DD2451
Parallel and Distributed Computing

FDD3008
Distributed Algorithms

Lecture 5

Spin Locks

Mads Dam

Autumn/Winter 2011

Slides: Much material due to M. Herlihy

Last Lecture

- No consensus from atomic read-write registers
- Atomic RMW operations needed
- Consensus hierarchy
- TAS consensus number 2
- CompareAndSet consensus number infinity

Focus

Up to now:

- Accurate models
- But idealized …
- Elegant and important constructions
- But maybe not very practical

Now and for next 2-3 lectures:

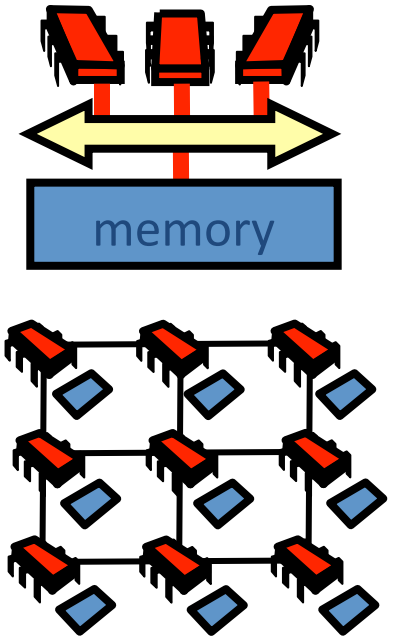
- Models more realistic
- Still focus on principles
- Elegant and important constructions
- Hopefully a bit more practical

Model

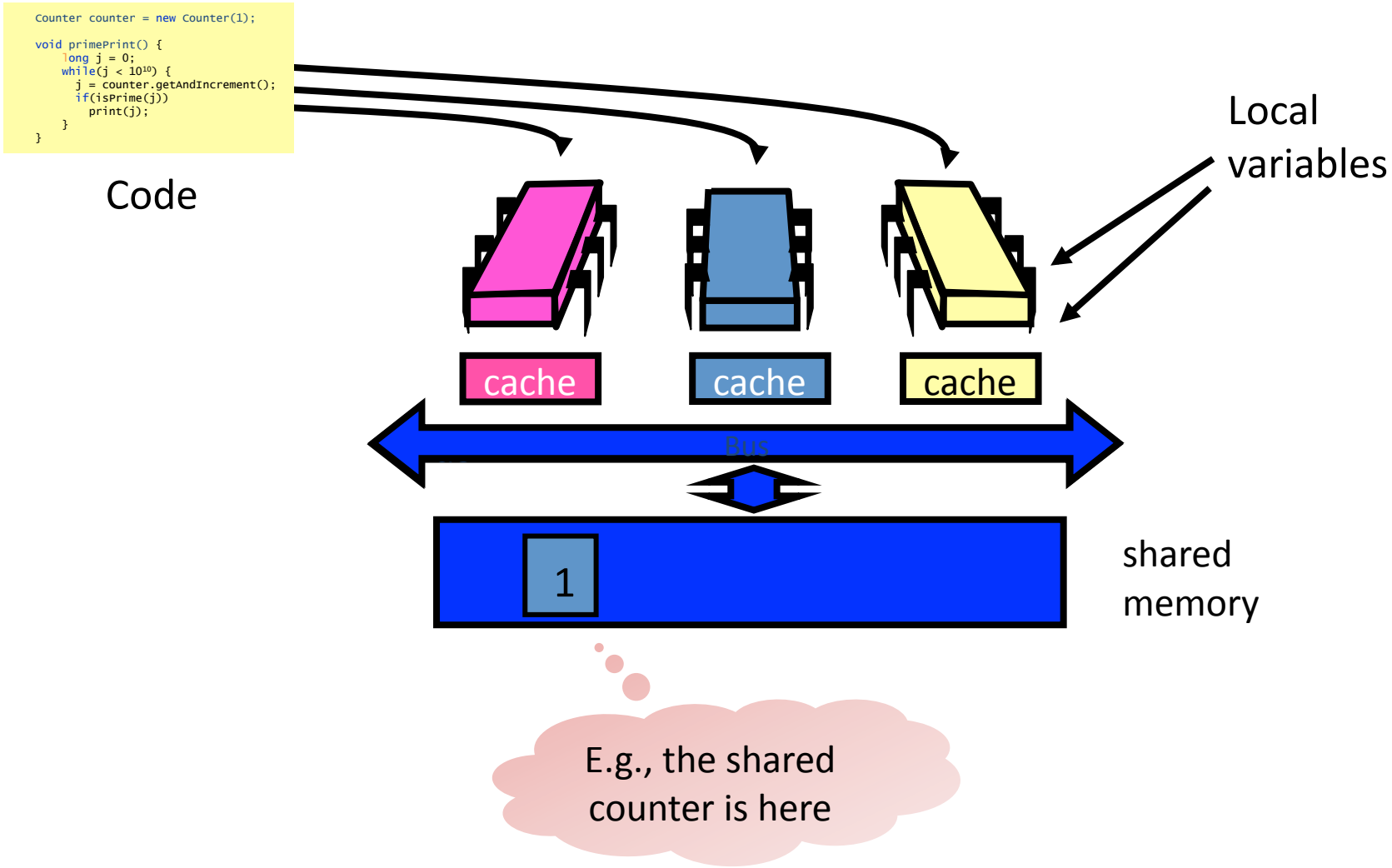
I.e., multiprocessors

- What remains the same?
  - Multiple instruction multiple data (MIMD) architecture
  - Each thread/process has its own code and local variables
  - There is a shared memory that all threads can access

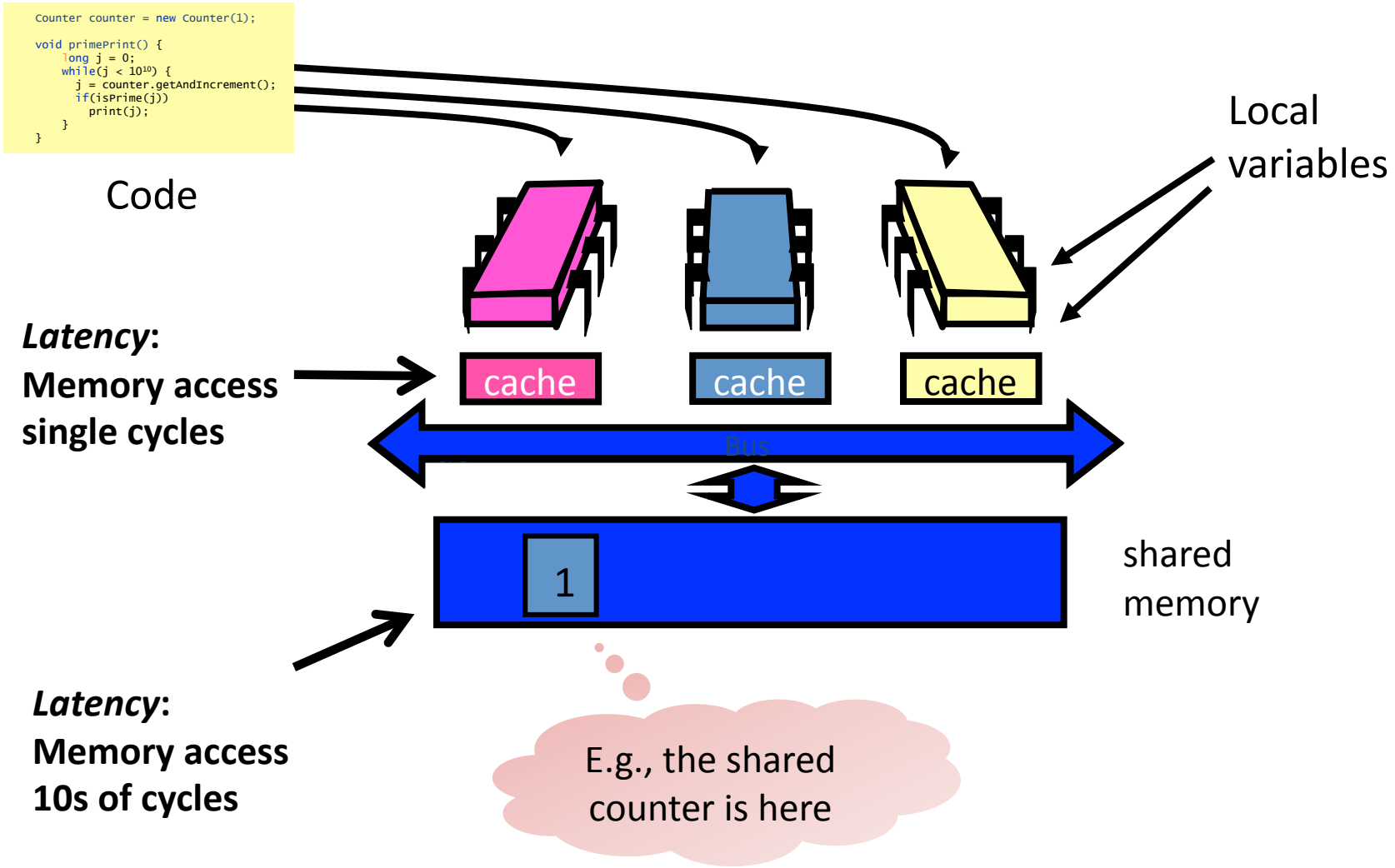
- What is new?
  - Shared bus or distributed memory
  - Communication contention
  - Communication latency
  - Each thread has a local cache



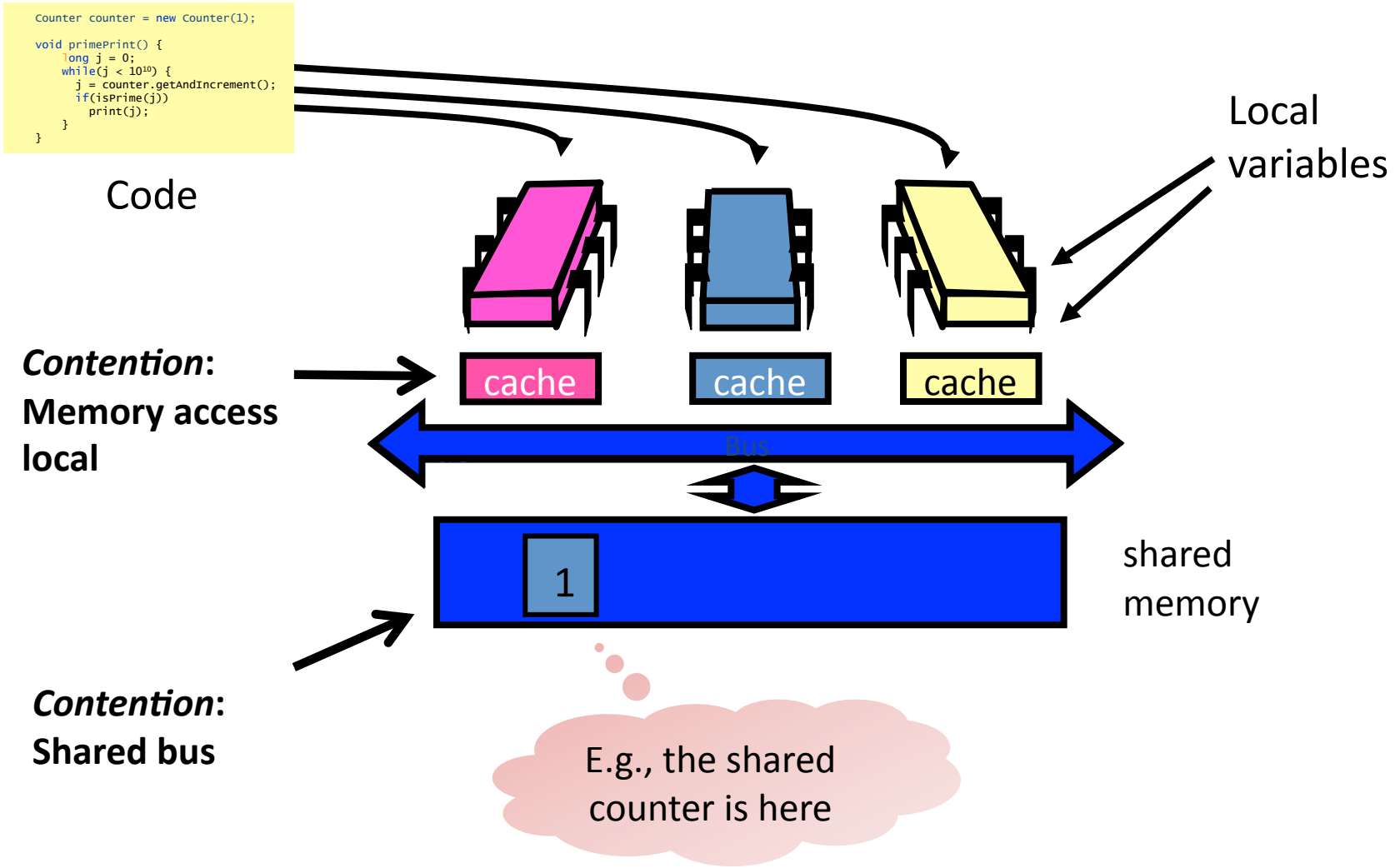
Model: Where Things Reside



Model: Where Things Reside



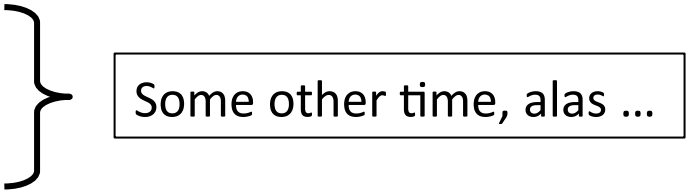
Model: Where Things Reside



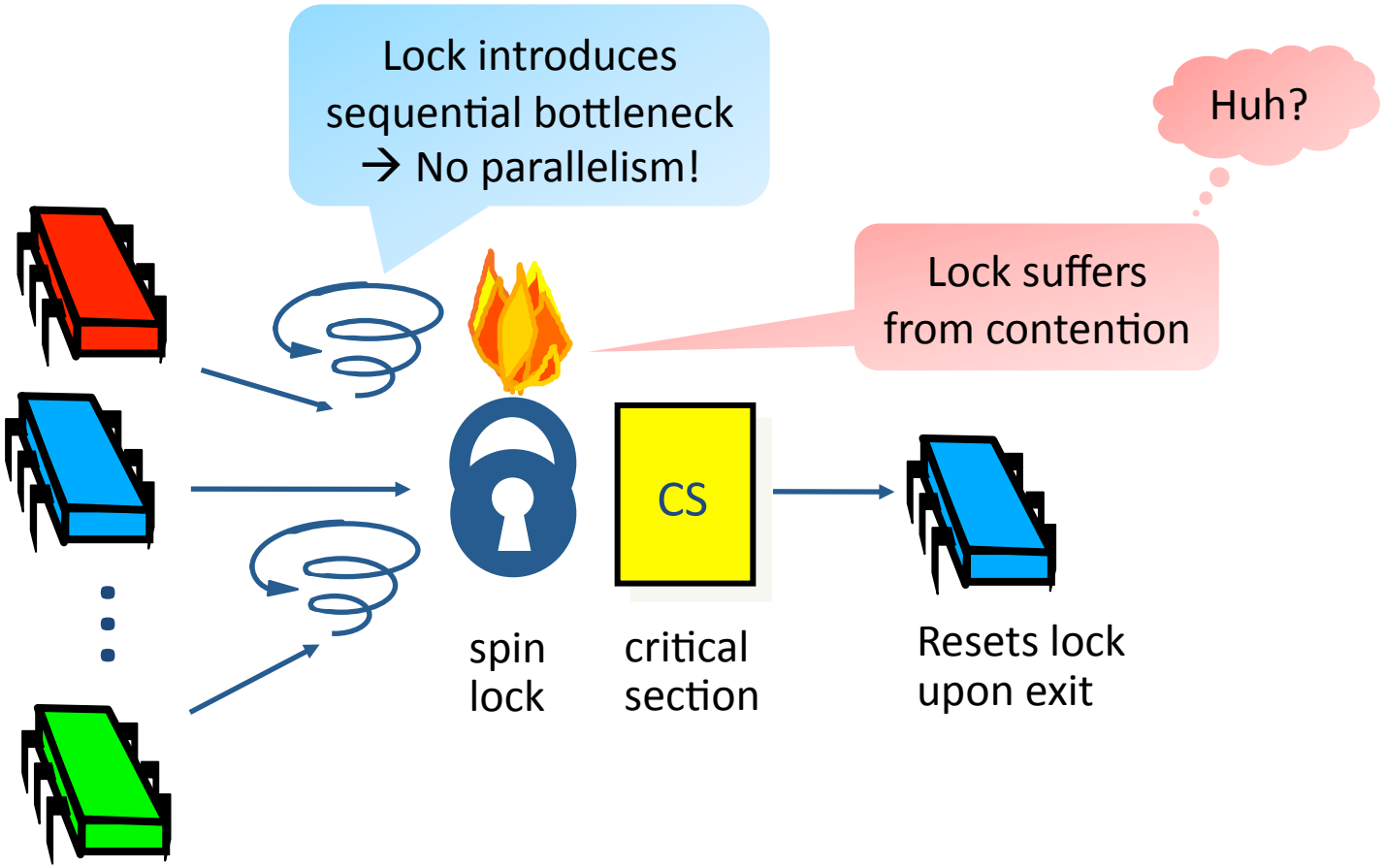
What Should You Do If You Can't Get a Lock?

- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short

- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor



Basic Spin-Lock



Review: Test&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**
- Also known as “getAndSet”

Review: Test&Set

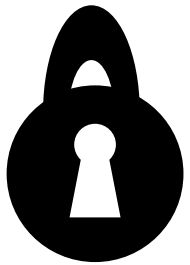
```
public class AtomicBoolean {
  private boolean value;

  public synchronized boolean getAndSet() {
    boolean prior = this.value;
    this.value = true;
    return prior;
  }

}
```

java.util.concurrent.atomic

Get current value and set value to true

Test&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&Set Lock

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

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

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

Lock state is AtomicBoolean

Keep trying until lock acquired

Release lock by resetting state to false

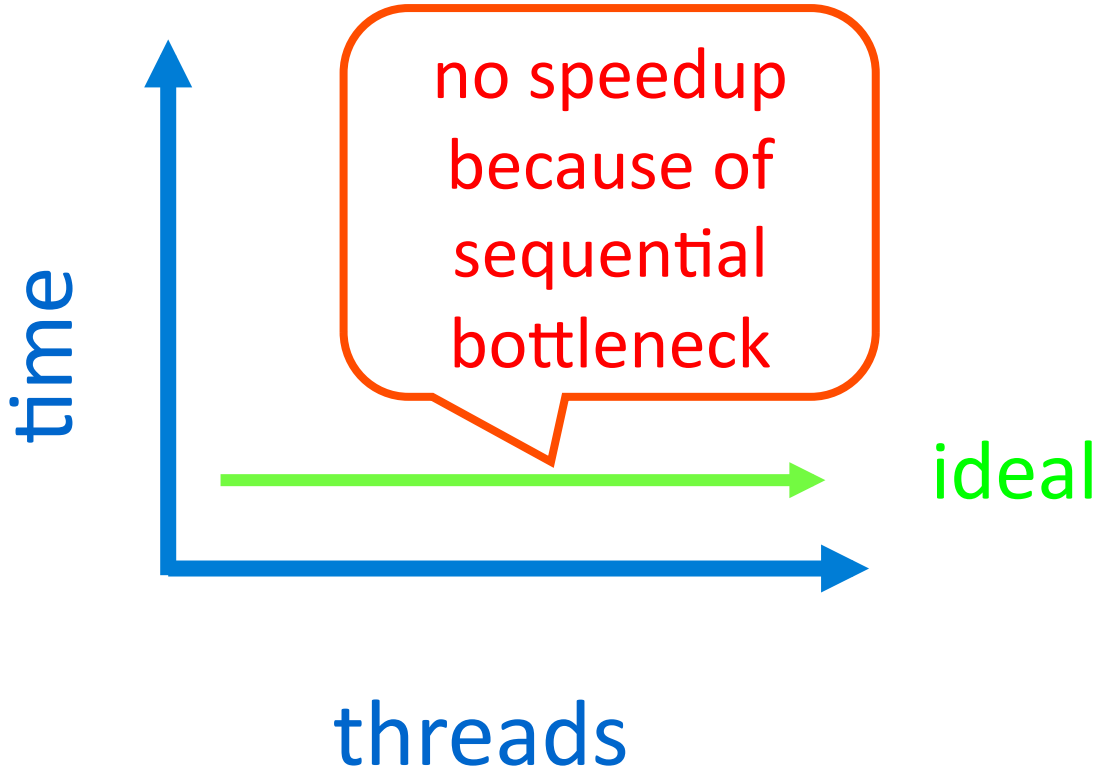
Space Complexity

- TAS spinlock has small “footprint”
- *n* thread spinlock uses O(1) space
- As opposed to *O*(*n*) Peterson/Bakery
- How did we overcome the big-omega(*n*) lower bound?
- We used an RMW operation

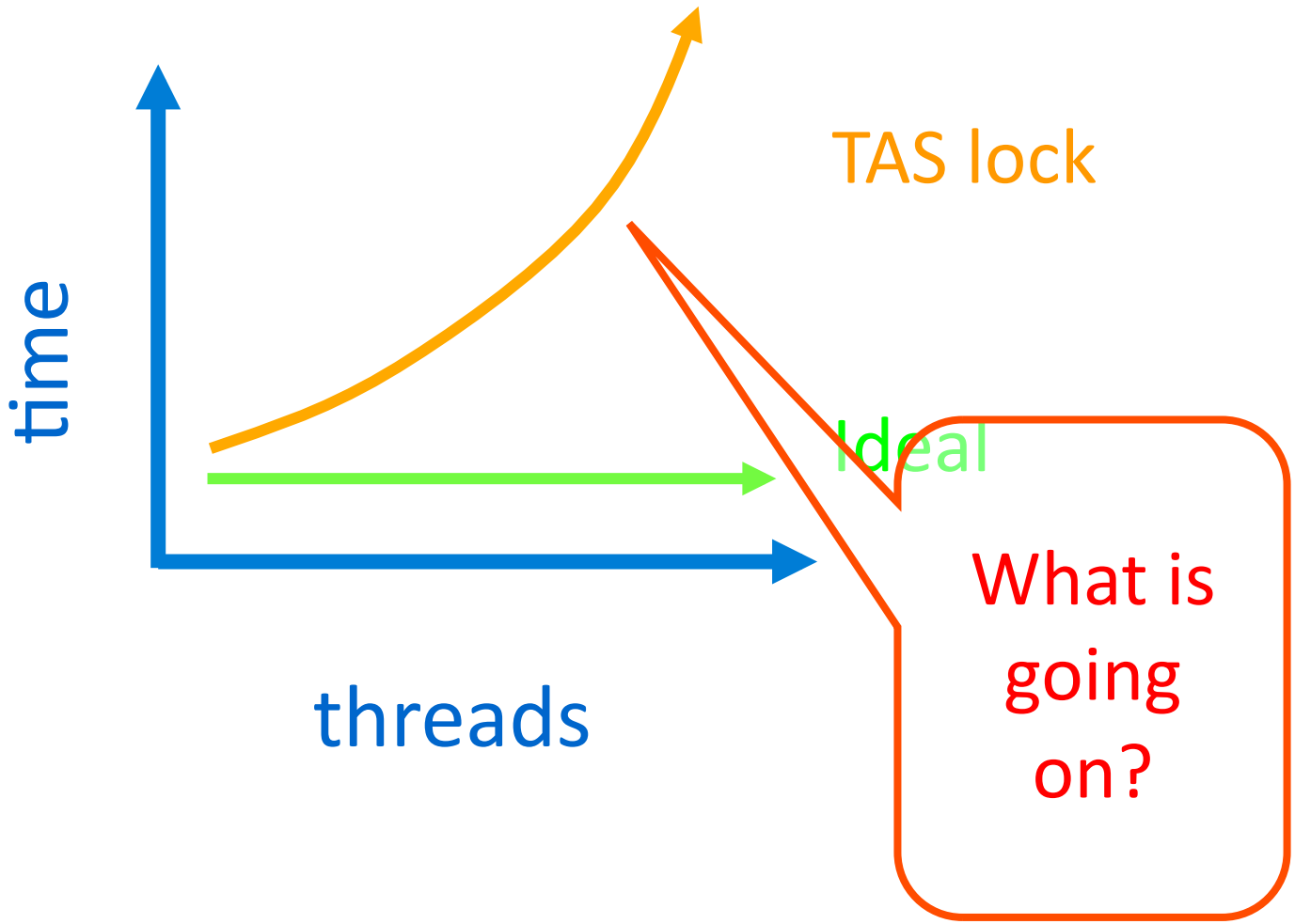
Performance

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

How Long Should It Take?



How Long Does It Take?



Test&Test&Set Locks

- Lurking stage
  - Wait until lock “looks” free
  - Spin while read returns true (i.e., the lock is taken)

- Pouncing state
  - As soon as lock “looks” available
  - Read returns false (i.e., the lock is free)
  - Call TAS to acquire the lock
  - If TAS loses, go back to lurking

Test&Test&Set Lock

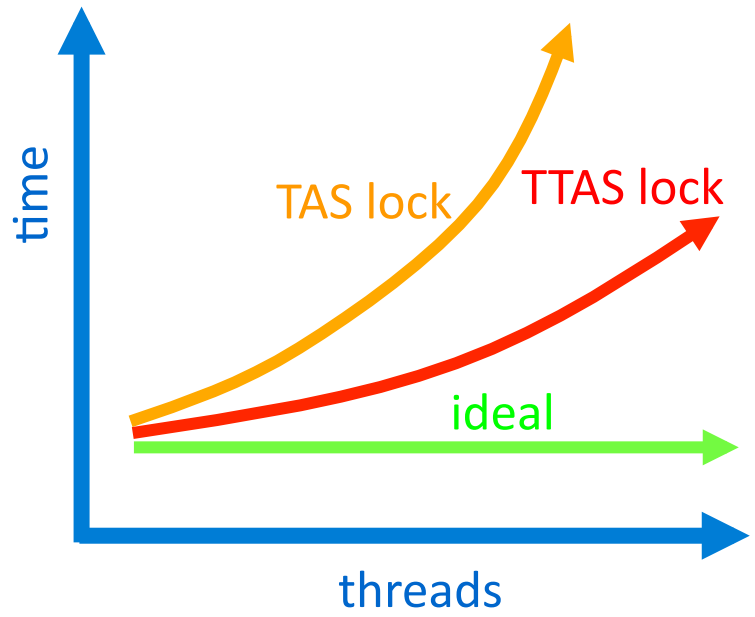
```
public class TTASLock implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);

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

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

Wait until lock looks free

Then try to acquire it

Performance

- Both TAS and TTAS do the same thing (in our old model)
- So, we would expect basically the same results



- Why is TTAS so much better than TAS? Why are both far from ideal?

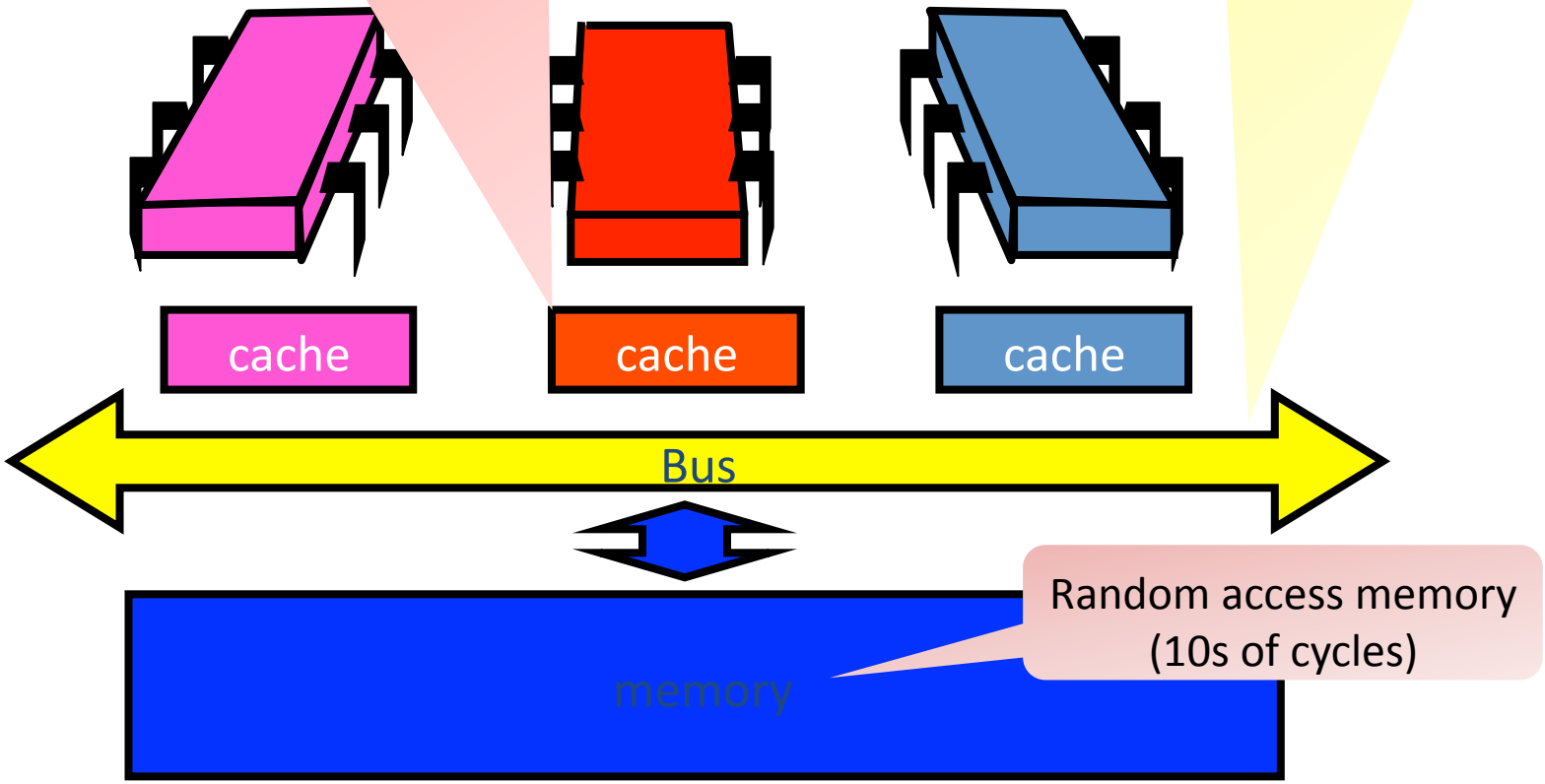
Opinion

- TAS & TTAS locks
  - are provably the same (in our old model)
  - except they aren't (in field tests)
- Obviously, it must have something to do with the model…
- Let's take a closer look at our new model and try to find a reasonable explanation!

Bus-Based Architectures

Per-processor caches
- Small
- Fast: 1 or 2 cycles
- Address and state information

Shared bus
- Broadcast medium
- One broadcaster at a time
- Processors (and memory) “snoop”

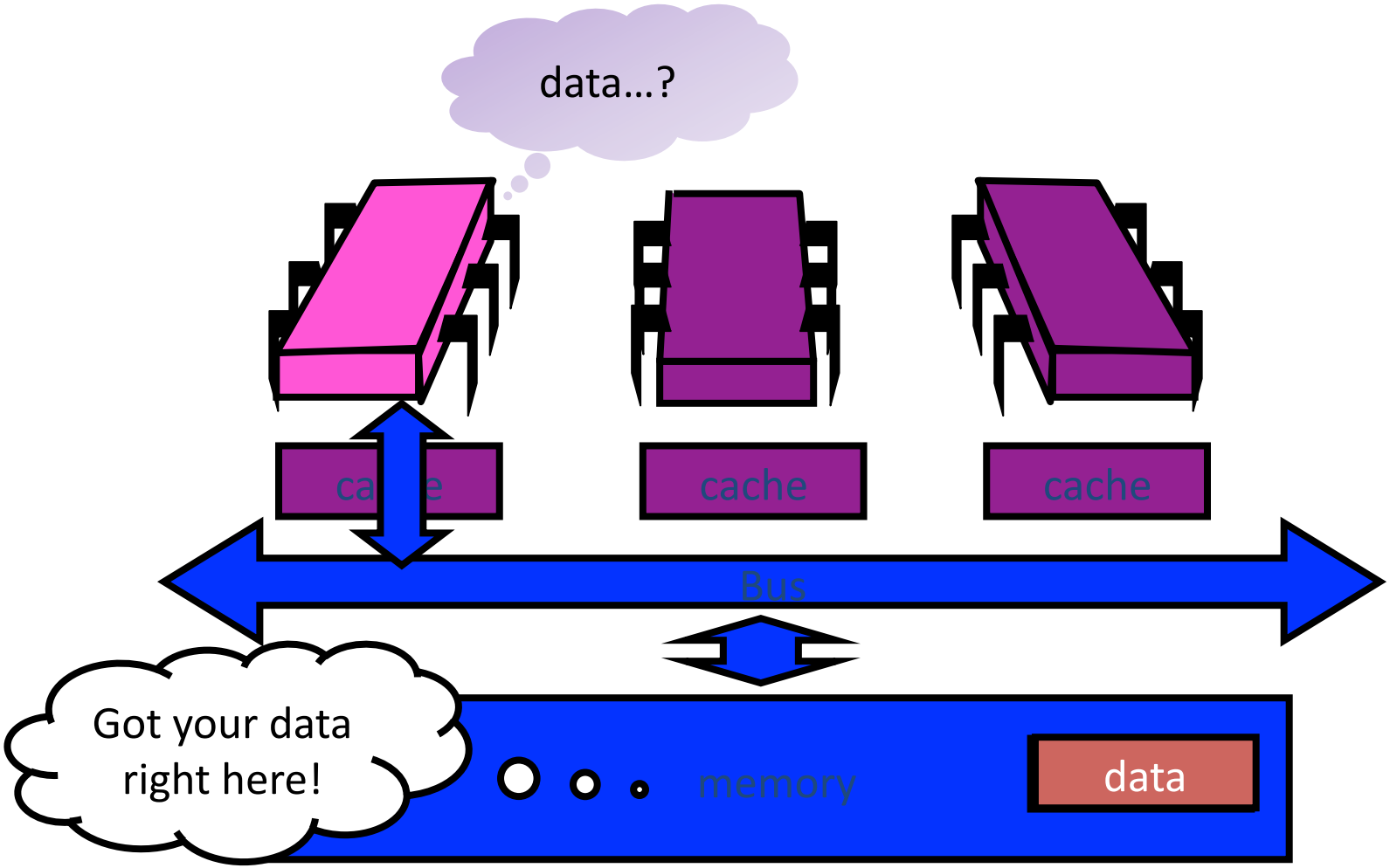


Jargon Watch

- Load request
  - When a thread wants to access data, it issues a load request
- Cache hit
  - The thread found the data in its own cache
- Cache miss
  - The data is not found in the cache
  - The thread has to get the data from memory

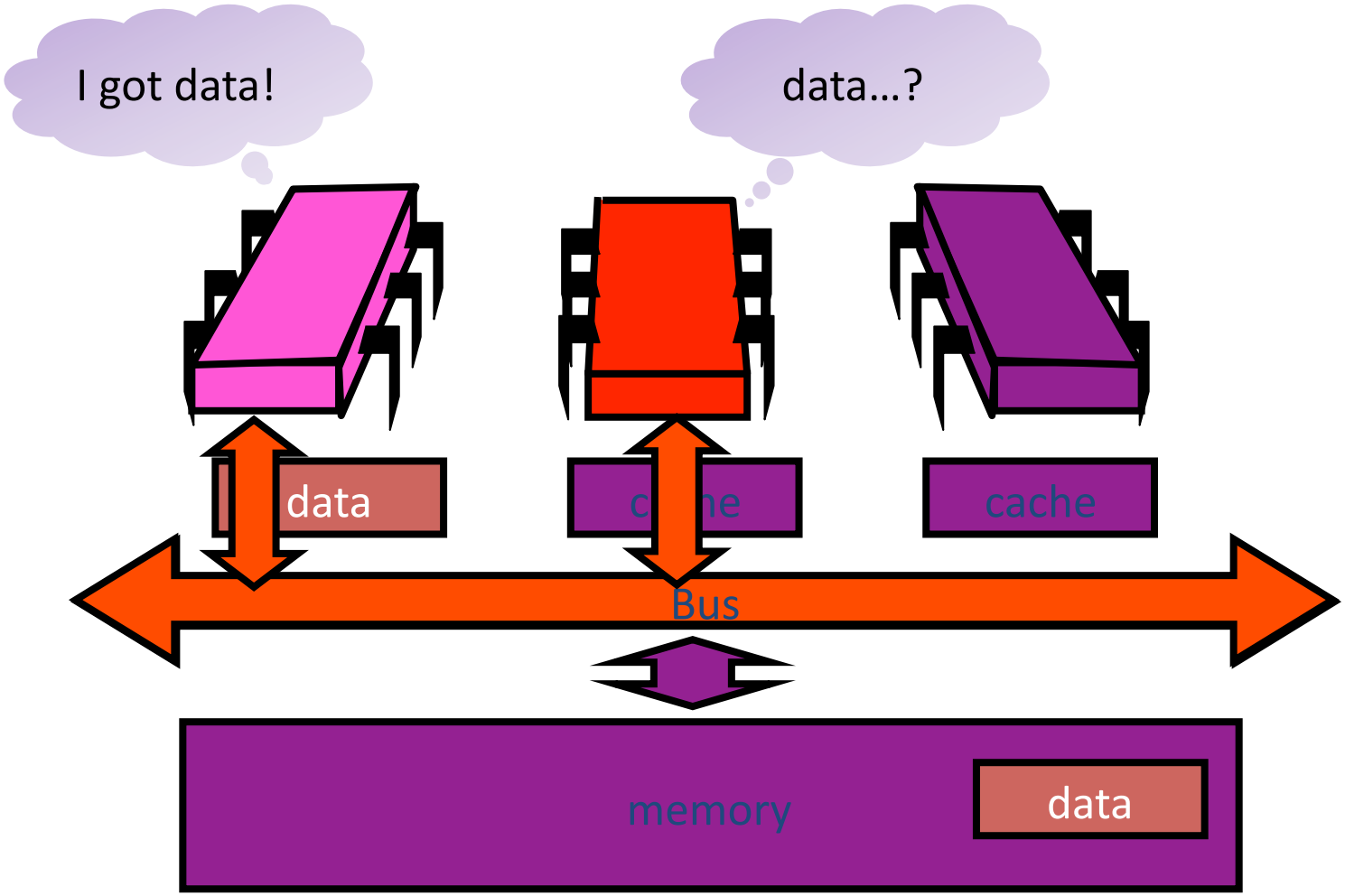
Load Request

- Thread issues load request and memory responds



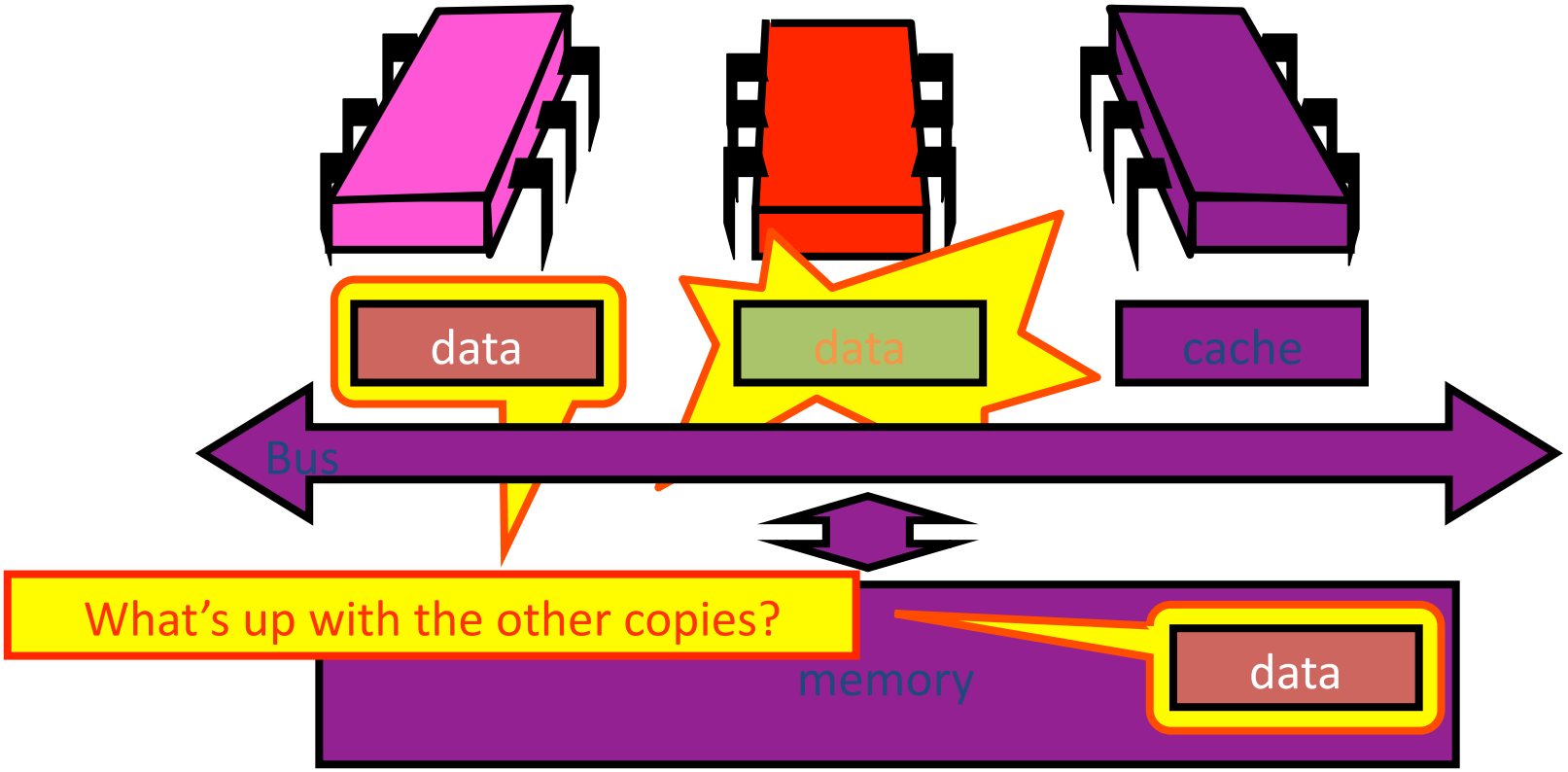
Another Load Request

- Another thread wants to access the same data. Get a copy from the cache!



Modify Cached Data

- Both threads now have the data in their cache
- What happens if the red thread now modifies the data…?



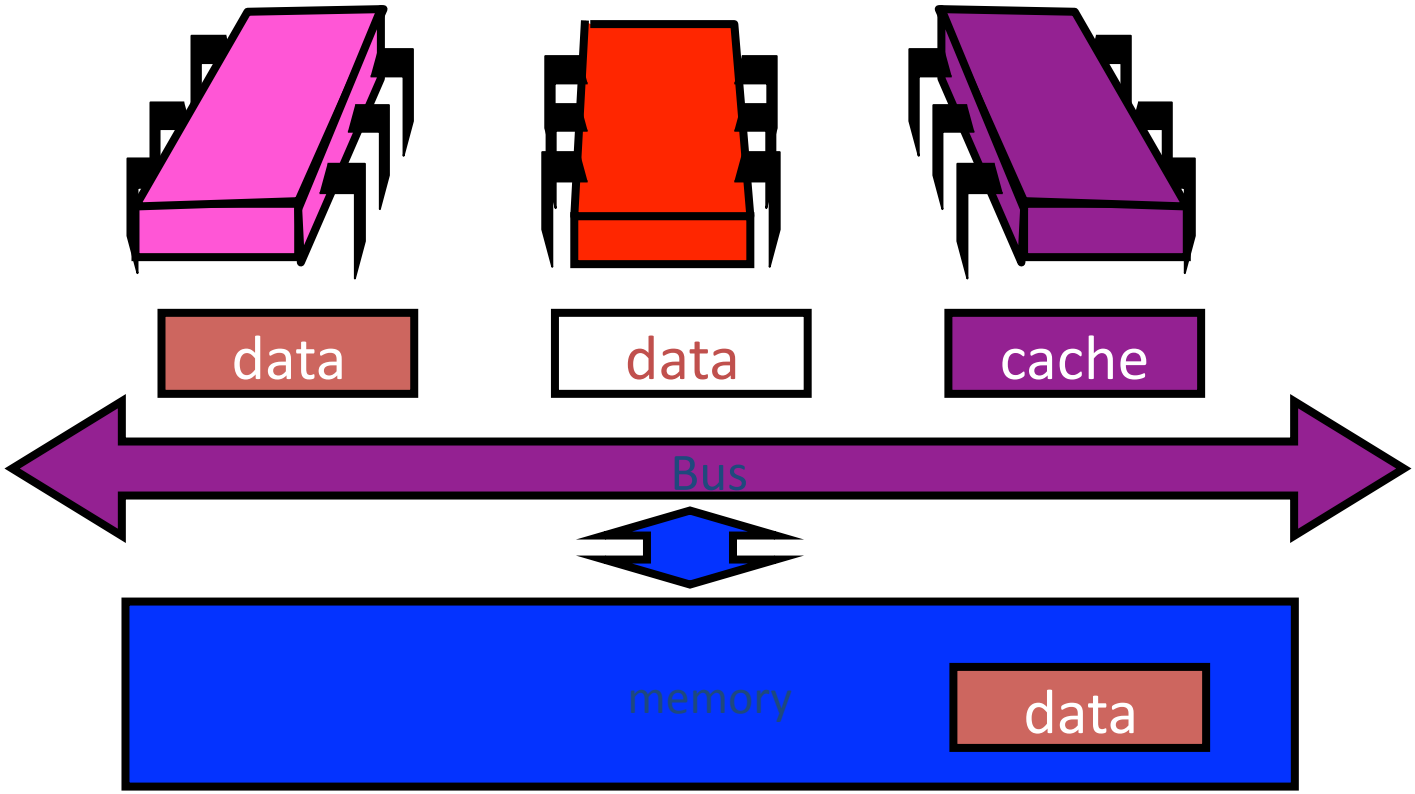
Cache Coherence

- 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?

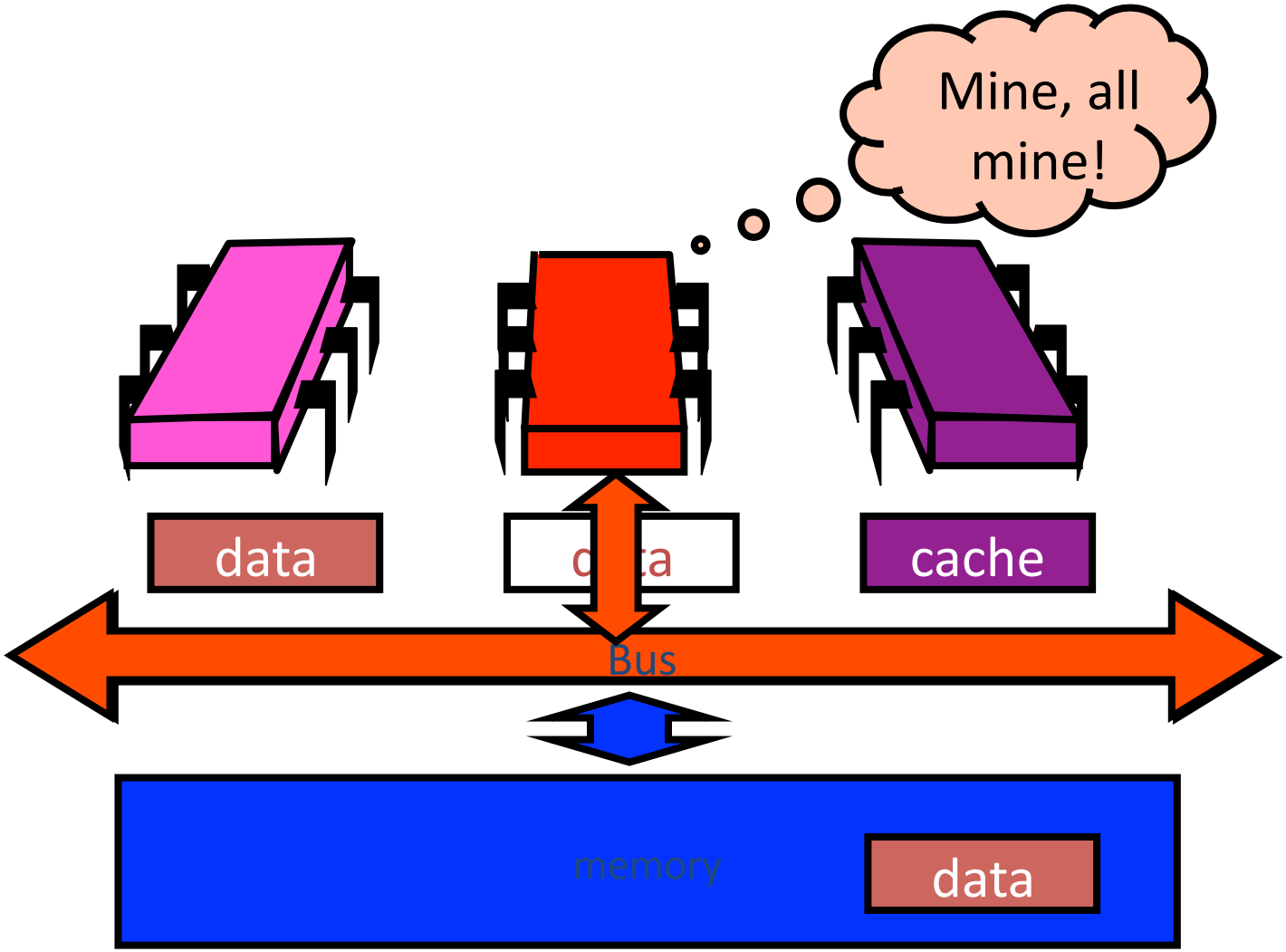
Write-Back Caches

- Accumulate changes in cache
- Write back when needed
  - Need the cache for something else
  - Another processor wants it
- On first modification
  - Invalidate other entries
  - Requires non-trivial protocol …
- Cache entry has three states:
  - Invalid: contains raw bits
  - Valid: I can read but I can't write
  - Dirty: Data has been modified
    - Intercept other load requests
    - Write back to memory before reusing cache

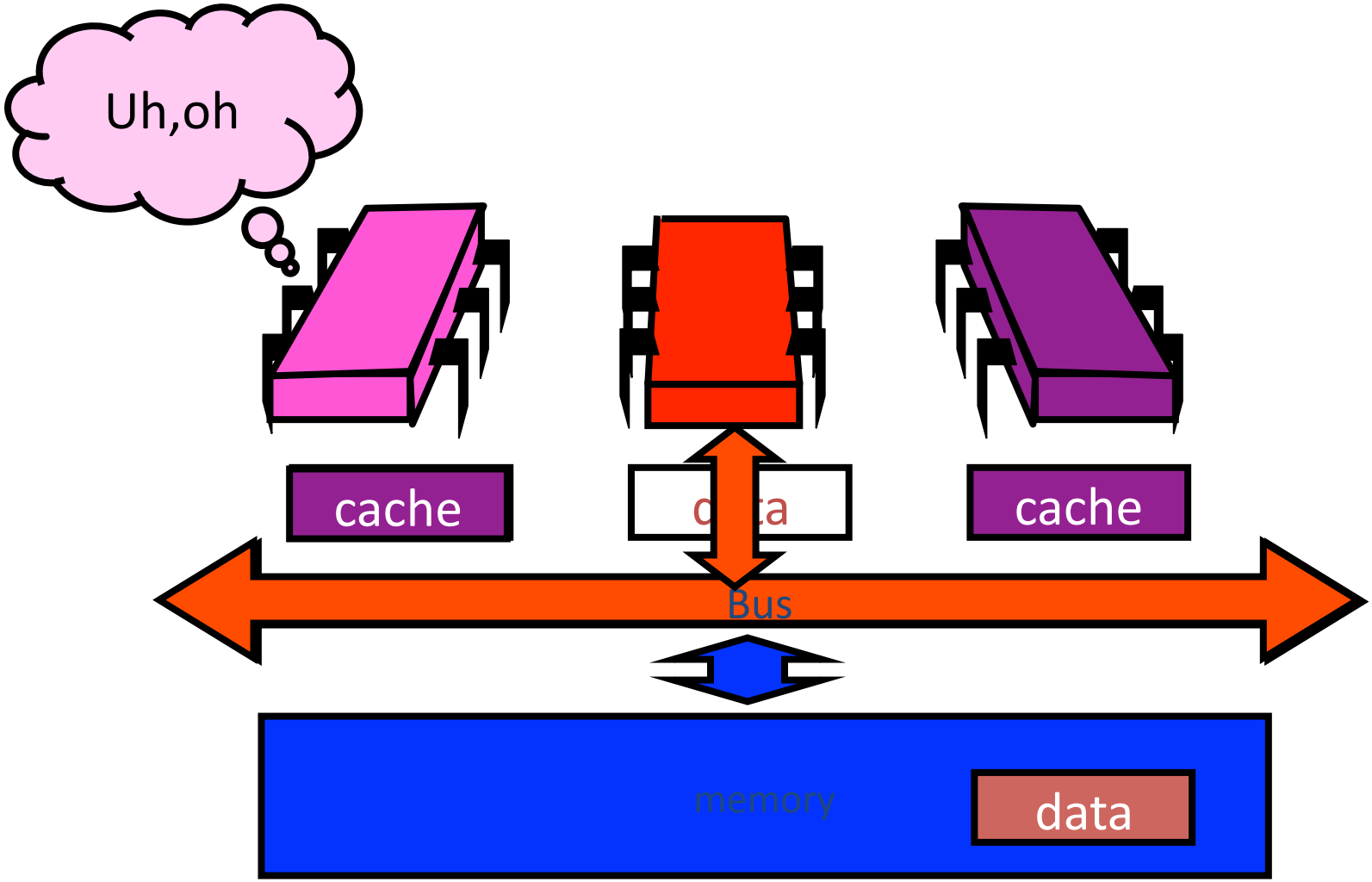
Invalidate



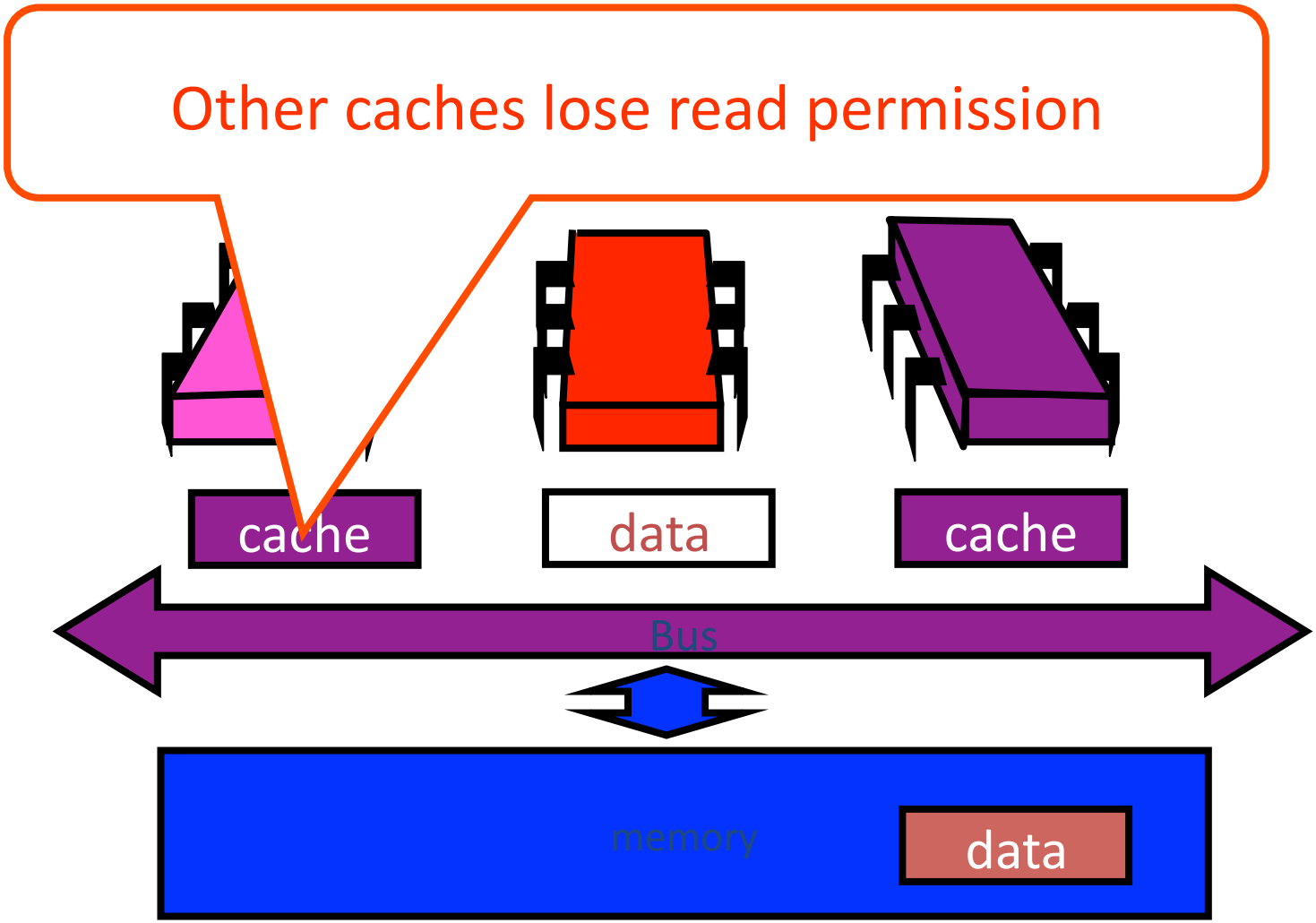
Invalidate



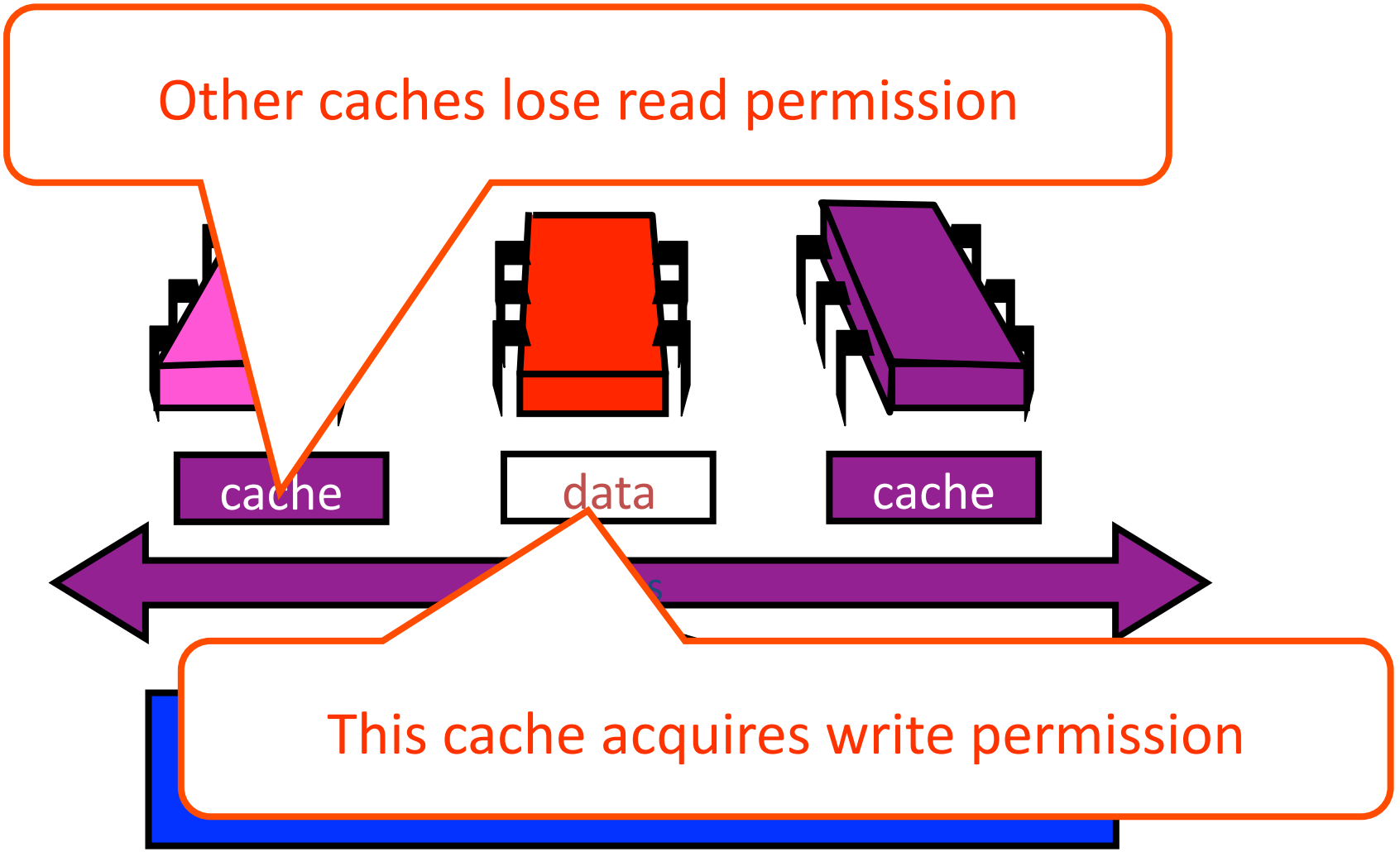
Invalidate



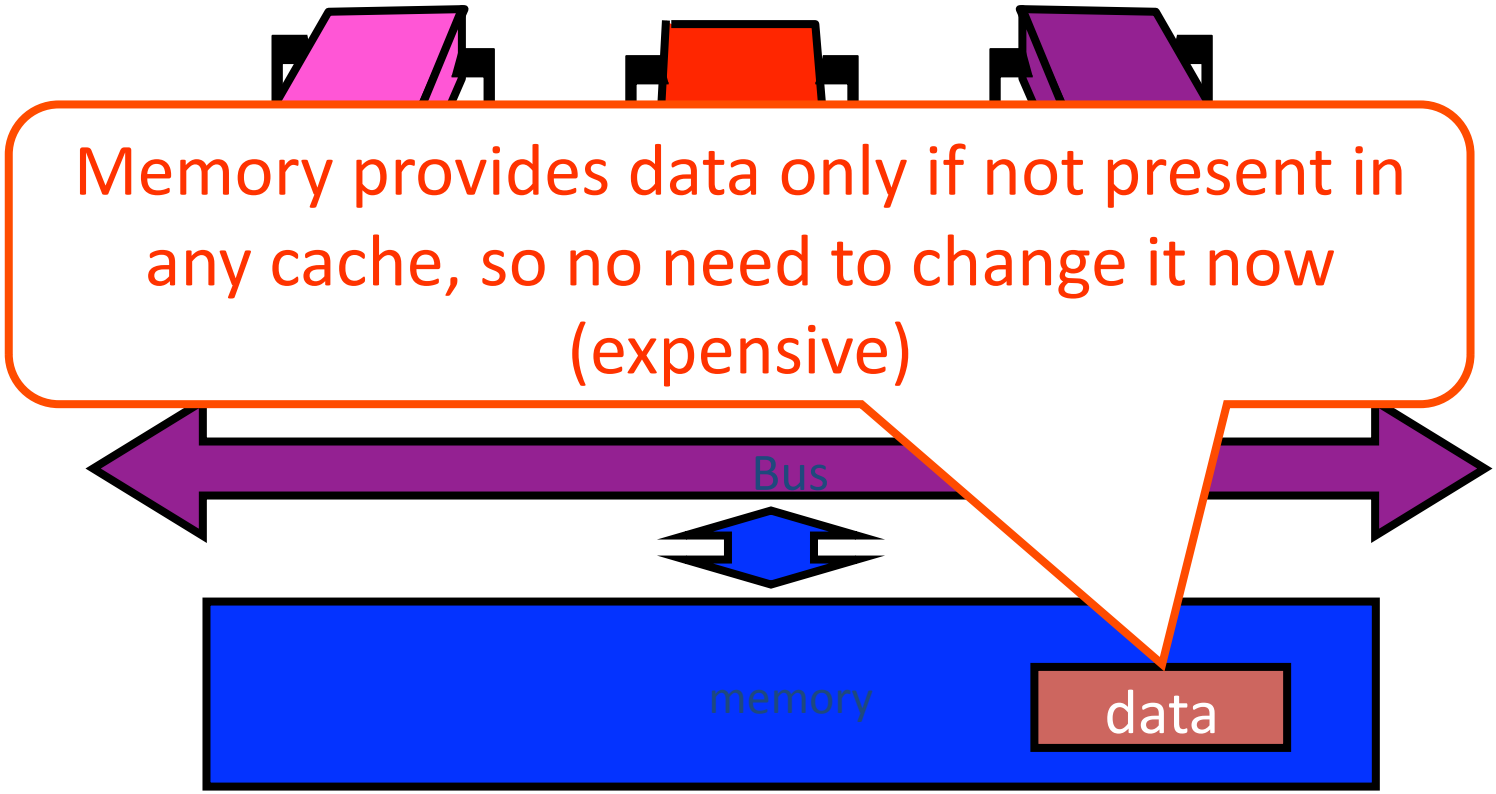
Invalidate


Other caches lose read permission
cache
data
cache
Bus
memory
data

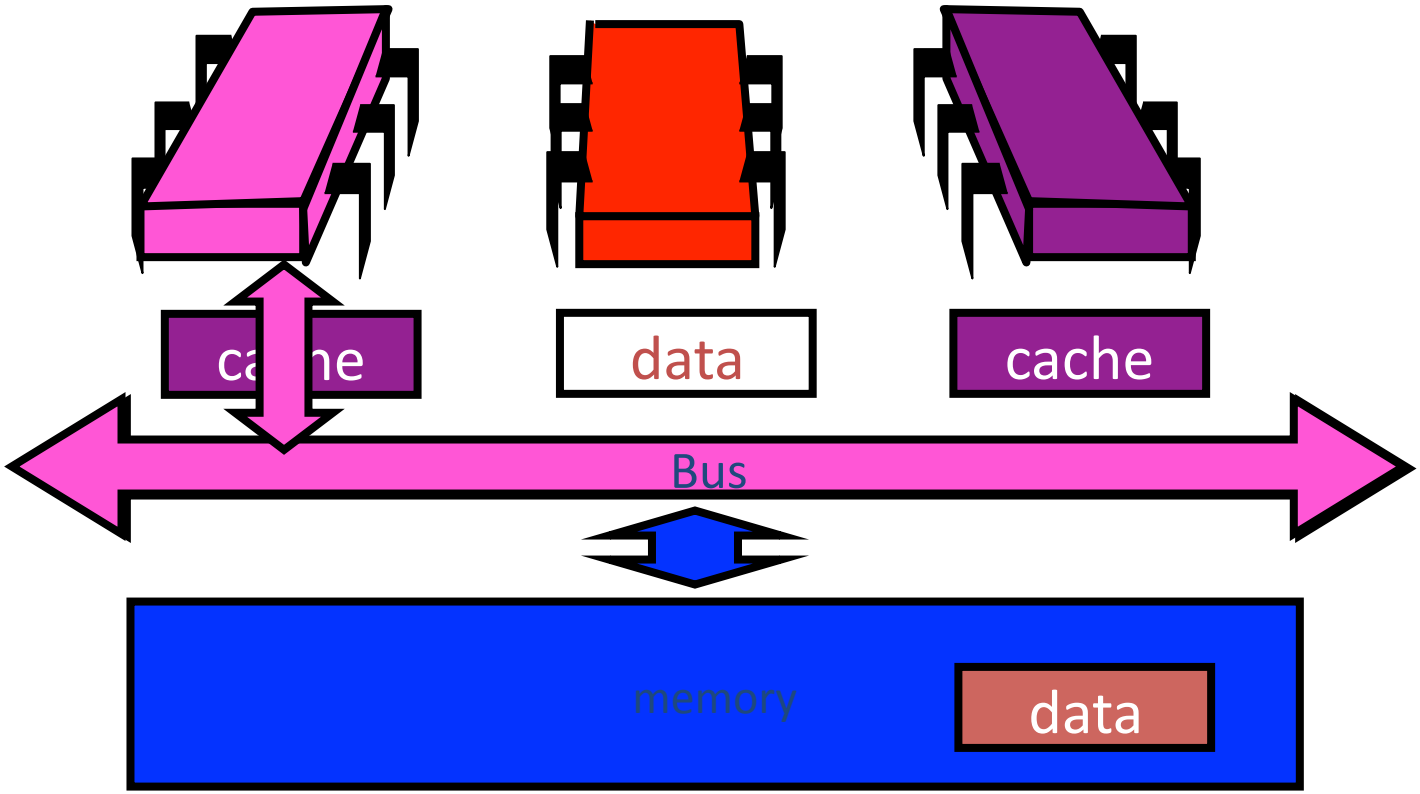
Invalidate


Other caches lose read permission
cache
data
cache
This cache acquires write permission

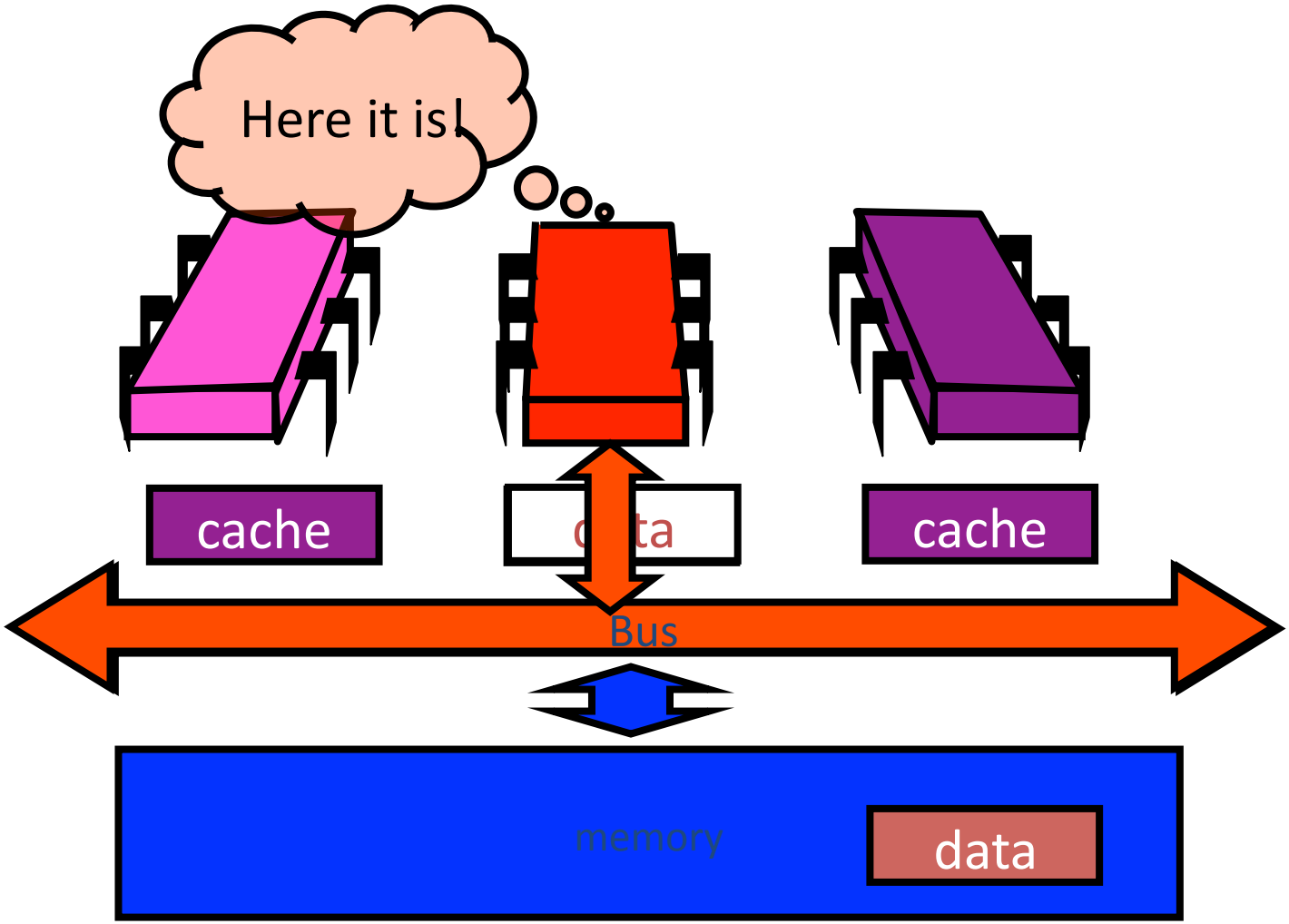
Invalidate


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

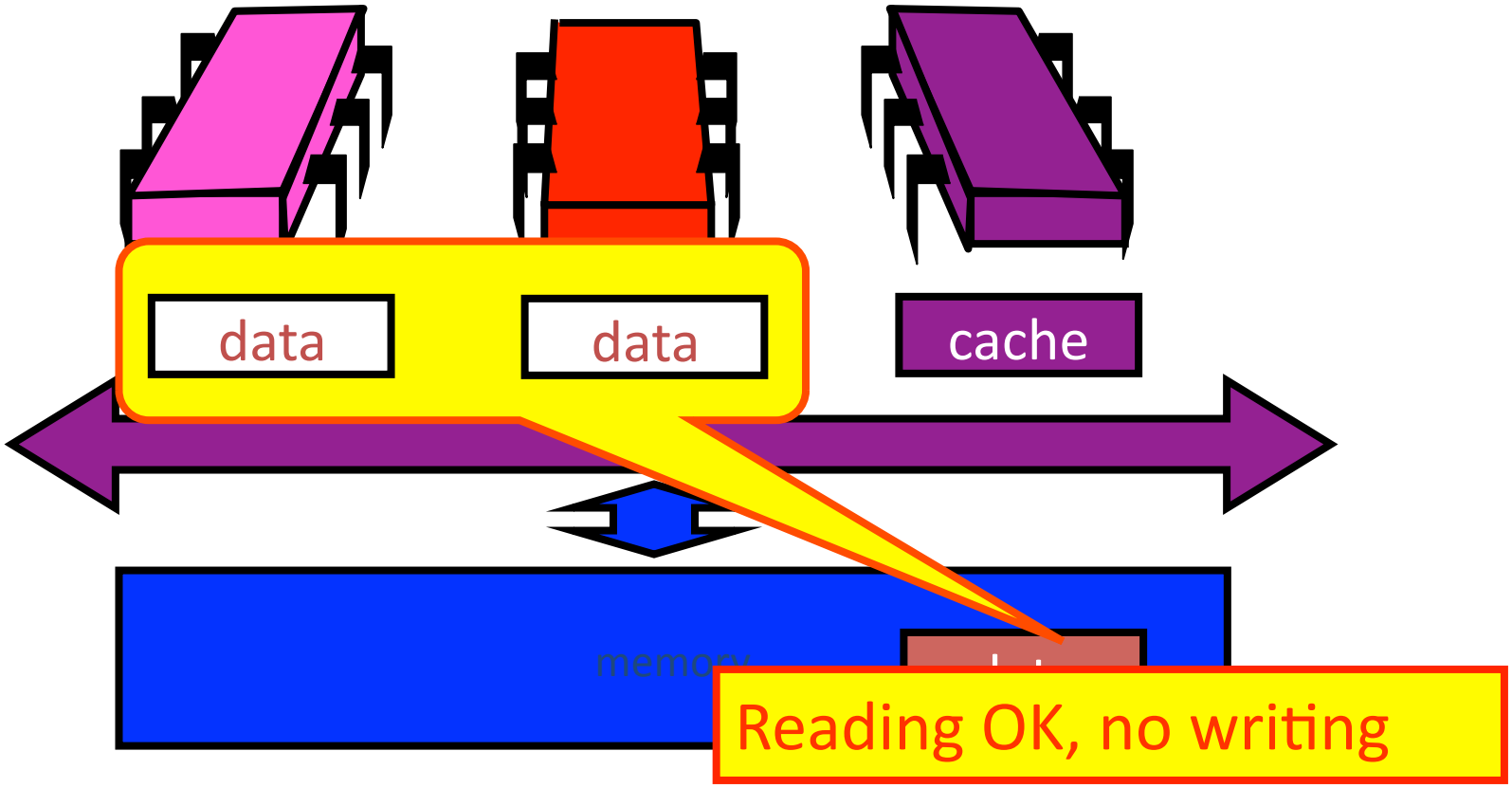
Another Processor Asks for Data



Owner Responds



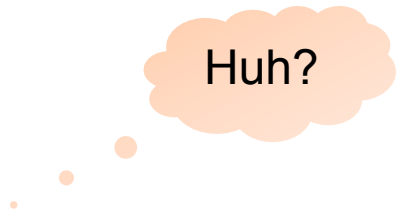
End of the Day …



TAS vs. TTAS

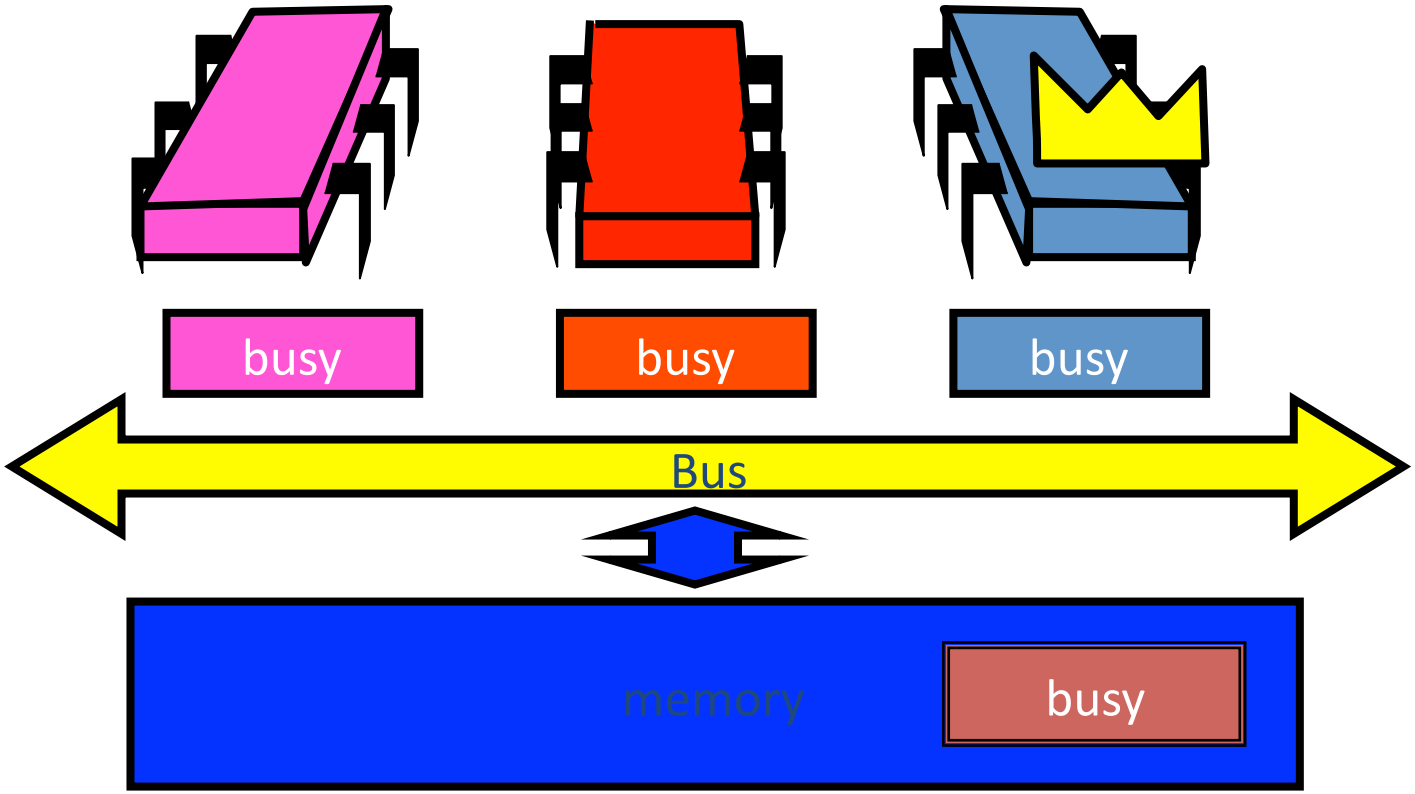
- Each getAndSet call broadcast on bus
  - invalidates cache lines
- Spinners
  - Miss in cache
  - Go to bus
- Thread wants to release lock
  - delayed behind spinners!!!

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



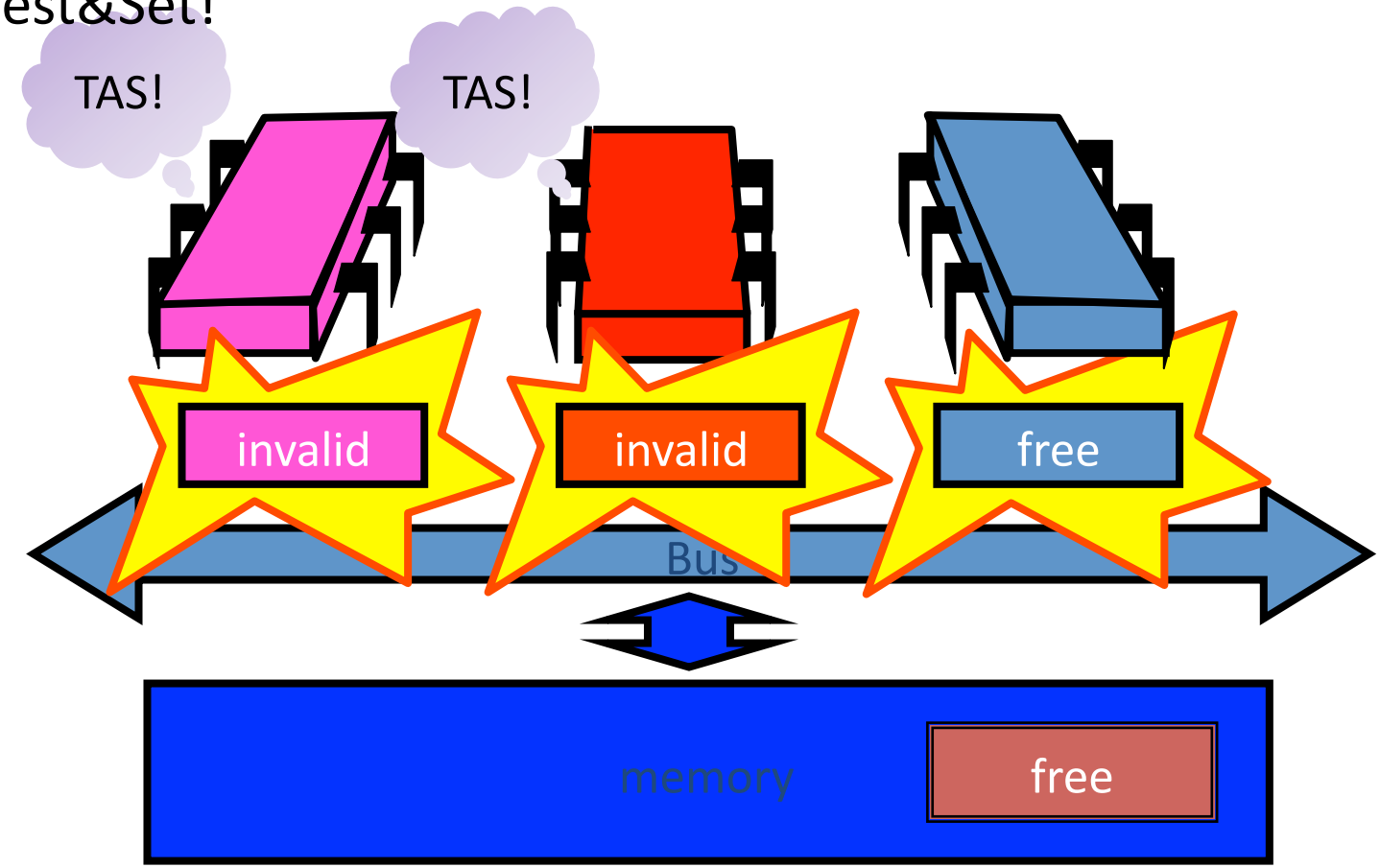
Local Spinning while Lock is Busy

- While the lock is held, all contenders spin in their caches, rereading cached data without causing any bus traffic



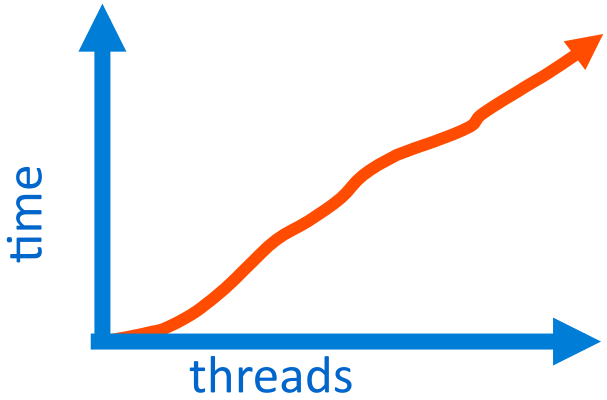
On Release

- The lock is released. All spinners take a cache hit and call Test&Set!



Time to Quiescence

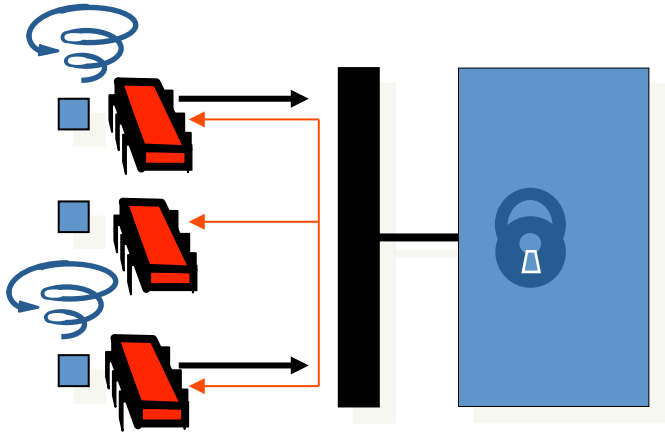
- Every process gets a cache miss
  - All state.get() satisfied sequentially
- Every process does TAS
  - Caches of other processes are invalidated
- Eventual quiescence (“silence”) after acquiring the lock
- The time to quiescence increases linearly with the number of processors for a bus architecture!



Measuring Quiescence Time

X = time of ops that don't use the bus

Y = time of ops that cause intensive bus traffic

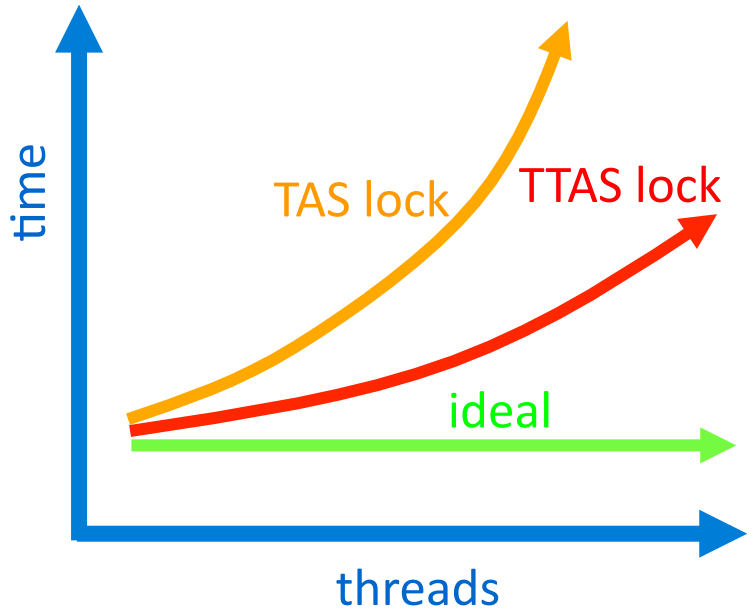


In critical section, run ops X then ops Y. As long as Quiescence time is less than X, no drop in performance.

By gradually varying X, can determine the exact time to quiesce.

Mystery Explained

- Now we understand why the TTAS lock performs much better than the TAS lock, but still much worse than an ideal lock!

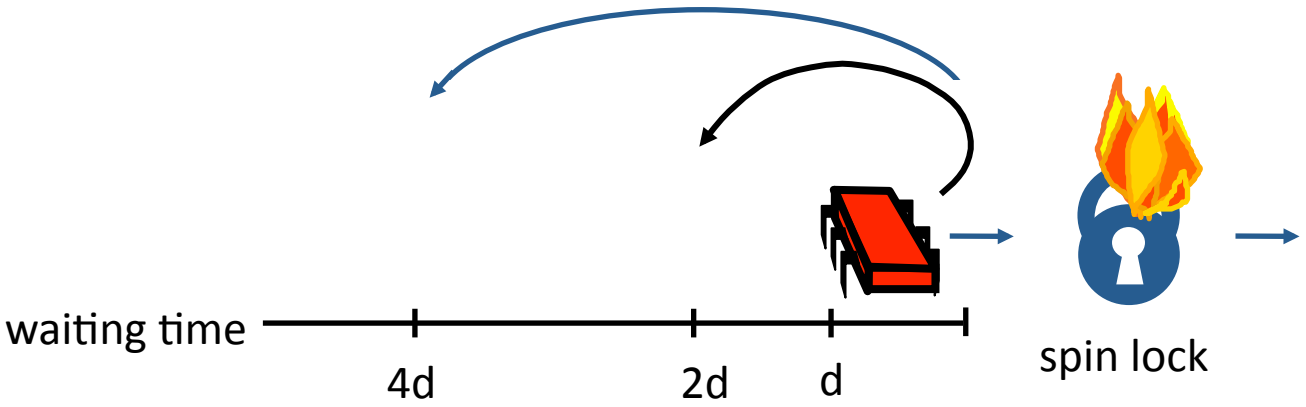


- How can we do better?

Introduce Delay

- If the lock looks free, but I fail to get it, there must be lots of contention
- It's better to back off than to collide again!

- Example: Exponential Backoff
- Each subsequent failure doubles expected waiting time



Exponential Backoff Lock

```
public class Backoff implements Lock {
  AtomicBoolean state = new AtomicBoolean(false);

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

  // unlock() remains the same

}
```

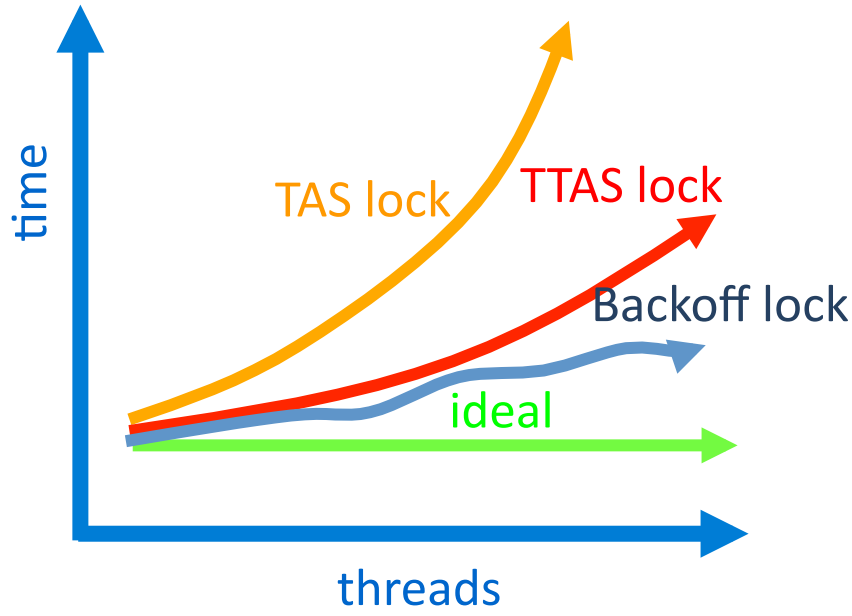
Fix minimum delay

Back off for random duration

Double maximum delay until an upper bound is reached

Backoff Lock: Performance

- The backoff log outperforms the TTAS lock!
- But it is still not ideal…



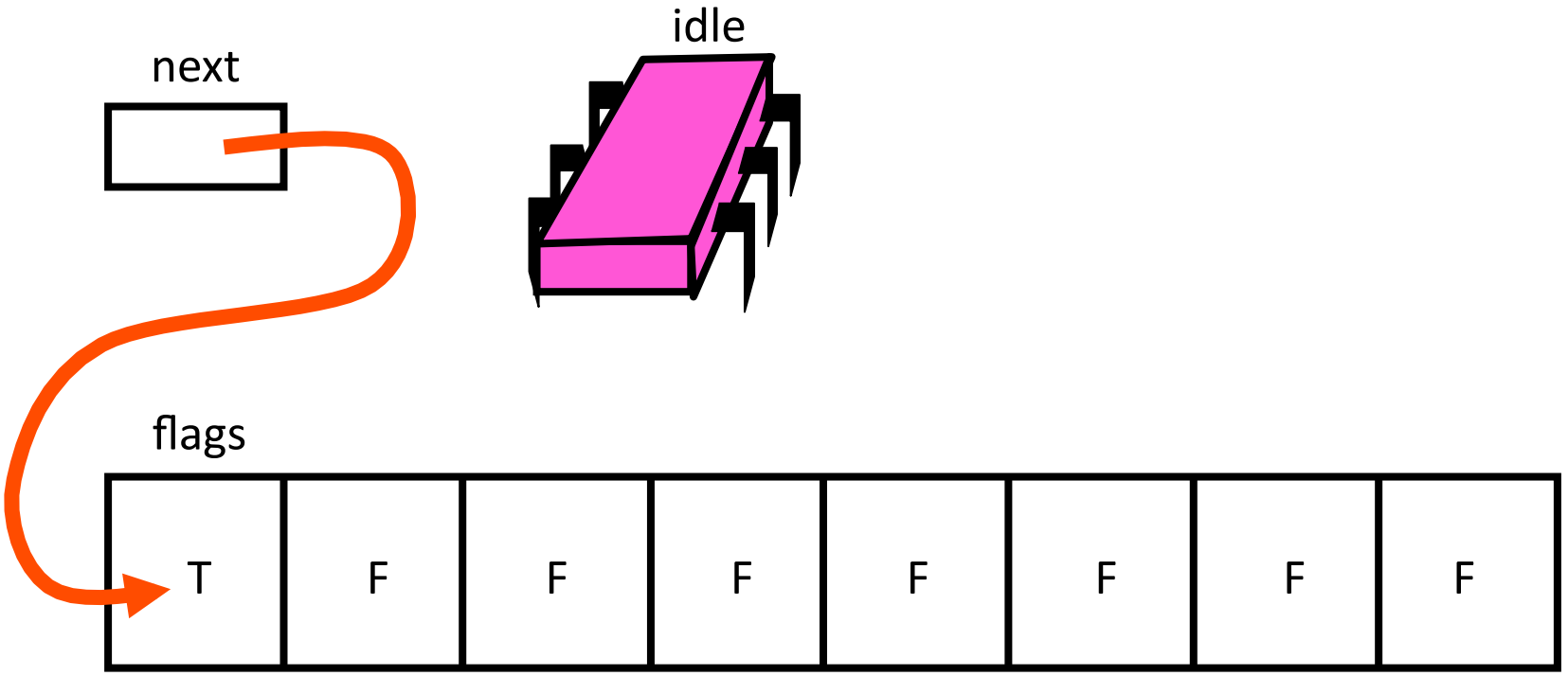
Backoff Lock: Evaluation

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

- How can we do better?
- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others

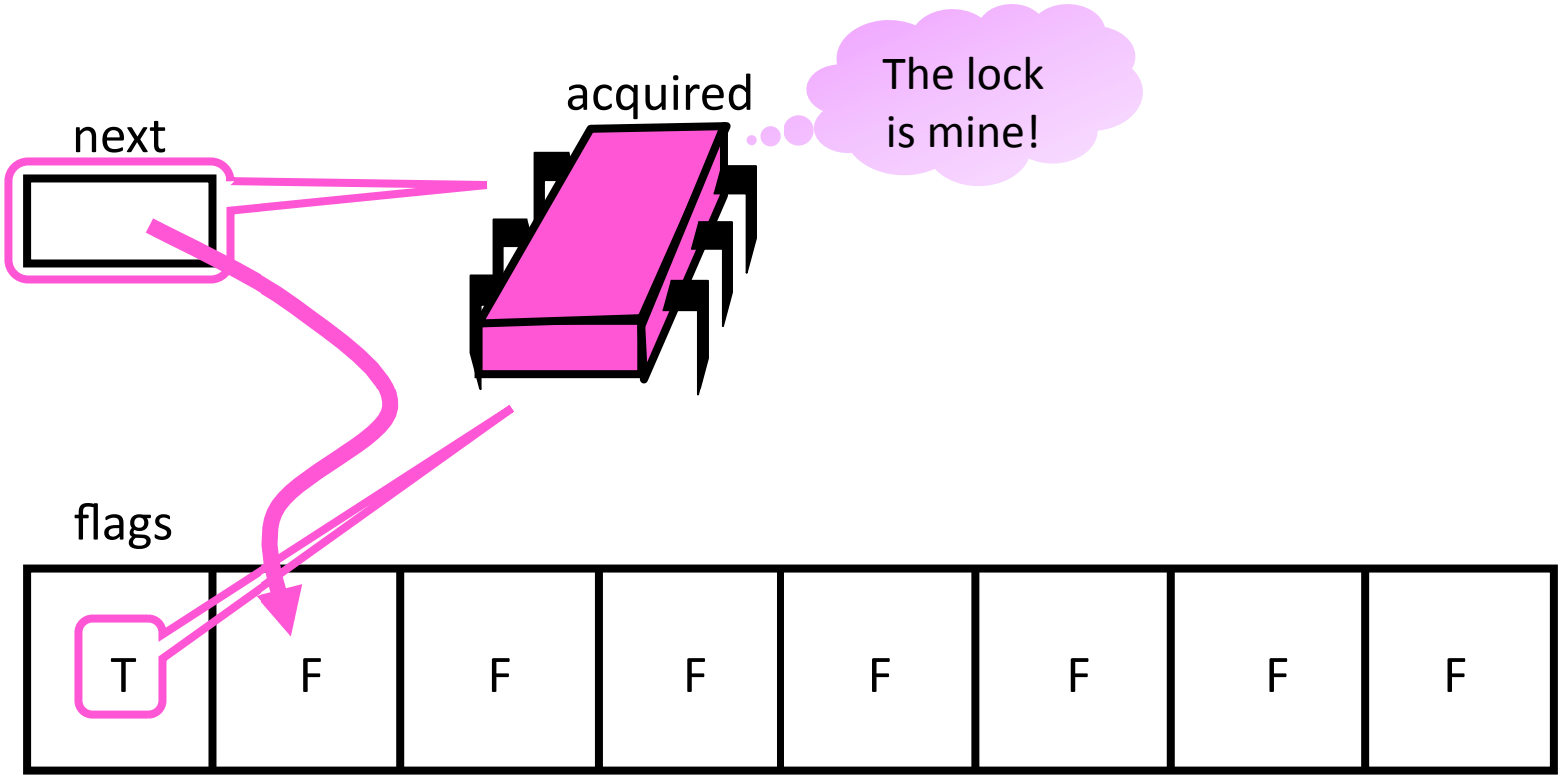
ALock: Initially

- The Anderson queue lock (ALock) is an array-based queue lock
- Threads share an atomic tail field (called next)



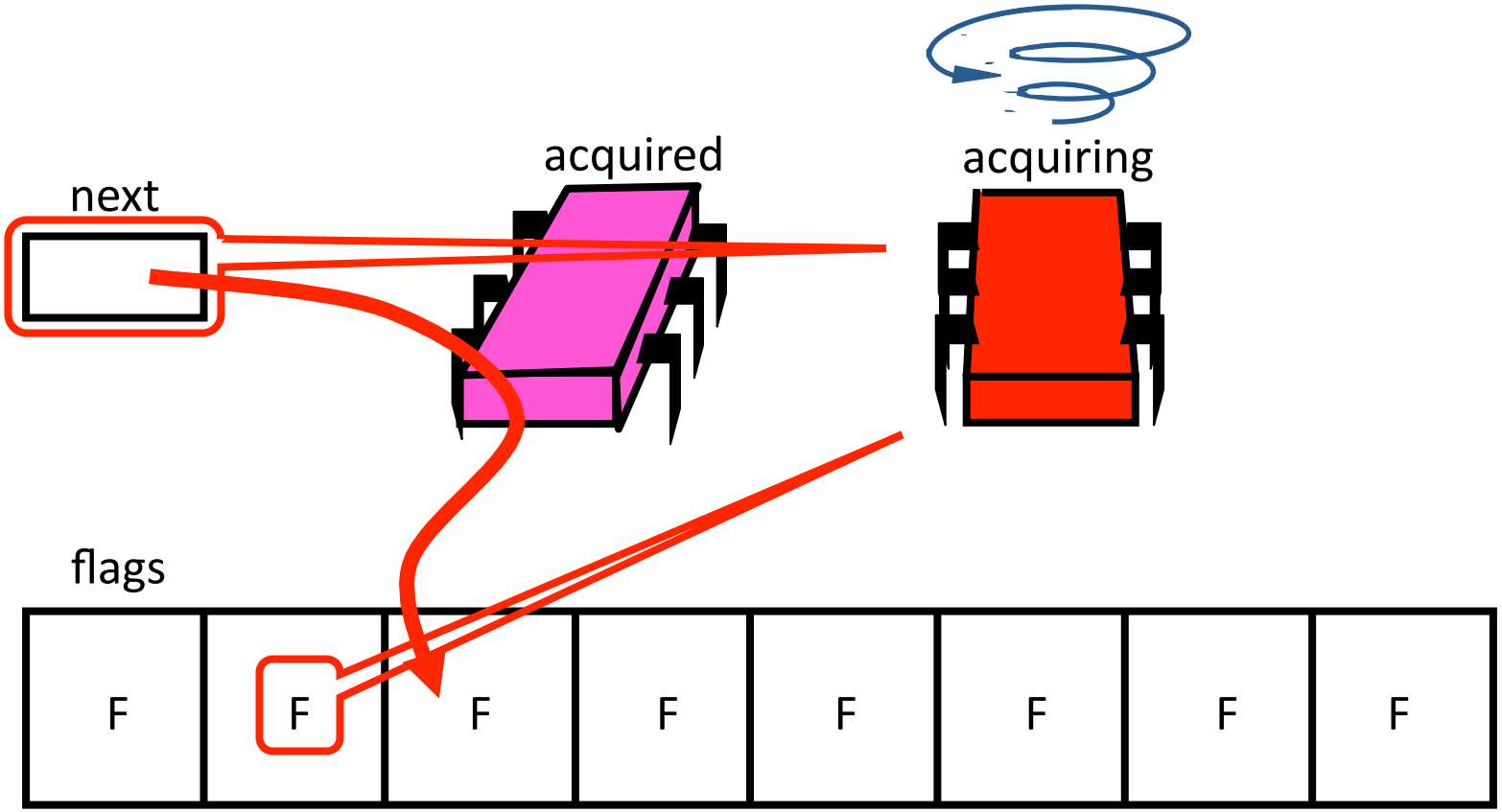
ALock: Acquiring the Lock

- To acquire the lock, each thread atomically increments the tail field
- If the flag is true, the lock is acquired
- Otherwise, spin until the flag is true



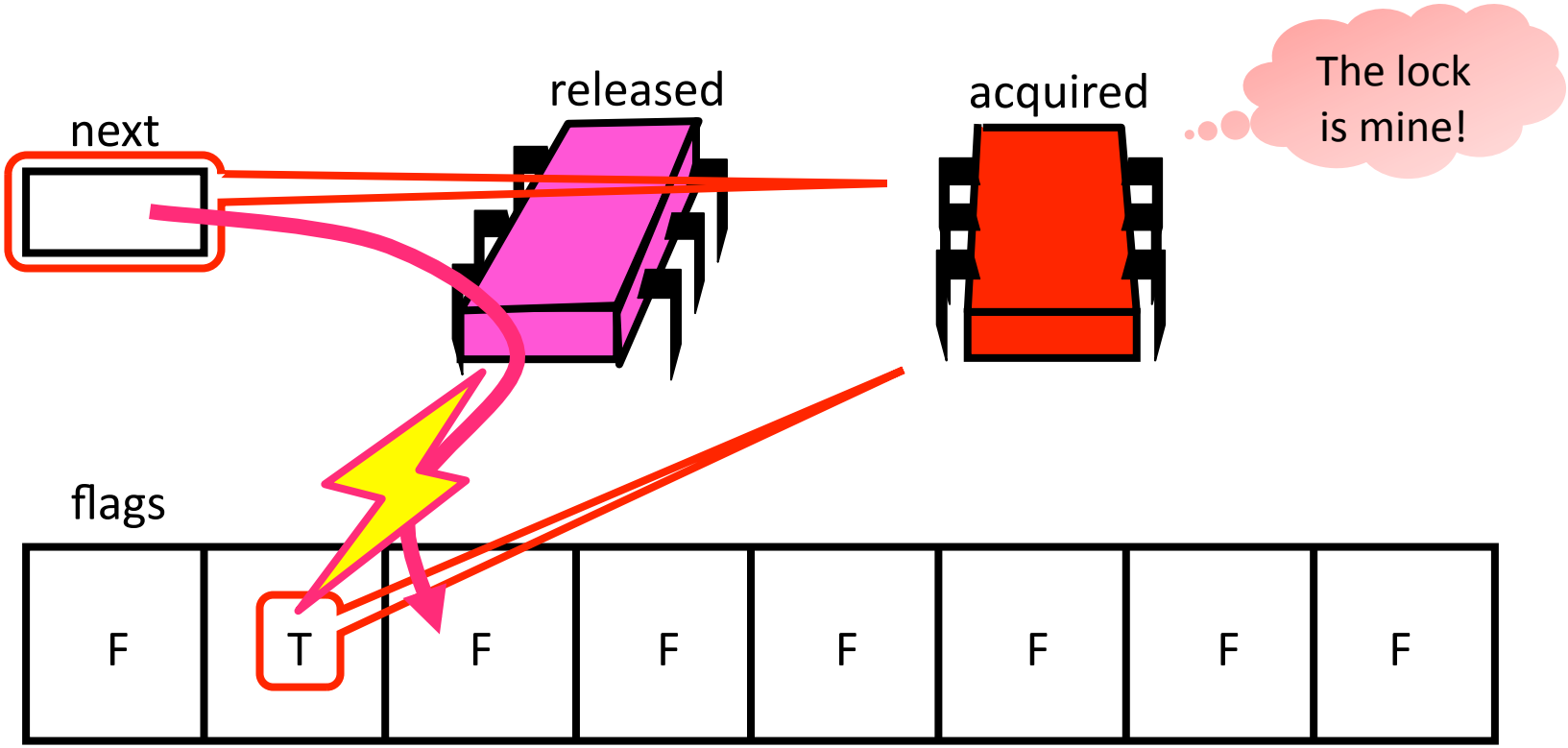
ALock: Contention

- If another thread wants to acquire the lock, it applies get&increment
- The thread spins because the flag is false



ALock: Releasing the Lock

- The first thread releases the lock by setting the next slot to true
- The second thread notices the change and gets the lock



ALock

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

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

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

One flag per thread

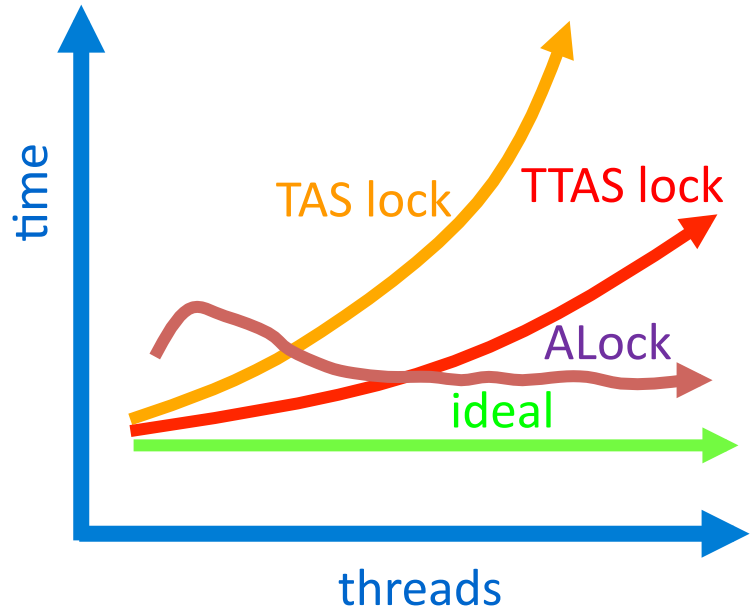
Thread-local variable

Take the next slot

Tell next thread to go

ALock: Performance

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

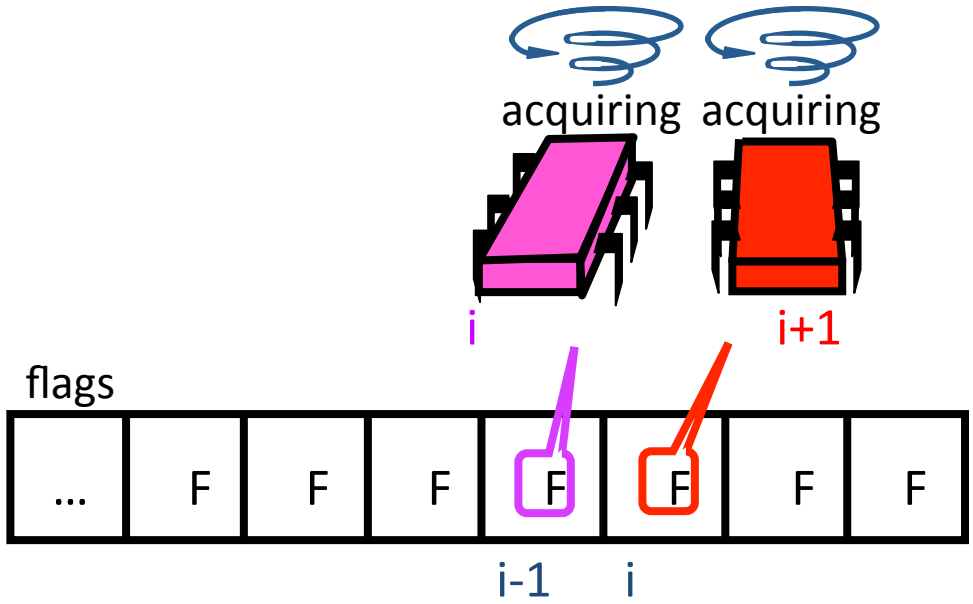


ALock: Evaluation

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

ALock: Alternative Technique

- The threads could update own flag and spin on their predecessor's flag

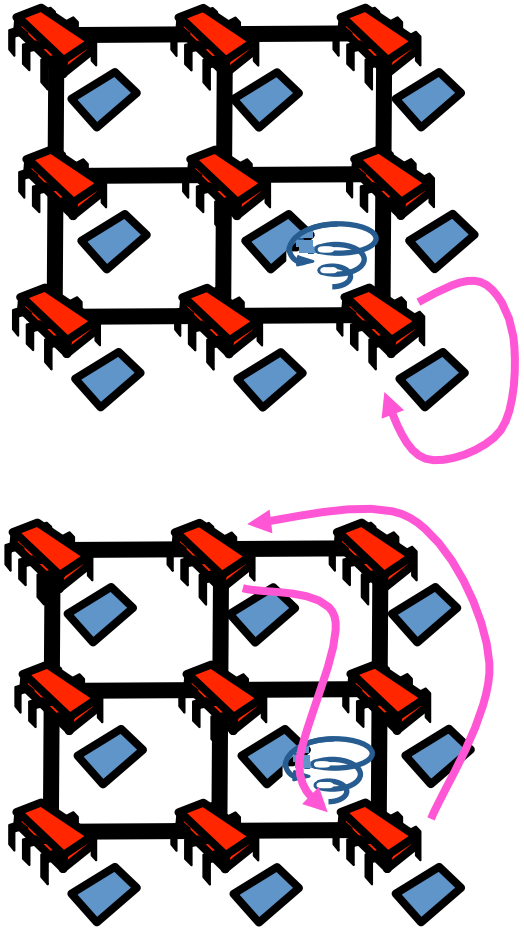


- This is basically what the CLH lock does, but using a linked list instead of an array
- Is this a good idea?

Not discussed in this lecture

NUMA Architectures

- **N**on-**U**niform **M**emory **A**rchitecture
- Illusion
  - Flat shared memory
- Truth
  - No caches (sometimes)
  - Some memory regions faster than others
- Spinning on local memory is fast
- Spinning on remote memory is slow



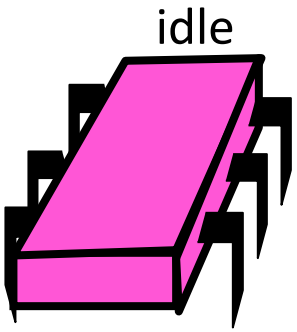
MCS Lock

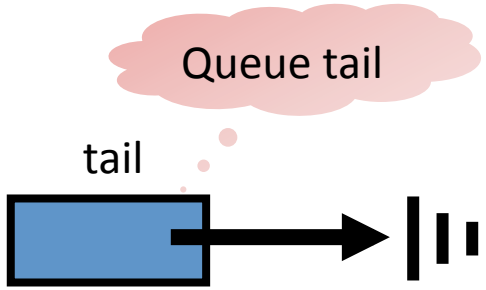
- Idea
  - Use a linked list instead of an array
    → Small, constant-sized space
  - Spin on own flag, just like the Anderson queue lock

- The space usage
  - L = number of locks
  - N = number of threads
- of the Anderson lock is O(LN)
- of the MCS lock is O(L+N)

MCS Lock: Initially

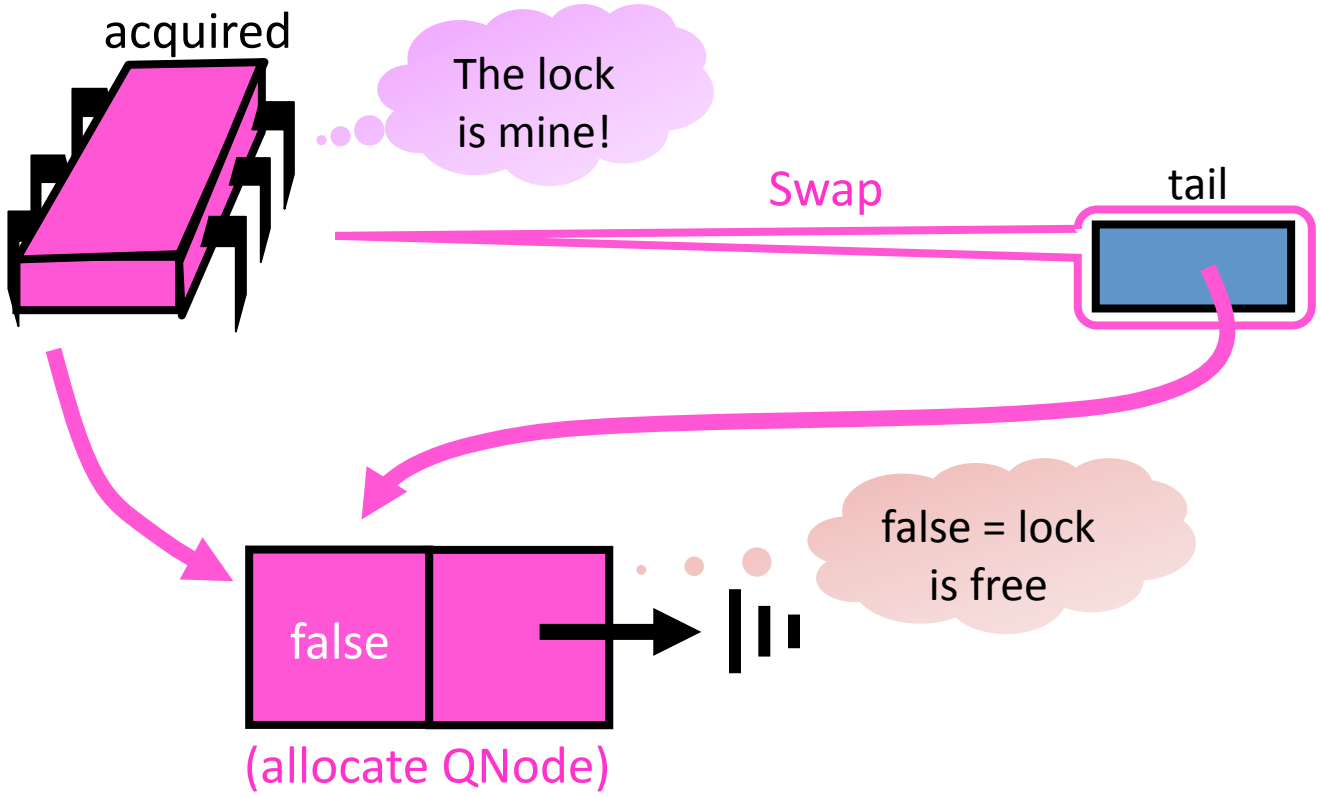
- The lock is represented as a linked list of QNodes, one per thread
- The tail of the queue is shared among all threads





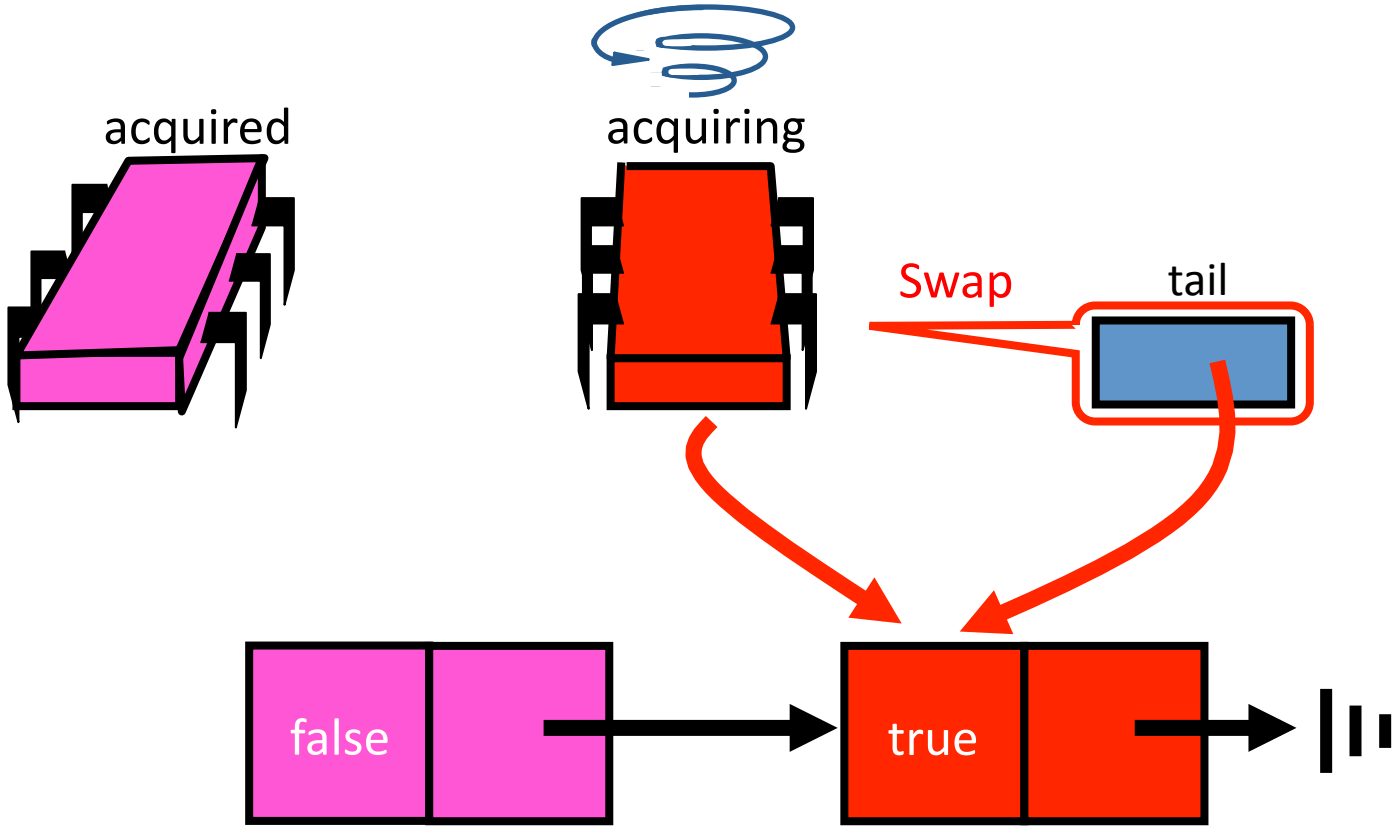
MCS Lock: Acquiring the Lock

- To acquire the lock, the thread places its QNode at the tail of the list by swapping the tail to its QNode
- If there is no predecessor, the thread acquires the lock



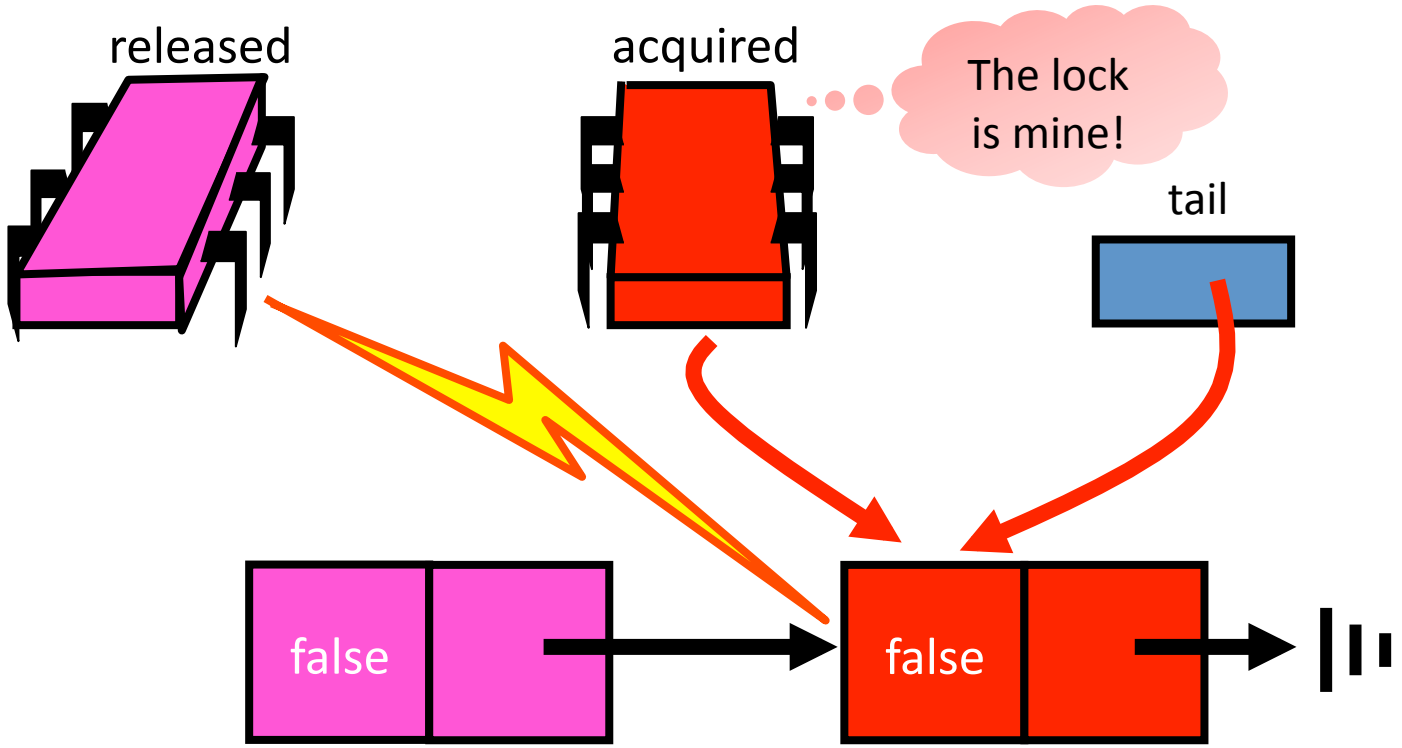
MCS Lock: Contention

- If another thread wants to acquire the lock, it again applies swap
- The thread spins on its own QNode because there is a predecessor



MCS Lock: Releasing the Lock

- The first thread releases the lock by setting its successor's QNode to false



MCS Queue Lock

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

MCS Queue Lock

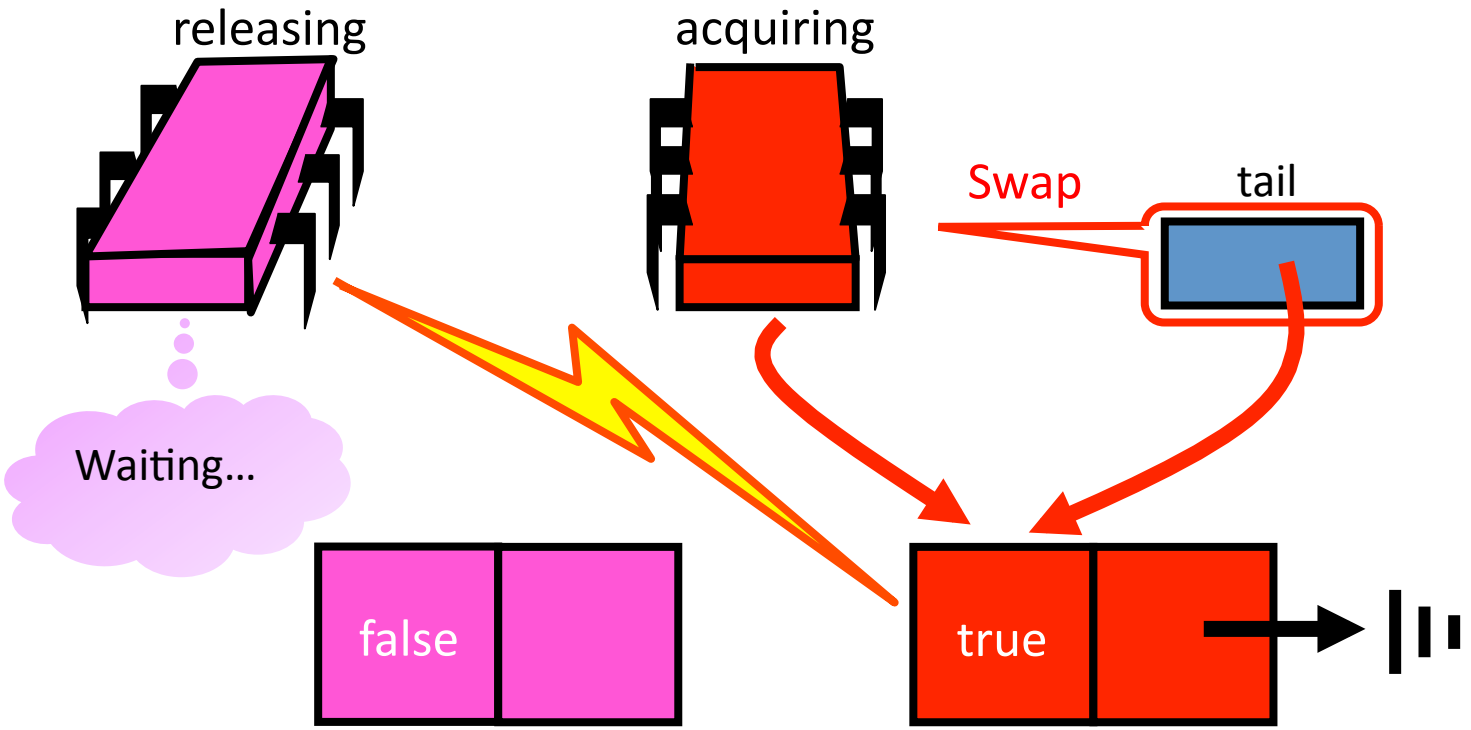
```
public class MCSLock implements Lock {
  AtomicReference tail;

  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

Fix if queue was non-empty

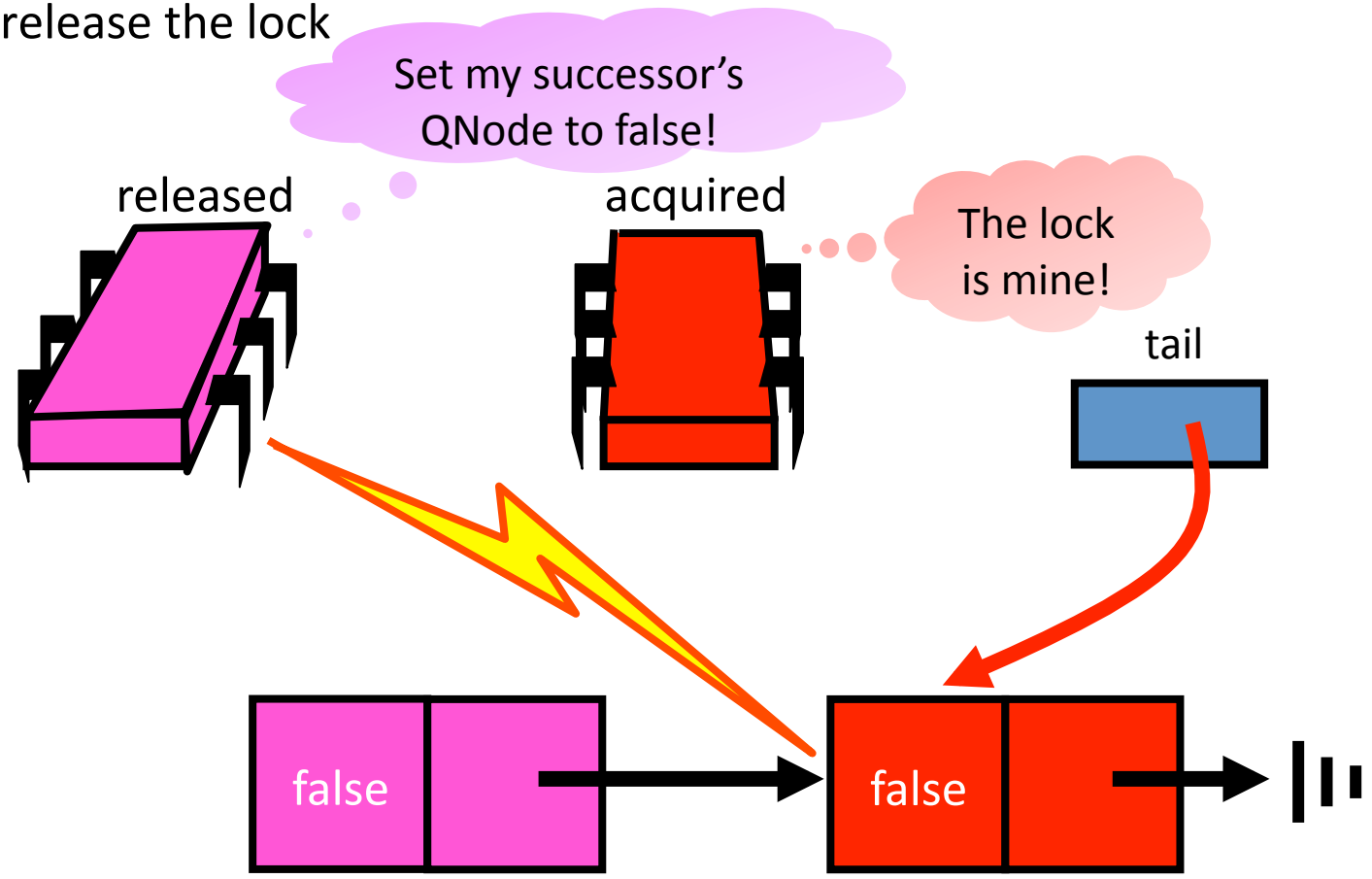
MCS Lock: Unlocking

- If there is a successor, unlock it. But, be cautious!
- Even though a QNode does not have a successor, the purple thread knows that another thread is active because tail does not point to its QNode!



MCS Lock: Unlocking Explained

- As soon as the pointer to the successor is set, the purple thread can release the lock



MCS Queue Lock

```
...

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

Missing successor?

If really no successor, return

Otherwise, wait for successor to catch up

Pass lock to successor

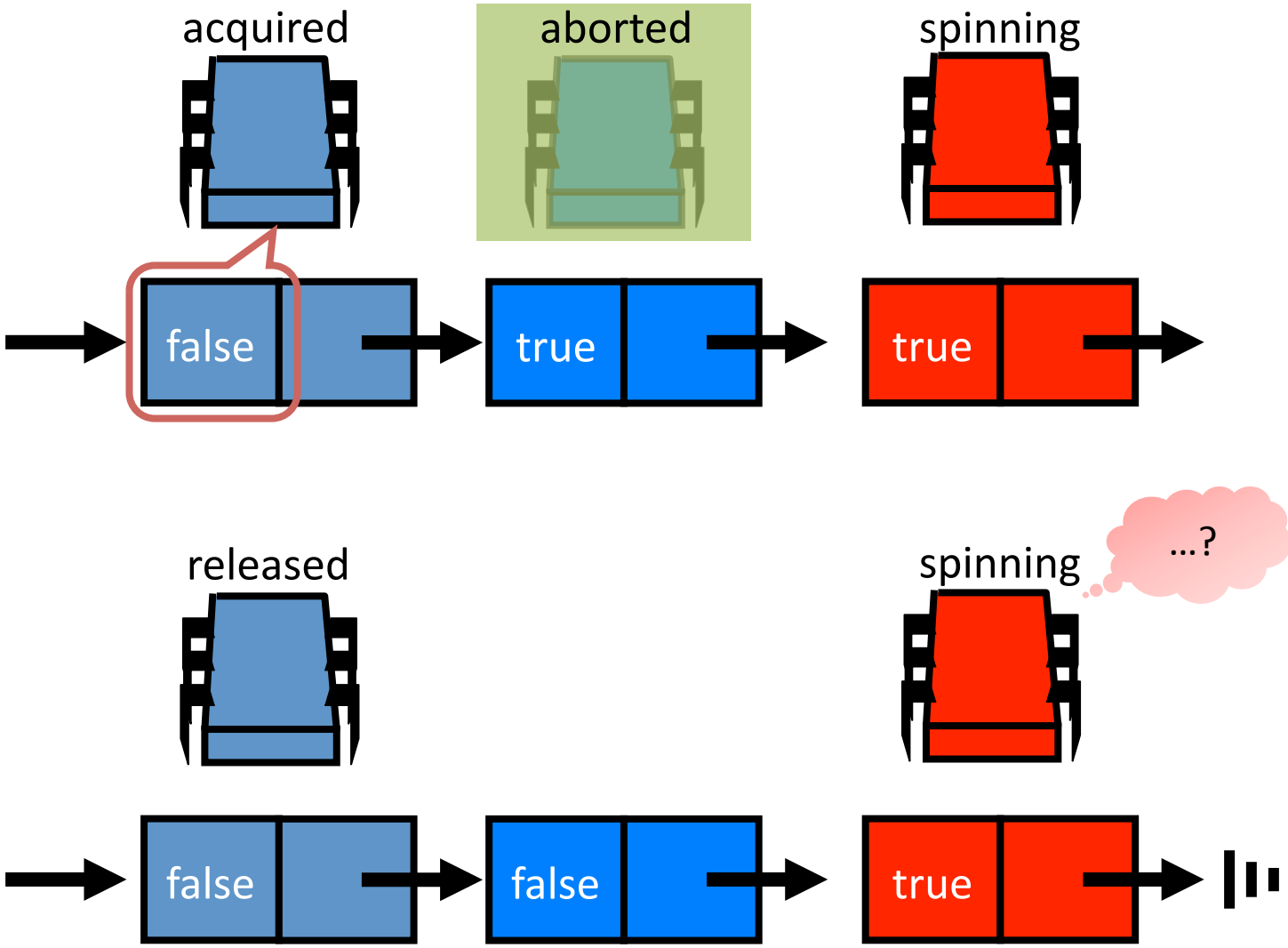
Abortable Locks

- What if you want to give up waiting for a lock?
- For example
  - Time-out
  - 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

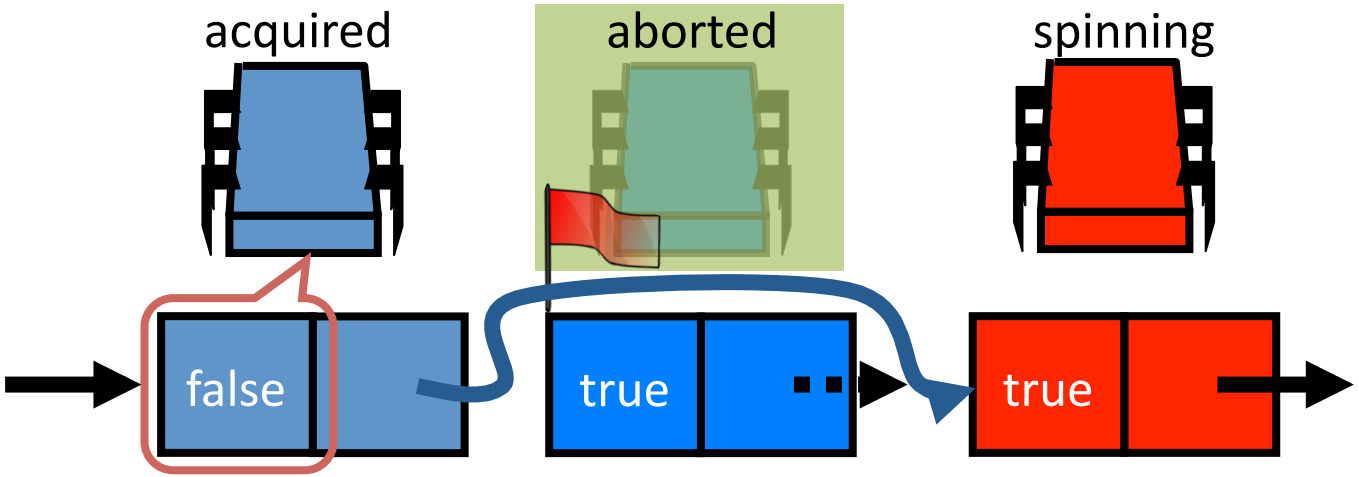
- Queue Locks
  - Can't just quit: Thread in line behind will starve
  - Need a graceful way out…

Problem with Queue Locks



Abortable MCS Lock

- A mechanism is required to recognize and remove aborted threads
  - A thread can set a flag indicating that it aborted
  - The predecessor can test if the flag is set
  - If the flag is set, its new successor is the successor's successor

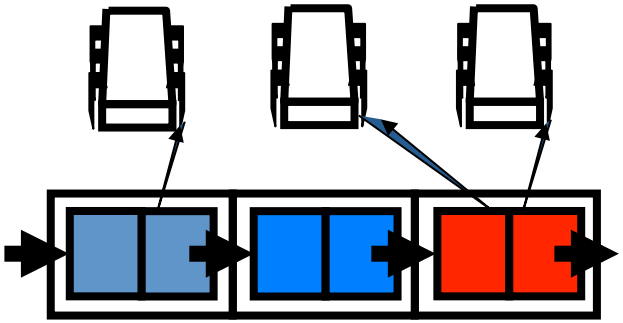


Composite Locks

- Queue locks:
  - \+ FIFO fairness, fast lock release, low contention
  - – Non-trivial abort protocols
- Backoff locks support trivial time-out protocols
  but are not scalable and may have slow lock release times

Composite lock - best of both approaches!

- Short fixed-sized array of lock nodes
- Threads randomly pick a node and try to acquire it
- Use backoff mechanism to acquire a node
- Nodes build a queue
- Use a queue lock mechanism to acquire the lock



One Lock To Rule Them All?

- TTAS+Backoff, MCS, Abortable MCS…
- Each better than others in some way
- There is not a single best solution
- Lock we pick really depends on
  - the application
  - the hardware
  - which properties are important