Spin Locks and Contention

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

Modified for Software1 students
by Lior Wolf and Mati Shomrat

Kinds of Architectures

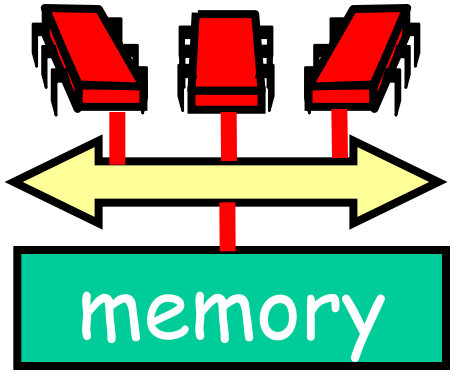
- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

Kinds of Architectures

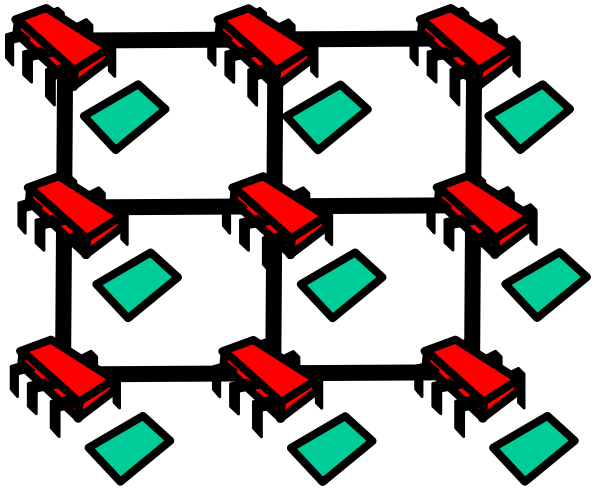
- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

Our space

MIMD Architectures



Shared Bus



Distributed

- Memory Contention
- Communication Contention
- Communication Latency

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

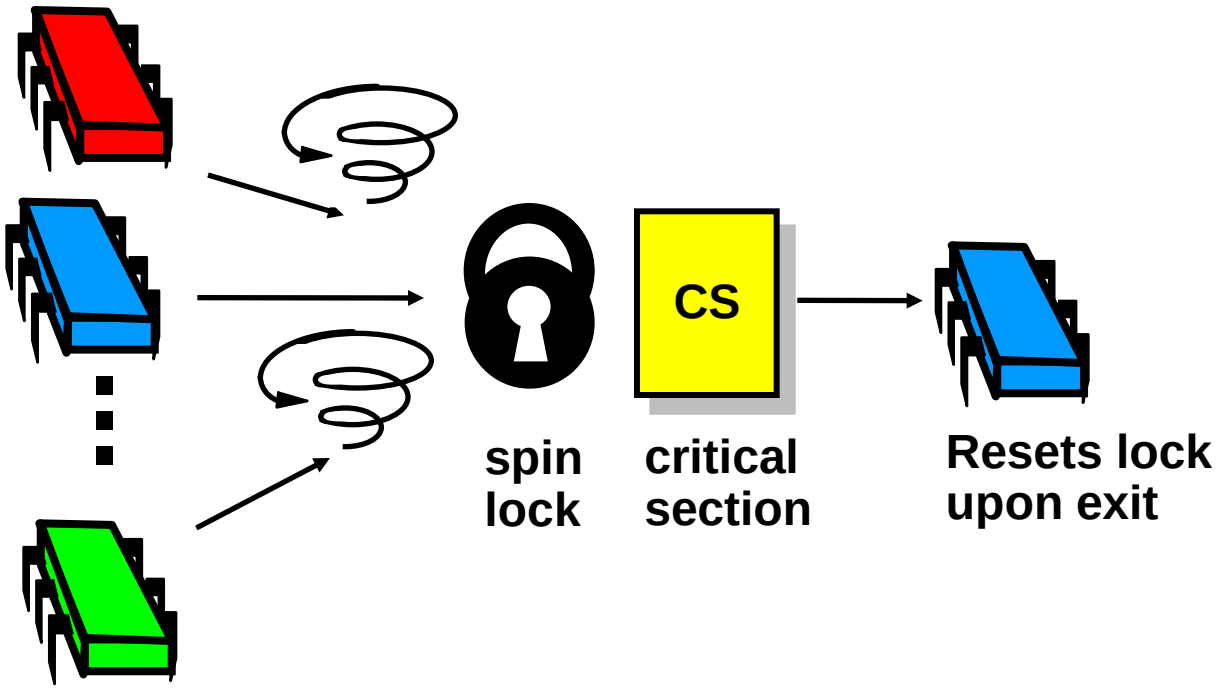
- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  [ask another thread to run
  expensive since switching is pricey]
  - Good if delays are long
  - Always good on uniprocessor

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

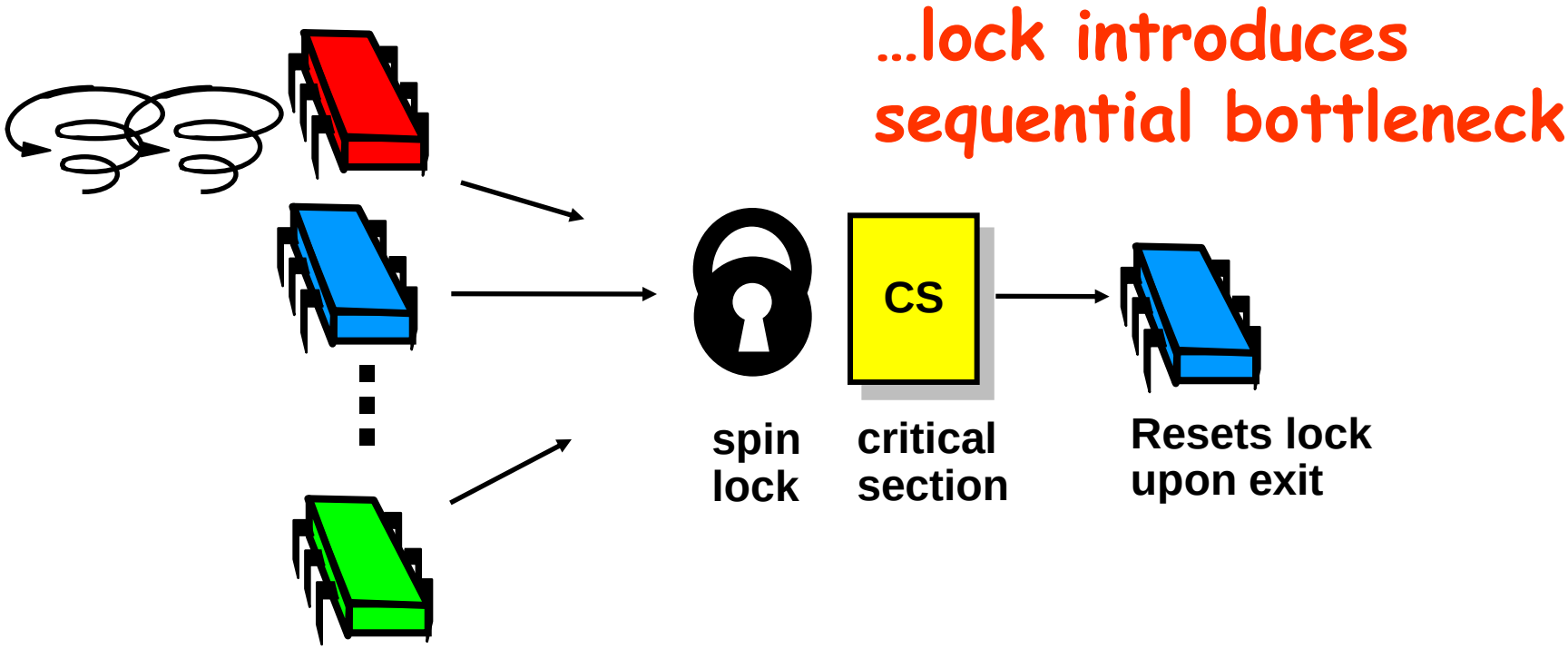
- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

our focus

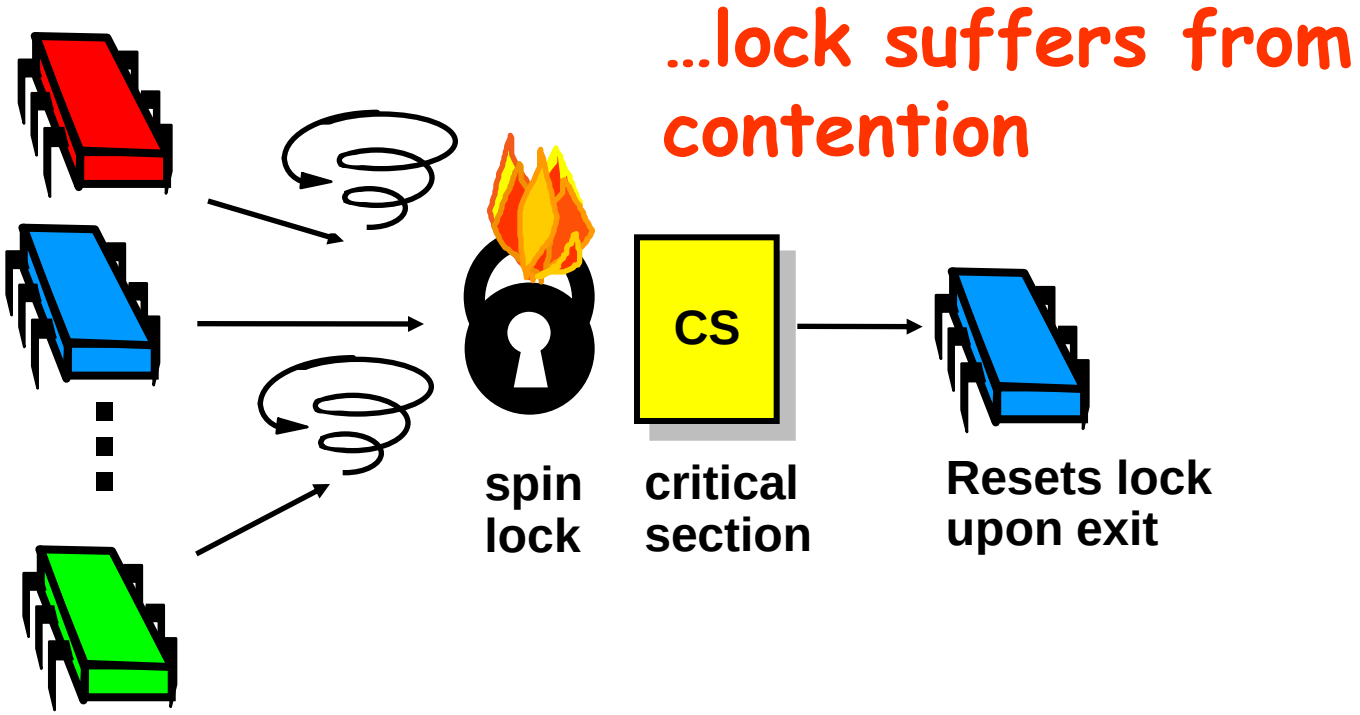
Basic Spin-Lock



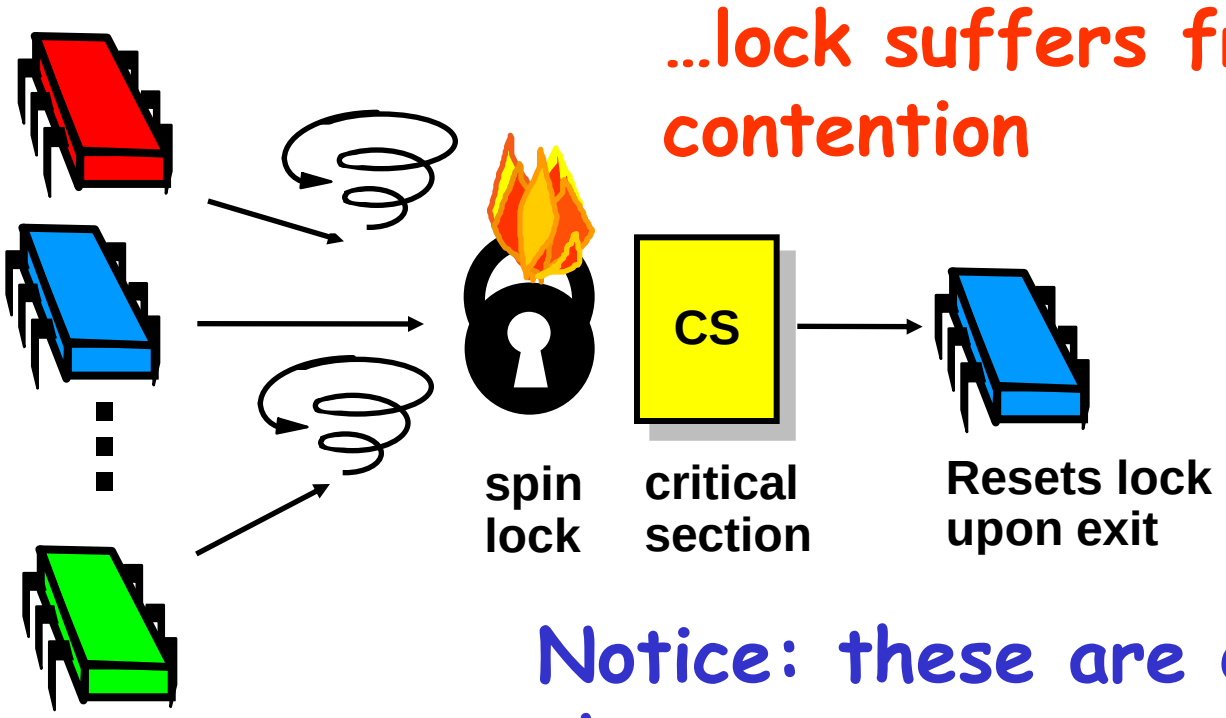
Basic Spin-Lock



Basic Spin-Lock

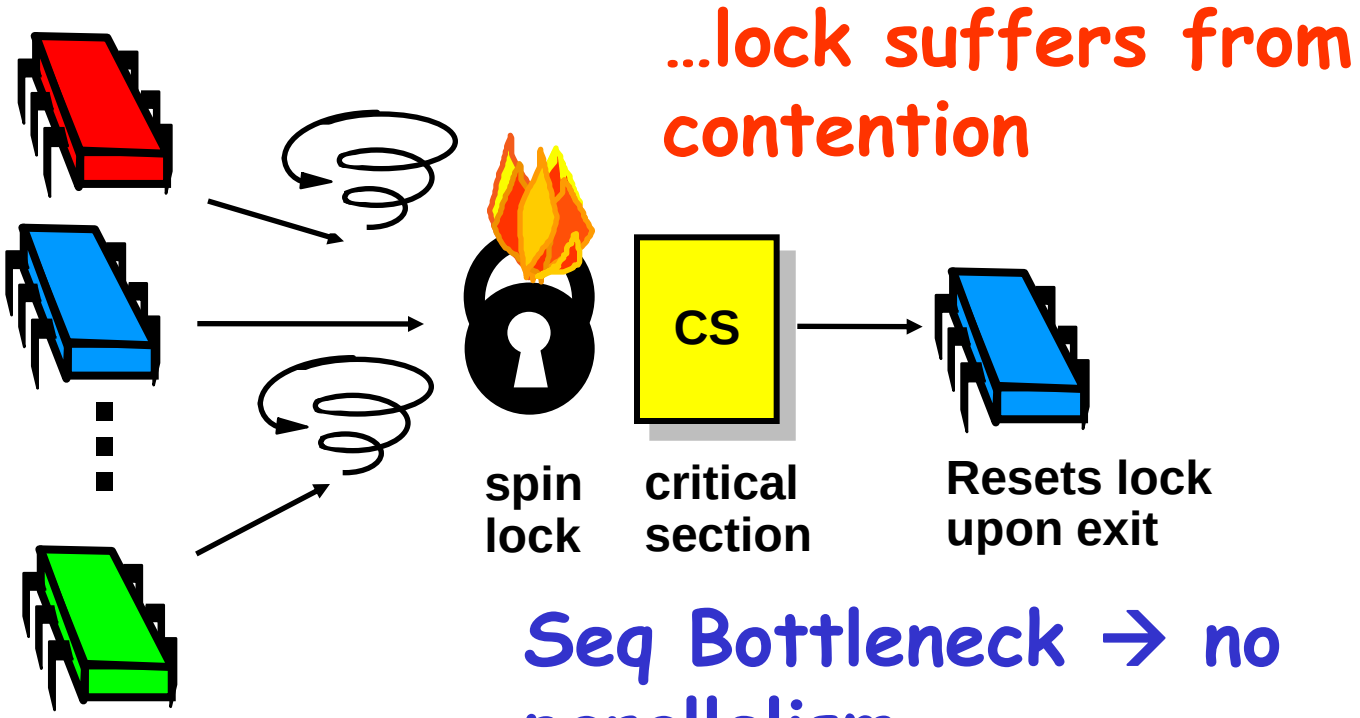


Basic Spin-Lock



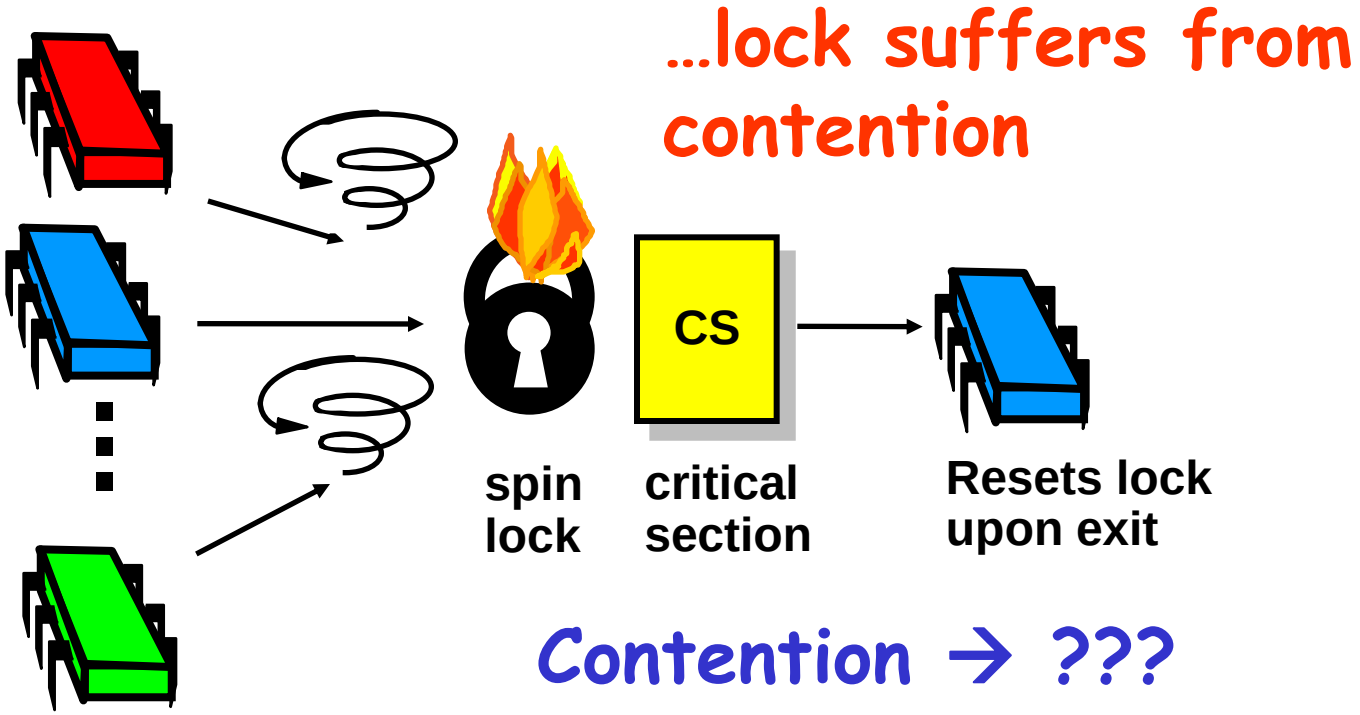
Notice: these are distinct phenomena

Basic Spin-Lock



Seq Bottleneck → no parallelism

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

```
public class AtomicBoolean {
 boolean value;

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

Review: Test-and-Set

```
public class AtomicBoolean {
  boolean value;

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

Package
java.util.concurrent.atomic

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

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

Swap old and new values

Review: Test-and-Set

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

Review: Test-and-Set

```
AtomicBoolean lock
 = new AtomicBoolean(false)

...
boolean 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 {
 AtomicBoolean state =
  new AtomicBoolean(false);

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

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

Test-and-set Lock

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

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

 void unlock() {
  state
 }}
```

Lock state is AtomicBoolean

Test-and-set Lock

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

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

 void unlock() {
  sta
 }}
```

Keep trying until lock acquired

Test-and-set Lock

```
class TA
 AtomicB
  new At

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

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

Release lock by resetting state to false

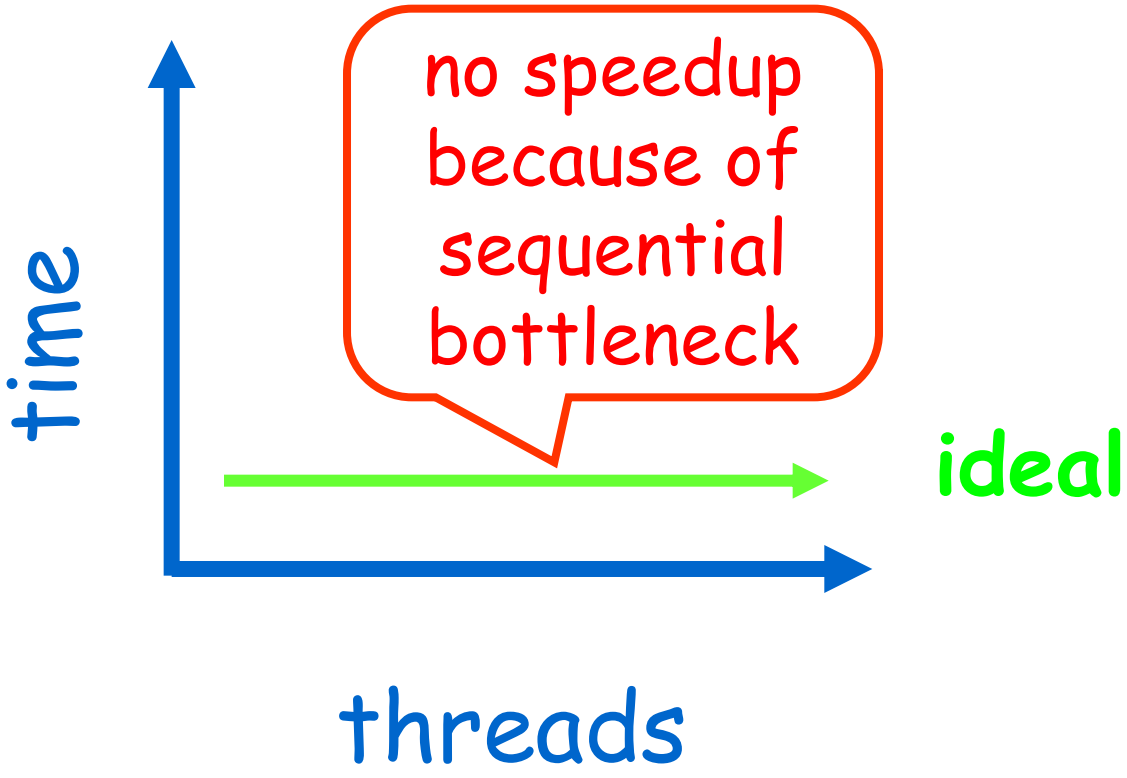
Space Complexity

- TAS spin-lock has small "footprint"
- N thread spin-lock uses $O(1)$ space
- As opposed to $O(n)$ in solutions that keep record of who else is interested (we'll see later)

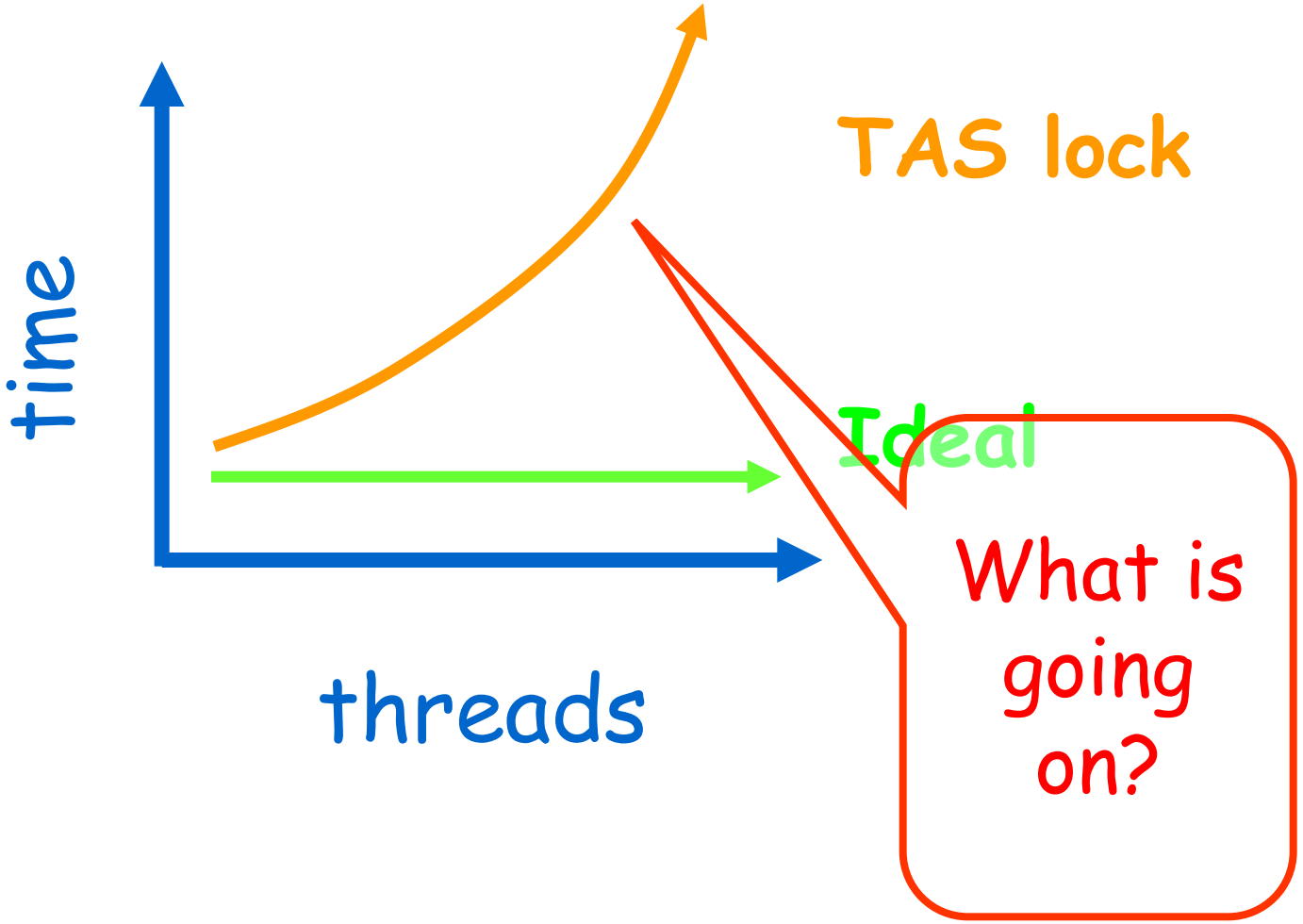
Performance

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

Graph


no speedup
because of
sequential
bottleneck
time
ideal
threads

Mystery #1



Test-and-Test-and-Set Locks

Main idea:

Split the following lock line to two

```
while (state.getAndSet(true)) {}
```

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 {
 AtomicBoolean state =
   new AtomicBoolean(false);

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

Test-and-test-and-set Lock

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

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

Wait until lock looks free

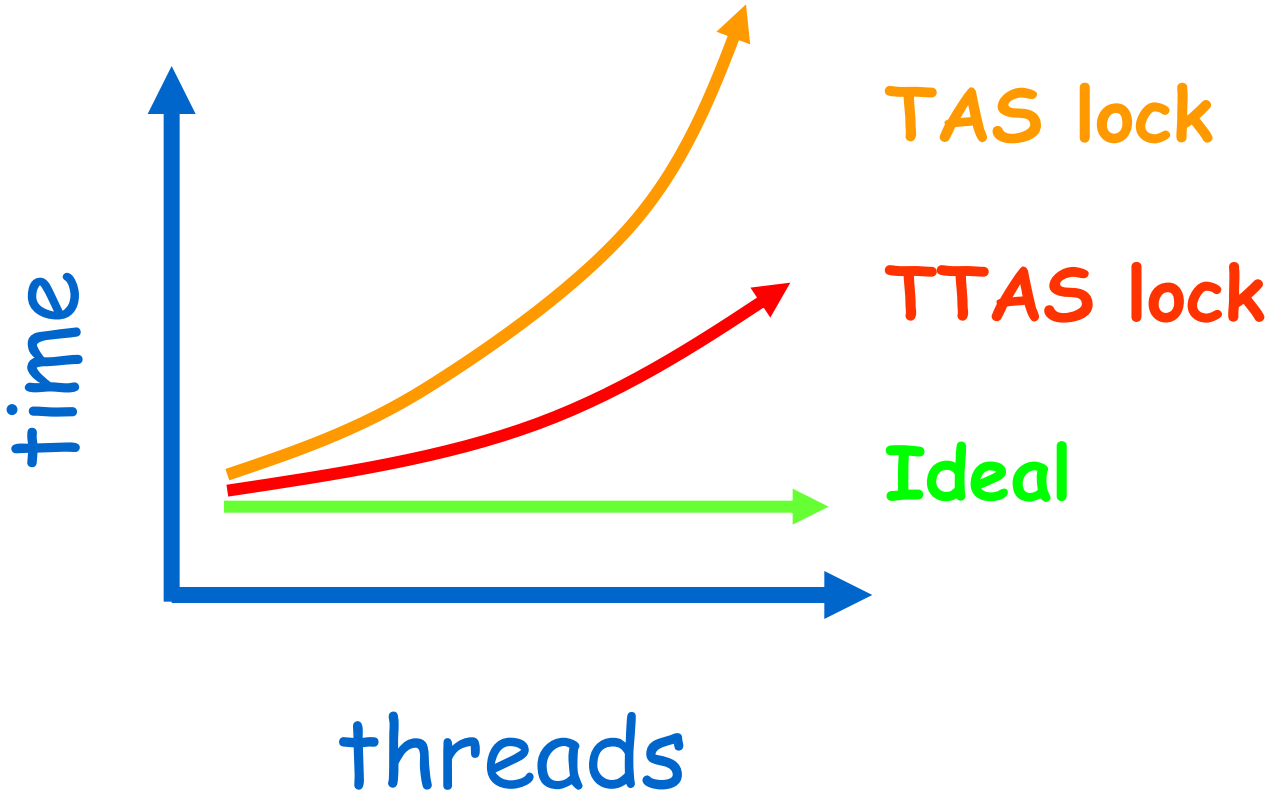
Test-and-test-and-set Lock

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

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

Then try to acquire it

Mystery #2



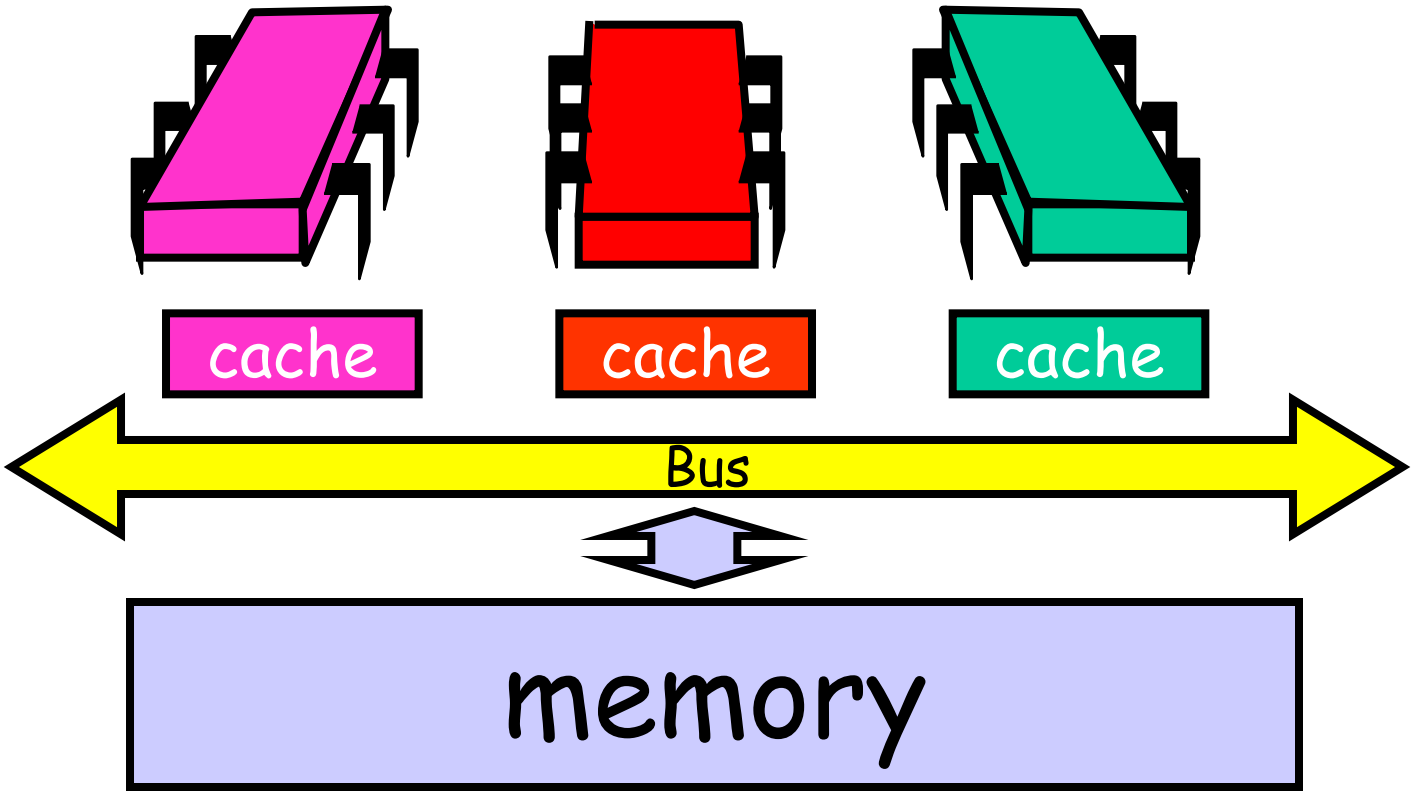
Mystery

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

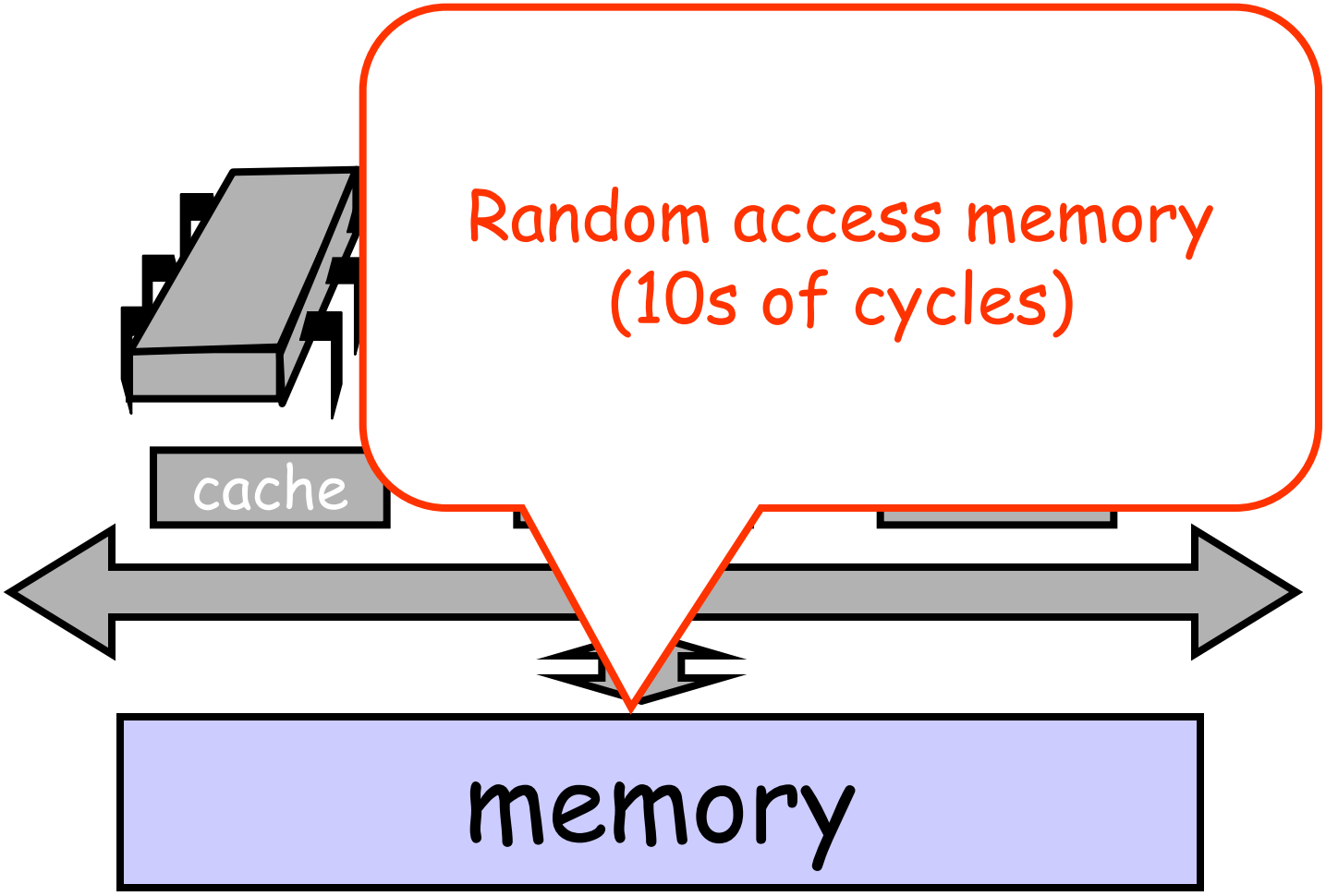
Opinion

- 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



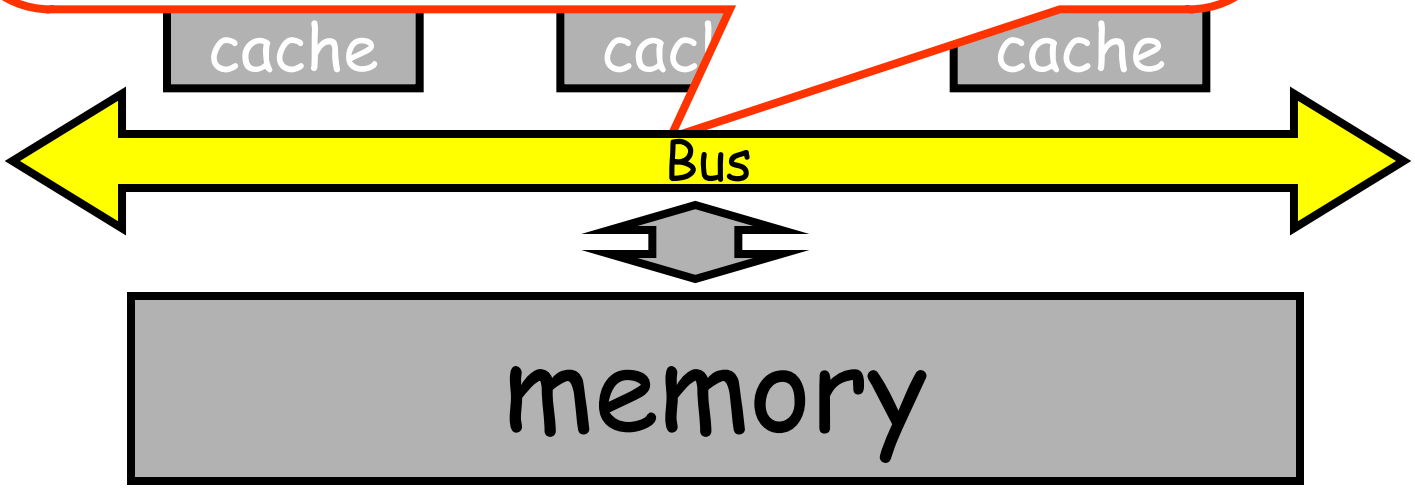
Bus-Based Architectures



Bus-Based Architectures

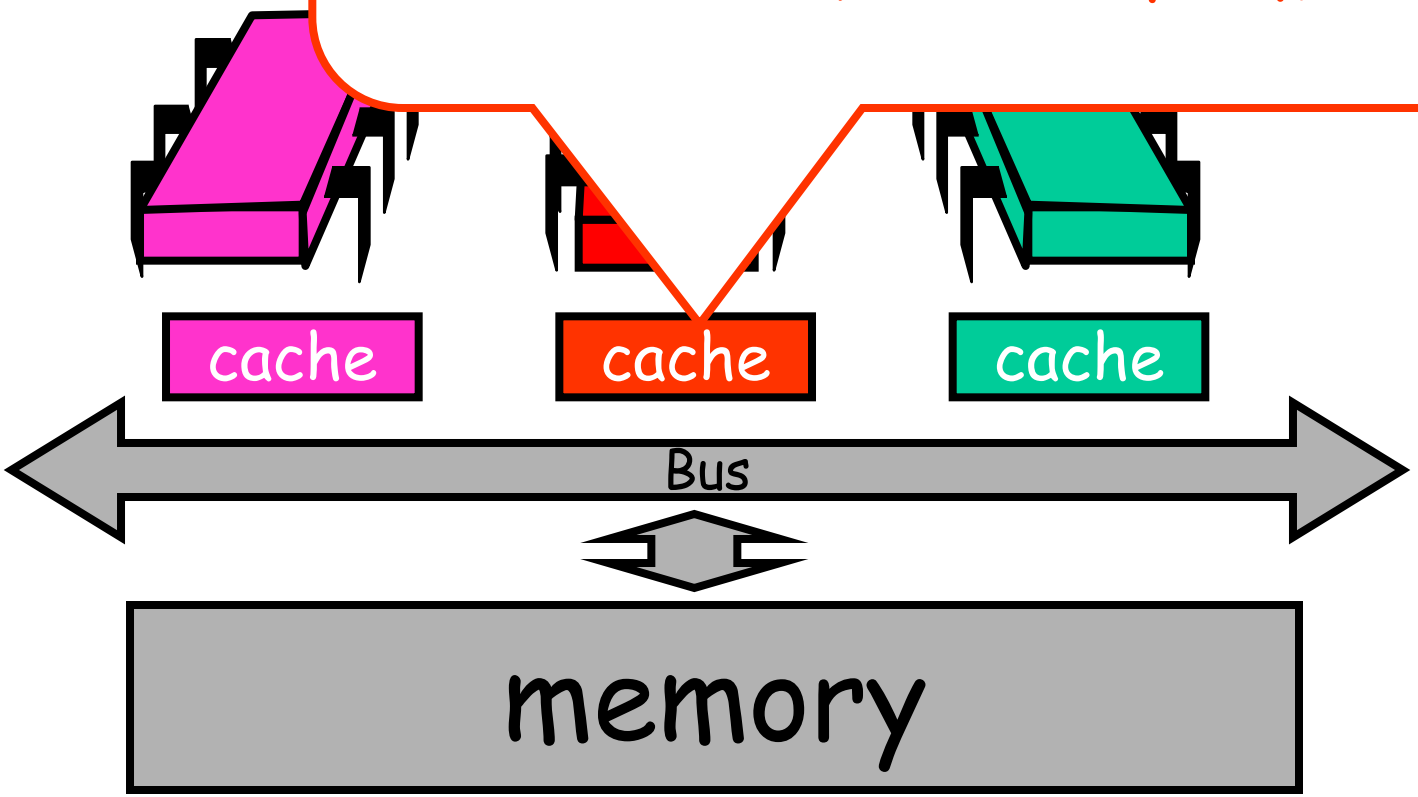
Shared Bus

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



Bus-B

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



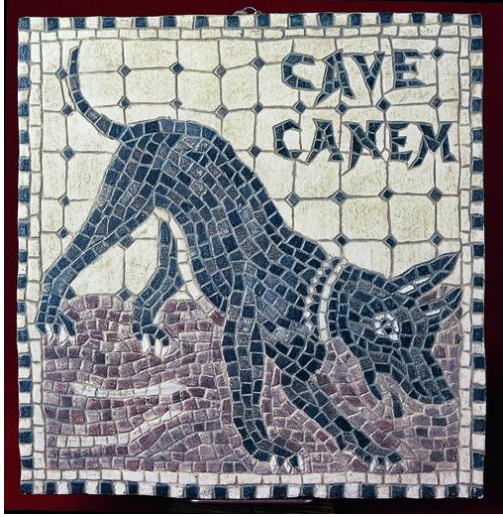
Jargon Watch

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

Jargon Watch

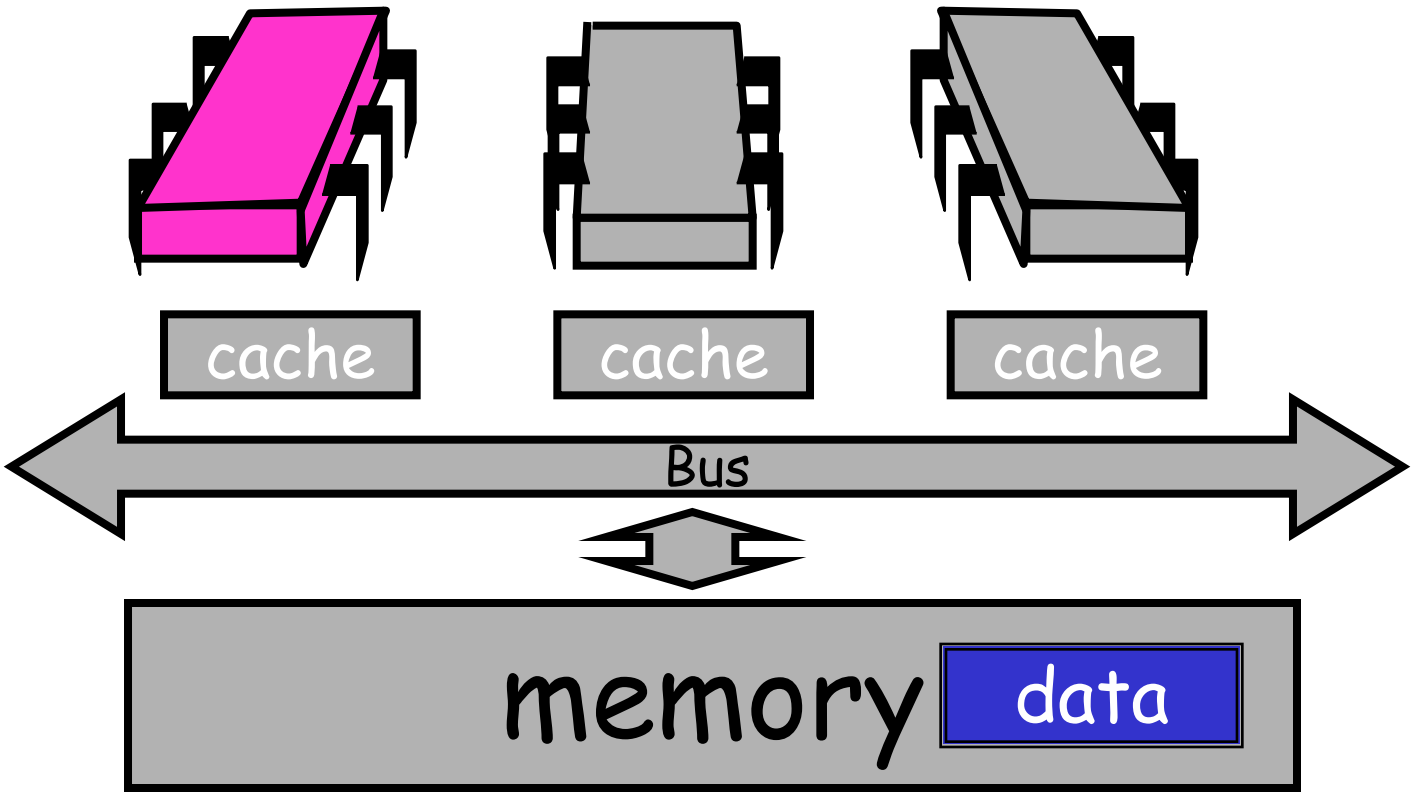
- Cache hit
  - "I found what I wanted in my cache"
  - Good Thing™
- Cache miss
  - "I had to shlep all the way to memory for that data"
  - Bad Thing™

Cave Canem

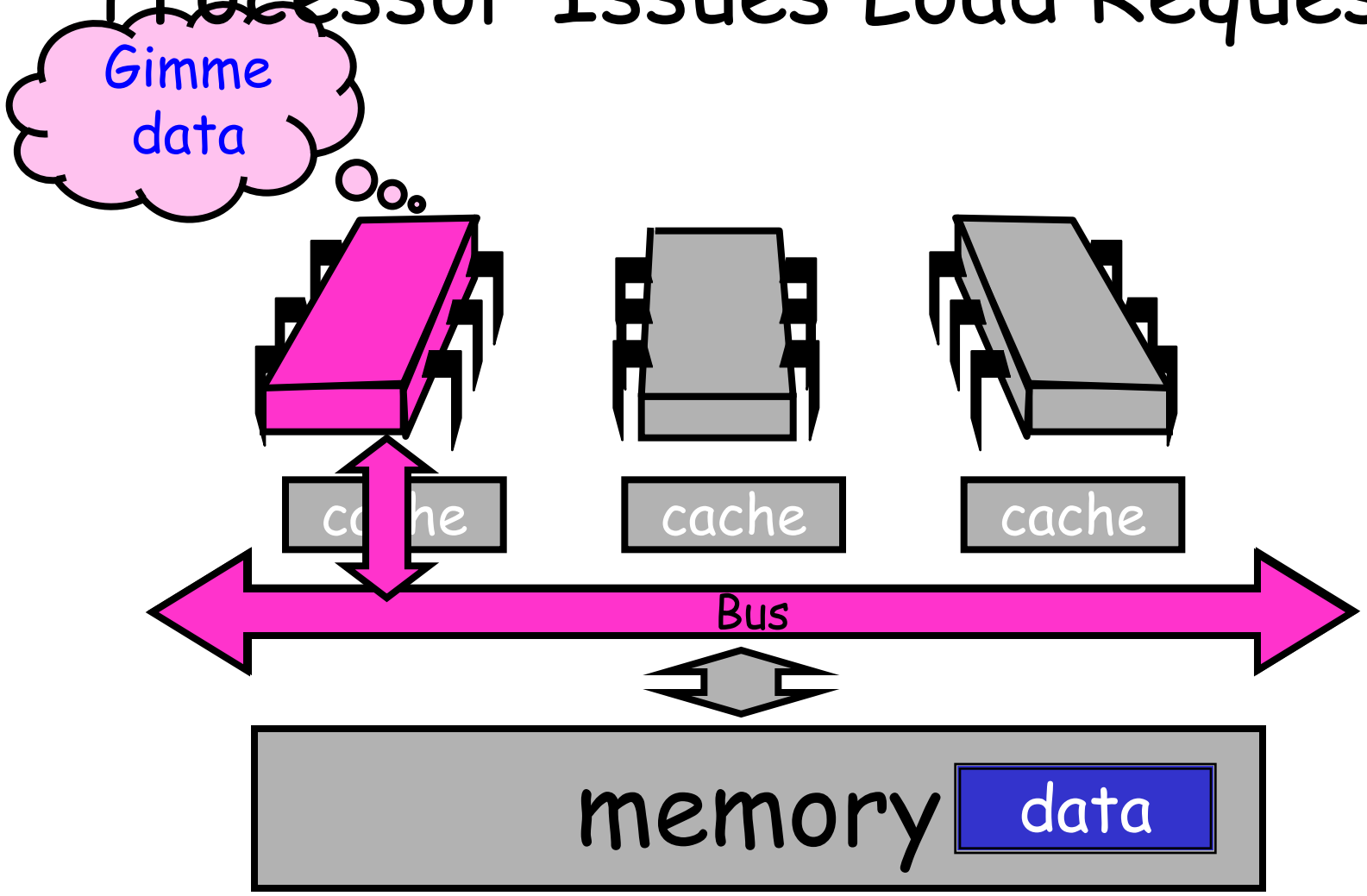


- This model is still a simplification
  - But not in any essential way
  - Illustrates basic principles

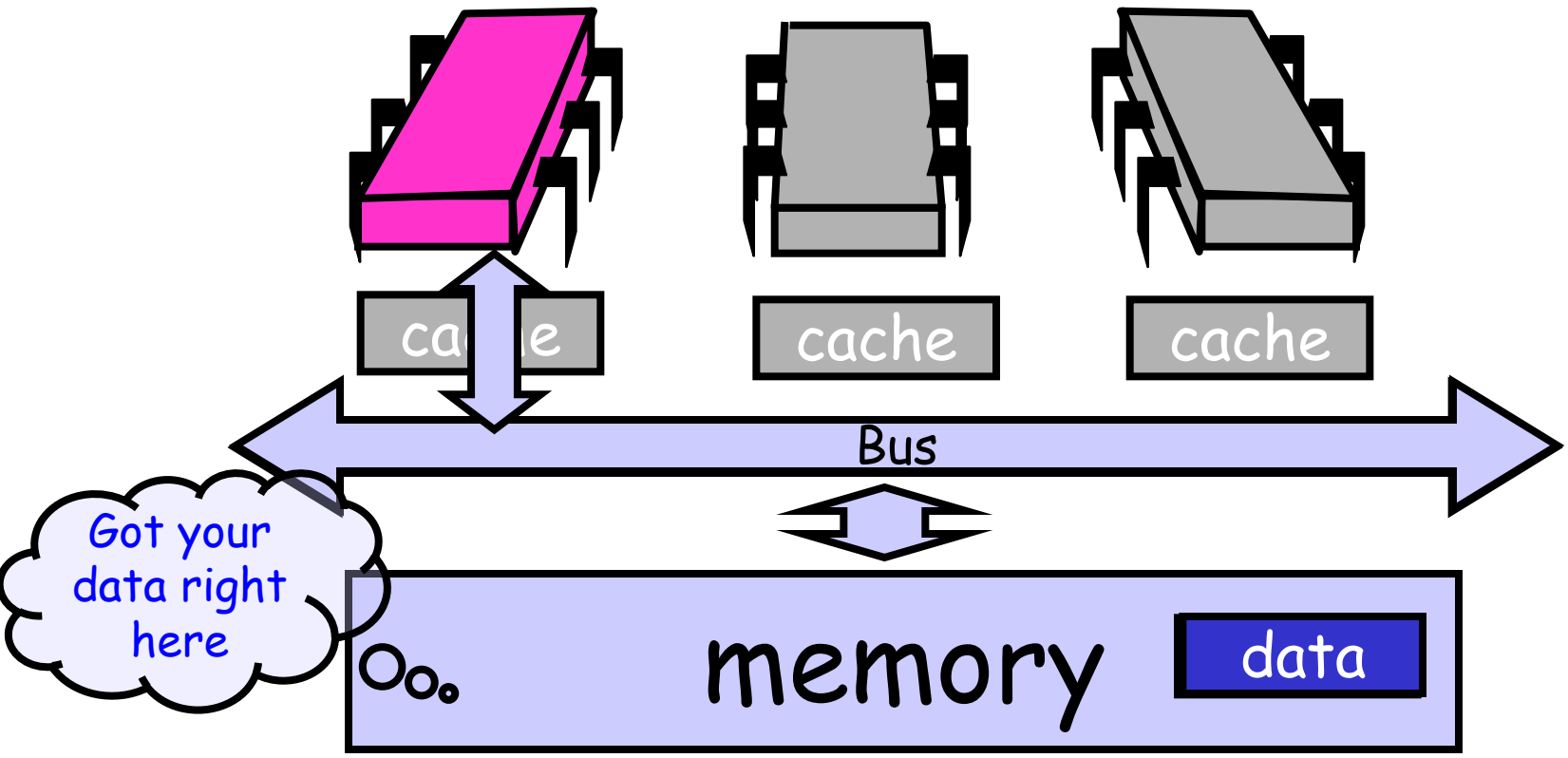
Processor Issues Load Request



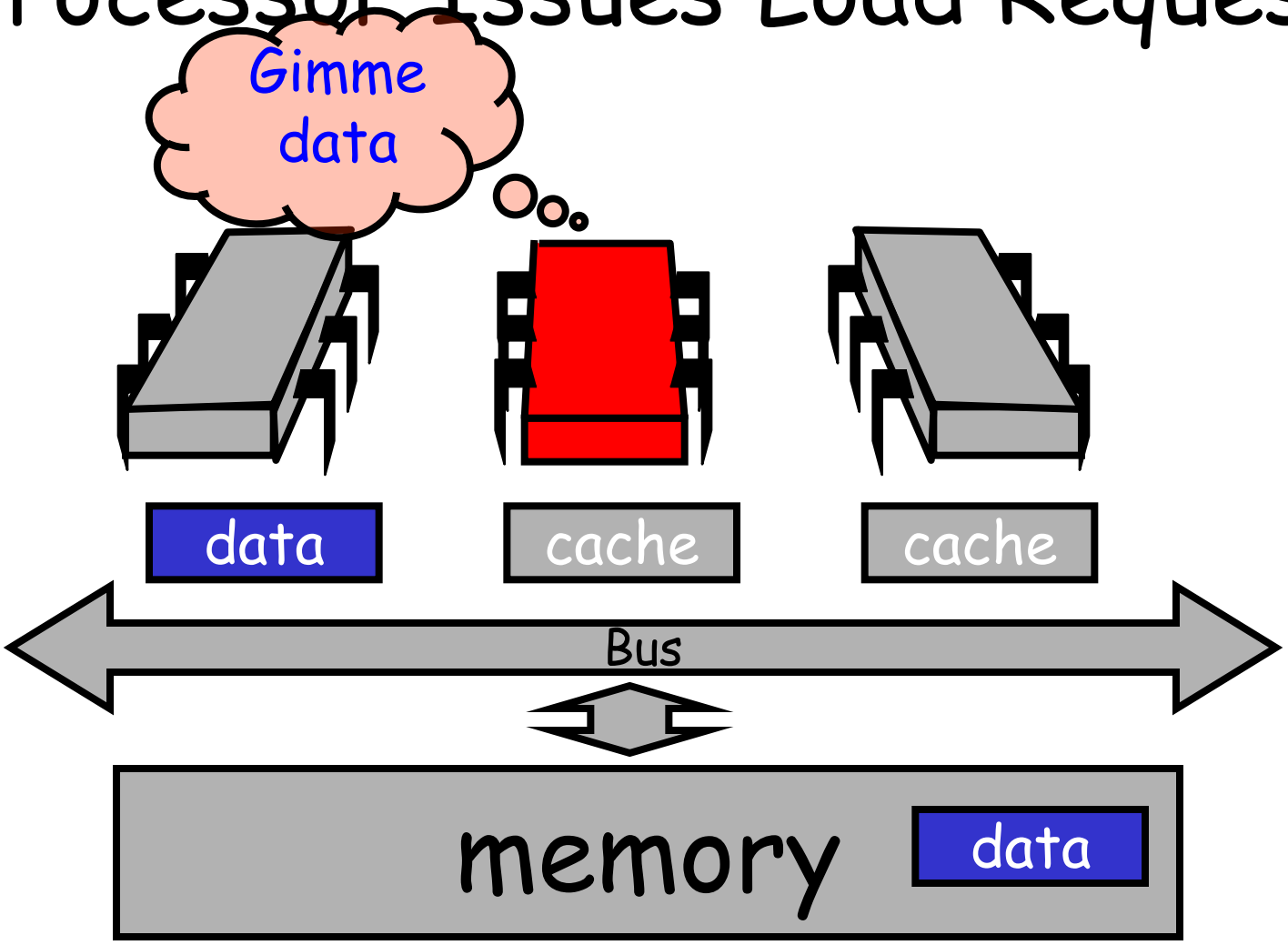
Processor Issues Load Request



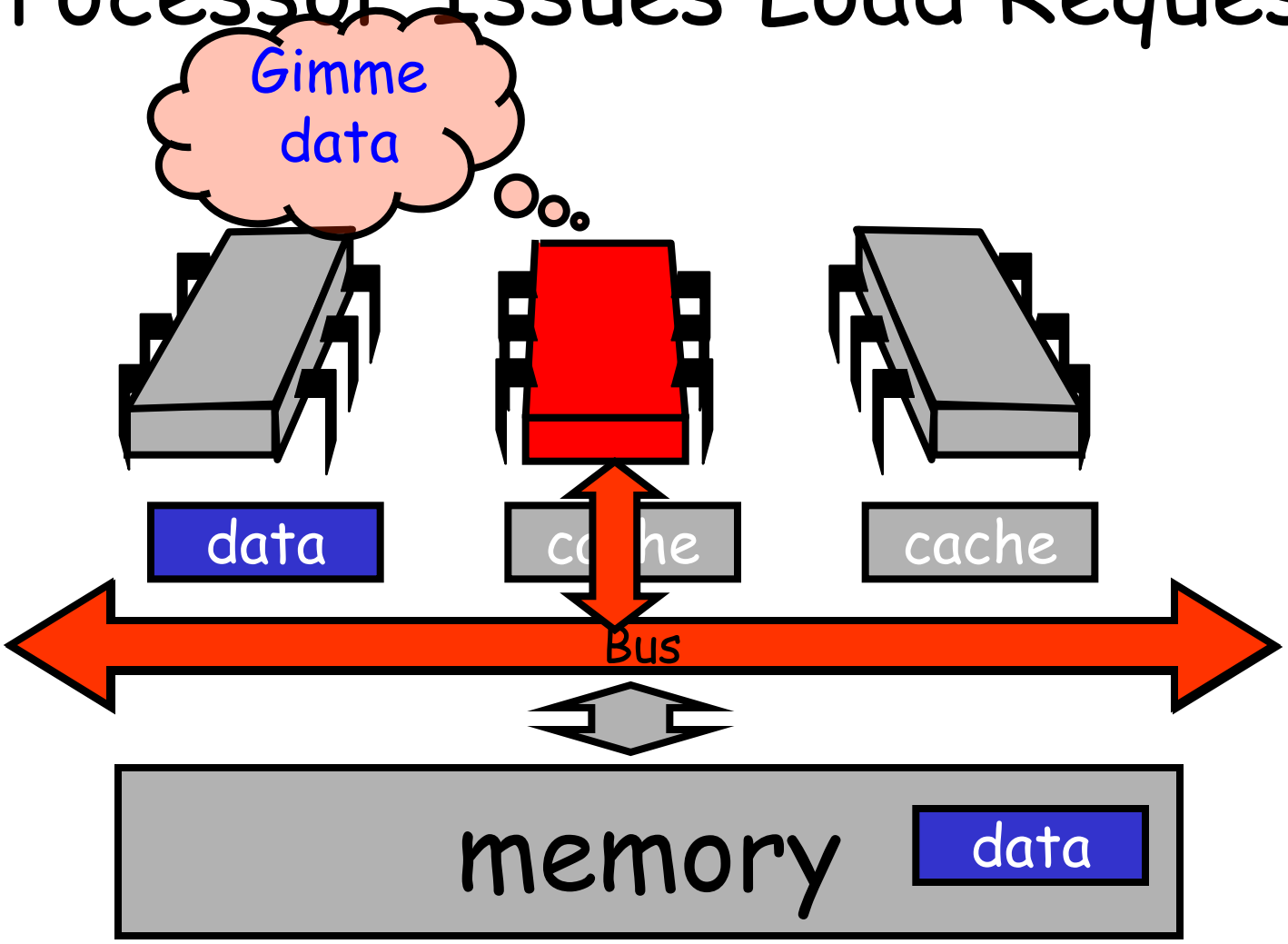
Memory Responds



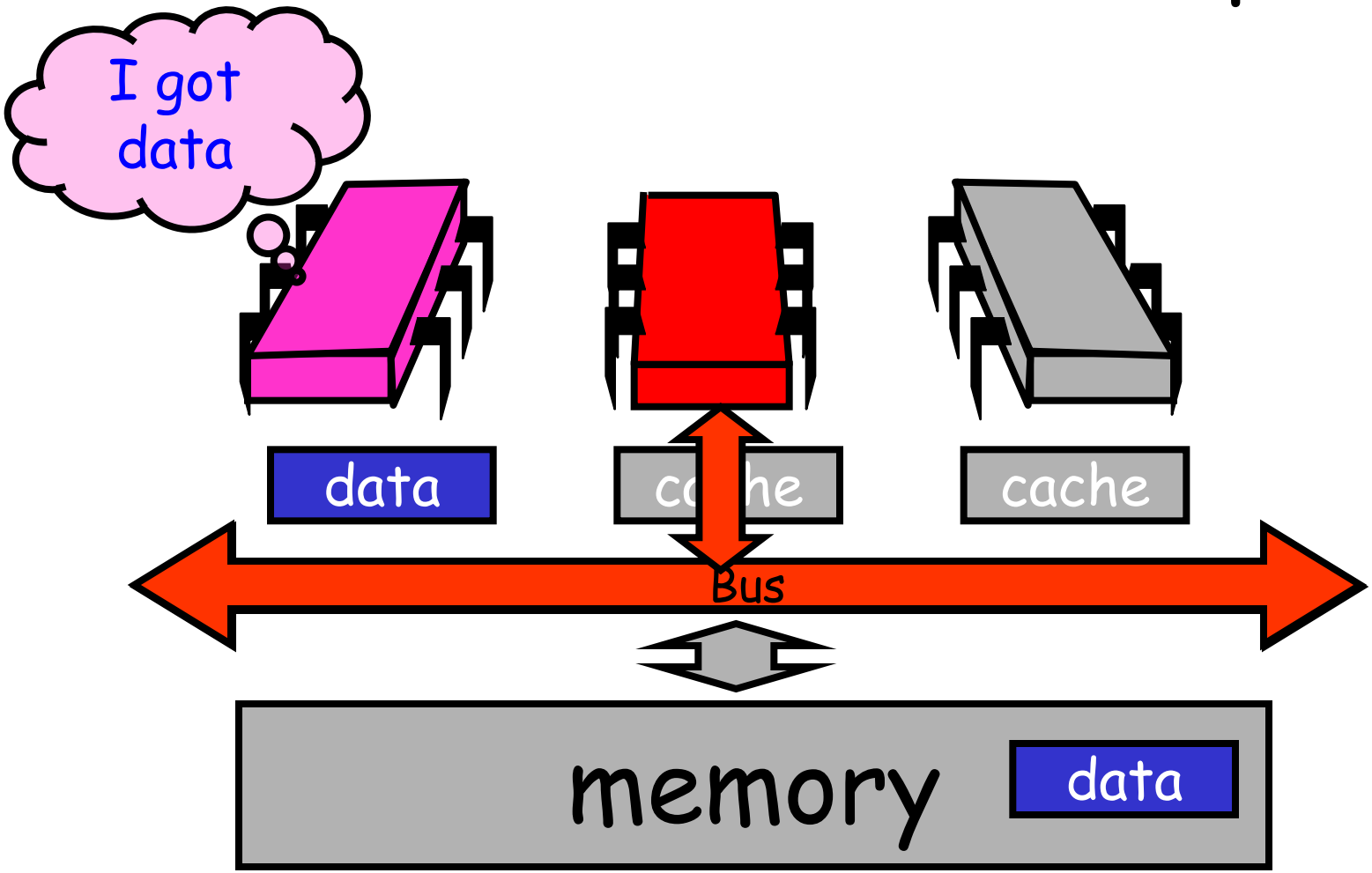
Processor Issues Load Request



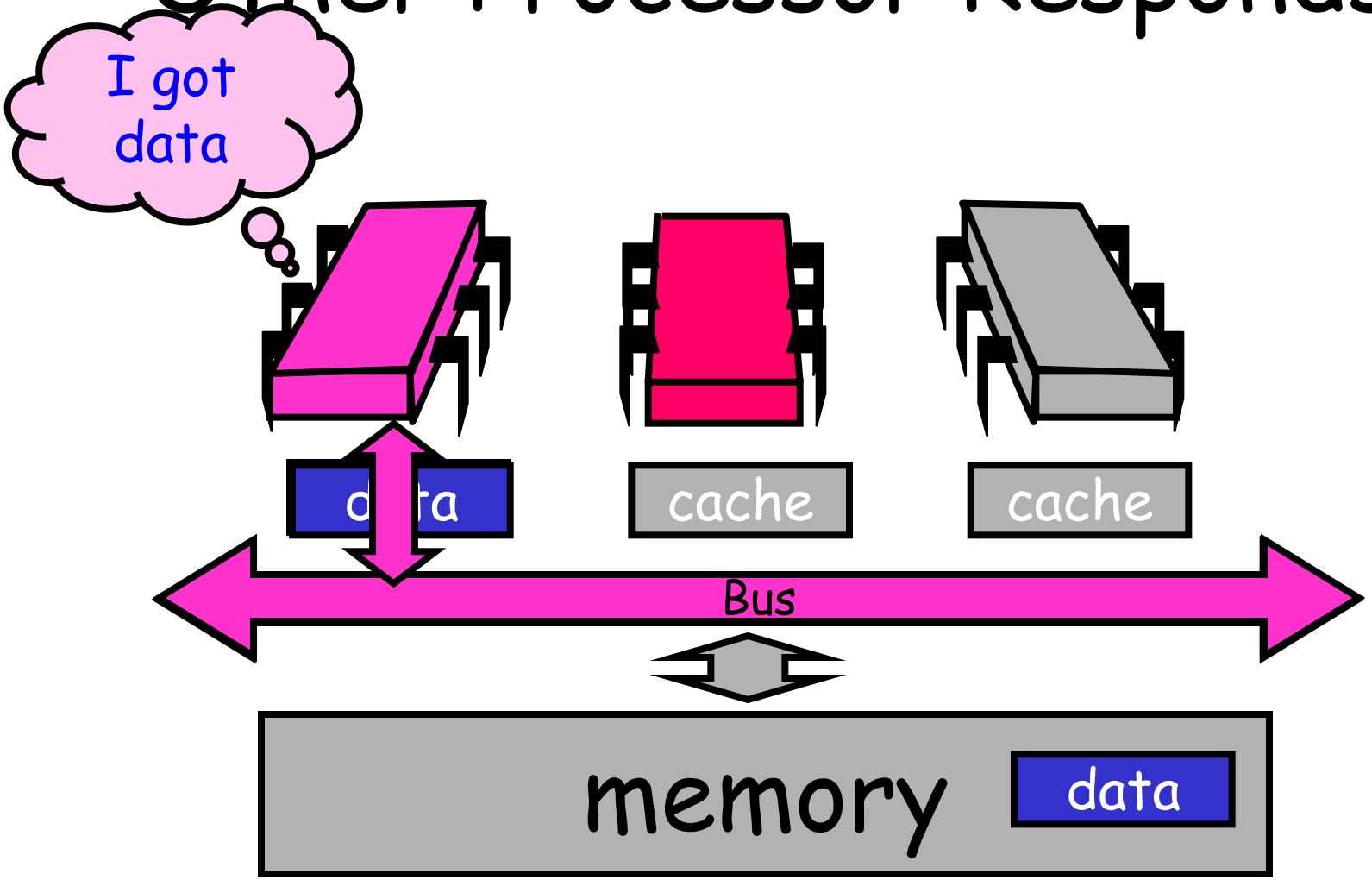
Processor Issues Load Request



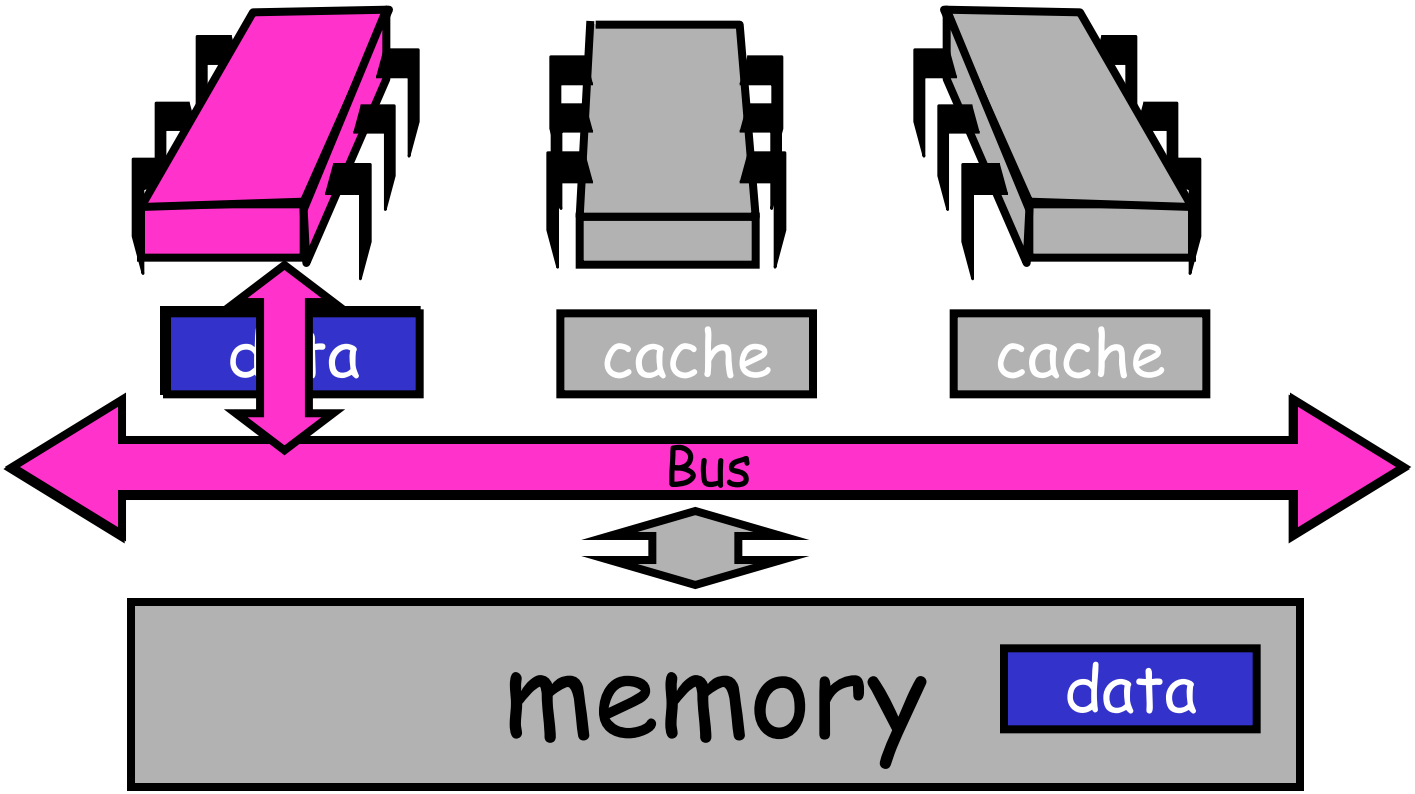
Processor Issues Load Request



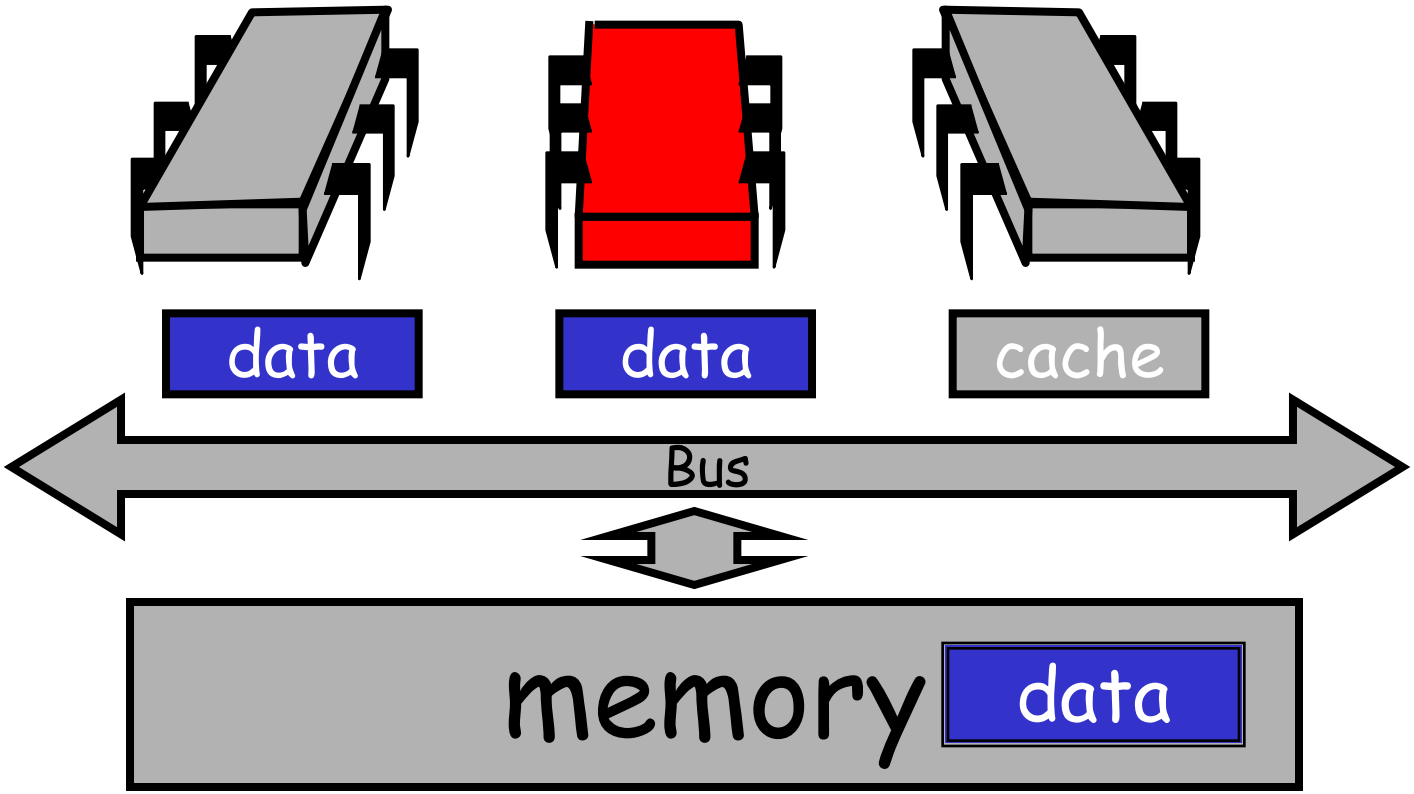
Other Processor Responds



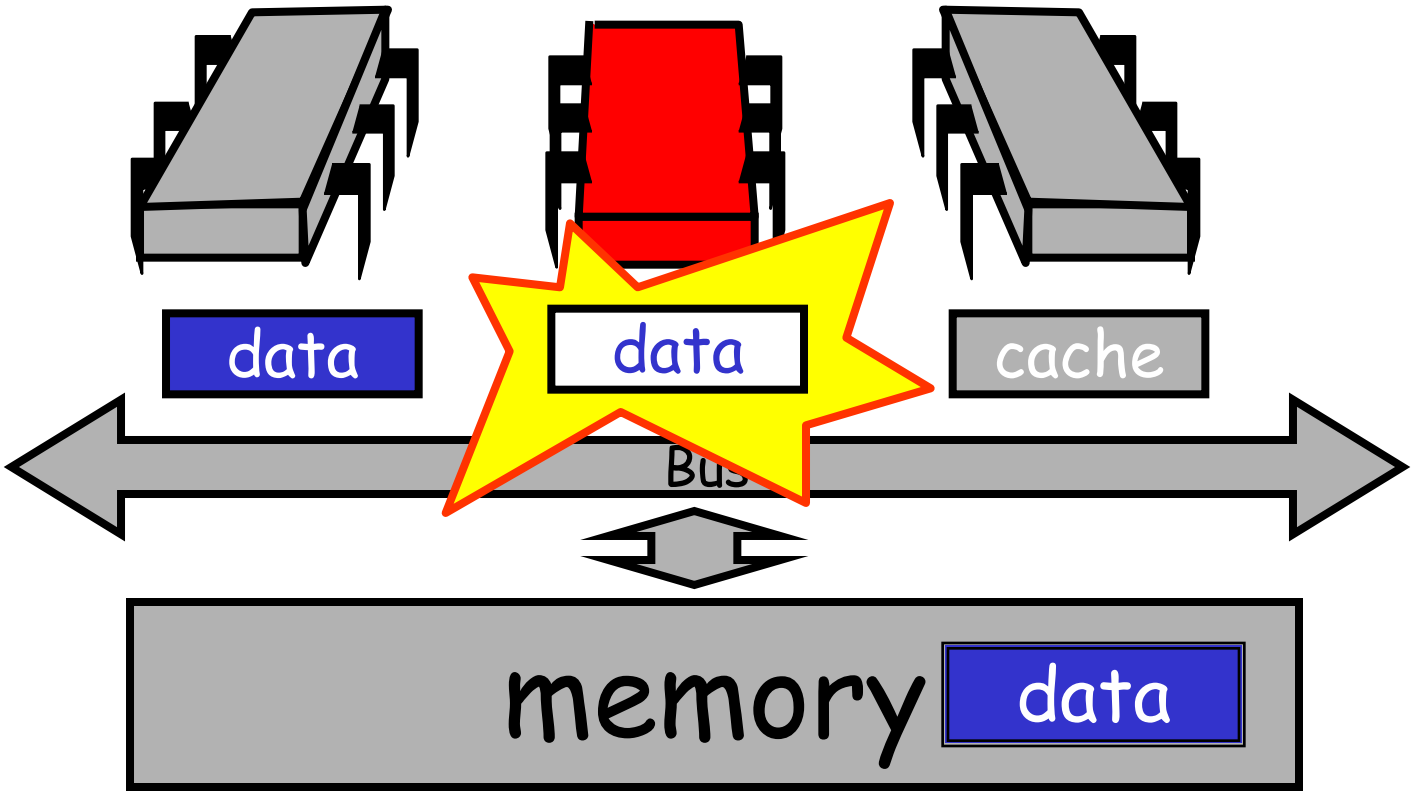
Other Processor Responds



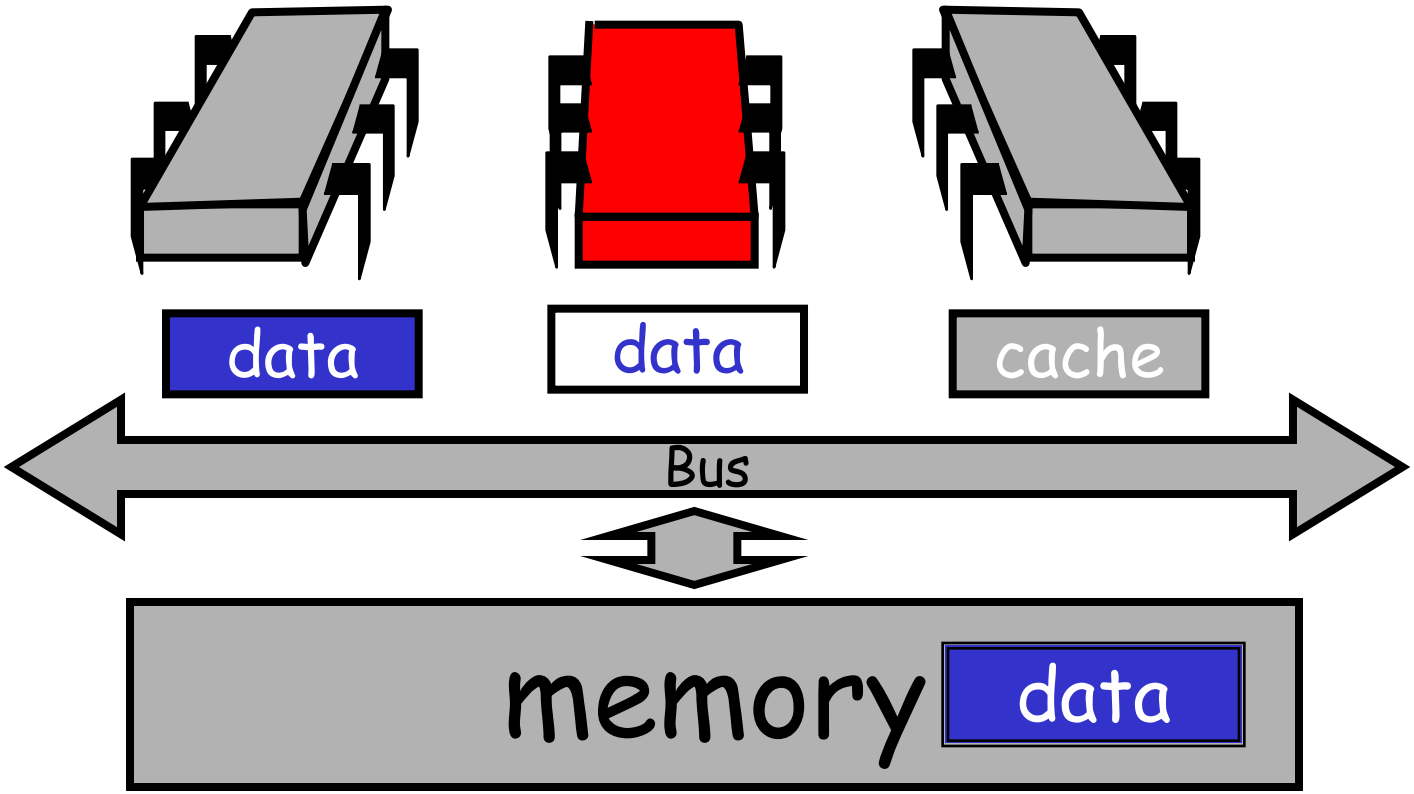
Modify Cached Data



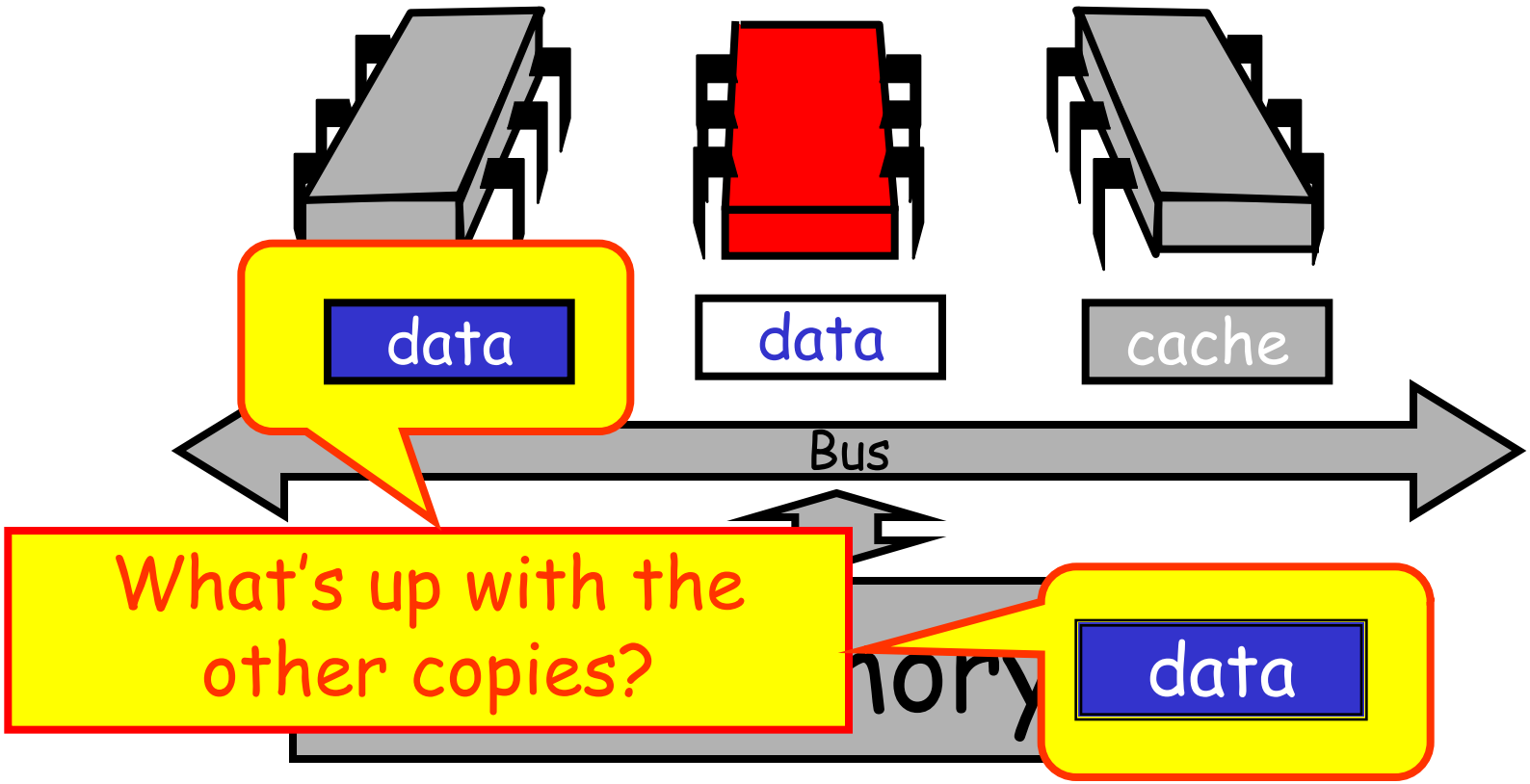
Modify Cached Data



Modify Cached Data



Modify Cached 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?

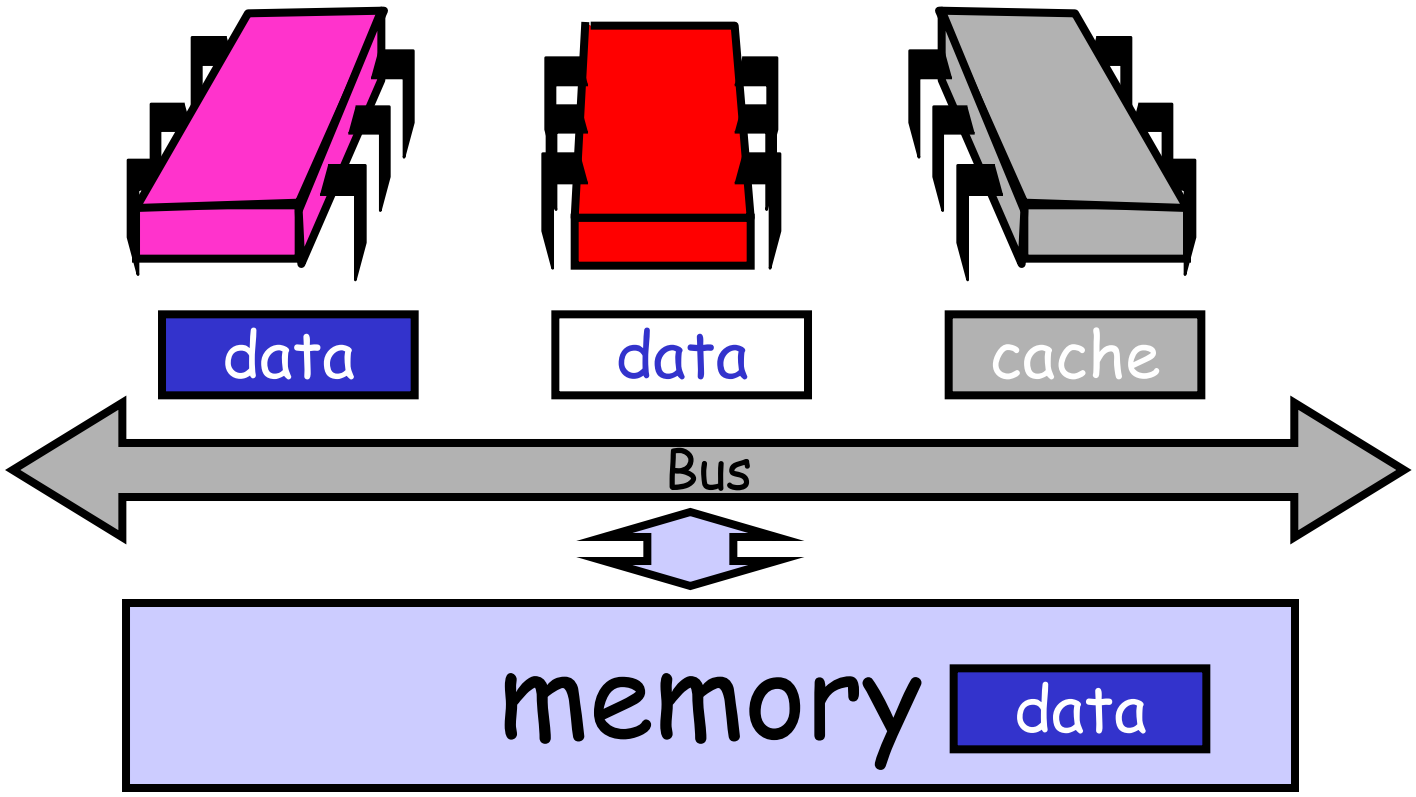
Write-Back Caches

- 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 …

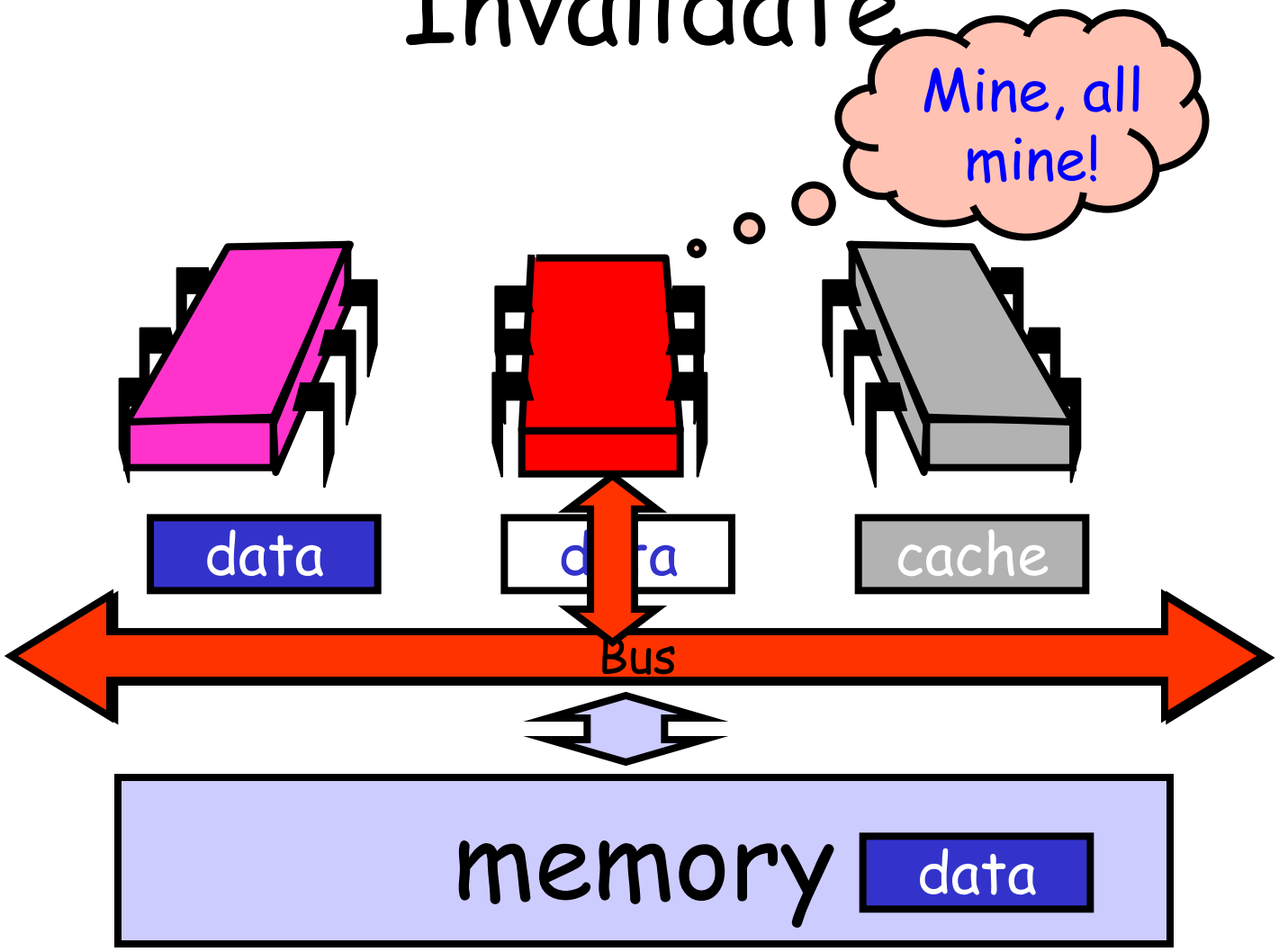
Write-Back Caches

- Cache entry has three states
  - Invalid: meaningless content
  - Valid: I can read but I can't write
    (may be cached elsewhere)
  - Dirty: Data has been modified
    - Intercept other load requests
    - Write back to memory before using cache

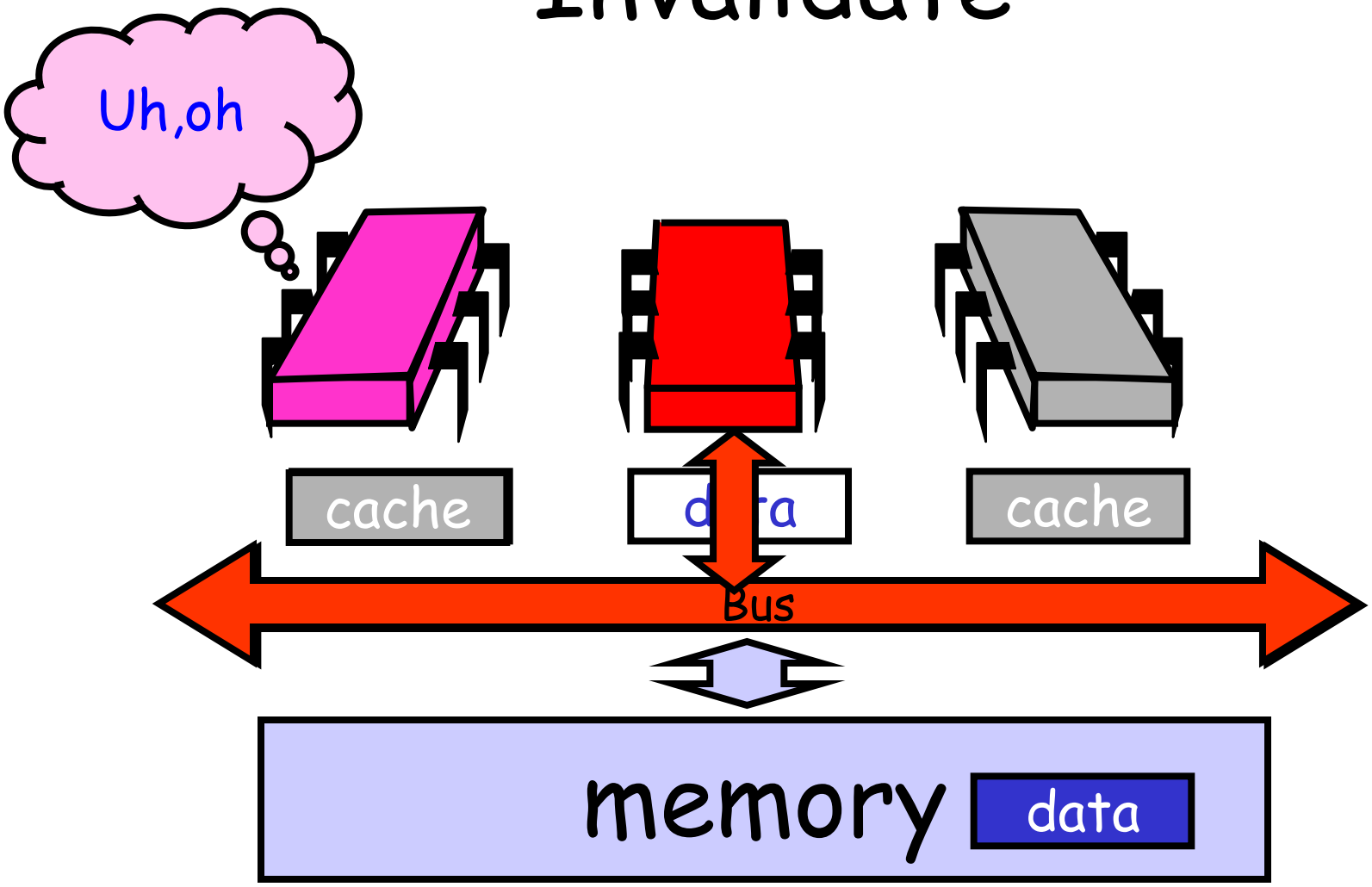
Invalidate



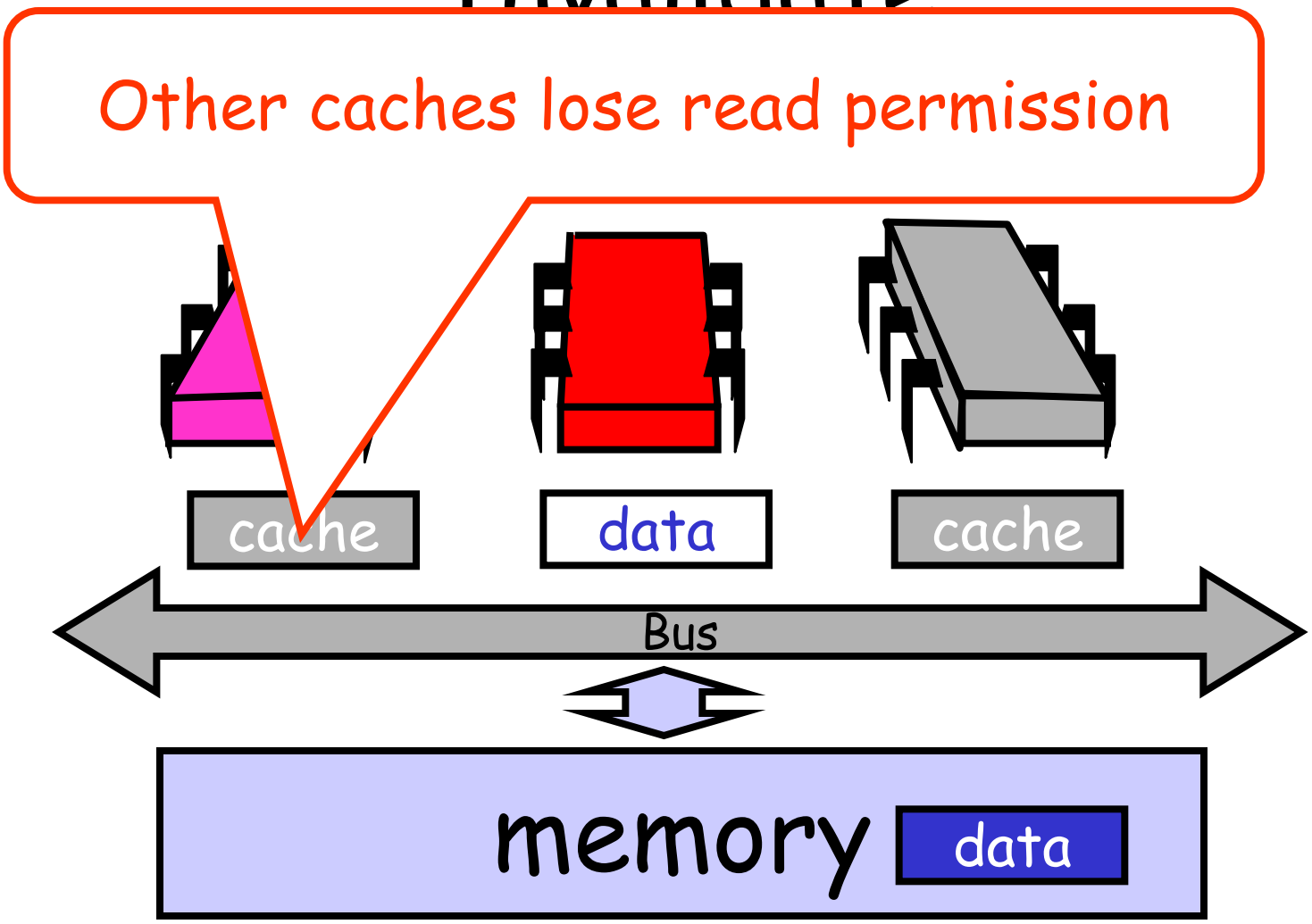
Invalidate



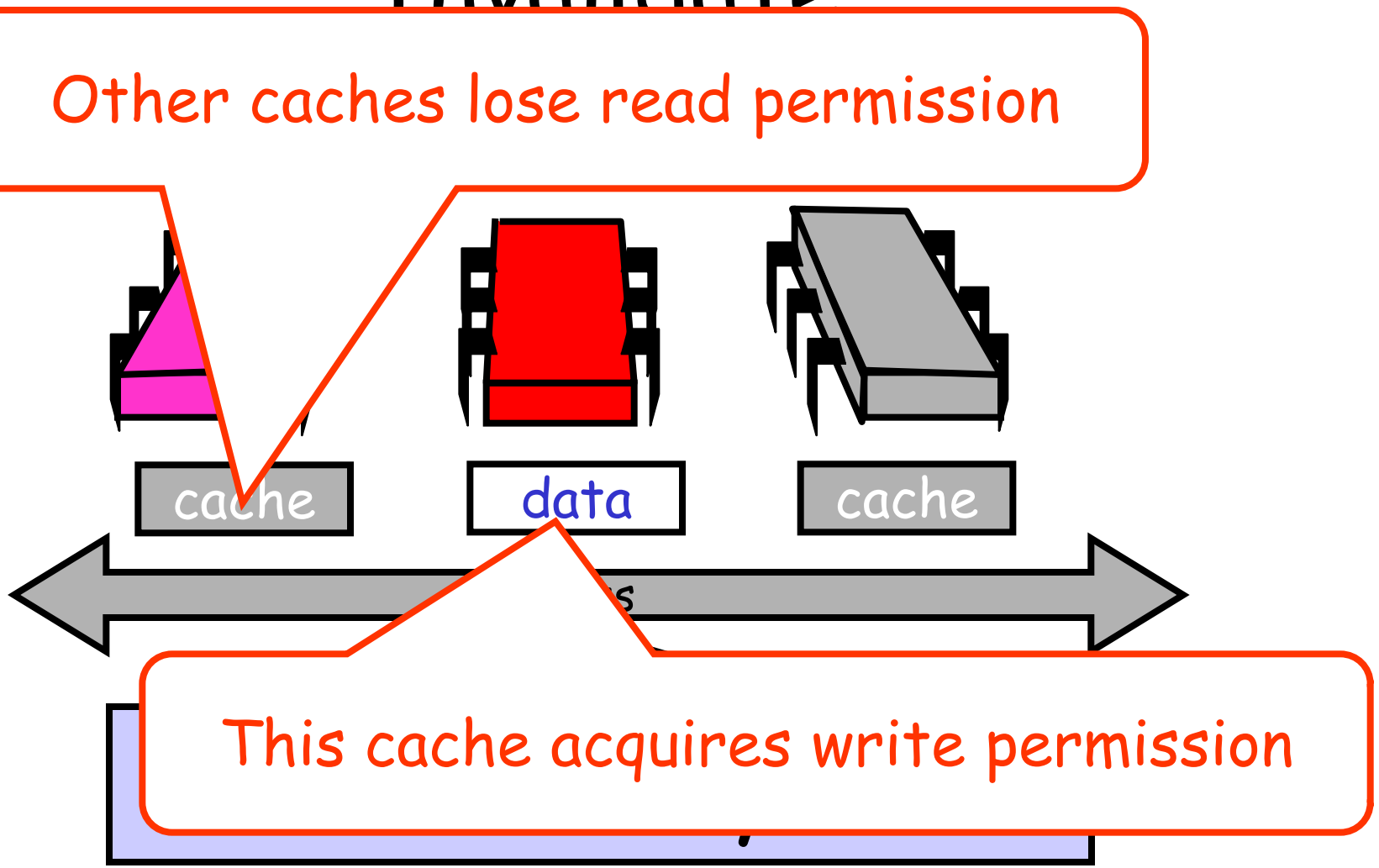
Invalidate



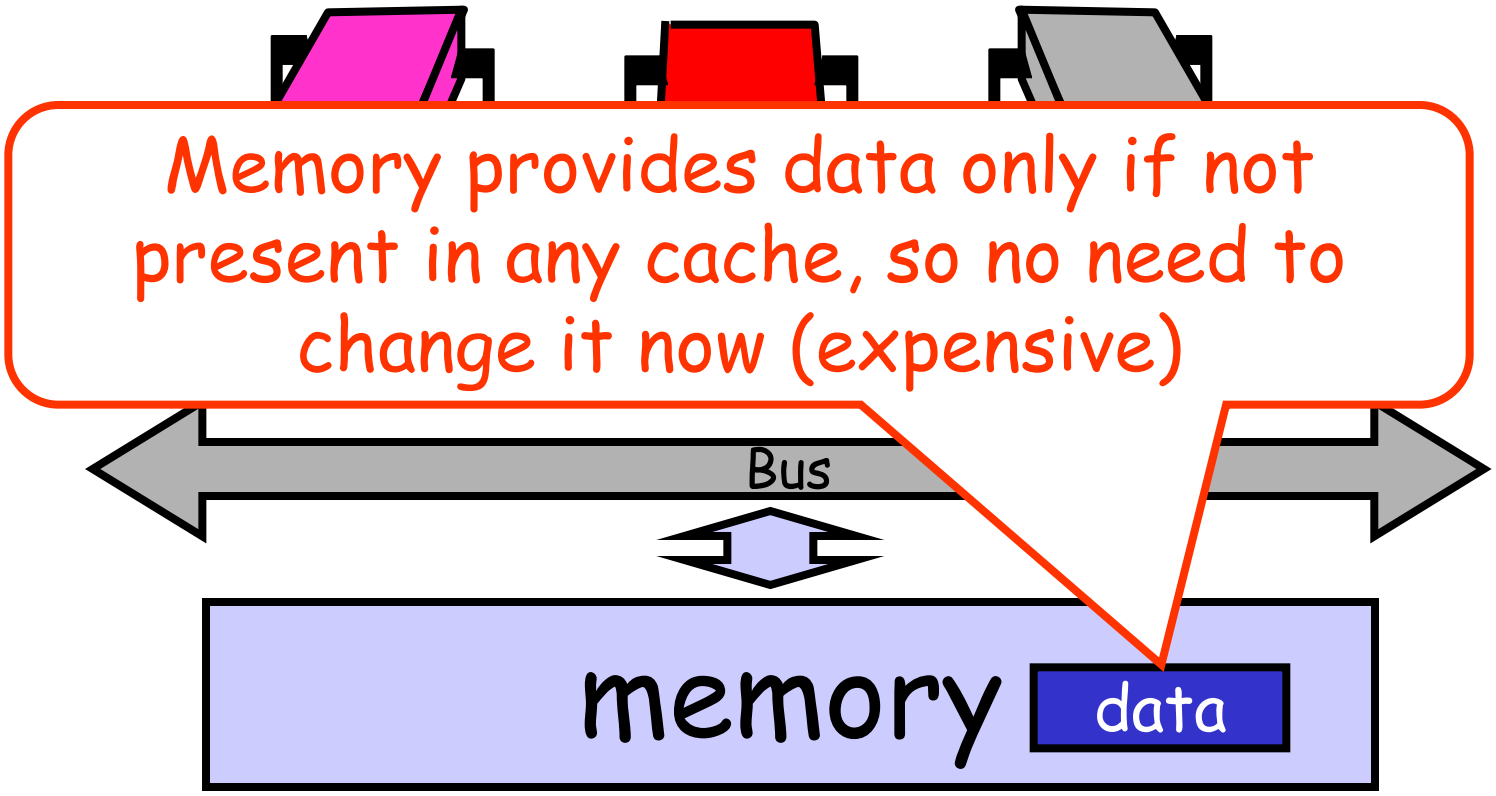
Invalidate


Other caches lose read permission
cache
data
cache
Bus
memory
data

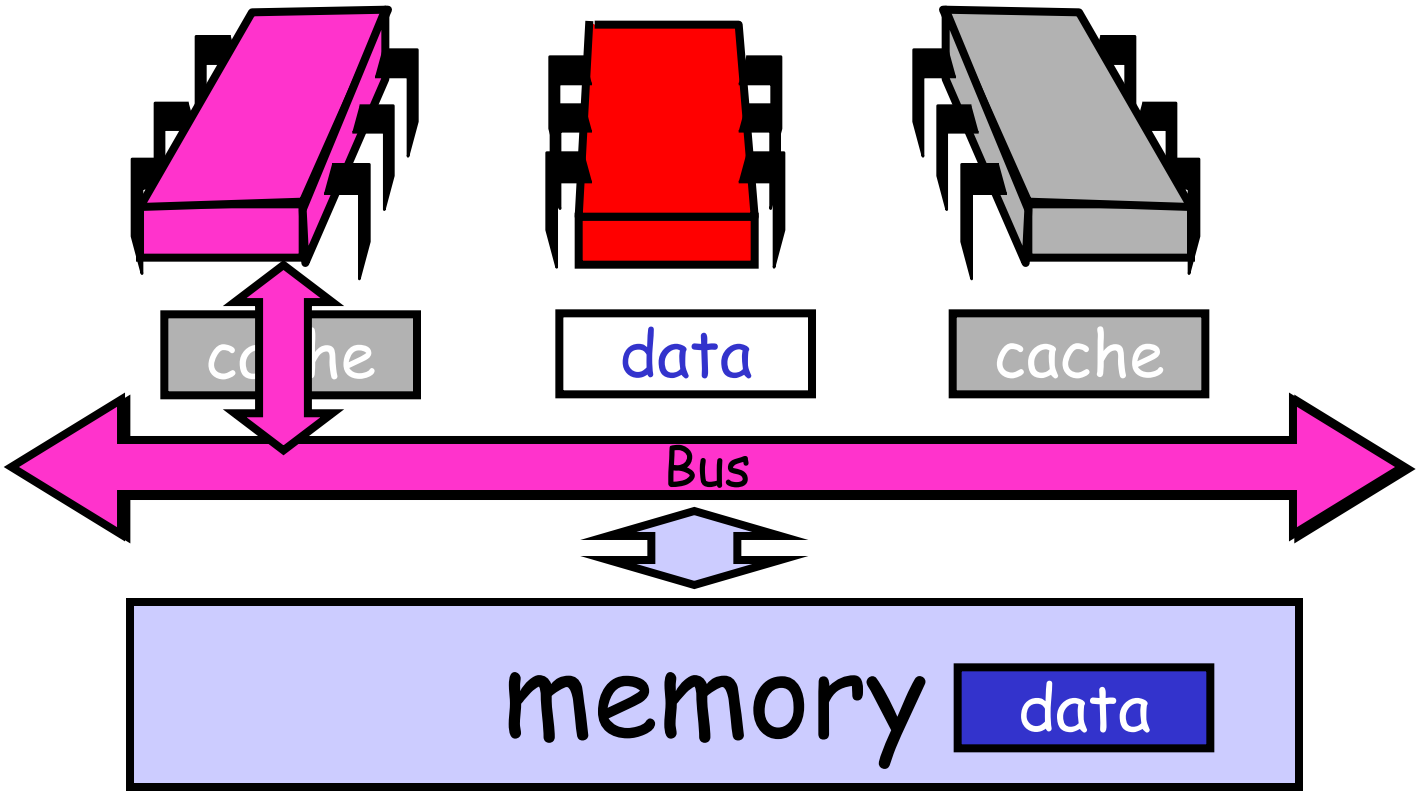
Invalidate


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

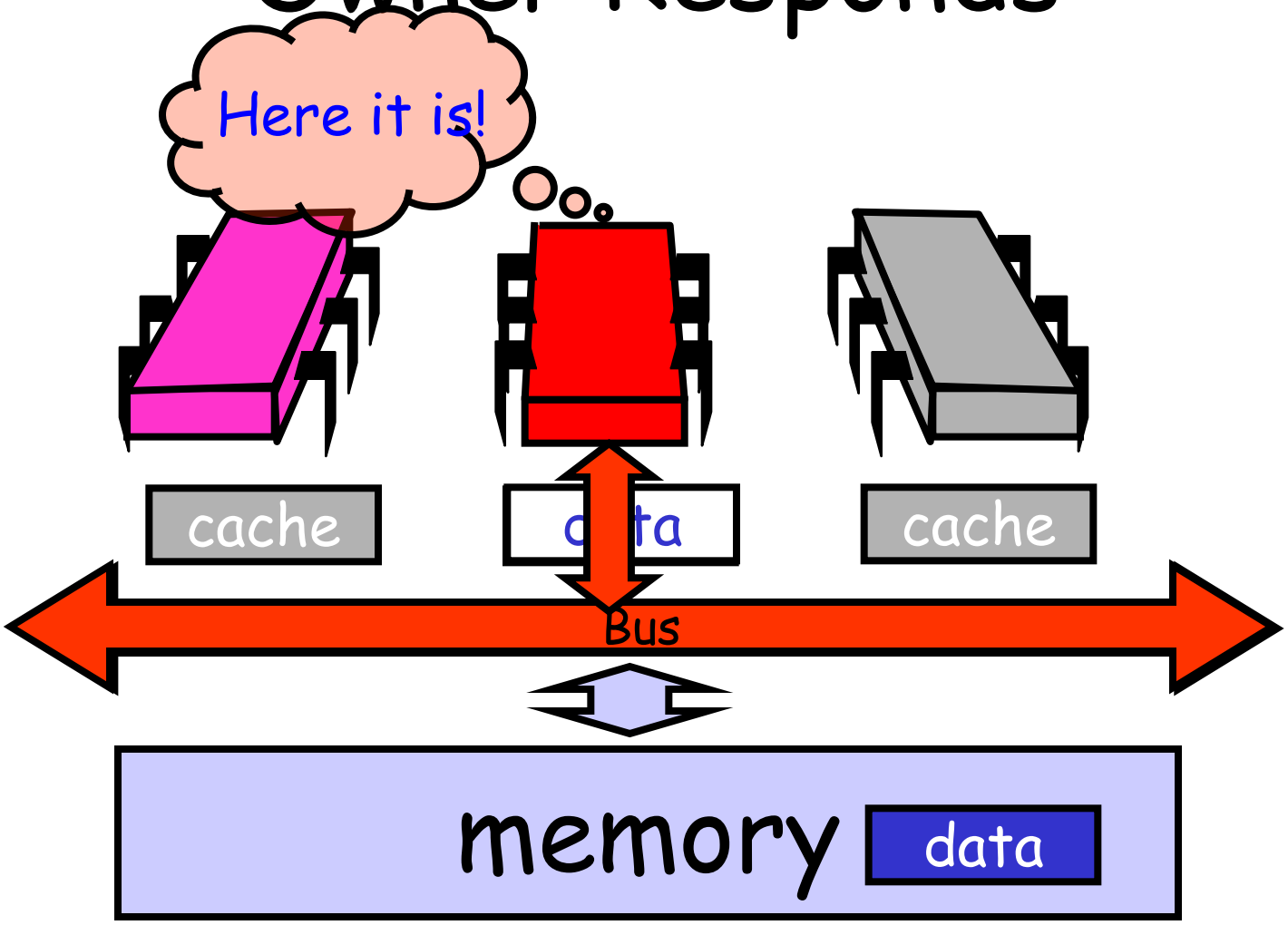
Invalidate


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

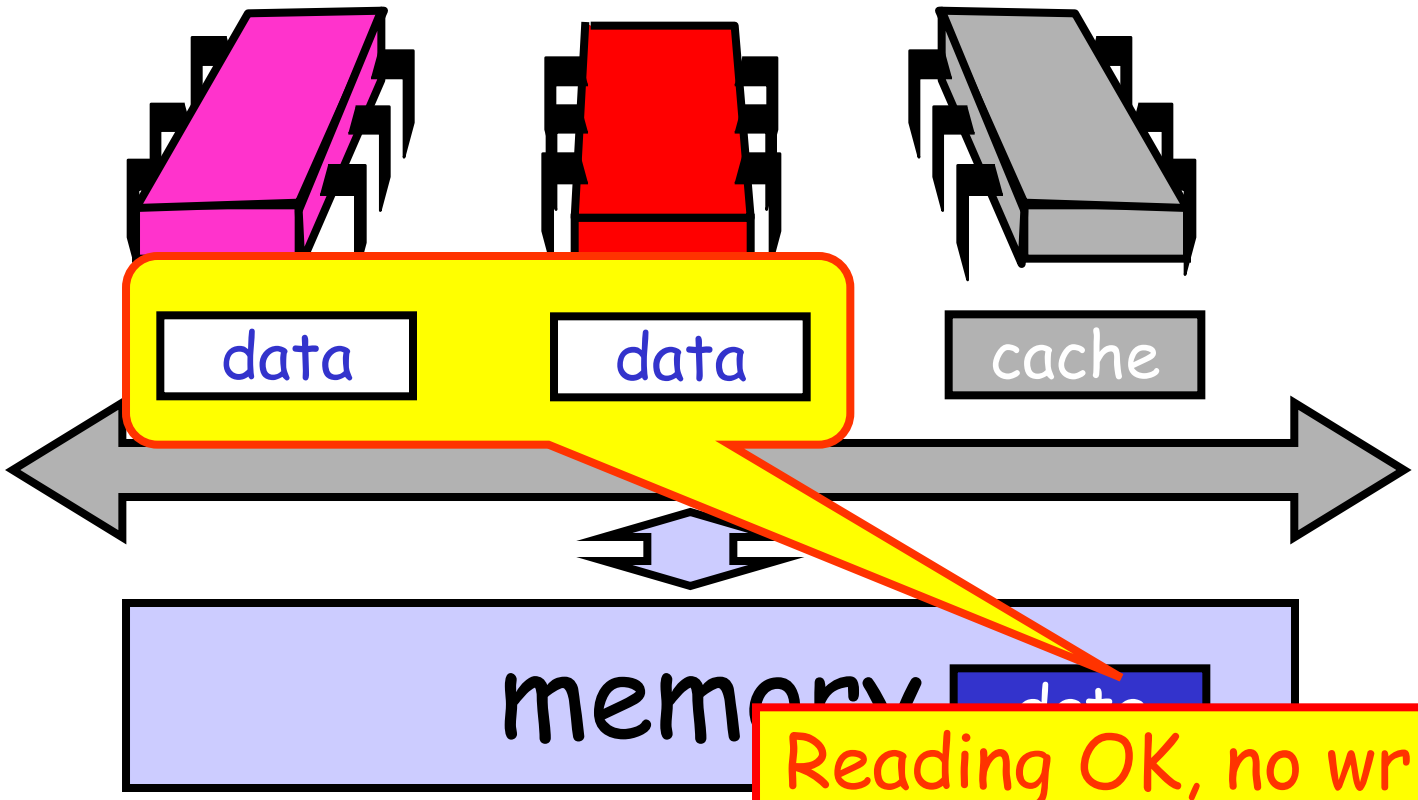
Another Processor Asks for Data



Owner Responds



End of the Day …



Mutual Exclusion

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

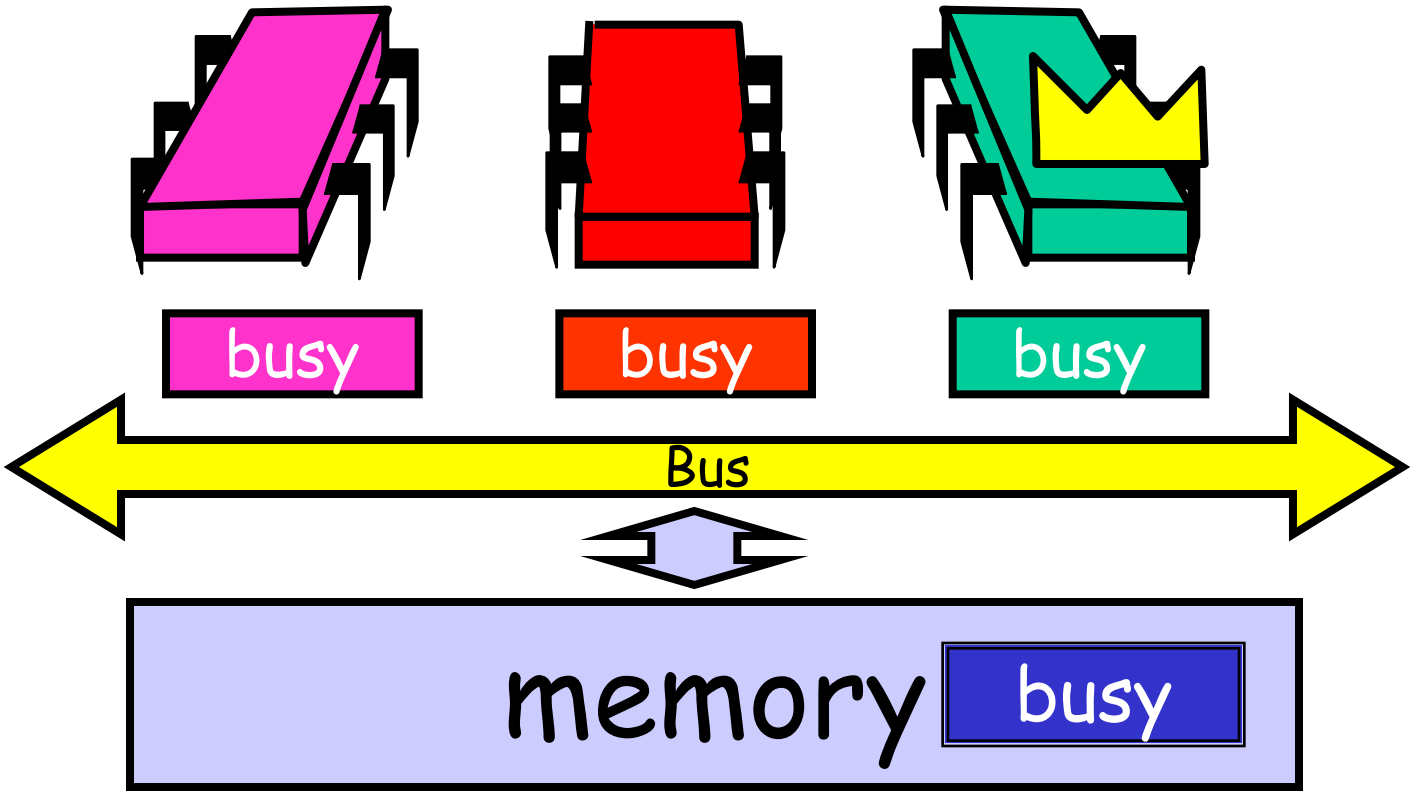
Simple TASLock

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

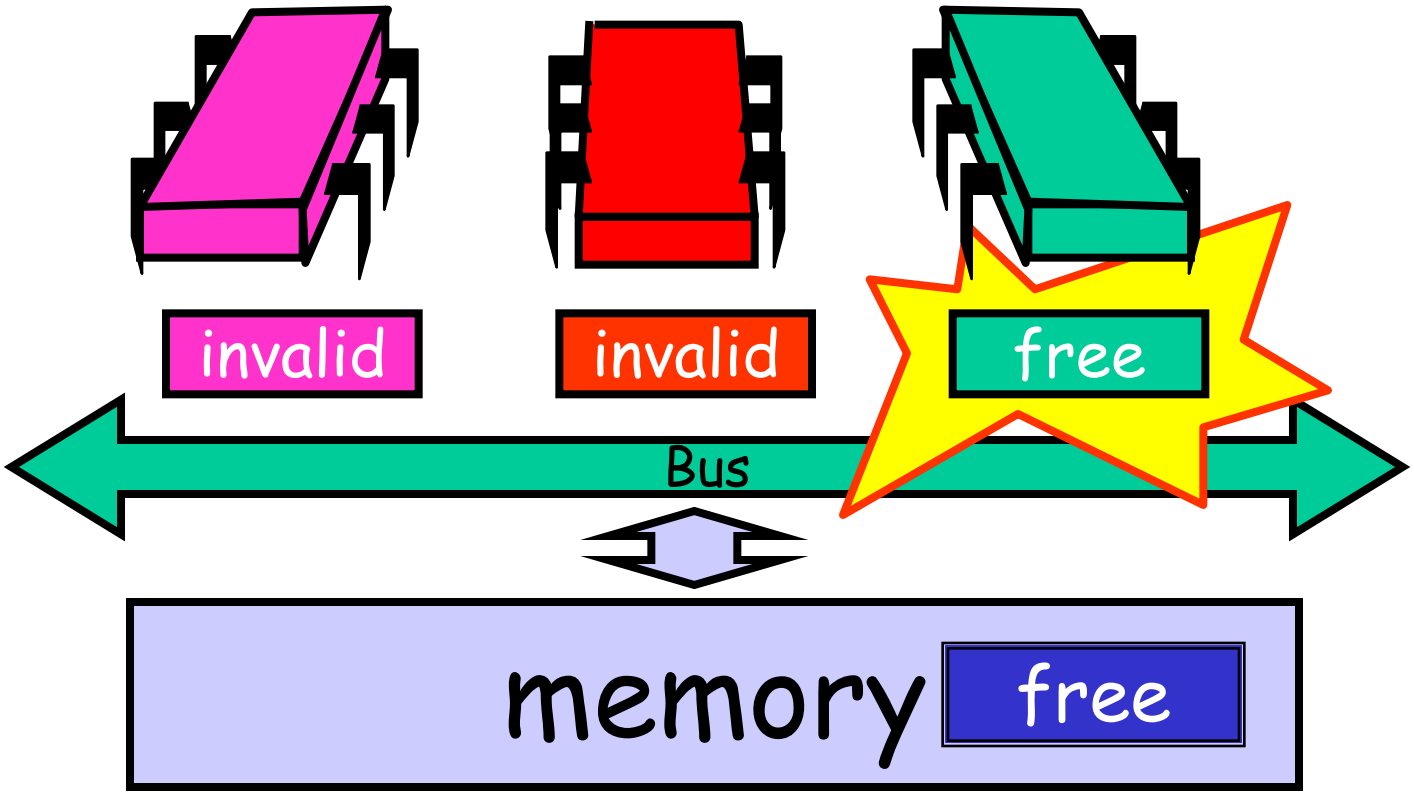
Test-and-test-and-set

- 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

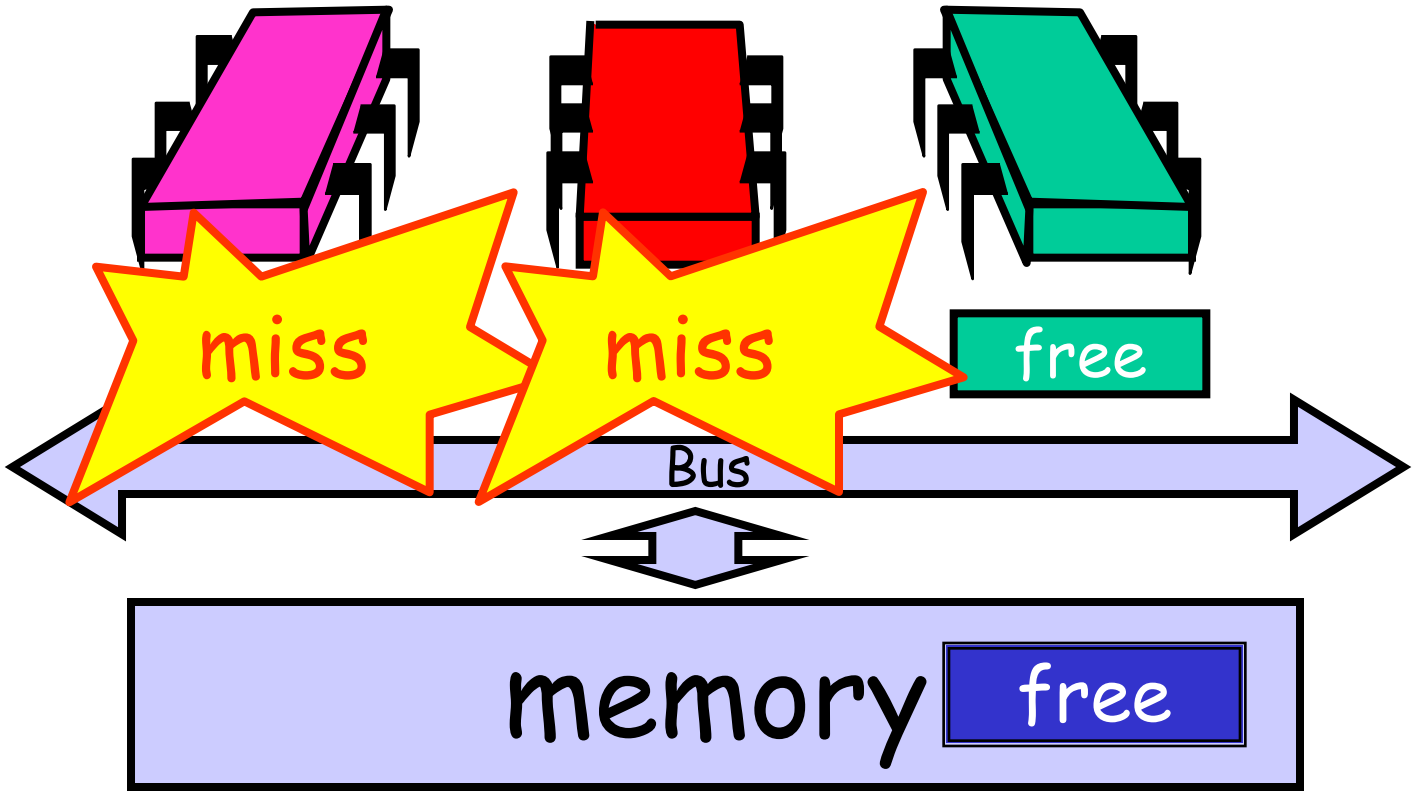


On Release



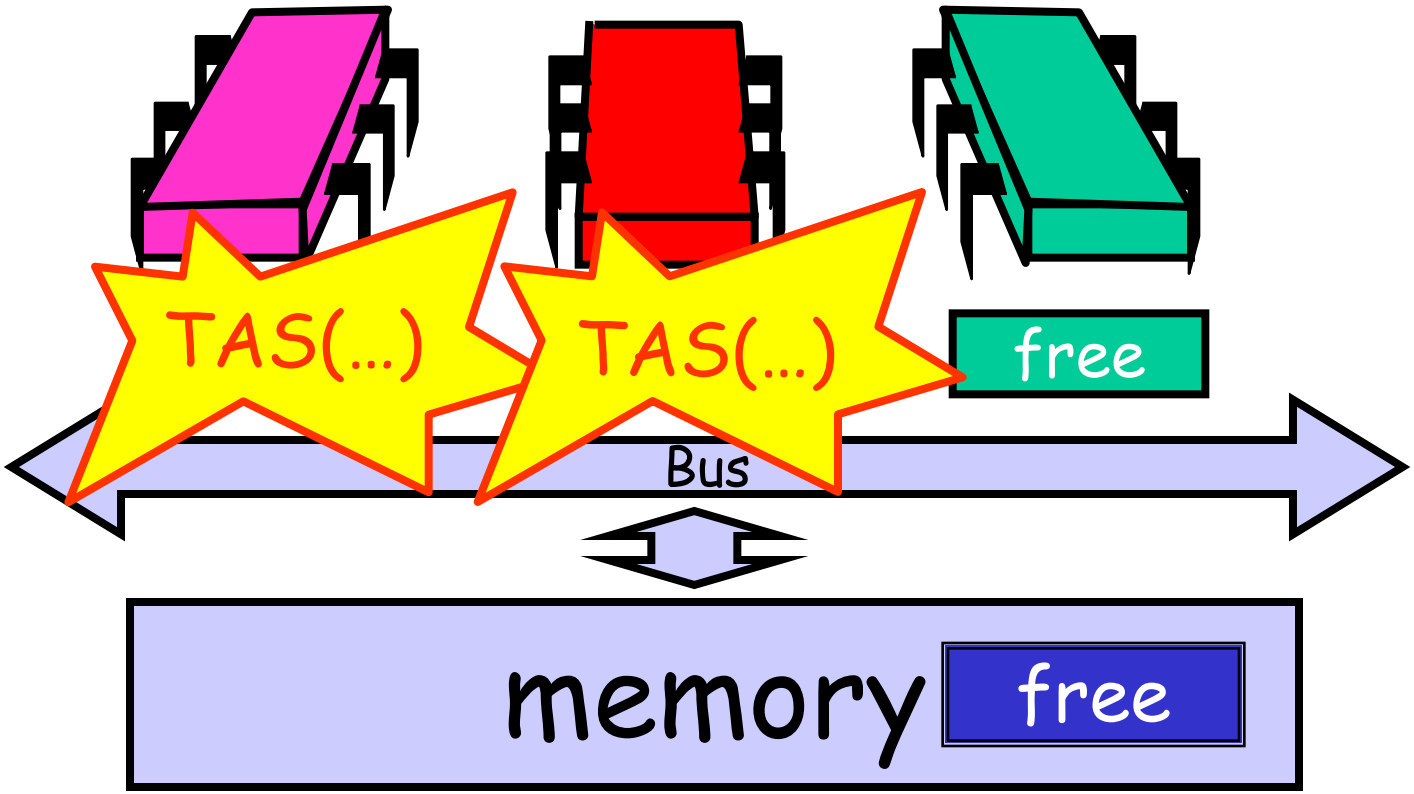
On Release

Everyone misses, rereads



On Release

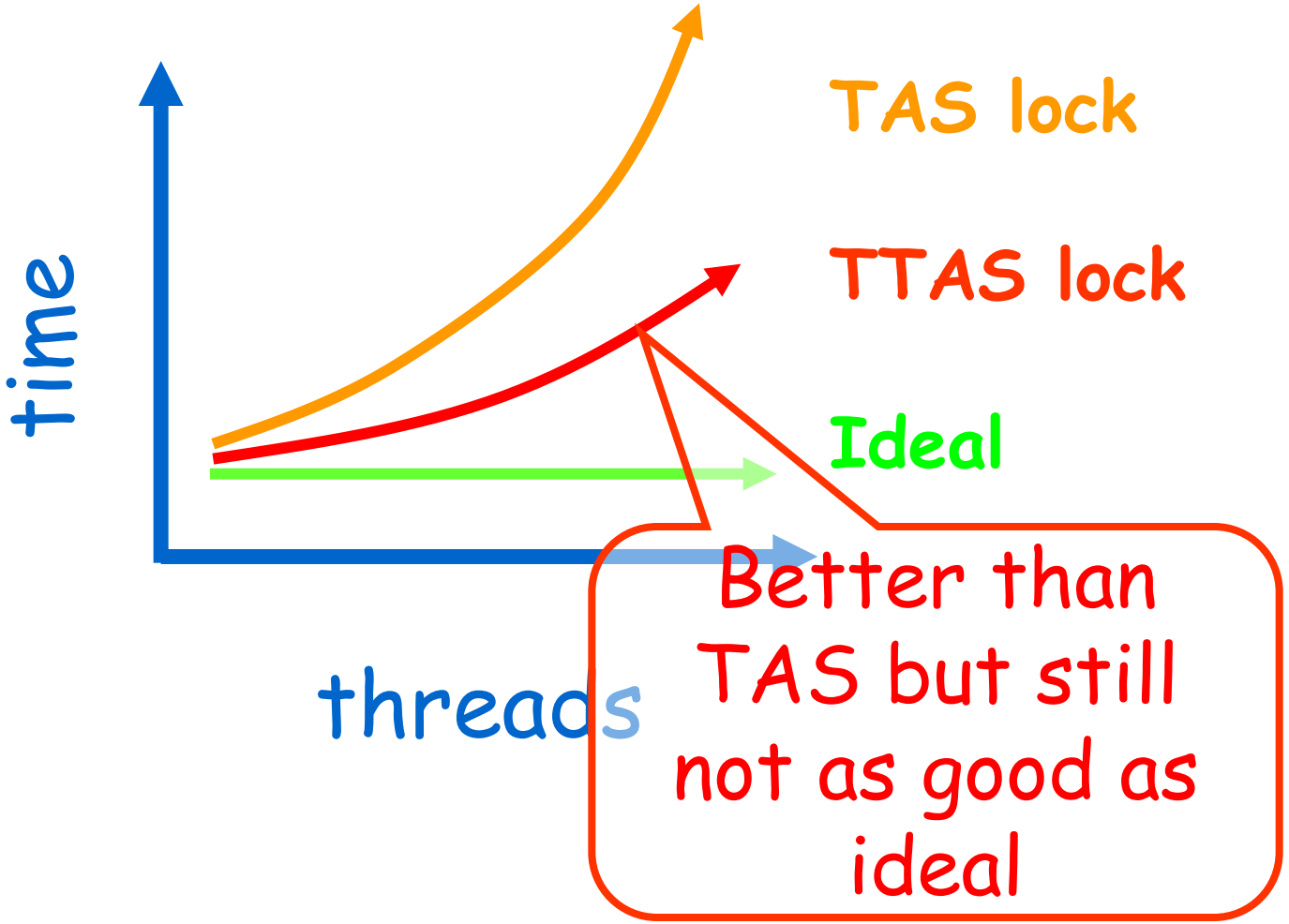
Everyone tries TAS



Problems

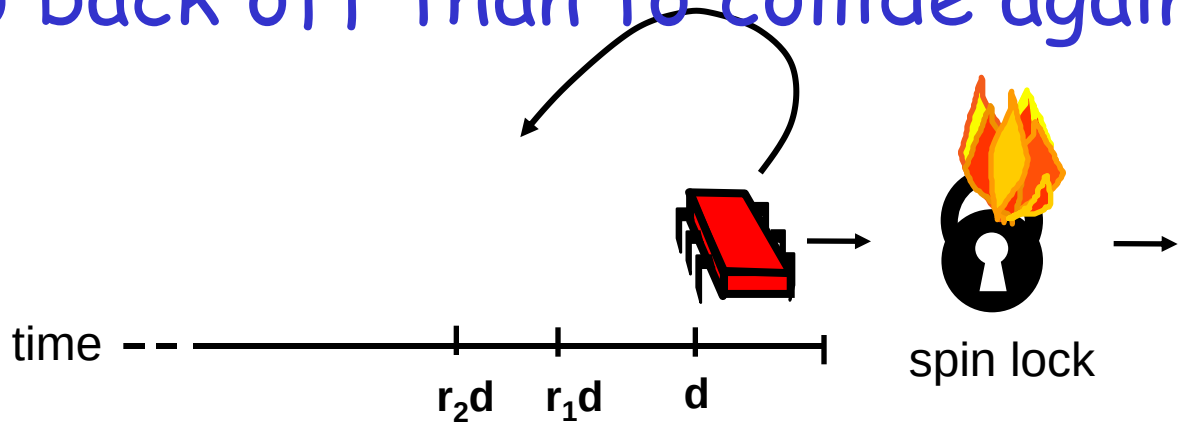
- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?
    Linearly with the number of processors

Mystery Explained

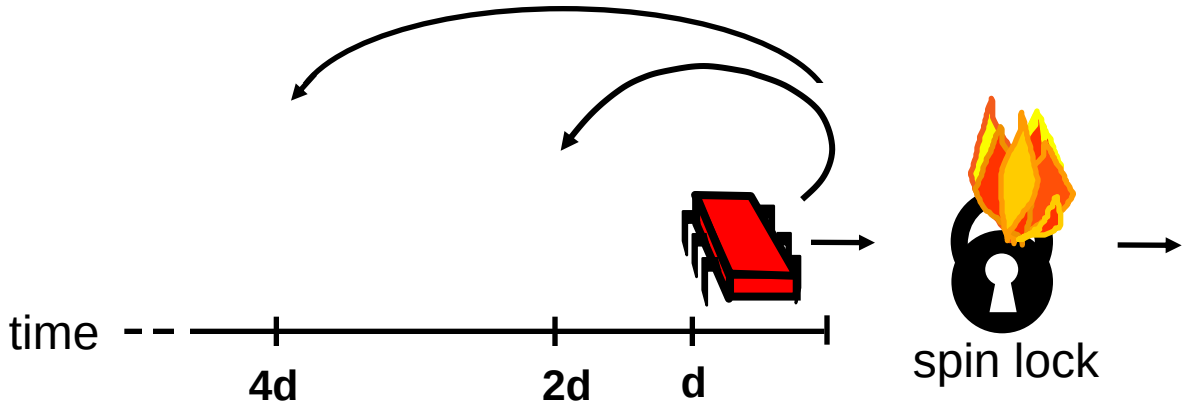


Solution: Introduce Delay

- If the lock looks free
  - But I fail to get it
- There must be lots of 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

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

Exponential Backoff Lock

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

Fix minimum delay

Exponential Backoff Lock

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

Wait until lock looks free

Exponential Backoff Lock

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

If we win, return

Exponential Backoff Lock

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

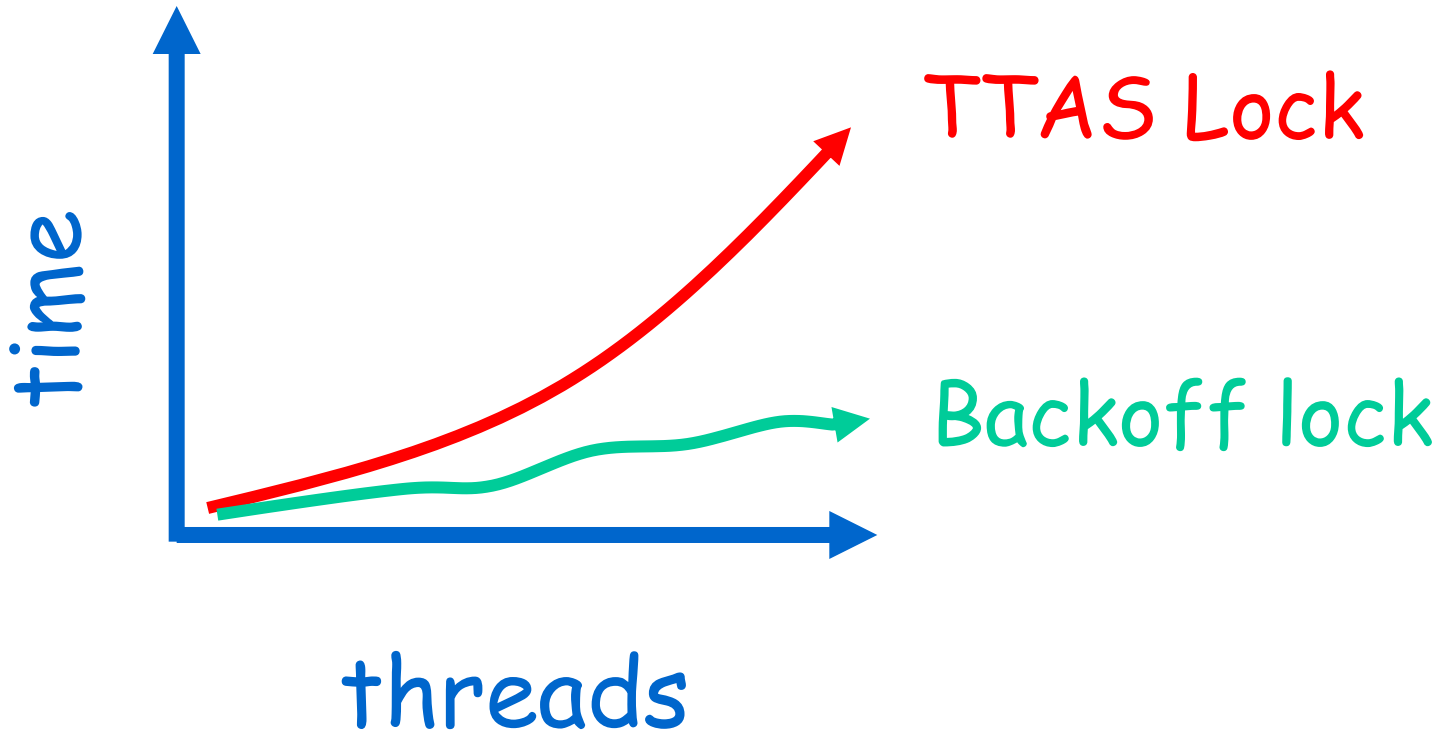
Back off for random duration

Exponential Backoff Lock

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

Double max delay, within reason

Spin-Waiting Overhead



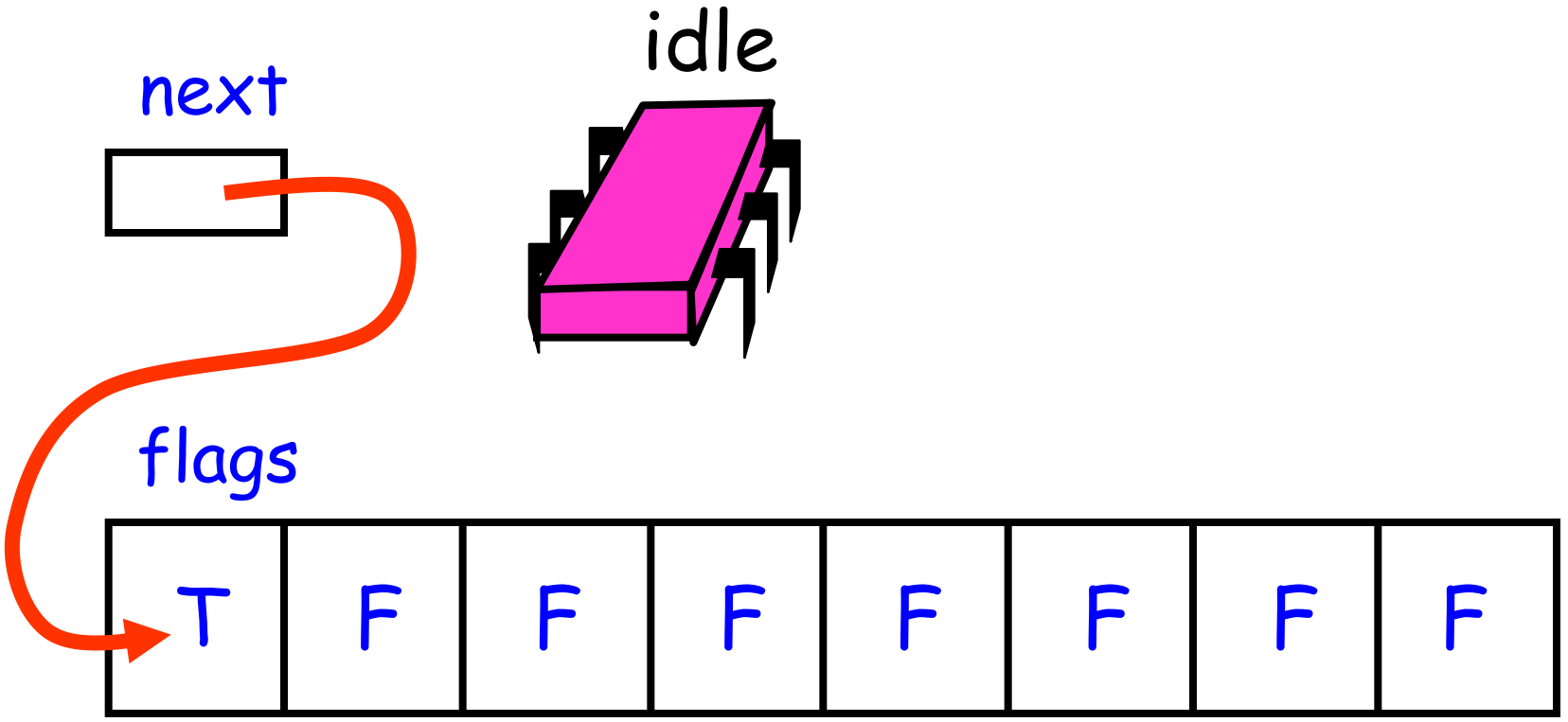
Backoff: Other Issues

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

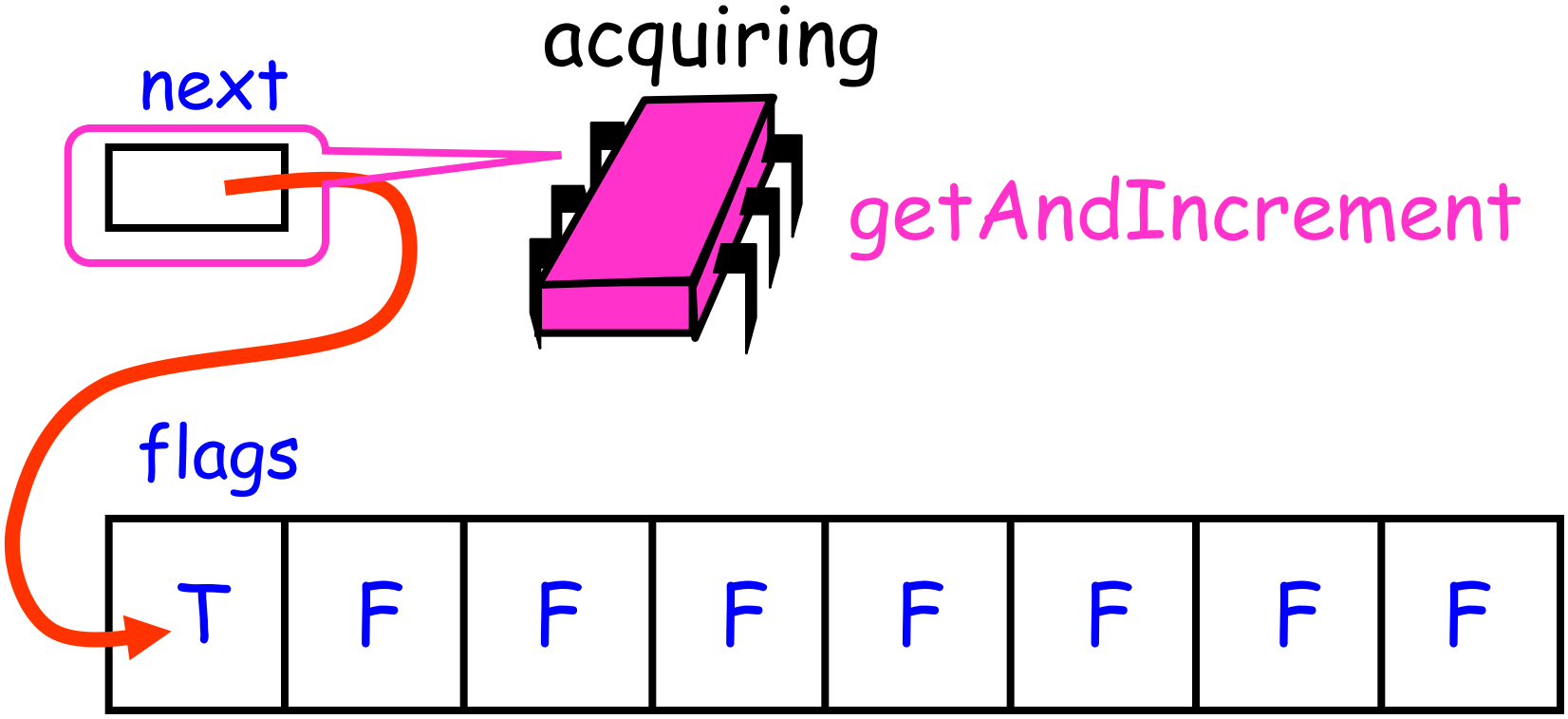
Idea

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

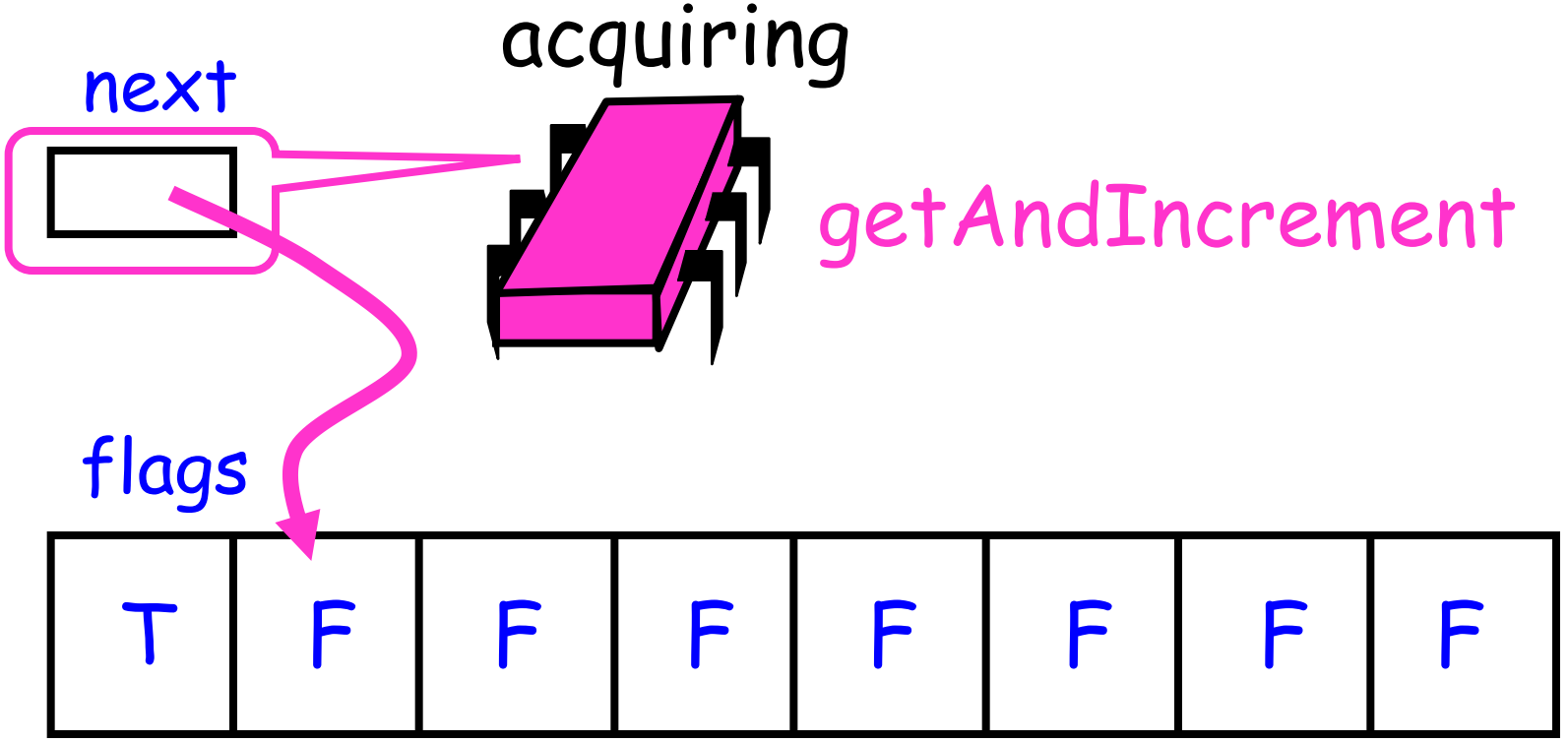
Anderson Queue Lock



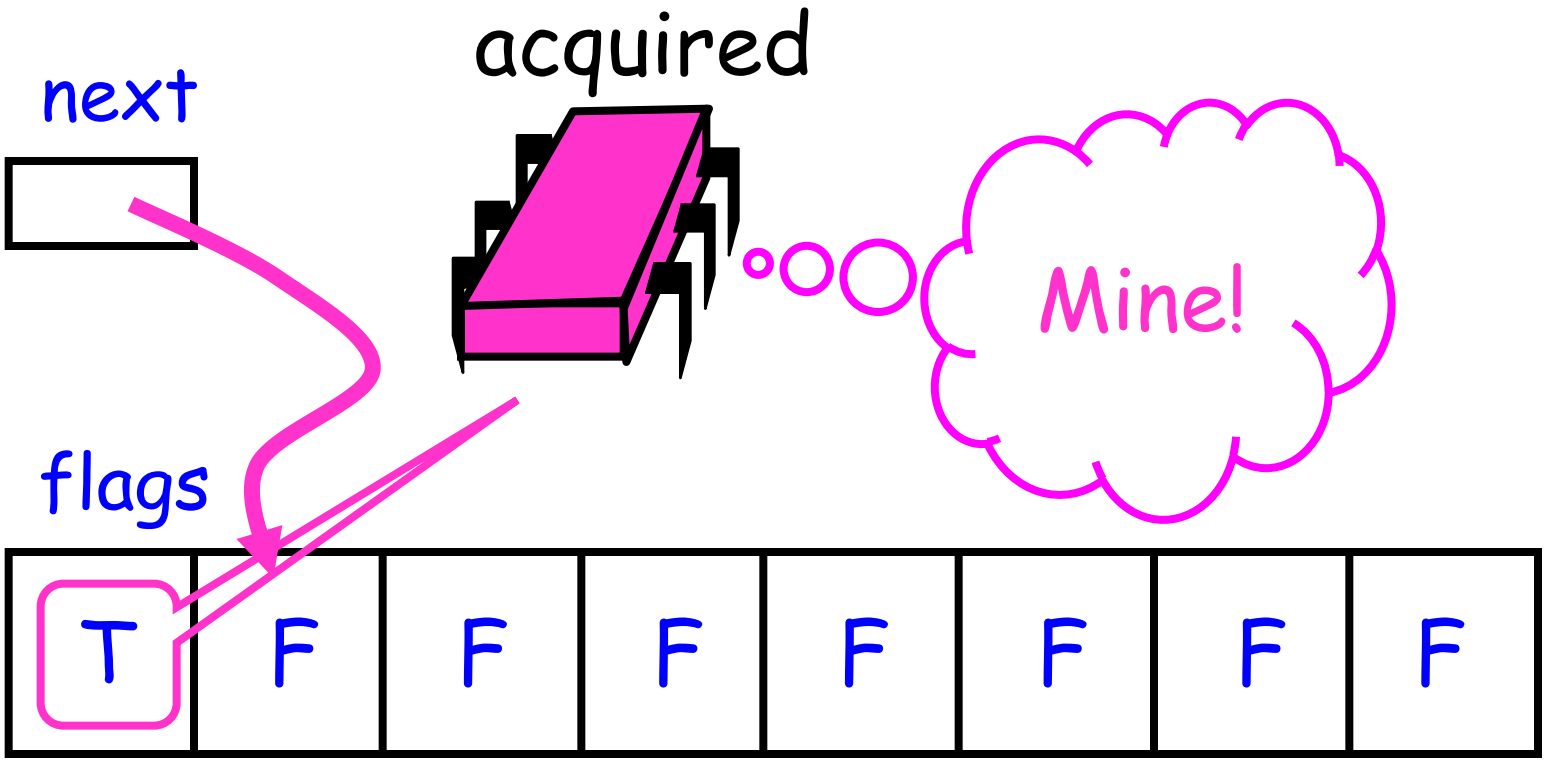
Anderson Queue Lock



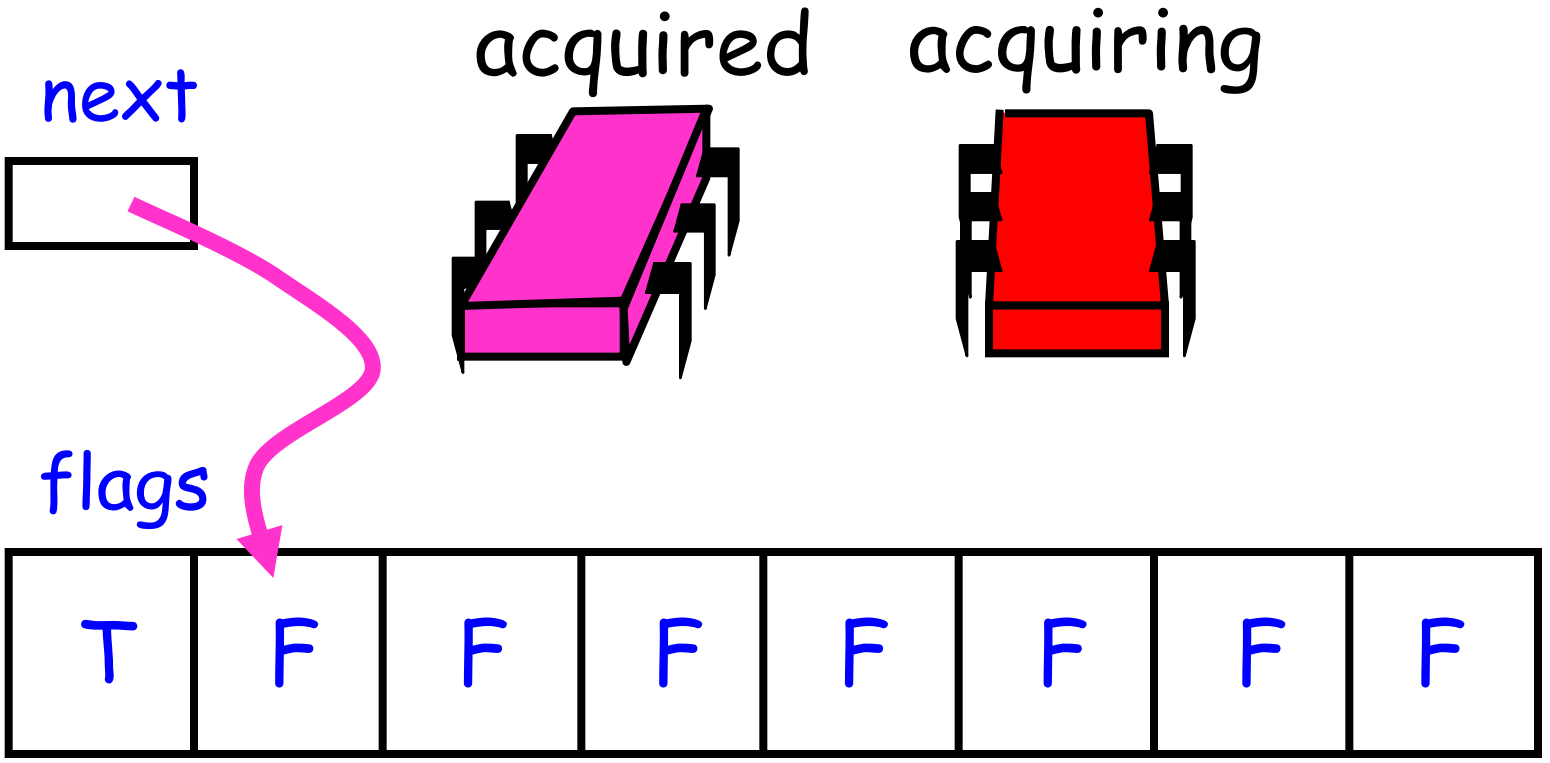
Anderson Queue Lock



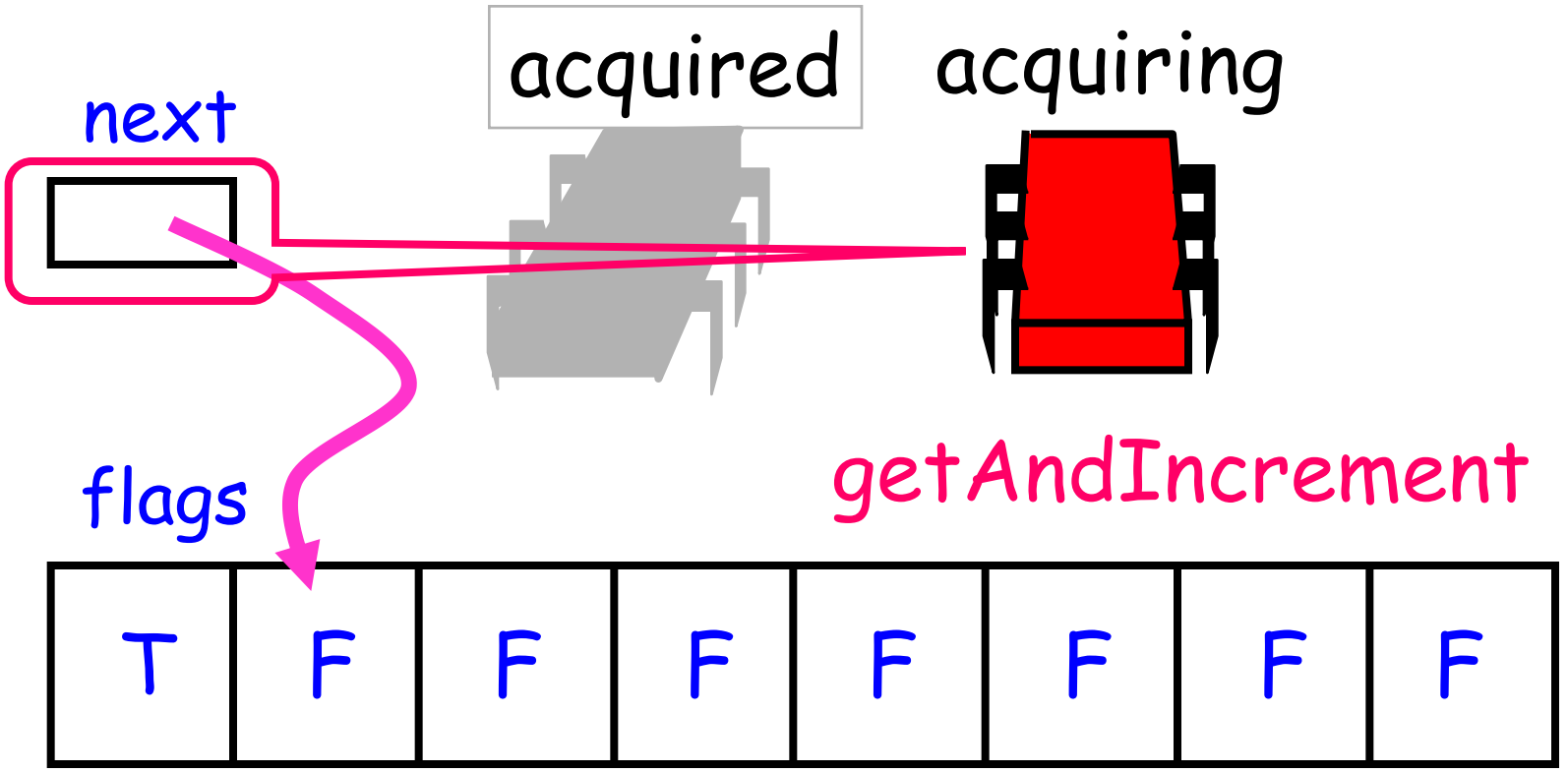
Anderson Queue Lock



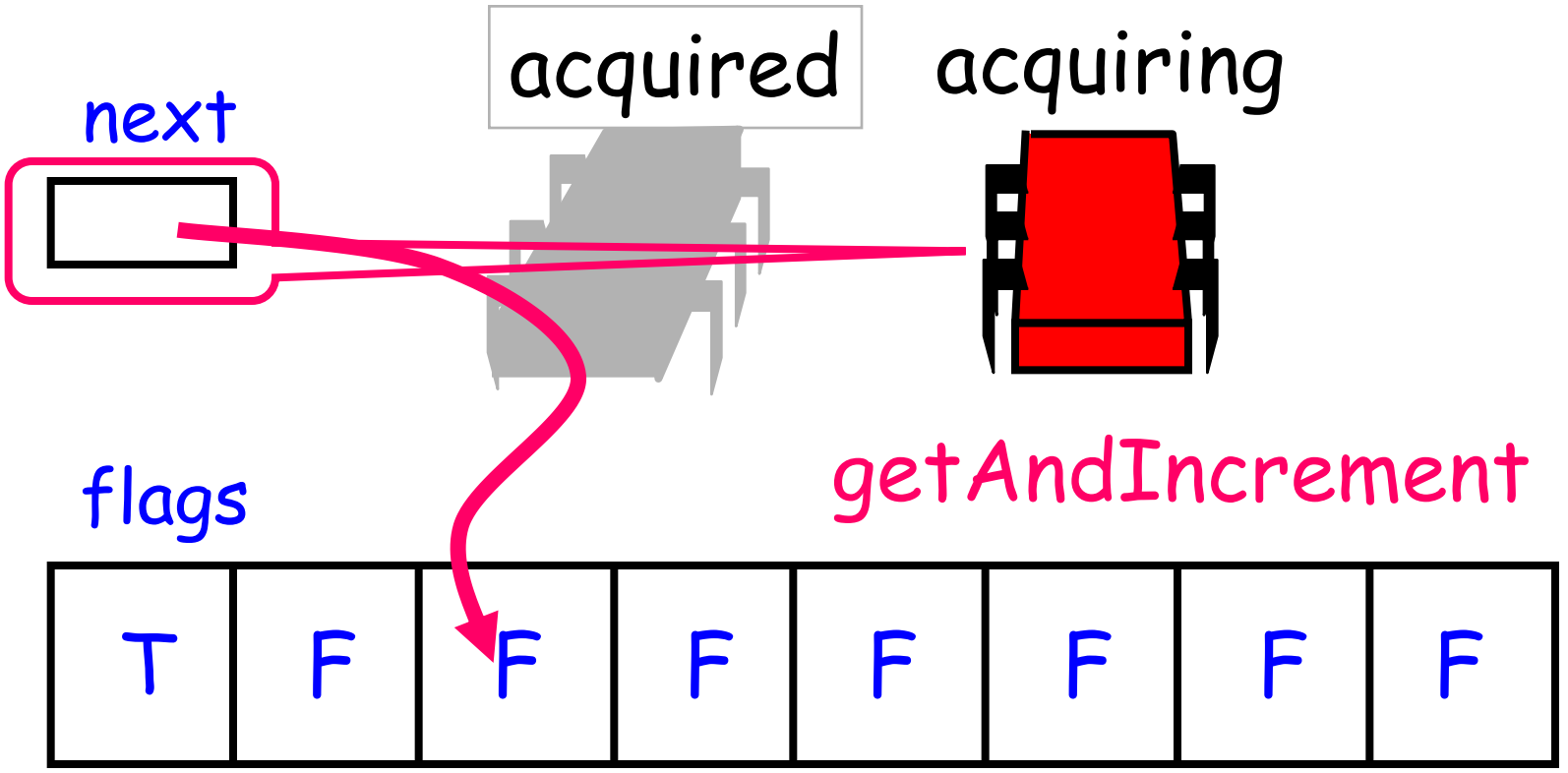
Anderson Queue Lock



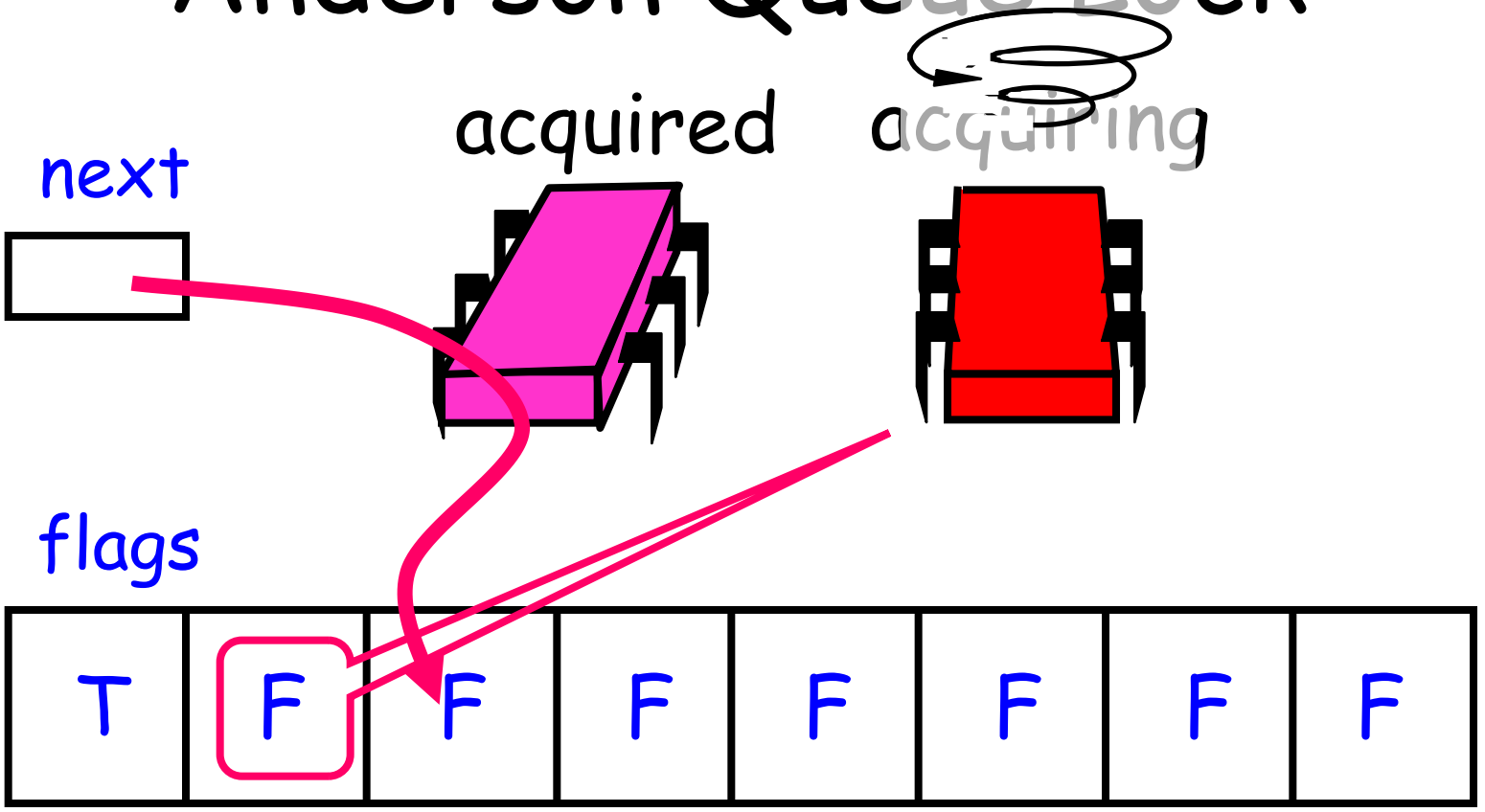
Anderson Queue Lock



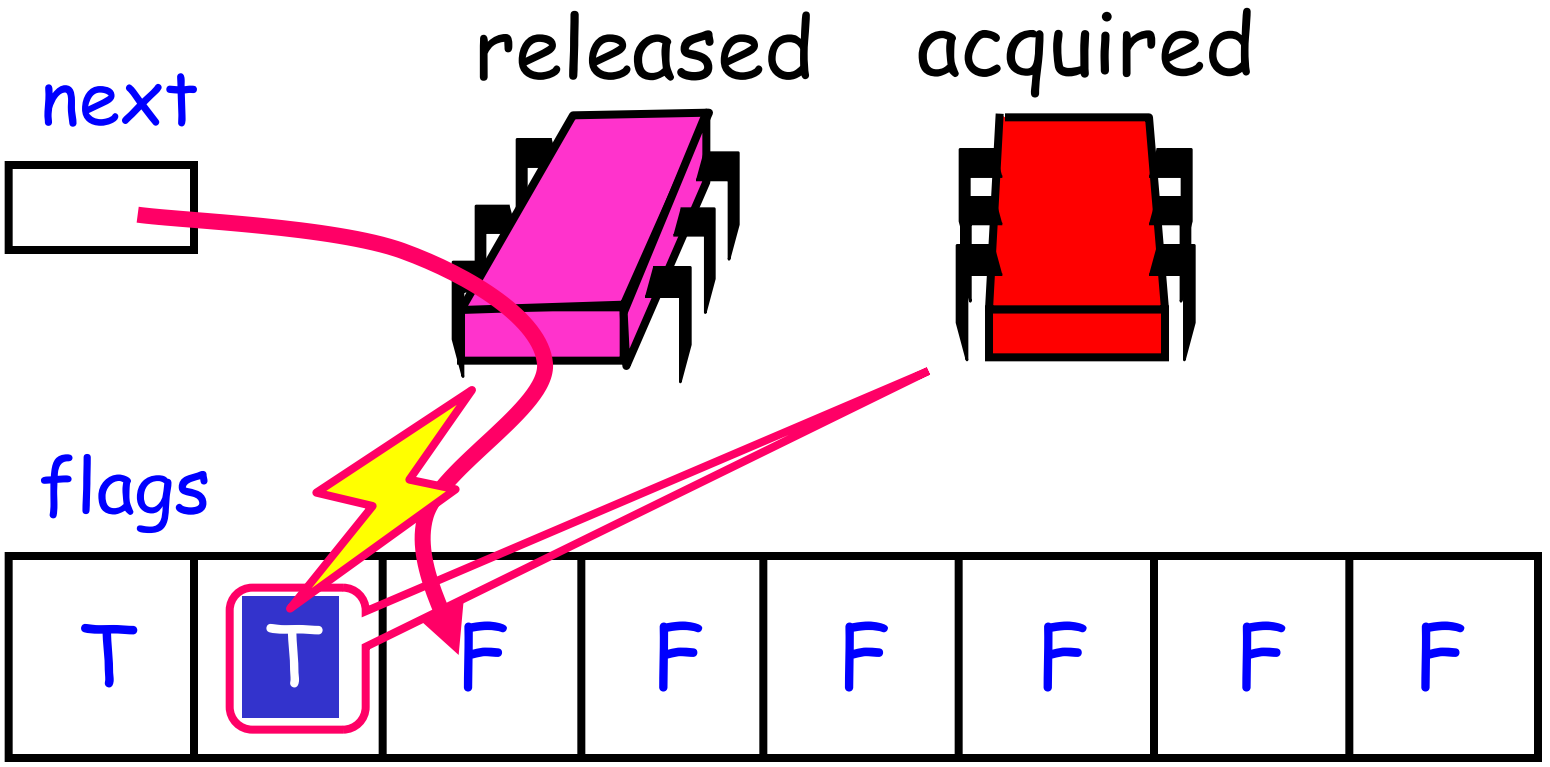
Anderson Queue Lock



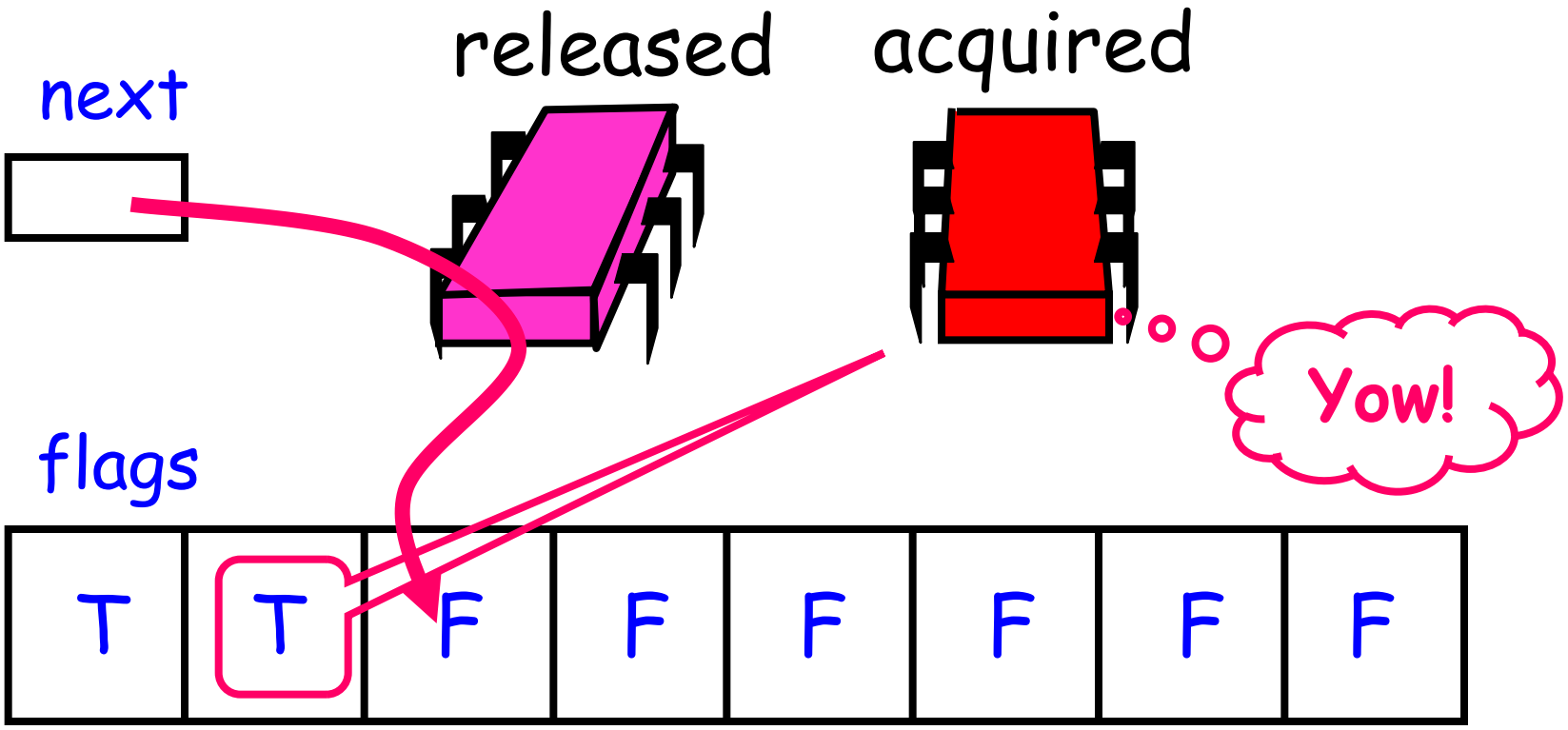
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```

Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```

One flag per thread

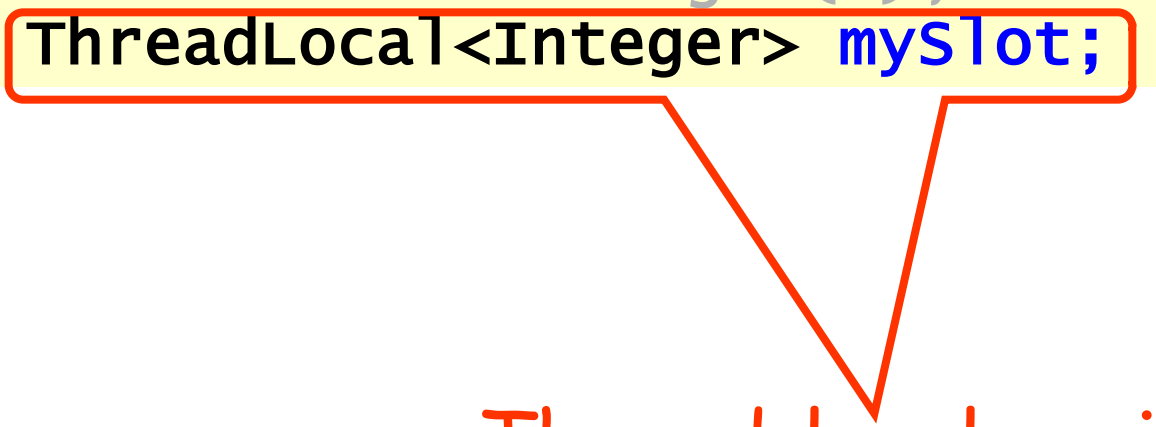
Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next
  = new AtomicInteger(0);
 int[] slot = new int[n];
```

Next flag to use

Anderson Queue Lock

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



Anderson Queue Lock

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

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

Anderson Queue Lock

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

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

Take next slot

Anderson Queue Lock

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

public unlock() {
 flags[(mySlot+1)
}
```

Spin until told to go

Anderson Queue Lock

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

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

Prepare slot for re-use

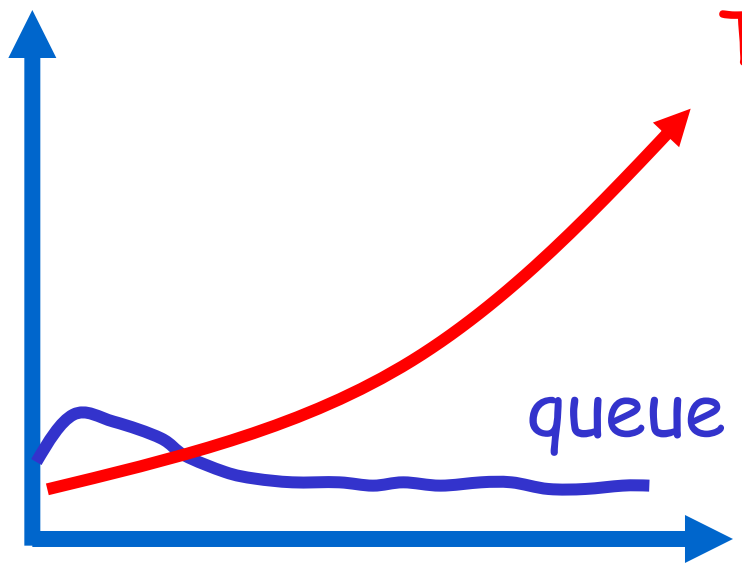
Anderson Queue Lock

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

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

Tell next thread to go

Performance



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

Anderson Queue Lock

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

One Lock To Rule Them All?

- TTAS+Backoff, CLH, MCS, ToLock…
- Each better than others in some way
- There is no one solution
- Lock we pick really depends on:
  - the application
  - the hardware
  - which properties are important