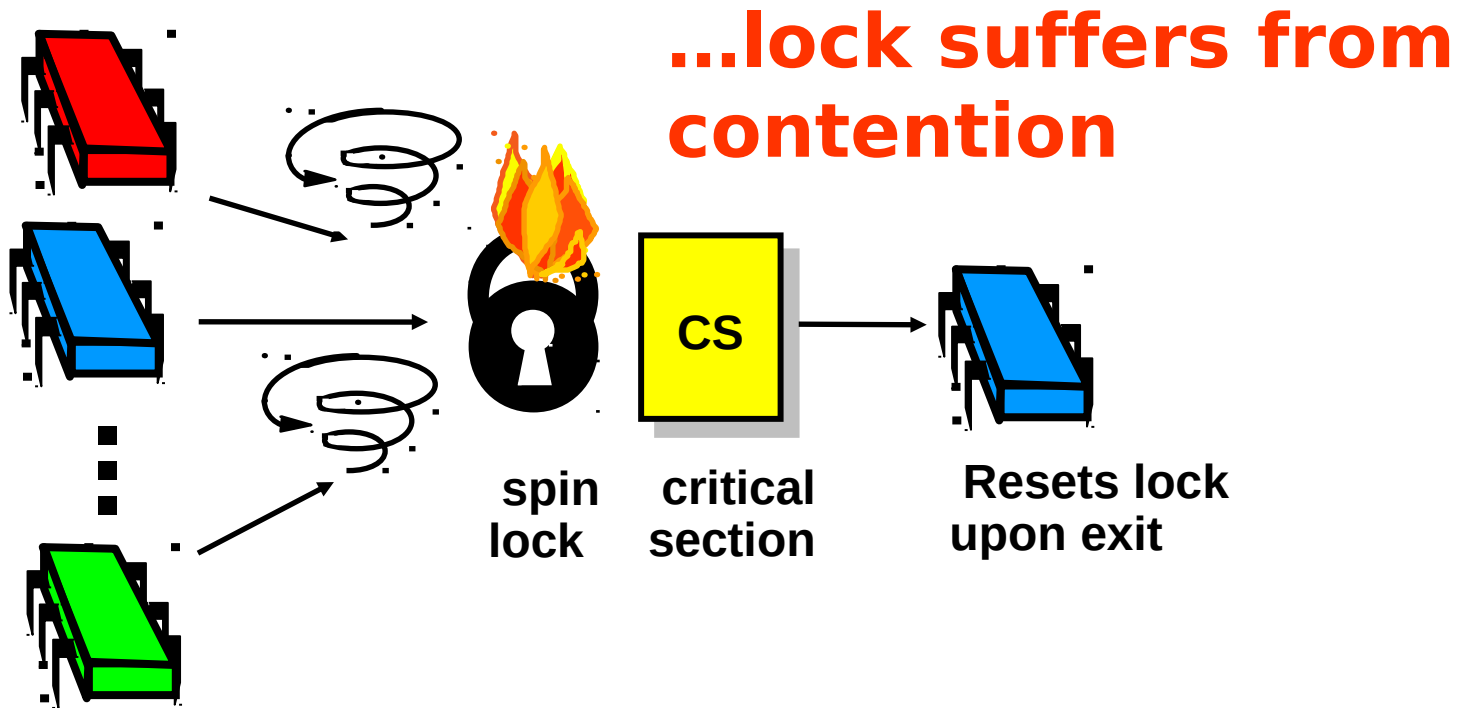


Spin Locks and Contention

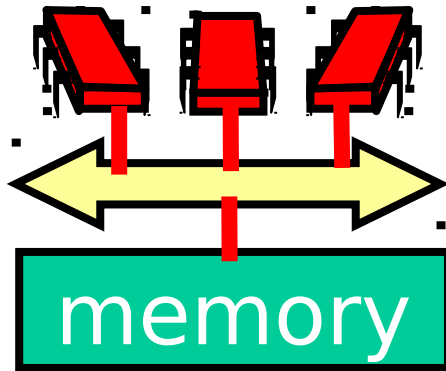
(Part 2)

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

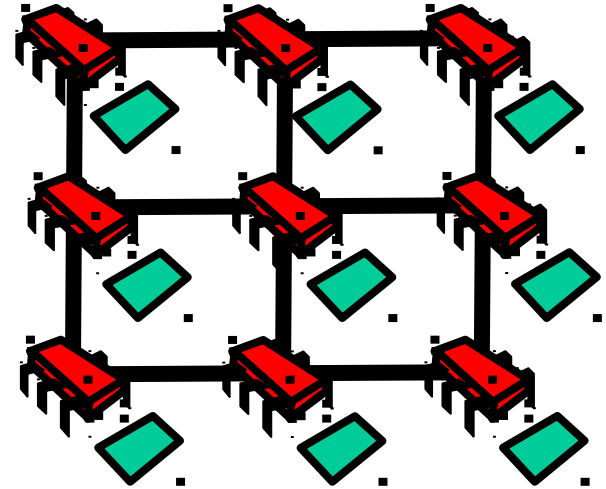
Basic Spin-Lock



MIMD Architectures



Shared Bus



Distributed

- Memory Contention
- Communication Contention
- Communication Latency

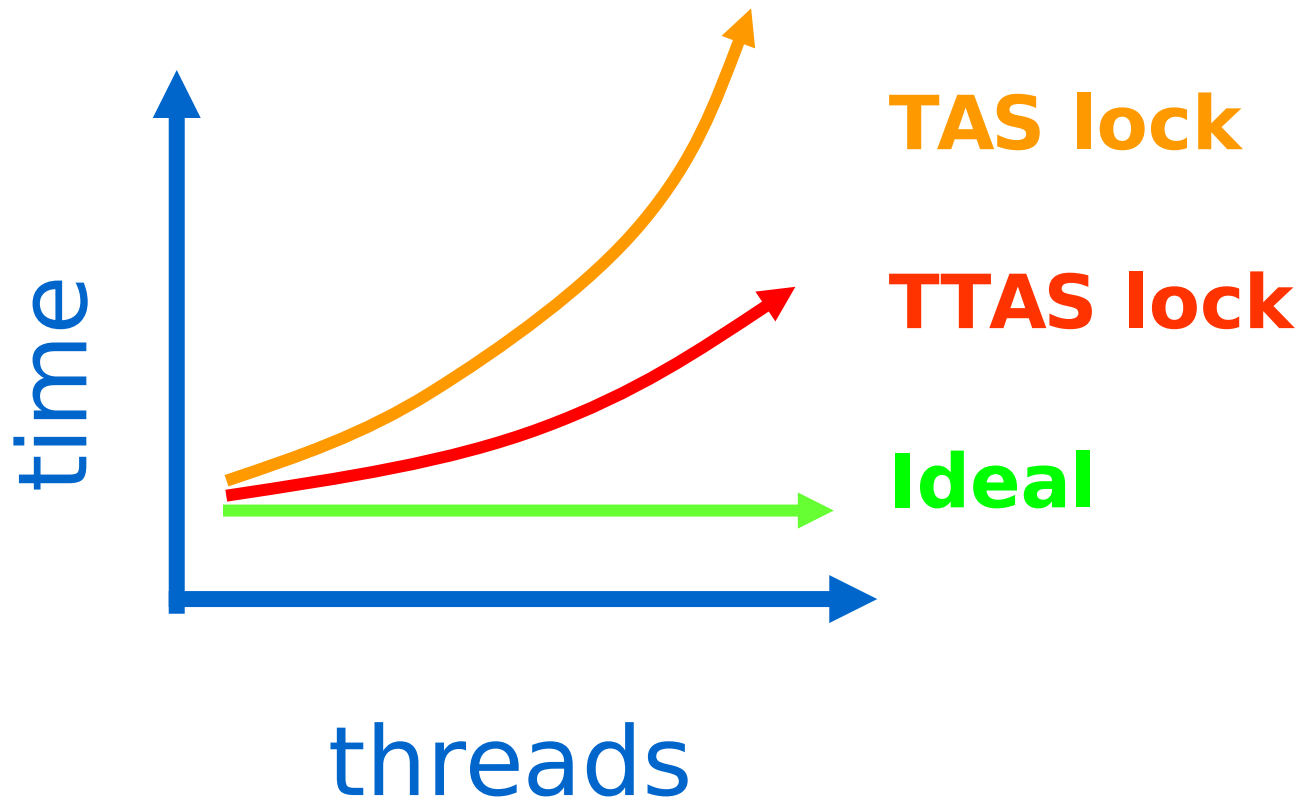
Test-and-set Lock

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

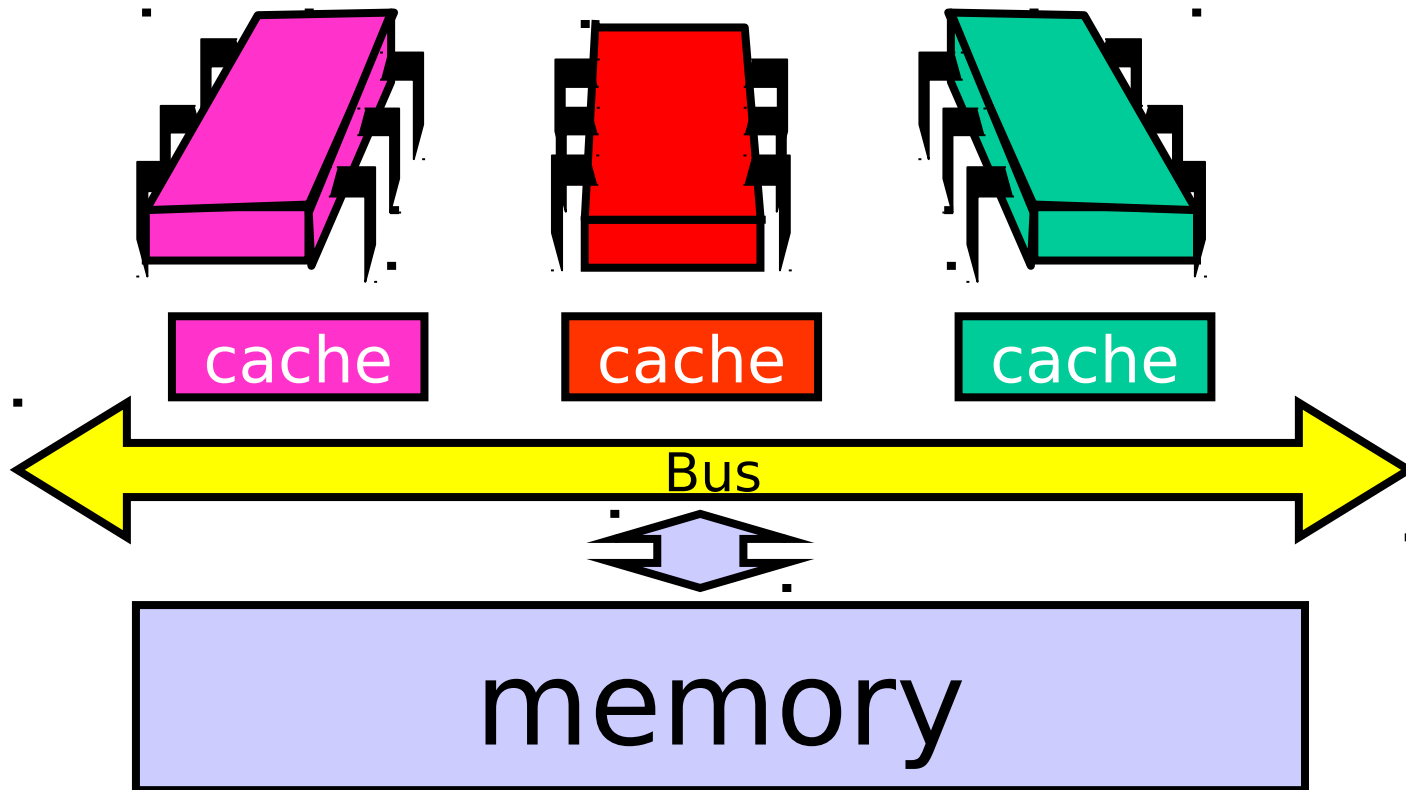
    void lock() {
        while (state.getAndSet(true)) {}
    }

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

Mystery #2

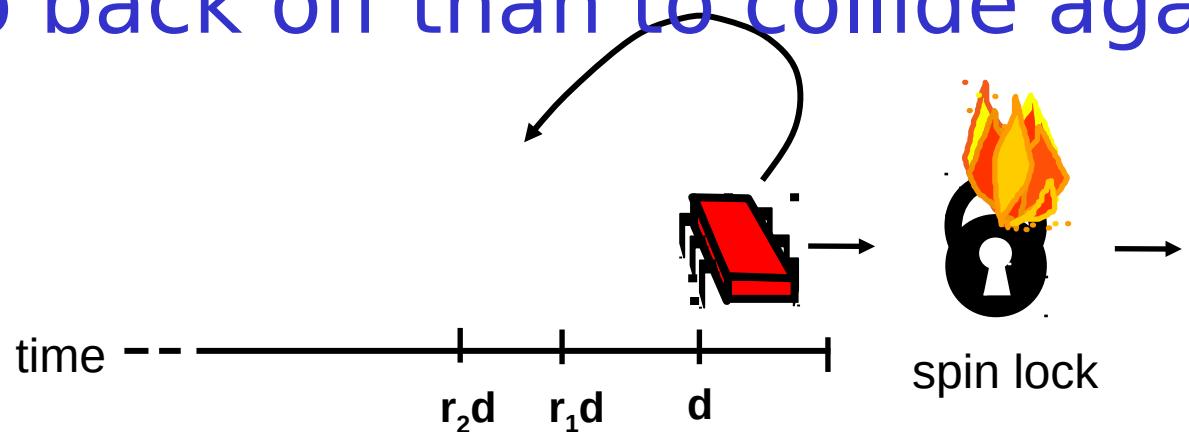


Bus-Based Architectures

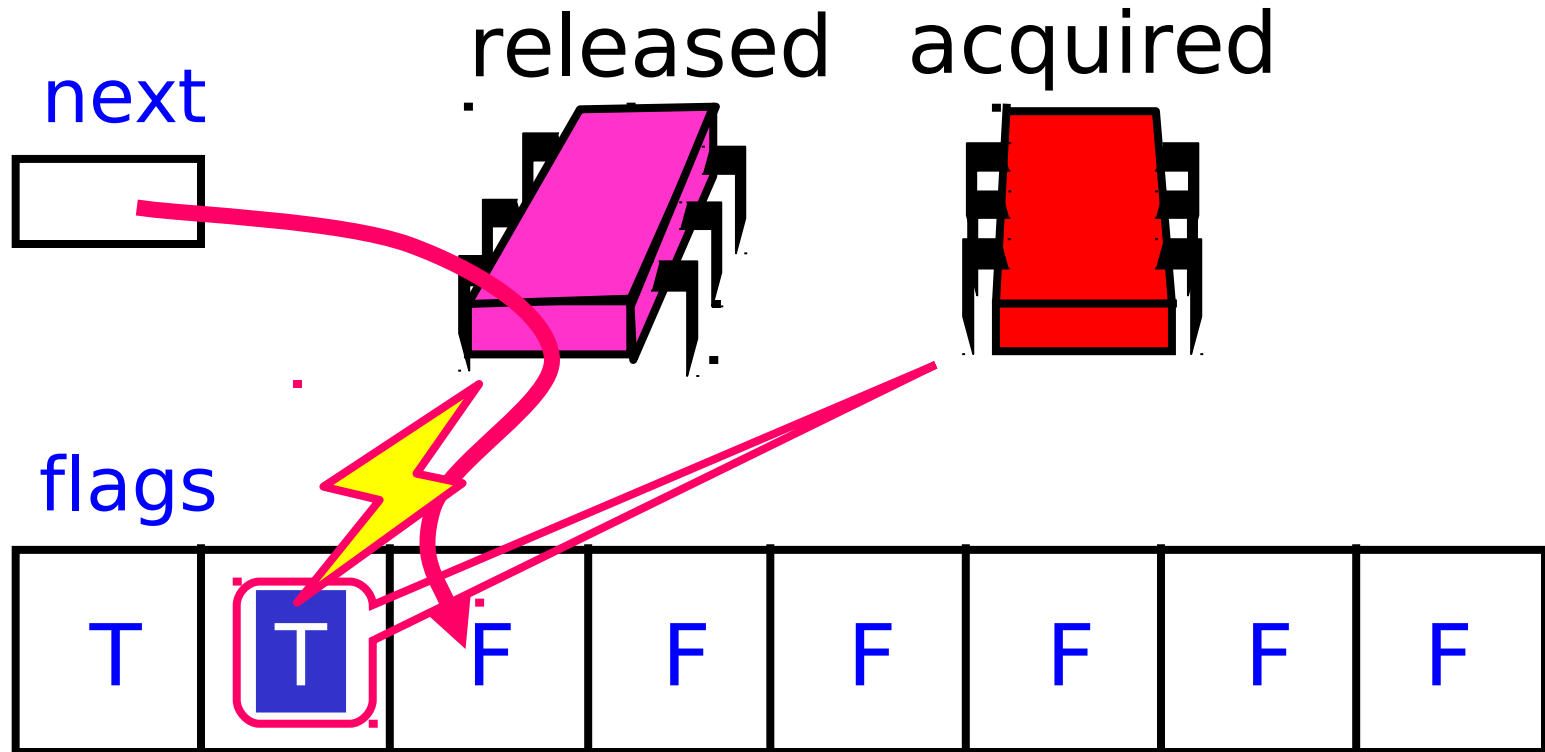


Solution: Introduce Delay

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



Anderson Queue Lock



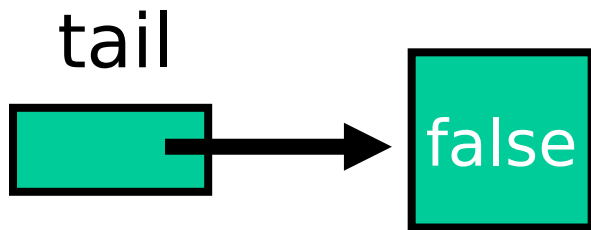
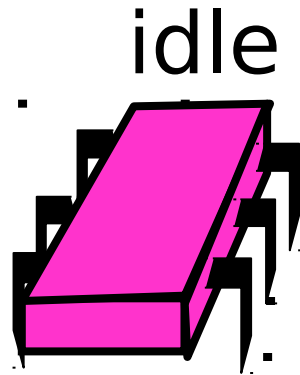
Anderson Queue Lock

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

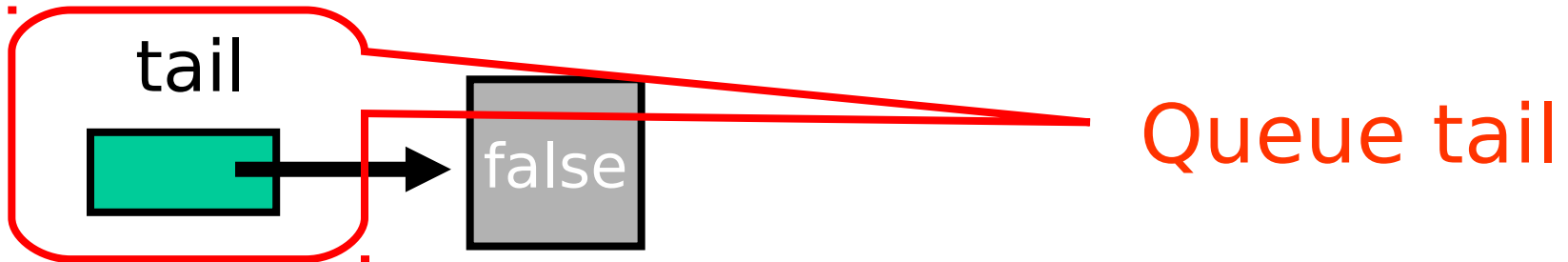
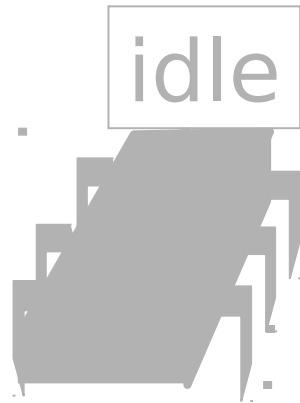
CLH Lock

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

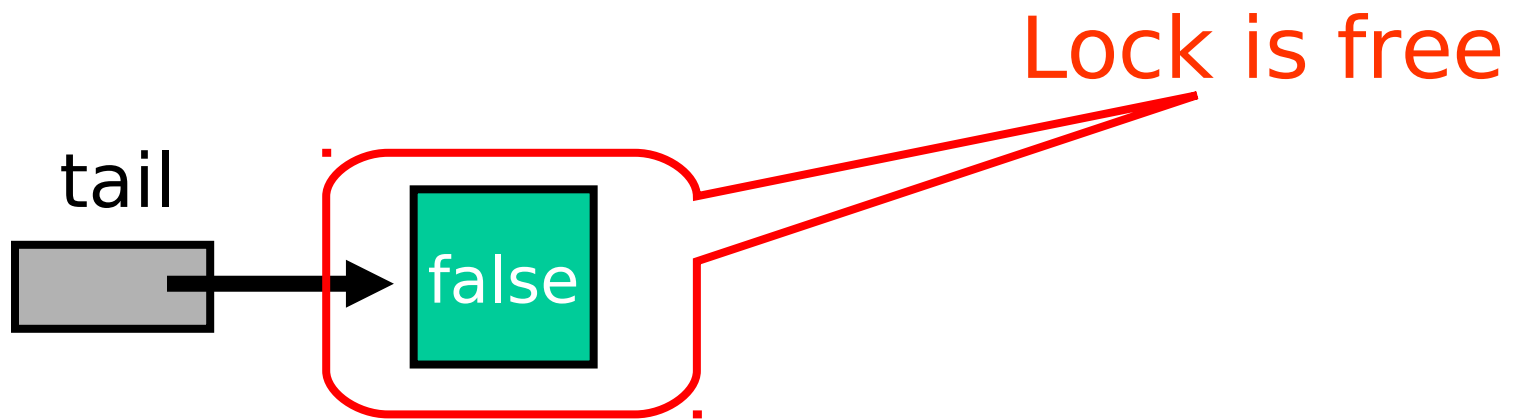
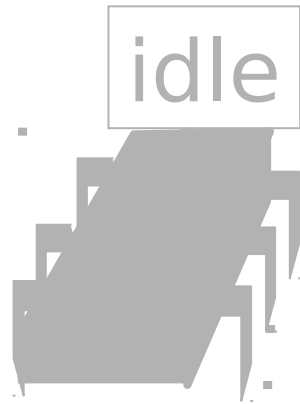
Initially



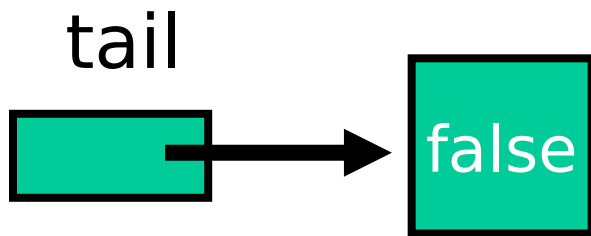
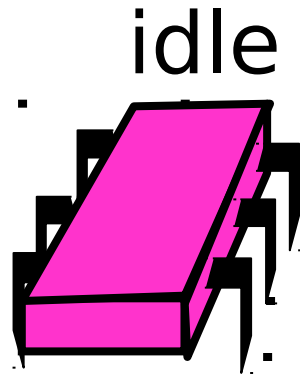
Initially



Initially

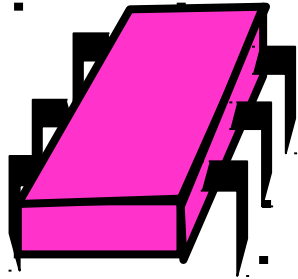


Initially

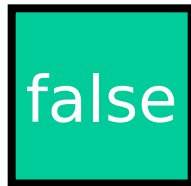


Purple Wants the Lock

acquiring

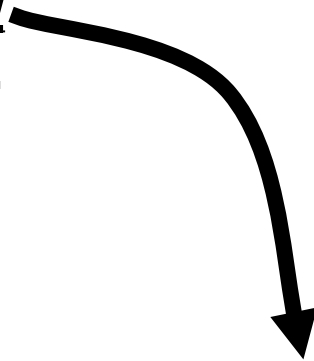
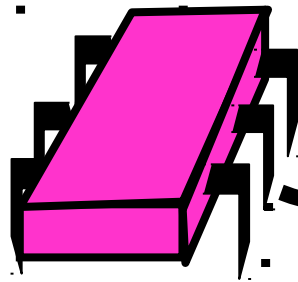


tail



Purple Wants the Lock

acquiring



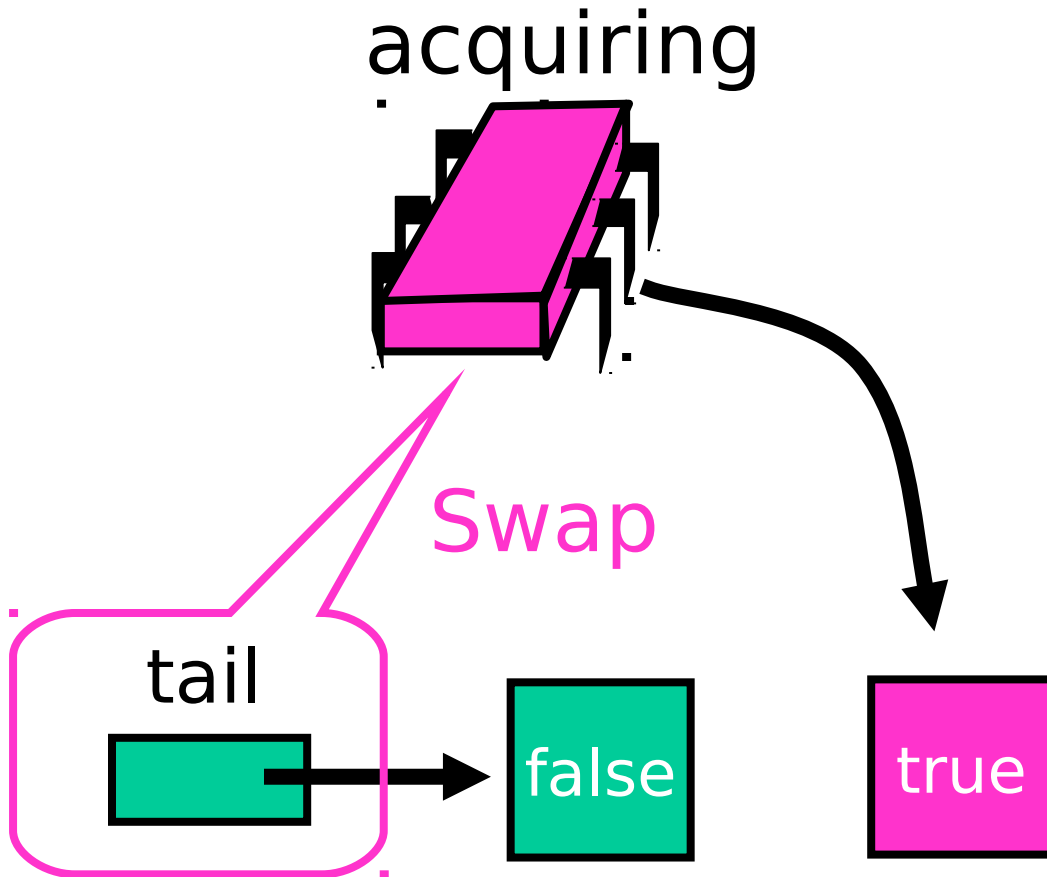
tail



false

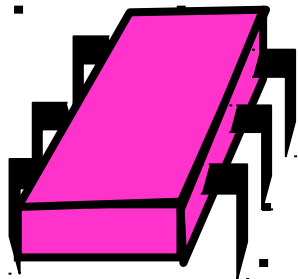
true

Purple Wants the Lock



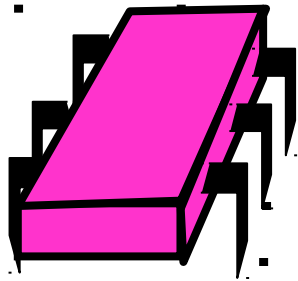
Purple Has the Lock

acquired

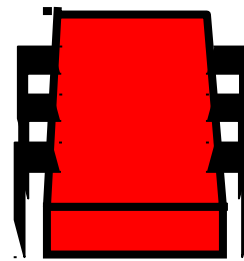


Red Wants the Lock

acquired



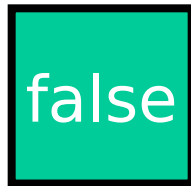
acquiring



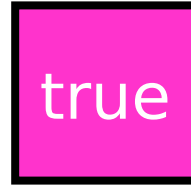
tail



false



true

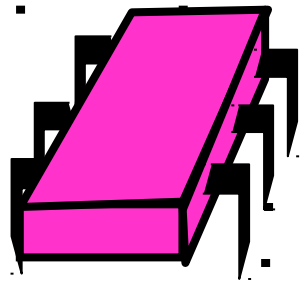


true

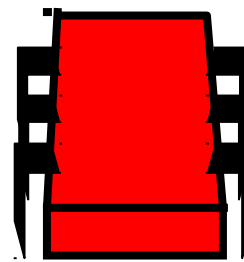


Red Wants the Lock

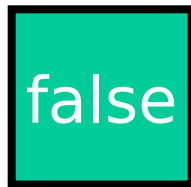
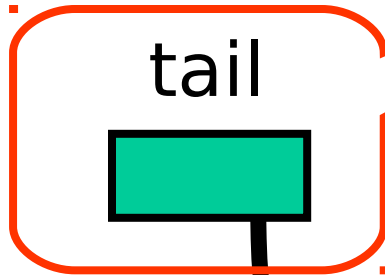
acquired



acquiring

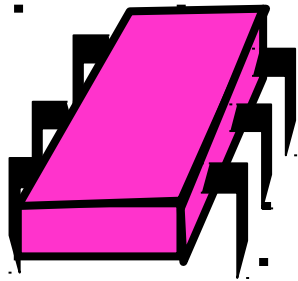


Swap

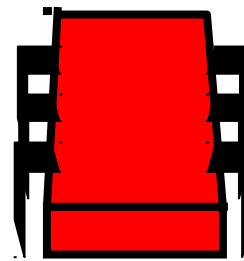


Red Wants the Lock

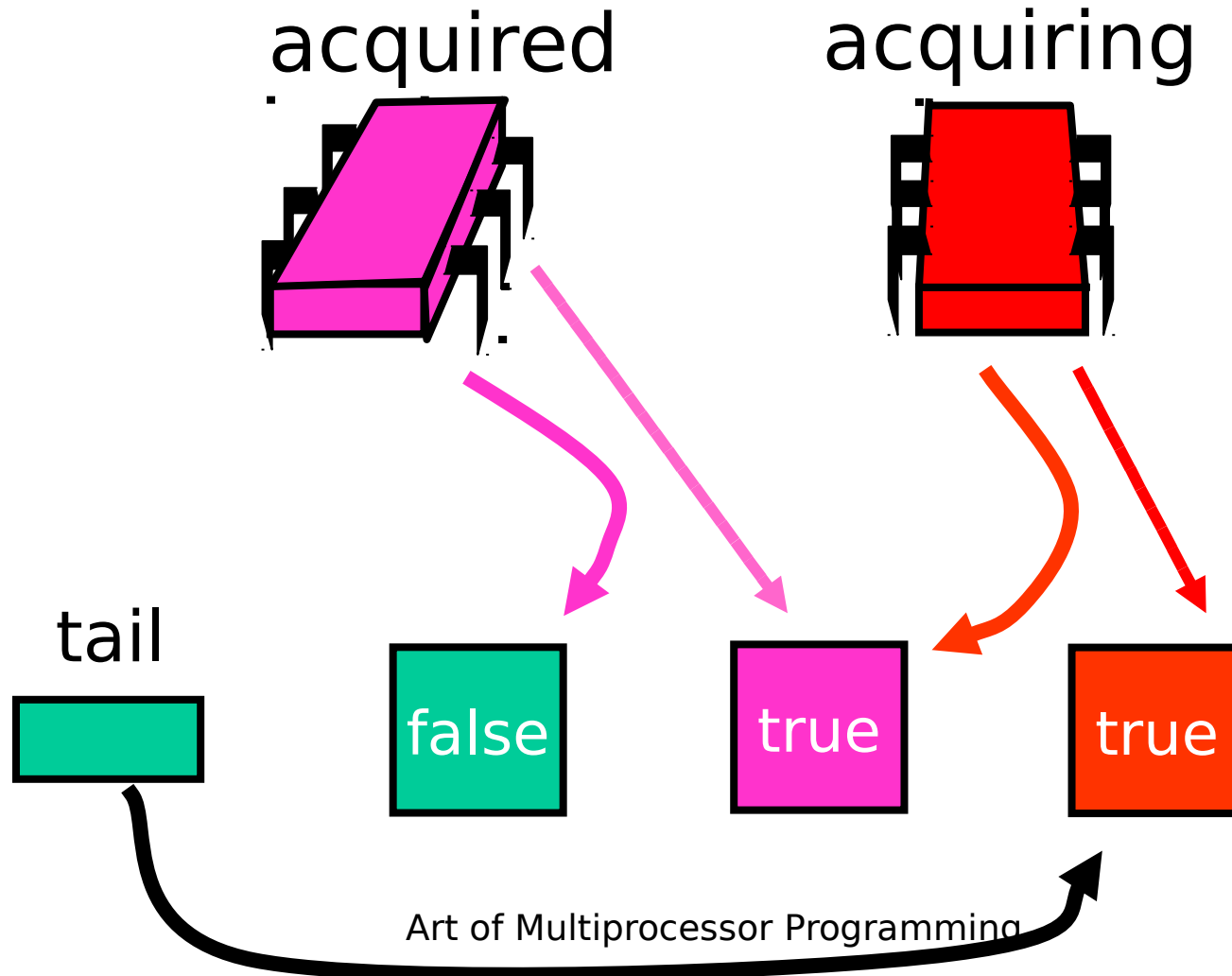
acquired



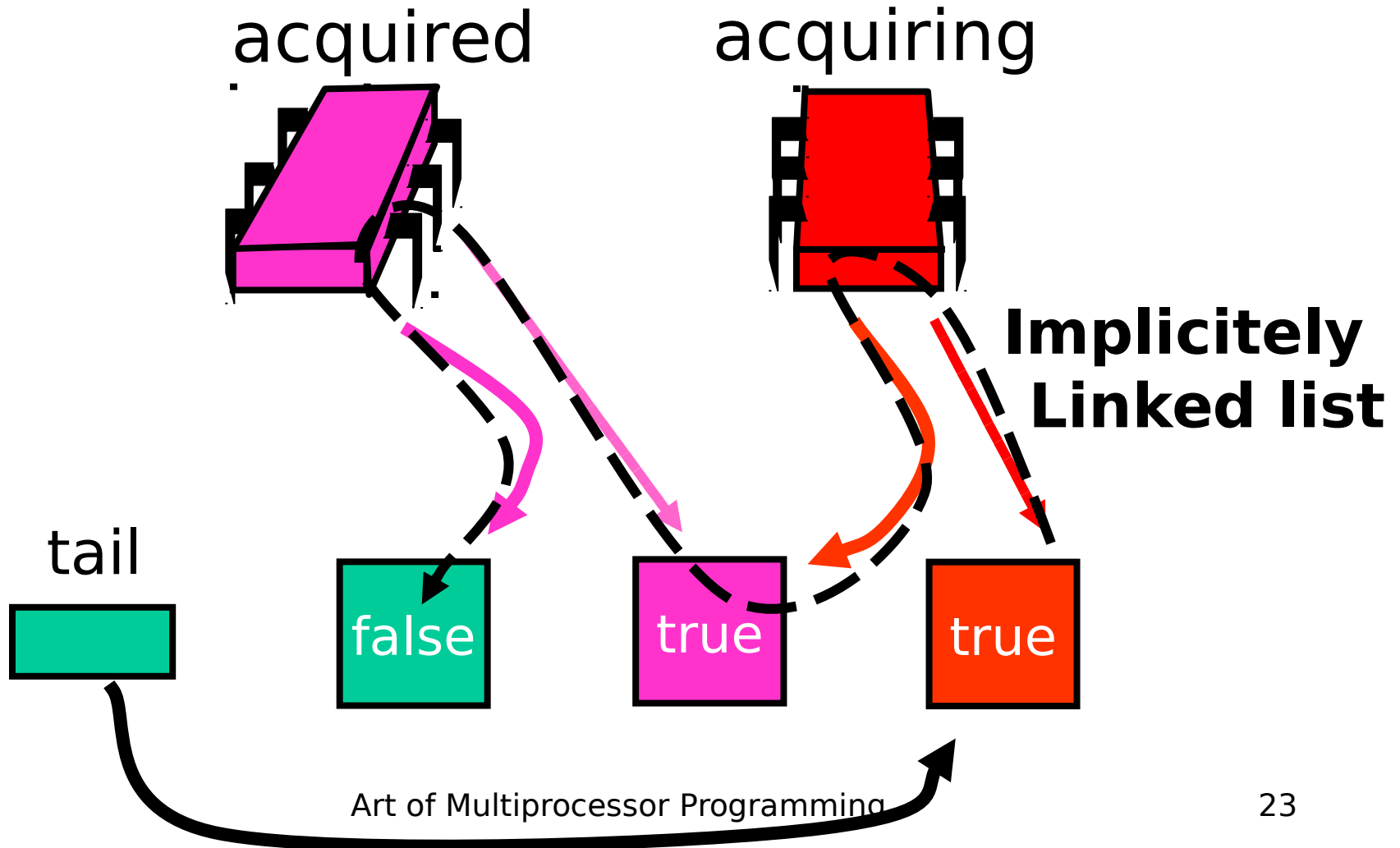
acquiring



Red Wants the Lock

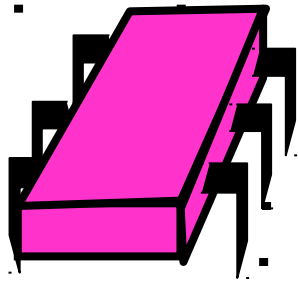


Red Wants the Lock

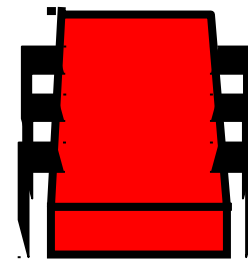


Red Wants the Lock

acquired

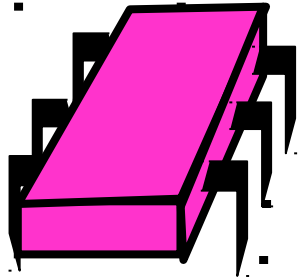


acquiring

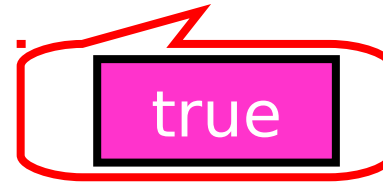
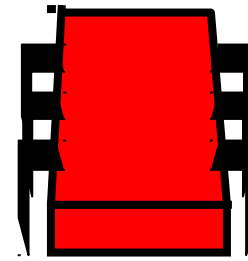


Red Wants the Lock

acquired



acquiring

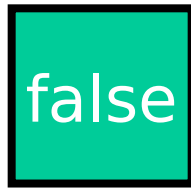


Actually, it spins on cached copy

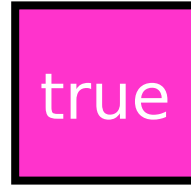
tail



false



true



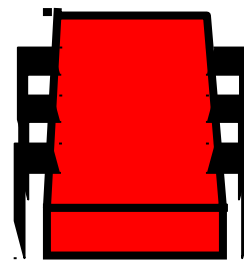
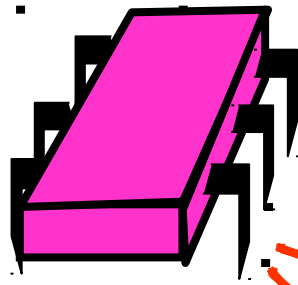
true



Purple Releases

release

acquiring



false



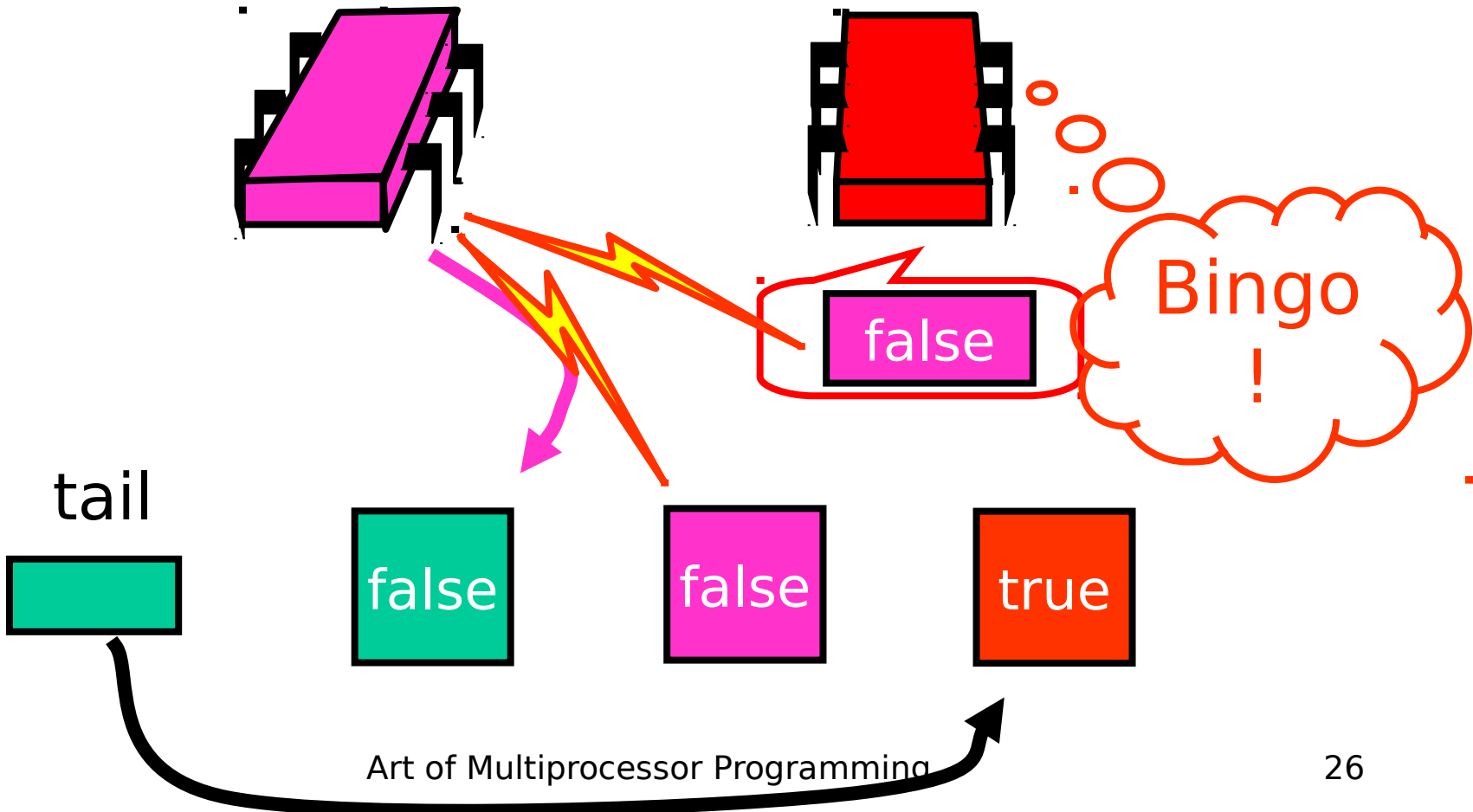
tail



false

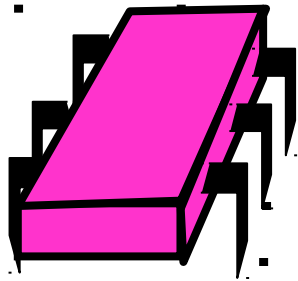
false

true

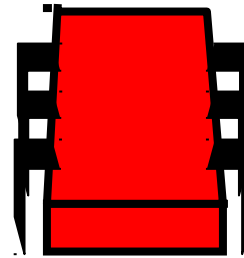


Purple Releases

released



acquired



tail



true



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(false);  
}
```

CLH Queue Lock


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

Pseudocode



CLH Queue Lock

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



**Special reset method
for ThreadLocals.**

**It does NOT
reset the content of myNode
in other Threads**

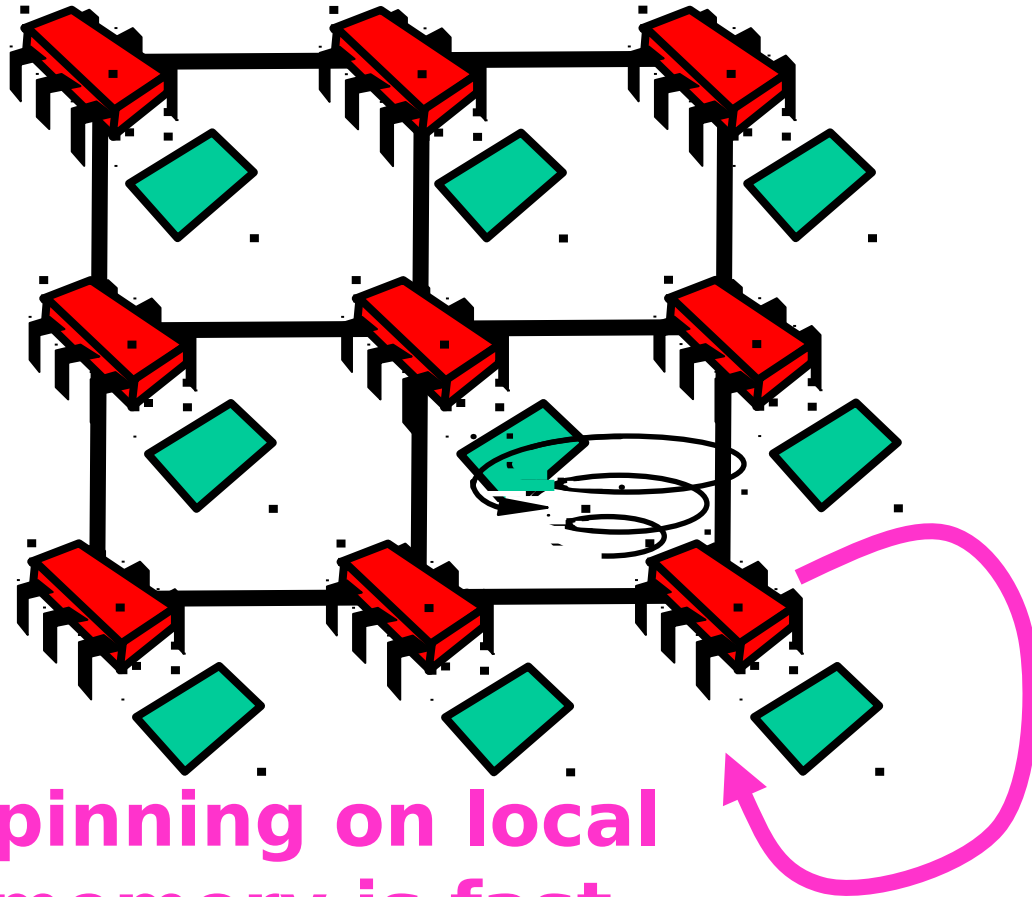
CLH Lock

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

NUMA Architecturs

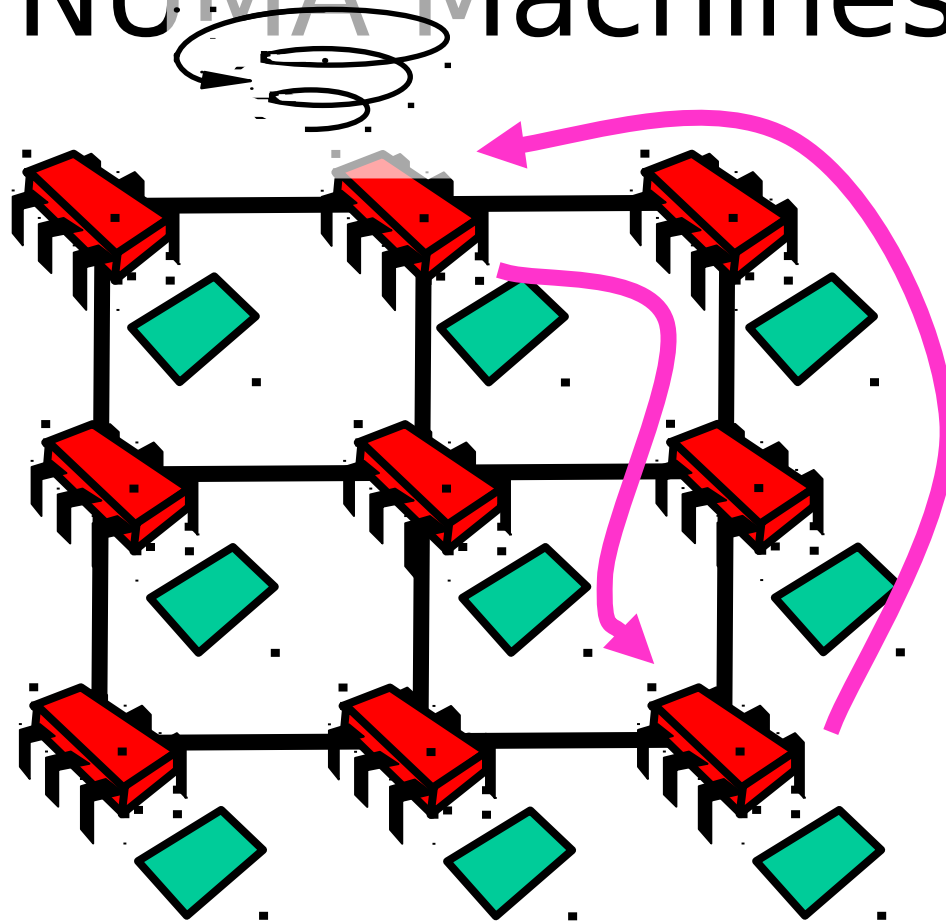
- Acronym:
 - **N**on-**U**niform **M**emory **A**rchitecture
- Illusion:
 - Flat shared memory
- Truth:
 - No caches (sometimes)
 - Some memory regions faster than others

NUMA Machines



**Spinning on local
memory is fast**

NUMA Machines



**Spinning on remote
memory is slow**

CLH Lock

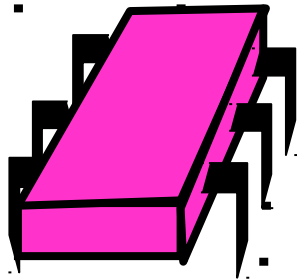
- Each thread spin's on predecessor's memory
- Could be far away ...

MCS Lock

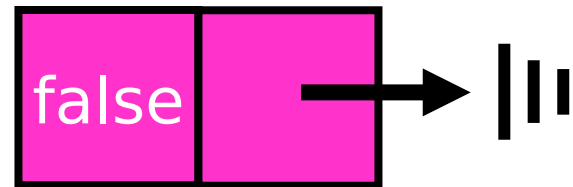
- FIFO order
- Spin on local memory only
- Small, Constant-size overhead

Acquiring

acquiring



(allocate Qnode)

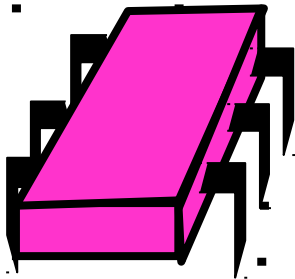


queue



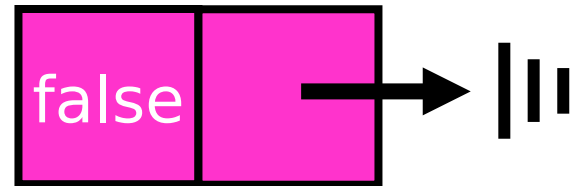
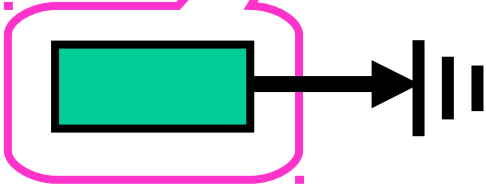
Acquiring

acquired

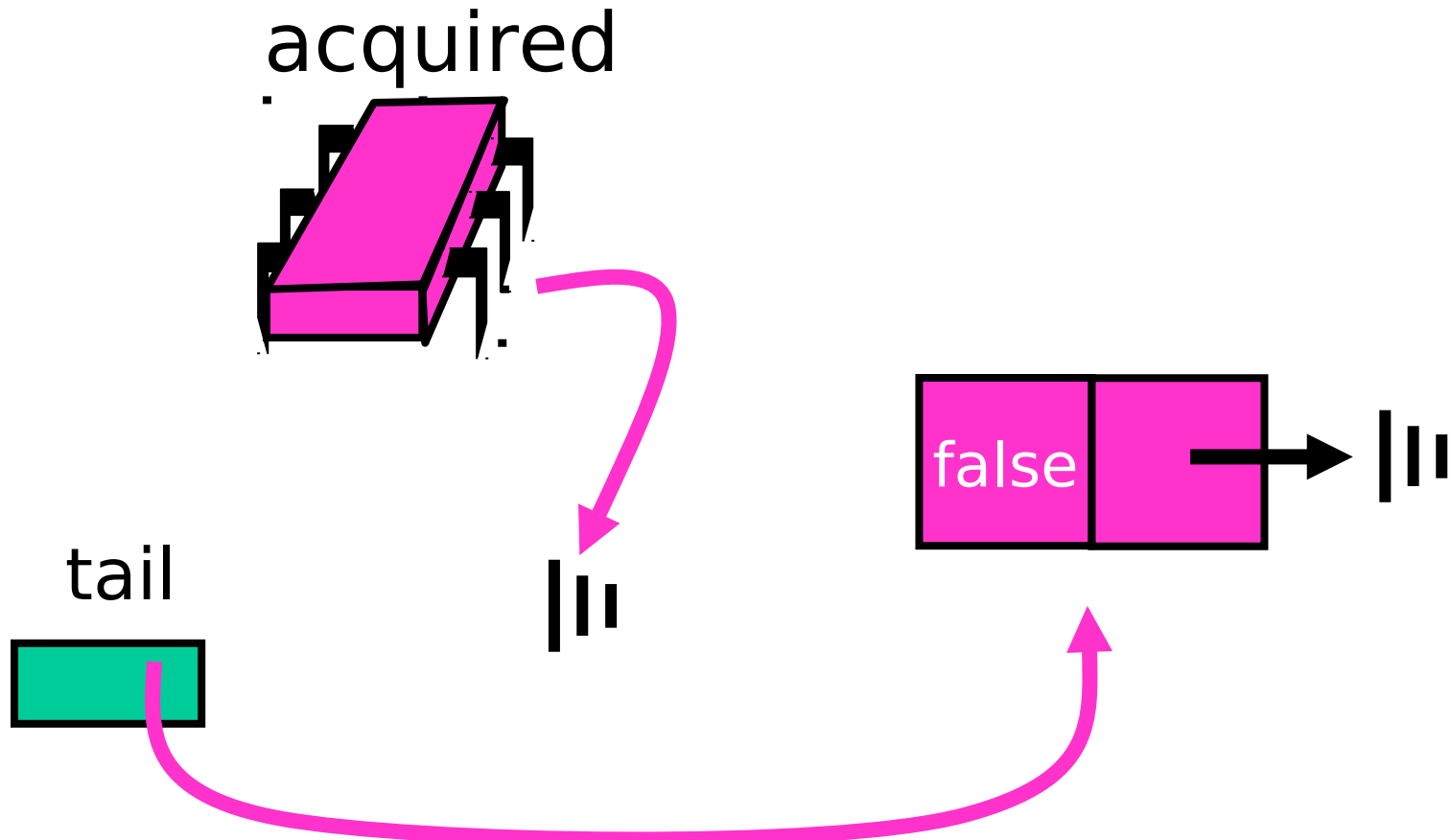


swap

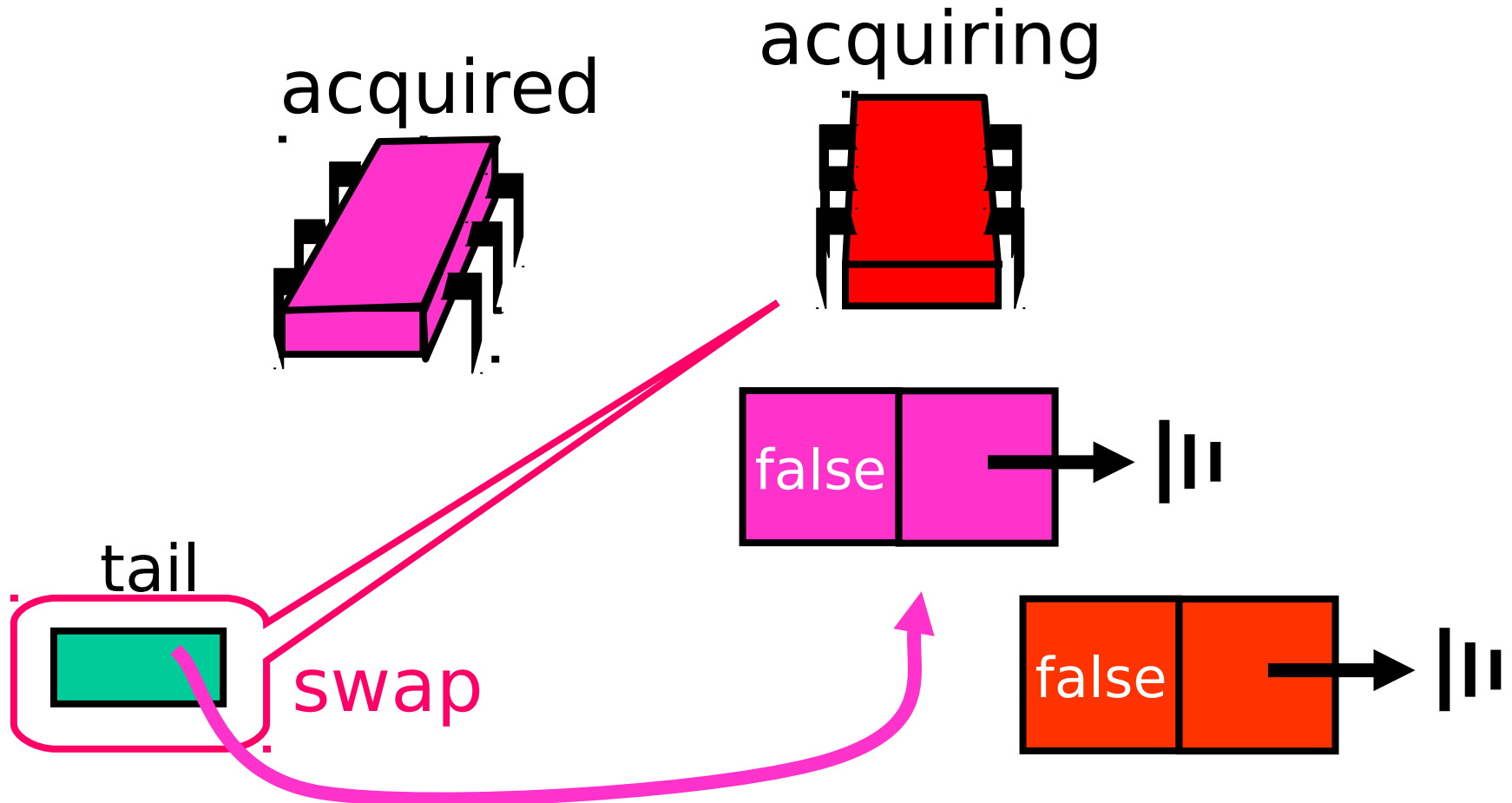
tail



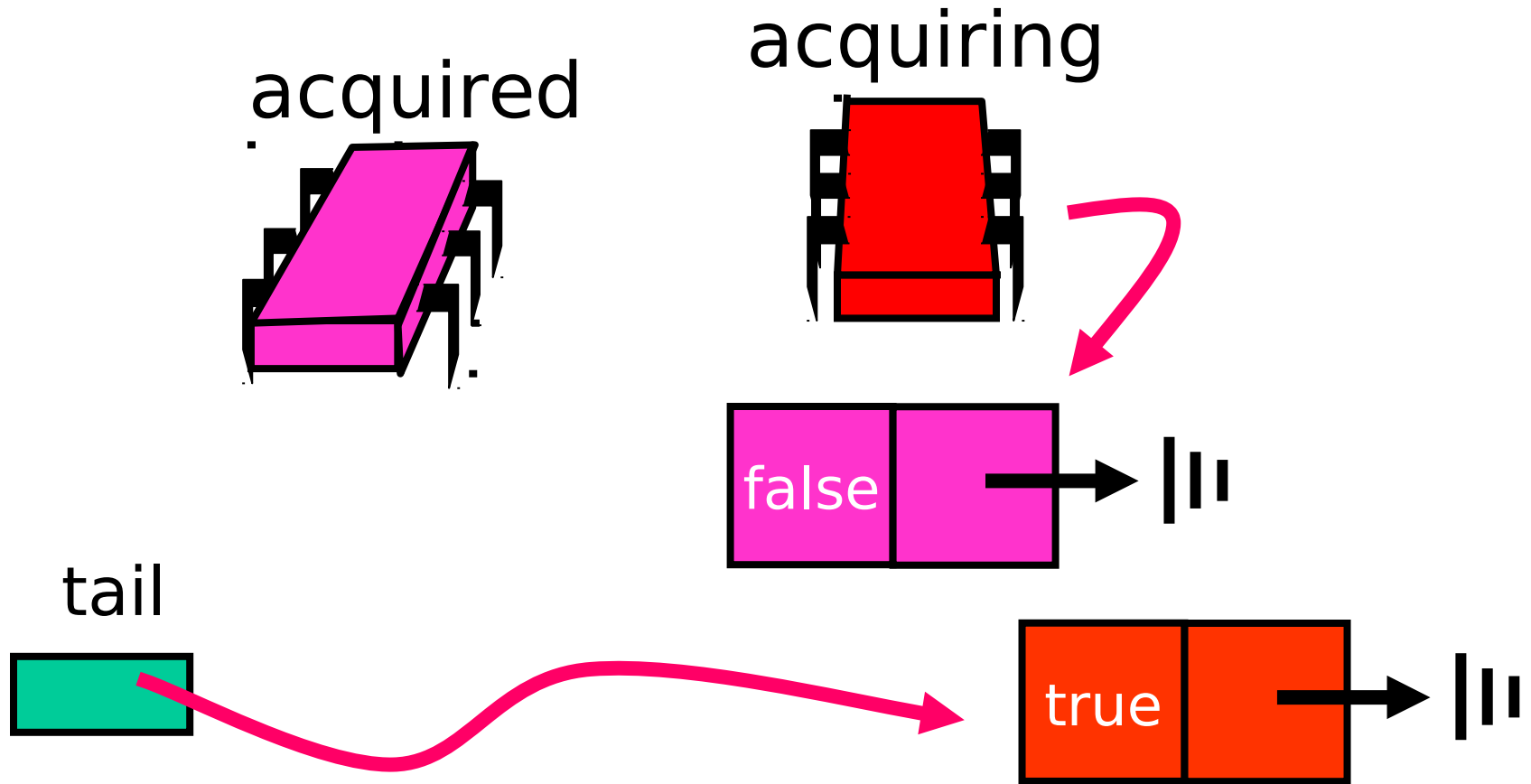
Acquired



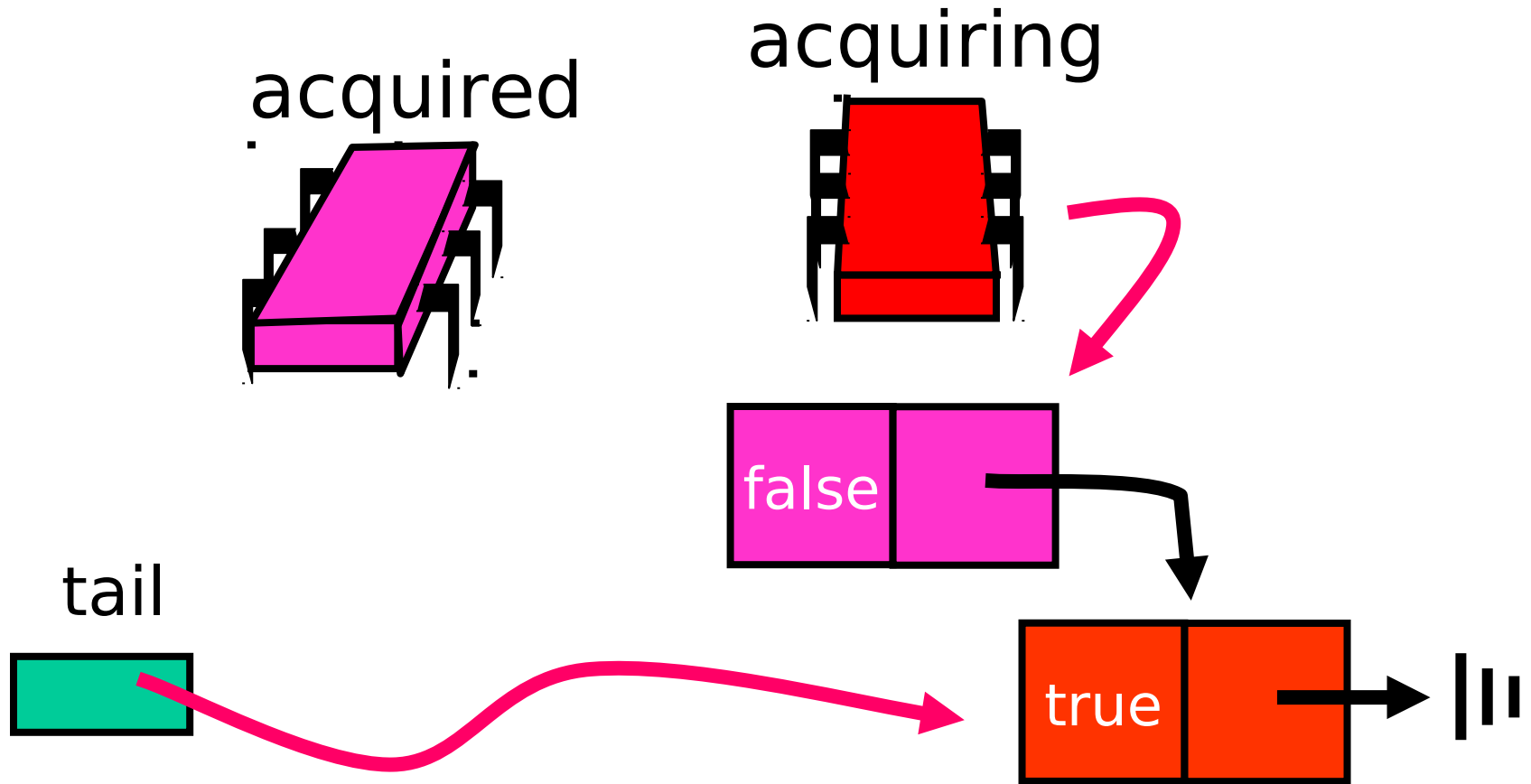
Acquiring



Acquiring



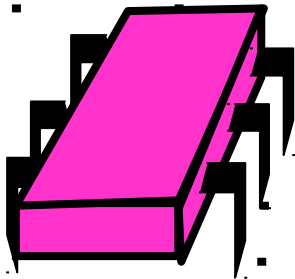
Acquiring



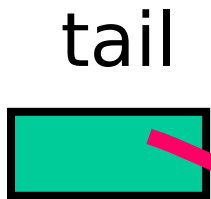
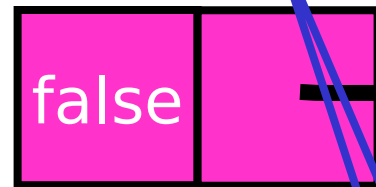
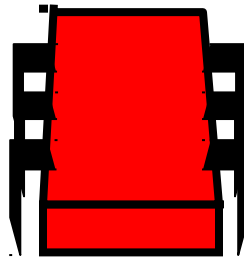
Acquiring



acquired



acquiring

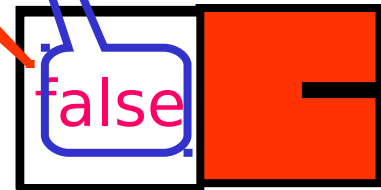
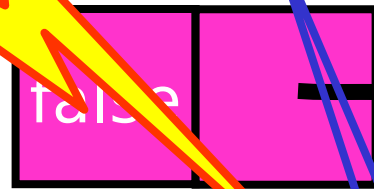
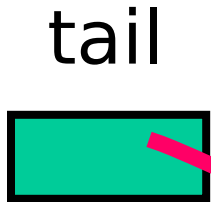
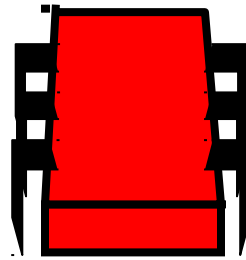
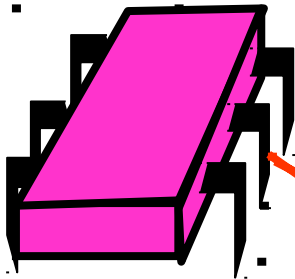


Acquiring

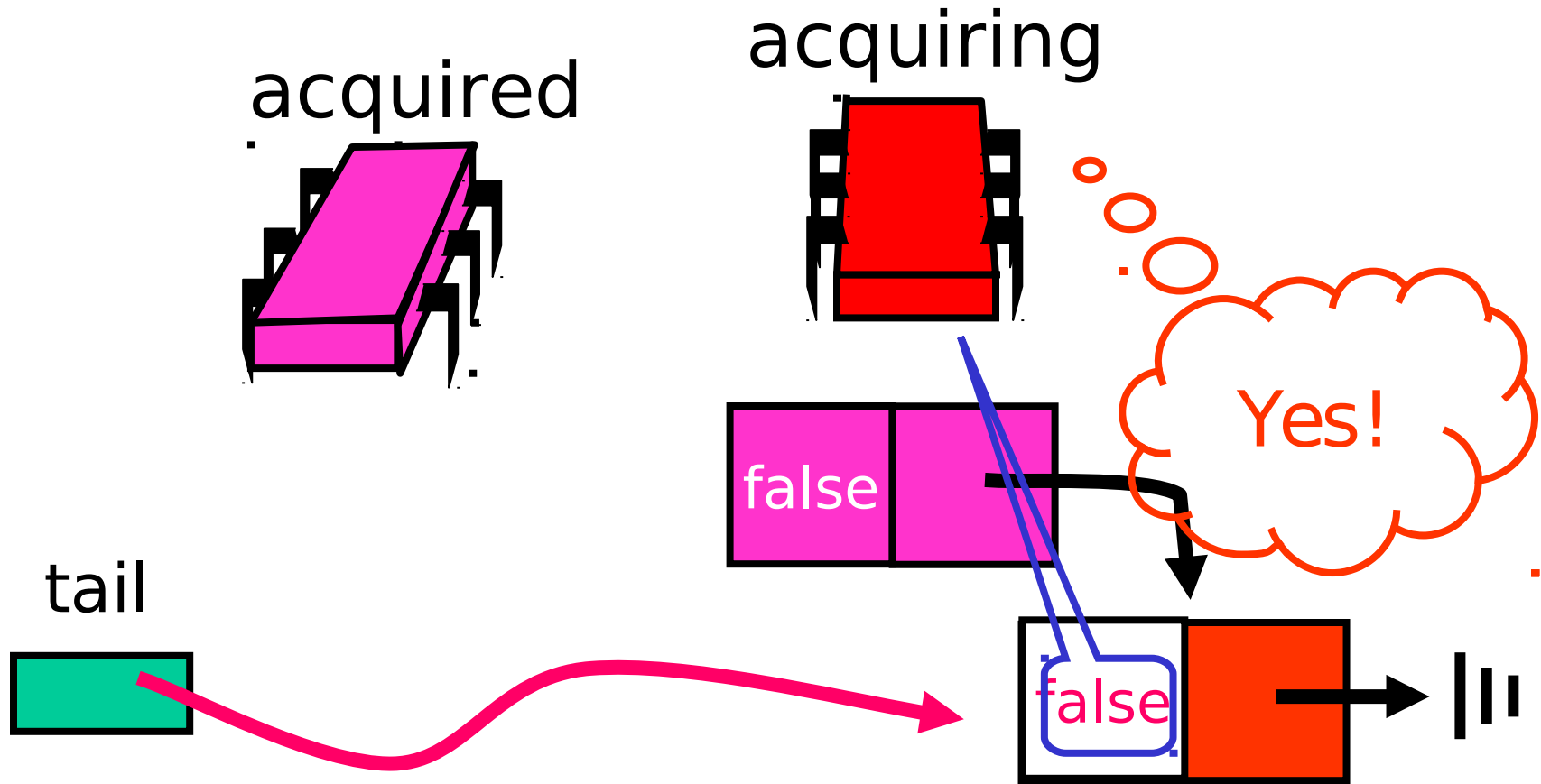


acquiring

acquired




Acquiring



MCS Queue Lock

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

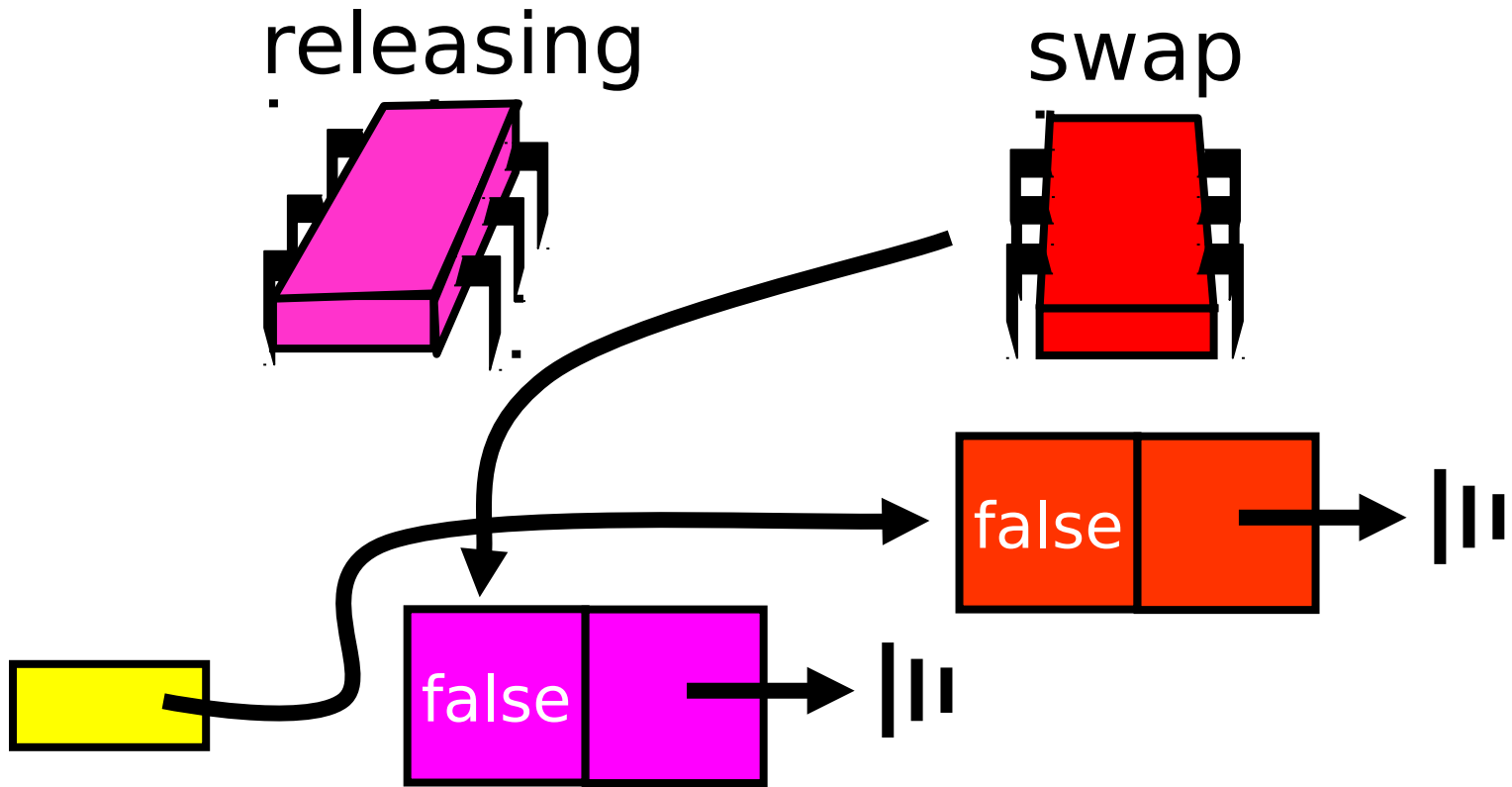
MCS Queue Lock

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

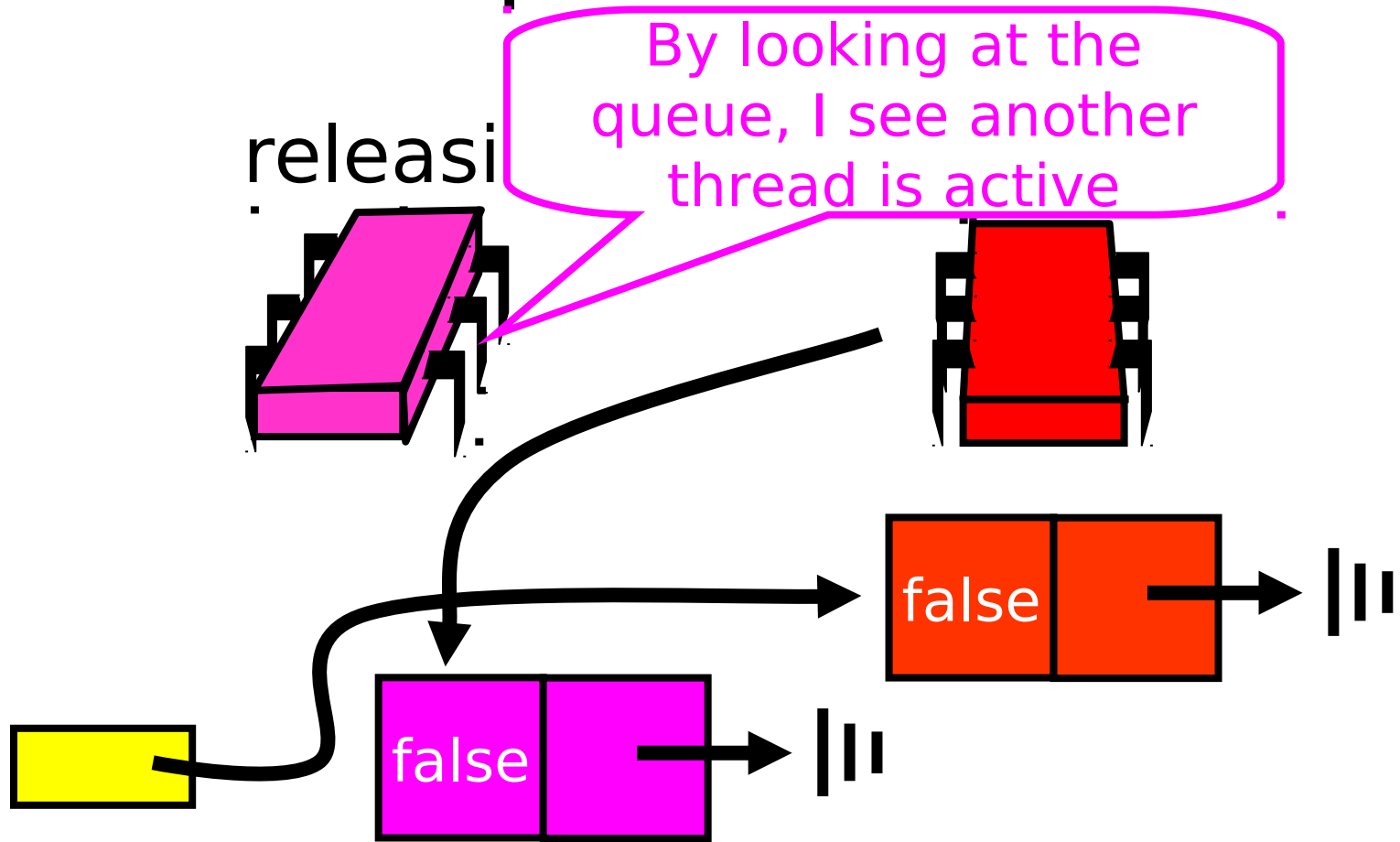

MCS Queue Unlock

```
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;
    }
}
```

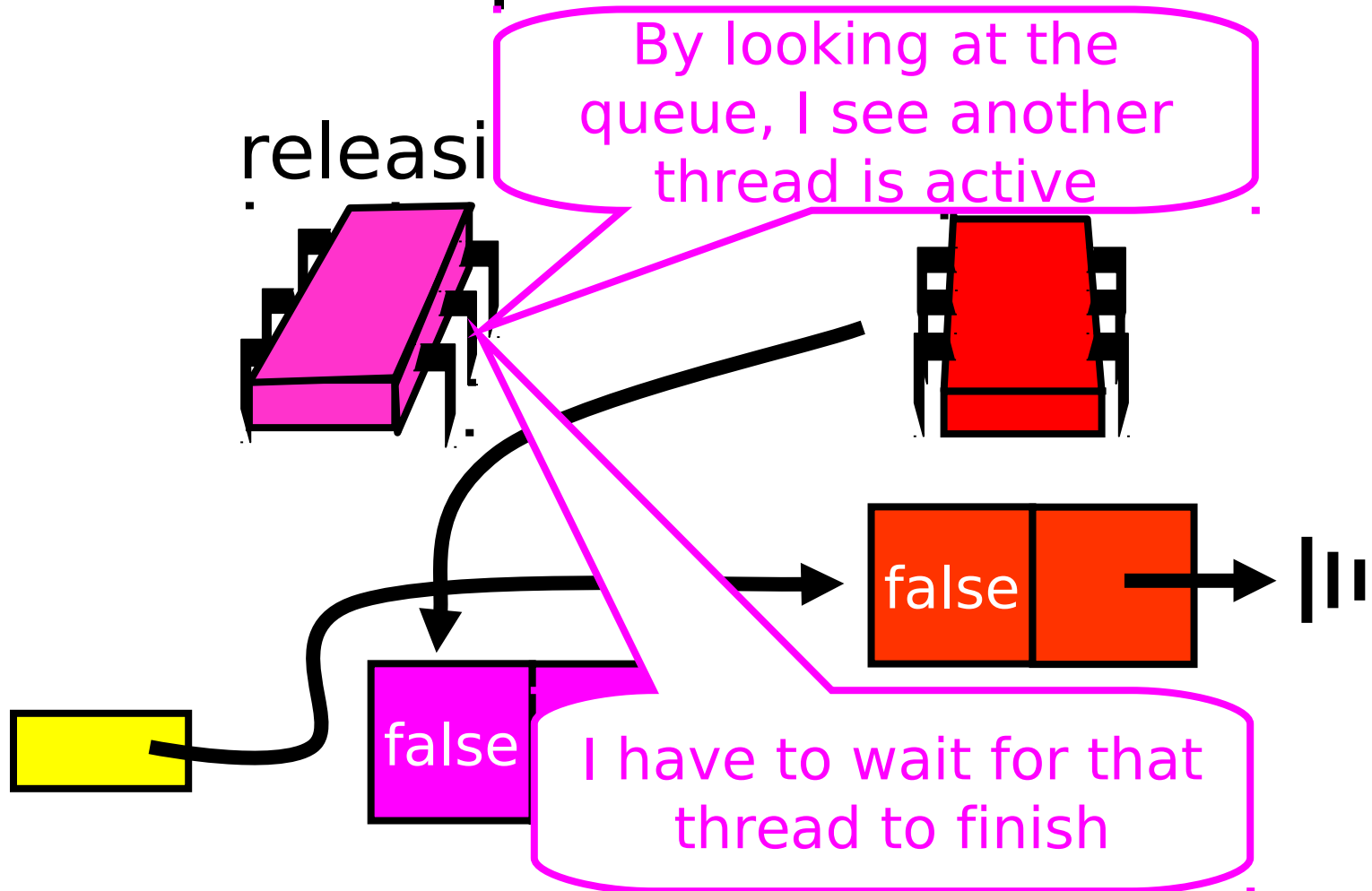
Purple Release



Purple Release

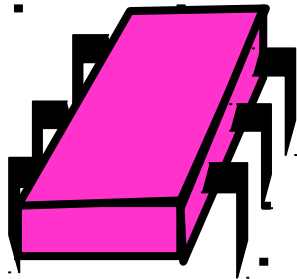


Purple Release

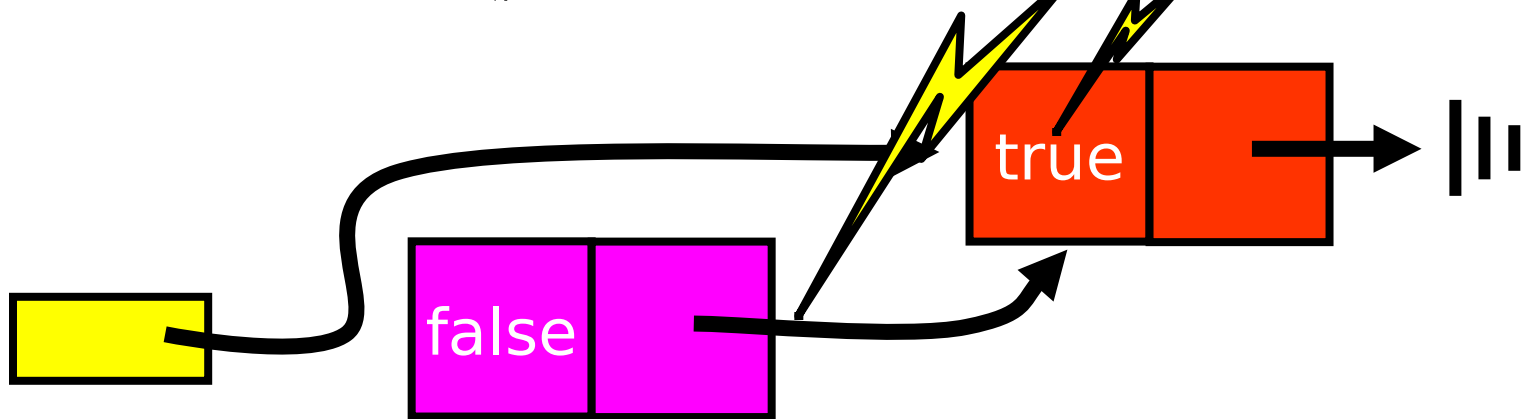
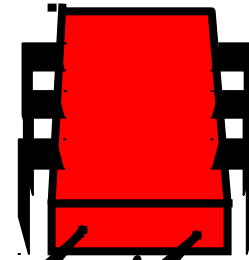


Purple Release

releasing

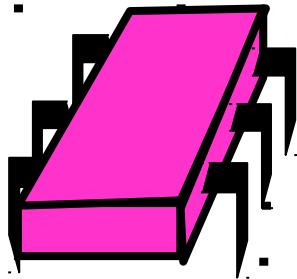


prepare to spin

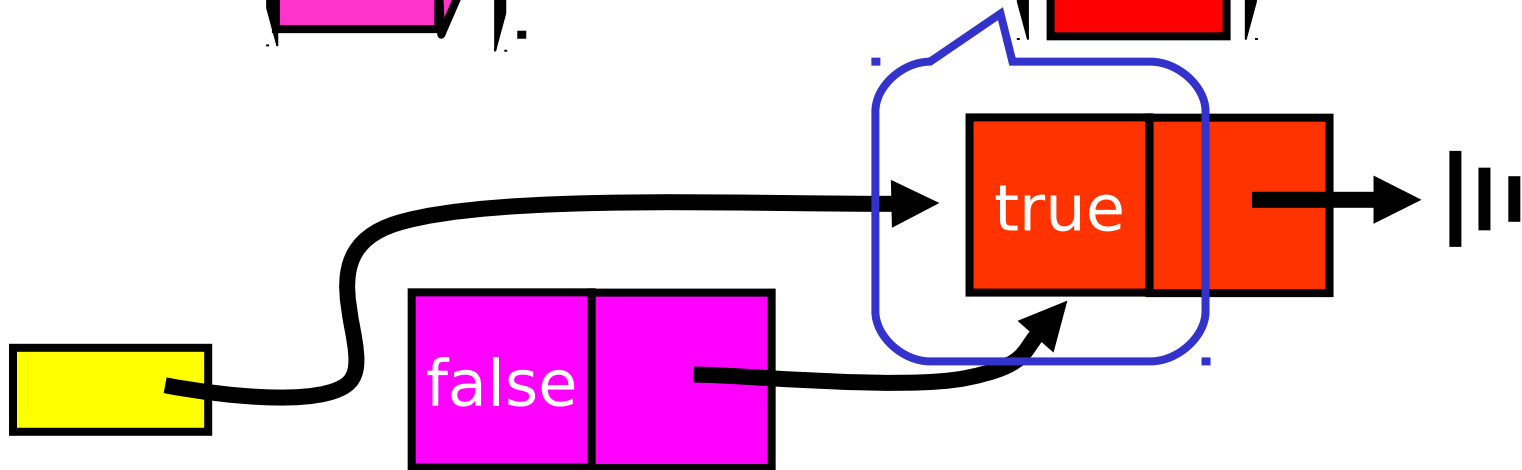
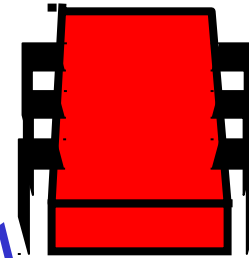


Purple Release

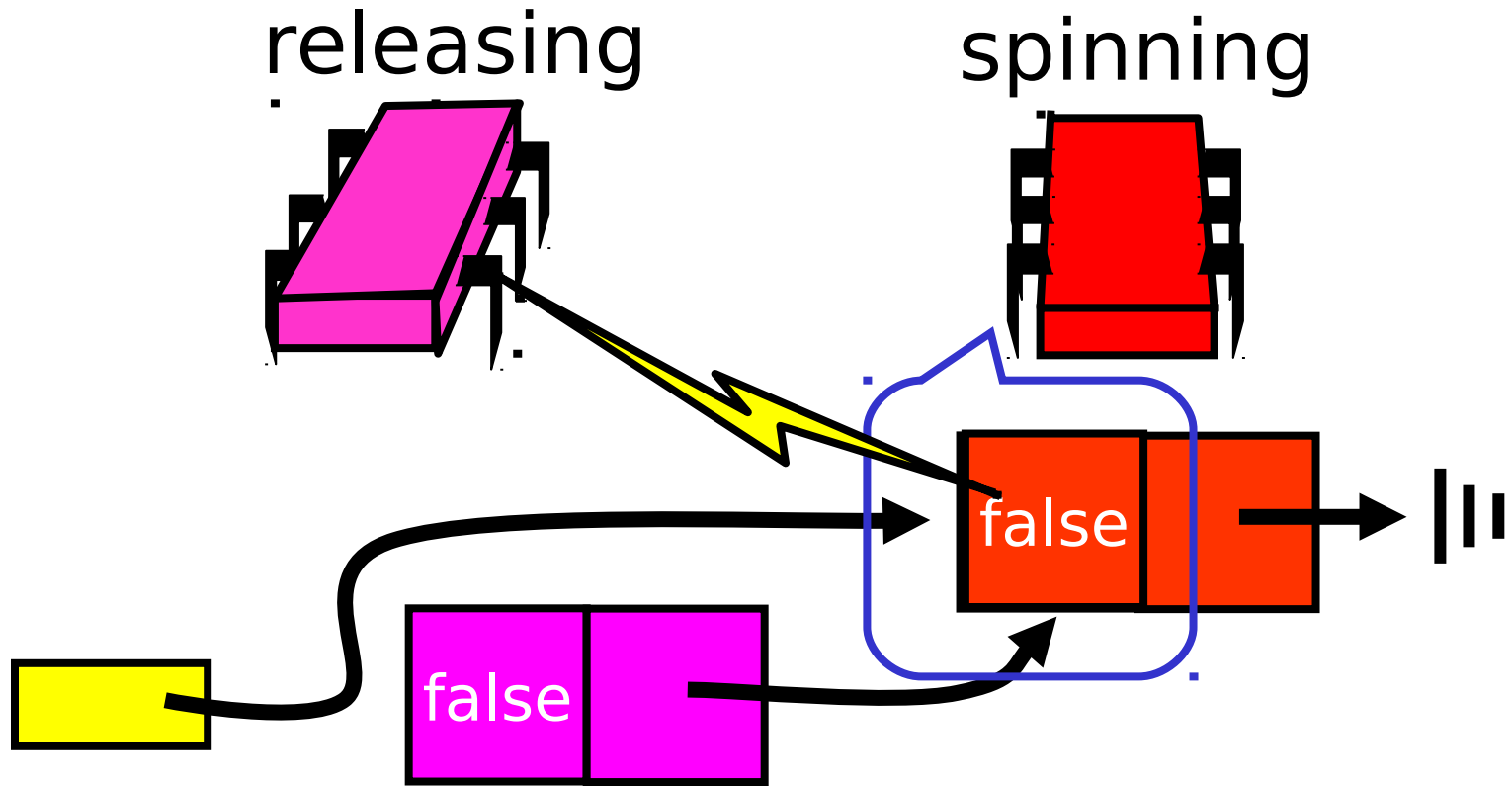
releasing



spinning

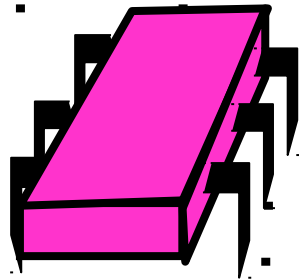


Purple Release

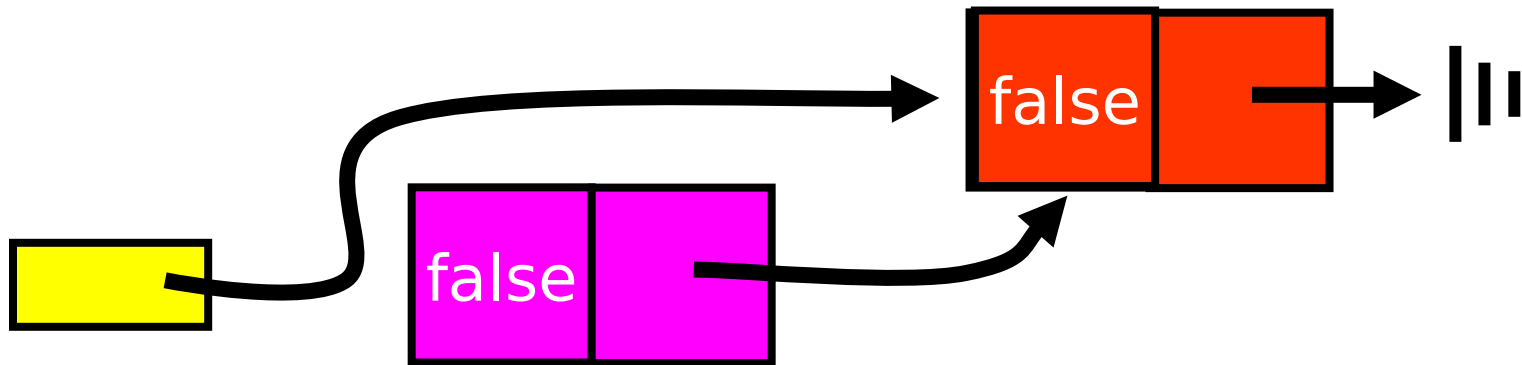
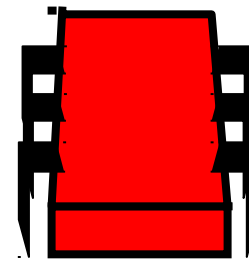


Purple Release

releasing



Acquired lock



Properties

- + Space: $O(L+N)$**
- + Local spinning (in the NUMA sense)**
- Spinning on unlock**
- needs more atomic operations (including CAS)**

Abortable Locks

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

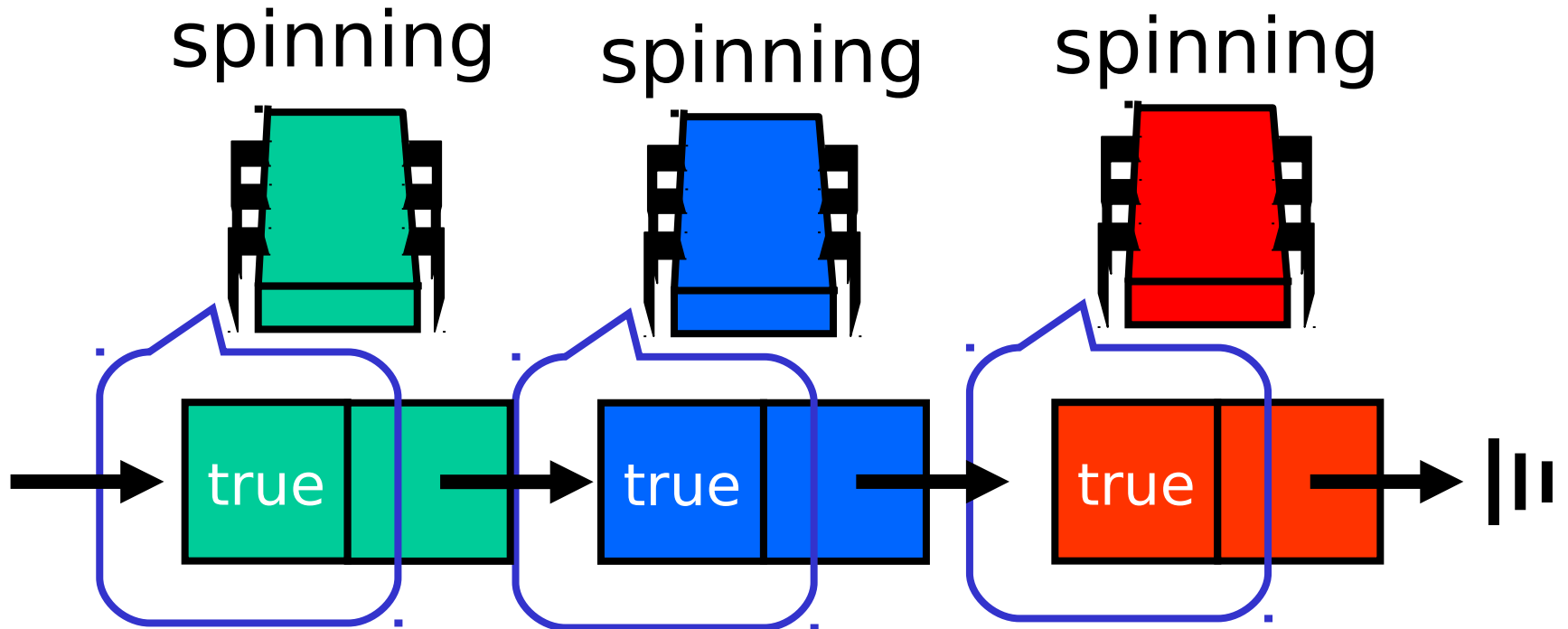
Back-off Lock

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

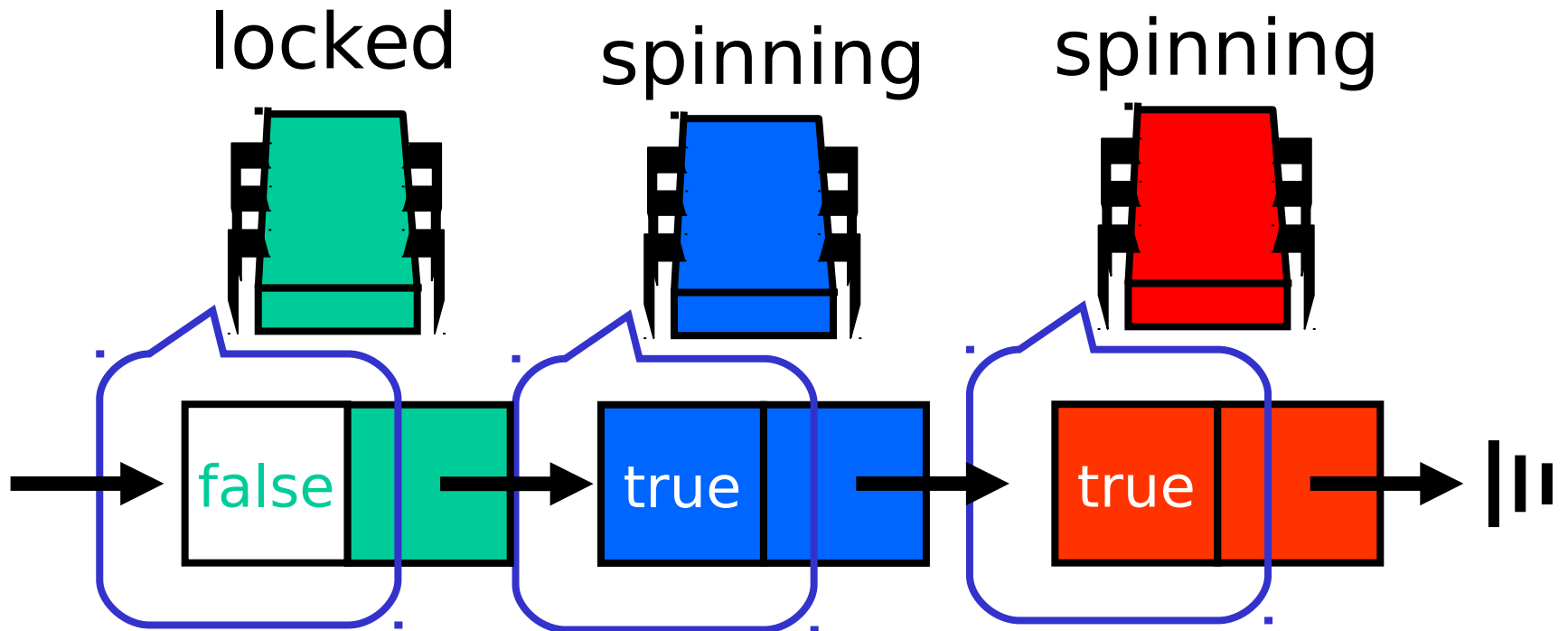
Queue Locks

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

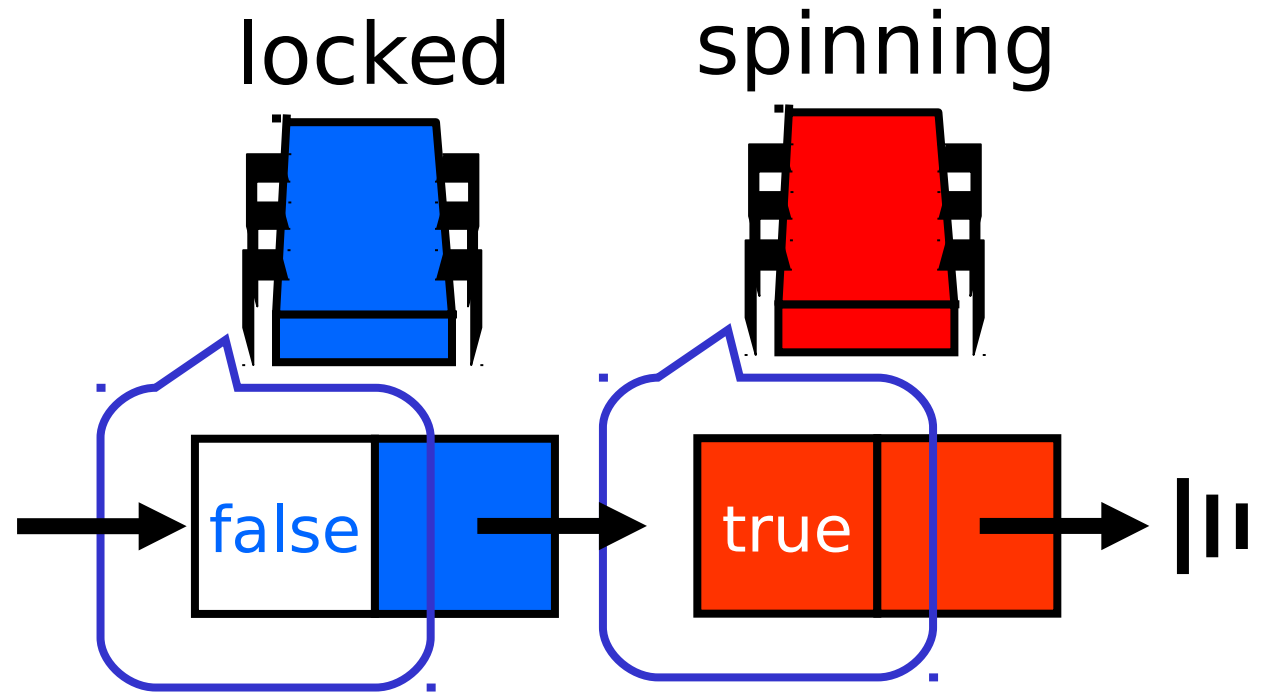
Queue Locks



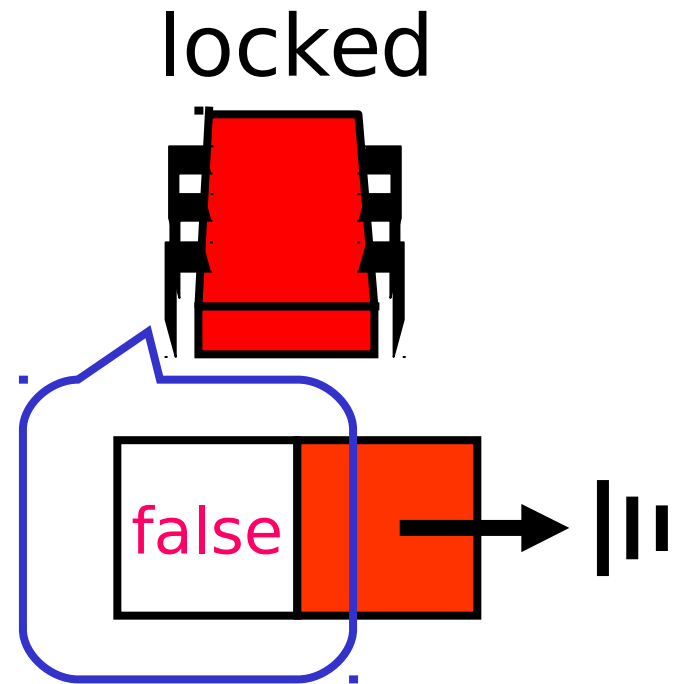
Queue Locks



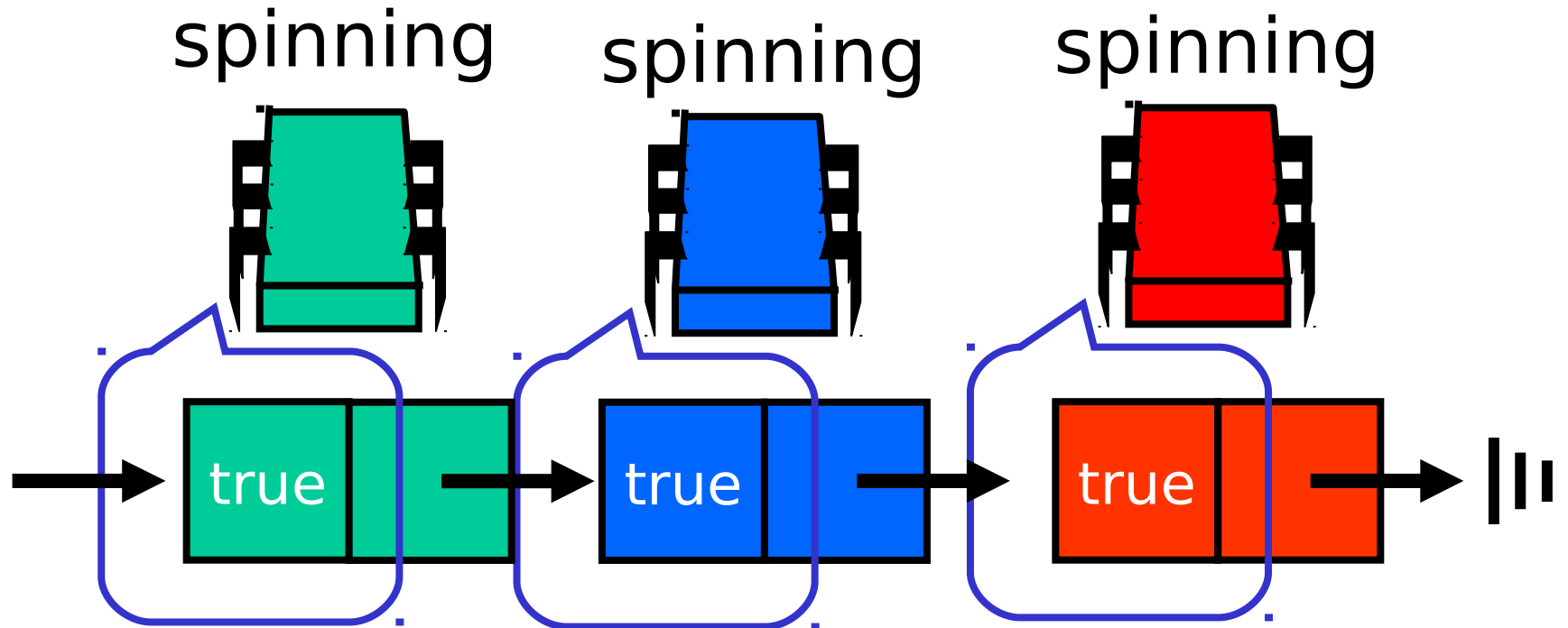
Queue Locks



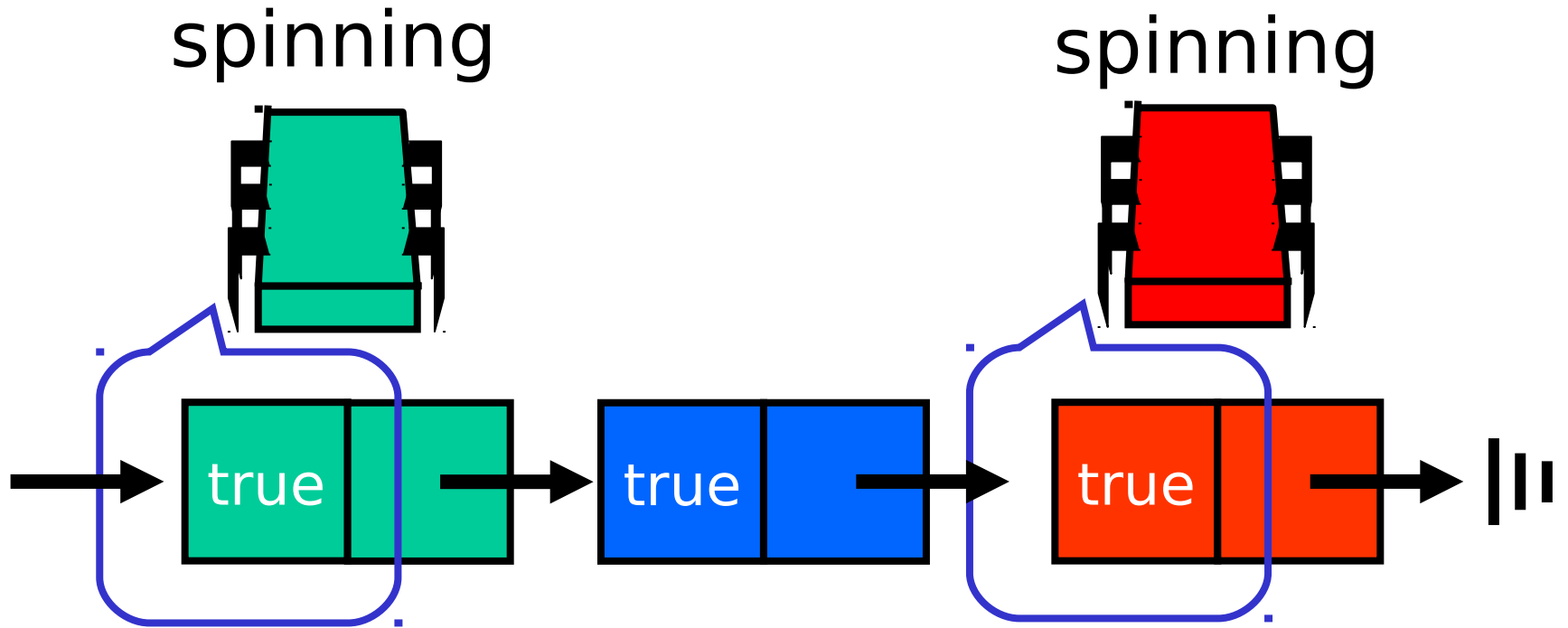
Queue Locks



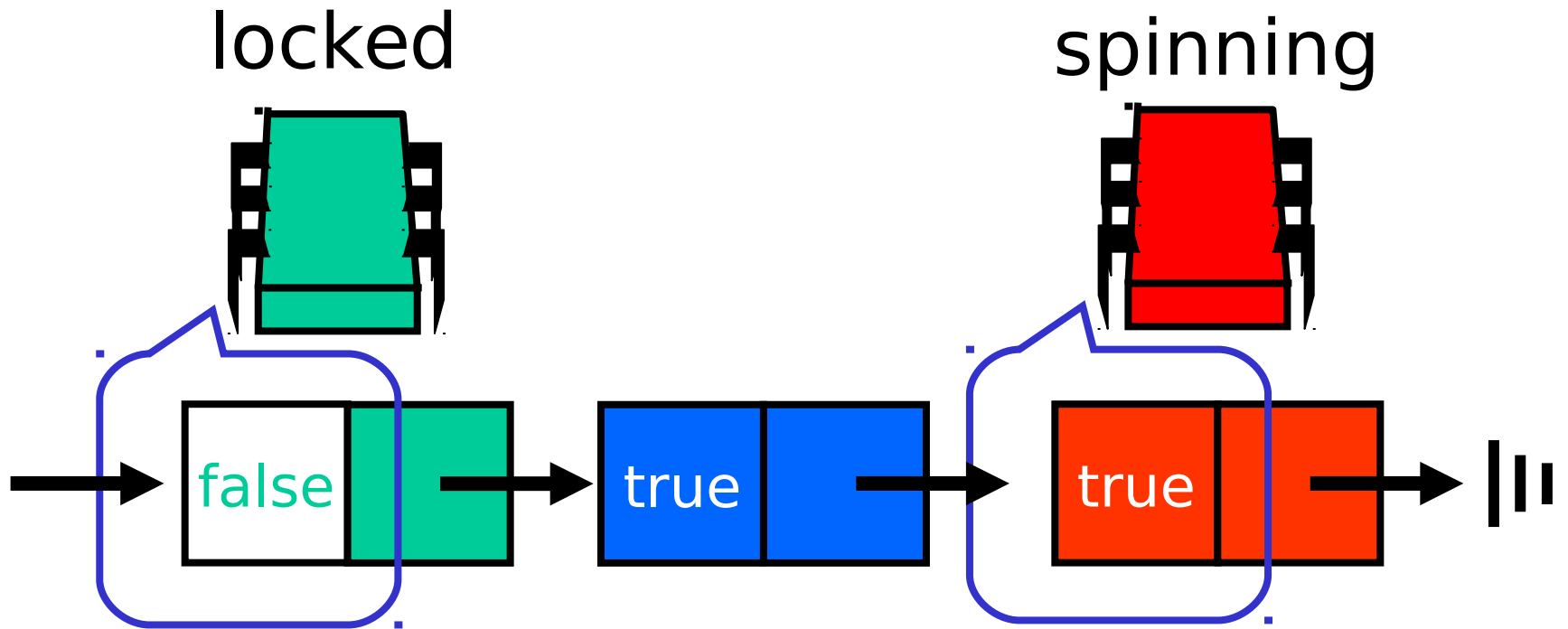
Queue Locks



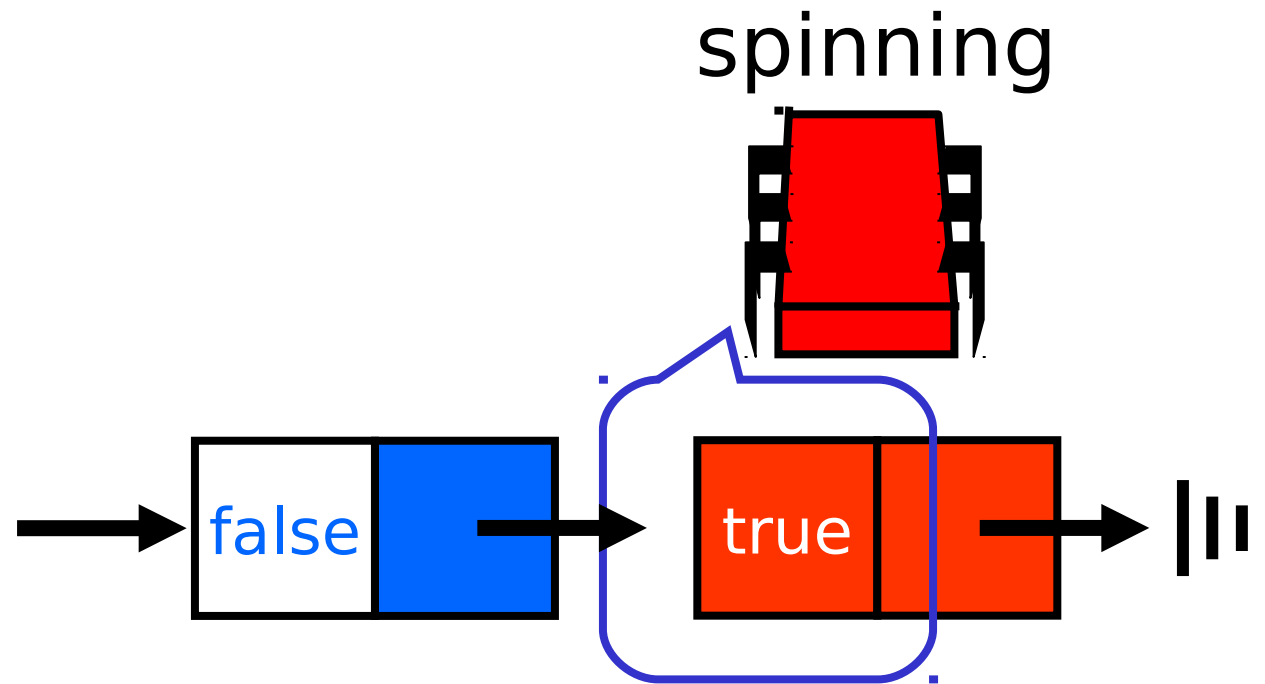
Queue Locks



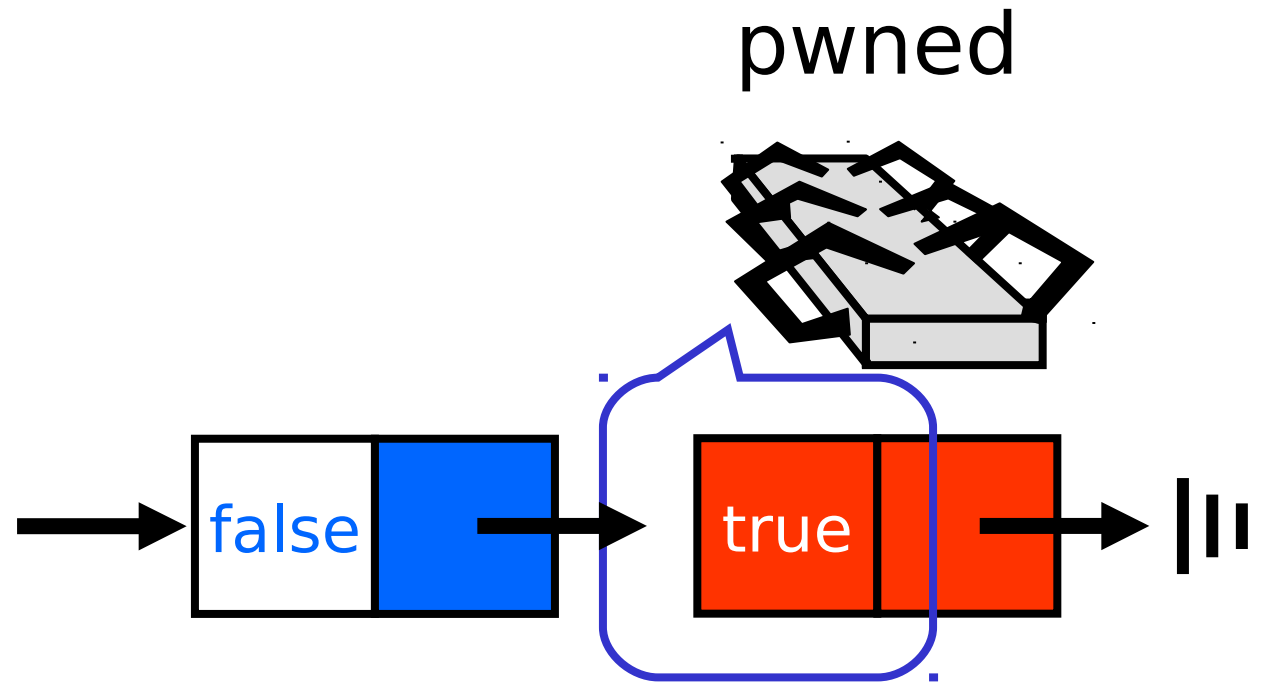
Queue Locks



Queue Locks



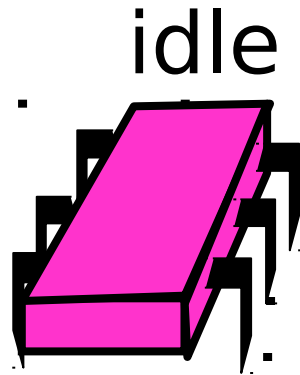
Queue Locks



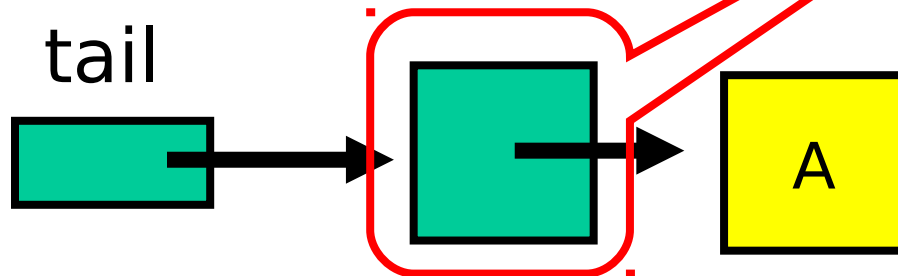
Abortable CLH Lock

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

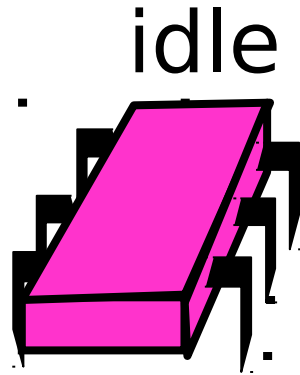
Initially



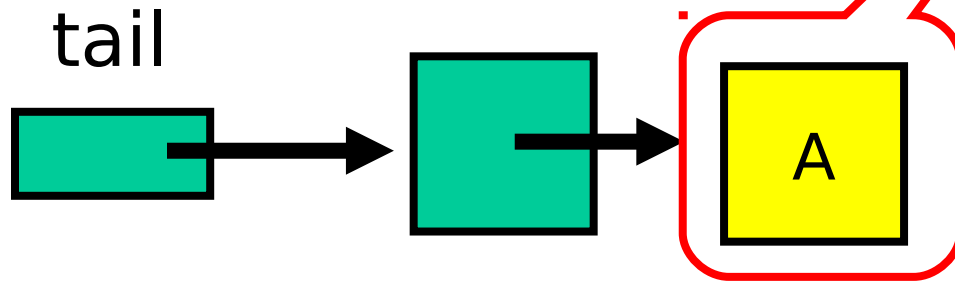
Pointer to
predecessor
(or null)



Initially

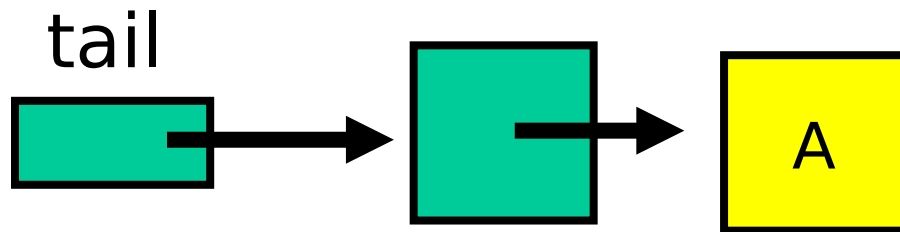
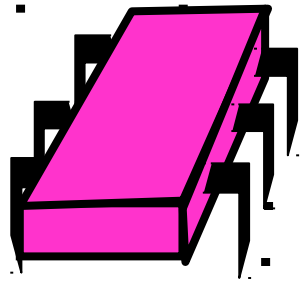


Distinguished
available
node means
lock is free



Acquiring

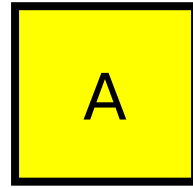
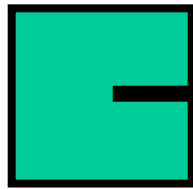
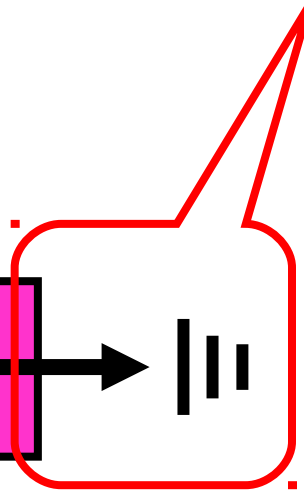
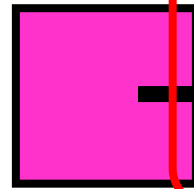
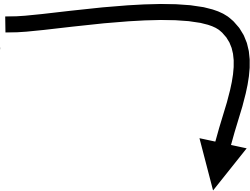
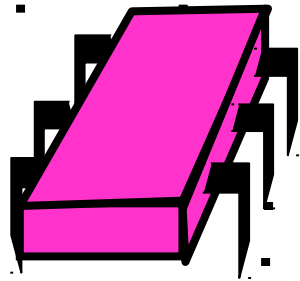
acquiring



Acquiring

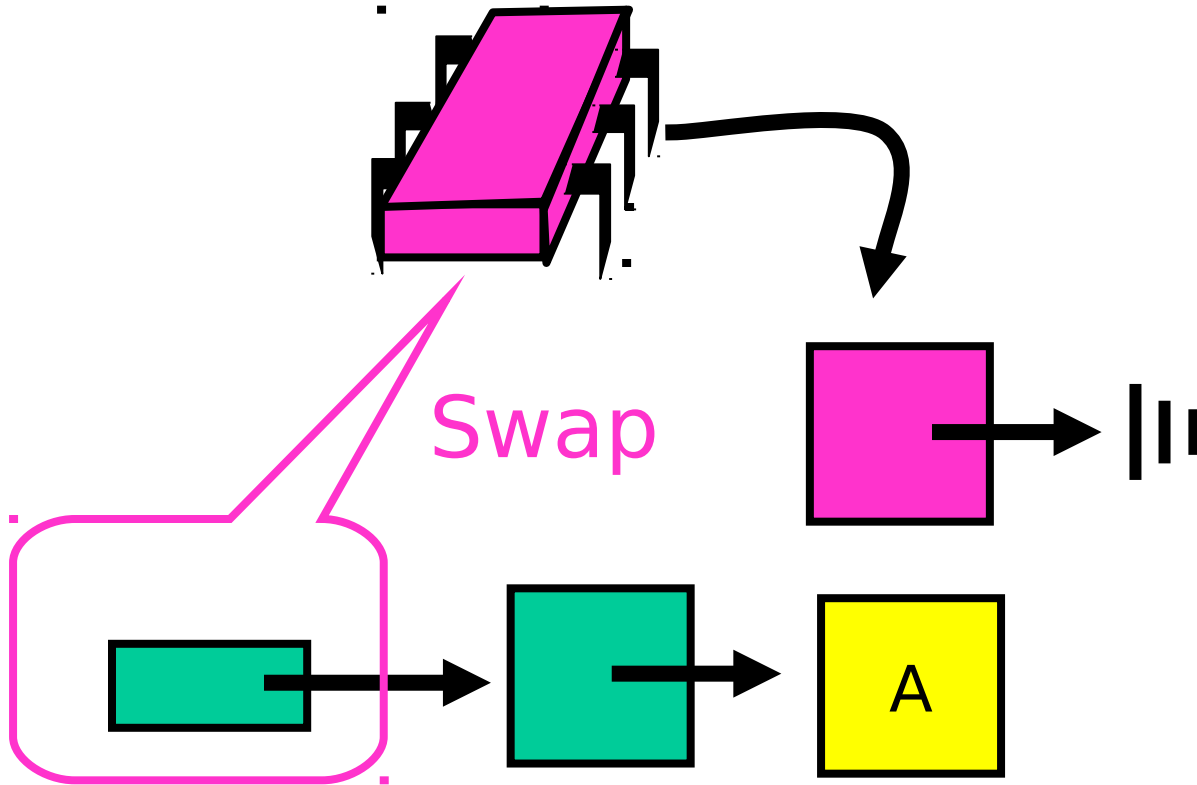
Null predecessor
means lock not
released or
aborted

acquiring



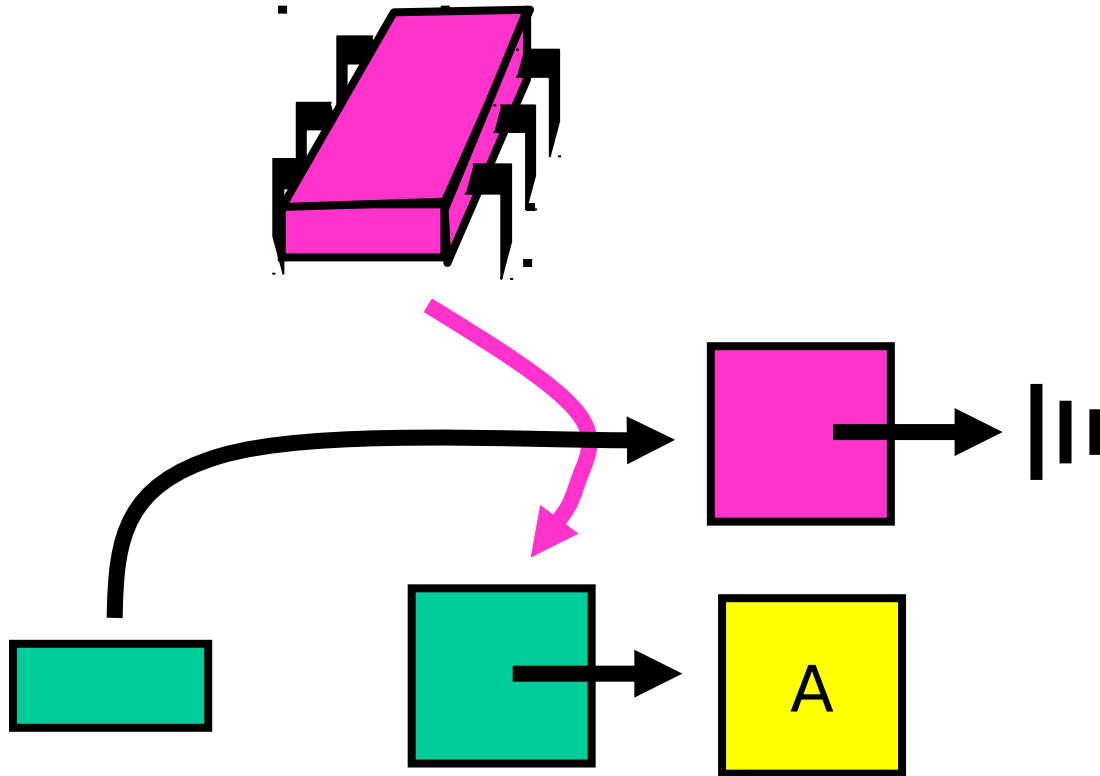
Acquiring

acquiring



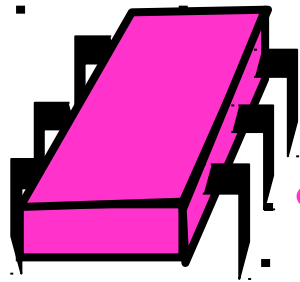
Acquiring

acquiring

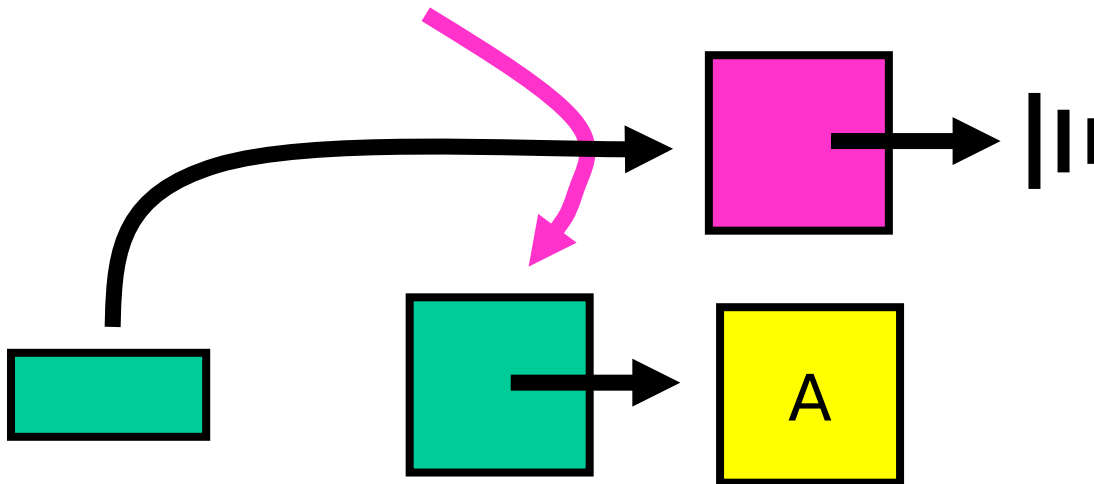


Acquired

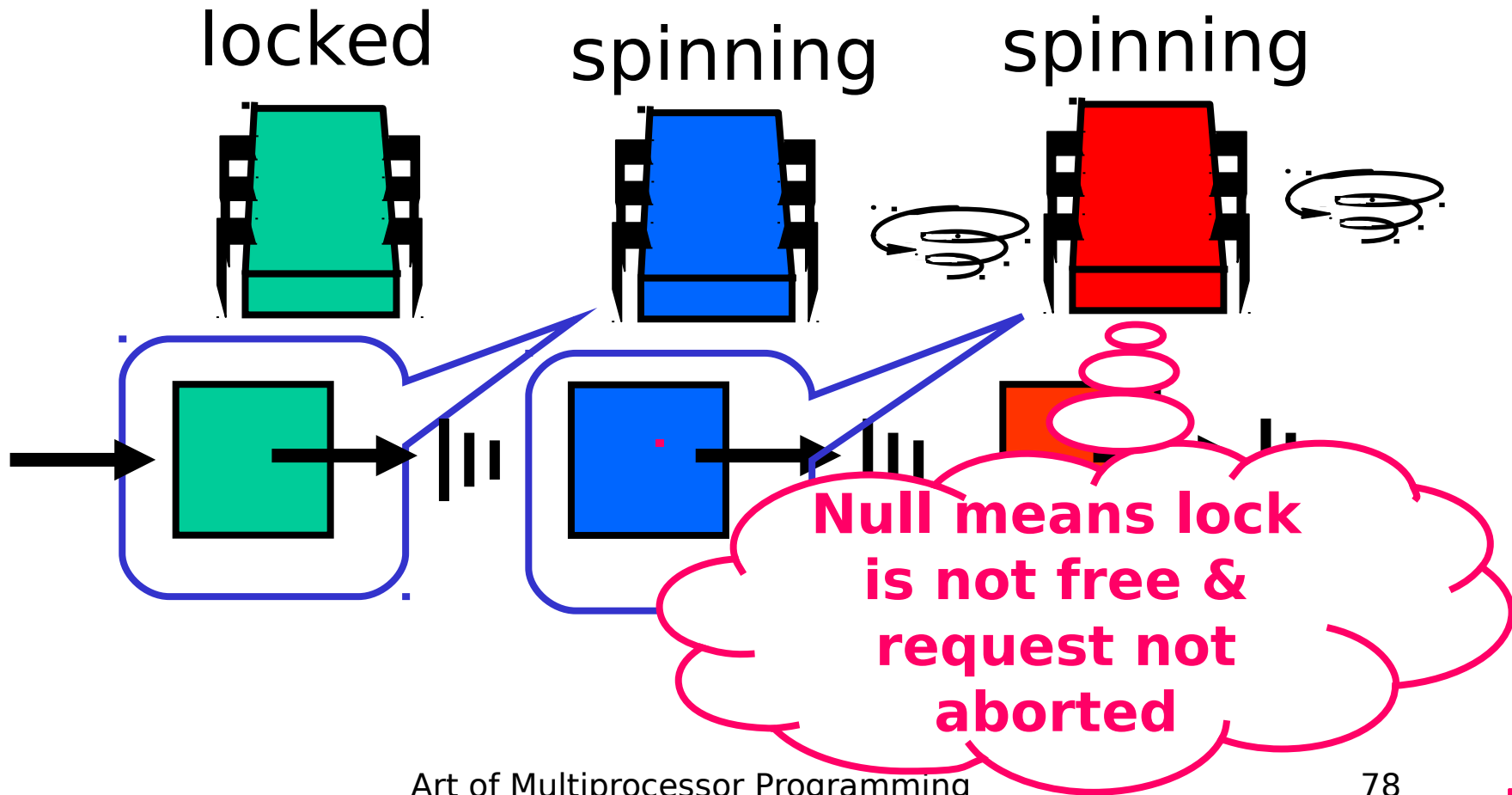
locked



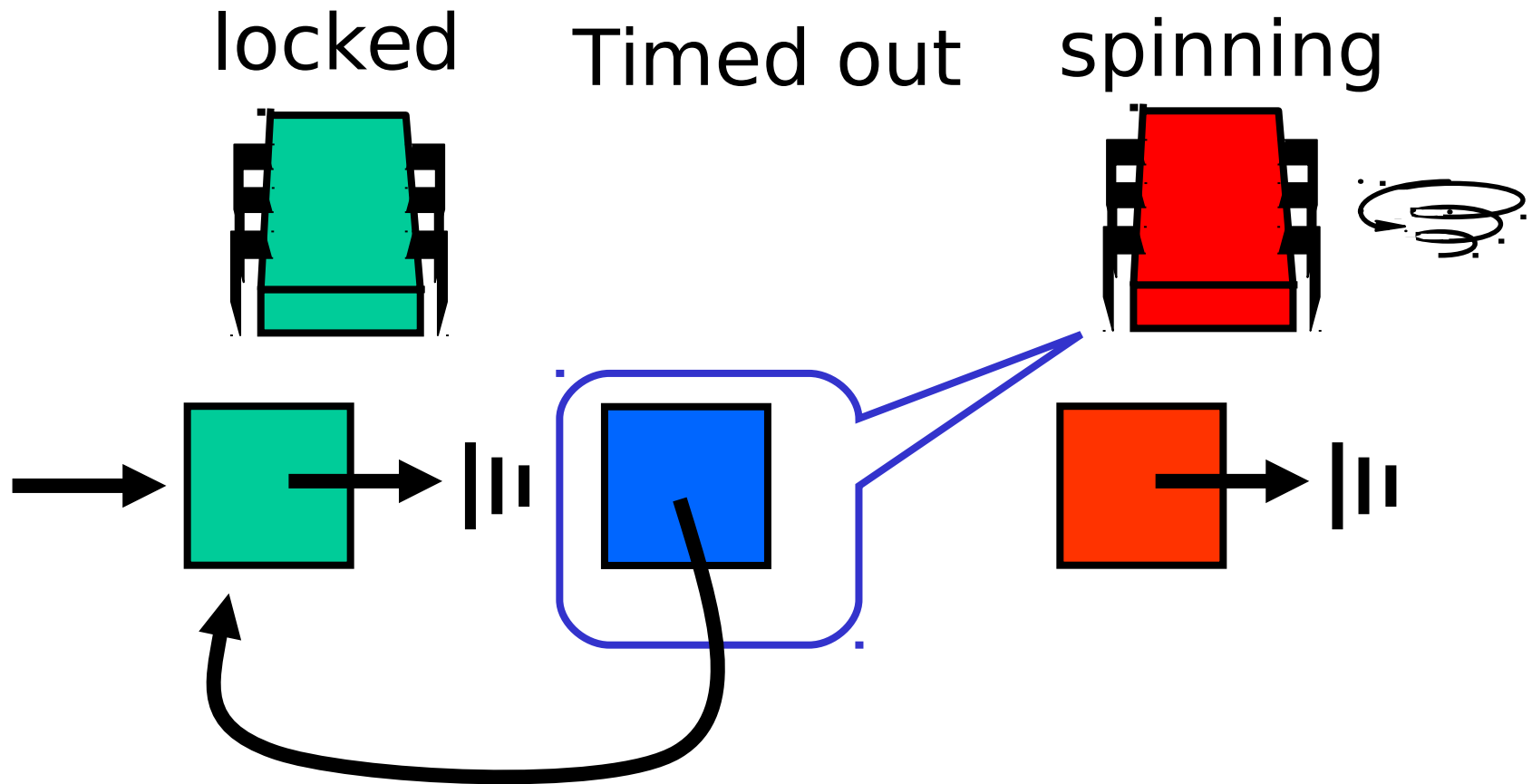
Pointer to
AVAILABLE
means lock is
free.



Normal Case

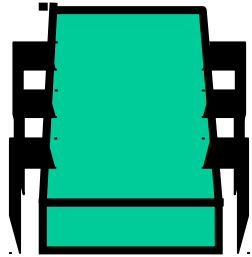


One Thread Aborts

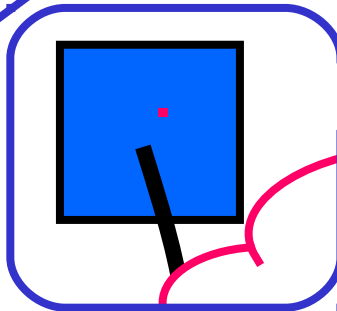


Successor Notices

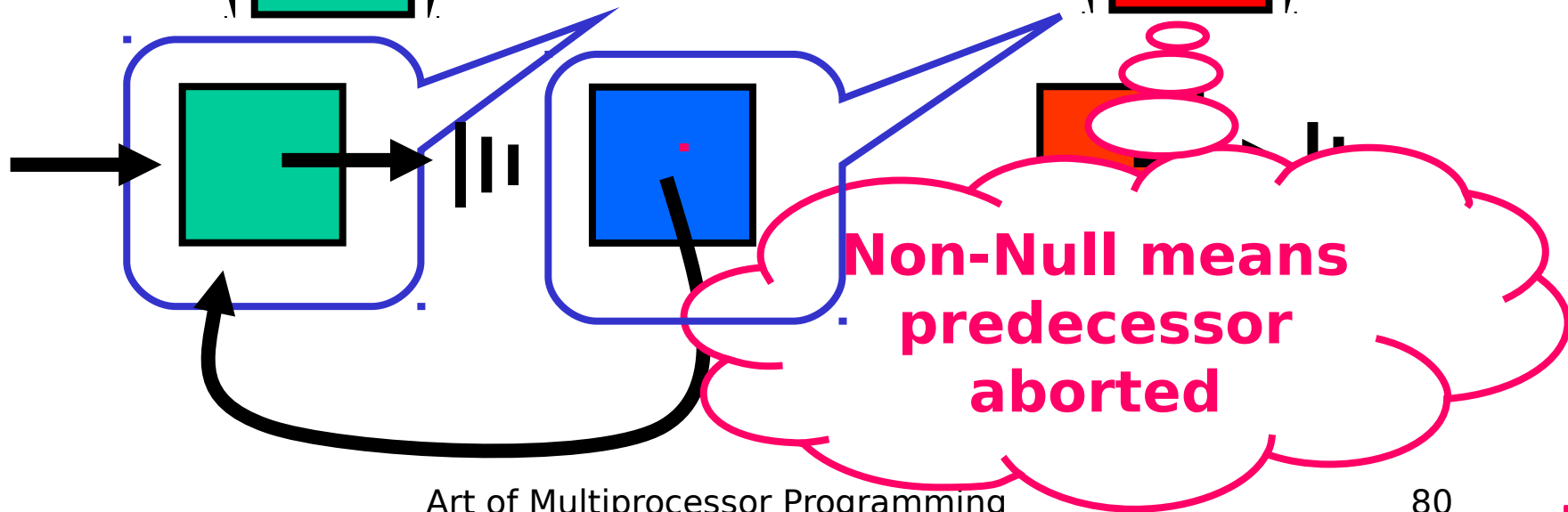
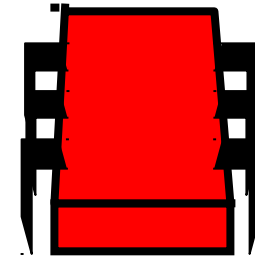
locked



Timed out

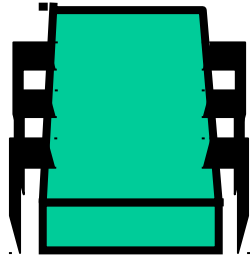


spinning

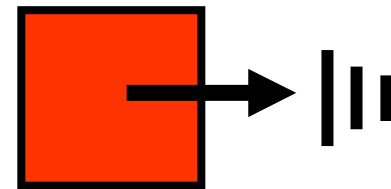
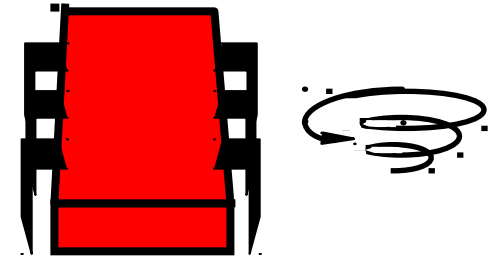


Recycle Predecessor's Node

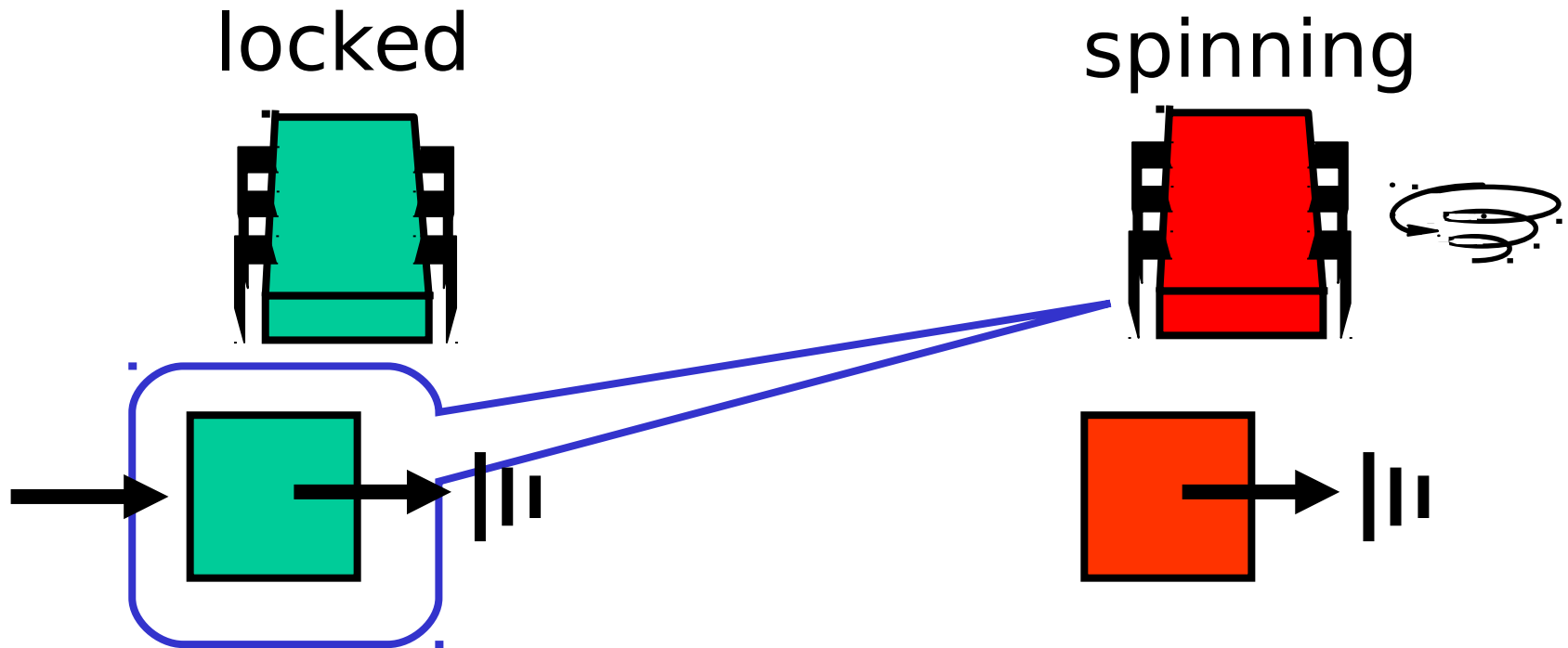
locked



spinning

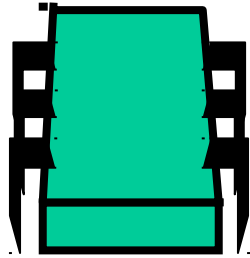


Spin on Earlier Node

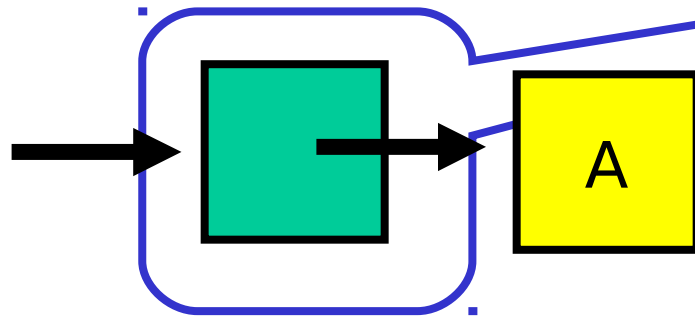
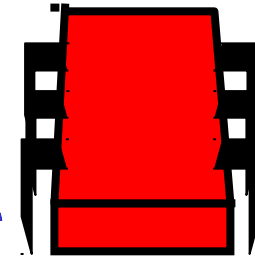


Spin on Earlier Node

released



spinning



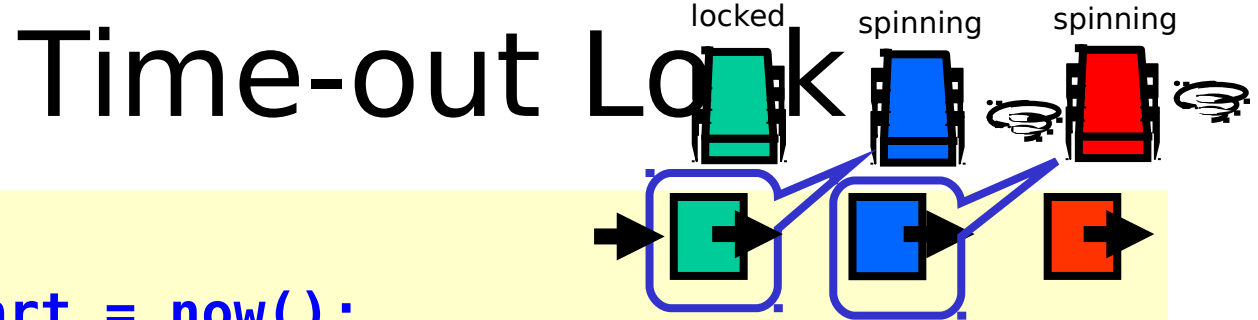
Time-out Lock

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

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;  
    }  
}
```

...



...

```
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

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

What do I do when I time out?

Time-Out Unlock

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


One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS, ToLock...
- Each better than others in some way
- There is no one solution
- Lock we pick really depends on:
 - the application
 - the hardware
 - which properties are important

This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **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.