

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.

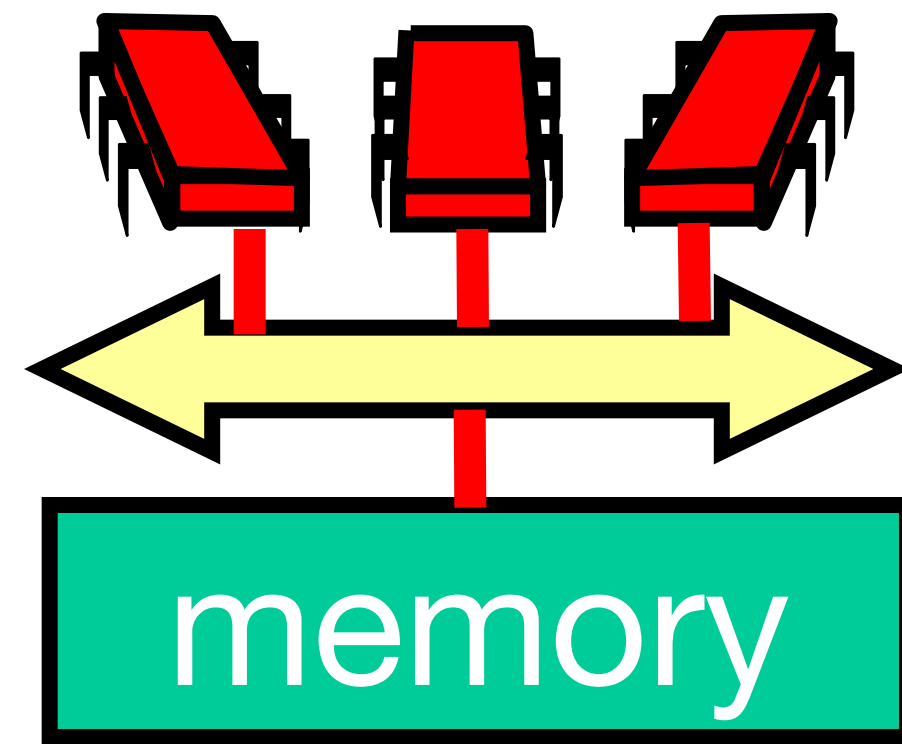
Kinds of Architectures

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

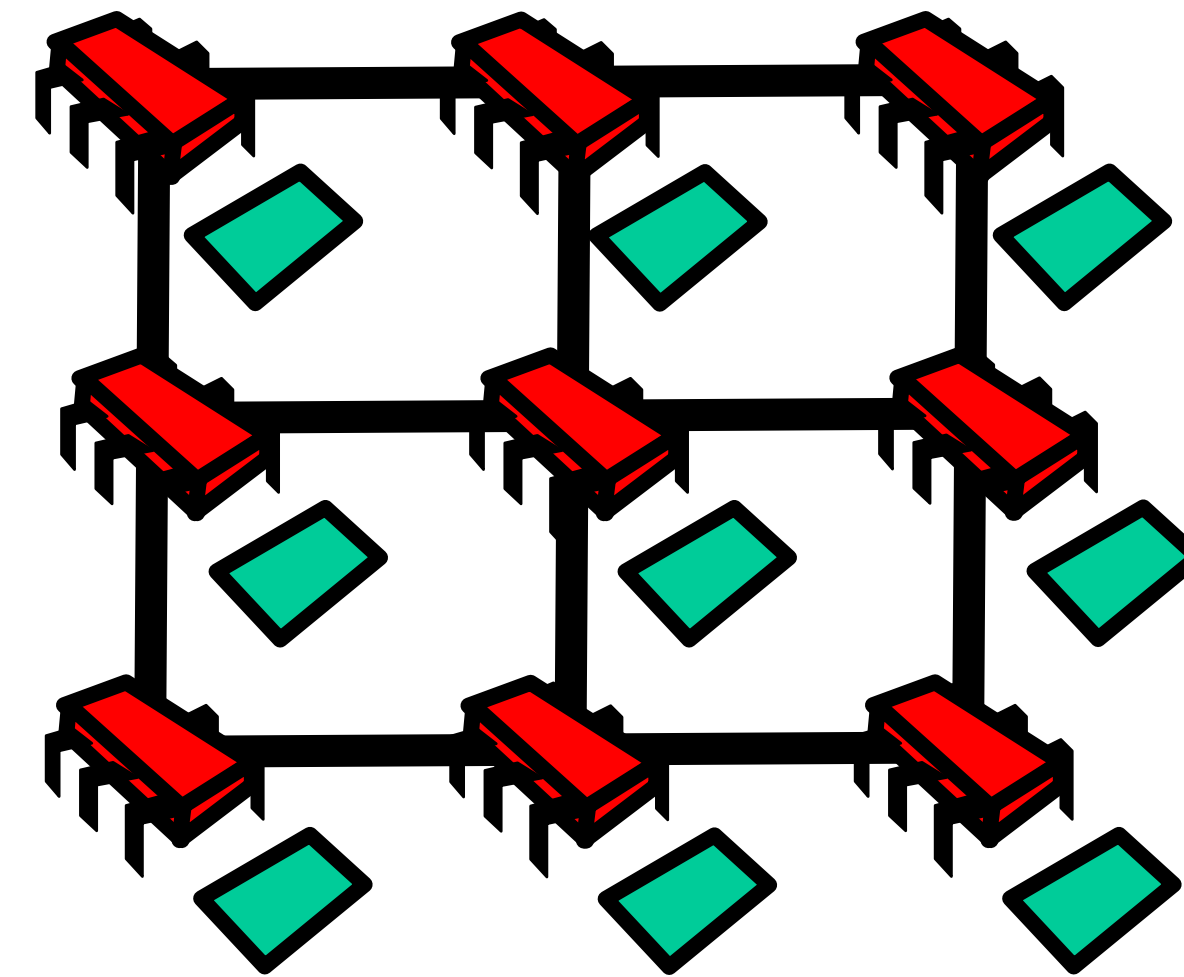


Our Focus

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

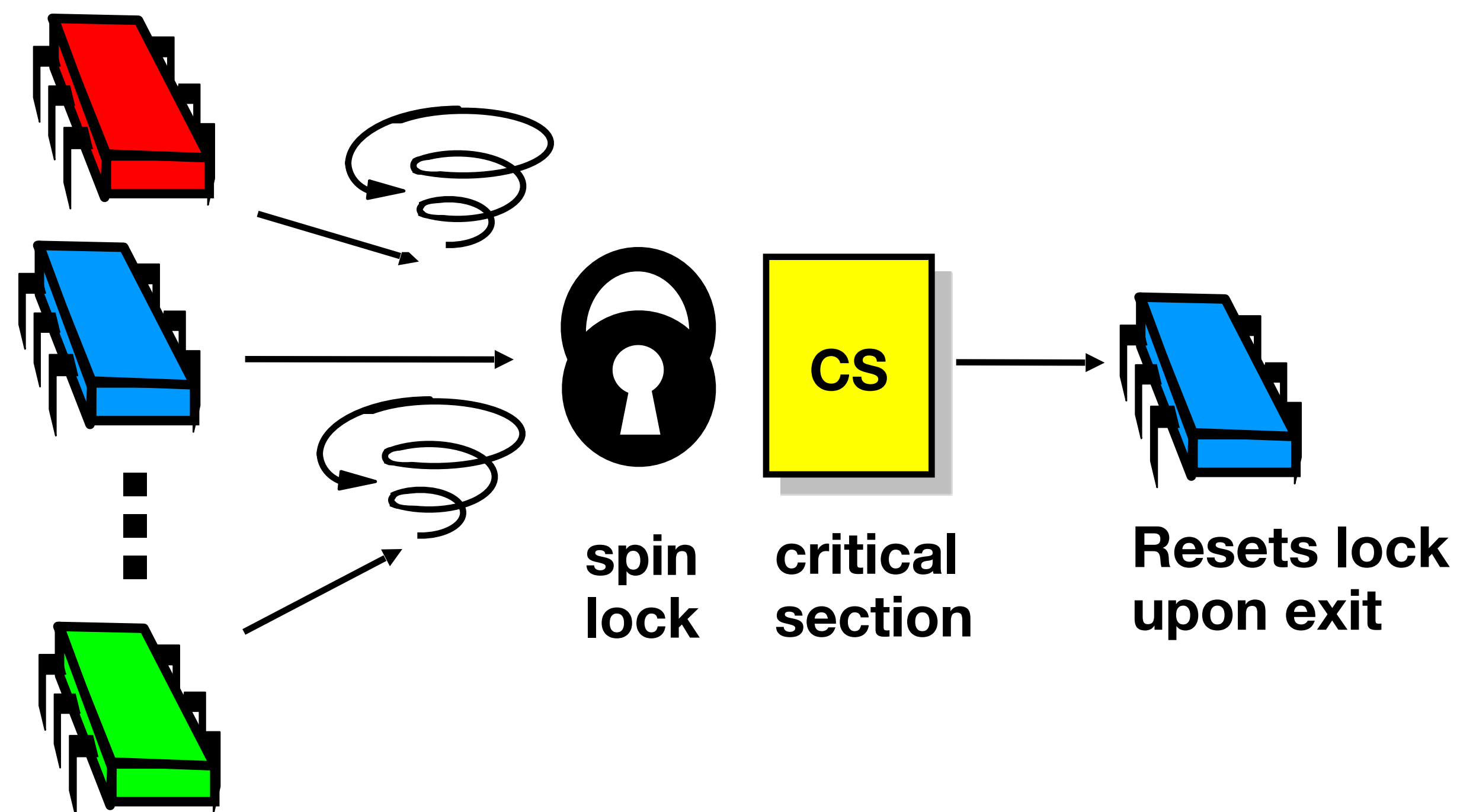
What should you do if you can't get a lock?

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

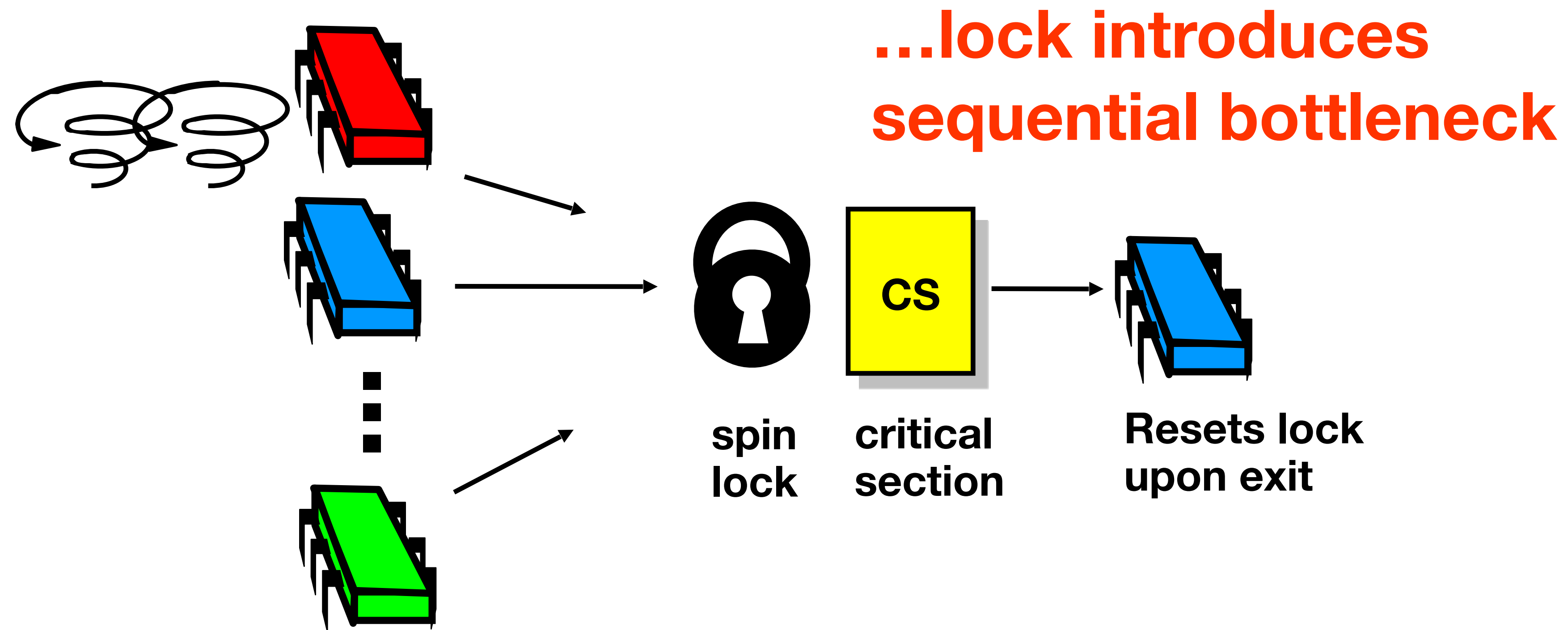


Our Focus

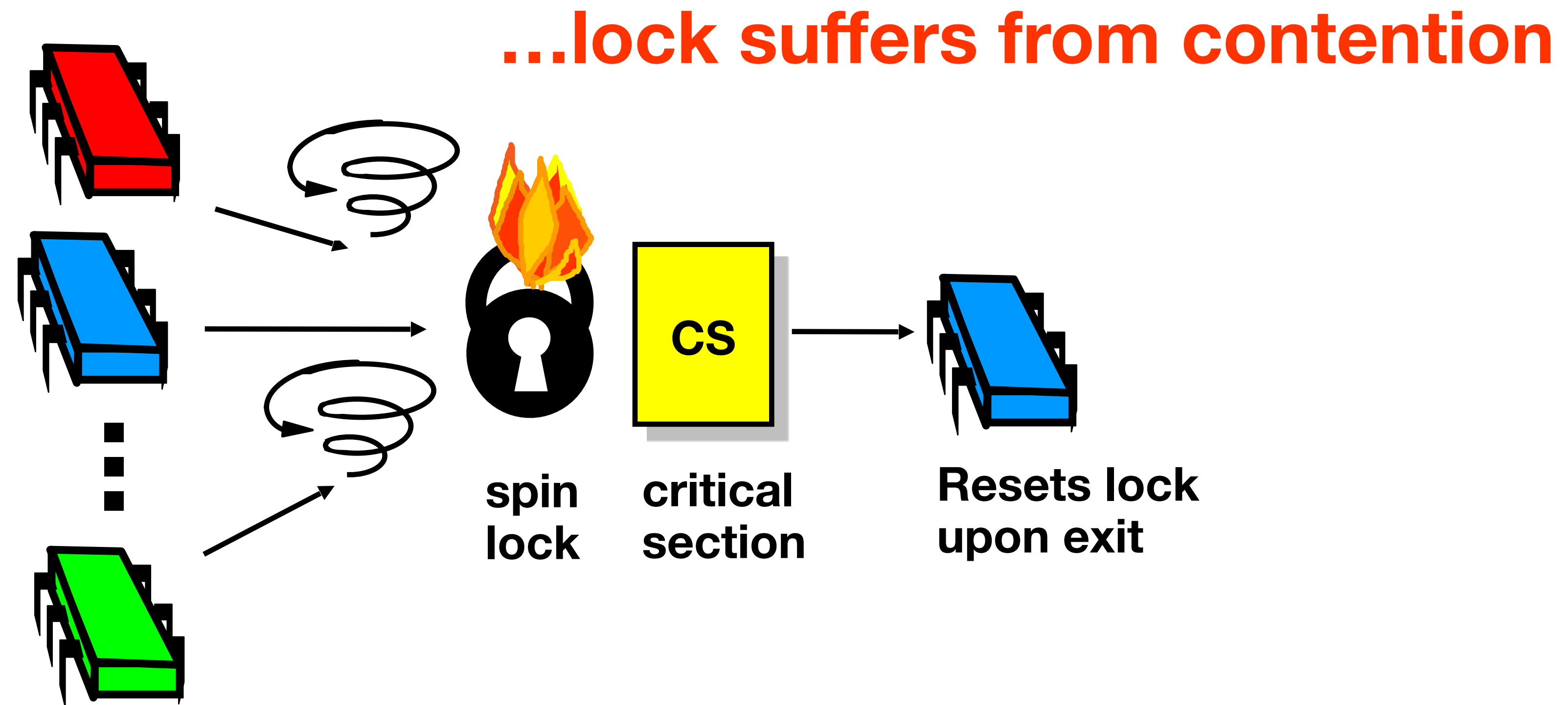
Basic Spin Lock



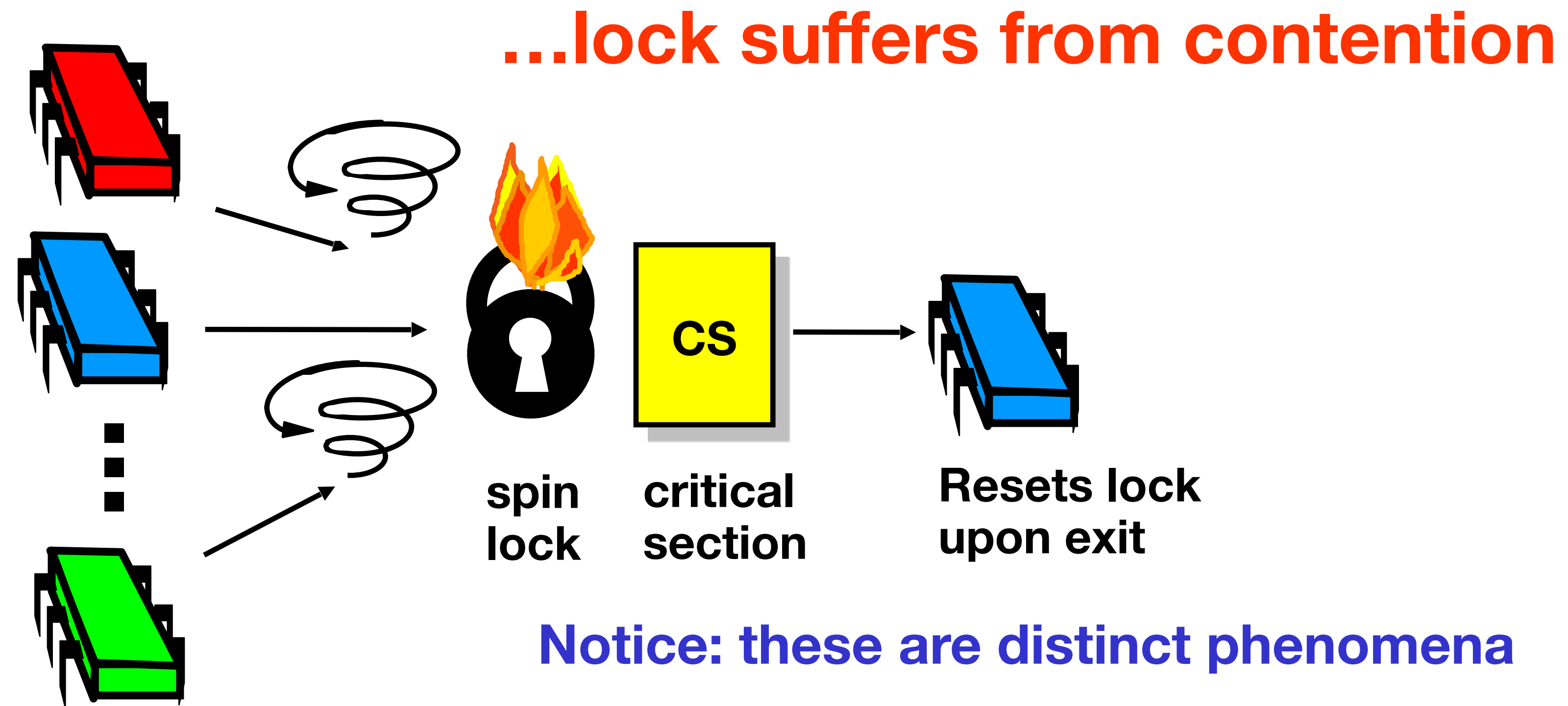
Basic Spin Lock



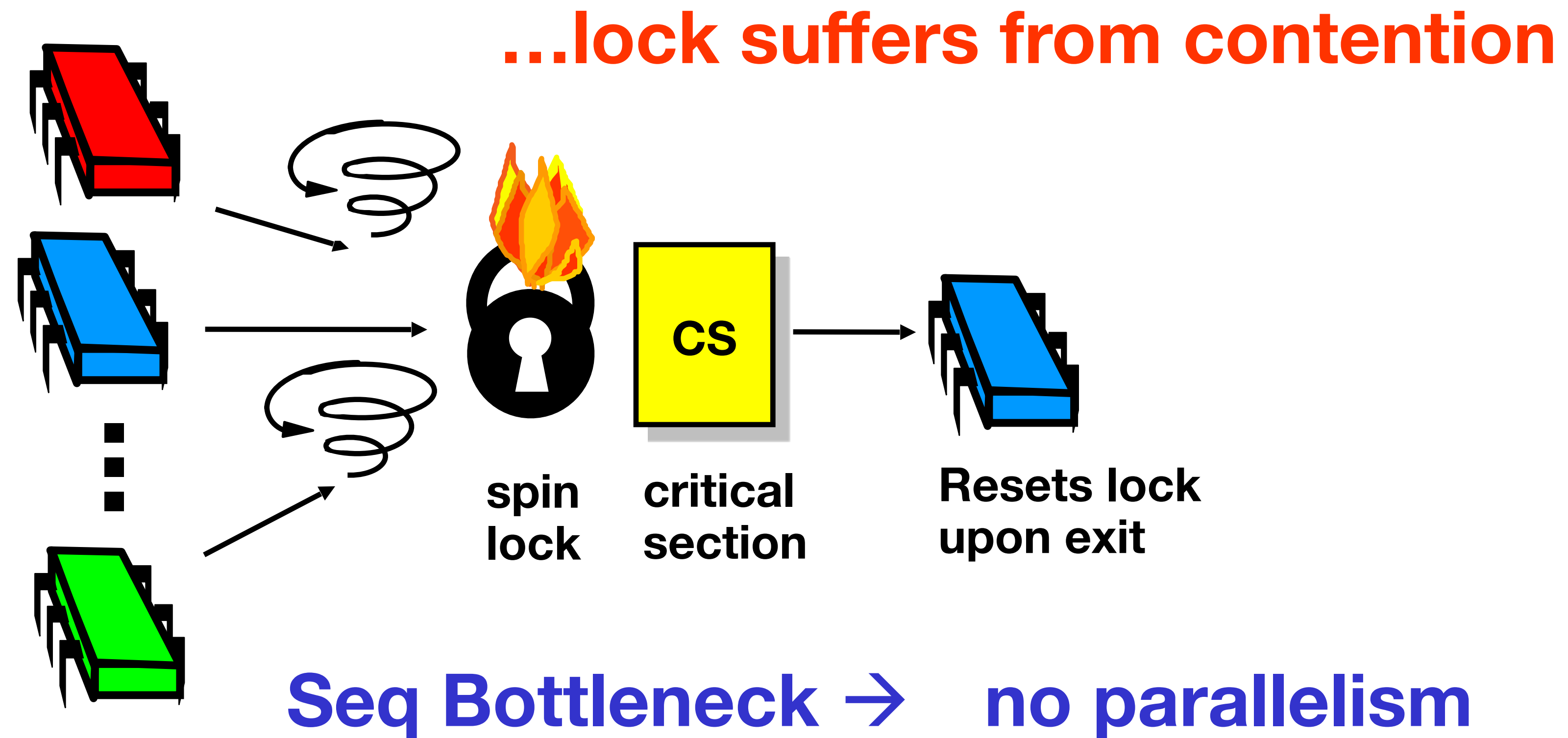
Basic Spin Lock



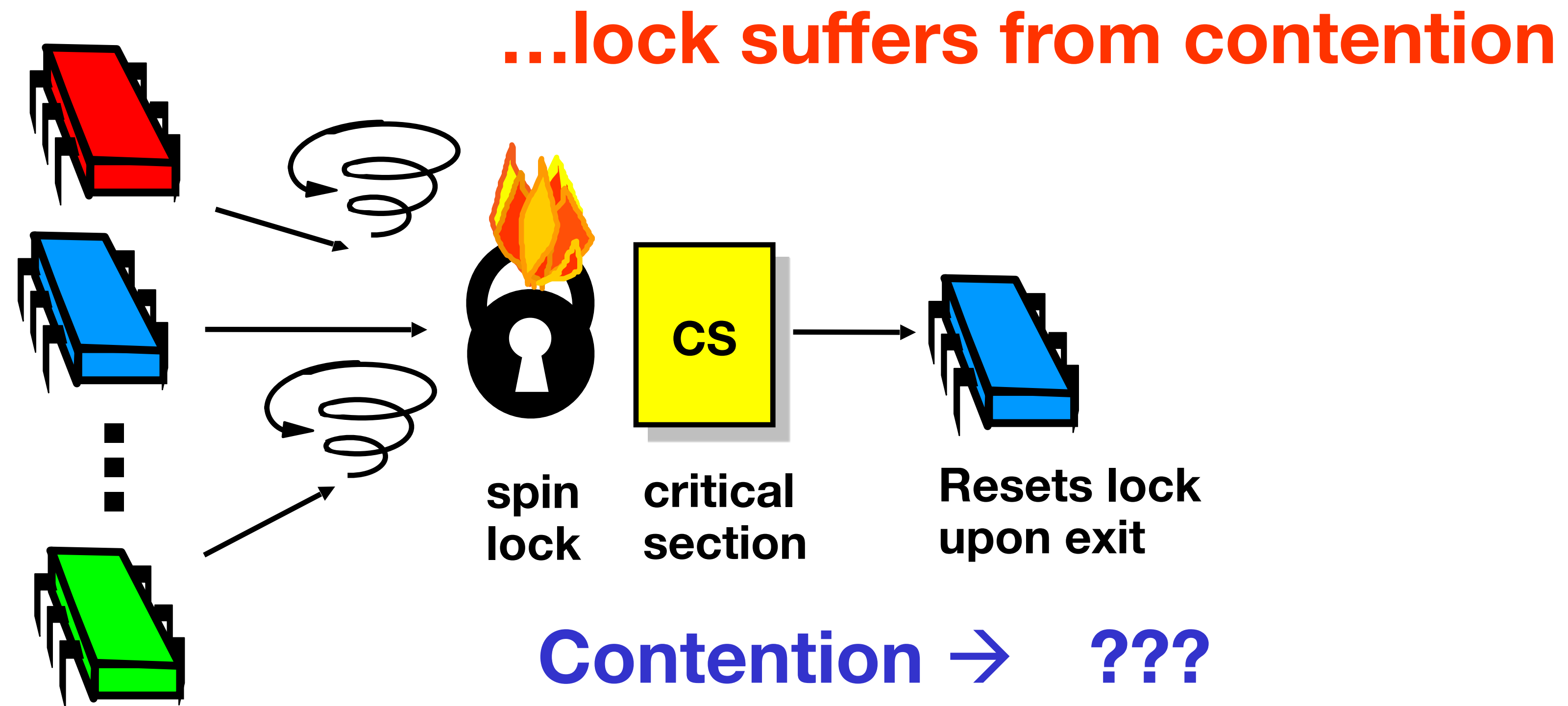
Basic Spin Lock



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 “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
```

Demo

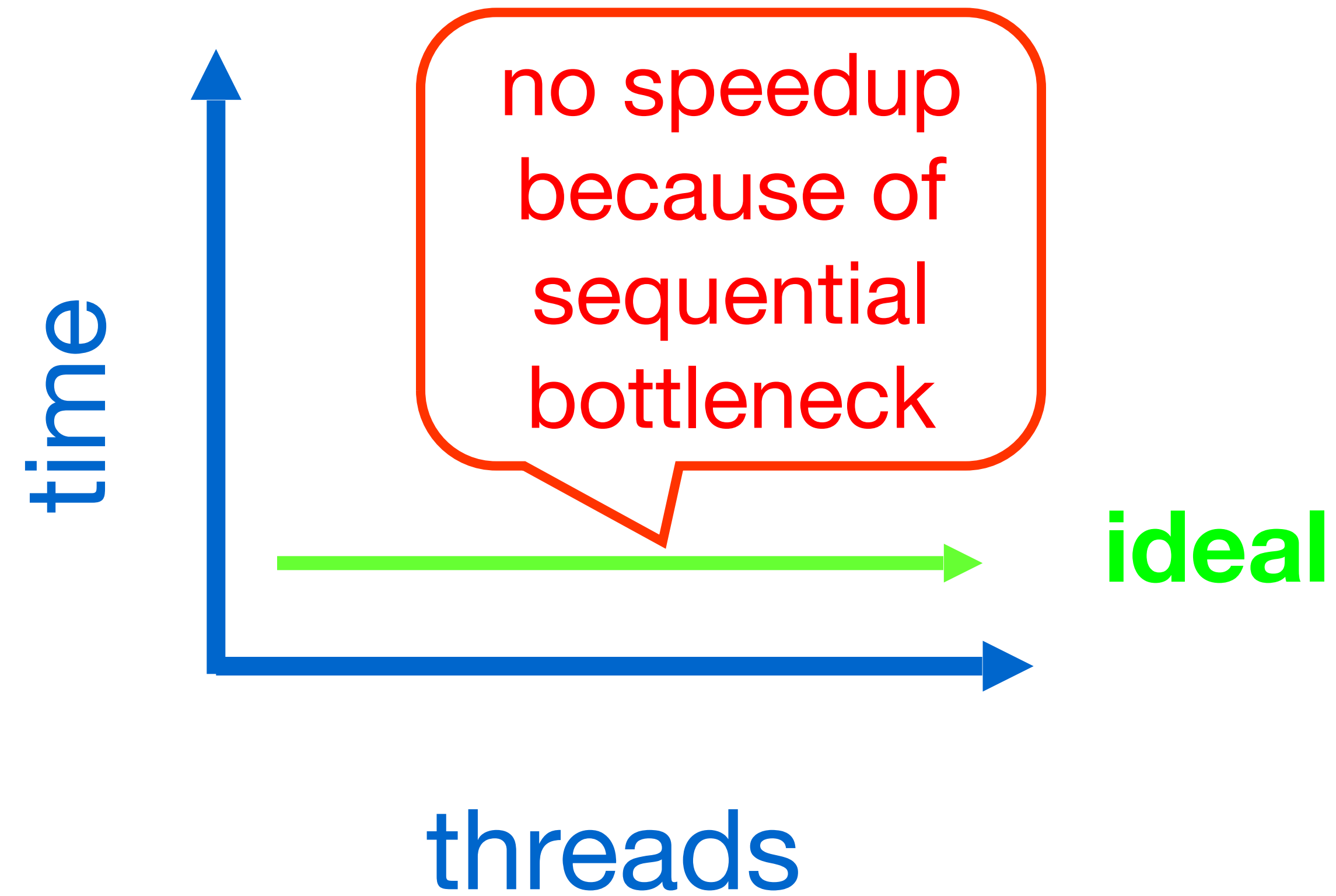
Space Complexity

- 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 **$\Omega(n)$** lower bound?
- We used an RMW operation...

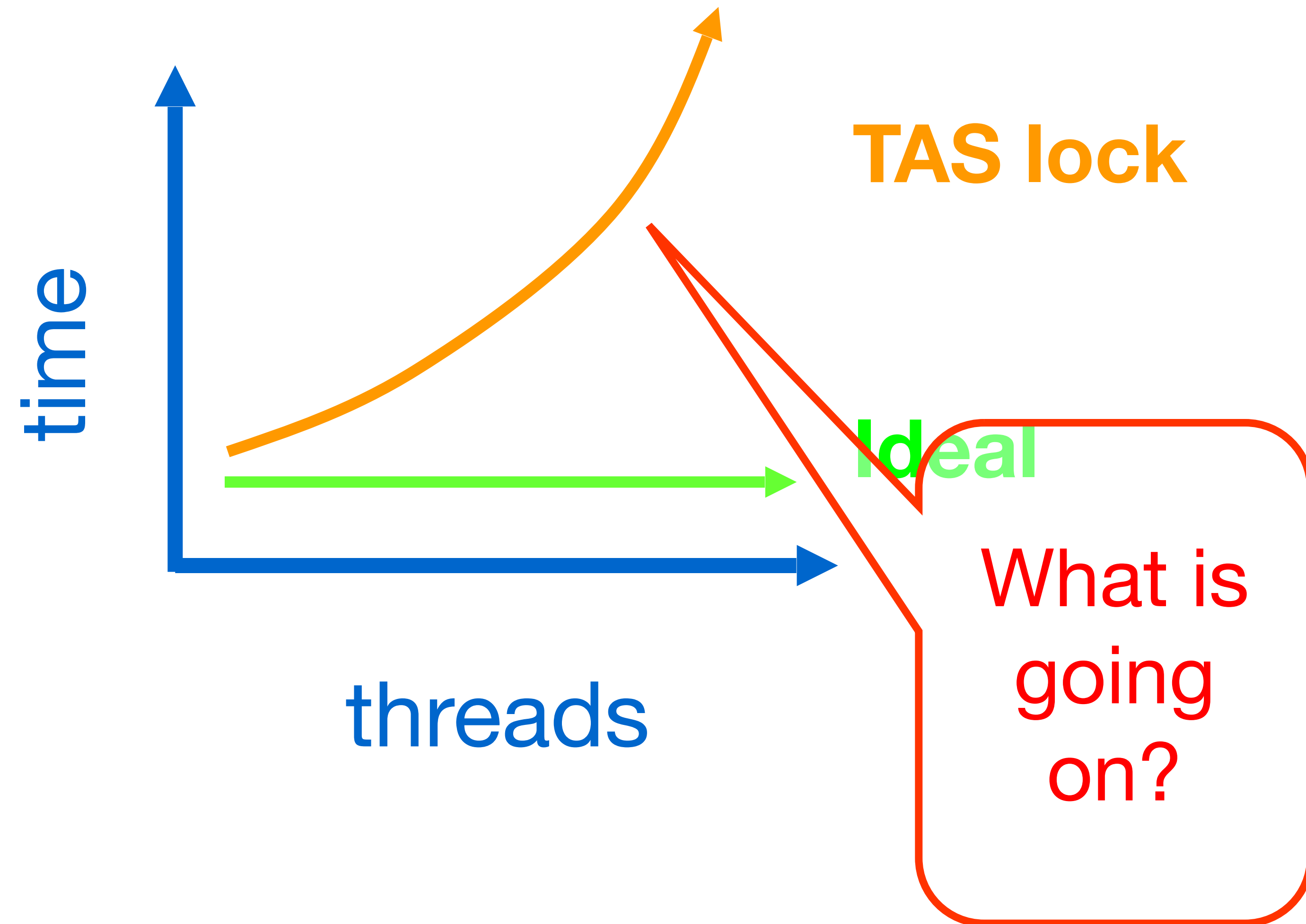
Space Complexity

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

Time taken



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

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

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

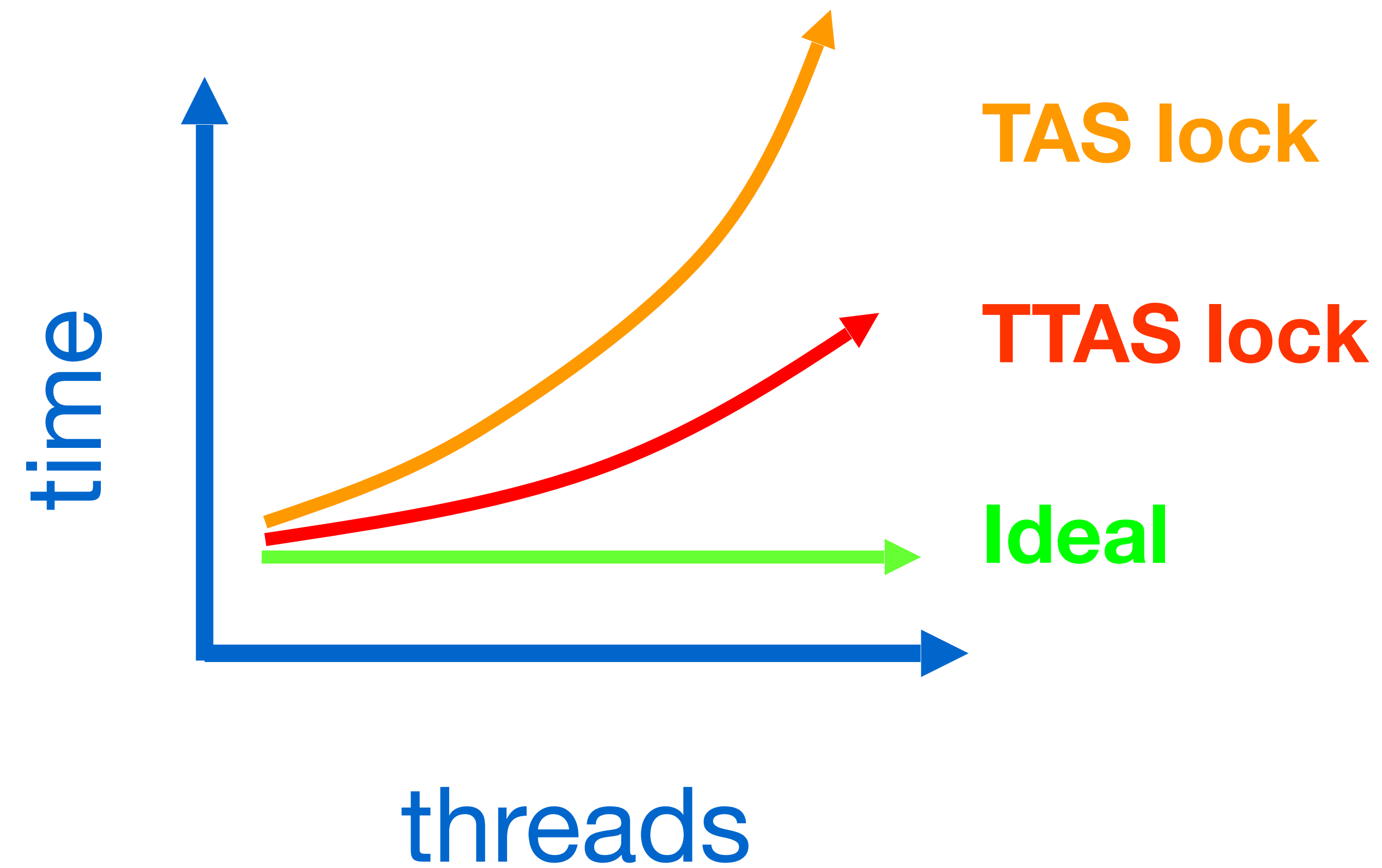
  let unlock t = Atomic.set t.state false

  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



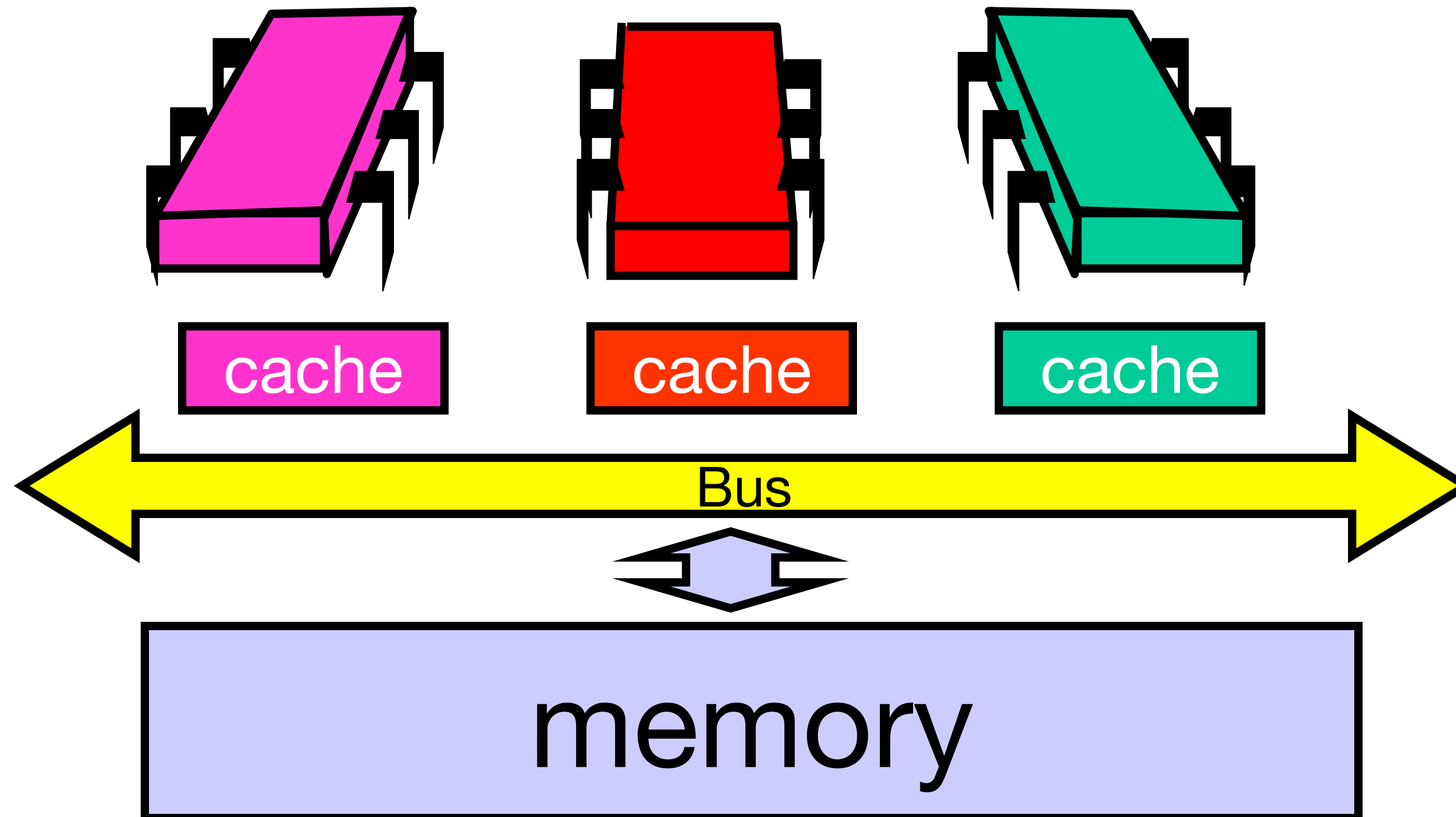
Mystery

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

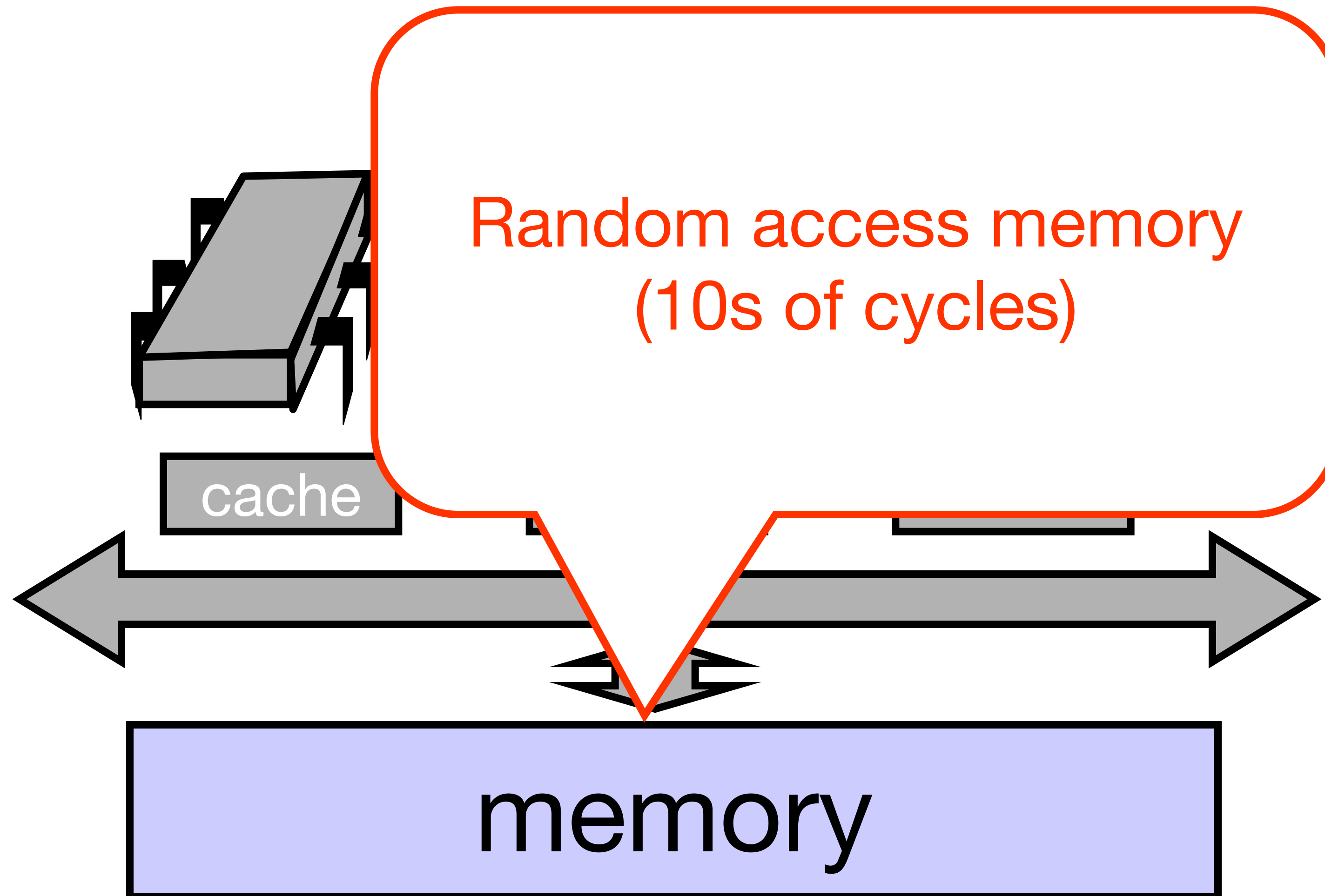
Opinion

- 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



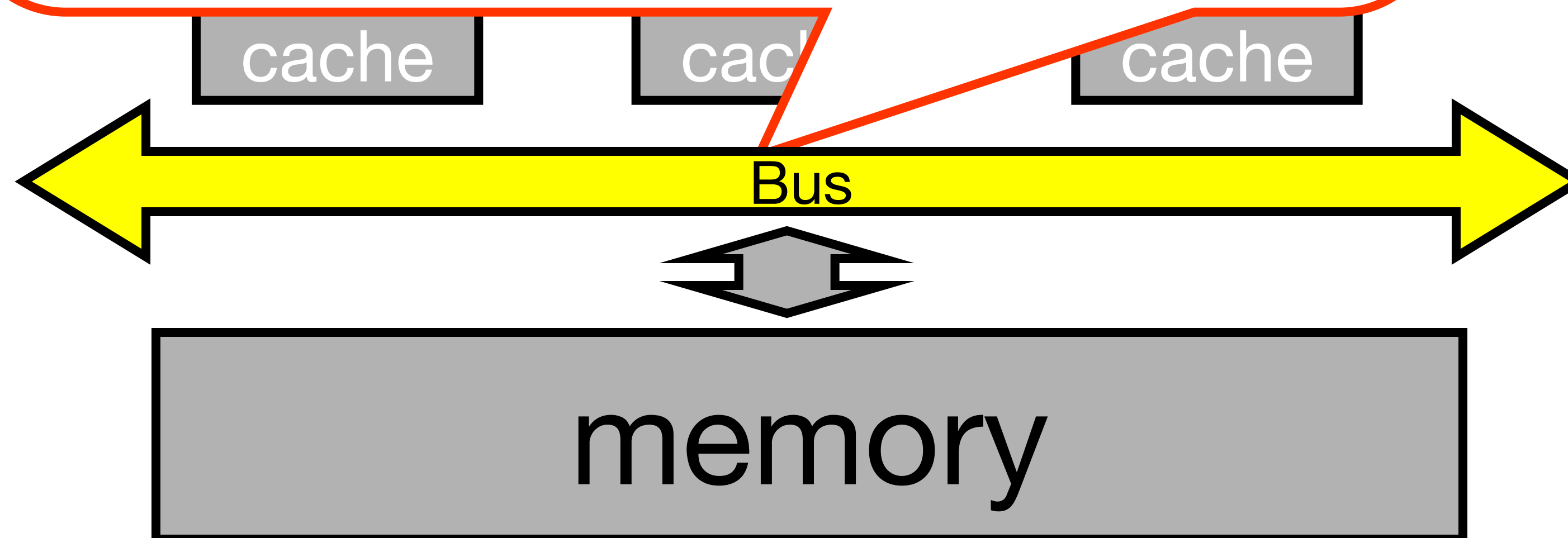
Bus-based Architectures



Bus-based Architectures

Shared Bus

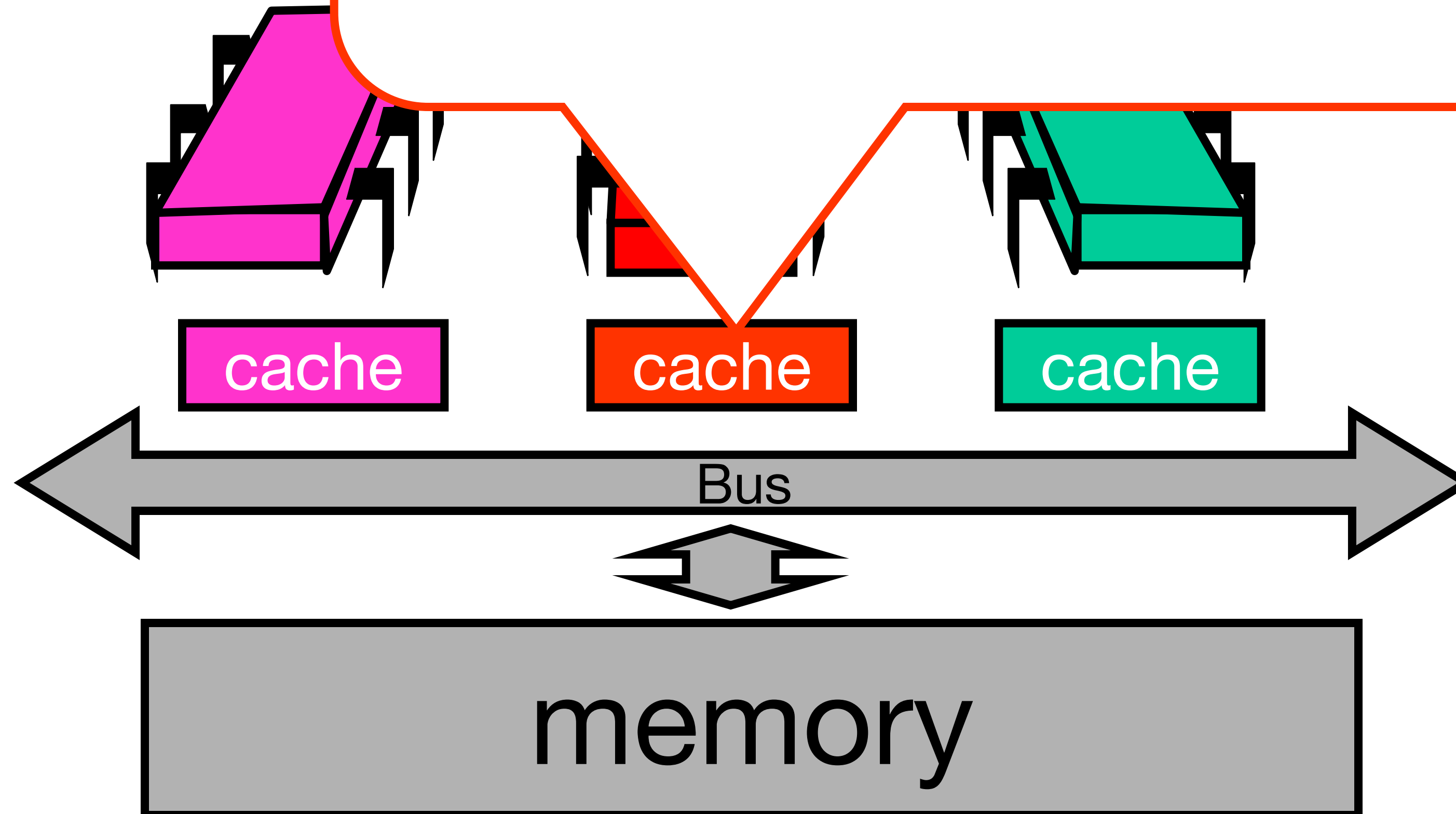
- Broadcast medium
- One broadcaster at a time
- Processors and memory all “snoop”



Bus-based Arch

Per-Processor Caches

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



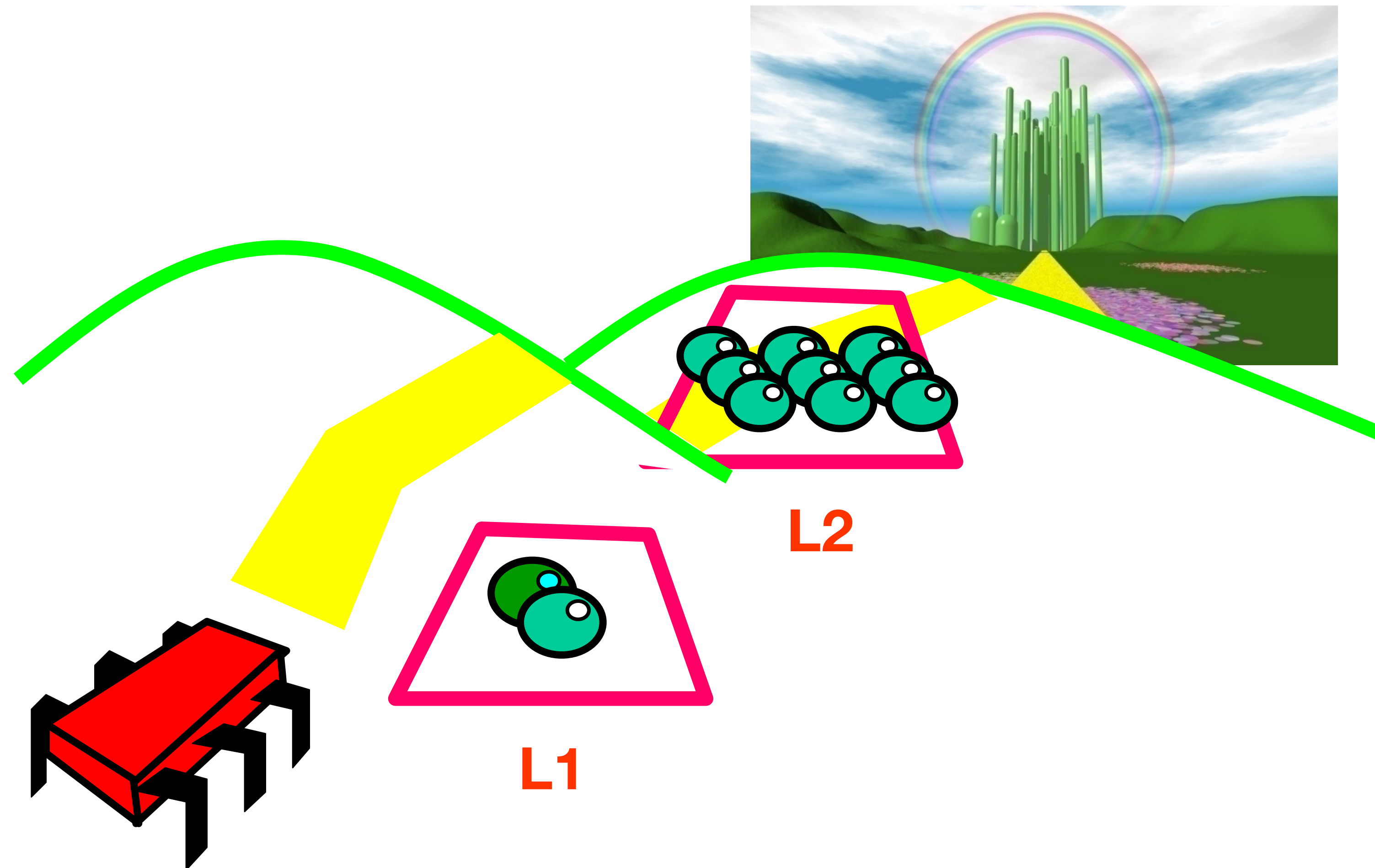
Granularity

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

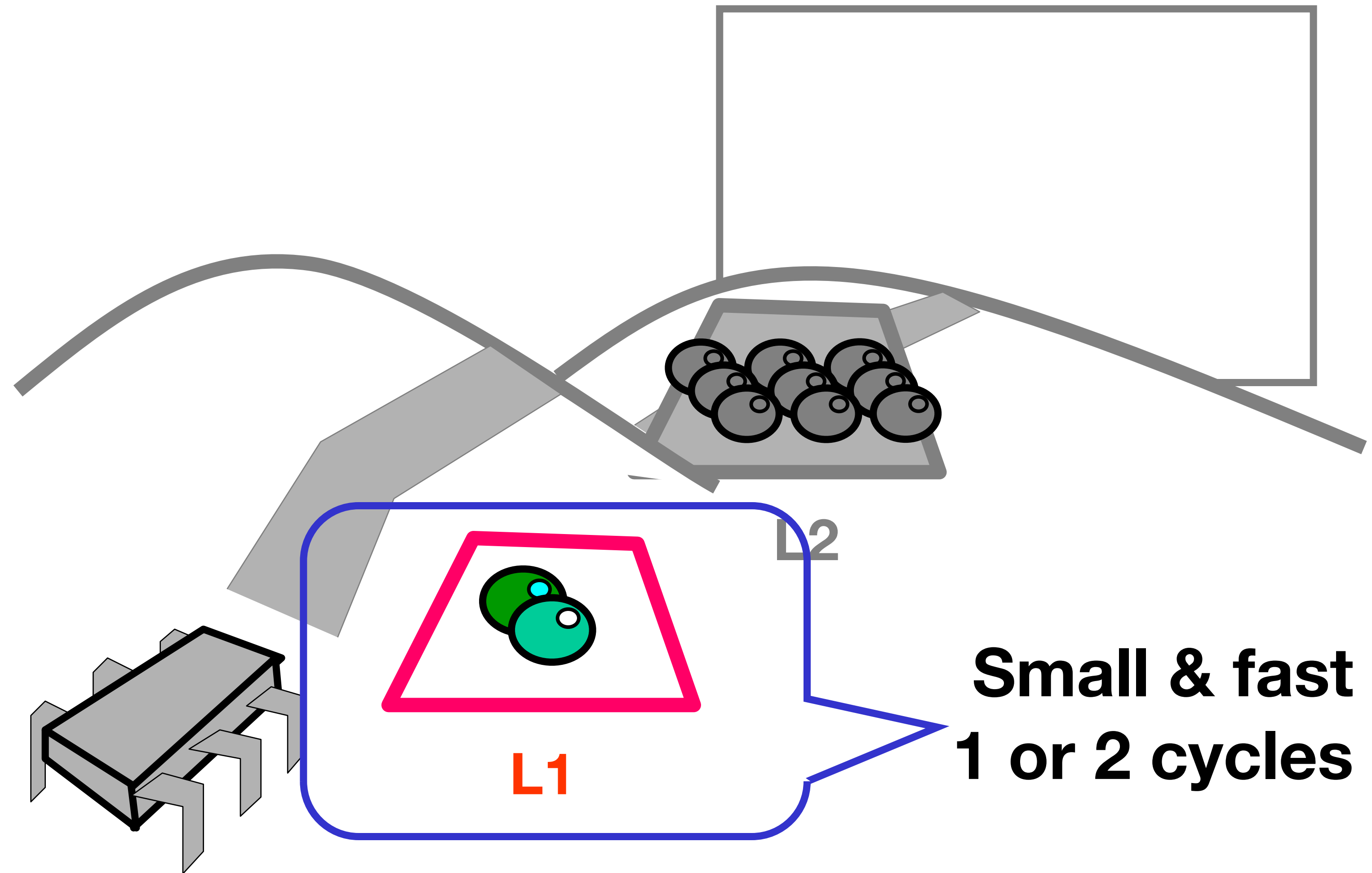
Locality

- 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

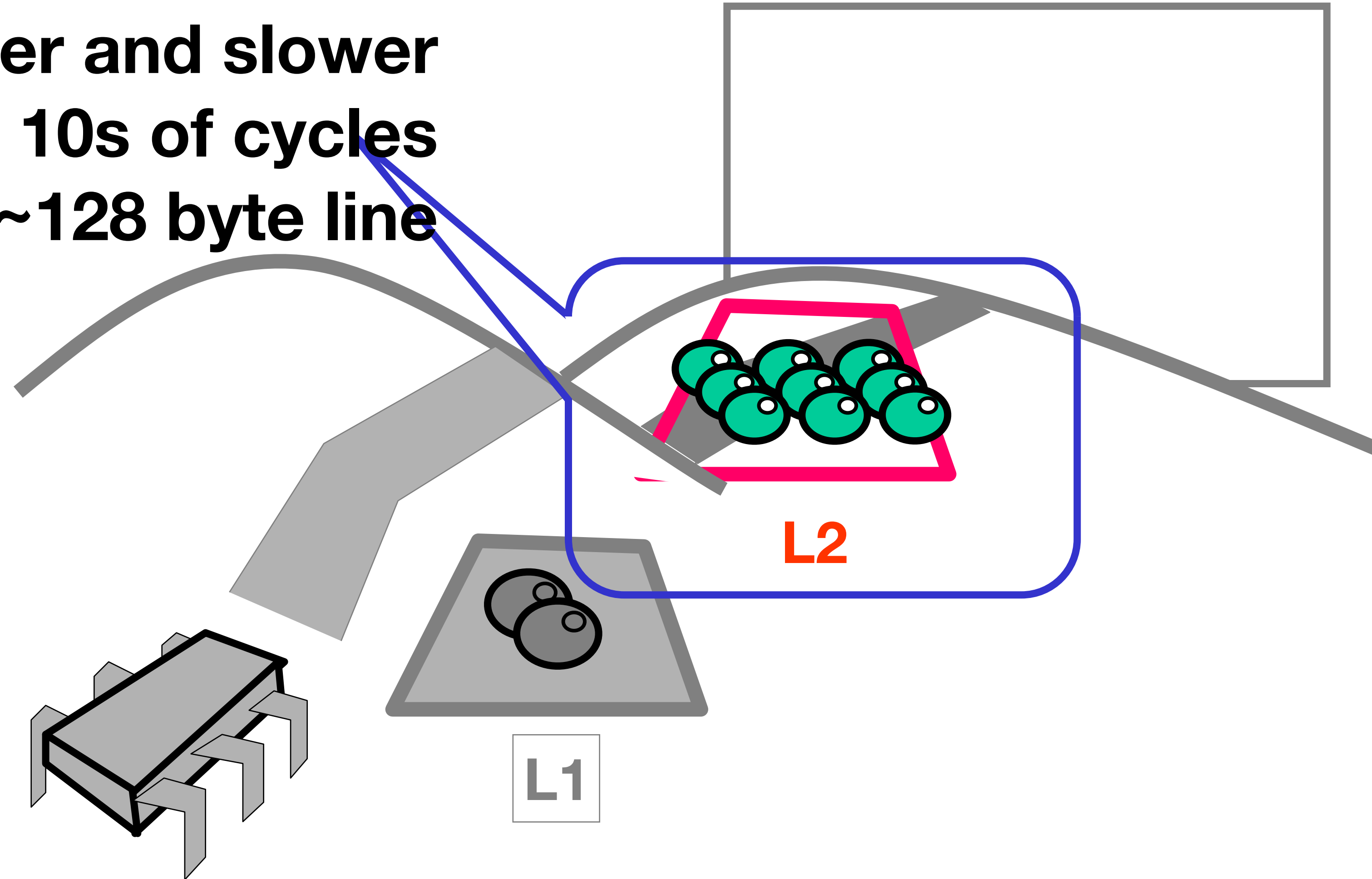


L1 and L2 caches



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

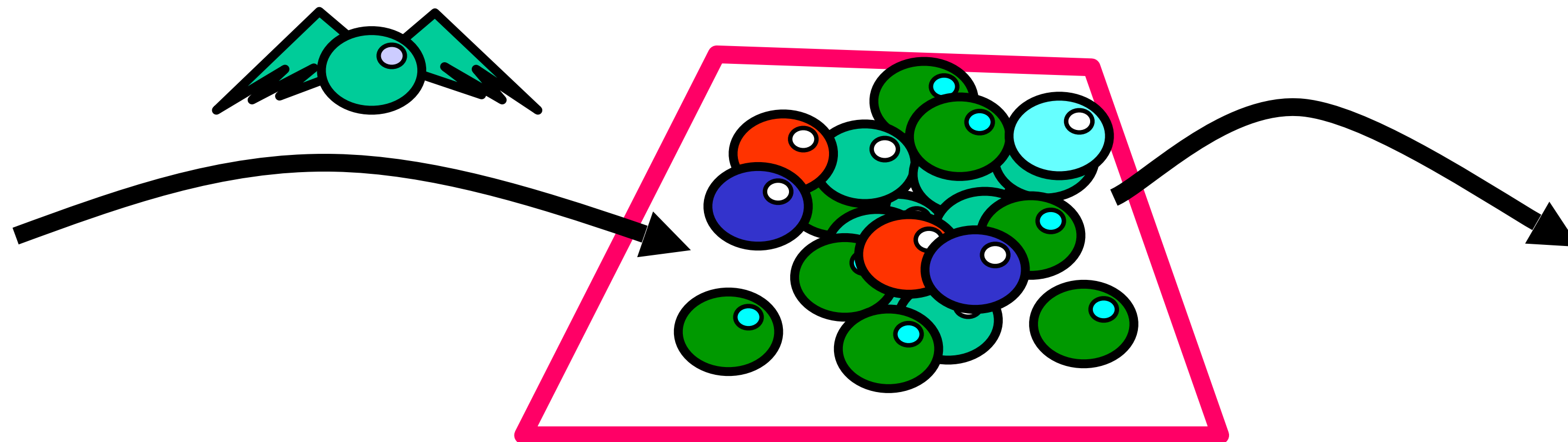
- This model is ***still*** a simplification
 - But not in any essential way
 - Illustrates basic principles
- Will discuss complexities later

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

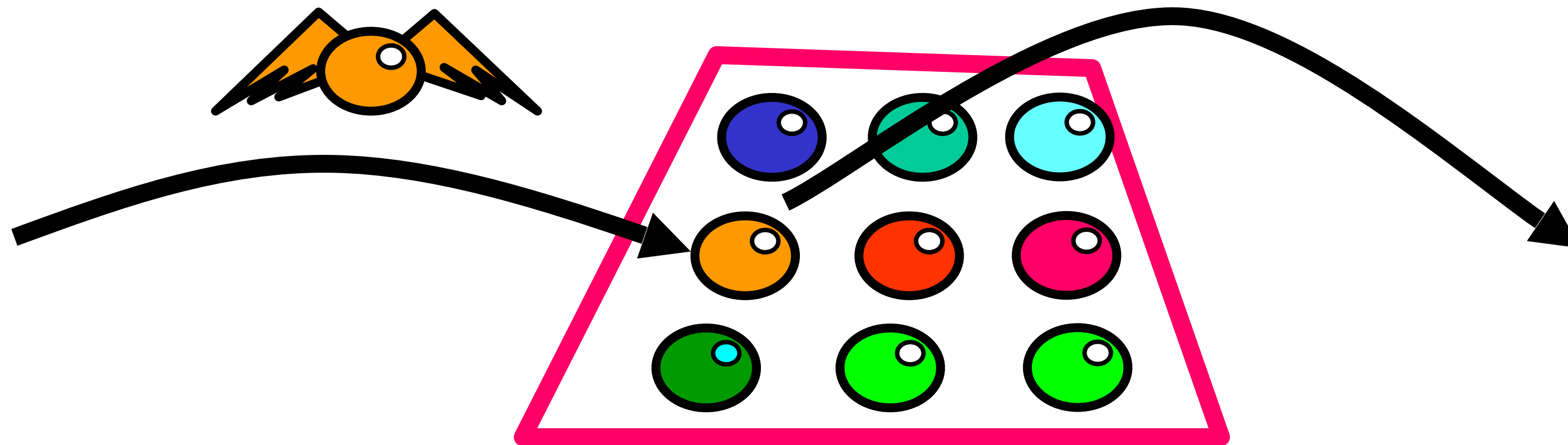
Fully Associative Cache

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



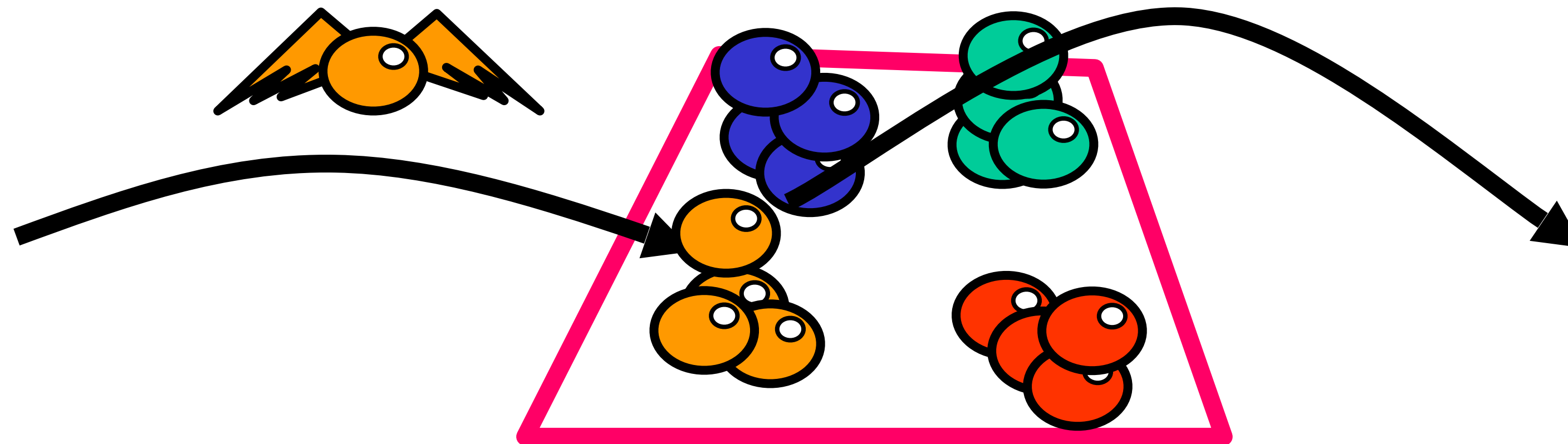
Direct Mapped Cache

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



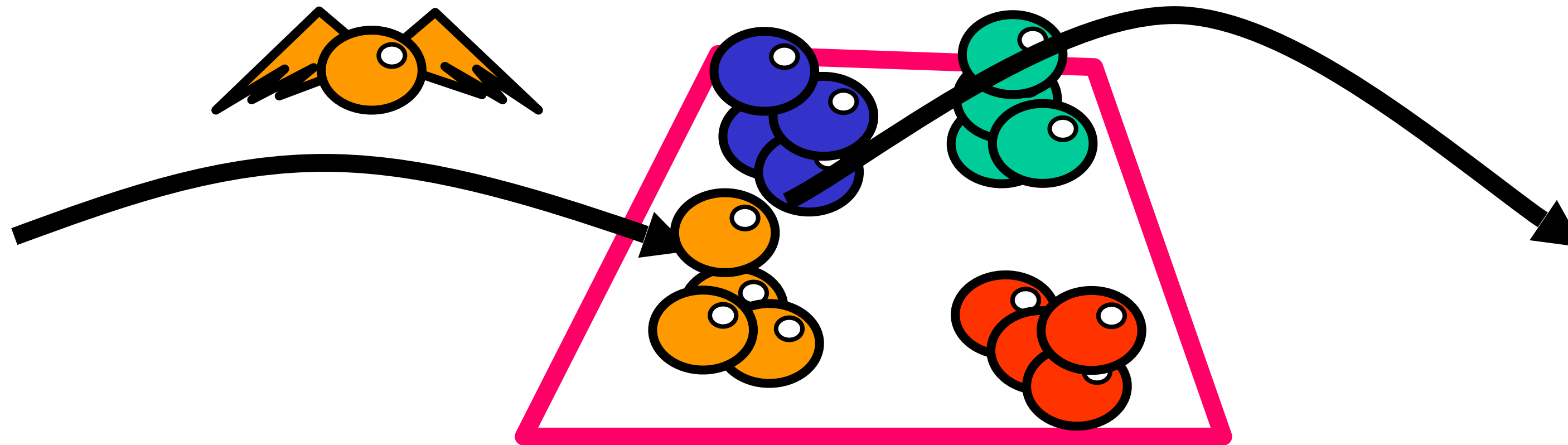
K-way set associative cache

- 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



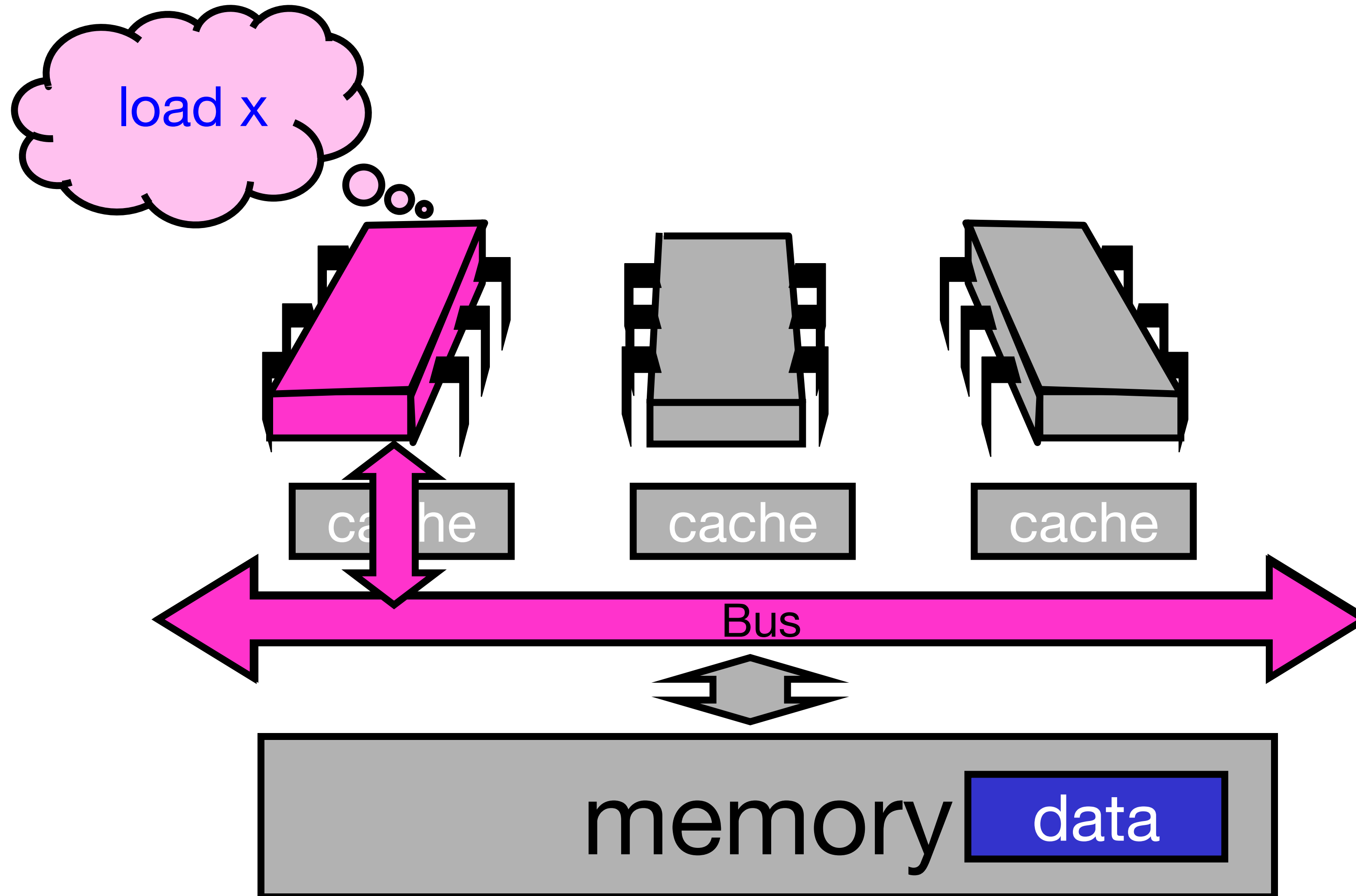
Cache Coherence

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

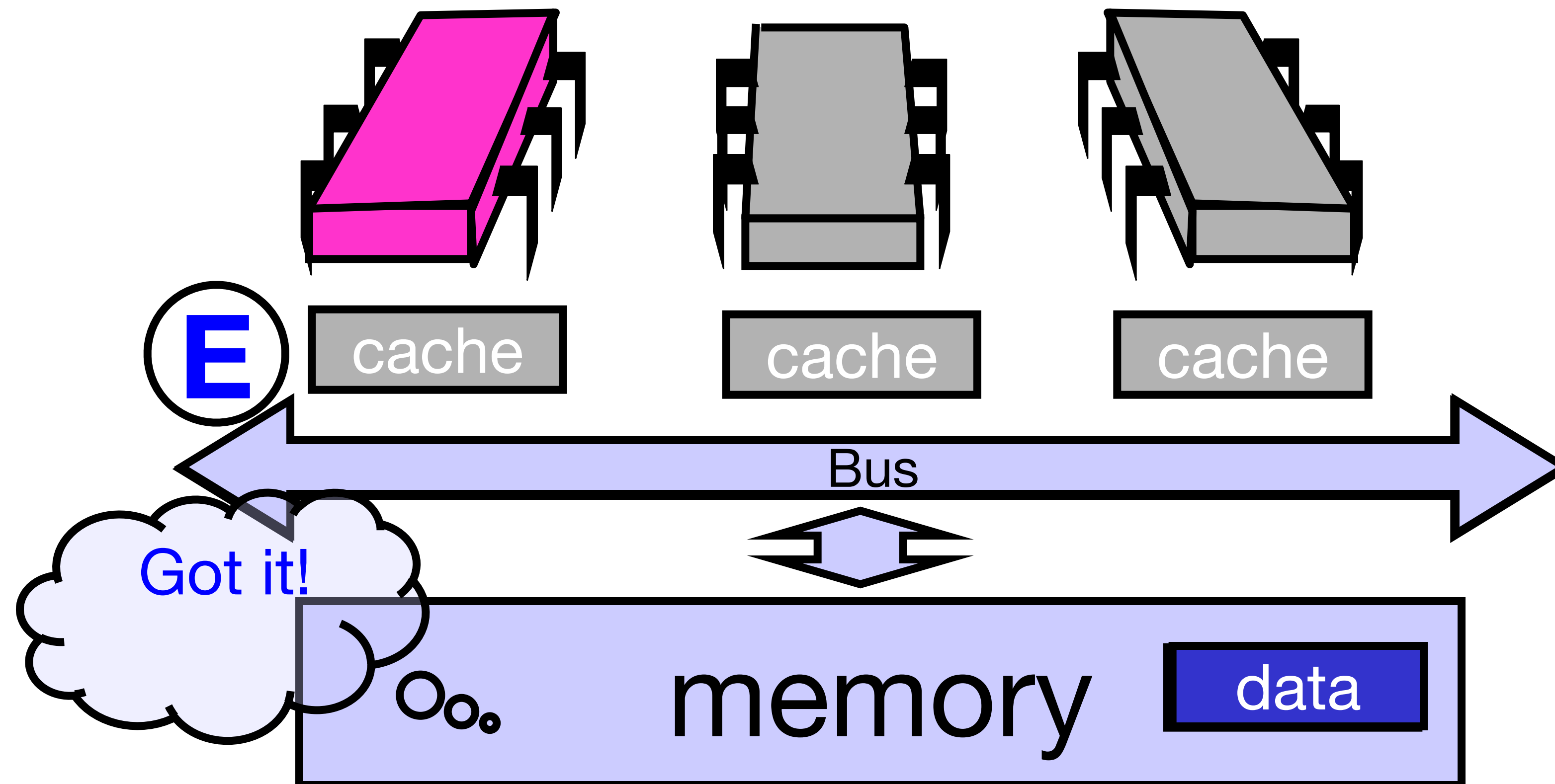
MESI

- **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

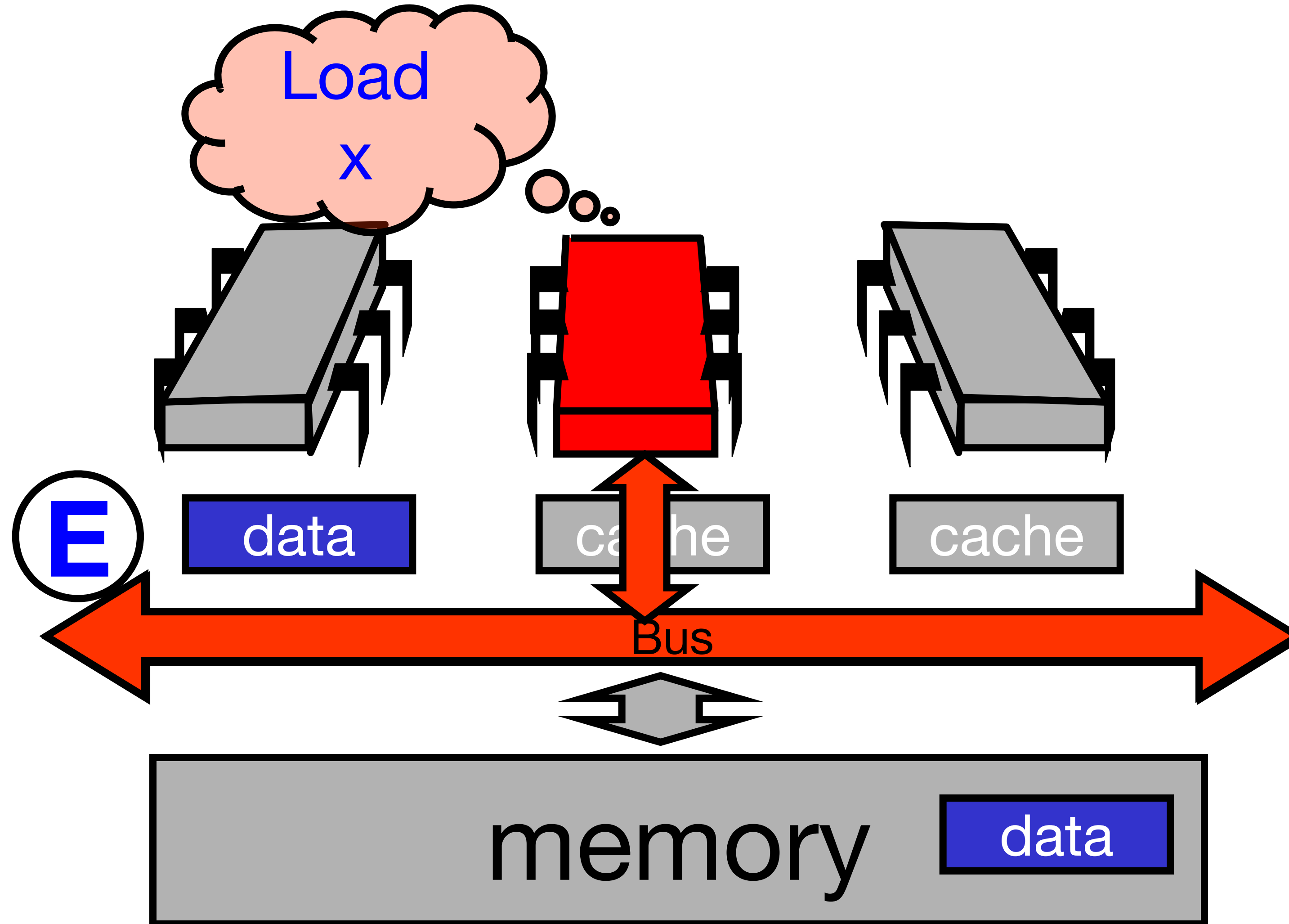
Processor Issues Load Request



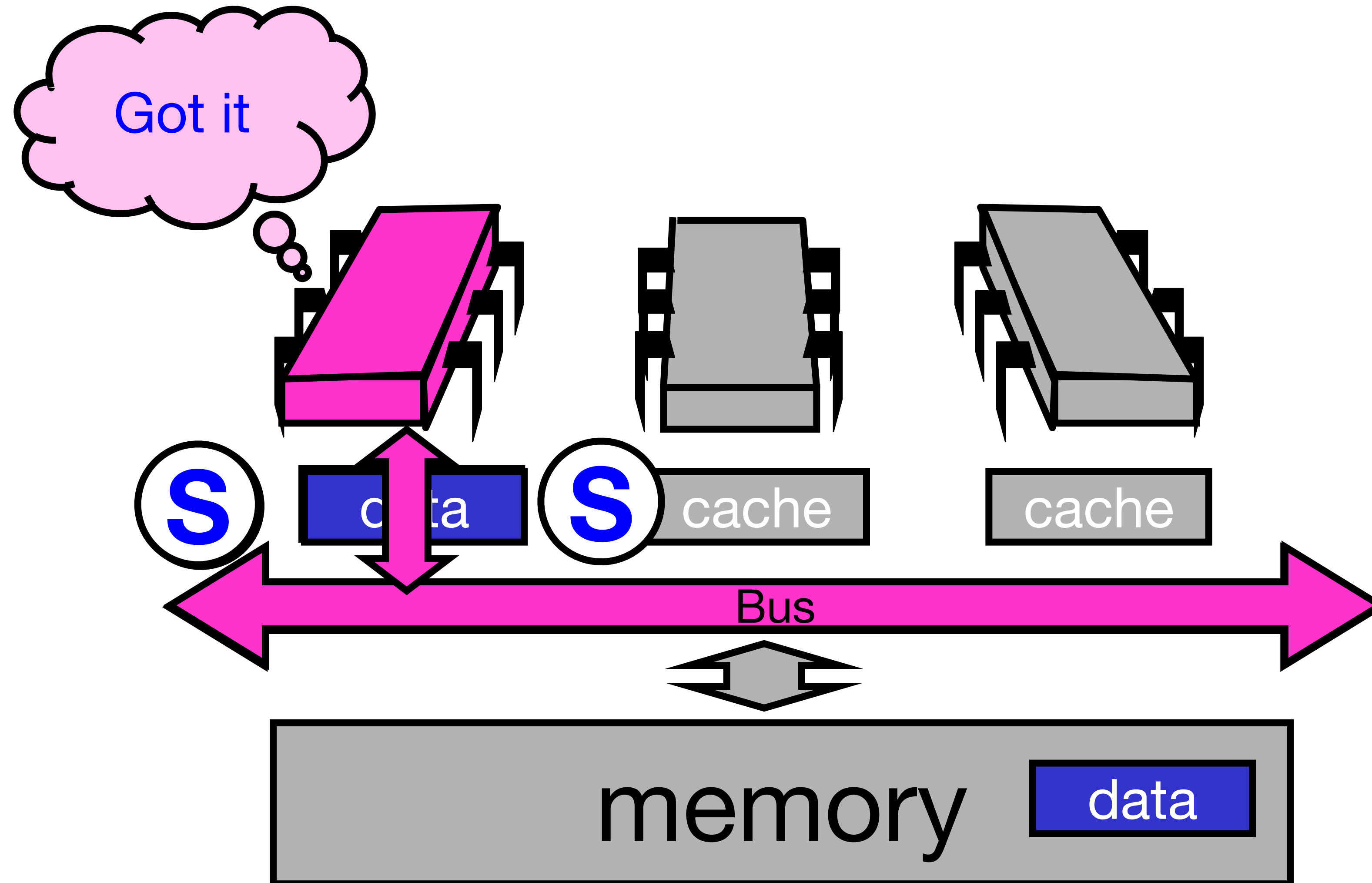
Memory Responds



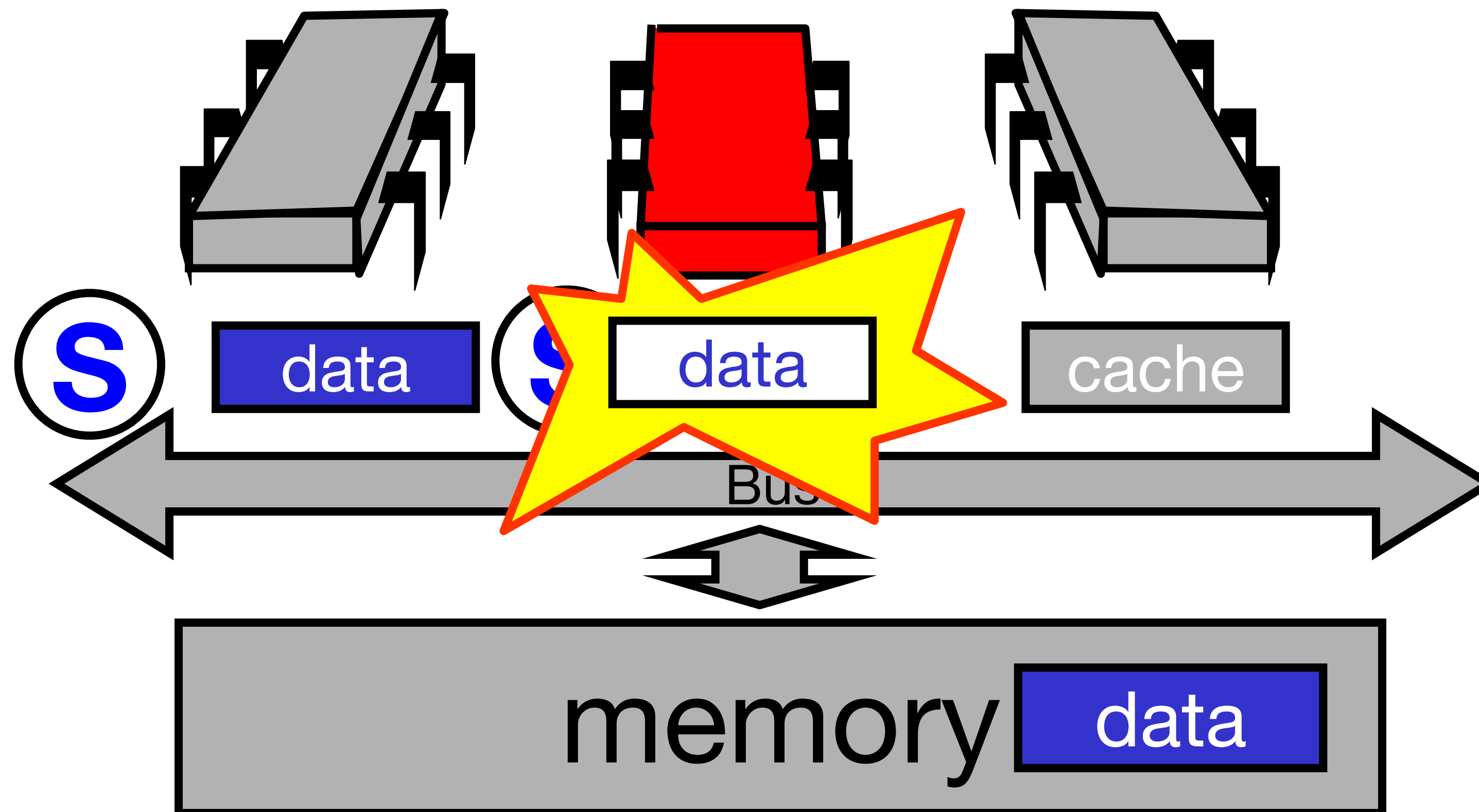
Processor Issues Load Request



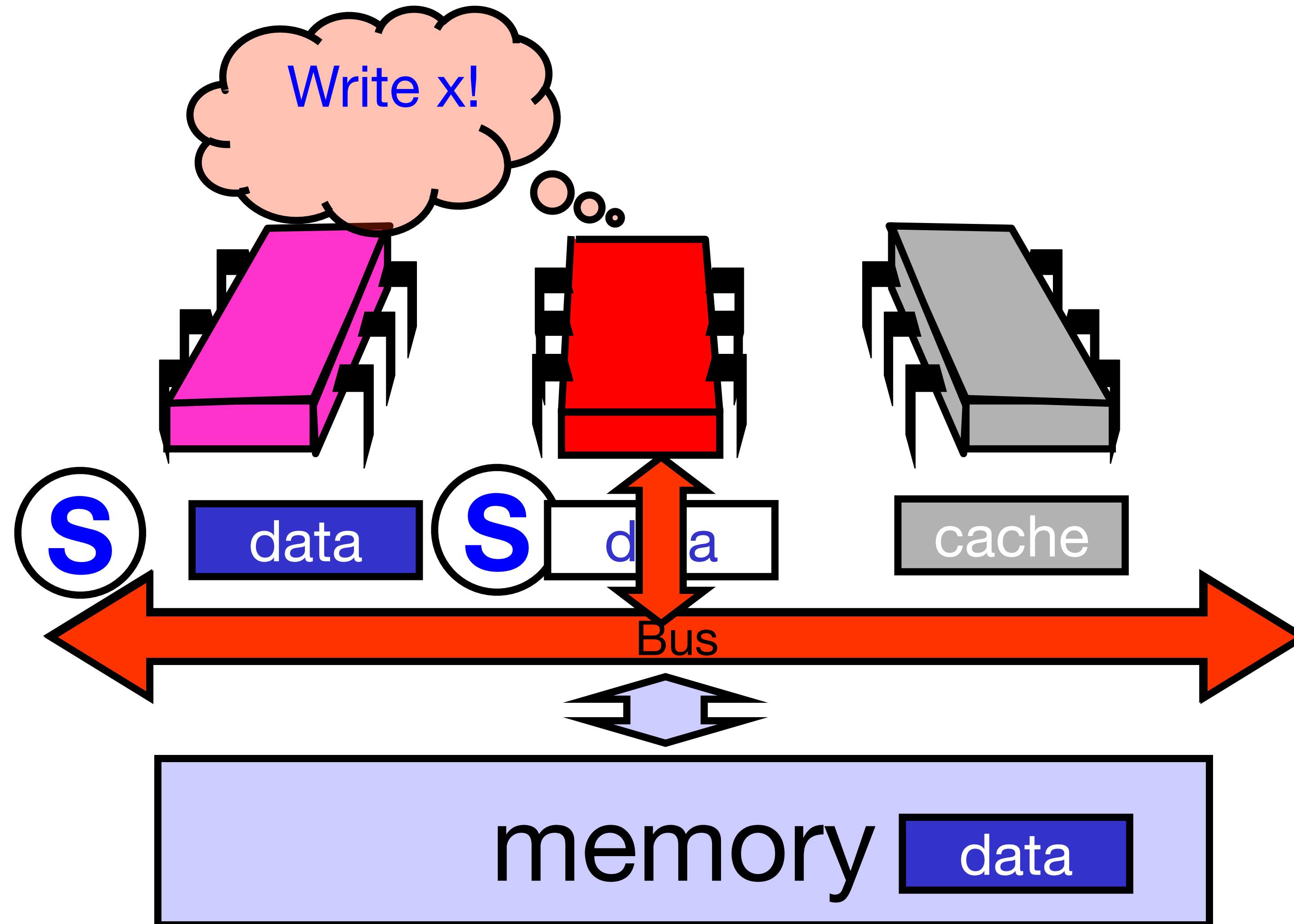
Other Processor Responds



Processor Issues Load Request



Write-through Cache



Write-through Cache

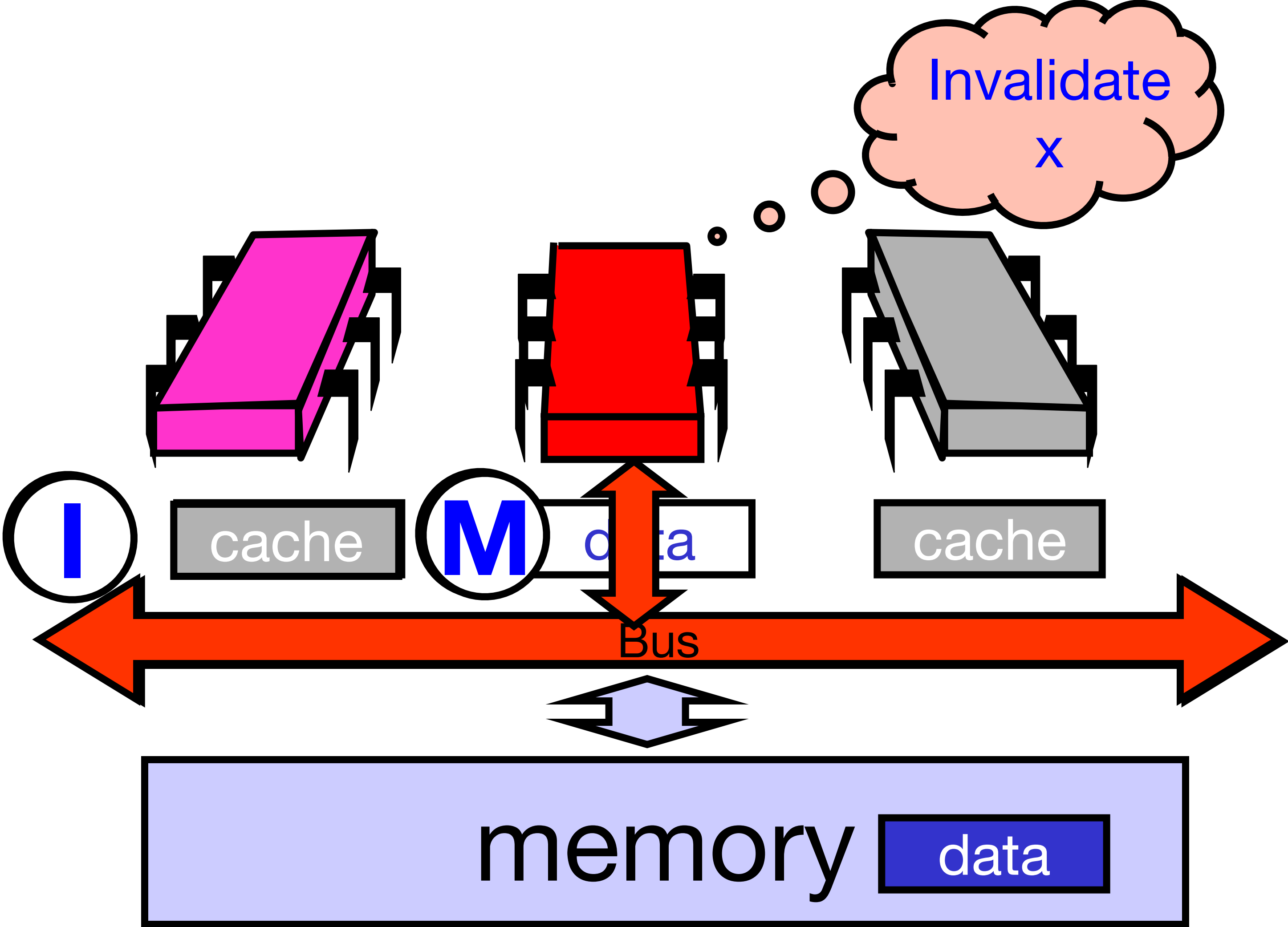
- Immediately broadcast changes
- Good
 - Memory, caches always agree
 - More read hits, maybe

“show stoppers”

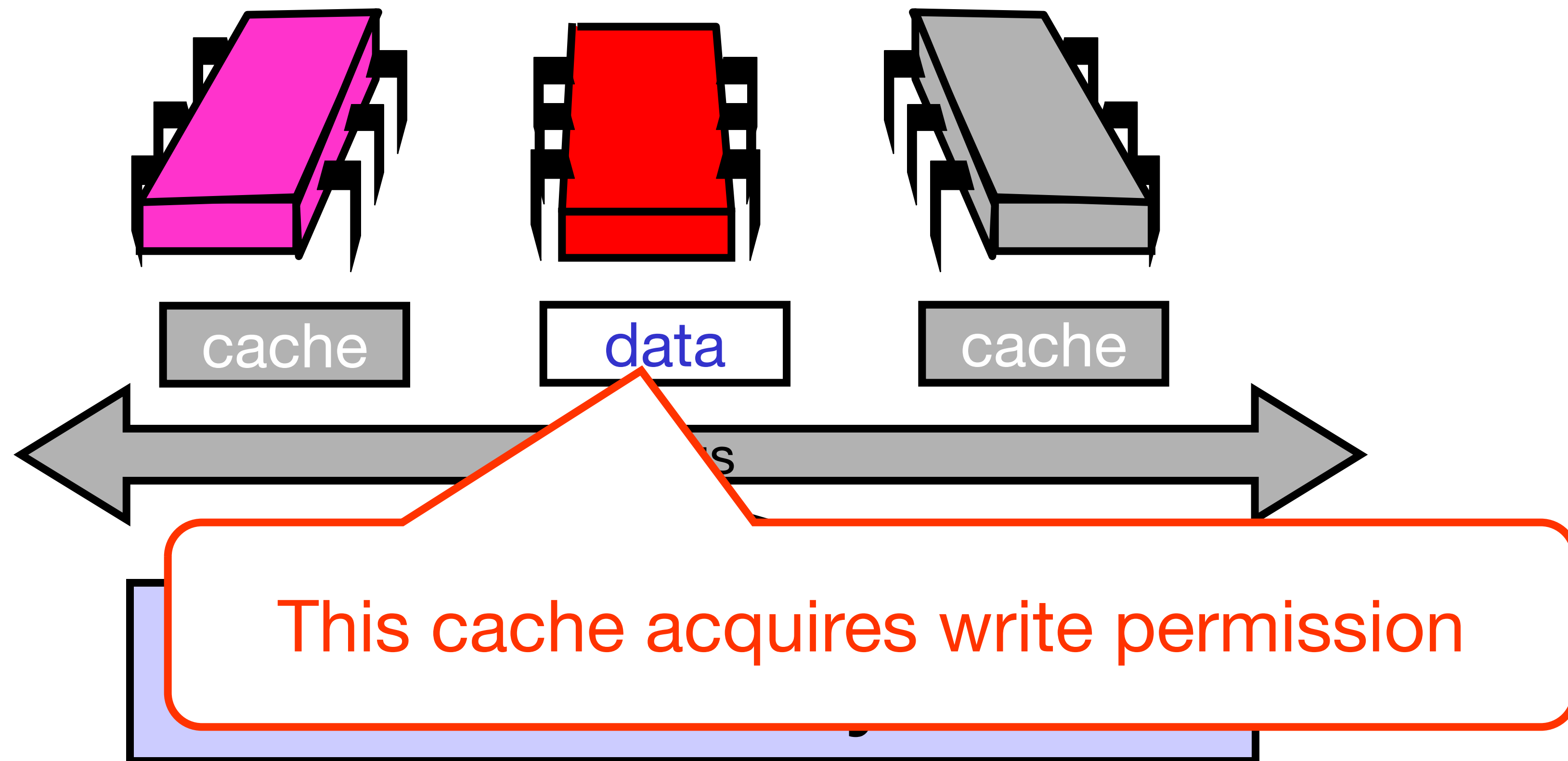


- Bad
 - Bus traffic on all writes
 - Most writes to unshared data
 - For example, loop indexes ...

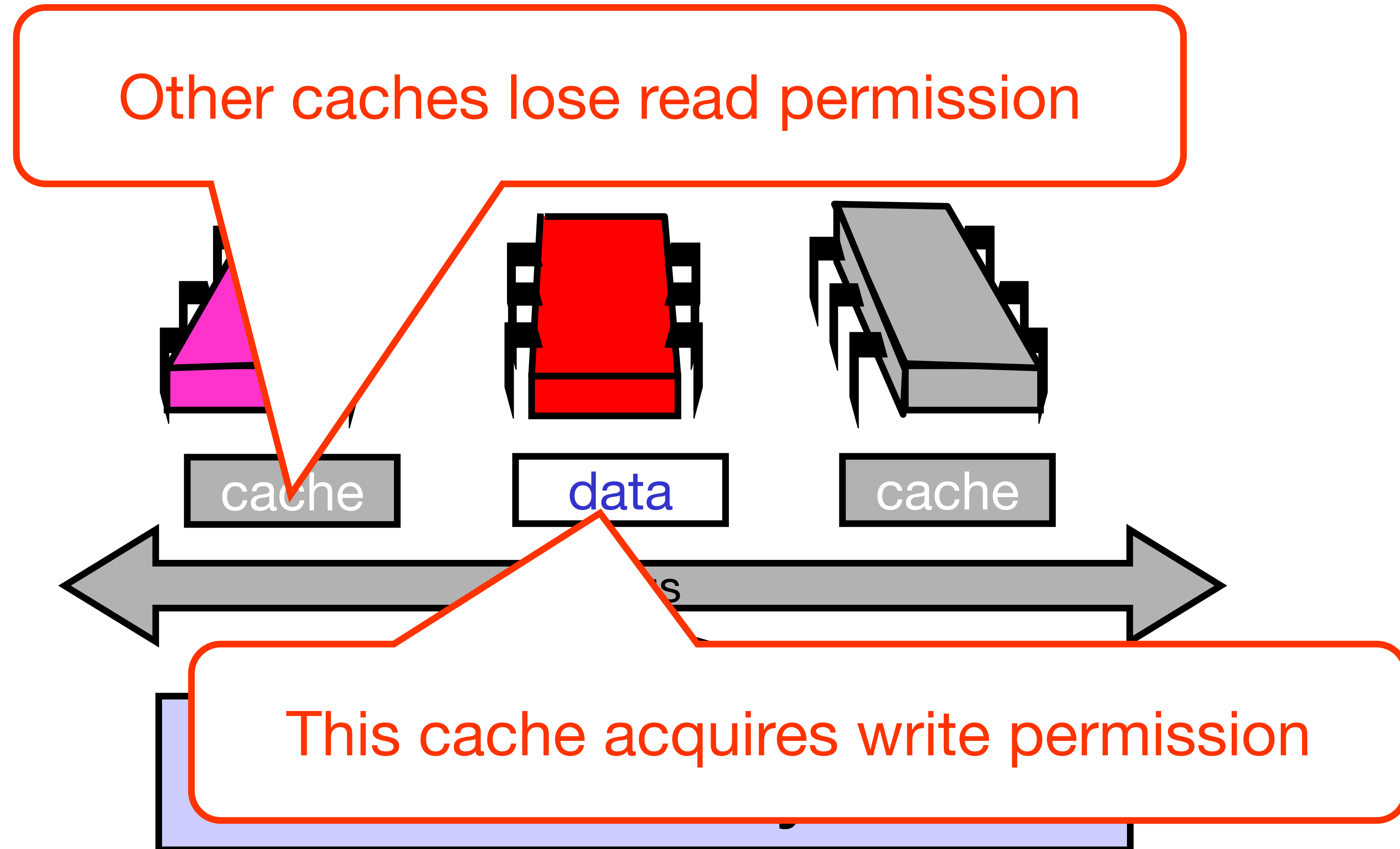
Invalidate



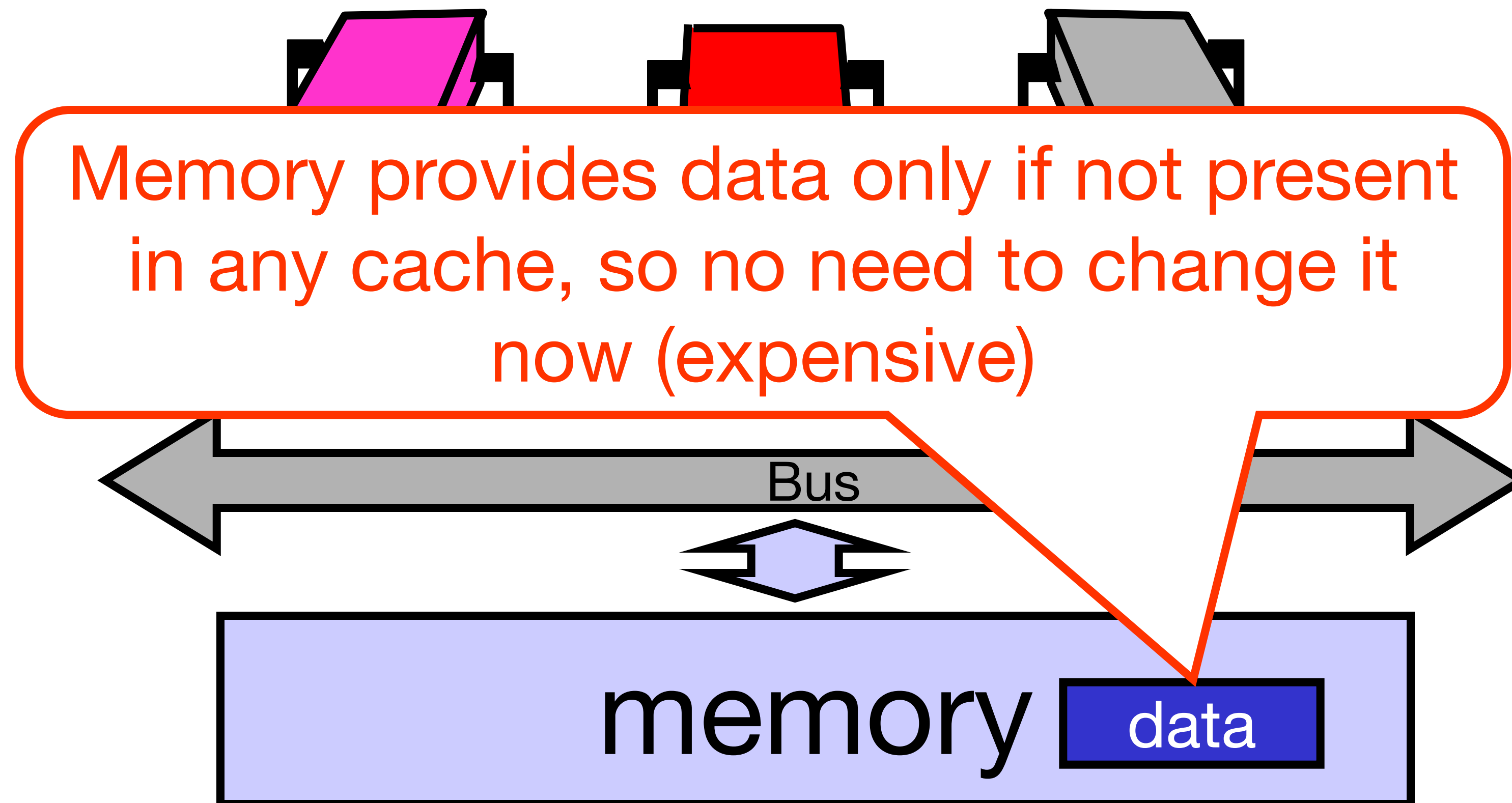
Invalidate



Invalidate



Invalidate



Mutual Exclusion

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

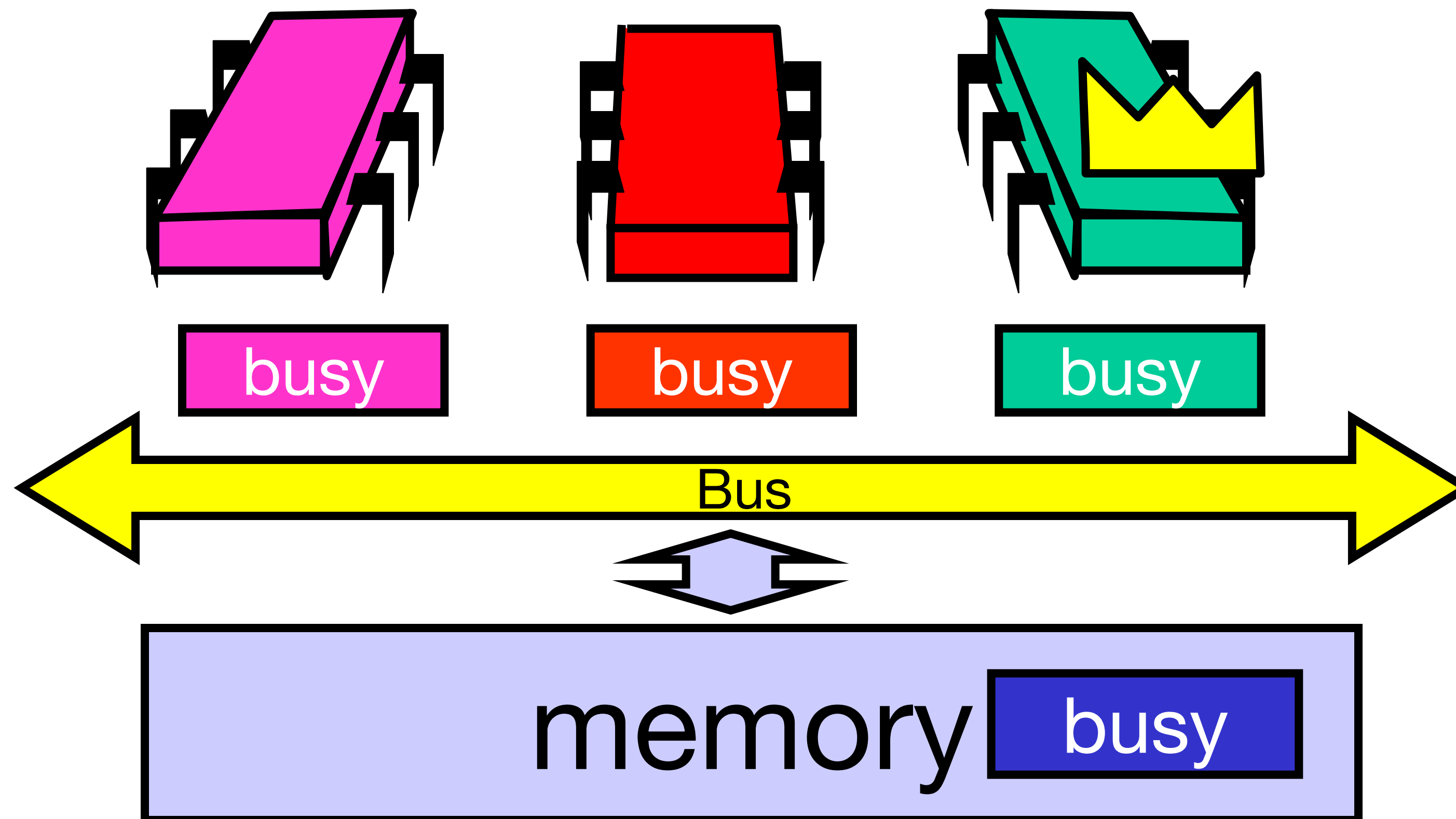
Test-and-set Lock (TASLock)

- 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

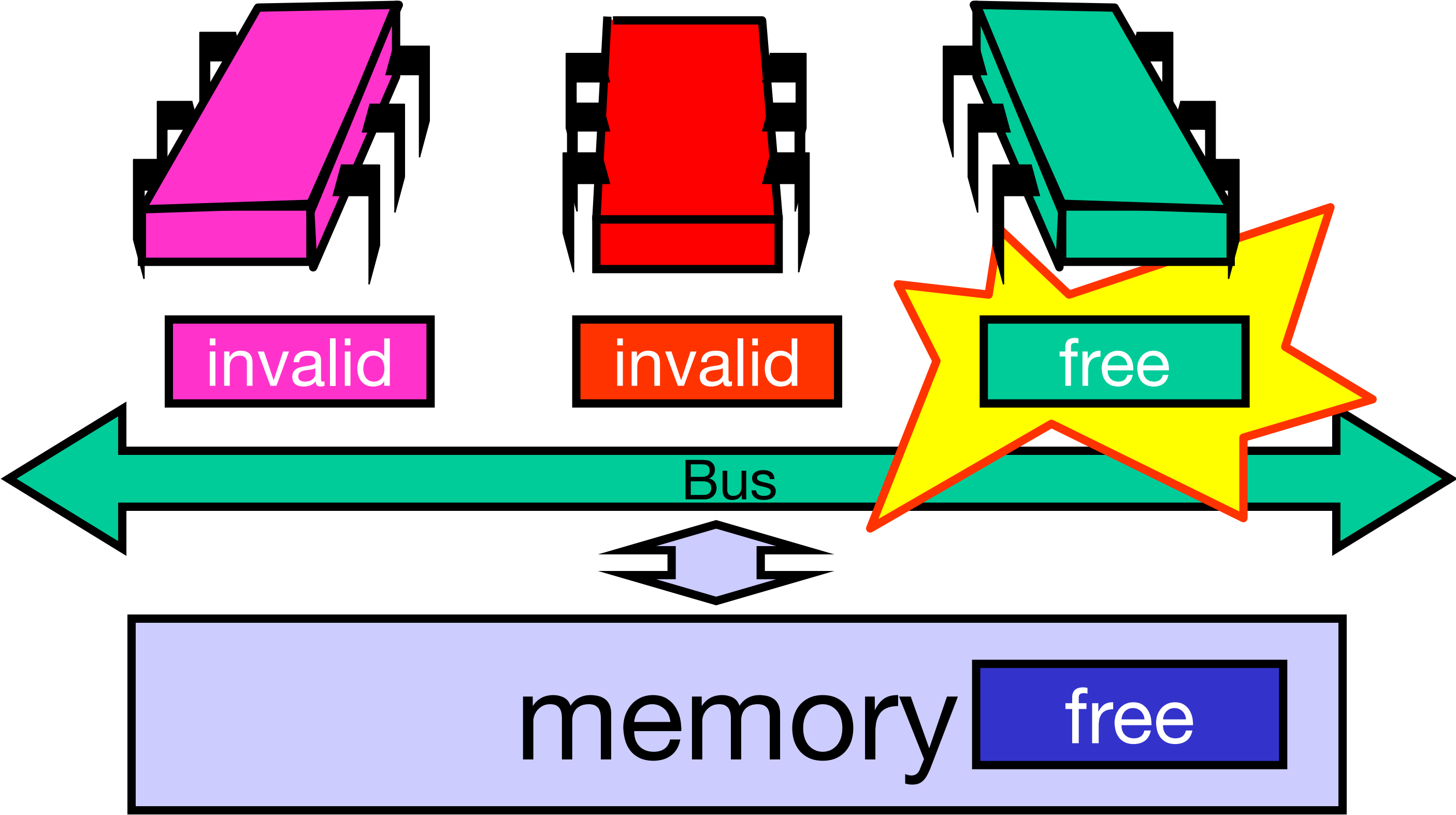
Test-and-test-and-set Lock (TTASLock)

- 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

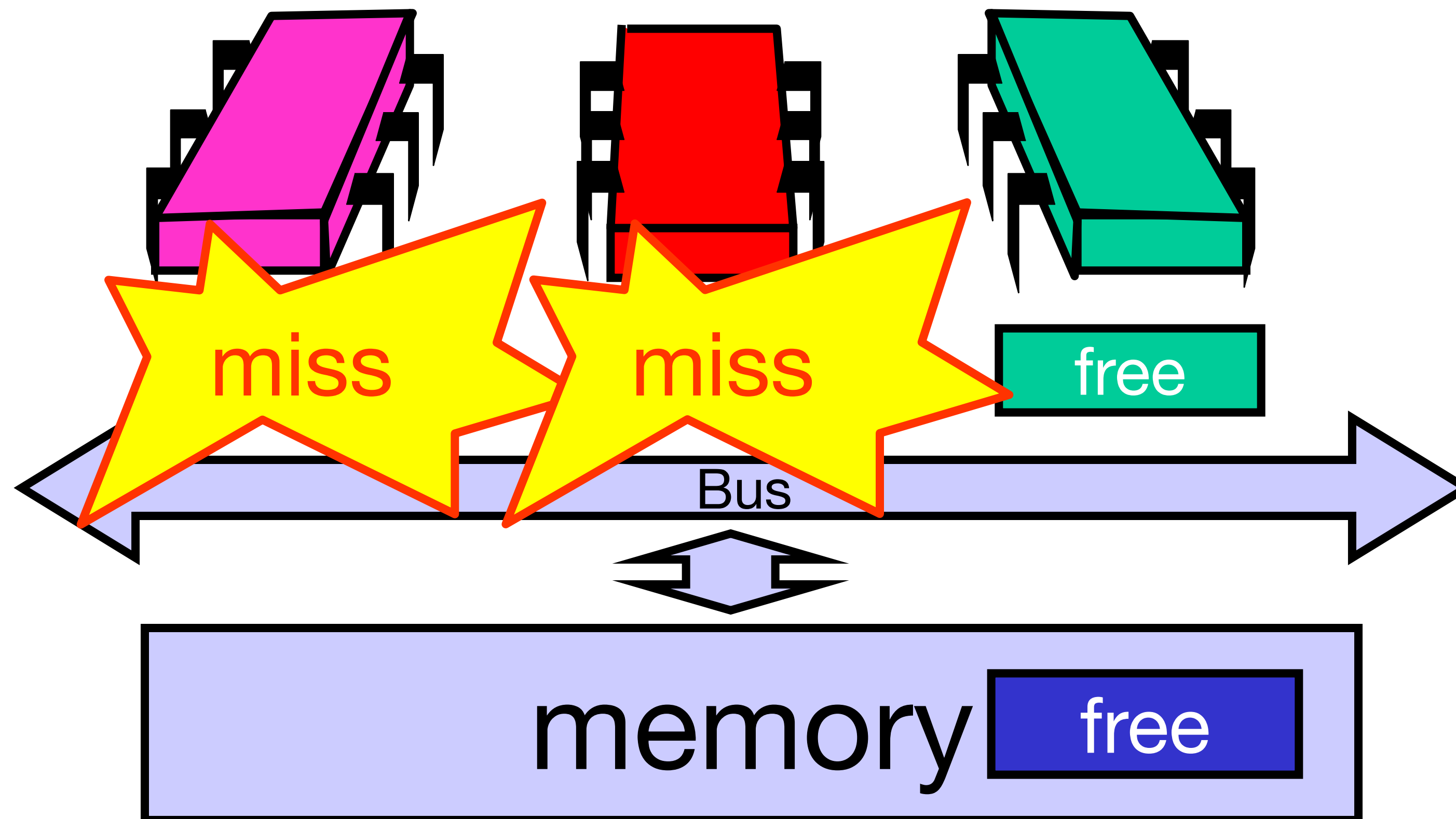


On Release



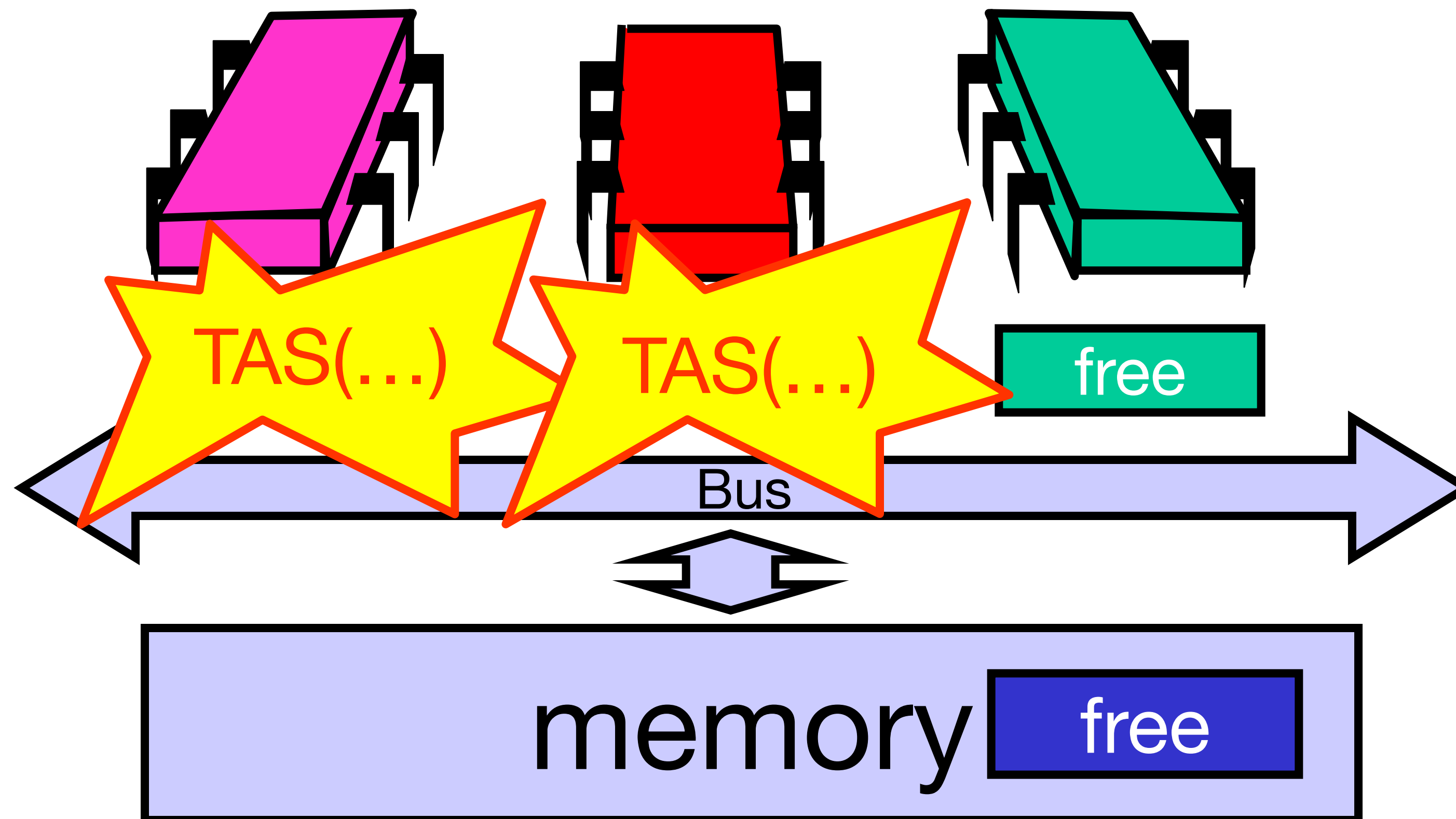
On Release

Everyone misses,
rereads



On Release

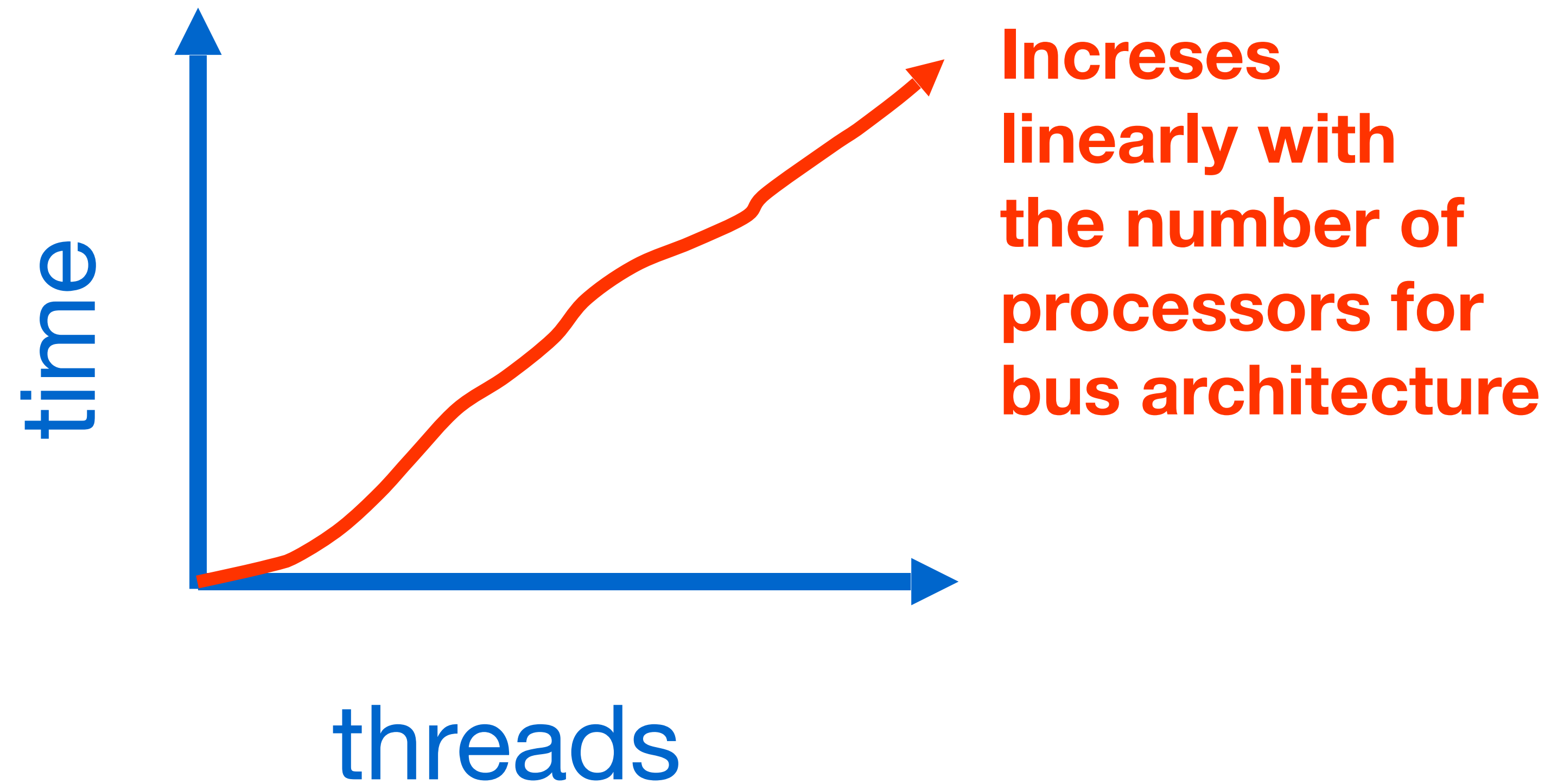
Everyone tries TAS



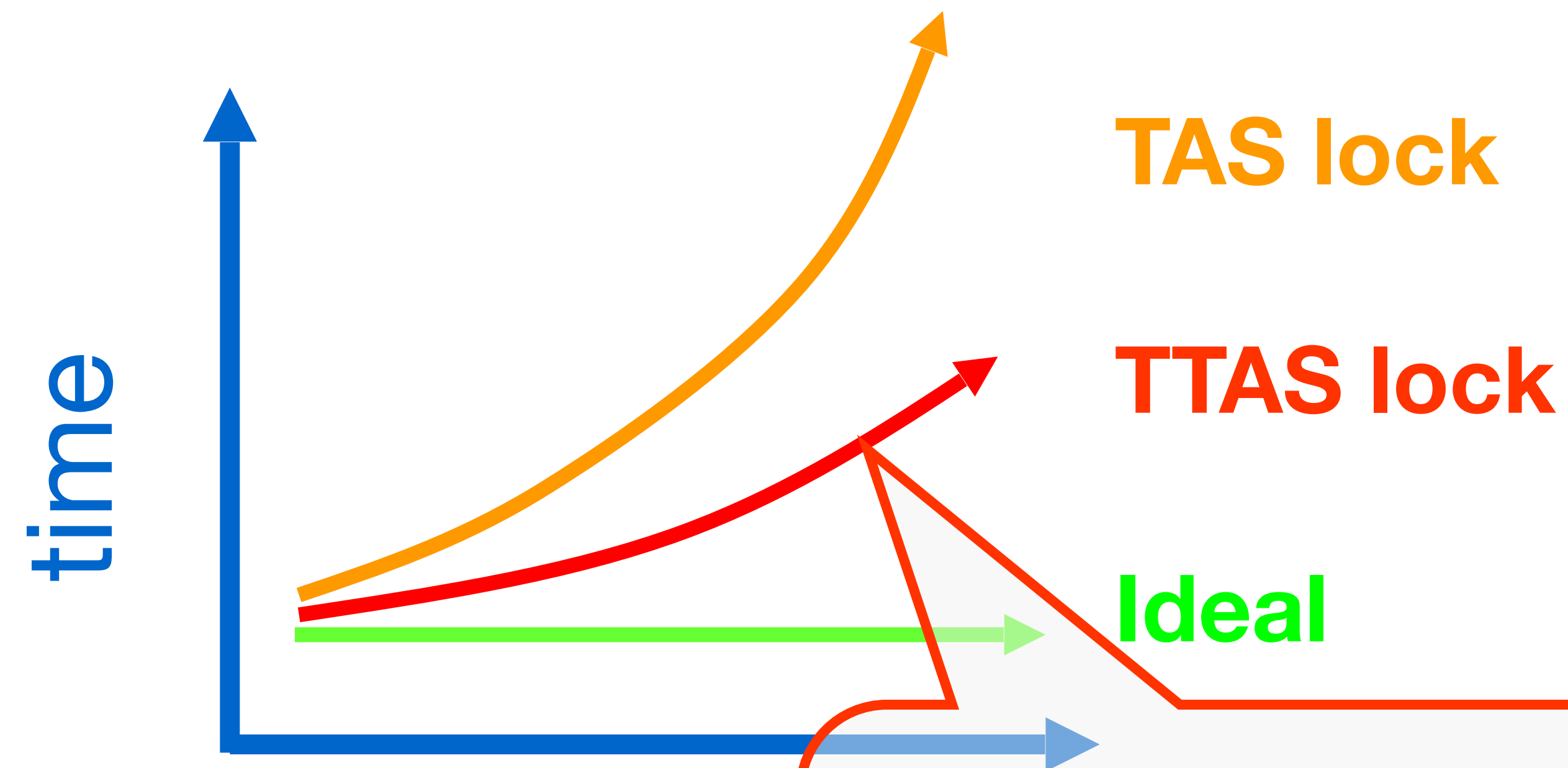
Problems

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

Quiescence Time



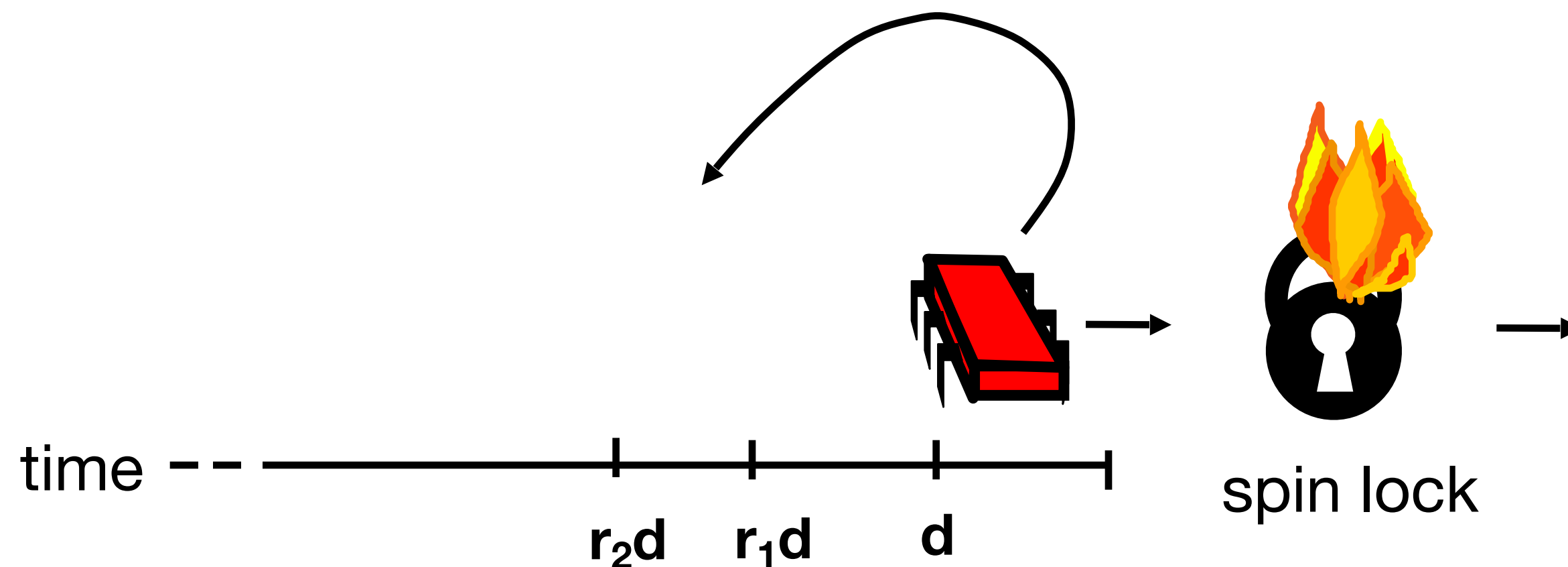
Mystery Explained



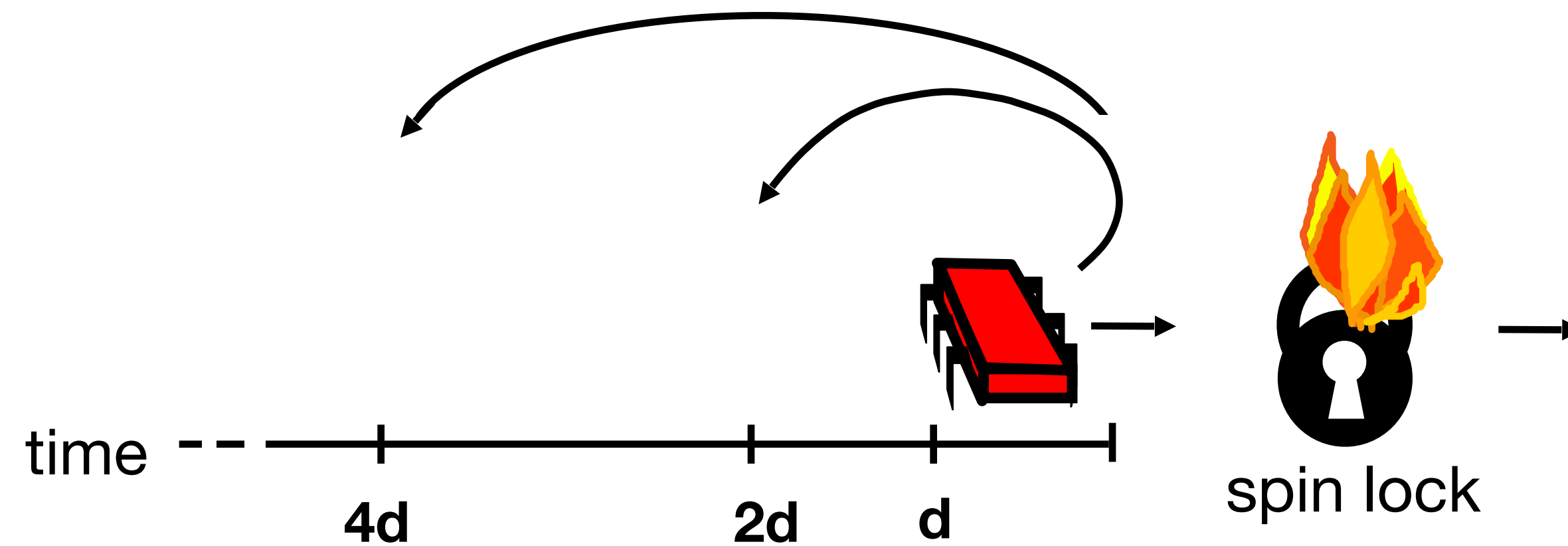
Better than TAS
but still not as
good as ideal

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 faster than sleep() *)
    done;
    (* Exponentially increase the limit, capped at max_delay *)
    t.limit <- min t.max_delay (t.limit * 2)
end
```

BackoffLock

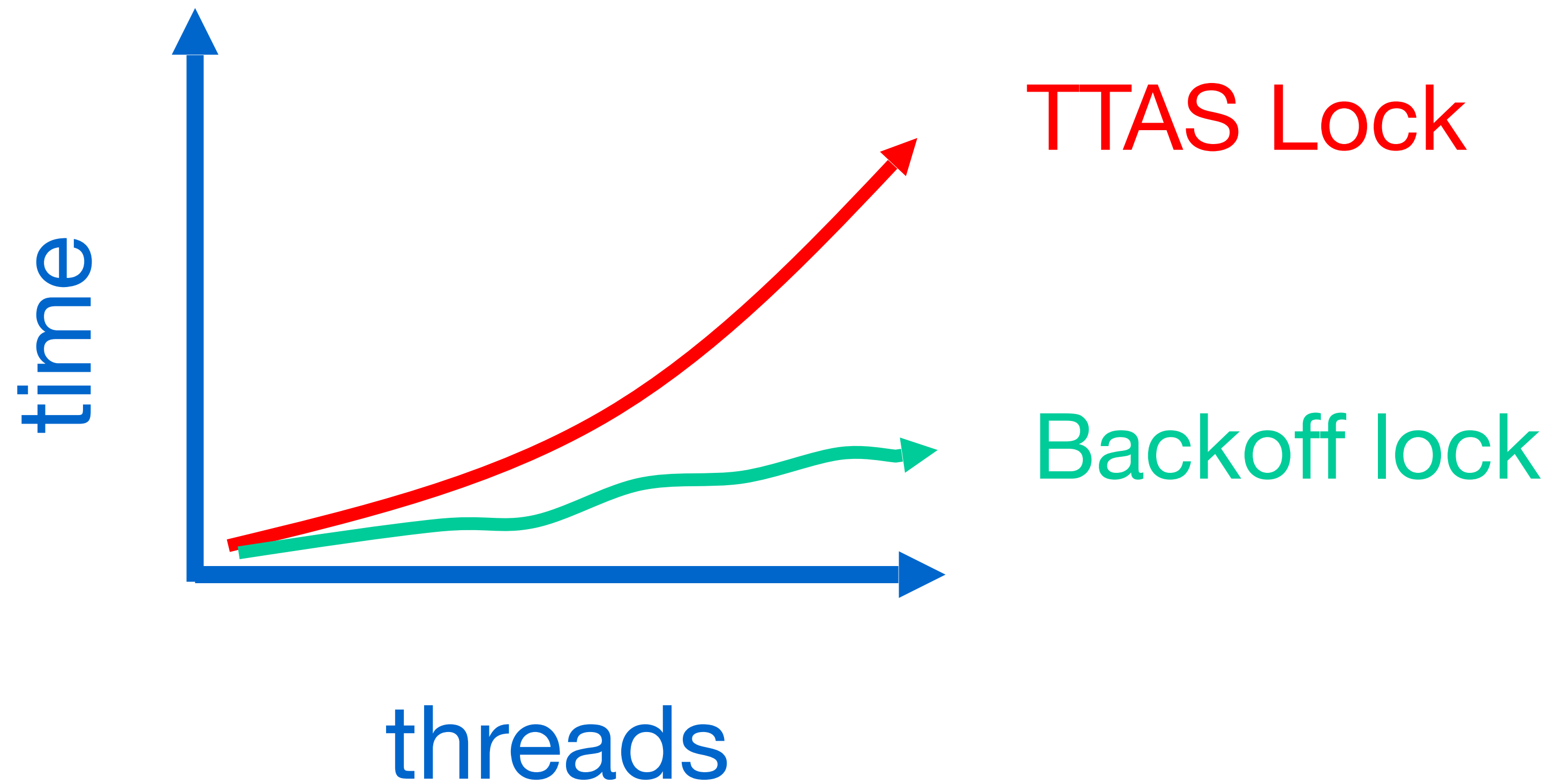
```
module MakeBackoffLock (P : BACKOFF_PARAMS) : Lock.LOCK = struct
  type t = { state : bool Atomic.t }

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

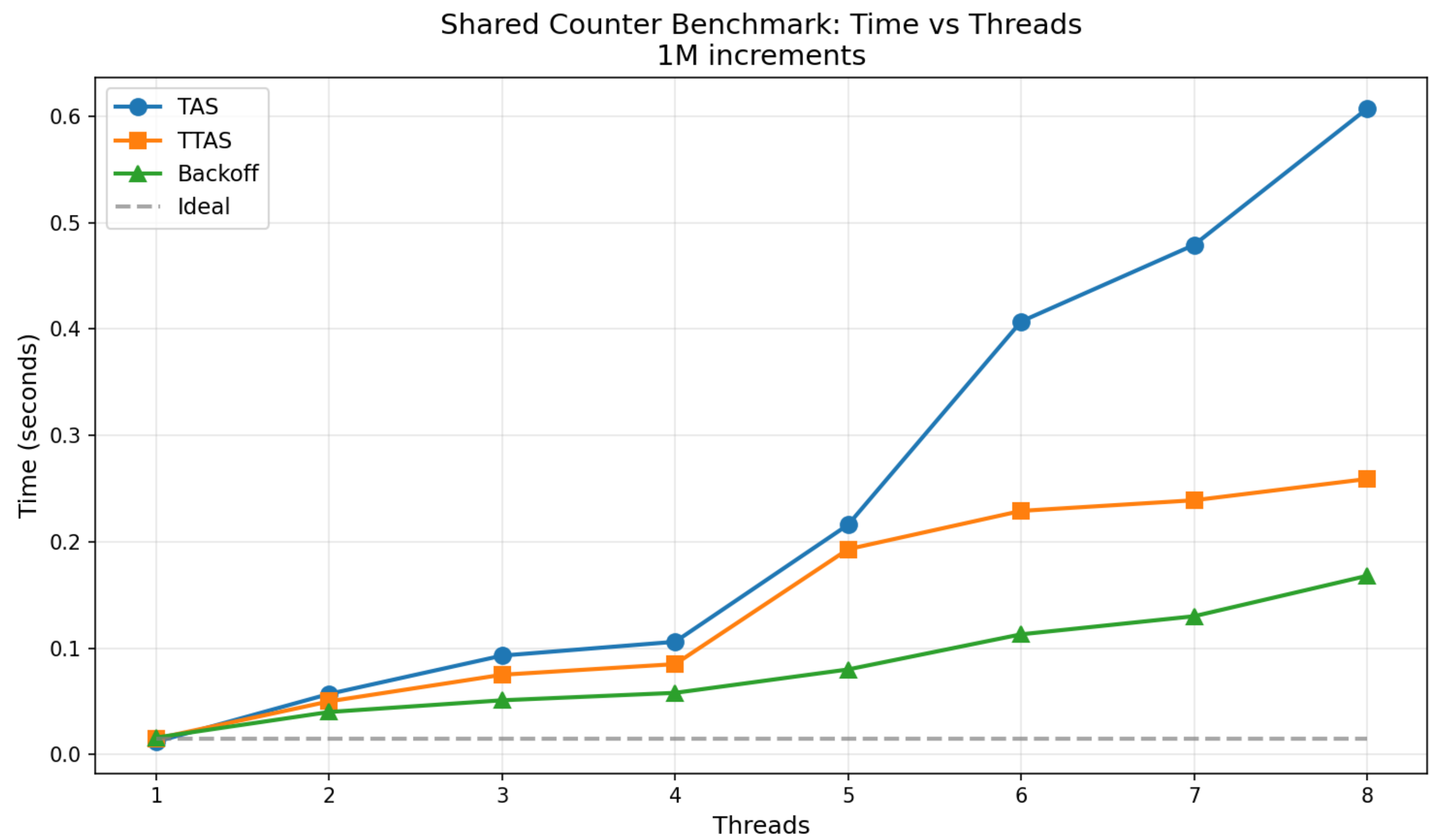
  let lock t =
    let backoff = Backoff.create P.min_delay P.max_delay in
    (* Outer loop: keep trying until we get the lock *)
    while
      (* Inner loop: spin-read until lock appears free *)
      while Atomic.get t.state do
        ()
      done;
      (* Lock looks free, try to acquire *)
  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



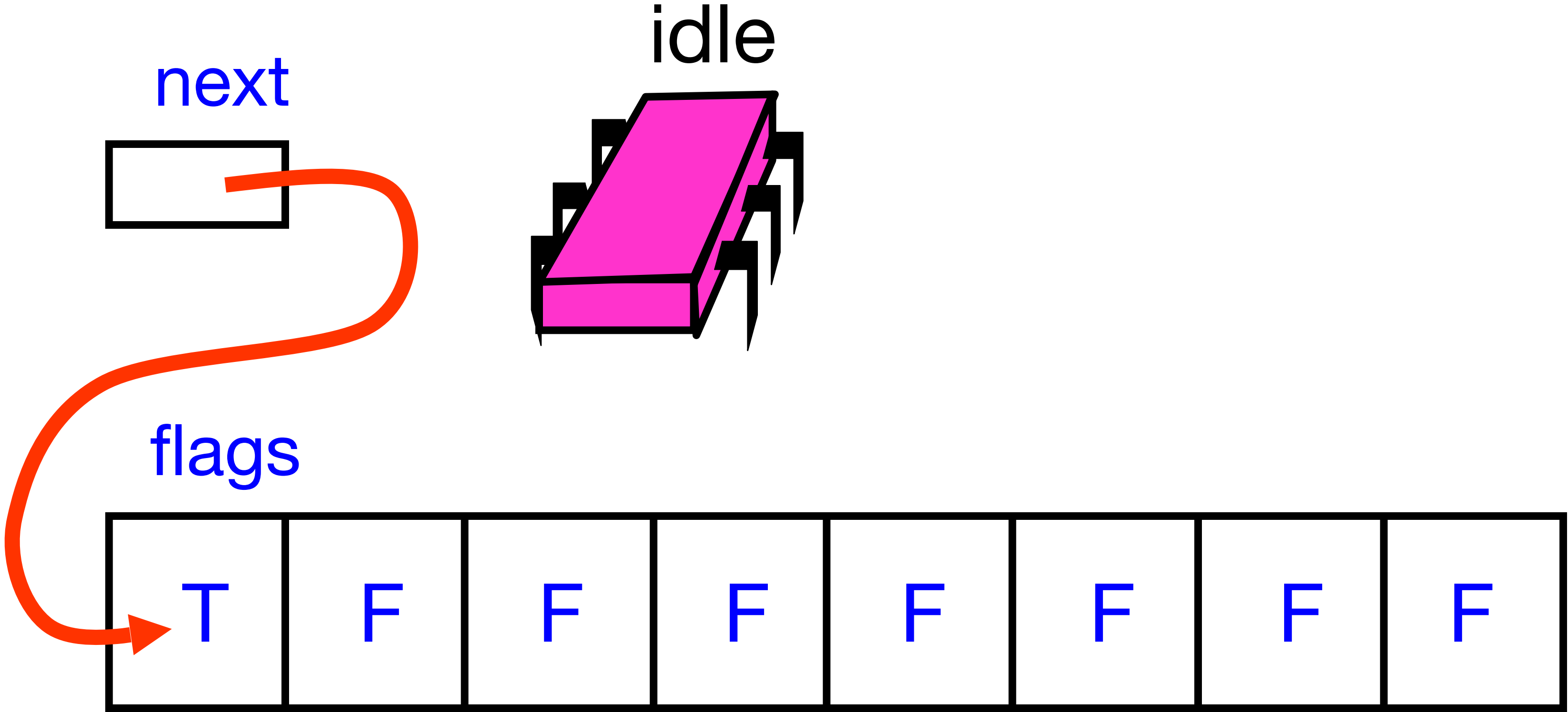
Backoff: Other issues

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

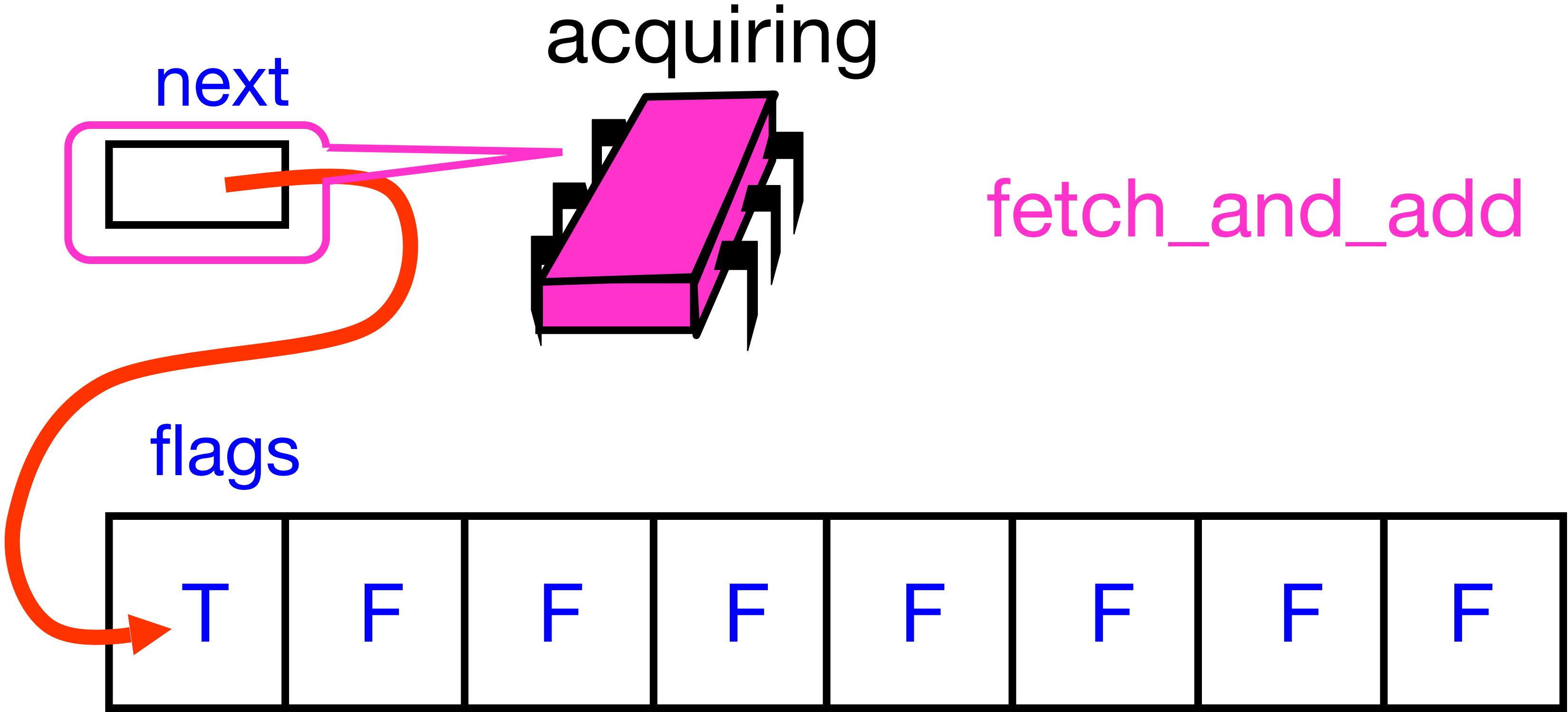
Idea

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

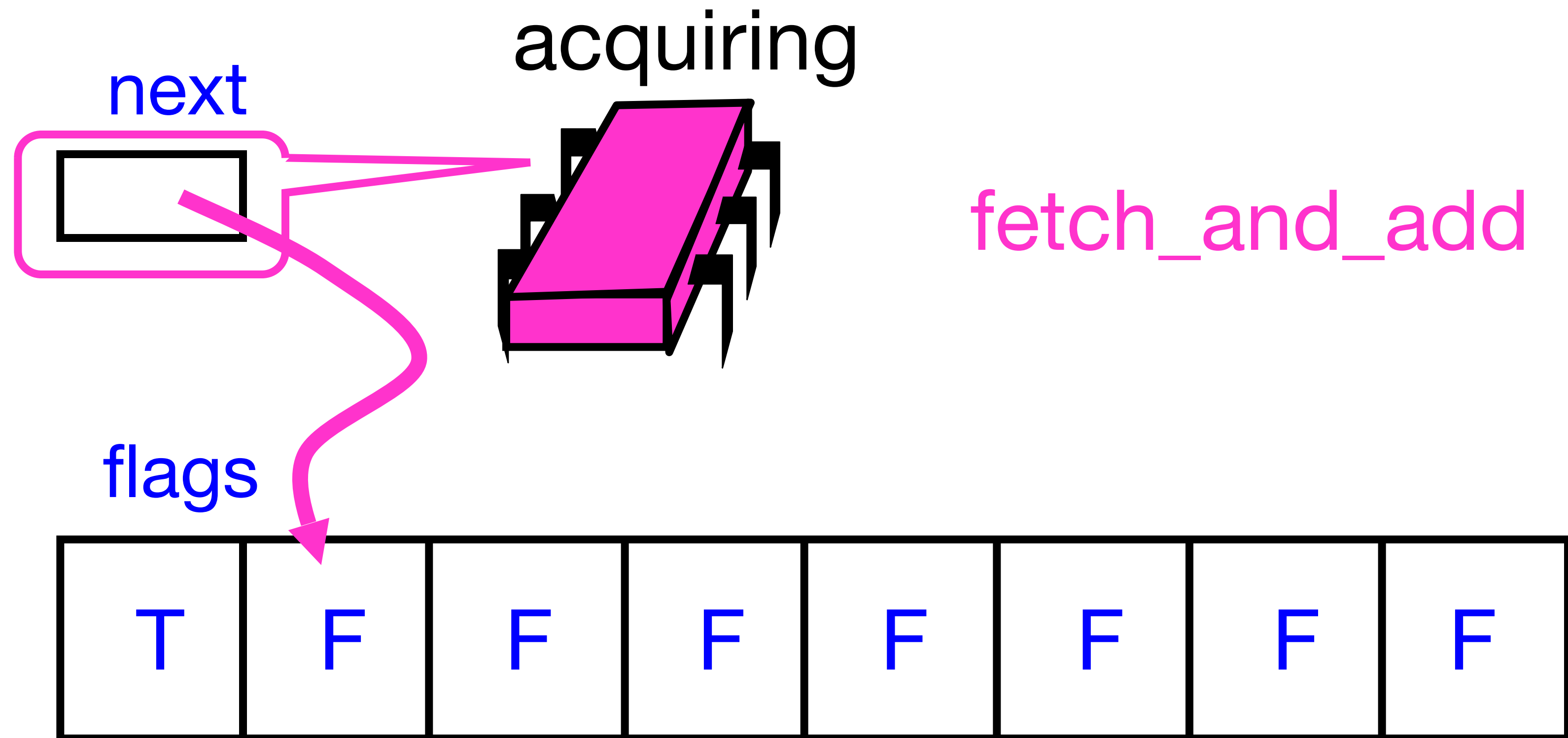
Anderson Queue Lock



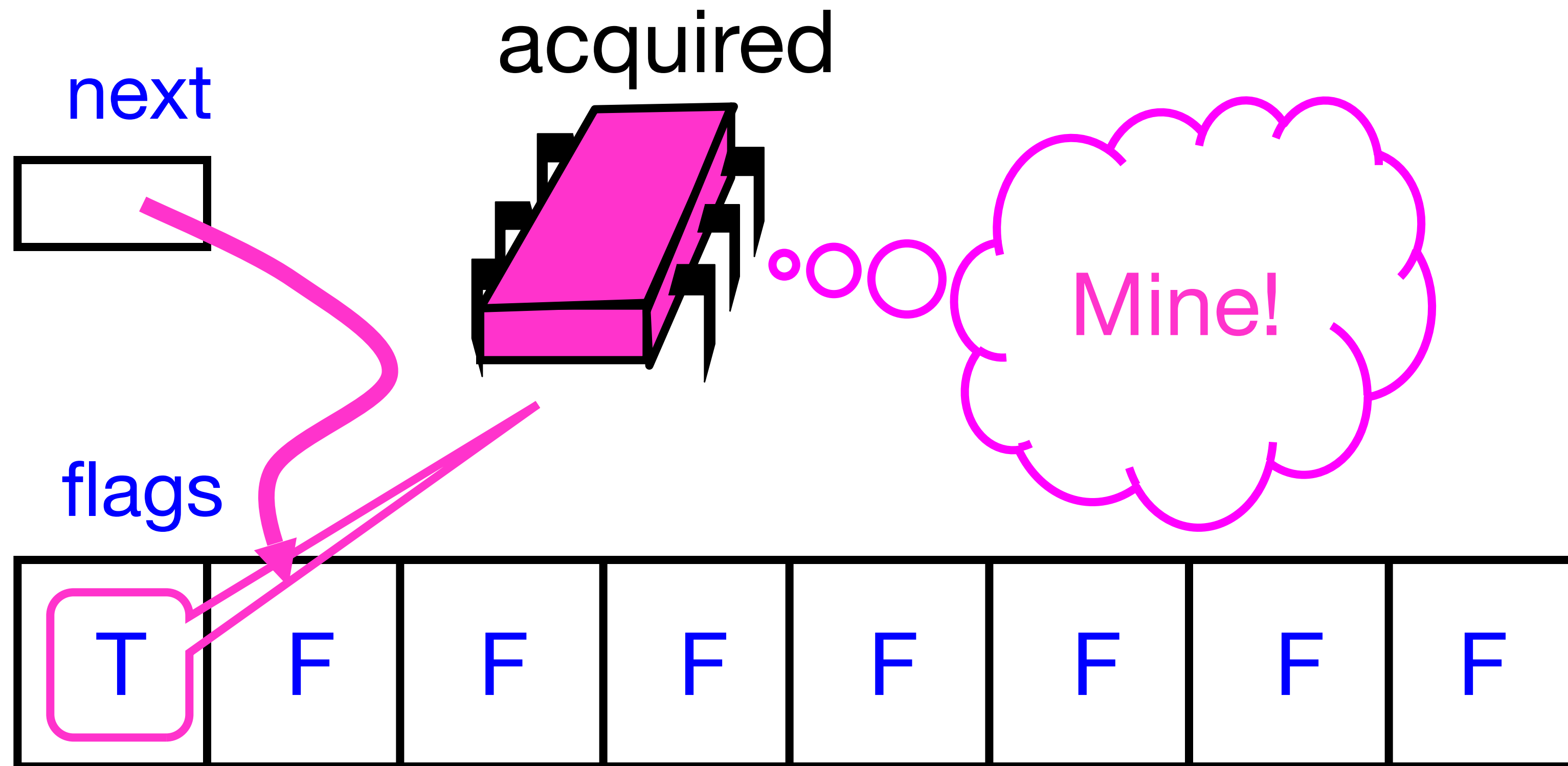
Anderson Queue Lock



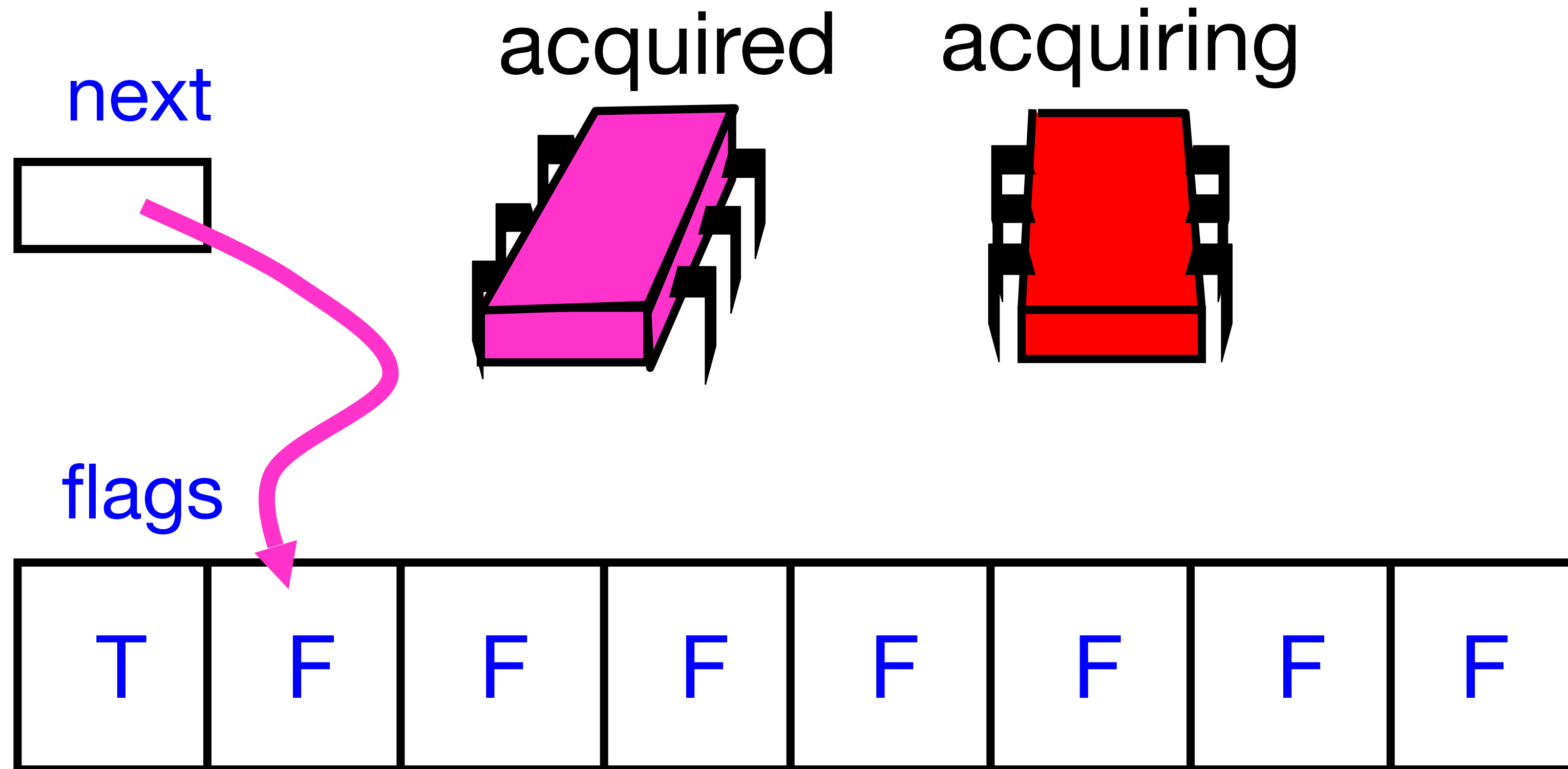
Anderson Queue Lock



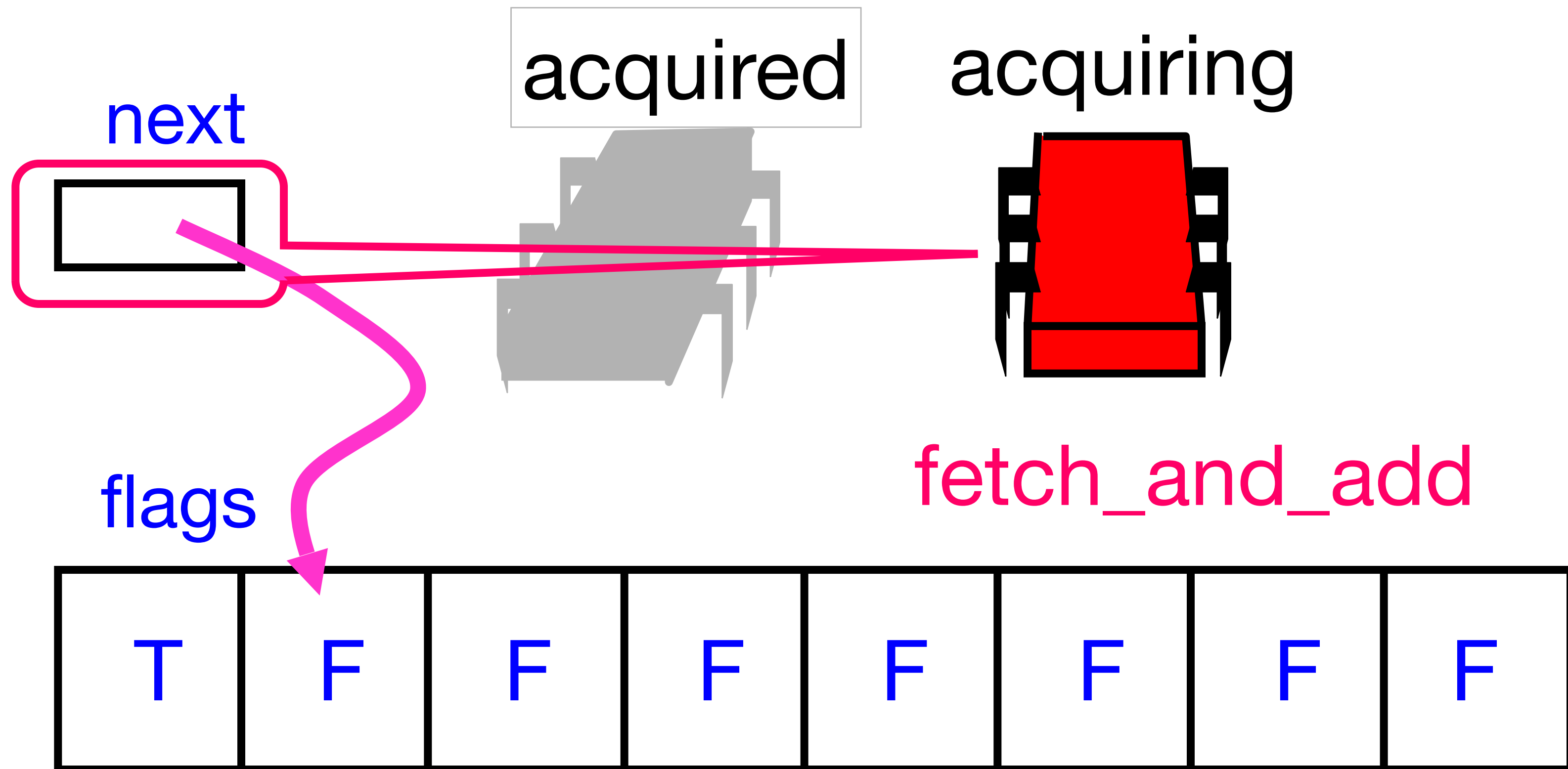
Anderson Queue Lock



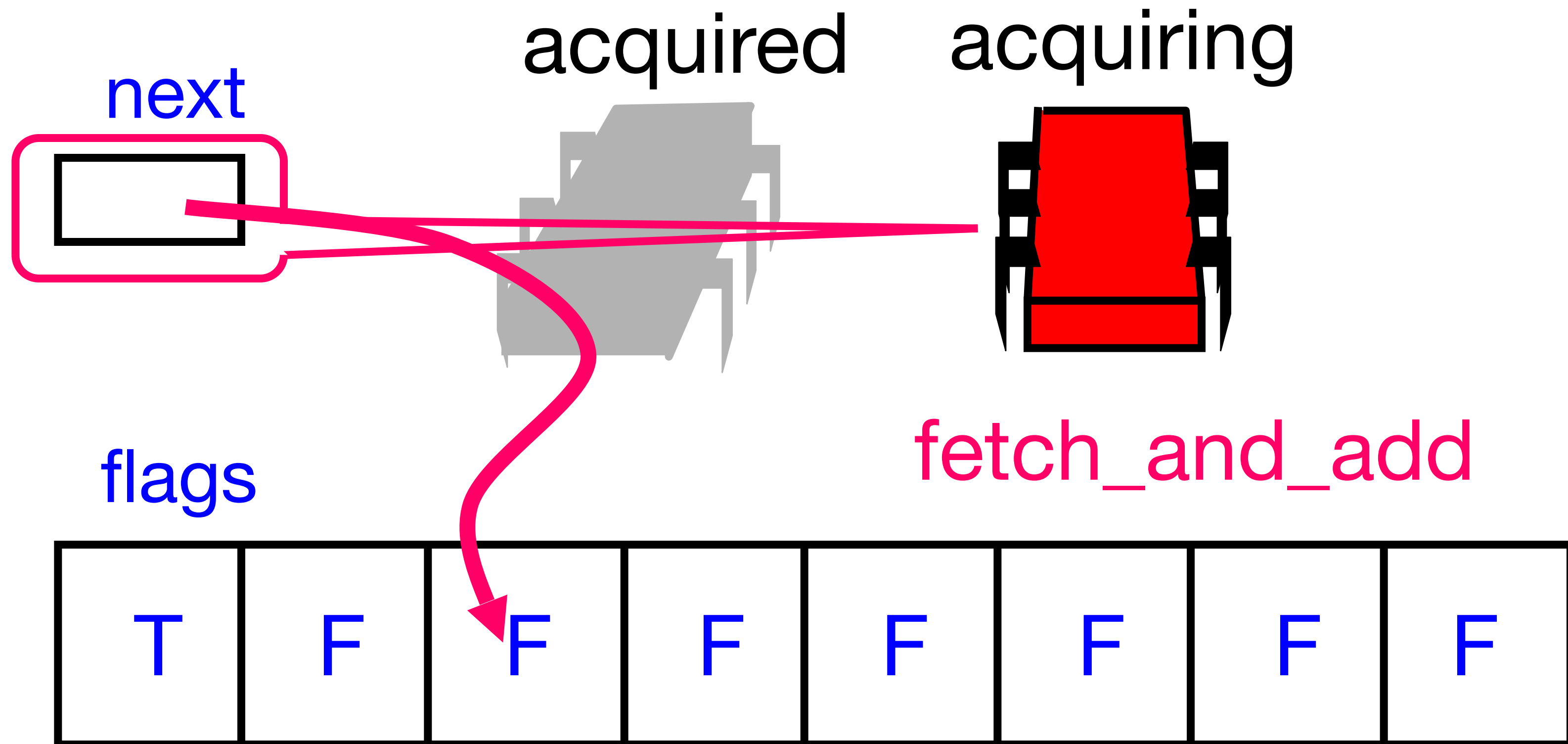
Anderson Queue Lock



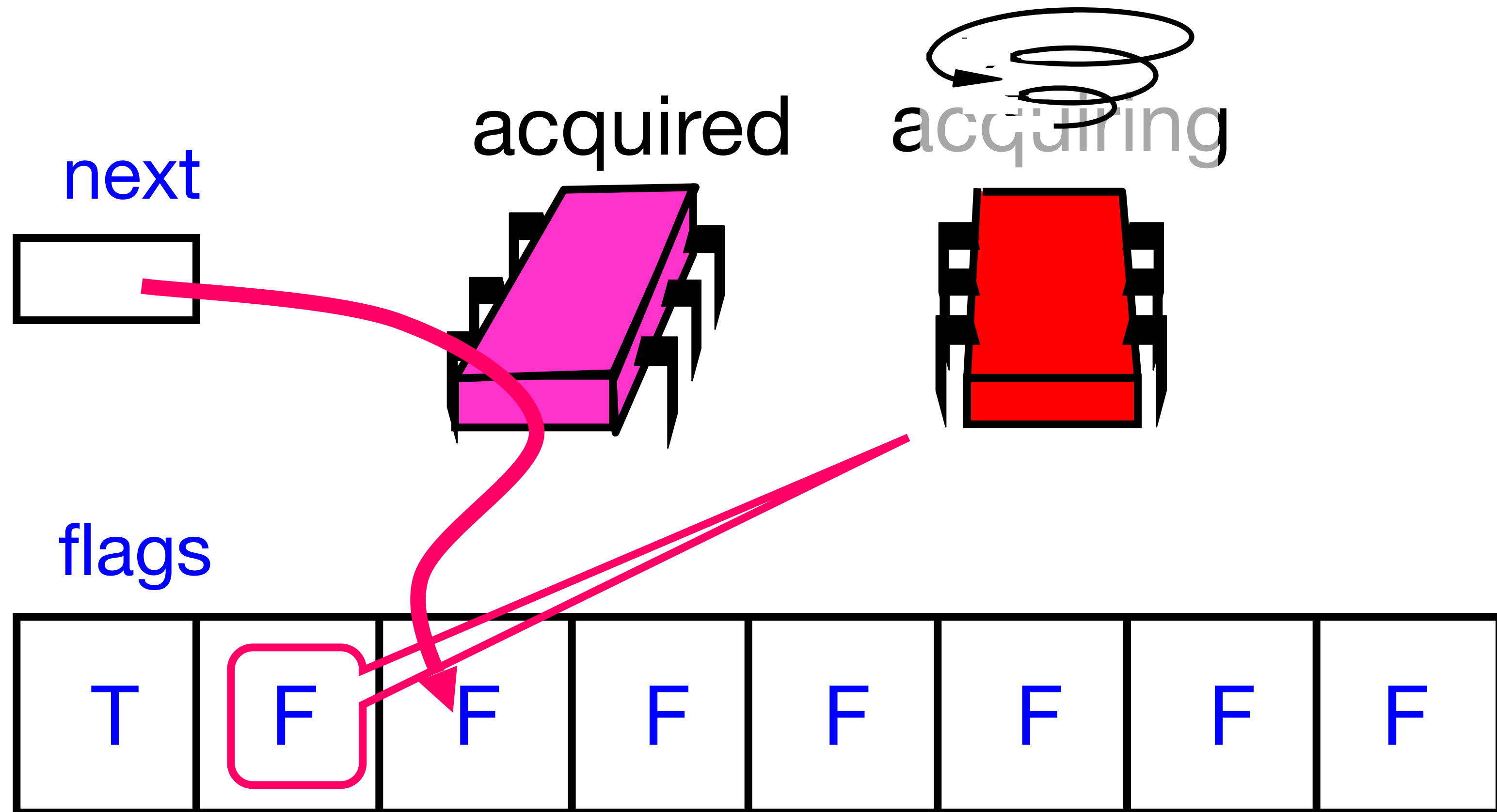
Anderson Queue Lock



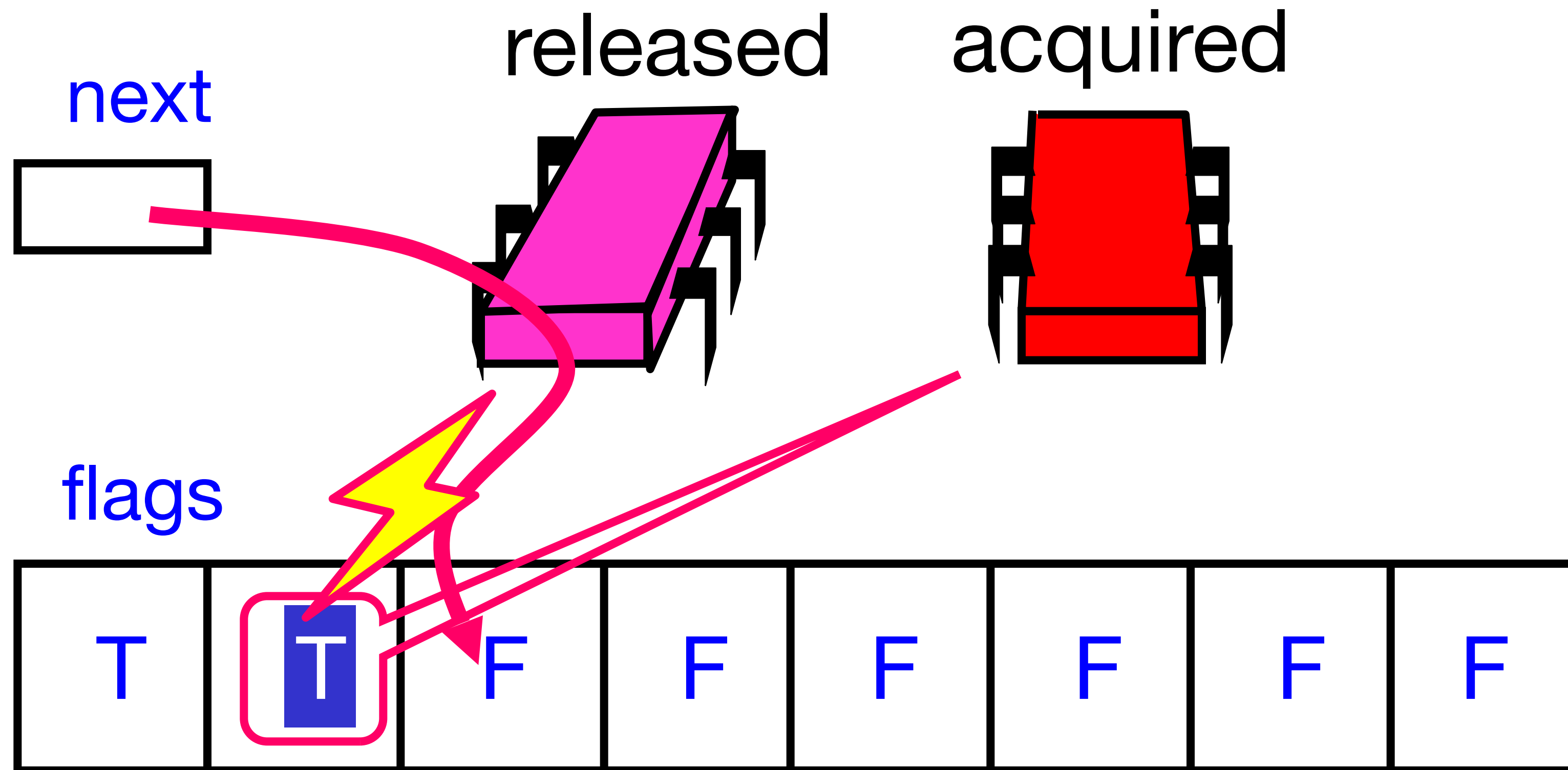
Anderson Queue Lock



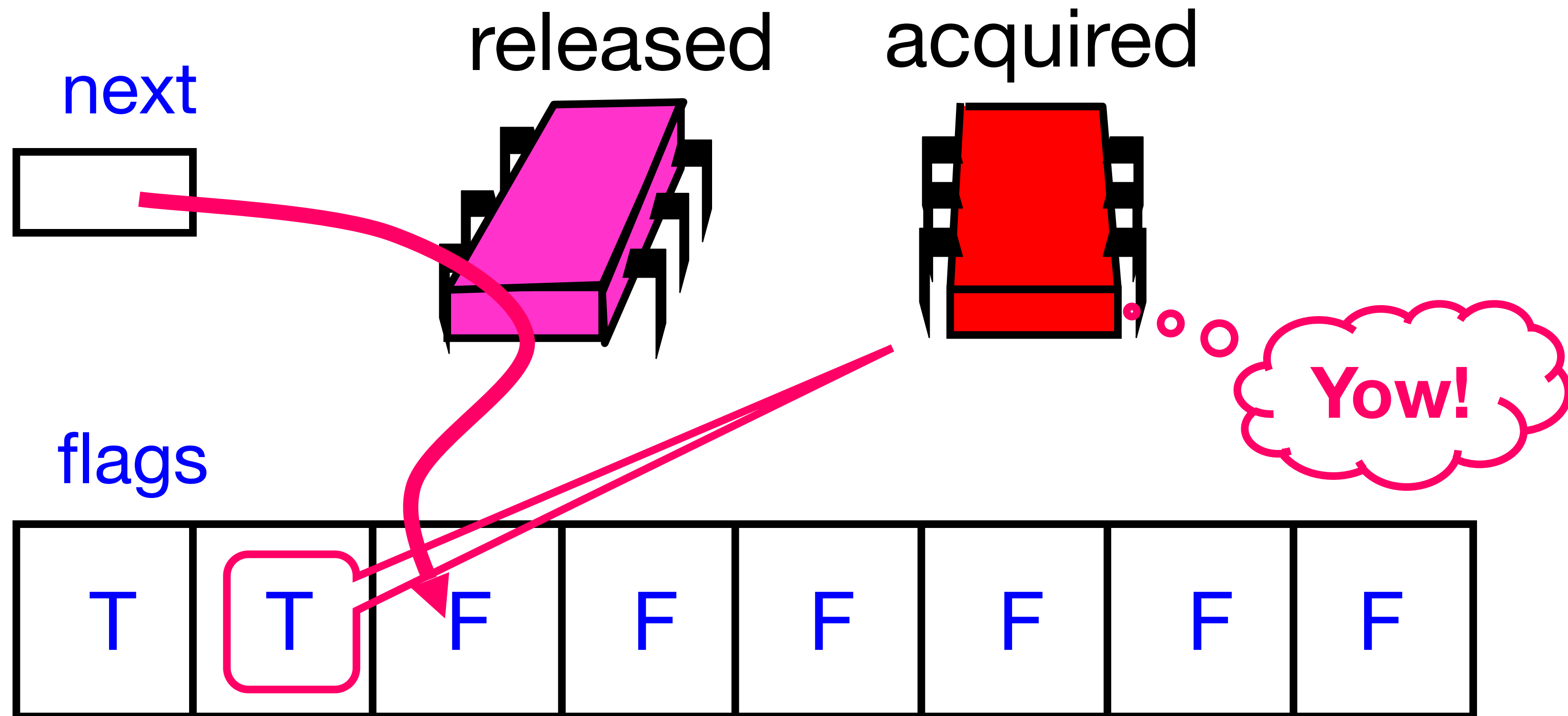
Anderson Queue Lock



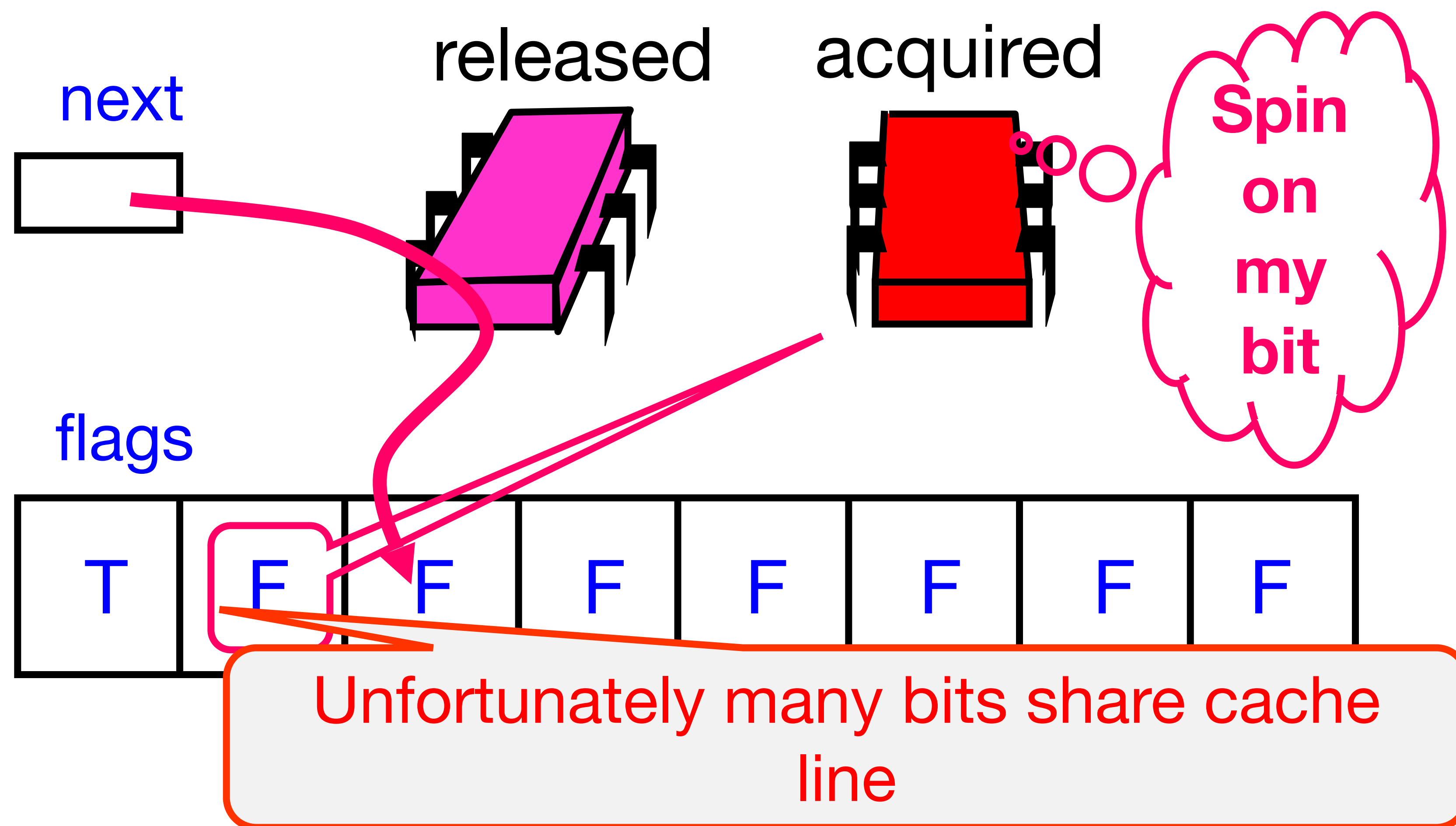
Anderson Queue Lock



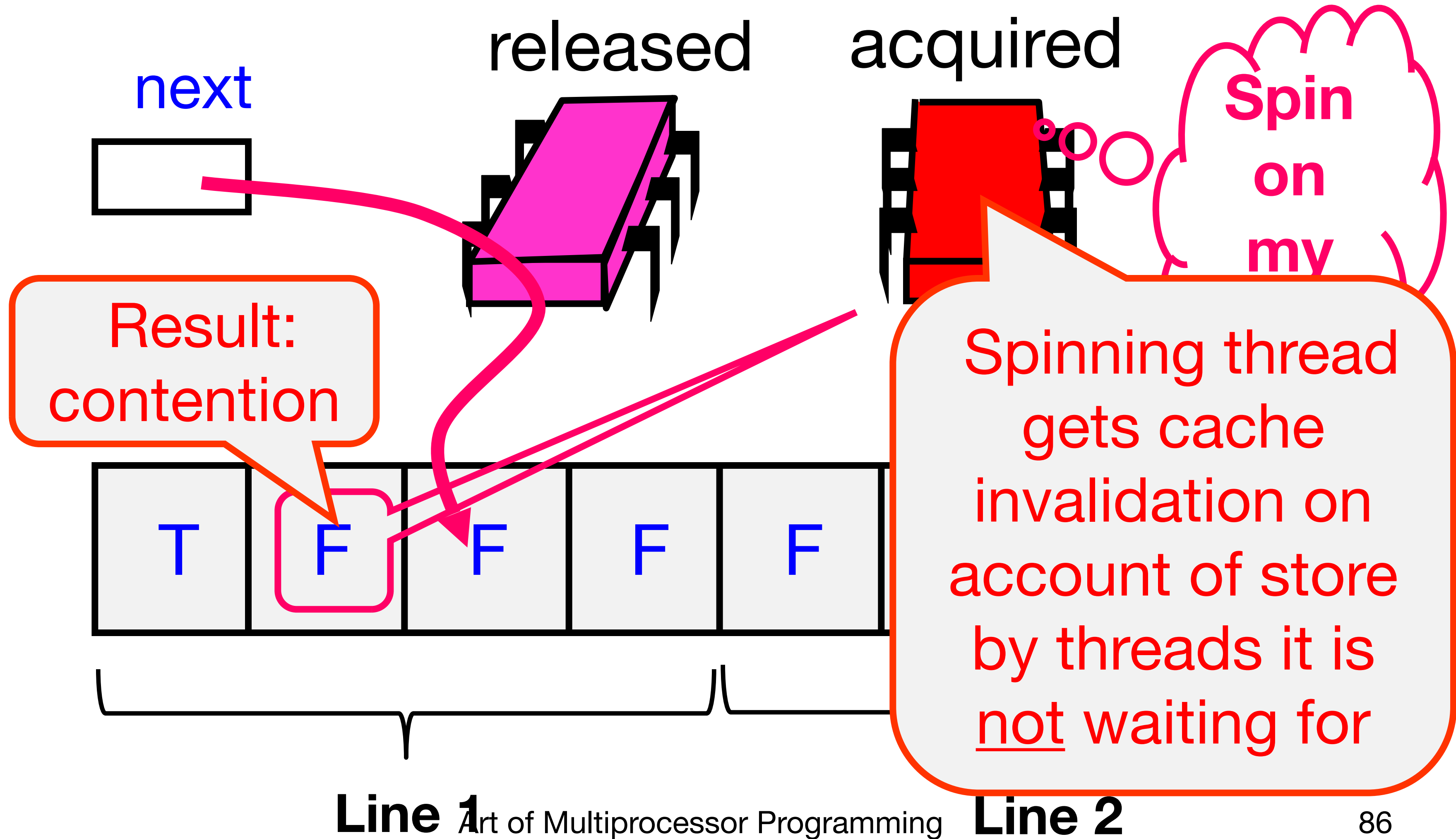
Anderson Queue Lock



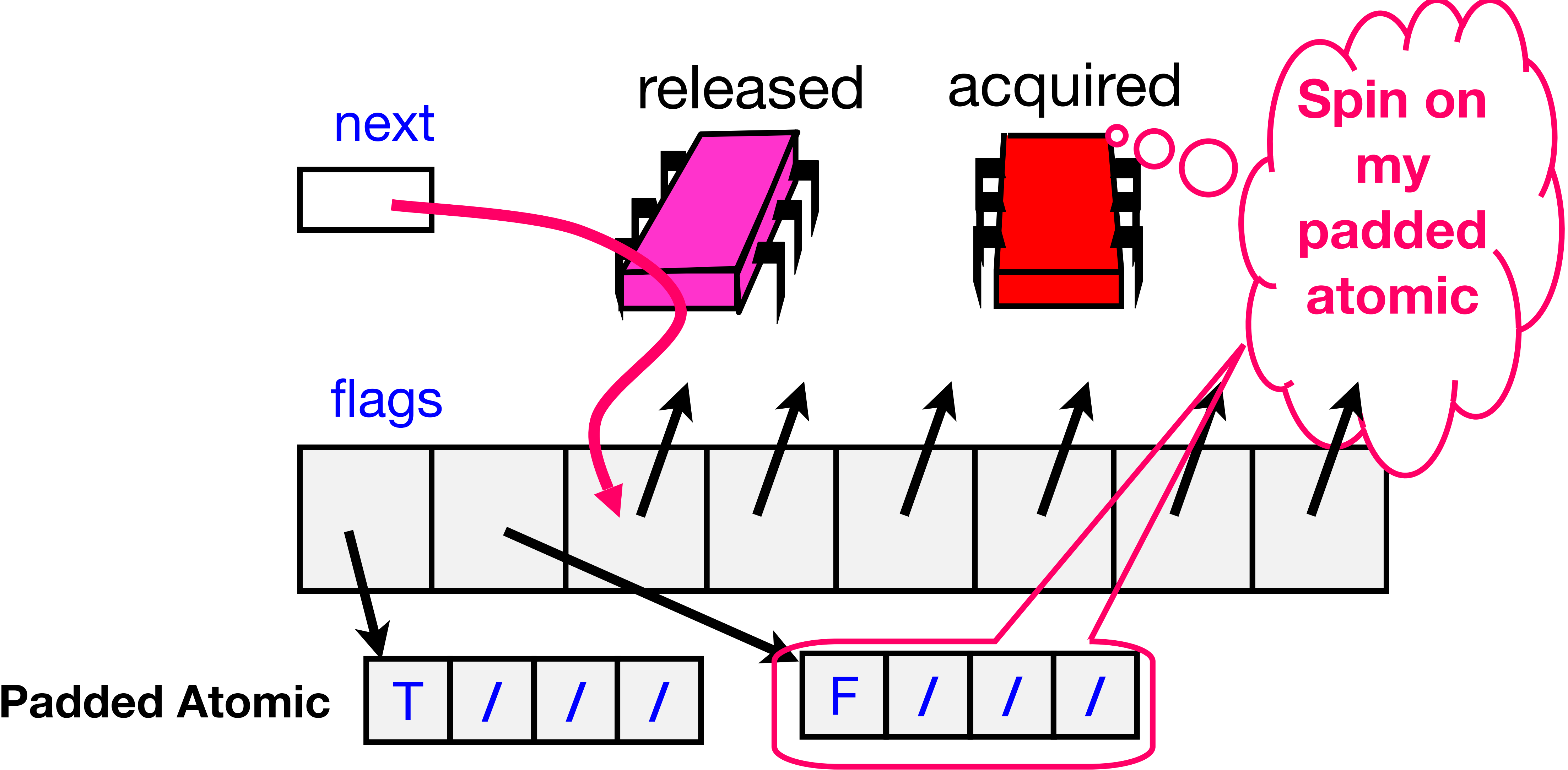
Local Spinning



False Sharing



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;  
}
```

```
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);  
  }
```

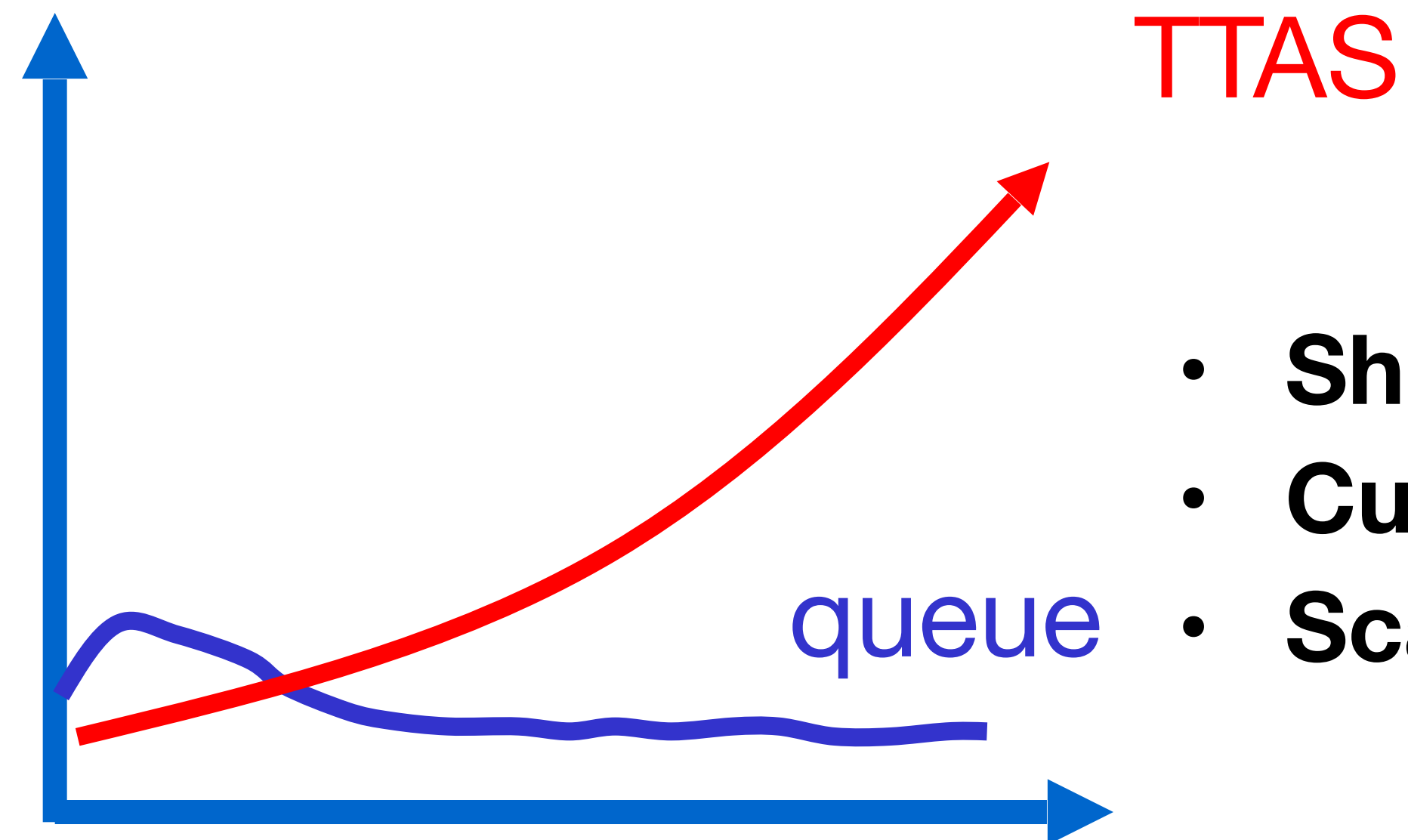

OCaml ALock

```
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

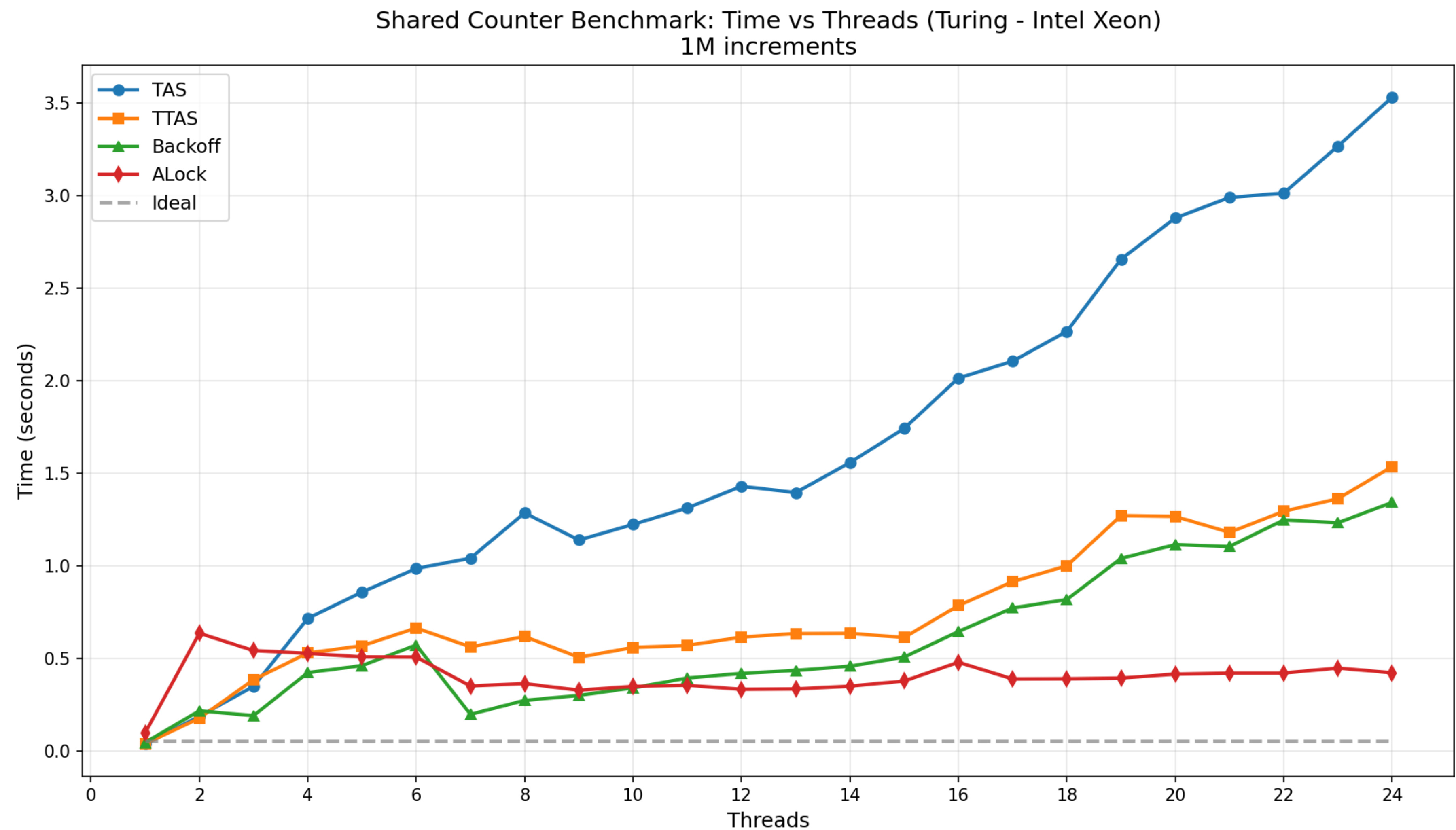
```
let unlock t =  
  (* Get my slot from domain-local storage *)  
  let slot = Domain.DLS.get t.my_slot in  
  
  if slot = -1 then  
    failwith "unlock called without corresponding lock";  
  
  (* Cache flag references *)  
  let my_flag = t.flags.(slot) in  
  let next_slot = (slot + 1) mod t.capacity 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

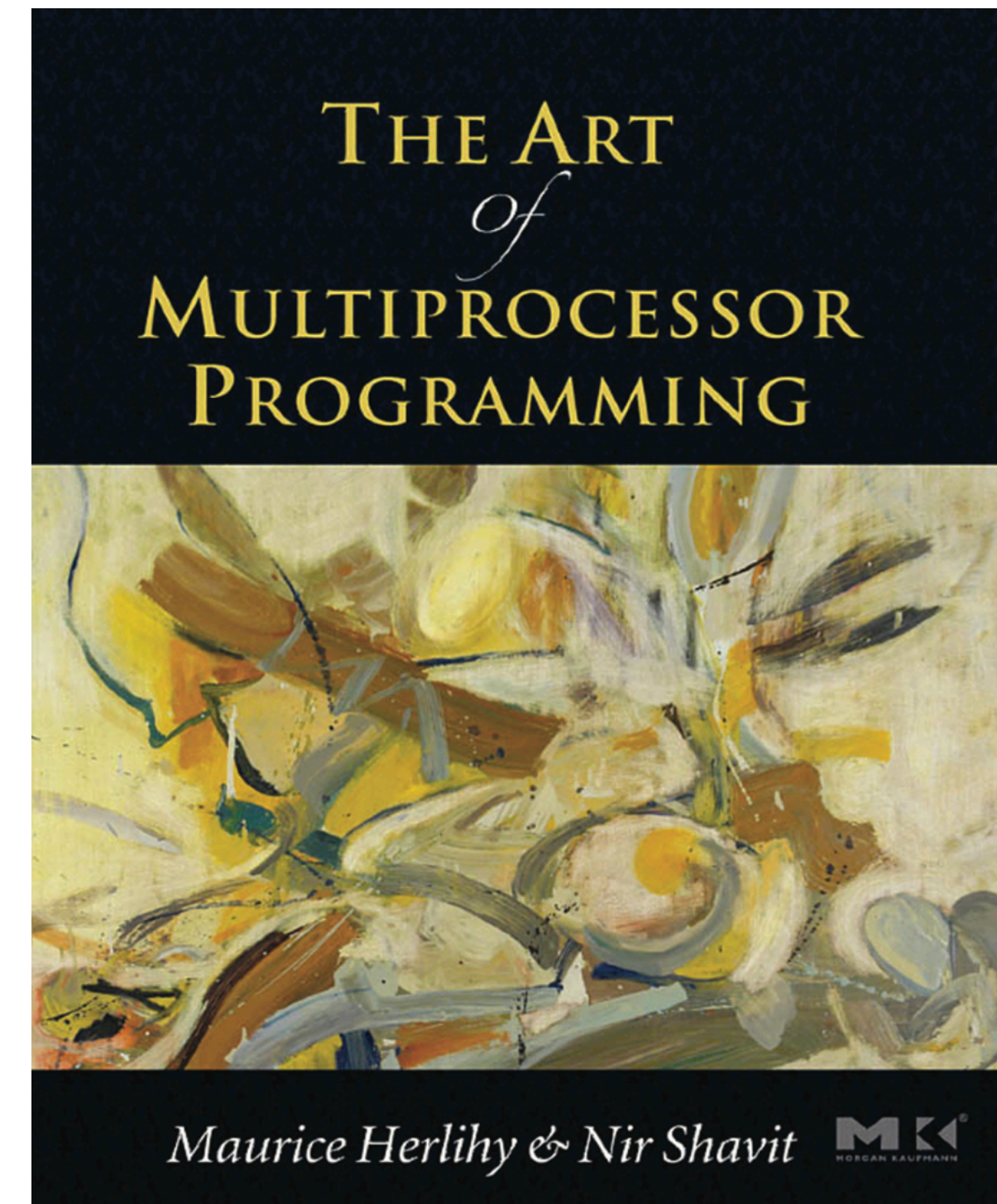


Anderson Queue Lock

- 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!



Chapter 7



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

- **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.