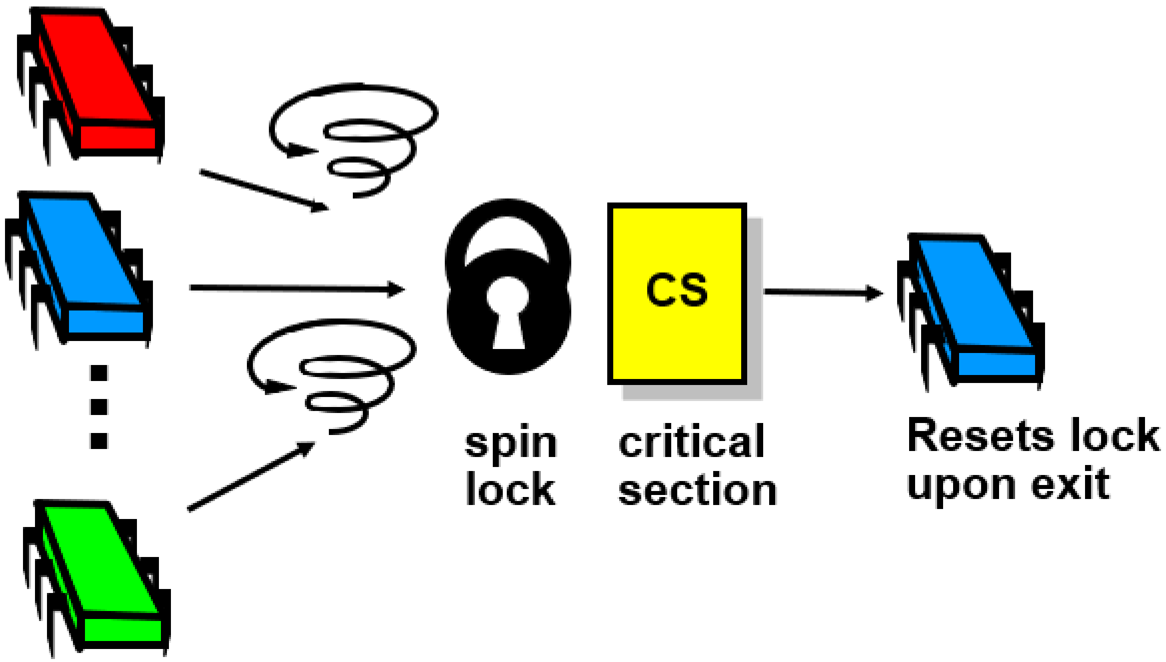
Spin Locks and Contention

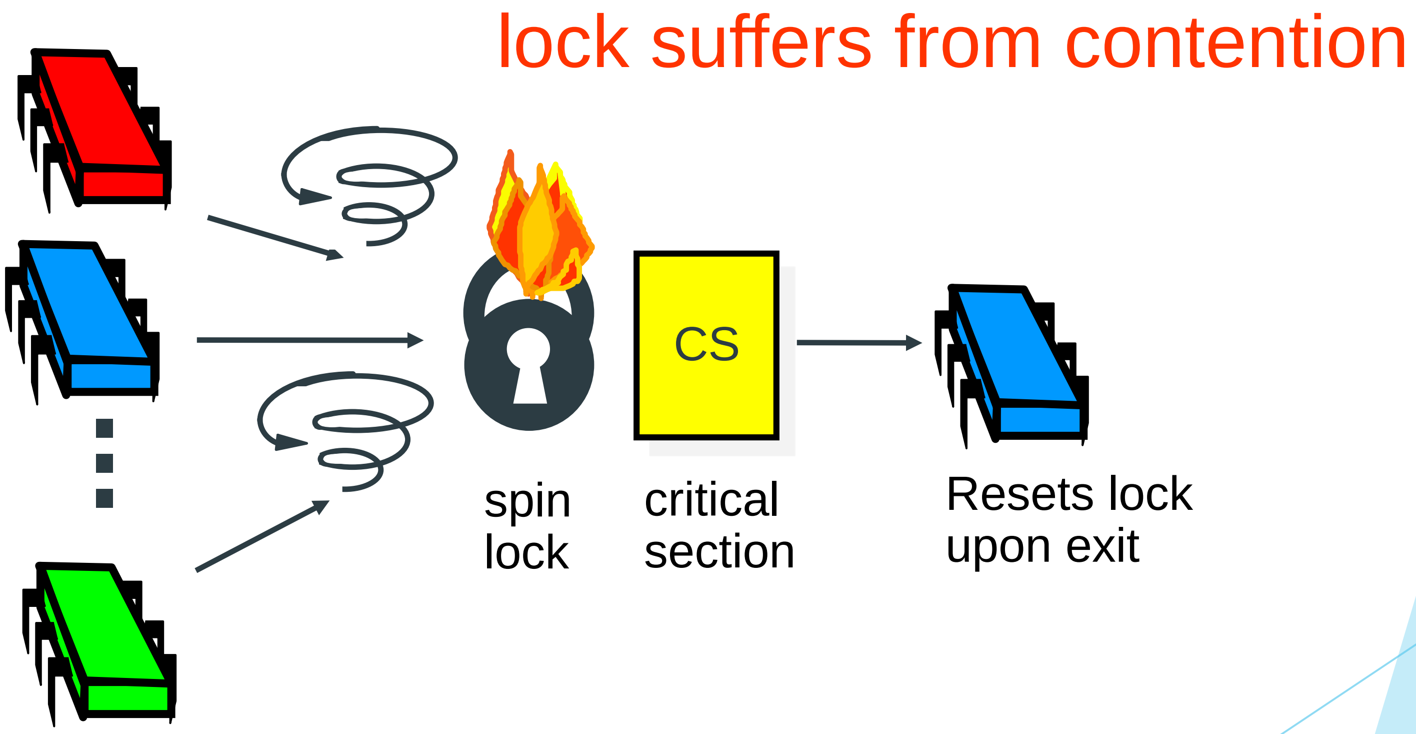
The goal

The goal of this lecture is to understand how architecture affects performance, and how to exploit this knowledge to write efficient concurrent programs.

Basic Spin-Lock



Basic Spin-Lock



Spin lock and blocking

- Spin lock- repeatedly testing the lock. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting.

- blocking- suspend yourself and ask the operating system's scheduler to schedule another thread on your processor. blocking makes sense if you expect the lock delay to be long.

Mutex lock-code

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

Peterson algorithm

Lock()

flag[i]=true

turn=1-i

While (flag[1-i] and turn=1-i)

Unlock()

flag[i]=false

Not providing mutual exclusion in practice

- Running the Peterson algorithm in practice will provide a slightly off value then expected.
- It must be that both threads are occasionally in the critical section at the same time, even though we have proved that this cannot happen.

"How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?"

Why?

- compilers that reorder instructions to enhance performance - It is therefore possible that the order of writes of flag[*i*] and turn by thread *i* will be reversed by the compiler.
- writes to multiprocessor memory do not necessarily take effect when they are issued – it is therefore possible that the order of *i*'s write to turn is delayed and may arrive in memory only after *i* reads flag[*j*].

GetAndSet()

- Atomic instruction.
- getAndSet(*b*) that atomically replaces the current value with *b*, and returns the previous value.

- We will now look at two algorithm that uses this method.

Solution- Test-And-Set Locks

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

TASLock algorithm

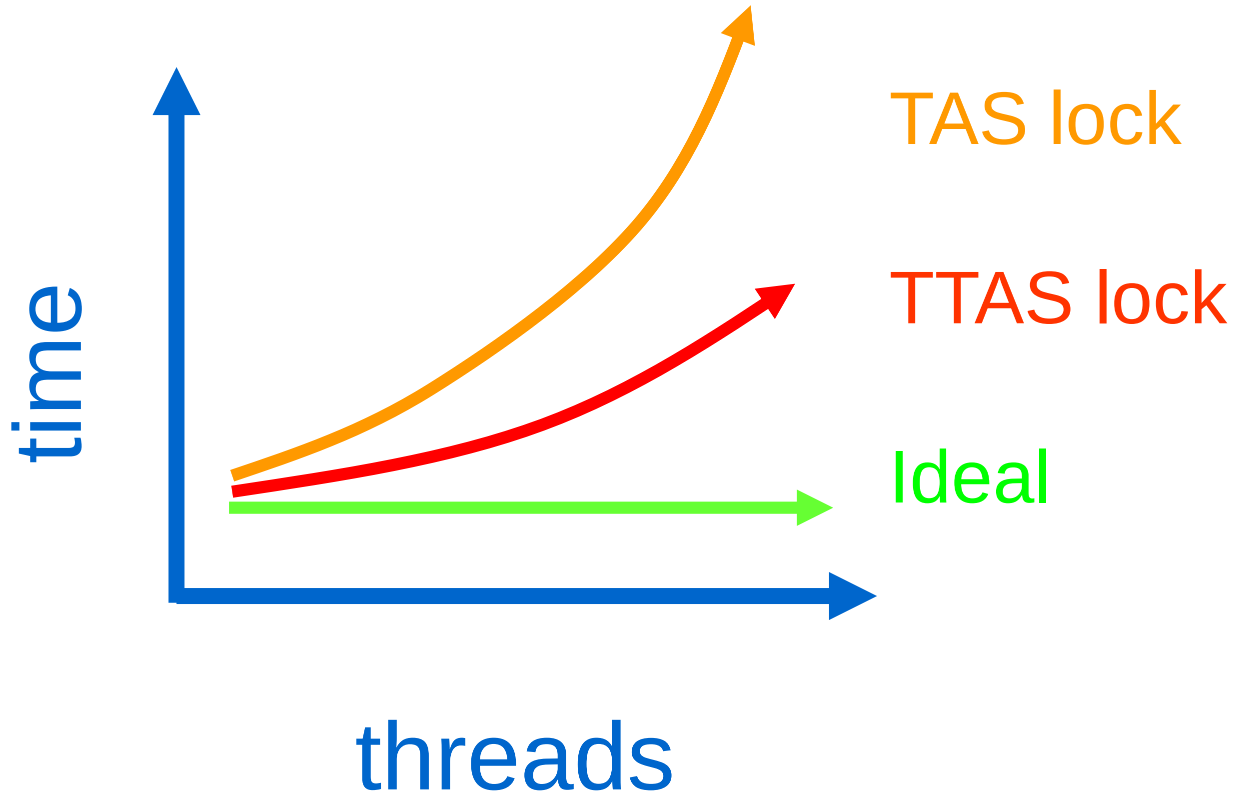
TTASLock lock algorithm

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

TTASLock un-lock algorithm

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

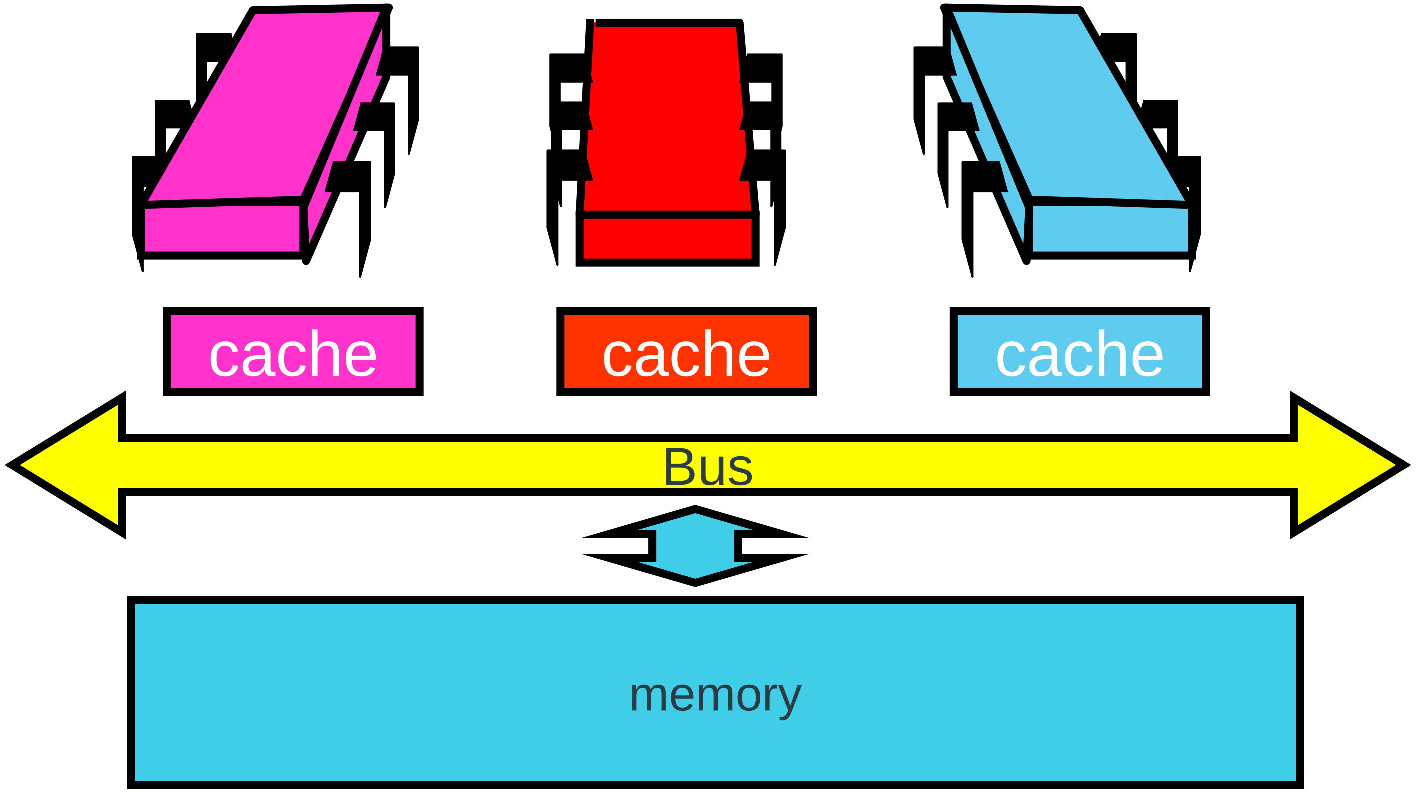
Schematic performance of a TASLock, a TTASLock, and an ideal lock



TASLock vs. TTASLock

- Both algorithms are the same.
- Yet they have different performant- TTASLock perform better then TASLock.
- Both algorithms preform poorly vs. the ideal performance.

Bus architecture



TASLock vs. TTASLock

TASLock:

- Each test-and-set call go over the bus.
- Thread that want to release the lock will be delayed.
- Threads not waiting to lock can be delayed when access. memory

TTASLock:

- Set calls not use bus if value in cache.
- Thread that want to release the lock will not be delayed.

TTASLock is still far from ideal because when the lock is free all threads try test-and-set call

Solution- Backoff

- When lock is free ⟶ tries to lock it
  - If succeed great
  - Else do backoff

- We will like that the larger the unsuccessful tries the longer the backoff

Backoff

```
public void backoff() throws
  InterruptedException {
  int delay = random.nextInt(limit);
  limit = Math.min(maxDelay, 2 * limit);
  Thread.sleep(delay);
}
```

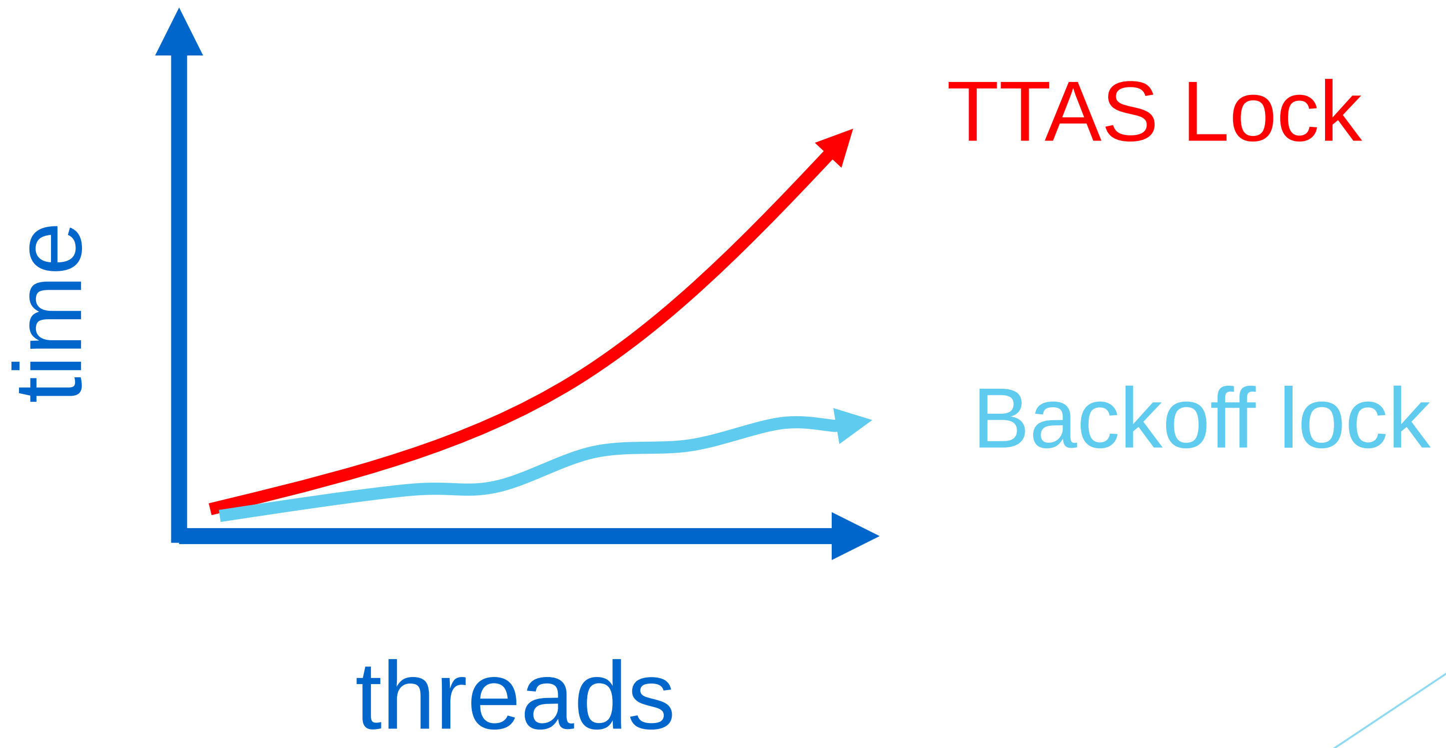
Backoff lock

```
public class BackoffLock implements Lock {
private AtomicBoolean state = new  AtomicBoolean(false);
public void lock() {
Backoff backoff = new Backoff(MIN_DELAY, MAX_DELAY);
while (true) {
    while (state.get()) {};
        if (!state.getAndSet(true)) {
            return;
        } else {
            backoff.backoff();
        }}}
```

Backoff un-lock

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

Performance graph



Backoff algorithm

Pros:

- Better then TTASLock in contention (less contention).
- Easy to implement.

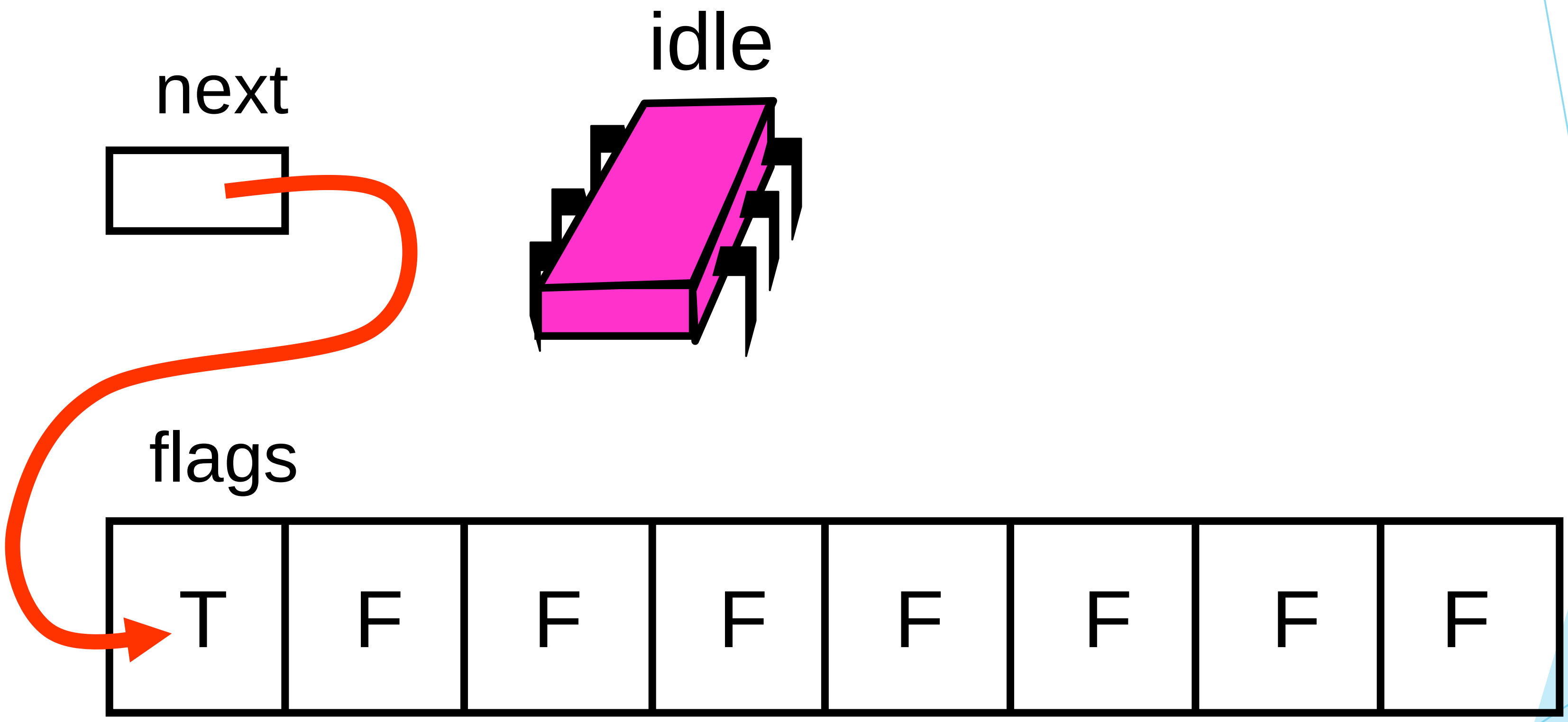
Cons:

- All threads spin on the same shared location causing cache-coherence traffic on every successful lock access.
- Threads delay longer than necessary, causing the critical section to be underutilized.
- Parameters need to be choose carefully.

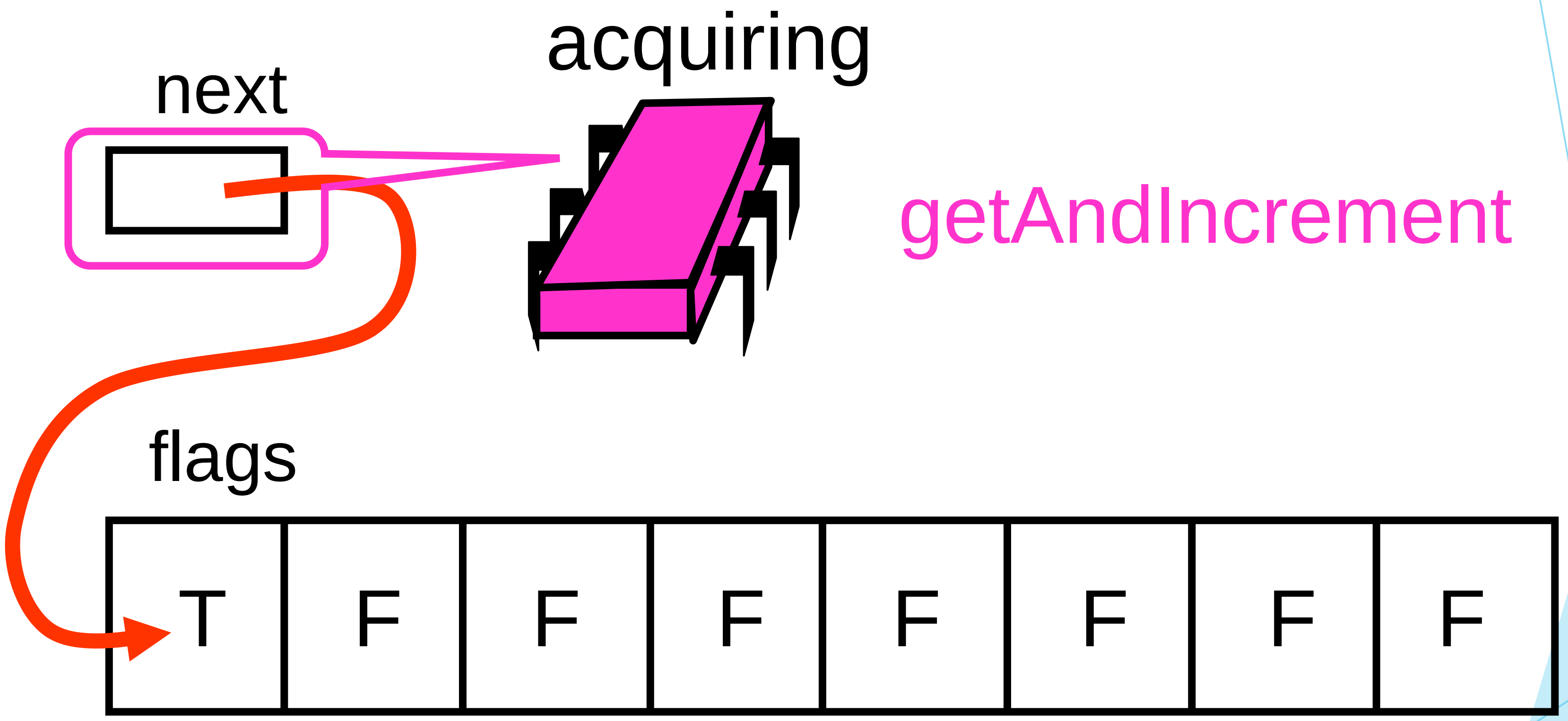
Solution- queue lock

- each thread spin on a different cell in the queue.
- Thread is notified when it is it’s time to go to the critical section through the queue.

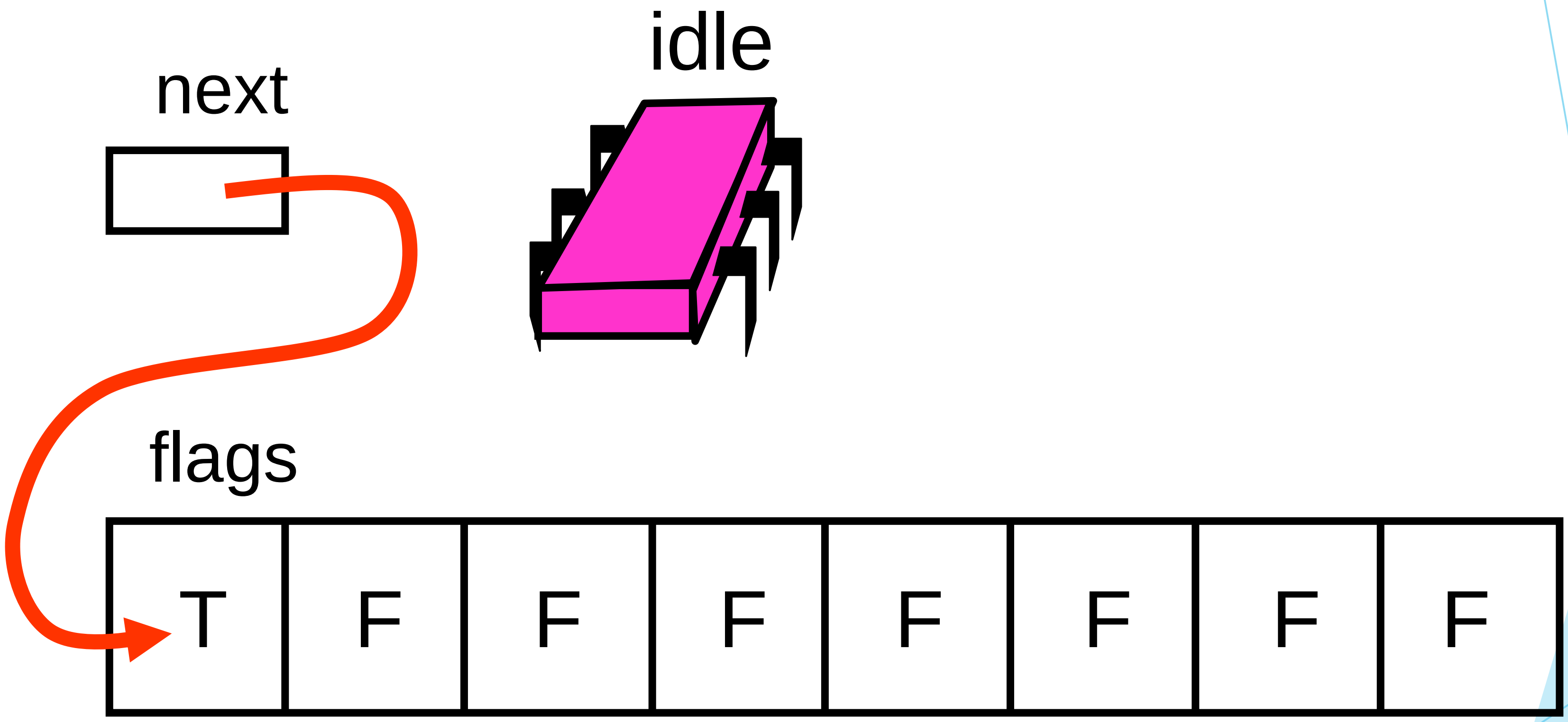
Queue Lock



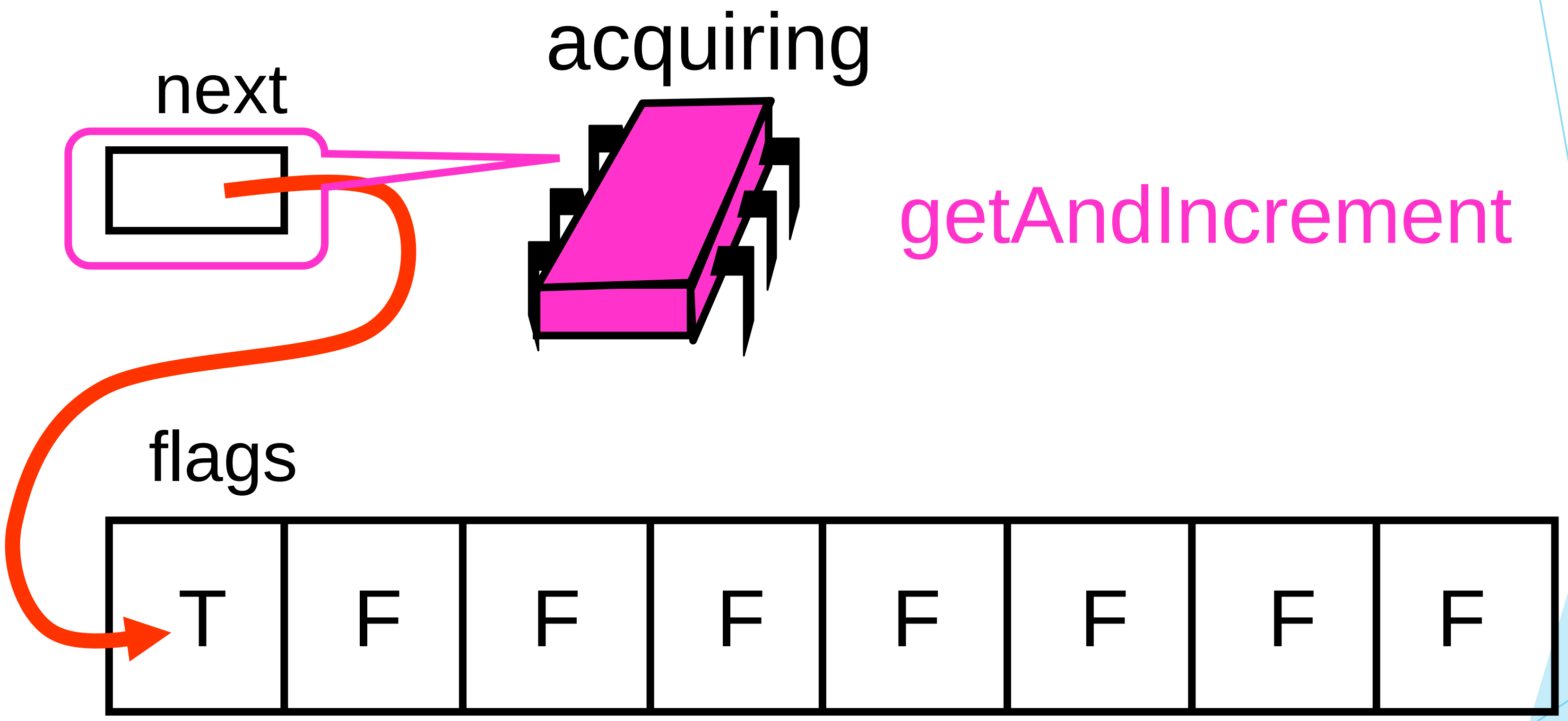
Queue Lock



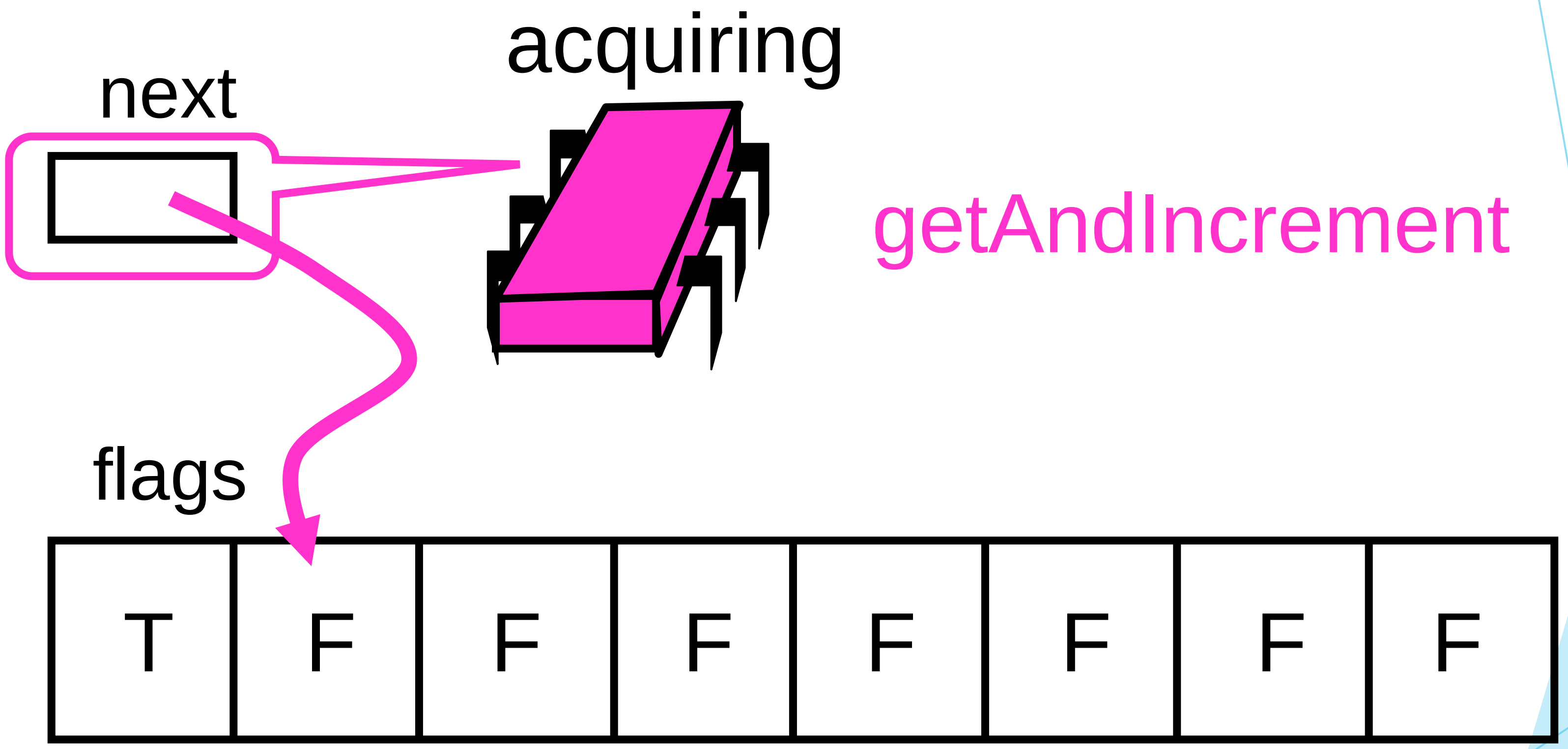
Queue Lock



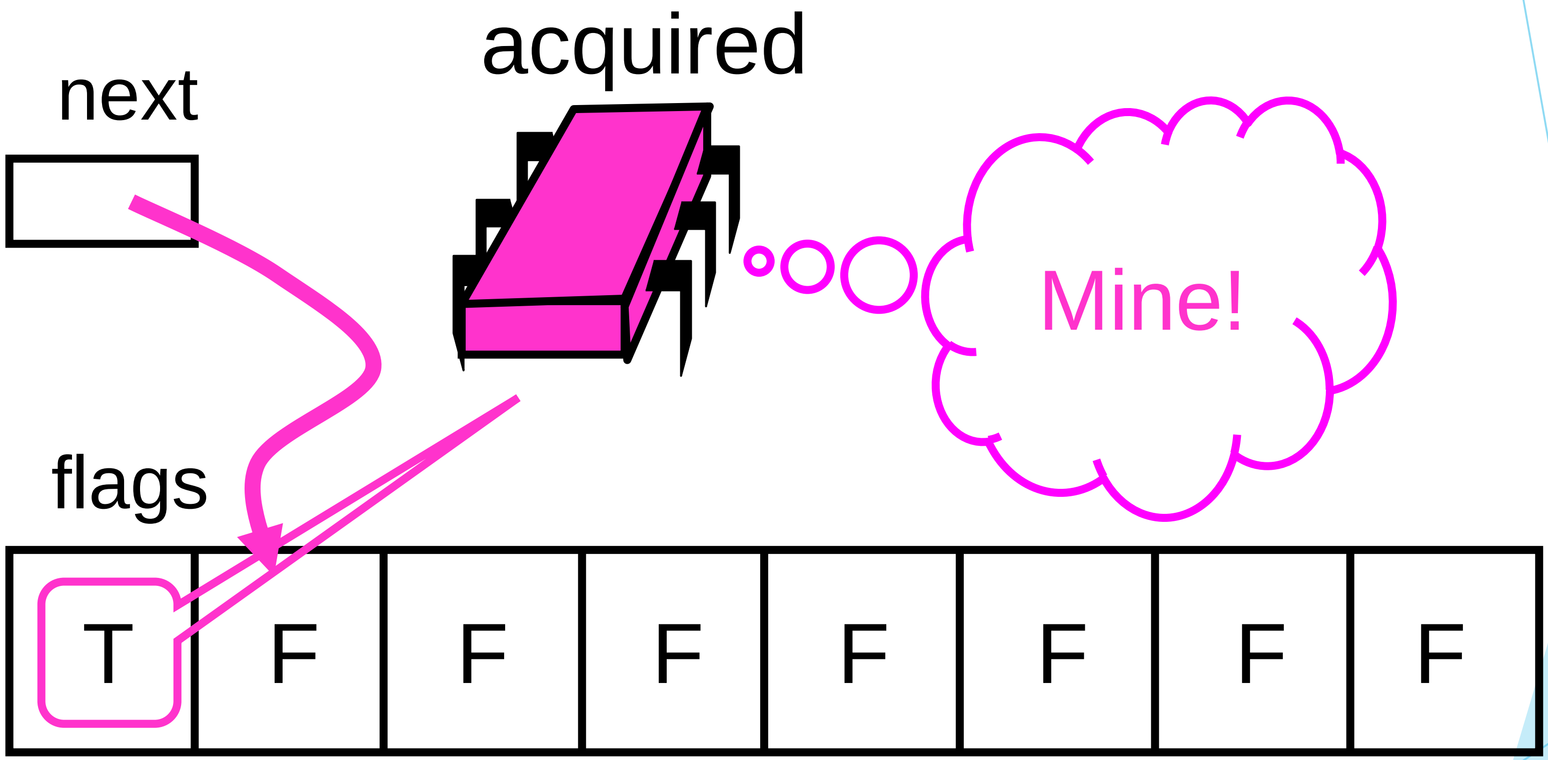
Queue Lock



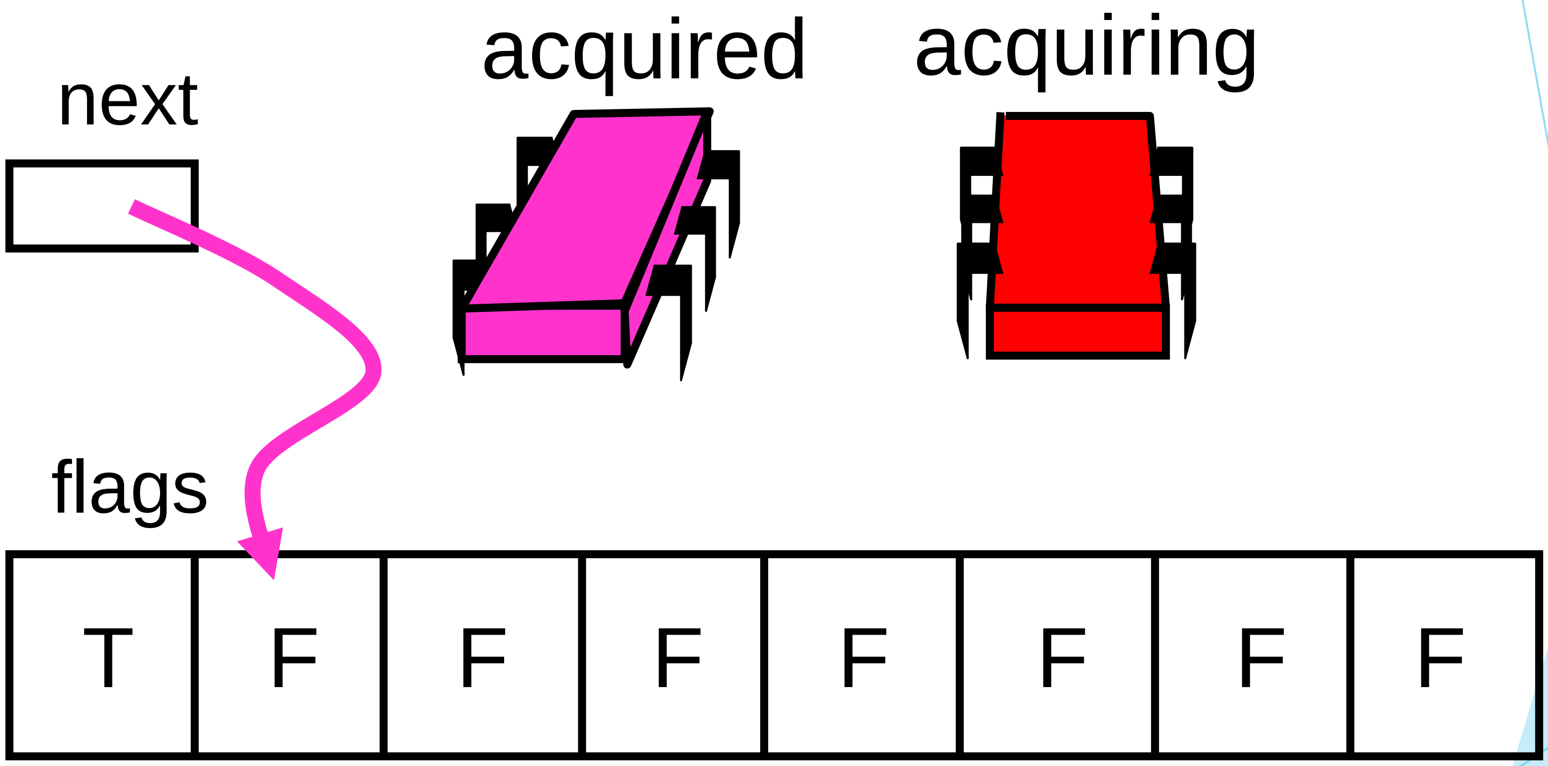
Queue Lock



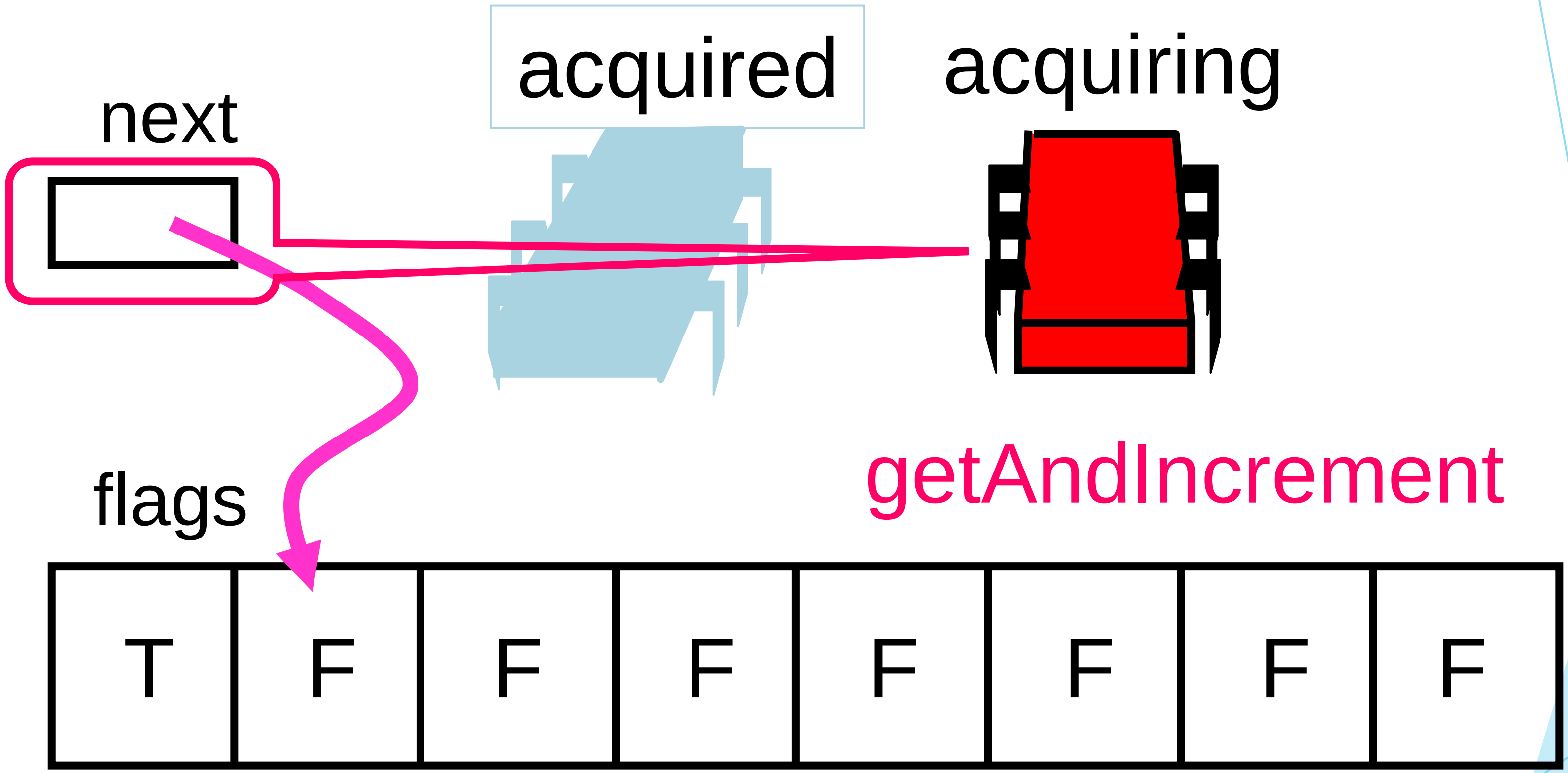
Queue Lock



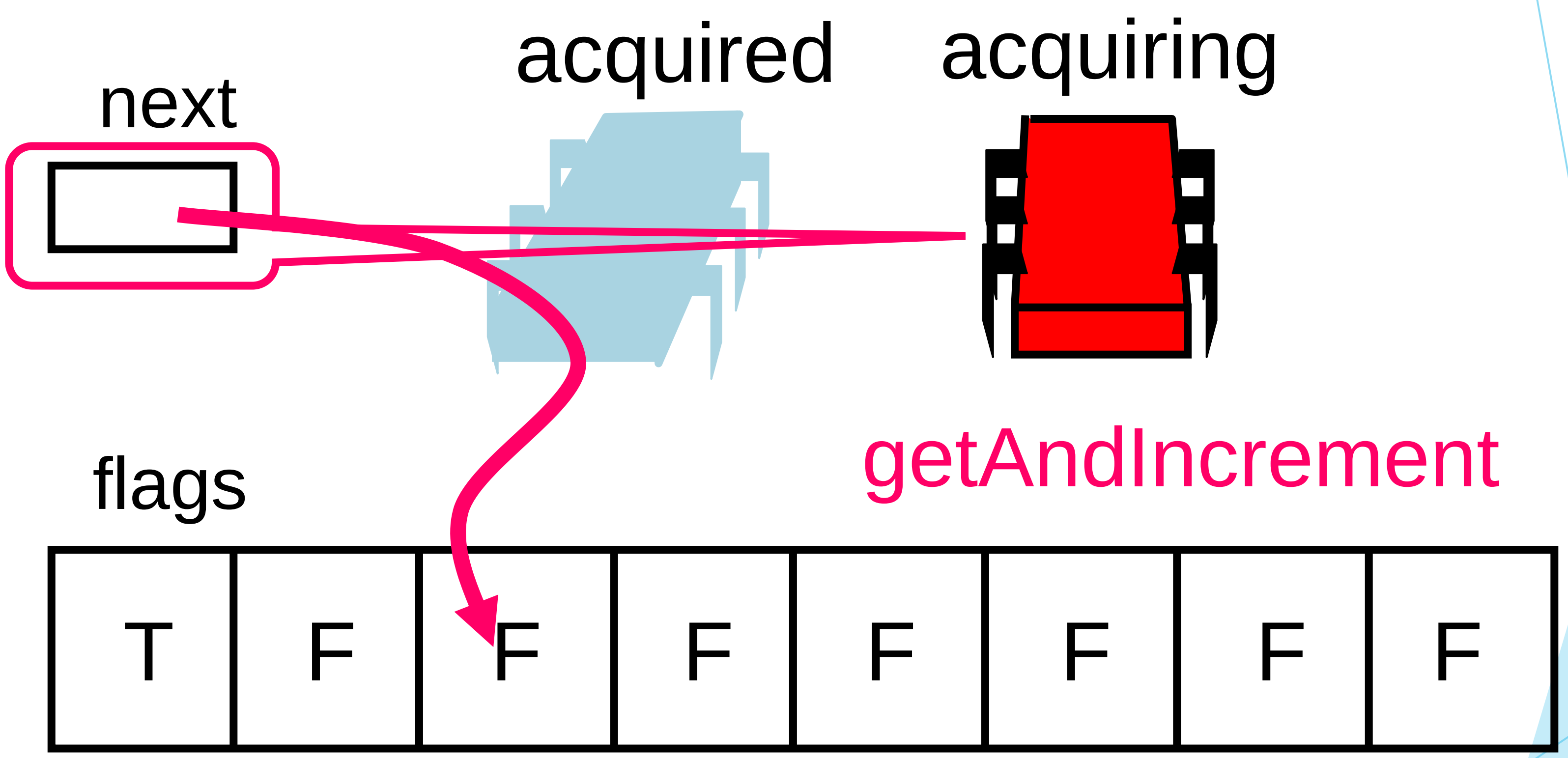
Queue Lock



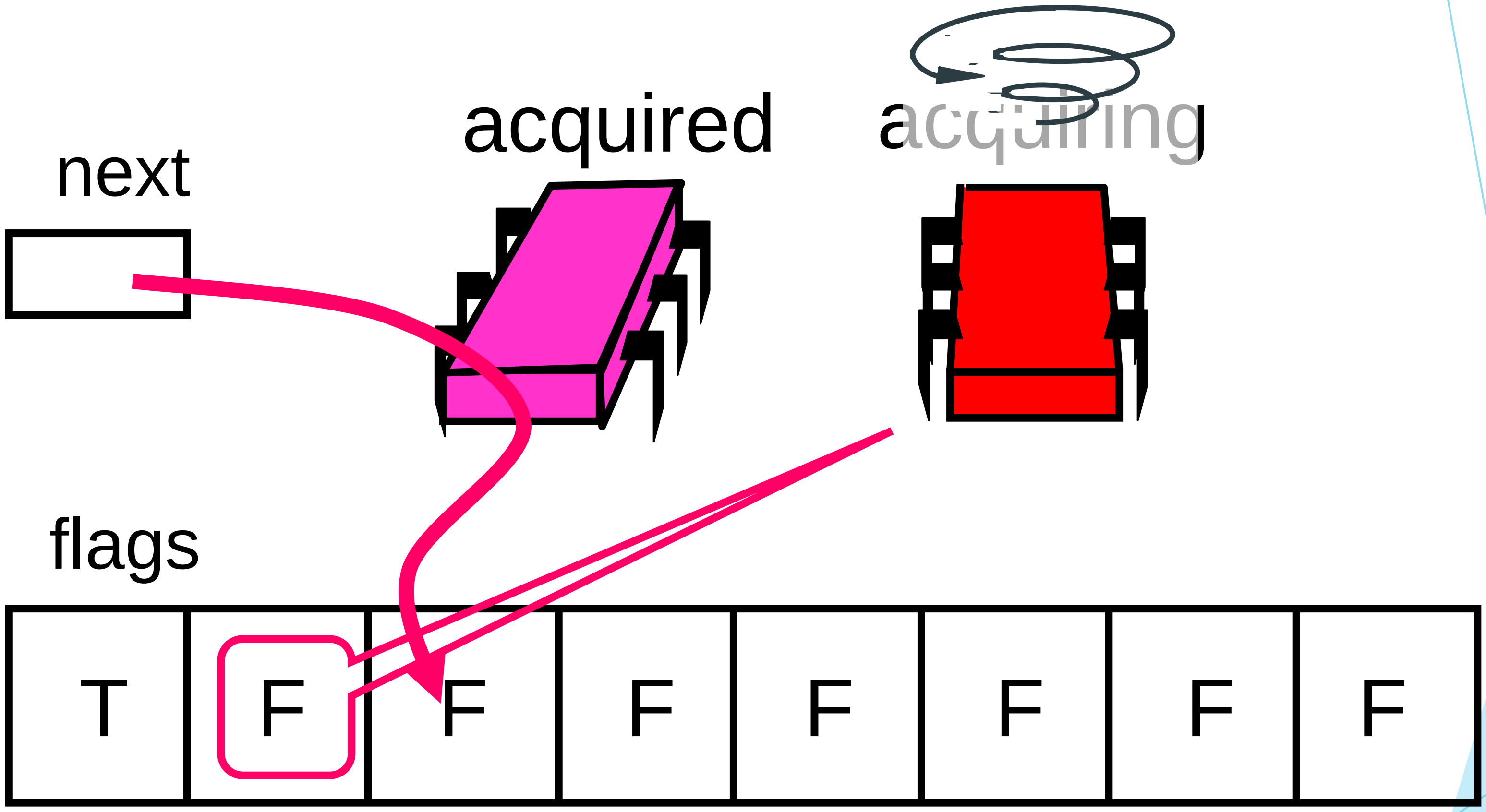
Queue Lock



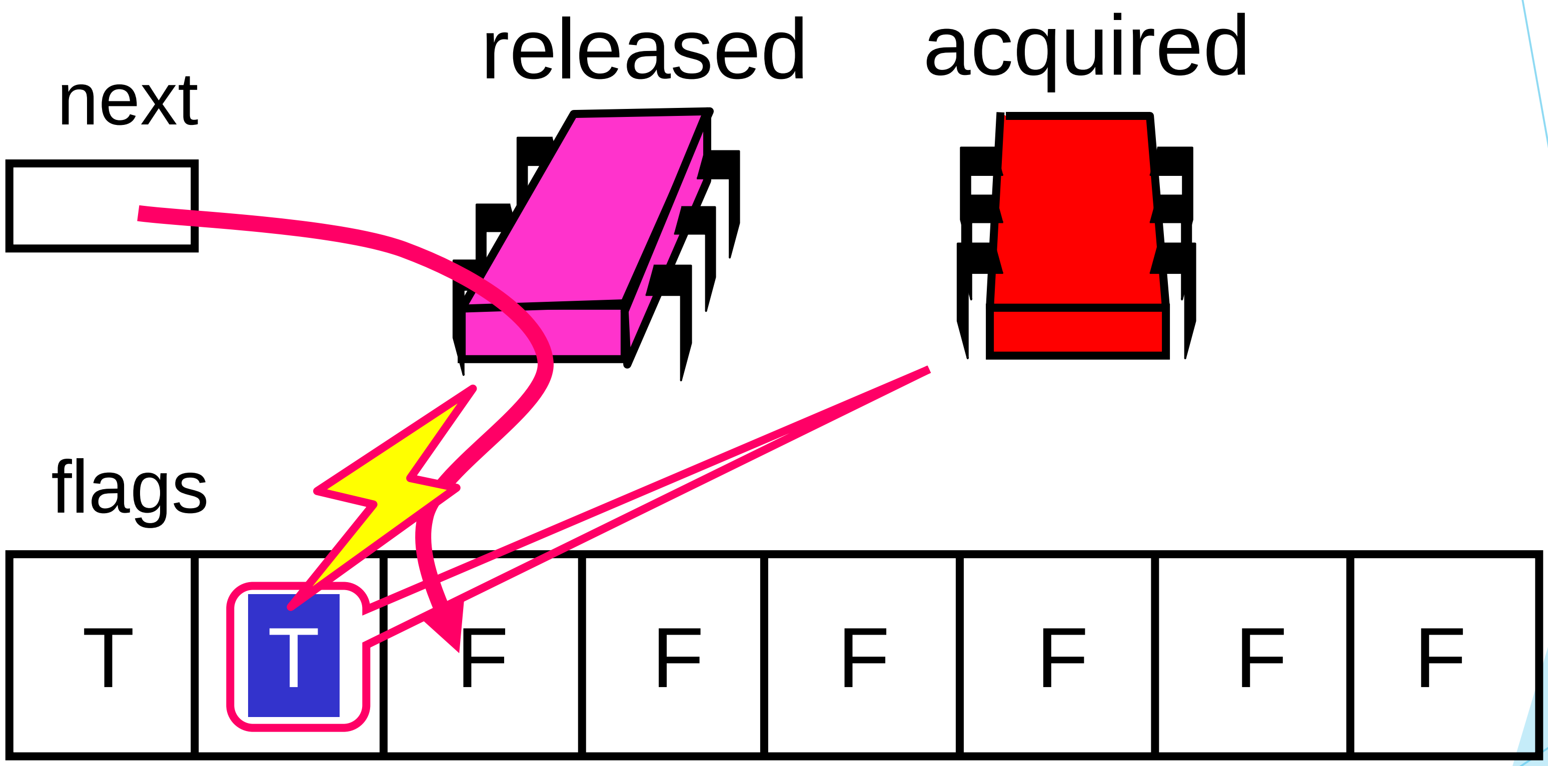
Queue Lock



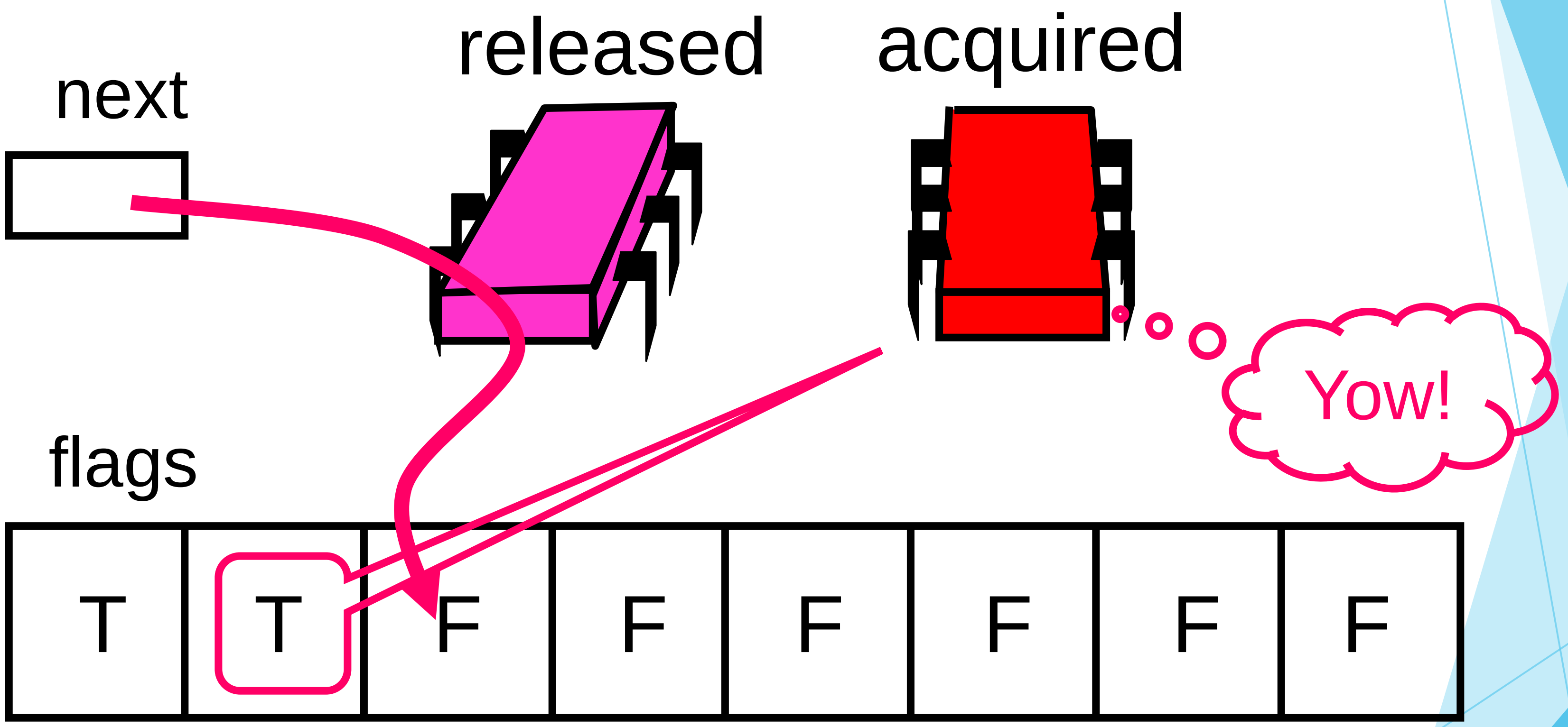
Queue Lock



Queue Lock



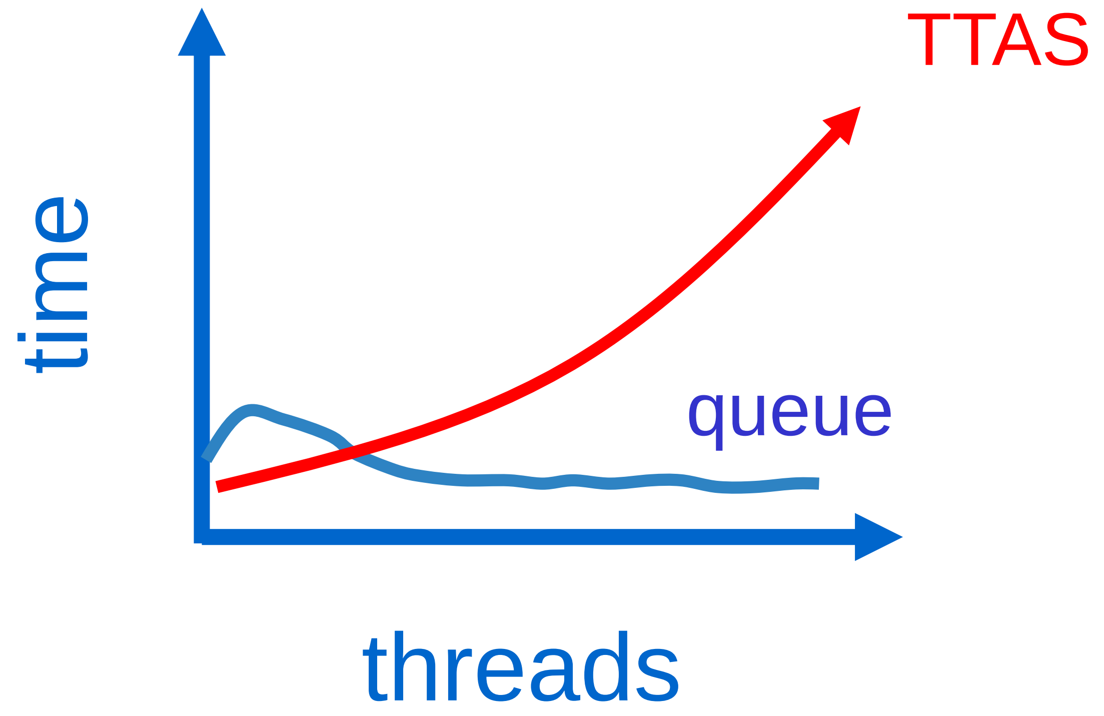
Queue Lock



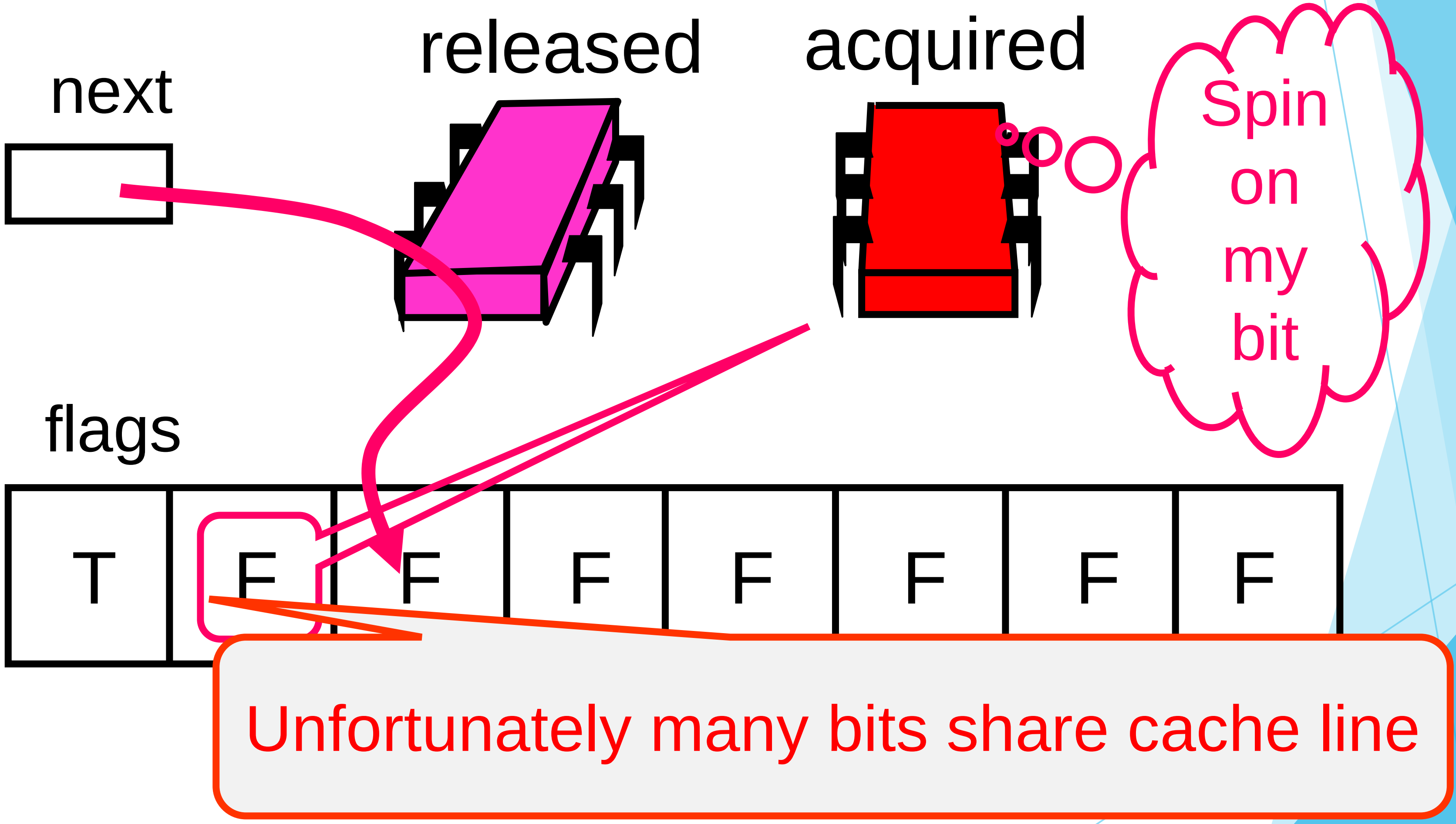
Queue Locks

```
public void lock() {
 int slot = tail.getAndIncrement() % size;
 mySlotIndex.set(slot);
 while (! flag[slot]) {};
 }
public void unlock() {
 int slot = mySlotIndex.get();
 flag[slot] = false;
 flag[(slot + 1) % size] = true;
 } }
```

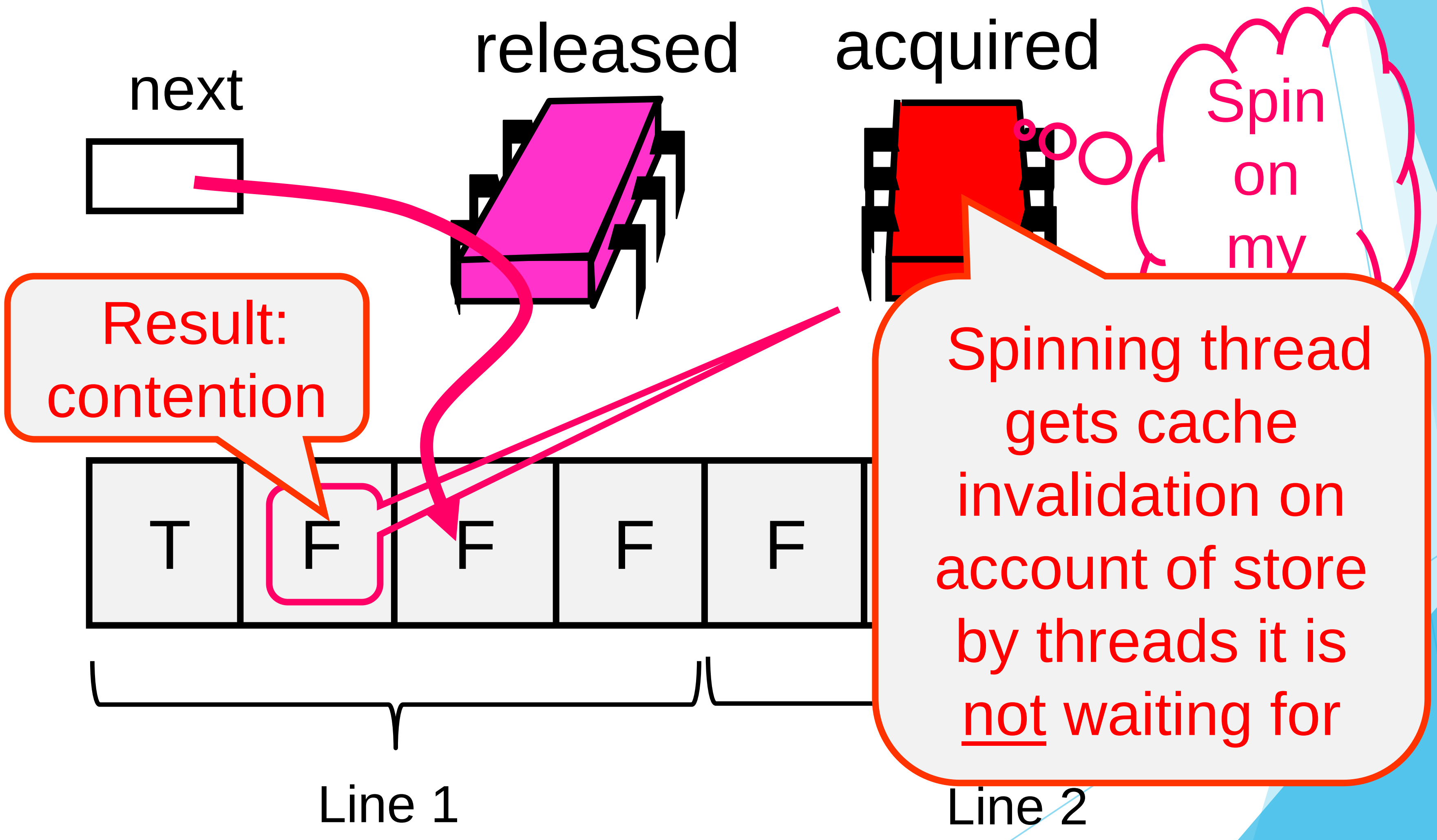
Performance graph



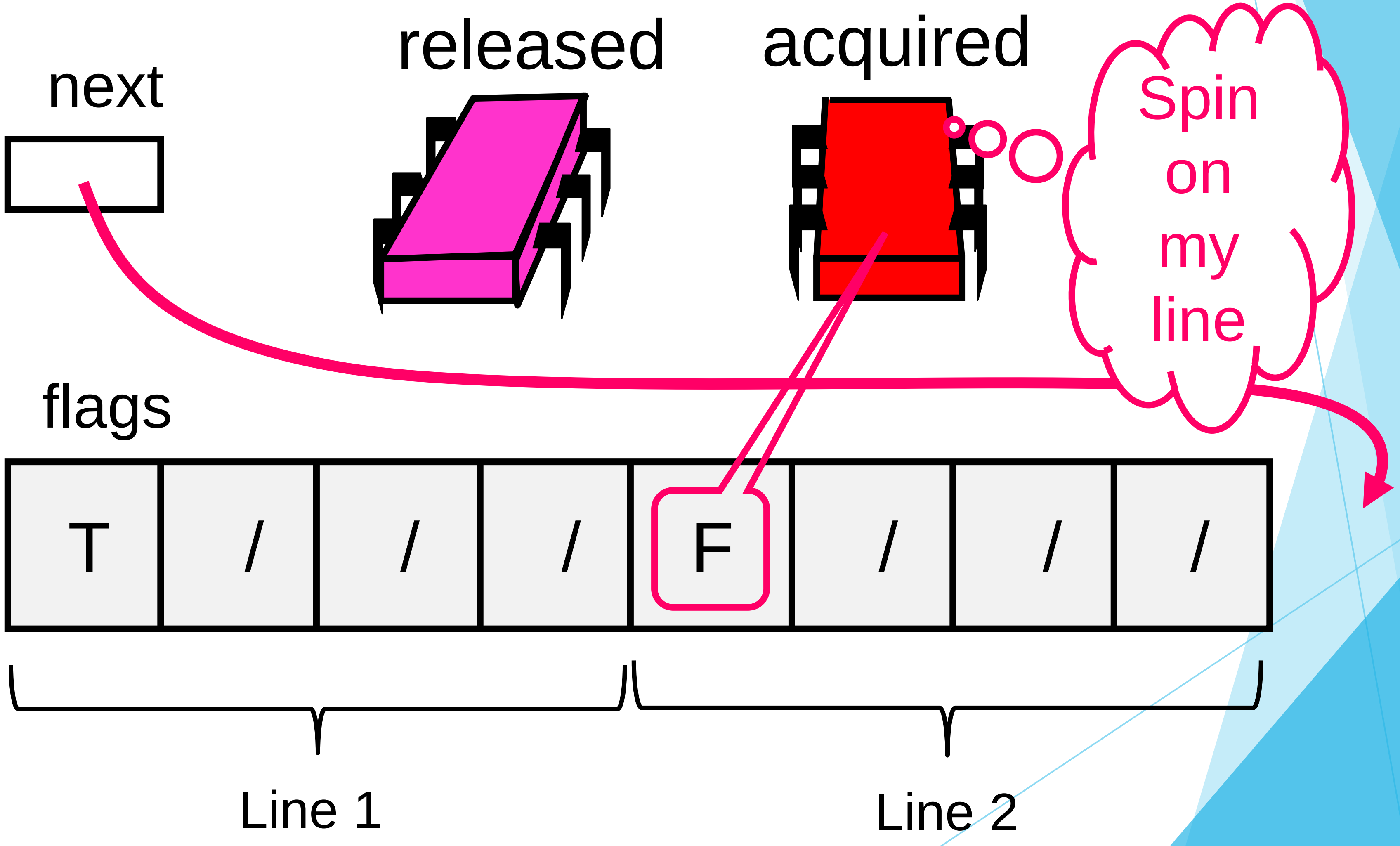
Local Spinning



False Sharing



Solution: Padding



Queue Locks

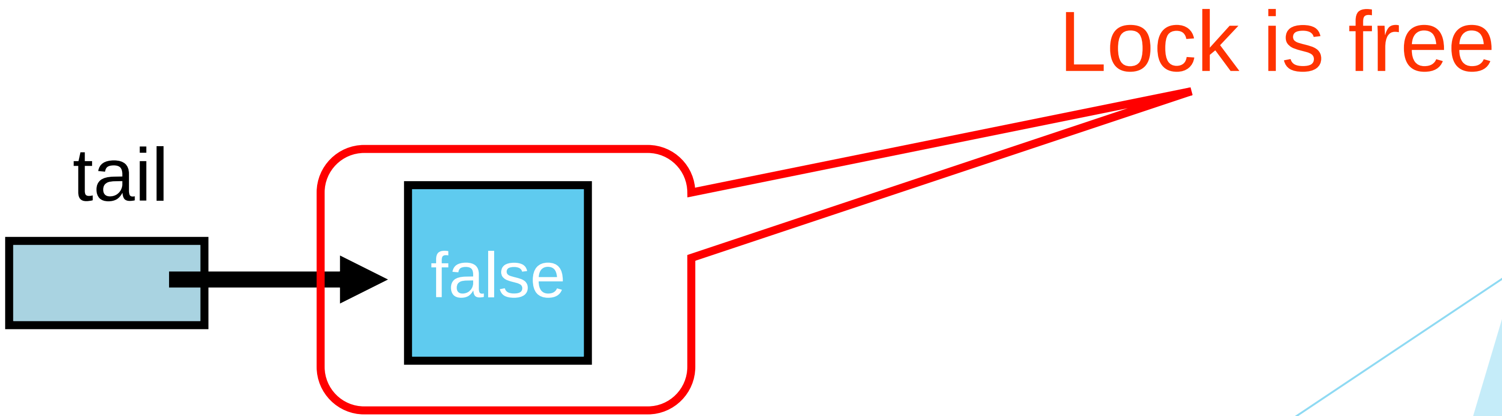
Pros:

- first-come-first-served fairness.
- Easy to implement.
- Scalable performance.

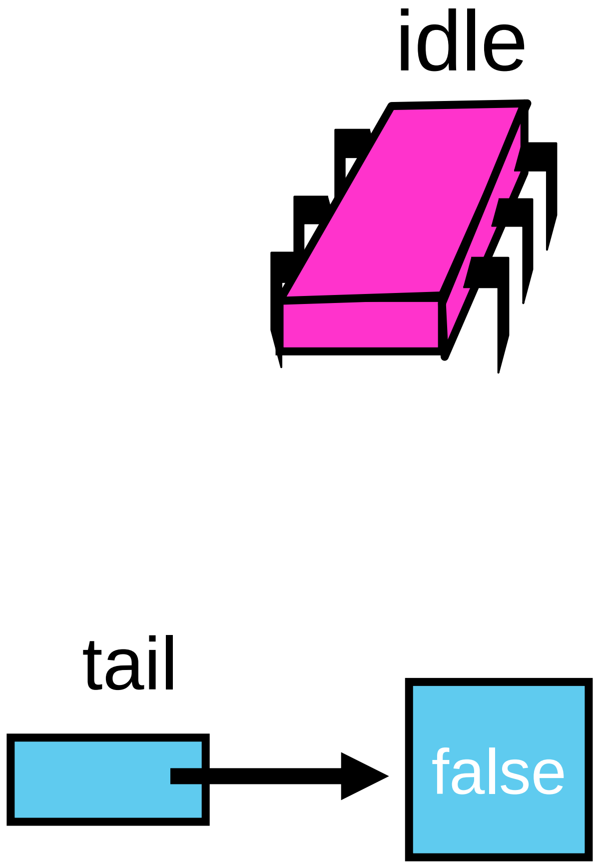
Cons:

- Not space-efficient o(n) **per lock** for n threads.
- Specifically one cache line per thread.

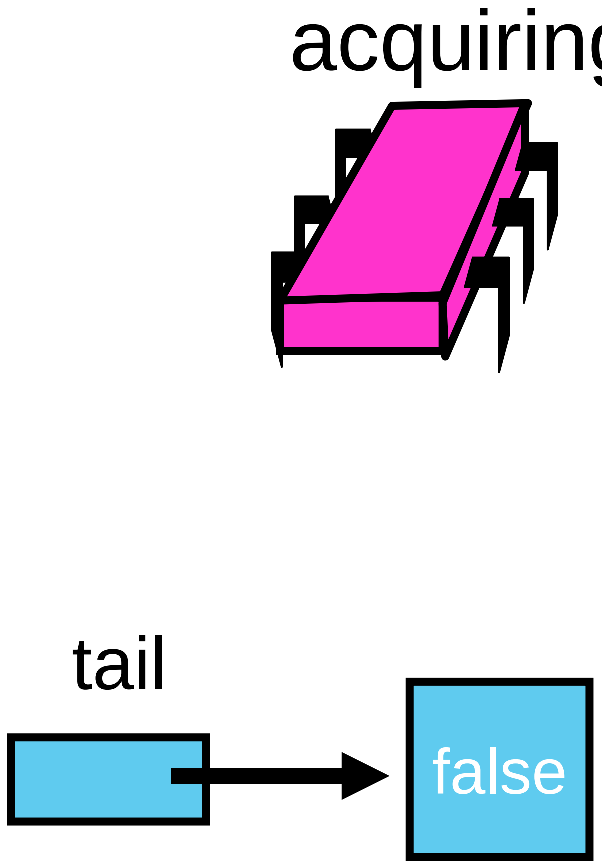
Initially



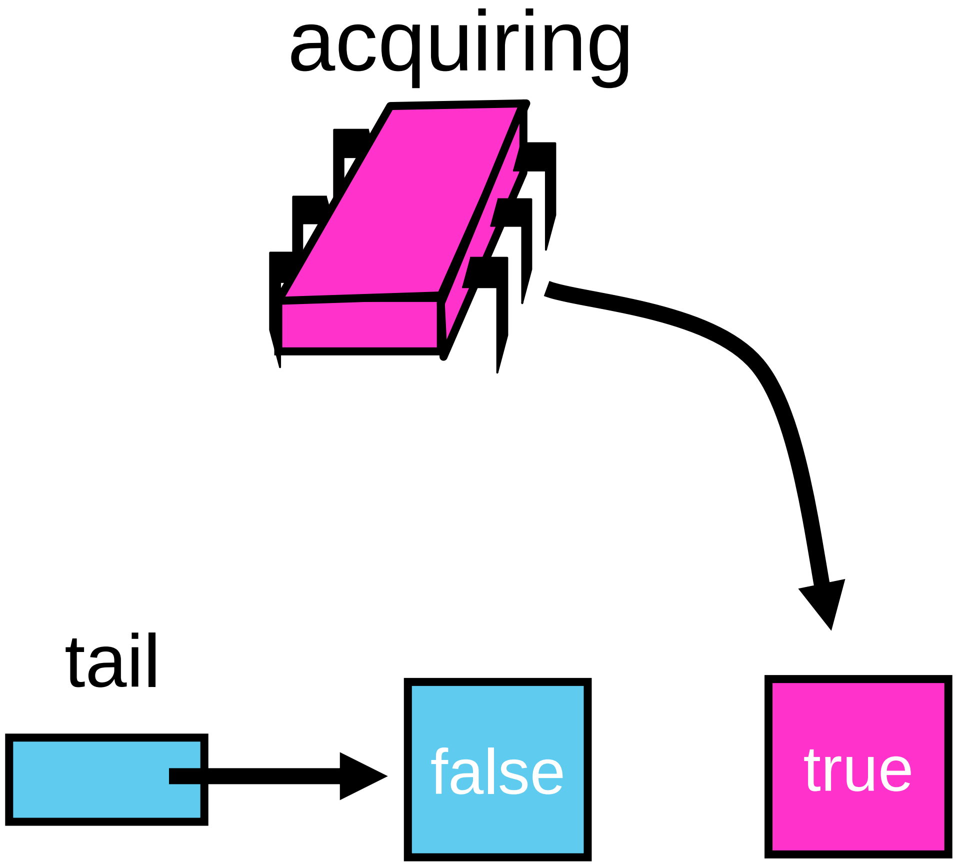
Initially



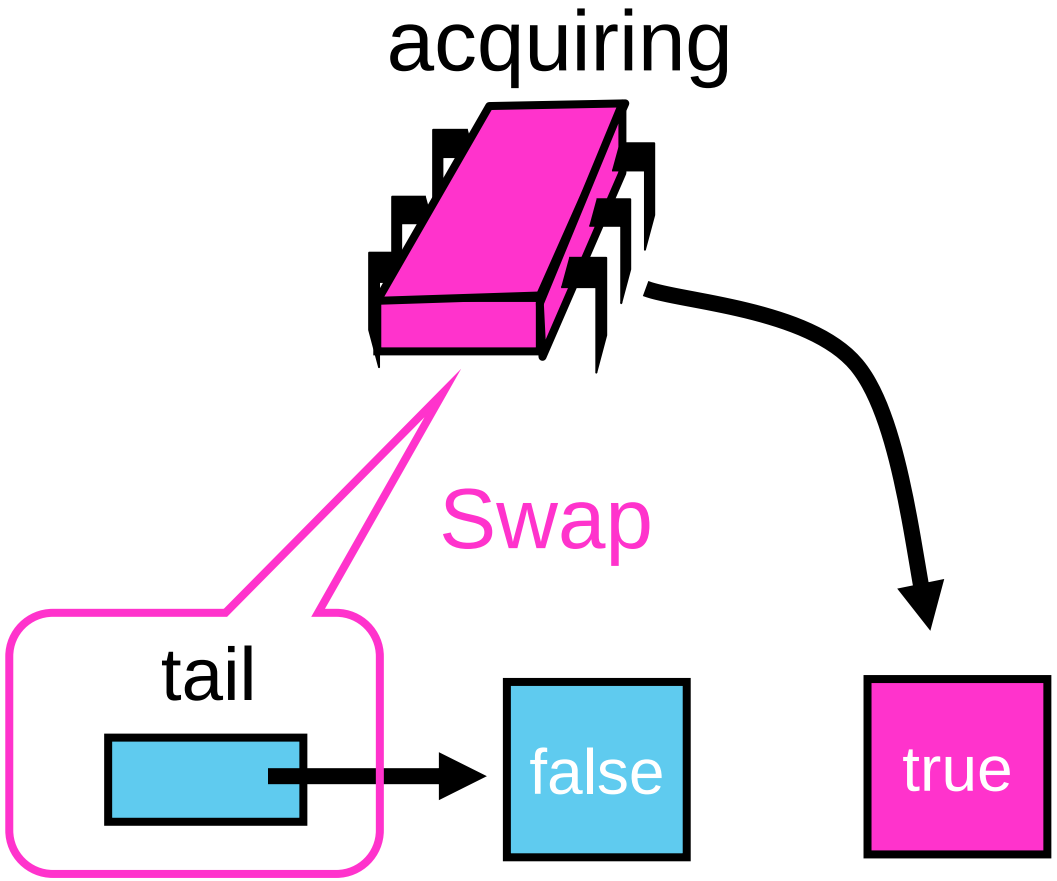
Pink Wants the Lock



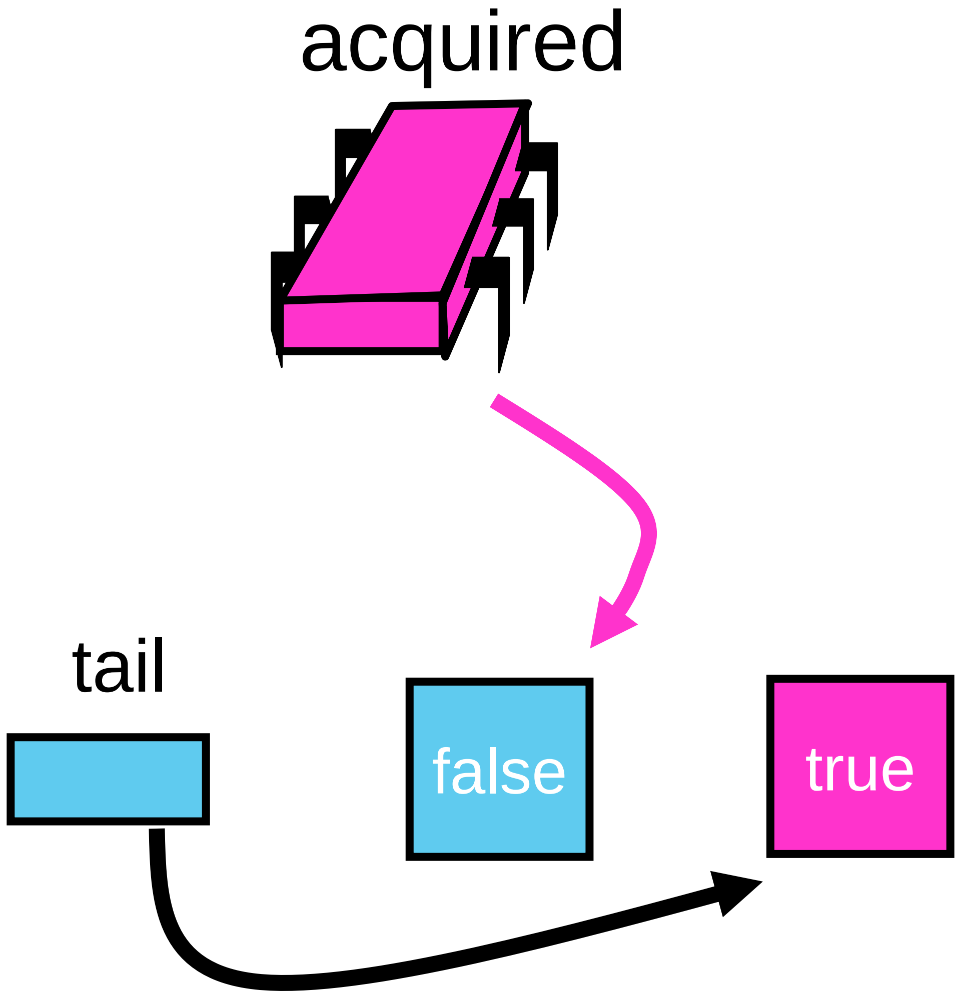
Pink Wants the Lock



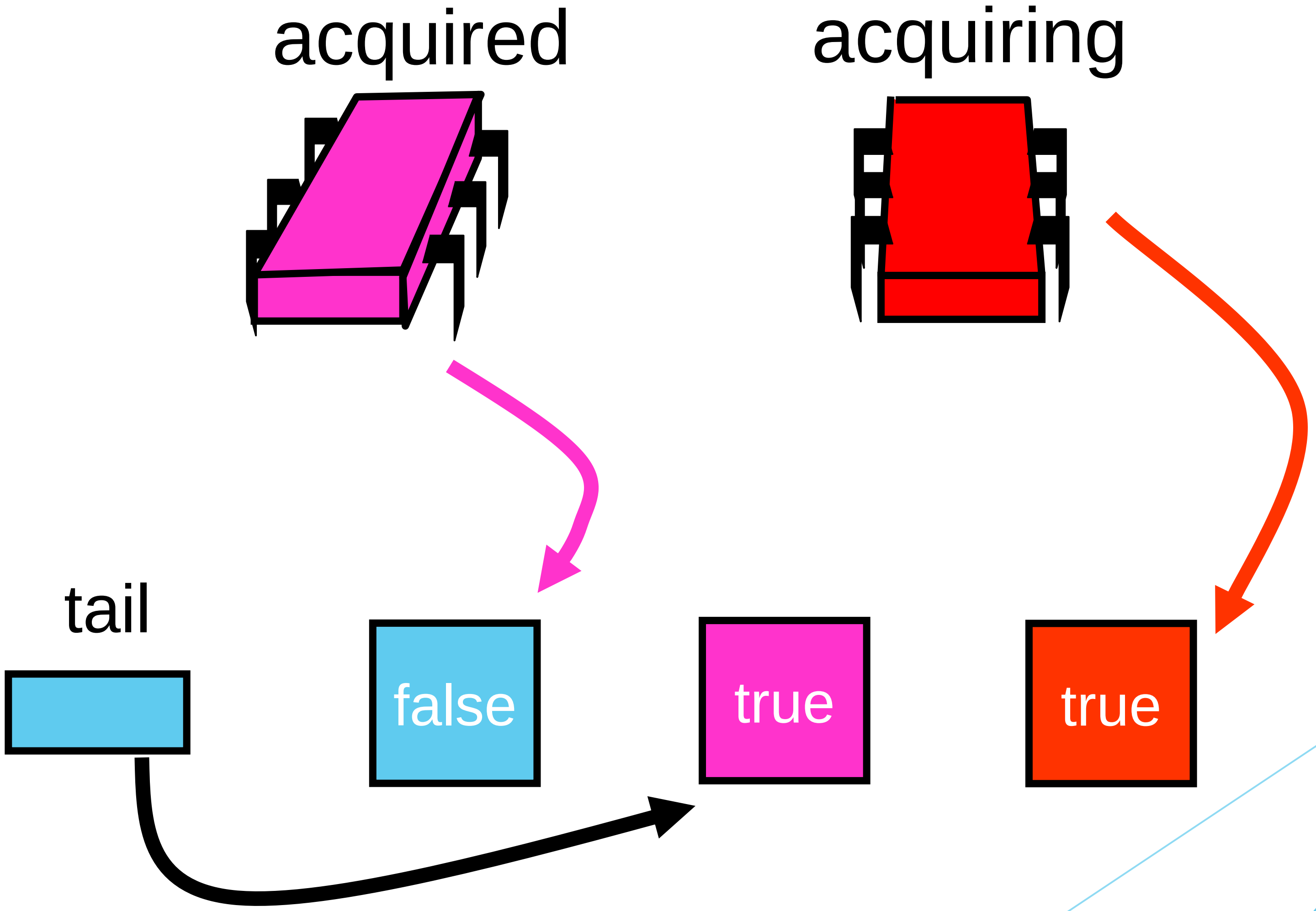
Pink Wants the Lock



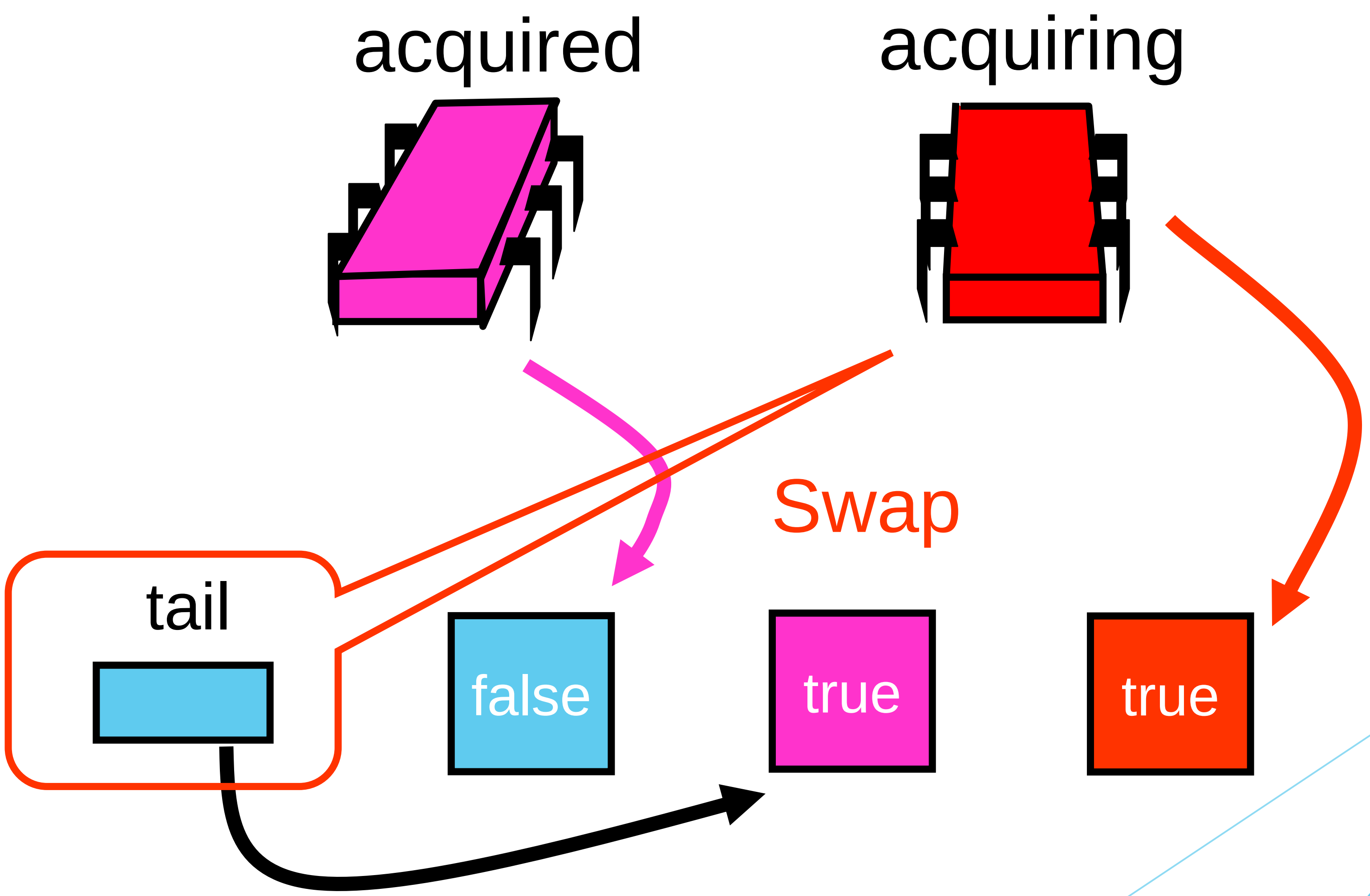
Pink Has the Lock



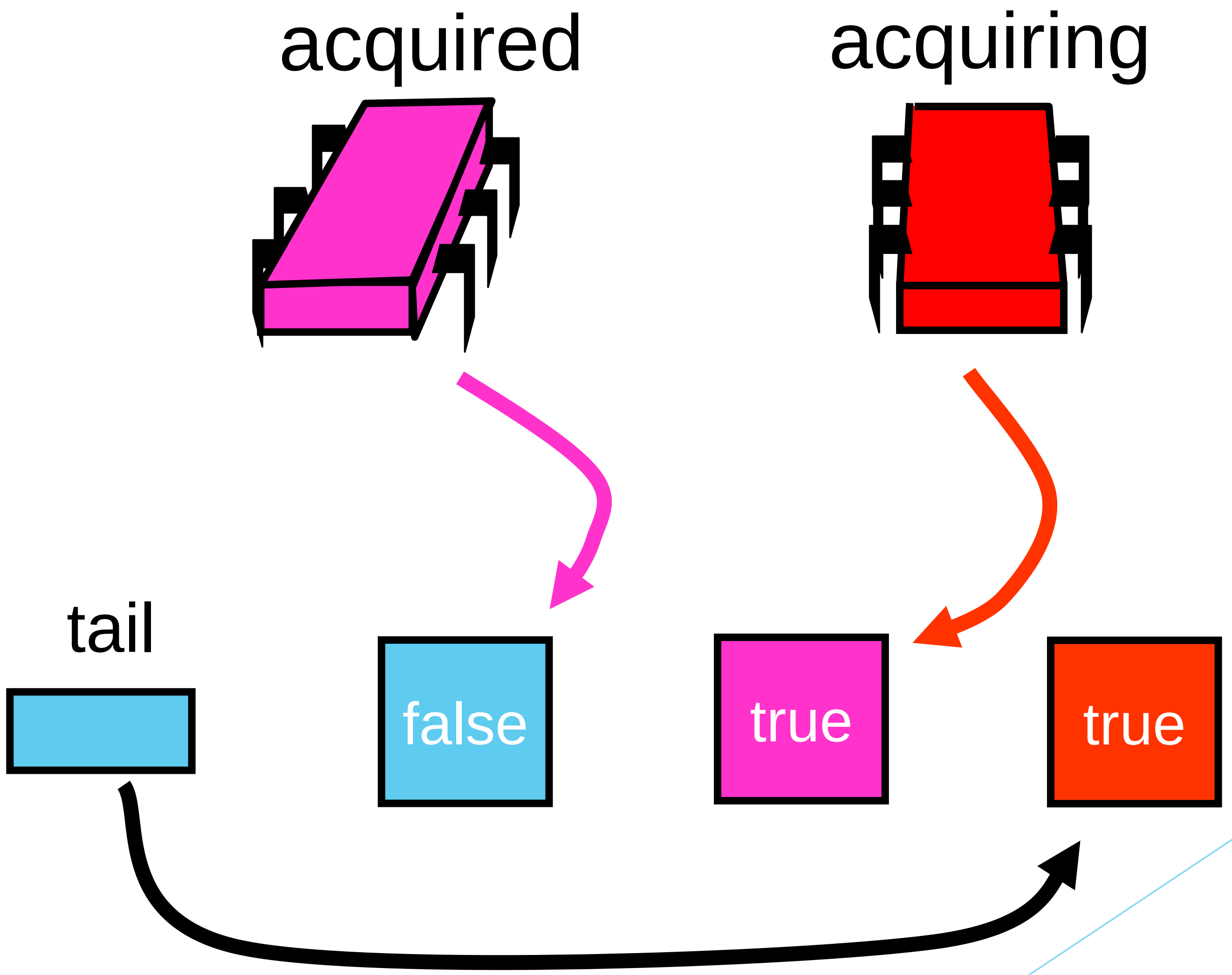
Red Wants the Lock



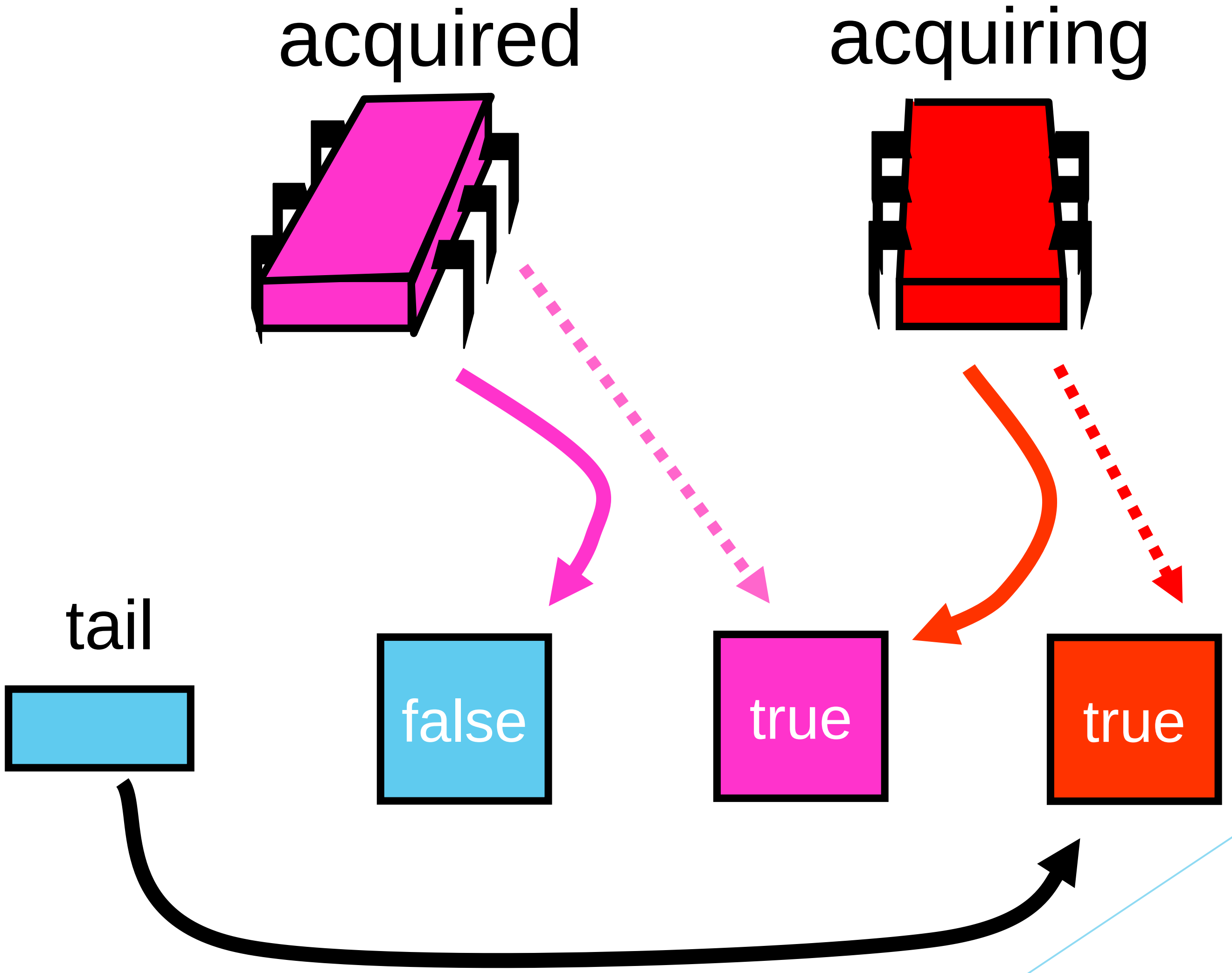
Red Wants the Lock



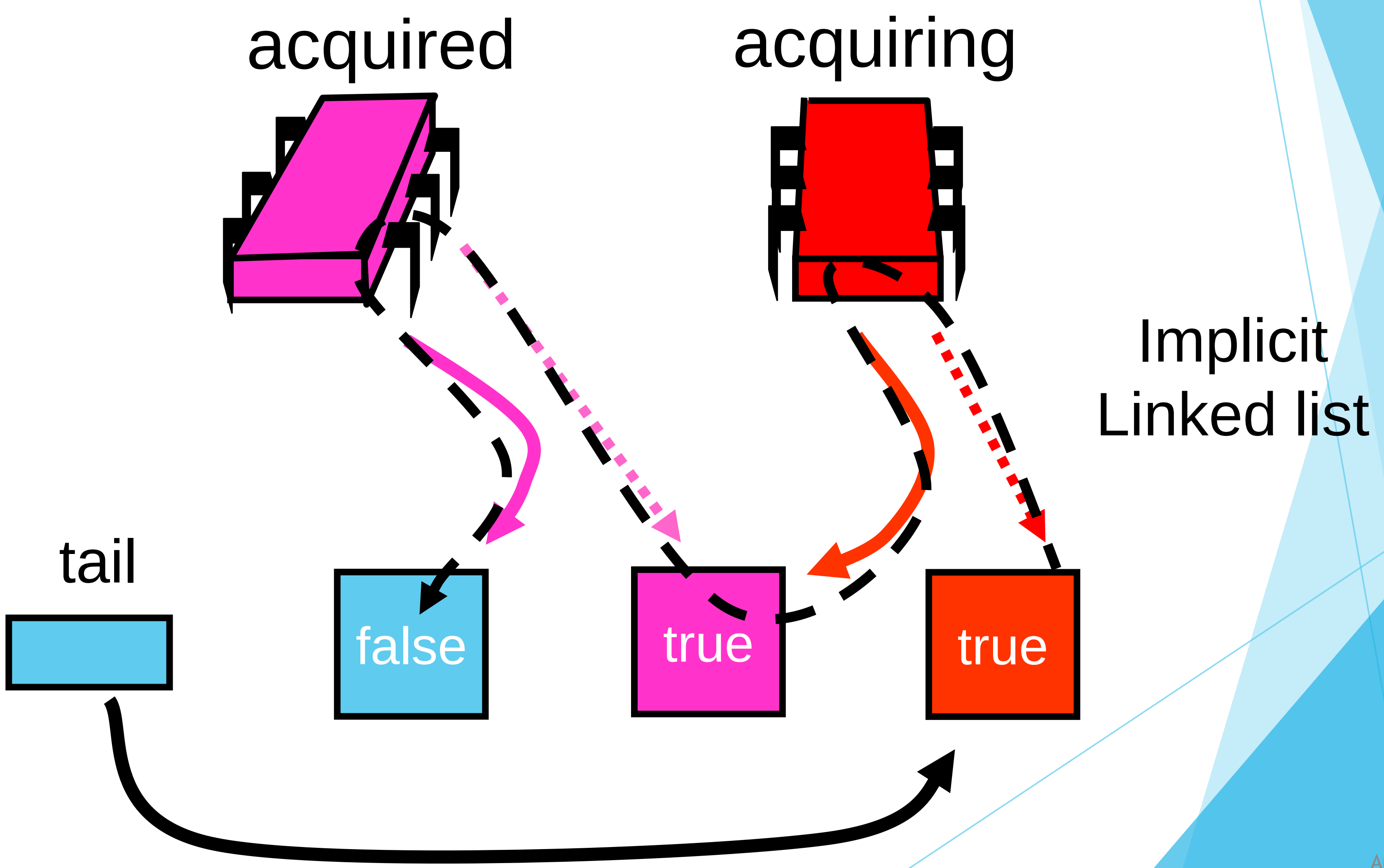
Red Wants the Lock



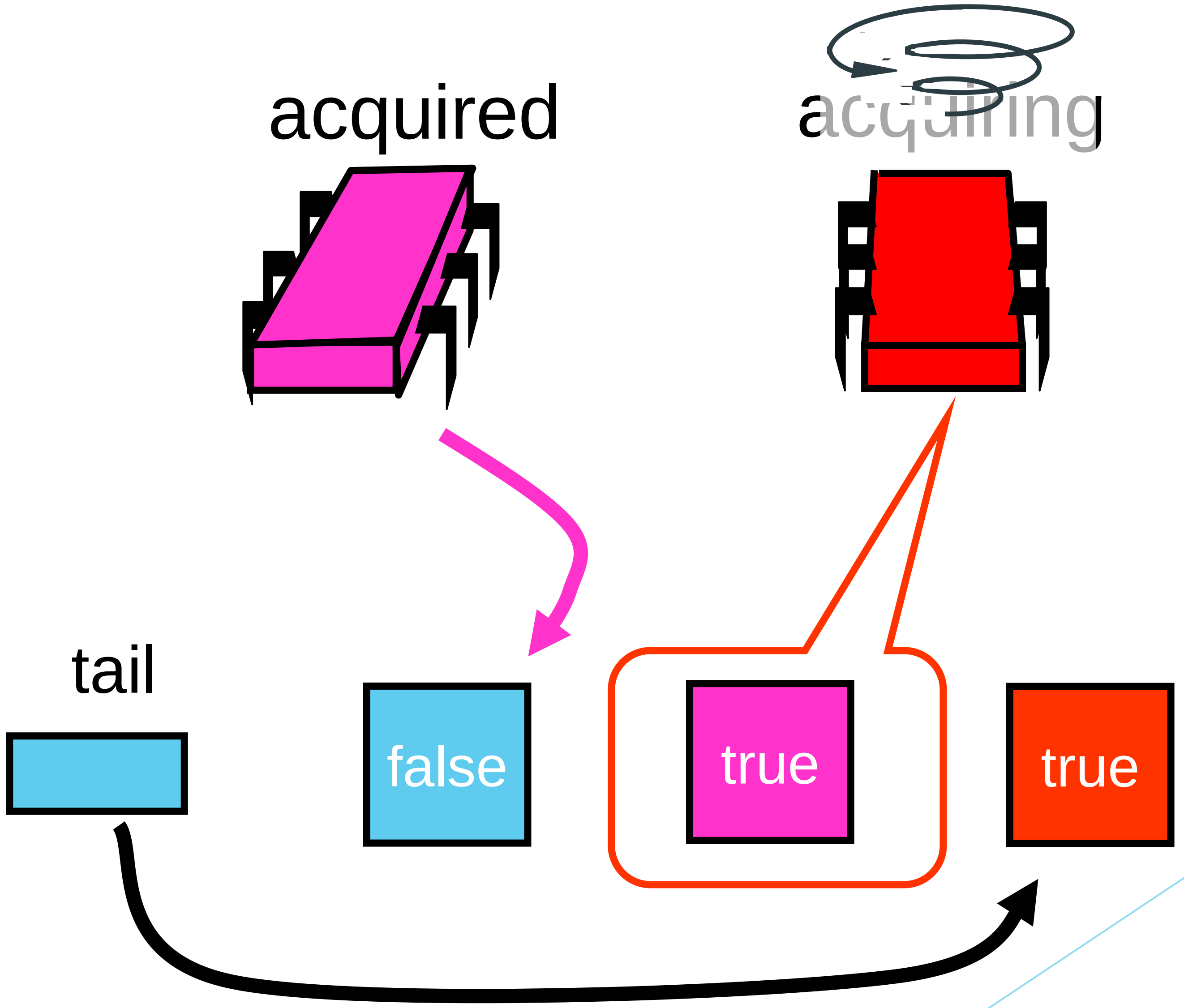
Red Wants the Lock



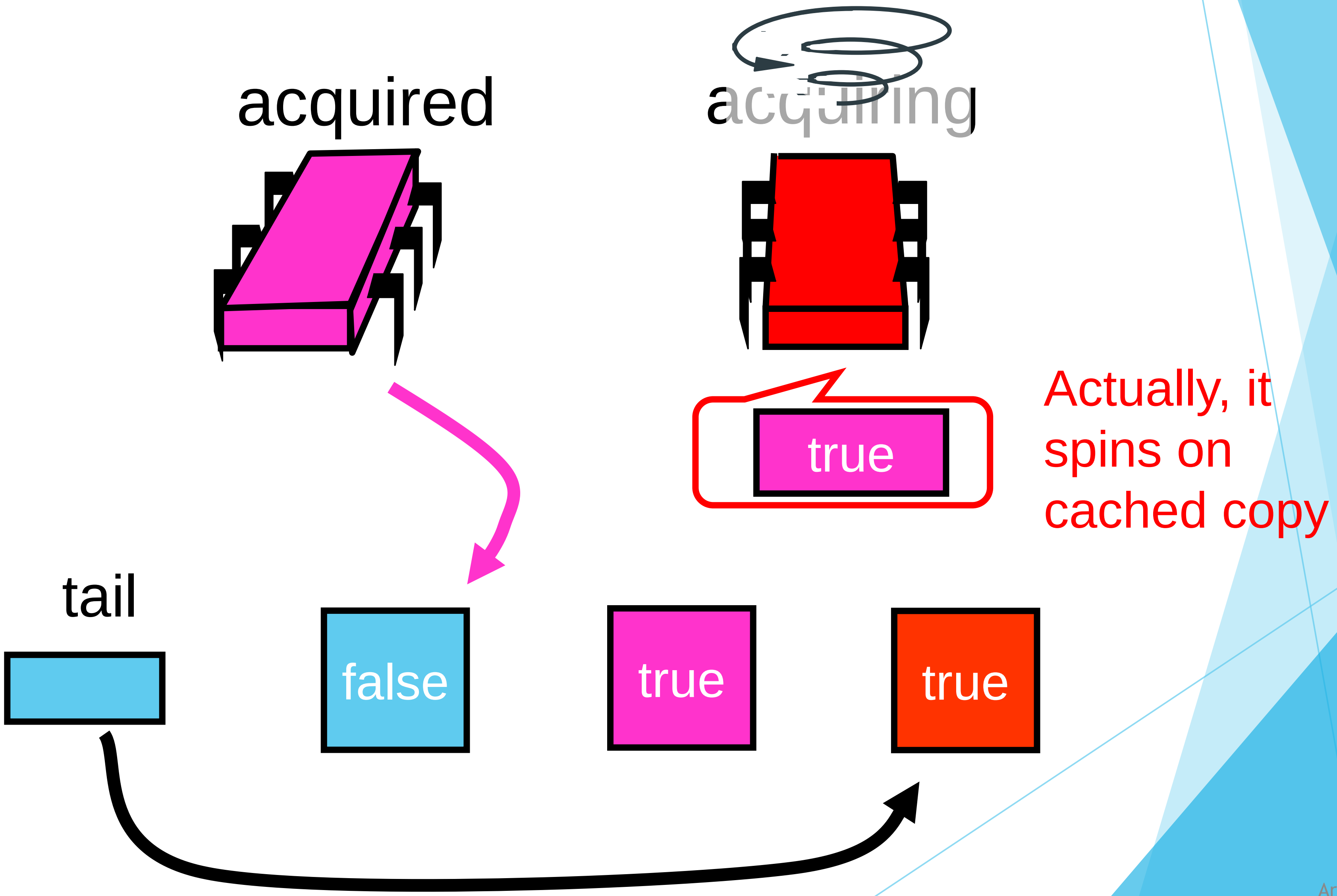
Red Wants the Lock



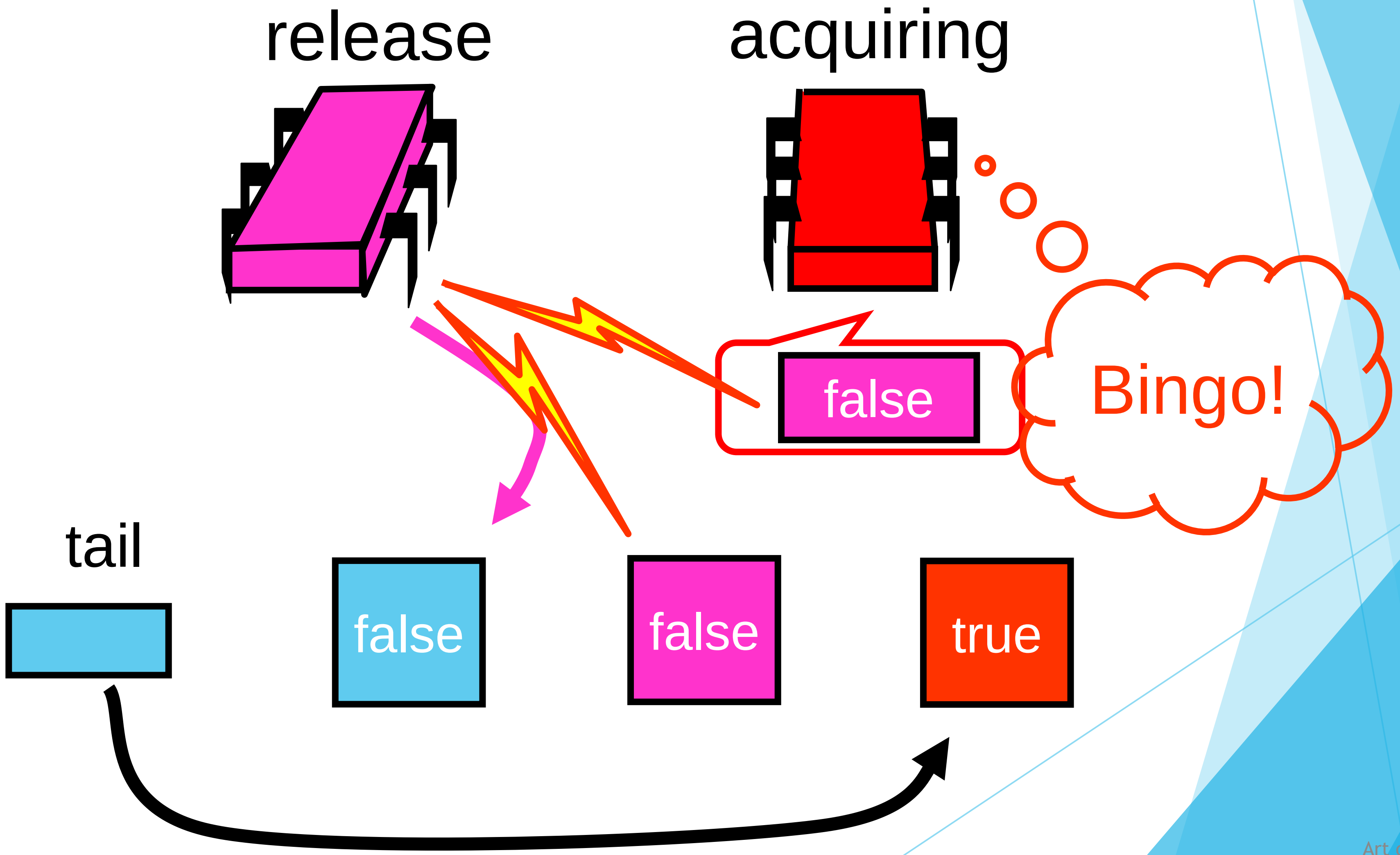
Red Wants the Lock



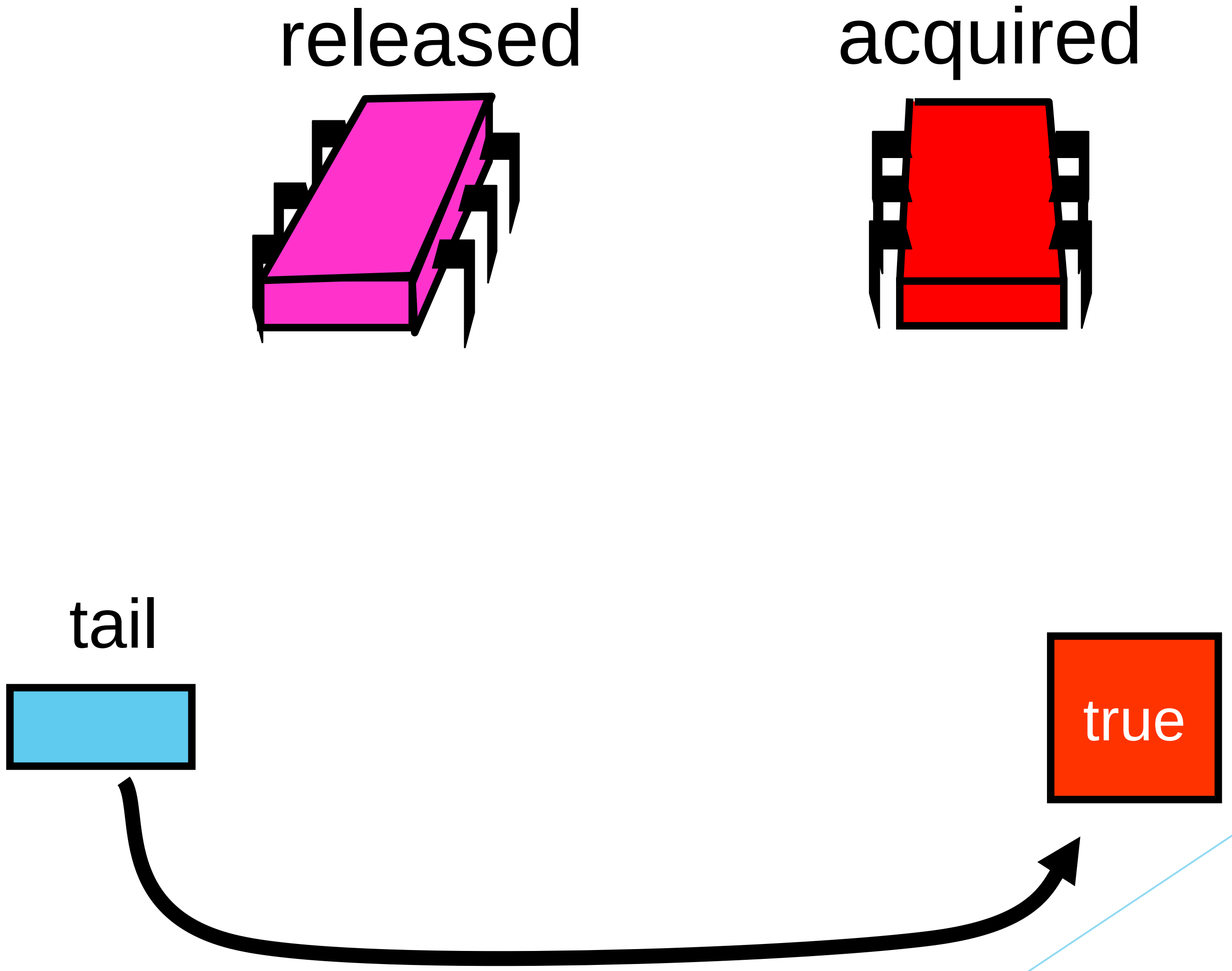
Red Wants the Lock



Pink Releases



Pink Releases



CLH Lock

```
public void lock() {
QNode qnode = myNode.get();
qnode.locked = true;
QNode pred = tail.getAndSet(qnode);
while (pred.locked) {}
}
public void unlock() {
 myNode.locked.set(false);
 myNode=pred
```

CLH Lock

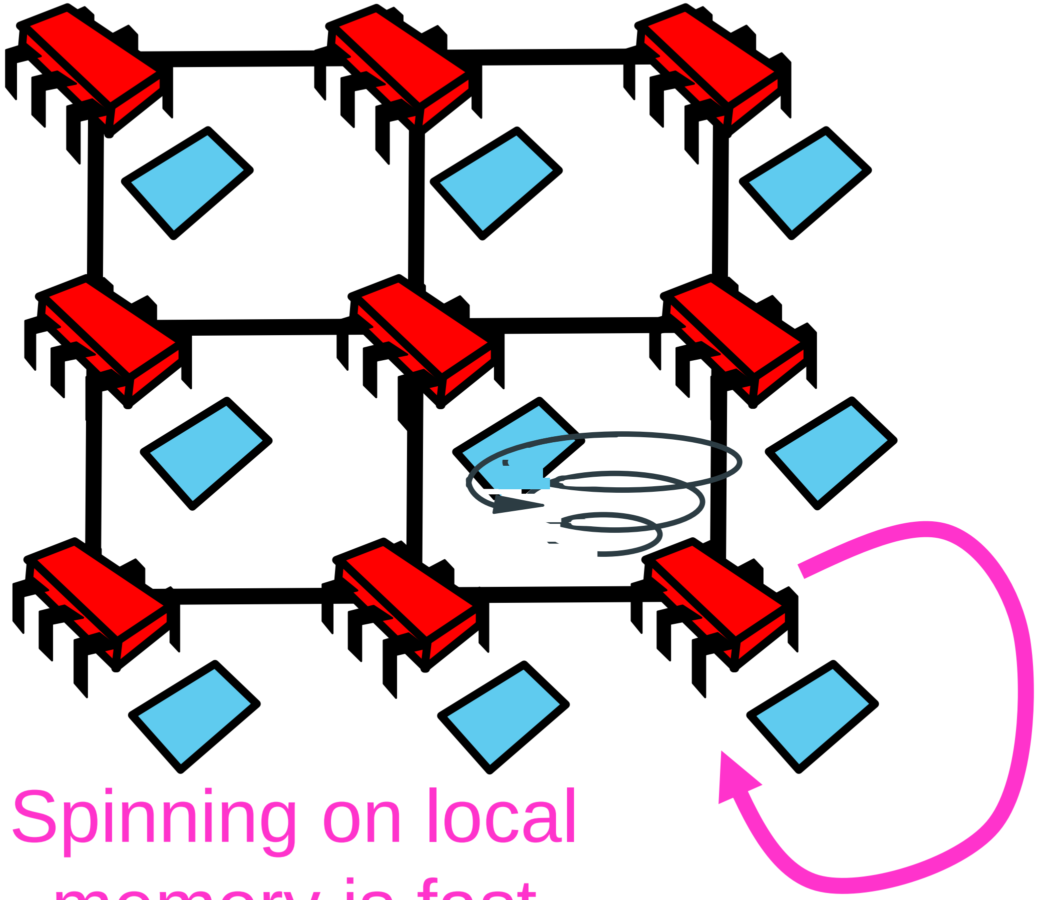
Pros:

- Constant size space.
- First-come-first-served fairness.
- Each thread spin on a distinct location.
- Does not require a knowledge of number of threads to come.

Cons:

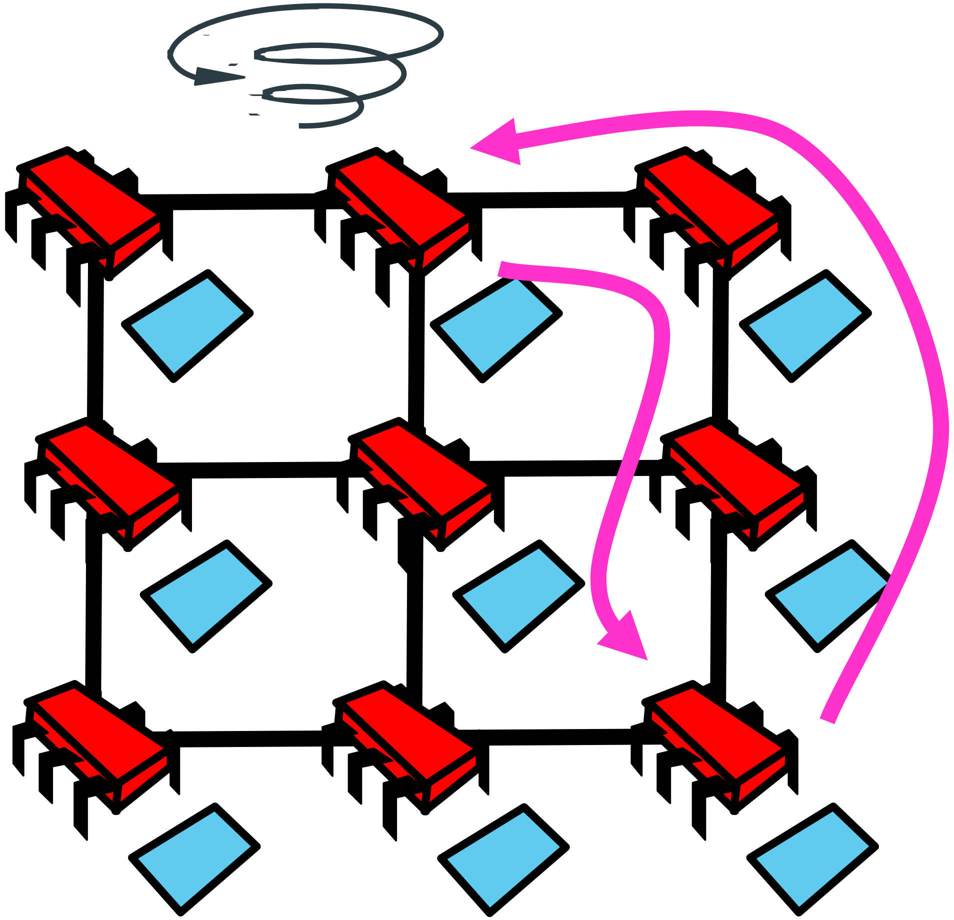
- Doesn’t work for un-cached NUMA architectures.

NUMA Machines



Spinning on local memory is fast

NUMA Machines



Spinning on remote memory is slow

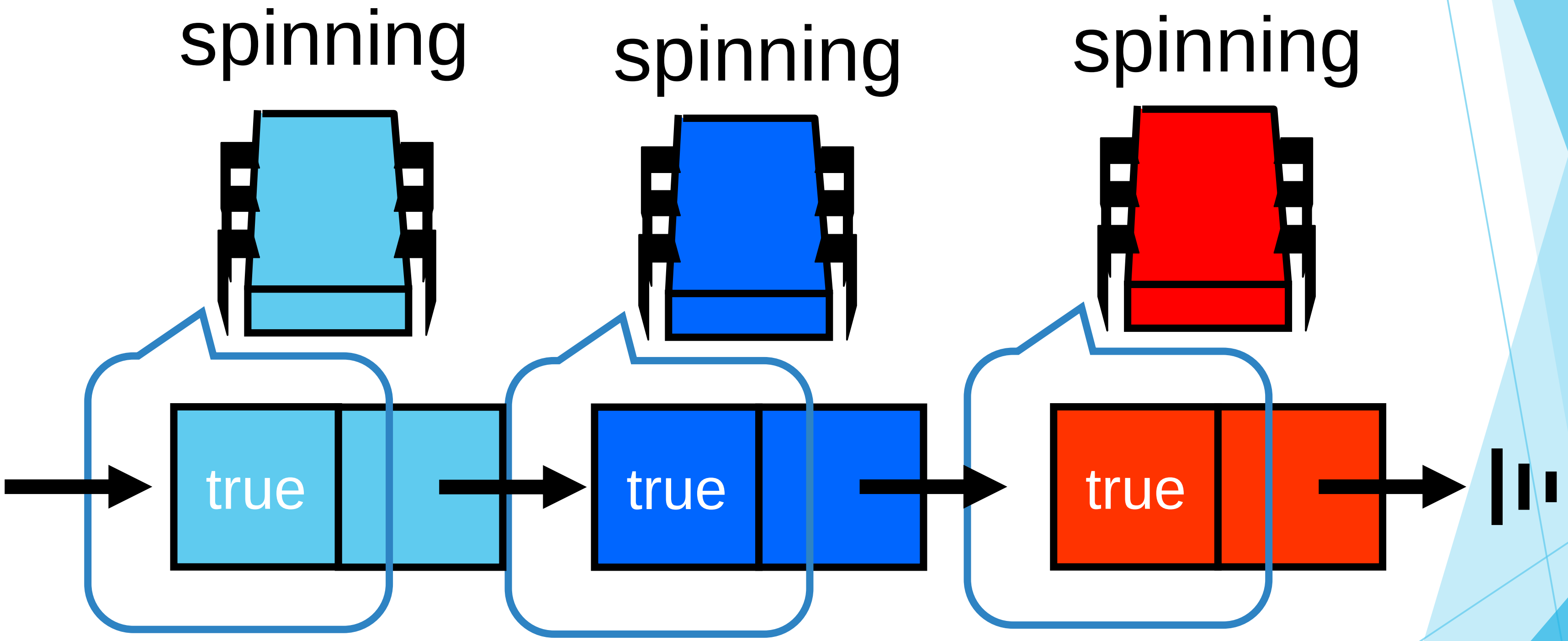
MCS Queue

- The thread spins on a (local) locked field in its own QNode waiting until its predecessor sets this field to **false**.
- As a result each thread controls the location on which it spins.

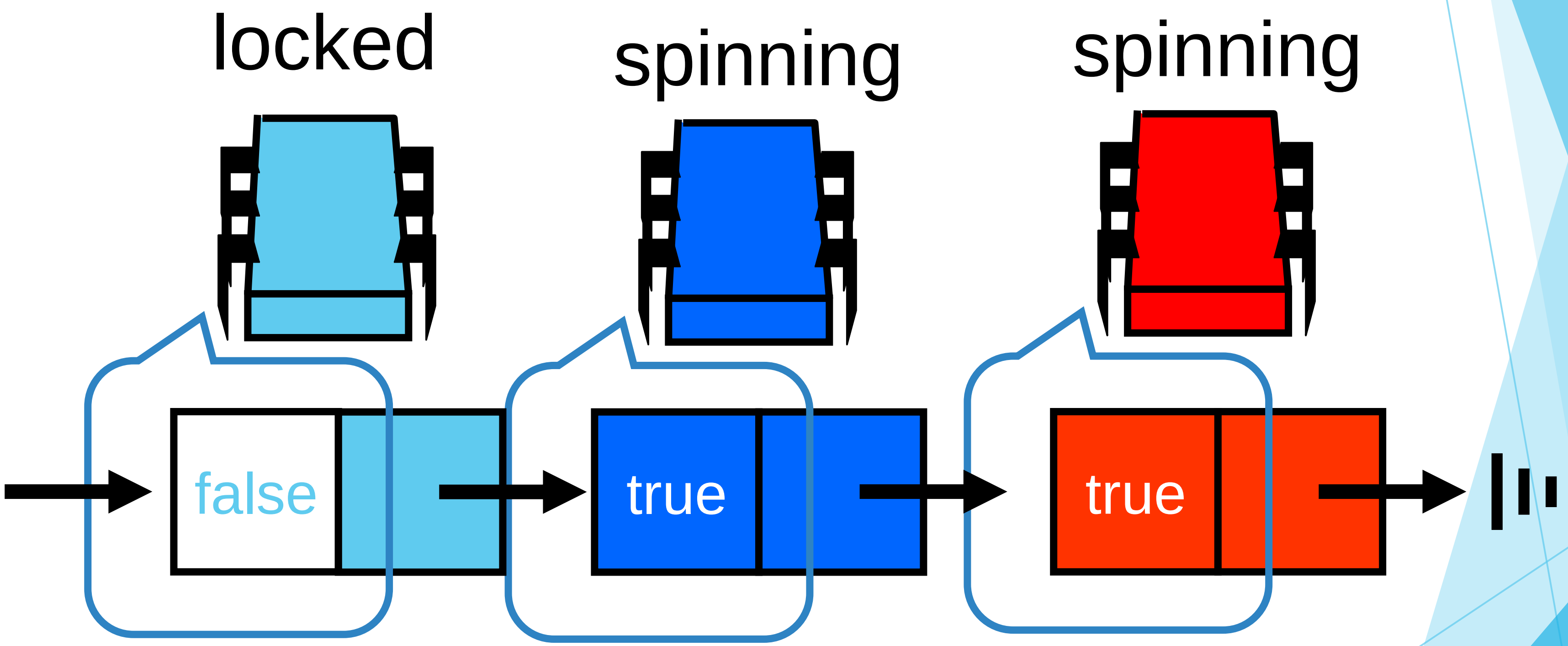
Lock with timeout

- Up till now we studied first come first serve with some contention algorithm.
- In real time system sometimes thread need the ability to give up waiting for lock.
- In Backoff lock the thread can simply return from the lock() function.
- In queue lock algorithm if a thread just returns the threads queued up behind it will starve.

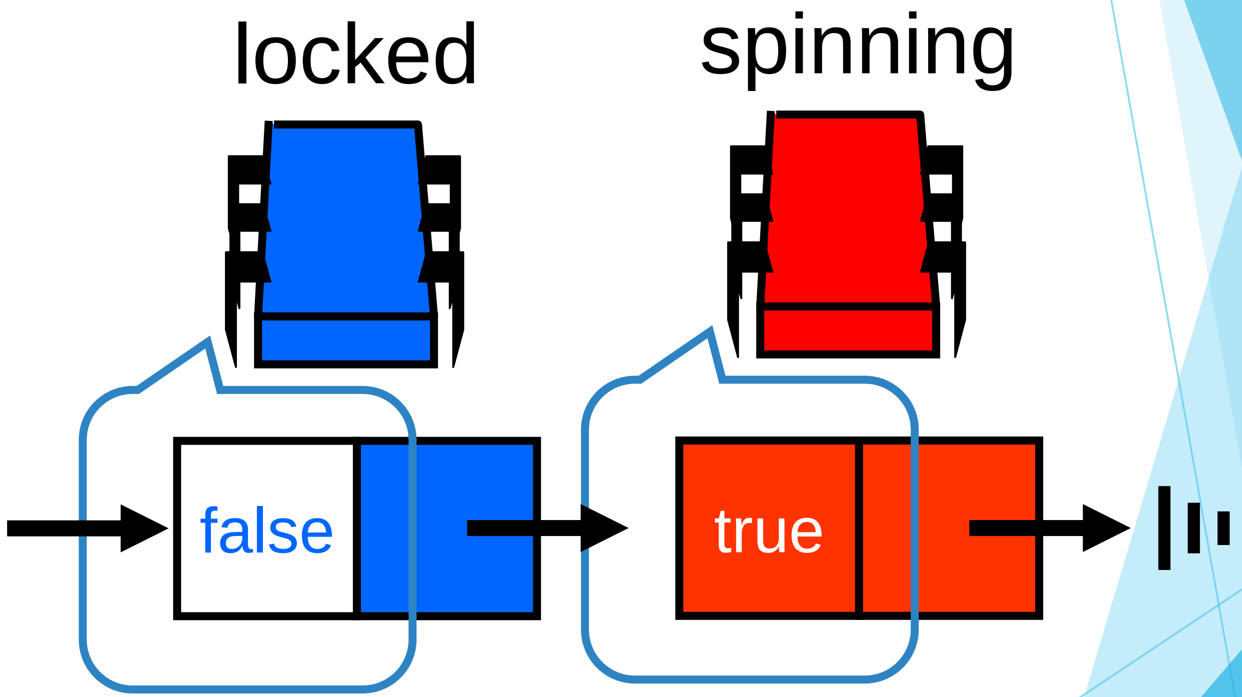
Queue Locks



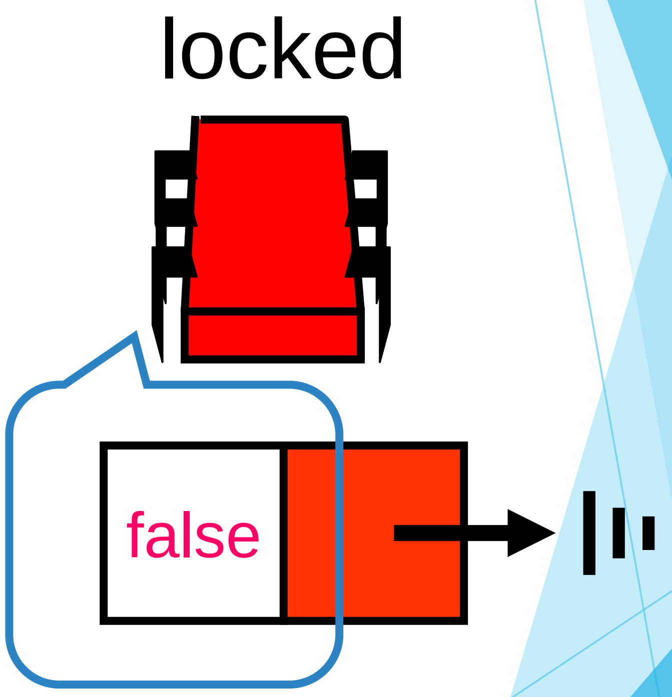
Queue Locks



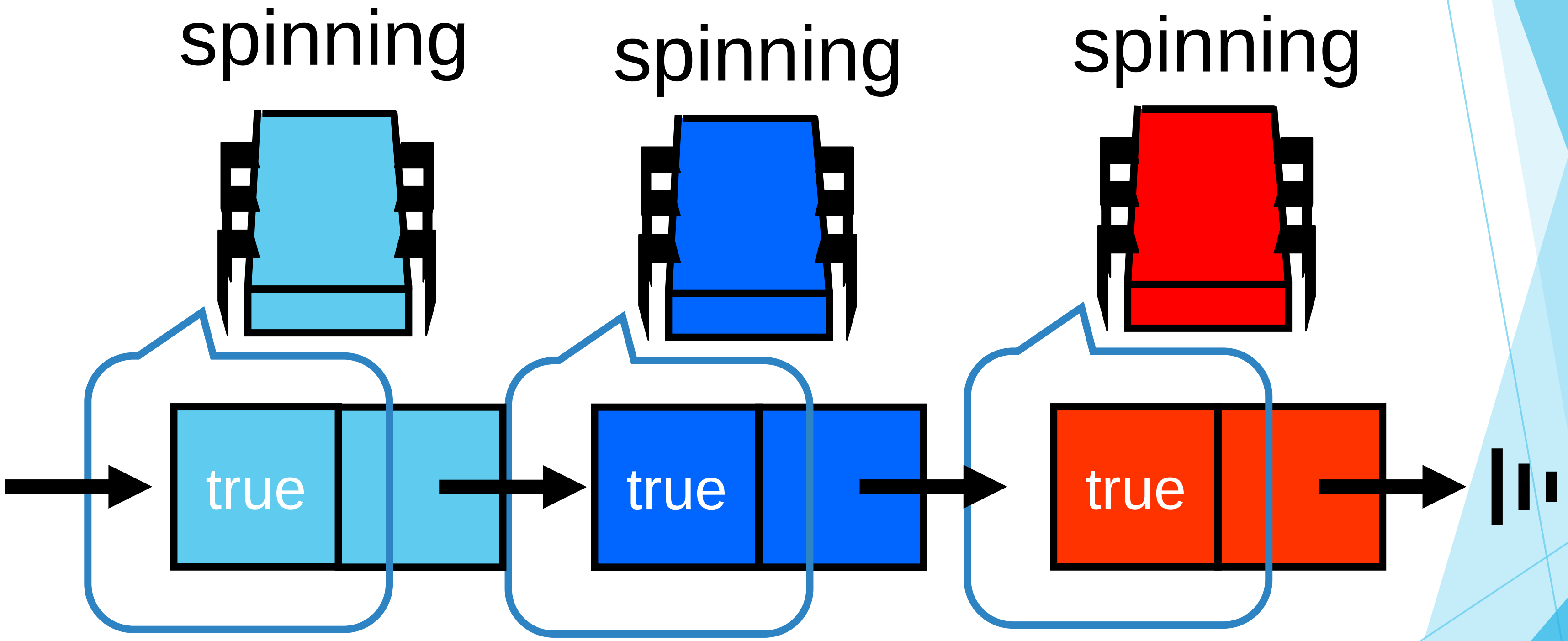
Queue Locks



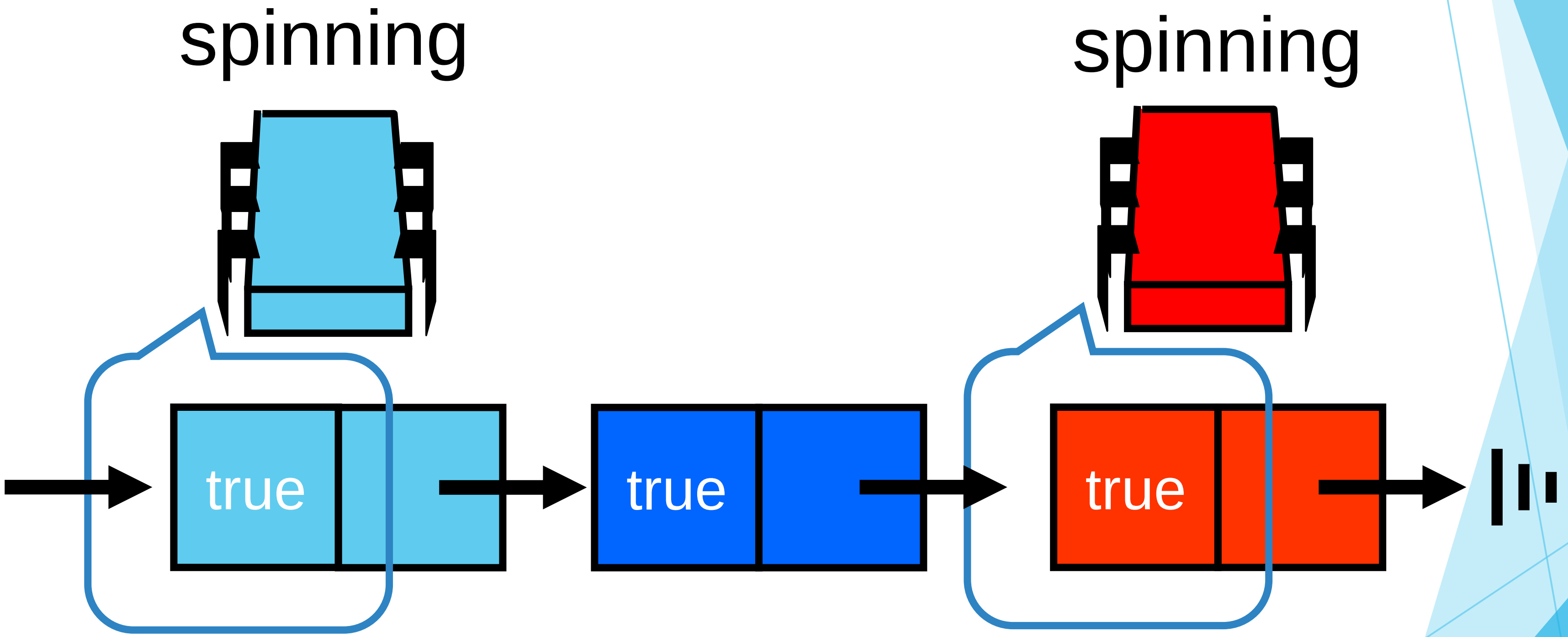
Queue Locks



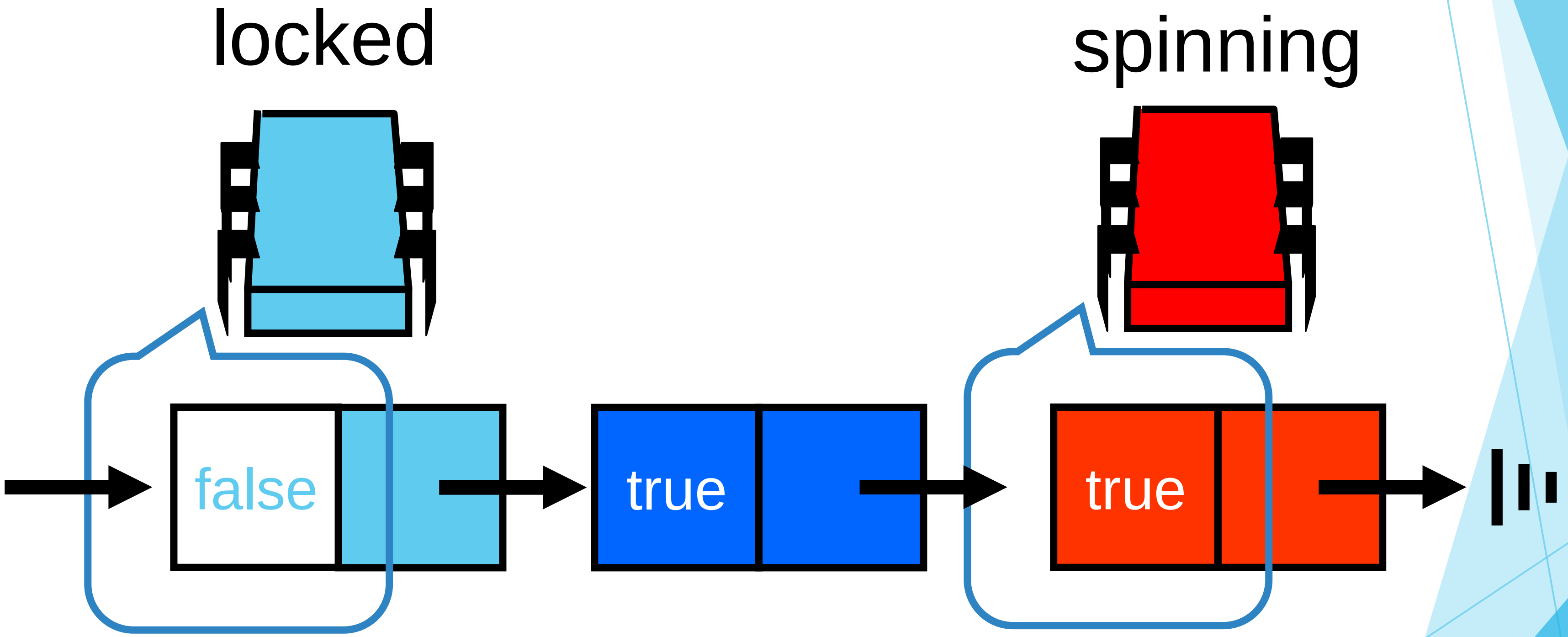
Queue Locks



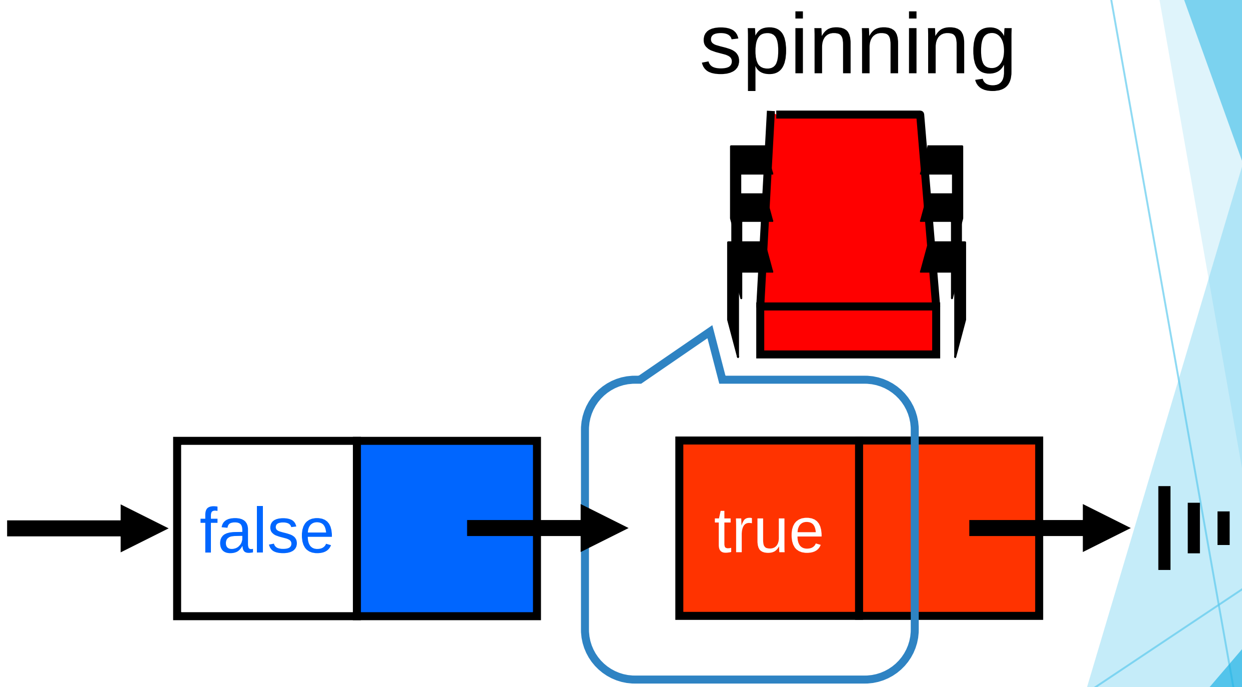
Queue Locks



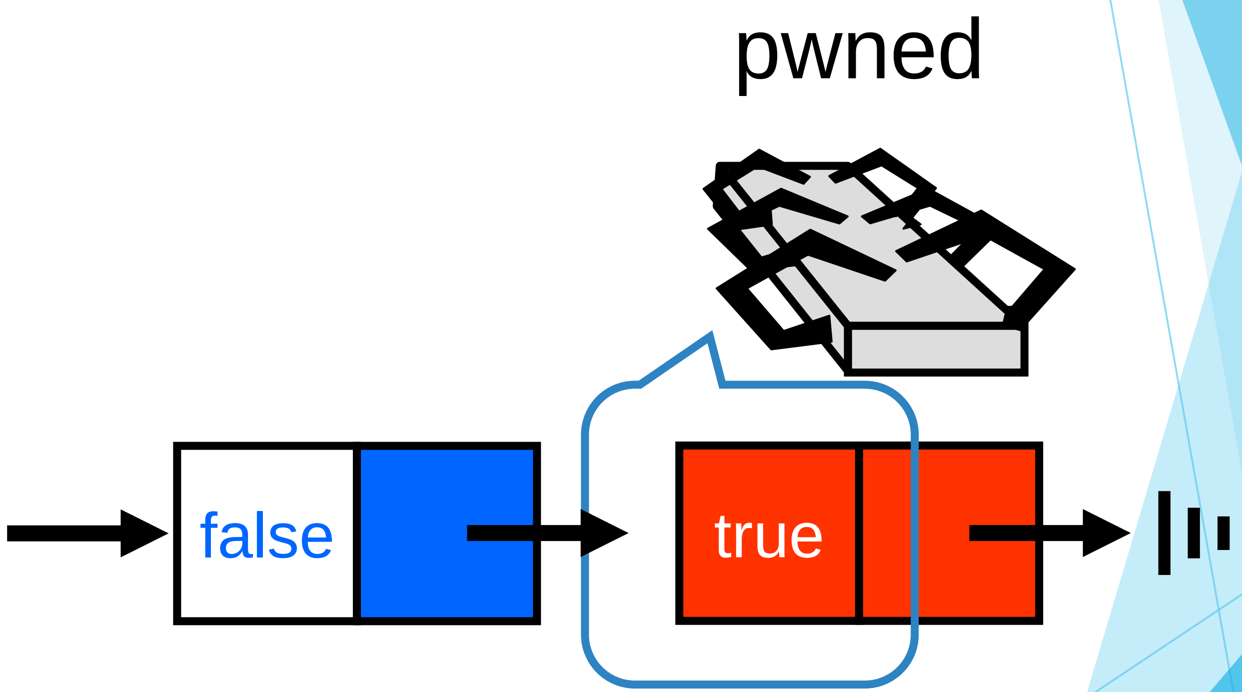
Queue Locks



Queue Locks



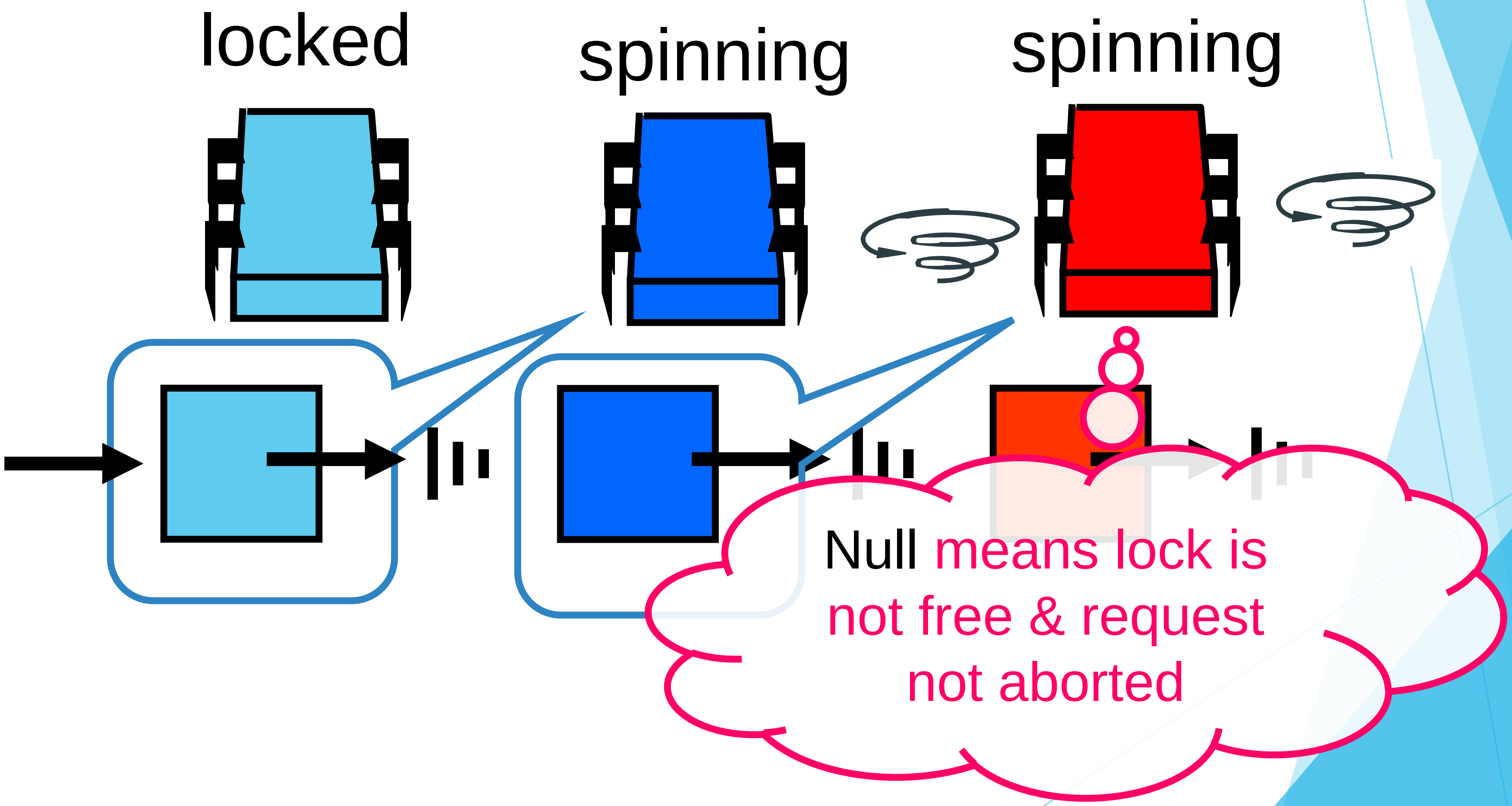
Queue Locks



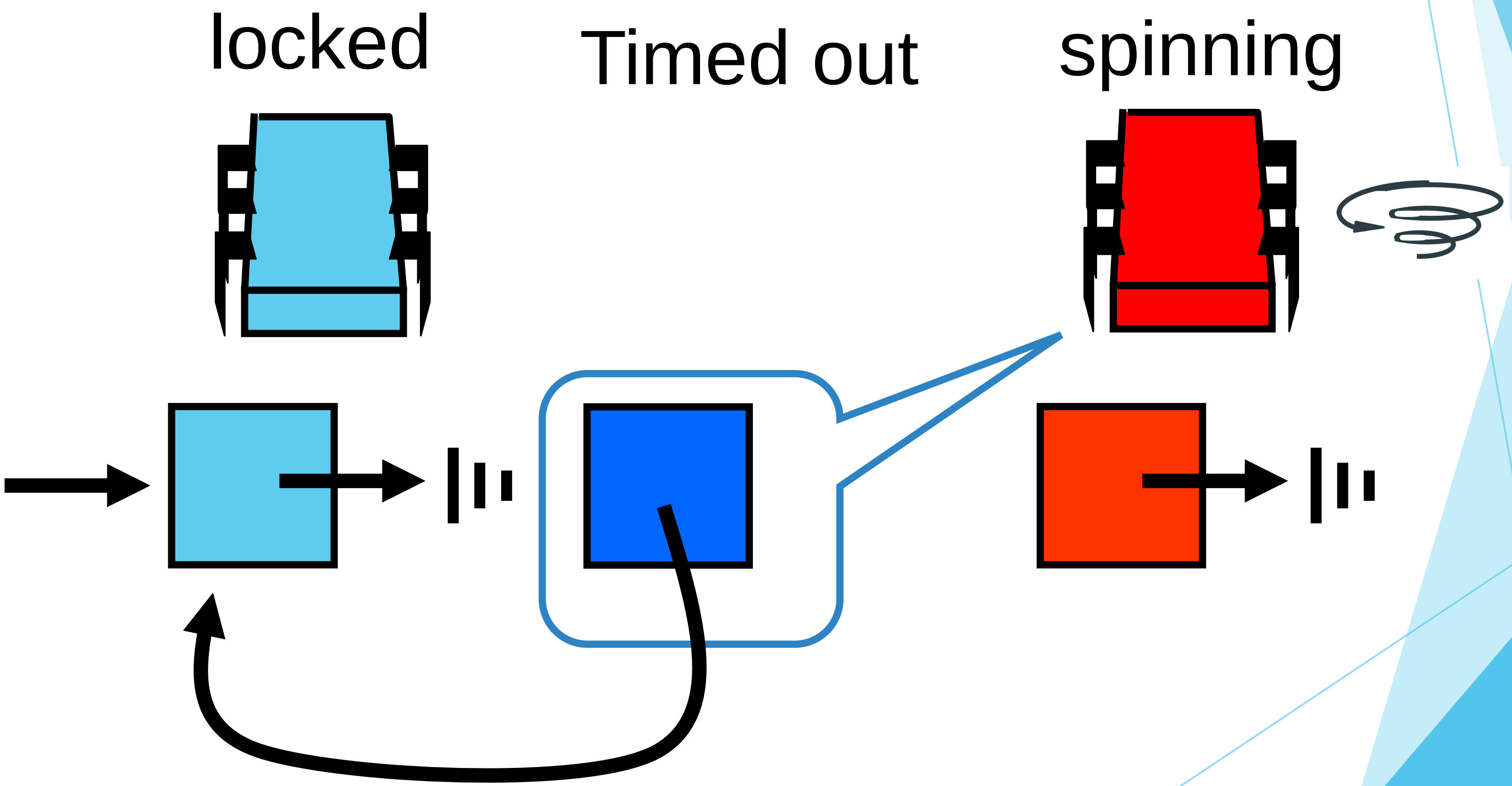
Timeout in CLH lock

- The idea here is to let the successor know about the abandoned so it will deal with it.
- The thread successor in the queue, if there is one, notices that the node on which it is spinning has been abandoned, and starts spinning on the abandoned node's predecessor.

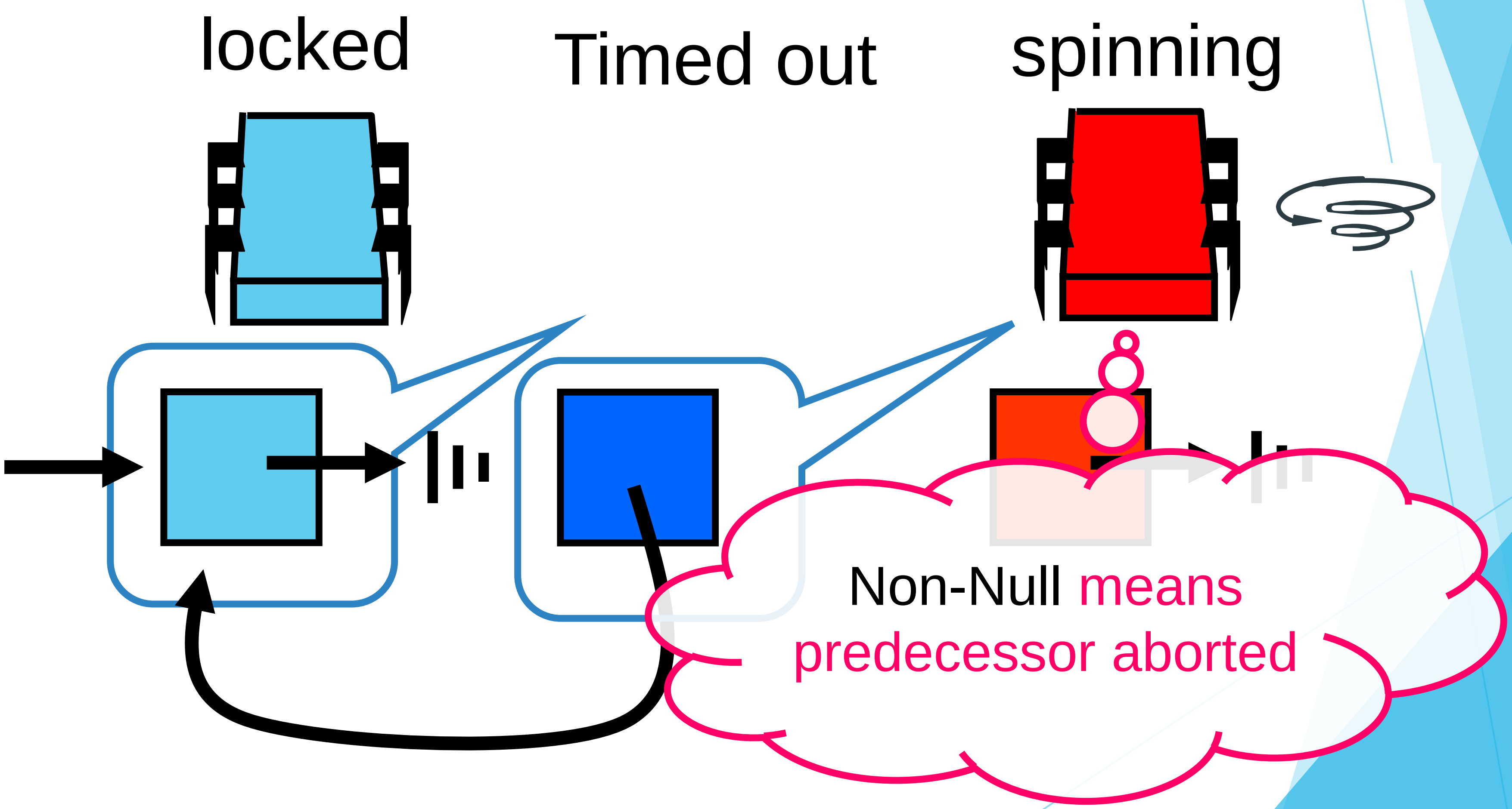
Normal Case


locked
spinning
spinning
Null means lock is
not free & request
not aborted

One Thread Aborts

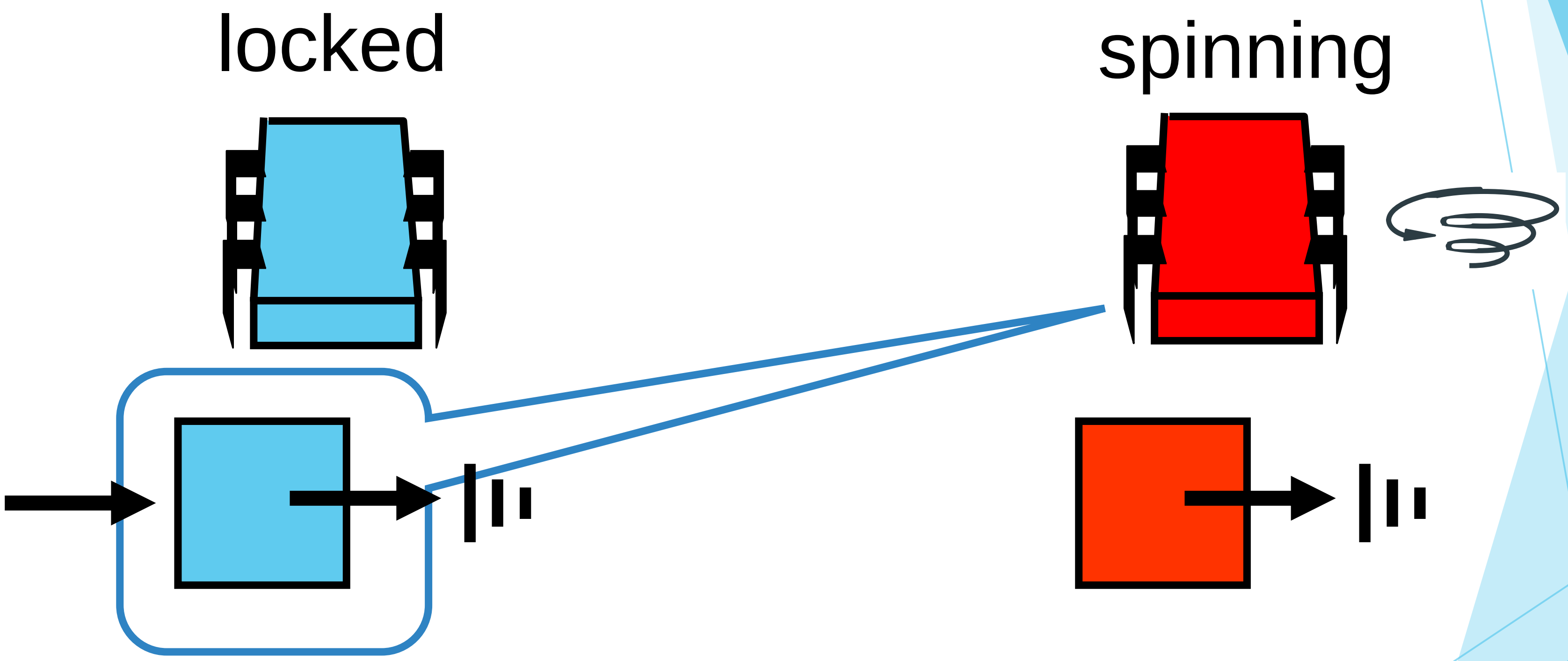


Successor Notices

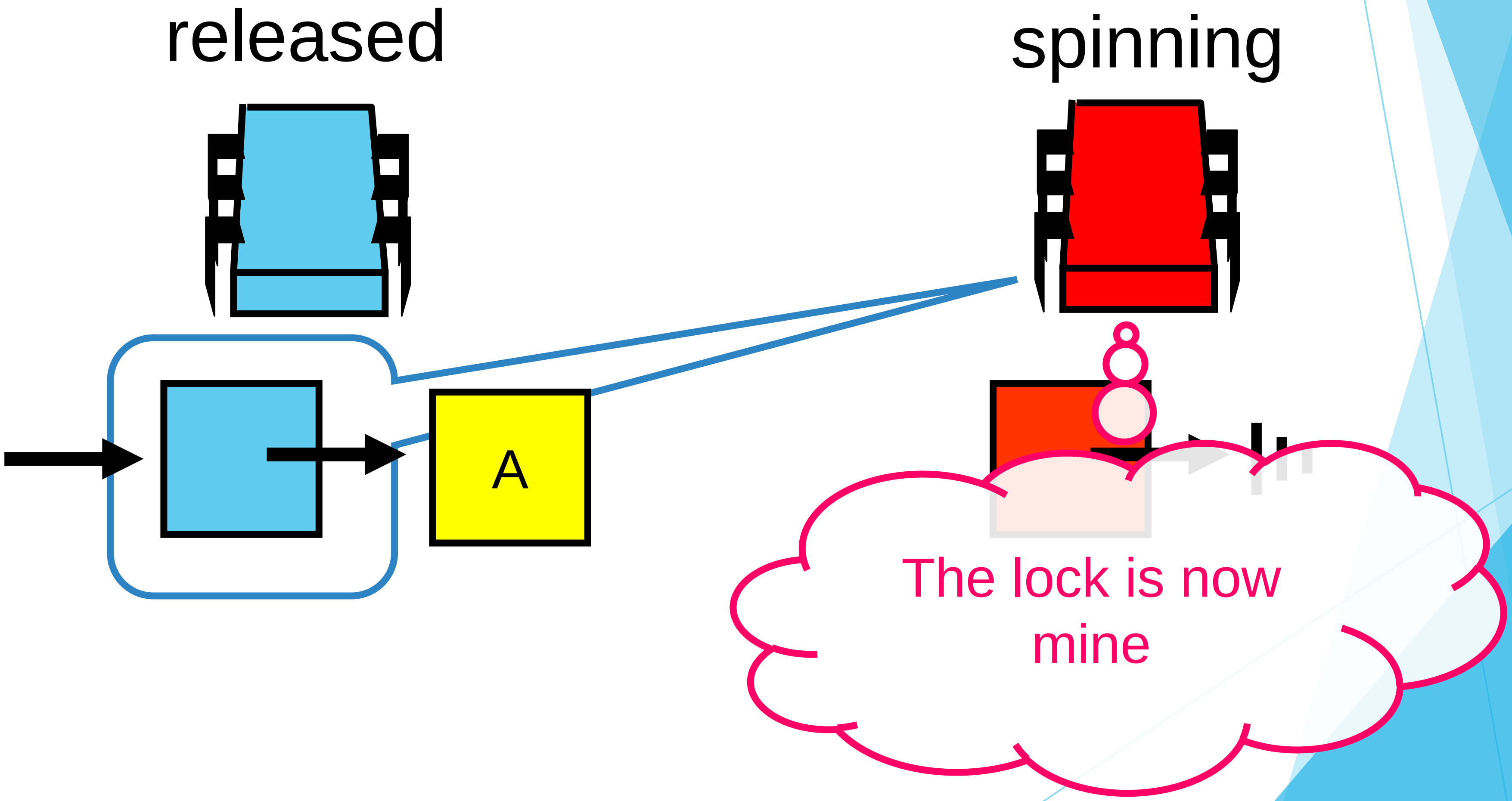


Recycle Predecessor's Node

Spin on Earlier Node



Spin on Earlier Node



Timeout in CLH lock

```
public boolean tryLock(long time, TimeUnit unit)
QNode qnode = new QNode();
myNode.set(qnode);
qnode.pred = null;
QNode myPred = tail.getAndSet(qnode);
if (myPred == null || myPred.pred == AVAILABLE) {
        return true;
```

Timeout in CLH lock

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

Timeout in CLH lock

```
if (!tail.compareAndSet(qnode, myPred))
      qnode.pred = myPred;
      Return }
```

If my time is up!

Timeout in CLH un-lock

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

Composite Lock

- consider an advanced lock algorithm that combines the best of Queue locks and backoff lock.
- The CompositeLock class keeps a short, fixed-size array of lock nodes.
- Each thread that tries to acquire the lock selects a node in the array at random.
- If: that node is in use then backoff.
- Else: The thread spins on the preceding node, and when that node's owner signals it is done, the thread enters the critical section.

Composite Lock

Pros:

- When threads back off, they access different locations, reducing contention.
- Abandoning a lock request is easy.
- For L locks and n threads, the CompositeLock class, requires only $O(L)$ space.

Cons:

- Complex code.
- No guarantee first-come-first-served.

Conclusion

- We have seen a variety of spin locks that vary in characteristics and performance.
- Each spin lock algorithm will work best in different situations.
- This depends on the application, hardware and which properties are important.

Thank you for listening!