CMPT 300

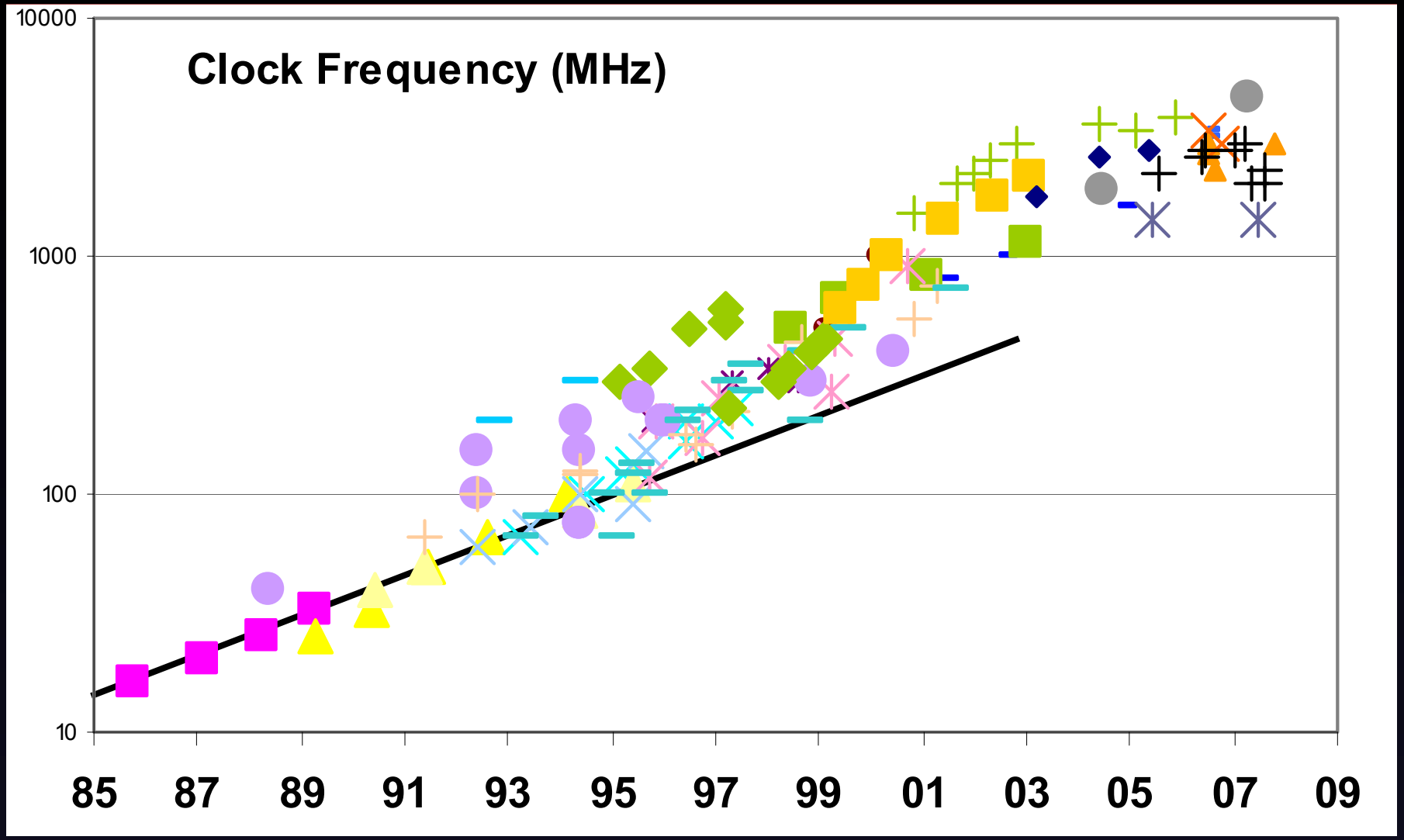
Introduction to Operating Systems

Introduction to Concurrency & Synchronization

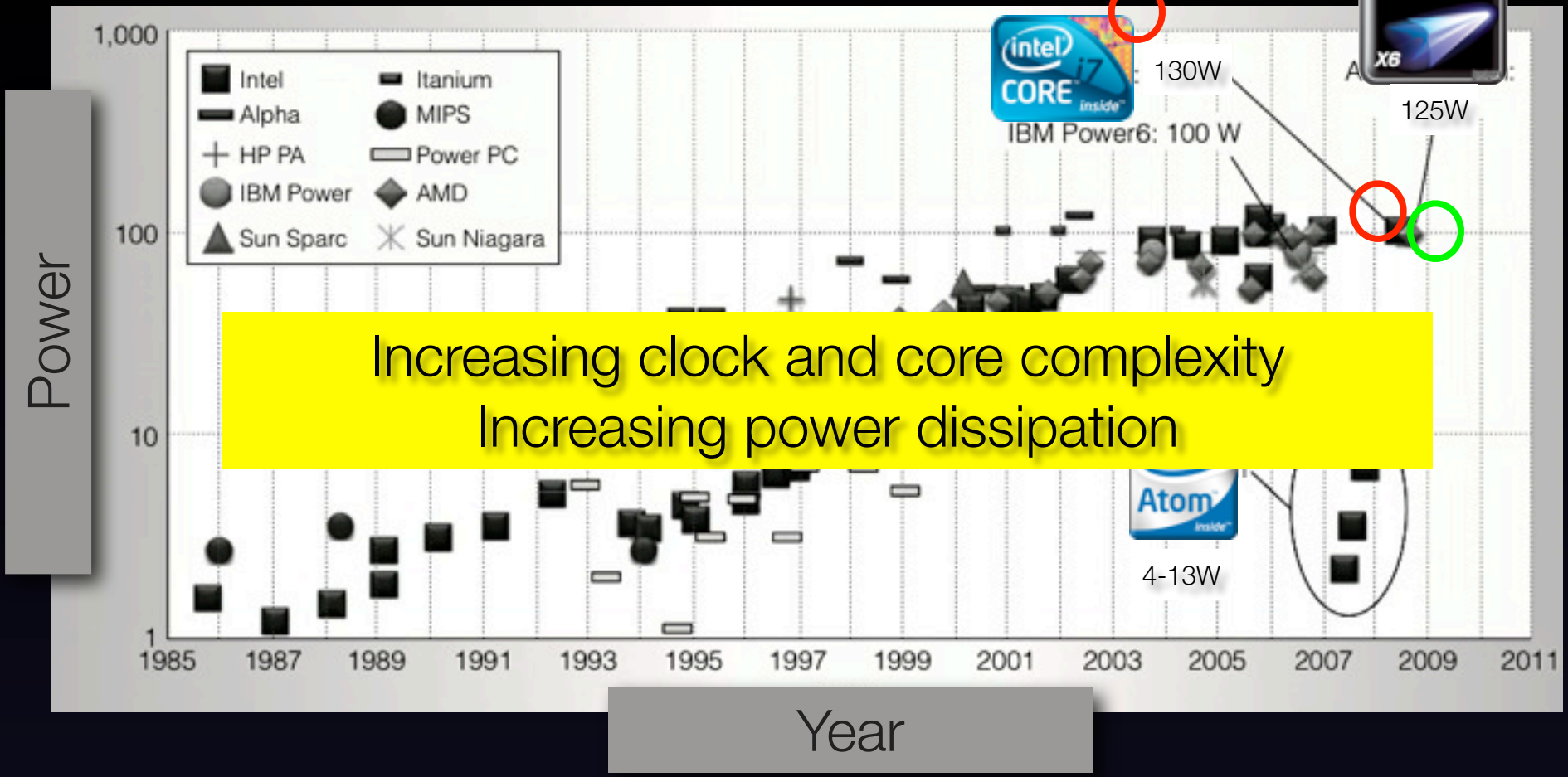
Acknowledgement: some slides are taken from Anthony D. Joseph's course material at UC Berkeley

- PARALLELISM : WHY ?
- ATOMICITY ?
- MUTUAL EXCLUSION ? WHY ? SAFETY LIVENESS
- MILK

Clock Frequency



CPUs consume a lot of power



Source: Rethinking digital design: Why design must change [IEEE Micro]

Development cycle

Multicore Revolution is here!

More cores on a chip

Each core ; 40% ↓ Ghz = 0.25x Power ↓

Overall Performance = 4 cores * 0.6x/core = 2.4x

Example

Multithreaded Programs

- Multithreaded programs must work for all interleavings of threads

- Bugs can be really insidious:
  - Extremely unlikely that this would happen, but may strike at worse possible time

  - Really hard to debug unless carefully designed!

Concurrency Quiz

If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

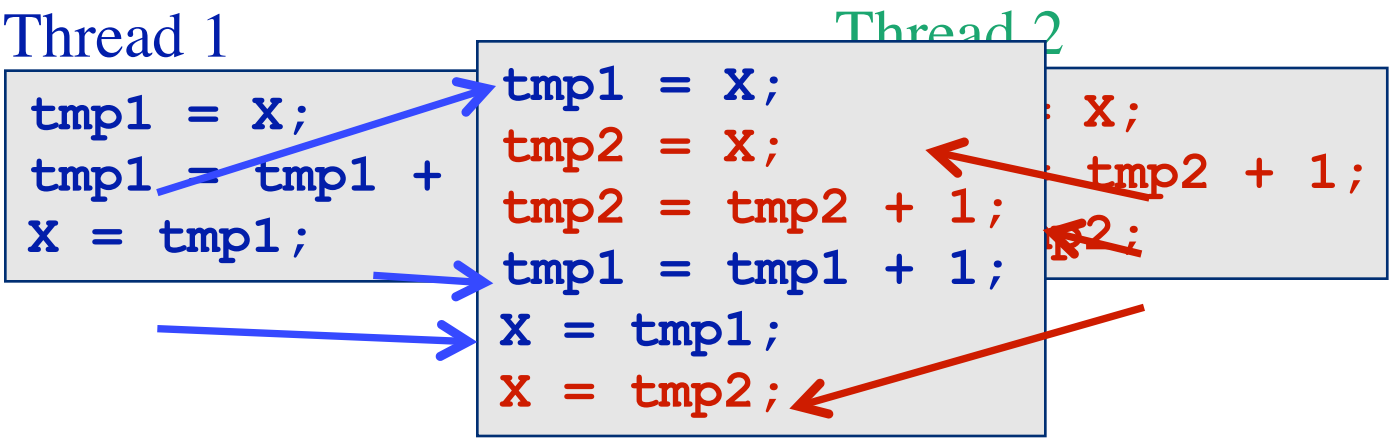
Thread 2

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Answer:
A.0
B.1
C.2
D.More than 2

Schedules/Interleavings

- Model of concurrent execution
- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, some synchronization is needed



If X==0 initially, X == 1 at the end. WRONG result!

Some More Examples

- What are the possible values of x in these cases?

Thread1: x = 1;	Thread2: x = 2;

Initially y = 10;

Thread1: x = y + 1;	Thread2: y = y * 2;

Initially x = 0;

Thread1: x = x + 1;	Thread2: x = x + 2;

Critical Sections

- A critical section is an abstraction
  - Consists of a number of consecutive program instructions
  - Usually, crit sec are mutually exclusive and can wait/signal
    - Later, we will talk about atomicity and isolation
- Critical sections are used frequently in an OS to protect data structures (e.g., queues, shared variables, lists, …)
- A critical section implementation must be:
  - Correct: the system behaves as if only 1 thread can execute in the critical section at any given time
  - Efficient: getting into and out of critical section must be fast. Critical sections should be as short as possible.
  - Concurrency control: a good implementation allows maximum concurrency while preserving correctness
  - Flexible: a good implementation must have as few restrictions as practically possible

The Need For Mutual Exclusion

- Running multiple processes/threads in parallel increases performance
- Some computer resources cannot be accessed by multiple threads at the same time
  - E.g., a printer can't print two documents at once
- Mutual exclusion is the term to indicate that some resource can only be used by one thread at a time
  - Active thread excludes its peers
- For shared memory architectures, data structures are often mutually exclusive
  - Two threads adding to a linked list can corrupt the list

Exclusion Problems, Real Life Example

- Imagine multiple chefs in the same kitchen
  - Each chef follows a different recipe
- Chef 1
  - Grab butter, grab salt, do other stuff
- Chef 2
  - Grab salt, grab butter, do other stuff
- What if Chef 1 grabs the butter and Chef 2 grabs the salt?
  - Yell at each other (not a computer science solution)
  - Chef 1 grabs salt from Chef 2 (preempt resource)
  - Chefs all grab ingredients in the same order
    - Current best solution, but difficult as recipes get complex
    - Ingredient like cheese might be sans refrigeration for a while

The Need To Wait

- Very often, synchronization consists of one thread waiting for another to make a condition true
  - Master tells worker a request has arrived
  - Cleaning thread waits until all lanes are colored
- Until condition is true, thread can sleep
  - Ties synchronization to scheduling
- Mutual exclusion for data structure
  - Code can wait (await)
  - Another thread signals (notify)

Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Definitions

- Synchronization: using atomic operations to coordinate multiple concurrent threads that are using shared state

- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - *excludes* the other while doing its work

- Critical Section: piece of code that only one thread can execute at once. Only one thread gets in.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

Motivation: “Too much milk”

- Consider two roommates who need to coordinate to get milk if out of milk:

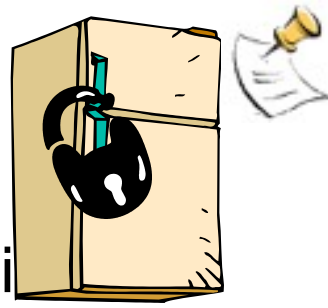
TOO MUCH MILK!

T		
3		
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

More Definitions

- Lock: prevents someone from doing something
  - Lock before entering critical section
  - Unlock when leaving,
  - Wait if locked
    - Important idea: all synchronization involves waiting

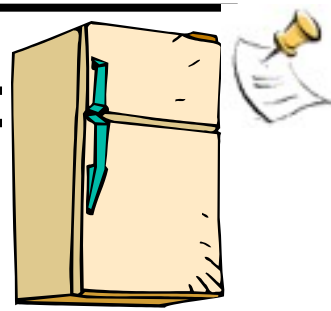
- Example: Lock on the refrigerator
  - Lock it and take key if you are going to go buy mi
  - Too coarse-grained: refrigerator is unavailable
    - Roommate gets angry if he only wants OJ



Too Much Milk: Correctness Properties

- Correctness for "Too much milk" problem
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load (read) and store (write) operations
- Concurrent programs are non-deterministic due to many possible interleavings

Too Much Milk: Solution #1



- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)

```
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove note;
    }
}
```

Context-switch point

- Result?
  - Still too much milk but only occasionally!

Too Much Milk: Solution #1½

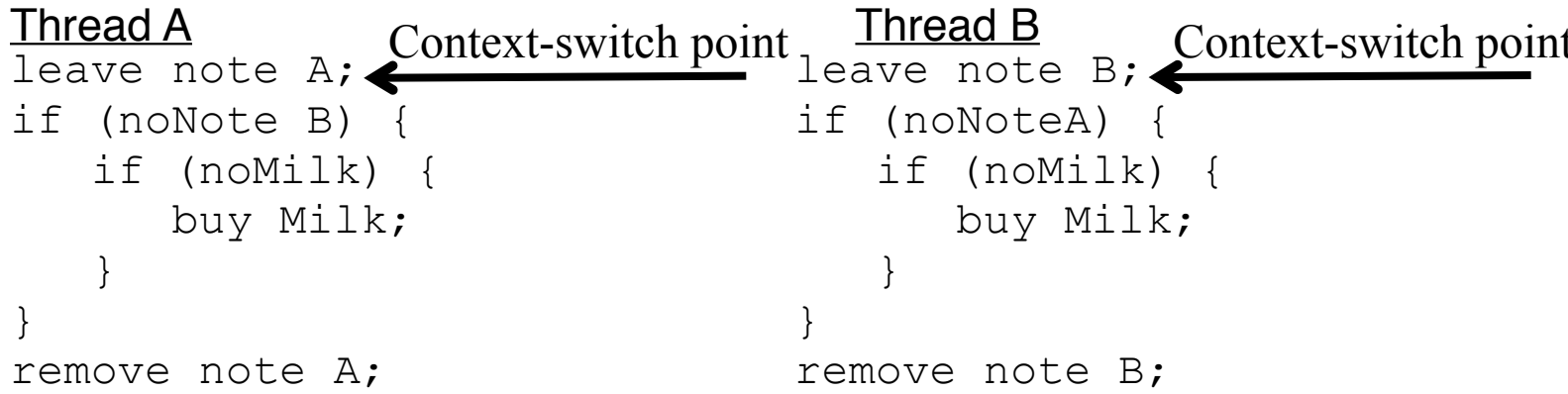
- Another try:

```
leave Note;
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
    }
}
remove note;
```

- What happens here?
  - "`leave Note; buy milk;`" will never run.
  - No one ever buys milk!

To Much Milk Solution #2

- How about labeled notes?



- Does this work? Still no
- Possible for neither thread to buy milk
  - Thread A leaves note A; Thread B leaves note B; each sees the other's note, thinking "*I'm* not getting milk, *You're* getting milk"
  - Each one thinks that the other is getting it.

Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) {  //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

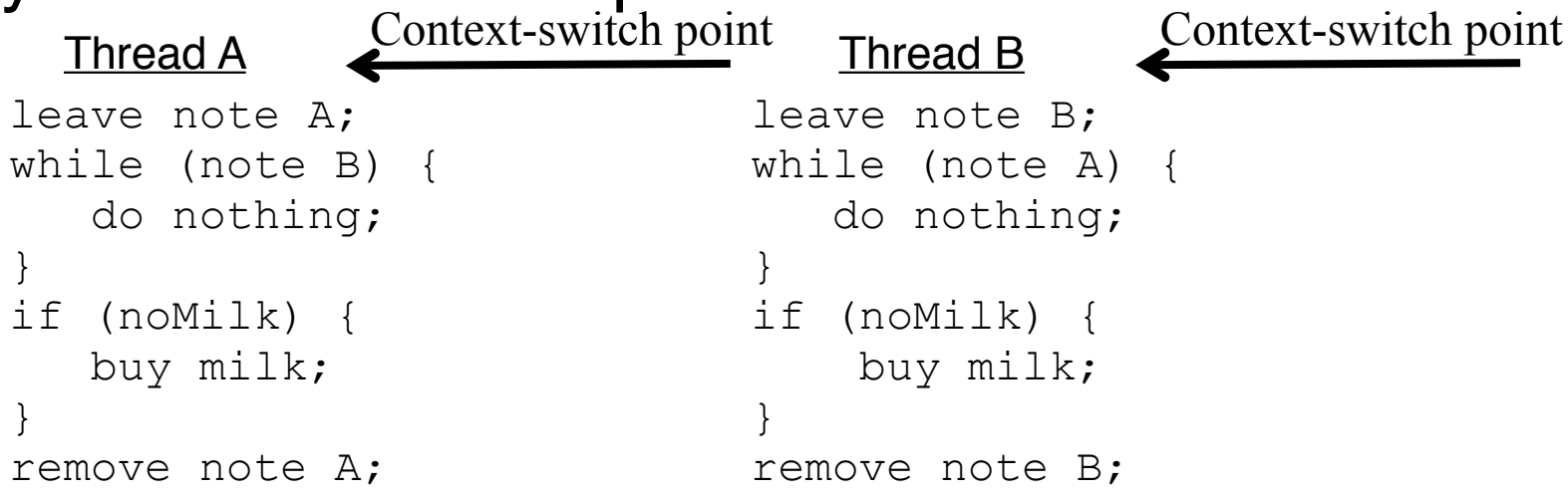
Thread B

```
leave note B;
if (noNote A) {  //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes.
- It is safe to buy, or Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
- At Y:
  - if no note A, safe for B to buy

Solution 3.5

- Note that the solution is asymmetric!
  - Quzz: does it work if Thread B also has a symmetric while loop?



No. Each thread can leave a note, then go into infinite while loop.

Solution #3 Discussions

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread

- There's a better way
  - Have HW provide better (higher-level) primitives
  - Build even higher-level programming abstractions on this new hardware support

Too Much Milk: Solution #4

- We need to protect a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Solution:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

The correctness conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region
- Bounded waiting
  - A thread that enters the entry section enters the critical section within some bounded number of operations.
- Failure atomicity
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity

```
while(1) {
Acquire (Lock)
    Critical section
    Exit section
Release (Lock)
}
```

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Therac-25 Example

- Example: Therac-25
  - Machine for radiation therapy
    - Software control of electron accelerator and electron beam/Xray production
    - Software control of dosage

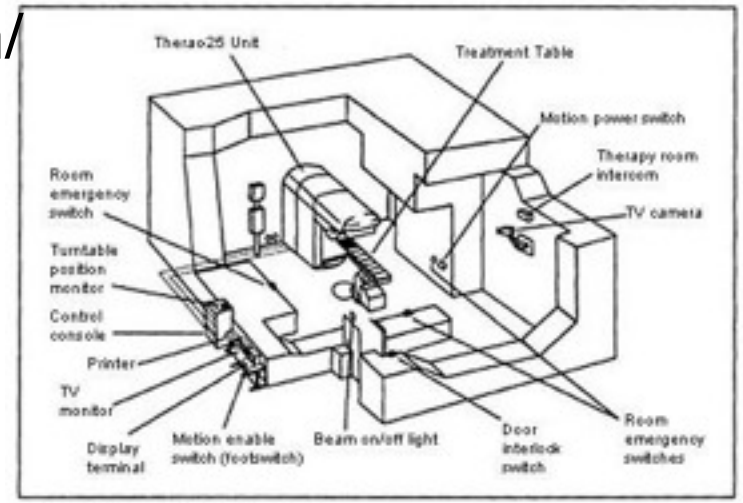
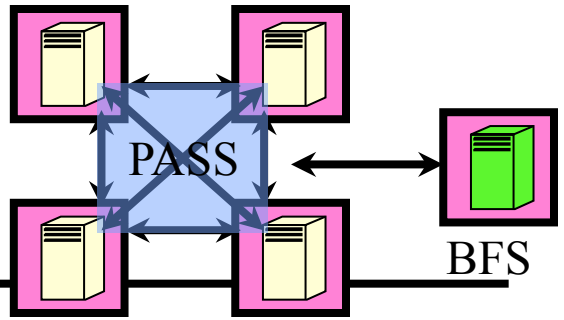


Figure 1. Typical Therac-25 facility

  - Software errors caused the death of several patients
    - A series of race conditions on shared variables and poor software design

Space Shuttle



- Launch aborted 20 minutes before T minus 0.
- Shuttle has five computers:
  - Four run the “Primary Avionics Software System” (PASS)
    - Asynchronous and real-time
    - Runs all of the control systems
    - Results synchronized and compared every 3 to 4 ms
  - The Fifth computer is the “Backup Flight System” (BFS)
    - stays synchronized in case it is needed
    - Written by completely different team than PASS
- Countdown aborted; BFS disagreed with PASS
  - A 1/67 chance that PASS was out of sync one cycle
  - Bug due to modifications in initialization code of PASS

Summary

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available

- Shared data introduces challenges.
  - Programs must be properly synchronized
  - Without careful design, shared variables can become completely inconsistent

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - Construct various synchronization primitives

High-Level Picture

- Implementing a concurrent program with only loads and stores would be tricky and error-prone
  - Consider “too much milk” example
  - Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!

- We’ll implement higher-level operations on top of atomic operations provided by HW
  - Develop a “synchronization toolbox”
  - Explore some common programming paradigms

How to implement Locks?

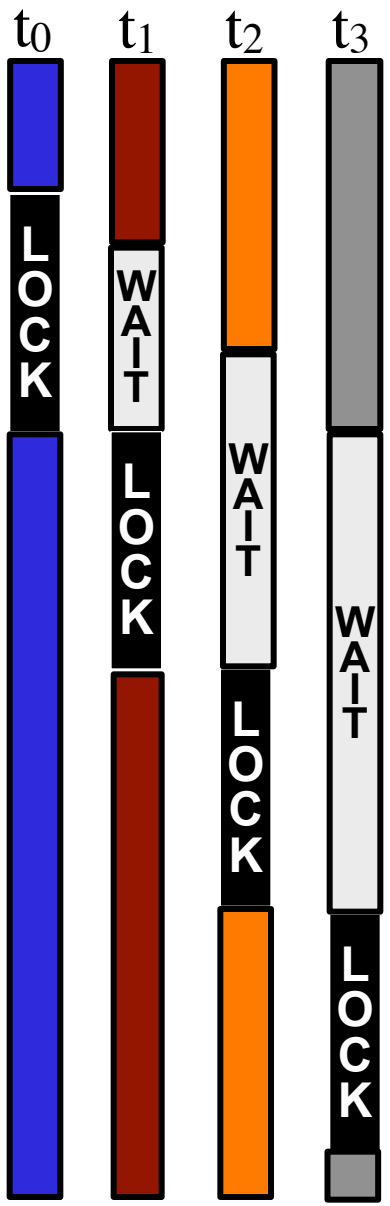
How to implement Locks?

- Lock: prevents someone from doing something
  - Lock before entering critical section
  - Unlock when leaving, after accessing shared data
  - Wait if locked

- Hardware Lock instruction
  - Is this a good idea?
  - Complexity?
    - Done in the Intel 432
    - Each feature makes hardware more complex and slow
  - What about putting a task to sleep?
    - How do you handle the interface between the hardware and scheduler?

Lock-Based Mutual Exclusion

- Only one thread can hold a "lock" at a time
  - Used a provide serialized access to a data object
- If another threads tries to acquire a held lock
  - Must wait until other thread performs a release
- Performance implications
  - Lock contention limits parallelism
  - Lock acquire/release time adds overheads
- Correctness implications
  - Just one example:
    - Thread #1: Holds lock A, tries to acquire B
    - Thread #2: Holds lock B, tries to acquire A
    - Classic deadlock!



Read-Modify-Write (RMW)

- Implement locks using read-modify-write instructions
  - As an atomic and isolated action
    1. read a memory location into a register, **AND**
    2. write a new value to the location
  - Implementing RMW is tricky in multi-processors
    - Requires cache coherence hardware. Caches snoop the memory bus.

- Examples:
  - Test&set instructions (most architectures)
    - Reads a value from memory
    - Write "1" back to memory location
  - Compare & swap (68000)
    - Test the value against some constant
    - If the test returns true, set value in memory to different value
    - Report the result of the test in a flag
    - if [addr] == r1 then [addr] = r2;
  - Exchange, locked increment, locked decrement (x86)
  - Load linked/store conditional (PowerPC,Alpha, MIPS)

Simple Boolean Spin Locks

- Simplest lock:
  - Single variable, two states: **locked**, **unlocked**
  - When unlocked: atomically transition from unlocked to locked
  - When locked: keep checking (spin) until the lock is unlocked
- Busy waiting versus "blocking"
  - In a multicore, **busy wait** for other thread to release lock
    - Likely to happen soon, assuming critical sections are small
    - Likely nothing "better" for the processor to do anyway
  - In a single processor, if trying to acquire a held lock, **block**
    - The only sensible option is to tell the O.S. to context switch
    - O.S. knows not to reschedule thread until lock is released
  - Blocking has high overhead (O.S. call)
    - IMHO, rarely makes sense in multicore (parallel) programs

Naïve use of Interrupt Enable/ Disable

- How can we build multi-instruction atomic ops?
  - OS dispatcher gets control in two ways.
    - Internal: Thread does something to relinquish the CPU
    - External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - Avoiding internal events ; preventing external events
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

- Challenges:
  - Can't let user do this! Consider following:

```
LockAcquire();
While(TRUE) {;}
```

  - Real-Time system—no guarantees on timing!
    - Critical Sections might be arbitrarily long

Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable
- Waiting thread goes to sleep

```
int value = FREE;
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```

New Lock Implementation: Discussion

- Why do we need to disable interrupts?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
disable interrupts;
if (value == BUSY) {
   put thread on wait queue;
   Go to sleep();
   // Enable interrupts?
} else {
   value = BUSY;
}
enable interrupts;
}
```

Critical Section

- Note: unlike previous solution, the critical section (inside `Acquire()`) is very short

Interrupt re-enable in going to sleep

Enable Position?
Enable Position?
Enable Position?

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

- Before Putting thread on the wait queue?
  - Release can check the queue and not wake up thread
- After putting the thread on the wait queue
  - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  - Misses wakeup and still holds lock (deadlock!)

Hardware Mechanism

- Problem with previous solution:
  - Relies on programmer discipline for correctness

- CPUs generally have hardware mechanisms to support this requirement.
  - For example, on the Atmega128 microcontroller, the `sei` instruction does not re-enable interrupts until two cycles after it is issued (so the instruction sequence `sei sleep` runs atomically).

Locks

Atomic Instruction Sequences

- Problem with disabling interrupts
  - Can be dangerous: interrupts should not be disabled for a long time, otherwise may miss important interrupts
  - Doesn't work well on multiprocessor
    - Disabling interrupts on all processors requires messages and would be very time consuming

- Alternative: atomic read-modify-write instruction sequences supported by hardware
  - These instructions read a value from memory and write a new value atomically
  - Can be used on both uniprocessors and multiprocessors

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

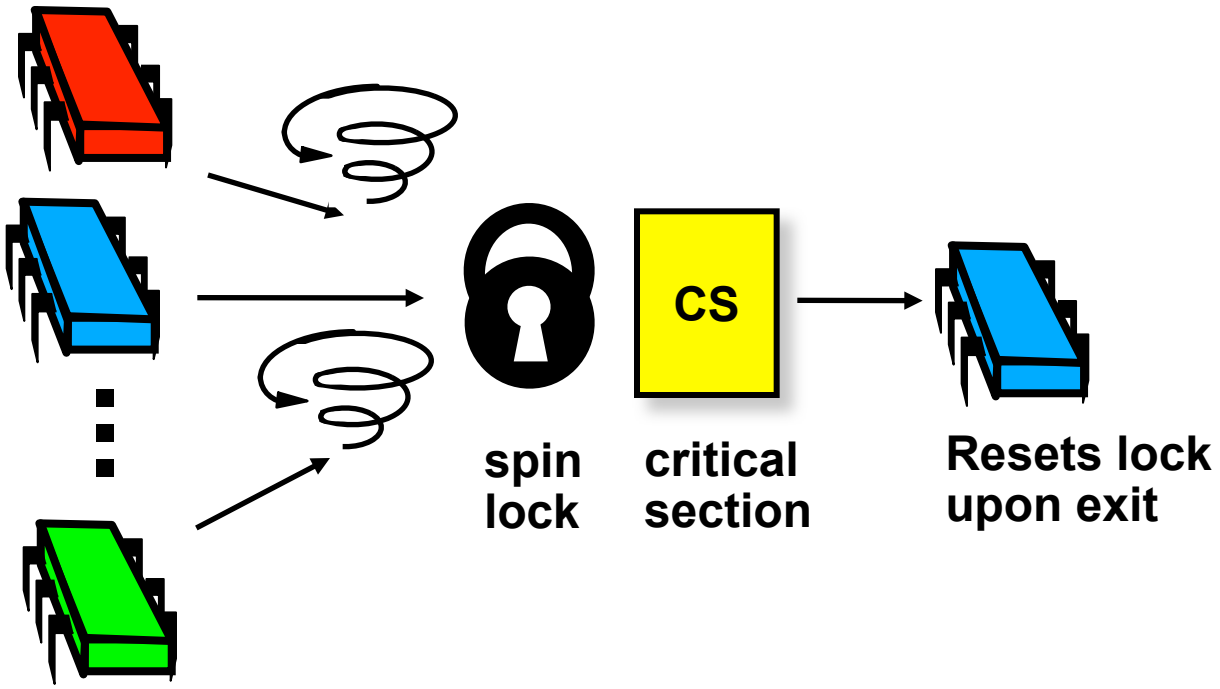
```
Lock::Acquire() {
while (test&set(lock) == 1)
   ; //spin
}
```

- If lock is free (lock_value == 0), then test&set reads 0 and sets value to 1 ➔ lock is set to busy and Acquire completes
- If lock is busy, the test&set reads 1 and sets value to 1 ➔ no change in lock's status and Acquire loops

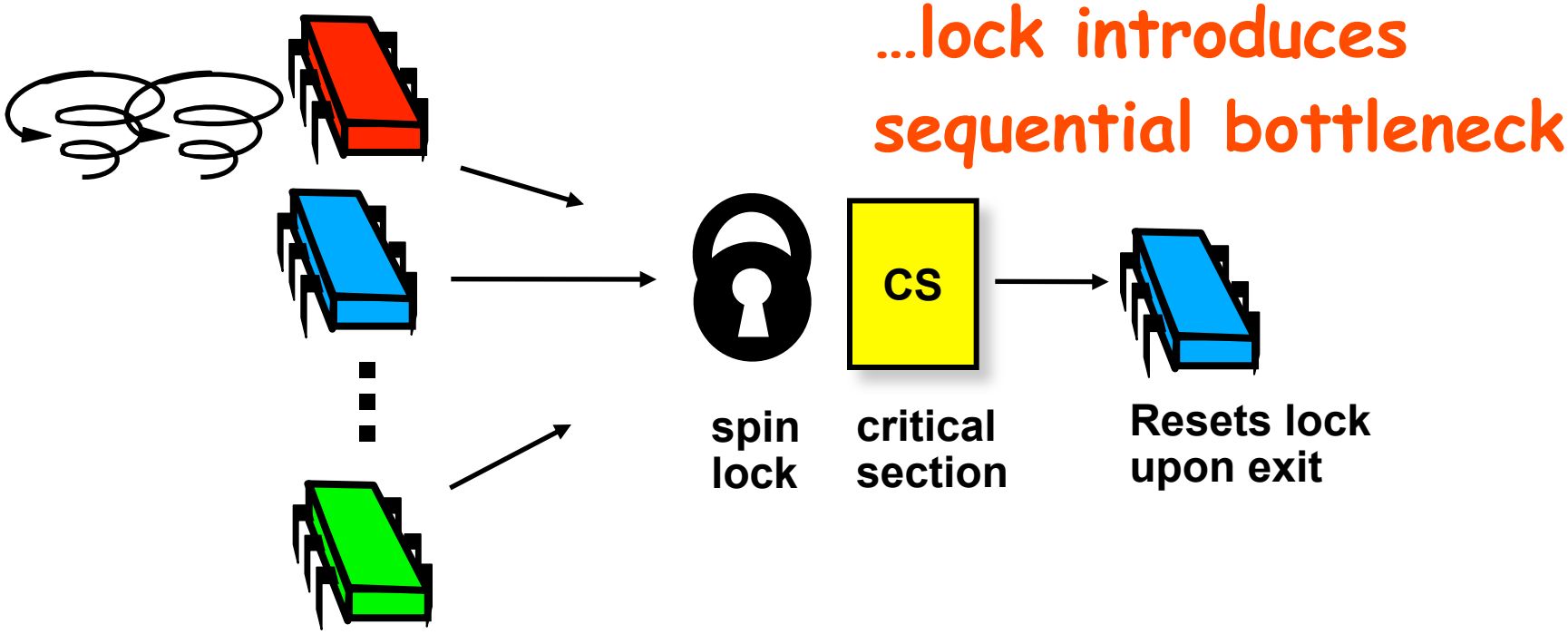
```
Lock::Release() {
    *lock = 0;
}
```

- Does this lock have bounded waiting?

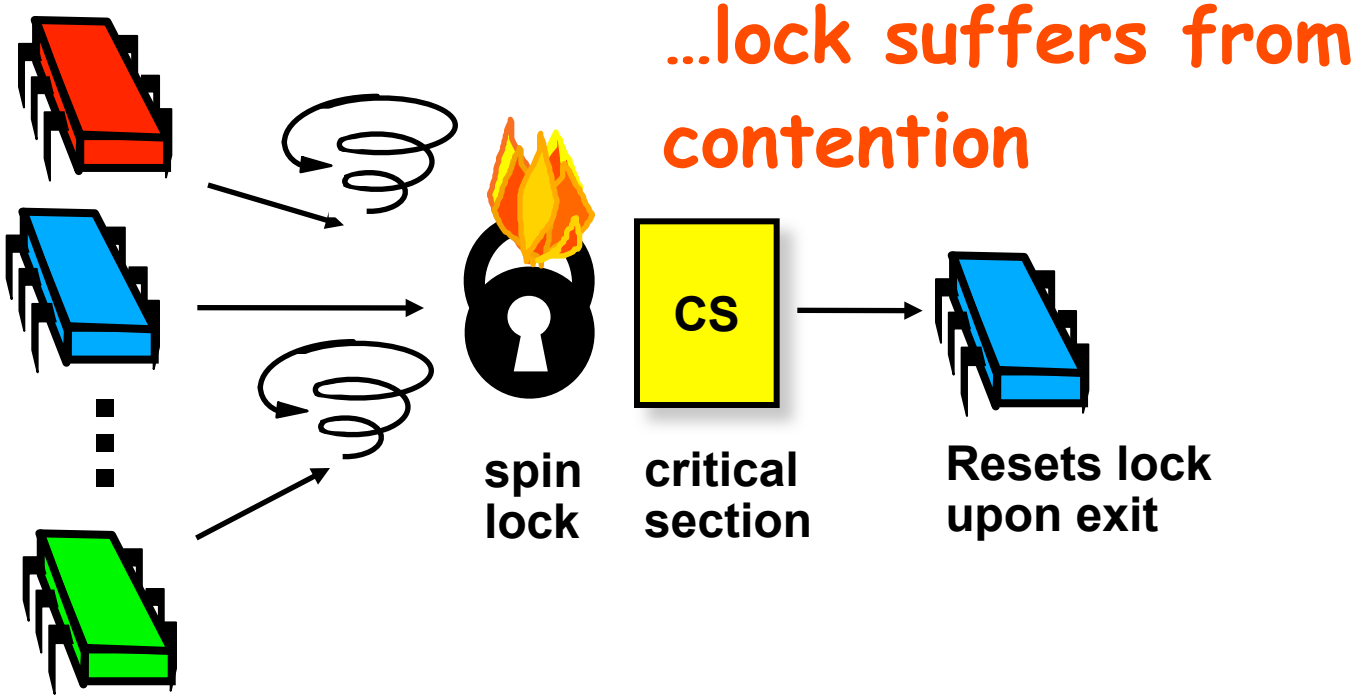
Basic Spin-Lock



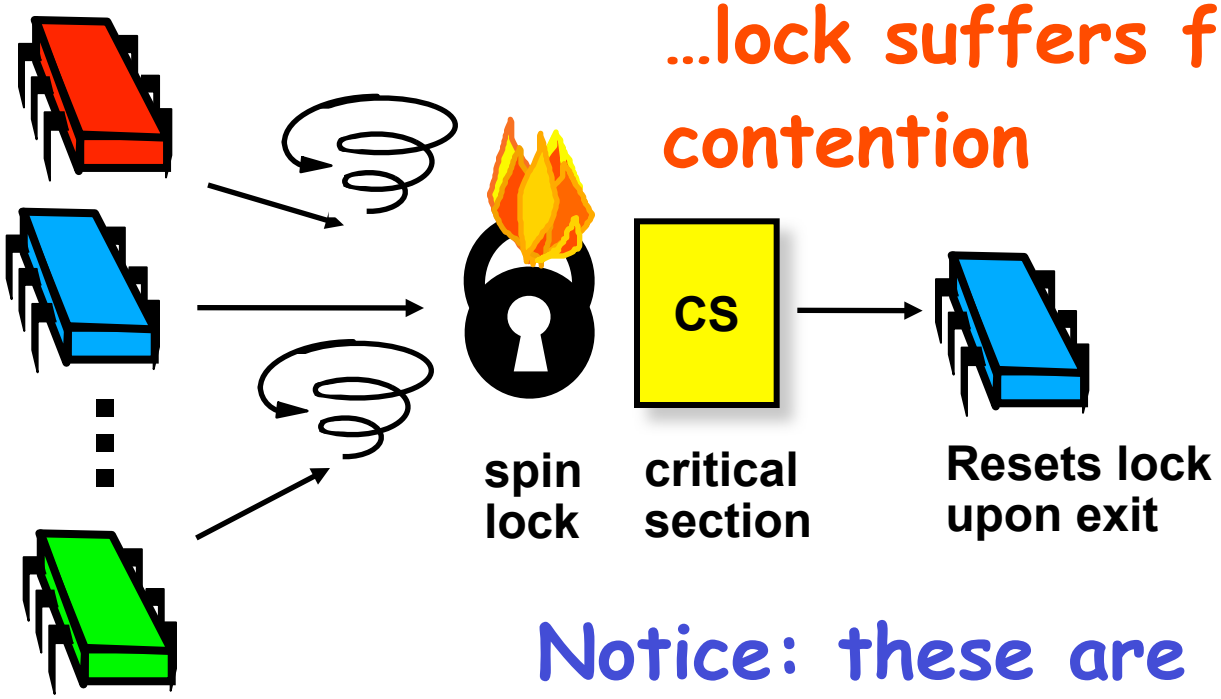
Basic Spin-Lock



Basic Spin-Lock

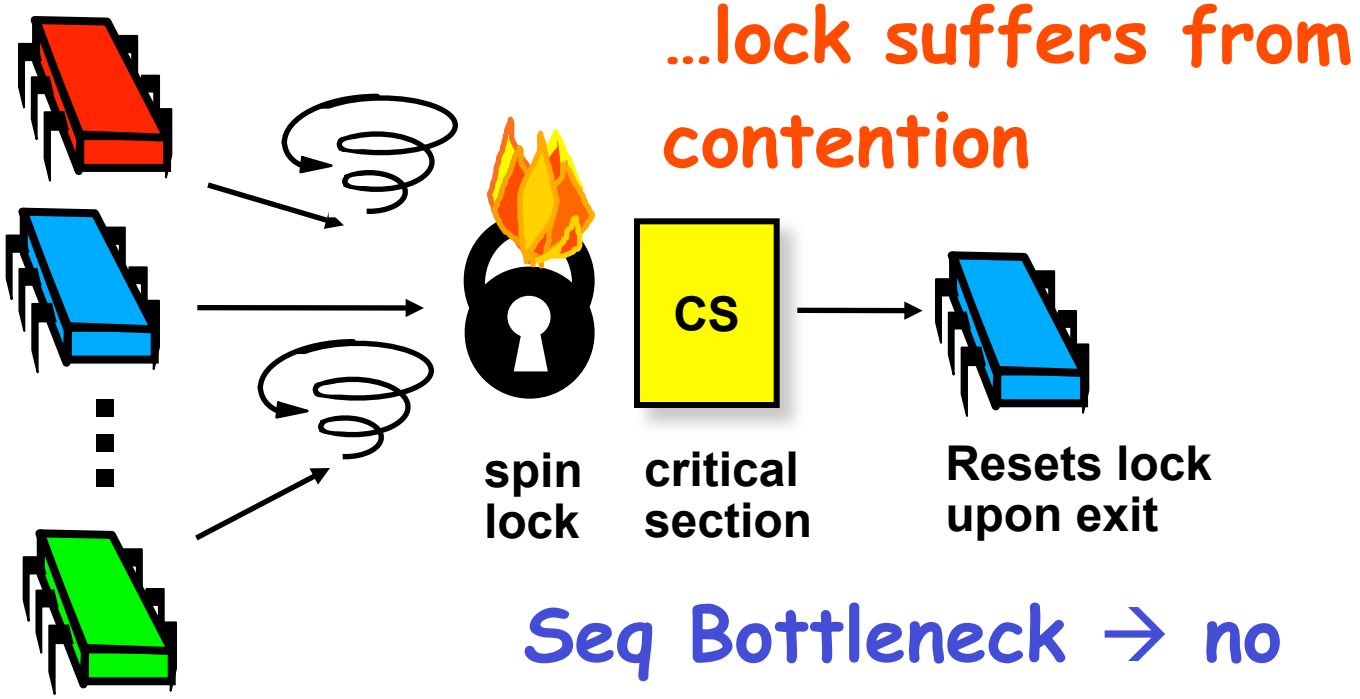


Basic Spin-Lock



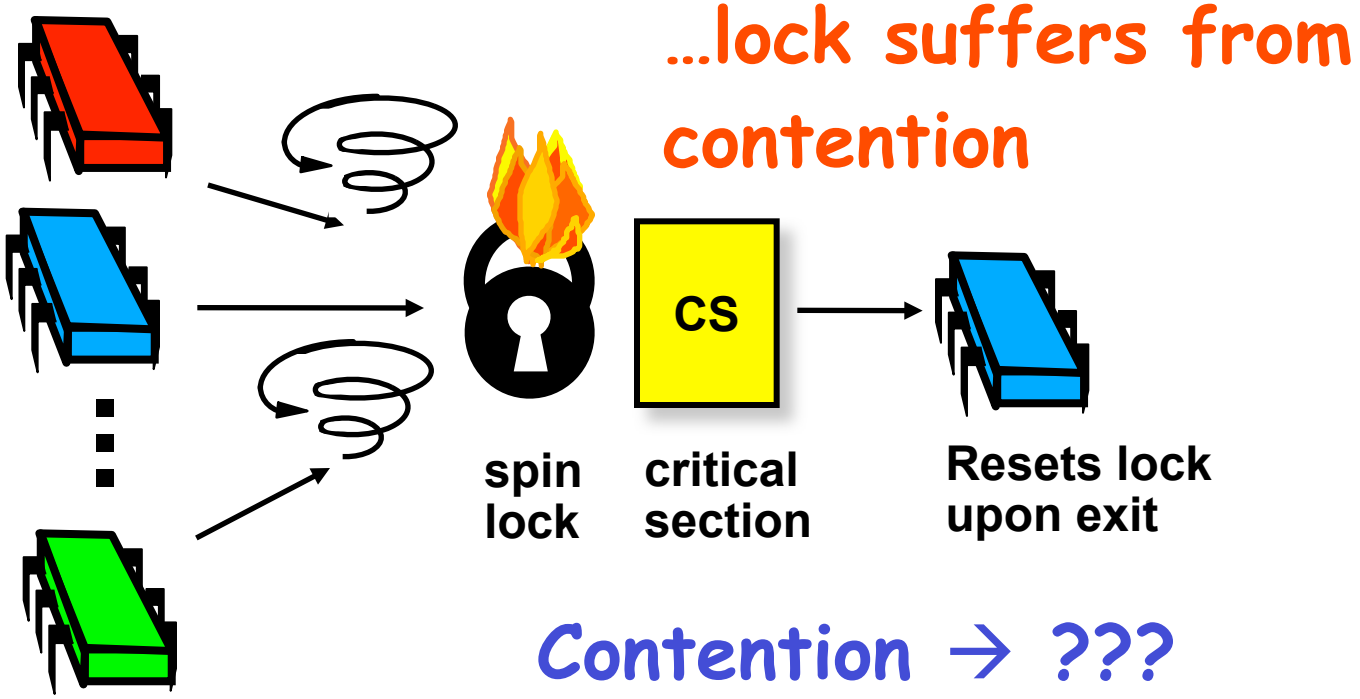
Notice: these are distinct phenomena

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.set(false);
 }}
```

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(false);
 }}
```

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

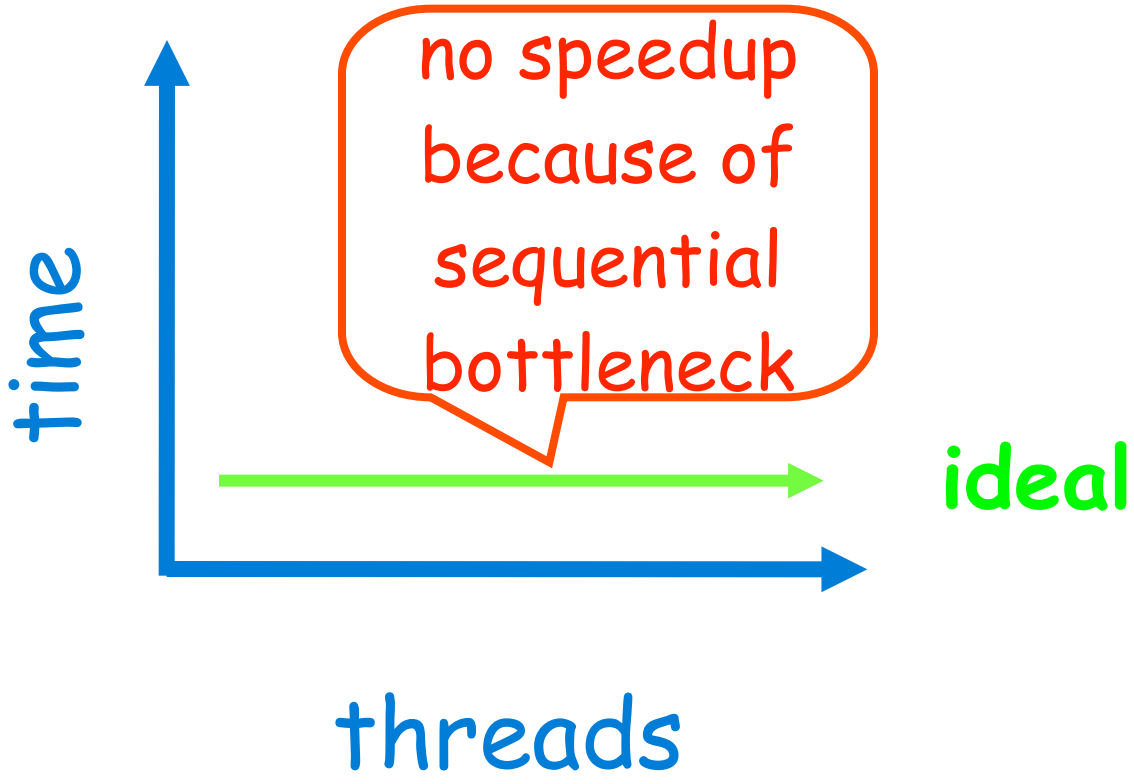
Space Complexity

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

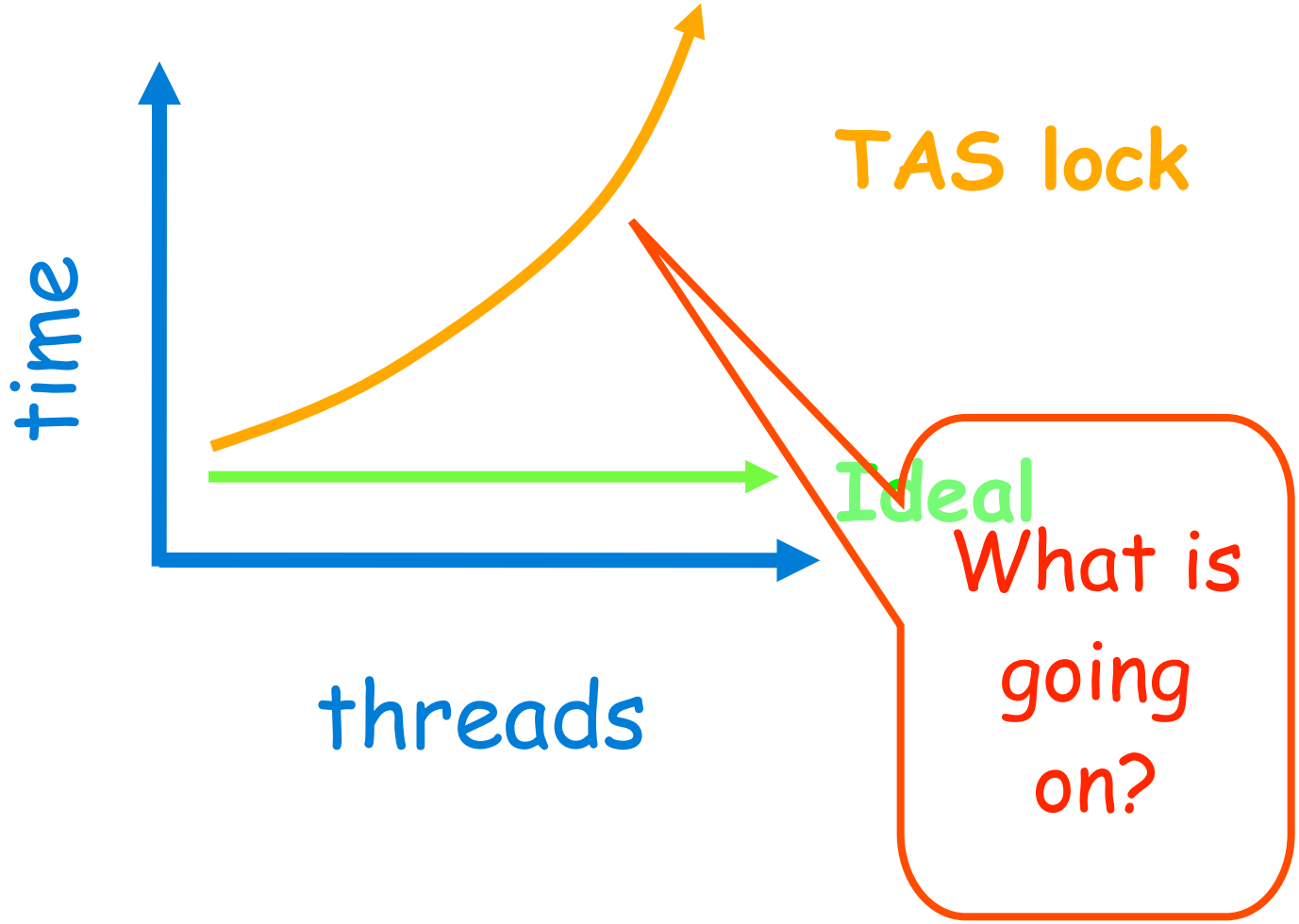
Performance

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

Graph


no speedup
because of
sequential
bottleneck
time
ideal
threads

Mystery #1



Test-and-Test-and-Set Locks

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

Test-and-test-and-set Lock

```
class TTASlock {
 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

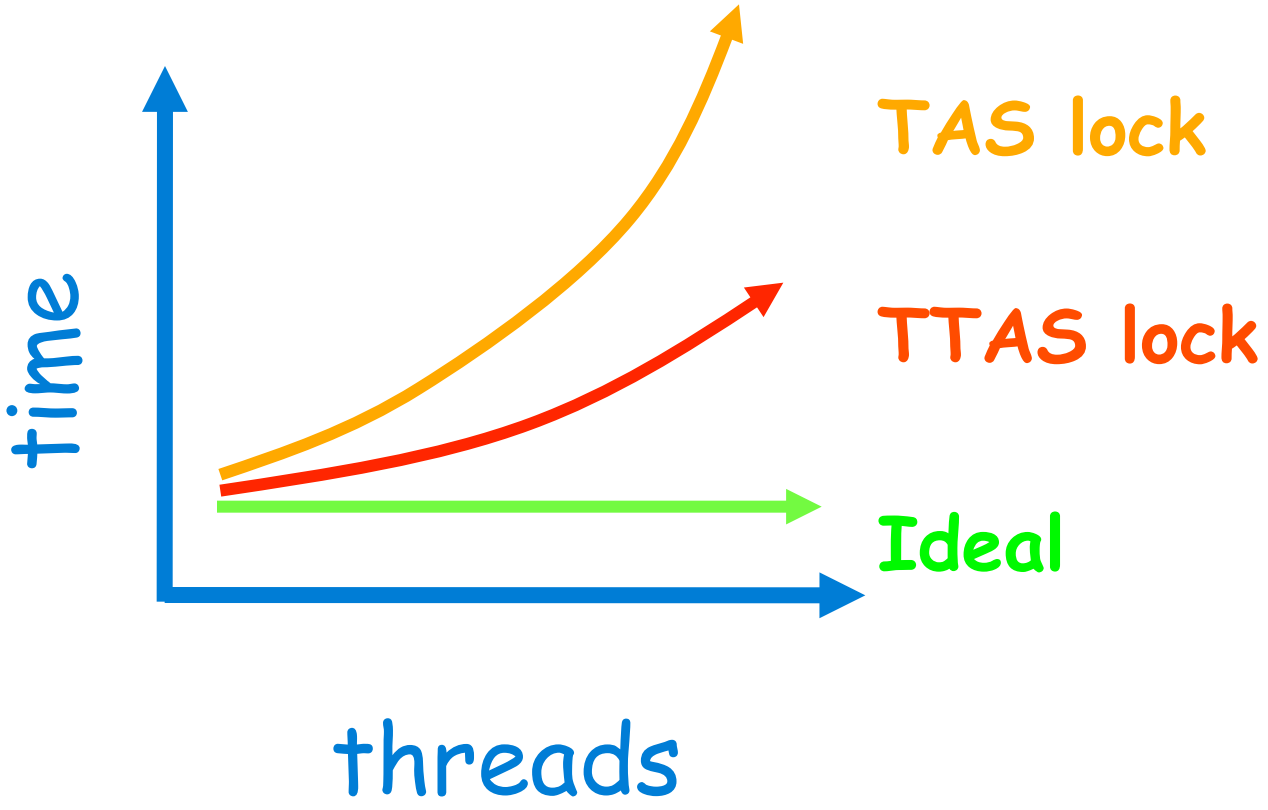
Test-and-test-and-set Lock

```
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



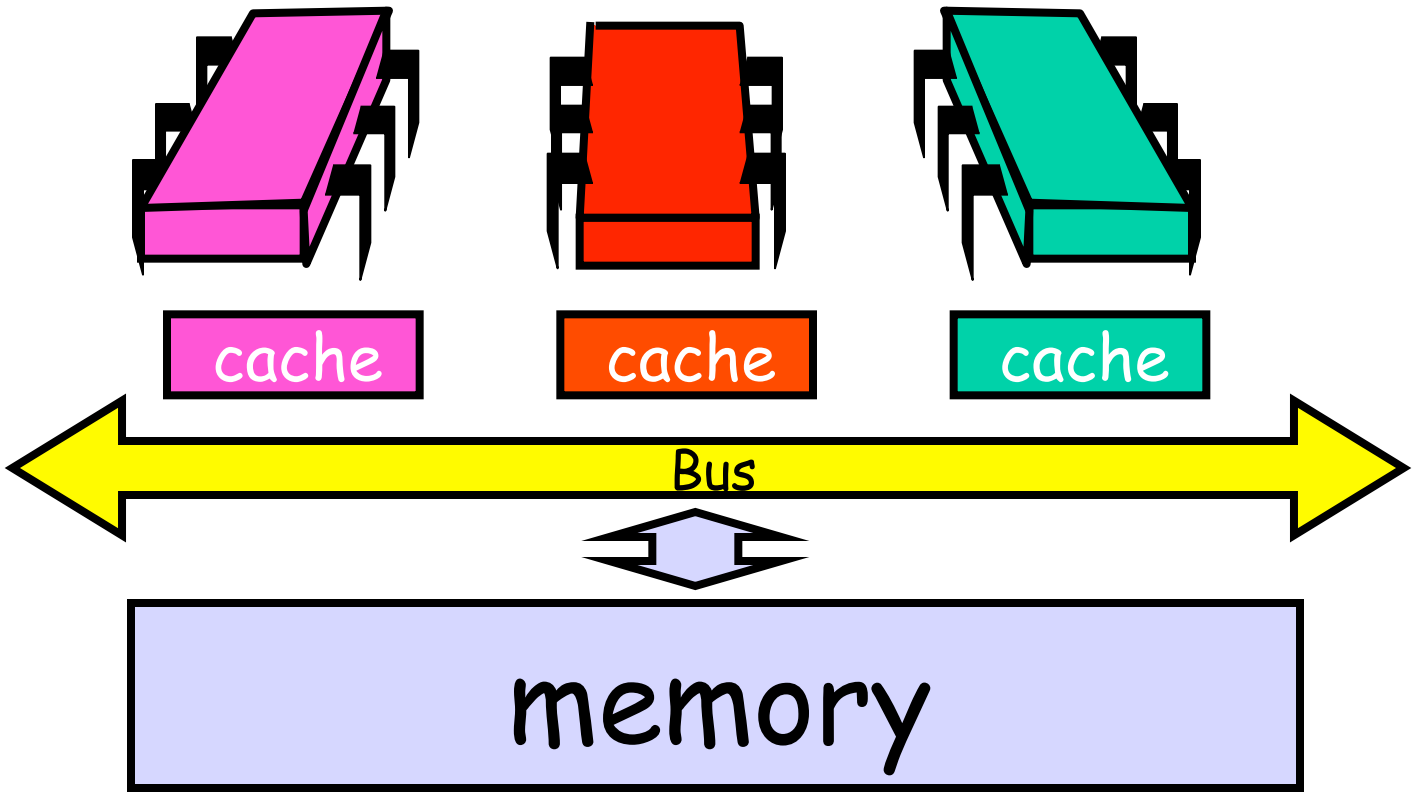
Mystery

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

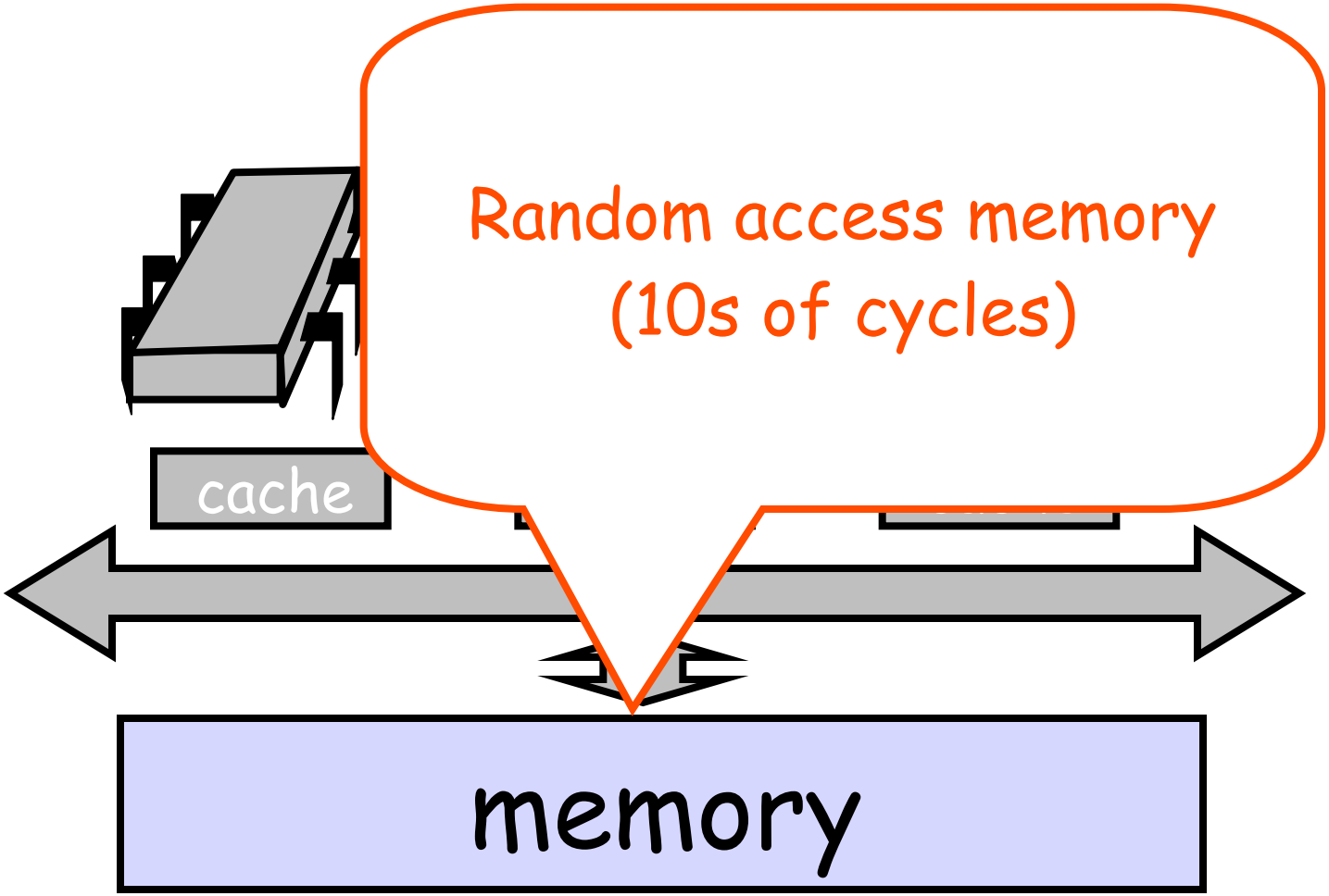
Opinion

- 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



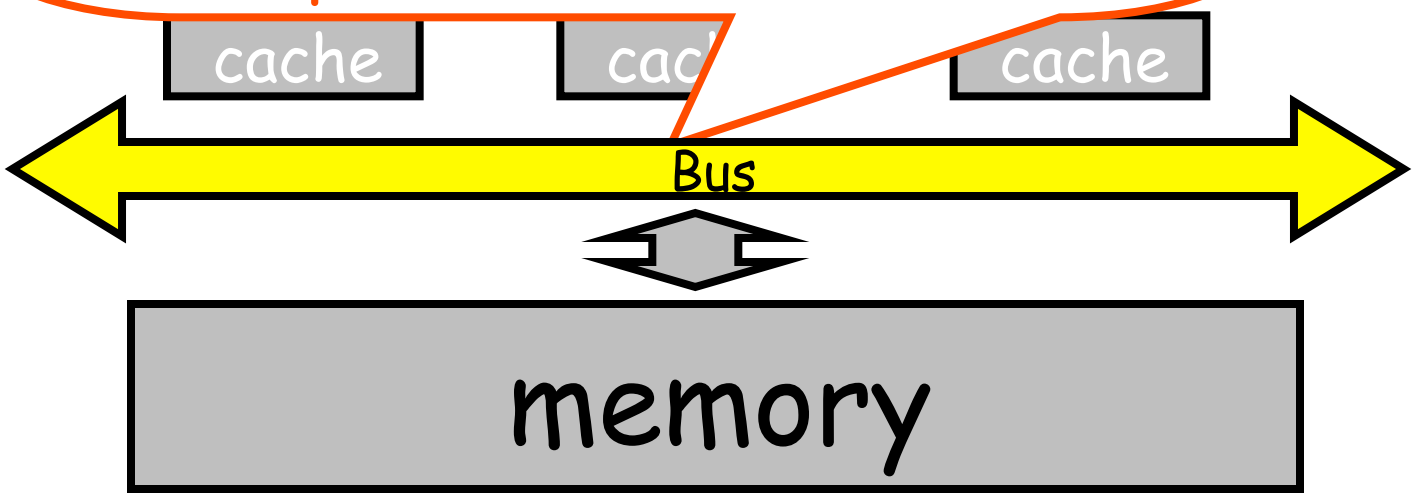
Bus-Based Architectures



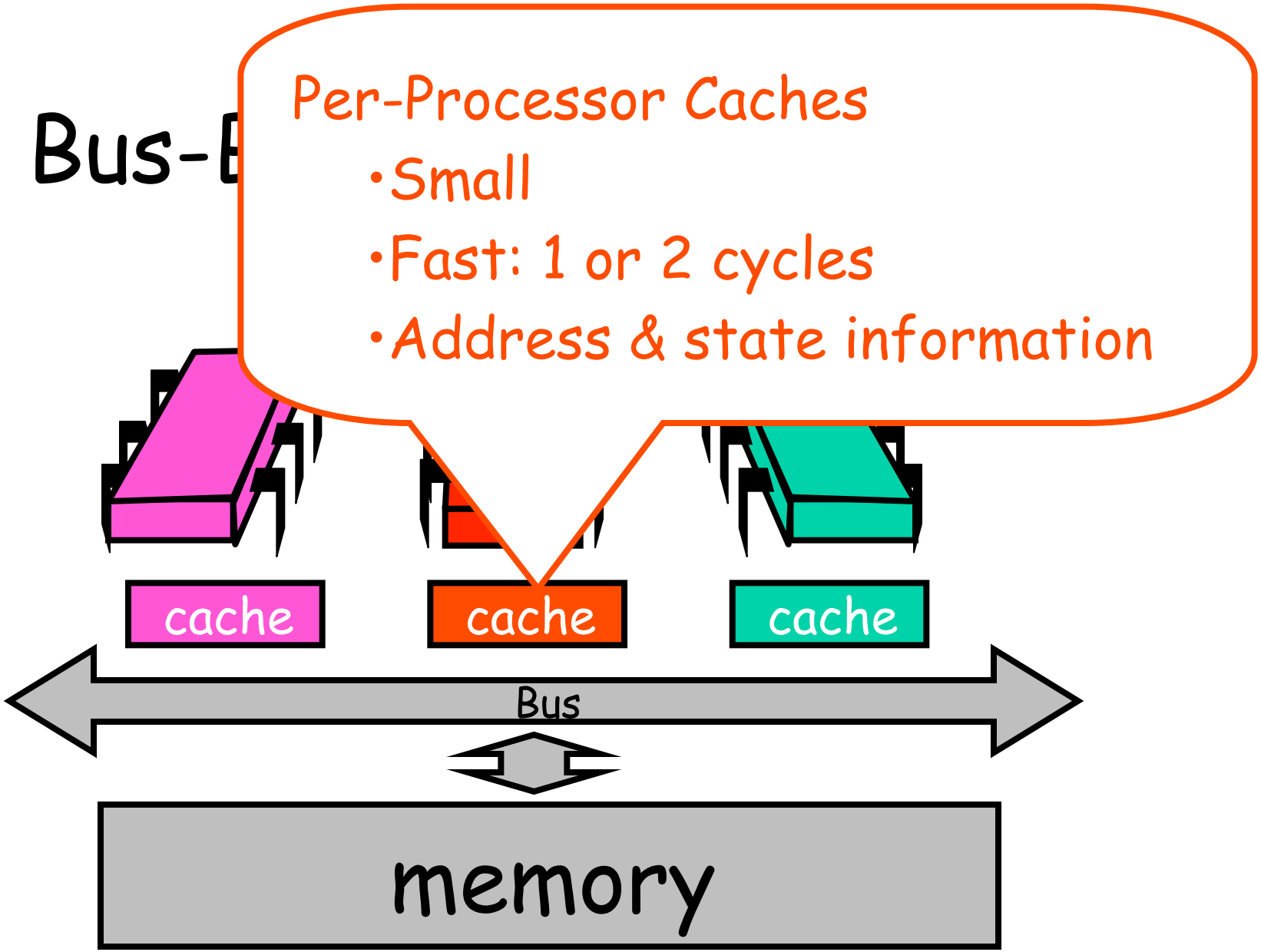
Bus-Based Architectures

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
cache
cache
cache
Bus
memory

Jargon Watch

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

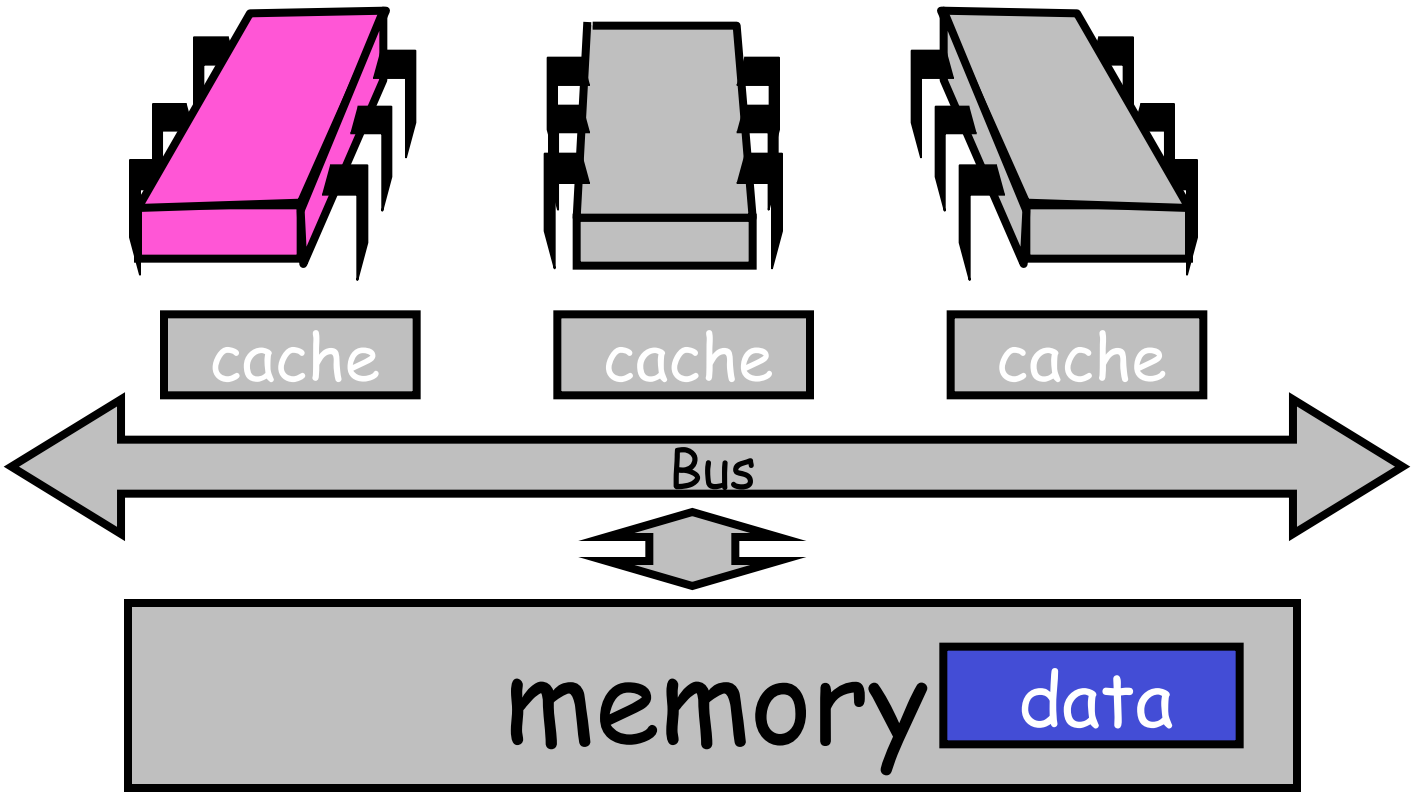
Jargon Watch

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

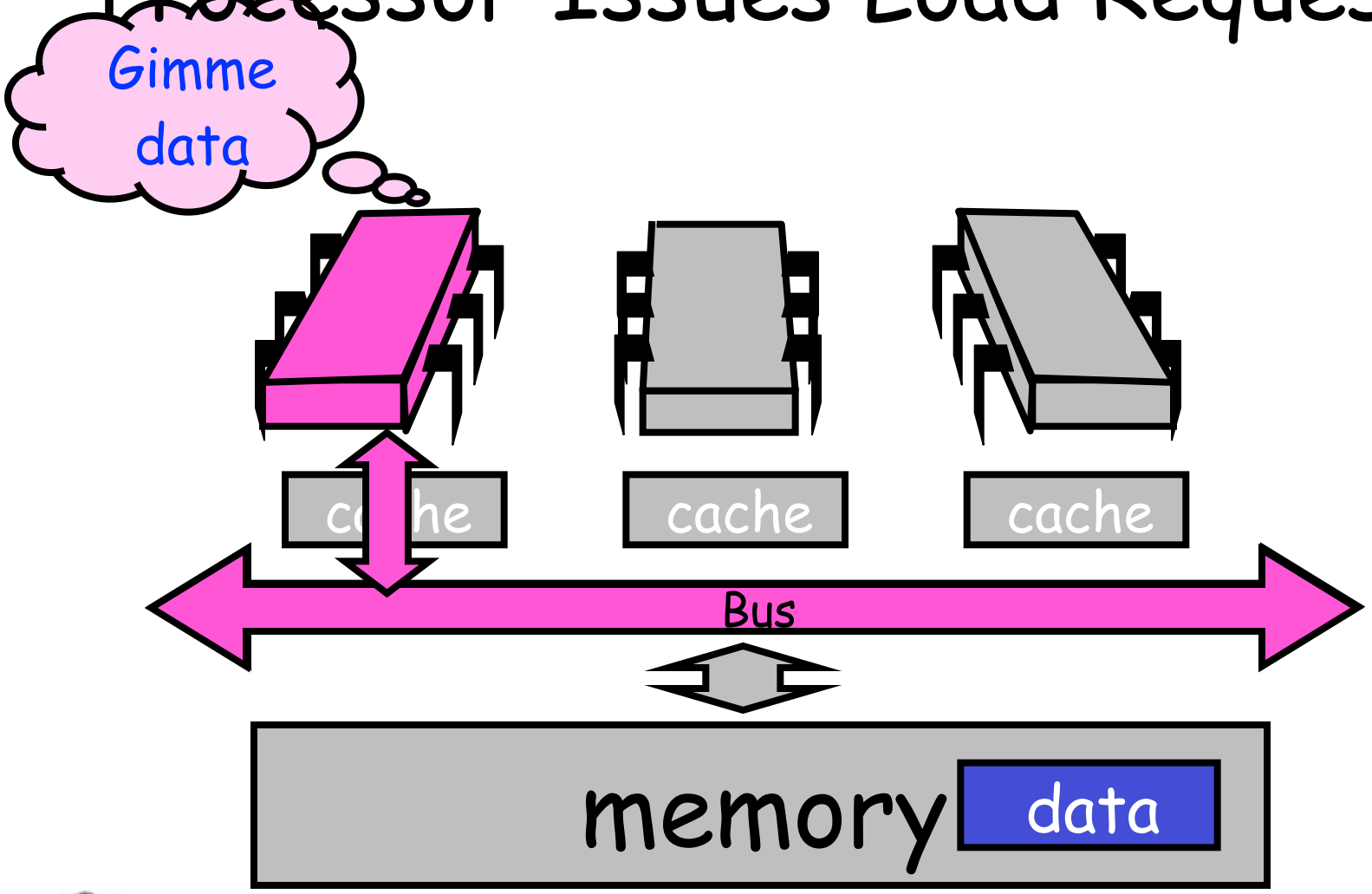
Cave Canem

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

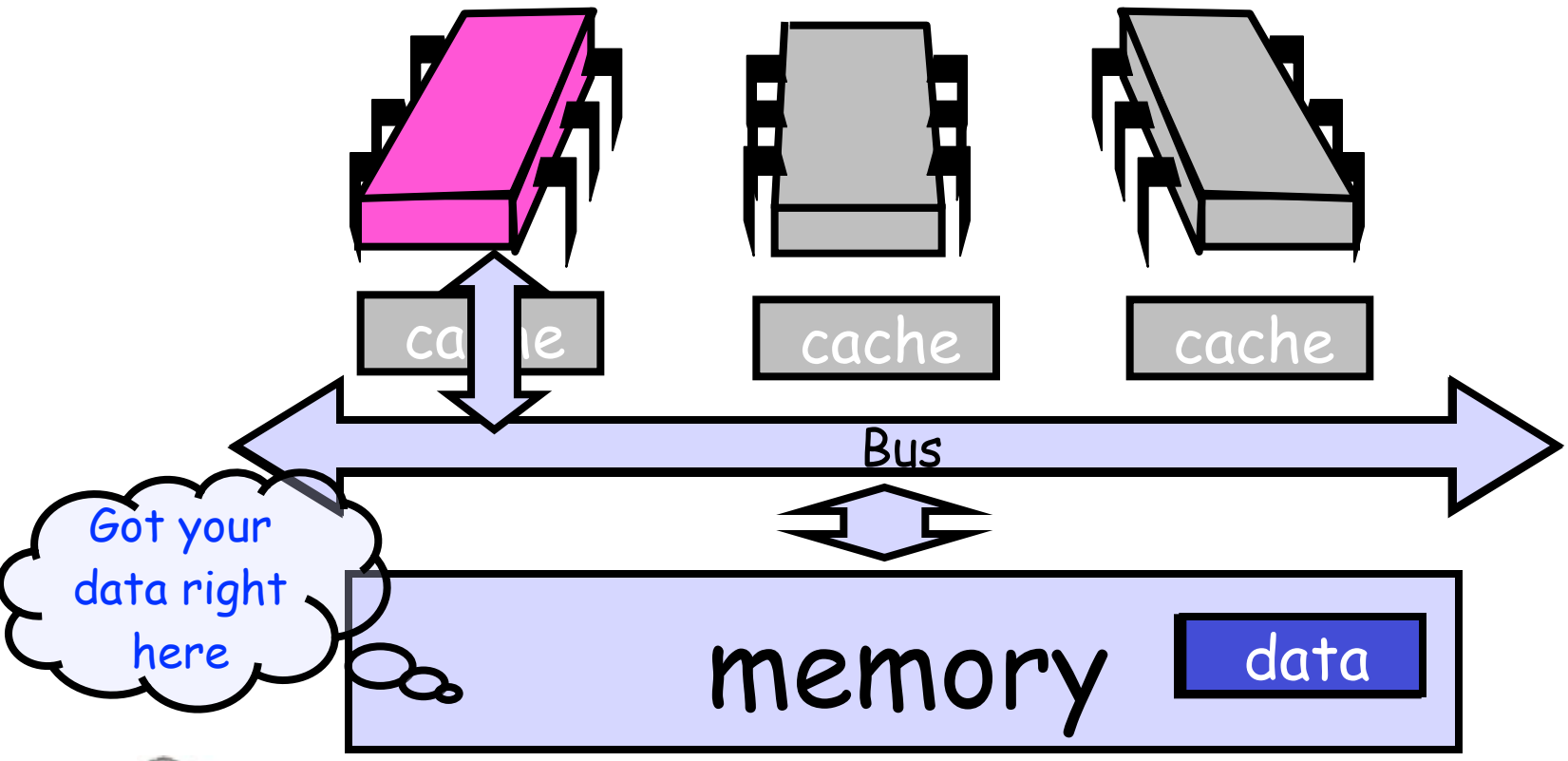
Processor Issues Load Request



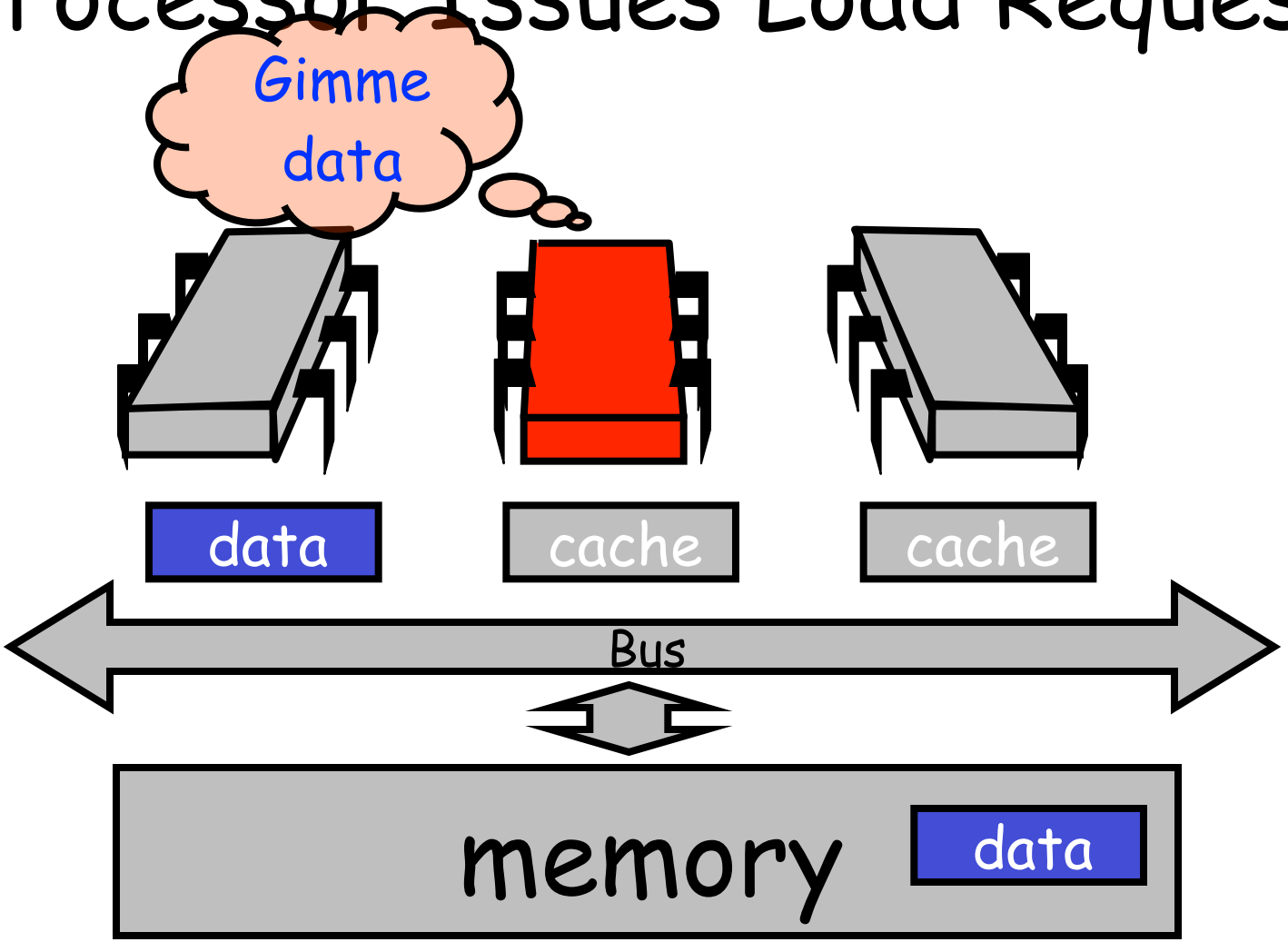
Processor Issues Load Request



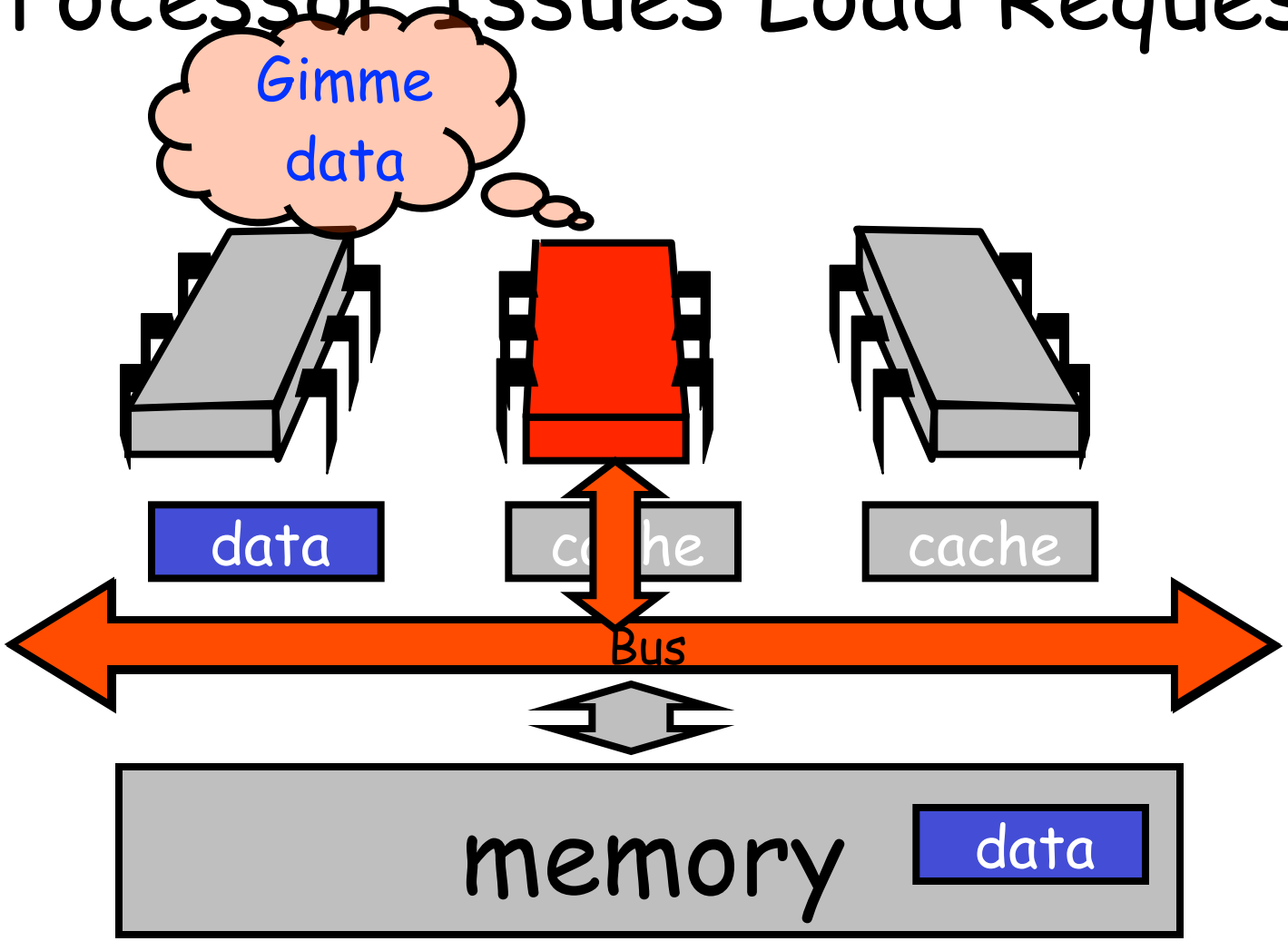
Memory Responds



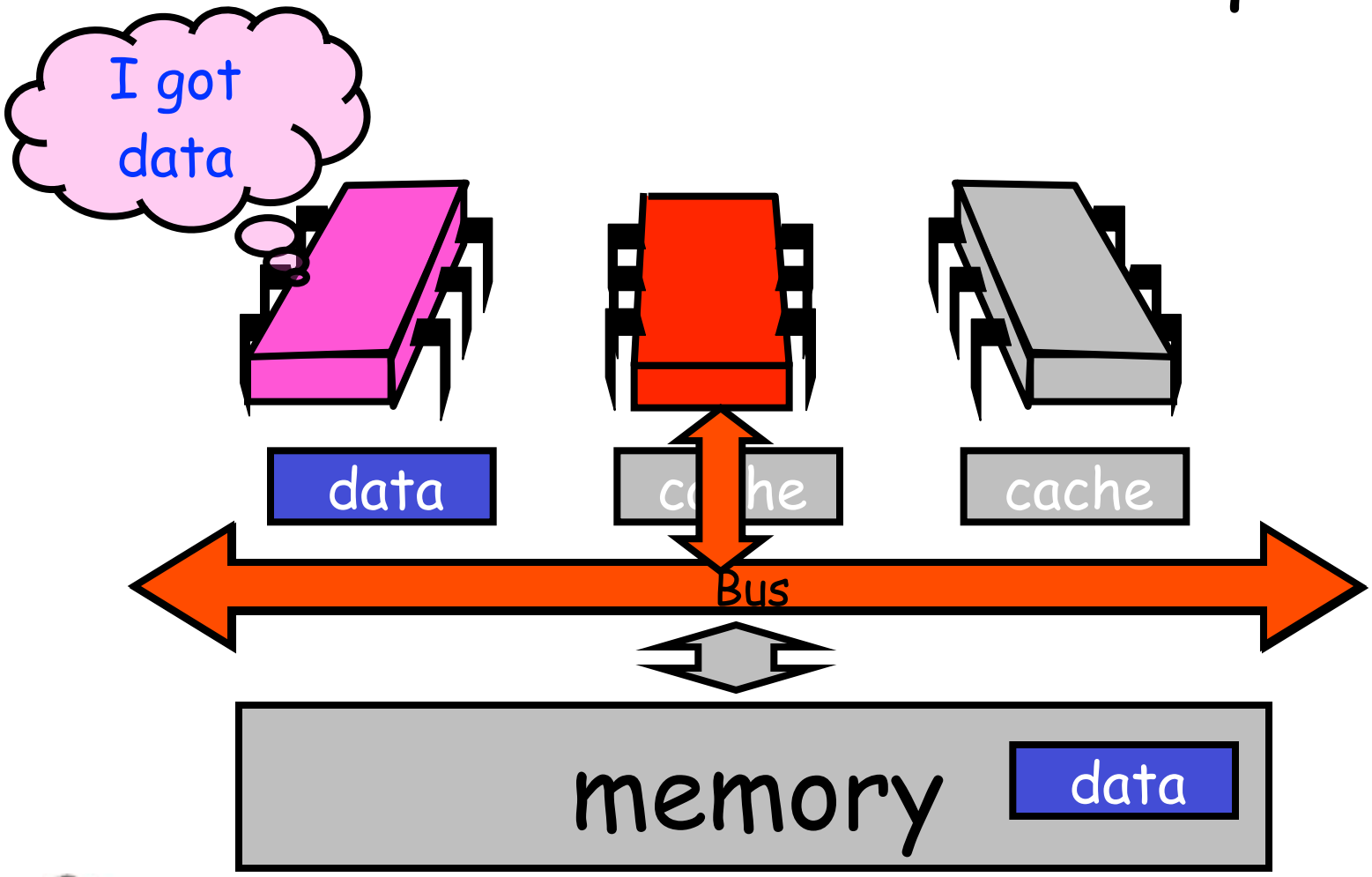
Processor Issues Load Request



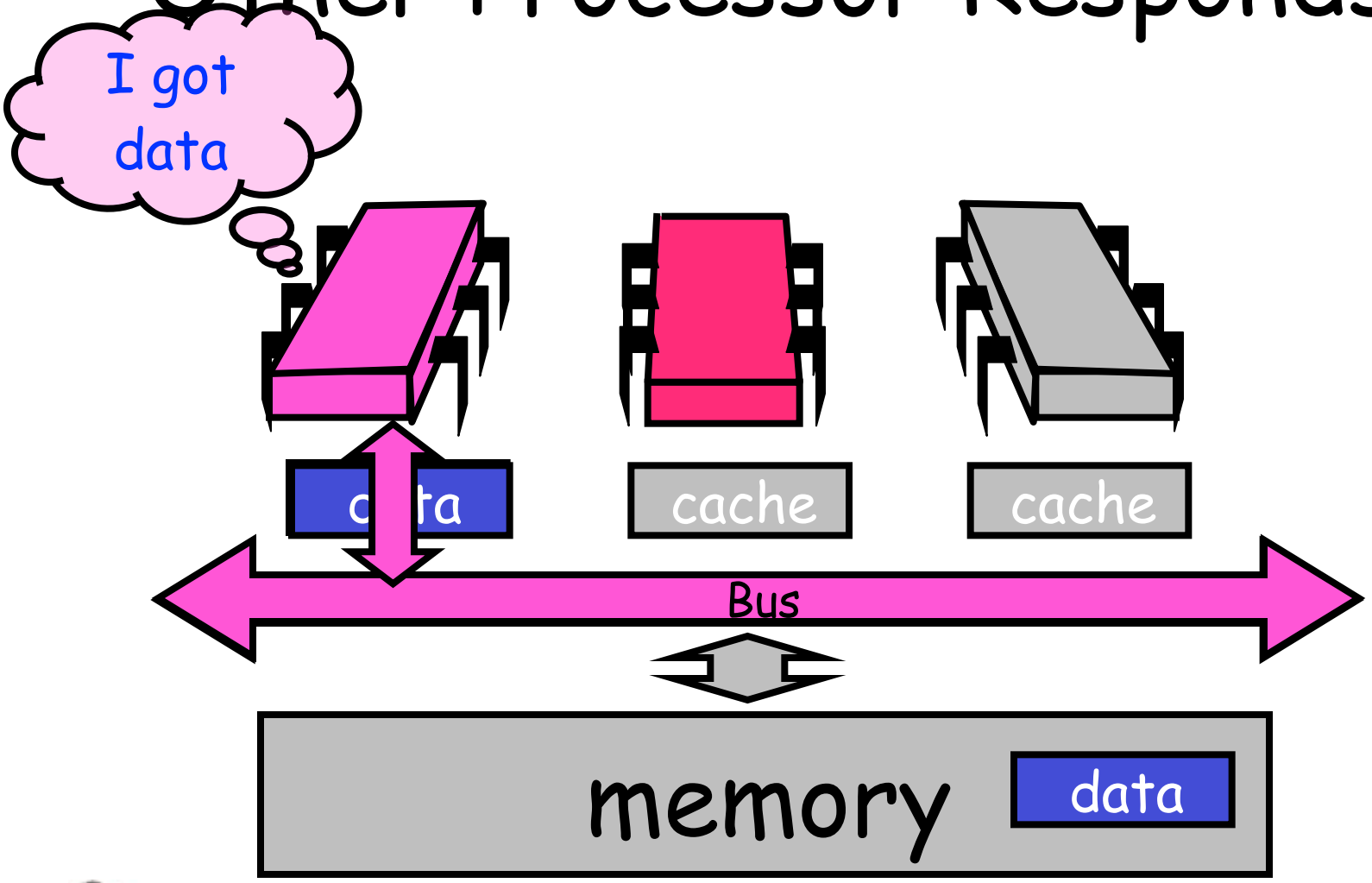
Processor Issues Load Request



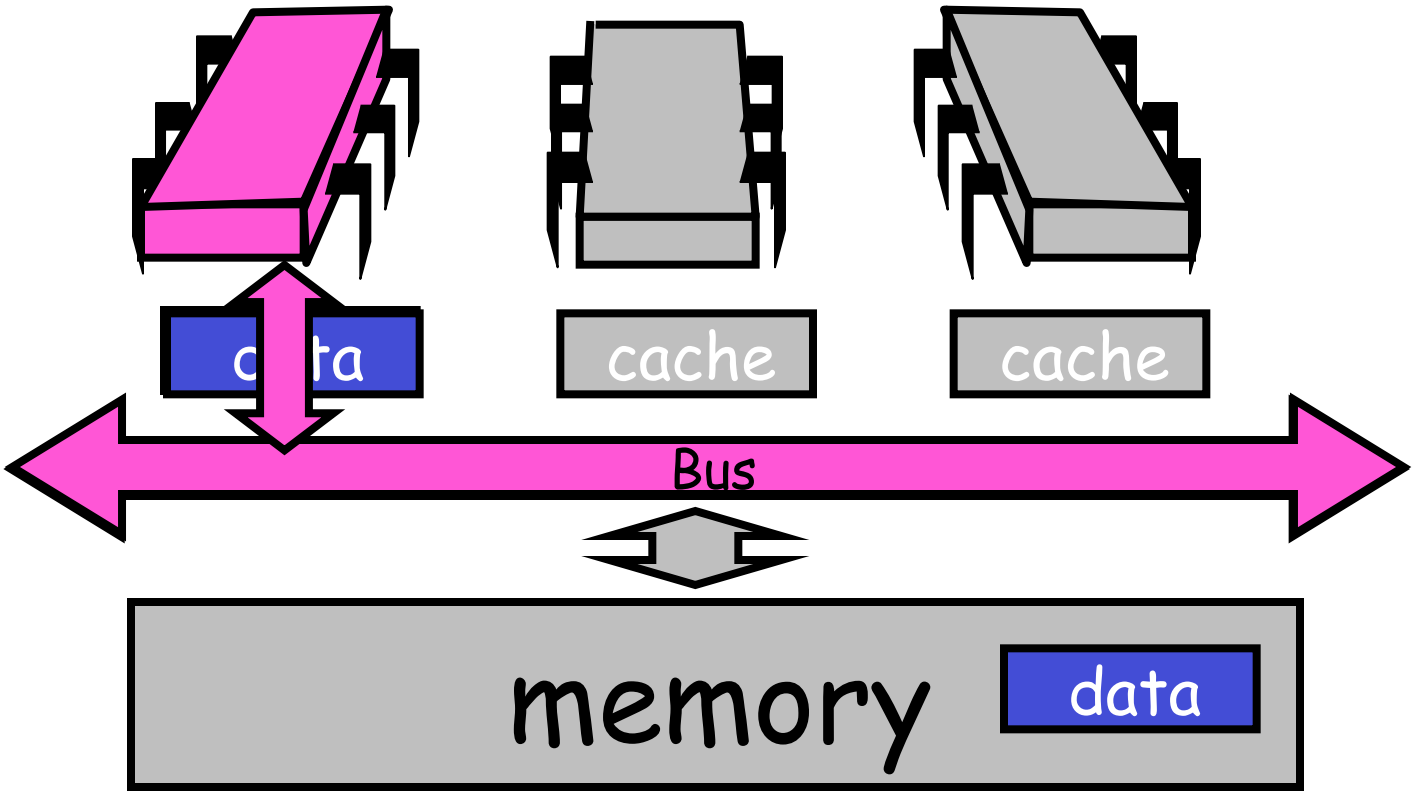
Processor Issues Load Request



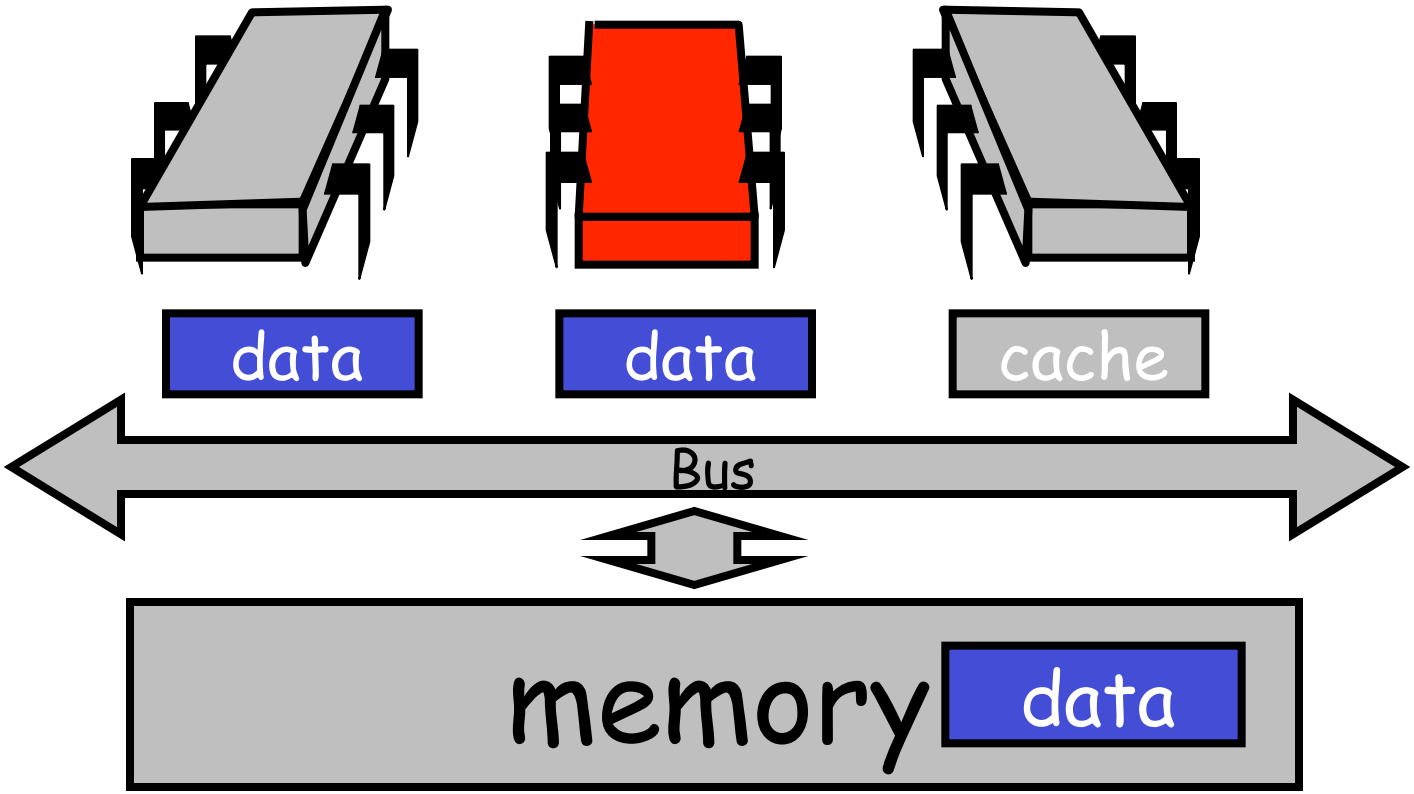
Other Processor Responds



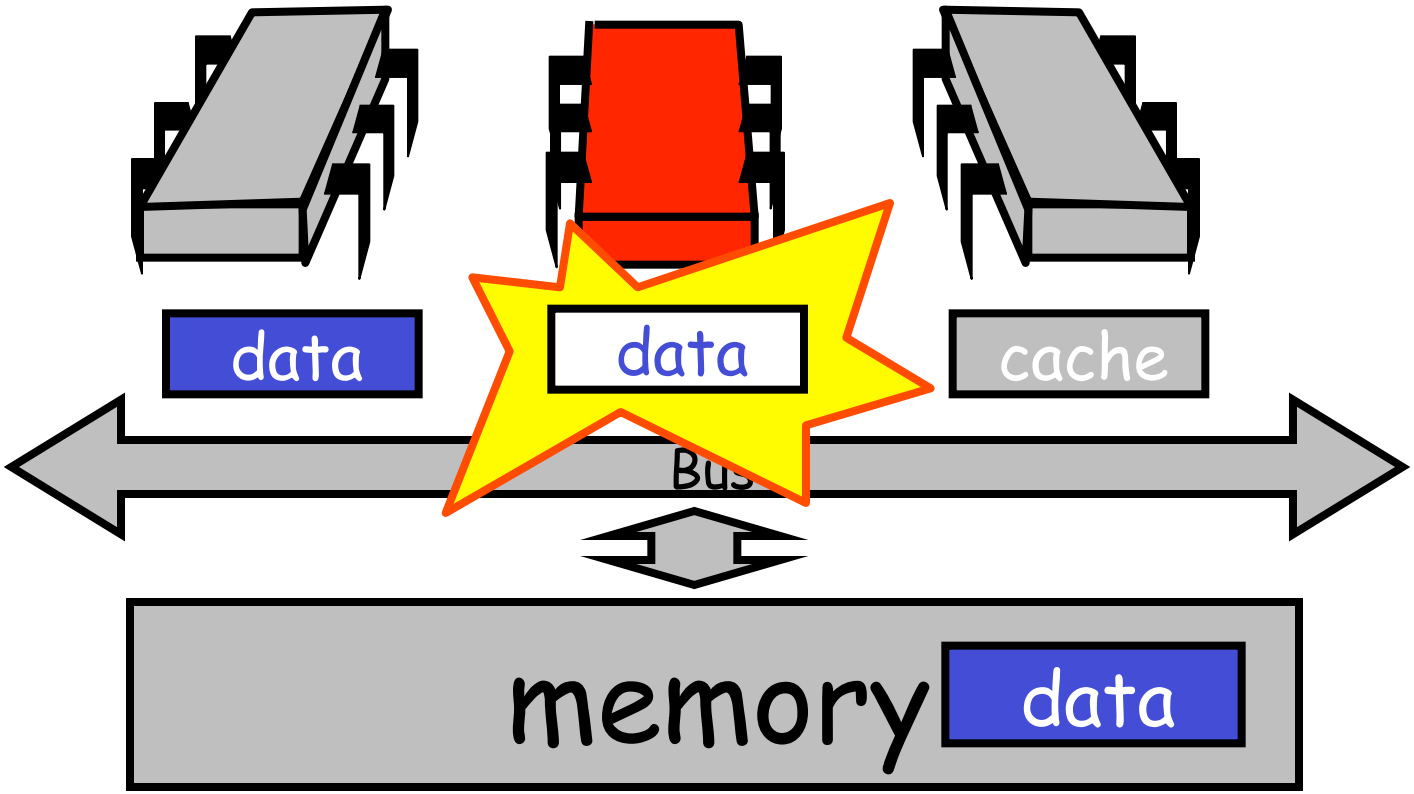
Other Processor Responds



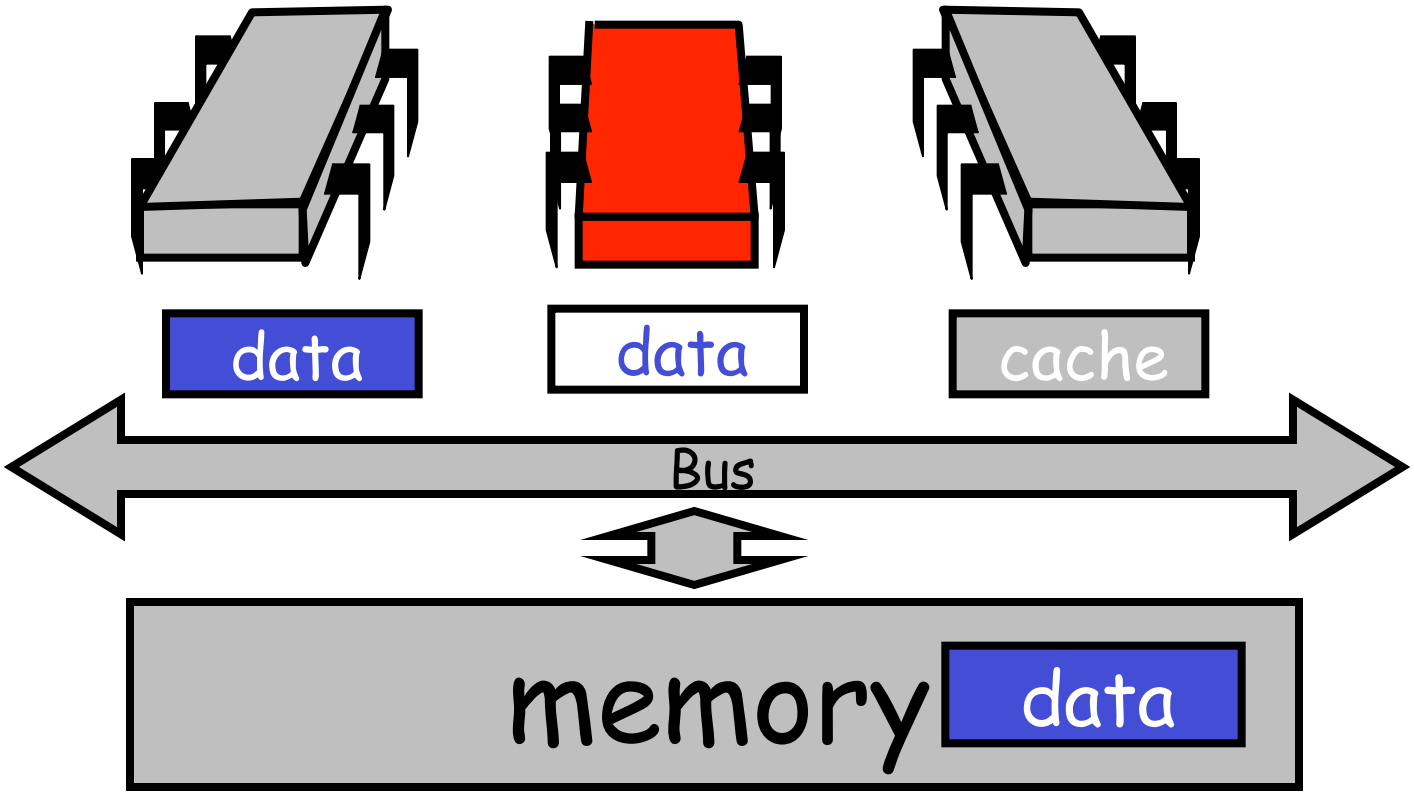
Modify Cached Data



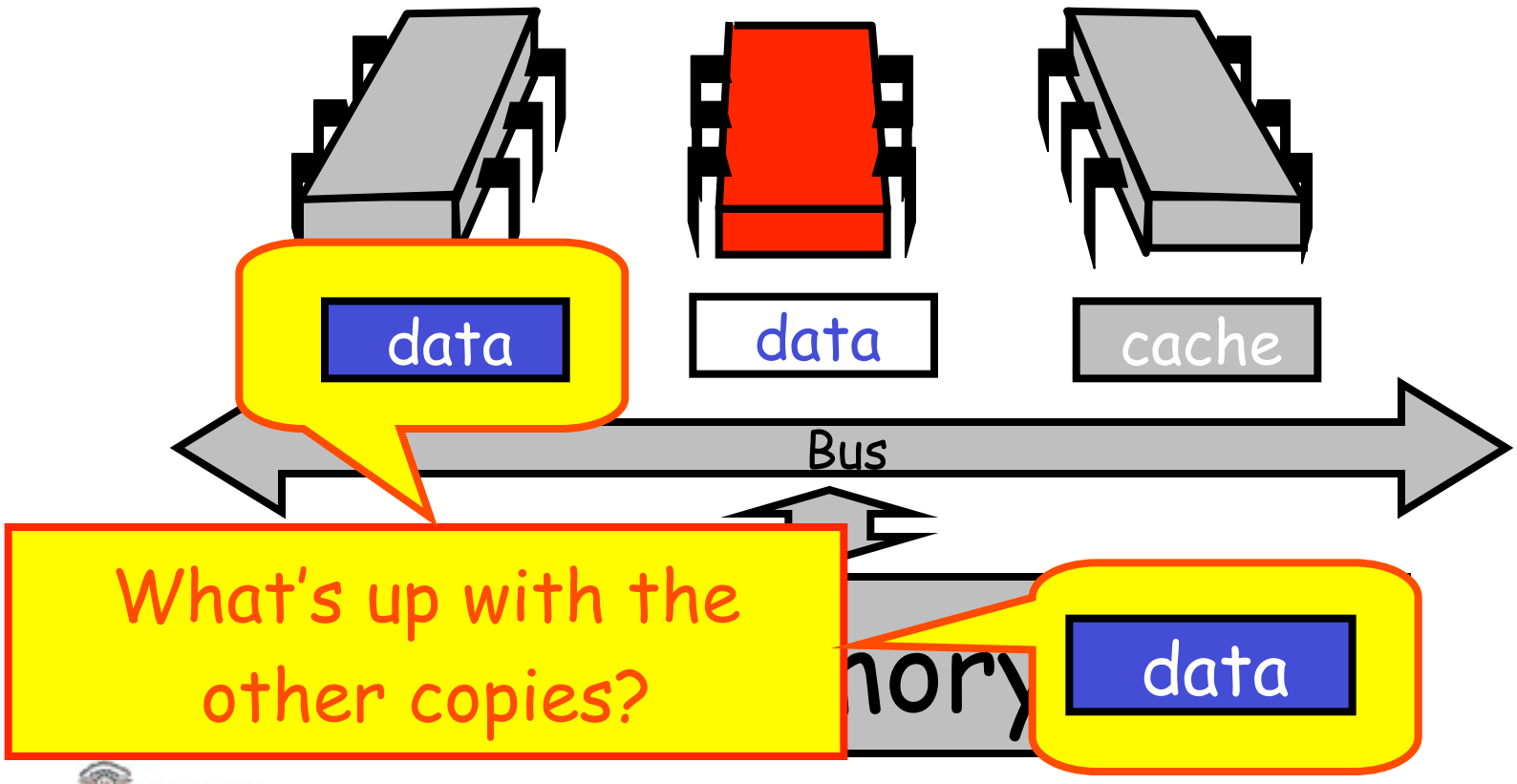
Modify Cached Data



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?

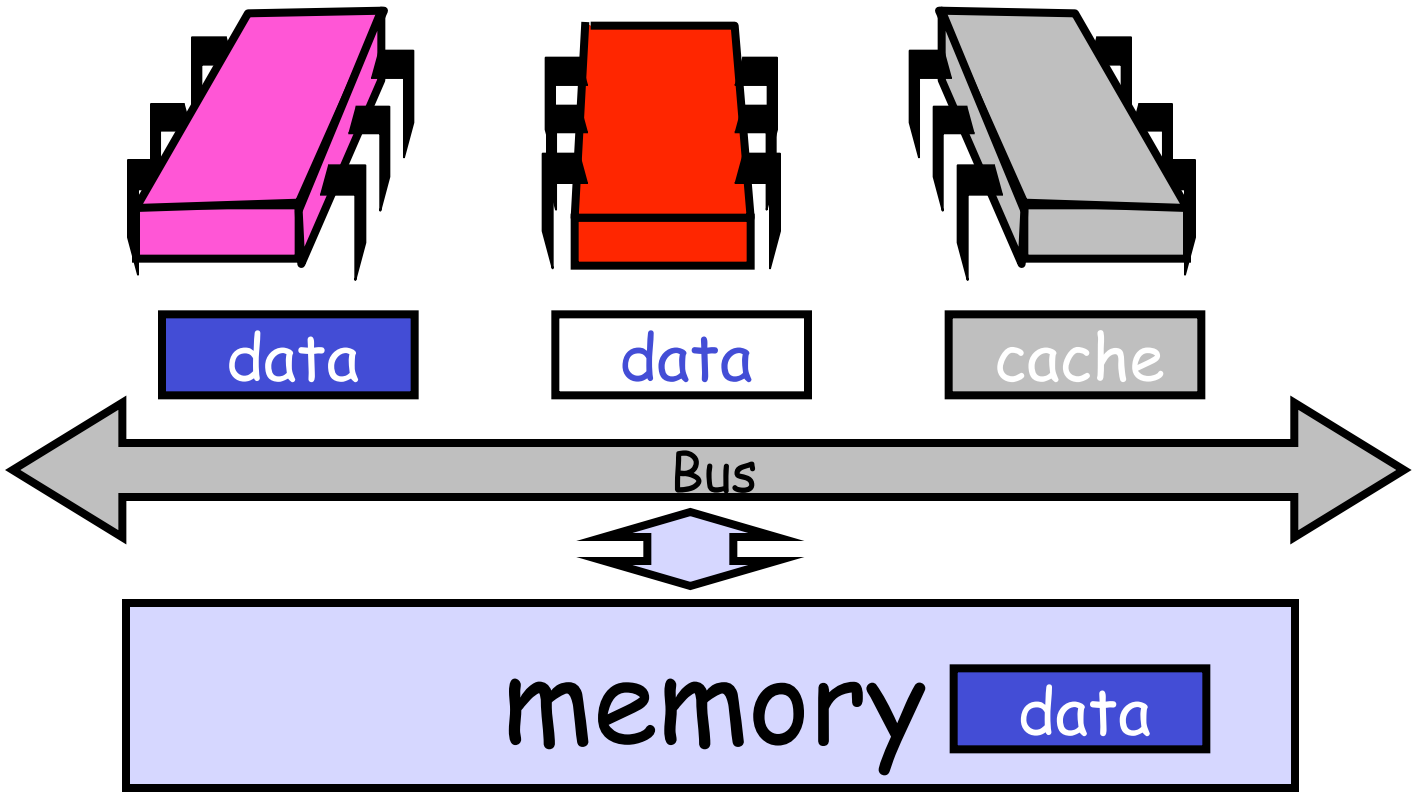
Write-Back Caches

- 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 …

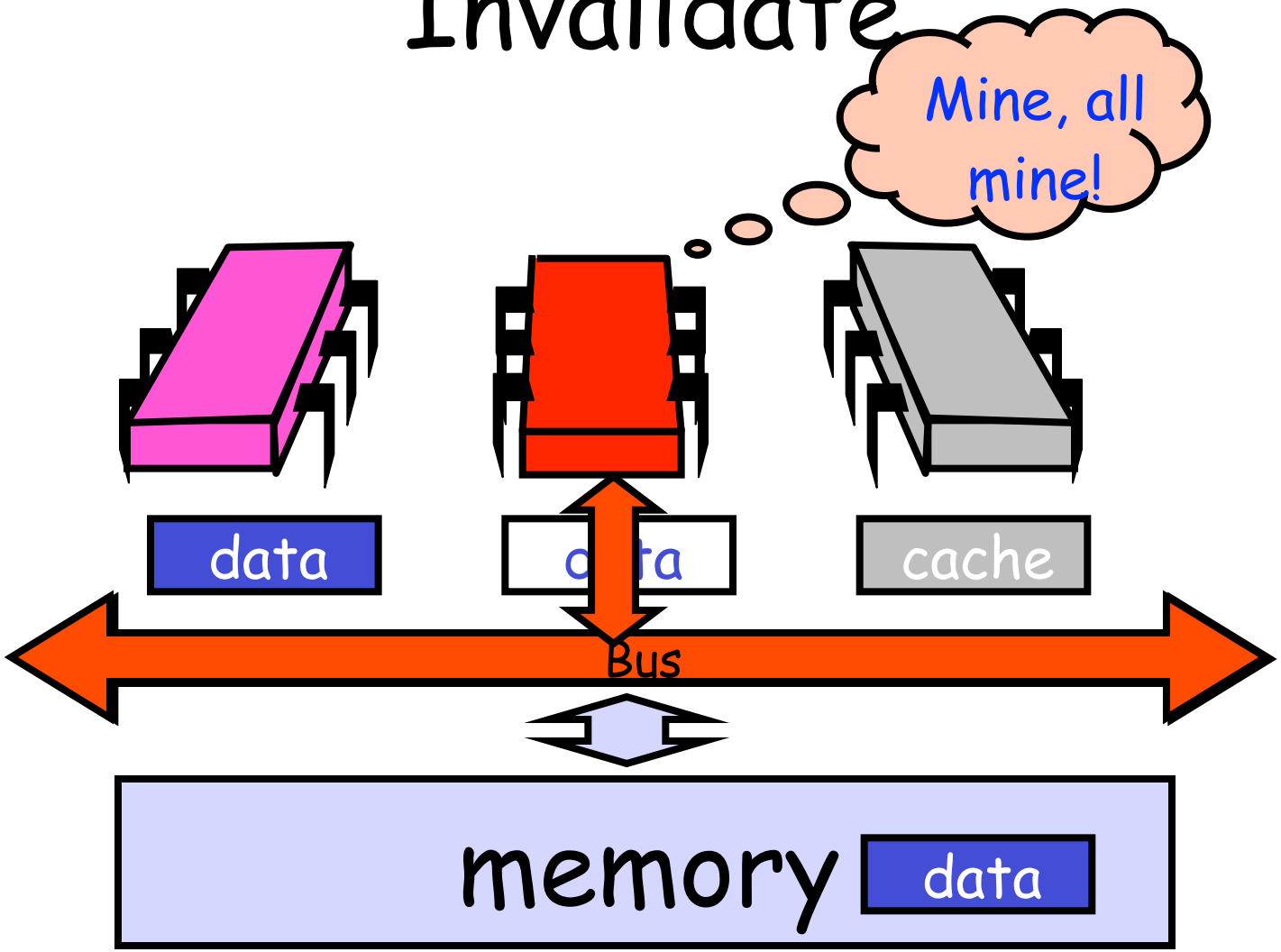
Write-Back Caches

- Cache entry has three states
  - Invalid: 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

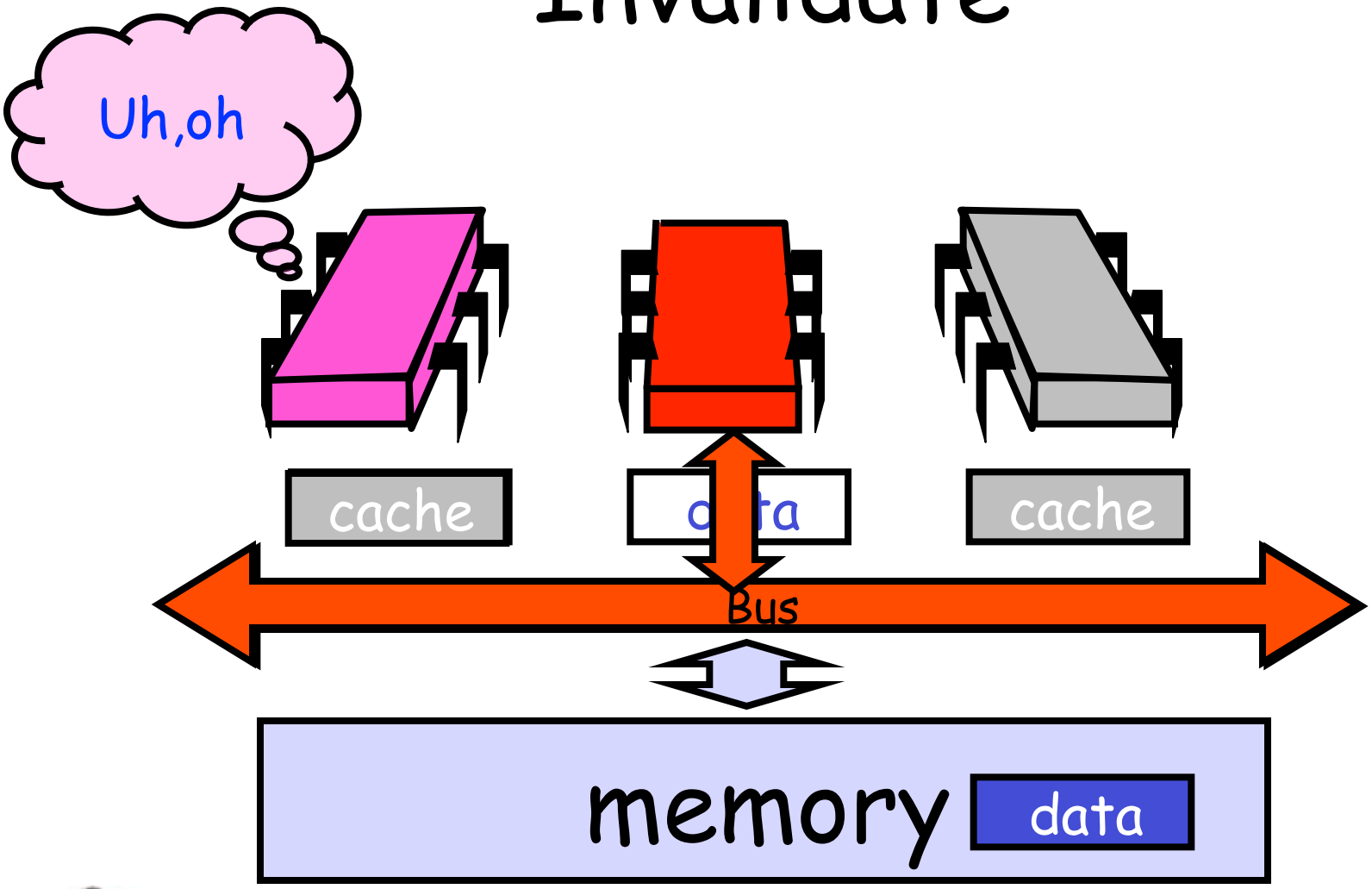
Invalidate



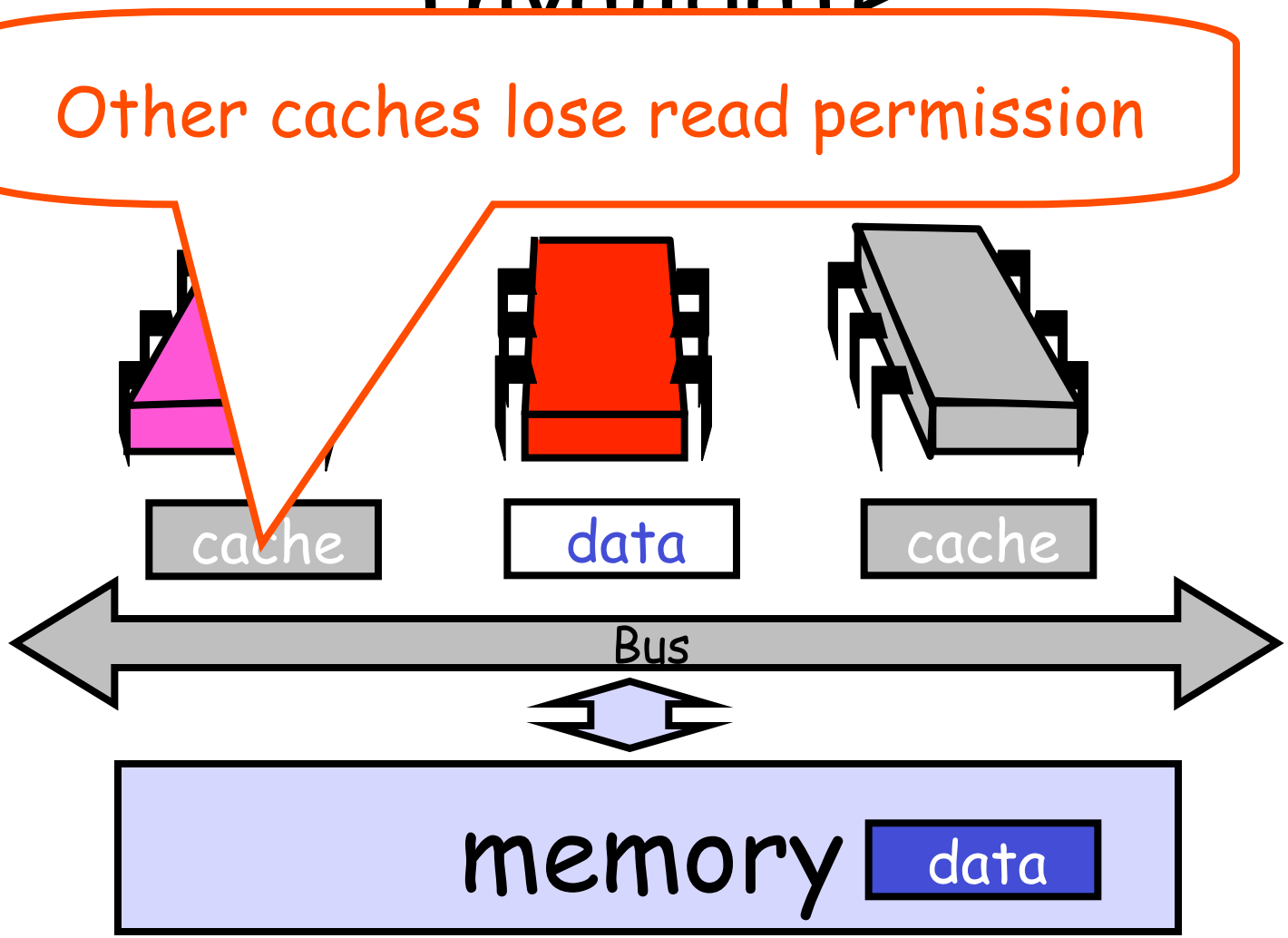
Invalidate



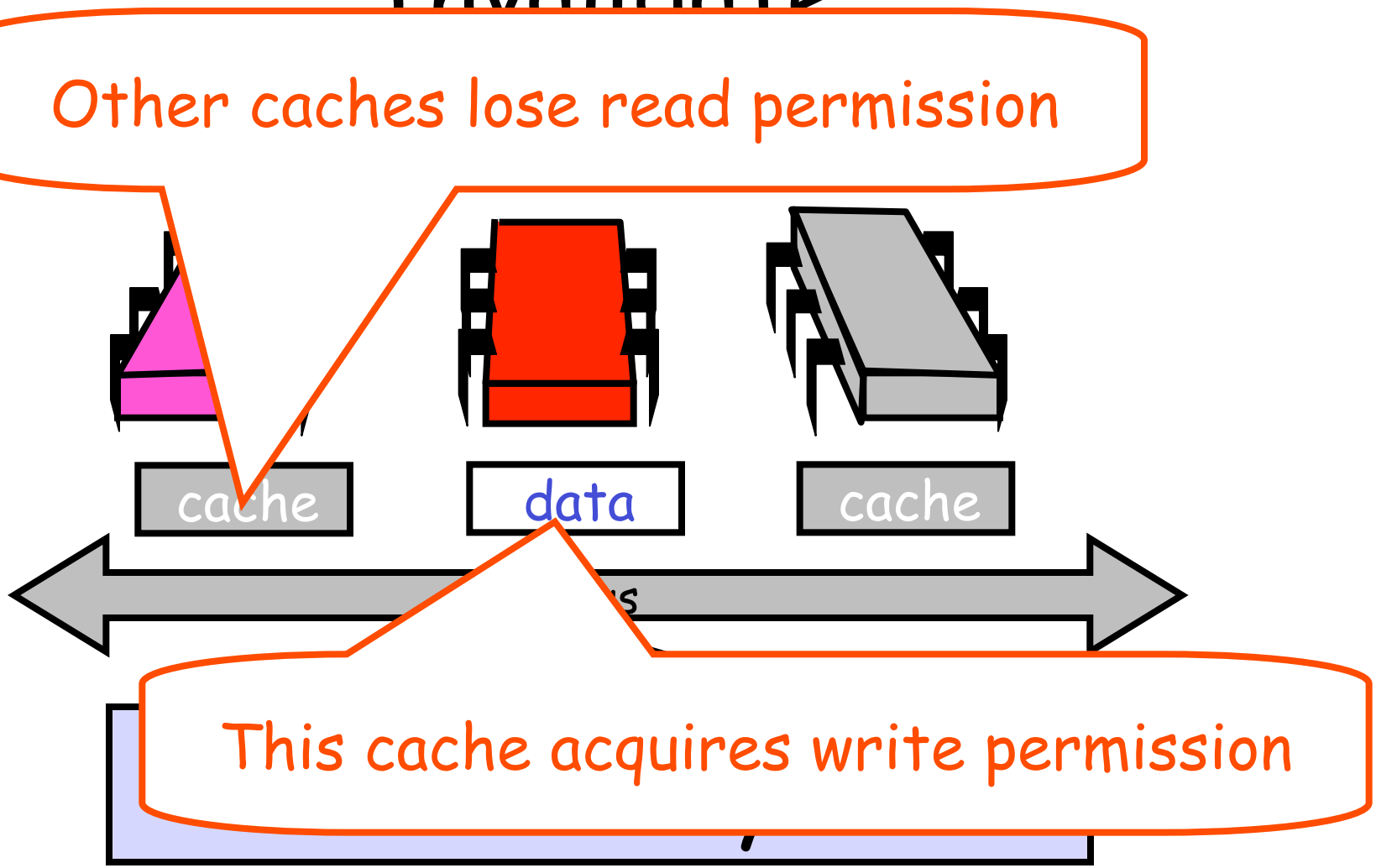
Invalidate



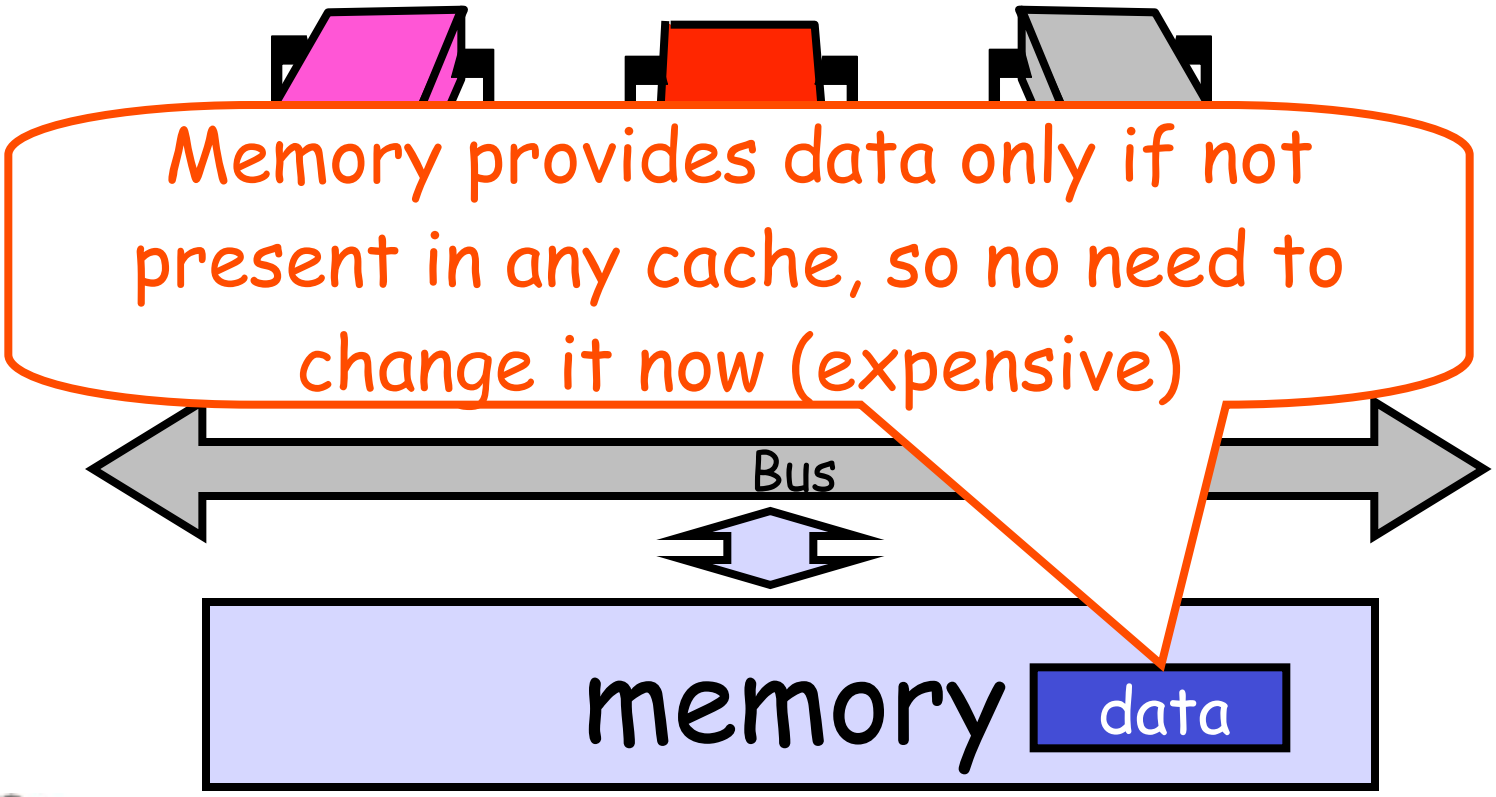
Invalidate


Other caches lose read permission
cache
data
cache
Bus
memory
data

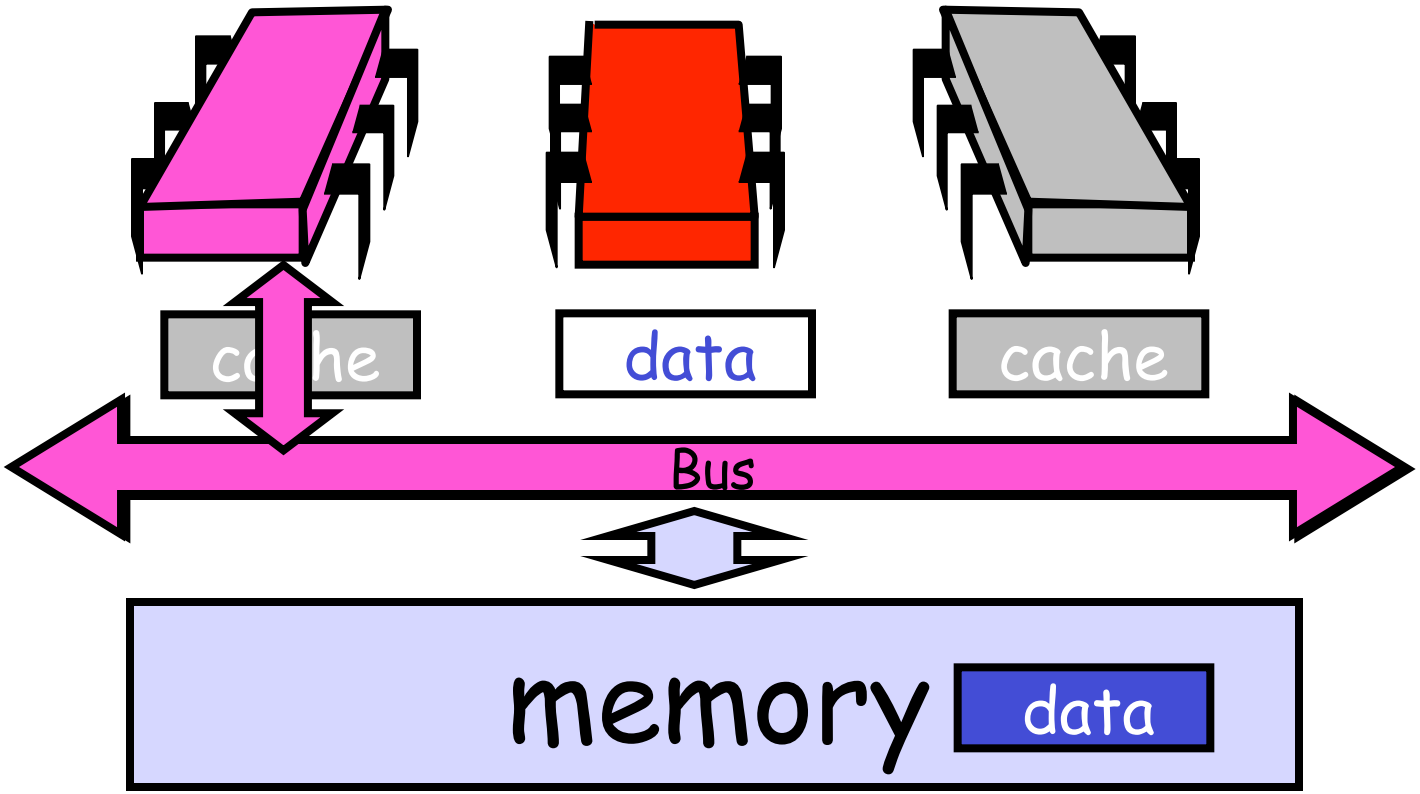
Invalidate


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

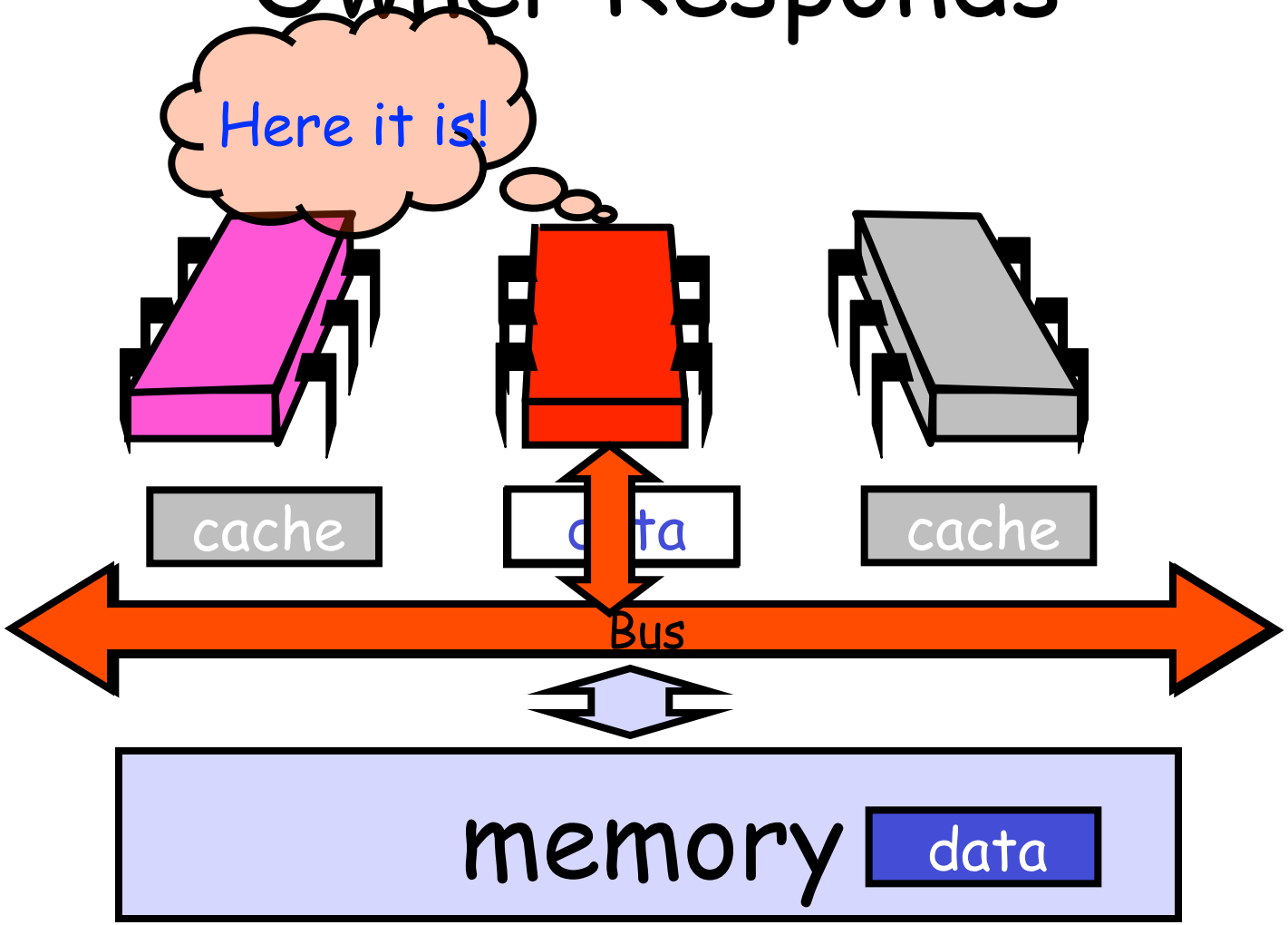
Invalidate


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

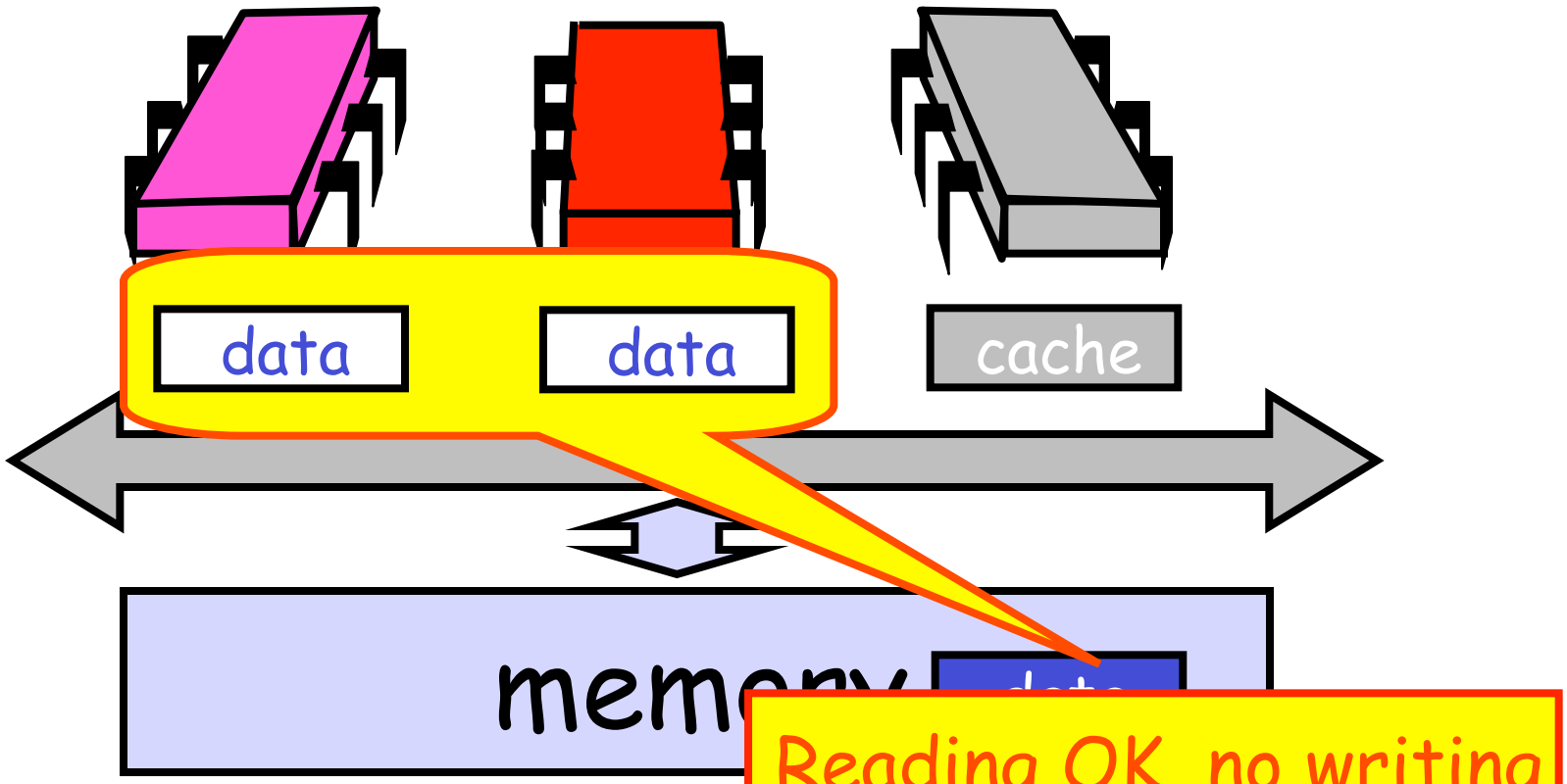
Another Processor Asks for Data



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

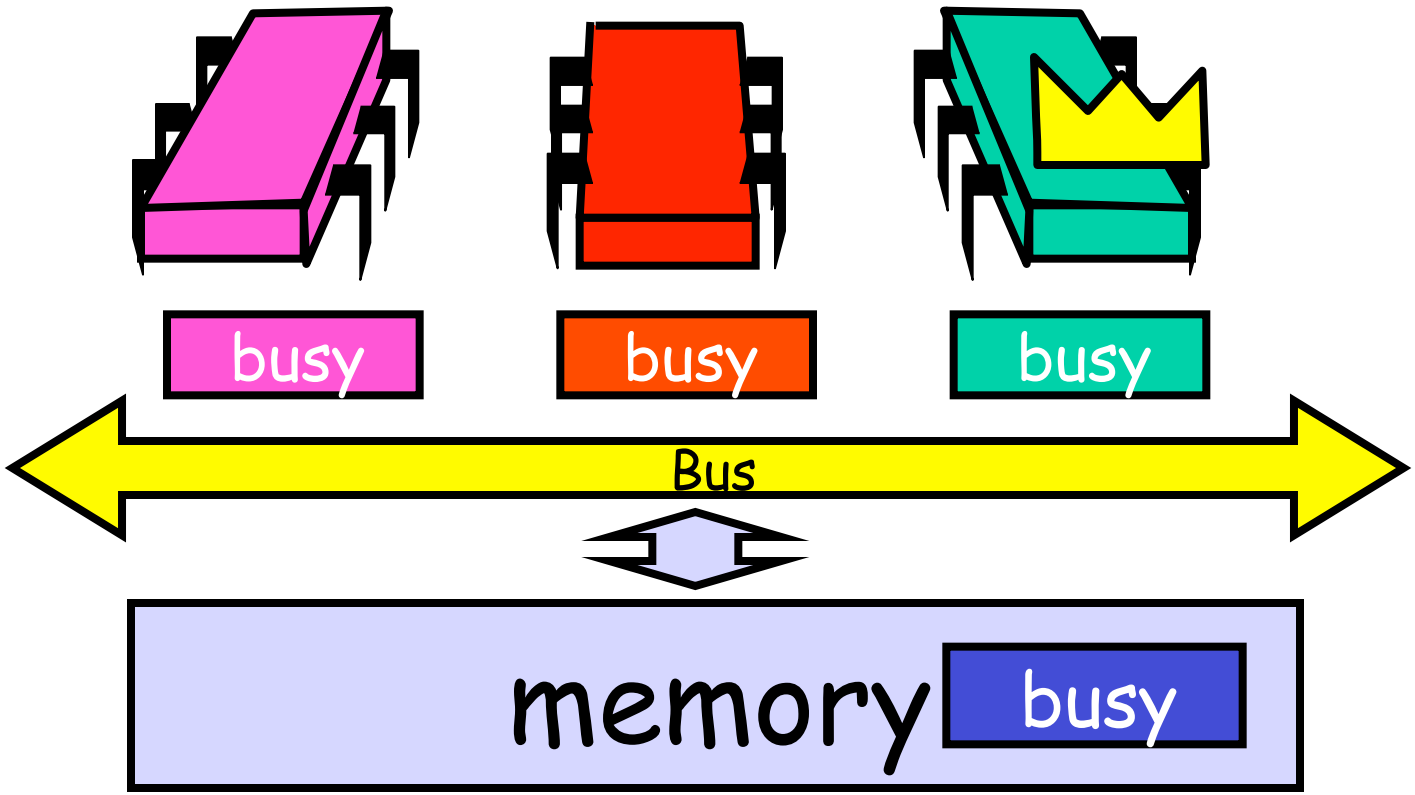
Simple TASLock

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

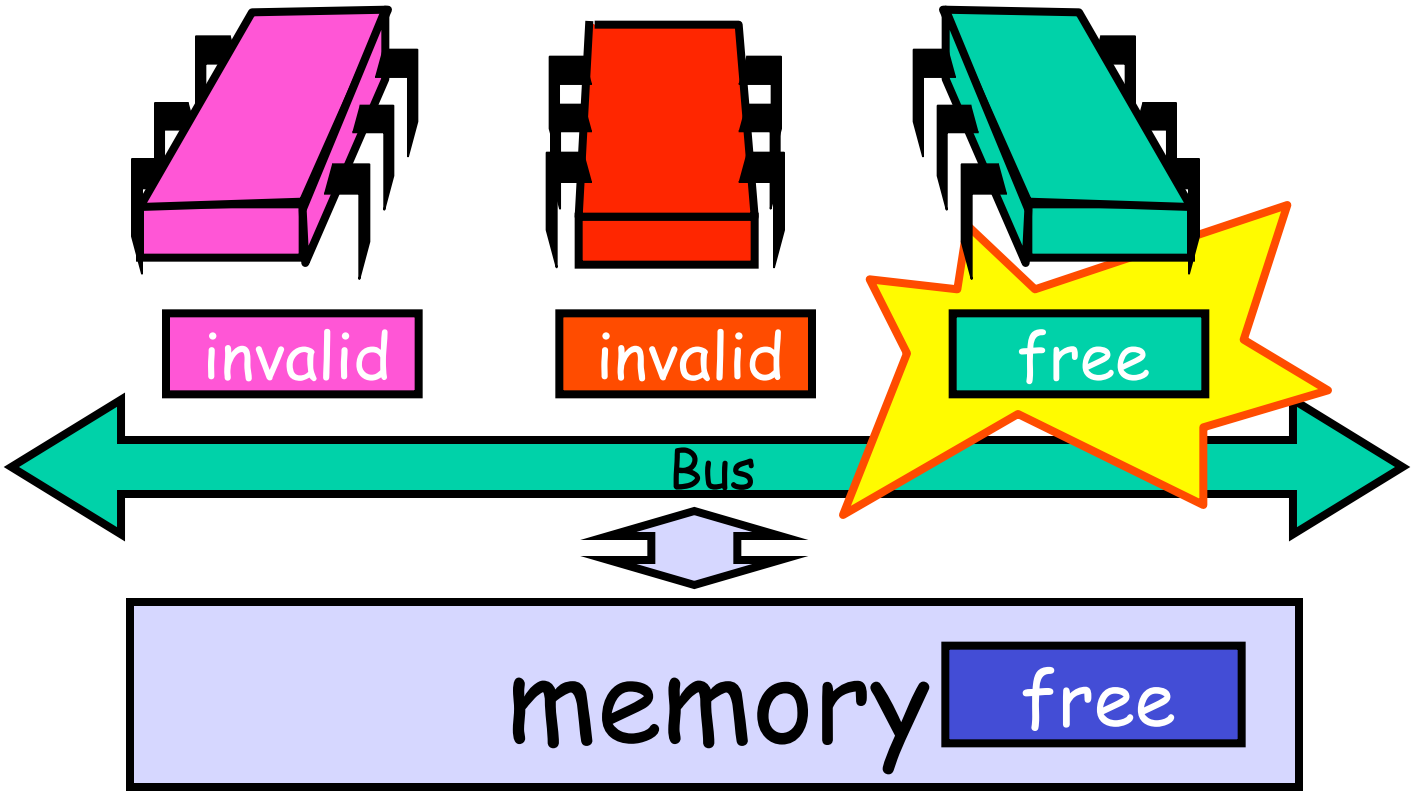
Test-and-test-and-set

- 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

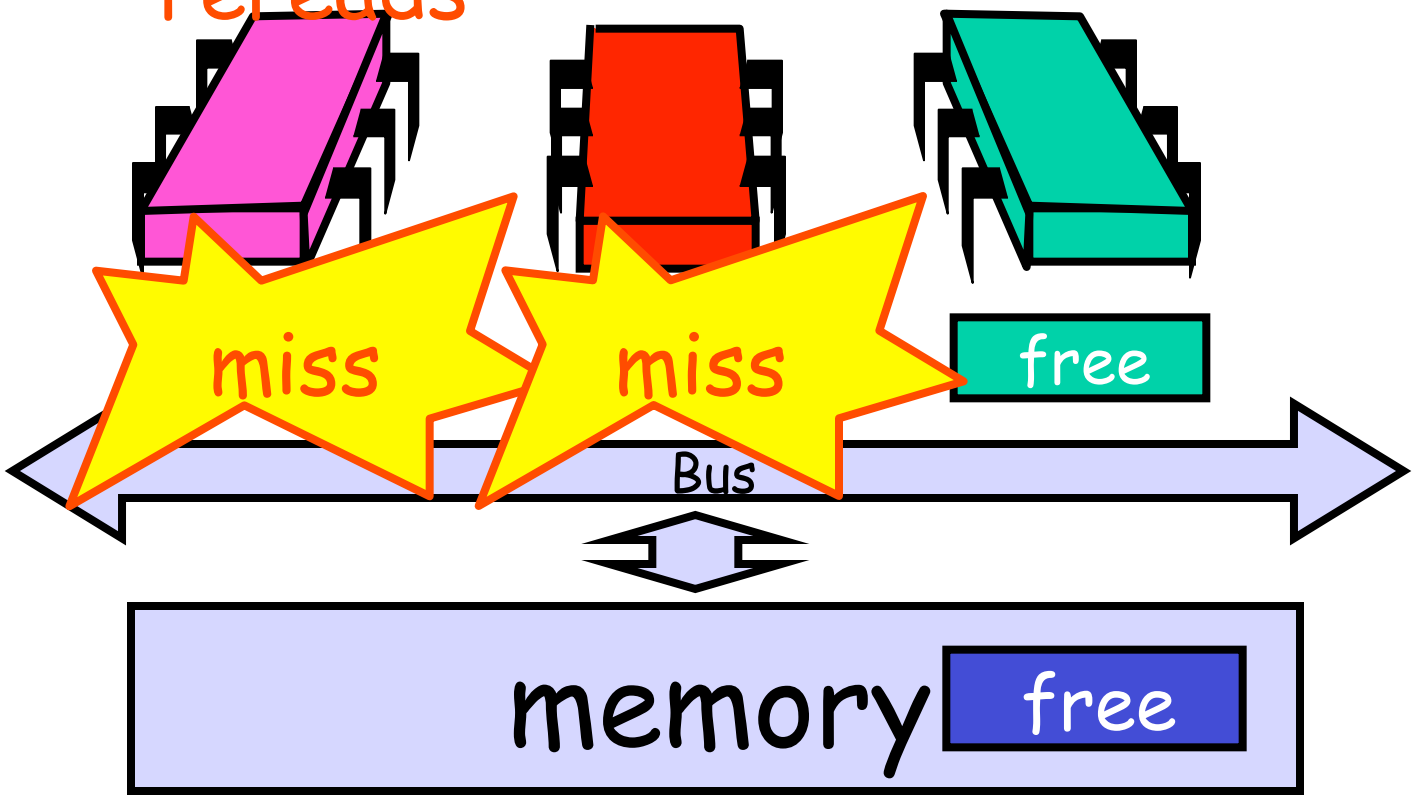


On Release



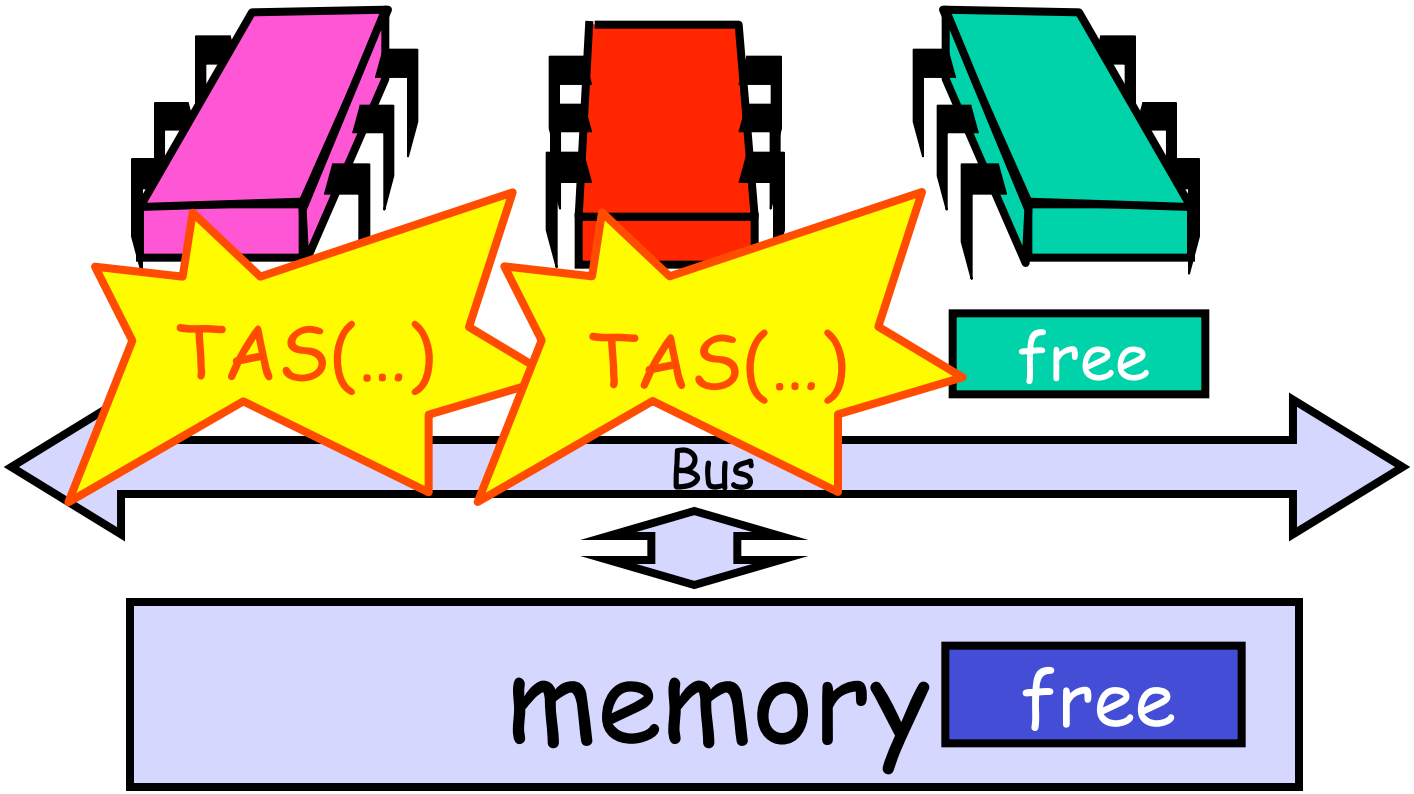
On Release

Everyone misses, rereads



On Release

Everyone tries TAS



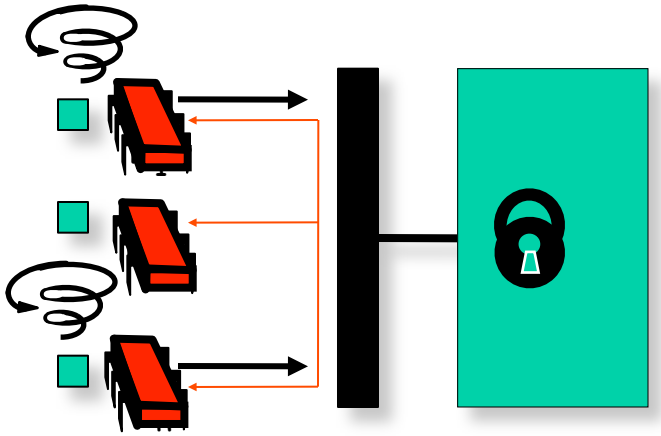
Problems

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

Measuring Quiescence Time

X = time of ops that don't use the bus

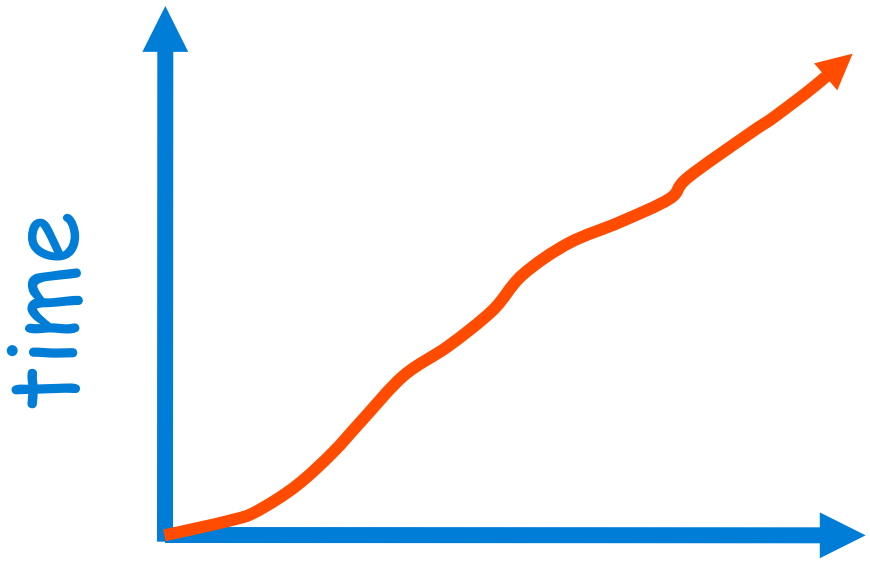
Y = time of ops that cause intensive bus traffic



In critical section, run ops X then ops Y. As long as Quiescence time is less than X, no drop in performance.

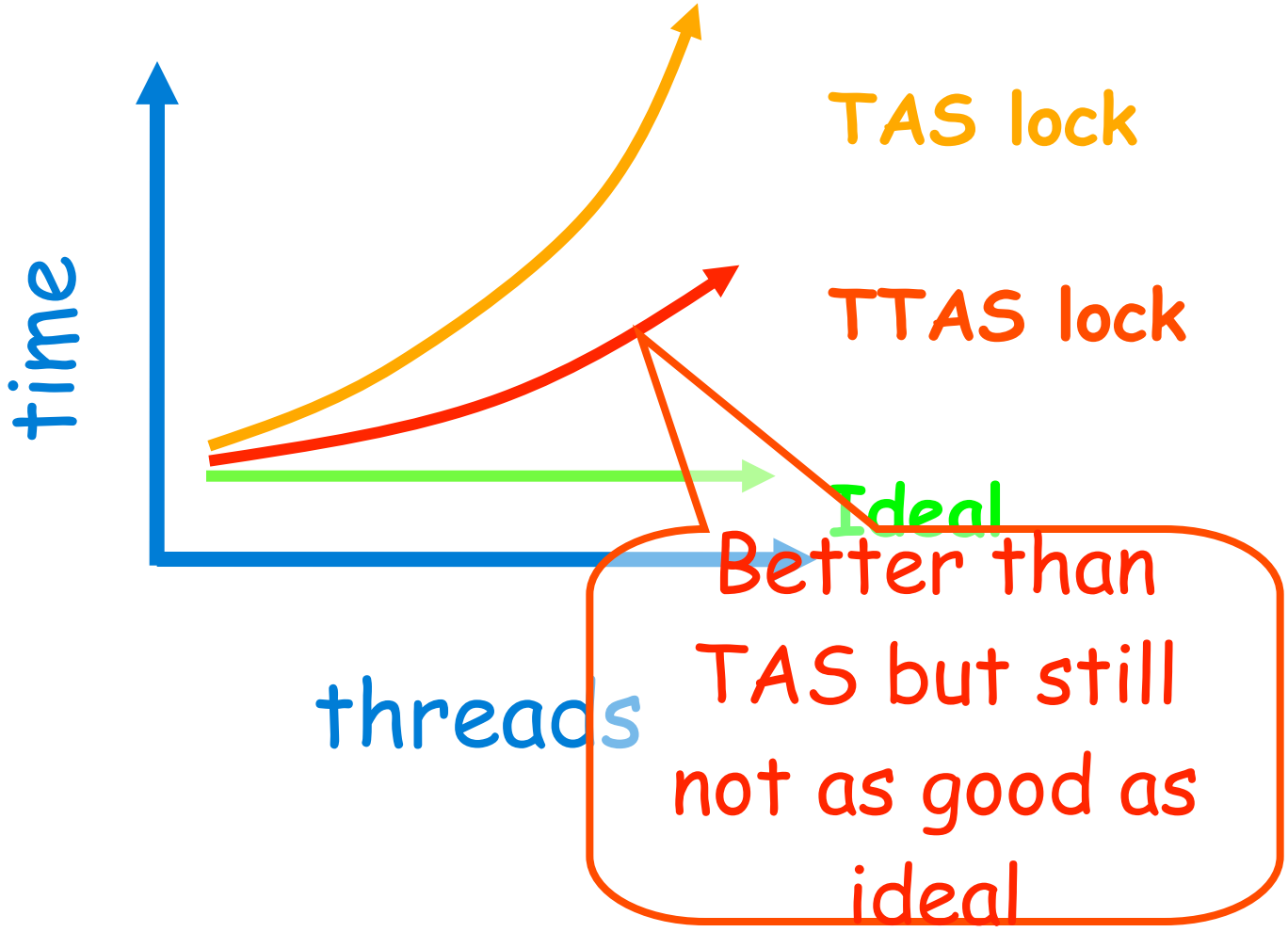
By gradually varying X, can determine the exact time to quiesce.

Quiescence Time



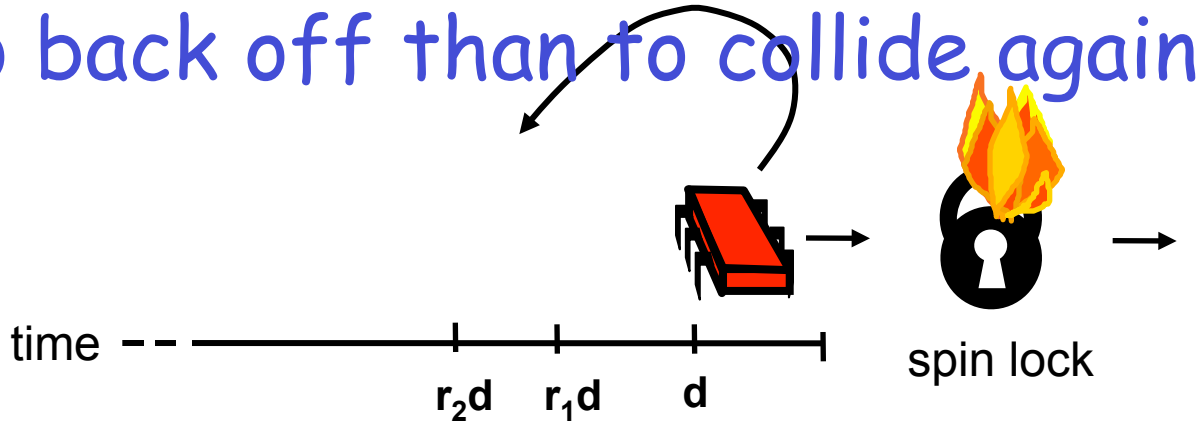
Increses linearly with the number of processors for bus architecture

Mystery Explained

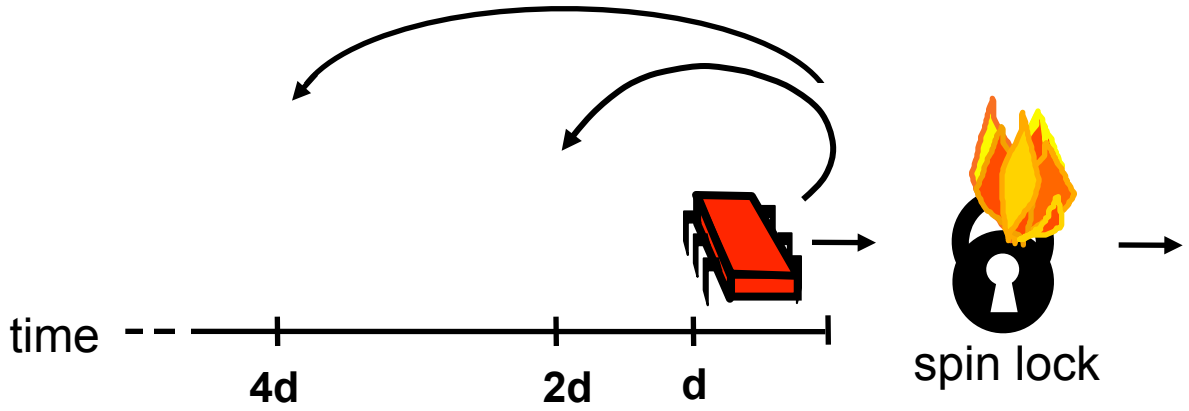


Solution: Introduce Delay

- 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 * 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 c
 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;
 }}}
```

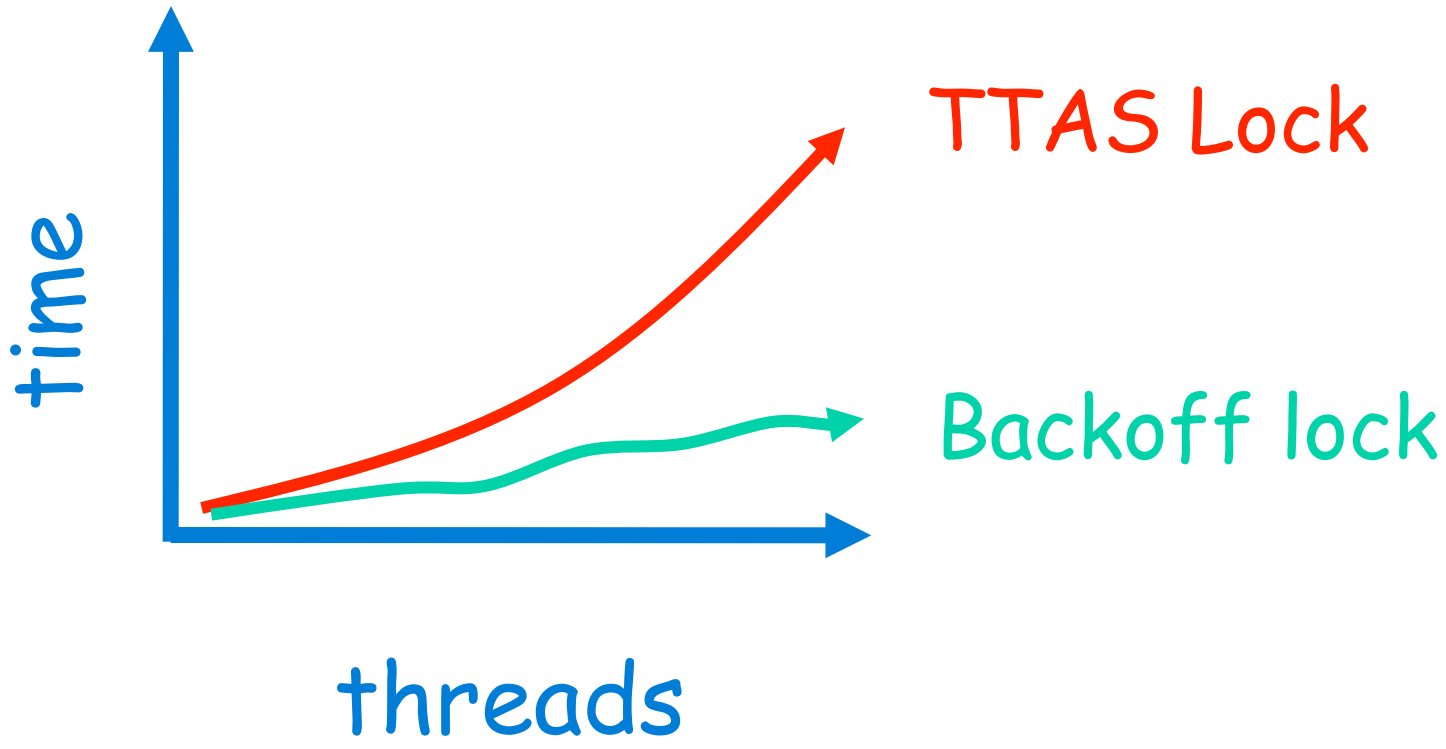
Back off for random duration

Exponential Backoff Lock

```
public
 publi
  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



Locks and Busy Waiting

```
Lock::Acquire() {
   while (test&set(lock) == 1)
      ; // spin
}
```

- Busy-waiting:
  - Threads consume CPU cycles while waiting
  - Low latency to acquire
- Limitations
  - Occupies a CPU core
  - What happens if threads have different priorities?
    - Busy-waiting thread remains runnable
    - If the thread waiting for a lock has higher priority than the thread occupying the lock, then ?
    - Ugh, I just wanted to lock a data structure, but now I'm involved with the scheduler!
  - What if programmer forgets to unlock?

Cheaper Locks with Cheaper busy waiting

Using Test&Set

```
Lock::Acquire() {
while (test&set(lock) == 1);
}
```

With busy-waiting

```
Lock::Acquire() {
while(1) {
  if (test&set(lock) == 0)  break;
  else sleep(1);
}
```

With voluntary yield of CPU

```
Lock::Release() {
   *lock = 0;
}
```

```
Lock::Release() {
*lock = 0;
}
```

- What is the problem with this?
  - A. CPU usage B. Memory usage C. Lock::Acquire() latency
  - D. Memory bus usage E. Messes up interrupt handling

Cheap Locks with Cheap busy waiting

Using Test&Test&Set

```
Lock::Acquire() {
while (test&set(lock) == 1);
}
```

Busy-wait on in-memory copy

```
Lock::Acquire() {
while(1) {
  while (*lock == 1) ; // spin just reading
  if (test&set(lock) == 0)  break;
}
```

Busy-wait on cached copy

```
Lock::Release() {
   *lock = 0;
}
```

```
Lock::Release() {
*lock = 0;
}
```

- What is the problem with this?
  - A. CPU usage B. Memory usage C. Lock::Acquire() latency
  - D. Memory bus usage E. Does not work

Implementing Locks: Summary

- Locks are higher-level programming abstraction
  - Mutual exclusion can be implemented using locks
- Lock implementation generally requires some level of hardware support
  - Details of hardware support affects efficiency of locking
- Locks can busy-wait, and busy-waiting cheaply is important
  - Soon come primitives that block rather than busy-wait

Hardware-Supported Atomic Read-Modify-Write Instructions

- `test&set`: set content of "address" to 1, and return its original content

```
test&set (&address)
  result = M[address];
  M[address] = 1;
  return result;
}
```

- `compare&swap`: compare content of "address" to reg1; if same, set it to reg2

```
compare&swap (&address, reg1, reg2) { /*
   68000 */
     if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
     } else {
        return failure;
     }
```

Implementing Locks with test&set

- A simple solution:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); // while busy
}
Release() {
   value = 0;
}
```

- Explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock
- Busy-Waiting: thread consumes cycles while waiting

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Interrupts are not disabled
  - User code can use this lock
  - Works on a multiprocessor

- Negatives
  - Inefficient, because the busy-waiting thread will consume cycles waiting idly
  - Waiting thread may take cycles away from thread holding lock
    - Priority Inversion: For priority-based scheduling, if busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
    - Round-robin scheduling is OK

Higher-Level Primitives than Locks

- Good primitives and practices important!
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Semaphores and Monitors next

Summary Cont'

- Semaphores: Like integers with restricted interface
  - Two operations:
    - `P()`: Wait if zero; decrement when becomes non-zero
    - `V()`: Increment and wake a sleeping task (if exists)
    - Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- Readers/Writers
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- Language support for synchronization:
  - Java provides synchronized keyword and one condition-