

Mutual Exclusion

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

Mutual Exclusion

- Today we will try to formalize our understanding of mutual exclusion
- We will also use the opportunity to show you how to argue about and prove various properties in an asynchronous concurrent setting

Mutual Exclusion

- Formal problem definitions
- Solutions for 2 threads
- Solutions for n threads
- Fair solutions
- Inherent costs

Warning

- You will never use these protocols
 - Get over it
- You are advised to understand them
 - The same issues show up everywhere
 - Except hidden and more complex

Why is Concurrent Programming so Hard?

- Try preparing a seven-course banquet
 - By yourself
 - With one friend
 - With twenty-seven friends ...
- Before we can talk about programs
 - Need a language
 - Describing time and concurrency

Time

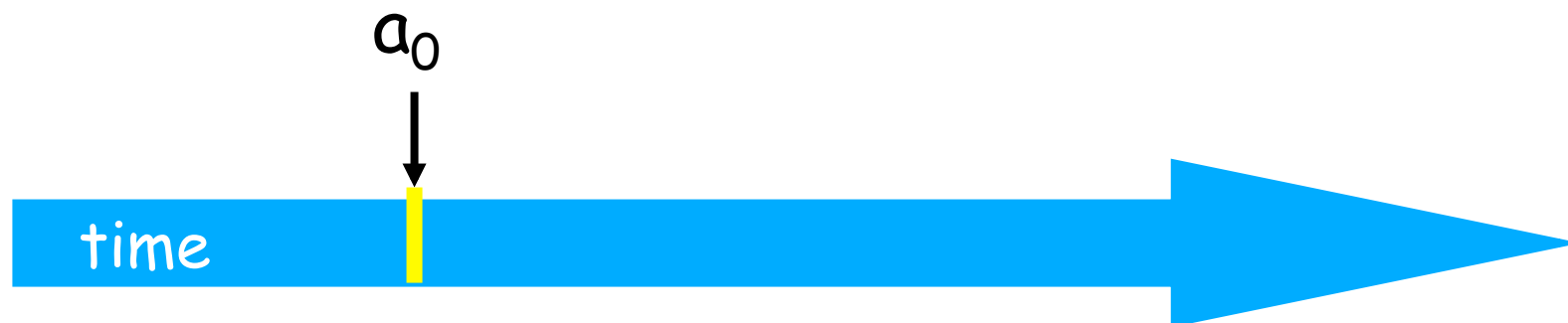
- "Absolute, true and mathematical time, of itself and from its own nature, flows equably without relation to anything external." (I. Newton, 1689)
- "Time is, like, Nature's way of making sure that everything doesn't happen all at once." (Anonymous, circa 1968)



time

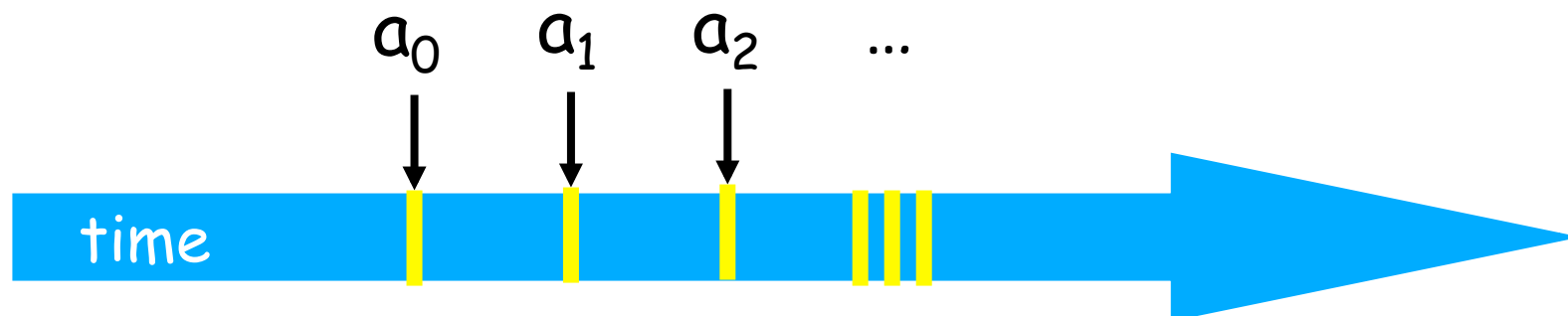
Events

- An *event* a_0 of thread A is
 - Instantaneous
 - No simultaneous events (break ties)



Threads

- A *thread* A is (formally) a sequence a_0, a_1, \dots of events
 - "Trace" model
 - Notation: $a_0 \rightarrow a_1$ indicates order

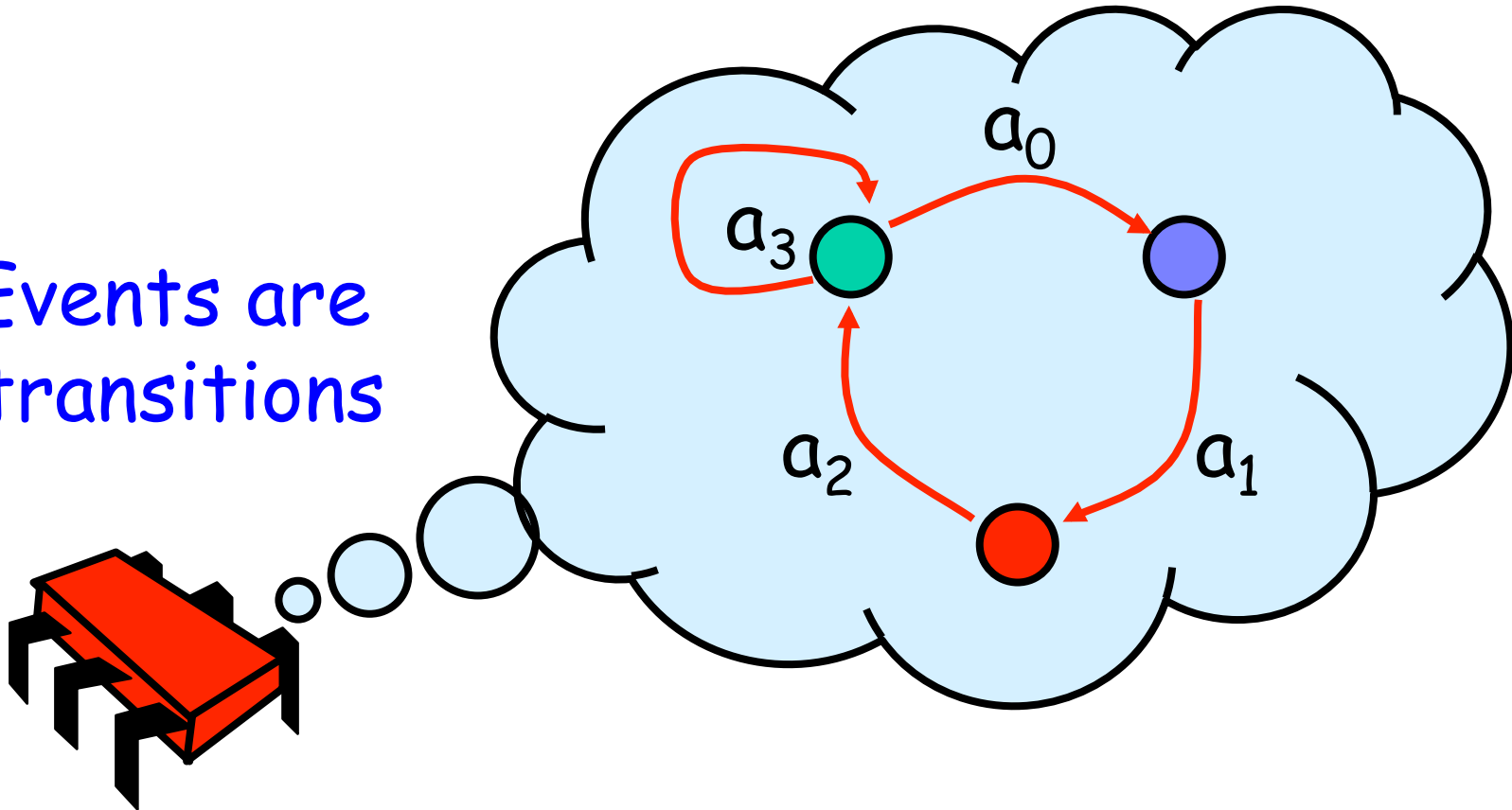


Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things ...

Threads are State Machines

Events are
transitions



States

- Thread State
 - Program counter
 - Local variables
- System state
 - Object fields (shared variables)
 - Union of thread states

Concurrency

- Thread A



Concurrency

- Thread A



- Thread B



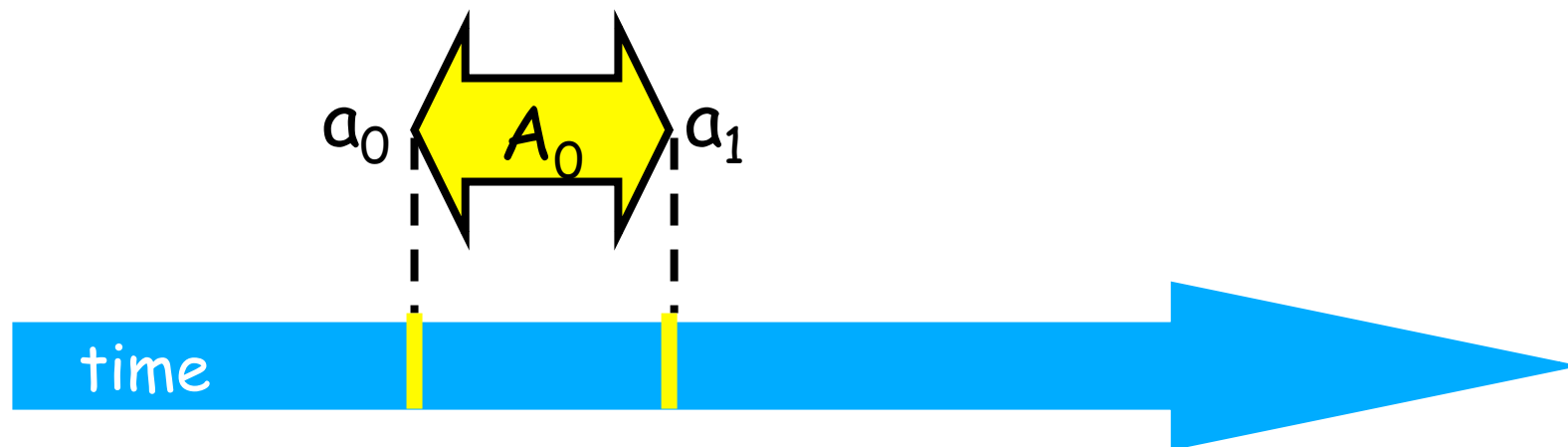
Interleavings

- Events of two or more threads
 - Interleaved
 - Not necessarily independent (why?)

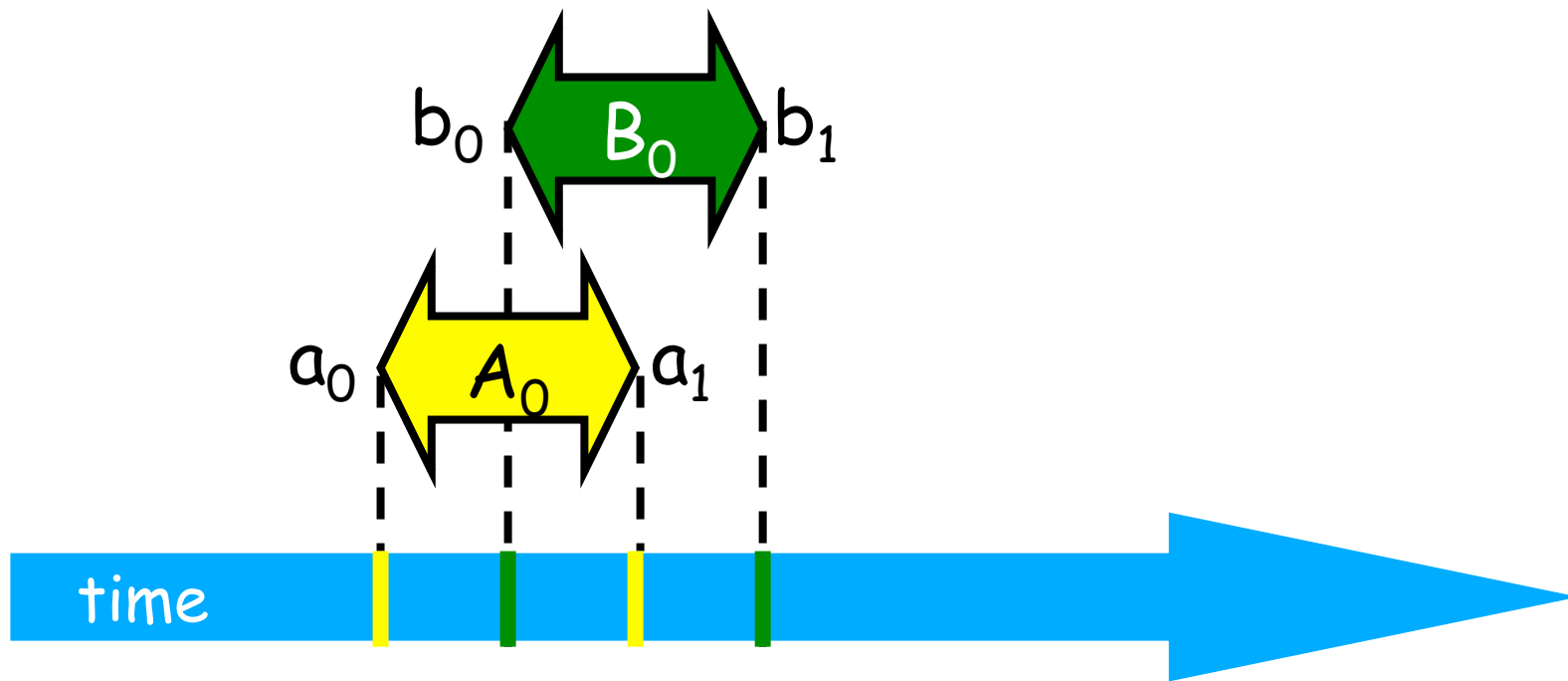


Intervals

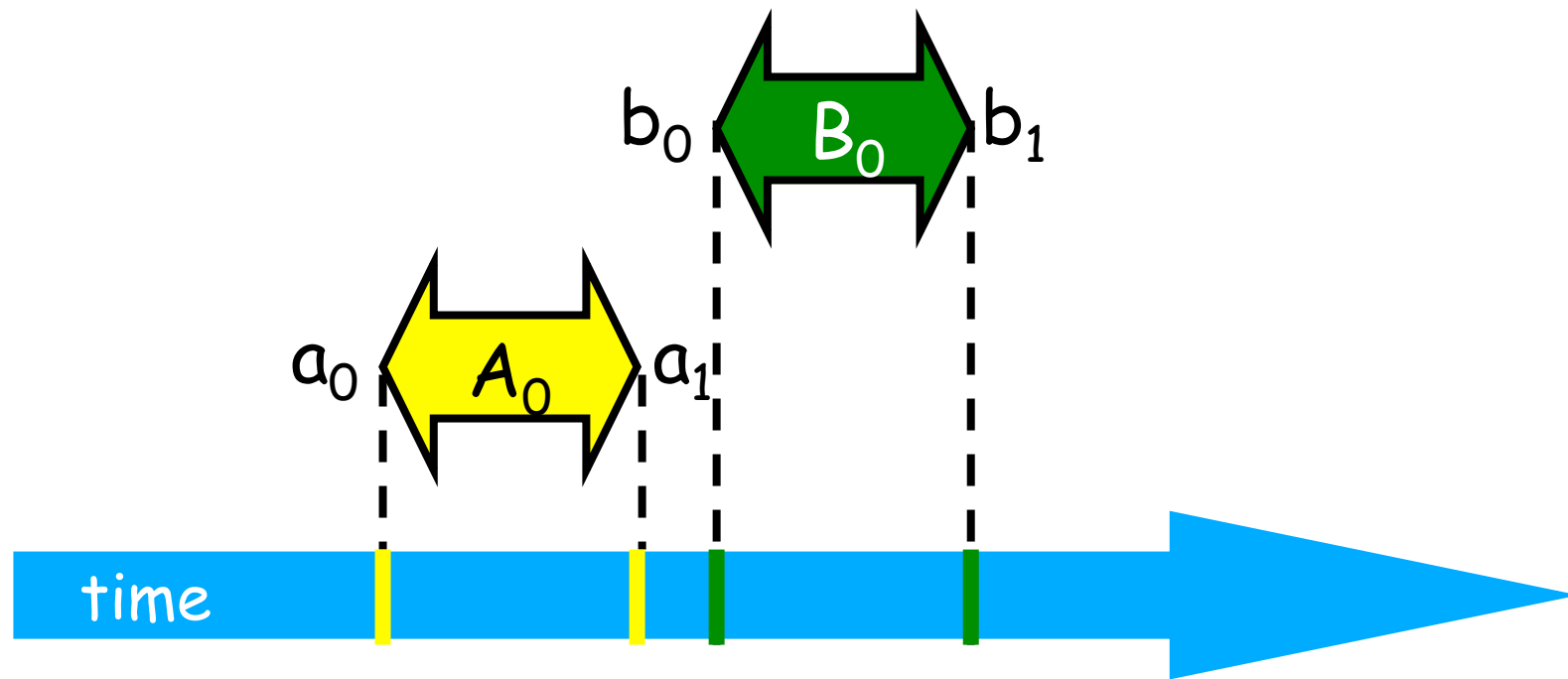
- An *interval* $A_0 = (a_0, a_1)$ is
 - Time between events a_0 and a_1



Intervals may Overlap

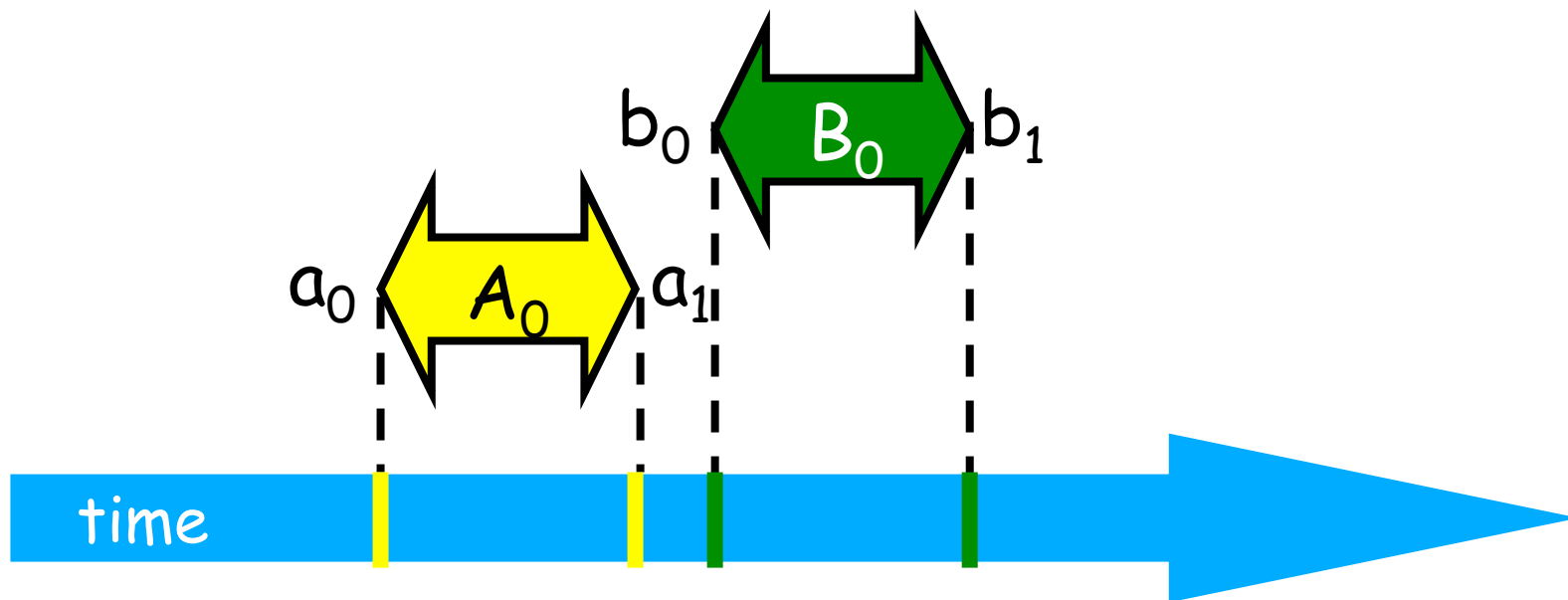


Intervals may be Disjoint

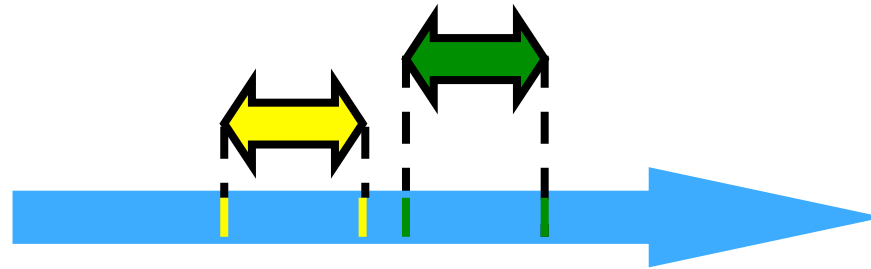


Precedence

Interval A_0 precedes interval B_0

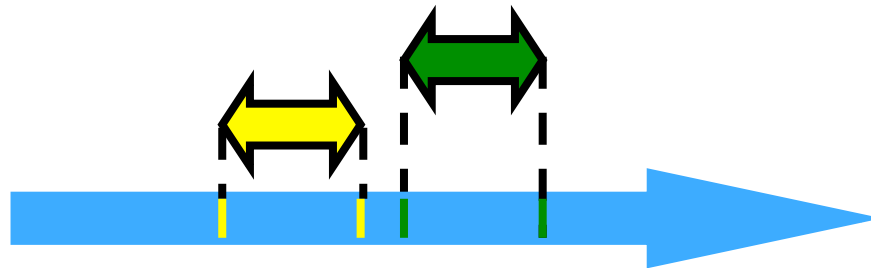


Precedence



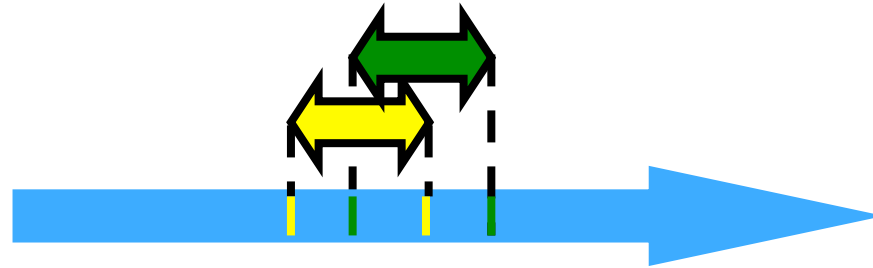
- Notation: $A_0 \rightarrow B_0$
- Formally,
 - End event of A_0 before start event of B_0
 - Also called "happens before" or "precedes"

Precedence Ordering



- Remark: $A_0 \rightarrow B_0$ is just like saying
 - 1066 AD \rightarrow 1492 AD,
 - Middle Ages \rightarrow Renaissance,
- Oh wait,
 - what about this week *vs* this month?

Precedence Ordering



- Never true that $A \rightarrow A$
- If $A \rightarrow B$ then not true that $B \rightarrow A$
- If $A \rightarrow B$ & $B \rightarrow C$ then $A \rightarrow C$
- Funny thing: $A \rightarrow B$ & $B \rightarrow A$ might both be false!

Partial Orders

(you may know this already)

- Irreflexive:
 - Never true that $A \rightarrow A$
- Antisymmetric:
 - If $A \rightarrow B$ then not true that $B \rightarrow A$
- Transitive:
 - If $A \rightarrow B$ & $B \rightarrow C$ then $A \rightarrow C$

Total Orders

(you may know this already)

- Also
 - Irreflexive
 - Antisymmetric
 - Transitive
- Except that for every distinct A, B ,
 - Either $A \rightarrow B$ or $B \rightarrow A$

Repeated Events

```
while (mumble) {  
    a0; a1;  
}
```

k -th occurrence
of event a_0

a_0^k

k -th occurrence of
interval $A_0 = (a_0, a_1)$

A_0^k

Implementing a Counter

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

*Make these steps
indivisible using
locks*

Locks (Mutual Exclusion)

```
public interface Lock {  
    public void lock();  
    public void unlock();  
}
```

Locks (Mutual Exclusion)

```
public interface Lock {
```

```
    public void lock();
```

acquire lock

```
    public void unlock();  
}
```

Locks (Mutual Exclusion)

```
public interface Lock {
```

```
    public void lock();
```

acquire lock

```
    public void unlock();
```

release lock

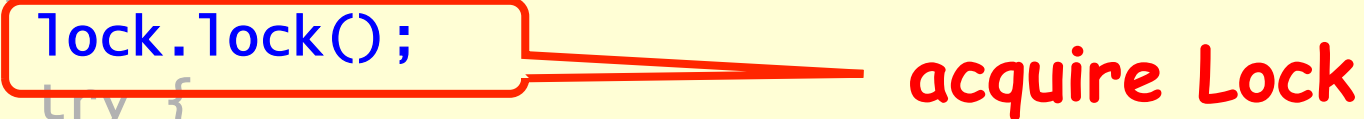
```
}
```

Using Locks

```
public class Counter {  
    private long value;  
    private Lock lock;  
    public long getAndIncrement() {  
        lock.lock();  
        try {  
            int temp = value;  
            value = value + 1;  
        } finally {  
            lock.unlock();  
        }  
        return temp;  
    }  
}
```

Using Locks

```
public class Counter {  
    private long value;  
    private Lock lock;  
    public long getAndIncrement() {  
        lock.lock();  
        try {  
            int temp = value;  
            value = value + 1;  
        } finally {  
            lock.unlock();  
        }  
        return temp;  
    }  
}
```



acquire Lock

Using Locks

```
public class Counter {  
    private long value;  
    private Lock lock;  
    public long getAndIncrement() {  
        lock.lock();  
        try {  
            int temp = value;  
            value = value + 1;  
        } finally {  
            lock.unlock();  
        }  
        return temp;  
    }  
}
```

Release lock
(no matter what)

Using Locks



```
public class Counter {  
    private long value;  
    private Lock lock;  
    public long getAndIncrement() {  
        lock.lock();  
        try {  
            int temp = value;  
            value = value + 1;  
        } finally {  
            lock.unlock();  
        }  
        return temp;  
    }  
}
```

Critical
section







Mutual Exclusion

- Let CS_i^k  be thread i's k-th critical section execution



Mutual Exclusion

- Let CS_i^k  be thread i's k-th critical section execution
- And CS_j^m  be thread j's m-th critical section execution

Mutual Exclusion

- Let CS_i^k  be thread i's k-th critical section execution
- And CS_j^m  be j's m-th execution
- Then either
 -   or  



Mutual Exclusion

- Let CS_i^k  be thread i's k-th critical section execution
- And CS_j^m  be j's m-th execution
- Then either

-   or  

$CS_i^k \rightarrow CS_j^m$

Mutual Exclusion

- Let CS_i^k  be thread i's k-th critical section execution
- And CS_j^m  be j's m-th execution
- Then either

-   or  

$CS_i^k \rightarrow CS_j^m$

$CS_j^m \rightarrow CS_i^k$

Deadlock-Free

- If some thread calls **lock()**
 - And never returns
 - Then other threads must complete **lock()** and **unlock()** calls infinitely often
- System as a whole makes progress
 - Even if individuals starve

Starvation-Free

- If some thread calls `lock()`
 - It will eventually return
- Individual threads make progress

Two-Thread vs n -Thread Solutions

- Two-thread solutions first
 - Illustrate most basic ideas
 - Fits on one slide
- Then n -Thread solutions

Two-Thread Conventions

```
class ... implements Lock {  
    ...  
    // thread-local index, 0 or 1  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
    ...  
    }  
}
```

Two-Thread Conventions

```
class ... implements Lock {  
    ...  
    // thread-local index, 0 or 1  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        ...  
    }  
}
```

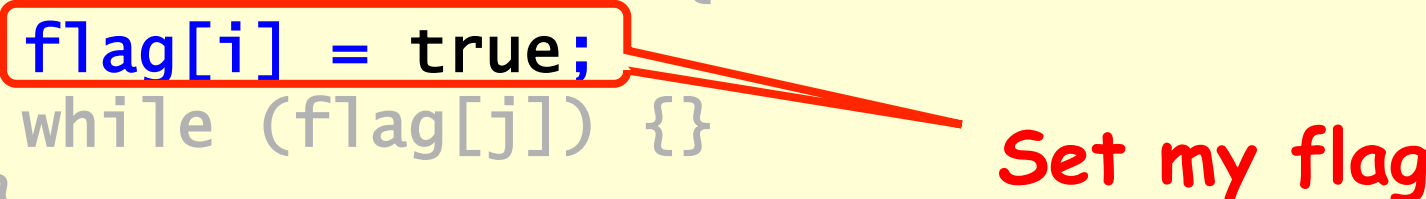
Henceforth: *i* is current thread, *j* is other thread

LockOne

```
class LockOne implements Lock {  
    private volatile boolean[] flag =  
        new boolean[2];  
  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

LockOne

```
class LockOne implements Lock {  
    private volatile boolean[] flag =  
        new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```



Set my flag

LockOne

```
class LockOne implements Lock {  
    private volatile boolean[] flag =  
        new boolean[2];  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

Set my flag

Wait for other
flag to go false

LockOne Satisfies Mutual Exclusion

- Assume CS_A^j overlaps CS_B^k
- Consider each thread's last (j-th and k-th) read and write in the lock() method before entering
- Derive a contradiction

From the Code

- $\text{write}_A(\text{flag}[A]=\text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false}) \rightarrow CS_A$
- $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false}) \rightarrow CS_B$

```
class LockOne implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        while (flag[j]) {}  
    }  
}
```

From the Assumption

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$
- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

Combining

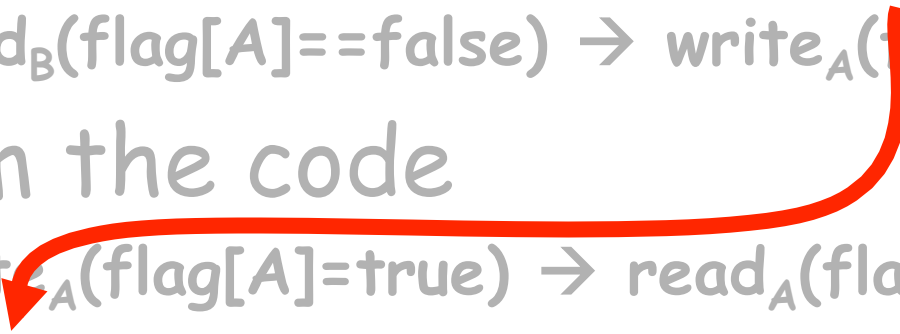
- Assumptions:

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$
- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$
- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

- Assumptions:
 - $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$
 - $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$
 - From the code
 - $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$
 - $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$
- 

Combining

- Assumptions:

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

- Assumptions:

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

- Assumptions.

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Combining

- Assumptions.

- $\text{read}_A(\text{flag}[B] == \text{false}) \rightarrow \text{write}_B(\text{flag}[B] = \text{true})$

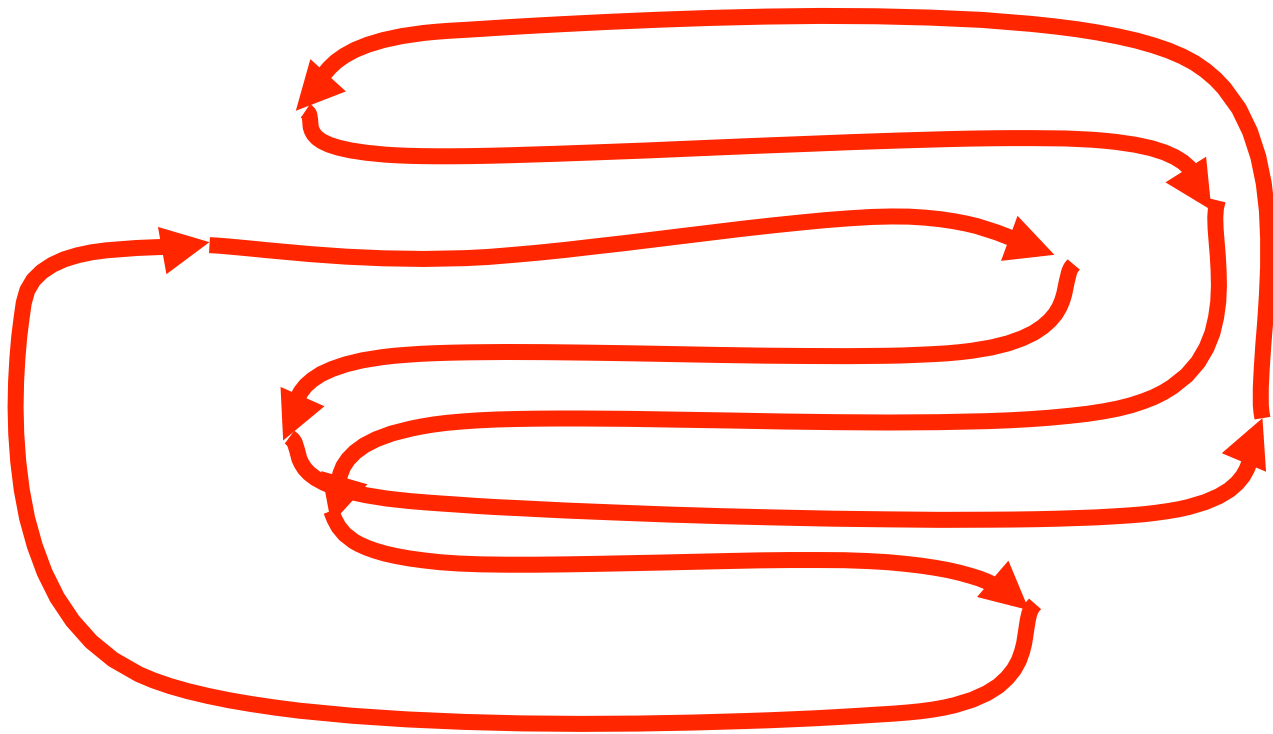
- $\text{read}_B(\text{flag}[A] == \text{false}) \rightarrow \text{write}_A(\text{flag}[A] = \text{true})$

- From the code

- $\text{write}_A(\text{flag}[A] = \text{true}) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})$

- $\text{write}_B(\text{flag}[B] = \text{true}) \rightarrow \text{read}_B(\text{flag}[A] == \text{false})$

Cycle!



Deadlock Freedom

- LockOne Fails deadlock-freedom
 - Concurrent execution can deadlock

```
flag[i] = true;    flag[j] = true;
while (flag[j]){   while (flag[i]){
```

- Sequential executions OK

LockTwo

```
public class LockTwo implements Lock {  
    private volatile int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

LockTwo

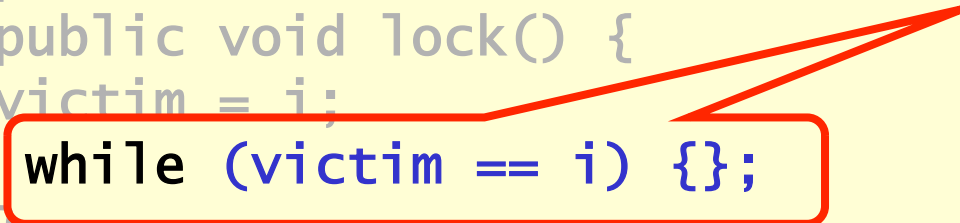
```
public class LockTwo implements Lock {  
    private volatile int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

Let other go first

LockTwo

```
public class LockTwo implements Lock {  
    private volatile int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
  
    public void unlock() {}  
}
```

Wait for permission



LockTwo

```
public class Lock2 implements Lock {  
    private volatile int victim;  
    public void lock() {  
        victim = i;  
        while (victim == i) {};  
    }  
    public void unlock() {}  
}
```

Nothing to do



LockTwo Claims

- Satisfies mutual exclusion
 - If thread *i* in CS
 - Then `victim == j`
 - Cannot be both 0 and 1
- Not deadlock free
 - Sequential execution deadlocks
 - Concurrent execution does not

```
public void LockTwo() {  
    victim = i;  
    while (victim == i) {}  
}
```

Peterson's Algorithm

```
1  public void lock() {  
2      flag[i] = true;  
3      victim = i;  
4      while (flag[j] && victim == i) {};  
5  }  
6  public void unlock() {  
7      flag[i] = false;  
8  }
```

Peterson's Algorithm

Announce I'm
interested

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

Peterson's Algorithm

```
public void lock() {
```

```
    flag[i] = true;
```

```
    victim = i;
```

```
    while (flag[j] && victim == i) {};
```

```
    public void unlock() {
```

```
        flag[i] = false;
```

```
    }
```

Announce I'm
interested

Defer to other

Peterson's Algorithm

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

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

Peterson's Algorithm

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

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

No longer interested

Mutual Exclusion

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

- If thread **0** in critical section,
 - flag[0]=true,
 - victim = 1
- If thread **1** in critical section,
 - flag[1]=true,
 - victim = 0

Cannot both be true

Deadlock Free

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

- Thread blocked
 - only at **while** loop
 - only if it is the victim
- One or the other must not be the victim

Starvation Free

- Thread *i* blocked only if *j* repeatedly re-enters so that

`flag[j] == true` and
`victim == i`

- When *j* re-enters
 - it sets `victim` to *j*.
 - So *i* gets in

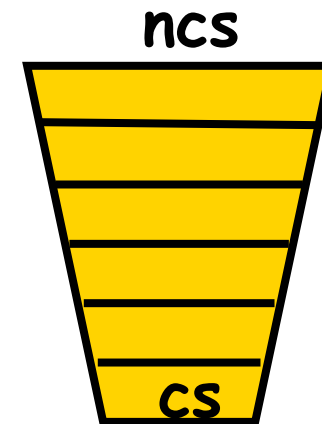
```
public void lock() {  
    flag[i] = true;  
    victim  = i;  
    while (flag[j] && victim == i) {};  
}
```

```
public void unlock() {  
    flag[i] = false;  
}
```

The Filter Algorithm for n Threads

There are $n-1$ "waiting rooms" called levels

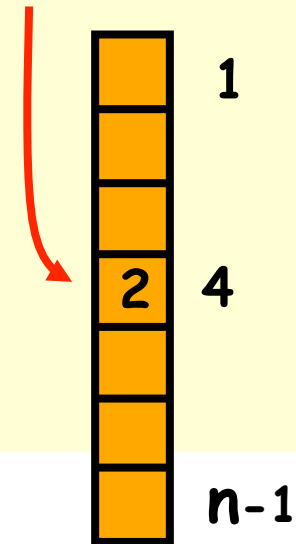
- At each level
 - At least one enters level
 - At least one blocked if many try
- Only one thread makes it through



Filter

```
class Filter implements Lock {
    volatile int[] level; // level[i] for thread i
    volatile int[] victim; // victim[L] for level L
```

```
    public Filter(int n) {
        level = new int[n];
        victim = new int[n];
        for (int i = 1; i < n; i++) {
            level[i] = 0;
        }
        ...
    }
```



Thread 2 at level 4

Filter

```
class Filter implements Lock {  
    ...  
  
    public void lock(){  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists$  k != i level[k] >= L) &&  
                victim[L] == i );  
        }  
    }  
    public void unlock() {  
        level[i] = 0;  
    }  
}
```


Filter

```
class Filter implements Lock {  
    ...  
  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists$  k != i) level[k] >= L) &&  
                victim[L] == i),  
        }  
    }  
    public void release(int i) {  
        level[i] = 0;  
    }  
}
```

One level at a time

Filter

```
class Filter implements Lock {  
    ...  
  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = 1;  
            while (( $\exists k \neq i$ ) level[k] >= L) &&  
                victim[L] == 1)  
            {}  
        }  
    }  
    public void release(int i)  
    {  
        level[i] = 0;  
    }  
}
```

Announce
intention to
enter level L

Filter

```
class Filter implements Lock {  
    int level[n];  
    int victim[n];  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists k \neq i$ ) level[k] >= L) &&  
                victim[L] == i);  
        }  
    }  
    public void release(int i)  
        level[i] = 0;  
}
```

*Give priority to
anyone but me*

Filter

Wait as long as someone else is at same or higher level, and I'm designated victim

```
public void lock() {  
    for (int L = 1; L < n; L++) {  
        level[i] = L;  
        victim[L] = i;  
        while (( $\exists k \neq i$ ) level[k] >= L) &&  
            victim[L] == i);  
    }  
}  
public void release(int i) {  
    level[i] = 0;  
}
```

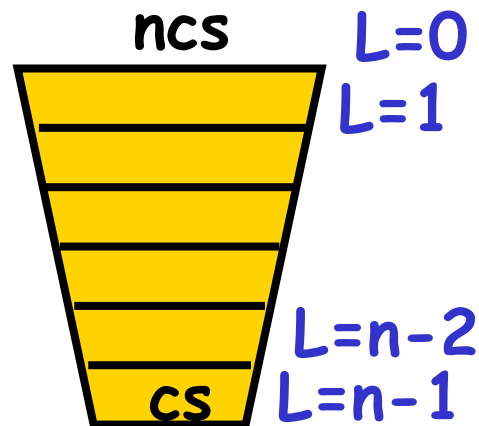
Filter

```
class Filter implements Lock {  
    int level[n];  
    int victim[n];  
    public void lock() {  
        for (int L = 1; L < n; L++) {  
            level[i] = L;  
            victim[L] = i;  
            while (( $\exists$  k != i) level[k] >= L) &&  
                victim[L] == i);  
        }  
    }  
}
```

Thread enters level L when it completes the loop

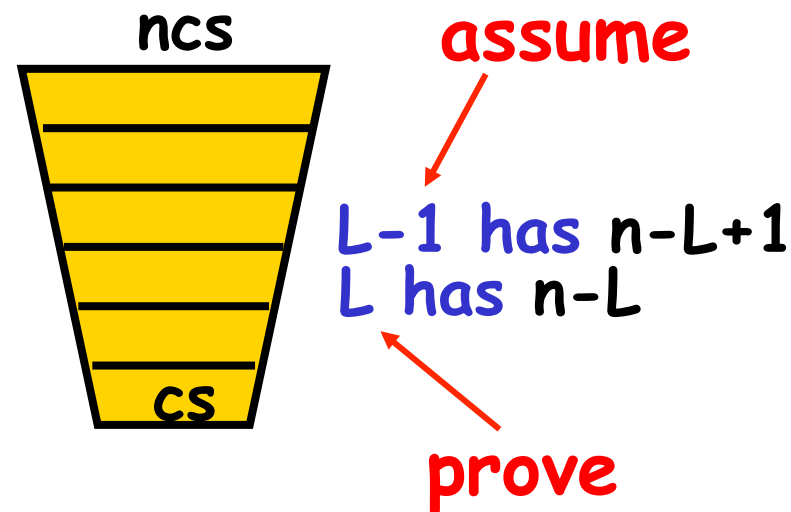
Claim

- Start at level $L=0$
- At most $n-L$ threads enter level L
- Mutual exclusion at level $L=n-1$

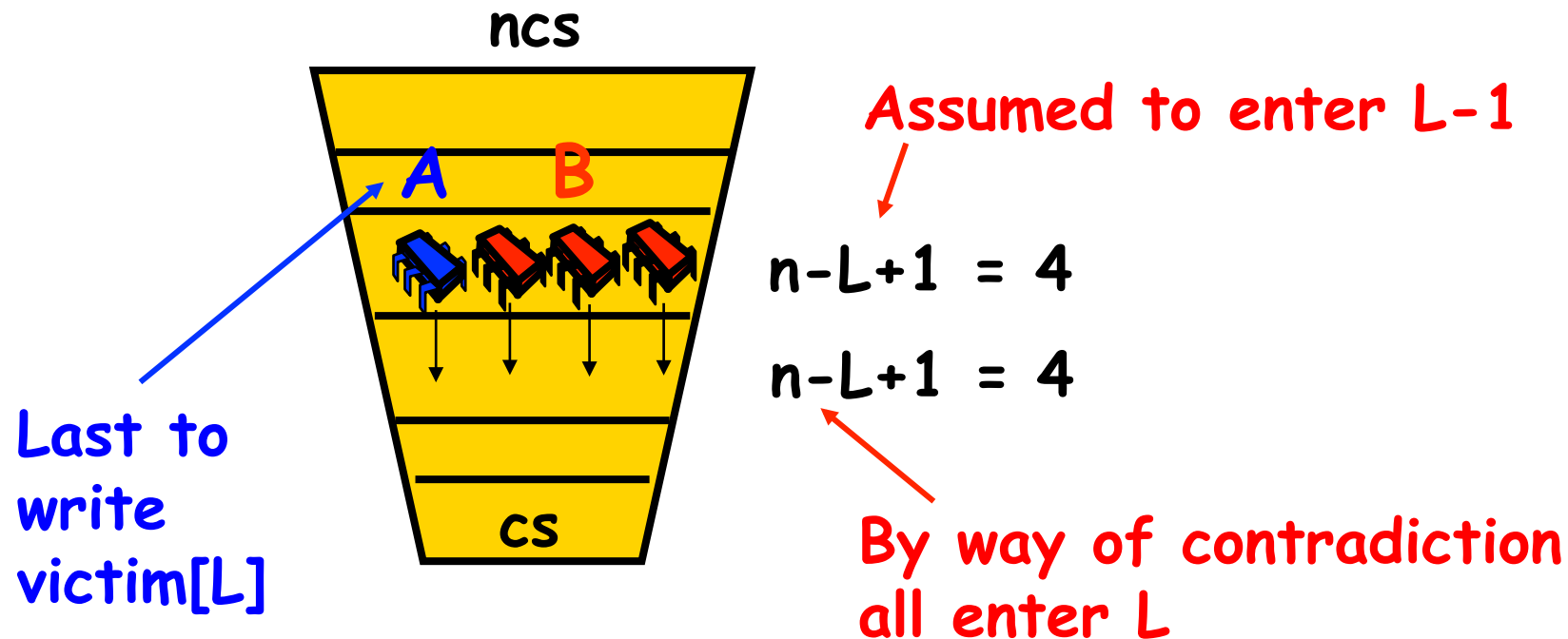


Induction Hypothesis

- No more than $n-L+1$ at level $L-1$
- Induction step: by contradiction
- Assume all at level $L-1$ enter level L
- A last to write `victim[L]`
- B is any other thread at level L



Proof Structure



Show that A must have seen B at level L and since $\text{victim}[L] == A$ could not have entered

From the Code

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B)$

```
public void lock() {  
    for (int l = 1; l < n; l++) {  
        level[l] = l;  
        victim[l] = i;  
        while (( $\exists k \neq i$ ) level[k] >= l  
                && victim[l] == i) {};  
    }  
}
```

From the Code

(2) $\text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B])$

```
public void lock() {  
    for (int L = 1; L < n; L++) {  
        level[i] = L;  
        victim[L] = i;  
        while (( $\exists k \neq i$ ) level[k] >= L)  
            && victim[L] == i) {},  
    }  
}
```

By Assumption

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

By assumption, A is the last
thread to write $\text{victim}[L]$

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow \text{write}_B(\text{victim}[L]=B)$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2) $\text{write}_A(\text{victim}[L]=A) \rightarrow \text{read}_A(\text{level}[B])$

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2) $\rightarrow \text{read}_A(\text{level}[B])$

```
public void lock() {  
    for (int L = 1; L < n; L++) {  
        level[i] = L;  
        victim[L] = i;  
        while (( $\exists k \neq i$ ) level[k] >= L  
                && victim[L] == i) {};  
    }  
}
```

Combining Observations

(1) $\text{write}_B(\text{level}[B]=L) \rightarrow$

(3) $\text{write}_B(\text{victim}[L]=B) \rightarrow \text{write}_A(\text{victim}[L]=A)$

(2) $\rightarrow \text{read}_A(\text{level}[B])$

Thus, A read $\text{level}[B] \geq L$,
A was last to write $\text{victim}[L]$,
so it could not have entered level L!

No Starvation

- Filter Lock satisfies properties:
 - Just like Peterson Alg at any level
 - So no one starves
- But what about fairness?
 - Threads can be overtaken by others

Bounded Waiting

- Want stronger fairness guarantees
- Thread not "overtaken" too much
- Need to adjust definitions

Bounded Waiting

- Divide `lock()` method into 2 parts:
 - Doorway interval:
 - Written D_A
 - always finishes in finite steps
 - Waiting interval:
 - Written W_A
 - may take unbounded steps

Bakery Algorithm

- Provides First-Come-First-Served
- How?
 - Take a "number"
 - Wait until lower numbers have been served
- Lexicographic order
 - $(a,i) > (b,j)$
 - If $a > b$, or $a = b$ and $i > j$

Bakery Algorithm

```
class Bakery implements Lock {  
    volatile boolean[] flag;  
    volatile Label[] label;  
    public Bakery (int n) {  
        flag = new boolean[n];  
        label = new Label[n];  
        for (int i = 0; i < n; i++) {  
            flag[i] = false; label[i] = 0;  
        }  
    }  
    ...  
}
```

Bakery Algorithm

```
class Bakery implements Lock {
```

```
    volatile boolean[] flag;
```

```
    volatile Label[] label;
```

```
    public Bakery (int n) {
```

```
        flag = new boolean[n];
```

```
        label = new Label[n];
```

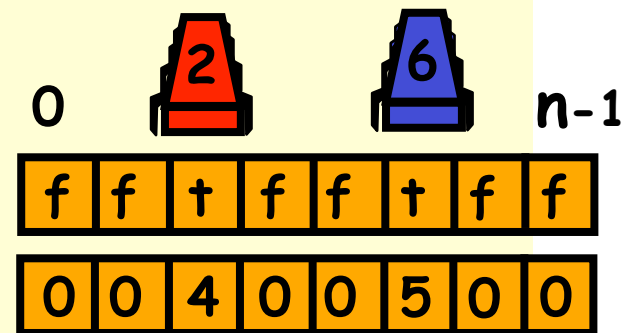
```
        for (int i = 0; i < n; i++) {
```

```
            flag[i] = false; label[i] = 0;
```

```
        }
```

```
    }
```

```
    ...
```



CS

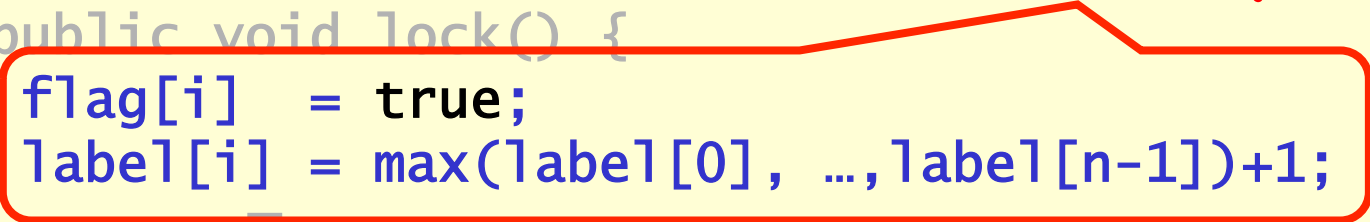
Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i],i) > (label[k],k));  
    }
```

Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }  
}
```

Doorway



Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }  
}
```

I'm interested

Bakery Algorithm

Take increasing
label (read
labels in some
arbitrary order)

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }  
}
```


Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) > (label[k], k));  
    }  
}
```

Someone is interested

Bakery Algorithm

```
class Bakery implements Lock {  
    boolean flag[n];  
    int label[n];  
  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i],i) > (label[k],k));  
    }
```

Someone is
interested

With lower (label,i) in
lexicographic order

Bakery Algorithm

```
class Bakery implements Lock {  
    ...  
    public void unlock() {  
        flag[i] = false;  
    }  
}
```

Bakery Algorithm

```
class Bakery implements Lock {
```

```
...
```

```
public void unlock() {
```

```
    flag[i] = false;
```

```
}
```

No longer
interested

labels are always increasing

No Deadlock

- There is always one thread with earliest label
- Ties are impossible (why?)

First-Come-First-Served

- If $D_A \rightarrow D_B$ then A's label is earlier
 - $\text{write}_A(\text{label}[A]) \rightarrow$
 $\text{read}_B(\text{label}[A]) \rightarrow$
 $\text{write}_B(\text{label}[B]) \rightarrow$
 $\text{read}_B(\text{flag}[A])$
- So B is locked out while $\text{flag}[A]$ is true

```
class Bakery implements Lock {  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0],  
                        ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) >  
                (label[k], k));  
    }  
}
```

Mutual Exclusion

- Suppose *A* and *B* in CS together
- Suppose *A* has earlier label
- When *B* entered, it must have seen
 - $\text{flag}[A]$ is false, or
 - $\text{label}[A] > \text{label}[B]$

```
class Bakery implements Lock {  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0],  
                        ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i], i) >  
                (label[k], k));  
    }  
}
```

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen `flag[A] == false`

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen $\text{flag}[A] == \text{false}$
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$

Mutual Exclusion

- Labels are strictly increasing so
- B must have seen $\text{flag}[A] == \text{false}$
- $\text{Labeling}_B \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{write}_A(\text{flag}[A]) \rightarrow \text{Labeling}_A$
- Which contradicts the assumption that A has an earlier label

Bakery Y2³²K Bug

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while ( $\exists k$  flag[k]  
                && (label[i],i) > (label[k],k));  
    }
```

Bakery Y2³²K Bug

```
class Bakery implements Lock {  
    ...  
    public void lock() {  
        flag[i] = true;  
        label[i] = max(label[0], ..., label[n-1])+1;  
        while (∃k flag[k]  
                && (label[i],i) > (label[k],k));  
    }  
}
```

**Mutex breaks if
label[i]
overflows**

Does Overflow Actually Matter?

- Yes
 - Y2K
 - 18 January 2038 (Unix `time_t` rollover)
 - 16-bit counters
- No
 - 64-bit counters
- Maybe
 - 32-bit counters

... spin locks e desempenho

- material cap 7 livro Herlihy

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?

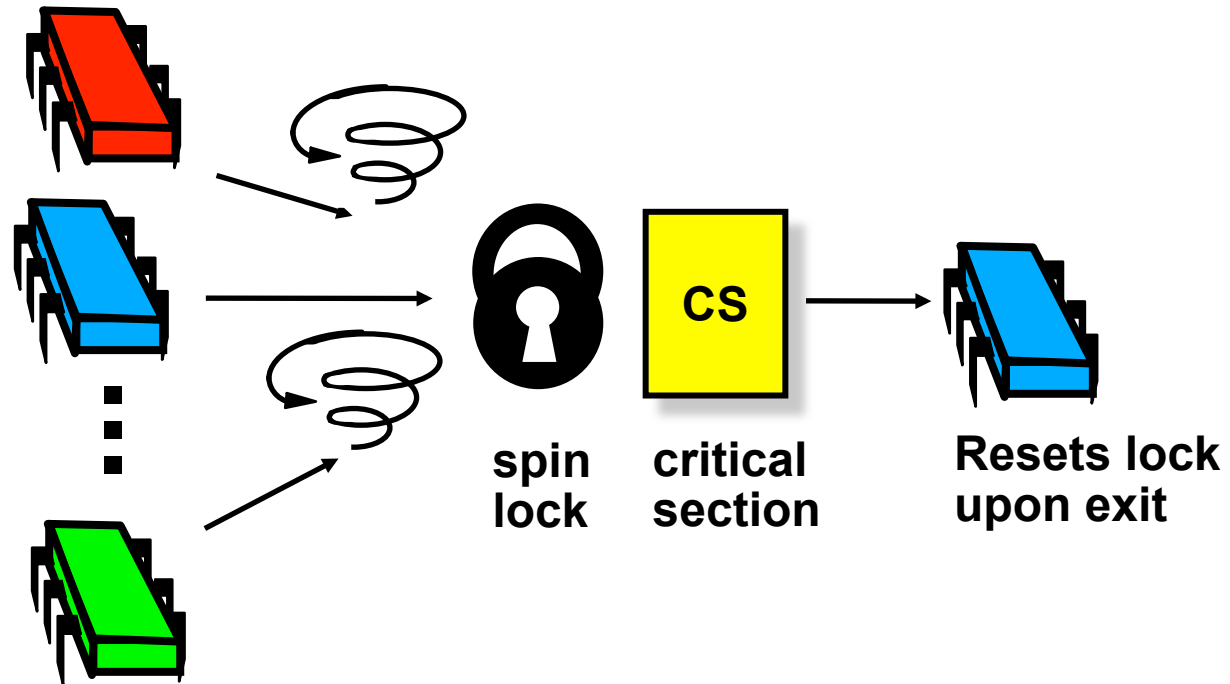
- Keep trying
 - "spin" or "busy-wait"
 - Good if delays are short
- Give up the processor
 - Good if delays are long
 - Always good on uniprocessor

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

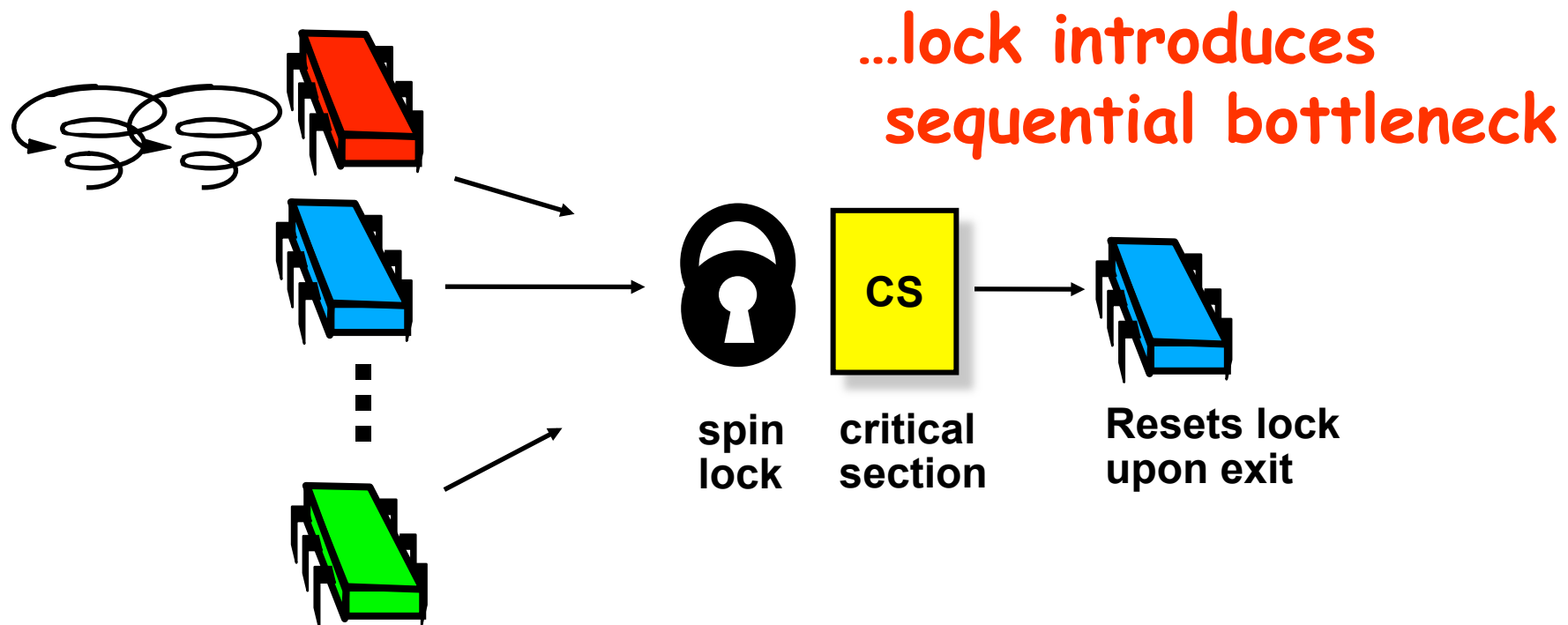
- Keep trying
 - "spin" or "busy-wait"
 - Good if delays are short
- Give up the processor
 - Good if delays are long
 - Always good on uniprocessor

our focus

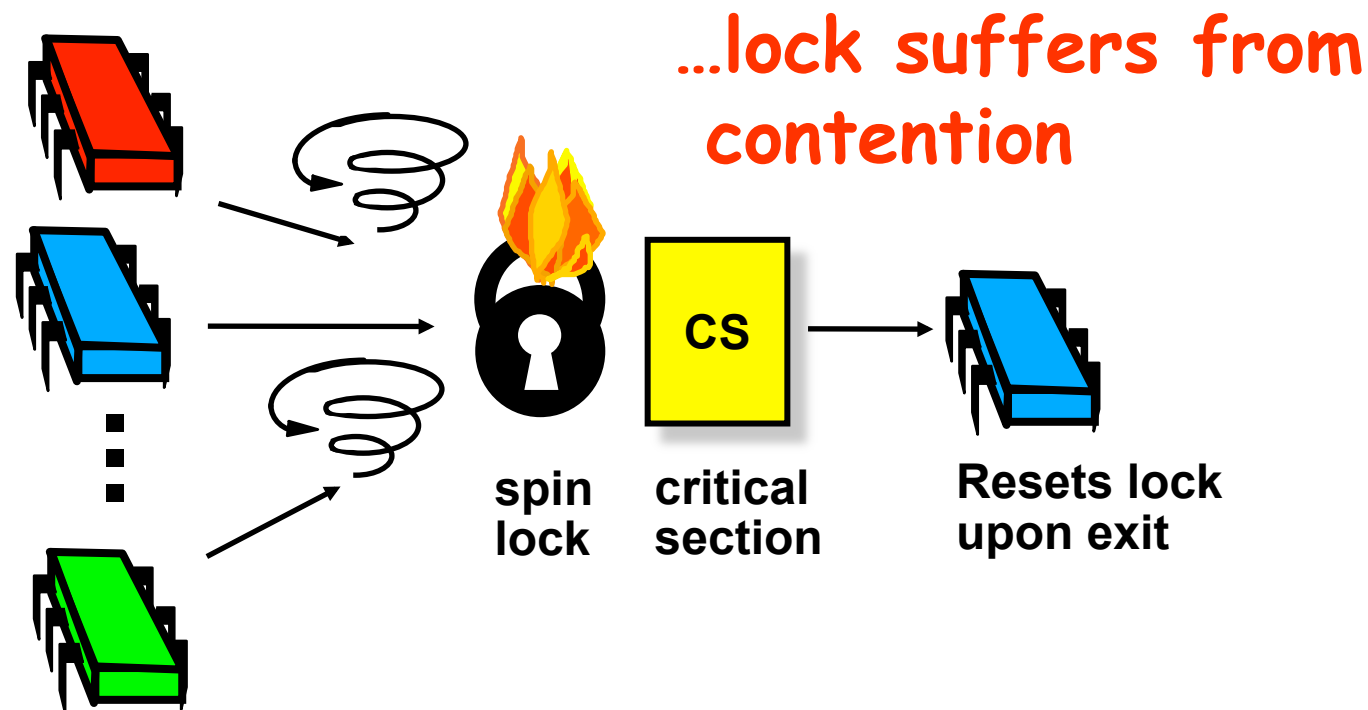
Basic Spin-Lock



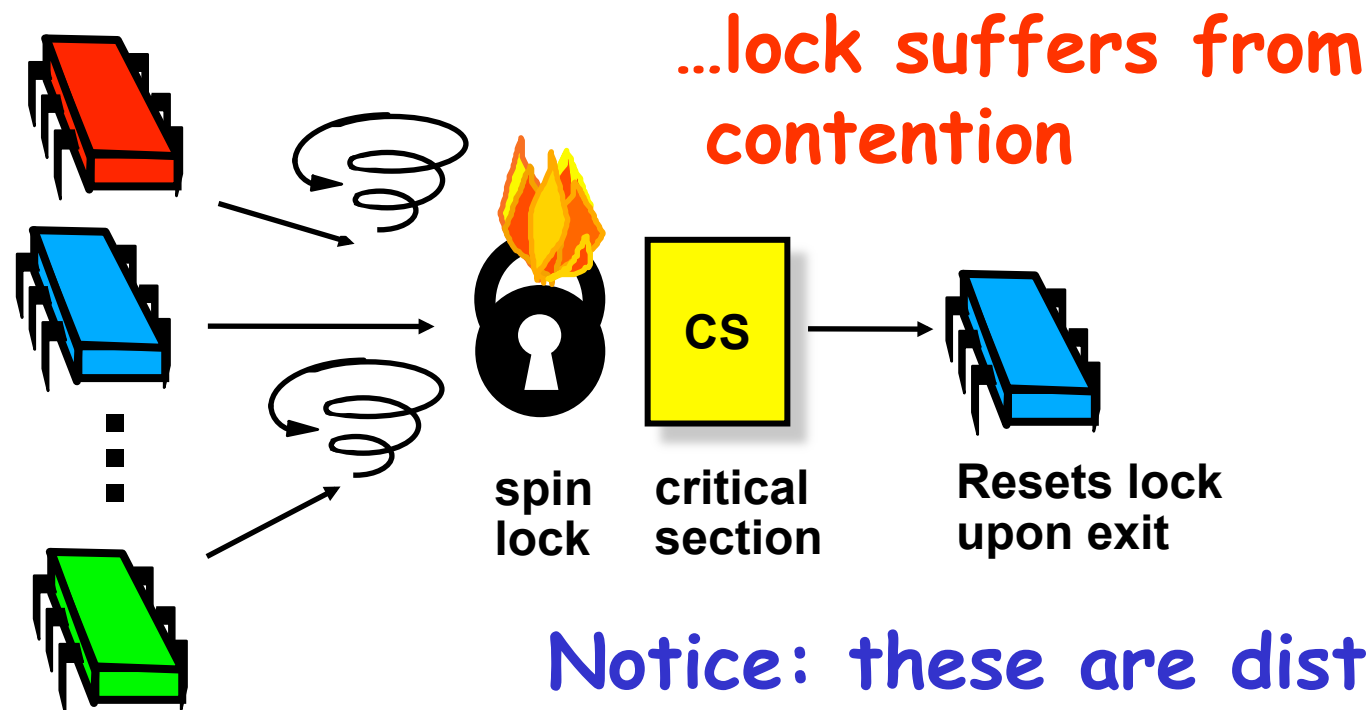
Basic Spin-Lock



Basic Spin-Lock

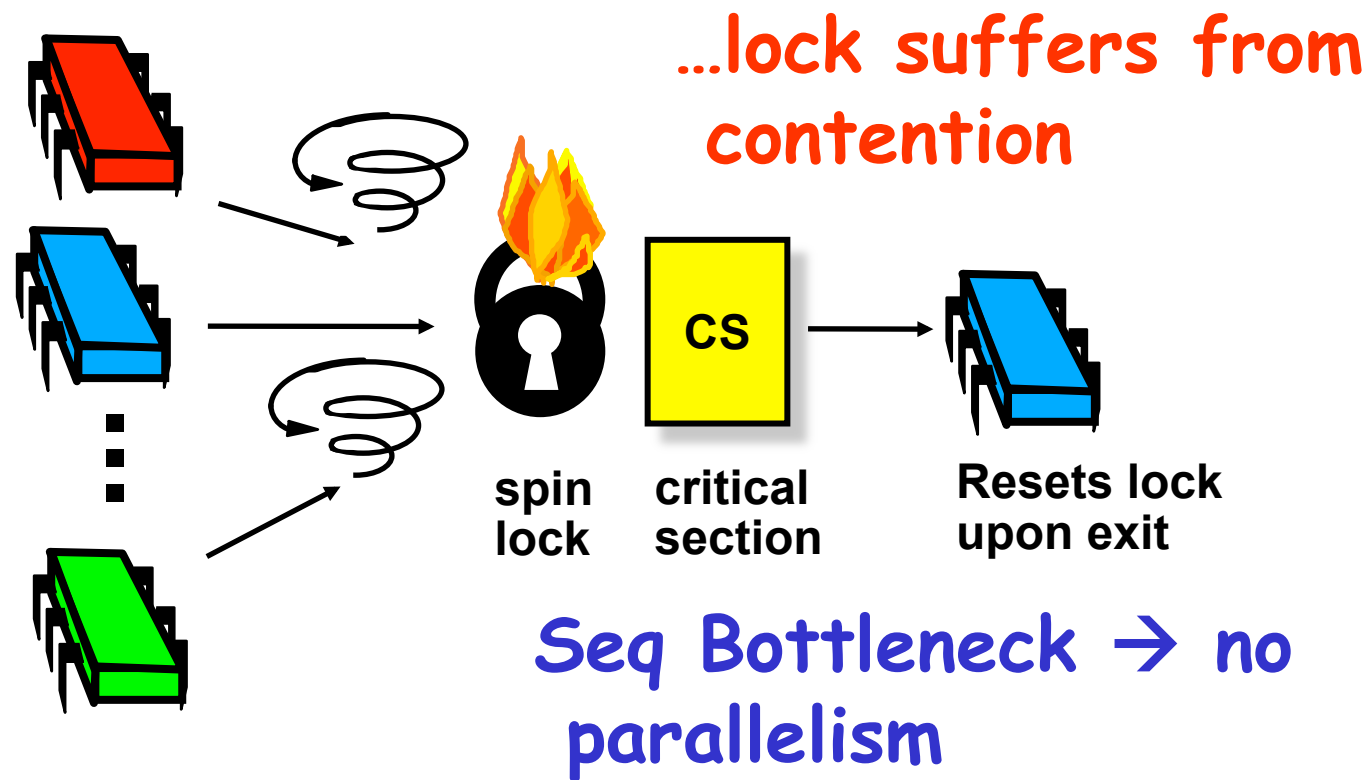


Basic Spin-Lock

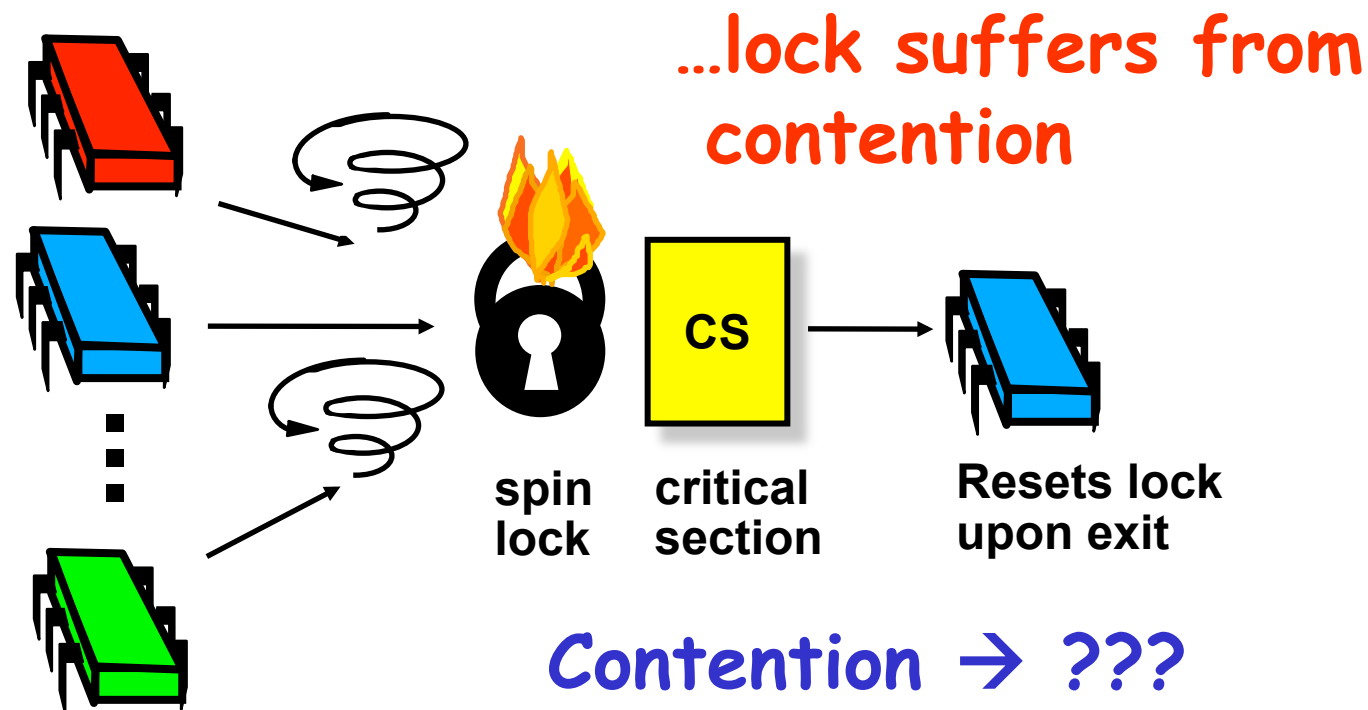


Notice: these are distinct phenomena

Basic Spin-Lock



Basic Spin-Lock



Review: Test-and-Set

- Boolean value
- Test-and-set (TAS)
 - Swap **true** with current value
 - Return value tells if prior value was **true** or **false**
- Can reset just by writing **false**
- TAS aka "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

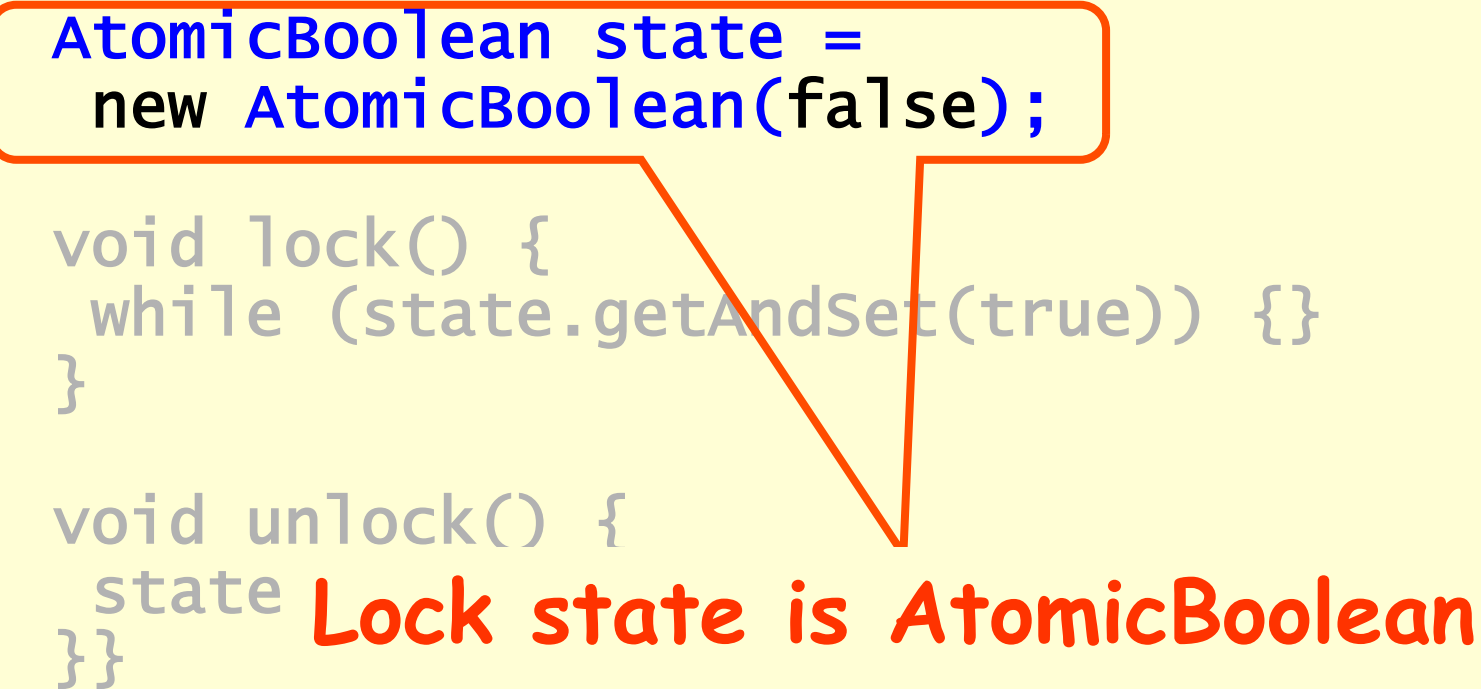
Test-and-set Lock

```
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

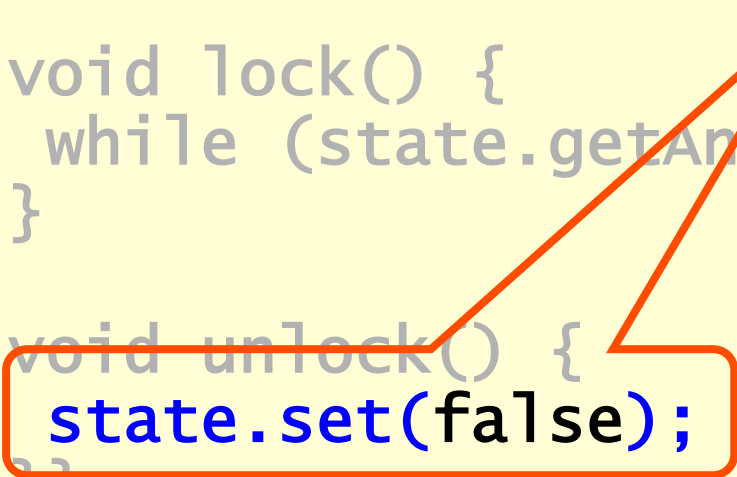
```
class TASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    }  
}
```

Keep trying until lock acquired

Test-and-set Lock

```
class TA {  
    AtomicCB  
    new At  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    }  
}
```

**Release lock by resetting
state to false**



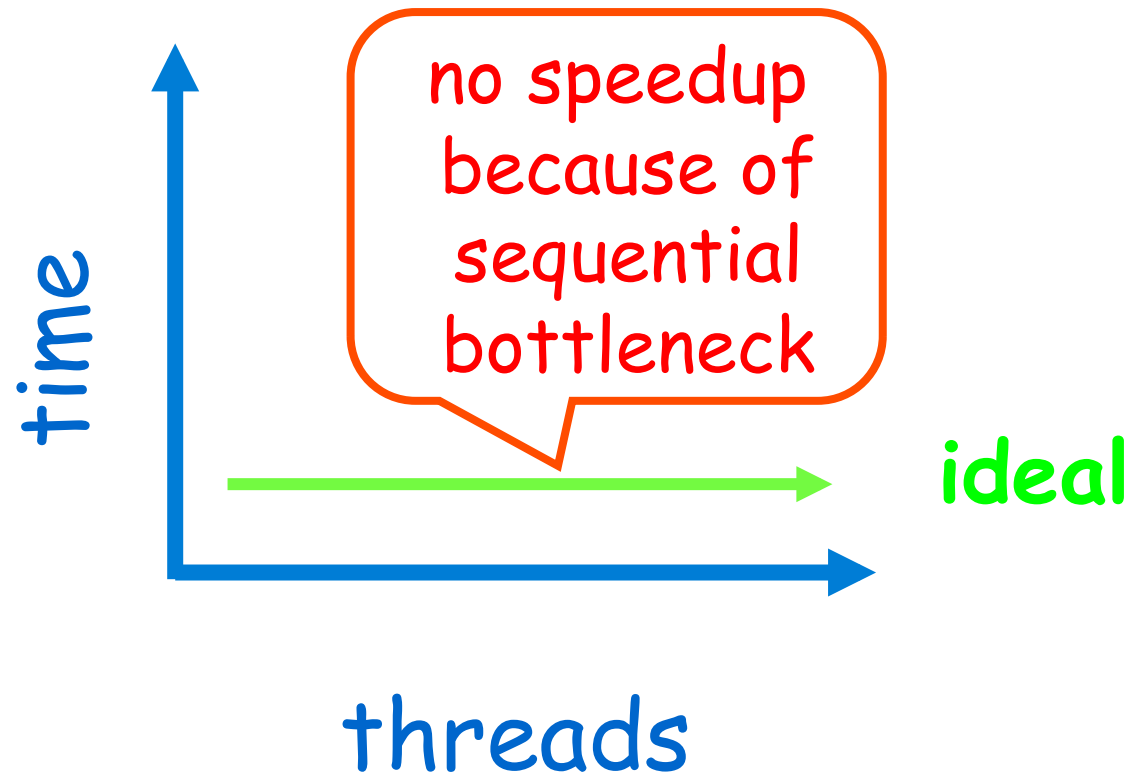
Space Complexity

- 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...

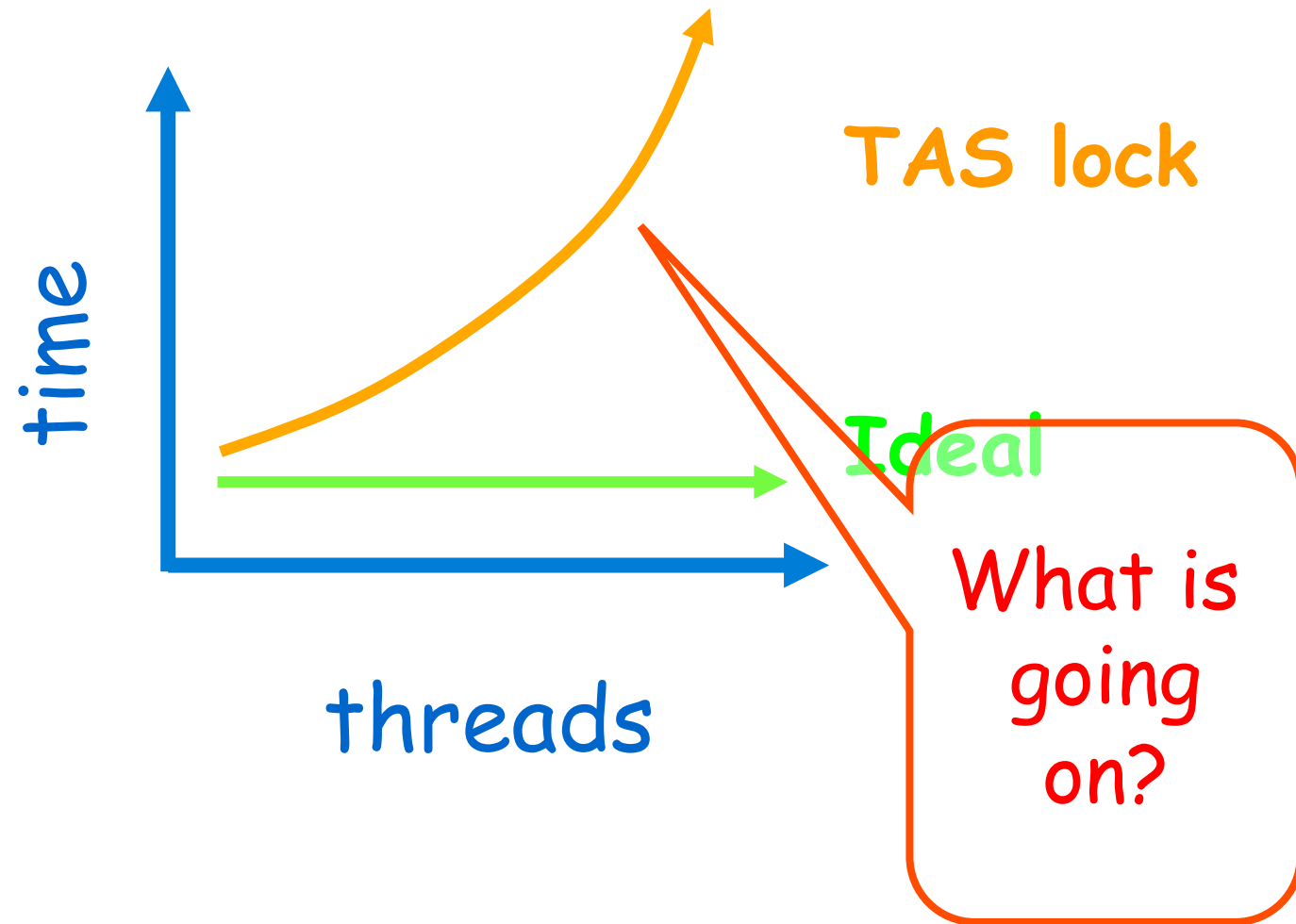
Performance

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

Graph



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

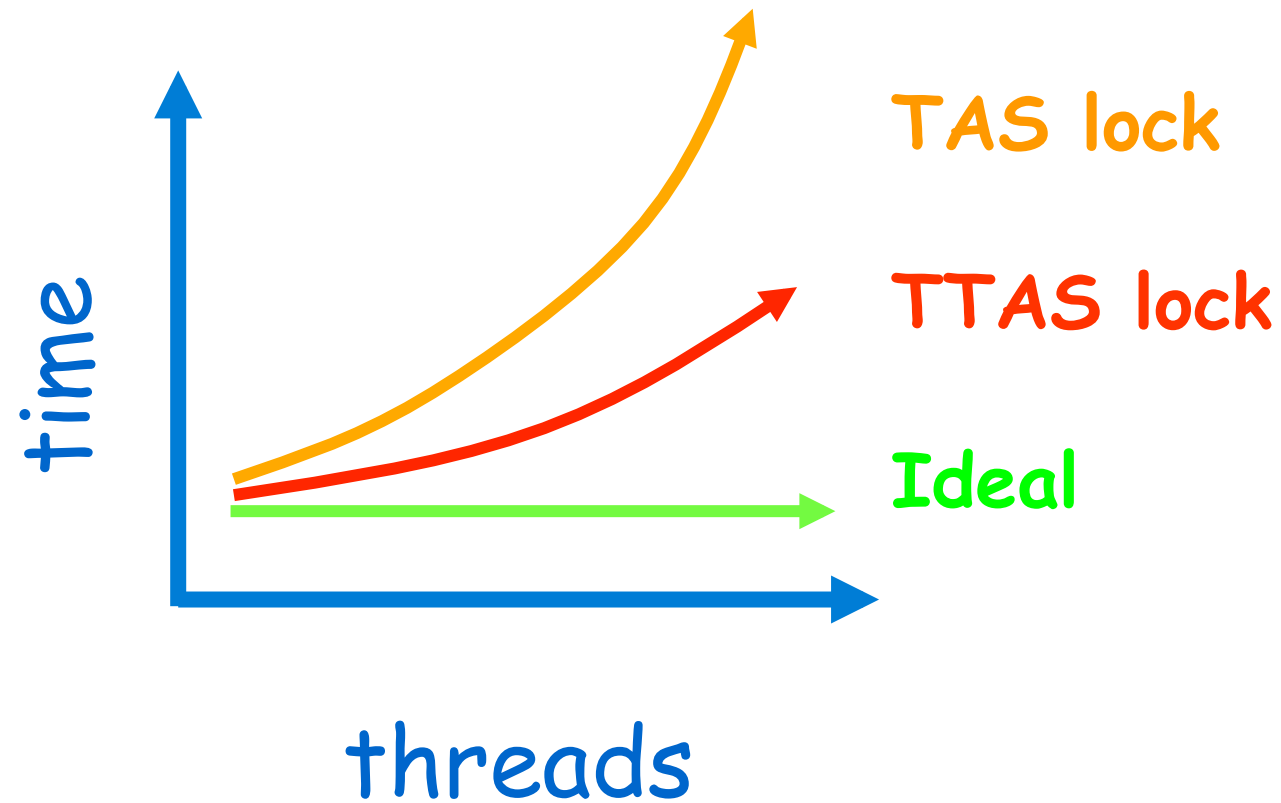
```
class TTASlock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);
```

```
    void lock() {  
        while (true) {  
            while (state.get()) {}
```

```
            if (!state.getAndSet(true))  
                return;  
        }  
    }
```

Then try to
acquire it

Mystery #2



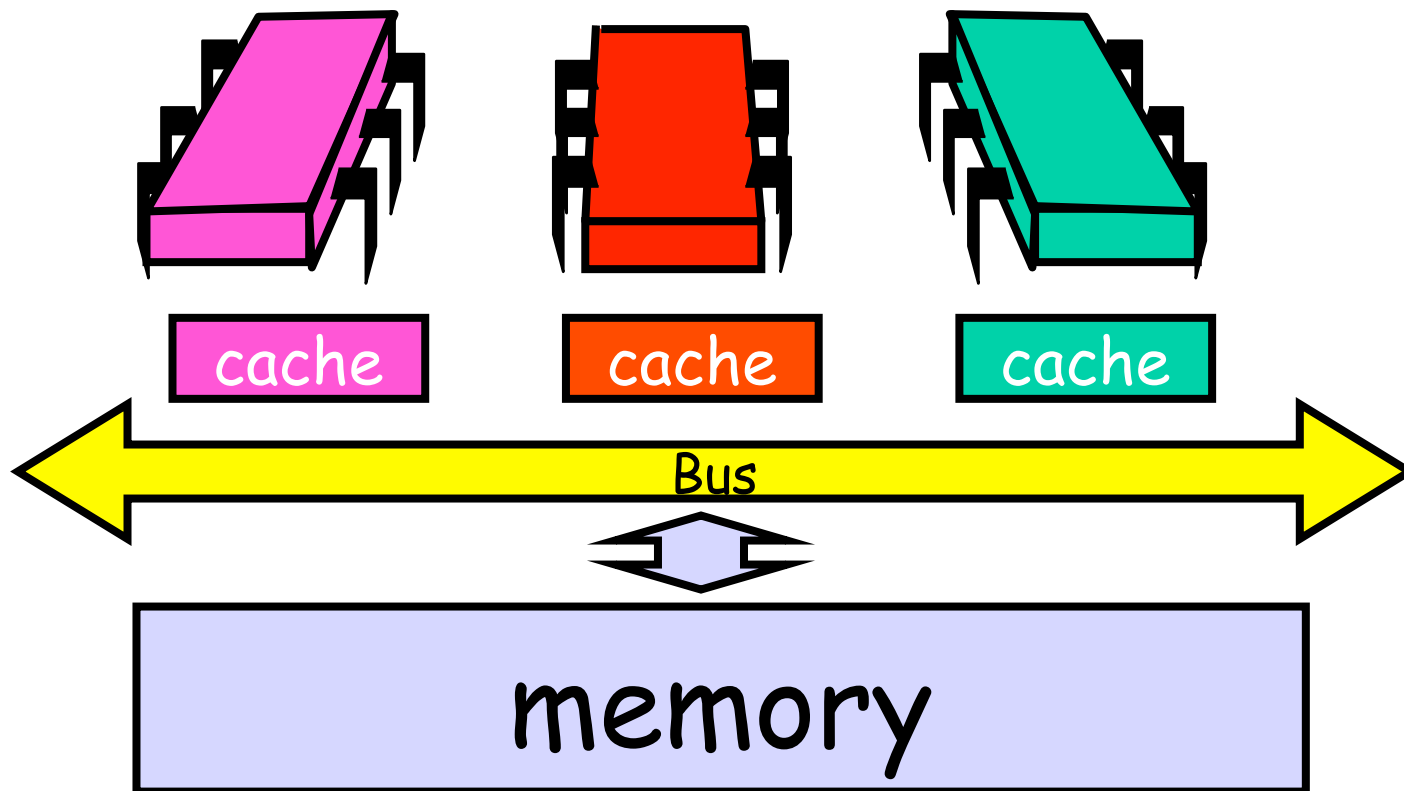
Mystery

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

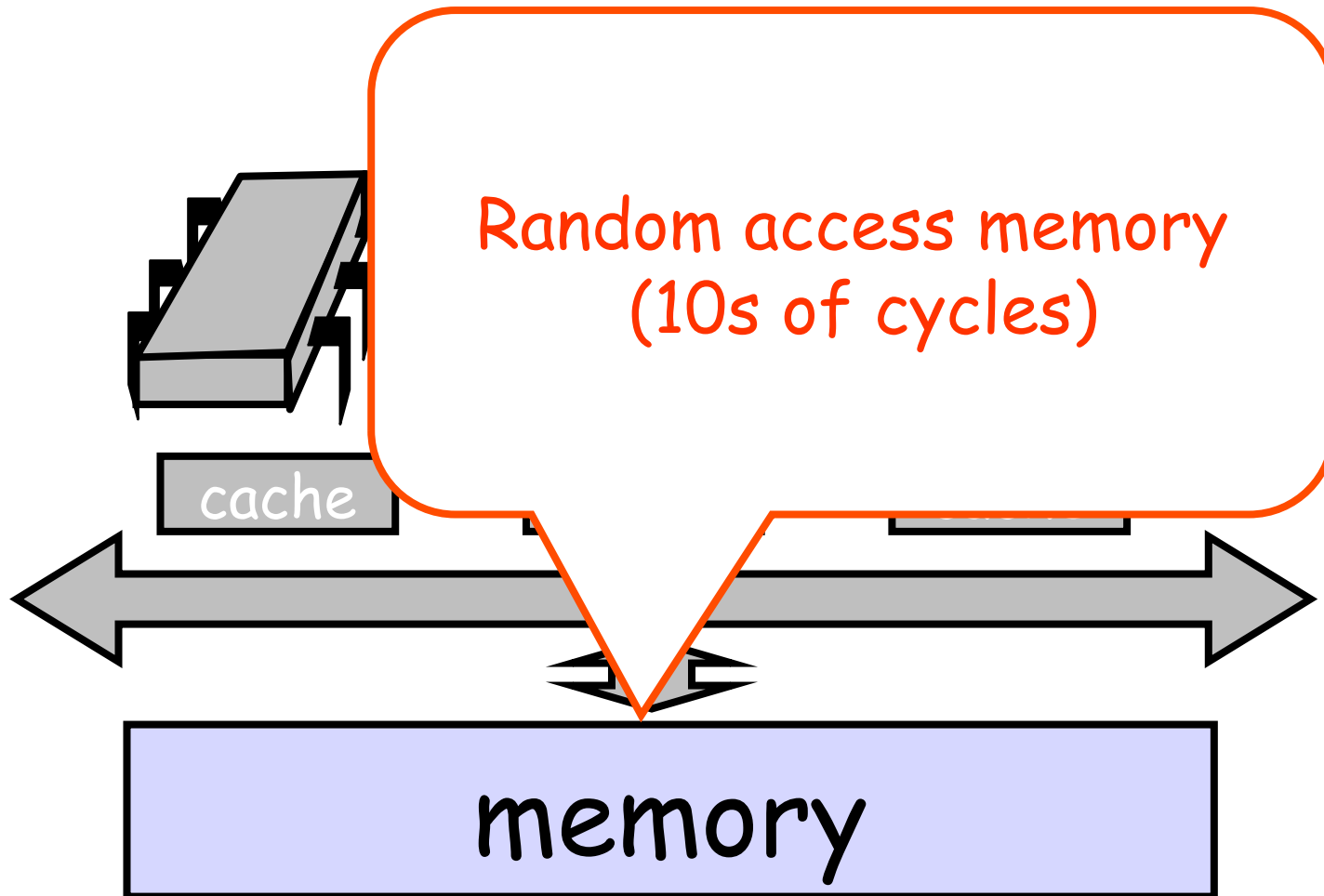
Opinion

- 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



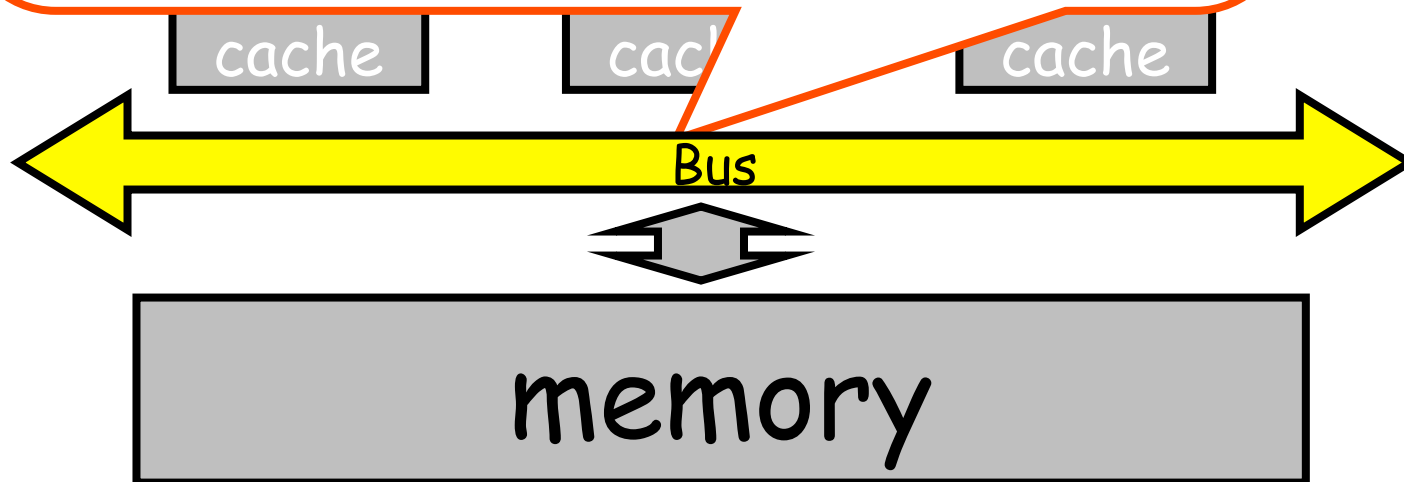
Bus-Based Architectures



Bus-Based Architectures

Shared Bus

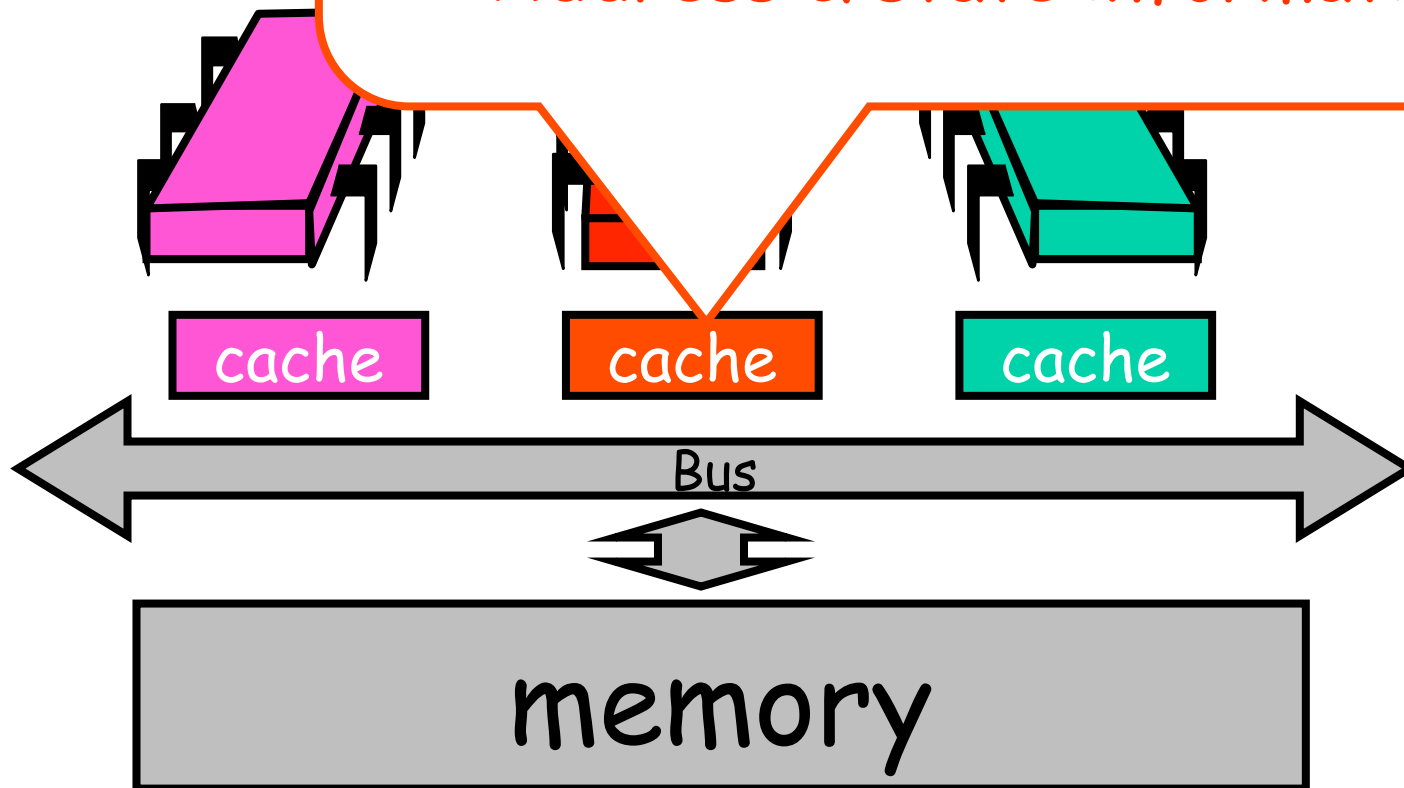
- broadcast medium
- One broadcaster at a time
- Processors and memory all "snoop"



Bus-E

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™

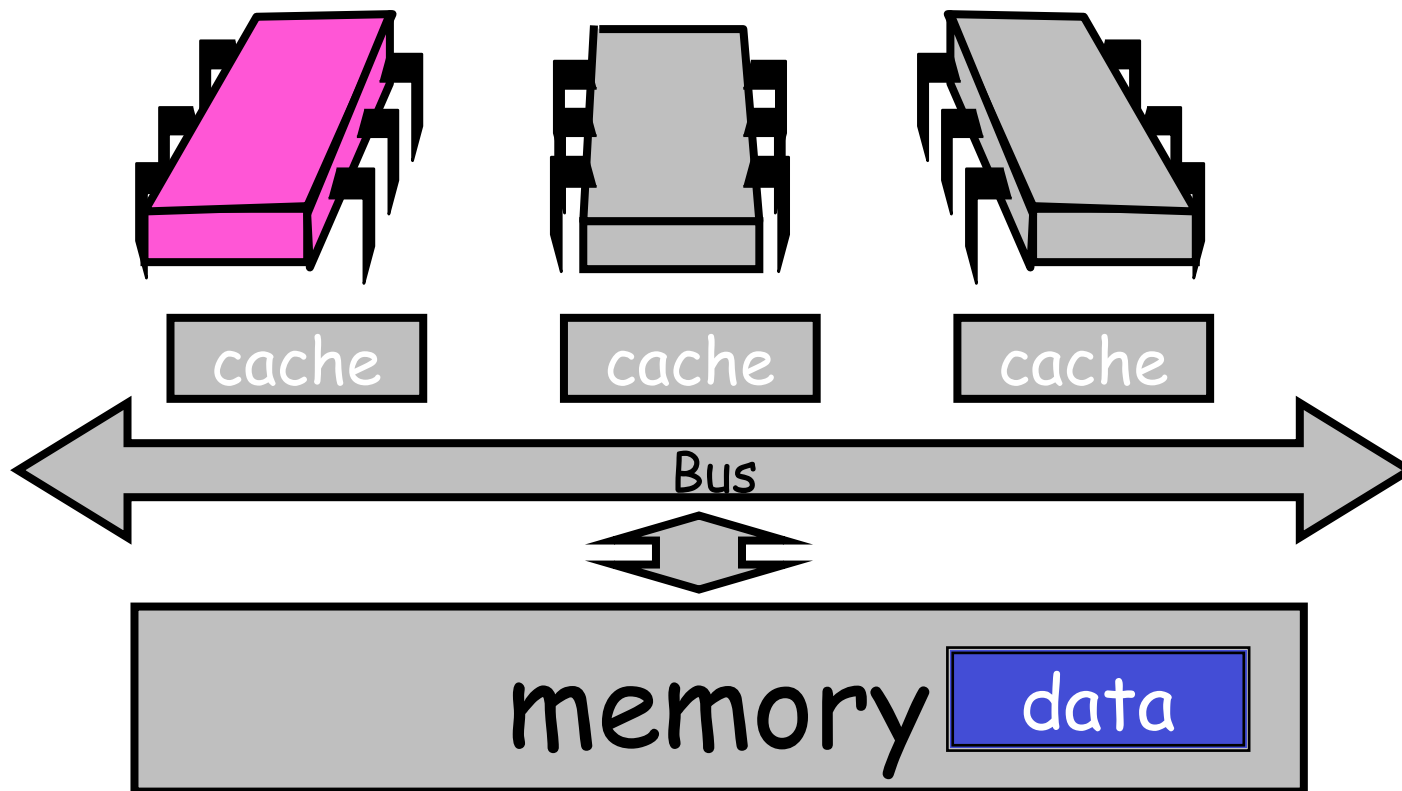
Jargon Watch

- 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™

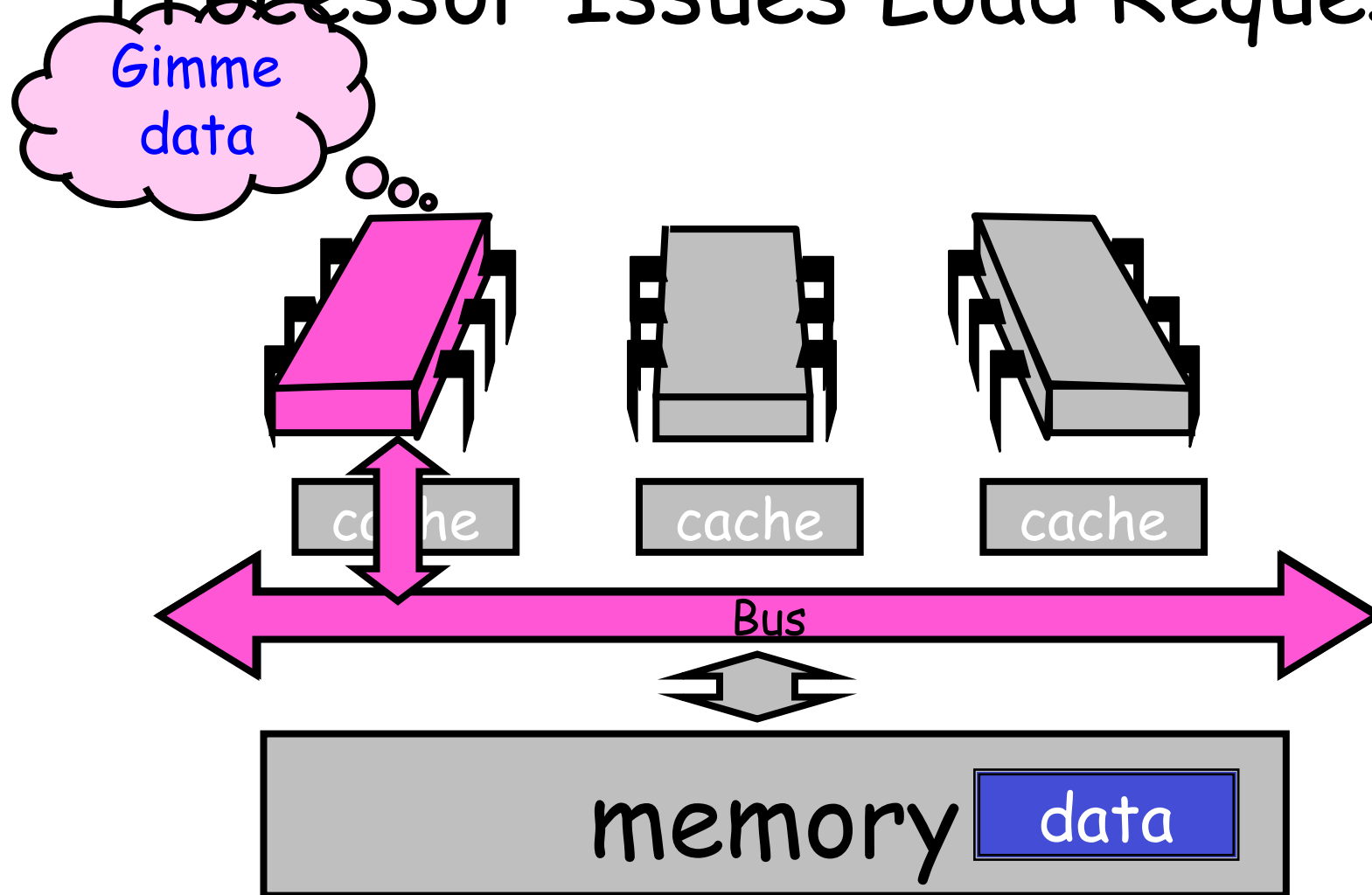
Cave Canem

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

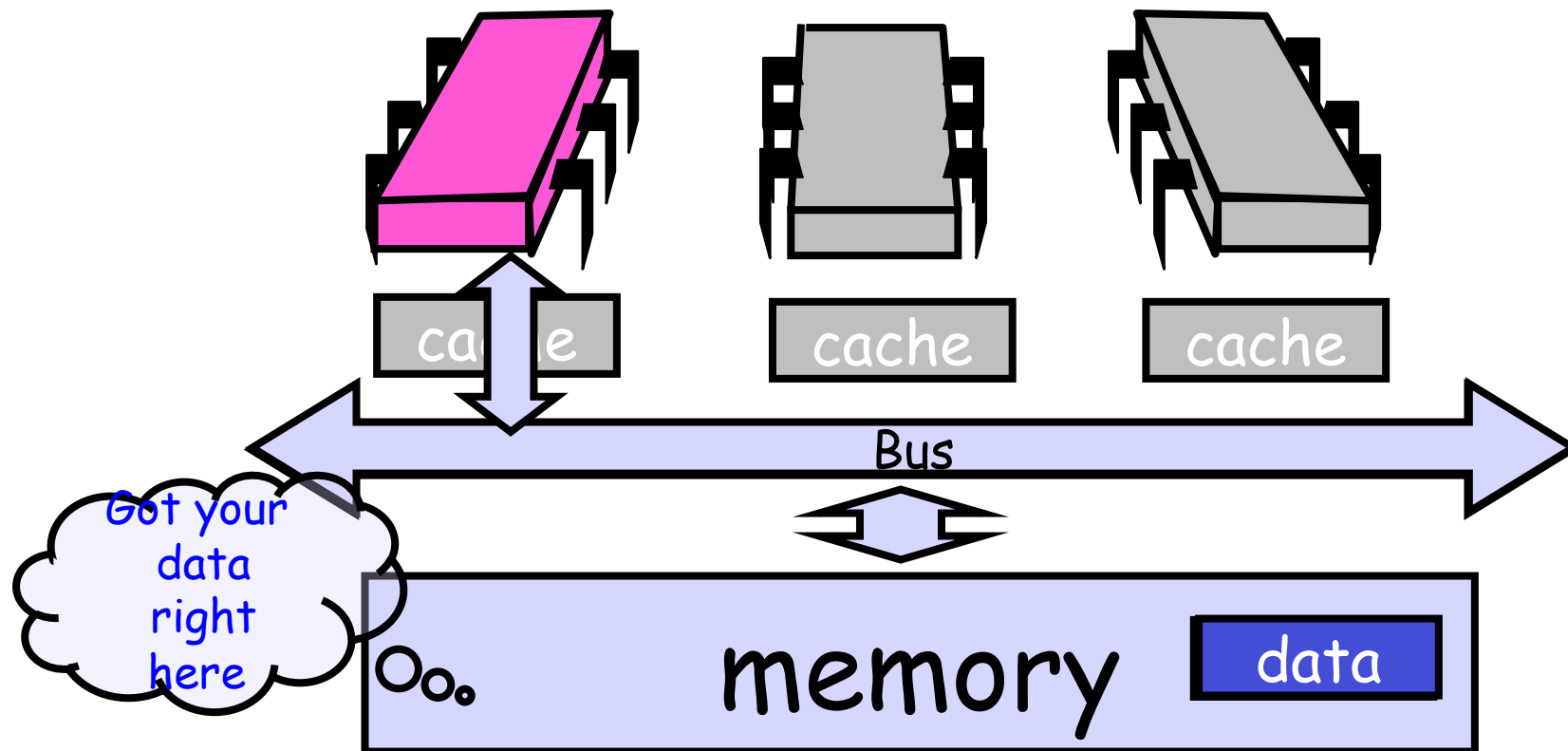
Processor Issues Load Request



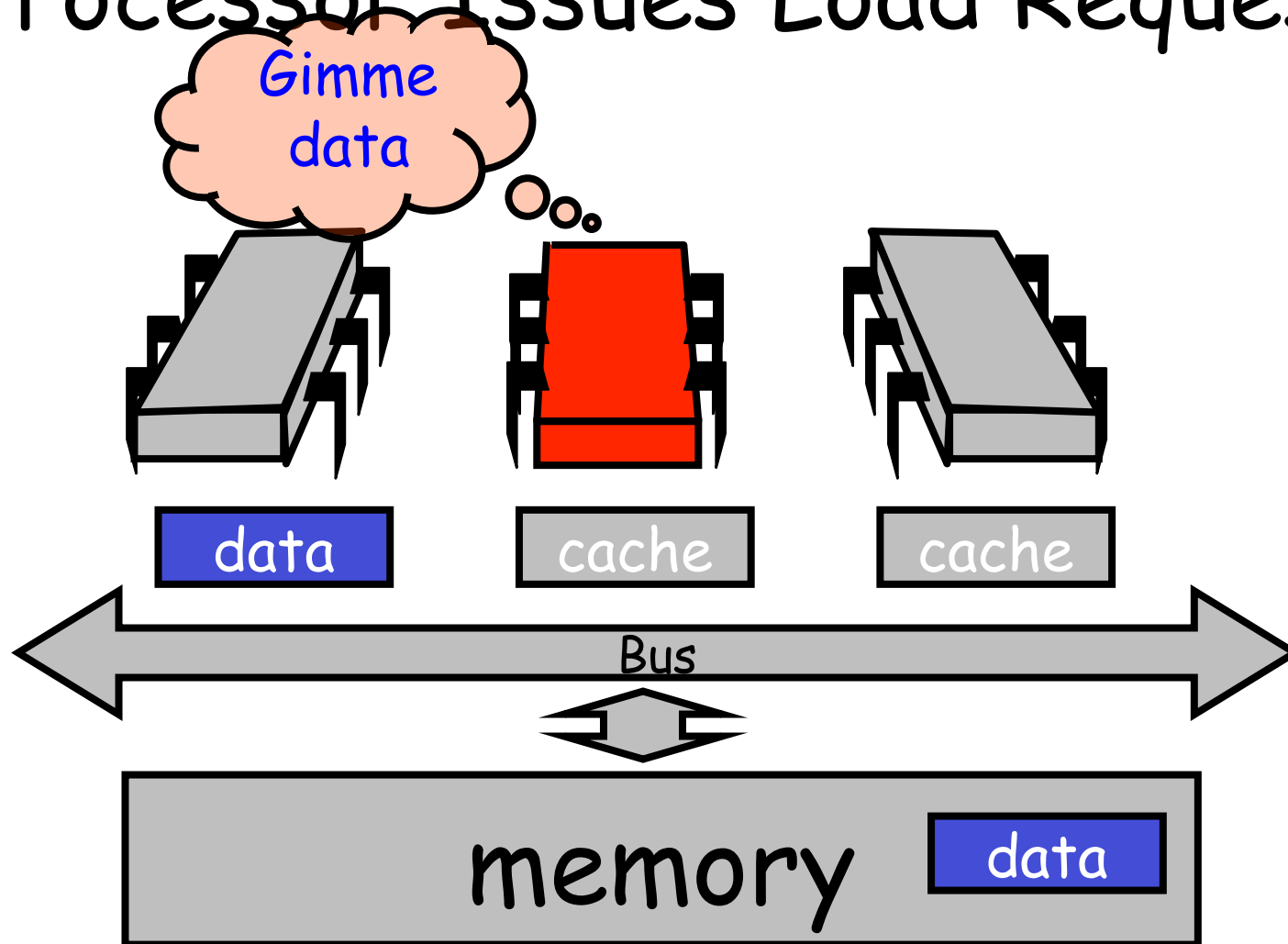
Processor Issues Load Request



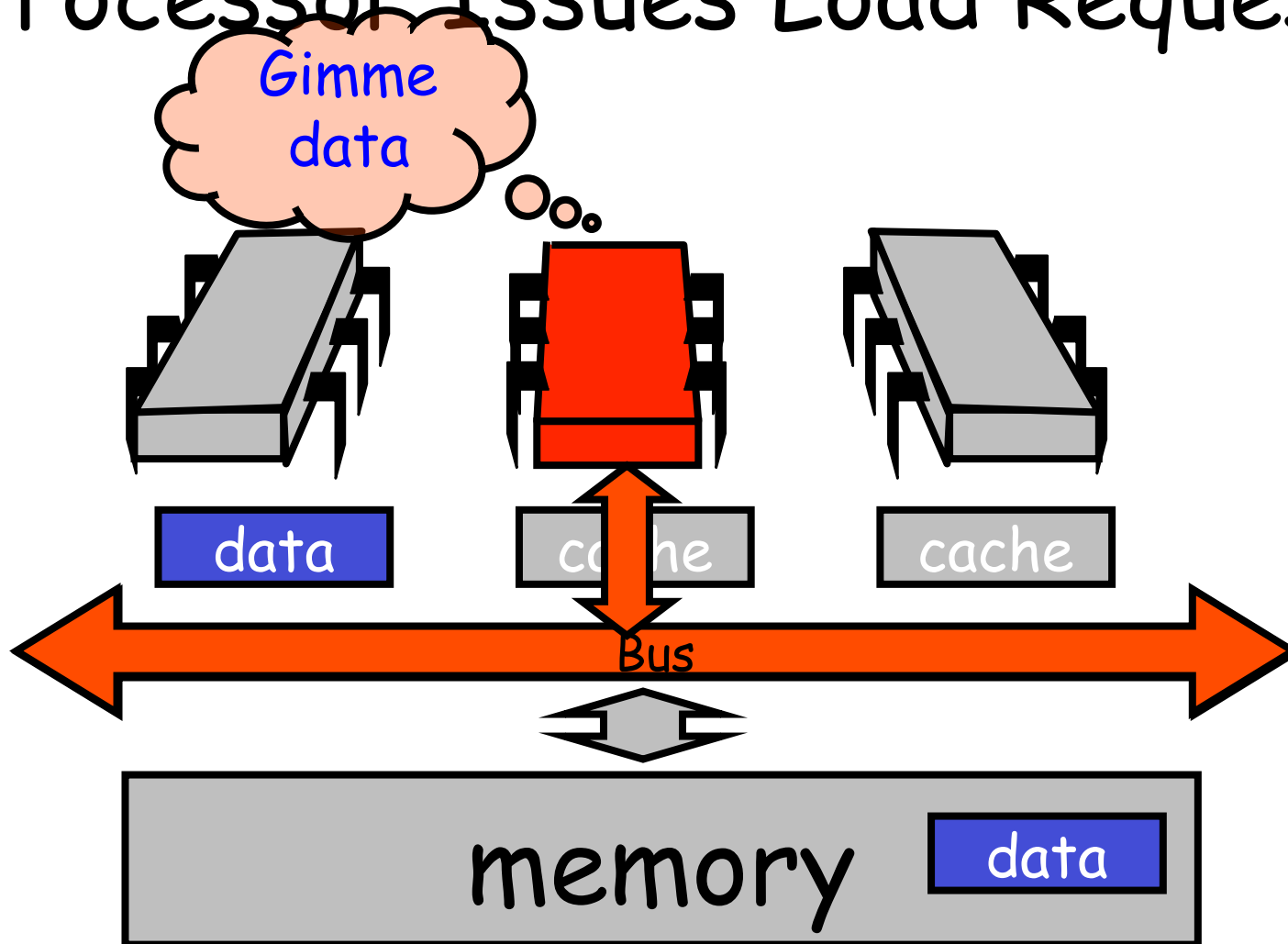
Memory Responds



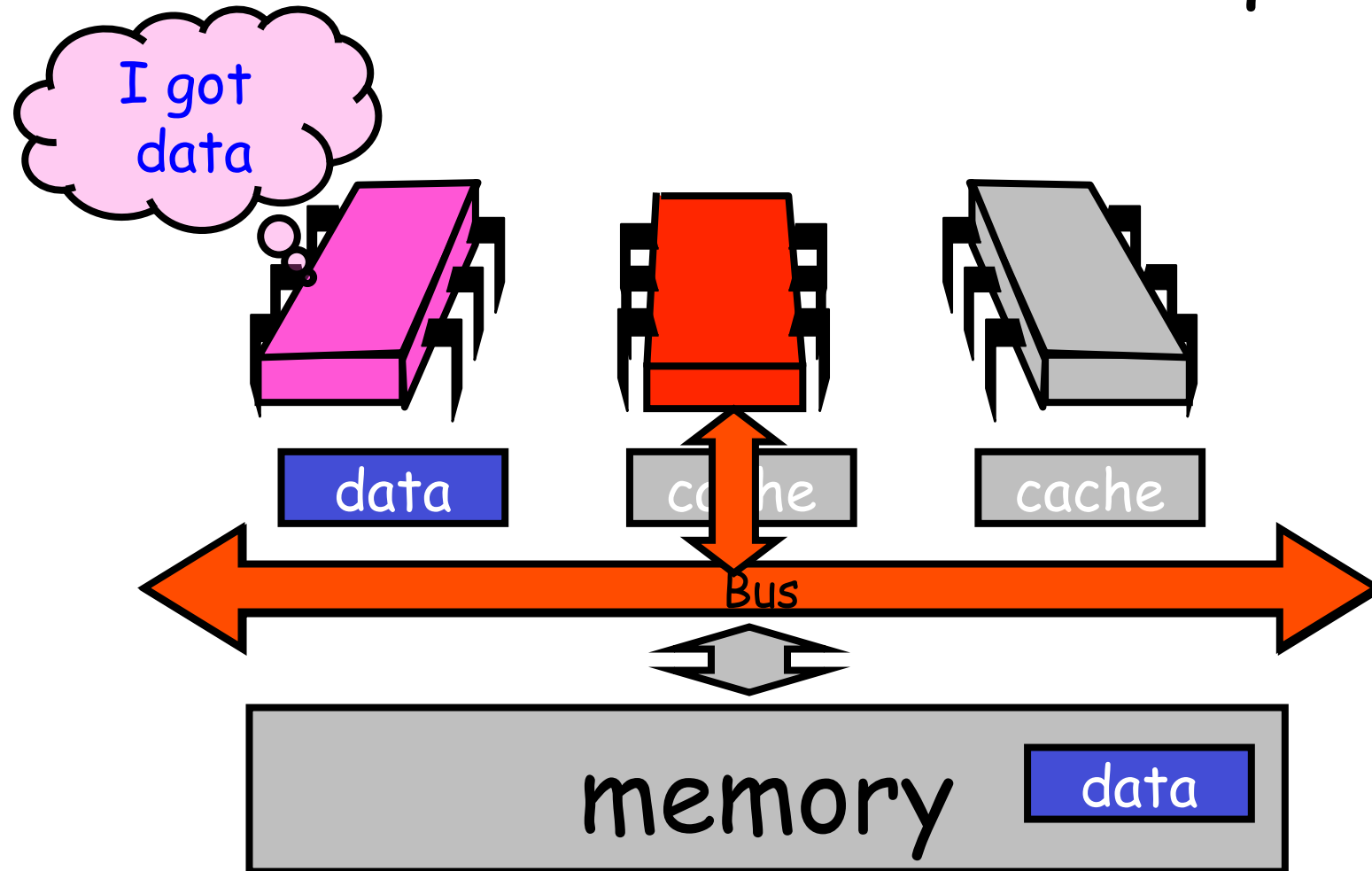
Processor Issues Load Request



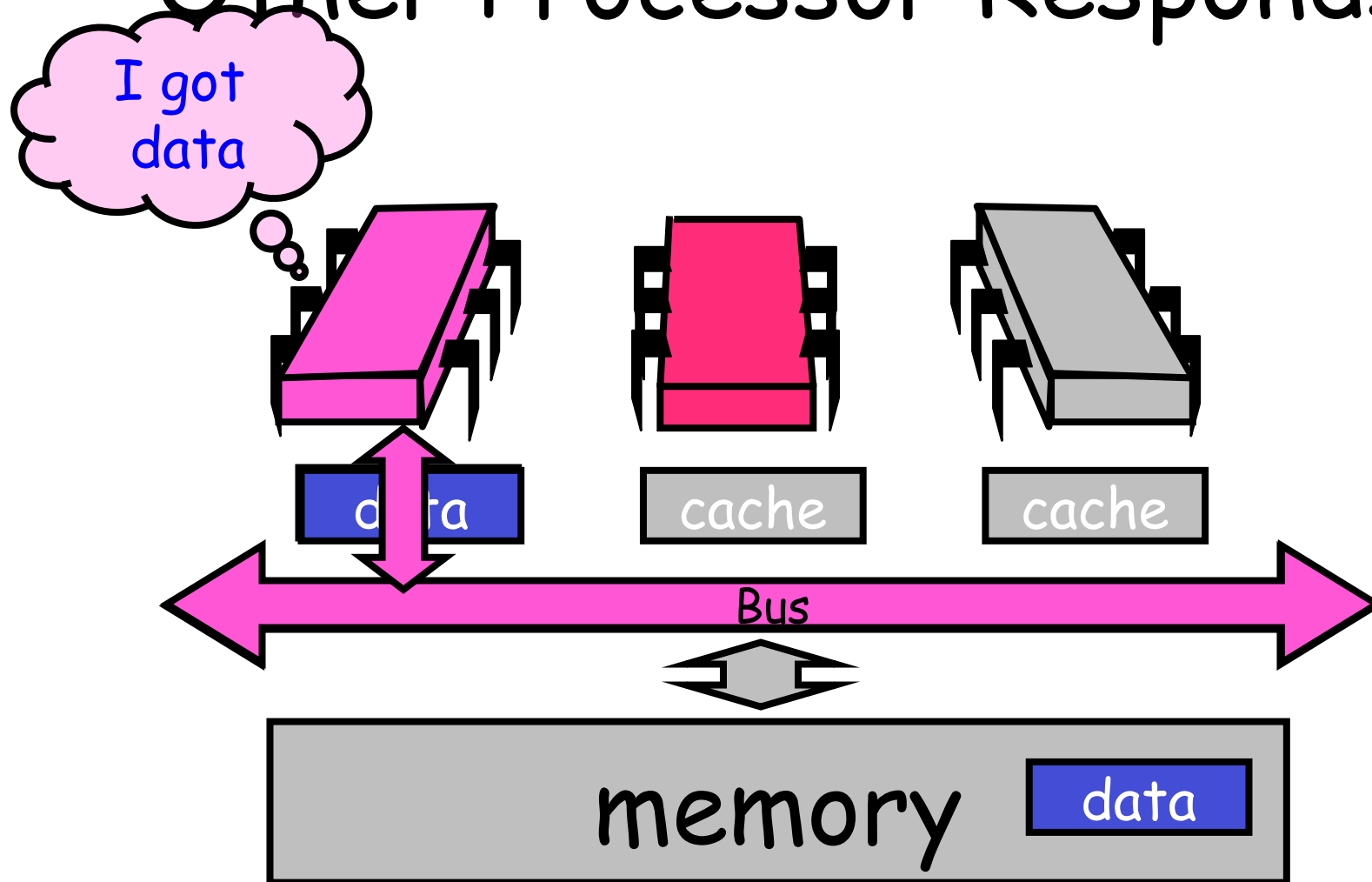
Processor Issues Load Request



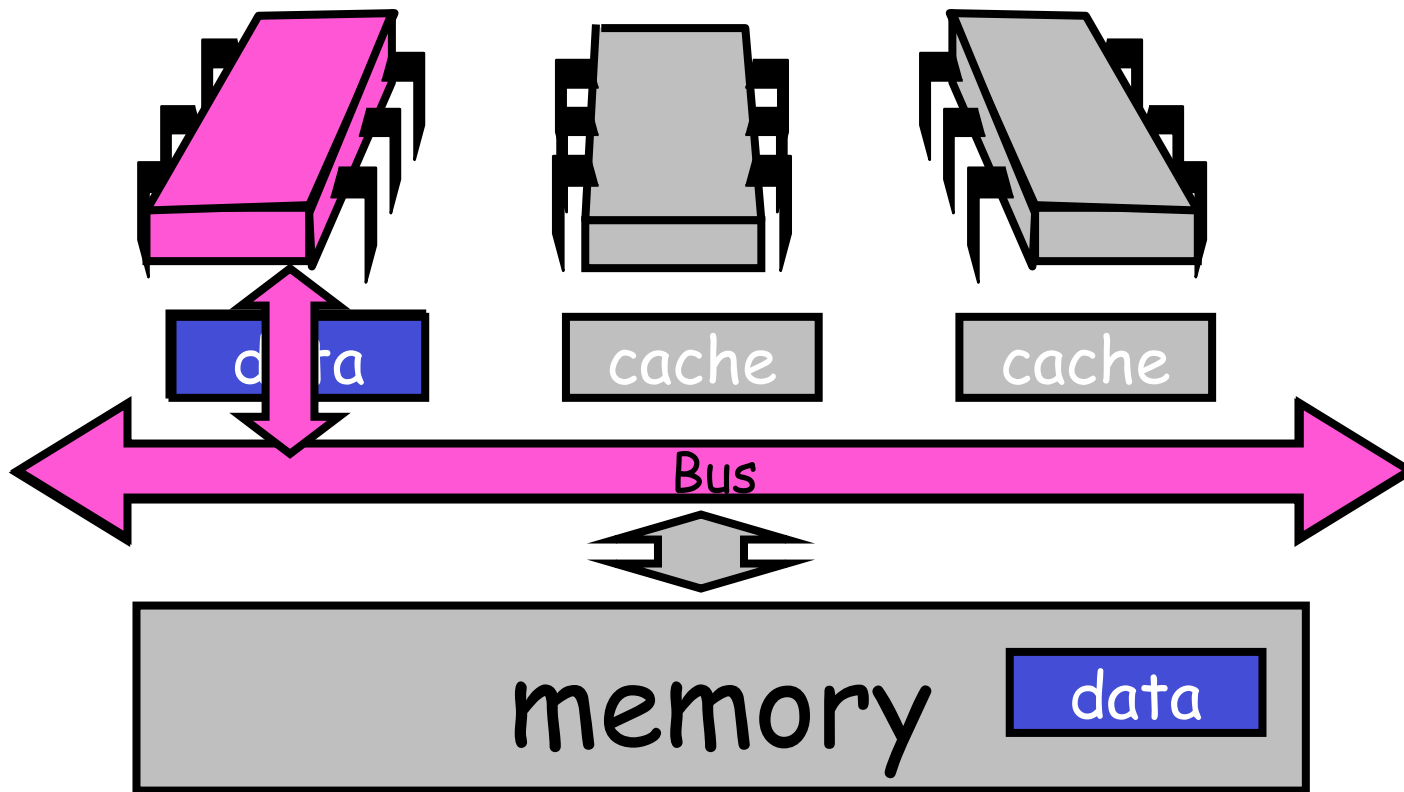
Processor Issues Load Request



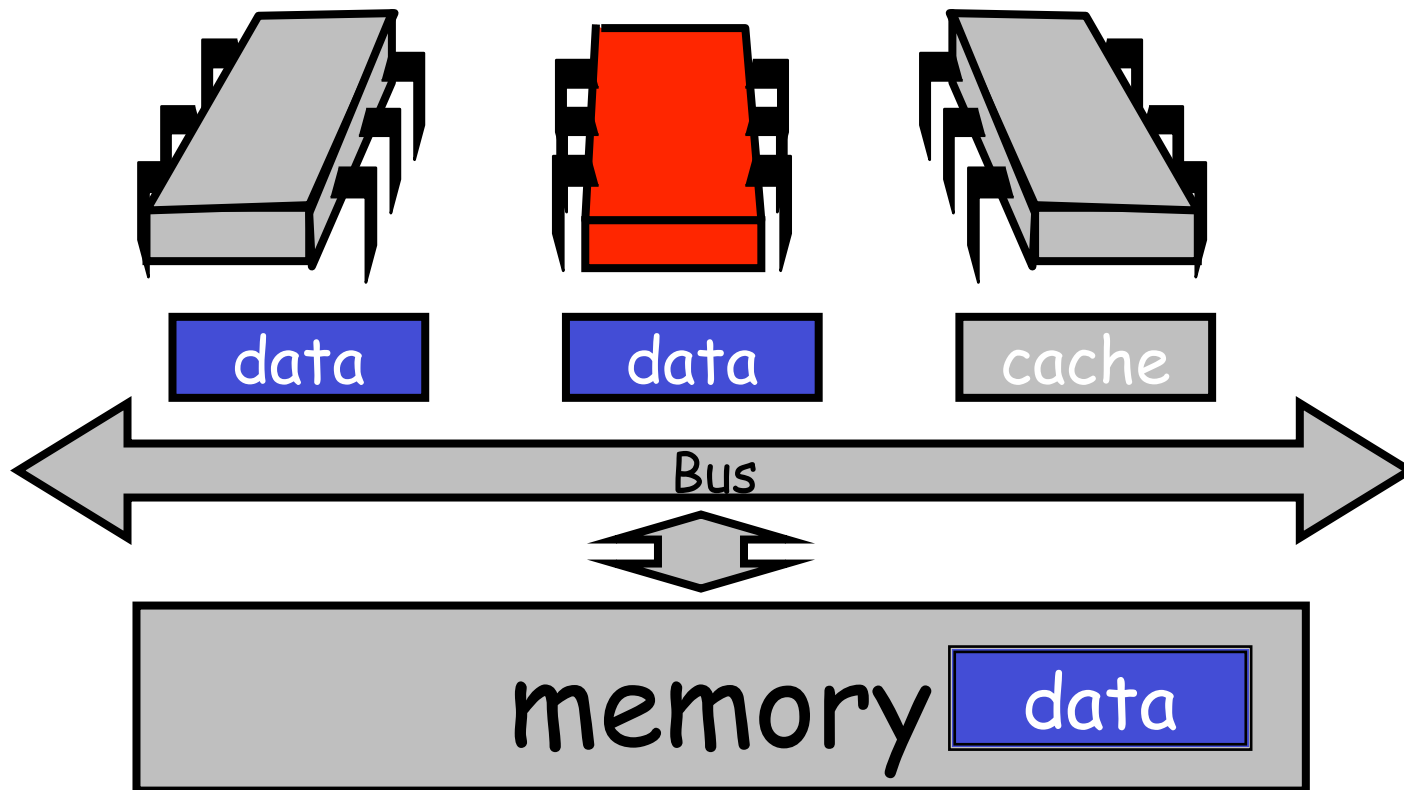
Other Processor Responds



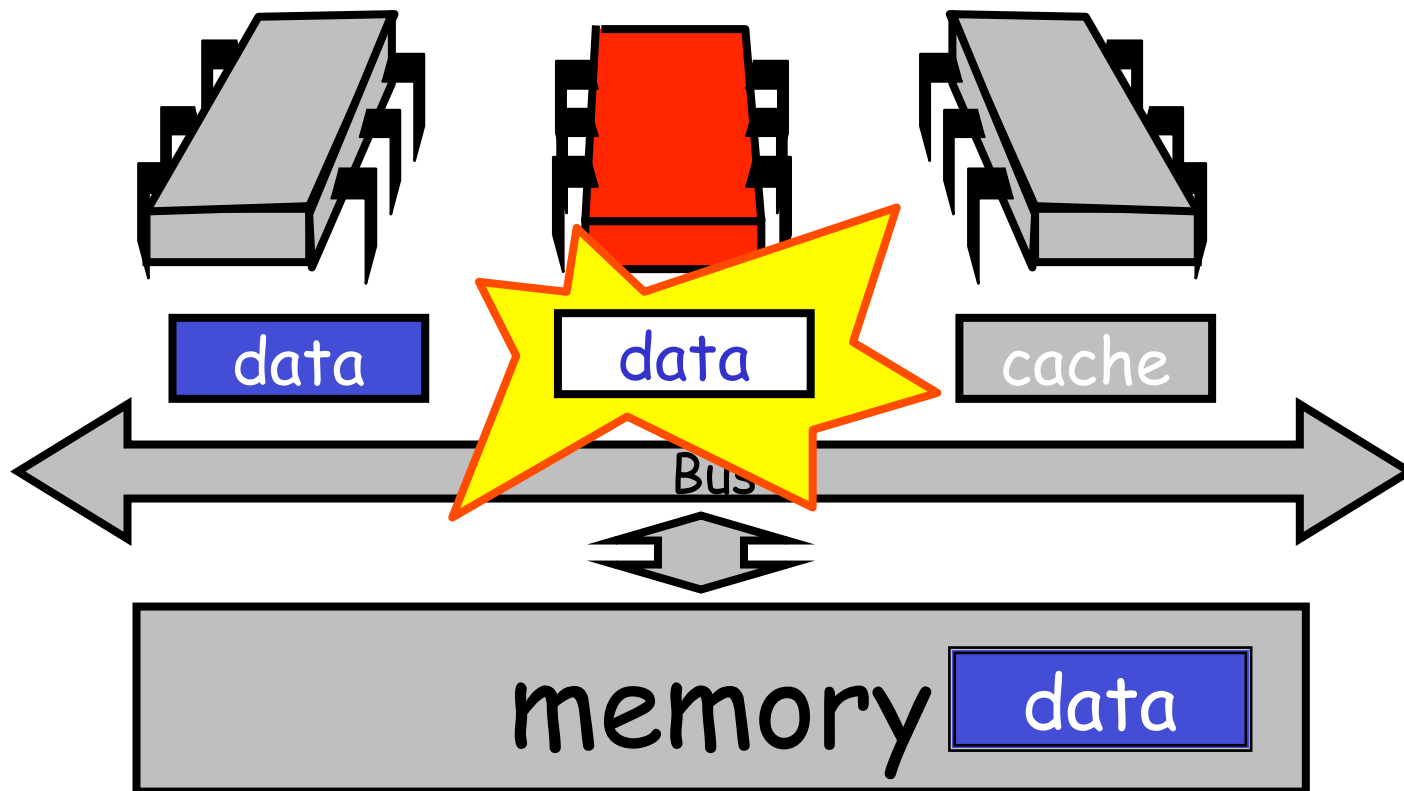
Other Processor Responds



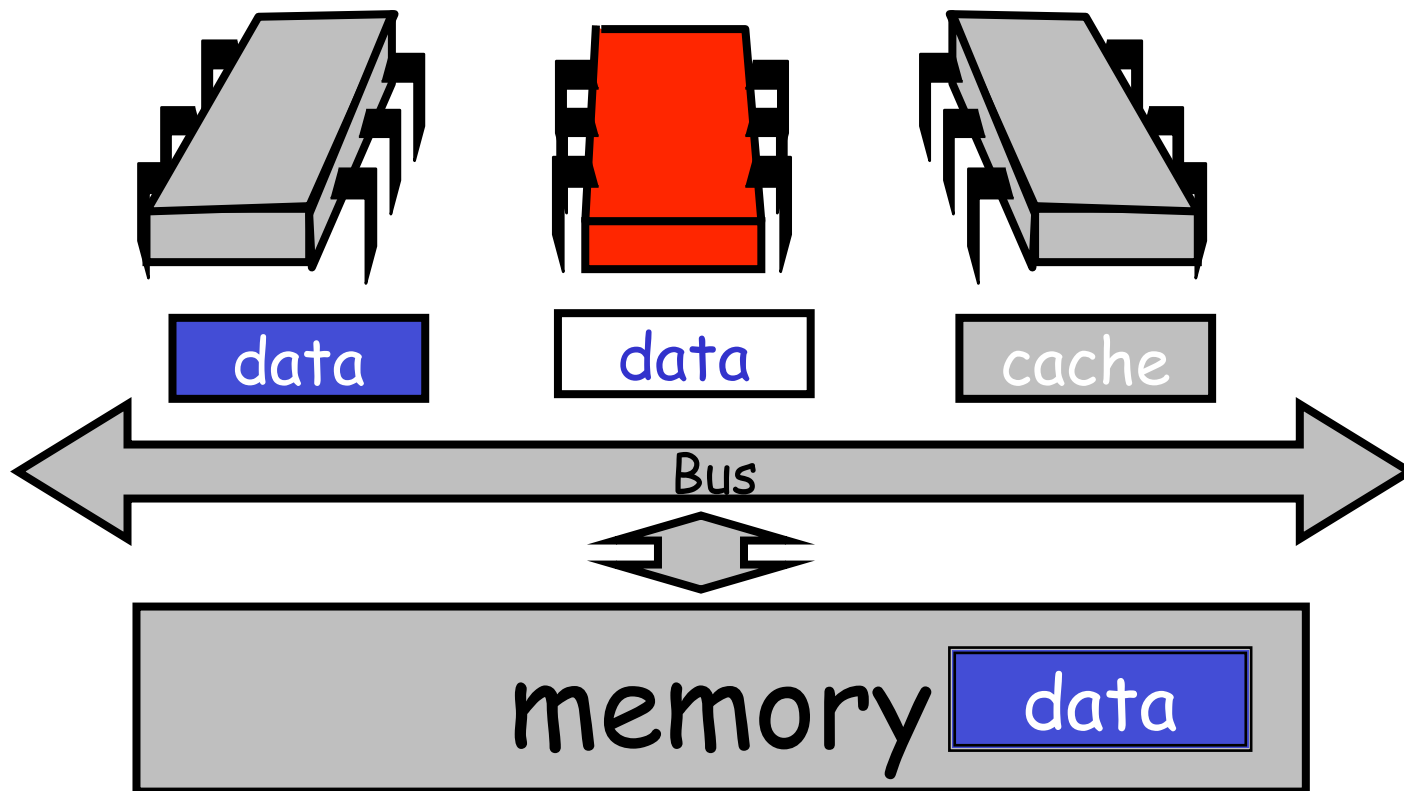
Modify Cached Data



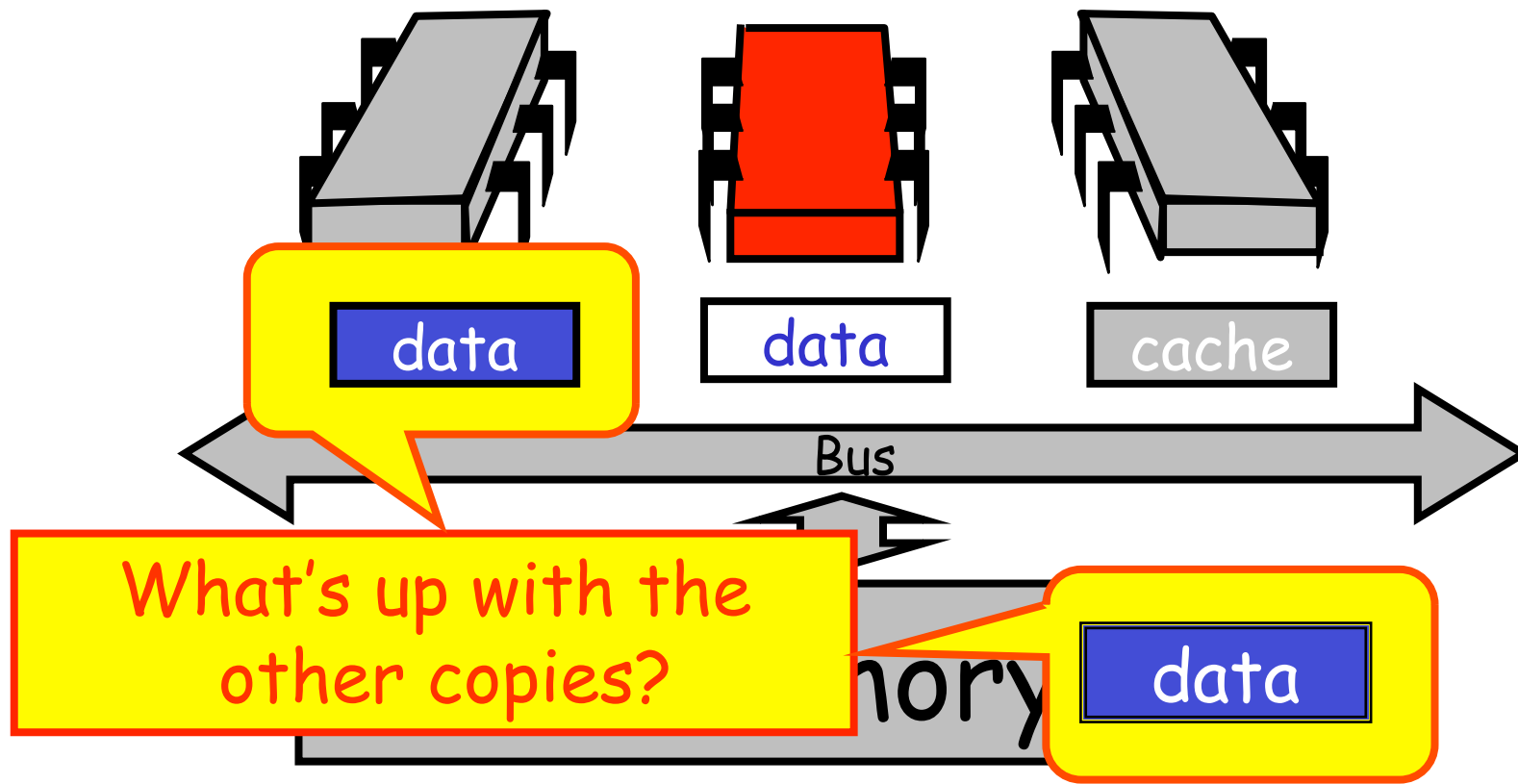
Modify Cached Data



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?

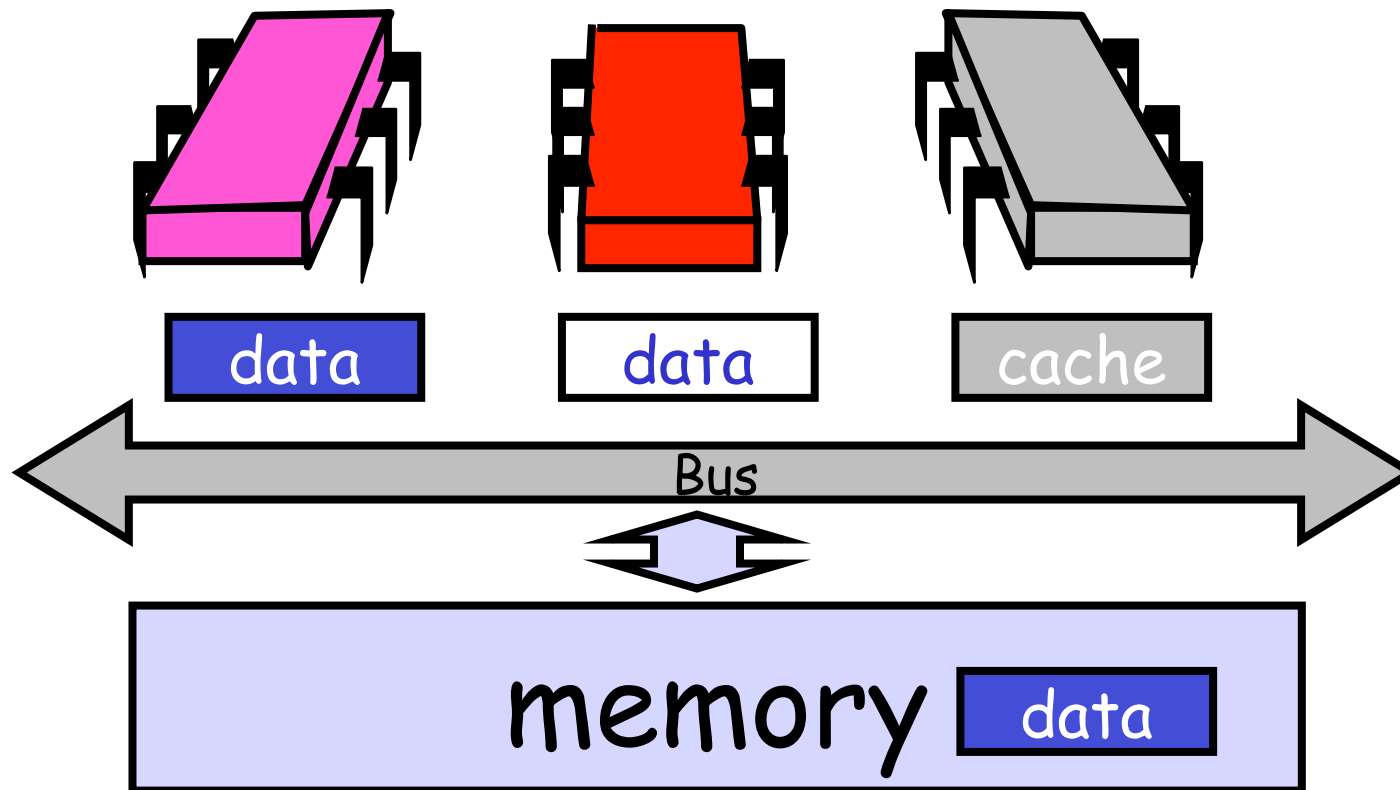
Write-Back Caches

- 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 ...

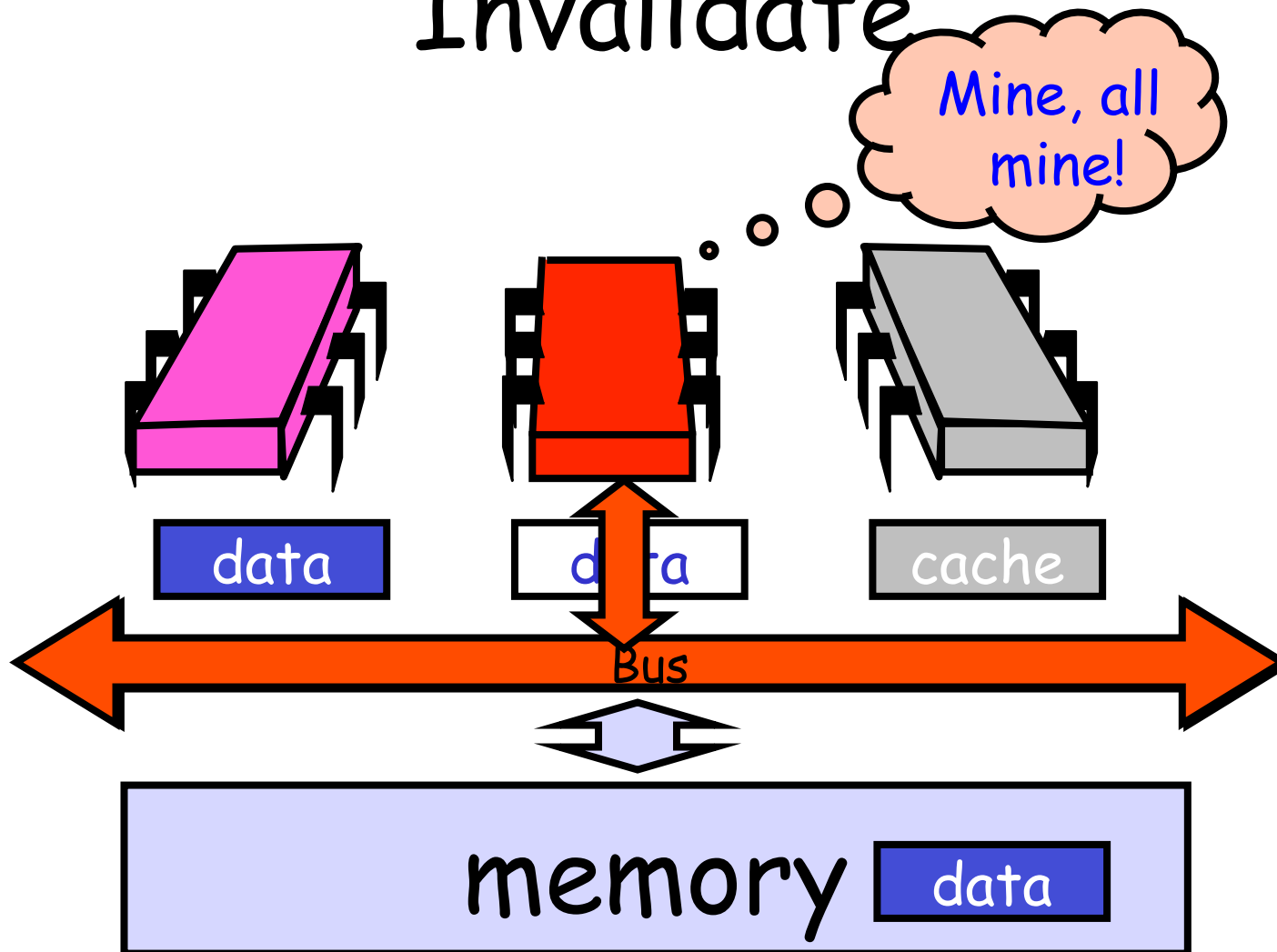
Write-Back Caches

- 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

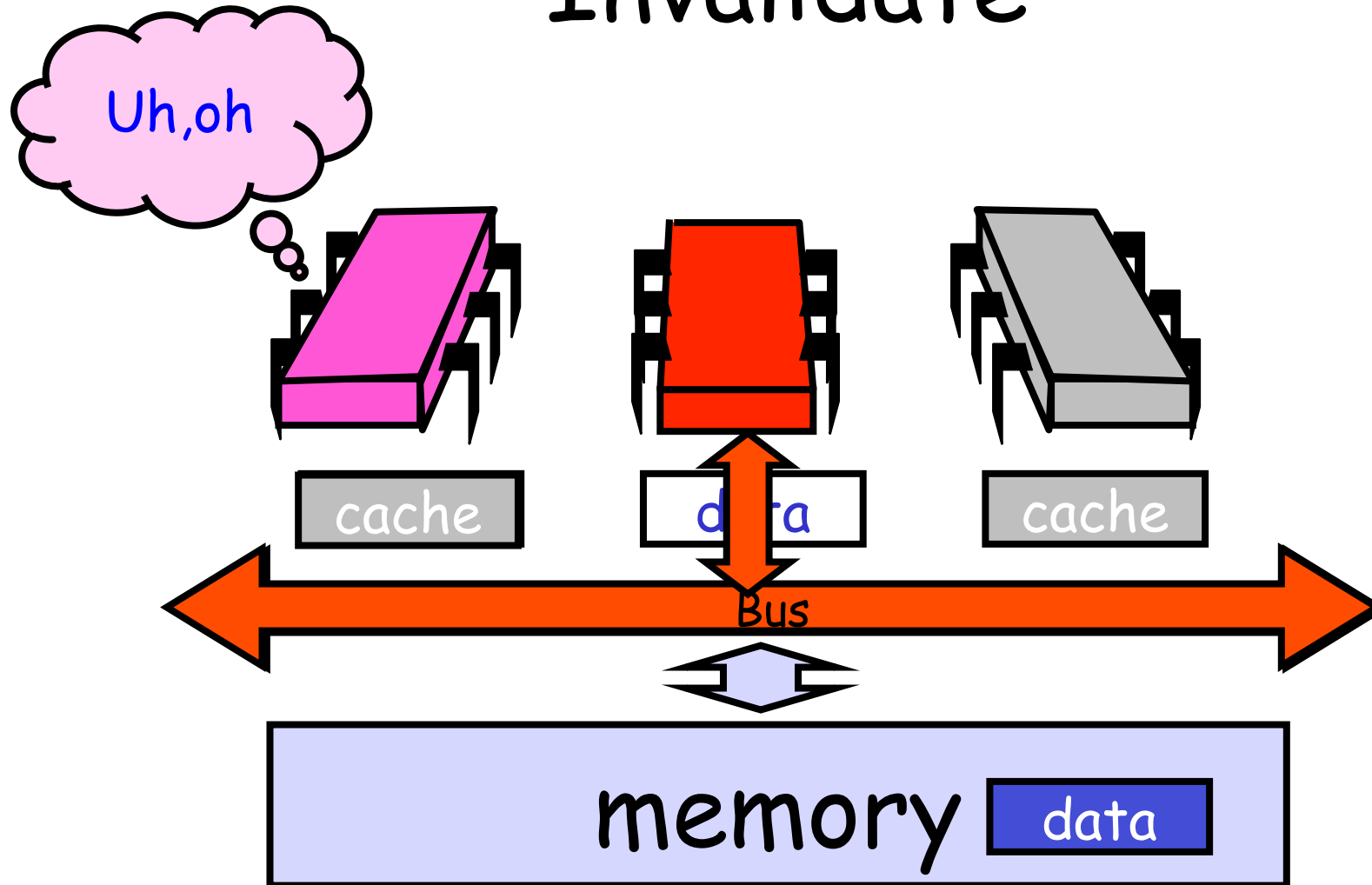
Invalidate



Invalidate

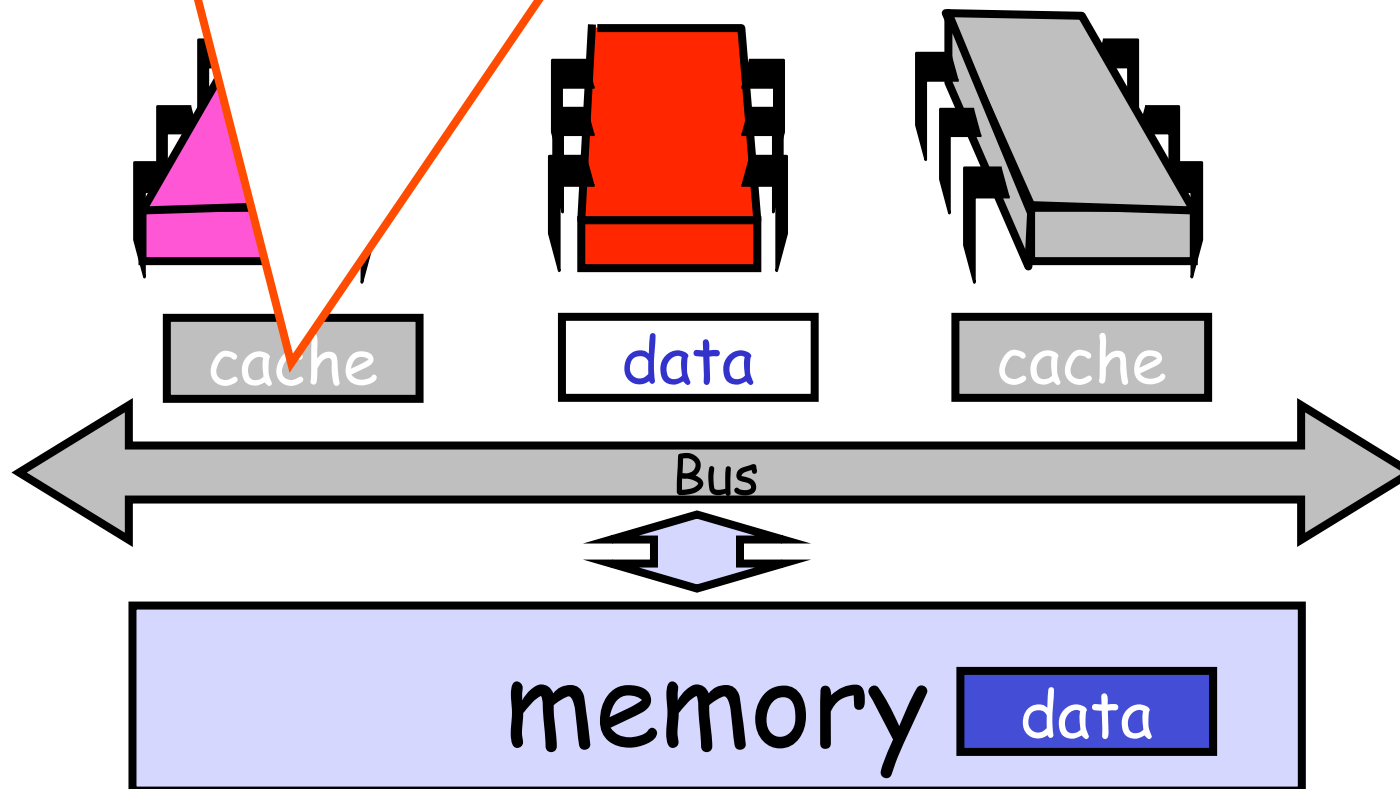


Invalidate



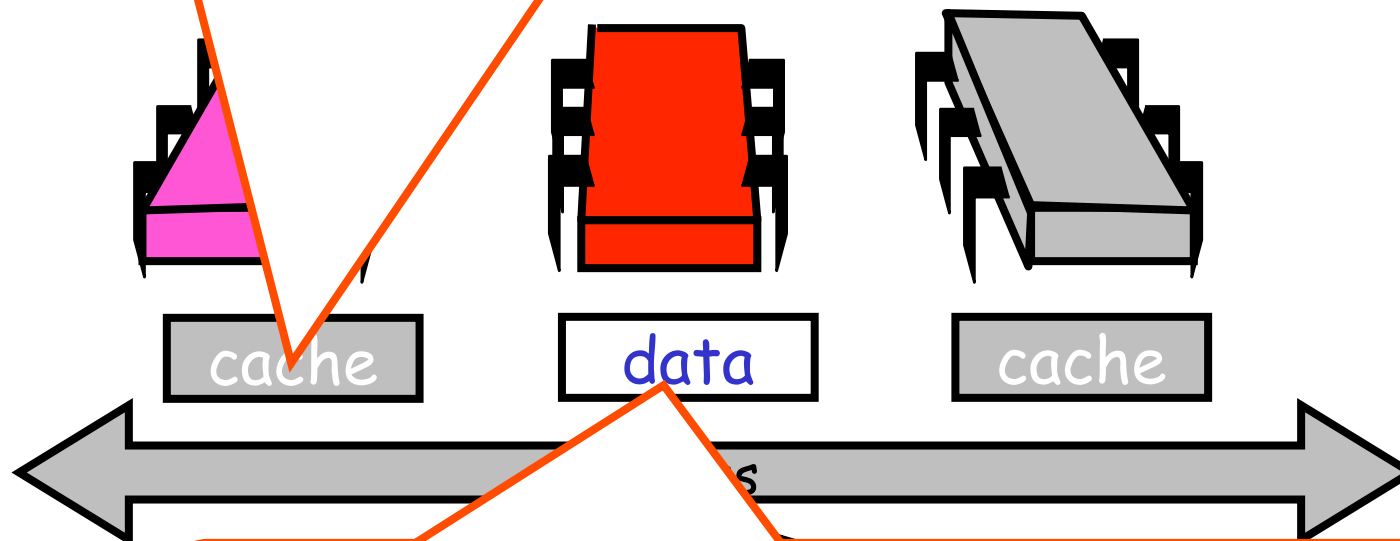
Invalidate

Other caches lose read permission



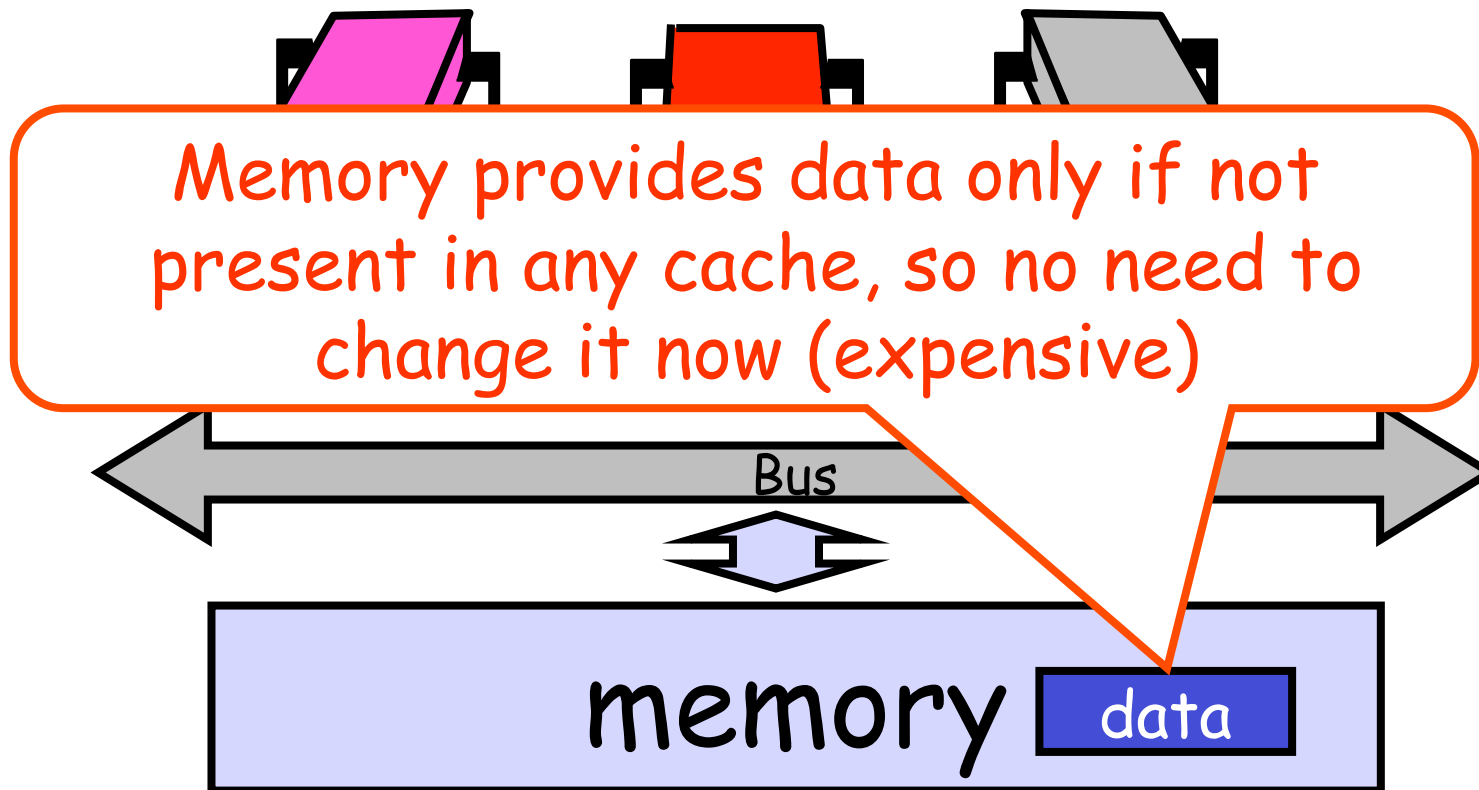
Invalidate

Other caches lose read permission

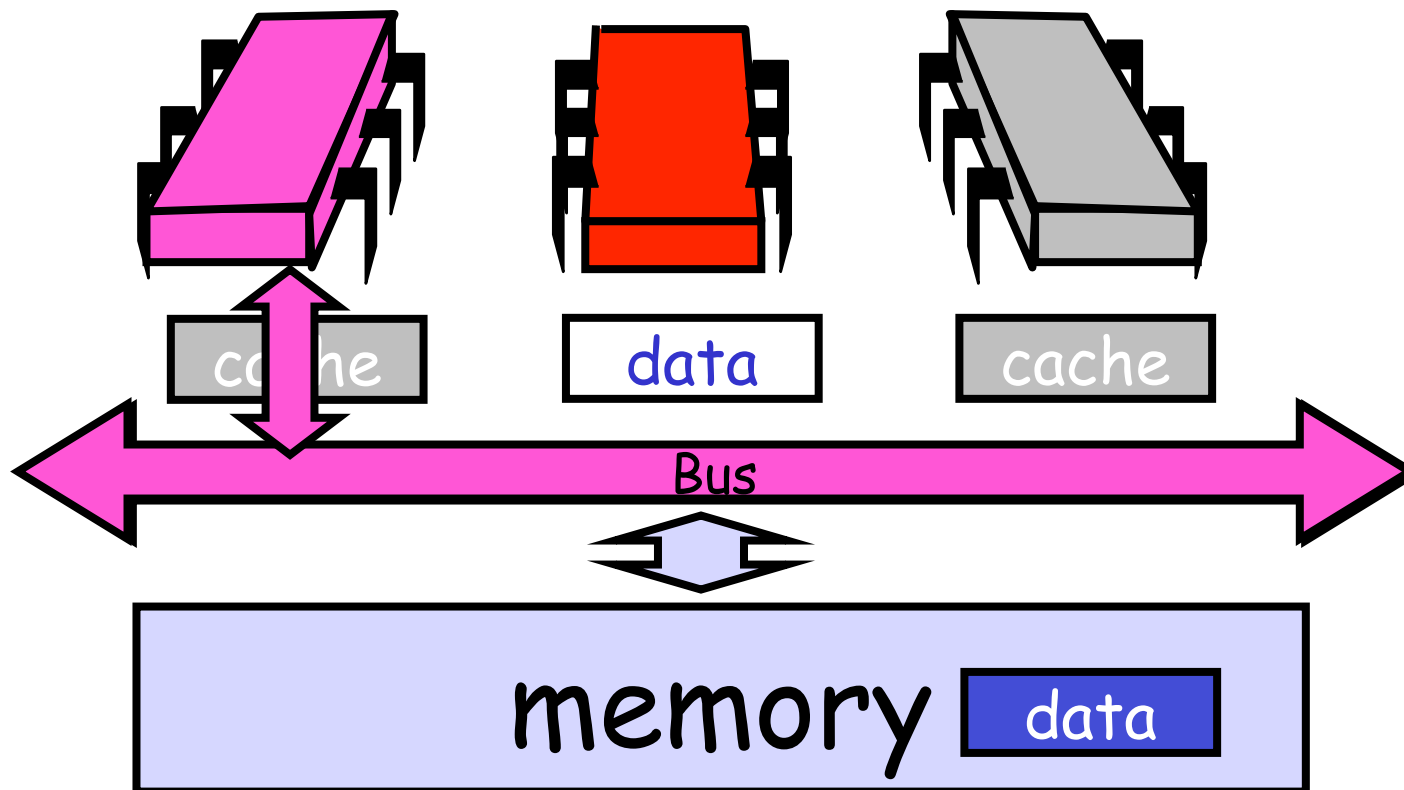


This cache acquires write permission

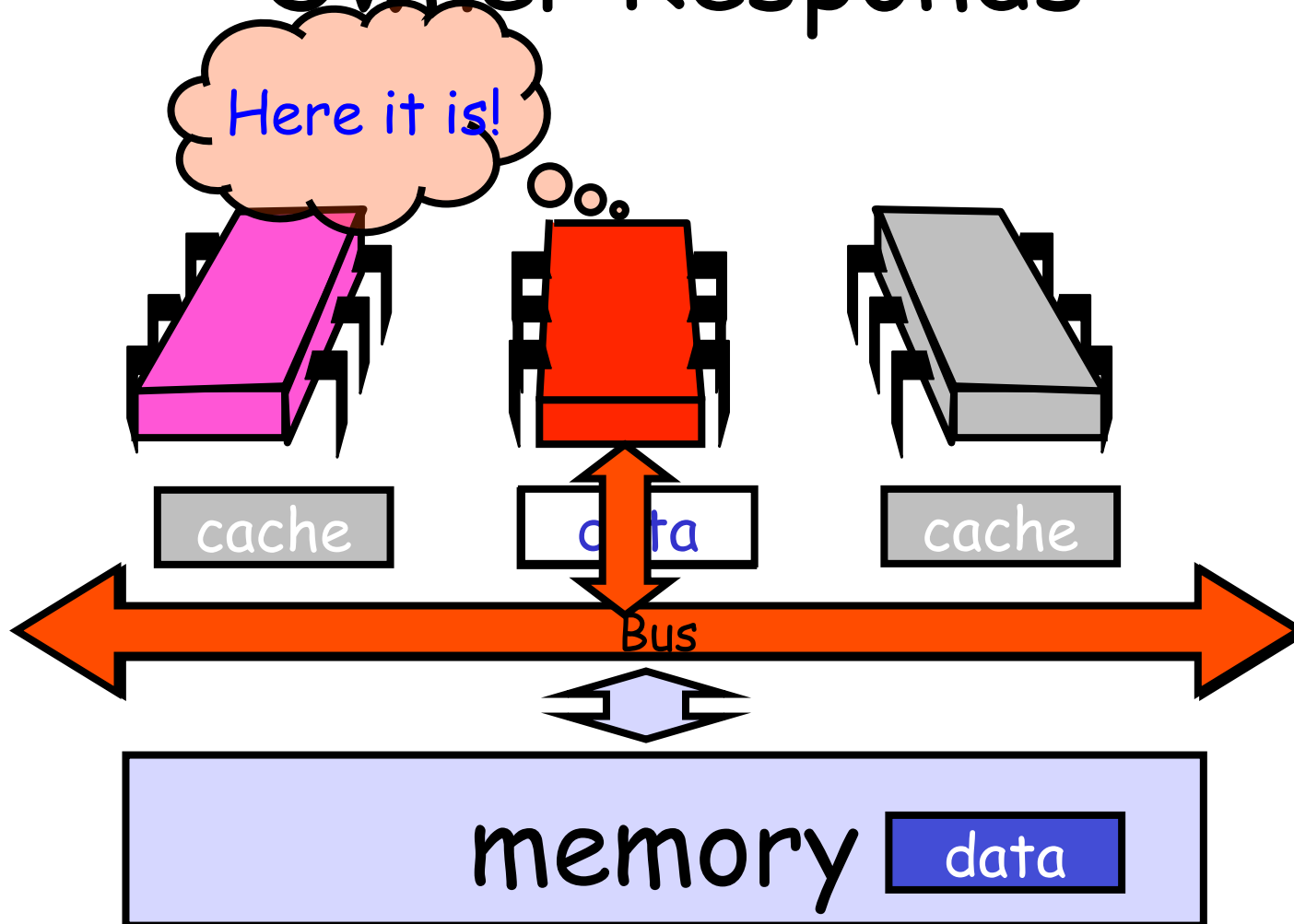
Invalidate



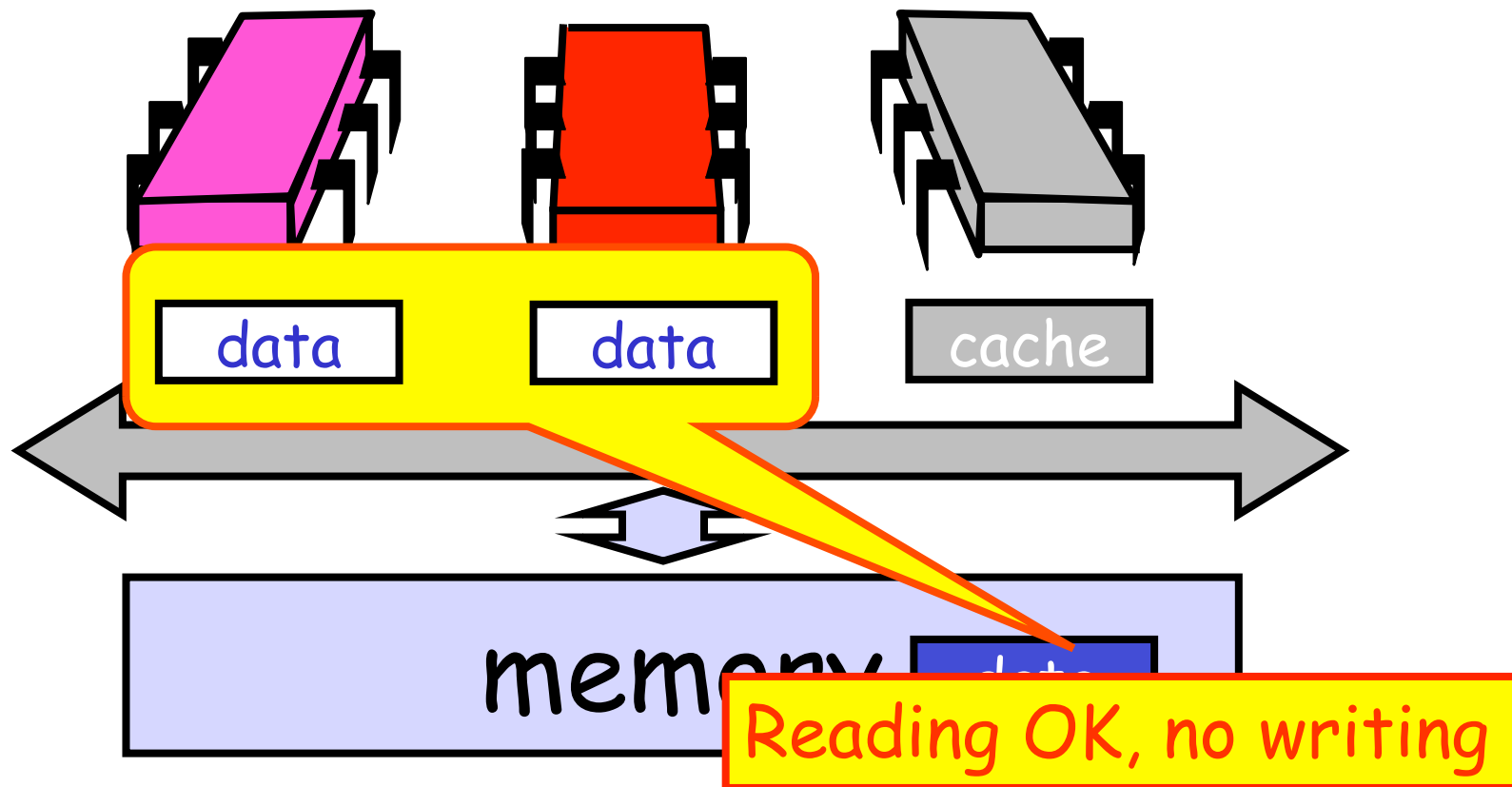
Another Processor Asks for Data



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

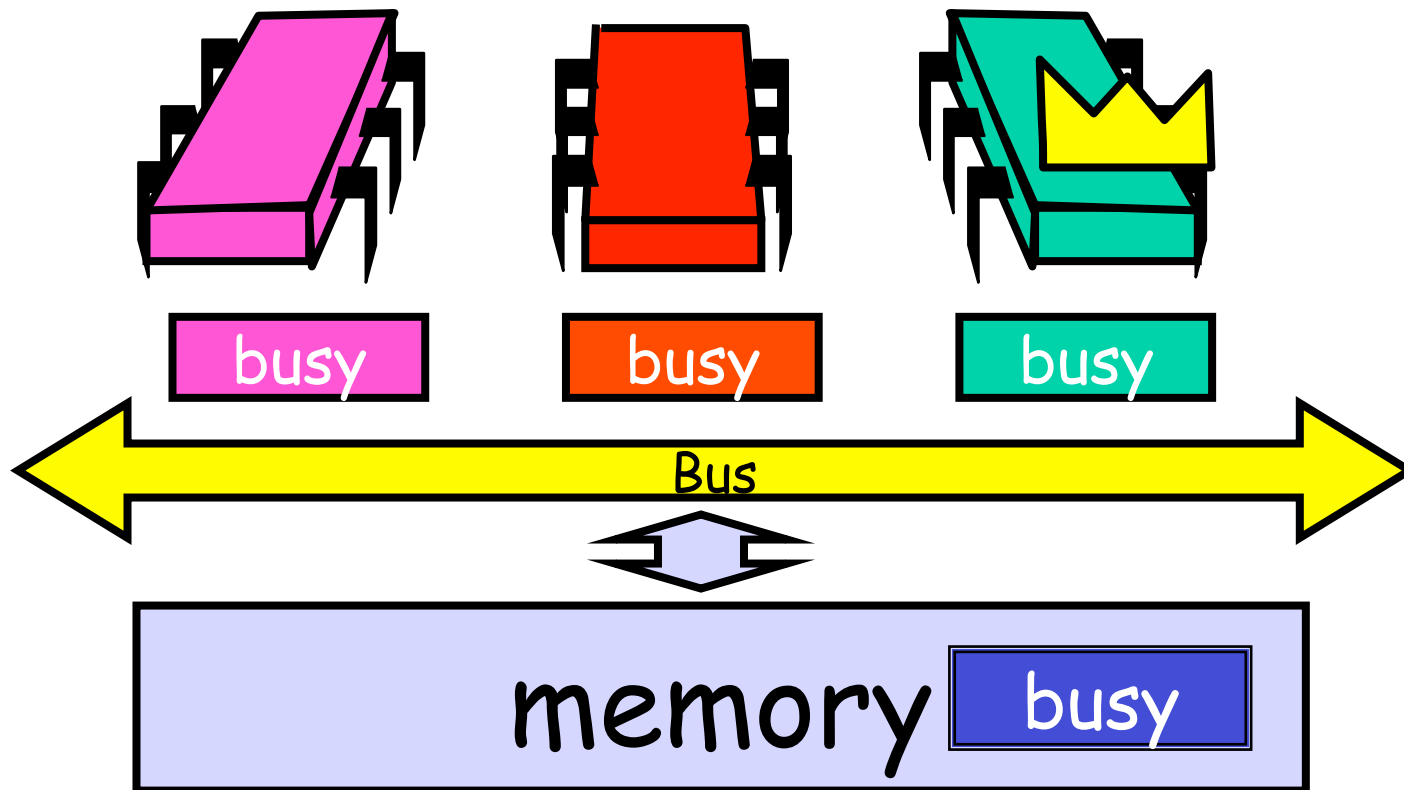
Simple TASLock

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

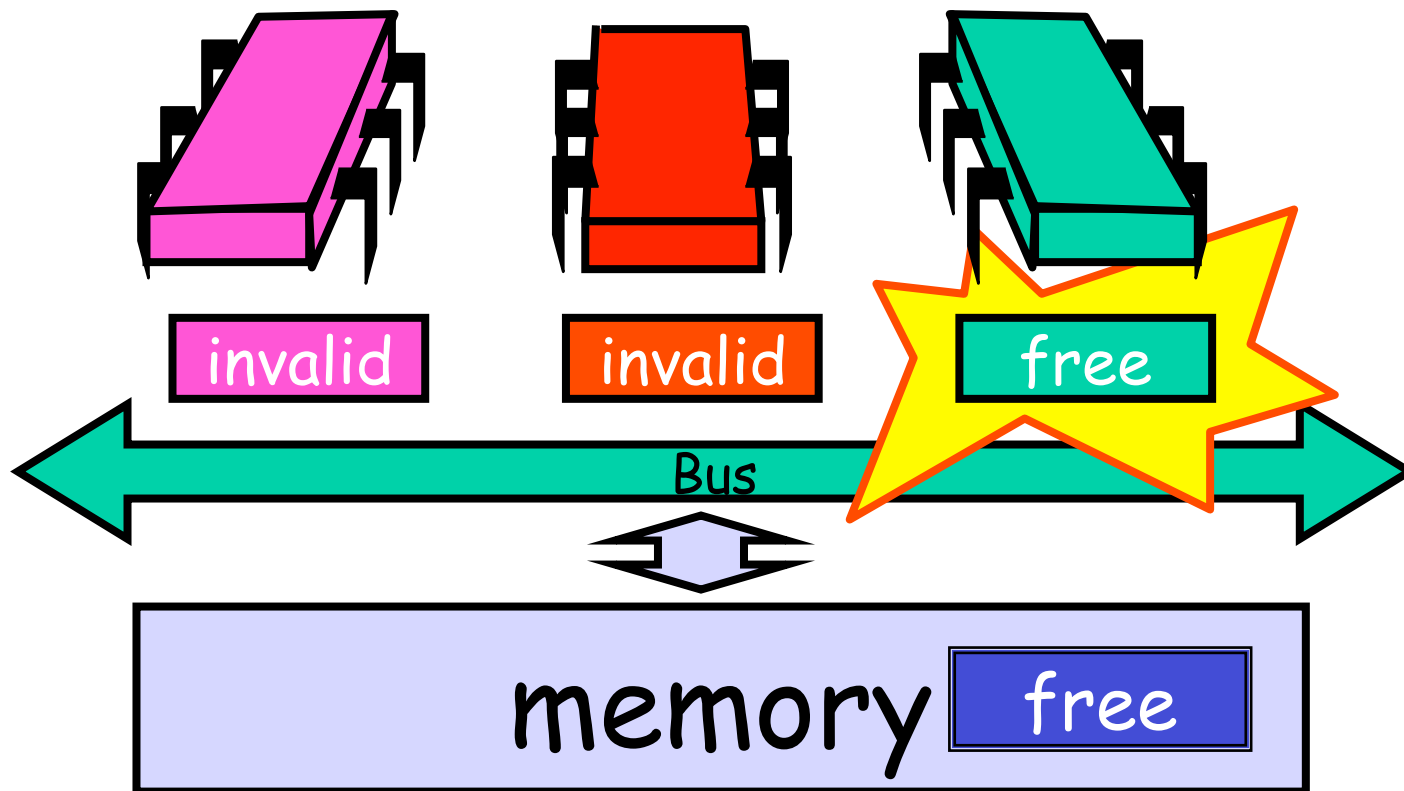
Test-and-test-and-set

- 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

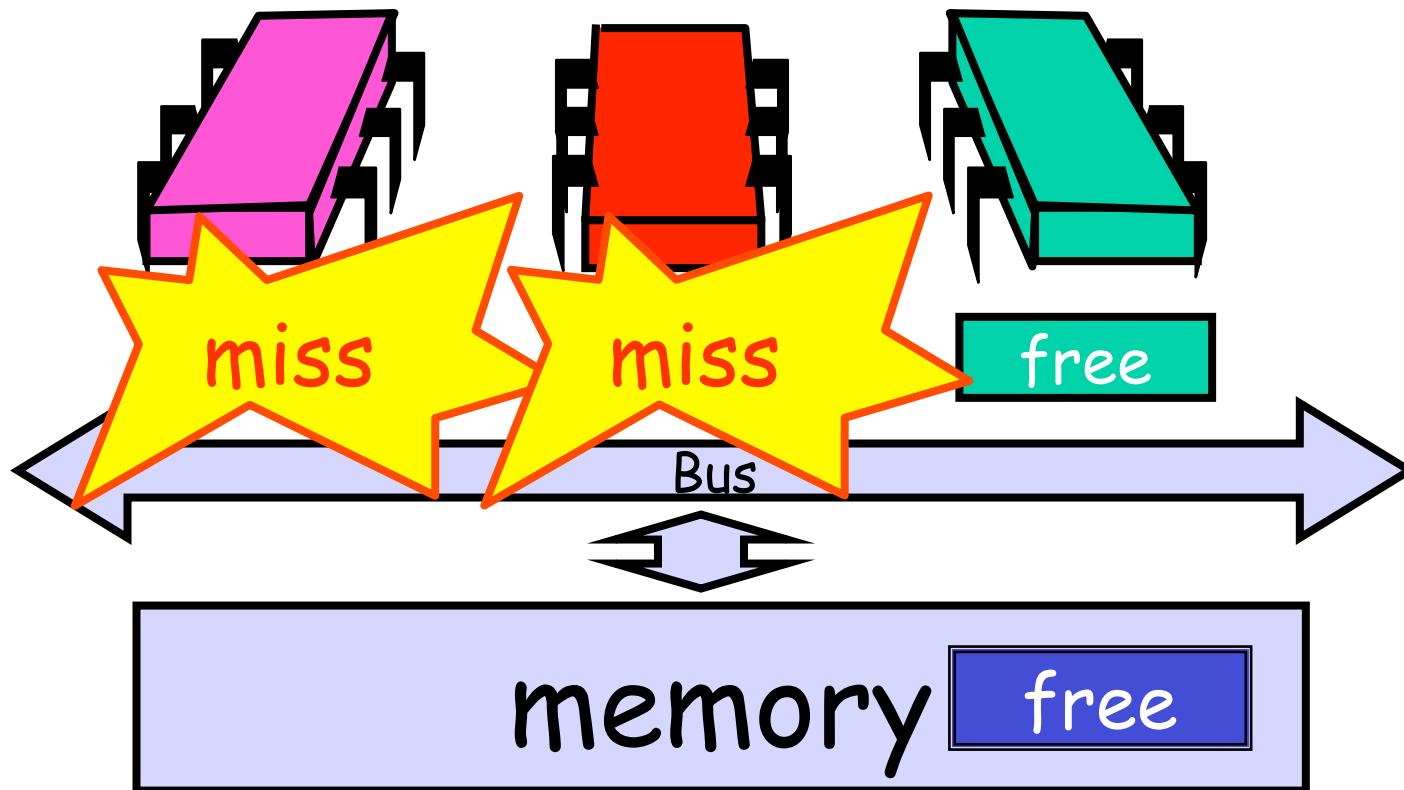


On Release



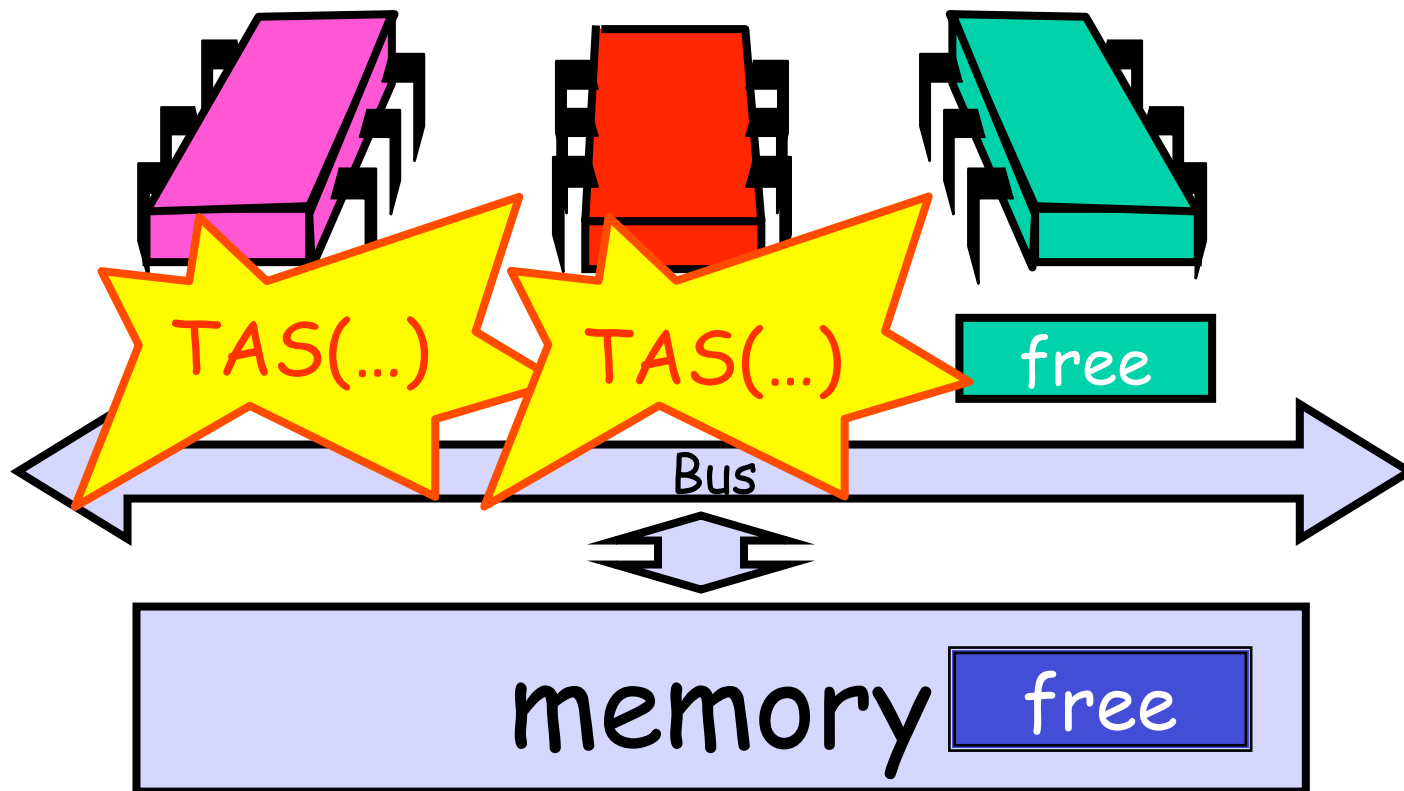
On Release

Everyone misses,
rereads



On Release

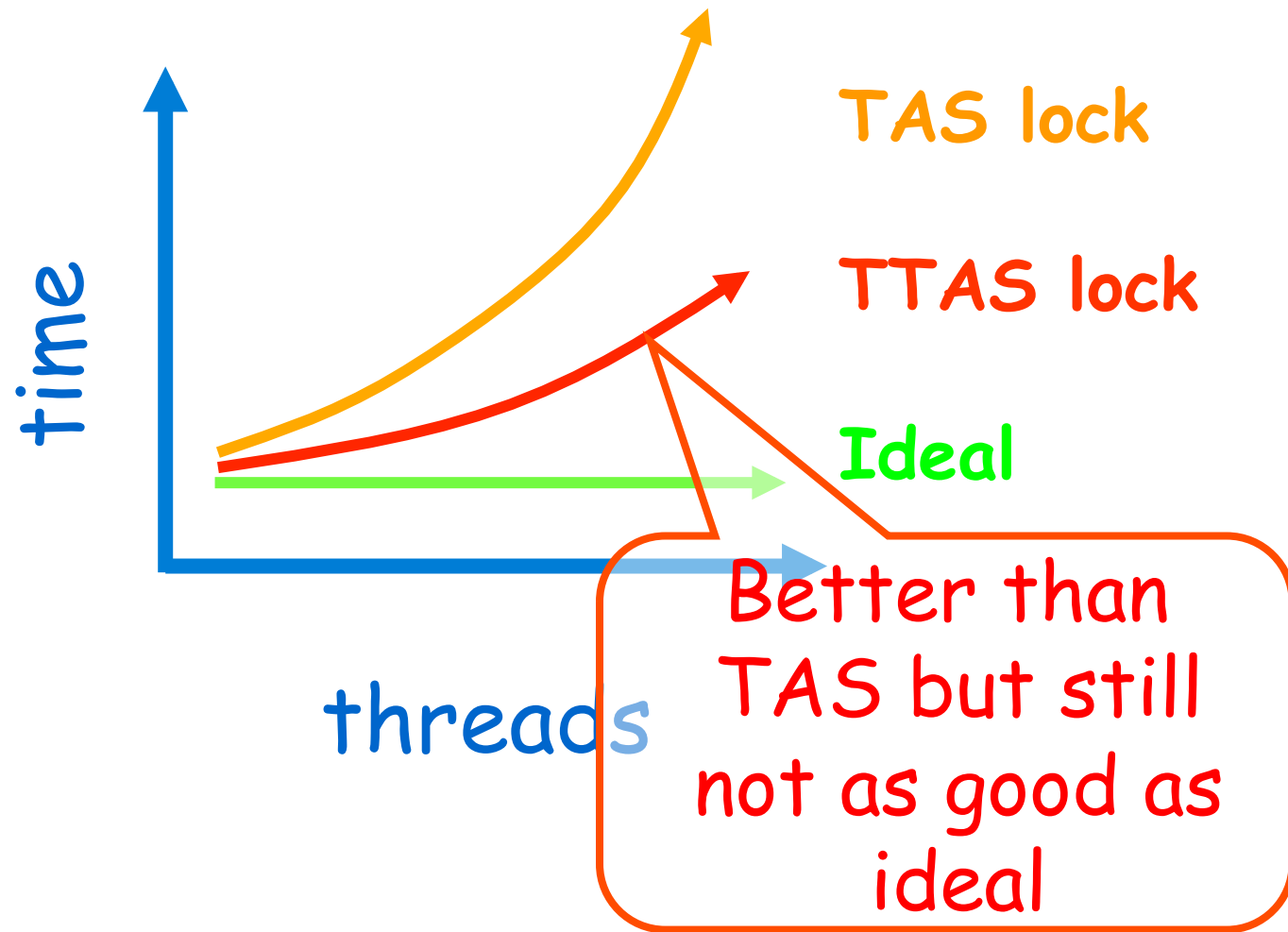
Everyone tries TAS



Problems

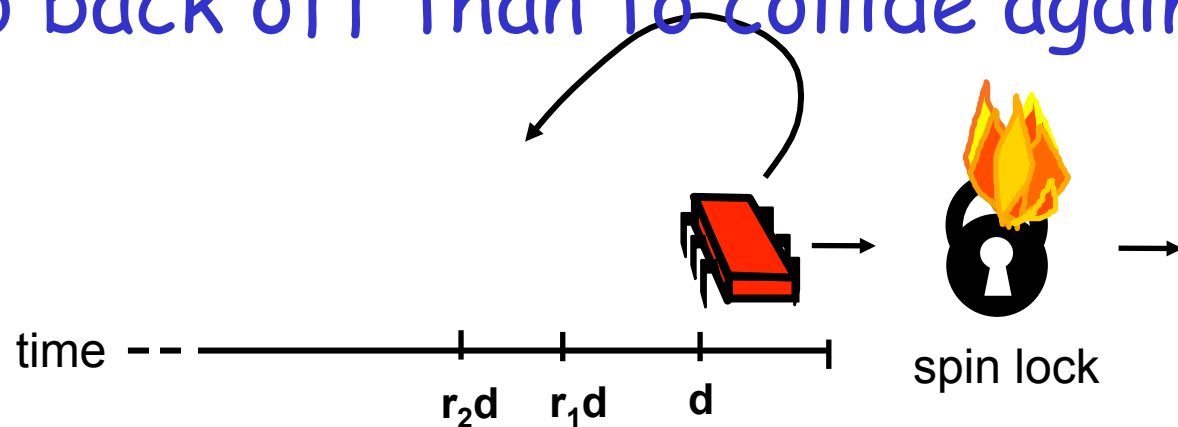
- Everyone misses
 - Reads satisfied sequentially
- Everyone does TAS
 - Invalidates others' caches
- Eventually quiesces after lock acquired
 - How long does this take?

Mystery Explained

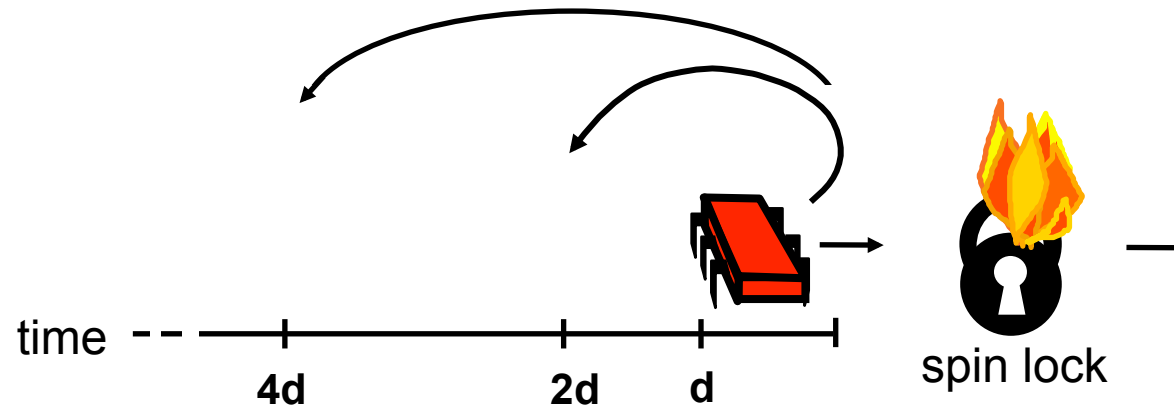


Solution: Introduce Delay

- 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

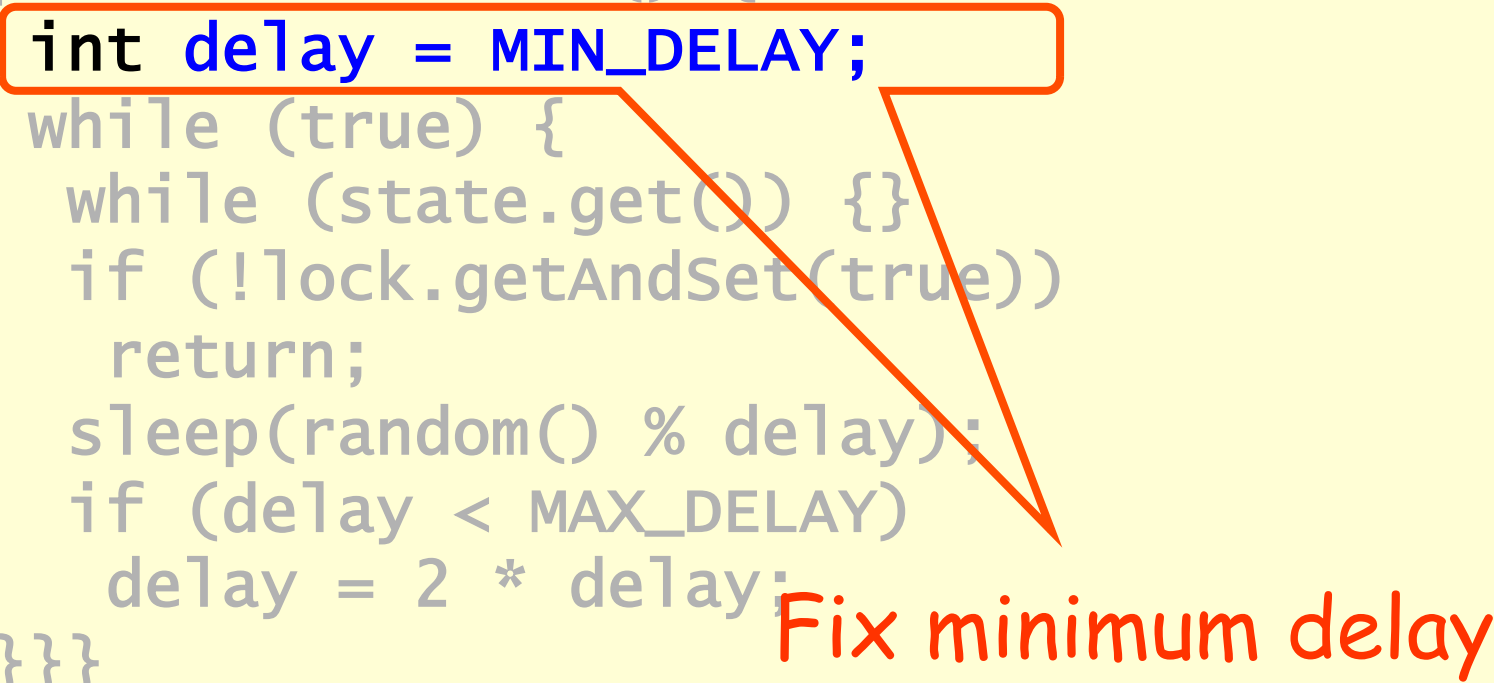
- wait random duration before retry
- Each subsequent failure doubles expected wait

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

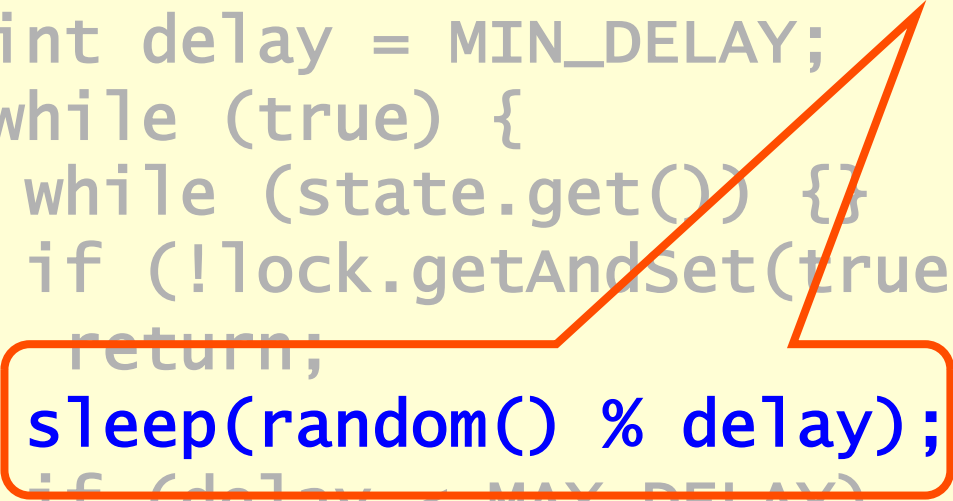
```
public class Backoff implements Lock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get()) {}  
            if (!lock.getAndSet(true))  
                return;  
            sleep(random() % delay);  
            if (delay < MAX_DELAY)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

If we win, return

Exponential Backoff Lock

```
public class ExponentialBackoffLock {  
    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

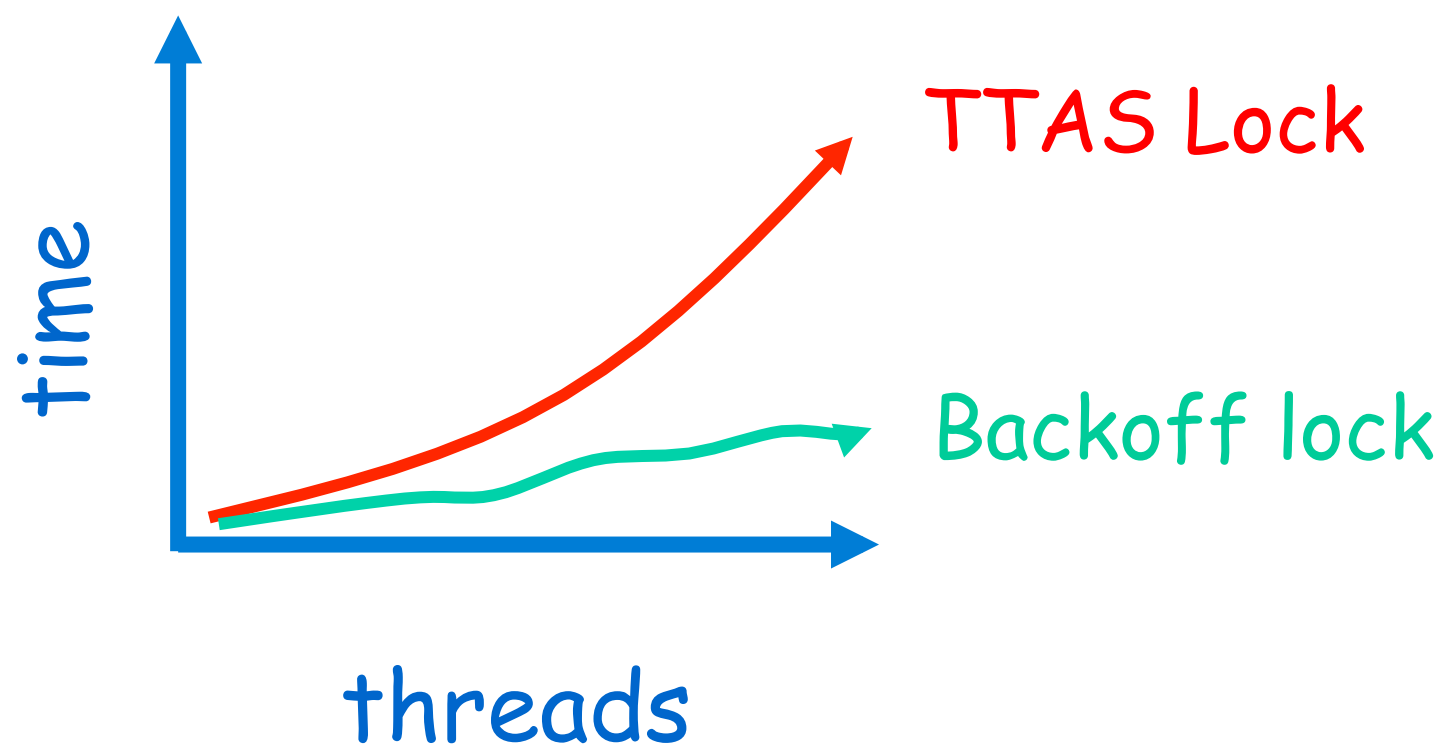


Exponential Backoff Lock

```
public class ExponentialBackoffLock {  
    public void lock() {  
        int delay = MIN_DELAY;  
        while (true) {  
            while (state.get() != LOCKED) {}  
            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



This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
 - **to Share** — to copy, distribute and transmit the work
 - **to Remix** — to adapt the work
- **Under the following conditions:**
 - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
 - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
 - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

exercícios - para 23/8

- analisar o código no slide 62 (alg Peterson) e discutir se há diferença se trocarmos as linhas 2 e 3;
- fazer um programa com threads, com pthreads+c ou Java, que tenha comportamento diferente do esperado quando se usam n threads;
- implementar um dos algoritmos de lock vistos na aula de hoje no mesmo programa.

mandar por email para noemi@inf.puc-rio.br¹⁹³