Concurrency:
Multi-core Programming & Data Processing

Spin Locks and Contention Management

Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/

Institut d'informatique
Université de Neuchâtel

*Based on slides by **Maurice Herlihy** and **Nir Shavit***

Focus so Far: Correctness

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

New Focus: Performance

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

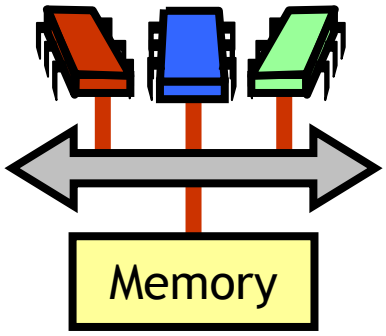
Kinds of Architectures

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

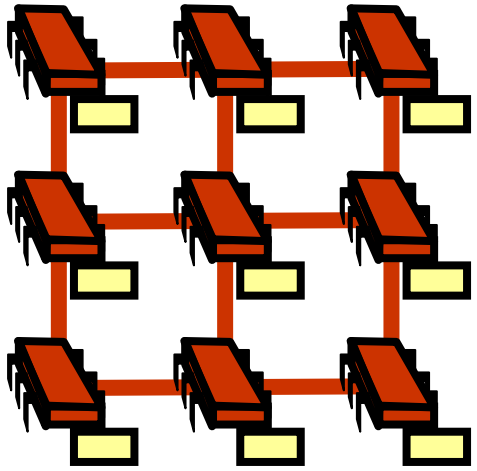
Our space

MIMD Architectures

- Memory contention
- Communication contention
- Communication latency



Shared Bus



Distributed

Today: Revisit Mutual Exclusion

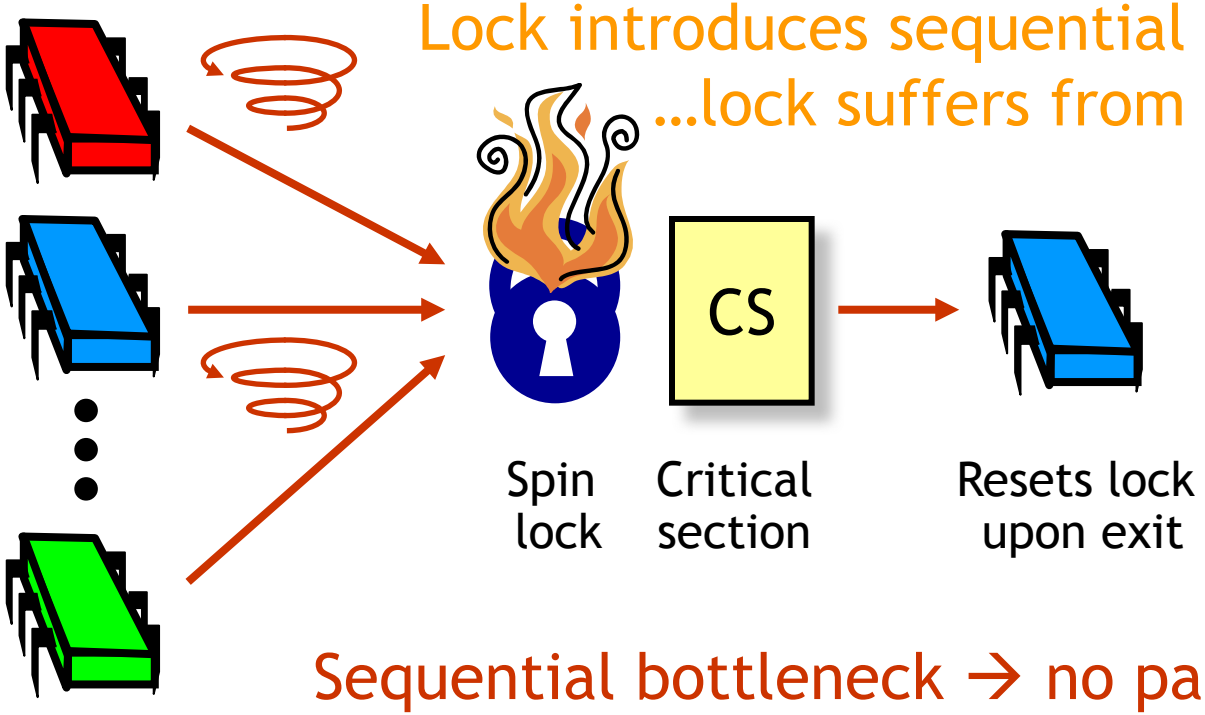
- Think of performance, not just correctness and progress
- Begin to understand how performance depends on our software properly utilizing the multiprocessor machines hardware
- And get to know a collection of locking algorithms…

What Should you Do If you Cannot Get a Lock?

Our focus

- Keep trying?
  - "Spin" or "busy-wait"
    (like filter and bakery algorithms)
  - Good if delays are short
- Give up the processor?
  - Good if delays are long
  - Always good on uniprocessor

Basic Spin-Lock



Sequential bottleneck → no parallelism

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 a.k.a. “getAndSet”

Review: Test-and-Set

```
public class AtomicBoolean {
  boolean value;

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

Package java.util.concurrent.atomic

Swap old and new value

Review: Test-and-Set

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

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

Test-and-Set Lock

- 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 implements Lock {
  AtomicBoolean state =
    new AtomicBoolean(false);

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

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

Lock state is an AtomicBoolean

Keep trying until lock acquired

Release lock by resetting state to false

Space Complexity

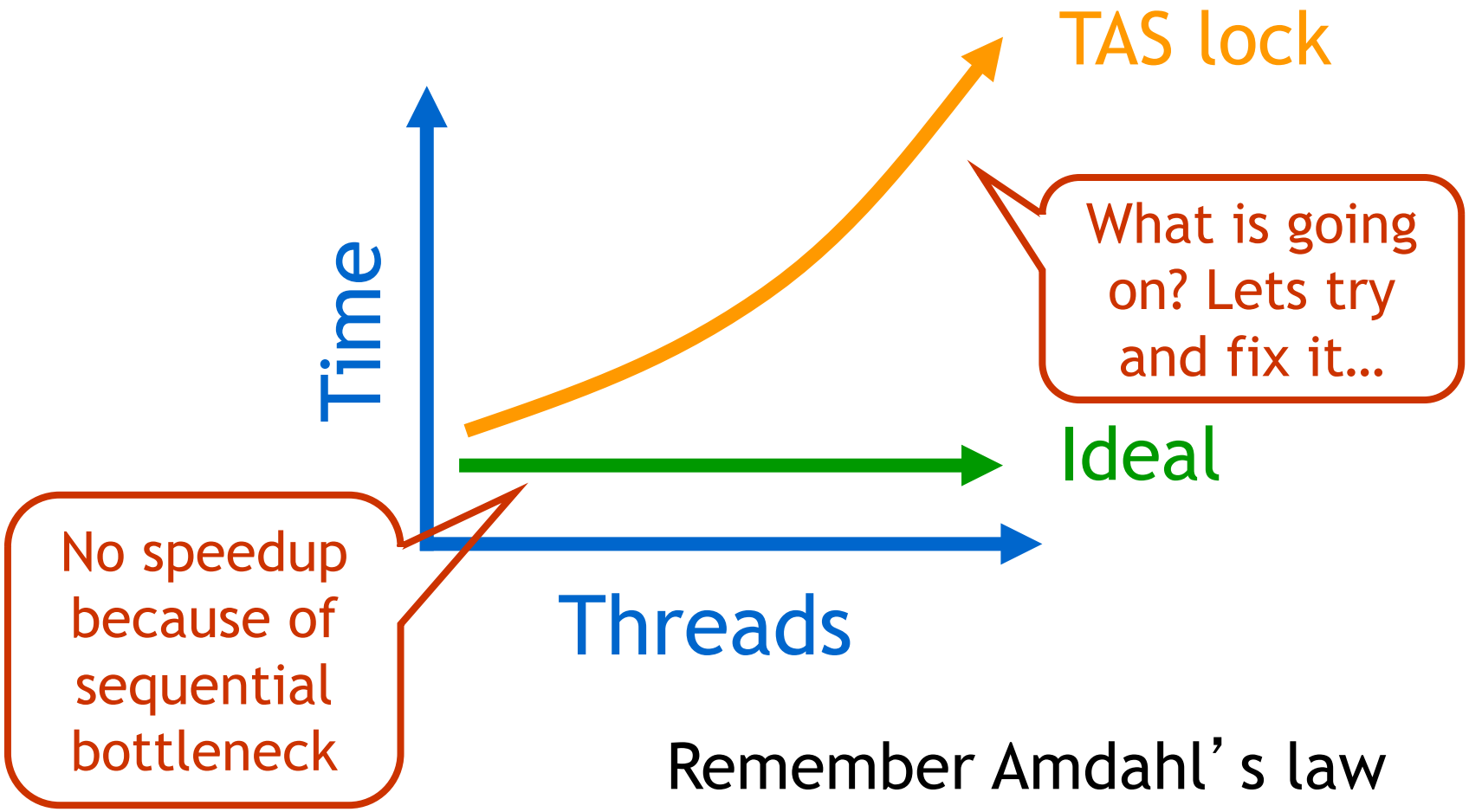
- TAS spin-lock has small “footprint”
- A spin-lock with **n** threads uses **O(1)** space
- As opposed to **O(n)** Peterson/Bakery

- How did we overcome the $\Omega(n)$ lower bound?
- We used a RMW operation…

Performance

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

Mystery #1



Remember Amdahl's law

Test-and-Test-and-Set Lock

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

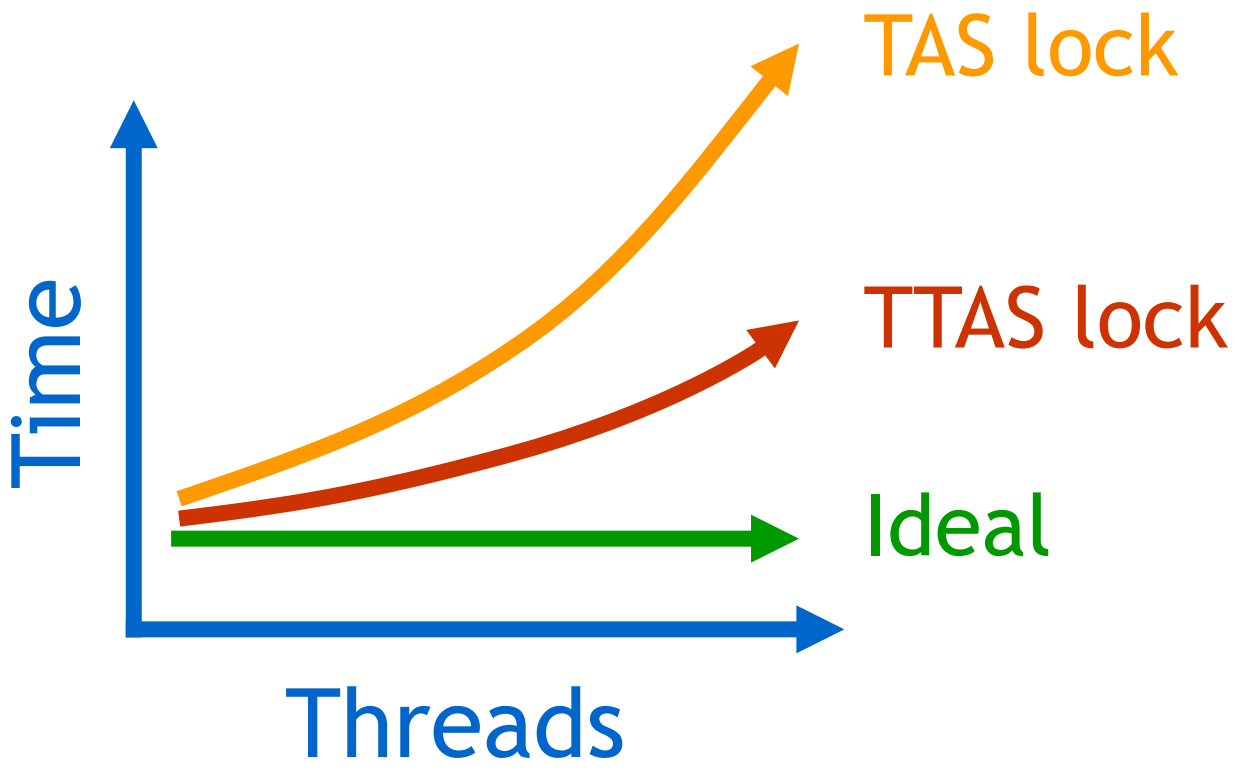
Test-and-Test-and-Set Lock

```
class TTASLock implements Lock {
  AtomicBoolean state =
    new AtomicBoolean(false);
  void lock() {
    while (true) {
      while (state.get()) {}
      if (!state.getAndSet(true))
        return;
    }
  }
}
```

Wait until lock looks free

Then try to acquire it

Mystery #2



Mystery

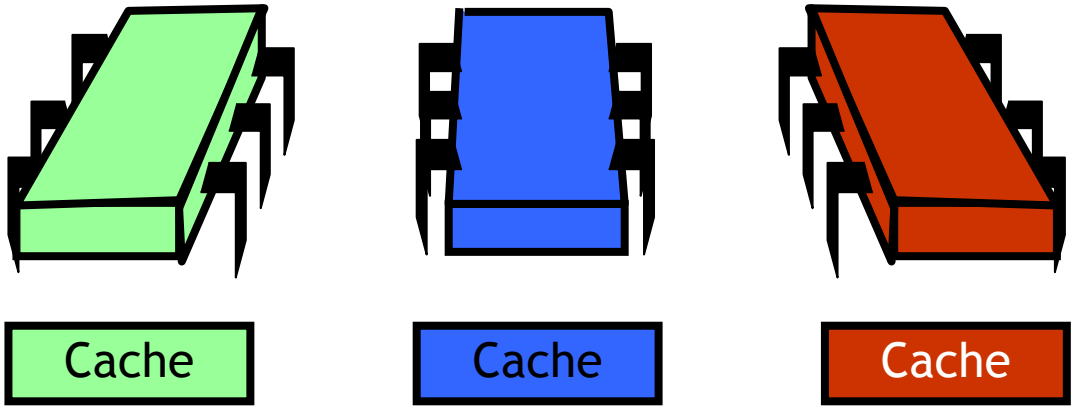
- Both
  - TAS and TTAS
  - Do the same thing (in our model)
- Except that
  - TTAS performs much better than TAS
  - Neither approaches ideal

Opinion

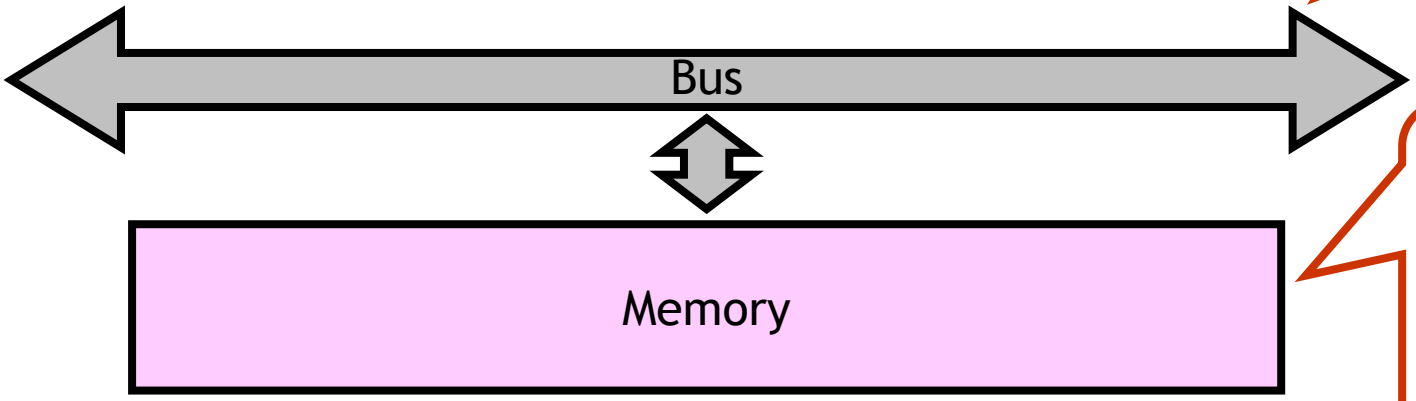
- Our memory abstraction is broken
- TAS & TTAS methods
  - Are provably the same (in our model)
  - Except they are not (in field tests)
- Need a more detailed model…

Bus-Based Architectures

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



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



Random access memory (100s of cycles)

Jargon Watch

- Cache hit
  - "I found what I wanted in my cache": good thing
- Cache miss
  - "I had to go all the way to memory for that data": bad thing
- Warning: this model is still a simplification
  - But not in any essential way
  - Illustrates basic principles
  - Will discuss complexities later

Mutual Exclusion

- What do we want to optimize?
  - Bus bandwidth used by spinning threads?
  - Release/acquire latency?
  - Acquire latency for idle lock?
- Different optimizations do not necessarily require the same algorithms

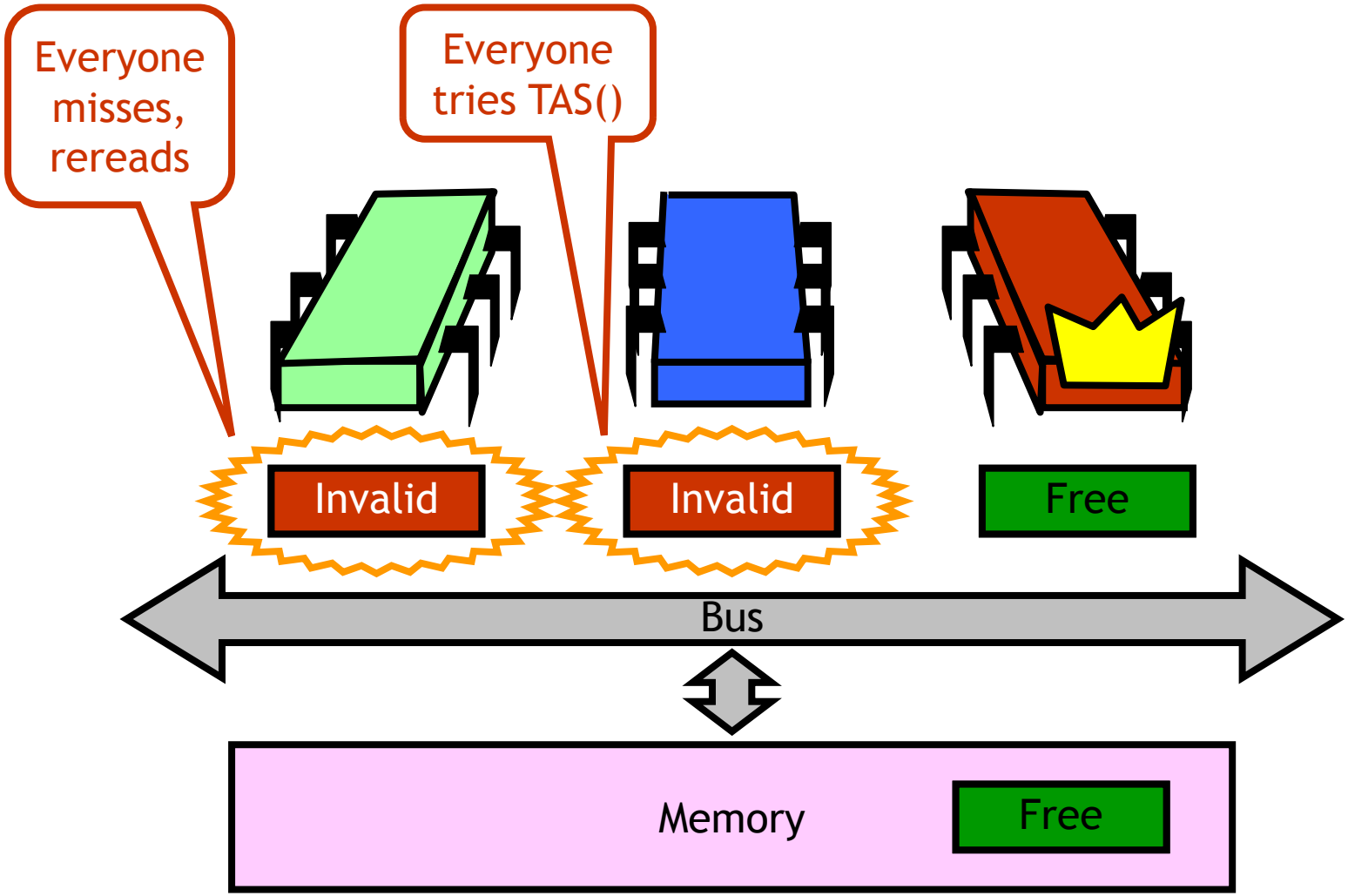
Simple TASLock

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

TTASLock

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

Local Spinning and Lock Release



Problems

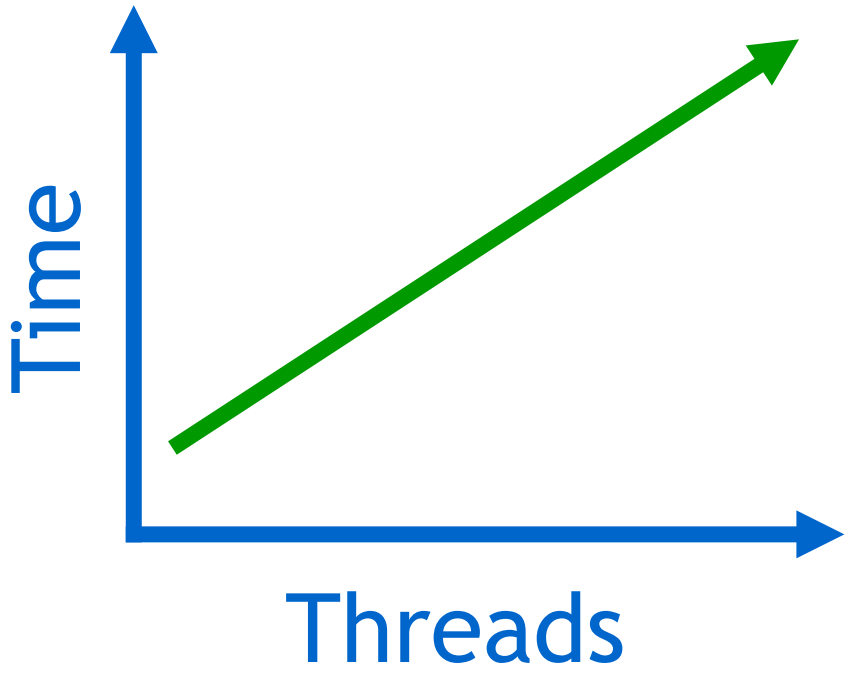
- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?

Measuring Quiescence Time

- X: time of ops that do not use the bus
- Y: time of ops that cause intensive bus traffic

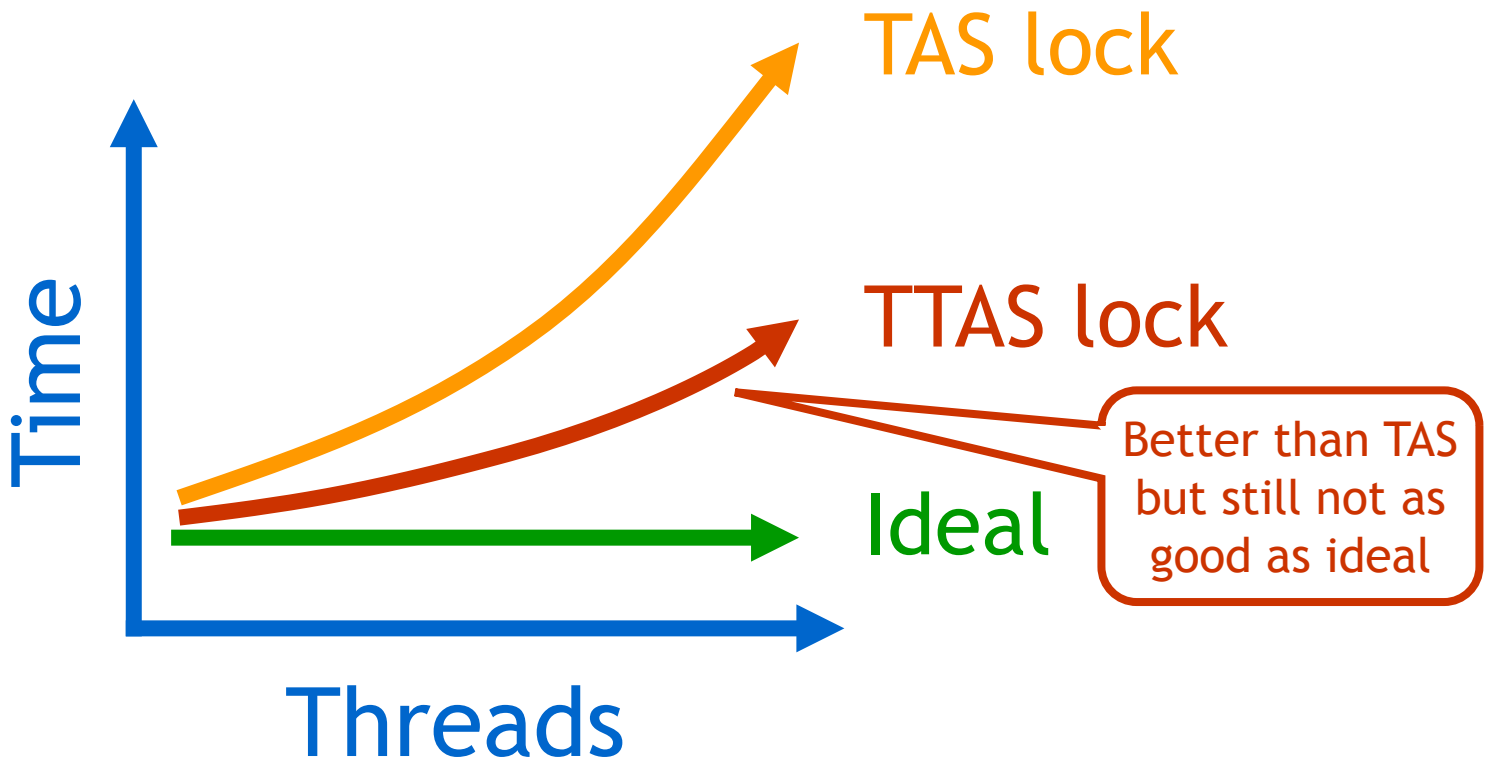
- In critical section, run ops X then ops Y
  - As long as quiescence time is less than X, no drop in performance
  - By gradually varying X, can determine the exact time to quiesce

Quiescence Time



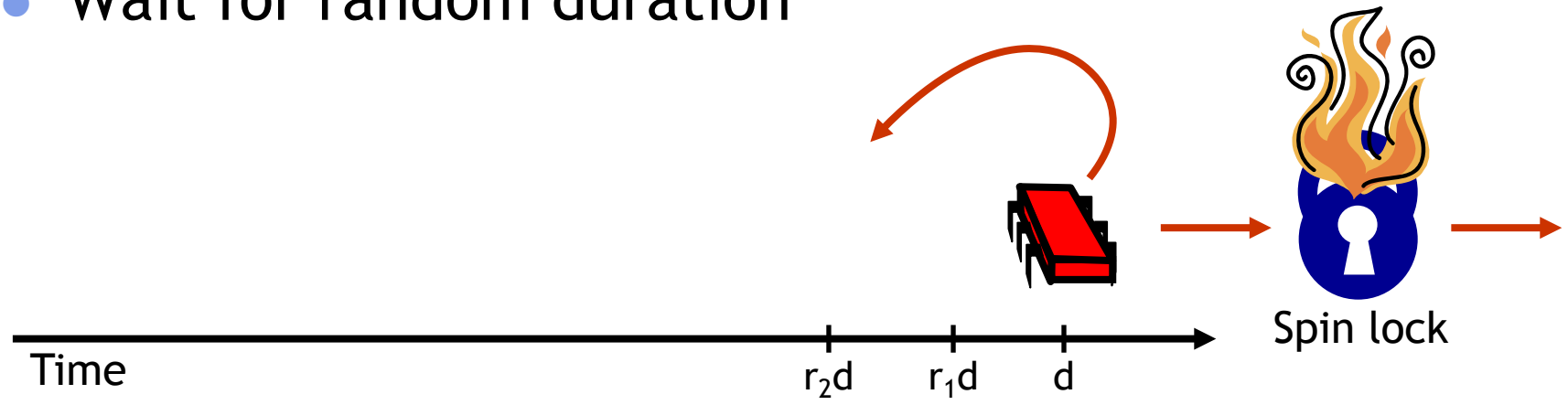
Increases linearly with the number of processors for bus architecture

Mystery Explained



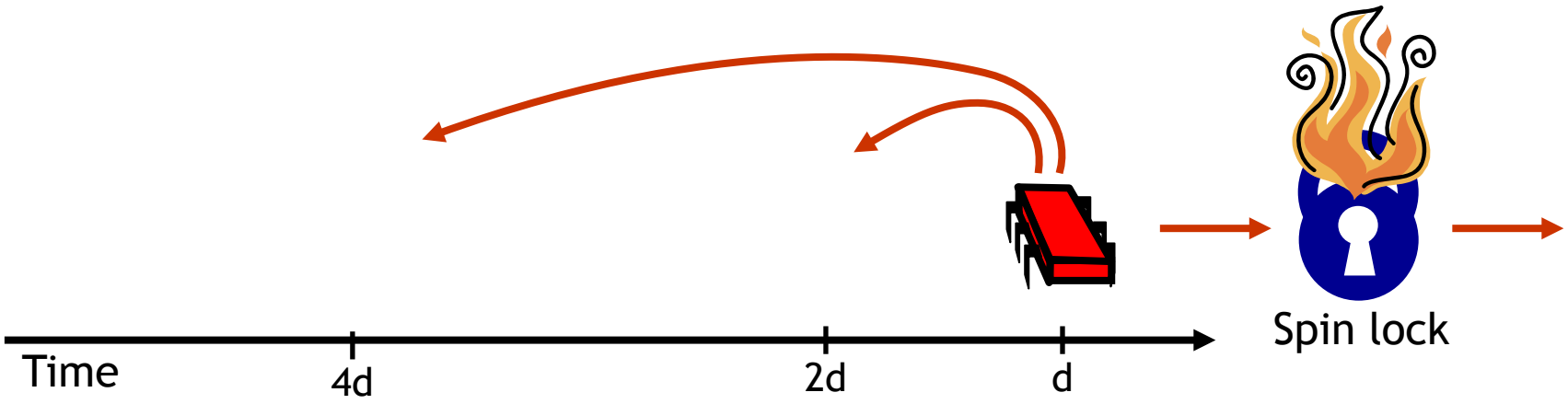
Solution: Introduce Delay

- If the lock looks free
  - But I fail to get it
- There must be much contention
  - Better to back off than to collide again
  - Wait for random duration



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;
    }
  }
}
```

Fix minimum delay

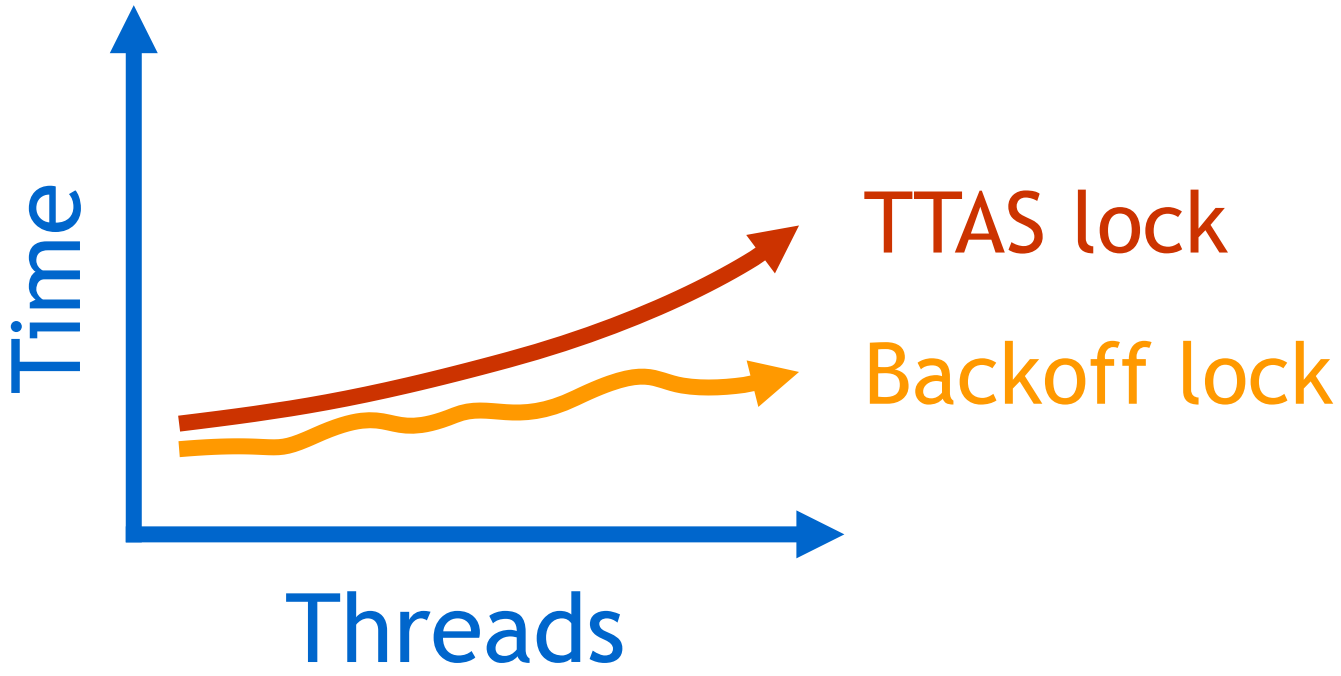
Wait until lock looks free

If we win, return

Back off for random duration

Double max delay, within reason

Spin-Waiting Overhead



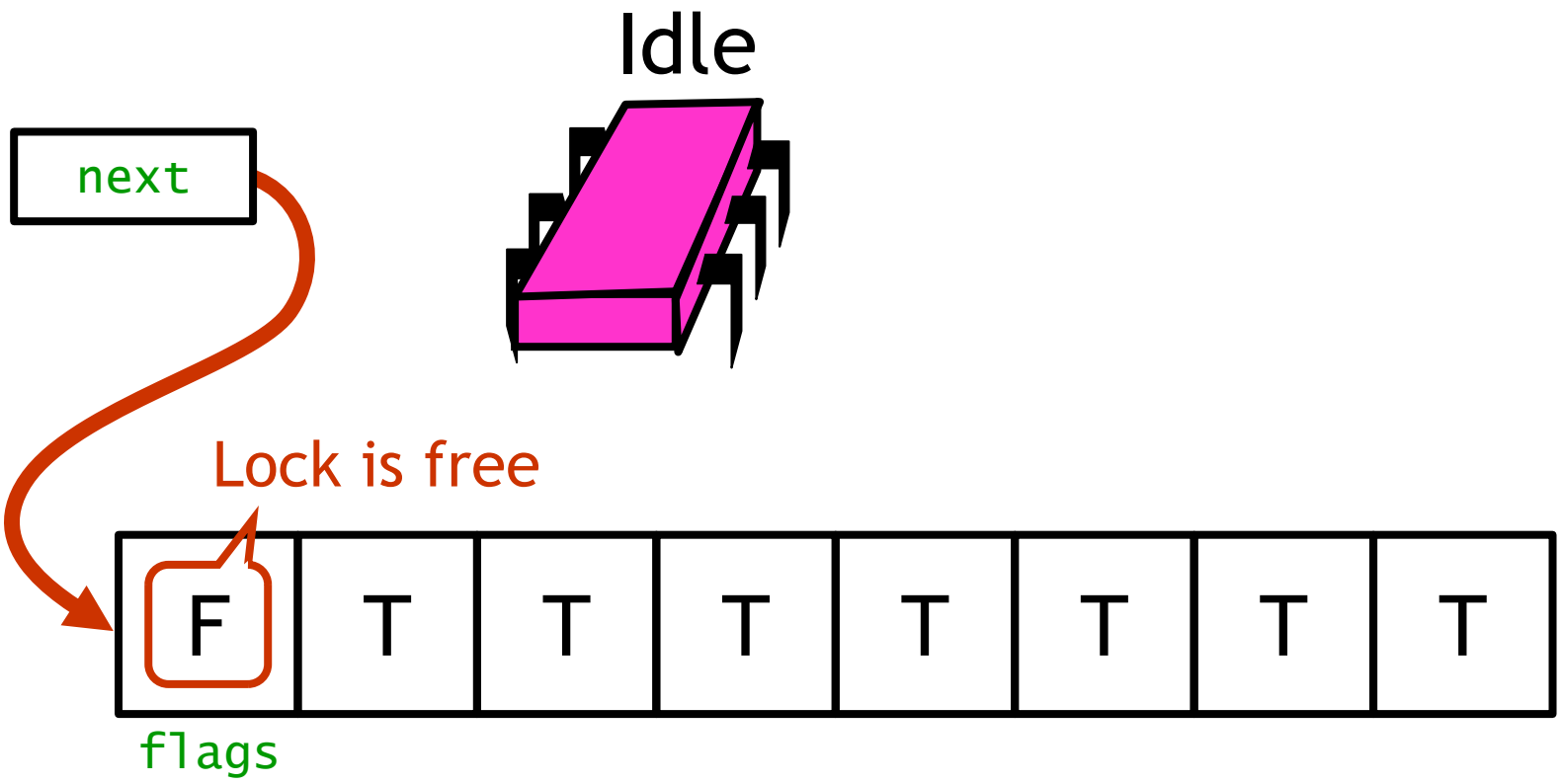
Backoff: Other Issues

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

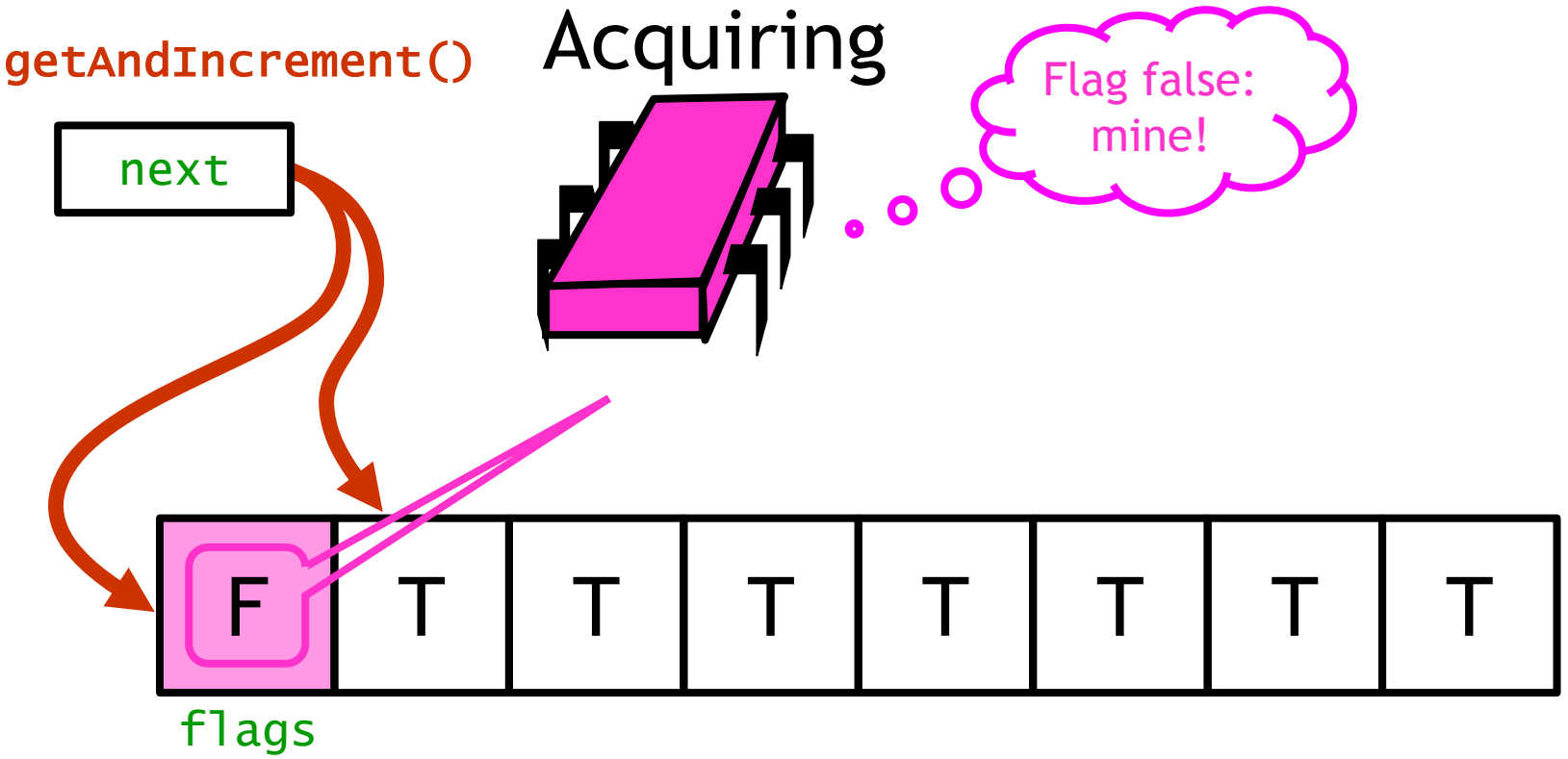
Idea

- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others
- Invalidation traffic reduced by having each thread spin on different location
- Provides first-come-first-served fairness

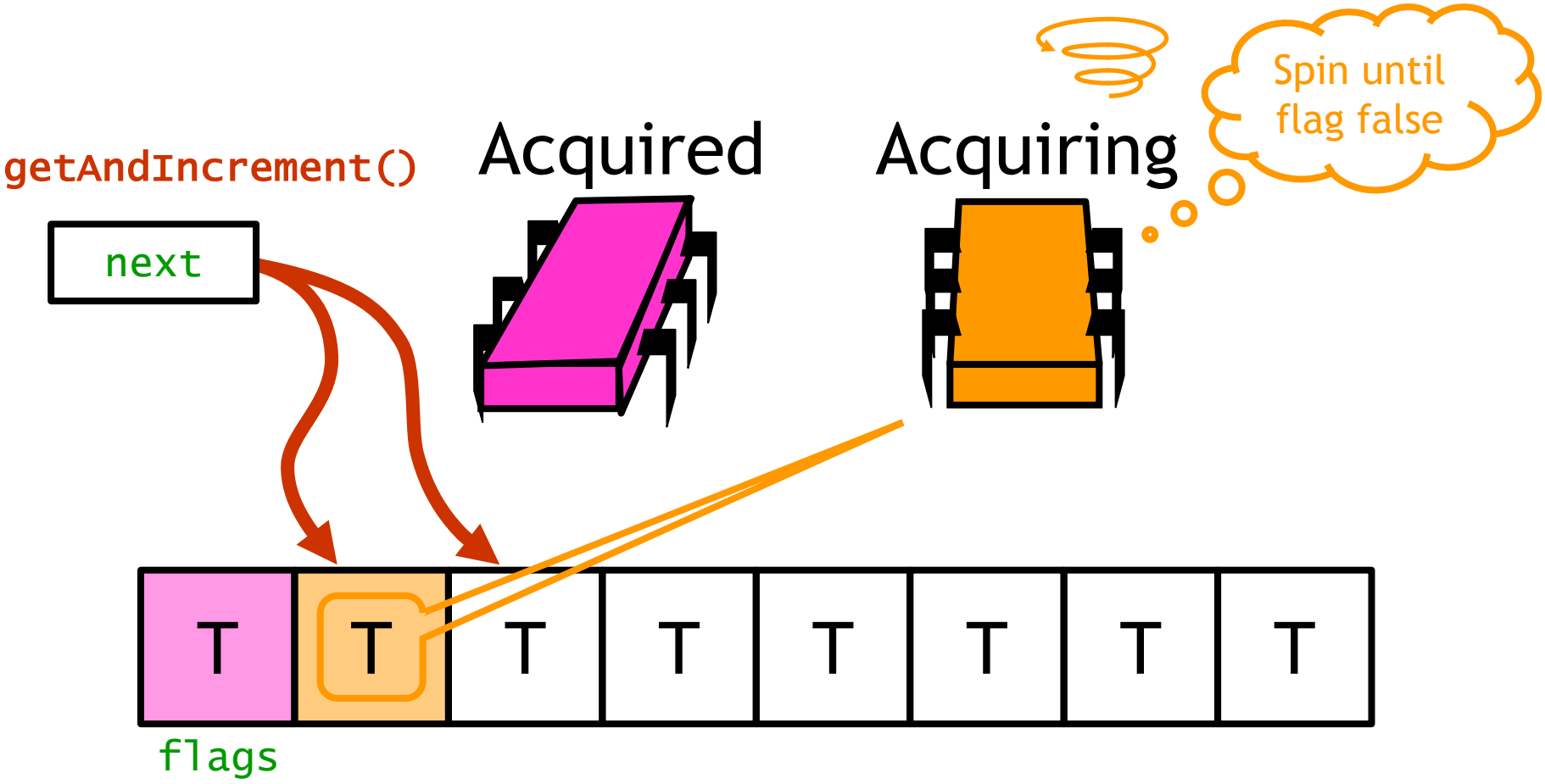
Anderson Queue Lock



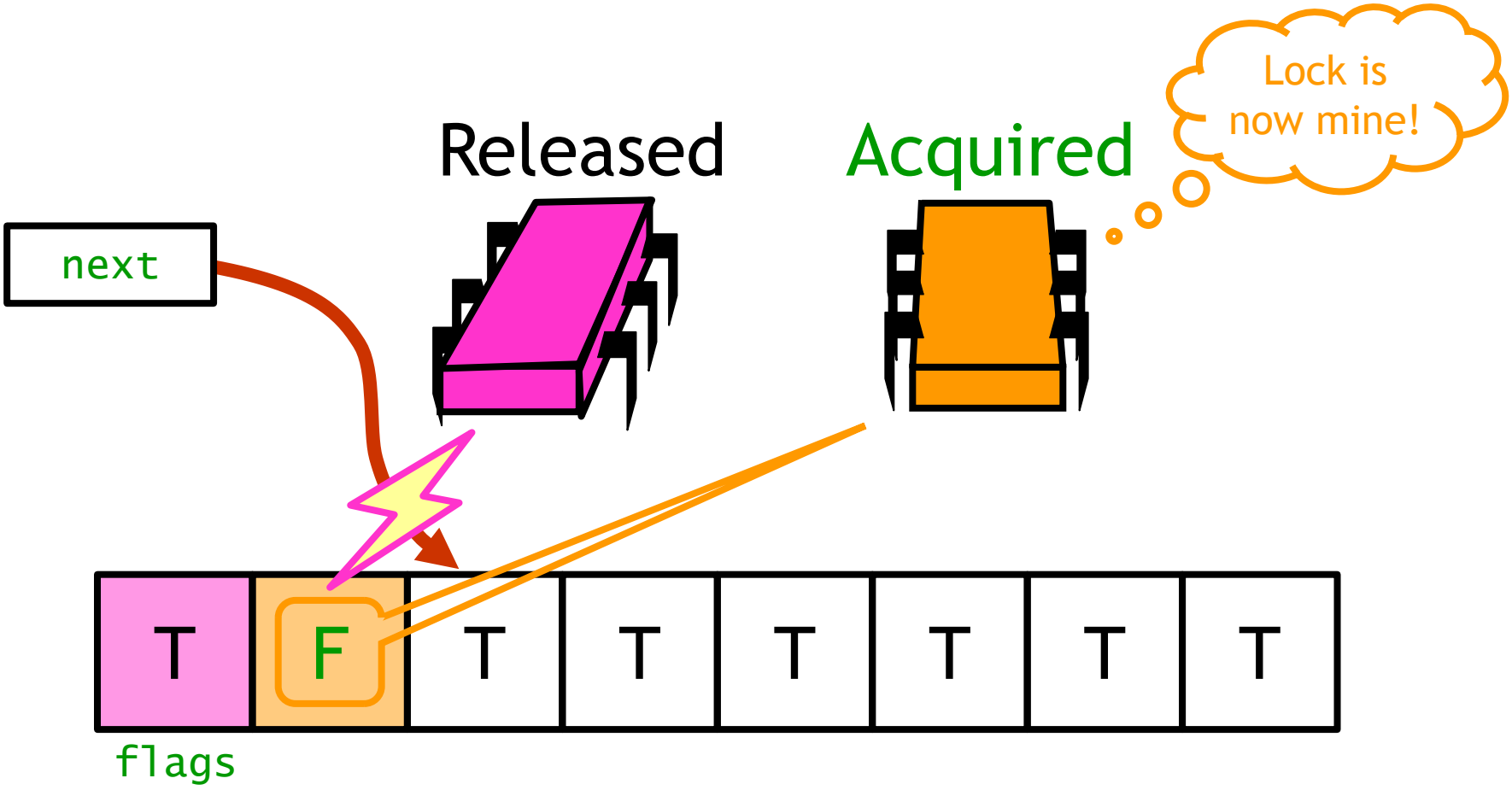
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock

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

At least one flag per thread (N threads)

Next flag to use

Thread-local variable to keep track of own slot

Anderson Queue Lock

```
class ALock implements Lock {
  …
  public lock() {
    int slot = next.getAndIncrement();
    while (flags[slot % N].get()) {}
    flags[slot % N].set(true);
    mySlot.set(new Integer(slot));
  }
  public unlock() {
    int slot = mySlot.get().intValue();
    flags[(slot + 1) % N].set(false);
  }
}
```

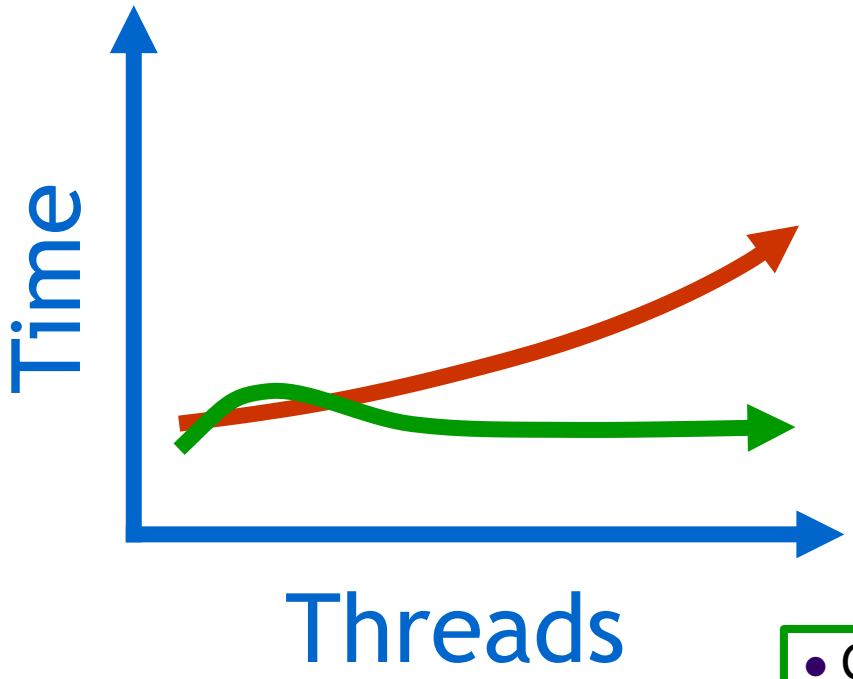
Take next slot

Spin until free

Prepare slot for re-use

Tell next thread to go

Performance



- Curve is practically flat
- Scalable performance
- FIFO fairness

Anderson Queue Lock

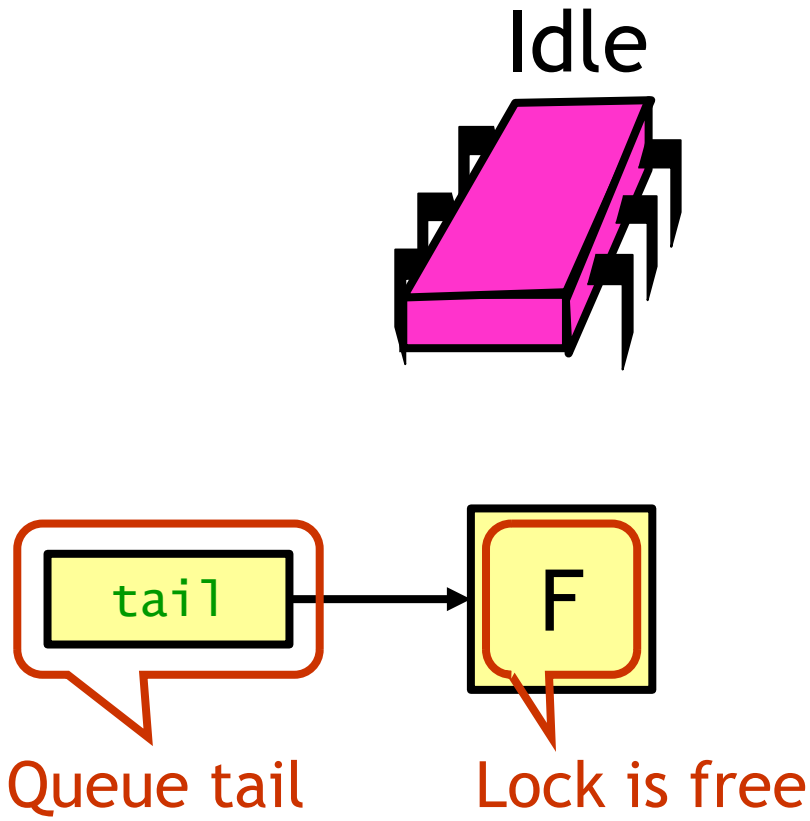
- Good
  - First truly scalable lock
  - Simple, easy to implement
- Bad
  - Space hog (one array per lock)
  - One bit per thread
    - Unknown number of threads
    - Small number of actual contenders?

CLH Lock

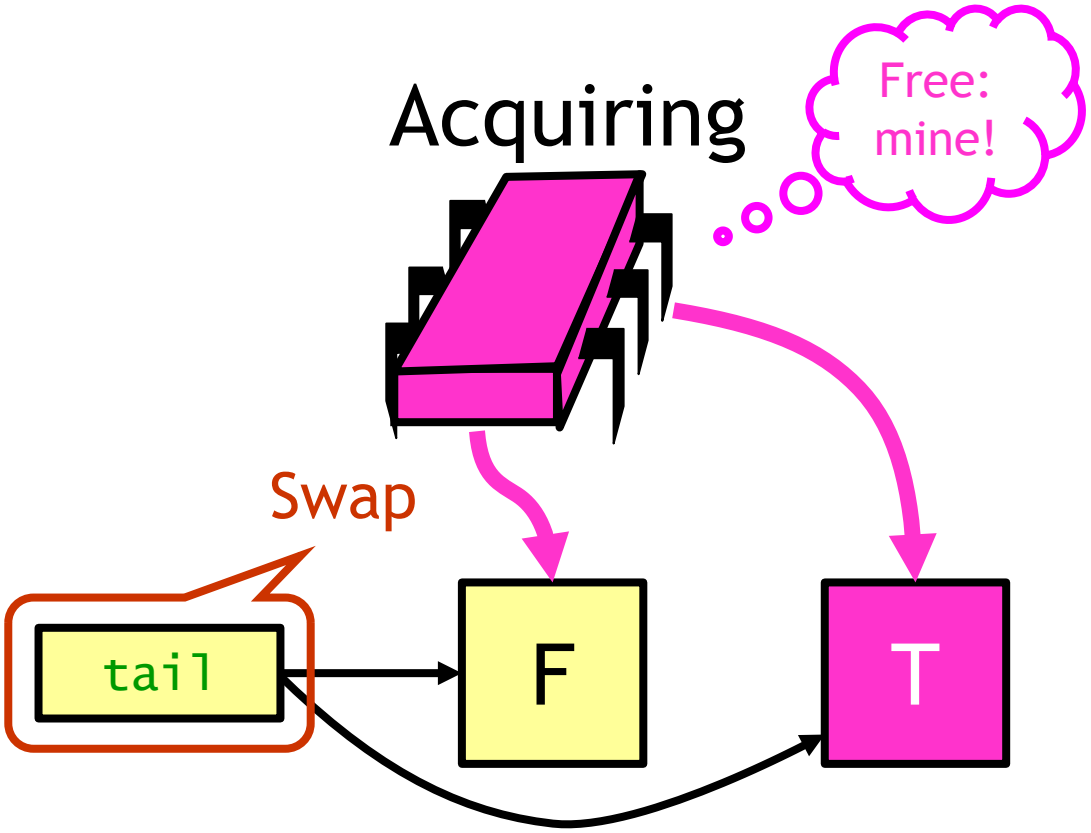
- CLH: T. Craig, A. Landin, E. Hagersten

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

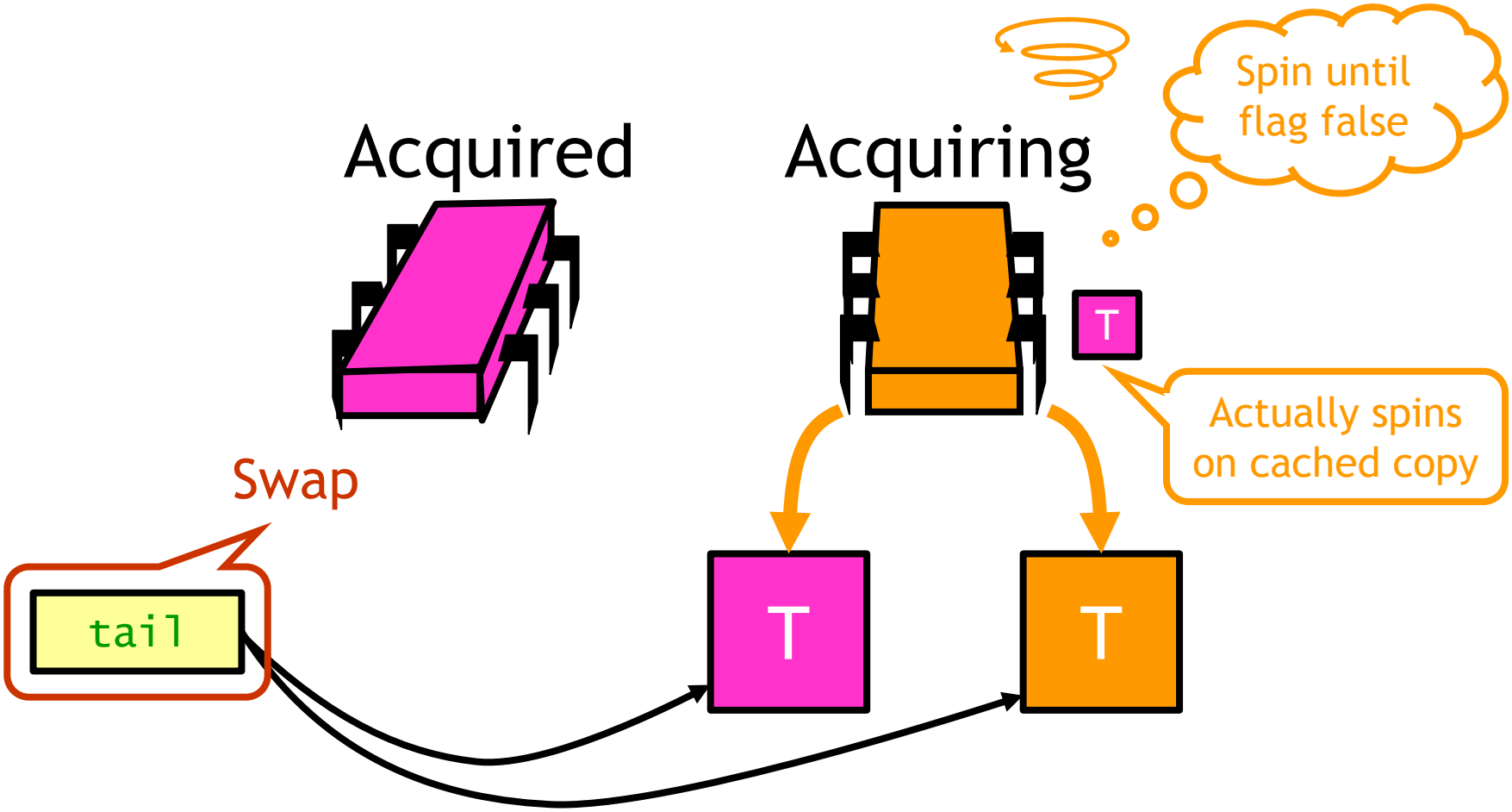
CLH: Initially



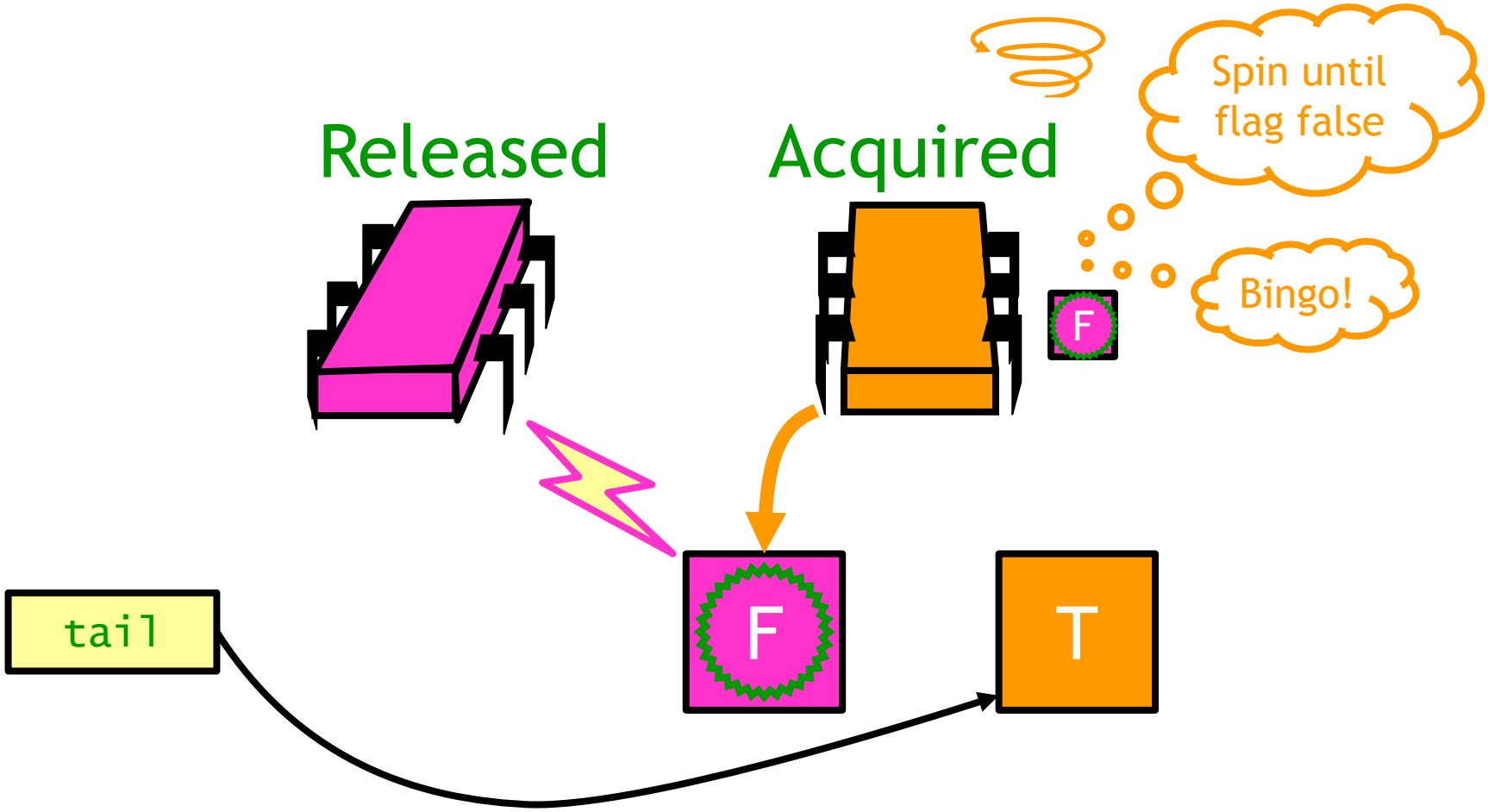
CLH: Purple Wants the Lock



CLH: Orange Wants the Lock



CLH: 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);
  …
}
```

Not released yet

CLH Queue Lock

```
class CLHLock implements Lock {
  AtomicReference<QNode> tail =
    new AtomicReference<QNode>(new QNode(false));
  ThreadLocal<QNode> myNode =
    new ThreadLocal<QNode>();
  public void lock() {
    QNode node = new QNode();
    QNode pred = tail.getAndSet(node);
    while (pred.locked.get()) {}
    myNode.set(node);
  }
  …
}
```

Tail of queue

Thread-local QNode

Swap in my node

Spin until predecessor releases lock

CLH Queue Lock

```
class CLHLock implements Lock {
  …
  public void unlock() {
    QNode node = myNode.get();
    node.locked.set(false);
  }
}
```

Notify successor

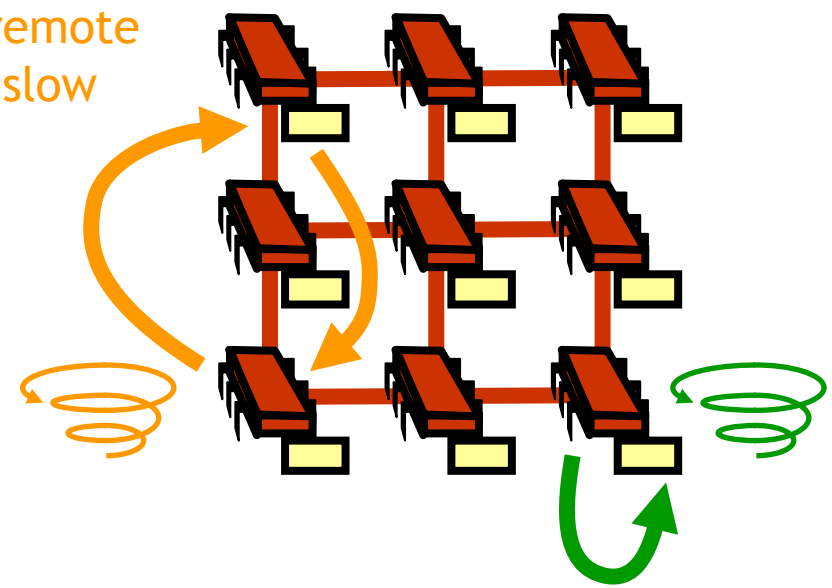
CLH Lock

- Good
  - Lock release affects successor only
  - Small, constant-sized space
- Bad
  - Does not work for uncached NUMA architectures

NUMA Architectures

- 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



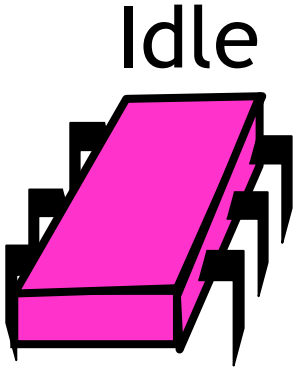
- CLH lock
  - Each thread spins on predecessor's memory
  - Could be far away…

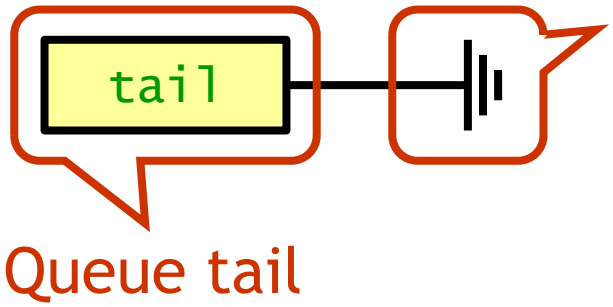
MCS Lock

- MCS: J. Mellor-Crummey, M. Scott

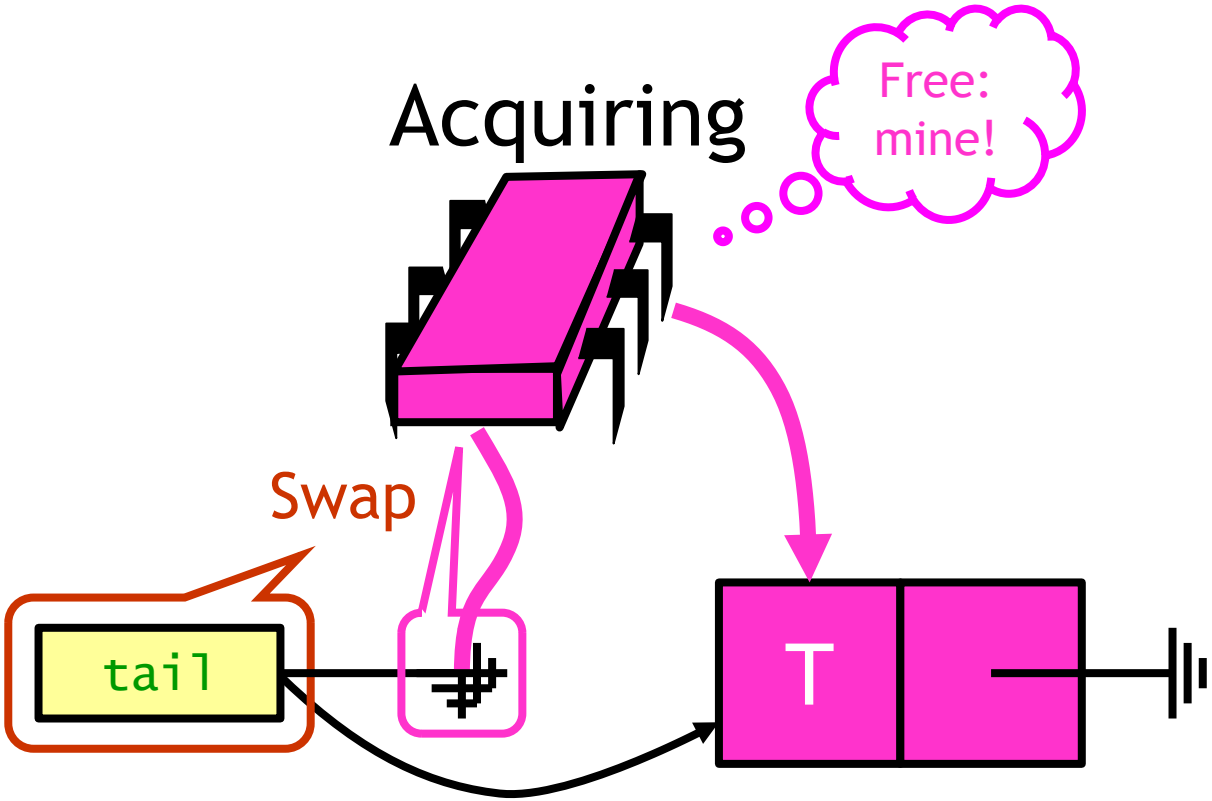
- FIFO order
- Spin on local memory only
- Small, constant-size overhead

MCS: Initially

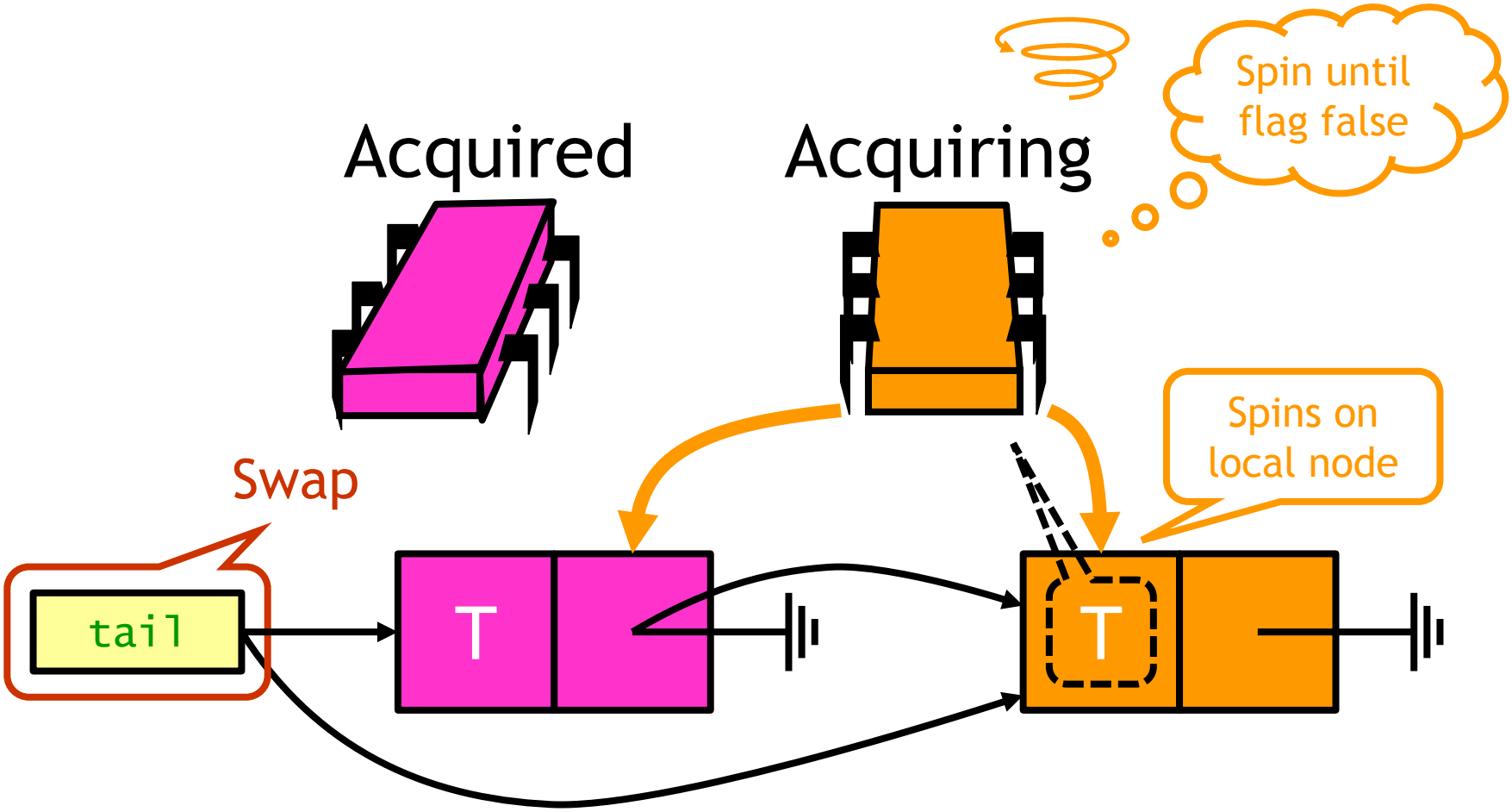




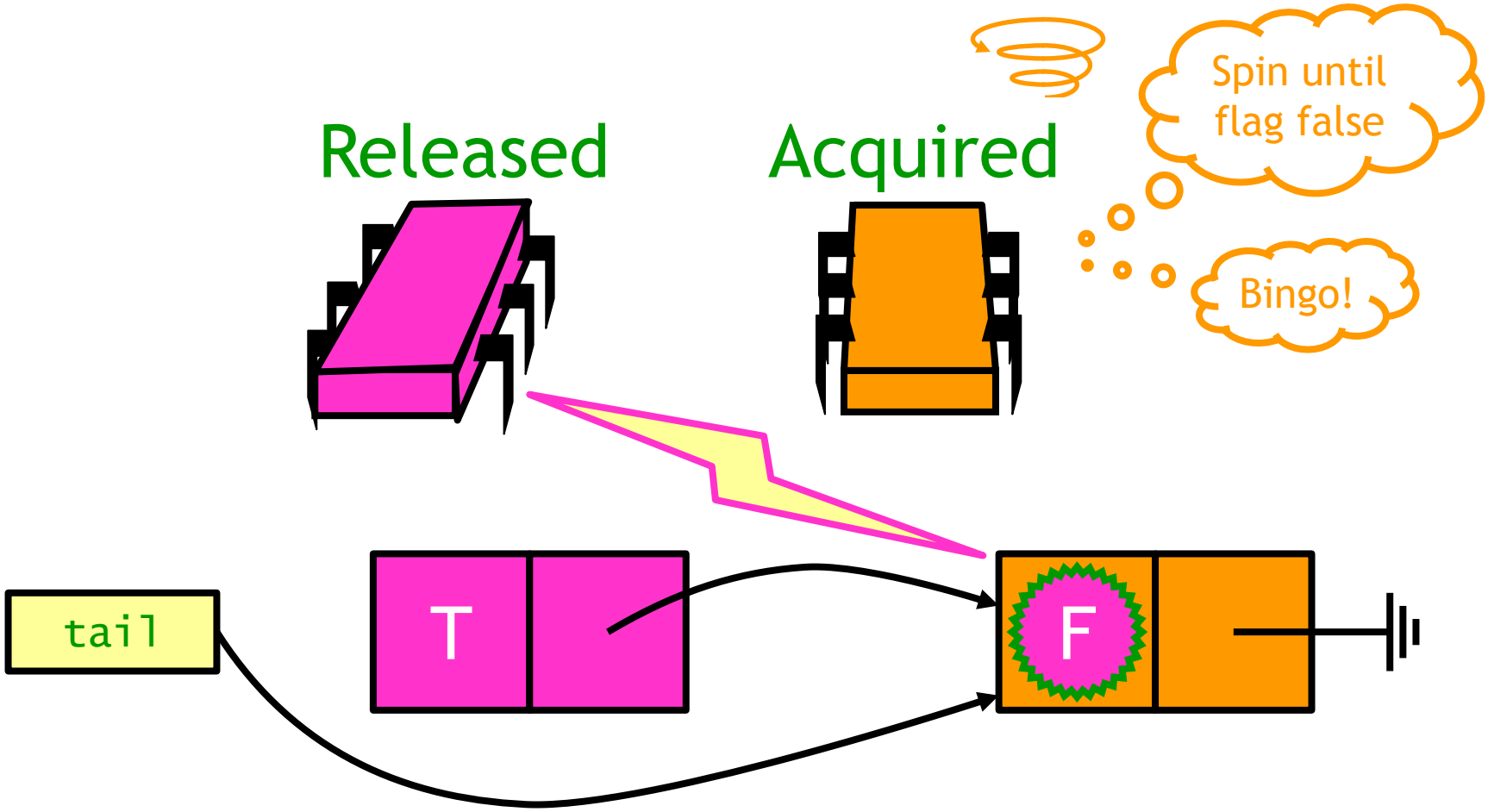
MCS: Purple Wants the Lock



MCS: Orange Wants the Lock



MCS: Purple Releases



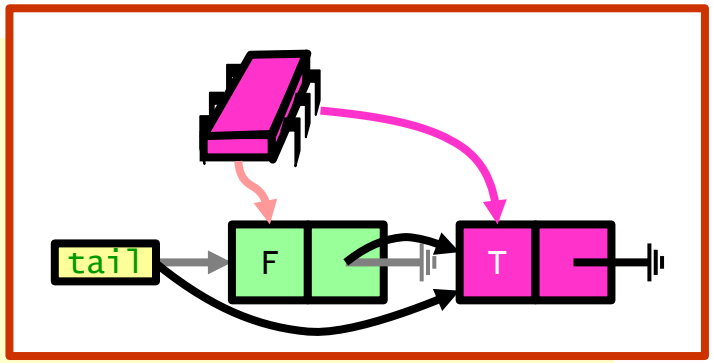
MCS Queue Lock

```
class QNode {
  AtomicBoolean locked =
    new AtomicBoolean(true);
  AtomicReference<QNode> next =
    new AtomicReference<QNode>();
  …
}
```

Next node in queue

MCS Queue Lock

```
class MCSLock implements Lock {
  AtomicReference<QNode> tail =
    new AtomicReference<QNode>();
  ThreadLocal<QNode> myNode =
    new ThreadLocal<QNode>();
  public void lock() {
    QNode node = new QNode();
    QNode pred = tail.getAndSet(node);
    if (pred != null) {
      pred.next.set(node);
      while (node.locked.get()) {}
    }
    myNode.set(node);
  }
  …
}
```



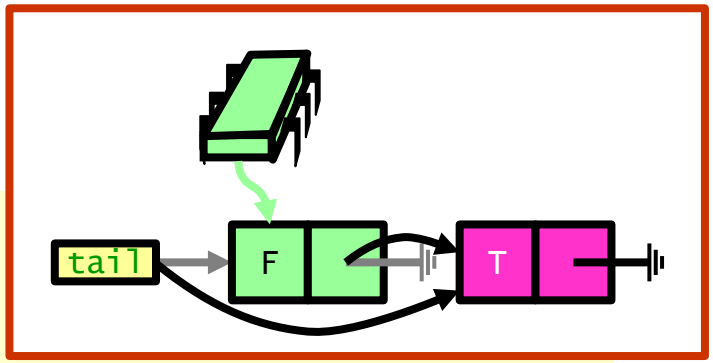
Make a QNode

Add my Node to tail of queue

Fix if queue was non-empty

Wait until unlocked

MCS Queue Lock




```
class MCSLock implements Lock {
  …
  public void unlock() {
    QNode node = myNode.get();
    if (node.next.get() == null) {
      if (tail.compareAndSet(node, null))
        return;
      while (node.next.get() == null) {}
    }
    QNode succ = node.next.get();
    succ.locked.set(false);  }
}
```

Missing successor?

If really no successor, return

Otherwise wait for successor to catch up

Pass lock to successor

Abortable Locks

- What if you want to give up waiting for lock?
  - E.g., timeout, transaction aborted by user
- Back-off lock
  - Aborting is trivial: just return from `lock()` call
- Queue locks
  - Cannot just quit: thread in line behind will starve

⇒ **Abortable CLH Lock** (ToLock)

  - Mark removed node
  - Successor detects and skips removed node

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