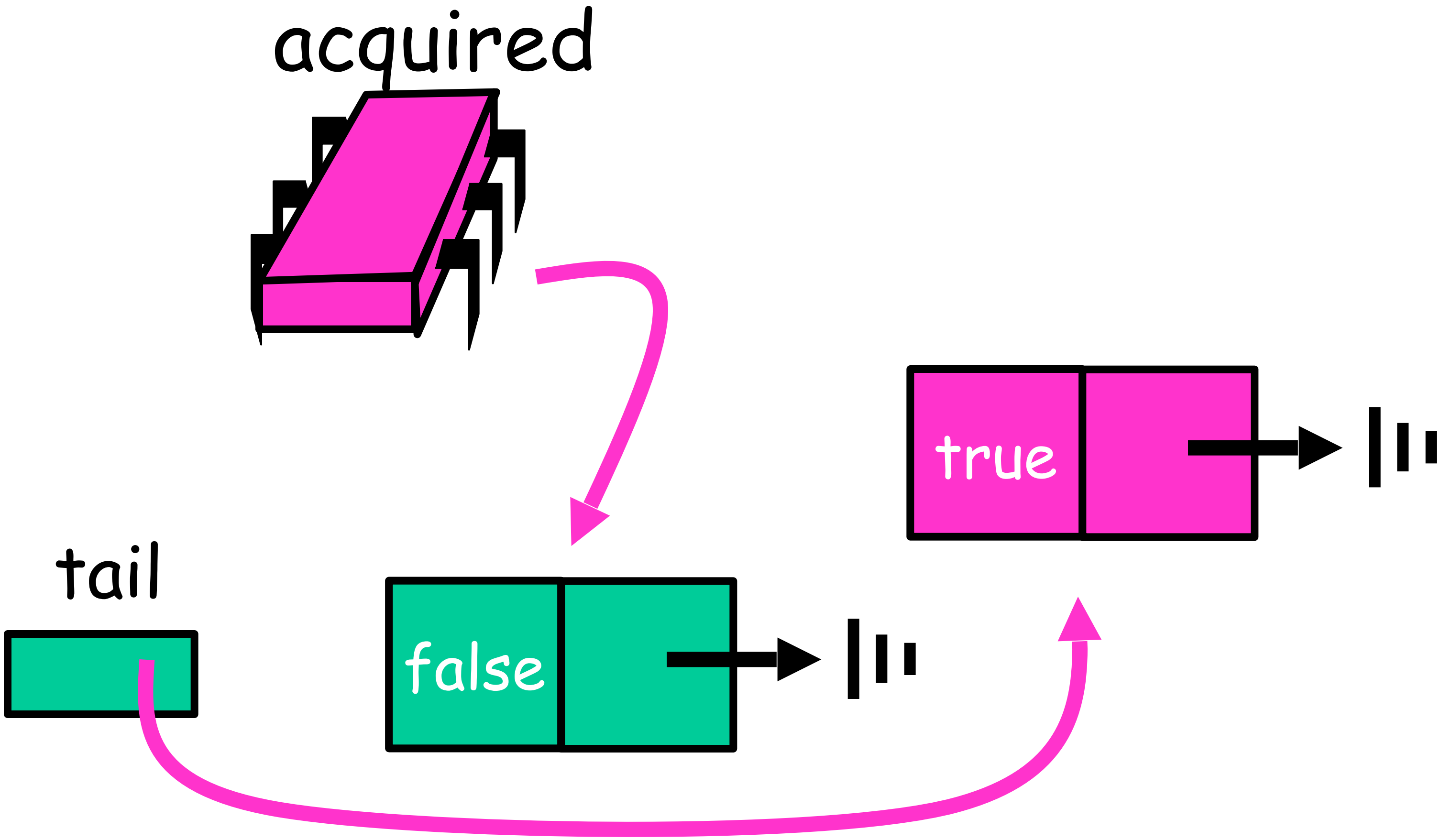
Acquiring



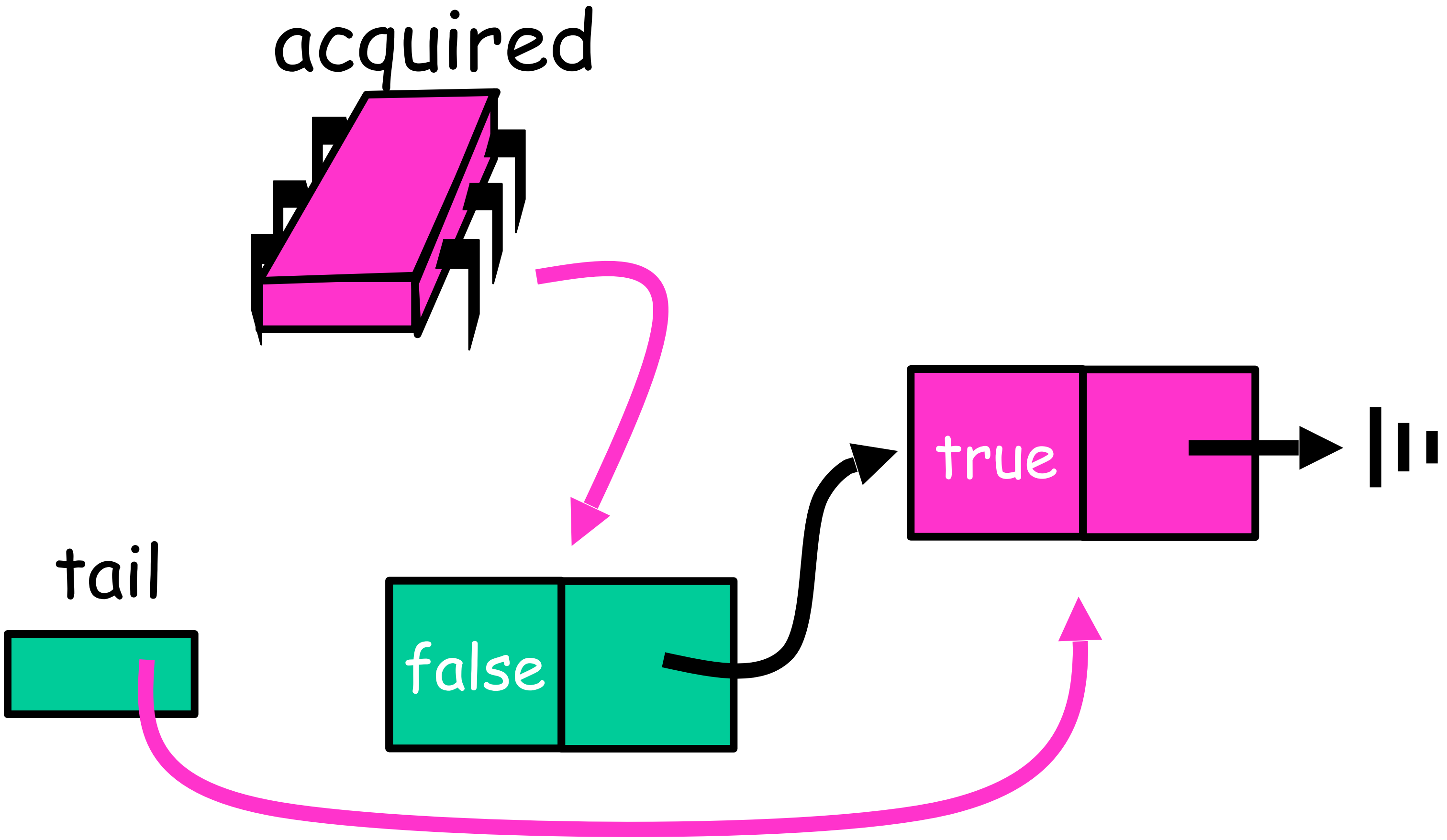
Acquiring



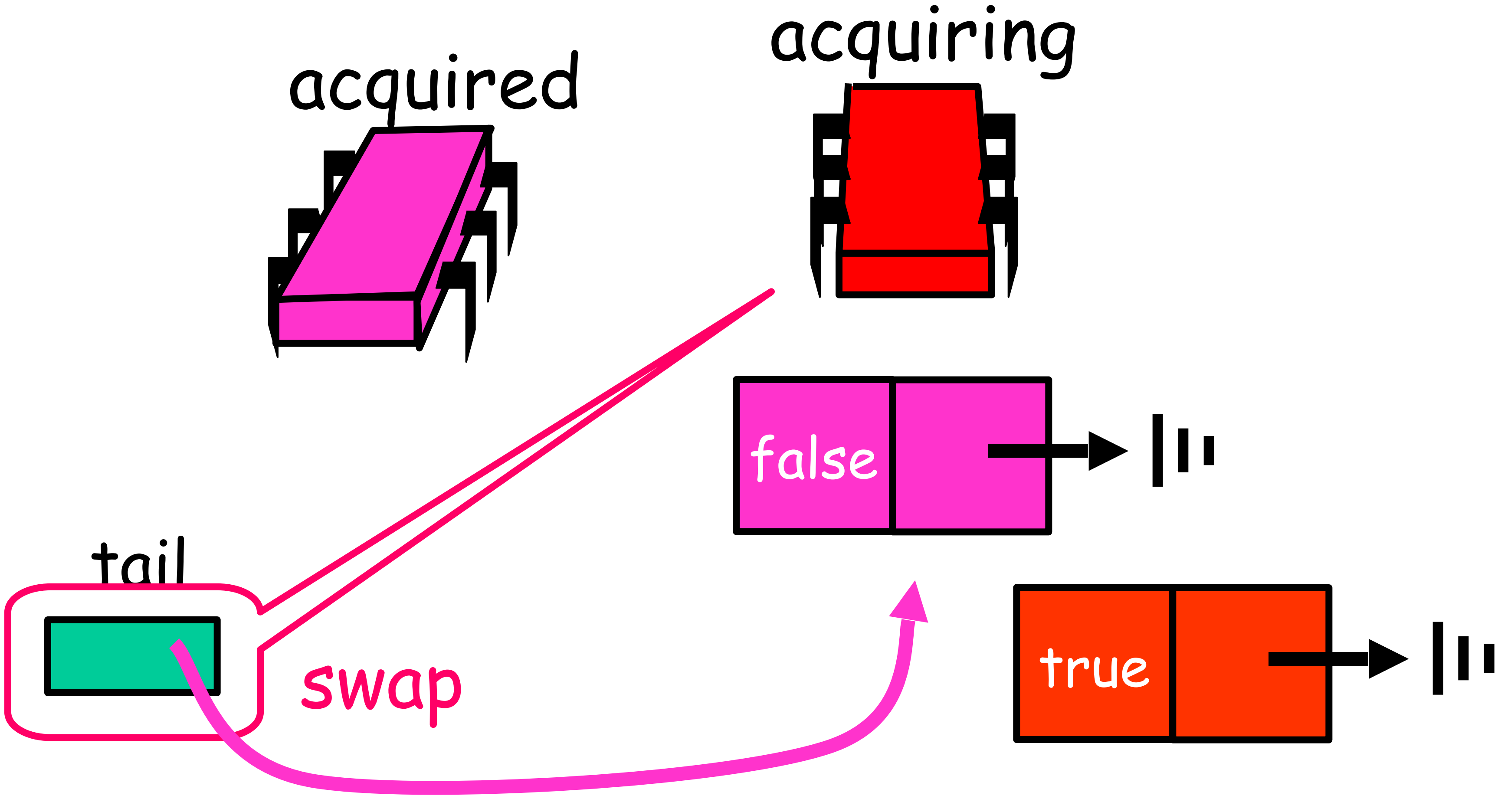
Acquiring



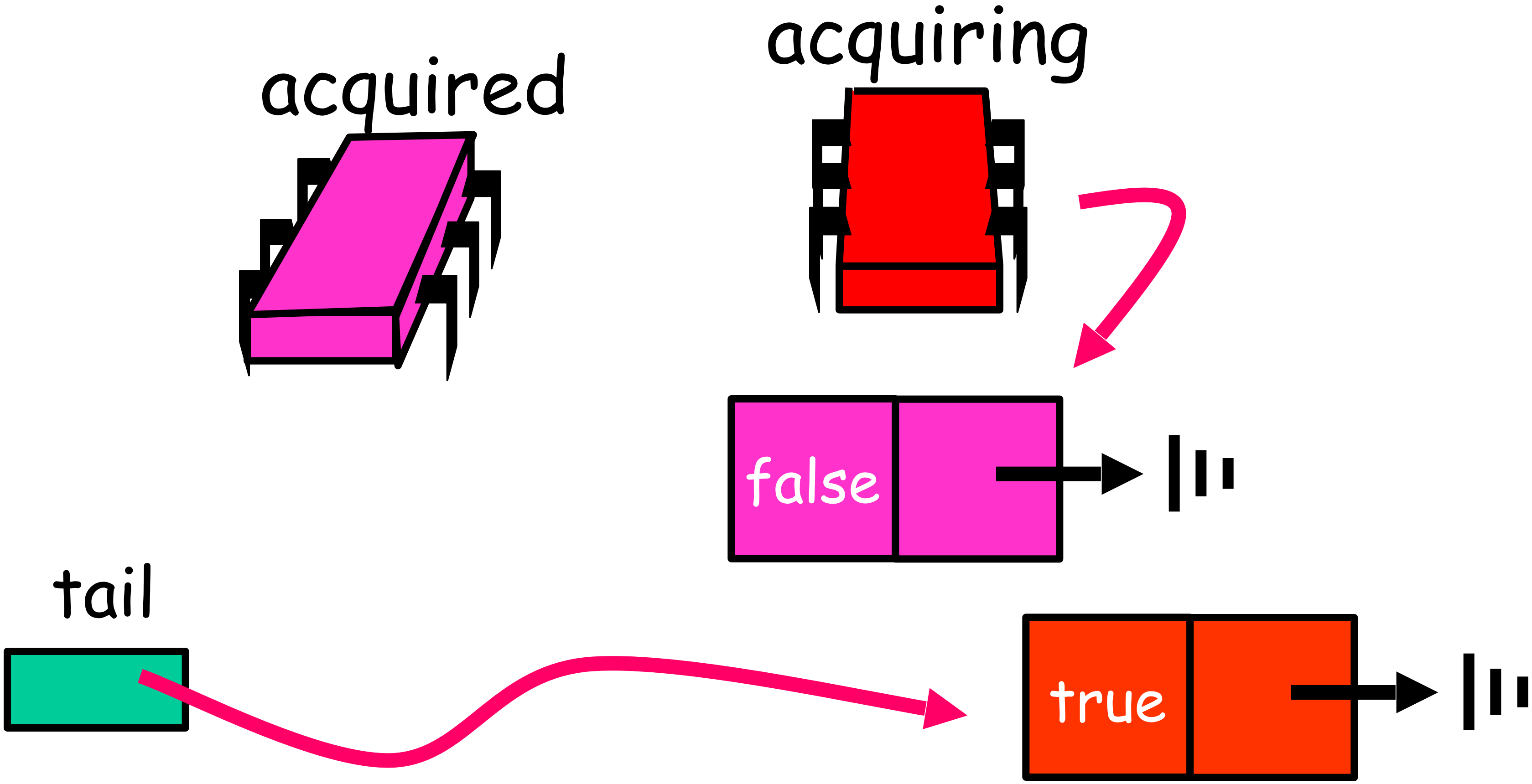
Acquired



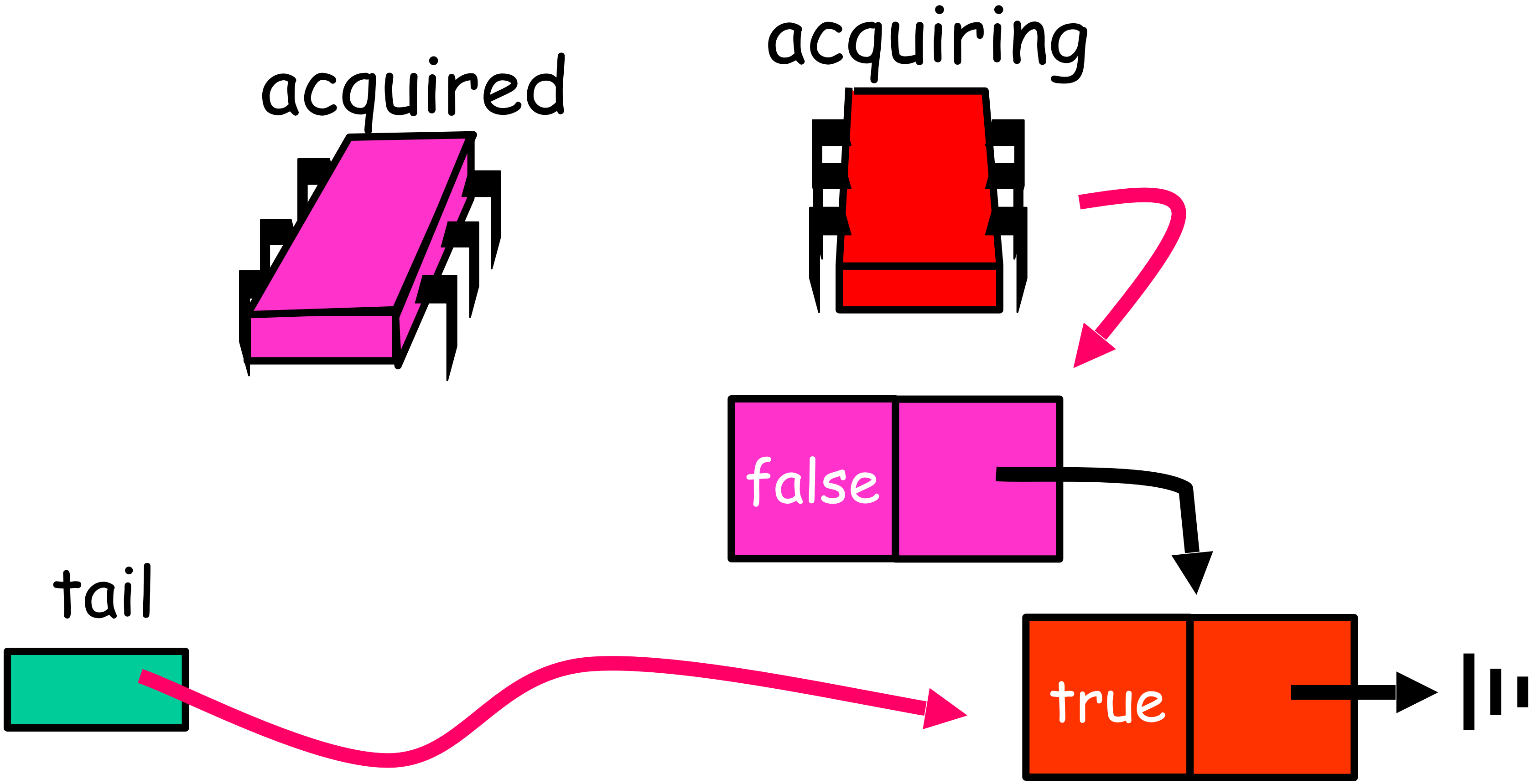
Acquiring



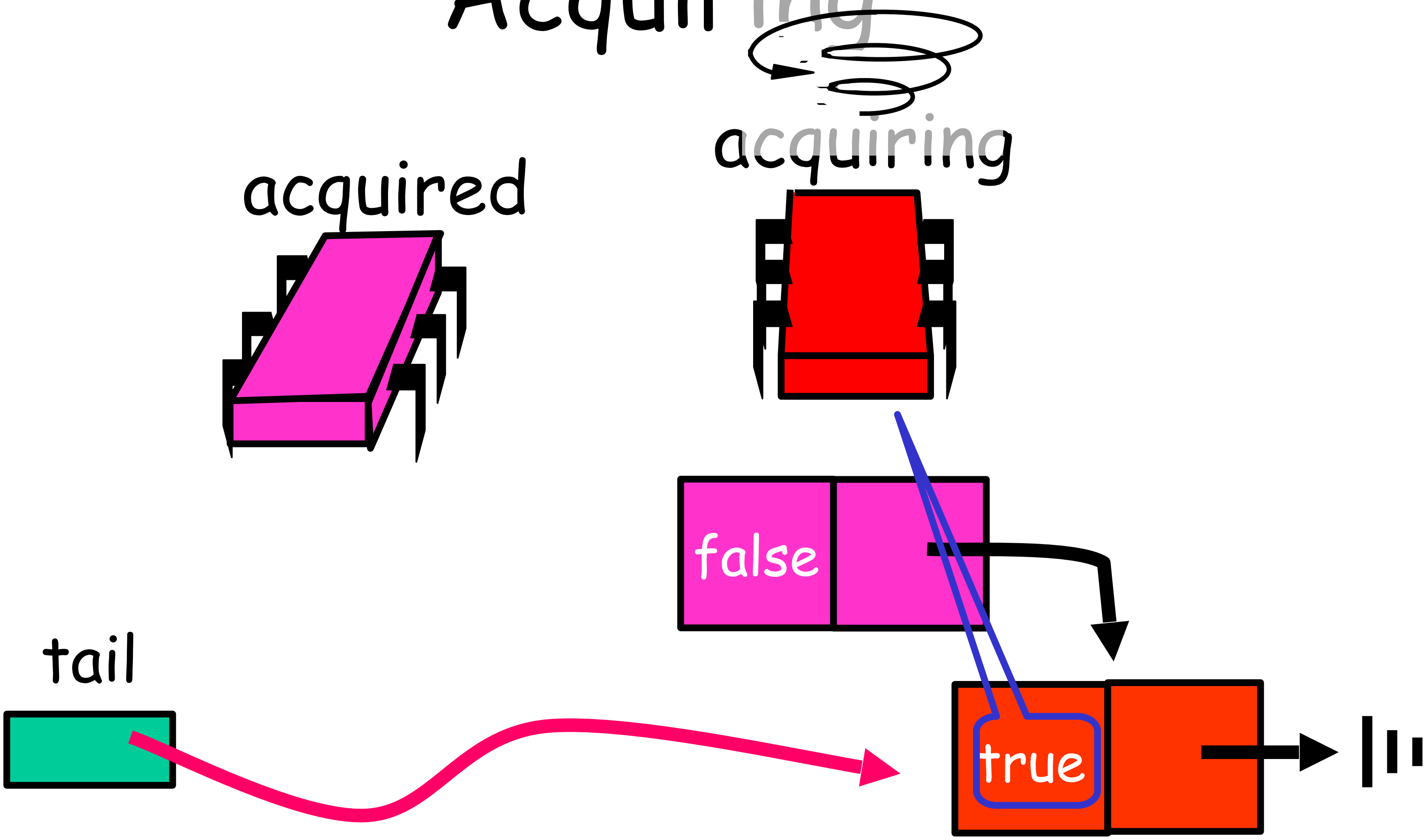
Acquiring



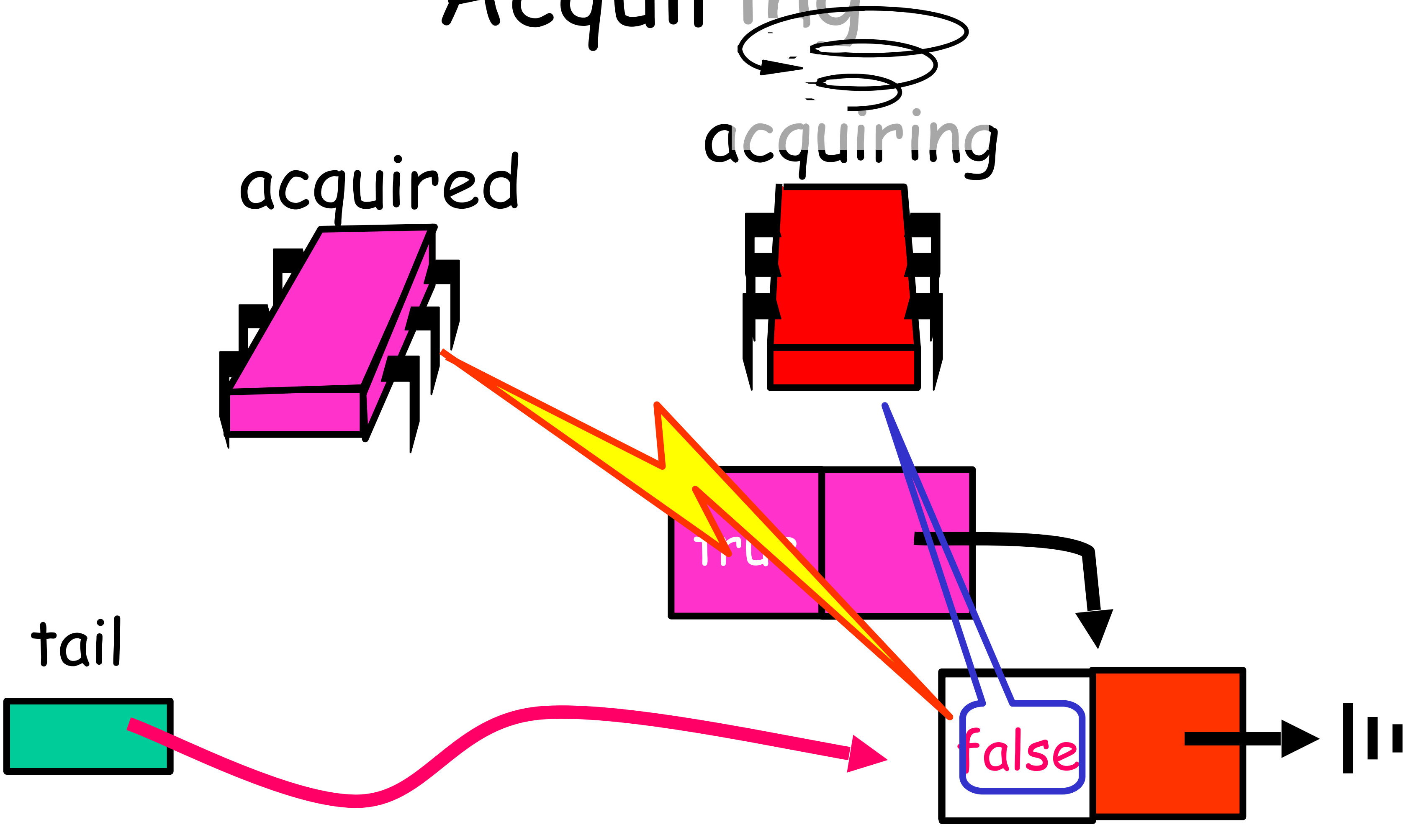
Acquiring



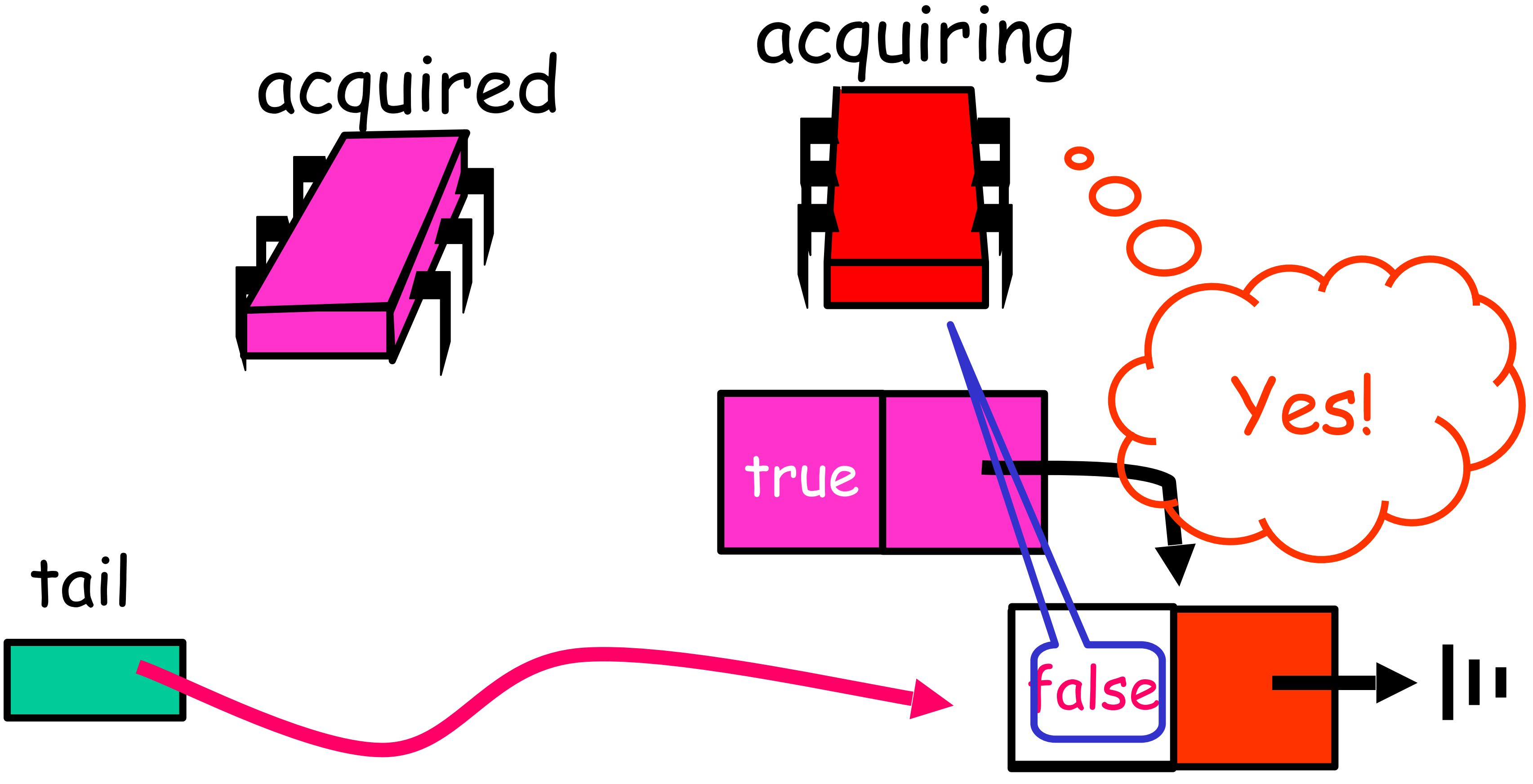
Acquiring



Acquiring



Acquiring



MCS Queue Lock

```
class Qnode {
 boolean locked = false;
 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

(3)

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

(3)

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

(3)

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

(3)

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;
}}
```


(3)

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?

(3)

MCS Queue Lock

If really no successor, return

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

 (3)

MCS Queue Lock

Otherwise wait for successor to catch up

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

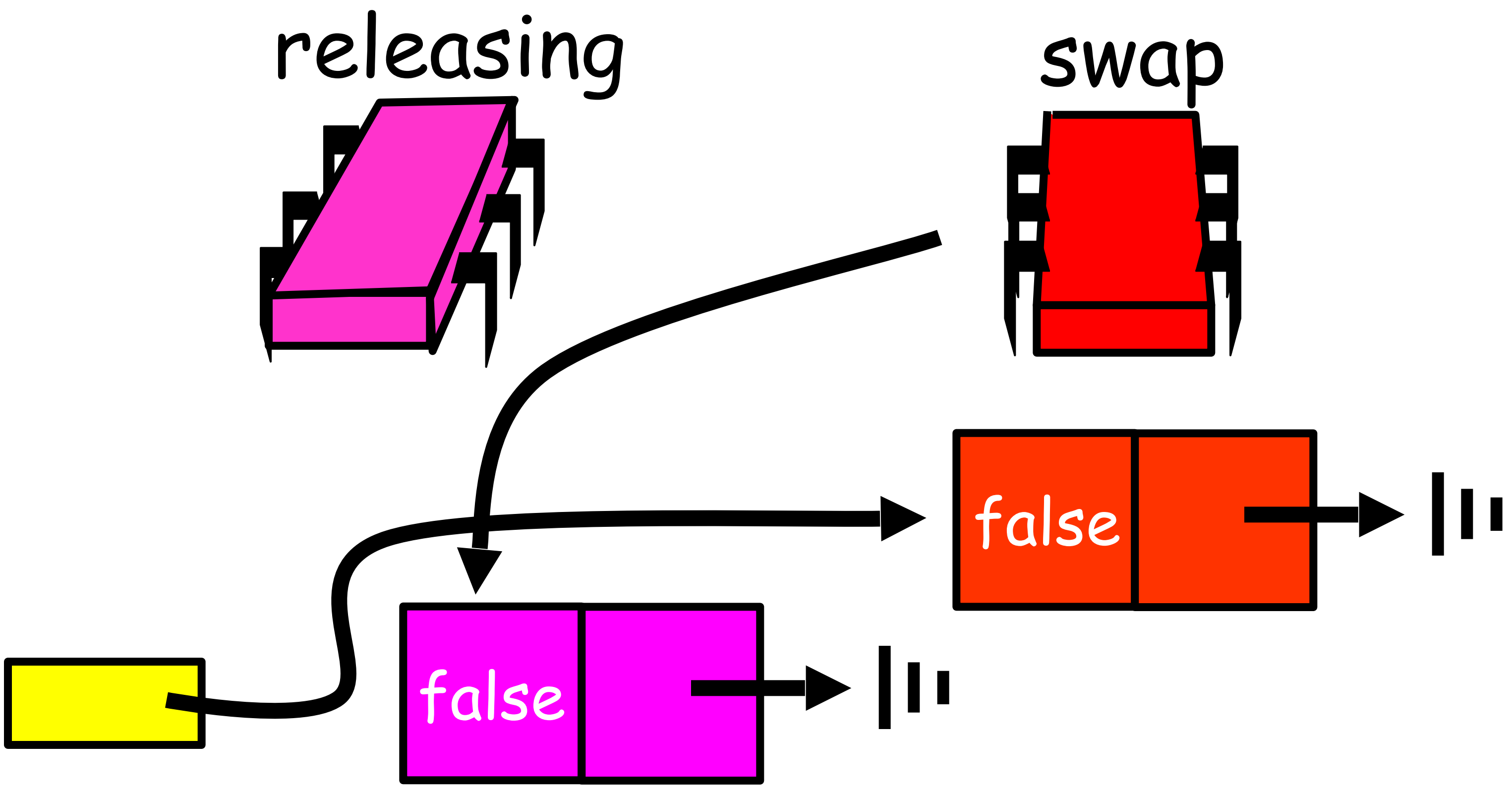
MCS Queue Lock

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

Pass lock to successor

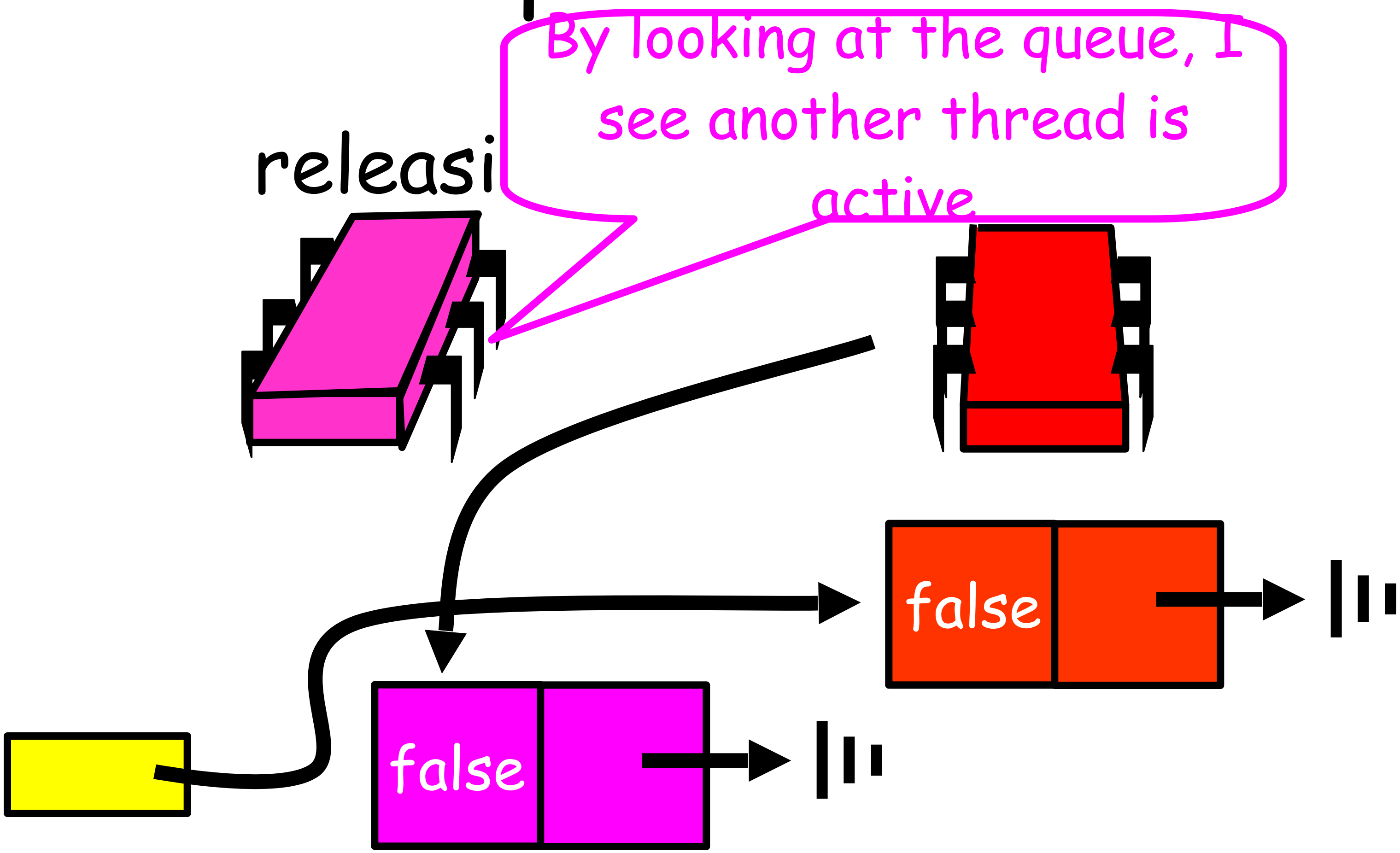
(3)

Purple Release



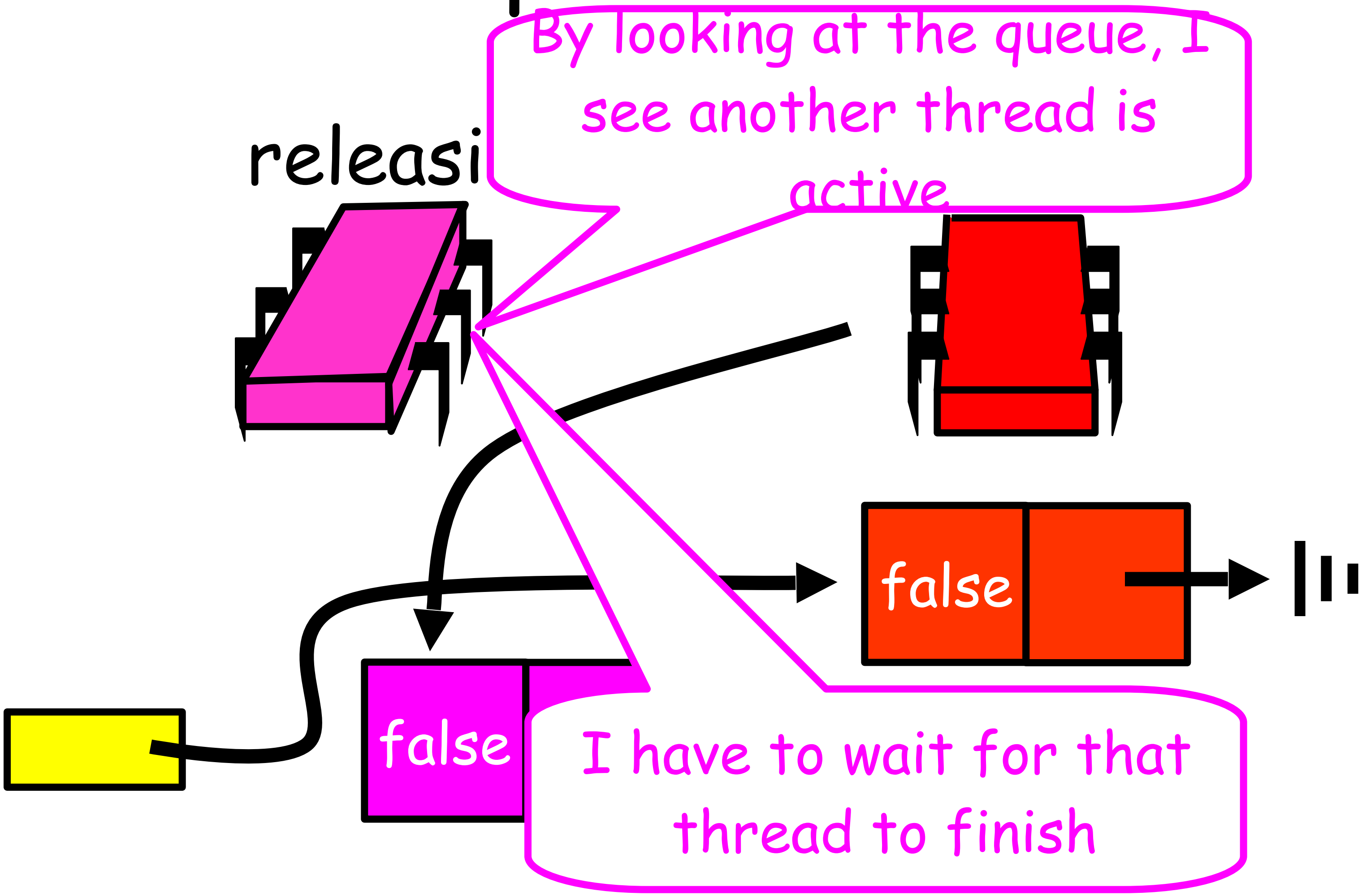
(2)

Purple Release



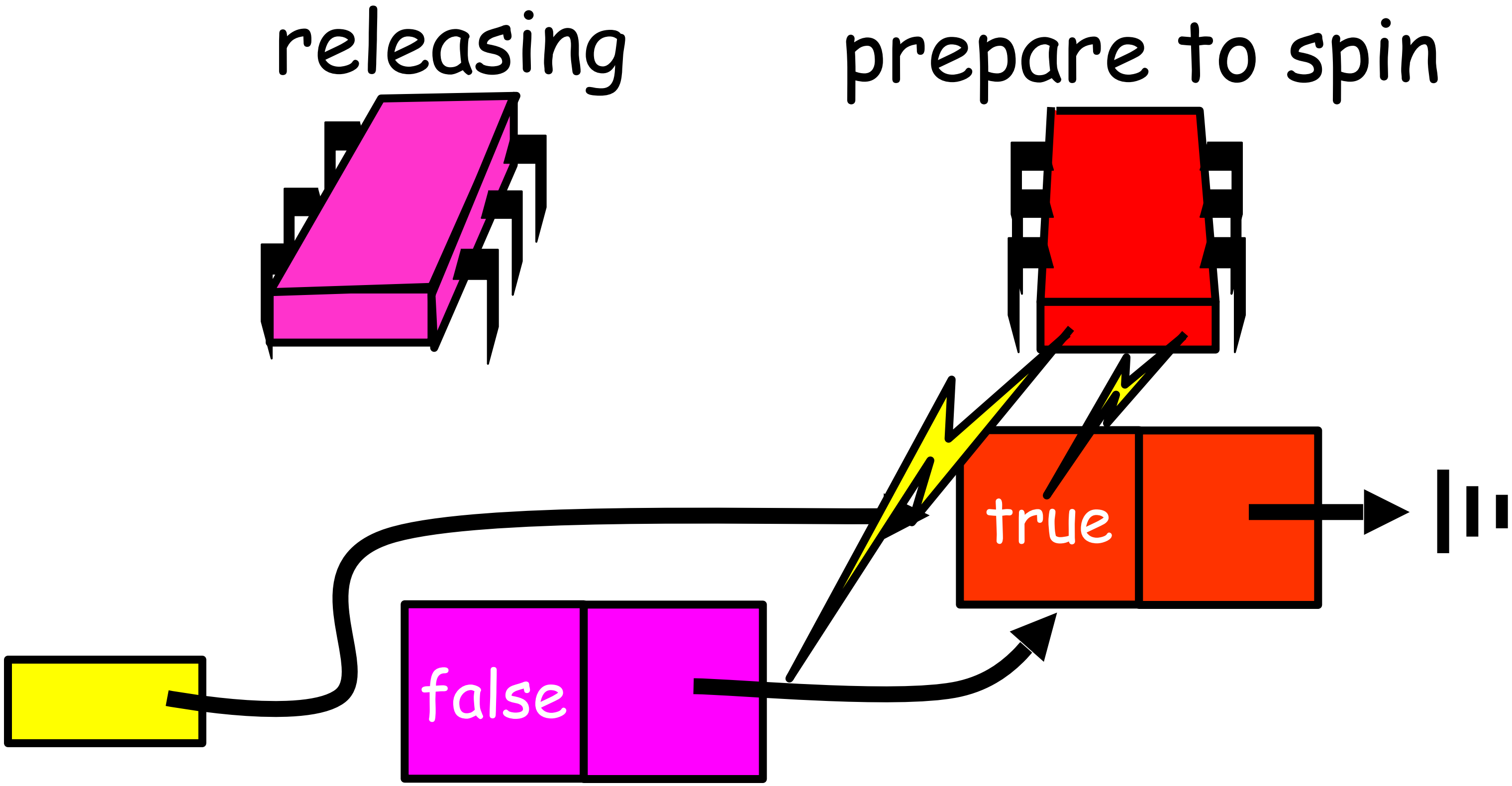
(2)

Purple Release

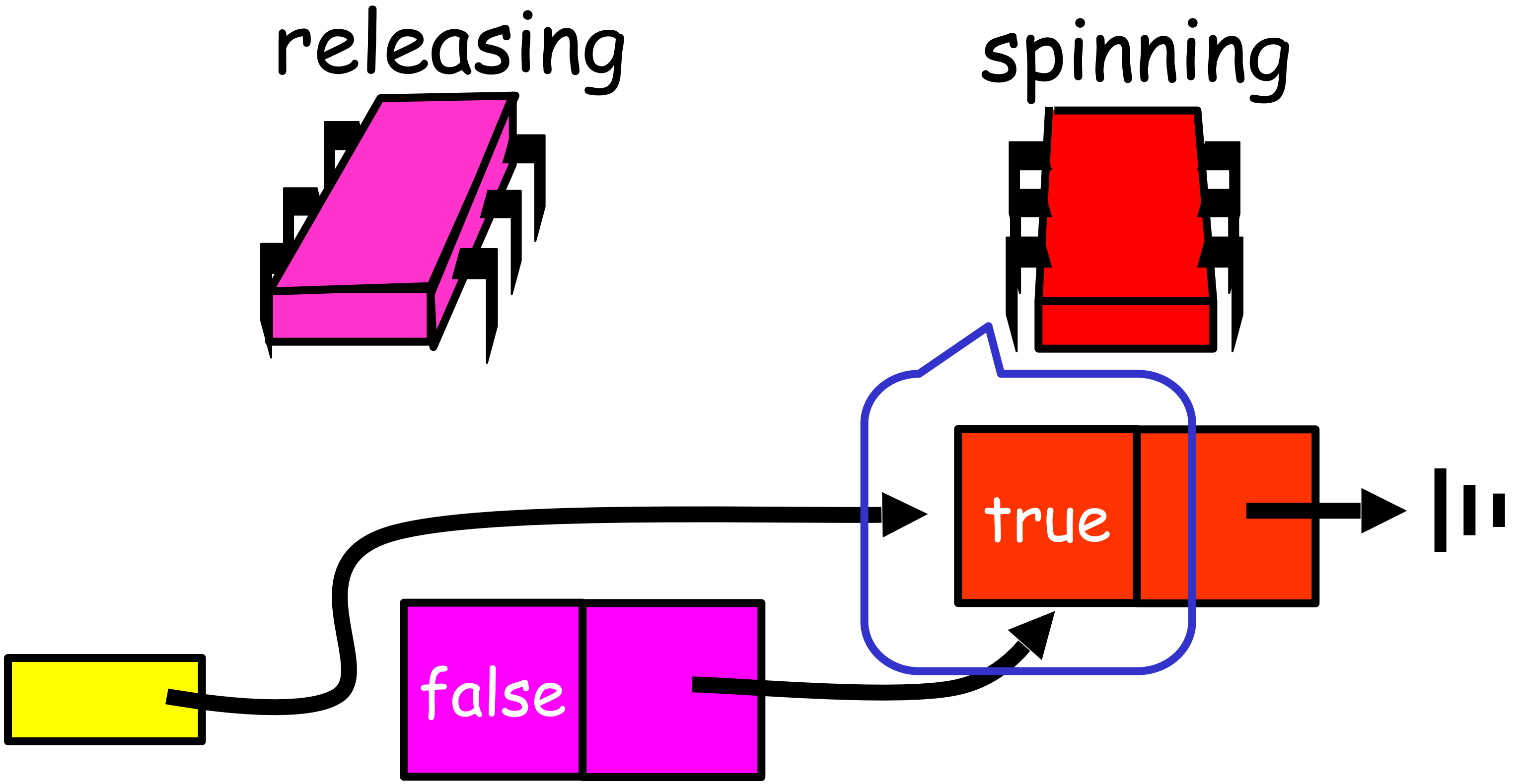


(2)

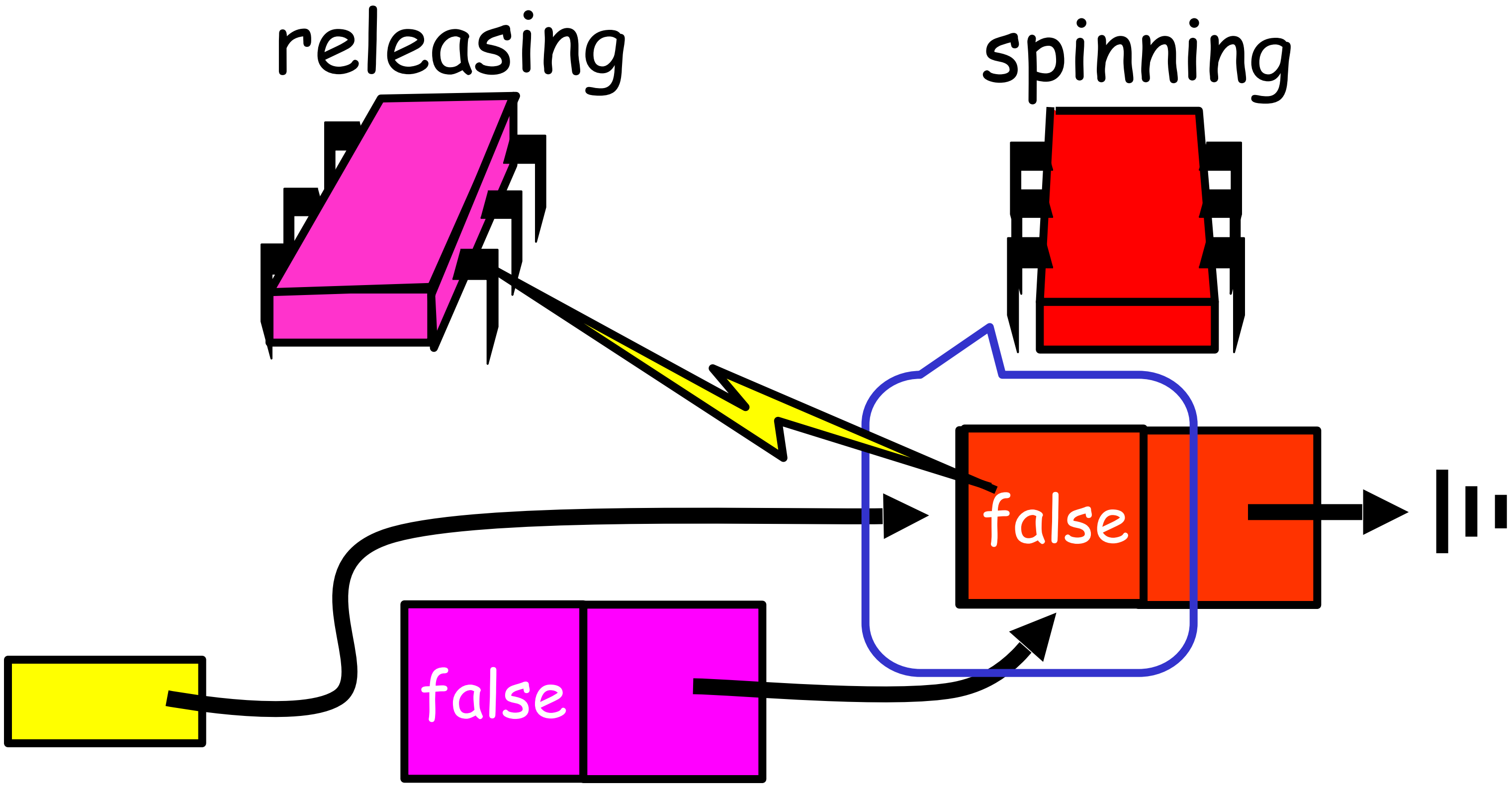
Purple Release



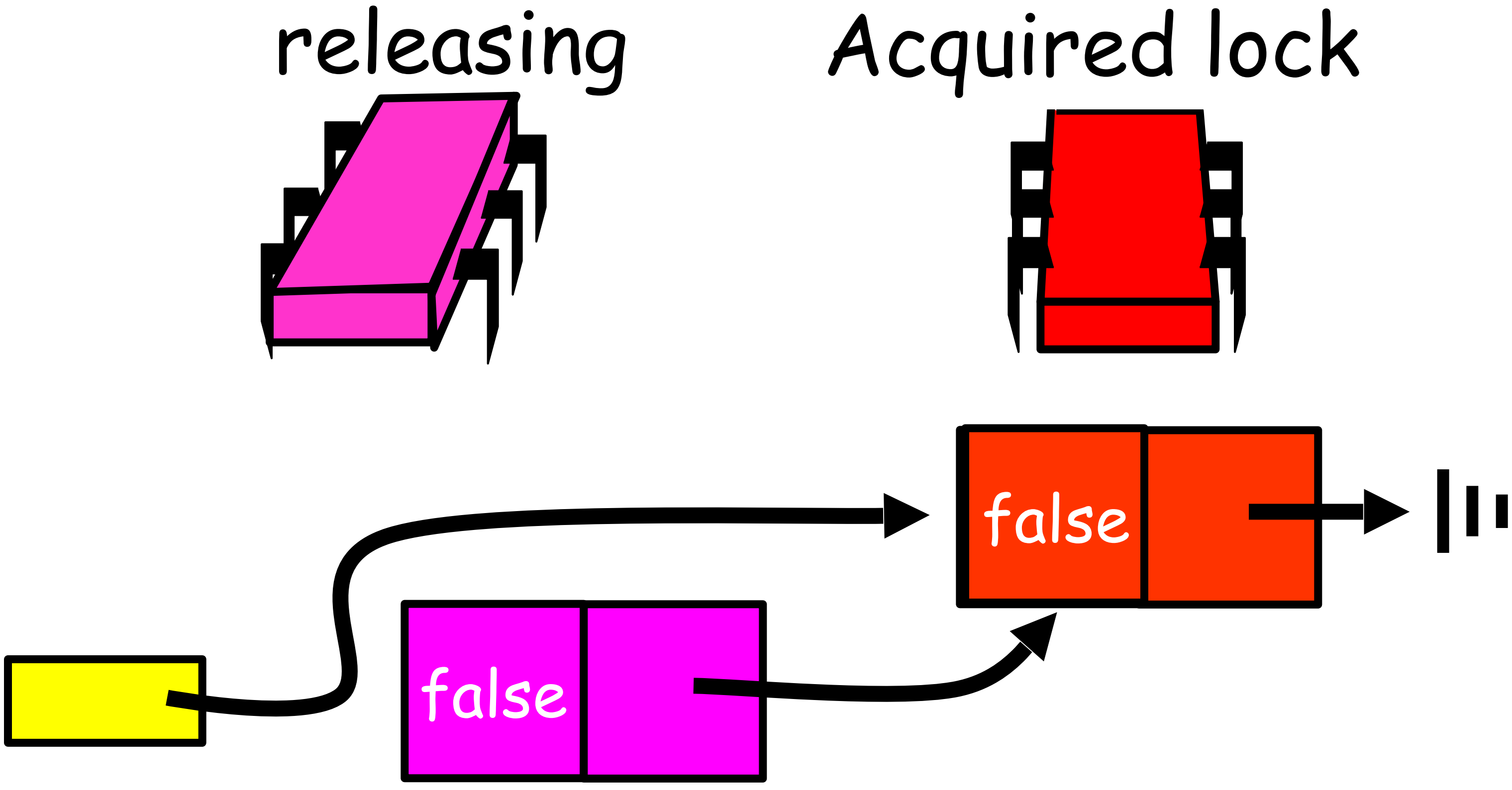
Purple Release



Purple Release



Purple Release



Abortable Locks

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

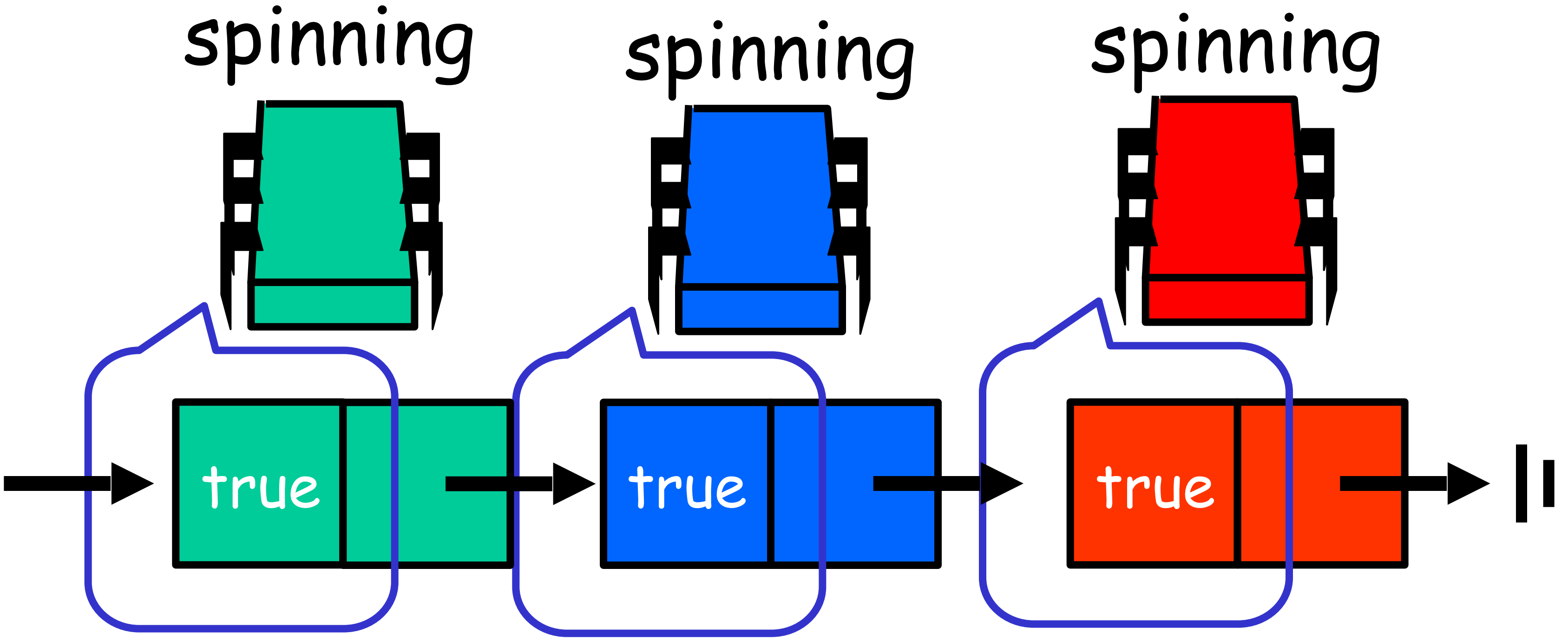
Back-off Lock

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

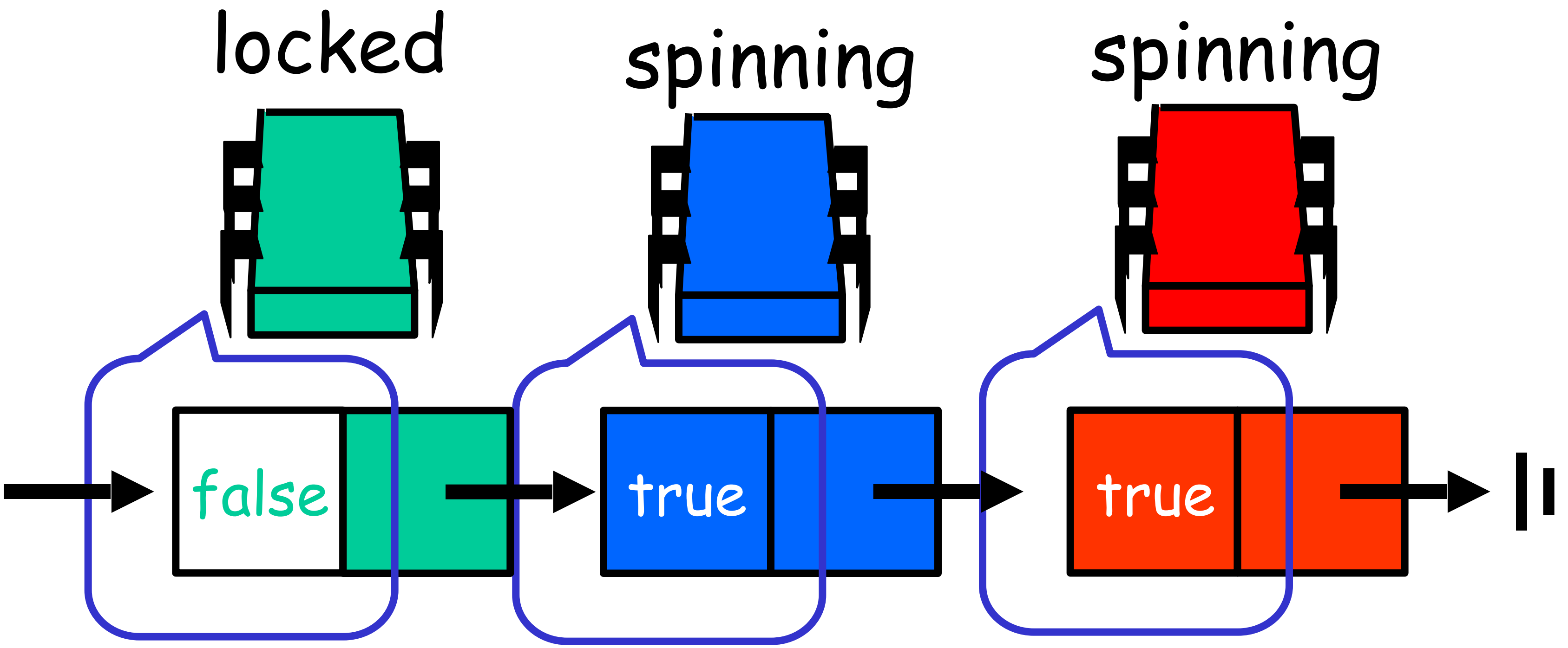
Queue Locks

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

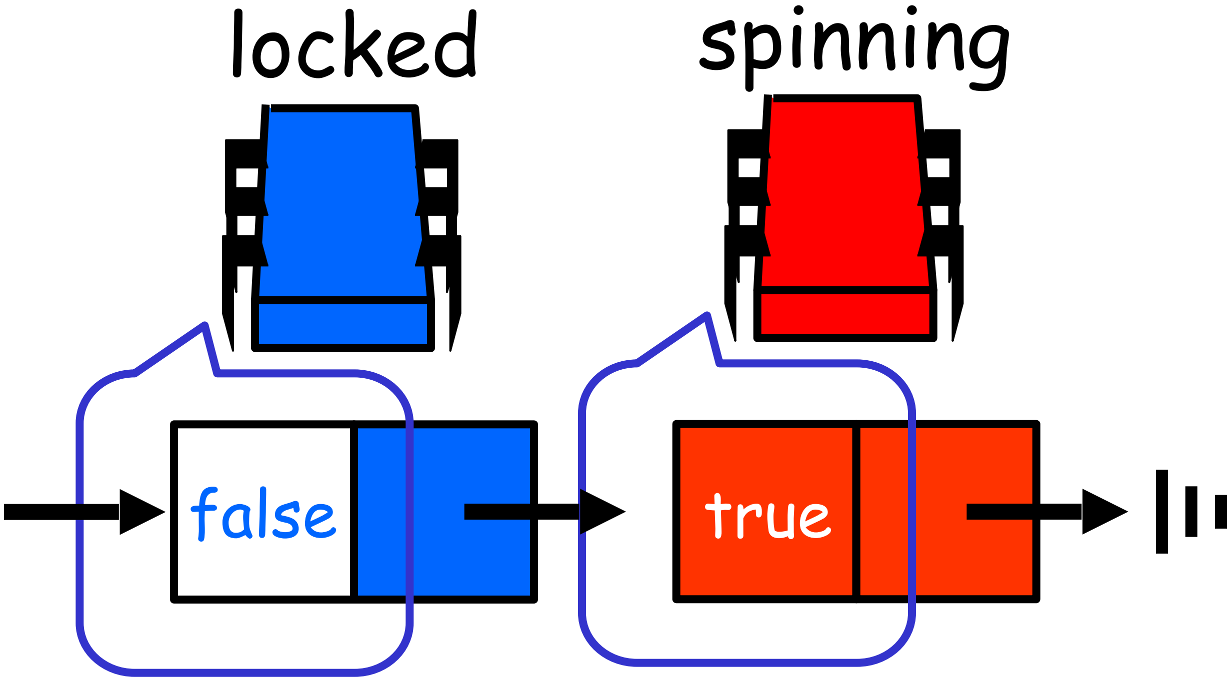
Queue Locks



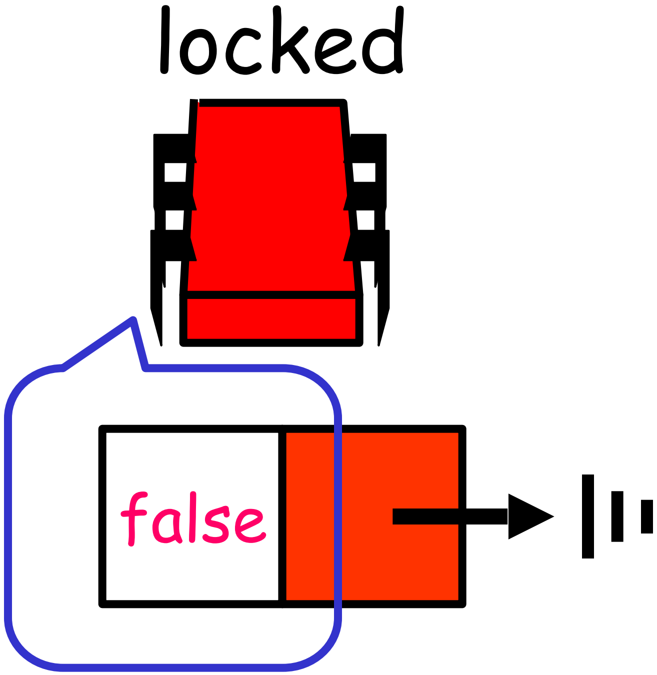
Queue Locks



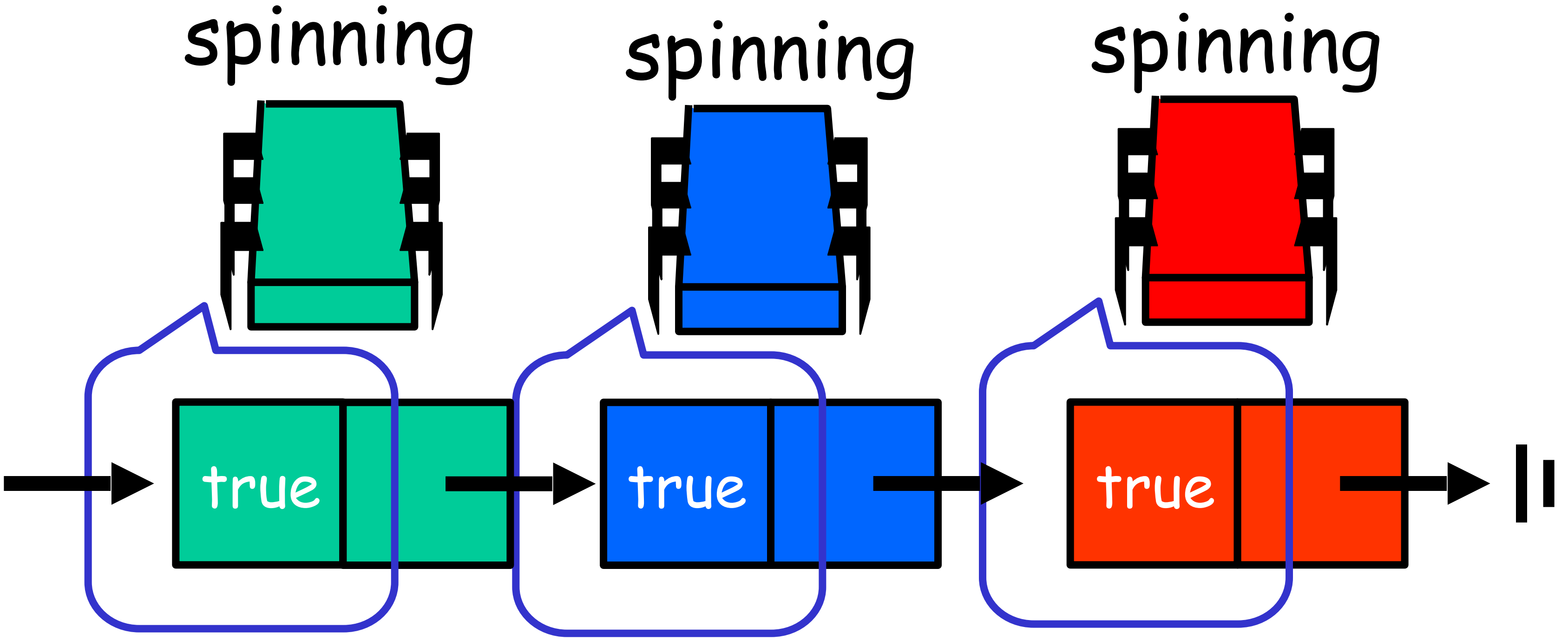
Queue Locks



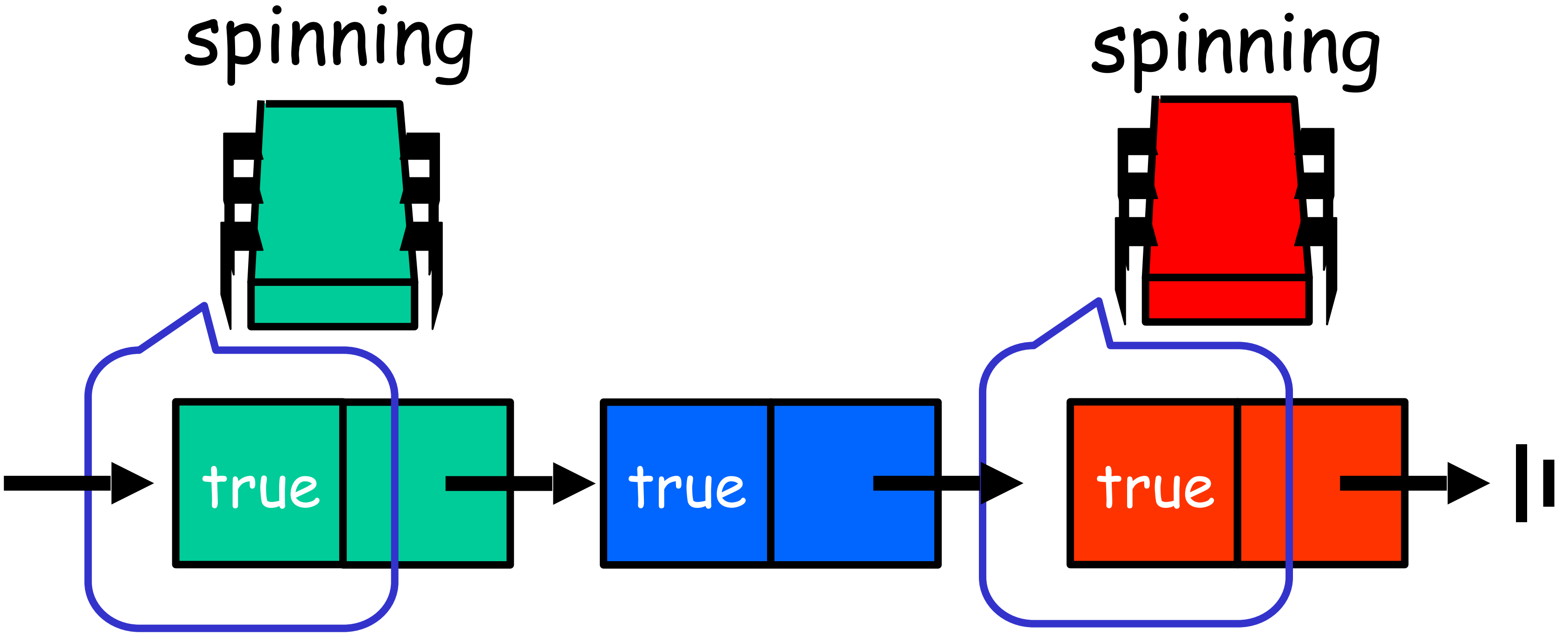
Queue Locks



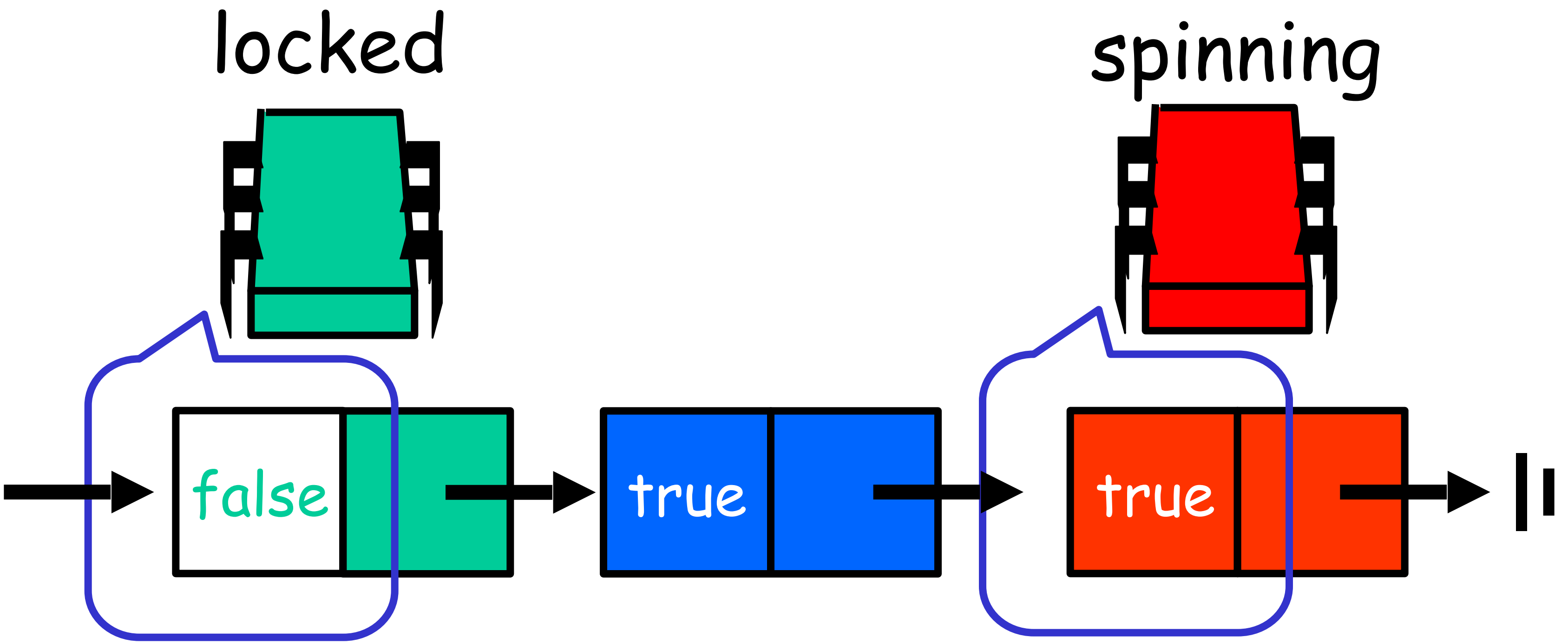
Queue Locks



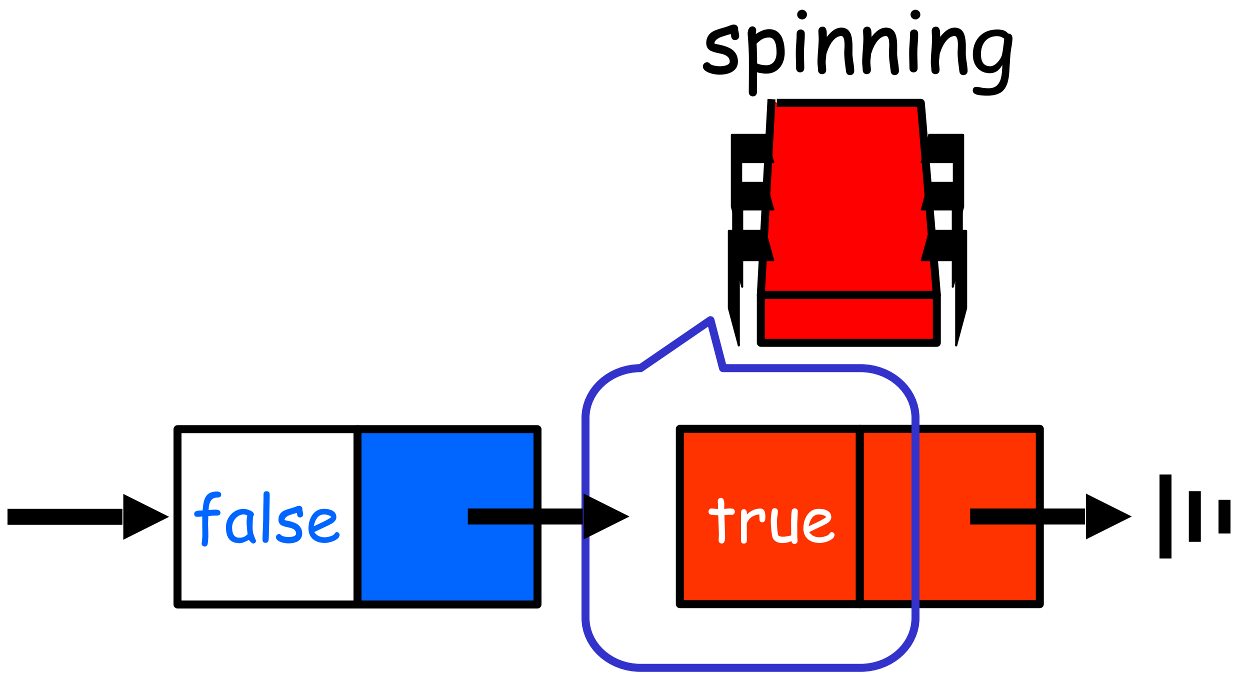
Queue Locks



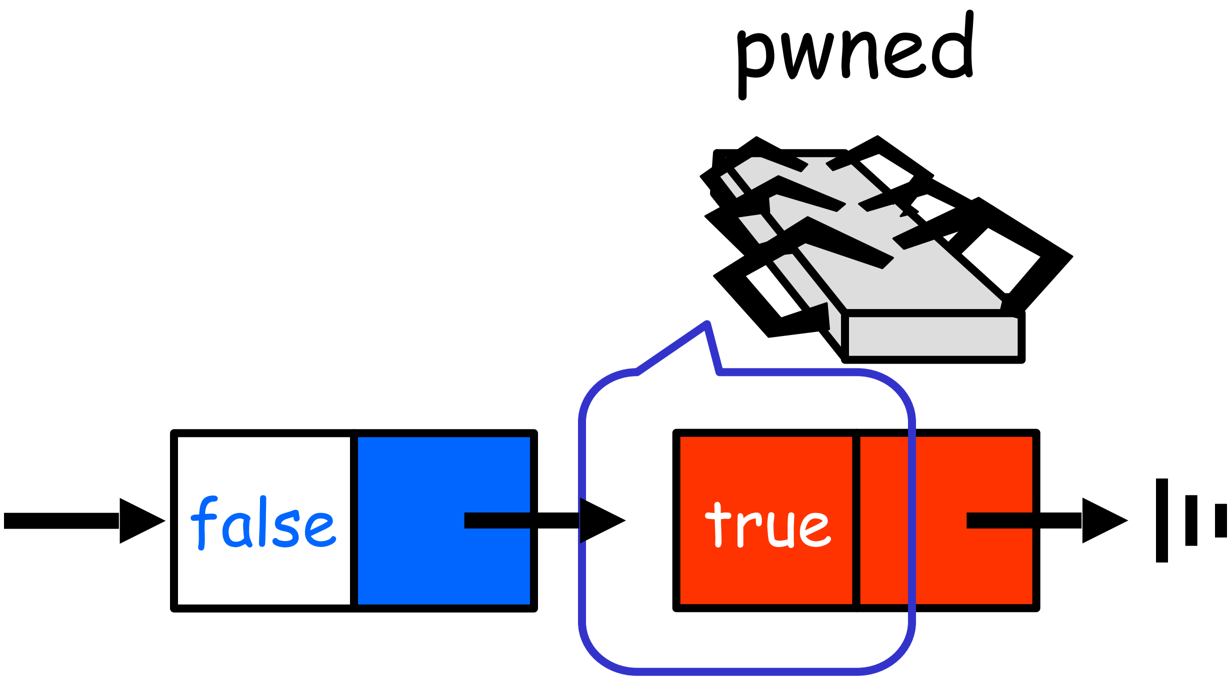
Queue Locks



Queue Locks



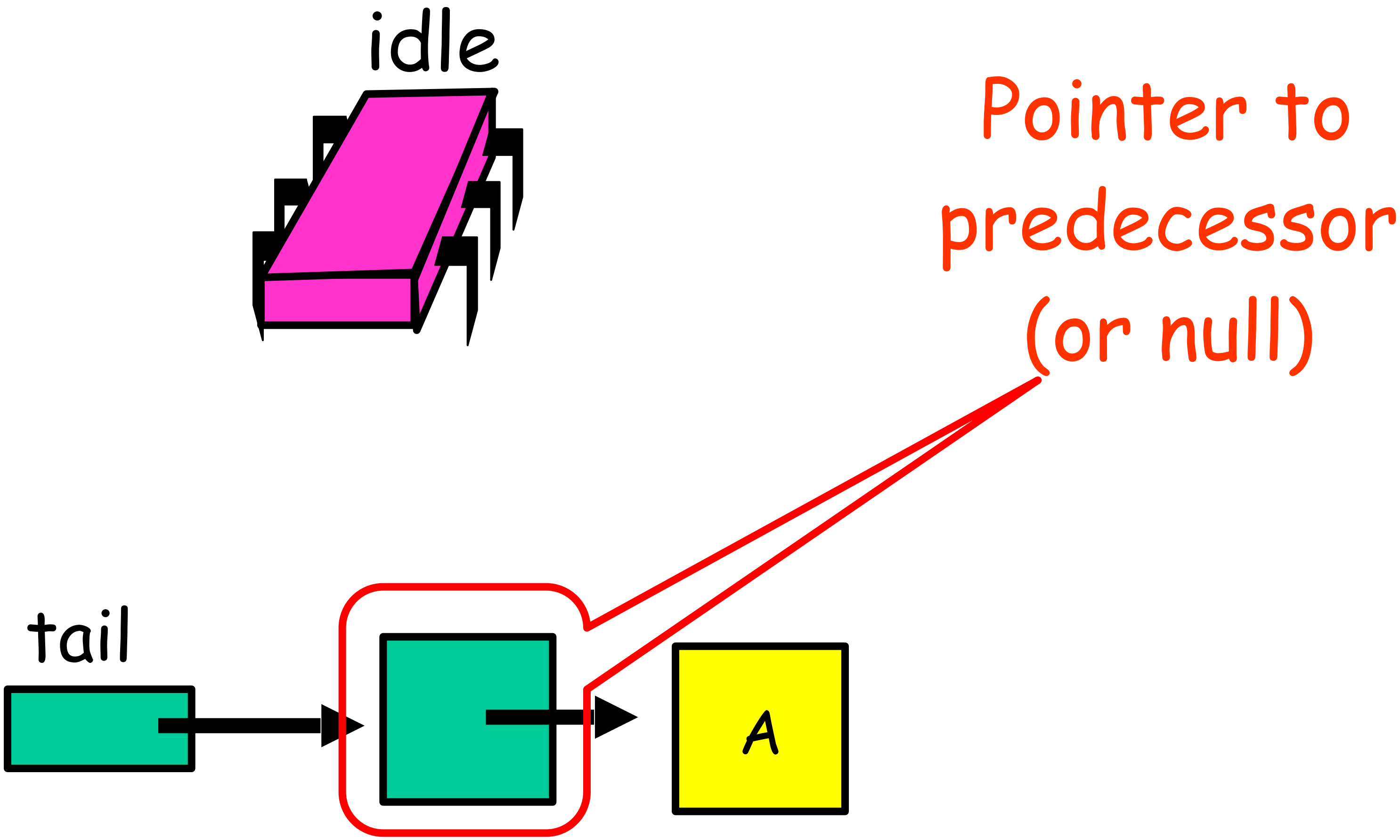
Queue Locks



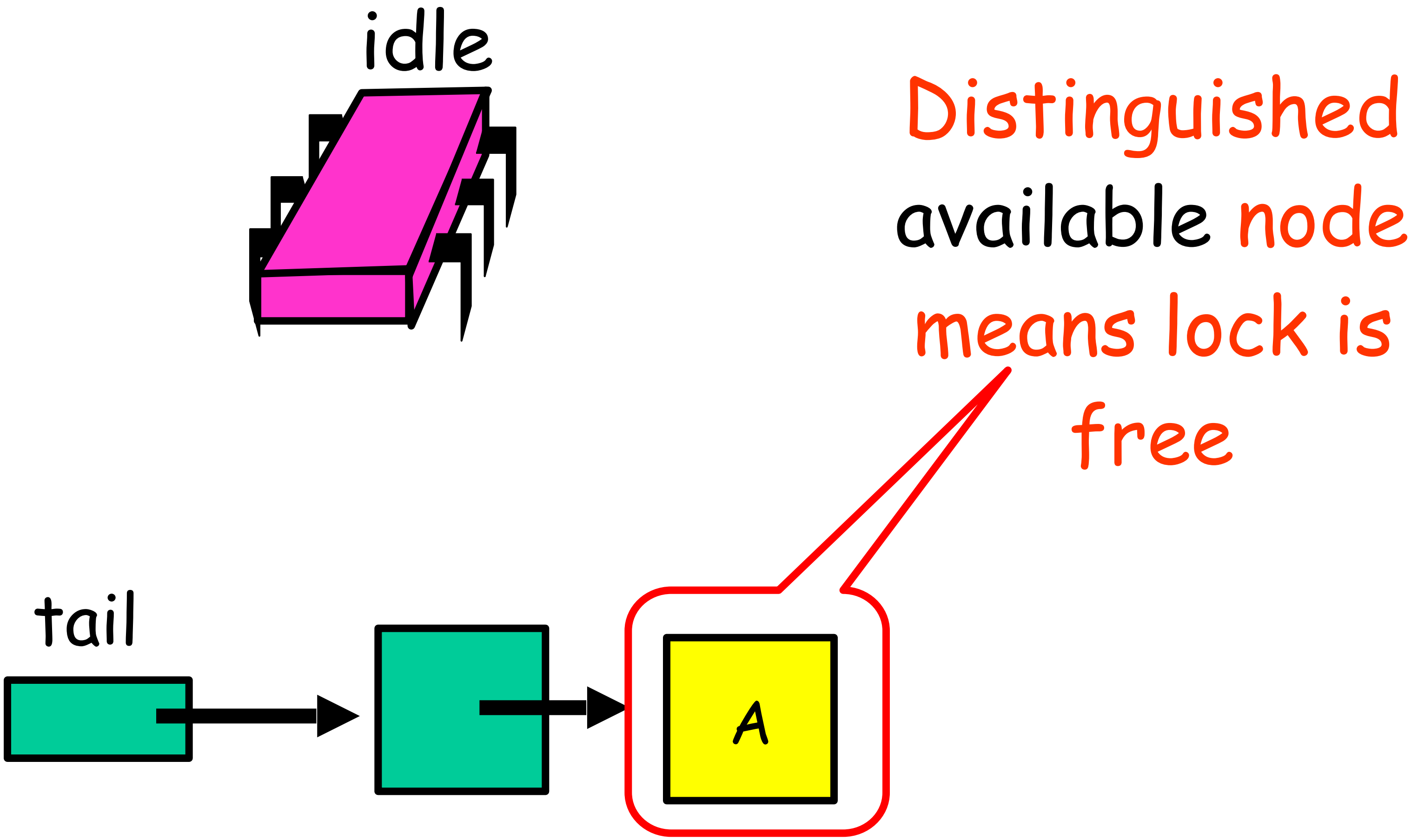
Abortable CLH Lock

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

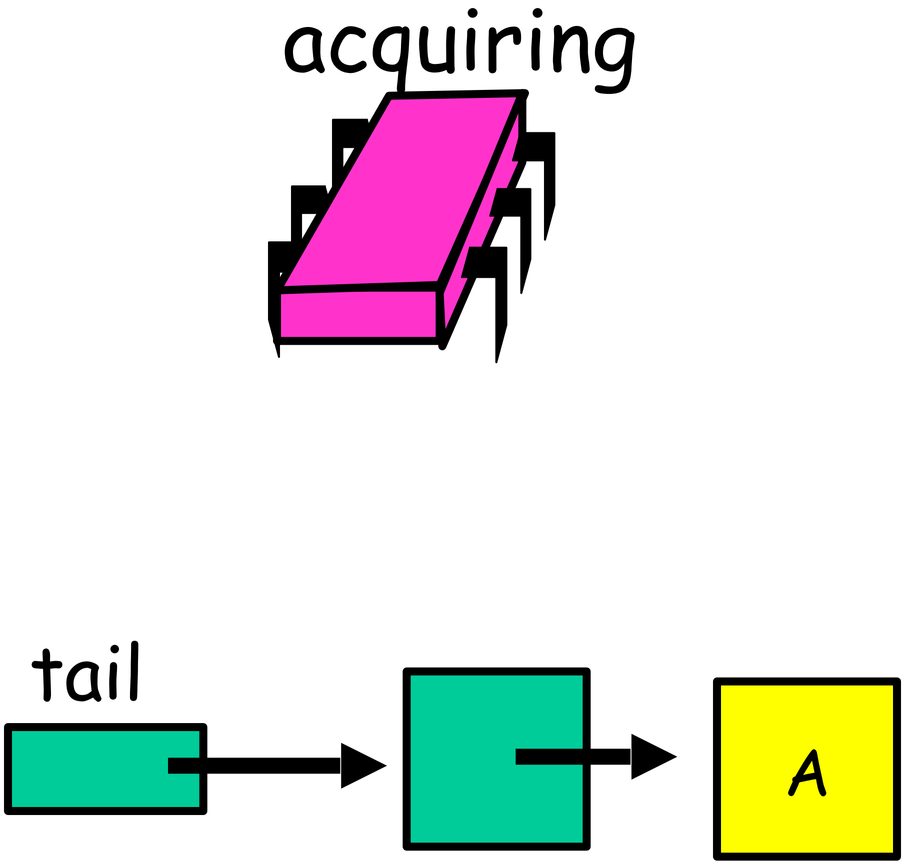
Initially



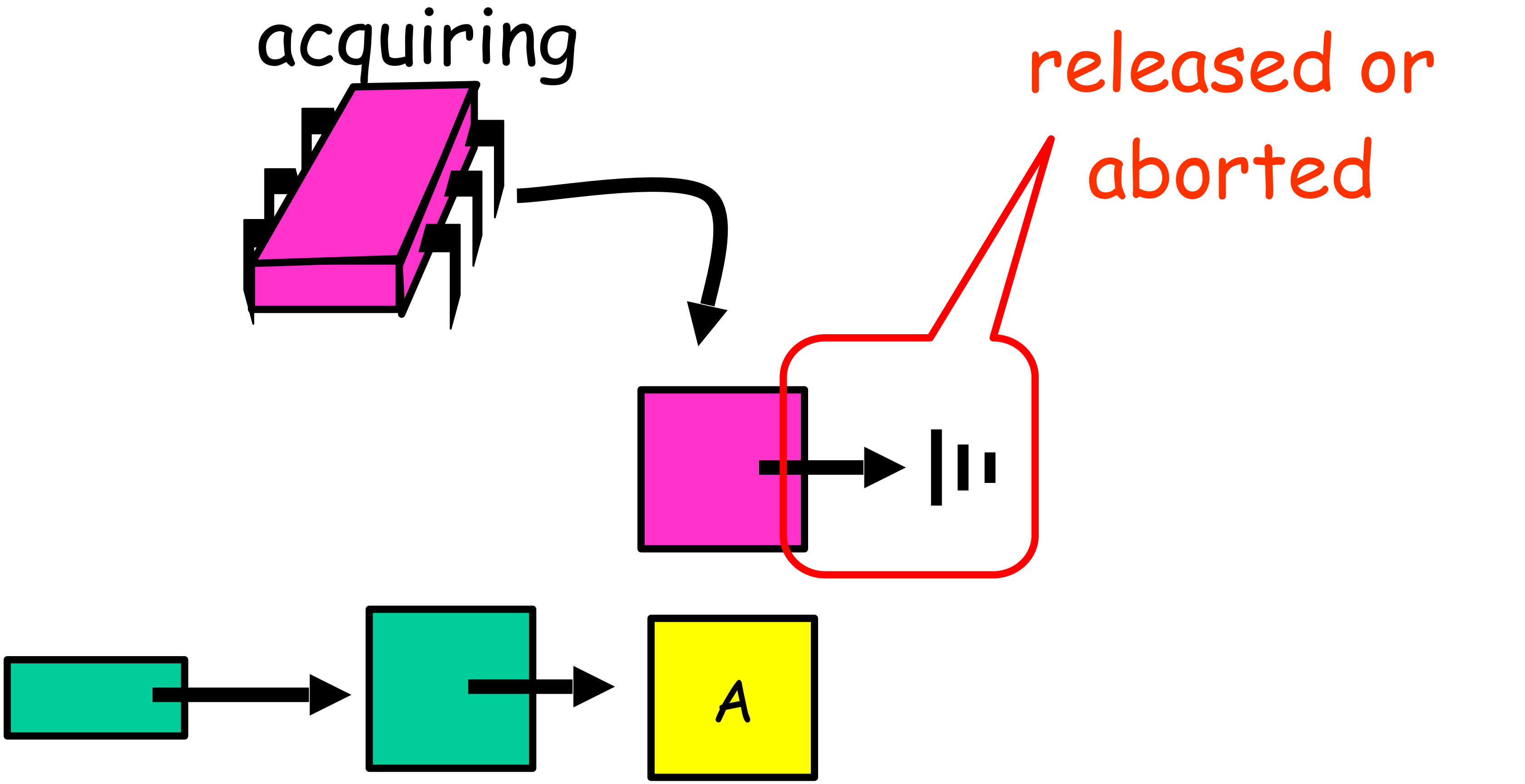
Initially


idle
Distinguished available node means lock is free
tail
A

Acquiring

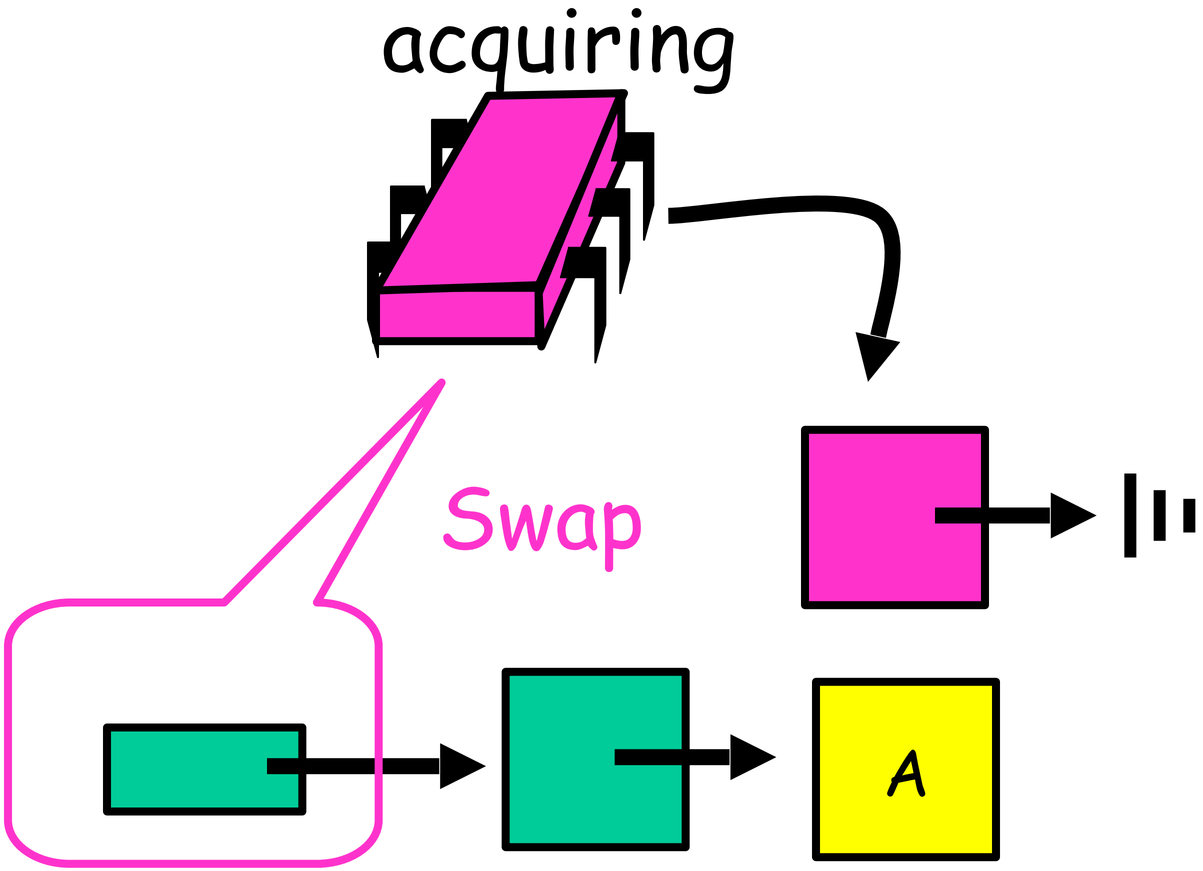


Acquiring

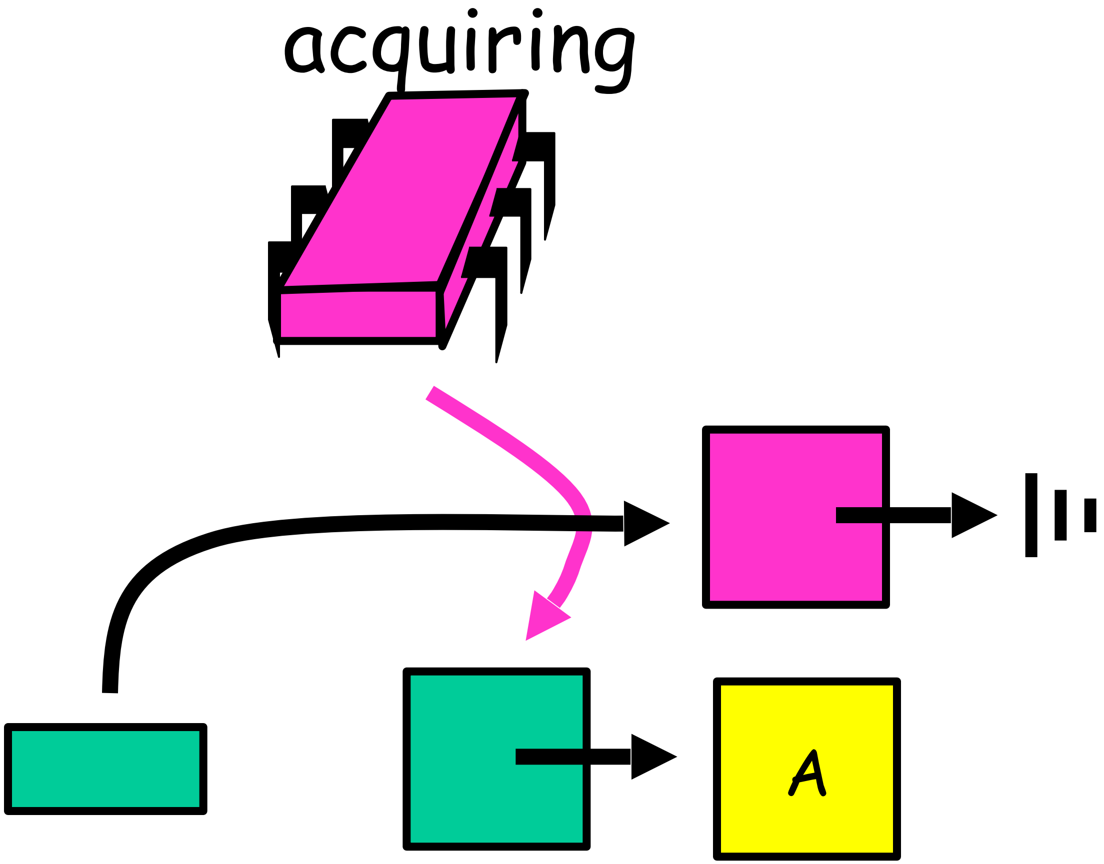


acquiring
Null predecessor means lock not released or aborted
A

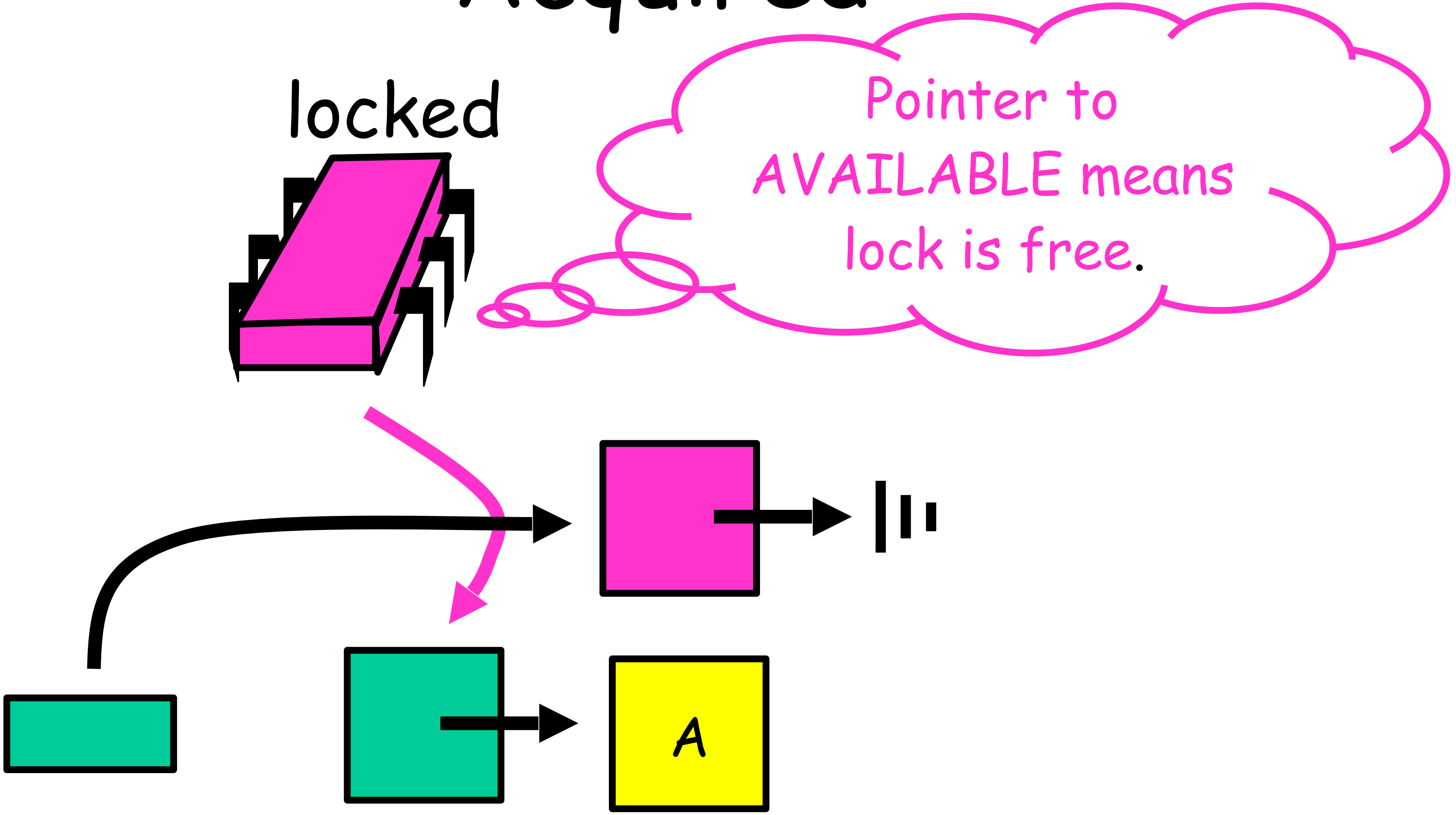
Acquiring



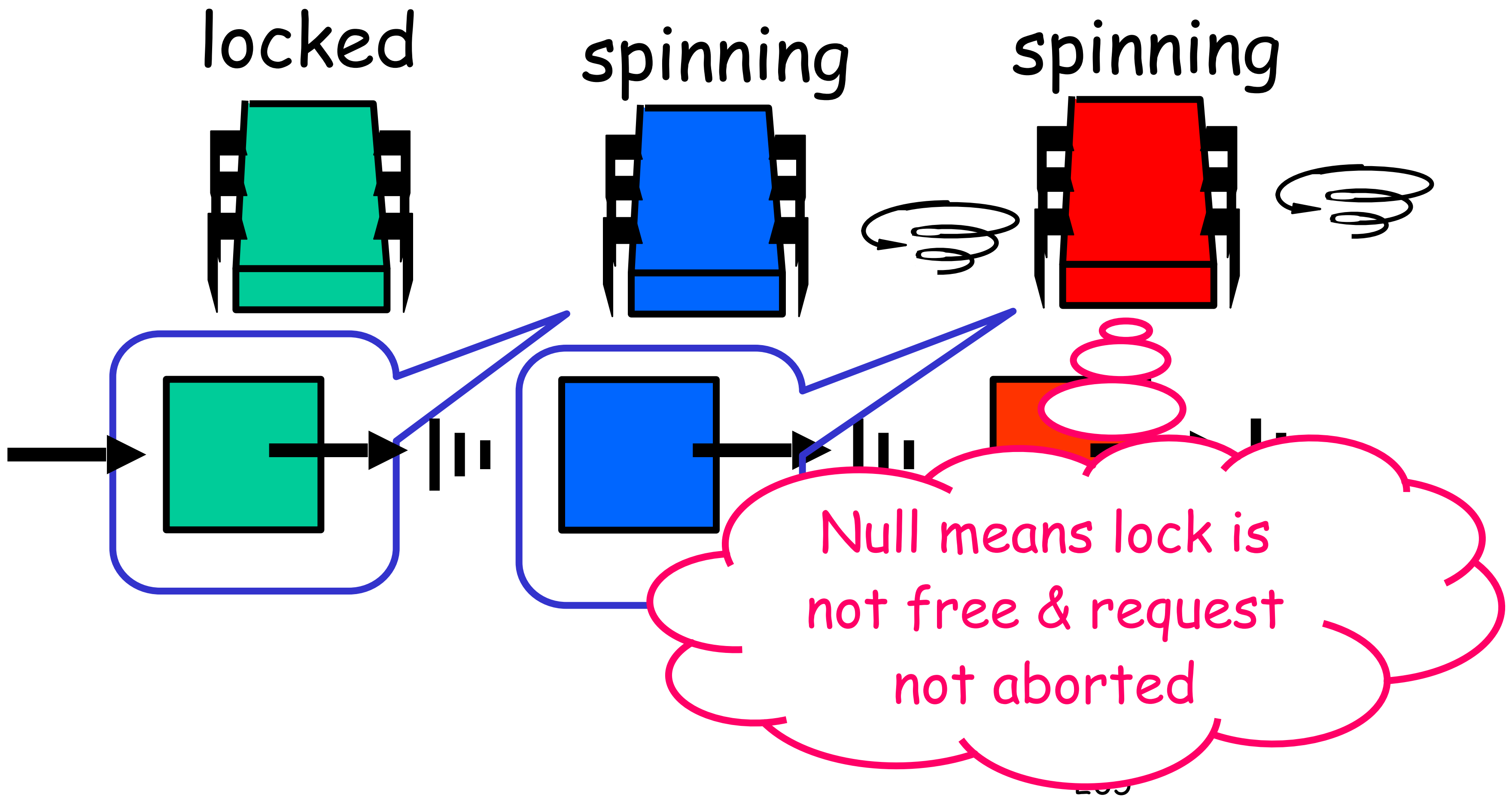
Acquiring



Acquired

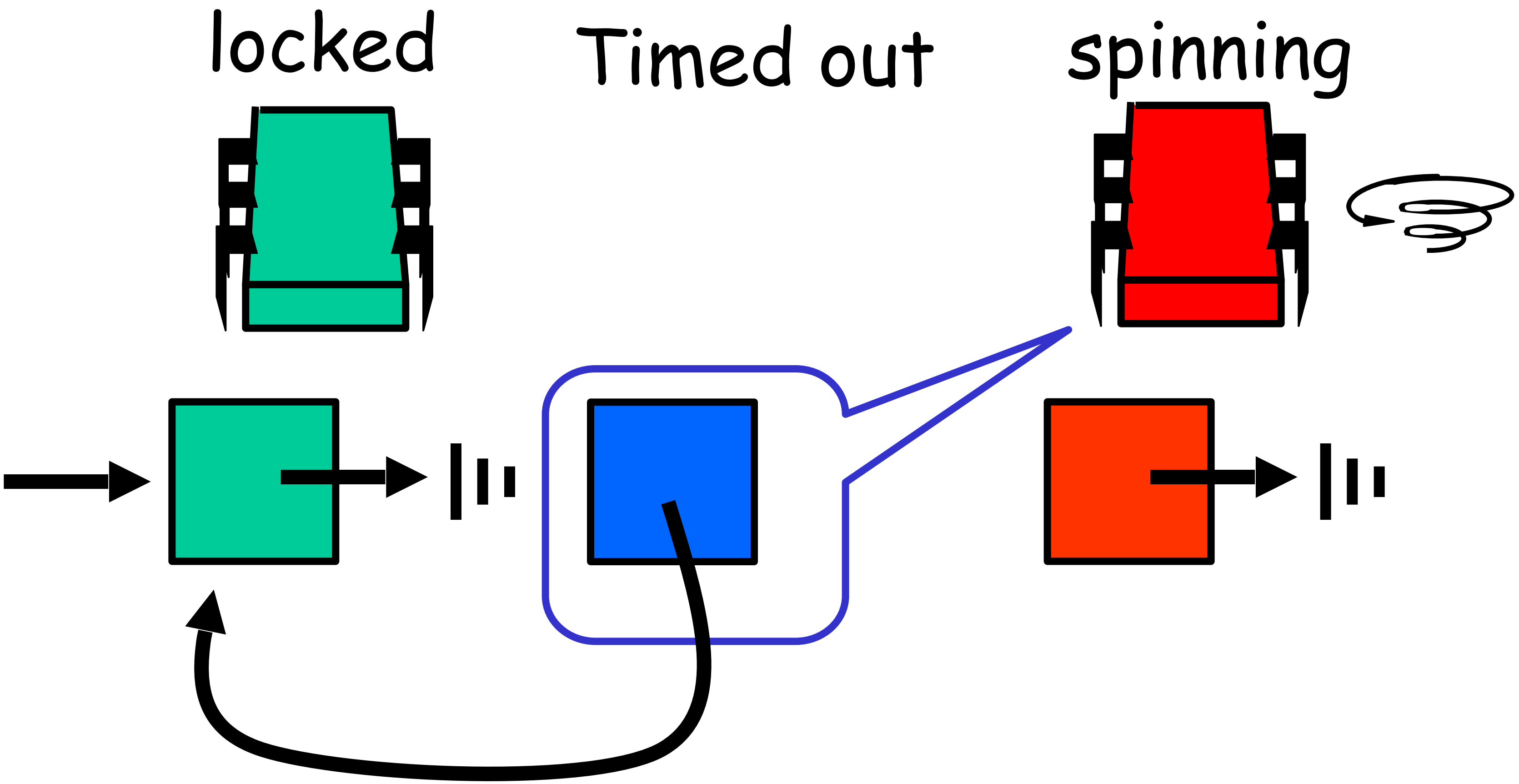


Normal Case

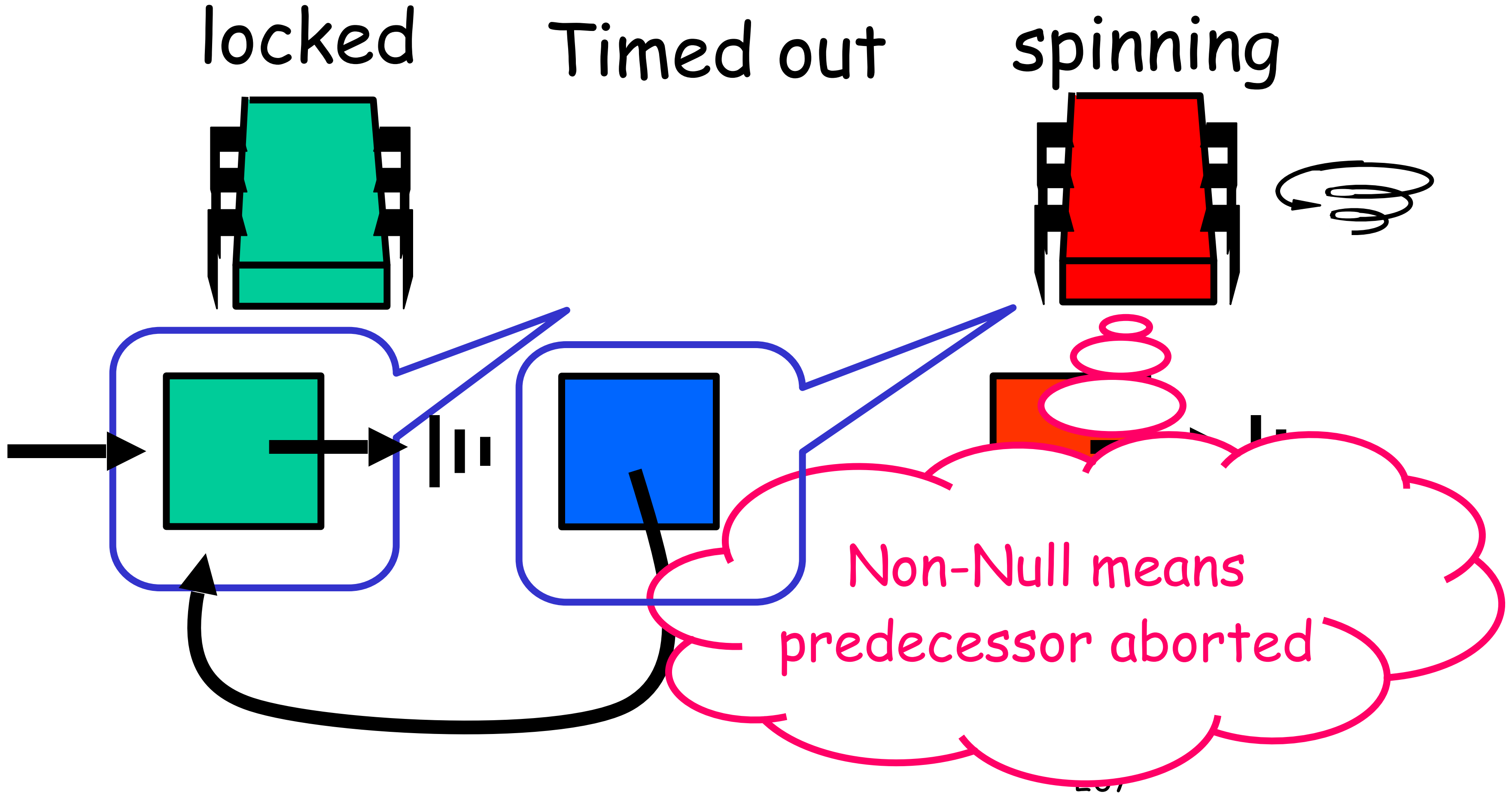


locked
spinning
spinning
Null means lock is not free & request not aborted

One Thread Aborts

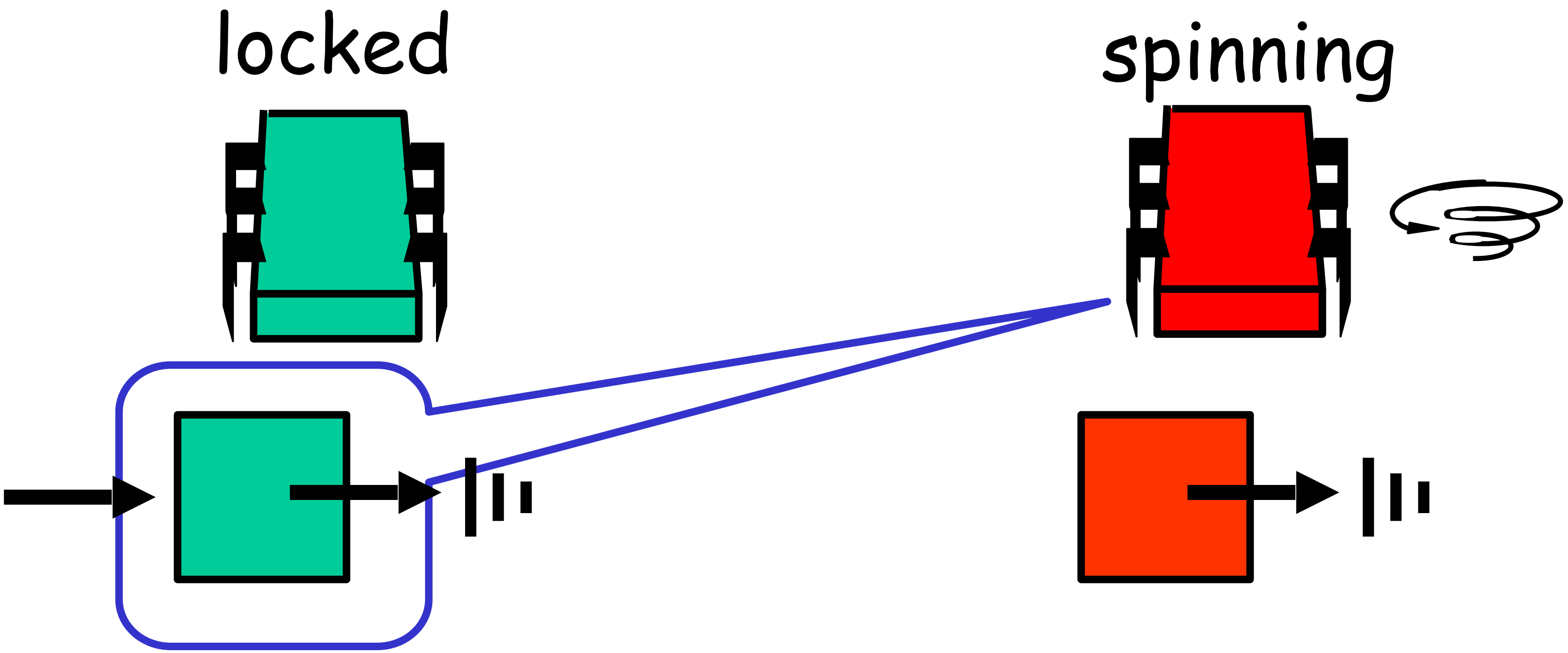


Successor Notices

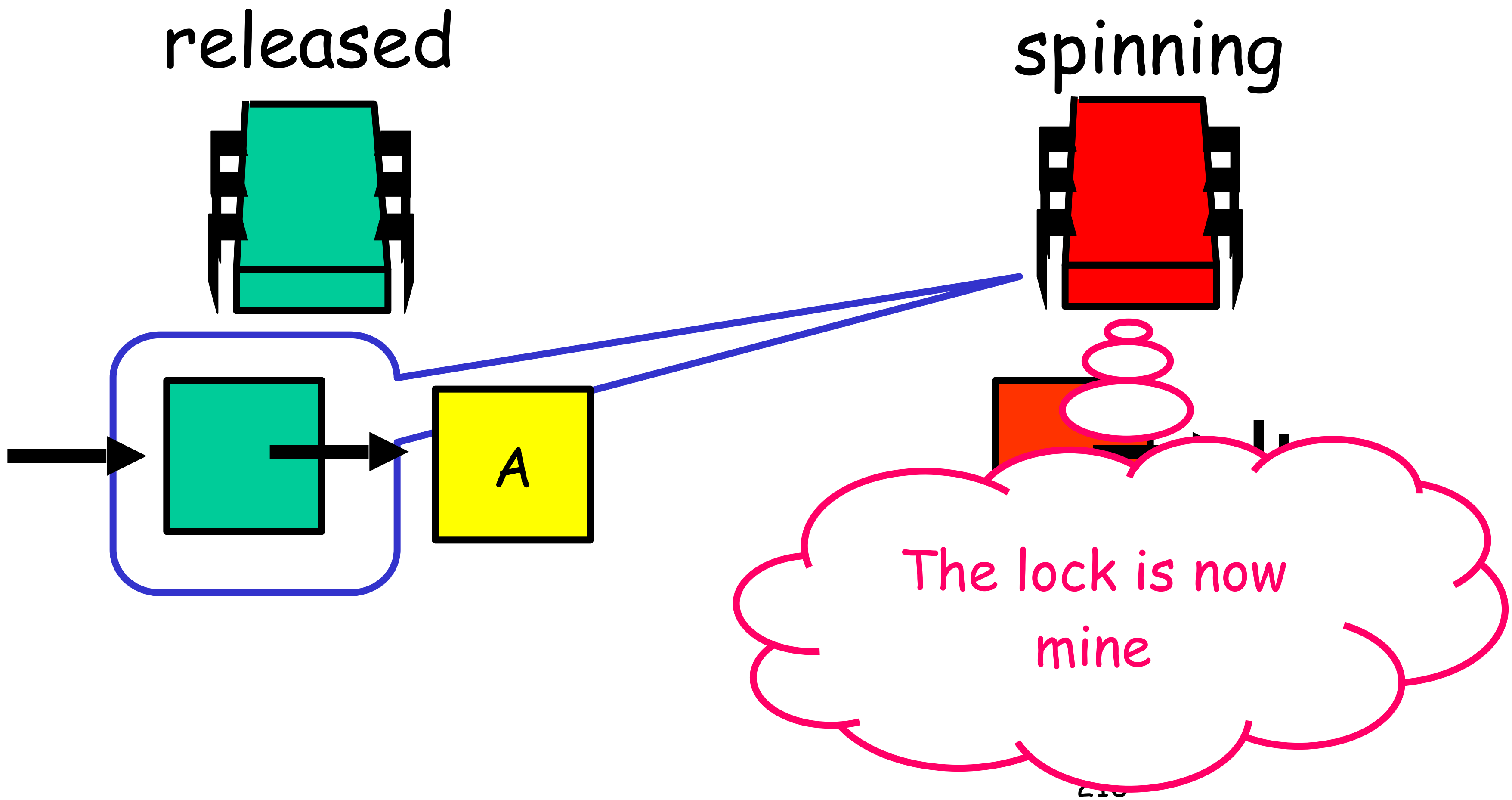


Recycle Predecessor's Node

Spin on Earlier Node



Spin on Earlier Node



Time-out Lock

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

Time-out Lock

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

Distinguished node to signify 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;
   }
```

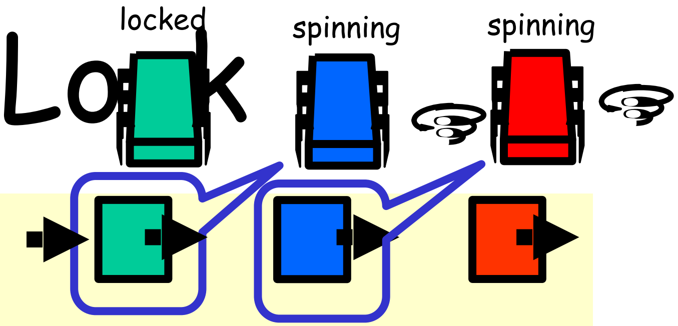
Swap with tail

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 have a successor, 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: exists successor, notify successor 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

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