Spin Locks and Contention

CS 475, Spring 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

With material from Herlihy & Shavit, Art of Multiprocessor Programming

Review: Sequential Consistency vs Linearizability

Linearizable

	t=0	t=1
CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 1

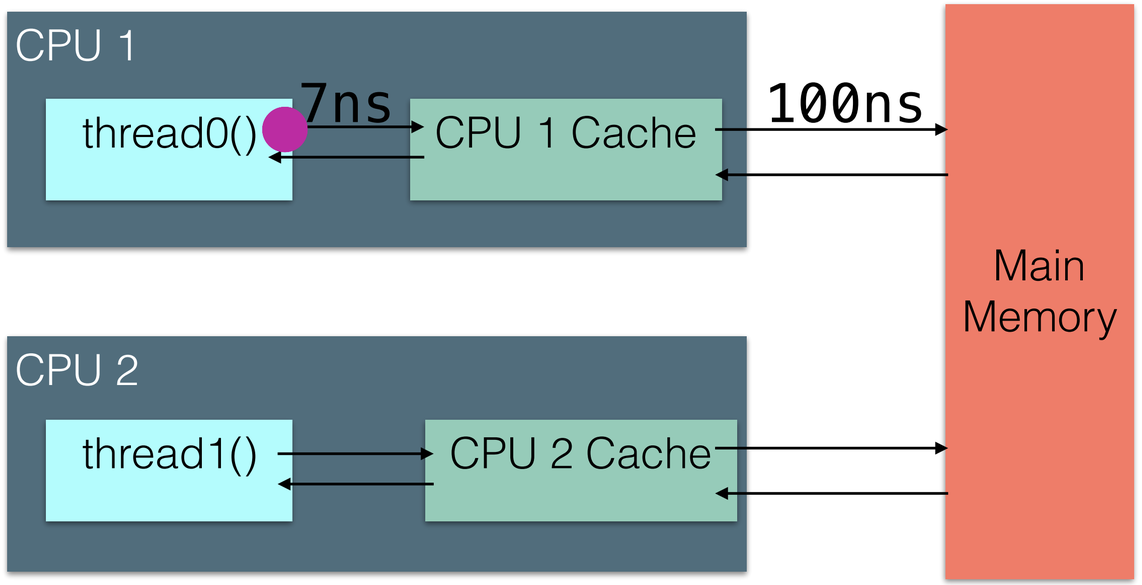
Sequential Consistency

CPU0	W(X) 1	R(Y) 1
CPU1	W(Y) 1	R(X) 0

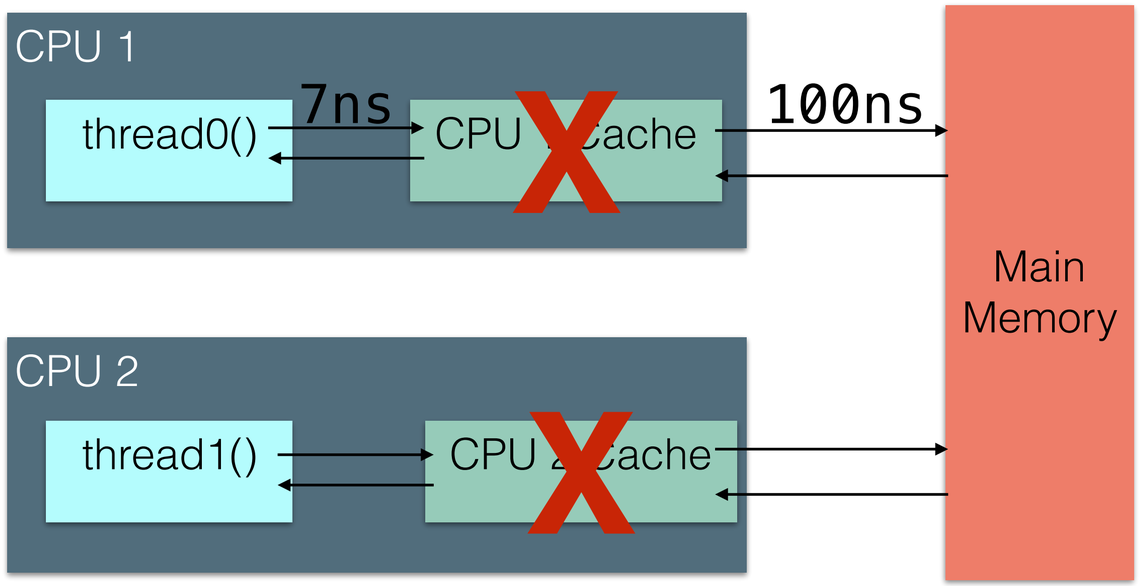
Review: Sequential Consistency vs Linearizability

- Linearizability can be composed:
  - If p's execution and q's execution are both linearizable, then the combination must also be linearizable
- Sequential consistency can not be composed:
  - If p's execution and q's execution are both sequential, then the combination MAY also be sequential (but not guaranteed!)
- Why use sequential consistency?
  - Does not require global clock

Review: Memory Operations are Slow



Review: Volatile Keyword



Conditions

- When a thread is waiting for something to happen, it might want to release the lock and be notified when that thing has happened
  - Ex: while queue is full, release the lock to let someone else empty it
- This is what a *condition* does
- Key methods: *await*, *signal*

```
Condition condition = lock.newCondition();
lock.lock();
try{
   while(!property)
      condition.await();
}catch(InterruptedException e){
   //Application-dependent response, property may still be false
}
//at this point, property must be true and we have the lock again
```

```
condition.signal(); //wake up one thread await'ing
condition.signalAll(); //wake up all threads await'ing
```

Monitors

- Java's **synchronized** keyword creates a *monitor*, which is both a lock and a condition variable
- Three relevant methods that have been there all along, and you may not have known what for:
  - object.notify();
  - object.notifyAll();
  - object.wait();

wait and notify()

- Two mechanisms to enable coordination between multiple threads using the same monitor (target of synchronized)
- While holding a monitor on an object, a thread can **wait** on that monitor, which will temporarily release it, and put that thread to sleep
- Another thread can then acquire the monitor, and can **notify** a waiting thread to resume and re-acquire the monitor

wait and notify() example

Only one thread can be in put or take of the same queue

```
public class BlockingQueue<T> {

    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(T element) throws InterruptedException {
        while(queue.size() == capacity) {
            wait();
        }

        queue.add(element);
        notify(); // notifyAll() for multiple producer/consumer threads
    }

    public synchronized T take() throws InterruptedException {
        while(queue.isEmpty()) {
            wait();
        }

        T item = queue.remove();
        notify(); // notifyAll() for multiple producer/consumer threads
        return item;
    }
}
```

Non-mutual exclusion locks

- The strict mutual exclusion property of locks is often relaxed. Three examples are:
  - Readers-writers lock: Allows concurrent readers,
    while a writer disallows concurrent readers and writers.
  - Reentrant lock: Allows a thread to acquire the same lock multiple times, to avoid deadlock (we have mostly used reentrant locks)
  - Semaphore: Allows at most c concurrent threads in their critical section, for some given capacity c.

Readers and Writers Problem

- Commonly called a read-write lock: data can be read by an unlimited number of threads at a time, written by at most one
- If a thread wants to write, nobody else can be reading
- High level approach:
  - Build on top of mutual exclusion locks with condition variables

Readers and Writers Lock

```
class ReadLock implements Lock{
   public void lock()
   {
      lock.lock();
      try{
         while(writer){
            condition.await();
         }
         readers++;
      } finally {
         lock.unlock();
      }
   }

   @Override
   public void unlock() {
      lock.lock();
      try{
         readers--;
         if(readers == 0)
            condition.signalAll();
      } finally{
         lock.unlock();
      }
   }
}
```

```
class WriteLock implements Lock{
   public void lock(){
      lock.lock();
      try{
         while(readers > 0 || writer)
            condition.await();
         writer =true;

      } finally{
         lock.unlock();
      }
   }

   @Override
   public void unlock() {
      writer = false;
      condition.signalAll();
   }
}
```

Readers and Writers Lock

- Note: not fair - many readers can prevent a single writer from getting in
- A fair solution is outlined in the textbook

Today

- Wrap up on conditions and read-write locks
- Shift focus to *performance* of mutual exclusion locks, focusing on spin locks

- Reading: H&S 7.1-7.4
- Note: Midterm is now before spring break (hooray?)

HW1 Discussion

Go to socrative.com and select “Student Login” Room: CS475; ID is your G-Number

1. How fair do you think this assignment was?
2. How difficult did you think this assignment was?
3. How long did you spend on this assignment?
4. How long did you spend ONLY parts 1-3 of this assignment?
5. How long did you spend on ONLY part 4 of this assignment?
6. Do you think that you spent more time on the assignment because we allowed you to see your grade and resubmit it?

Reminder: If you are not in class, you may not complete the activity. If you do anyway, this will constitute a violation of the honor code.

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)

Today: Revisit Mutual Exclusion

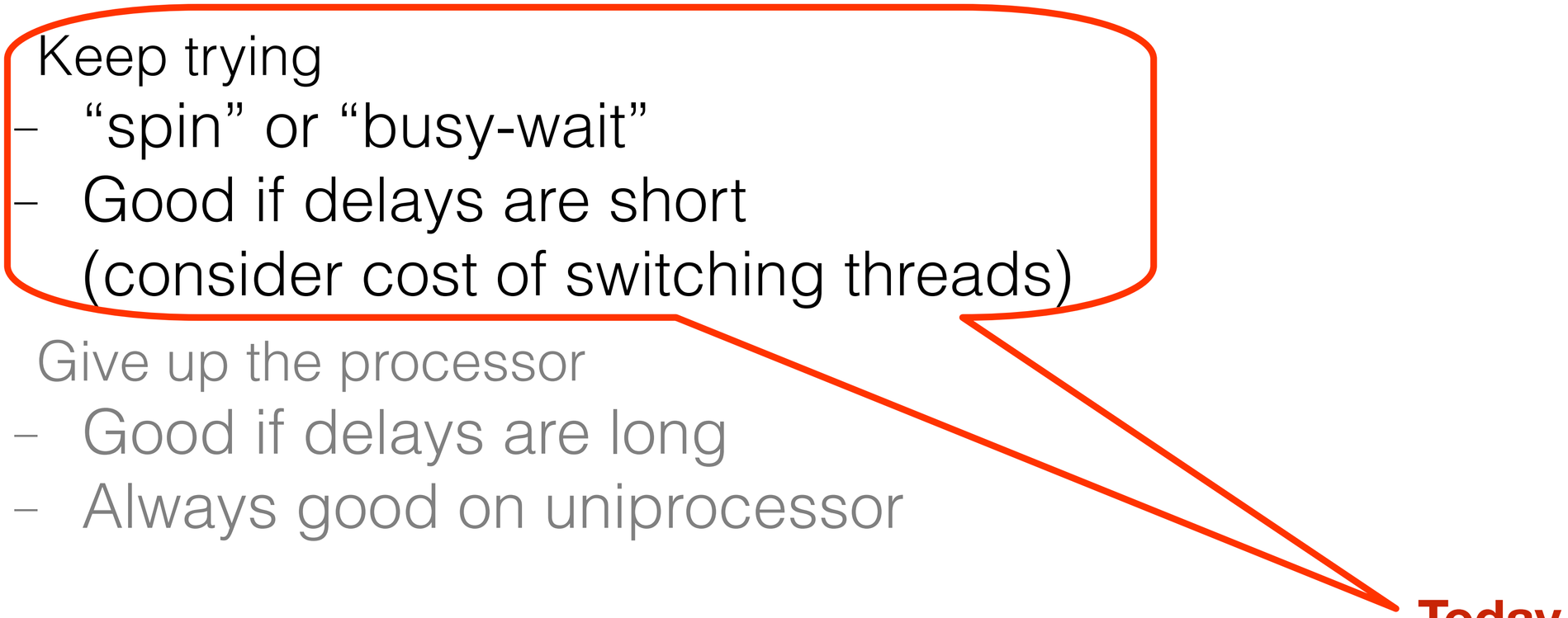
- Think of performance, not just correctness and progress
- Begin to understand how performance depends on our software properly utilizing the multiprocessor machine's hardware
- And get to know a collection of locking algorithms…

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

- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short (consider cost of switching threads)
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

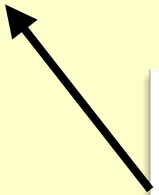
What Should you do if you can't get a lock?

- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short (consider cost of switching threads)
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor



Peterson's Algorithm - Reminder (Lecture 3)

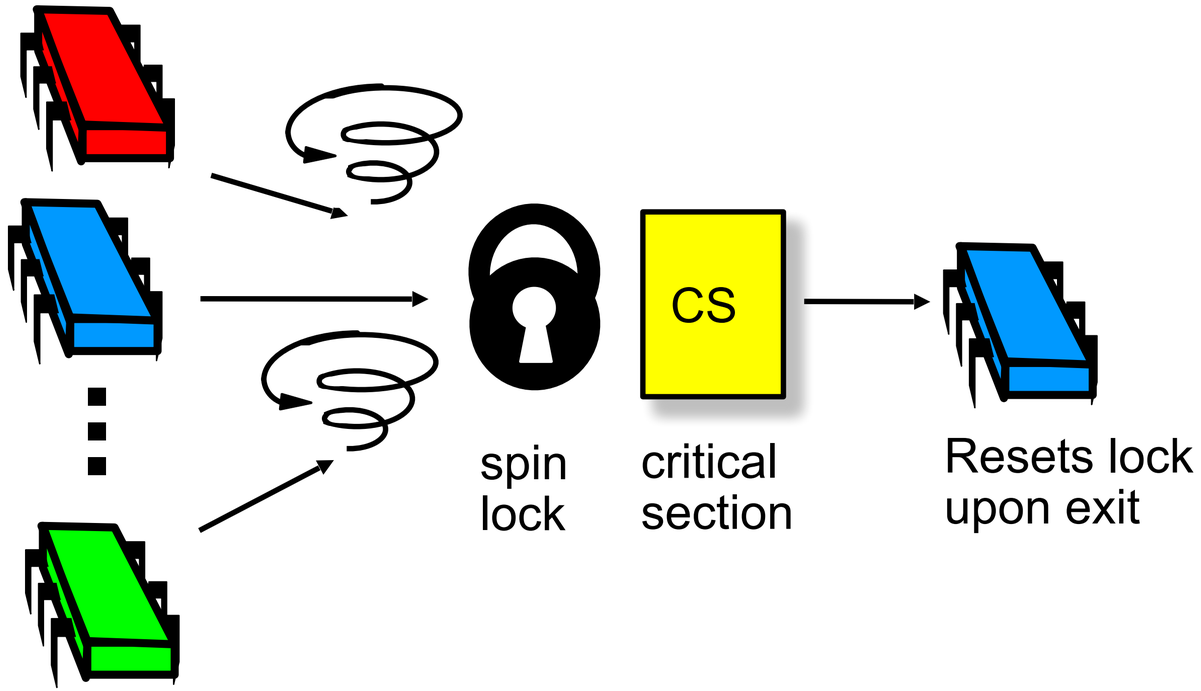
```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
 }
public void unlock() {
 flag[i] = false;
}
```



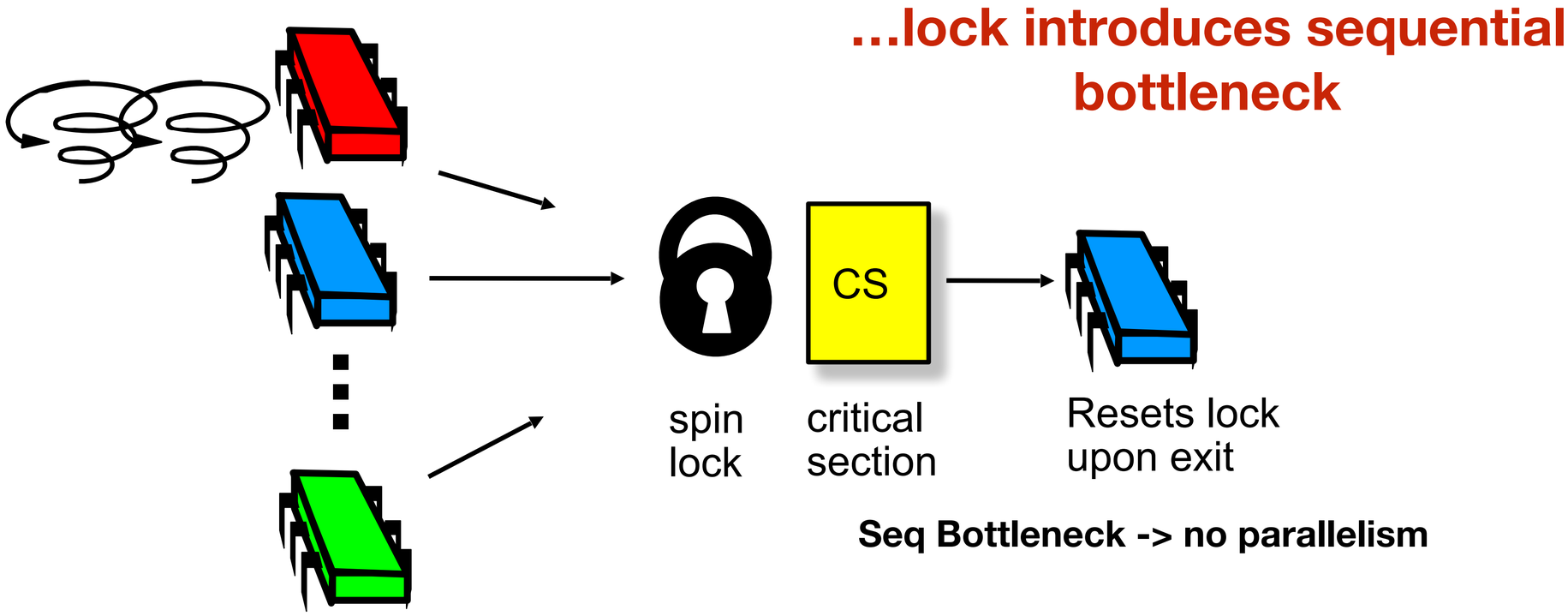
Threads "spin" in this empty loop while waiting

"Busy" waiting - thread keeps running on the CPU

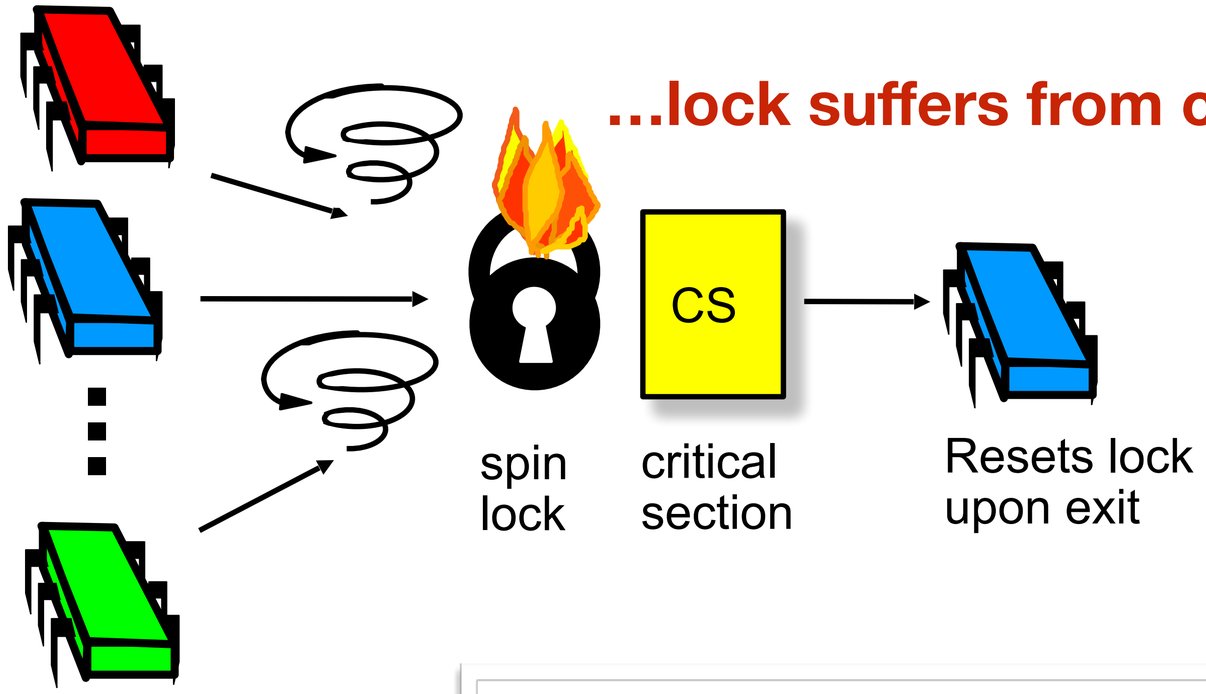
Basic Spin-Lock



Basic Spin-Lock

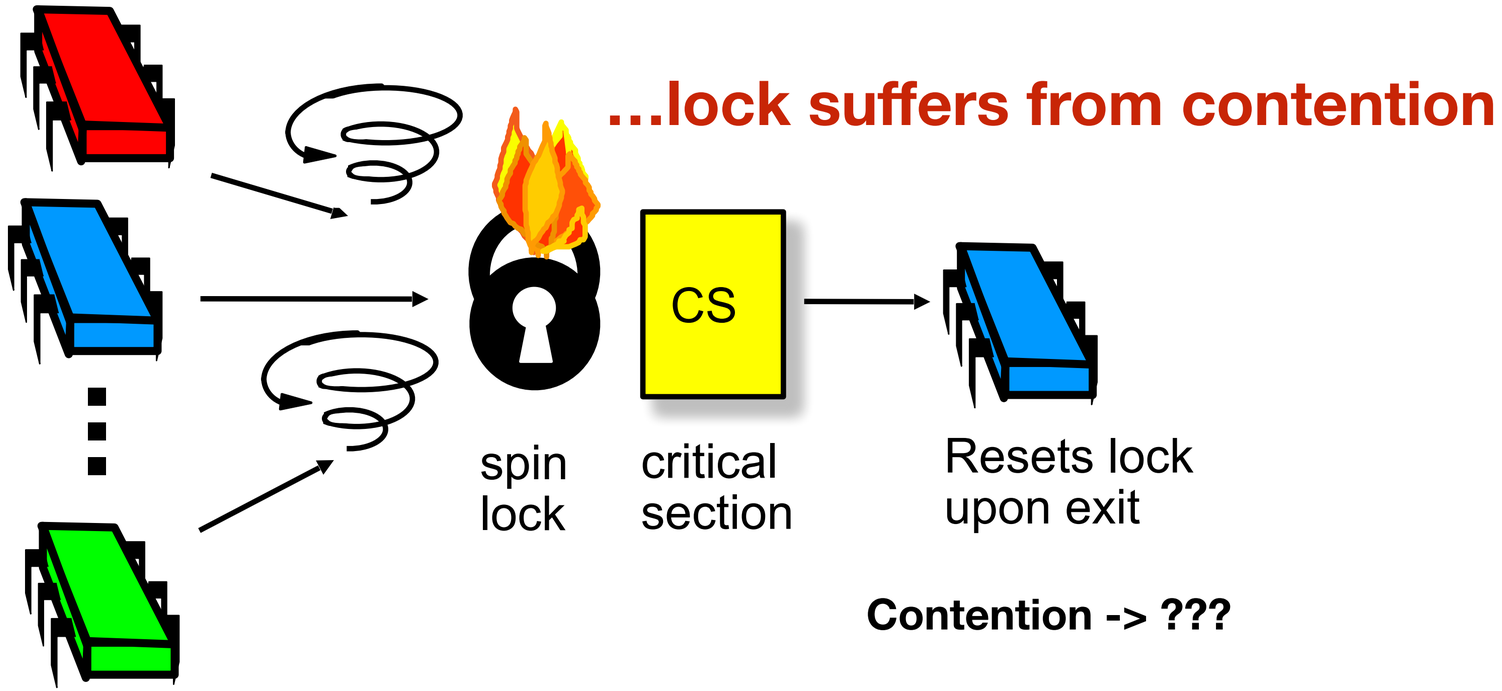


Basic Spin-Lock



Note: Contention and bottlenecking are separate phenomena

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.set(fa
 }}
```

Lock state is AtomicBoolean

Test-and-set Lock

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

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

 void unlock() {
  state.set(f
 }}
```

Keep trying until lock acquired

Test-and-set Lock

```
class TASlock {
 AtomicBoolean state =
  new At

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

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

Release lock by resetting state to false

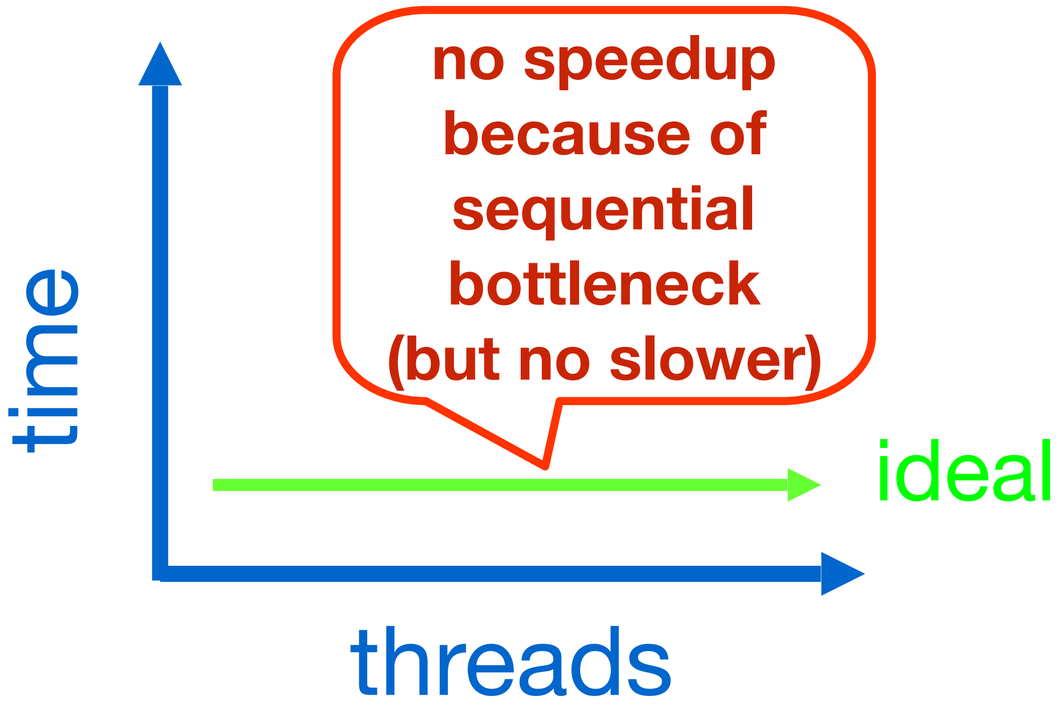
Space Complexity

- TAS spin-lock has small “footprint”
- N thread spin-lock uses O(1) space
- As opposed to O(n) Peterson/Bakery
- How did we overcome the Ω(n) lower bound?
- We used an atomic Read/Modify/Write operation
  - Under the covers uses some special hardware features, *not* just the synchronized keyword as in the slides (if so, still O(n)?)

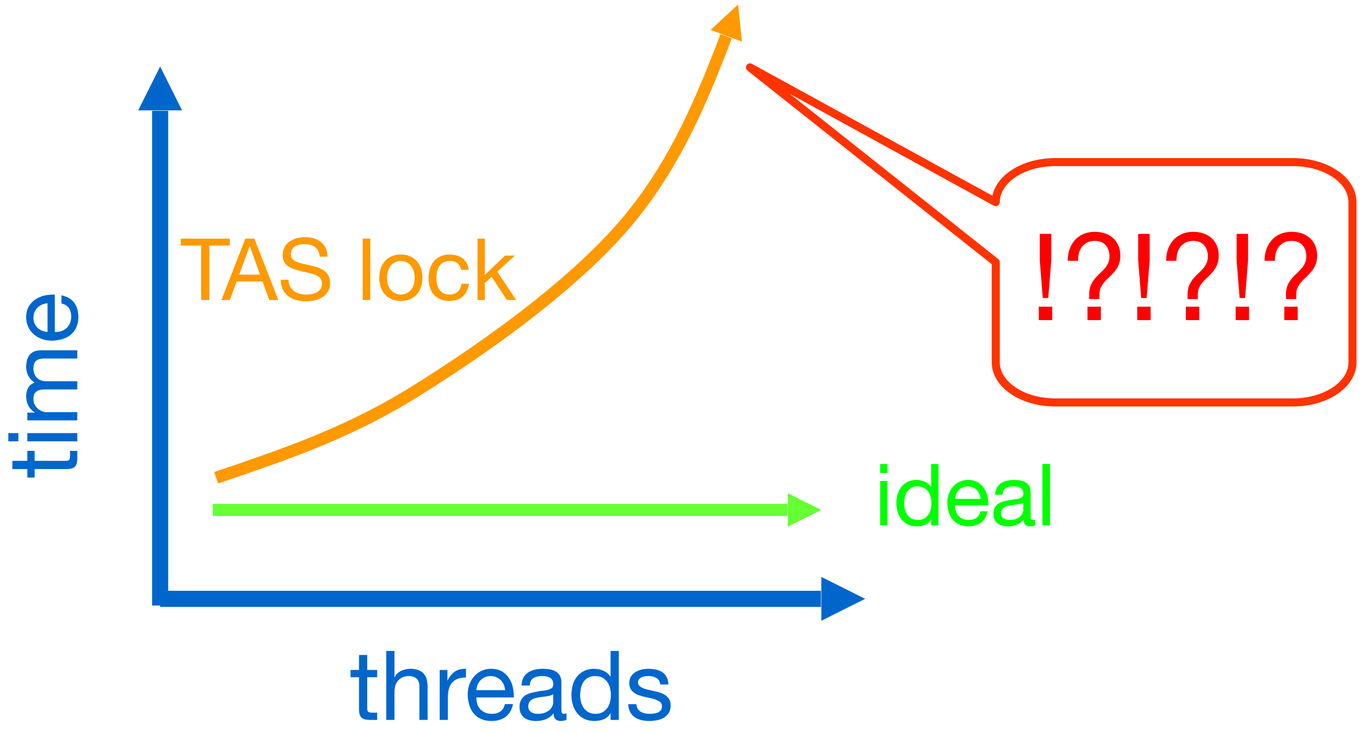
Performance

- 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
(but no slower)
time
ideal
threads

Mystery #1



Adding MORE threads makes it SLOWER!

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

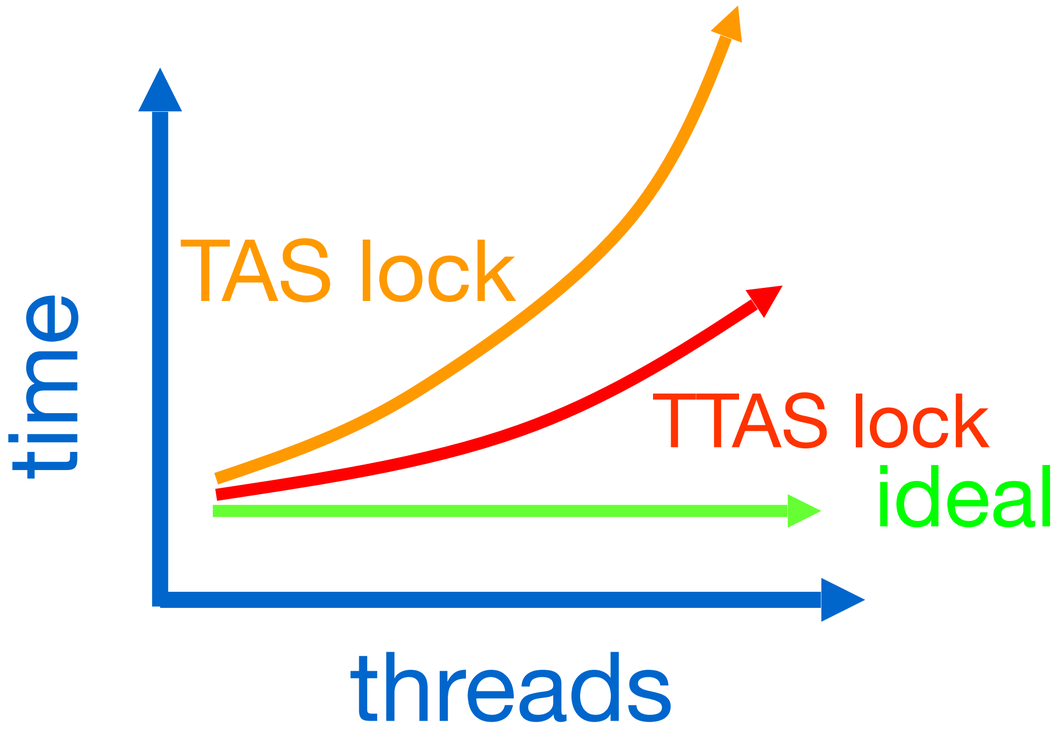
Test-and-test-and-set Lock

```
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 …

CPU Architectures

Kinds of Architectures

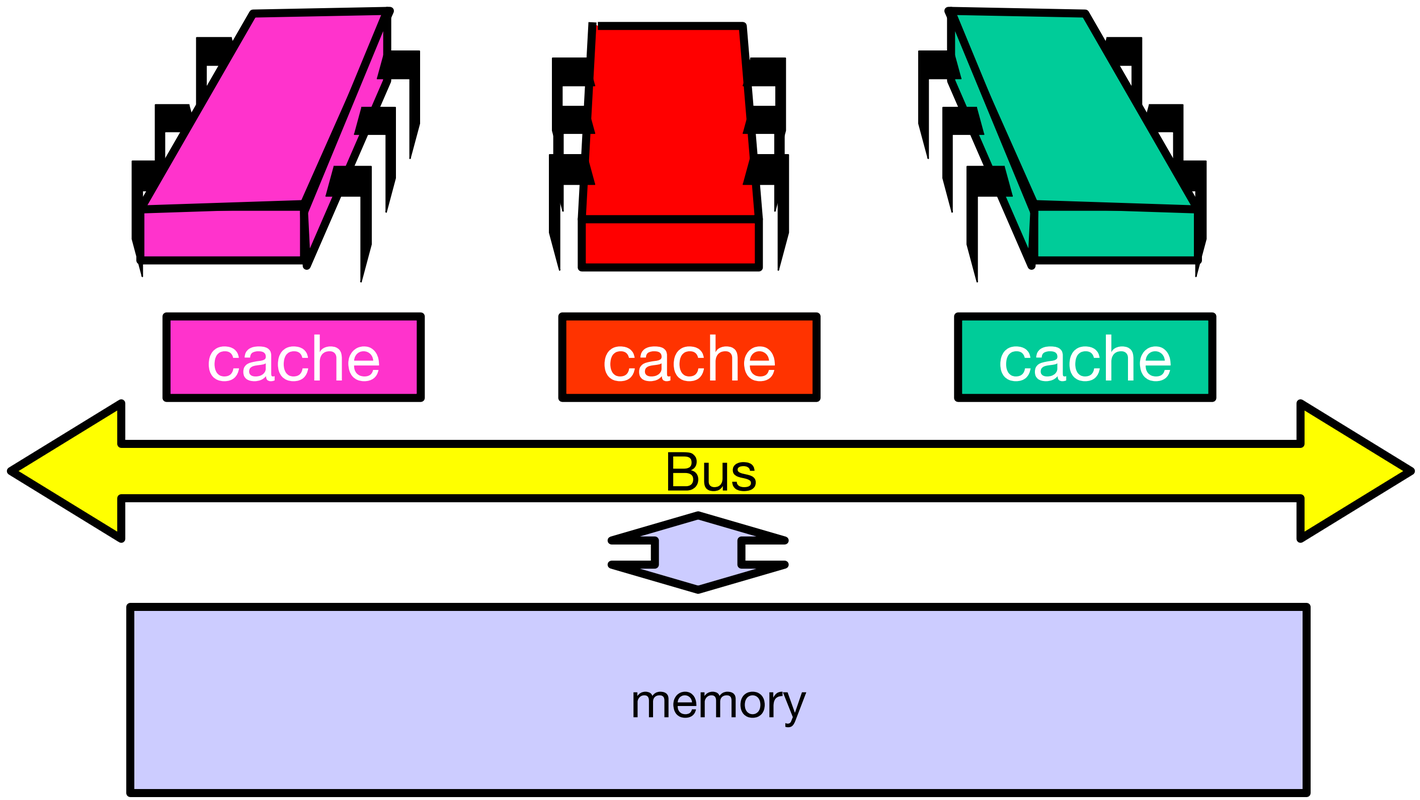
- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

Kinds of Architectures

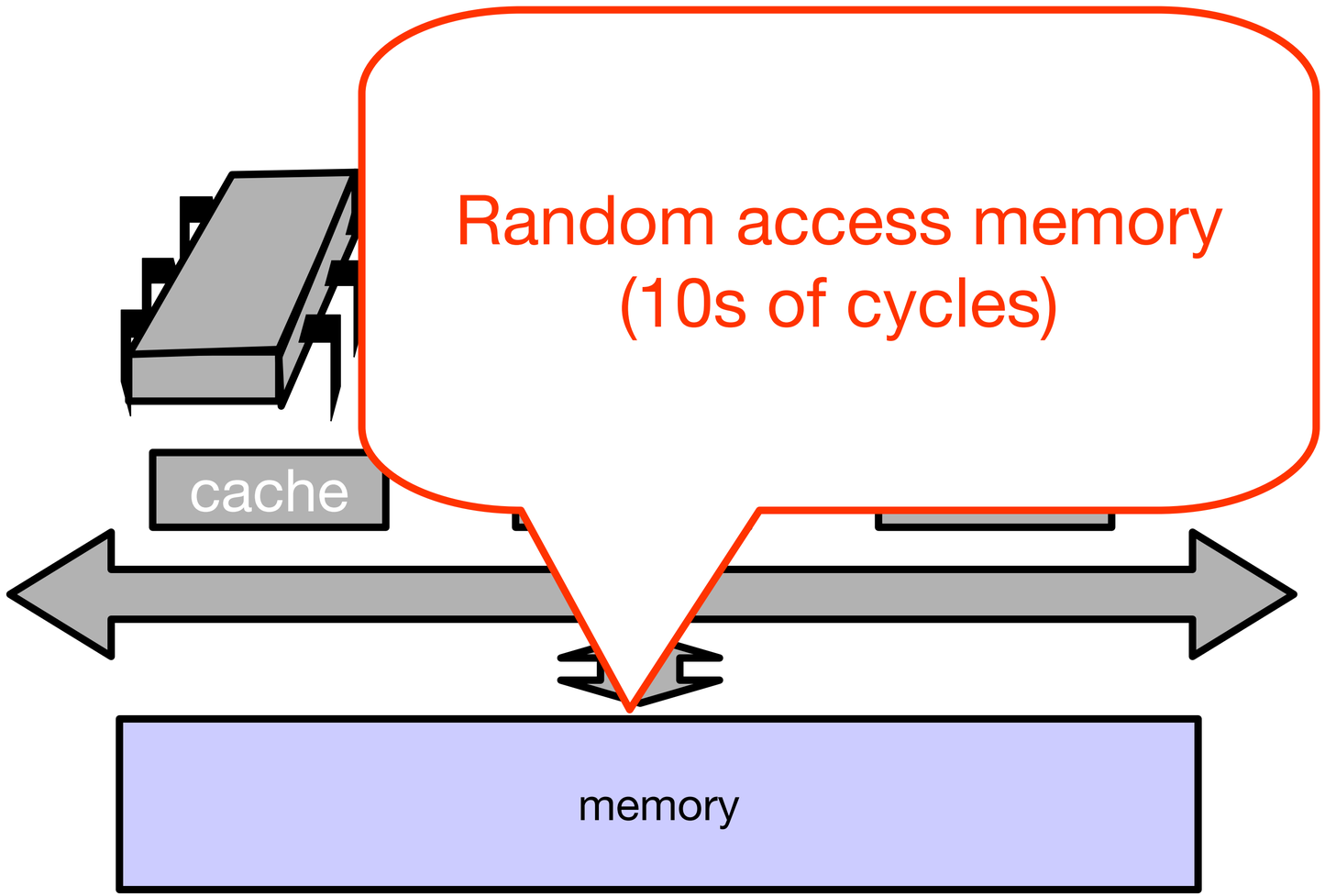
- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

Most modern desktop/laptop CPUs

MIMD Bus-Based Architectures

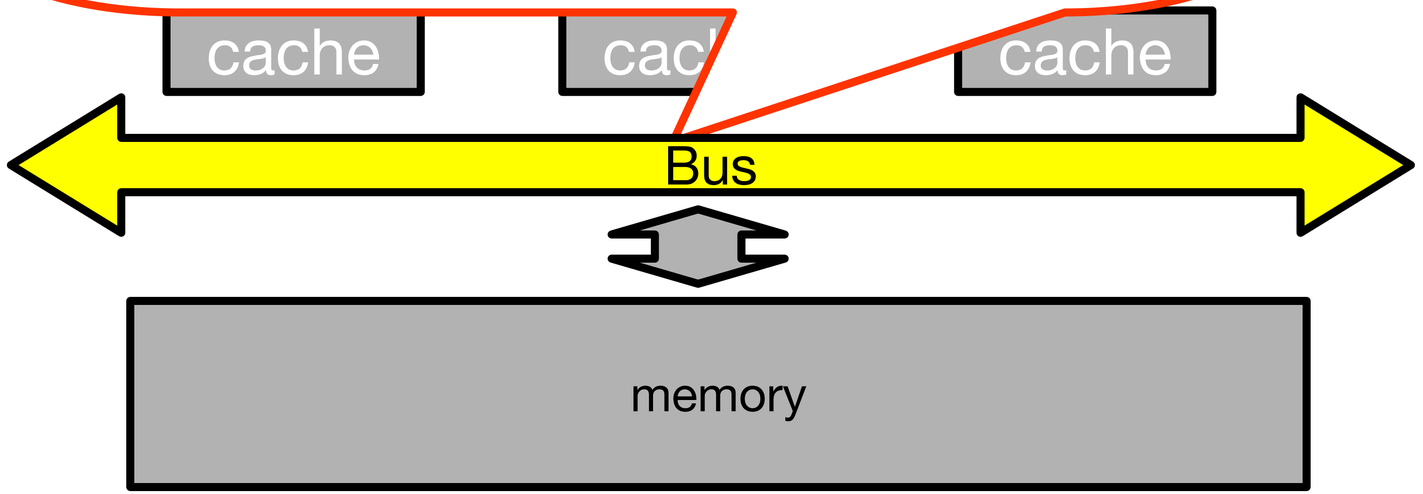


Bus-Based Architectures



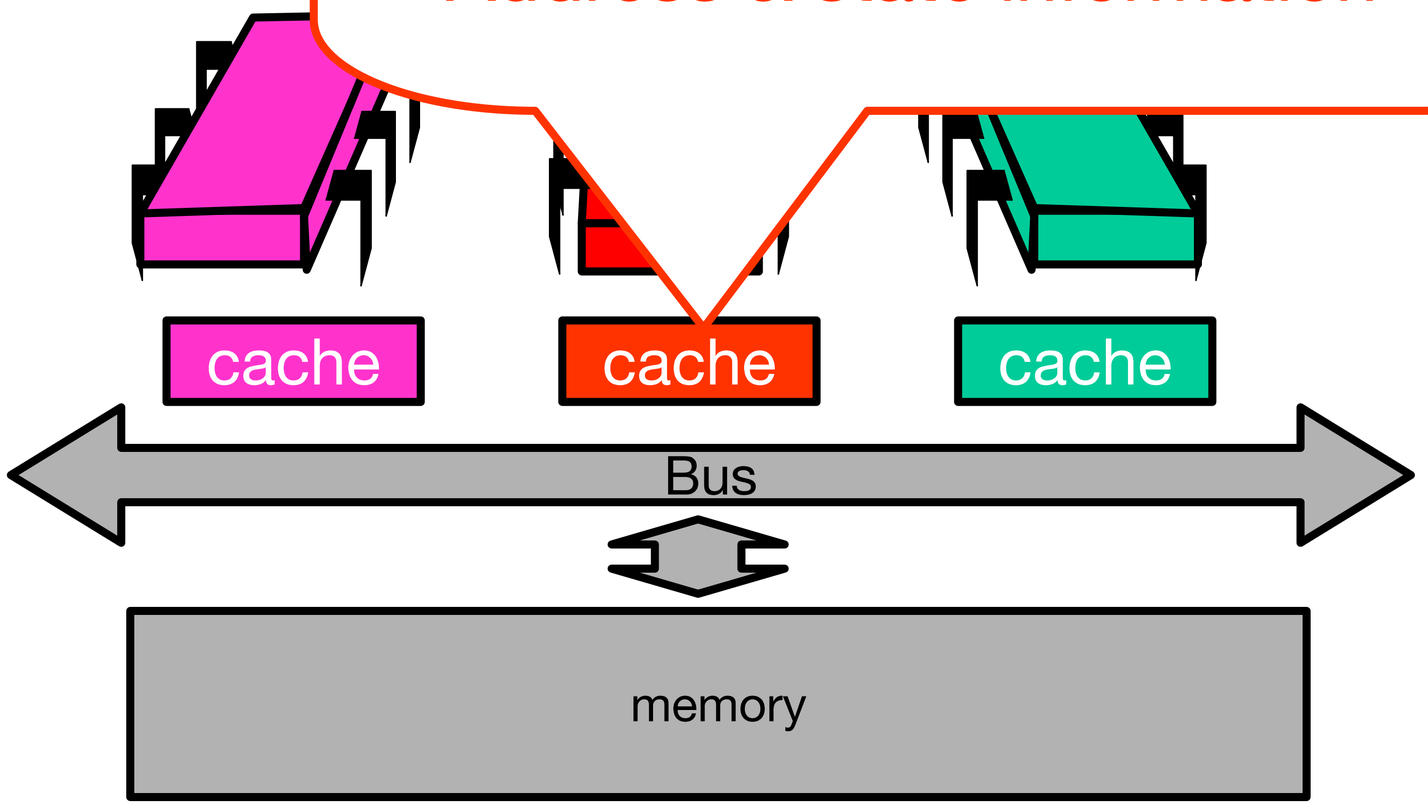
Bus-Based Architectures

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



Bus-Ba

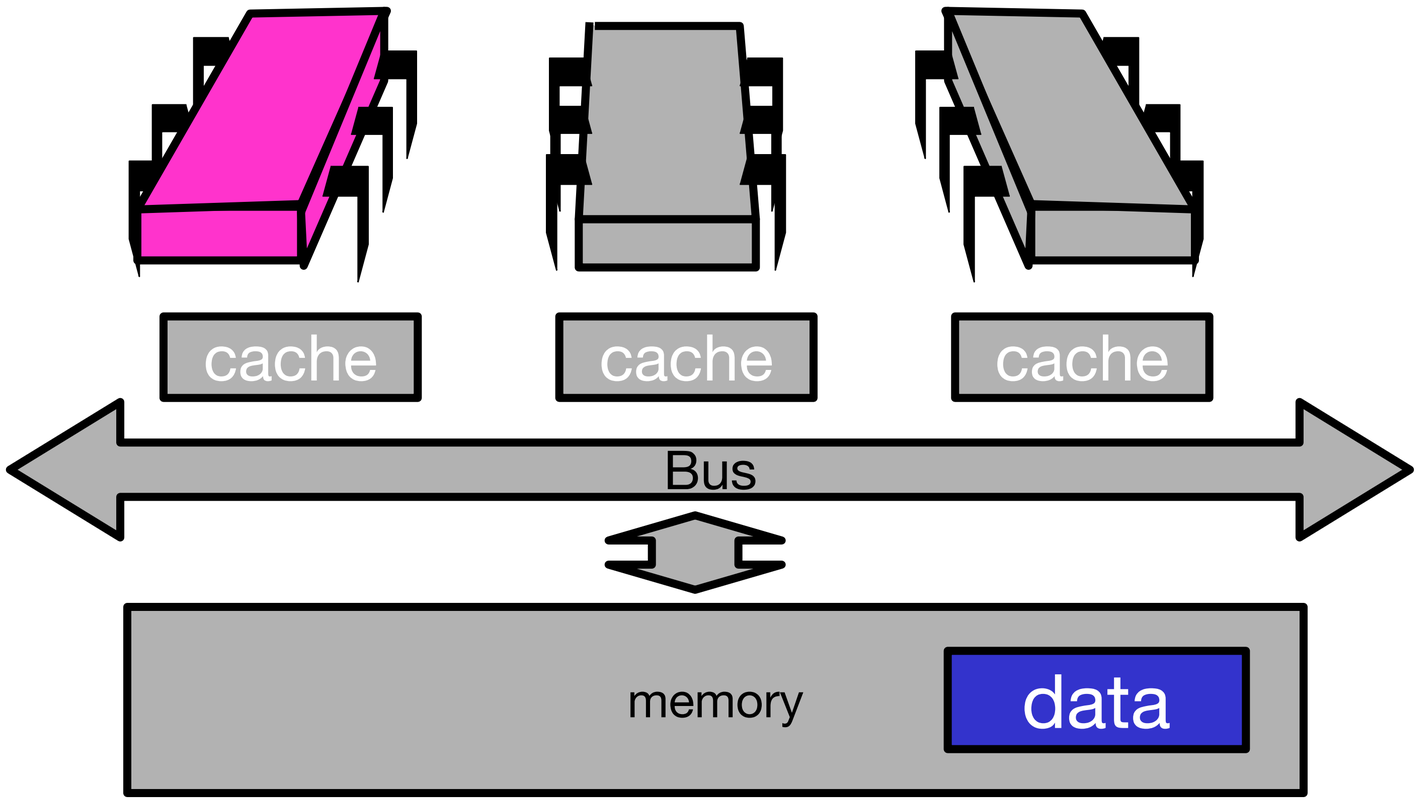
Per-Processor Caches
- Small
- Fast: 1 or 2 cycles
- Address & state information



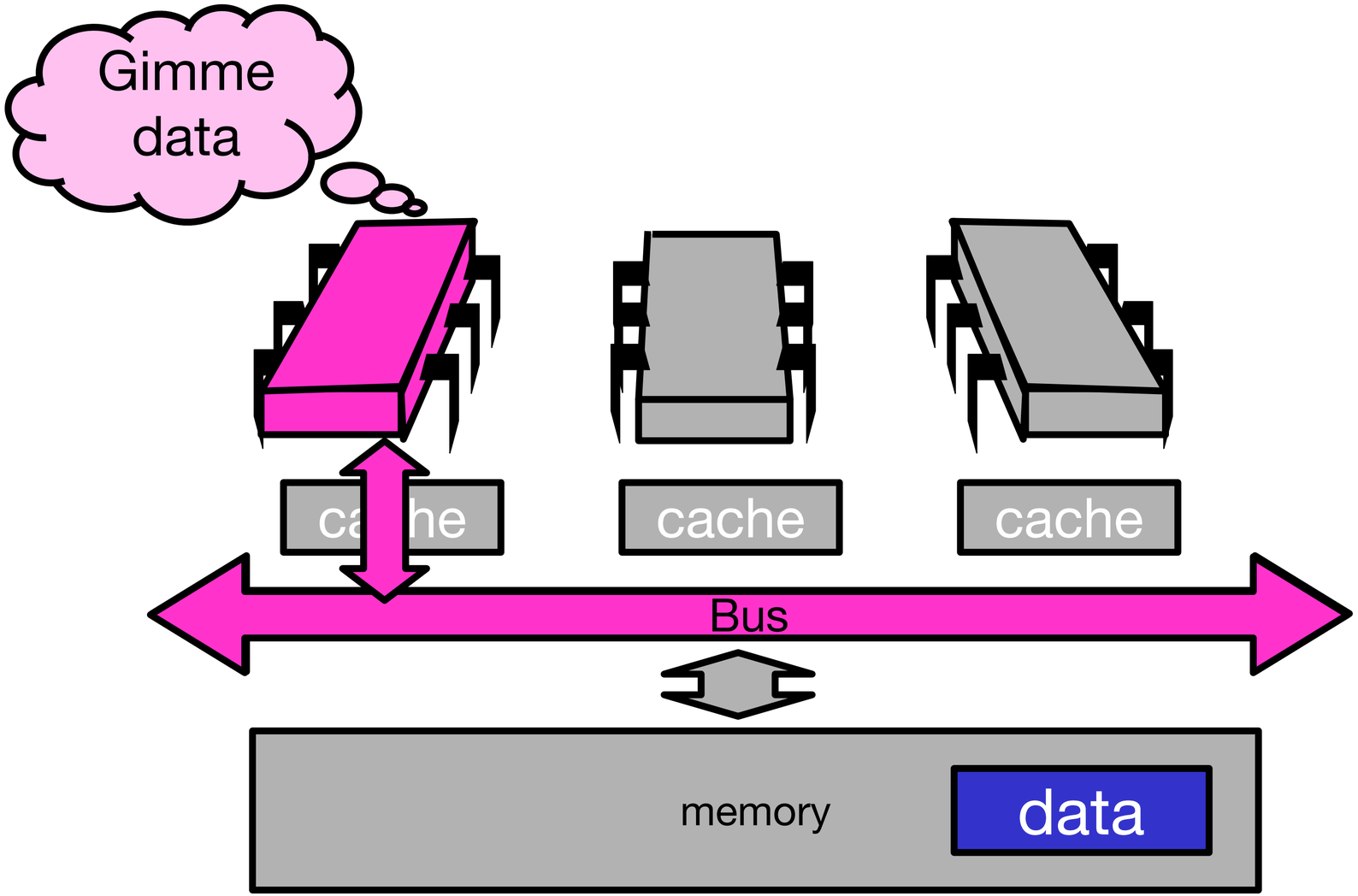
Jargon Watch

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

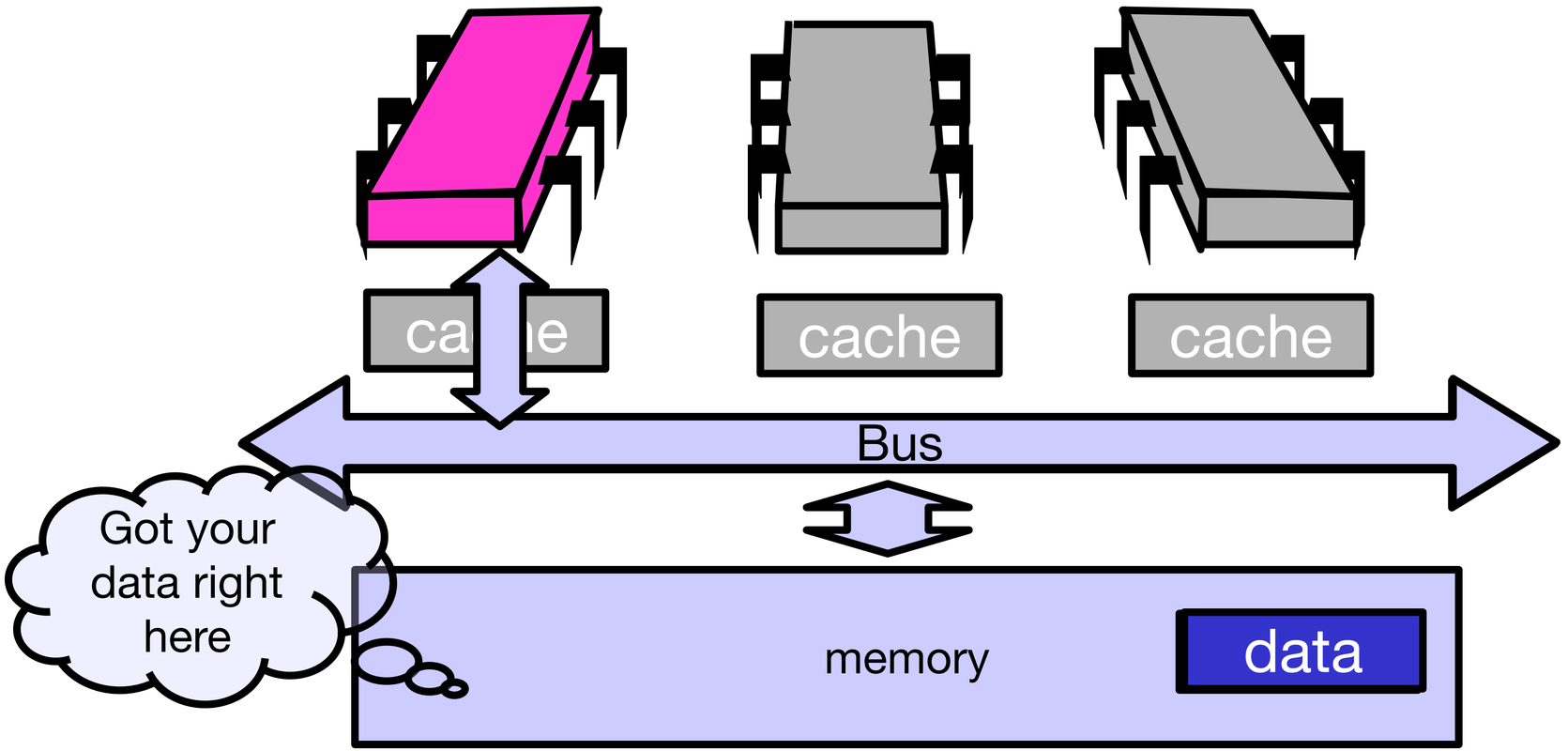
Processor Issues Load Request



Processor Issues Load Request

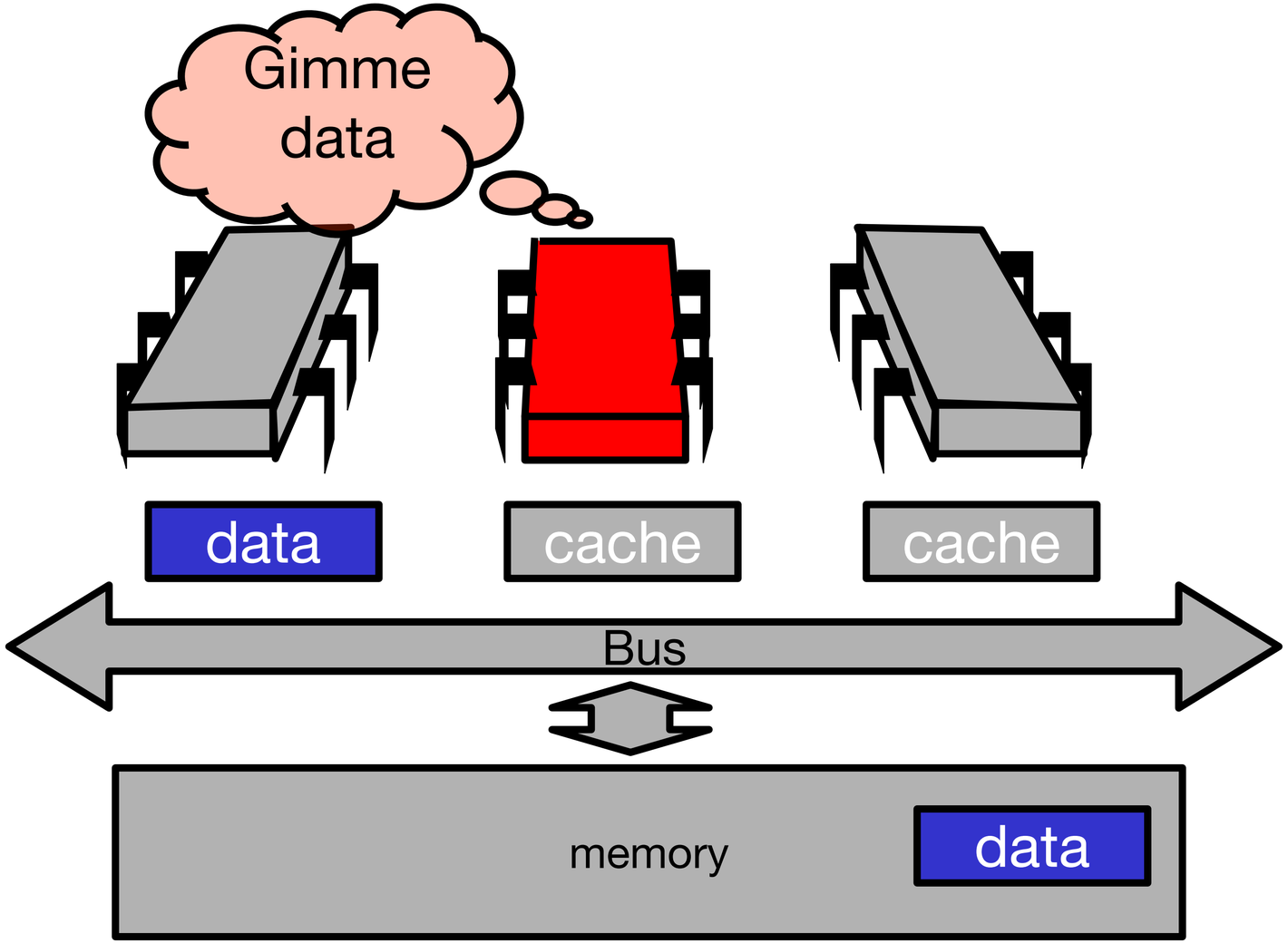


Memory Responds

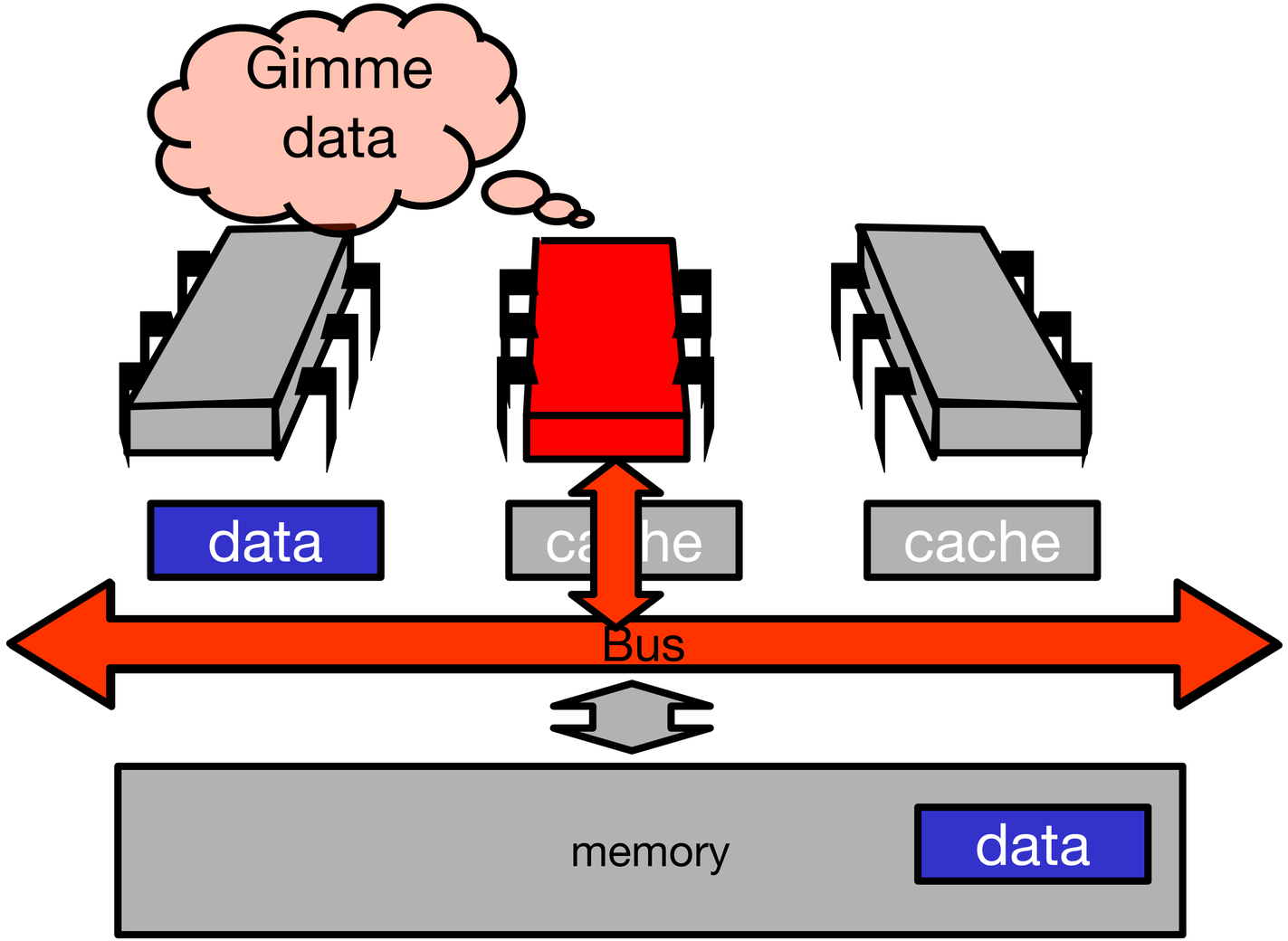


Art of Multiprocessor Programming

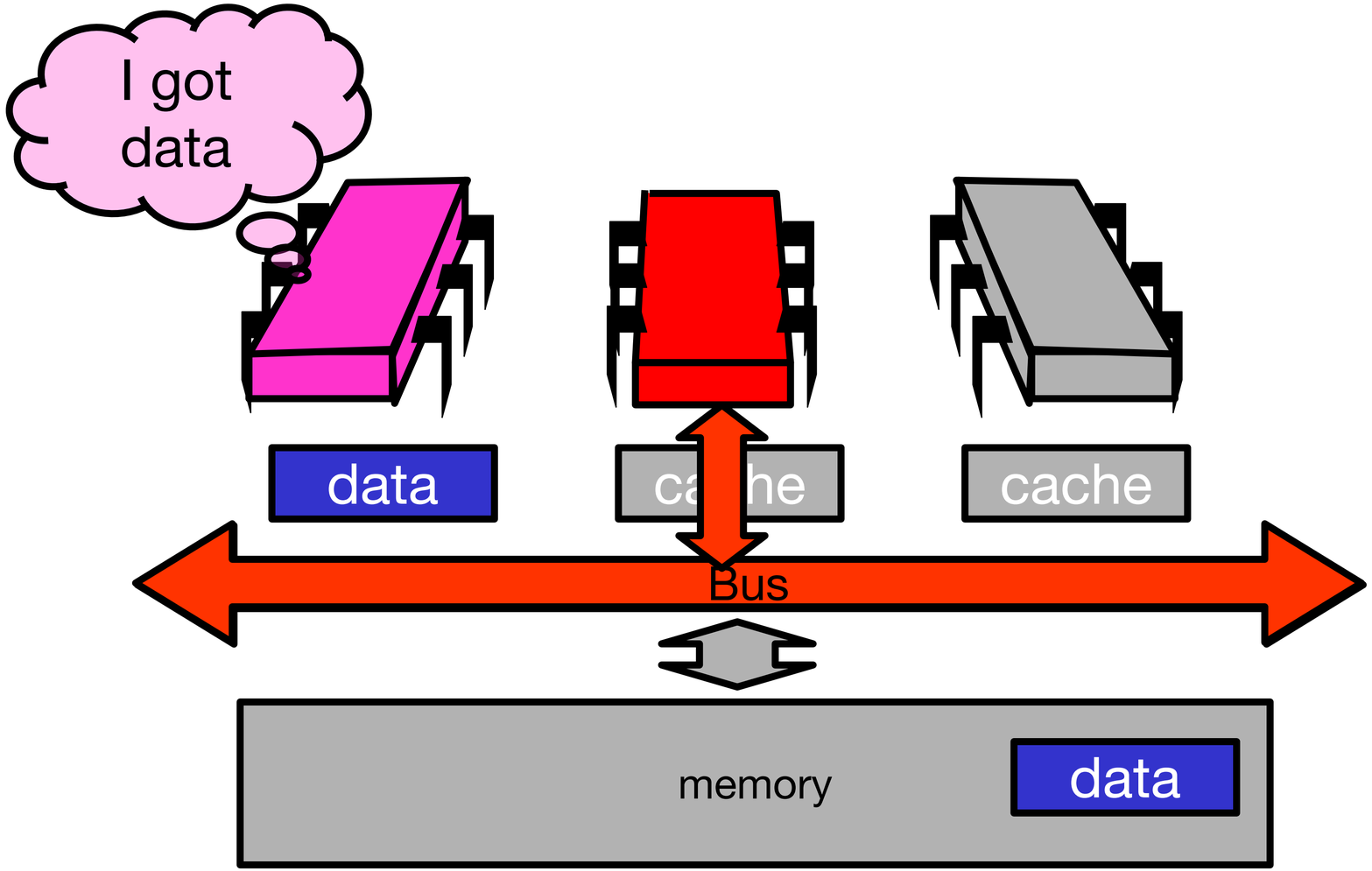
Processor Issues Load Request



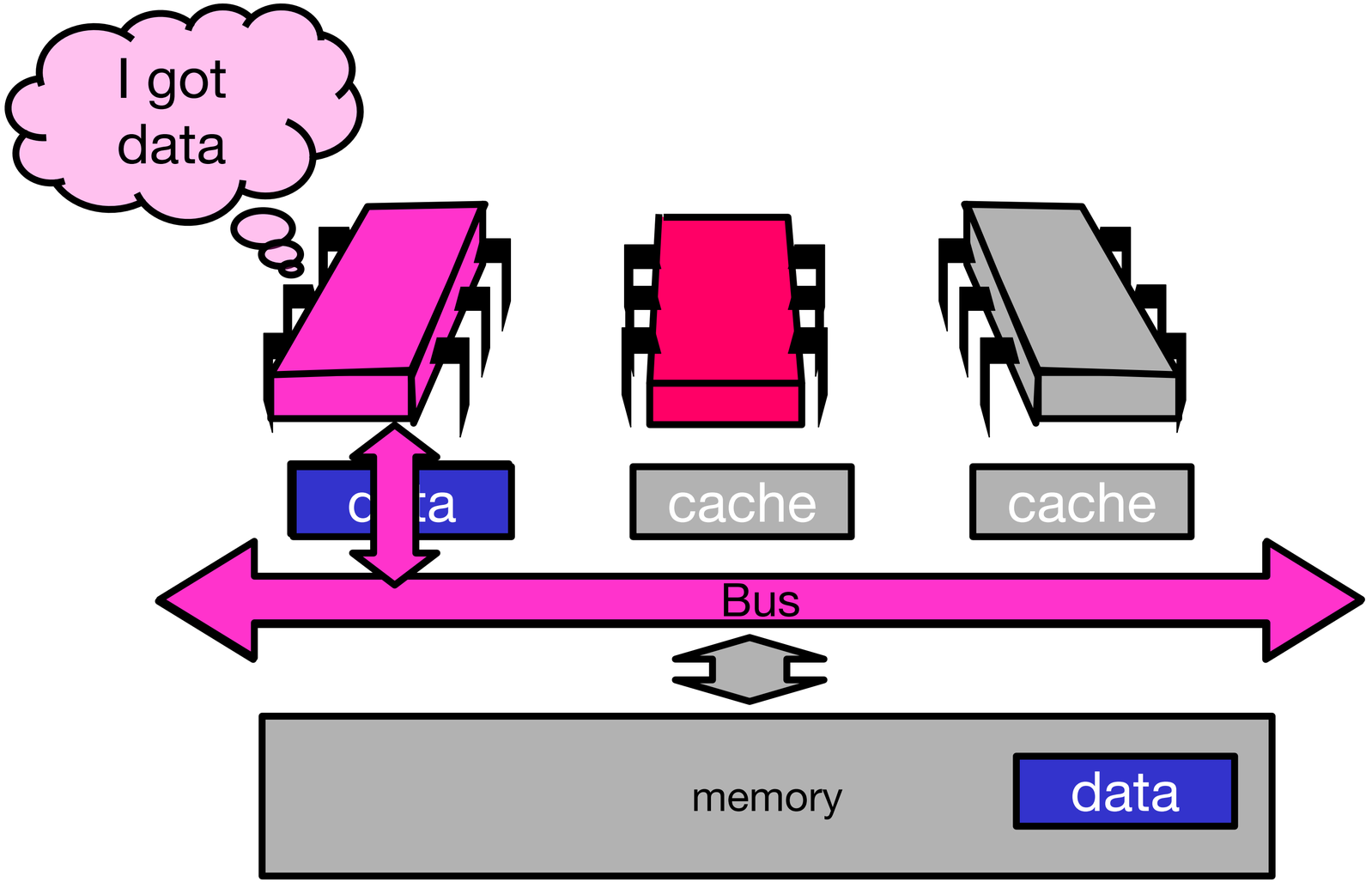
Processor Issues Load Request



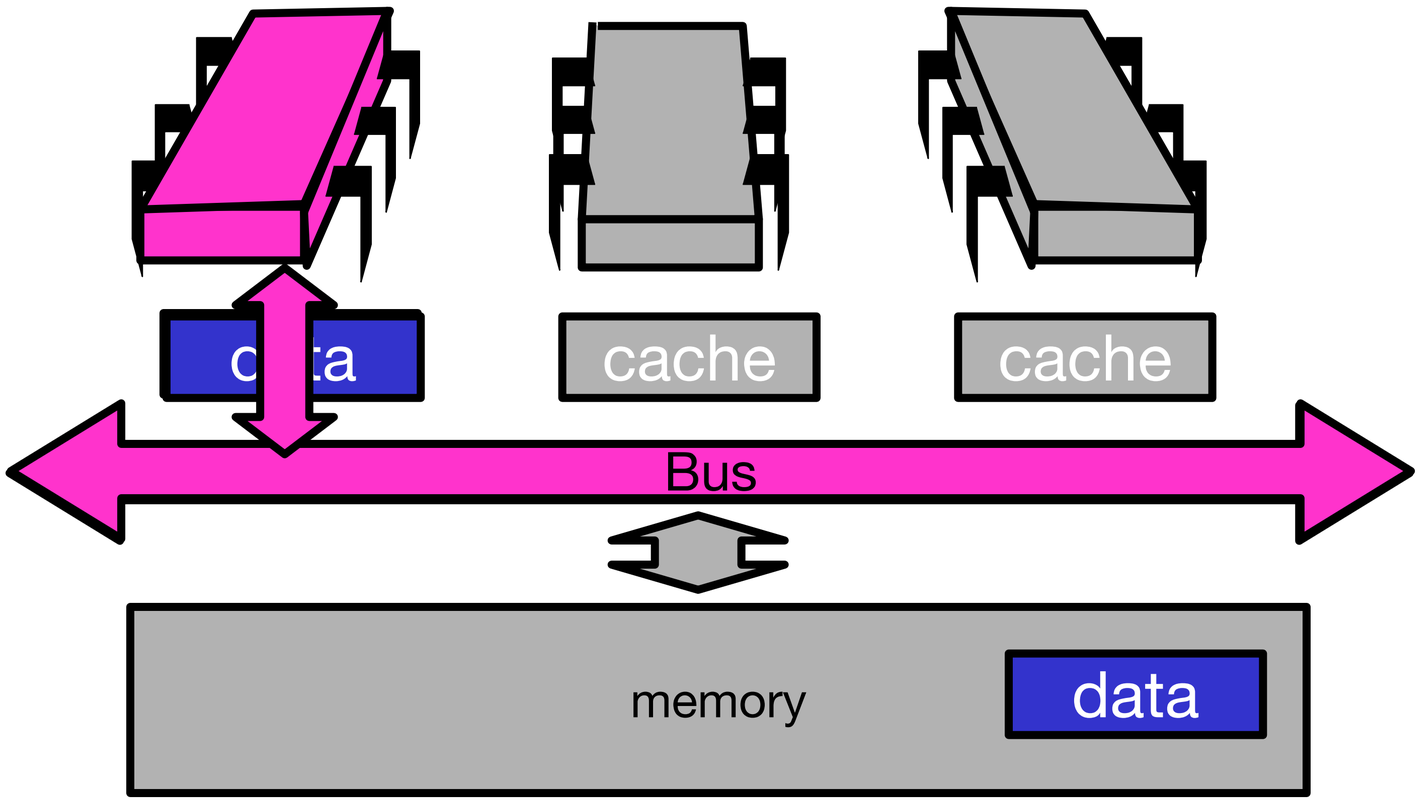
Processor Issues Load Request



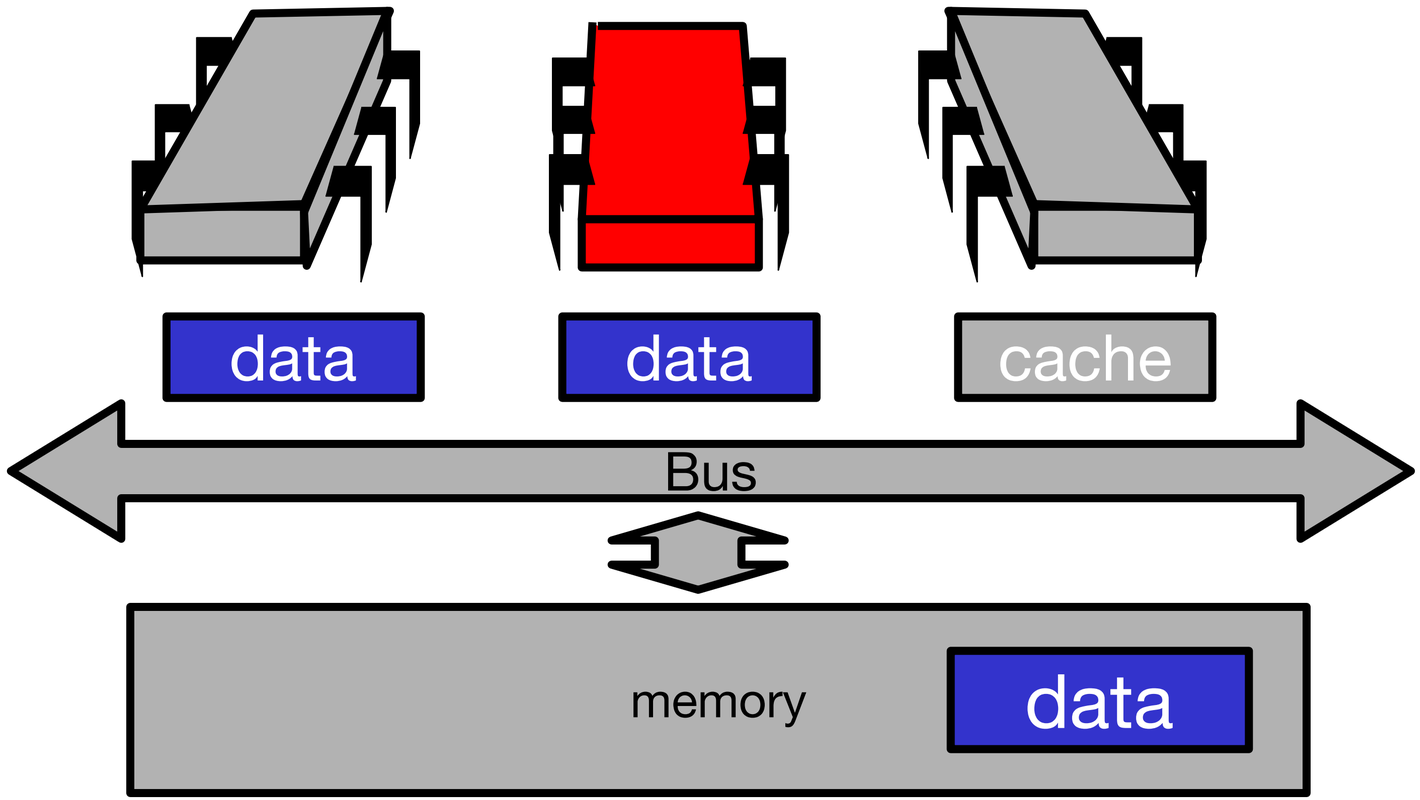
Other Processor Responds



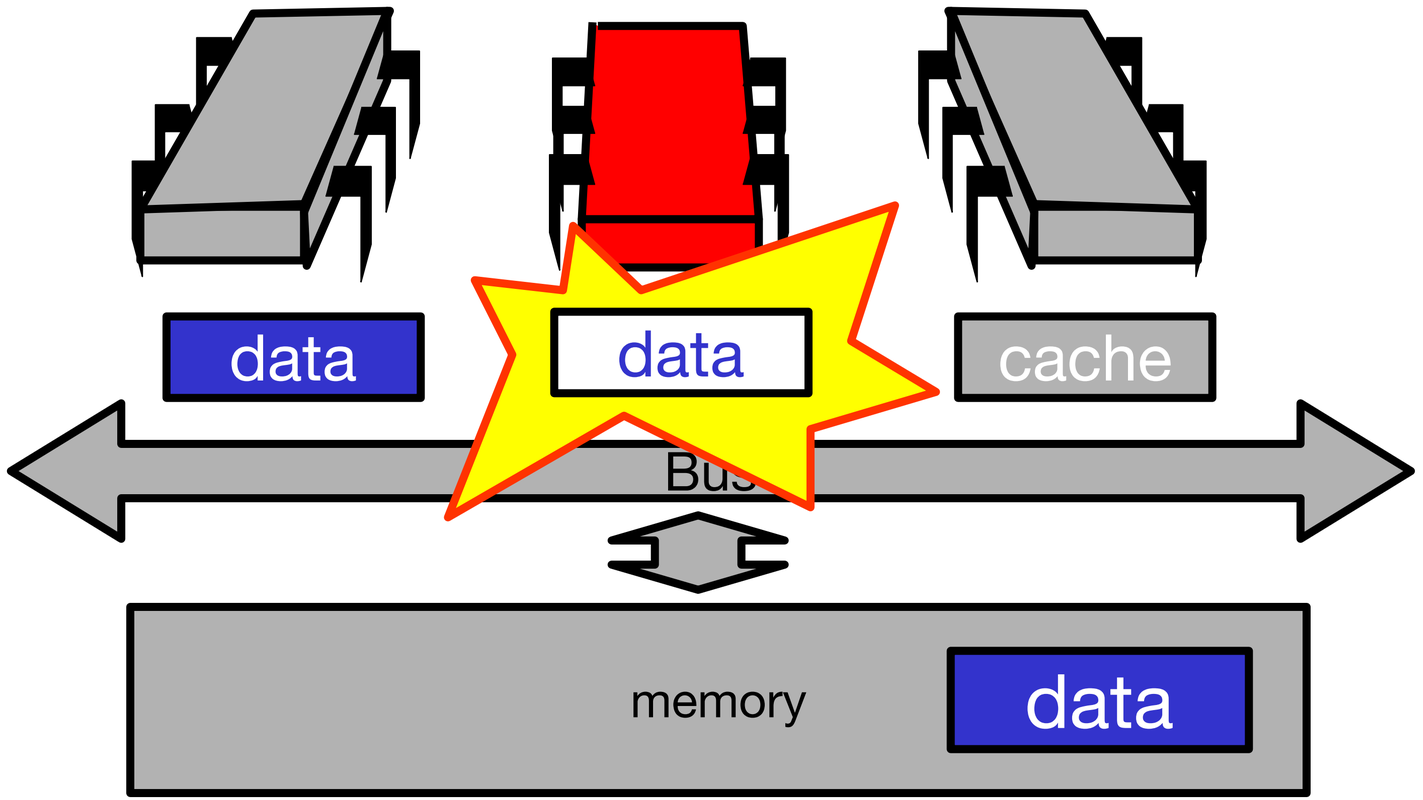
Other Processor Responds



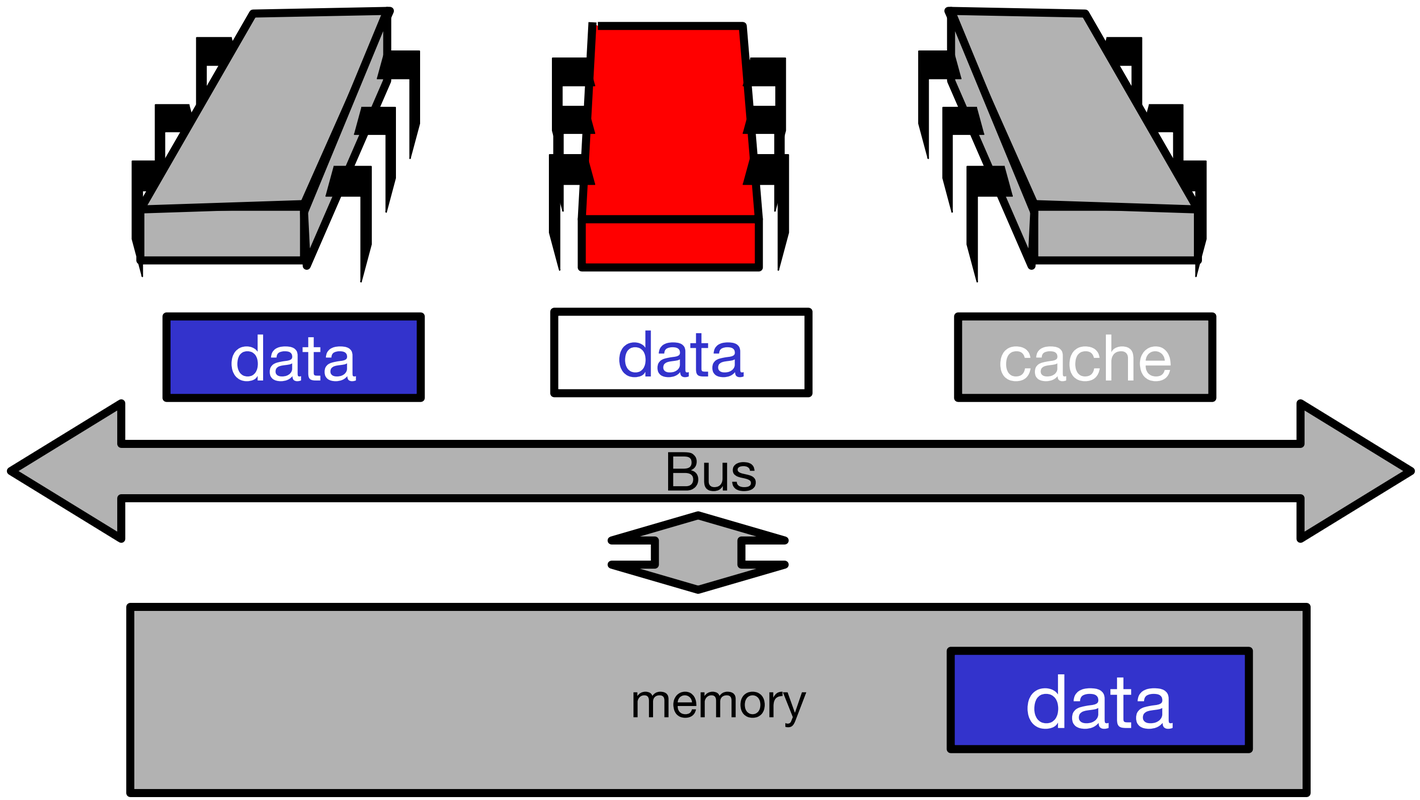
Modify Cached Data



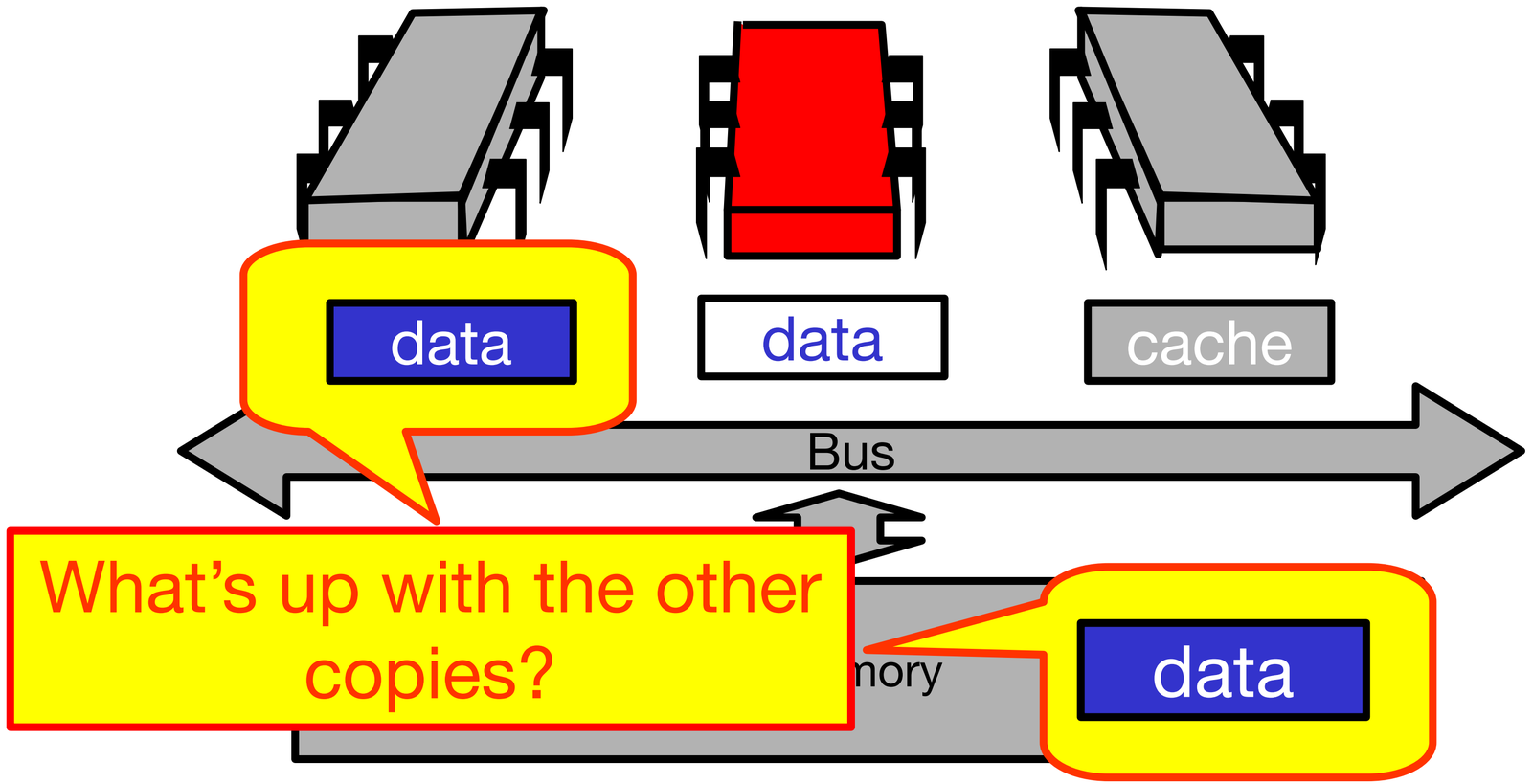
Modify Cached Data



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?

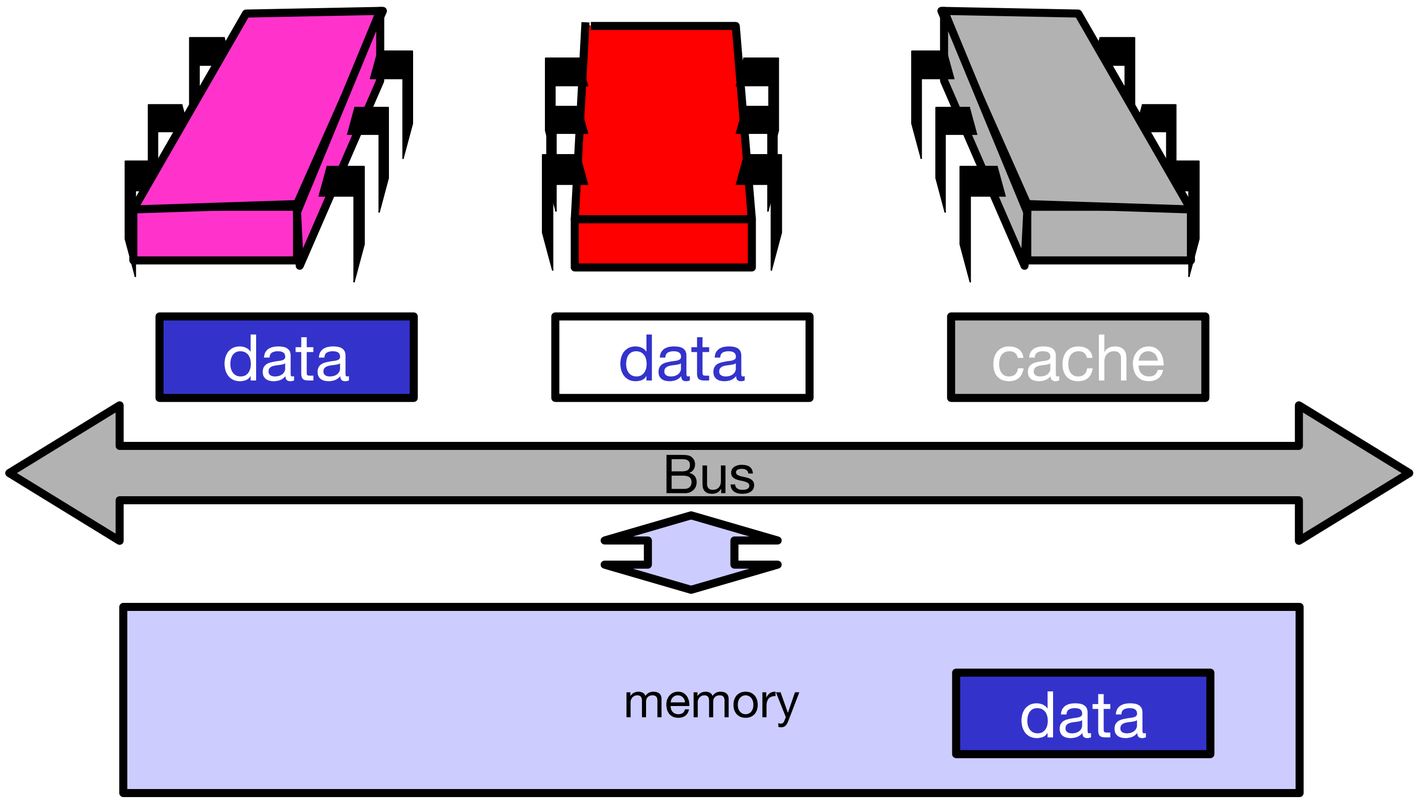
Write-Back Caches

- 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 …

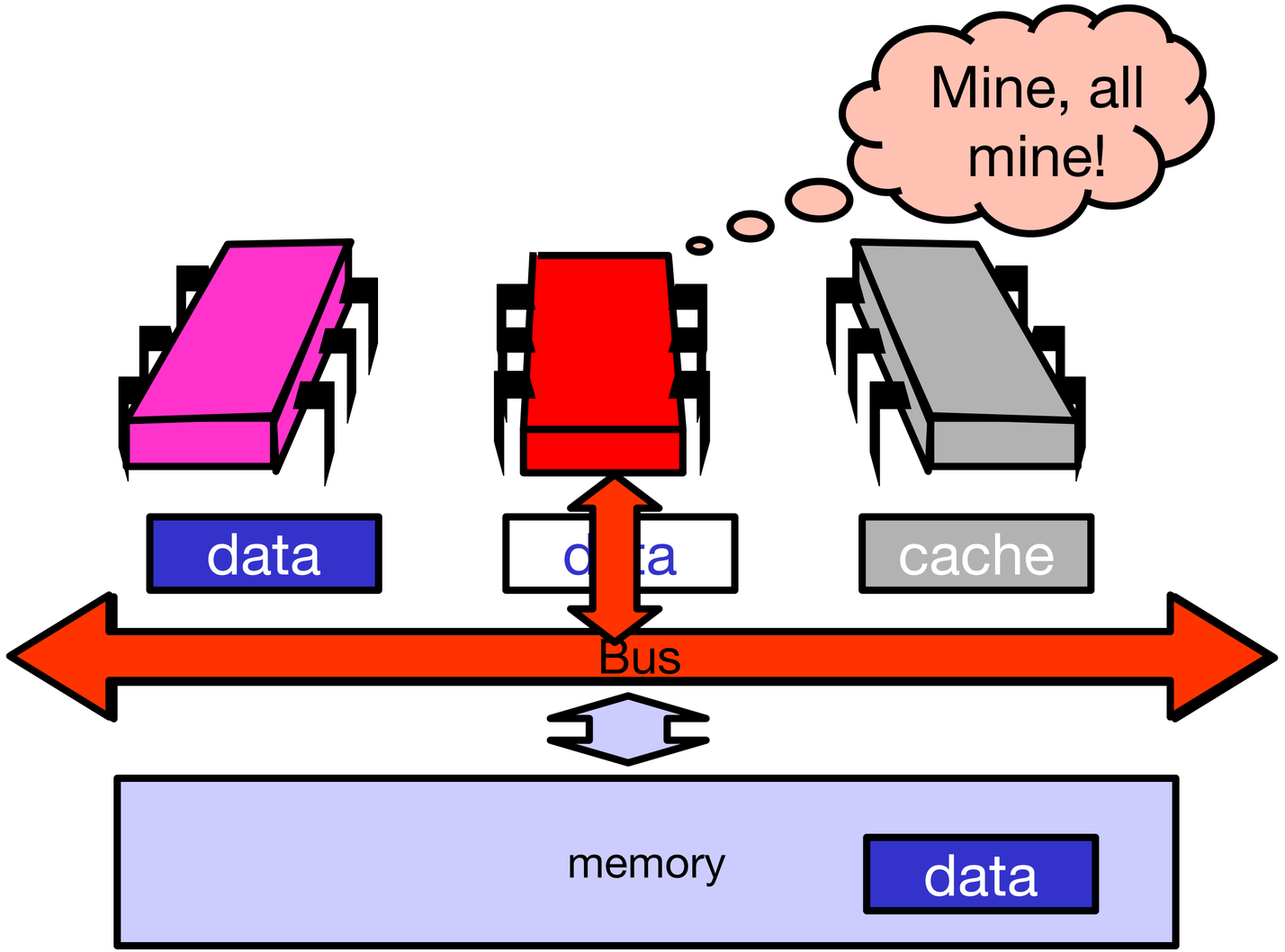
Write-Back Caches

- Cache entry has three states
  - Invalid: contains raw seething bits
  - Valid: I can read but I can't write
  - Dirty: Data has been modified
    - Intercept other load requests
    - Write back to memory before using cache

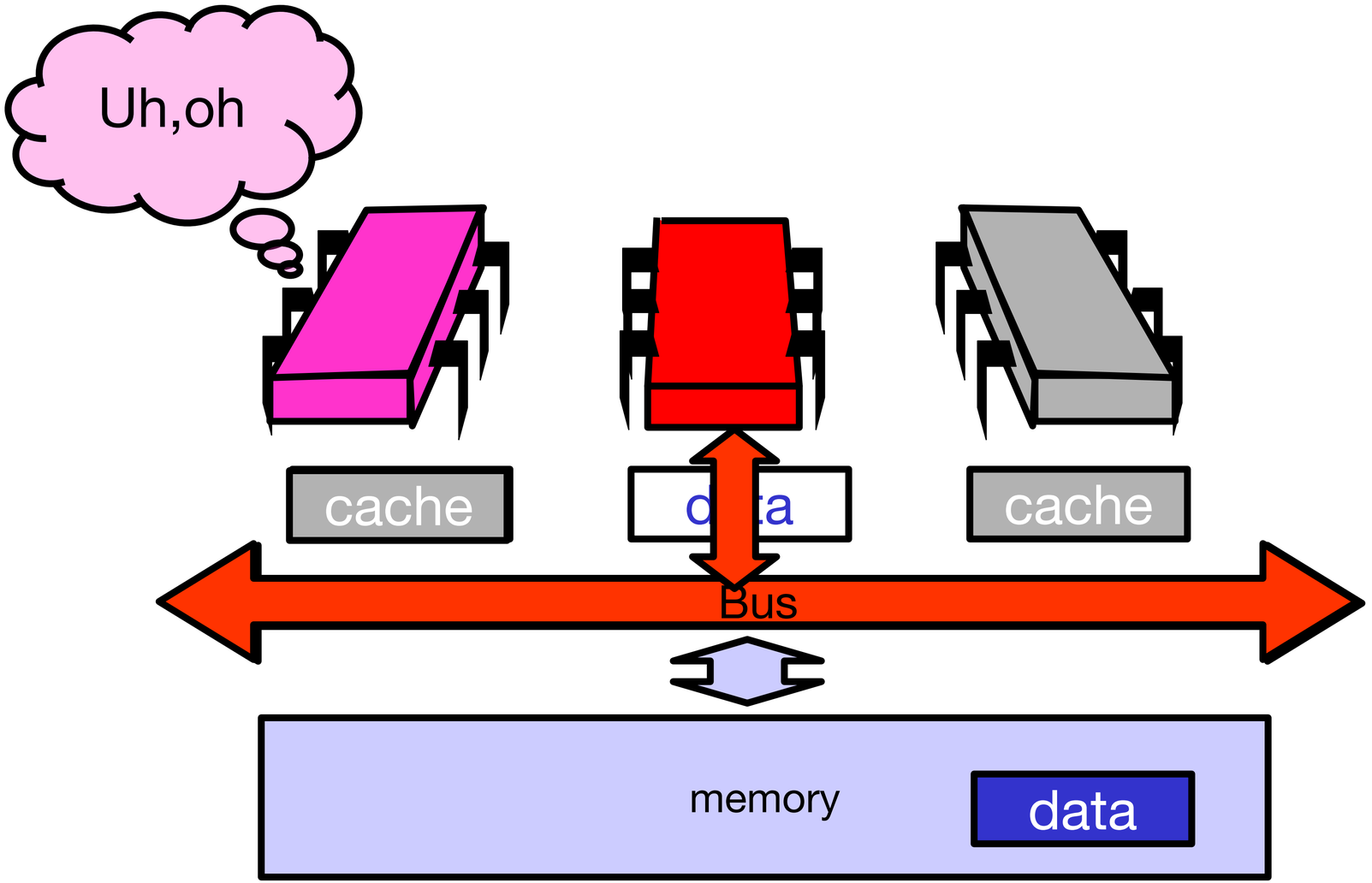
Invalidate



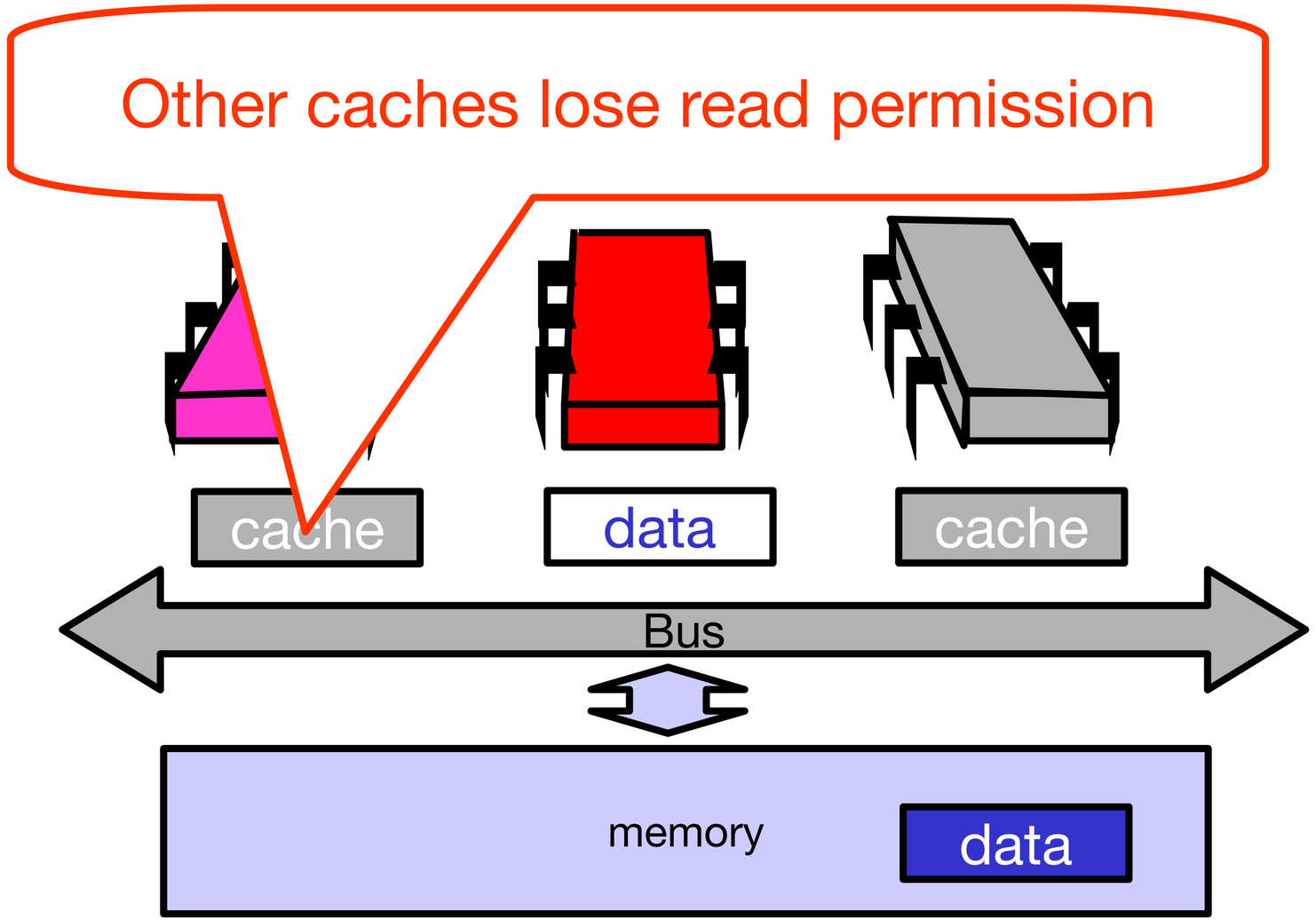
Invalidate



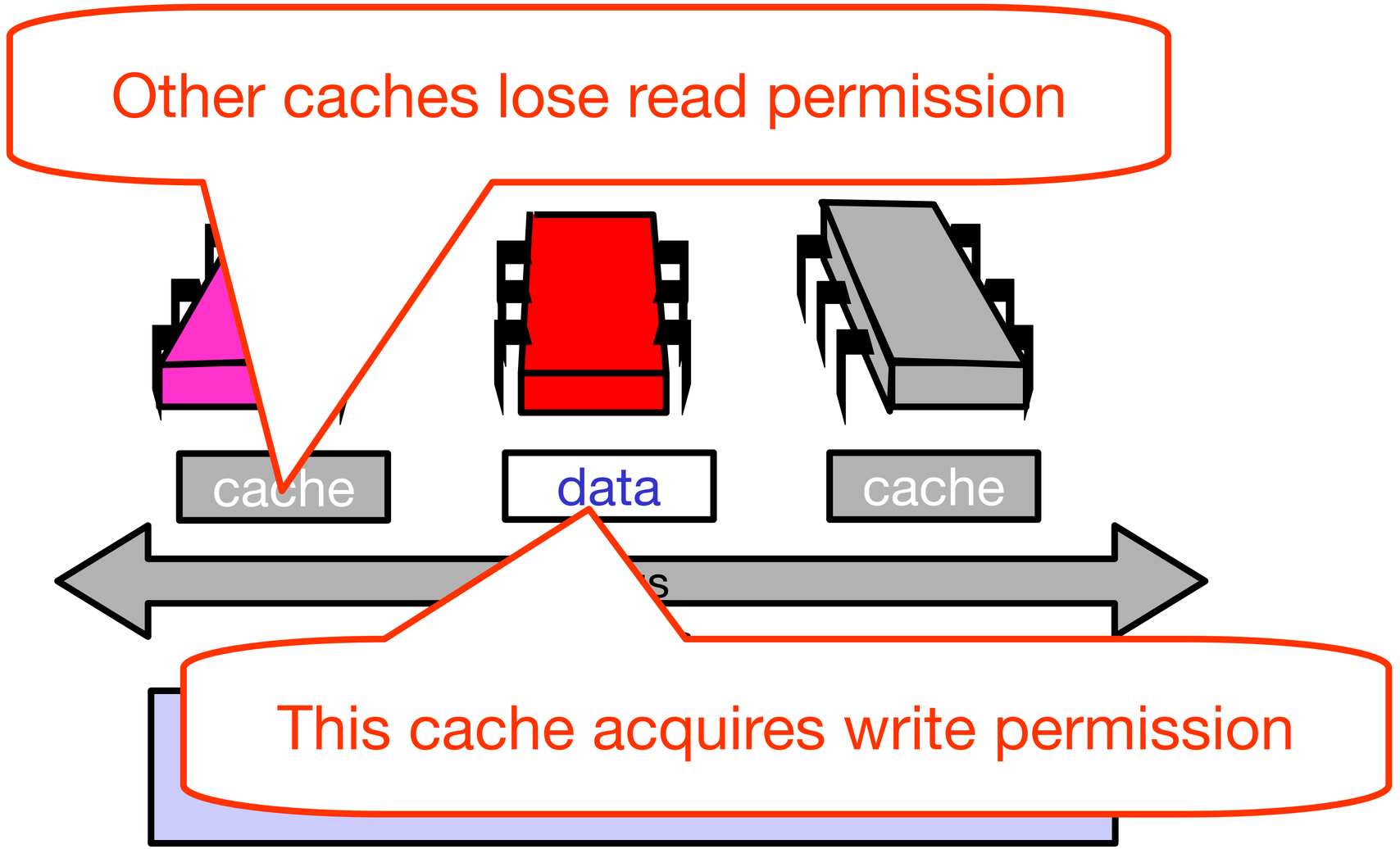
Invalidate



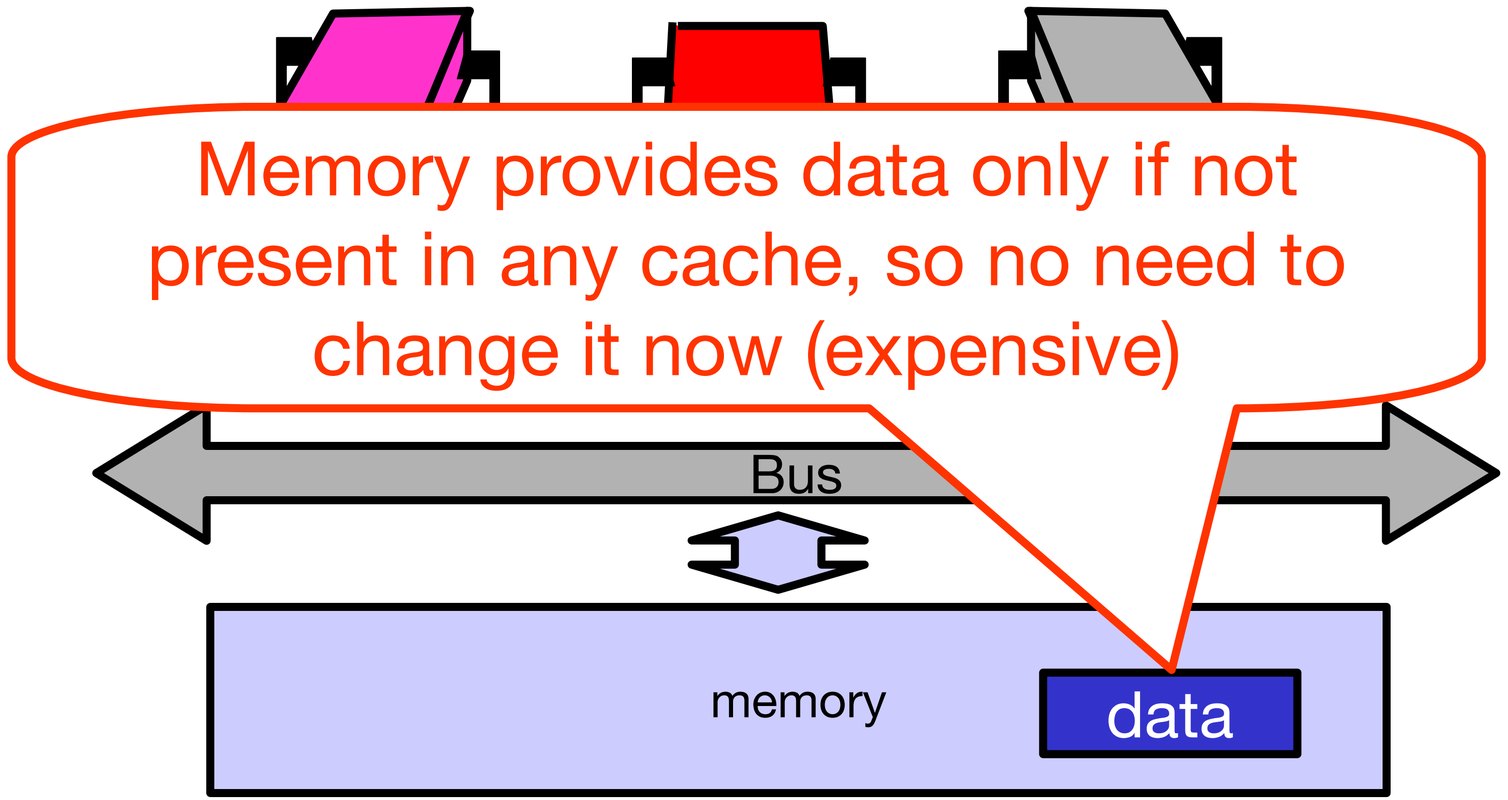
Invalidate


Other caches lose read permission
cache
data
cache
Bus
memory
data

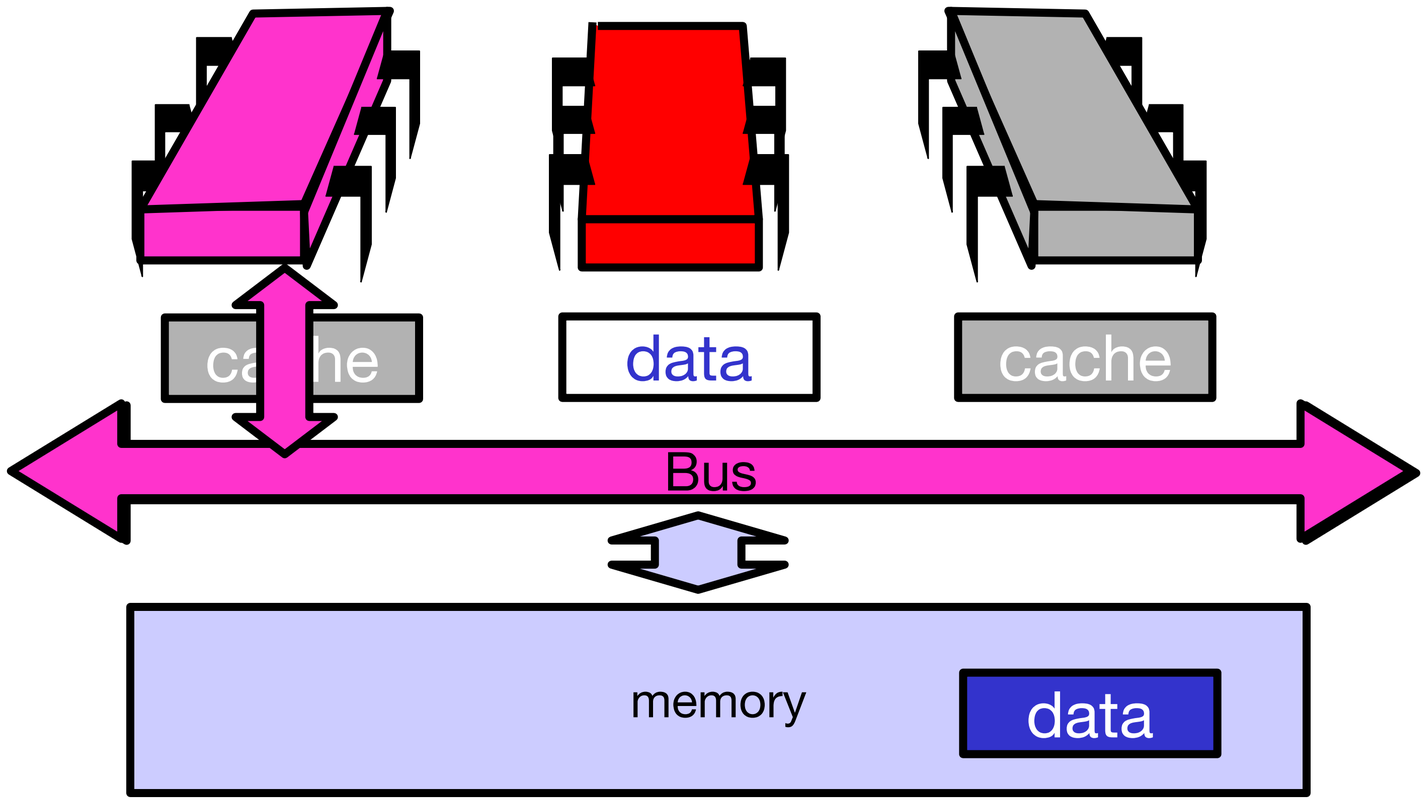
Invalidate


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

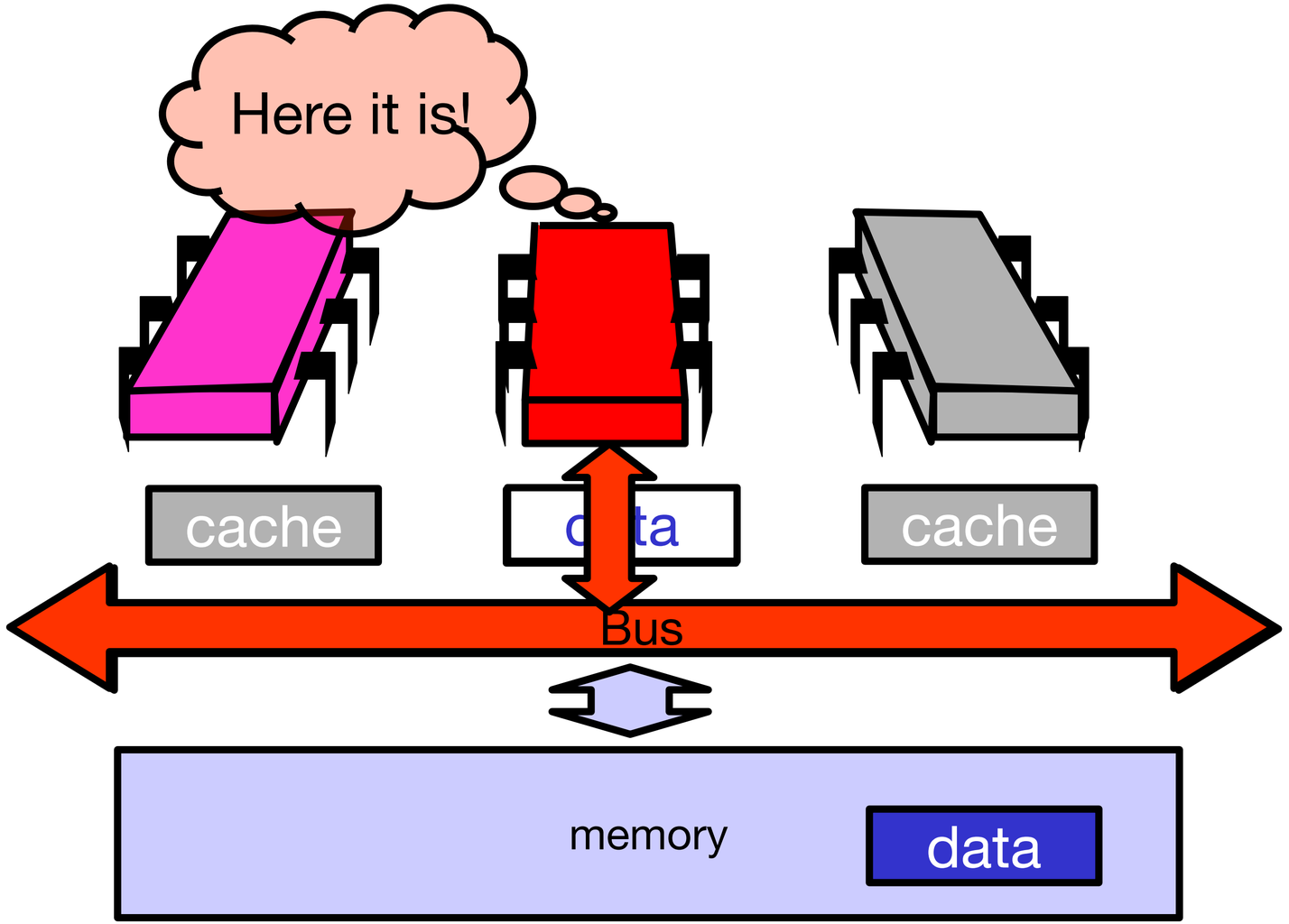
Invalidate


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

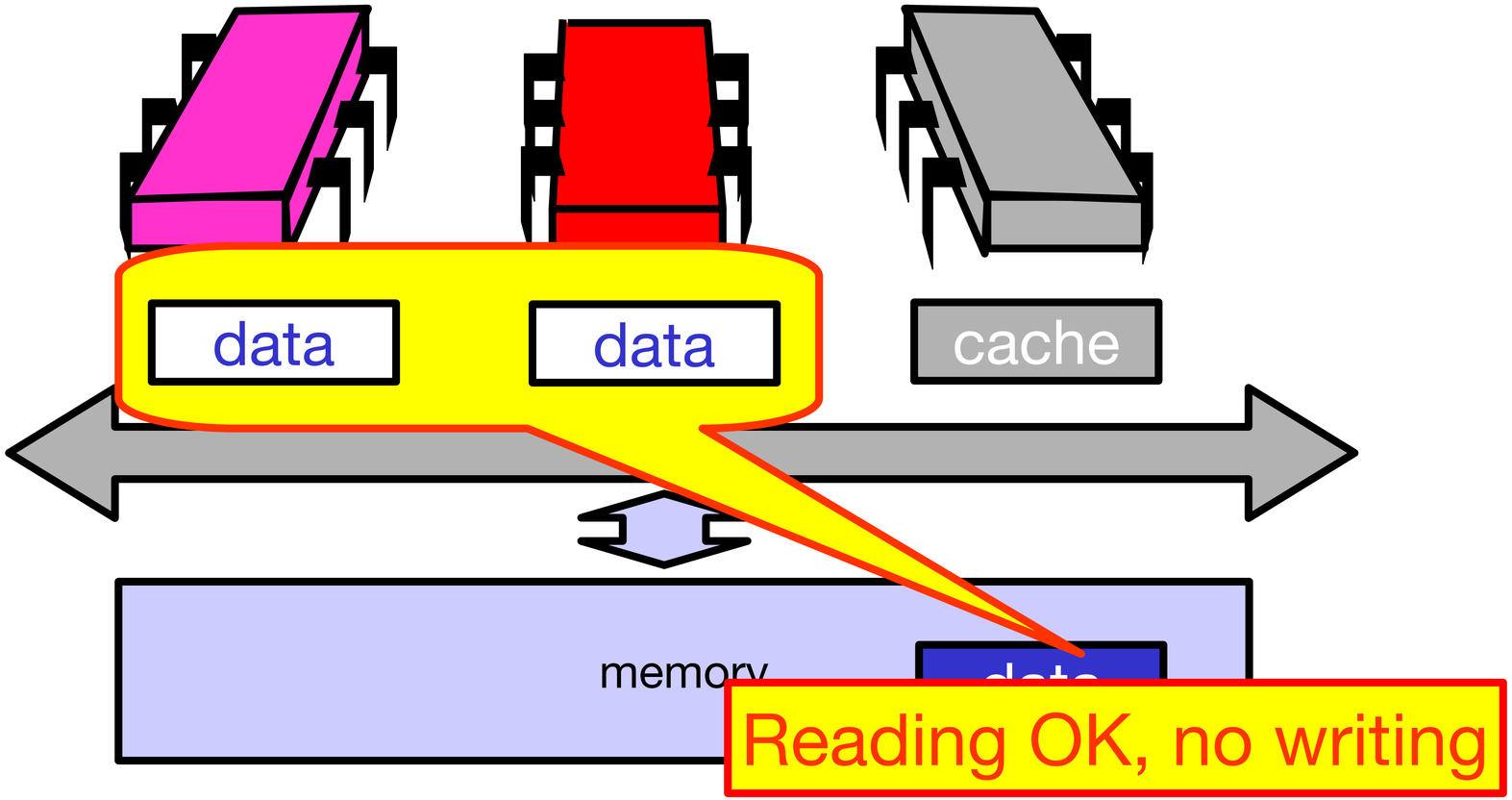
Another Processor Asks for Data



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

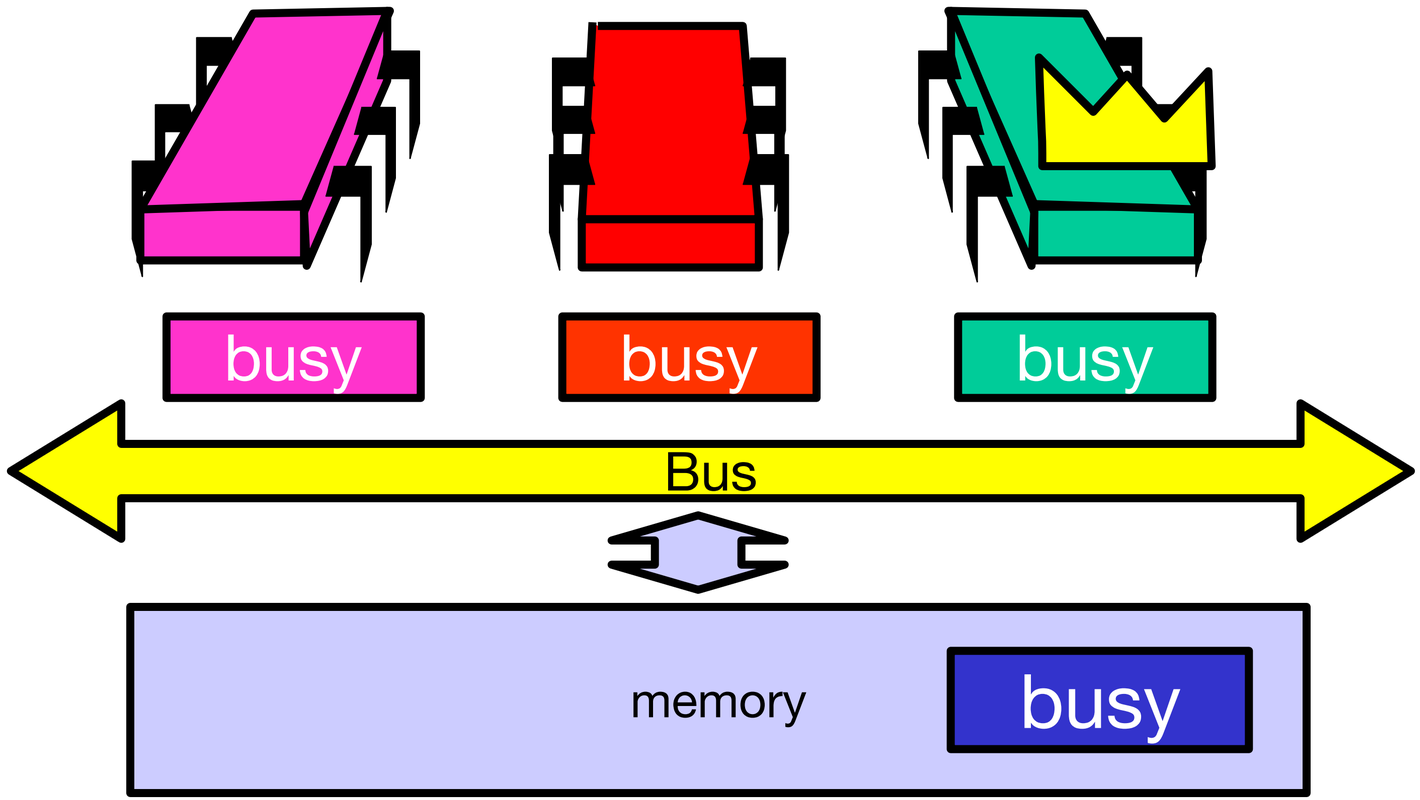
Simple TASLock

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

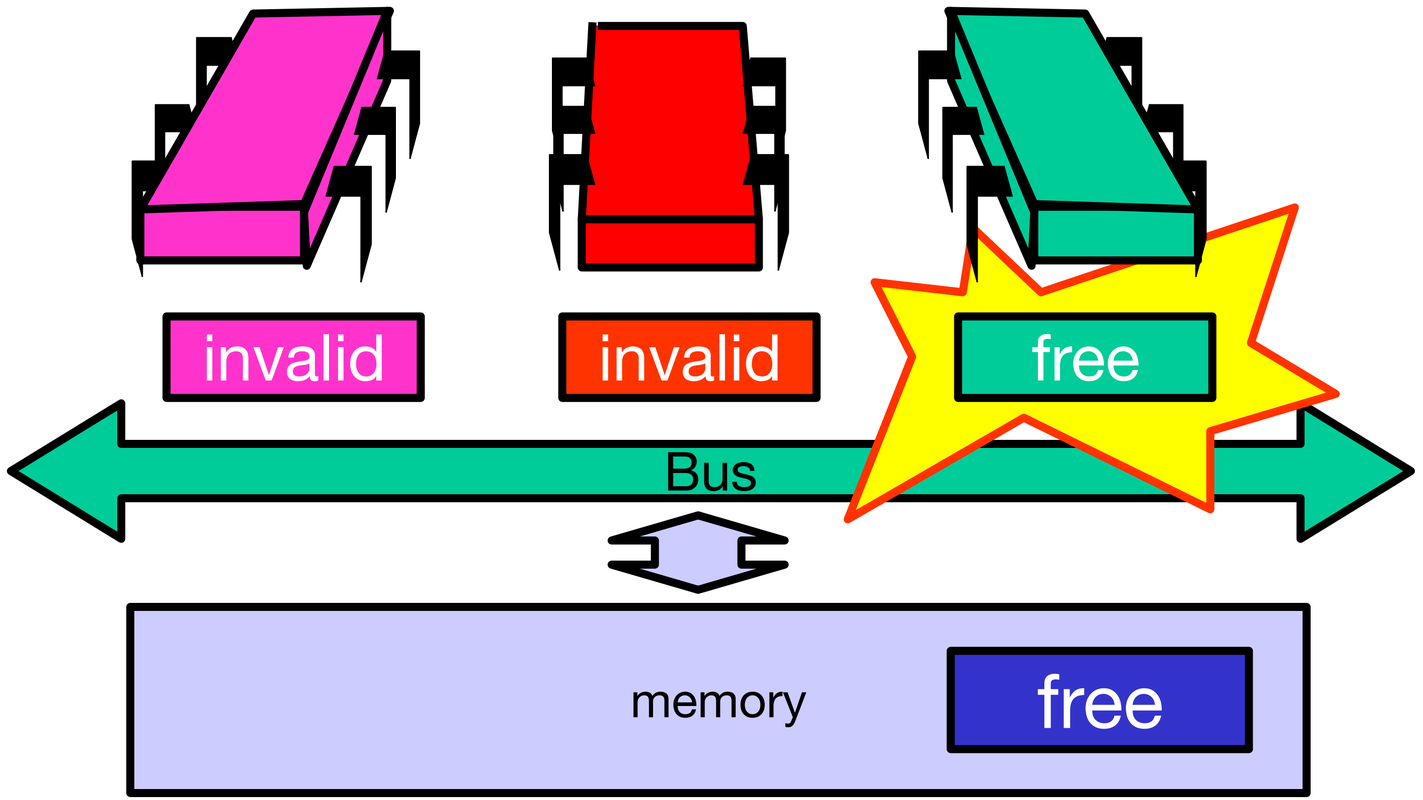
Test-and-test-and-set

- 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

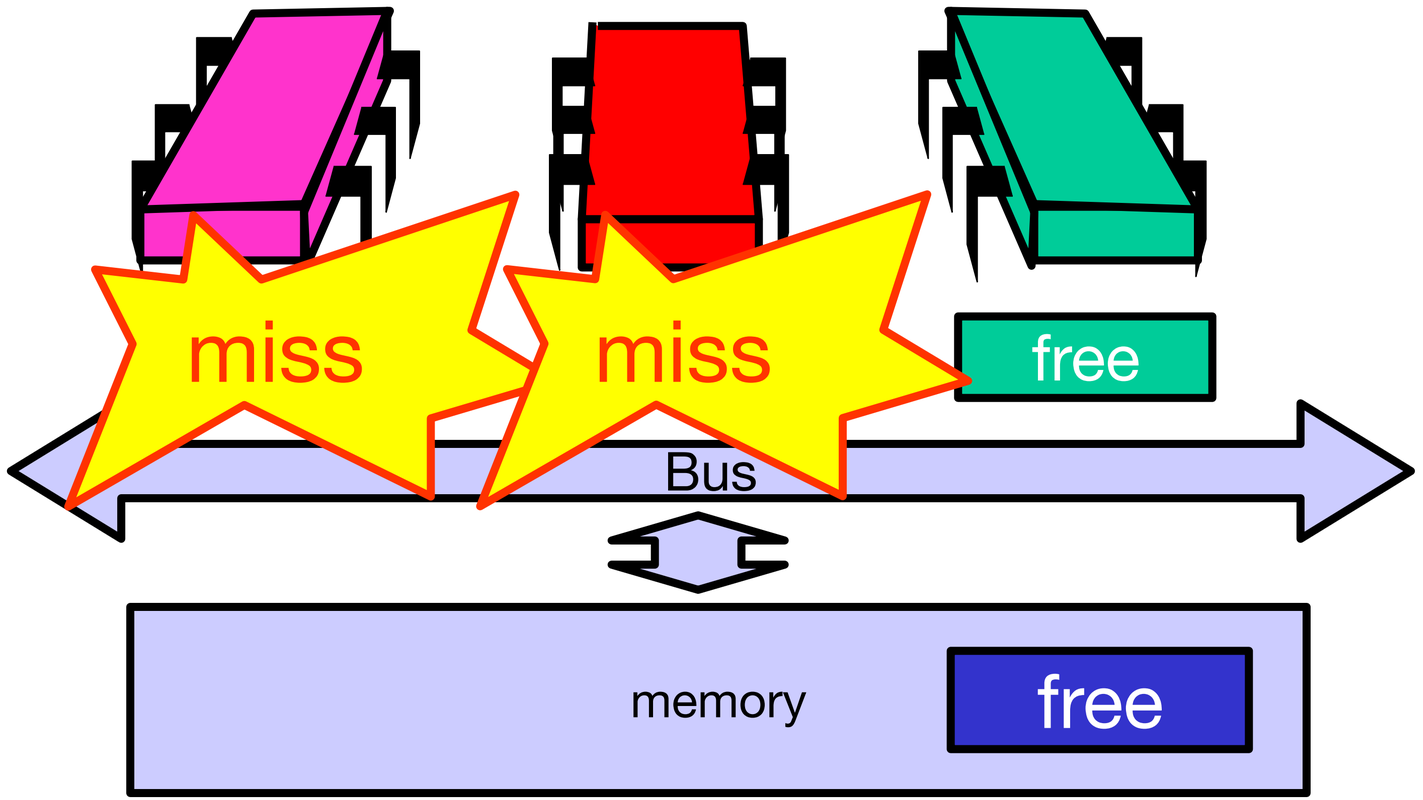


On Release



On Release

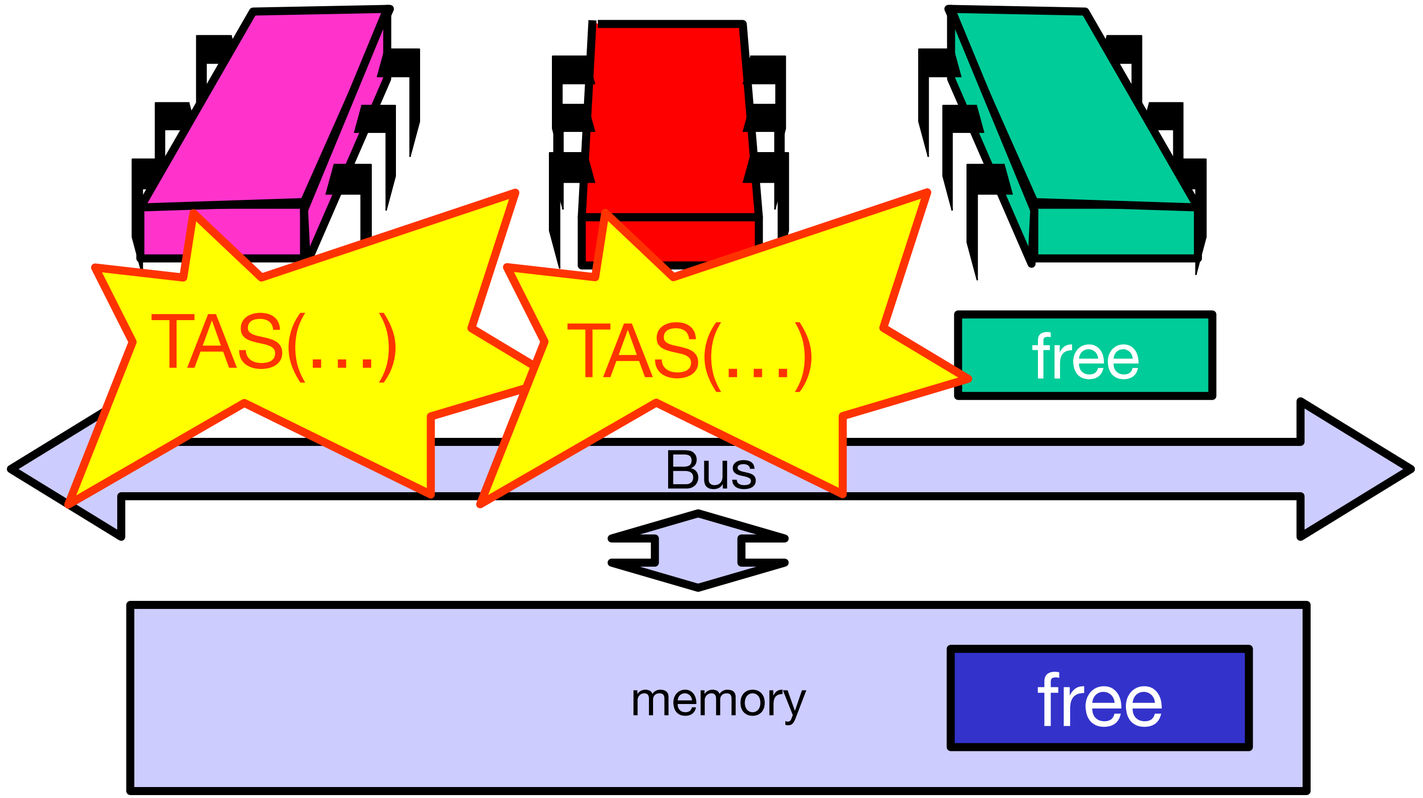
Everyone misses, rereads



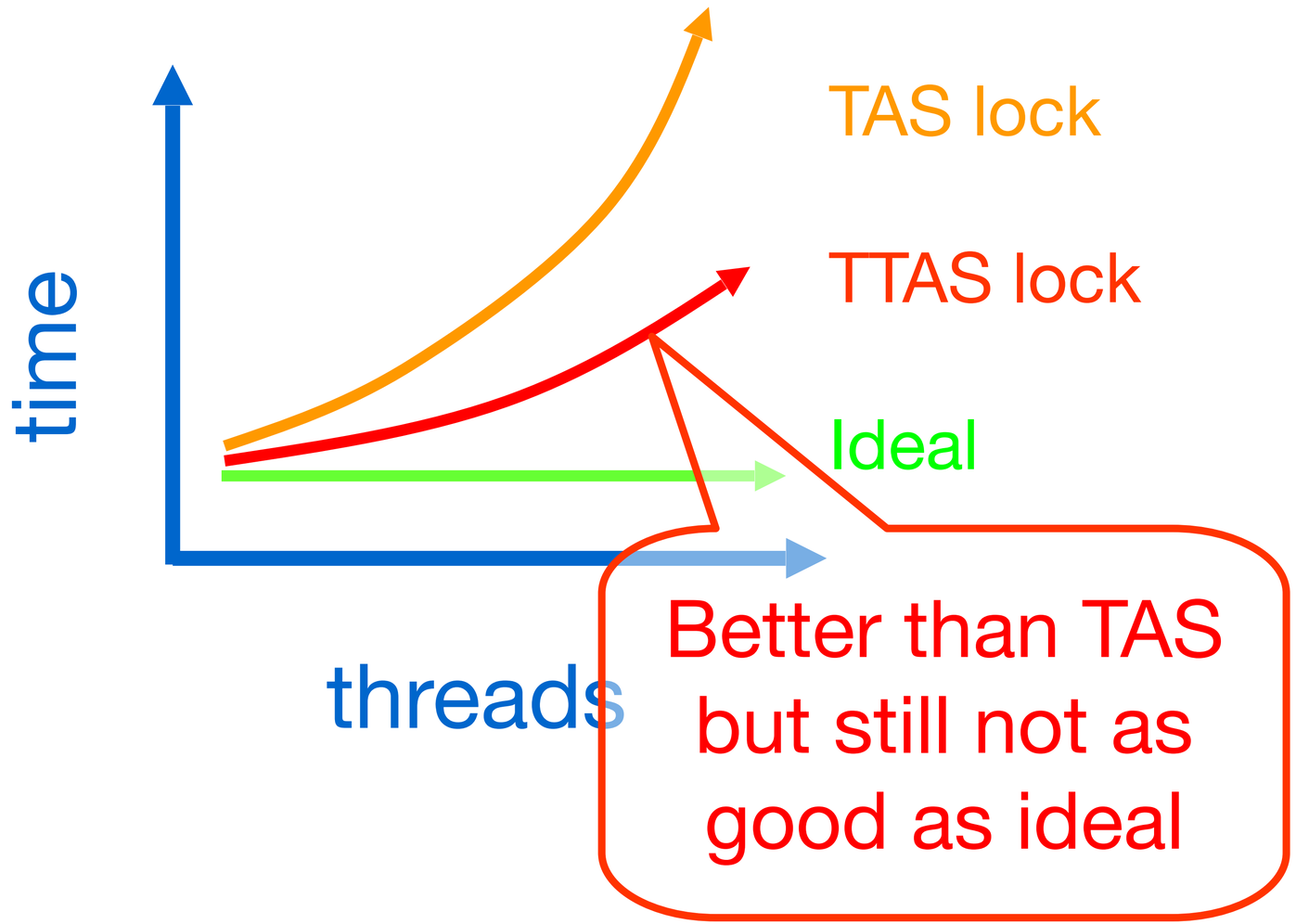
(1)

On Release

Everyone tries TAS



Mystery Explained

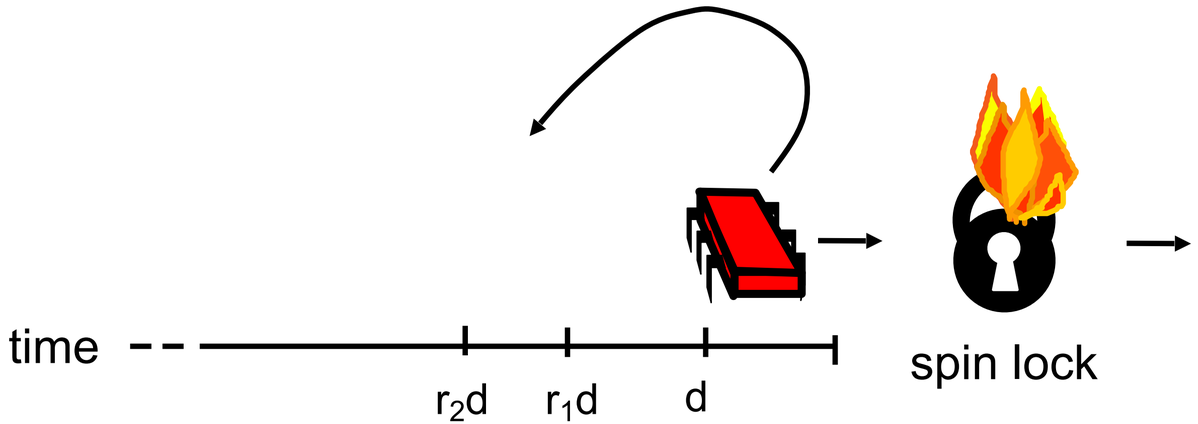


Problems

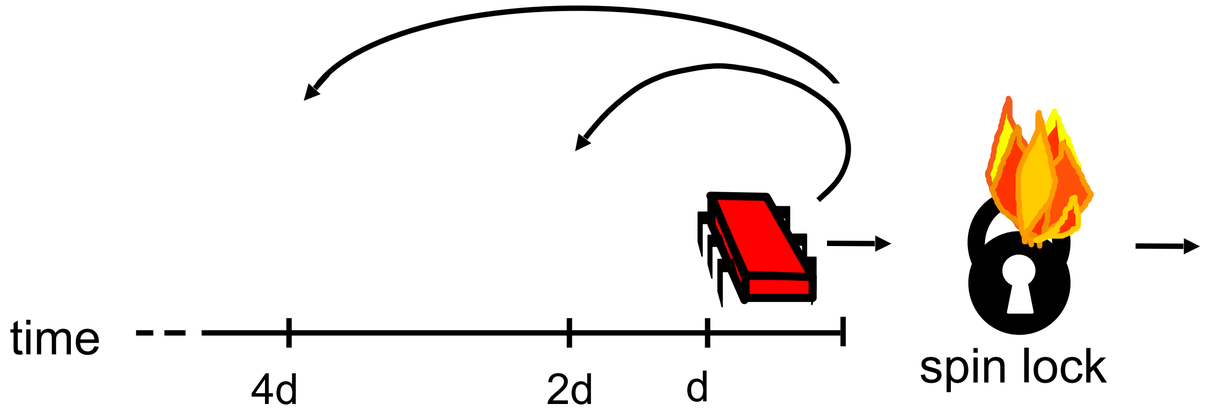
- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?

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 * delay
 }}}
```

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 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;
 }}}
```

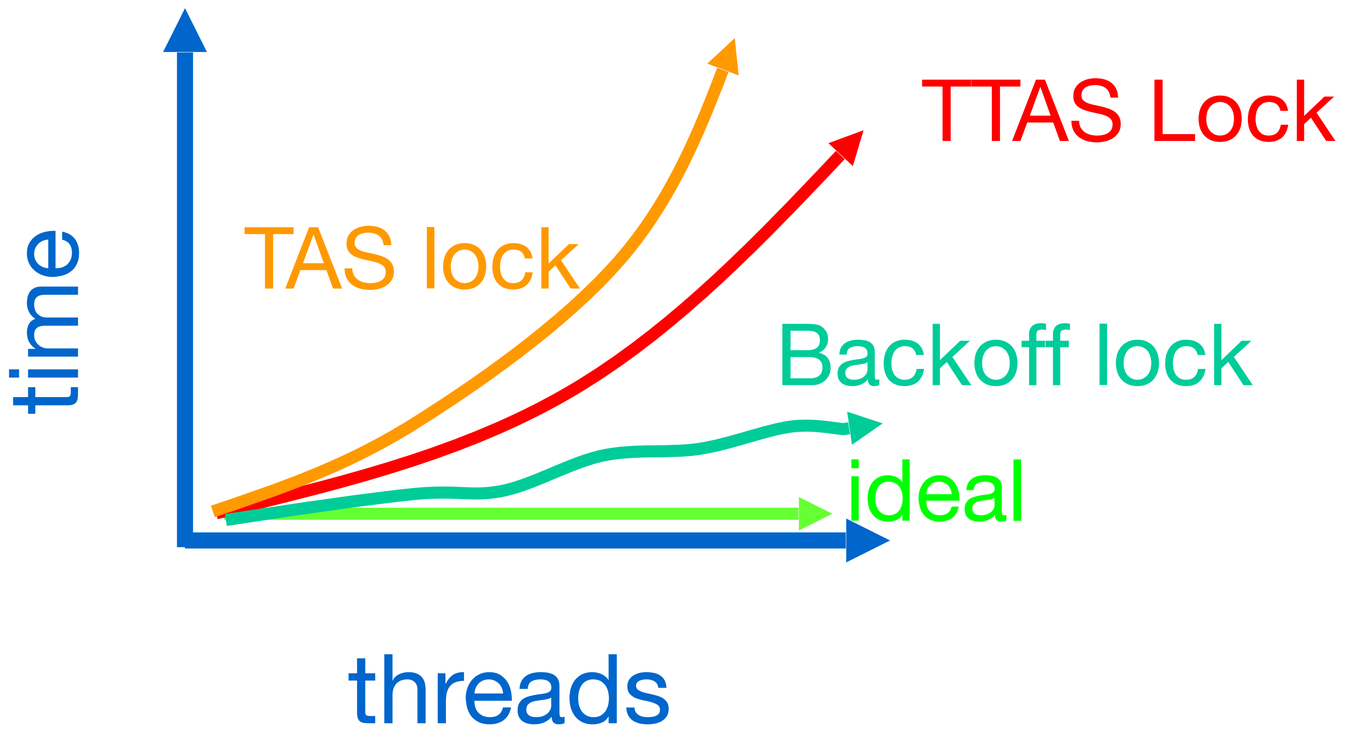
Back off for random duration

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;
 }}}
```

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

Moral of the story?

- EVEN IF we do pretty good with parallelizing most parts our application, we can still see slowdown from **contention** for locks
- For resources infrequently contended, spin locks can be fast because no context switch is necessary
- But, contention in spin locks can have repercussions due to hardware architectures

This work is licensed under a Creative Commons Attribution-ShareAlike license