Linked Lists

Thomas Schwarz, SJ

Locks

- Spin locks:
  - Retry over and over again
- Blocking locks
  - Instead of spinning, go to sleep
  - Mechanism to

Locks

- Peterson lock
    - Starvation-free mutual exclusion

```
class Peterson implements Lock {
private boolean[] flag = new boolean[2]; private int victim;
public void lock() {
int i = ThreadID.get(); // either 0 or 1 int j = 1-i;
flag[i] = true;
victim = i;
while (flag[j] && victim == i) {}; // spin }
}
```

Locks

- We have shown that the Peterson lock works
  - **But not in practice!**
  - Multiprocessor do **not**
    - provide sequentially consistent memory
    - guarantee program order among read-writes of a single thread
  - Why?:
    - Compilers, Hardware (write buffer)

Locks

- To prevent re-ordering:
  - Use special memory barrier instruction
    - A.K.A. memory fence
  - Can fix Peterson's algorithm by a barrier before each read
  - Memory fences are expensive:
    - About same as a Compare-And-Set instruction

Test and Set Locks

- Test And Set operation:
  - Consensus number 2
  - Used in early multiprocessor architectures
- Test-And-Set (TAS) operates on a single memory word / byte holding a binary value
  - Atomically swaps the value true for the current value
  - Return value tells if prior value was true or false

Test and Set Locks in CPP

```
std::atomic_flag lock_stream = ATOMIC_FLAG_INIT;
std::stringstream stream;

void append_number(int x) {
  while (lock_stream.test_and_set()) {}
  stream << "thread #" << x << '\n';
  lock_stream.clear();
}

int main ()
{
  std::vector<std::thread> threads;
  for (int i=1; i<=10; ++i)
threads.push_back(std::thread(append_number,i));
  for (auto& th : threads) th.join();

  std::cout << stream.str();
  return 0;
}
```

Test and Set Locks in Java

```
public class AtomicBoolean {
 boolean value;

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

Test and Set Locks

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

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 Locks

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

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

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

Test and Set Locks

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

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

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

Lock state is atomic boolean

Test and Set Locks

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

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

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

Keep trying until lock is acquired

Test and Set Locks

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

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

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

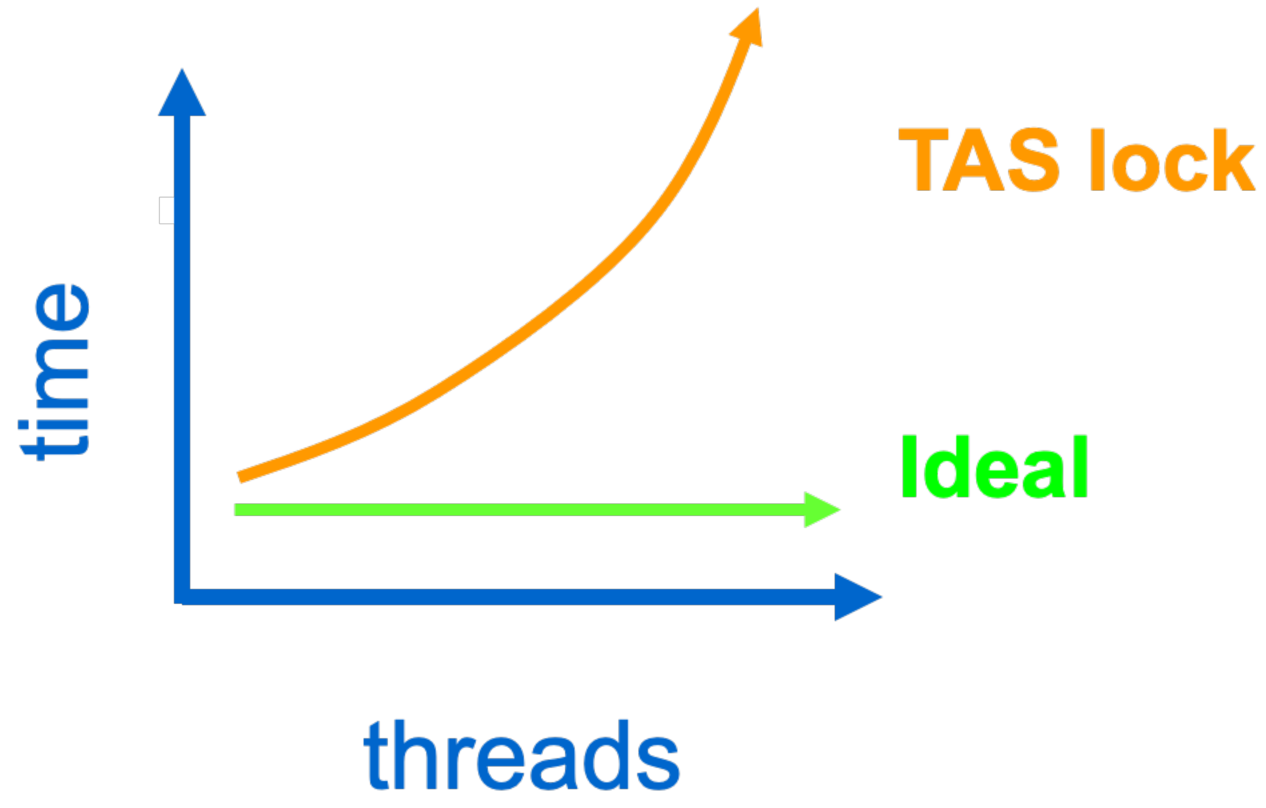
Release lock by
resetting state to false

Test and Set Locks

- TAS is a spin-lock
- Uses constant space

Test and Set Locks

- Performance:



- Each processor needs to invalidate cache lines if the value of the lock changes

Test and Test and Set Locks

- Lurking state:
  - 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 fails, go back to lurking

Test and Test and Set Locks

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

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
 }
}
```

Lurking

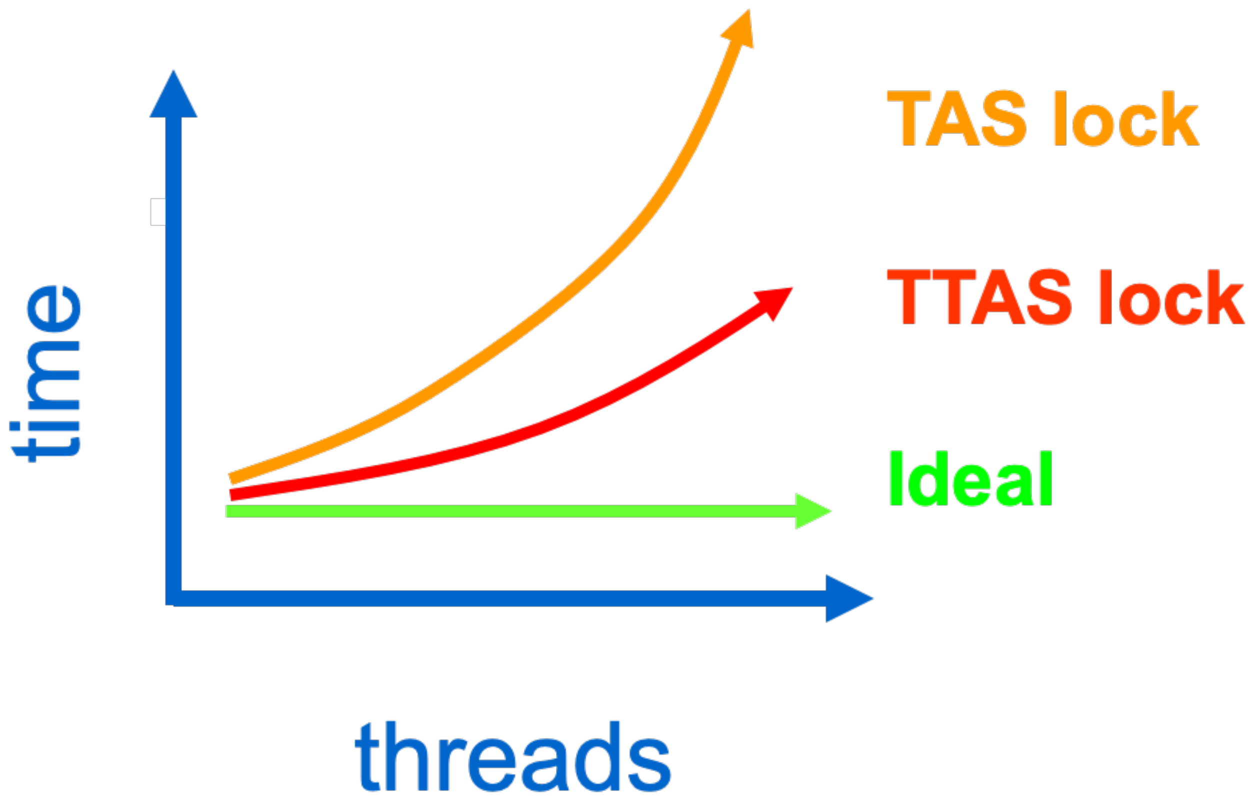
Test and Test and Set Locks

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

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
 }
}
```

Pouncing

Test and Test and Set Locks



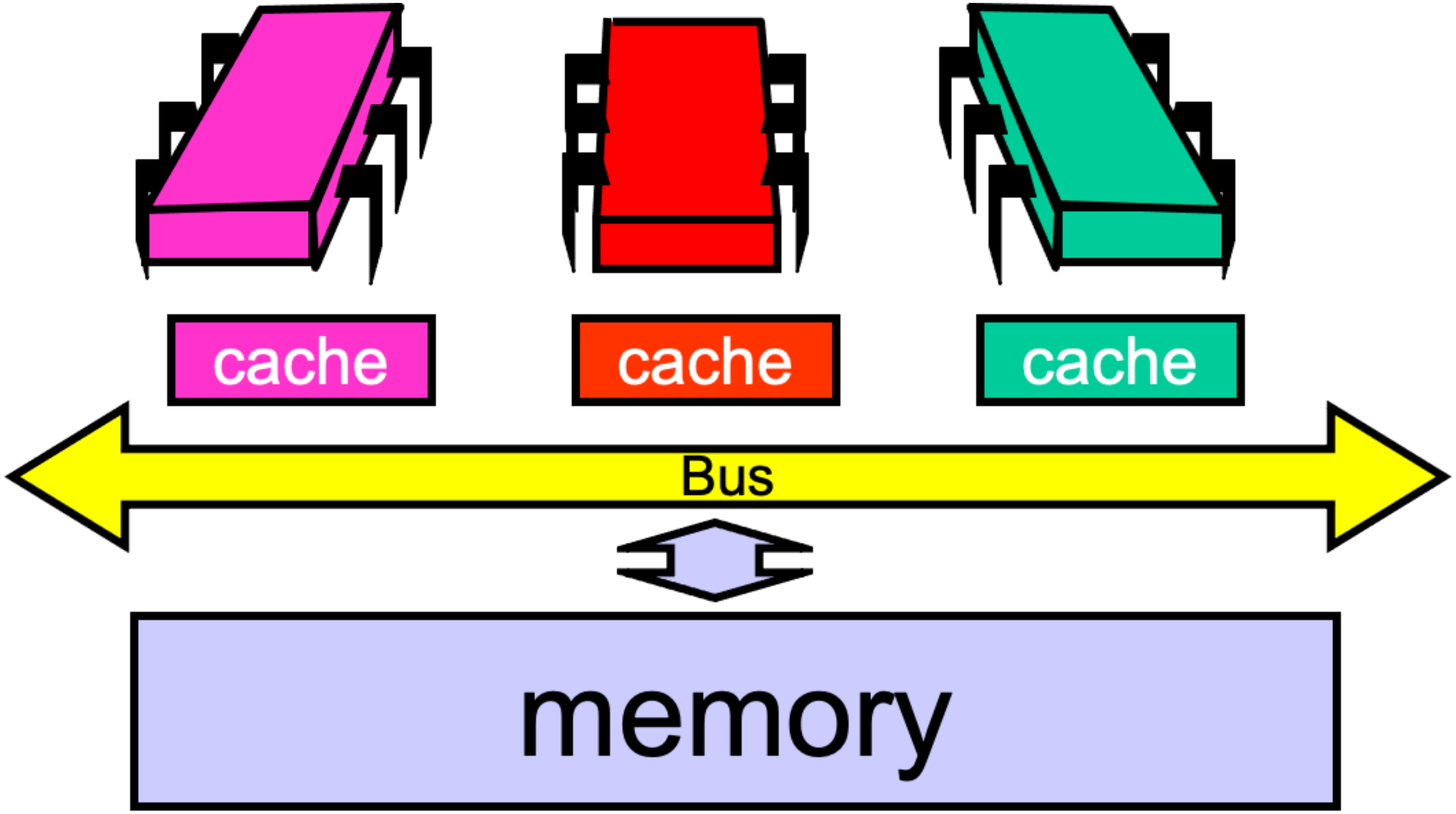
Test and Test and Set Locks

- state.get( ) does not interfere with other processors

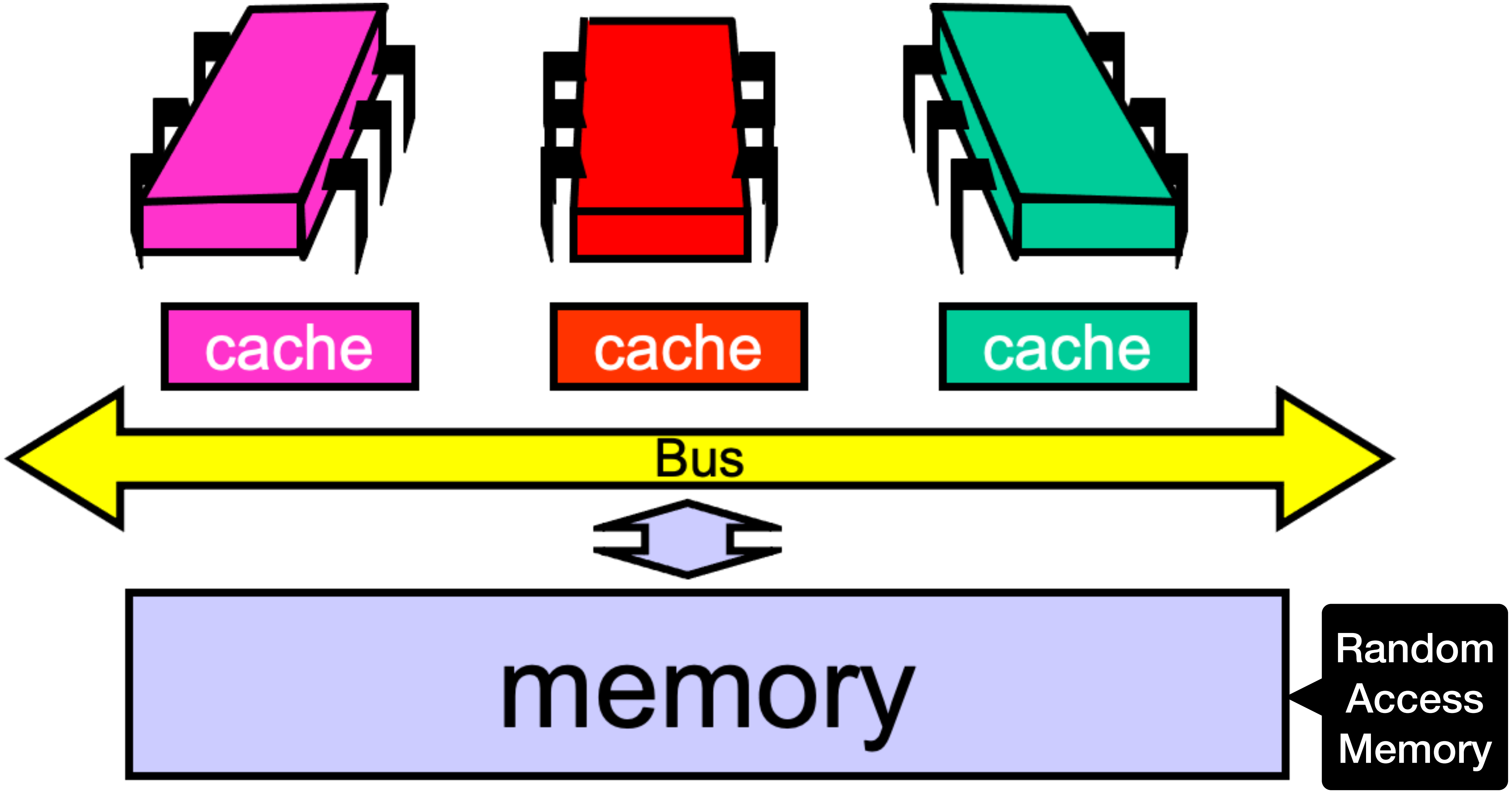
```
class TTASlock {
 AtomicBoolean state = new AtomicBoolean(false);

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
 }
}
```

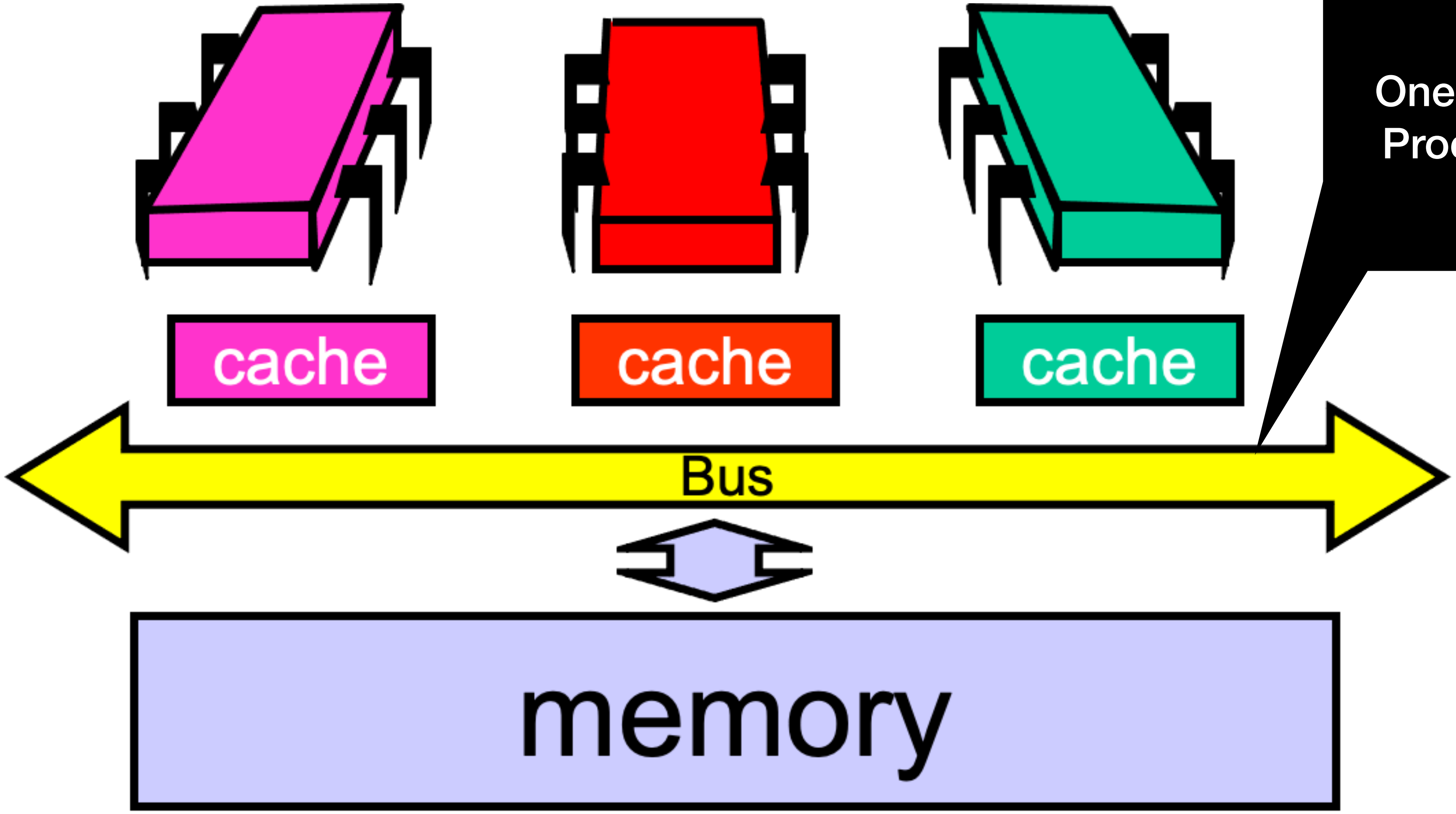
Test and Test and Set Locks



Test and Test and Set Locks

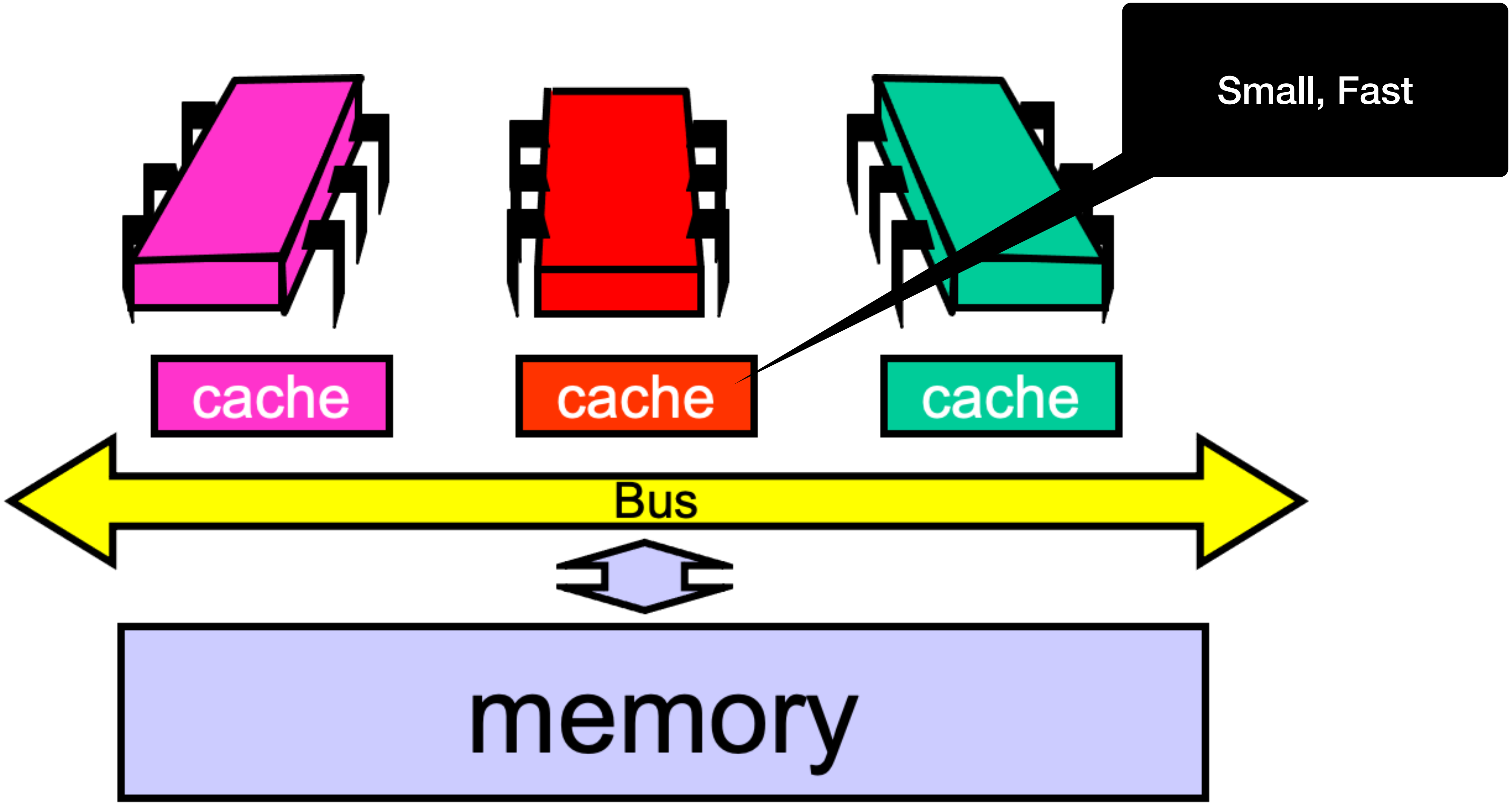


Test and Test and Set Locks



Shared Bus
One processor at a time
Processor and memory
snoop

Test and Test and Set Locks



Test and Test and Set Locks

- Granularity
  - Caches operate with Cache Lines
    - fixed sized: 64B or 128B

Test and Test and Set Locks

- Cache coherence protocols
    - What happens if A updates something also kept in B's cache?
    -

Test and Test and Set Locks

- Cache contents are in a state of MESI
- MESI
  - Modified: Have modified cached data, must write back to memory
  - Exclusive: Not modified, I have only copy
  - Shared: Not modified, may be cached elsewhere
  - Invalid: Cache contents not meaningful

TTAS Exponential Back-off

- Contention:
  - Multiple nodes try to acquire a lock at the same time
- Observation:
  - If in TTAS:
    - Lock is free
    - But TaS fails:
      - Probably high contention

TTAS Exponential Back-off

- Backoff:
  - Thread sleeps for some time
- Exponential backoff:
  - Known from network protocols to deal with channel contention
  - Increase backoff time by doubling up to a maximum delay
- In practice: Improves performance but improvement depends on the minimum and maximum delay times

TTAS Exponential Back-off

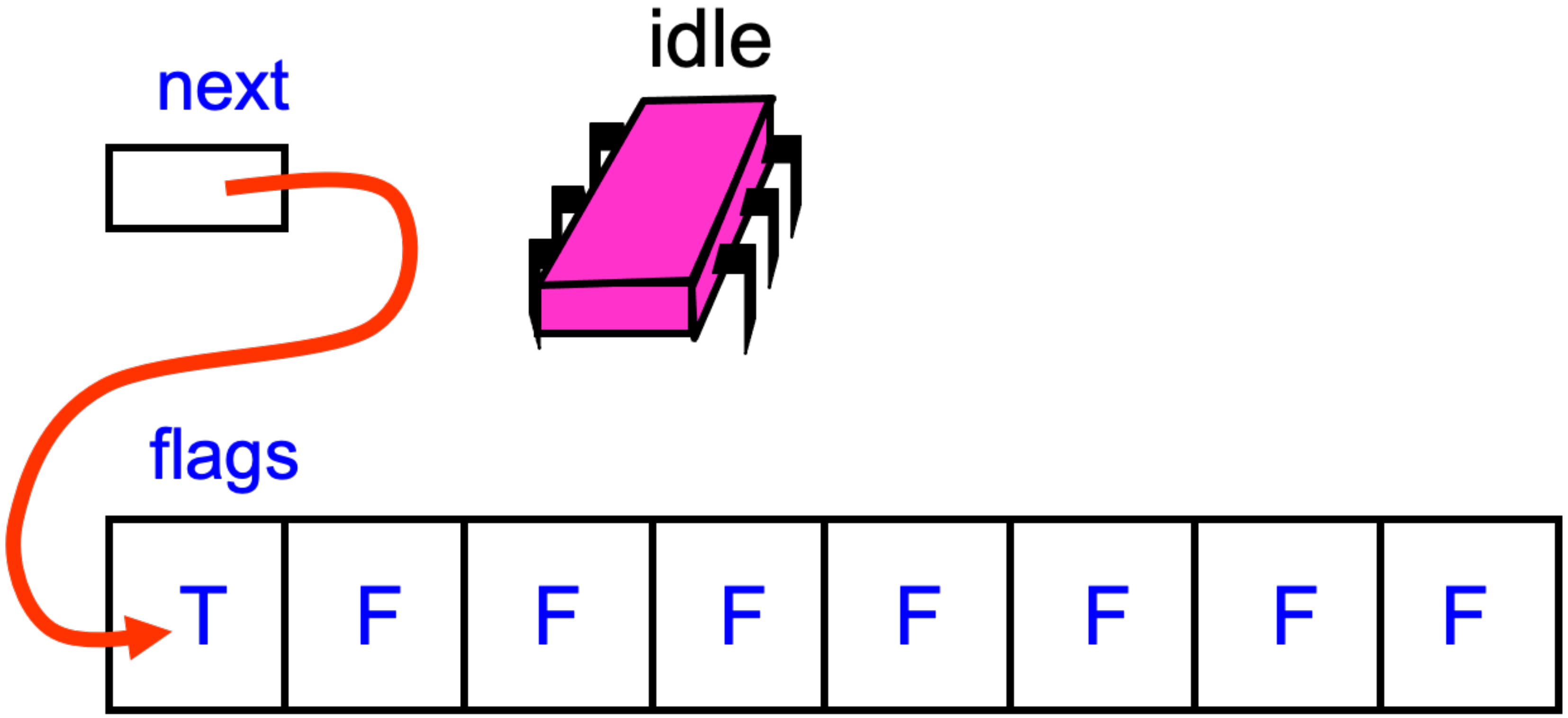
- Still causes cache coherence traffic:
  - All threads spin on the same shared location
- Underuses critical section:
  - Threads can delay longer than necessary
  - Critical section is then under-utilized

Queue Locks

- Let threads form a queue
  - Each thread can learn of its turn by checking predecessor
- Causes:
  - Less cache-coherence traffic as threads spin on different locations
  - Critical section better utilized
  - Provides first-come-first-served fairness

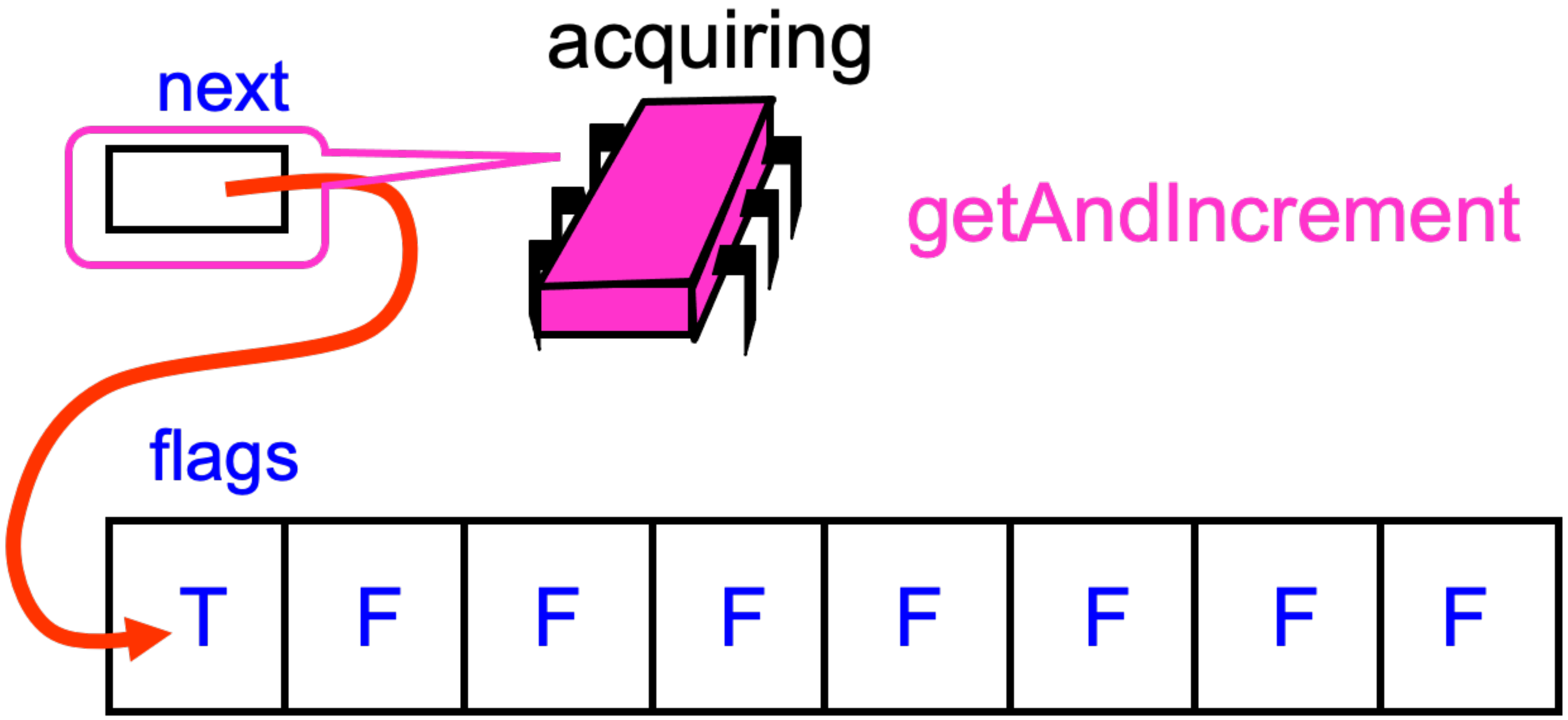
Array Based Lock

- Anderson Queue Lock

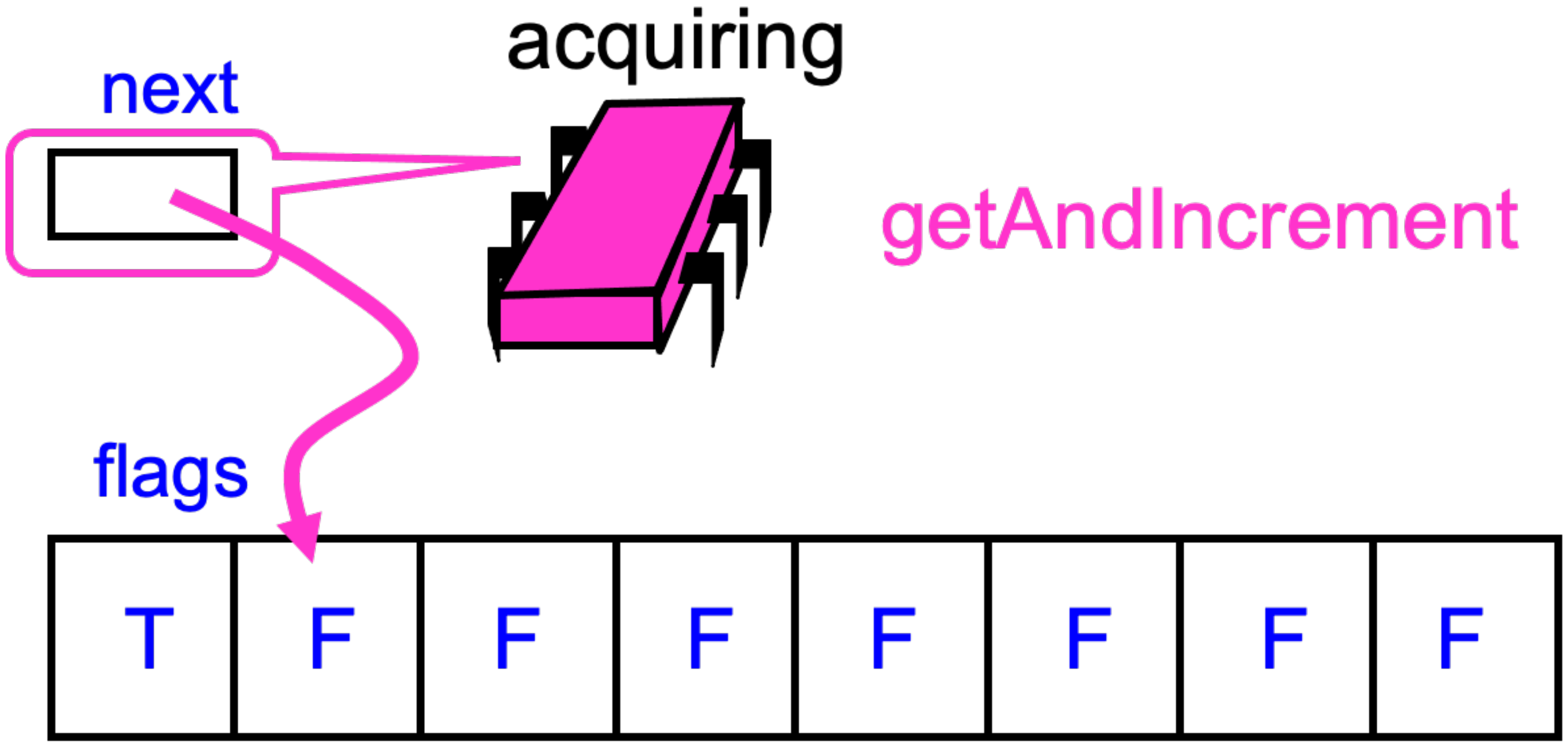


Array Based Lock

- To acquire the lock:
  - Each thread atomically increments the tail field
  - This gives it its slot



Array Based Lock



acquiring

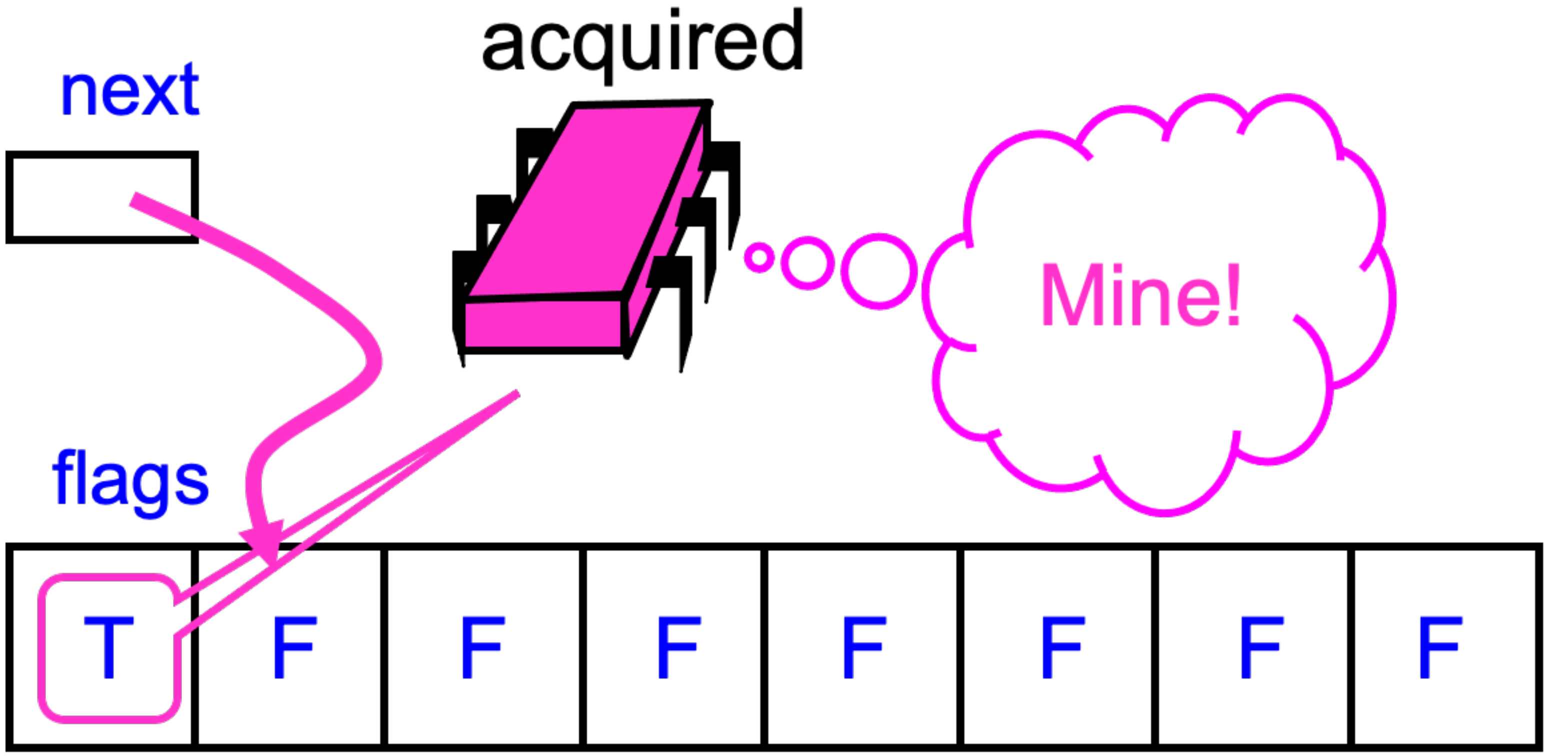
next

getAndIncrement

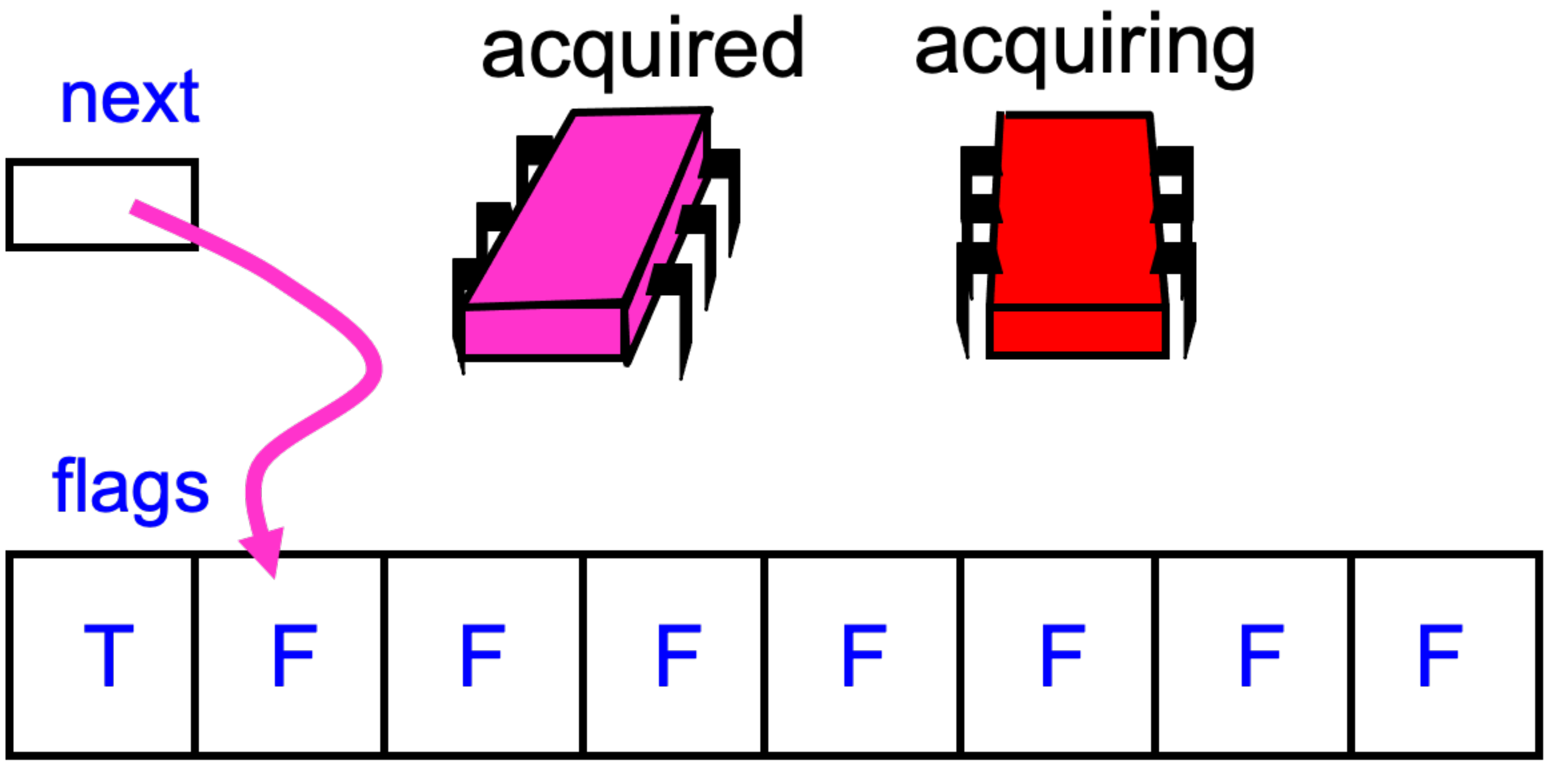
flags

T	F	F	F	F	F	F	F

Array Based Lock

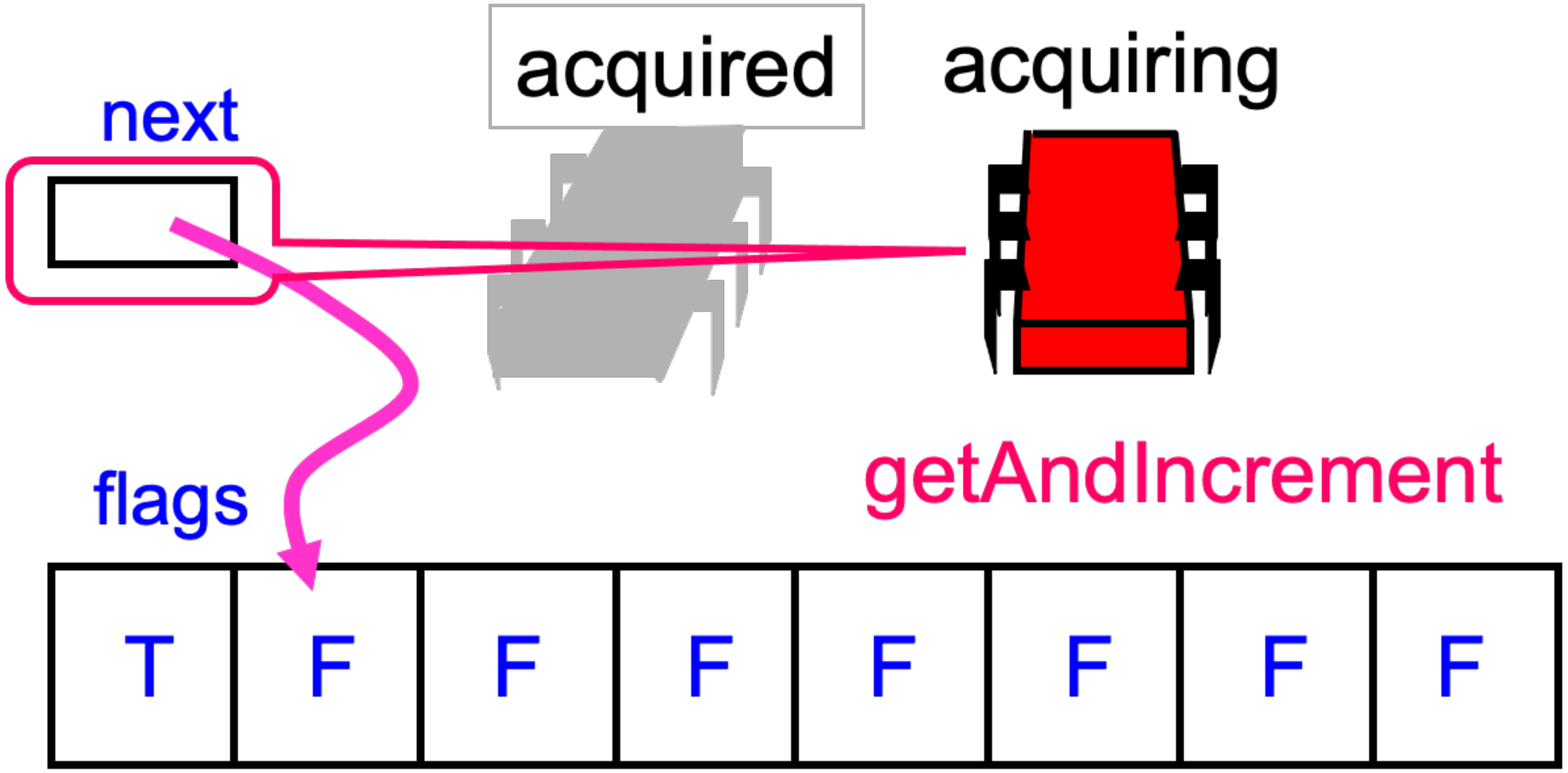


Array Based Lock

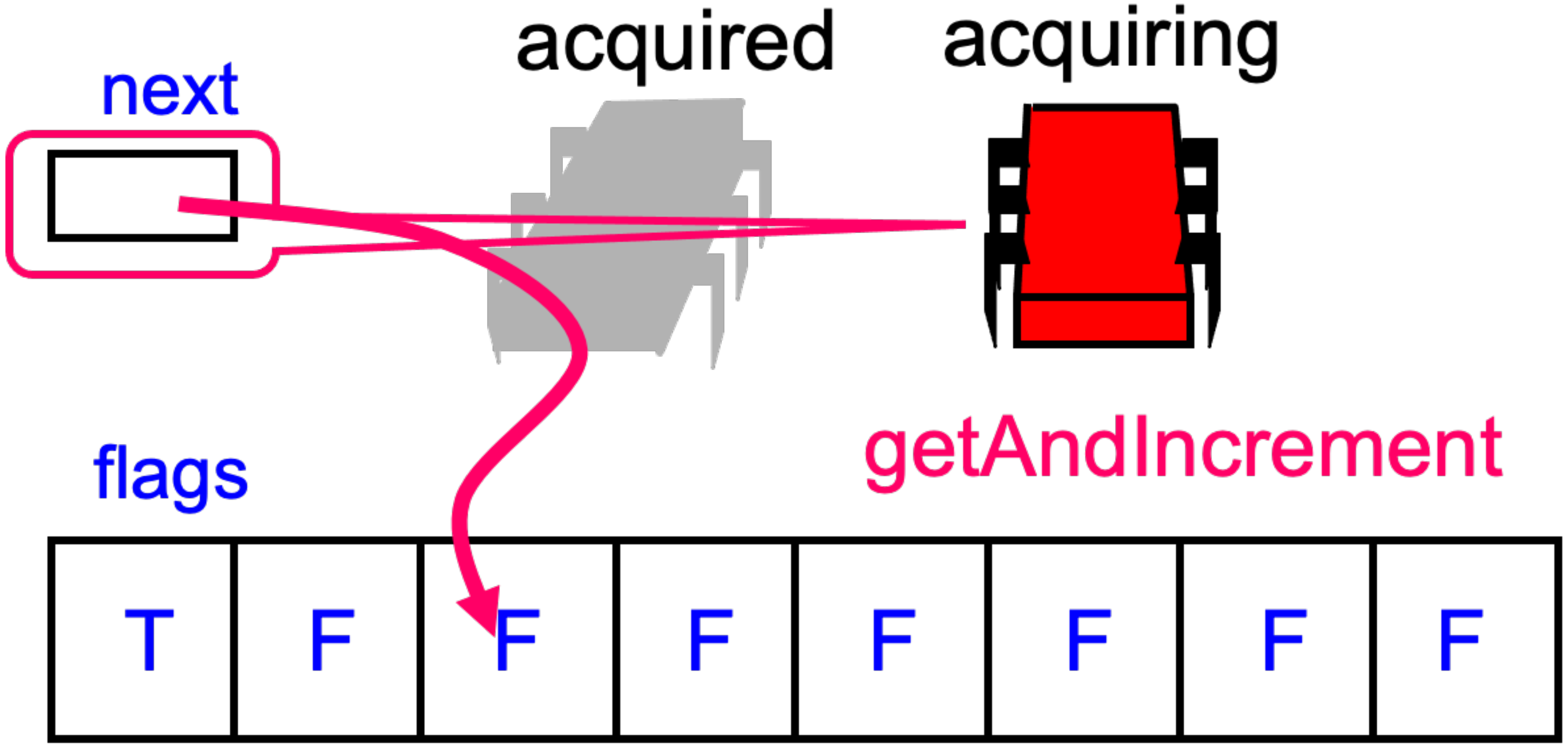


`Another thread wants to acquire the lock`

Array Based Lock

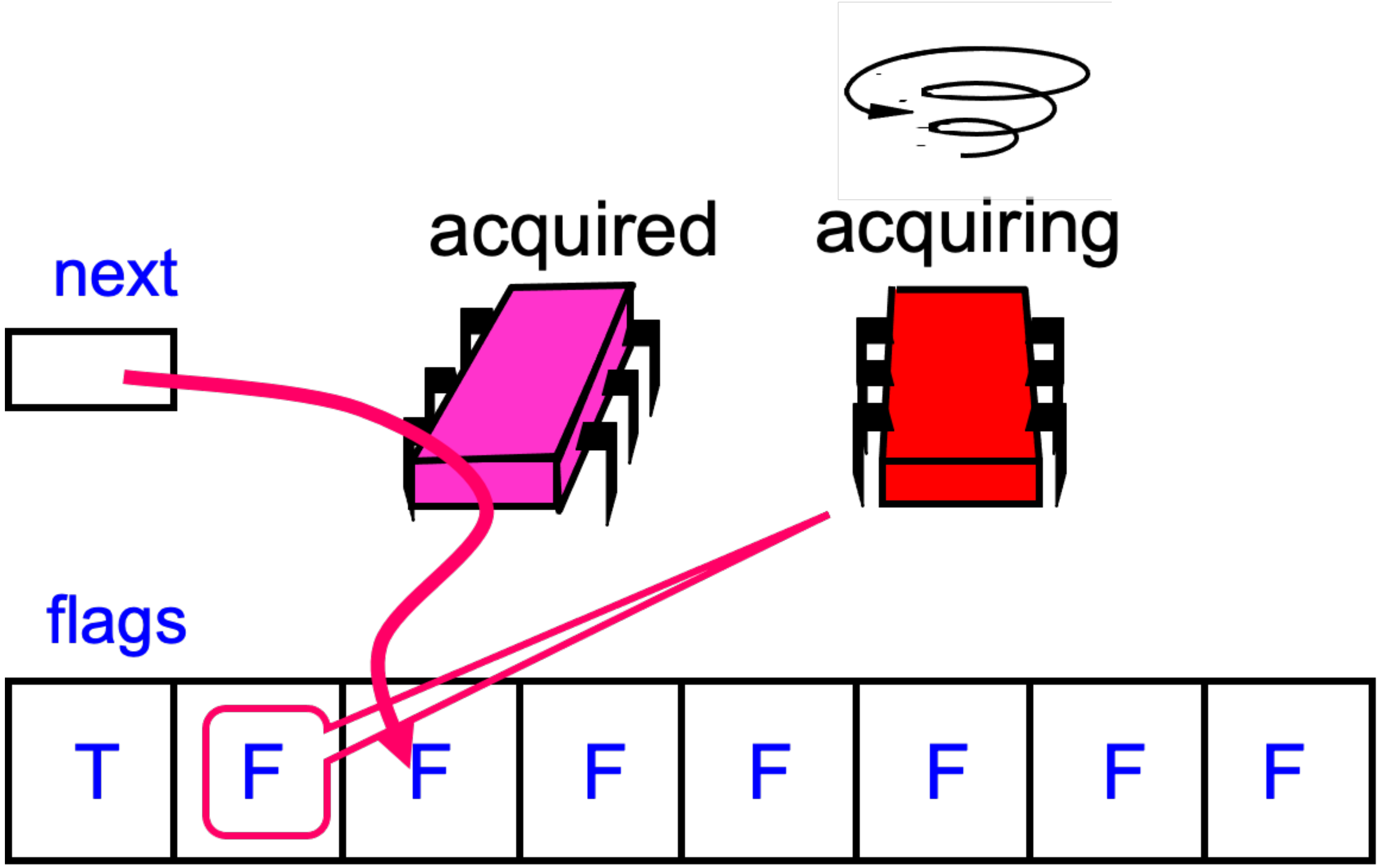


Array Based Lock



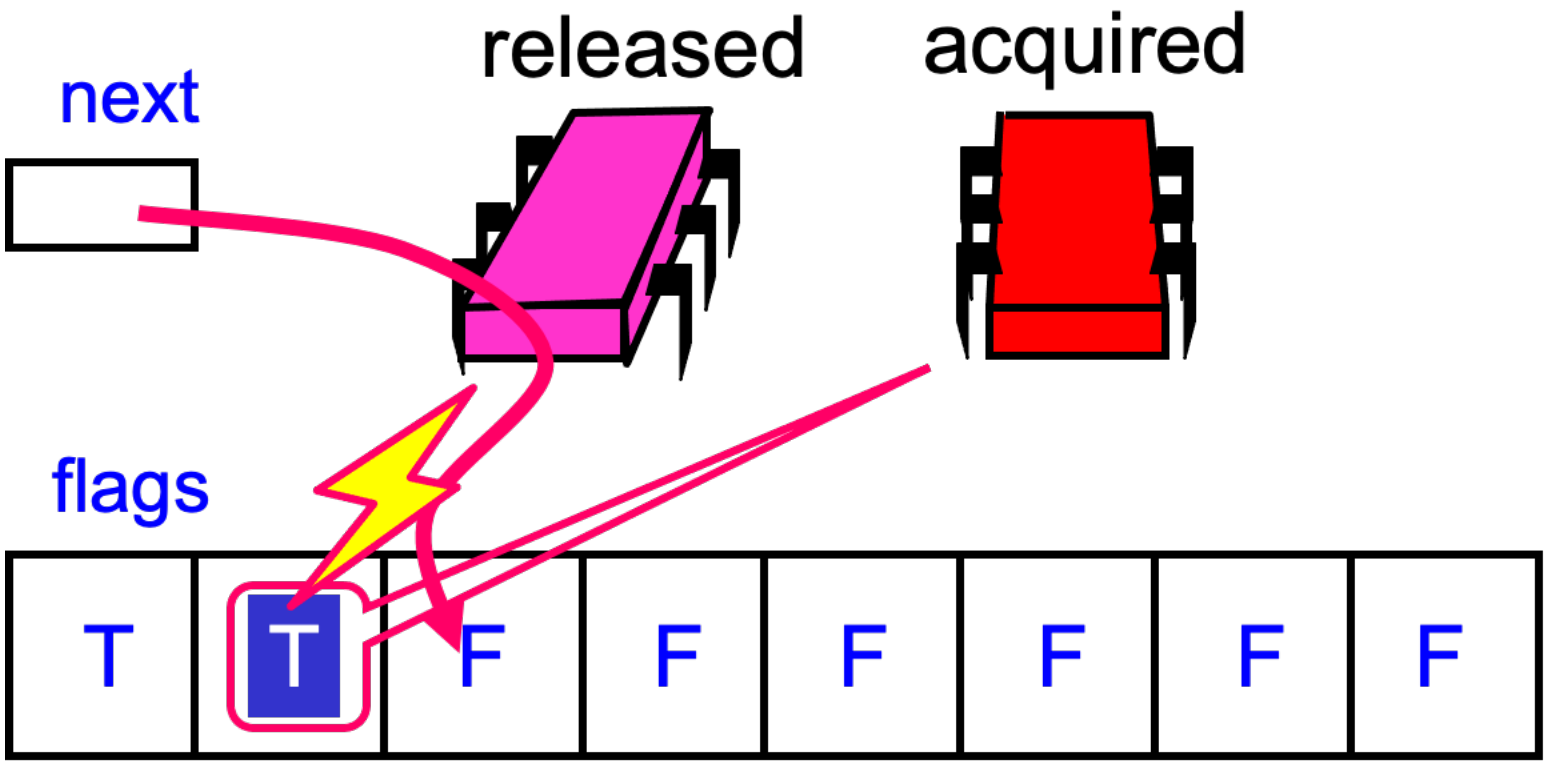
Advances the next-pointer to acquire its own slot

Array Based Lock



Spins until the flag variable at that slot becomes true

Array Based Lock



The first thread now releases the lock by setting the next slot to true

Array Based Lock

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

One slot per thread

Array Based Lock

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

Next flag to use

Array Based Lock

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

This is a threadlocal variable

Array Based Lock

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

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

Claim a slot by atomically incrementing the next field

Array Based Lock

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

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

Wait until predecessor has released the lock

Array Based Lock

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

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

Spin until predecessor has released the lock

Array Based Lock

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

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

Prepare slot for reuse

Array Based Lock

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

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

To release slot, set the slot after mine to True

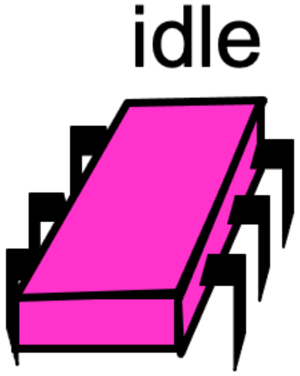
Array Based Lock

- To avoid cache-coherence traffic:
  - Pad the array fields so that each array element is in its own cache-line
- Performance is then essentially flat!
- But:
  - Uses up a lot of space
  - Needs to know the maximum number of threads

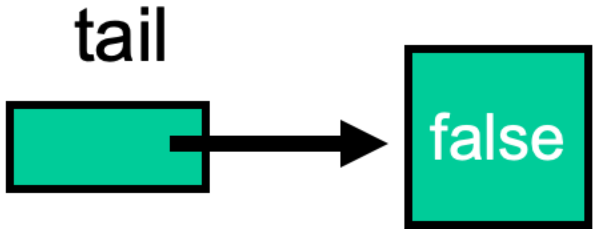
CLH Lock

- Craig, Hagersten, Landin
  - Still FCFS
  - Small, constant-size overhead per thread

CLH Lock

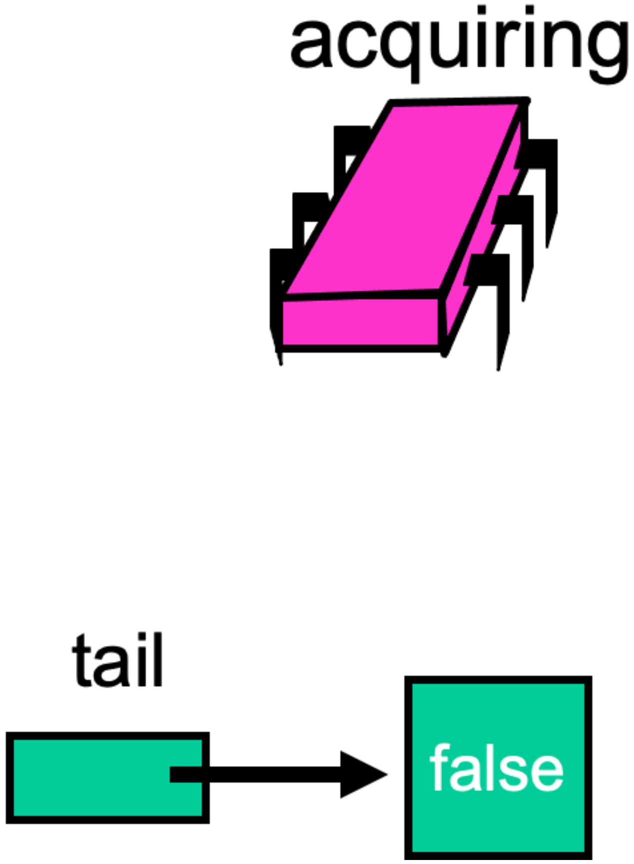


- Thread status recorded in a QNode object
  - Contains Boolean locked field
    - Field true:
      - Thread has acquired the lock
      - Or: is waiting for the lock
    - Field false
      - Thread has released the lock
- Lock is a virtual linked list of QNode objects
  - List is virtual: Each thread points to its predecessor through a thread-local pred-variable
  - Public tail points to the last node in the queue



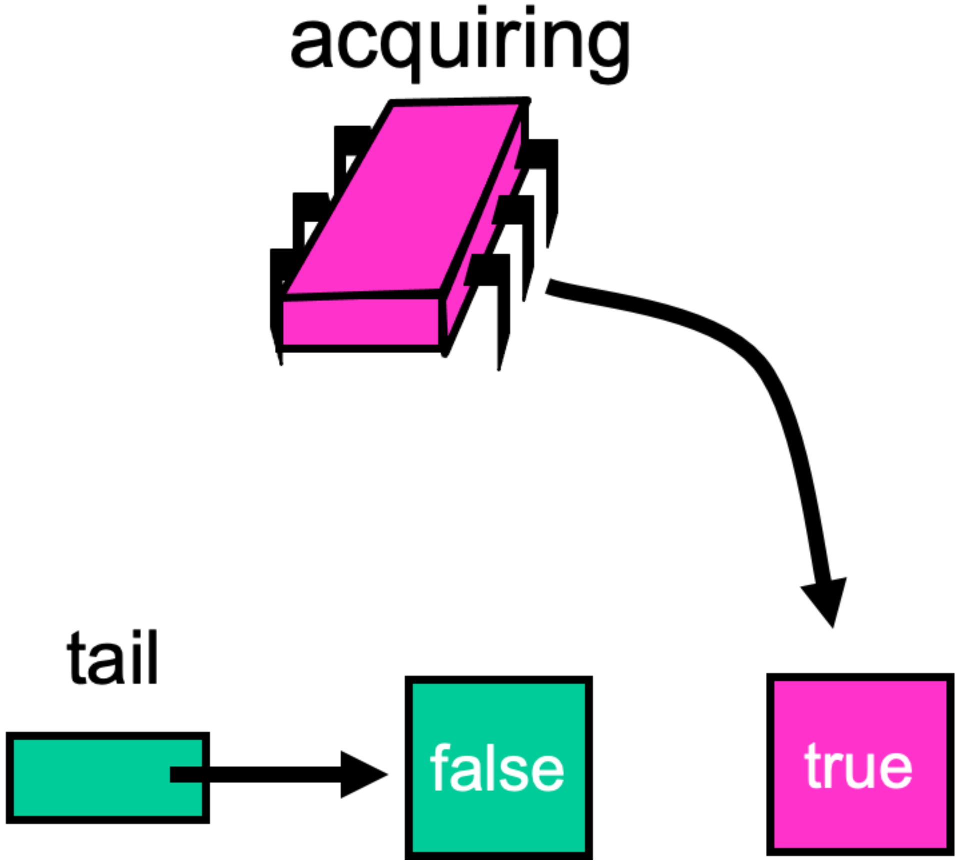
CLH Lock

- Thread wants to acquire the lock



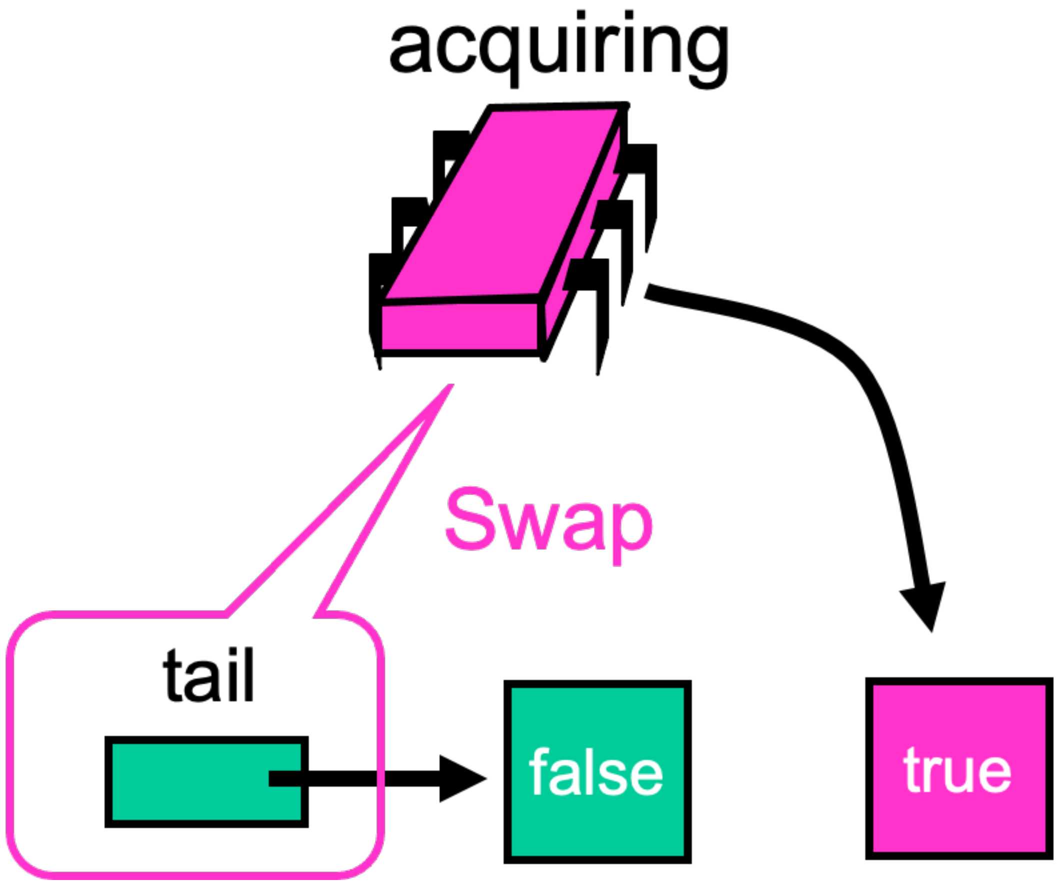
CLH Lock

- Thread sets the locked field of its QNode to true



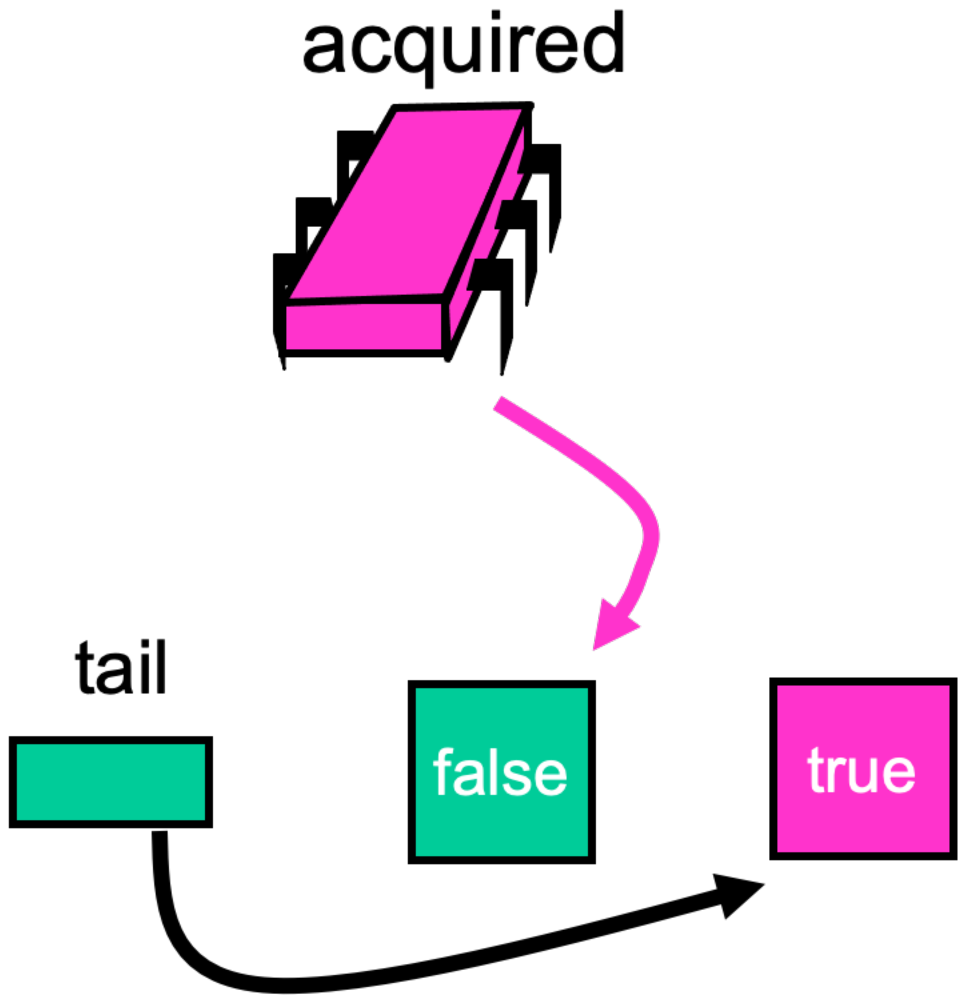
CLH Lock

- Thread applies Swap to the tail:
    - Makes its own node the tail of the queue
    - Acquires a reference to its predecessor's QNode



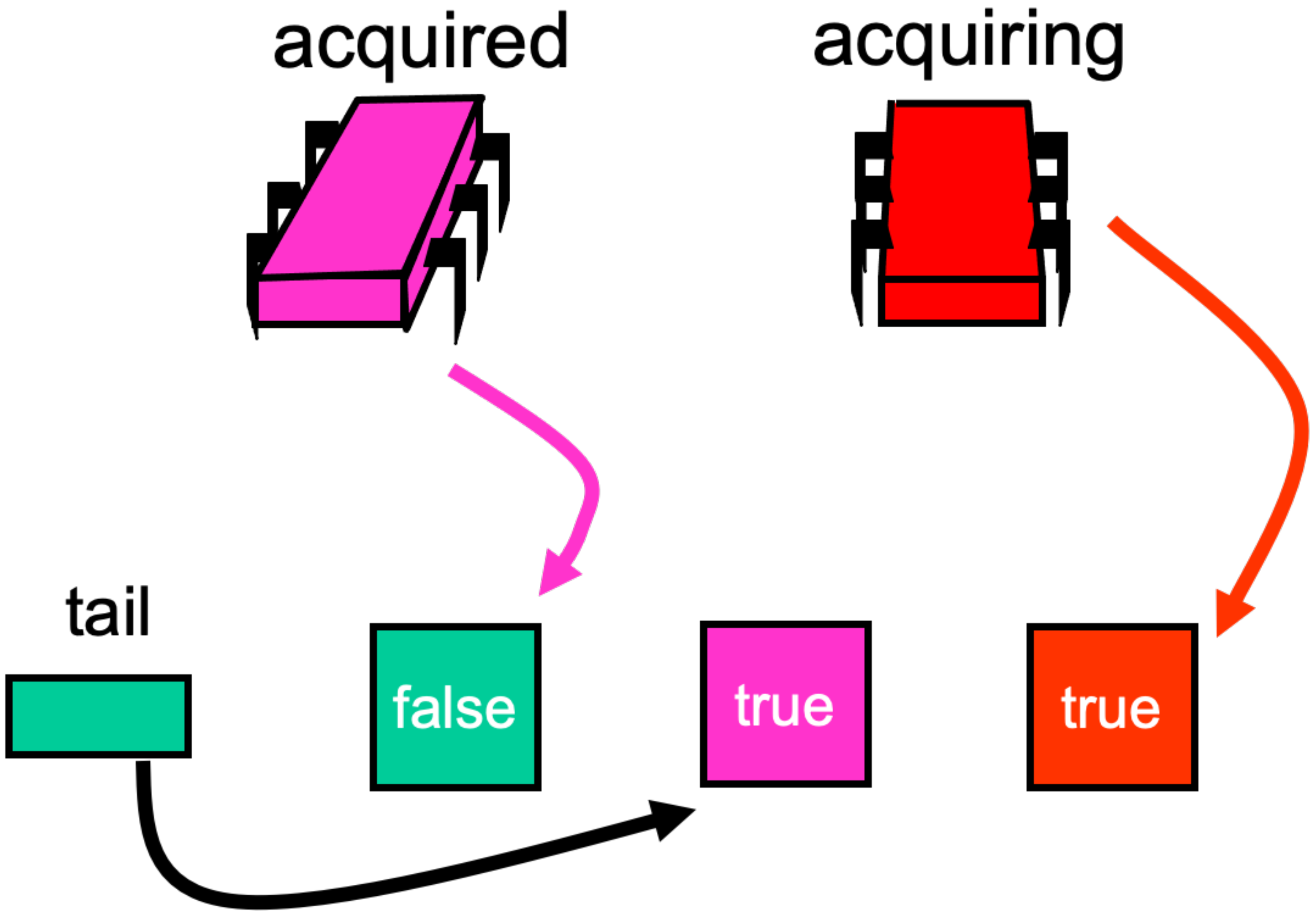
CLH Lock

- Because the predecessor's QNode is false, this thread now has the lock

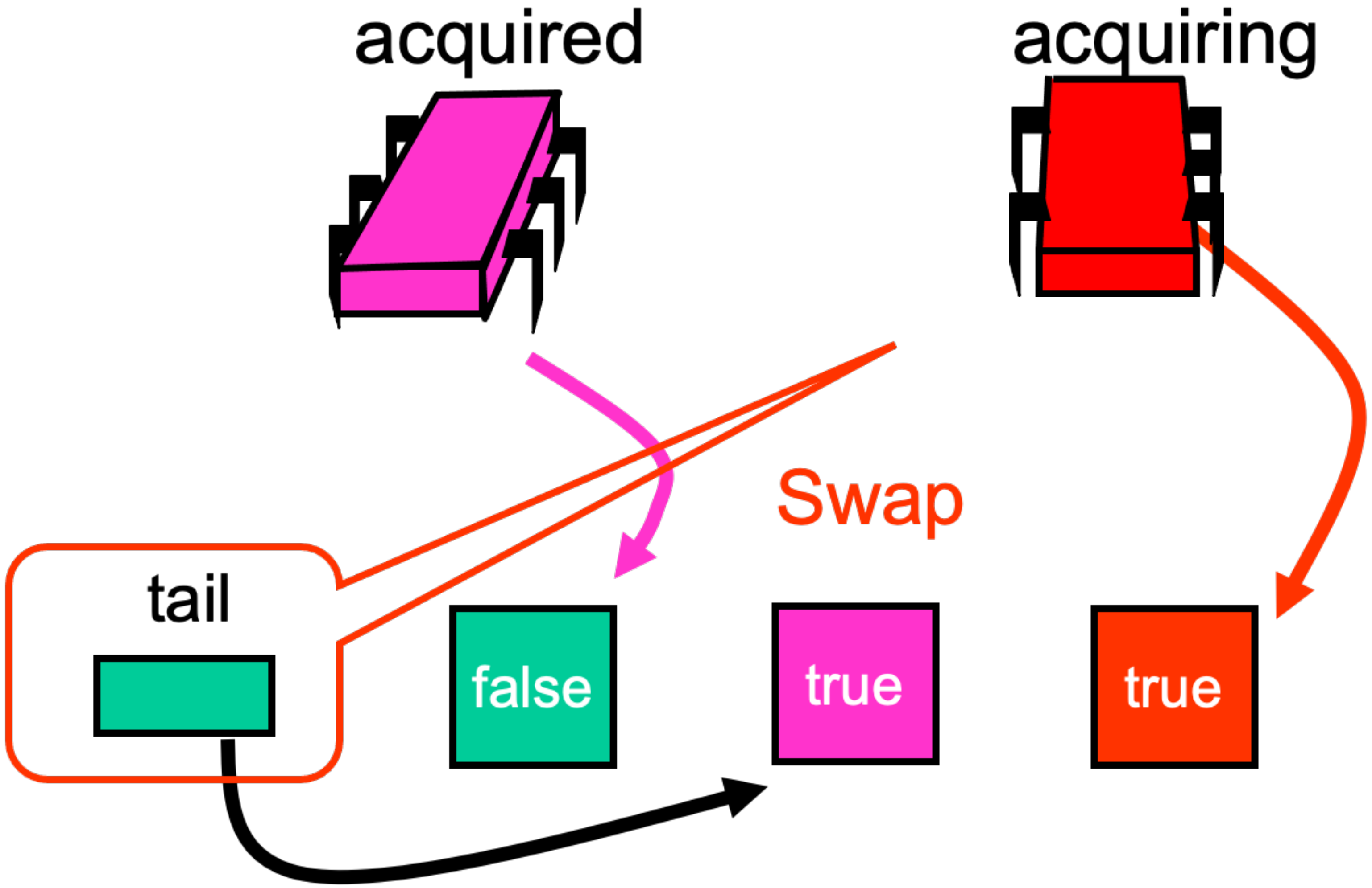


CLH Lock

- Another thread wants the lock does the same

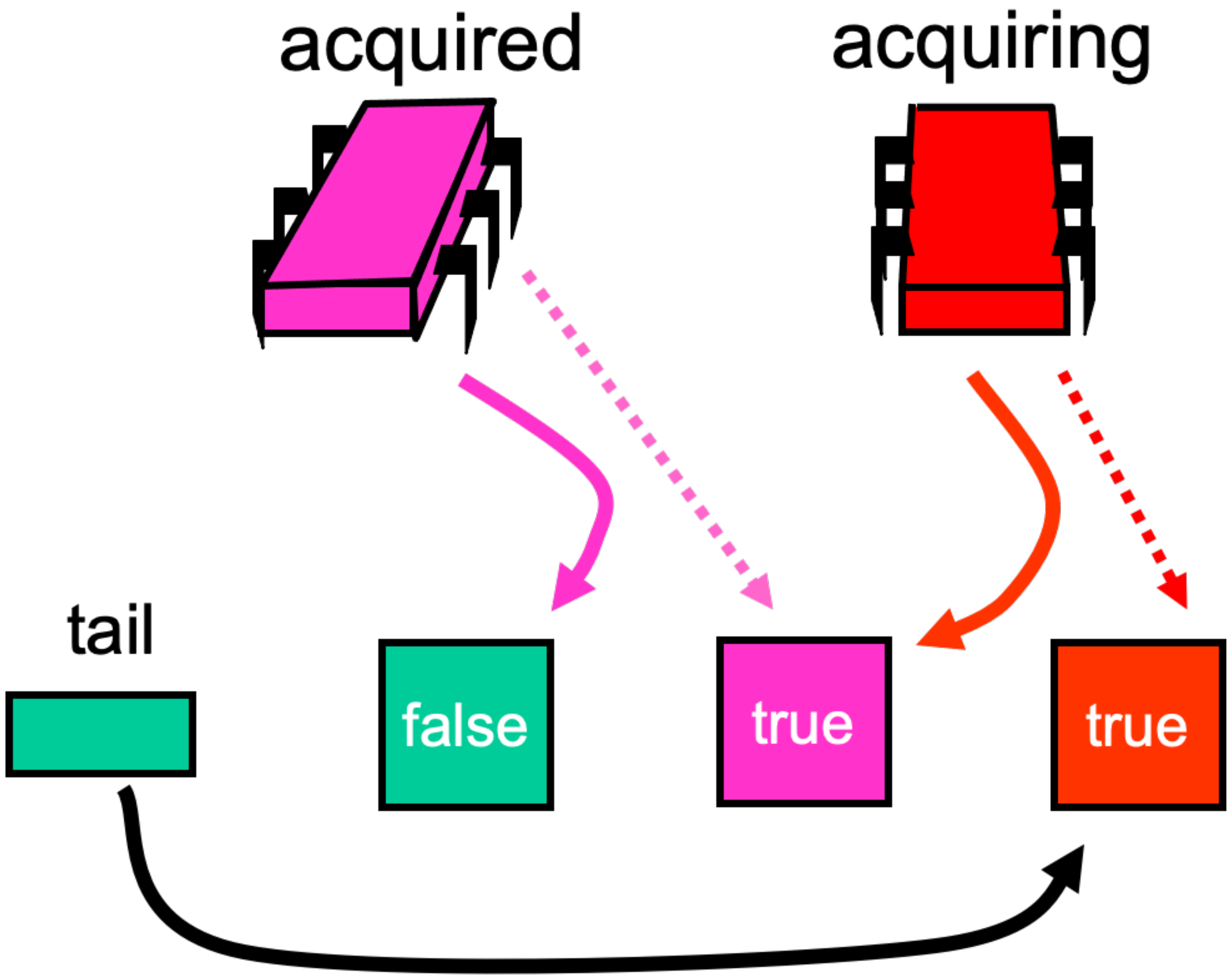


CLH Lock



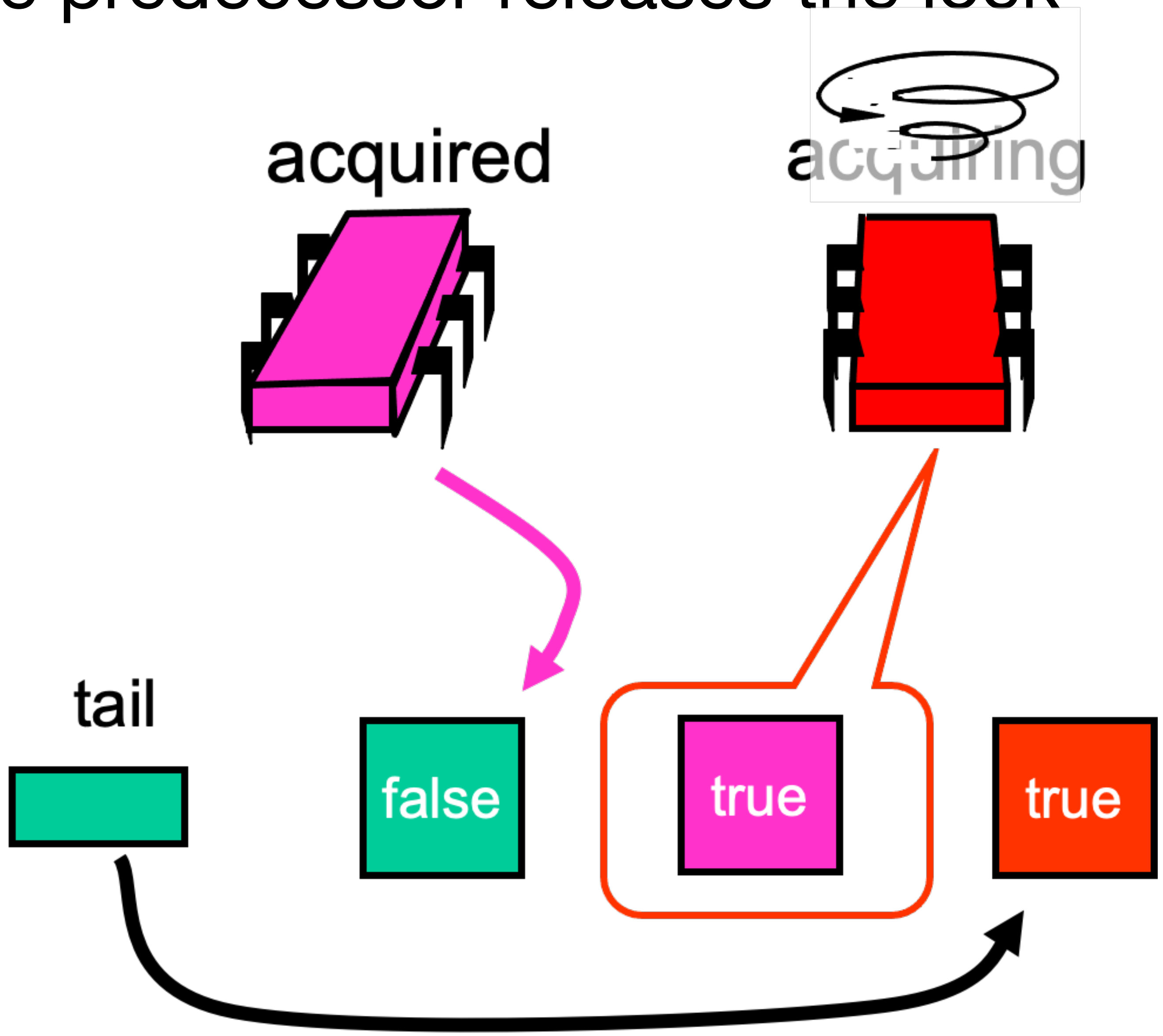
CLH Lock

- Its a virtual list because there are no real pointers between the nodes



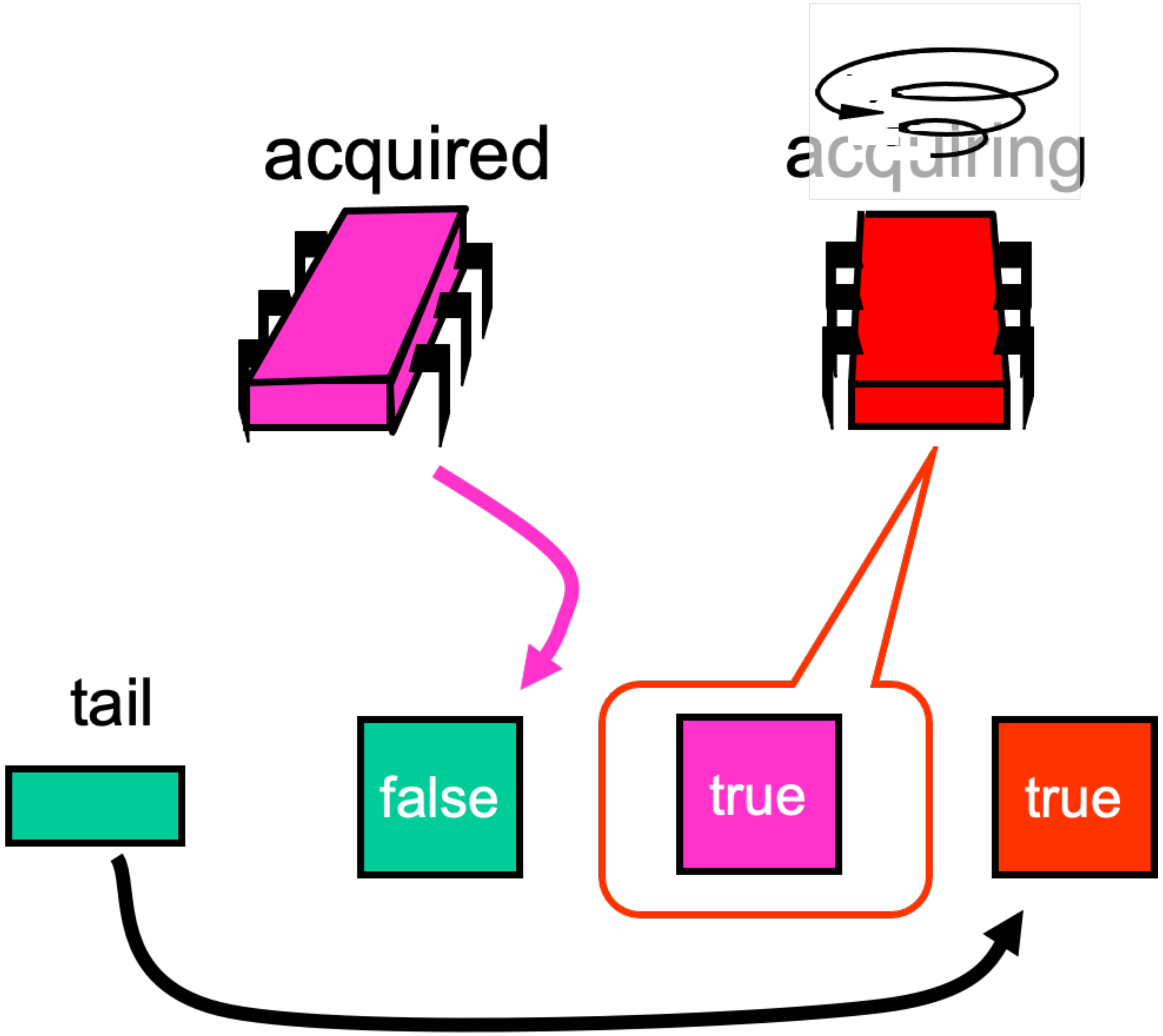
CLH Lock

- The second thread spins on the predecessors QNode
  - Until the predecessor releases the lock

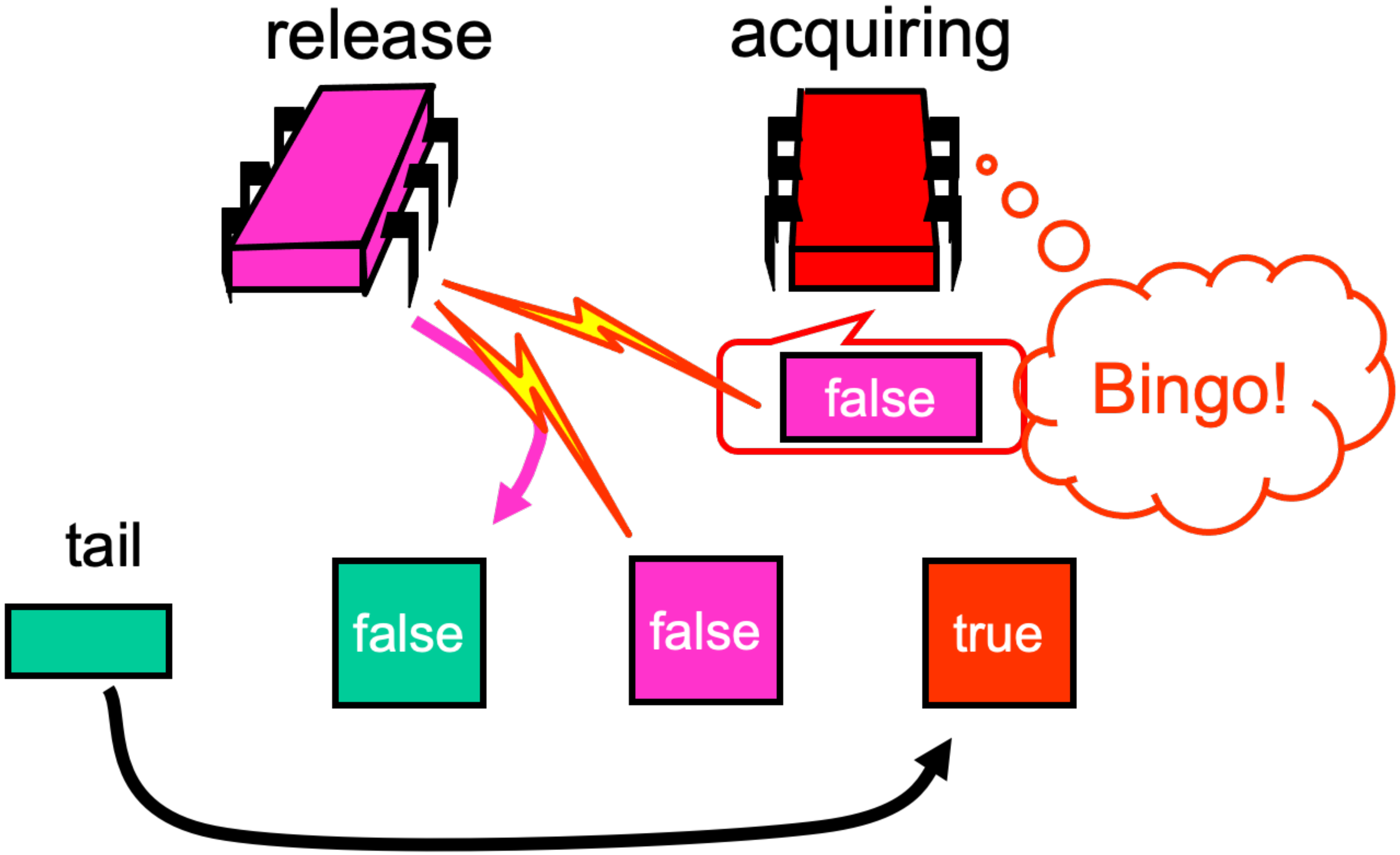


CLH Lock

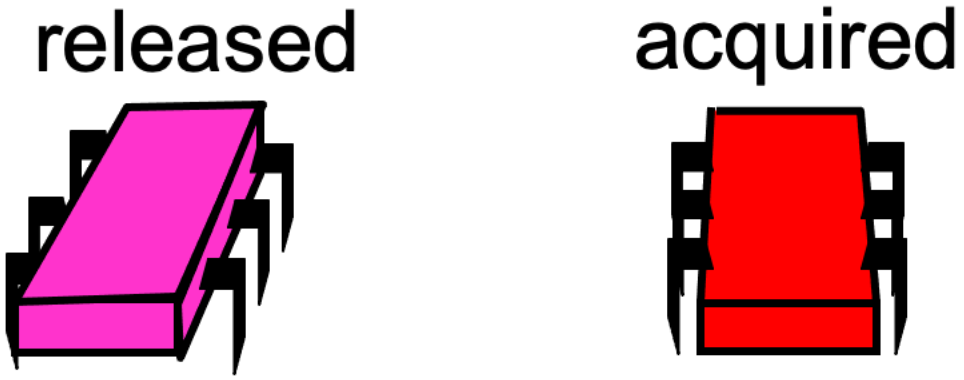
- In fact, it spins on the cached copy of the first thread's node

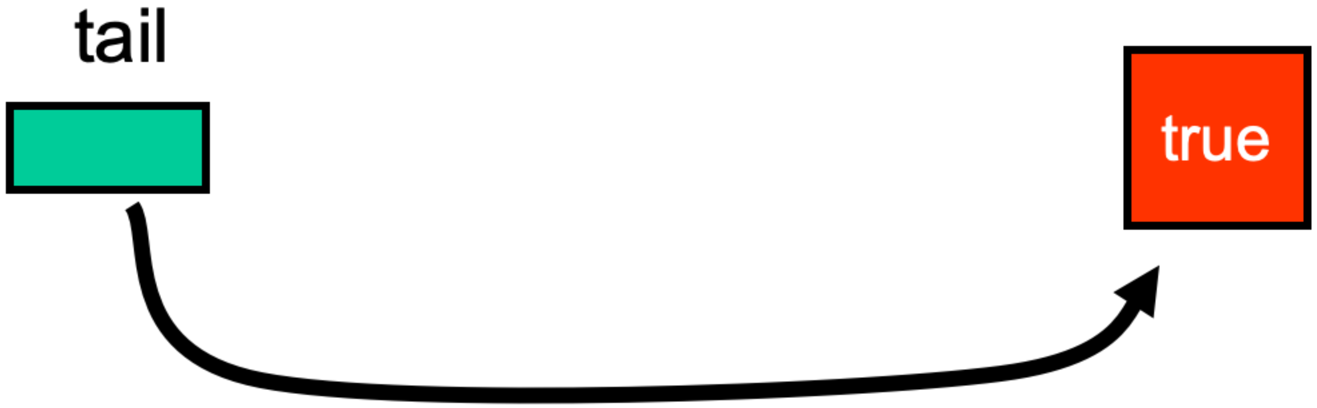


CLH Lock



CLH Lock





tail

true

CLH Lock

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

CLH Lock

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

CLH Lock

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

Queue
Tail

CLH Lock

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

Thread-local QNode

CLH Lock

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

Swap in my node

CLH Lock

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

Spin until predecessor is released

CLH Lock

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

Notify successor

CLH Lock

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

Recycle predecessor's node