

# YSC3242: Parallel, Concurrent and Distributed Programming

Spin Locks and Contention

# Focus so far: Correctness and Progress

- Models
  - Accurate (we never lied to you)
  - But idealized (so we forgot to mention a few things)
- Protocols
  - Elegant
  - Important
  - But naïve

# New Focus: Performance

- Models
  - More complicated (not the same as complex!)
  - Still focus on principles (not soon obsolete)
- Protocols
  - Elegant (in their fashion)
  - Important (why else would we pay attention)
  - And realistic (your mileage may vary)

# Kinds of Architectures

- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

# Kinds of Architectures

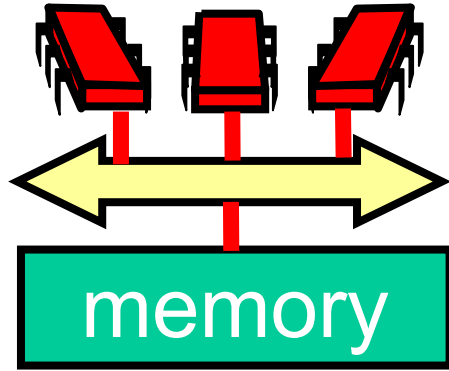
- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data

Our space

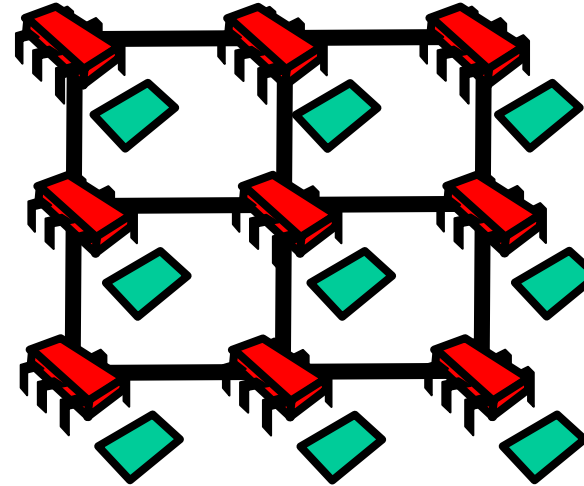


- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

# MIMD Architectures



**Shared Bus**



**Distributed**

- Memory Contention
- Communication Contention
- Communication Latency

# Today: Revisit Mutual Exclusion

- Performance, not just correctness
- Proper use of multiprocessor architectures
- A collection of locking algorithms...

# 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

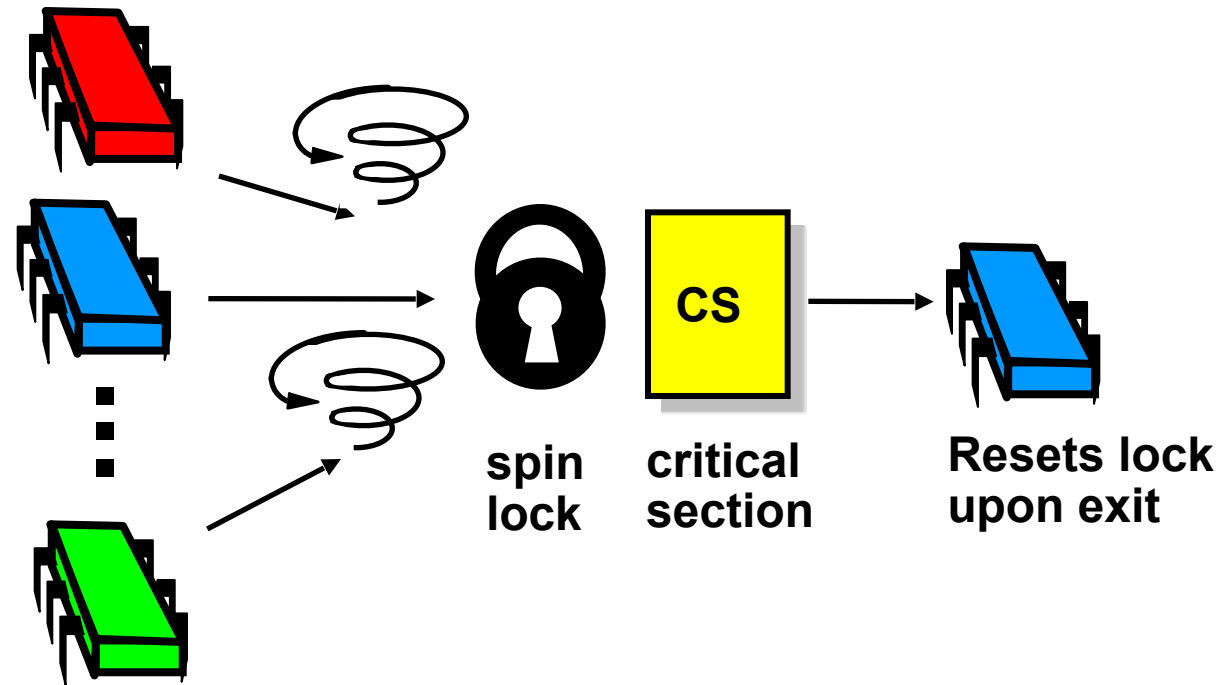


# What Should you do if you can't get a lock?

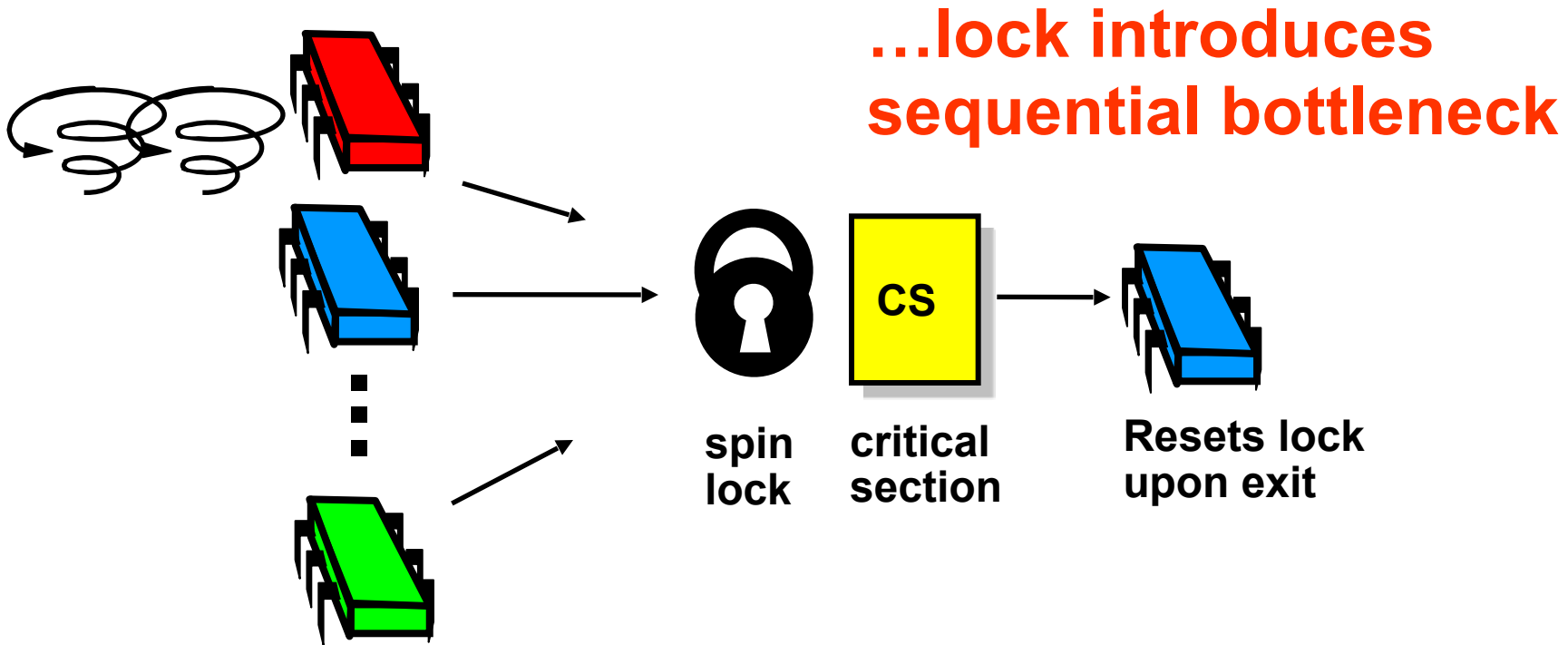
- Keep trying
  - “spin” or “busy-wait”
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

our focus today

# Basic Spin-Lock

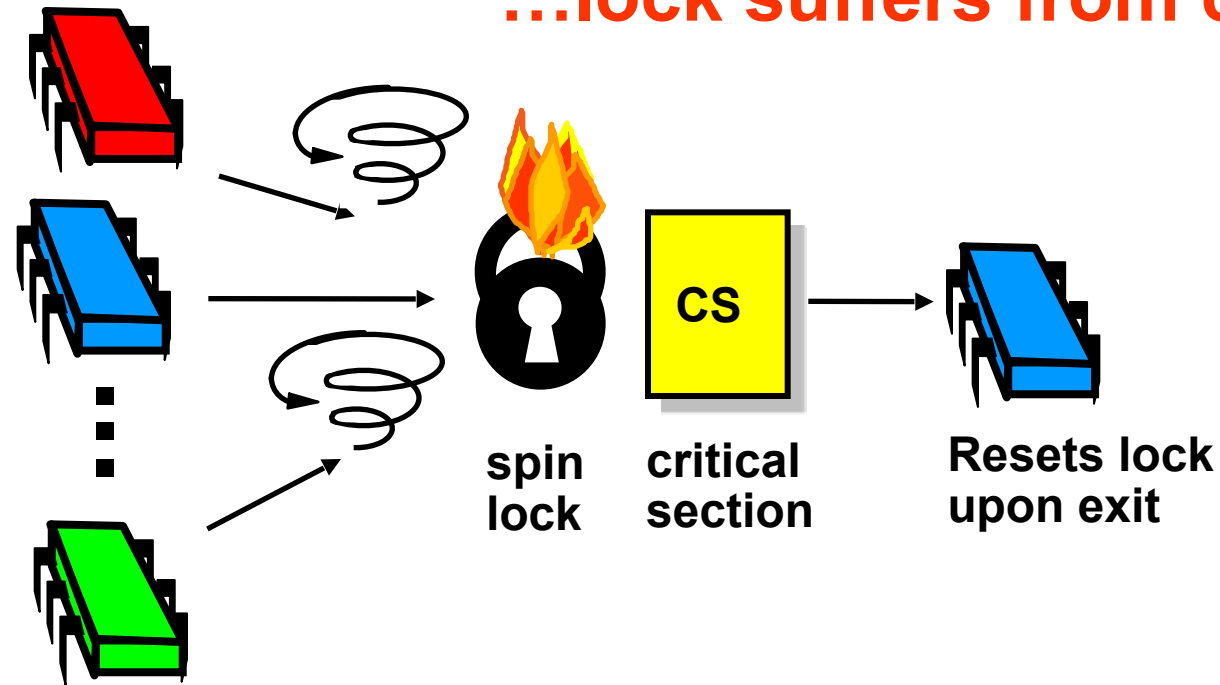


# Basic Spin-Lock



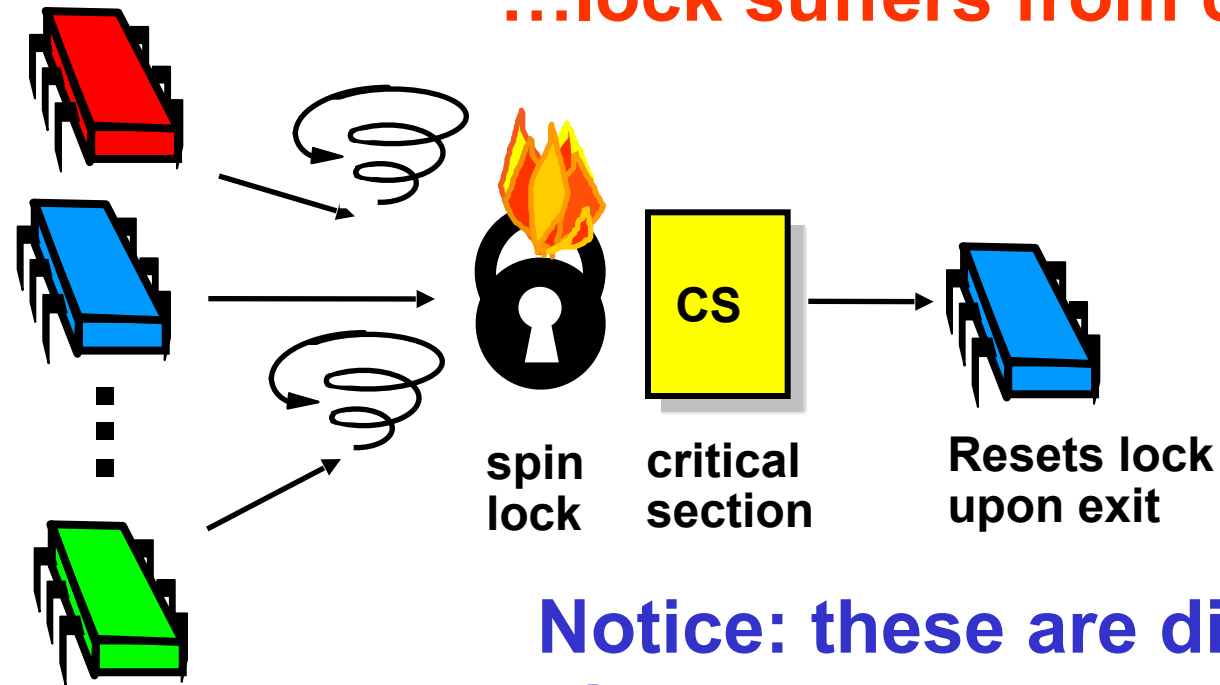
# Basic Spin-Lock

...lock suffers from contention



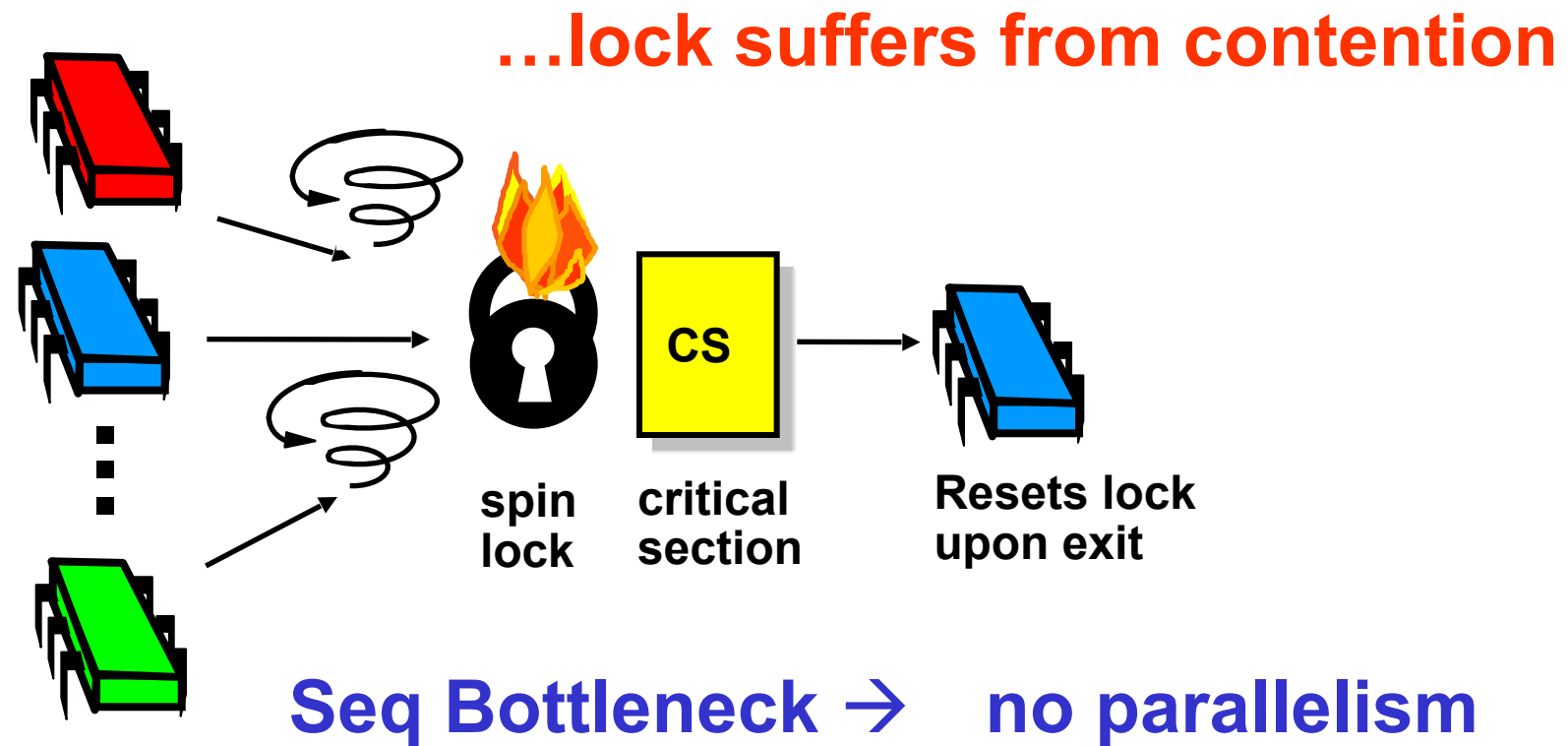
# Basic Spin-Lock

...lock suffers from contention

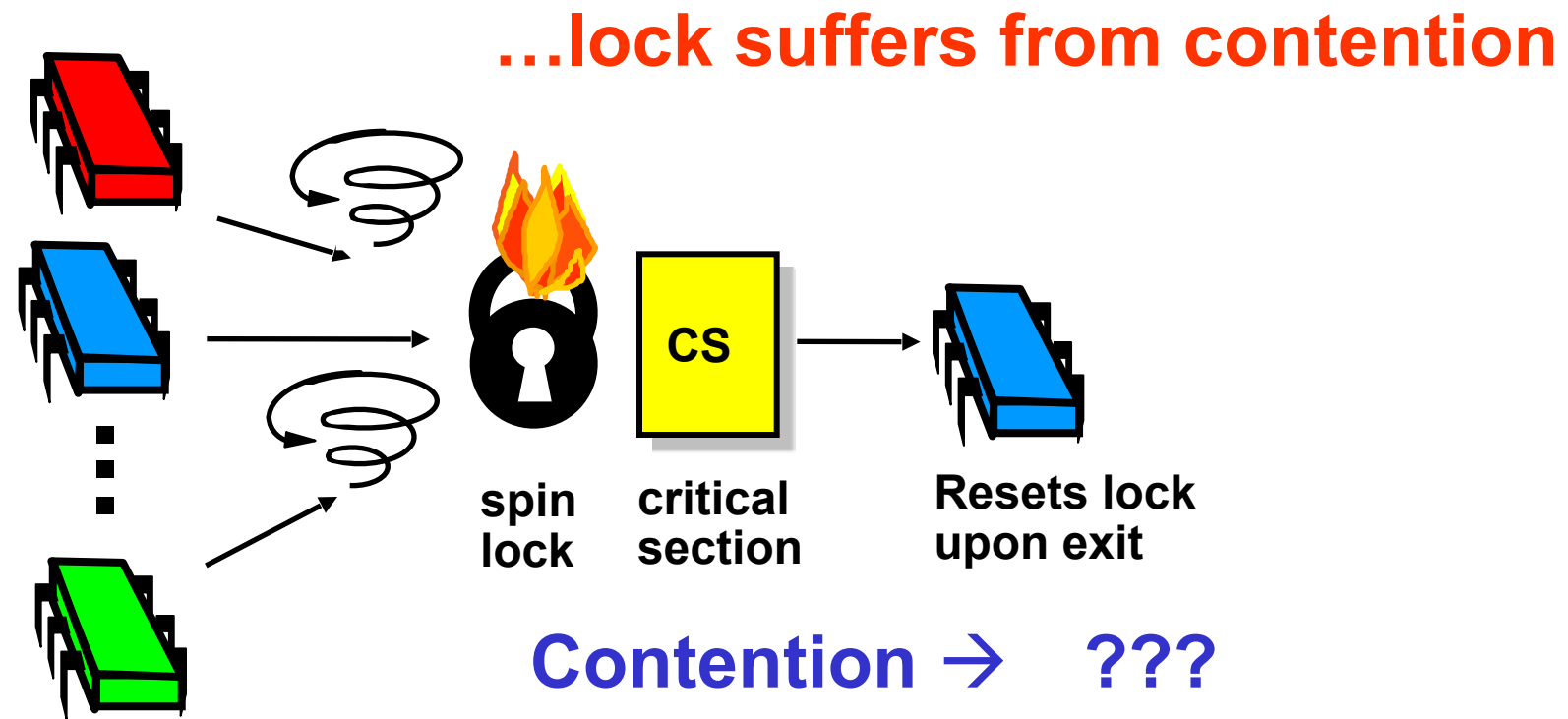


Notice: these are distinct phenomena

# Basic Spin-Lock



# Basic Spin-Lock







# Review: Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap **true** with current value
  - Return value tells if prior value was **true** or **false**
- Can reset just by writing **false**
- TAS aka “getAndSet”

# Review: Test-and-Set

```
class AtomicBoolean {  
    var value: Boolean  
  
    def getAndSet(newValue: Boolean) =  
        this.synchronized {  
            val prior = value  
            value = newValue  
            prior  
        }  
}
```

# Review: Test-and-Set

```
class AtomicBoolean {  
    var value: Boolean  
  
    def getAndSet(newValue: Boolean) =  
        this.synchronized {  
            val prior = value  
            value = newValue  
            prior  
        }  
}
```

**Package**

**java.util.concurrent.atomic**

# Review: Test-and-Set

```
class AtomicBoolean {  
    var value: Boolean  
  
    def getAndSet(newValue: Boolean) =  
        this.synchronized {  
            val prior = value  
            value = newValue  
            prior  
        }  
}
```

**Swap old and new  
values**

# Review: Test-and-Set

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

# Review: Test-and-Set

```
val lock = new AtomicBoolean(false)
```

```
val prior = lock.getAndSet(true)
```

**Swapping in `true` is called  
“test-and-set” or TAS**

# Test-and-Set Locks

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

# Test-and-set Lock

```
class TASLock extends SpinLock {  
    val state = new AtomicBoolean(false)  
  
    override def lock() = {  
        while(state.getAndSet(true)) {  
            // spin  
        }  
    }  
  
    override def unlock() = {  
        state.set(false)  
    }  
}
```



# Test-and-set Lock

```
class TASLock extends SpinLock {  
    val state = new AtomicBoolean(false)  
  
    override def lock() = {  
        while (state.getAndSet(true)) {  
            // spin  
        }  
    }  
  
    override def Lock state is AtomicBoolean  
        state.set(false)  
    }  
}
```

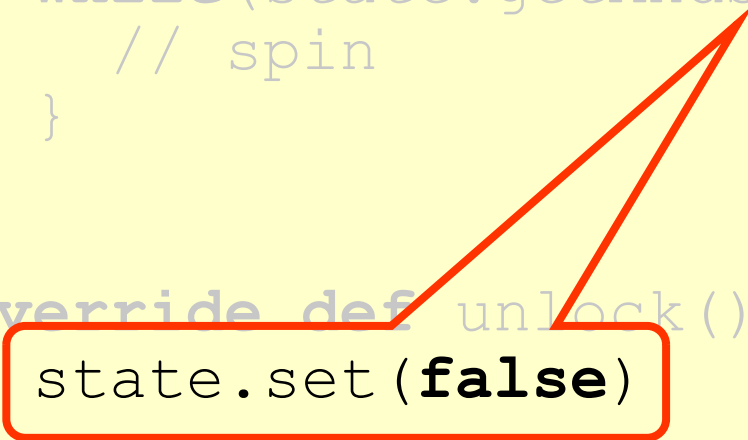
# Test-and-set Lock

```
class TASLock extends SpinLock {  
  val state = new AtomicBoolean(false)  
  
  override def lock() = {  
    while (state.getAndSet(true)) {  
      // spin  
    }  
  }  
  
  override def unlock() = {  
    state.set(false)  
  }  
}
```

**Keep trying until lock acquired**

# Test-and-set Lock

```
class TASLock extends SpinLock {  
  val st  
  Release lock by resetting  
  state to false  
  override while (state.getAndSet(true)) {  
    // spin  
  }  
  override def unlock() = {  
    state.set(false)  
  }  
}
```



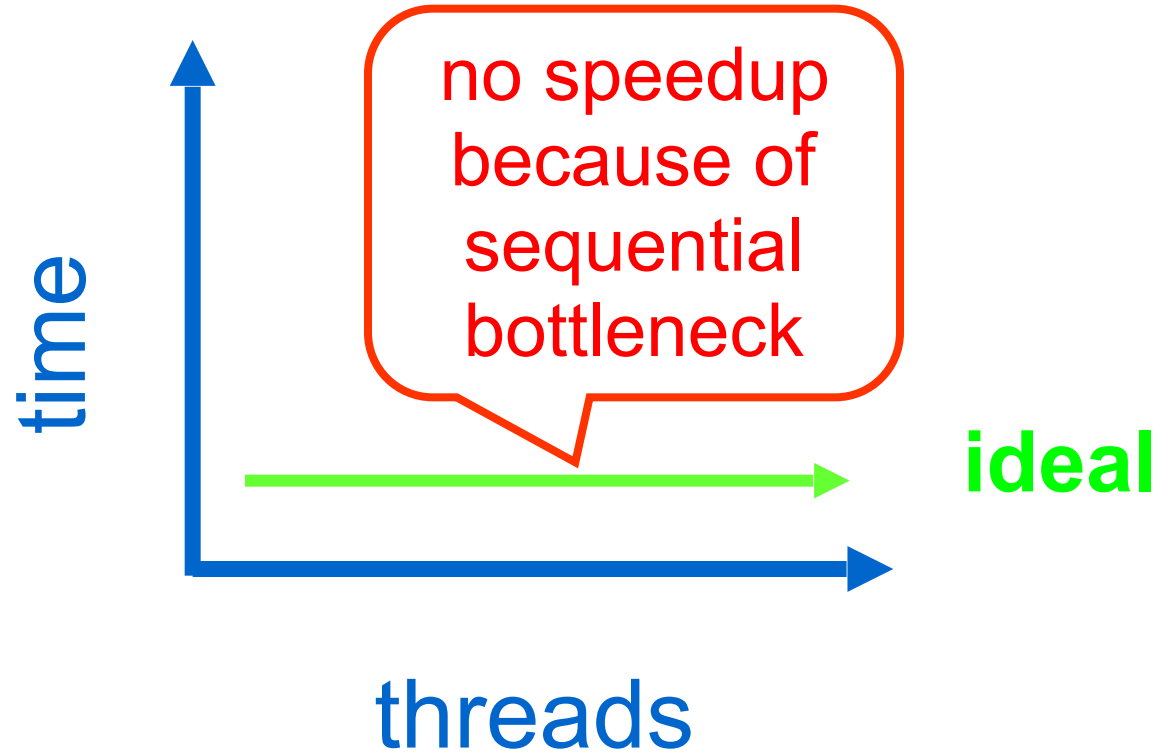
# Space Complexity

- TAS spin-lock has small “footprint”
- N thread spin-lock uses  $O(1)$  space
- As opposed to  $O(n)$  Peterson/Bakery
- How did we overcome the  $\Omega(n)$  lower bound?
- We used a RMW operation...

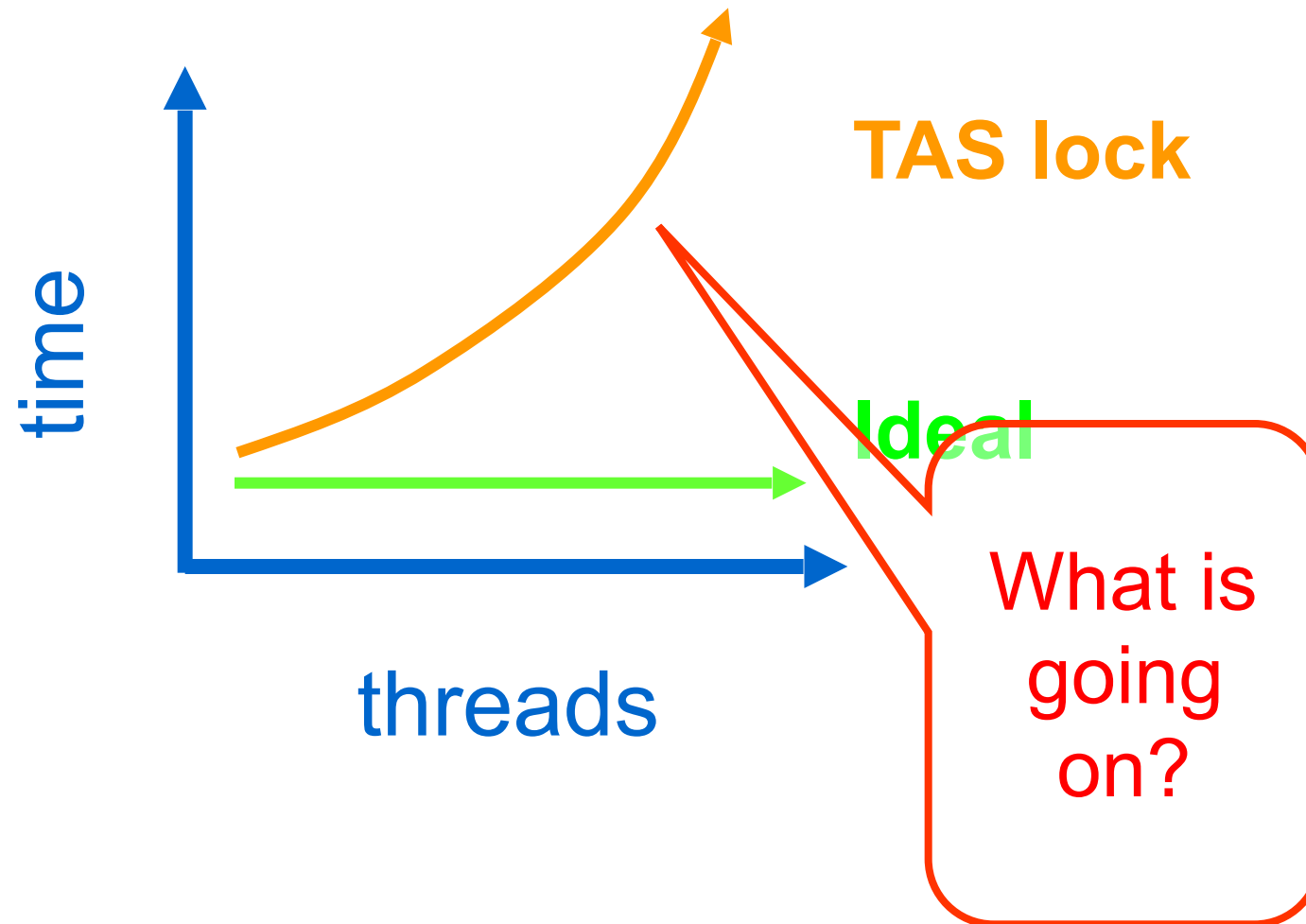
# Performance

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

# Graph



# Mystery #1



# Test-and-Test-and-Set Locks

- Lurking stage
  - Wait until lock “looks” free
  - Spin while read returns **true** (lock taken)
- Pouncing state
  - As soon as lock “looks” available
  - Read returns **false** (lock free)
  - Call TAS to acquire lock
  - If TAS loses, back to lurking



# Test-and-test-and-set Lock

```
class TTASLock extends SpinLock {  
    val state = new AtomicBoolean(false)  
  
    override def lock(): Unit = {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true)) {  
                return  
            }  
        }  
    }  
    ...  
}
```

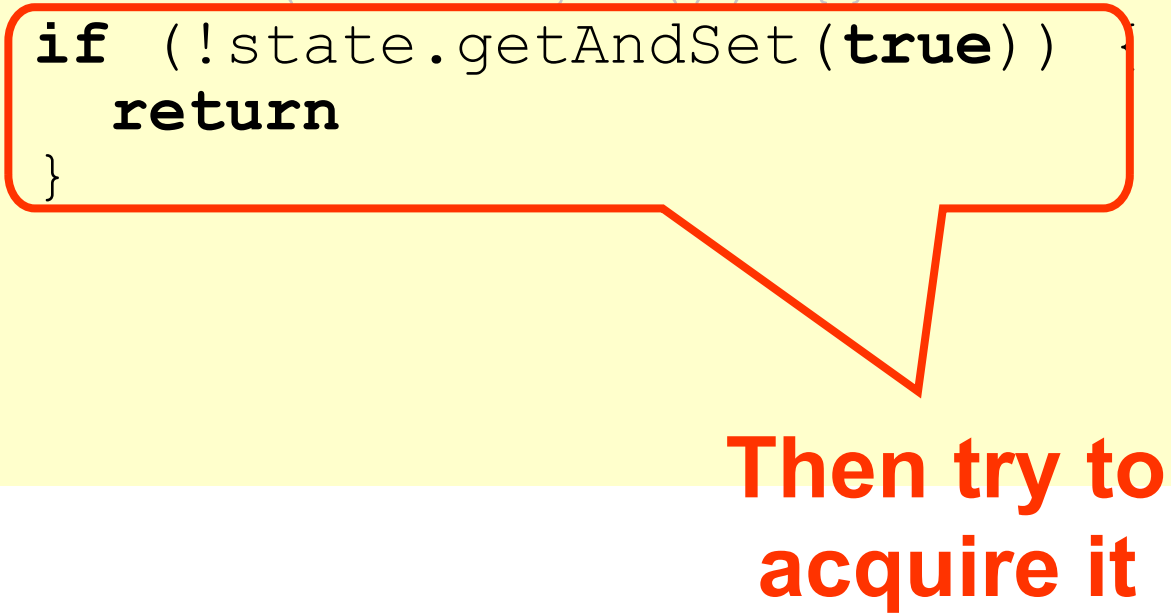
# Test-and-test-and-set Lock

```
class TTASLock extends SpinLock {  
  val state = new AtomicBoolean(false)  
  
  override def lock(): Unit = {  
    while (true) {  
      while (state.get()) {}  
      if (!state.getAndSet(true)) {  
        return  
      }  
    }  
  }  
  ...  
}
```

**Wait until lock looks free**

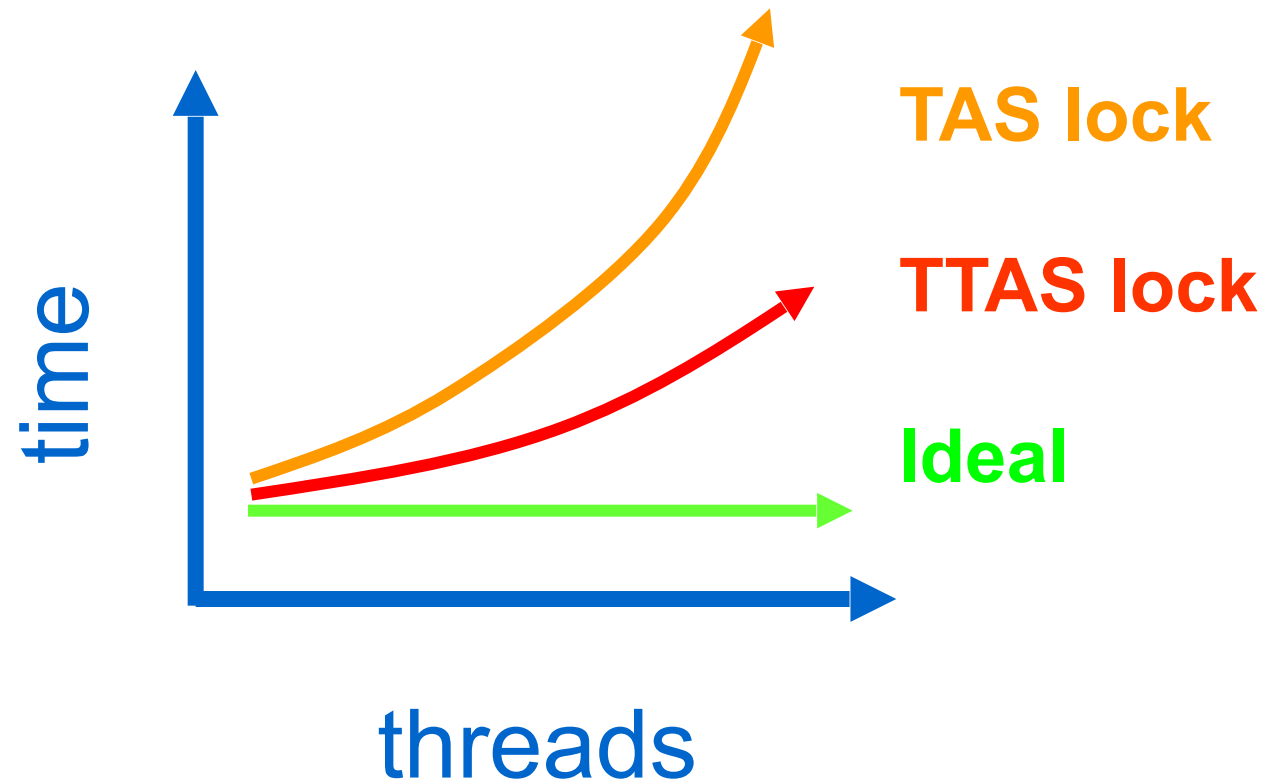
# Test-and-test-and-set Lock

```
class TTASLock extends SpinLock {  
  val state = new AtomicBoolean(false)  
  
  override def lock(): Unit = {  
    while (true) {  
      while (state.get()) {}  
      if (!state.getAndSet(true))  
        return  
    }  
  }  
  ...  
}
```



Then try to  
acquire it

## Mystery #2



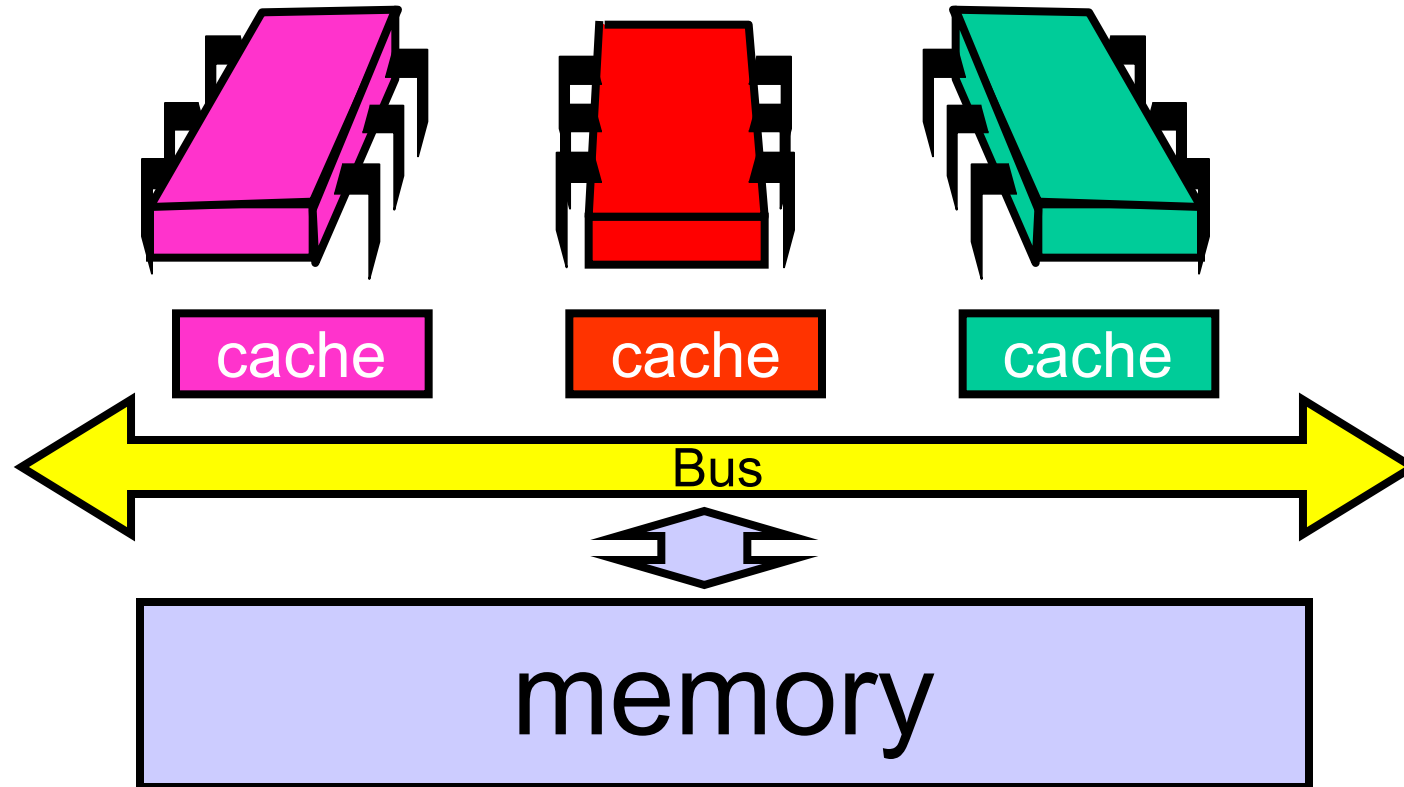
# Mystery

- Both
  - TAS and TTAS
  - Do the same thing (in our model)
- Except that
  - TTAS performs much better than TAS
  - Neither approaches ideal

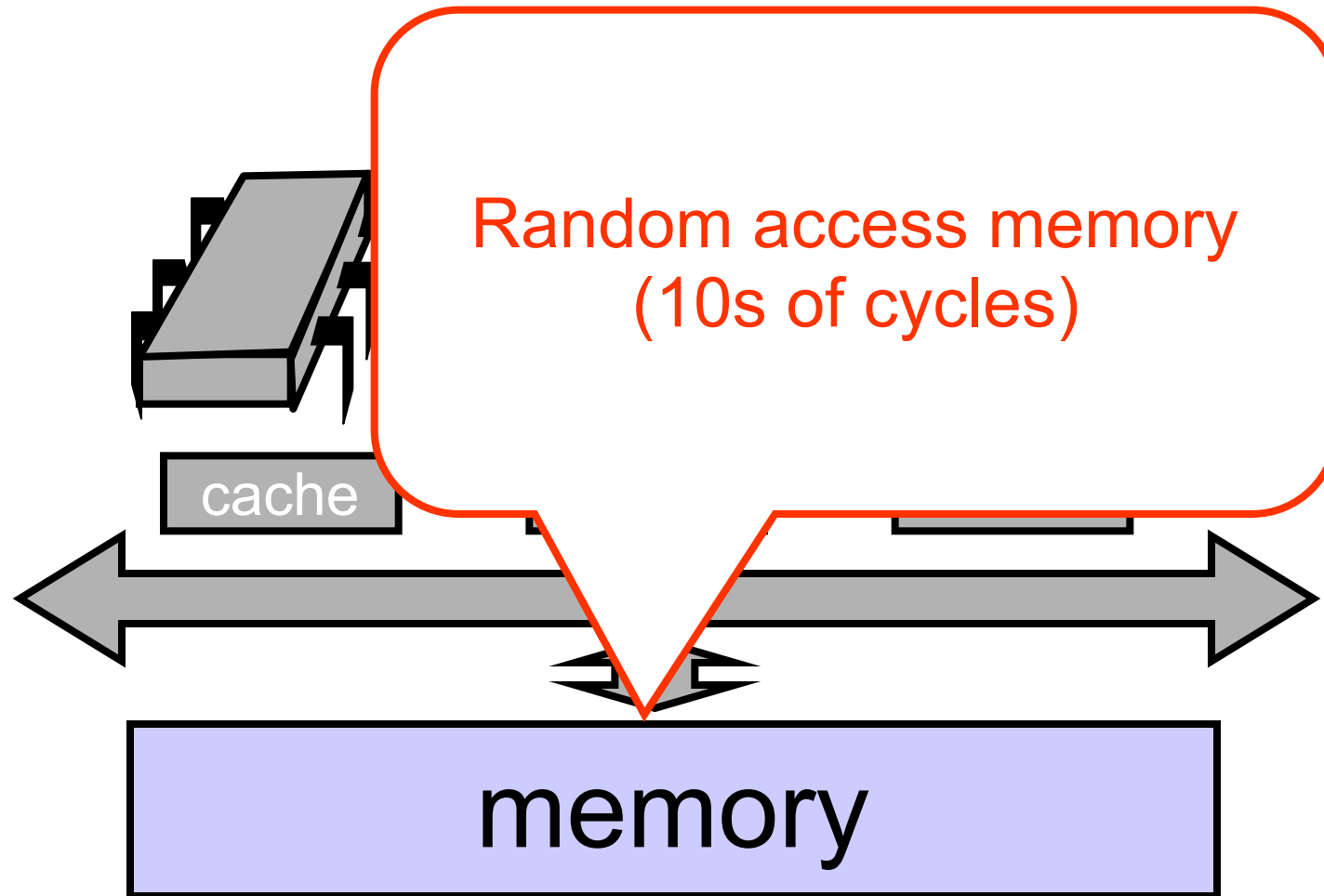
# Opinion

- Our memory abstraction is broken
- TAS & TTAS methods
  - Are provably the same (in our model)
  - Except they aren't (in field tests)
- Need a more detailed model ...

# Bus-Based Architectures



# Bus-Based Architectures

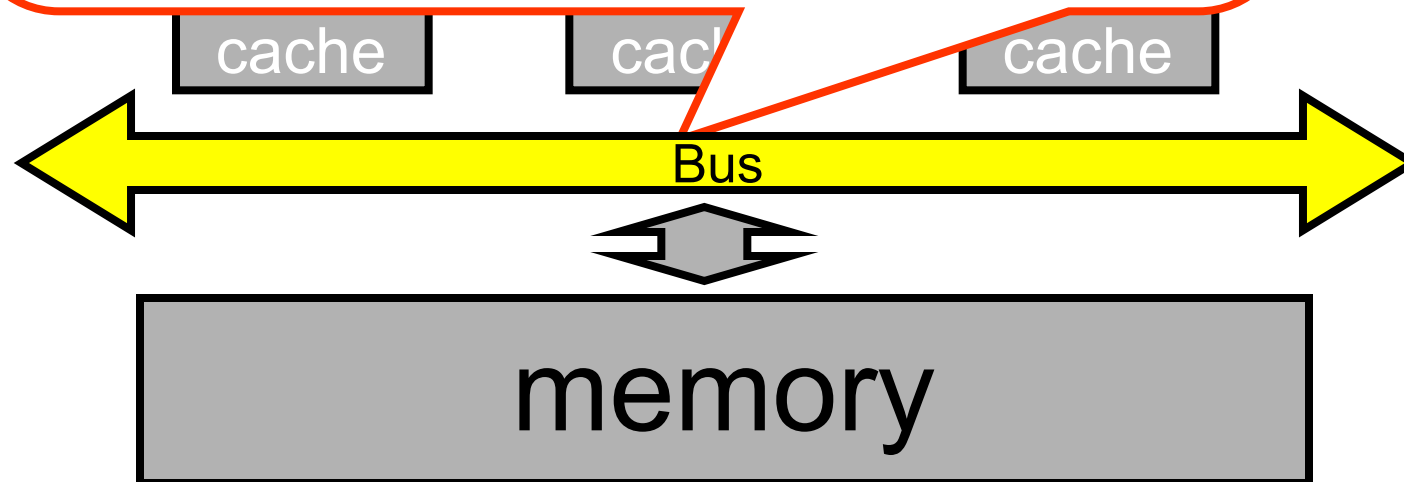




# Bus-Based Architectures

## Shared Bus

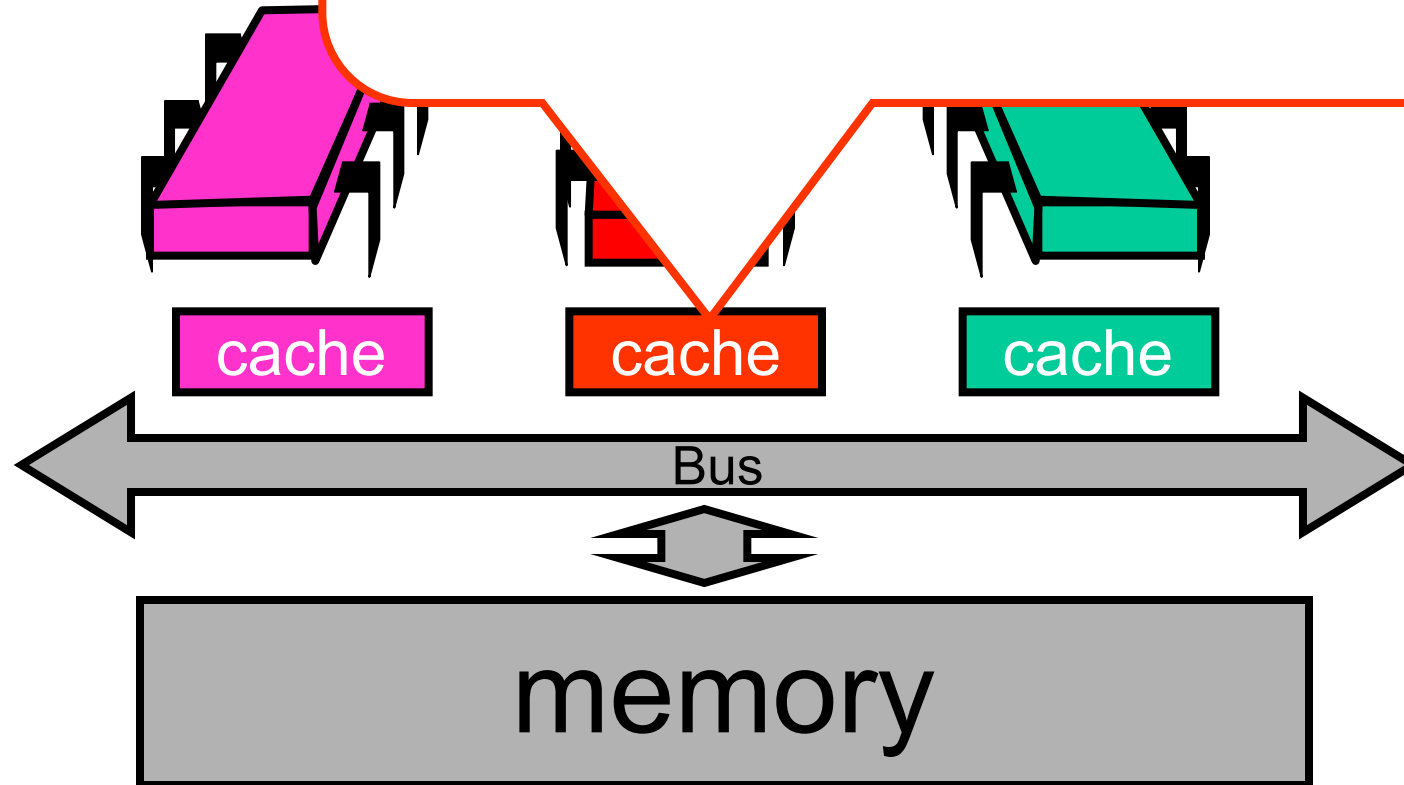
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”



Bus-

## Per-Processor Caches

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



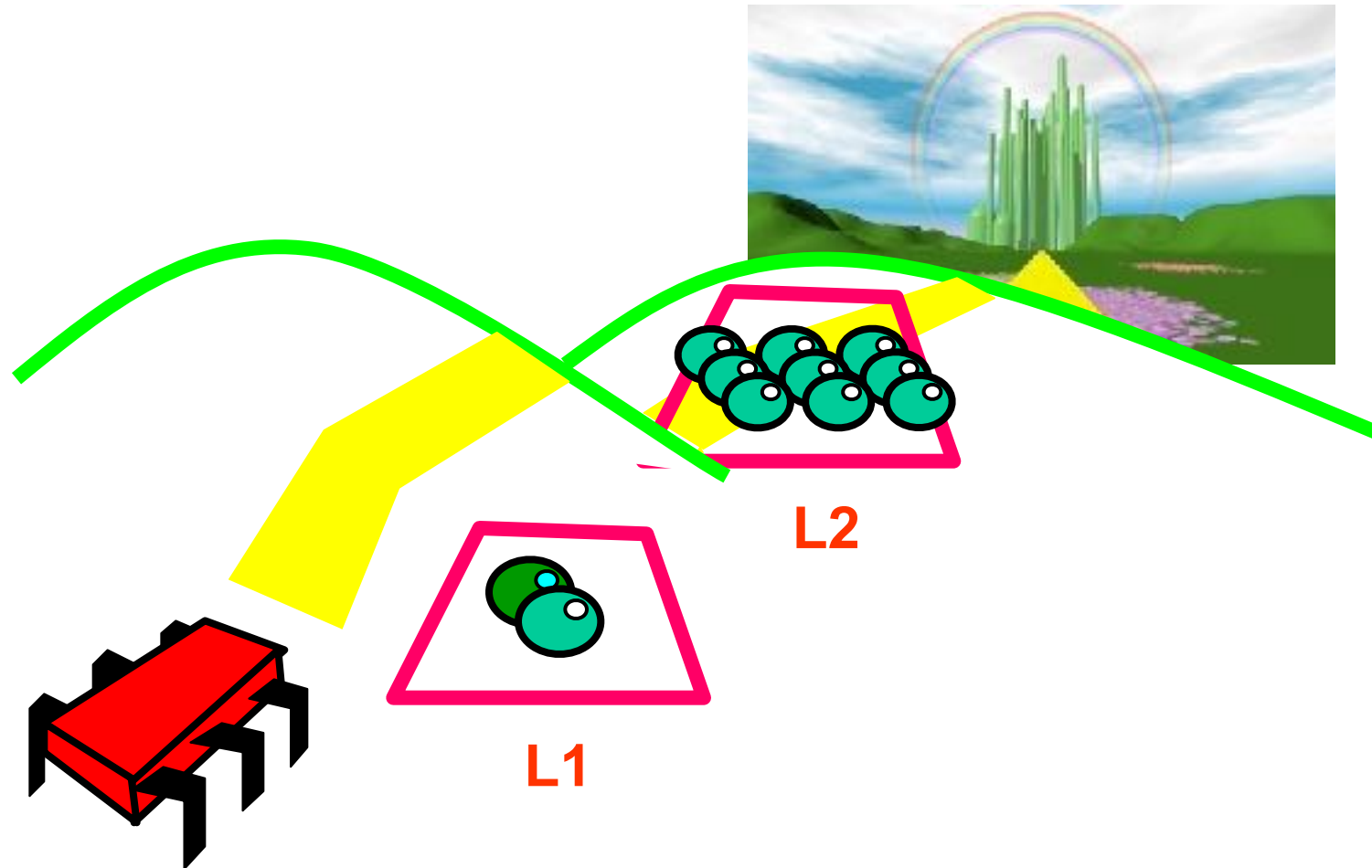
# Granularity

- Caches operate at a larger granularity than a word (32 or 64 bits)
- Cache line: fixed-size block containing of neighbouring words (today 64 or 128 bytes)

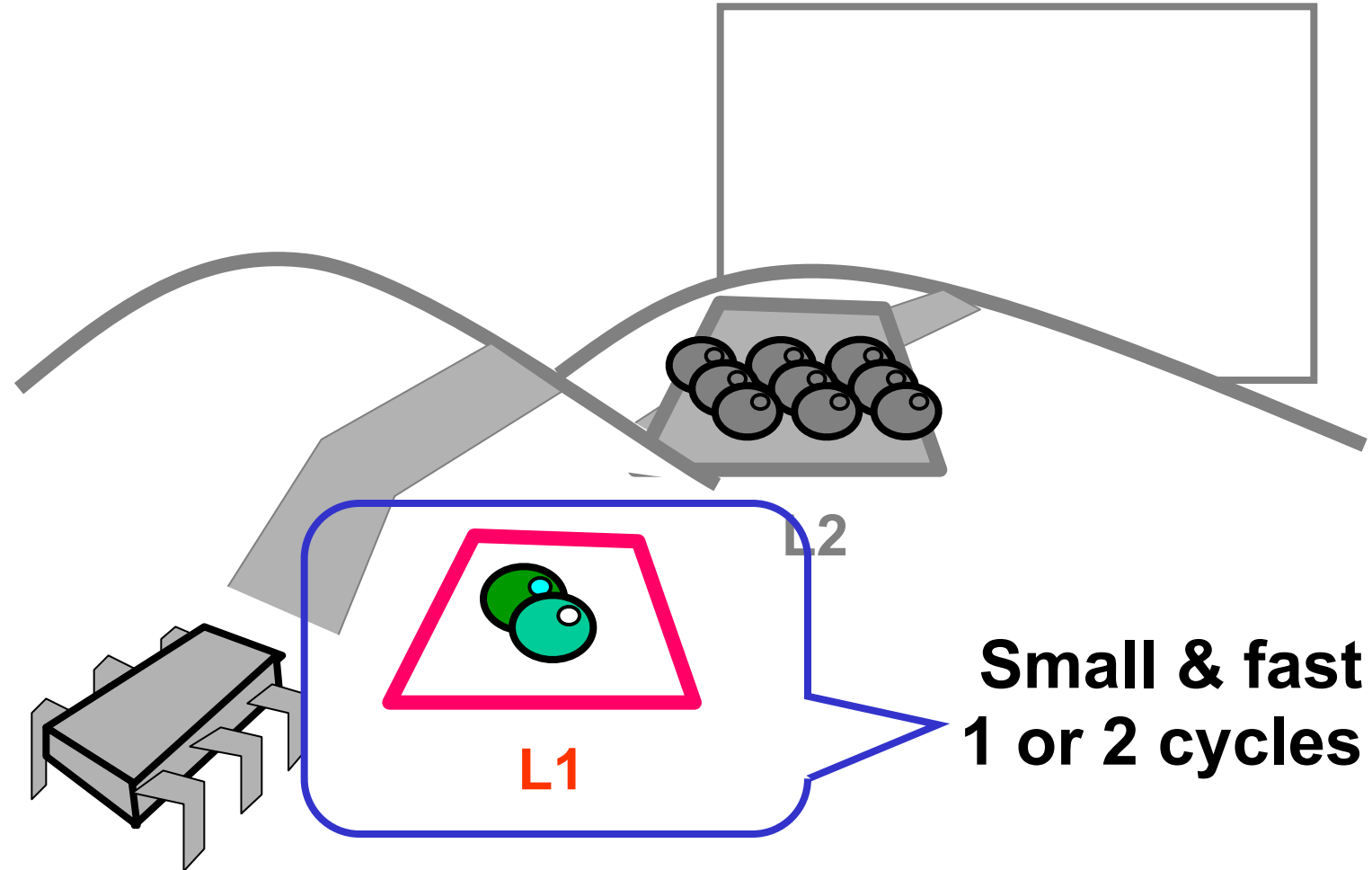
# Locality

- If you use an address now, you will probably use it again soon
  - Fetch from cache, not memory
- If you use an address now, you will probably use a nearby address soon
  - In the same cache line

# L1 and L2 Caches

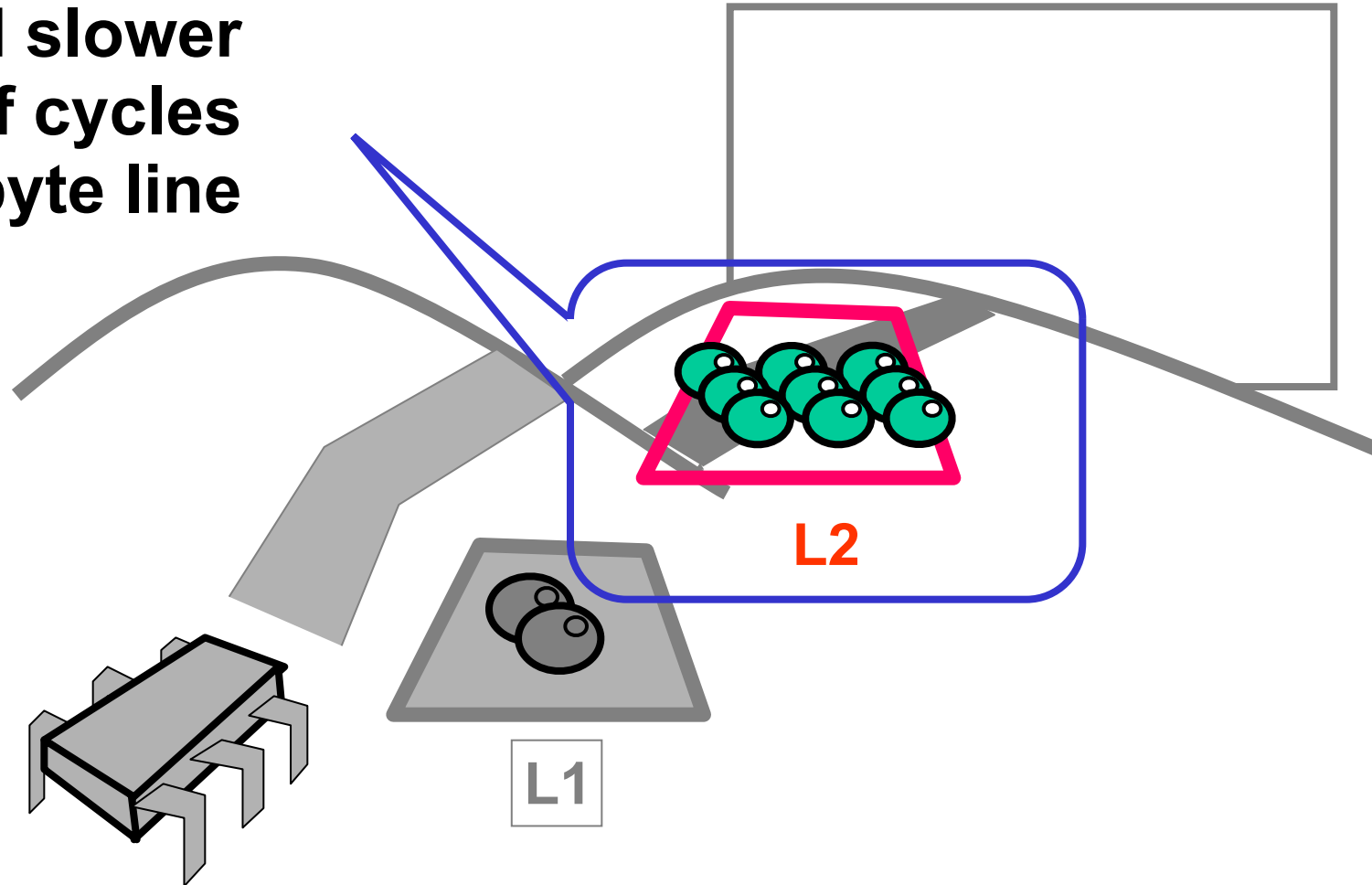


# L1 and L2 Caches



# L1 and L2 Caches

**Larger and slower**  
**10s of cycles**  
**~128 byte line**



# Jargon Watch

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



# Jargon Watch

- Cache hit
  - “I found what I wanted in my cache”
  - Good Thing™
- Cache miss
  - “I had to shlep all the way to memory for that data”
  - Bad Thing™

# Cave Canem

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

# When a Cache Becomes Full...

- Need to make room for new entry
- By evicting an existing entry
- Need a replacement policy
  - Usually some kind of least recently used heuristic

# Cache Coherence

- A and B both cache address  $x$
- A writes to  $x$ 
  - Updates cache
- How does B find out?
- Many cache coherence protocols in literature

# MESI

- Modified
  - Have modified cached data, must write back to memory

# MESI

- Modified
  - Have modified cached data, must write back to memory
- Exclusive
  - Not modified, I have only copy

# MESI

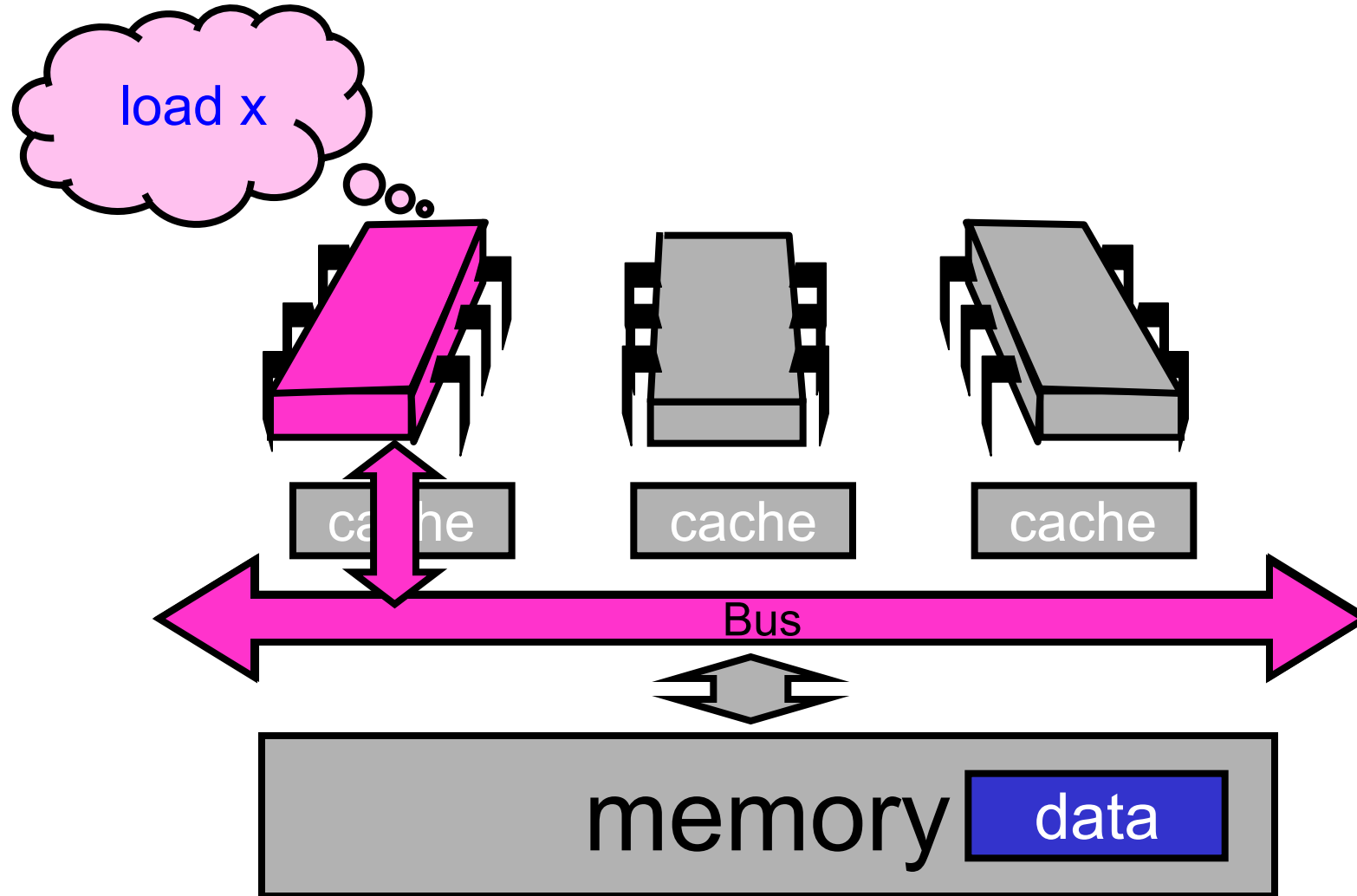
- Modified
  - Have modified cached data, must write back to memory
- Exclusive
  - Not modified, I have only copy
- Shared
  - Not modified, may be cached elsewhere

# MESI

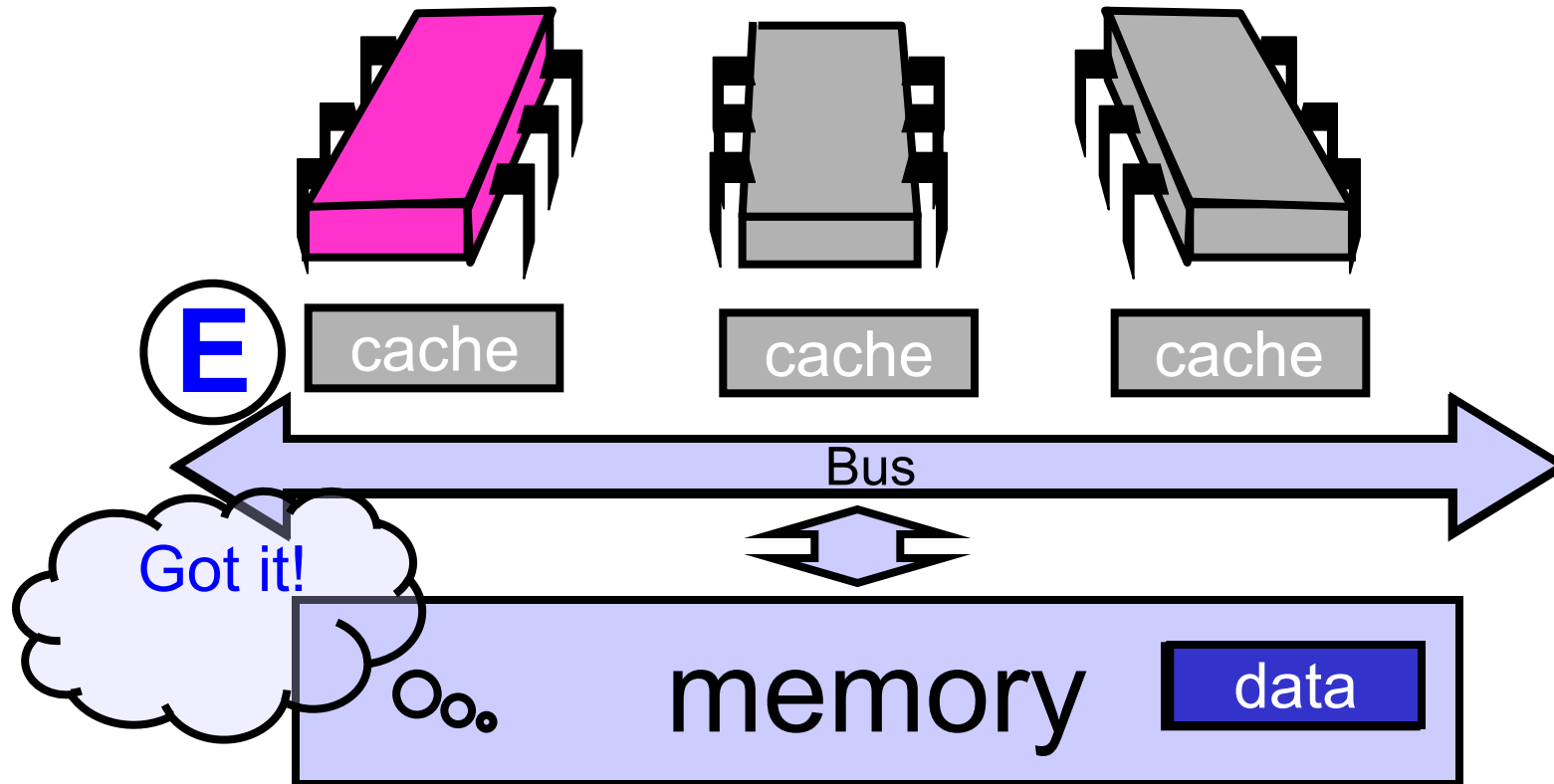
- **Modified**
  - Have modified cached data, must write back to memory
- **Exclusive**
  - Not modified, I have only copy
- **Shared**
  - Not modified, may be cached elsewhere
- **Invalid**
  - Cache contents not meaningful



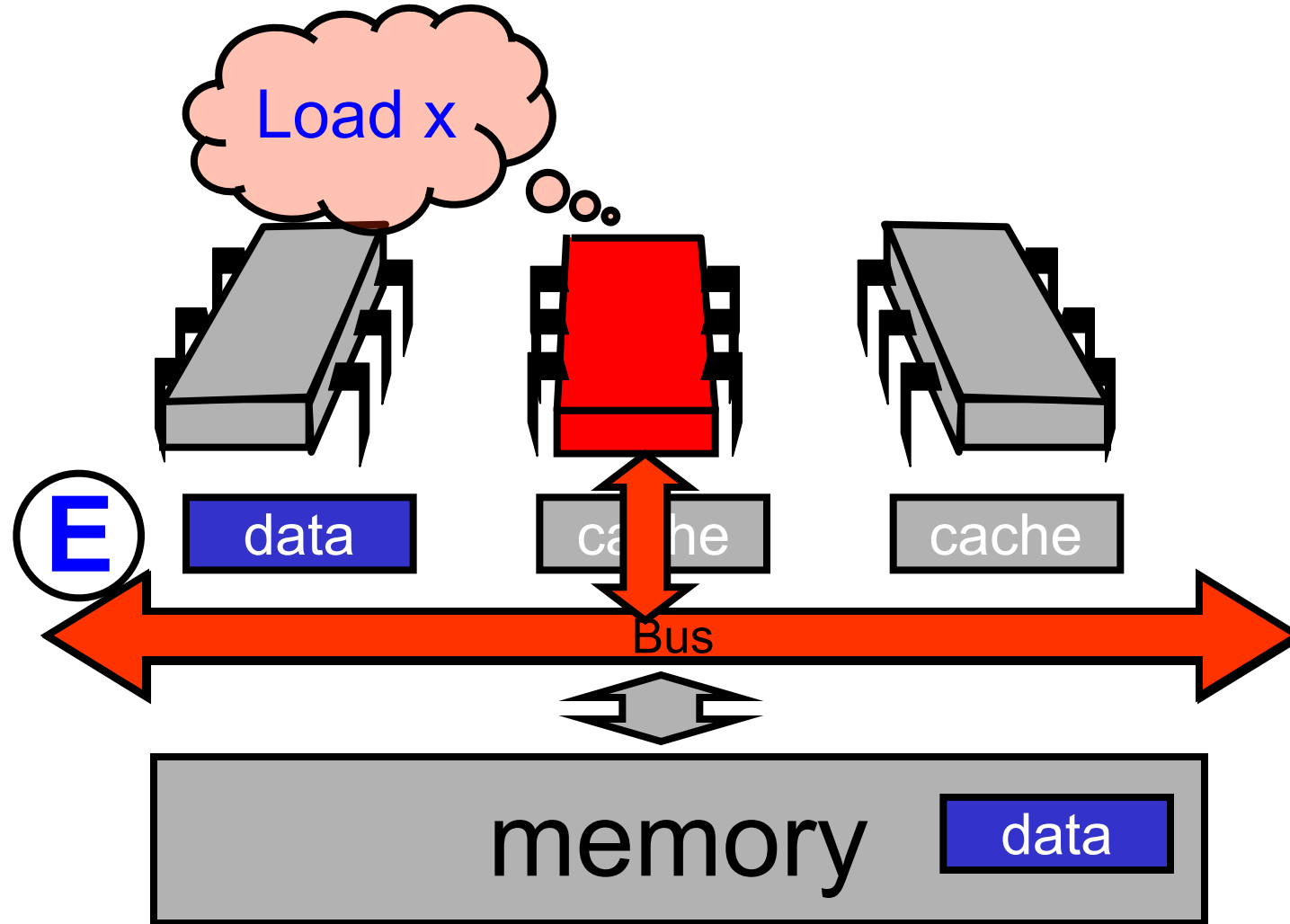
# Processor Issues Load Request



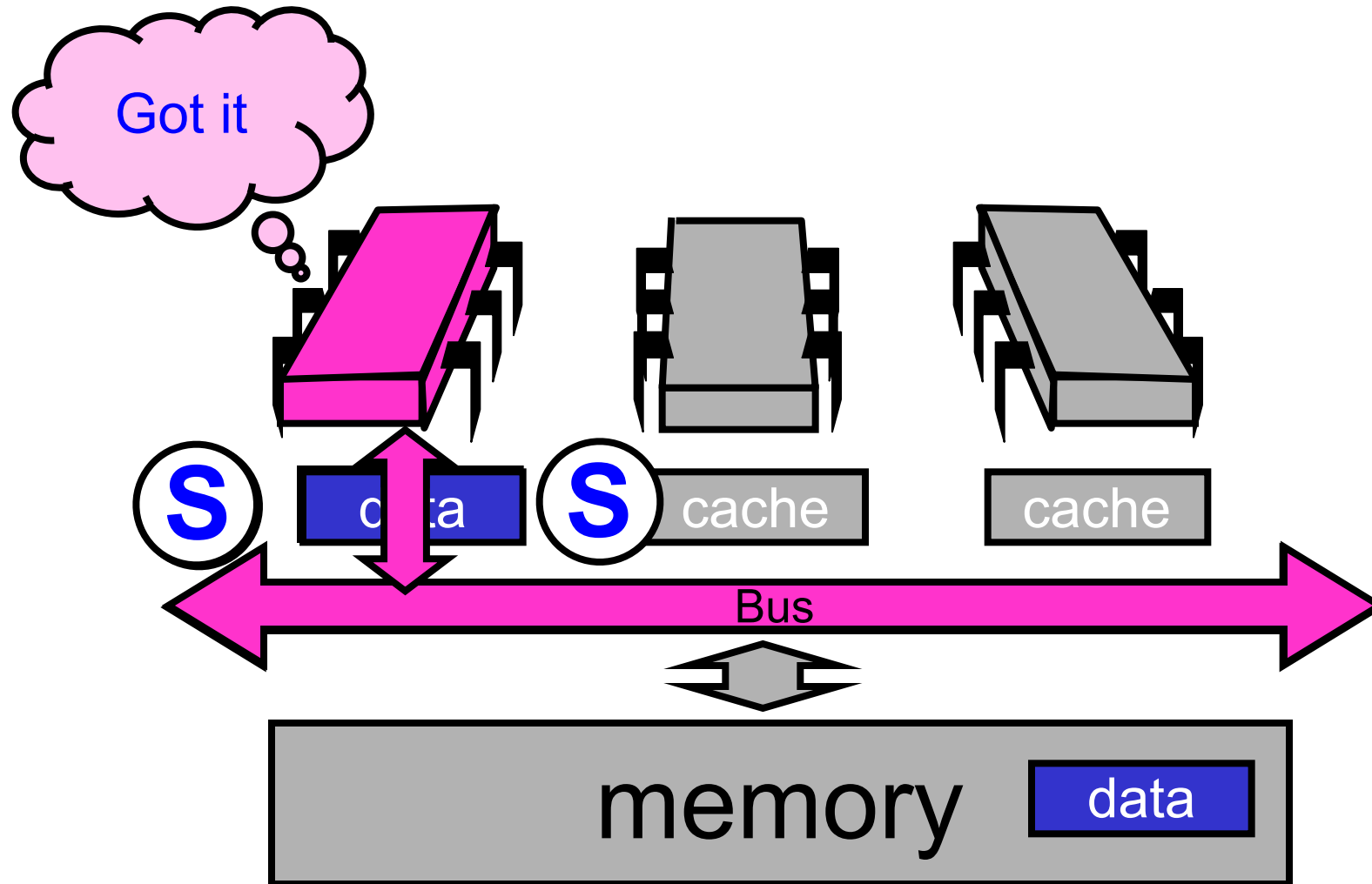
# Memory Responds



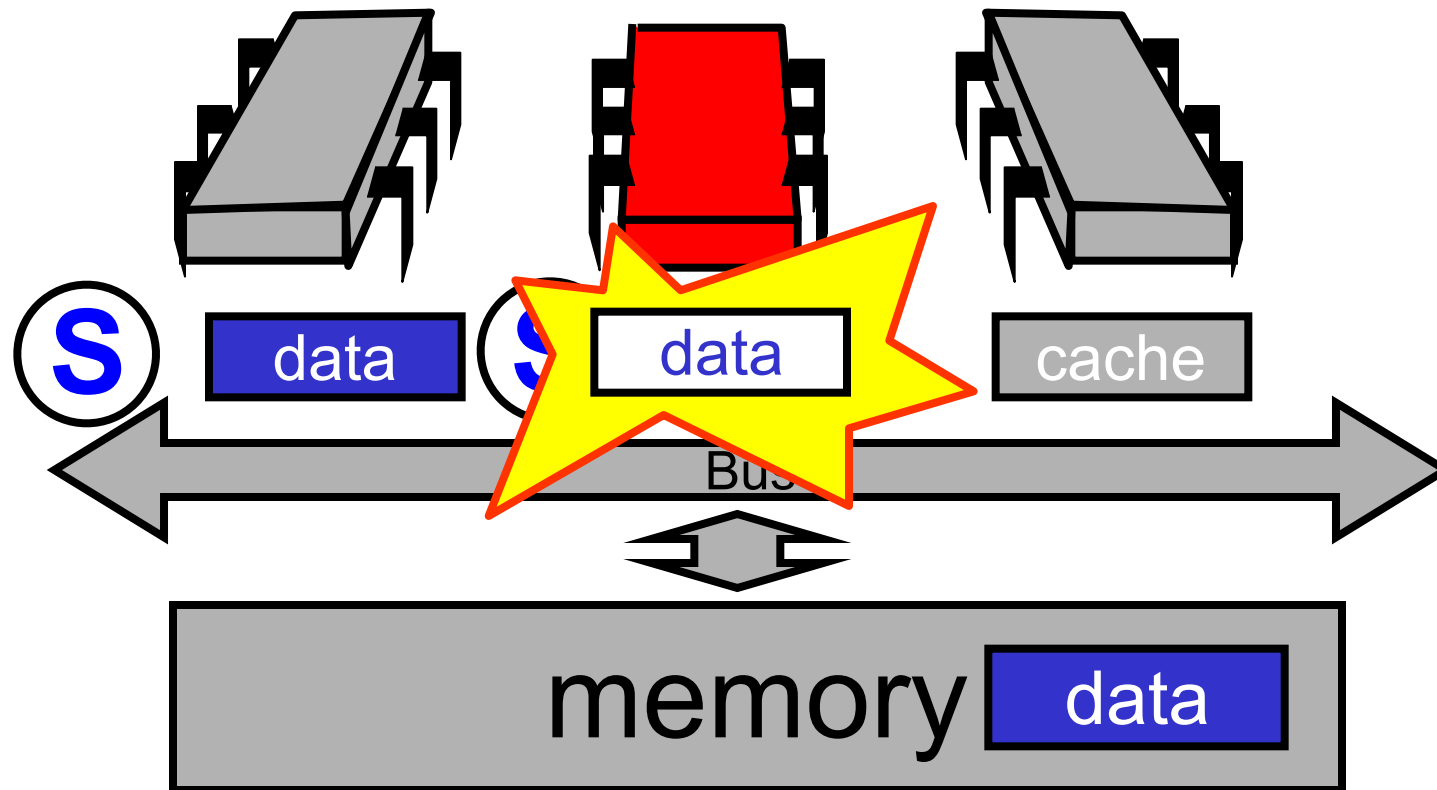
# Processor Issues Load Request



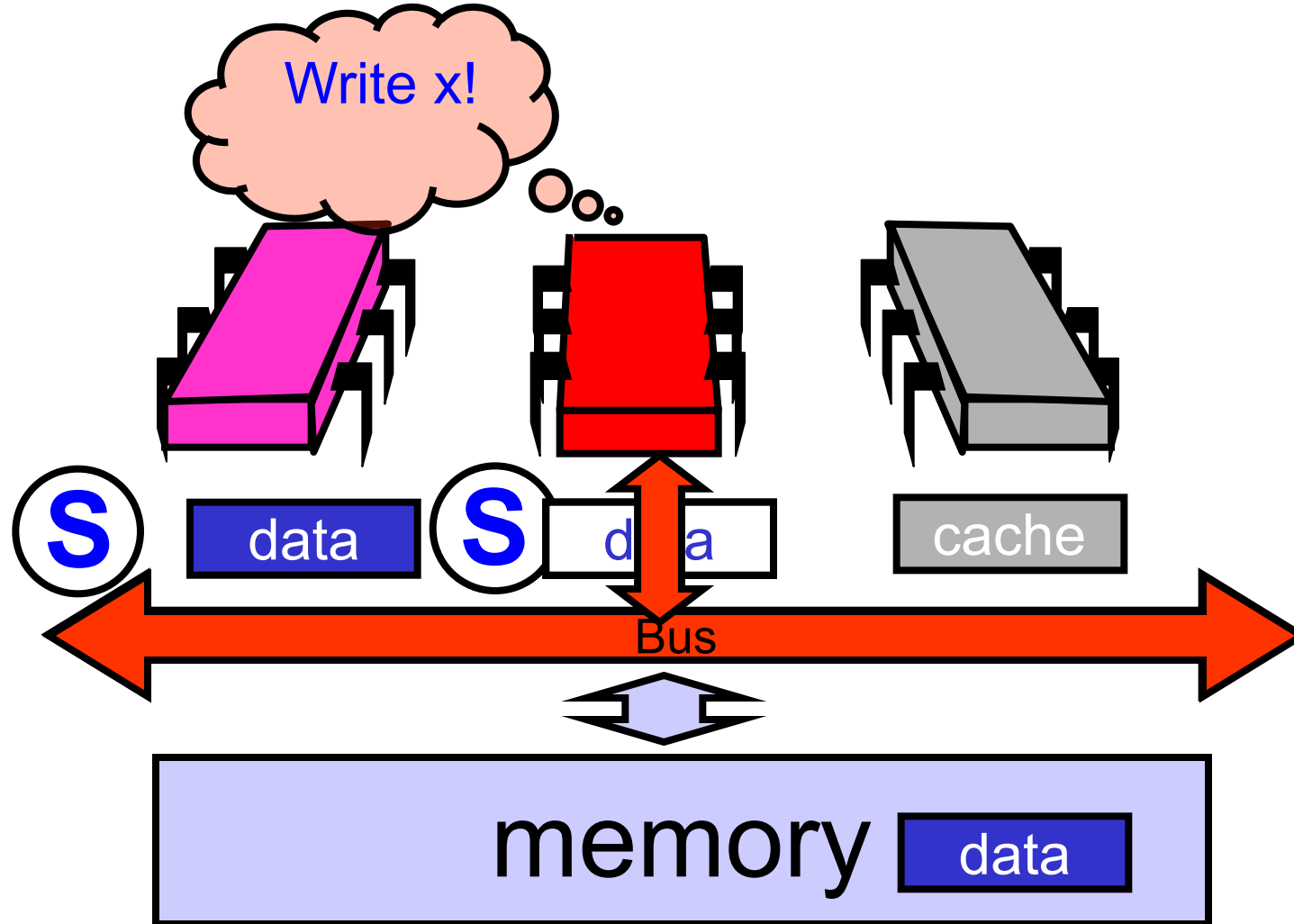
# Other Processor Responds



# Modify Cached Data



# Write-Through Cache



# Write-Through Caches

- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes ...

# Write-Through Caches

- Immediately broadcast changes

- Good

- Memory, caches always agree
  - More read hits, maybe

“show stoppers”

- Bad

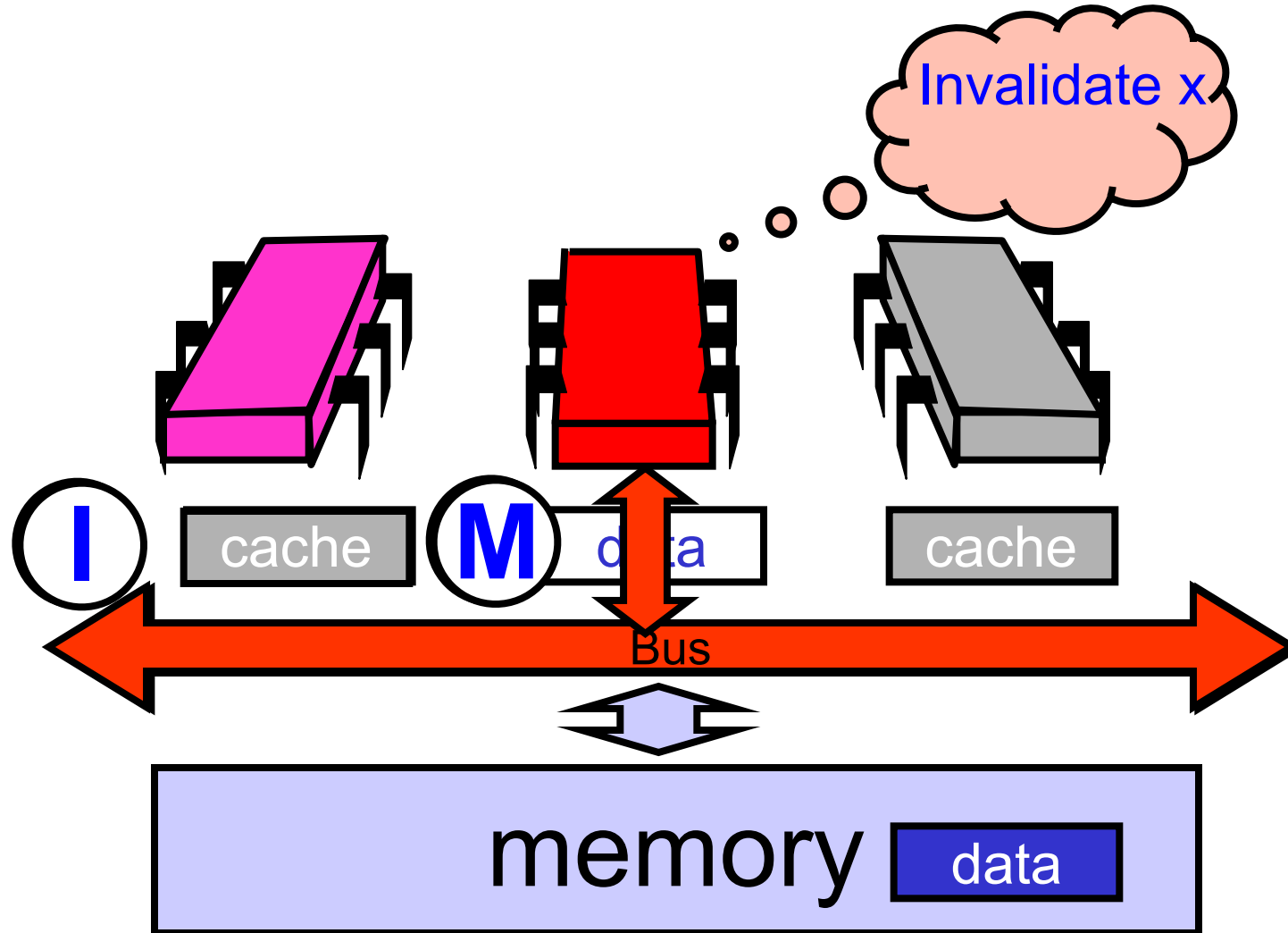
- Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes ...



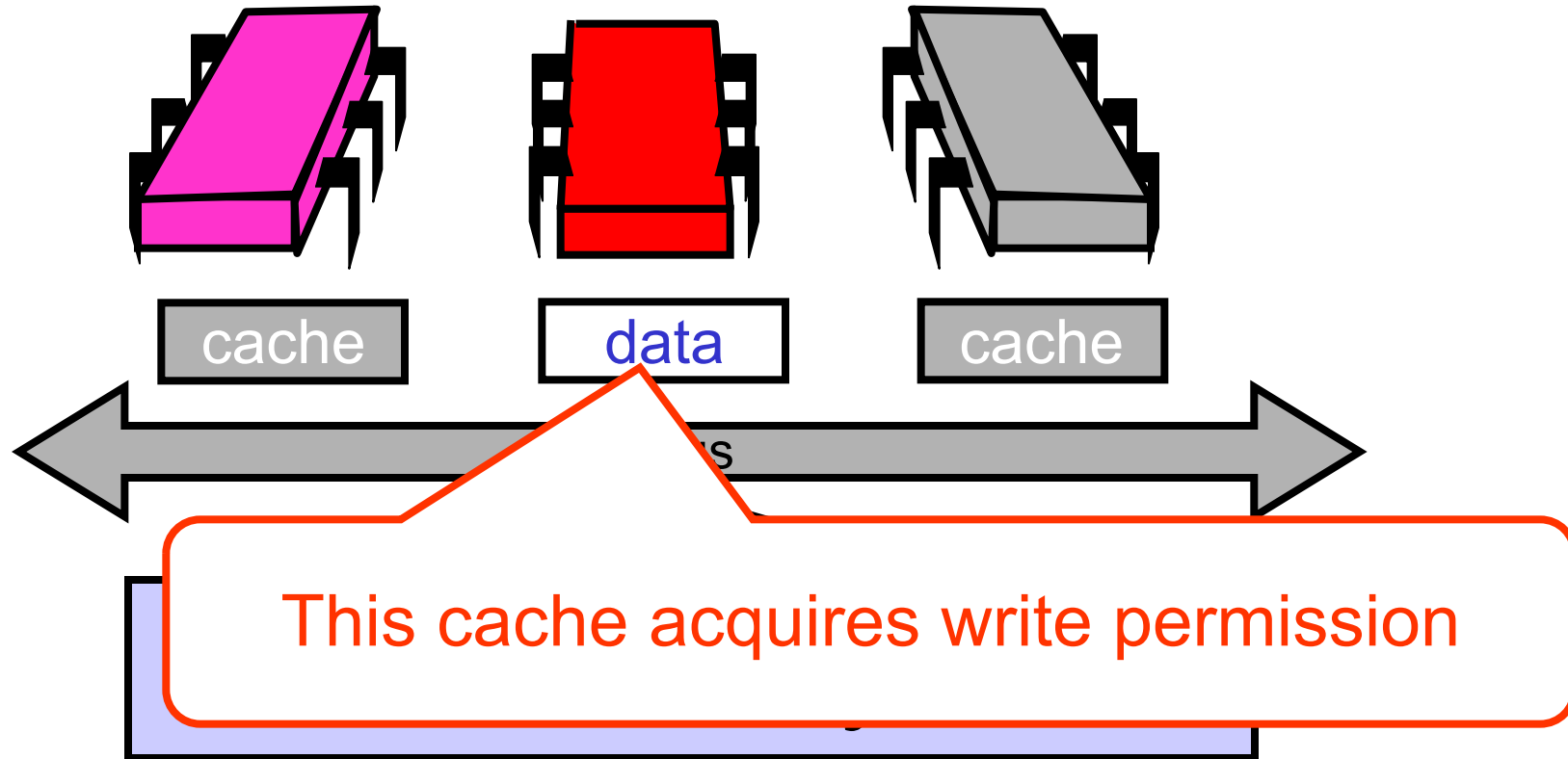
# Write-Back Caches

- Accumulate changes in cache
- Write back when line evicted
  - Need the cache for something else
  - Another processor wants it

# Invalidate

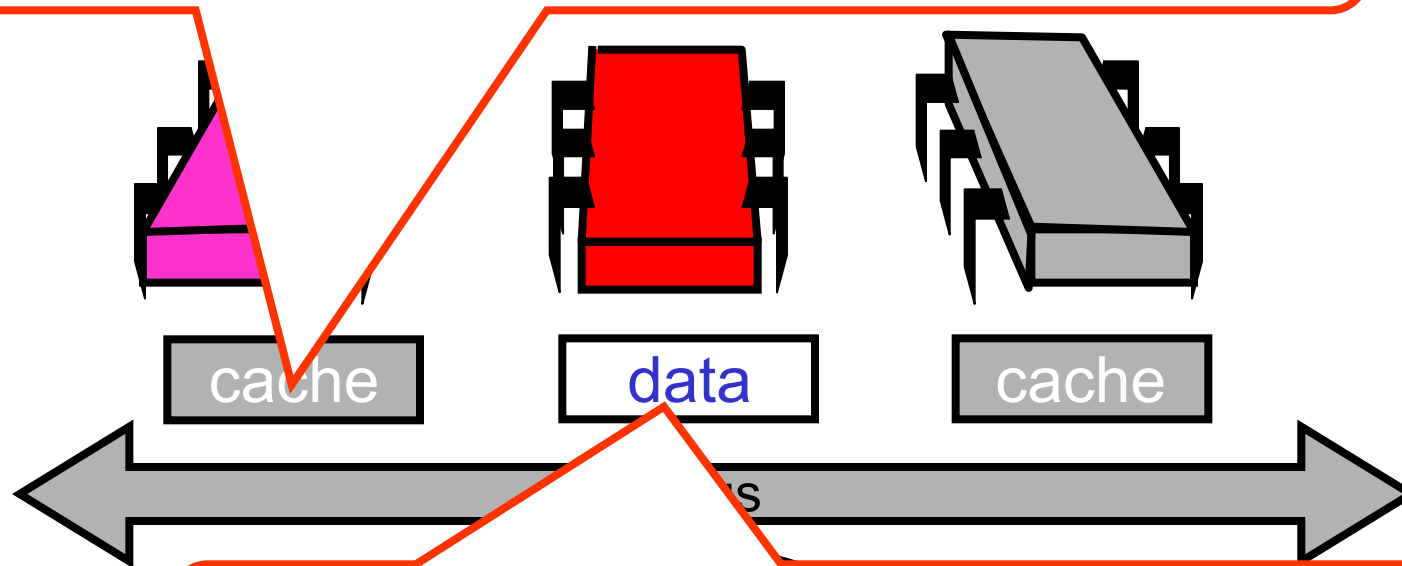


# Invalidate



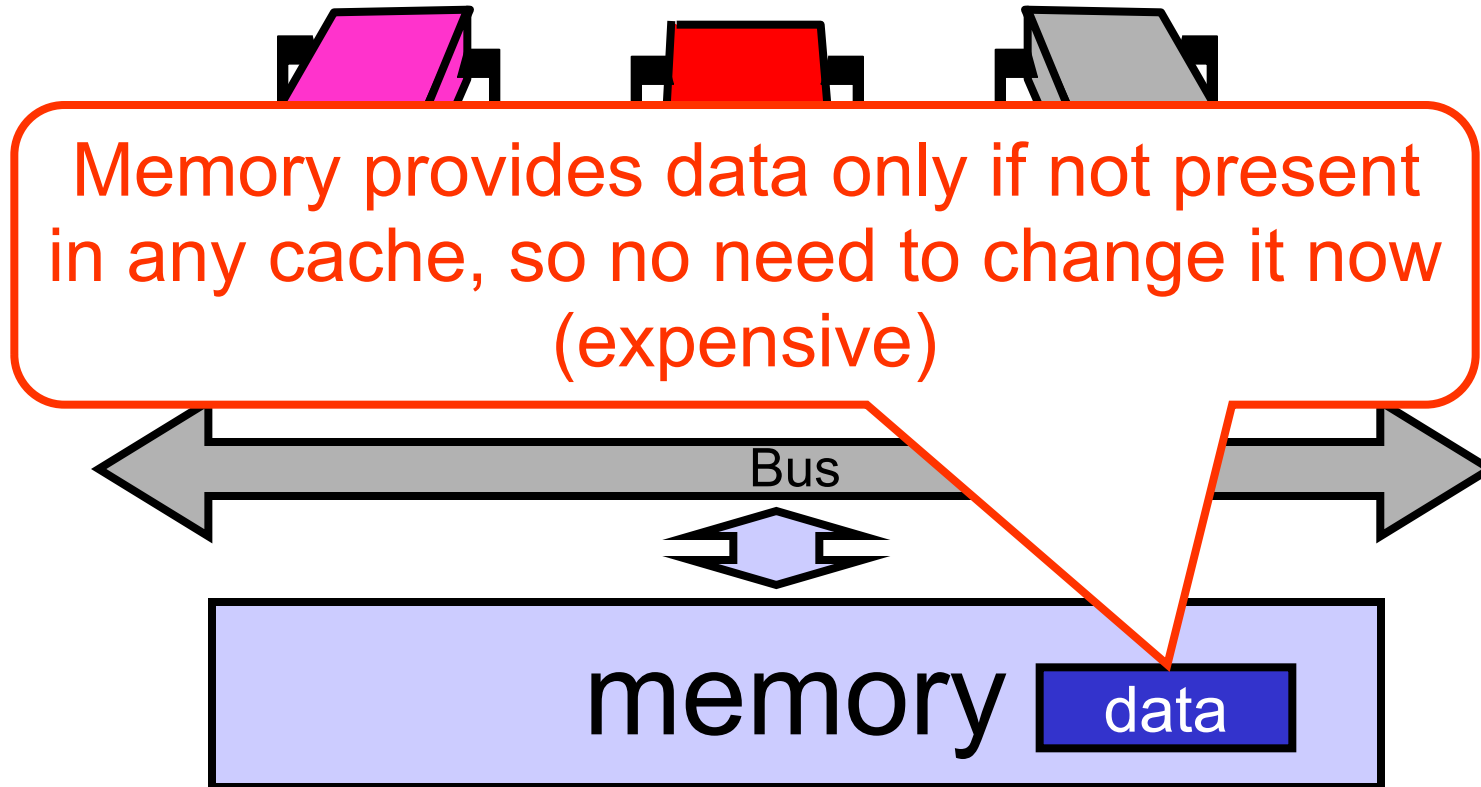
# Invalidate

Other caches lose read permission



This cache acquires write permission

# Invalidate



# Mutual Exclusion

- What do we want to optimize?
  - Bus bandwidth used by spinning threads
  - Release/Acquire latency
  - Acquire latency for idle lock

# Simple TASLock

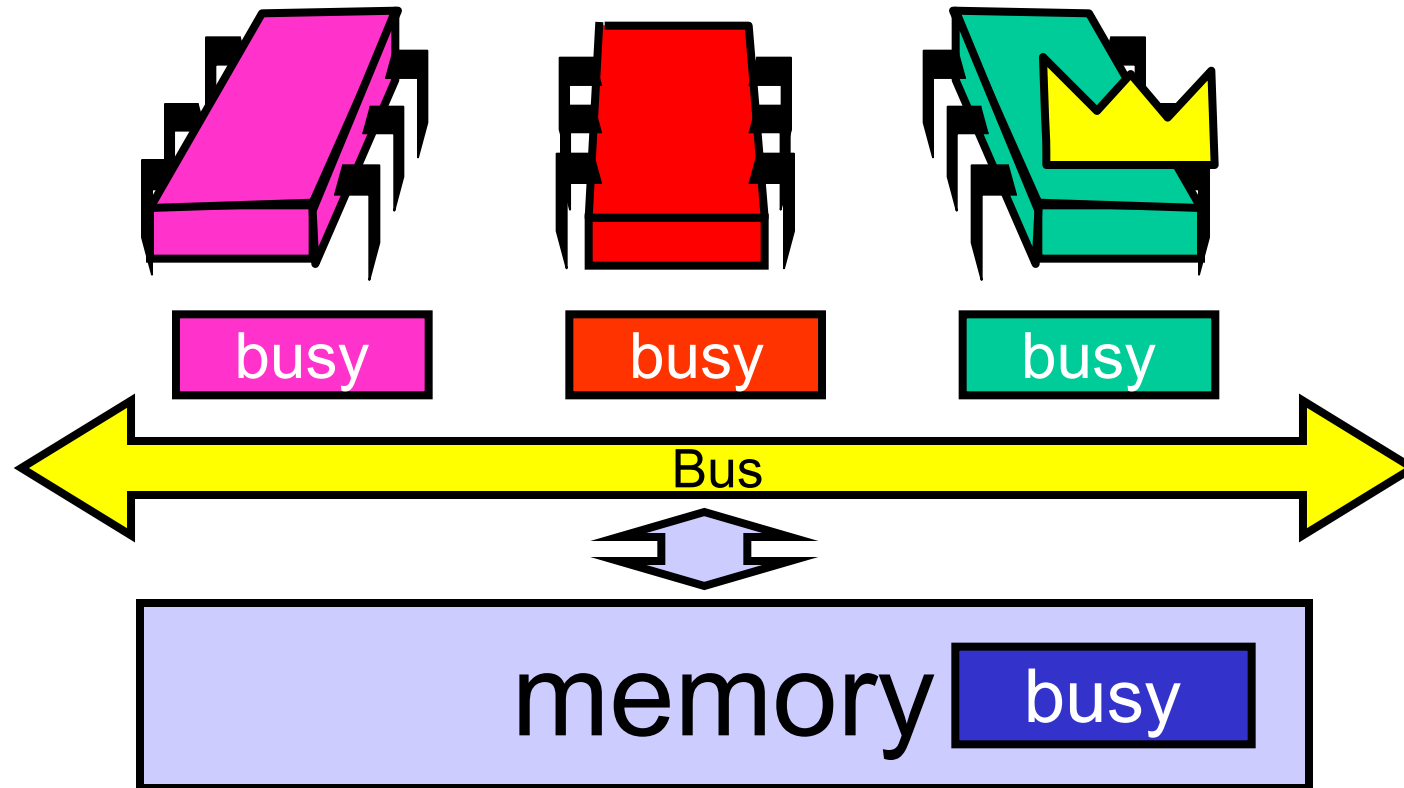
- TAS invalidates cache lines
- Spinners
  - Miss in cache
  - Go to bus
- Thread wants to release lock
  - delayed behind spinners

# Test-and-test-and-set

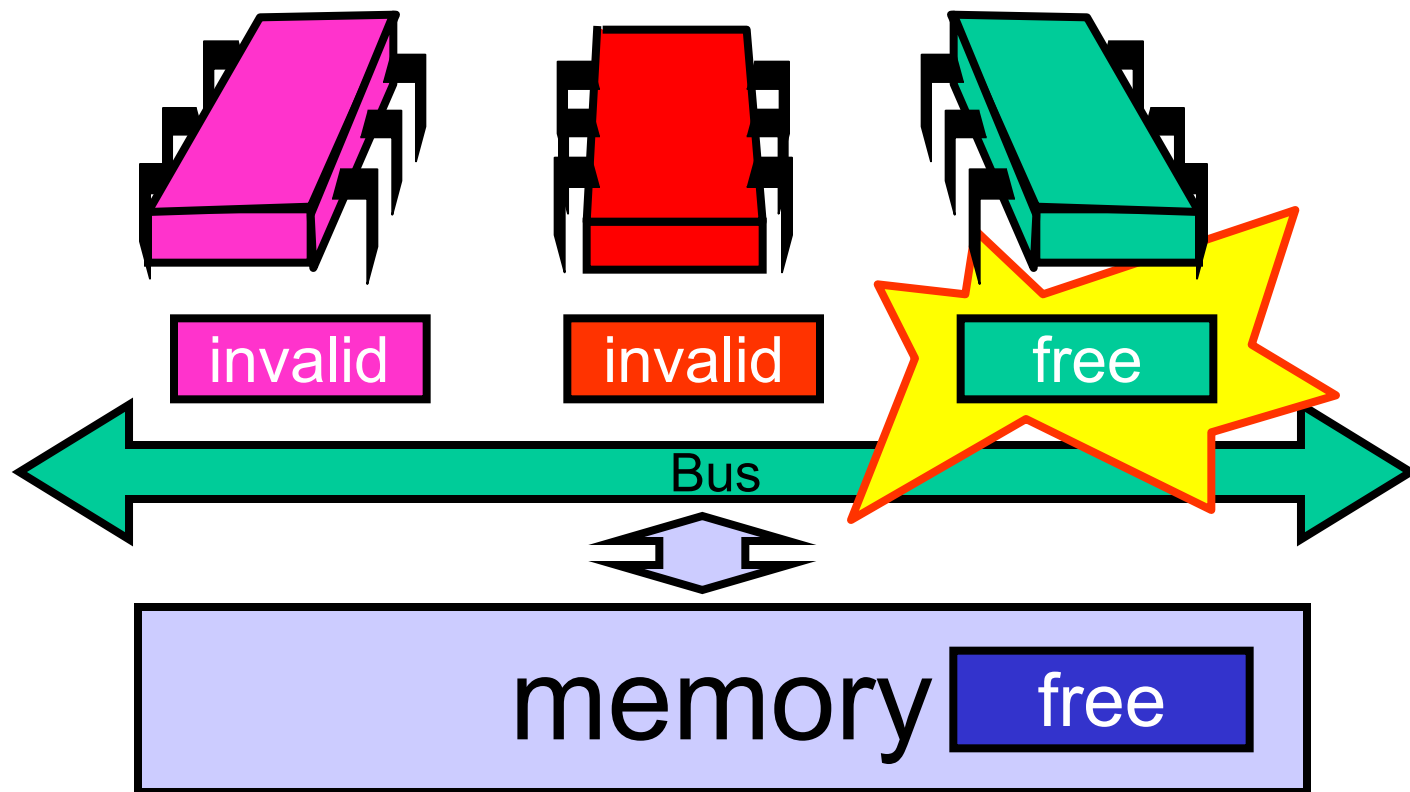
- Wait until lock “looks” free
  - Spin on local cache
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm ...



# Local Spinning while Lock is Busy

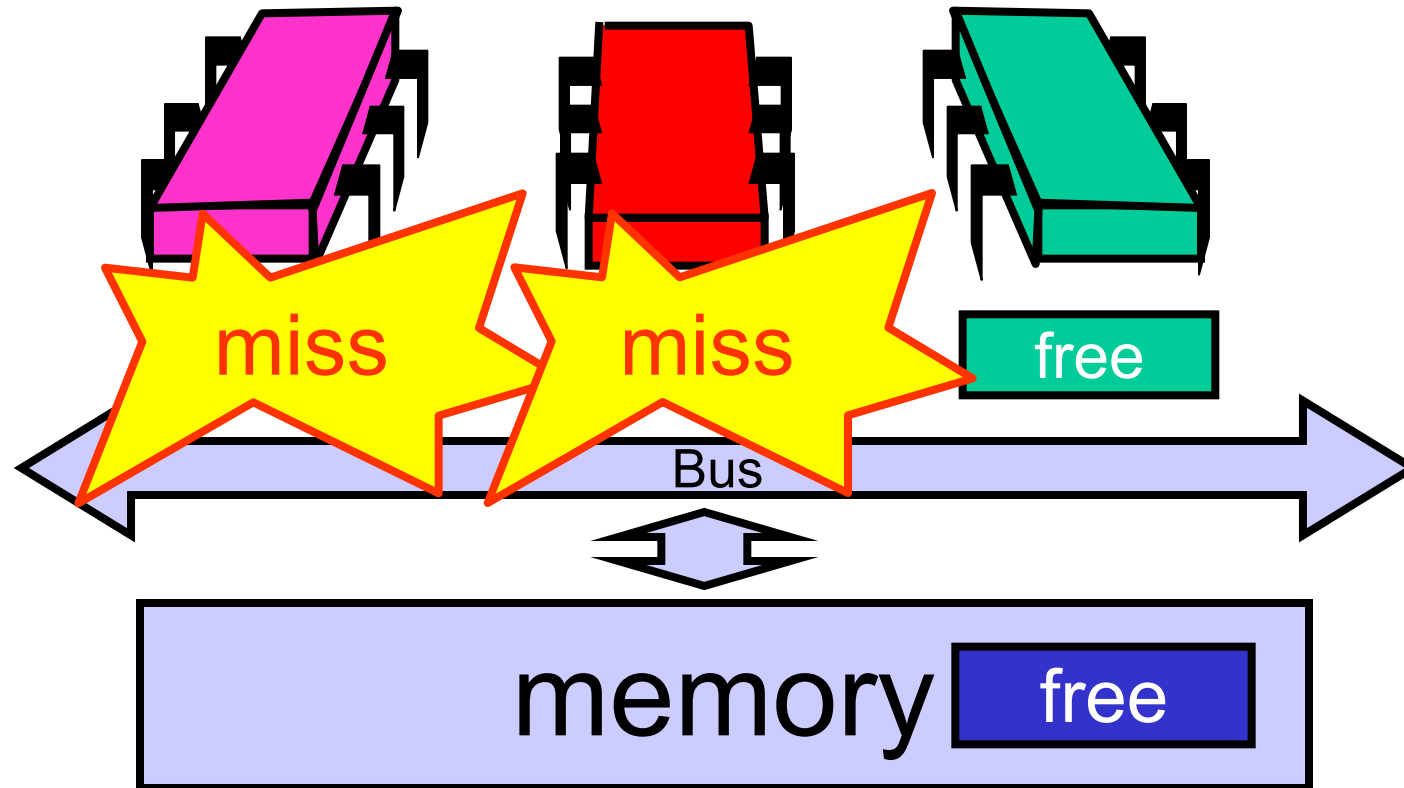


# On Release



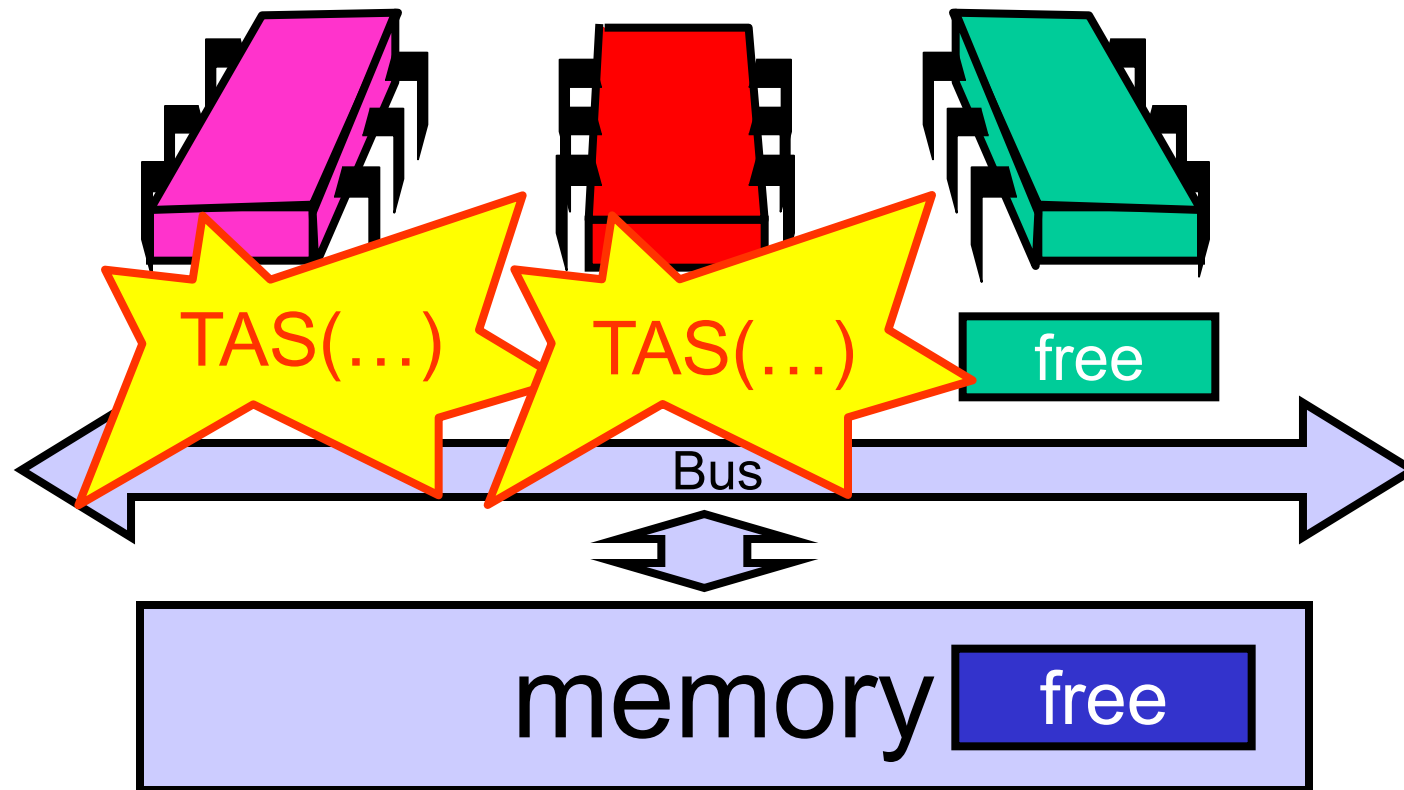
# On Release

Everyone misses,  
rereads



# On Release

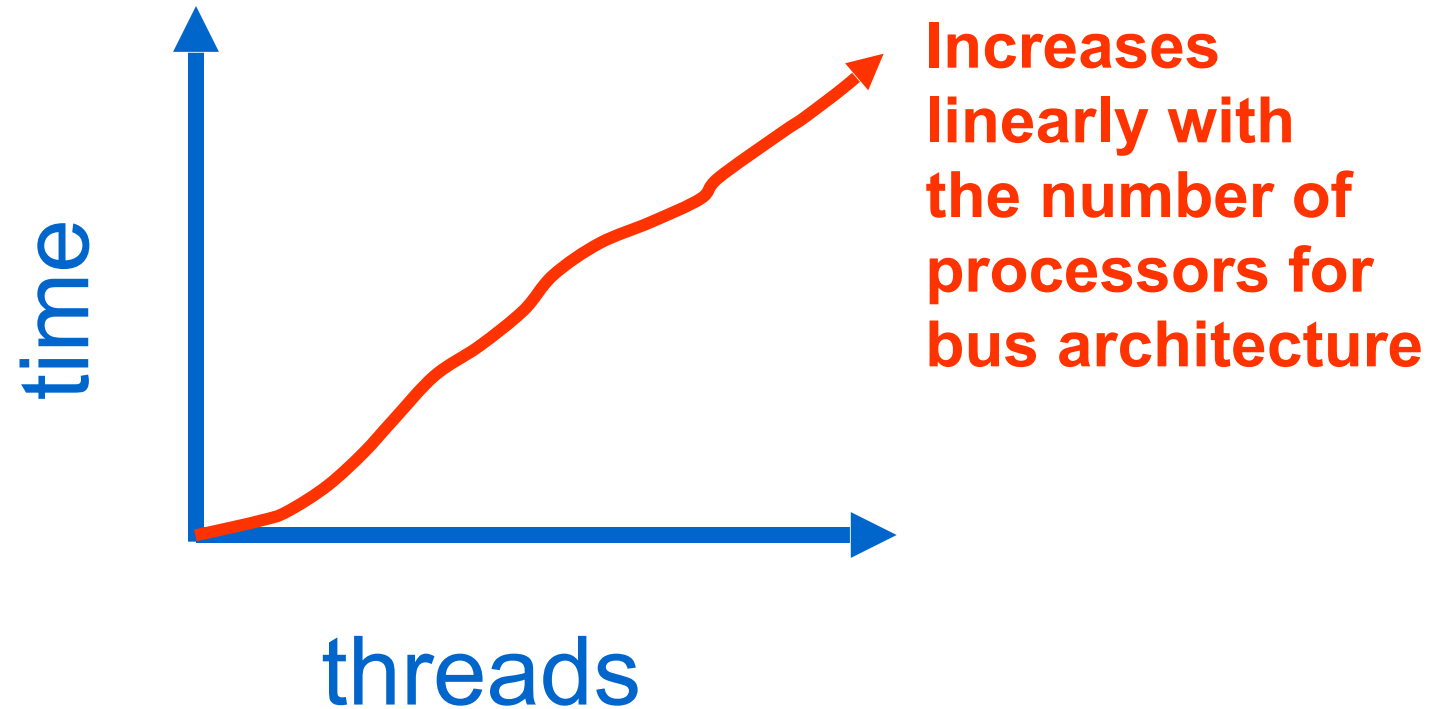
Everyone tries TAS



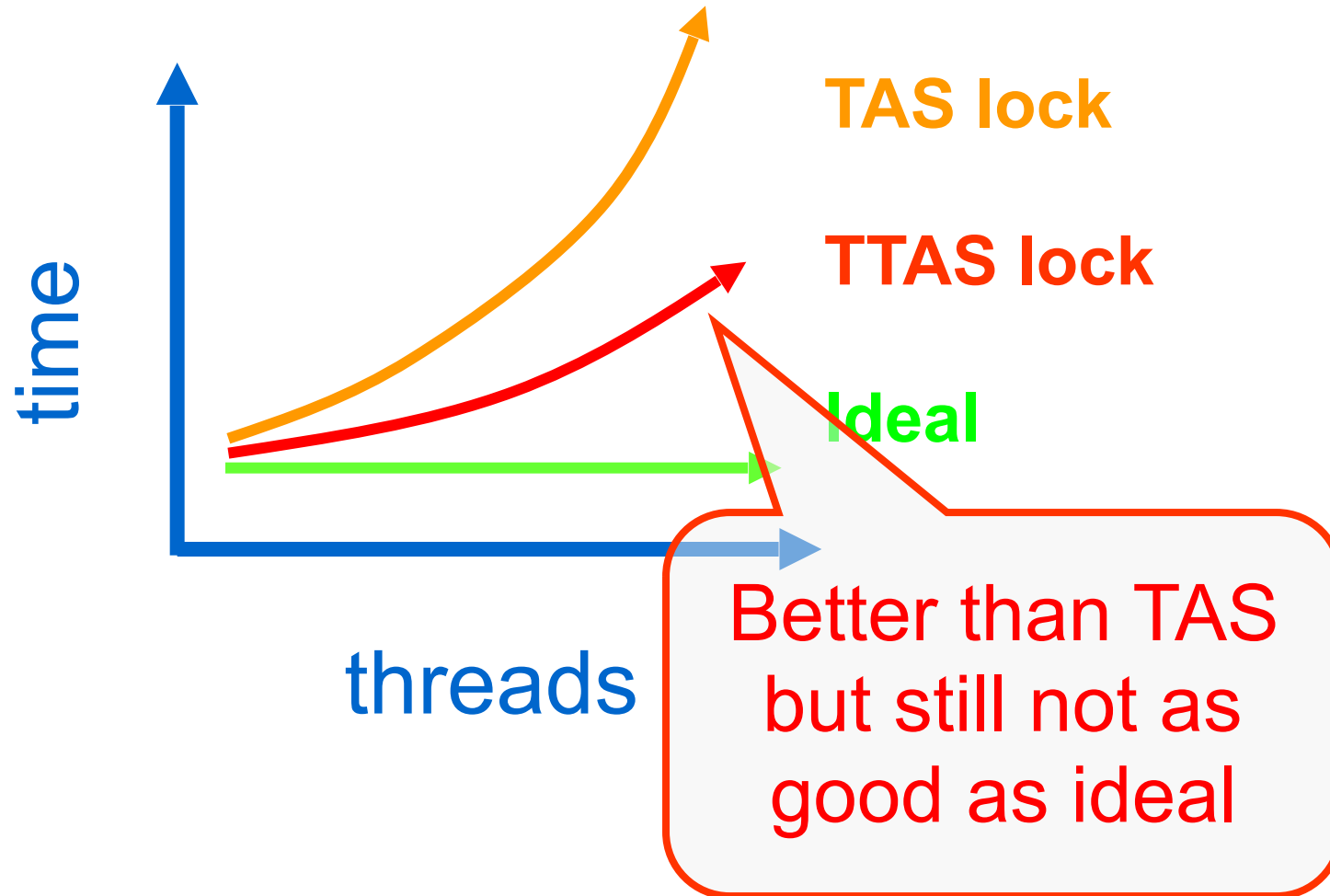
# Problems

- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?

# Quiescence Time

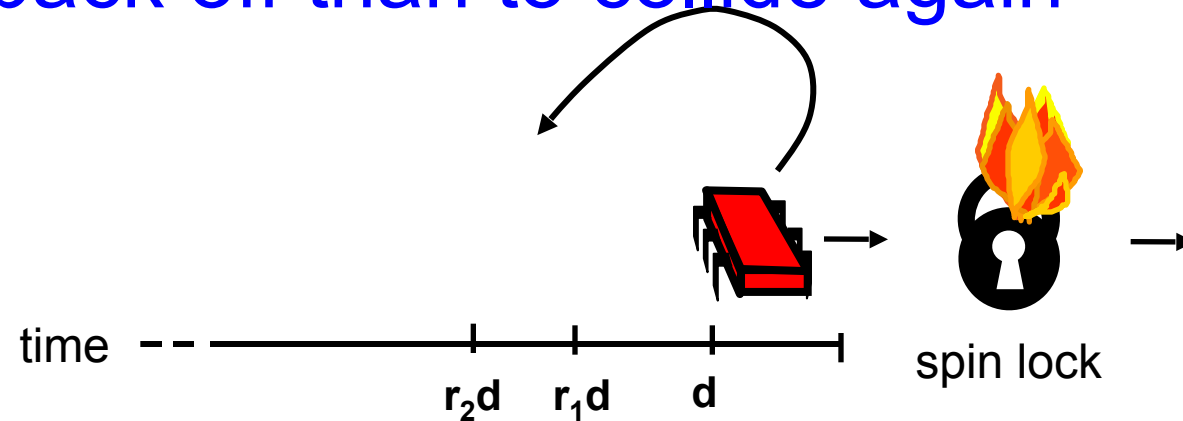


# Mystery Explained



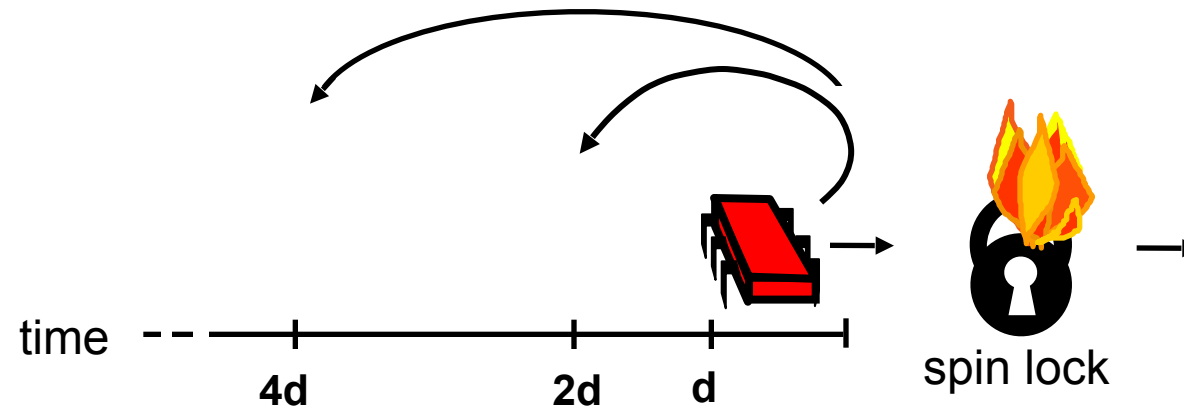
# Solution: Introduce Delay

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





# Dynamic Example: Exponential Backoff



If I fail to get lock

- Wait random duration before retry
- Each subsequent failure doubles expected wait

# Exponential Backoff Lock

```
class BackoffLock extends SpinLock {  
  private var delay = MIN_DELAY  
  override def lock(): Unit = {  
    while (true) {  
      while (state.get()) {}  
      if (!state.getAndSet(true)) { return } else {  
        Thread.sleep(random() % delay);  
        if (delay < MAX_DELAY) delay = 2 * delay  
      }  
    }  
  }  
}
```

# Exponential Backoff Lock

```
class BackoffLock extends SpinLock {  
  private var delay = MIN_DELAY  
  override def lock(): Unit = {  
    while (true) {  
      while (state.get()) {}  
      if (!state.getAndSet(true)) { return } else {  
        Thread.sleep(random() % delay);  
        if (delay < MAX_DELAY) delay = 2 * delay  
      }  
    }  
  }  
}
```

**Fix minimum delay**

# Exponential Backoff Lock

```
class BackoffLock extends SpinLock {  
  private var delay = MIN_DELAY  
  override def lock(): Unit = {  
    while (true) {  
      while (state.get()) {}  
      if (!state.getAndSet(true)) { return } else {  
        Thread.sleep(random() % delay);  
        if (delay < MAX_DELAY) delay = 2 * delay  
      }  
    }  
  }  
}
```

**Wait until lock looks free**

# Exponential Backoff Lock

```
class BackoffLock extends SpinLock {  
  private var delay = MIN_DELAY  
  override def lock(): Unit = {  
    while (true) {  
      while (state.get()) {}  
      if (!state.getAndSet(true)) { return } else {  
        Thread.sleep(random() % delay);  
        if (delay < MAX_DELAY) delay = 2 * delay  
      }  
    }  
  }  
}
```

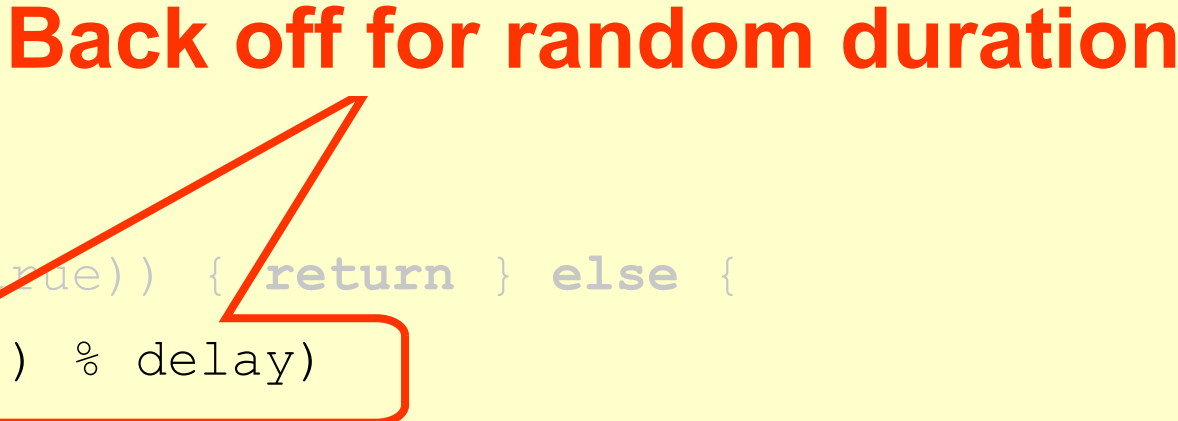
If we win, return



# Exponential Backoff Lock

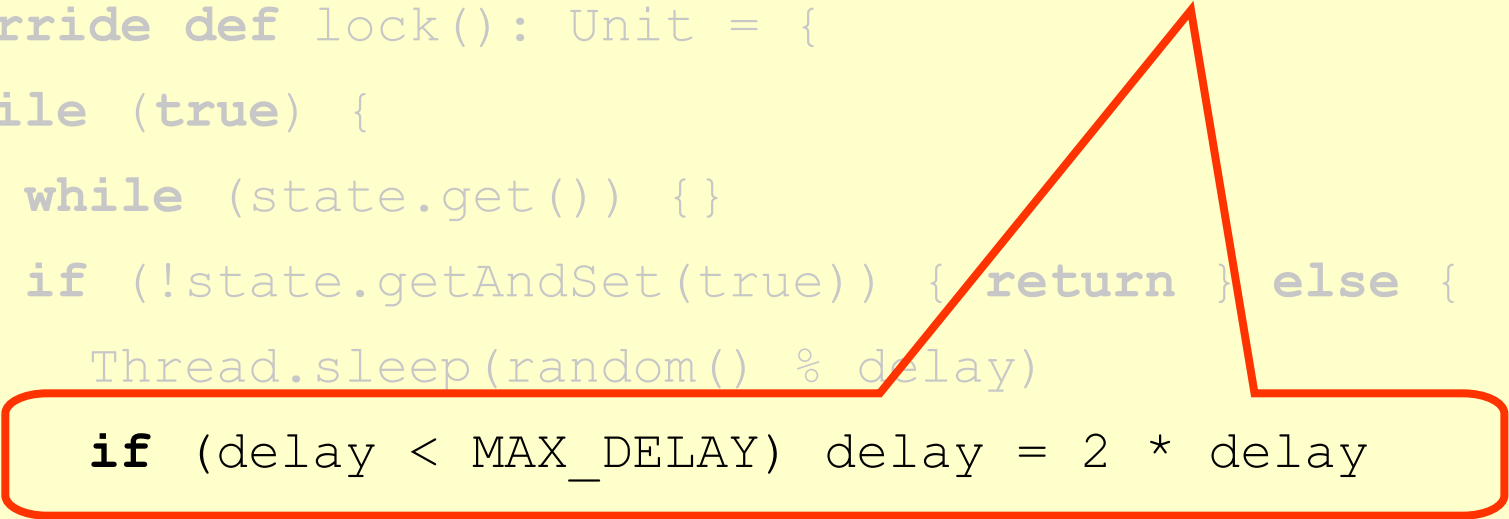
```
class BackoffLock extends SpinLock {  
  private var delay = MIN_DELAY  
  override def lock(): Unit  
    while (true) {  
      while (state.get()) {}  
      if (!state.getAndSet(true)) { return } else {  
        Thread.sleep(random() % delay)  
        if (delay < MAX_DELAY) delay = 2 * delay  
      }  
    }  
}
```

**Back off for random duration**

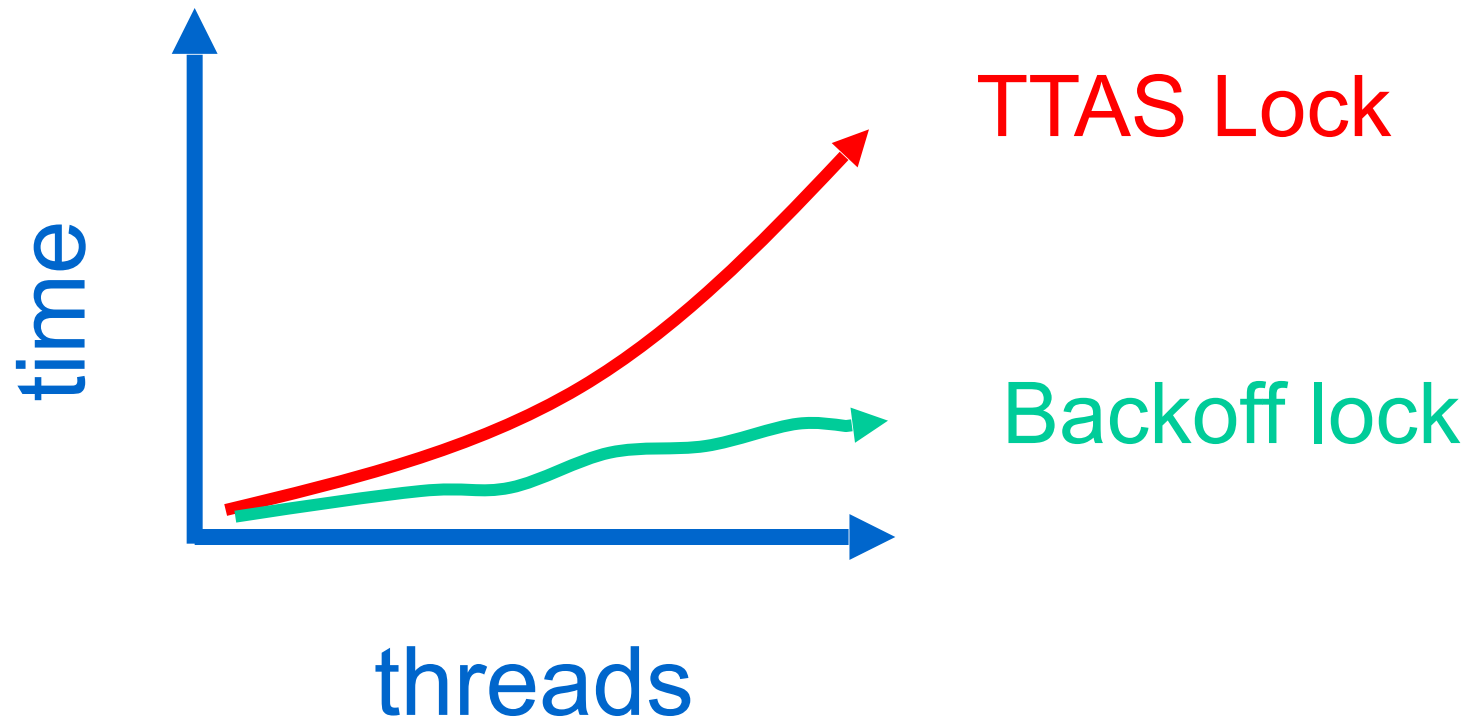


# Exponential Backoff Lock

```
class BackoffLock extends SpinLock {  
  private var delay = Double max delay, within reason  
  override def lock(): Unit = {  
    while (true) {  
      while (state.get()) {}  
      if (!state.getAndSet(true)) { return } else {  
        Thread.sleep(random() % delay)  
        if (delay < MAX_DELAY) delay = 2 * delay  
      }  
    }  
  }  
}
```



# Spin-Waiting Overhead

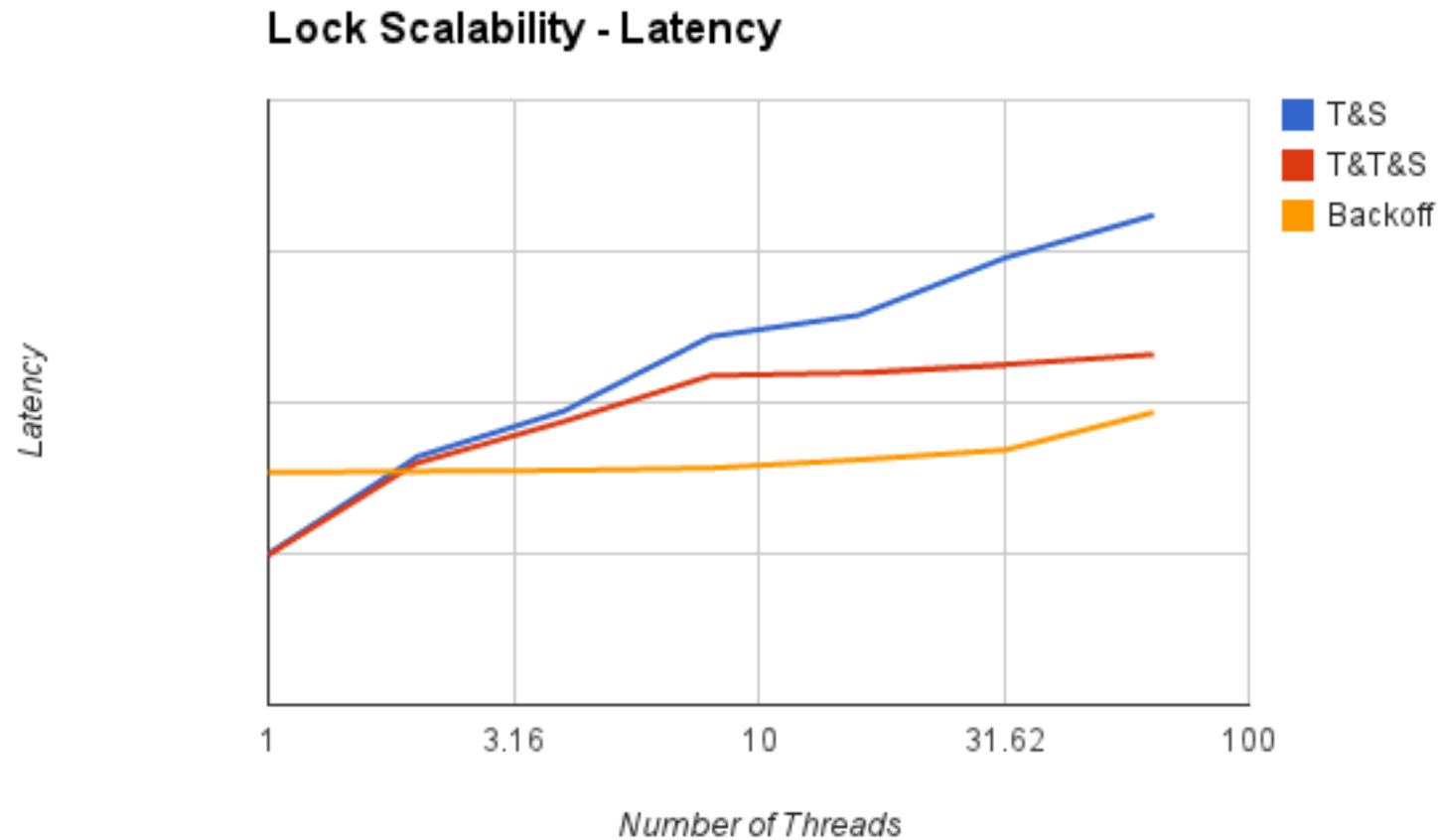




# Backoff: Other Issues

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

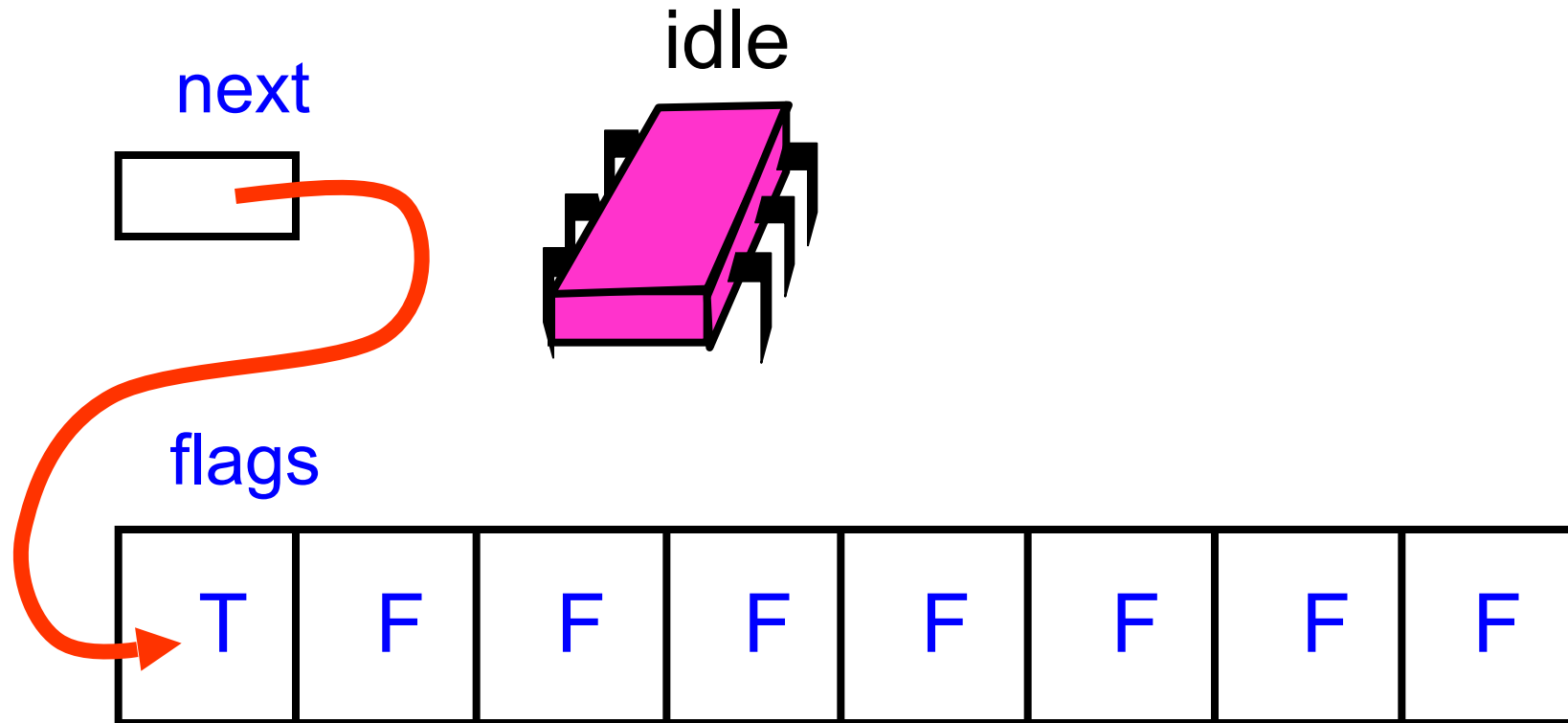
# Actual Data on 40-Core Machine



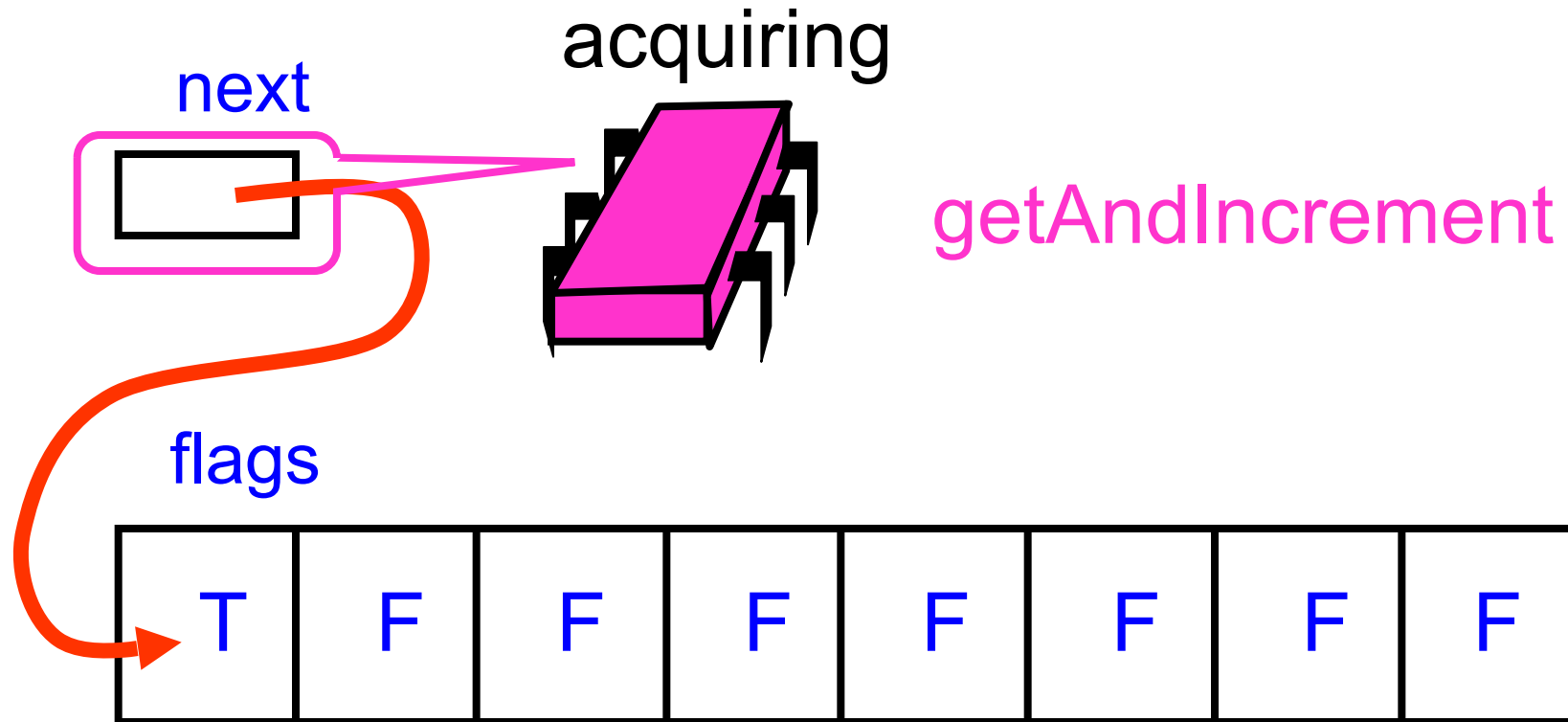
# A Prominent Idea

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

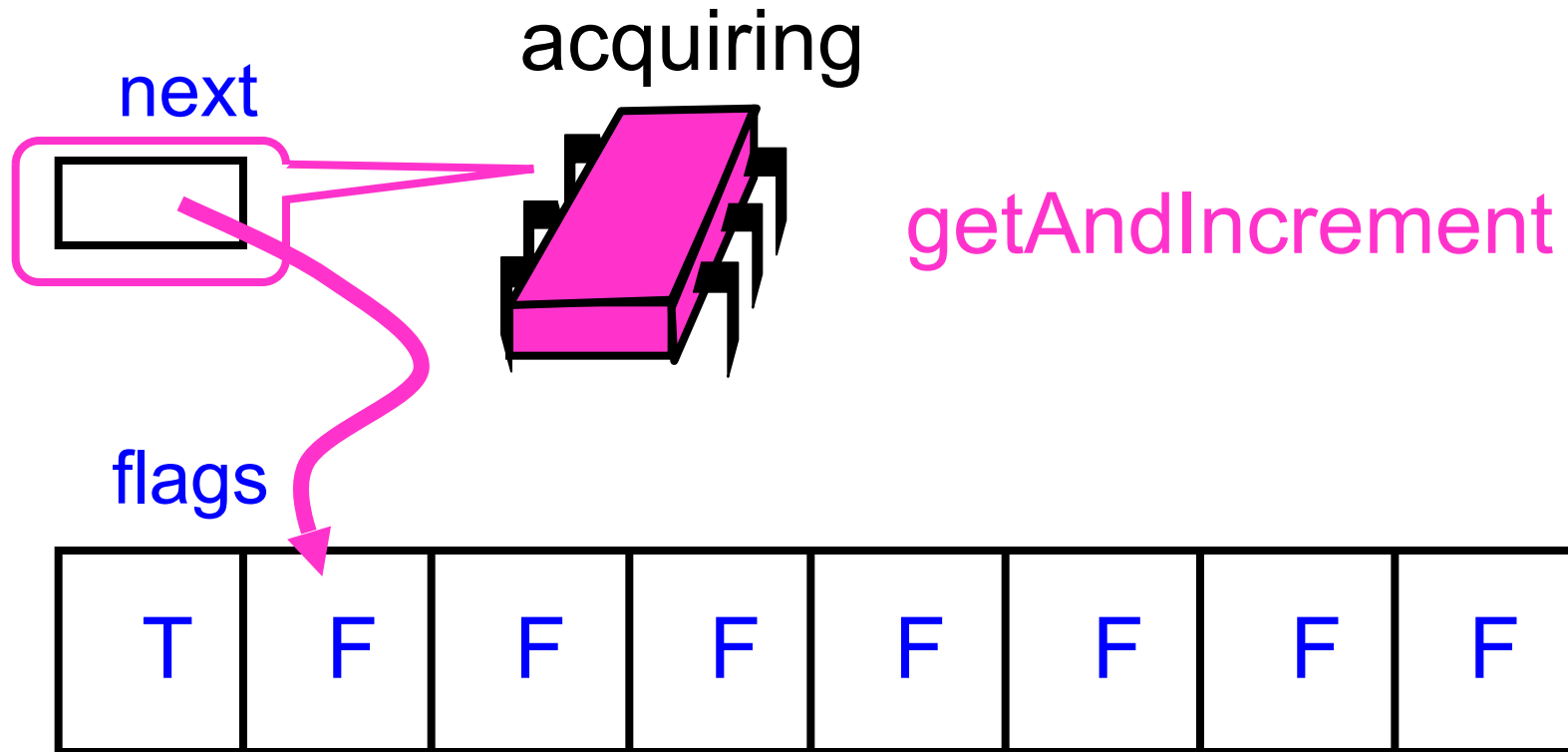
# Anderson Queue Lock



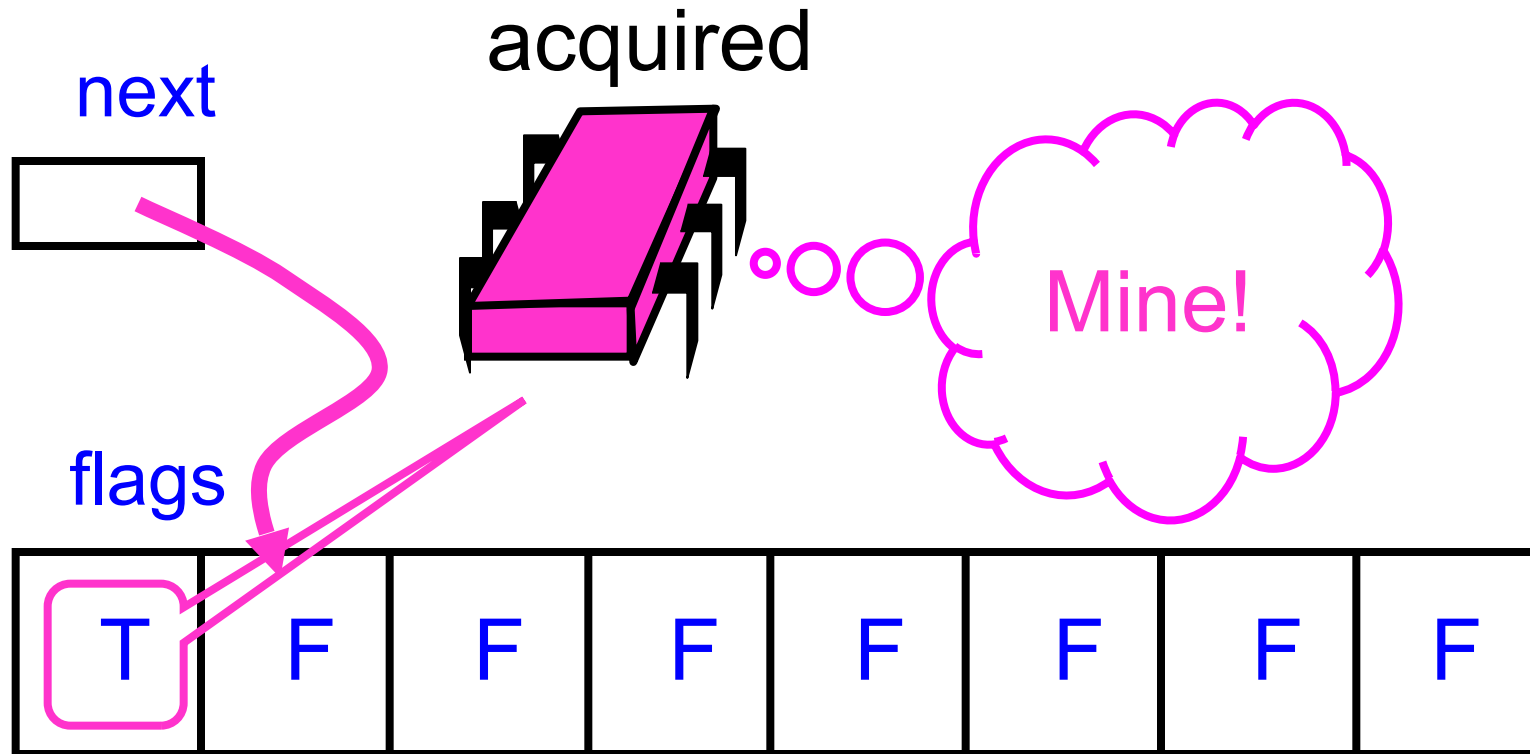
# Anderson Queue Lock



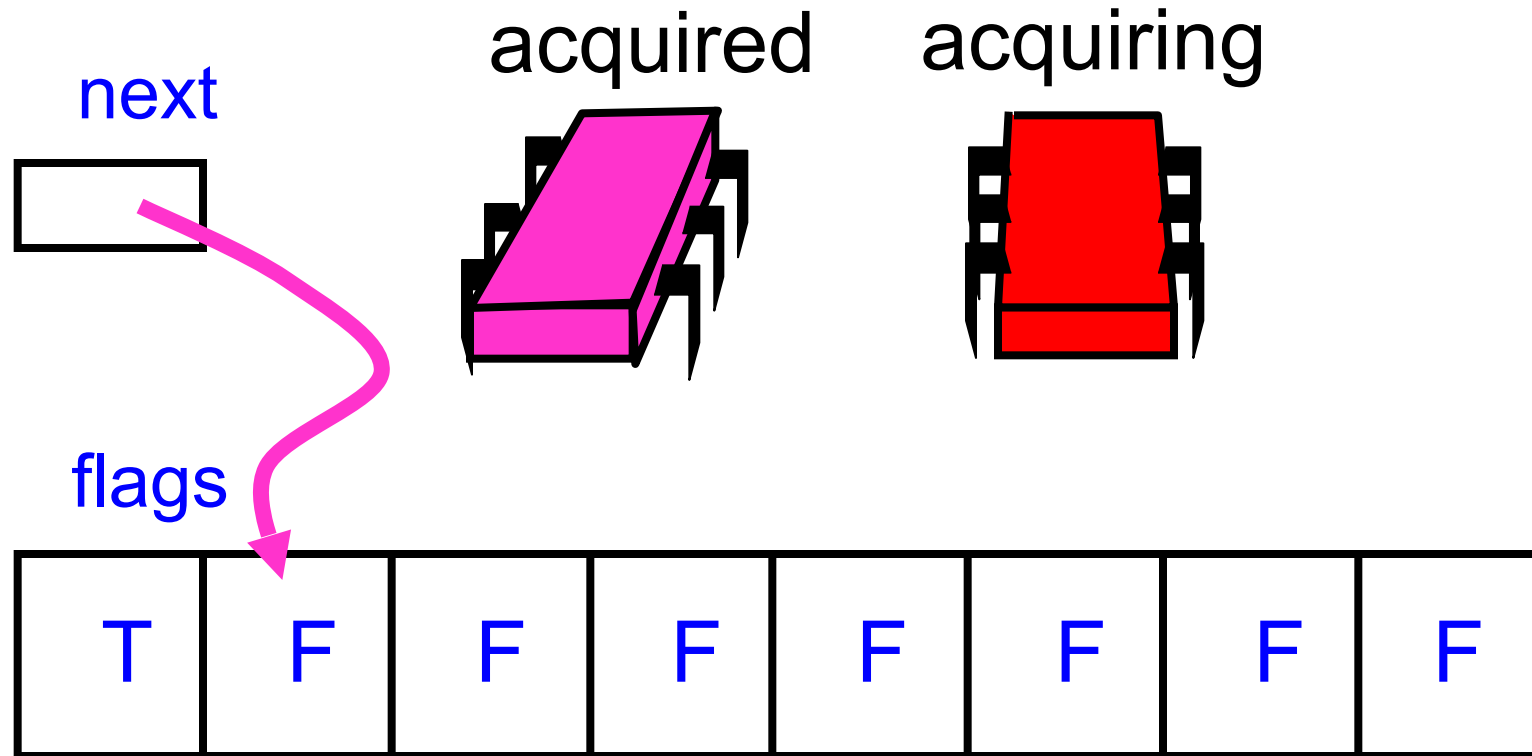
# Anderson Queue Lock



# Anderson Queue Lock

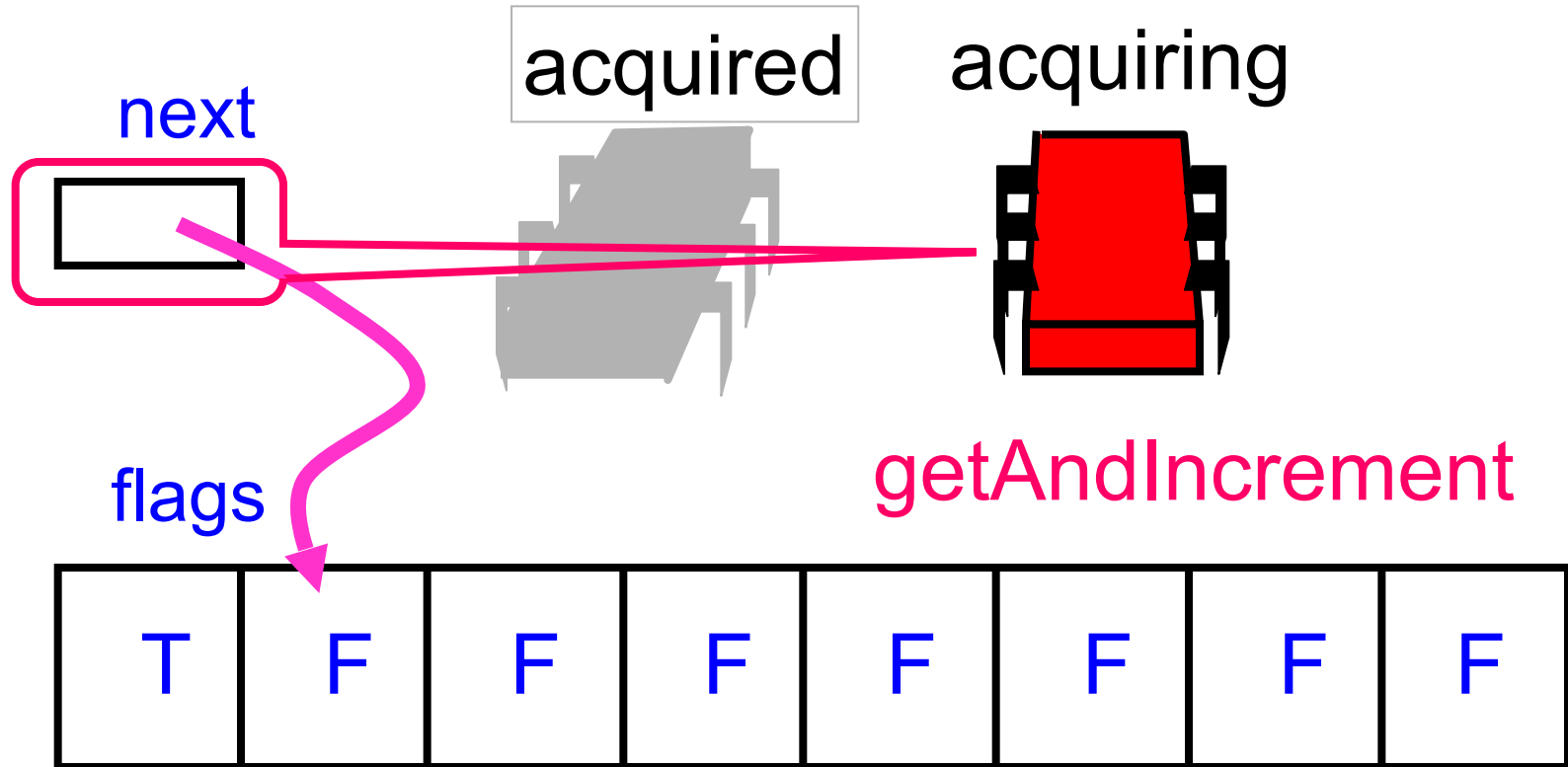


# Anderson Queue Lock

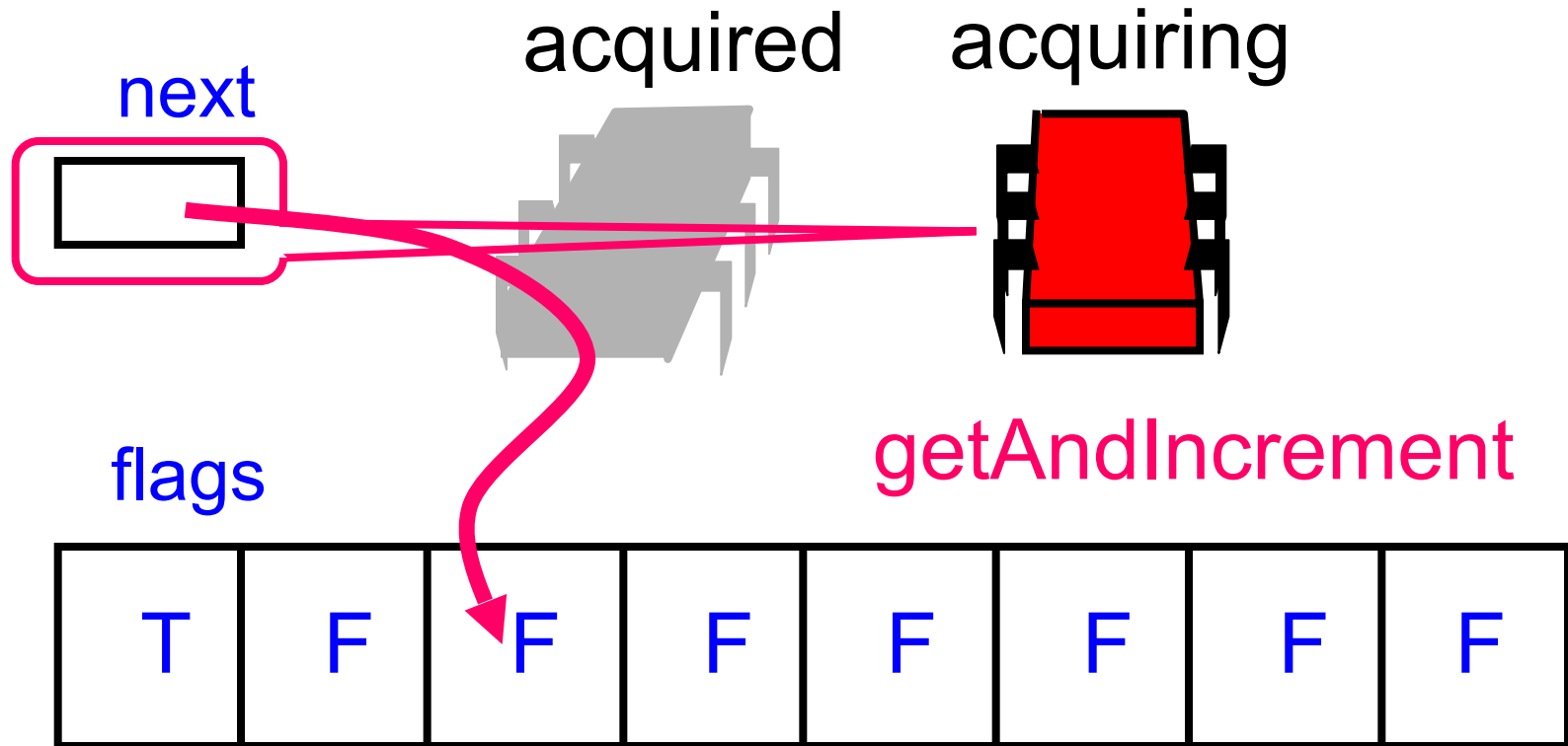




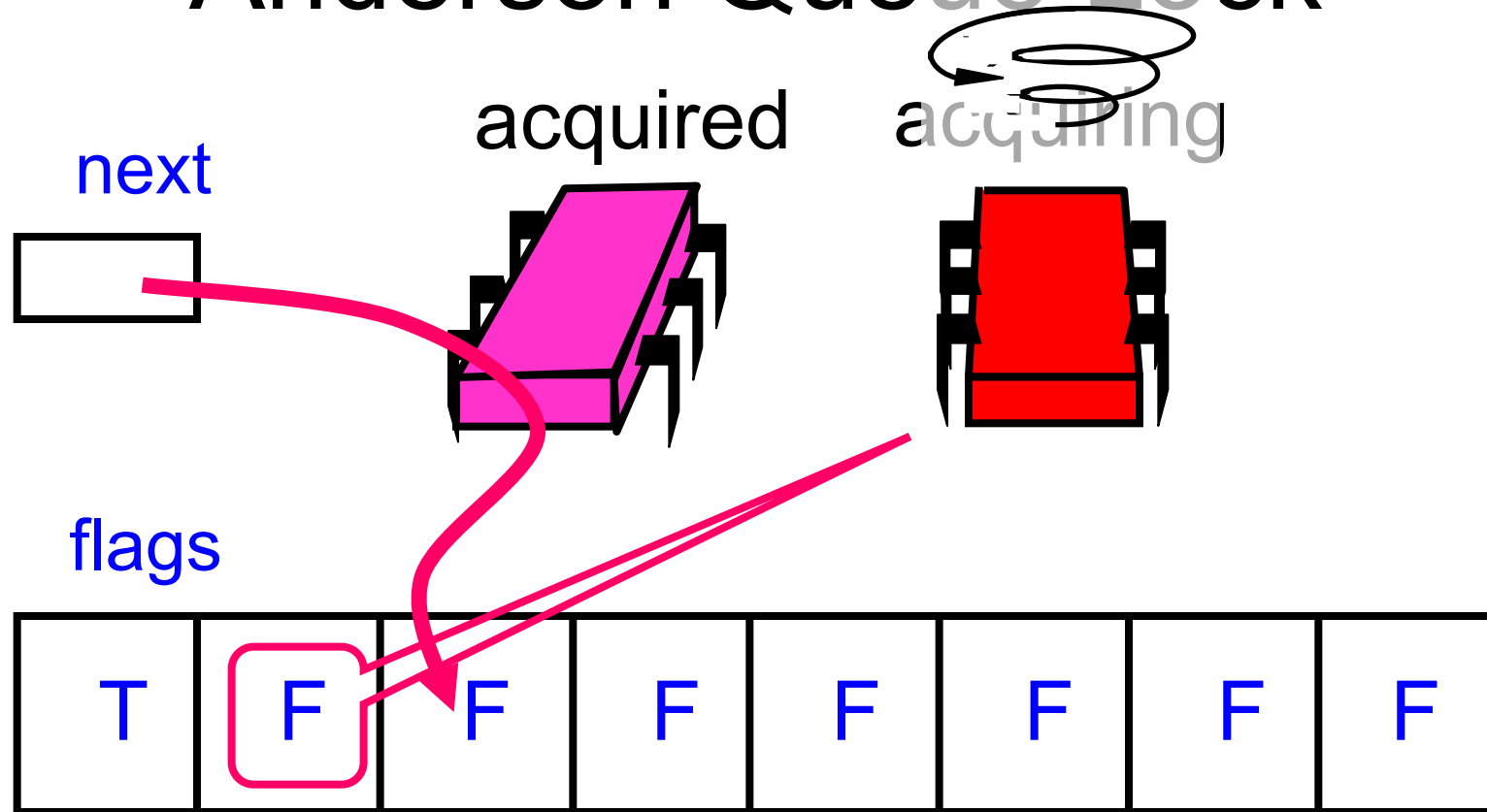
# Anderson Queue Lock



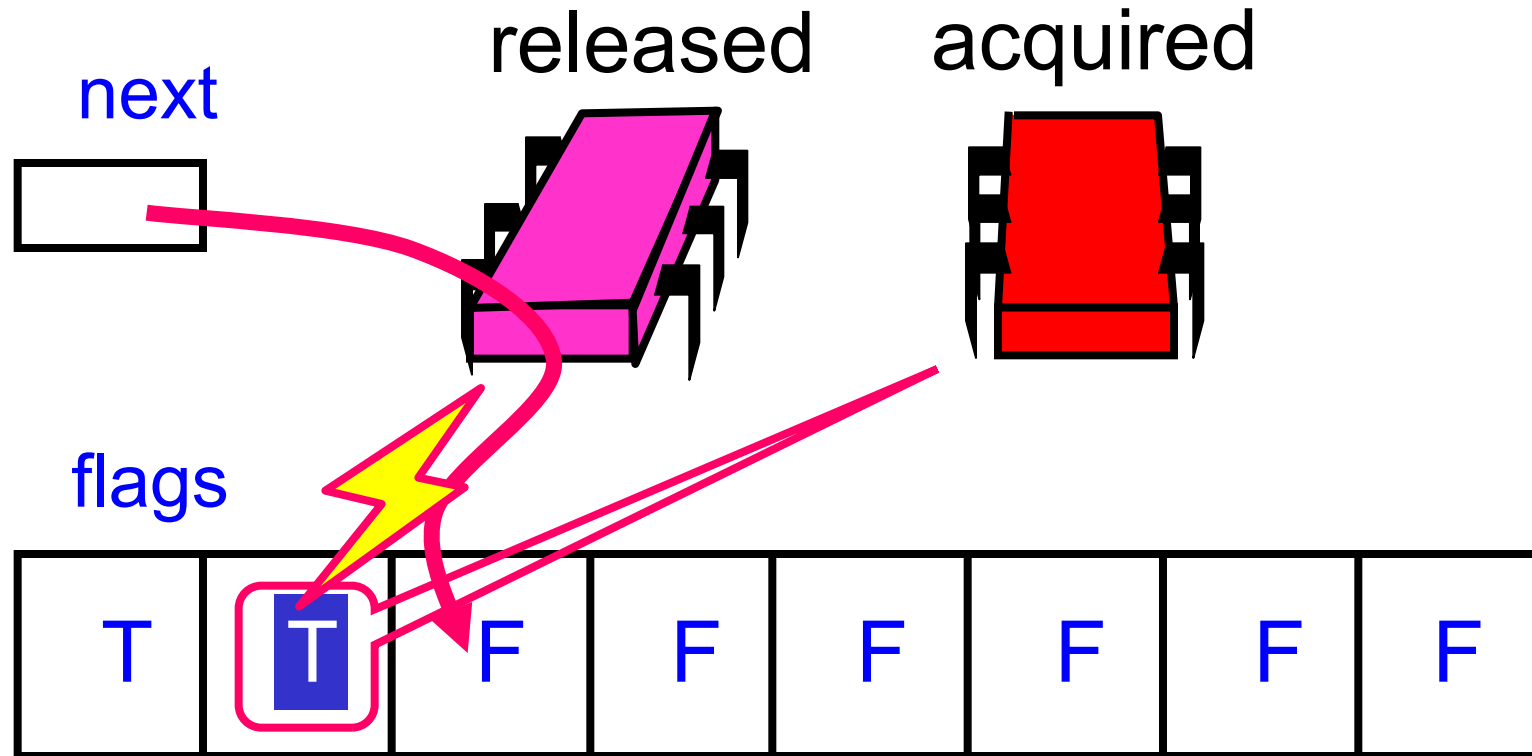
# Anderson Queue Lock



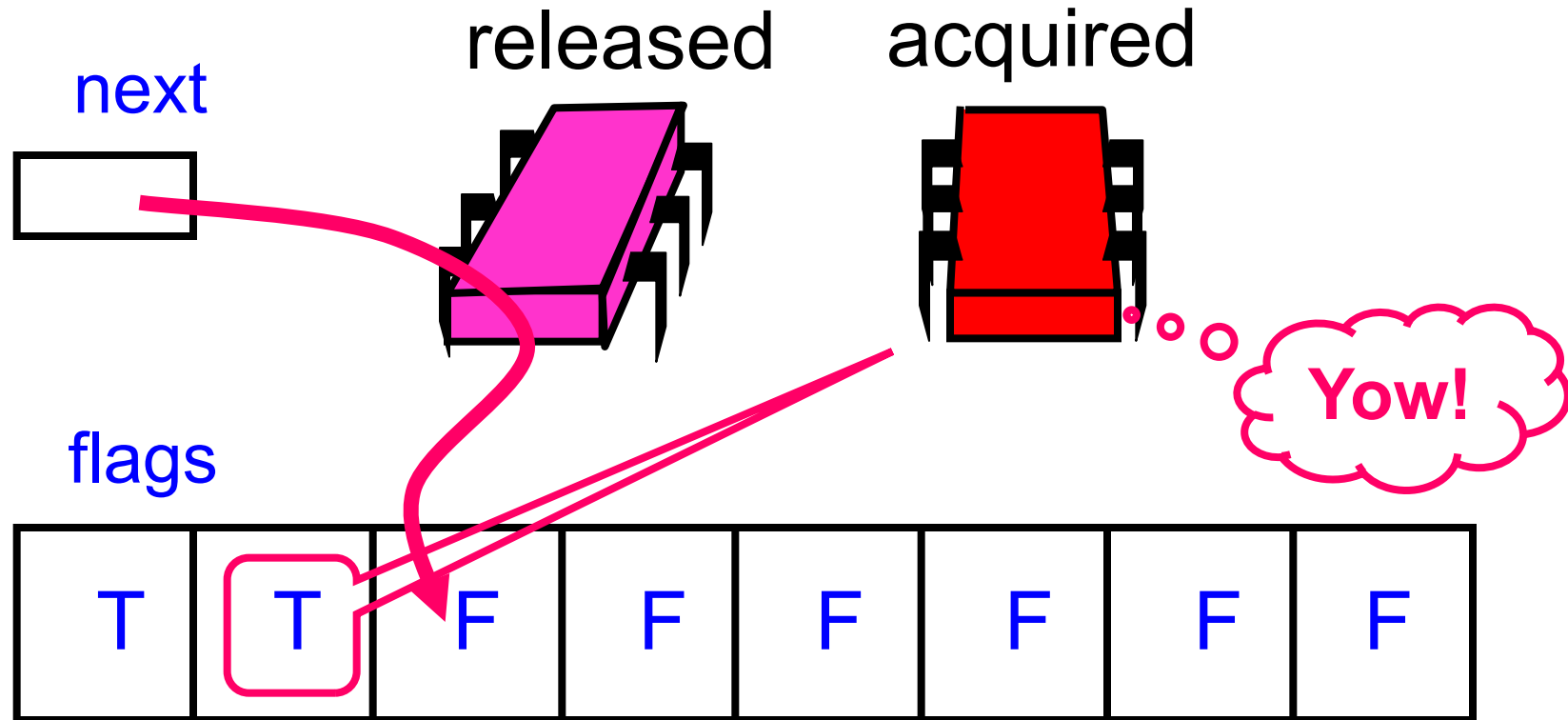
# Anderson Queue Lock



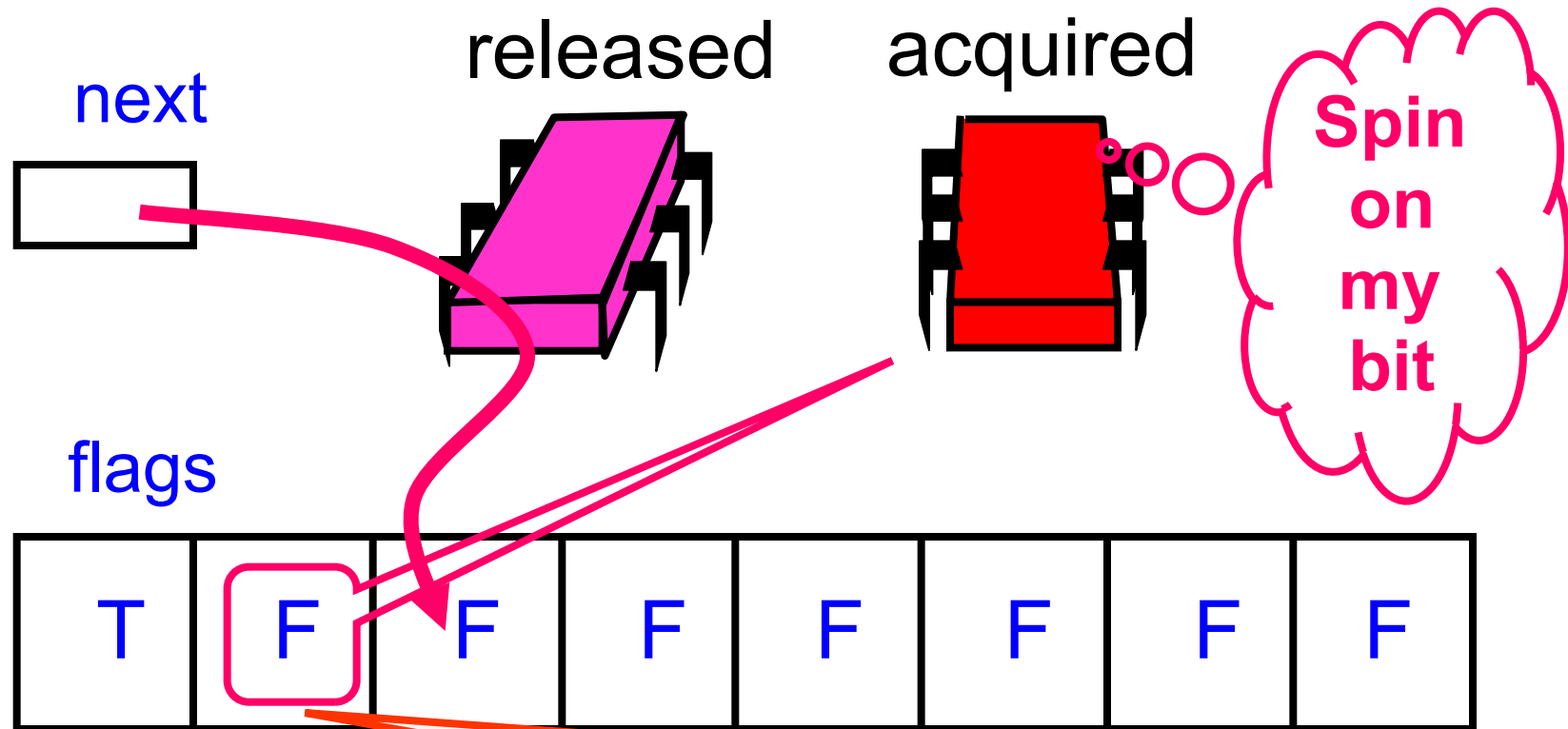
# Anderson Queue Lock



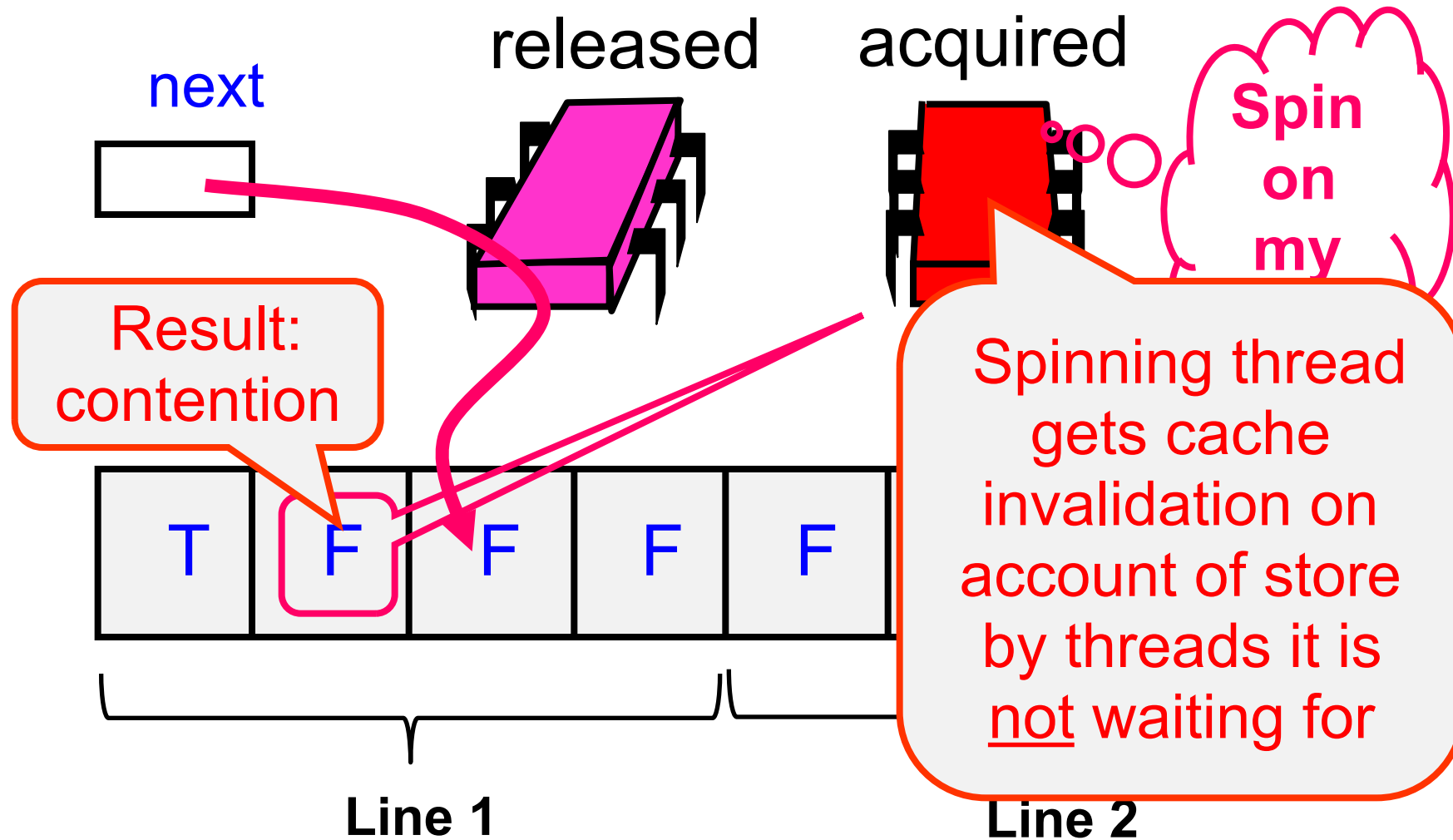
# Anderson Queue Lock



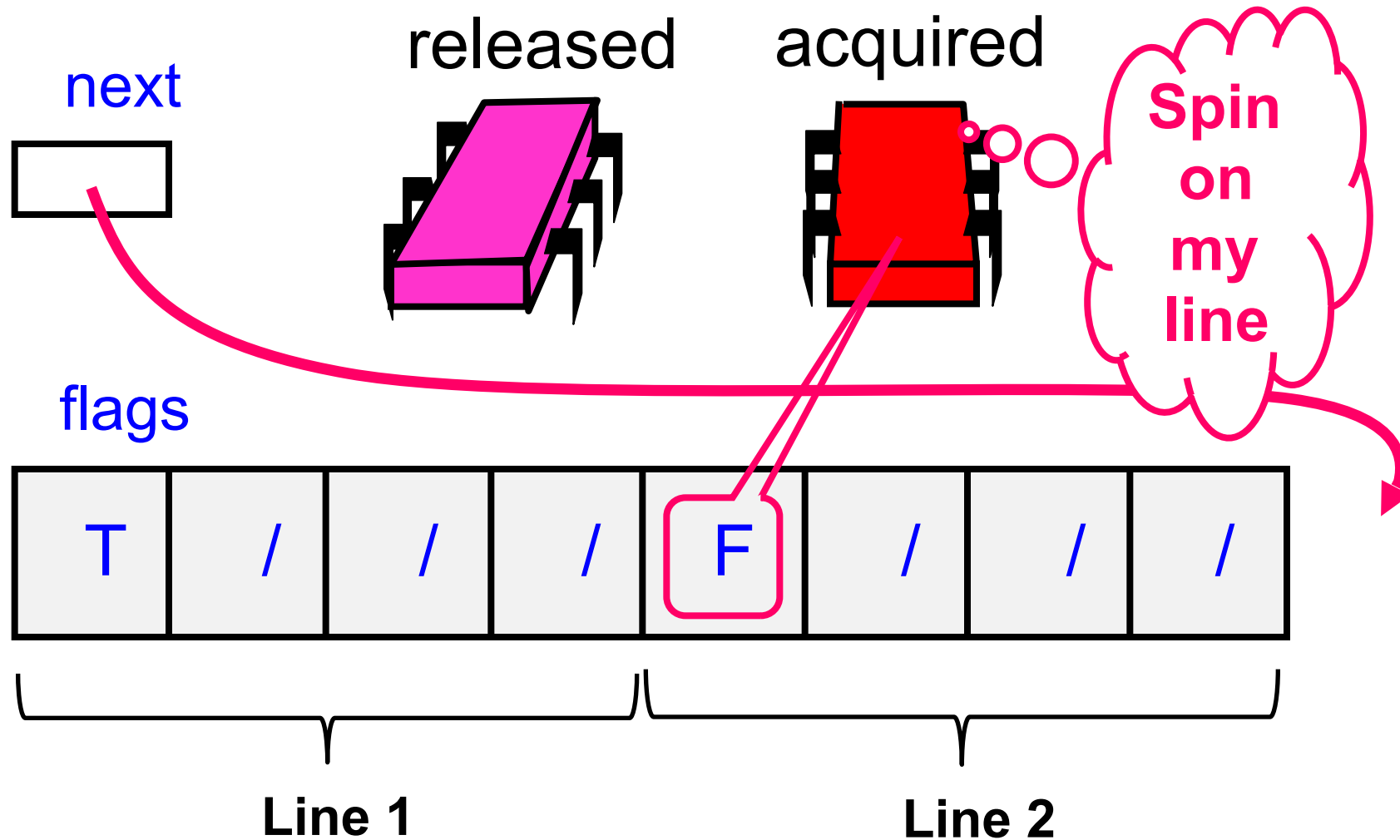
# Local Spinning



# False Sharing

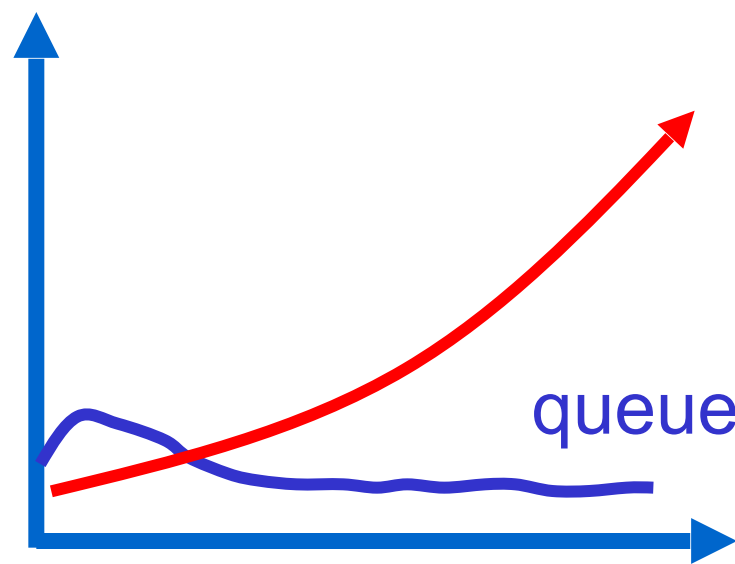


# The Solution: Padding





# Performance

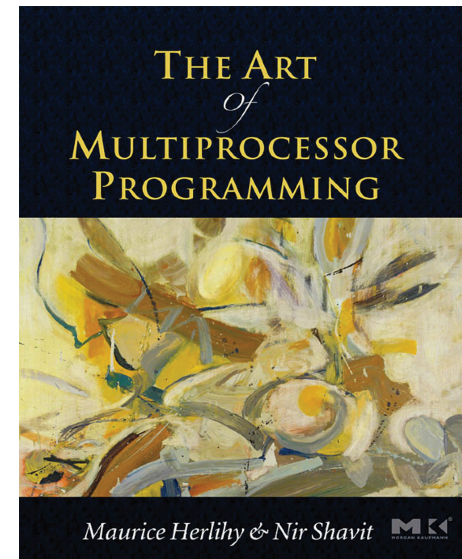


TTAS

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

# More spin-locks in the Book

- CHL Lock
- MCS Lock
- Fast-path composite locks
- Hierarchical backoff locks
- ...
- No silver bullet!



## Chapter 7

# Mind the gap!

- ALock in Java is vulnerable to *false sharing*, which is easy to avoid in C (where you can pad and align flags) but harder in JVM, which tend to pack flags into one cache line.
- Thread-local vars can be very slow. One can implement them by hand as an array indexed by thread ID.
- The standard Java Random class uses an internal static lock.
- Java code for `java.util.concurrent` has lots of low-level Java locks and data structures, but it makes heavy use of the Unsafe package for cache alignment, etc.

# Why should we care?

- Spin-locks are useful when *critical sections are small*, but the the numbers of threads are *large*
- Typical for *high-performance computing* (most of the tasks done in parallel) or low-level kernel drivers. Those are typically not implemented in Java. :-)
- Regular applications (desktop, web) favour the “blocking” model (threads yield the processor to each other).
- We will consider it in the next lecture.



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](http://creativecommons.org/licenses/by-sa/3.0/).

- **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.

