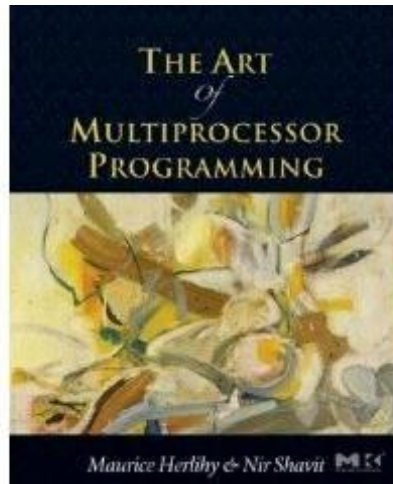


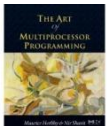
Spin Locks and Contention



Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

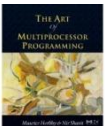
Today: Revisit Mutual Exclusion

- 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?

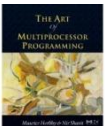
- Keep trying
 - “spin” or “busy-wait”
 - Good if delays are short
- Give up the processor
 - Good if delays are long
 - Always good on uniprocessor



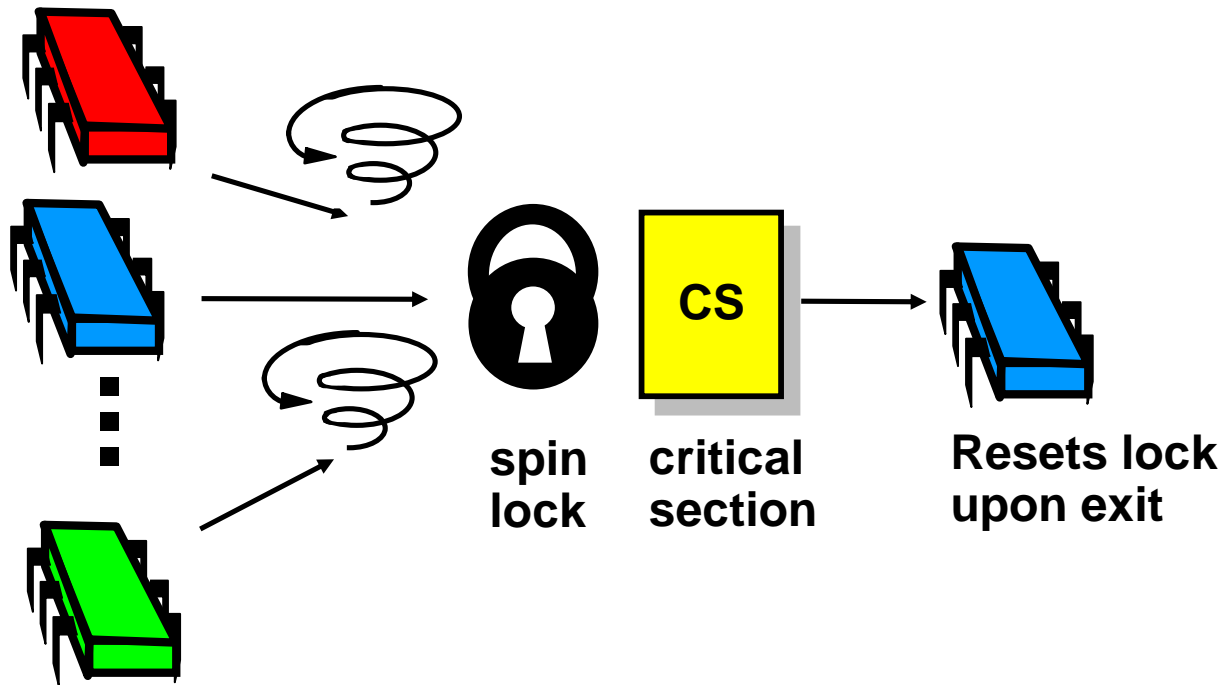
What Should you do if you can't get a lock?

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

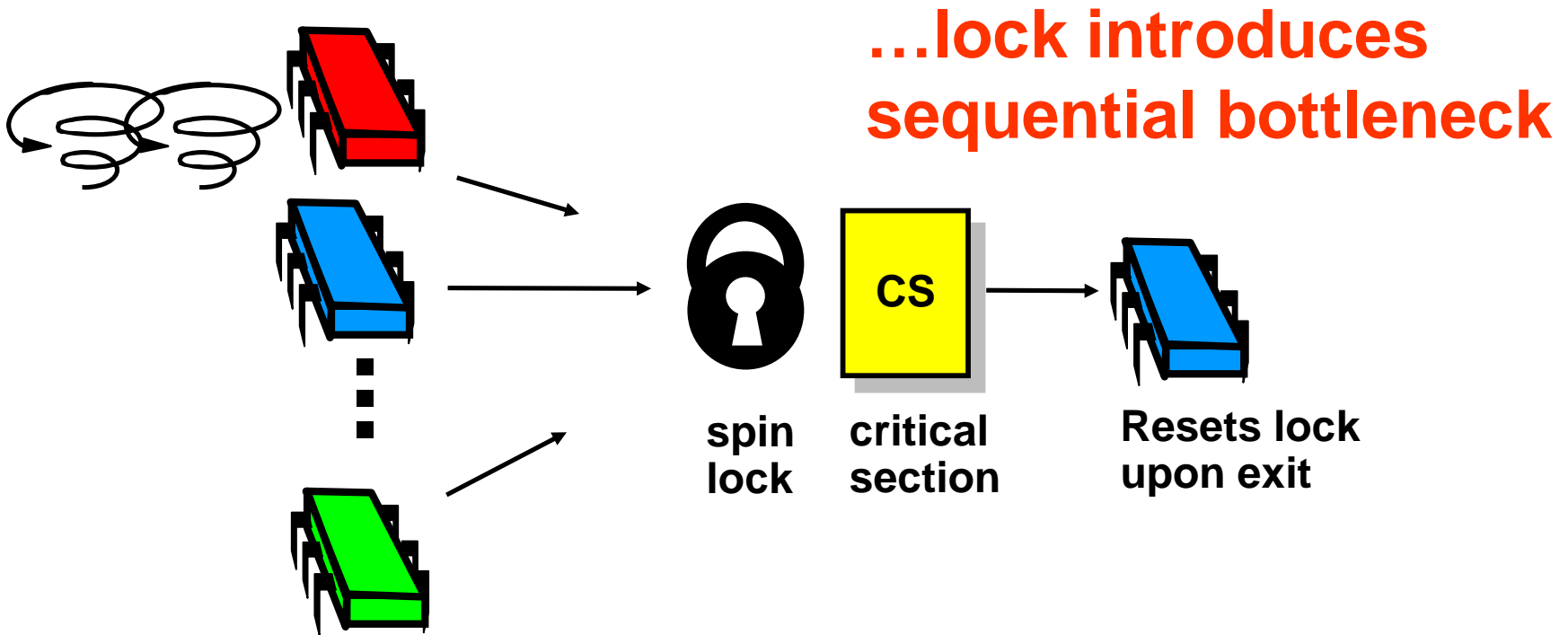
our focus



Basic Spin-Lock

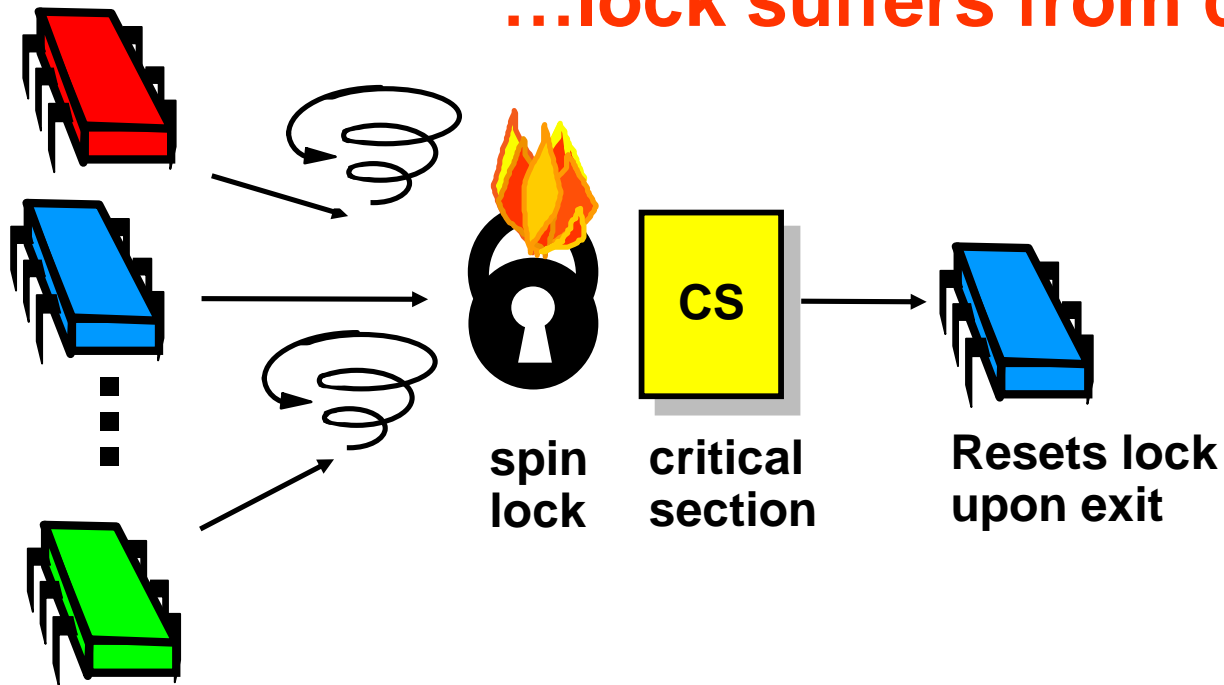


Basic Spin-Lock



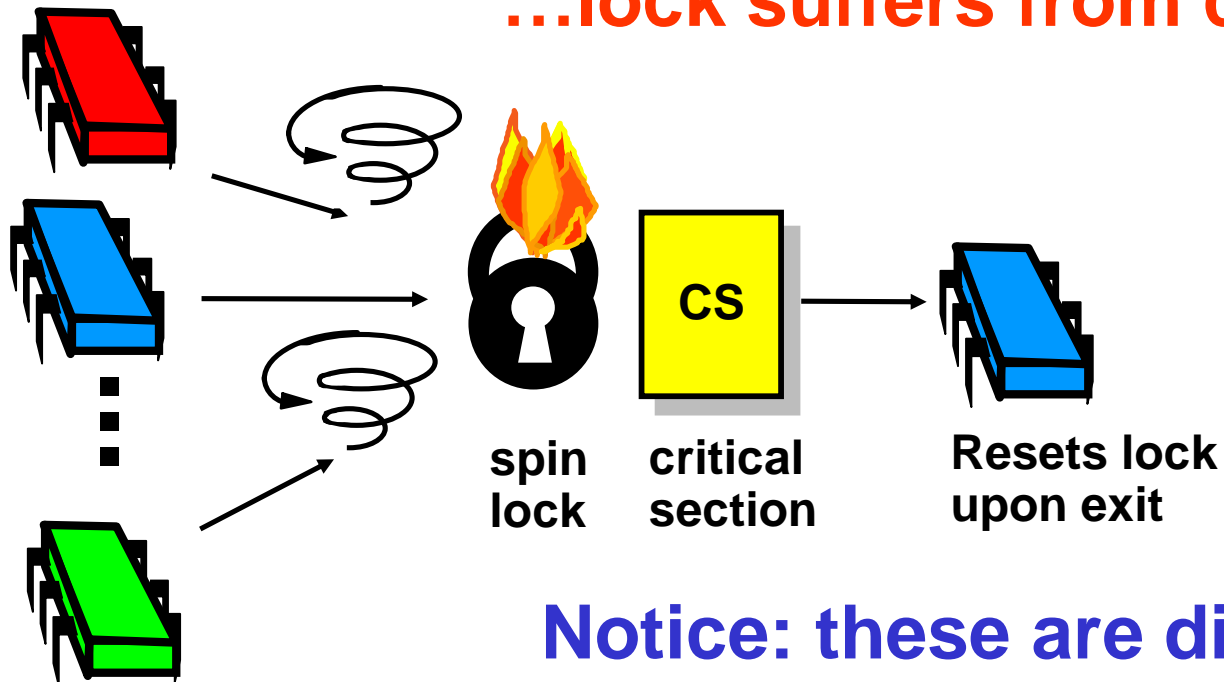
Basic Spin-Lock

...lock suffers from contention



Basic Spin-Lock

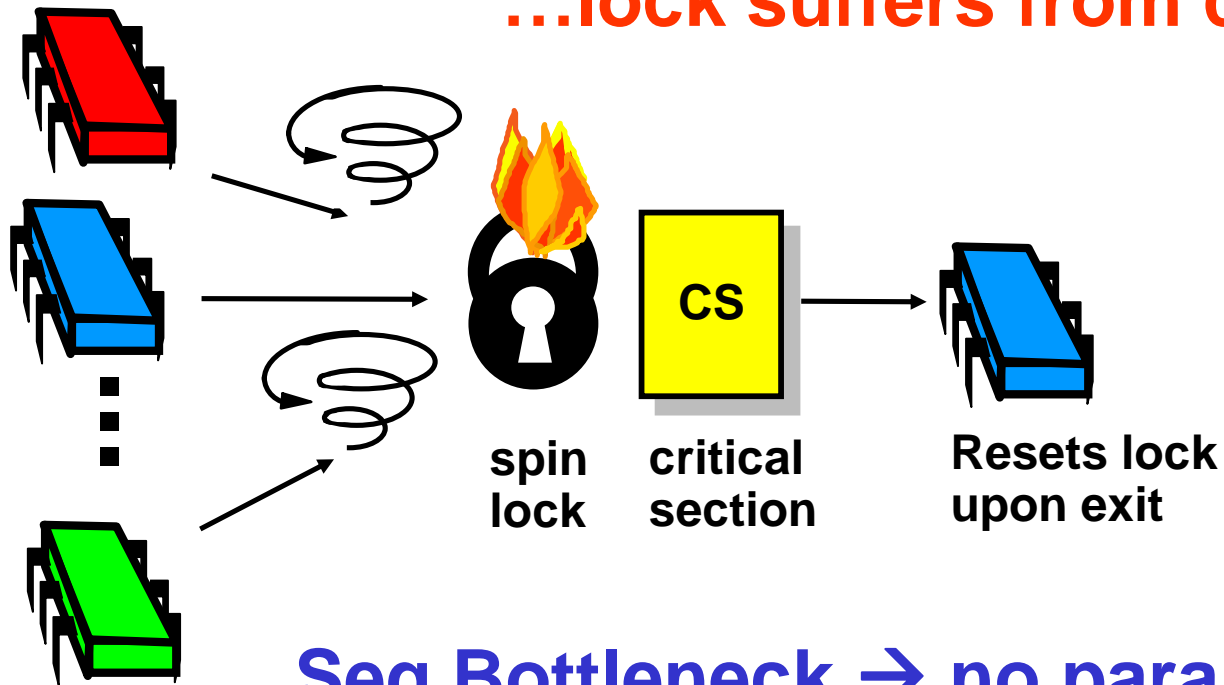
...lock suffers from contention



Notice: these are distinct phenomena

Basic Spin-Lock

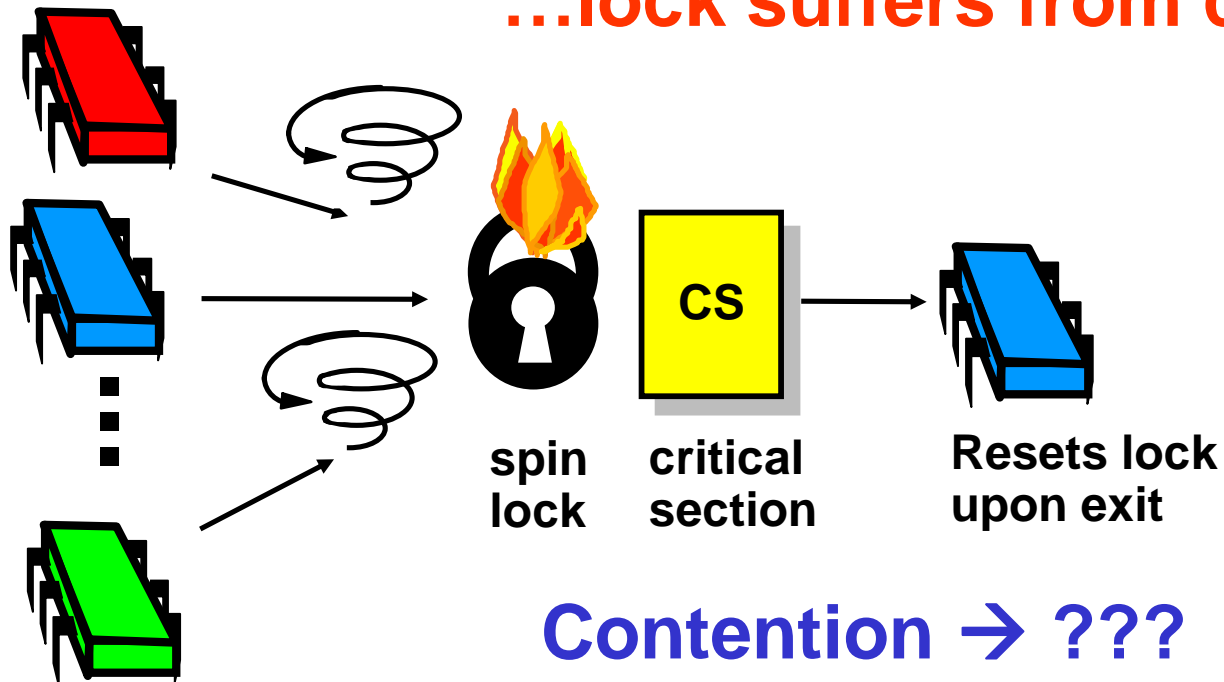
...lock suffers from contention



Seq Bottleneck → no parallelism

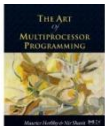
Basic Spin-Lock

...lock suffers from contention



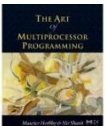
Review: Test-and-Set

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



Review: Test-and-Set

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



Review: Test-and-Set

```
public class AtomicBoolean {
```

```
    boolean value;
```

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

```
        boolean prior = value;
```

```
        value = newValue;
```

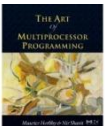
```
        return prior;
```

```
    }
```

```
}
```

Package

java.util.concurrent.atomic



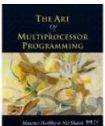
Review: Test-and-Set

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

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

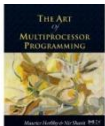
```
}
```

**Swap old and new
values**



Review: Test-and-Set

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

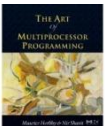


Review: Test-and-Set

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

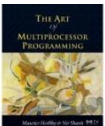
```
boolean prior = lock.getAndSet(true)
```

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



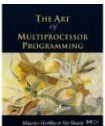
Test-and-Set Locks

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



Test-and-set Lock

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



Test-and-set Lock

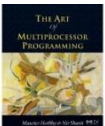
```
class TASlock {
```

```
    AtomicBoolean state =  
        new AtomicBoolean(false);
```

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

```
    void unlock() {  
        state  
    }  
}
```

Lock state is AtomicBoolean



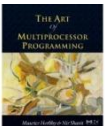
Test-and-set Lock

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

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

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

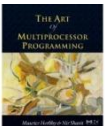
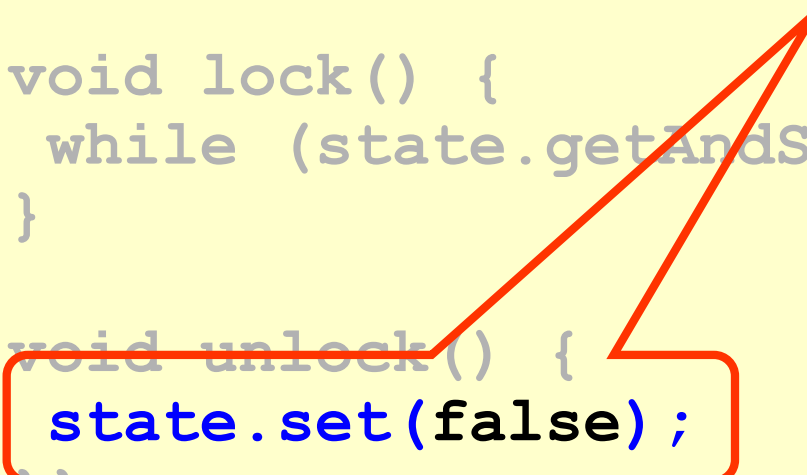
Keep trying until lock acquired



Test-and-set Lock

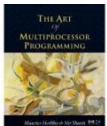
```
class TA {  
    AtomicB state;  
    new At  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    }  
}
```

**Release lock by resetting
state to false**

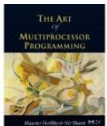
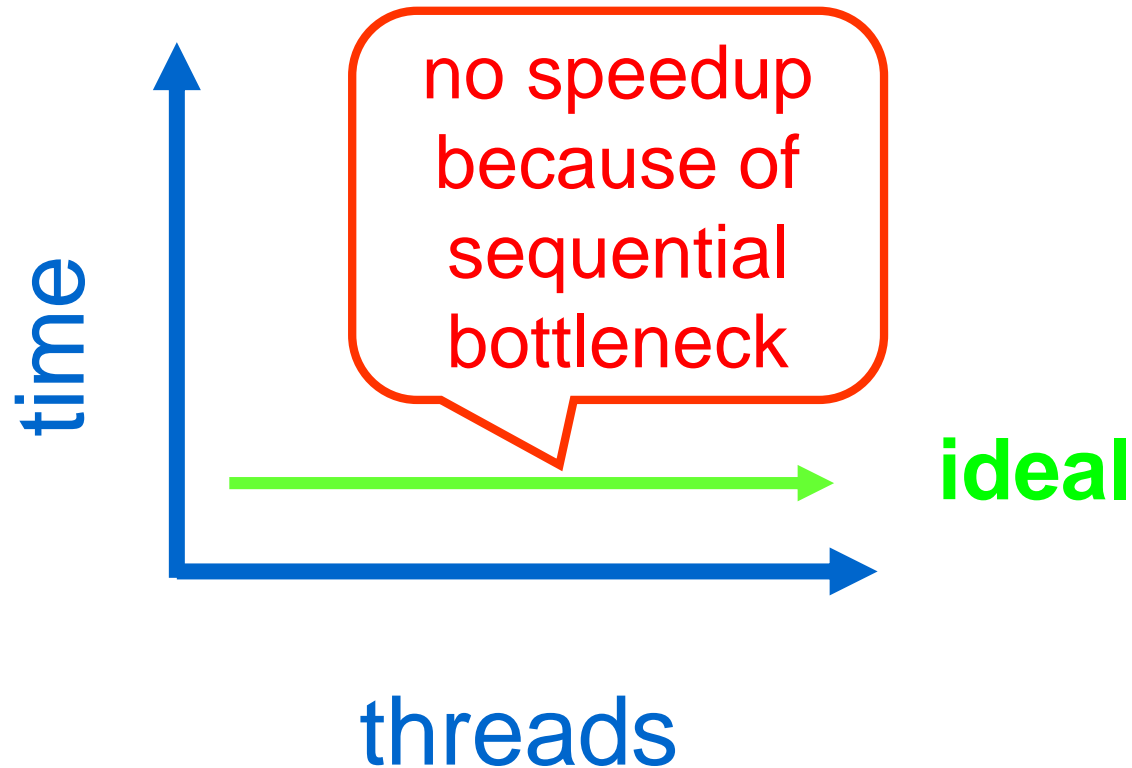


Performance

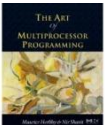
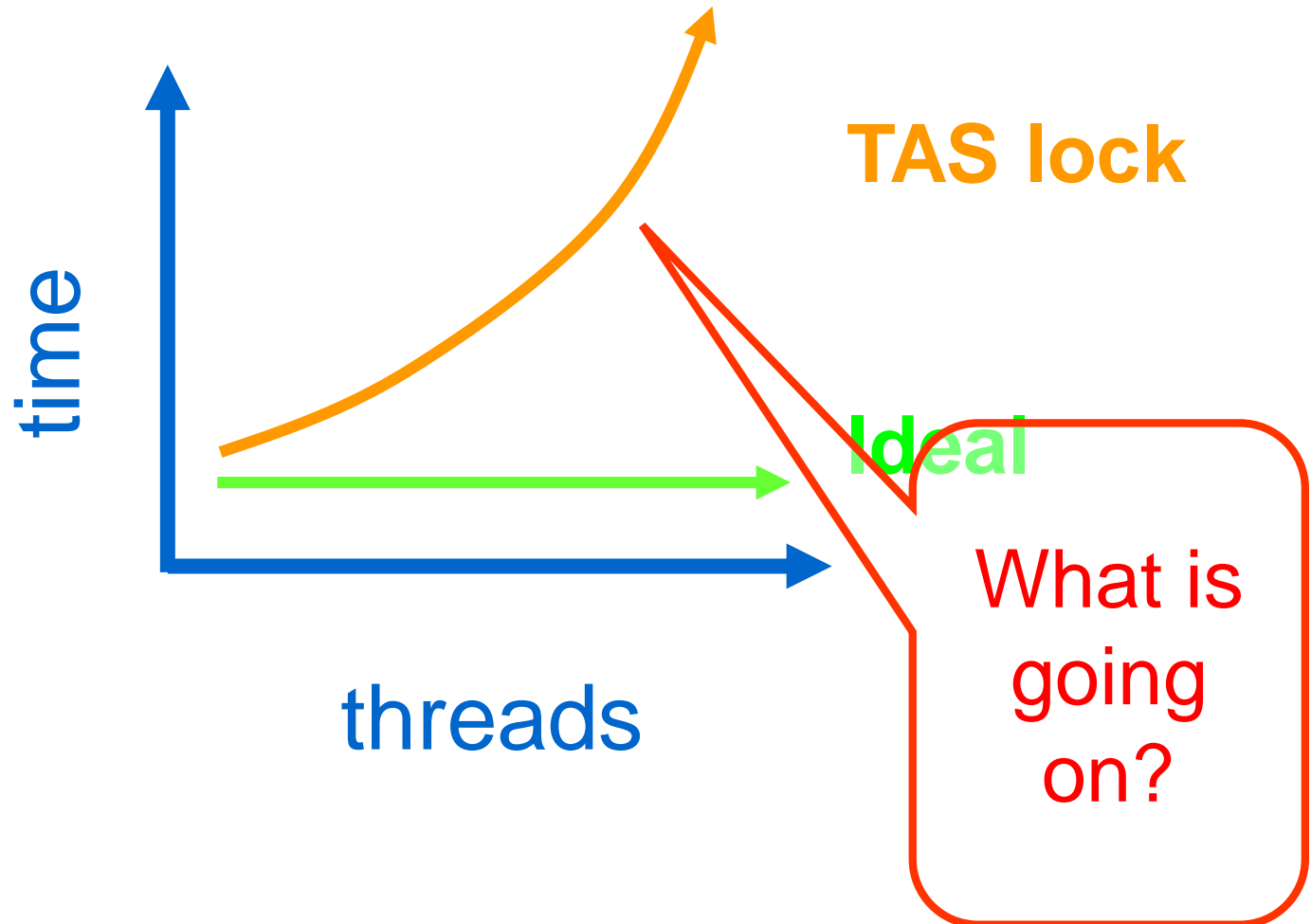
- Experiment
 - n threads
 - Increment shared counter 1 million times
- How long should it take?
- How long does it take?



Graph

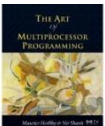


Mystery #1



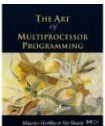
Test-and-Test-and-Set Locks

- 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

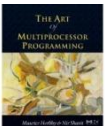
```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
        }  
    }  
}
```



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



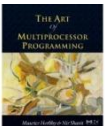
Test-and-test-and-set Lock

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

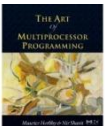
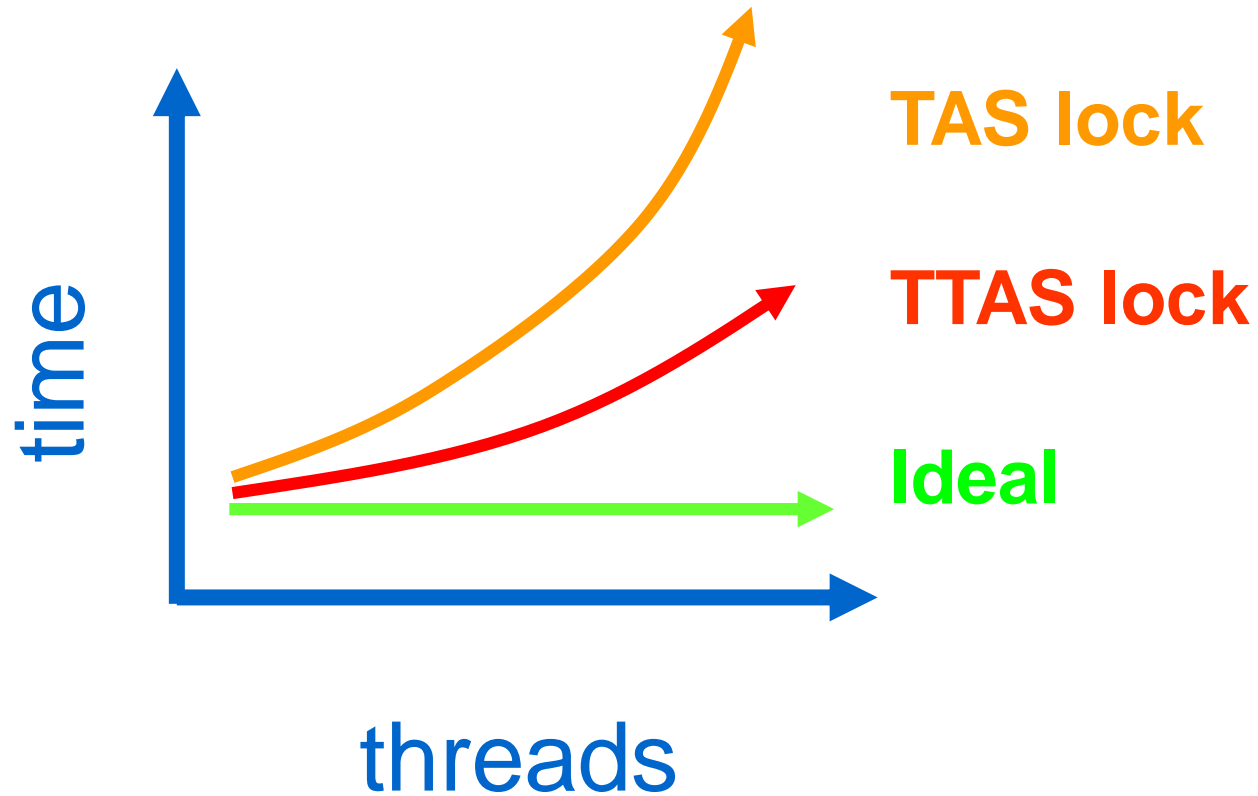
```
    void lock() {  
        while (true) {  
            while (state.get()) {}
```

```
            if (!state.getAndSet(true))  
                return;  
        }  
    }
```

Then try to
acquire it

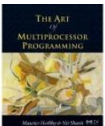


Mystery #2



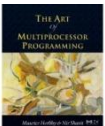
Mystery

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

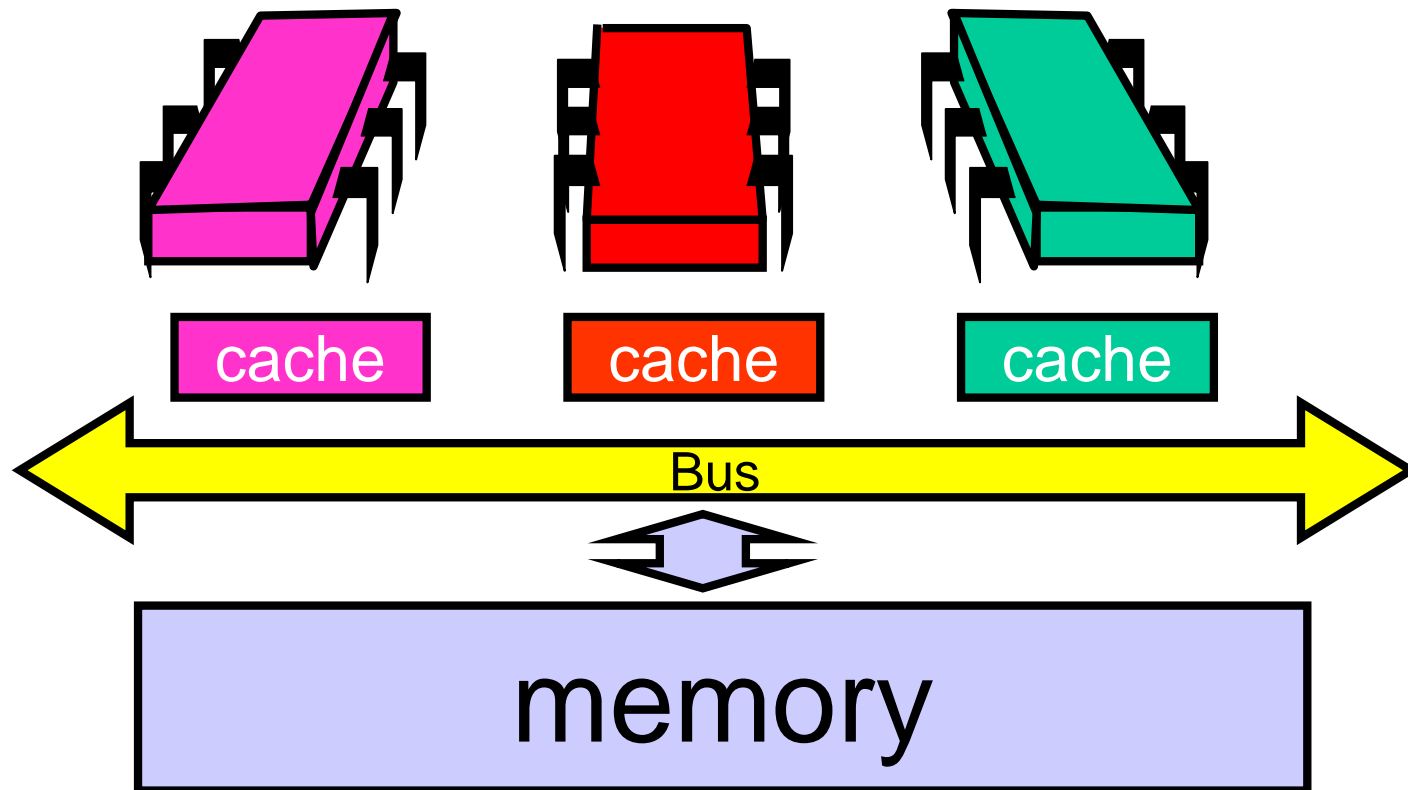


Opinion

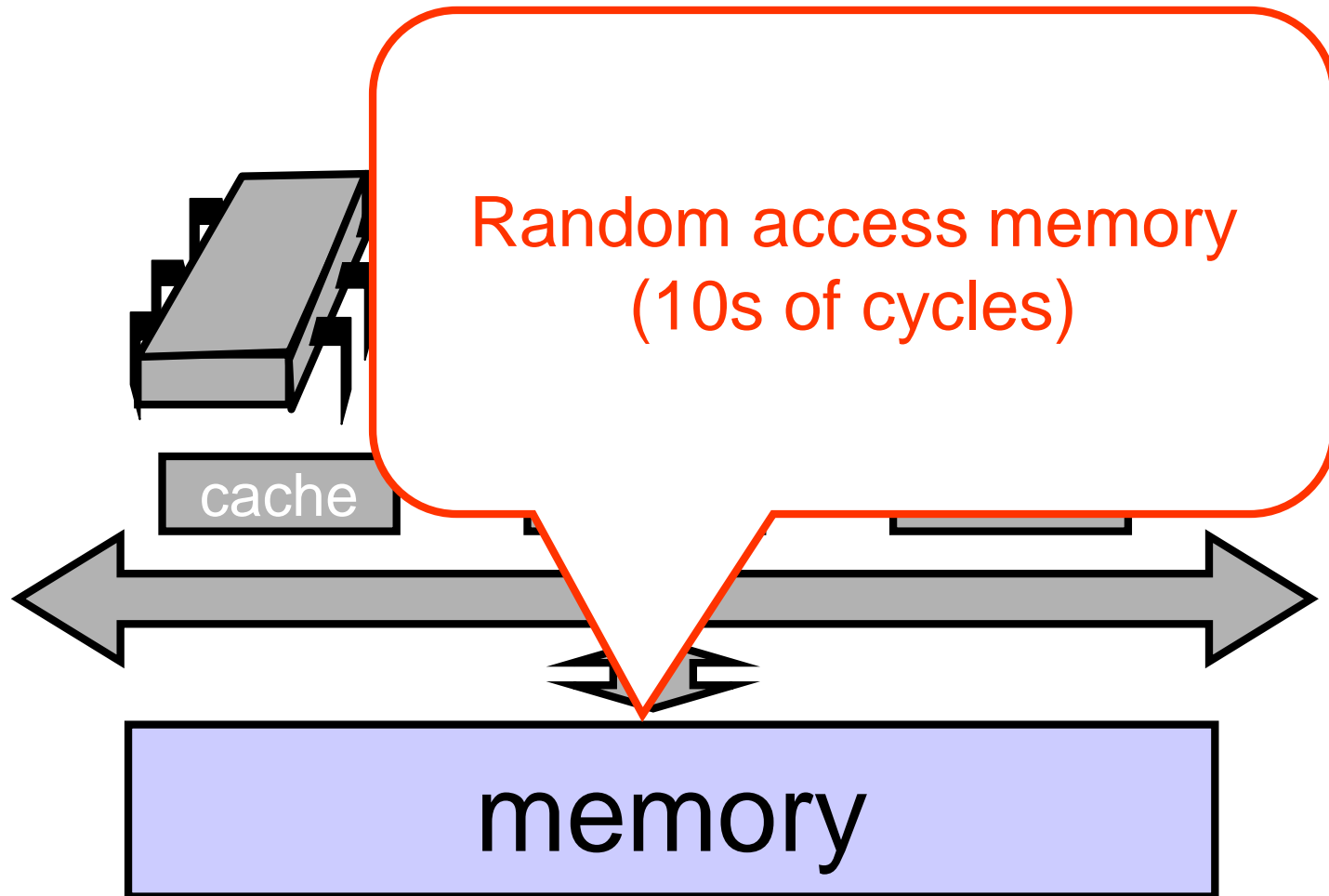
- Our memory abstraction is broken
- TAS & TTAS methods
 - Are provably the same (in our model)
 - Except they aren't (in field tests)
- Need a more detailed model ...



Bus-Based Architectures



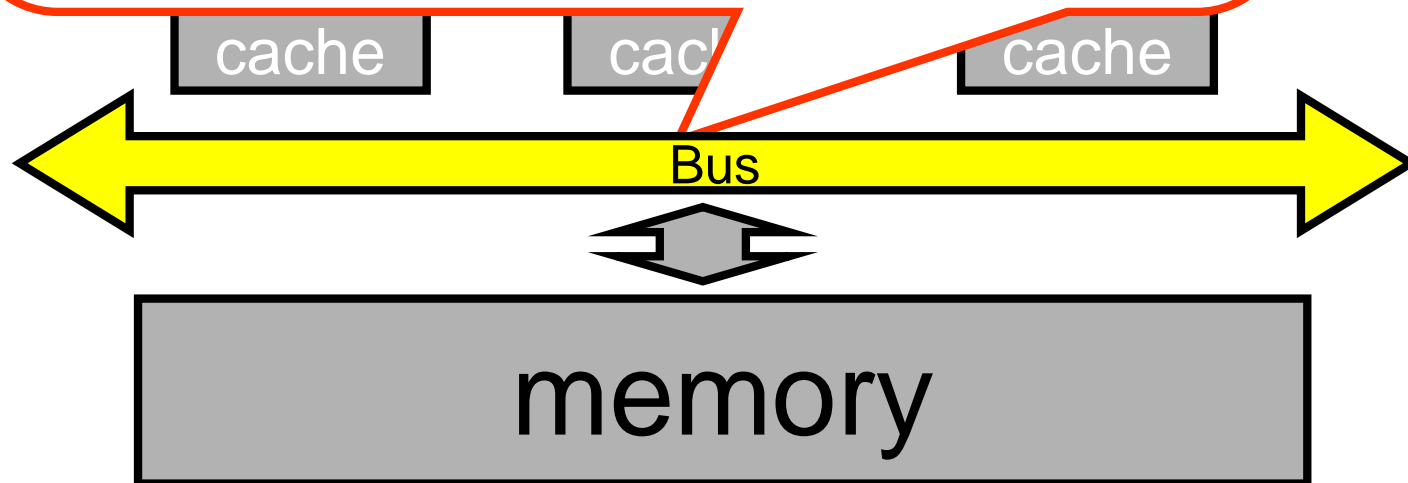
Bus-Based Architectures



Bus-Based Architectures

Shared Bus

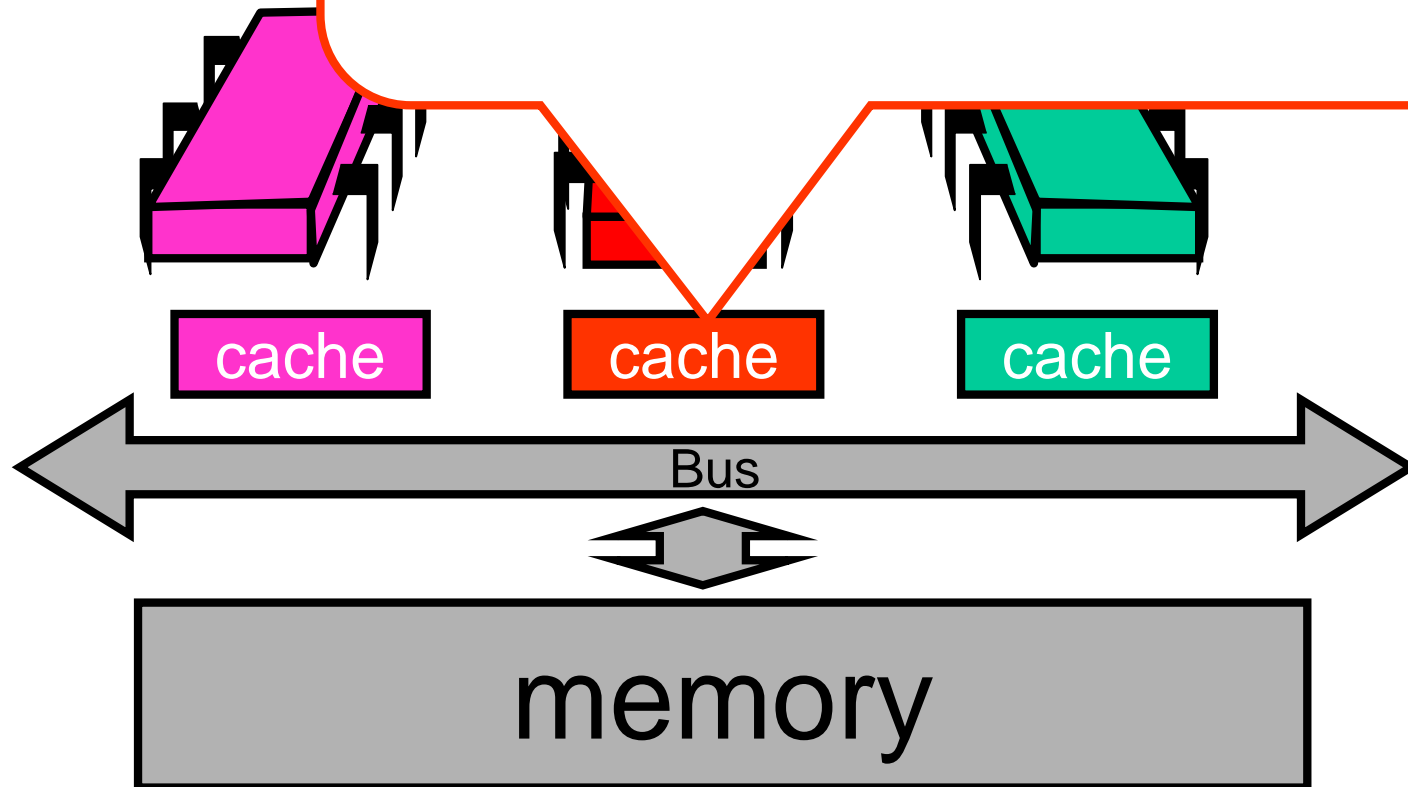
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”



Bus-

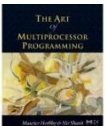
Per-Processor Caches

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



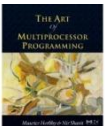
Granularity

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

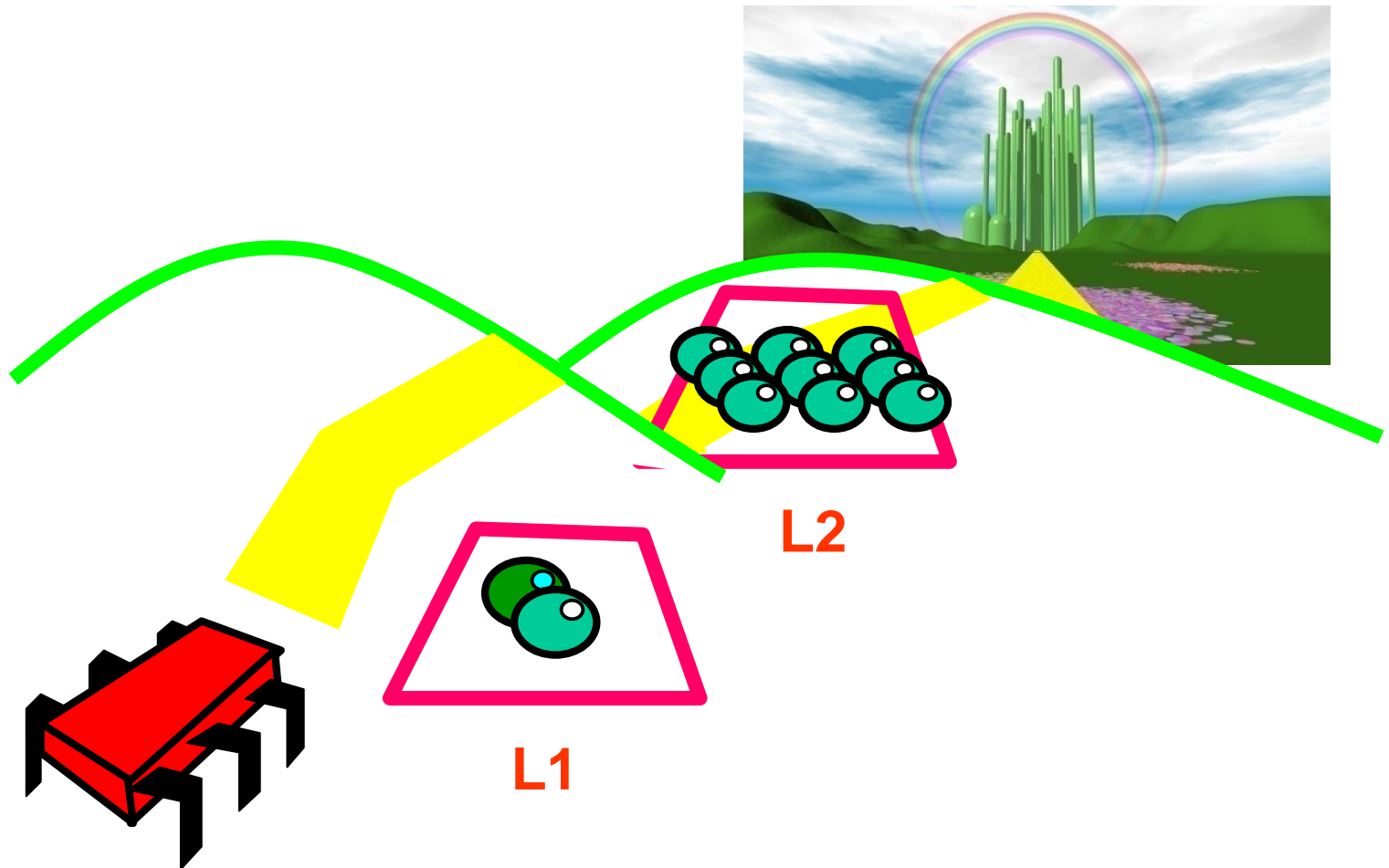


Locality

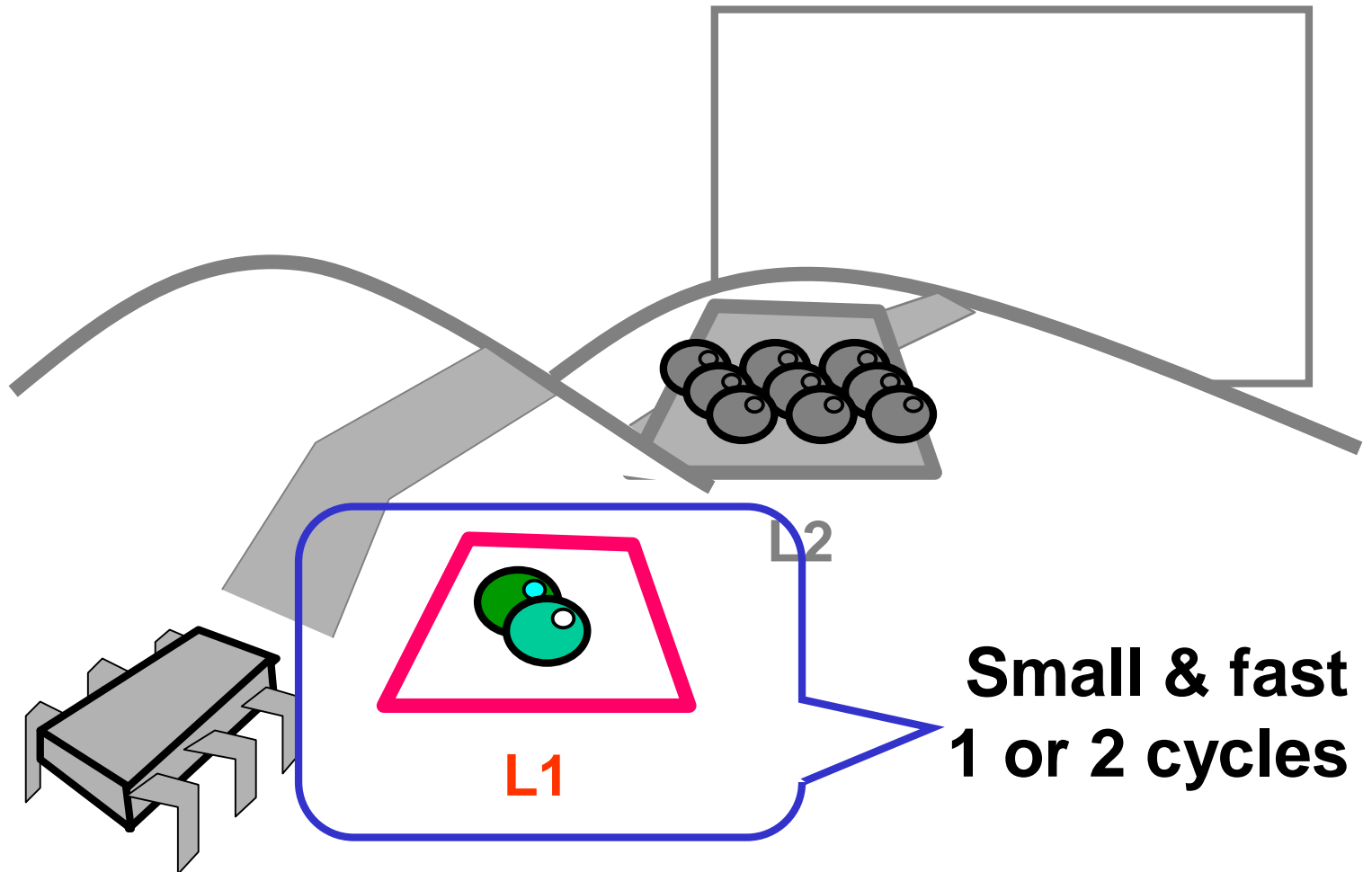
- If you use an address now, you will probably use it again soon
 - Fetch from cache, not memory
- If you use an address now, you will probably use a nearby address soon
 - In the same cache line



L1 and L2 Caches

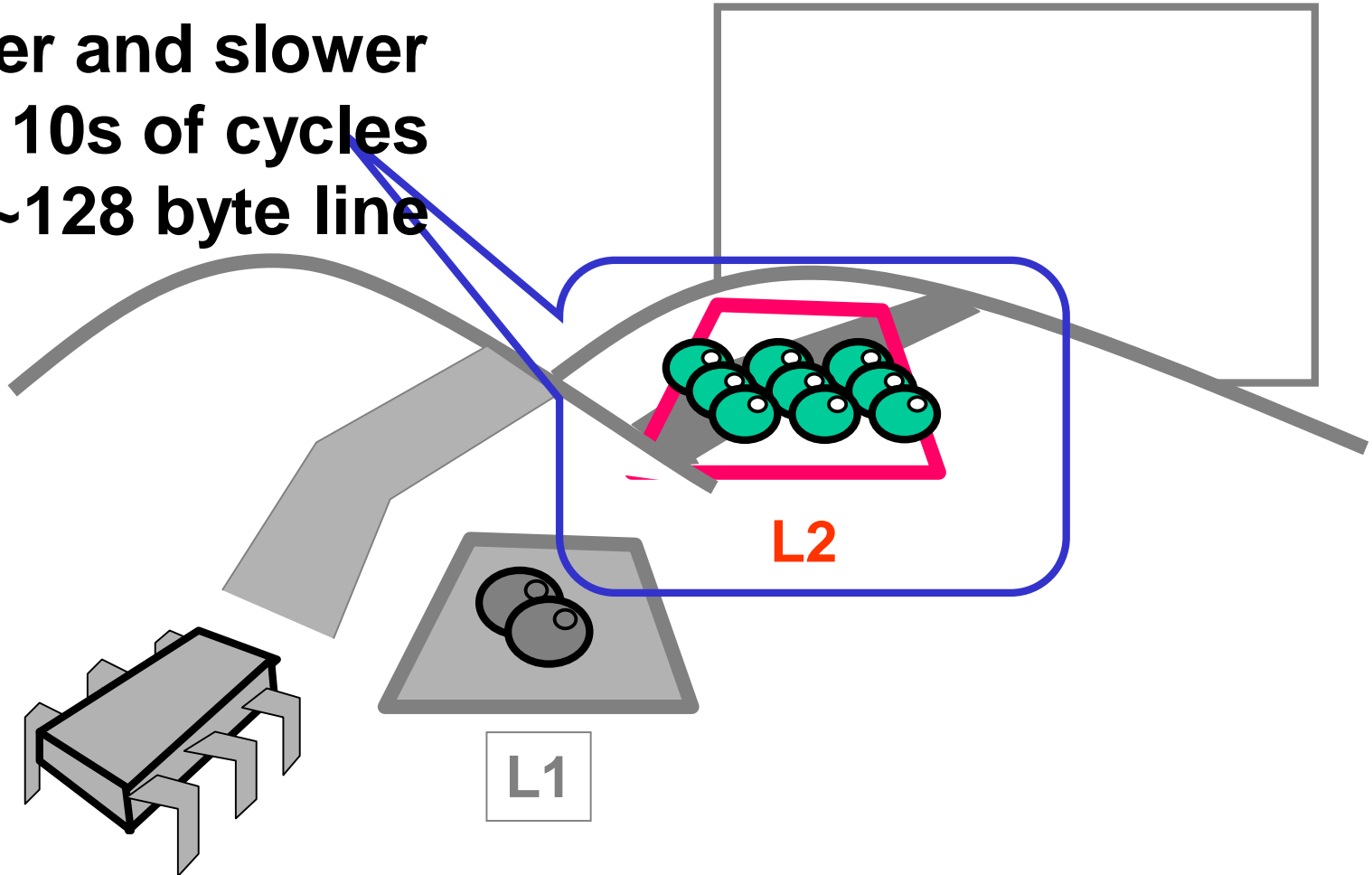


L1 and L2 Caches



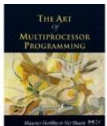
L1 and L2 Caches

Larger and slower
10s of cycles
~128 byte line



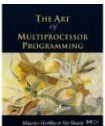
Jargon Watch

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



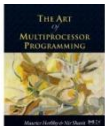
Jargon Watch

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



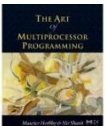
Cave Canem

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



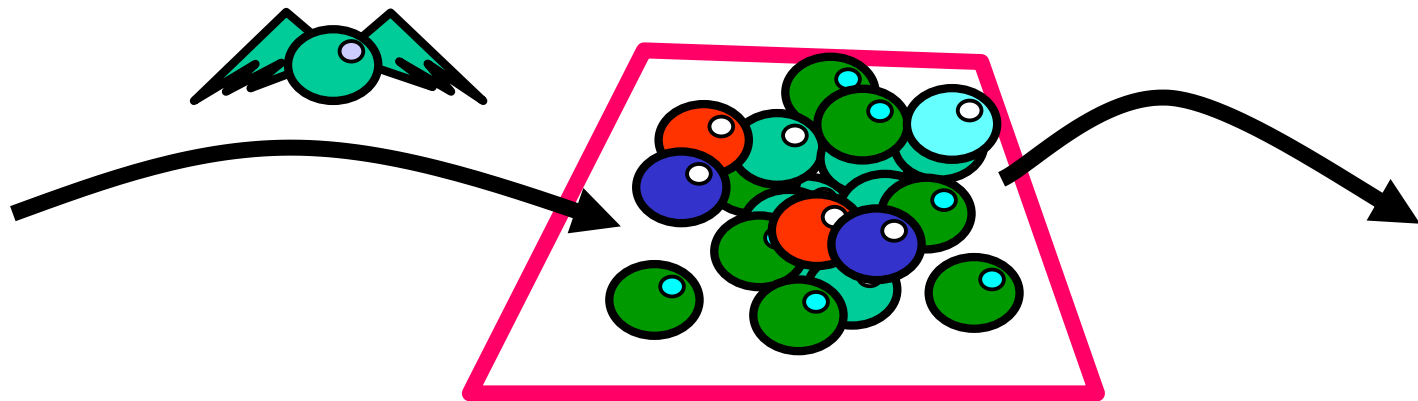
When a Cache Becomes Full...

- Need to make room for new entry
- By evicting an existing entry
- Need a replacement policy
 - Usually some kind of least recently used heuristic



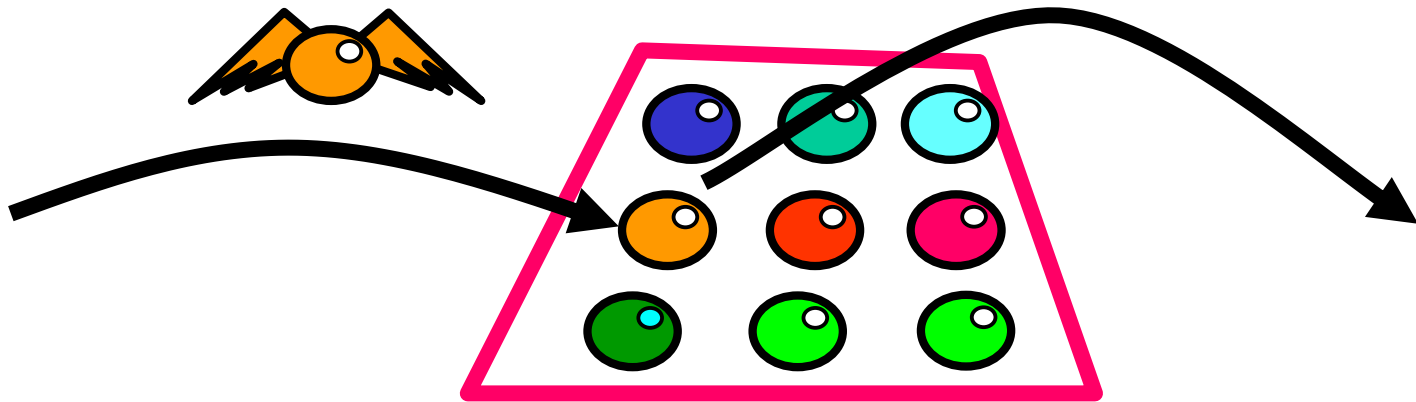
Fully Associative Cache

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



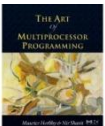
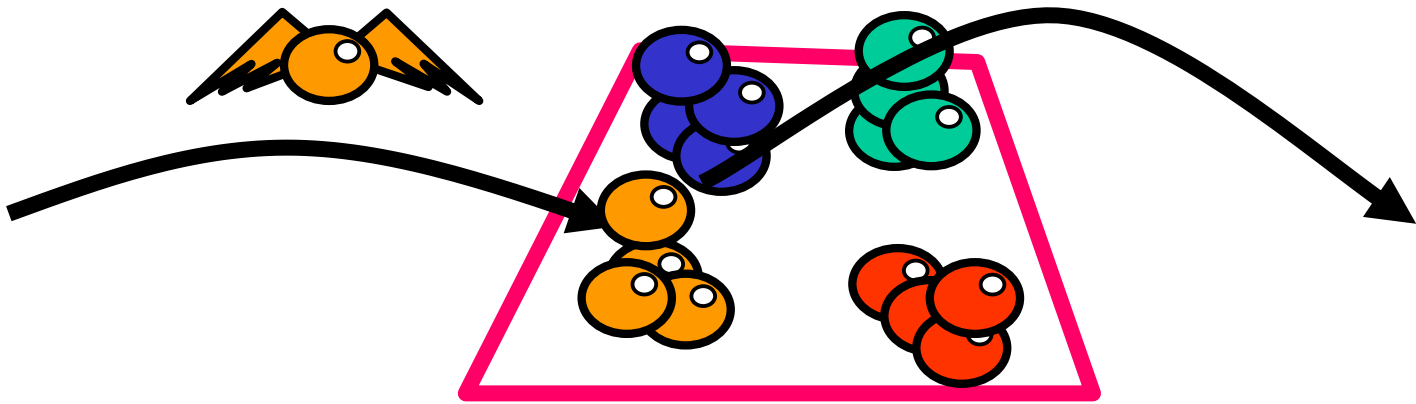
Direct Mapped Cache

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



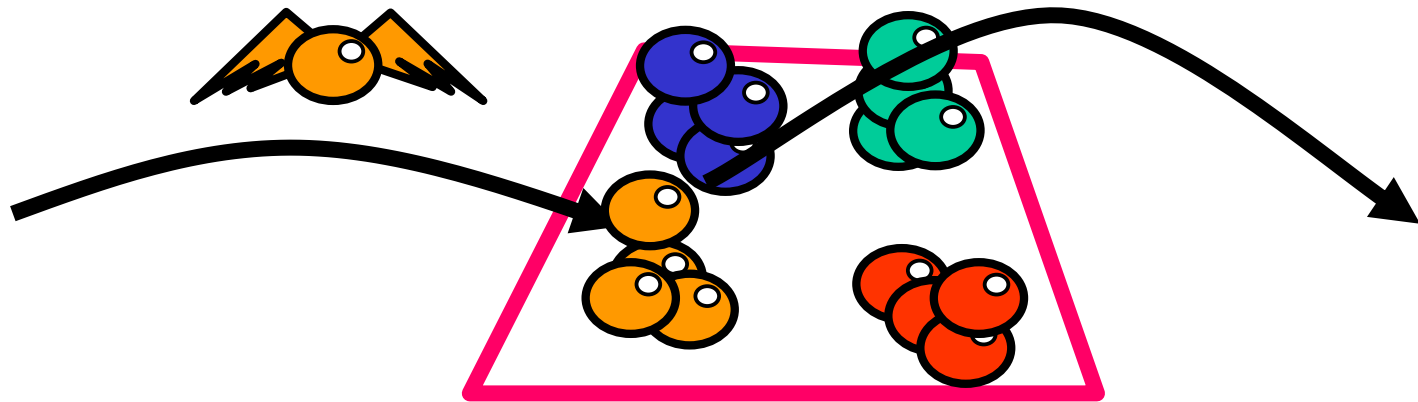
K-way Set Associative Cache

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



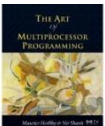
Multicore Set Associativity

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



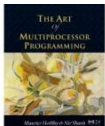
Cache Coherence

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



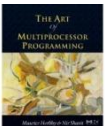
MESI

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



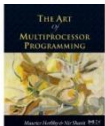
MESI

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



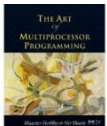
MESI

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

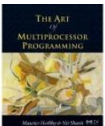
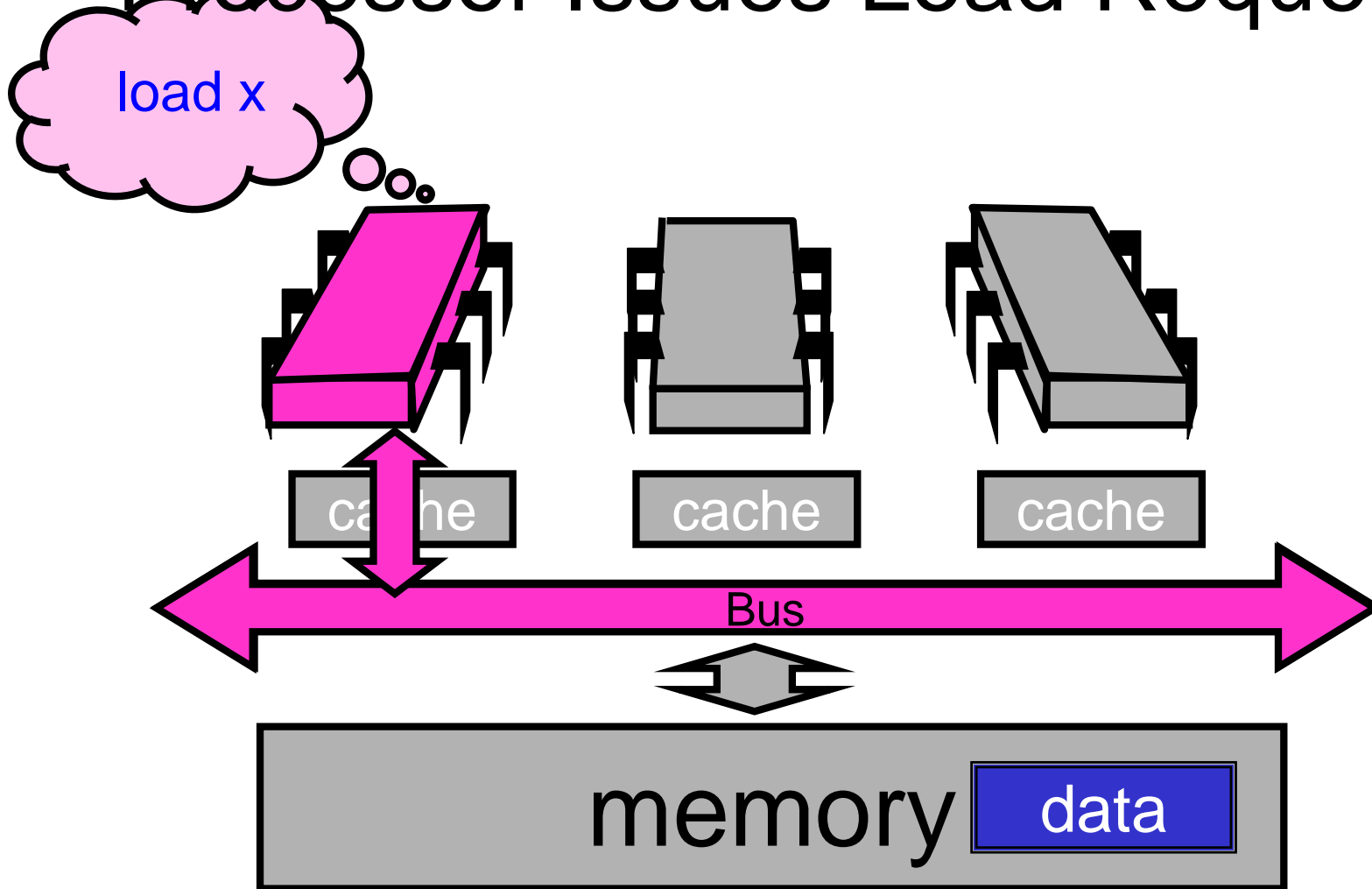


MESI

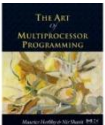
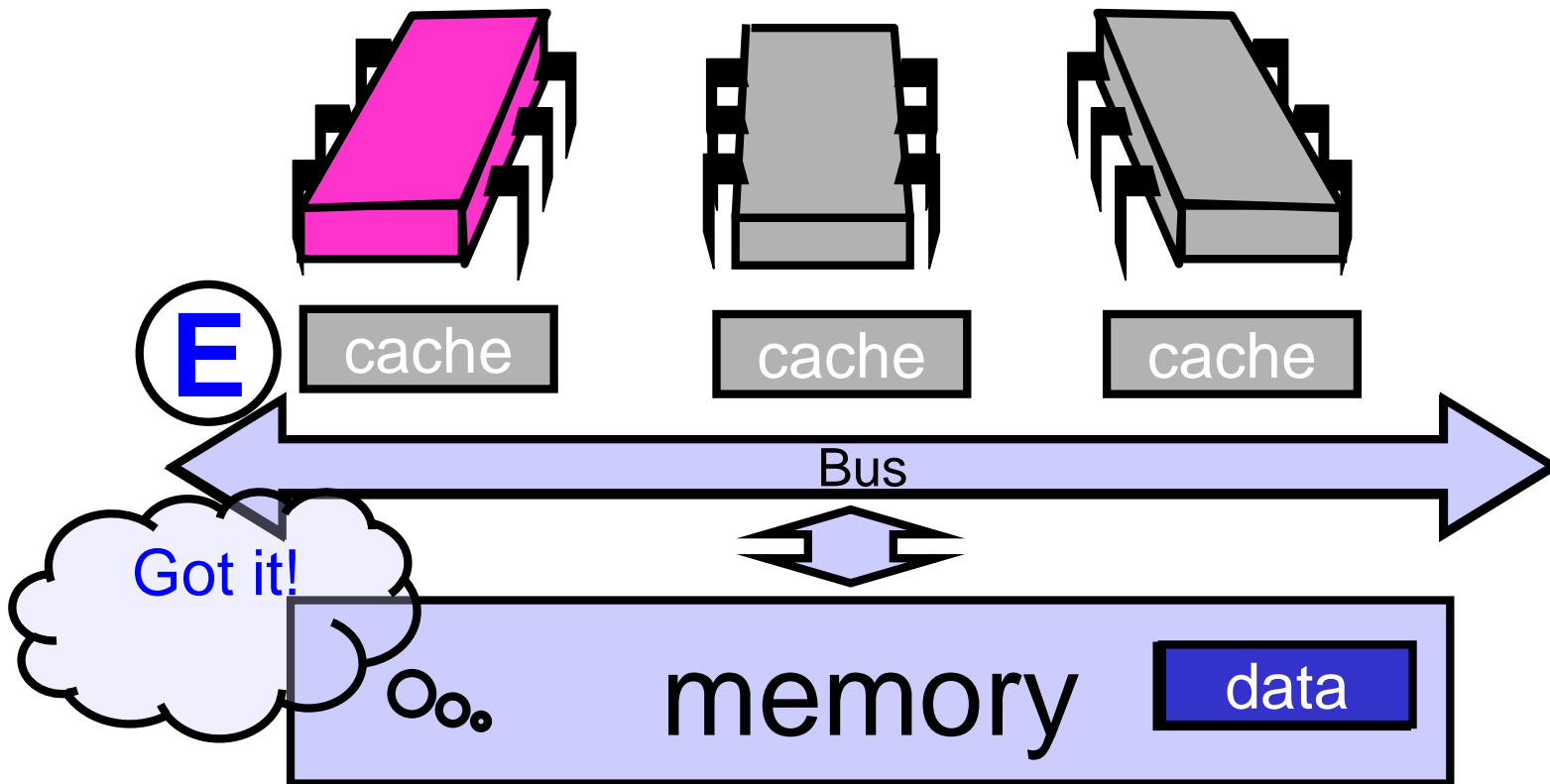
- Modified
 - Have modified cached data, must write back to memory
- Exclusive
 - Not modified, I have only copy
- Shared
 - Not modified, may be cached elsewhere
- Invalid
 - Cache contents not meaningful



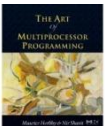
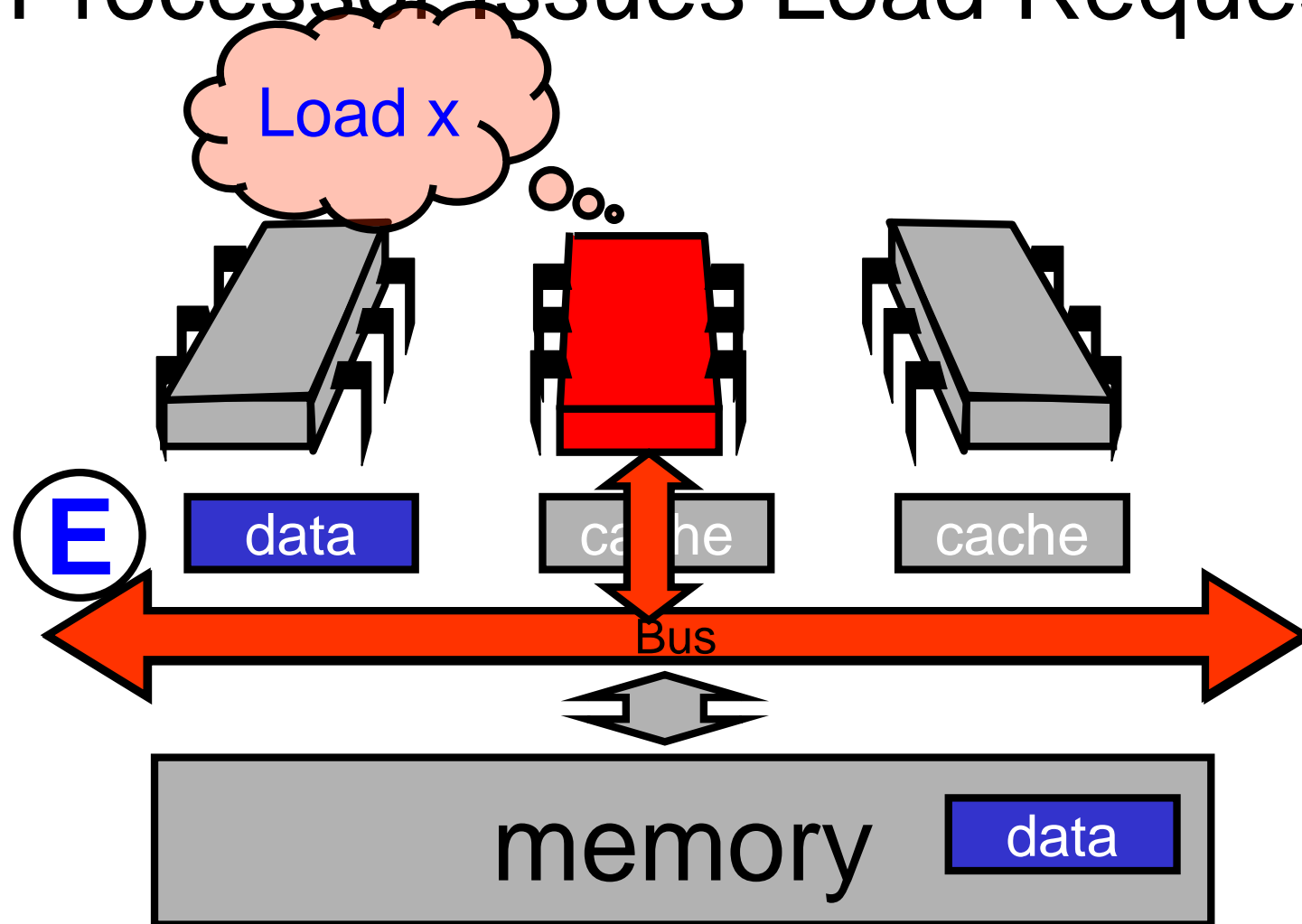
Processor Issues Load Request



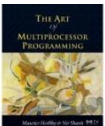
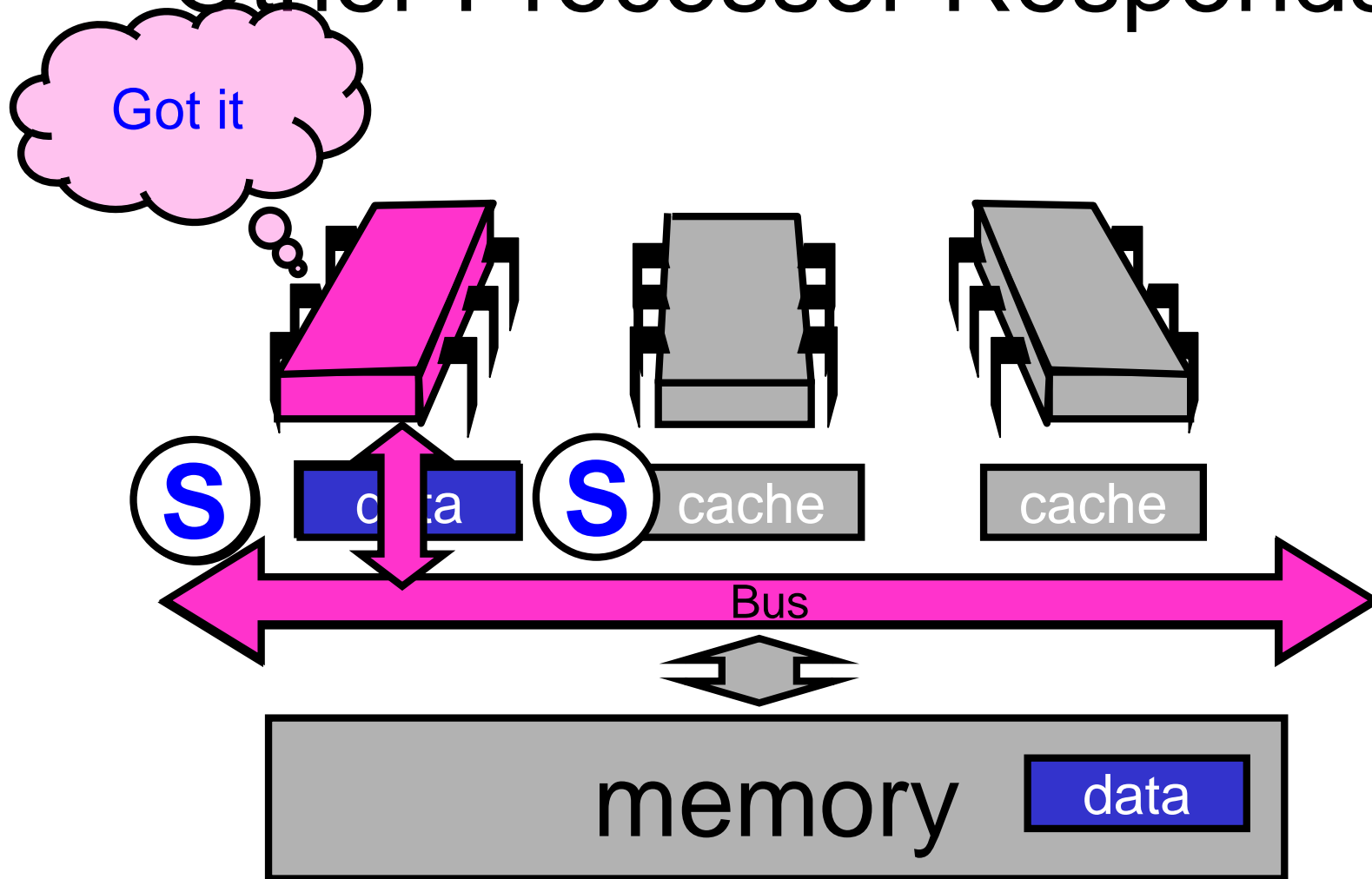
Memory Responds



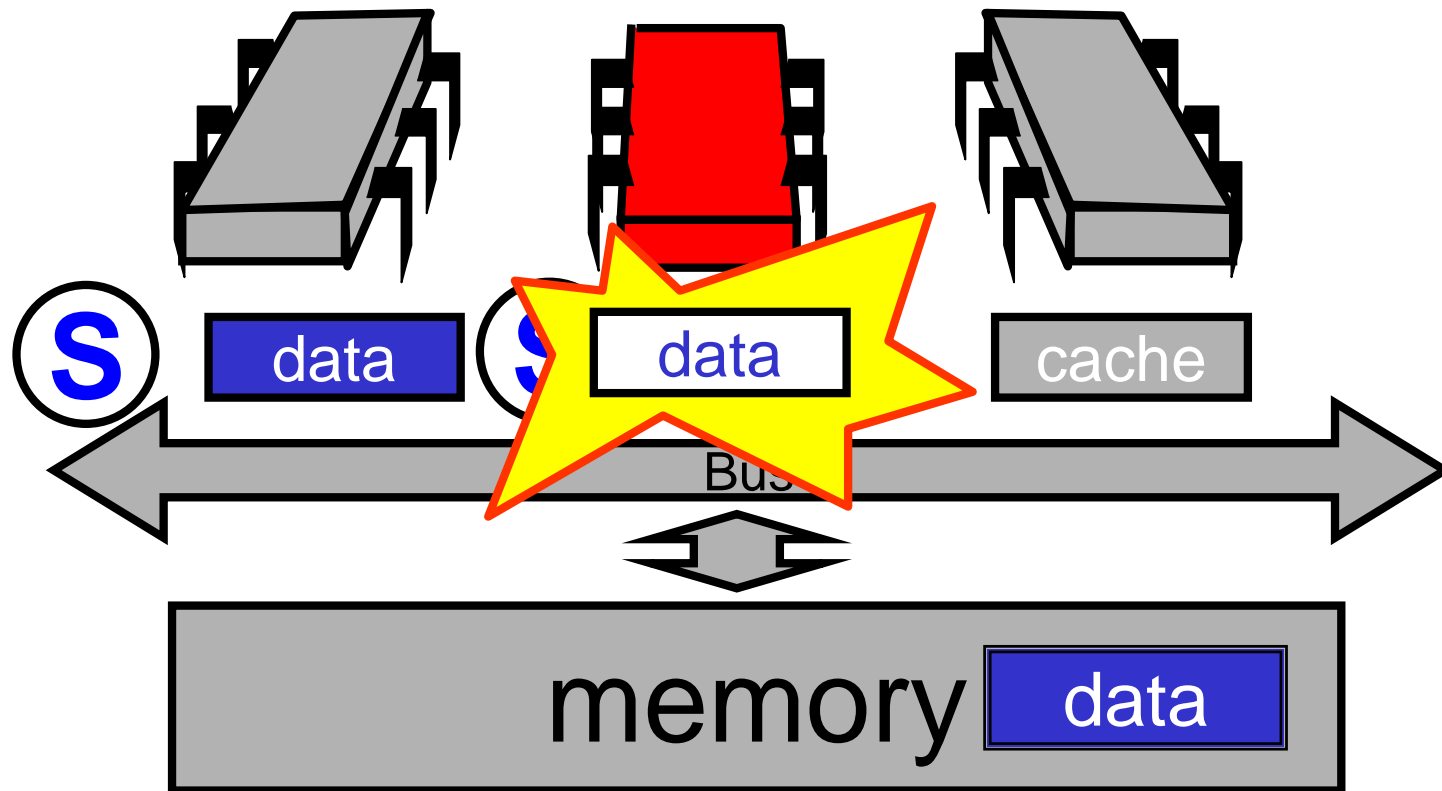
Processor Issues Load Request



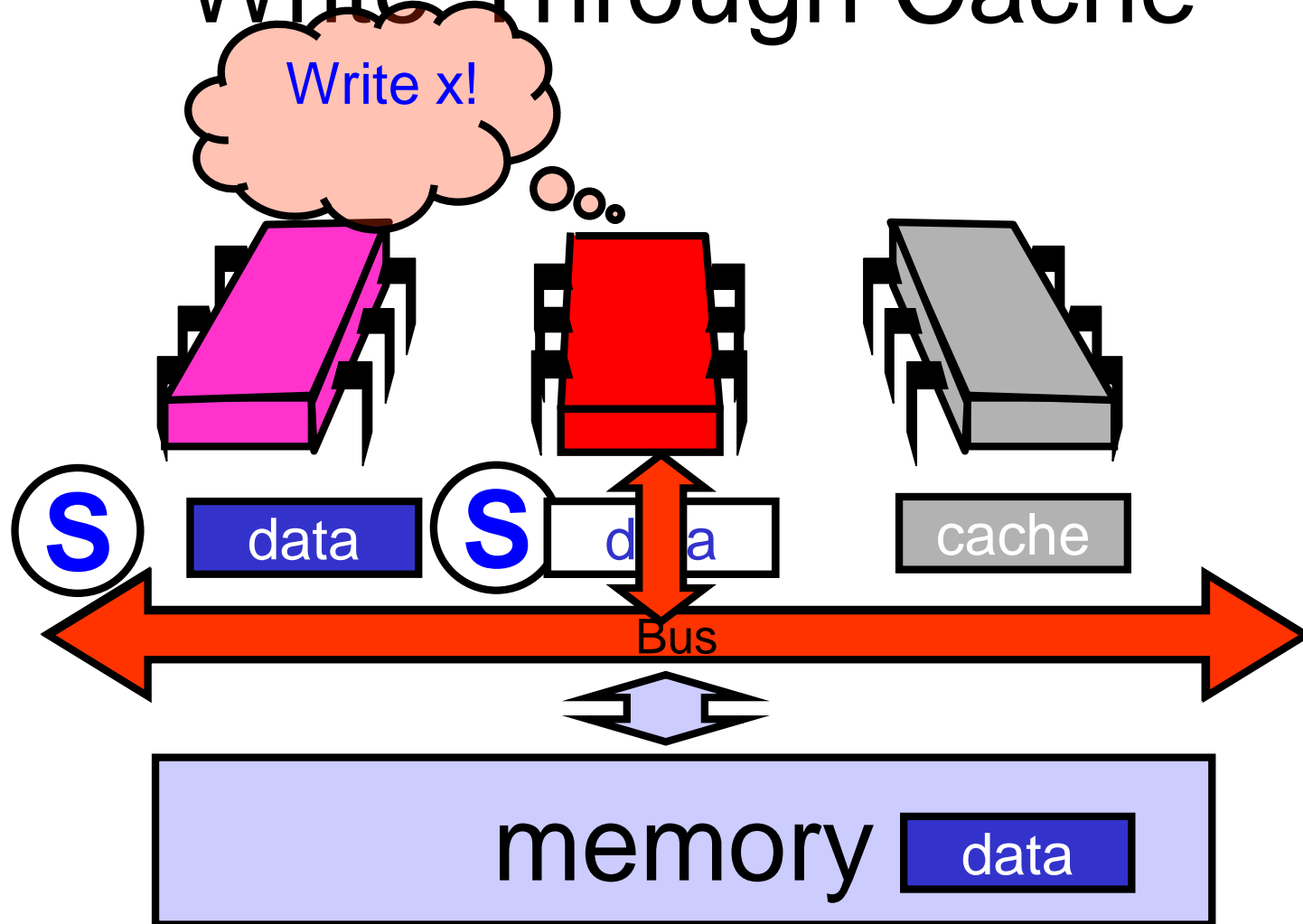
Other Processor Responds



Modify Cached Data

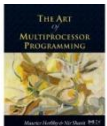


Write-Through Cache



Write-Through Caches

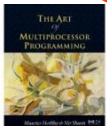
- Immediately broadcast changes
- Good
 - Memory, caches always agree
 - More read hits, maybe
- Bad
 - Bus traffic on all writes
 - Most writes to unshared data
 - For example, loop indexes ...



Write-Through Caches

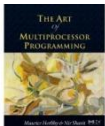
- Immediately broadcast changes
- Good
 - Memory, caches always agree
 - More read hits, maybe
- Bad
 - Bus traffic on all writes
 - Most writes to unshared data
 - For example, loop indexes ...

“show stoppers”

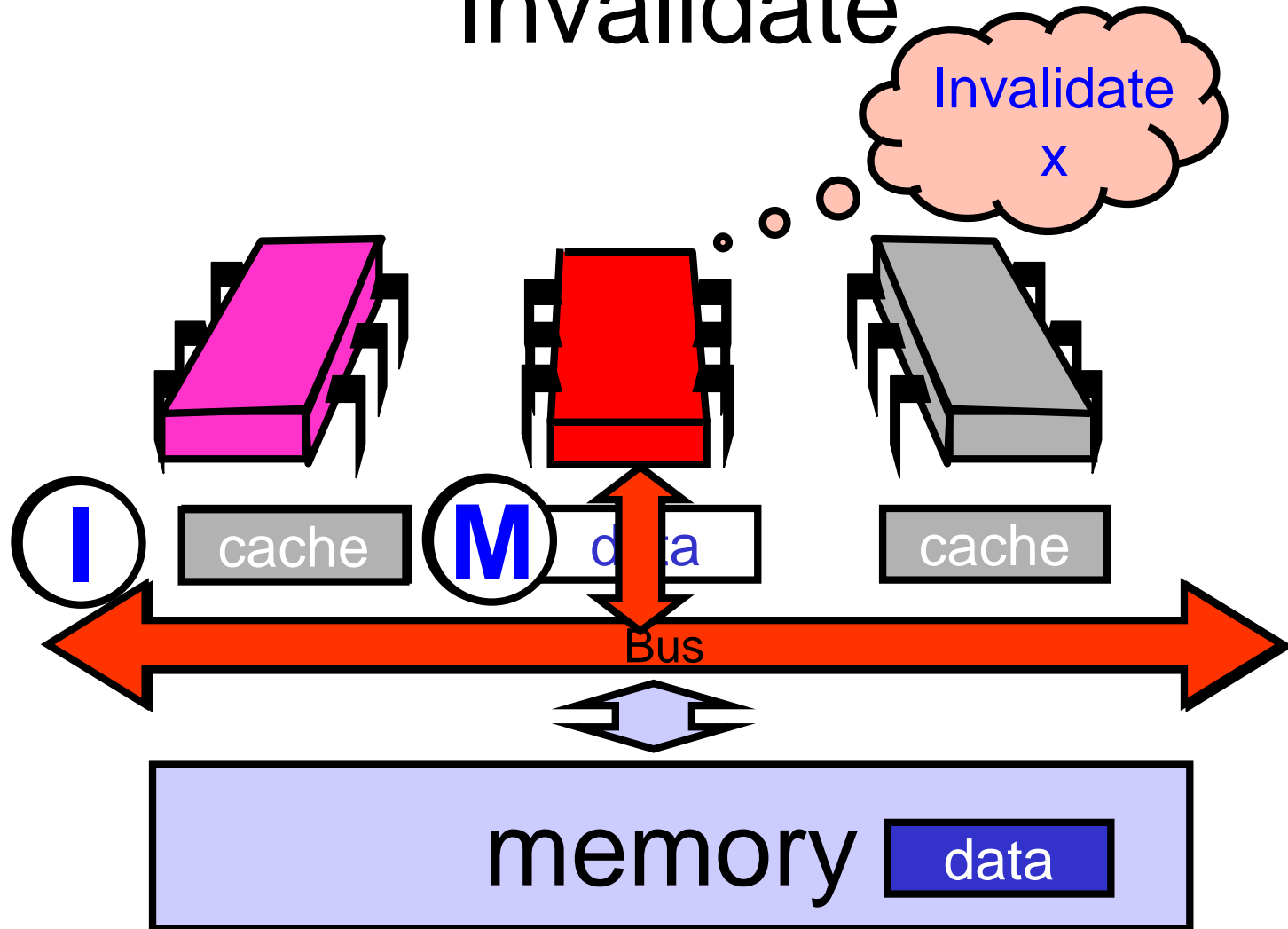


Write-Back Caches

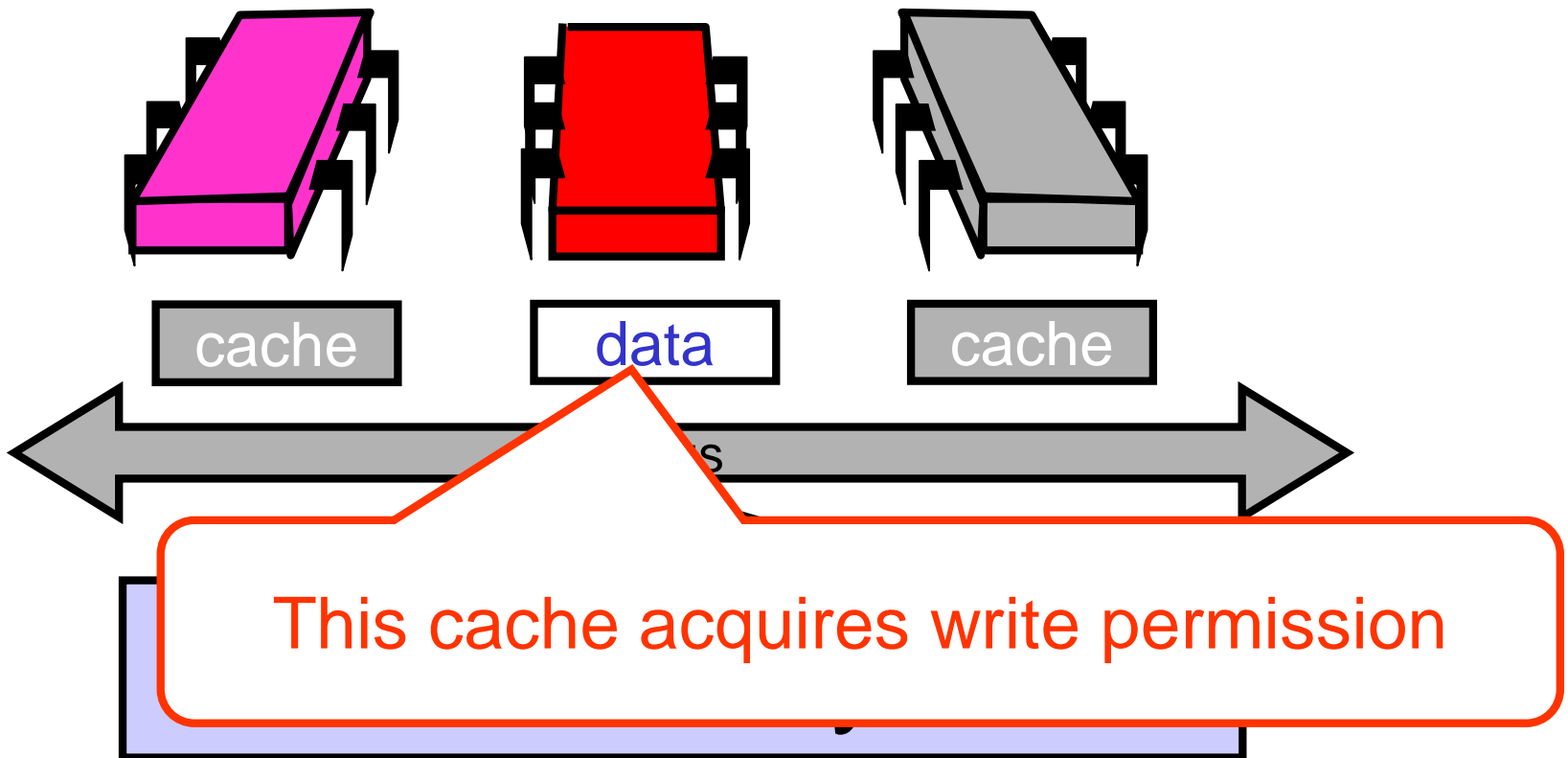
- Accumulate changes in cache
- Write back when line evicted
 - Need the cache for something else
 - Another processor wants it



Invalidate

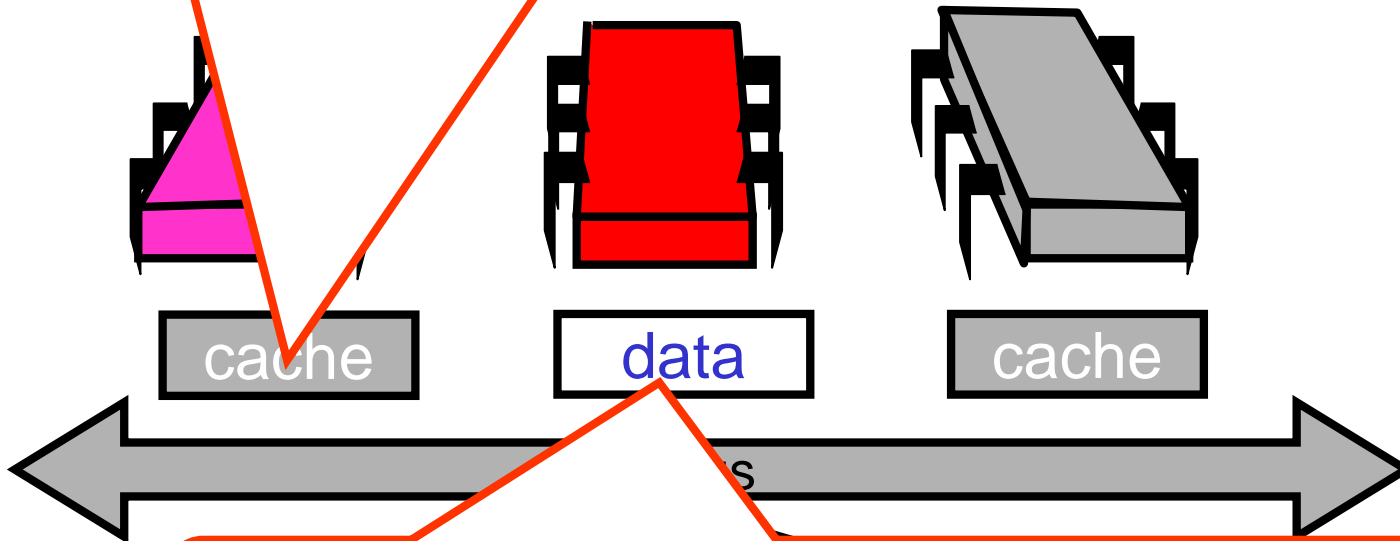


Invalidate



Invalidate

Other caches lose read permission



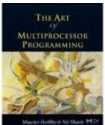
This cache acquires write permission

Invalidate



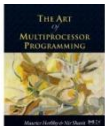
Memory provides data only if not present
in any cache, so no need to change it
now (expensive)

memory data



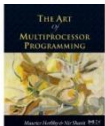
Mutual Exclusion

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



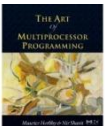
Simple TASLock

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



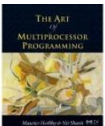
Test-and-Test-and-Set Locks

- 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

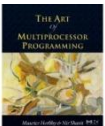
```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
        }  
    }  
}
```



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



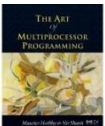
Test-and-test-and-set Lock

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

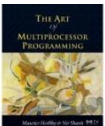
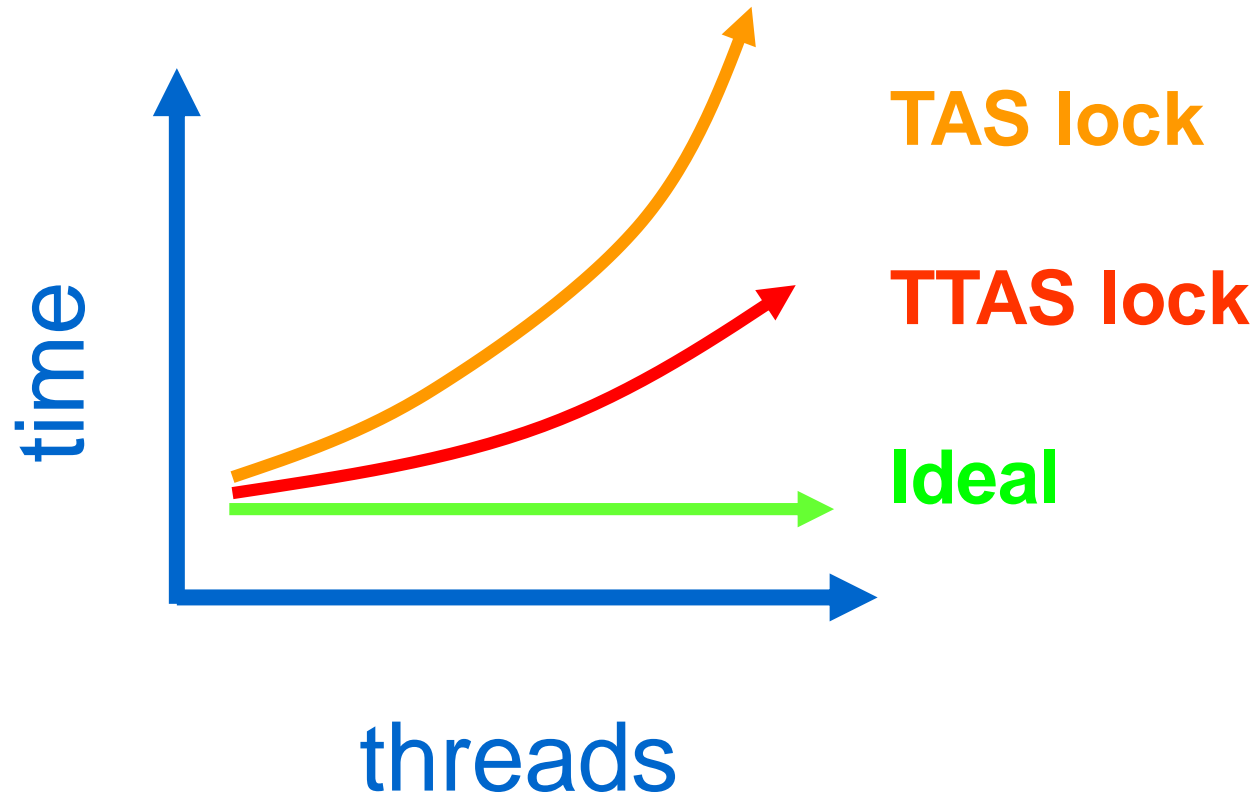
```
    void lock() {  
        while (true) {  
            while (state.get()) {}
```

```
            if (!state.getAndSet(true))  
                return;  
        }  
    }
```

Then try to
acquire it

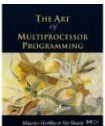


Mystery #2

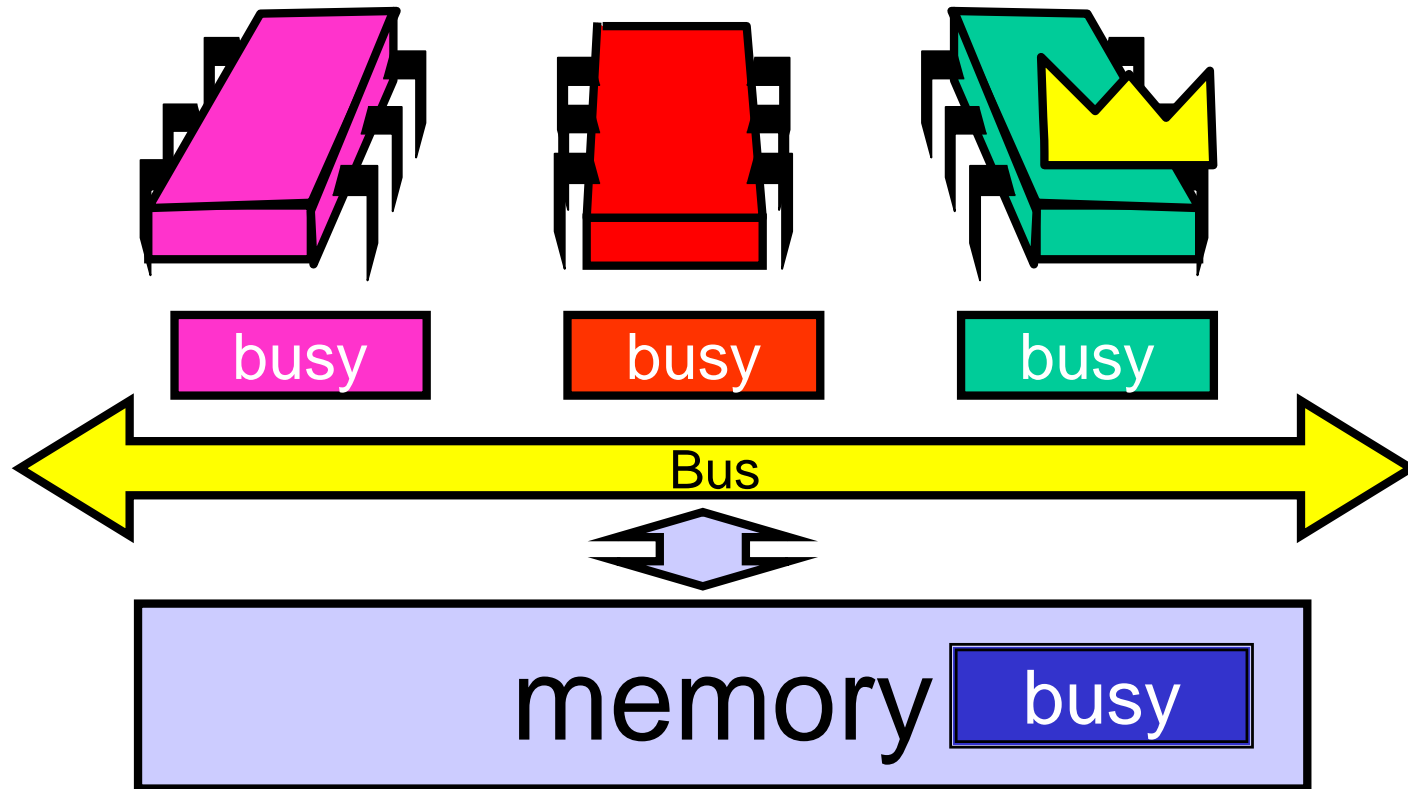


Test-and-test-and-set

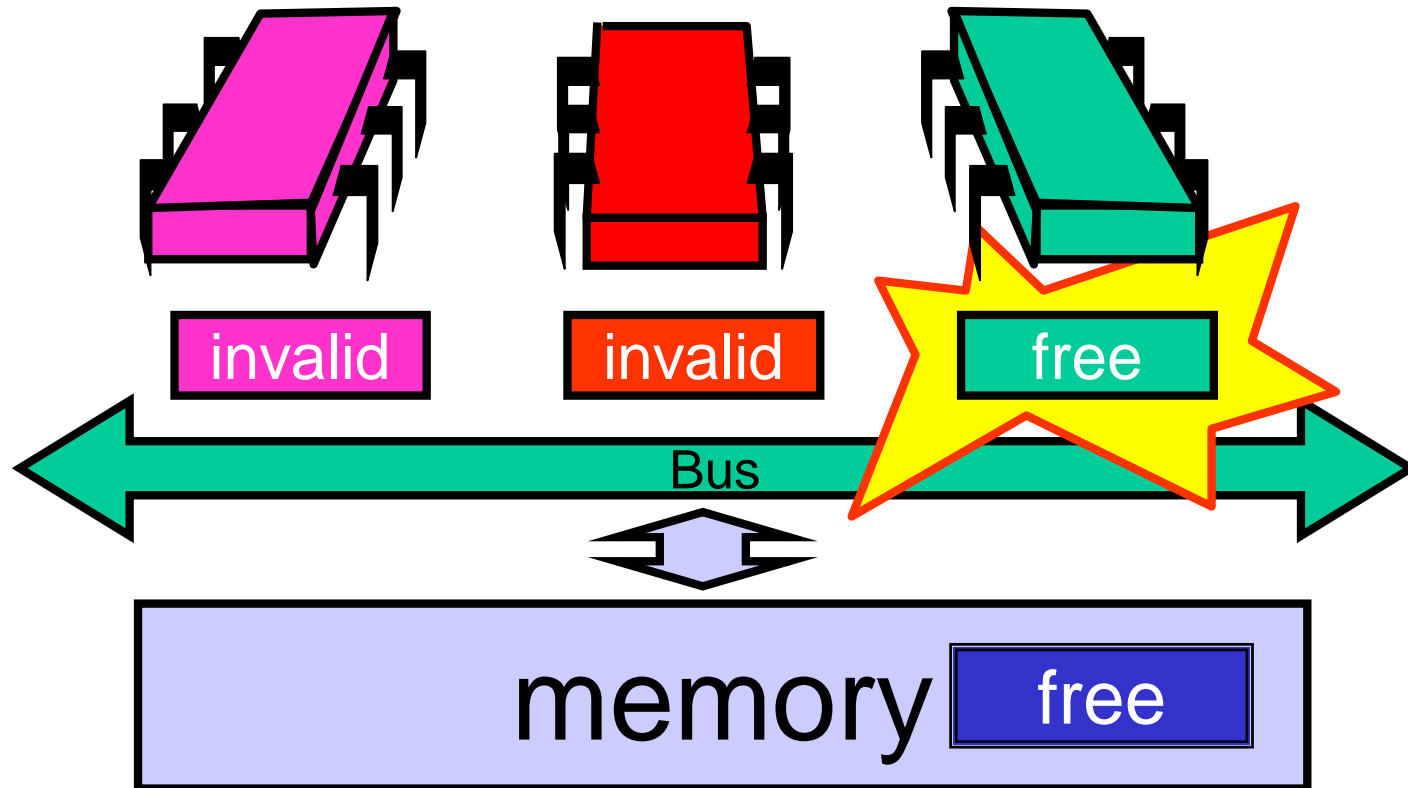
- Wait until lock “looks” free
 - Spin on local cache
 - No bus use while lock busy
- Problem: when lock is released
 - Invalidation storm ...



Local Spinning while Lock is Busy

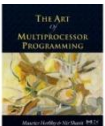
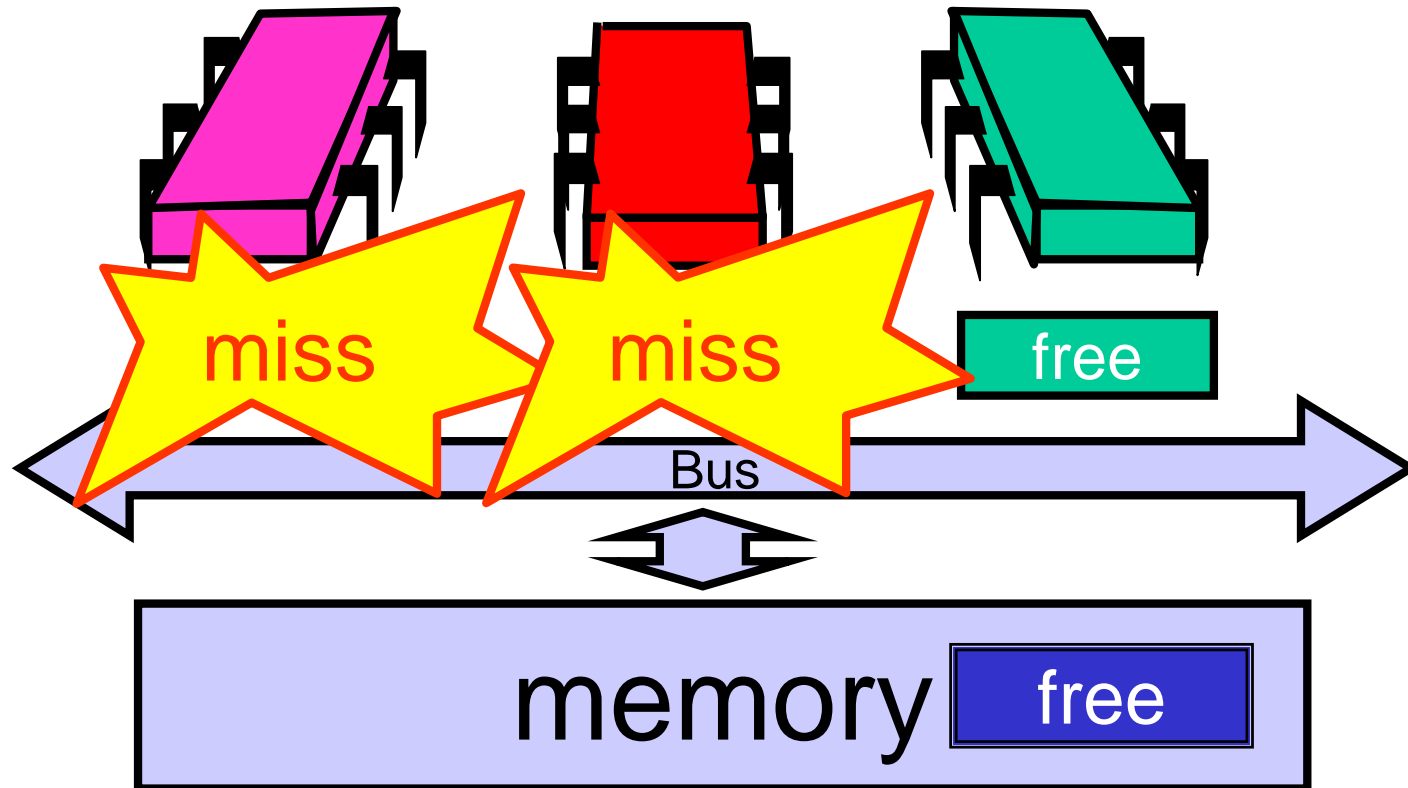


On Release



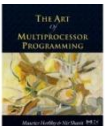
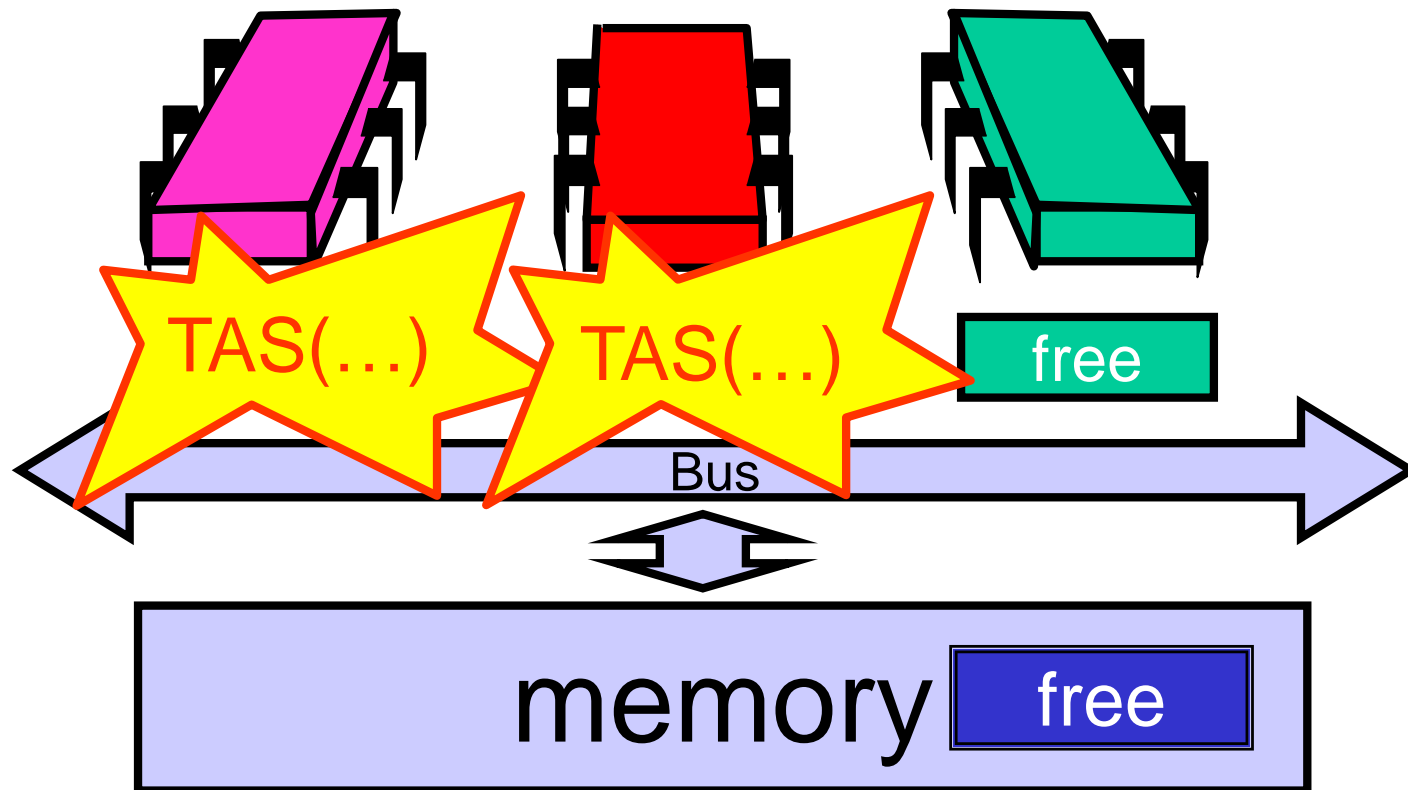
On Release

Everyone misses,
rereads



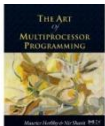
On Release

Everyone tries TAS

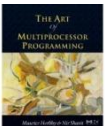
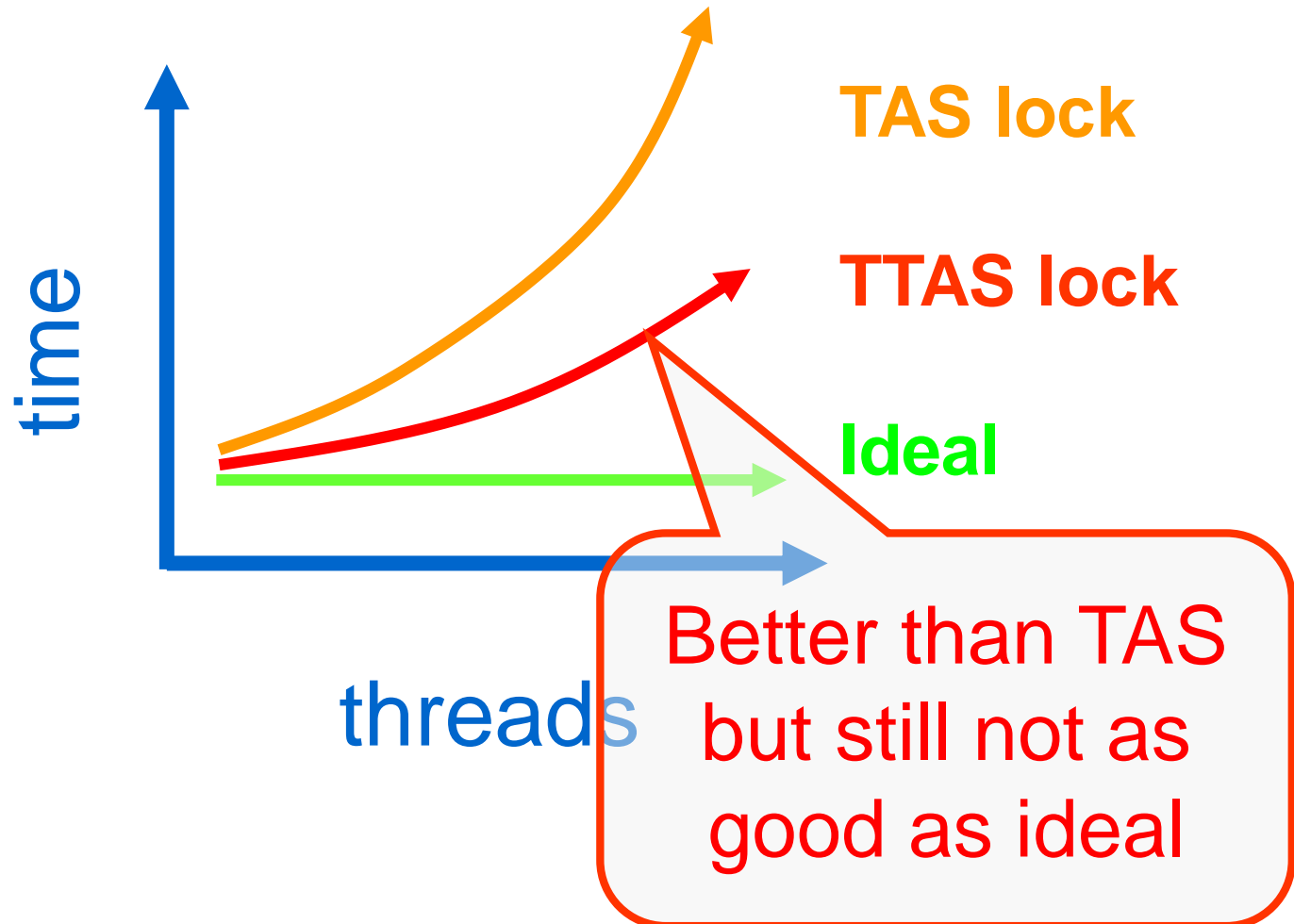


Problems

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

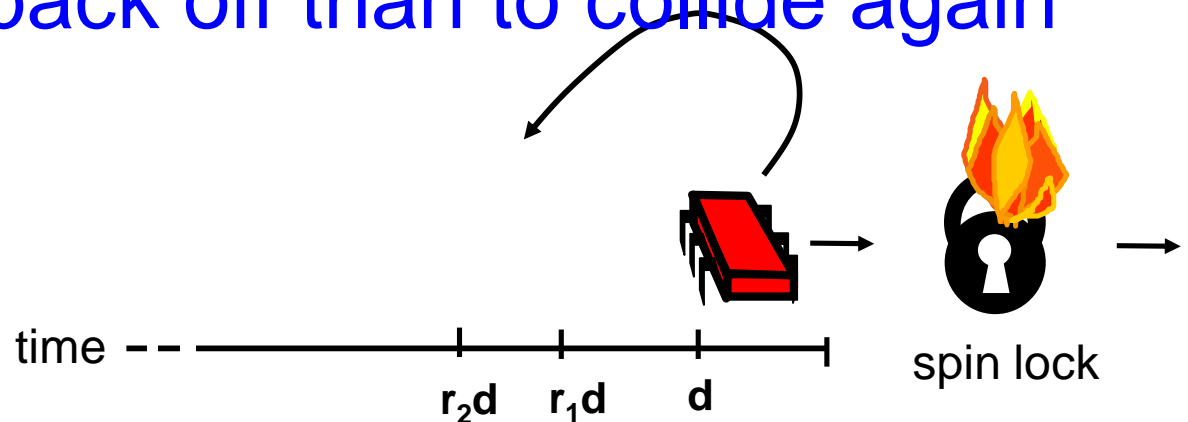


Mystery Explained

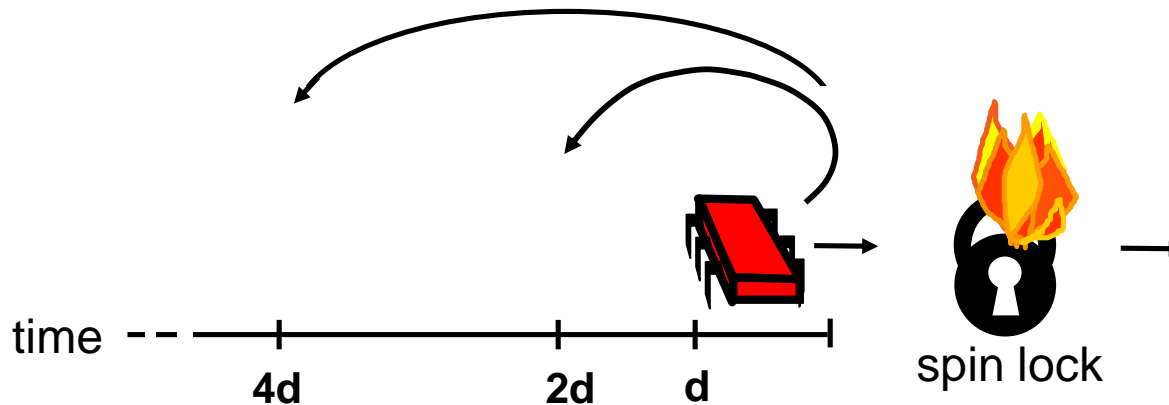


Solution: Introduce Delay

- 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

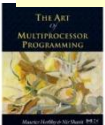


If I fail to get lock

- Wait random duration before retry
- Each subsequent failure doubles expected wait

Exponential Backoff Lock

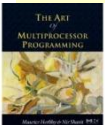
```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)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

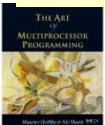
Fix minimum delay



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;  
        }  
    }  
}
```

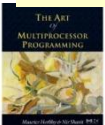
Wait until lock looks free



Exponential Backoff Lock

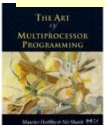
```
public class Backoff implements lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)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

If we win, return



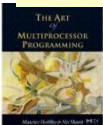
Exponential Backoff Lock

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

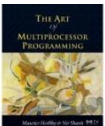
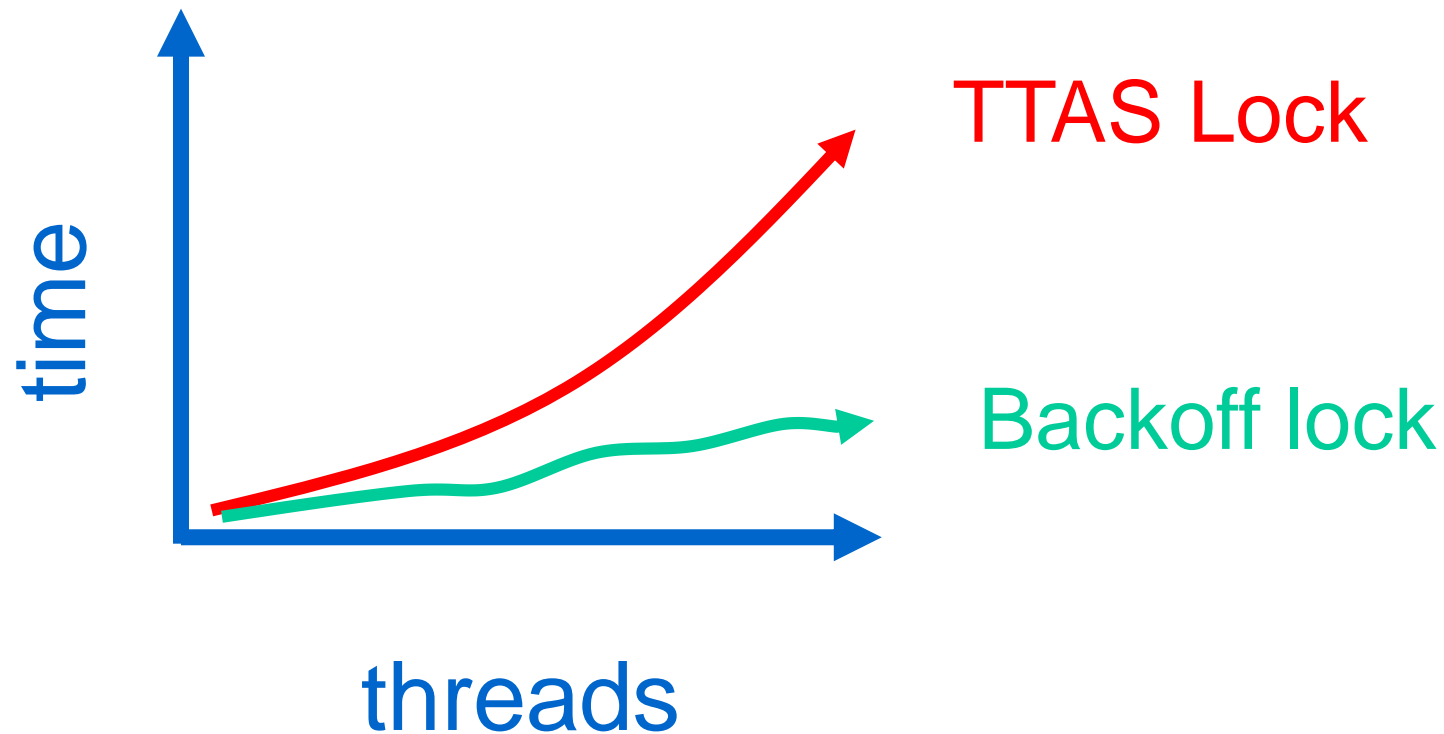


Exponential Backoff Lock

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

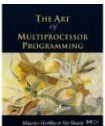


Spin-Waiting Overhead



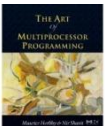
Backoff: Other Issues

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

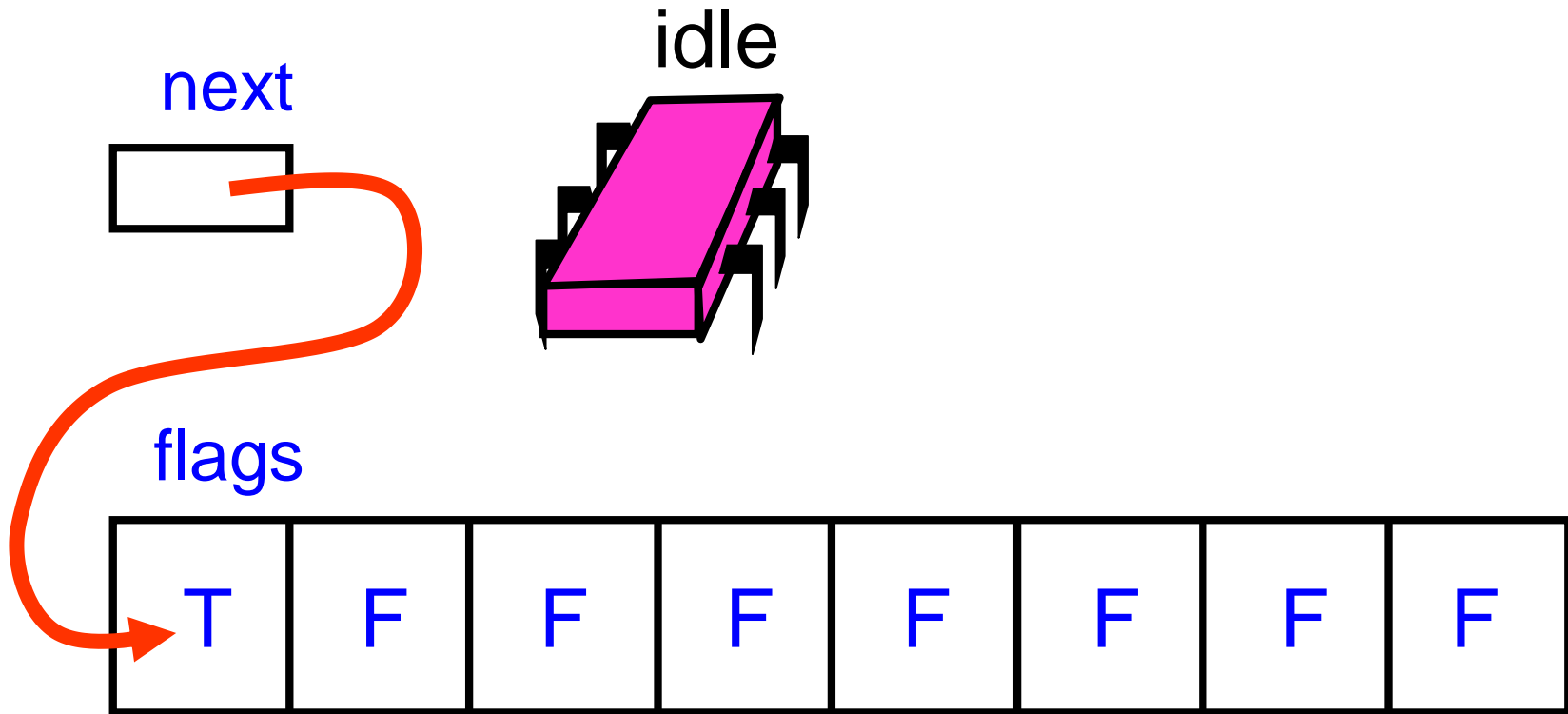


Idea

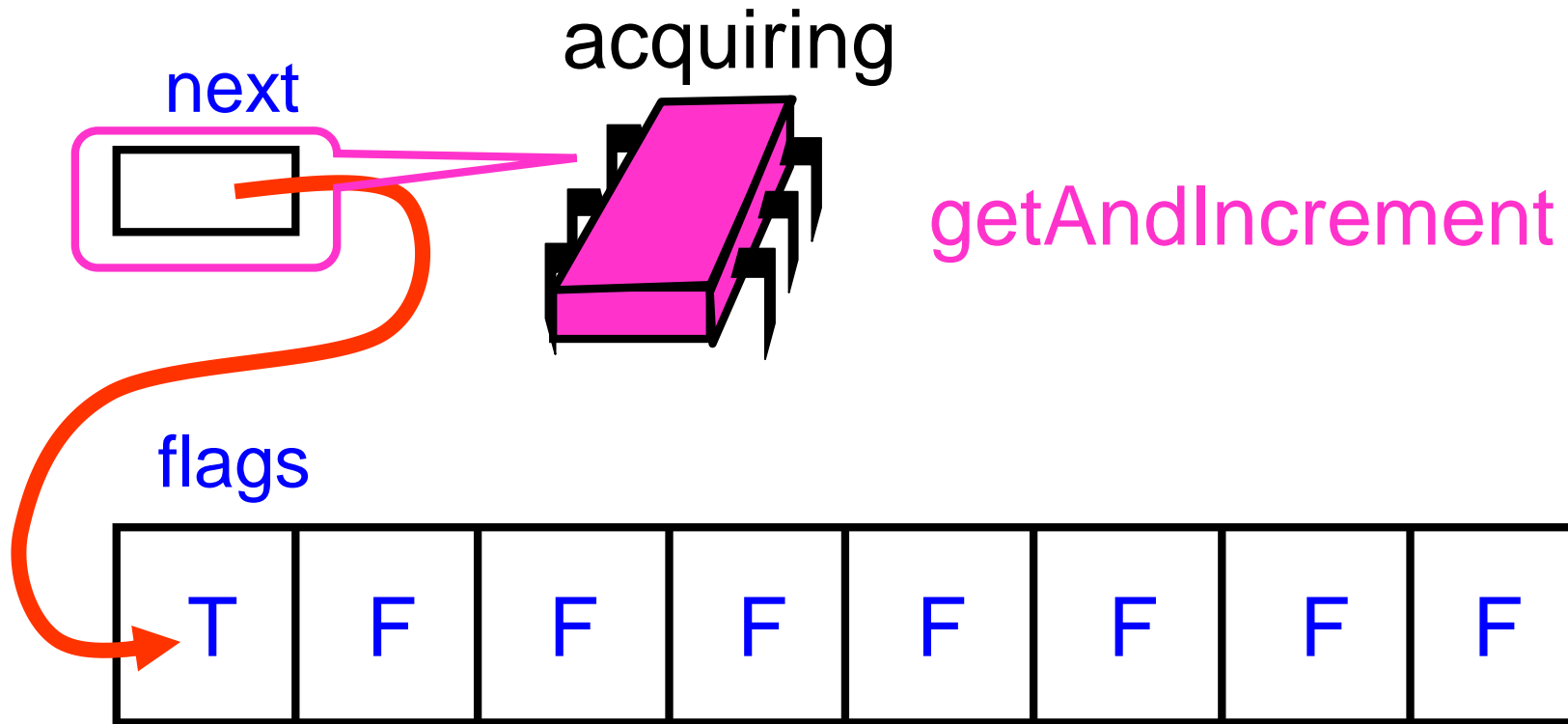
- Avoid useless invalidations
 - By keeping a queue of threads
- Each thread
 - Notifies next in line
 - Without bothering the others



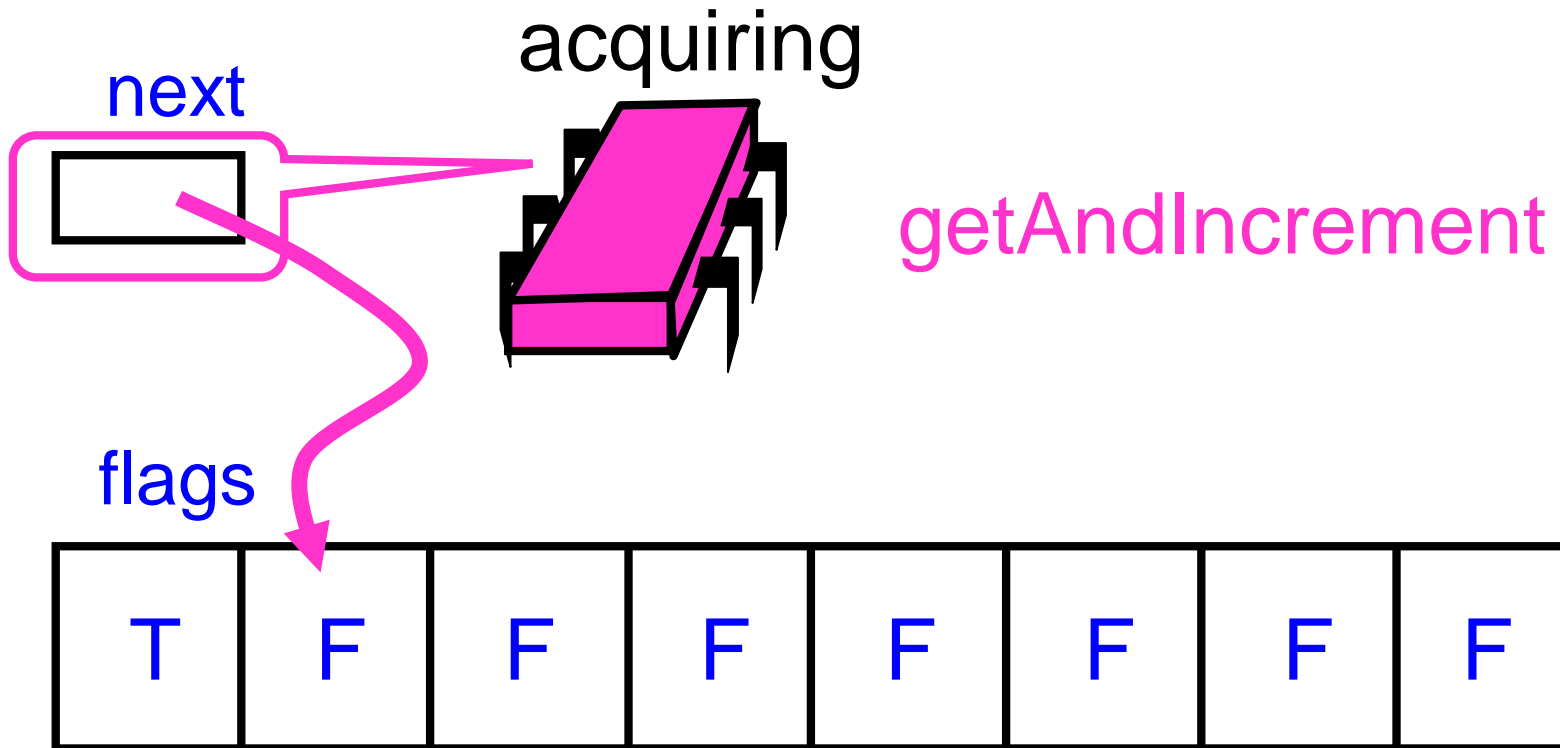
Anderson Queue Lock



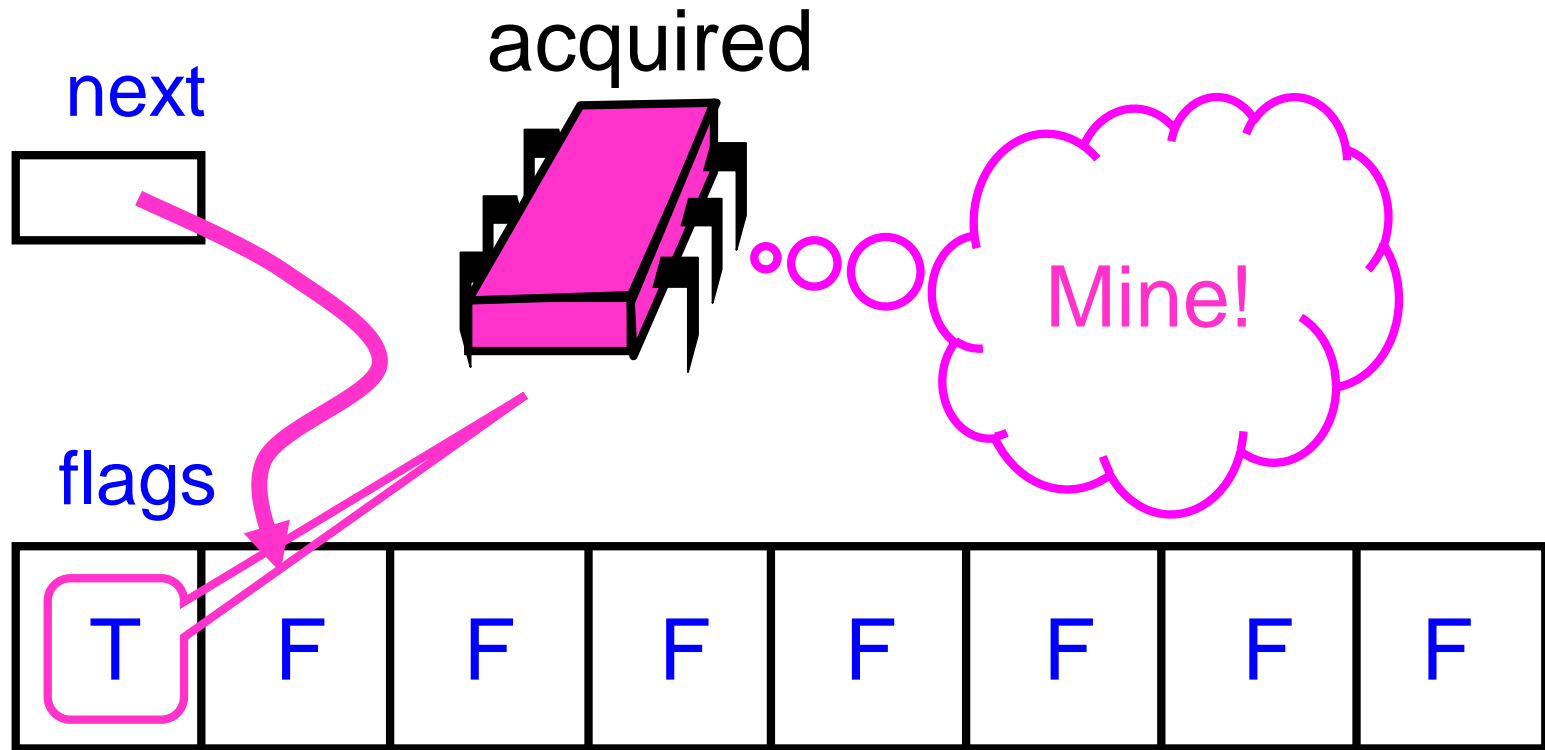
Anderson Queue Lock



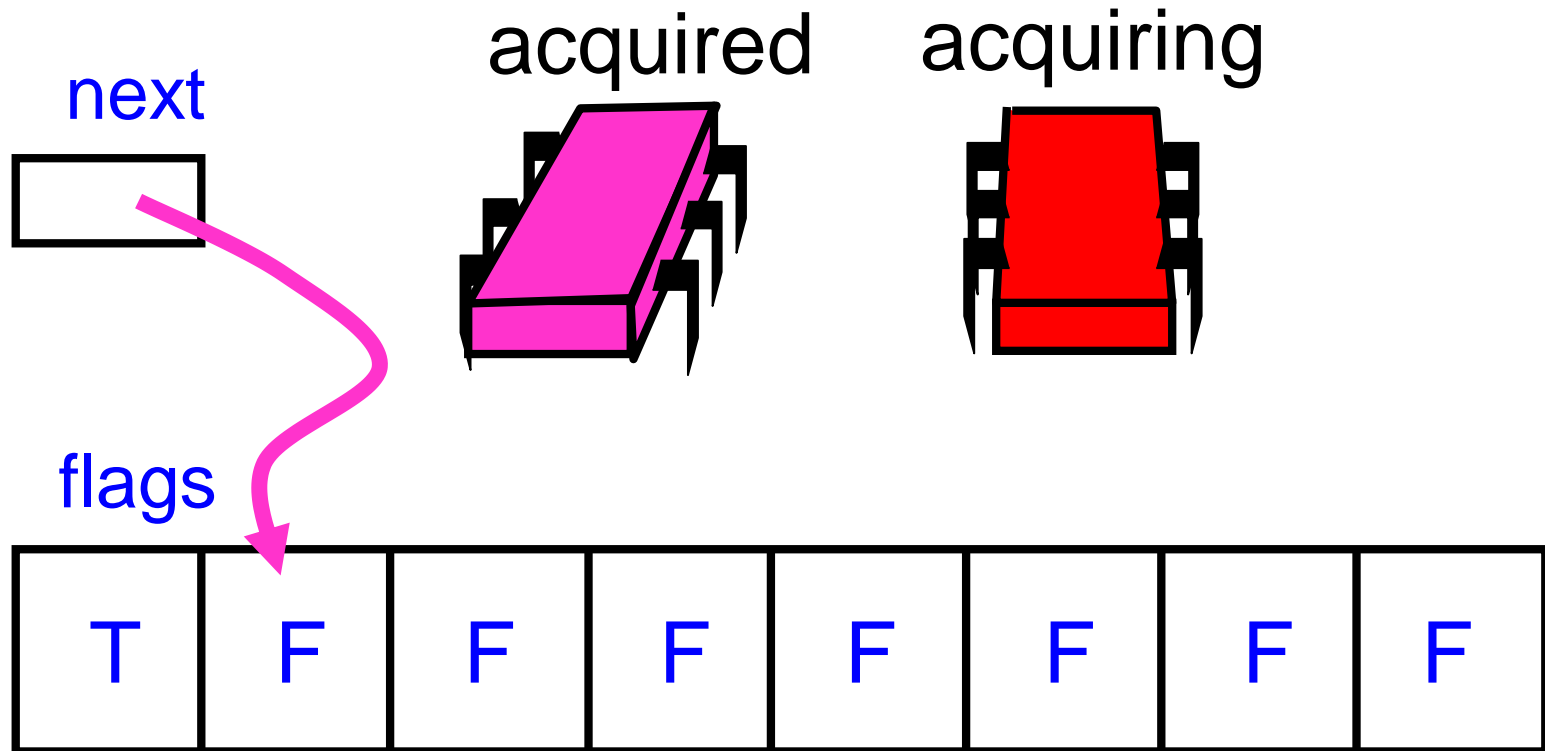
Anderson Queue Lock



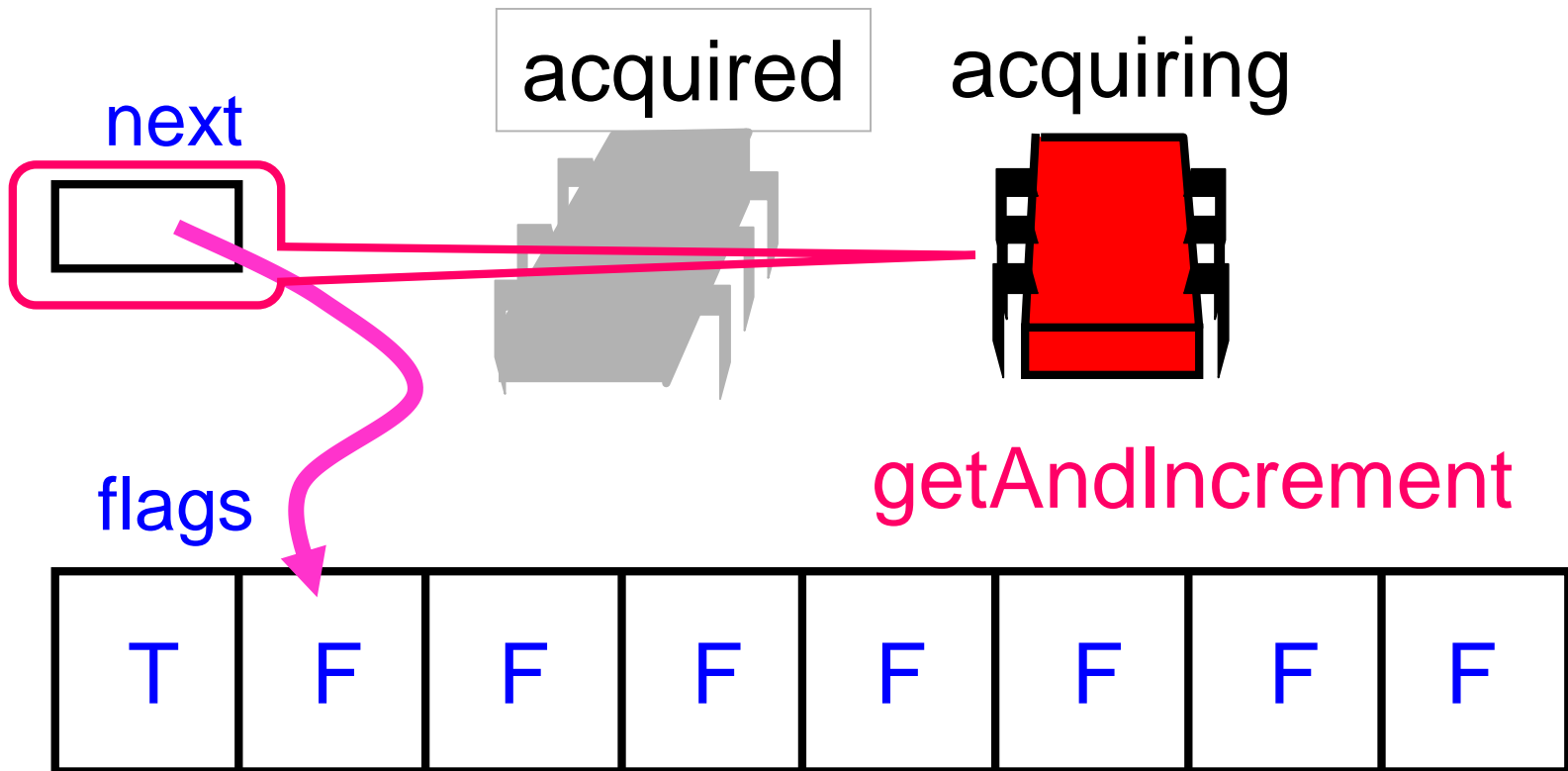
Anderson Queue Lock



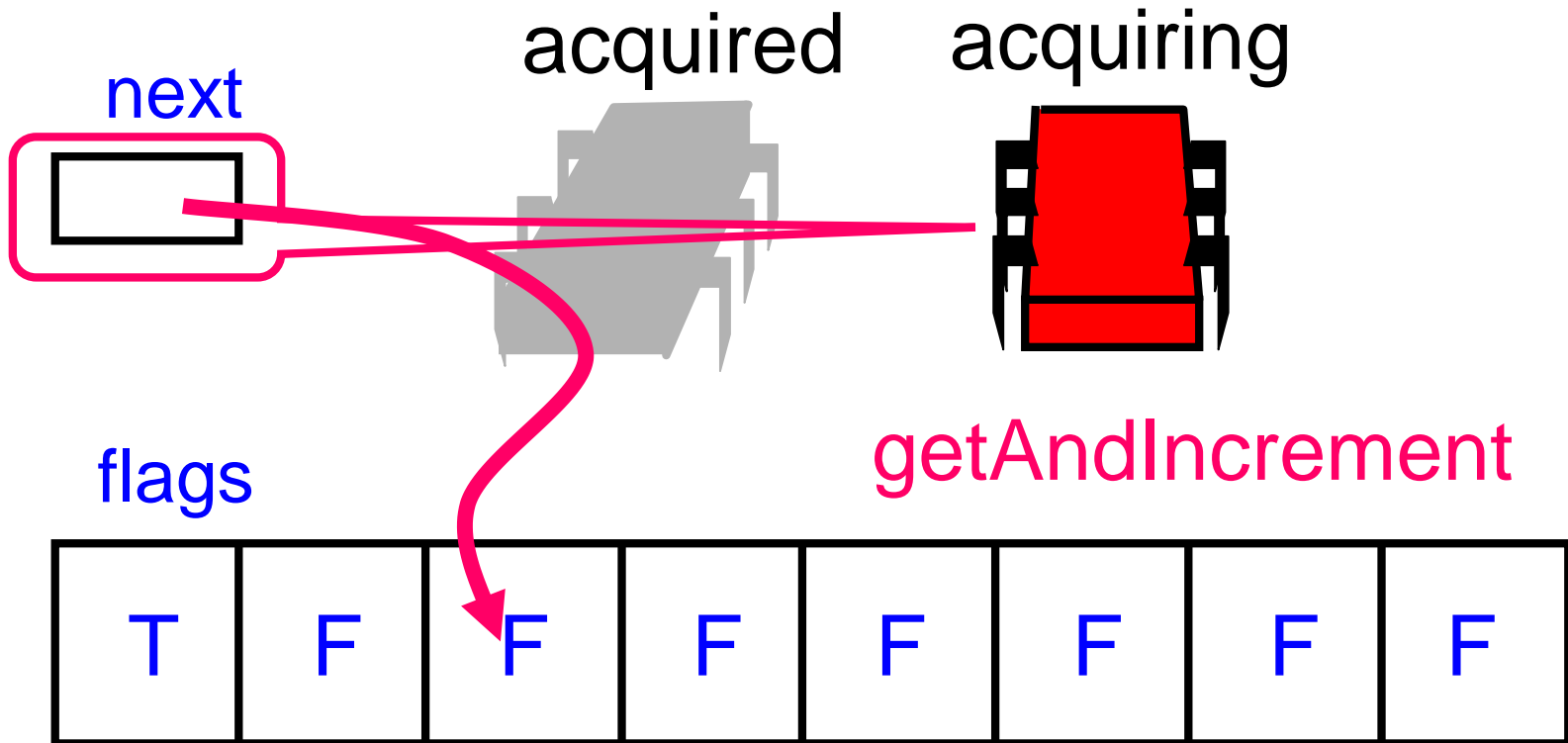
Anderson Queue Lock



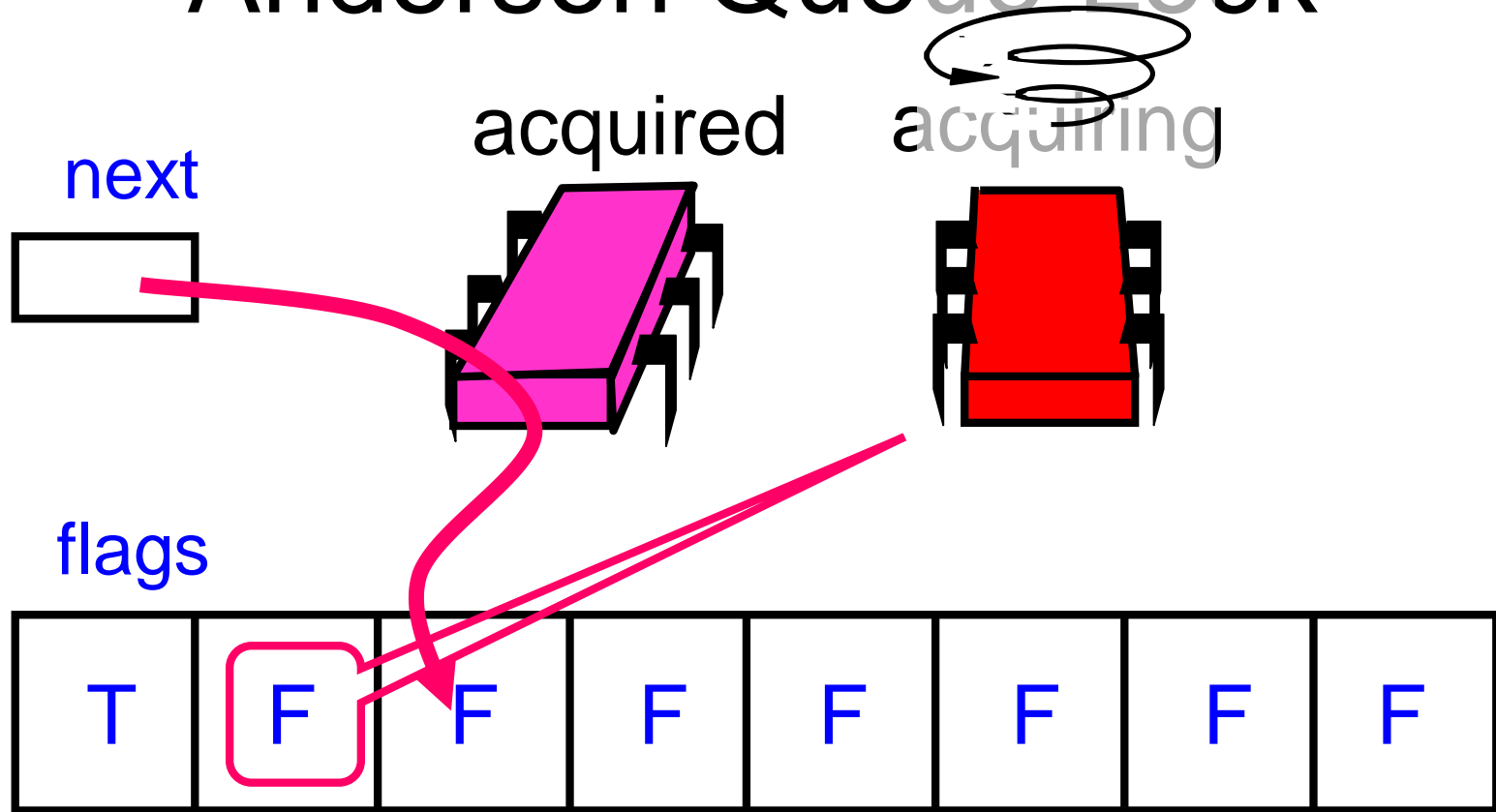
Anderson Queue Lock



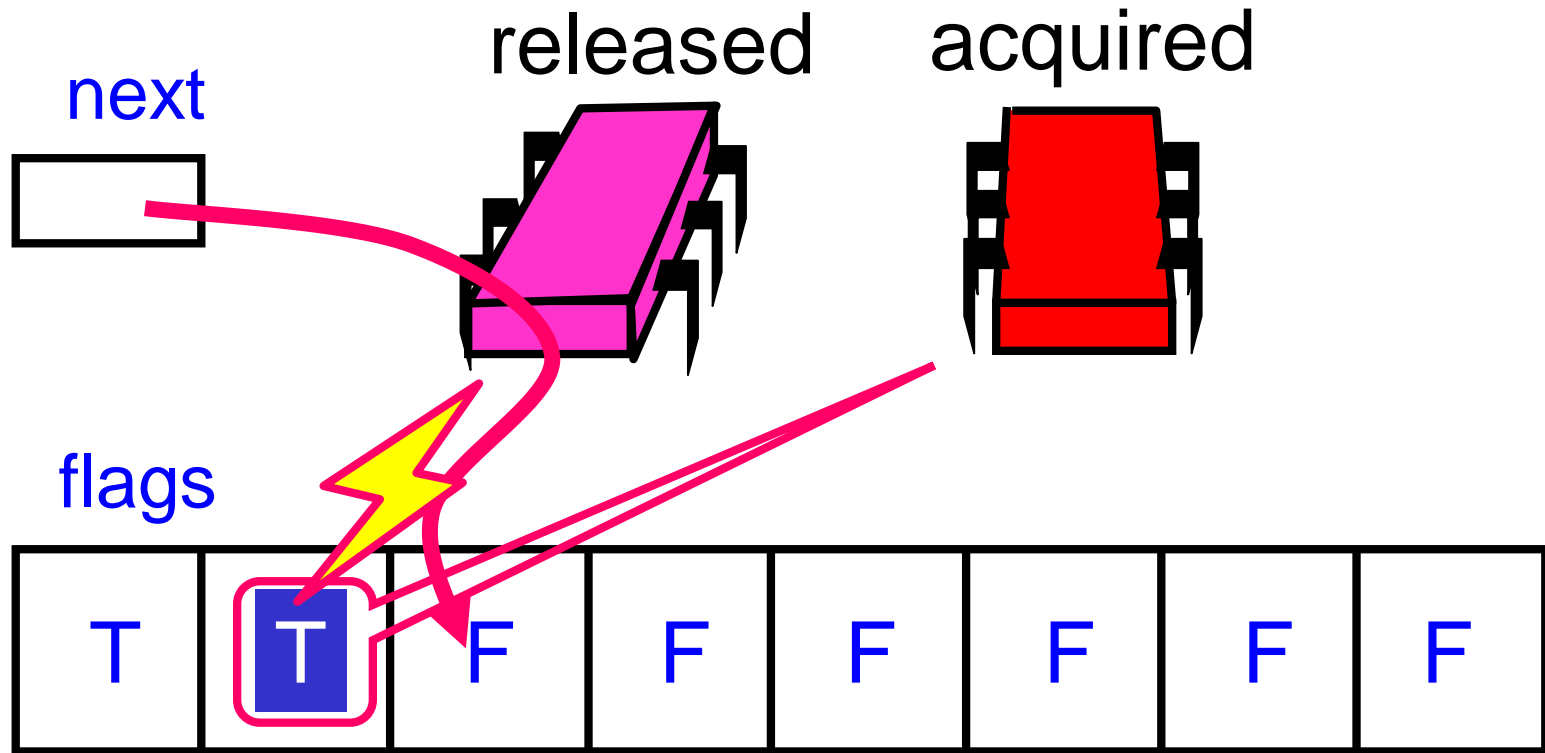
Anderson Queue Lock



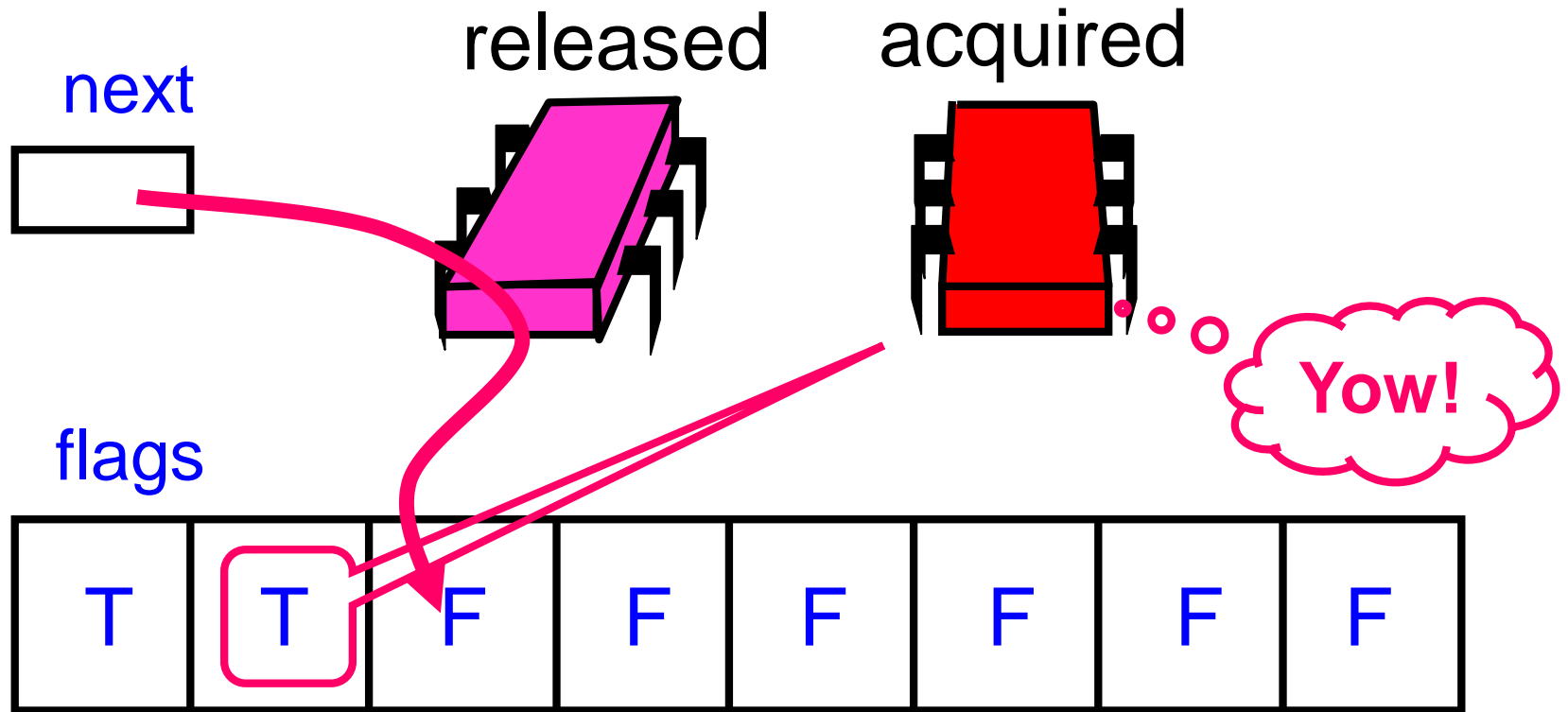
Anderson Queue Lock



Anderson Queue Lock

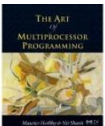


Anderson Queue Lock



Anderson Queue Lock

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



Anderson Queue Lock

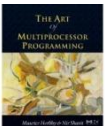
```
class ALock implements Lock {  
    boolean[] flags={true,false,...,false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

One flag per thread

Anderson Queue Lock

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

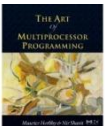
Next flag to use



Anderson Queue Lock

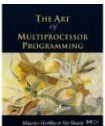
```
class ALock implements Lock {  
    boolean[] flags={true,false,...,false};  
    AtomicInteger next  
        = new AtomicInteger(0);  
    ThreadLocal<Integer> mySlot;
```

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;
```

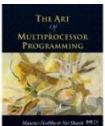
```
}
```

```
public unlock() {
```

```
    flags[(mySlot+1) % n]
```

```
}
```

Take next slot

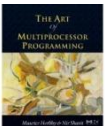


Anderson Queue Lock

```
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

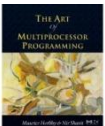


Anderson Queue Lock

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

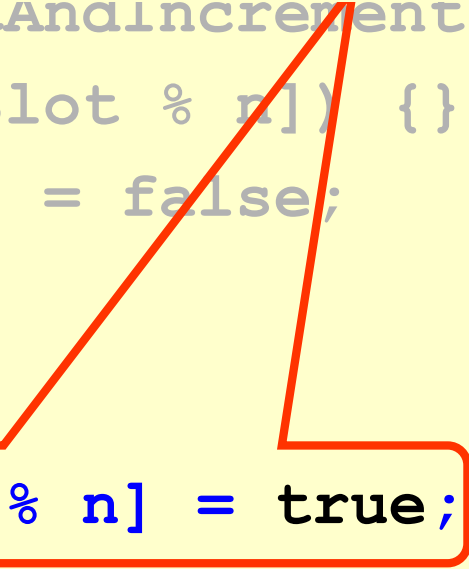
```
public unlock() {  
    flags[(myslot+1) % n] = true;  
}
```

Prepare slot for re-use

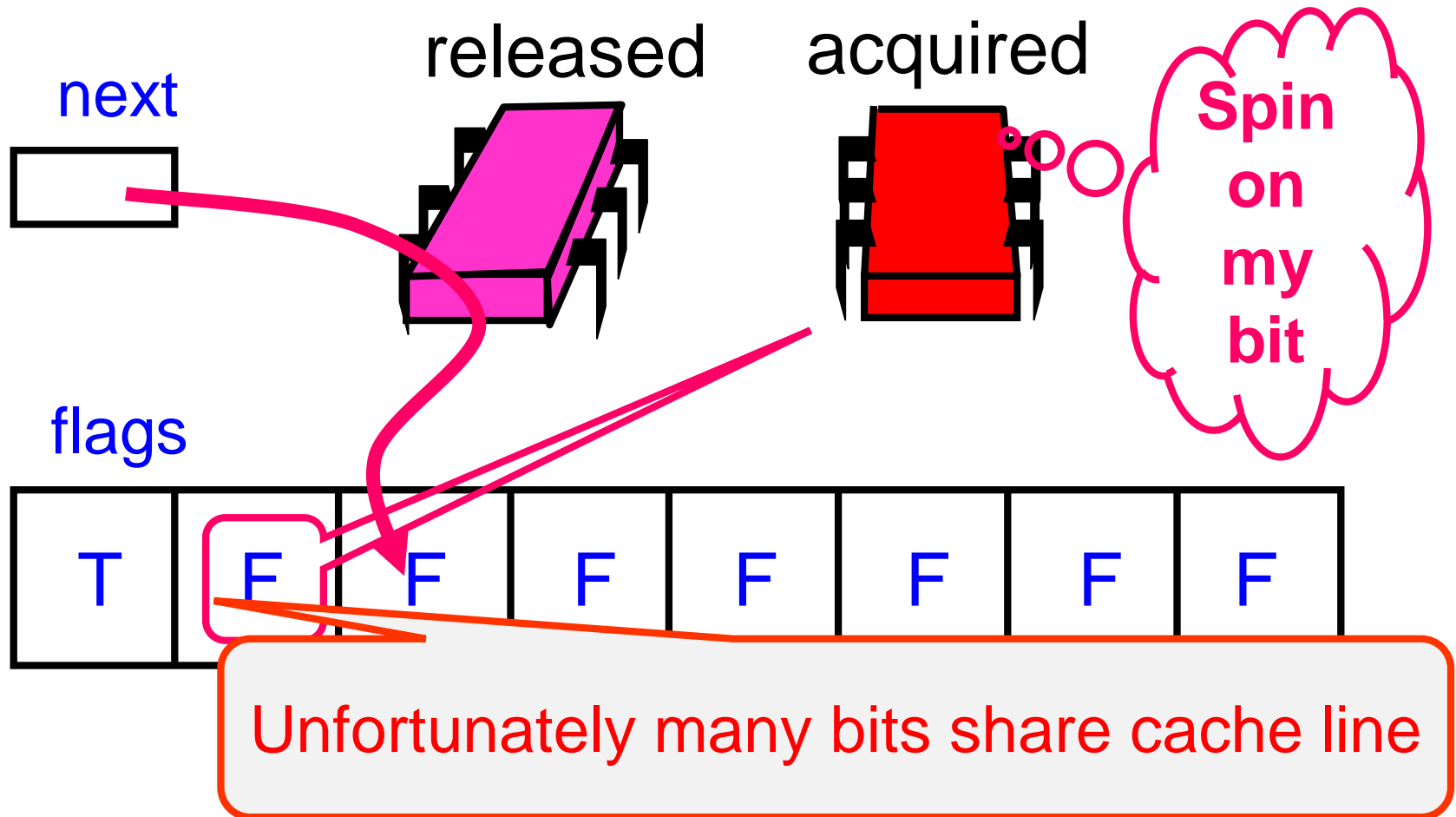


Anderson Queue Lock

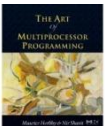
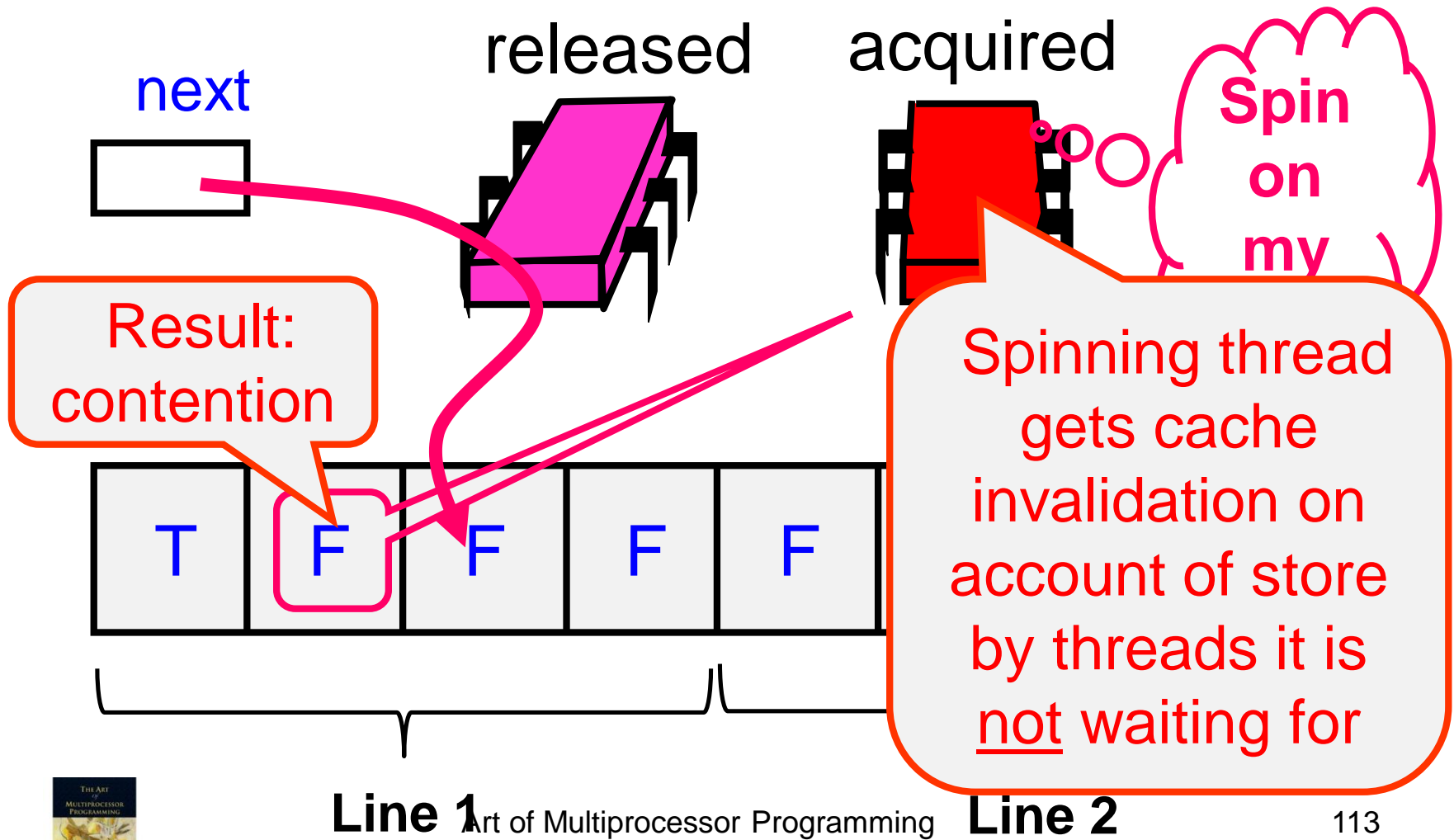
```
public lock() { Tell next thread to go  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot % n]) {};  
    flags[mySlot % n] = false;  
}  
  
public unlock() {  
    flags[(mySlot+1) % n] = true;  
}
```



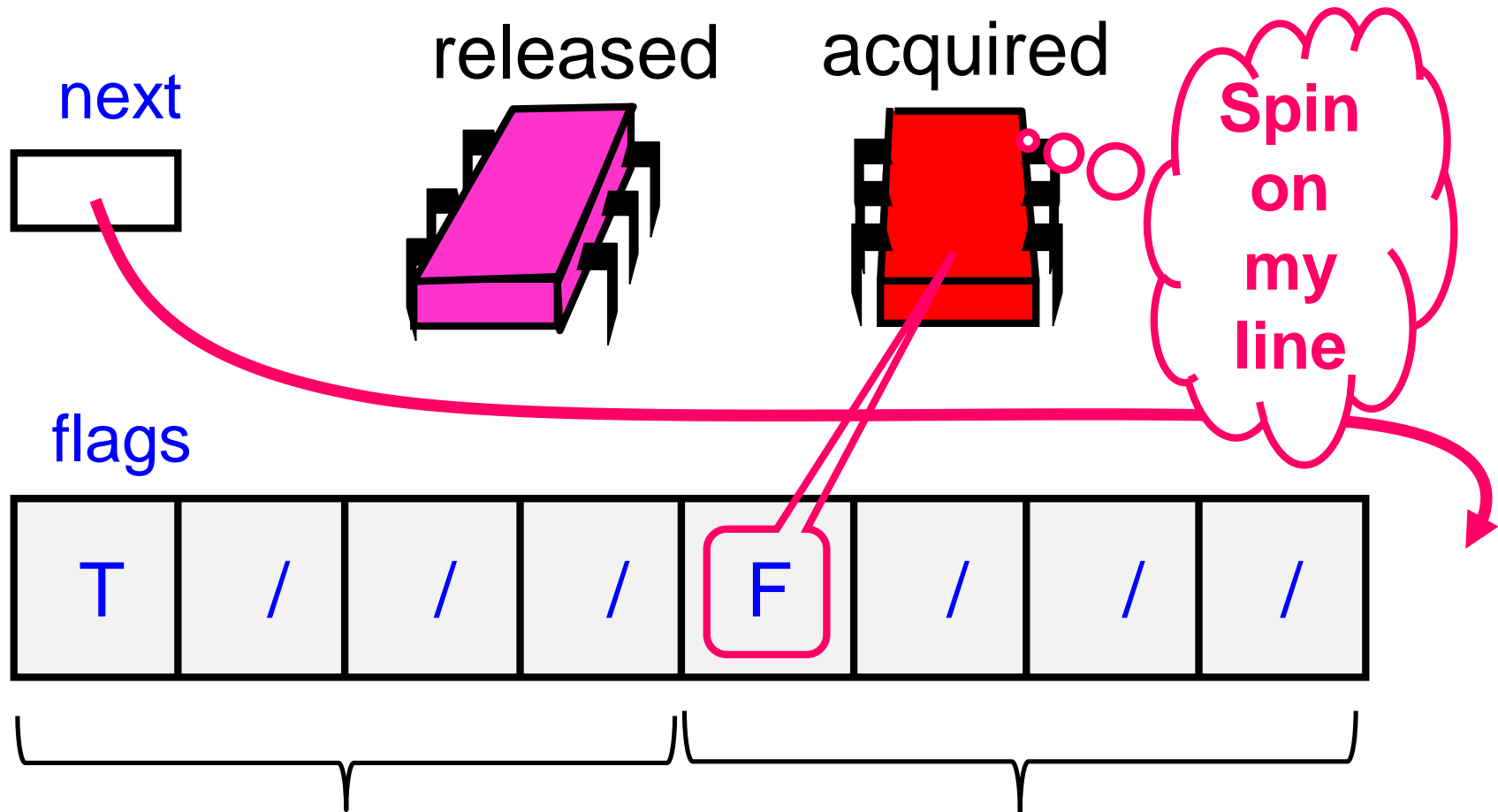
Local Spinning



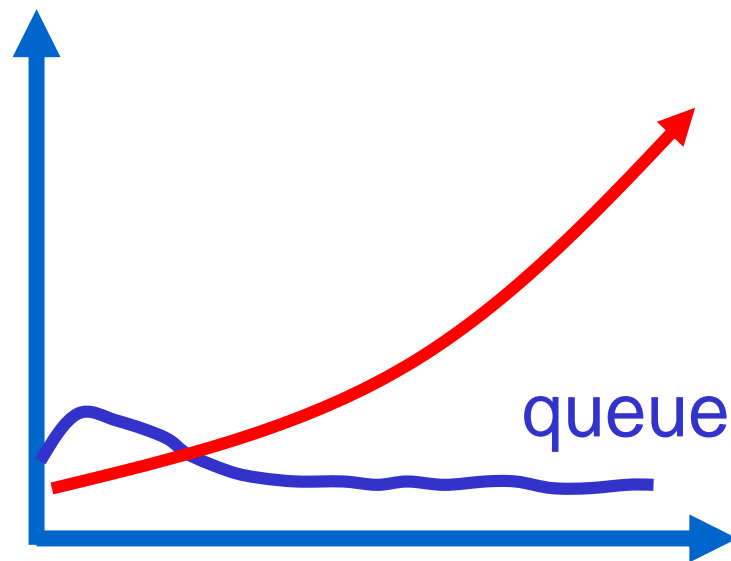
False Sharing



The Solution: Padding

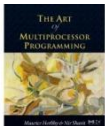


Performance



TTAS

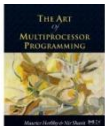
- Shorter handover than backoff
- Curve is practically flat
- Scalable performance



Anderson Queue Lock

Good

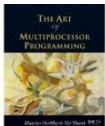
- First truly scalable lock
- Simple, easy to implement
- Back to FCFS order (like Bakery)



Anderson Queue Lock

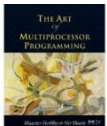
Bad

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

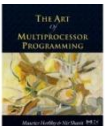
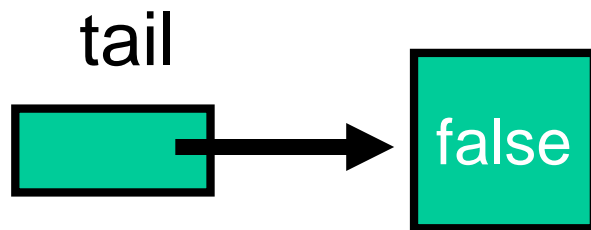
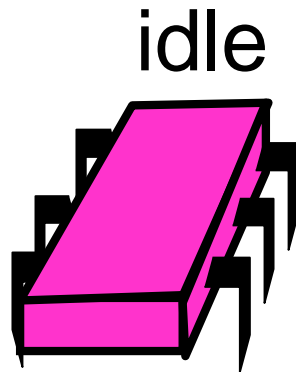


CLH Lock

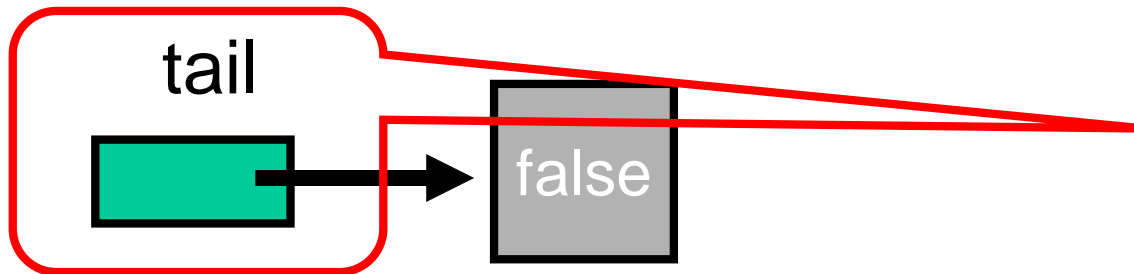
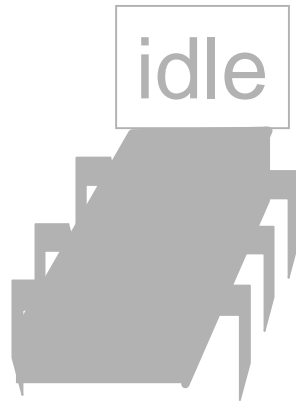
- FCFS order
- Small, constant-size overhead per thread



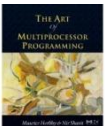
Initially



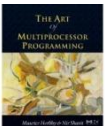
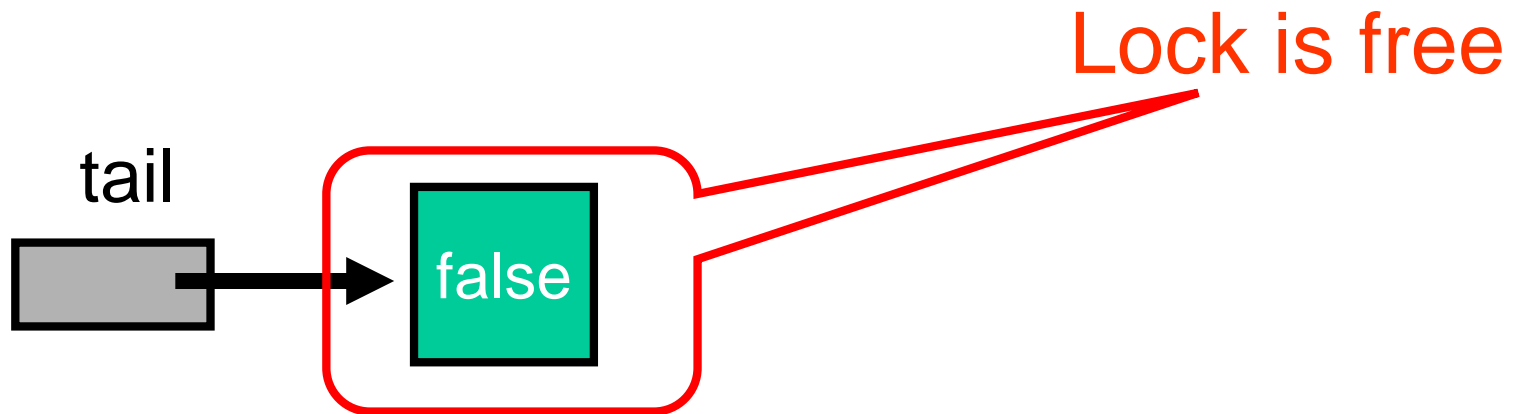
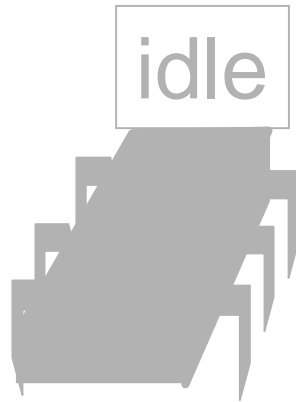
Initially



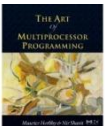
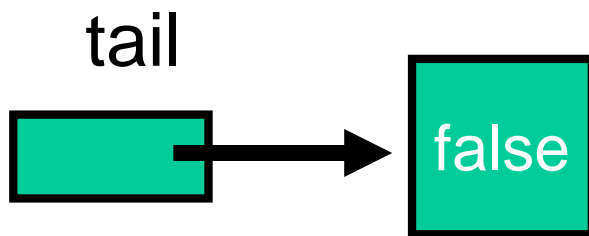
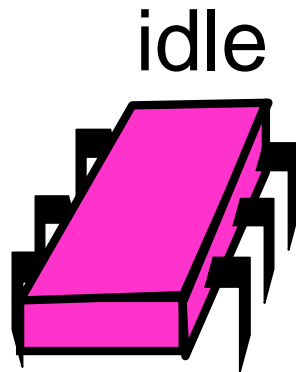
Queue tail



Initially

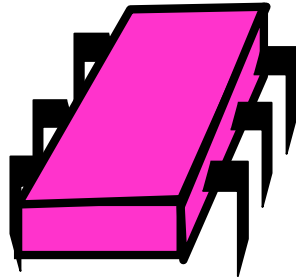


Initially

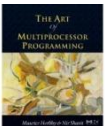
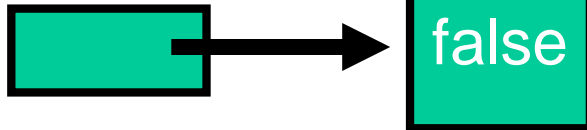


Purple Wants the Lock

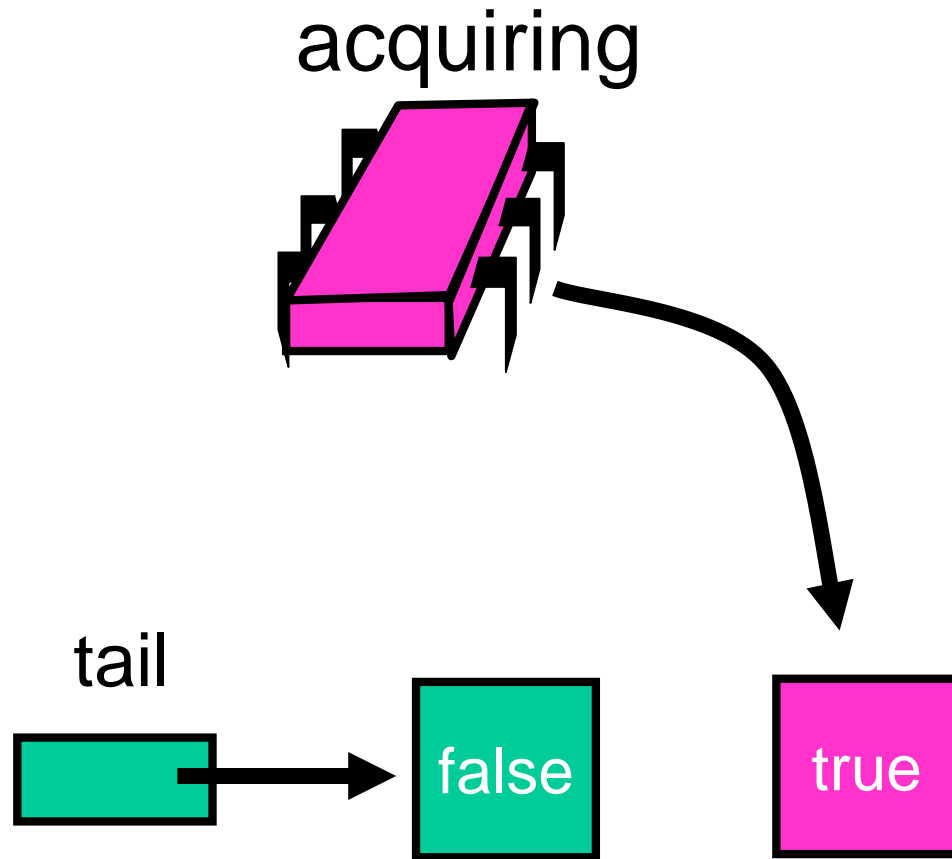
acquiring



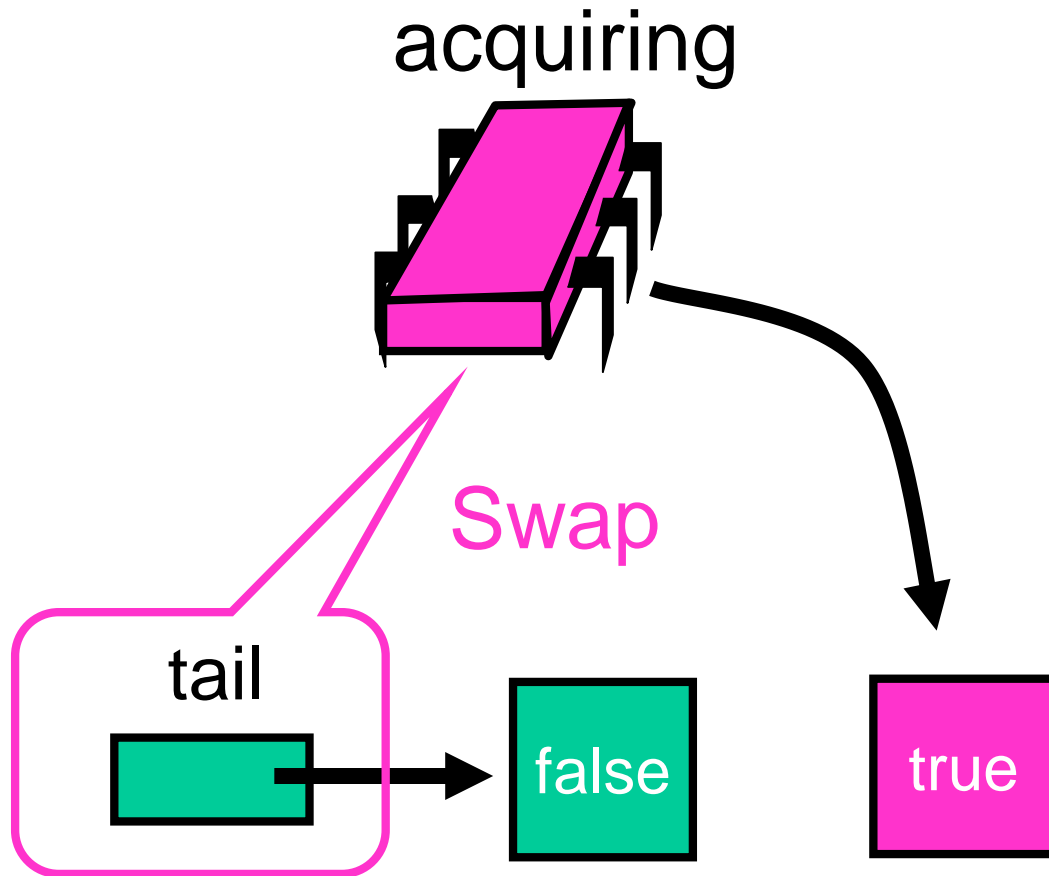
tail



Purple Wants the Lock

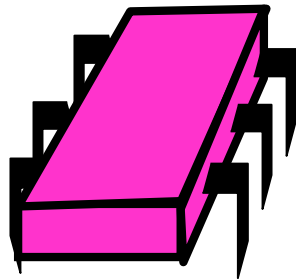


Purple Wants the Lock

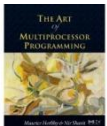
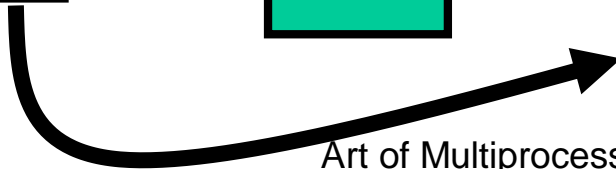
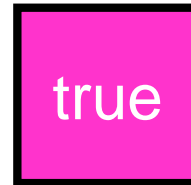
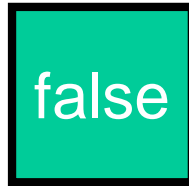


Purple Has the Lock

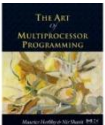
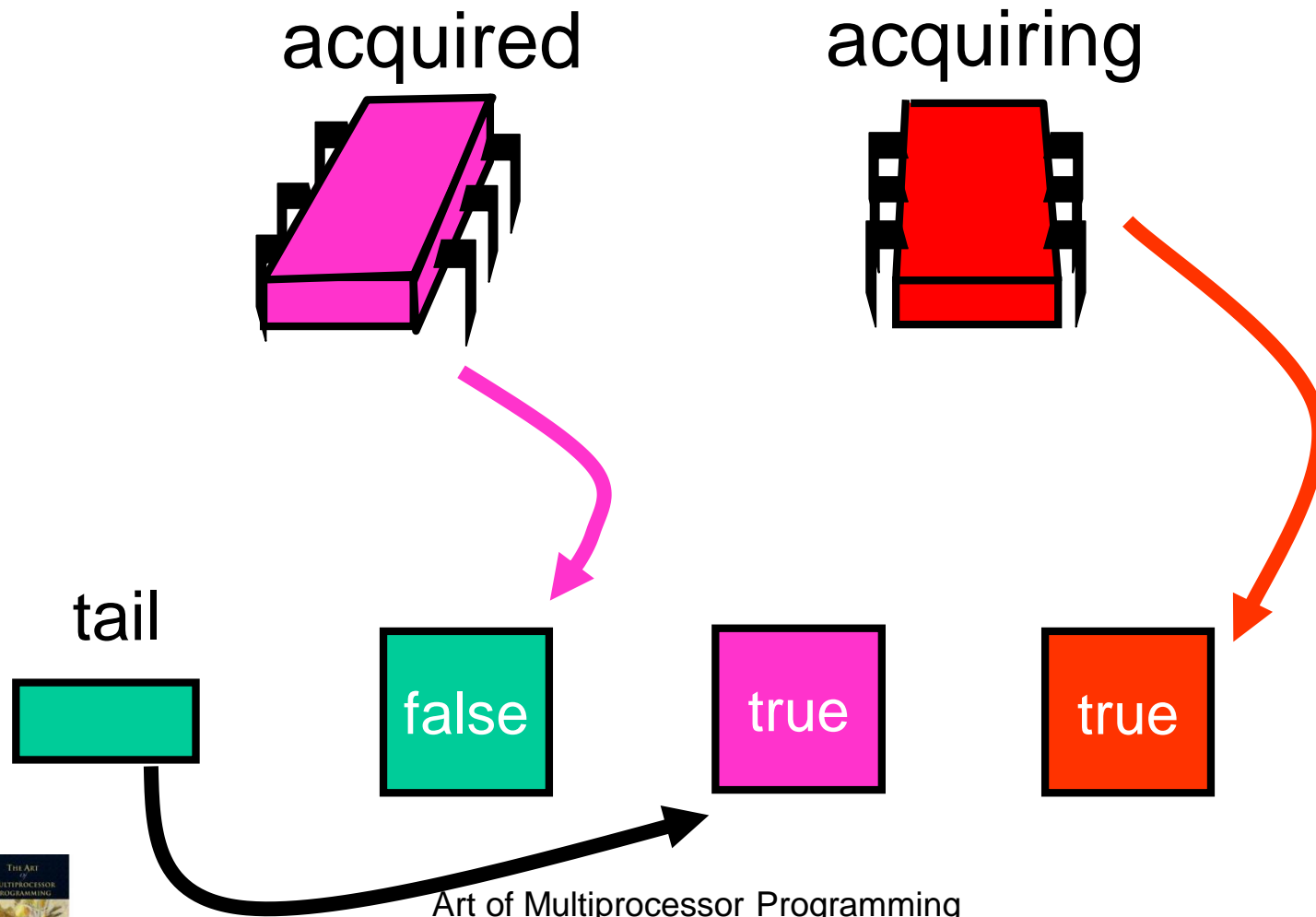
acquired



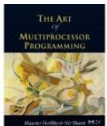
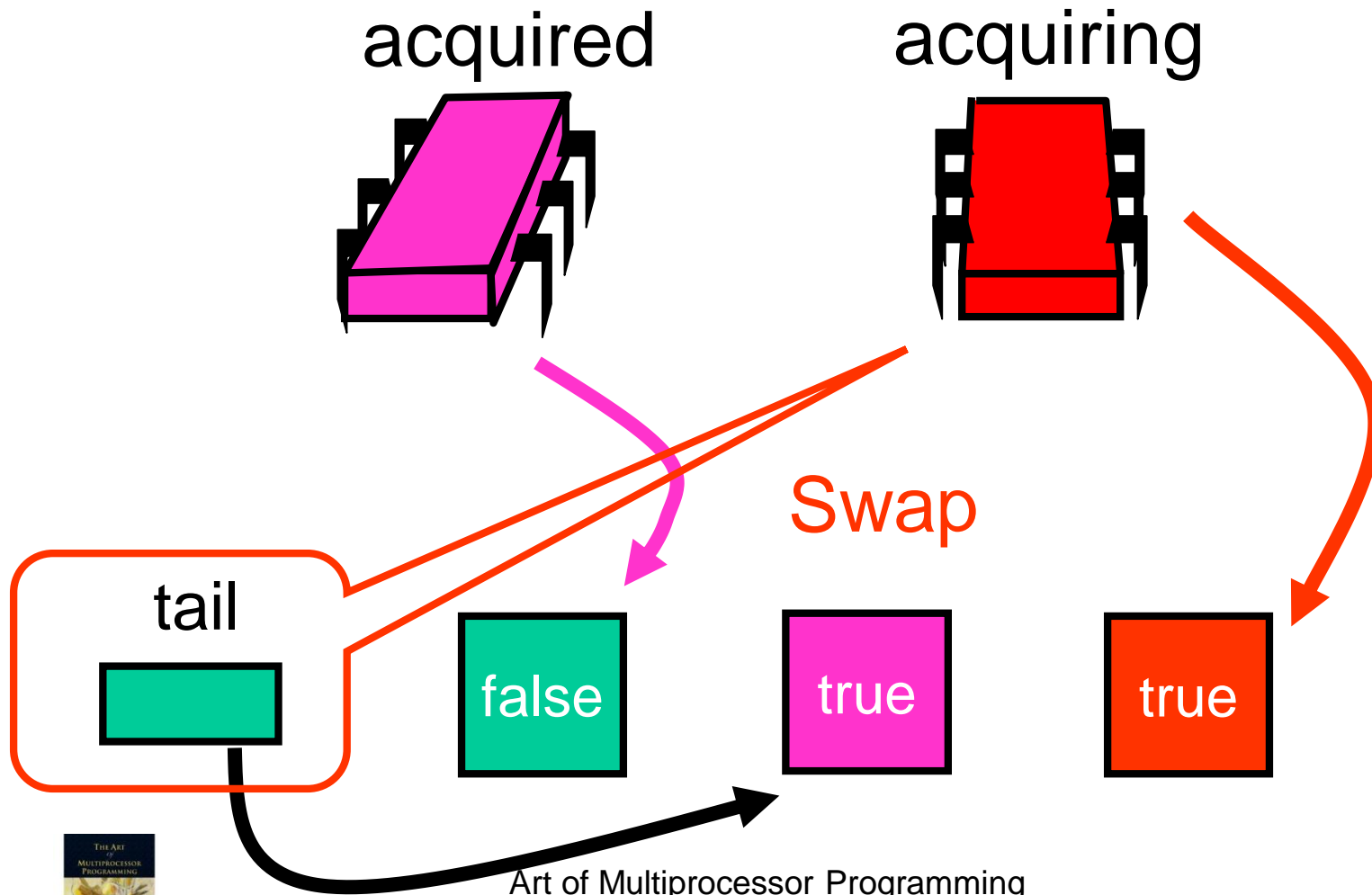
tail



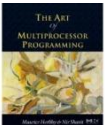
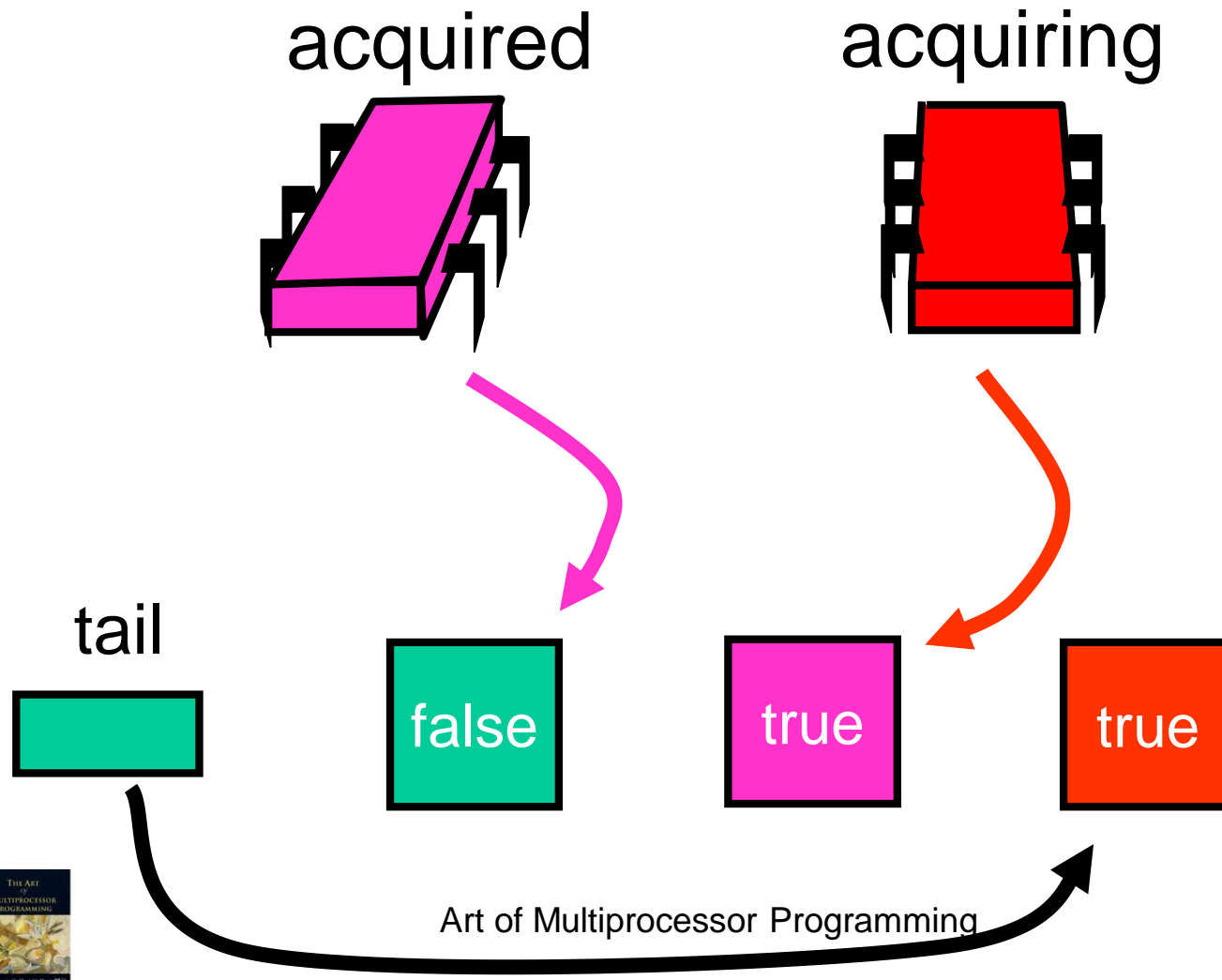
Red Wants the Lock



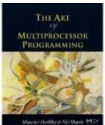
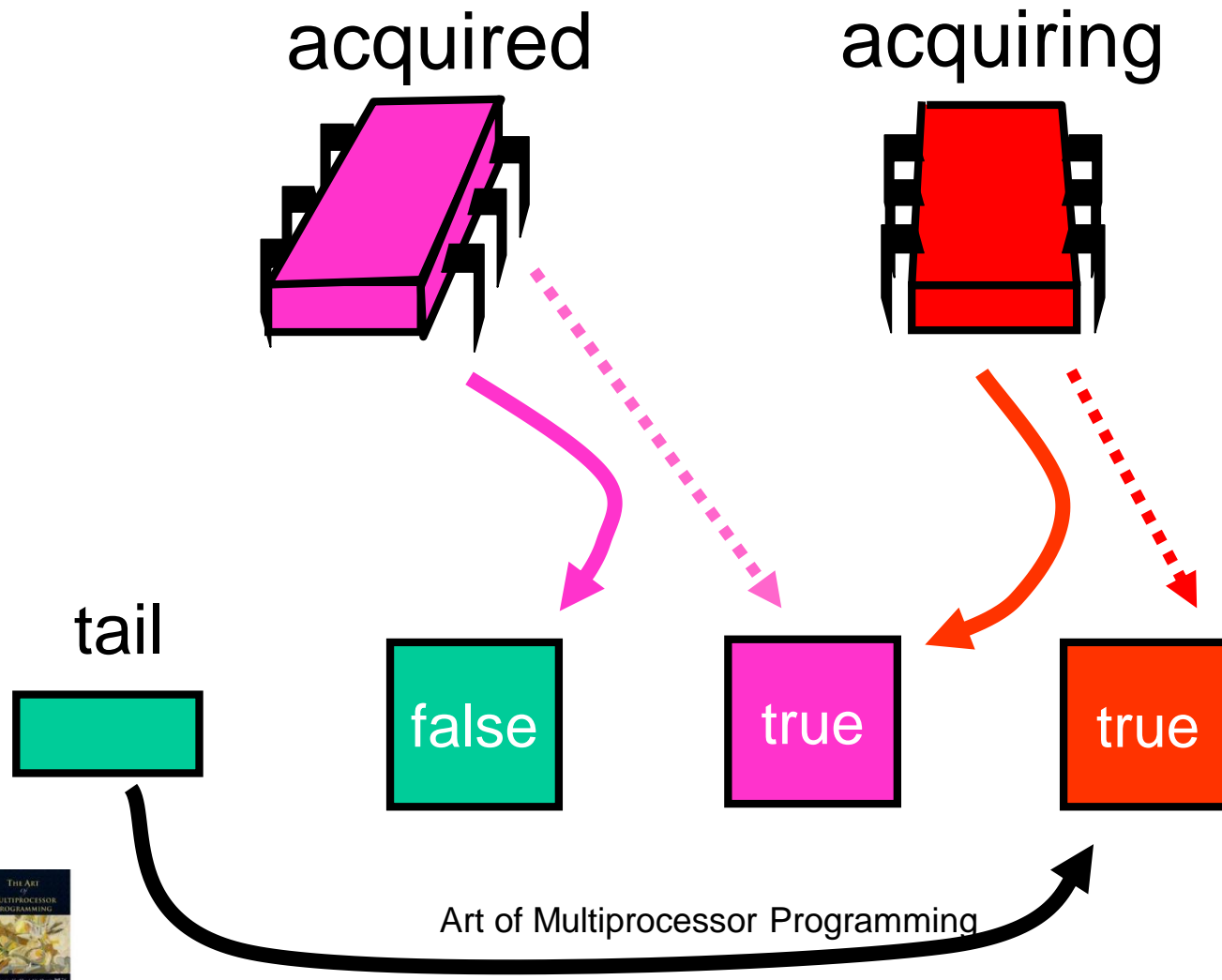
Red Wants the Lock



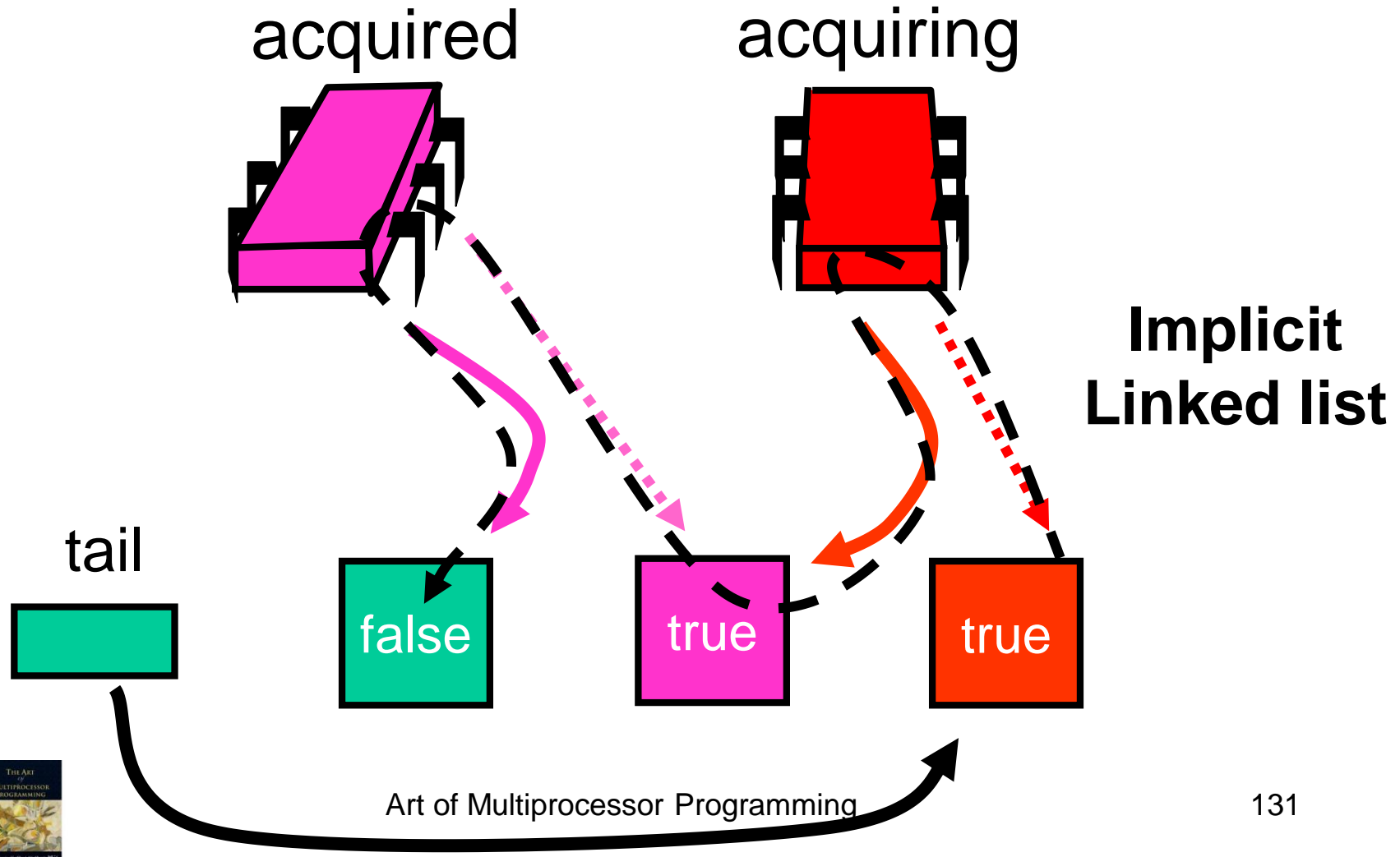
Red Wants the Lock



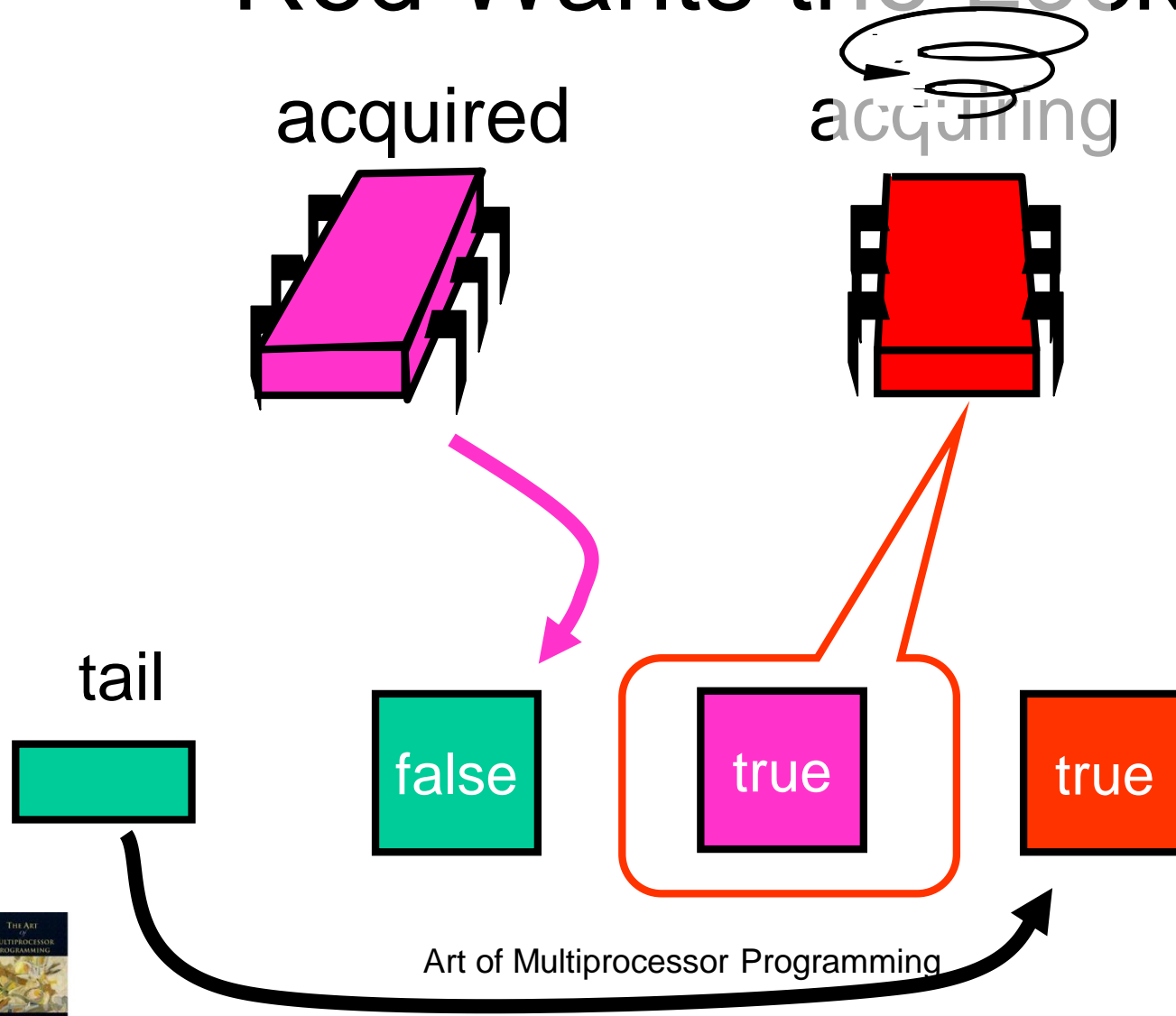
Red Wants the Lock



Red Wants the Lock

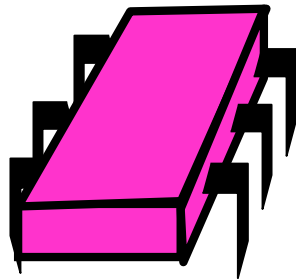


Red Wants the Lock

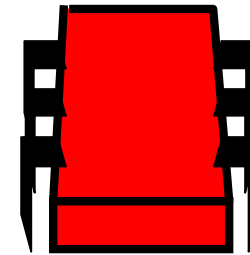


Red Wants the Lock

acquired



acquiring



true

Actually, it spins on cached copy

tail

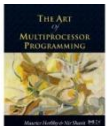


false

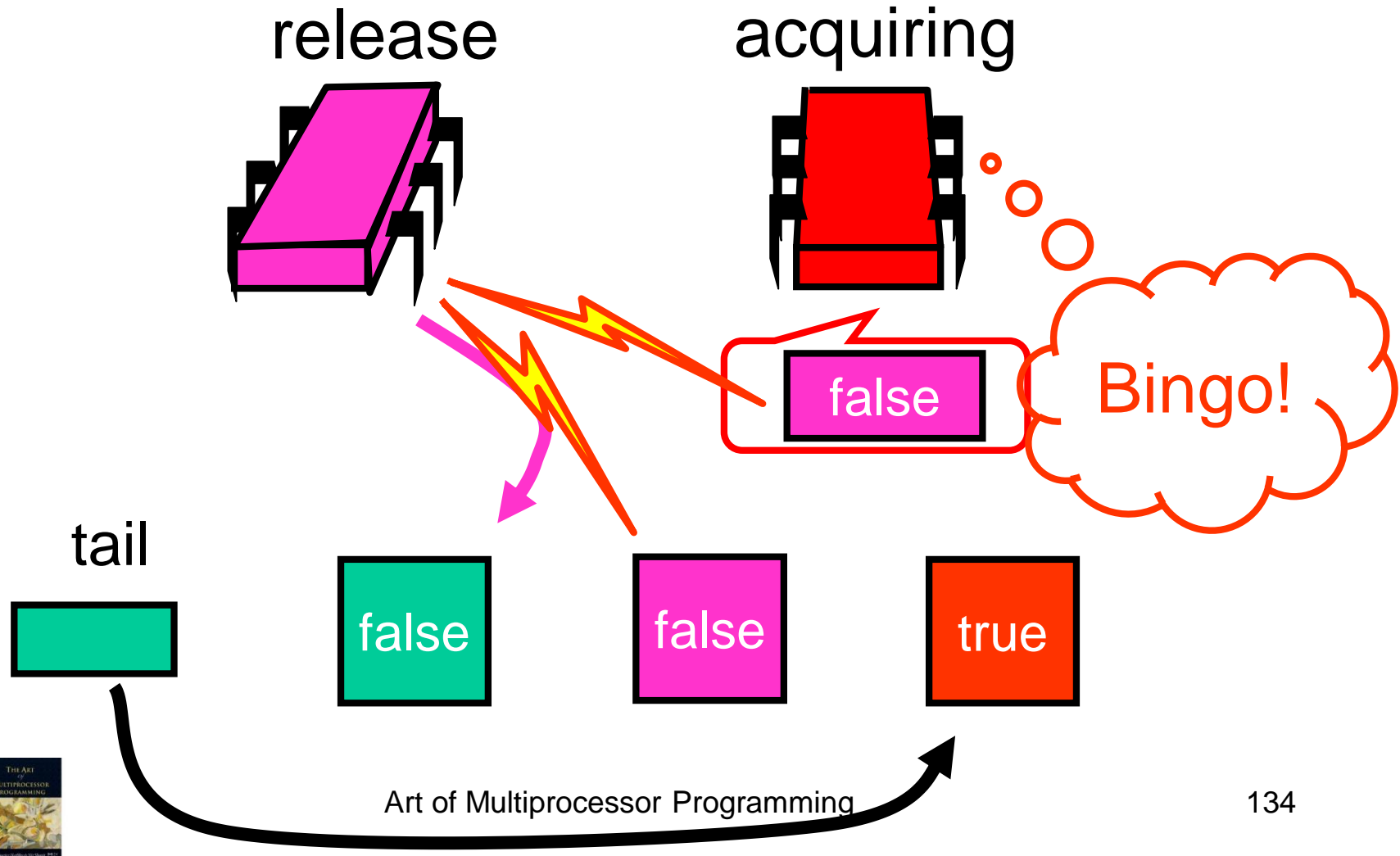
true

true

Art of Multiprocessor Programming

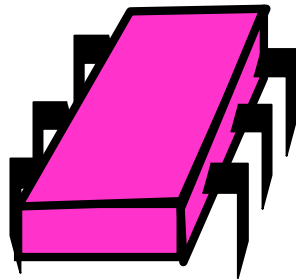


Purple Releases

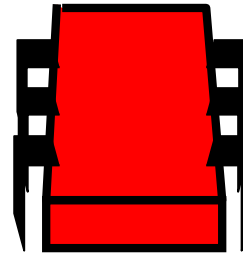


Purple Releases

released



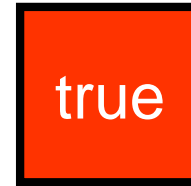
acquired



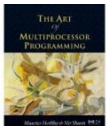
tail



true

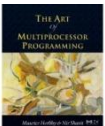


Art of Multiprocessor Programming



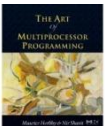
Space Usage

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



CLH Queue Lock

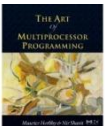
```
class QNode {  
    AtomicBoolean locked =  
        new AtomicBoolean(true) ;  
}
```



CLH Queue Lock

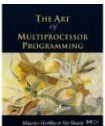
```
class QNode {  
    AtomicBoolean locked =  
        new AtomicBoolean(true);  
}
```

Not released yet



CLH Queue Lock

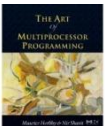
```
class CLHLock implements Lock {  
    AtomicReference<QNode> tail;  
    ThreadLocal<QNode> myNode  
        = new QNode();  
    public void lock() {  
        QNode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```



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) {}  
    }  
}
```

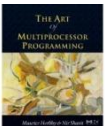
Queue tail



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) {}  
    }  
}
```

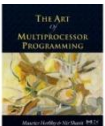
Thread-local QNode



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) {}  
    }  
}
```

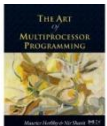
Swap in my node



CLH Queue Lock

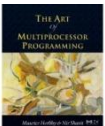
```
class CLHLock implements Lock {  
    AtomicReference<QNode> tail;  
    ThreadLocal<QNode> myNode  
        = new QNode();  
    public void lock() {  
        QNode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

Spin until predecessor
releases lock



CLH Queue Lock

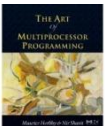
```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```



CLH Queue Lock

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

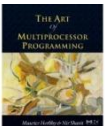
Notify successor



CLH Queue Lock

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

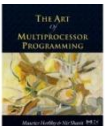
Recycle
predecessor's node



CLH Queue Lock

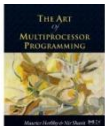
```
Class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

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



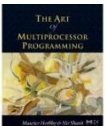
CLH Lock

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

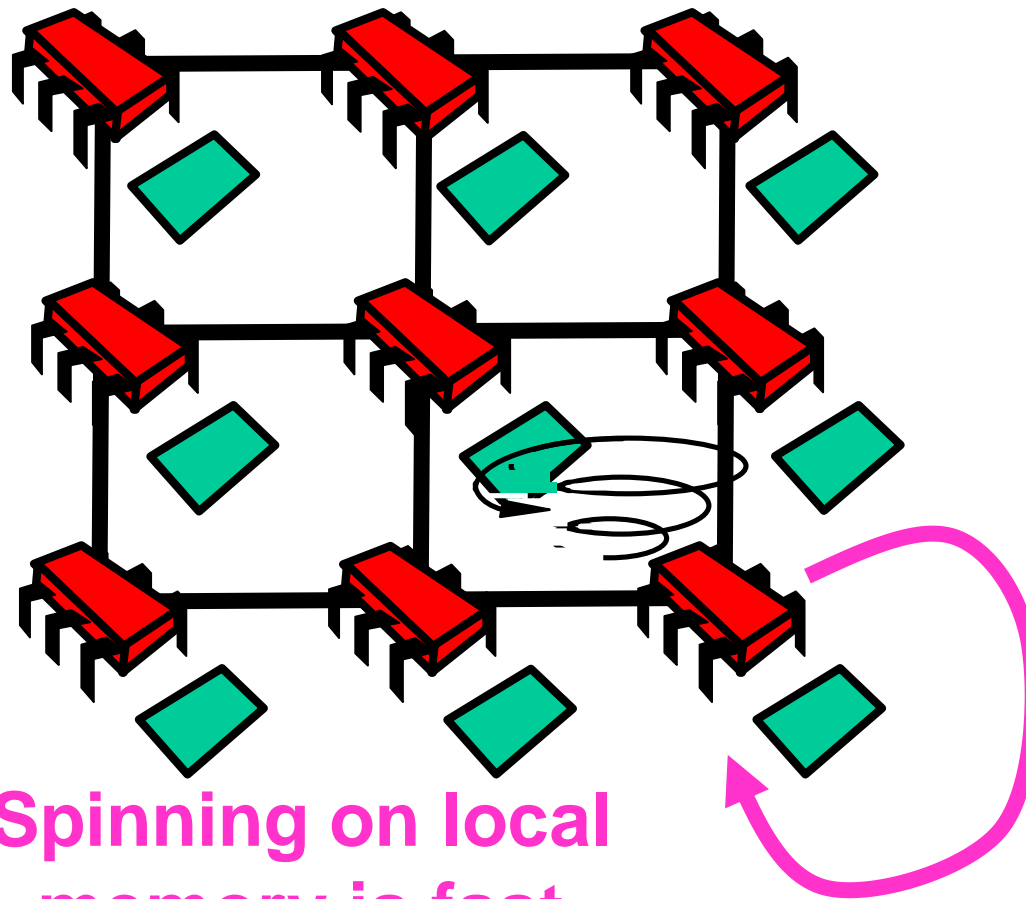


NUMA and cc-NUMA Architectures

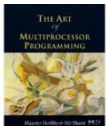
- Acronym:
 - **N**on-**U**niform **M**emory **A**rchitecture
 - ccNUMA = cache coherent NUMA
- Illusion:
 - Flat shared memory
- Truth:
 - No caches (sometimes)
 - Some memory regions faster than others



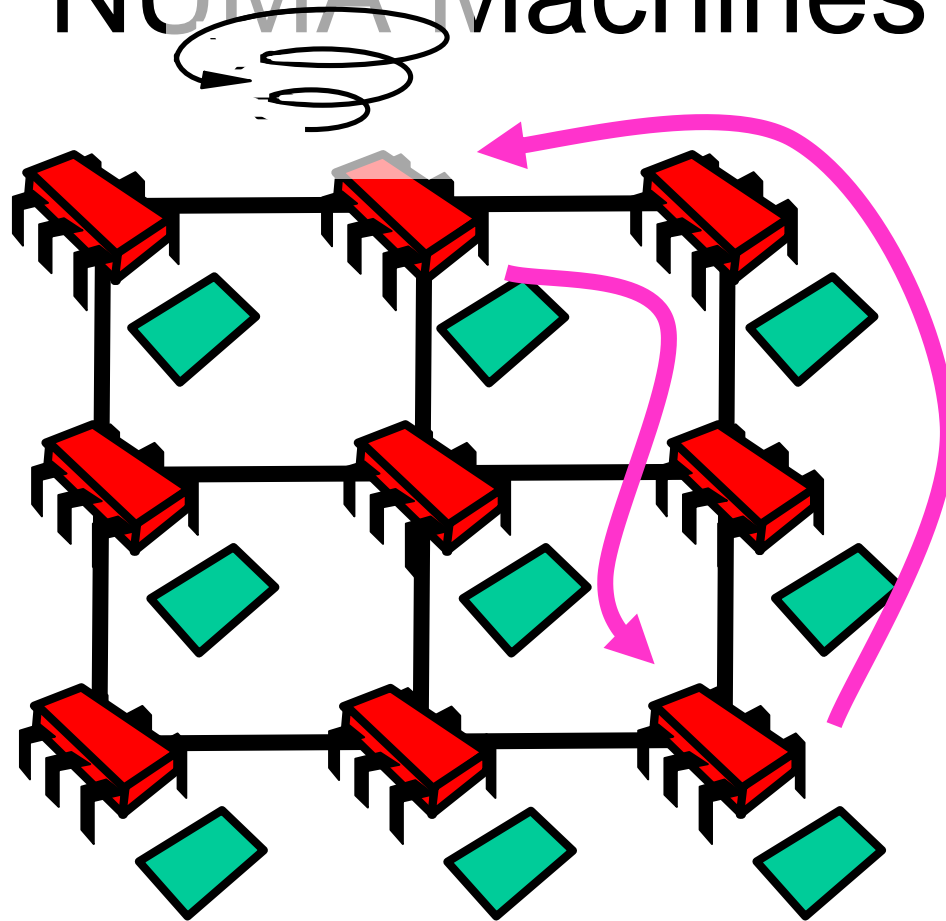
NUMA Machines



Spinning on local
memory is fast



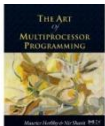
NUMA Machines



**Spinning on remote
memory is slow**

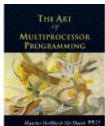
CLH Lock

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

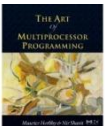
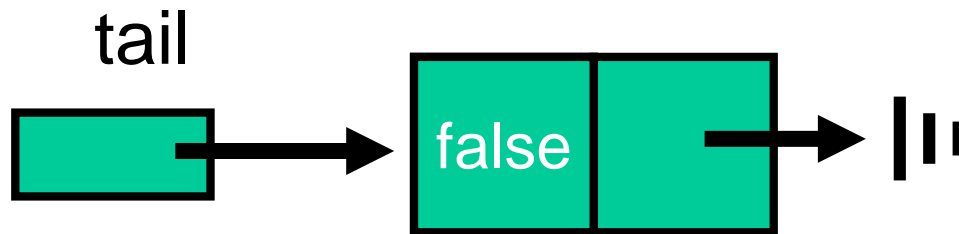
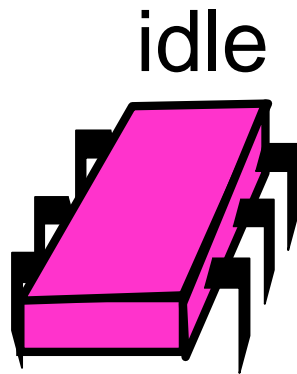


MCS Lock

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

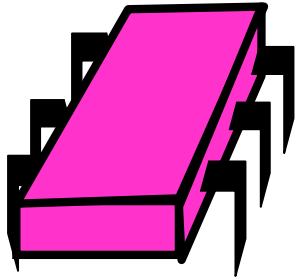


Initially

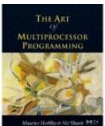
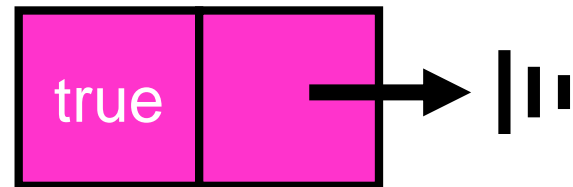
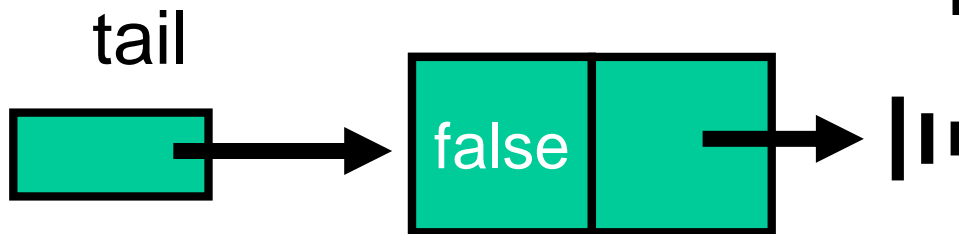


Acquiring

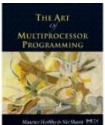
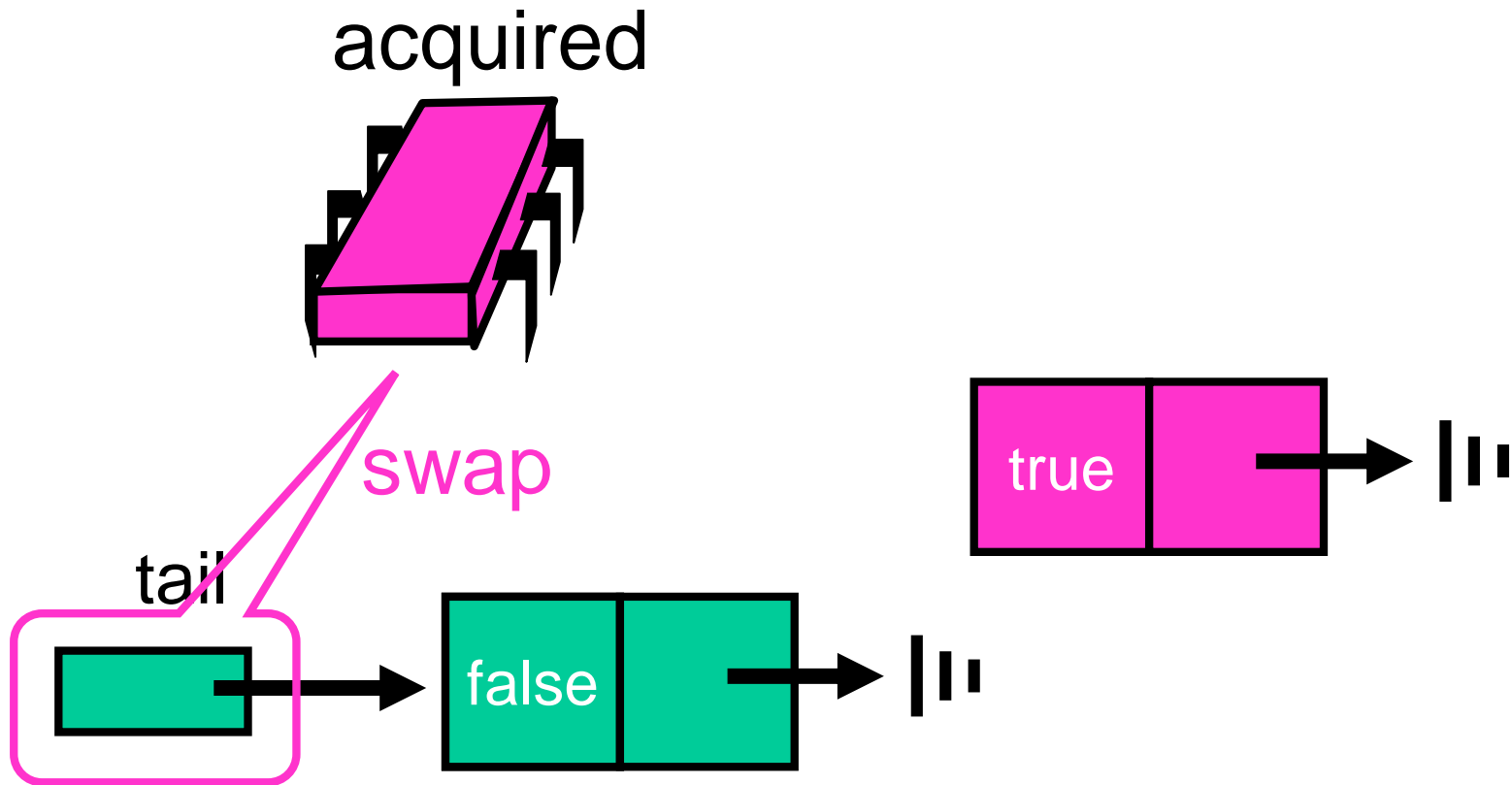
acquiring



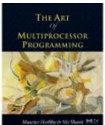
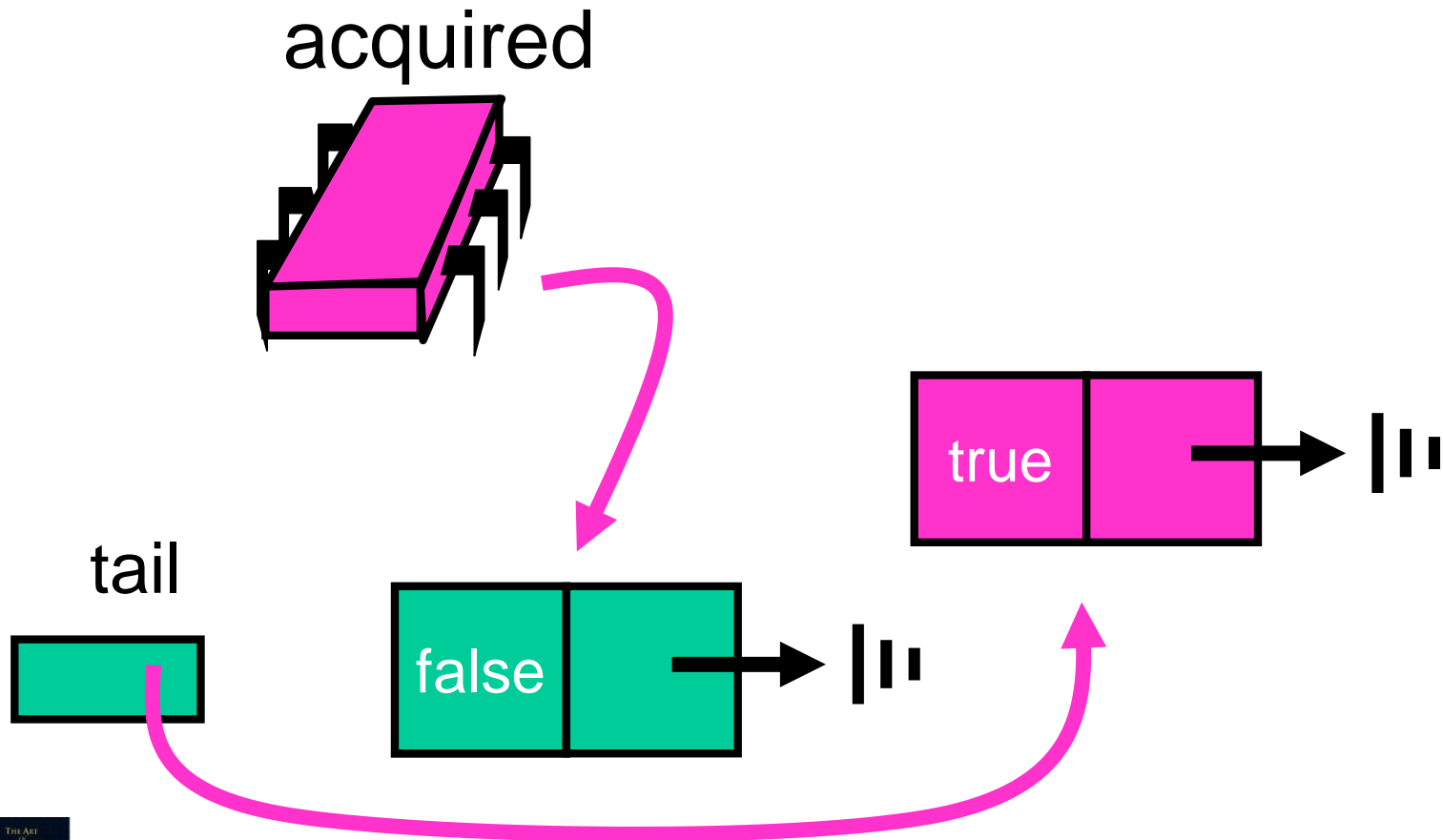
(allocate QNode)



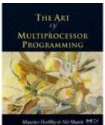
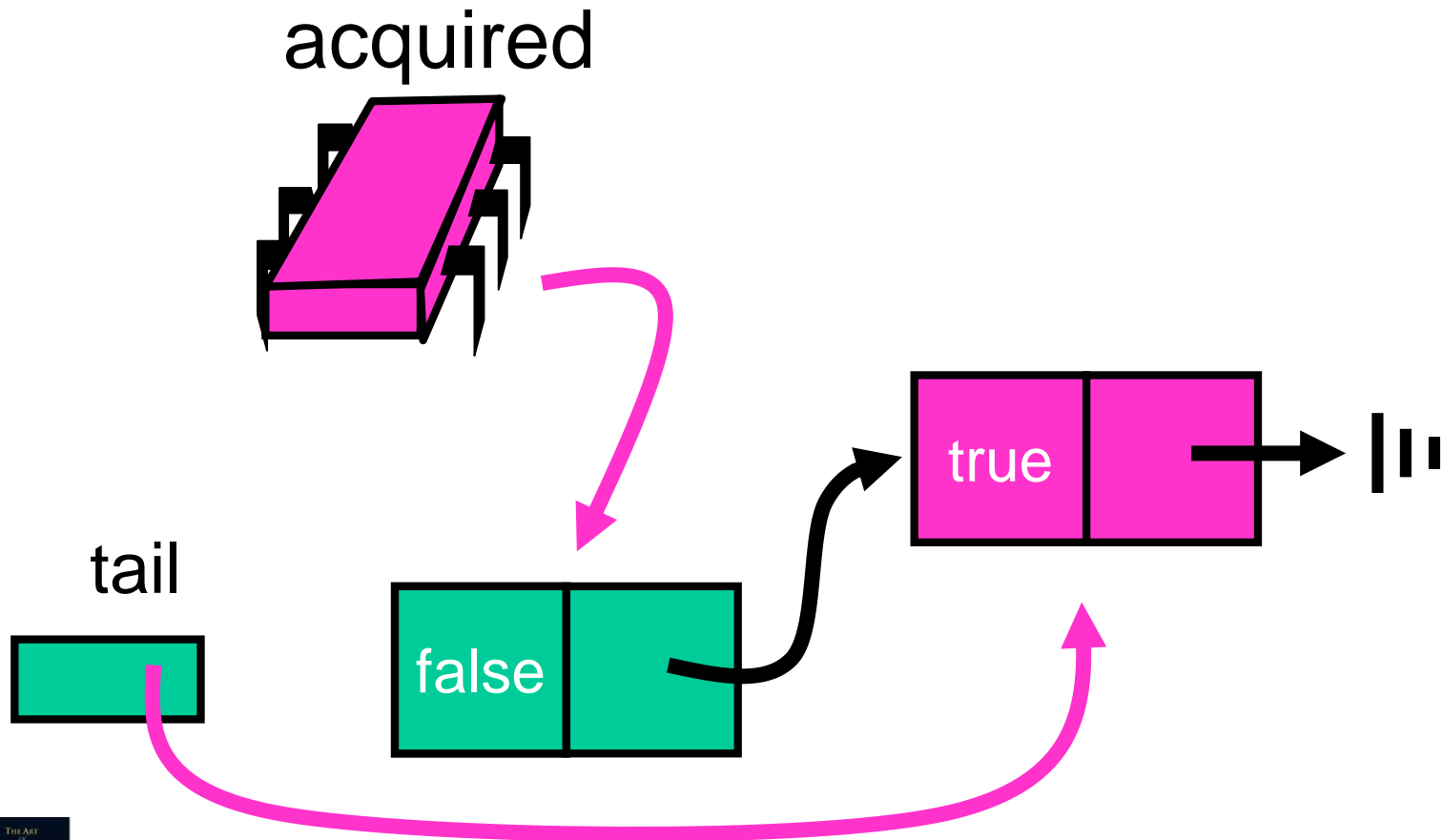
Acquiring



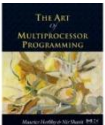
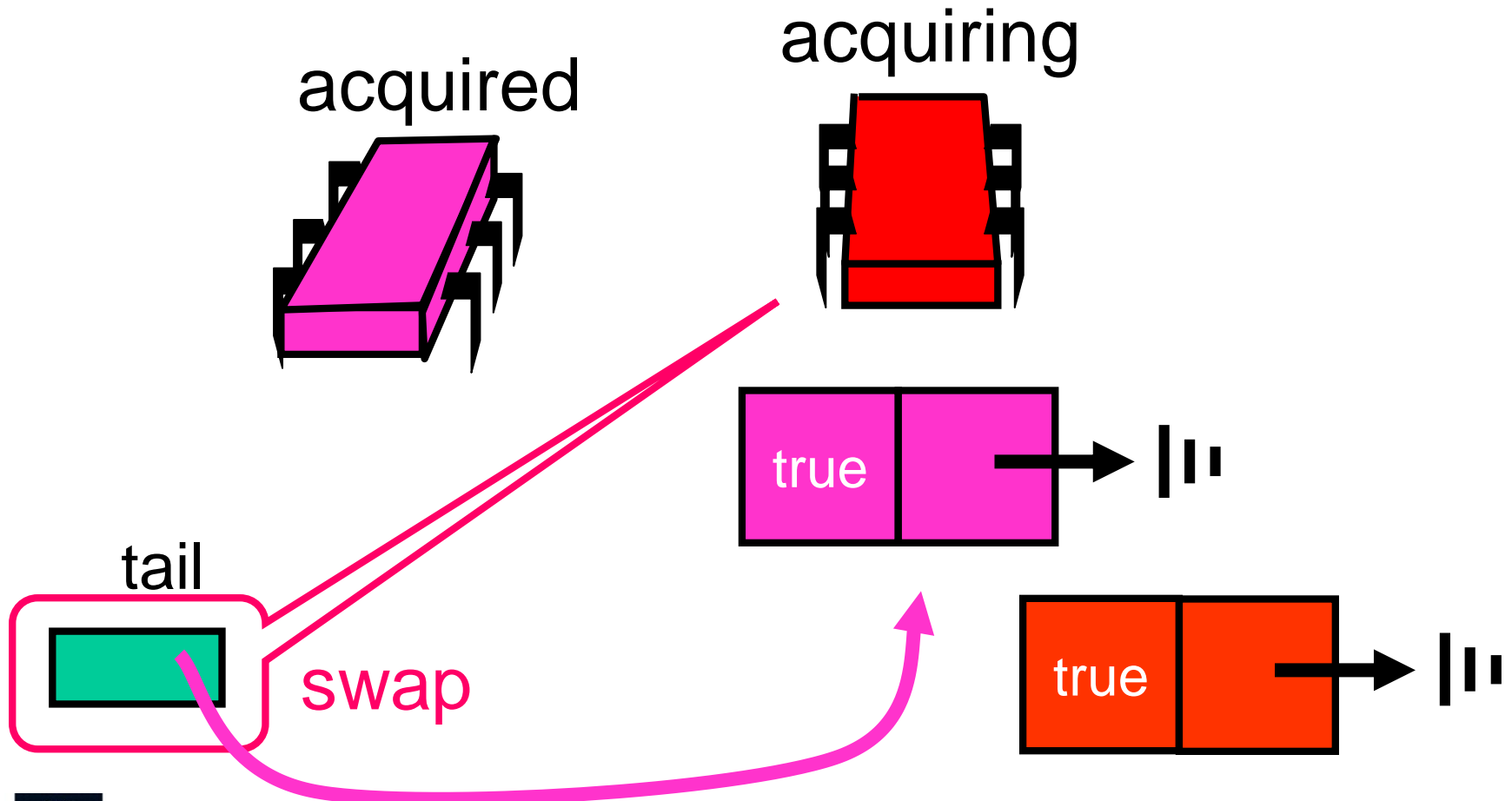
Acquiring



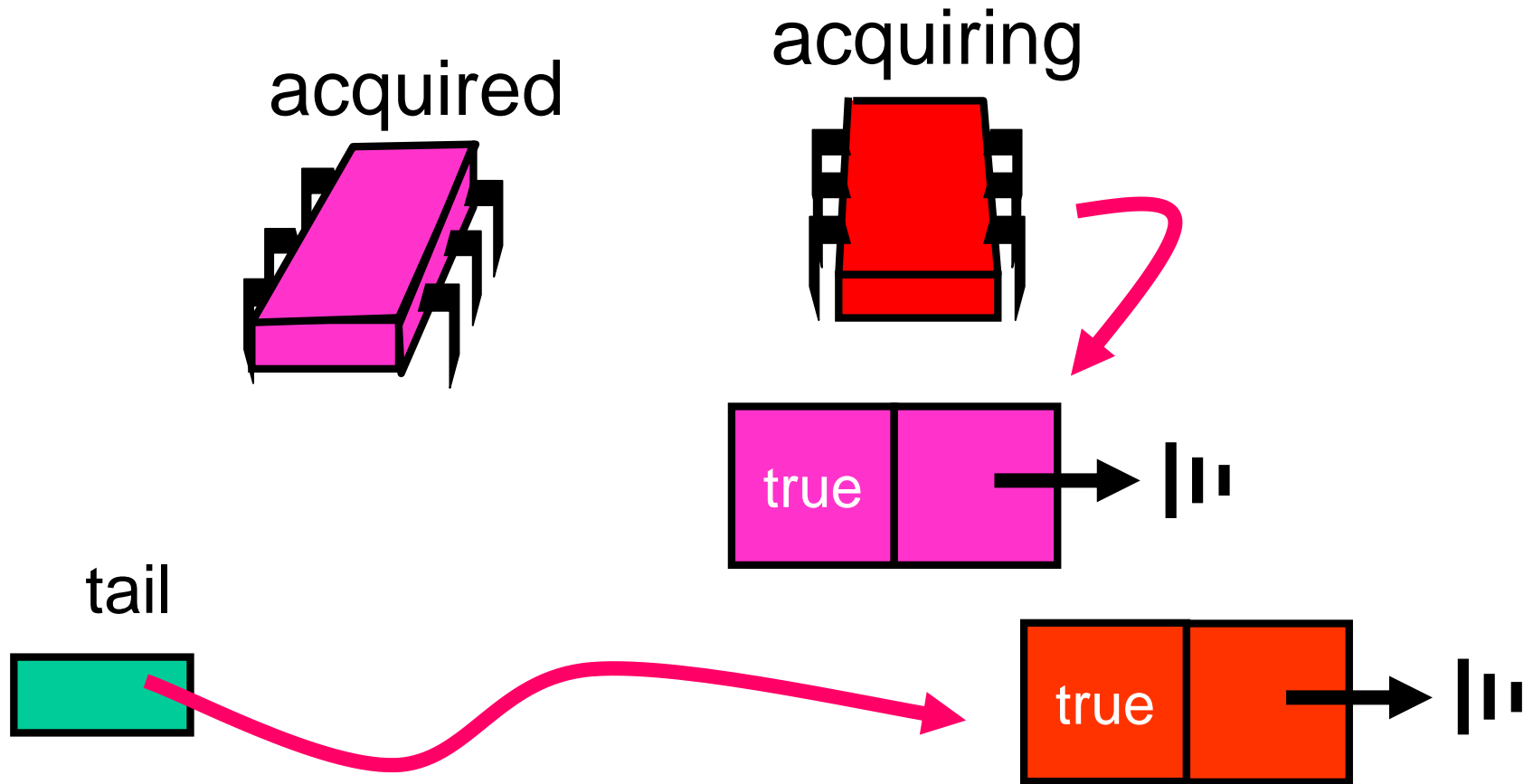
Acquired



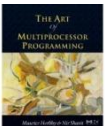
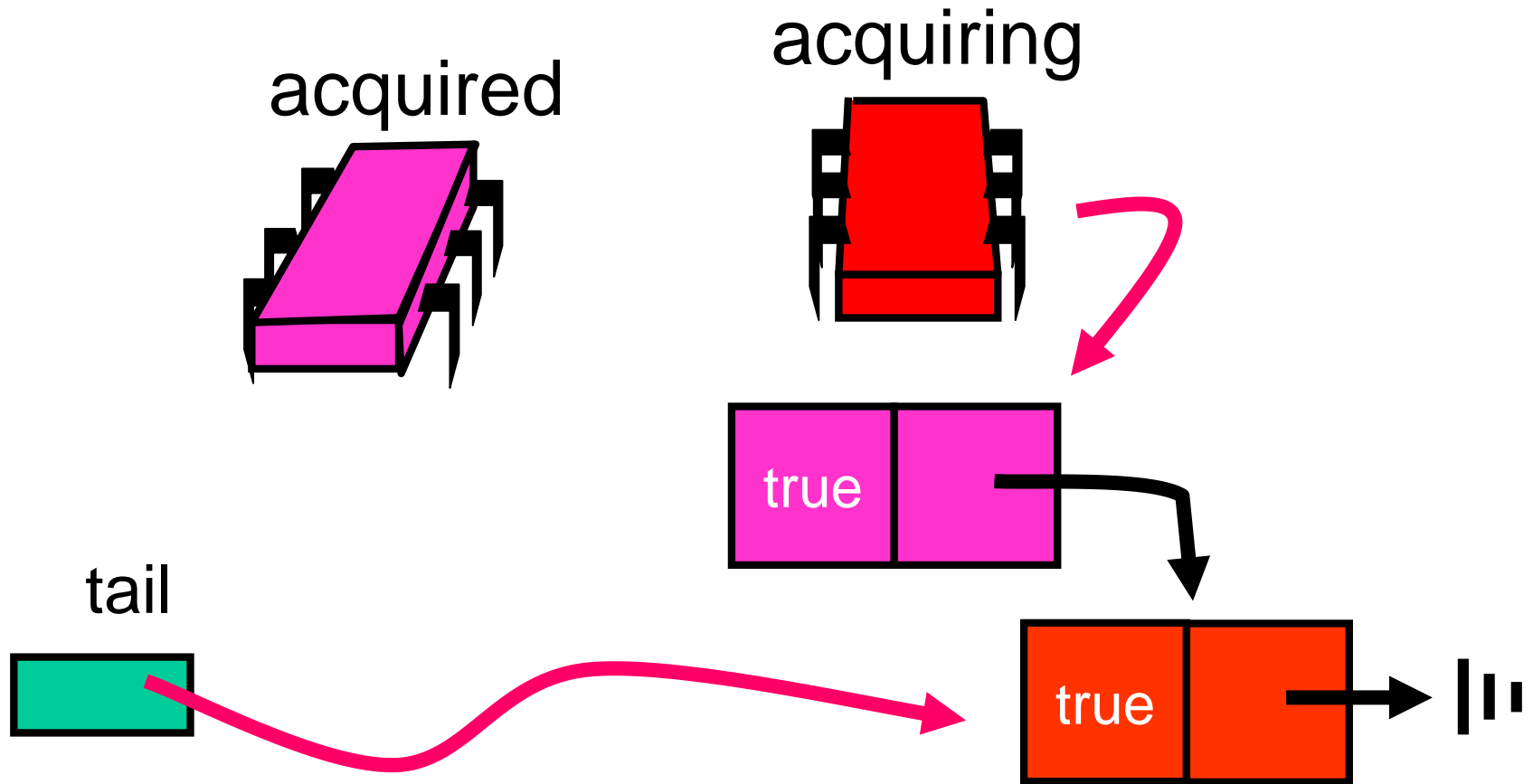
Acquiring



Acquiring



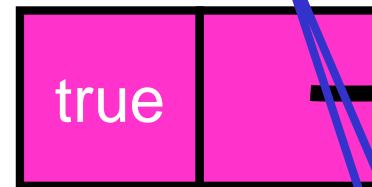
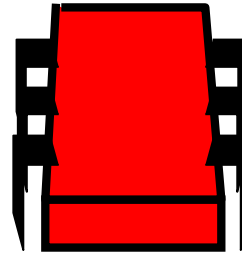
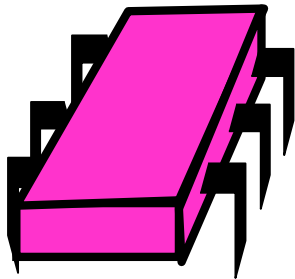
Acquiring



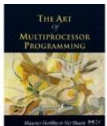
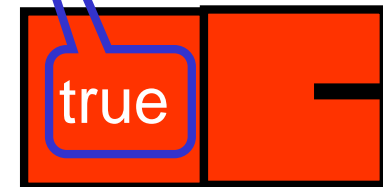
Acquiring

acquiring

acquired



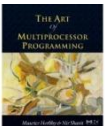
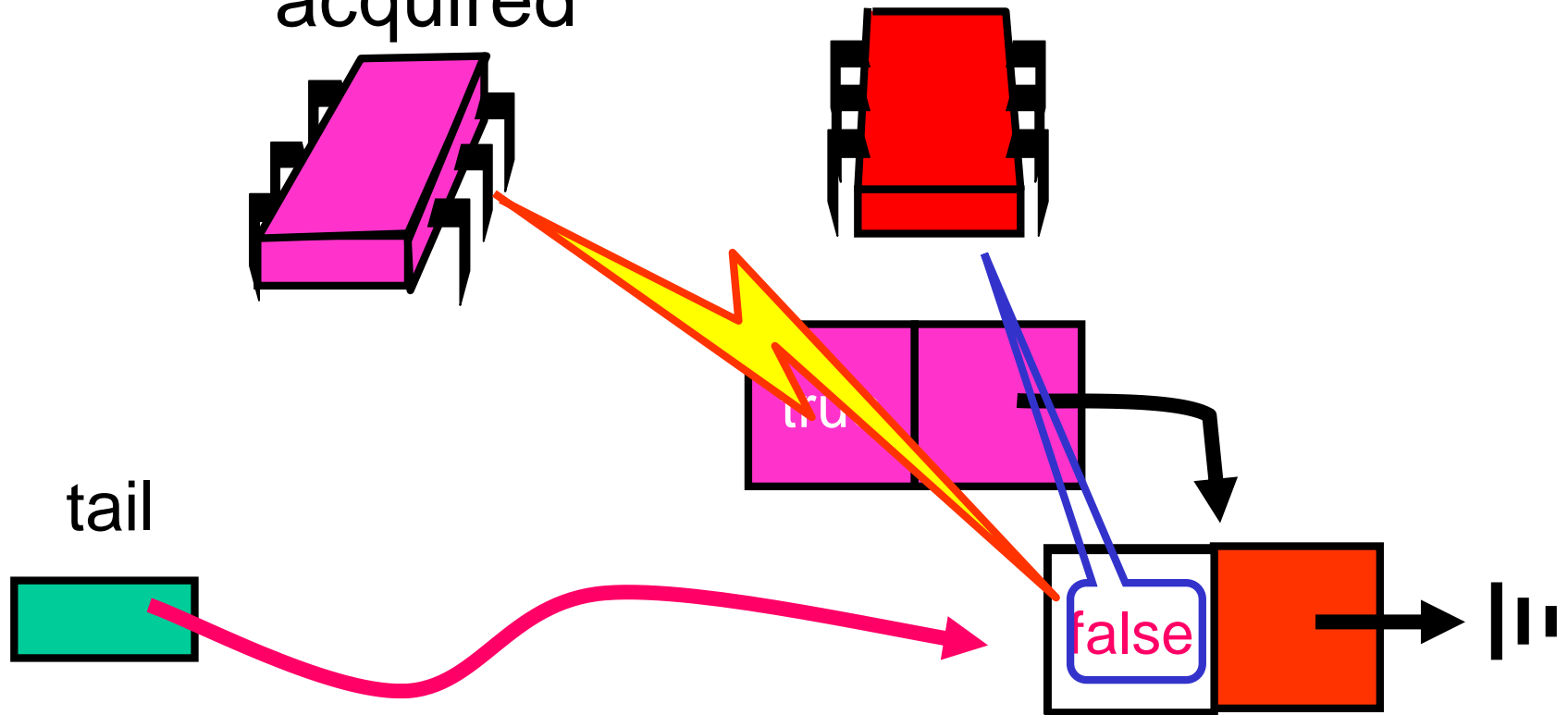
tail



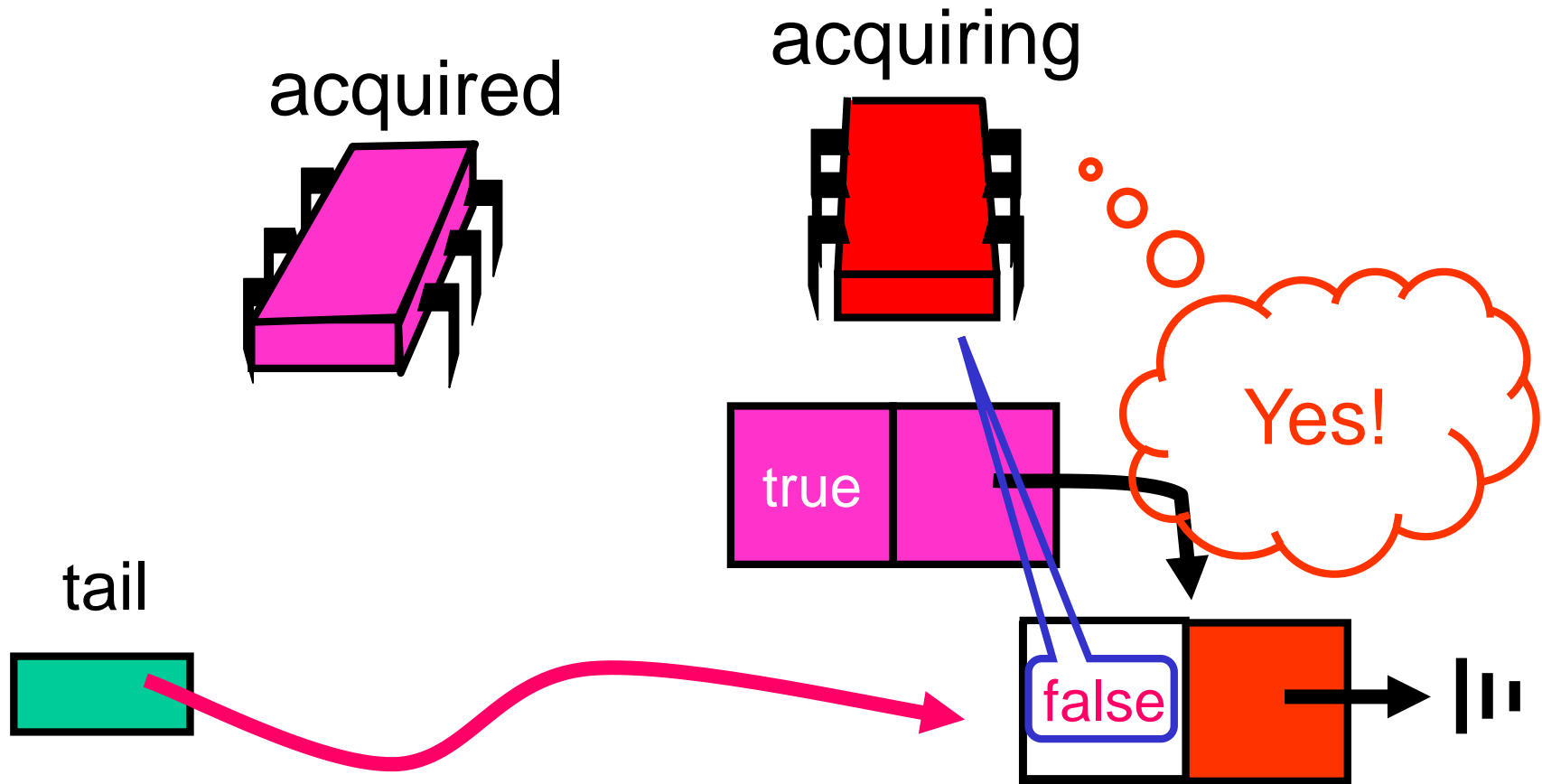
Acquiring

acquiring

acquired

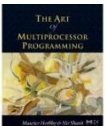


Acquiring



MCS Queue Lock

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



MCS Queue Lock

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



MCS Queue Lock

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

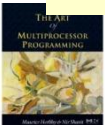
**Make a
QNode**



MCS Queue Lock

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

**add my Node to
the tail of
queue**



MCS Queue Lock

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

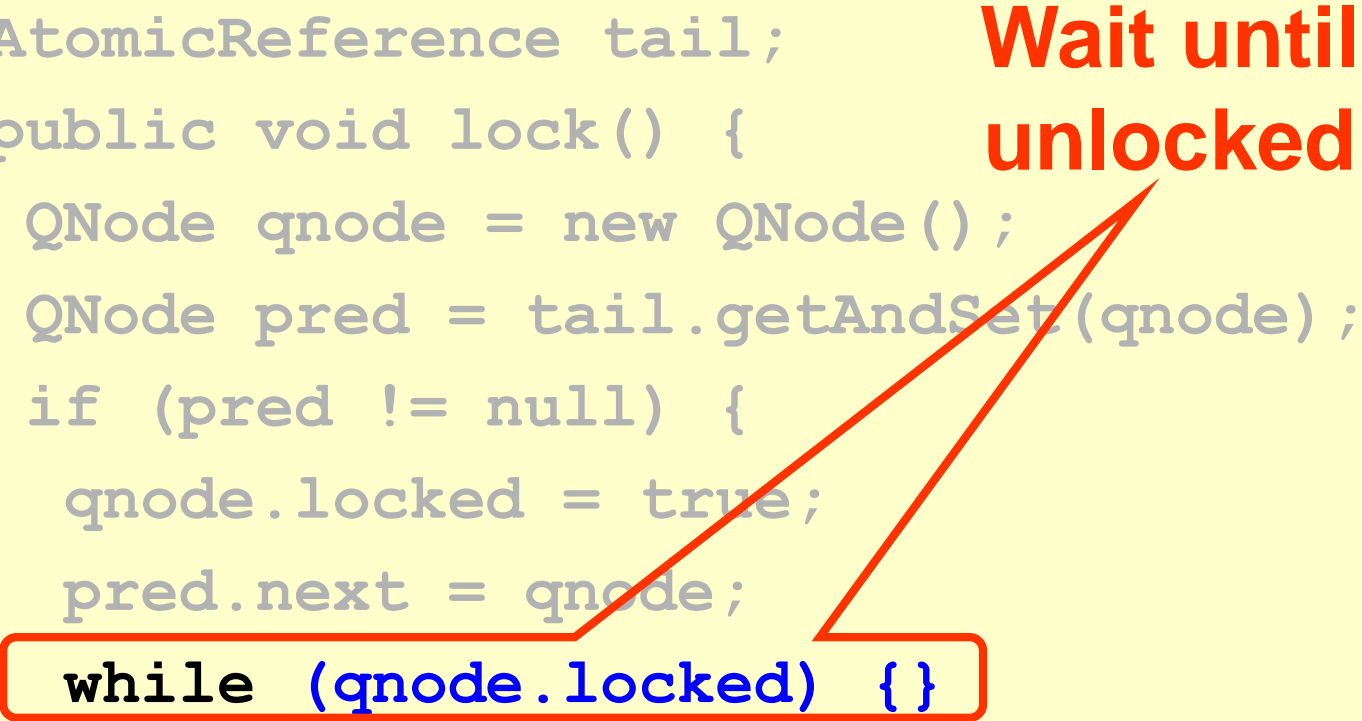
**Fix if queue was
non-empty**



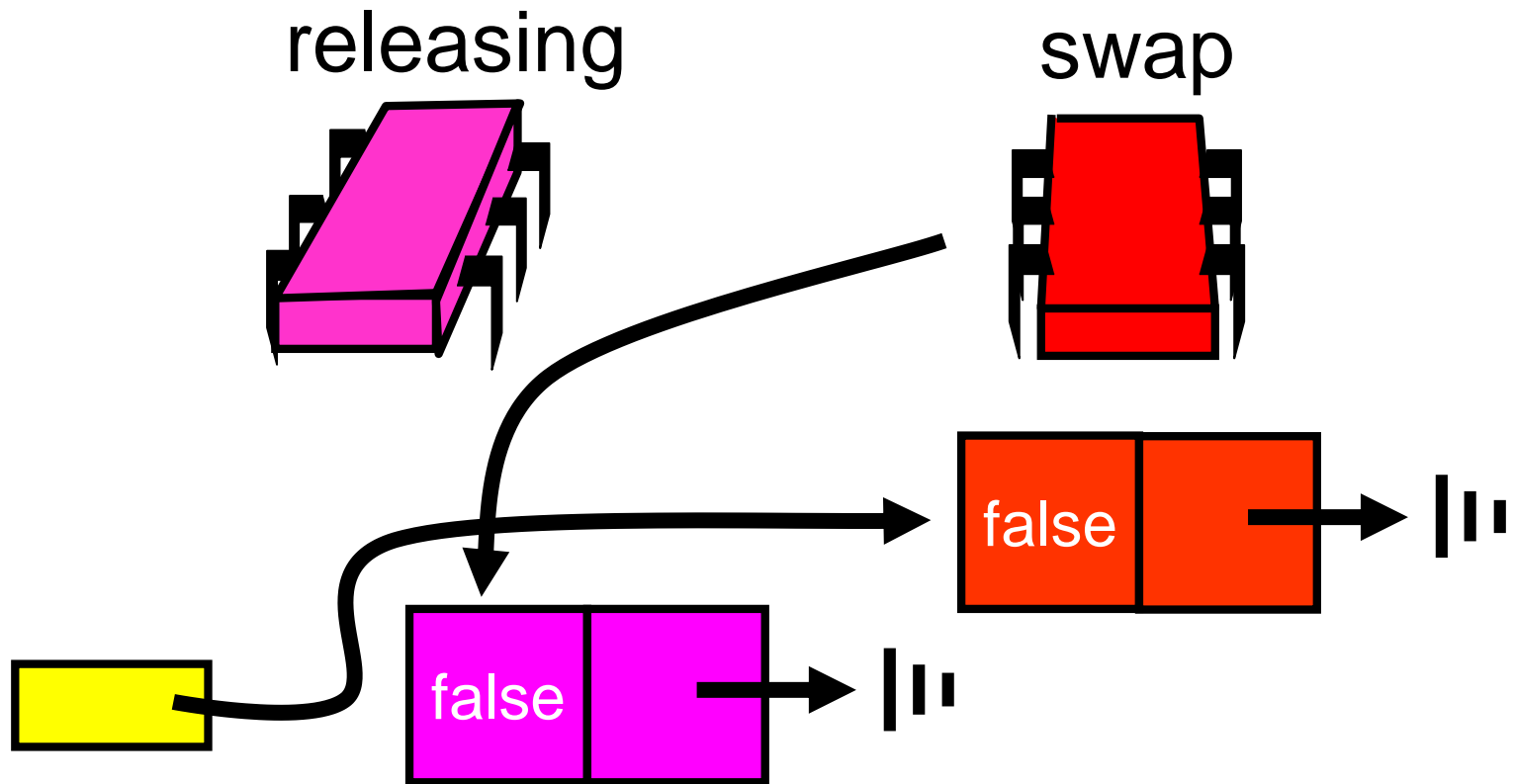
MCS Queue Lock

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

Wait until unlocked



Purple Release

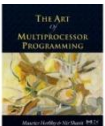
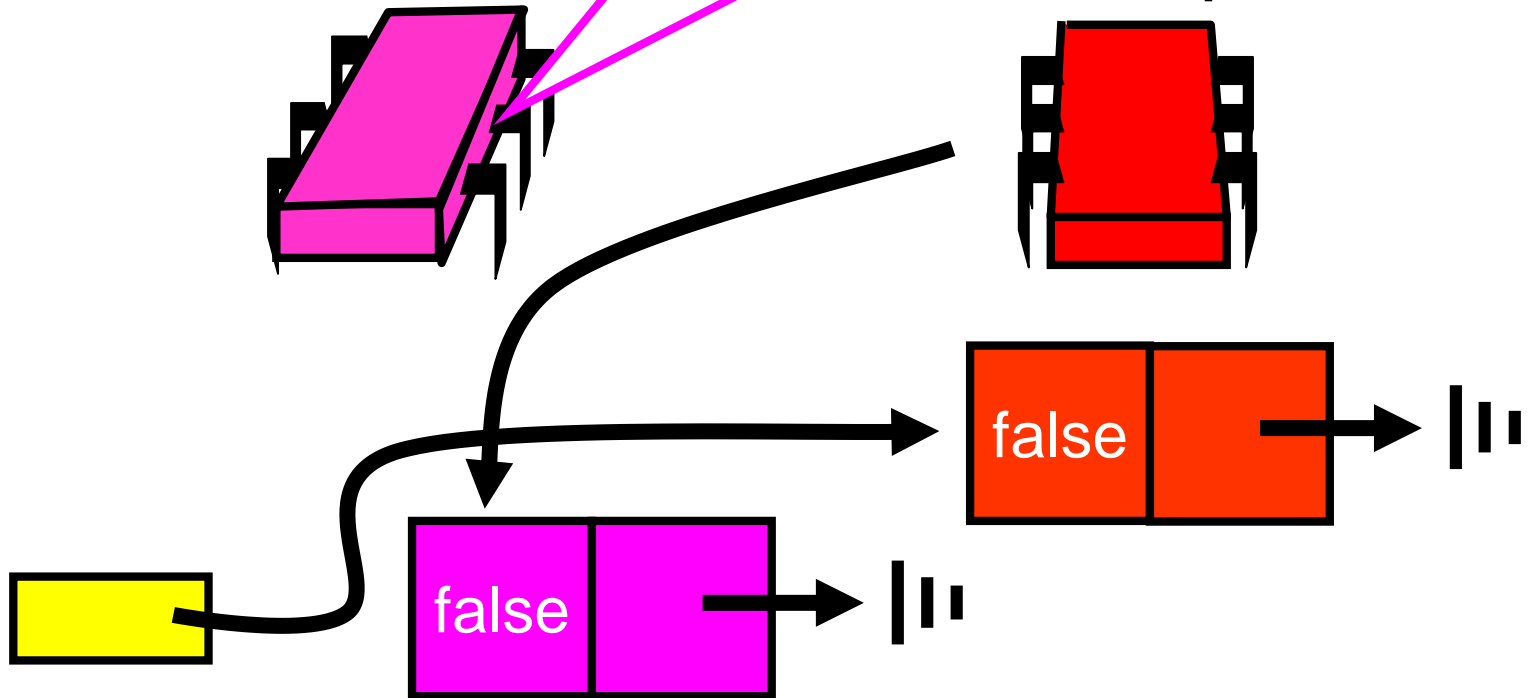


Pu

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

releasing

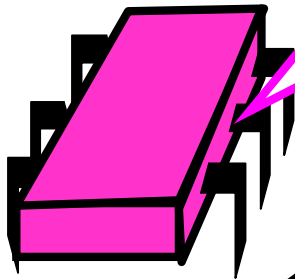
swap



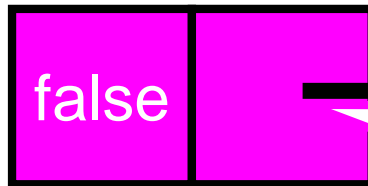
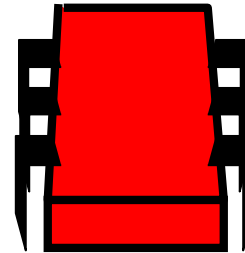
Pu

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

releasing



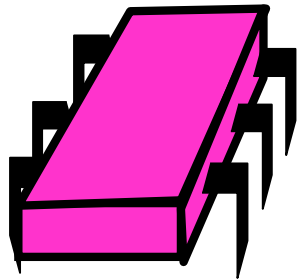
swap



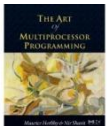
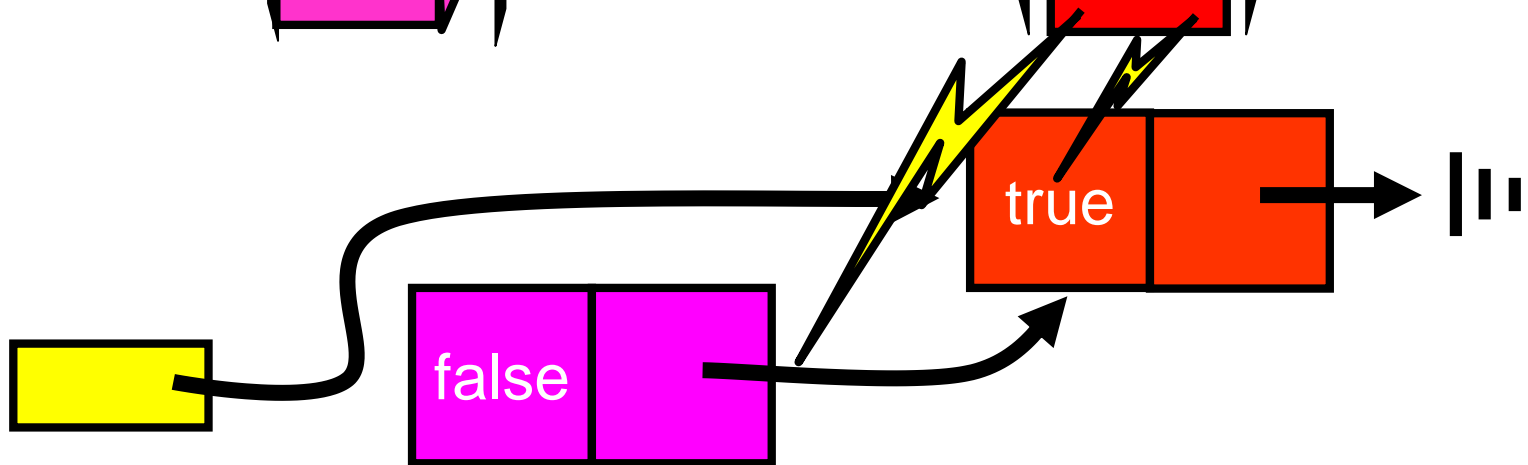
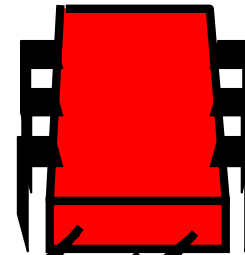
I have to release that thread so must wait for it to identify its node

Purple Release

releasing

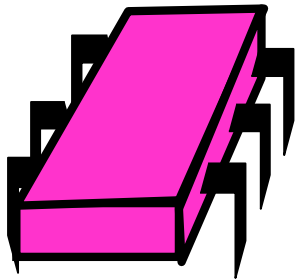


prepare to spin

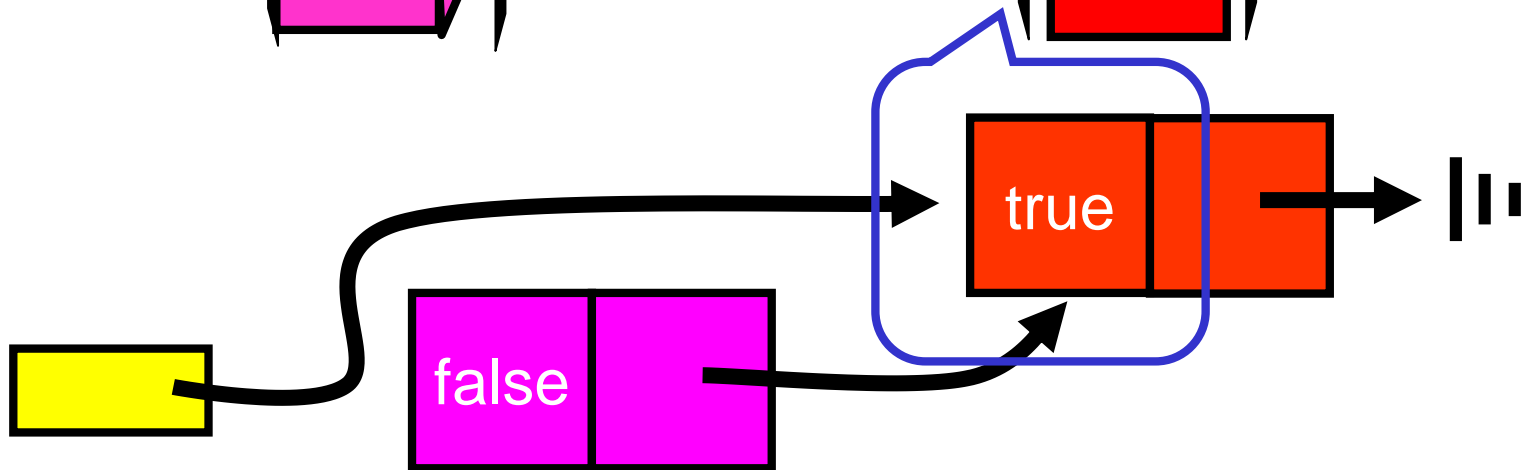
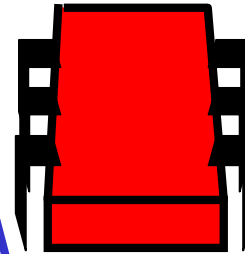


Purple Release

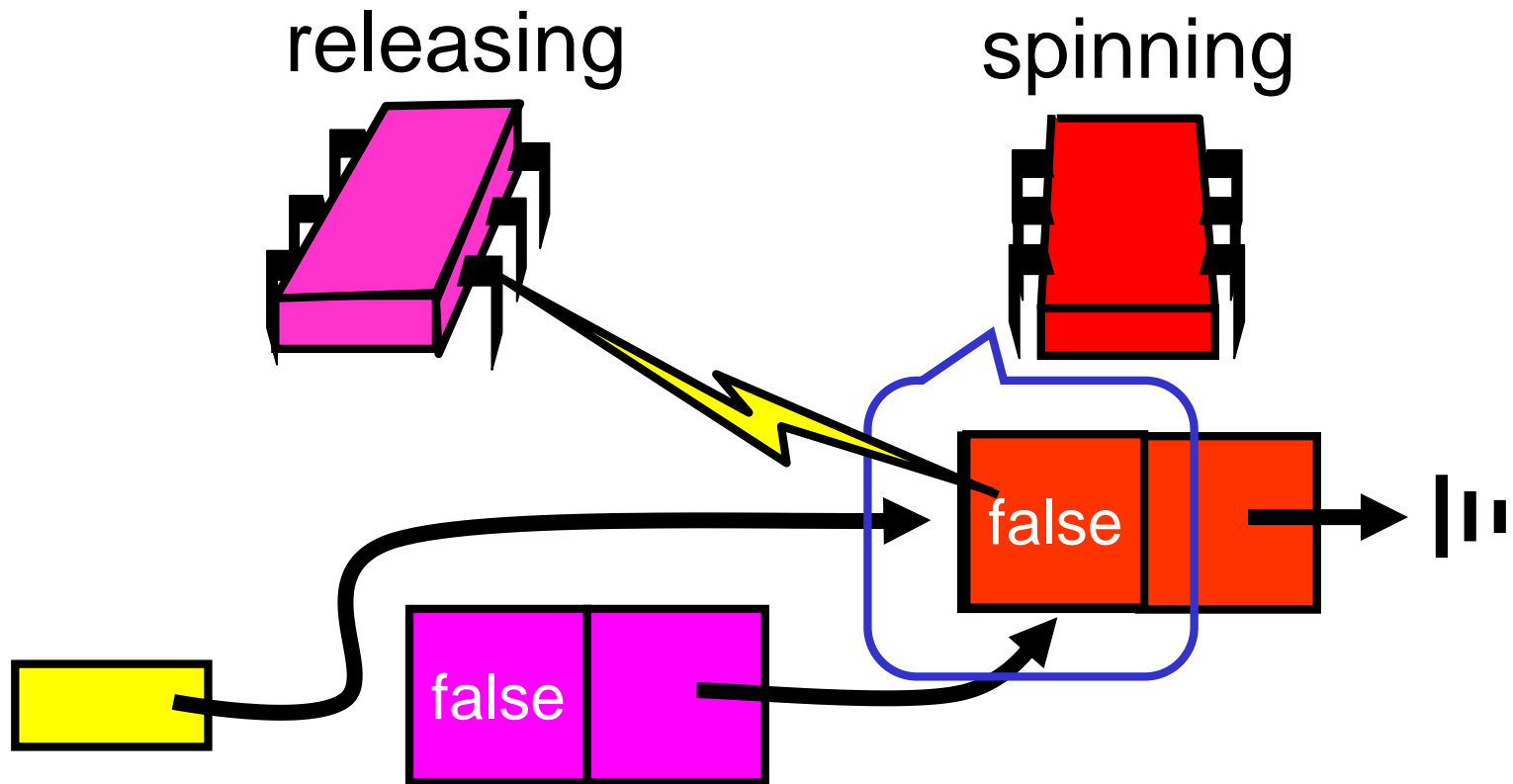
releasing



spinning

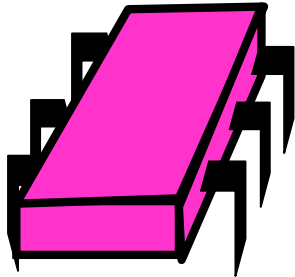


Purple Release

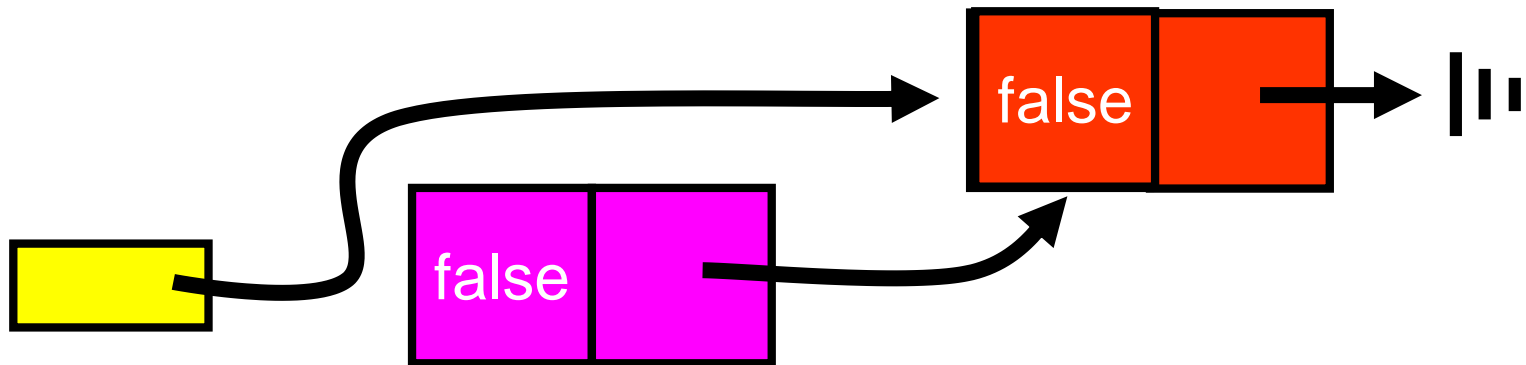
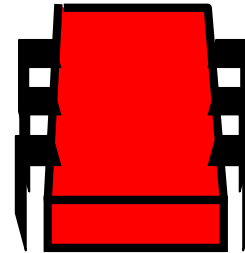


Purple Release

releasing



Acquired lock



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**

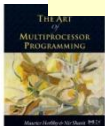
?



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

Pass lock to successor

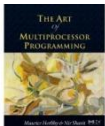


qnode.next.locked = false;



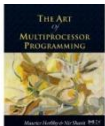
Abortable Locks

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



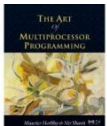
Back-off Lock

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

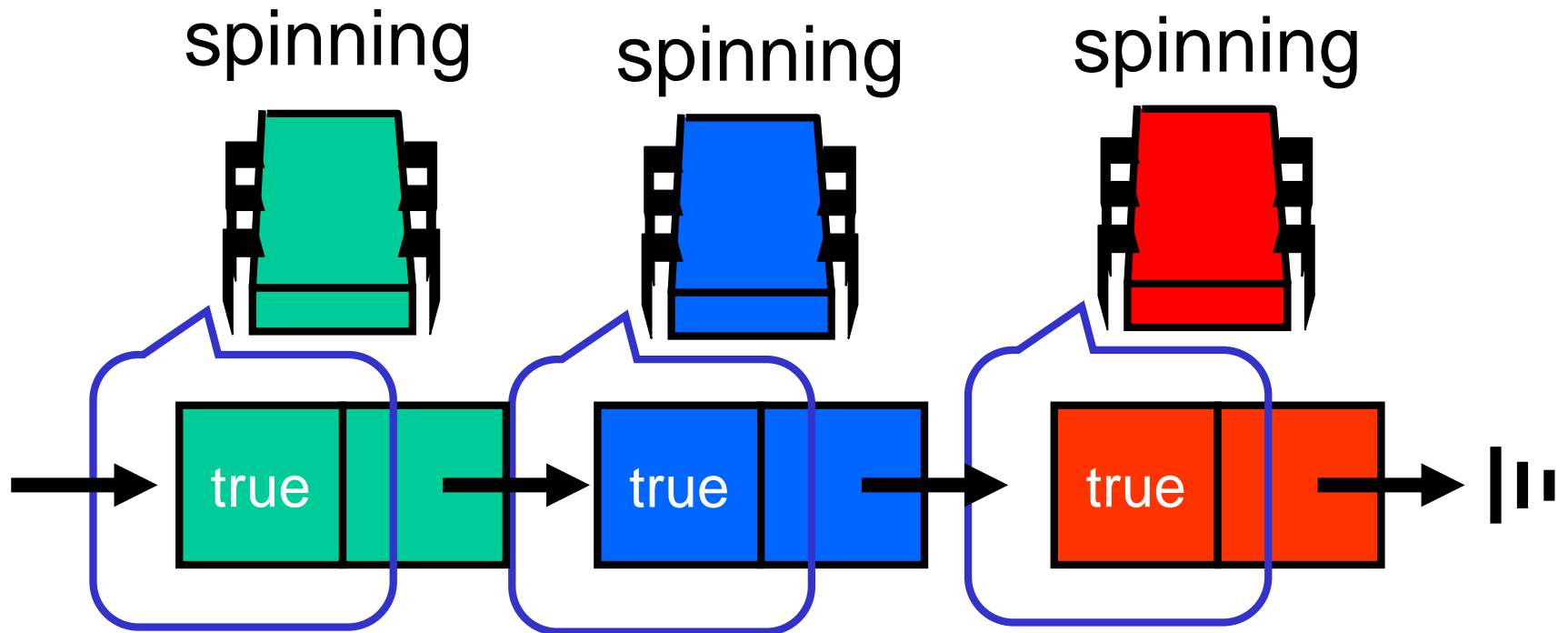


Queue Locks

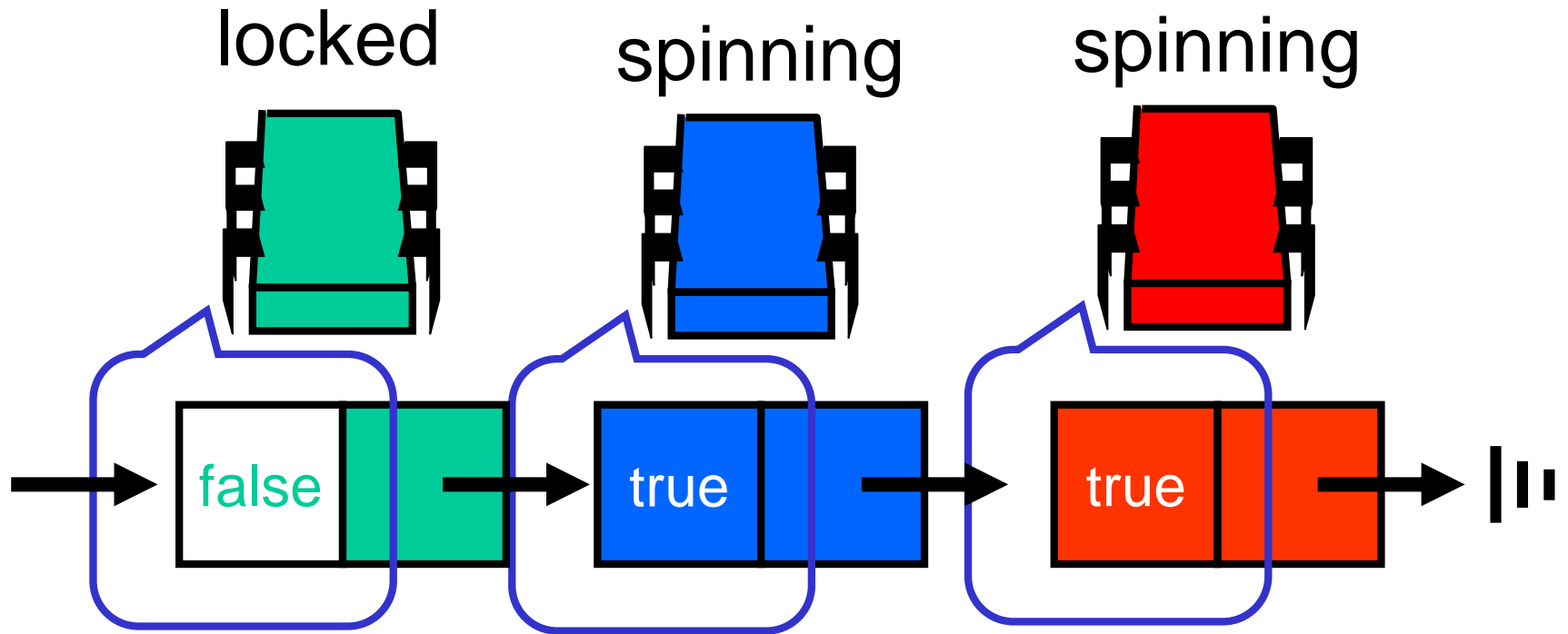
- Can't just quit
 - Thread in line behind will starve
- Need a graceful way out



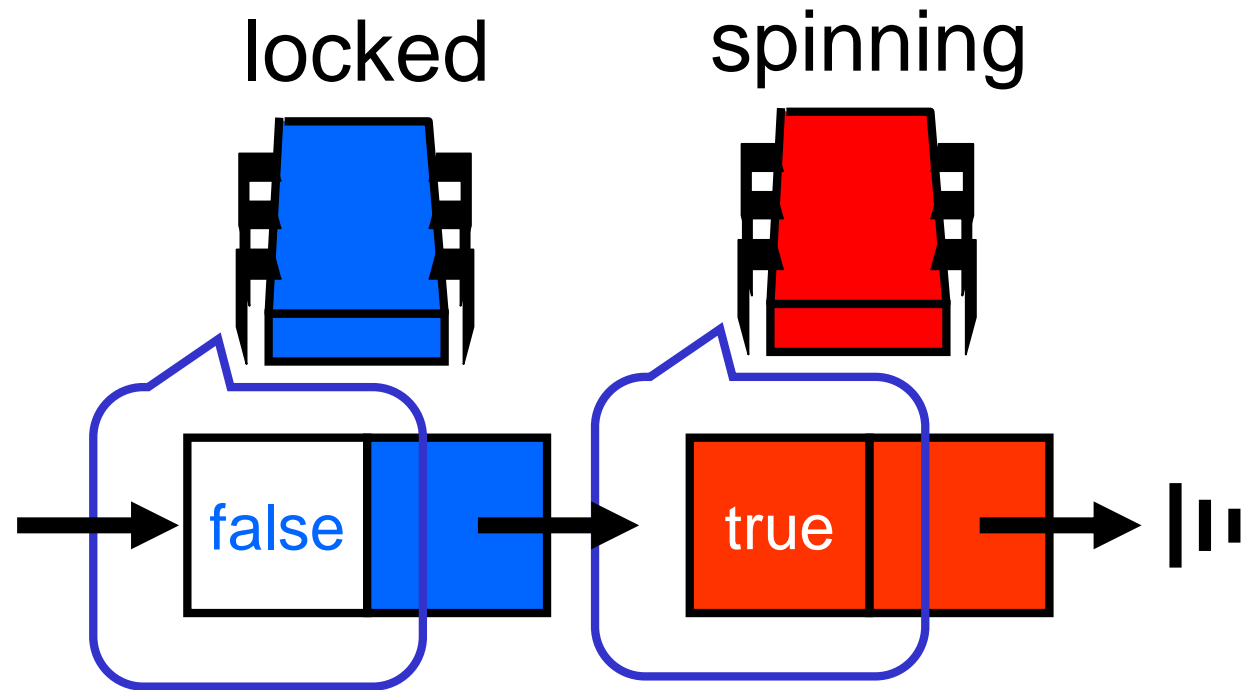
Queue Locks



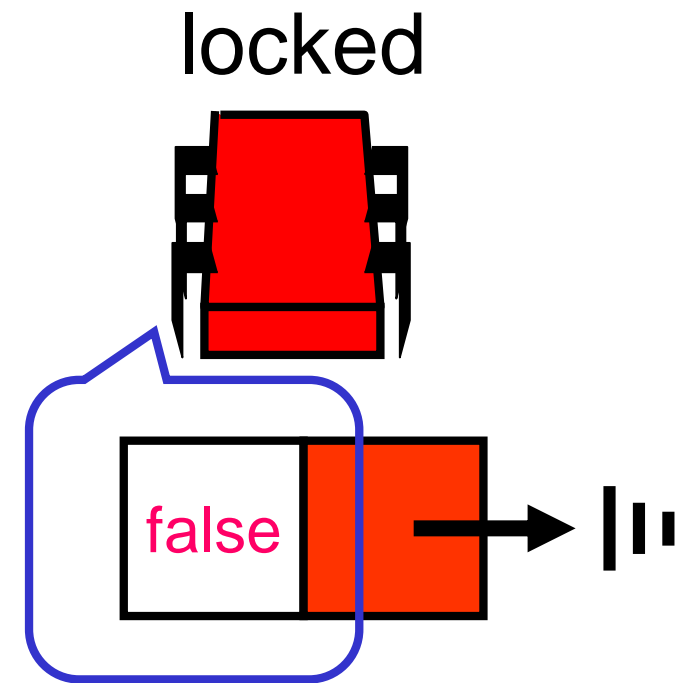
Queue Locks



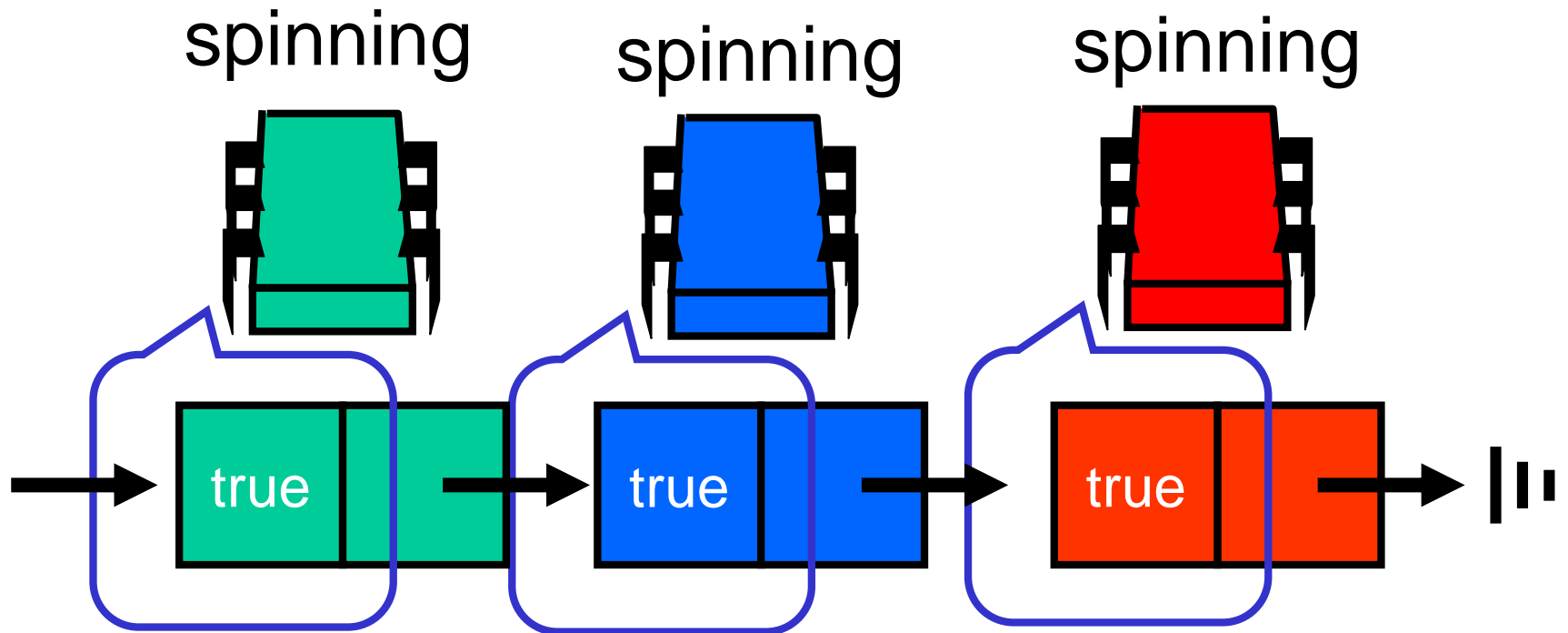
Queue Locks



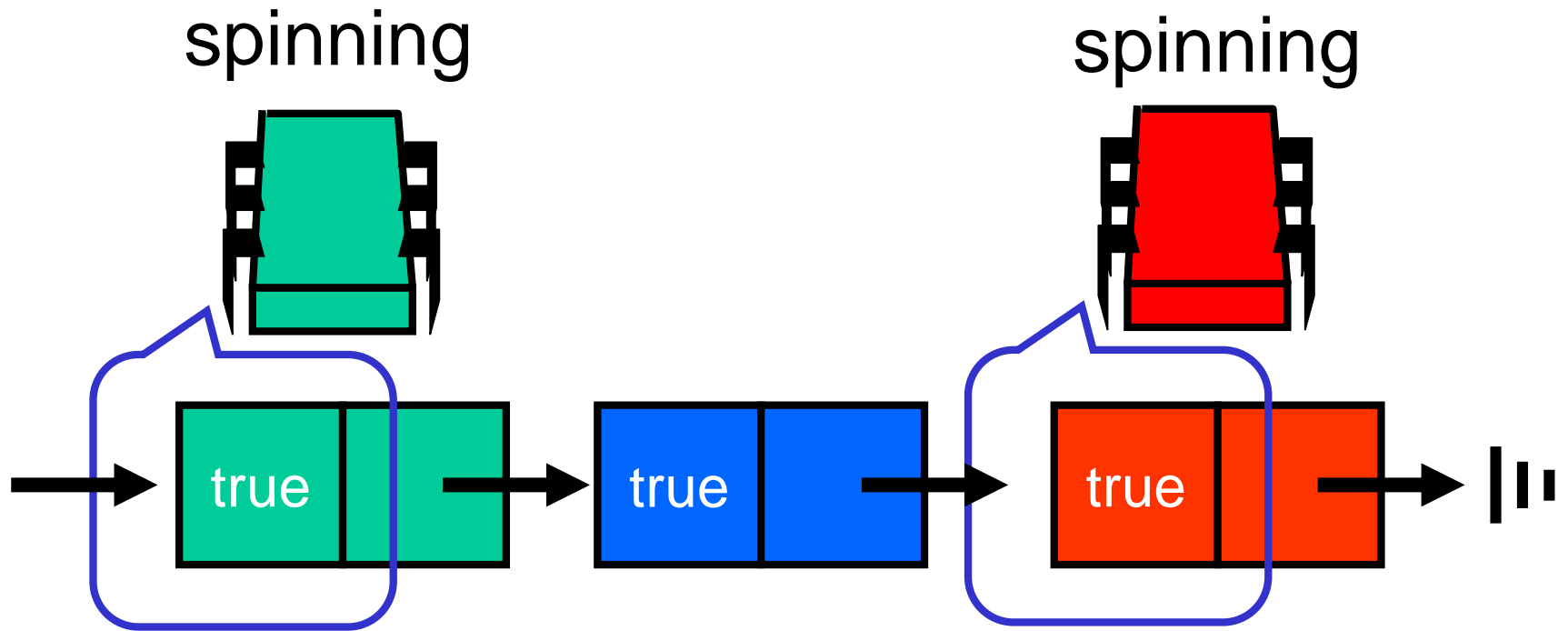
Queue Locks



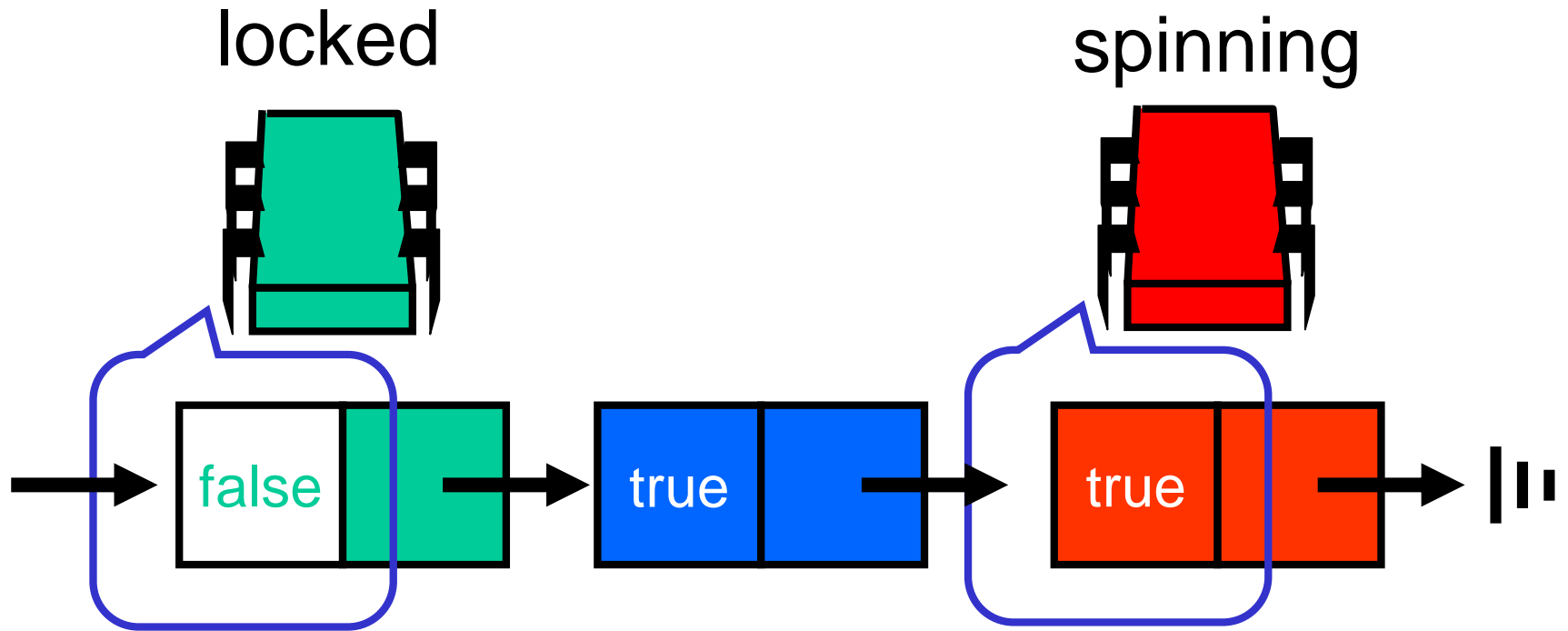
Queue Locks



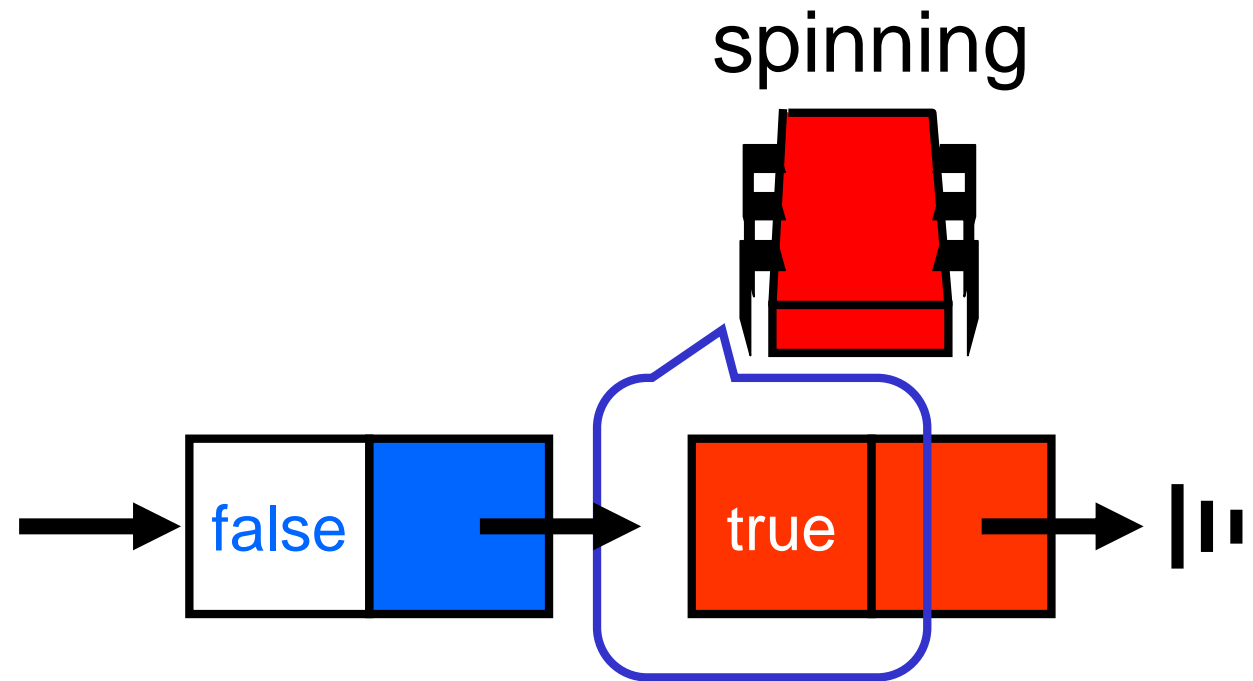
Queue Locks



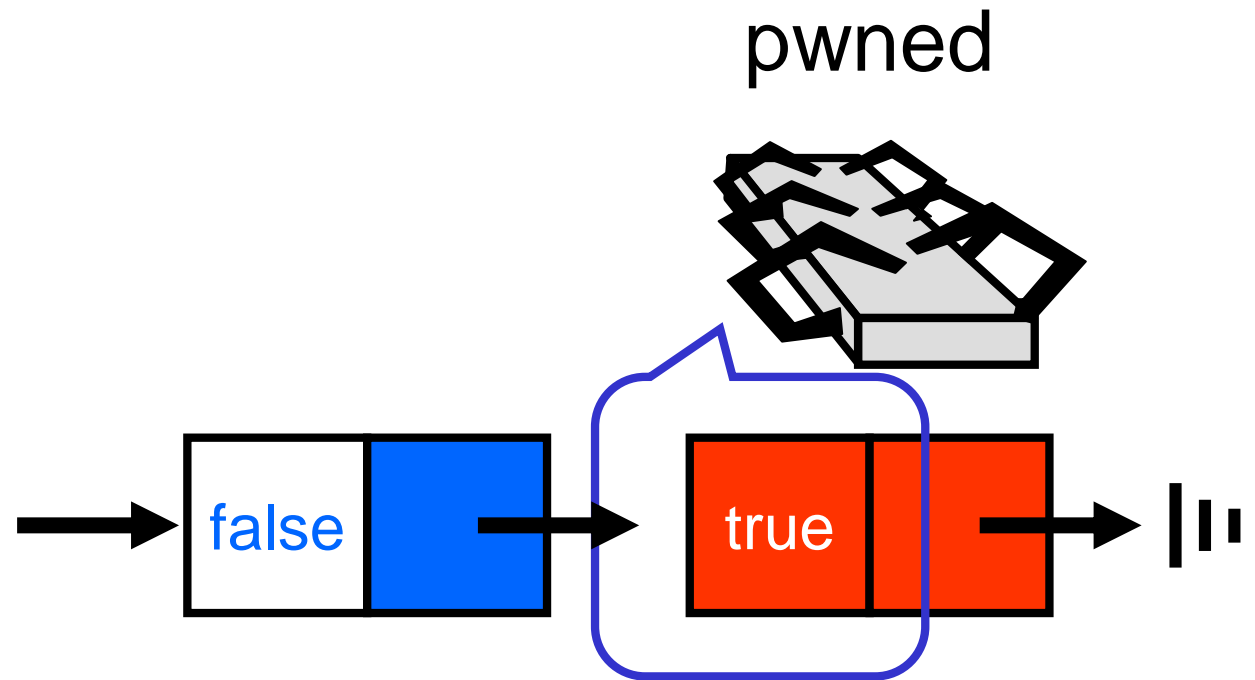
Queue Locks



Queue Locks

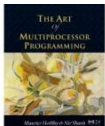


Queue Locks

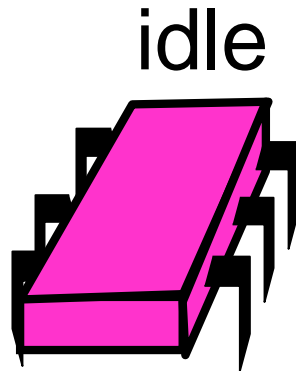


Abortable CLH Lock

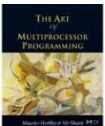
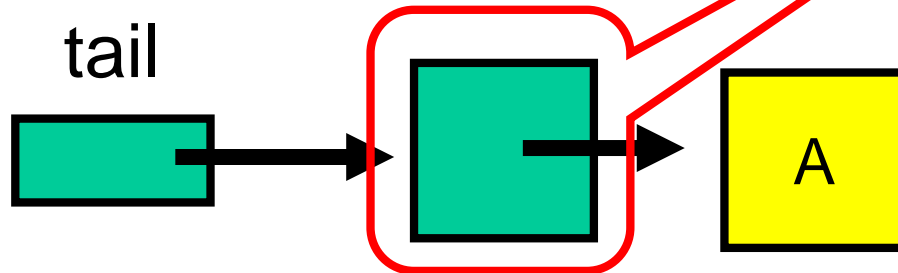
- When a thread gives up
 - Removing node in a wait-free way is hard
- Idea:
 - let successor deal with it.



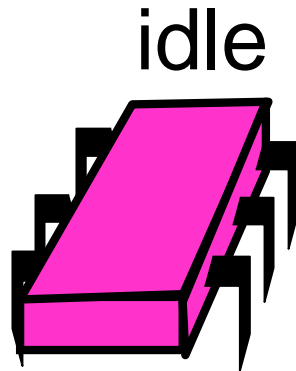
Initially



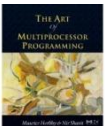
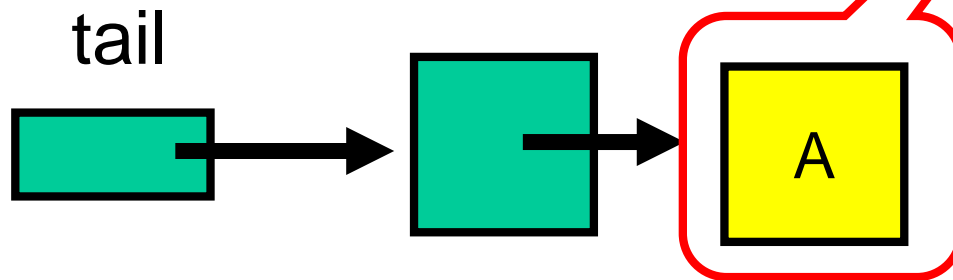
Pointer to
predecessor
(or null)



Initially

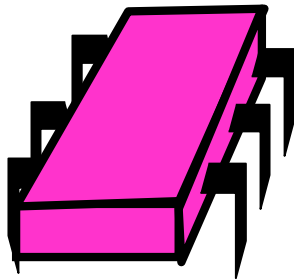


Distinguished
available node
means lock is
free

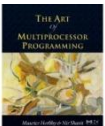
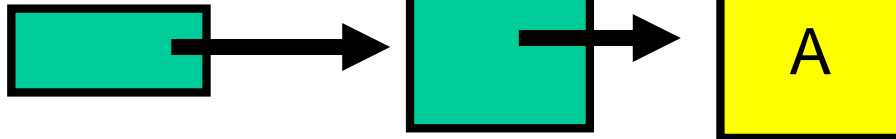


Acquiring

acquiring



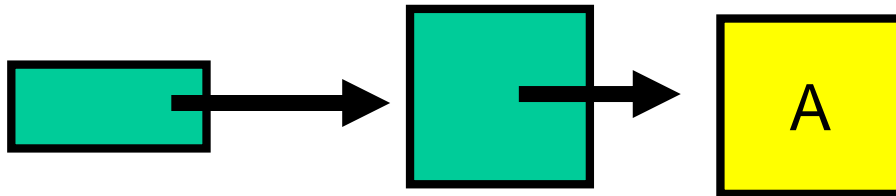
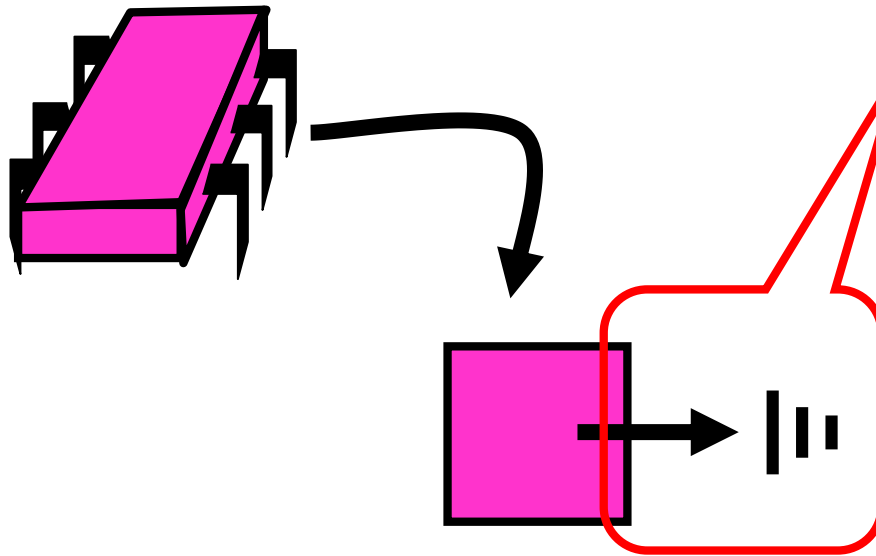
tail



Acquiring

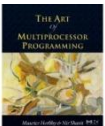
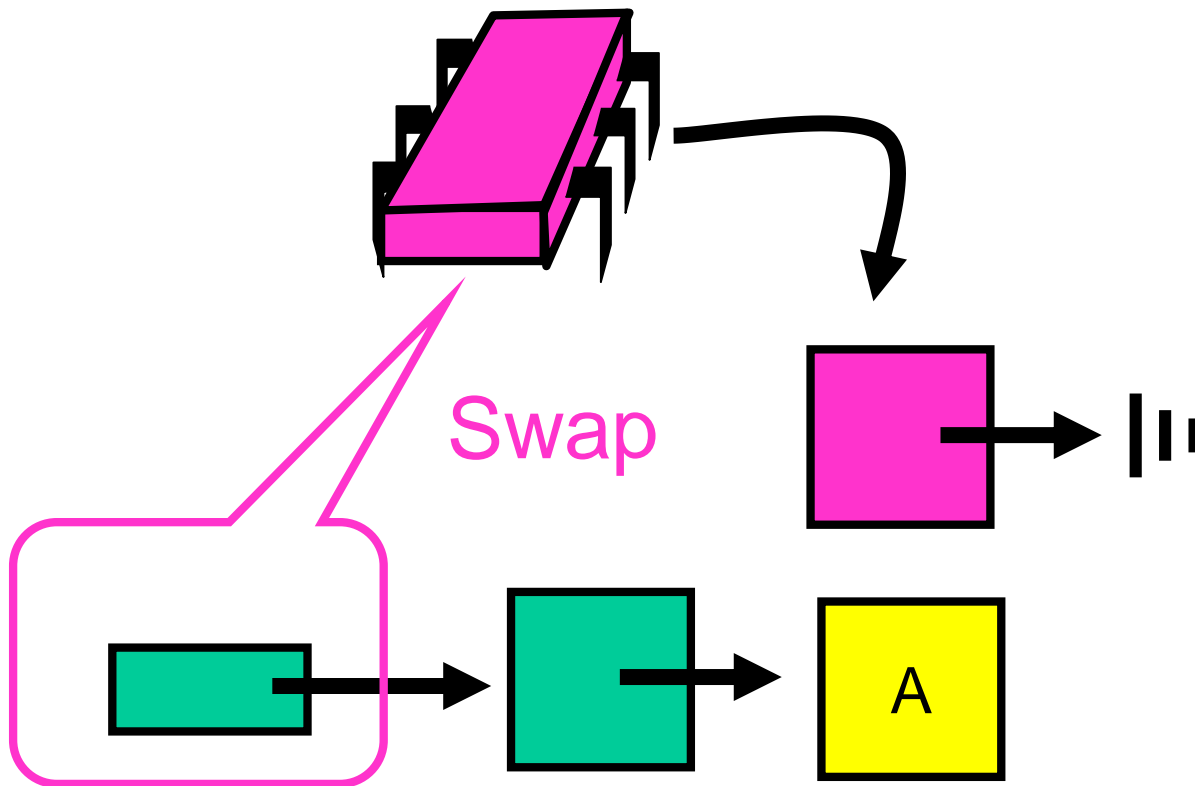
Null predecessor
means lock not
released or
aborted

acquiring



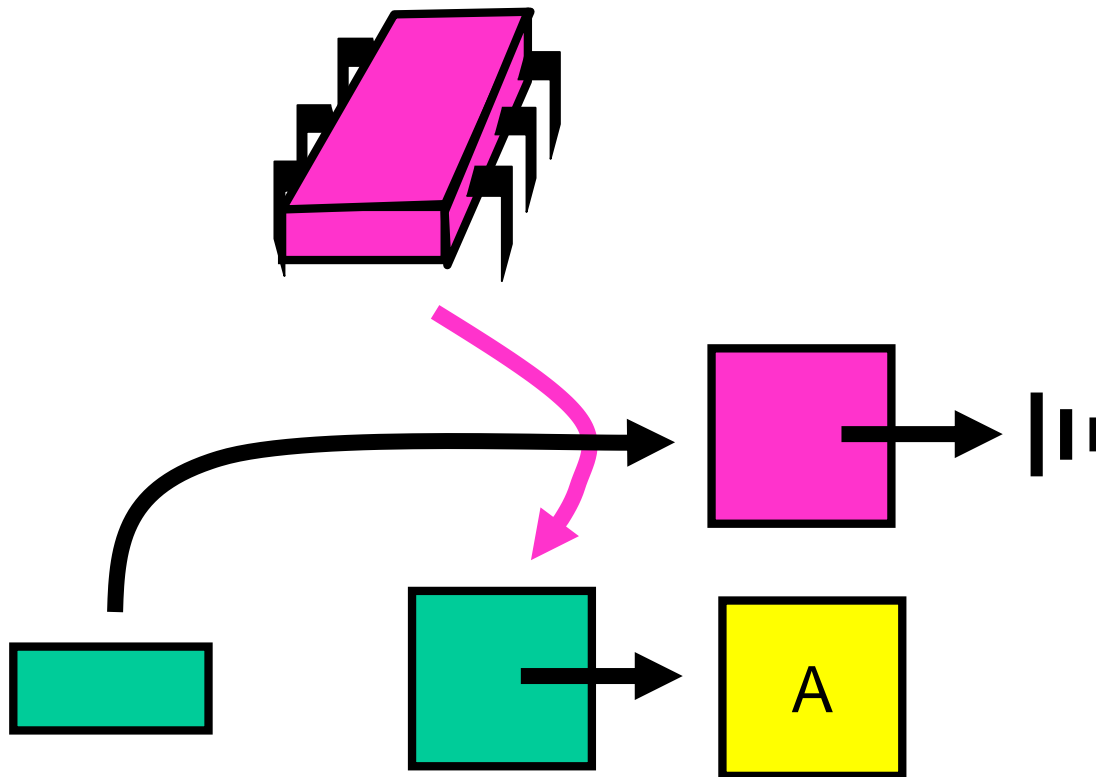
Acquiring

acquiring



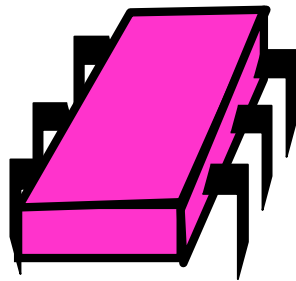
Acquiring

acquiring

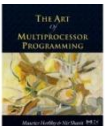
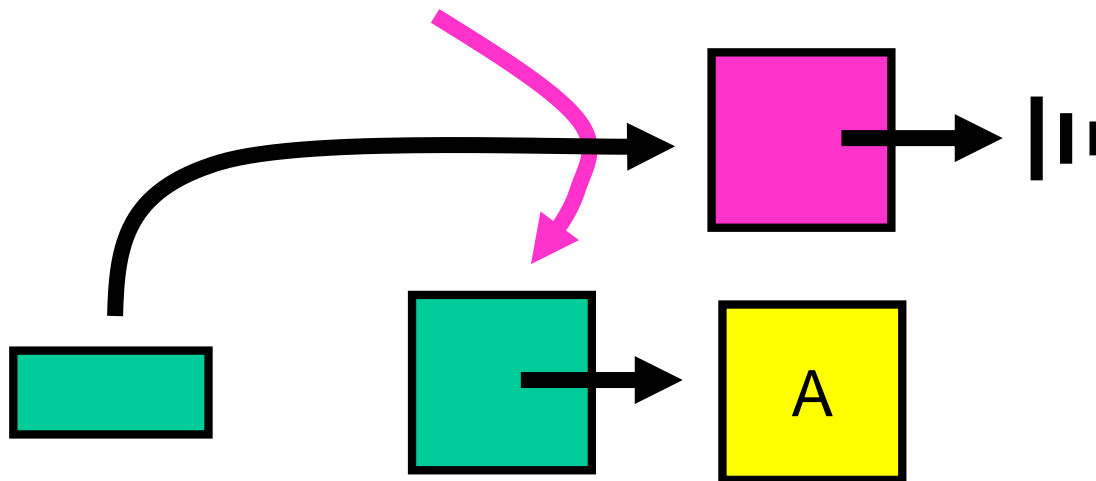


Acquired

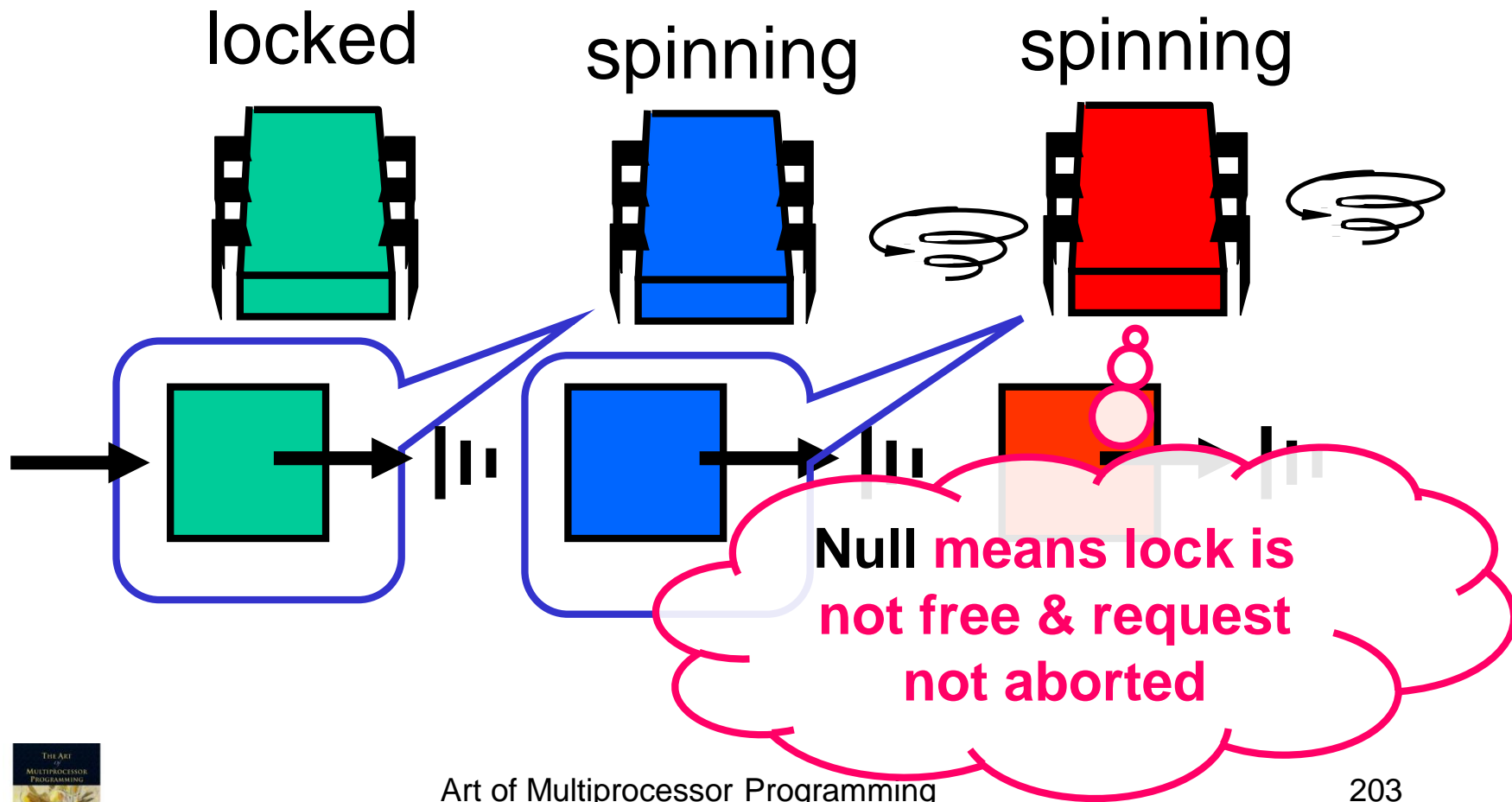
locked



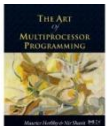
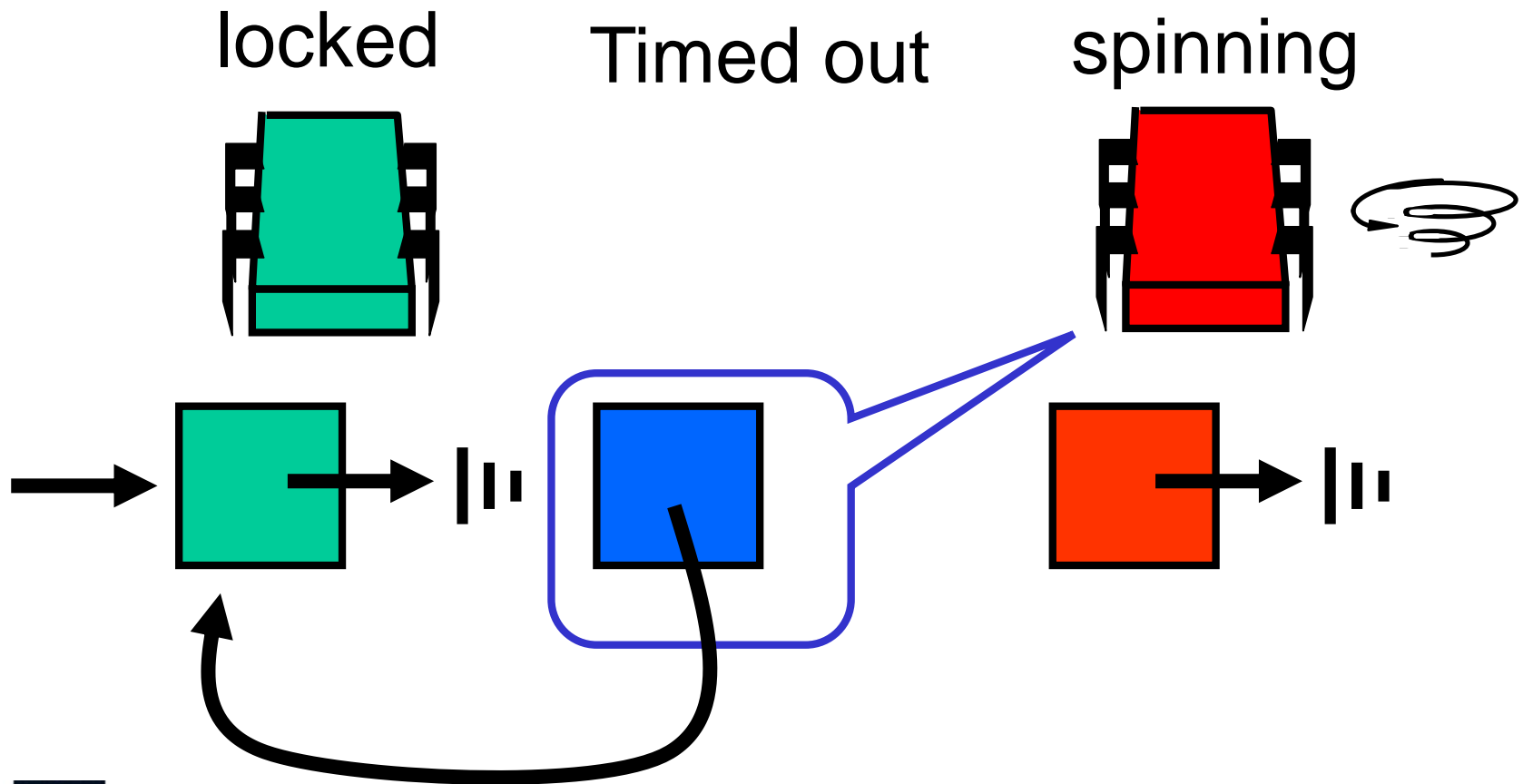
Reference to
AVAILABLE means
lock is free.



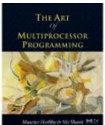
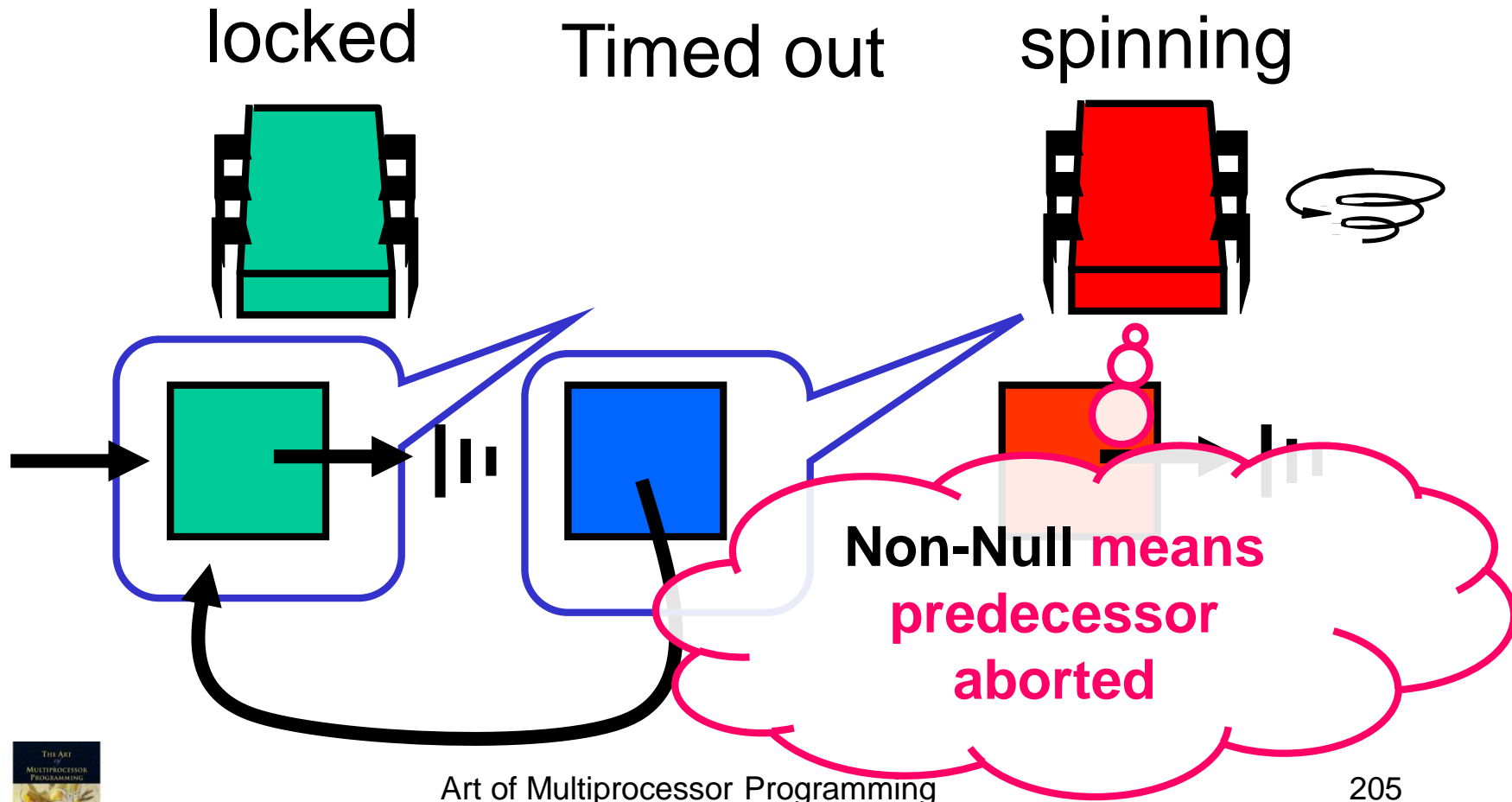
Normal Case



One Thread Aborts

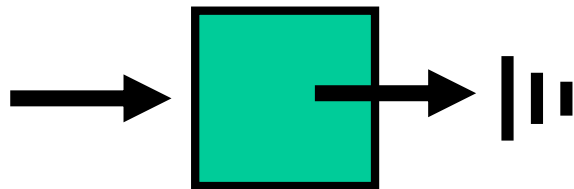
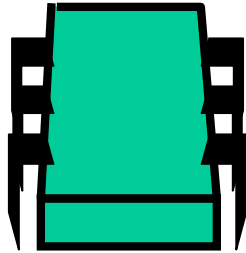


Successor Notices

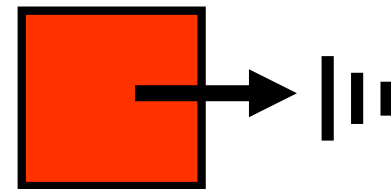
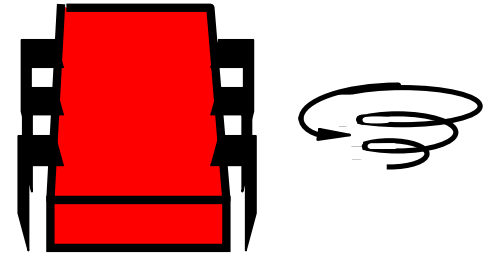


Recycle Predecessor's Node

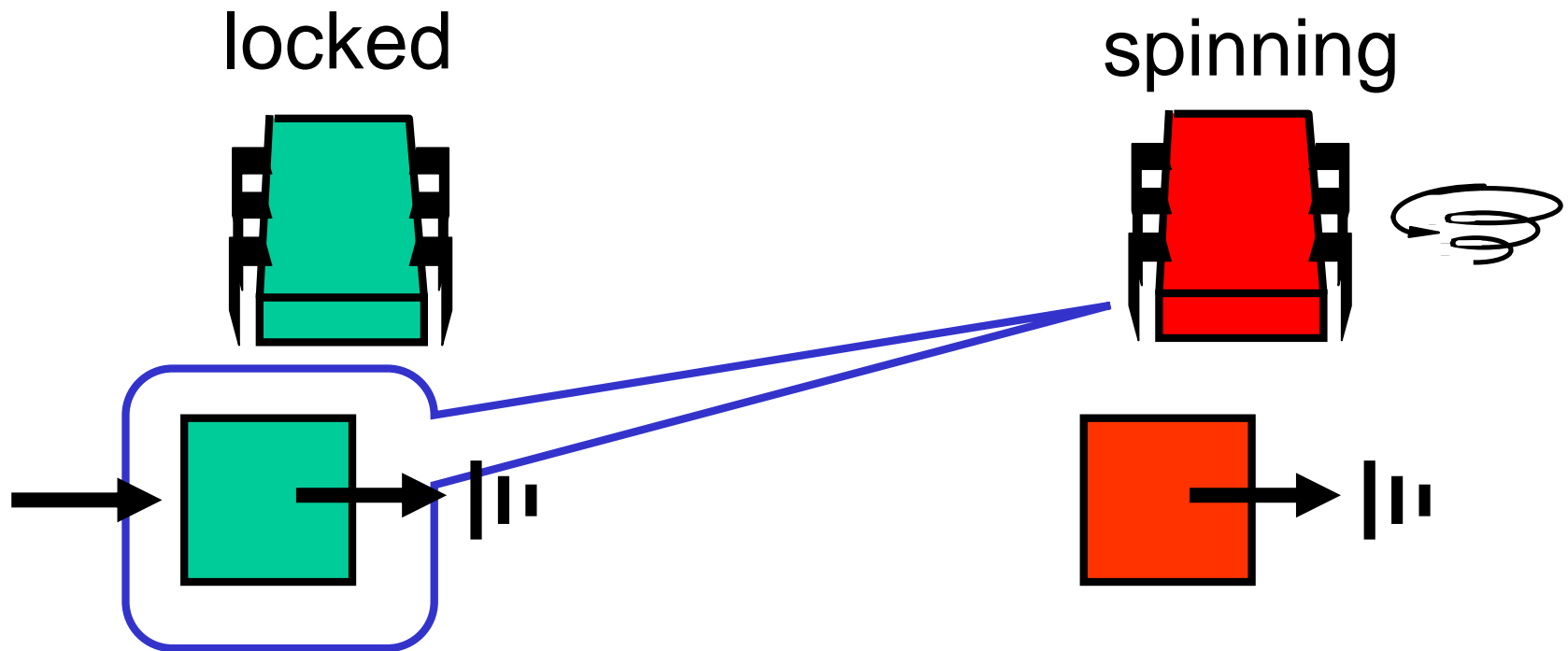
locked



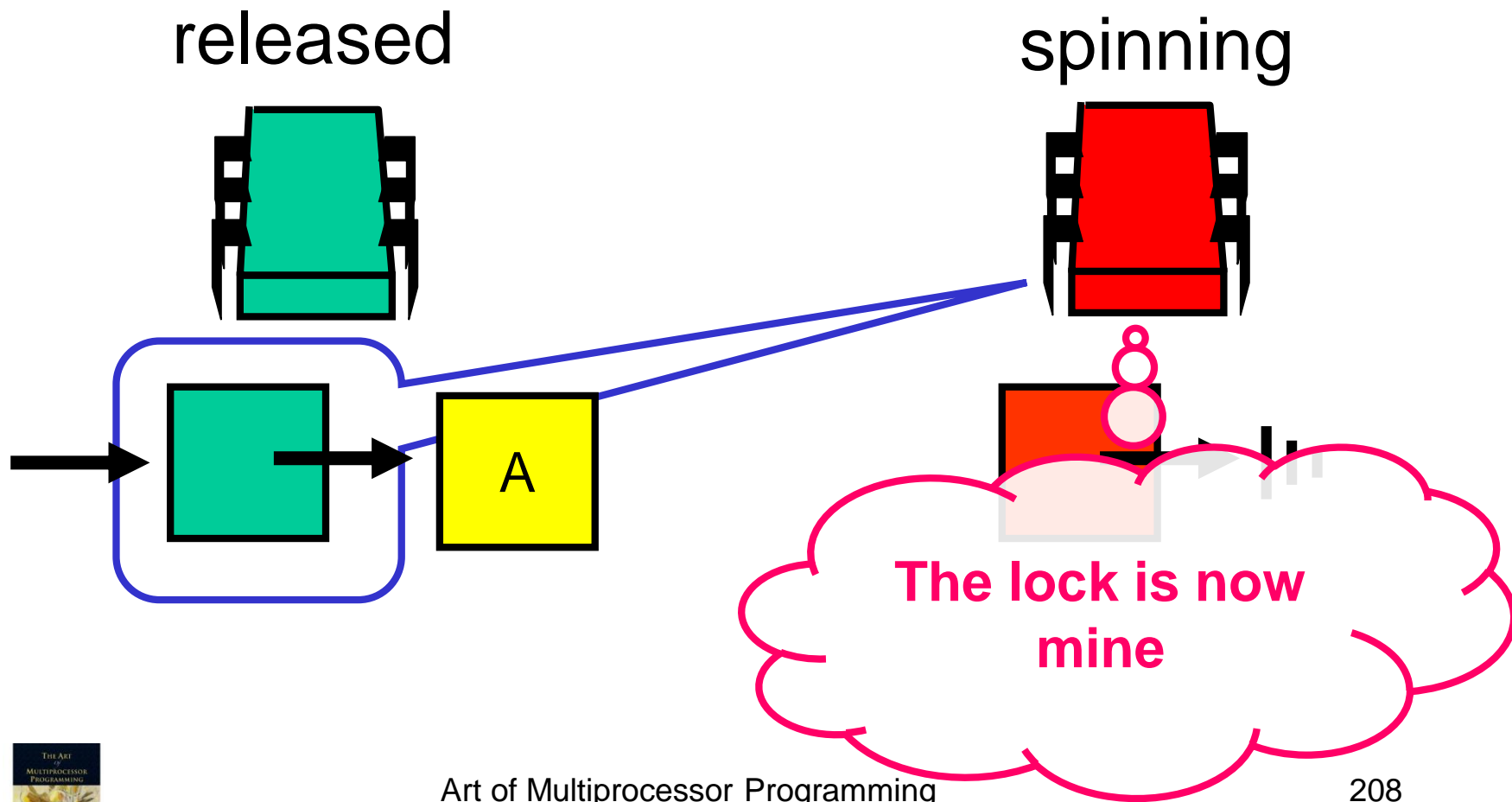
spinning



Spin on Earlier Node

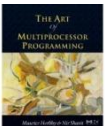


Spin on Earlier Node



Time-out Lock

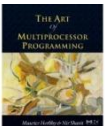
```
public class TOLock implements Lock {  
    static QNode AVAILABLE  
        = new QNode();  
    AtomicReference<QNode> tail;  
    ThreadLocal<QNode> myNode;
```



Time-out Lock

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

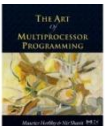
**AVAILABLE node
signifies free lock**



Time-out Lock

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

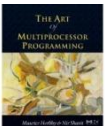
Tail of the queue



Time-out Lock

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

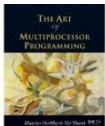
Remember my node ...



Time-out Lock

```
public boolean lock(long timeout) {  
    QNode qnode = new QNode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    QNode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

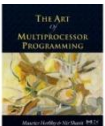
...



Time-out Lock

```
public boolean lock(long timeout) {  
    QNode qnode = new QNode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    QNode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

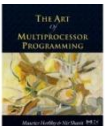
Create & initialize node



Time-out Lock

```
public boolean lock(long timeout) {  
    QNode qnode = new QNode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    QNode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

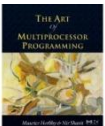
Swap with tail



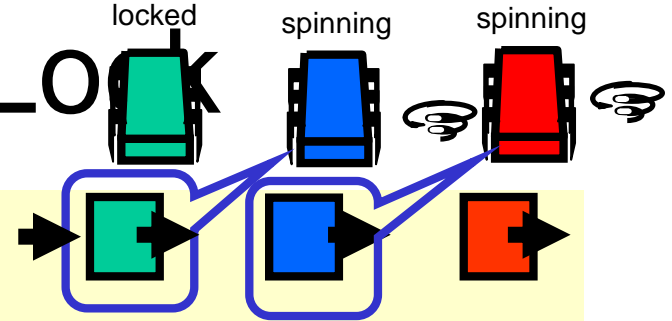
Time-out Lock

```
public boolean lock(long timeout) {  
    QNode qnode = new QNode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    QNode myPred = tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
    ...  
}
```

If predecessor absent or released, we are done



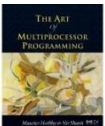
Time-out Lock



...

```
long start = now();  
while (now() - start < timeout) {  
    QNode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}
```

...



Time-out Lock

...

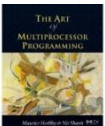
```
long start = now();  
while (now() - start < timeout) {
```

```
    QNode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}
```

...

...

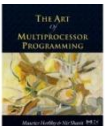
Keep trying for a while



Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    QNode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...
```

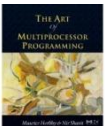
**Spin on predecessor's
prev field**



Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    QNode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...
```

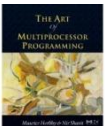
Predecessor released lock



Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    QNode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...
```

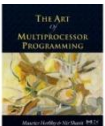
**Predecessor aborted,
advance one**



Time-out Lock

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

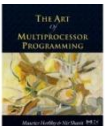
What do I do when I time out?



Time-out Lock

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

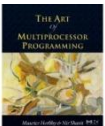
**Do I have a successor?
If CAS fails, I do.
Tell it about myPred**



Time-out Lock

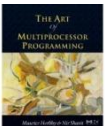
```
...  
if (!tail.compareAndSet(qnode, myPred))  
    qnode.prev = myPred;  
    return false;  
}  
}
```

**If CAS succeeds: no
successor, simply return false**



Time-Out Unlock

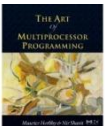
```
public void unlock() {  
    QNode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode, null))  
        qnode.prev = AVAILABLE;  
}
```



Time-out Unlock

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

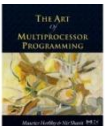
**If CAS failed:
successor exists,
notify it can enter**



Timing-out Lock

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

CAS successful: set tail to null, no clean up since no successor waiting



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/3.0/).

- **You are free:**
 - **to Share** — to copy, distribute and transmit the work
 - **to Remix** — to adapt the work
- **Under the following conditions:**
 - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
 - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
 - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.