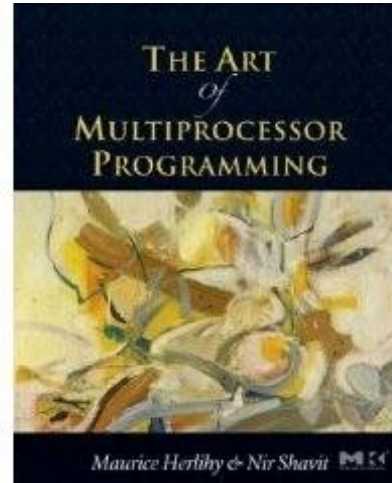
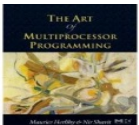


Spin Locks and Contention



Hyungsoo Jung



Focus so far: Correctness and Progress

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



New Focus: Performance

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



Kinds of Architectures

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

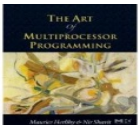


Kinds of Architectures

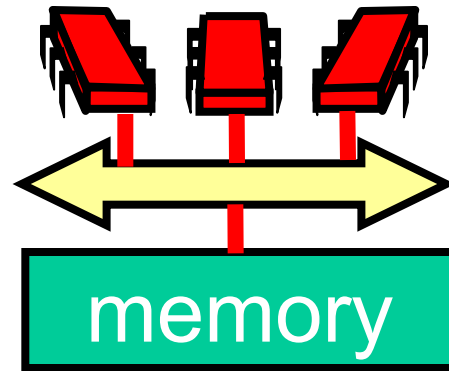
- SISD (Uniprocessor)
 - Single instruction stream
 - Single data stream
- SIMD (Vector)
 - Single instruction
 - Multiple data

Our space

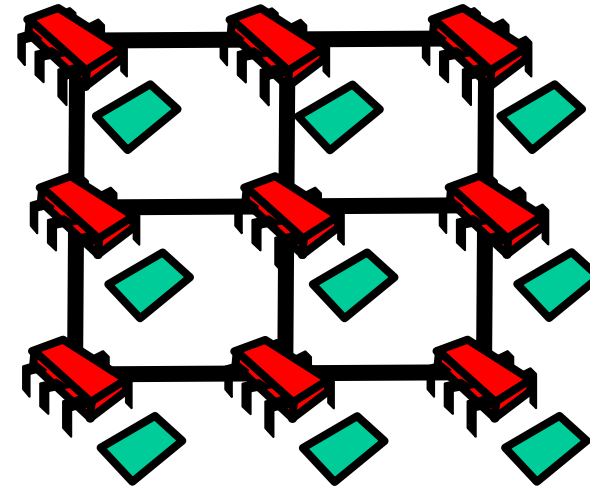
- MIMD (Multiprocessors)
 - Multiple instruction
 - Multiple data.



MIMD Architectures



Shared Bus



Distributed

- Memory Contention
- Communication Contention
- Communication Latency

Today: Revisit Mutual Exclusion

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



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

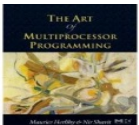
- Keep trying
 - “spin” or “busy-wait”
 - Good if delays are short
- Give up the processor
 - Good if delays are long
 - Always good on uniprocessor



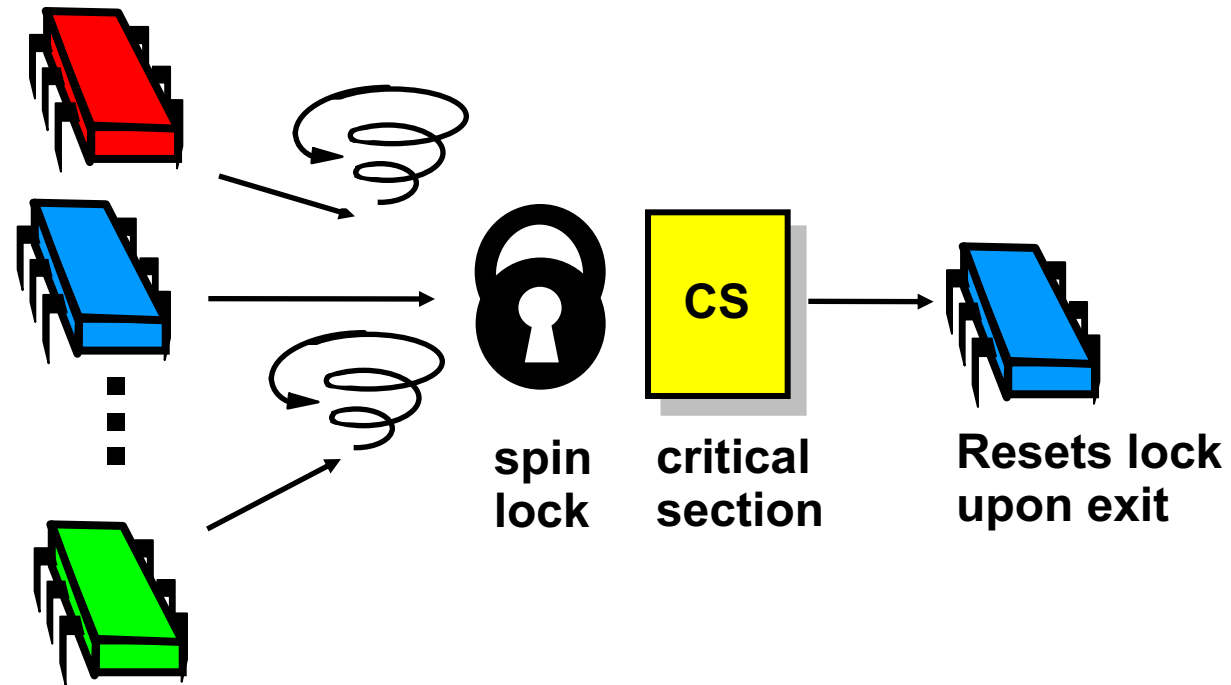
What Should you do if you can't get a lock?

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

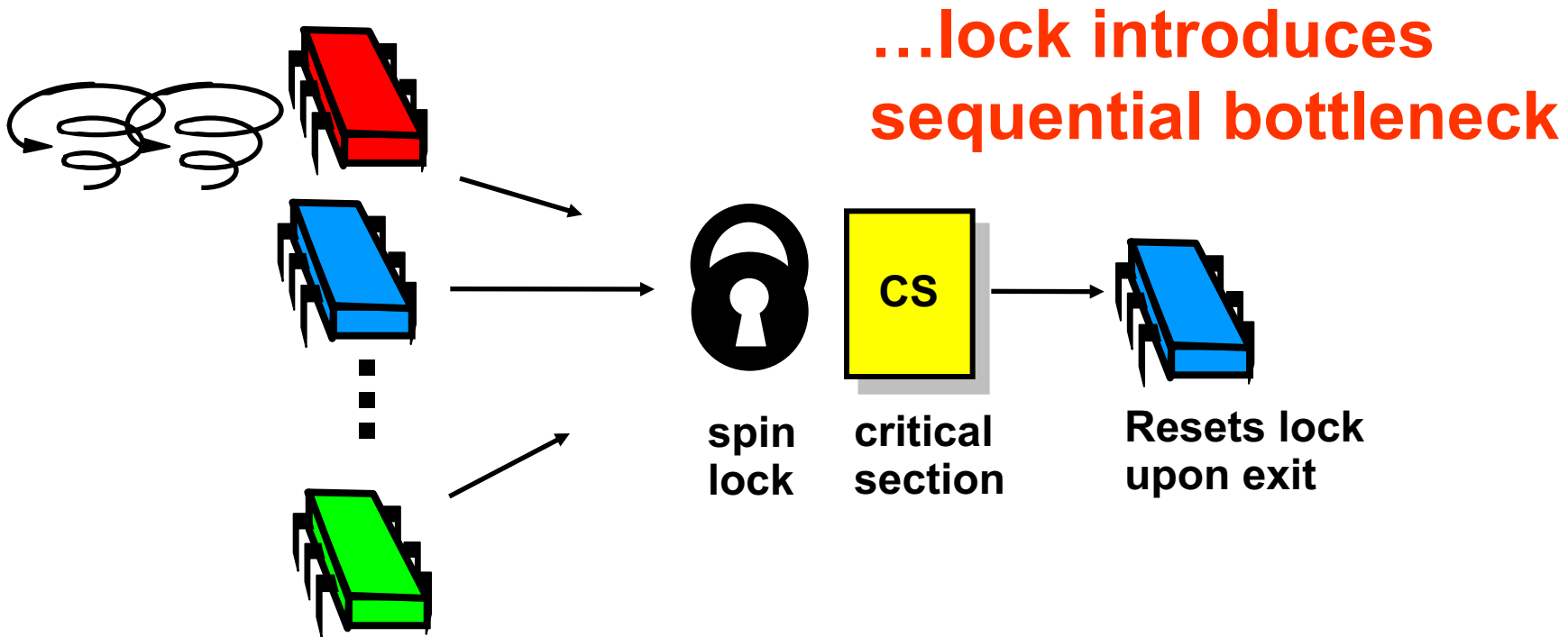
our focus



Basic Spin-Lock

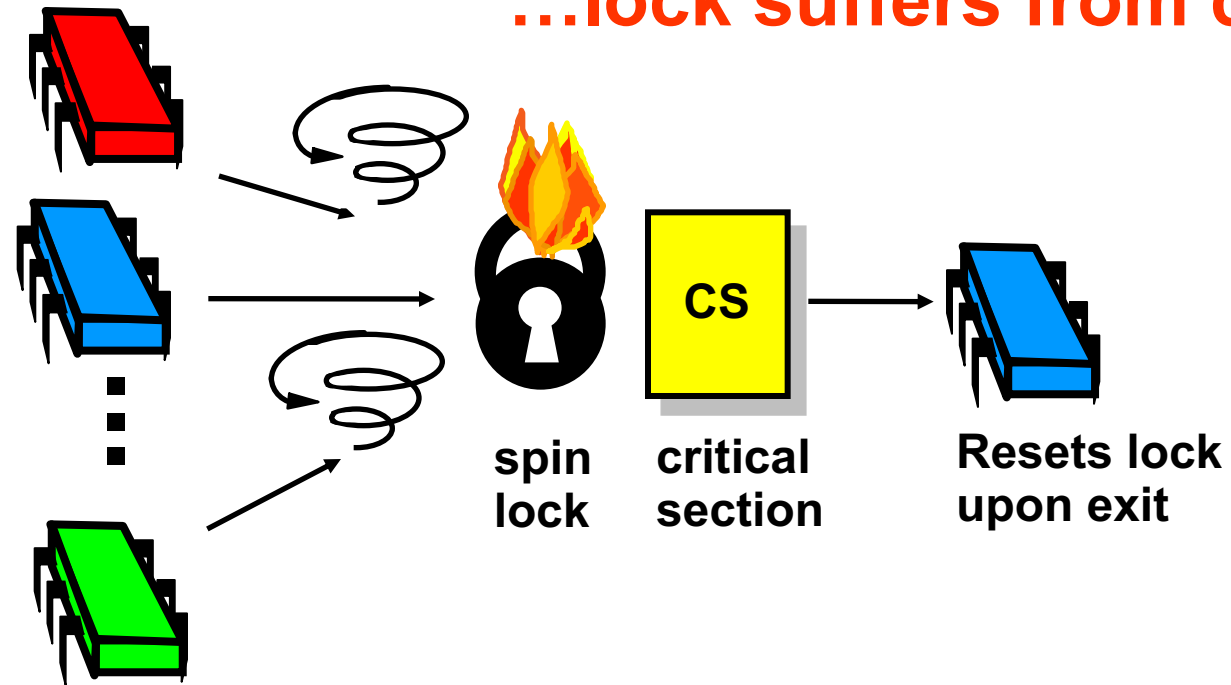


Basic Spin-Lock



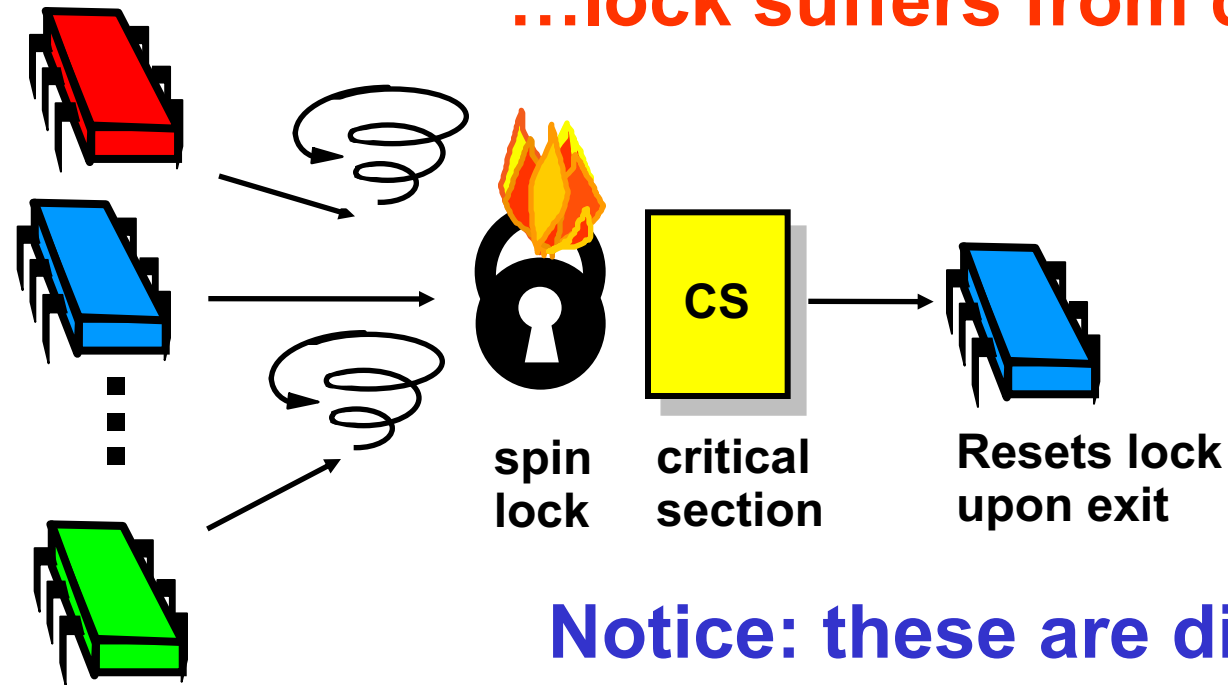
Basic Spin-Lock

...lock suffers from contention



Basic Spin-Lock

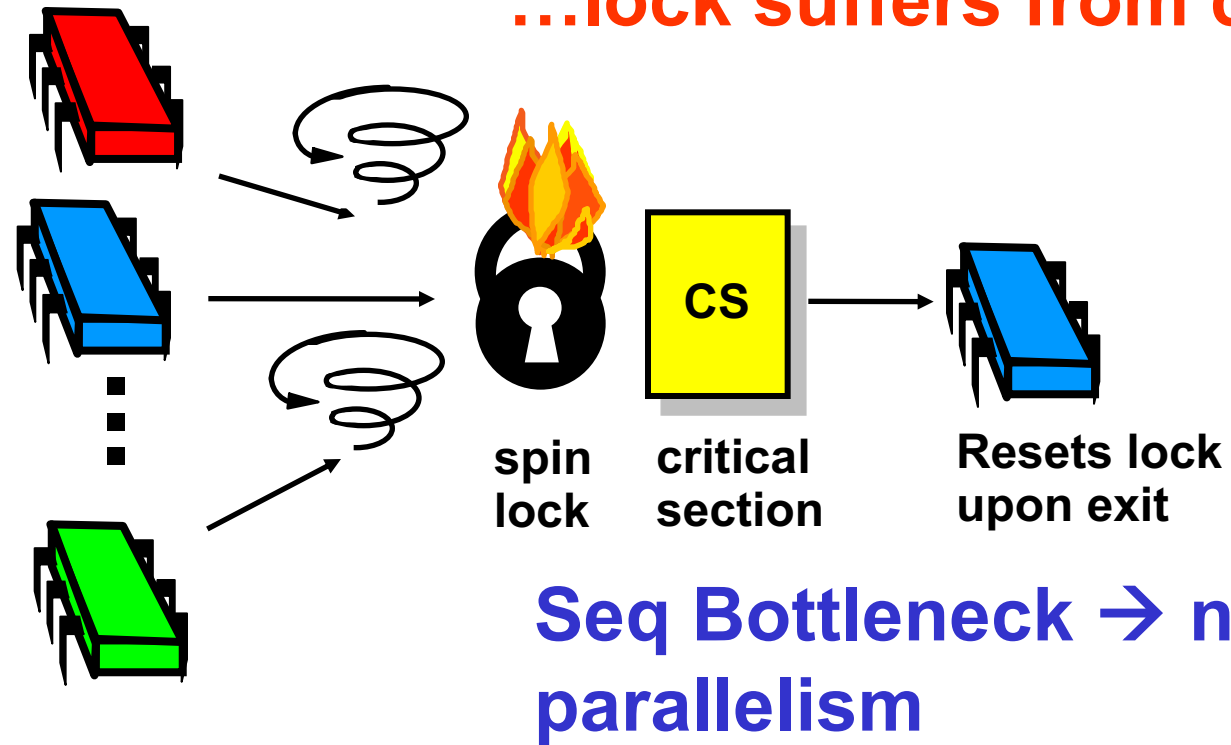
...lock suffers from contention



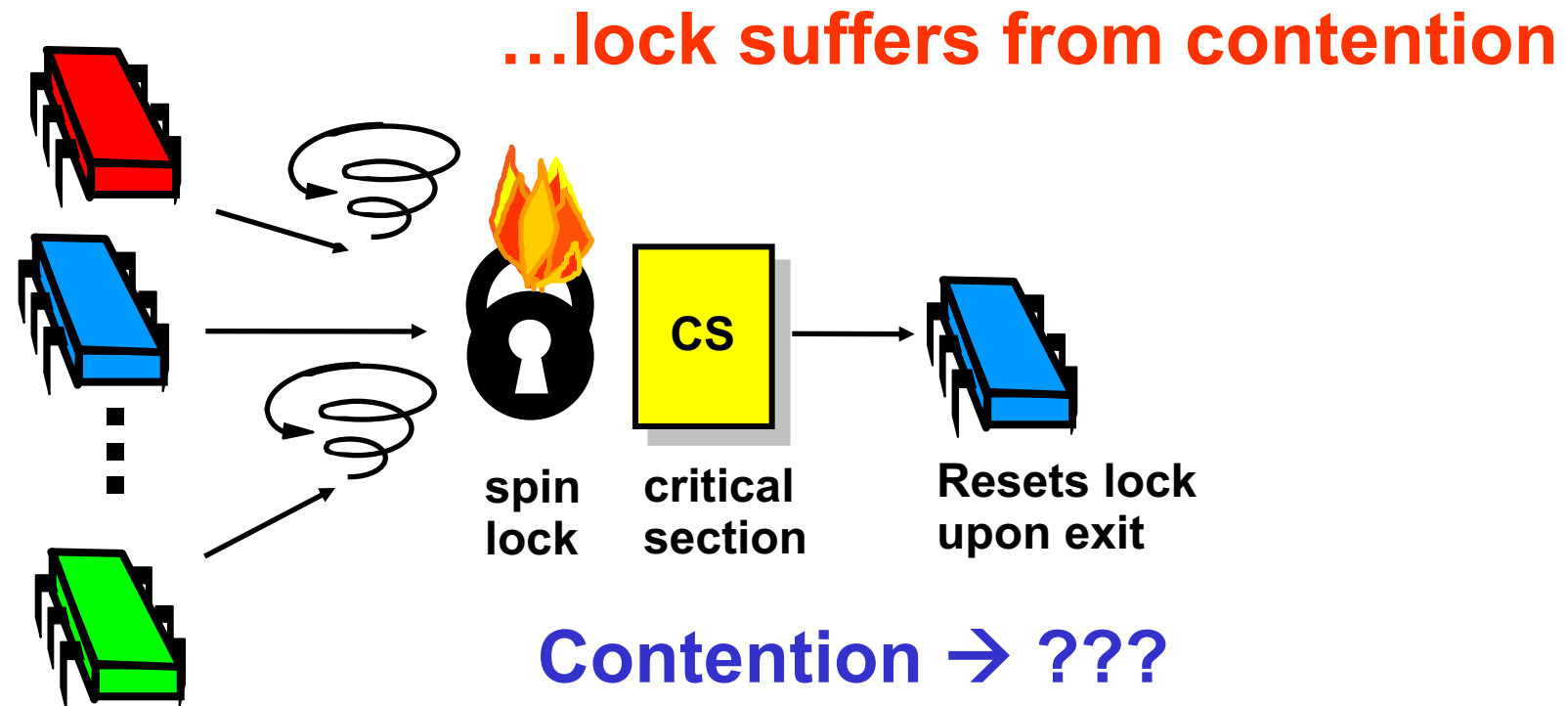
Notice: these are distinct phenomena

Basic Spin-Lock

...lock suffers from contention

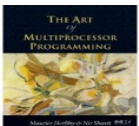


Basic Spin-Lock



Review: Test-and-Set

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



Review: Test-and-Set

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



Review: Test-and-Set

```
public class AtomicBoolean {
```

```
    boolean value;
```

```
    public synchronized boolean  
    getAndSet(boolean newValue) {
```

```
        boolean prior = value;
```

```
        value = newValue;
```

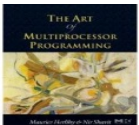
```
        return prior;
```

```
    }
```

```
}
```

Package

java.util.concurrent.atomic

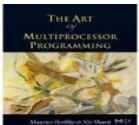


Review: Test-and-Set

```
public class AtomicBoolean {  
    boolean value;
```

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

**Swap old and new
values**



Review: Test-and-Set

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

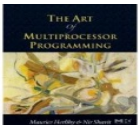


Review: Test-and-Set

```
AtomicBoolean lock  
= new AtomicBoolean(false)
```

```
boolean prior = lock.getAndSet(true)
```

**Swapping in true is called
“test-and-set” or TAS**



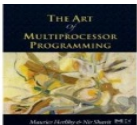
Test-and-Set Locks

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



Test-and-set Lock

```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    }  
}
```



Test-and-set Lock

```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state  
    }}  
    Lock state is AtomicBoolean
```


Test-and-set Lock

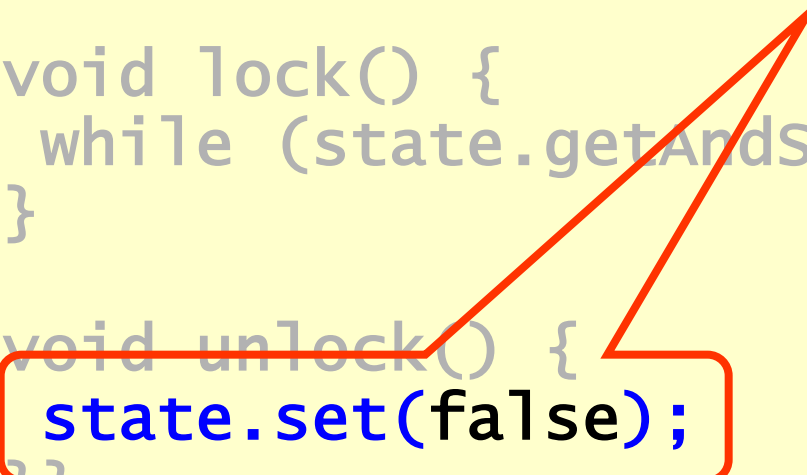
```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    }  
}
```

Keep trying until lock acquired

Test-and-set Lock

```
class TAILOR {  
    AtomicCB  
    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 an RMW operation...

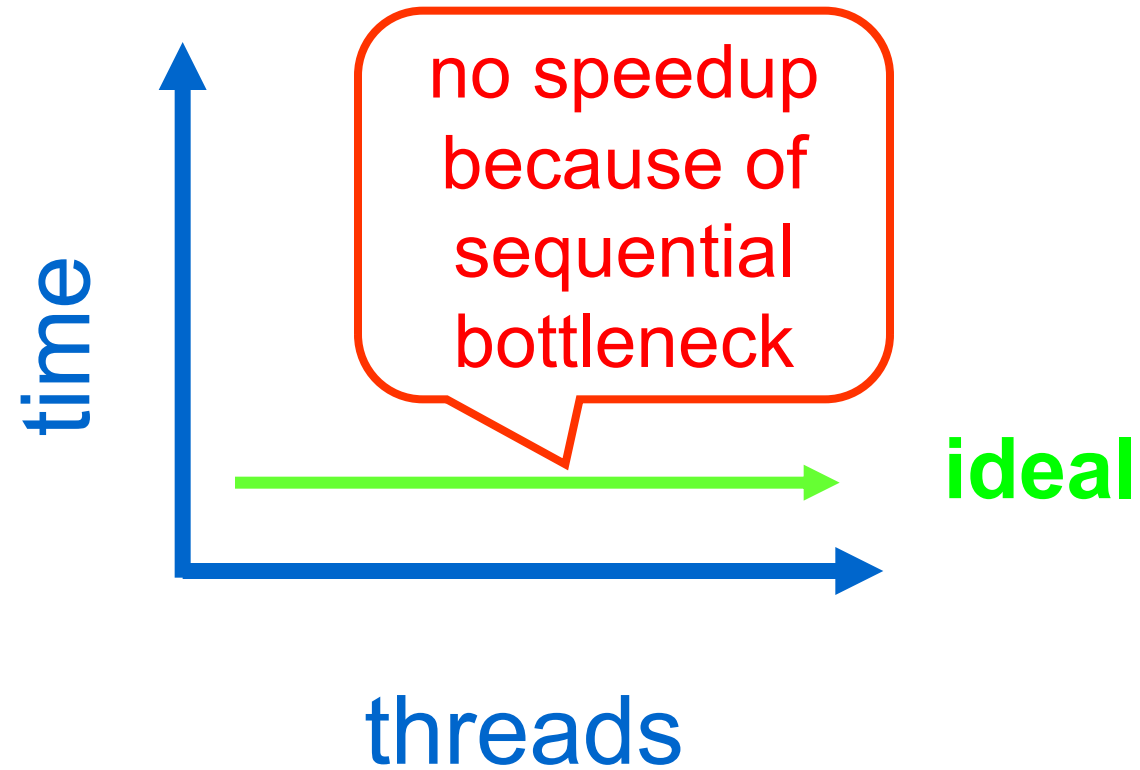


Performance

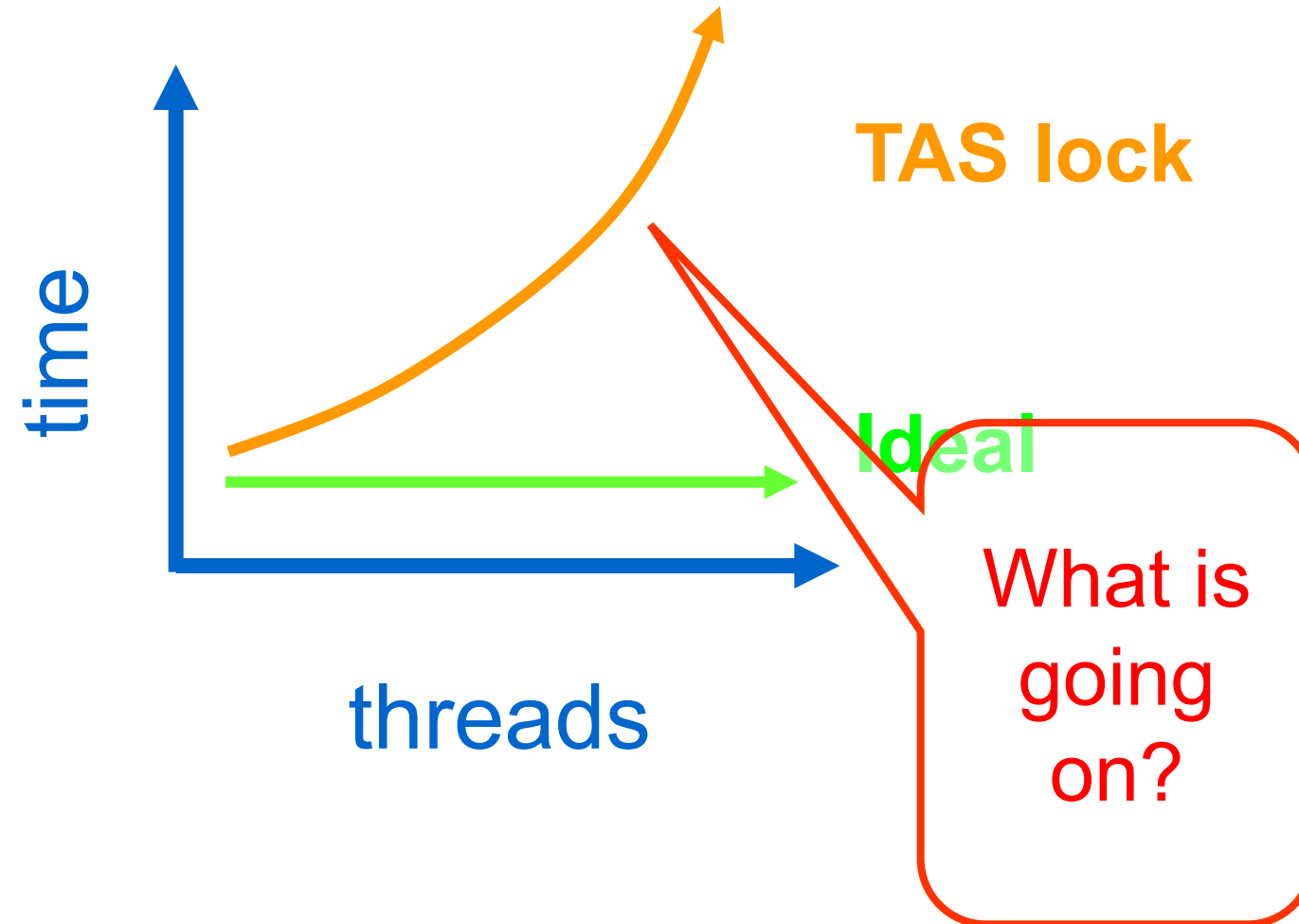
- Experiment
 - n threads
 - Increment shared counter 1 million times
- How long should it take?
- How long does it take?



Graph



Mystery #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;  
        }  
    }  
}
```

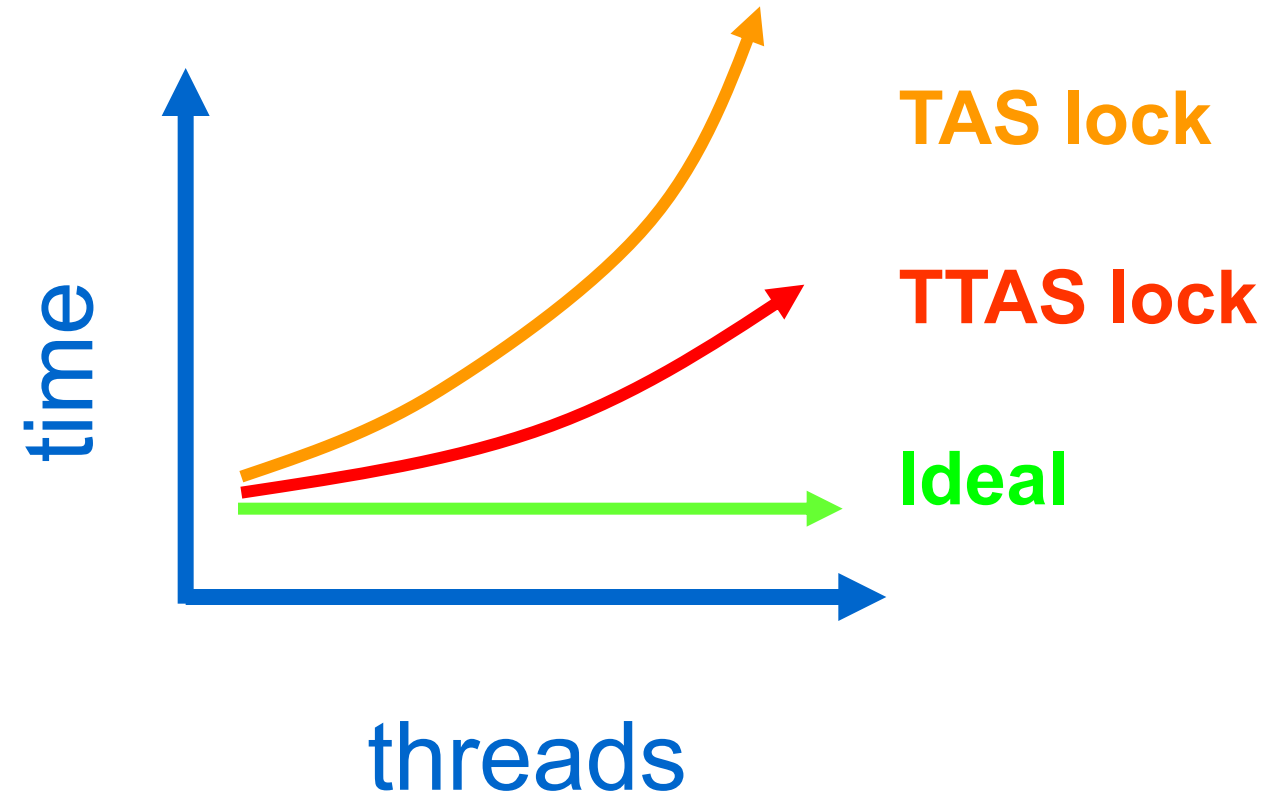
Wait until lock looks free

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

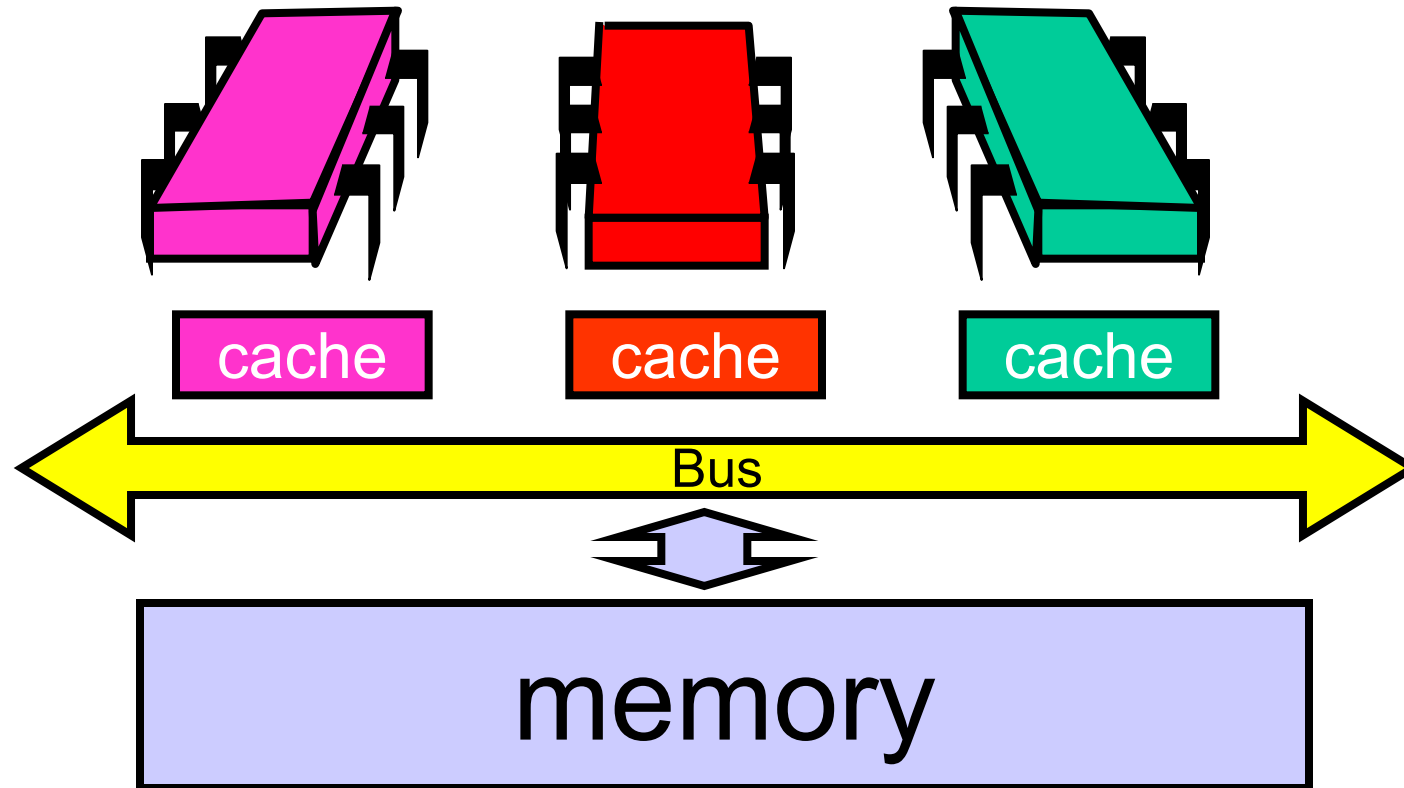


Opinion

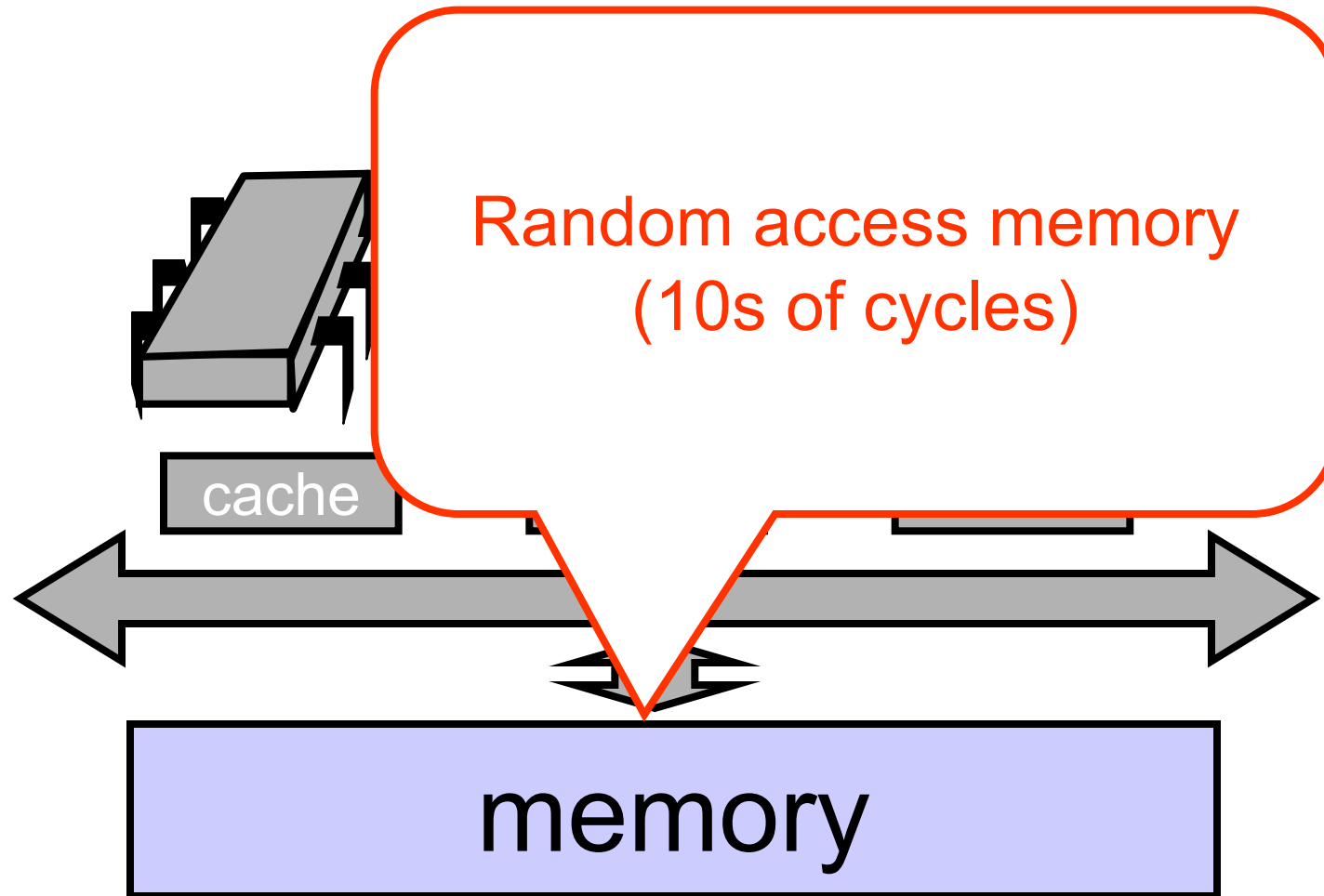
- Our memory abstraction is broken
- TAS & TTAS methods
 - Are provably the same (in our model)
 - Except they aren't (in field tests)
- Need a more detailed model ...



Bus-Based Architectures



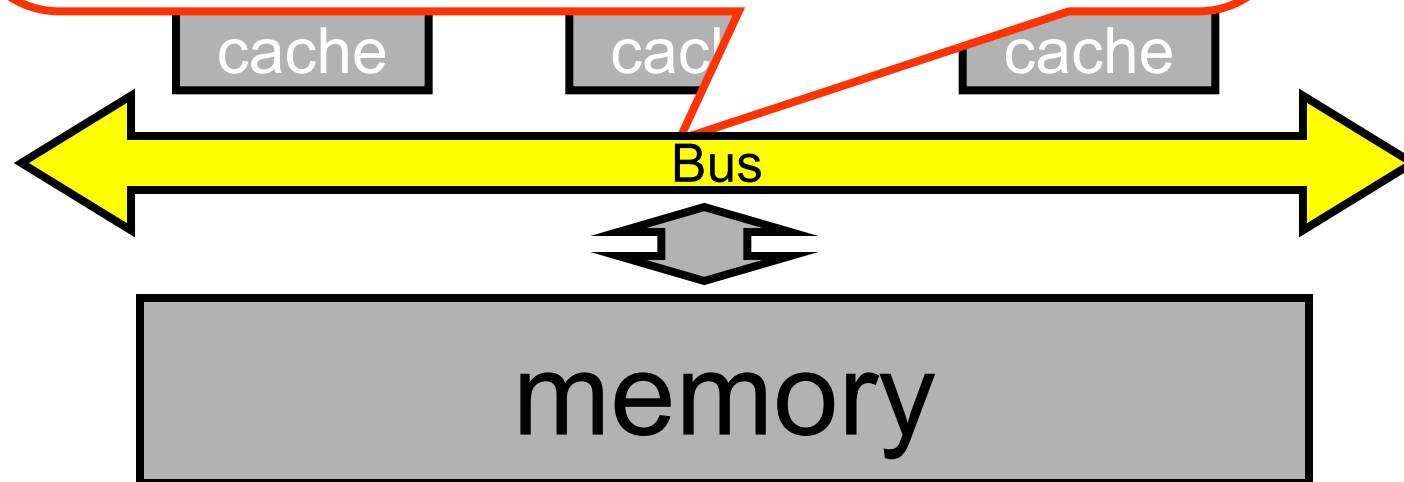
Bus-Based Architectures



Bus-Based Architectures

Shared Bus

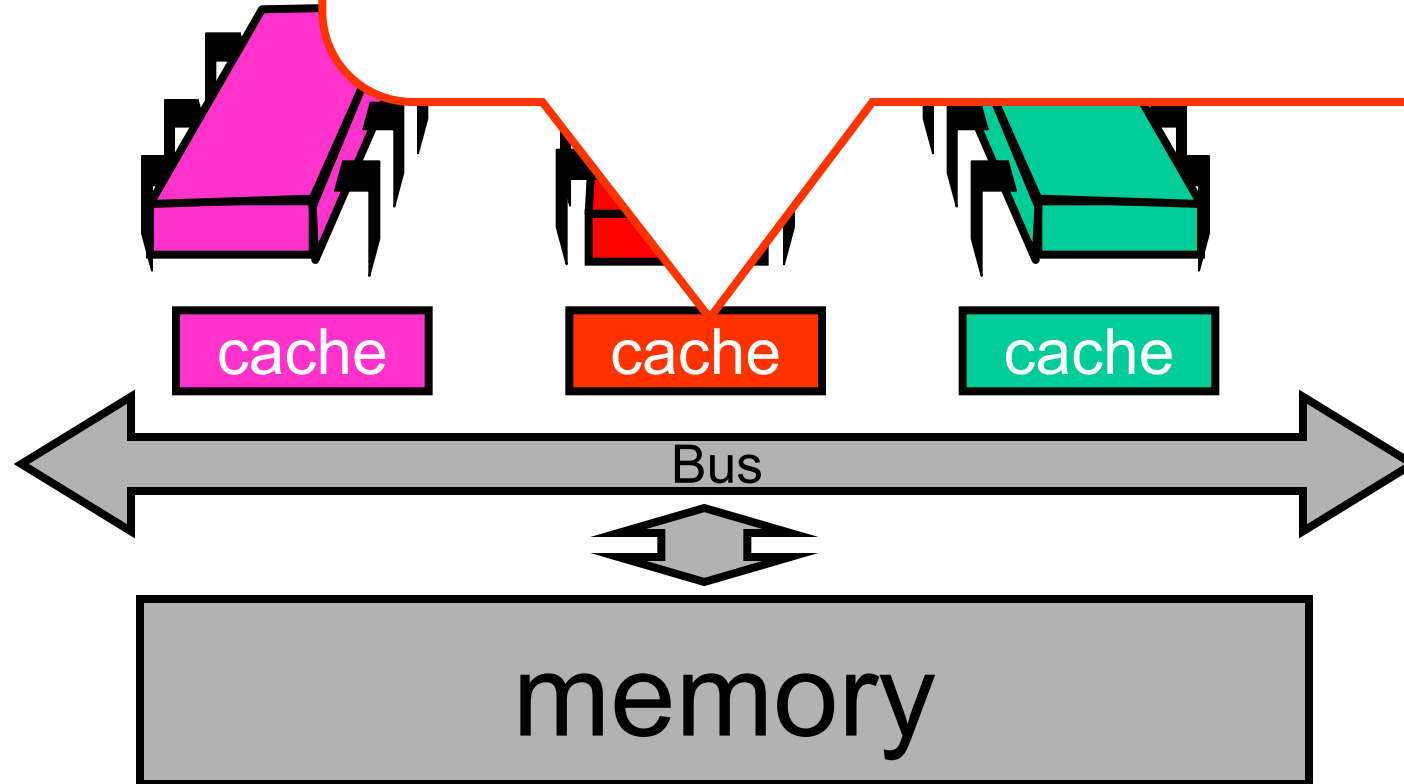
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”



Bus-

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™

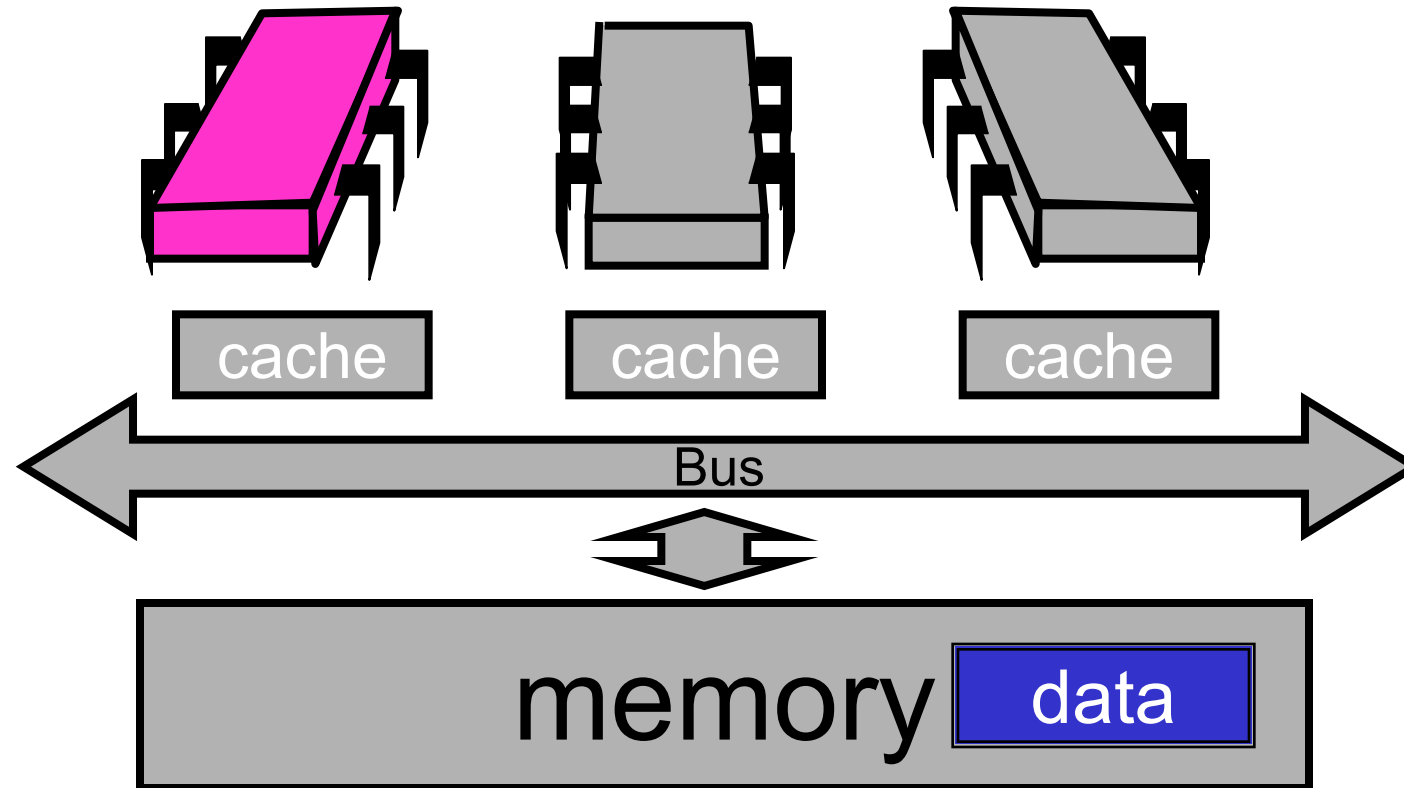


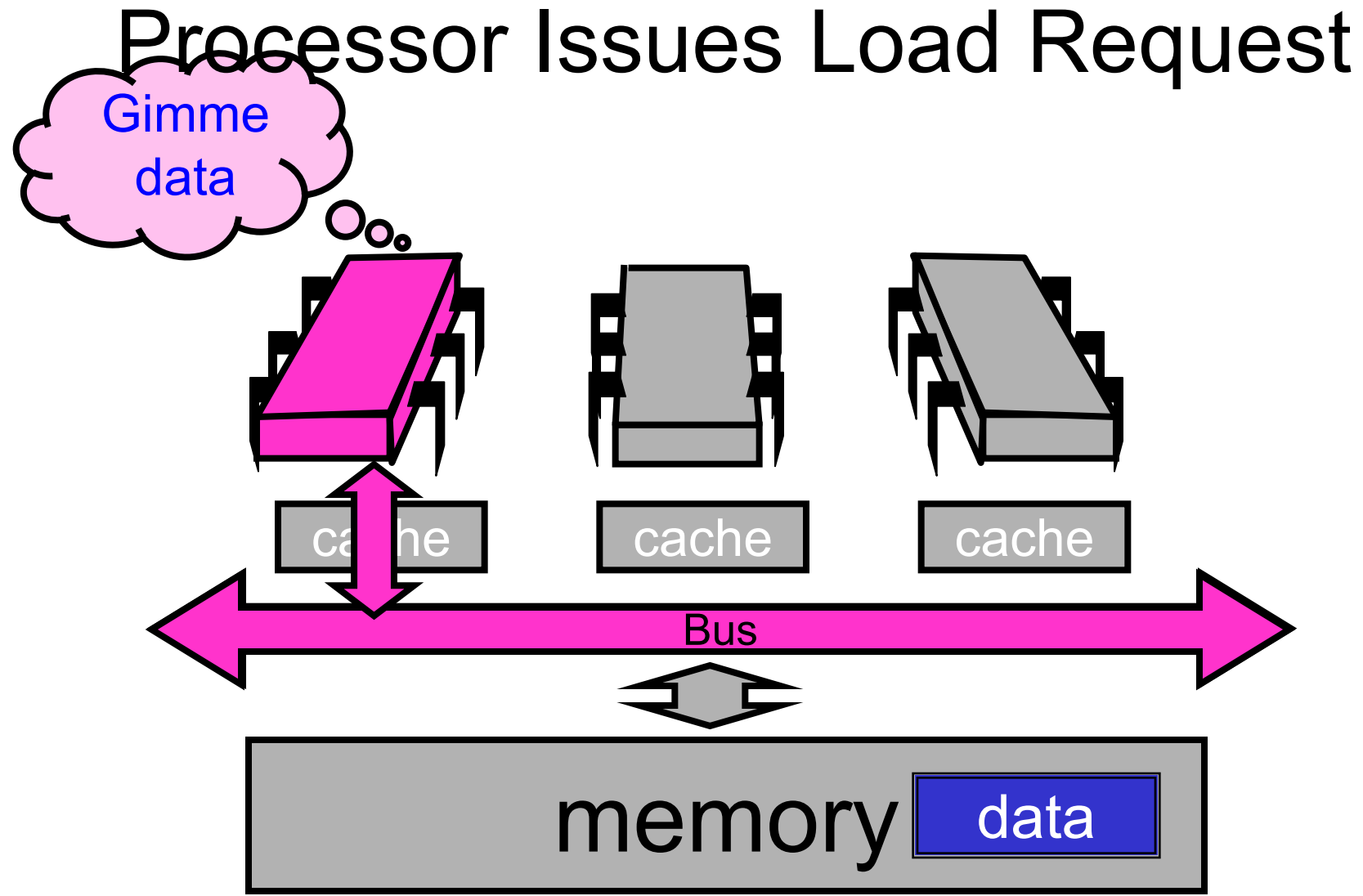
Cave Canem (=beware of a dog)

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

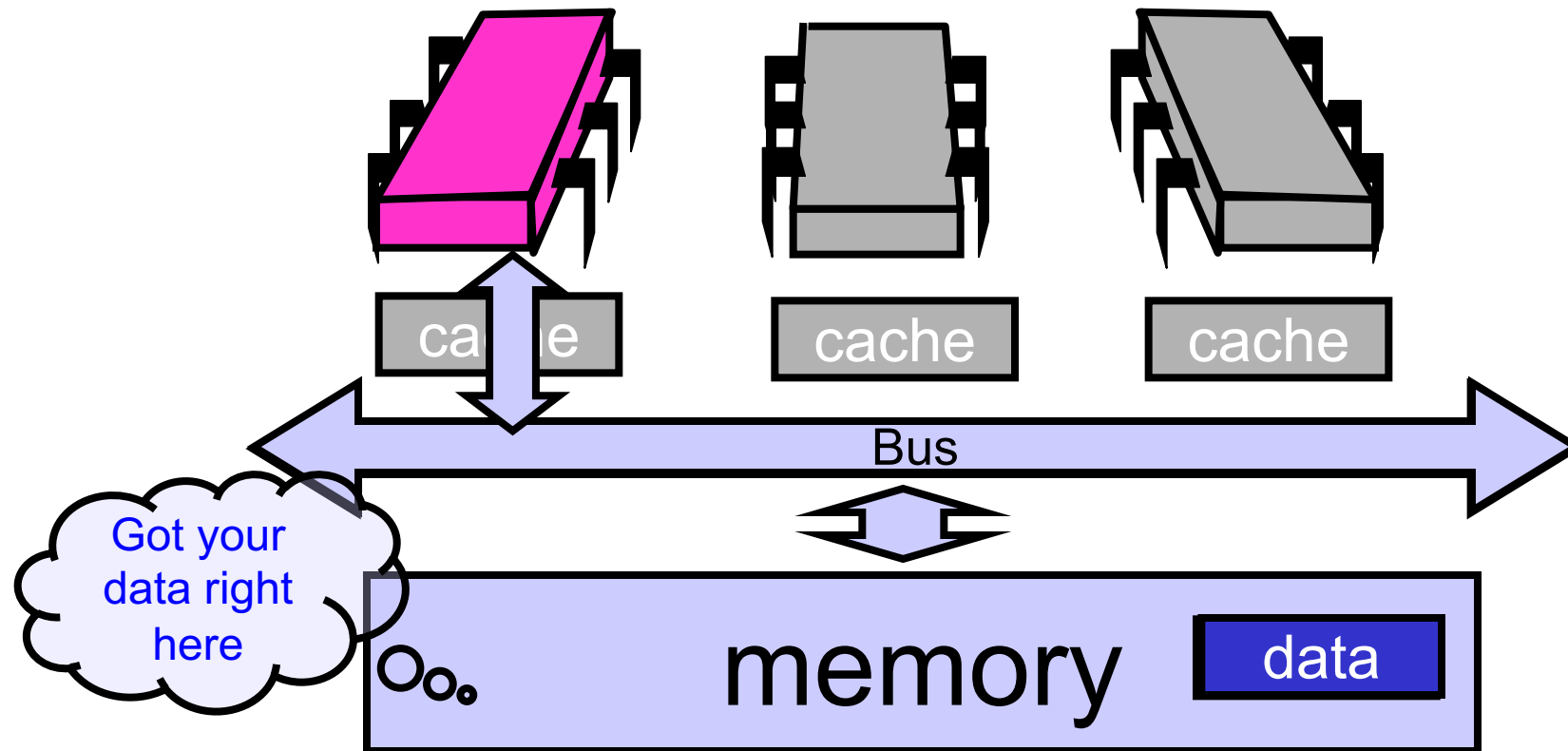


Processor Issues Load Request

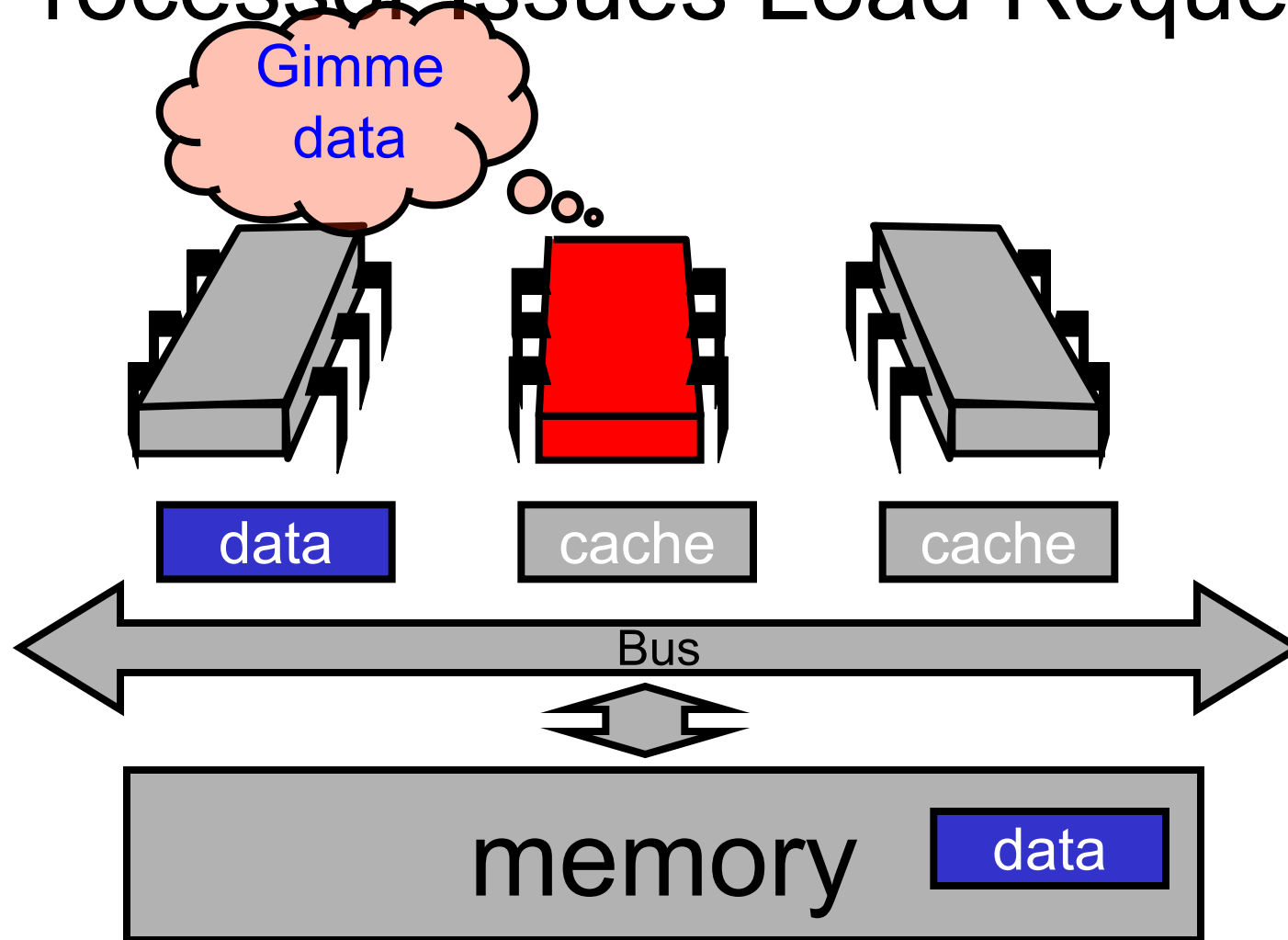




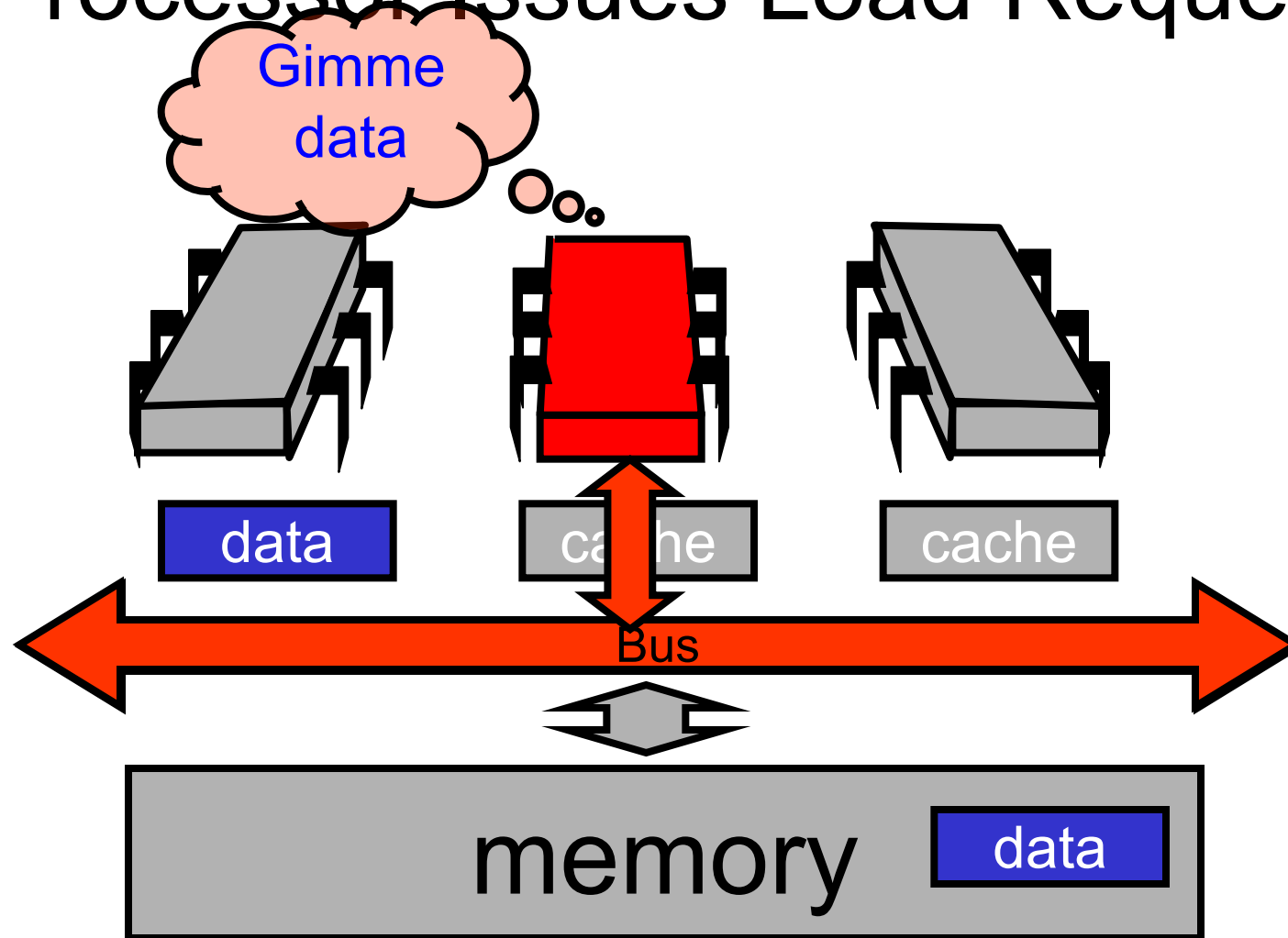
Memory Responds



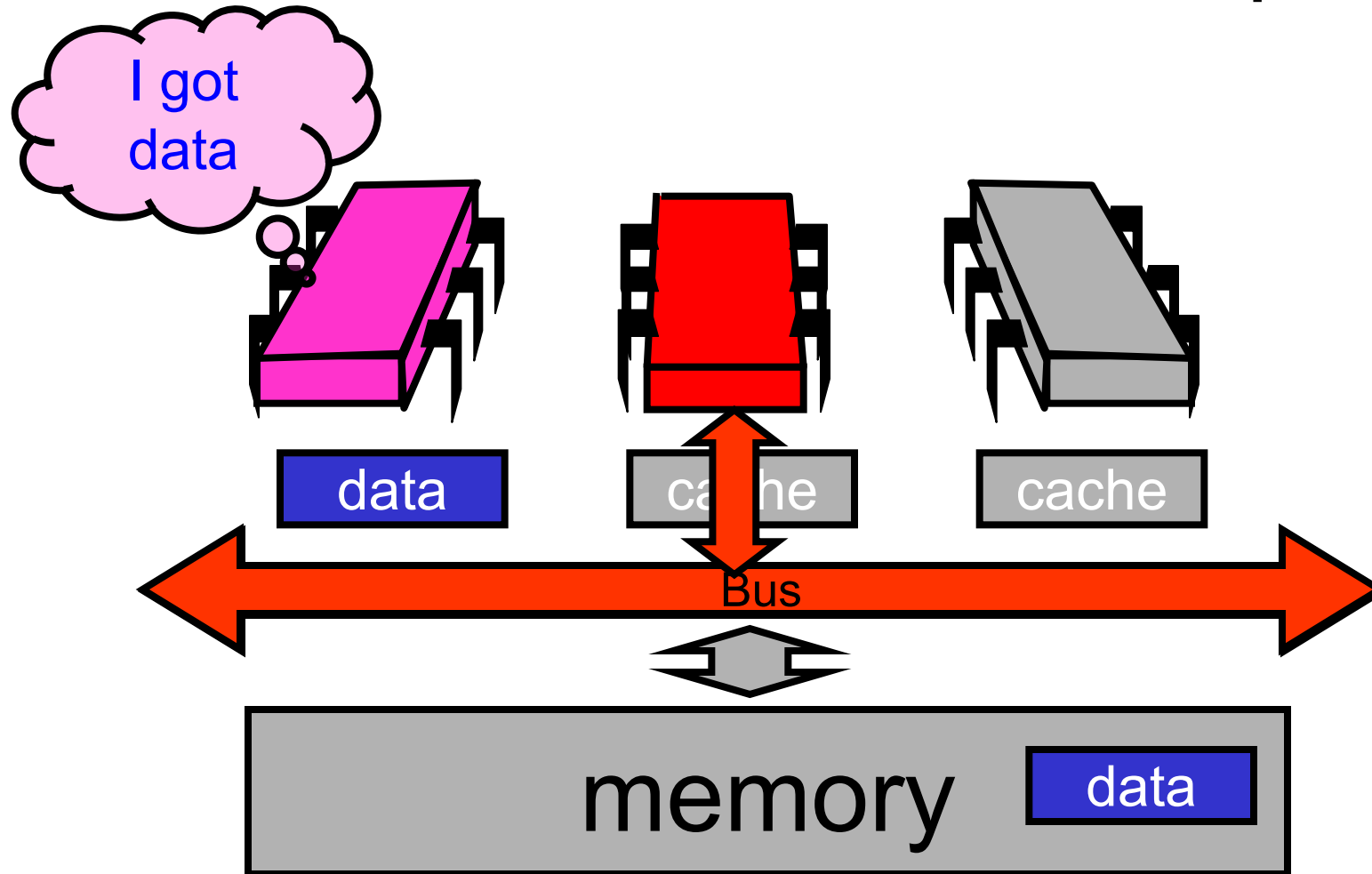
Processor Issues Load Request



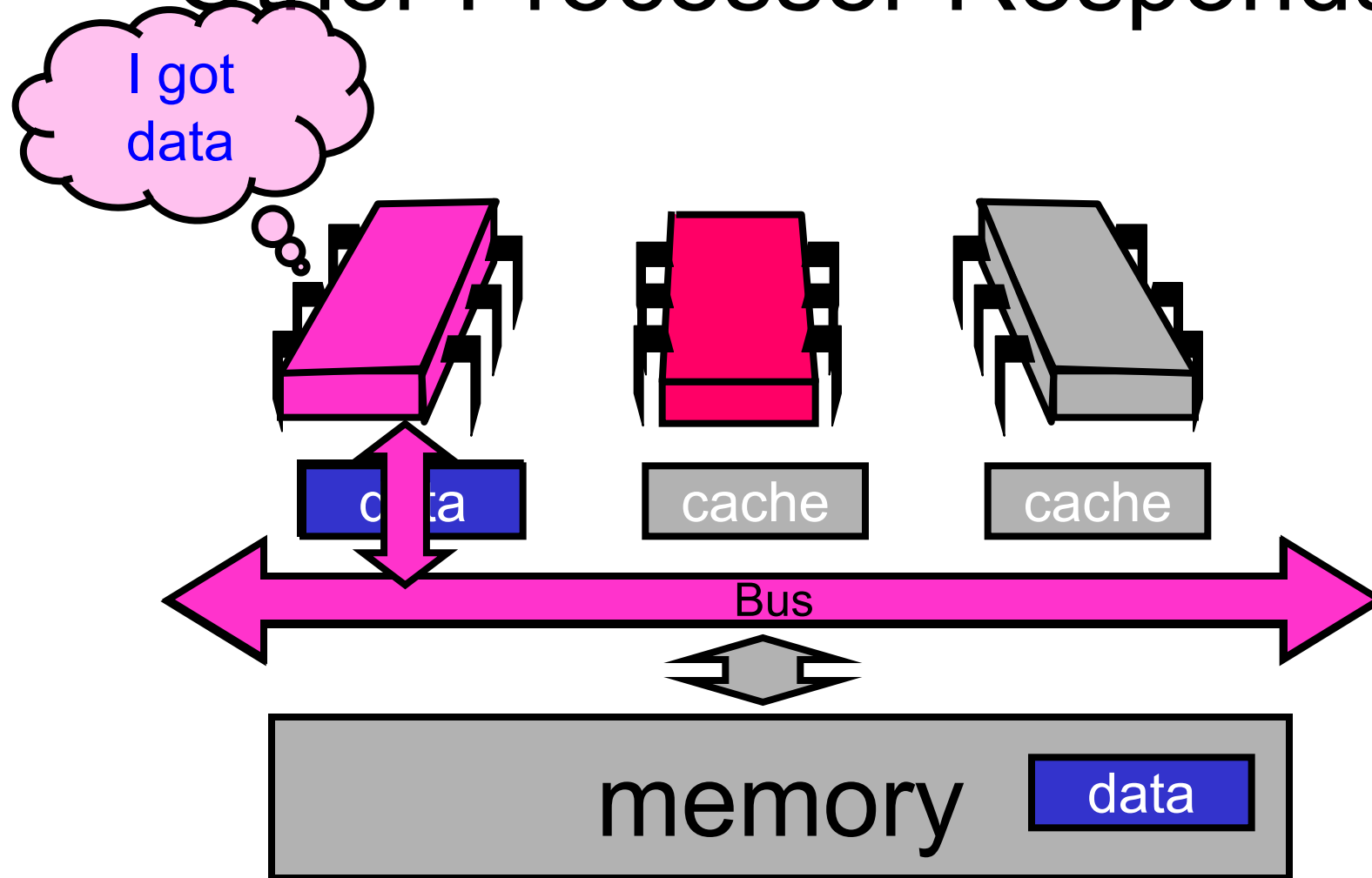
Processor Issues Load Request



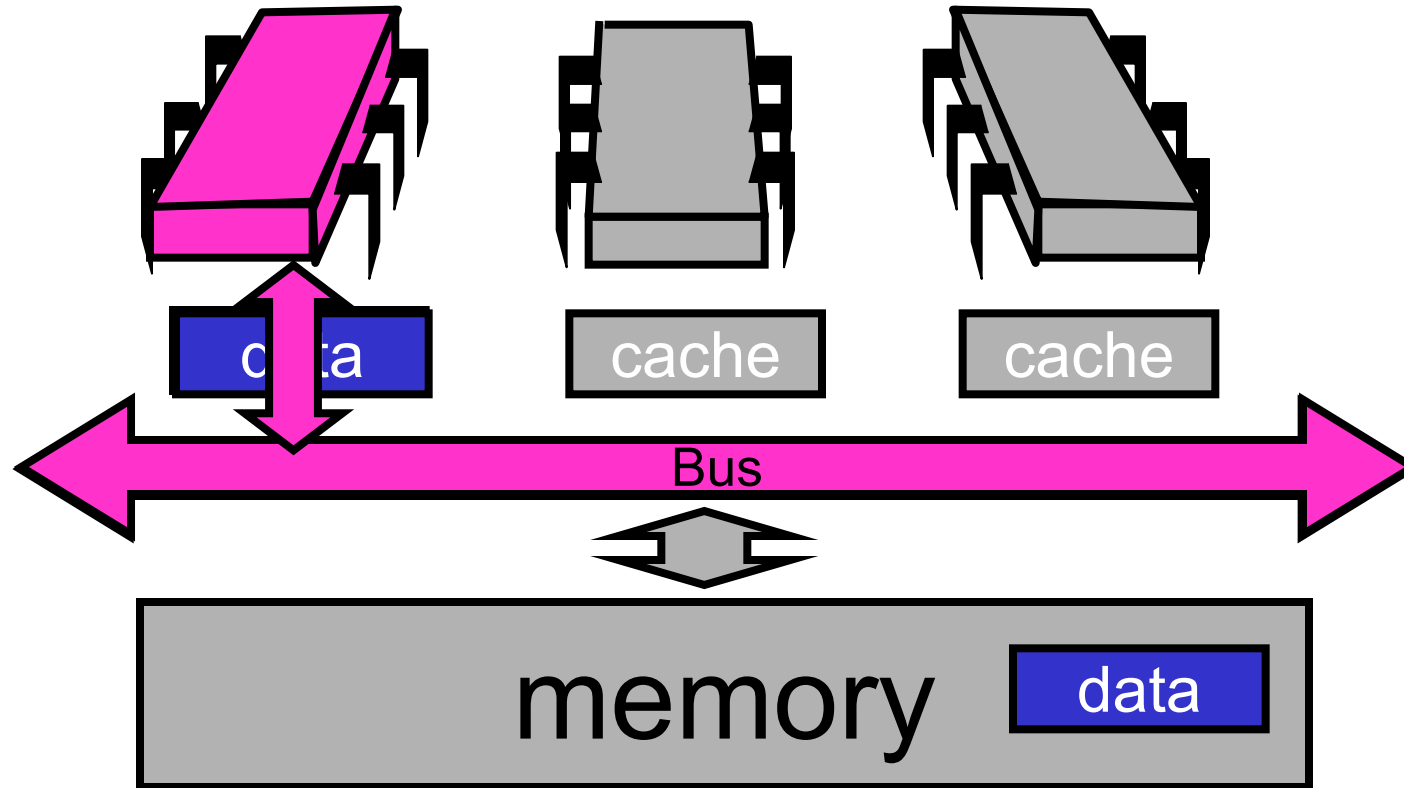
Processor Issues Load Request



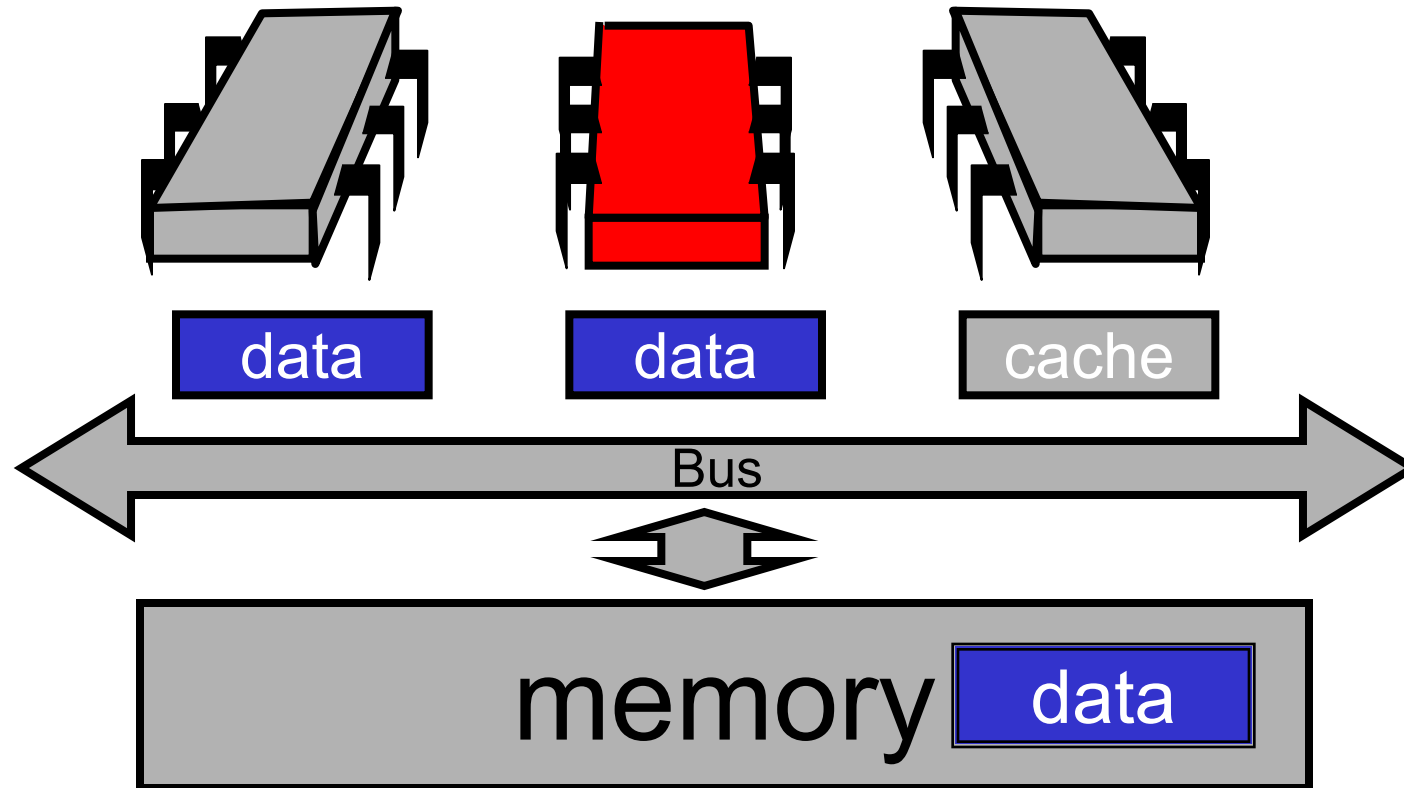
Other Processor Responds



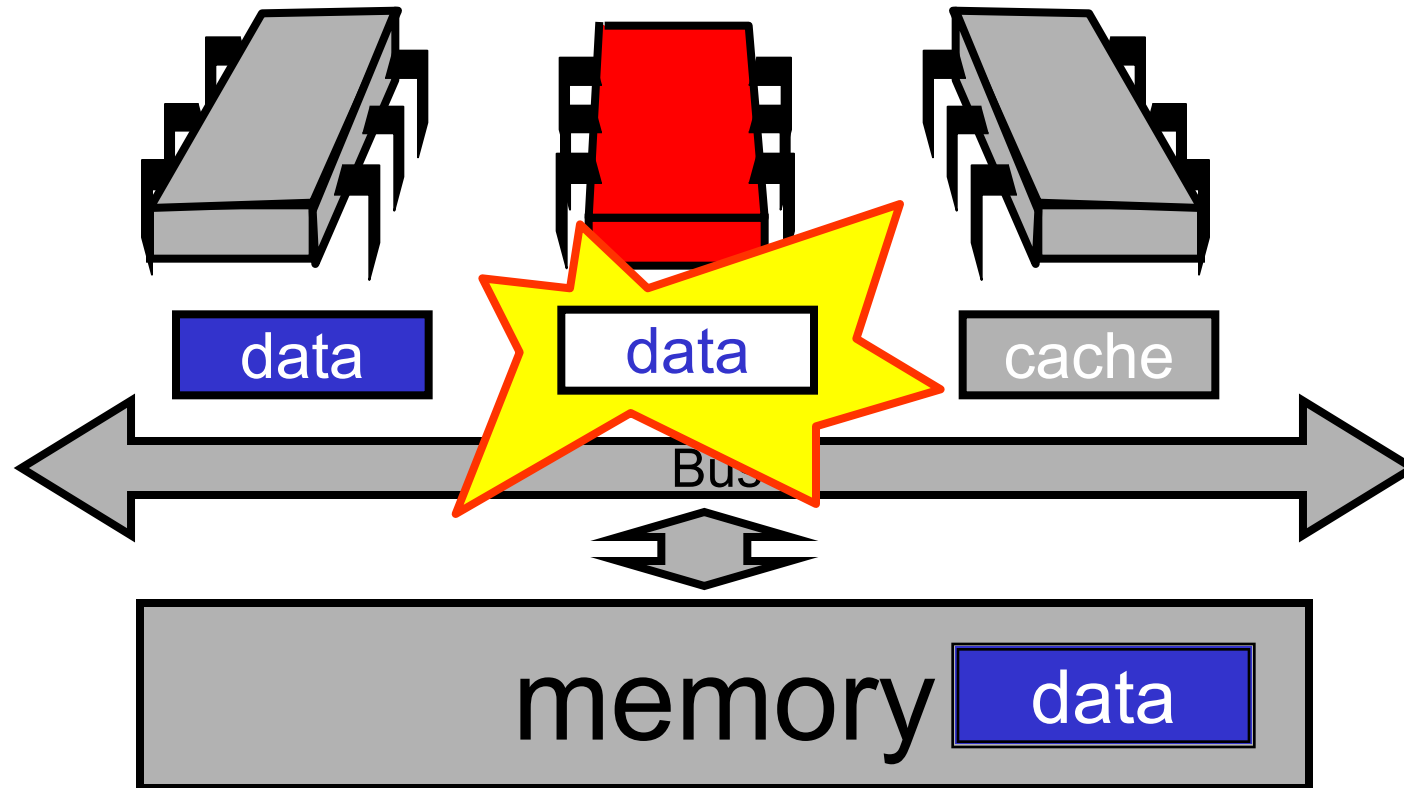
Other Processor Responds



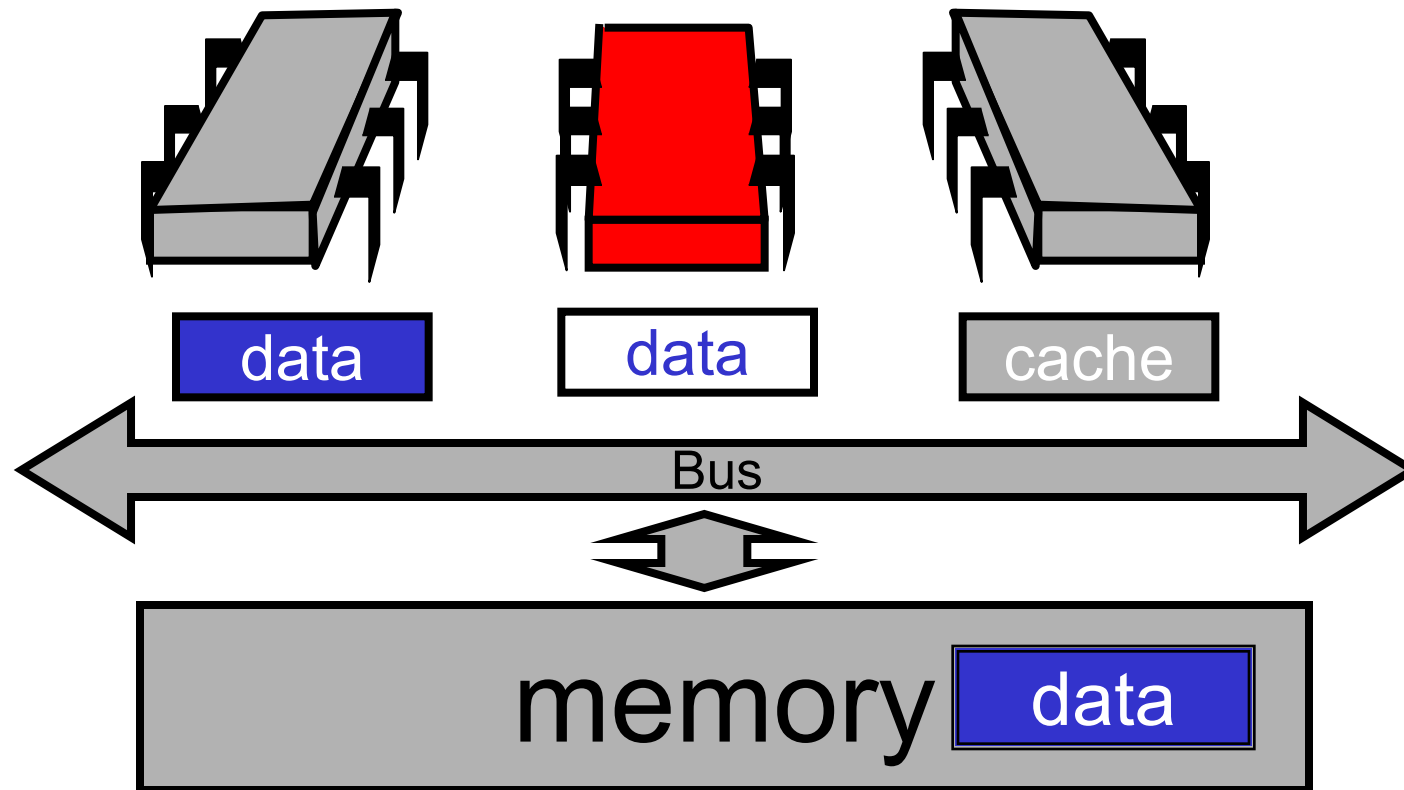
Modify Cached Data



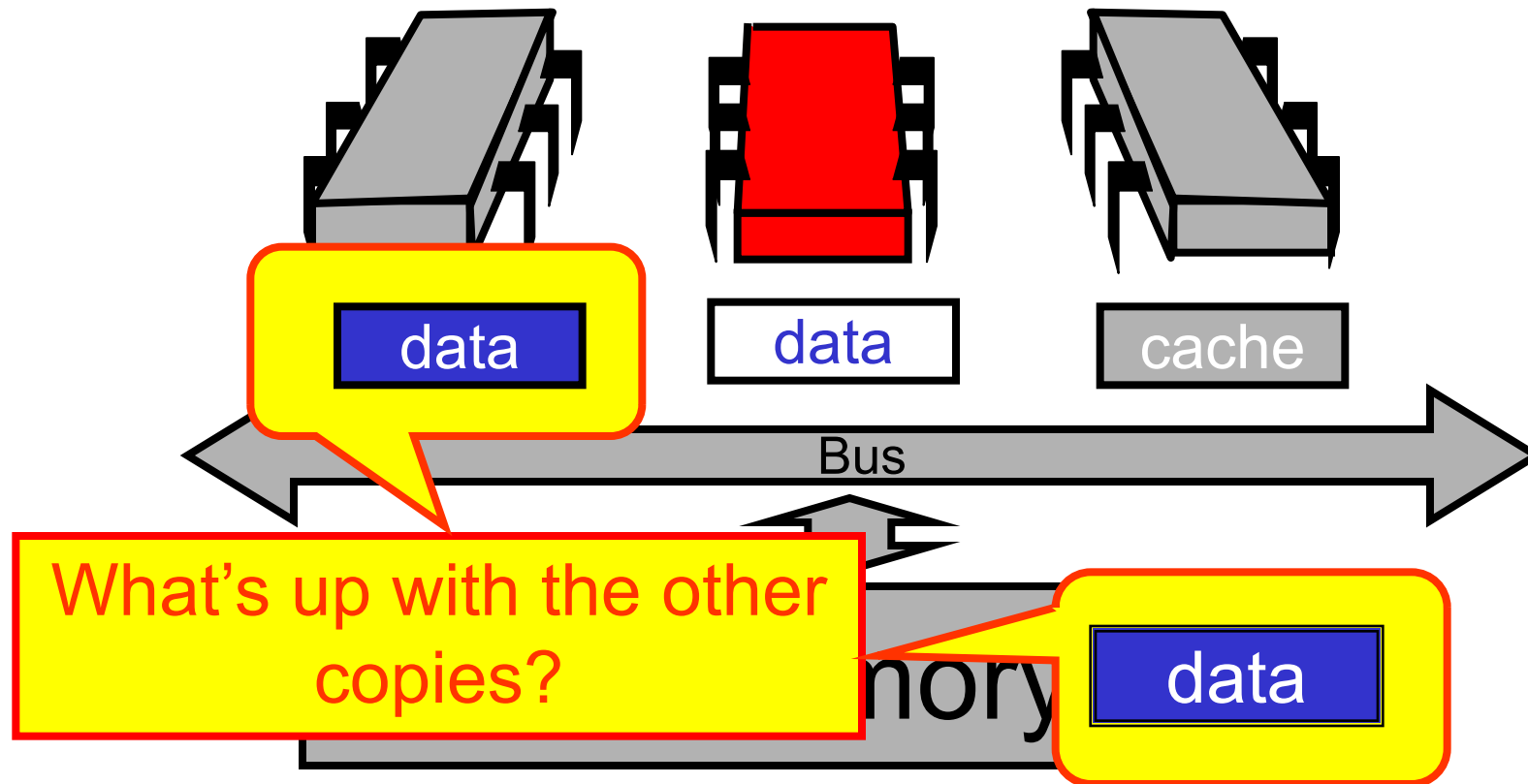
Modify Cached Data



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 ...

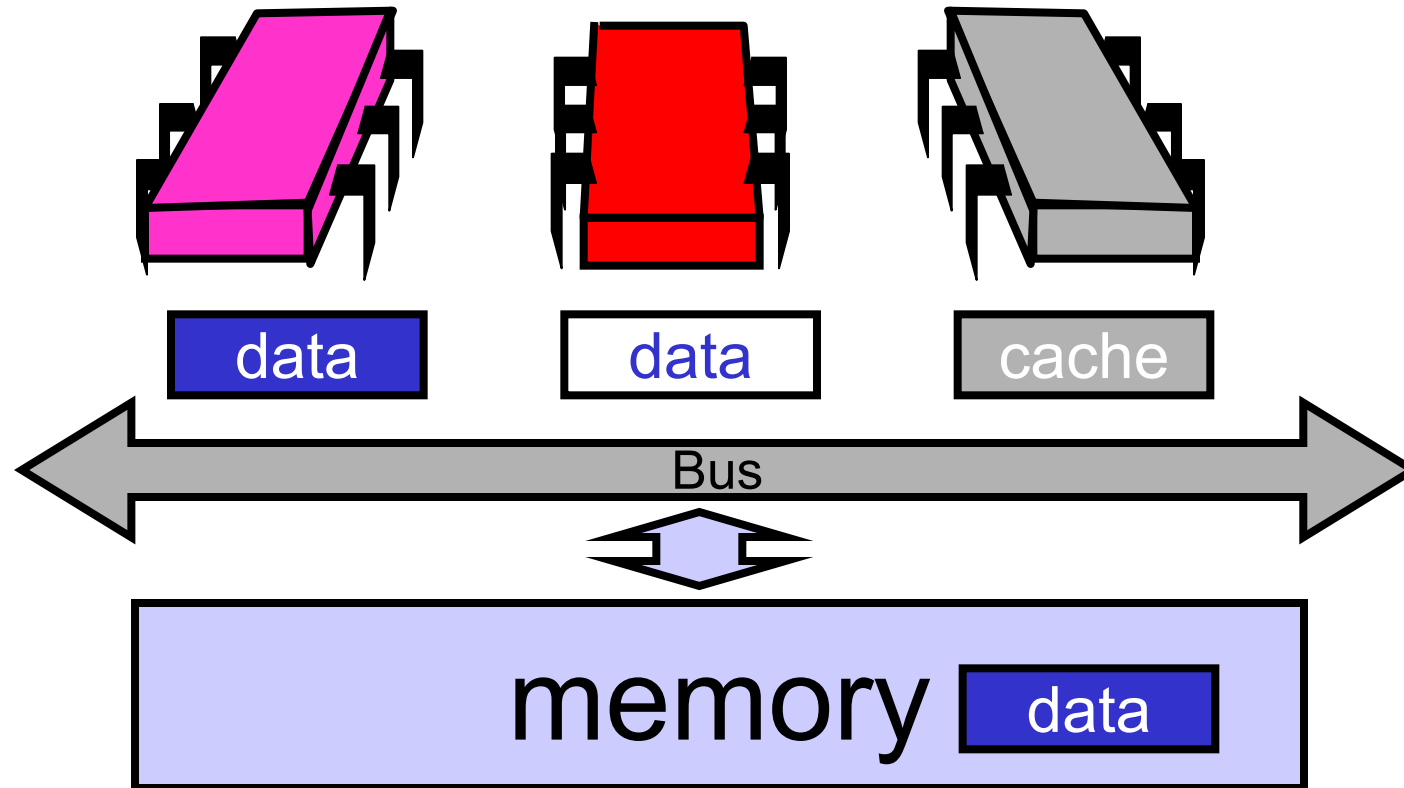


Write-Back Caches

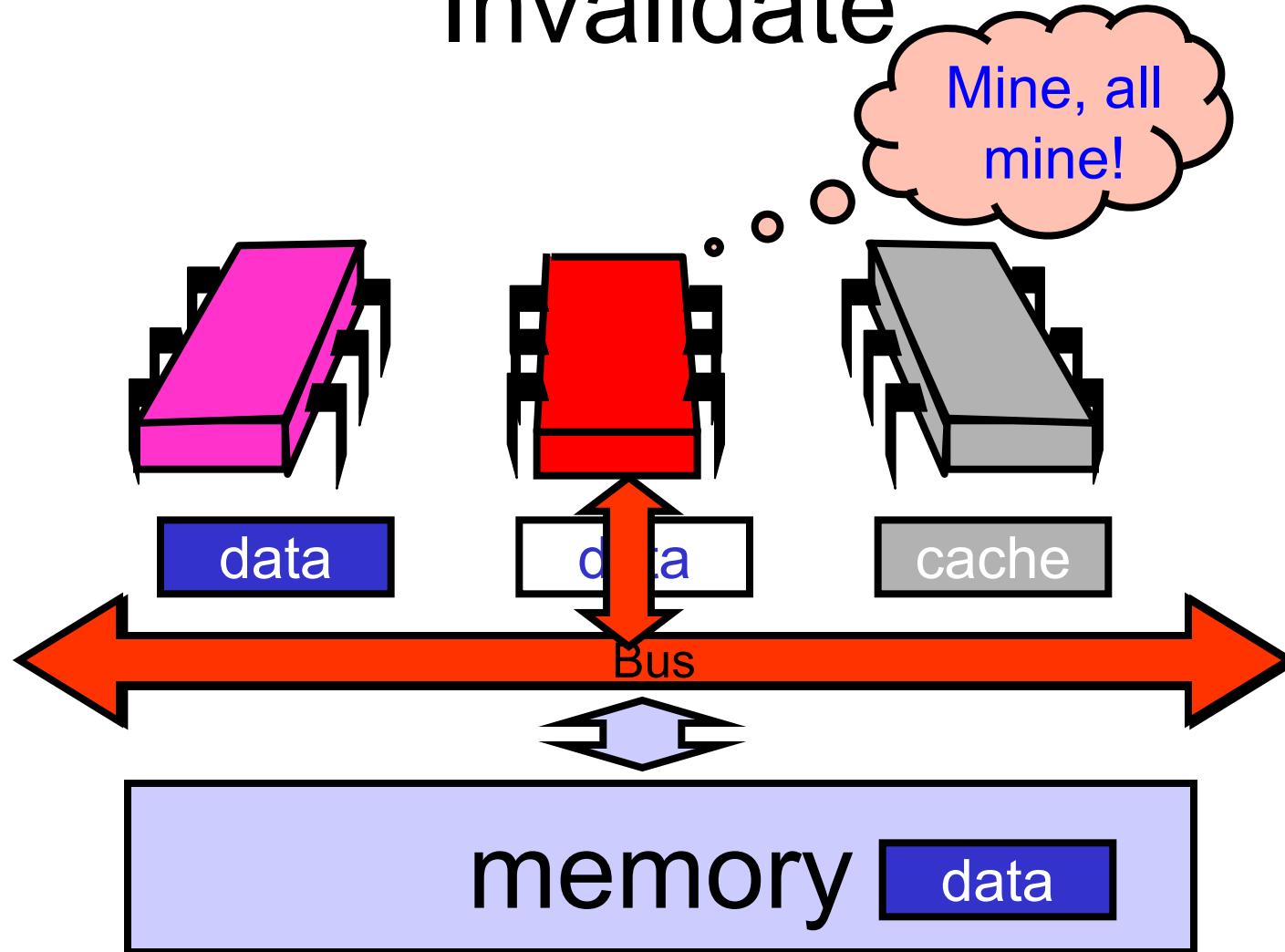
- Cache entry has three states
 - Invalid: contains raw seething bits
 - Valid: I can read but I can't write
 - Dirty: Data has been modified
 - Intercept other load requests
 - Write back to memory before using cache



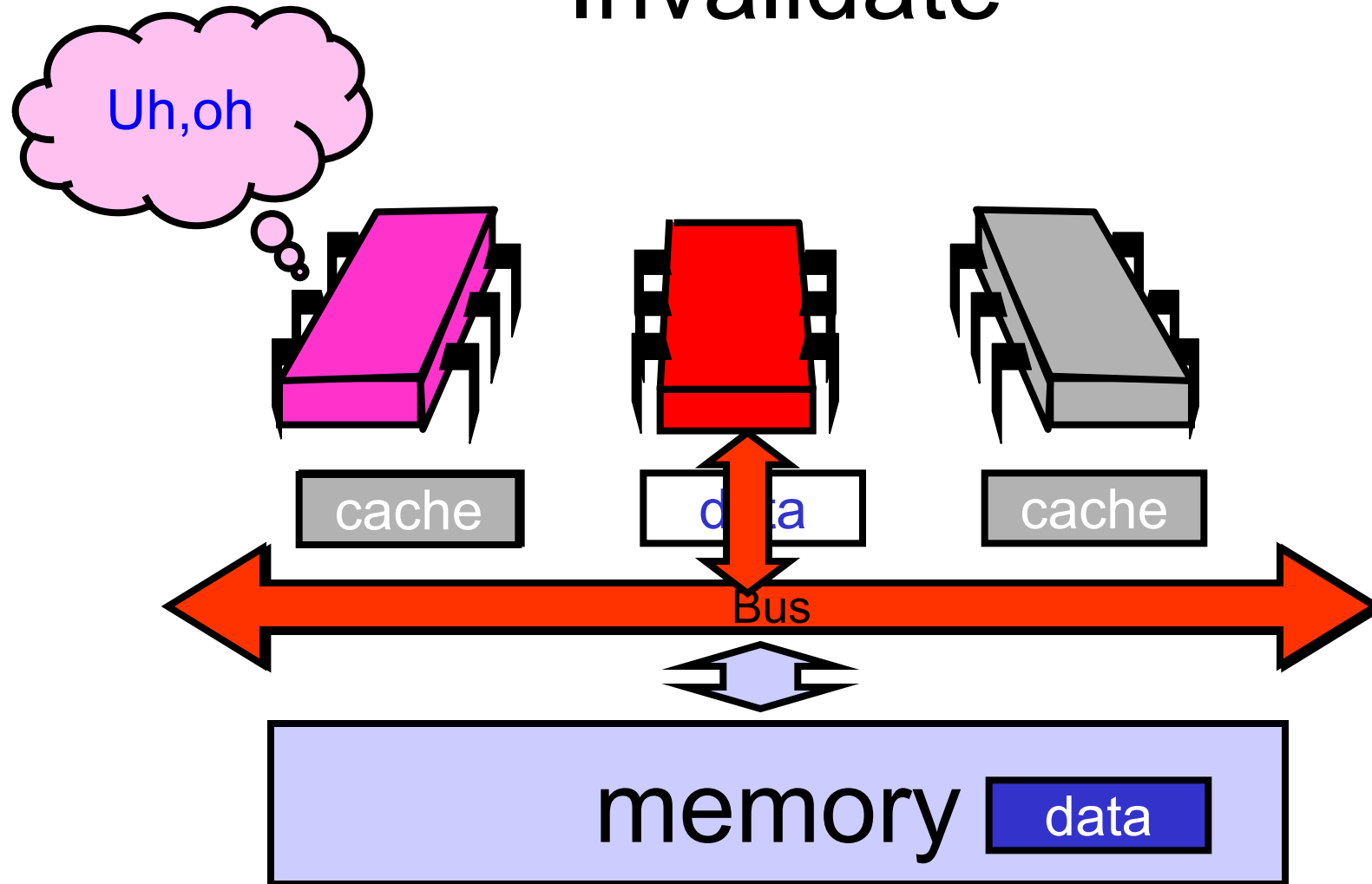
Invalidate



Invalidate

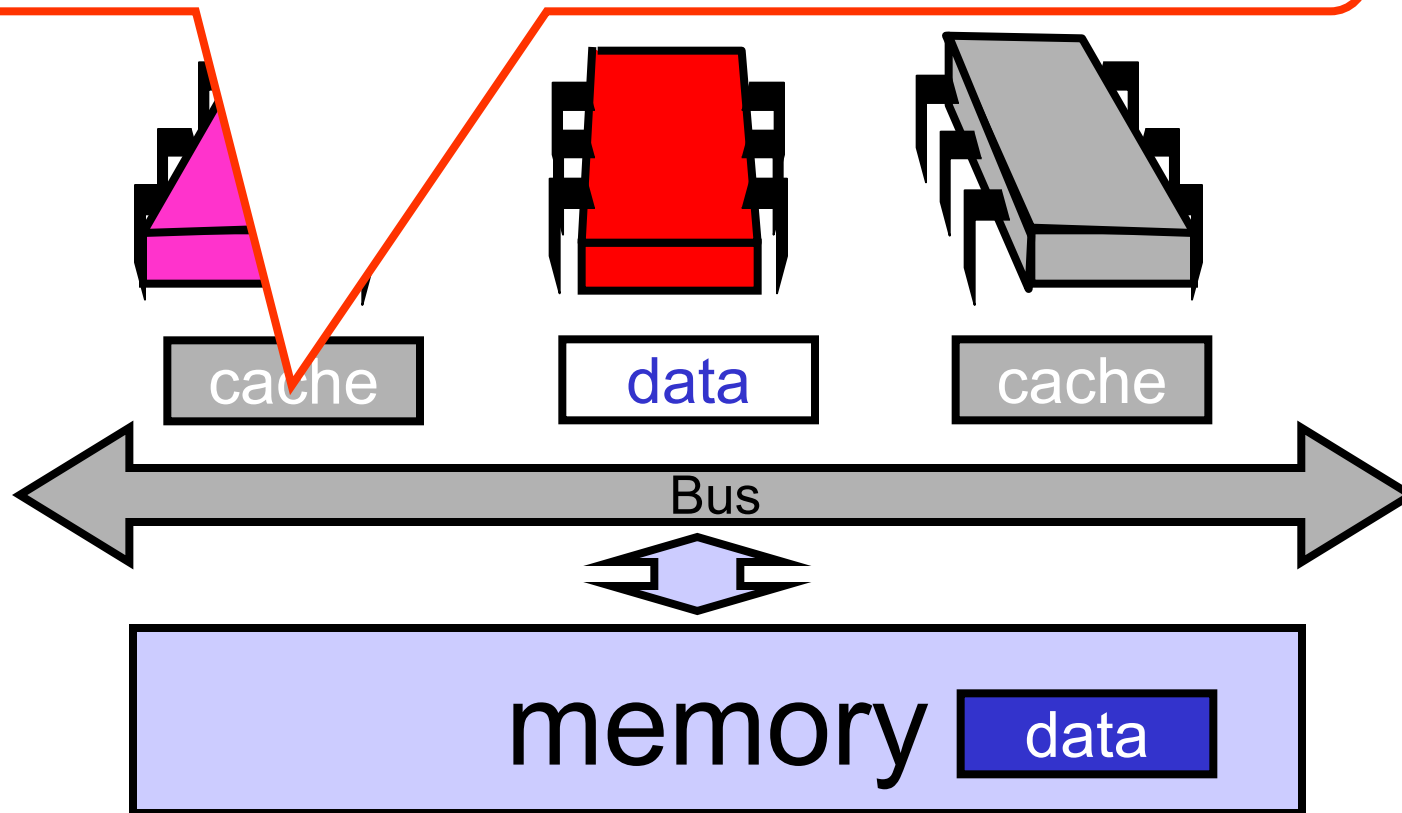


Invalidate



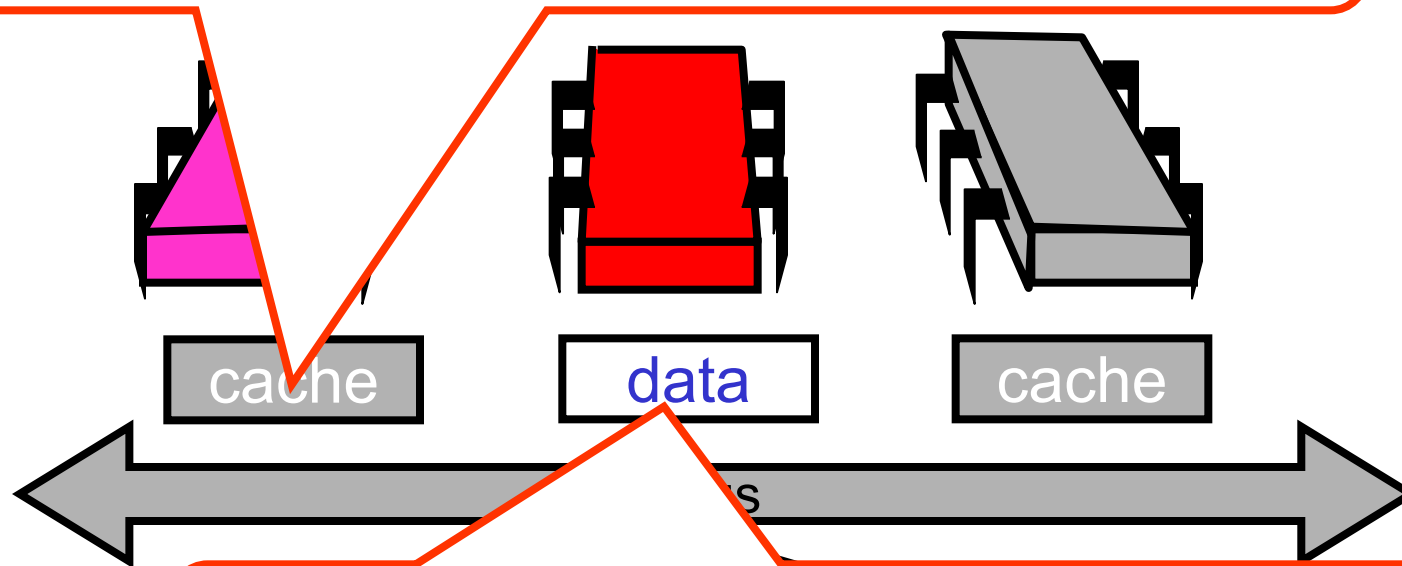
Invalidate

Other caches lose read permission



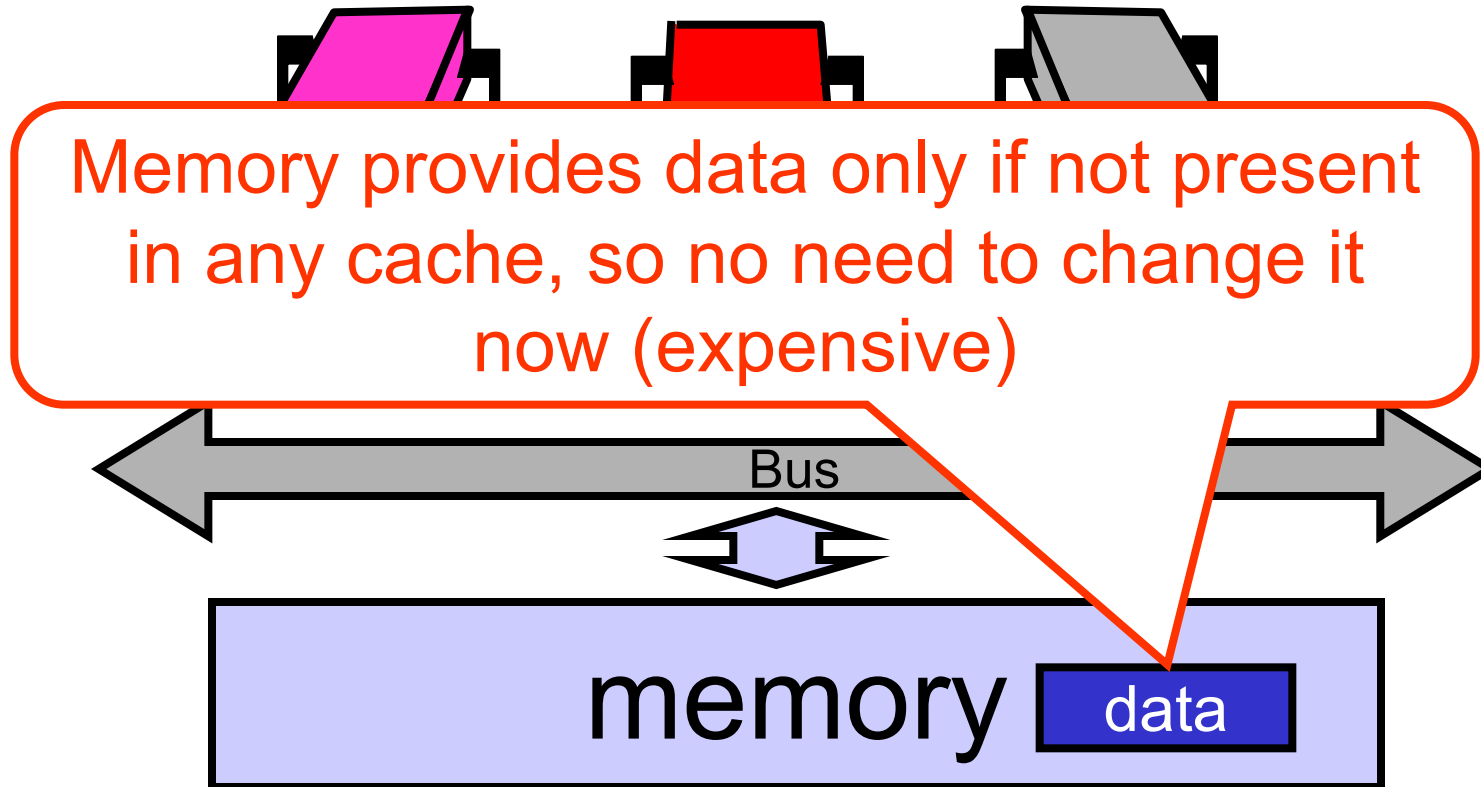
Invalidate

Other caches lose read permission

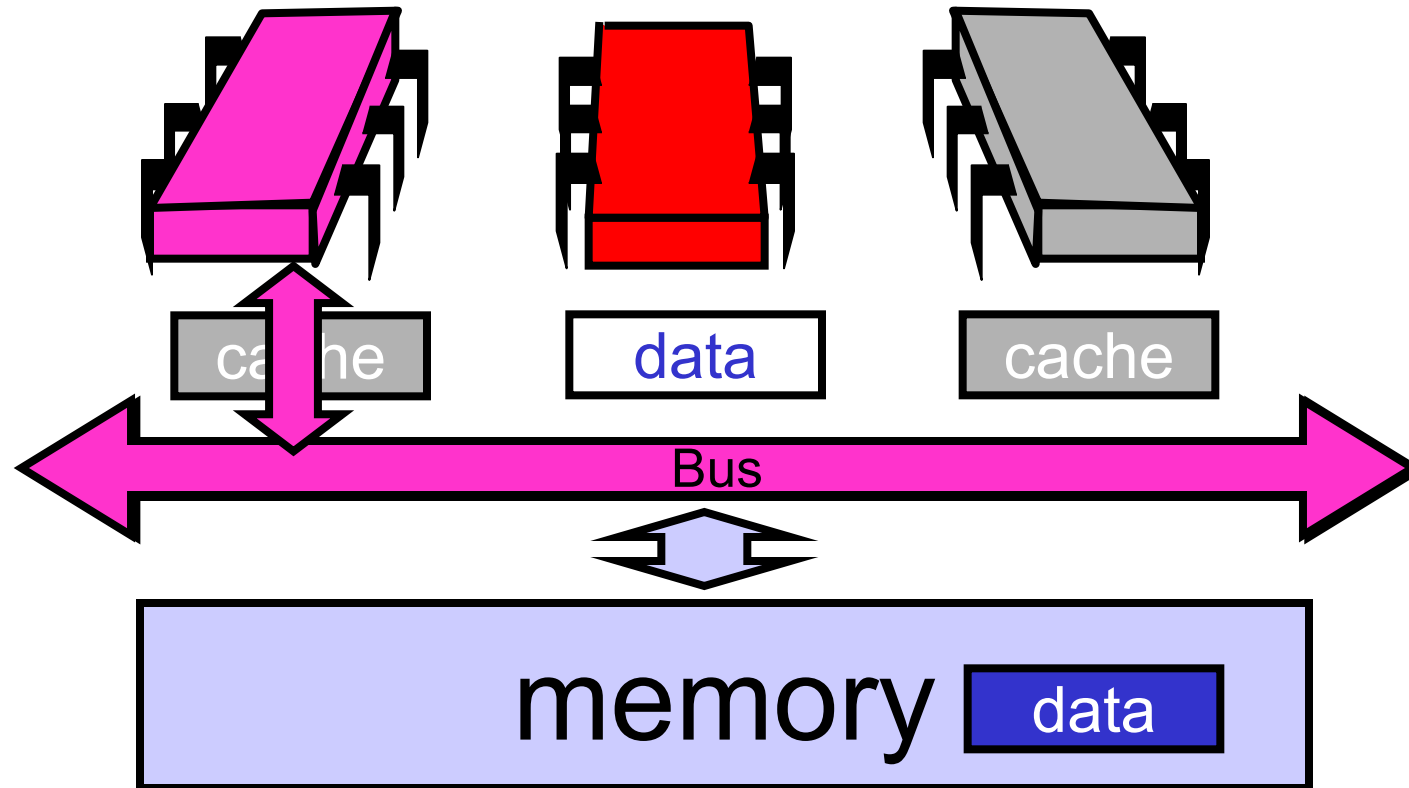


This cache acquires write permission

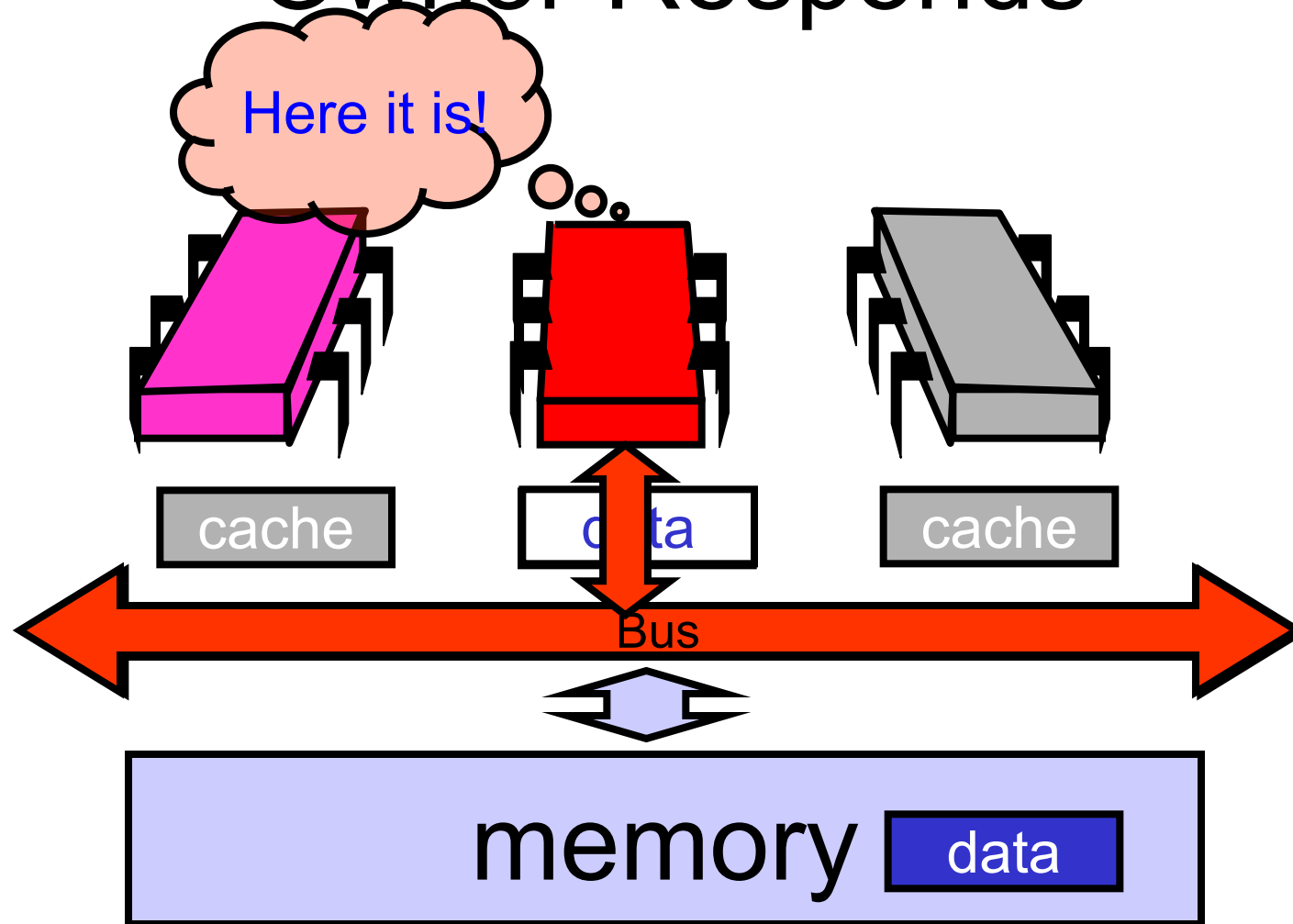
Invalidate



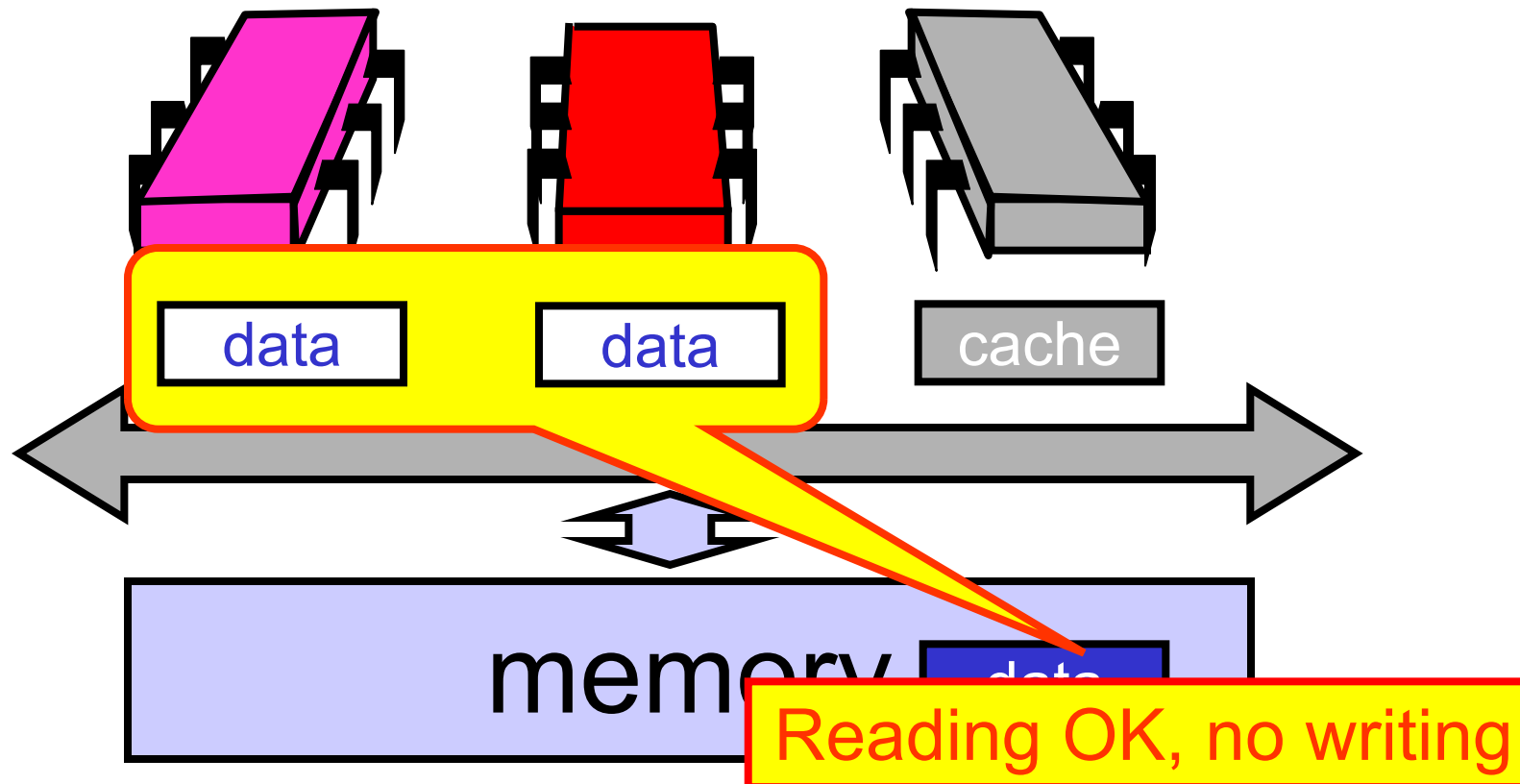
Another Processor Asks for Data



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

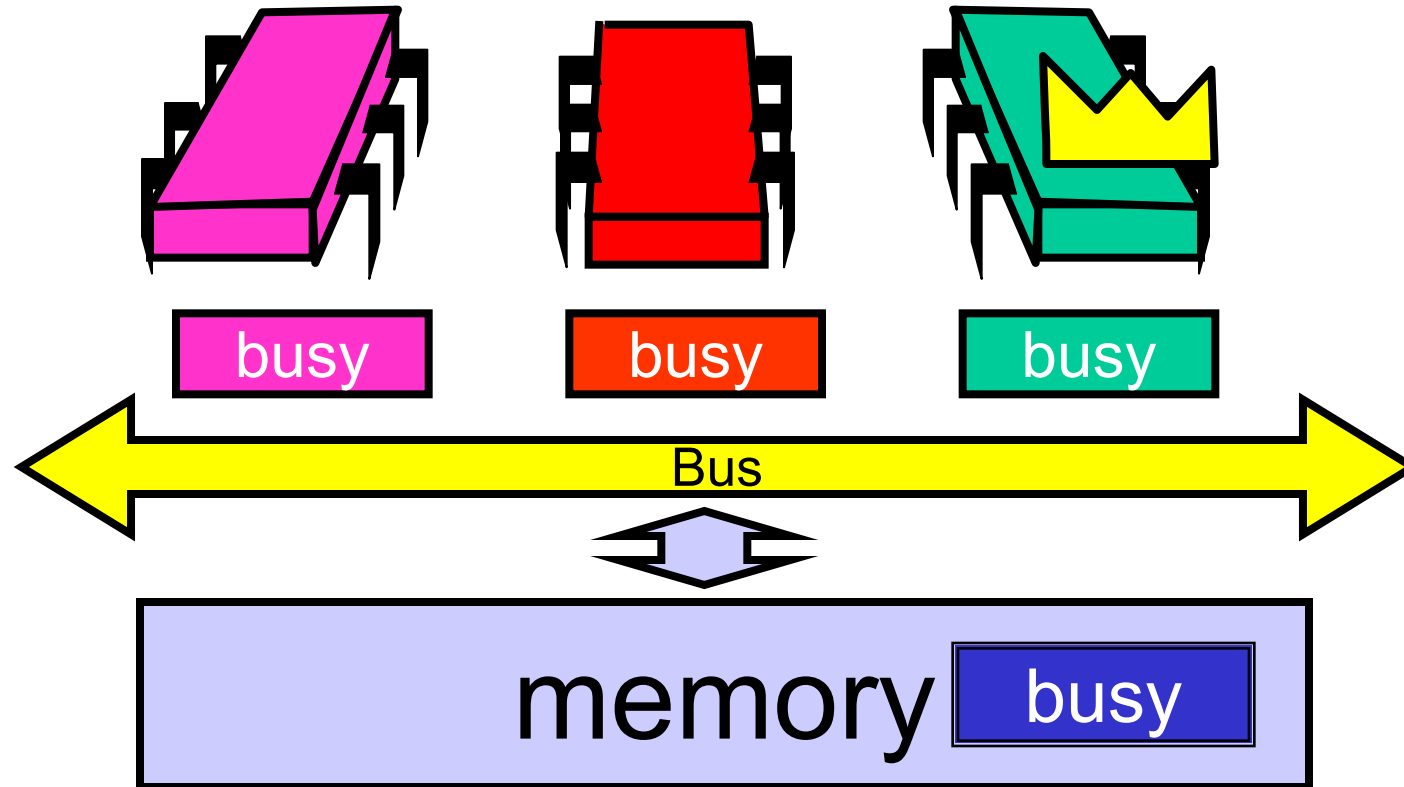


Test-and-test-and-set

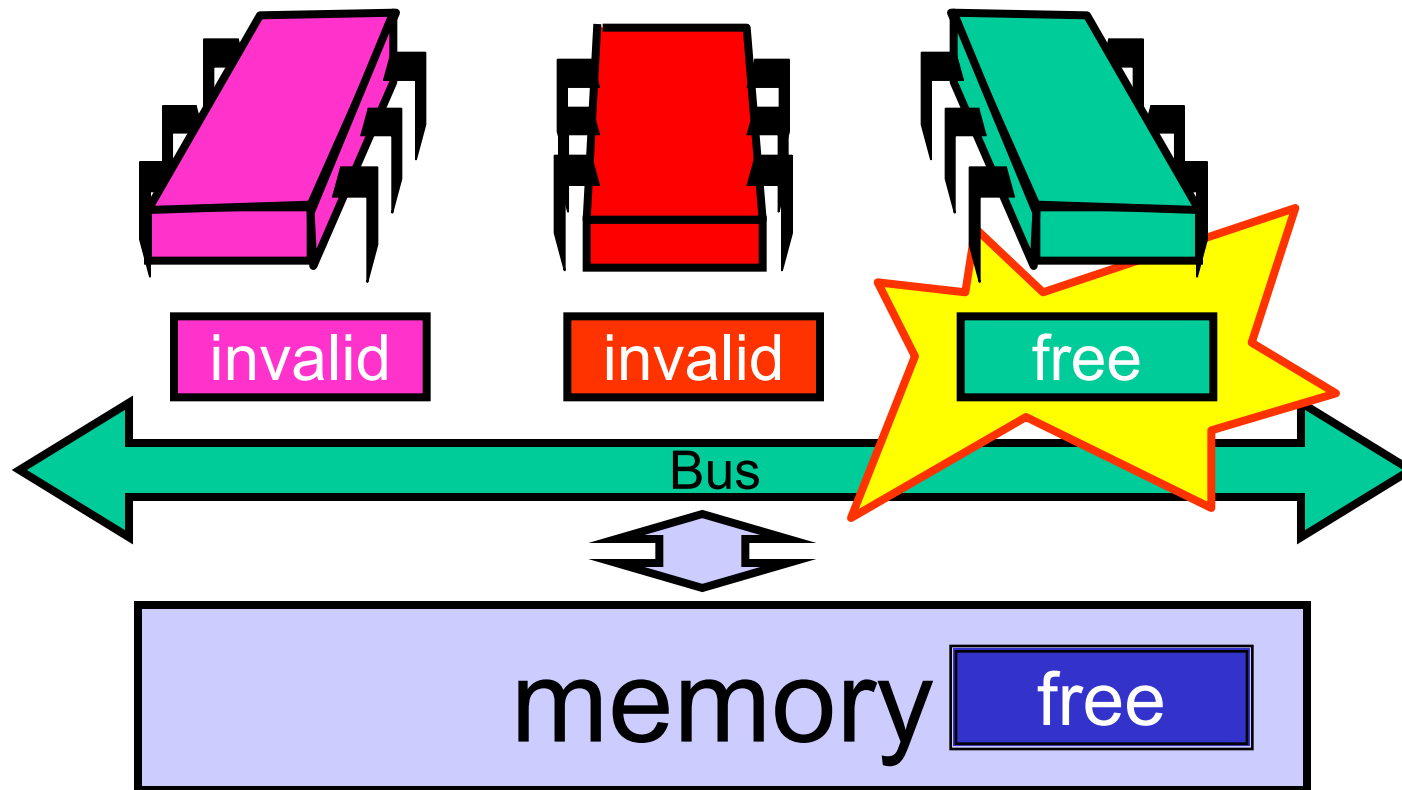
- Wait until lock “looks” free
 - Spin on local cache
 - No bus use while lock busy
- Problem: when lock is released
 - Invalidation storm ...



Local Spinning while Lock is Busy

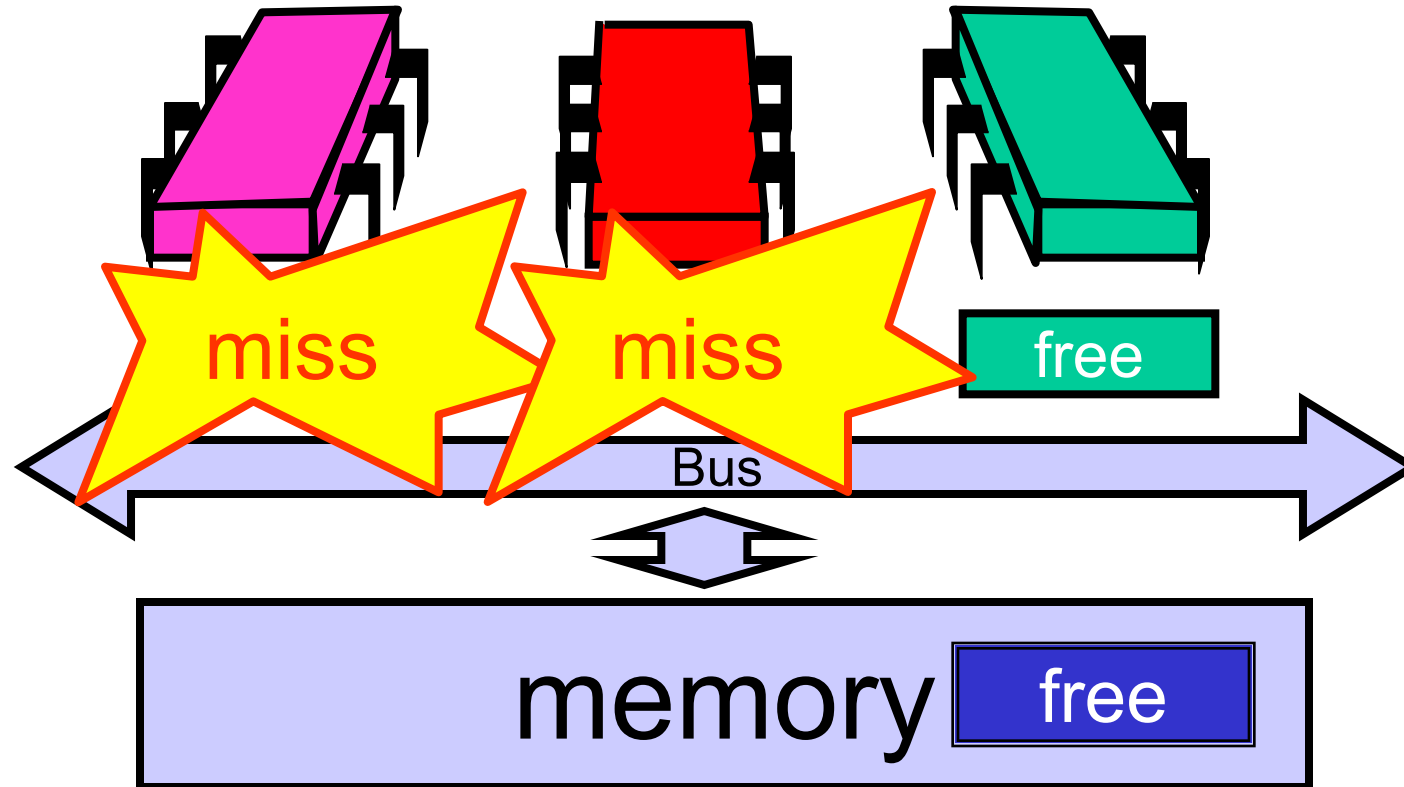


On Release



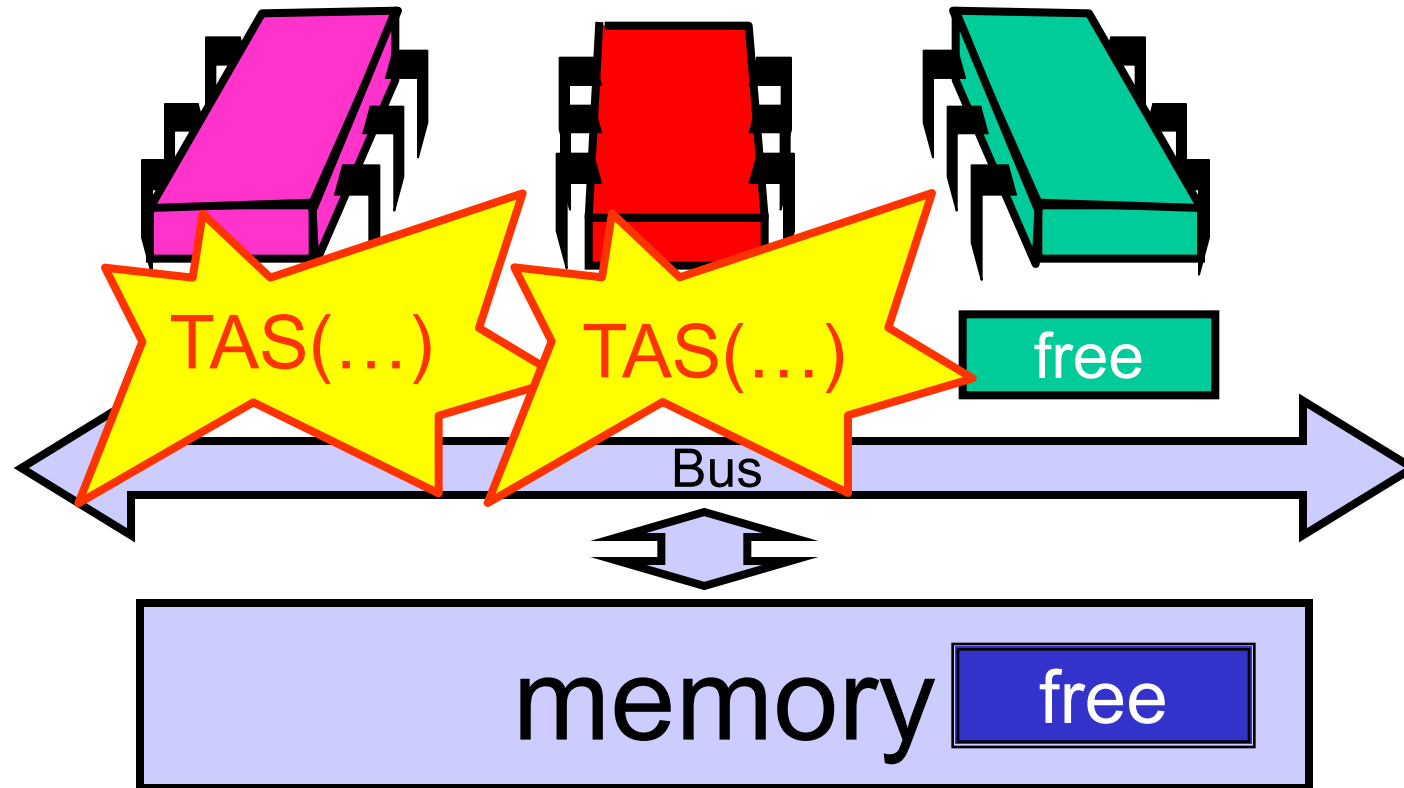
On Release

Everyone misses,
rereads



On Release

Everyone tries TAS



Problems

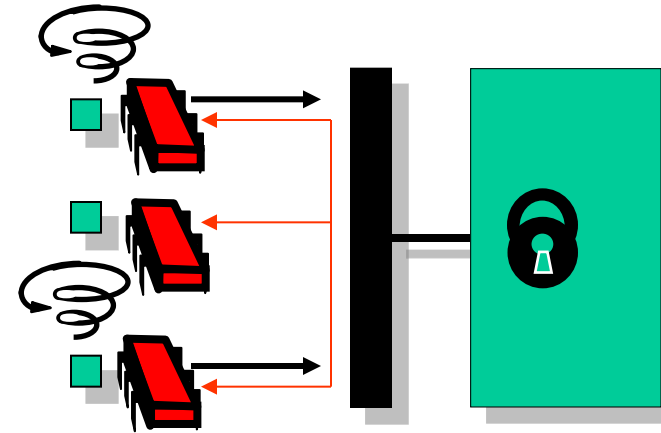
- Everyone misses
 - Reads satisfied sequentially
- Everyone does TAS
 - Invalidates others' caches
- Eventually quiesces after lock acquired
 - How long does this take?



Measuring Quiescence Time

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

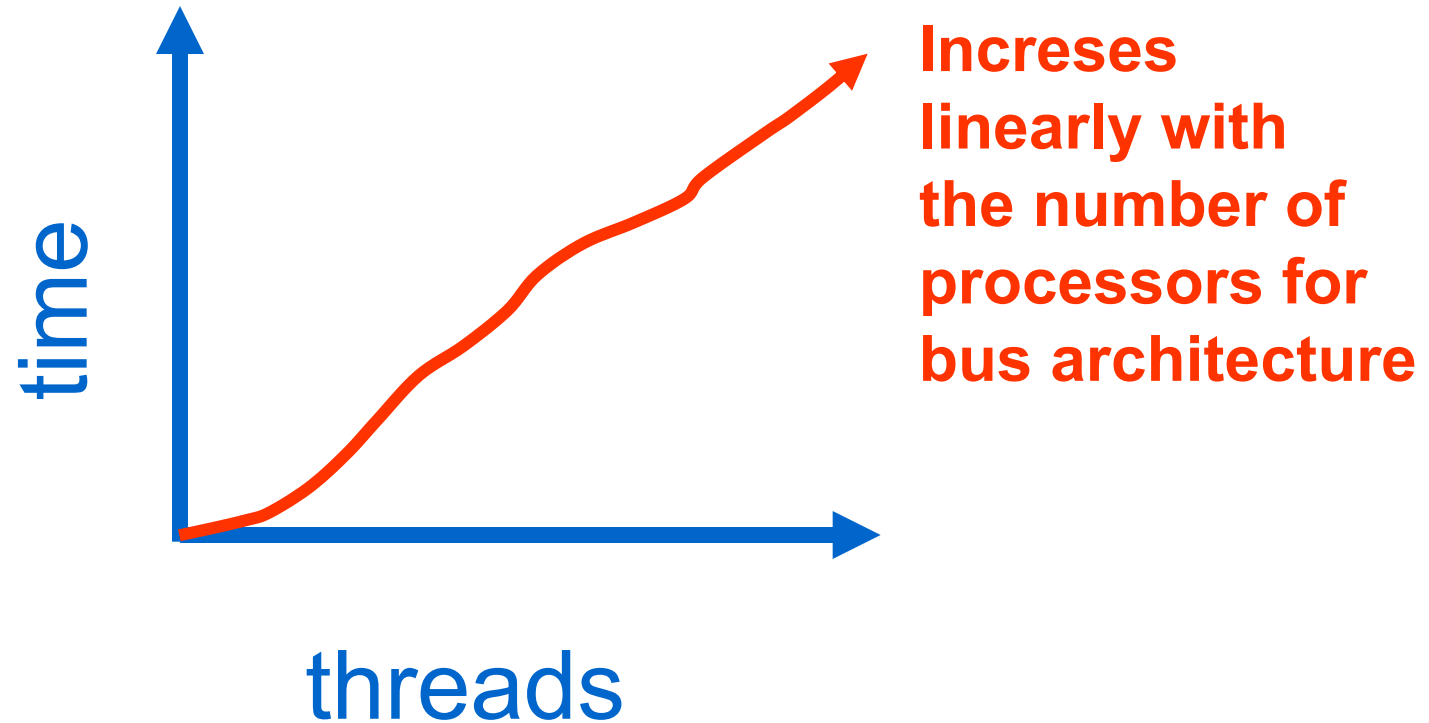
Y = time of ops that cause intensive bus traffic



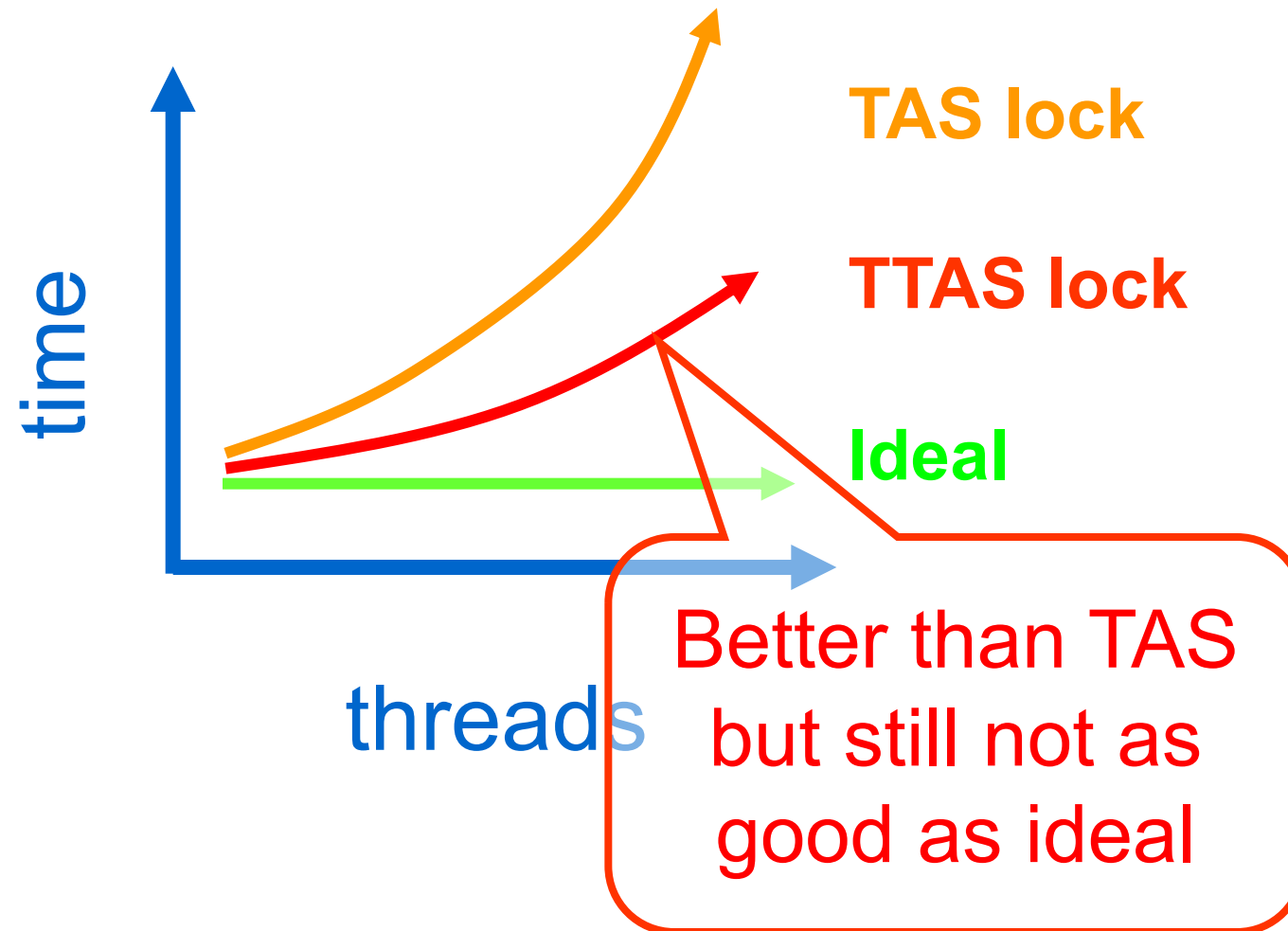
In critical section, run ops X then ops Y . As long as Quiescence time is less than X , no drop in performance.

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

Quiescence Time

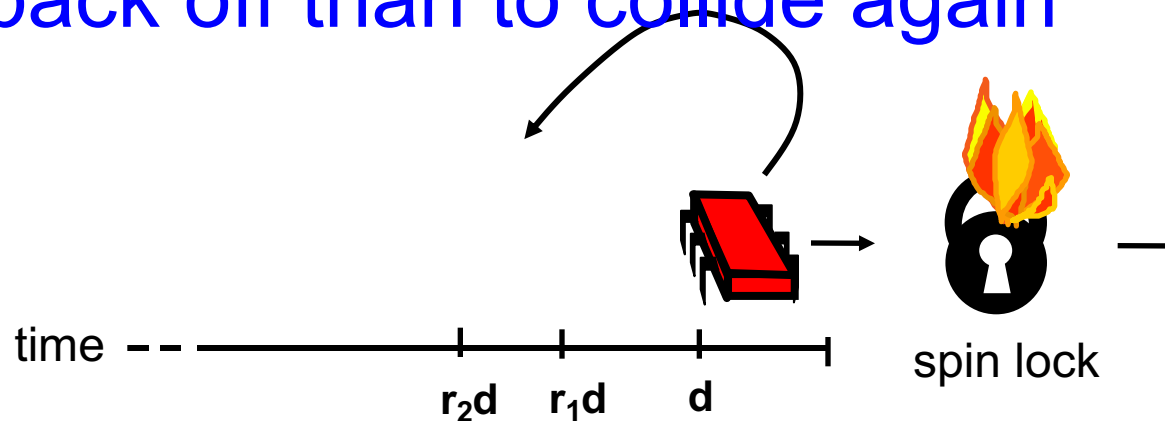


Mystery Explained

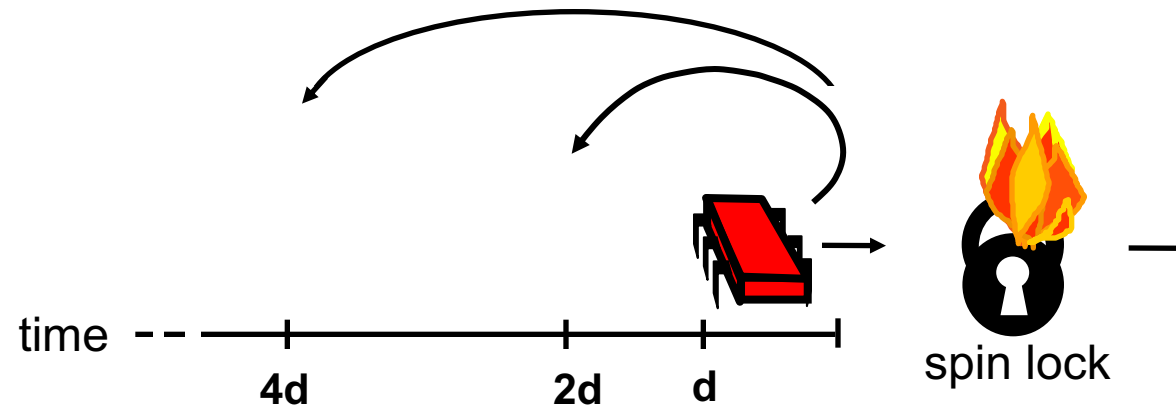


Solution: Introduce Delay

- 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

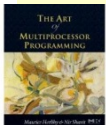
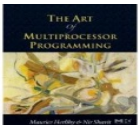
```
public class Backoff implements Lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)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

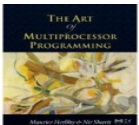
Fix minimum delay



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;  
        }  
    }  
}
```

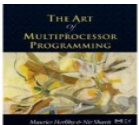
Wait until lock looks free



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

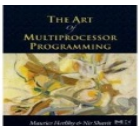
```
public class Backoff implements Lock {  
    public Back off for random duration  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get() != LOCKED) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
  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



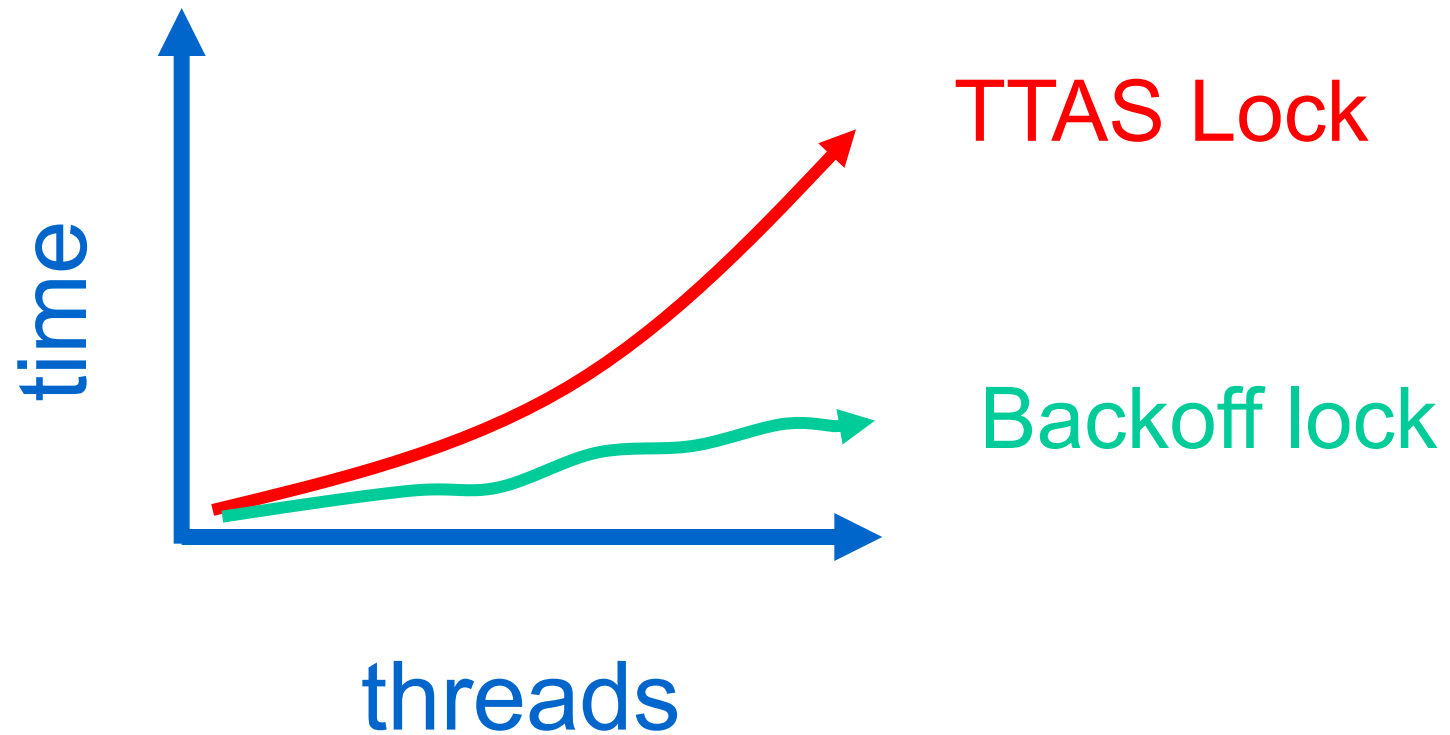
Exponential Backoff Lock

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

Double max delay, within reason



Spin-Waiting Overhead



Backoff: Other Issues

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

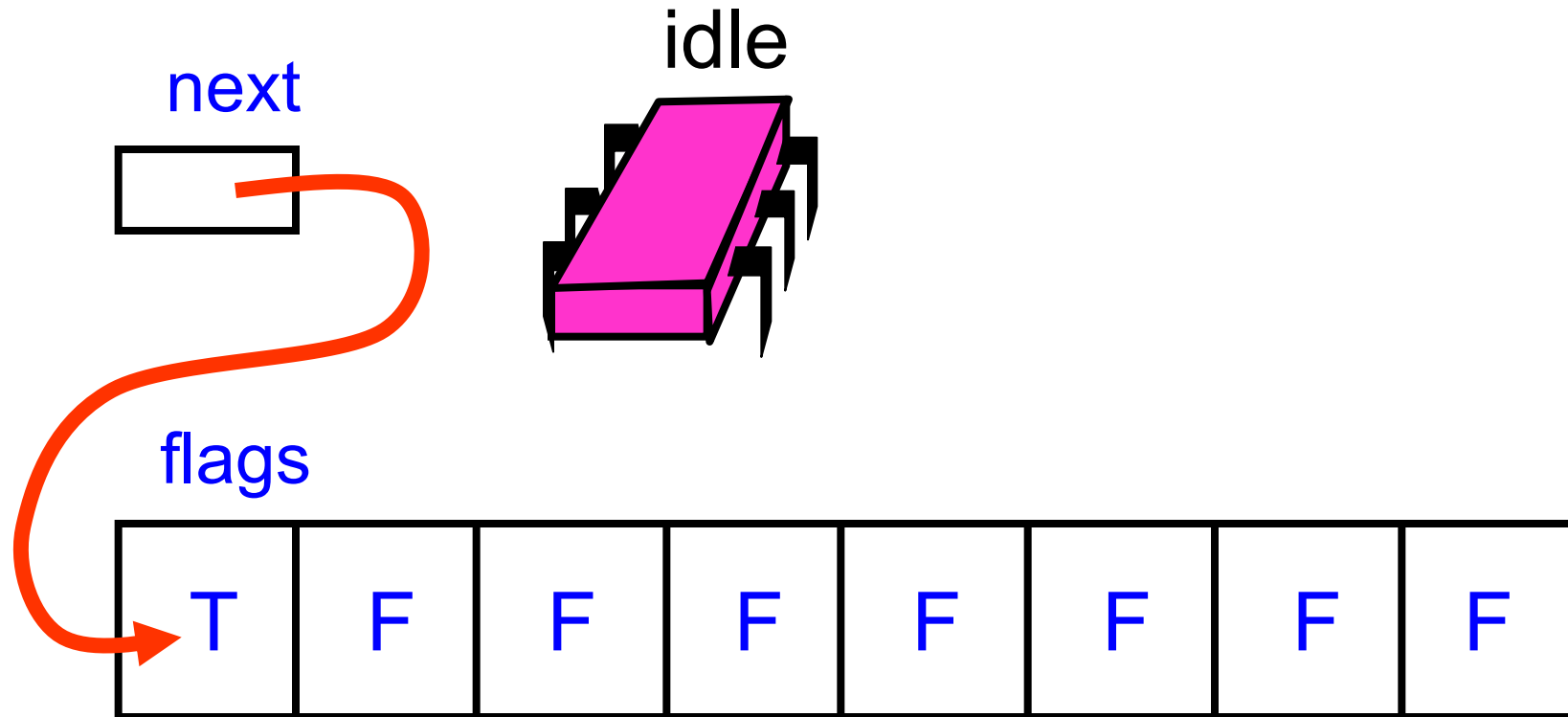


Idea

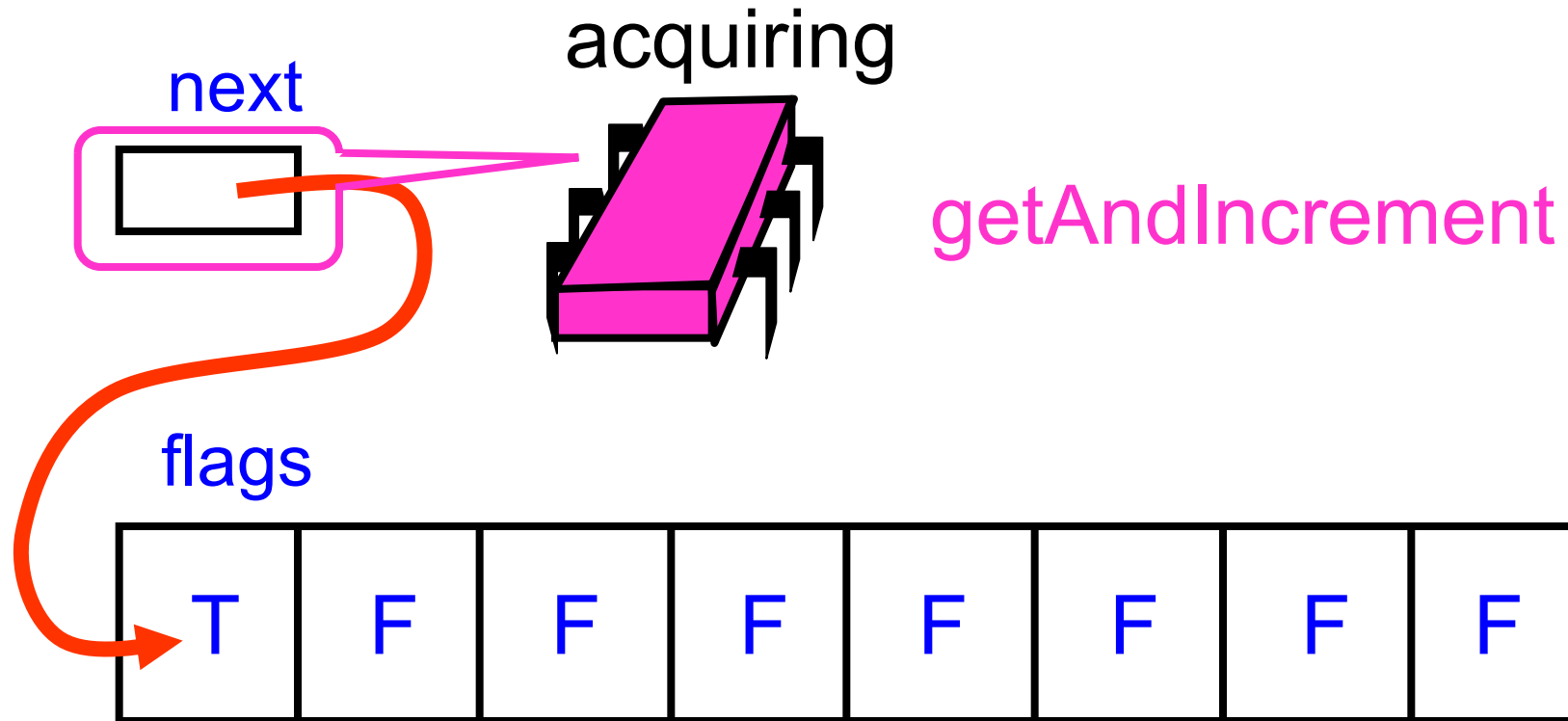
- Avoid useless invalidations
 - By keeping a queue of threads
- Each thread
 - Notifies next in line
 - Without bothering the others



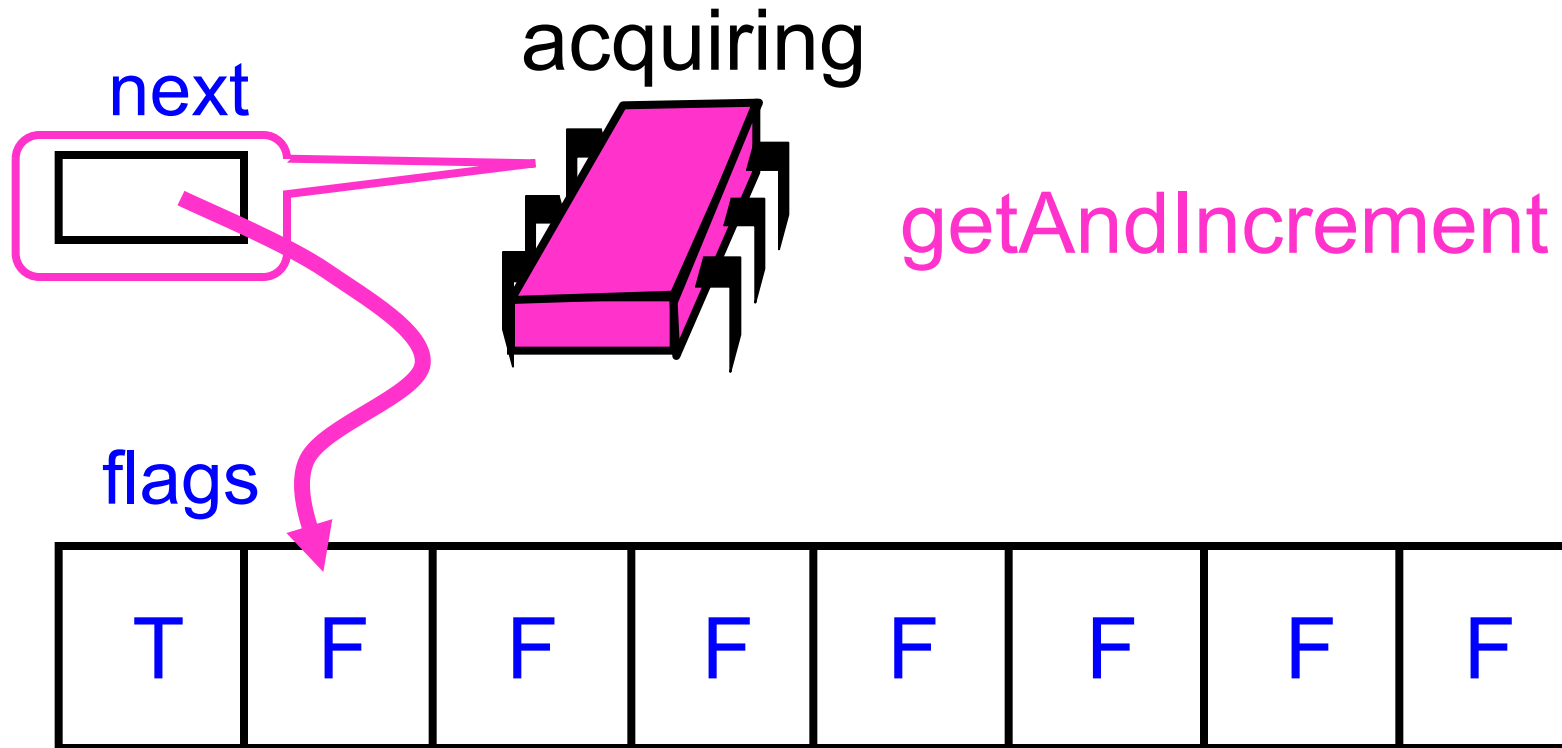
Anderson Queue Lock



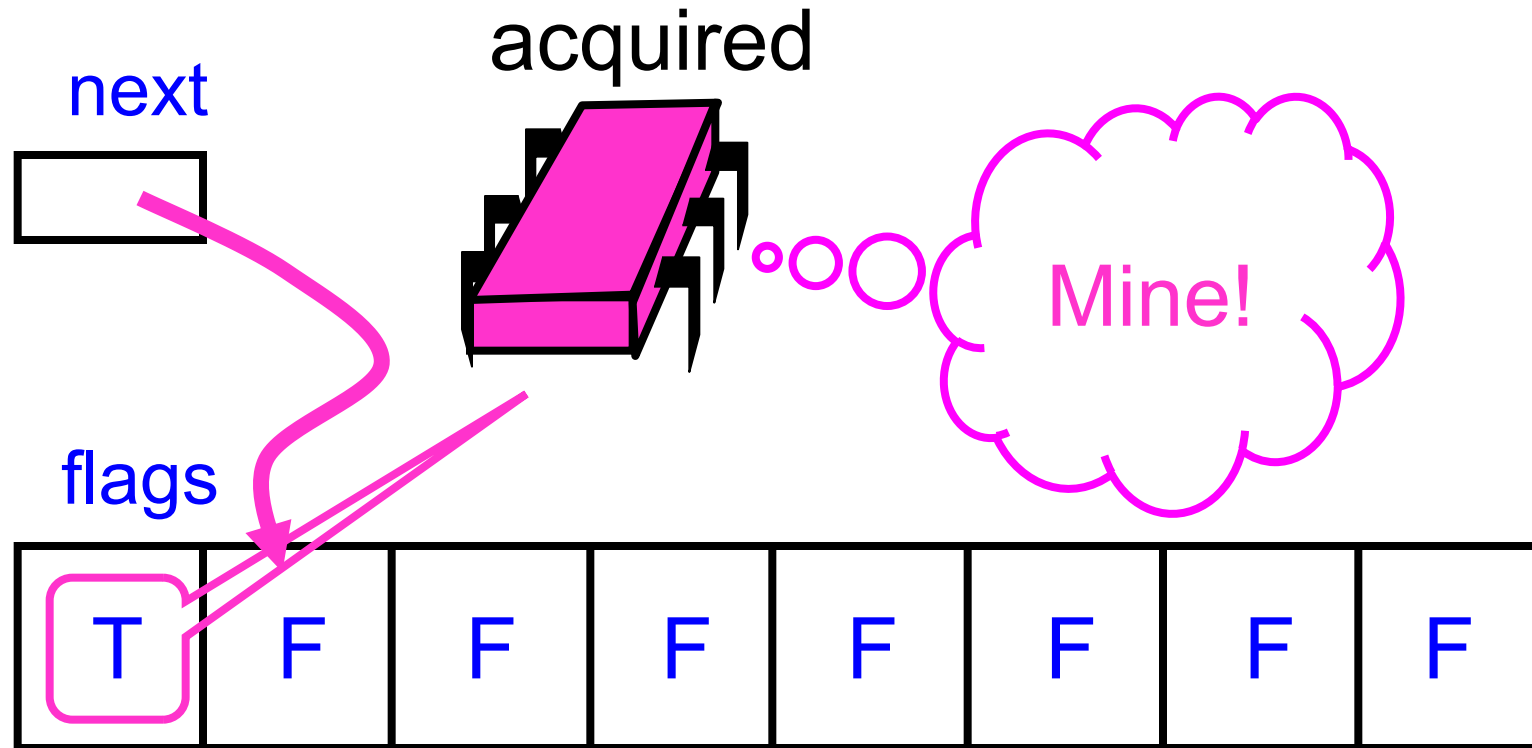
Anderson Queue Lock



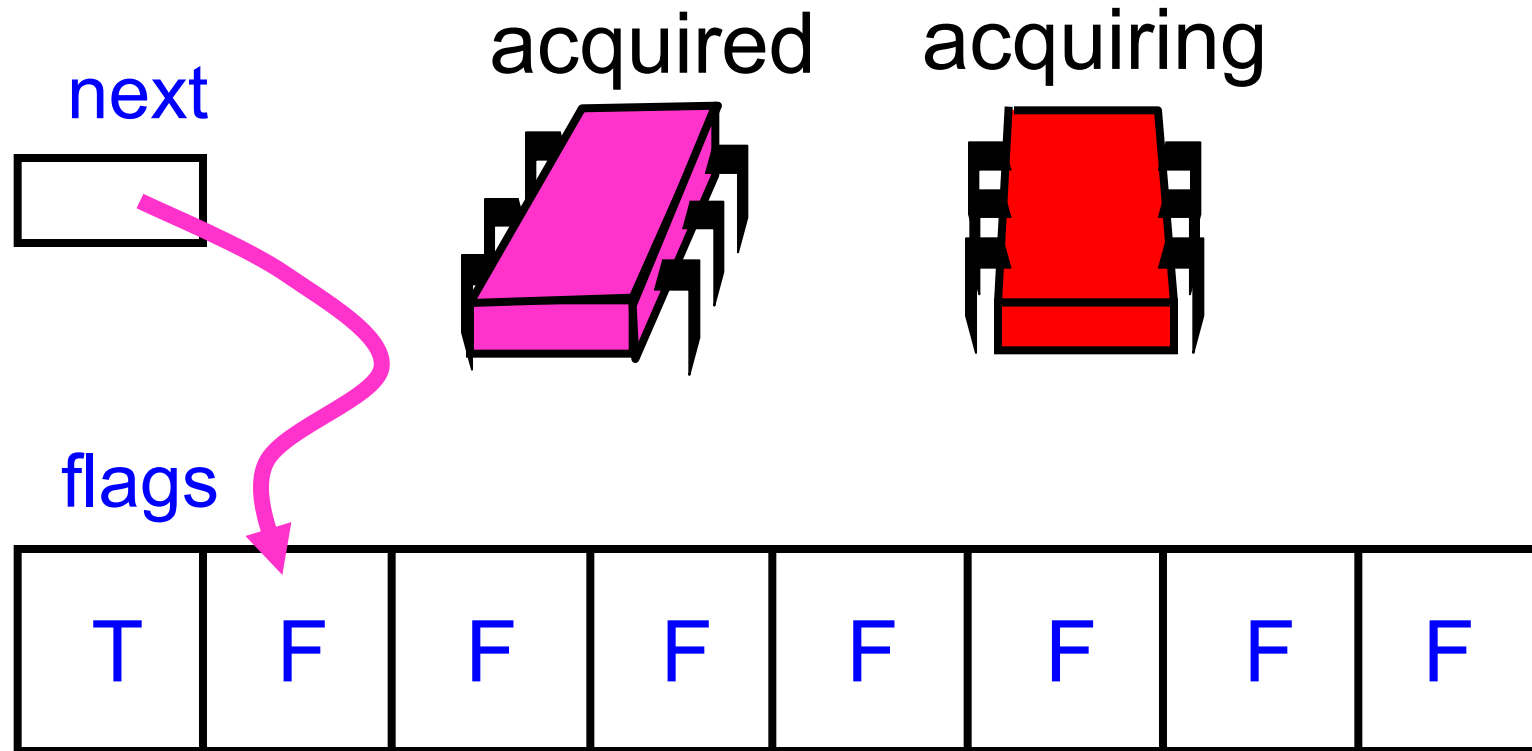
Anderson Queue Lock



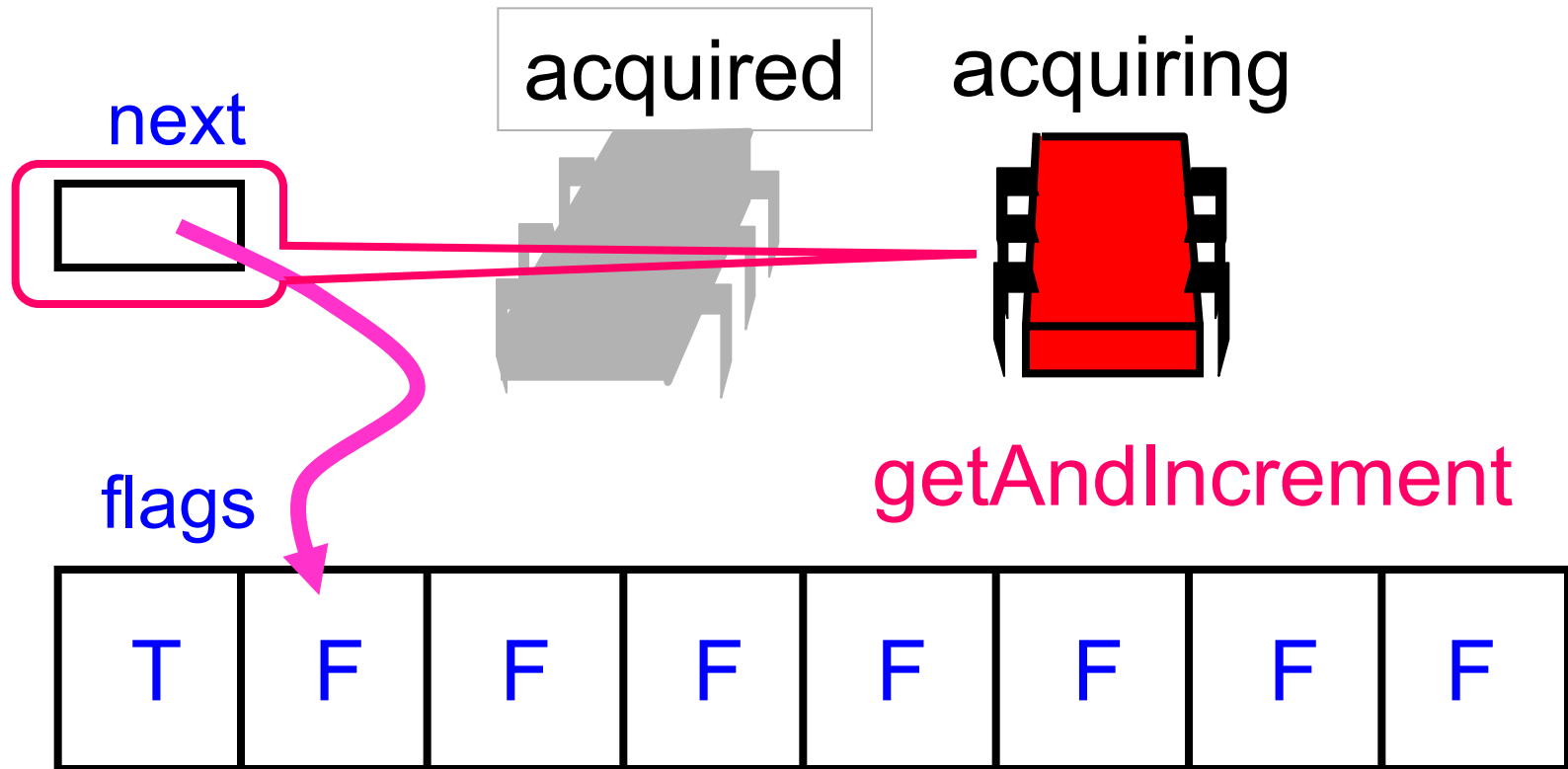
Anderson Queue Lock



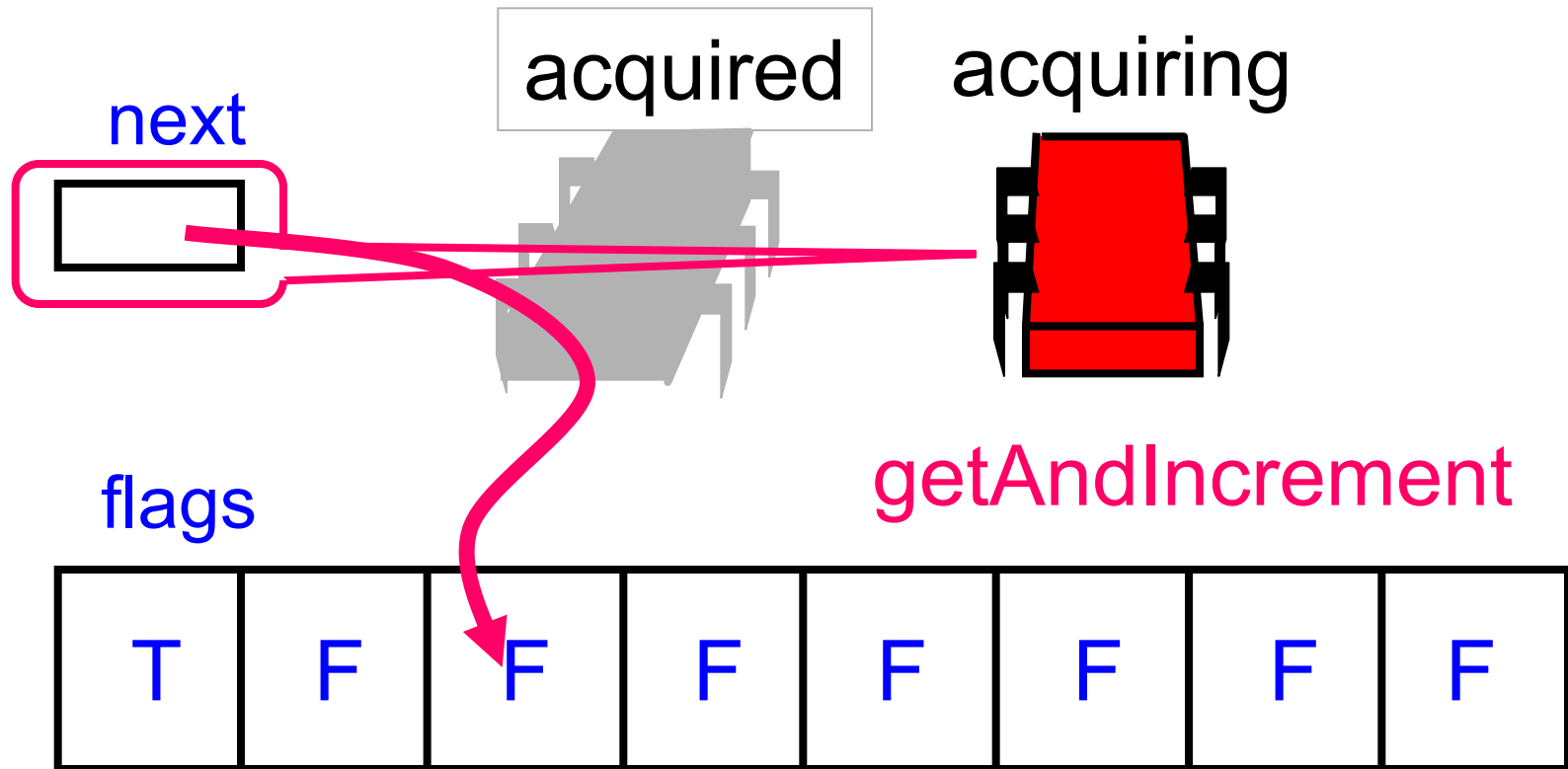
Anderson Queue Lock



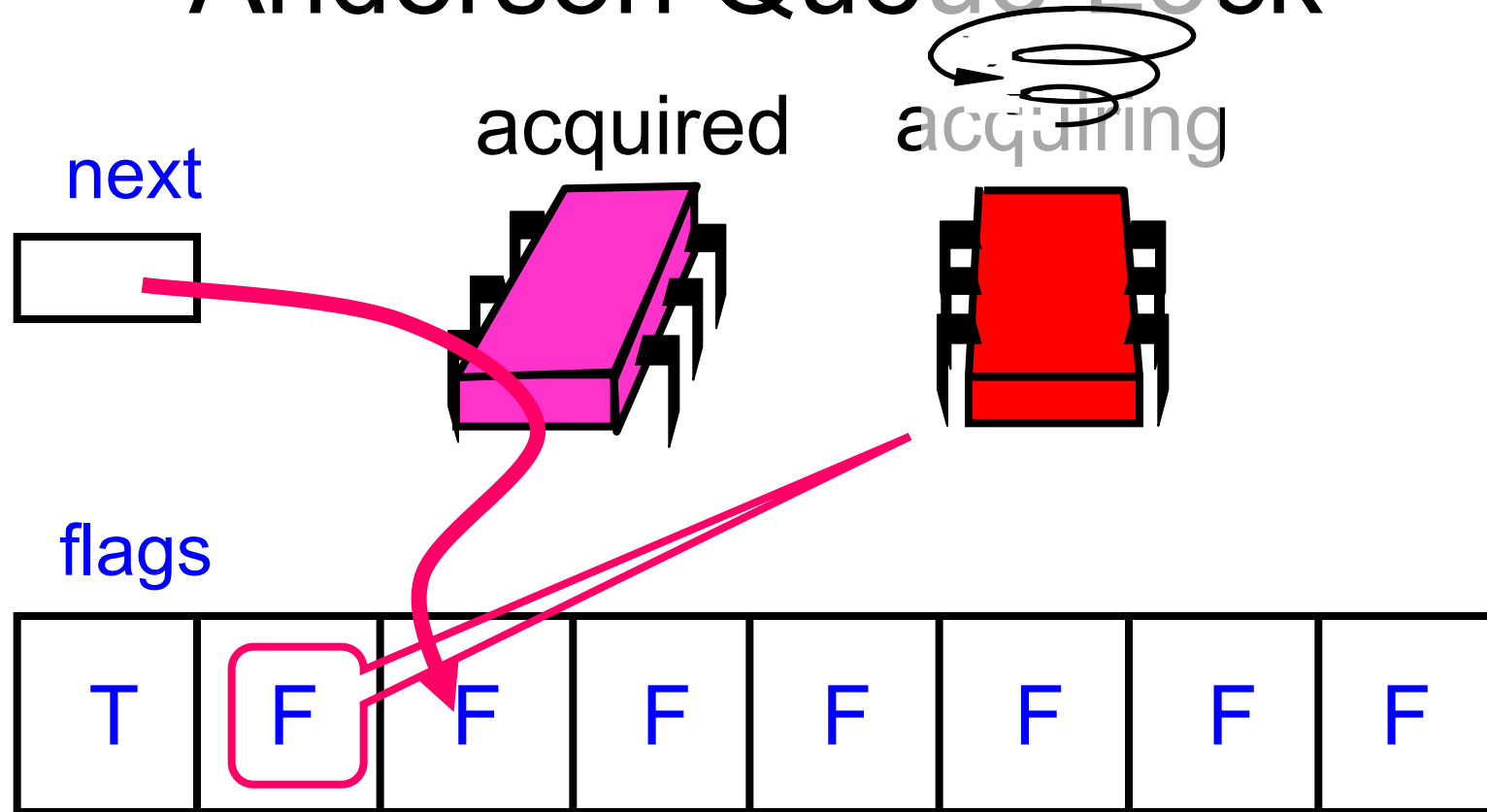
Anderson Queue Lock



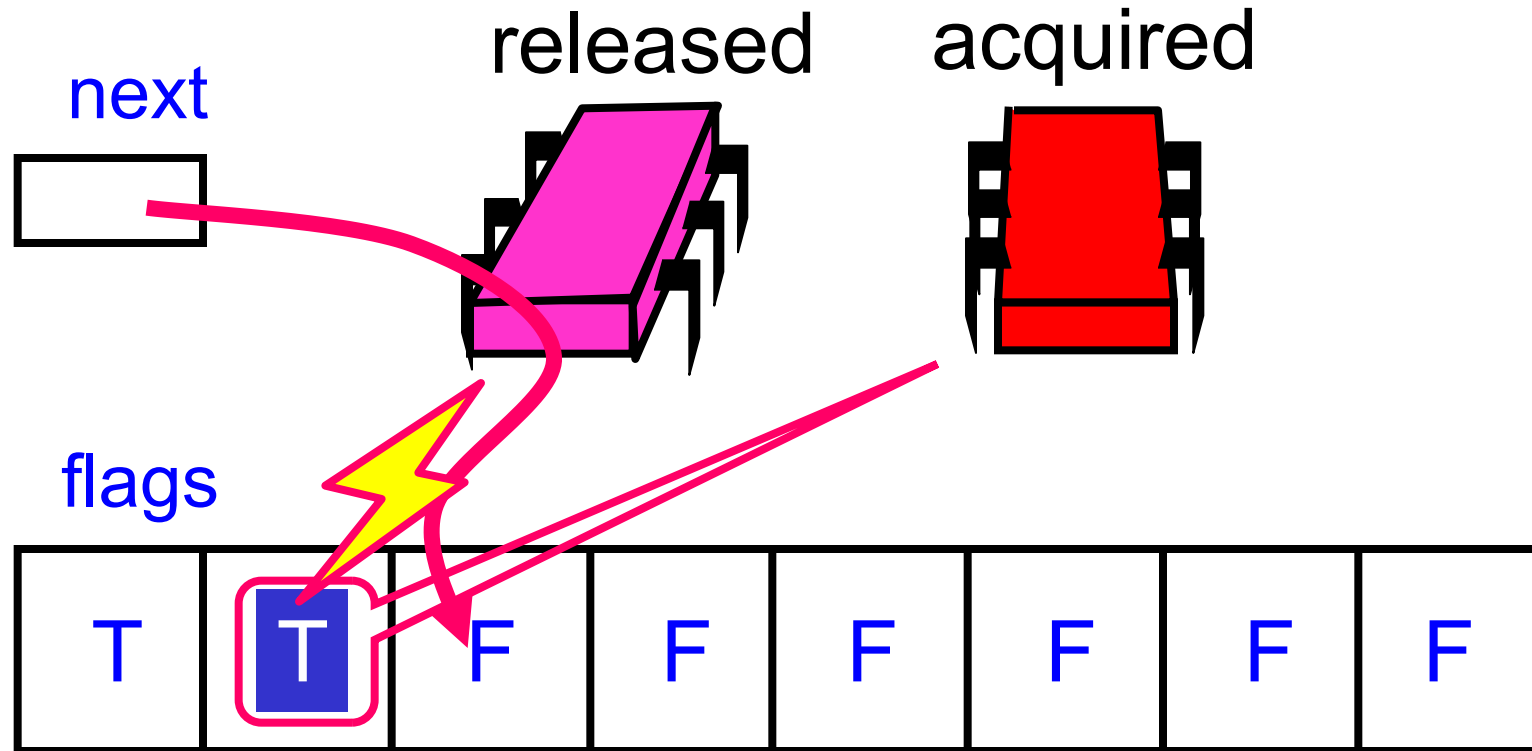
Anderson Queue Lock



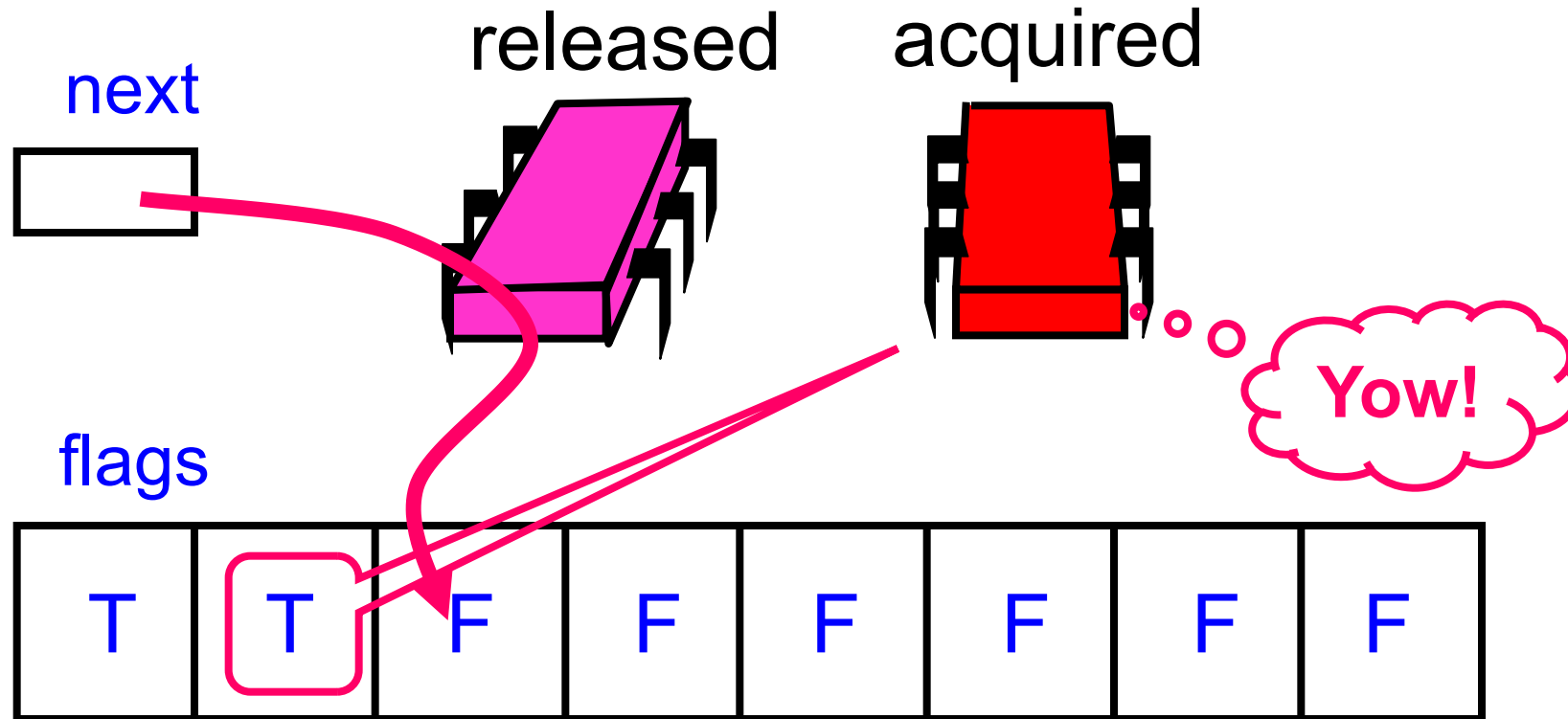
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock

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

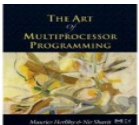
```
class ALock implements Lock {  
    boolean[] flags={true,false,...,false};  
    AtomicInteger next  
    = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

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

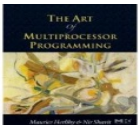
```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n]  
}
```

Take next slot

Anderson Queue Lock

```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```

Spin until told to go

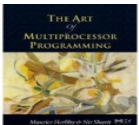


Anderson Queue Lock

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

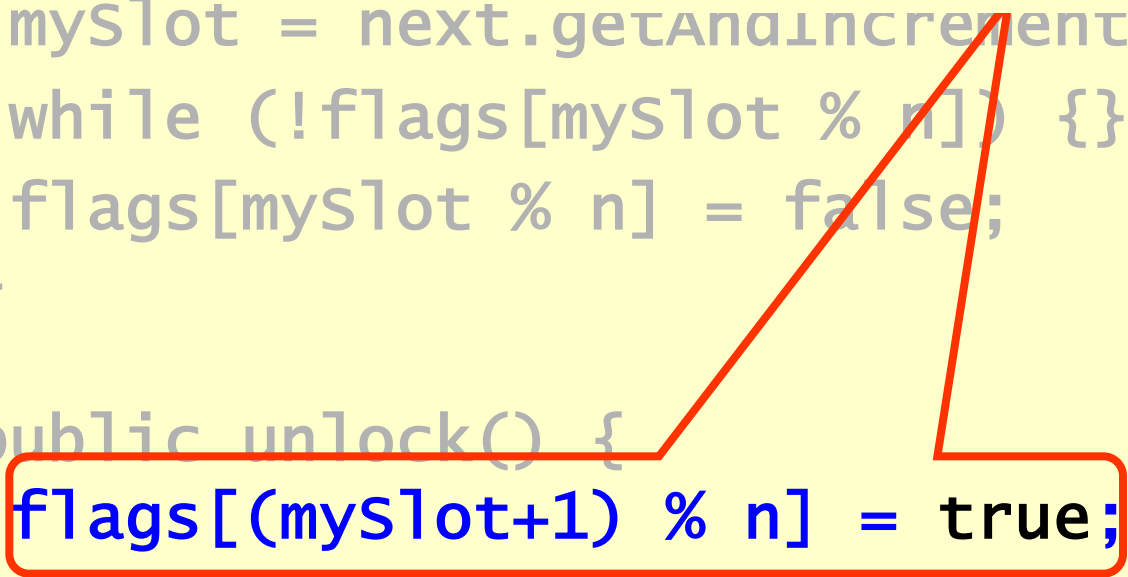
```
public unlock() {  
    flags[(myslot+1) % n] = true;  
}
```

Prepare slot for re-use

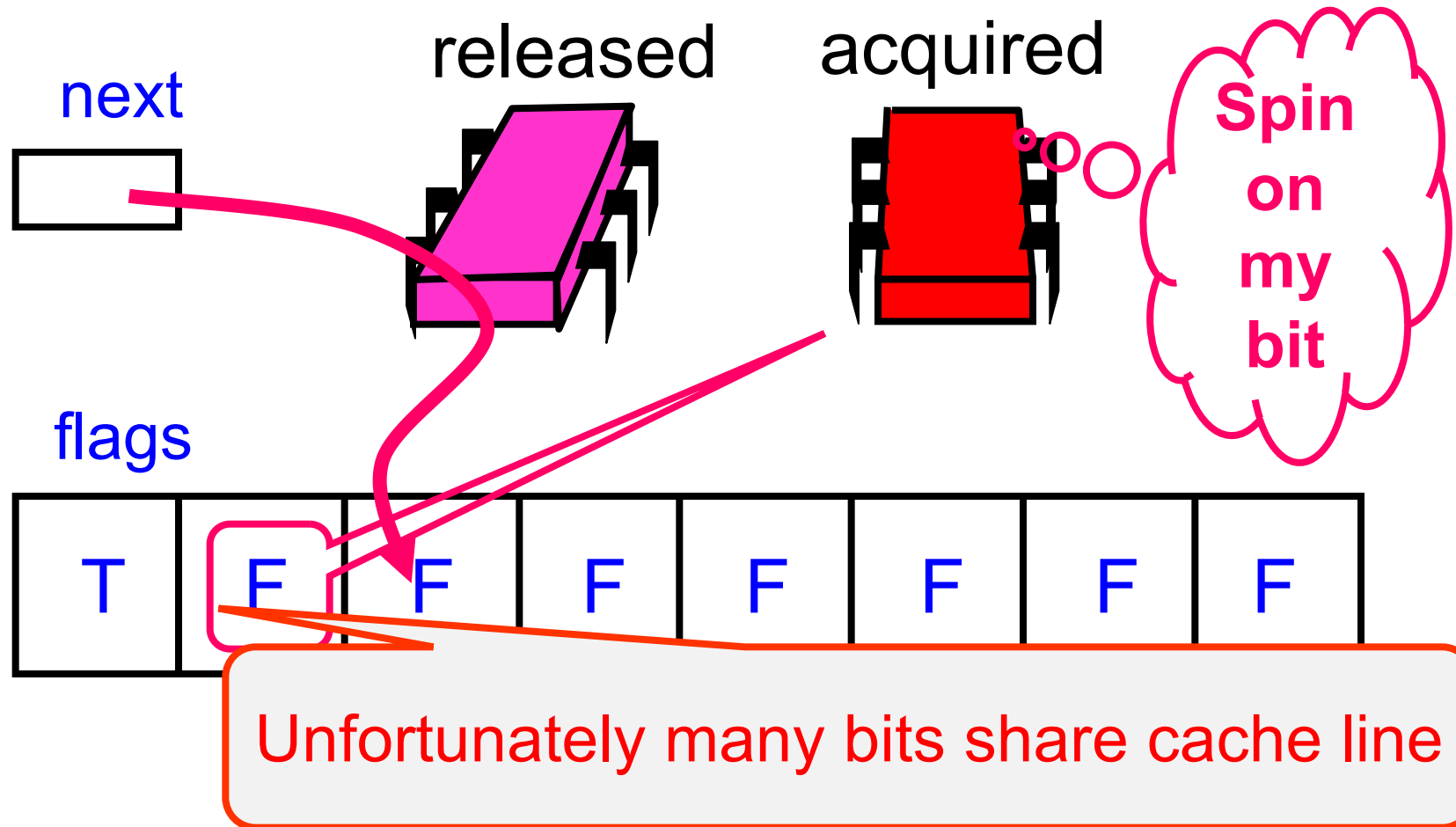


Anderson Queue Lock

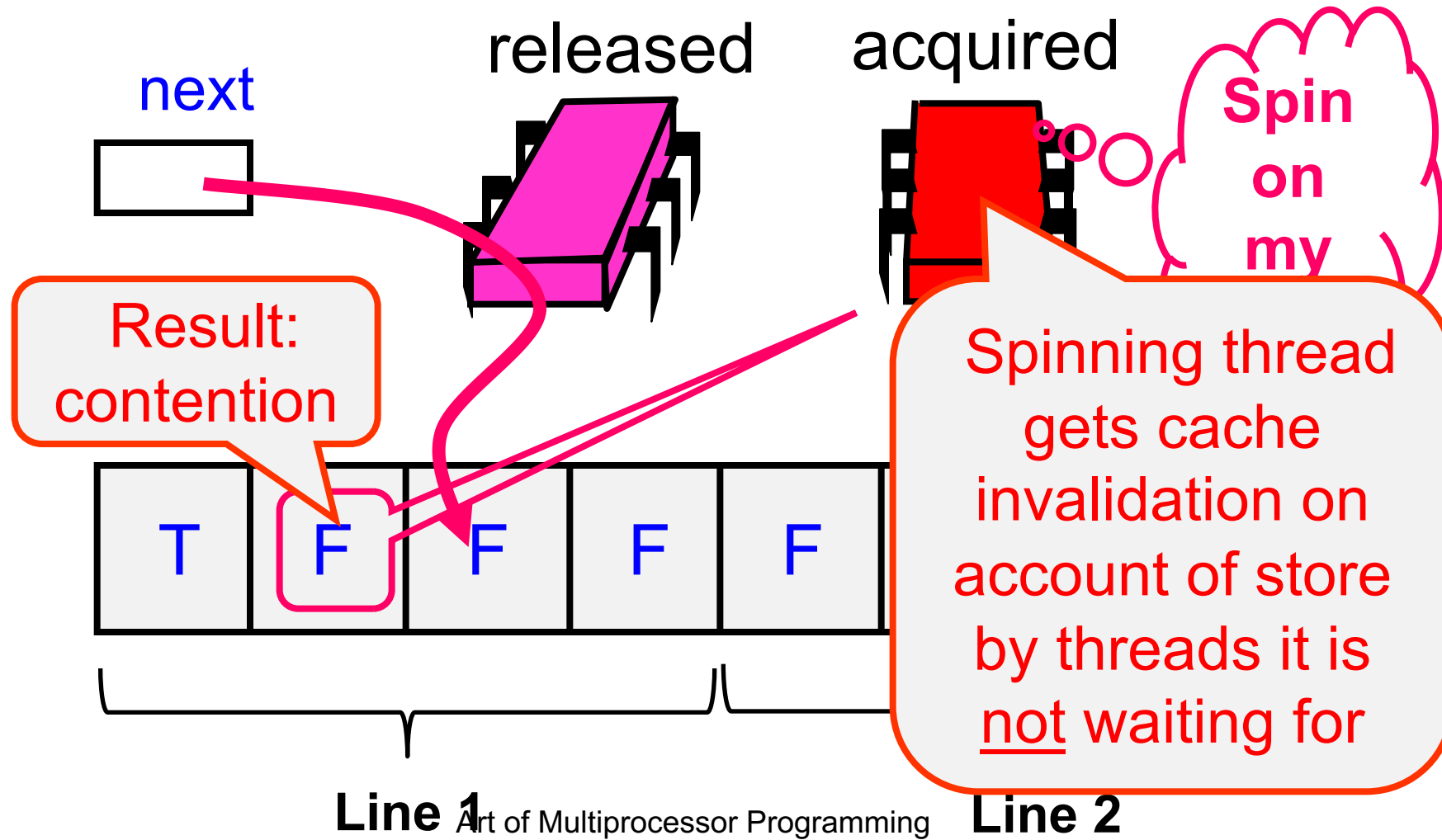
```
public lock() { Tell next thread to go  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```



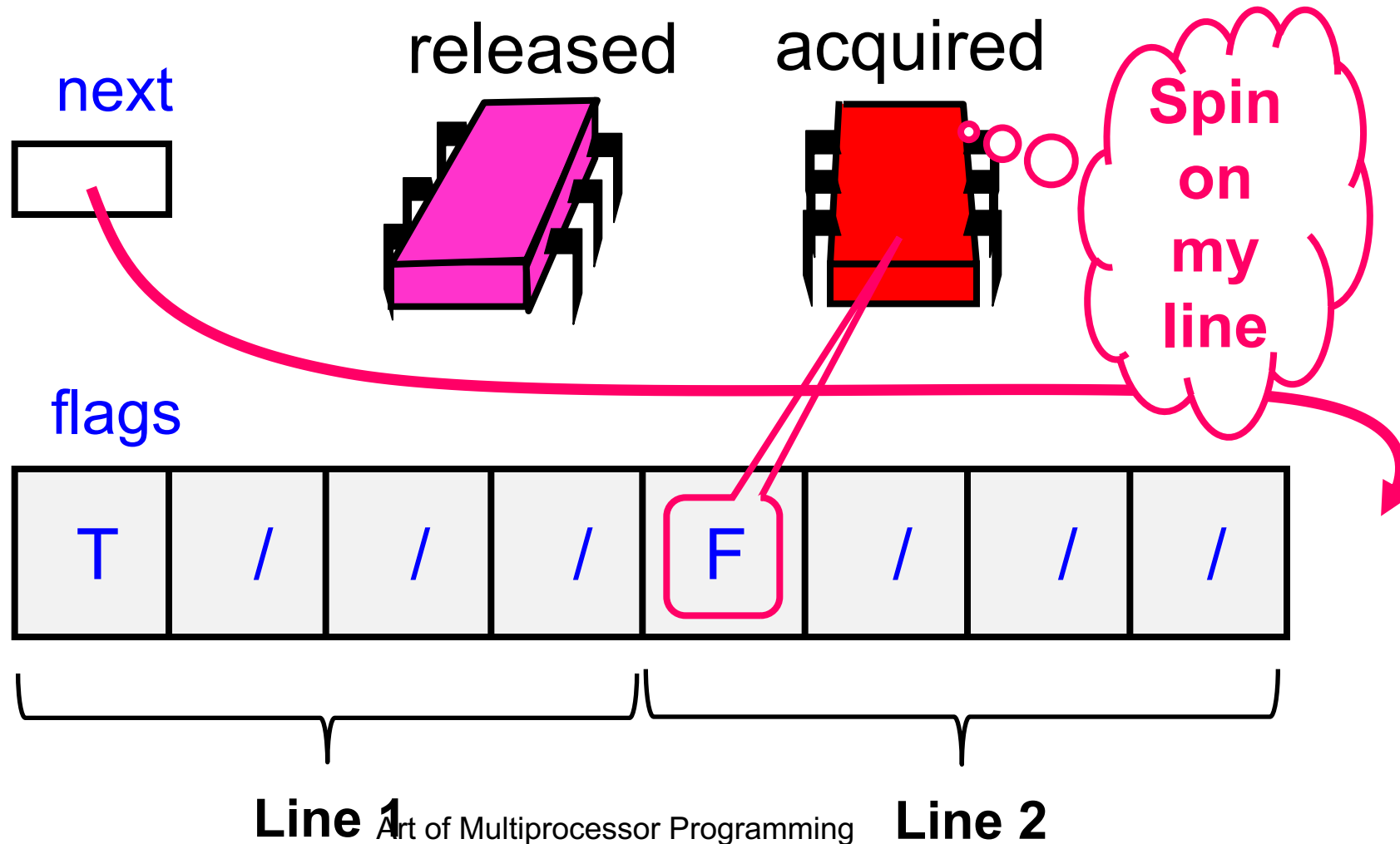
Local Spinning



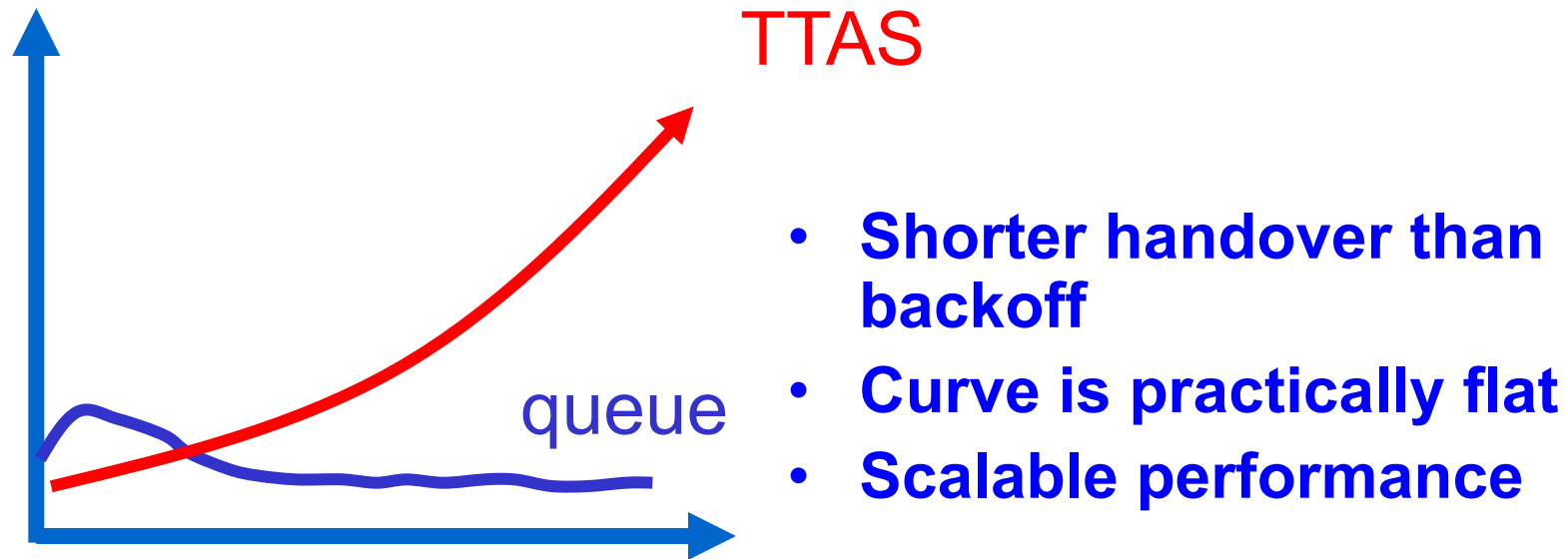
False Sharing



The Solution: Padding



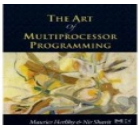
Performance



Anderson Queue Lock

Good

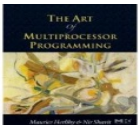
- First truly scalable lock
- Simple, easy to implement
- Back to FIFO order (like Bakery)



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?

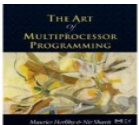
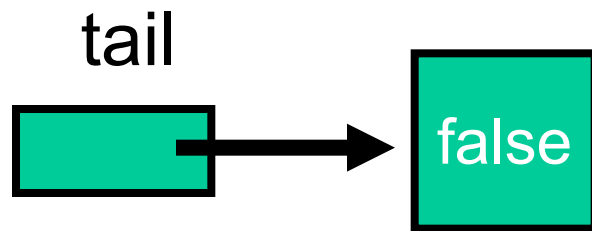
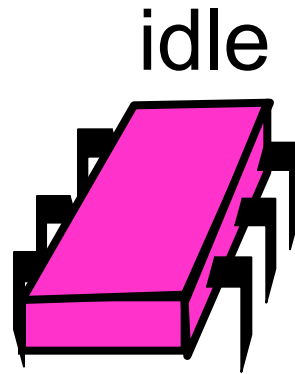


CLH (Craig-Landin-Hagersten) Lock

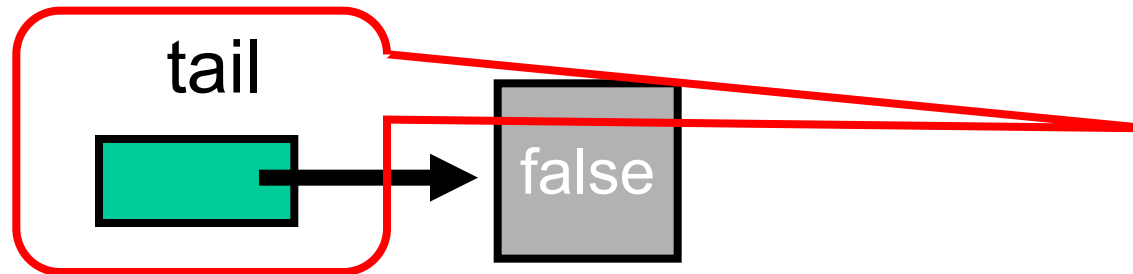
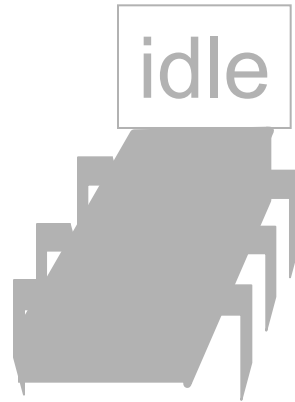
- FIFO order
- Small, constant-size overhead per thread



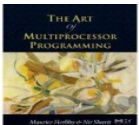
Initially



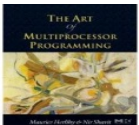
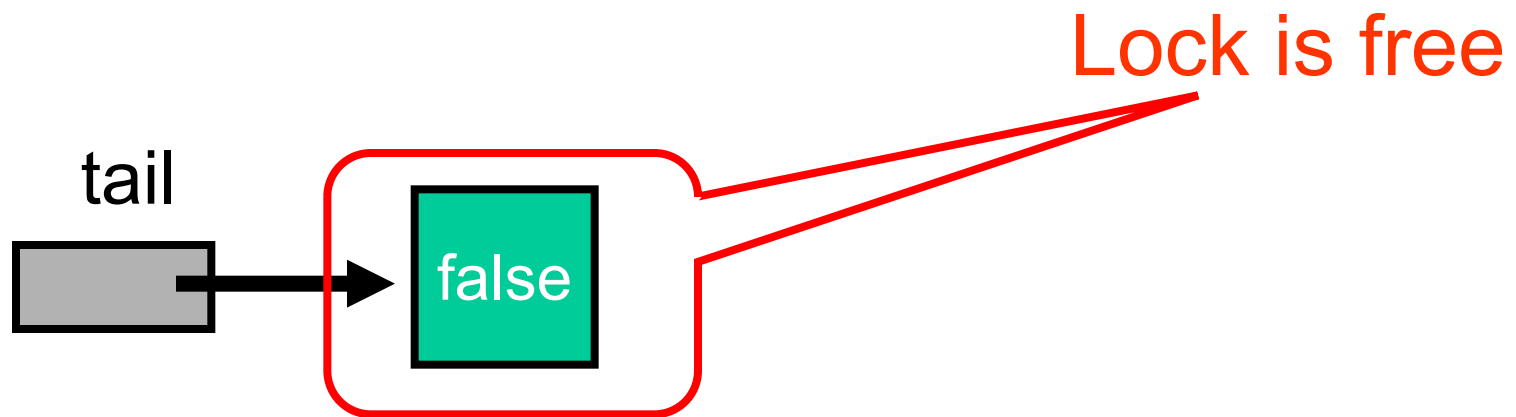
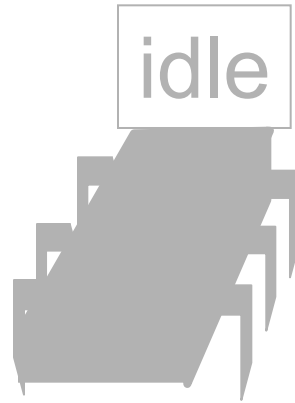
Initially



Queue tail

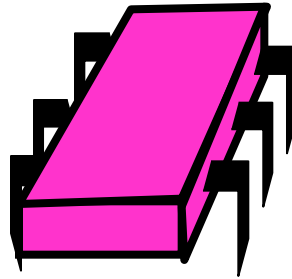


Initially

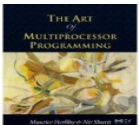
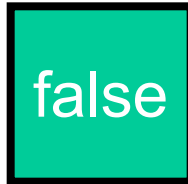


Initially

idle

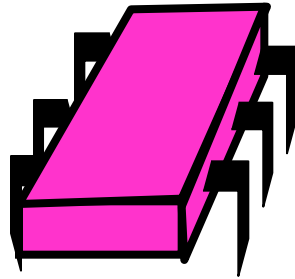


tail

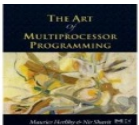
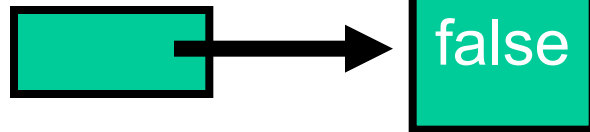


Purple Wants the Lock

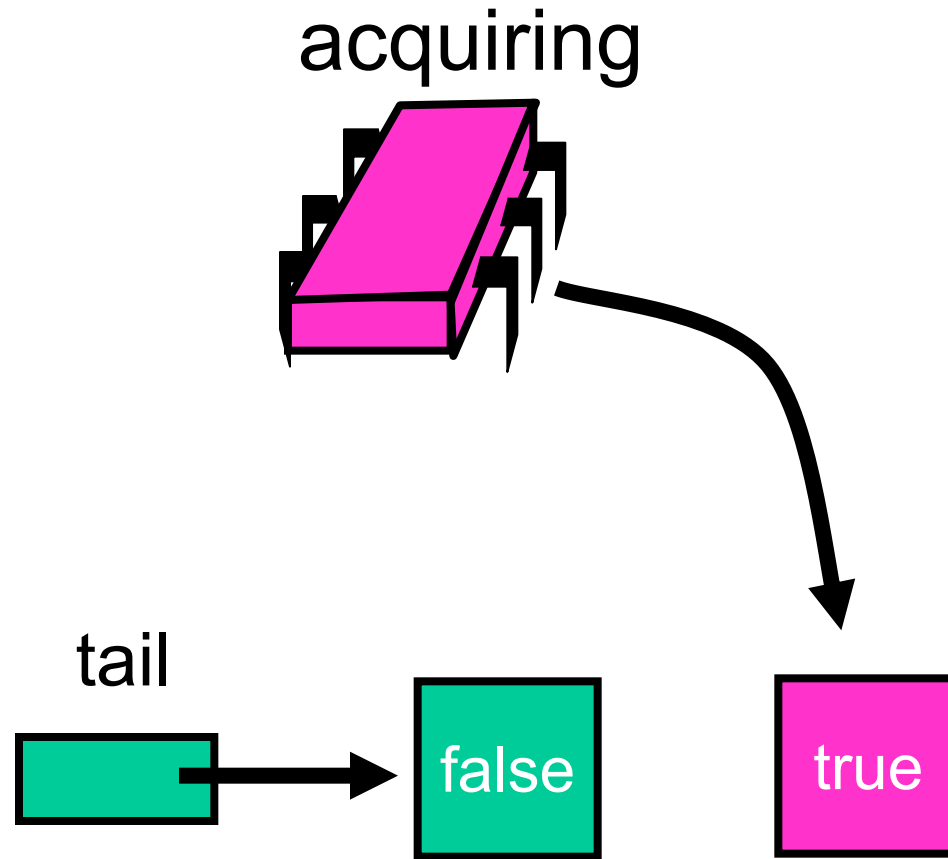
acquiring



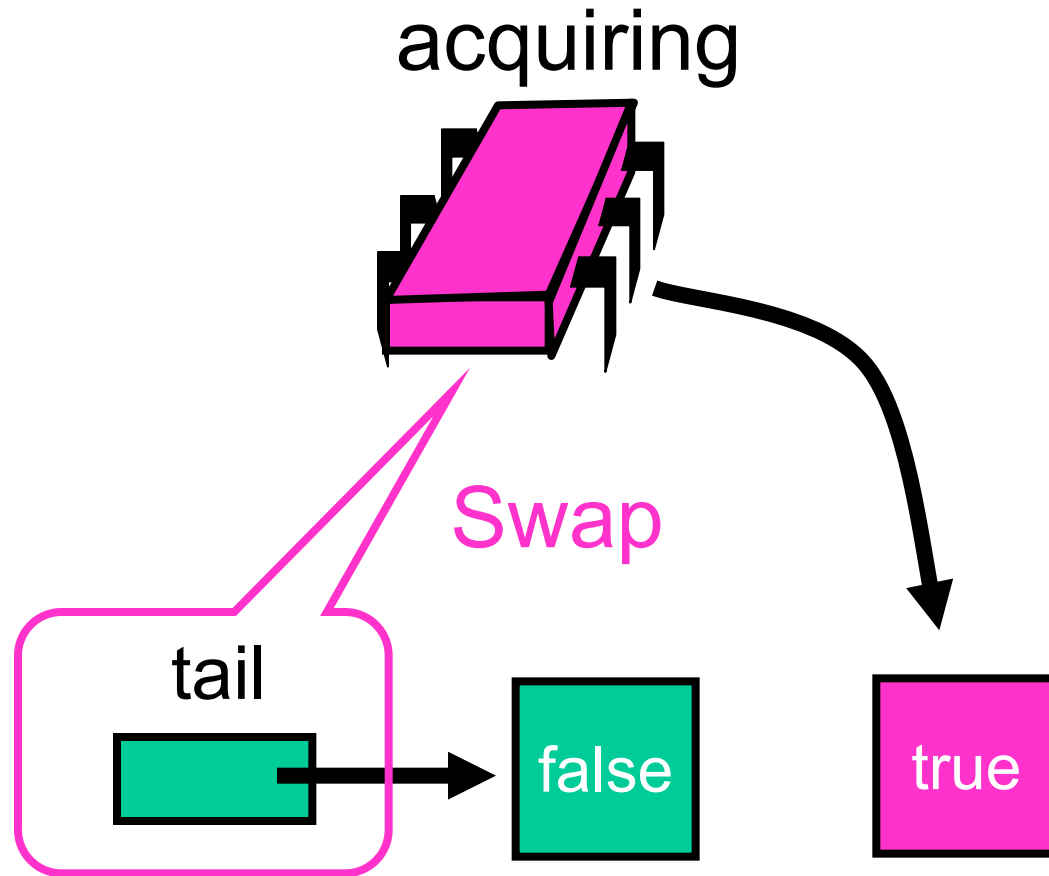
tail



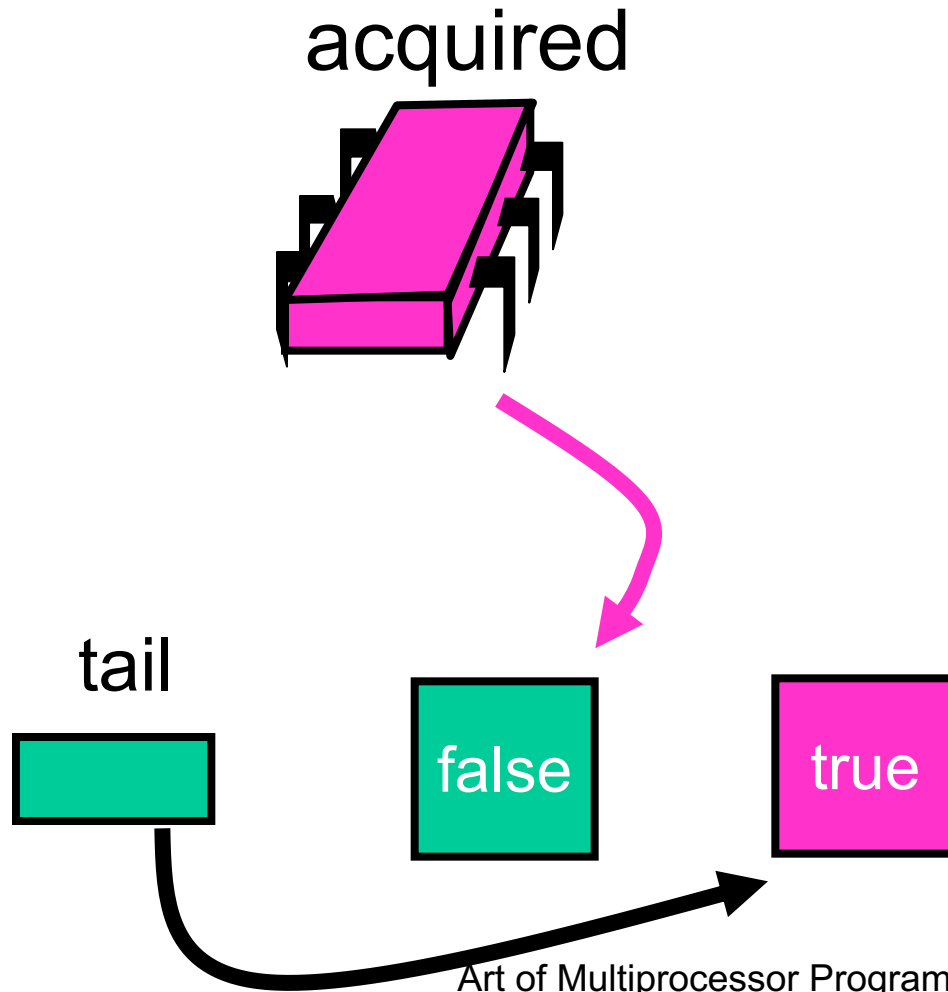
Purple Wants the Lock



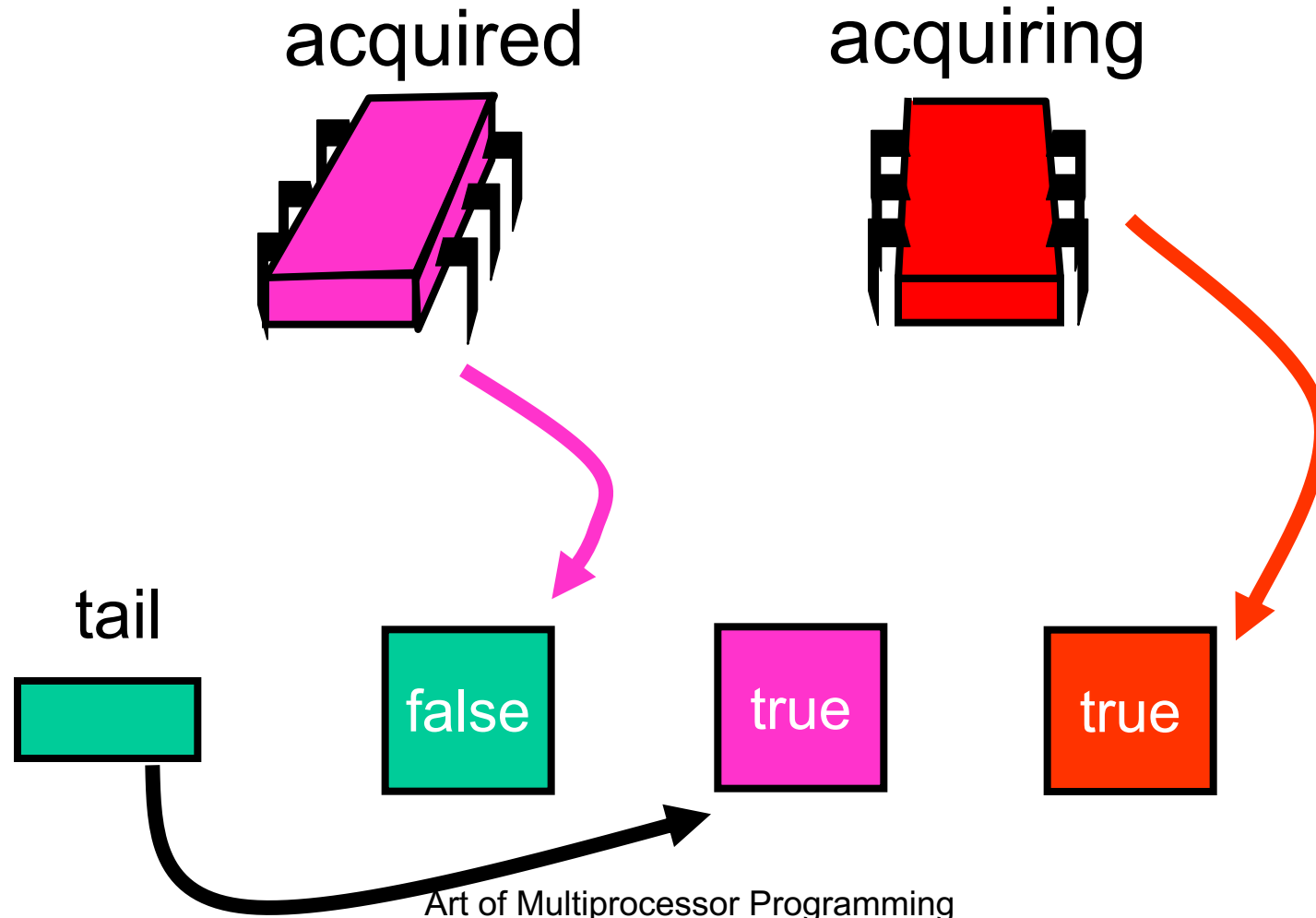
Purple Wants the Lock



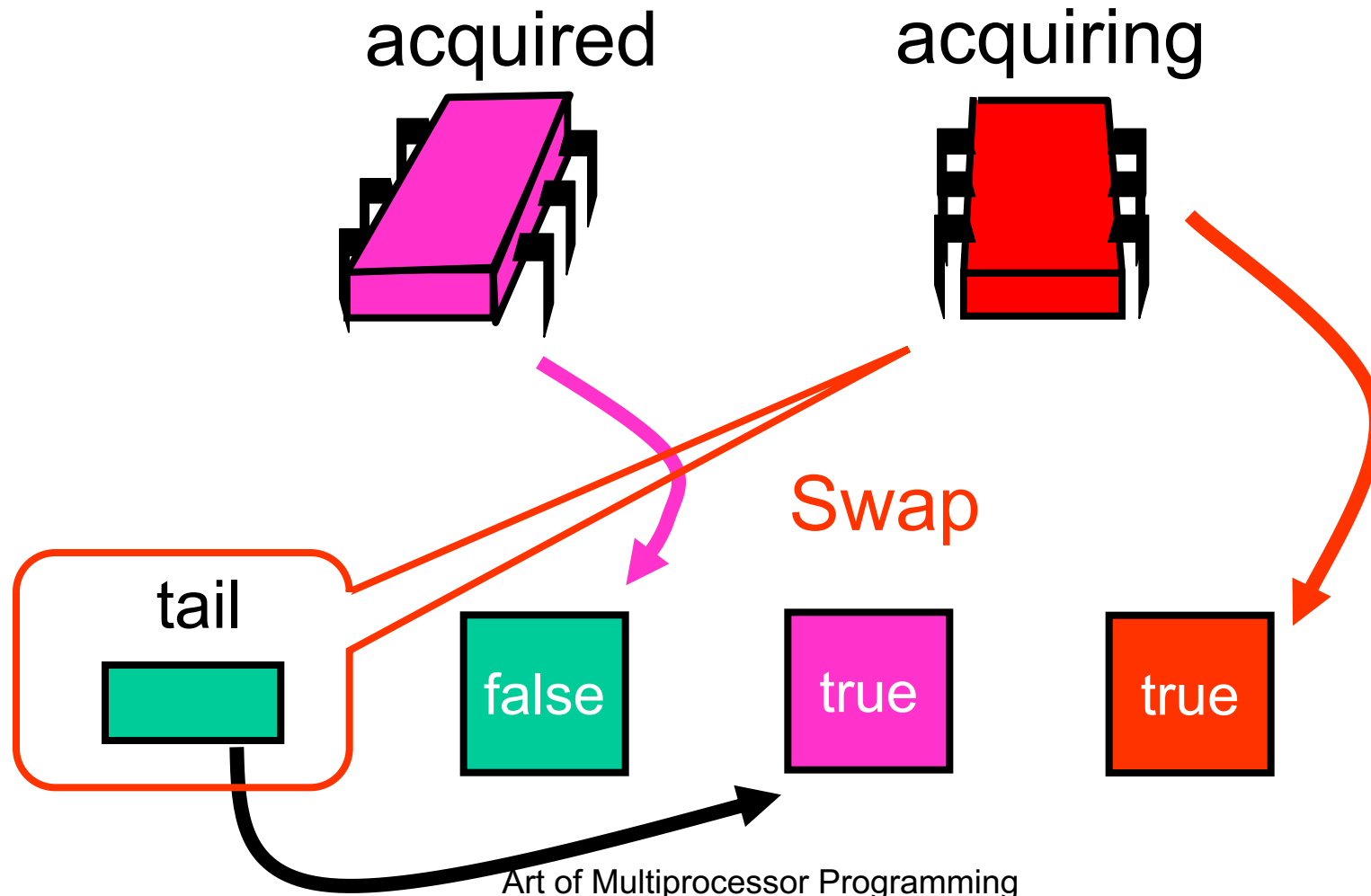
Purple Has the Lock



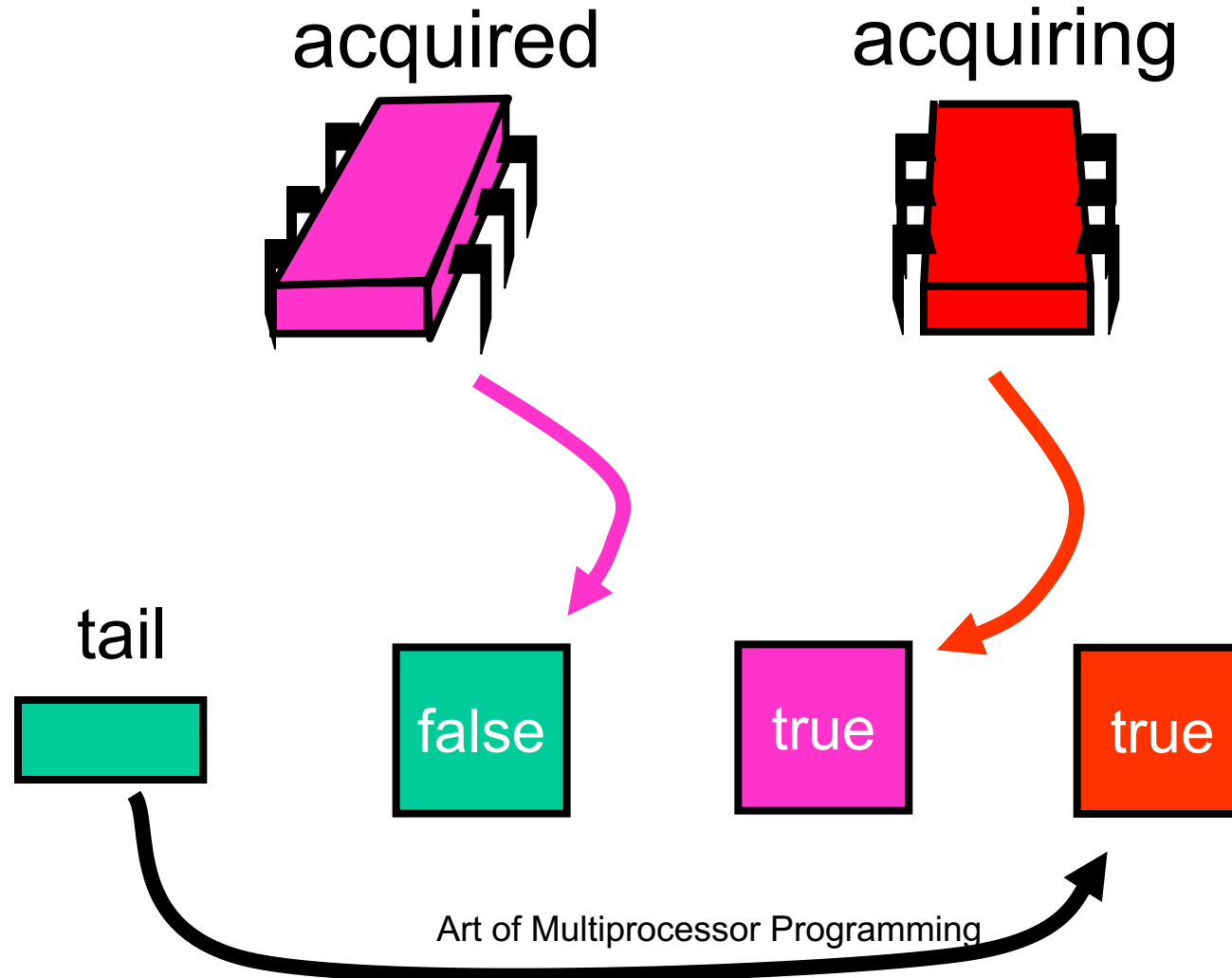
Red Wants the Lock



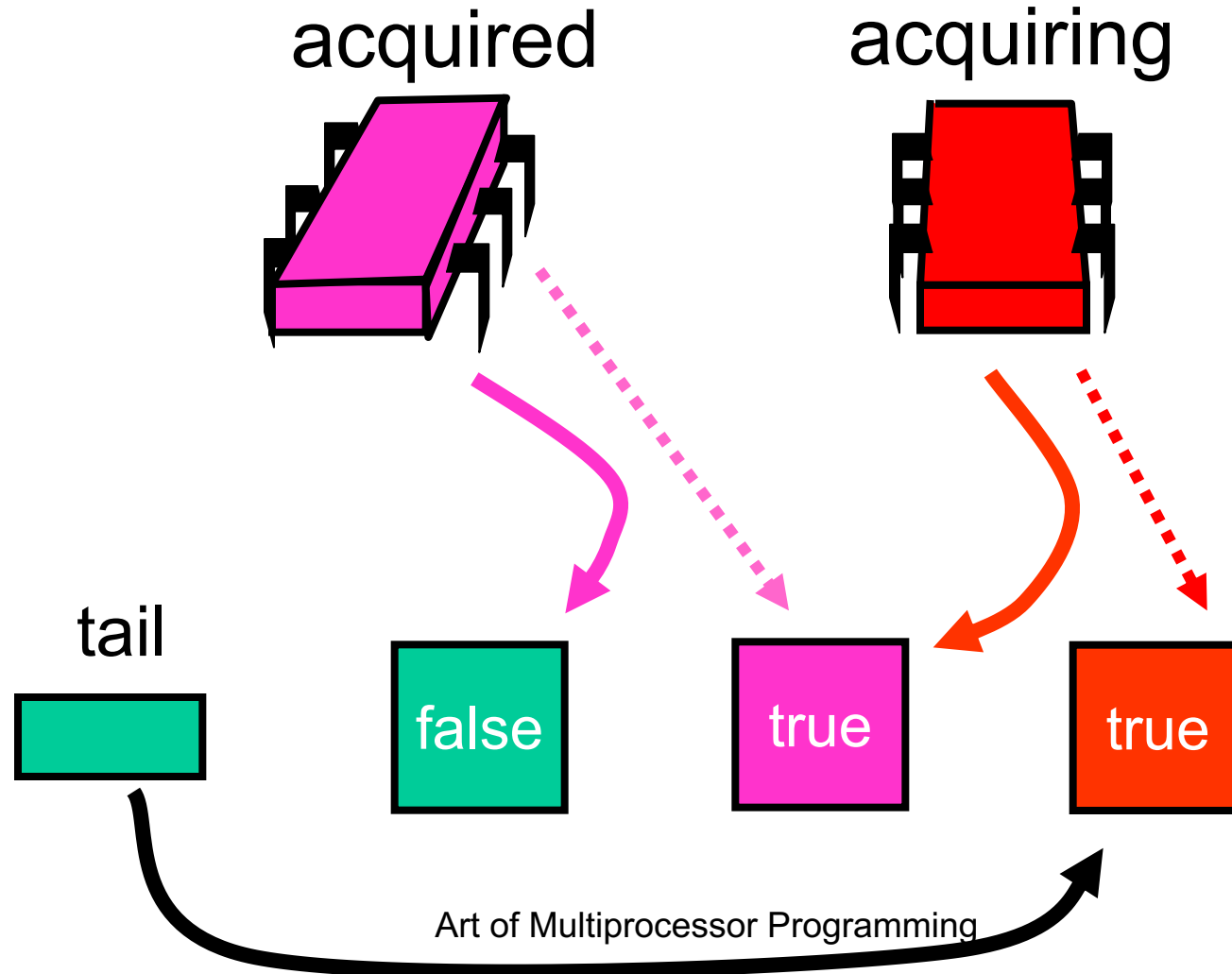
Red Wants the Lock



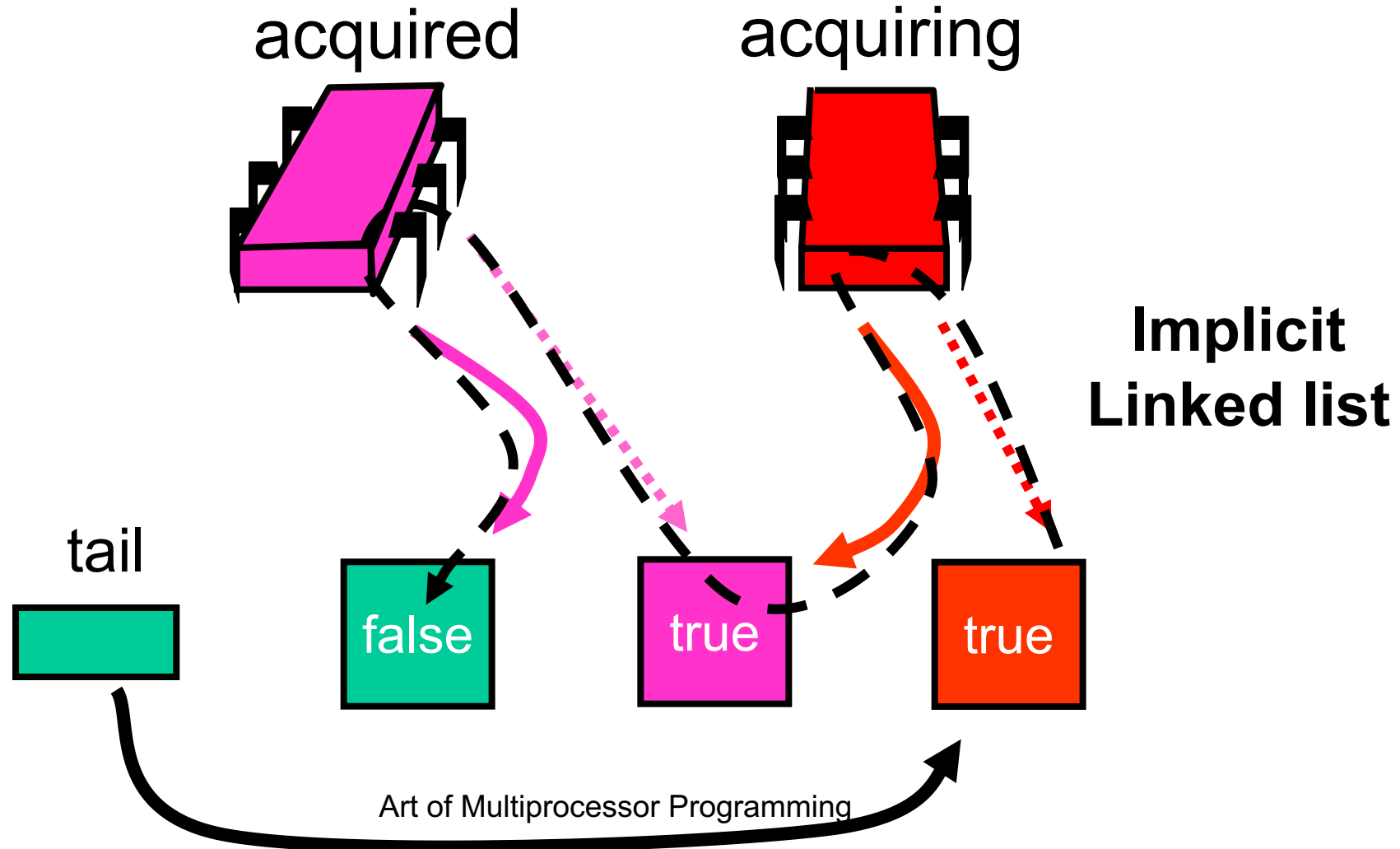
Red Wants the Lock



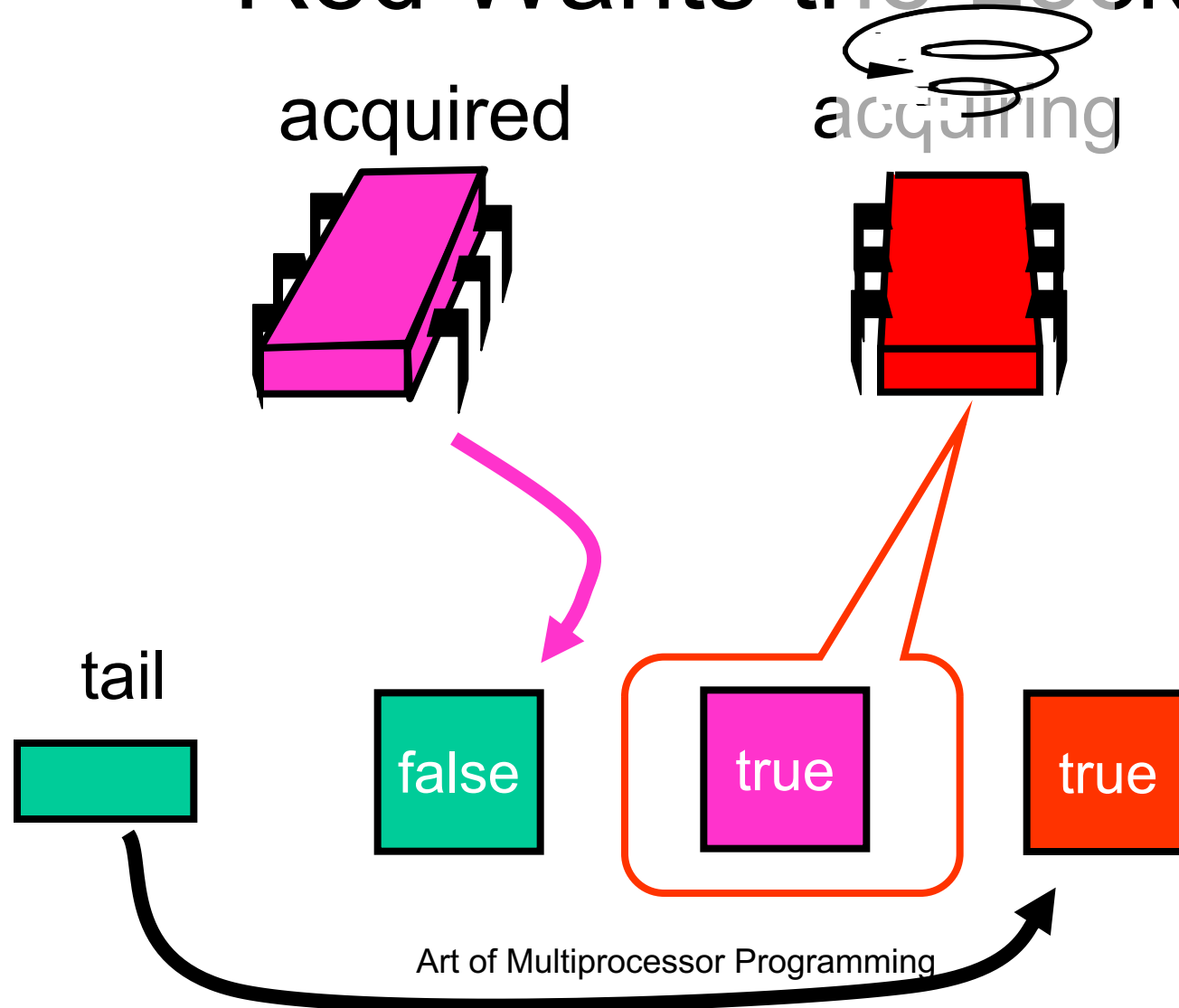
Red Wants the Lock



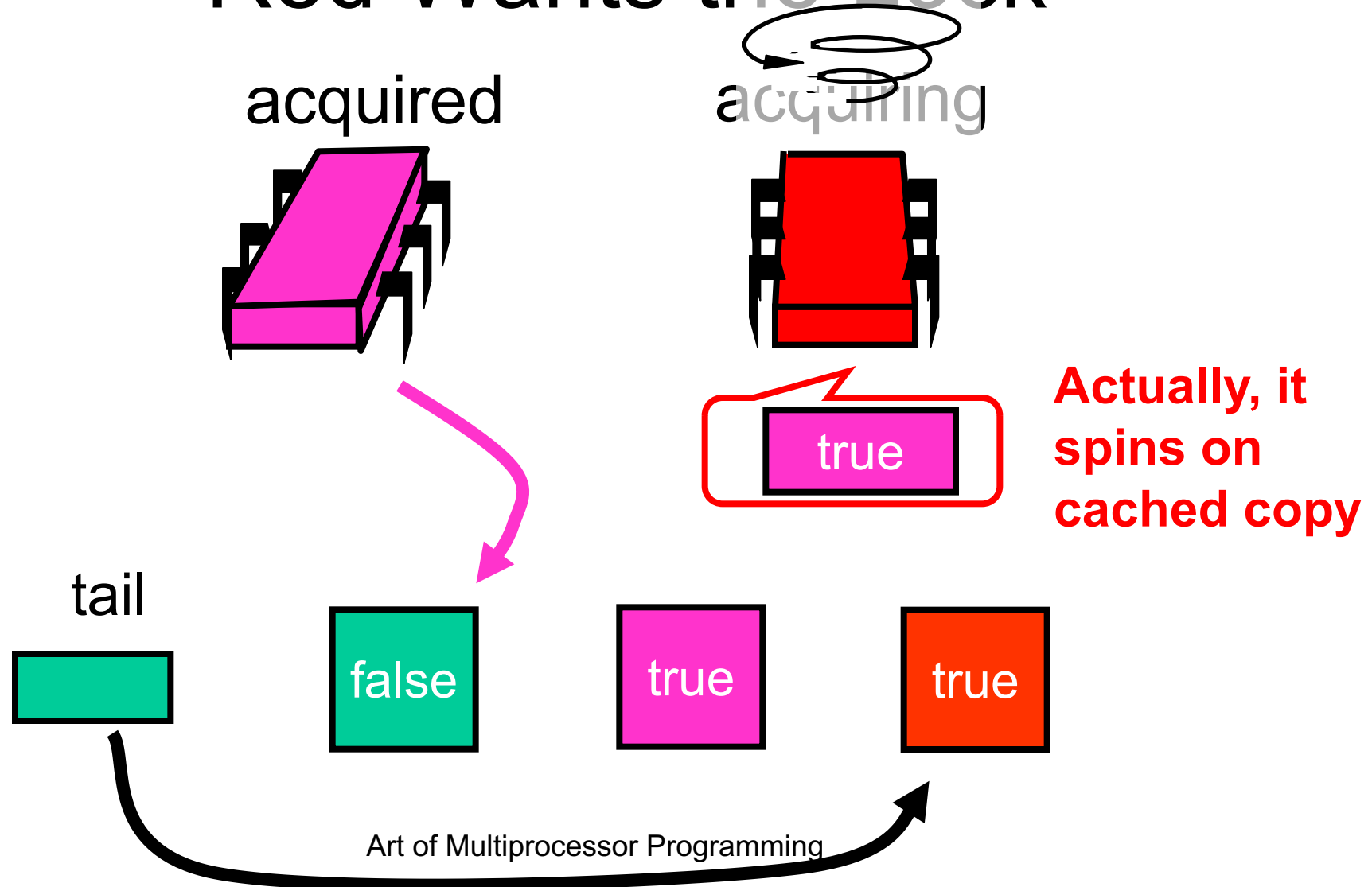
Red Wants the Lock



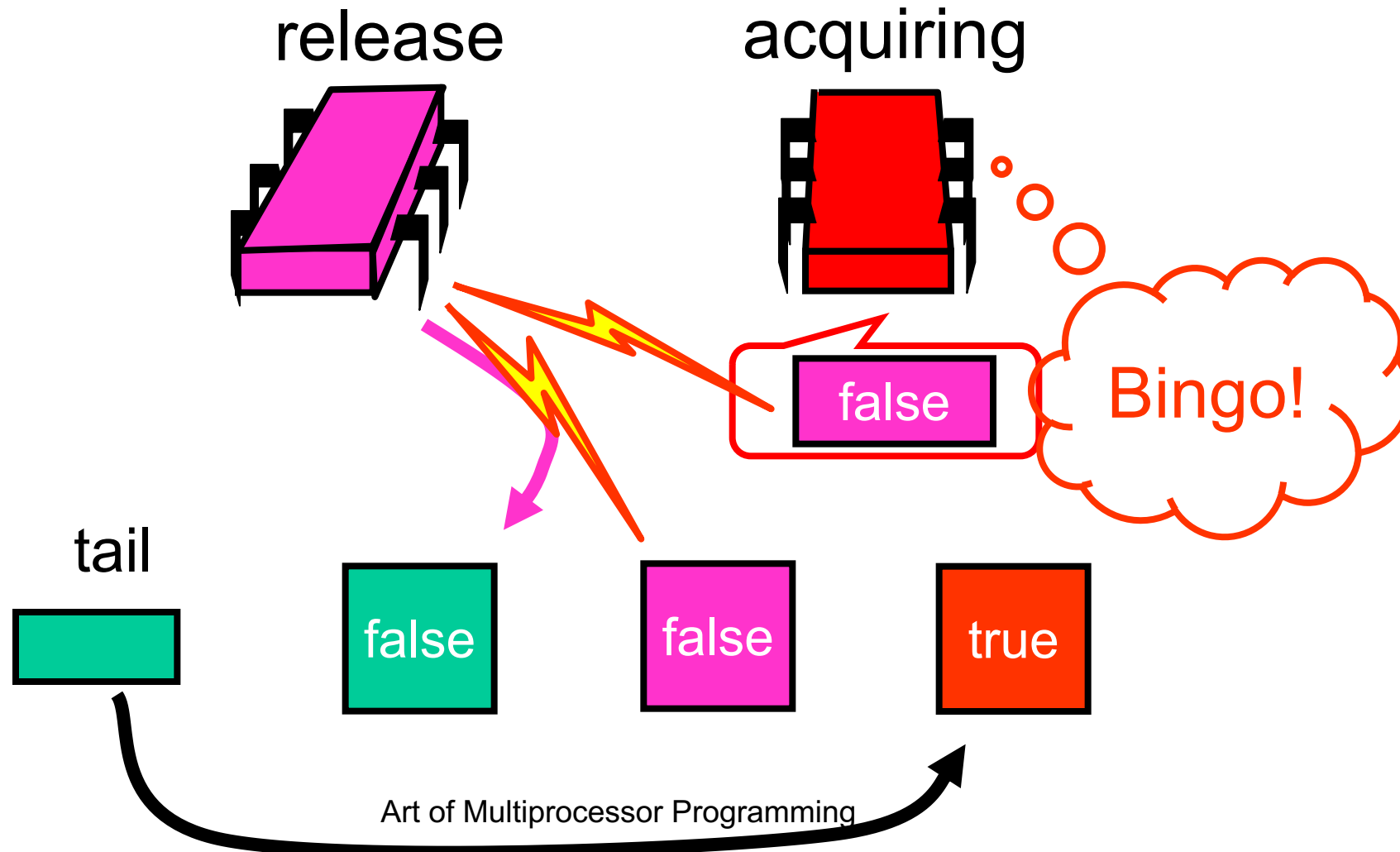
Red Wants the Lock



Red Wants the Lock

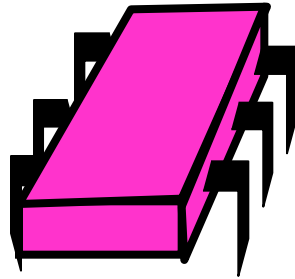


Purple Releases

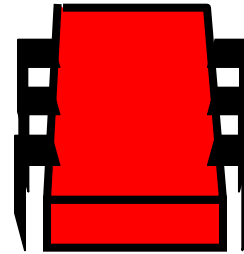


Purple Releases

released



acquired



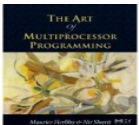
tail



true



Art of Multiprocessor Programming



Space Usage

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



CLH Queue Lock

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



CLH Queue Lock

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

Not released yet

CLH Queue Lock

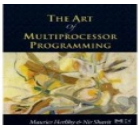
```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```



CLH Queue Lock

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

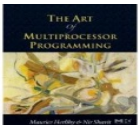
Queue tail



CLH Queue Lock

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

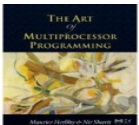
Thread-local Qnode



CLH Queue Lock

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

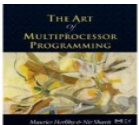
Swap in my node



CLH Queue Lock

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

Spin until predecessor
releases lock



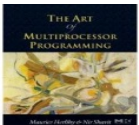
CLH Queue Lock

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

CLH Queue Lock

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

Notify successor



CLH Queue Lock

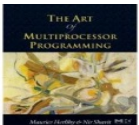
```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

Recycle
predecessor's node

CLH Queue Lock

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

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



CLH Lock

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

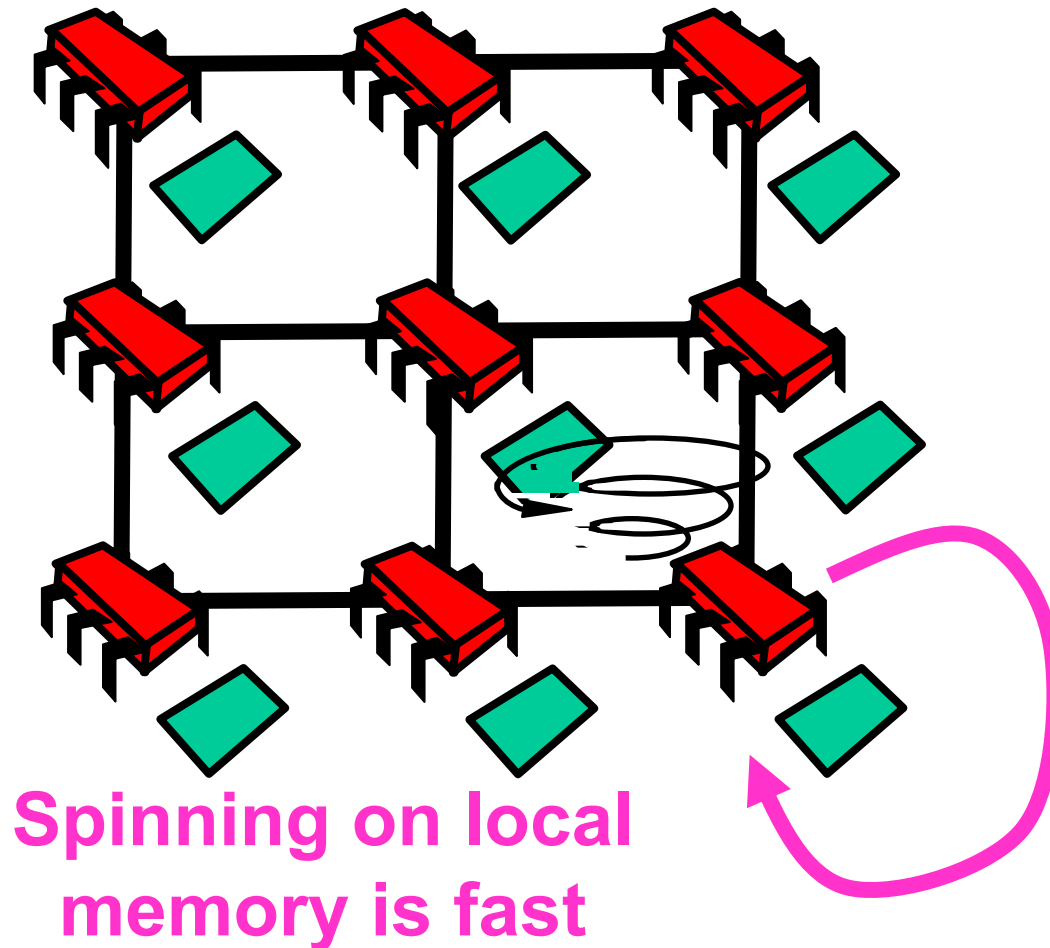


NUMA Architecturs

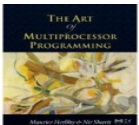
- Acronym:
 - **N**on-**U**niform **M**emory **A**rchitecture
- Illusion:
 - Flat shared memory
- Truth:
 - No caches (sometimes)
 - Some memory regions faster than others



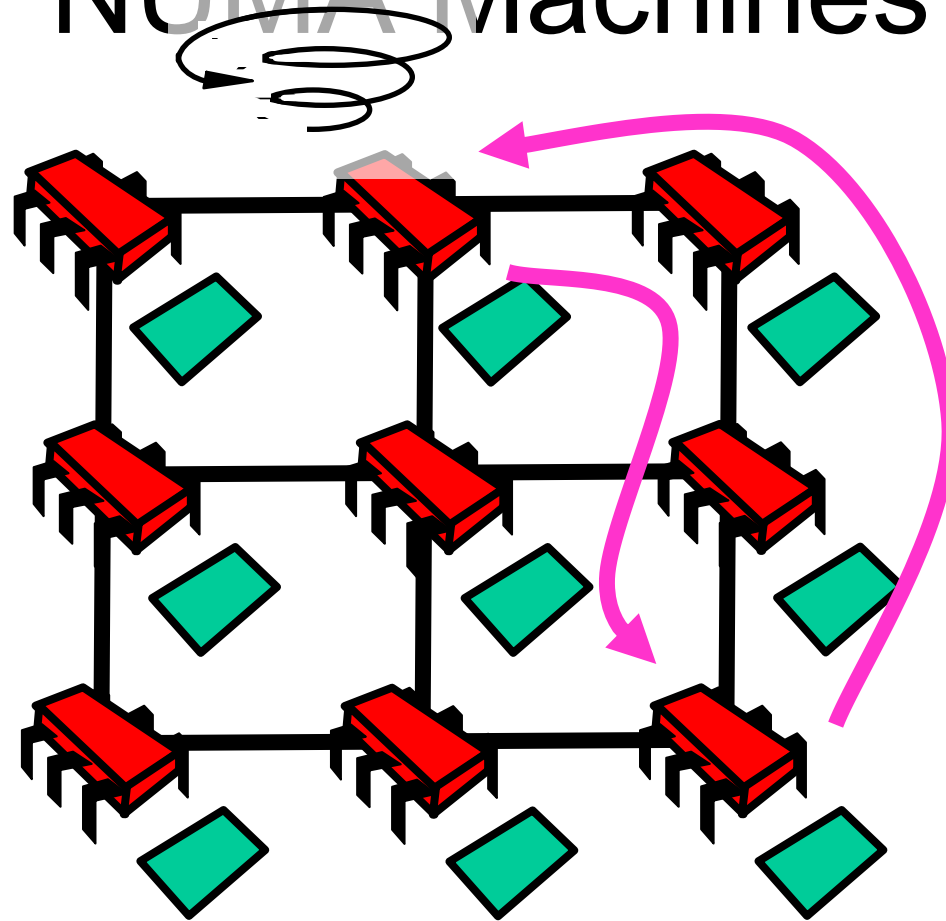
NUMA Machines



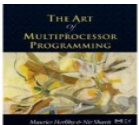
Spinning on local
memory is fast



NUMA Machines



**Spinning on remote
memory is slow**



CLH Lock

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

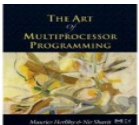
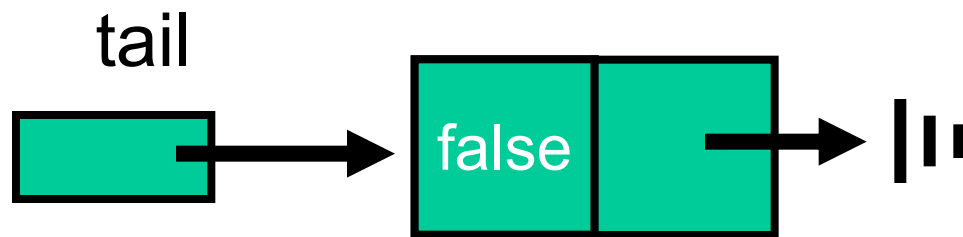
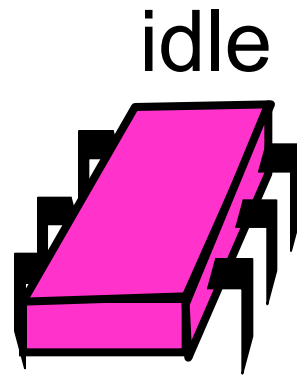


MCS Lock

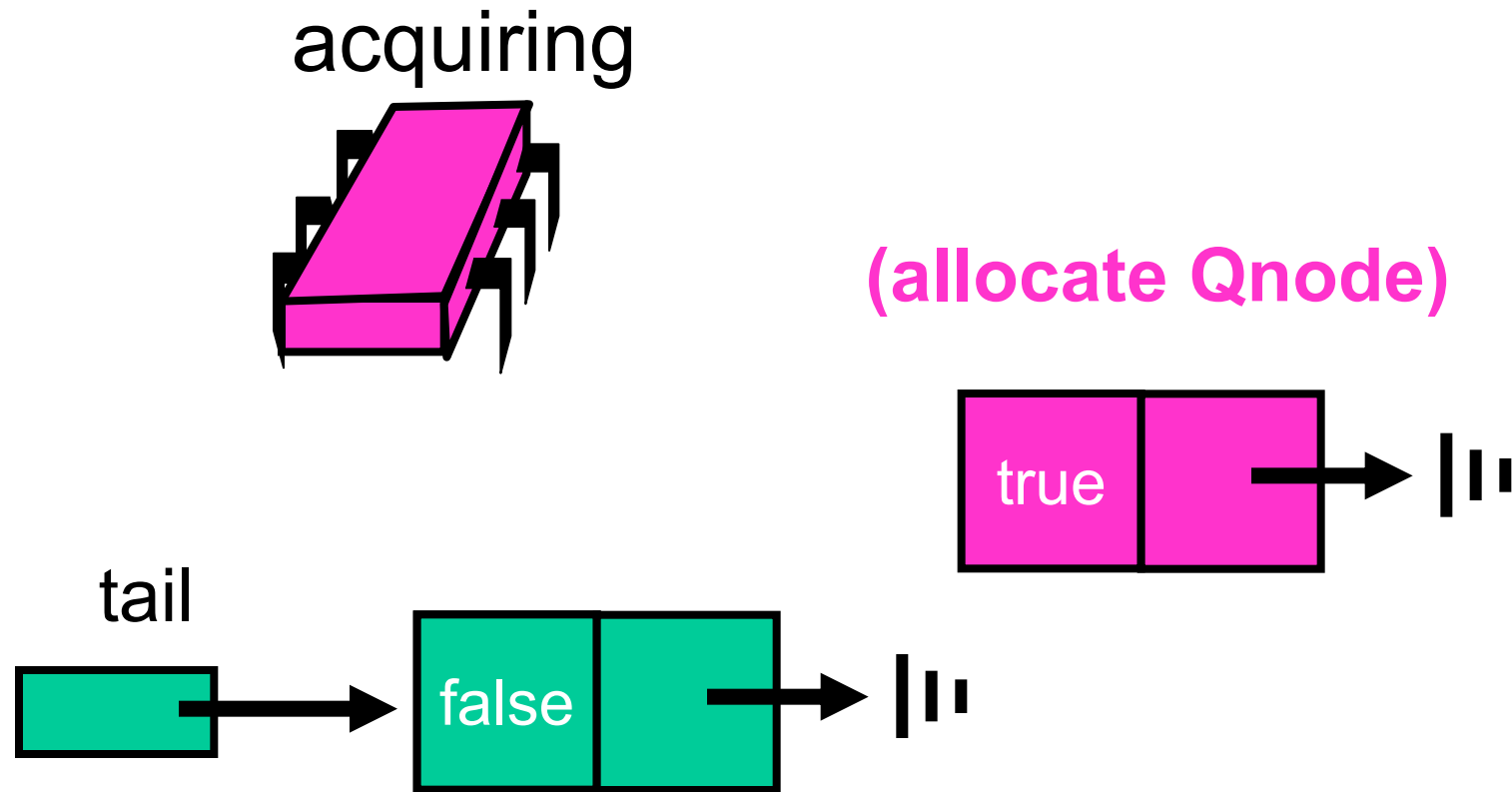
- FIFO order
- Spin on local memory only
- Small, Constant-size overhead
- Linux implemented MCS
 - kernel/locking/qspinlock.c



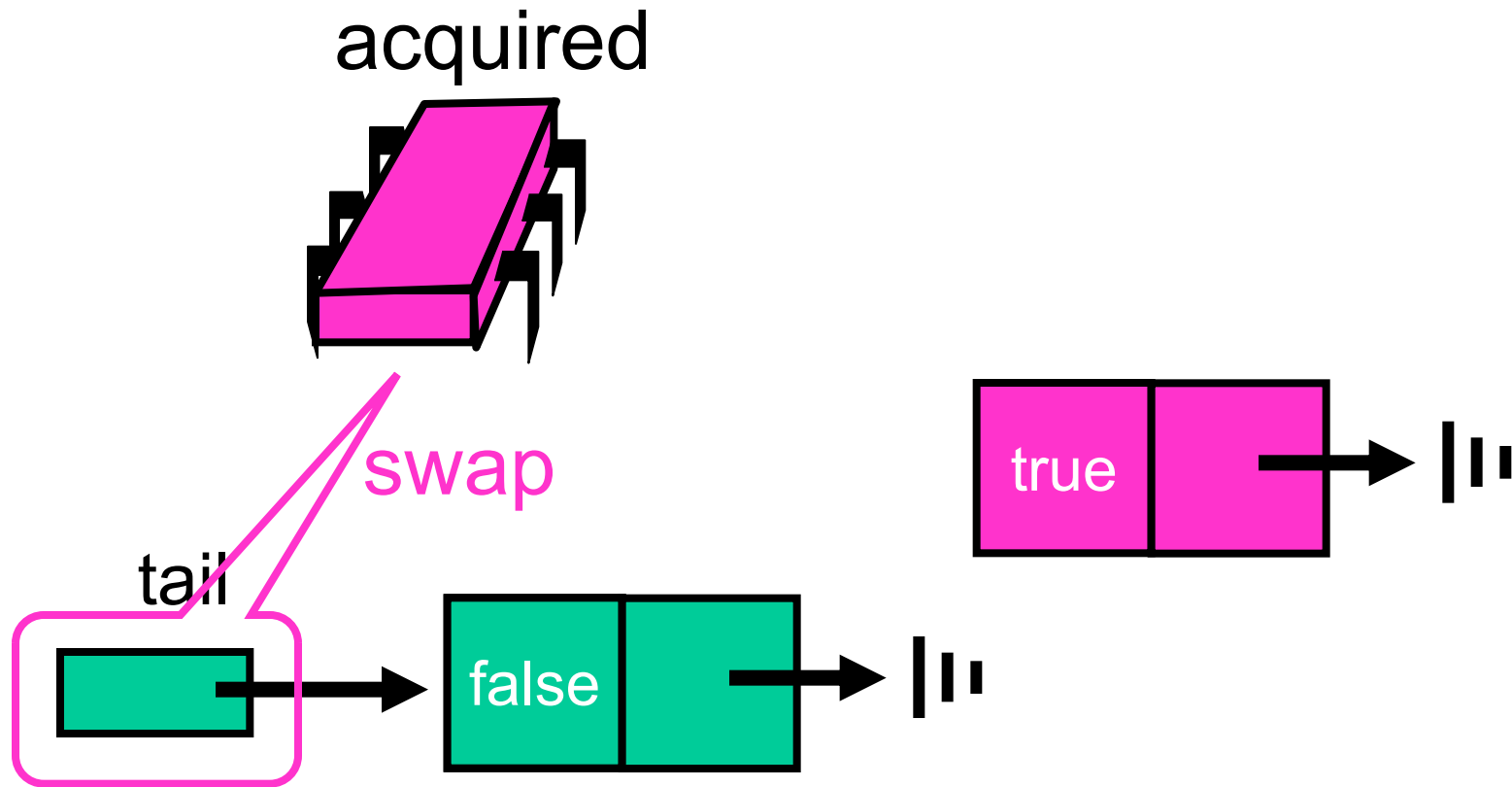
Initially



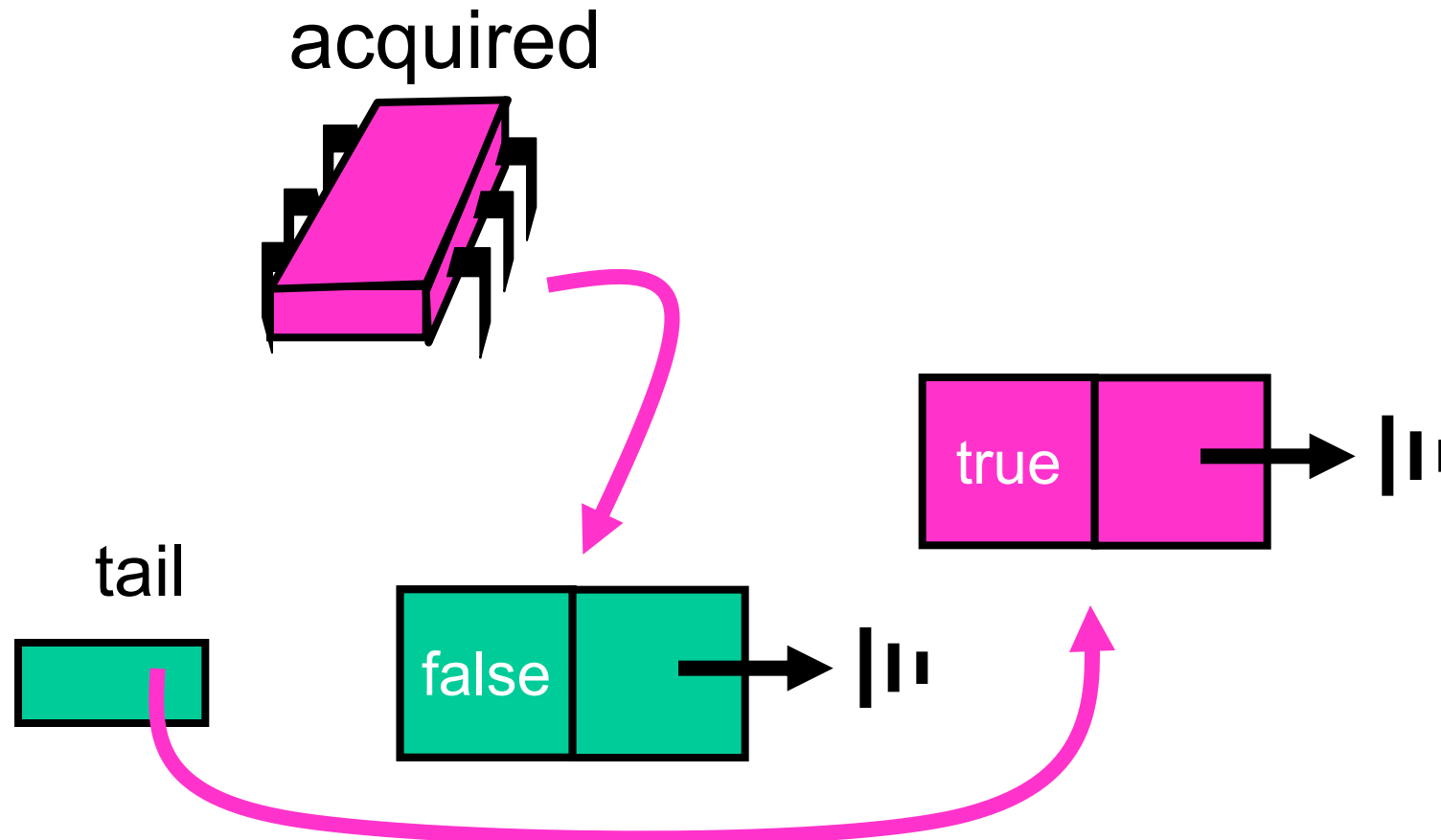
Acquiring



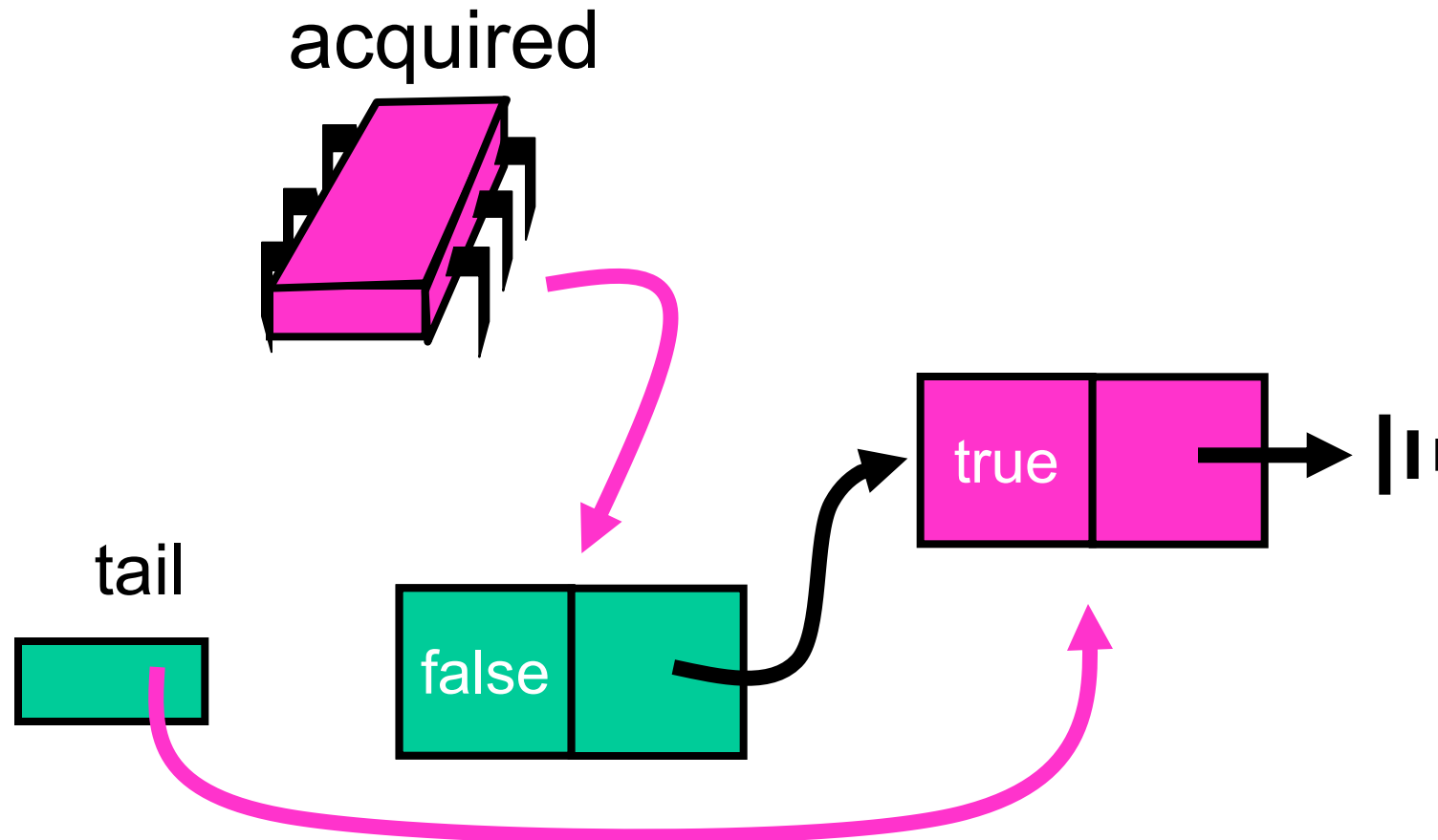
Acquiring



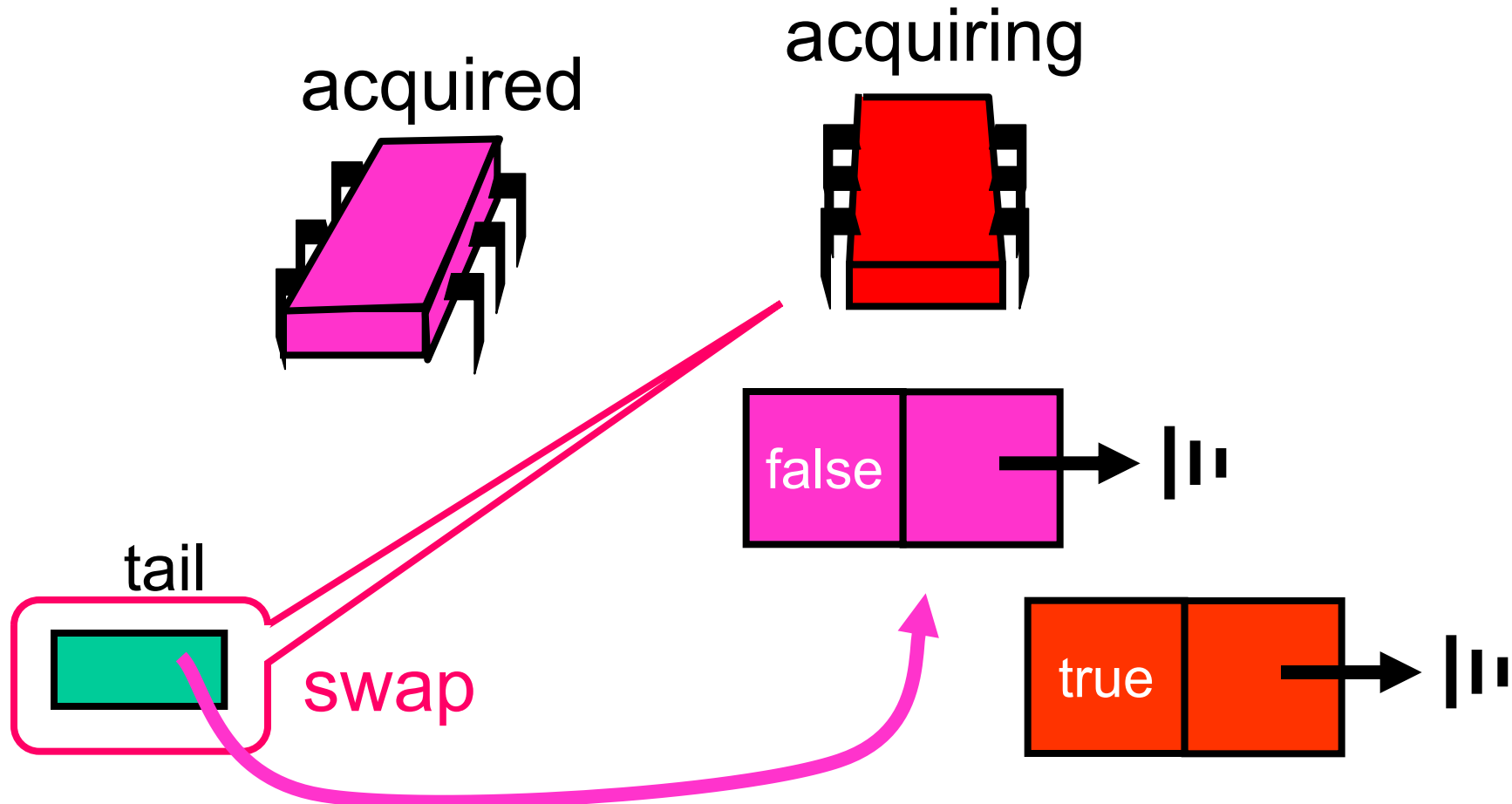
Acquiring



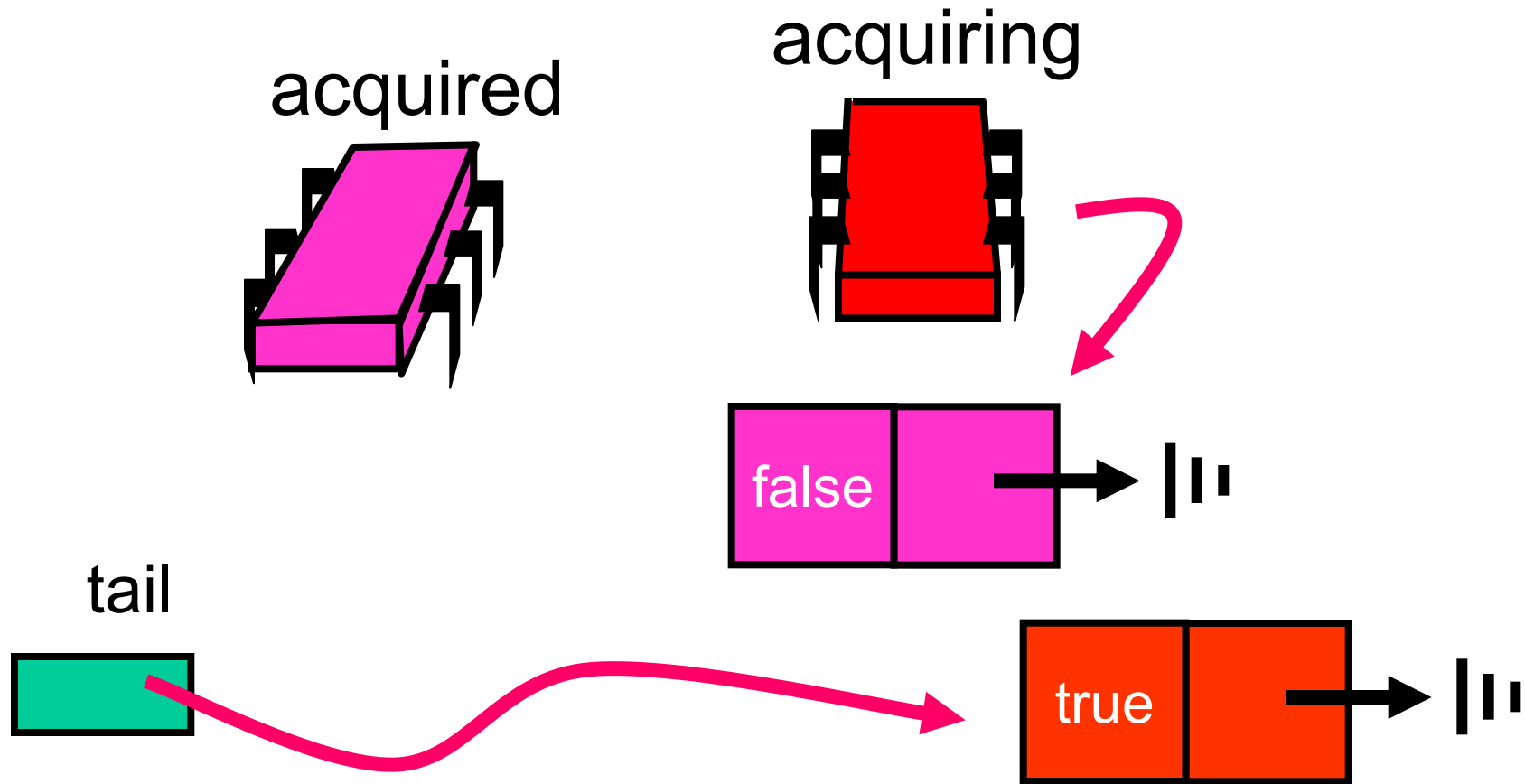
Acquired



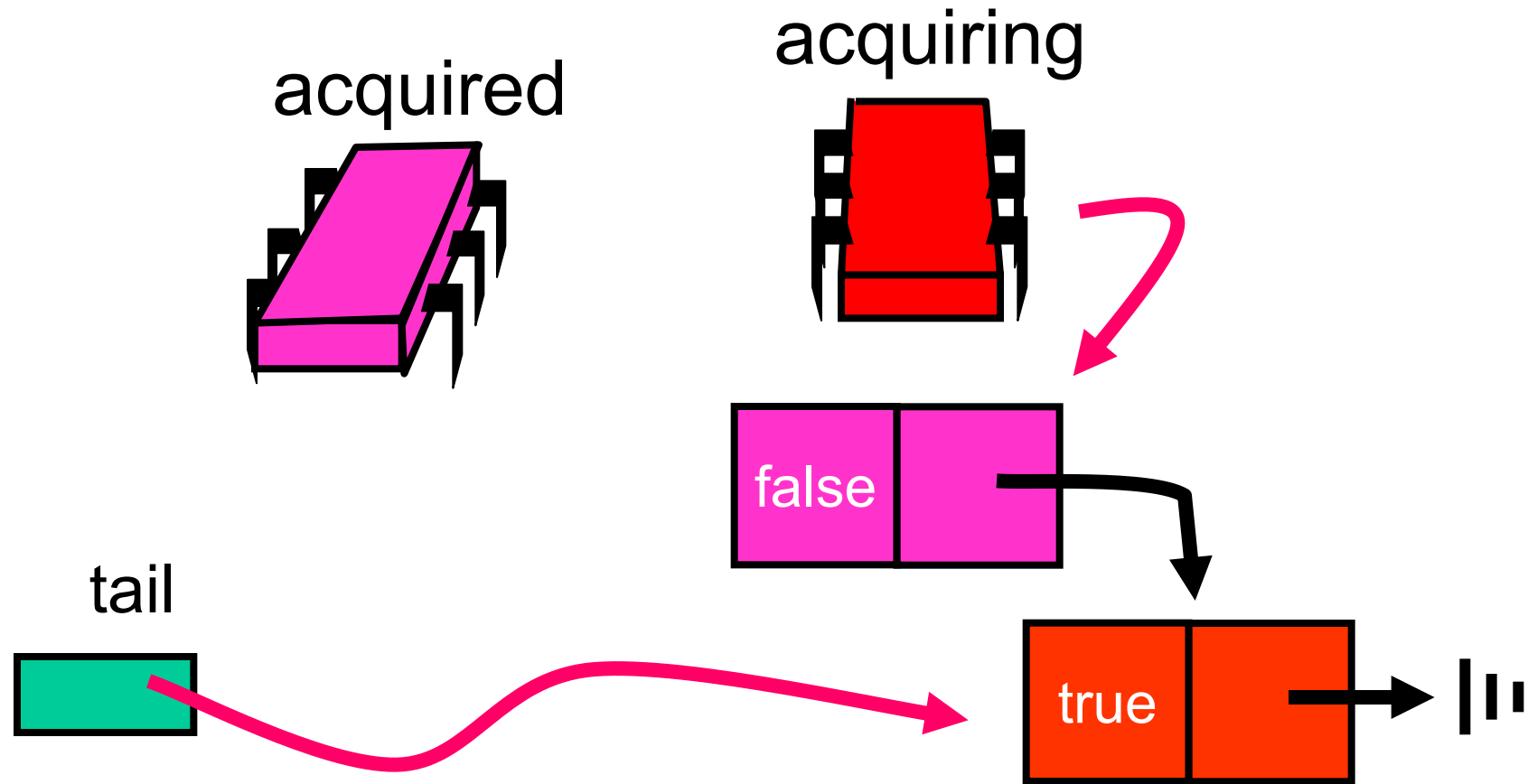
Acquiring

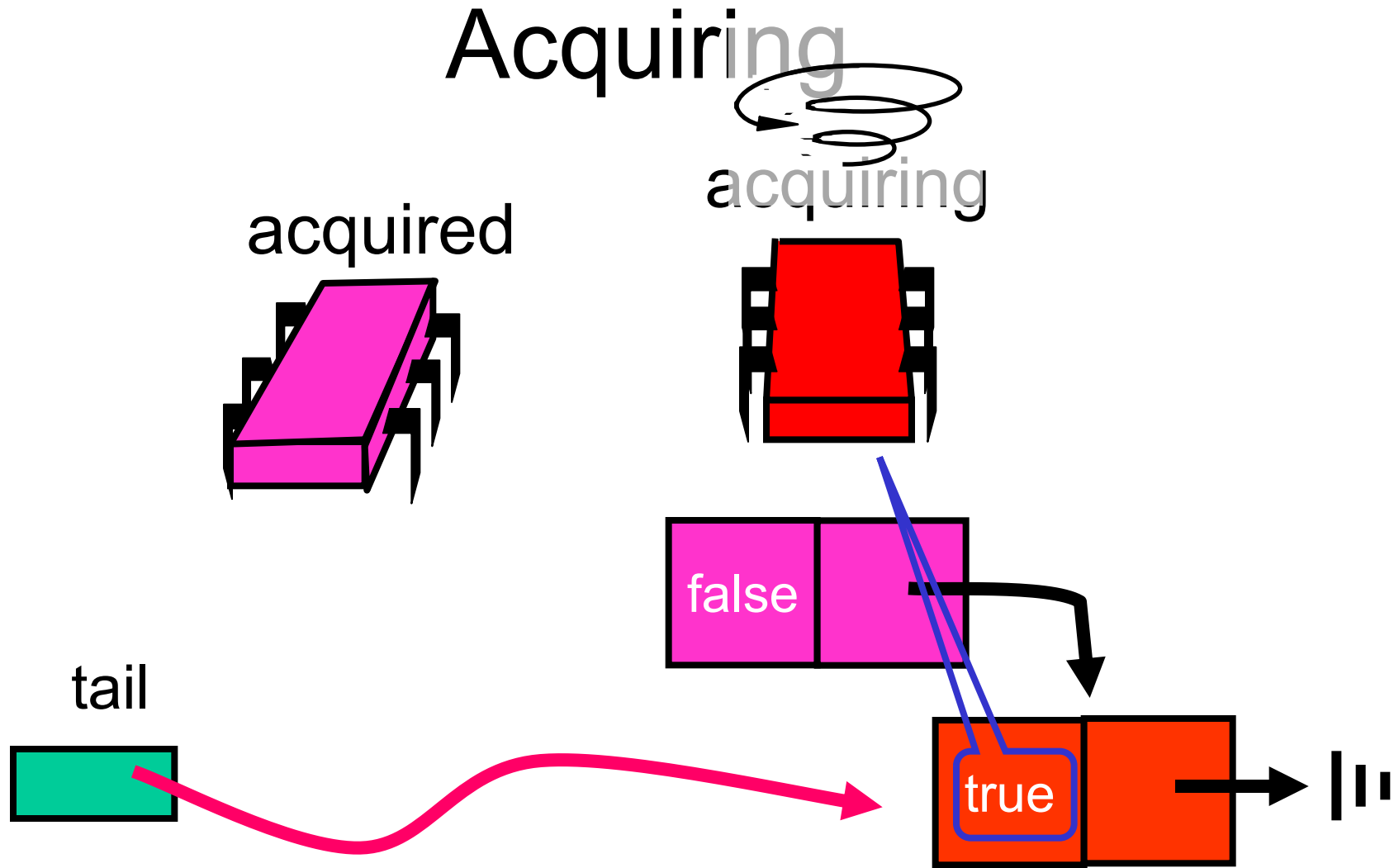


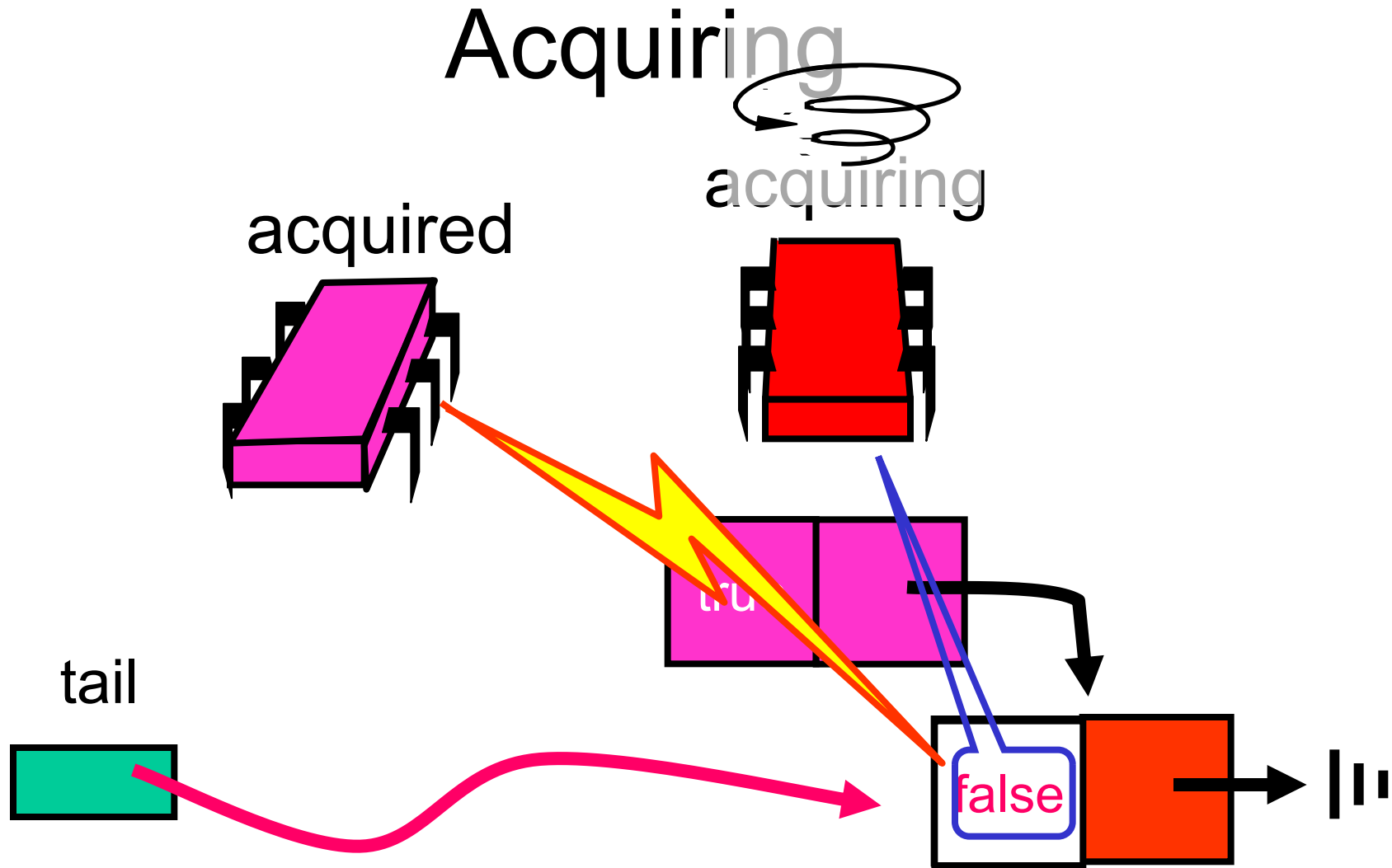
Acquiring



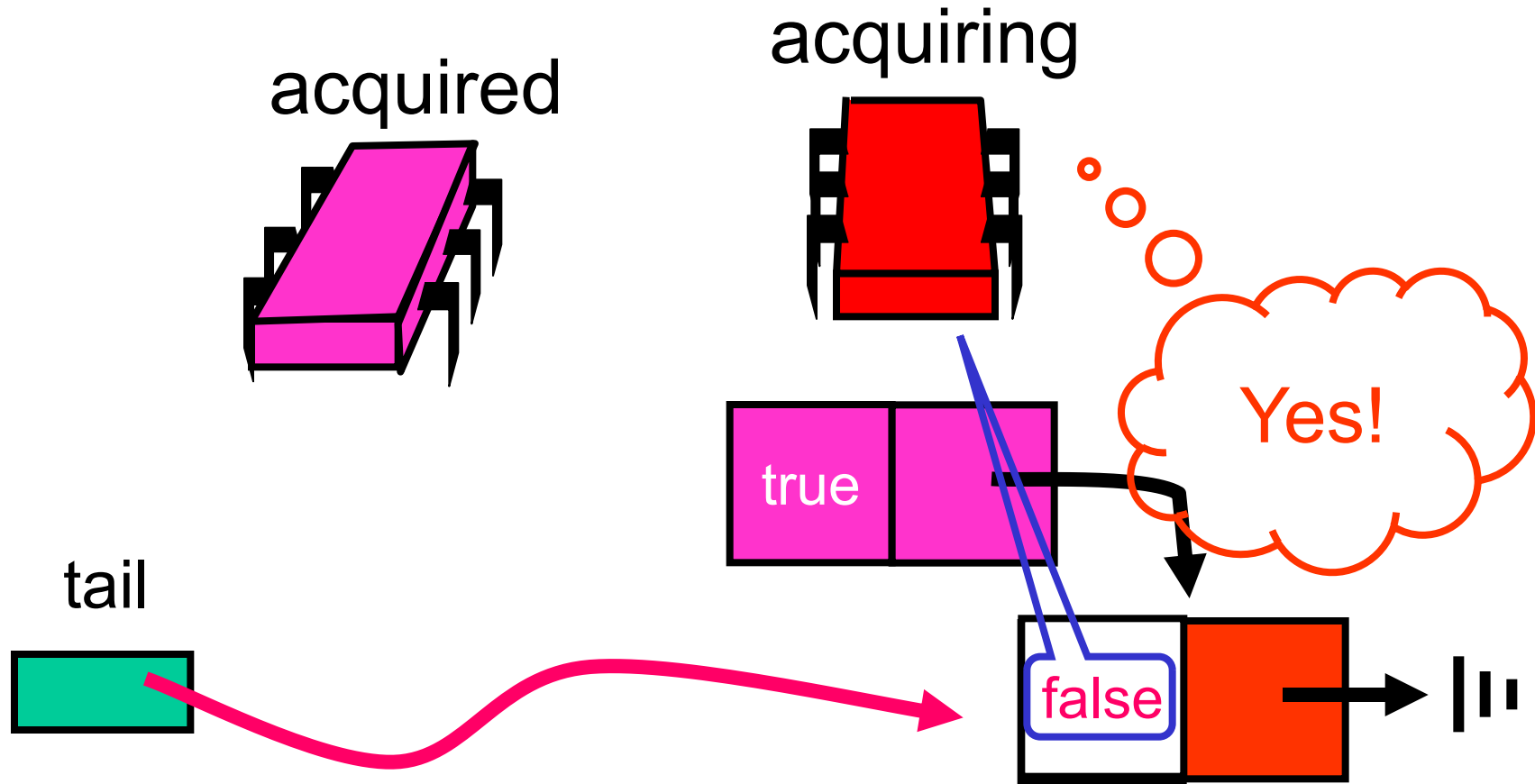
Acquiring







Acquiring



MCS Queue Lock

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

MCS Queue Lock

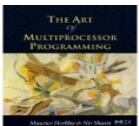
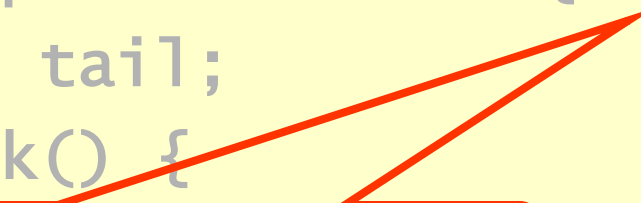
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked)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



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**



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) {}  
    }  
}
```

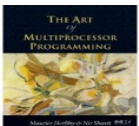
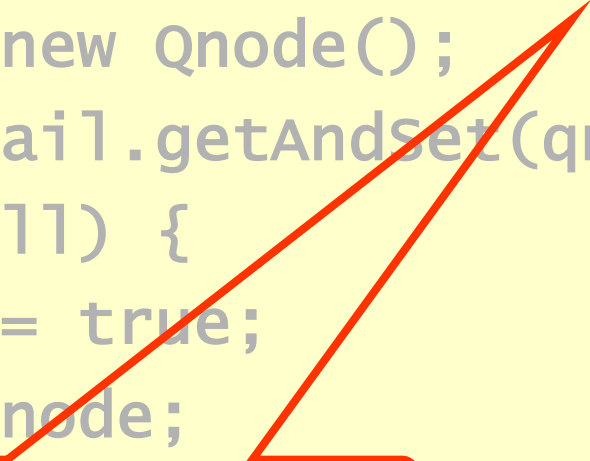
Fix if queue was non-empty



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 Unlock

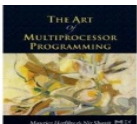
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```



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;  
    }  
}
```

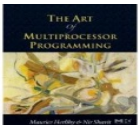
**Missing
successor
?**



MCS Queue Lock

**If really no successor,
return**

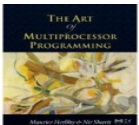
```
public void unlock() {  
    if (qnode.next == null) {  
        if (tail.CAS(qnode, null)  
            return;  
        while (qnode.next == null) {}  
    }  
    qnode.next.locked = false;  
}}
```



MCS Queue Lock

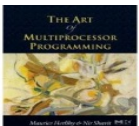
**Otherwise wait for
successor to catch up**

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

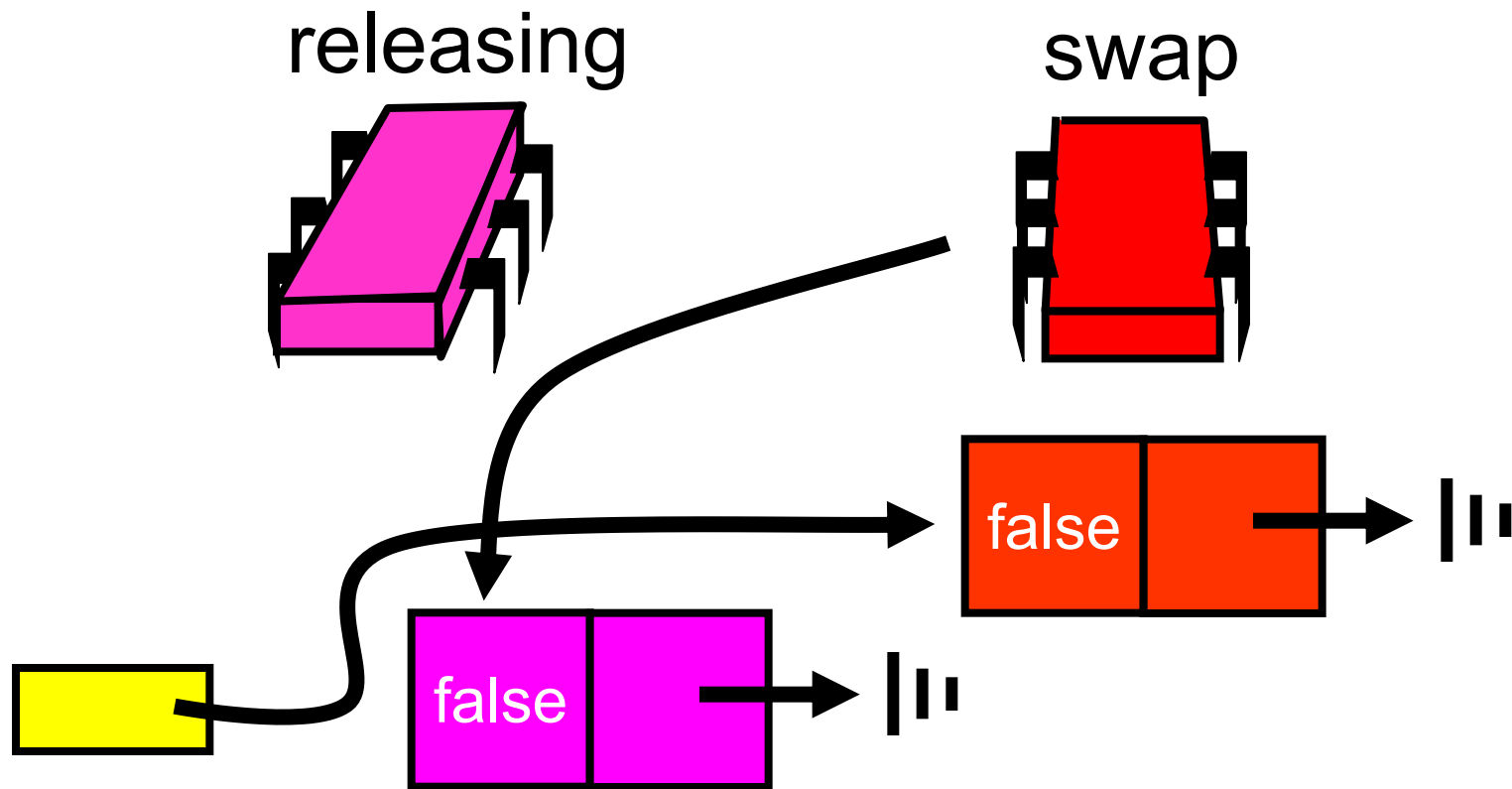


MCS Queue Lock

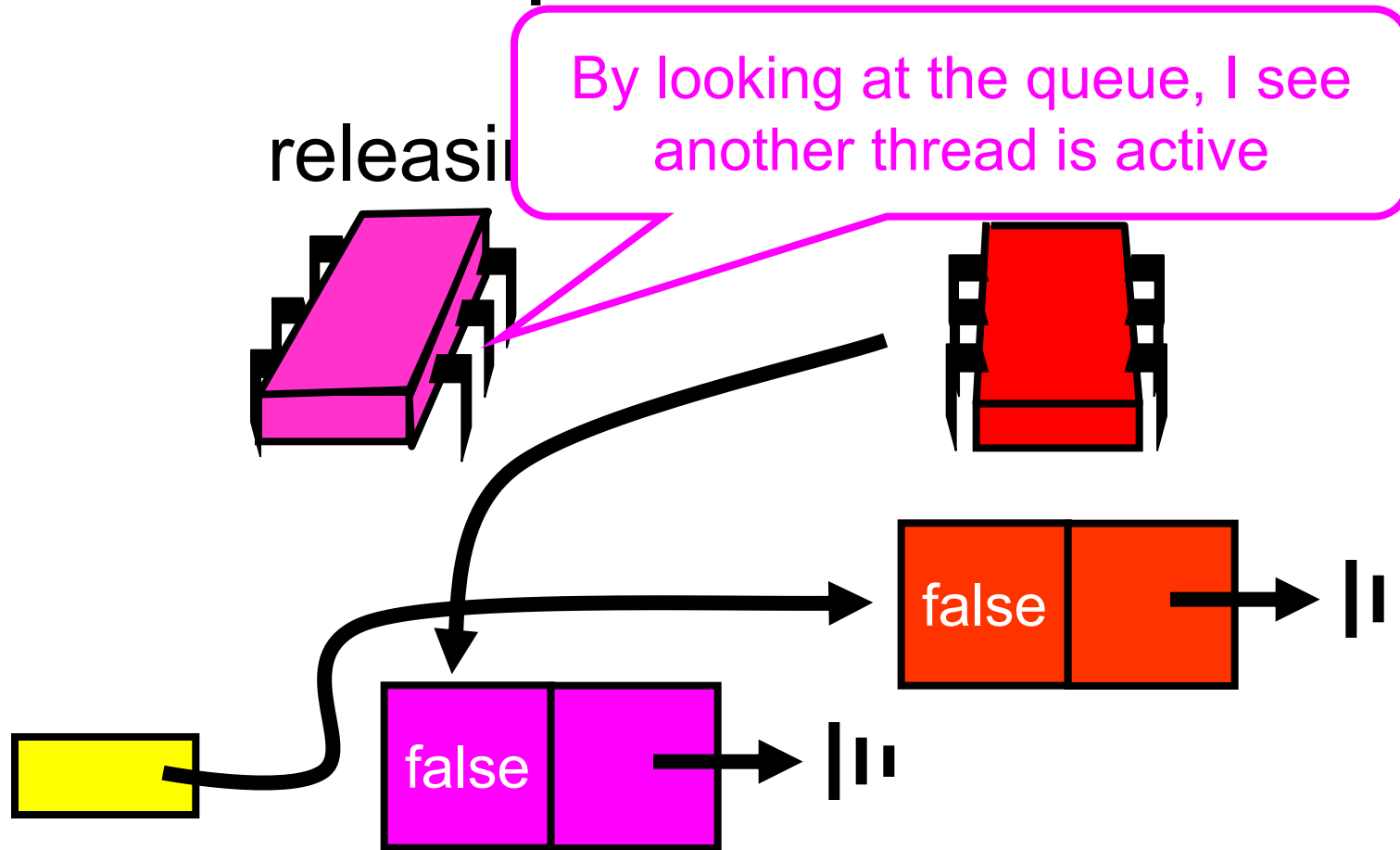
```
class MCSLock implements Lock {  
    AtomicReference<QNode> tail = new AtomicReference<QNode>();  
    public void lock() {  
        Pass lock to successor  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null))  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```



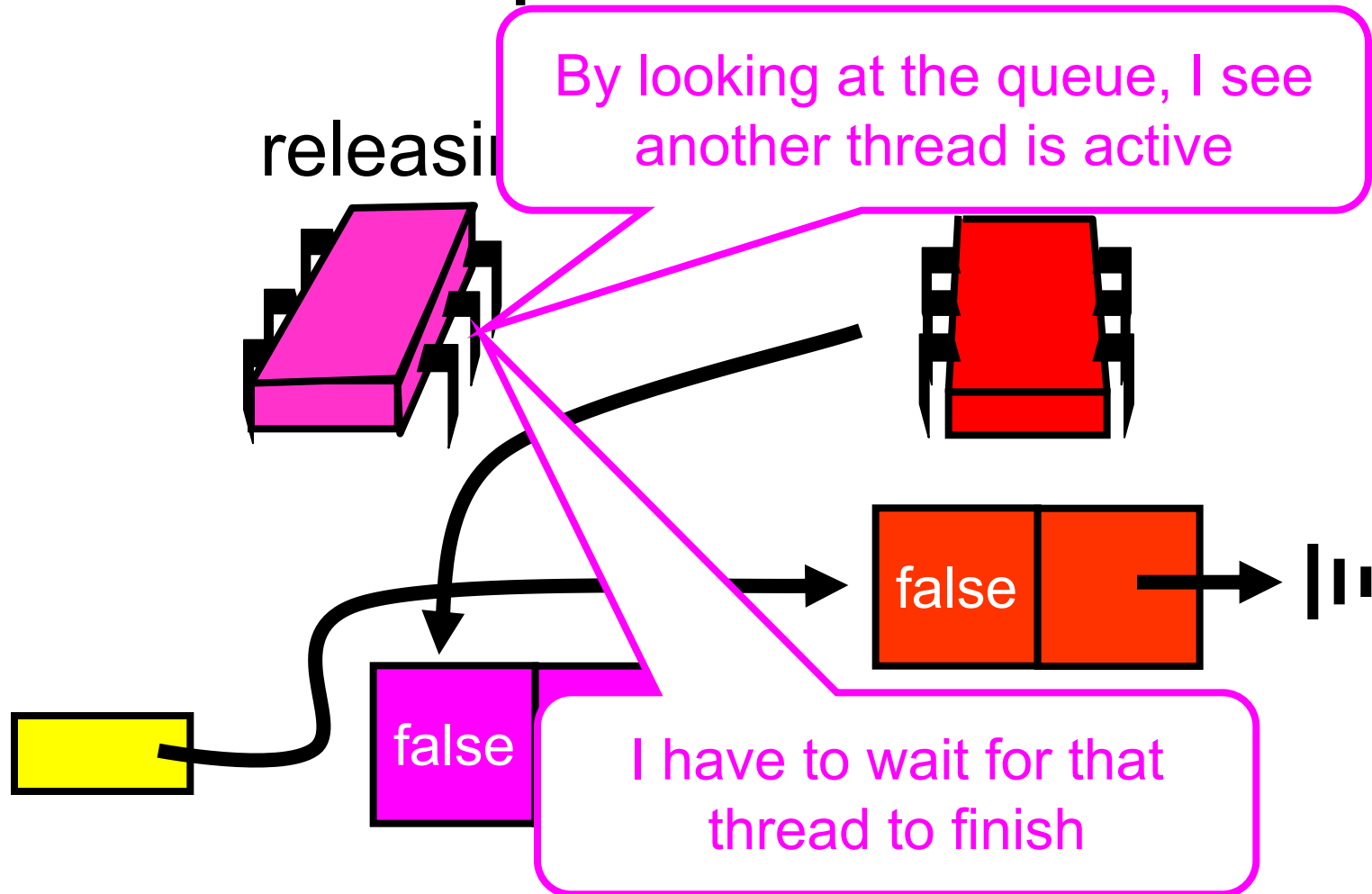
Purple Release



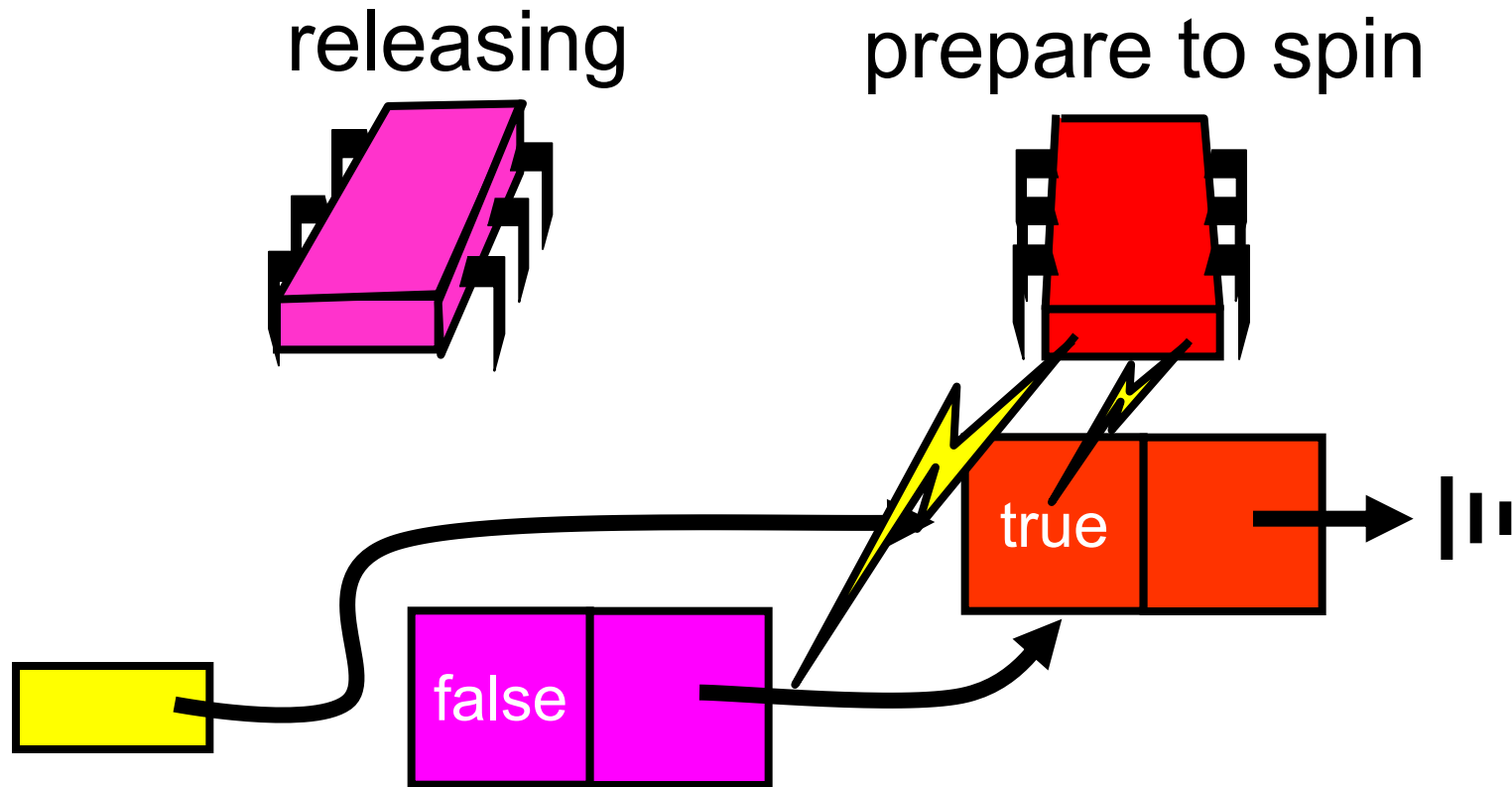
Purple Release



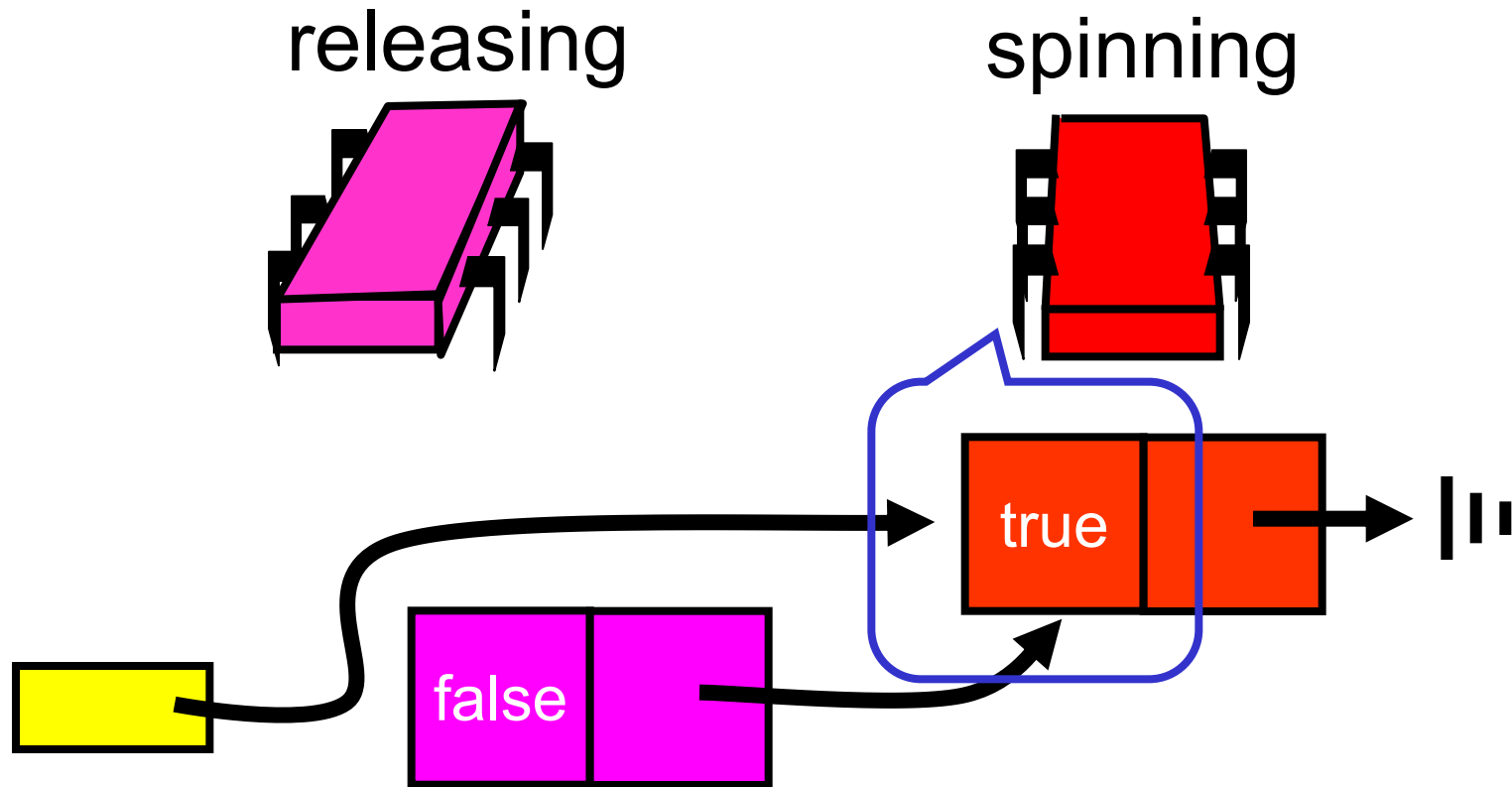
Purple Release



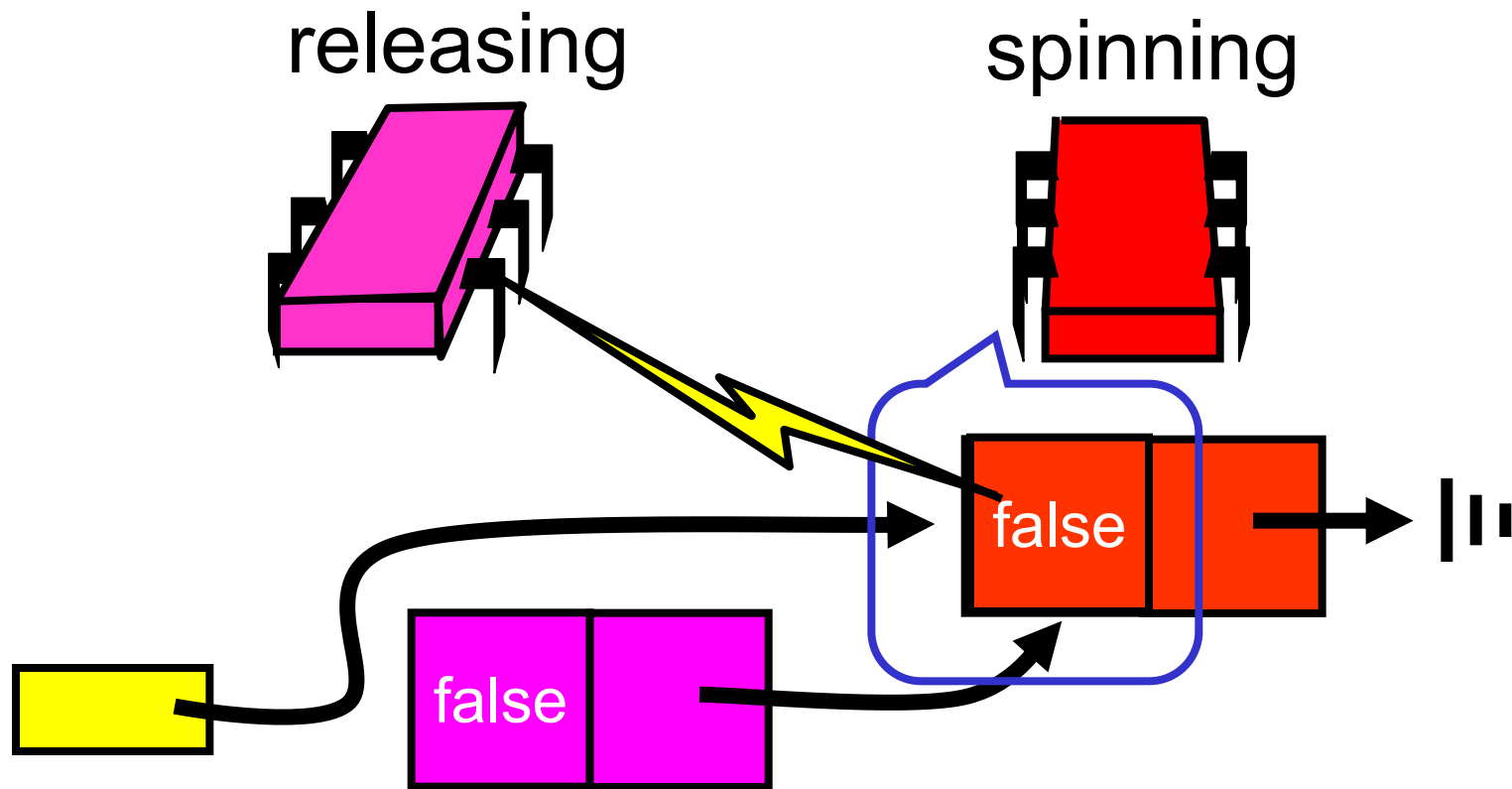
Purple Release



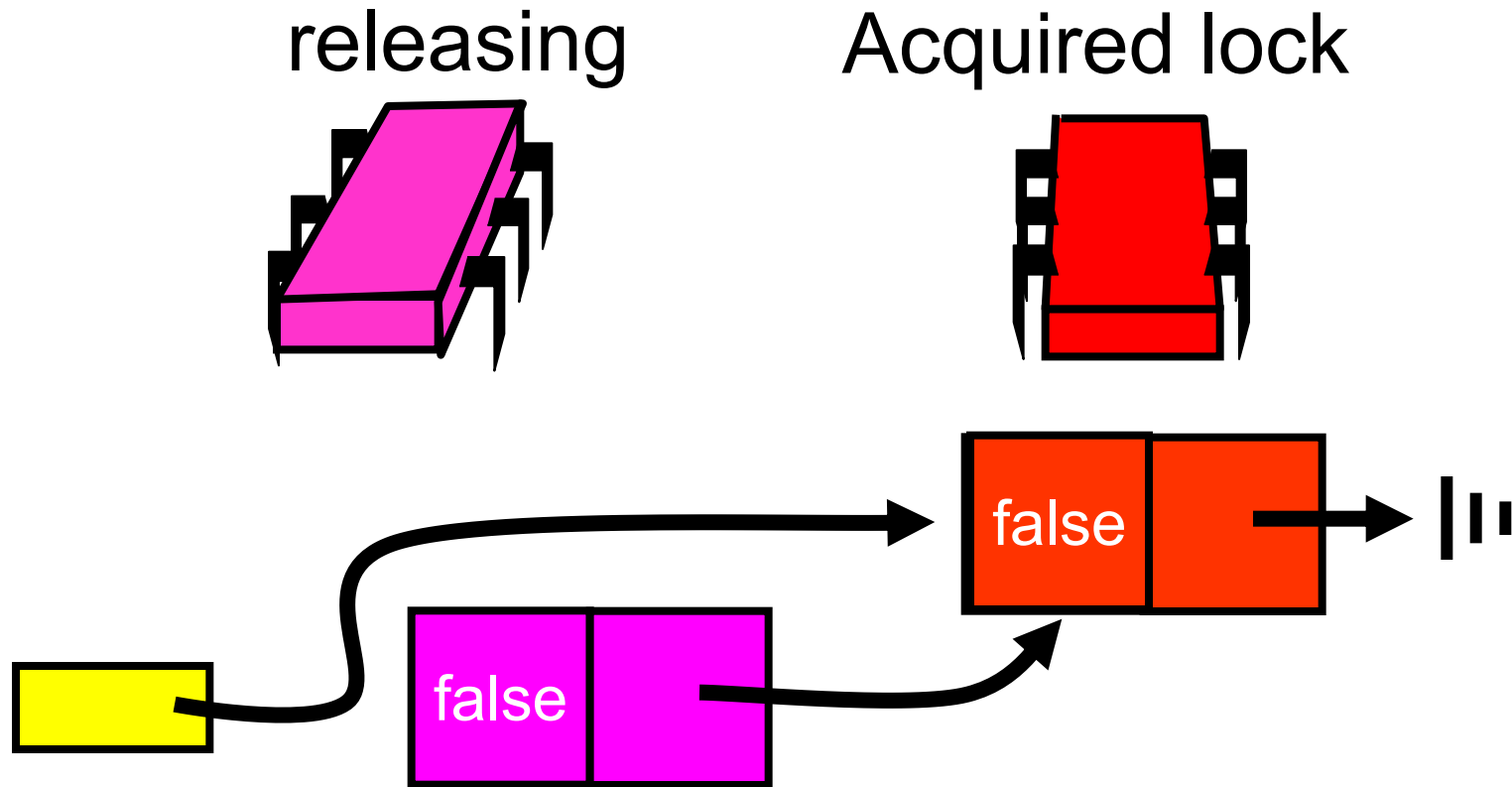
Purple Release



Purple Release



Purple Release



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

