04 Spin Locks and Contention

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

Focus so far: Correctness

- Models
  - Accurate *(we never lied to you)*
  - But idealised *(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.

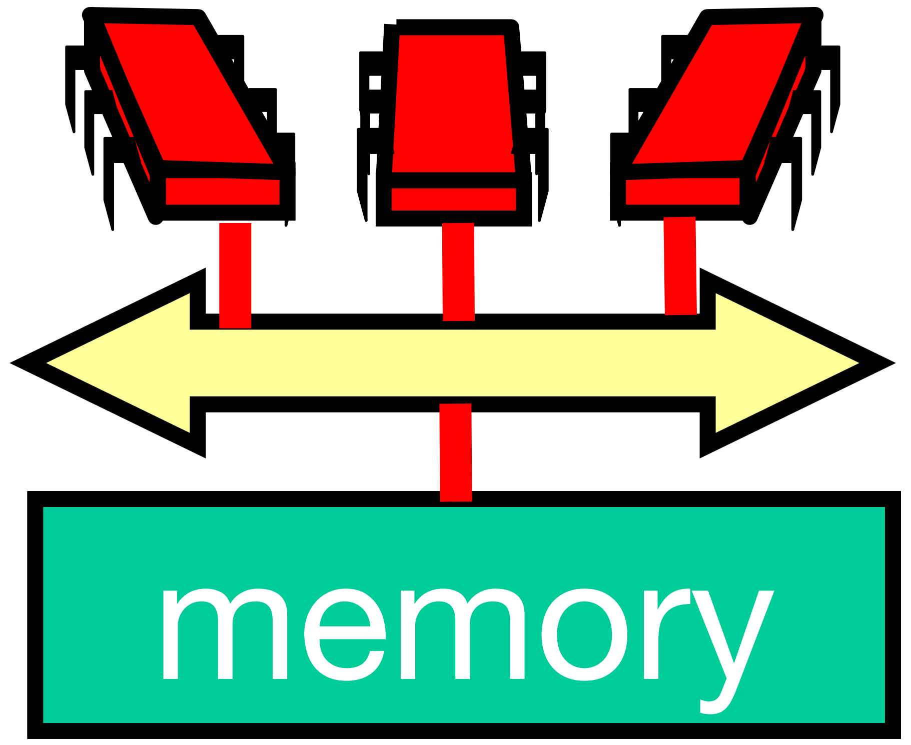
Kinds of Architectures

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

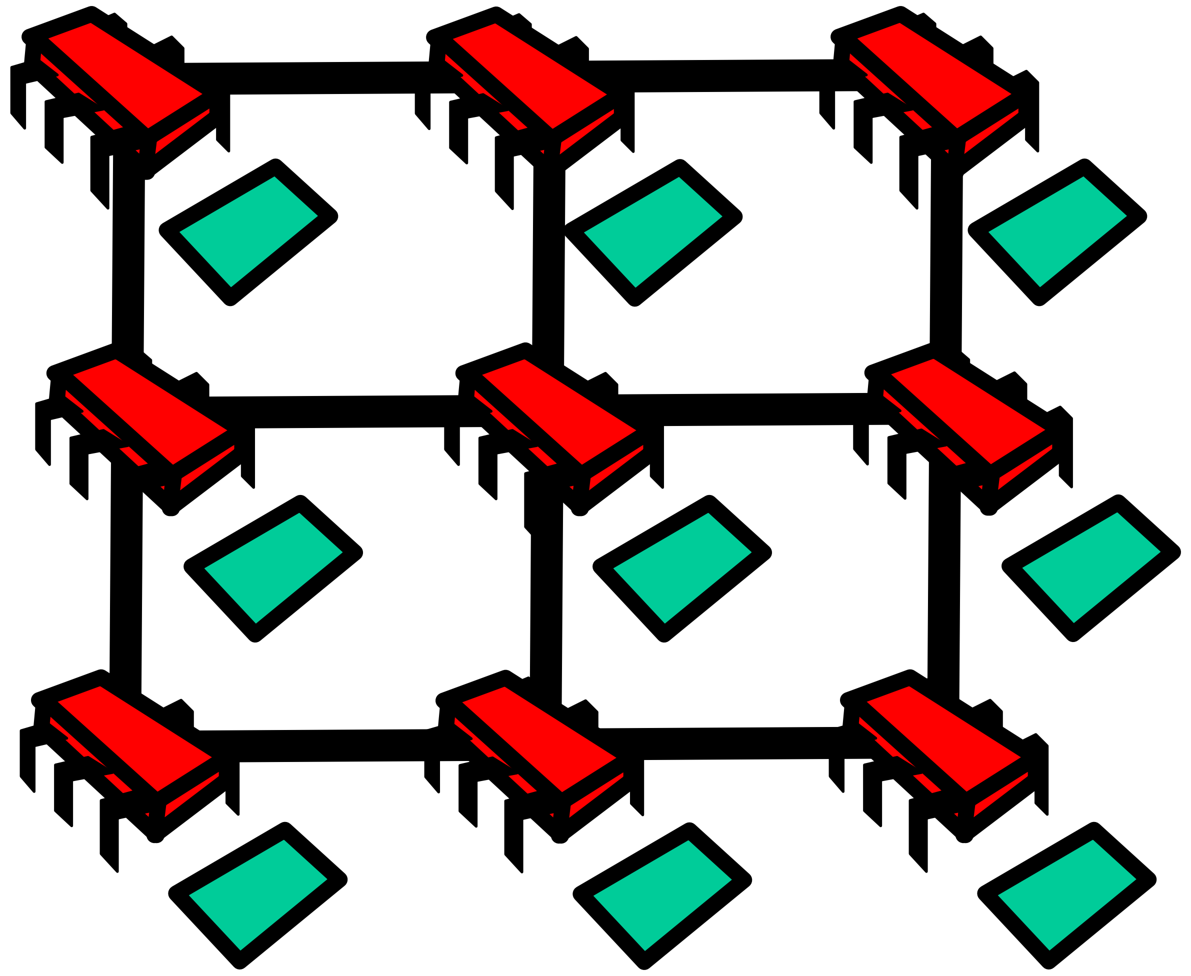
  Our Focus

  - 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?

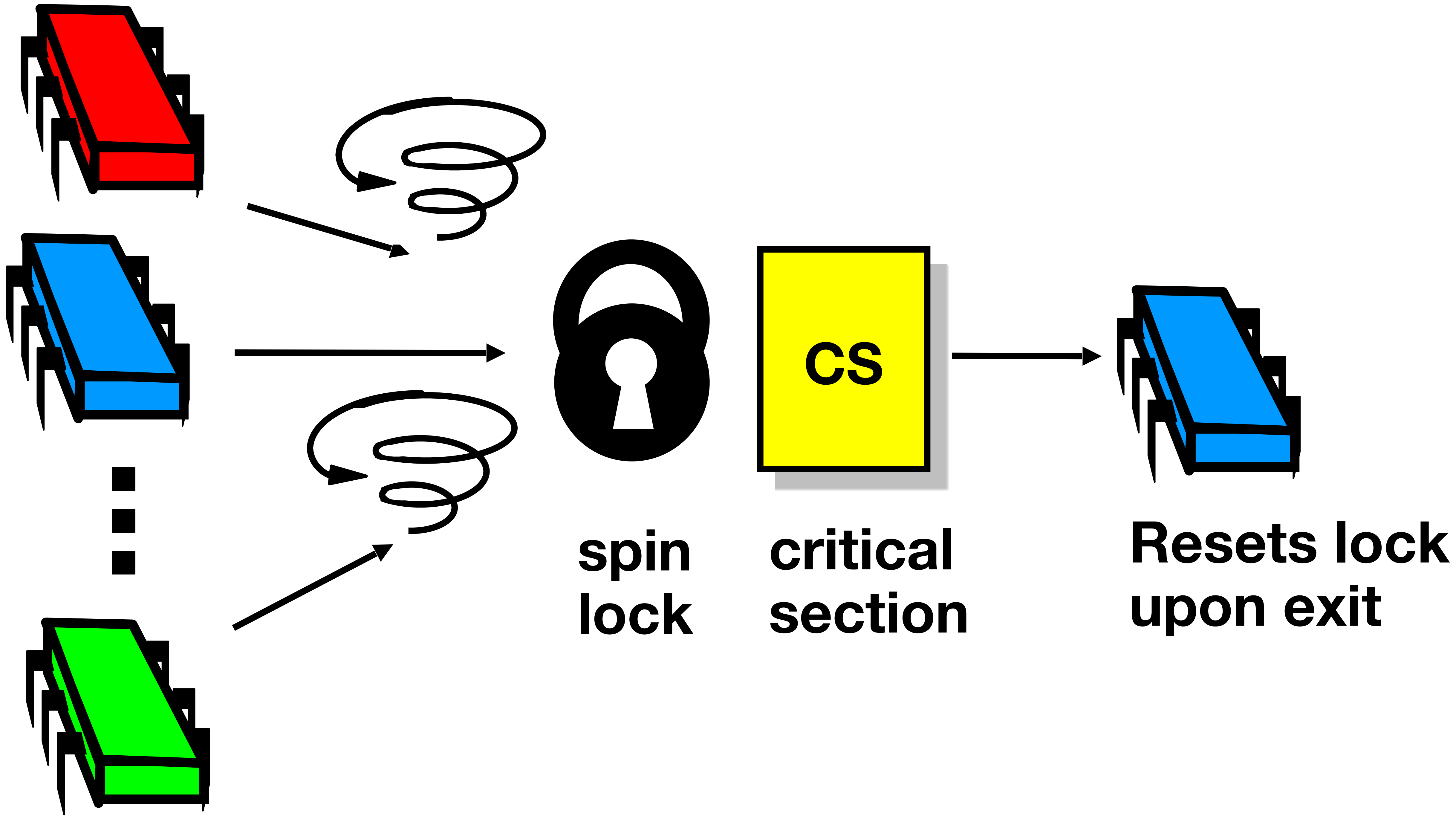
- Keep trying
  - “spin” or “busy-wait”
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

What should you do if you can’t get a lock?

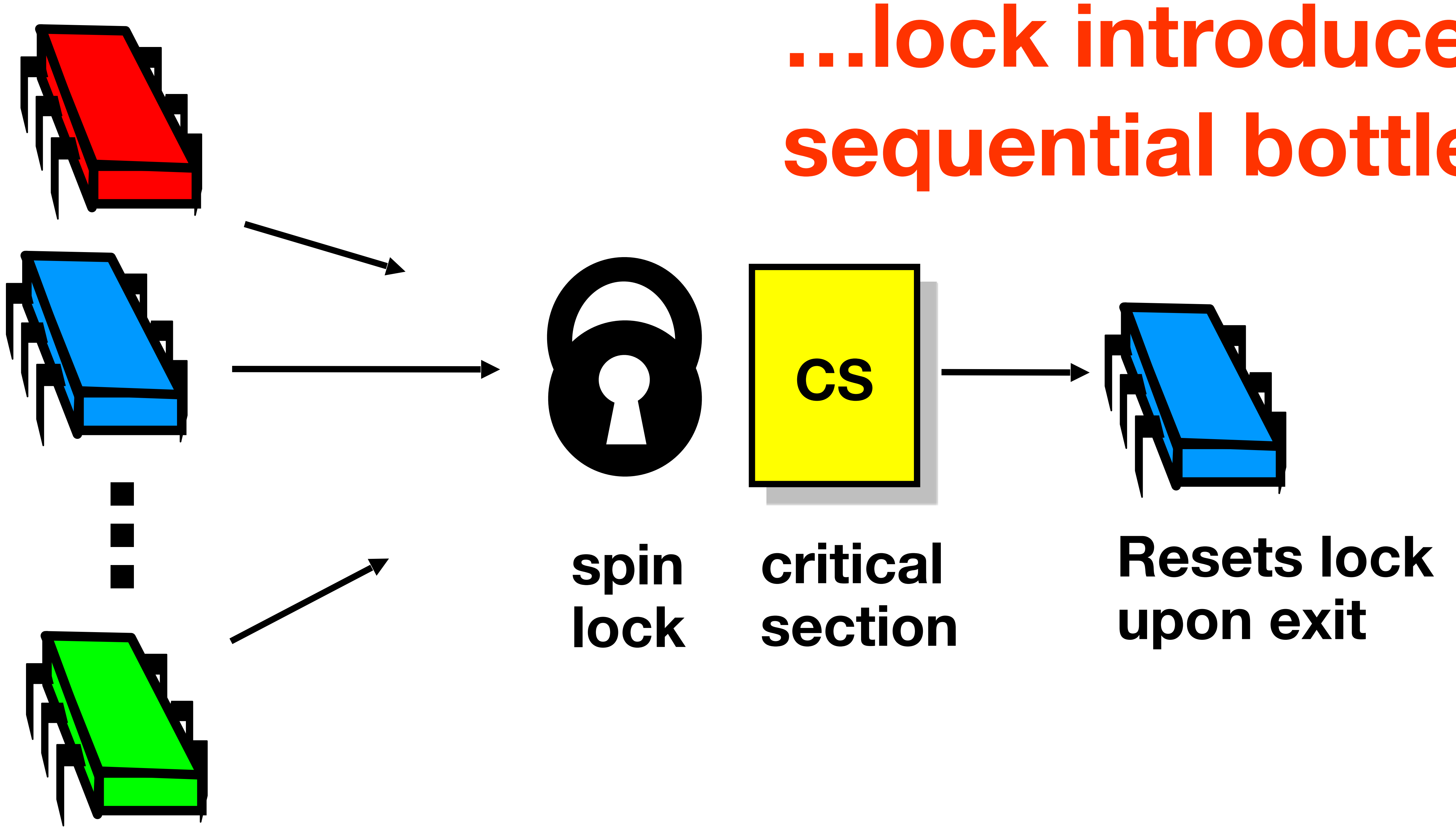
- Keep trying
  - “spin” or “busy-wait”
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

Our Focus

Basic Spin Lock



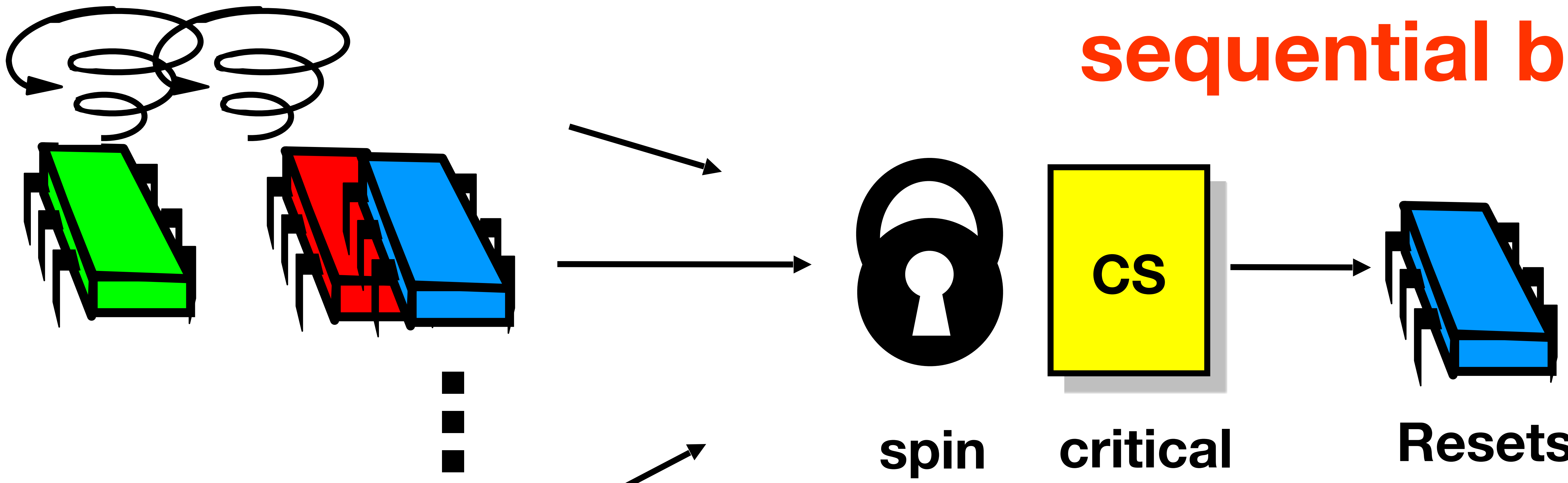
Basic Spin Lock



...lock introduces sequential bottleneck

Basic Spin Lock

…lock introduces sequential bottleneck



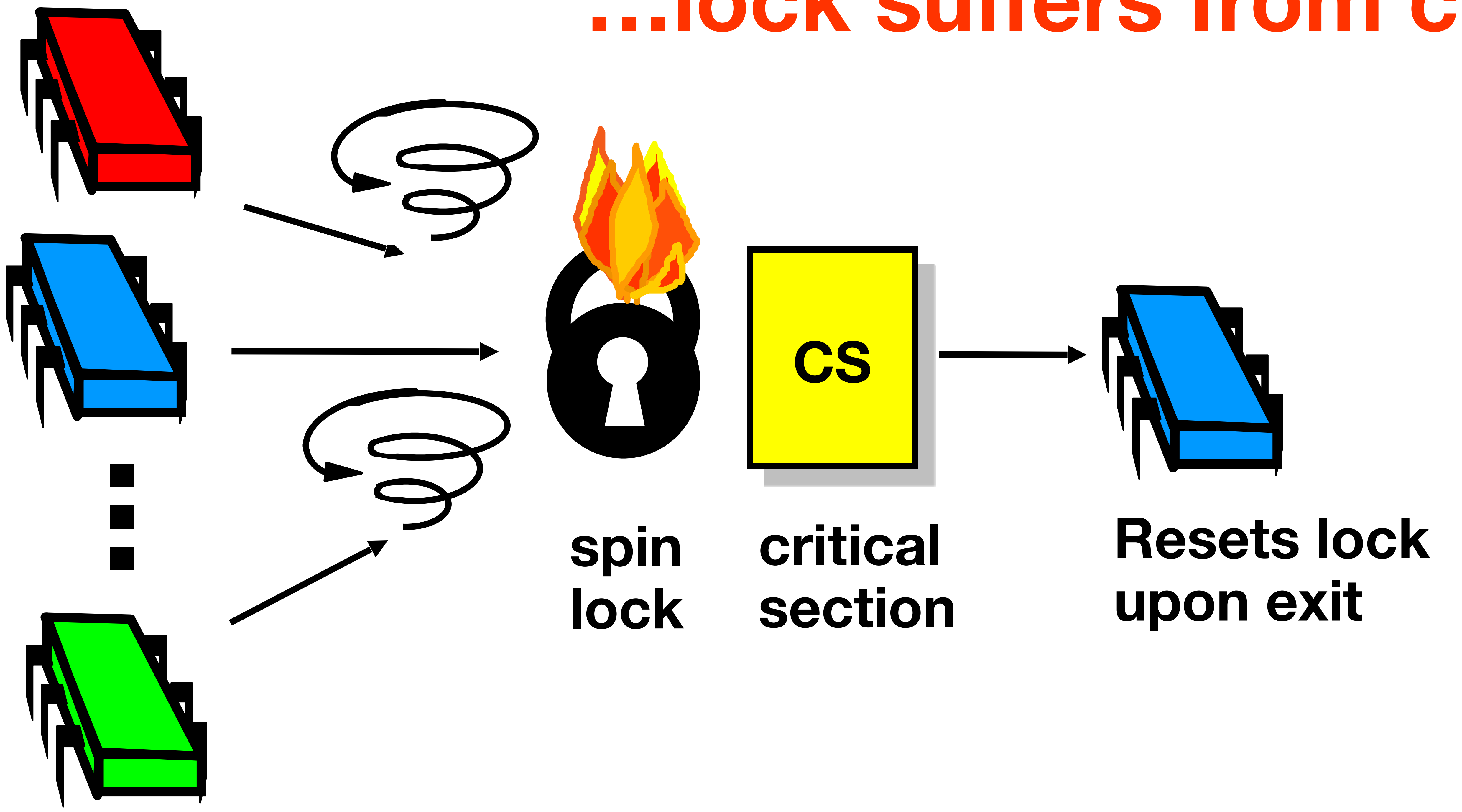
spin lock

critical section

Resets lock upon exit

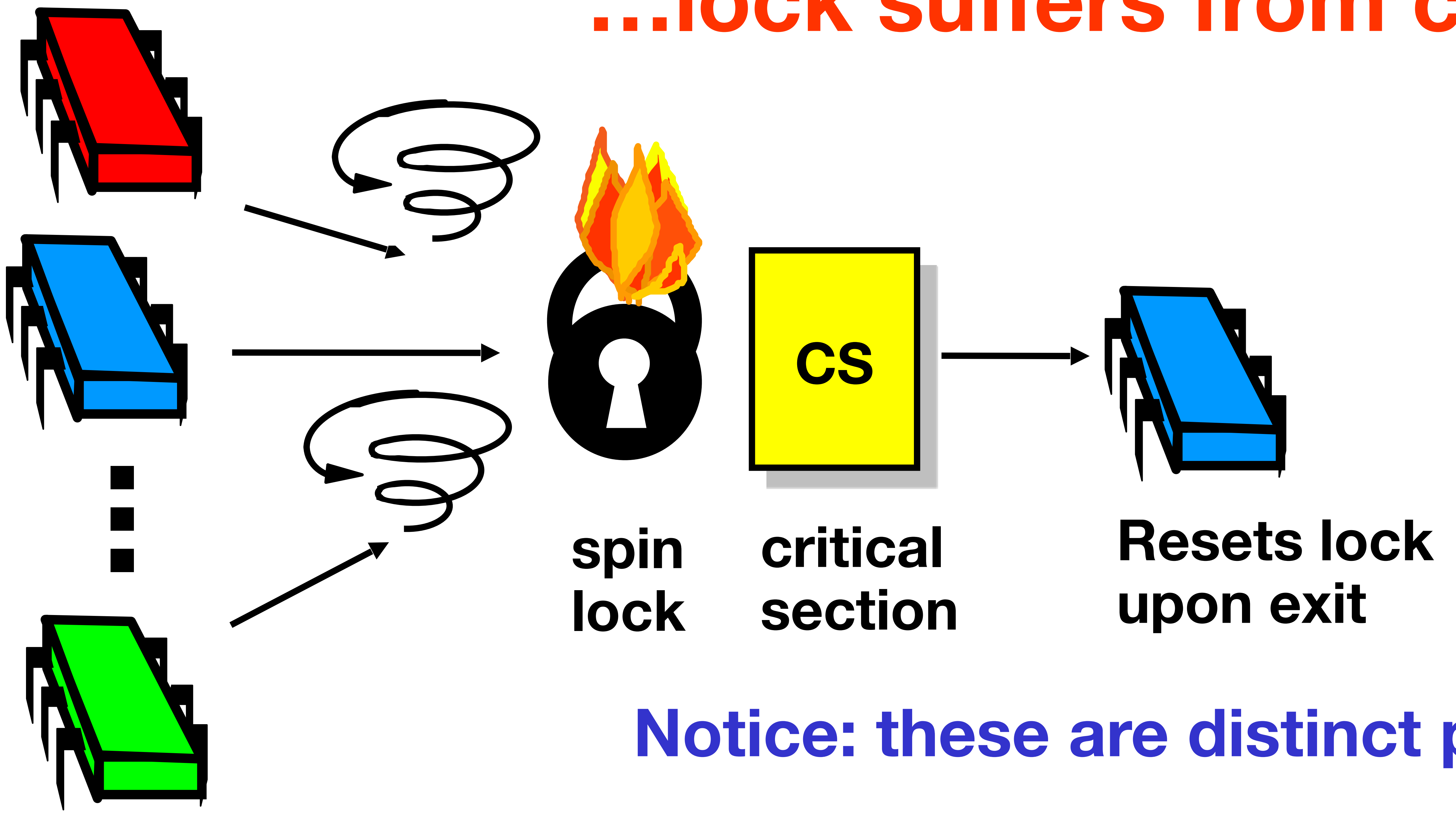
Basic Spin Lock

...lock suffers from contention



Basic Spin Lock

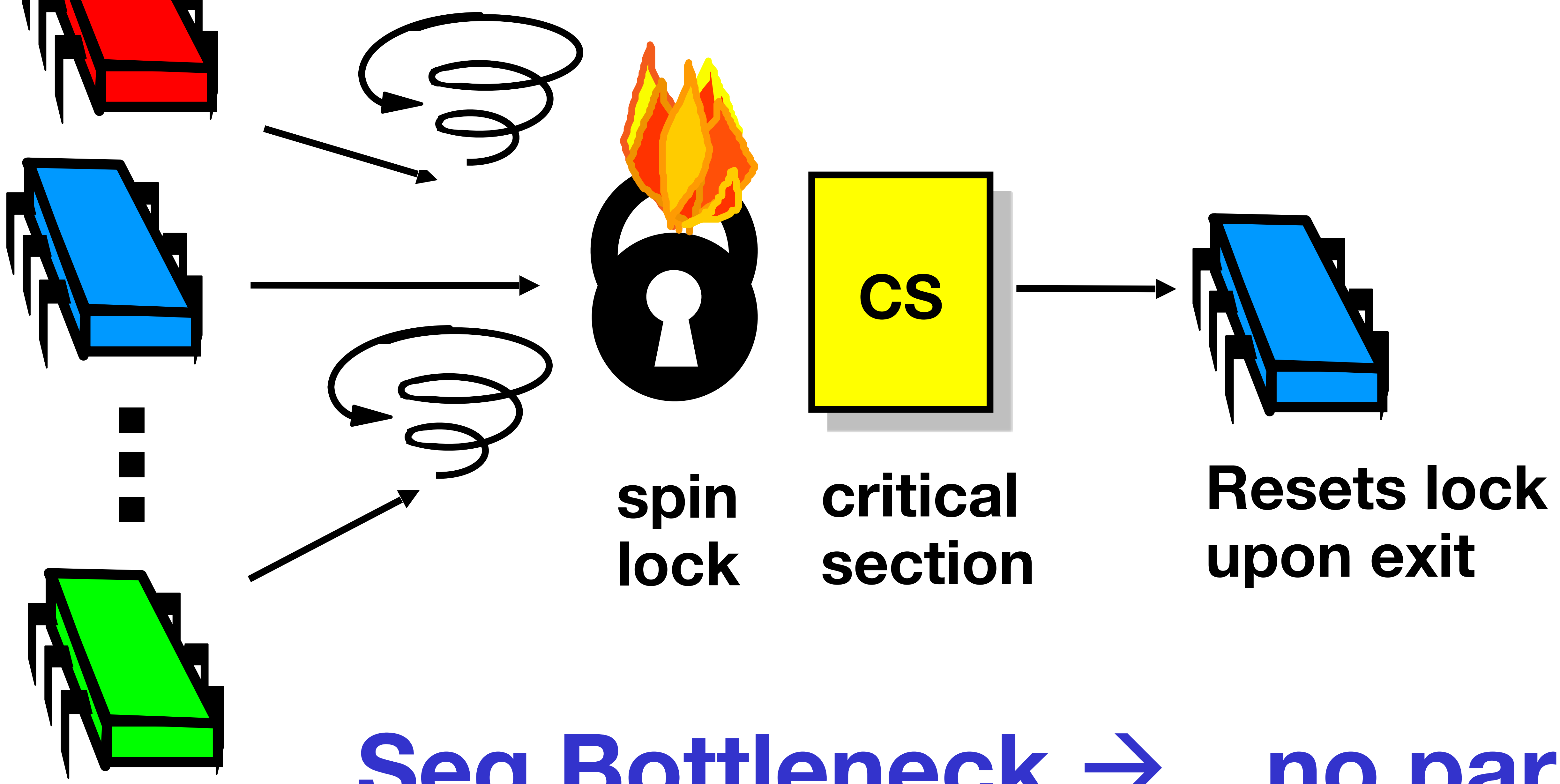
…lock suffers from contention



Notice: these are distinct phenomena

Basic Spin Lock

...lock suffers from contention



CS

spin lock

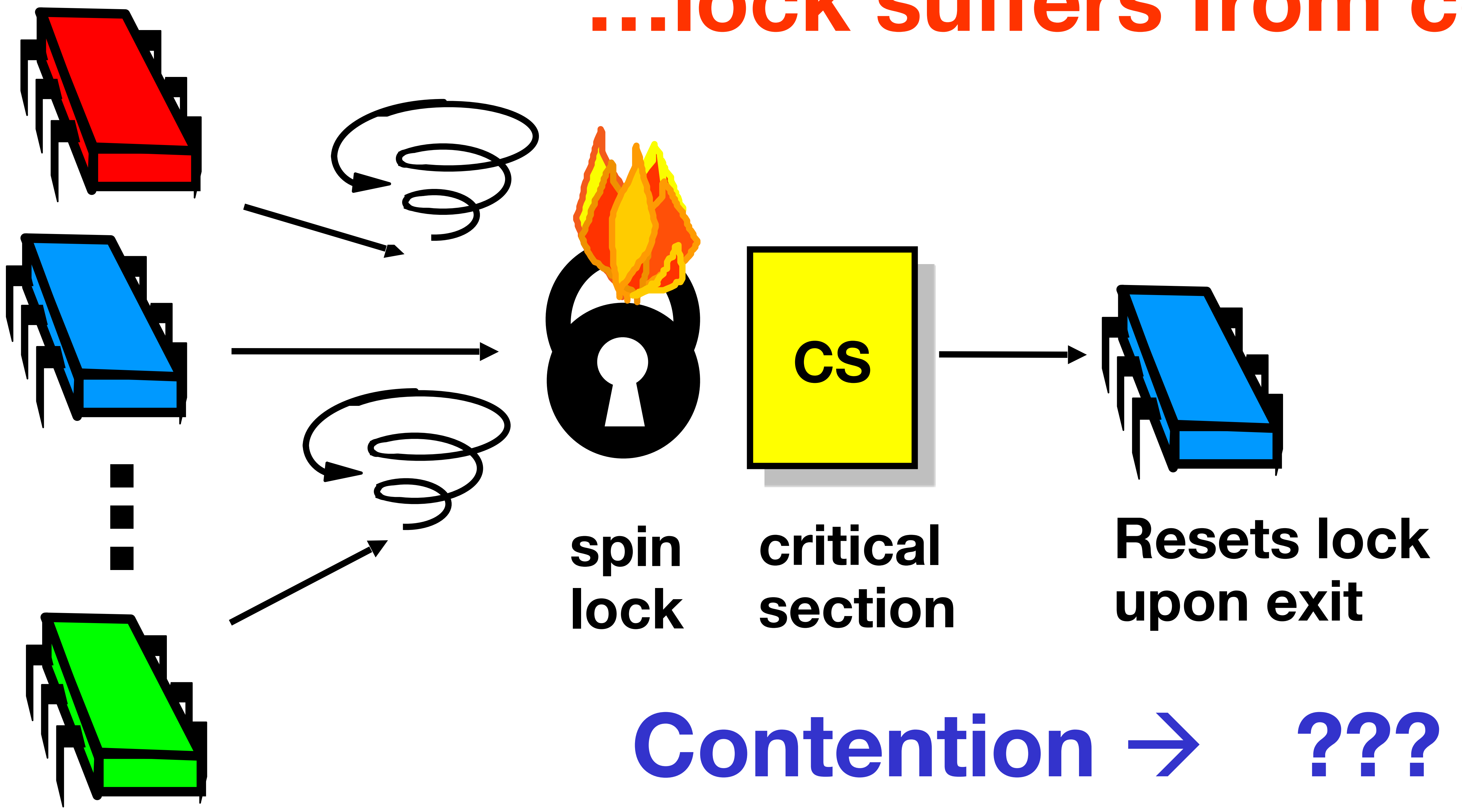
critical section

Resets lock upon exit

Seq Bottleneck → no parallelism

Basic Spin Lock

...lock suffers from contention



Contention → ???

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 “exchange”

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 “exchange”

```
Atomic.exchange : 'a Atomic.t -> 'a -> 'a
```

is semantically equivalent to

```
let exchange (r : 'a Atomic.t) (v : 'a) : 'a =
  let old = Atomic.get r in
  Atomic.set r v;
  old
```

*but **atomic***

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 (TASLock)

```
module TASLock : Lock.LOCK = struct
  type t = { state : bool Atomic.t }

  let create () = { state = Atomic.make false }

  let lock t =
    (* Keep spinning until we successfully acquire the lock *)
    (* Atomic.exchange sets the value and returns the old value *)
    while Atomic.exchange t.state true do
      (* Spin - this is where the thread wastes CPU cycles *)
      ()
    done

  let unlock t = Atomic.set t.state false
end
```

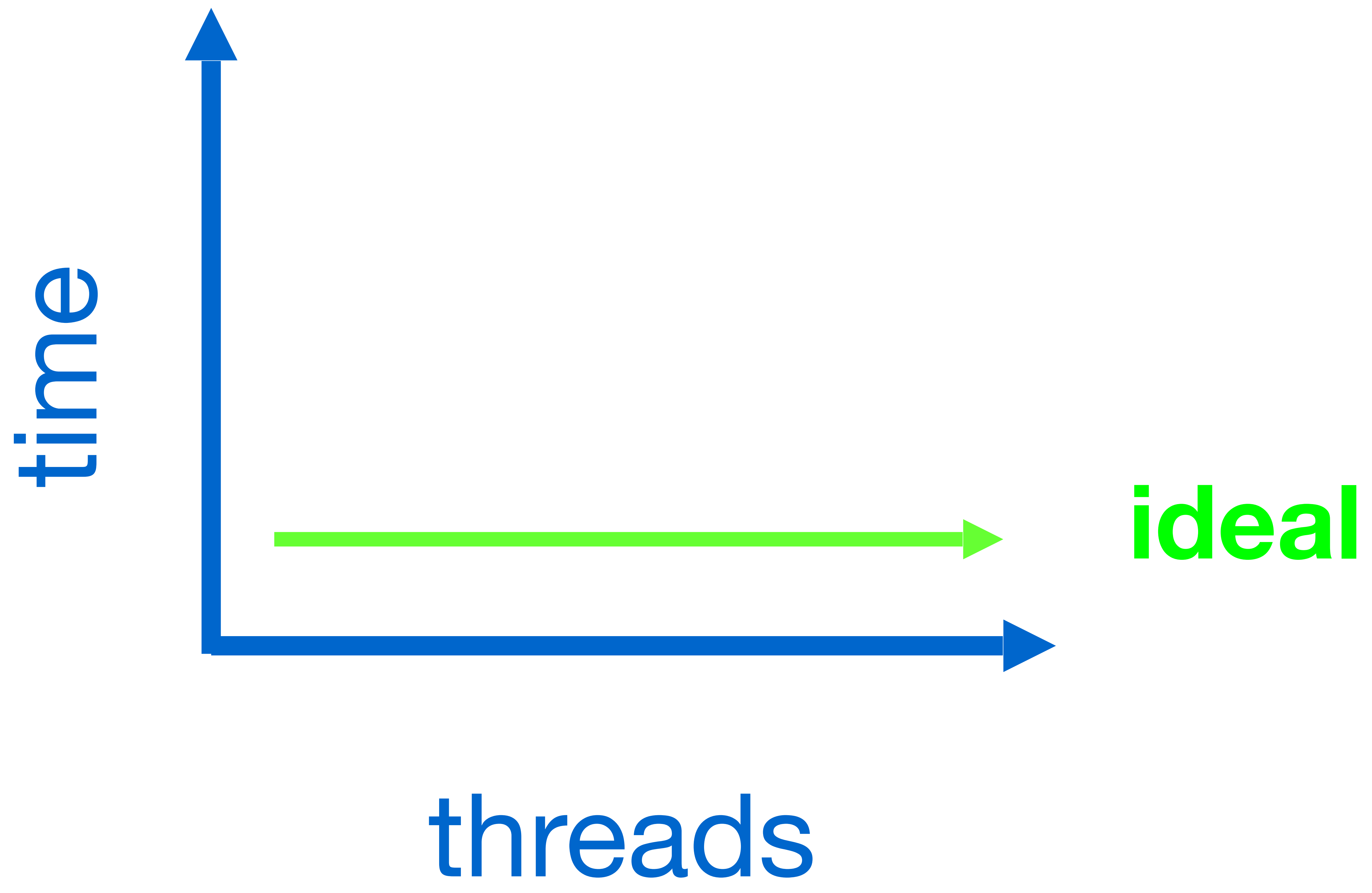
Space Complexity

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

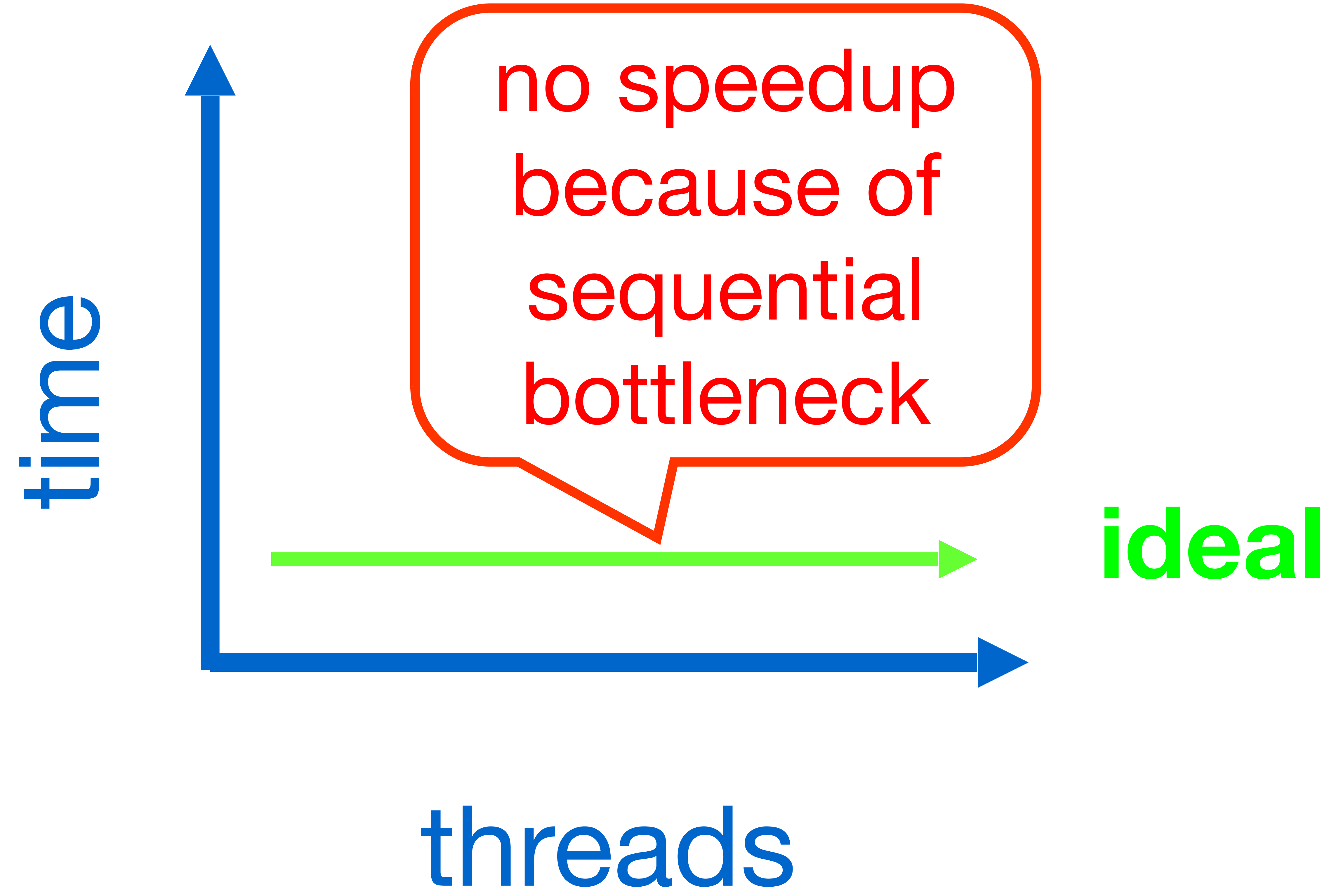
Space Complexity

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

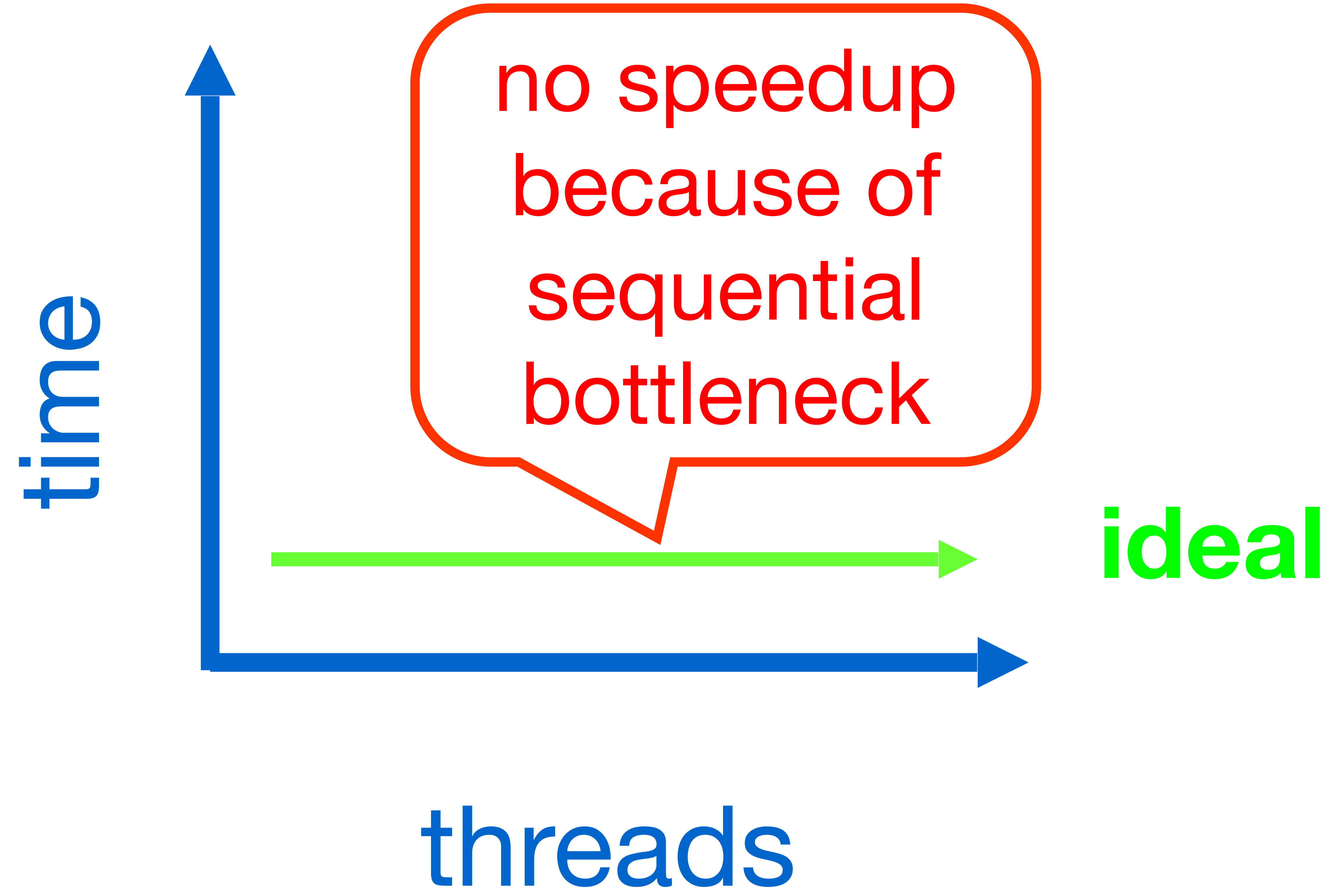
Time taken



Time taken

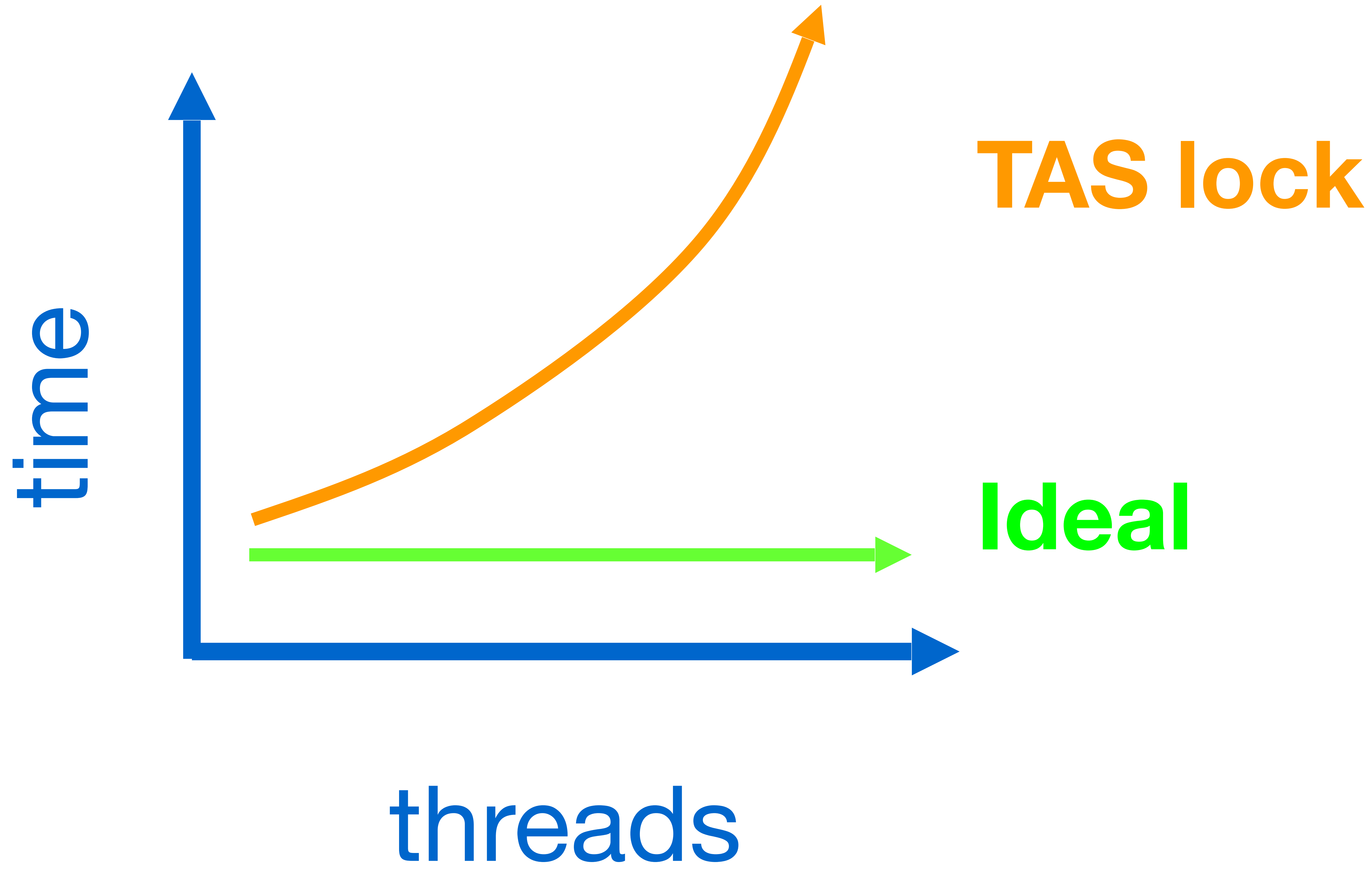

no speedup because of sequential bottleneck
ideal
time
threads

Time taken

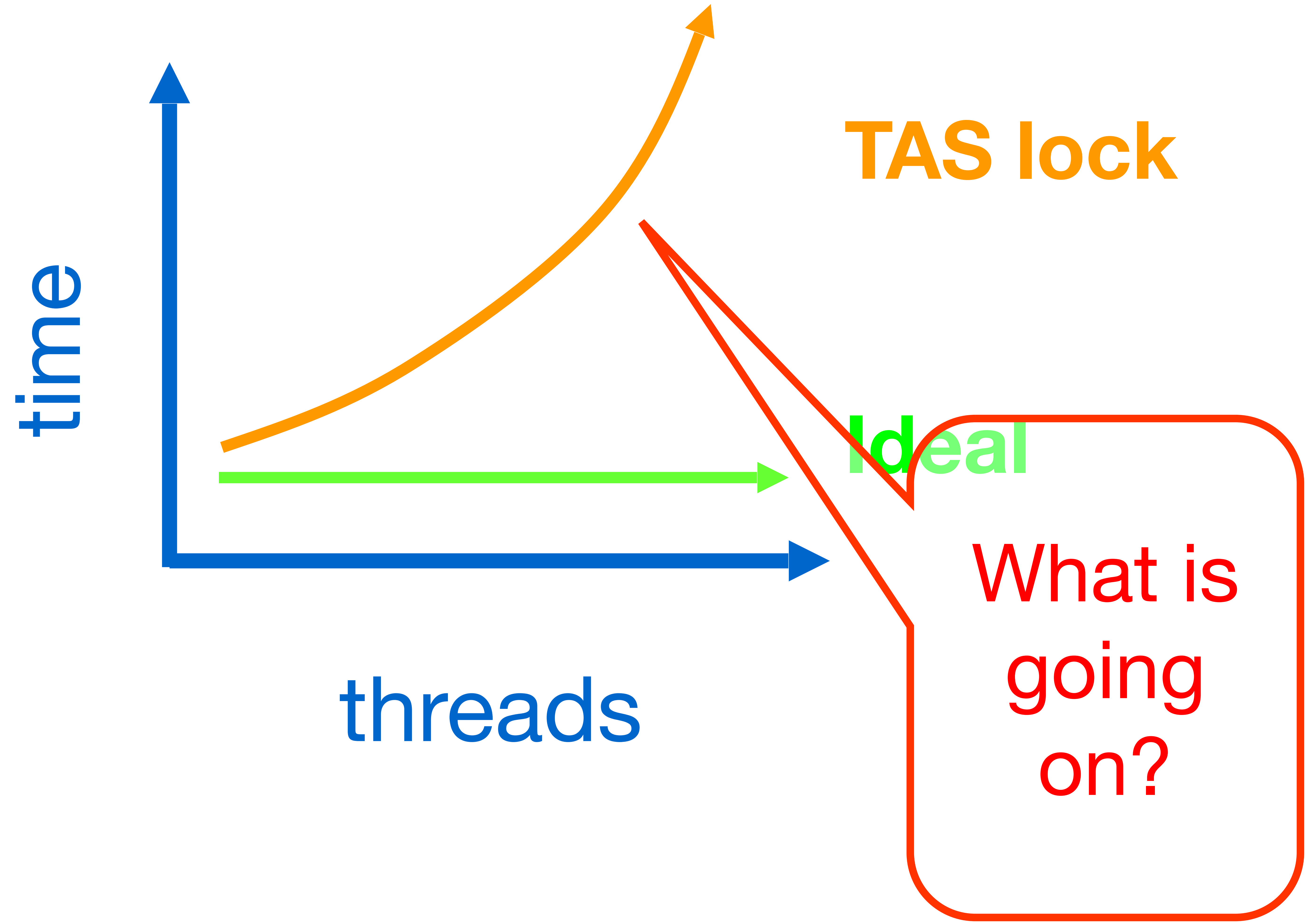


Demo

Mystery #1



Mystery #1



What is going on?

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

```
module TTASLock : Lock.LOCK = struct
  type t = { state : bool Atomic.t }

  let create () = { state = Atomic.make false }

  let unlock t = Atomic.set t.state false
```

```
 let lock t =
    (* Outer loop: keep trying until we get the lock *)
    while
      (* Inner loop: spin-read until lock appears free *)
      while Atomic.get t.state do () done;
      (* Lock looks free, try to acquire with exchange *)
     Atomic.exchange t.state true
    do
      (* If we're here, exchange returned true *)
      (* Somebody has the lock; spin-read again *)
      ()
    done
    (* If we exit the while loop, exchange returned false
       => we have the lock! *)
end
```

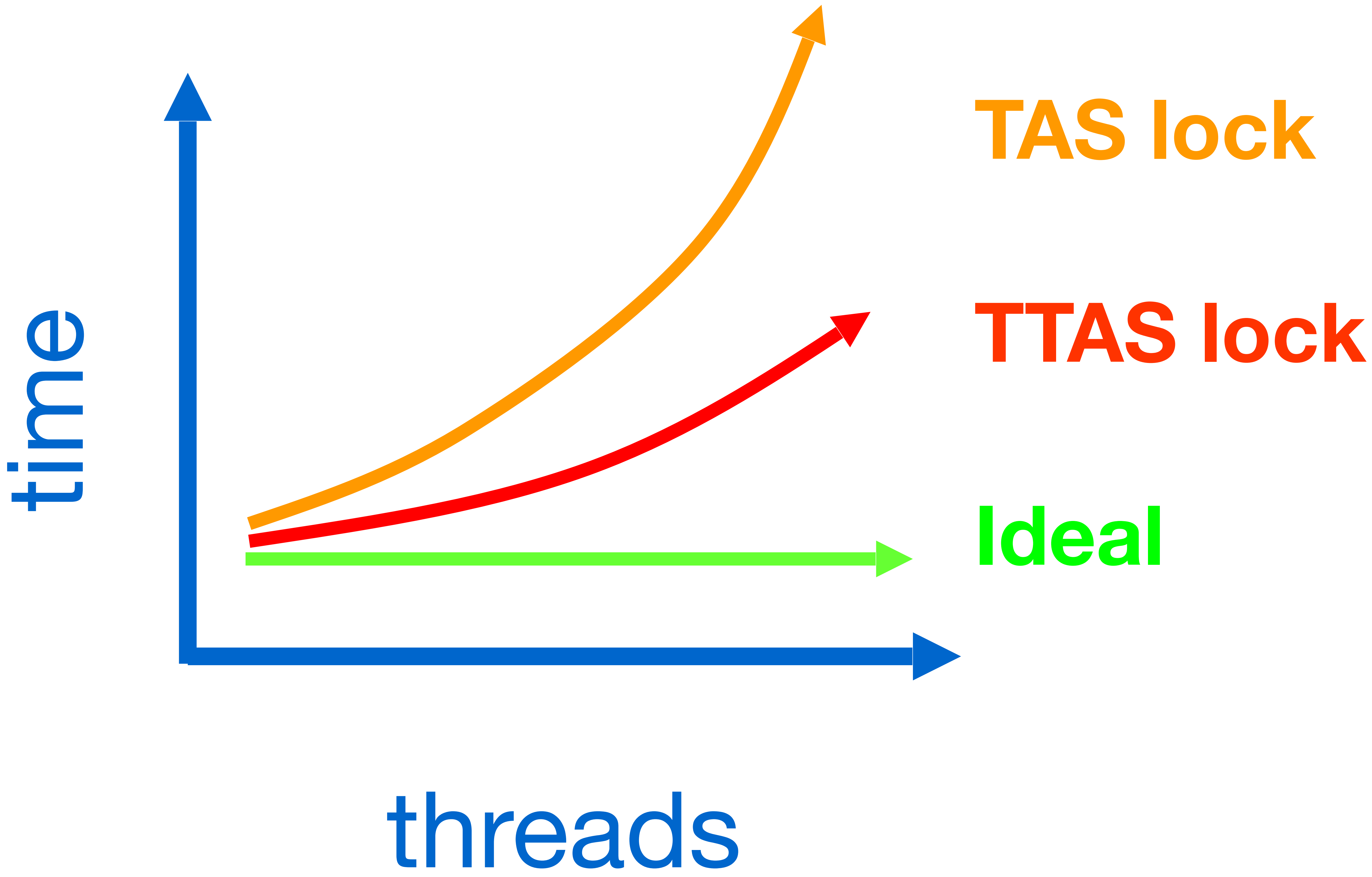
Test-and-Test-and-Set Lock

```
module TTASLock : Lock.LOCK = struct
  type t = { state : bool Atomic.t }

  let create () = { state = Atomic.make false }

  let unlock t = Atomic.set t.state false
```

```
 let lock t =
    (* Outer loop: keep trying until we get the lock *)
    while
      (* Inner loop: spin-read until lock appears free *)
      while Atomic.get t.state do () done;
      (* Lock looks free, try to acquire with exchange *)
     Atomic.exchange t.state true
    do
      (* If we're here, exchange returned true *)
      (* Somebody has the lock; spin-read again *)
      ()
    done
    (* If we exit the while loop, exchange returned false
       => we have the lock! *)
end
```

Demo

Mystery #2



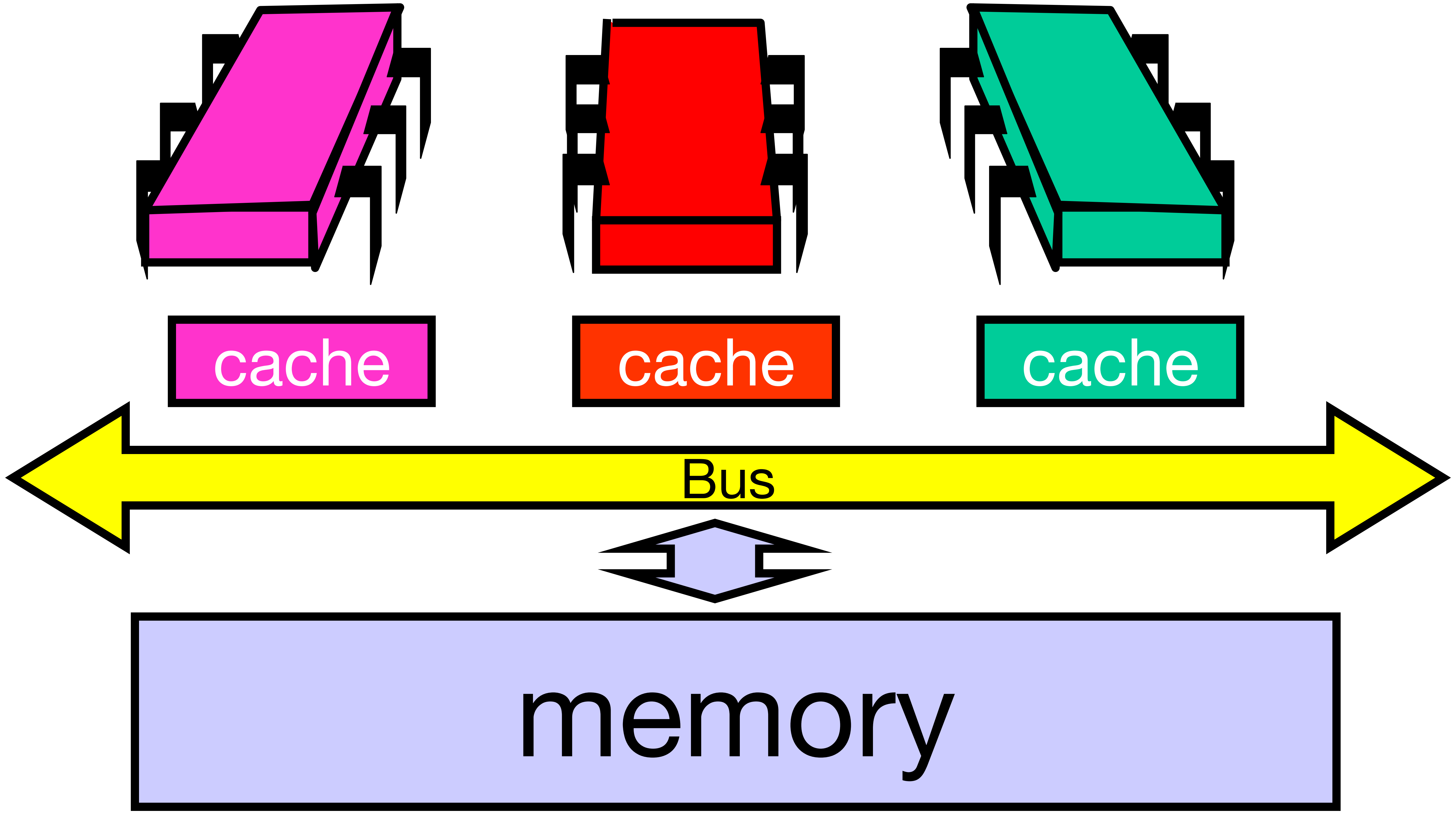
Mystery

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

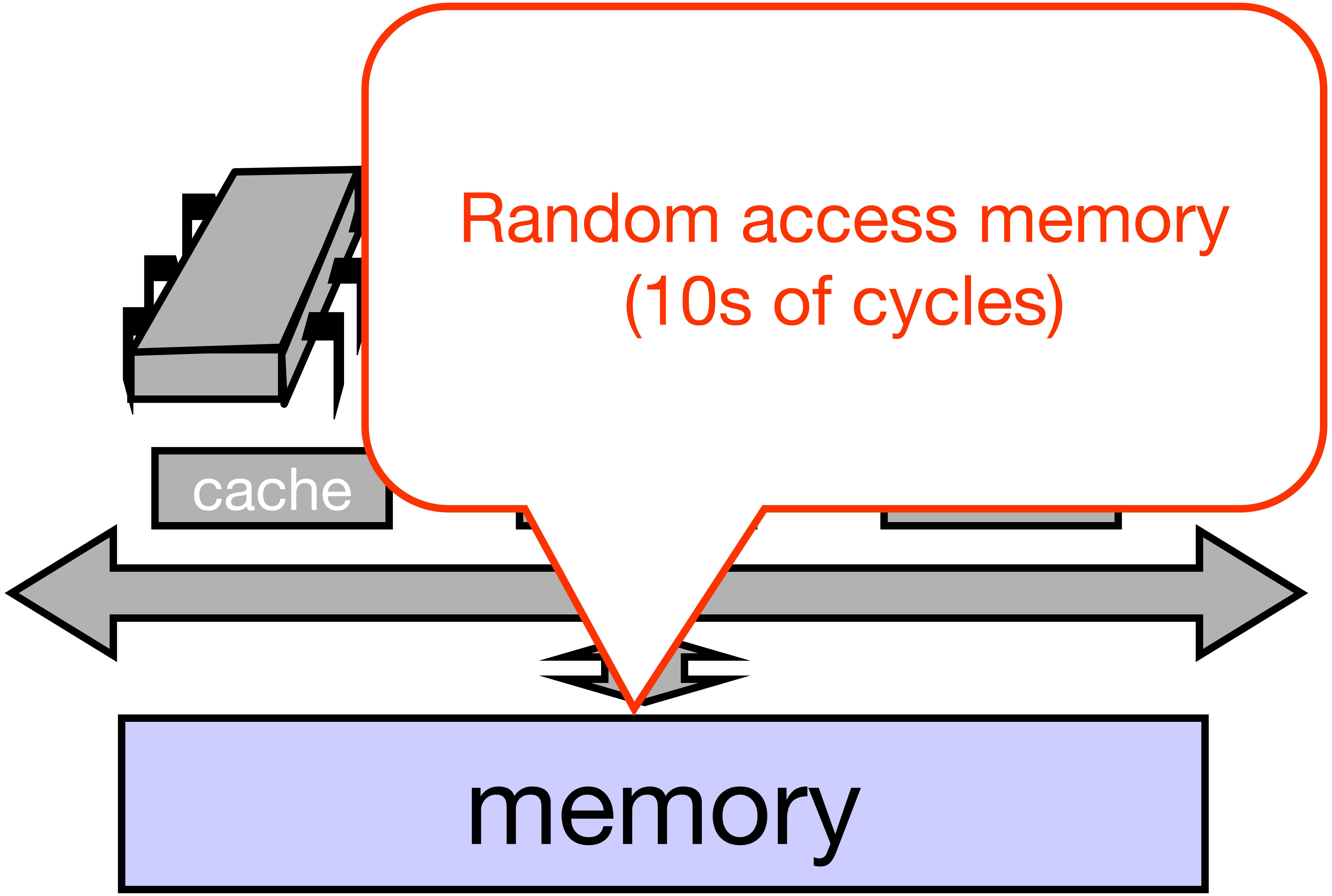
Opinion

- 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

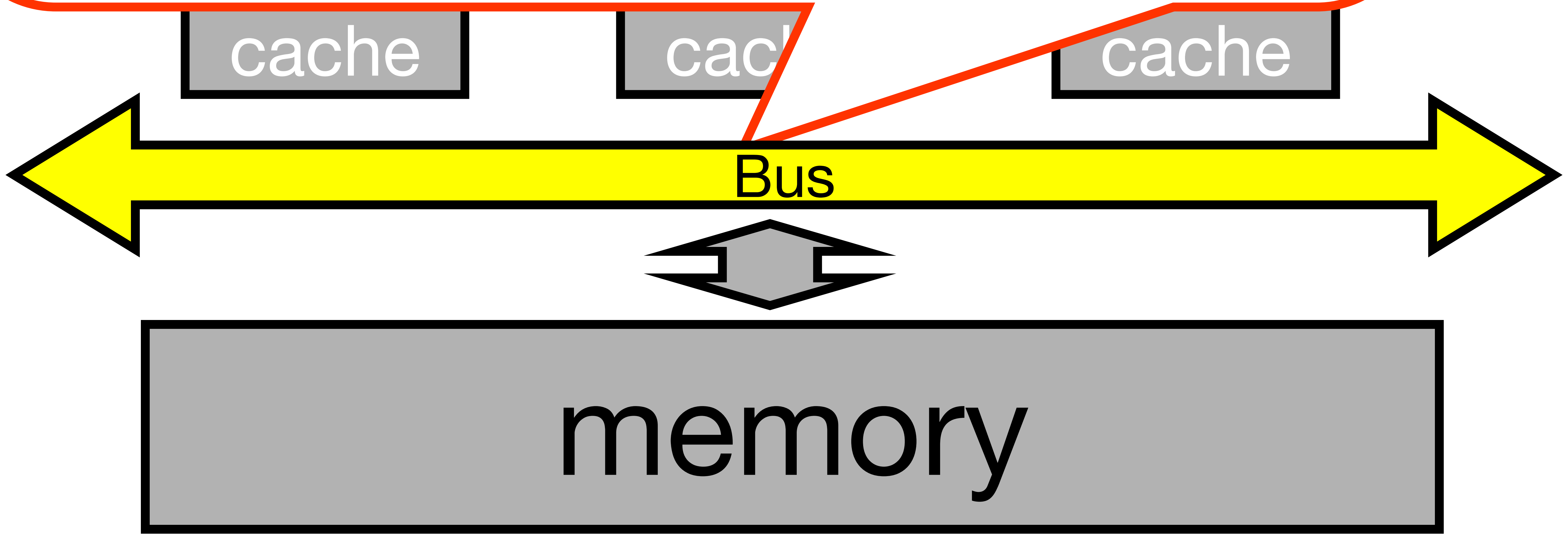


Random access memory
(10s of cycles)

Bus-based Architectures

Shared Bus

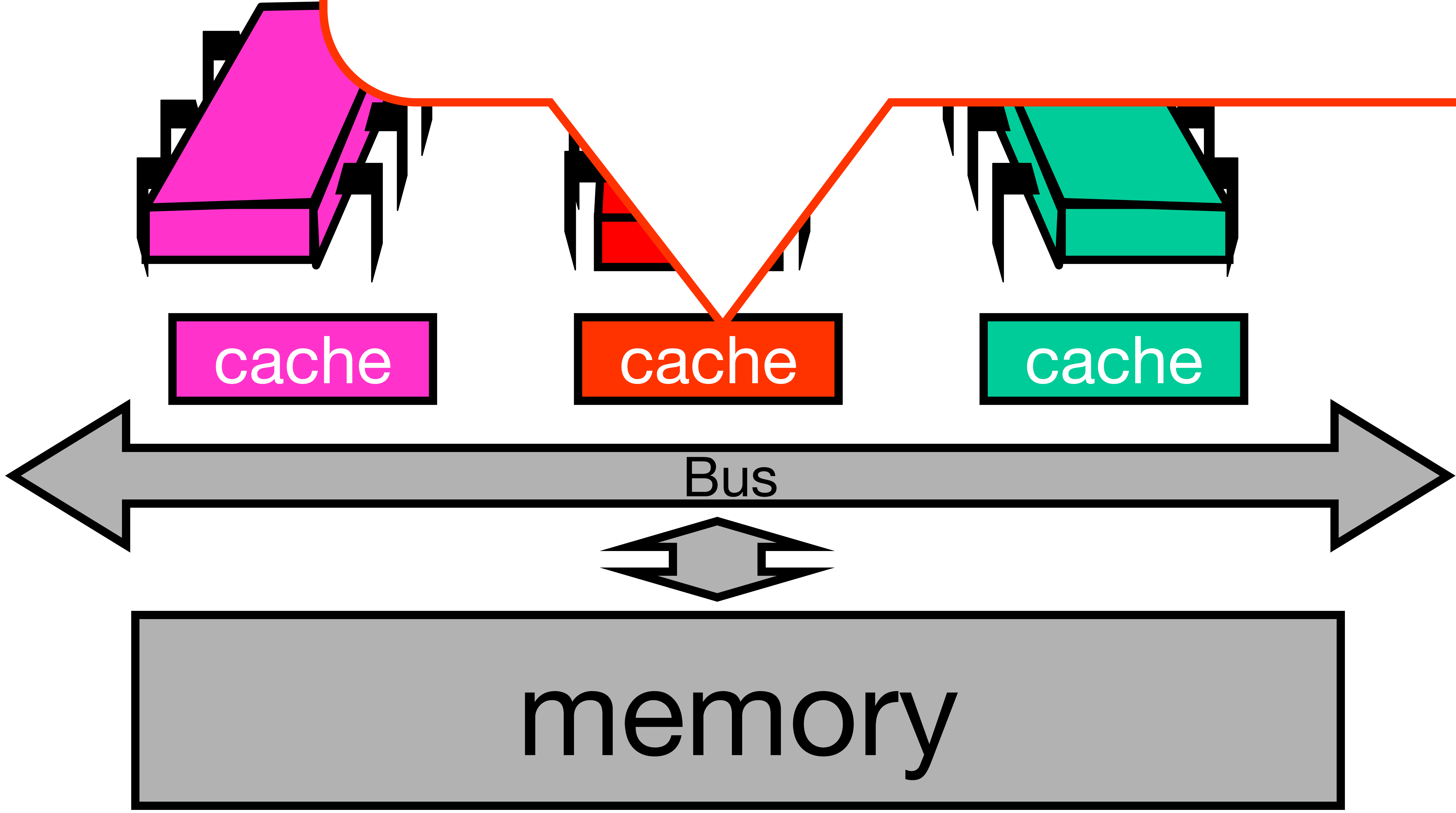
- Broadcast medium
- One broadcaster at a time
- Processors and memory all "snoop"



Bus-based Arc

Per-Processor Caches

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



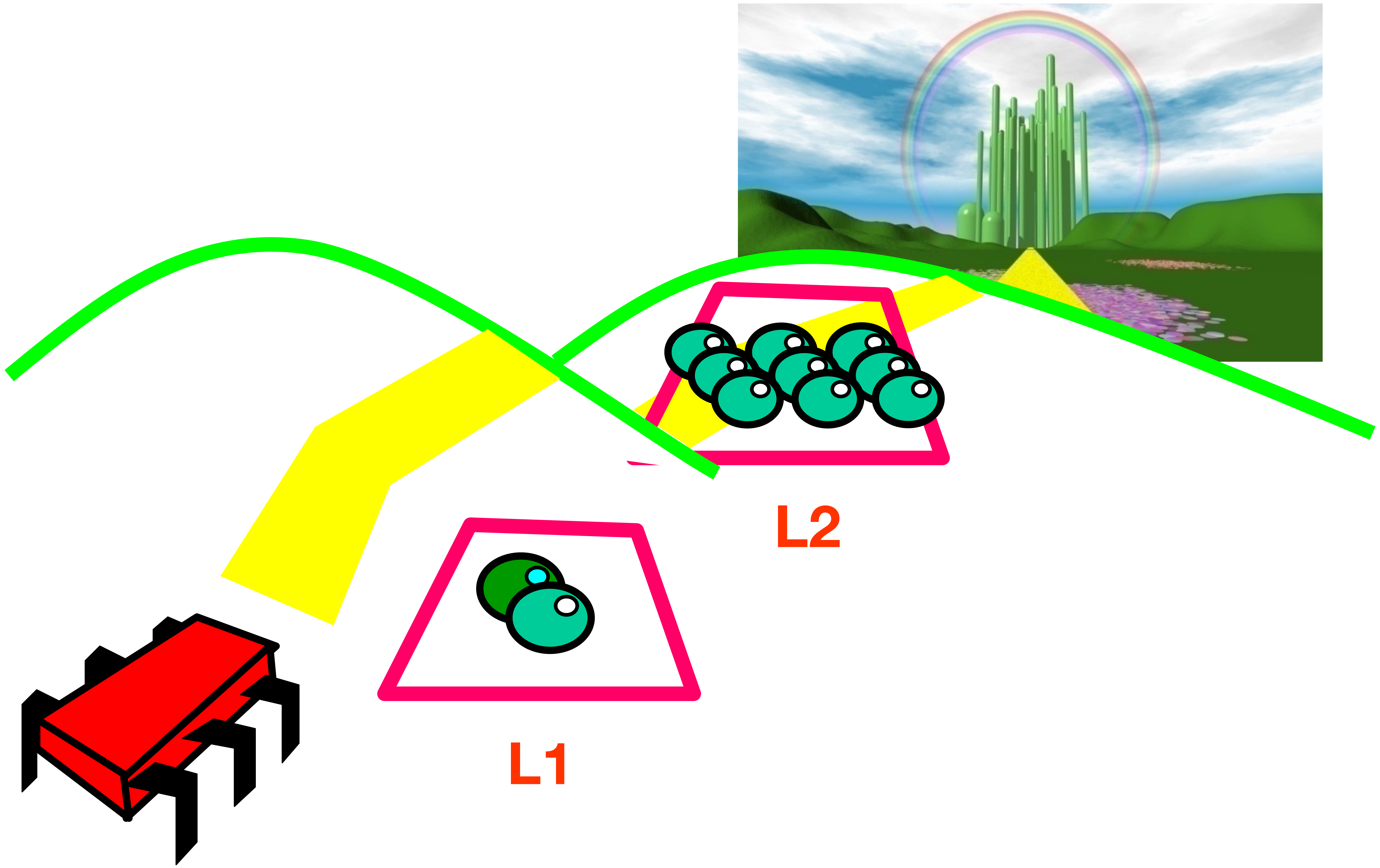
Granularity

- Caches operate at a larger granularity than a word
- **Cache line:** fixed-size block containing the address (today 64 or 128 bytes)

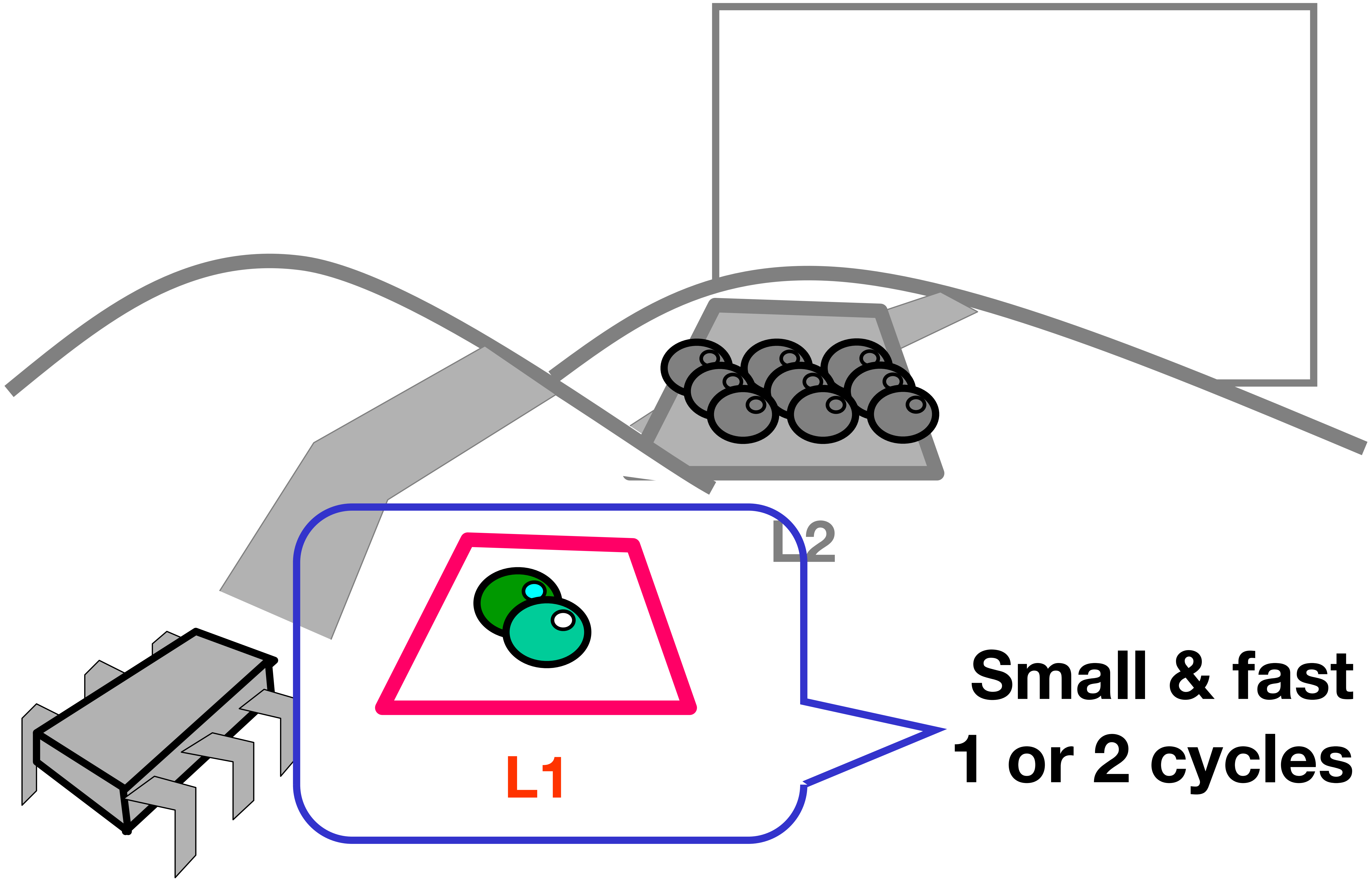
Locality

- 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



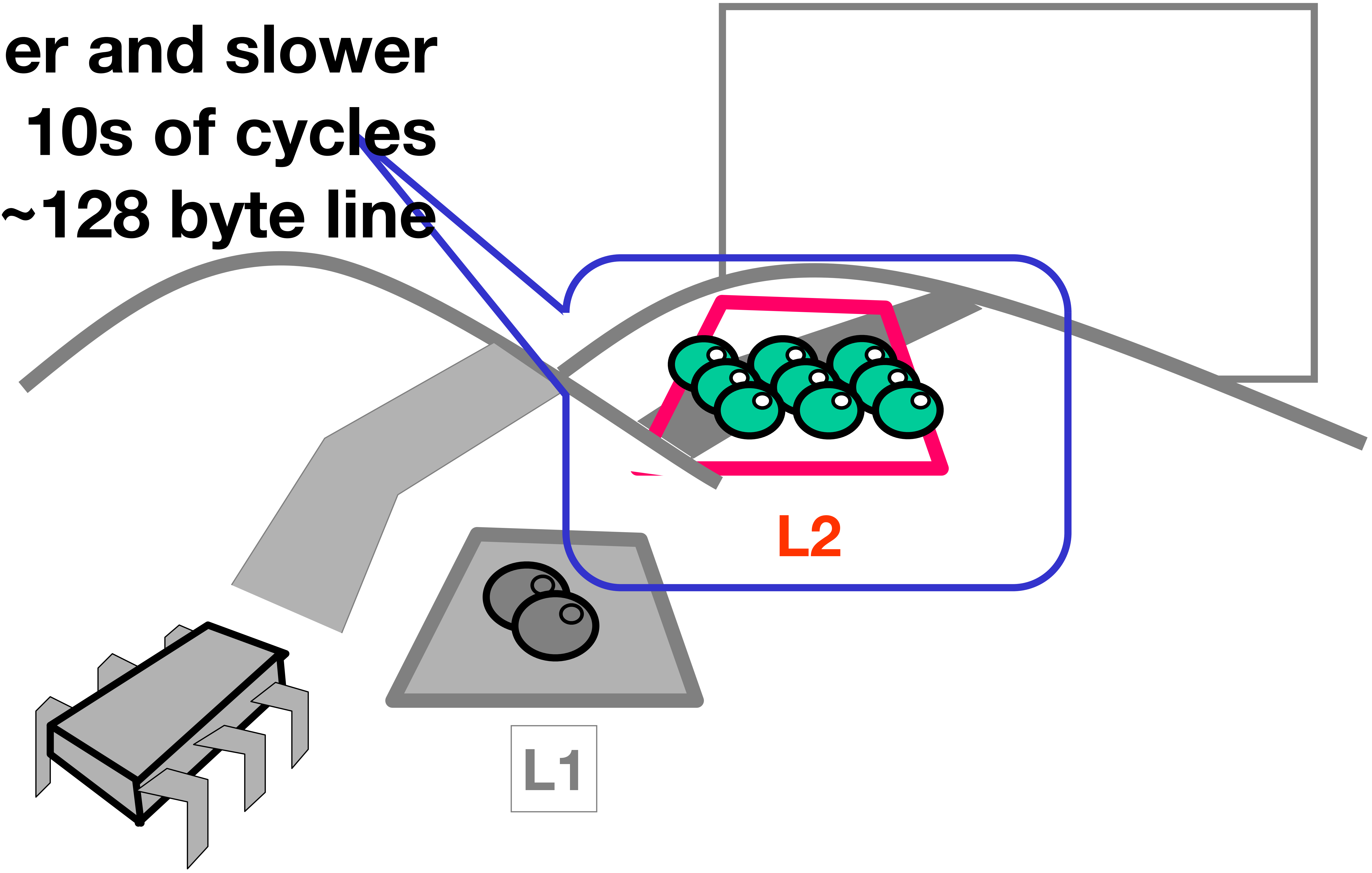
L1 and L2 caches



Small & fast
1 or 2 cycles

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™
- ***Cache miss***
  - “I had to shlep all the way to memory for that data”
  - Bad Thing™

Cave Canem

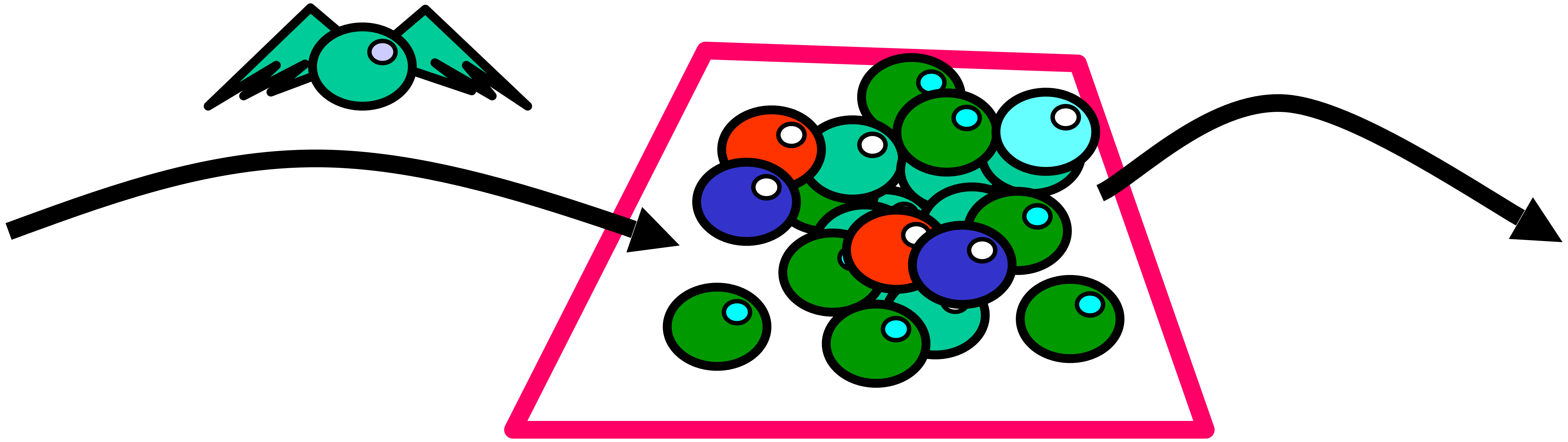
- This model is ***still*** a simplification
  - But not in any essential way
  - Illustrates basic principles
- Will discuss complexities later

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

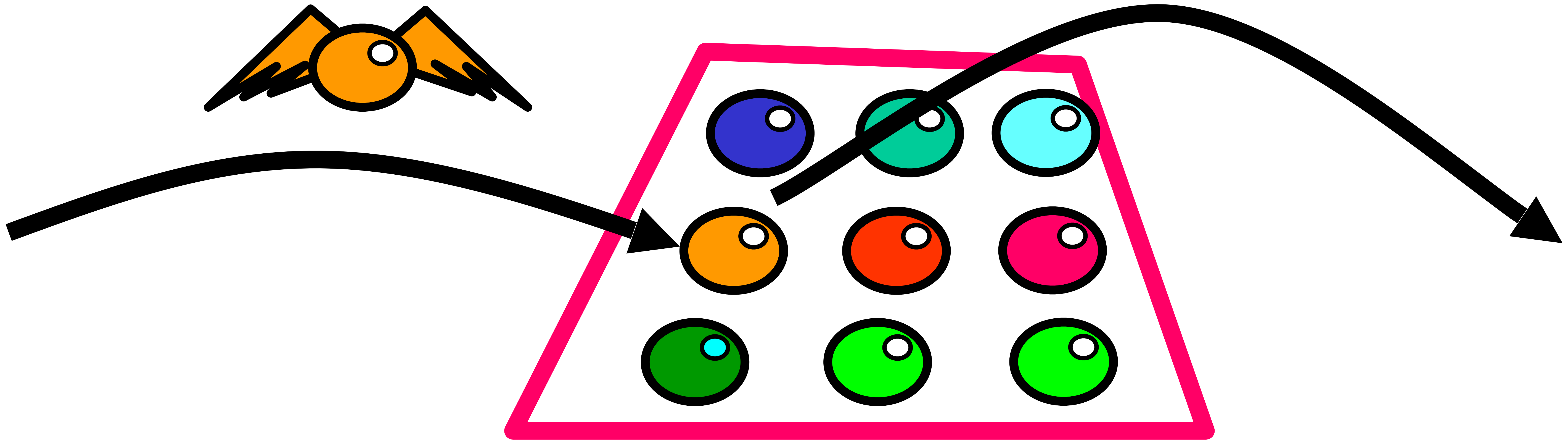
Fully Associative Cache

- Any line can be anywhere in the cache
  - Advantage: can replace any line
  - Disadvantage: hard to find lines



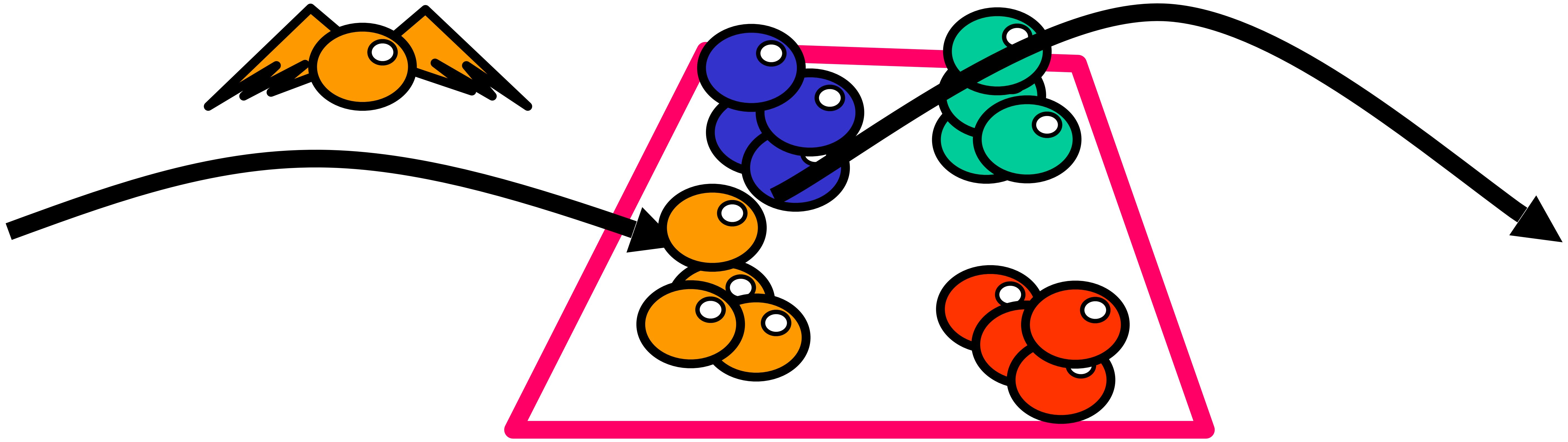
Direct Mapped Cache

- Every address has exactly 1 slot
  - Advantage: easy to find a line
  - Disadvantage: must replace fixed line



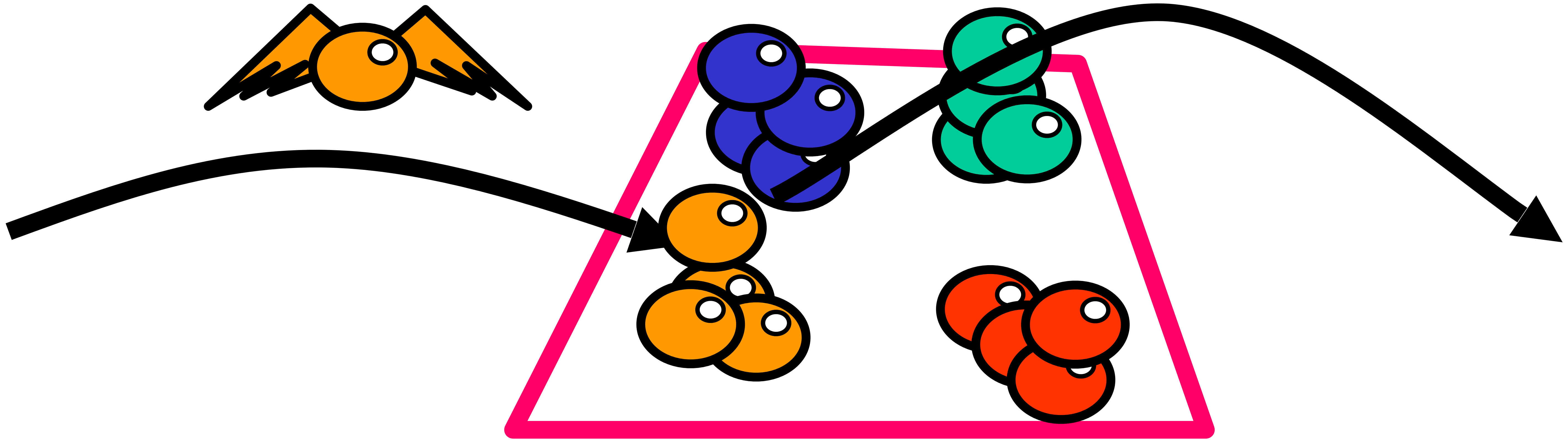
K-way set associative cache

- Each slot holds ***k*** lines
  - Advantage: pretty easy to find a line
  - Advantage: some choice in replacing line



Multicore Set Associativity

- ***k*** is ***8*** or even ***16*** and growing…
  - Why? Because cores share sets
  - Threads cut effective size if accessing different data



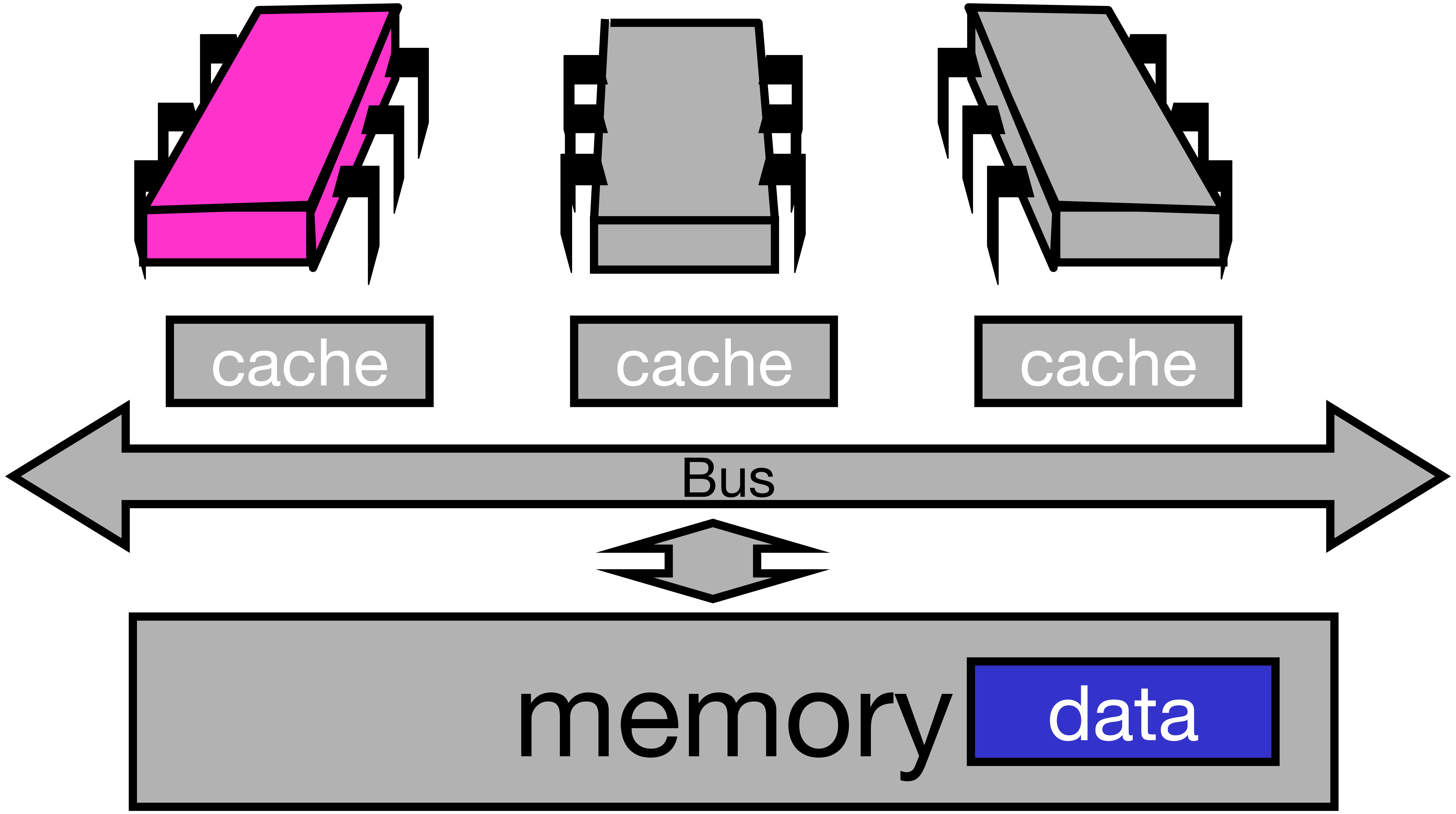
Cache Coherence

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

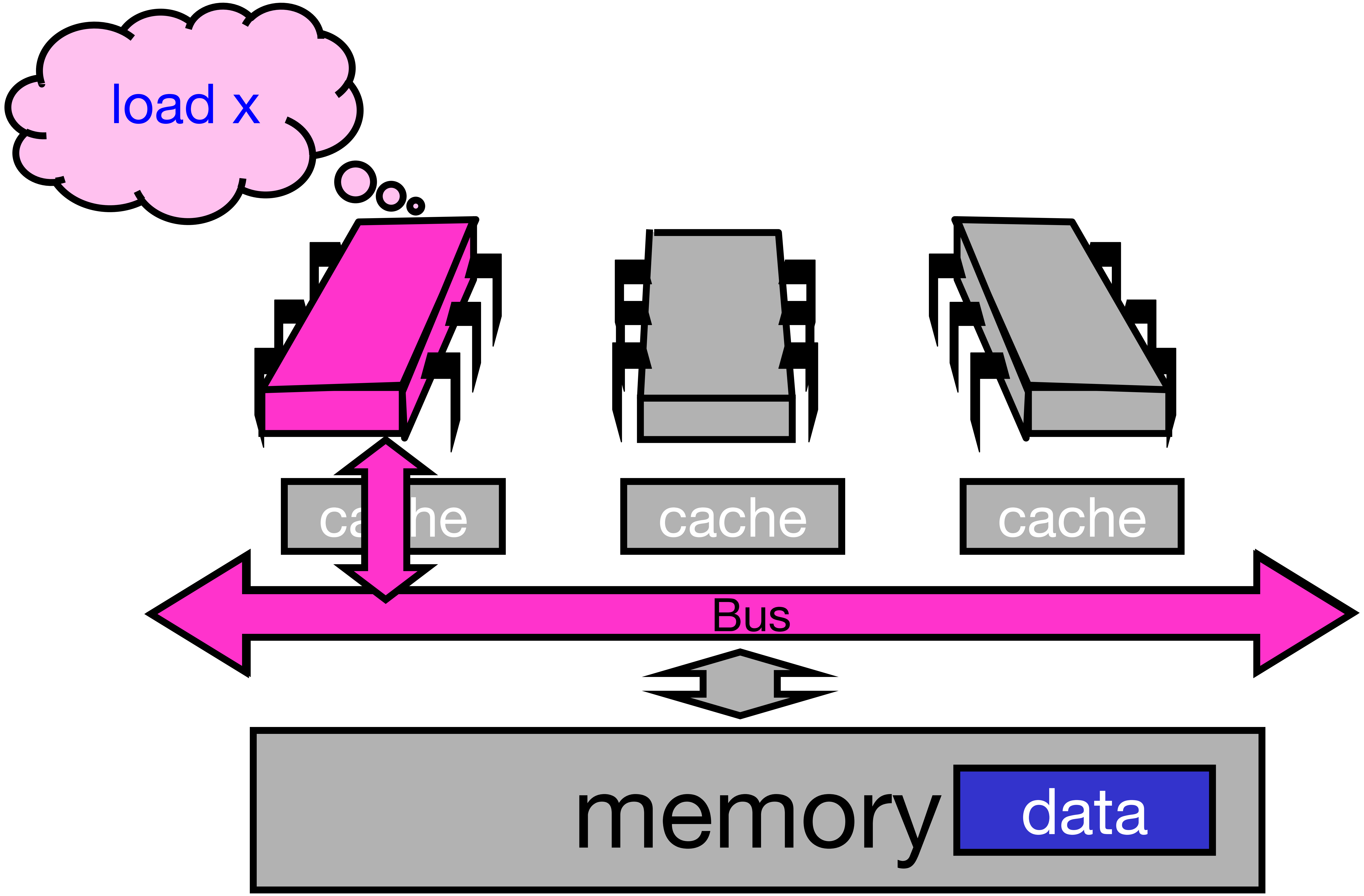
MESI

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

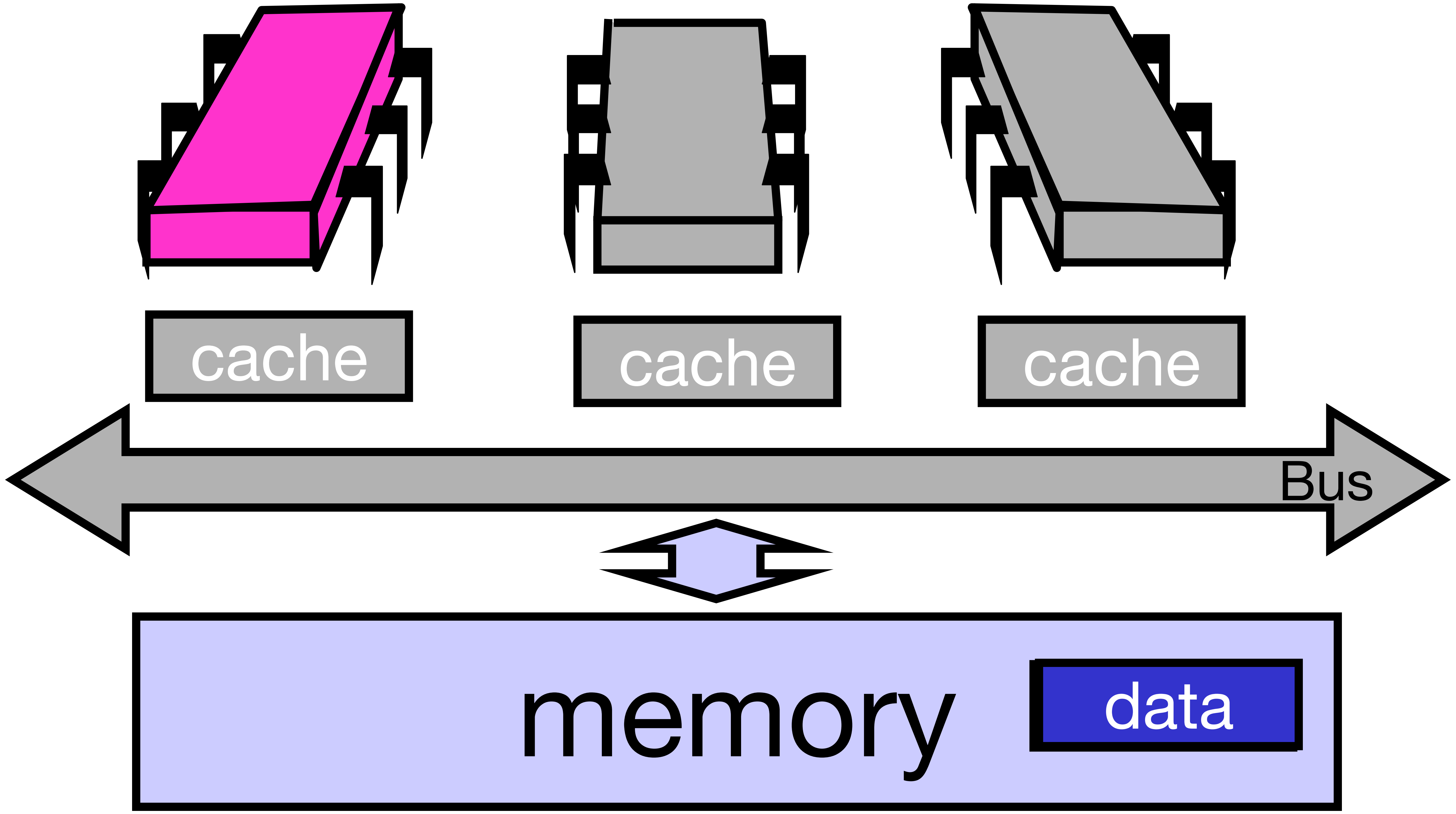
Processor Issues Load Request



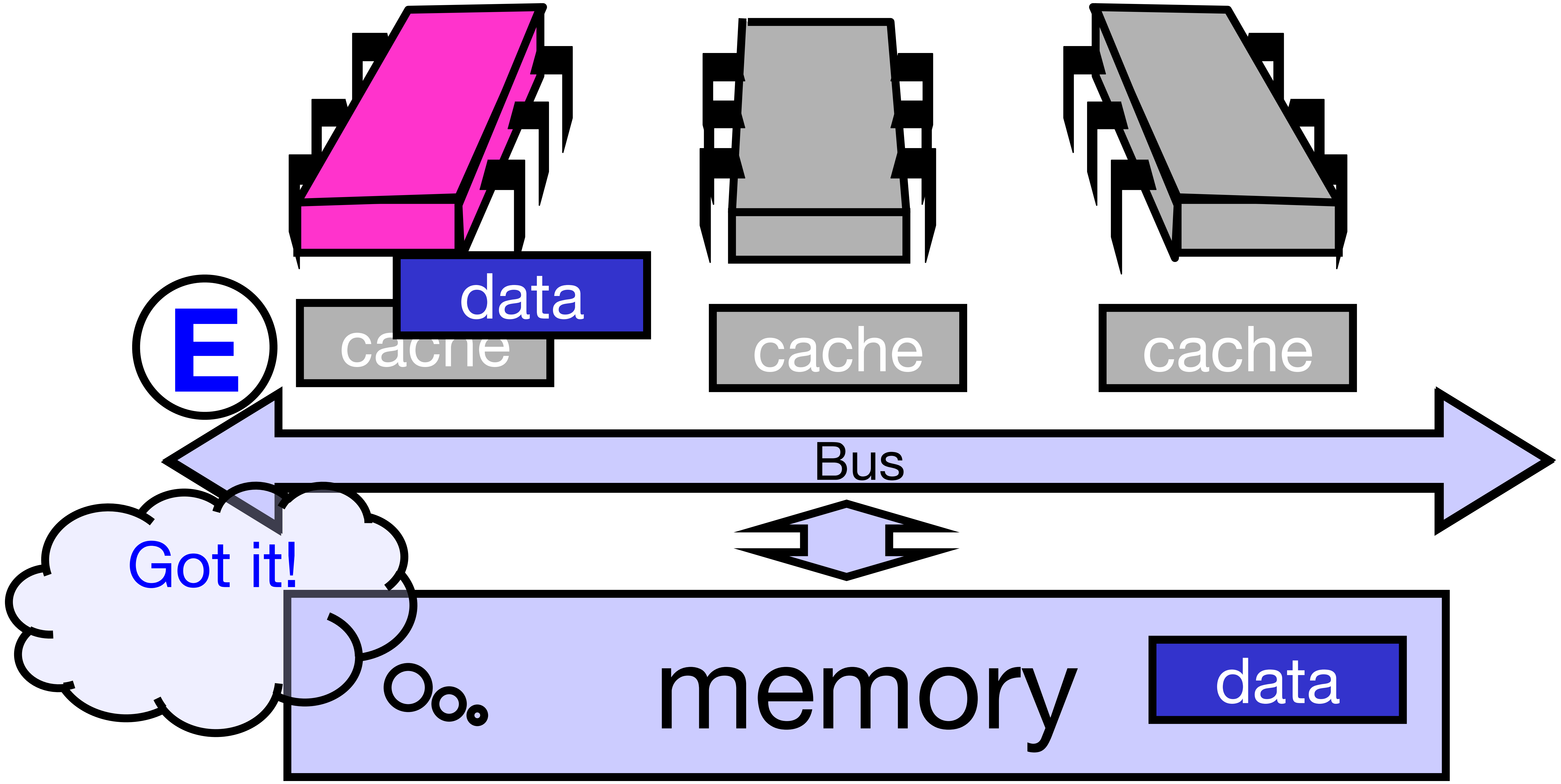
Processor Issues Load Request



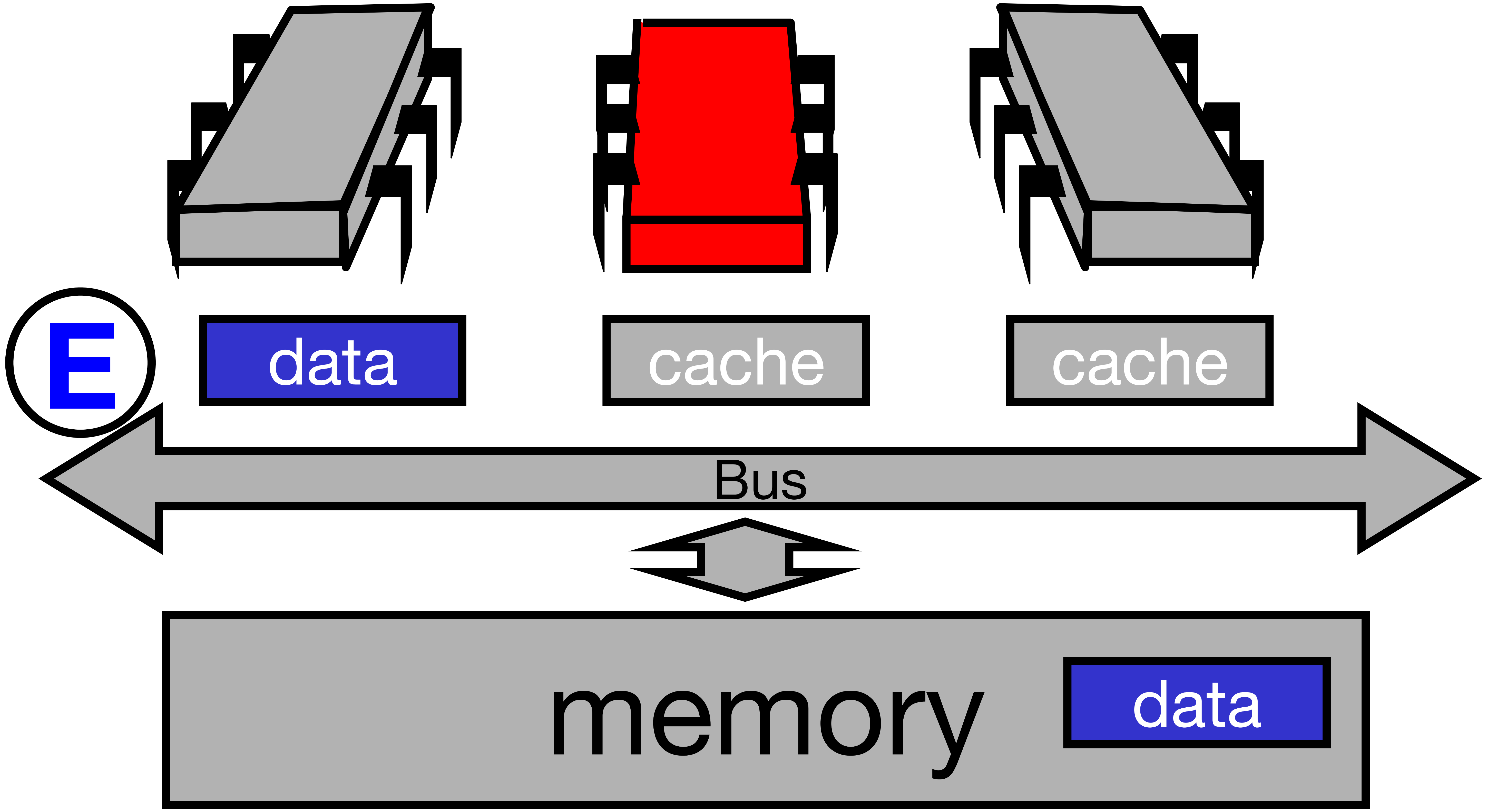
Memory Responds



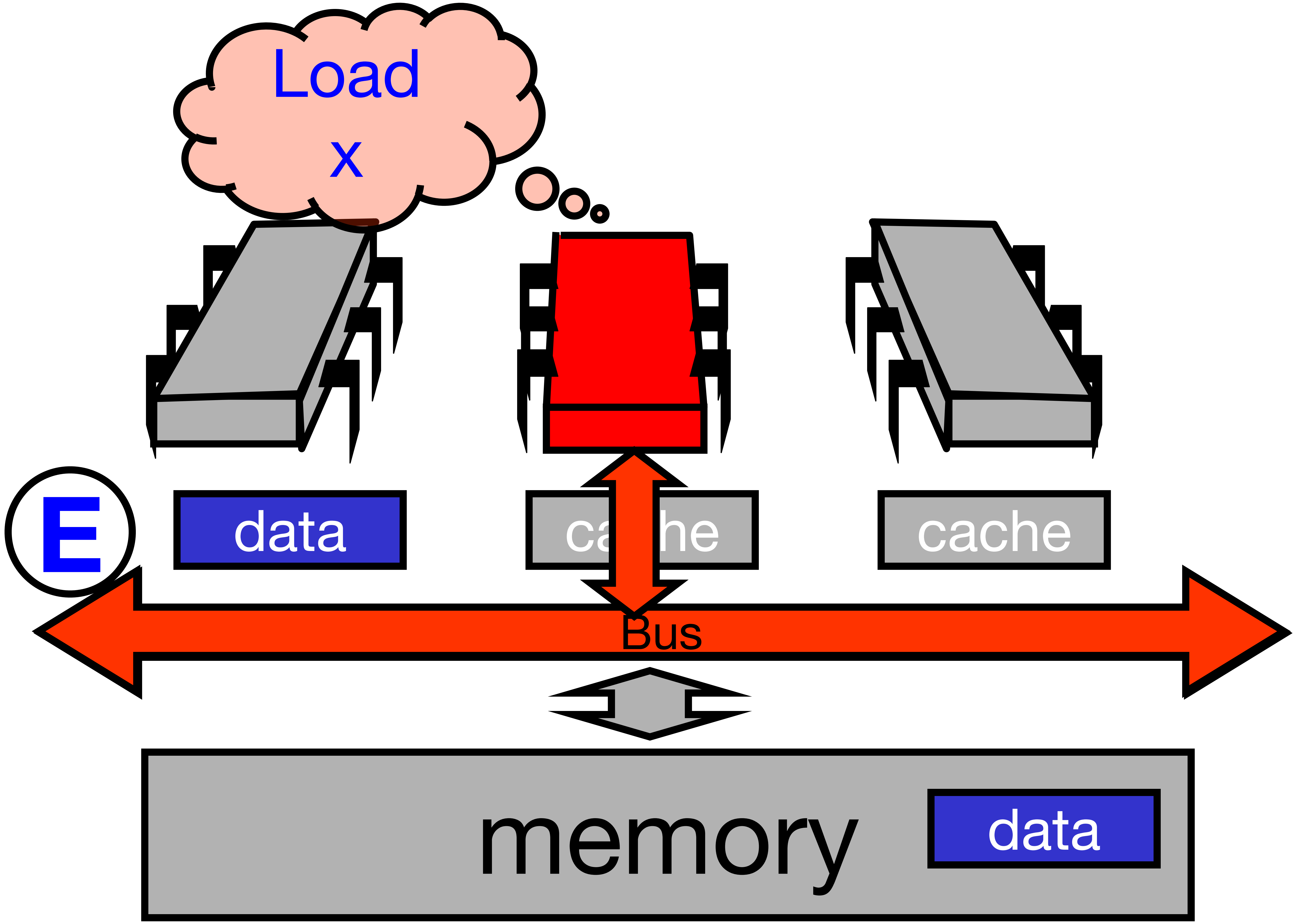
Memory Responds



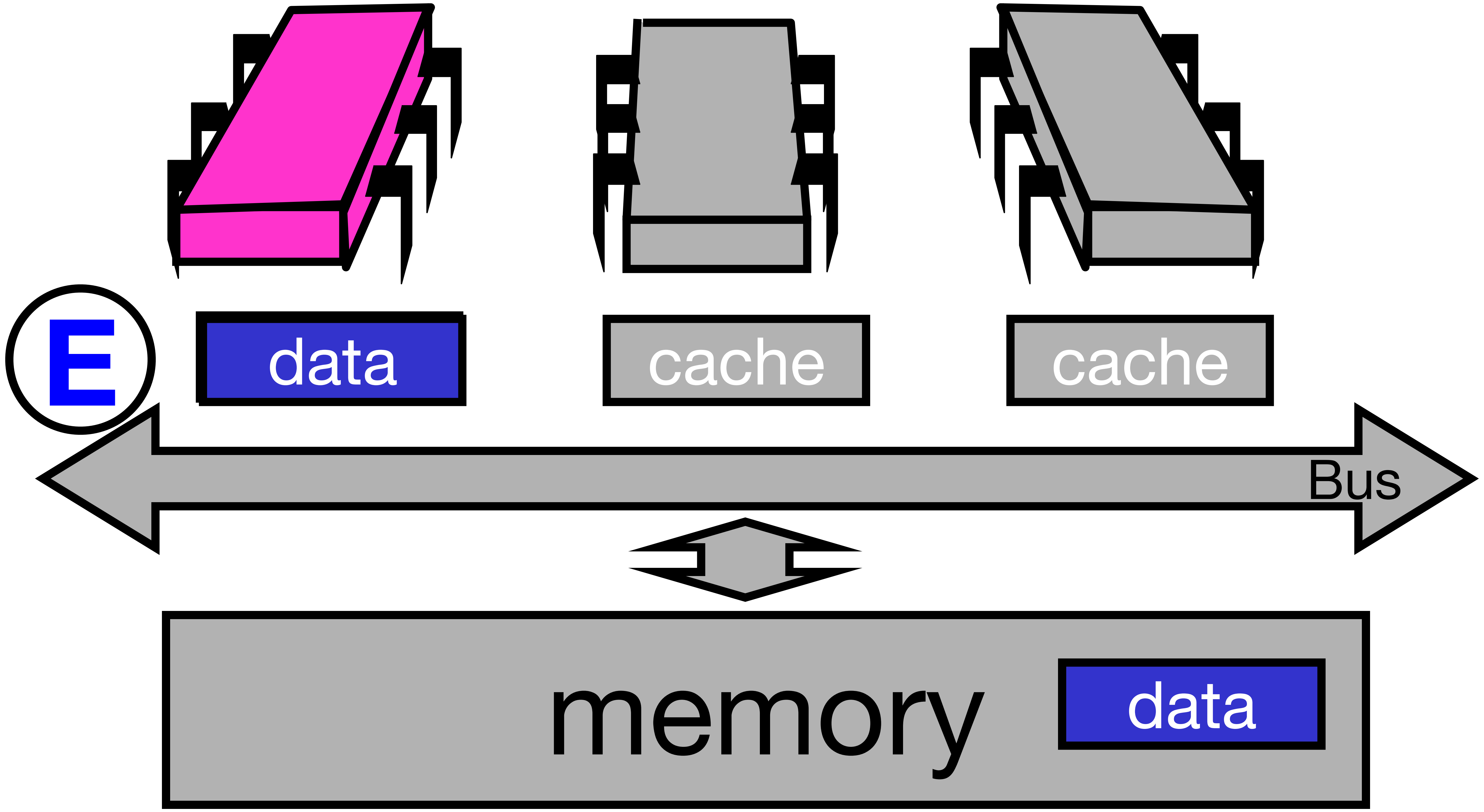
Processor Issues Load Request



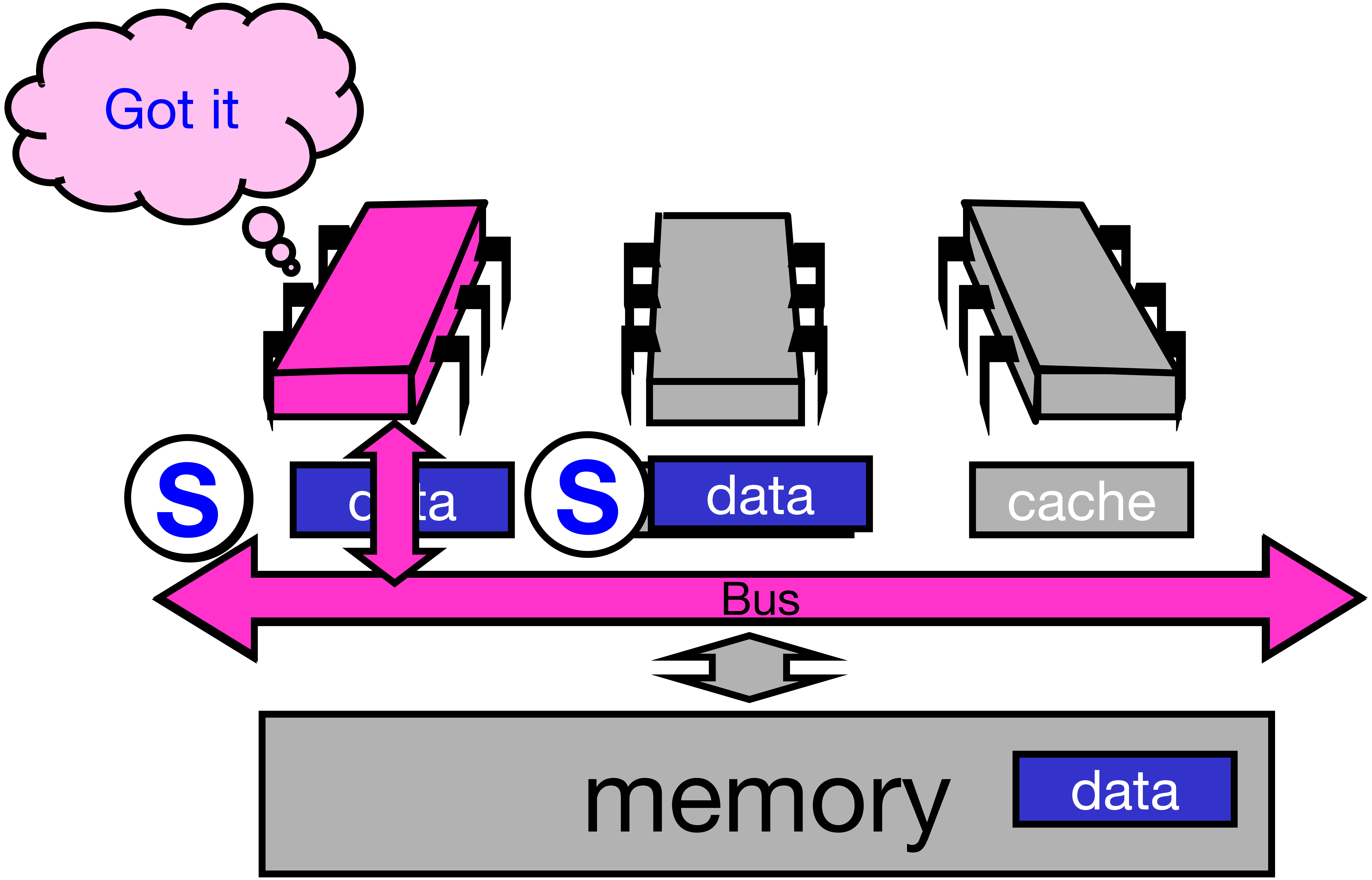
Processor Issues Load Request



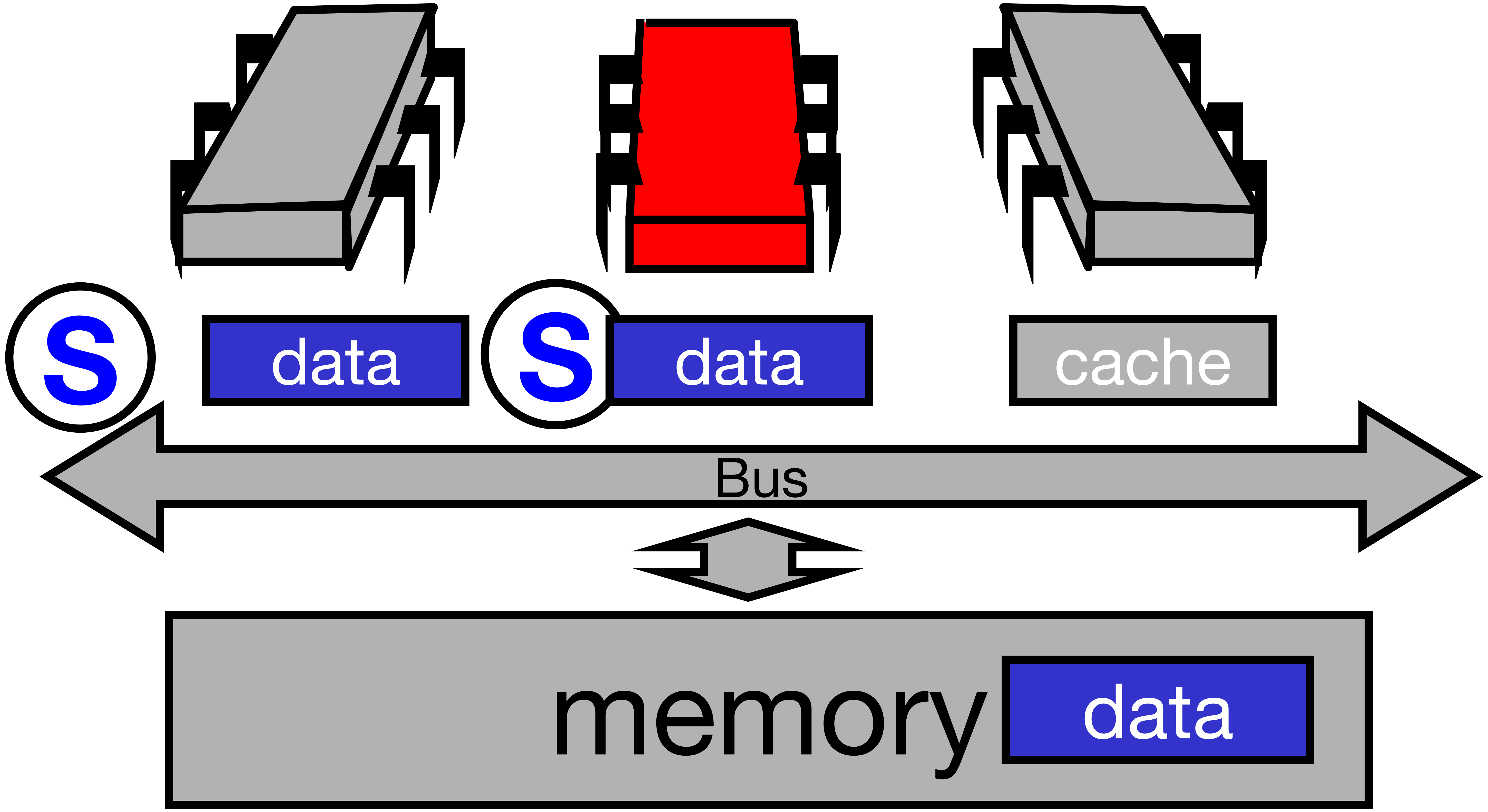
Other Processor Responds



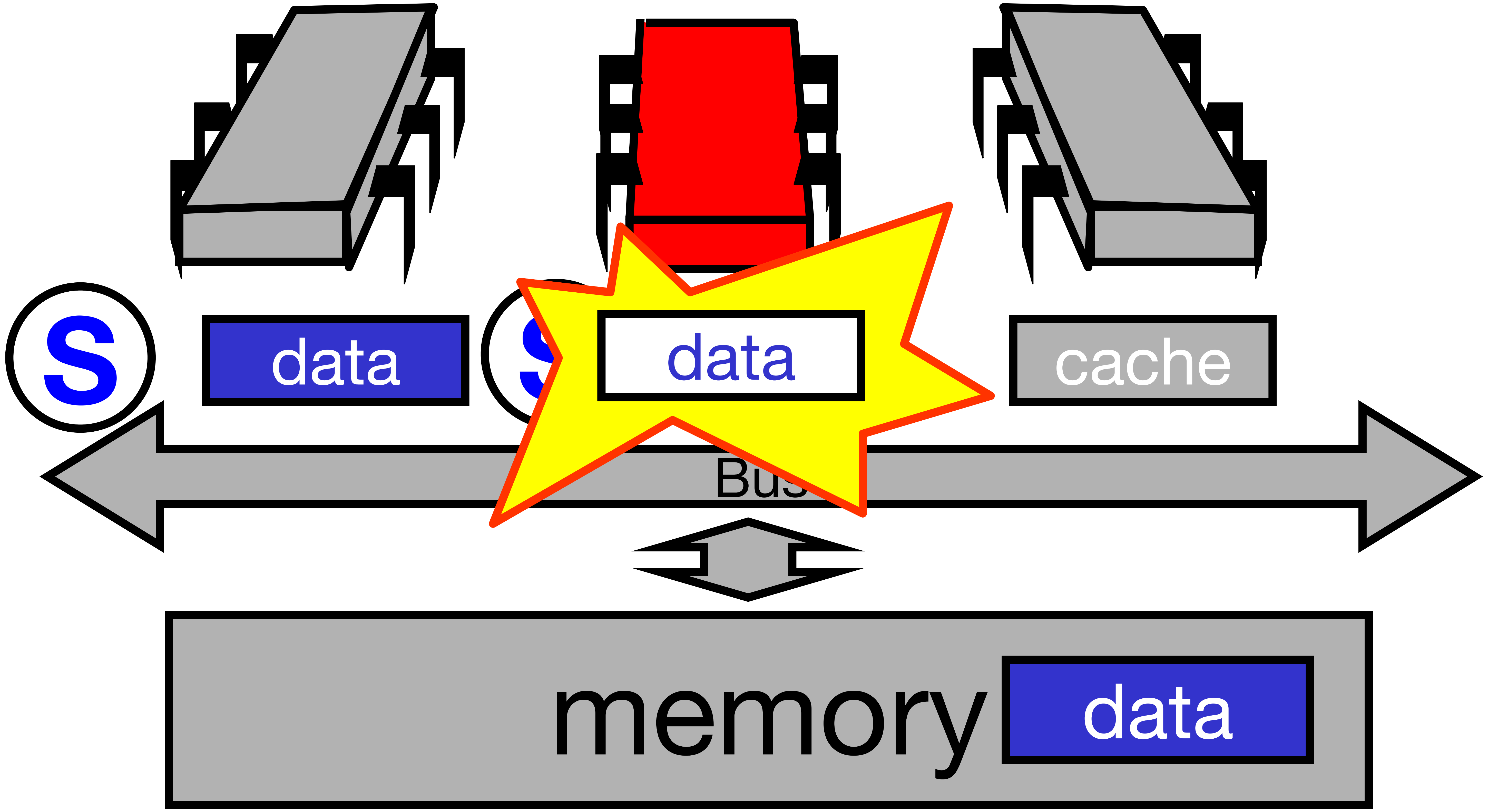
Other Processor Responds



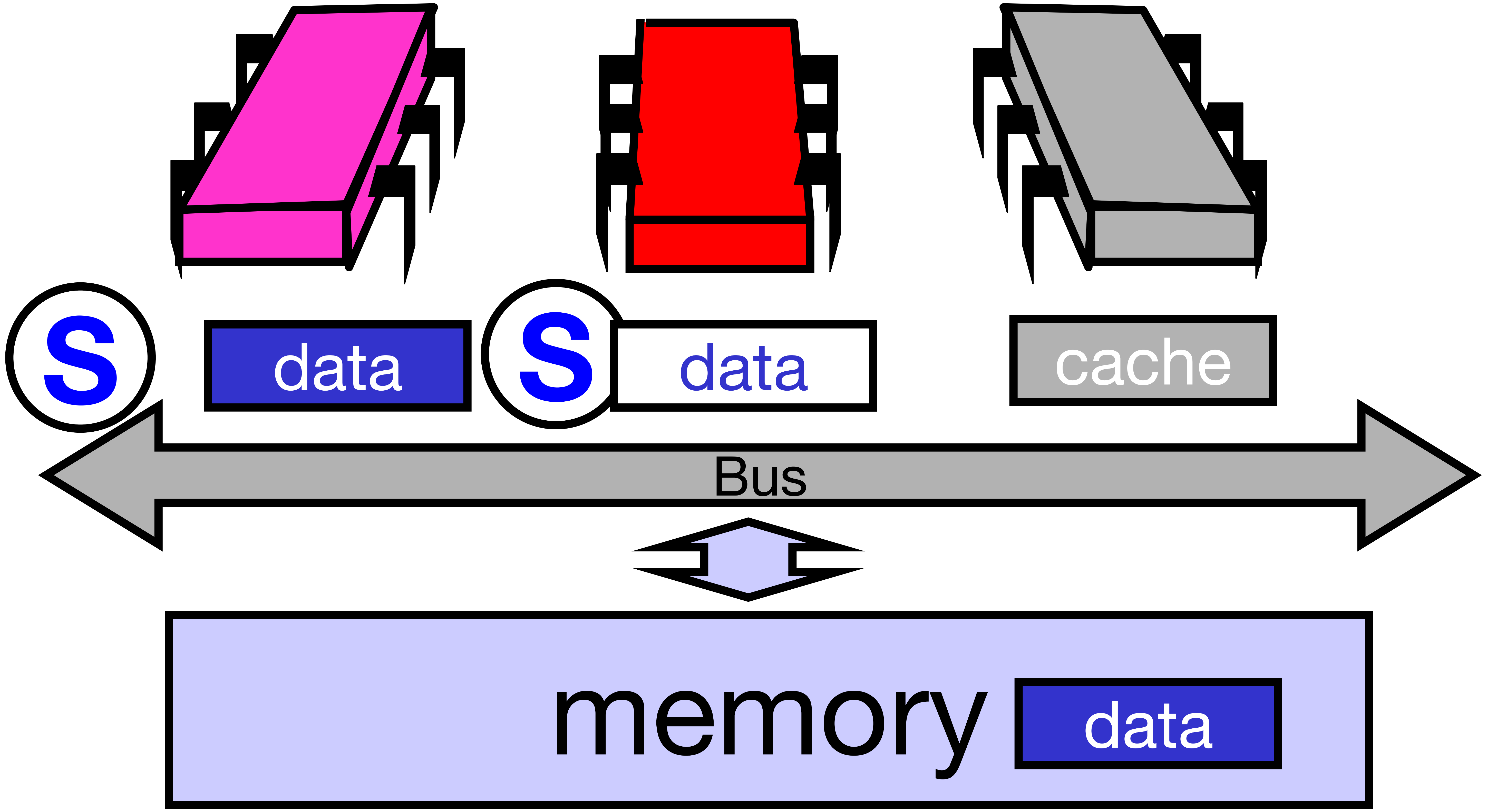
Processor Issues Load Request



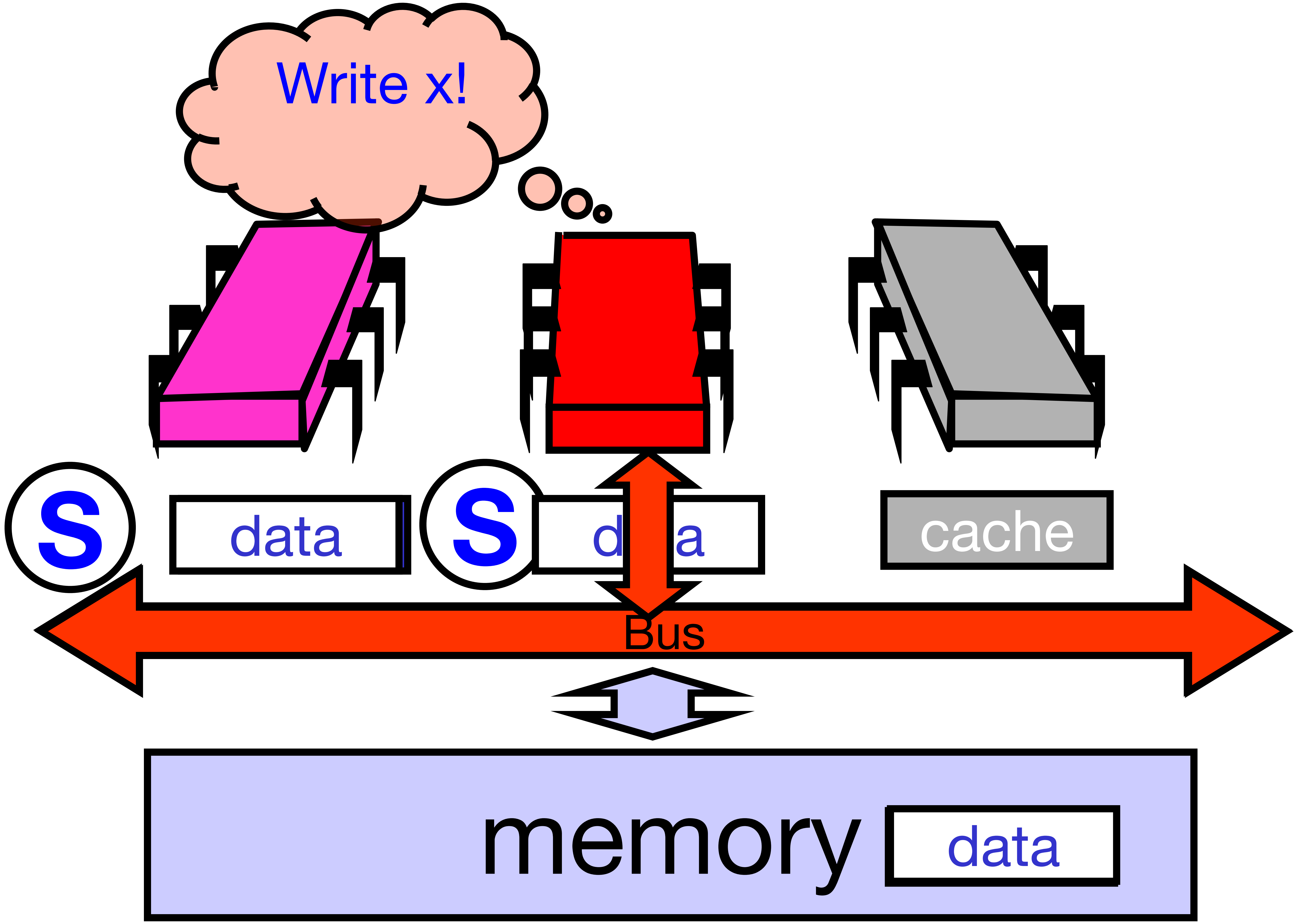
Processor Issues Load Request



Write-through Cache



Write-through Cache



Write-through Cache

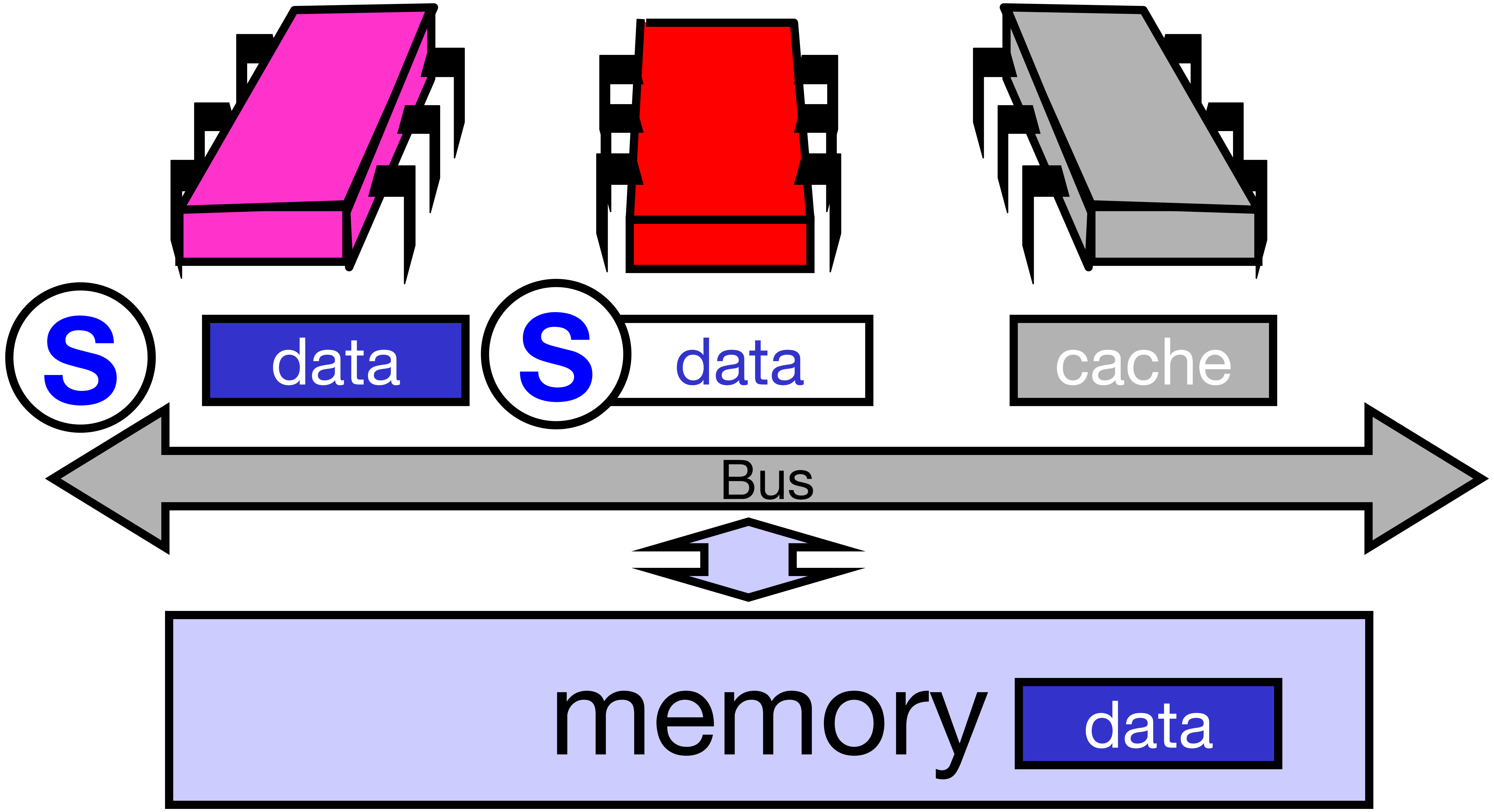
- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes …

Write-through Cache

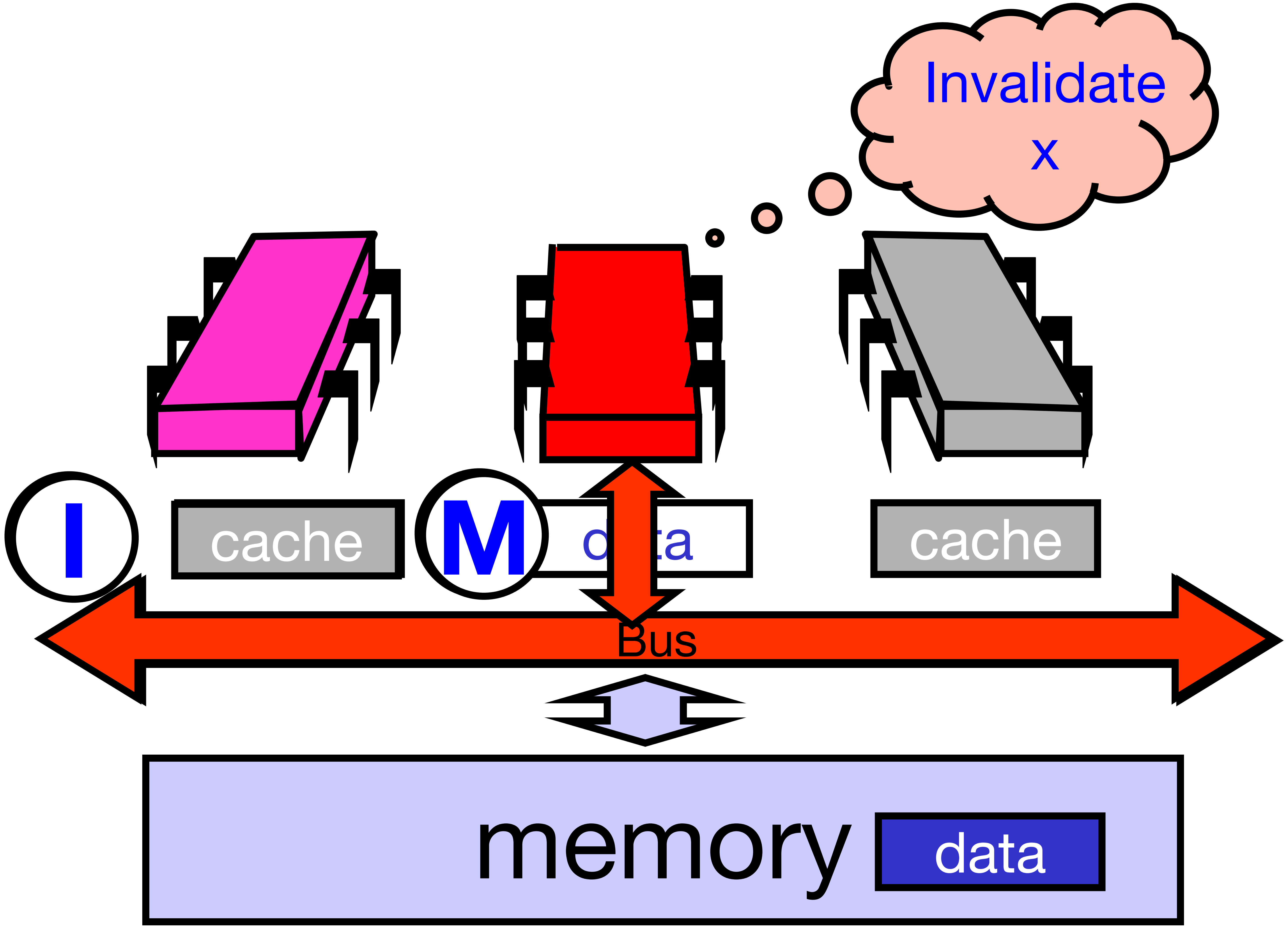
- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
  - For example, loop indexes …

“show stoppers”

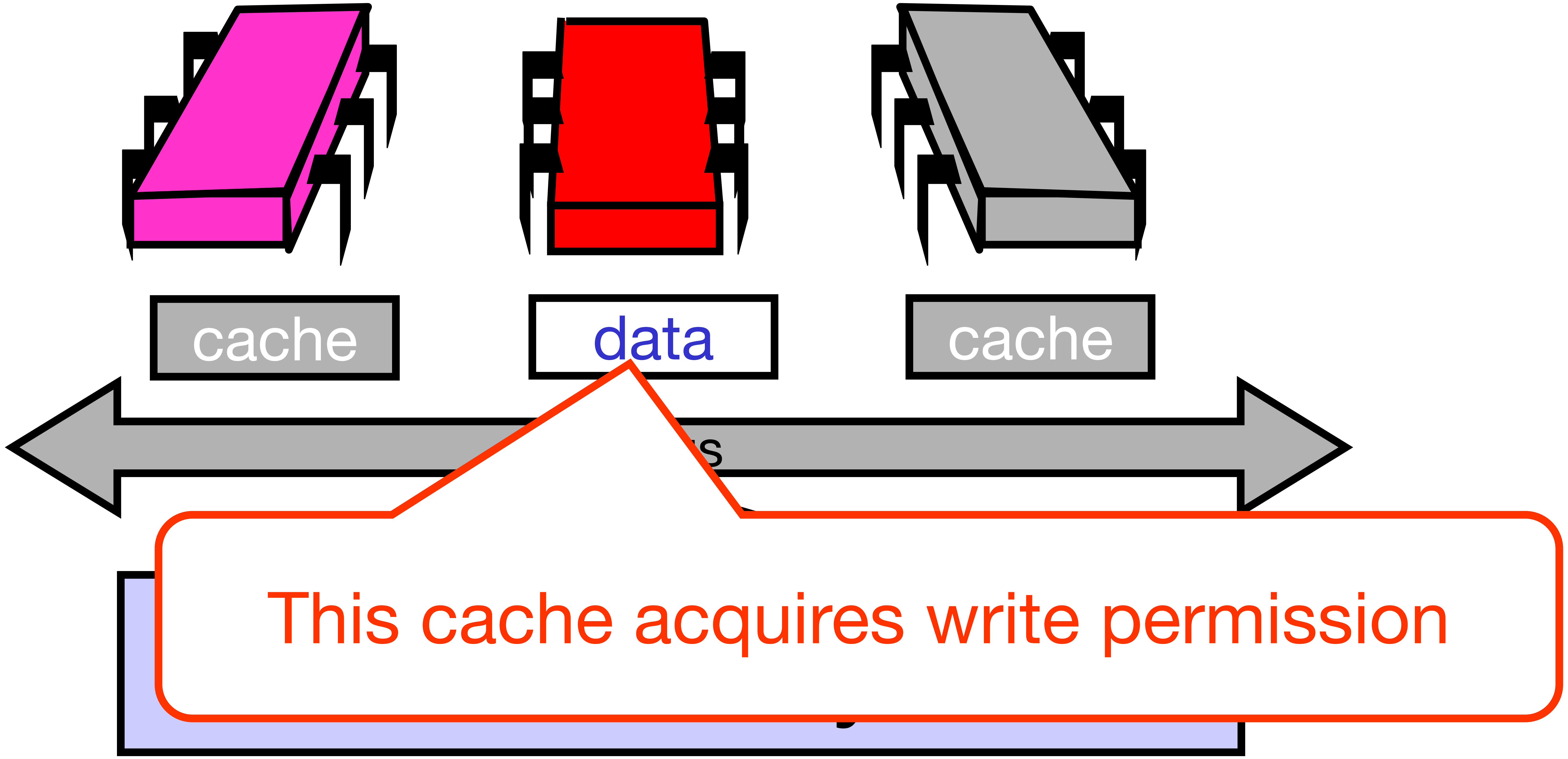
Invalidate



Invalidate

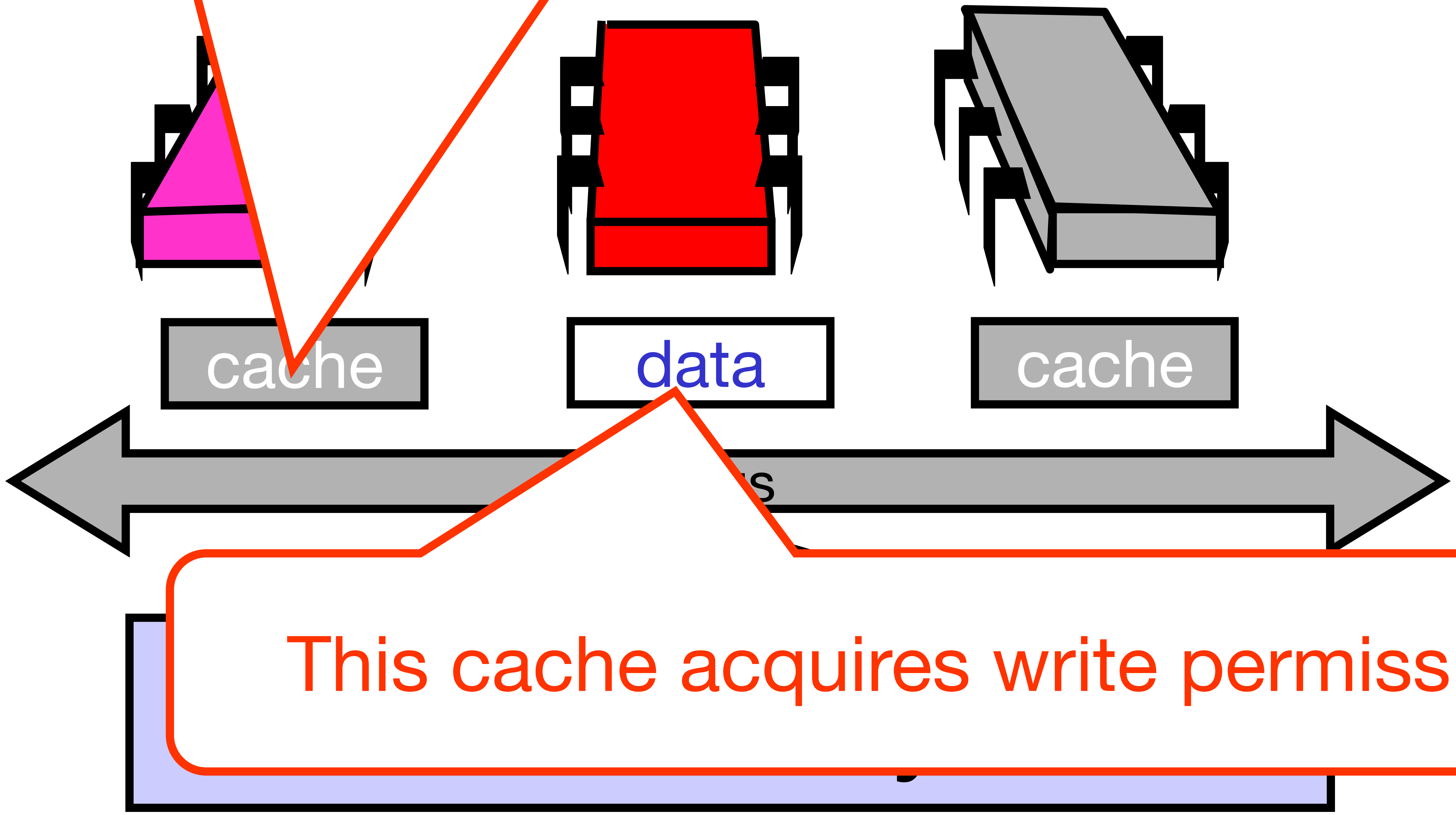


Invalidate



This cache acquires write permission

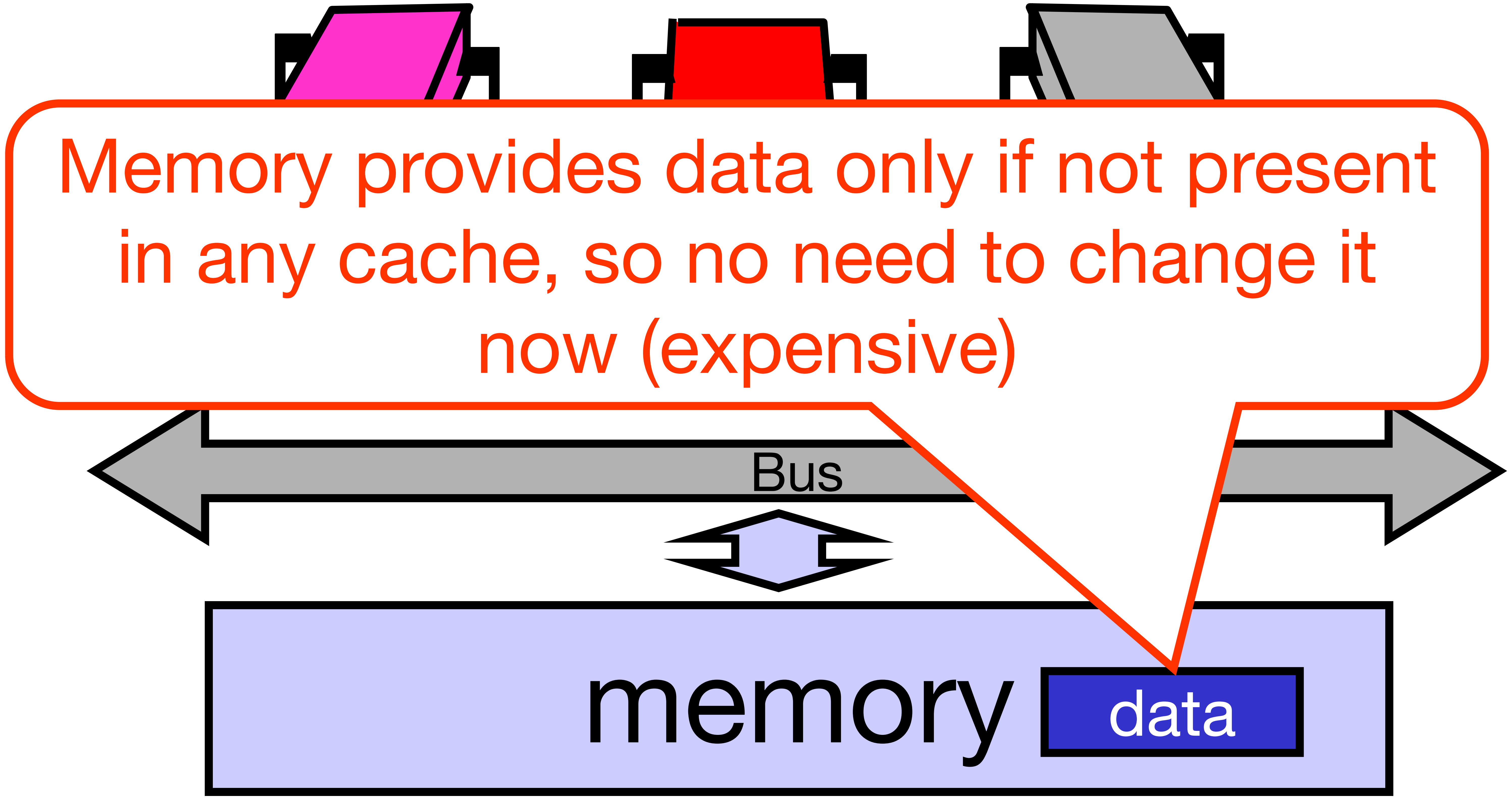
Invalidate



Other caches lose read permission

This cache acquires write permission

Invalidate



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

Mutual Exclusion

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

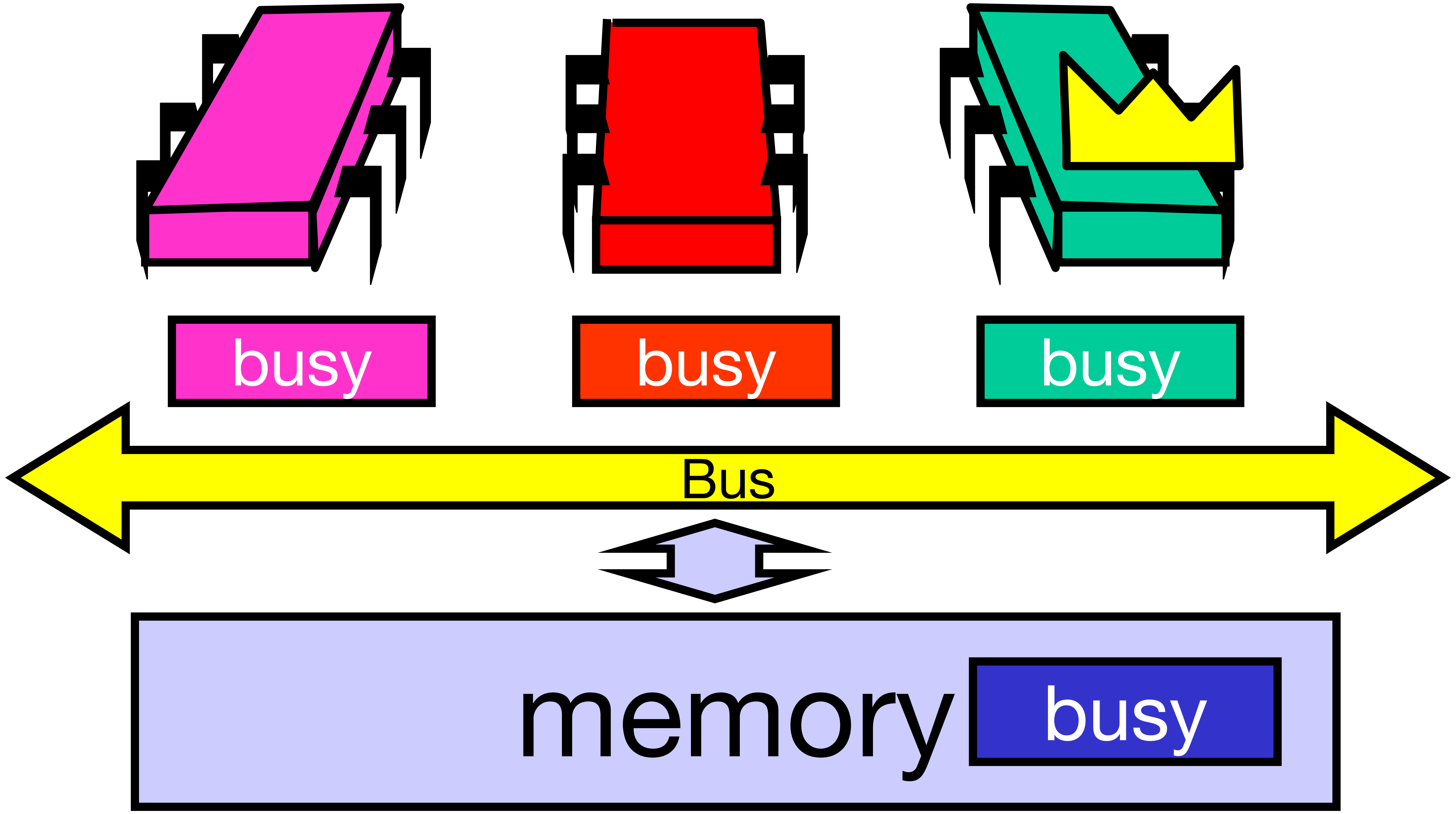
Test-and-set Lock (TASLock)

- TAS *invalidates* (I) cache lines
- Spinners
  - Miss in cache
  - Go to bus
- Cache line ***bounces*** between all the spinners in modified (M) state
- Thread wants to release lock
  - delayed behind spinners

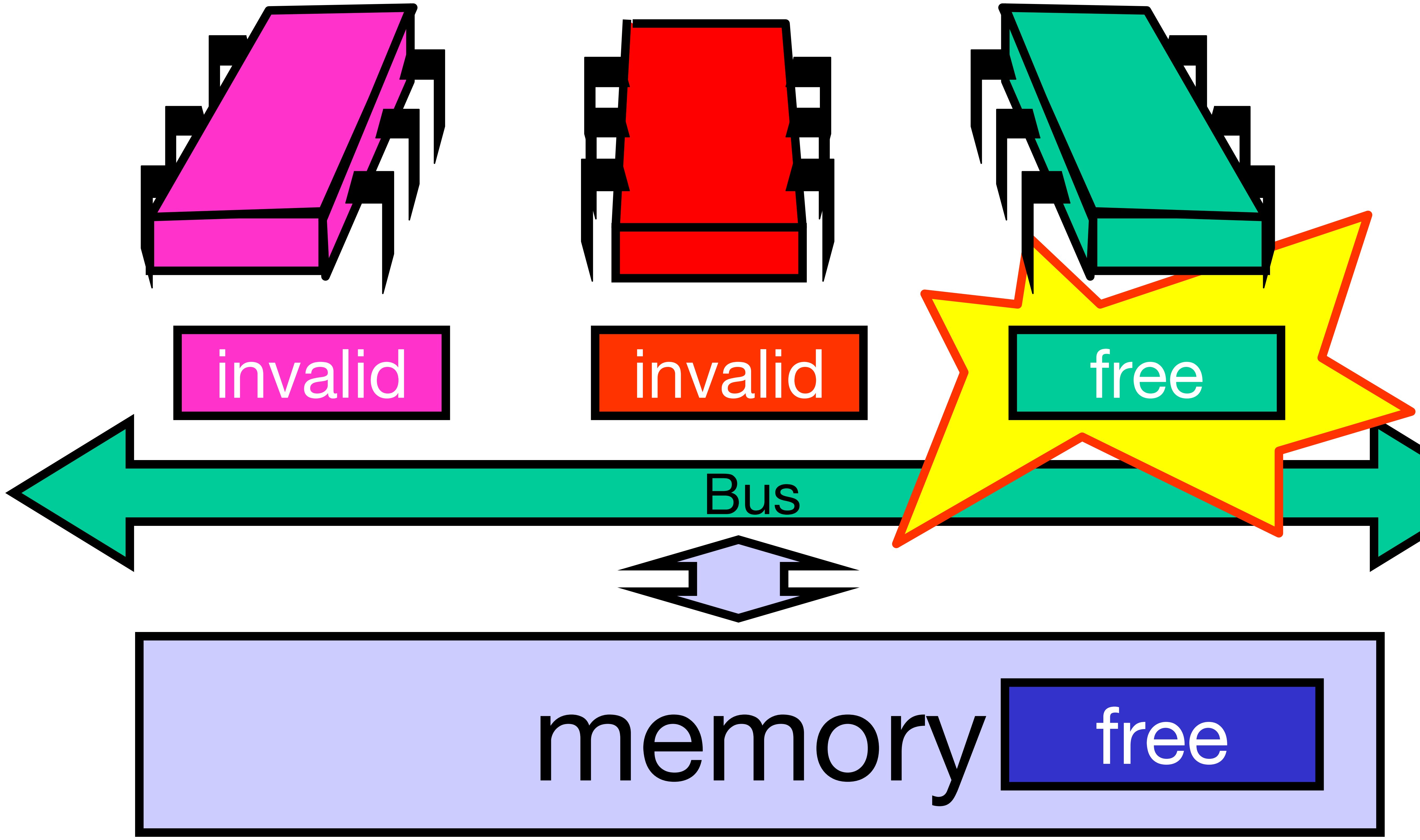
Test-and-test-and-set Lock (TTASLock)

- Wait until lock “looks” free
  - Spin on local cache
  - Cache line is in Shared (S) state
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm …

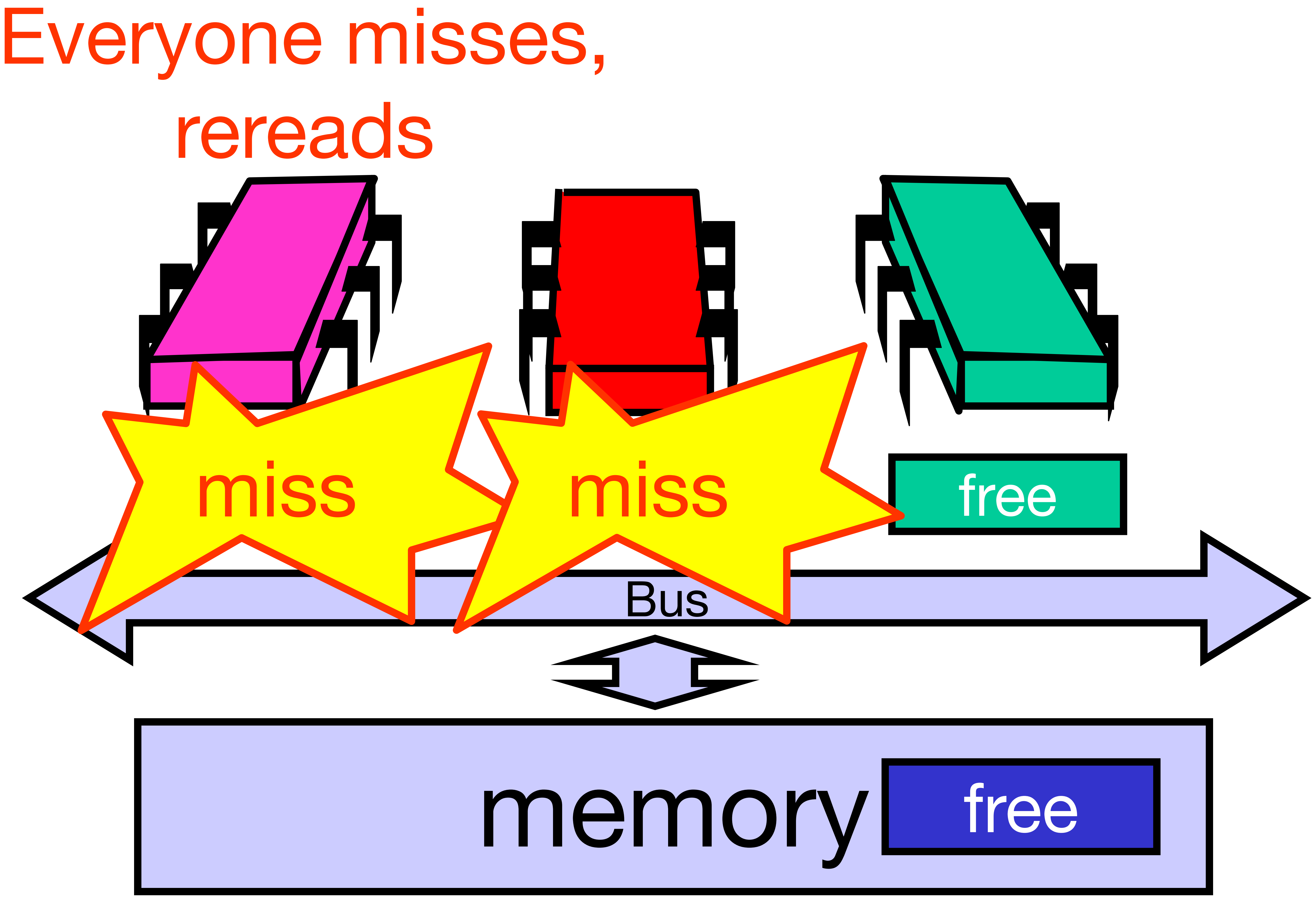
Local spinning while lock is busy



On Release

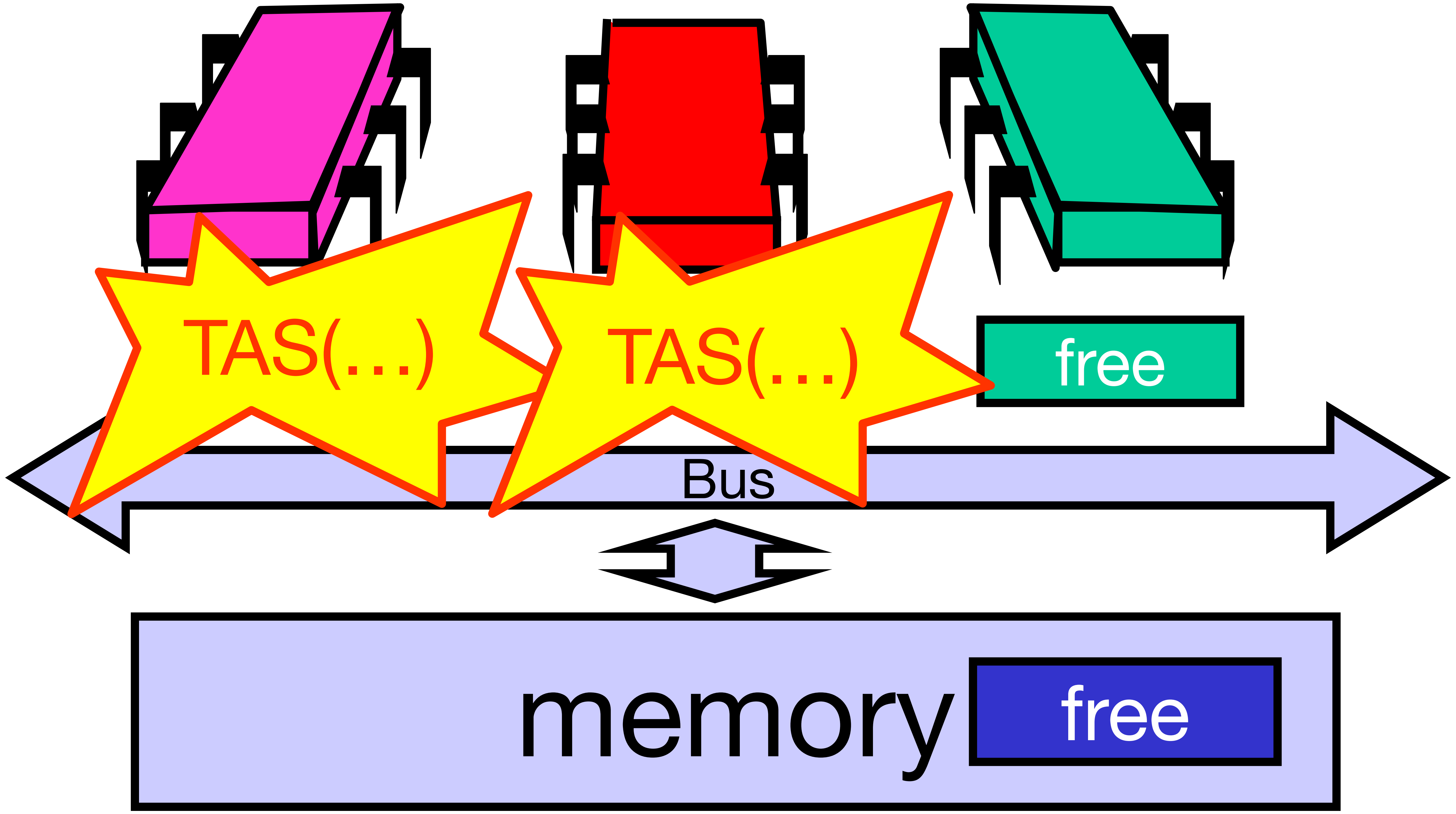


On Release



On Release

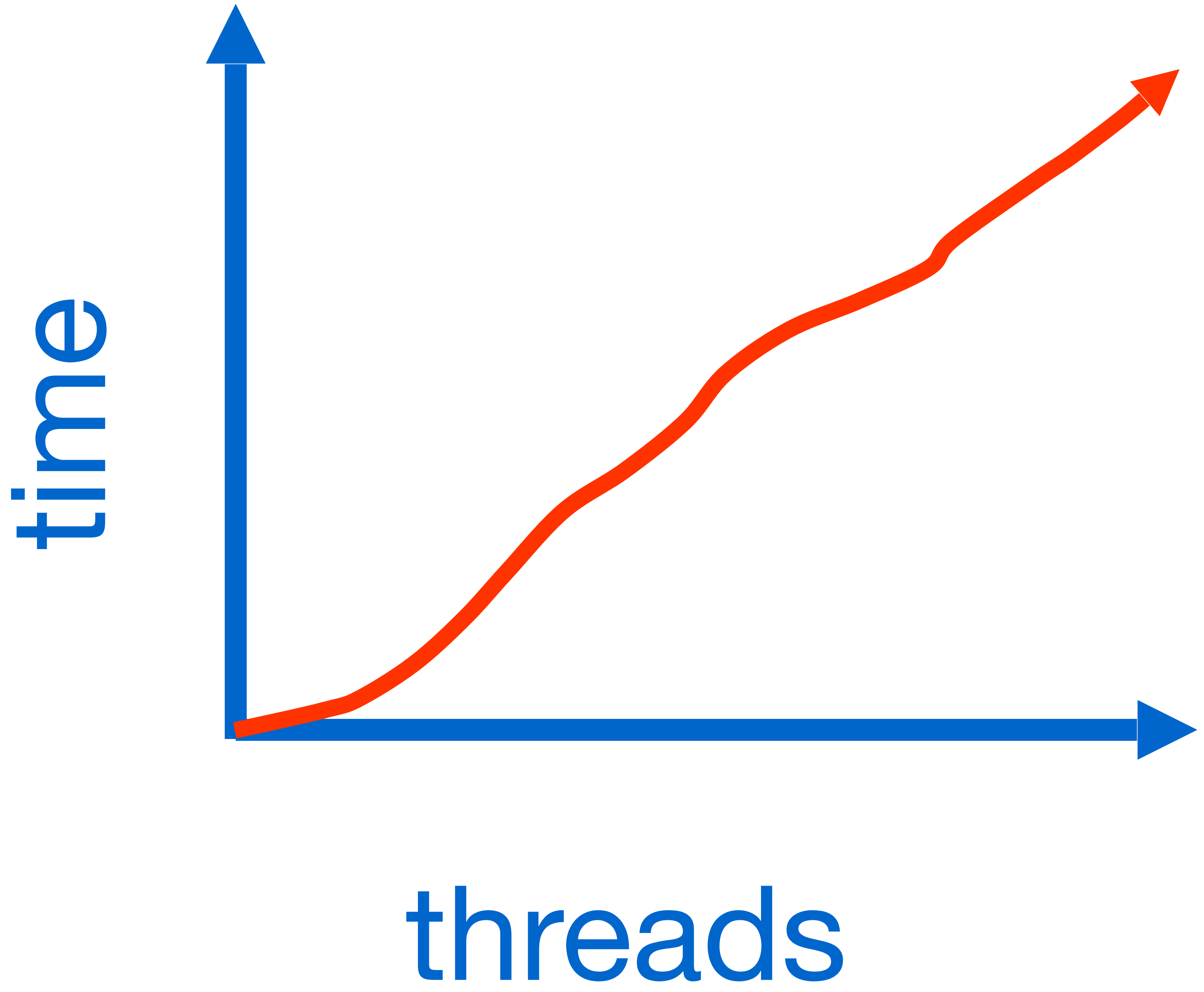
Everyone tries TAS



Problems

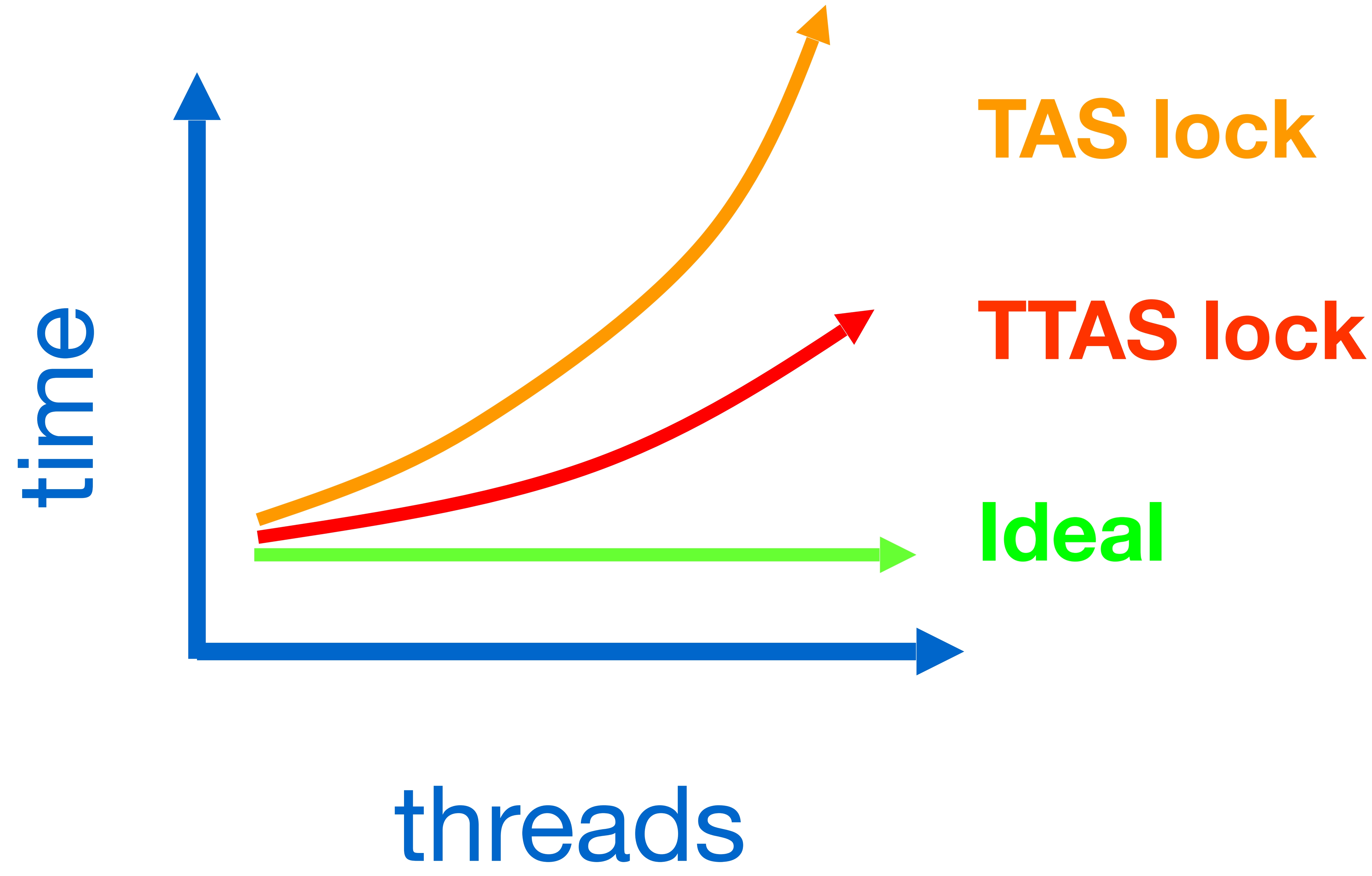
- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others’ caches
- Eventually ***quiesces*** after lock acquired
- ***Quiescence duration*** is the time between lock release and lock acquire
  - How long does this take?

Quiescence Time

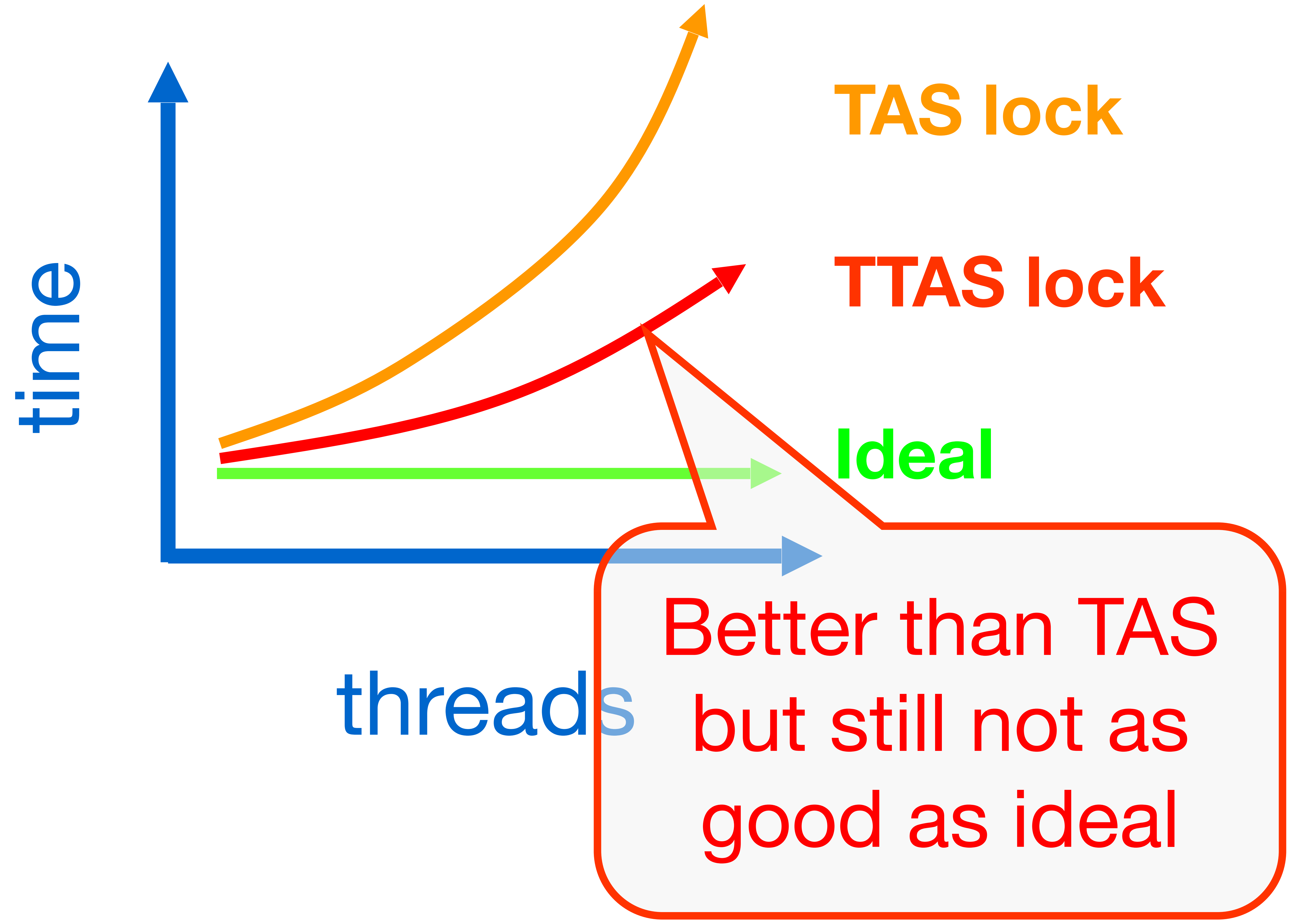


Increses linearly with the number of processors for bus architecture

Mystery Explained



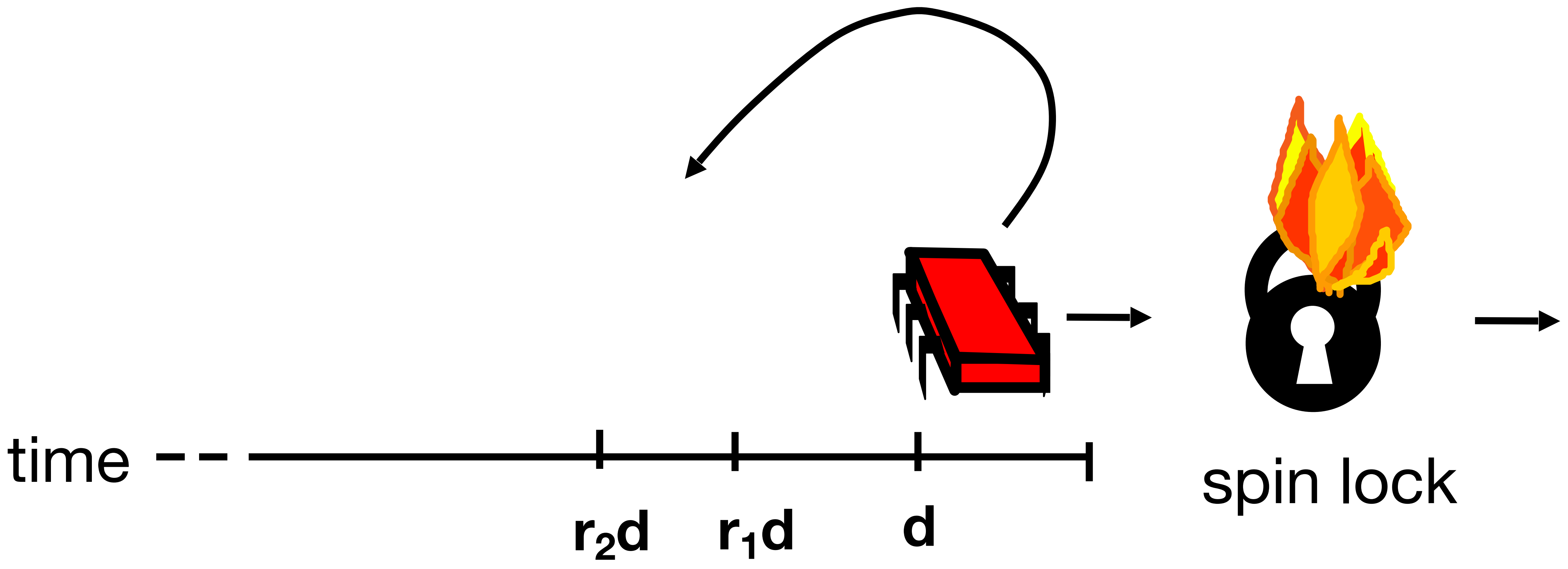
Mystery Explained



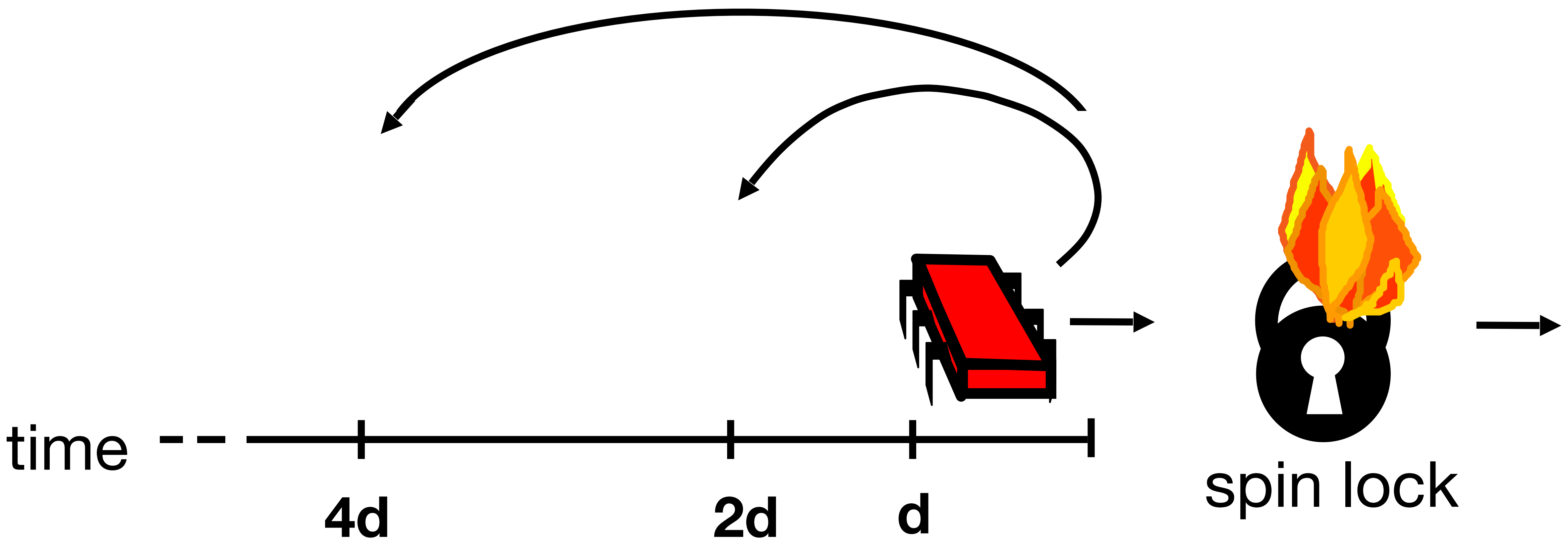
Better than TAS but still not as good as ideal

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 Solution — Exponential backoff



If I fail to get lock

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

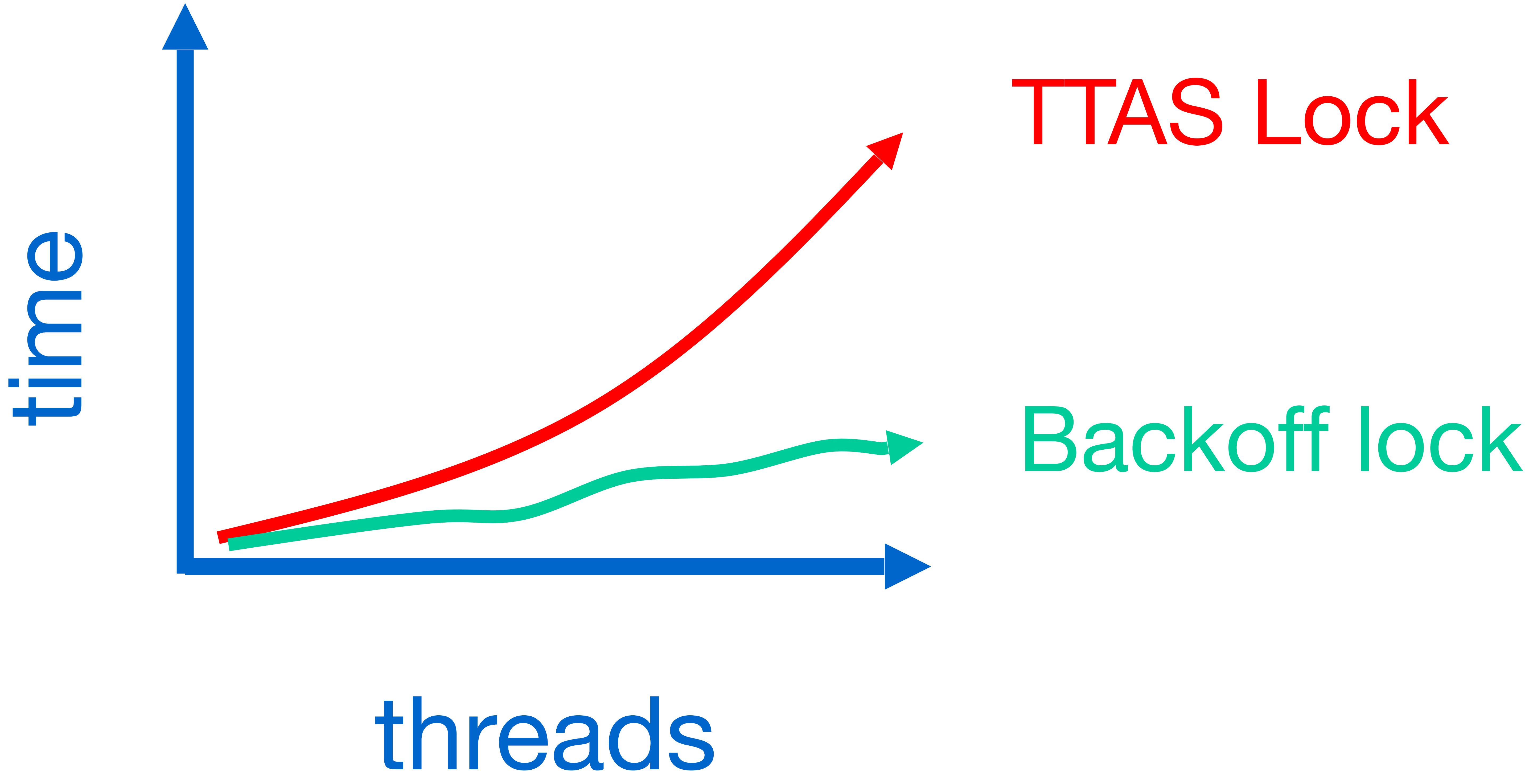
Exponential Backoff

```
module Backoff = struct
  type t = { min_delay : int; max_delay : int; mutable limit : int }

  let create min_delay max_delay = { min_delay; max_delay; limit = min_delay }

  let backoff t =
    (* Backoff for a random duration between 0 and 'limit' using cpu_relax *)
    (* Randomization prevents synchronized collisions between threads *)
    let delay = Random.int (t.limit + 1) in
    for _ = 1 to delay do
      Domain.cpu_relax ()
      (* Instructs the core to slow down fast(er than sleep()) *)
    done;
    (* Exponentially increase the limit, capped at max_delay *)
    t.limit <- min t.max_delay (t.limit * 2)
end
```

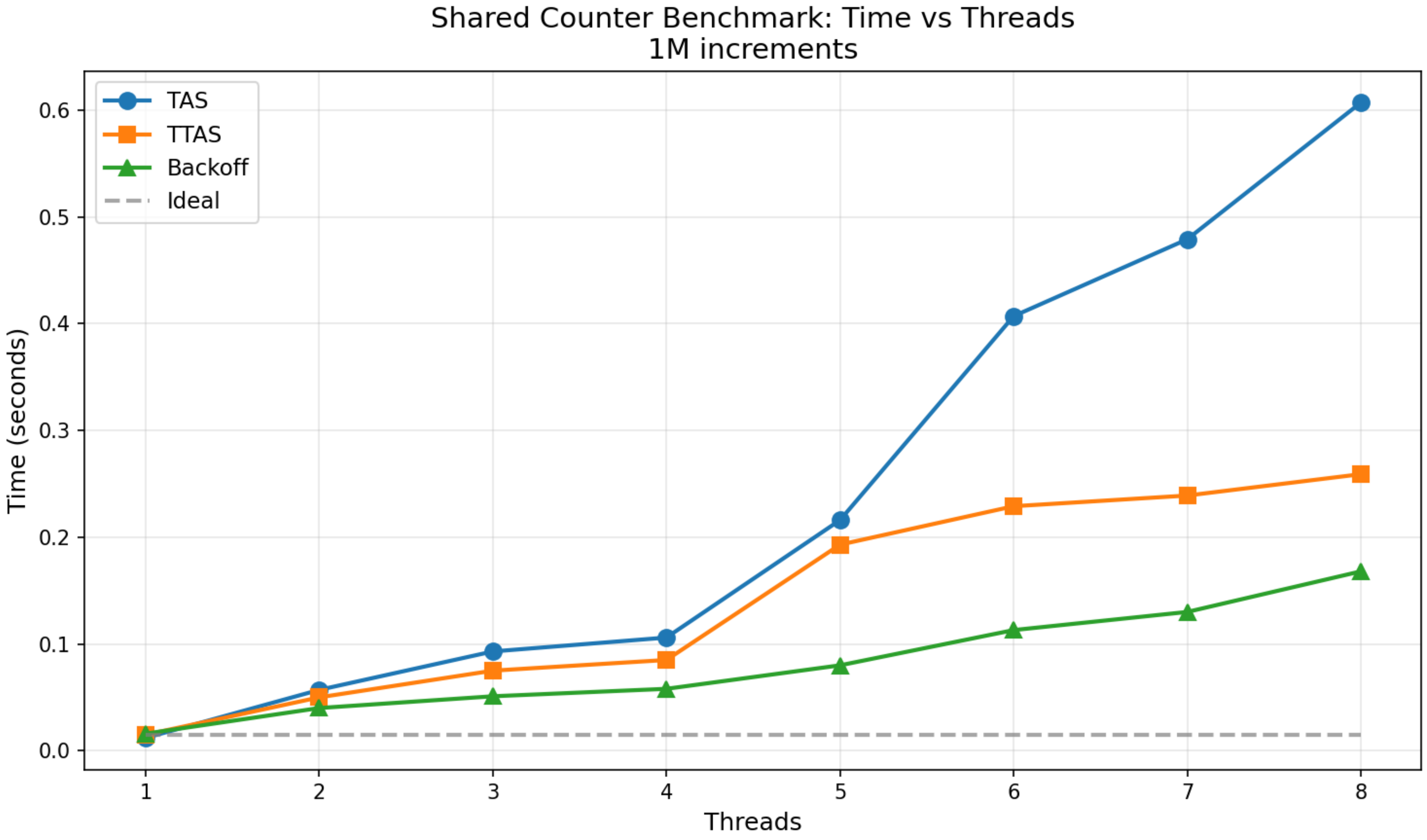
BackoffLock

```
module MakeBackoffLock (P : BACKOFF_PARAMS) : Lock.LOCK = struct
  type t = { state : bool Atomic.t }

  let create () = { state = Atomic.make false }

  let lock t =
    let backoff = Backoff.create P.min_delay P.max_delay in
    (* Outer loop: keep trying until we get the lock *)
    while
      (* Inner loop: spin-read until lock appears free *)
      while Atomic.get t.state do
        ()
      done;
      (* Lock looks free, try to acquire *)
      Atomic.exchange t.state true
    do
      (* Failed to acquire - back off before trying again *)
      Backoff.backoff backoff
    done

  let unlock t = Atomic.set t.state false
end
```

Spin-waiting Overhead



Results on 8-core M2 Apple Silicon



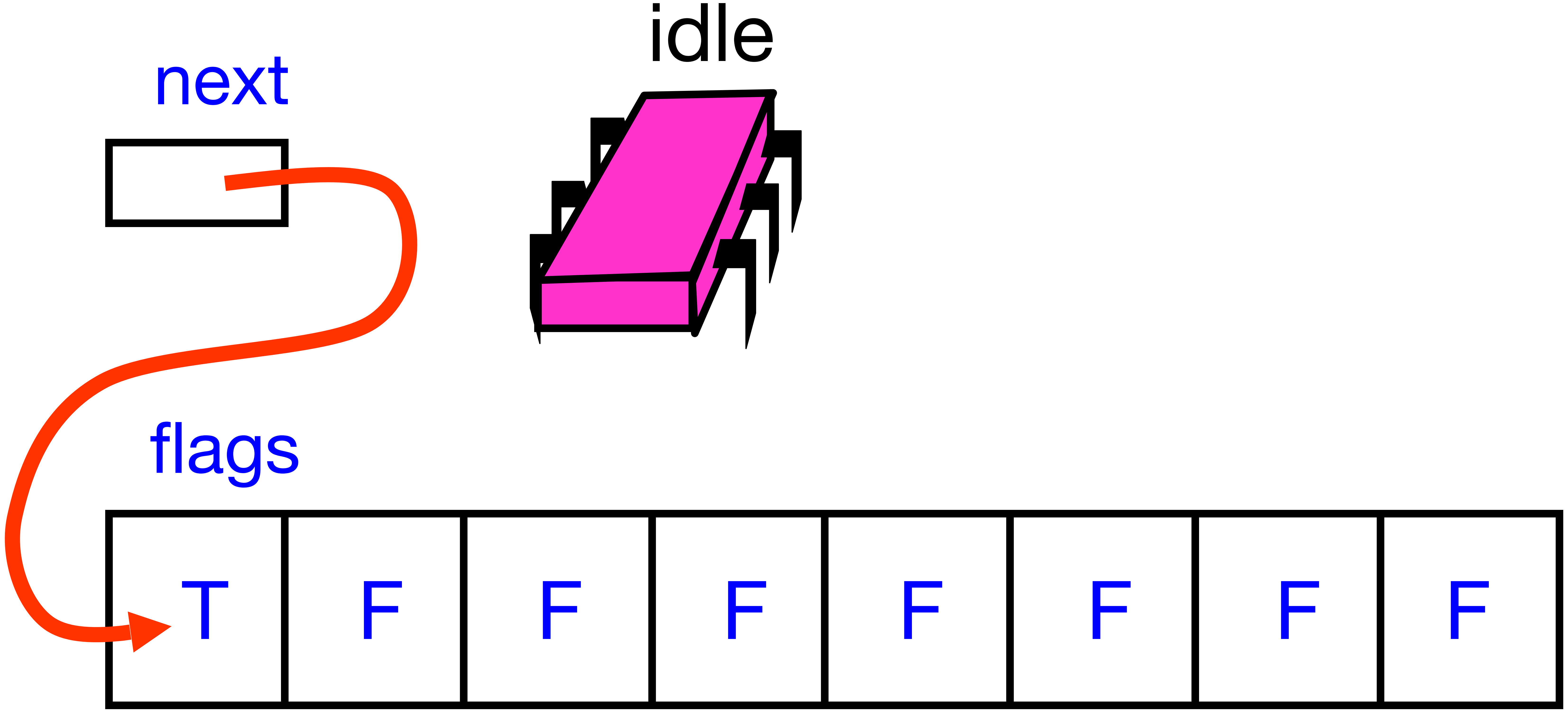
Backoff: Other issues

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

Idea

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

Anderson Queue Lock



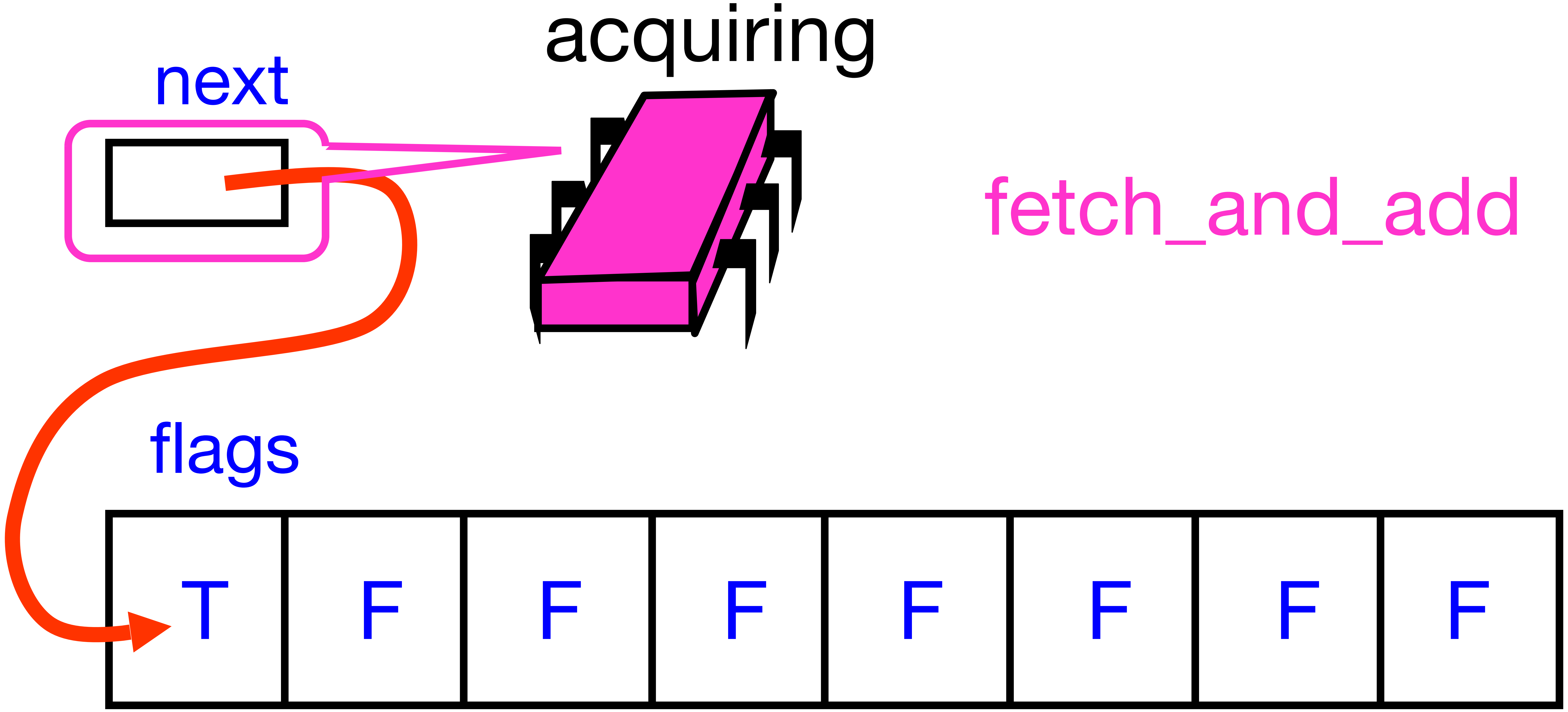
idle

next

flags

T	F	F	F	F	F	F	F

Anderson Queue Lock



acquiring

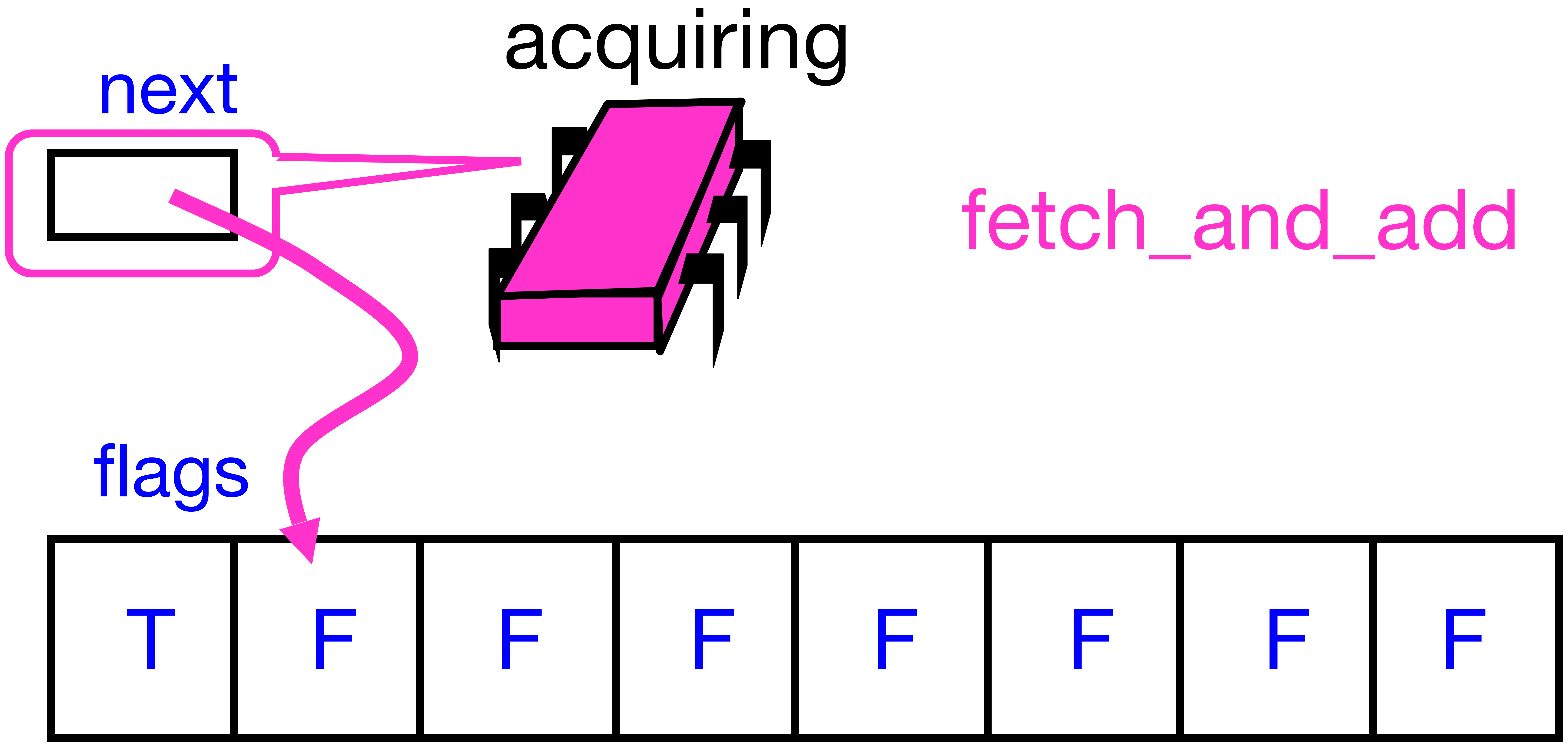
next

fetch_and_add

flags

T	F	F	F	F	F	F	F

Anderson Queue Lock



acquiring

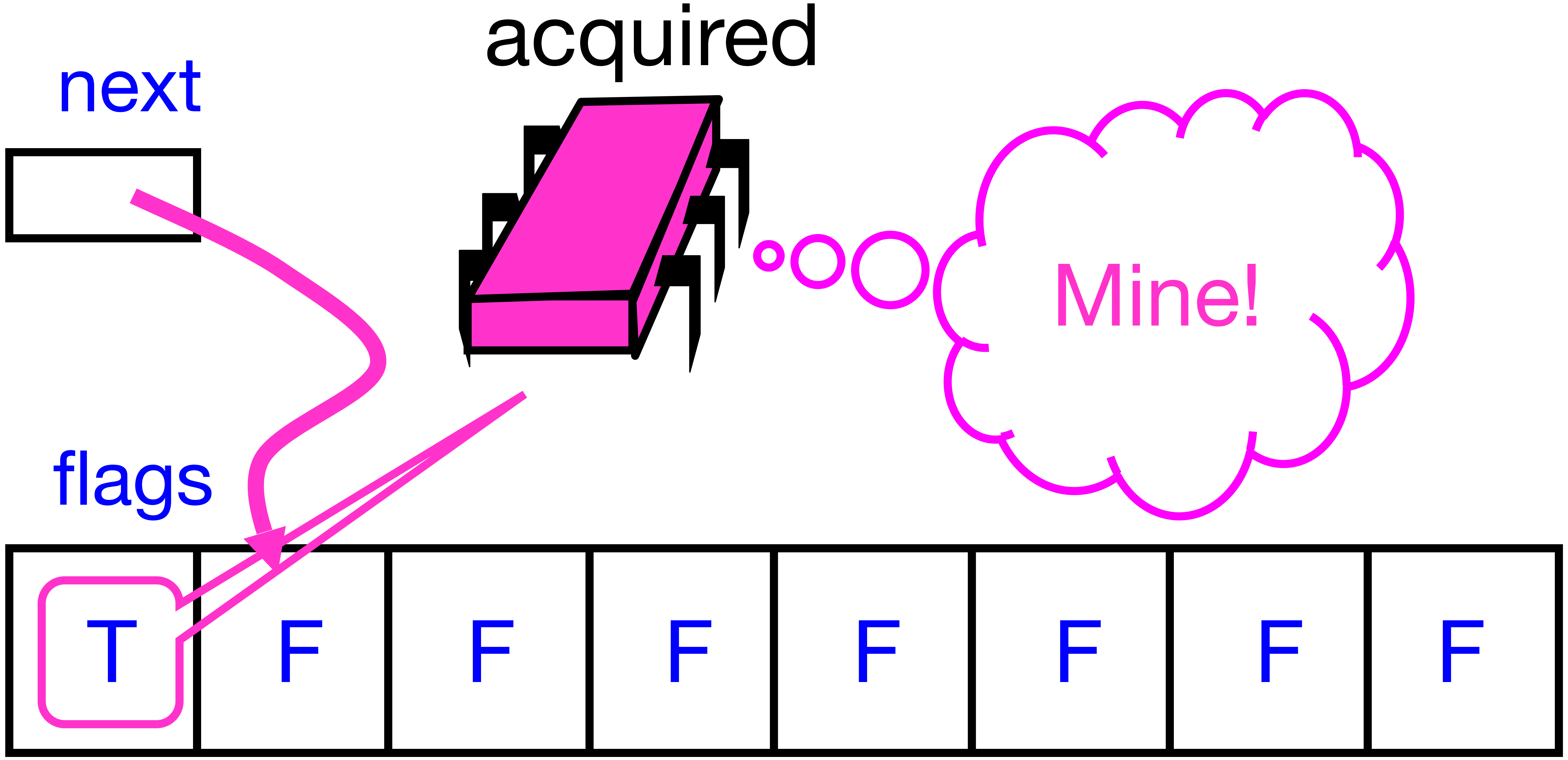
next

fetch_and_add

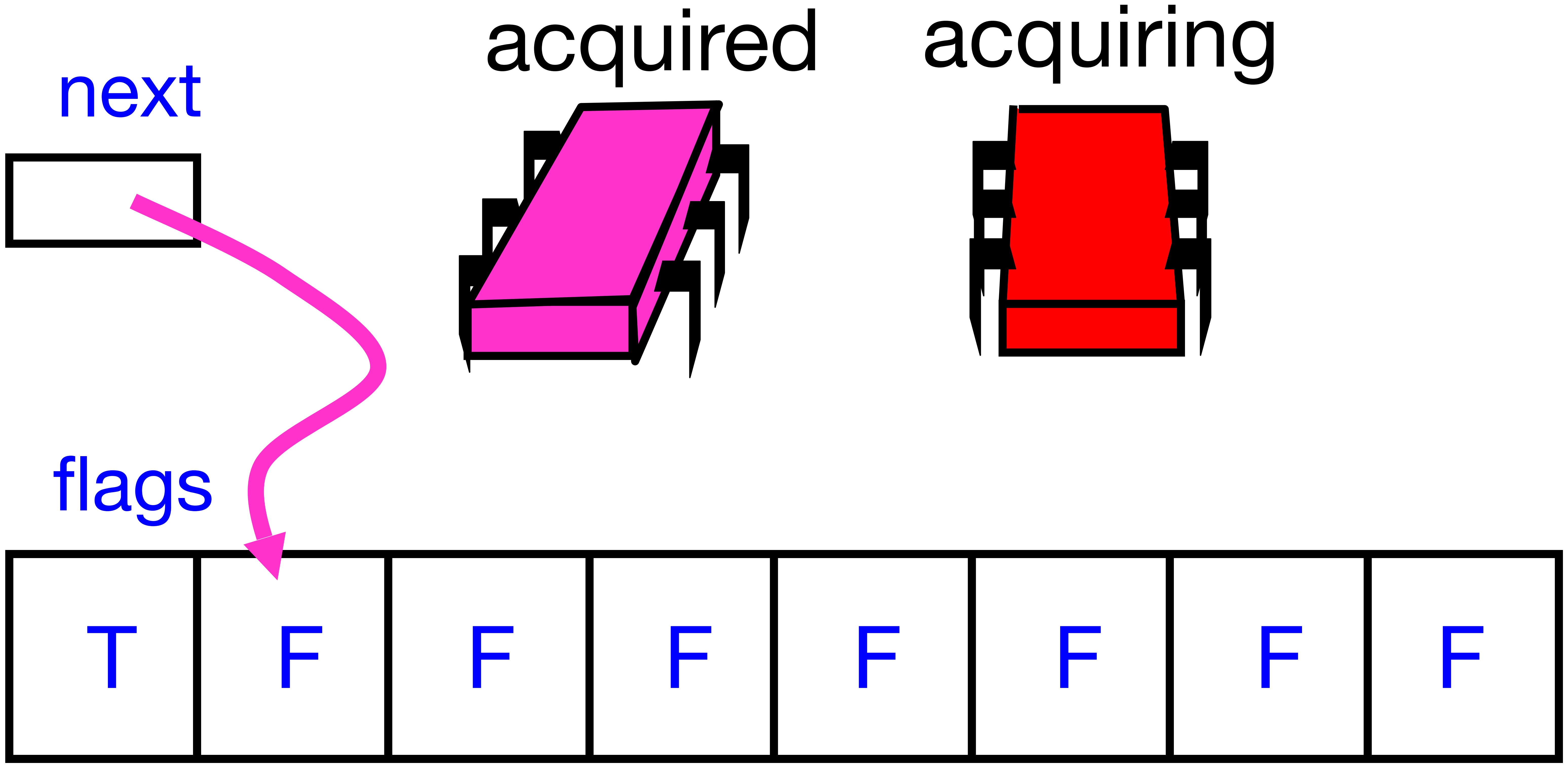
flags

T	F	F	F	F	F	F	F

Anderson Queue Lock

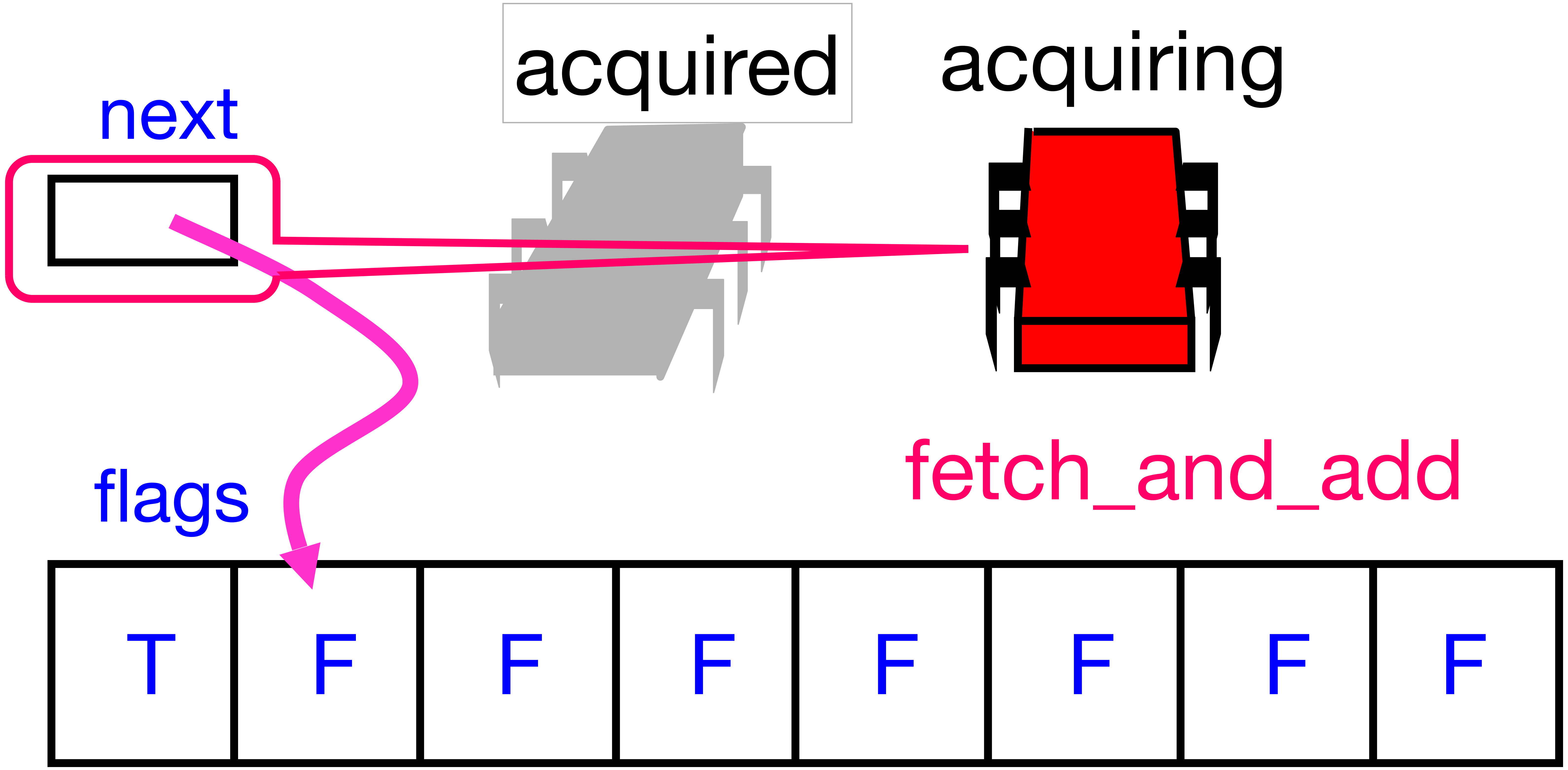


Anderson Queue Lock



T	F	F	F	F	F	F	F

Anderson Queue Lock



next

acquired

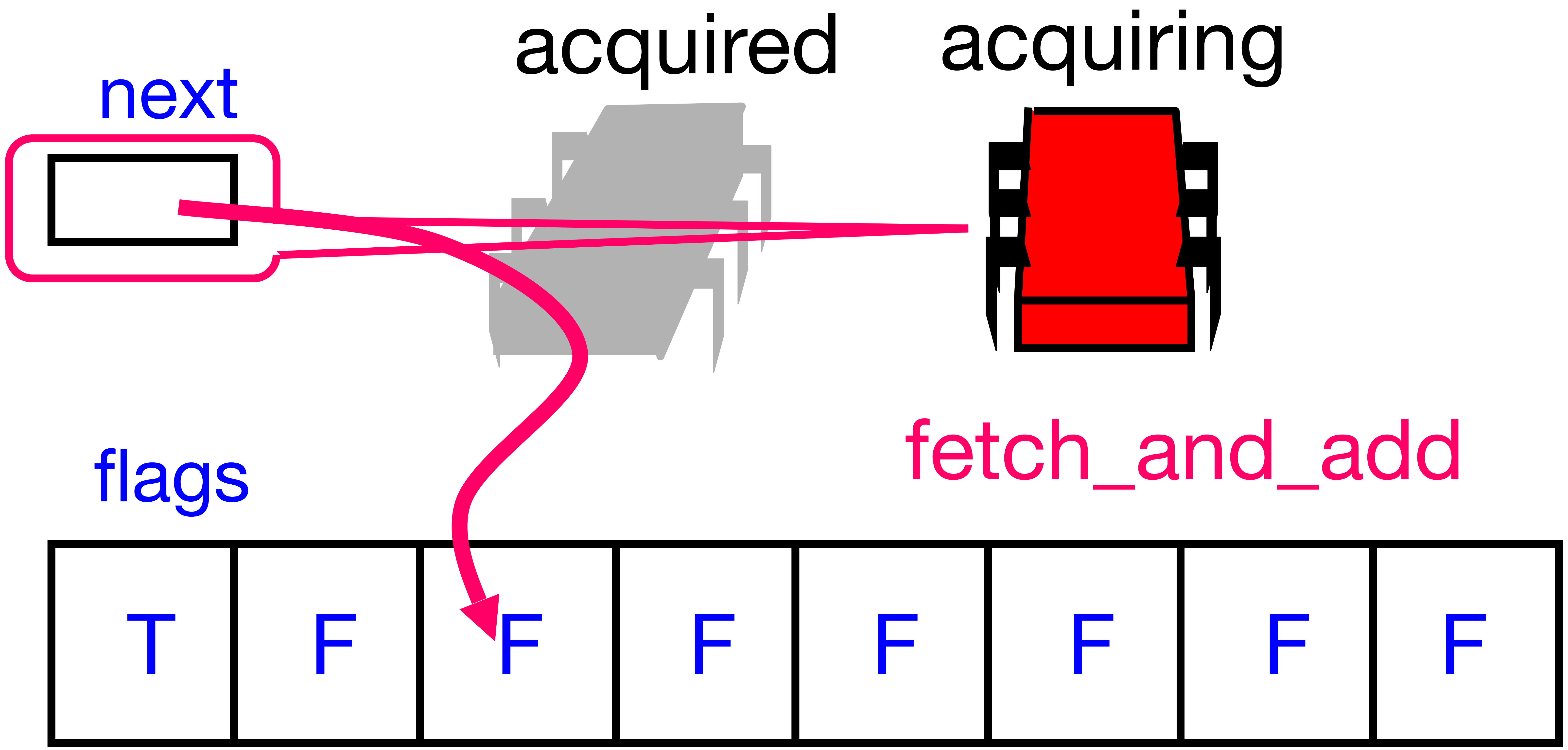
acquiring

fetch_and_add

flags

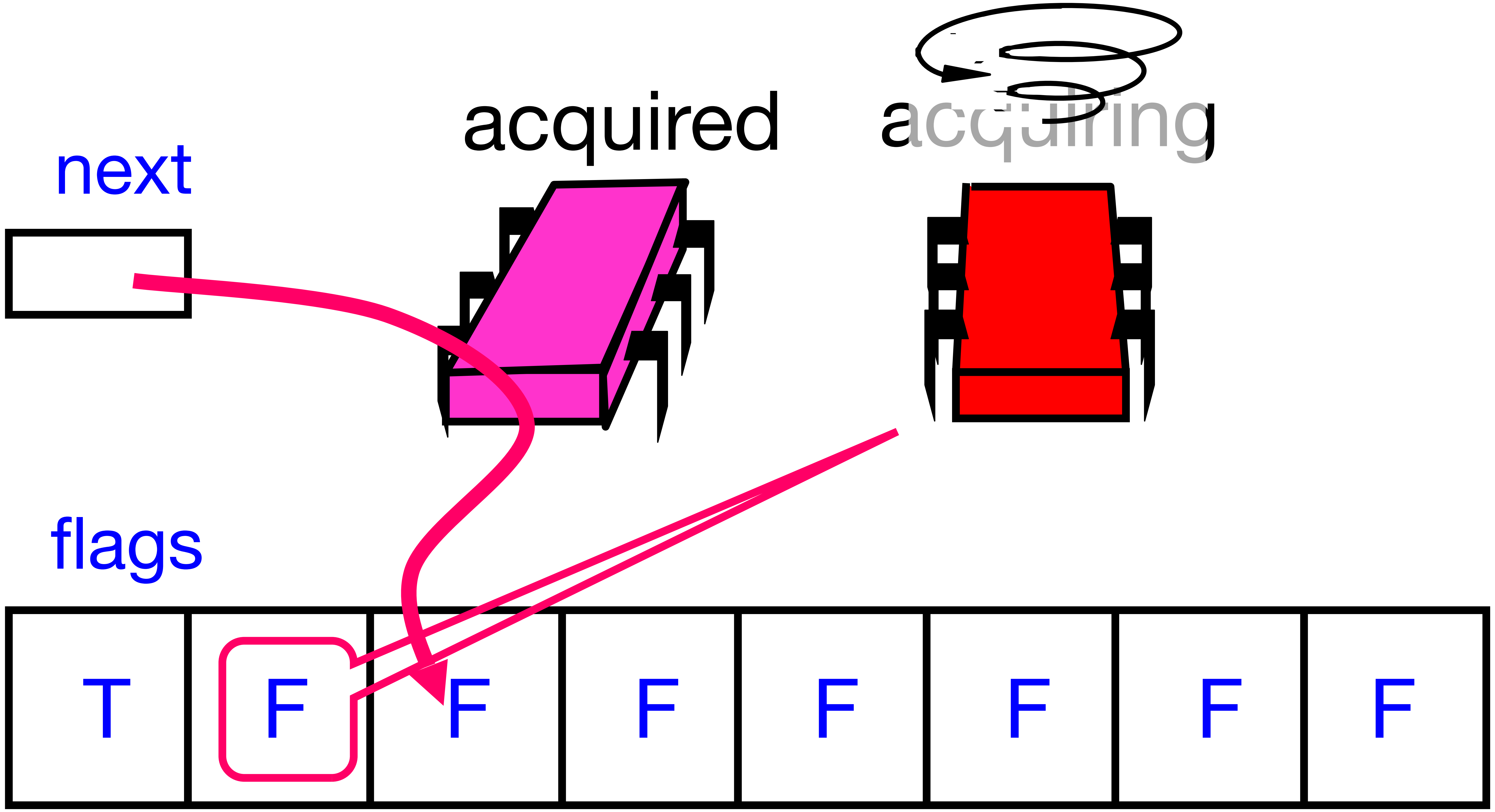
T	F	F	F	F	F	F	F

Anderson Queue Lock



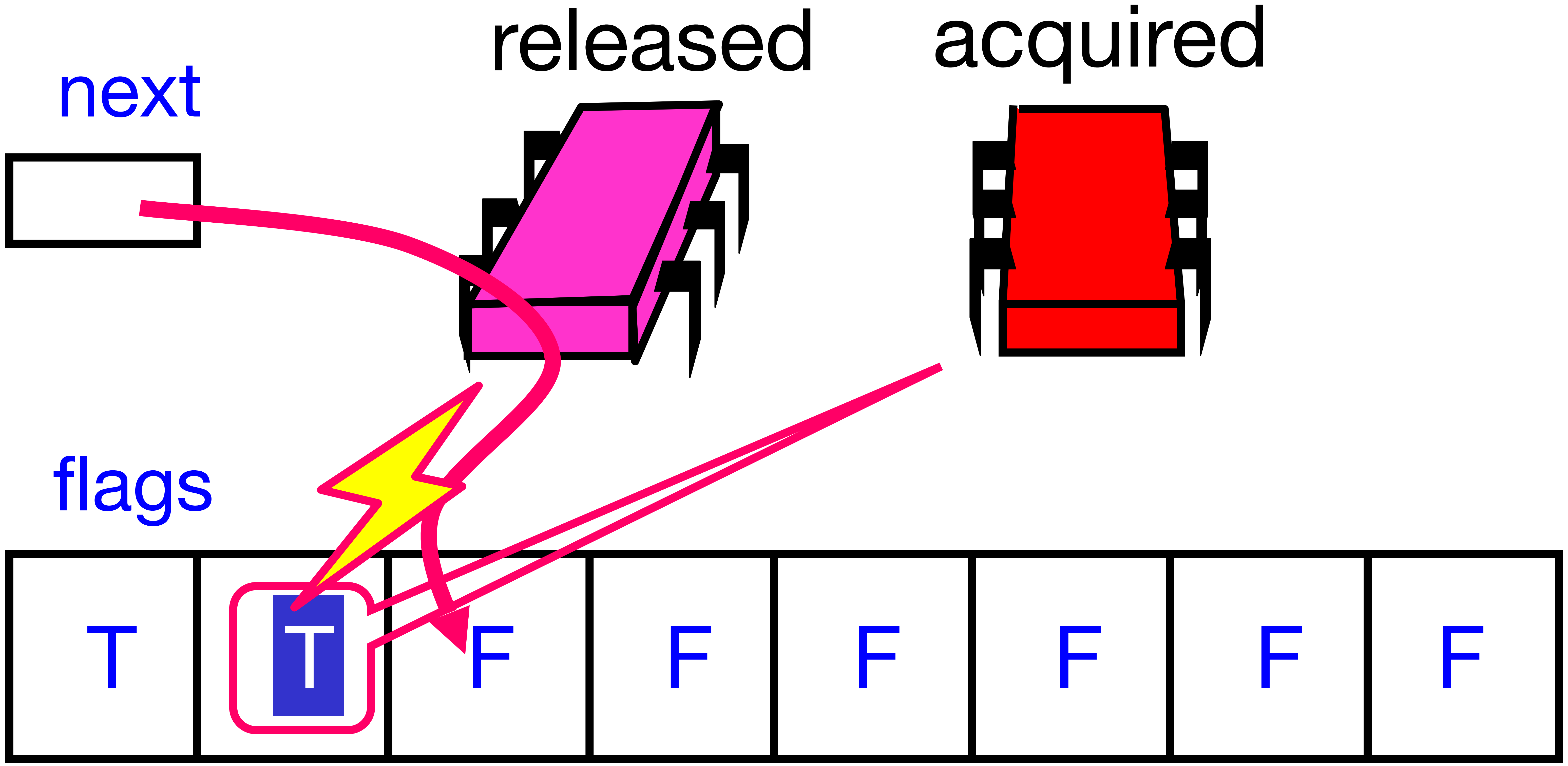
T	F	F	F	F	F	F	F

Anderson Queue Lock

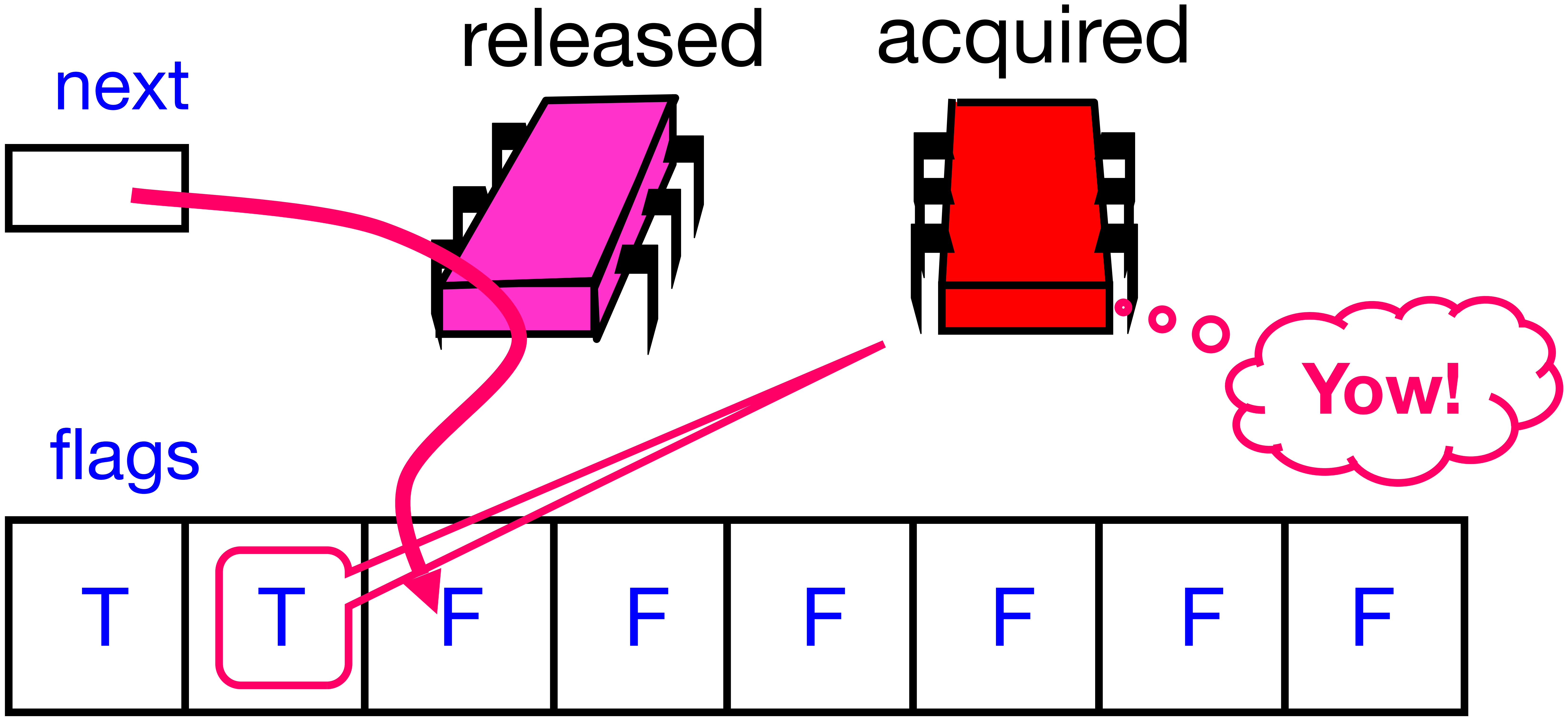


T	F	F	F	F	F	F	F

Anderson Queue Lock



Anderson Queue Lock



released

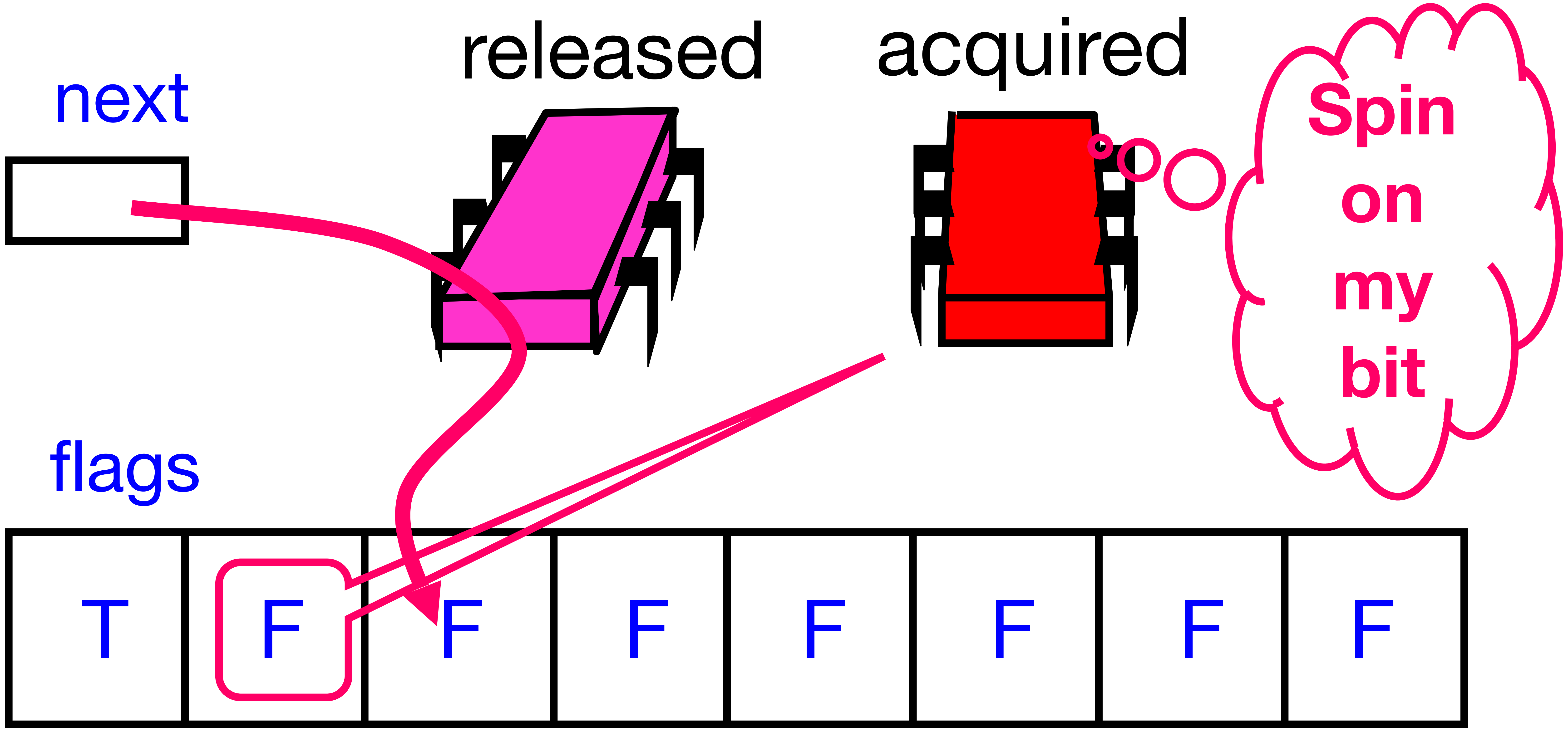
acquired

next

flags

T	T	F	F	F	F	F	F

Local Spinning



next

released

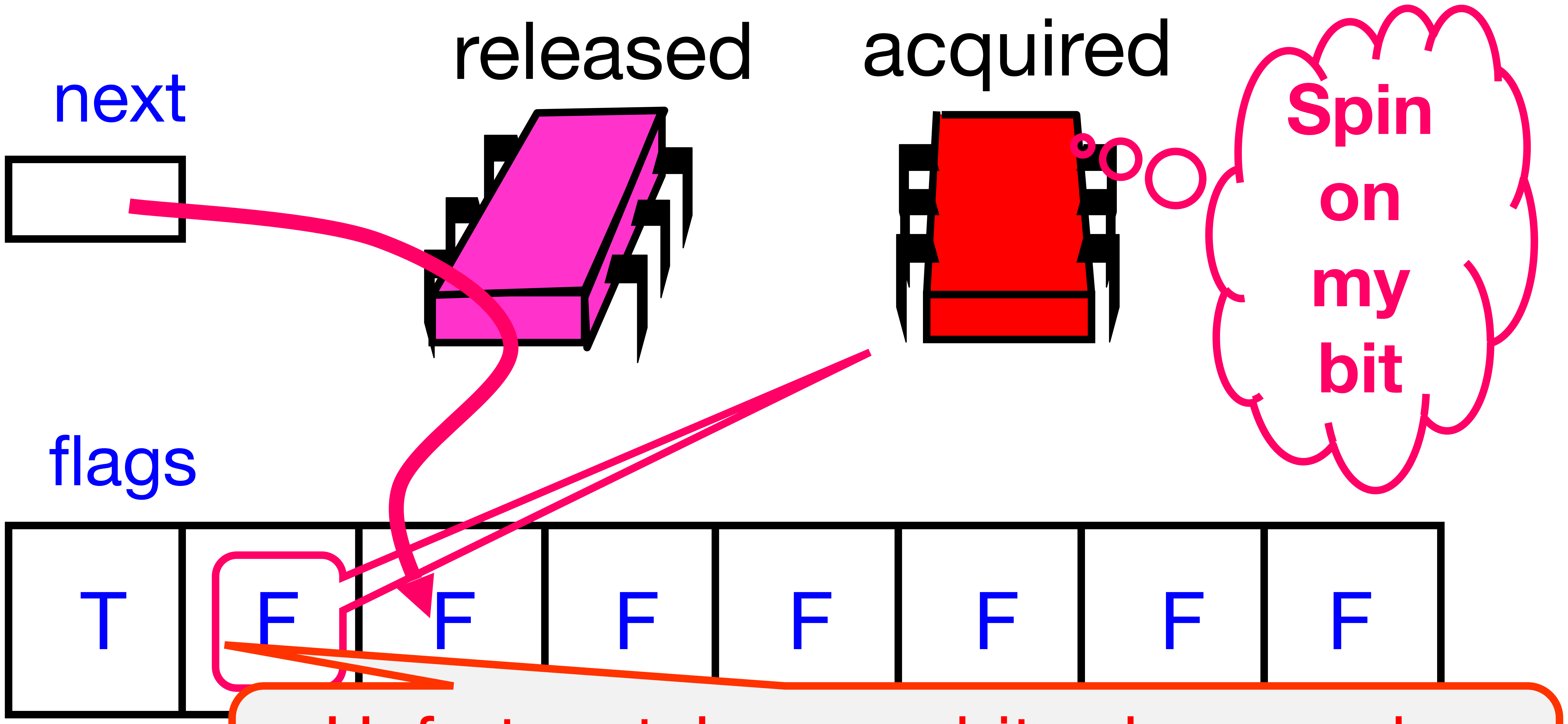
acquired

Spin on my bit

flags

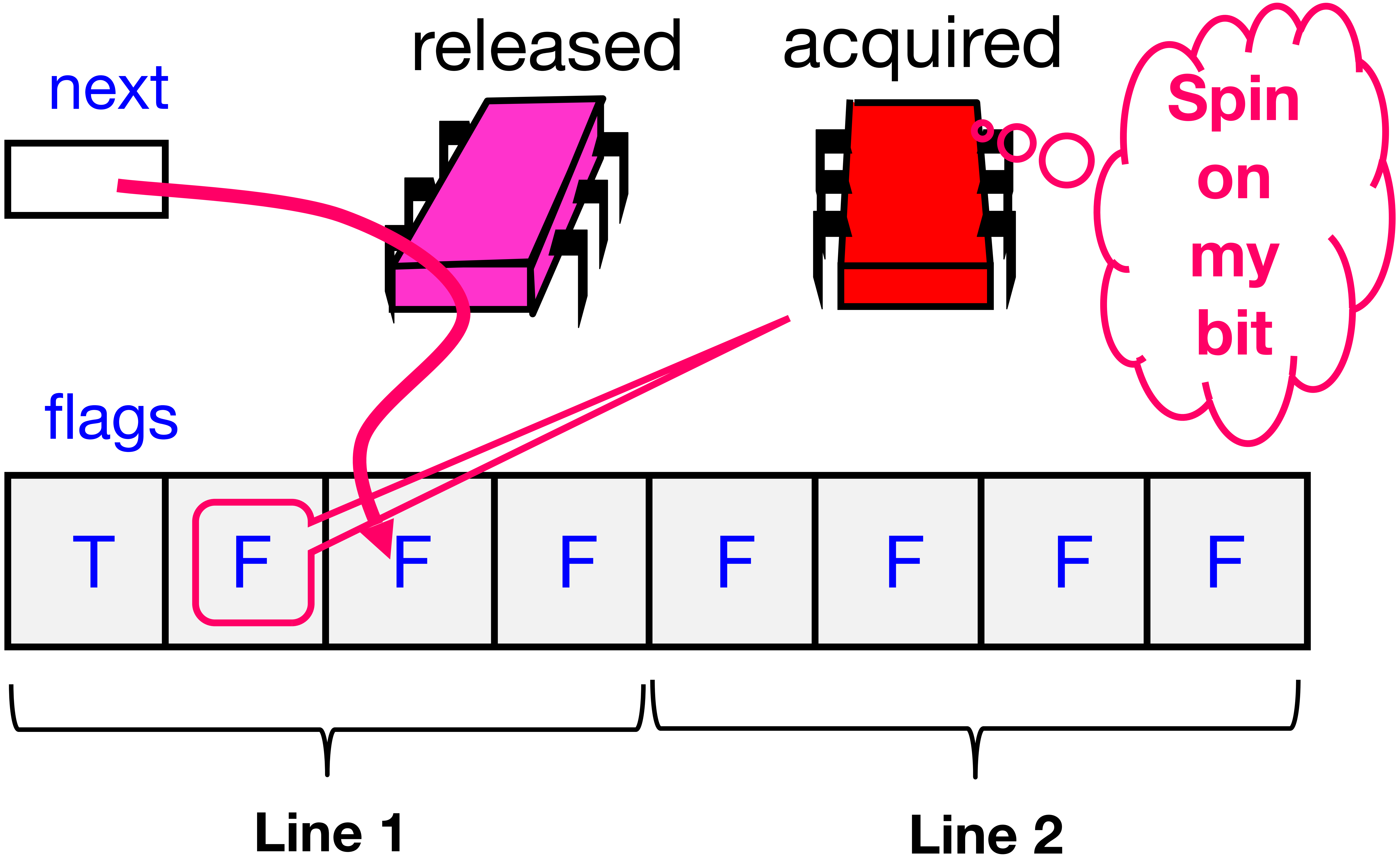
T	F	F	F	F	F	F	F

Local Spinning

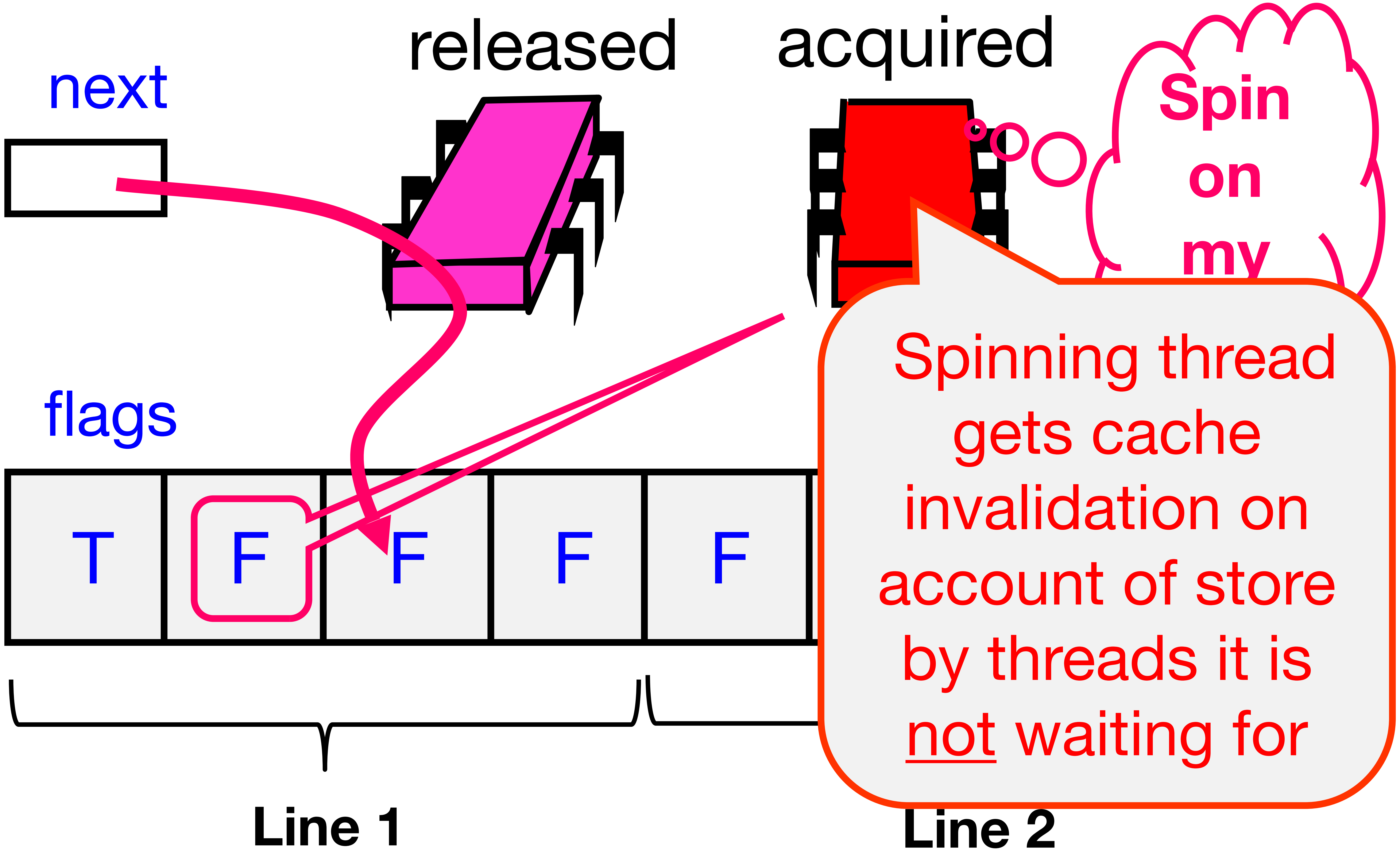


Unfortunately many bits share cache line

False Sharing

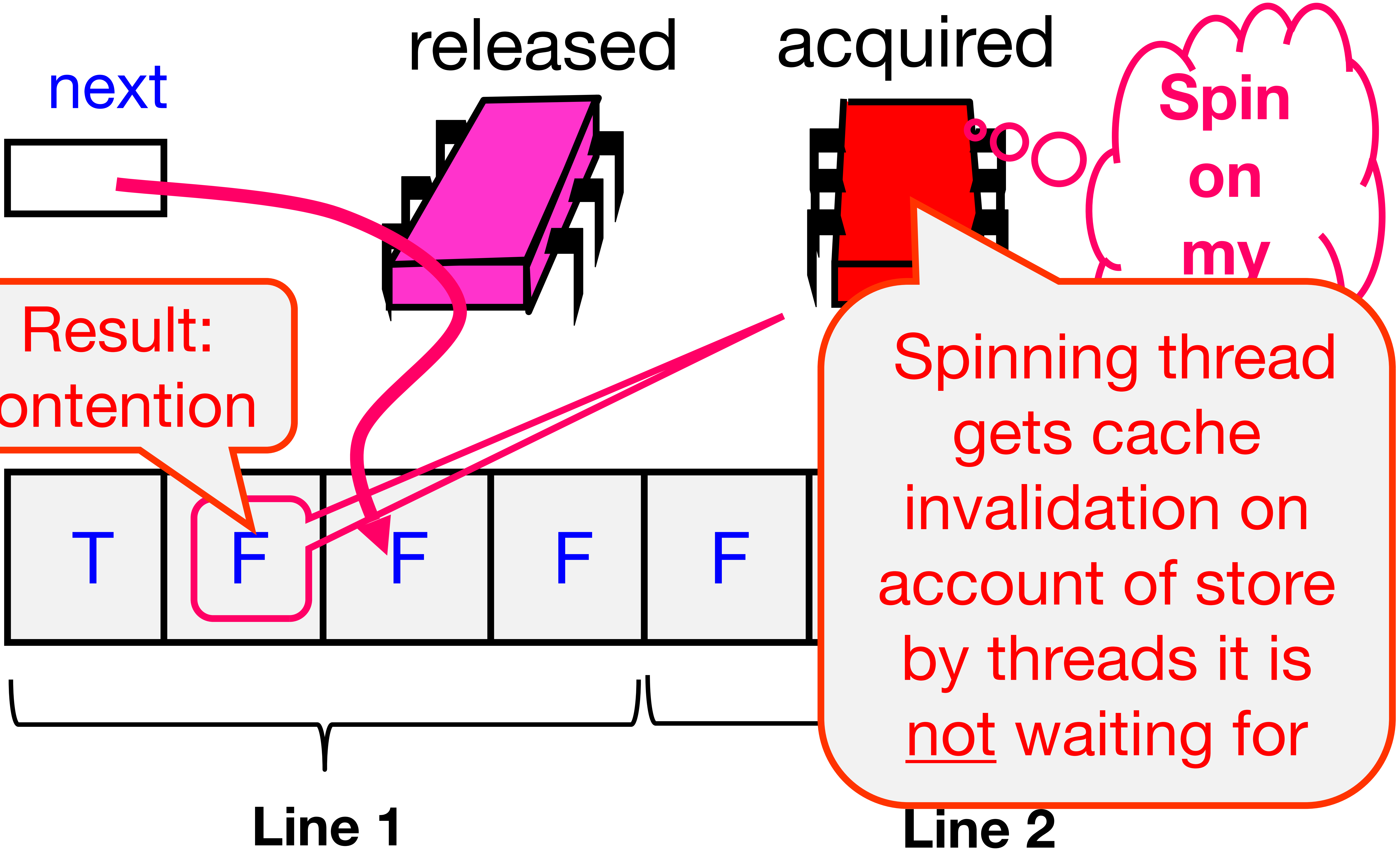


False Sharing



Spinning thread gets cache invalidation on account of store by threads it is not waiting for

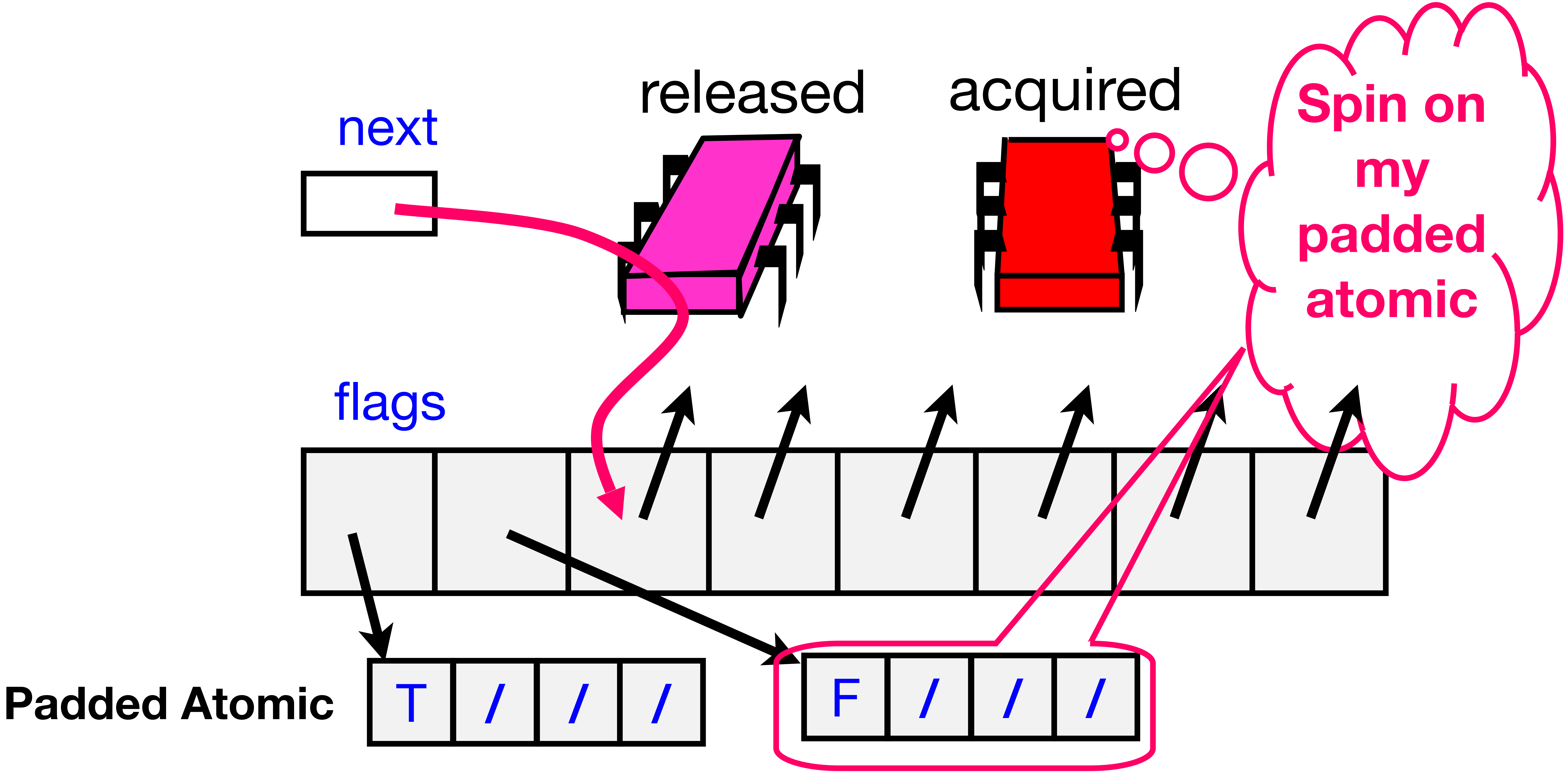
False Sharing



Result: contention

Spinning thread gets cache invalidation on account of store by threads it is not waiting for

OCaml Solution: Heap-separated allocation + padding



OCaml ALock

```
type t = {
  flags : bool Atomic.t array;
  tail : int Atomic.t;
  capacity : int;
  my_slot : int Domain.DLS.key;
}
```

OCaml ALock

```
type t = {
  flags : bool Atomic.t array;
  tail : int Atomic.t;
  capacity : int;
  my_slot : int Domain.DLS.key;
}
```

```
let create_with_capacity capacity =
  if capacity <= 0 then
    invalid_arg "ALock capacity must be positive";

  (* Create array of atomic booleans using
     make_contended to prevent false sharing *)
  let flags =
    Array.init capacity (fun i ->
        (* Only slot 0 starts as true (available) *)
        Atomic.make_contended (i = 0))
  in
  {
    flags;
    tail = Atomic.make 0;
    capacity;
    (* DLS for remembering slot taken by domain *)
    my_slot = Domain.DLS.new_key (fun () -> -1);
  }
```

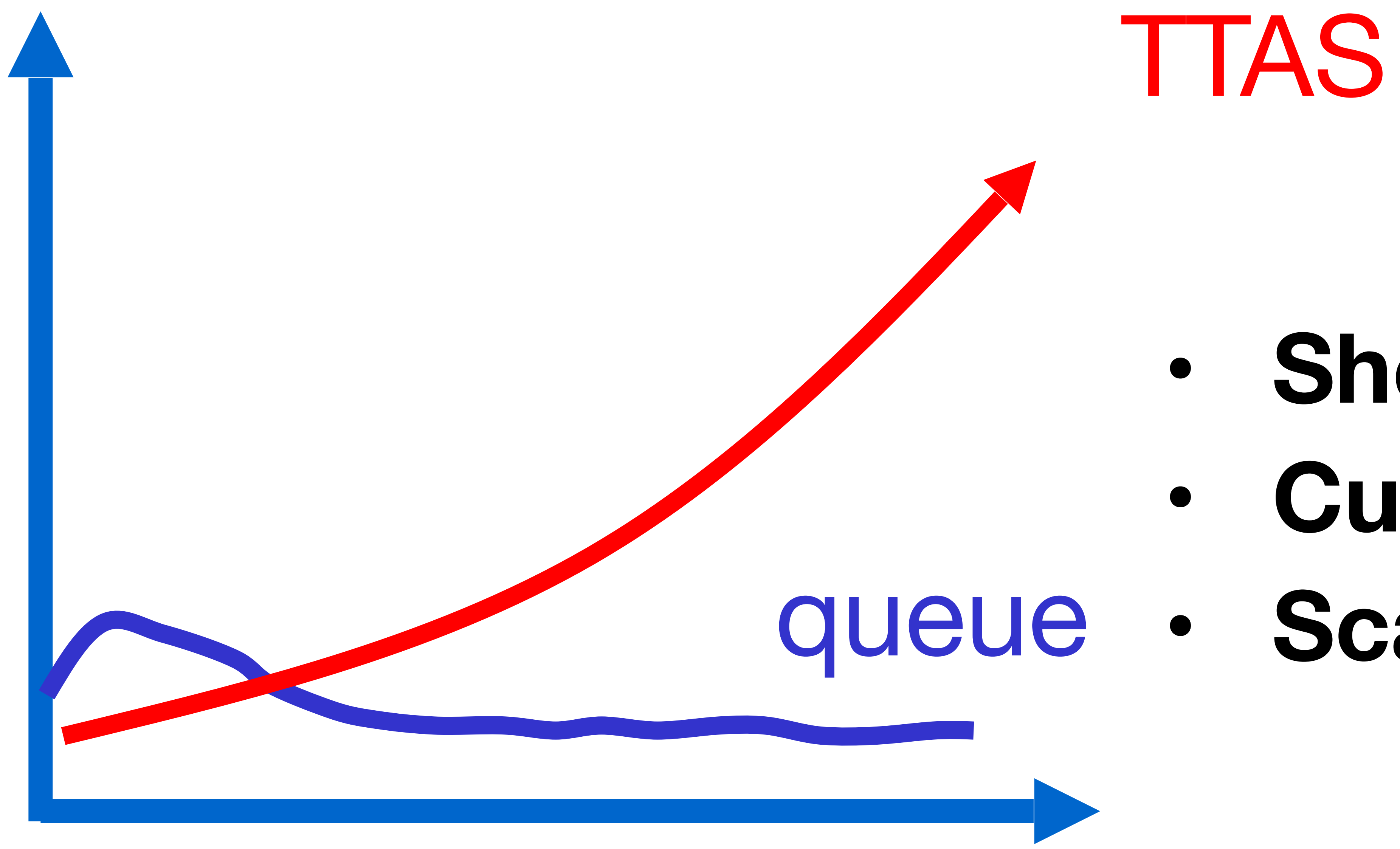
OCaml ALock

```
let lock t =
  (* Get my slot using atomic fetch-and-increment *)
  let slot = (Atomic.fetch_and_add t.tail 1) mod t.capacity in

  (* Store slot in domain-local storage for unlock *)
  Domain.DLS.set t.my_slot slot;

  (* Cache the flag reference to avoid repeated array indexing *)
  let my_flag = t.flags.(slot) in

  (* Spin on MY flag until it becomes true *)
  (* This is the key: each thread spins on a DIFFERENT location *)
  while not (Atomic.get my_flag) do
    Domain.cpu_relax ()
  done
```

OCaml ALock

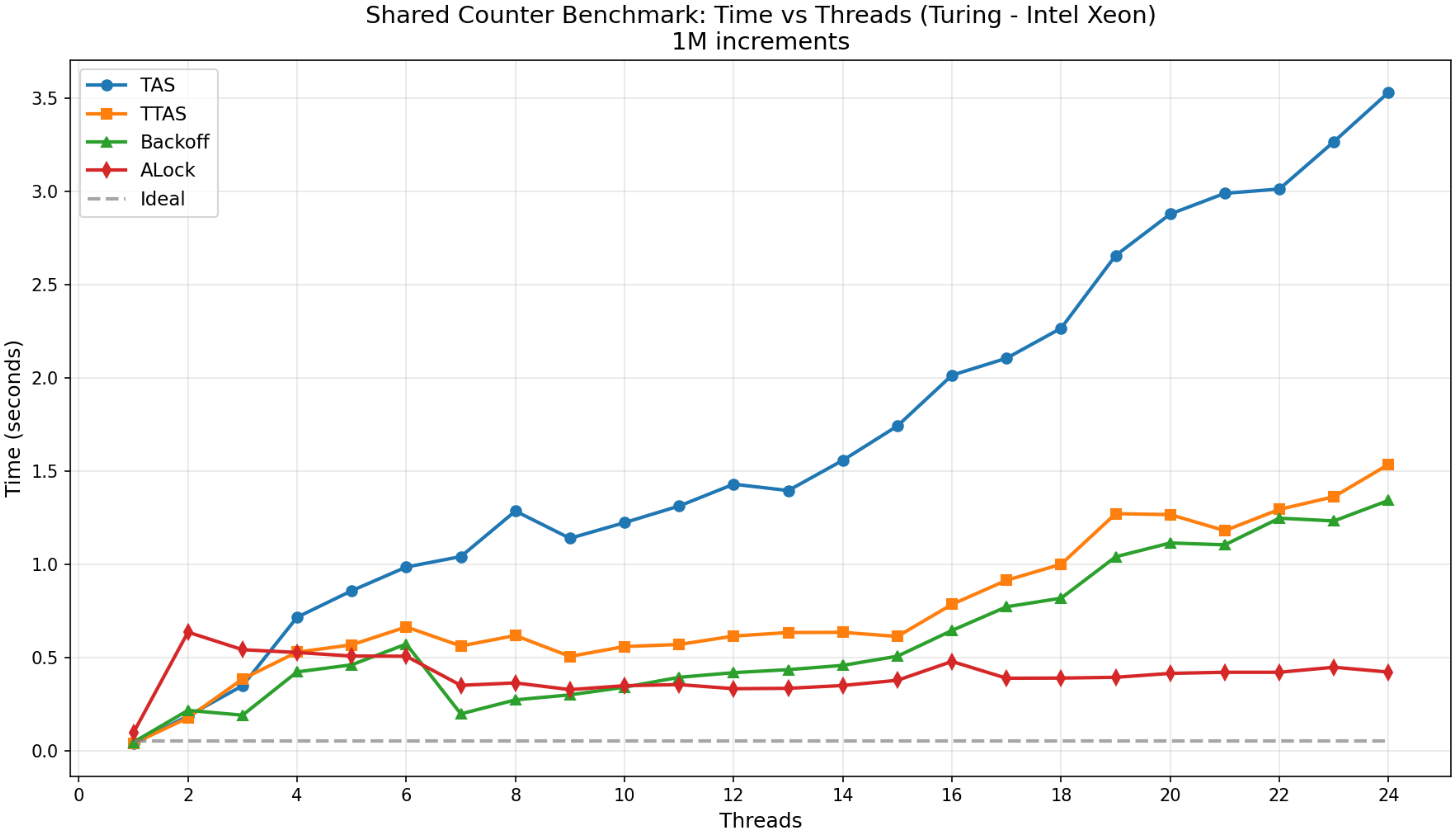
```
let unlock t =
   (* Get my slot from domain-local storage *)
   let slot = Domain.DLS.get t.my_slot in

   if slot = -1 then
     failwith "unlock called without corresponding lock";

   (* Cache flag references *)
   let my_flag = t.flags.(slot) in
   let next_slot = (slot + 1) mod t.capacity in
   let next_flag = t.flags.(next_slot) in

   (* Clear my flag *)
   Atomic.set my_flag false;

   (* Signal the next thread *)
   Atomic.set next_flag true
```

Performance



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

Results on 28-core Intel Xeon Gold 5120

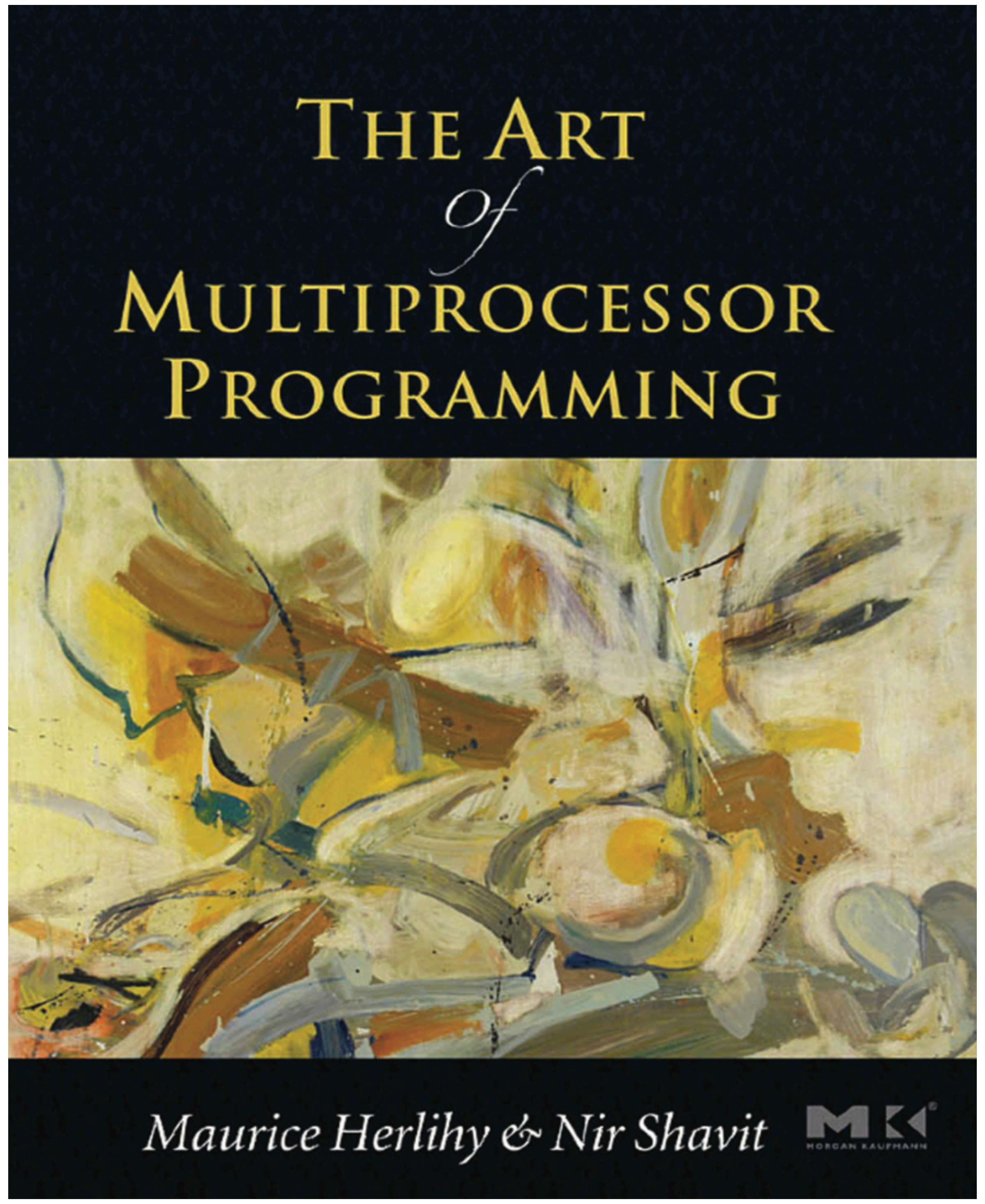


Anderson Queue Lock

- Good
  - First truly scalable lock
  - Simple, easy to implement
  - Back to FCFS order (like Bakery)
- Bad
  - Space hog…
  - One bit per thread ➔ one cache line per thread
    - What if unknown number of threads?
    - What if small number of actual contenders?

More Spinlocks in the Book

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


The Art of Multiprocessor Programming
Maurice Herlihy & Nir Shavit
MK

Chapter 7