

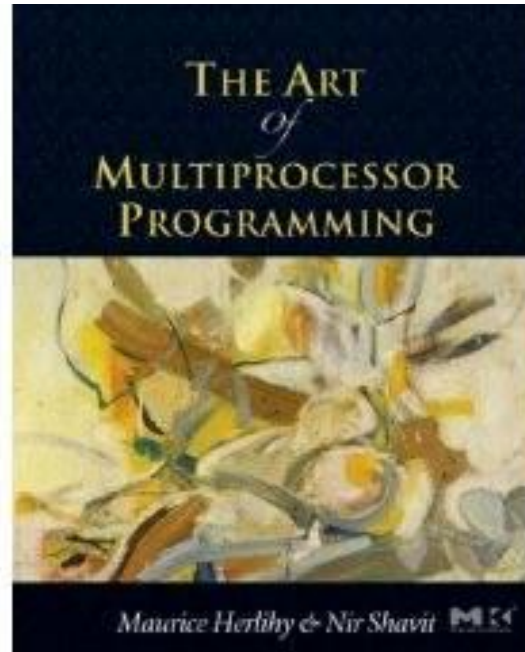


COS 226

Chapter 7

Spin Locks and Contention

Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



Concurrency issues

- Memory contention:
 - Not all processors can access the same memory at the same time and if they try they will have to queue
- Contention for communication medium:
 - If everyone wants to communicate at the same time, some of them will have to wait
- Communication latency:
 - If takes more time for a processor to communicate with memory or with another processor.



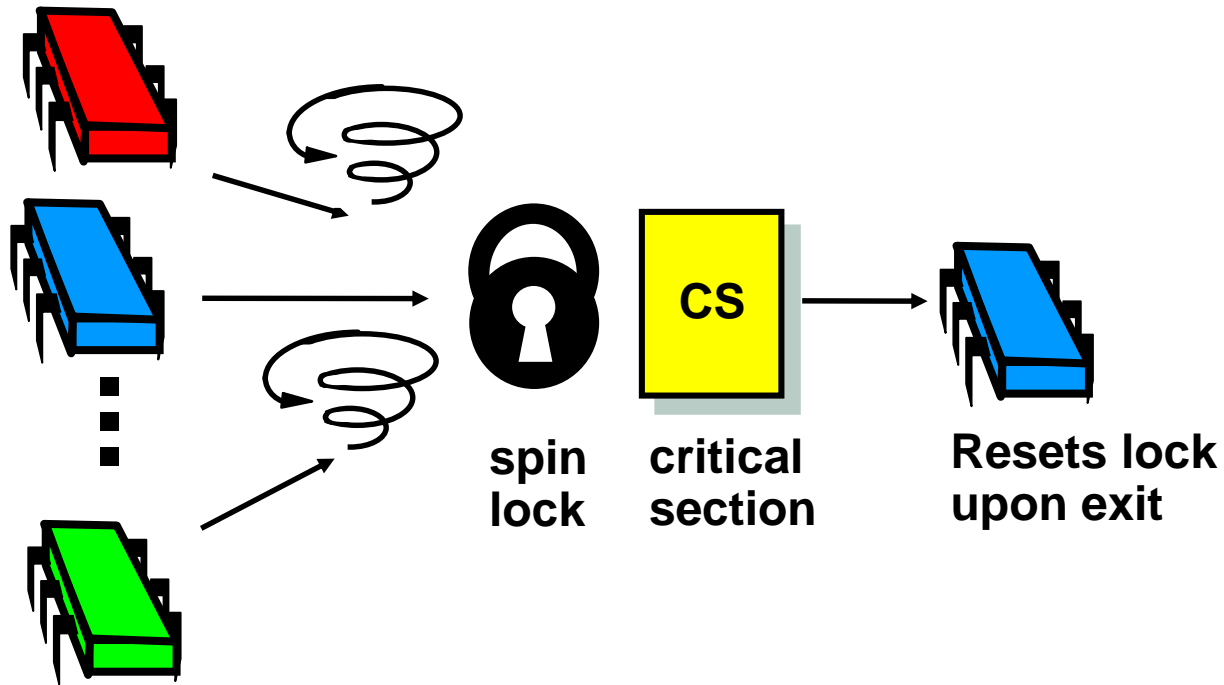
New goals

- Think of performance, not just correctness and progress
- Understand the underlying architecture
- Understand how the architecture affects performance
- Start with Mutual Exclusion

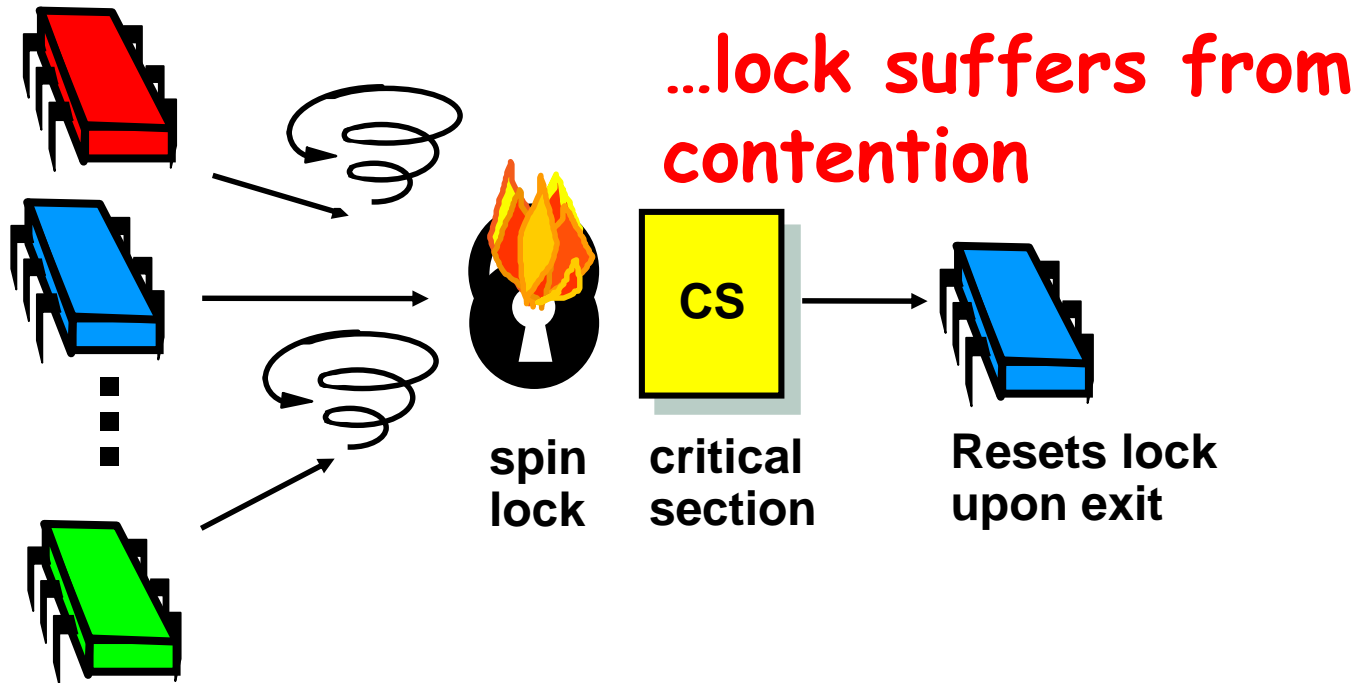
What should you do if you can't get a lock?

- Keep trying
 - “spin” or “busy-wait”
 - Good if delays are short
 - Give up the processor
 - Suspend yourself and ask the schedule to create another thread on your processor
 - Good if delays are long
 - Always good on uniprocessor
- our focus

Basic Spin-Lock



Basic Spin-Lock





Contention

- Contention:

- When multiple threads try to acquire a lock at the same time

- High contention:

- There are many such threads

- Low contention:

- The opposite



Welcome to the real world

■ Java Lock interface

- `java.util.concurrent.locks` package

```
Lock mutex = new LockImpl (...);  
...  
mutex.lock();  
try {  
    ...  
} finally {  
    mutex.unlock();  
}
```



Why don't we just use the Filter or Bakery Locks?

- The principal drawback is the need to read and write n distinct locations where n is the number of concurrent threads
- This means that the locks require space linear in n


What about the Peterson lock?

```
class Peterson implements Lock {  
    private boolean[] flag = new boolean[2];  
    private int victim;  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;  
        victim = i;  
        while (flag[j] && victim == i) {}  
    }  
}
```



Peterson lock?

- It is not our logic that fails, but our assumptions about the real world
- We assumed that read and write operations are atomic
- Our proof relied on the assumption that any two memory accesses by the same thread, even to different variables, take place in program order



Why does it not take place in program order?

- Modern multiprocessors do not guarantee program order
- Due to:
 - Compilers
 - reorder instructions to enhance performance
 - Multiprocessor hardware itself
 - writes to multiprocessor memory do not necessarily take effect when they are issued
 - writes to shared memory are buffered and written to memory only when needed

What about the Peterson lock?

```
class Peterson implements Lock {  
    private boolean[] flag = new boolean[2];  
    private int victim;  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;  
        victim = i;  
        while (flag[j] && victim == i) {};  
    }  
}
```

Important that these
steps take place in
program order



How can one fix this?

- Memory barriers (or memory fences) can be used to force outstanding operations to take effect
- It is the programmer's responsibility to know when to insert a memory barrier
- However, memory barriers are expensive



Memory barriers

- Synchronization instructions such as `getAndSet()` or `compareAndSet()` often include memory barriers
- As do reads and writes to **volatile** fields



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.set(false);  
    }  
}
```

**Lock state is
AtomicBoolean**

Test-and-set Lock

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

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

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

**Keep trying until
lock acquired**

Test-and-set Lock

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

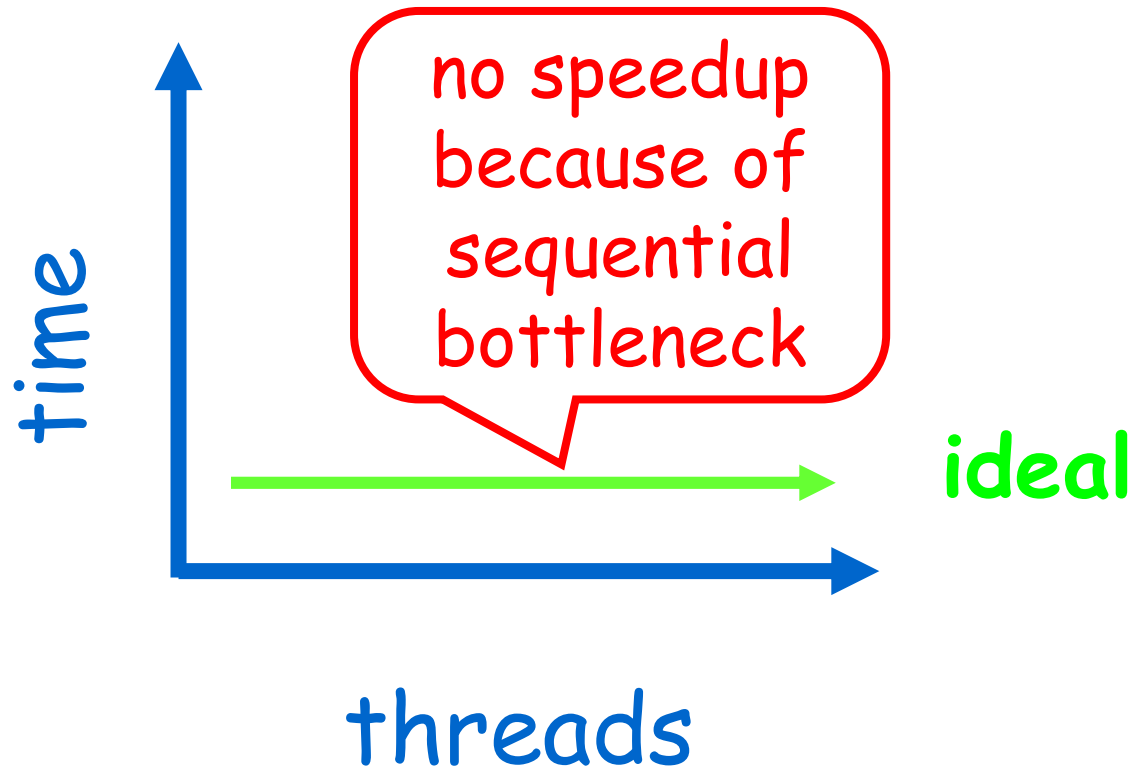
**Release lock by
resetting state to
false**



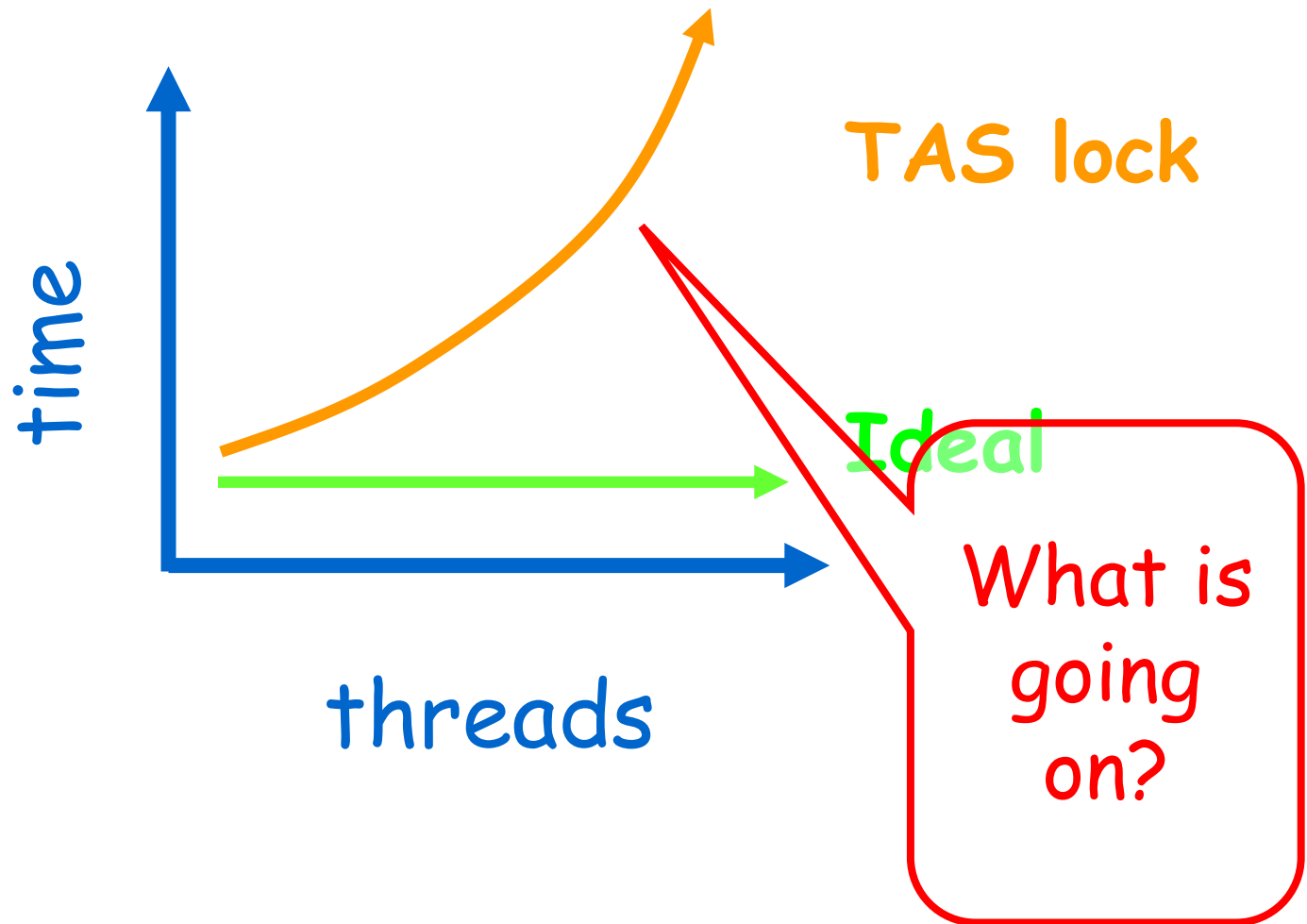
Performance

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

Graph



Mystery #1

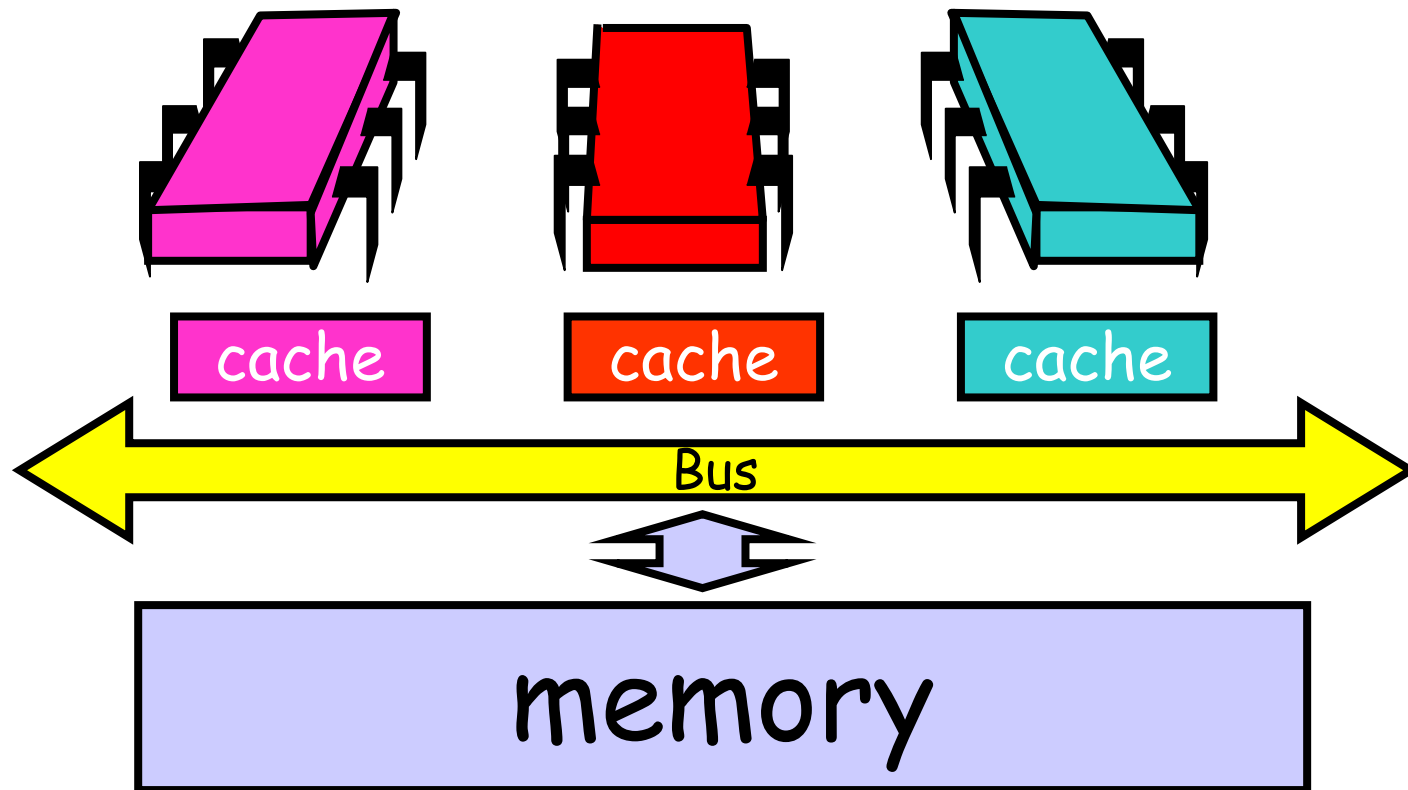




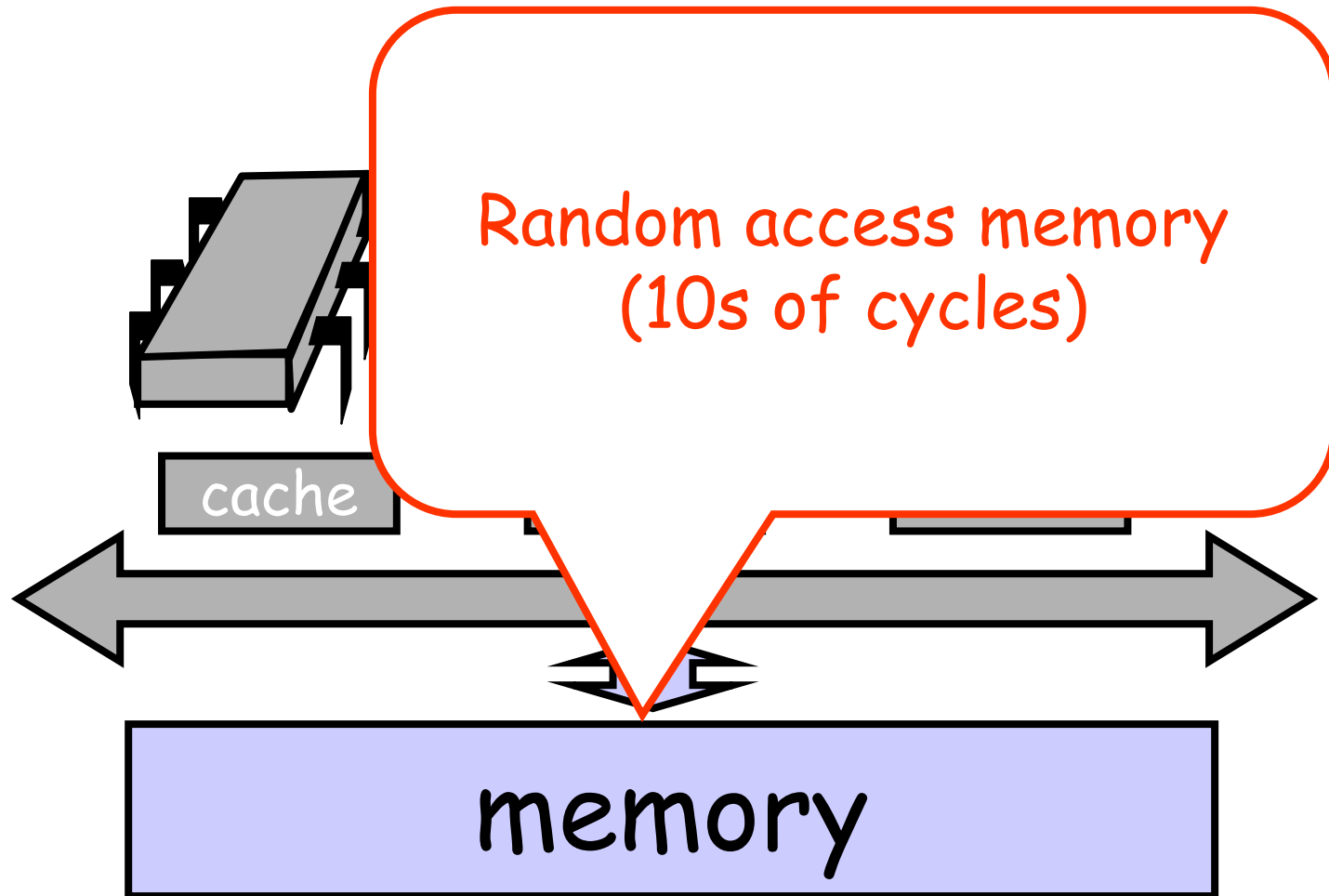
Questions

- Why is TAS so bad (so much worse than ideal)?

Bus-Based Architectures



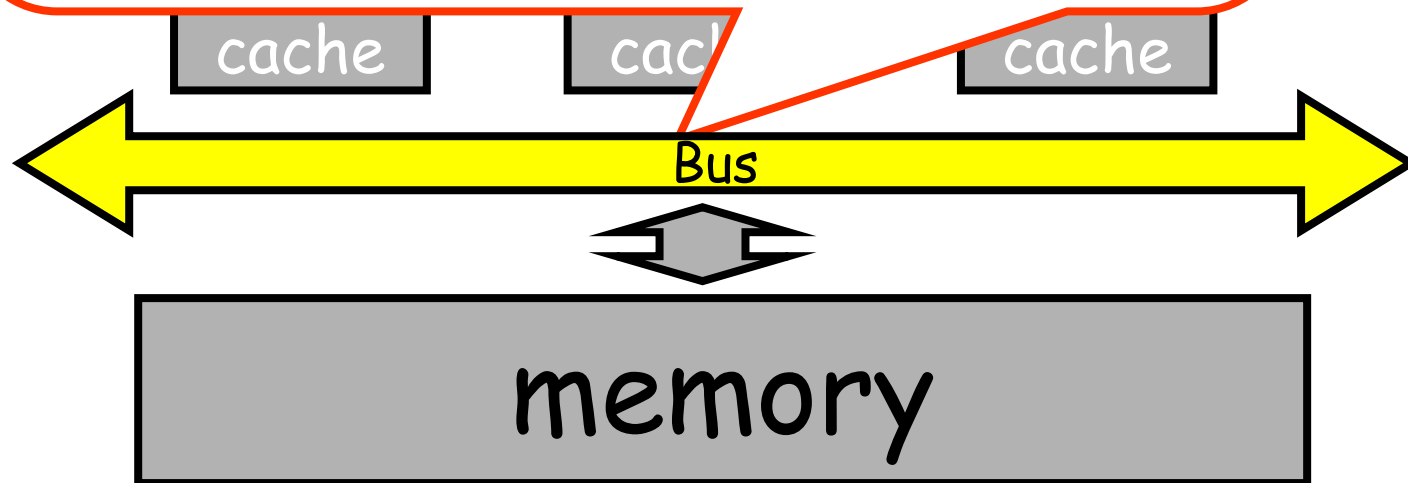
Bus-Based Architectures



Bus-Based Architectures

Shared Bus

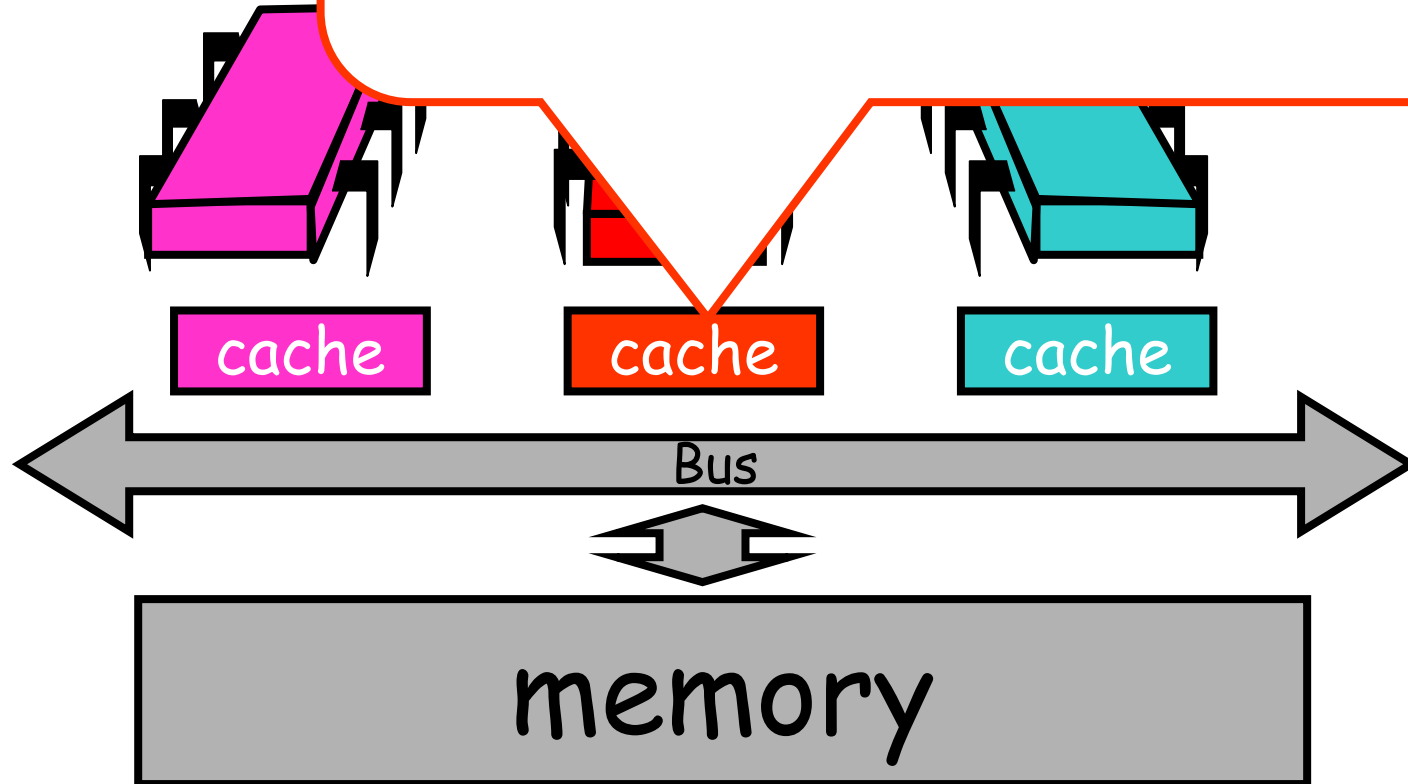
- Broadcast medium
- One broadcaster at a time
- Processors and memory all "snoop"



Bus-Bas

Per-Processor Caches

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





Jargon Watch

- Cache hit

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



Jargon Watch

■ Cache hit

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

■ Cache miss

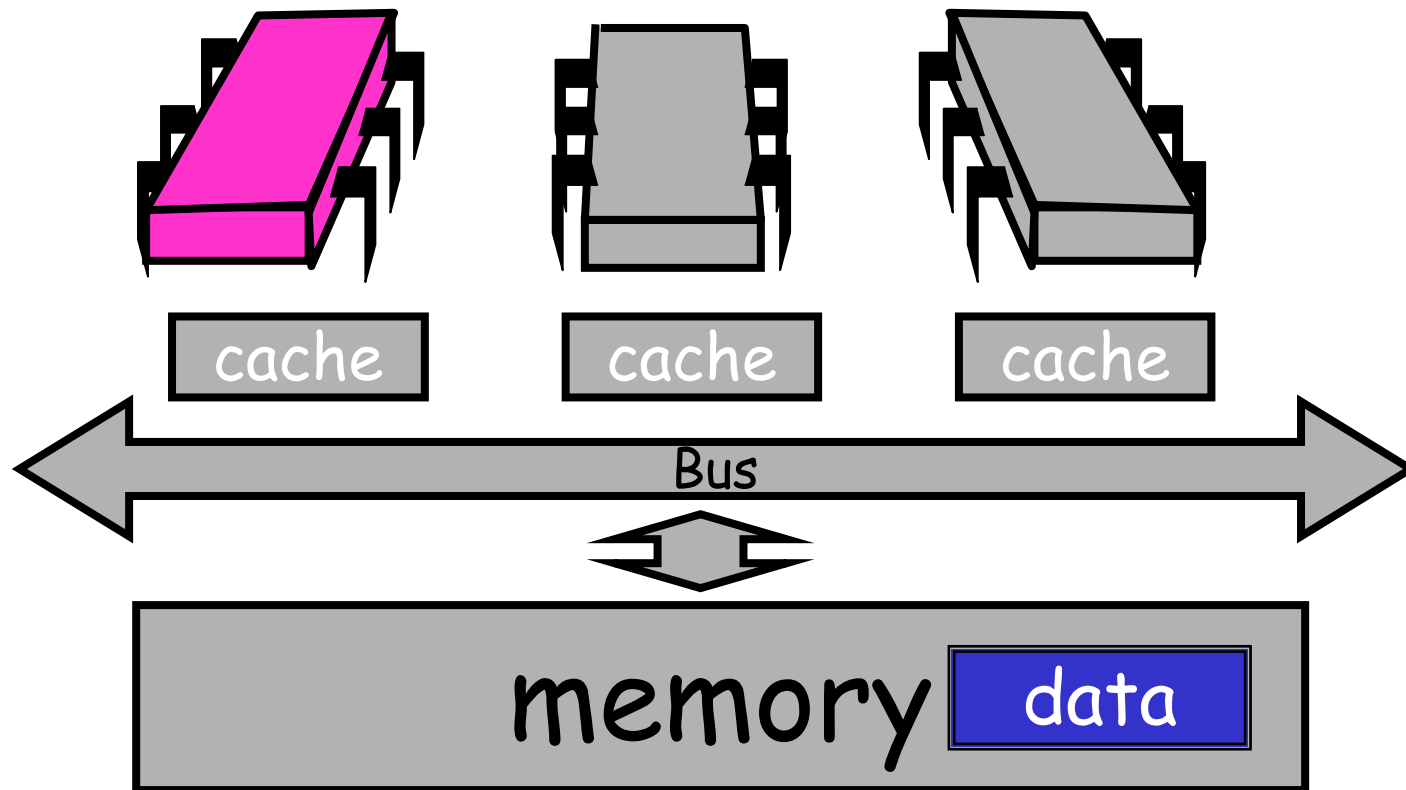
- “I had to shlep all the way to memory for that data”
- Bad Thing™



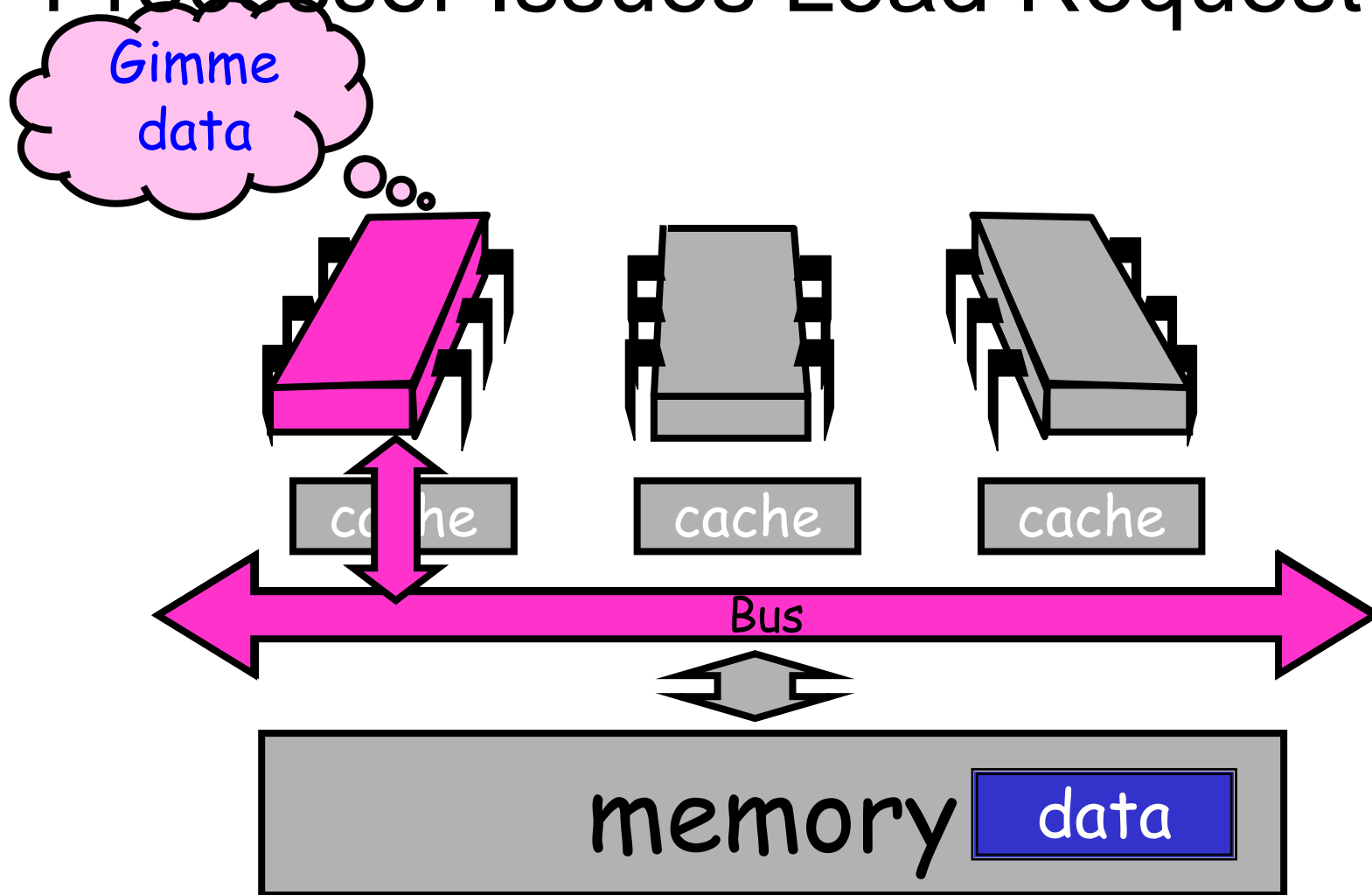
Cave Canem

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

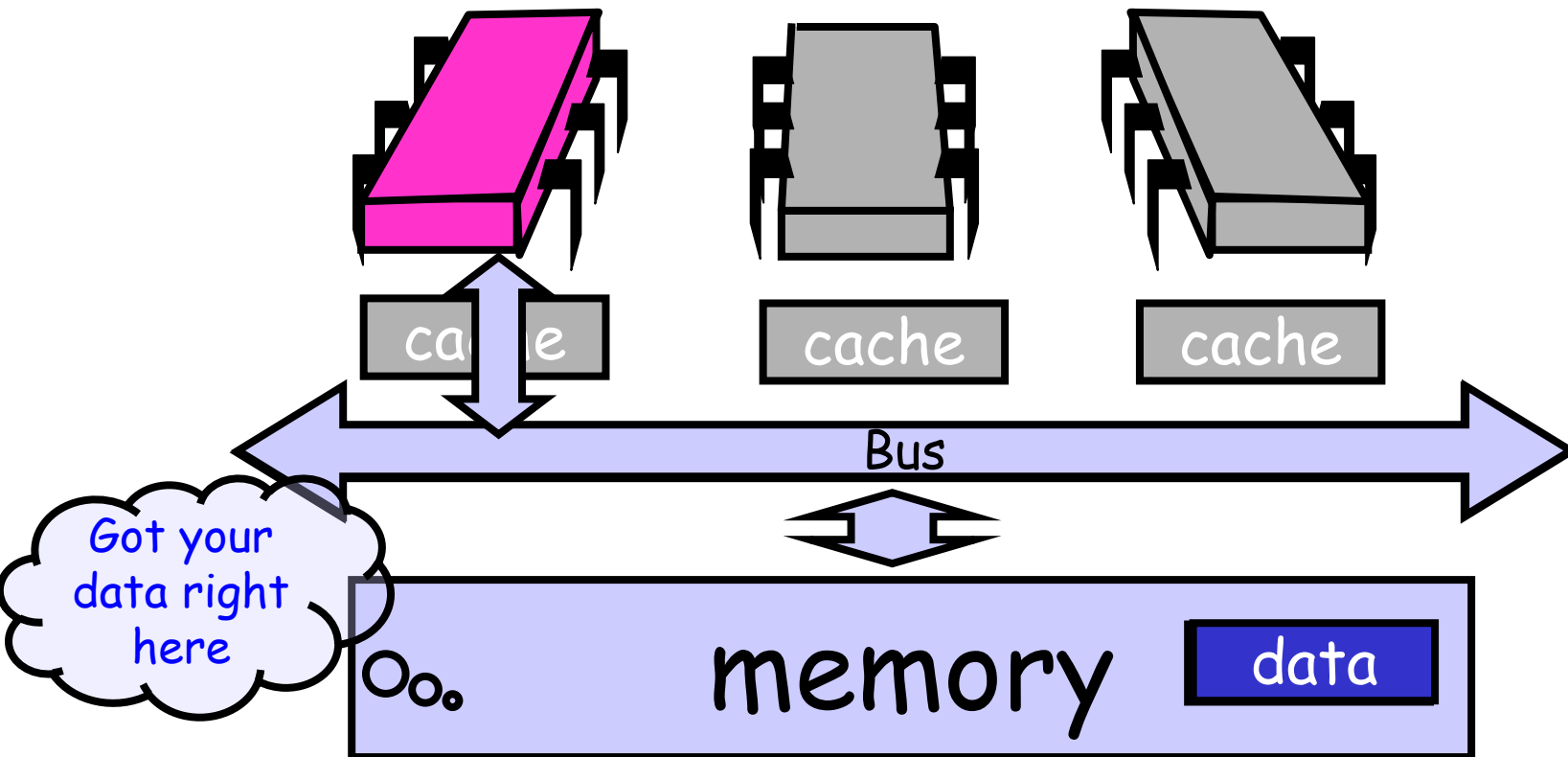
Processor Issues Load Request



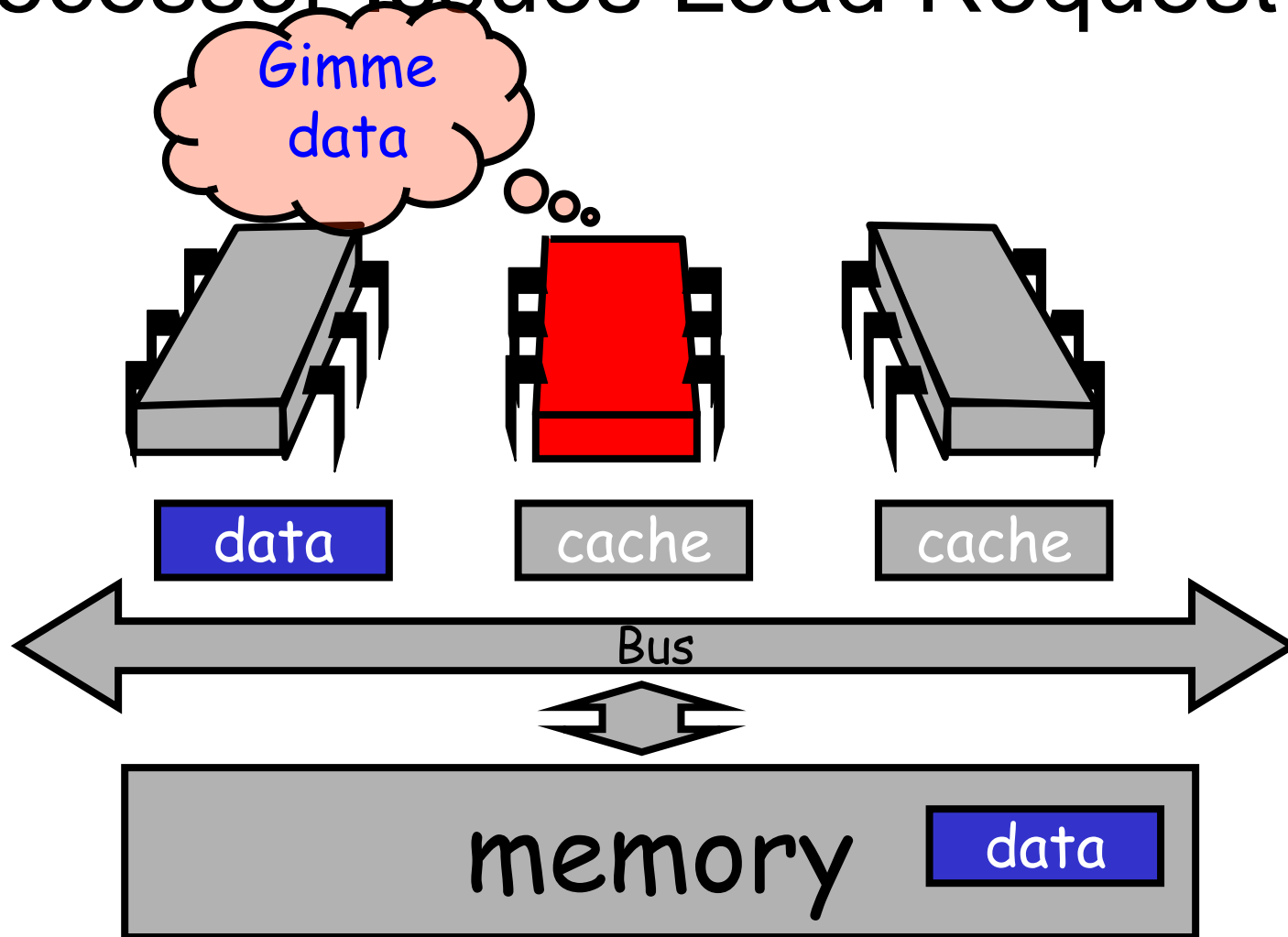
Processor Issues Load Request



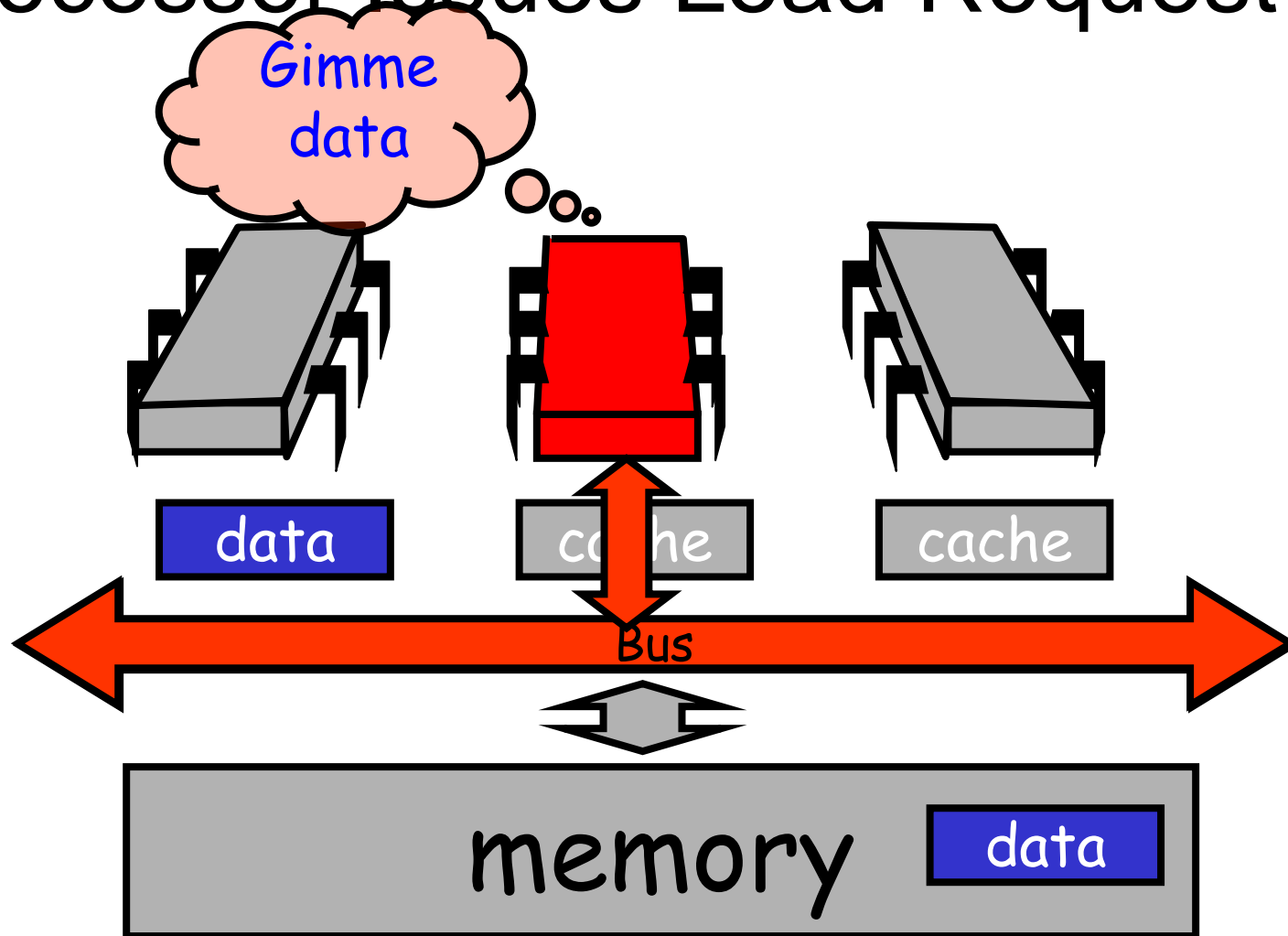
Memory Responds



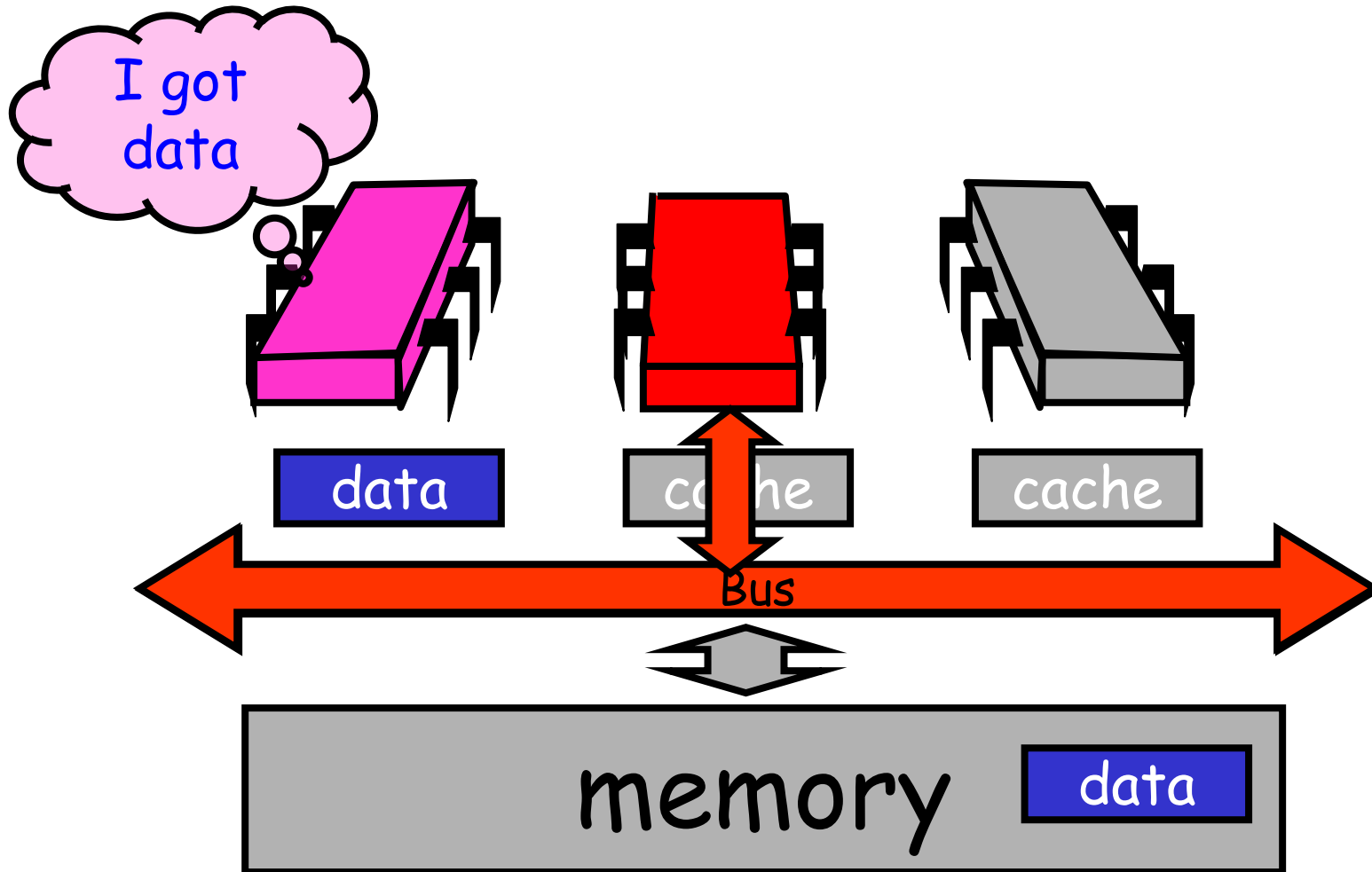
Processor Issues Load Request



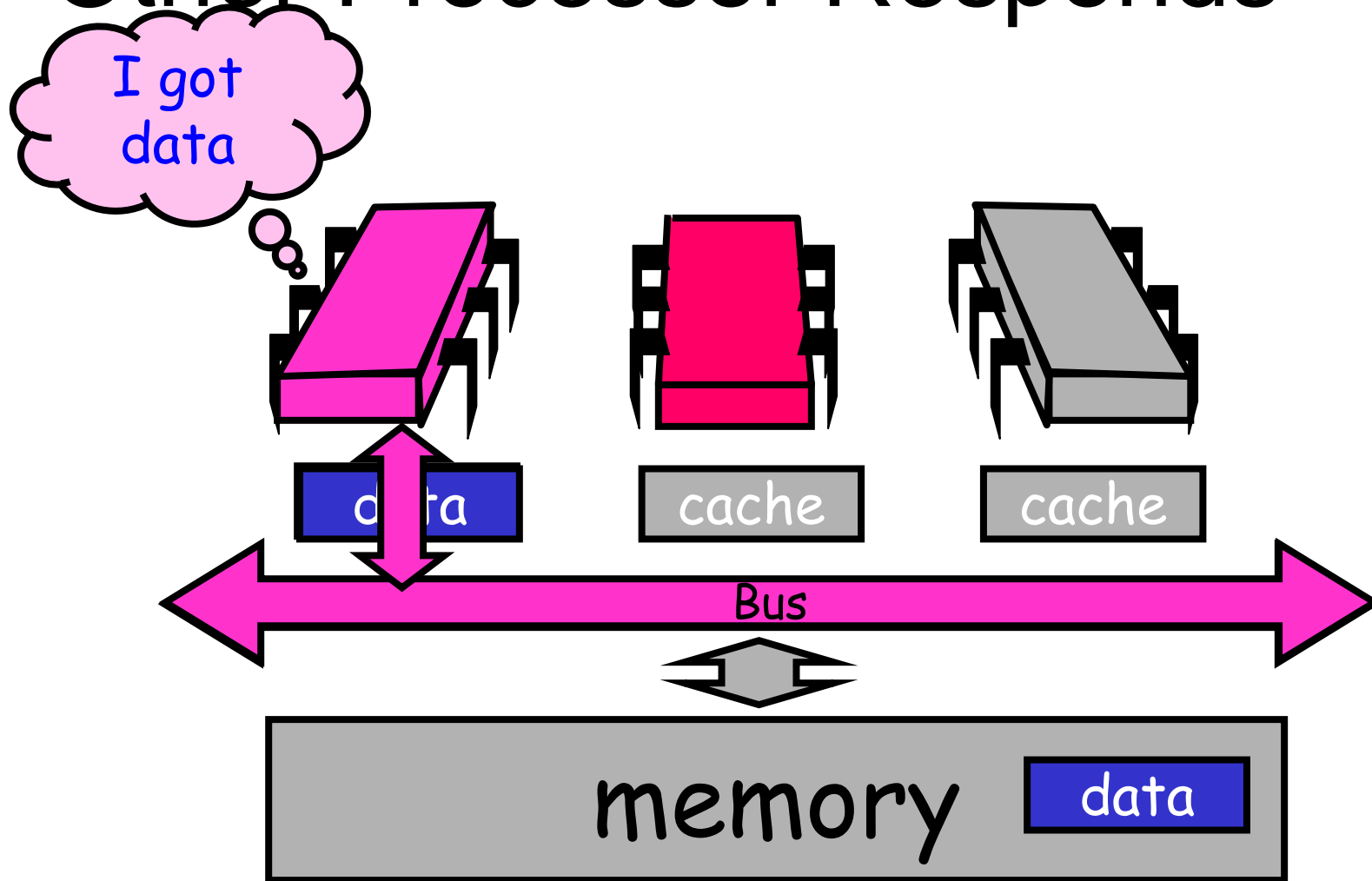
Processor Issues Load Request



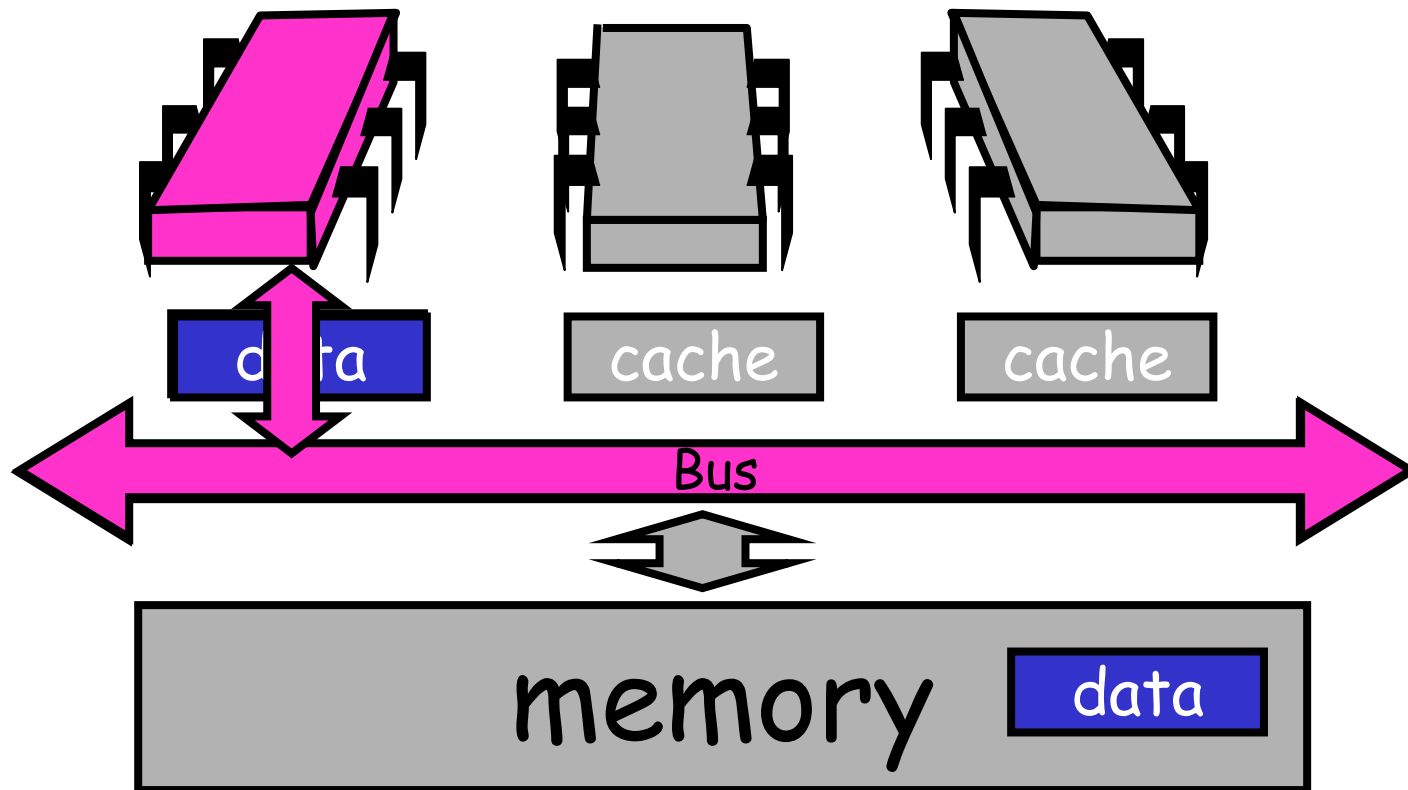
Processor Issues Load Request



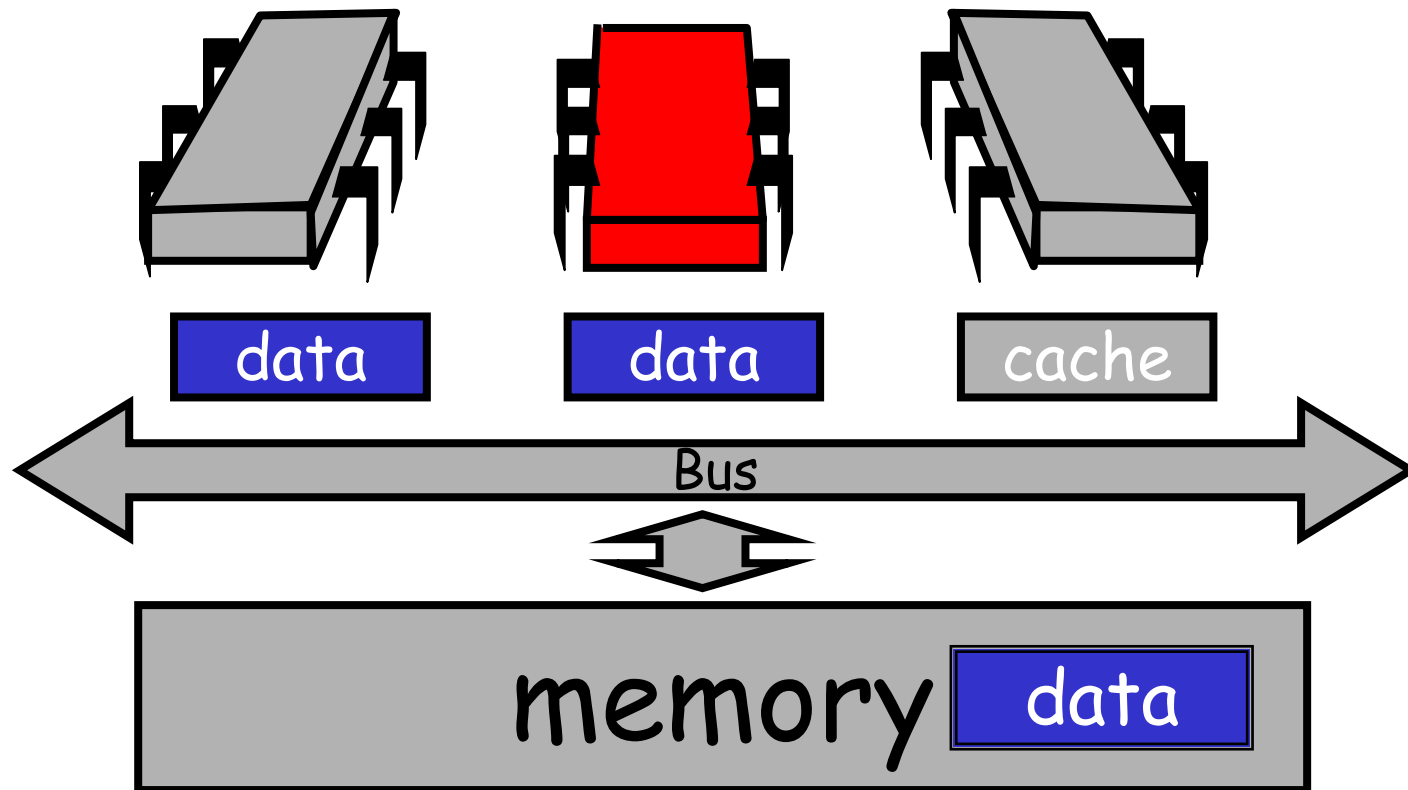
Other Processor Responds



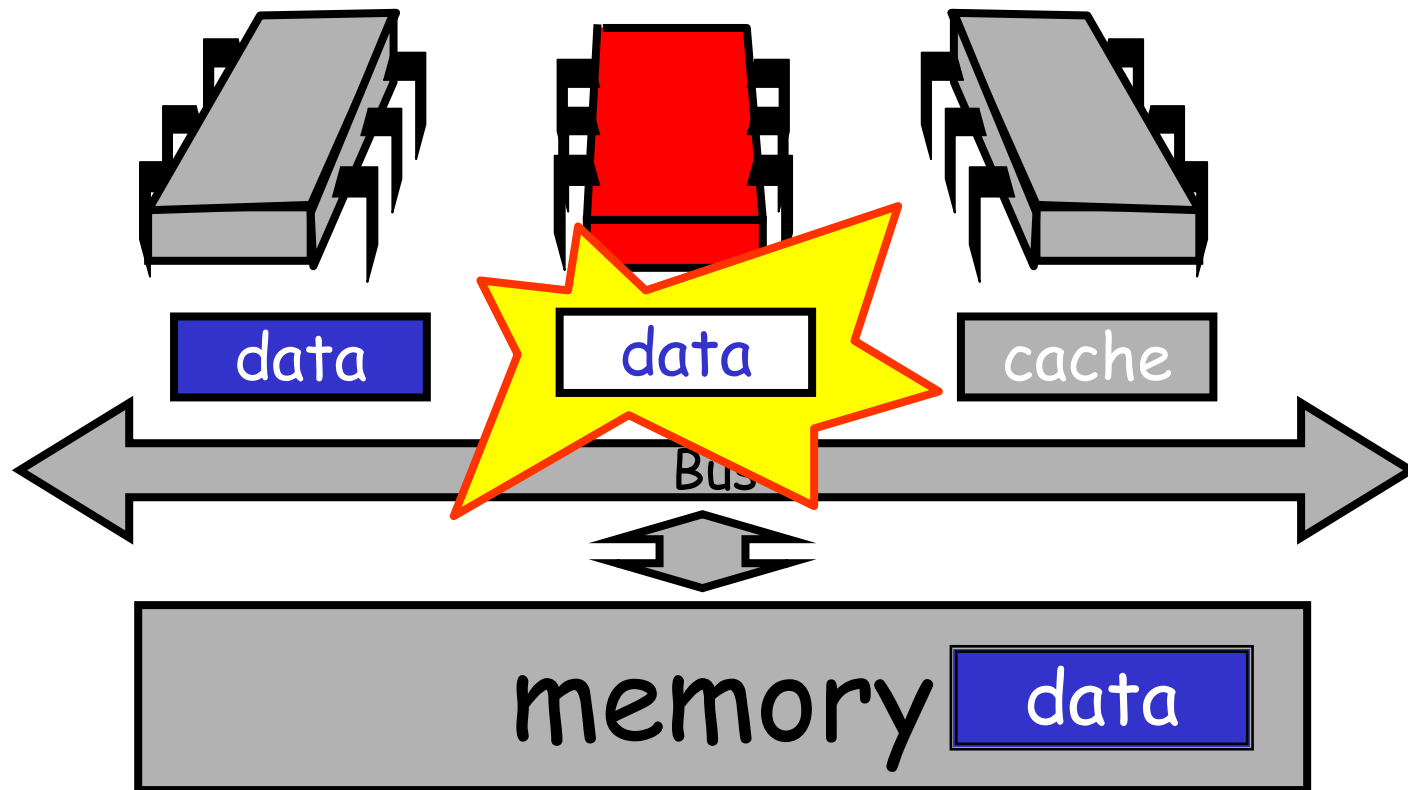
Other Processor Responds



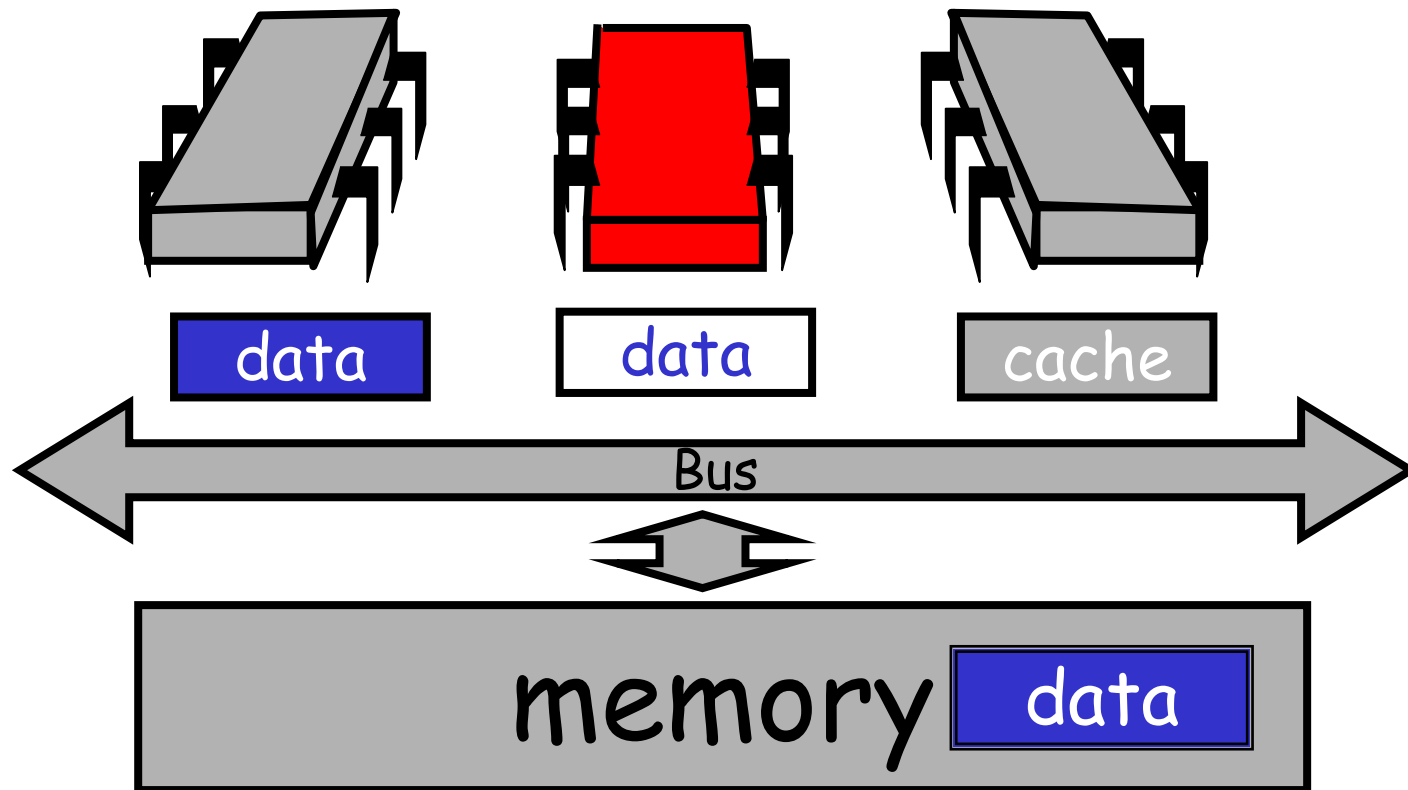
Modify Cached Data



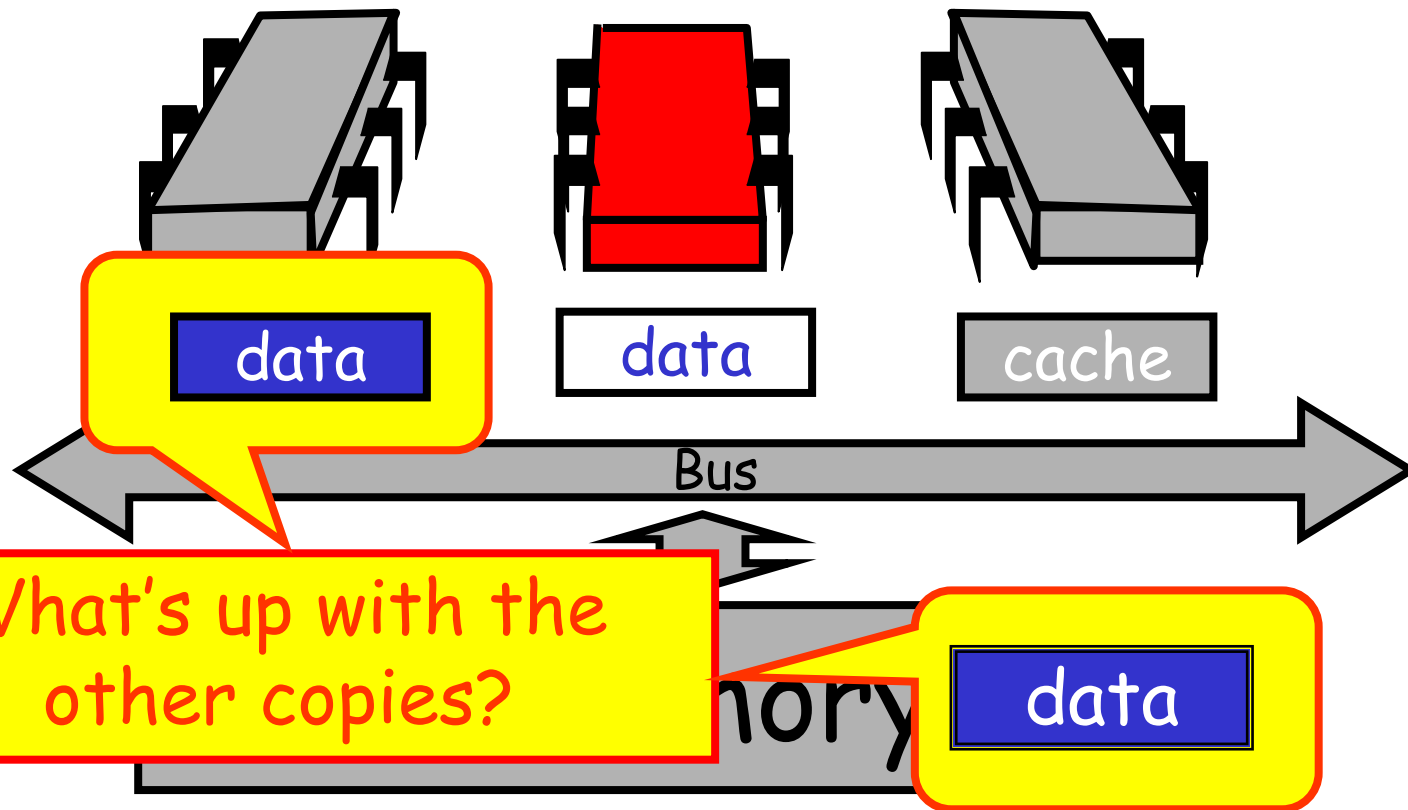
Modify Cached Data



Modify Cached Data



Modify Cached Data





Cache Coherence

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



Write-Back Cache Coherence Protocol

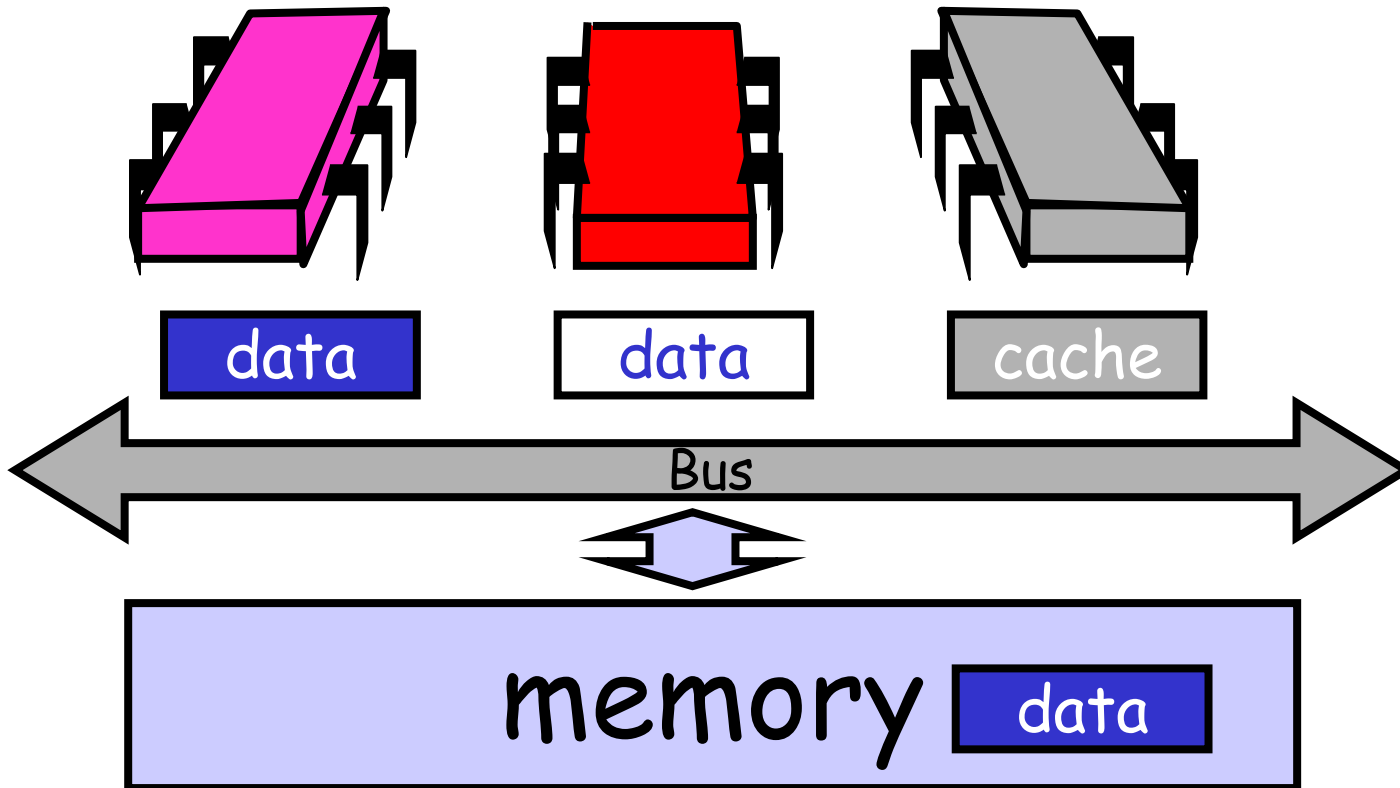
- Accumulate changes in cache
- Write back when needed
 - Need the cache for something else
 - Another processor wants it
- Write-back coherence protocol:
 - Invalidate other entries
 - Requires non-trivial protocol ...



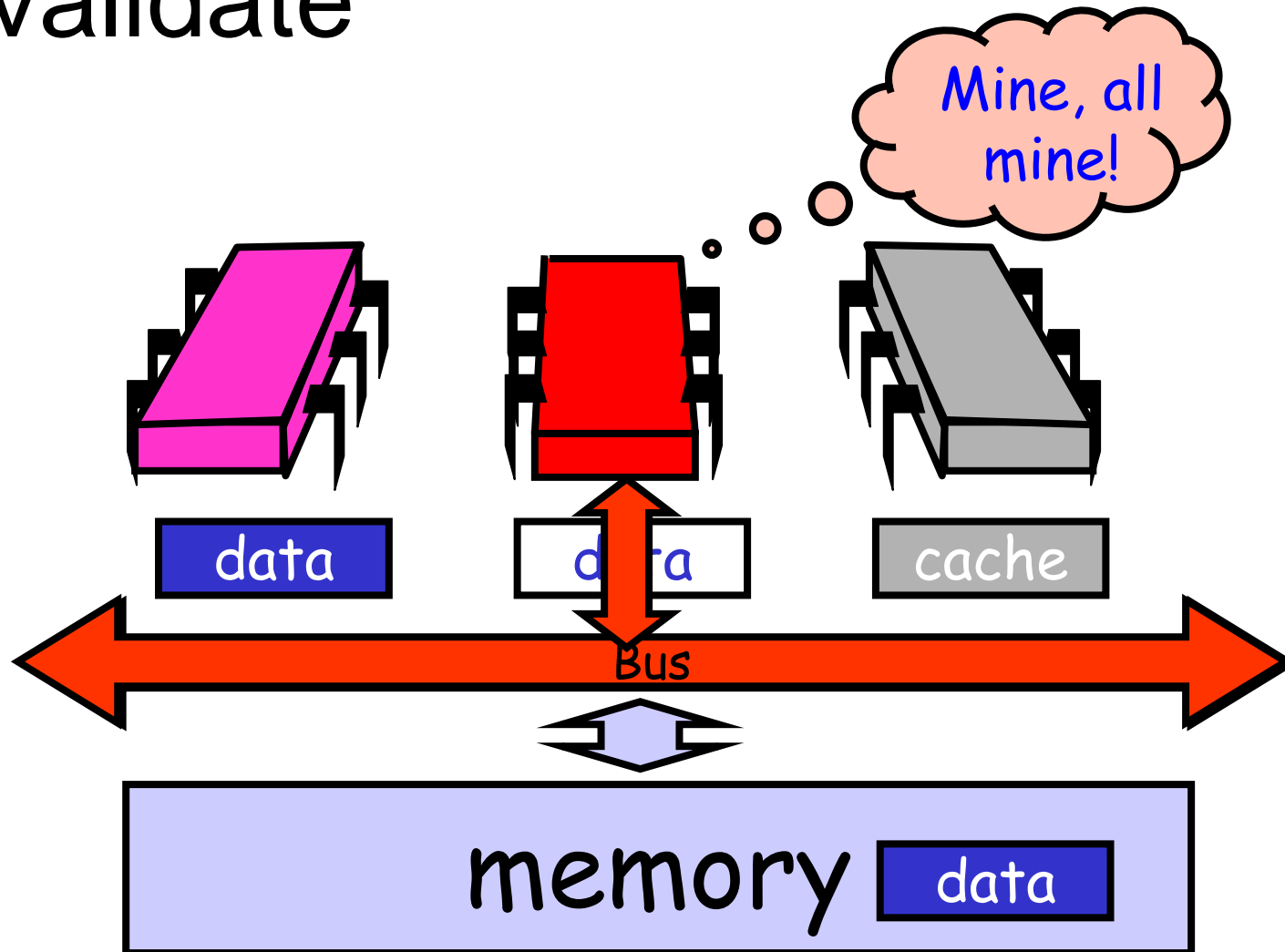
Write-Back Caches

- Cache entry has three states
 - Invalid: contains raw seething bits (meaningless)
 - Valid: I can read but I can't write because it may be cached elsewhere
 - Dirty: Data has been modified
 - Intercept other load requests
 - Write back to memory before using cache

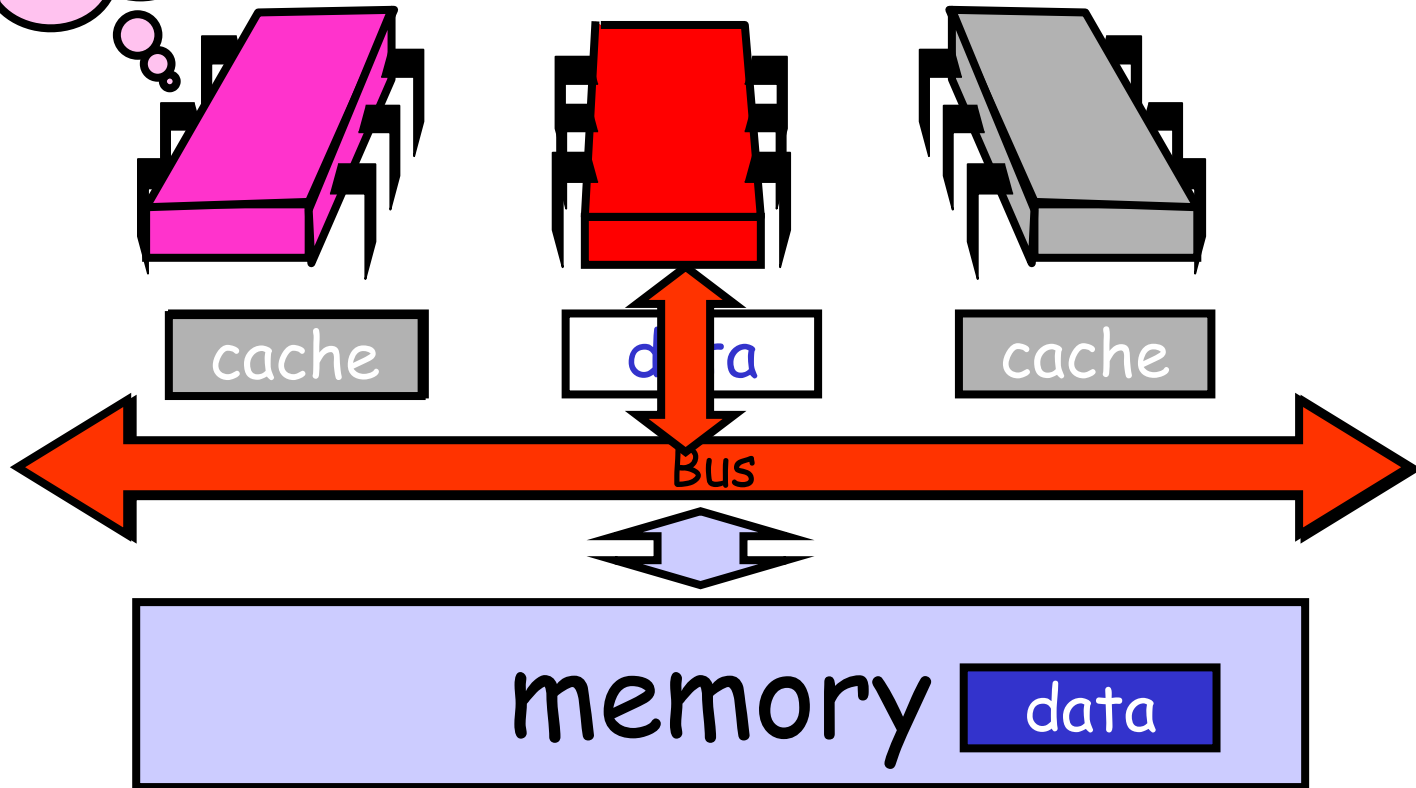
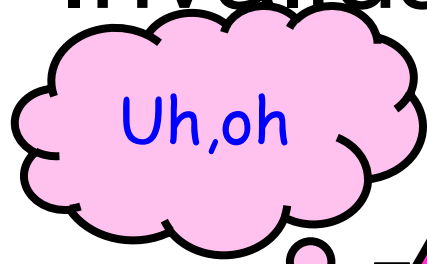
Invalidate



Invalidate

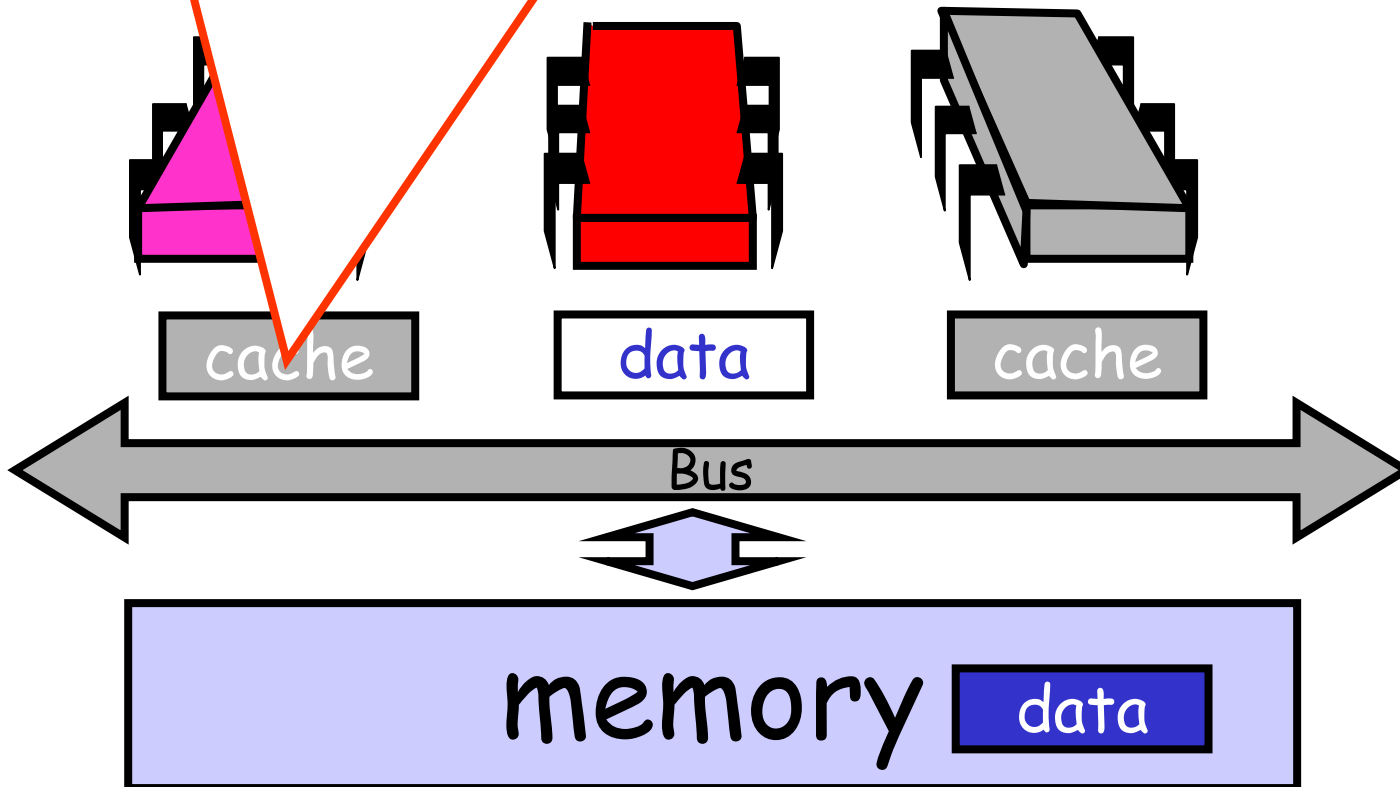


Invalidate



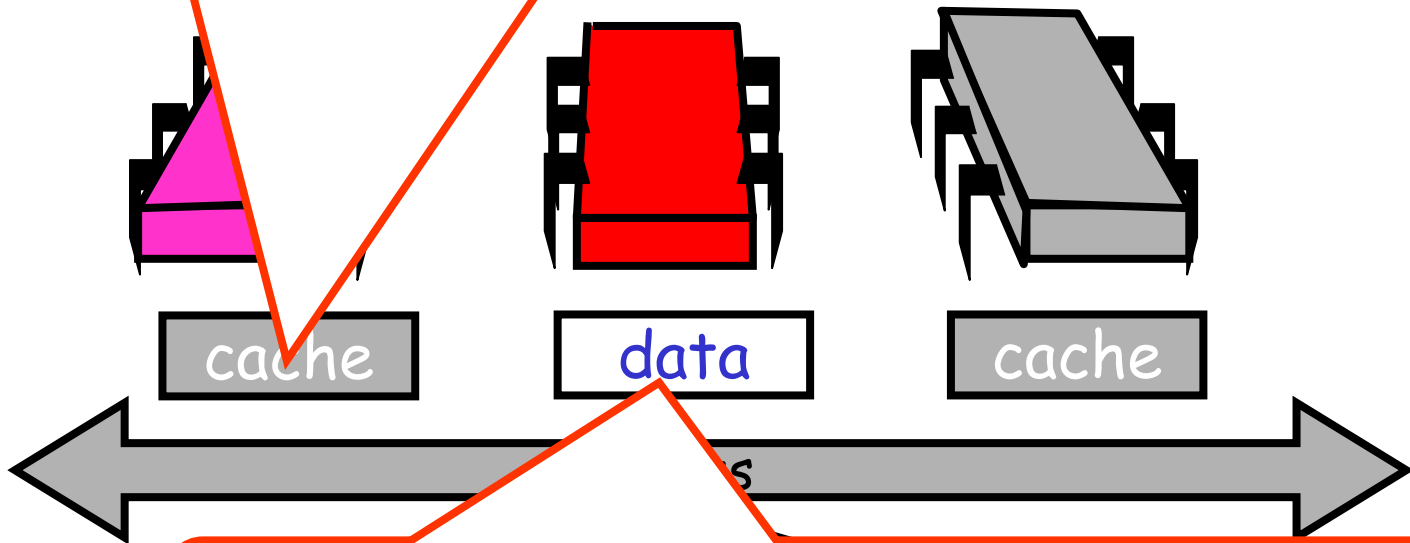
Invalidate

Other caches lose read permission



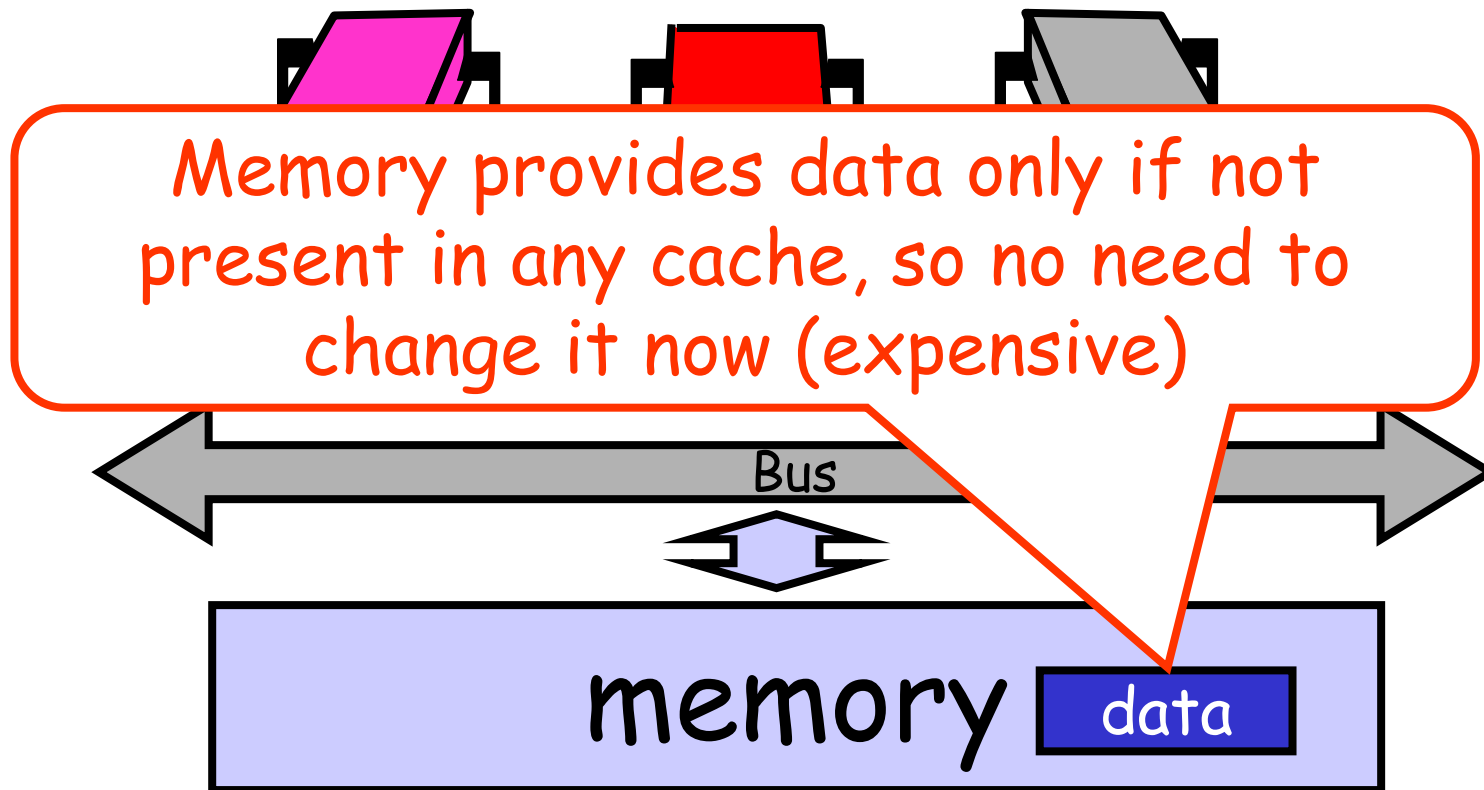
Invalidate

Other caches lose read permission

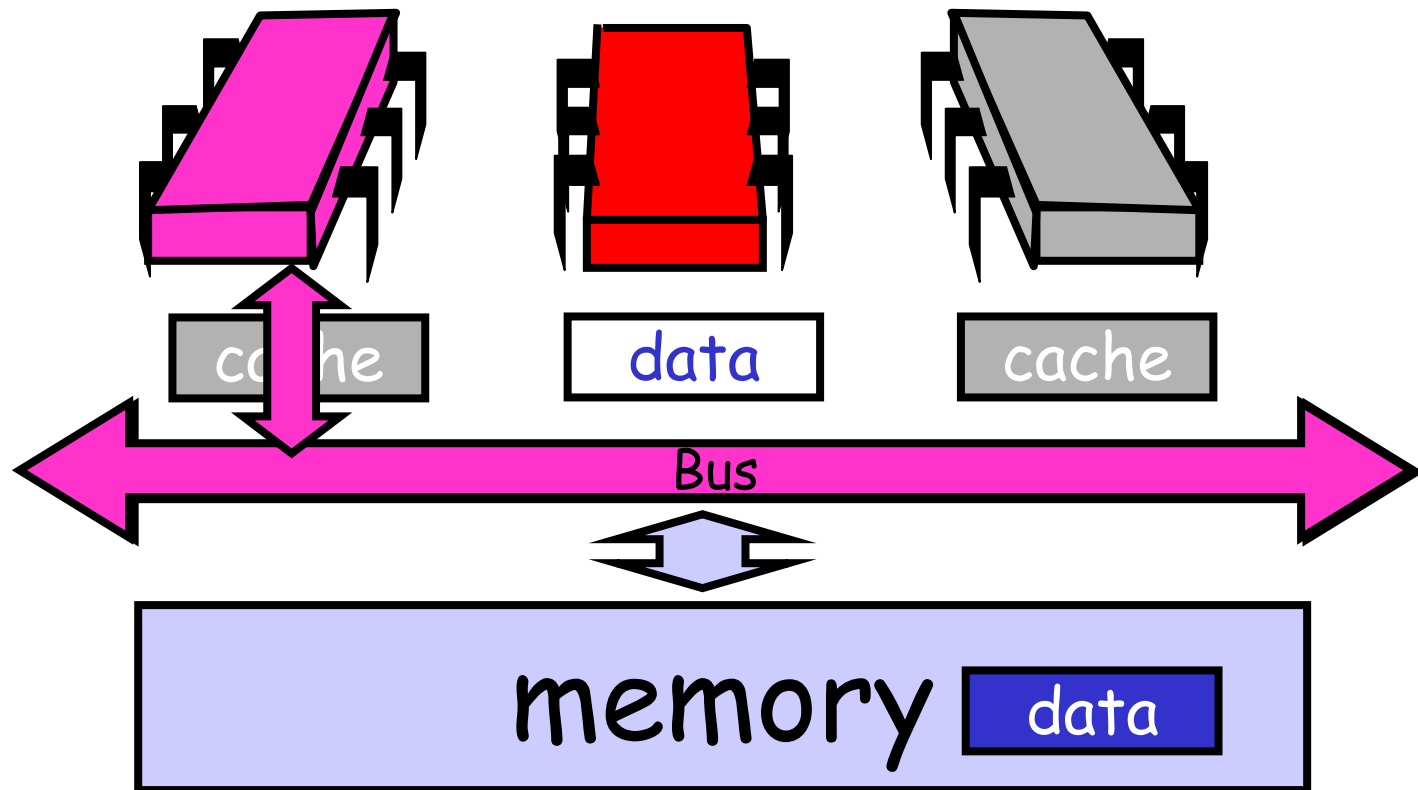


This cache acquires write permission

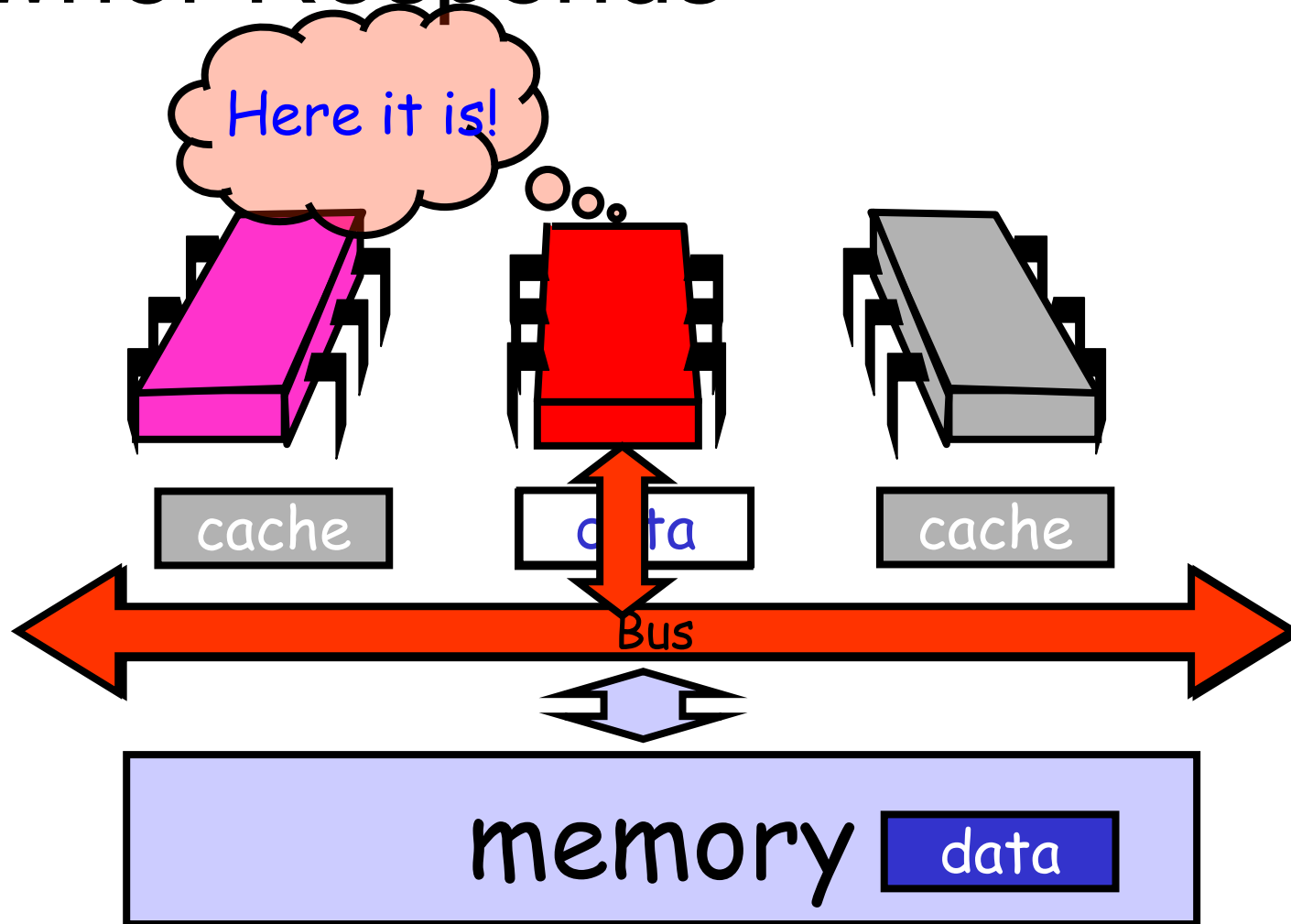
Invalidate



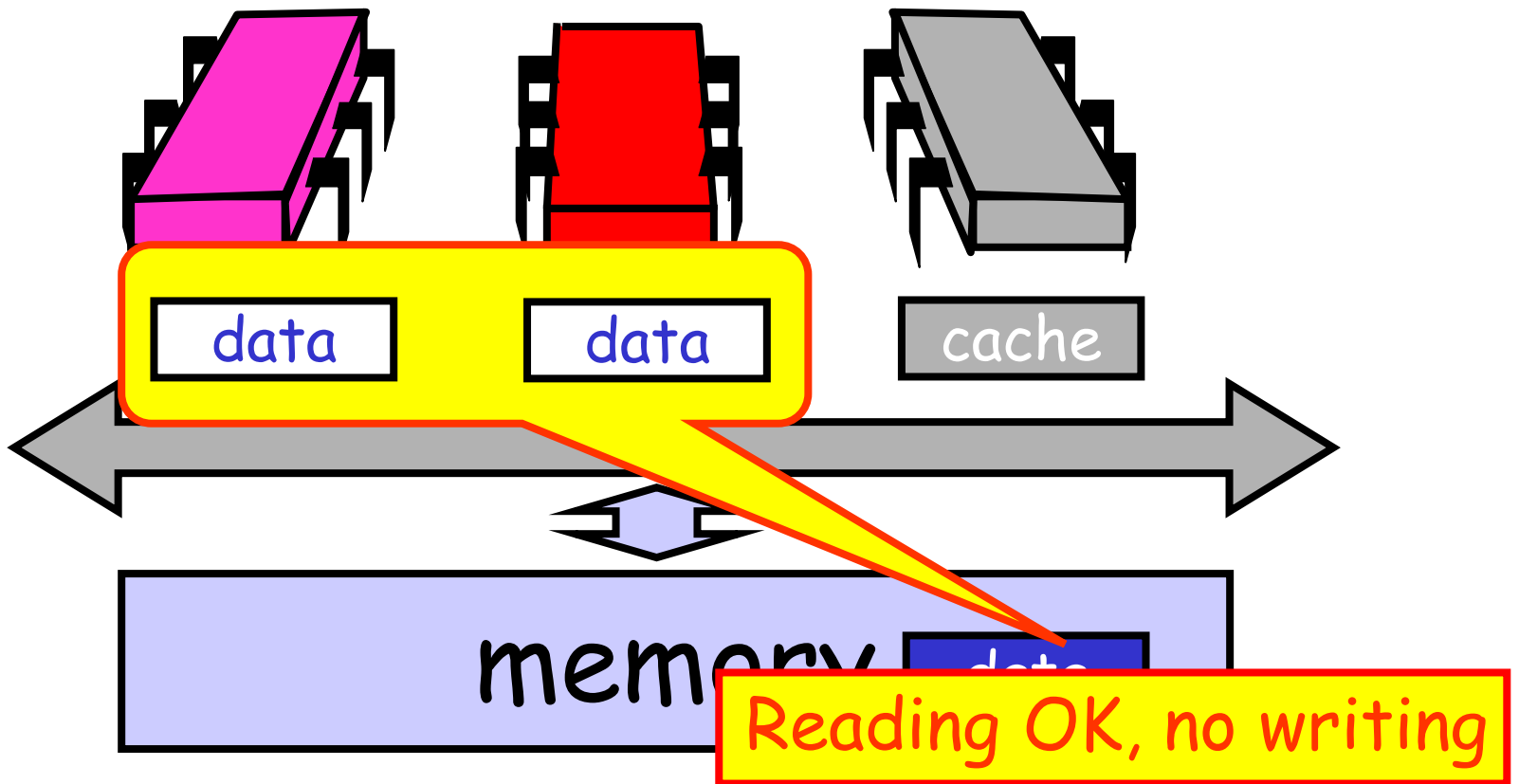
Another Processor Asks for Data



Owner Responds



End of the Day ...



Test-and-set Lock

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

Has to change to
current value while
spinning



Back to TASLocks

- How does a TASLock perform on a write-back shared-bus architecture?
 - Because it uses the bus, each getAndSet() call delays all the other threads
 - Even those not waiting for the lock
 - The getAndSet() call forces the other processors to discard their own cached copies – resulting in a cache miss every time
 - They must then use the bus to fetch the new, but unchanged value



TASLock

- When the thread wants to release the lock it may be delayed because the bus is being monopolized by the 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;  
        }  
    }  
}
```

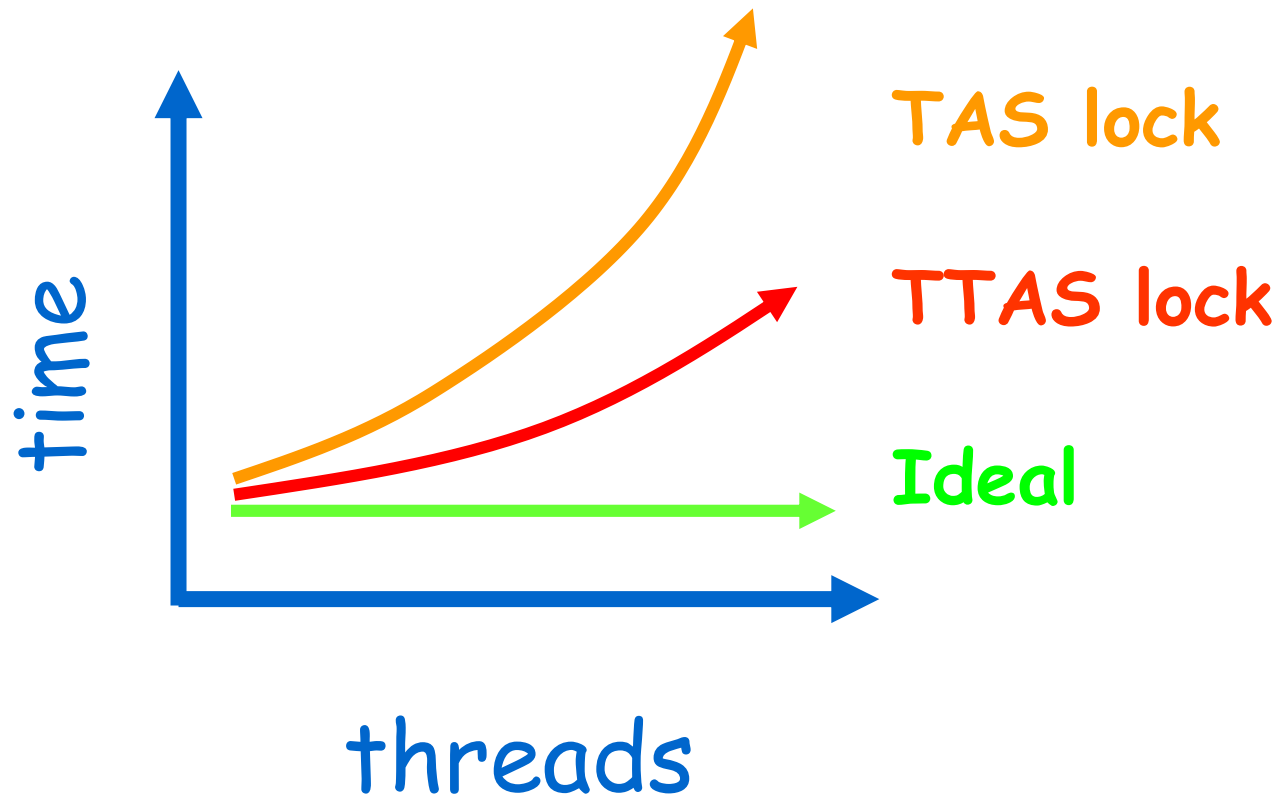
Wait until lock looks free

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





What about the TTASLock?

- Suppose thread A acquires the lock.
- The first time thread B reads the lock it takes a cache miss and has to use the bus to fetch the new value
- As long as A holds the lock however, B repeatedly rereads the value – resulting in a cache hit every time
- B thus produces no extra traffic



What about the TTASLock?

- However when A releases the lock:
 - A writes false to the lock variable
 - The spinner's cached copies are invalidated
 - Each one takes a cache miss
 - They all use the bus to read a new value
 - They all call getAndSet() to acquire the lock
 - The first one to acquire the lock invalidates the others who must then reread the value
 - Storm of traffic



Local spinning

- Threads repeatedly reread cached values instead of repeatedly using the bus



Exponential Backoff

- Recall that in the TTASLock, the thread first reads the lock and if it appears to be free it attempts to acquire the lock
- “If I see that the lock is free, but then another thread acquires it before I can, then there must be high contention for that lock”
- Better to back off and try again later



For how long should a thread back off?

- Rule of thumb:

- The larger number of unsuccessful tries, the higher the contention, the longer the thread should back off.

- Approach:

- Whenever the thread sees the lock has become free, but fails to acquire it, it backs off before retrying



What about lock-step?

- What happens if all the threads backs off the same amount of time?
- Instead the threads should back off for a random amount of time
- Each time the thread tries and fails to get the lock, it doubles the back-off time, up to a fixed maximum.



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 *  
            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 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;  
        }  
    }  
}
```

Otherwise back off for random duration

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;  
        }  
    }  
}
```

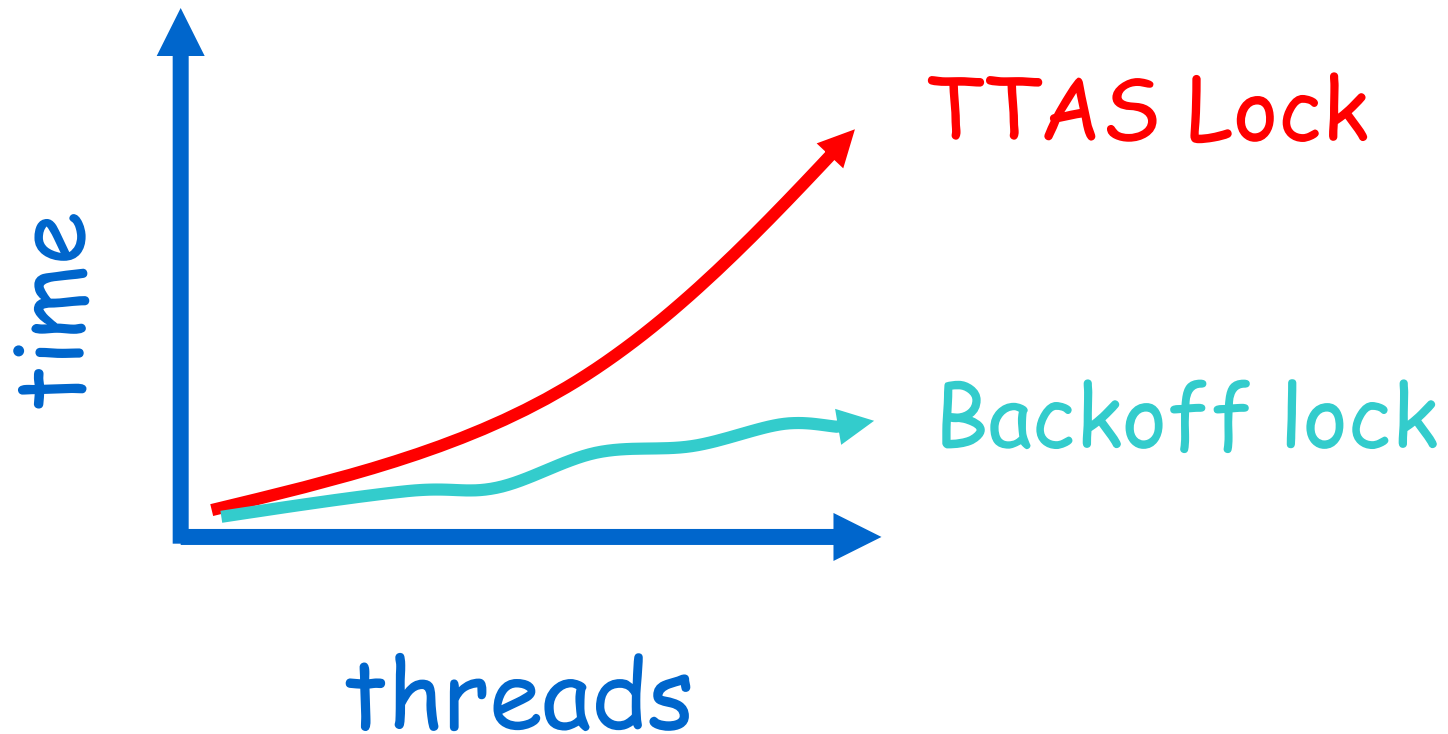
Double max delay,
within reason



Exponential Backoff Lock

- Important to note that a thread backs off only when it fails to acquire a lock that was just available, but is not available anymore
- Observing that a lock is held by another thread does not imply backoff

Spin-Waiting Overhead





Backoff: Other Issues

■ Good

- Easy to implement
- Beats TTAS lock

■ Bad

- Must choose parameters carefully
 - Sensitive to choice of minimum and maximum delays
- Sensitive to number of processors and their speed
 - Cannot have a general solution for all platforms and machines



BackoffLock drawbacks

- Cache-coherence Traffic:
 - All threads spin on the same location
- Critical Section Underutilization:
 - Threads delay longer than necessary



Idea

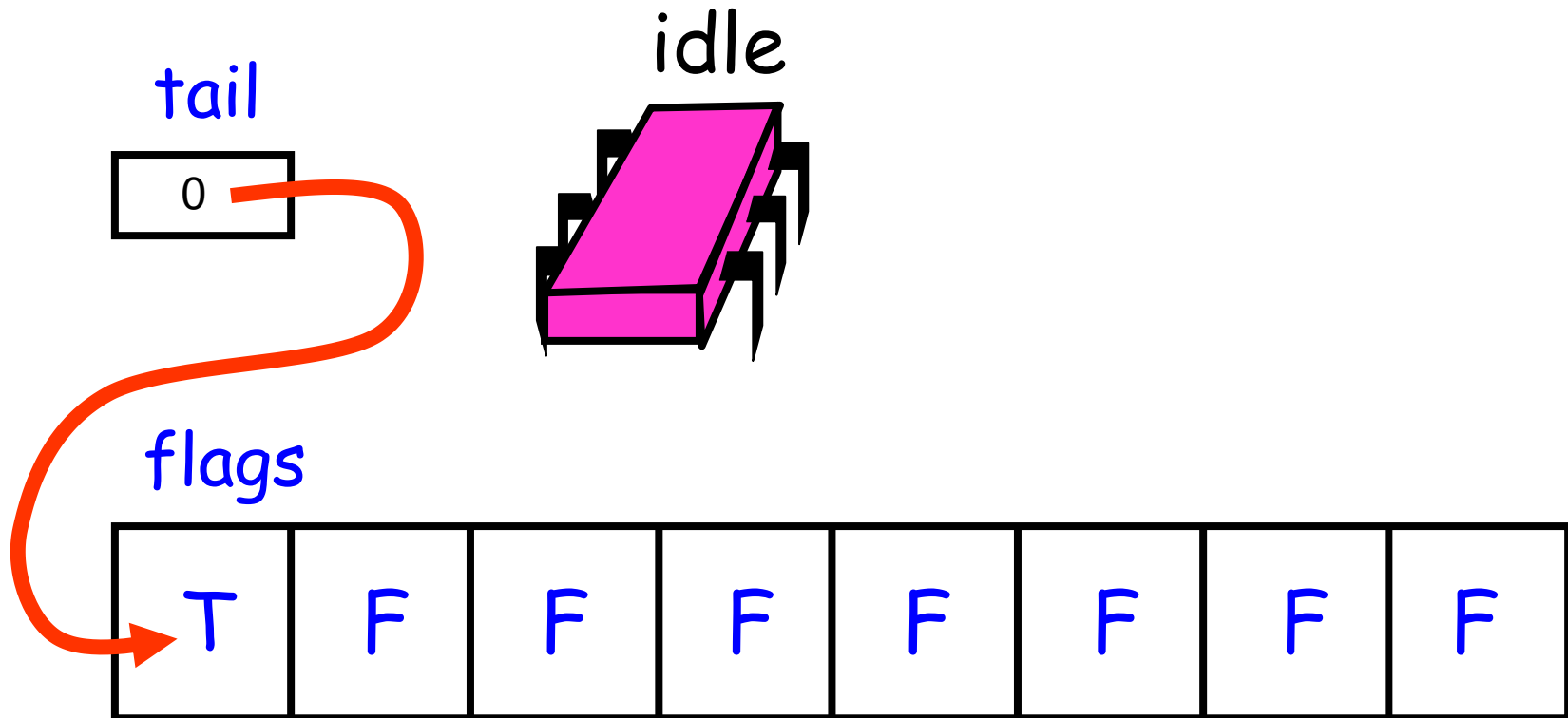
- Avoid useless invalidations
 - By keeping a queue of threads
- Each thread
 - Notifies next in line
 - Without bothering the others



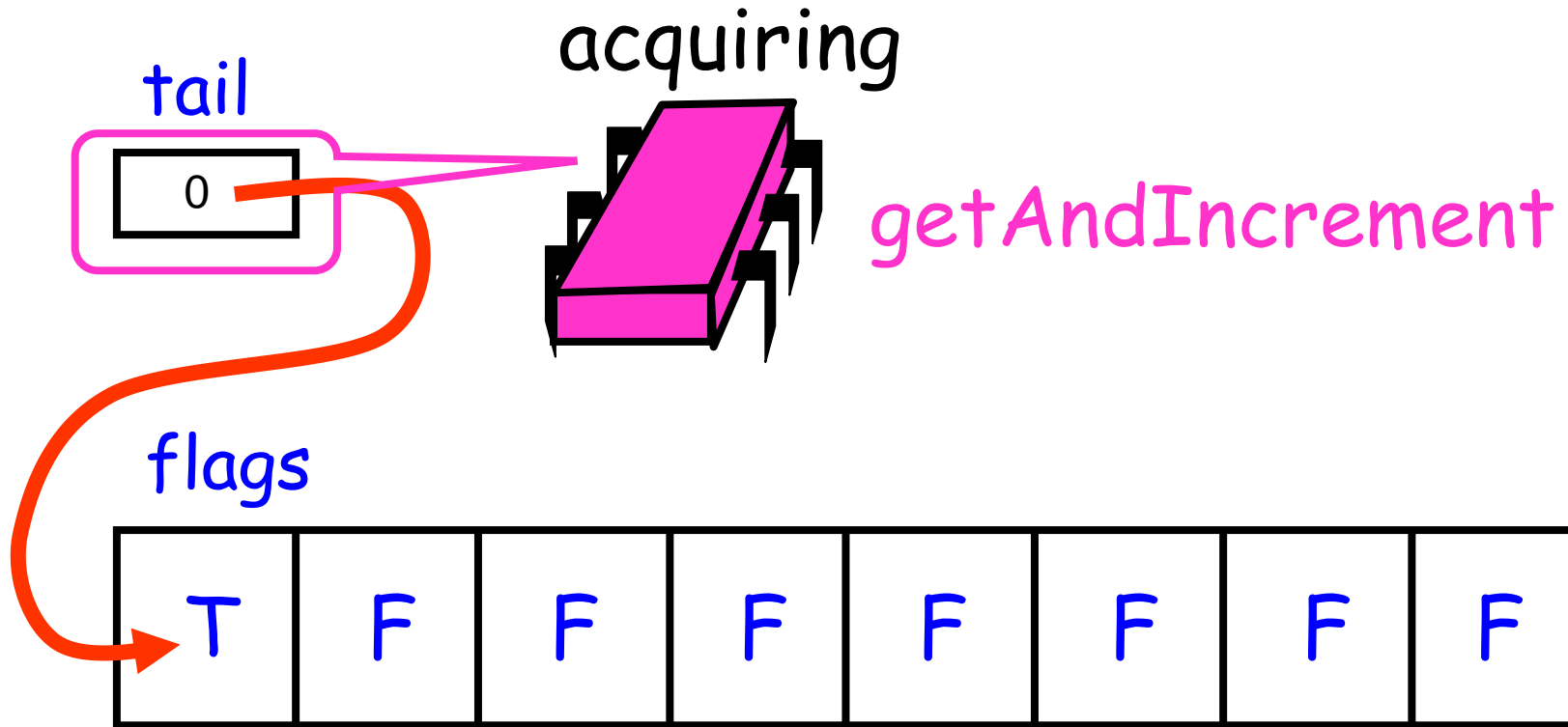
Queue Locks

- Cache-coherence traffic is reduced since each thread spins on a different location
- No need to guess when to attempt to access lock – increase critical section utilization
- First-come-first-served fairness

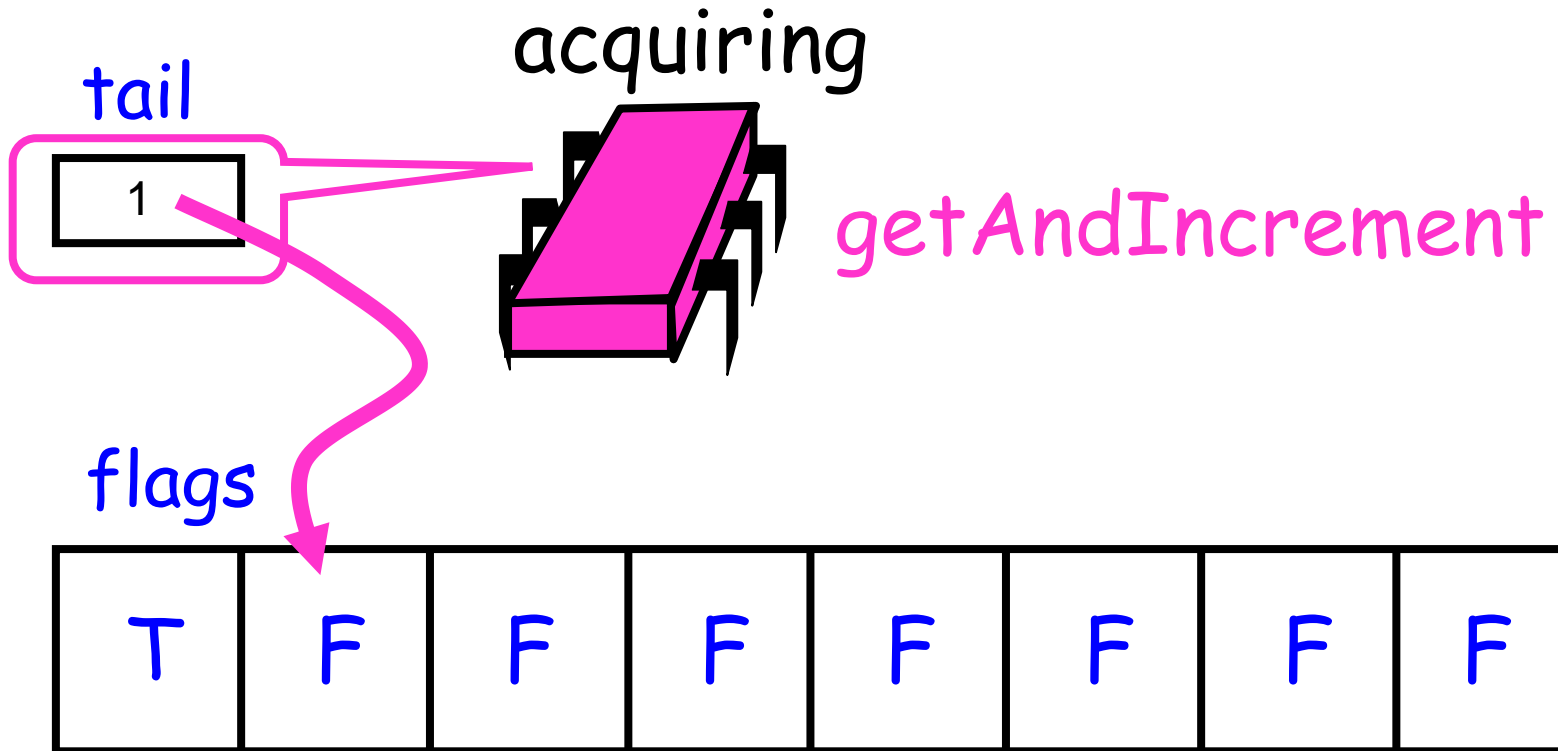
Anderson Queue Lock



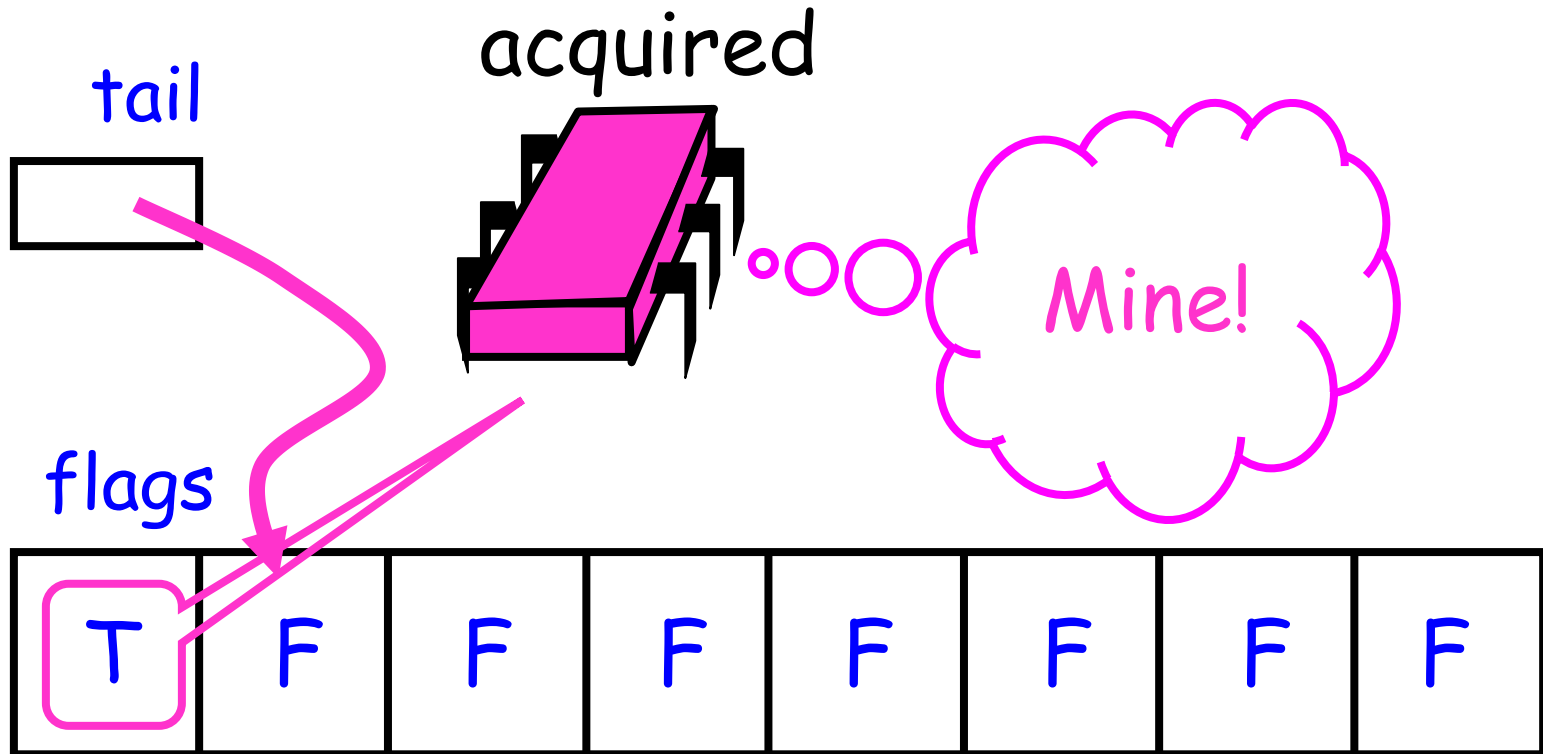
Anderson Queue Lock



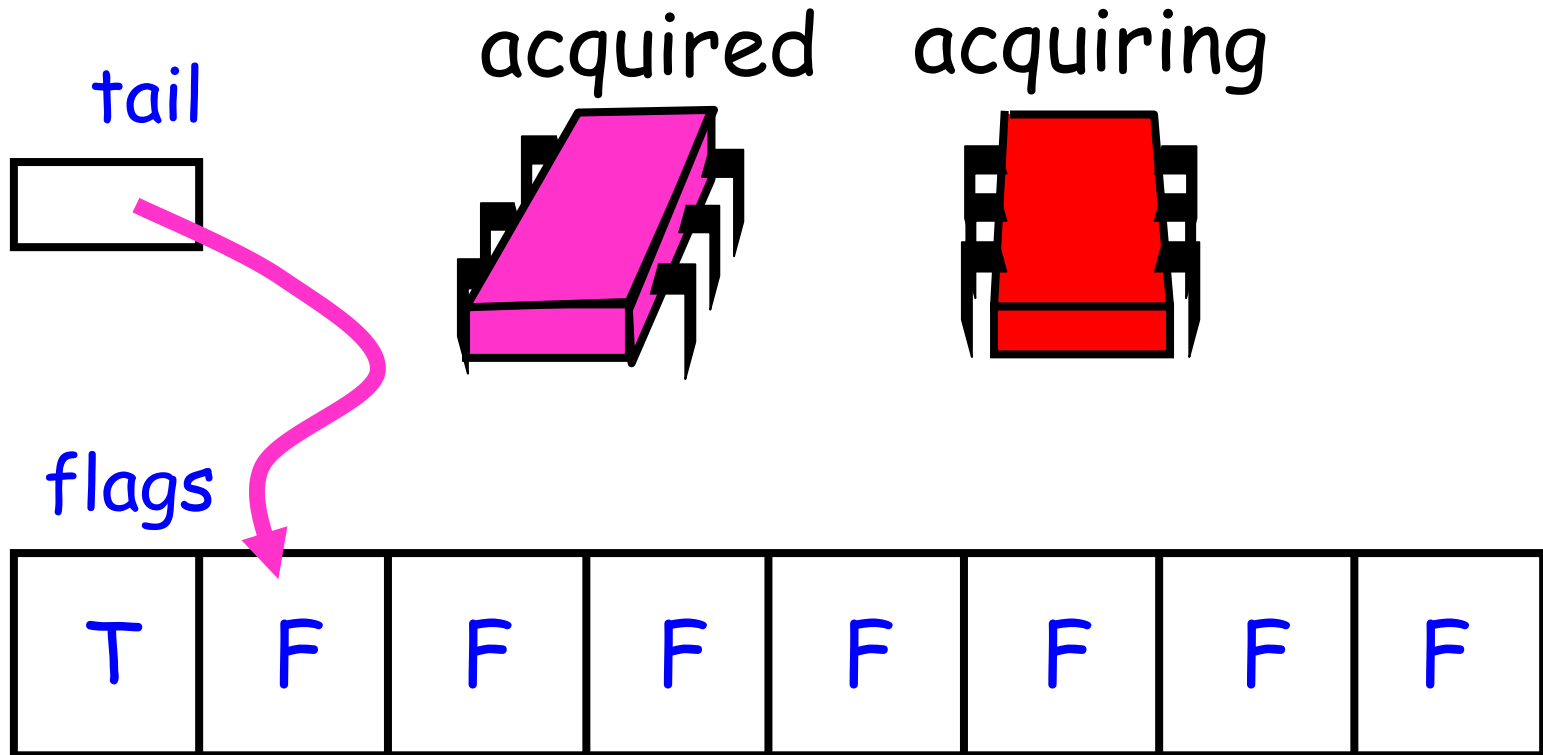
Anderson Queue Lock



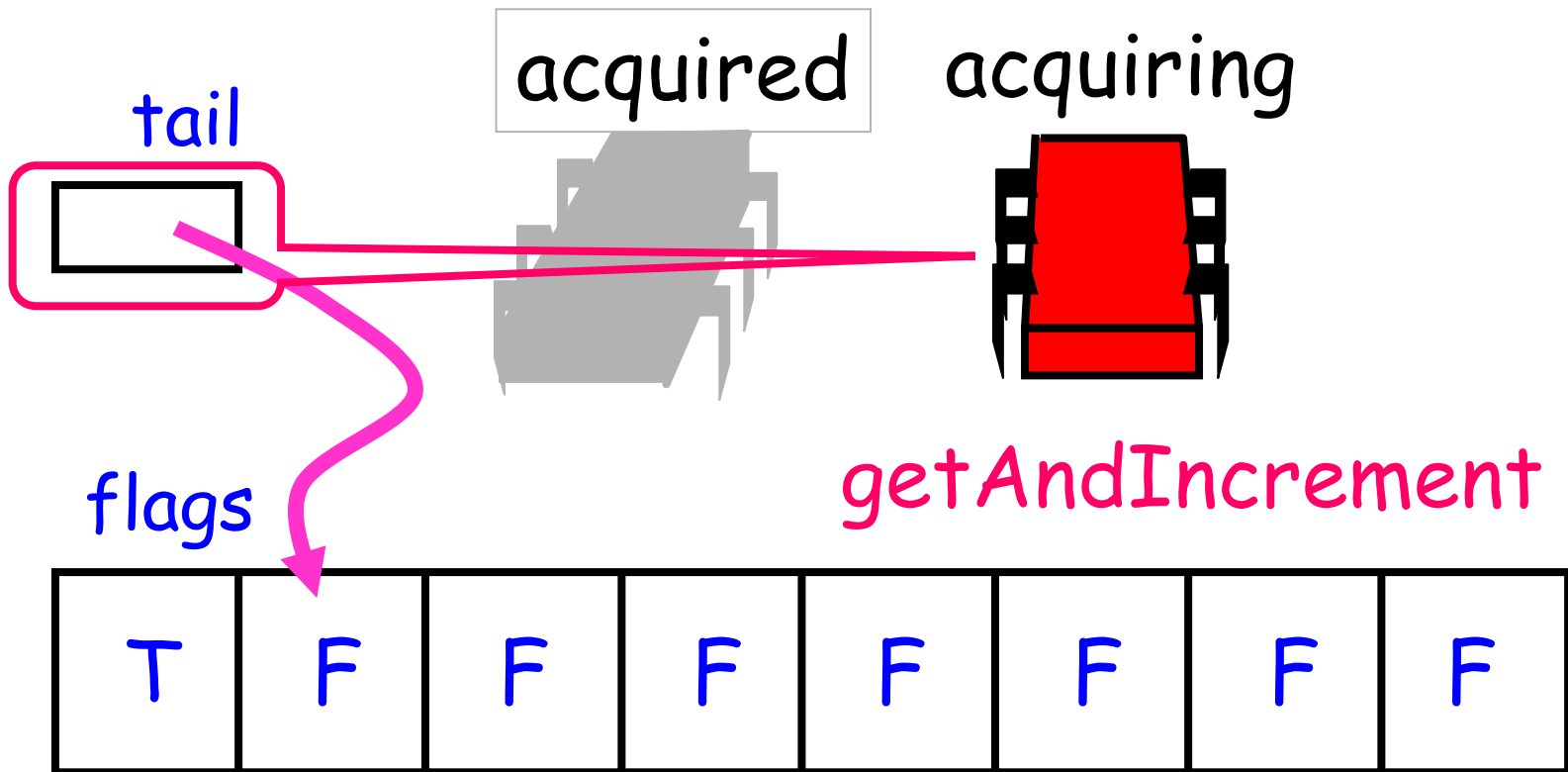
Anderson Queue Lock

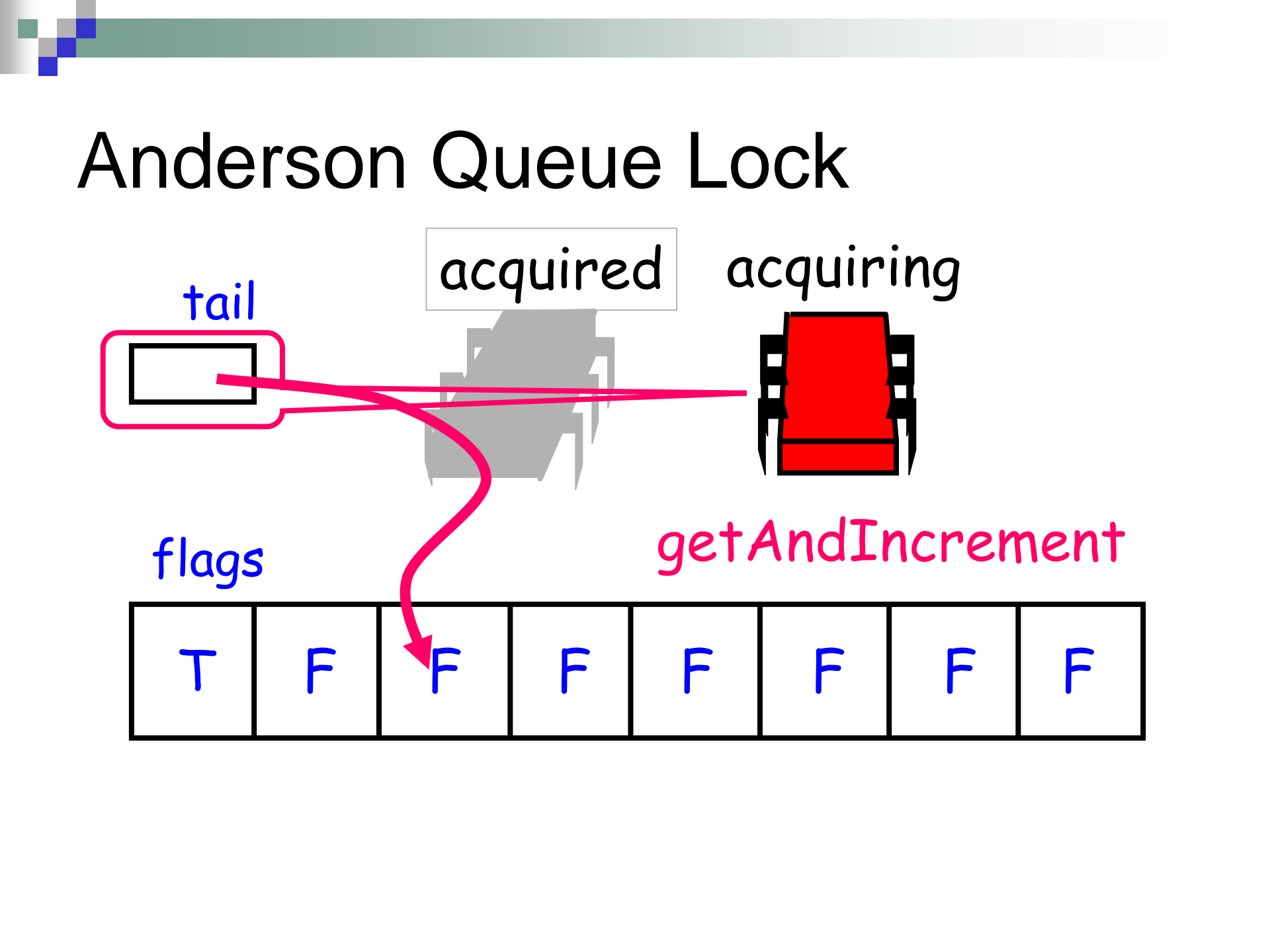


Anderson Queue Lock

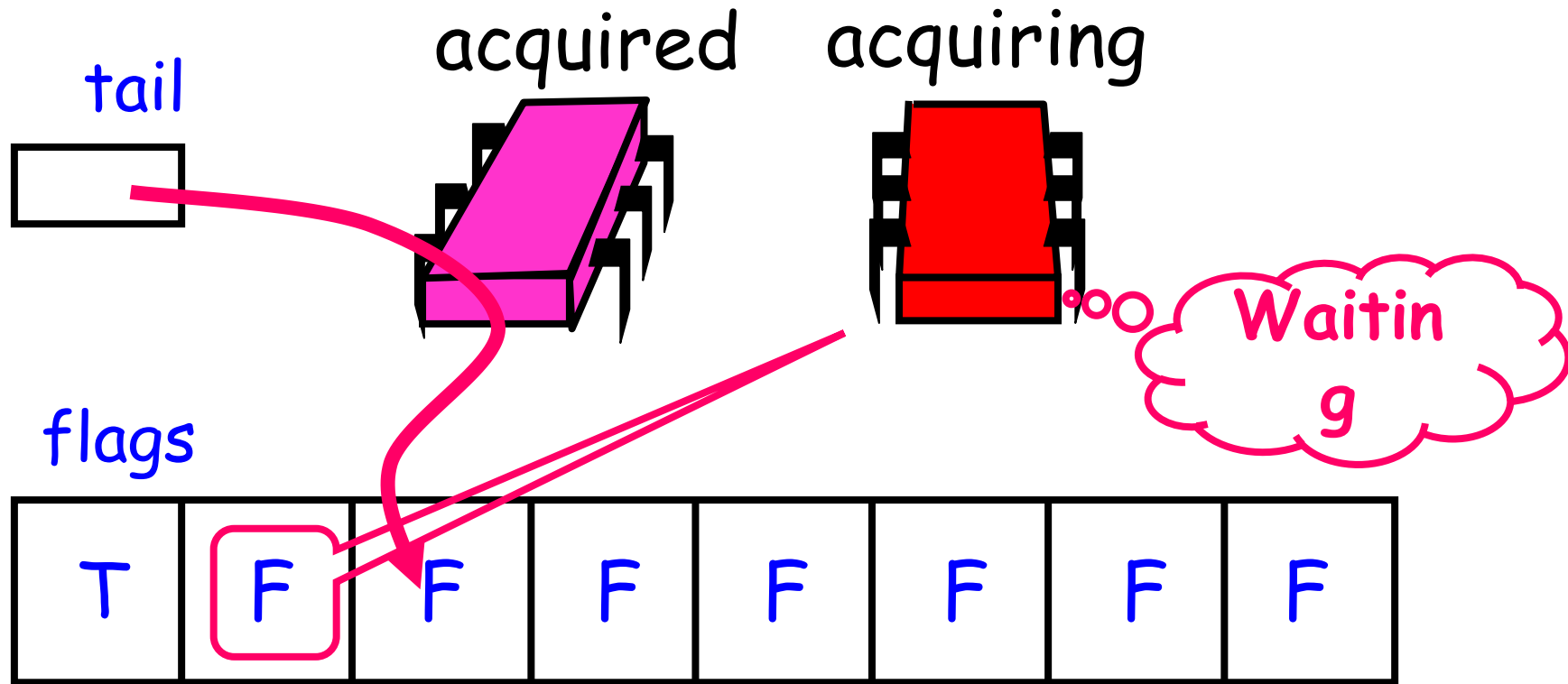


Anderson Queue Lock

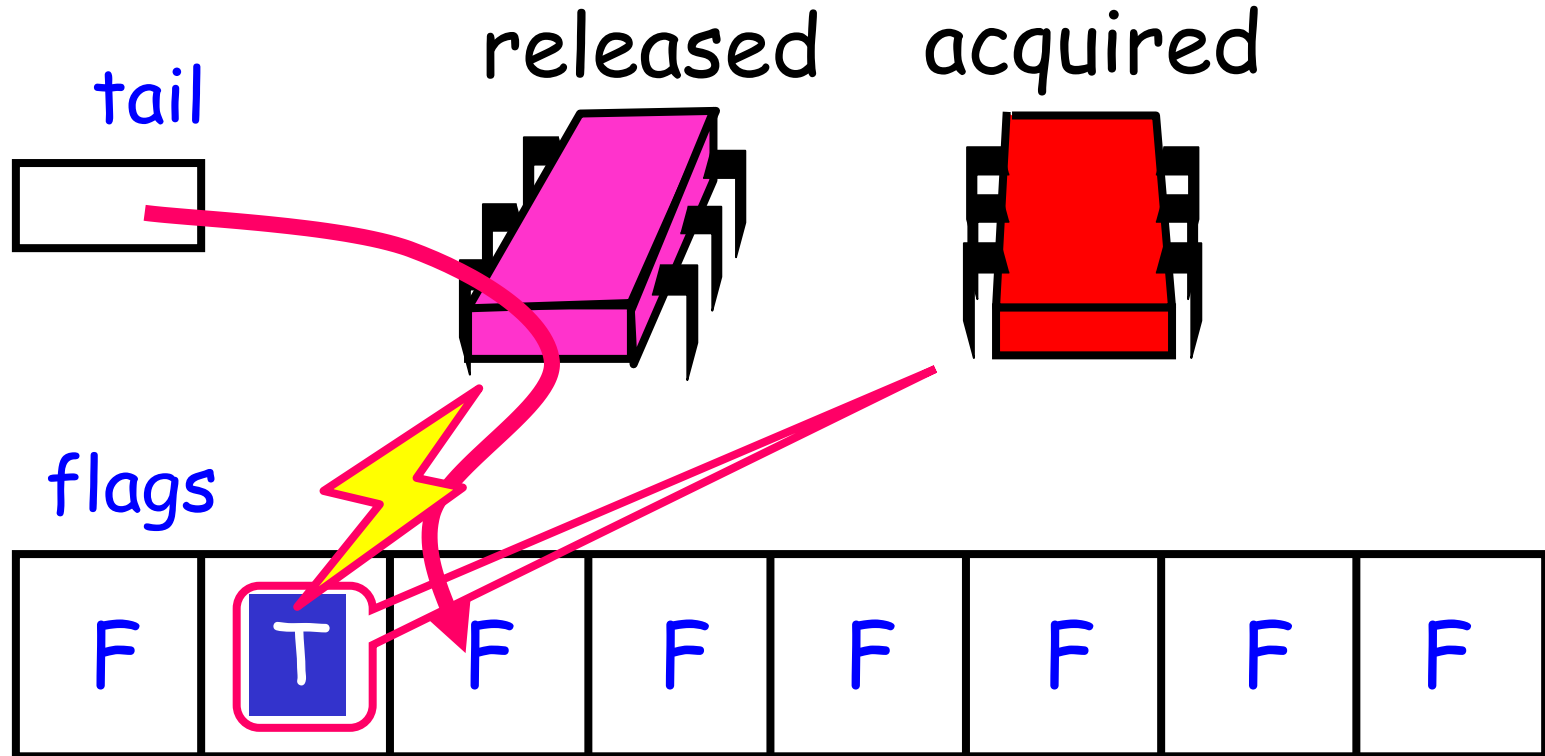




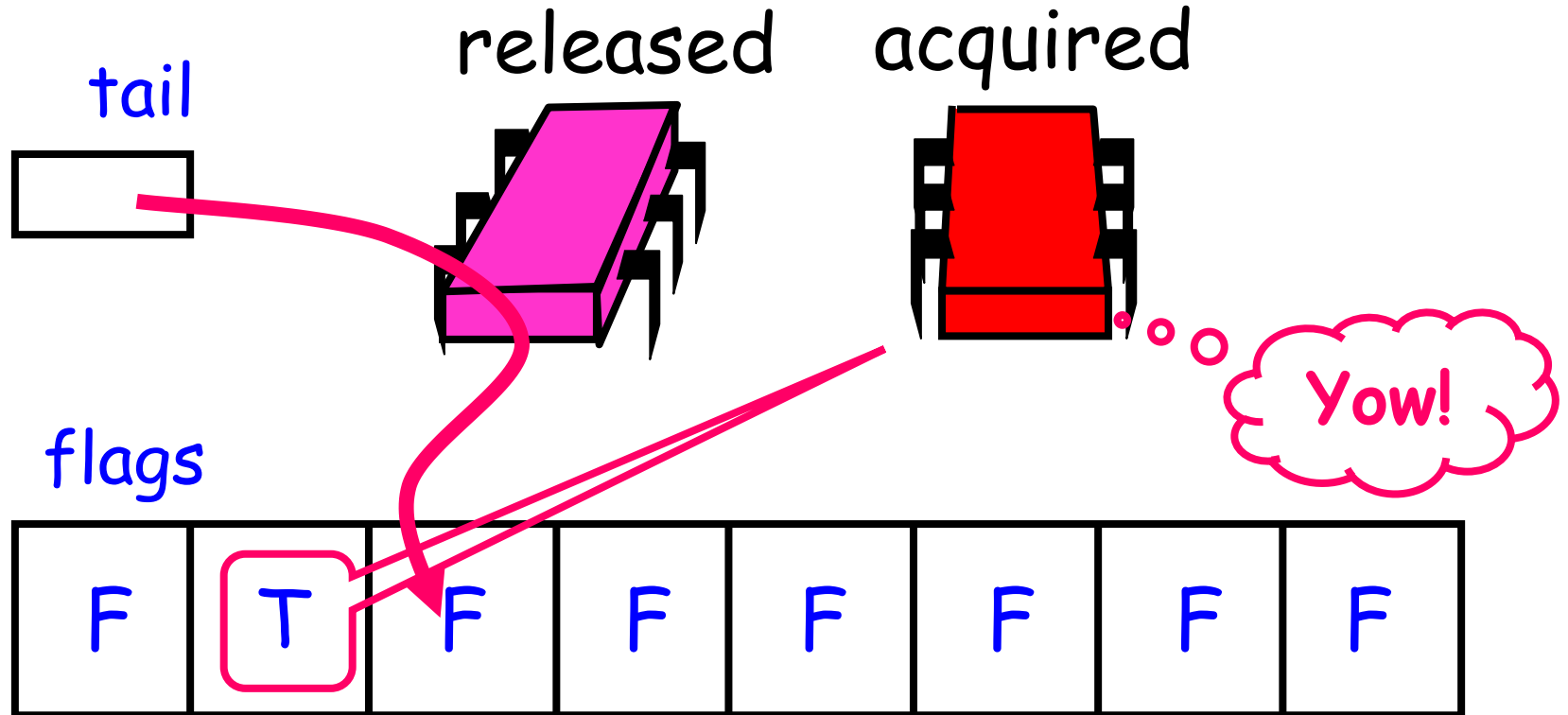
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock





Anderson Queue Lock

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

Anderson Queue Lock

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

One flag per thread

Anderson Queue Lock

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

Next flag to use

Anderson Queue Lock

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

Thread-local variable

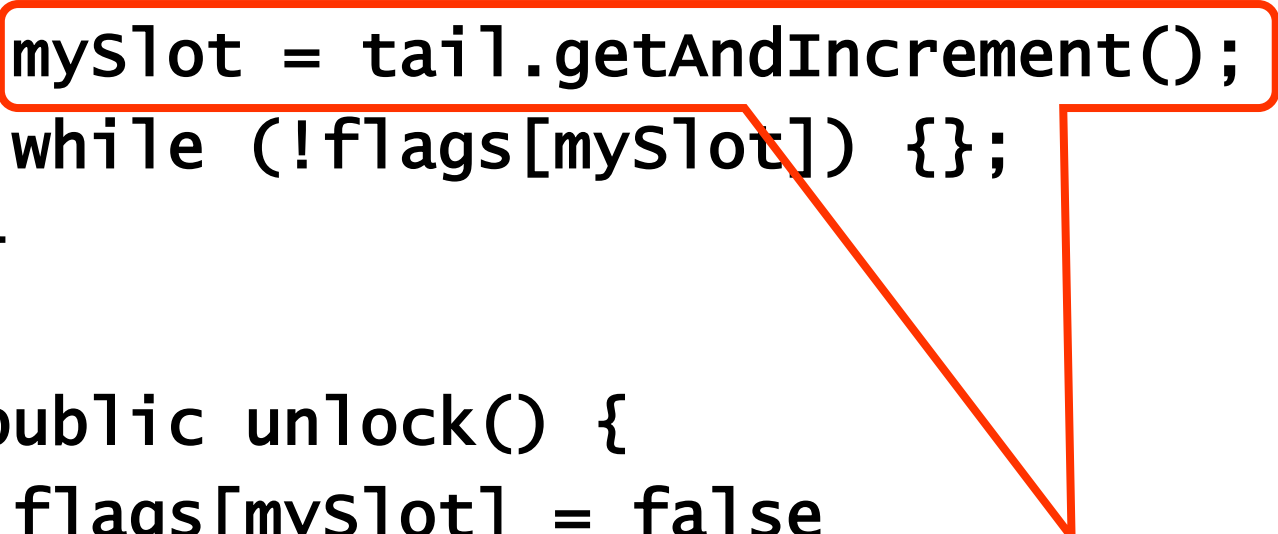


Anderson Queue Lock

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

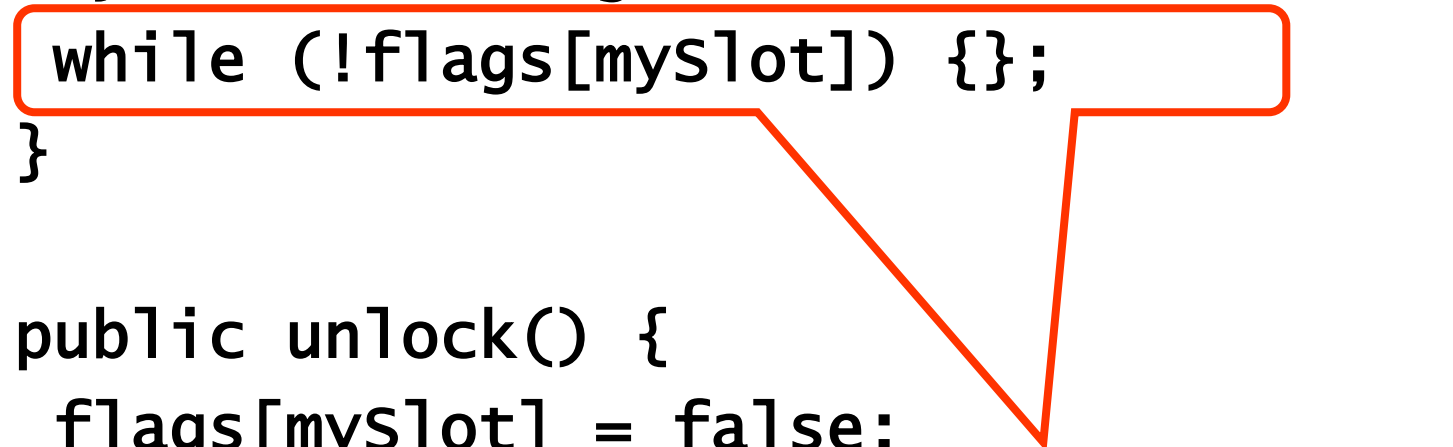
Anderson Queue Lock

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

A red line originates from the `mySlot` variable in the `lock()` method, extends to the right, and then diagonals down to point at the `mySlot` variable in the `unlock()` method, illustrating that the slot acquired during locking is the one being released during unlocking.

Anderson Queue Lock

```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot]) {};  
}  
  
public unlock() {  
    flags[mySlot] = false;  
    flags[(mySlot+1) % flags.length] = true;  
    // Spin until told to go  
}
```

A red rectangular box highlights the while loop in the lock() method. A red line extends from the bottom of this box, pointing towards the comment 'Spin until told to go' in the unlock() method, indicating that the thread spins during this period.

Anderson Queue Lock

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

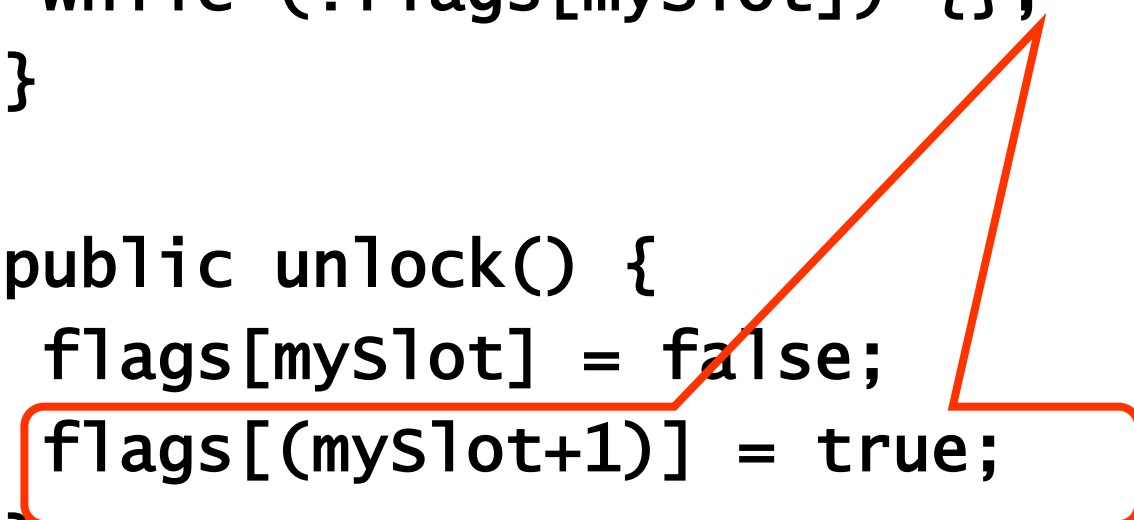
```
public unlock() {  
    flags[myslot] = false;  
    flags[(myslot+1)] = true;  
}
```

Prepare slot for re-use

Anderson Queue Lock

```
public lock() {  
    mySlot = nextSlot;  
    while (!flags[mySlot]) {}  
}  
  
public unlock() {  
    flags[mySlot] = false;  
    flags[(mySlot+1)] = true;  
}
```

Tell next thread to go

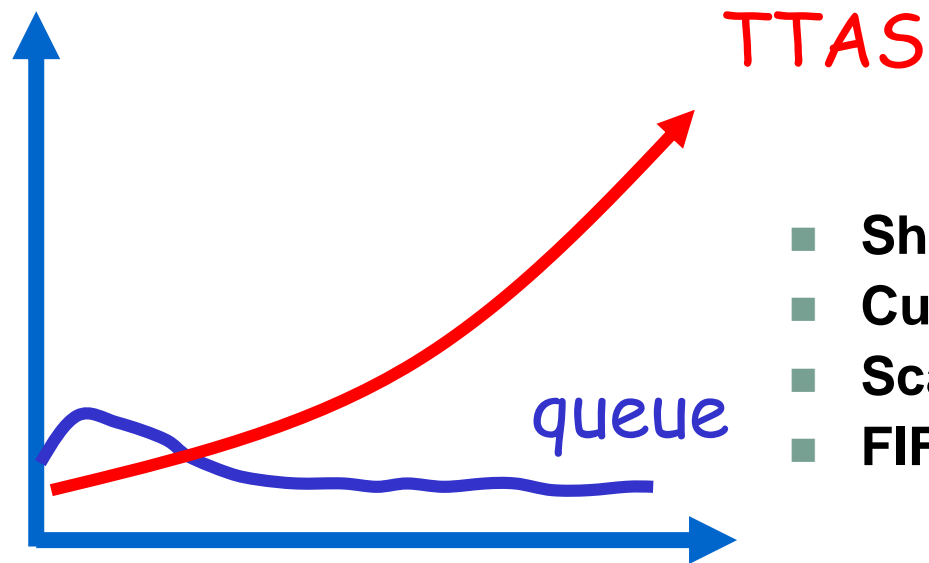




Anderson Queue Lock

- Although the flags [] array is shared, contention on the array locations are minimised since each thread spins on its own locally cached copy of a single array location

Performance



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



Anderson Queue Lock

- Good

- ☐ First truly scalable lock
- ☐ Simple, easy to implement

- Bad

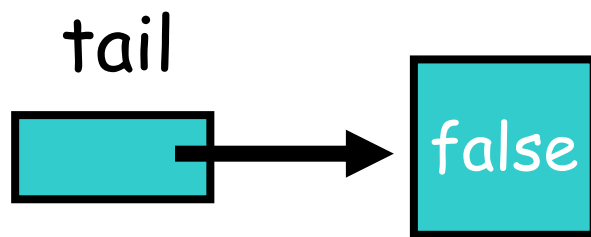
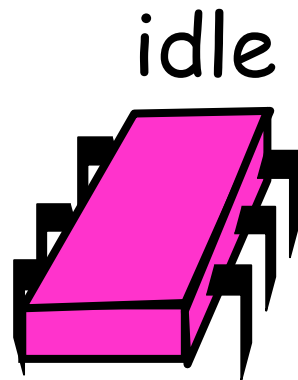
- ☐ Not space efficient
- ☐ One bit per thread
 - Unknown number of threads?
 - Small number of actual contenders?



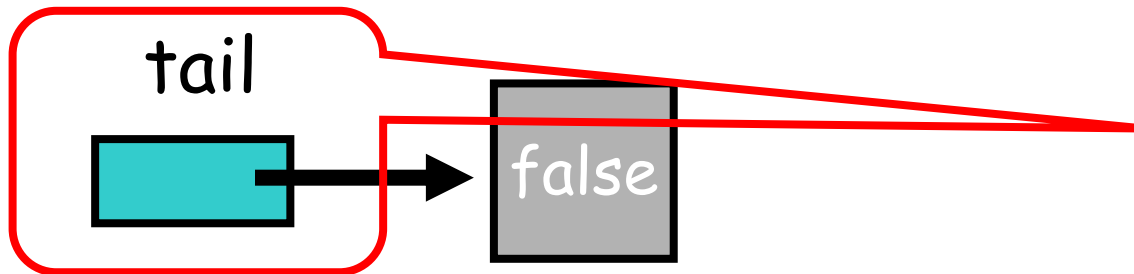
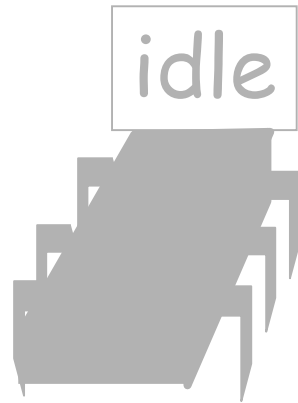
CLH Queue Lock

- Virtual Linked List keeps track of the queue
- Each thread's status is saved in its node:
 - ☐ True – has acquired the lock or wants to acquire the lock
 - ☐ False – is finished with the lock and has released it
- Each node keeps track of its predecessors status

Initially

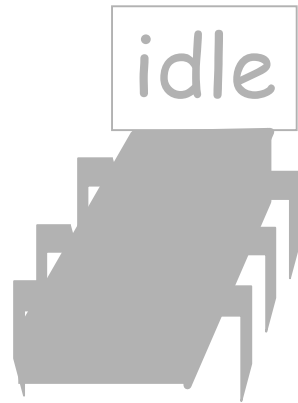


Initially

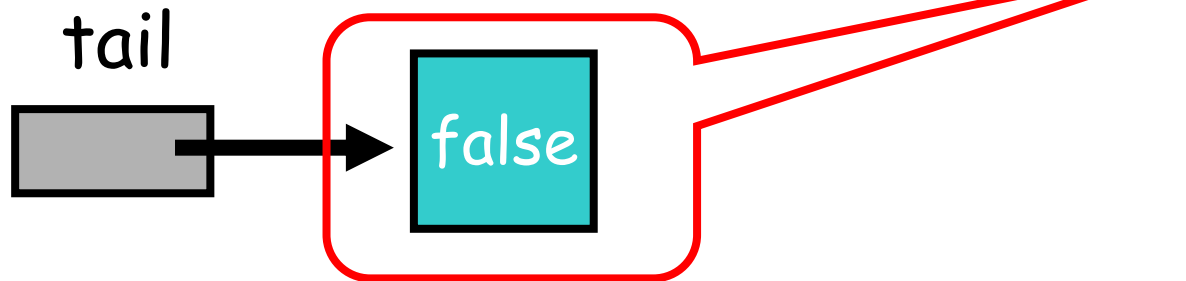


Queue tail

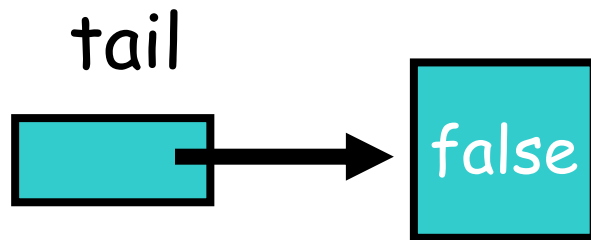
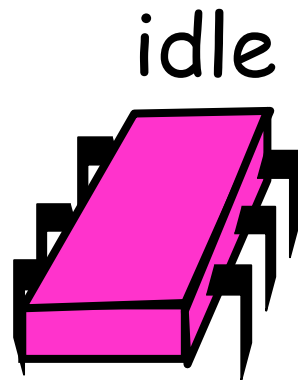
Initially



Locked field: Lock is free

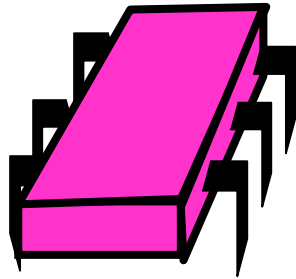


Initially



Purple Wants the Lock

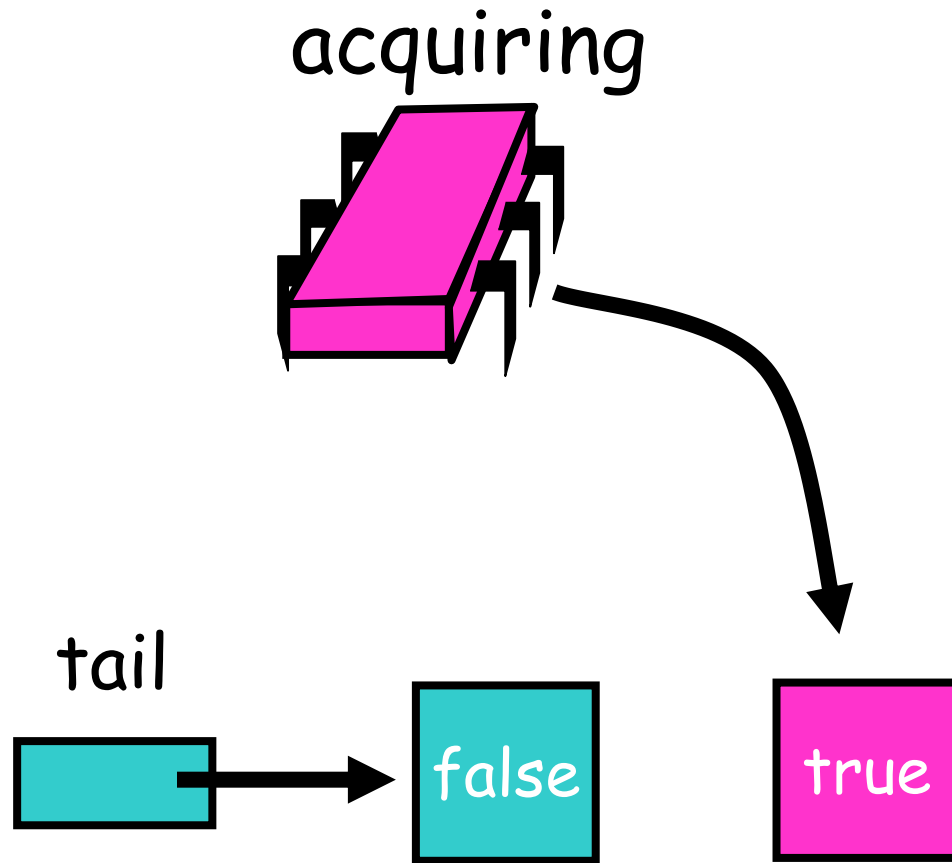
acquiring



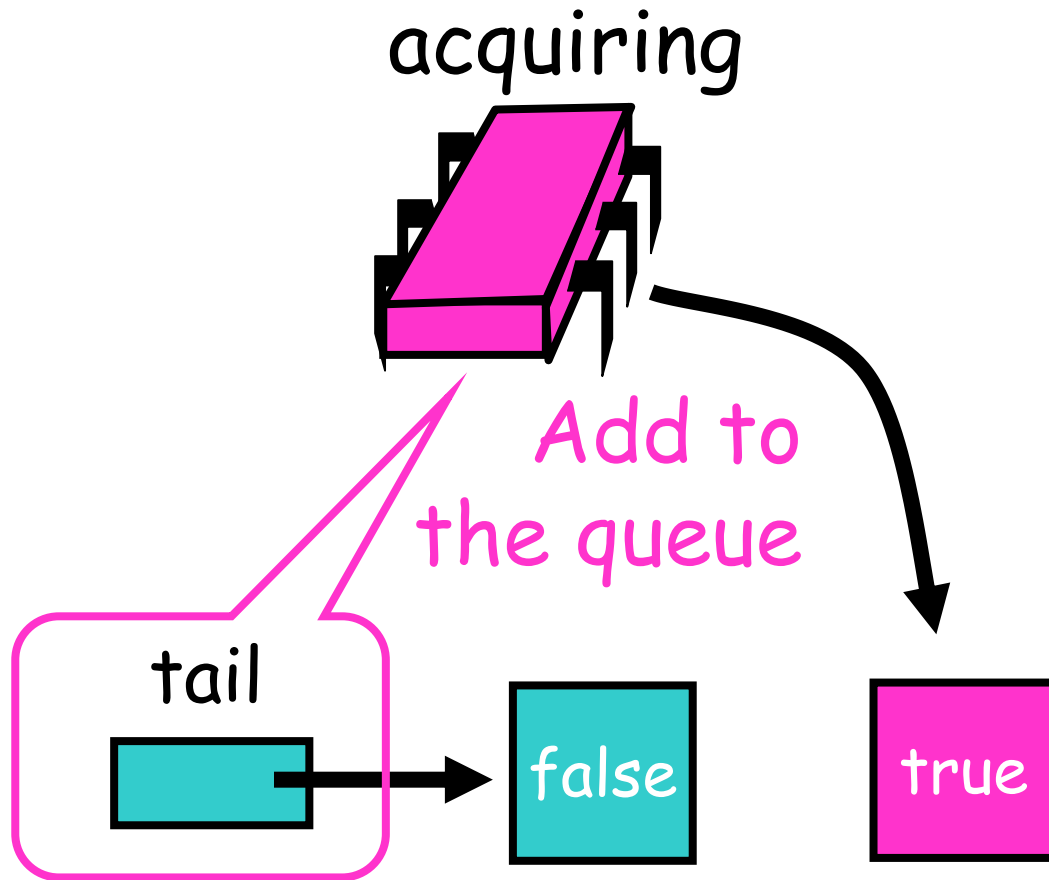
tail



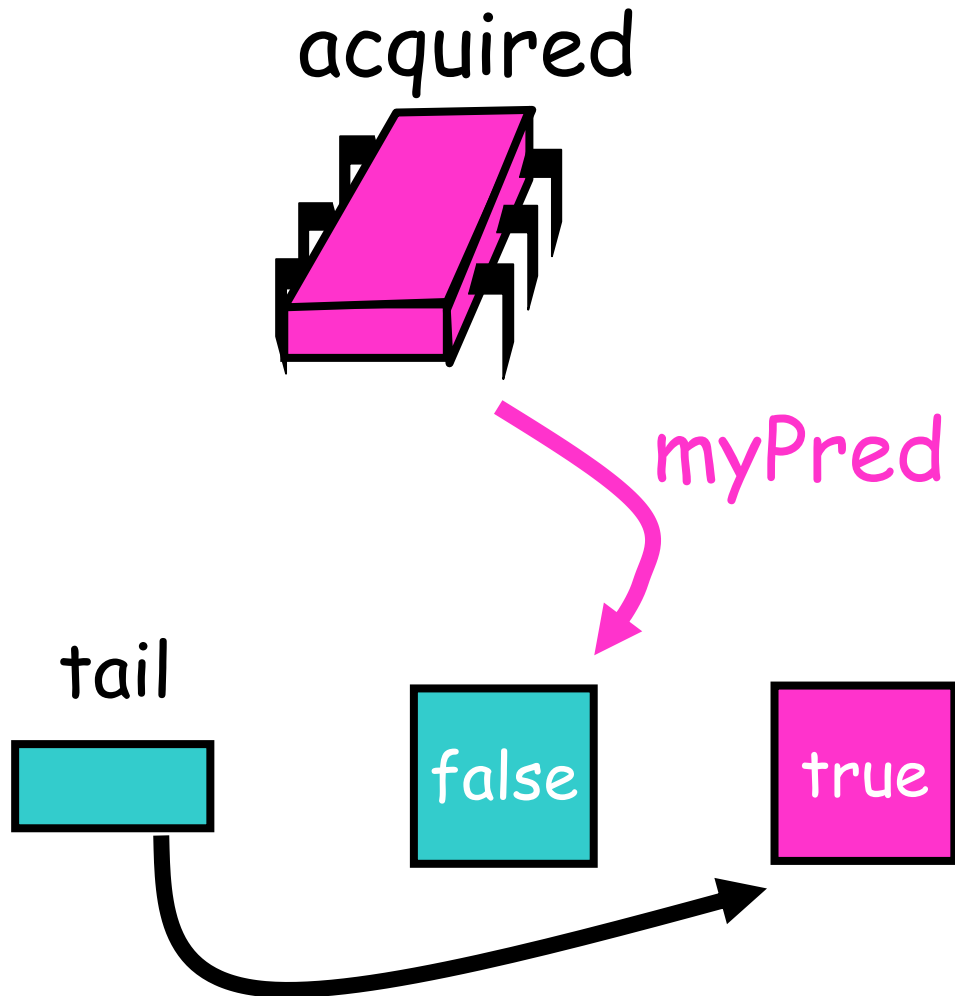
Purple Wants the Lock



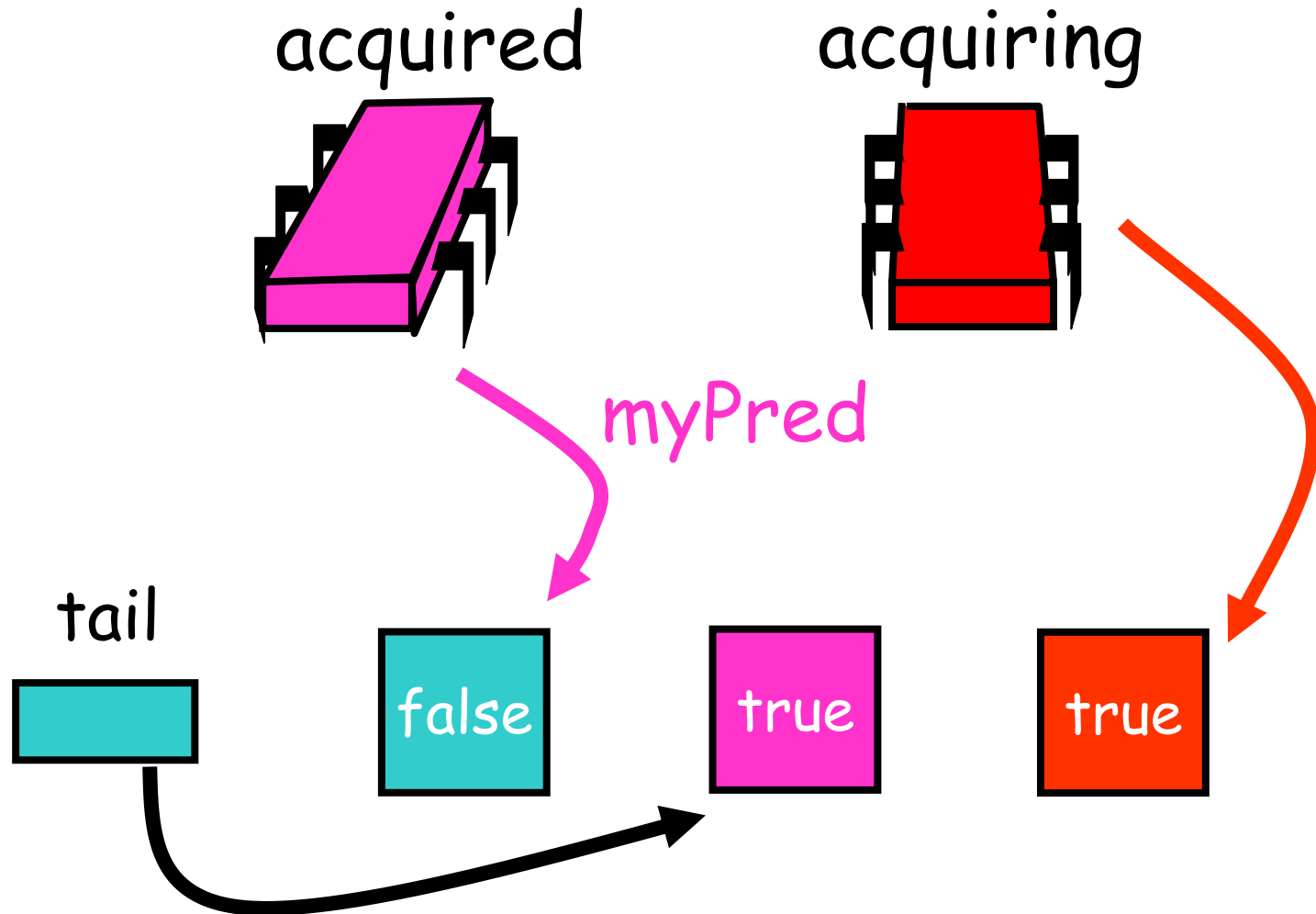
Purple Wants the Lock



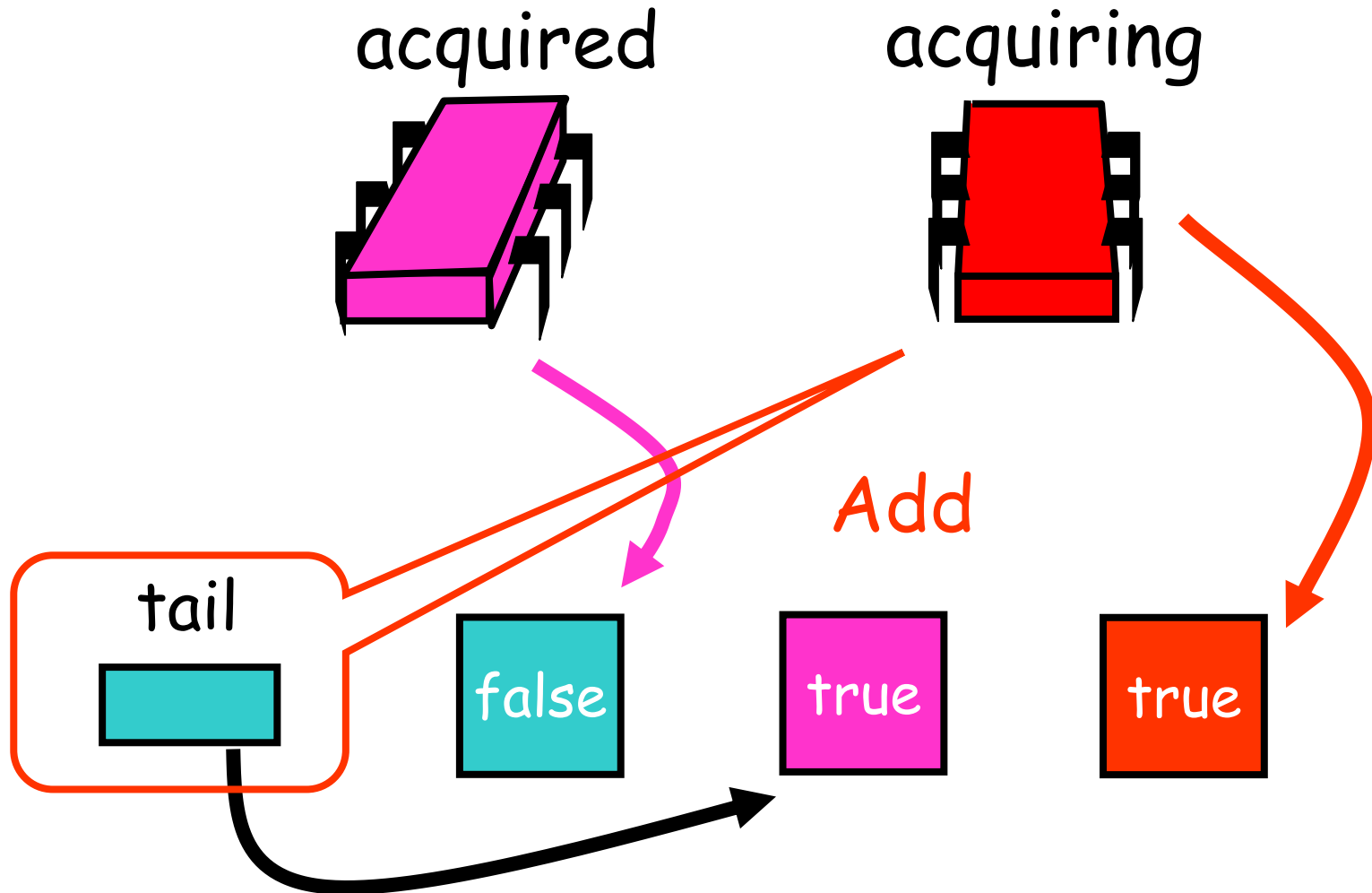
Purple Has the Lock



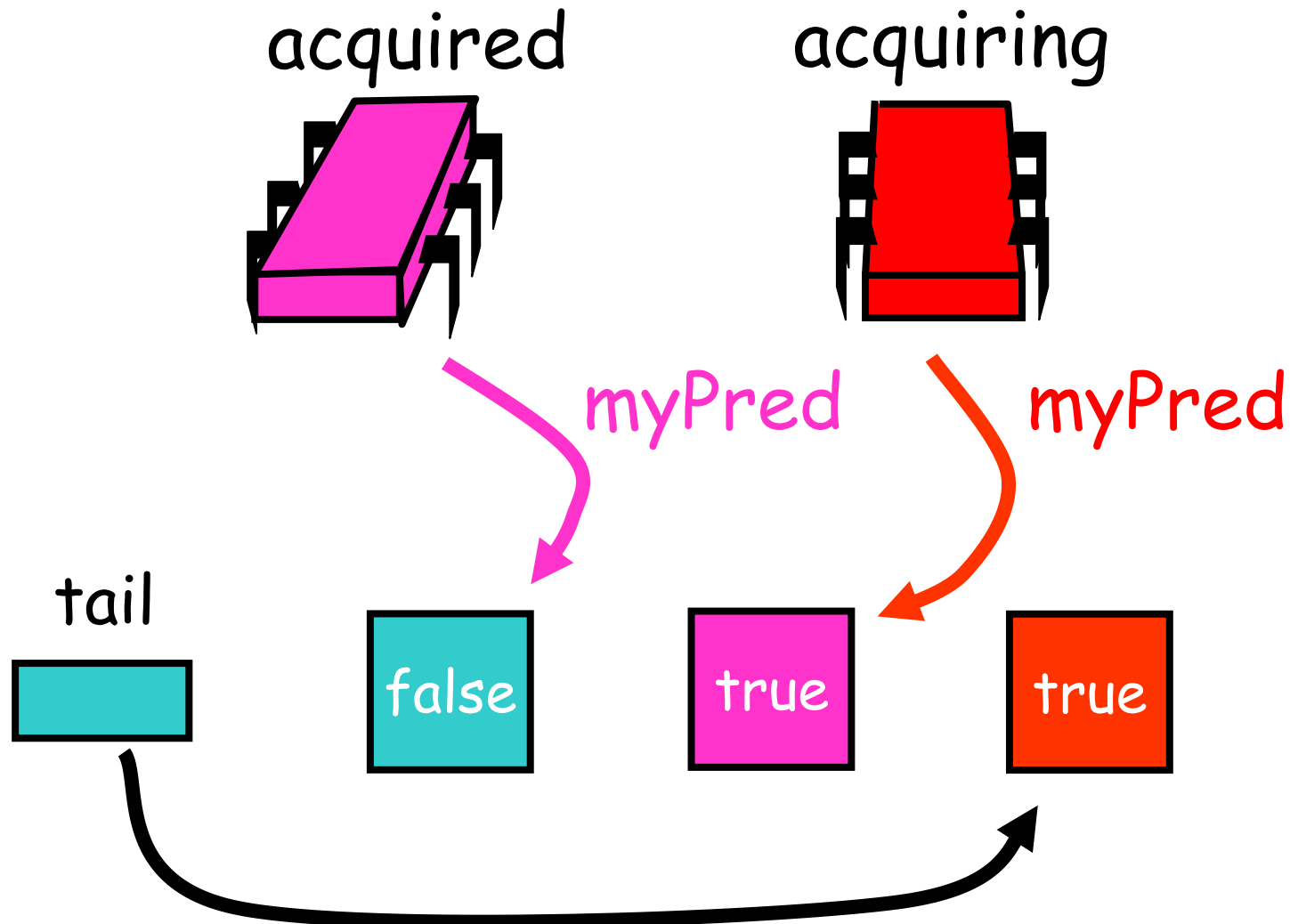
Red Wants the Lock



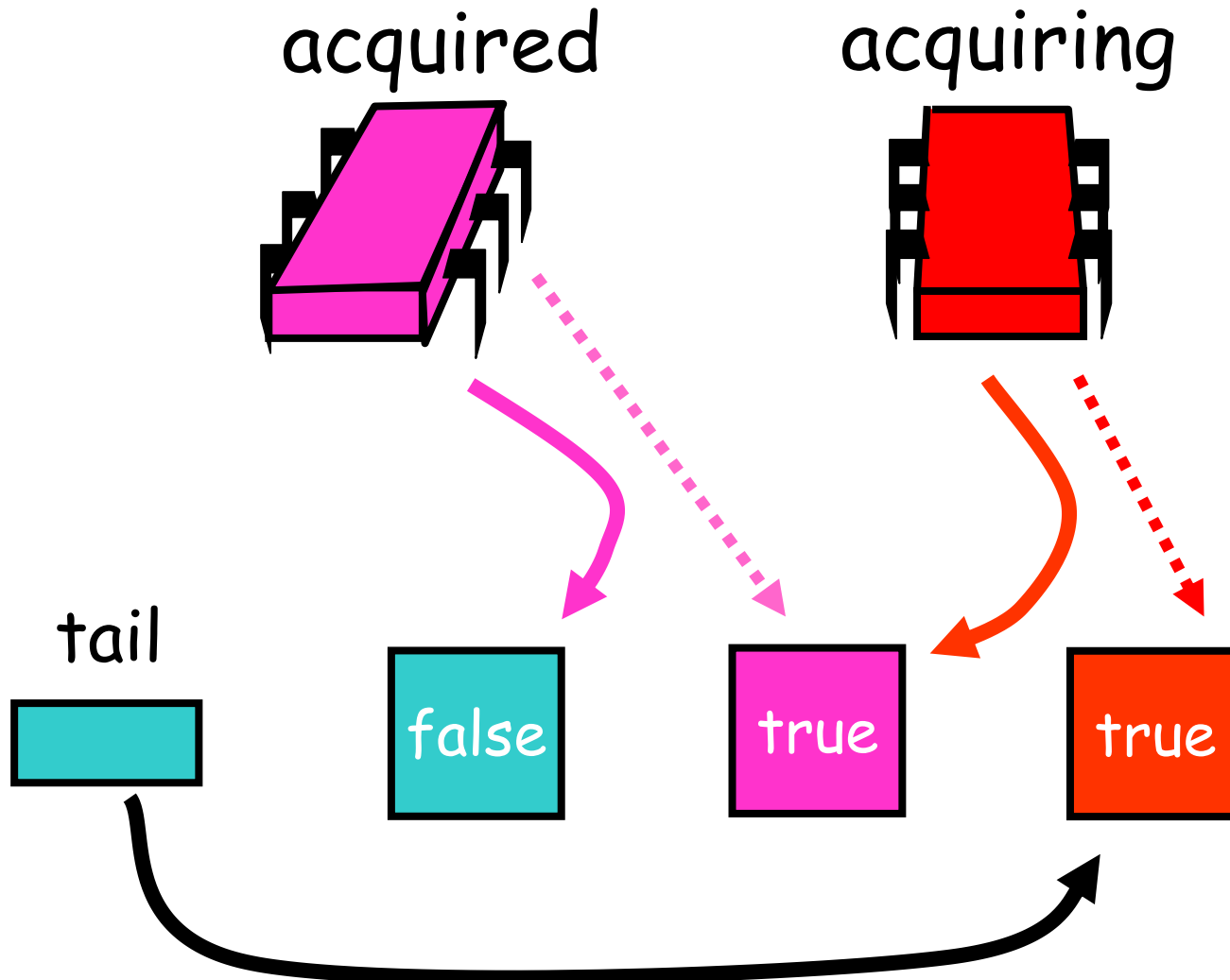
Red Wants the Lock



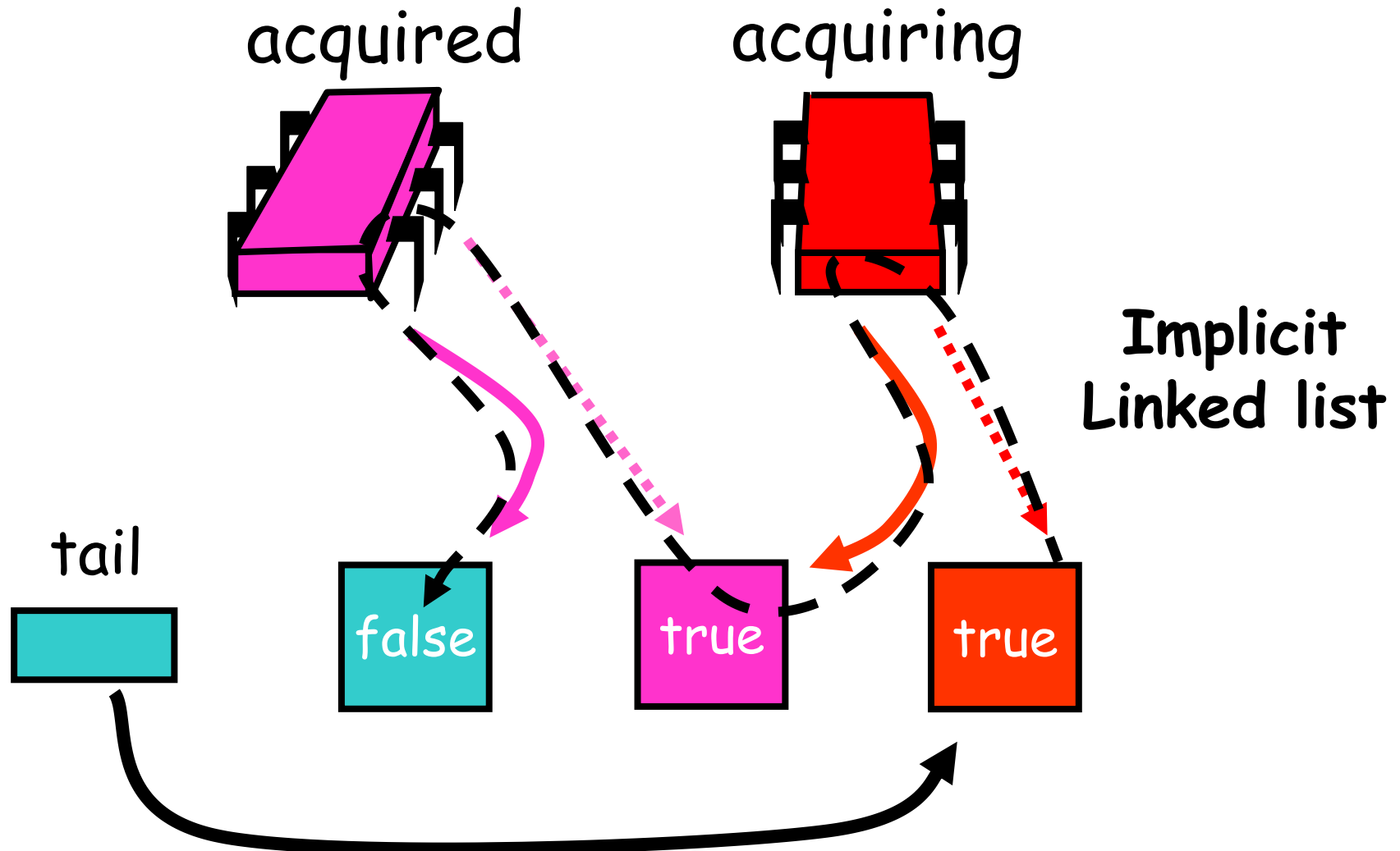
Red Wants the Lock



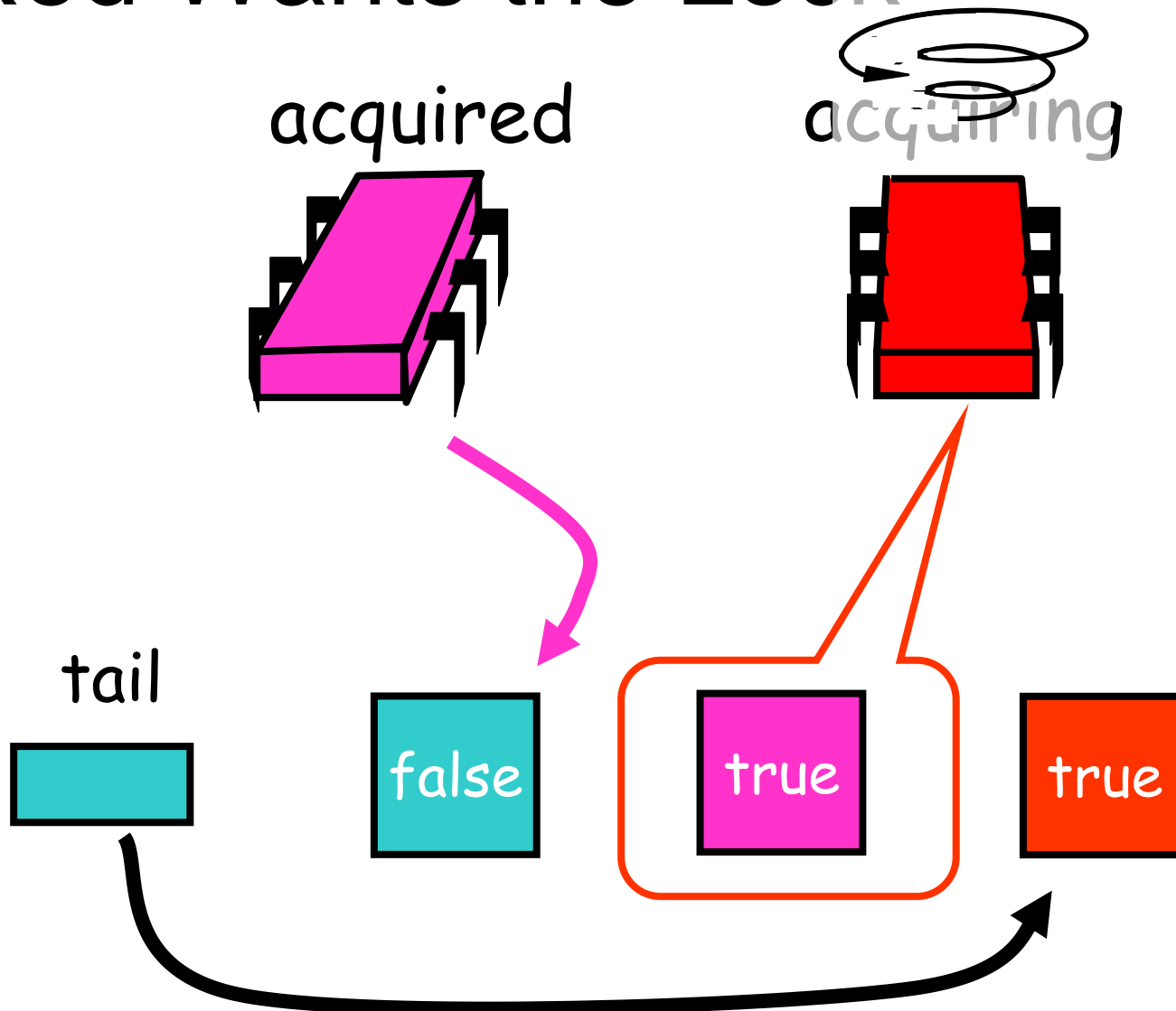
Red Wants the Lock



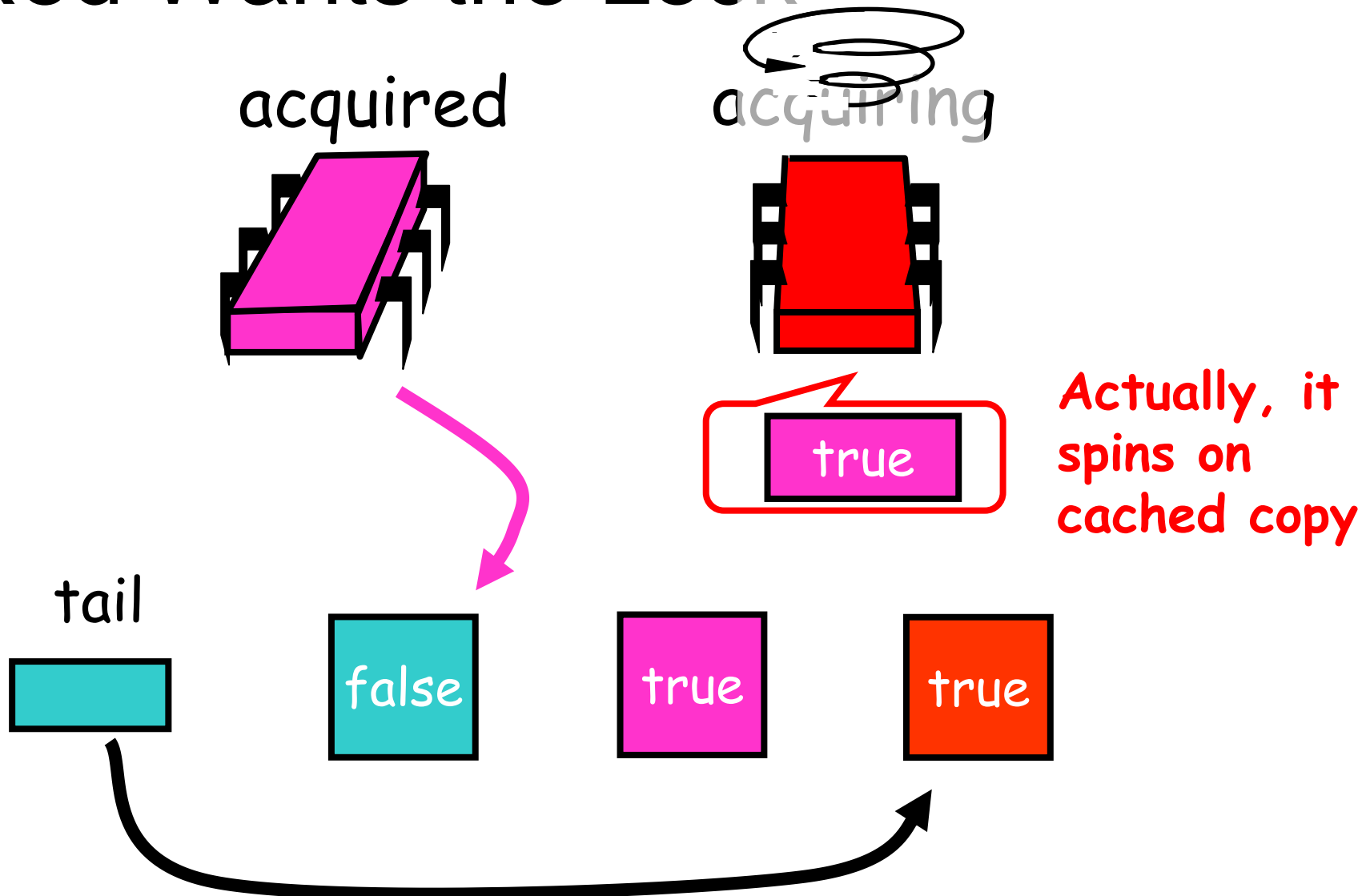
Red Wants the Lock



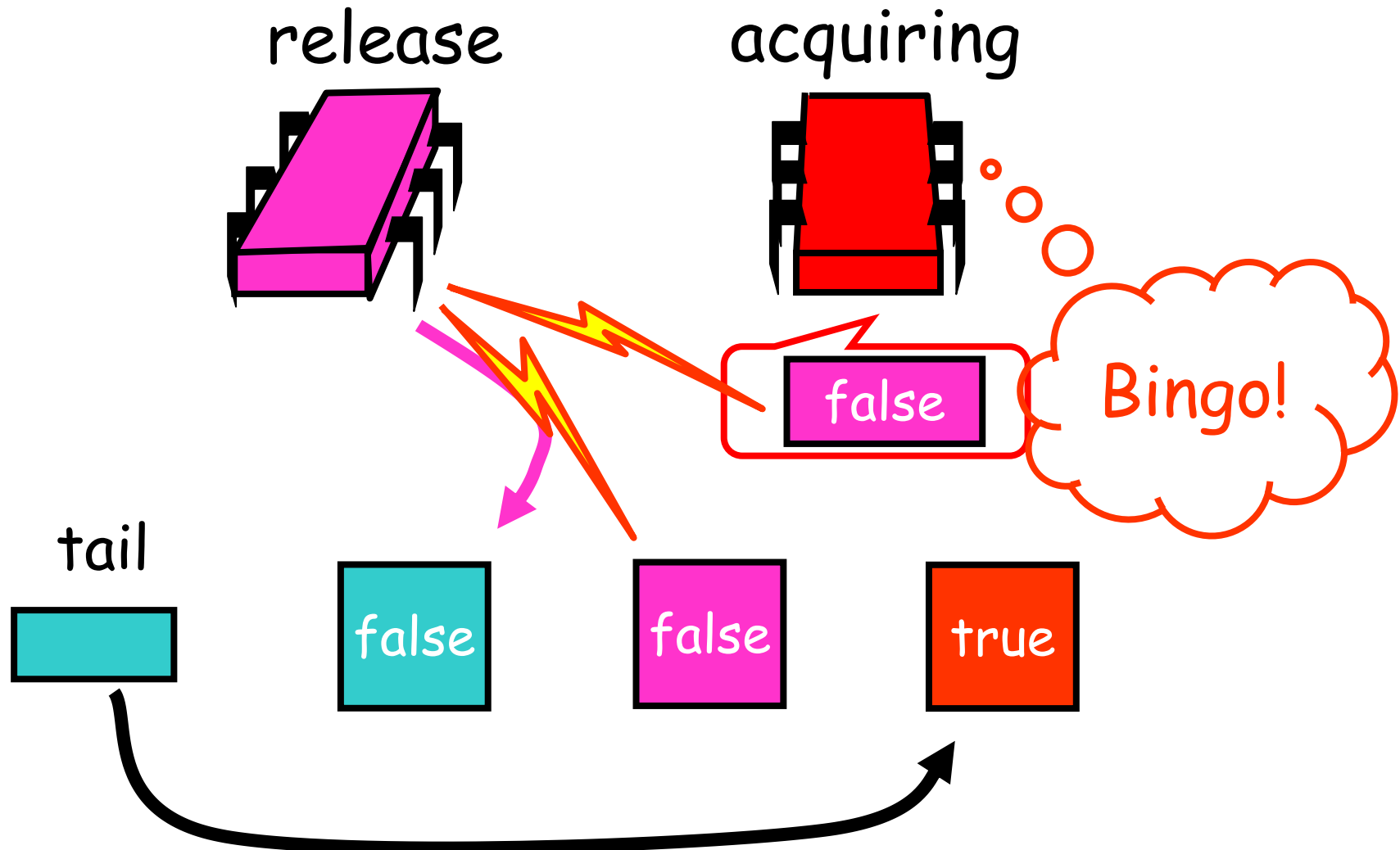
Red Wants the Lock



Red Wants the Lock

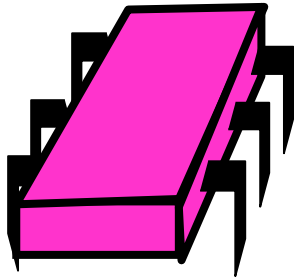


Purple Releases

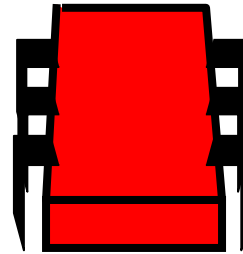


Purple Releases

released



acquired



tail



true





CLH Queue Lock

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

Add my node

CLH Queue Lock

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

- Good

- ☐ Lock release affects predecessor only
- ☐ Small, constant-sized space



MCS Lock

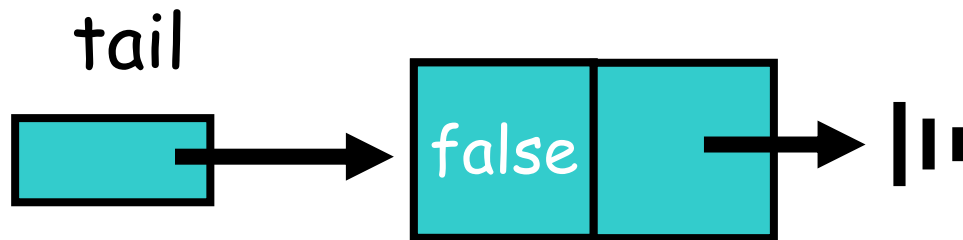
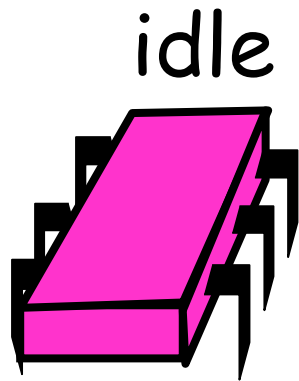
- FIFO order
- Spin on local memory only
- Small, Constant-size overhead



MCS Queue Lock

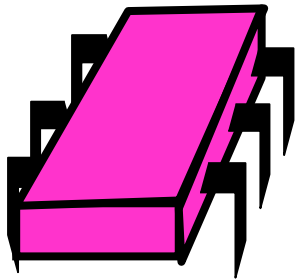
- Similar to CLHLock, but the linked list is explicit instead of implicit
- Each node in the Queue has a next field

Initially

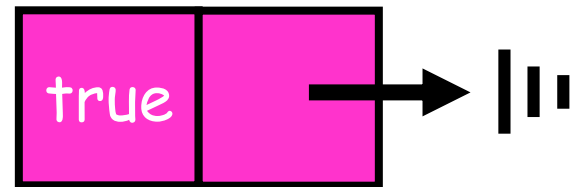


Acquiring

acquiring



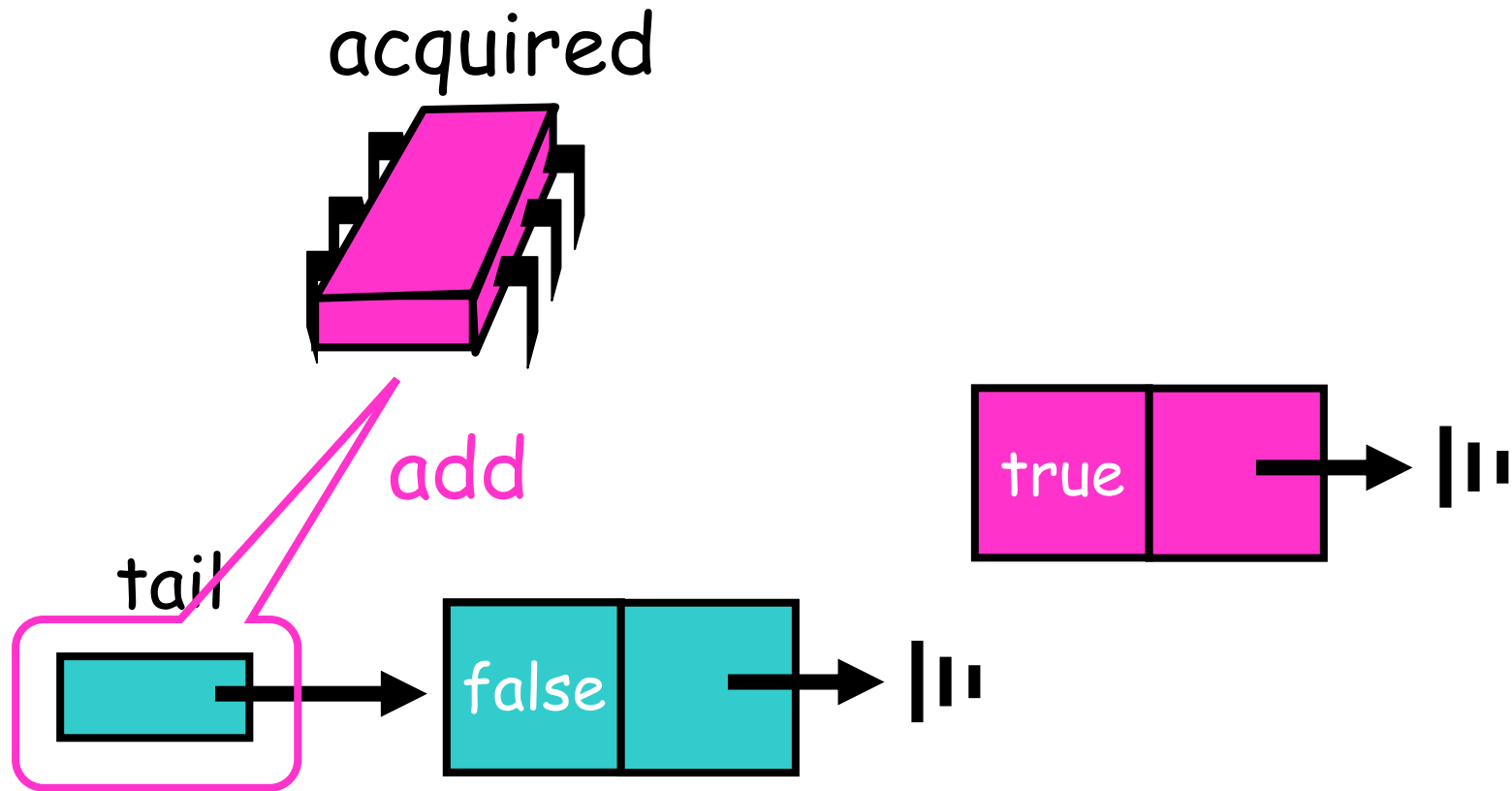
(add Qnode)



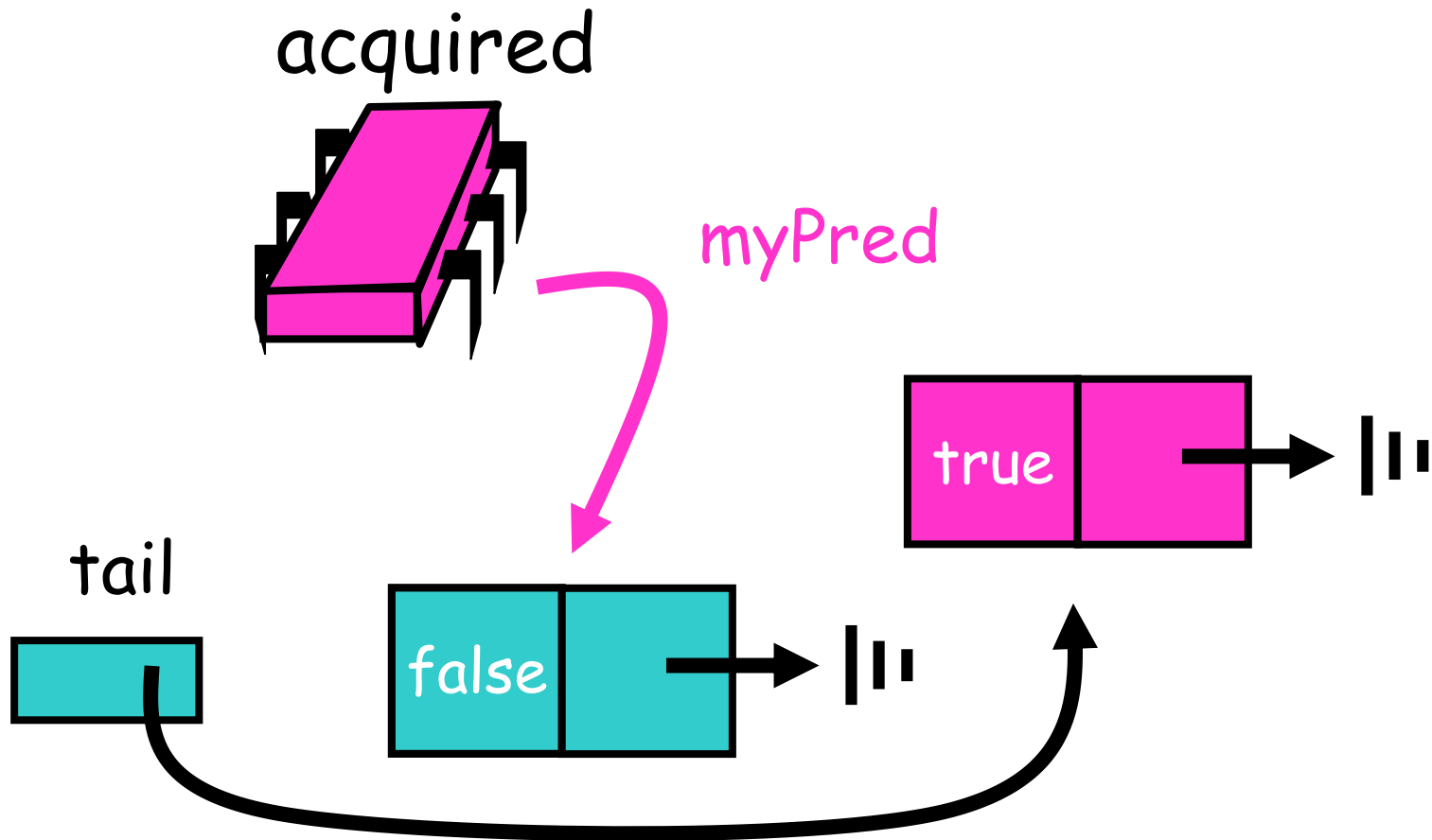
tail



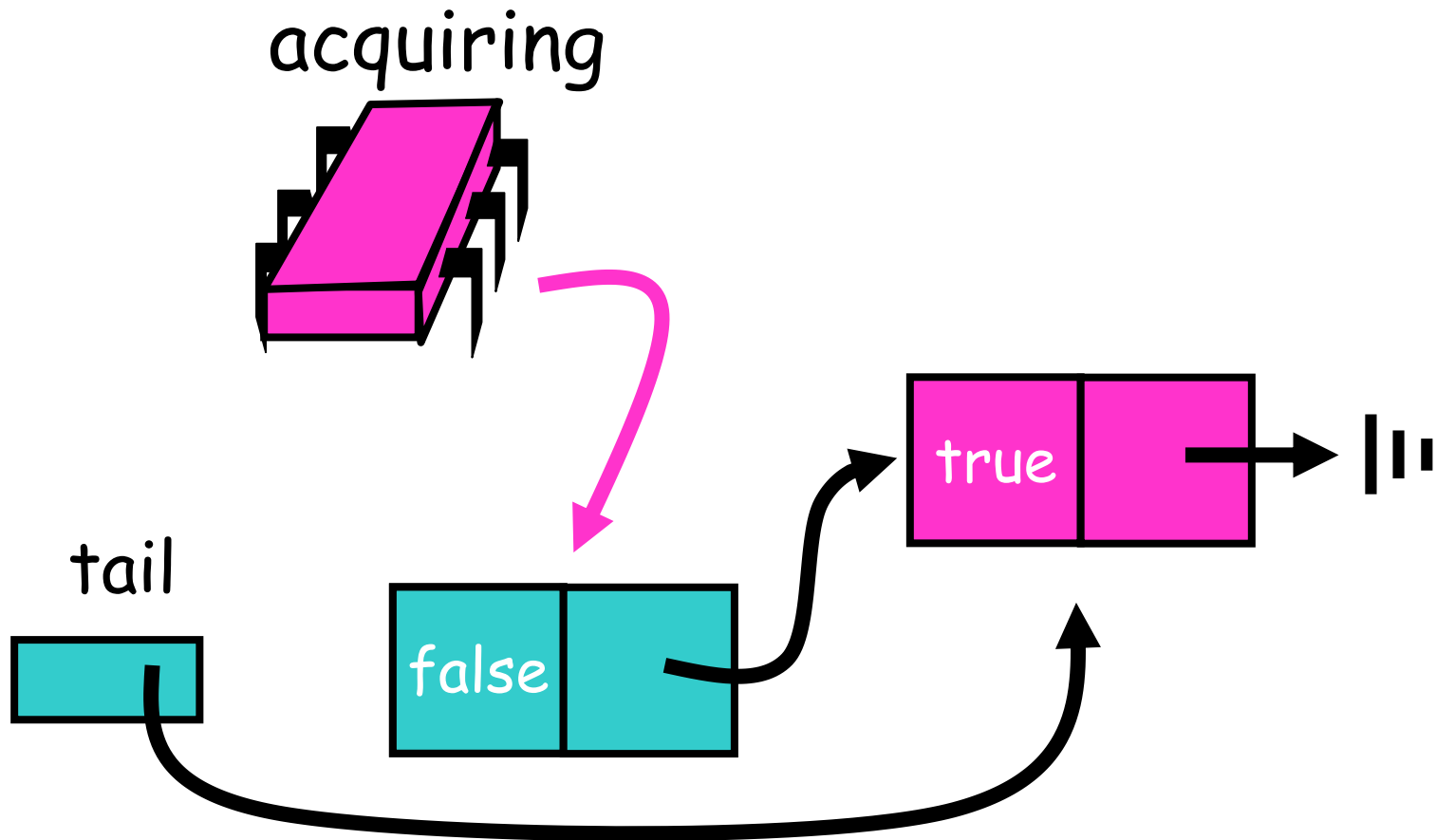
Acquiring



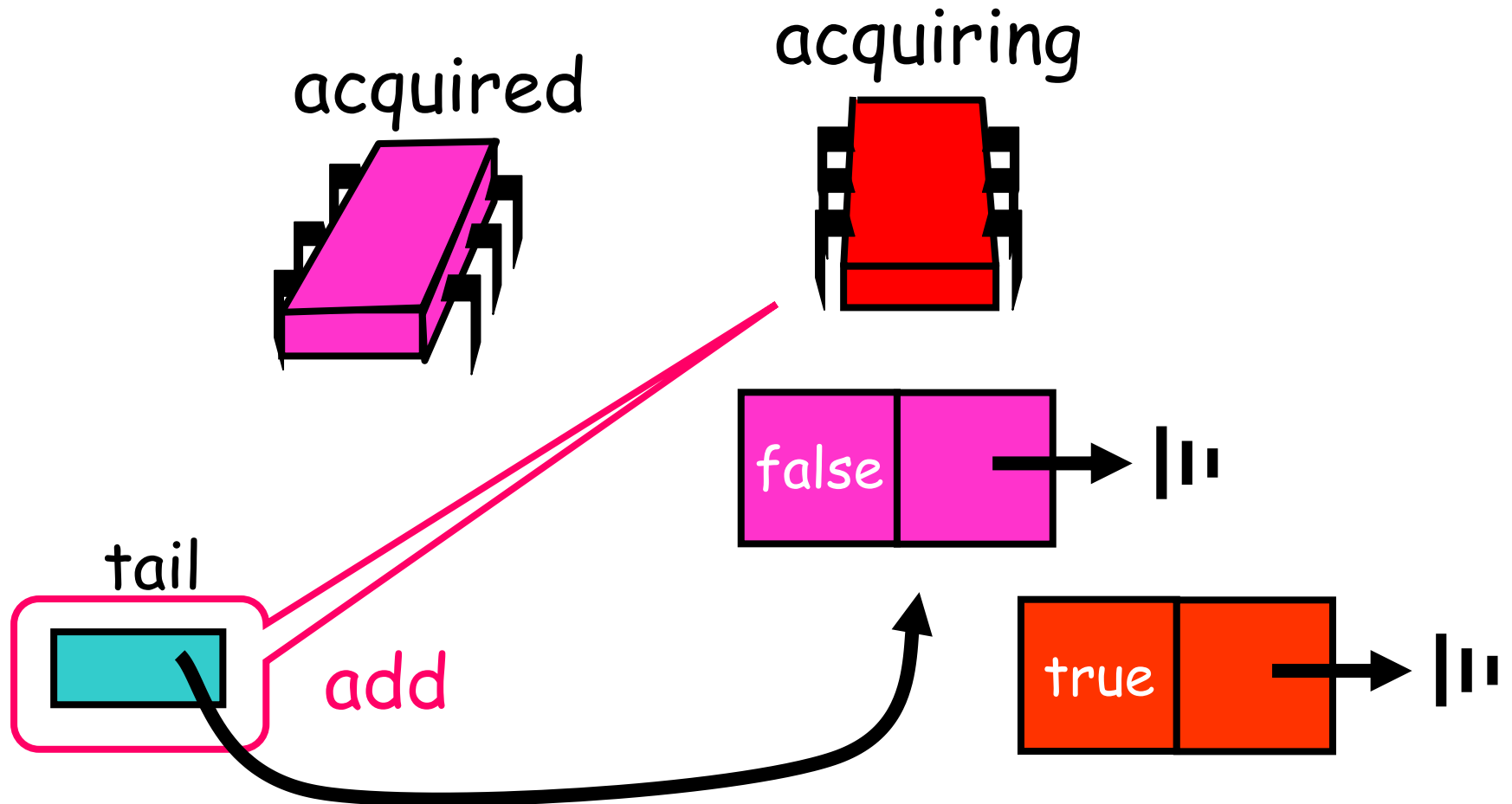
Acquiring



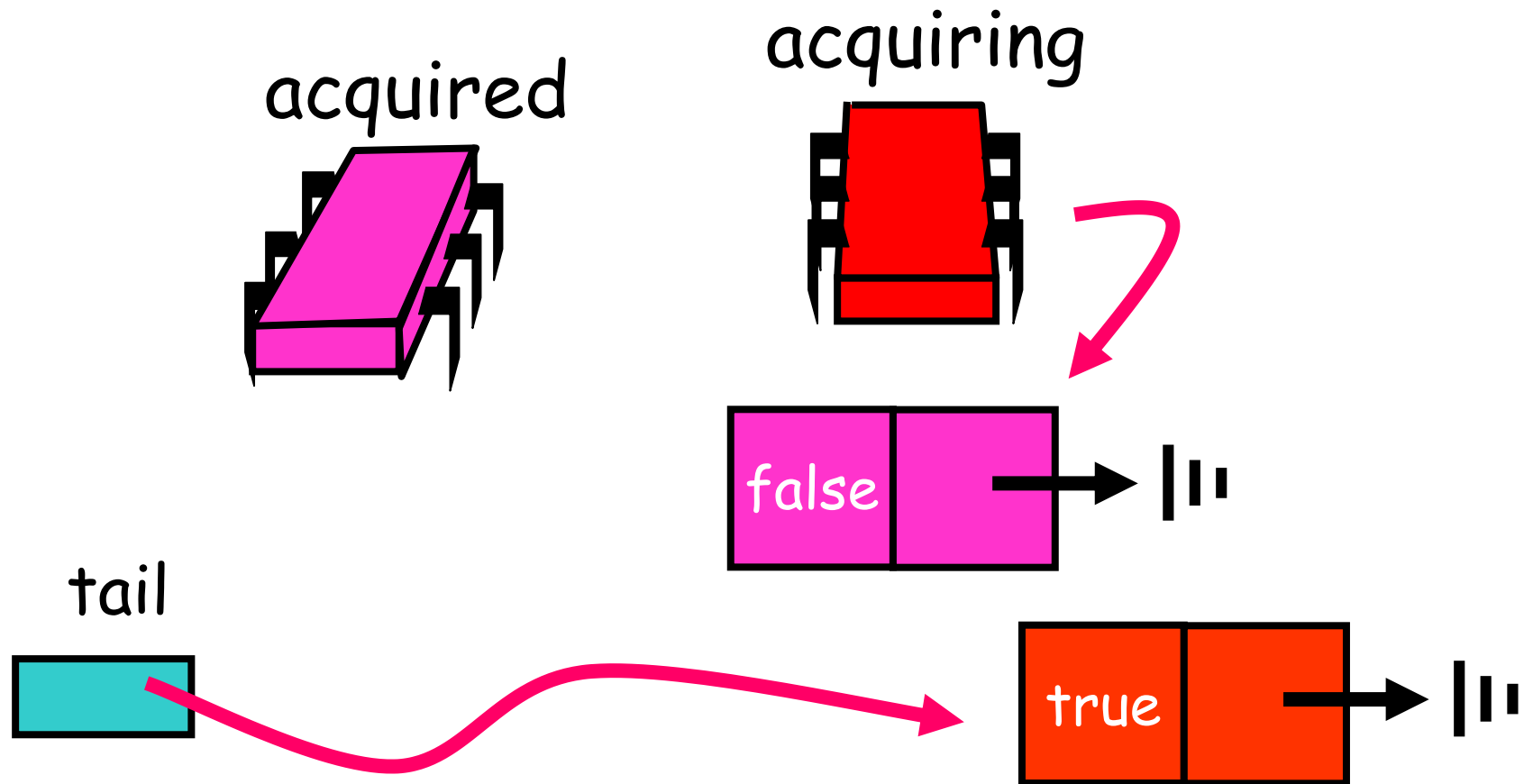
Acquiring



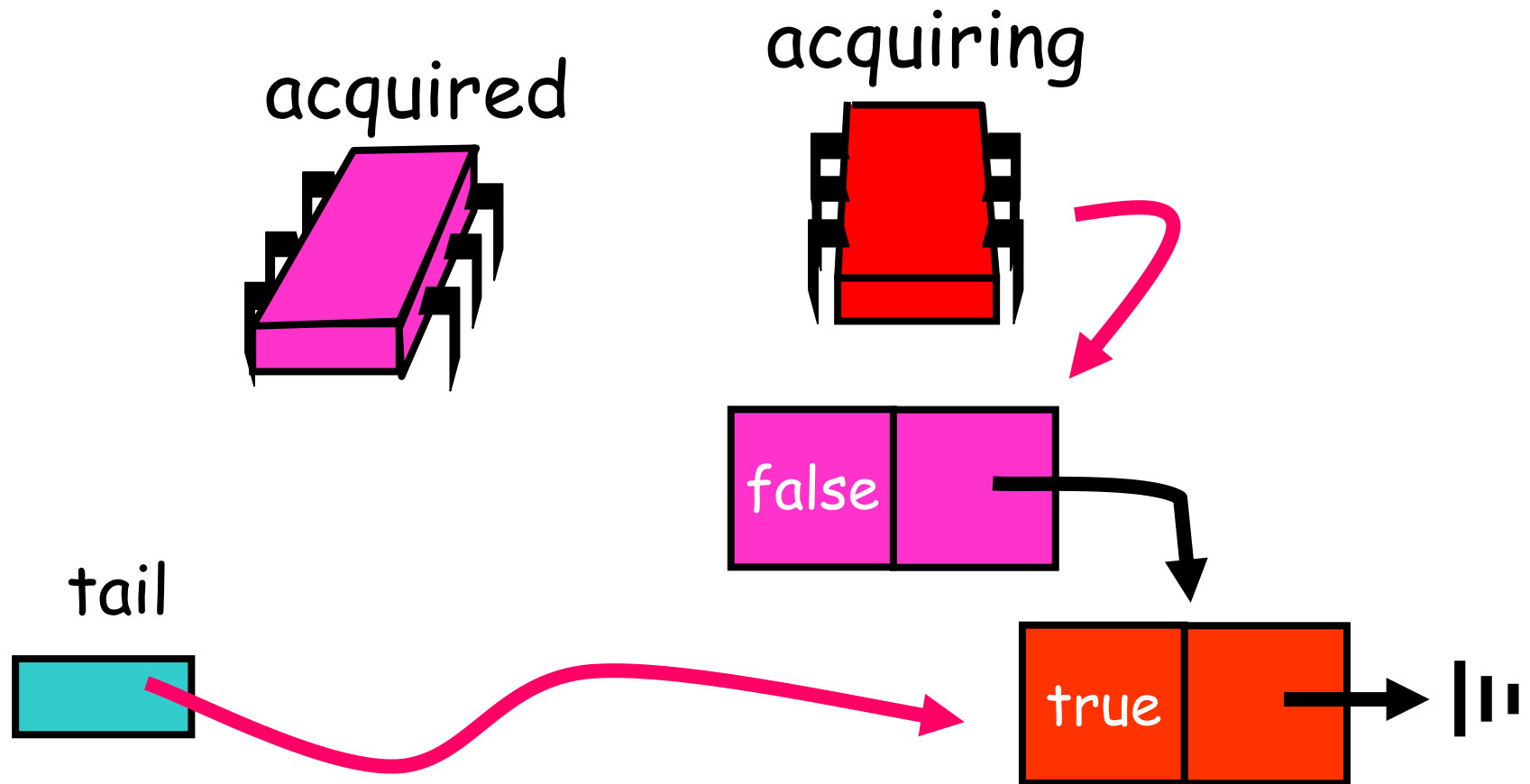
Acquiring



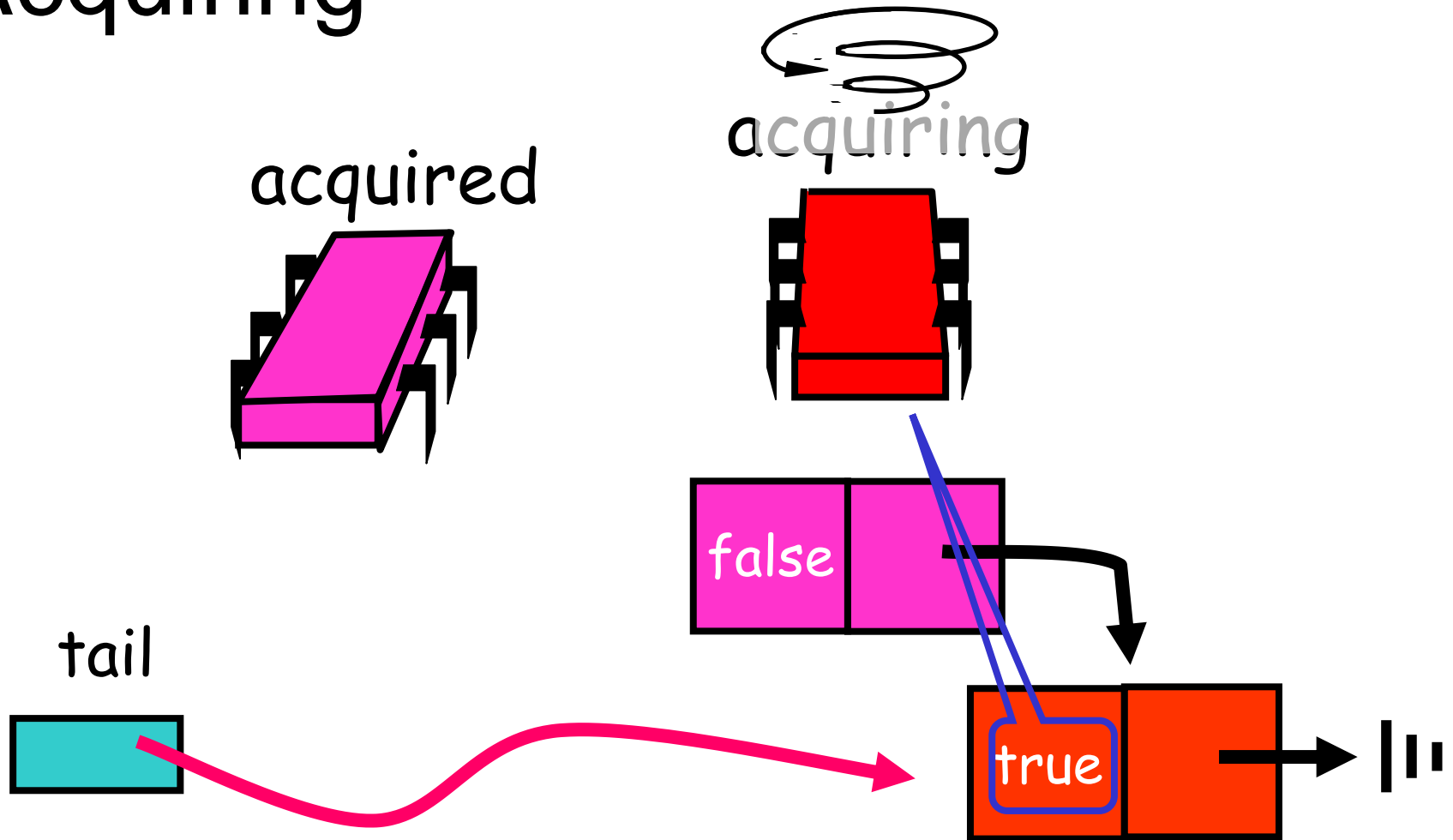
Acquiring



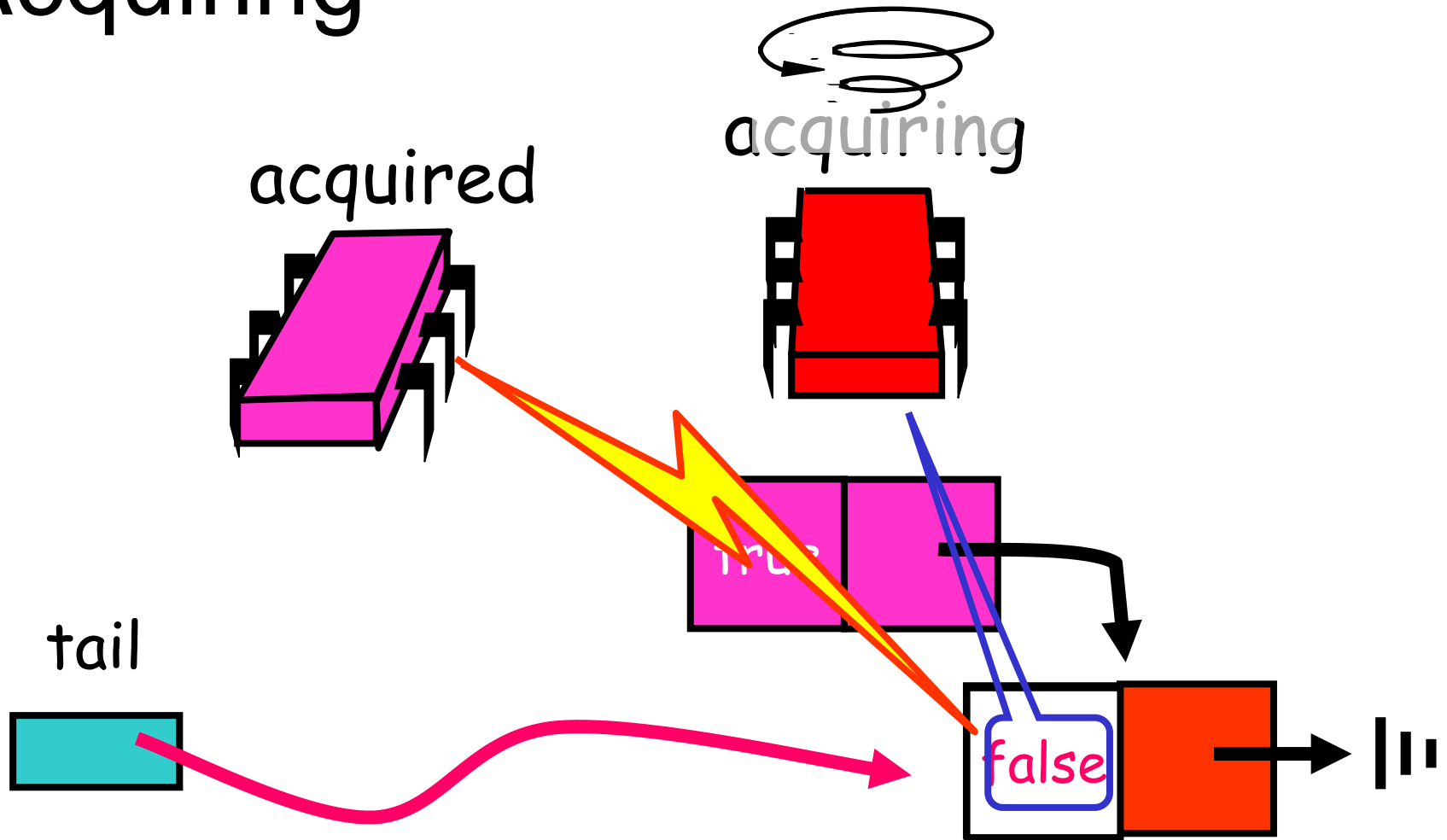
Acquiring



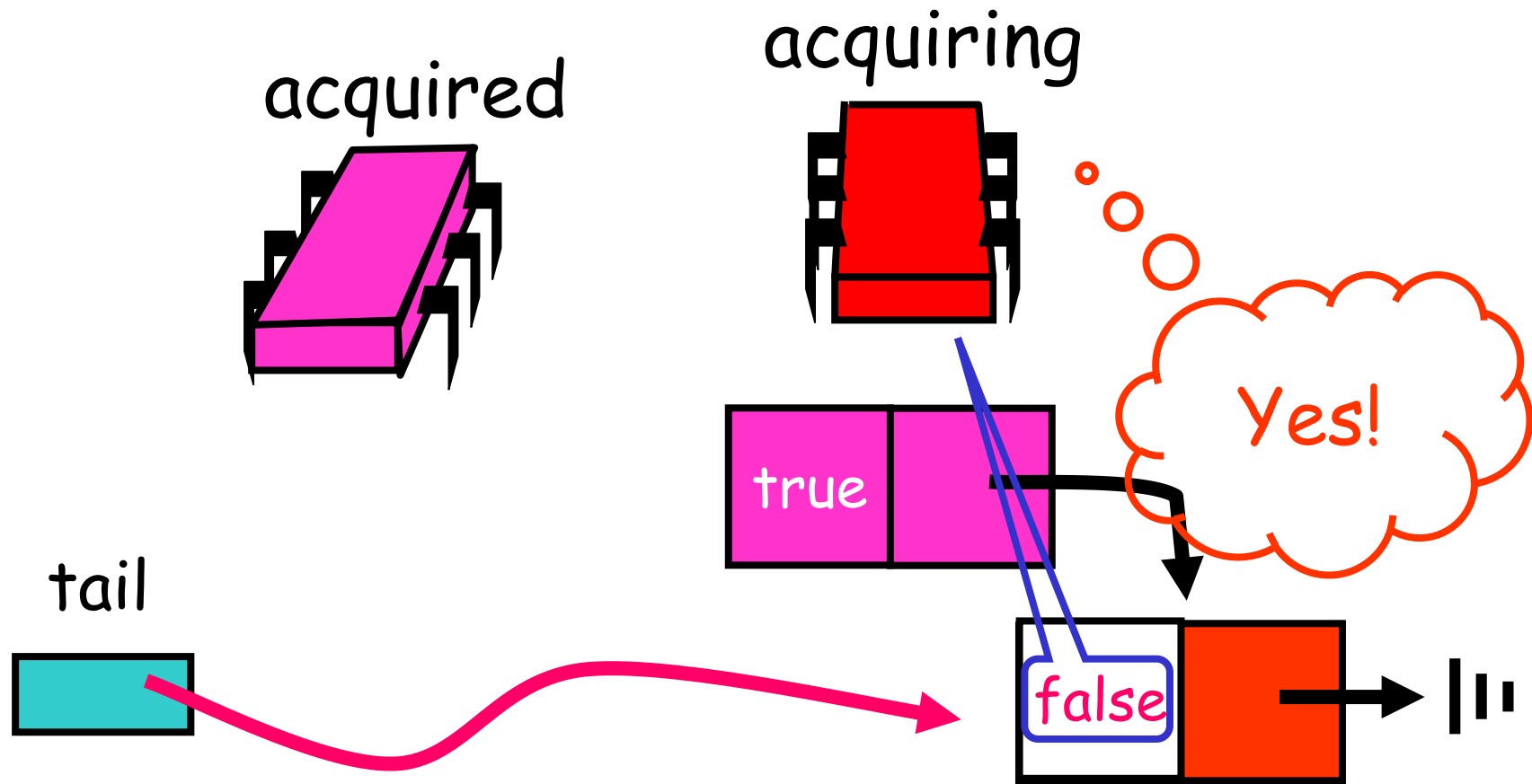
Acquiring



Acquiring



Acquiring





MCS Queue Lock

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

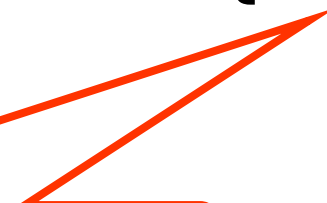
MCS Queue Lock

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


MCS Queue Lock

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

Make a
QNode



MCS Queue Lock

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

add my Node to
the tail of
queue

MCS Queue Lock

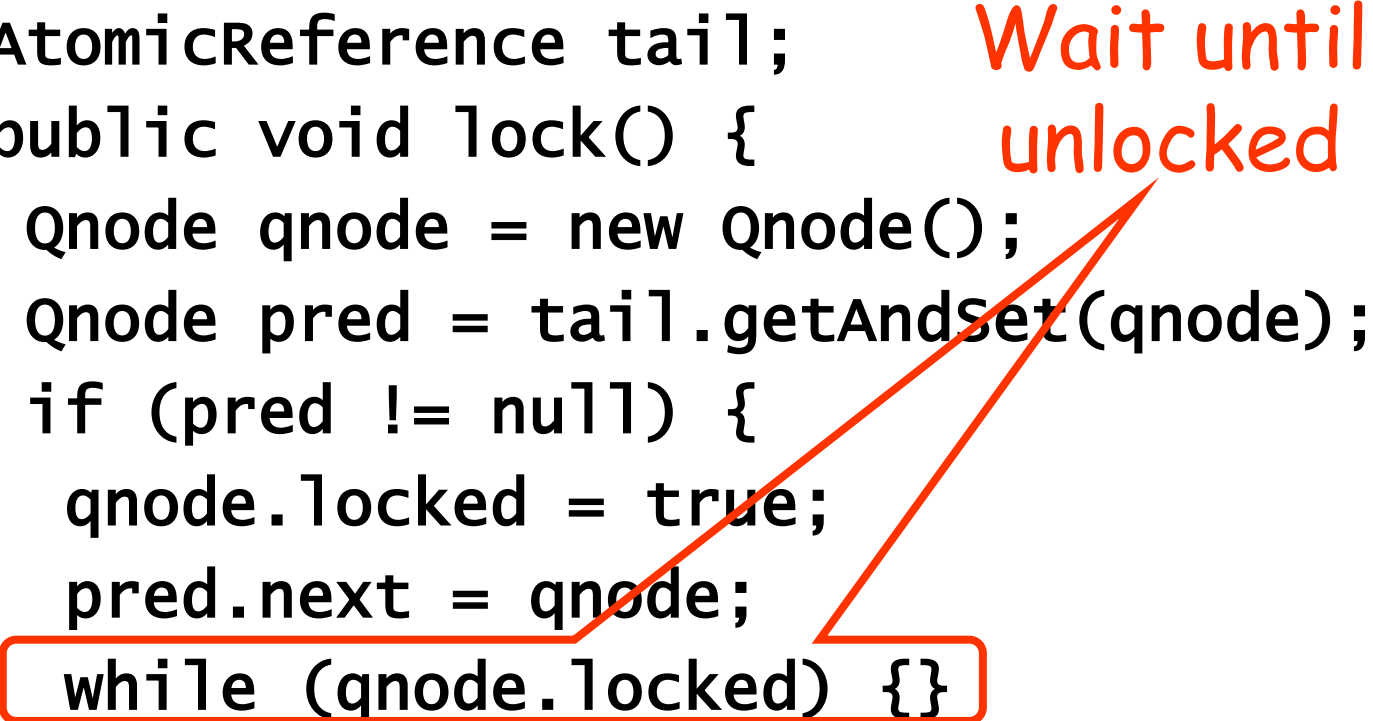
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked)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

Fix if queue
was non-empty

MCS Queue Lock

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

Wait until
unlocked





MCS Queue Unlock

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

MCS Queue Lock

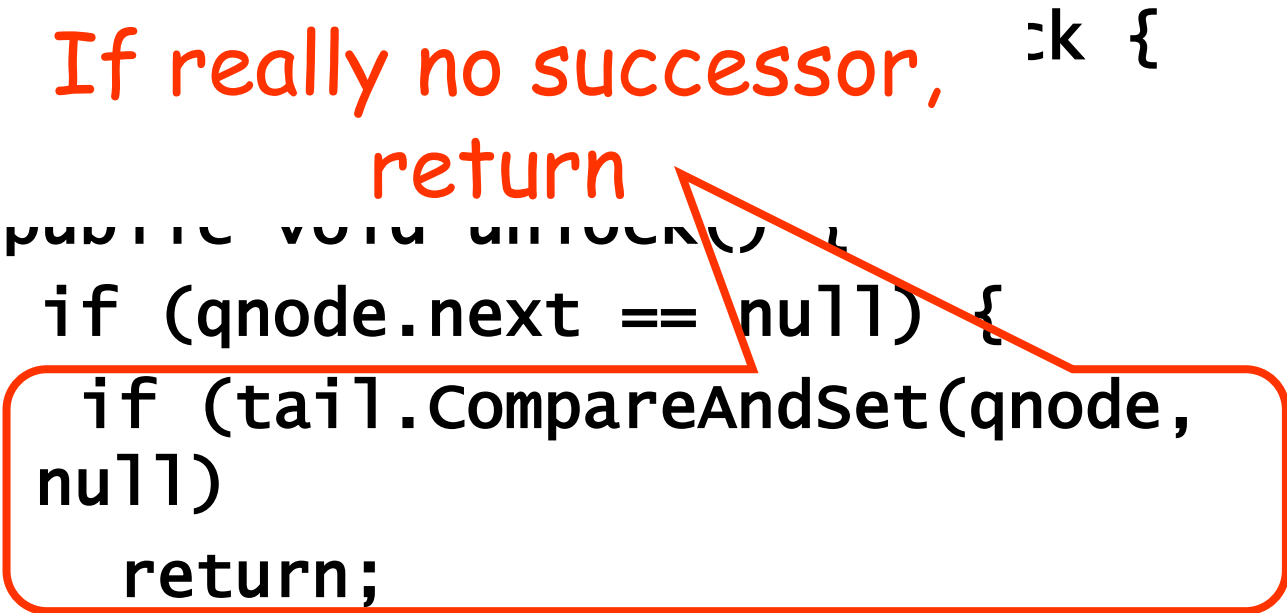
```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```

Missing
successor?

MCS Queue Lock

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

*If really no successor,
return*



```

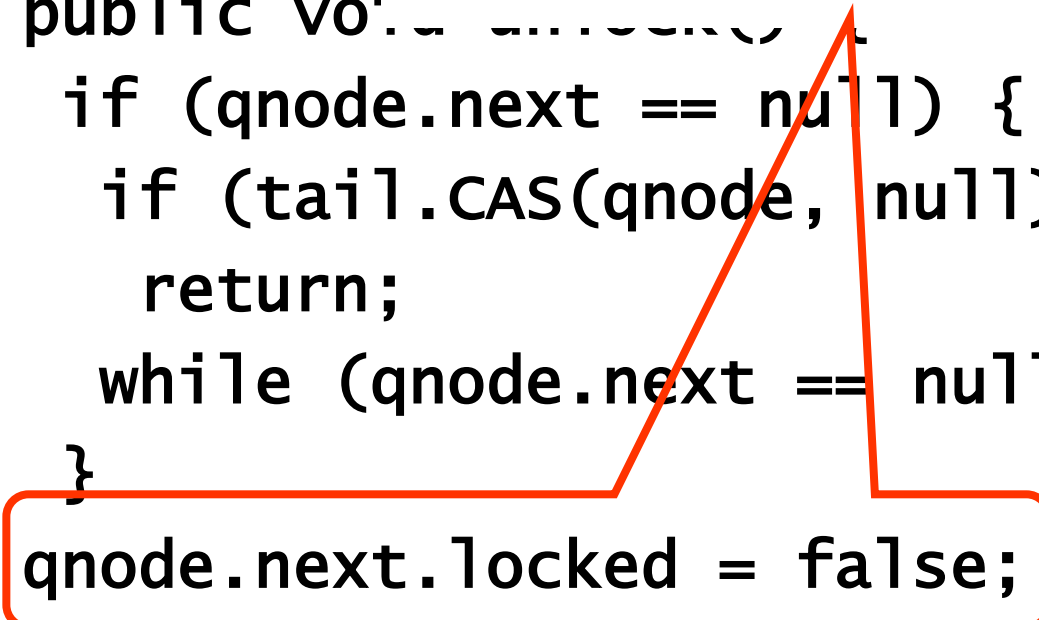
    public void unlock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null)
                return;
            while (qnode.next == null) {}
        }
        qnode.next.locked = false;
    }
}

```


MCS Queue Lock

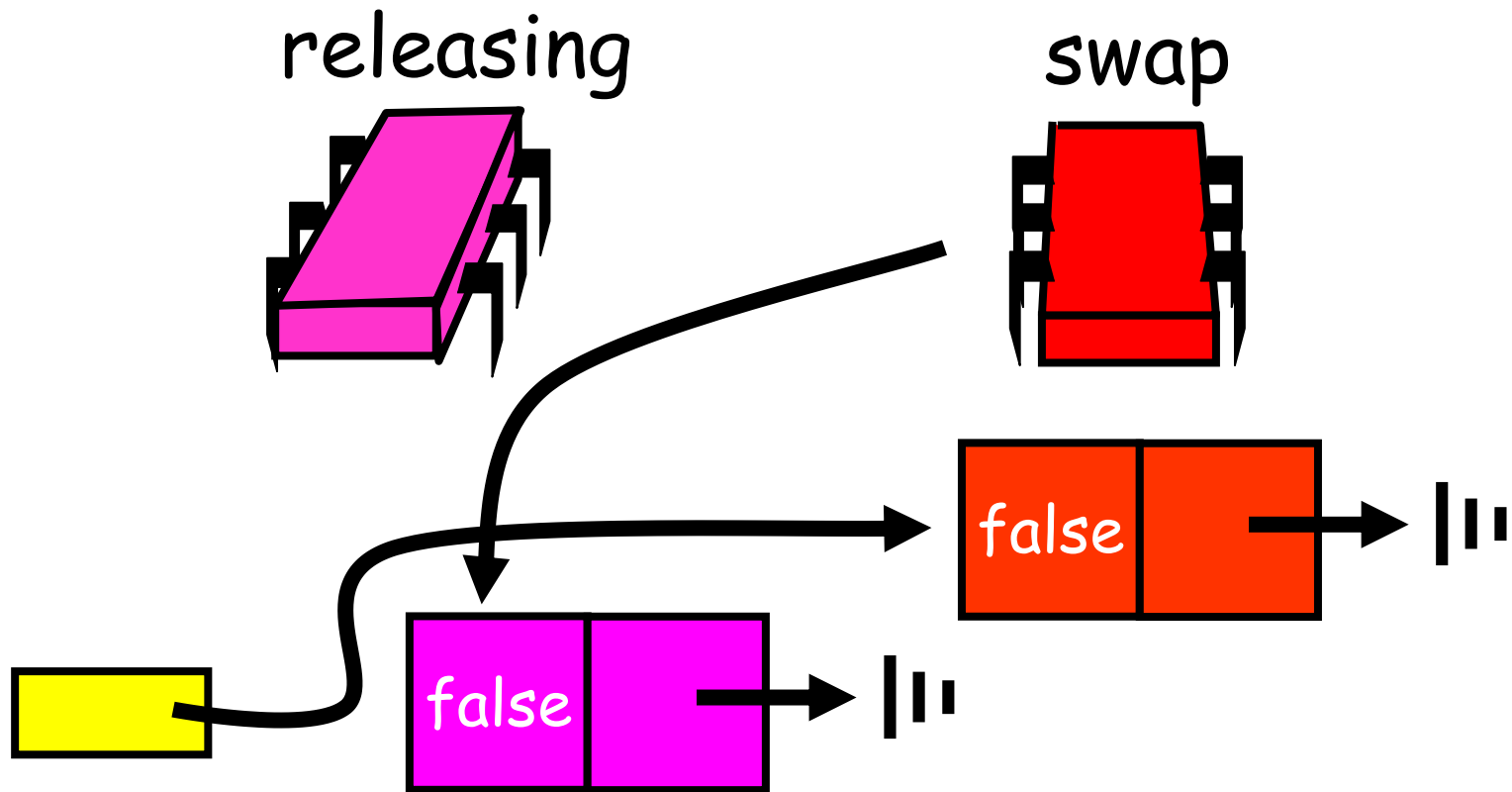
```
class MCSLock implements Lock {
    AtomicRef<QNode> tail;
    public void lock() {
        if (qnode.next == null) {
            if (tail.CAS(qnode, null))
                return;
            while (qnode.next == null) {}
        }
        qnode.next.locked = false;
    }
}
```

Pass lock to successor

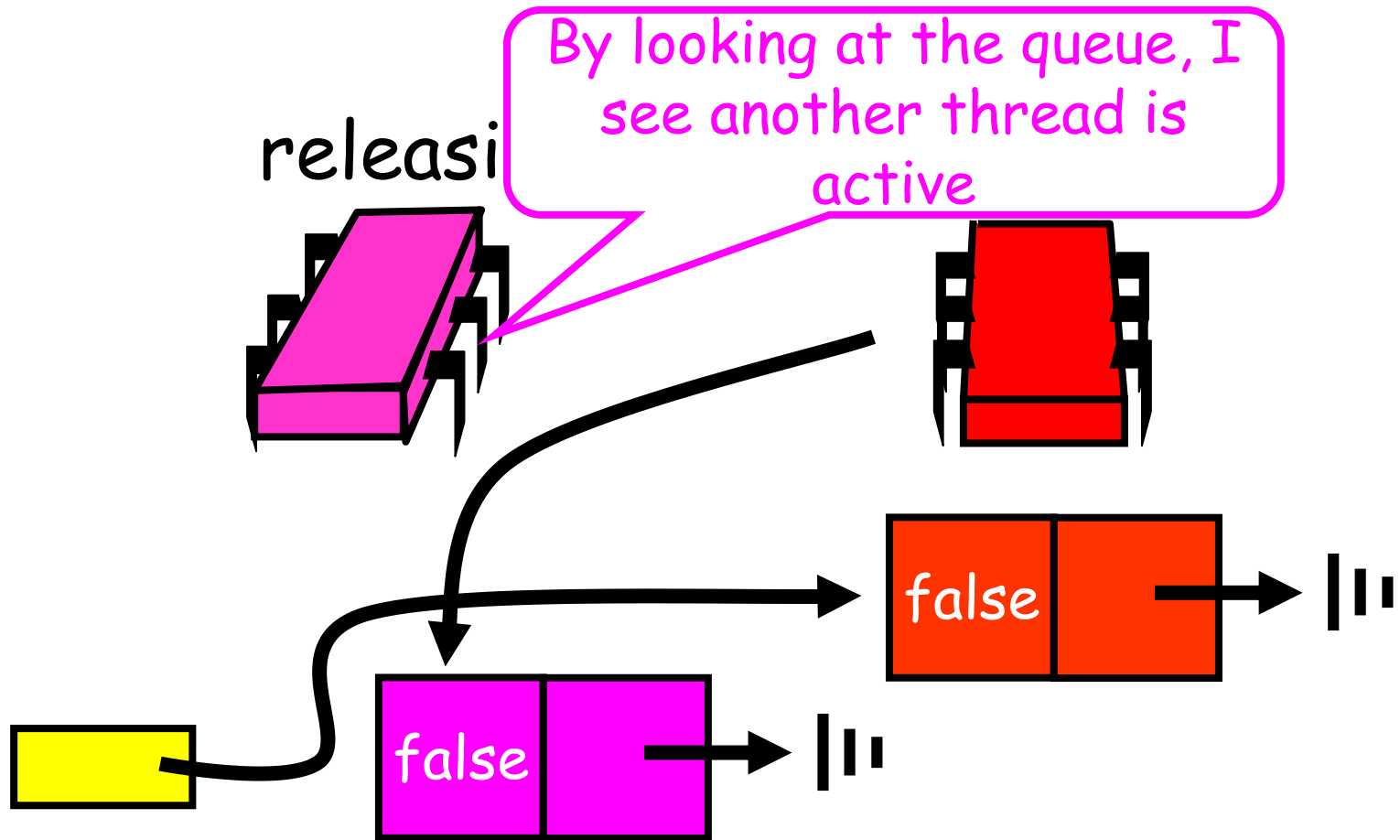


The diagram consists of a red arrow pointing from the `qnode` parameter in the `lock()` method to the `qnode` parameter in the `locked` field access. A red box highlights the `qnode.next.locked = false;` line, and another red box highlights the `qnode` parameter in the `locked` field access. The text "Pass lock to successor" is written in red above the arrow.

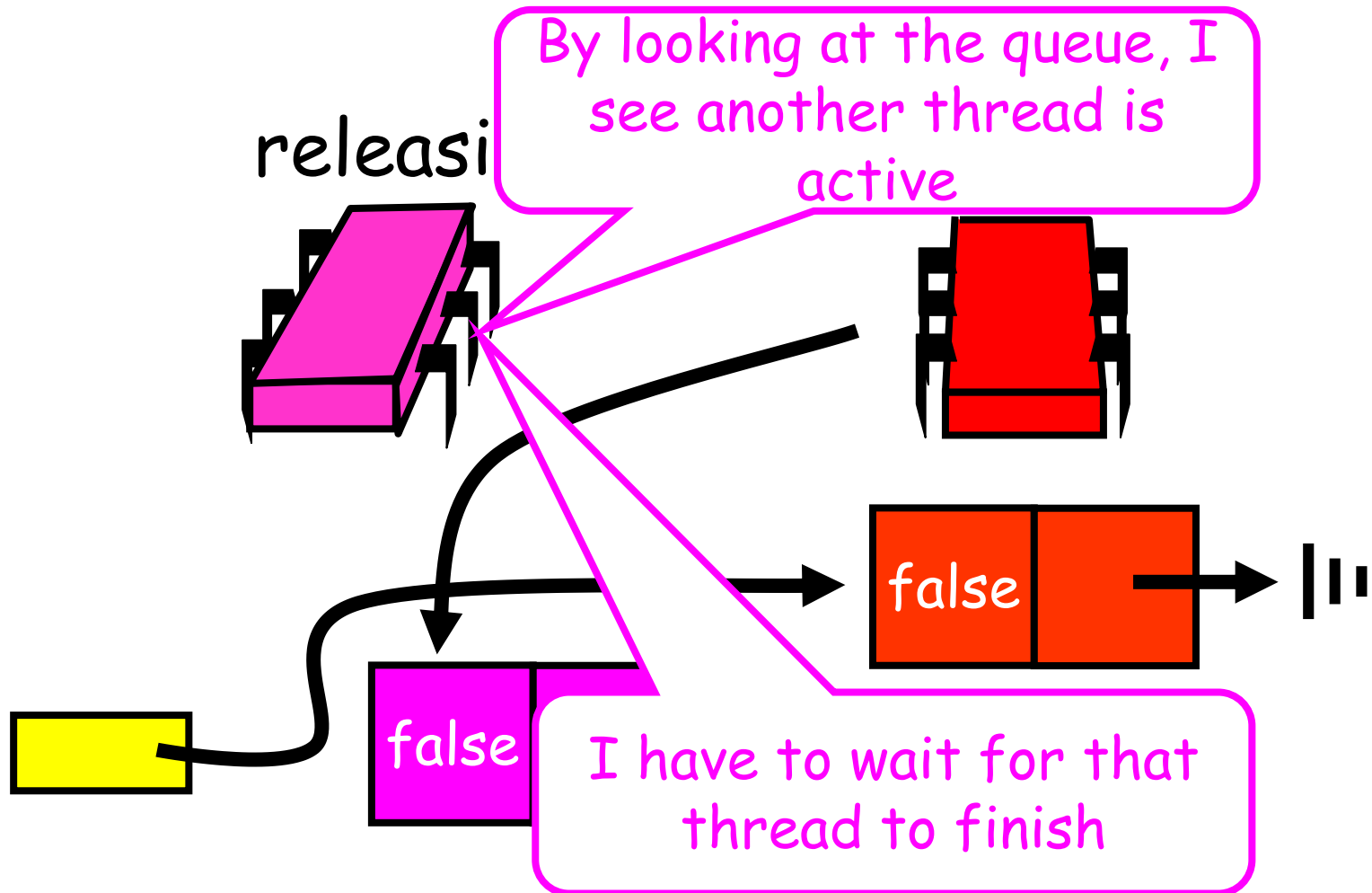
Purple Release



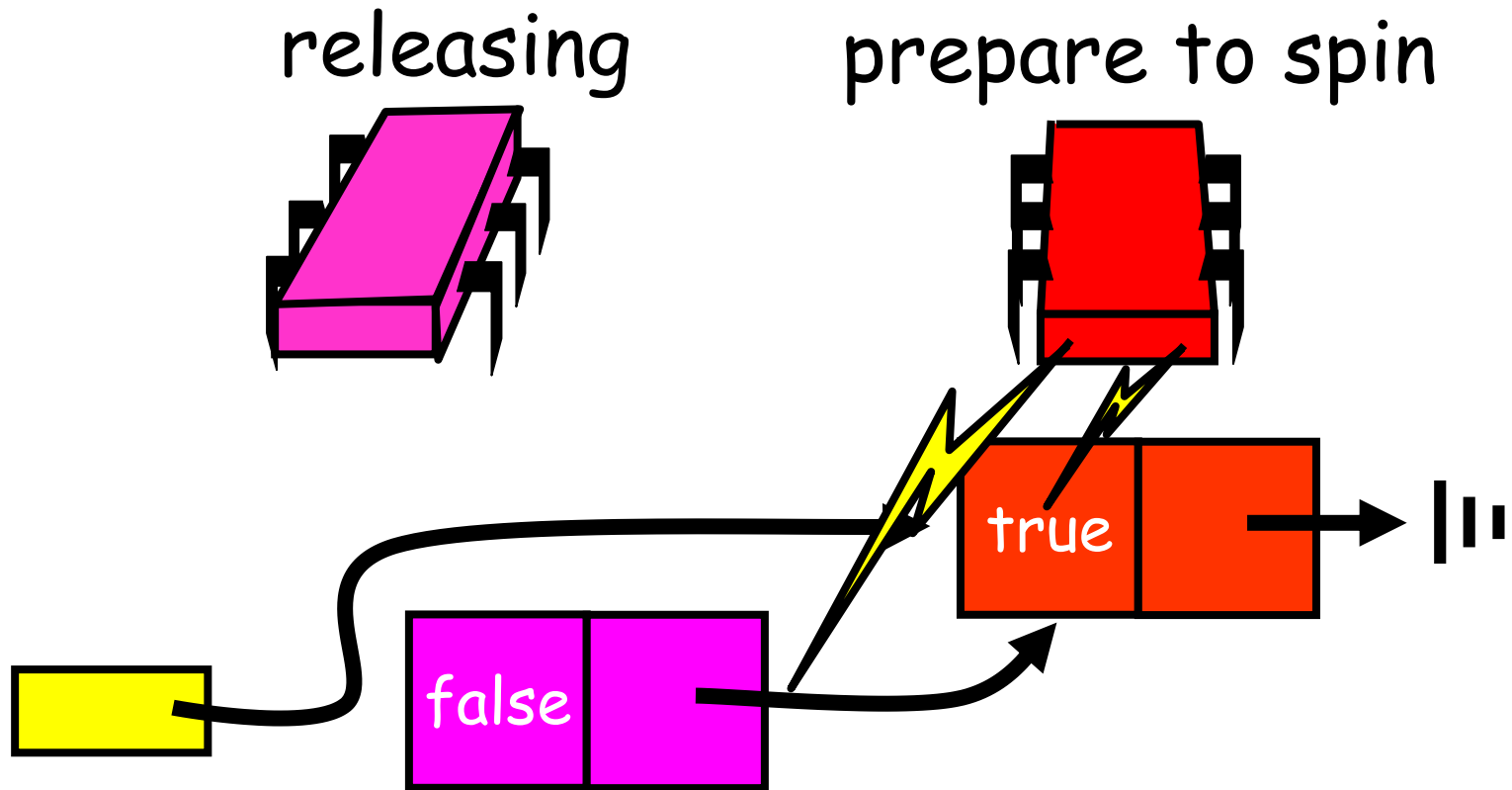
Purple Release



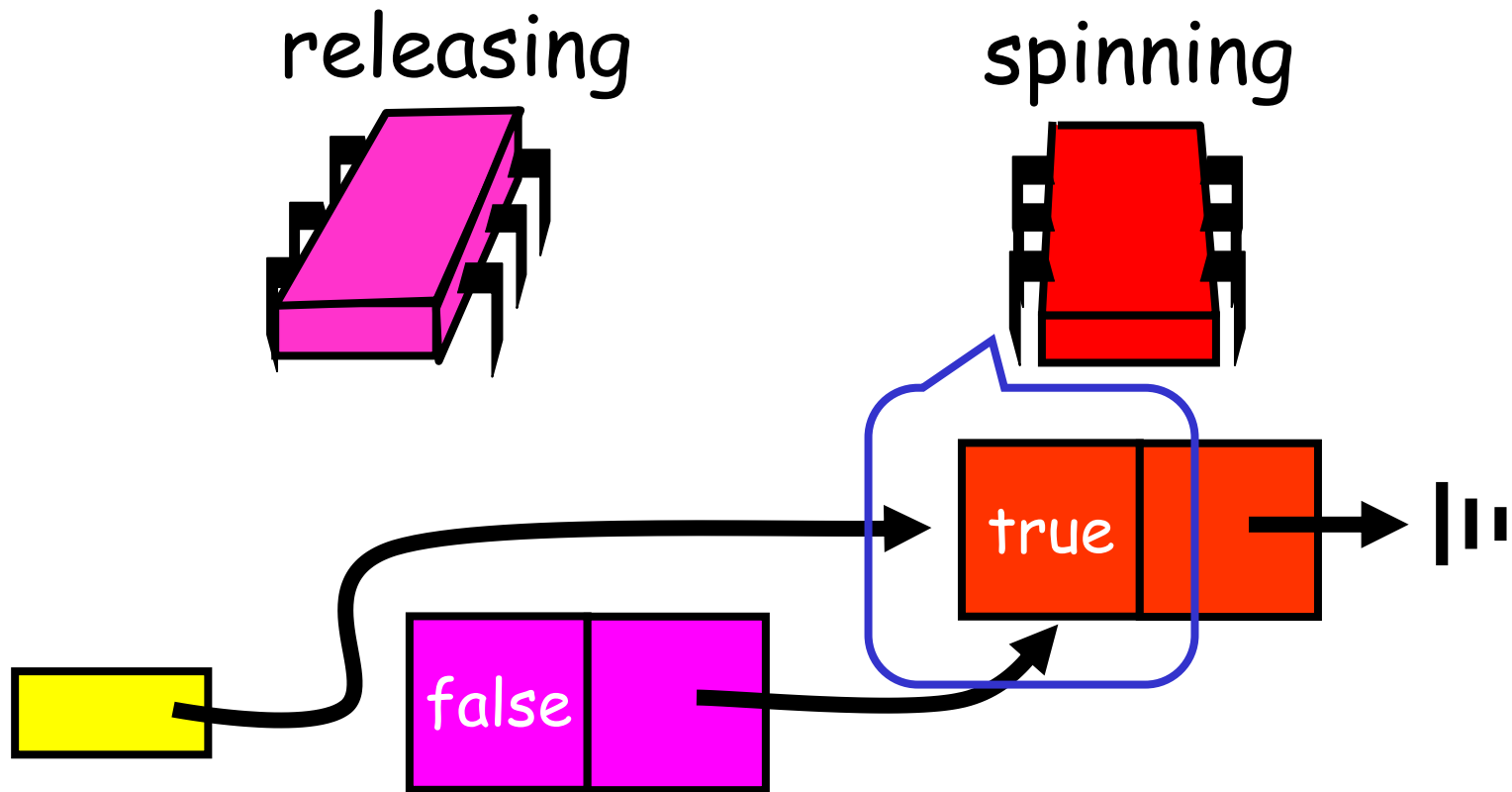
Purple Release



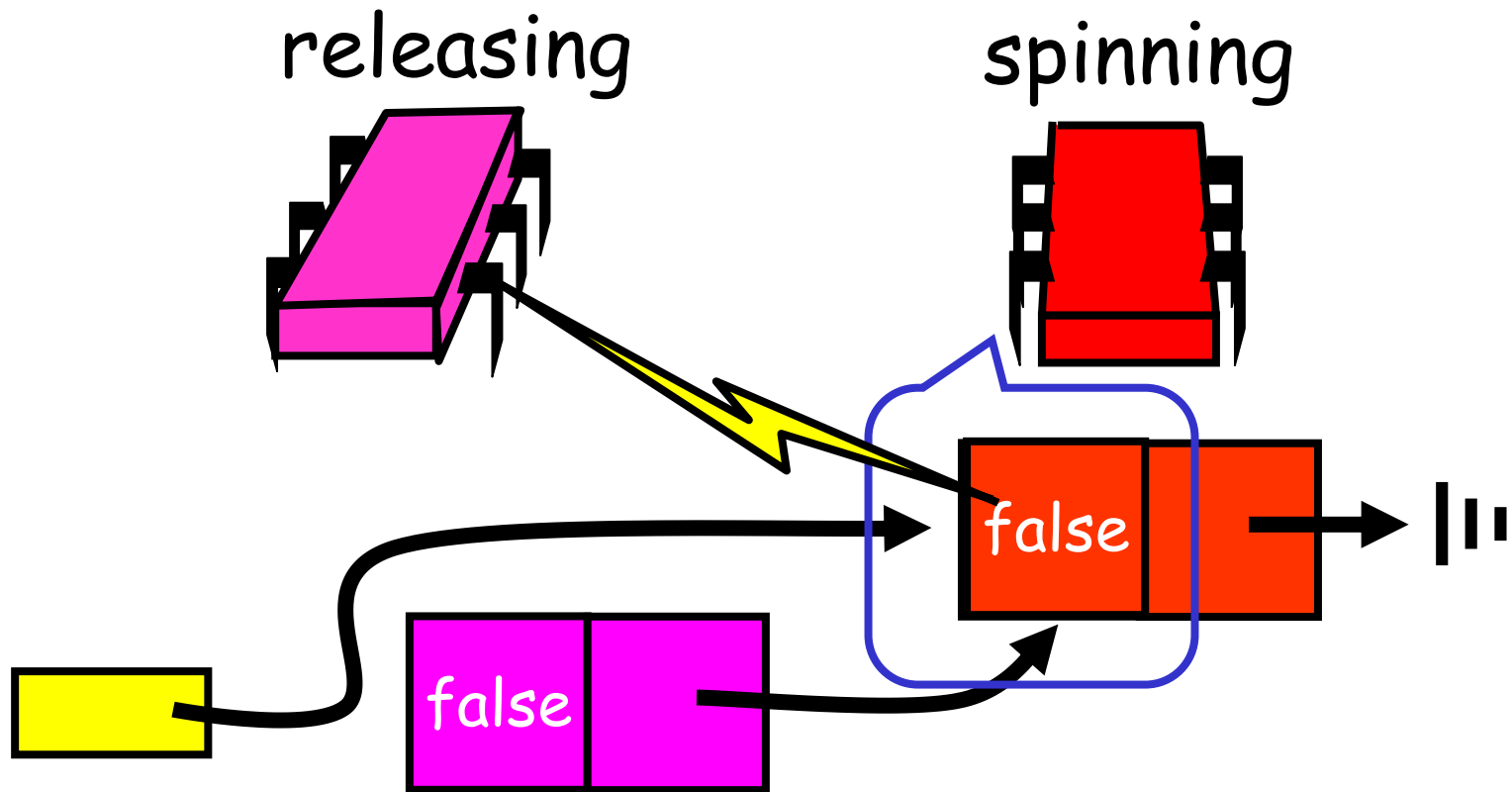
Purple Release



Purple Release

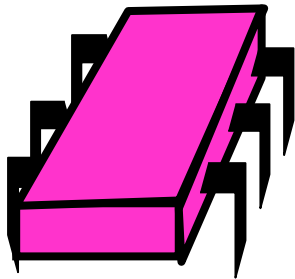


Purple Release

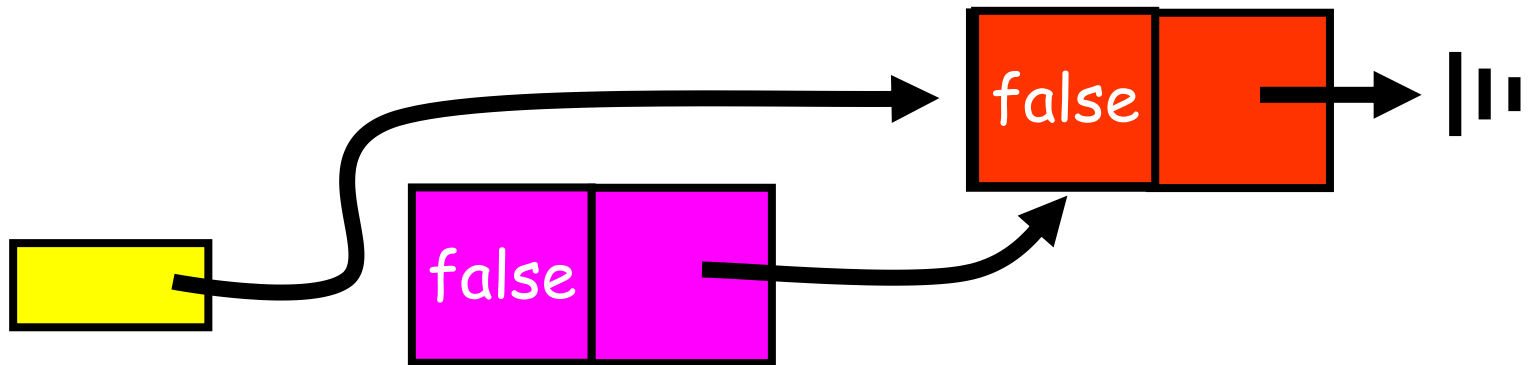
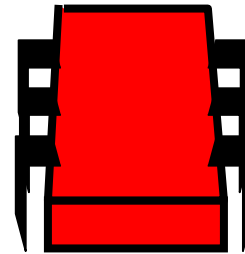


Purple Release

releasing



Acquired lock





Lock with timeout

- Java interface includes a `tryLock()` method to specify a maximum duration the thread is willing to wait to acquire the lock
- Should the thread not acquire the lock in the designated time, the thread will timeout



Abortable Locks

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



Lock with timeout

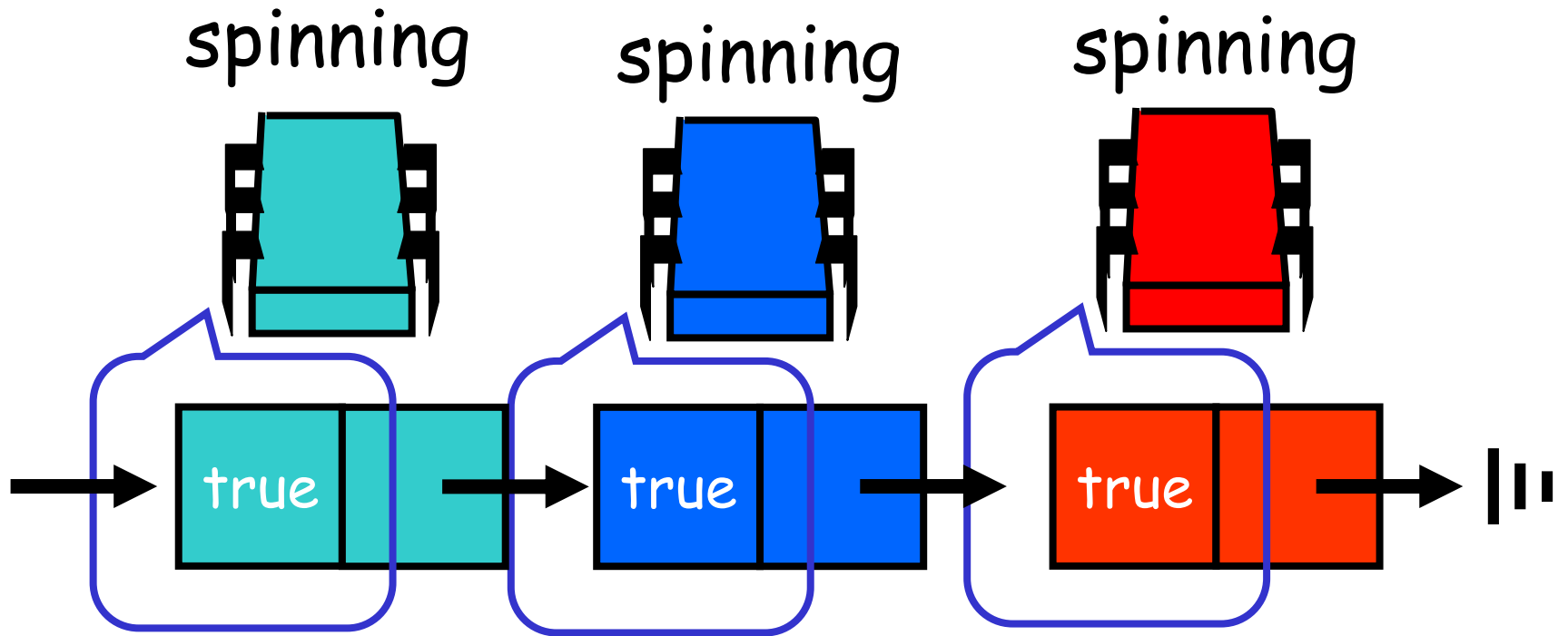
- What happens to other threads when you timeout?



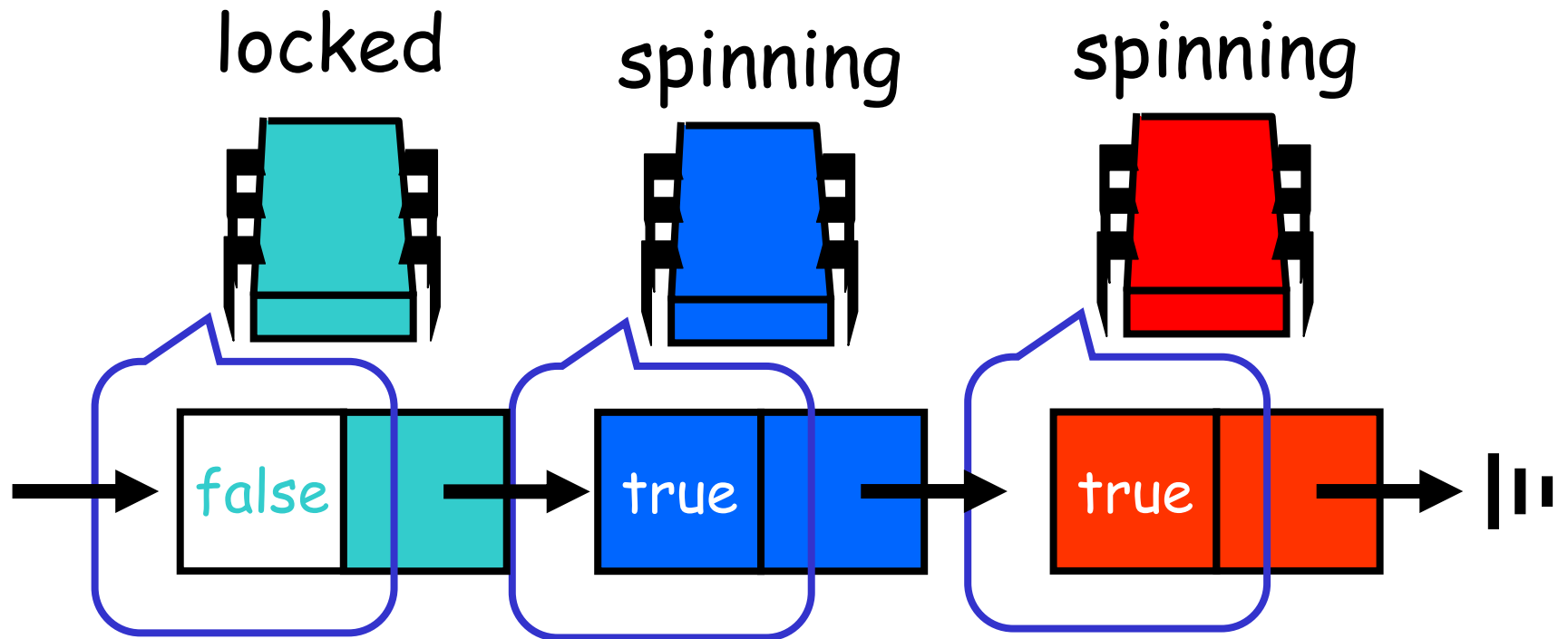
Back-off Lock

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

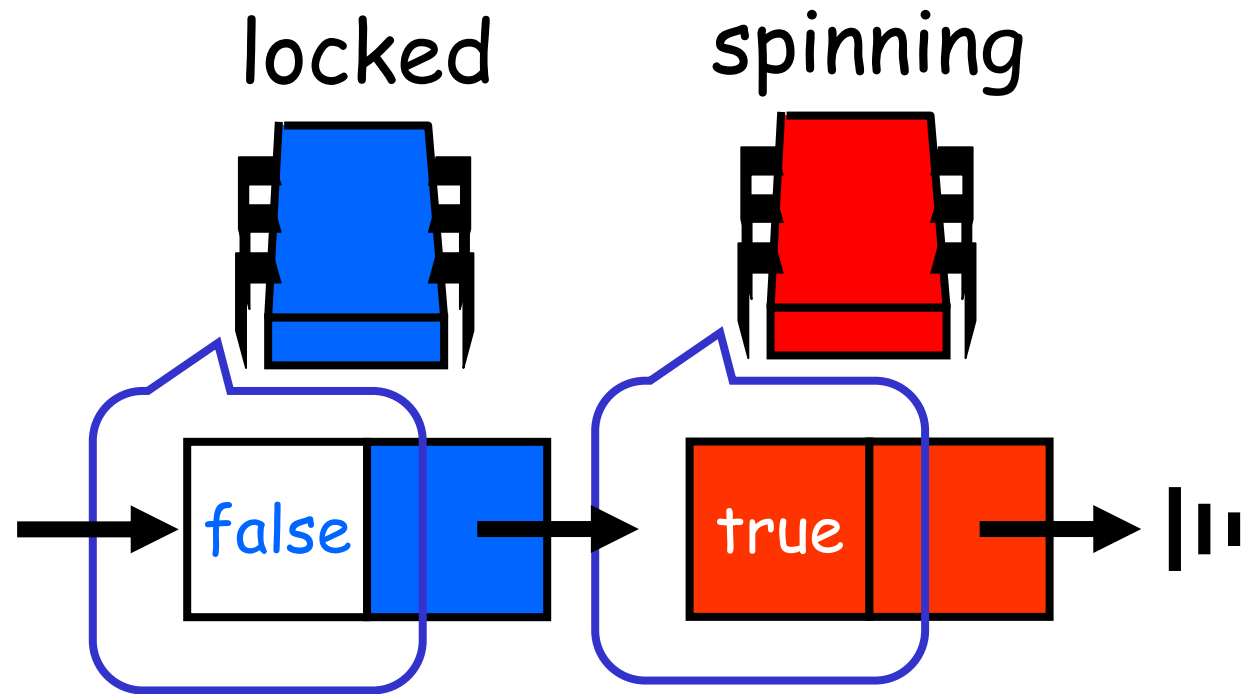
MCS Queue Locks



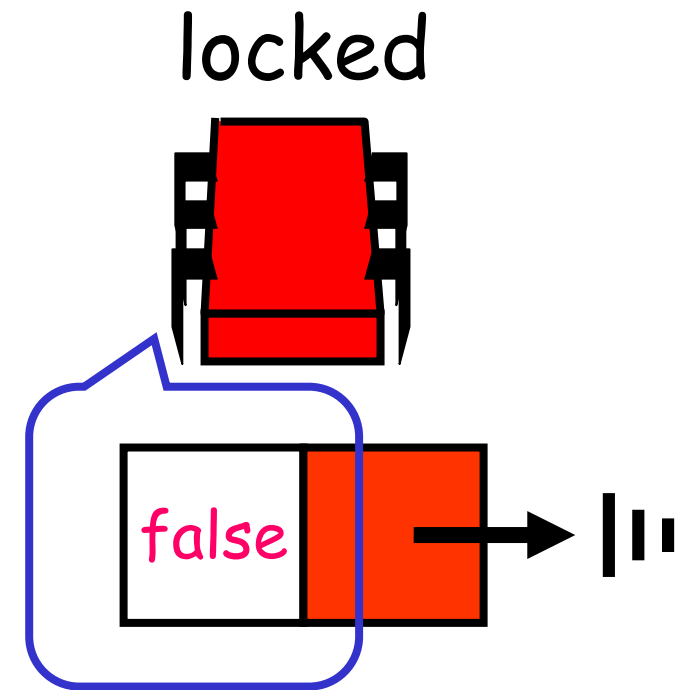
MCS Queue Locks



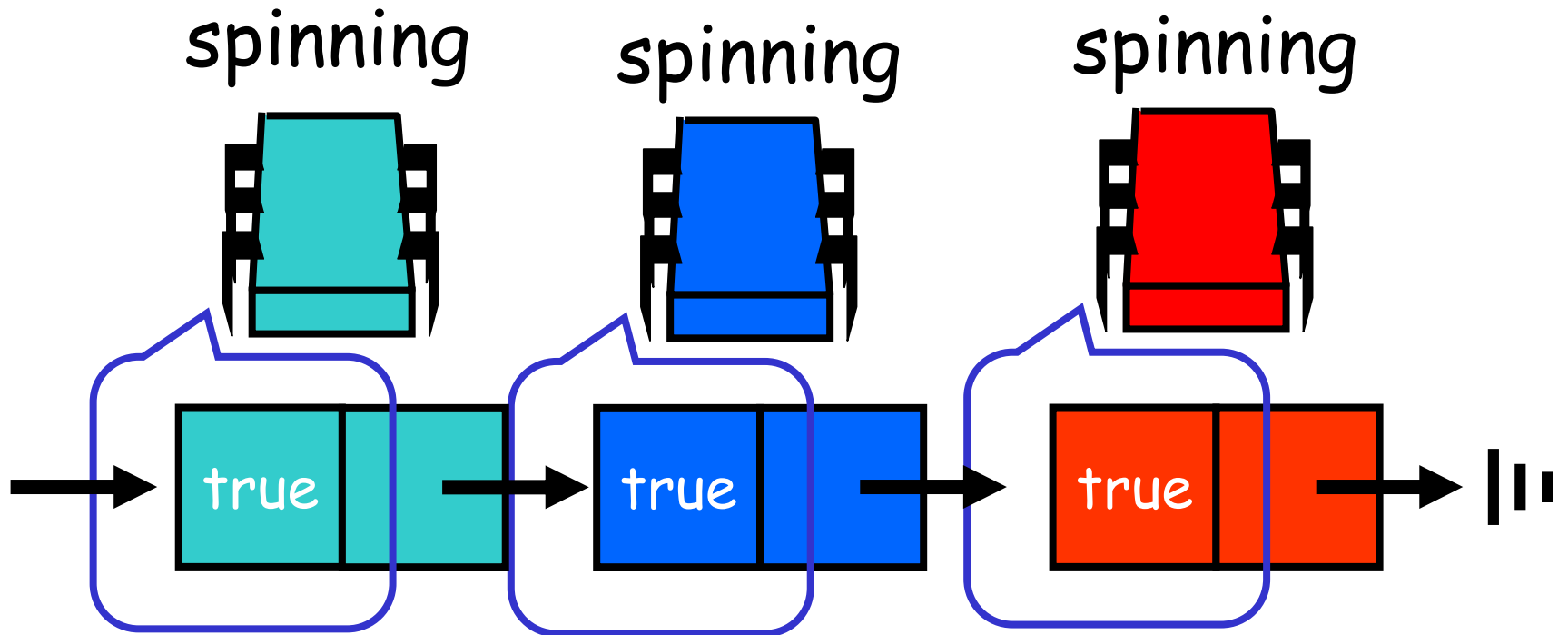
MCS Queue Locks



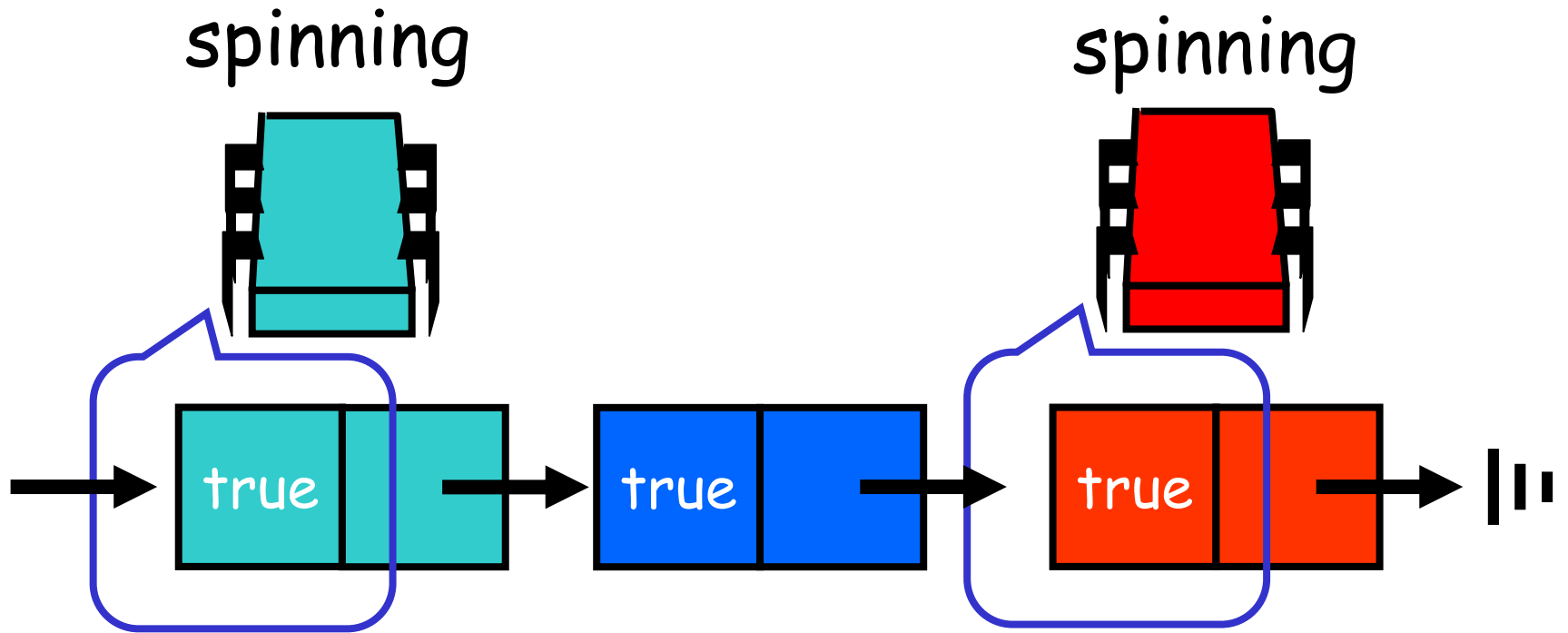
MCS Queue Locks



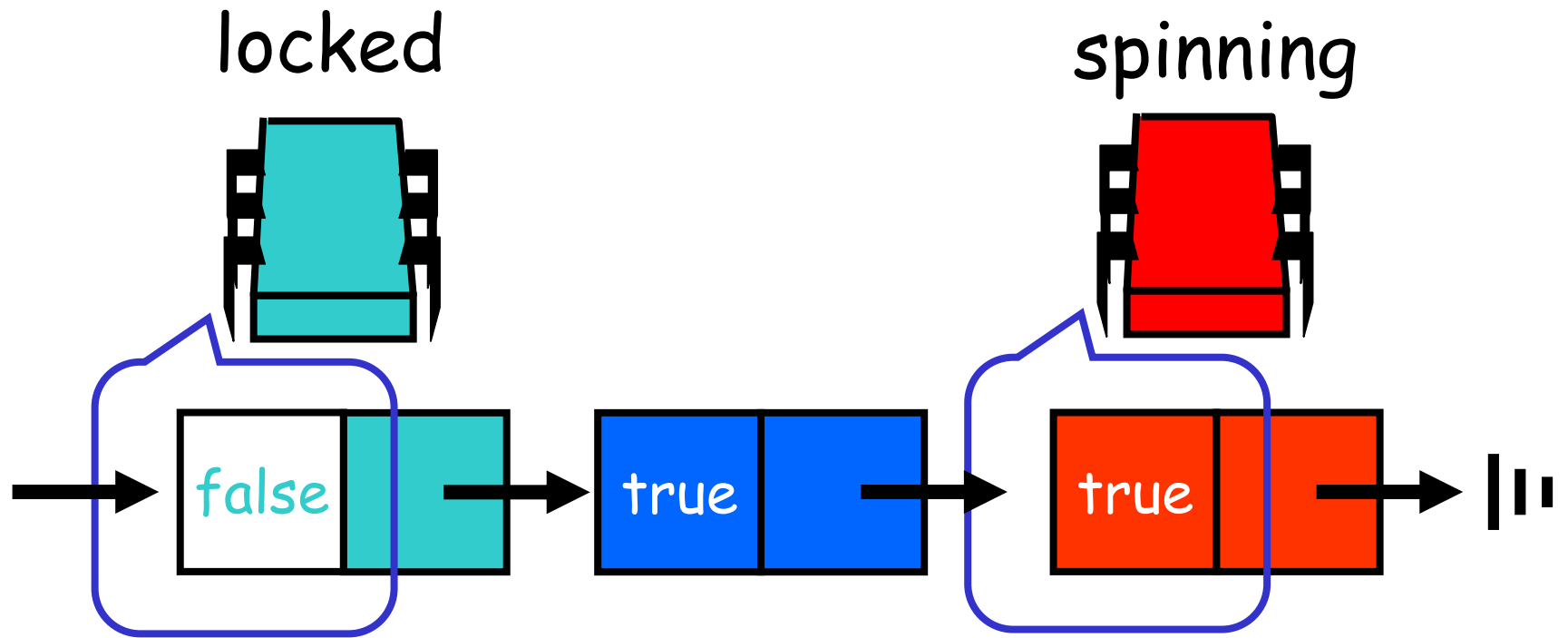
MCS Queue Locks



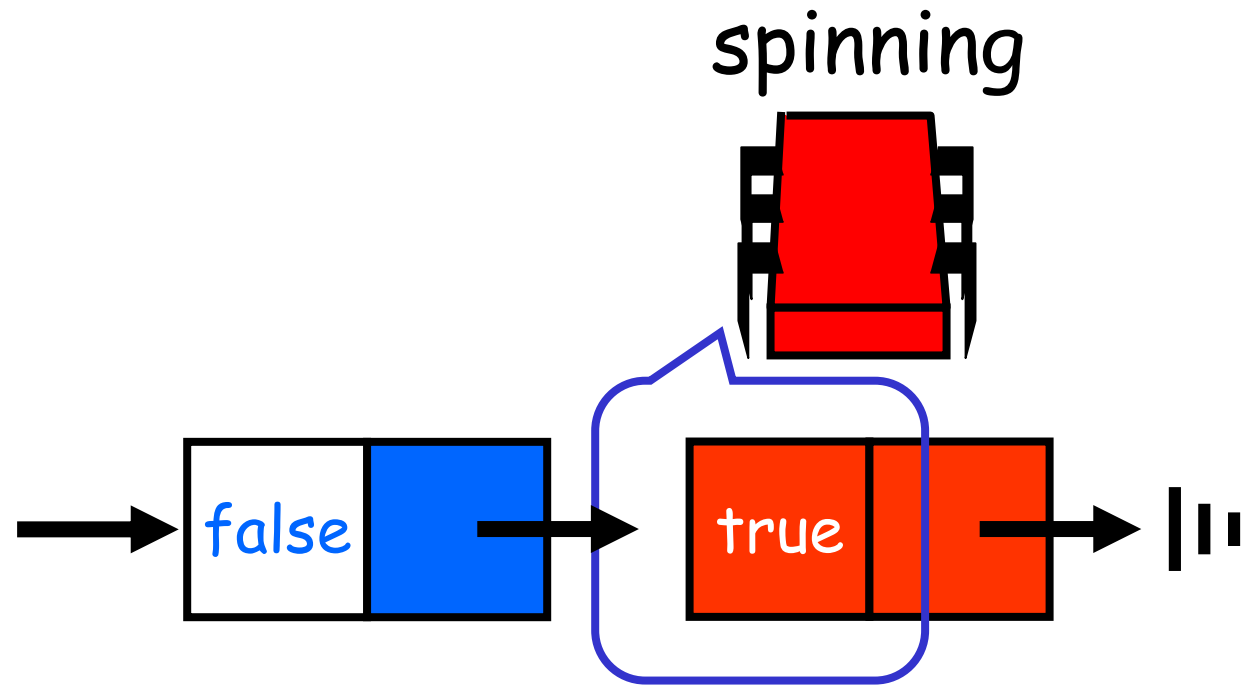
MCS Queue Locks



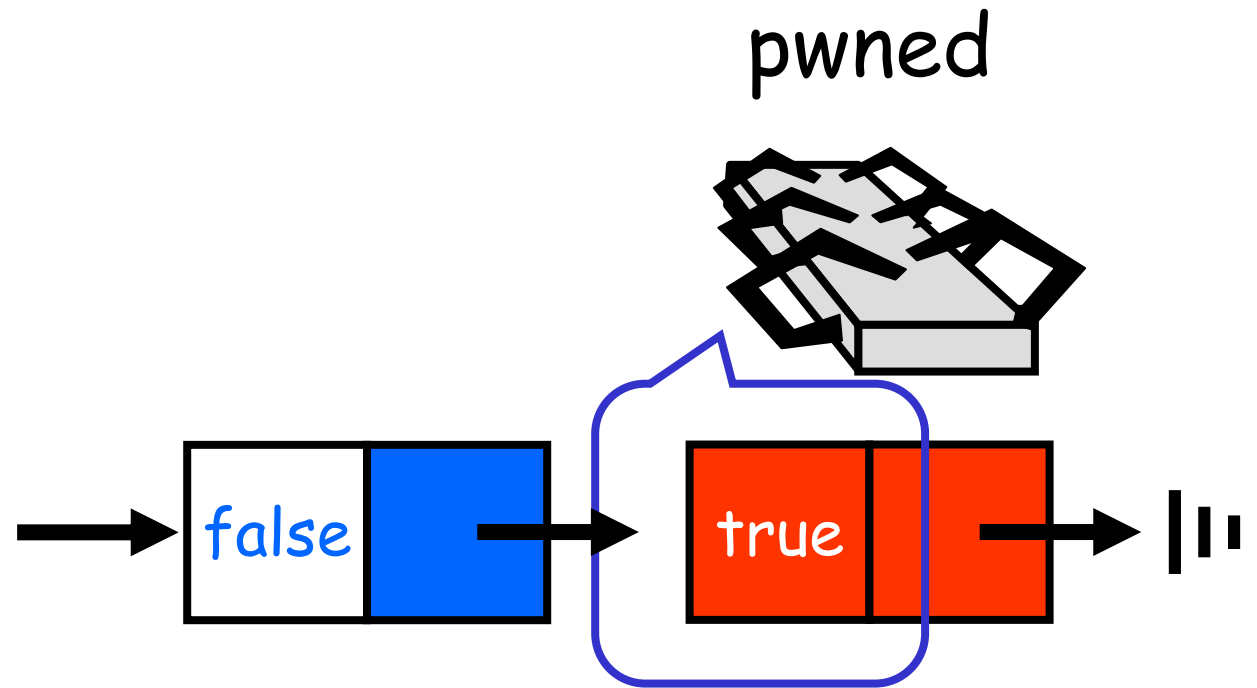
MCS Queue Locks



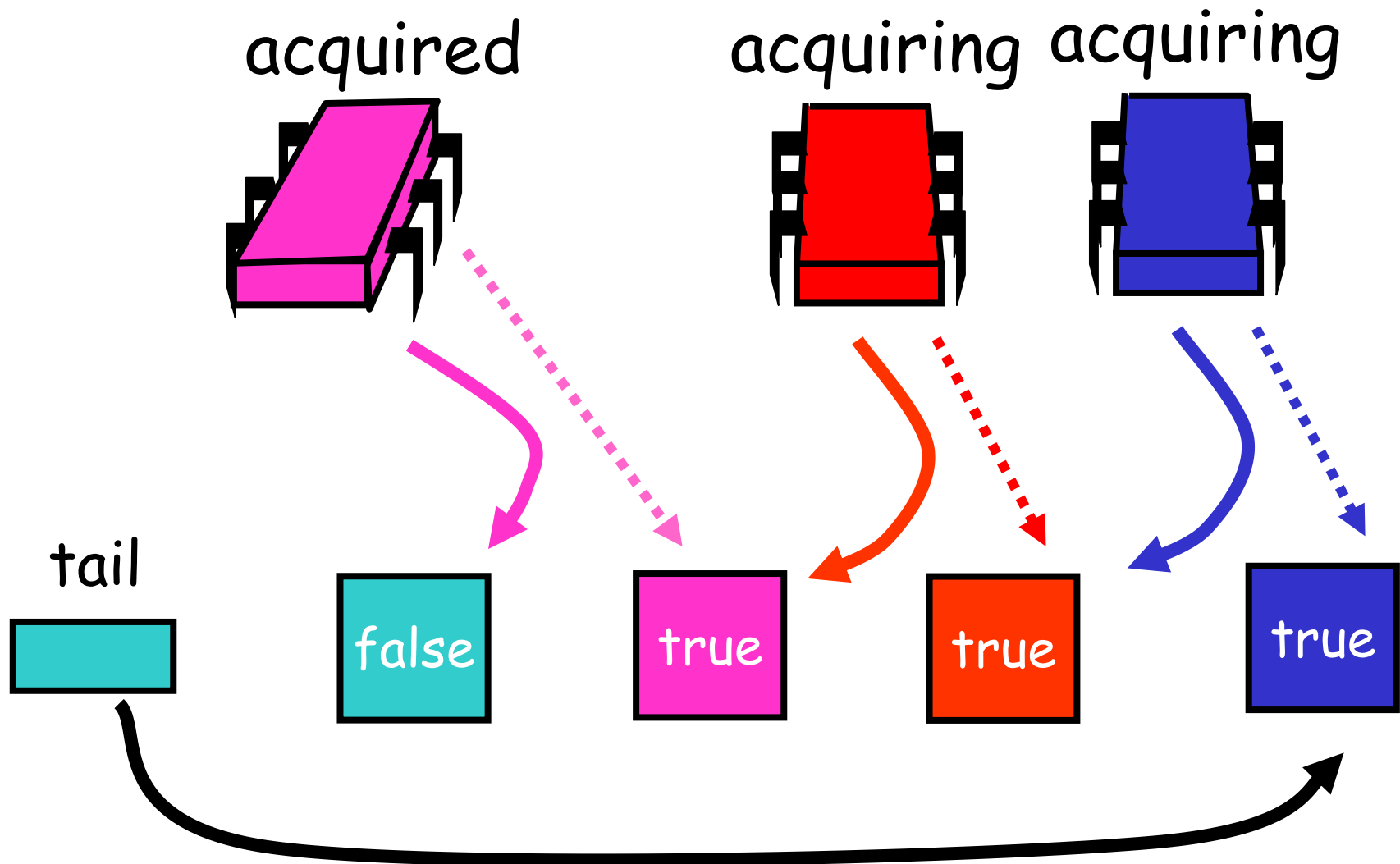
MCS Queue Locks

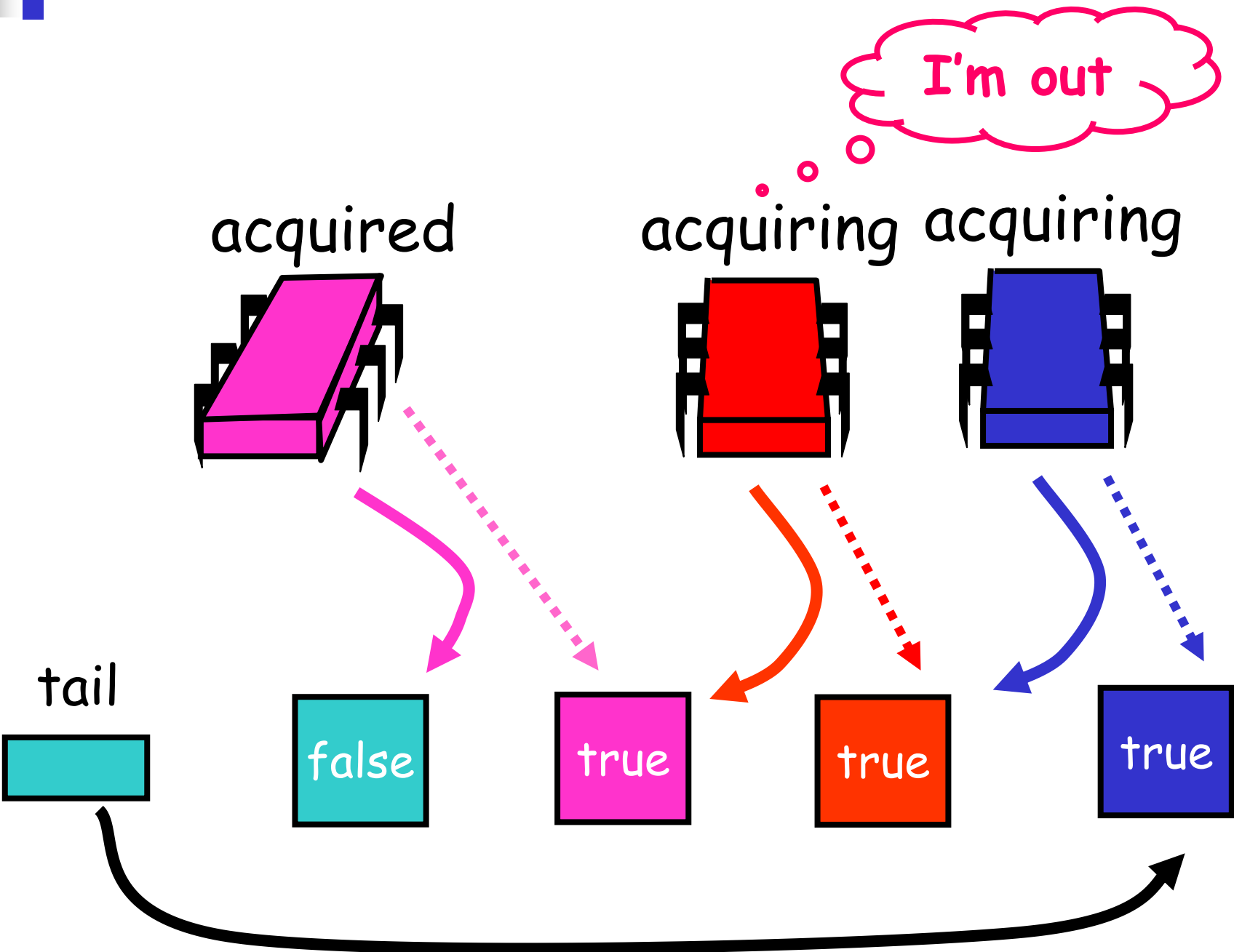


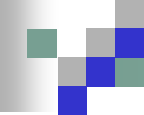
MCS Queue Locks



CLH Queue Lock

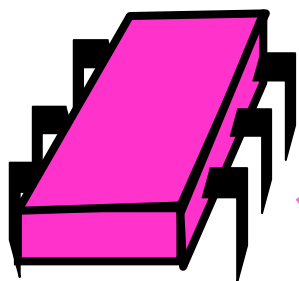




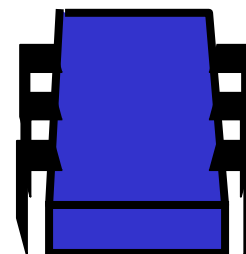


Finished

release



acquiring



tail

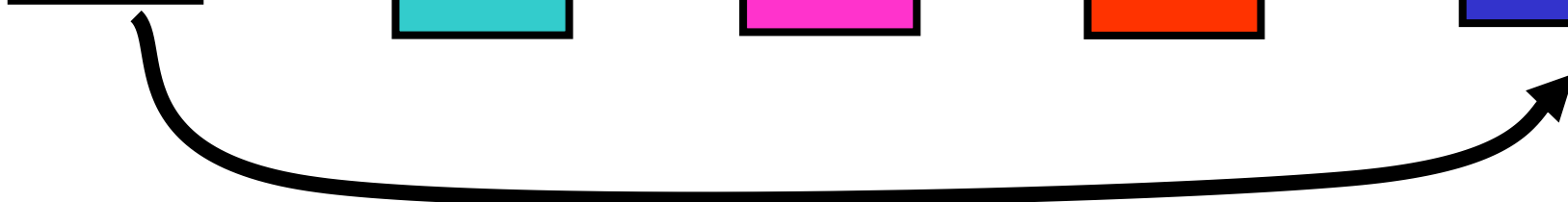


false

false

true

true





Queue Locks

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



Abortable CLH Lock

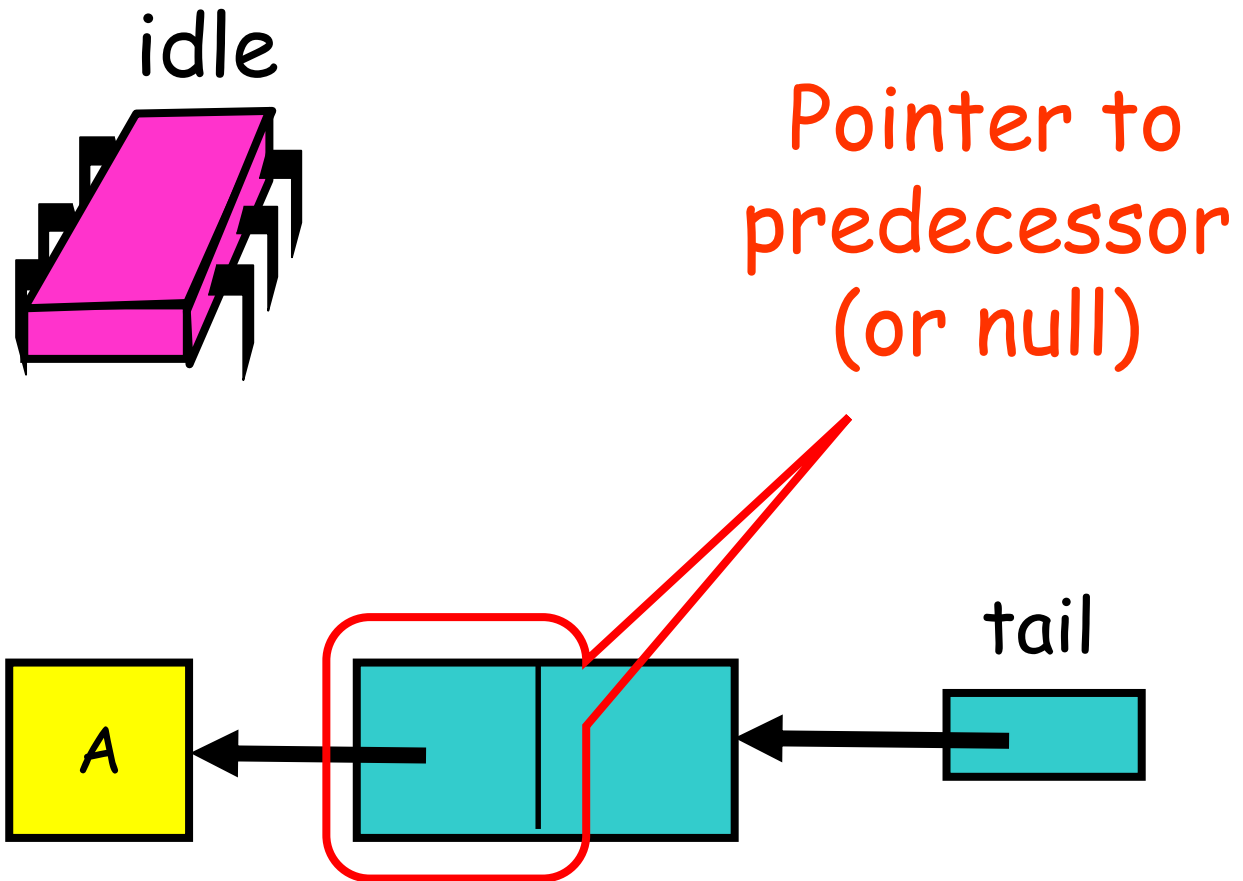
- When a thread gives up
 - Removing node from queue in a wait-free way is hard
- Idea for lazy approach:
 - let successor deal with it.



A queue lock with timeouts

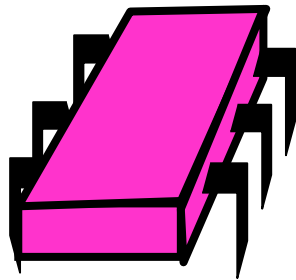
- When a thread times out, it marks its node as abandoned
- The successor in the queue notices that the node has been abandoned
- Successor starts spinning on the abandoned node's predecessors

Initially

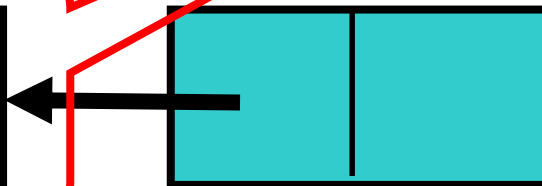
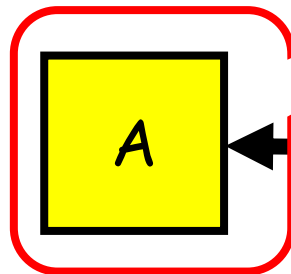


Acquiring

idle



Distinguished
available node
means lock is
free

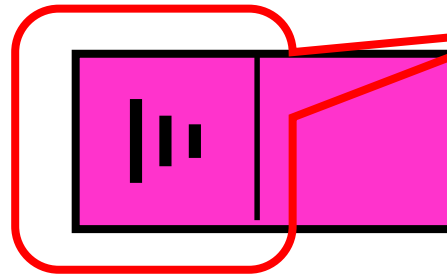
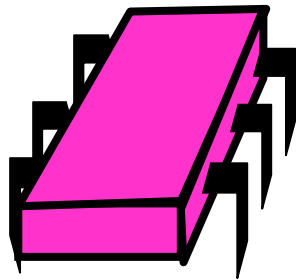


tail

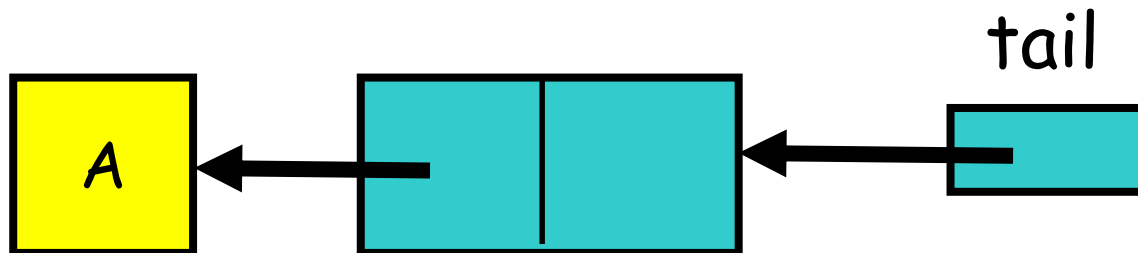


Acquiring

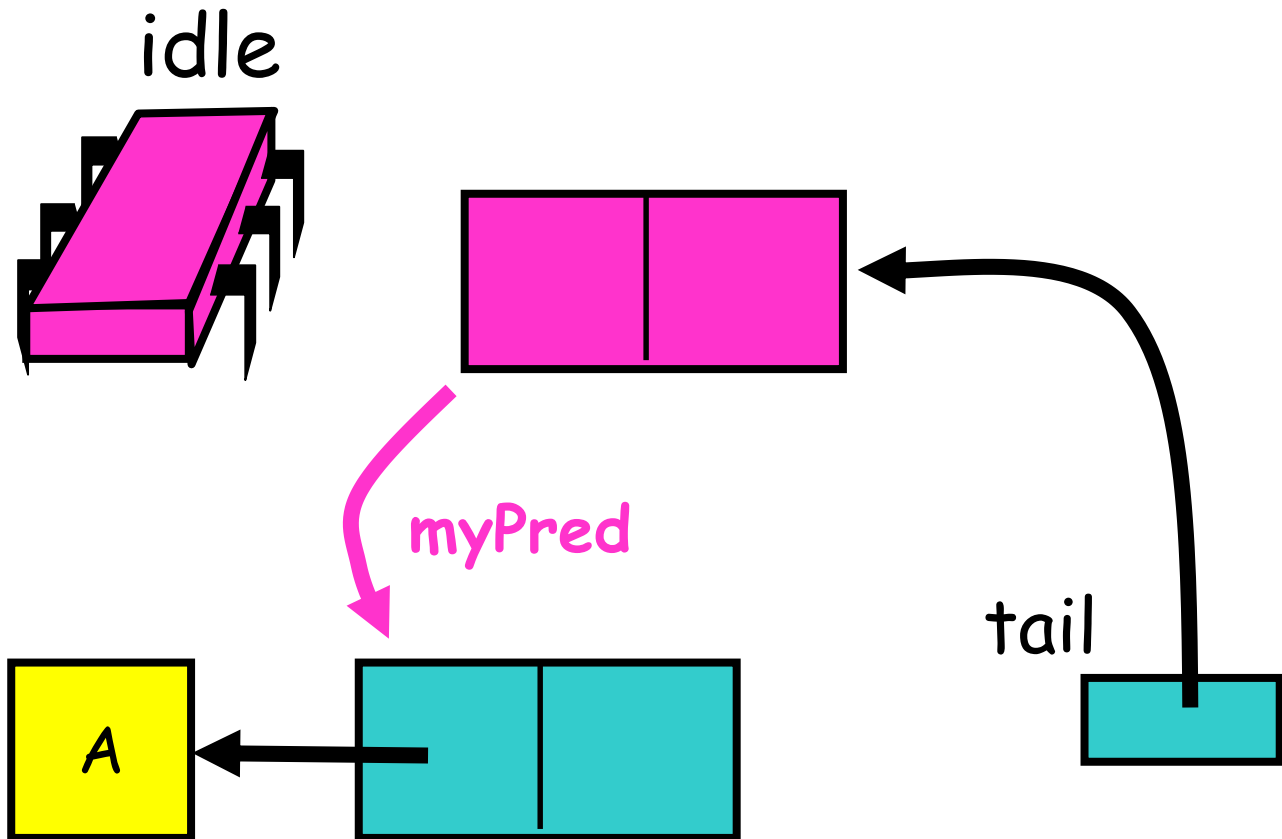
idle



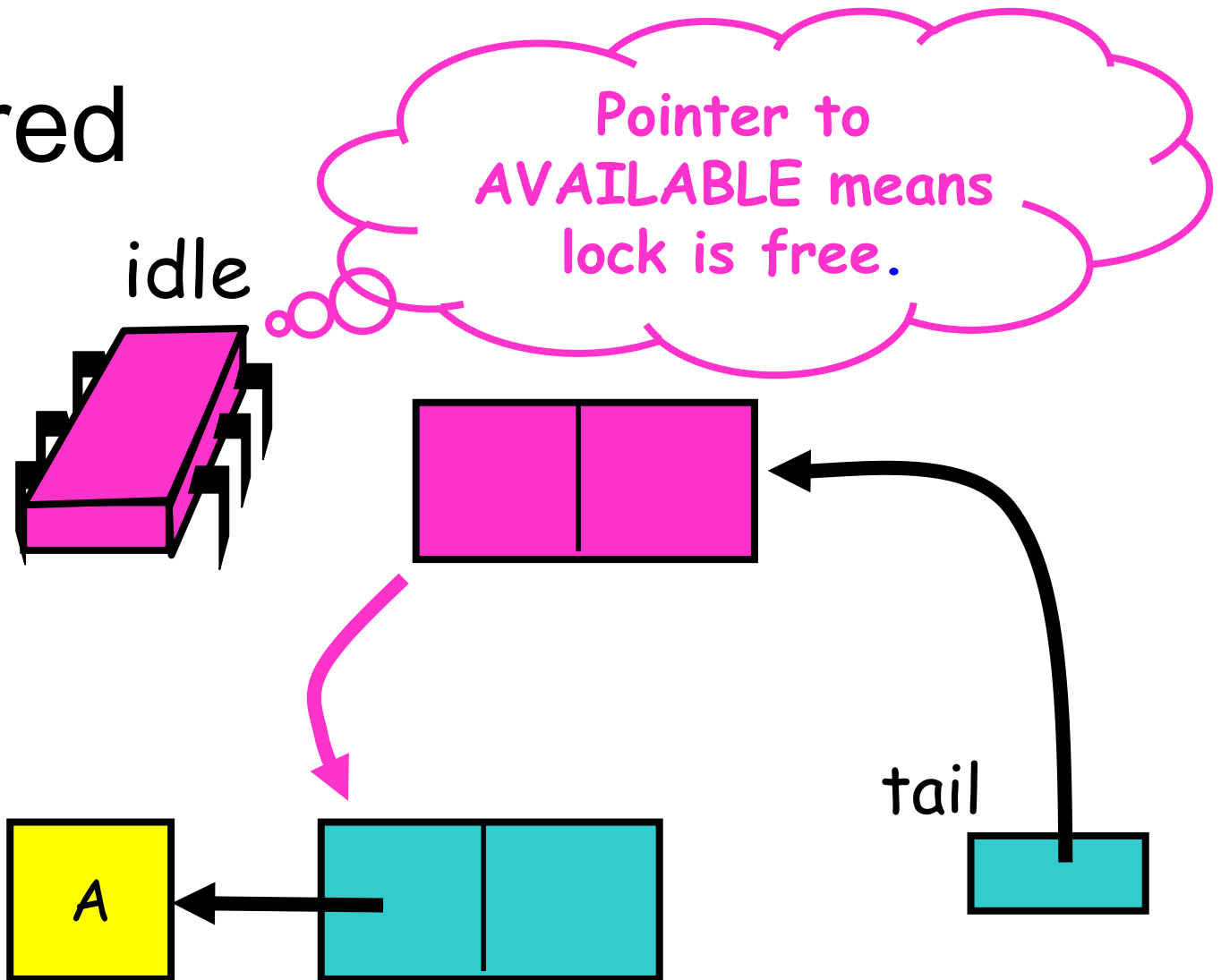
Null predecessor
means lock not
released or
aborted



Acquiring



Acquired





Time-out Lock

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

Time-out Lock

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

Distinguished node to
signify free lock

Time-out Lock

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

Tail of the queue

Time-out Lock

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

Remember my node ...



Time-out Lock

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

...

Time-out Lock

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

Create & initialize node

Time-out Lock

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

Swap with tail

Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred =  
    tail.getAndSet(qnode);
```

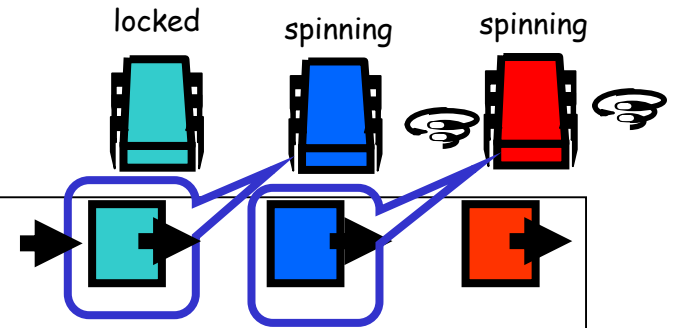
```
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;
```

```
    }
```

```
    ...
```

If predecessor absent or
released, we are done

Time-out Lock



...

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

...

Time-out Lock

...

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

Keep trying for a while ...

...

Time-out Lock

...

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

...

Spin on predecessor's
prev field

Time-out Lock

...

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

Predecessor released lock

...

Time-out Lock

...

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

...

Predecessor aborted,
advance one



Time-out Lock

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

What do I do when I time out?

Time-out Lock

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

Do I have a successor? If CAS
fails: I do have a successor,
tell it about myPred

Time-out Lock

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

If CAS succeeds: no successor,
simply return false



Time-Out Unlock

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

Time-out Unlock

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

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

Timing-out Lock

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

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



Composite Locks

- Spin lock algorithms impose trade-offs
- Most spin locks have advantages and disadvantages



Composite Locks

■ Backoff Lock:

- + provide trivial timeout protocols
- are not scalable
- may have critical section underutilization if timeout parameters are not well-tuned.



Composite Locks

■ Queue locks:

- + provide first-come-first-served fairness
- + fast lock release
- nontrivial protocols for abandoned nodes



Composite Locks

- Composite locks combine the best of both approaches



Composite Locks

- Idea:

- In a queue only the threads at the front of the queue require lock handoffs

- Solution:

- Keep a small number of waiting threads in a queue and have the rest use exponential backoff



Composite Locks

- Keep a short, fixed-sized array of lock nodes
- Each thread that tries to acquire the lock selects a node in the array at random
- If the node is in use, the thread backs off and tries again



Composite Locks

- Once the thread acquires a node, it enqueues that node in a TOLock-style queue
- The thread spins on the preceding node and when that node's owner signals it is done, the thread enters the critical section



Composite Locks

- Every node in the array has a state:
 - FREE, WAITING, RELEASED, ABORTED
- Actions depend on the state of the randomly selected node



Composite Lock

- A FREE node is available for other threads to acquire
- A WAITING node is linked into the queue and either in the critical section or waiting to enter
- A node becomes RELEASED when the owner leaves the critical section
- If a node is enqueued but the owner has quit, the node is ABORTED



Composite Lock

- A thread acquires the lock in three steps:
 - Acquires a node in the waiting array
 - Enqueues the node in the queue
 - Waits until the node reaches the head of the queue

Qnode:

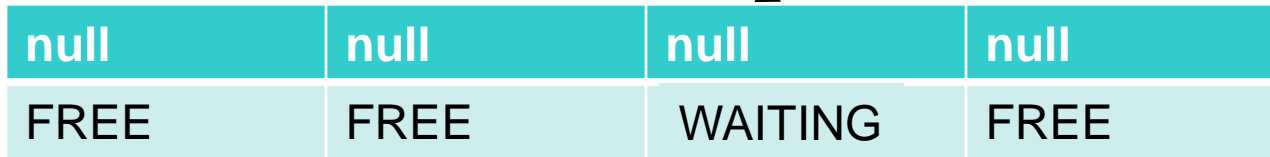
pred

state



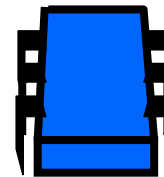
myPred

Tail:



null	null	null	null
FREE	FREE	WAITING	FREE

myPred = null



In CS

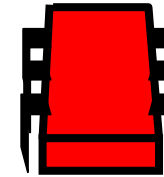
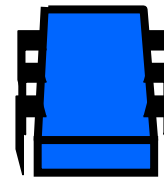
Randomly
select array
location

Tail:

null	null	null	null
FREE	FREE	WAITING	WAITING

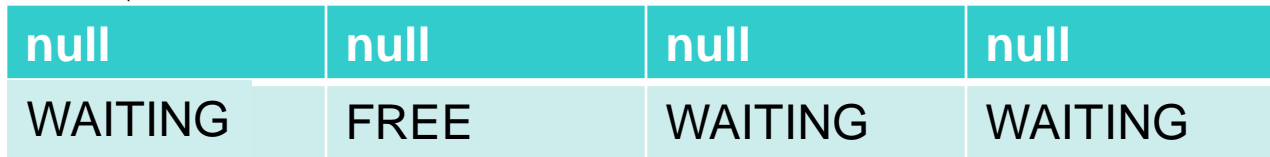
myPred = null

myPred



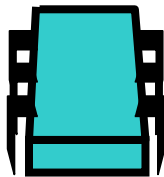
In CS

Tail:

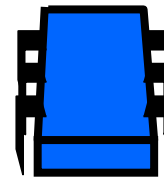


null	null	null	null
WAITING	FREE	WAITING	WAITING

myPred

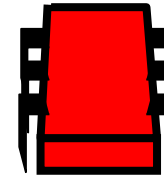


myPred = null



In CS

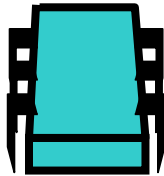
myPred



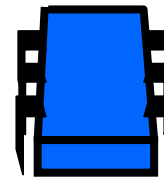
Tail:

null	null	null	
WAITING	FREE	WAITING	ABORTED

myPred

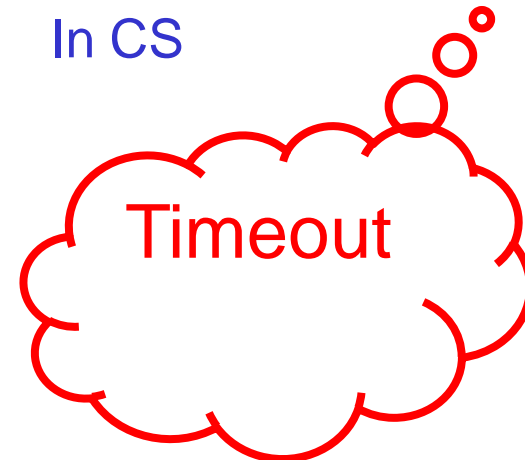
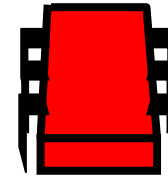


myPred = null



In CS

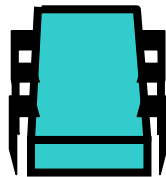
myPred



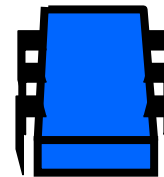
Tail:

null	null	null	
WAITING	FREE	WAITING	FREE

myPred

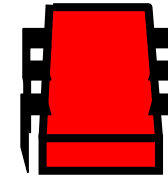


myPred = null



In CS

myPred



Better
clean this
up

Composite Lock

```
public class CompositeLock implements  
    Lock {  
    AtomicReference<Qnode> tail;  
    Qnode [] waiting;  
    ...
```

Tail field is either null or
points to last value entered
into queue (waiting array)

Composite Lock

```
public class CompositeLock implements  
    Lock {  
    AtomicReference<Qnode> tail;  
    Qnode [] waiting;  
    ...
```

Short queue of waiting
threads - implemented as
array



Composite Lock

```
enum State {FREE, WAITING, RELEASED,
            ABORTED}
class Qnode {
    AtomicReference<State> state;
    Qnode pred;

    public Qnode() {
        state = new
        AtomicReference<State>
        (State.FREE);
    }
}
```

Composite Lock

```
enum State {FREE, WAITING, RELEASED,
            ABORTED}
class Qnode {
    AtomicReference<State> state;
    Qnode pred;
    public Qnode() {
        state = new
        AtomicReference<State>
        (State.FREE);
    }
}
```

Starting states for all threads is FREE



Composite Lock

```
public boolean tryLock() {  
    Qnode node = acquireNode();  
    Qnode pred = spliceNode(node);  
    waitForPredecessor(pred, node);  
}
```


Composite Lock

```
public boolean tryLock() {  
    Qnode node = acquireNode();  
    Qnode pred = spliceNode(node);  
    waitForPredecessor(pred, node);  
}
```

Randomly select node from
waiting array and return
available node

Composite Lock

```
public boolean tryLock() {  
    Qnode node = acquireNode();  
    Qnode pred = spliceNode(node);  
    waitForPredecessor(pred, node);  
}
```

Add selected node to queue

Composite Lock

```
public boolean tryLock() {  
    Qnode node = acquireNode();  
    Qnode pred = spliceNode(node);  
    waitForPredecessor(pred, node);  
}
```

Add selected node to queue



acquireNode()

- Thread selects a node at random
- Tries to acquire the node by setting its state from FREE to WAITING
- If it fails it examines the state
 - If ABORTED or RELEASED it recycles the node



Composite Lock

```
private Qnode acquireNode() {  
    Qnode node = waiting[random.nextInt()];  
    while (true) {  
        if (state.compareAndSet(FREE, WAITING))  
            return node;  
    }  
}
```

Composite Lock

Choose random location
in waiting array

```
private Qnode acquireNode() {  
    Qnode node = waiting[random.nextInt()];  
    while (true) {  
        if (state.compareAndSet(FREE, WAITING))  
            return node;  
    }  
}
```

Composite Lock

If state was FREE then
no problem

```
private Qnode acquireNode() {  
    Qnode node = waiting[random.nextInt()];  
    while (true) {  
        if (state.compareAndSet(FREE, WAITING))  
            return node;  
    }  
}
```

If state ABORTED or
RELEASED

Composite Lock

```
currTail = tail;  
if (state == ABORTED || state ==  
RELEASED) {  
    if (node == currTail) {  
        Qnode myPred = null;  
        if (state == ABORTED)  
            myPred = node.pred;  
        if (tail.compareAndSet(  
            currTail, myPred)  
            node.state.set(WAITING);  
        return node;  
    }  
}
```


Composite Lock

Can only be recycled if
last node in queue

```
currTail = tail;
if (state == ABORTED || state ==
RELEASED) {
    if (node == currTail) {
        Qnode myPred = null;
        if (state == ABORTED)
            myPred = node.pred;
        if (tail.compareAndSet(
            currTail, myPred)
            node.state.set(WAITING);
        return node;
    }
}
```

Composite Lock

If last node is
ABORTED tail is set to
predecessor otherwise

```
currTail = tail;
if (state == ABORTED || state ==  
RELEASED) {  
    if (node == currTail) {  
        Qnode myPred = null;  
        if (state == ABORTED)  
            myPred = node.pred;  
        if (tail.compareAndSet(  
            currTail, myPred)  
            node.state.set(WAITING);  
        return node;  
    }  
}
```

Composite Lock

Successfully recycle
node and set to
WAITING

```
currTail = tail;
if (state == ABORTED || state ==
RELEASED) {
    if (node == currTail) {
        Qnode myPred = null;
        if (state == ABORTED)
            myPred = node.pred;
        if (tail.compareAndSet(
            currTail, myPred)
            node.state.set(WAITING);
        return node;
    }
}
```



spliceNode()

- Once node is acquired, the node is spliced into the queue
- Thread tries to set tail to node
- If timeout, set node to FREE and throw TimeoutException
- If succeeds return prior value of tail (node's predecessor)



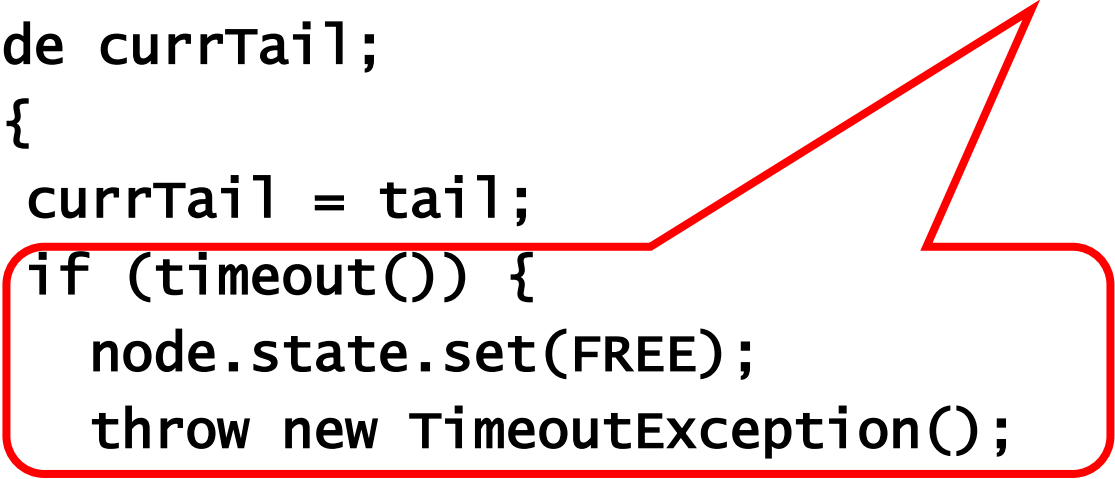
Composite Lock

```
private Qnode spliceNode(Qnode node) {
    Qnode currTail;
    do {
        currTail = tail;
        if (timeout()) {
            node.state.set(FREE);
            throw new TimeoutException();
        }
    } while (!tail.compareAndSet(currTail,
    node));
    return currTail;
}
```

If waited too long...

Composite Lock

```
private Qnode spliceNode(Qnode node) {  
    Qnode currTail;  
    do {  
        currTail = tail;  
        if (timeout()) {  
            node.state.set(FREE);  
            throw new TimeoutException();  
        }  
    } while (!tail.compareAndSet(currTail,  
        node));  
    return currTail;  
}
```



Composite Lock

```
private Qnode spliceNode(Qnode node) {  
    Qnode currTail;  
    do {  
        currTail = tail;  
        if (timeout()) {  
            node.state.set(FREE);  
            throw new TimeoutException();  
        }  
    } while (!tail.compareAndSet(currTail,  
        node);  
    return currTail;  
}
```

Try to set tail to new
node



Composite Lock

- Finally thread has to wait its turn
- If predecessor is null thread can enter CS
- If predecessor is not RELEASED, check if it is ABORTED
 - If so, mark as FREE and get ABORTED node's predecessor
- If predecessor is RELEASED enter CS



Composite Lock

```
private void waitforPredecessor {  
    if (pred == null)  
        return;  
    State predState = pred.state.get();  
    while (predState != RELEASED) {  
        if (predState == ABORTED) {  
            Qnode temp = pred;  
            pred = pred.pred;  
            temp.state.set(FREE);  
        }  
    }  
}
```

Composite Lock

```
private void waitforPredecessor {  
    if (pred == null)  
        return;  
    State predState = pred.state.get();  
    while (predState != RELEASED) {  
        if (predState == ABORTED) {  
            Qnode temp = pred;  
            pred = pred.pred;  
            temp.state.set(FREE);  
        }  
    }  
}
```

If I am first in line

Composite Lock

```
private void waitforPredecessor {  
    if (pred == null)  
        return;
```

```
    State predState = pred.state.get();  
    while (predState != RELEASED) {  
        if (predState == ABORTED) {  
            Qnode temp = pred;  
            pred = pred.pred;  
            temp.state.set(FREE);  
        }  
    }
```

Spin on predecessor's
state

Composite Lock

```
private void waitforPredecessor {  
    if (pred == null)  
        return;  
    State predState = pred.state.get();  
    while (predState != RELEASED) {  
        if (predState == ABORTED) {  
            Qnode temp = pred;  
            pred = pred.pred;  
            temp.state.set(FREE);  
        }  
    }
```

If predecessor **ABORTED** set to
FREE and spin on predecessors
predecessor

Composite Lock

```
    if (timeout()) {  
        node.pred = pred;  
        node.state.set(ABORTED);  
    } throw new TimeoutException();  
    predState = pred.state.get();  
}  
pred.state.set(FREE);  
return;  
}
```

If I have waited long enough...

Composite Lock

```
    if (timeout()) {  
        node.pred = pred;  
        node.state.set(ABORTED);  
    } throw new TimeoutException();  
    predState = pred.state.get();  
}  
pred.state.set(FREE);  
return;  
}
```

When predecessor RELEASED



Composite Lock

```
public void unlock() {  
    myNode.state.set(RELEASED);  
}
```