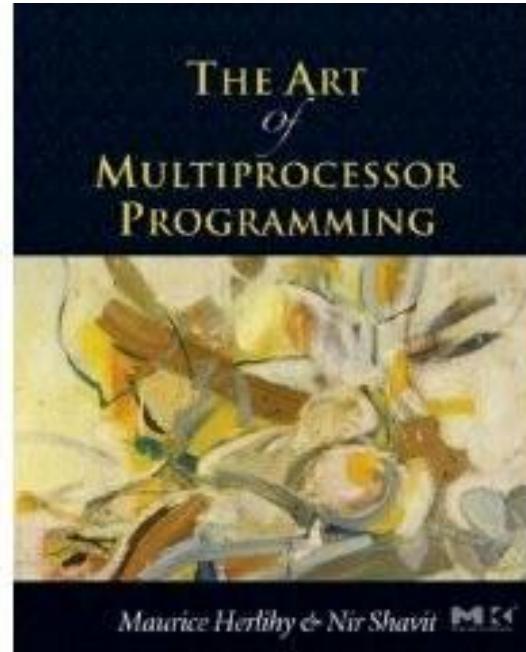


COS 226

# Concurrency

## Chapter 1

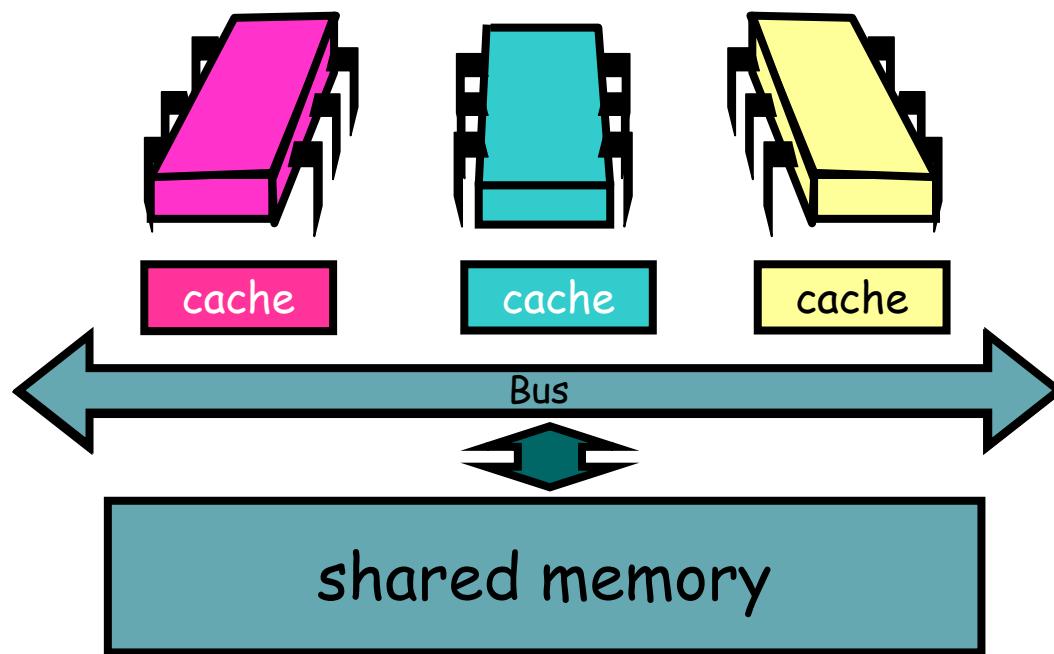
# Acknowledgement



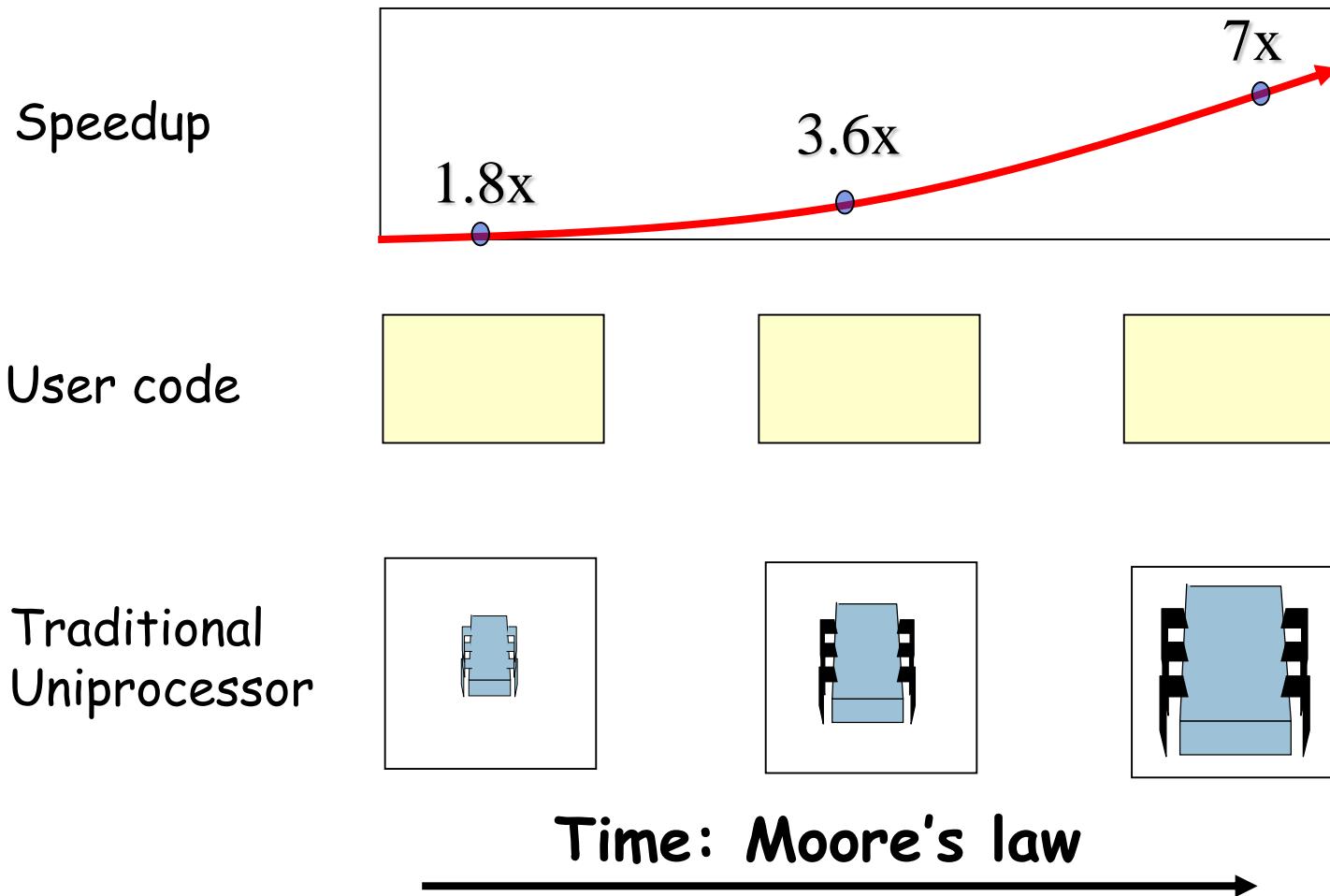
- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



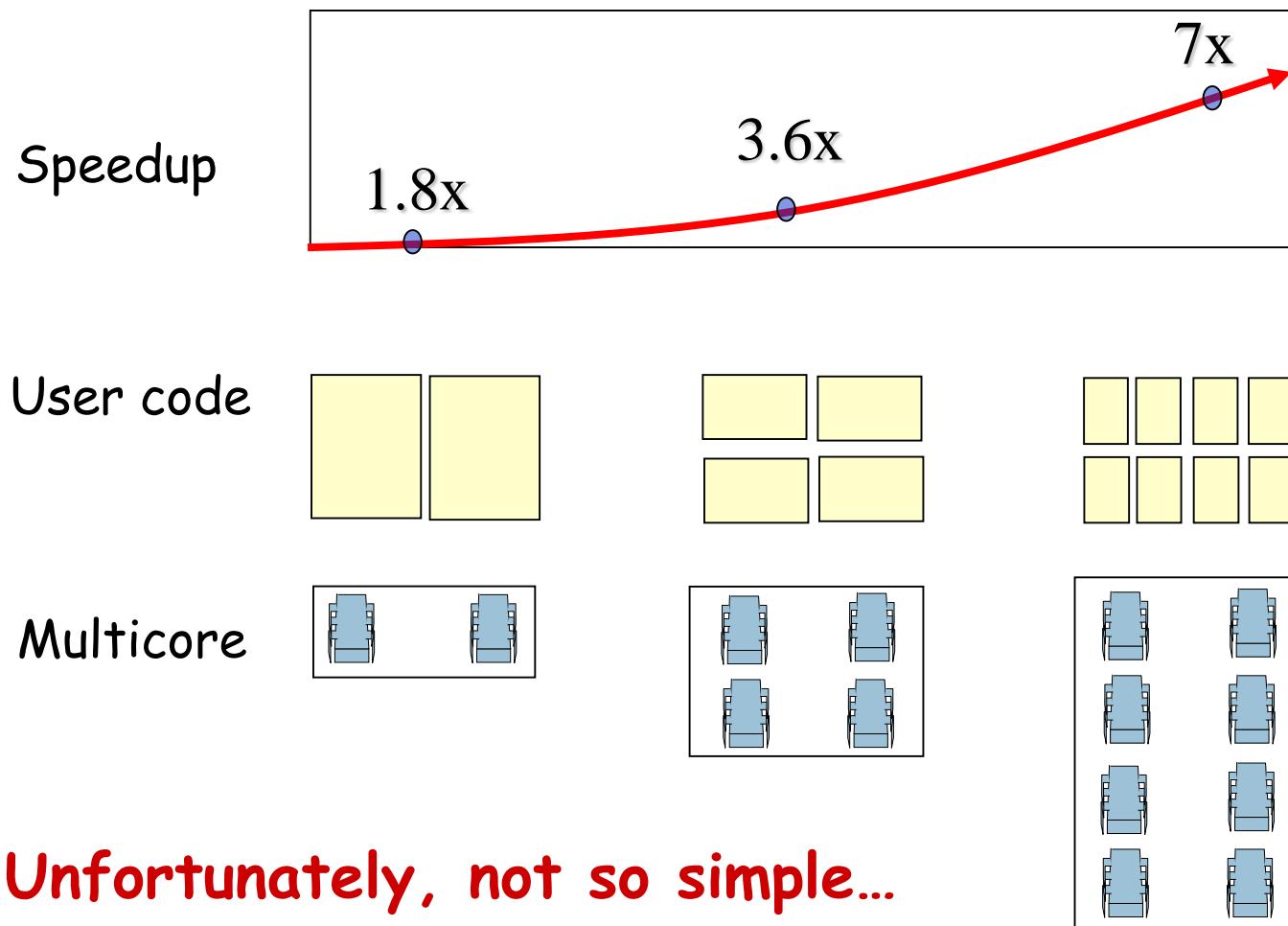
# The Shared Memory Multiprocessor (SMP)



# Traditional Scaling Process

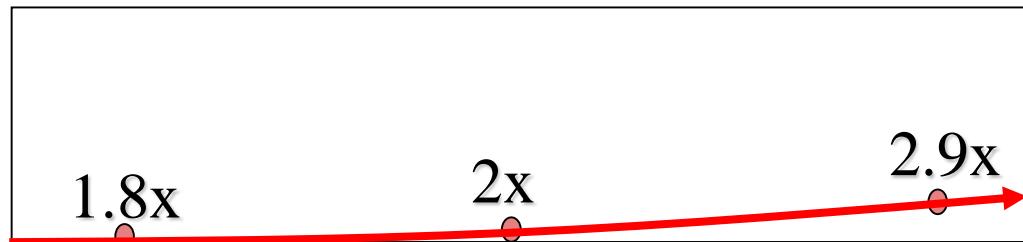


# Multicore Scaling Process

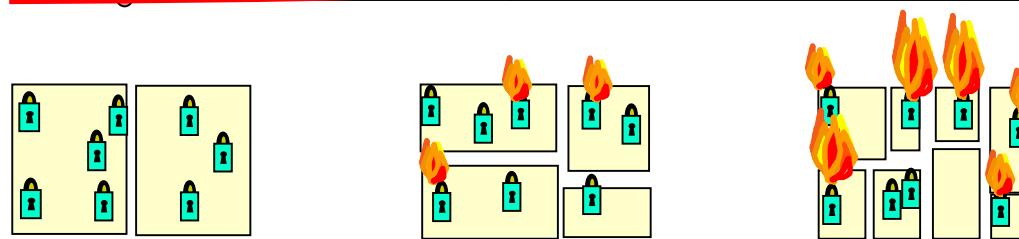


# Real-World Scaling Process

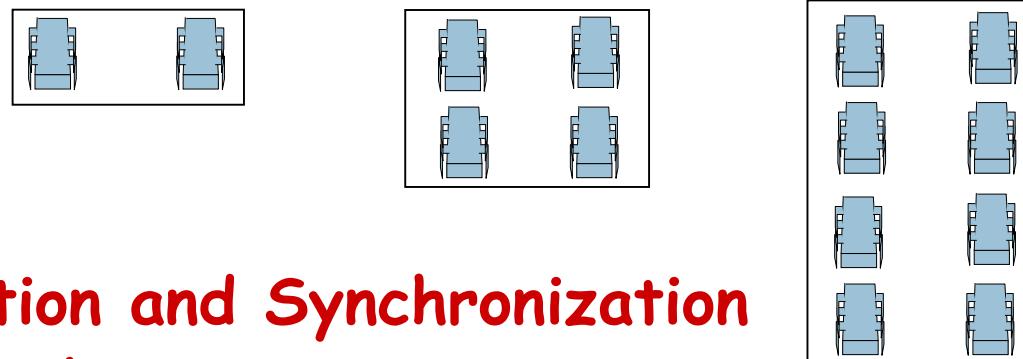
Speedup



User code



Multicore



Parallelization and Synchronization  
require great care...



# Multiprocessor programming

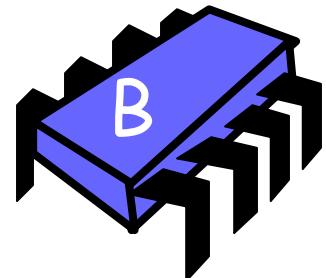
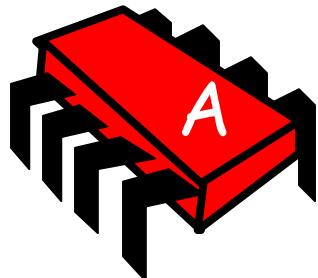
- We look at concurrency from two directions:
  - Principles
    - Computability
  - Practice
    - Performance



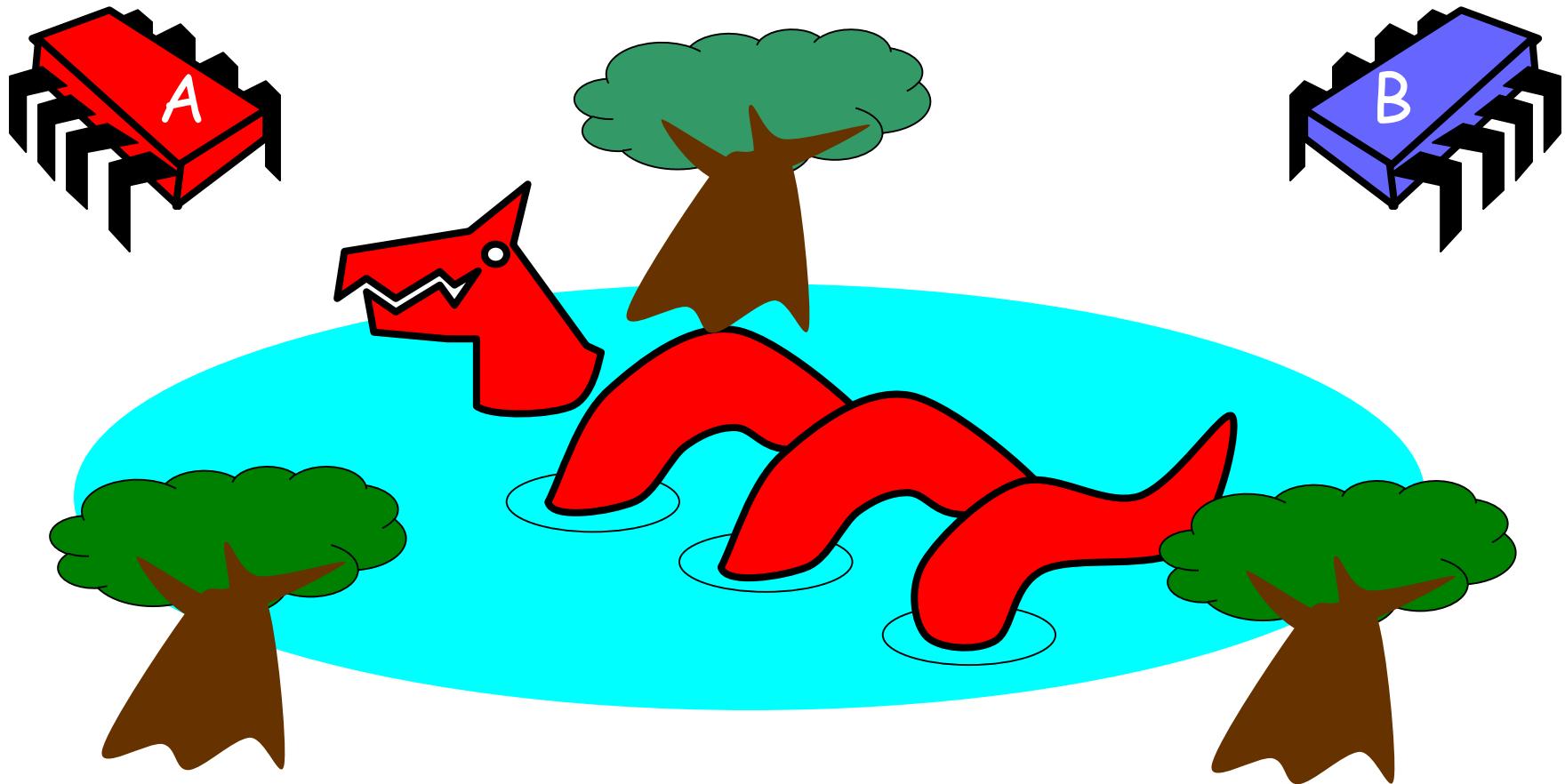
# Model Summary

- Multiple threads
  - Sometimes called processes
- Single shared memory
- Objects live in memory
- Unpredictable asynchronous delays

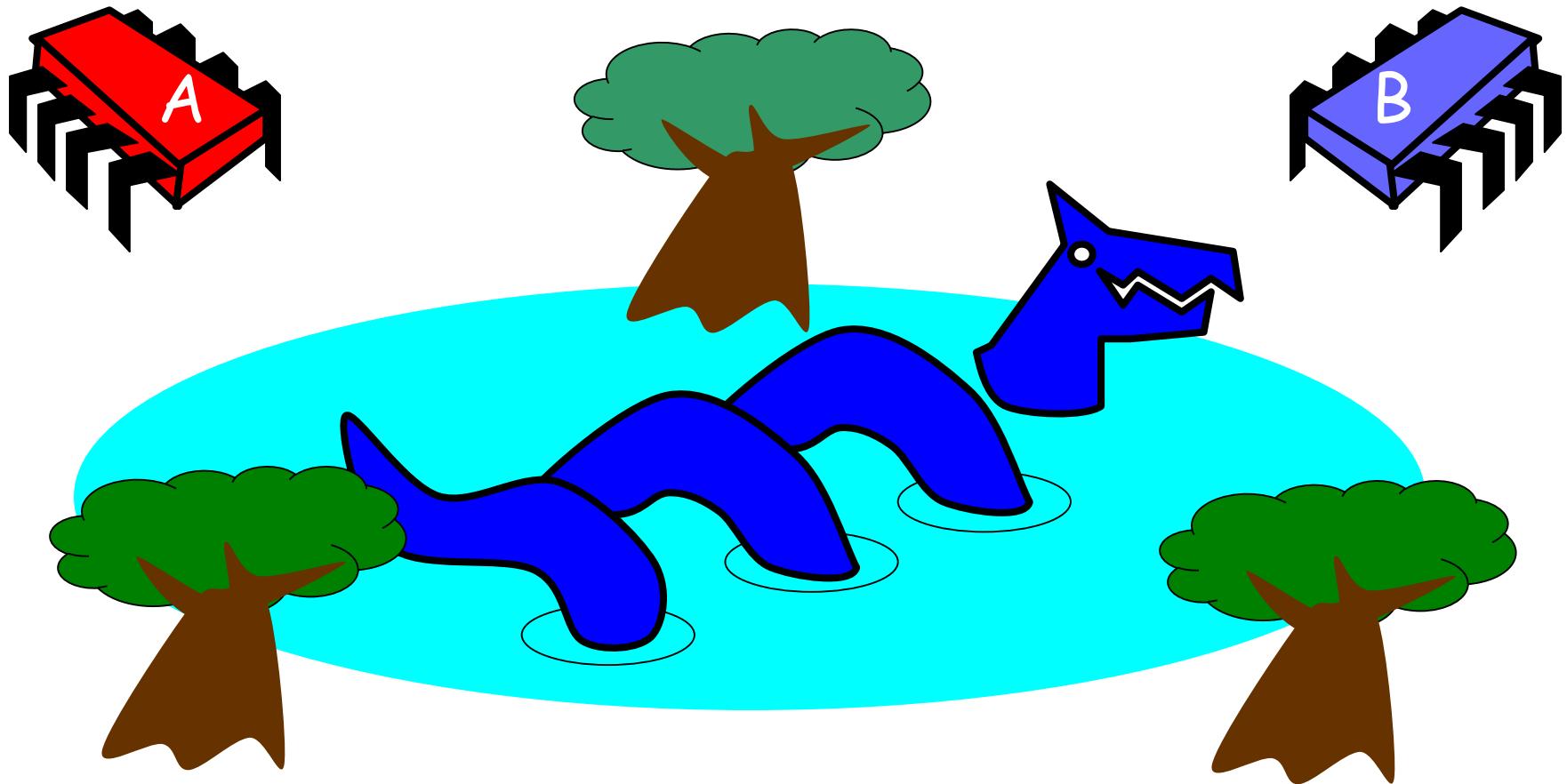
# Mutual Exclusion or “Alice & Bob share a pond”



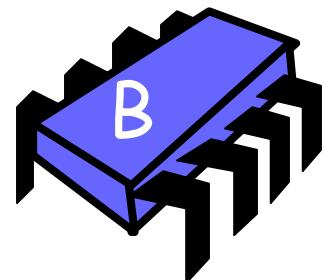
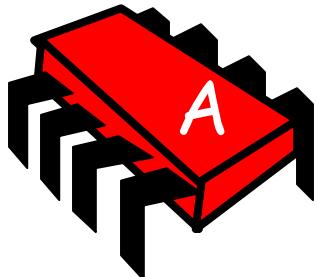
# Alice has a pet



# Bob has a pet



# The Problem



The pets don't  
get along



# Formalizing the problem

- First:
- Both pets should never be in pond at the same time
  - Mutual exclusion
  - This is a **safety** property – makes sure that nothing bad happens

# And...

- If only one wants in, it gets in, but if both want in, only one gets in.
  - No deadlock
  - This is a *liveness* property – makes sure that something good happens eventually



# Simple Protocol

- A possible solution
  - Just look at the pond and see if the coast is clear
- Problem
  - Trees obscure the view



# Interpretation

- Threads can't "see" what other threads are doing
- Explicit communication required for coordination



# Cell Phone Protocol

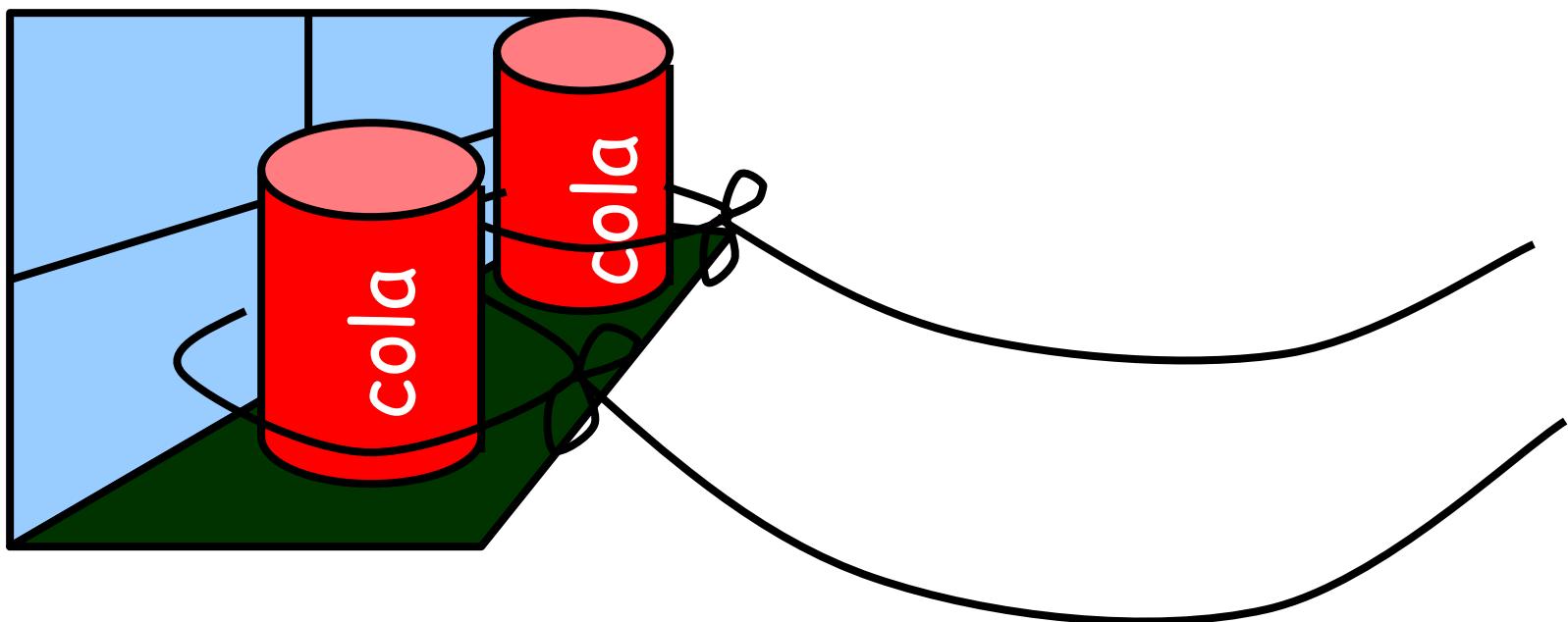
- Another possible solution
  - Bob calls Alice (or vice-versa)
- Problem
  - Bob takes shower
  - Alice recharges battery
  - Bob out shopping for pet food ...



# Interpretation

- Message-passing doesn't work
- Recipient might not be
  - Listening
  - There at all
- Communication must be
  - Persistent (like writing)
  - Not transient (like speaking)

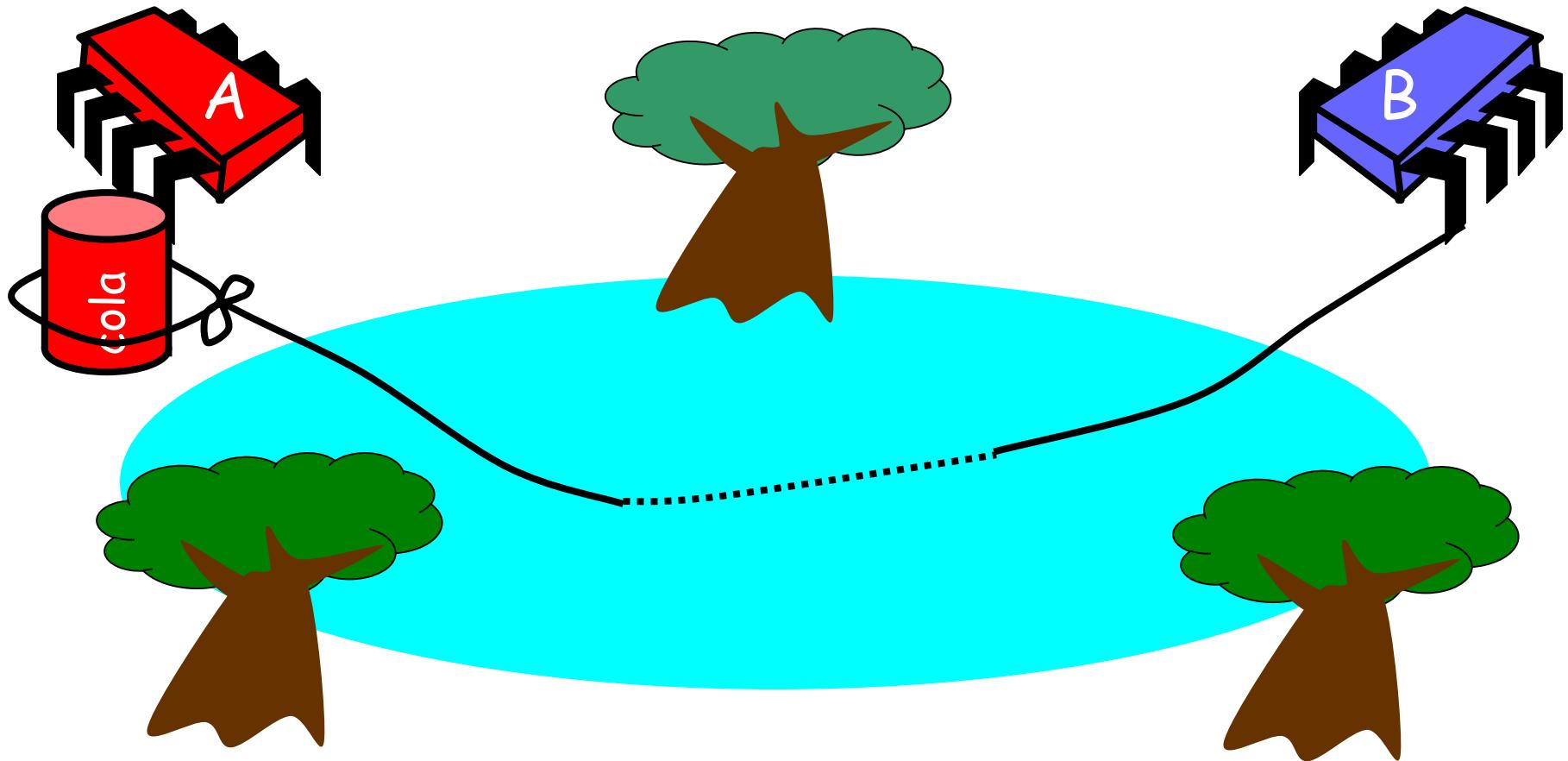
# Possible solution: Can Protocol



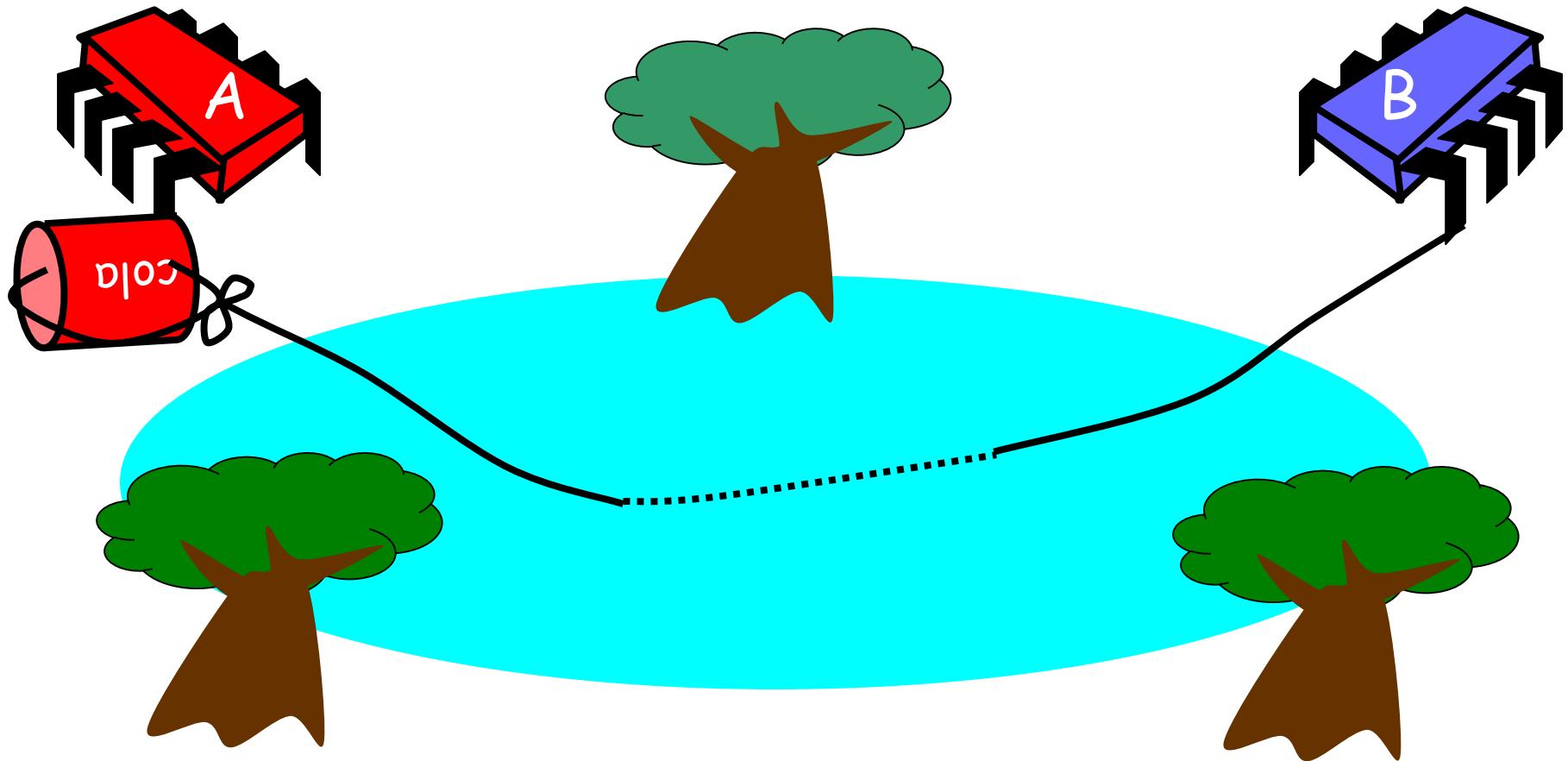
# Can Protocol

- A possible solution:
  - Bob puts one or more cans on Alice's windowsill attached to strings that lead to Bob's house
  - When he wants to send a message he knocks over one of the cans
  - When Alice sees the knocked over can, she resets them

# Bob conveys a bit



# Bob conveys a bit





# Can Protocol

## ■ Protocol

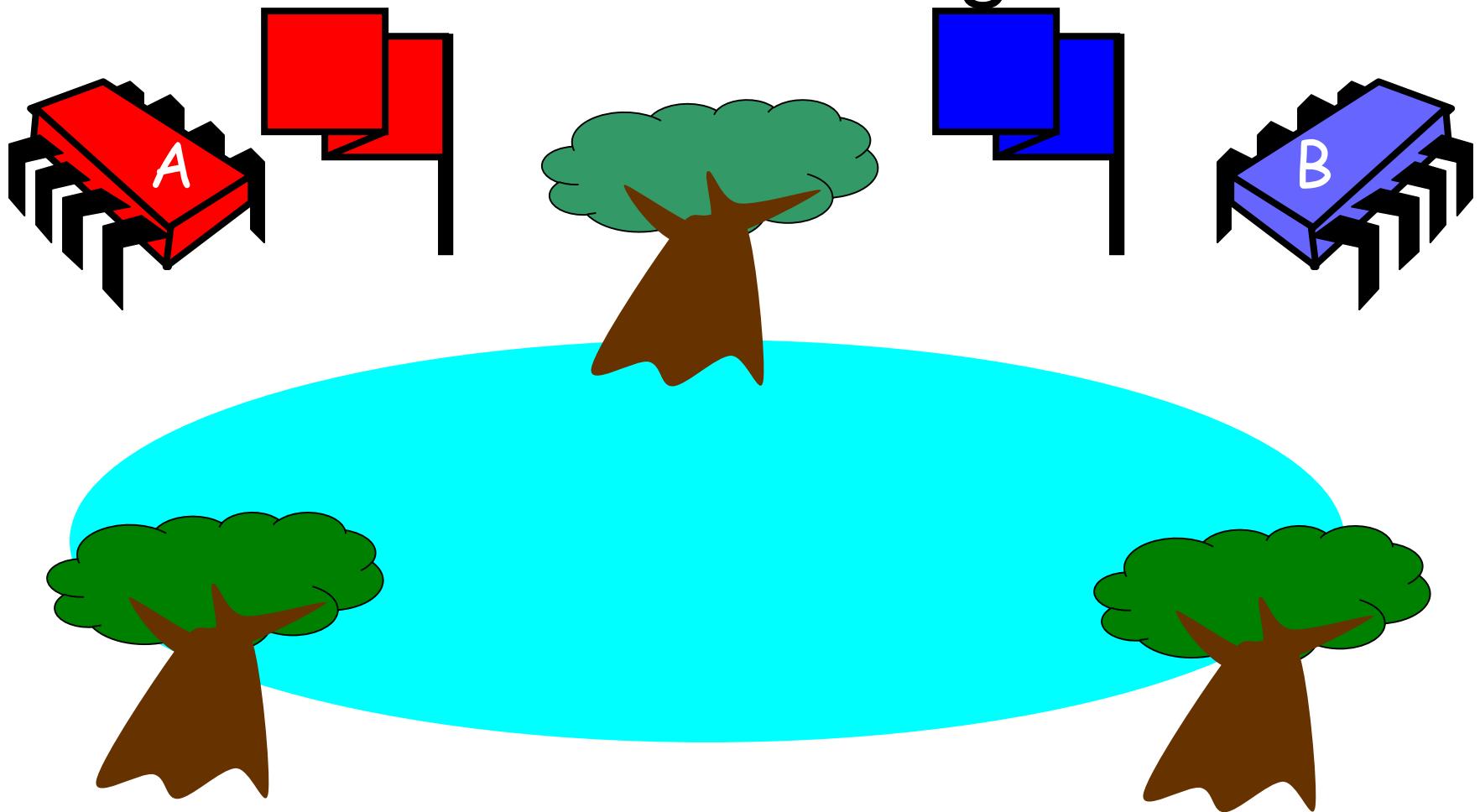
- Bob relies on Alice resetting the cans
- What if Alice goes away on holiday?
- Cans cannot be reused
- Bob runs out of cans



# Interpretation

- Cannot solve mutual exclusion with interrupts
  - Sender sets fixed bit in receiver's space
  - Receiver resets bit when ready
  - Requires infinite number of available bits

# Possible solution: Flag Protocol

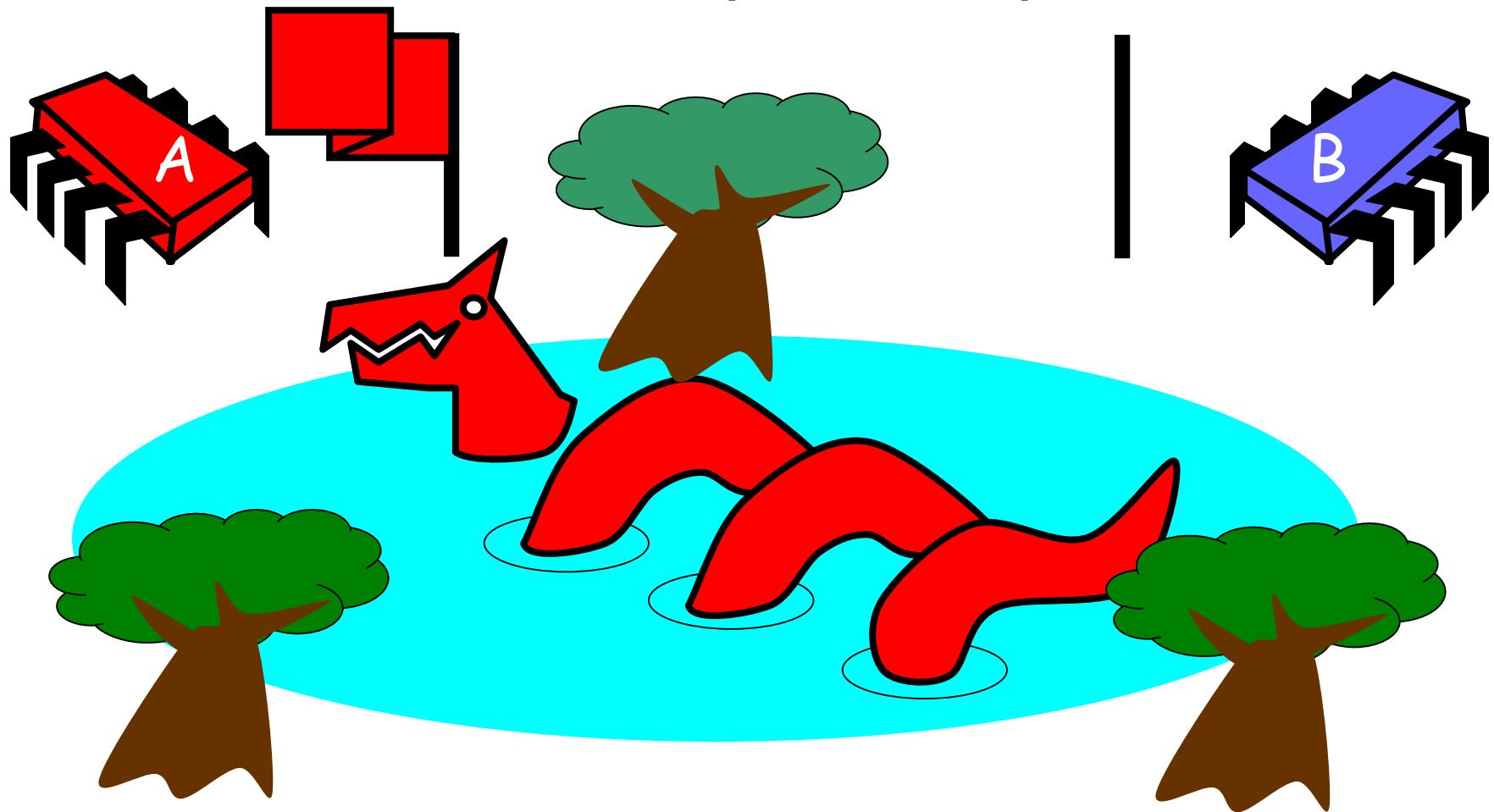




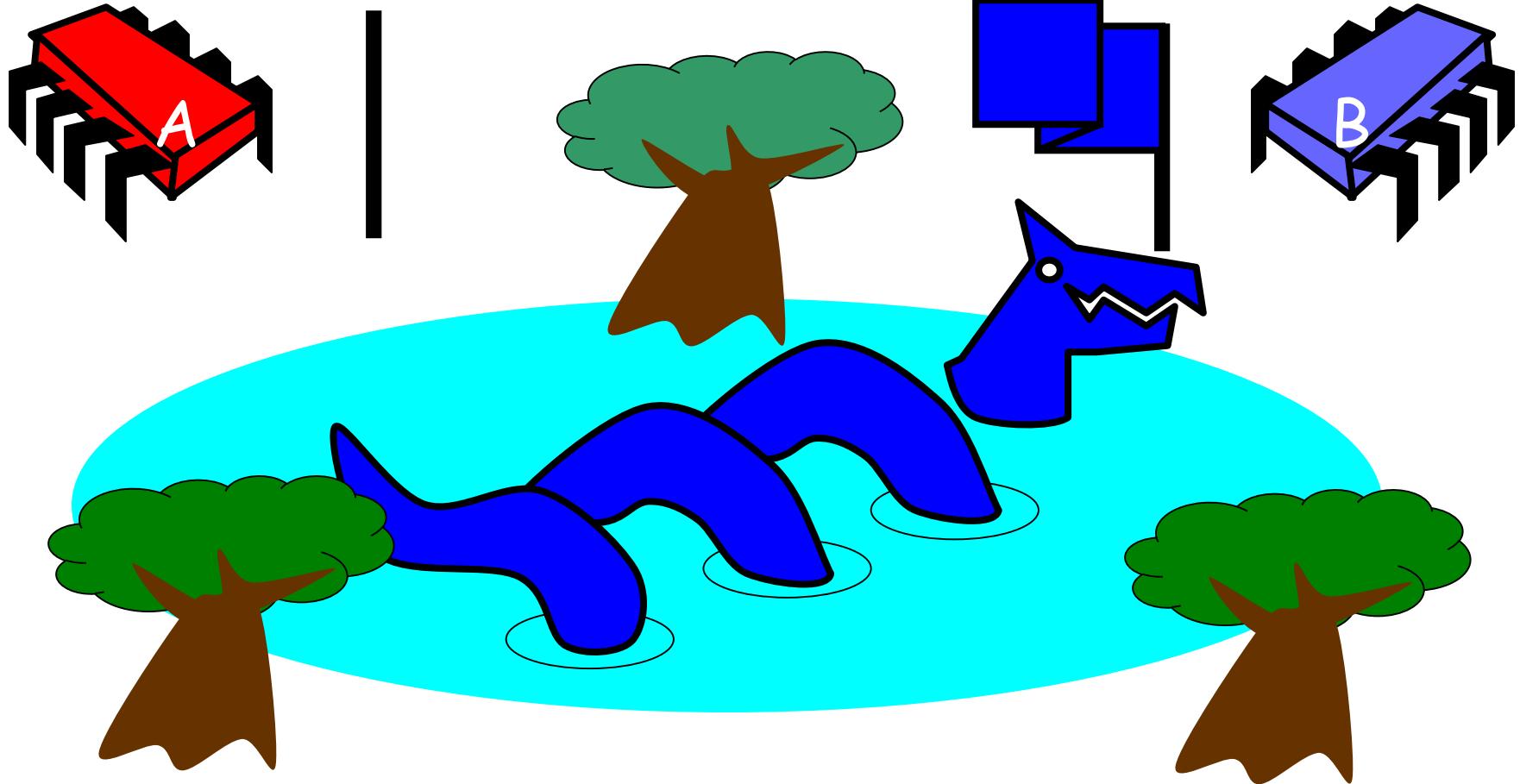
# Alice's Protocol

- If Alice wants to release her pet she raises her flag
- If Bob's flag is down, she can release her pet
- When her pet returns, she lowers her flag again

# Alice's Protocol (sort of)



# Bob's Protocol (sort of)



# Bob's Protocol

- Raise flag
- When Alice's flag is down unleash pet
- Lower flag when pet returns





# Bob's Protocol (2<sup>nd</sup> try)

- Raise flag
- While Alice's flag is up
  - Lower flag
  - Wait for Alice's flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

# Bob's Protocol

- Raise flag
- While Alice's flag is up
  - Lower flag
  - Wait for Alice's flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

**Bob defers  
to Alice**



# The Flag Principle

- Raise the flag
- Look at other's flag
- Flag Principle:
  - If each raises and looks, then
  - Last to look must see both flags up

# Does it work?

- Mutual exclusion?
  - YES
  - Pets are not in the yard at the same time
- Deadlock-freedom?
  - YES
  - If both pets want to use the yard, Bob defers to Alice



# Starvation-freedom

- If a pet wants to enter the yard, will it eventually succeed?
  - NO.
  - Whenever Alice and Bob are in conflict, Bob defers to Alice, thus it is possible that Alice's pet uses the pond over and over and Bob's pet doesn't get a turn



# Waiting

- If Alice raises her flag and suddenly becomes ill, Bob's pet cannot use the pond until Alice returns
- Bob must *wait* for Alice to lower her flag



# Remarks

- Protocol is *unfair*
  - Bob's pet might never get in
- Protocol uses *waiting*
  - If Bob is eaten by his pet, Alice's pet might never get in



# The Fable Continues

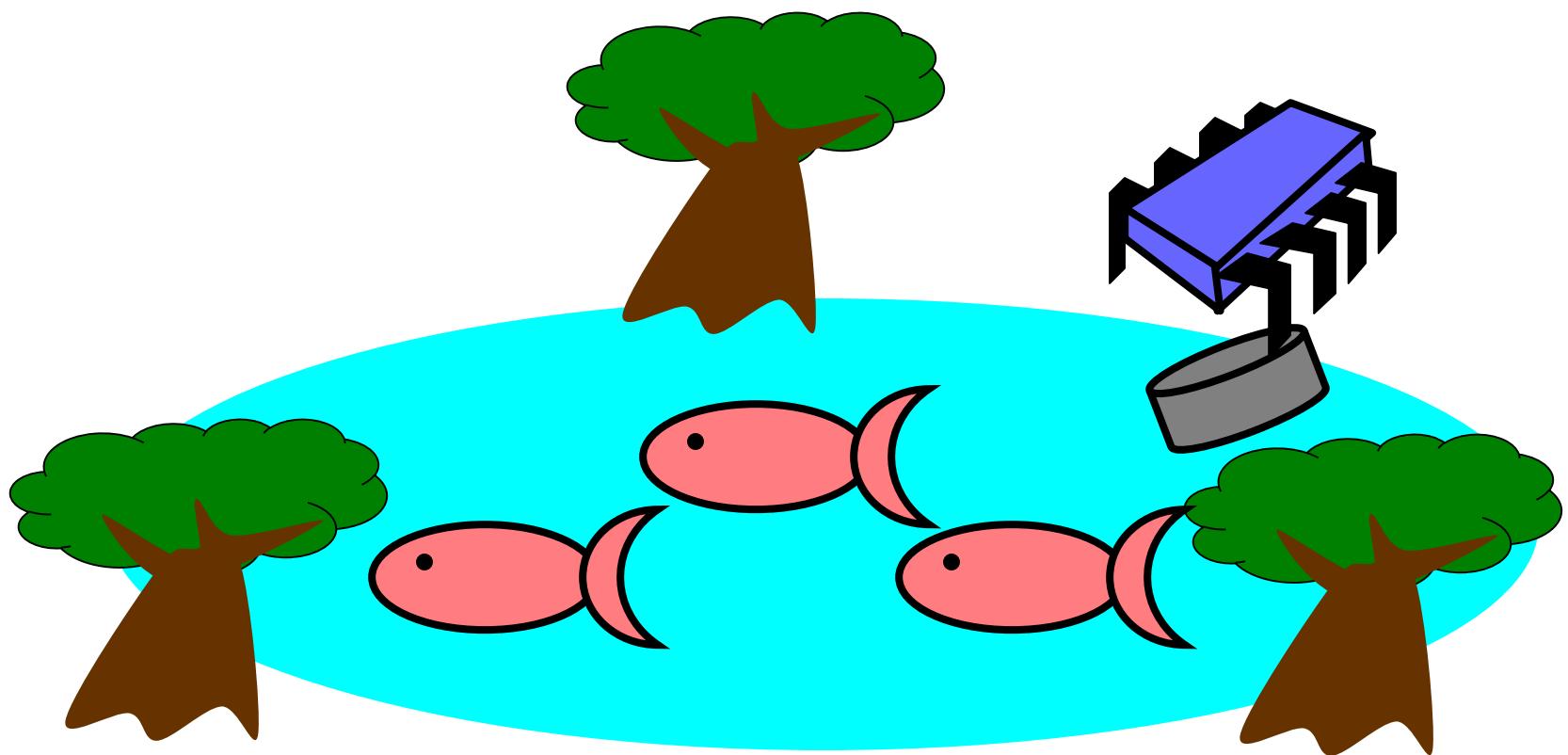
- Alice and Bob fall in love & marry
- Then they fall out of love & divorce
  - She gets the pets – they now get along
  - He has to feed them – the pets however side with Alice and attacks Bob



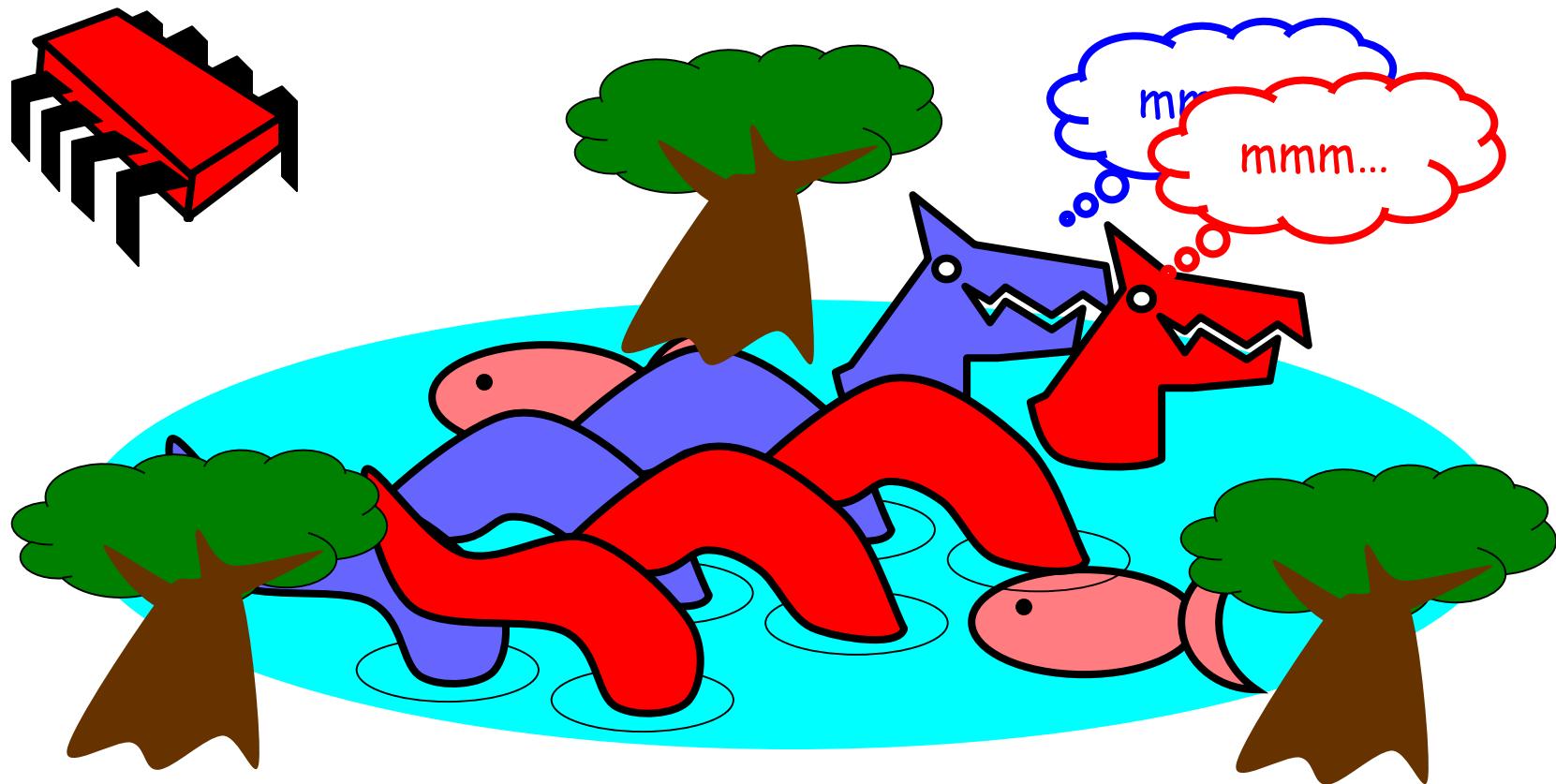
# Producer-Consumer Problem

- A new coordination problem

# Bob Puts Food in the Pond



# Alice releases her pets to Feed





# Producer/Consumer

- Alice and Bob can't meet
  - Each has restraining order on other
  - So he puts food in the pond
  - And later, she releases the pets
- Avoid
  - Releasing pets when there's no food
  - Putting out food if uneaten food remains



# Producer/Consumer

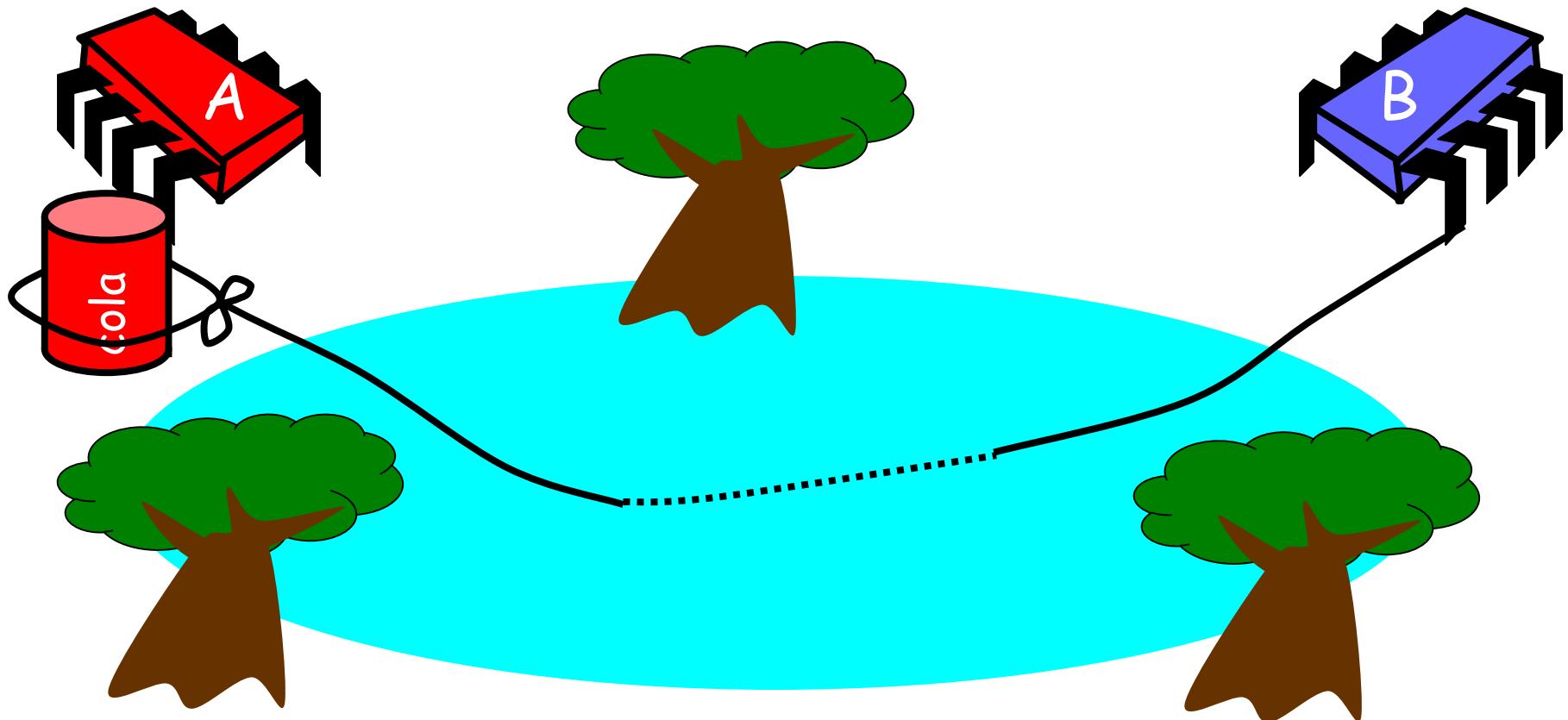
- Need a mechanism so that
  - Bob lets Alice know when food has been put out
  - Alice lets Bob know when to put out more food



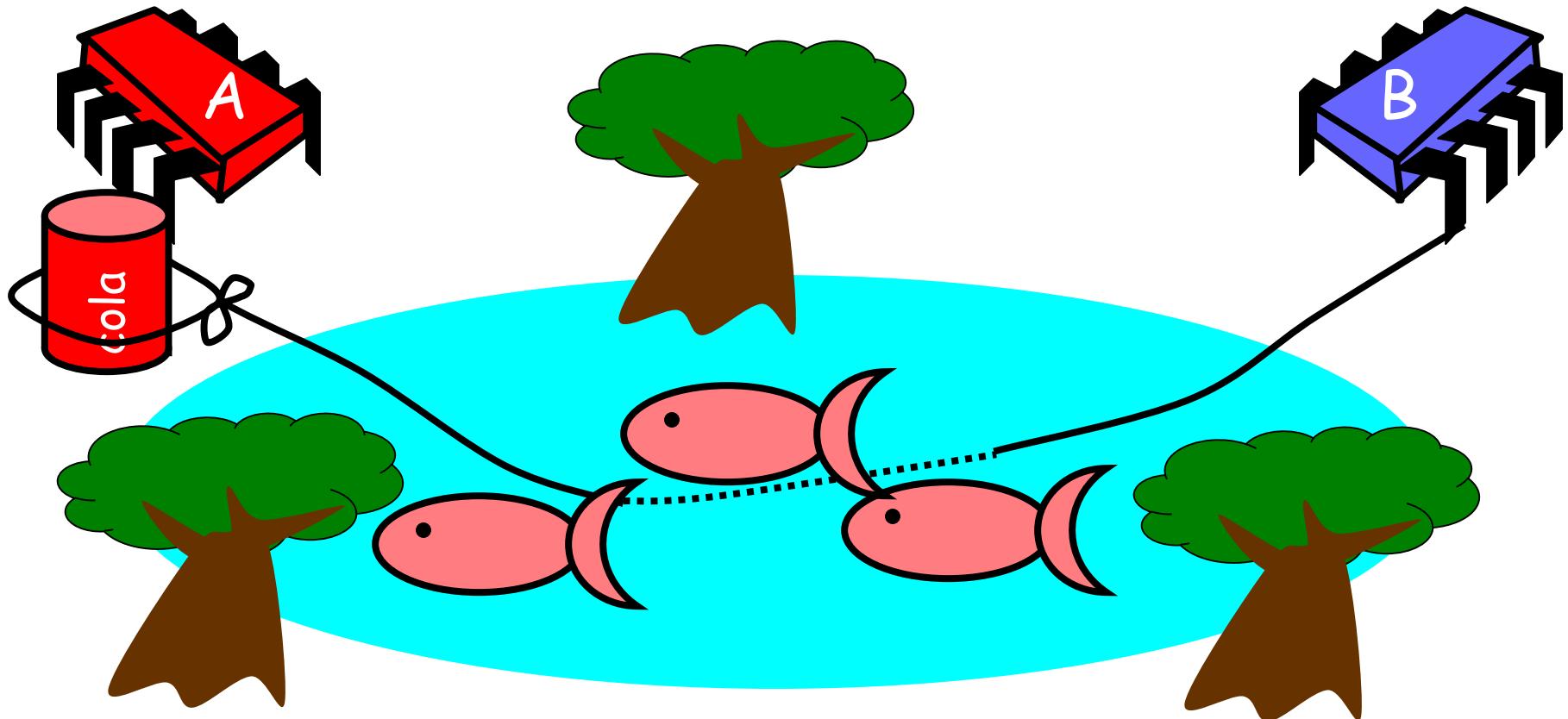
# Also known as bounded buffer problem

- Two processes – producer and consumer – share a common fixed-size buffer
- The producer generates data, puts it into the buffer and start again
- At the same time the consumer, consumes the data one piece at a time
- Problem:
  - Producer should not try to add data if the buffer is full
  - Consumer should not try to remove data from an empty buffer

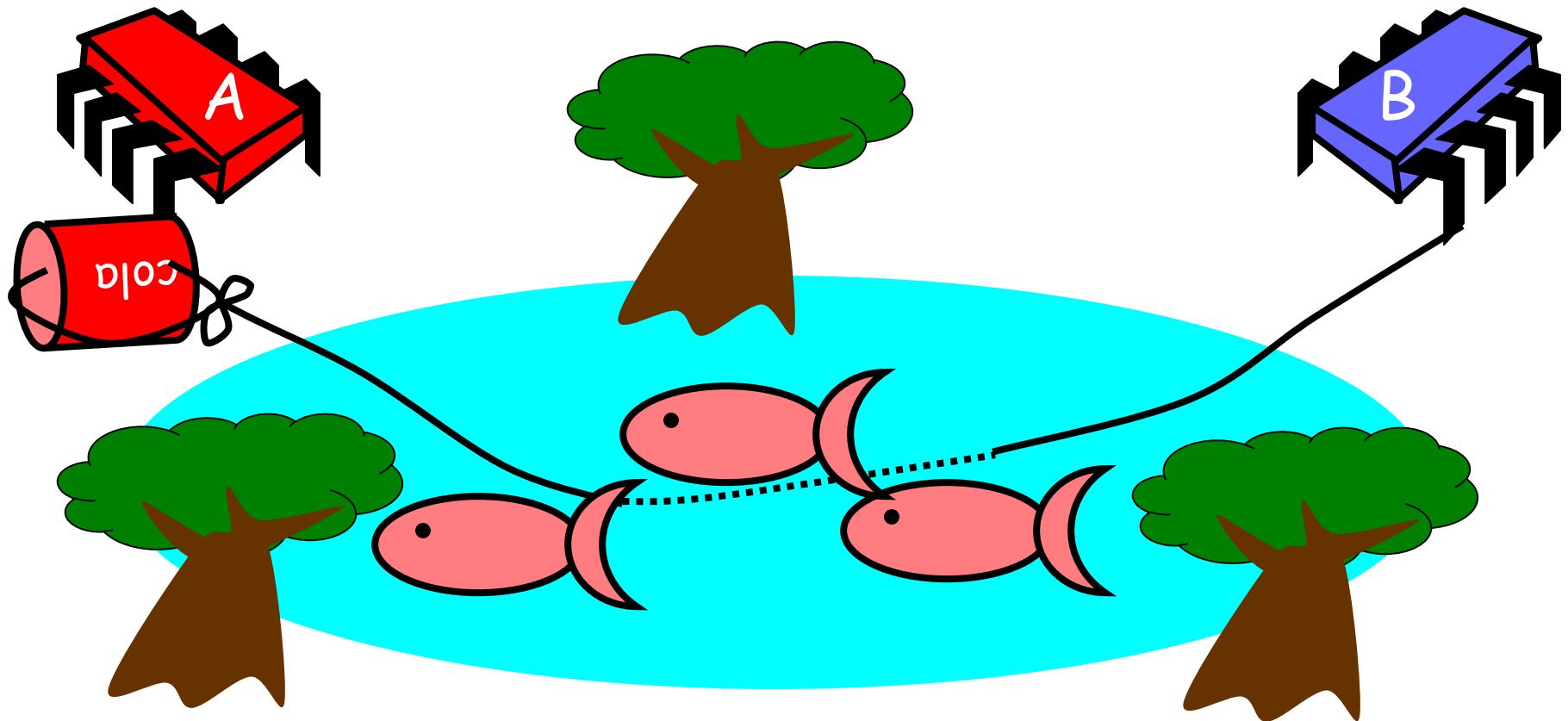
# Surprise Solution



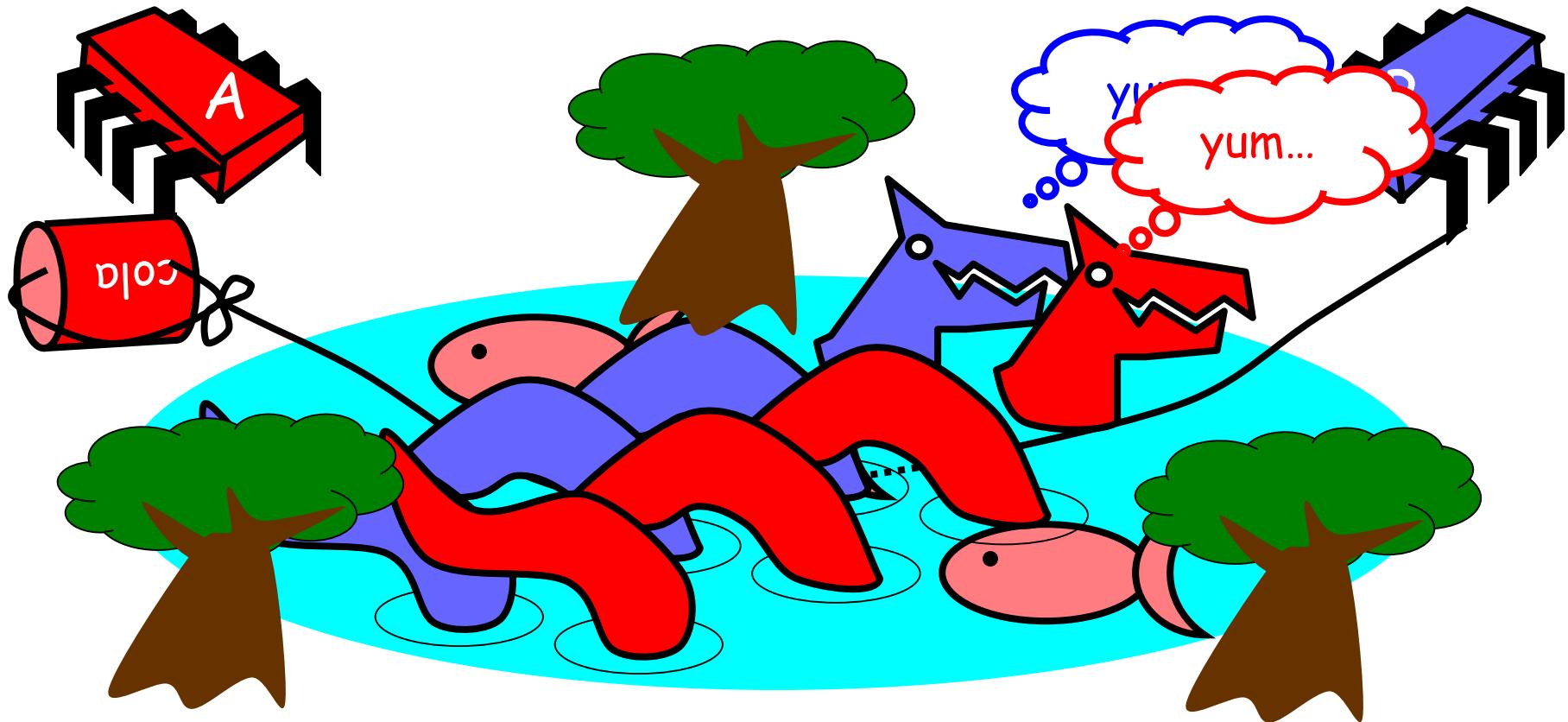
# Bob puts food in Pond



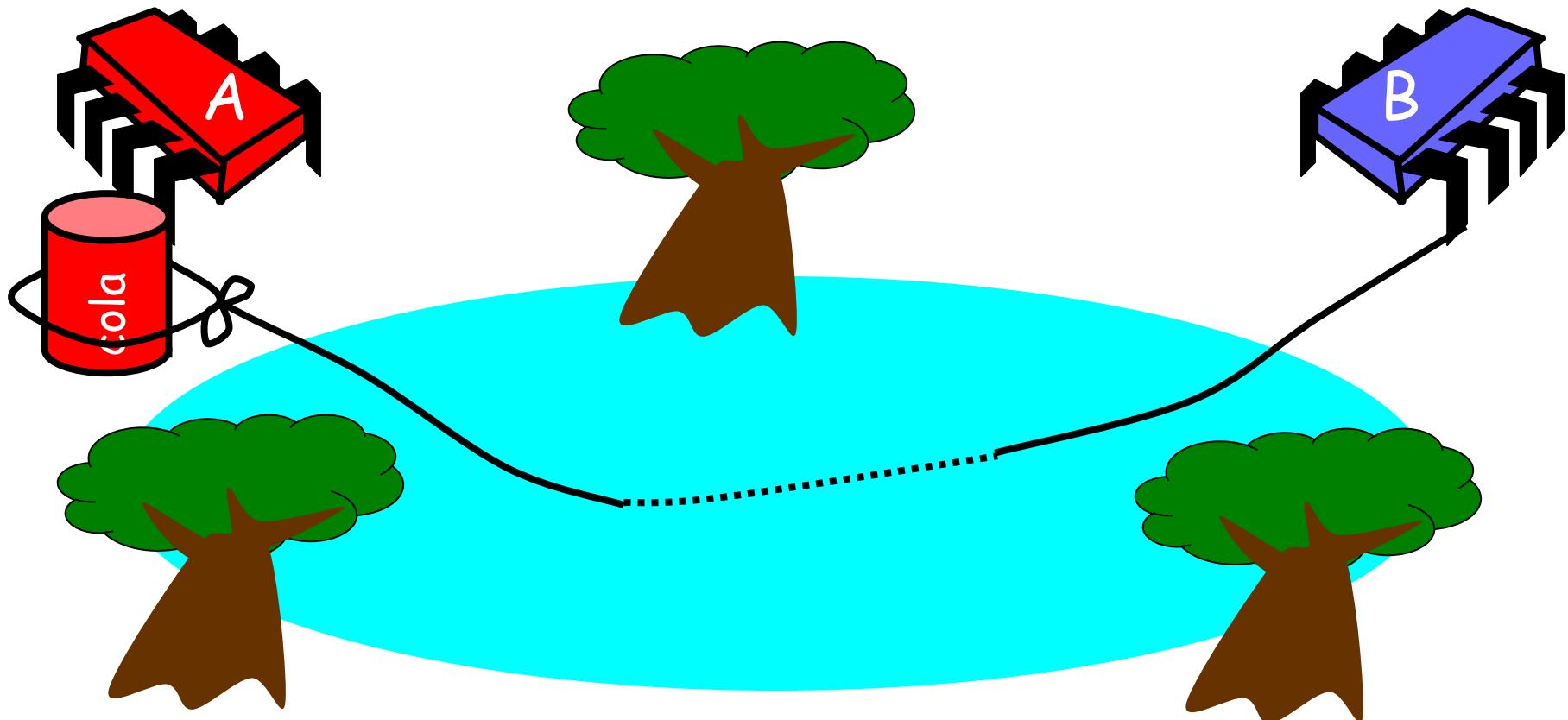
# Bob knocks over Can



# Alice Releases Pets



# Alice Resets Can when Pets are Fed

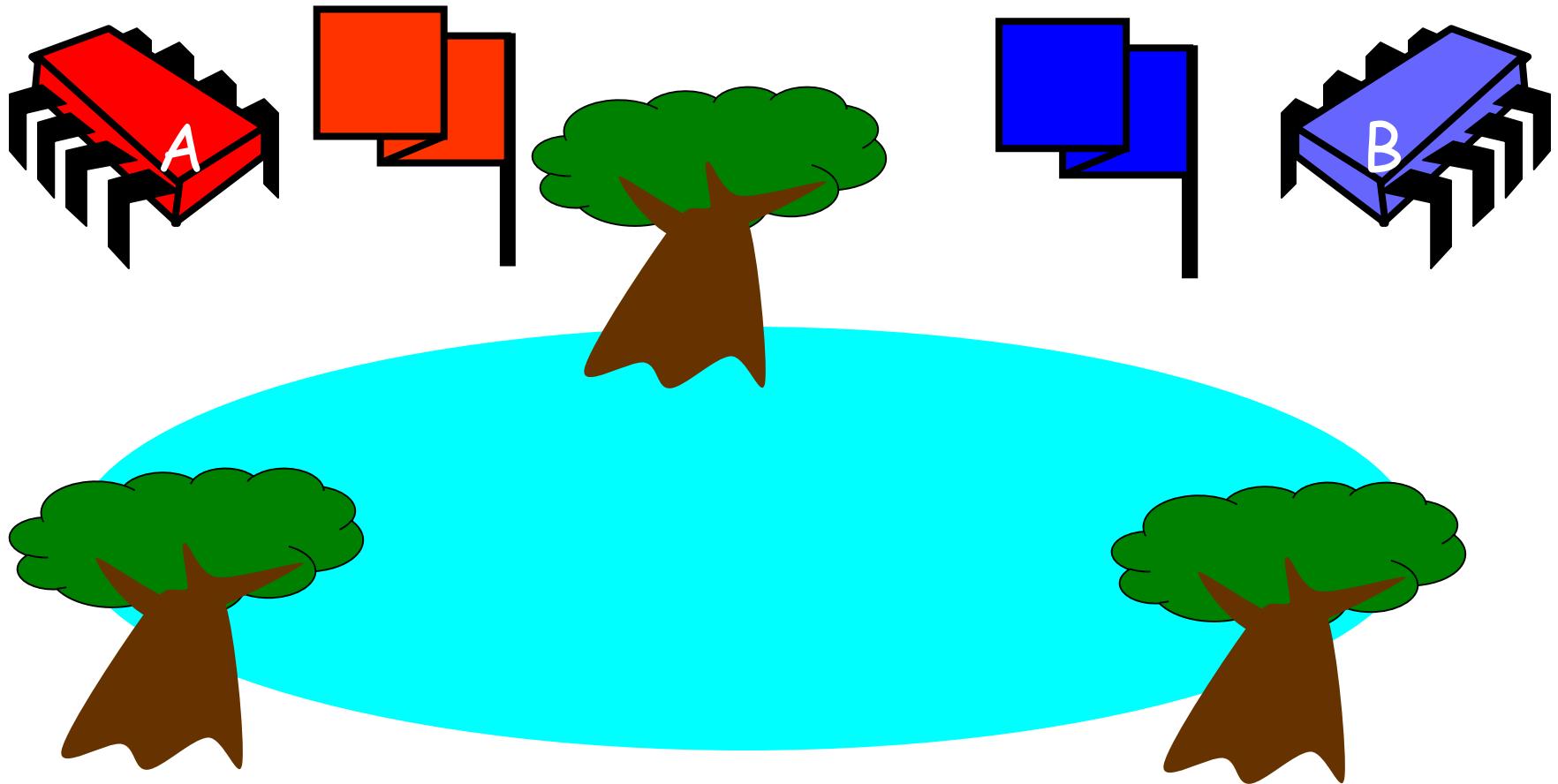




# Correctness

- Mutual Exclusion
  - Pets and Bob never together in pond
- No Starvation
  - if Bob always willing to feed, and pets always famished, then pets eat infinitely often.
- Producer/Consumer
  - The pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food.

# Could Also Solve Using Flags





# Waiting

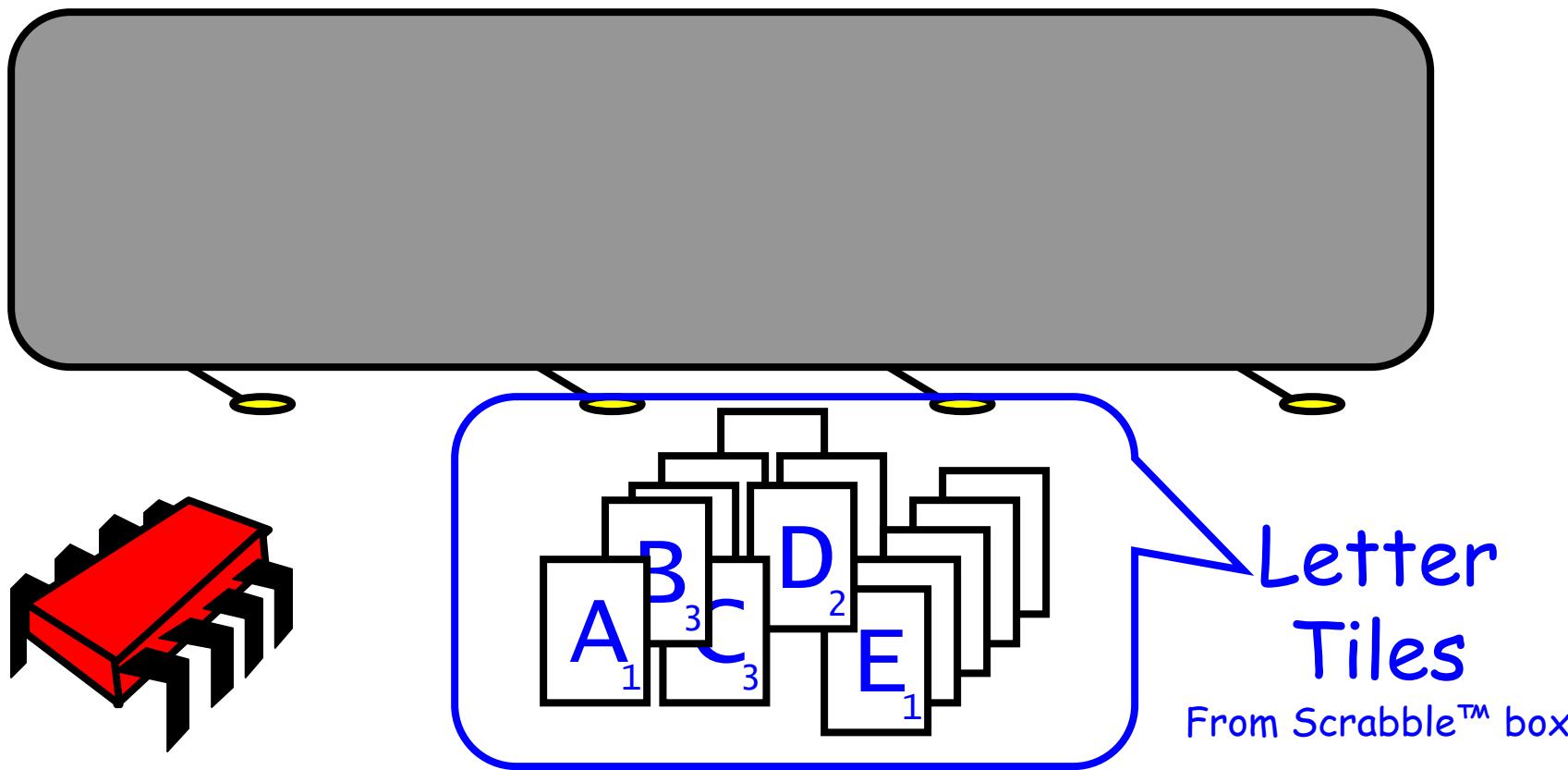
- Both solutions use waiting
- Waiting is ***problematic***
  - If one participant is delayed
  - So is everyone else
  - But delays are common & unpredictable



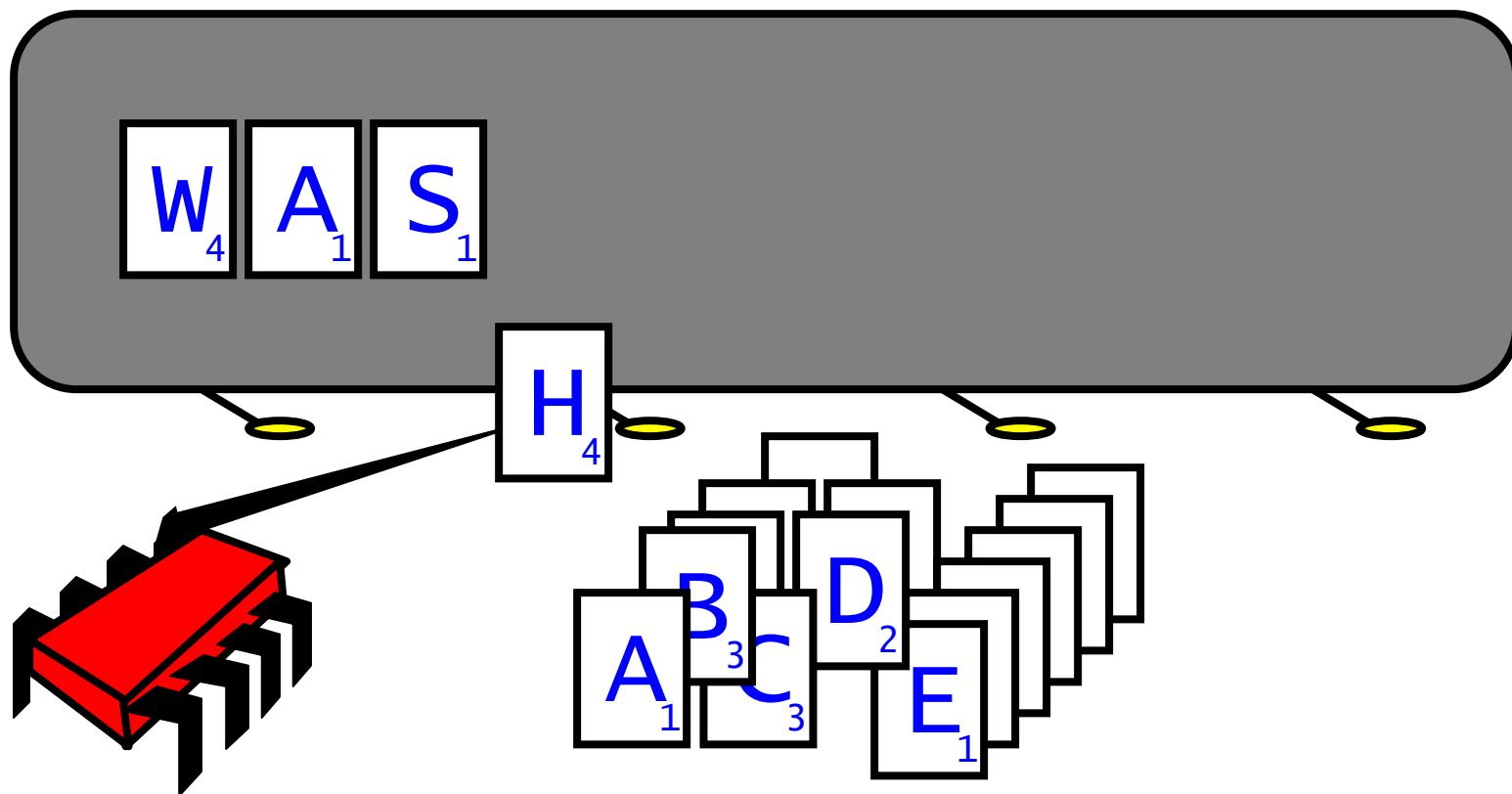
# The Fable drags on ...

- Bob and Alice still have issues
- So they need to communicate
- So they agree to use billboards ...

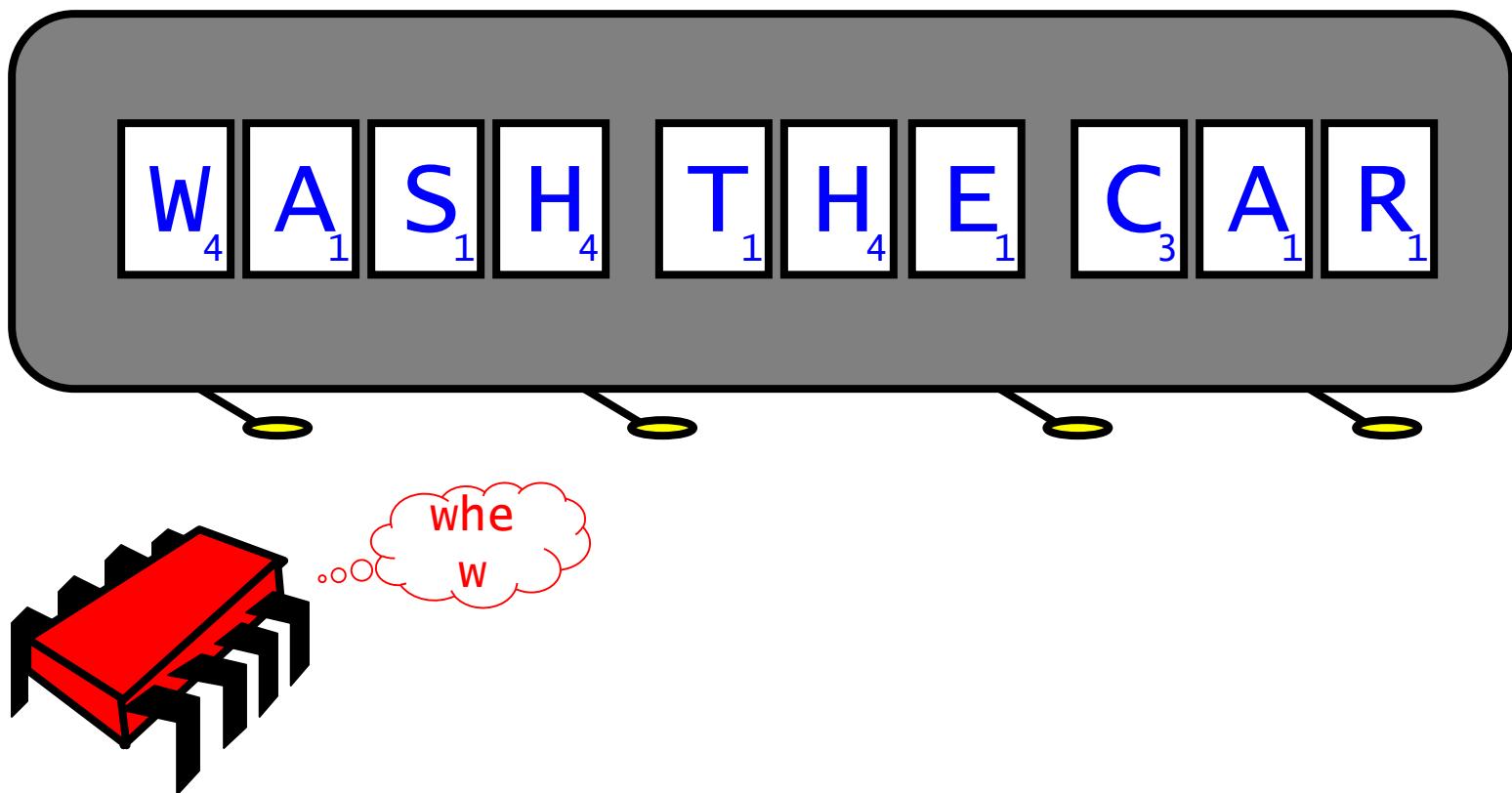
# Billboards are Large



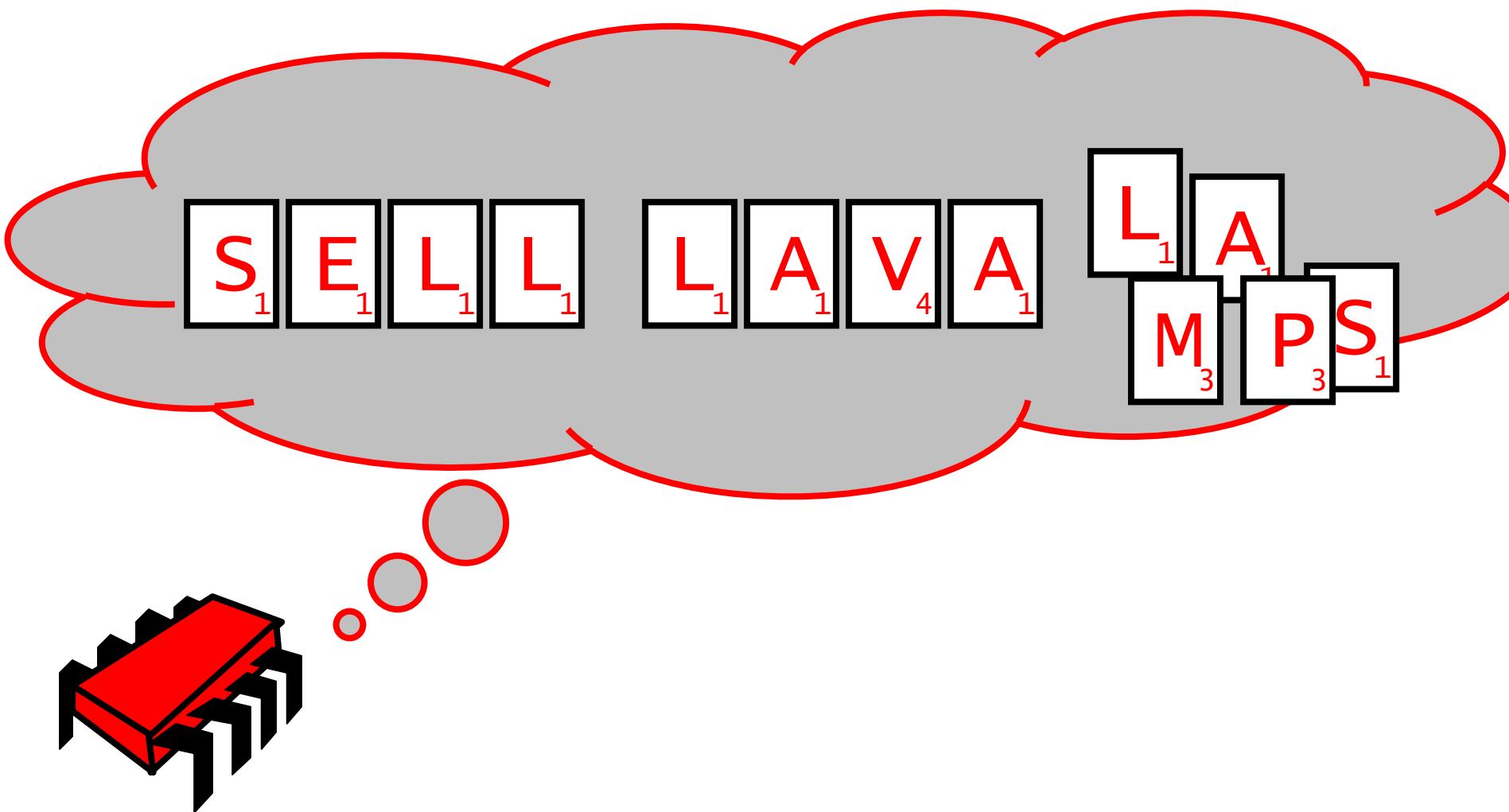
# Write One Letter at a Time ...



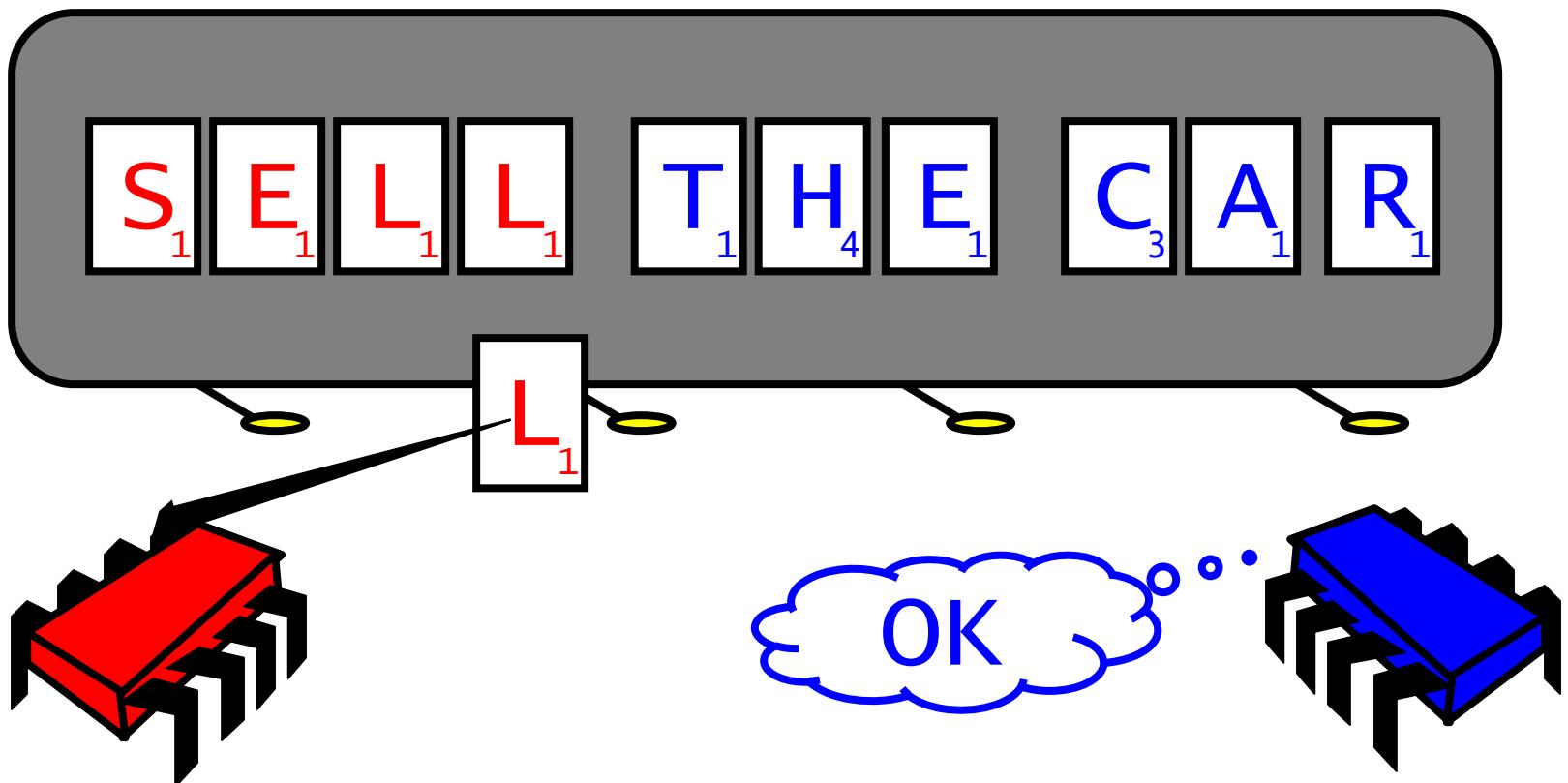
# To post a message



# Let's send another message



# Uh-Oh





# Readers/Writers

- Devise a protocol so that
  - Writer writes one letter at a time
  - Reader reads one letter at a time
  - Reader sees
    - Old message or new message
    - No mixed messages



# Why do we care?

- Upgrading from a uniprocessor to a n-way multiprocessor does not mean in n-fold increase in performance
- We want as much of the code as possible to execute concurrently (in parallel)
- A larger sequential part implies reduced performance



# Amdahl's law

- The extent to which we can speed up a complex job is limited by how much of the job must be executed sequentially.

# Amdahl's law

- Speedup = ratio between:
  - time it takes one processor to complete the task
    - Vs
  - time if takes  $n$  concurrent processors to complete the same task

# Amdahl's law

- $n$  – number of processors
- $p$  – fraction of task that can be executed in parallel
- Then:
  - The parallel part of the task will take  $p/n$  time
  - The sequential part of the task will take  $1 - p$  time
  - Parallelization is thus:  $1 - p + p/n$

# Amdahl's Law

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$

# Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 2.17 = \frac{1}{1 - 0.6 + \frac{0.6}{10}}$$

# Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}$$

# Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$



# The Moral

- Making good use of our multiple processors (cores) means
- Finding ways to effectively parallelize our code
  - Minimize sequential parts
  - Reduce idle time in which threads **wait** without



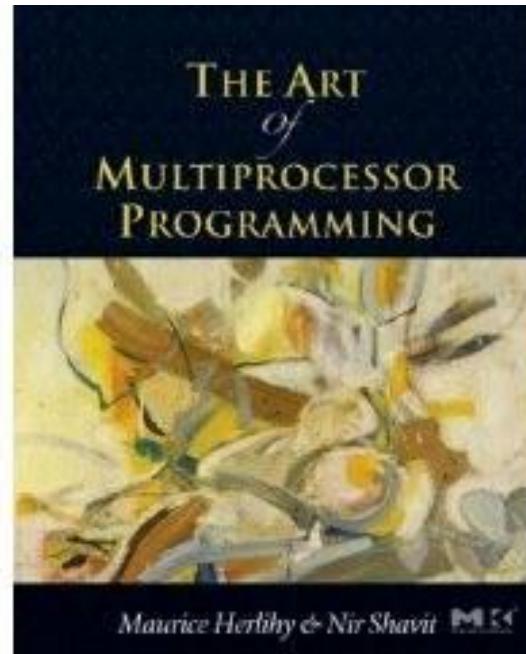
# Multicore Programming

- This is what this course is about...
  - The % that is not easy to make concurrent yet may have a large impact on overall speedup
- Next Week:
  - A more serious look at mutual exclusion

COS 226

Chapter 2  
Mutual Exclusion

# Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

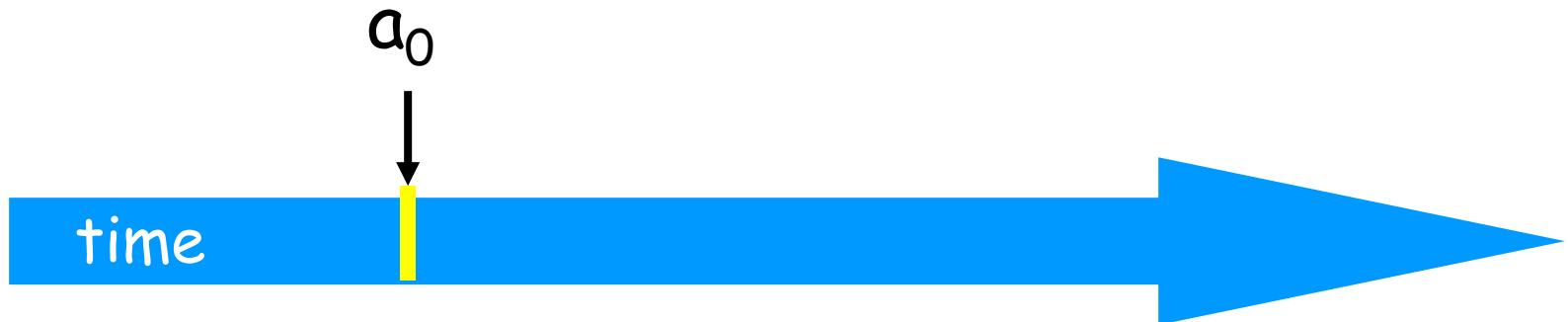


# Why is Concurrent Programming so Hard?

- 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

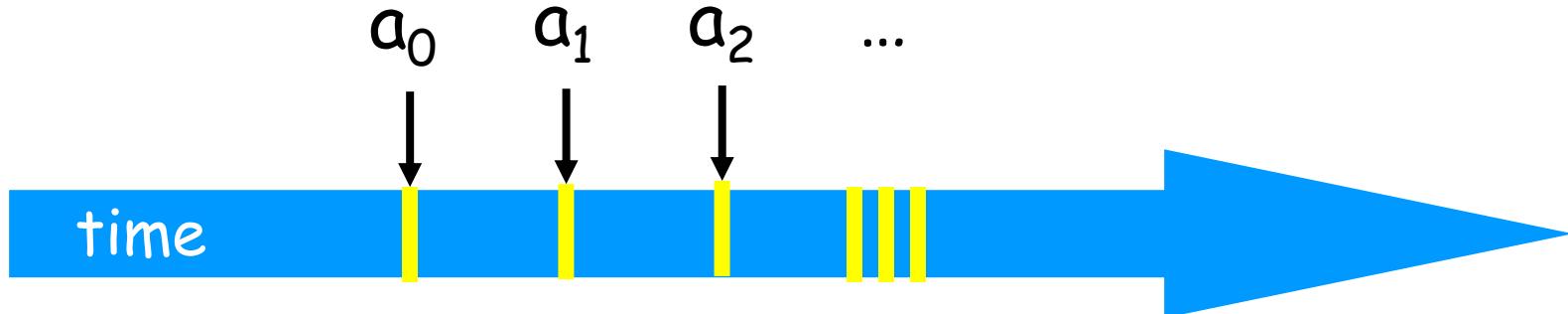
# Events

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



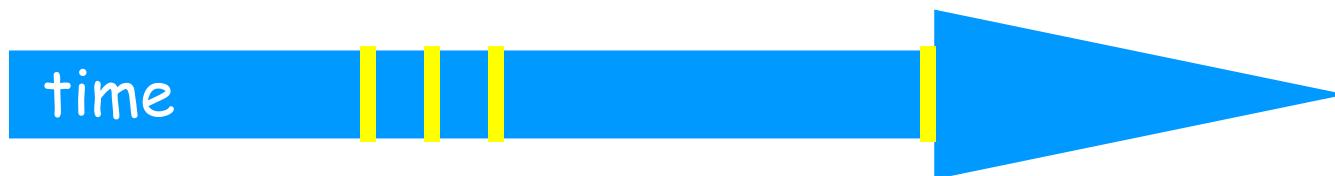
# Threads

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

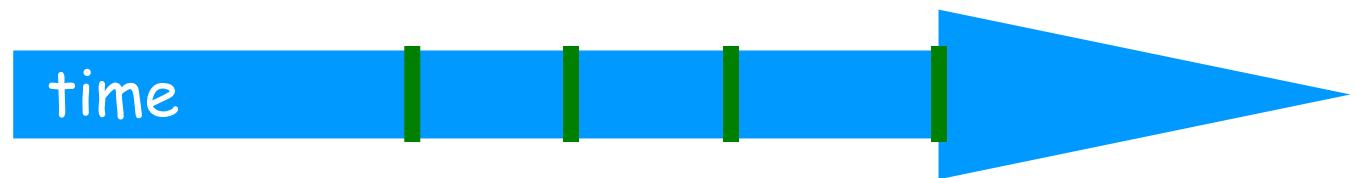


# Concurrency

## ■ Thread A

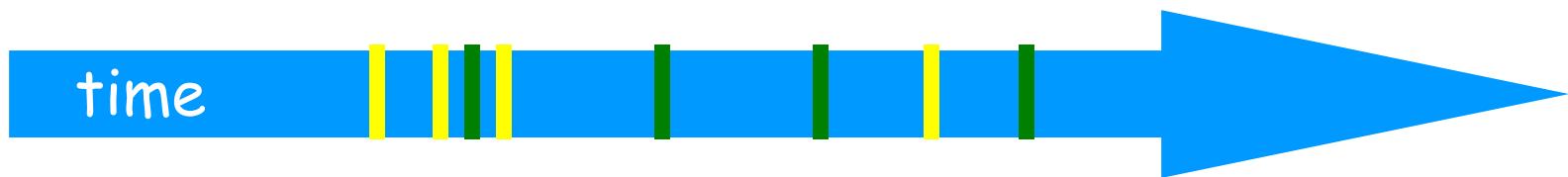


## ■ Thread B



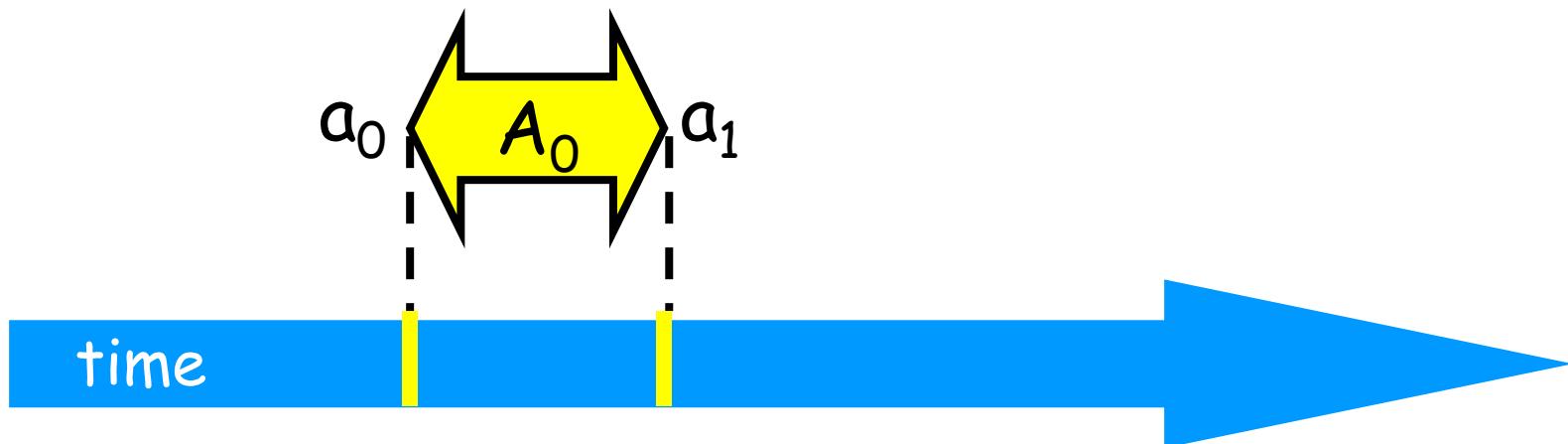
# Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent

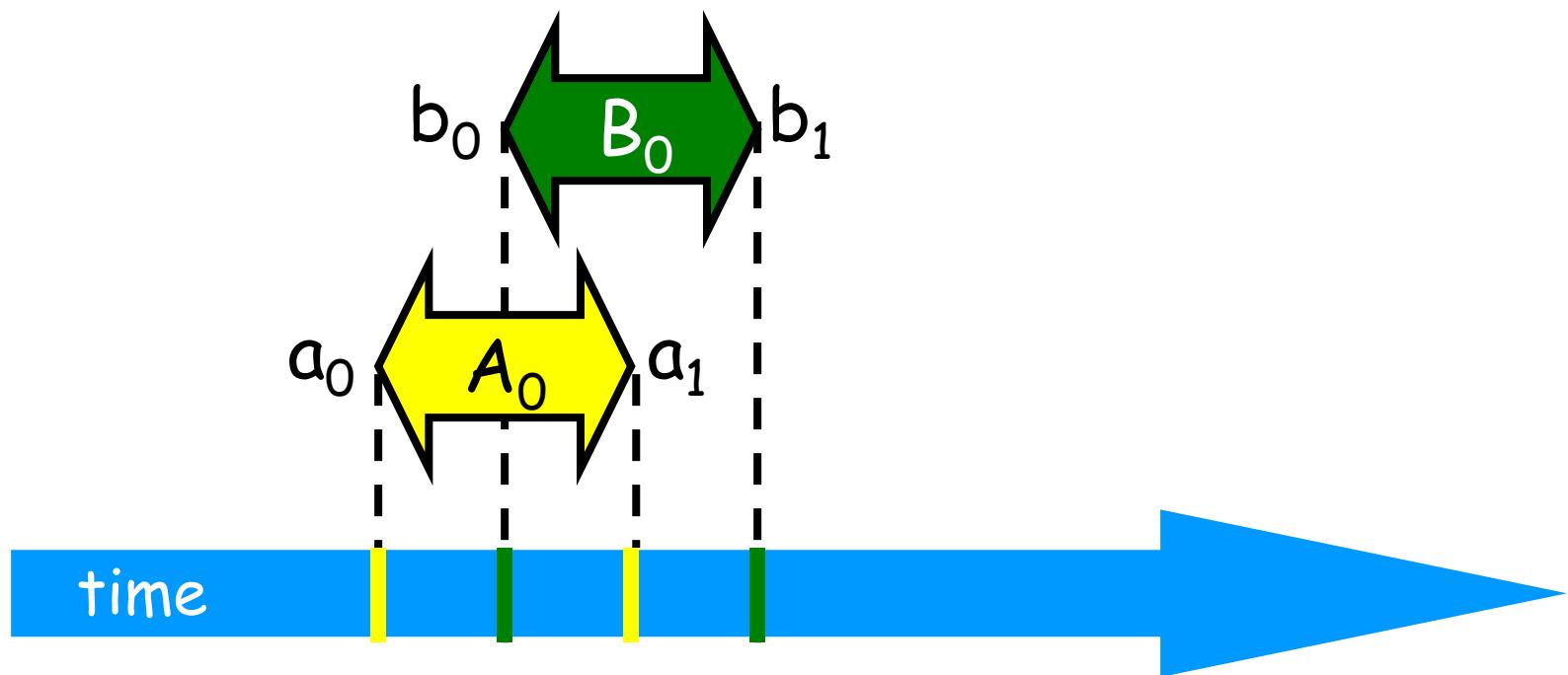


# Intervals

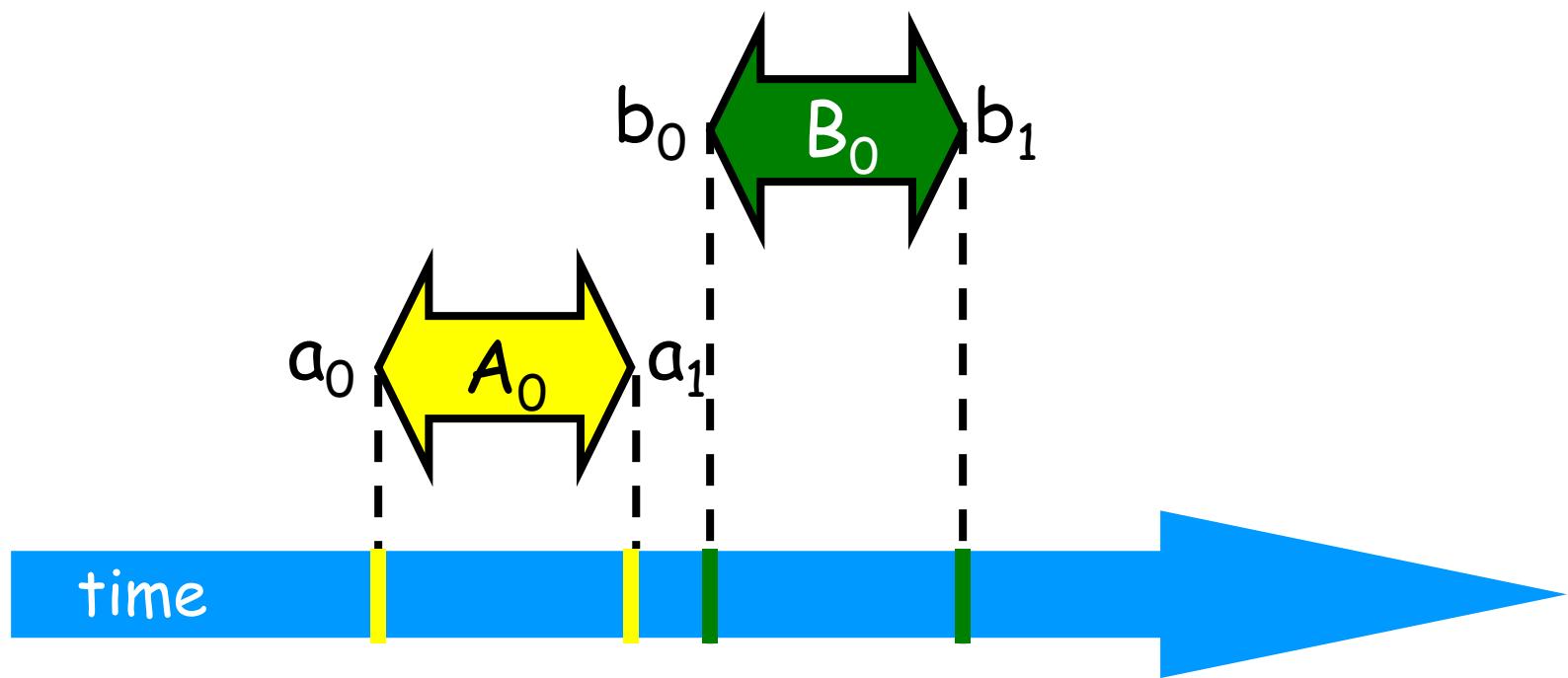
- An *interval*  $A_0 = (a_0, a_1)$  is
  - Time between events  $a_0$  and  $a_1$



# Intervals may Overlap

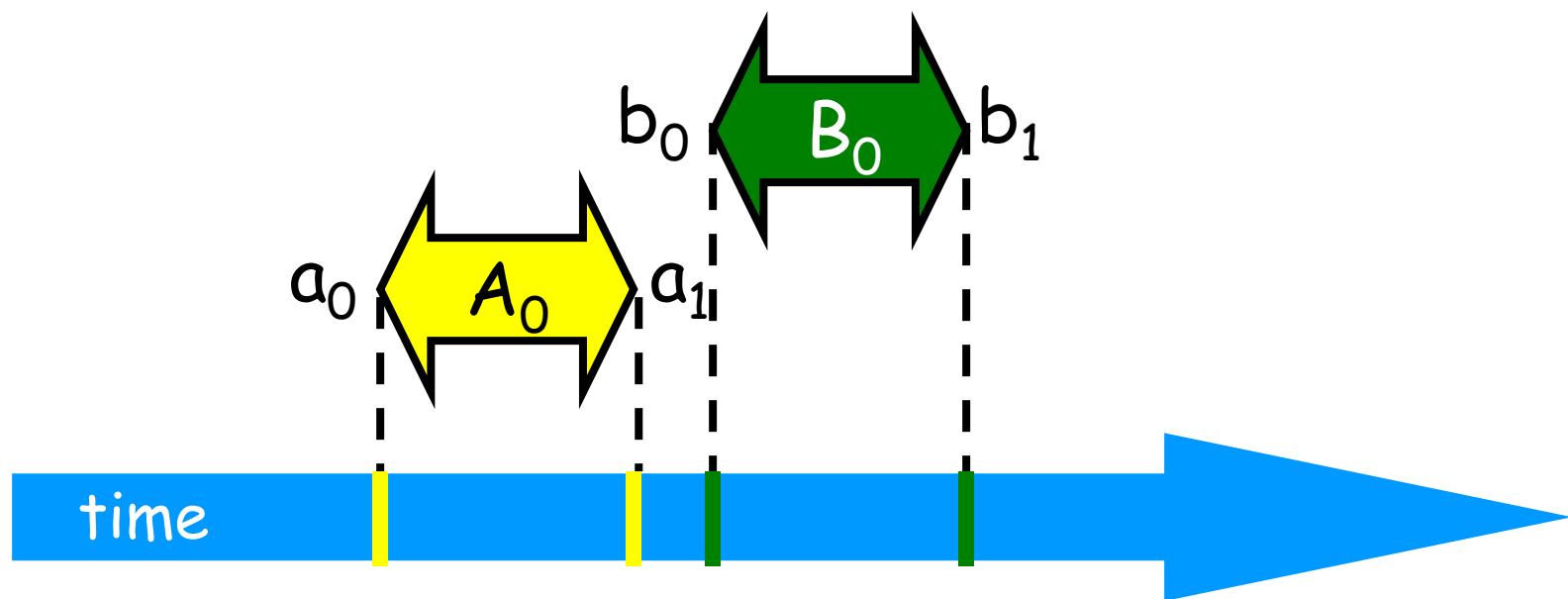


# Intervals may be Disjoint

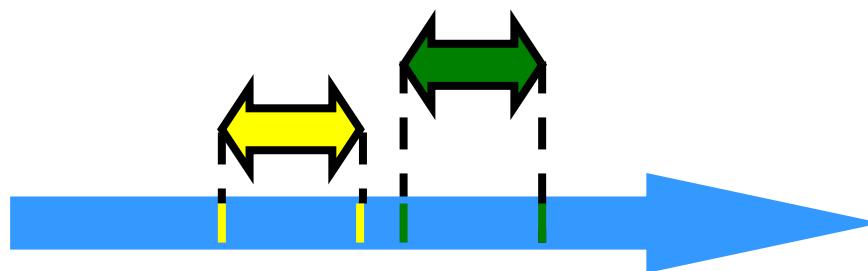


# Precedence

Interval  $A_0$  **precedes** interval  $B_0$

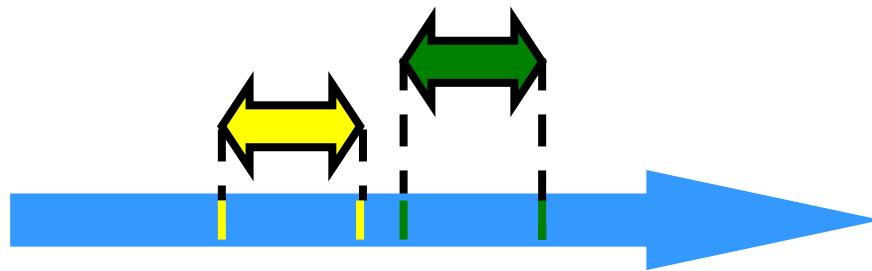


# Precedence



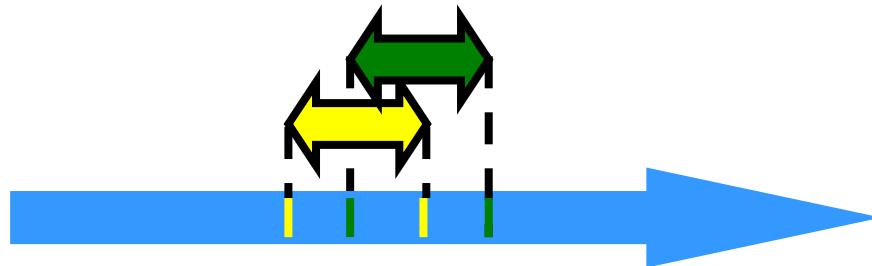
- Notation:  $A_0 \rightarrow B_0$
- Formally,
  - End event of  $A_0$  before start event of  $B_0$
  - Also called “happens before” or “precedes”

# Precedence Ordering



- Remark:  $A_0 \rightarrow B_0$  is just like saying
  - 1066 AD  $\rightarrow$  1492 AD,
  - Middle Ages  $\rightarrow$  Renaissance,

# 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!

# Repeated Events

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

*k-th occurrence  
of event a<sub>0</sub>*

a<sub>0</sub><sup>k</sup>

*k-th occurrence of  
interval A<sub>0</sub> = (a<sub>0</sub>, a<sub>1</sub>)*

A<sub>0</sub><sup>k</sup>

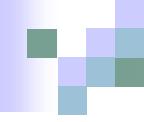


# Implementing a Counter

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        value++;  
    }  
}
```

# Implementing a Counter

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



# Critical Section

- Block of code that can be executed by only one thread at a time
- Needs Mutual Exclusion
- Standard way to approach mutual exclusion is through 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

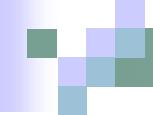
```
public class Counter {  
    private long value;  
    private Lock lock;  
    public long getAndIncrement() {  
        lock.lock();  
        try {  
            int temp = value;  
            value = value + 1;  
        } finally {  
            lock.unlock();  
        }  
        return temp;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

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



# Properties of a good Lock algorithm

- Mutual Exclusion
- Deadlock-free
- Starvation-free



# Mutual Exclusion

- Threads do not access critical section at same time



# Deadlock-free

- If some thread attempts to acquire the lock, some thread will succeed in acquiring the lock



# Deadlock-free

- If some thread attempts to acquire the lock, some thread will succeed in acquiring the lock
  - System as a whole makes progress
  - Even if individuals starve
  - At least one thread is completing



# Starvation-free

- Every thread that attempts to acquire the lock will eventually succeed



# Starvation-free

- **Every** thread that attempts to acquire the lock will eventually succeed
  - If a thread calls lock() it will eventually acquire the lock
  - Individual threads make progress



# Locks

- Let's start with lock solutions for 2 concurrent threads...

# 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

- Basic idea:
  - Thread indicates interest in acquiring lock
  - Checks to see if other thread is currently in critical section
    - If true, waits until other thread finishes
    - If not, enters critical section

# LockOne

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

# LockOne

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



**Set my flag**

# LockOne

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

**Set my flag**

**Wait for other  
flag to go false**

# LockOne

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

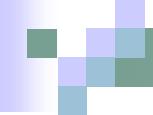
**When release lock  
set my flag again**

# Deadlock Freedom?

- Concurrent execution:

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

- If each thread sets its flag to true and waits for the other, they will wait forever
- No deadlock freedom



# LockOne Summary

- LockOne offers mutual exclusion
- When accessed sequentially, LockOne works fine
- However with concurrent threads, LockOne is not Deadlock-free



# LockTwo

- Basic idea:
  - When attempting to acquire lock, offer to be the victim that has to defer to other thread
  - While current thread is the victim, wait until other thread becomes the victim
  - When current thread no longer the victim, enter the critical section

# LockTwo

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

# LockTwo

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

Let other go first

# LockTwo

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

**Wait for permission**

# LockTwo

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

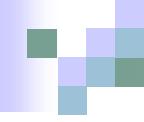
Nothing to do

# LockTwo Claims

- Satisfies mutual exclusion

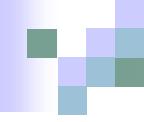
- If thread **i** in CS
  - Then **victim == j**
  - Cannot be both 0 and 1

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



# LockTwo Summary

- LockTwo offers Mutual Exclusion
  - Works fine with concurrent threads
  - However results in Deadlock with sequential threads
- 
- LockOne and LockTwo thus complement each other



# Peterson Lock

- Combine LockOne and LockTwo
  - Enable successful sequential access provided by LockOne
  - Enables successful concurrent access provided by LockTwo

# Peterson Lock

- Basic idea:
  - Current thread indicates interest in acquiring the lock
  - Current thread offers to be the victim
  - If no interest from other thread and no longer the victim, then continue to critical section

# Peterson's Algorithm

```
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

# 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 other's flag is true
  - only if it is the **victim**
- Solo: other's flag is false
- Both: one or the other must not be the victim

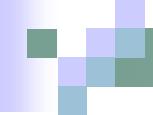
# Starvation Free

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

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

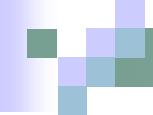
- When **j** re-enters
  - it sets **victim** to **j**.
  - So **i** gets in
- Thus: Starvation free

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



# Locks

- Moving on to solutions for  $n$  concurrent threads



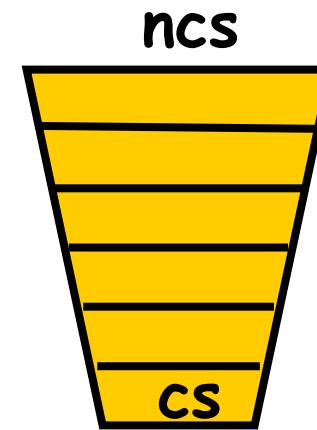
# Filter Lock

- Peterson lock adapted to work with  $n$  threads instead of just 2
- Thread has to traverse  $n-1$  waiting rooms in order to acquire the lock

# 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 {  
    int[] level; // level[i] for thread i  
    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;  
        }  
        ...  
    }  
}
```

# 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; // busy wait  
        }  
        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

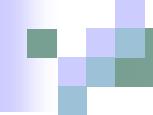
Wait as long as someone else is at  
same or higher level, and I'm  
designated victim

```
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;  
        }  
    }
```

# 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;  
        }  
    }
```

Thread enters level L when it completes  
the loop



# No Starvation

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



# Waiting

- Starvation freedom guarantees that every thread that calls lock() eventually enters the critical section
- It however makes no guarantee about how long that can take



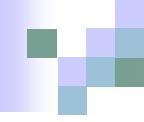
# Waiting

- Ideally if A calls lock() before B, then A should enter critical section before B
- However this does not currently work since we cannot determine which thread called lock() first
- Locks should thus be further defined



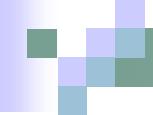
# Fairness

- Locks should be first-some-first served



# Filter Lock again

- Filter Lock satisfies properties:
  - No one starves
  - But very weak fairness
    - Can be overtaken **arbitrary** # of times
  - That's pretty lame...



# Bakery Algorithm

- Provides First-Come-First-Served
- How?
  - Take a “number”
  - Wait until lower numbers have been served
- Each thread takes a number when attempting to acquire the lock and waits until no thread with an earlier number is trying to acquire it

# Bakery Algorithm

```
class Bakery implements Lock {  
    boolean[] flag;  
    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

- $\text{flag}[A]$  is a boolean flag indicating whether A wants to enter the critical section
- $\text{label}[A]$  is an integer that contains thread A's "number" when entering the bakery

# Bakery Algorithm

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

# Bakery Algorithm

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

I'm interested

# Bakery Algorithm

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

Take increasing  
label (read labels  
in some arbitrary  
order)

Label is created as one greater than the maximum of the other thread's labels

# 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[k] < label[i]));  
    }  
}
```

**Someone is interested**

# 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[k] < label[i]));  
    }  
}
```

**Someone is interested**

**And someone has  
a smaller number  
than me**

**THEORETICALLY**



# Bakery Algorithm

- If two threads try to acquire the lock concurrently, they may read the same maximum number
- Threads thus have unique pairs consisting of number as well as thread ID

# Bakery Algorithm

$(\text{label}[i], i) \ll (\text{label}[j], j)$

If and only if

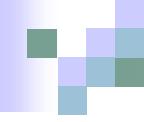
$\text{label}[i] < \text{label}[j]$  OR  $\text{label}[i] = \text{label}[j]$  and  $i < j$

# 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[k],k) << label[i],i));  
    }  
}
```

Someone is interested

With lower (label,i)  
in lexicographic order



# Bakery Algorithm

- In other words:
- Thread A must wait if:
  - Another thread is interested AND the other thread's number is lower than thread A
    - OR
  - Another thread is interested AND the two threads have the same number but the other's threads ID is smaller than A

# 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



# To analyse:

- Does the lock provide:
  - Mutual exclusion?
    - YES – two threads cannot be in the critical section at the same time since one of them will have an earlier label pair



# To analyse:

- Starvation freedom?
  - YES – if a thread exists the critical section and immediately wants to reacquire the lock, he will first have to take a new, later number allowing the other waiting threads to gain access first



# To analyse:

- Deadlock freedom?
  - YES – there is always one thread with the earliest label, ties are not possible because of labels consist of number and order in array

# To analyse:

- The Bakery algorithm also provides First-come-first-served
  - If A calls lock() before B, then A's number is smaller than B's number
  - So B is locked out while  $\text{flag}[A]$  is true



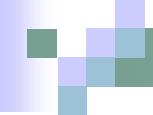
# Potential issue:

- With the current Bakery algorithm we are assuming that we have an infinite amount of numbers to use
- In practice this is not the case



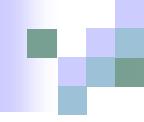
# Bounded timestamps

- Labels in the Bakery lock grow without bounds
- In a long-lived system we may have to worry about overflow
- If a thread's label silently rolled over from a large number to zero, the first-come-first-served property no longer holds



# Bounded timestamps

- In the Bakery algorithm, the idea of labels can be replaced by timestamps
- Timestamps can ensure order among the contending threads
- We will thus need to ensure that if one thread takes a label after another, then the latter has the higher timestamp



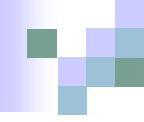
# Bounded timestamps

- Timestamps need the ability to:
  - Scan – read the other thread's timestamps
  - Label – assign itself a larger timestamp



# Possible solution

- To construct a **Sequential** timestamping system
- Each thread perform scan-and-label completely one after the other
- Uses mutual exclusion



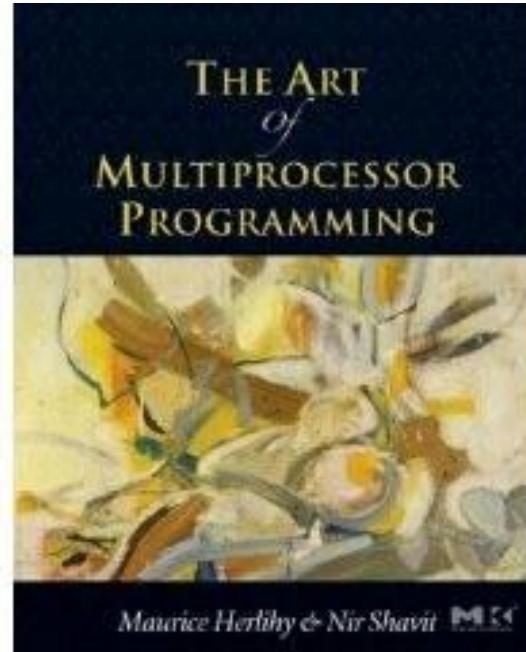
# Bakery algorithm

- The Bakery algorithm is elegant and fair
- However it is not considered practical
  - Why?
  - Principal drawback is the need to read  $n$  distinct location where  $n$  can be very large

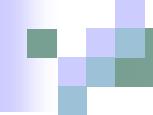
COS 226

Chapter 3  
Concurrent objects

# Acknowledgement

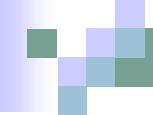


- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



# Sequential Objects

- In OO programming, an object is a container for data
- Each object has a ***state***
  - Usually given by a set of ***fields***
  - Queue example: sequence of items
- Each object has a set of ***methods***
  - Only way to manipulate state
  - Queue example: **enq** and **deq** methods



# Objectivism

- What is a concurrent object?
  - How do we **describe** one?
  - How do we **implement** one?
  - How do we **tell if we're right**?



# Correctness and Progress

- In a concurrent setting, we need to specify both the safety and the liveness properties of an object
  - Safety – nothing bad happens (also known as correctness)
  - Liveness – something good eventually happens (also known as progress)



# Sequential objects

- With sequential objects, one way to determine if an object's methods are behaving correctly is through pre and postconditions.



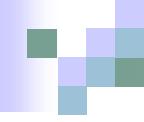
# Sequential Specifications

- If (precondition)
  - the object is in such-and-such a state
  - before you call the method,
- Then (postcondition)
  - the method will return a particular value
  - or throw a particular exception.
- and (postcondition, con't)
  - the object will be in some other state
  - when the method returns,



# Pre and PostConditions for Dequeue

- Precondition:
  - Queue is non-empty
- Postcondition:
  - Returns first item in queue
- Postcondition:
  - Removes first item in queue



# Pre and PostConditions for Dequeue

- Precondition:
  - Queue is empty
- Postcondition:
  - Throws Empty exception
- Postcondition:
  - Queue state unchanged



# Sequential Specifications

- Interactions among methods captured by resulting object state
  - State meaningful between method calls
- Documentation size linear in number of methods
  - Each method described in isolation
- Can add new methods
  - Without changing descriptions of old methods



# What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

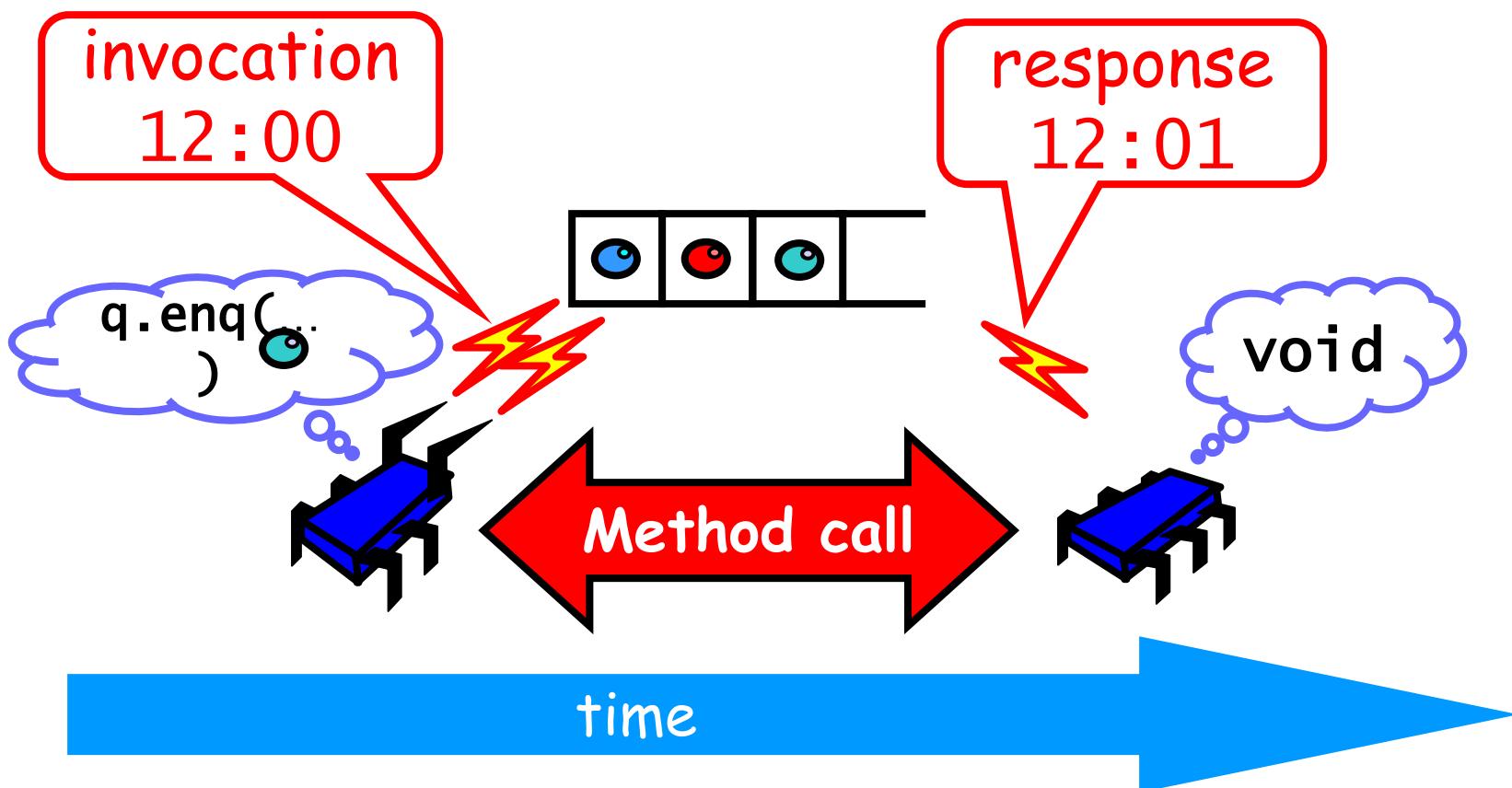


# Correctness and Progress

- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Lets begin with correctness

# Methods Take Time

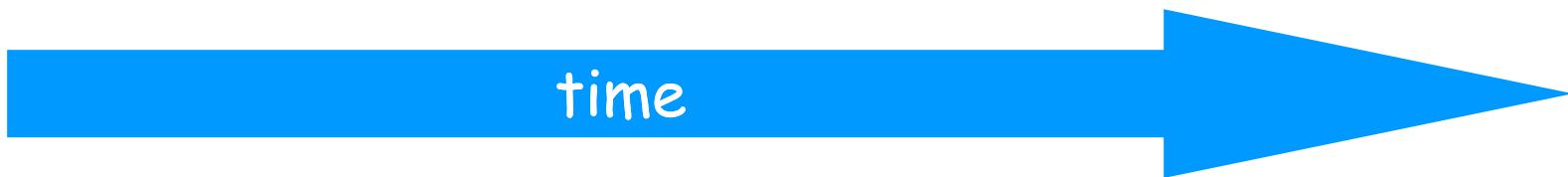
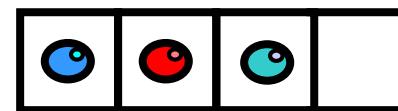




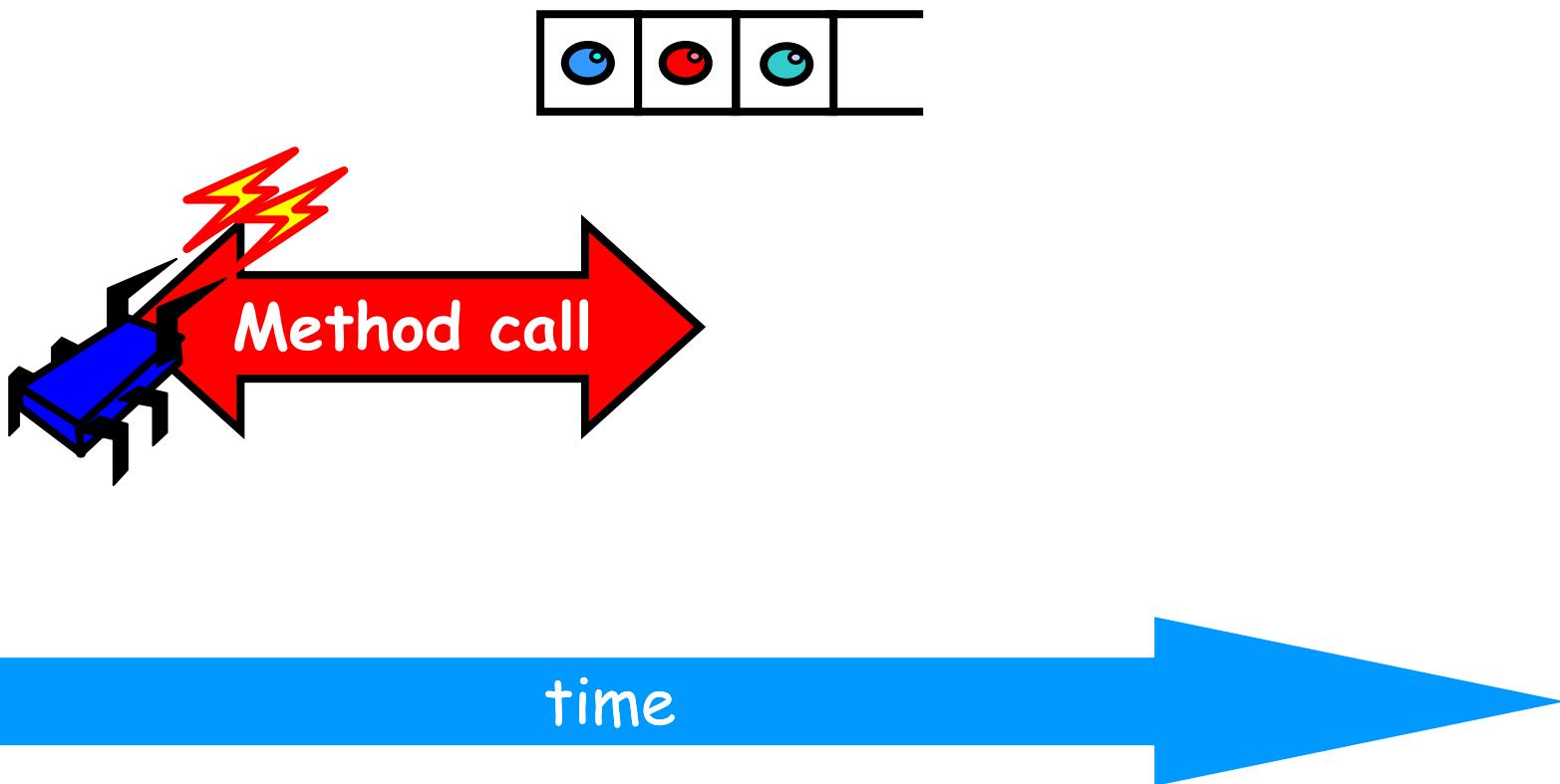
# Sequential vs Concurrent

- Sequential
  - Method calls take time? Who knew?
- Concurrent
  - Method call is not an event
  - Method call is an interval.
    - Starts with invocation event
    - Ends with response event
    - Method is pending if invocation has occurred but not yet response

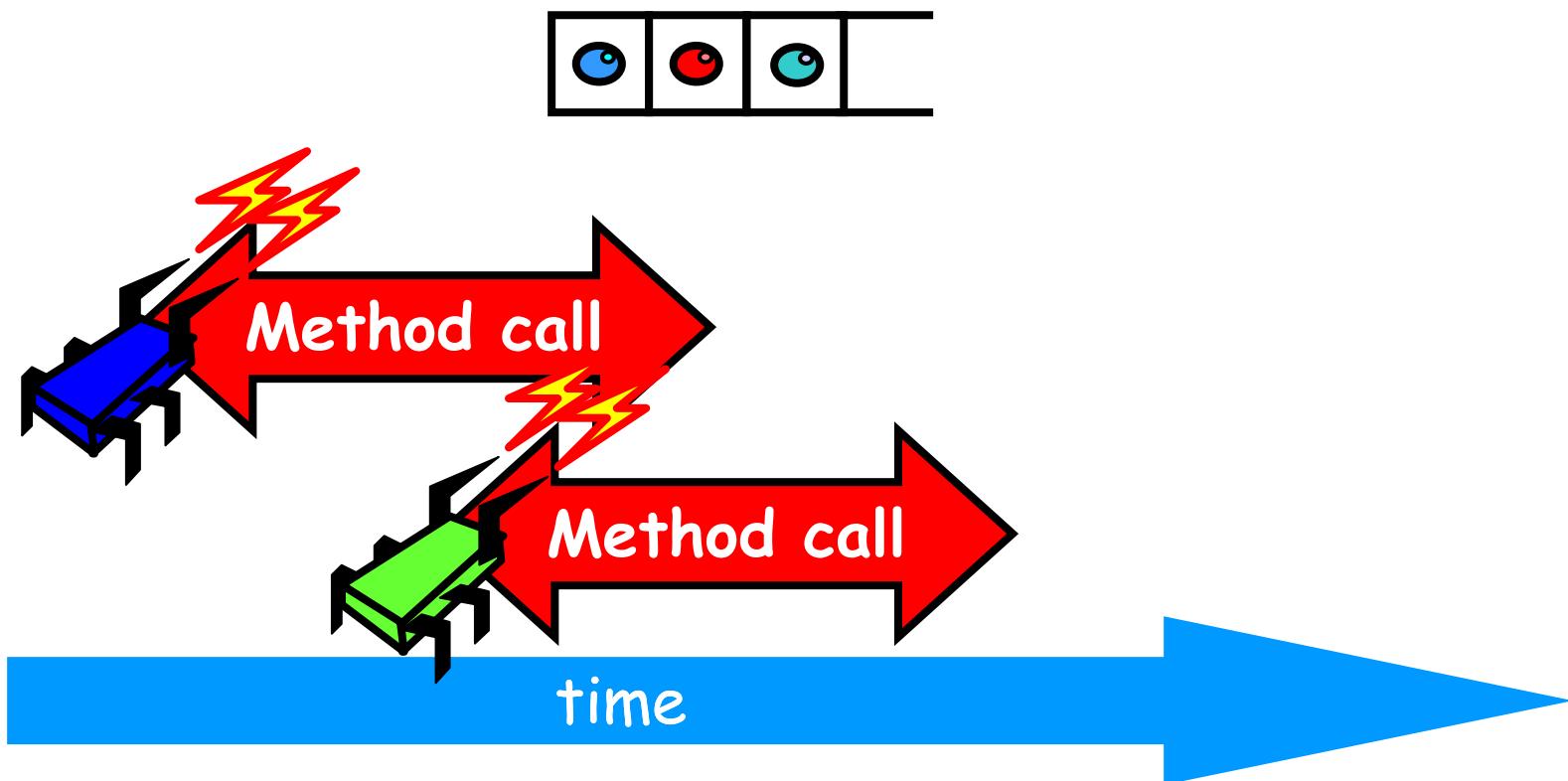
# Concurrent Methods Take Overlapping Time



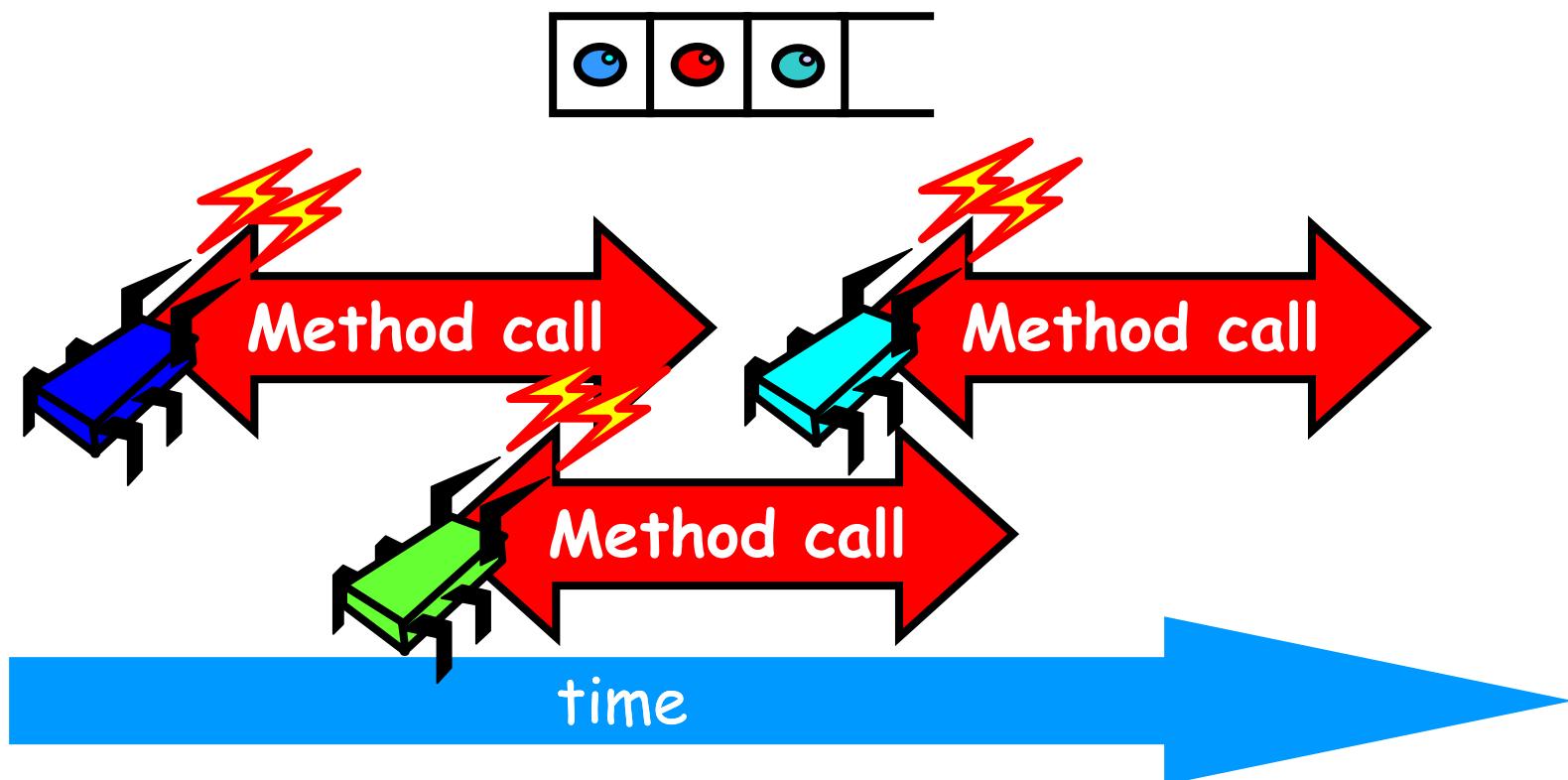
# Concurrent Methods Take Overlapping Time



# Concurrent Methods Take Overlapping Time



# Concurrent Methods Take Overlapping Time





# Sequential vs Concurrent

- Sequential:
  - Object needs meaningful state only ***between*** method calls
- Concurrent
  - Because method calls overlap, object might ***never*** be between method calls



# Sequential vs Concurrent

- Sequential:
  - Each method described in isolation
- Concurrent
  - Must characterize *all* possible interactions with concurrent calls
    - What if two enqs overlap?
    - Two deqs? enq and deq? ...

# Sequential vs Concurrent

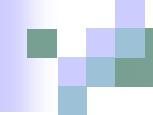
- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

Panic!



# The Big Question

- What does it **mean** for a *concurrent* object to be correct?



# A Lock-Based Queue

```
class LockBasedQueue<T> {
    int head, tail;
    T[] items;
    Lock lock;
    public LockBasedQueue(int capacity) {
        head = 0; tail = 0;
        lock = new ReentrantLock();
        items = (T[]) new Object[capacity];
    }
}
```

# A Lock-Based Queue

```
class LockBasedQueue<T> {  
    int head, tail;  
    T[] items;  
    Lock lock;  
    public LockBasedQueue(int capacity) {  
        head = 0; tail = 0;  
        lock = new ReentrantLock();  
        items = (T[]) new Object[capacity];  
    }  
}
```

Queue fields  
protected by  
single shared lock

# A Lock-Based Queue

```
class LockBasedQueue<T> {  
    int head, tail;  
    T[] items;  
    Lock lock;  
    public LockBasedQueue(int capacity) {  
        head = 0; tail = 0;  
        lock = new ReentrantLock();  
        items = (T[]) new Object[capacity];  
    }  
}
```

Initially  $\text{head} = \text{tail}$

# A Lock-Based Queue

```
public void enq(T x) throws  
    FullException {  
    if (tail - head == items.length)  
        throw new FullException();  
    items[tail] = x;  
    tail++;  
}
```

Mutual  
Exclusion?

# A Lock-Based Queue

```
public void enq(T x) throws FullException {  
    lock.lock();  
    try {  
        if (tail - head == items.length)  
            throw new FullException();  
        items[tail] = x;  
        tail++;  
    } finally {  
        lock.unlock();  
    }  
}
```

Method calls  
mutually exclusive

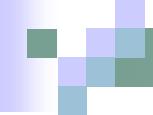
# Implementation: Deq

```
public T deq() throws EmptyException {  
    if (tail == head)  
        throw new EmptyException();  
    T x = items[head];  
    head++;  
    return x;  
}
```

# Implementation: Deq

```
public T deq() throws EmptyException {  
    lock.lock();  
    try {  
        if (tail == head)  
            throw new EmptyException();  
        T x = items[head];  
        head++;  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```

Method calls  
mutually exclusive



# Now consider the following implementation

- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread **enq only**
  - The other **deq only**

# Wait-free 2-Thread Queue

```
public class WaitFreeQueue {  
  
    int head = 0, tail = 0;  
    items = (T[]) new Object[capacity];  
  
    public void enq(Item x) {  
        while (tail-head == capacity); // busy-wait  
        items[tail % capacity] = x; tail++;  
    }  
    public Item deq() {  
        while (tail == head);  
        Item item = items[head];  
        return item;  
    }  
}
```

How do we define "correct"  
when modifications are not  
mutually exclusive?

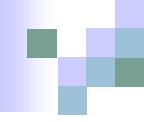


# Read-write example

- Two threads concurrently write -3 and 7 to a register
  - Register – object version of memory location
- Later when another thread accesses the register it returns -7
- Clearly this is wrong – we expect either -3 or 7, but not a mixture

# Principle 3.3.1

- Method calls should appear to happen in a one-at-a-time sequential order
  - By itself this principle is too weak to be useful
  - Has to combine it with a stronger condition...



# Quiescence

- A object is **quiescent** if it has no pending method calls
  - Can think of it as object is *inactive*

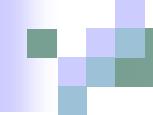
# Principle 3.3.2

- **Method calls separated by a period of quiescence should appear to take effect in real-time order**
  - In other words, method calls who are separated by a period of inactivity should appear in the order of their execution
  - Suppose A and B concurrently enqueue x and y, C then enqueues z. We may not be able to predict the order of x and y, but we know they are ahead of z



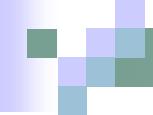
# Quiescent consistency

- Together principle 3.3.1 and 3.3.2 form a correctness property:
  - Quiescent consistency



# Quiescent consistency

- An object is quiescent consistent if:
  - Its method calls appear to be in a sequential order
  - Its method calls take place in a real-time order if separated by a period of inactivity



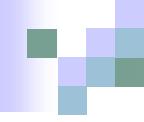
# Quiescent consistency

- A shared counter is thus quiescently consistent if:
  - When two concurrent threads write -3 and 7 to a register a later thread will read either -3 or 7 but not a mixture of the two



# Quiescent consistency

- Quiescent consistency is **compositional**
  - If each object in the system is quiescent consistent, the whole system will be quiescent consistent.

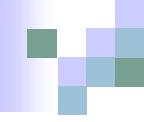


# Another read-write example

- A *single* thread writes 7 and then -3 to a shared register
- Later it reads the register and returns 7
- This is also not acceptable since the value it read is not the last value it wrote

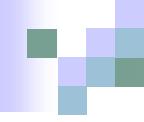
# Principle 3.4.1

- Method calls should take effect in program order
  - Program order – The order in which a single thread issues method calls
  - Method calls by different threads are unrelated by program order



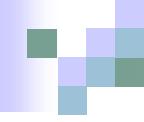
# Sequential consistency

- Together principles 3.3.1 and 3.4.1 form a second correctness property:
  - Sequential consistency



# Sequential consistency

- An object is sequential consistent if:
  - Its method calls are in a sequential order
  - Its method calls are in program order



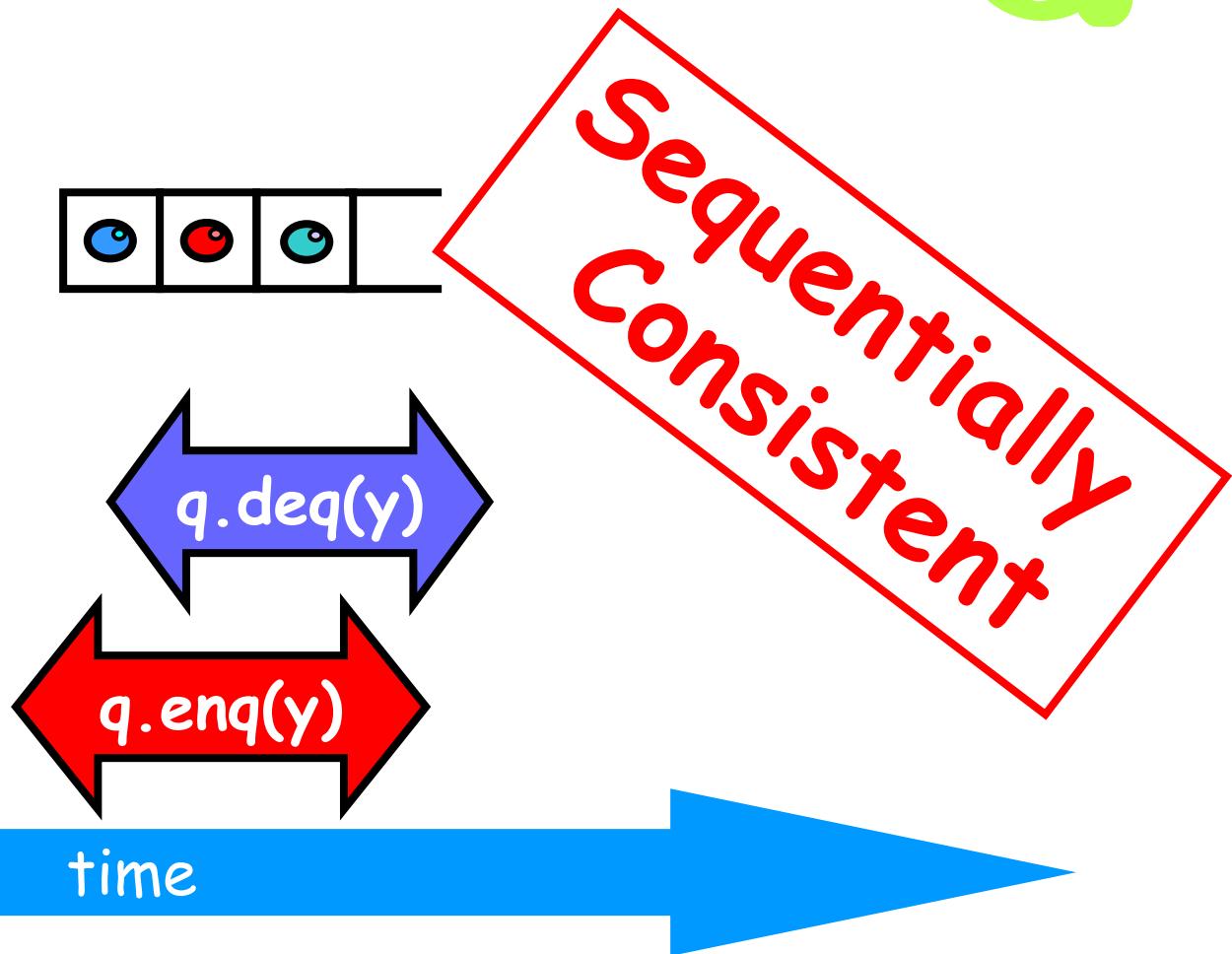
# Sequential consistency

- In any concurrent execution, there is a way to order the method calls sequentially so that
  - They are consistent with program order
  - They meet the object's sequential specifications
- There may be more than one order that satisfies these conditions

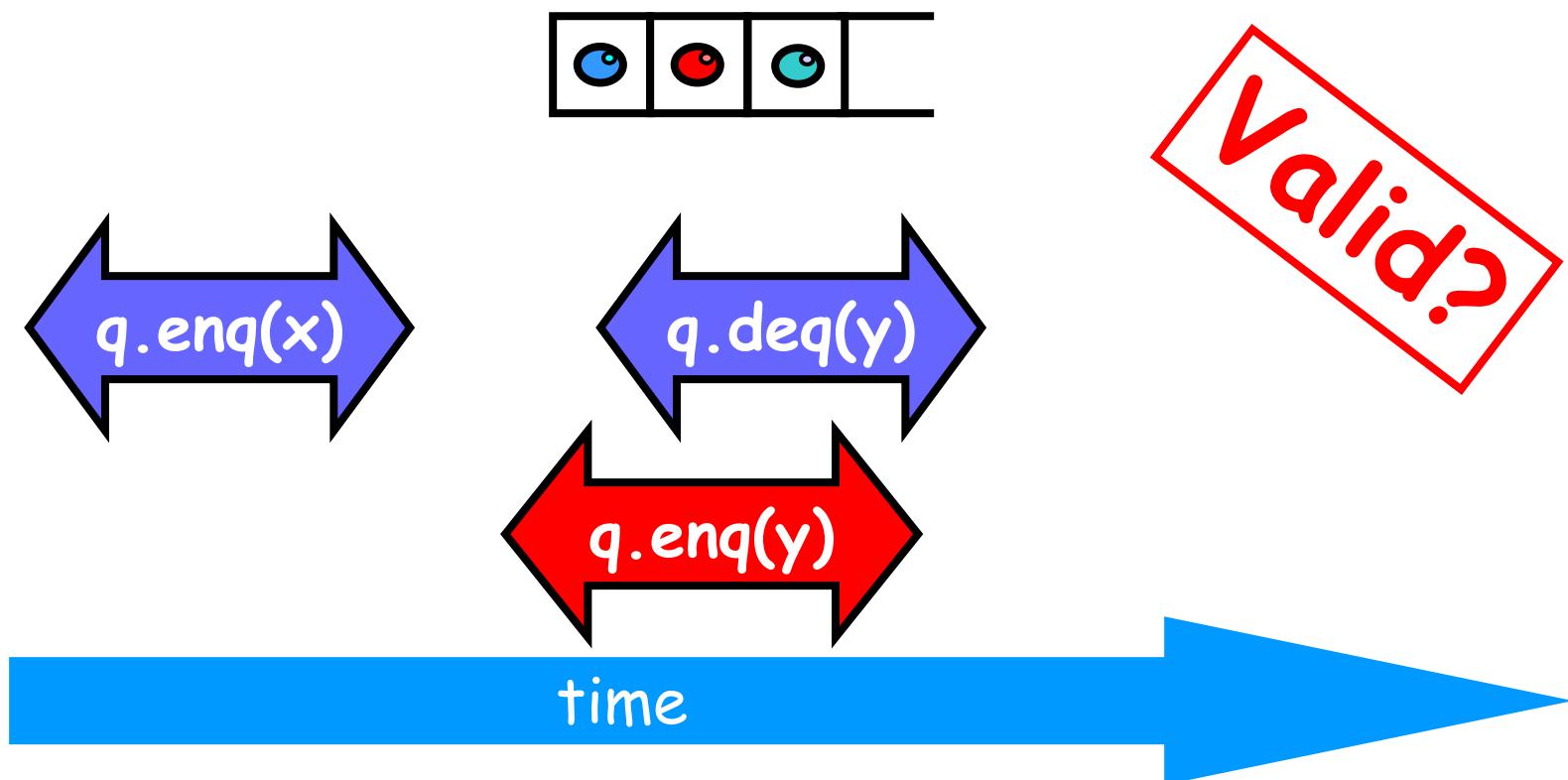
# Sequential consistency

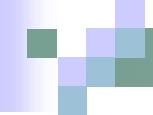
- A.enq(x) concurrent with B.enq(y), then  
A.deq(y) concurrent with B.deq(x)
- Two possible sequential orders:
  - A.enq(x) → B.enq(y) → B.deq(x) → A.deq(y)
  - B.enq(y) → A.enq(x) → A.deq(y) → B.deq(x)
- Both are in program order

# Example



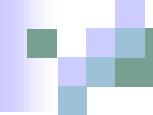
# Example





# Consistency

- Quiescent and sequential consistency are incomparable:
  - The one does not necessarily exist when the other exists
- Quiescent consistency does not necessarily preserve program order
- Sequential consistency is unaffected by quiescent periods



# Sequential consistency

- Sequential consistency is not compositional



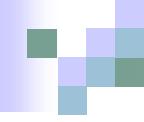
# Principle 3.5.1

- Each method call should appear to take effect instantaneously at some moment between its invocation and response



# Linearizability

- Principle 3.5.1 defines a third correctness property:
  - Linearizability
- Each linearizable execution is sequentially consistent, but not vice versa



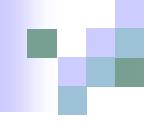
# Linearizability

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Object is correct if this “sequential” behavior is correct
- Any such concurrent object is
  - **Linearizable™**



# Linearizability

- To show that a concurrent object is linearizable one should identify for each method a linearization point where the method takes effect



# Linearization points

- For lock-based implementations:
  - Critical section
- For other methods:
  - The single step where the effects of the method call become visible to other methods



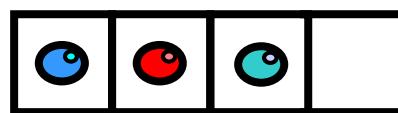
# Linearizability

- Sequential consistency is good way to describe standalone systems
- Linearizability is good way to describe components of large system

# Single-enqueuer/single-dequeuer

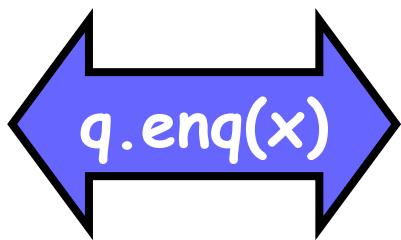
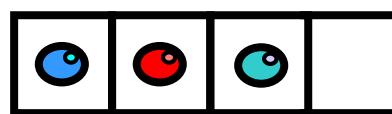
- No critical section
- Linearization points depend on execution
- If `deq()` returns a value:
  - Linearization point = head field is updated
- If list is empty:
  - Linearization point = `deq()` throws an exception

# Example



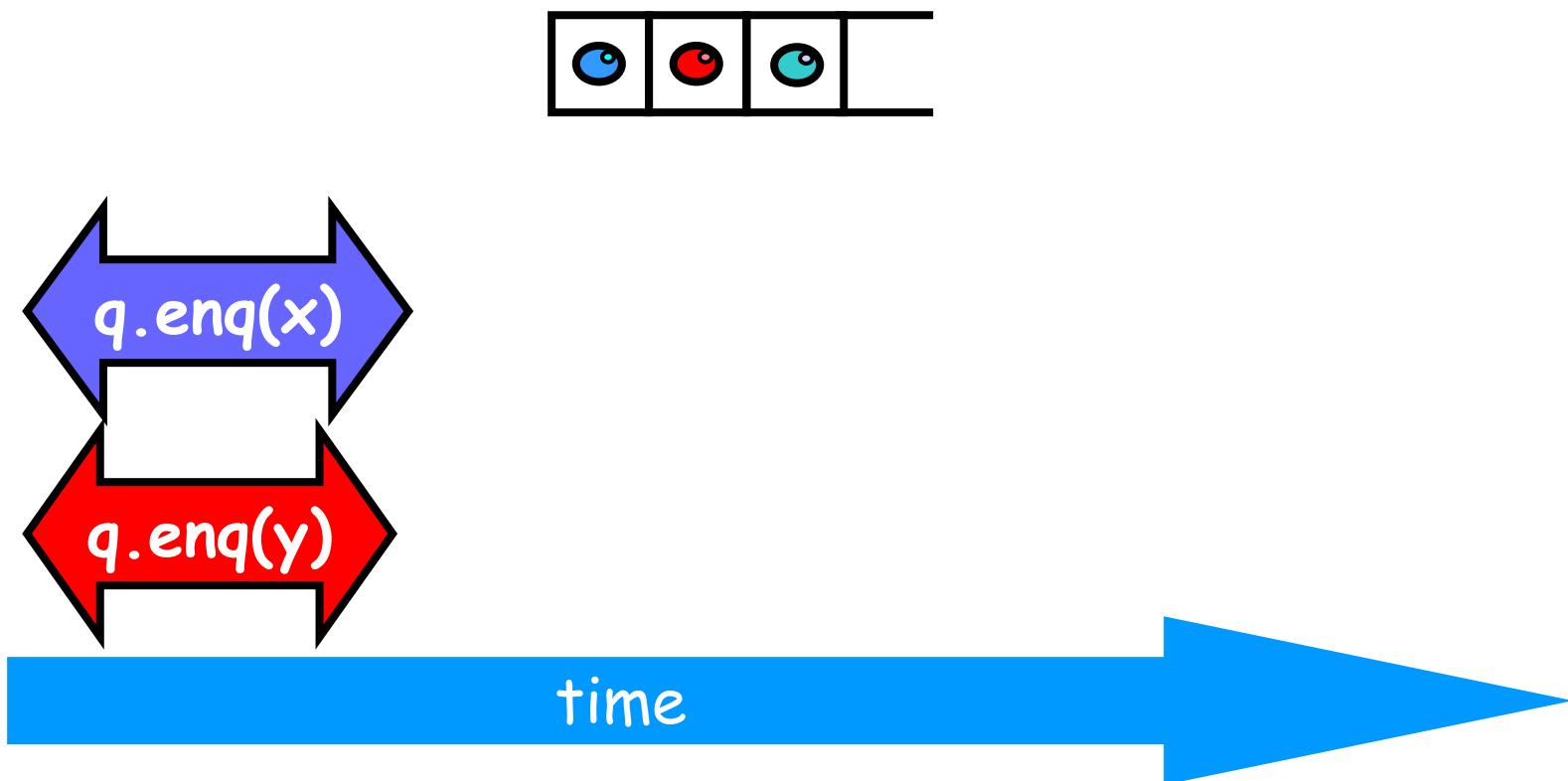
time

# Example

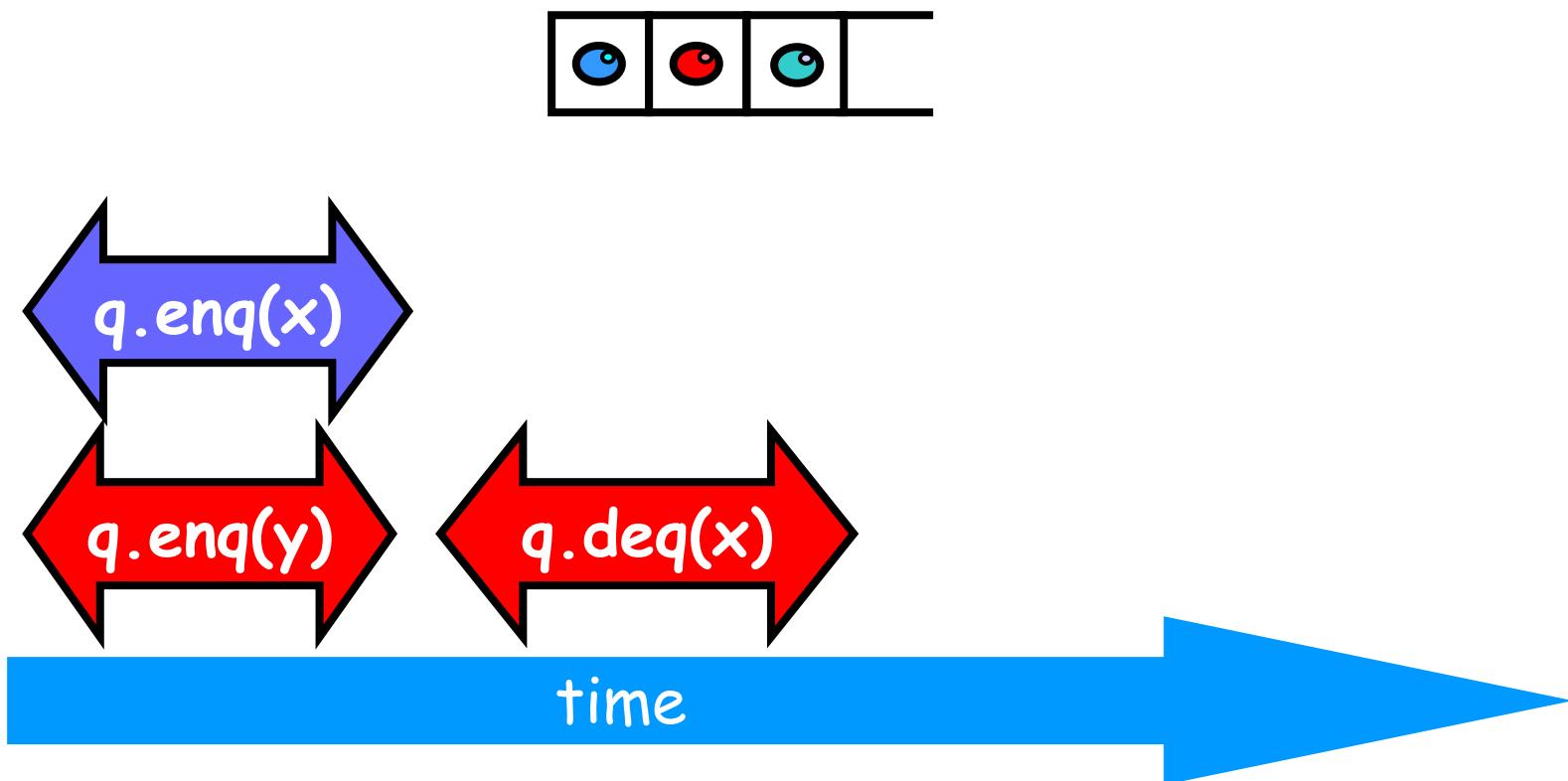


time

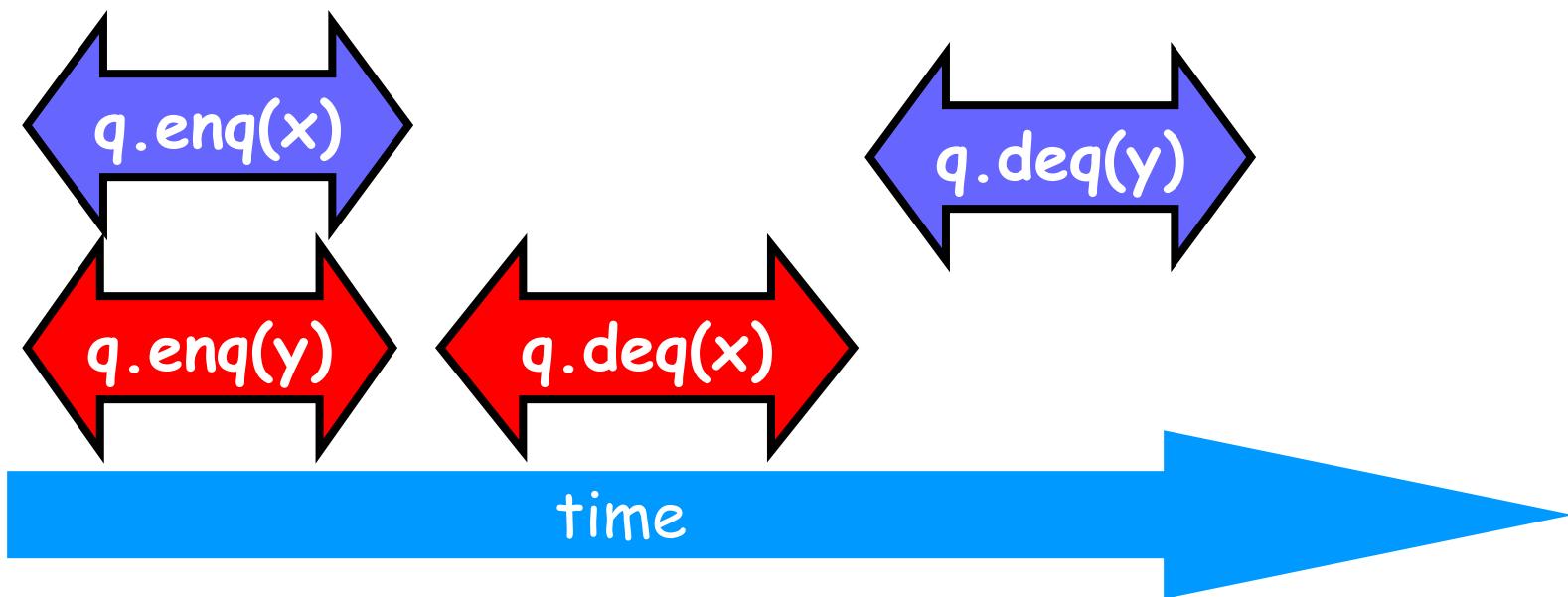
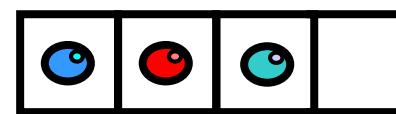
# Example



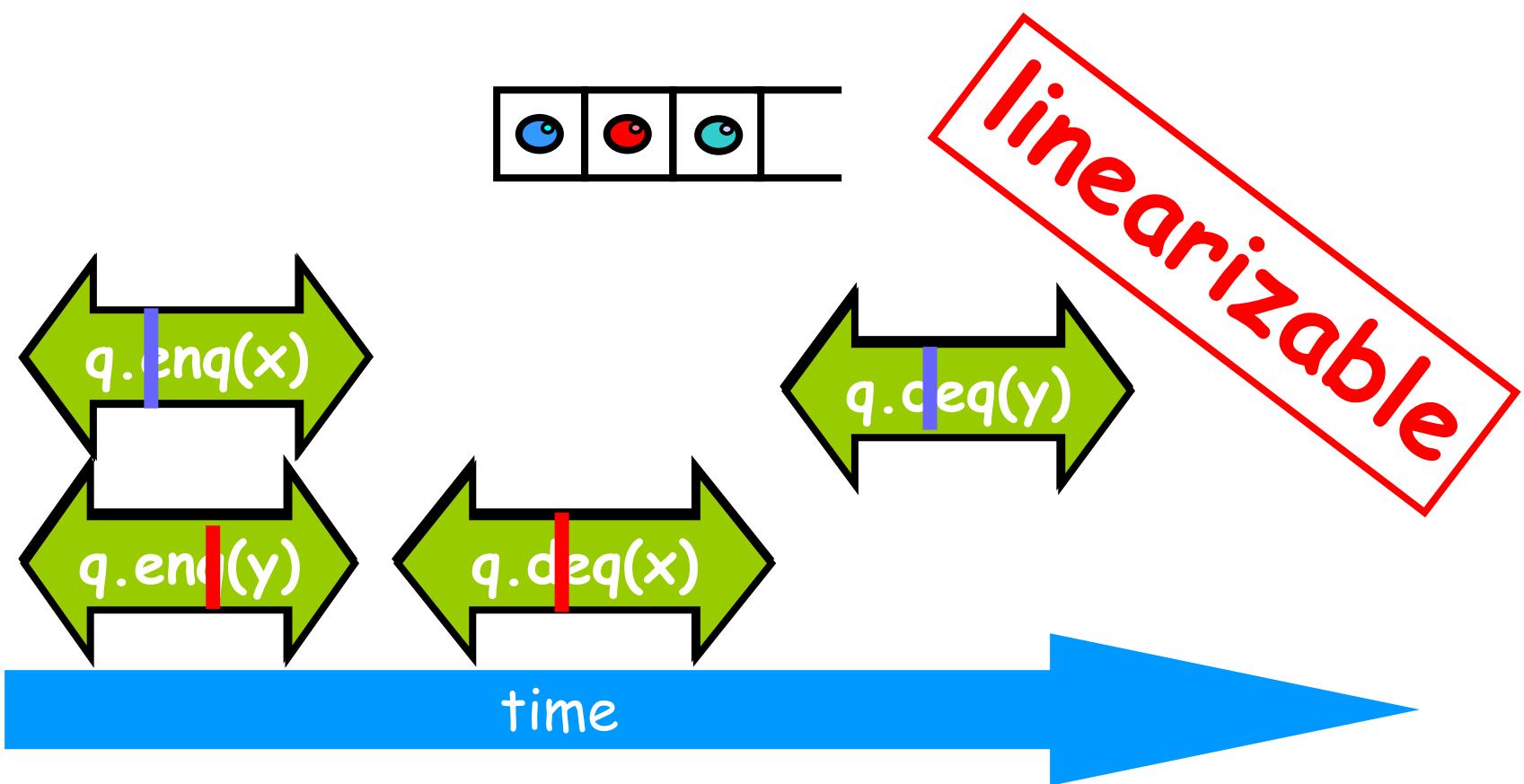
# Example



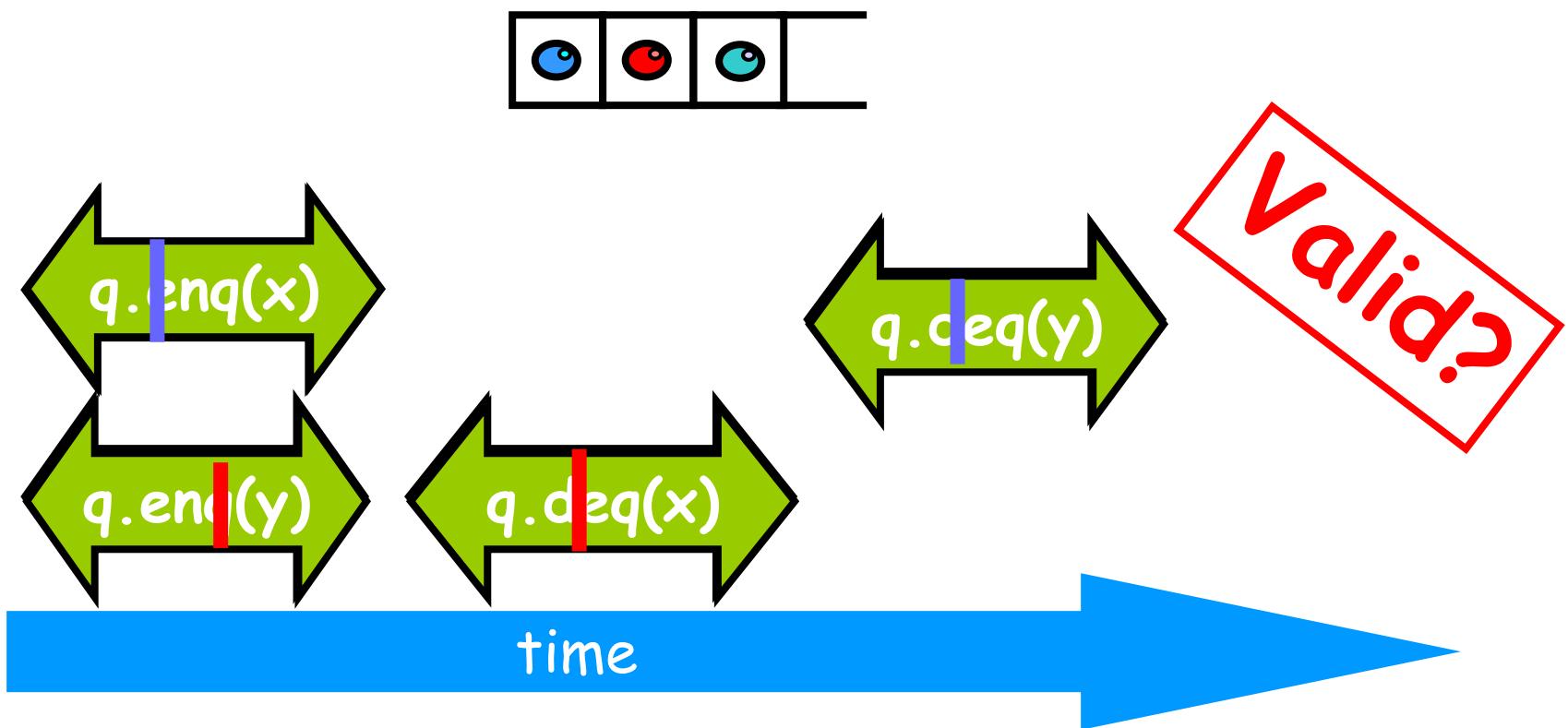
# Example



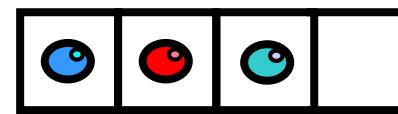
# Example



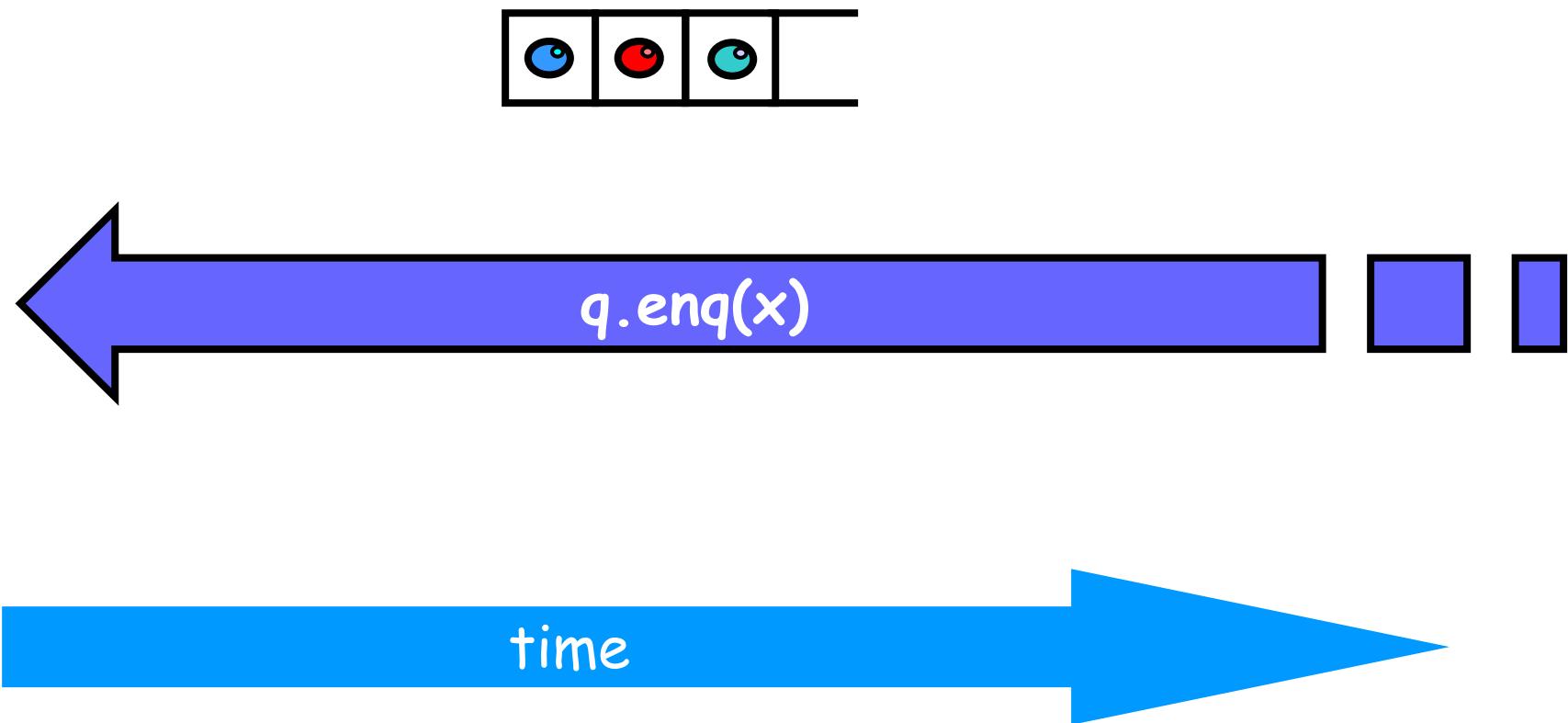
# Example



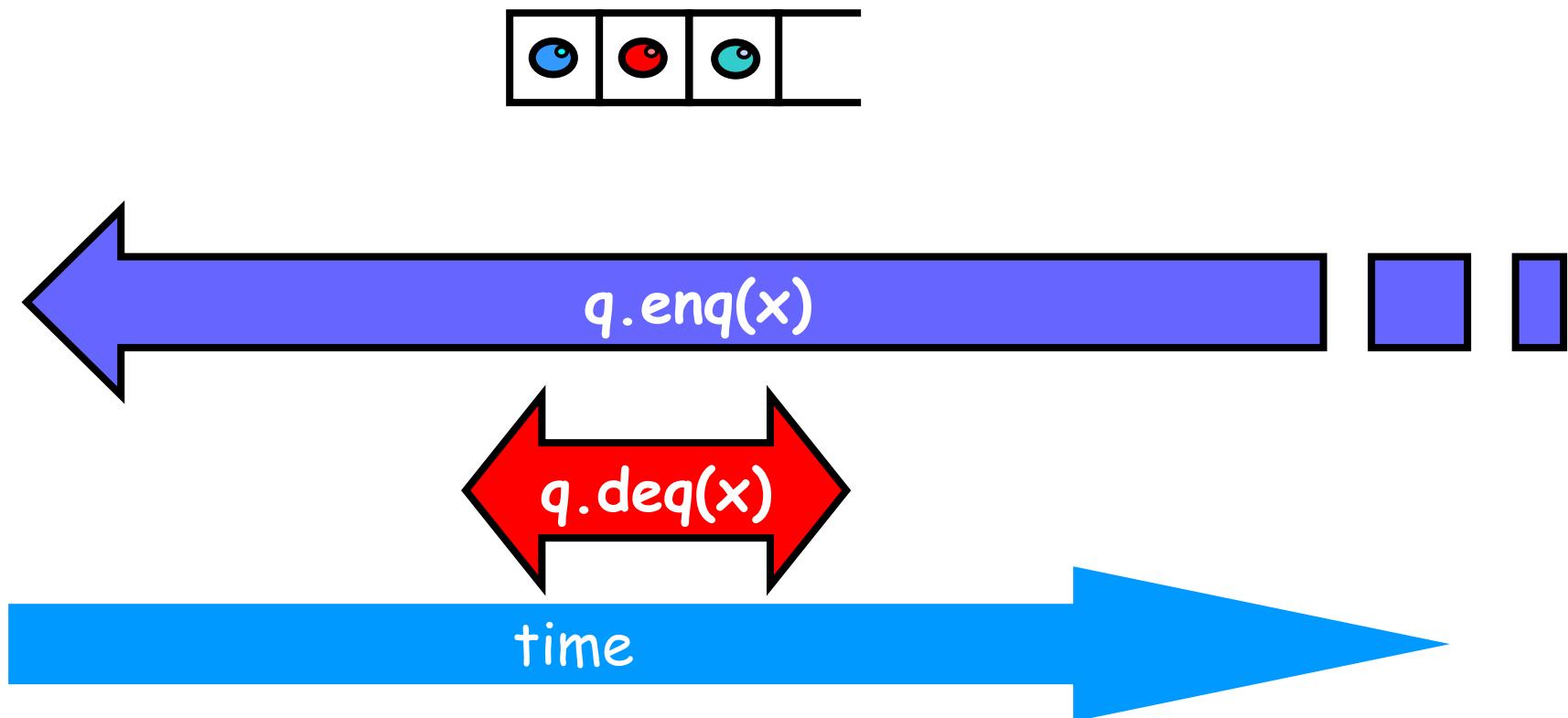
# Example



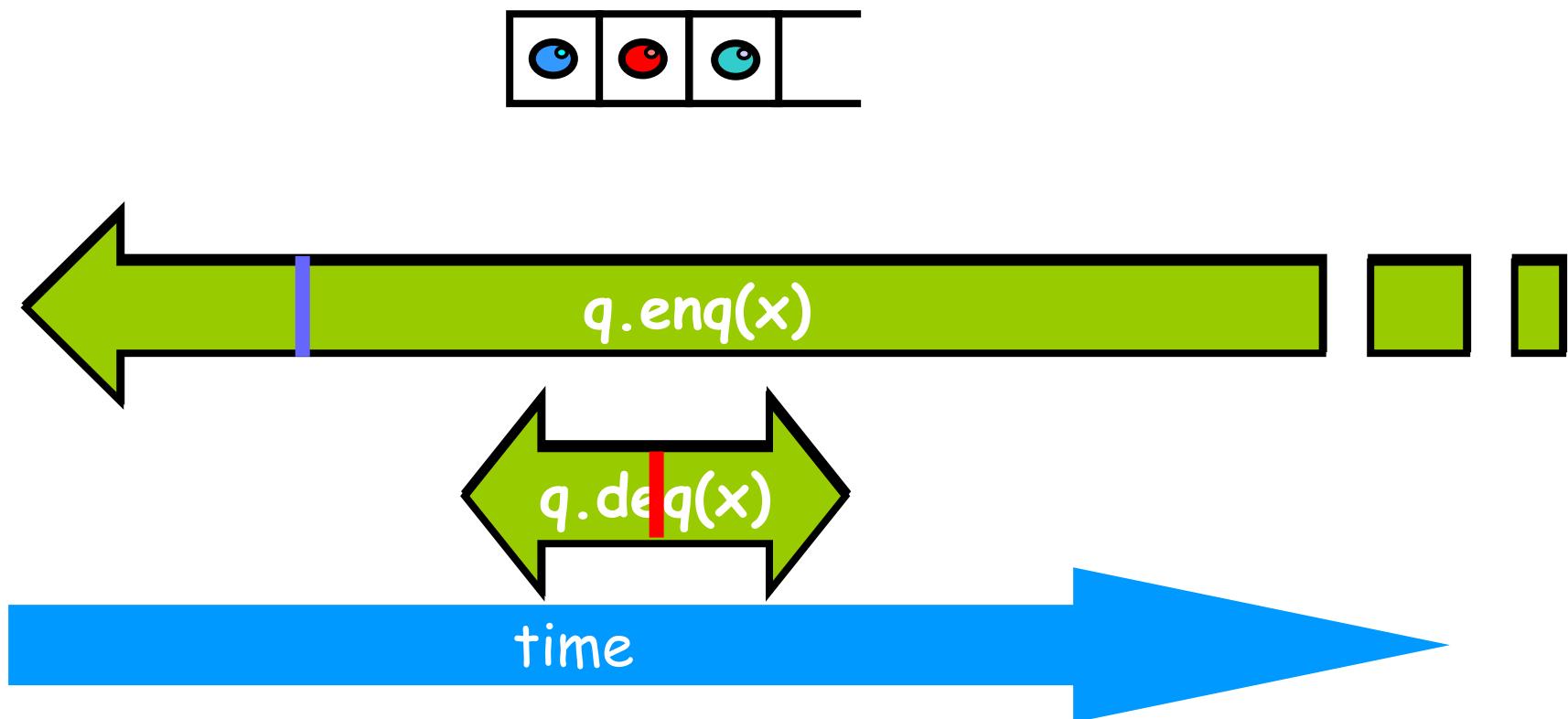
# Example



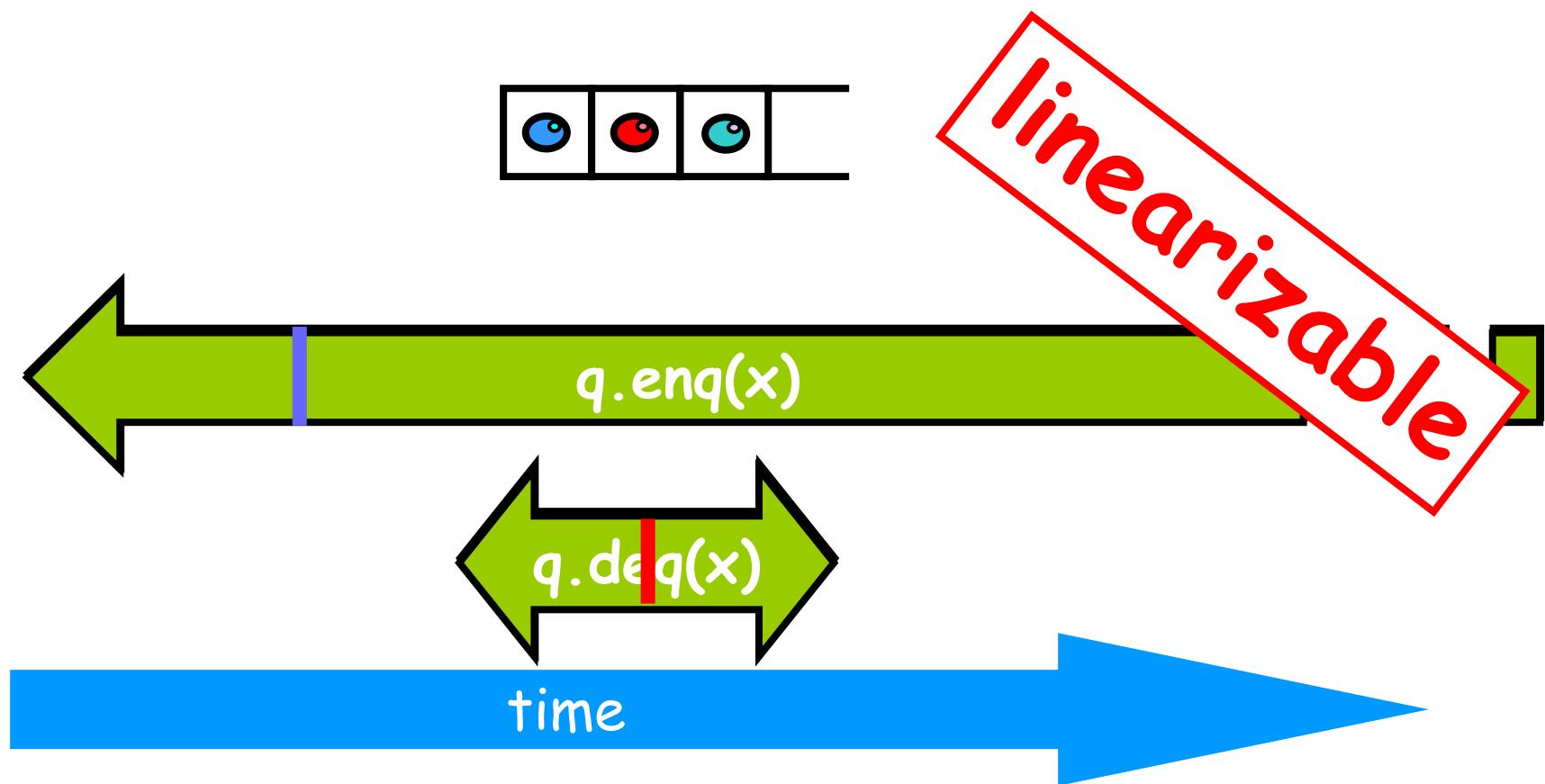
# Example



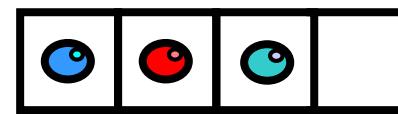
# Example



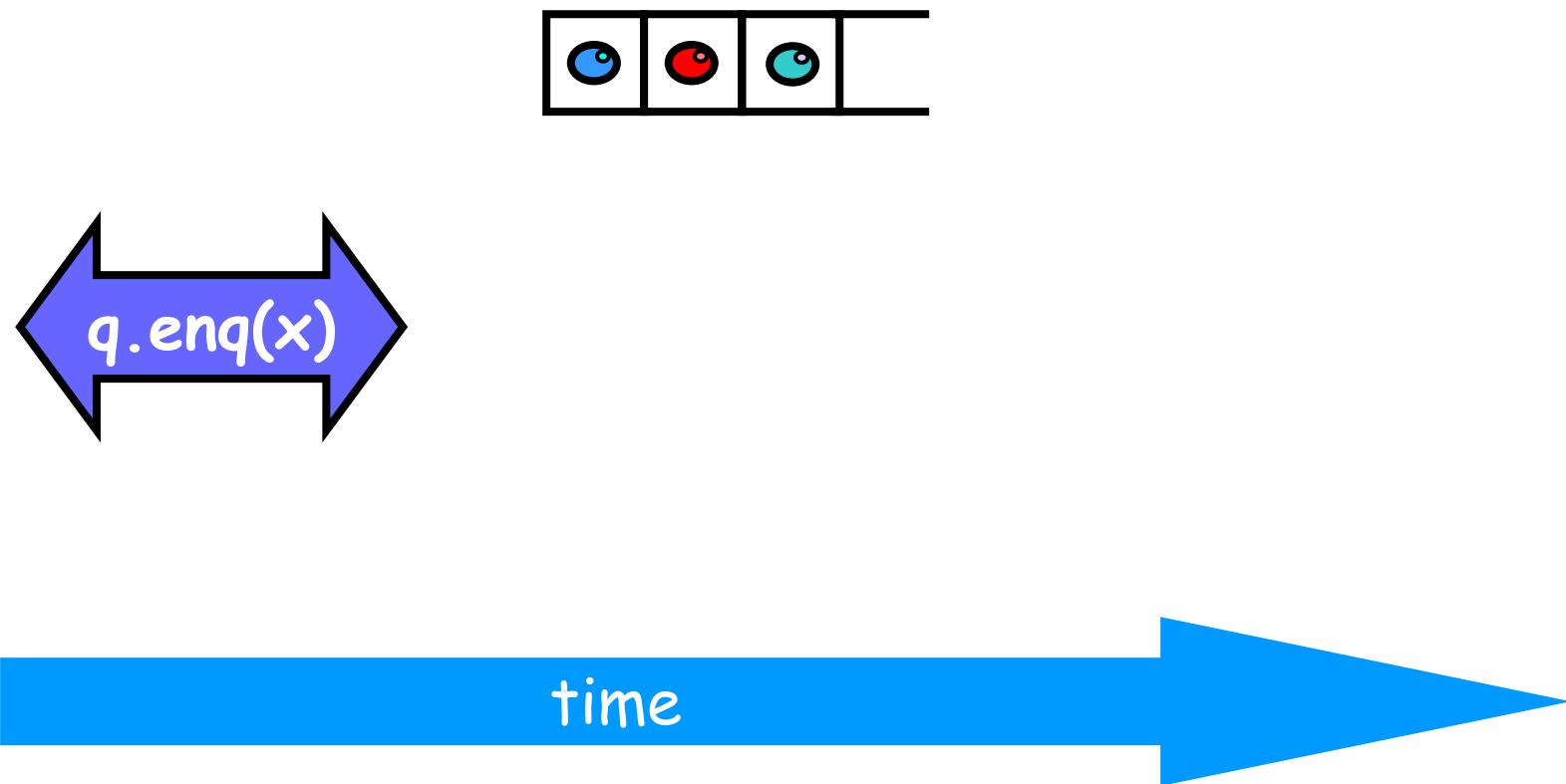
# Example



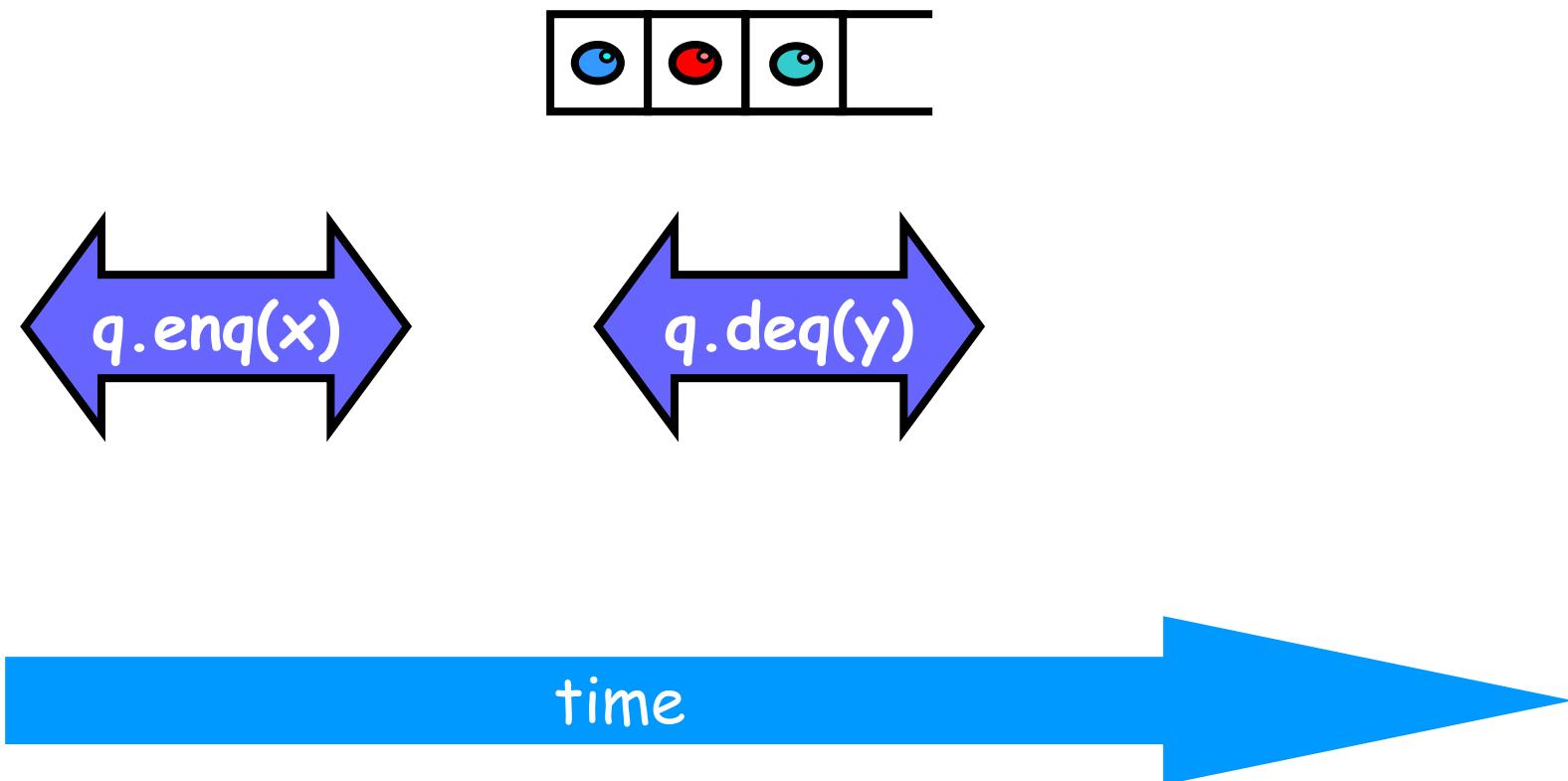
# Example



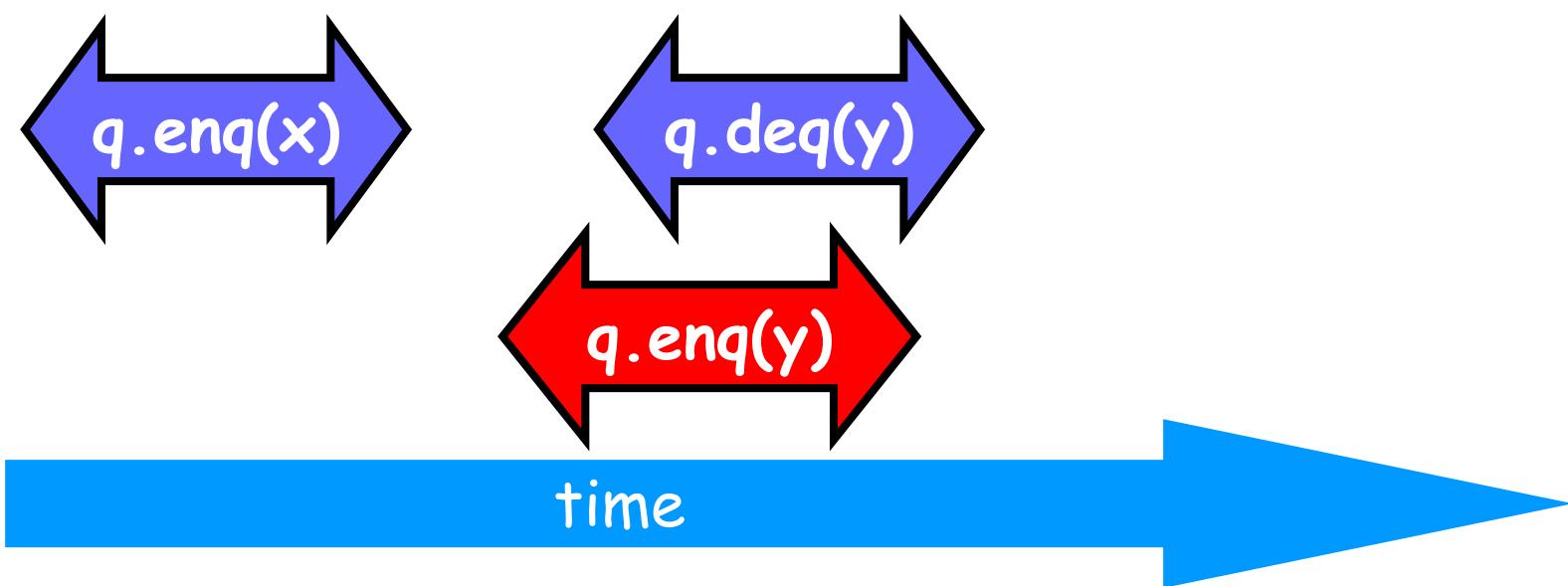
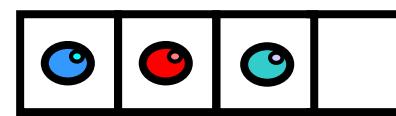
# Example



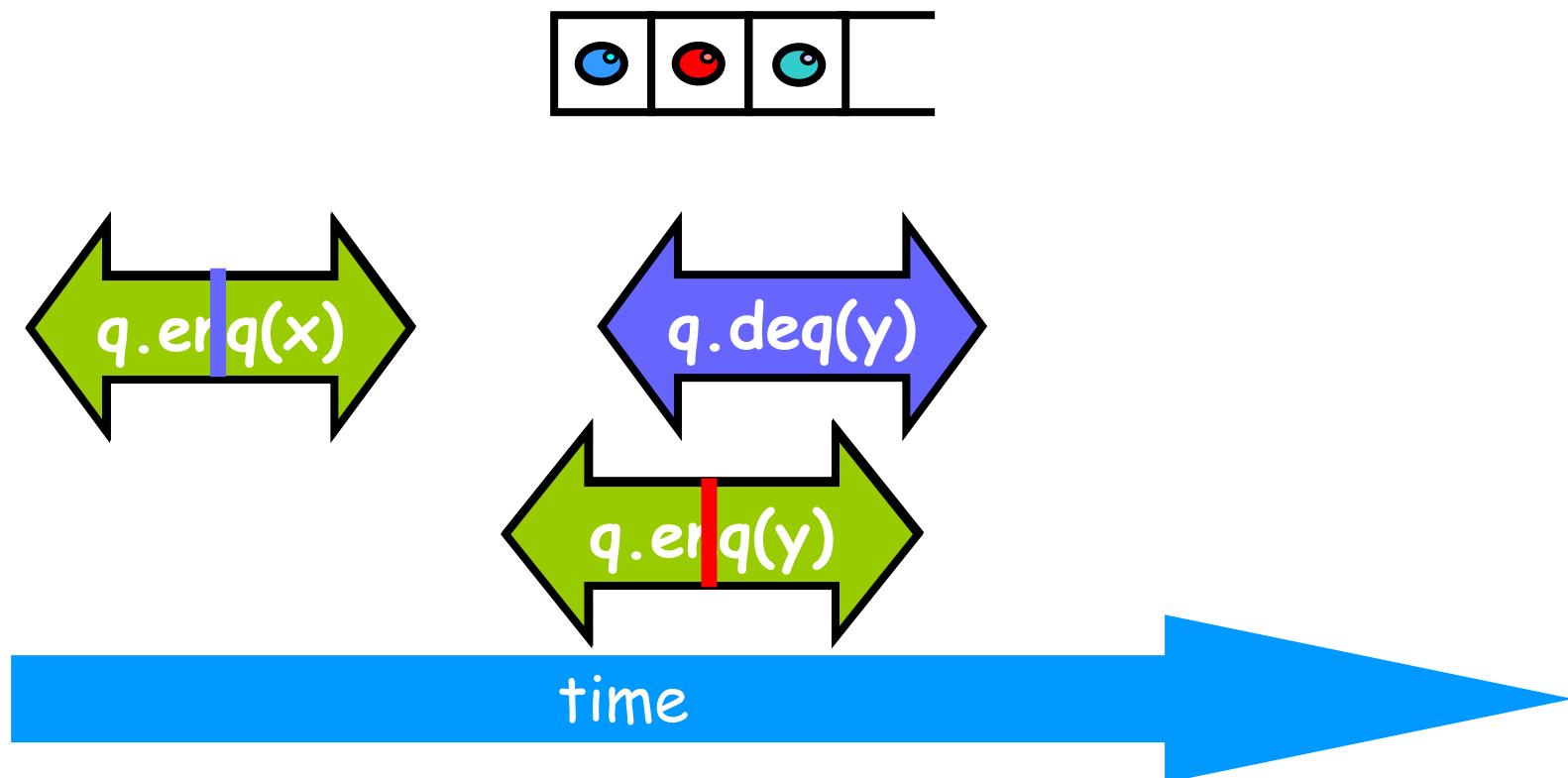
# Example



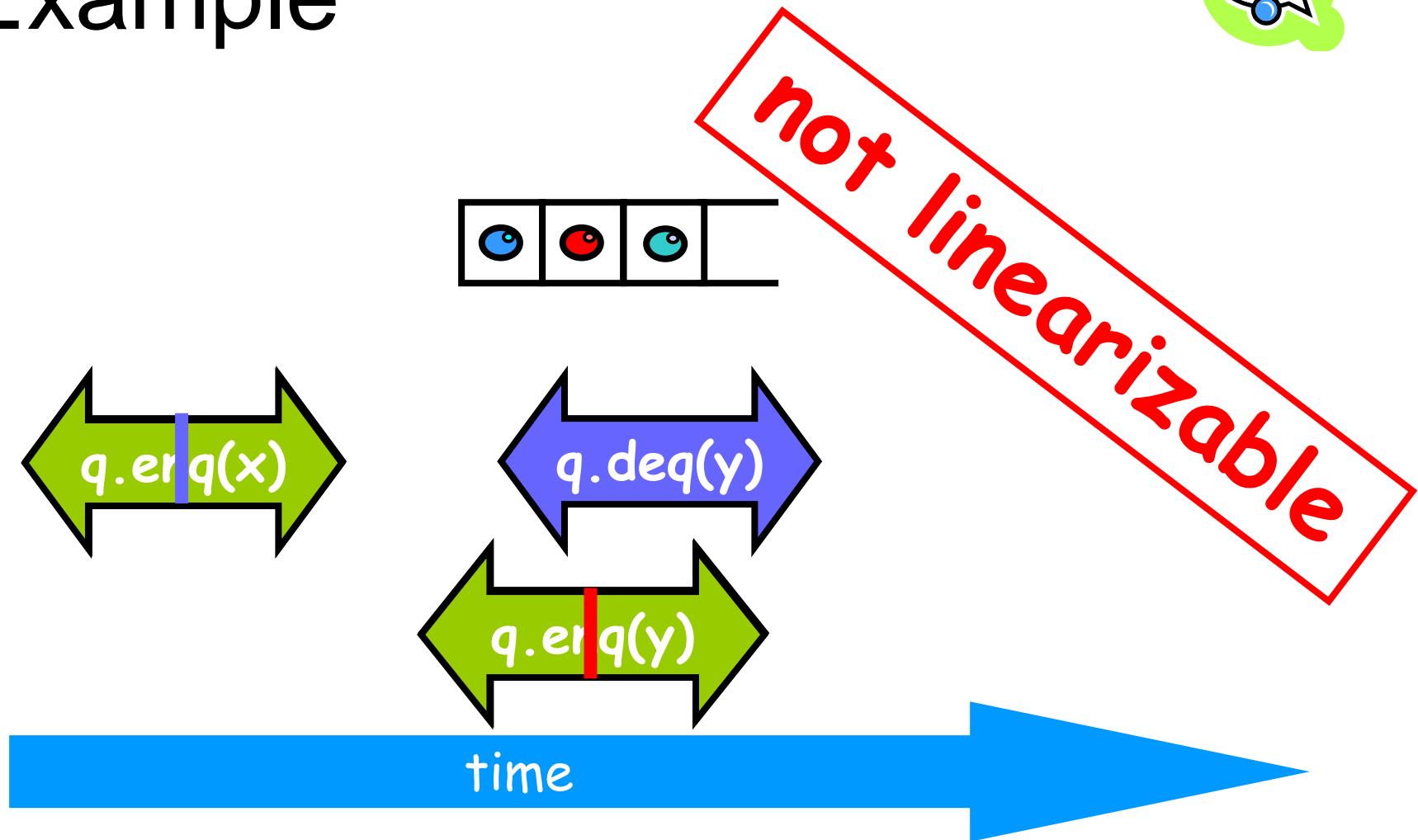
# Example

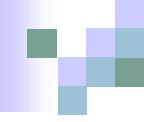


# Example



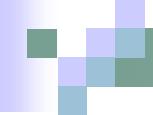
# Example





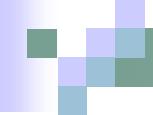
# Correctness

- Three correctness conditions:
  - Quiescent consistency
    - Applications that require high performance with weak constraints on object behaviour
  - Sequential consistency
    - Describe low-level systems such as hardware memory interfaces
  - Linearizability
    - Describe higher-level systems composed of linearizable components



# Correctness

- Safety property
- Deals with correctness of concurrent execution
  - In correct order?
  - No collisions?



# Quiescent consistency

- Checks that method calls appear to be made in sequential order
  - If write 7 and then -3 a read should not be -7
- AND
- Checks that method calls are in real-time order
  - We do not care about the order of concurrent method calls, but when separated by a period of inactivity, method calls should take place in the correct order



# Sequential consistency

- Checks that method calls appear to be made in sequential order
  - If write 7 and then -3 a read should not be -7
- AND
- Checks that method calls are made in program order
  - If write 7 and then -3 a read should not be 7



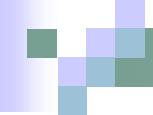
# Linearizability

- Checks that method calls appear to take place instantaneously
- Linearization points
  - If one method's linearization point is in the correct program order than a overlapping method, those methods are linearizable



# Progress

- Liveness property
- Deals with if different threads have to wait
  - For how long?
  - Will they ever reach the critical section?



# Progress

- Progress guarantees can be either:
  - Blocking
    - Delay of any one thread can delay others
  - Non-blocking
    - Delay of one thread cannot delay the others

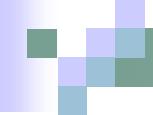


# Lock-free

- A method is lock-free if **some** method calls finishes in a finite number of steps

# Wait-free

- A method is wait-free if it **guarantees** that **every call** finishes its execution in a finite number of steps
- It is *bounded* wait-free if there is a limit on the number of steps a method call can take



# Lock-free vs. wait-free

- Any wait-free implementation is lock-free, but not vice versa



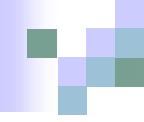
# Lock-free vs. wait-free

- A non-blocking algorithm is:
  - Lock-free if there is guaranteed system-wide progress
  - Wait-free if there is also per-thread progress



# Progress Conditions

- *Deadlock-free*: some thread trying to acquire the lock eventually succeeds.
- *Starvation-free*: every thread trying to acquire the lock eventually succeeds.
- *Lock-free*: some thread calling a method eventually returns (succeeds)
- *Wait-free*: every thread calling a method eventually returns (succeeds)



# Progress Conditions

- *Deadlock-free*: some thread trying to acquire the **lock** eventually succeeds.
- *Starvation-free*: every thread trying to acquire the **lock** eventually succeeds.
- *Lock-free*: some thread calling a method eventually returns (succeeds)
- *Wait-free*: every thread calling a method eventually returns (succeeds)



# Progress Conditions

	Non-Blocking	Blocking
Everyone makes progress	Wait-free	Starvation-free
Someone makes progress	Lock-free	Deadlock-free



# Java Memory Model

- Java programming language does not guarantee linearizability when reading and writing fields of shared objects
- Due to compiler optimization memory reads and writes are often reordered

# Singleton object

```
public static Singleton getInstance() {  
    if (instance == null)  
        instance = new Singleton();  
    return instance;  
}
```

Problem



# Singleton object

- Create a single instance of the class
- Method must guard against multiple threads each seeing instance to be null and create new instances

# Singleton object

```
public static Singleton getInstance() {  
    synchronized(this) {  
        if (instance == null)  
            instance = new Singleton();  
    }  
    return instance;  
}
```

**Lock down  
critical section to  
avoid collisions**

**But what about optimization?**



# Singleton object

- Once the instance has been created, however no further synchronization should be necessary

# Singleton object

```
public static Singleton getInstance() {  
    if (instance == null) {  
        synchronized(this) {  
            instance = new Singleton();  
        }  
    }  
    return instance; What if two threads call  
synchronized  
simultaneously?  
}
```

# Singleton object

```
public static Singleton getInstance() {  
    if (instance == null) {  
        synchronized(this) {  
            if (instance == null)  
                instance = new Singleton();  
        }  
    }  
    return instance;  
}
```

Double-checked locking

# Singleton object

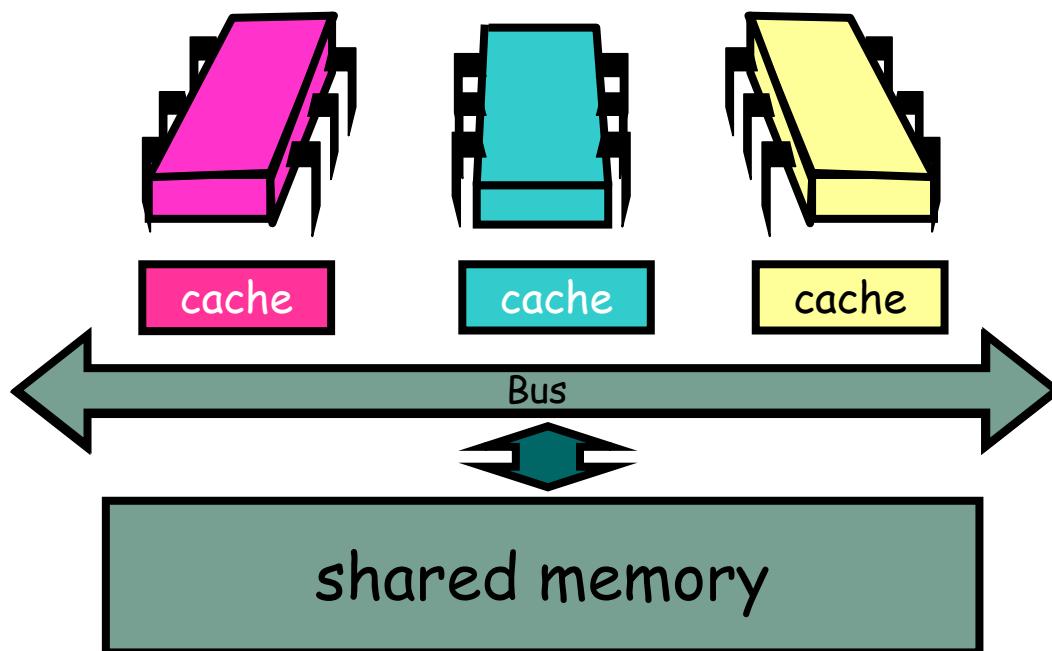
- In theory a double-checked lock is correct, however:
  - In theory, the constructor call takes place before the instance field is assigned
  - However, the java memory model allows these steps to occur out of order = making a partially initialized Singleton object visible to other programs

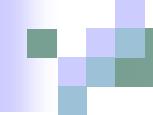


# Java Memory Model

- In the Java memory model:
  - Objects reside in shared memory
  - Each thread has a private working memory that contains cached copies of fields it has read or written

# Java Memory Model





# Java Memory Model

- In the absence of explicit synchronization:
  - A thread that writes to a field may not update the memory right away, and
  - A thread that reads from a field may not update its working memory if the field's value in memory changes



# Java Memory Model

- Need a way to force a thread to change the shared memory object when changing his own private memory copy and to read an object from the shared memory instead of reading from his private memory



# Synchronization events

- Synchronization usually implies mutual exclusion
- In Java, it also implies reconciling a thread's working memory with the shared memory



# Synchronization events

- In Java usually in one of two ways:
  - Cause a thread to write changes back to shared memory immediately
  - Cause thread to invalidate its working memory values and forces it to reread the fields from shared memory



# Synchronization events

- Synchronization events are linearizable:
  - They are ordered
  - All threads agree on the ordering



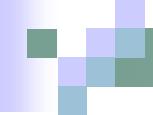
# Synchronization events

- Locks and synchronized blocks
- Volatile fields
- Final fields



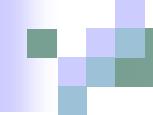
# Locks and synchronized blocks

- Thread can achieve mutual exclusion through implicit lock (synchronized block) or explicit lock
- If all accesses to a particular field are protected by the same lock, then the reads-writes to that field is linearizable



# Locks and synchronized blocks

- When a thread releases a lock the changes are written to shared memory immediately
- When a thread acquires a lock it invalidates its own memory and rereads the value from shared memory



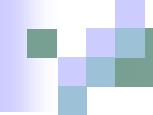
# Volatile fields

- Reading a volatile field is like acquiring a lock – value is reread from shared memory
- Writing a volatile field is like releasing a lock – changes immediately written to shared memory



# Volatile fields

- However, multiple reads-writes are not successful
- Some form of mutual exclusion is then needed



# Final fields

- A field declared to be final cannot be modified once it has been initialized in its constructor

# Final fields

```
class FinalFieldExample {  
    final int x; int y;  
    static FinalFieldExample;  
    public FinalFieldExample() {  
        x = 3;  
        y = 4;  
    }  
    static void reader() {  
        int i = x; int j = y;  
    }  
}
```

**Correct!**  
**Thread that calls  
reader() is guaranteed  
to see x equal to 3**

# In summary

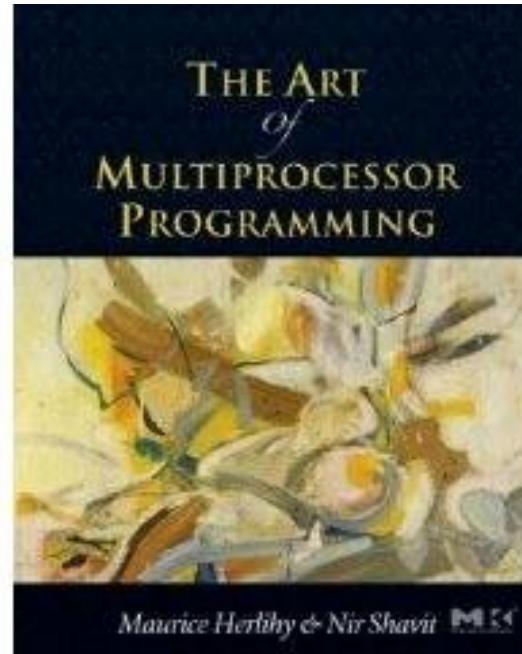
- Reads-writes to fields are linearizable if:
  - The field is volatile
  - The field is protected by a unique lock used by all readers and writers

COS 226

# Chapter 4

# Foundations of Shared Memory

# Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

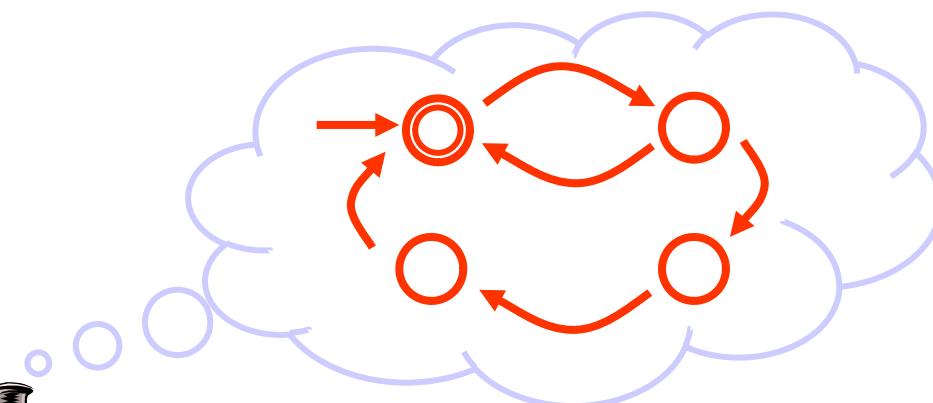
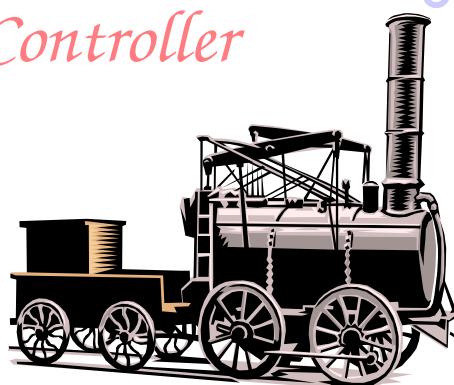
# Church-Turing Thesis



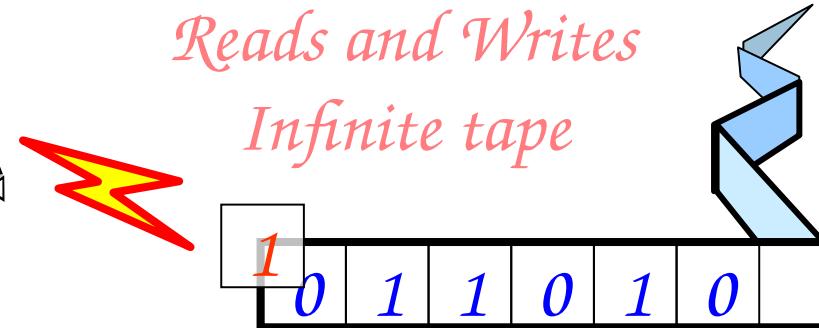
- Anything that can be computed, can be computed by a Turing Machine.
- Foundations of sequential computing

# *Turing Machine*

*Finite State  
Controller*



*Reads and Writes  
Infinite tape*





# Concurrent shared-memory computing

- Consists of multiple threads – each a sequential program
- That communicate by calling methods of objects in shared memory



# Threads

- Threads are asynchronous
  - They run at different speeds and can be halted for an unpredictable duration at any time



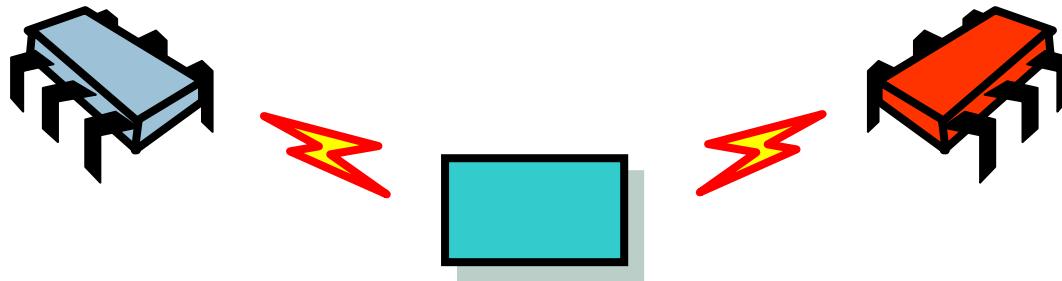
# *Shared-Memory Computability?*



- Mathematical model of **concurrent** computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant

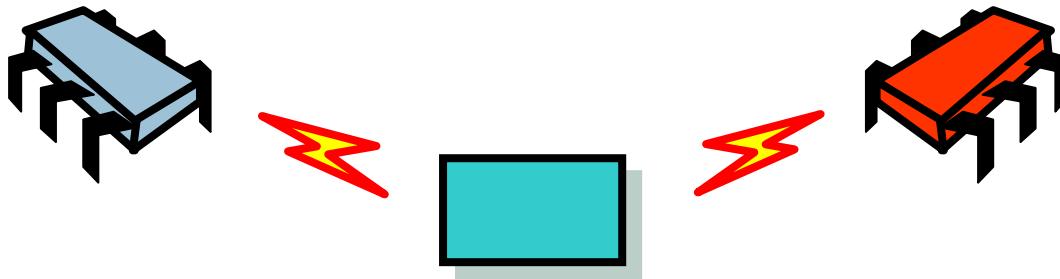
# Foundations of Shared Memory

*To understand modern multiprocessors we need  
to ask some basic questions ...*



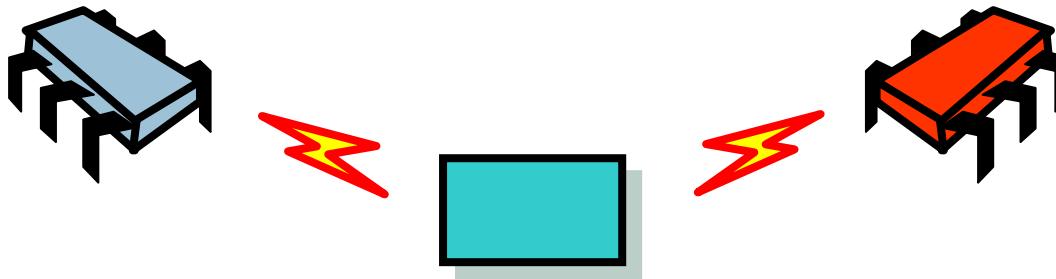
# Foundations of Shared Memory

*What is the weakest useful form of shared memory?*

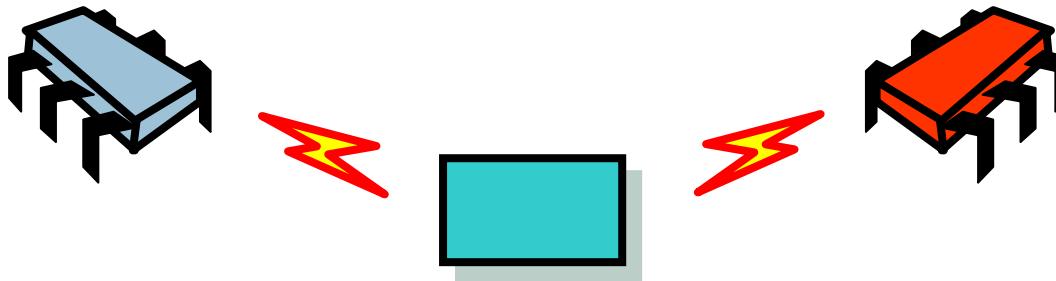
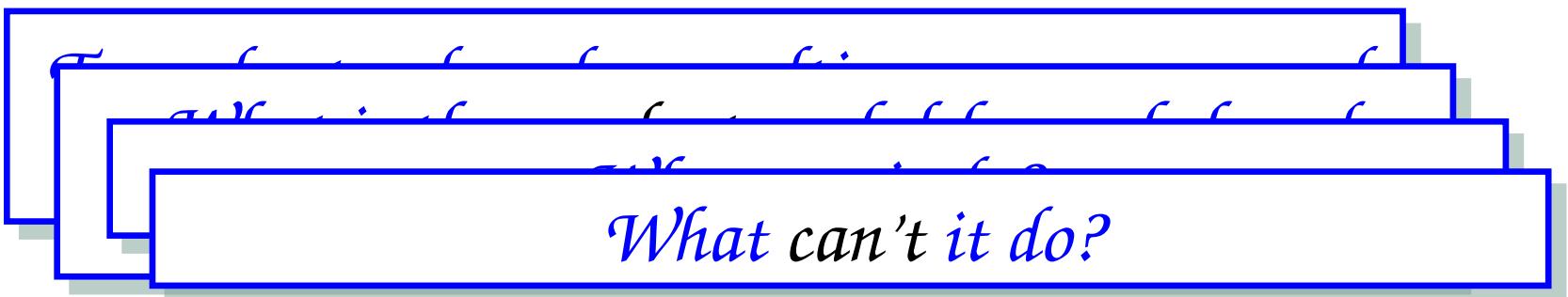


# Foundations of Shared Memory

*What can it do?*



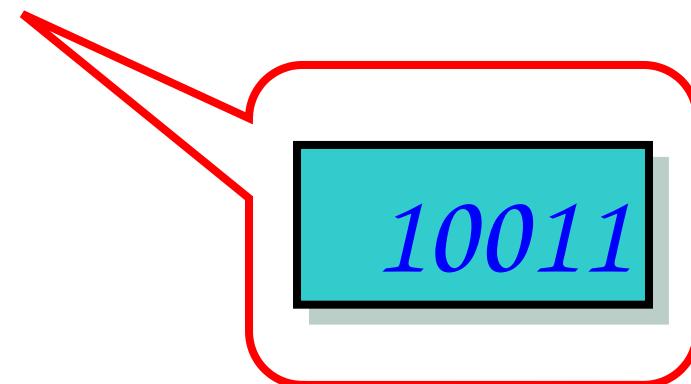
# Foundations of Shared Memory



# Register

\*

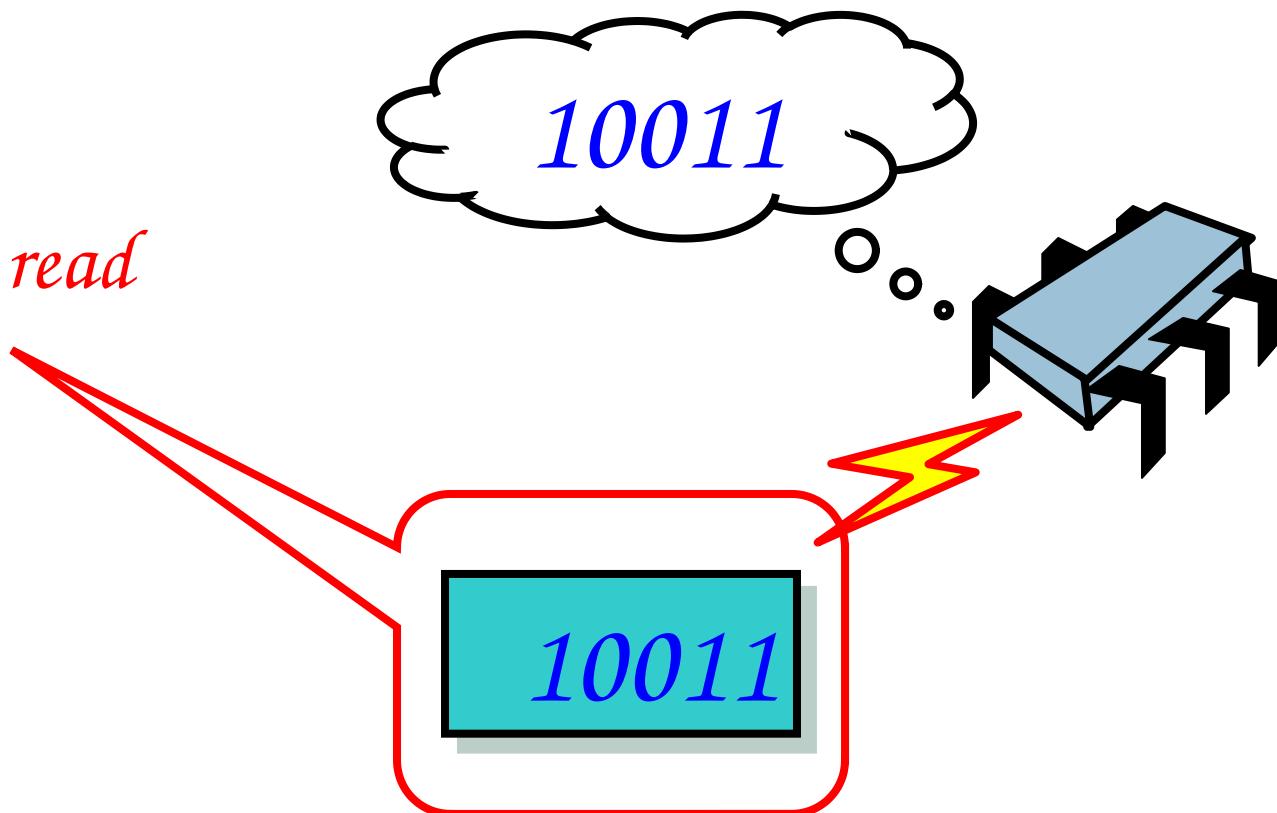
*Holds a (binary)  
value*



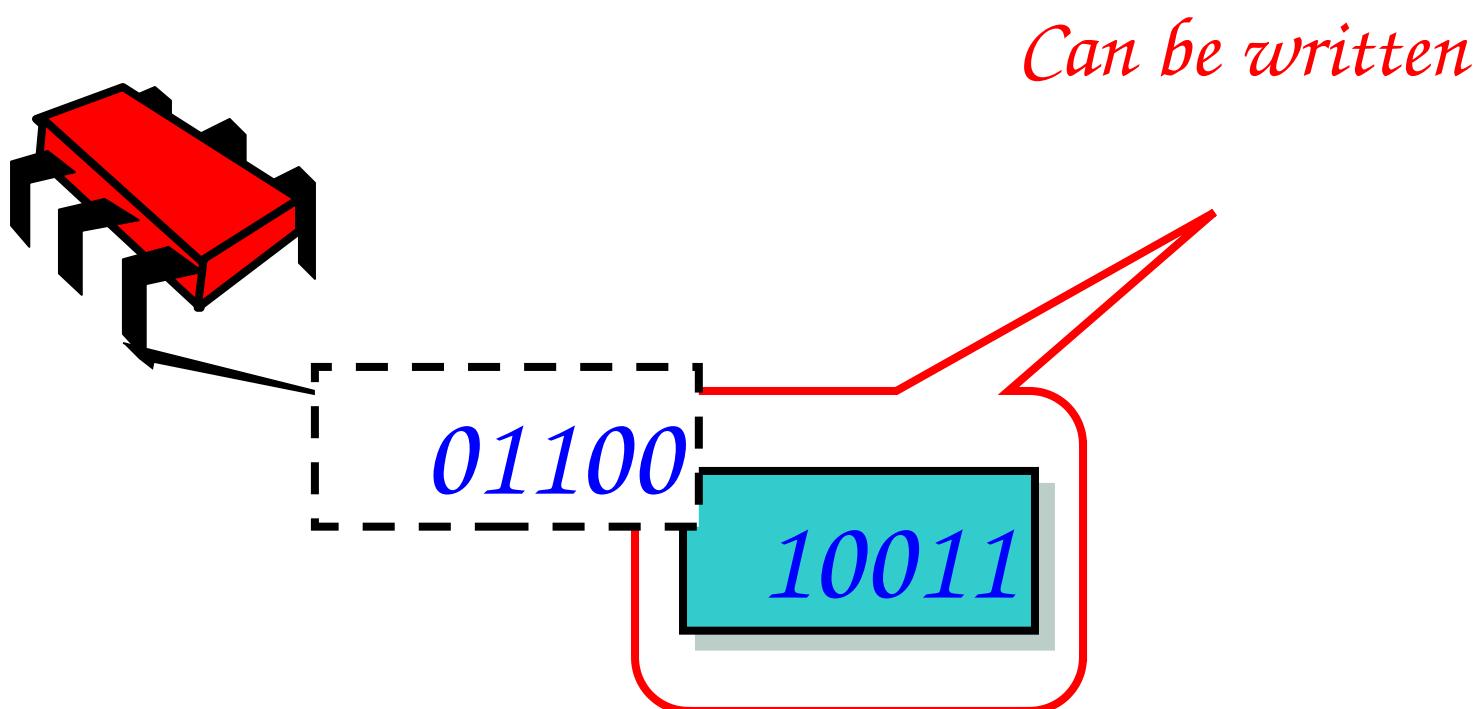
*\* A memory location: name is historical*

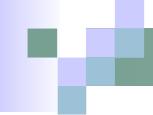
# Register

*Can be read*



# Register





# Registers

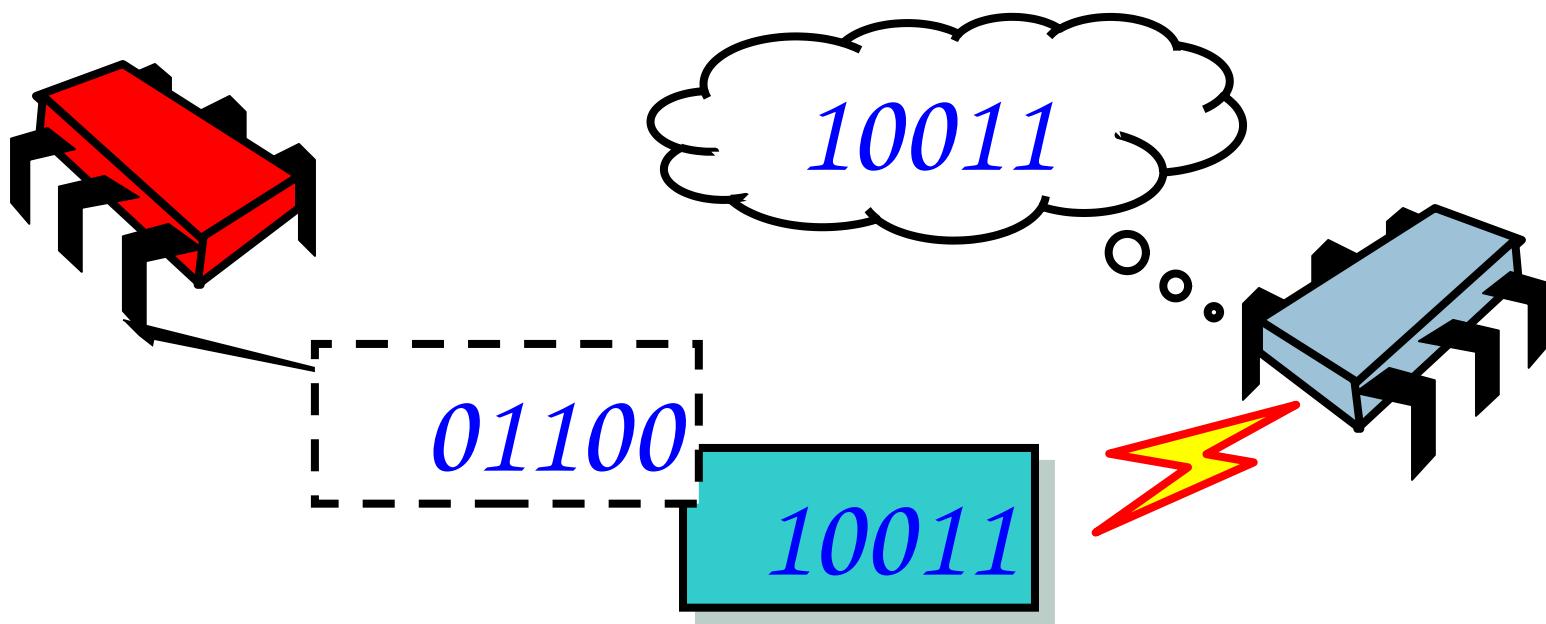
```
public interface Register<T> {  
    public T read();  
    public void write(T v);  
}
```

# Registers

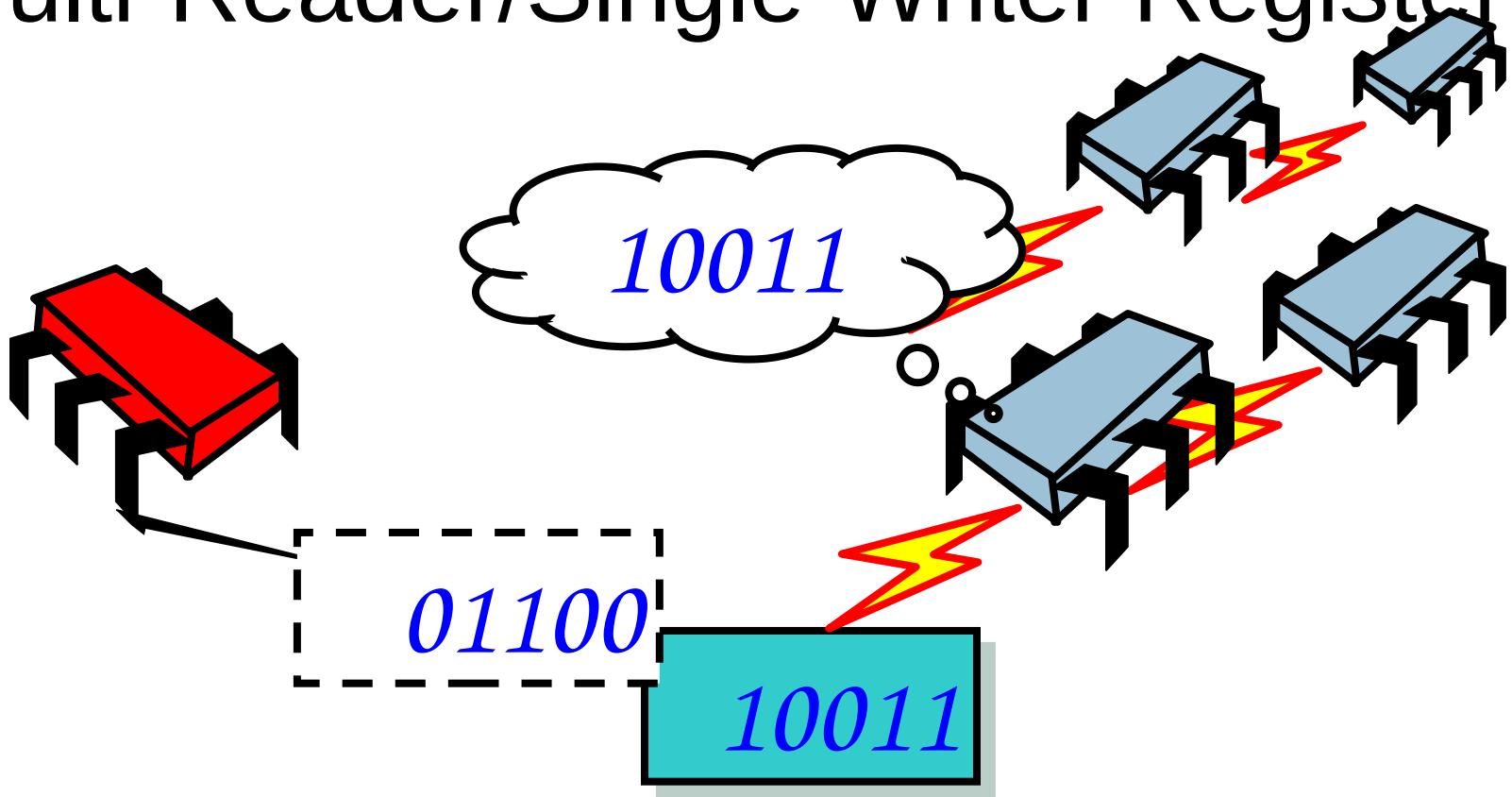
```
public interface Register<T> {  
    public T read();  
    public void write(T v);  
}
```

*Type of register  
(usually Boolean or m-bit Integer)*

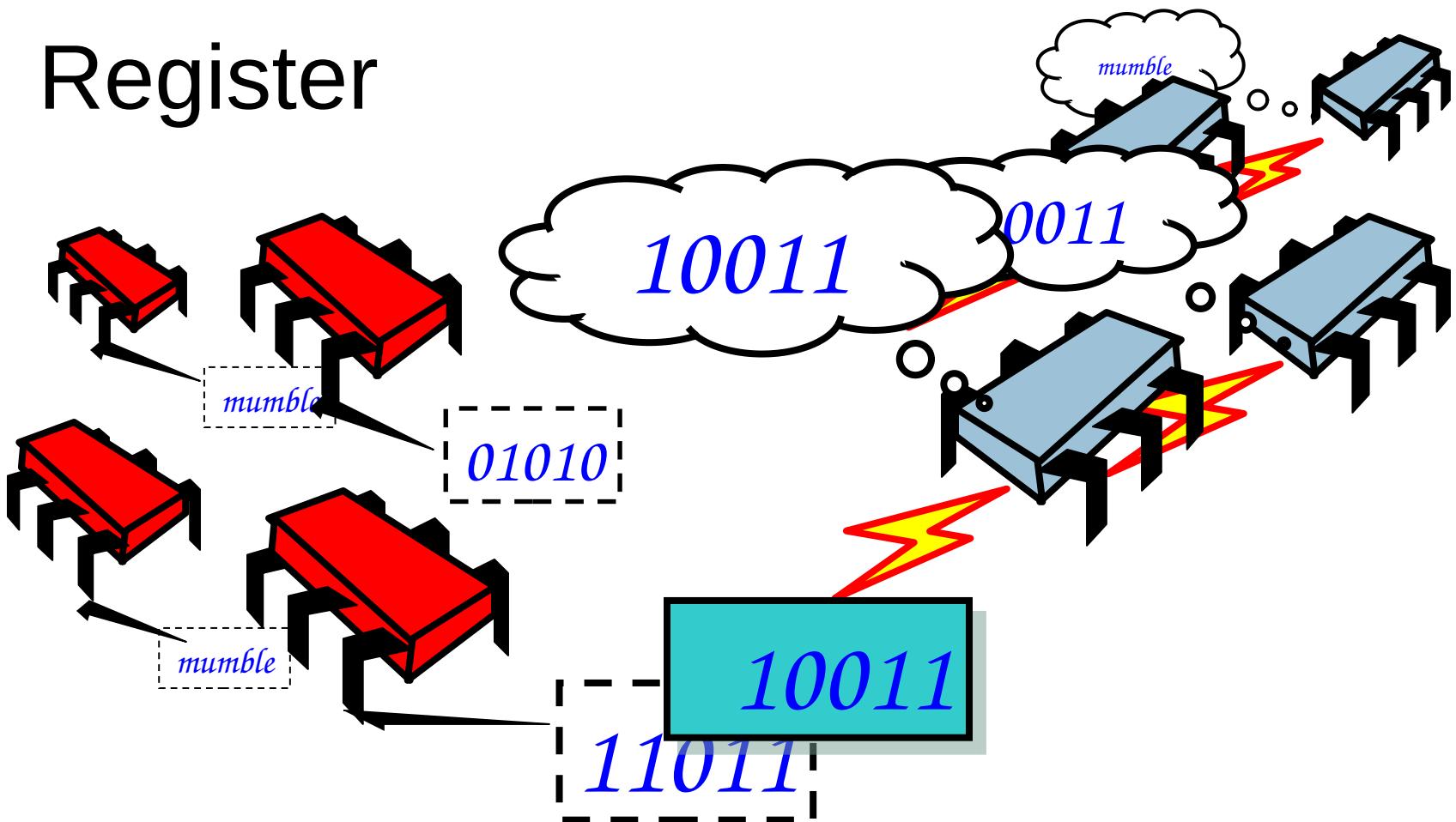
# Single-Reader/Single-Writer Register

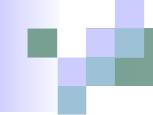


# Multi-Reader/Single-Writer Register



# Multi-Reader/Multi-Writer Register





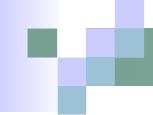
# Jargon Watch

- SRSW
  - Single-reader single-writer
- MRSW
  - Multi-reader single-writer
- MRMW
  - Multi-reader multi-writer



# Concurrent registers

- On a multiprocessor, we expect reads and writes to overlap
- How do we specify what a concurrent method call mean?



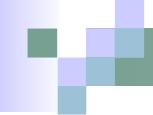
# One approach

- Rely on mutual exclusion:
  - Protect each register with a mutex lock acquired by each `read()` and `write()` call
  - Possible problems?

# Different approach: Wait-Free Implementation

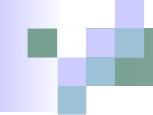
Definition: An object implementation is ***wait-free*** if every method call completes in a finite number of steps

- *No mutual exclusion*
- *Guarantees independent progress*
- *We require register implementations to be wait-free*



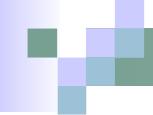
# Different kinds of registers

- According to:
  - Range of values
    - Boolean or Integer (M-valued)
  - Number of readers and writers
  - Degree of consistency



# Degree of consistency

- Safe
- Regular
- Atomic

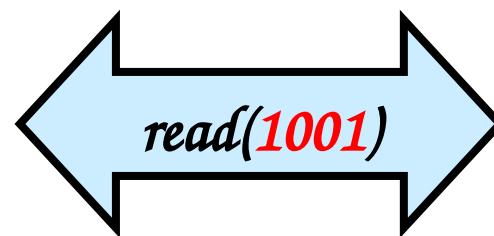
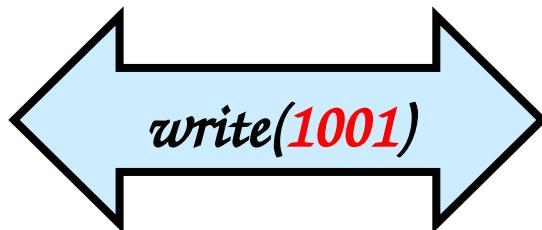


# Safe Register

- A single-writer, multi-reader register is safe if:
  - A read() that does not overlap a write() return the last value
  - If a read() overlaps a write() it can return any value within the register's range

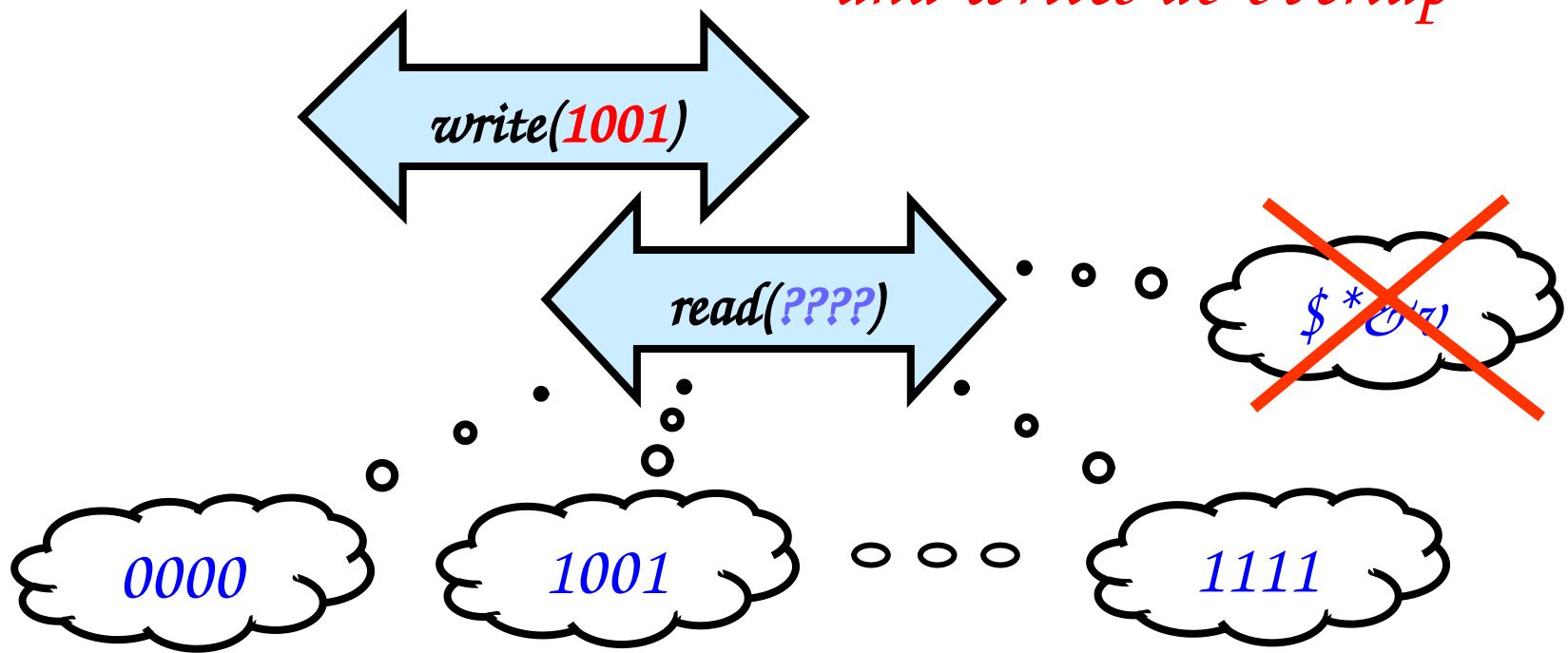
# Safe Register

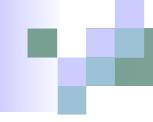
*OK if reads and  
writes don't  
overlap*



# Safe Register

*Some valid value if reads  
and writes do overlap*

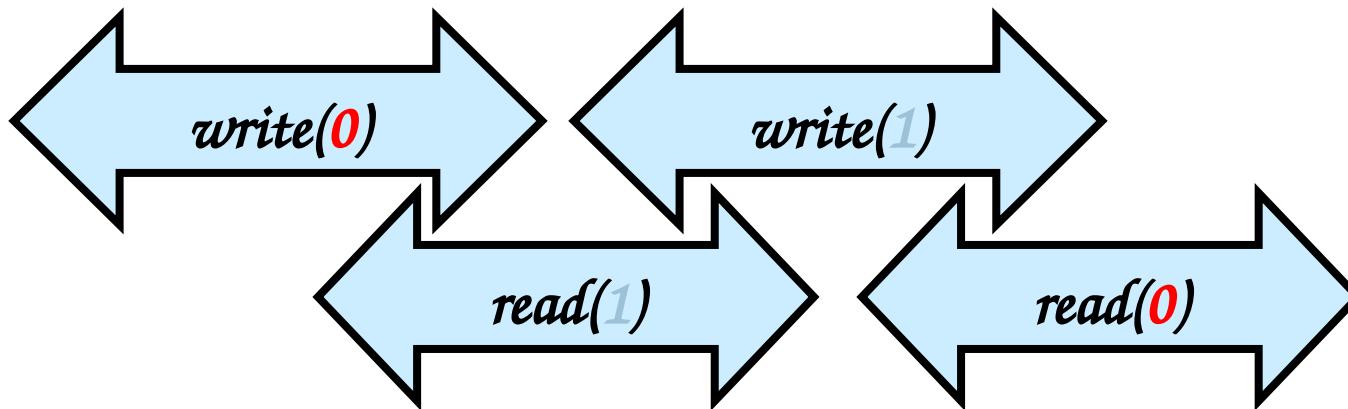




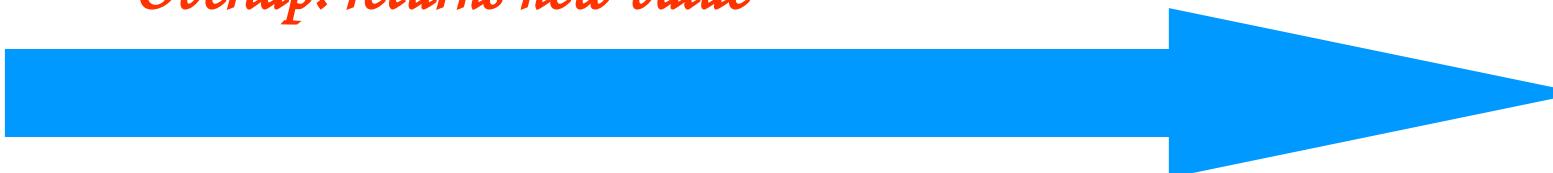
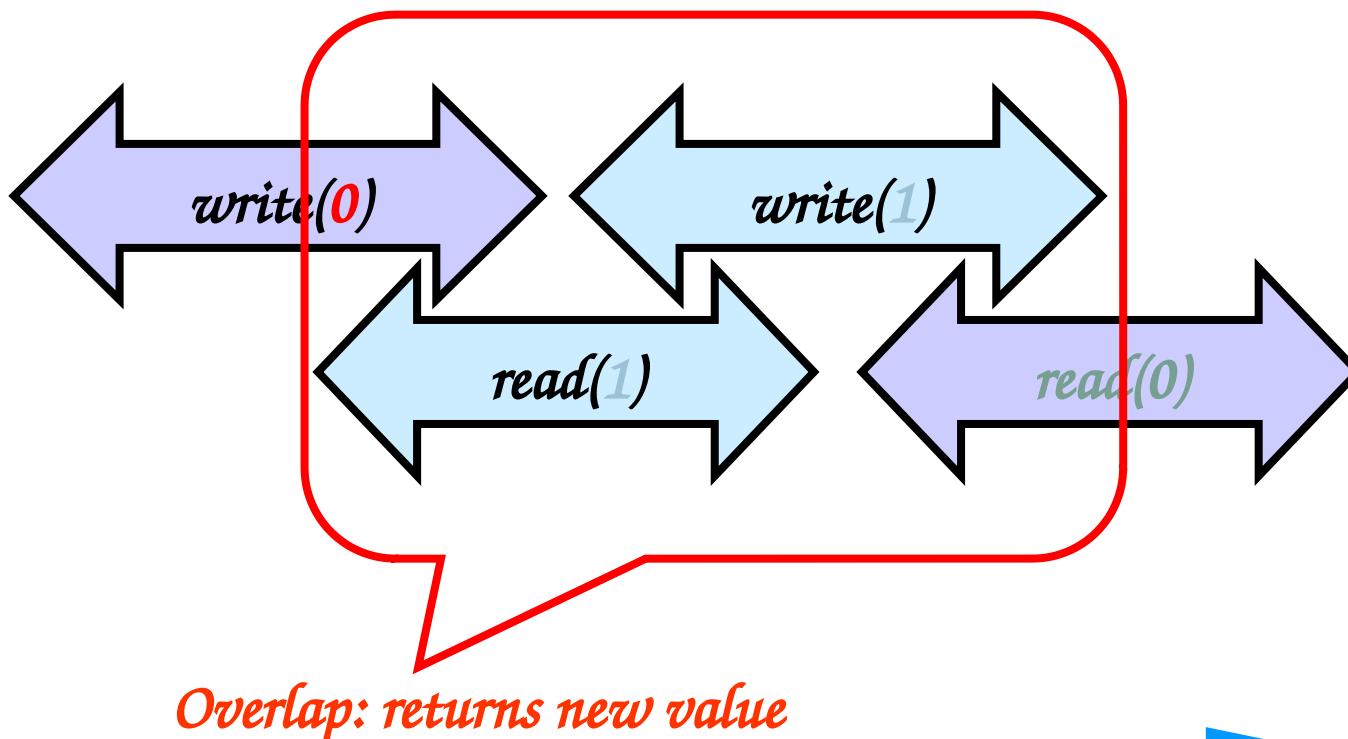
# Regular register

- A single-writer, multi-reader register is regular if:
  - A read() that does not overlap a write() returns the last value
  - If a read() overlaps a write() it returns either the old value or the new value
    - Value being read may “flicker” between the old and new value before finally changing to the new value

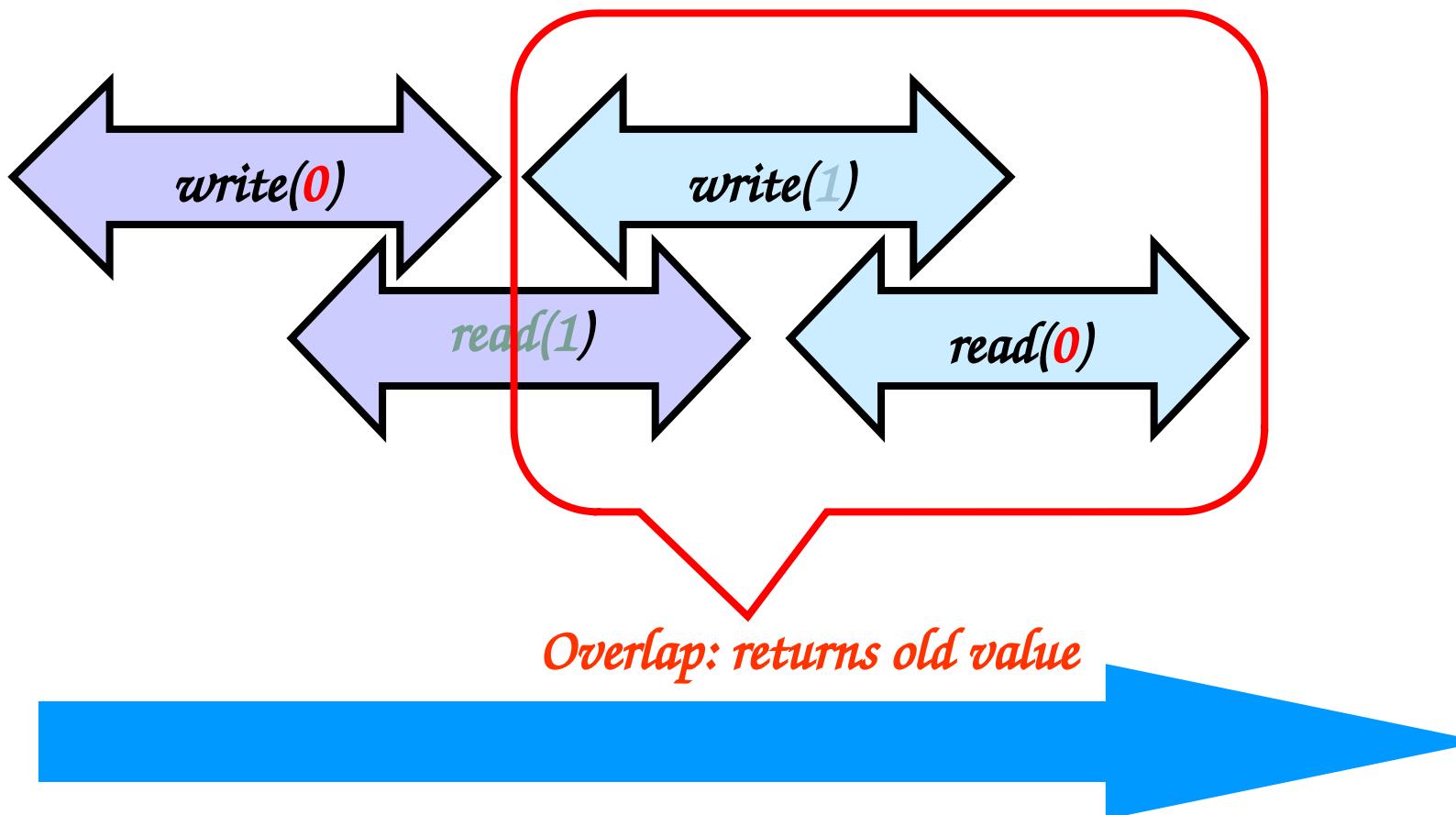
# Regular or Not?



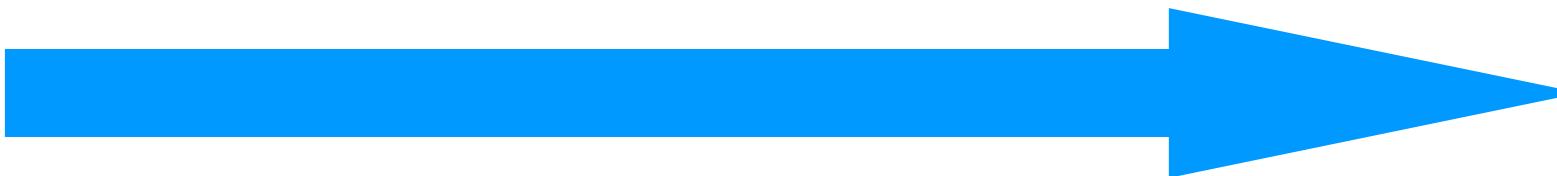
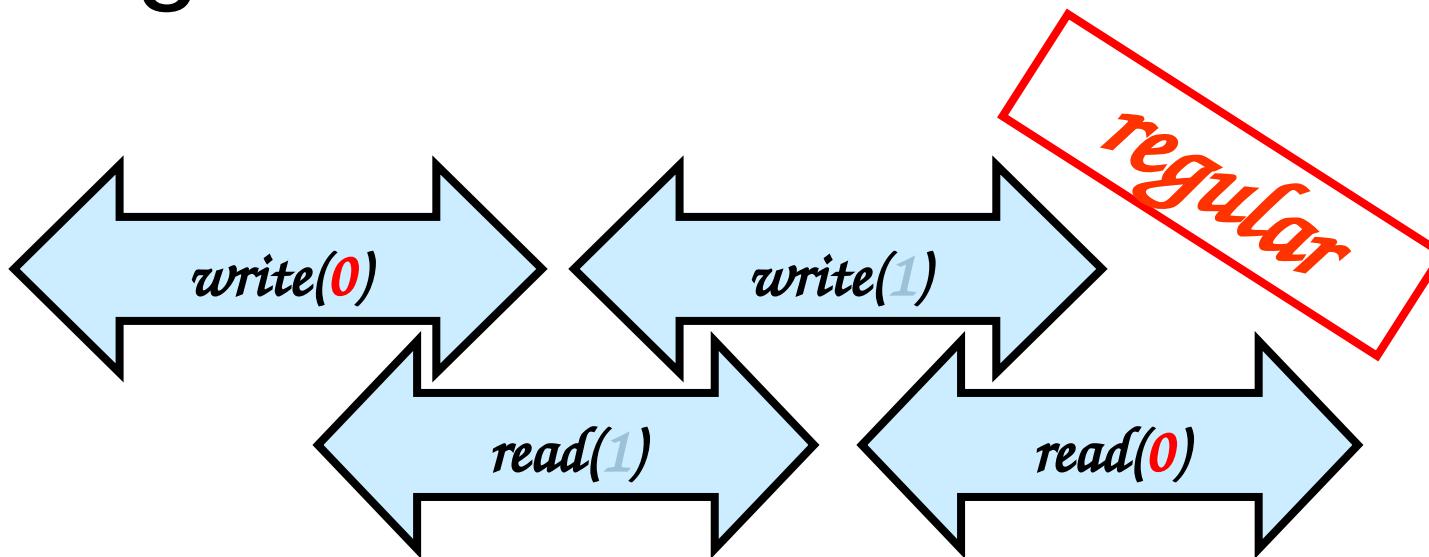
# Regular or Not?



# Regular or Not?

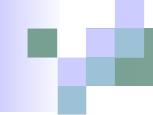


# Regular or Not?





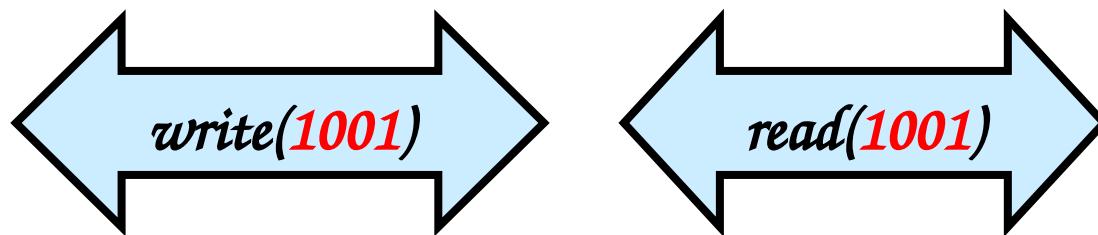
# **Regular $\neq$ Linearizable**



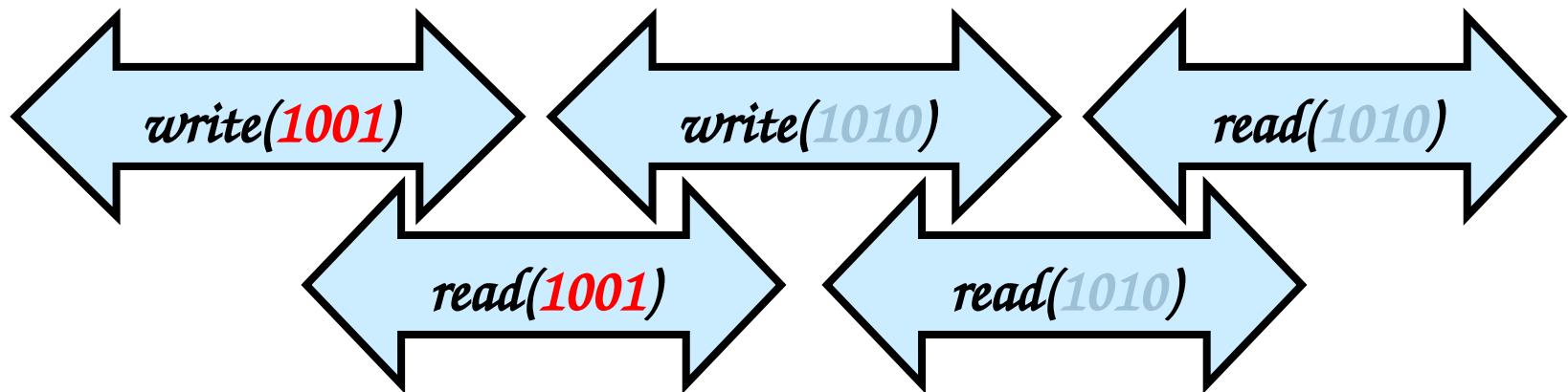
# Atomic register

- Linearizable implementation of sequential register
- A single-writer, multi-reader register is atomic if:
  - Each read() returns the last value written

# Sequential Register

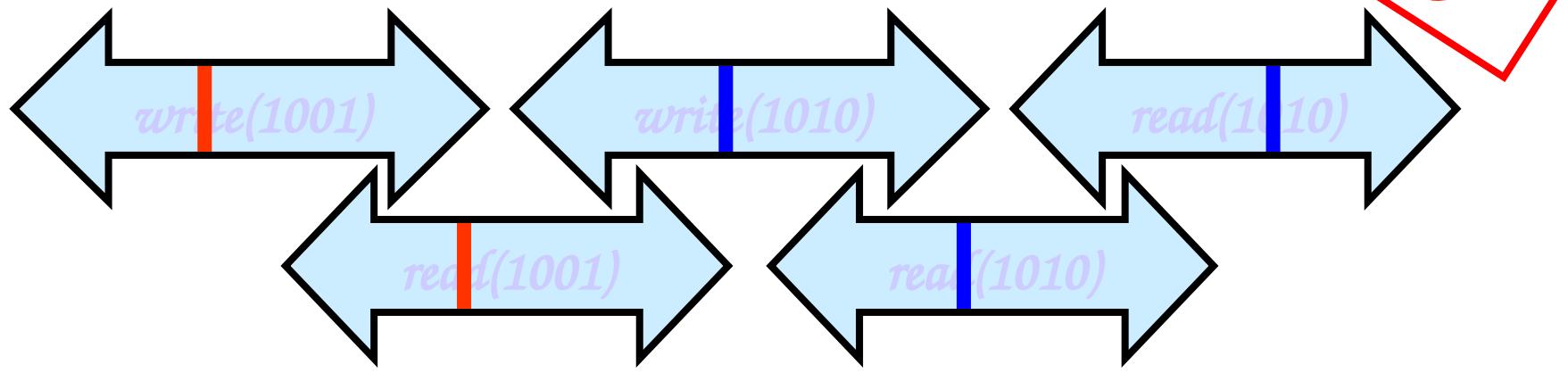


# Atomic Register

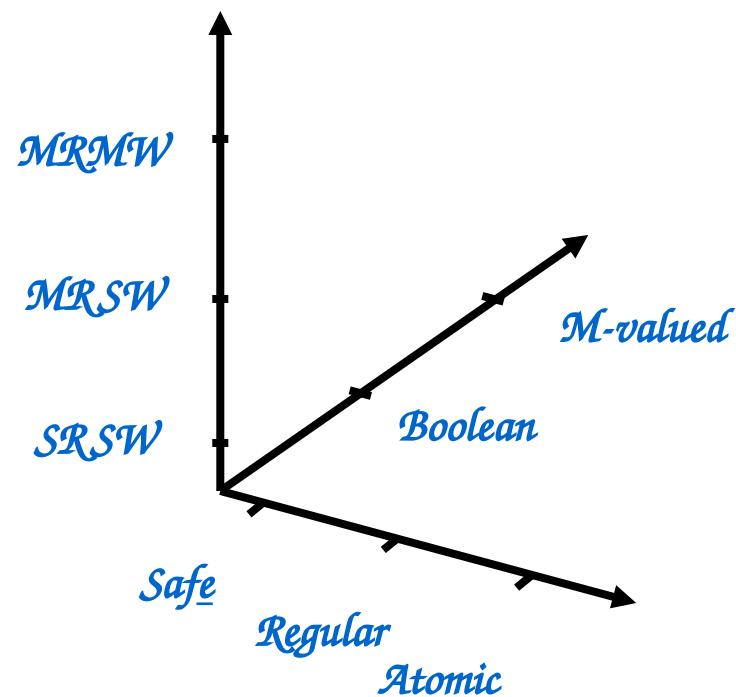


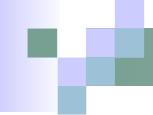
Linearizable?

# Atomic Register



# Register Space



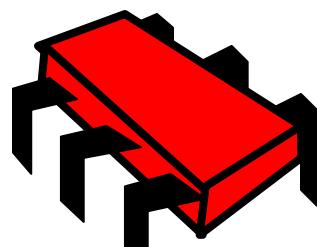


# Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic

# Weakest Register

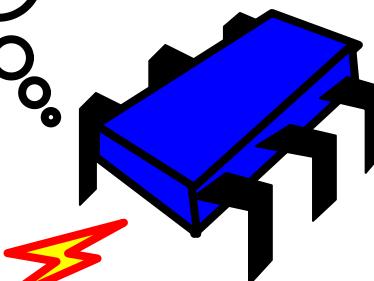
*Single writer*



*0*



*Single reader*



*1*

*Safe Boolean register*

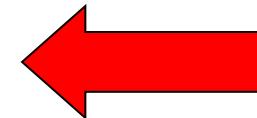


# Register construction

- We will now build a range of registers from single-reader, single-writer Boolean safe registers

# Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic



*Next*

# Register Names

```
public class SafeBoolMRSWRegister  
    implements Register<Boolean> {  
    public boolean read() { ... }  
    public void write(boolean x) { ... }  
}
```

# Register Names

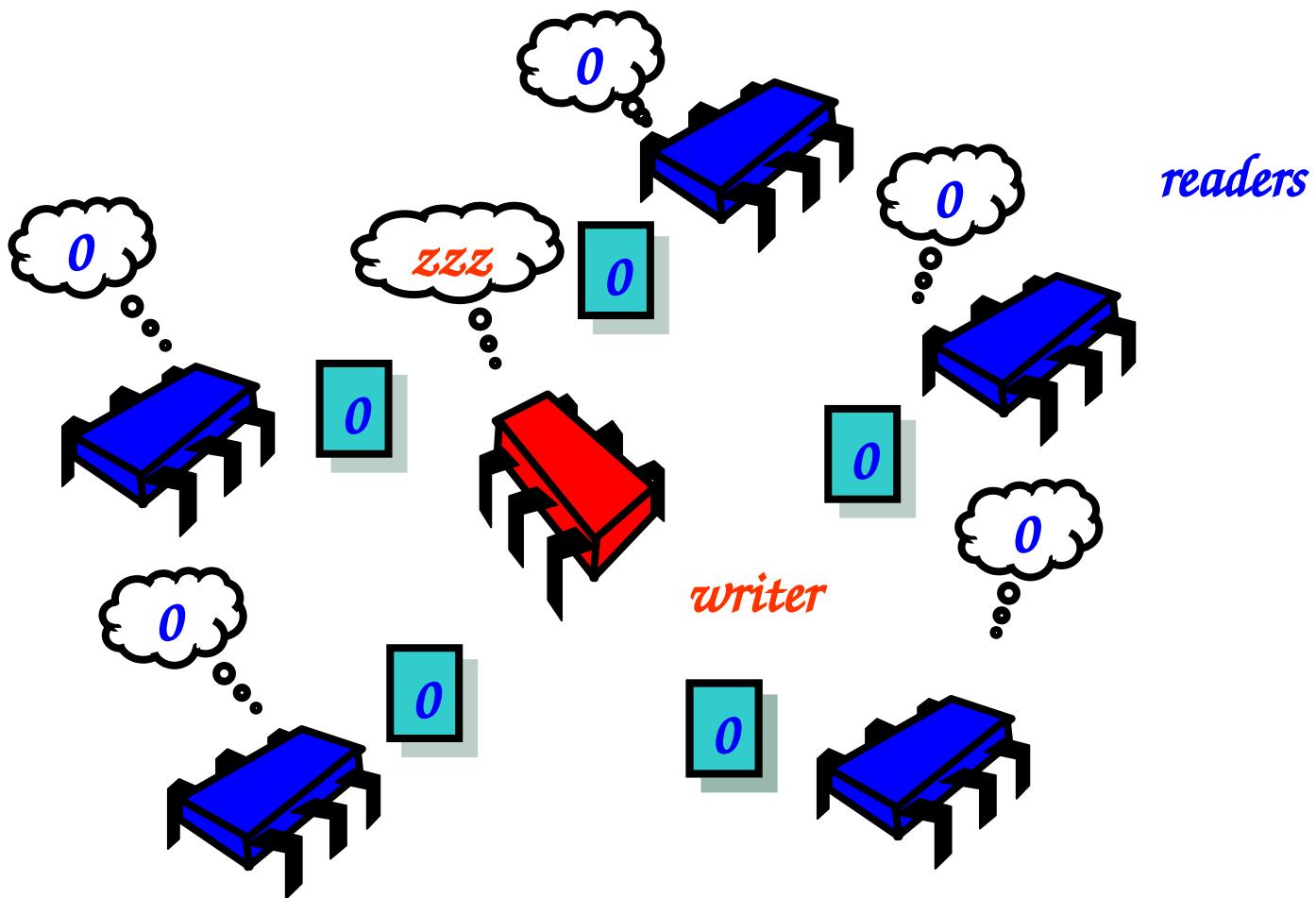
```
public class SafeBoolMRSWRegister  
    implements Register<Boolean> {  
    public boolean read() { ... }  
    public void write(boolean x) { ... }  
}
```

*property*

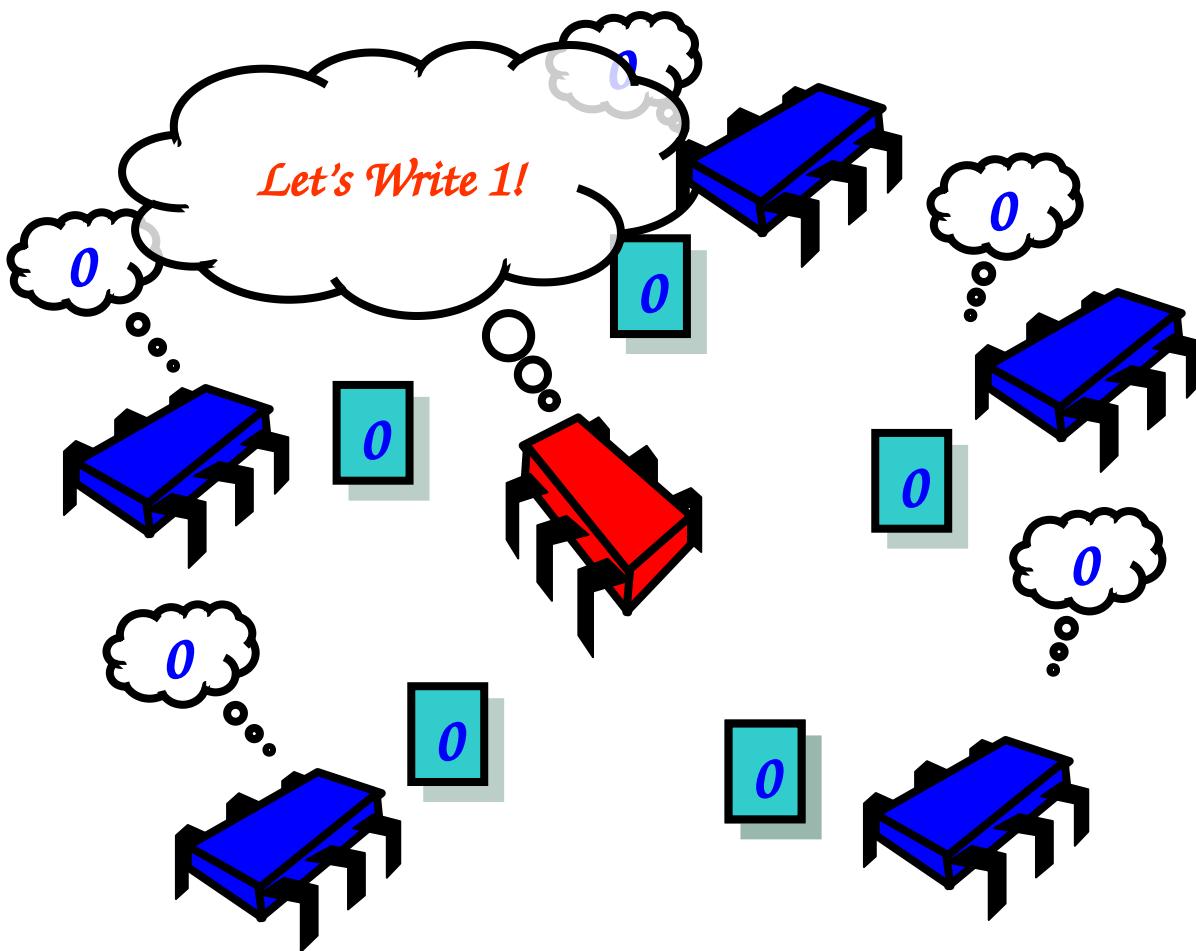
*type*

*How many readers &  
writers?*

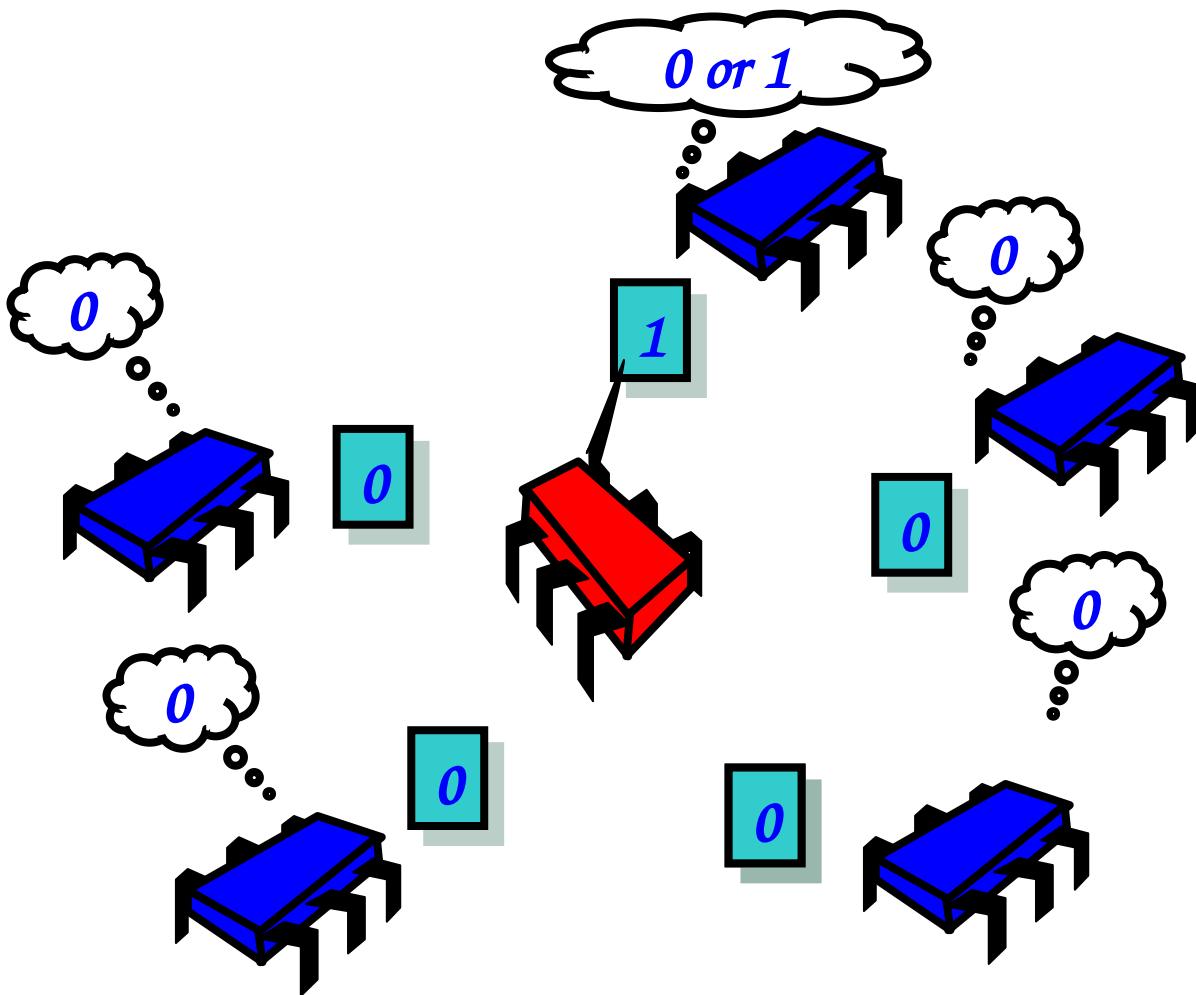
# Safe Boolean MRSW



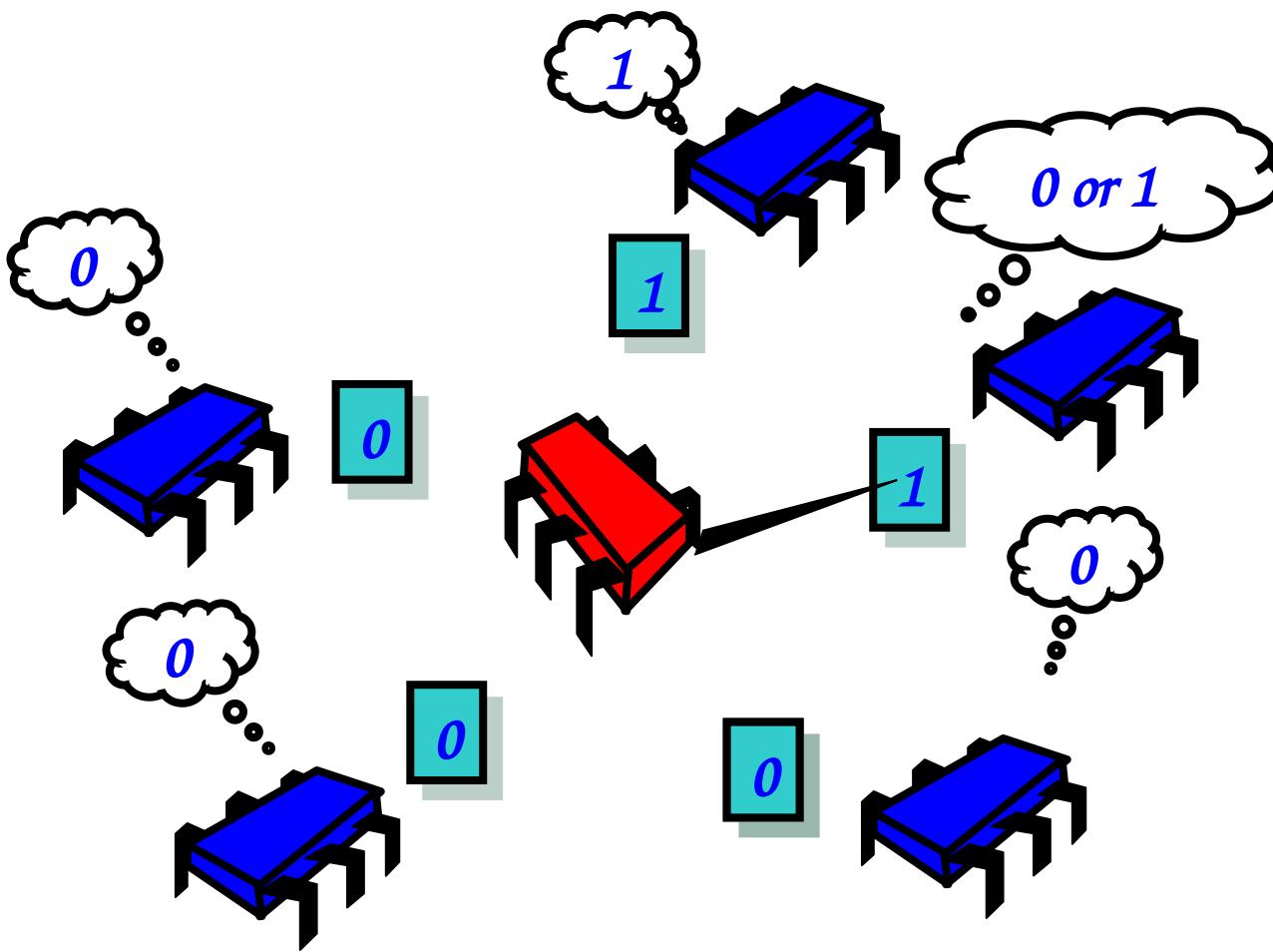
# Safe Boolean MRSW



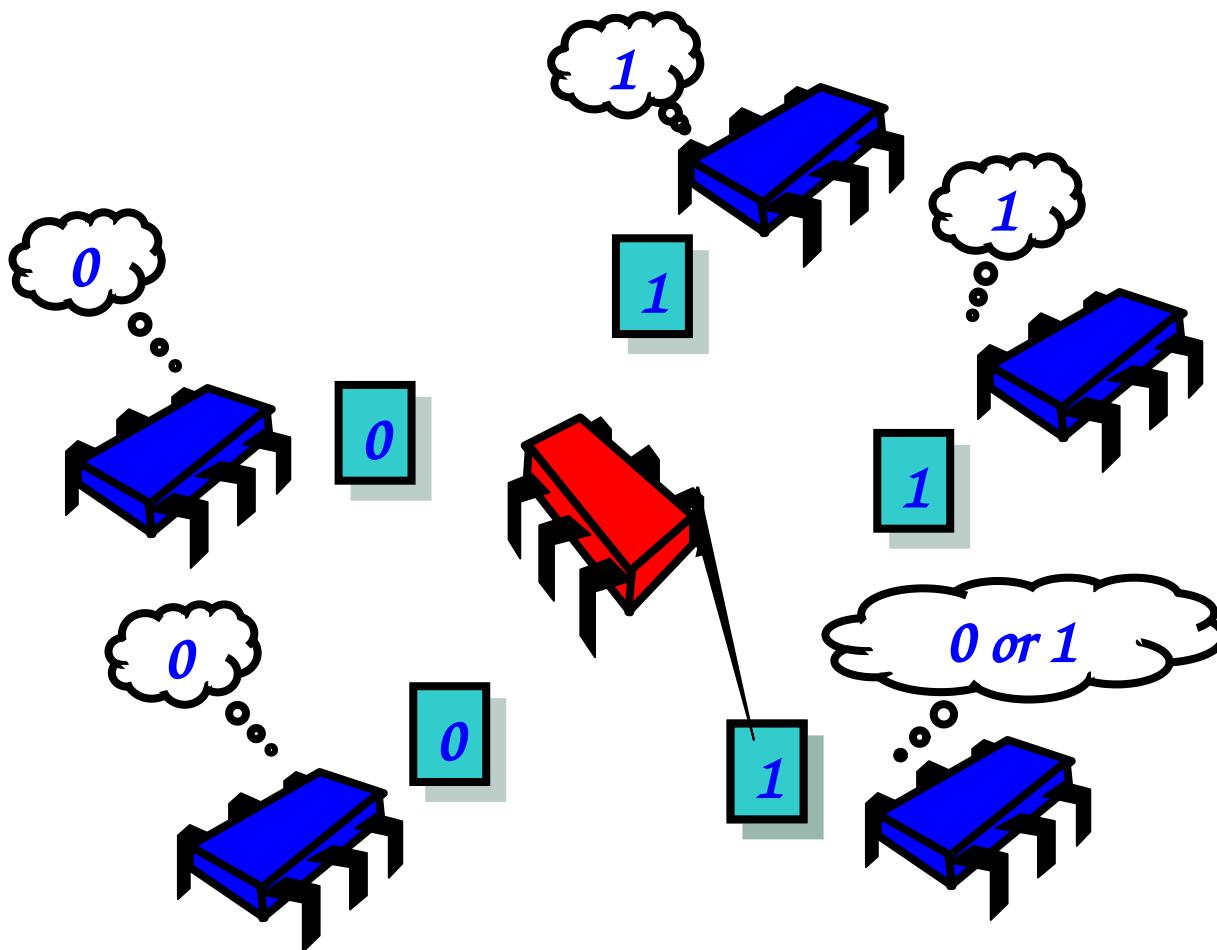
# Safe Boolean MRSW



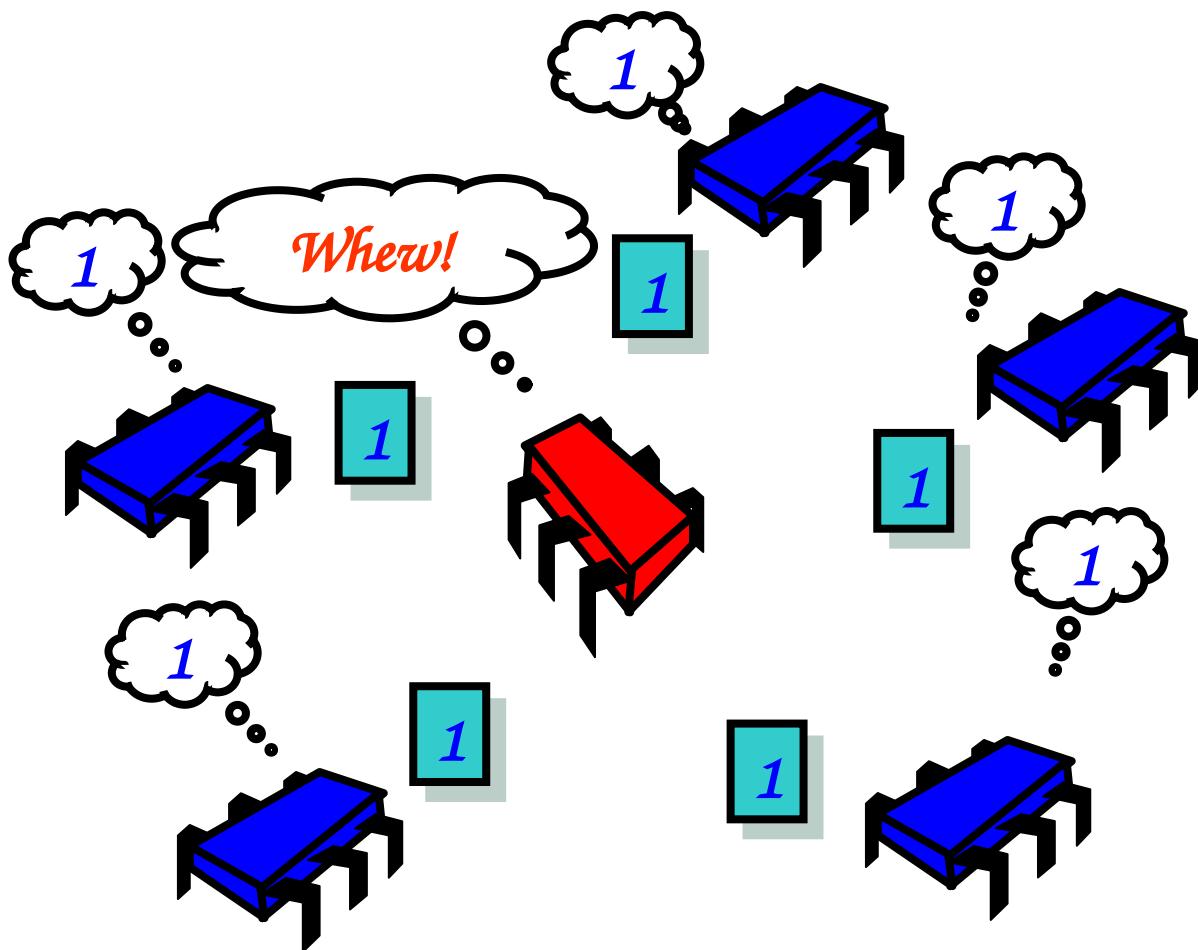
# Safe Boolean MRSW



# Safe Boolean MRSW



# Safe Boolean MRSW



# Safe Boolean MRSW

```
public class SafeBoolMRSWRegister implements  
    Register<boolean> {  
  
    boolean[] s_table; //array of SRSW registers  
  
    public SafeBoolMRSWRegister(int capacity) {  
        s_table = new boolean[capacity];  
    }  
    public boolean read() {  
        return s_table[ThreadID.get()];  
    }  
    public void write(boolean x) {  
        for (int i = 0; i < s_table.length; i++)  
            s_table[i] = x;  
    }  
}
```

*Each thread has  
own safe SRSW  
register*

# Safe Boolean MRSW

```
public class SafeBoolMRSWRegister implements  
    Register<boolean> {  
  
    boolean[] s_table; //array of SRSW registers  
  
    public SafeBoolMRSWRegister(int capacity) {  
        s_table = new boolean[capacity];  
    }  
    public boolean read() {  
        return s_table[ThreadID.get()];  
    }  
    public void write(boolean x) {  
        for (int i = 0; i < s_table.length; i++)  
            s_table[i] = x;  
    }  
}
```

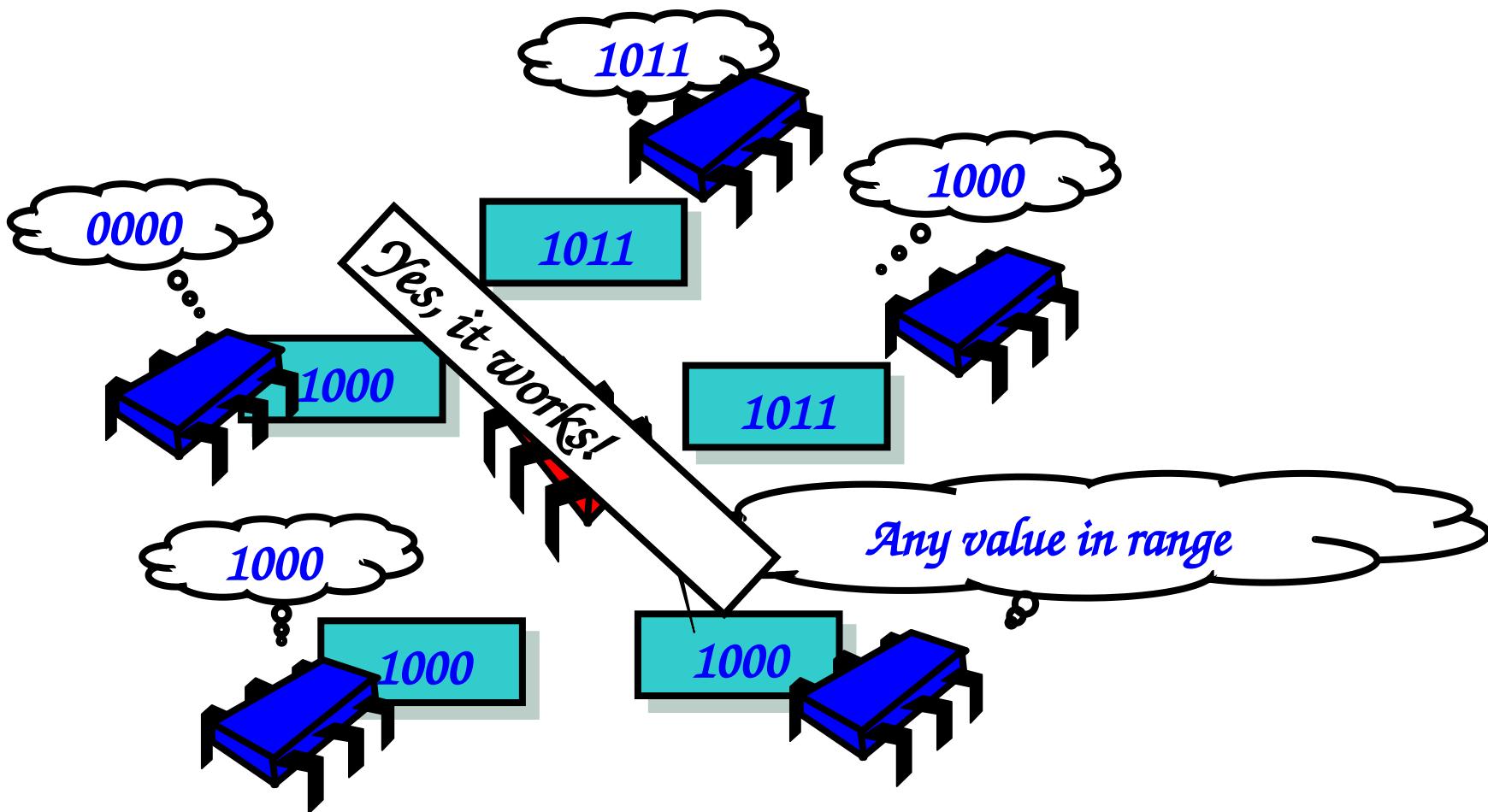
*Write each thread's register one at a time*

# Safe Boolean MRSW

```
public class SafeBoolMRSWRegister implements  
    Register<boolean> {  
  
    boolean[] s_table; //array of SRSW registers  
  
    public SafeBoolMRSWRegister(int capacity) {  
        s_table = new boolean[capacity];  
    }  
    public boolean read() {  
        return s_table[threadID.get()];  
    }  
    public void write(boolean x) {  
        for (int i = 0; i < s_table.length; i++)  
            s_table[i] = x;  
    }  
}
```

*Each thread reads  
own register*

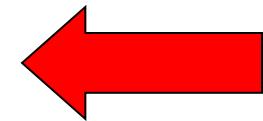
# Safe Multi-Valued MRSW?



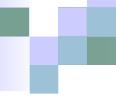


# Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic



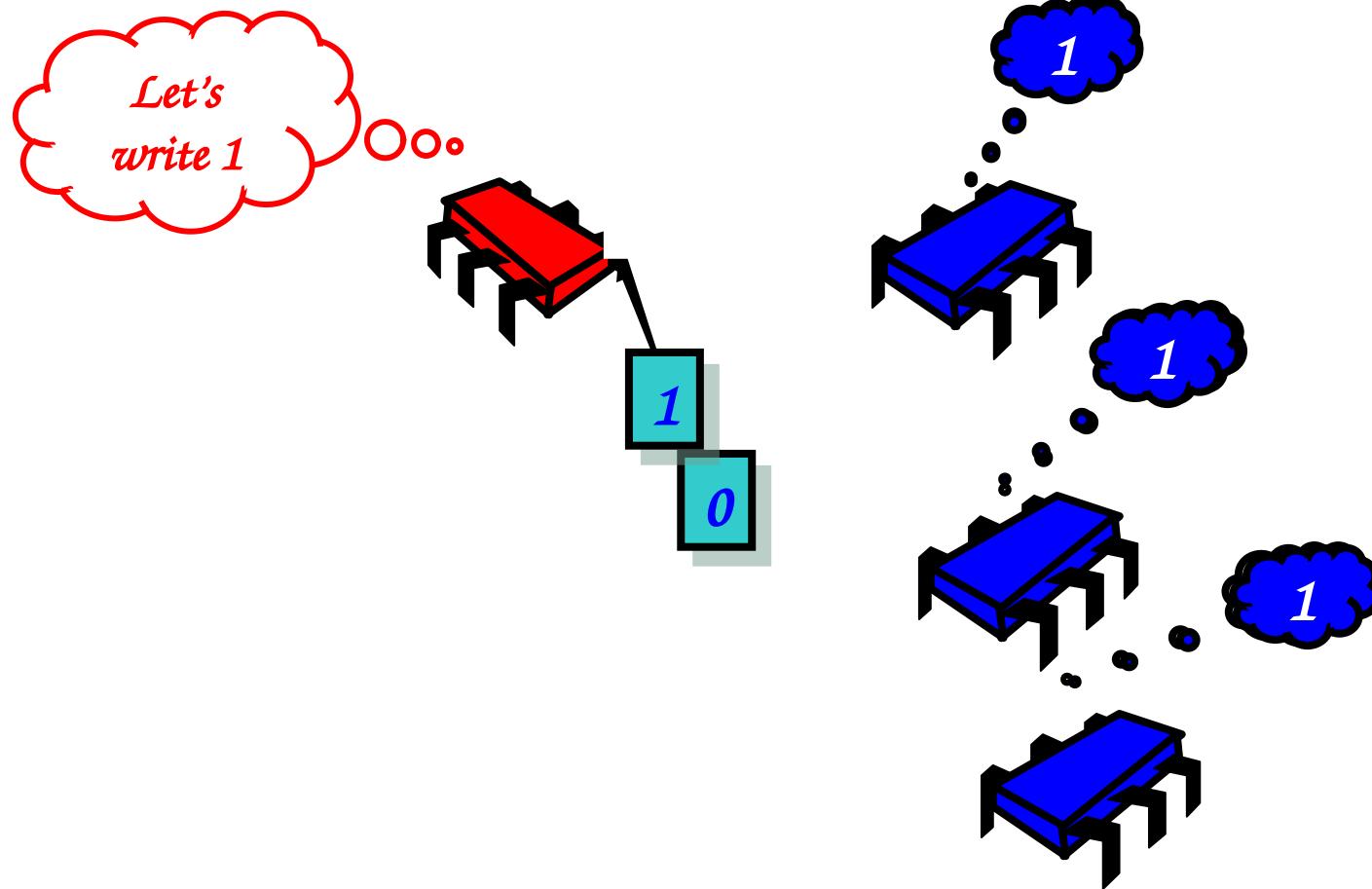
*Next*



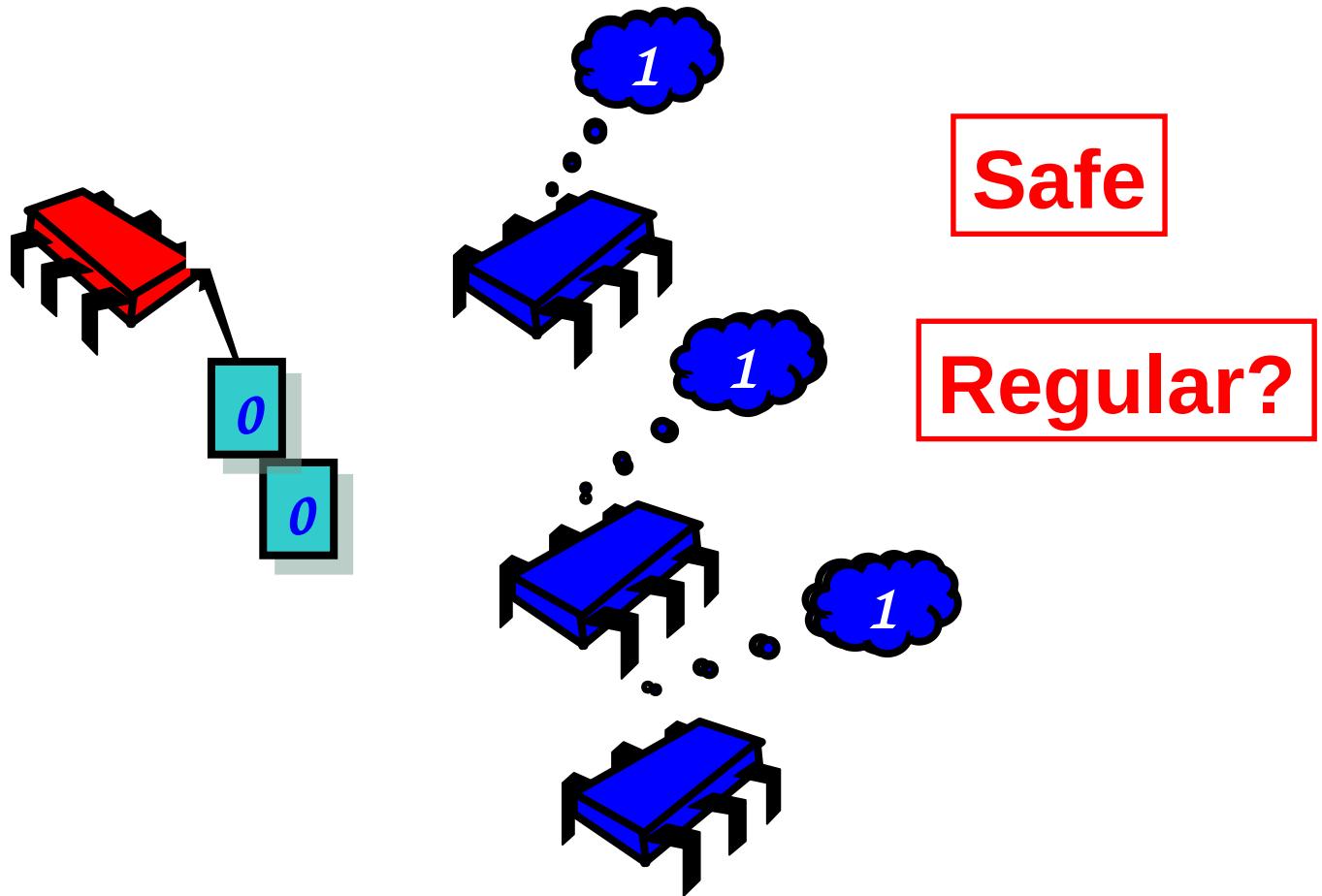
# Safe BooleanMRSW vs Regular BooleanMRSW

- Only difference is when newly written value is same as old value:
  - Safe register can return either Boolean value
  - Regular register can return either new value or old value – if both new and old is x, then regular can only return x
  - So... write value only if distinct from previous written value

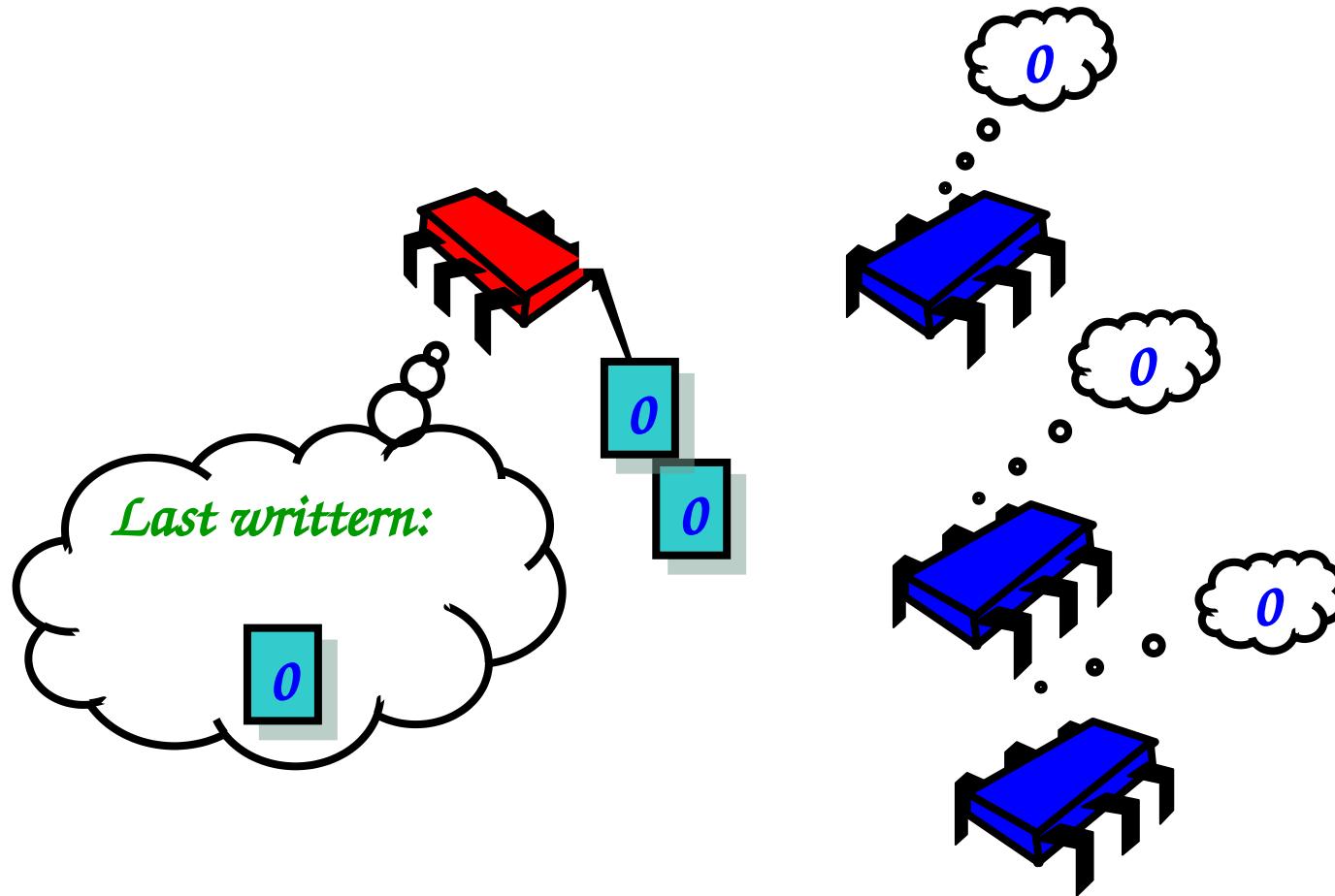
# Safe Boolean MRSW = Regular Boolean MRSW



# Safe Boolean MRSW = Regular Boolean MRSW



# Safe Boolean MRSW = Regular Boolean MRSW



# Safe Boolean MRSW $\sqsubseteq$ Regular Boolean MRSW

```
public class RegBoolMRSWRegister
    implements Register<Boolean> {
    private boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

# Safe Boolean MRSW □ Regular Boolean MRSW

```
public class RegBoolMRSWRegister
    implements Register<Boolean> {
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

*Last bit this thread wrote  
(made-up syntax)*

# Safe Boolean MRSW □ Regular Boolean MRSW

```
public class RegBoolMRSWRegister
    implements Register<Boolean> {
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

*Actual value*

# Safe Boolean MRSW $\sqsubseteq$ Regular Boolean MRSW

```
public class RegBoolMRSWRegister
    implements Register<Boolean> {
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

*Is new value different from last  
value I wrote?*

# Safe Boolean MRSW $\sqsubseteq$ Regular Boolean MRSW

```
public class RegBoolMRSWRegister
    implements Register<Boolean> {
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) {
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

*If so, change it  
(otherwise don't!)*

# Safe Boolean MRSW $\sqsubseteq$ Regular Boolean MRSW

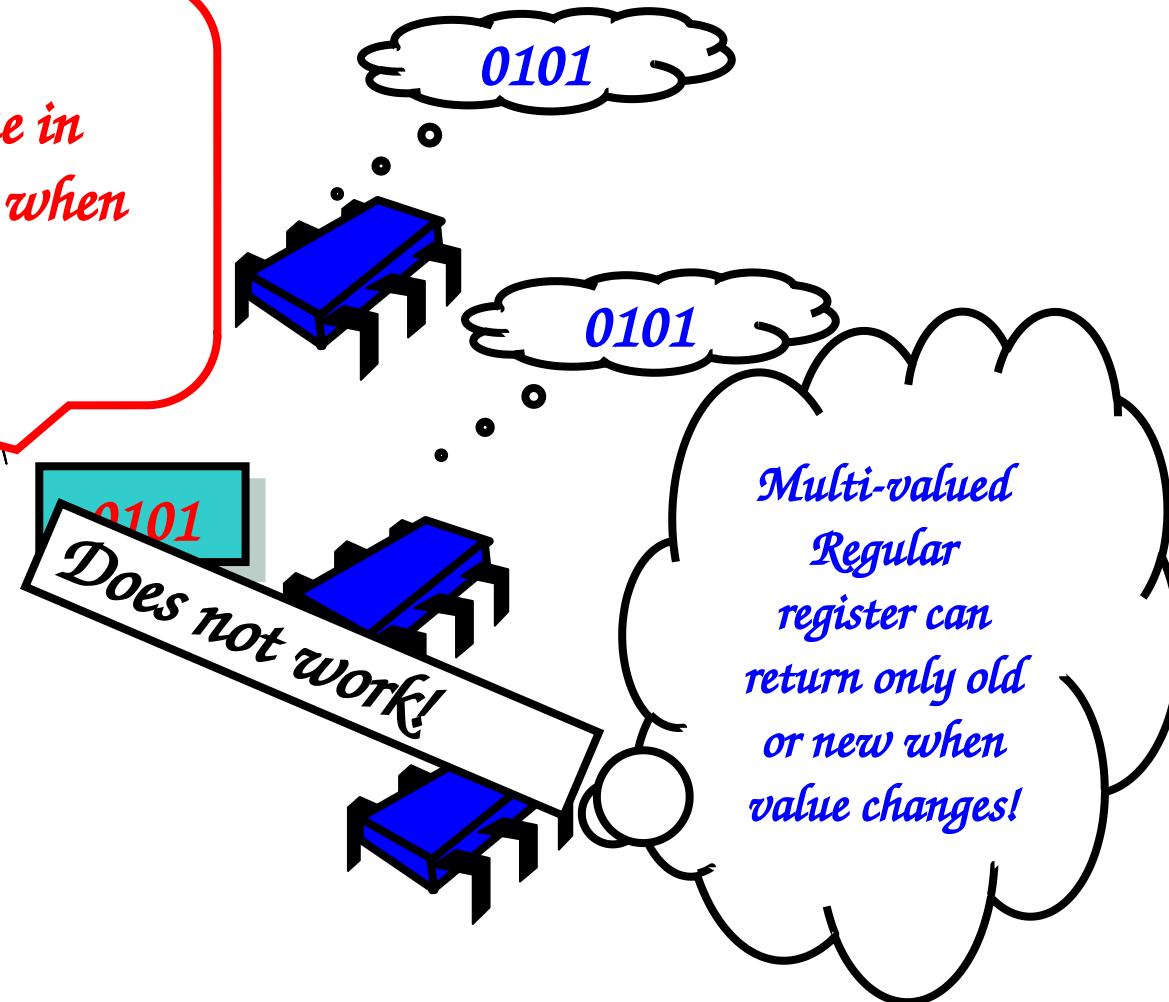
```
public class RegBoolMRSWRegister
    implements Register<Boolean>{
    threadLocal boolean old;
    private SafeBoolMRSWRegister value;
    public void write(boolean x) {
        if (old != x) { • Overlap? No Overlap?
            value.write(x);
            old = x;
        }
    }
    public boolean read() {
        return value.read();
    }
}
```

• Overlap? No Overlap?  
No problem  
either Boolean value works

# Safe Multi-Valued MRSW □

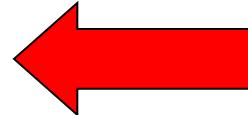
## Regular Multi-Valued MRSW

*Safe register can return value in range other than old or new when value changes*



# Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic



*Next*

# Regular M-Valued MRSW Register

- Values are represented using unary notation
- An M-valued register is implemented as an array of  $m$  regular MRSW Boolean registers
- Initially the register is set to 0

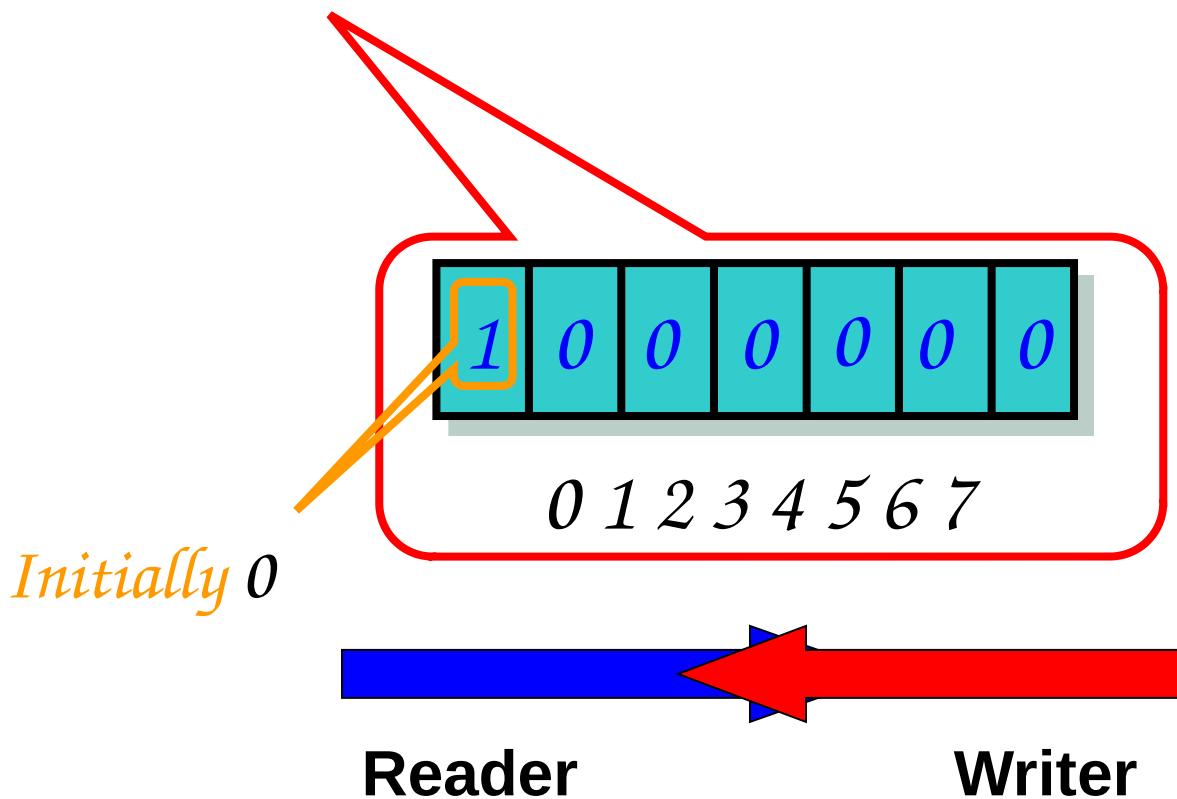
# Regular M-Valued MRSW Register

- write():
  - A write() of value x, writes true to location x – which is a Regular Boolean MRSW register
  - It then sets all the lower locations to false
- read():
  - Reads the locations from lower to higher values until it reaches a value that is true

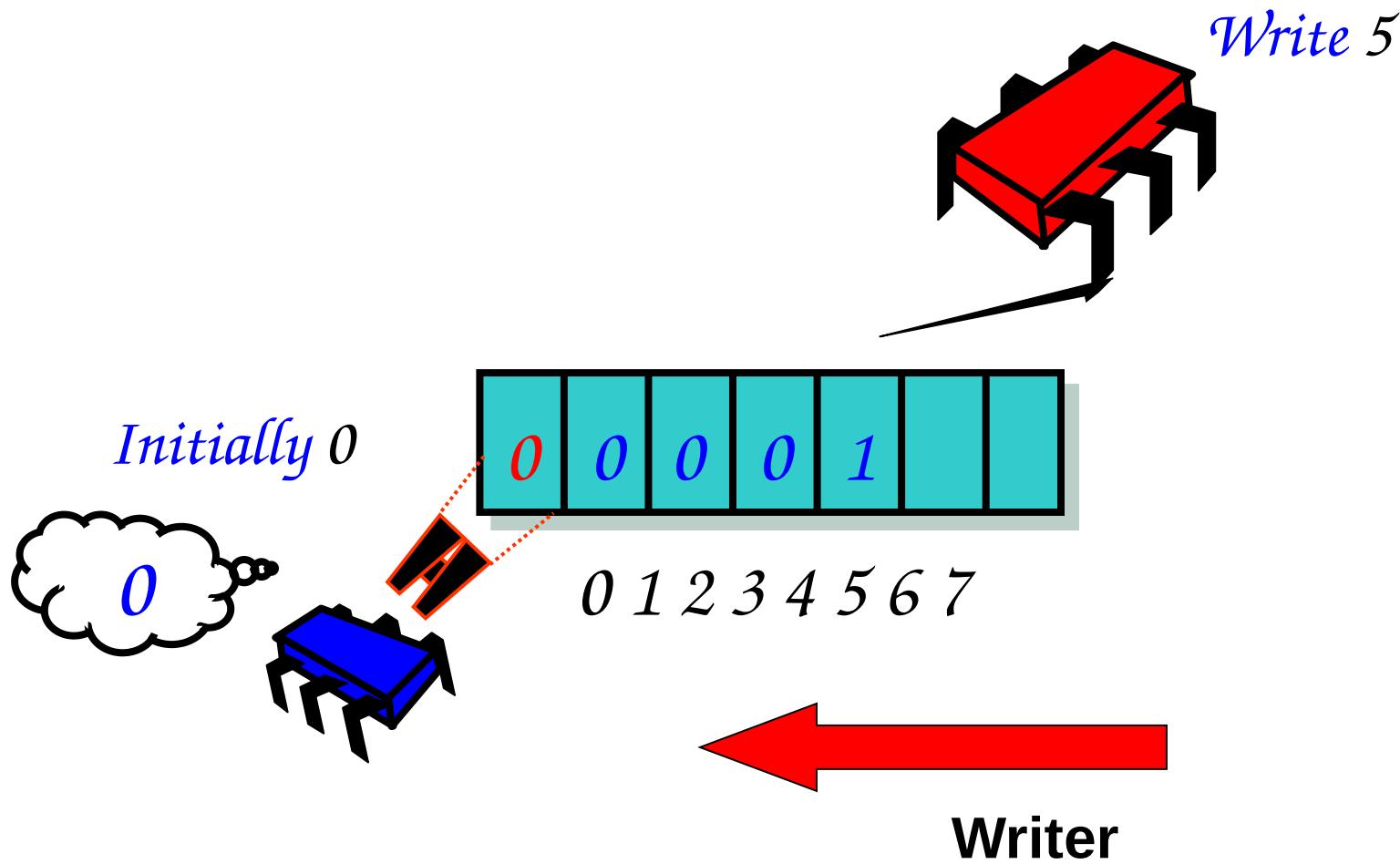
# Writing M-Valued

*Unary representation: bit[i]*

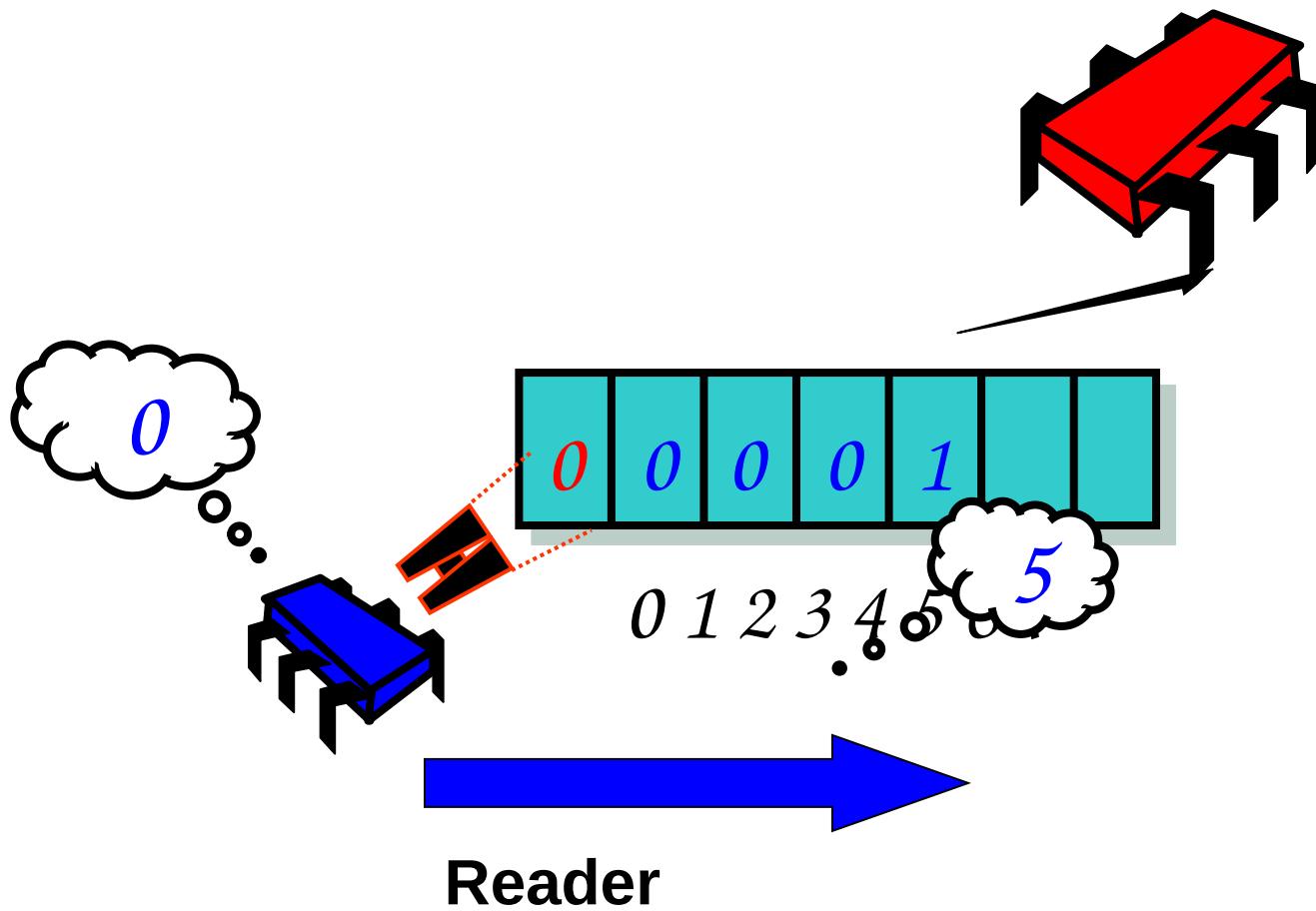
*means value i*



# Writing M-Valued



# Writing M-Valued



# MRSW Regular Boolean $\sqsubseteq$ MRSW Regular M-valued

```
public class RegMRSWRegister implements Register{
    RegBoolMRSWRegister[M] bit;

    public void write(int x) {
        this.bit[x].write(true);
        for (int i=x-1; i>=0; i--)
            this.bit[i].write(false);
    }

    public int read() {
        for (int i=0; i < M; i++)
            if (this.bit[i].read())
                return i;
    }
}
```

# MRSW Regular Boolean $\equiv$ MRSW Regular M-valued

```
public class RegMRSWRegister implements Register{  
  
    RegBoolMRSWRegister[M] bit;  
  
    public void write(int x) {  
        this.bit[x].write(true);  
        for (int i=x-1; i>=0; i--)  
            this.bit[i].write(false);  
    }  
  
    public int read() {  
        for (int i=0; i < M; i++)  
            if (this.bit[i].read())  
                return i;  
    }  
}
```

*Unary representation: bit[i]  
means value i*

# MRSW Regular Boolean $\equiv$ MRSW Regular M-valued

```
public class RegMRSWRegister implements Register {  
  
    RegBoolMRSWRegister[m] bit;  
  
    public void write(int x) {  
        this.bit[x].write(true);  
        for (int i=x-1; i>=0; i--)  
            this.bit[i].write(false);  
    }  
  
    public int read() {  
        for (int i=0; i < M; i++)  
            if (this.bit[i].read())  
                return i;  
    }  
}
```

*Set bit  $x$*

# MRSW Regular Boolean $\equiv$ MRSW Regular M-valued

```
public class RegMRSWRegister implements Register {  
  
    RegBoolMRSWRegister[m] bit;  
  
    public void write(int x) {  
        this.bit[x].write(true);  
        for (int i=x-1; i>=0; i--)  
            this.bit[i].write(false);  
    }  
  
    public int read() {  
        for (int i=0; i < M; i++)  
            if (this.bit[i].read())  
                return i;  
    }  
}
```

*Clear bits from  
higher to lower*

# MRSW Regular Boolean $\equiv$ MRSW Regular M-valued

```
public class RegMRSWRegister implements Register {  
  
    RegBoolMRSWRegister[m] bit;  
  
    public void write(int x) {  
        this.bit[x].write(true);  
        for (int i=x-1; i>=0; i--)  
            this.bit[i].write(false);  
    }  
  
    public int read() {  
        for (int i=0; i < M; i++)  
            if (this.bit[i].read())  
                return i;  
    }  
}
```

*Scan from lower to  
higher & return first bit  
set*

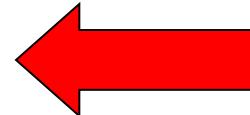


# Regular Register Conditions

- Further conditions for a register to be regular:
  - No read() call should return a value from the future
  - No read() call should return a value from the distant past – only the most recently written non-overlapping value must be returned

# Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic



*Next*

# Road Map (Slight Detour)

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic  **SRSW Atomic**
- MRMW atomic



# Atomic Register Conditions

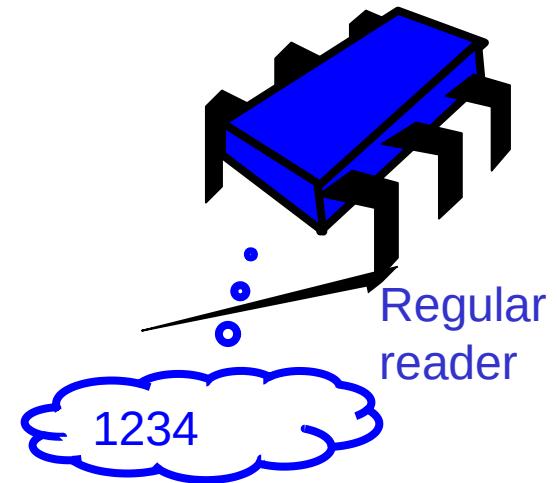
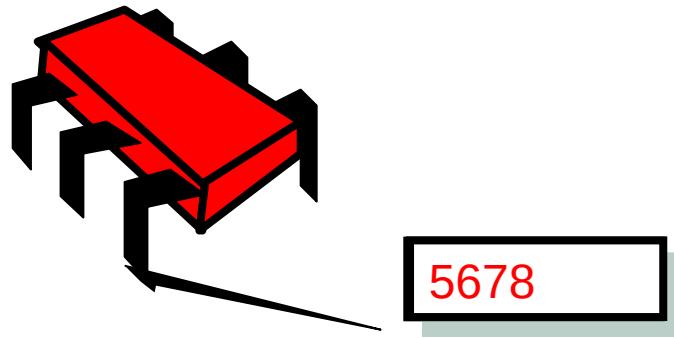
- Together with the conditions for a regular register, an additional condition for an atomic register is:
  - An earlier `read()` cannot return a value later than that returned by a later `read()`
  - In other words, values `read()` should be in the correct order

# SRSW register

- Since a SRSW register has no concurrent reads, the only way that the condition for an atomic register can be violated is when two reads that overlap the same write read values out of order

# SRSW Regular

Regular writer



Regular  
reader

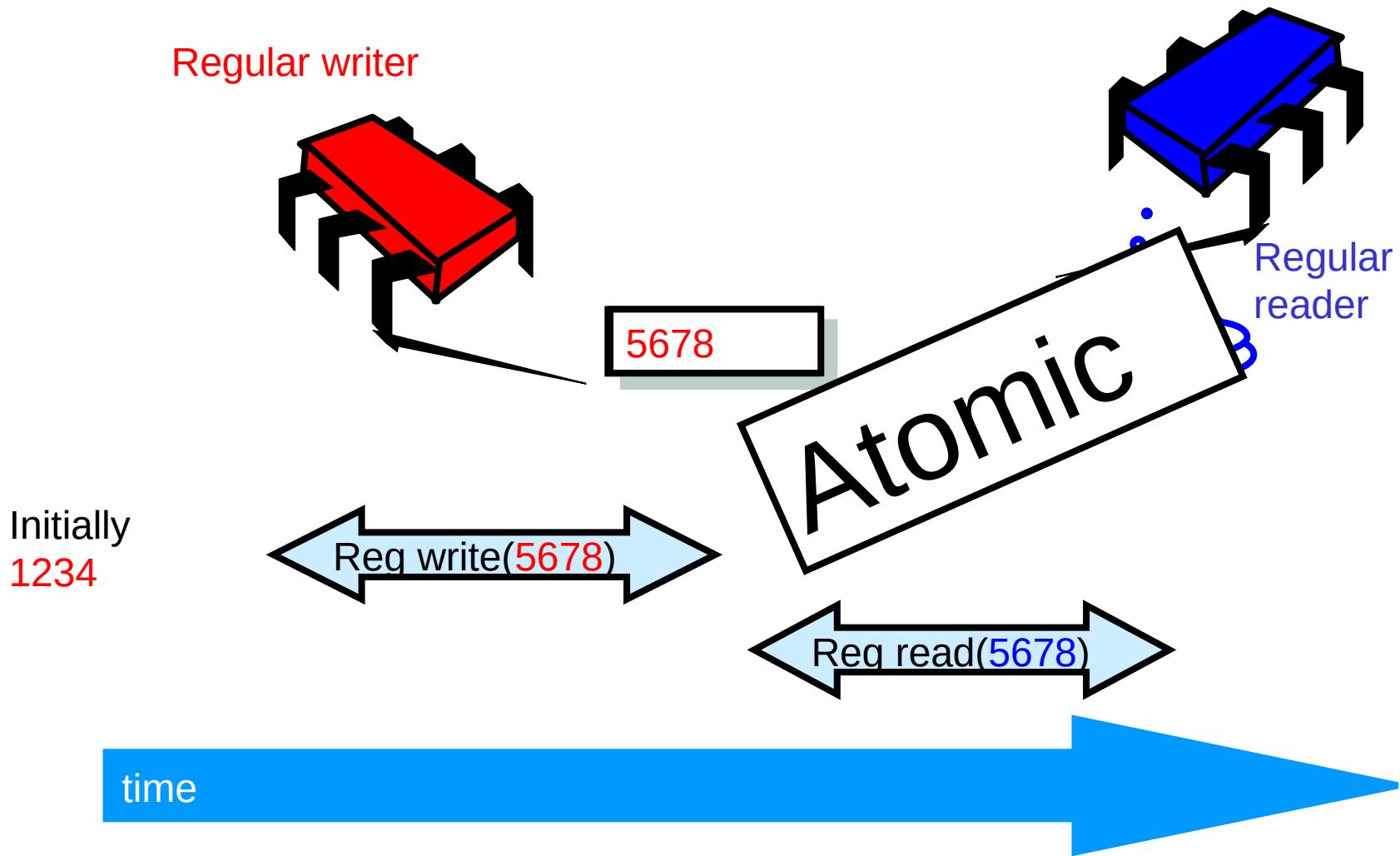
1234

Instead of 5678...

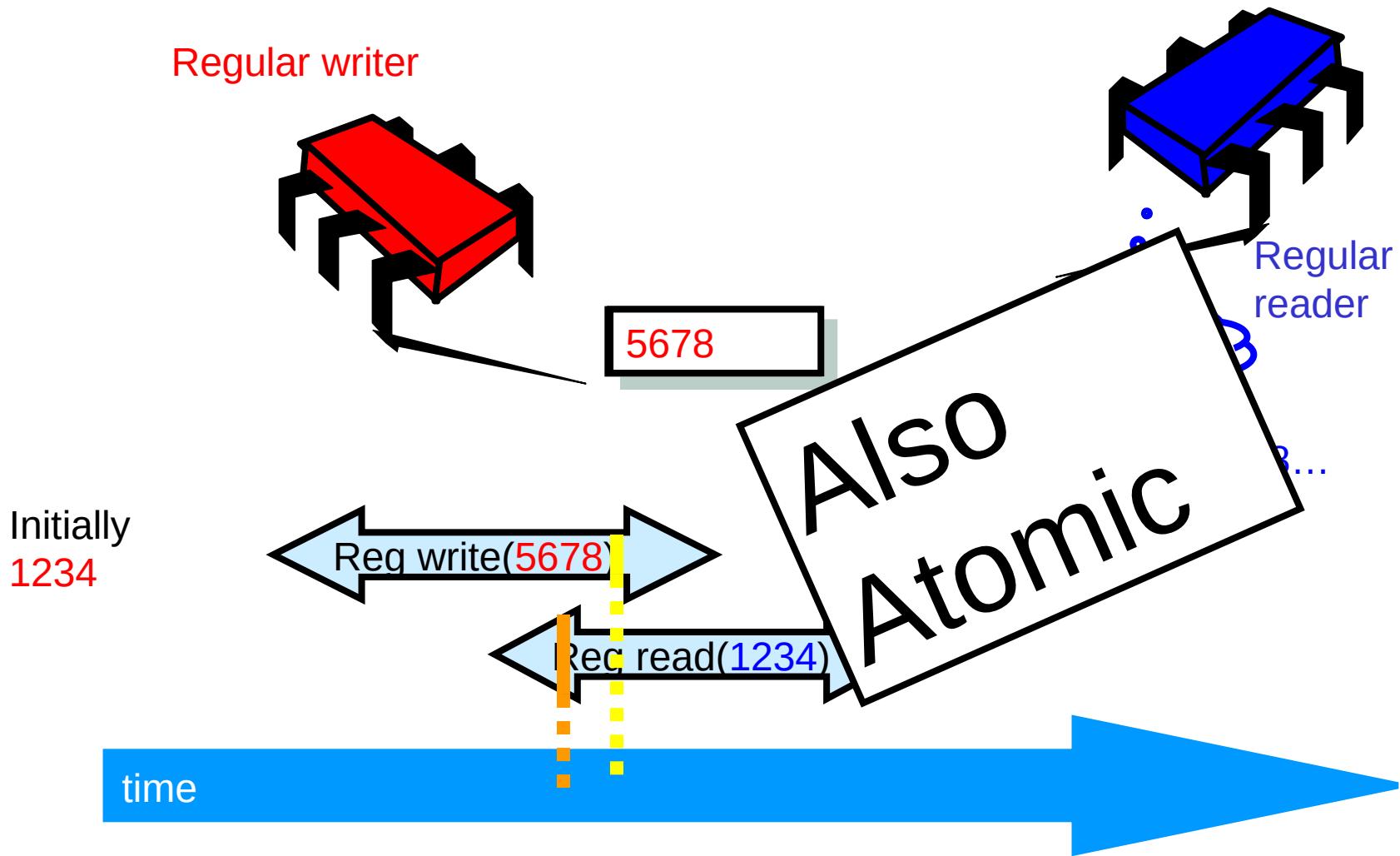
Concurrent  
Reading

When could this happen?

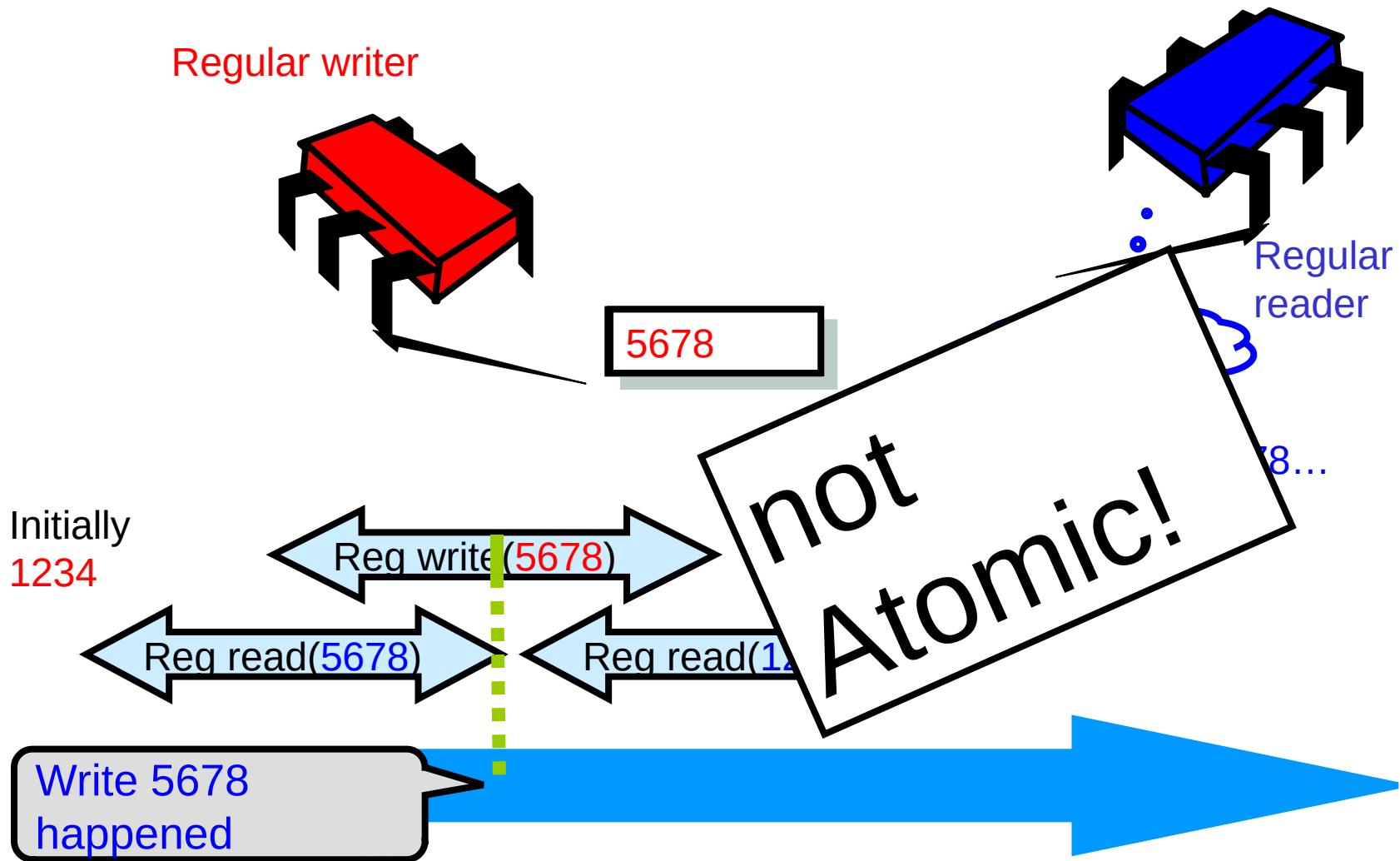
# SRSW Regular

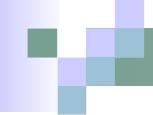


# SRSW Regular



# SRSW Regular





# Timestamps

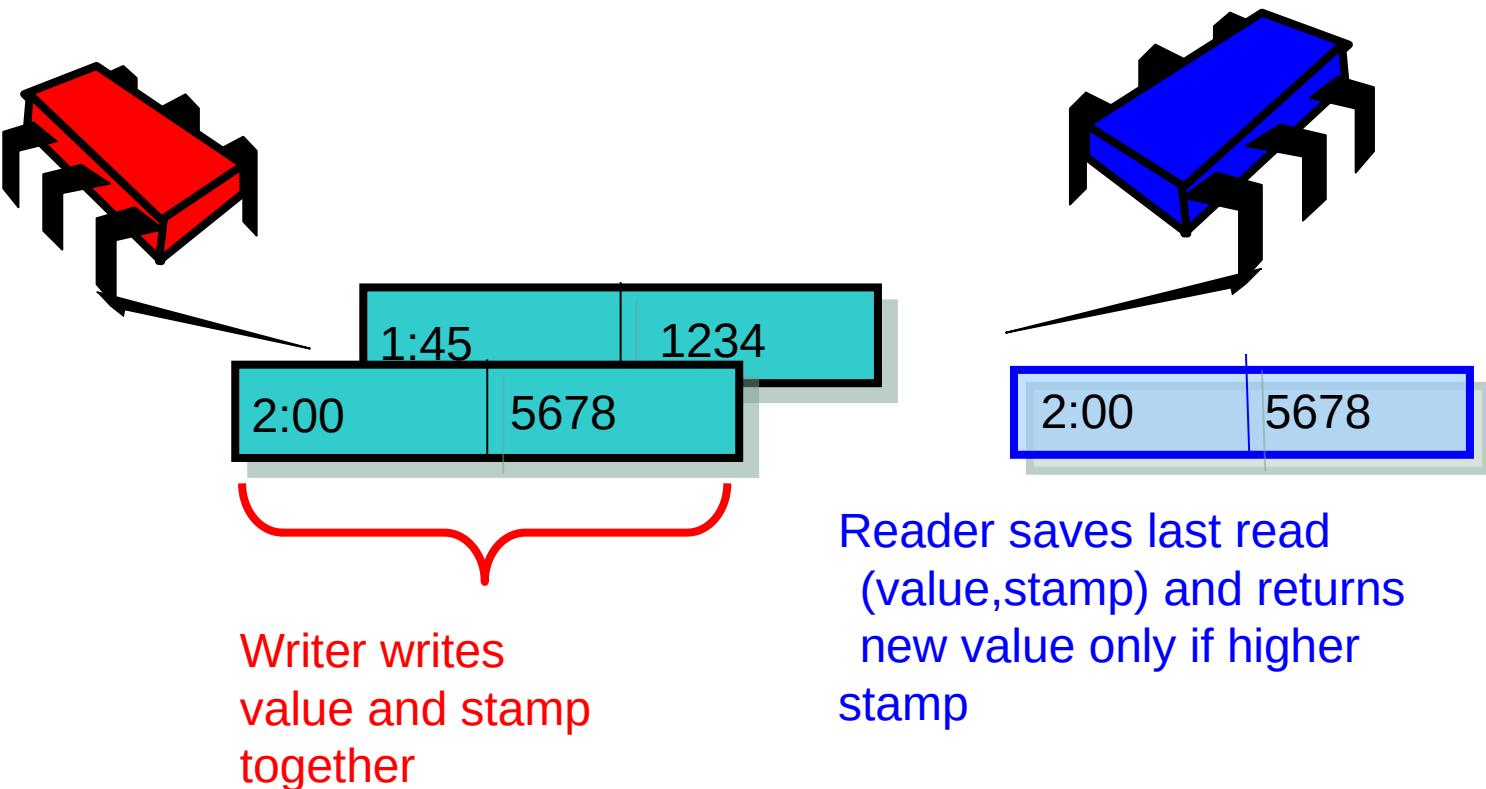
- Solution is to for each value to have an added tag – a timestamp
- Timestamps are used to order concurrent calls



# Timestamps

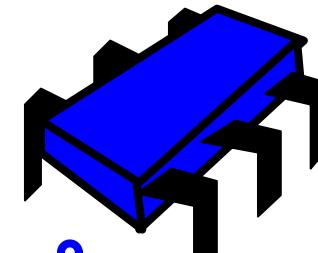
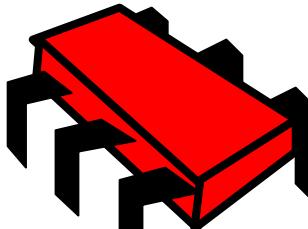
- The writer writes a timestamp to a value
- Each reader remembers the latest timestamp/value pair ever read
- If a later read() then returns an earlier value the value is discarded and the reader uses the last value

# Timestamped Values



# SRSW Regular $\sqsubseteq$ SRSW Atomic

writer



reader

Same as  
Atomic

1:45  
1234

Reg write(2:00 5678)

read(2:00 5678)

read(1:45 1234)

time

old = 2:00

5678

# Atomic SRSW

```
public class StampedValue<T> {
    public long stamp;
    public T value;
    public StampedValue (T init) {
        stamp = 0;
        value = init;
    }
    public StampedValue max (StampedValue x,StampedValue y)
    {
        if (x.stamp > y.stamp)
            return x;
        else return y;
    }
}
```

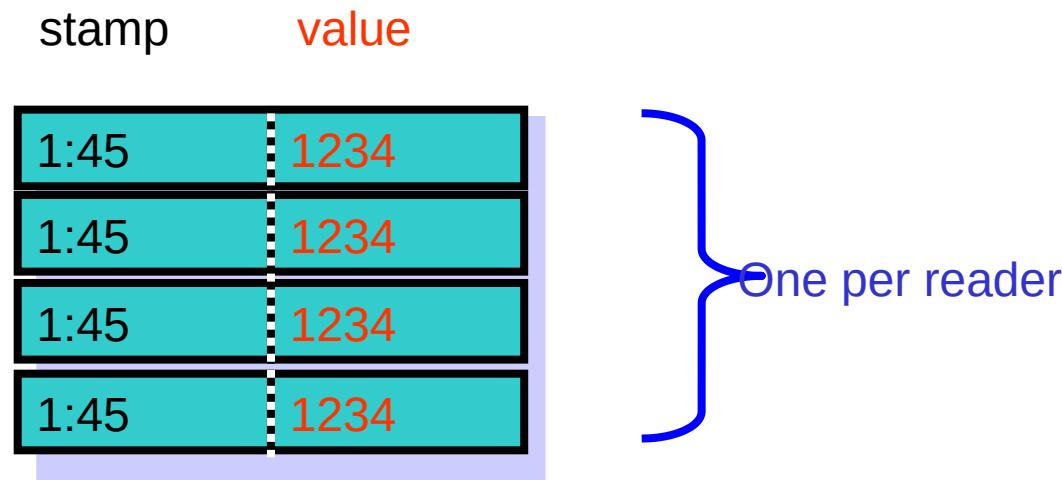
# Atomic SRSW

```
public class AtomicSRSWRegister<T> implements Register<T> {  
    long lastStamp;  
    StampedValue<T> lastRead;  
    StampedValue<T> value;  
    public T read() {  
        StampedValue<T> result = StampedValue.max(value,  
lastRead);  
        lastRead = result;  
        return result.value;  
    }  
    public void write(T v) {  
        long stamp = lastStamp + 1;  
        value = new StampedValue(stamp, v);  
        lastStamp = stamp;  
    }  
}
```

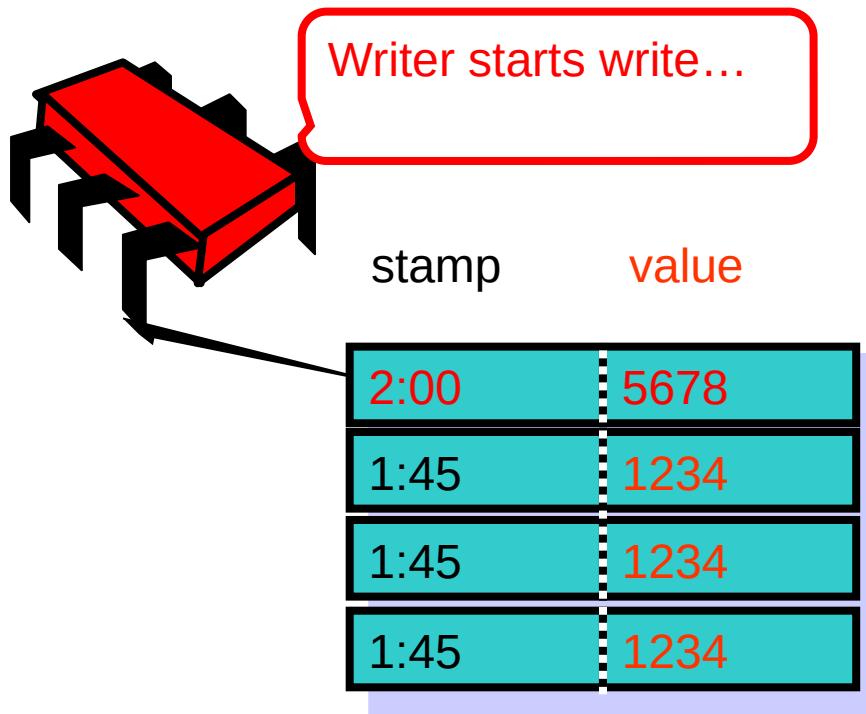
# Atomic SRSW $\sqsubseteq$ Atomic MRSW

- Can the atomic SRSW be used to built an atomic MRSW?
- Solution of Safe MRSW Registers:
  - Every thread in array
  - Write starts at the beginning of the array and iterates through array
  - Read reads only its own array location

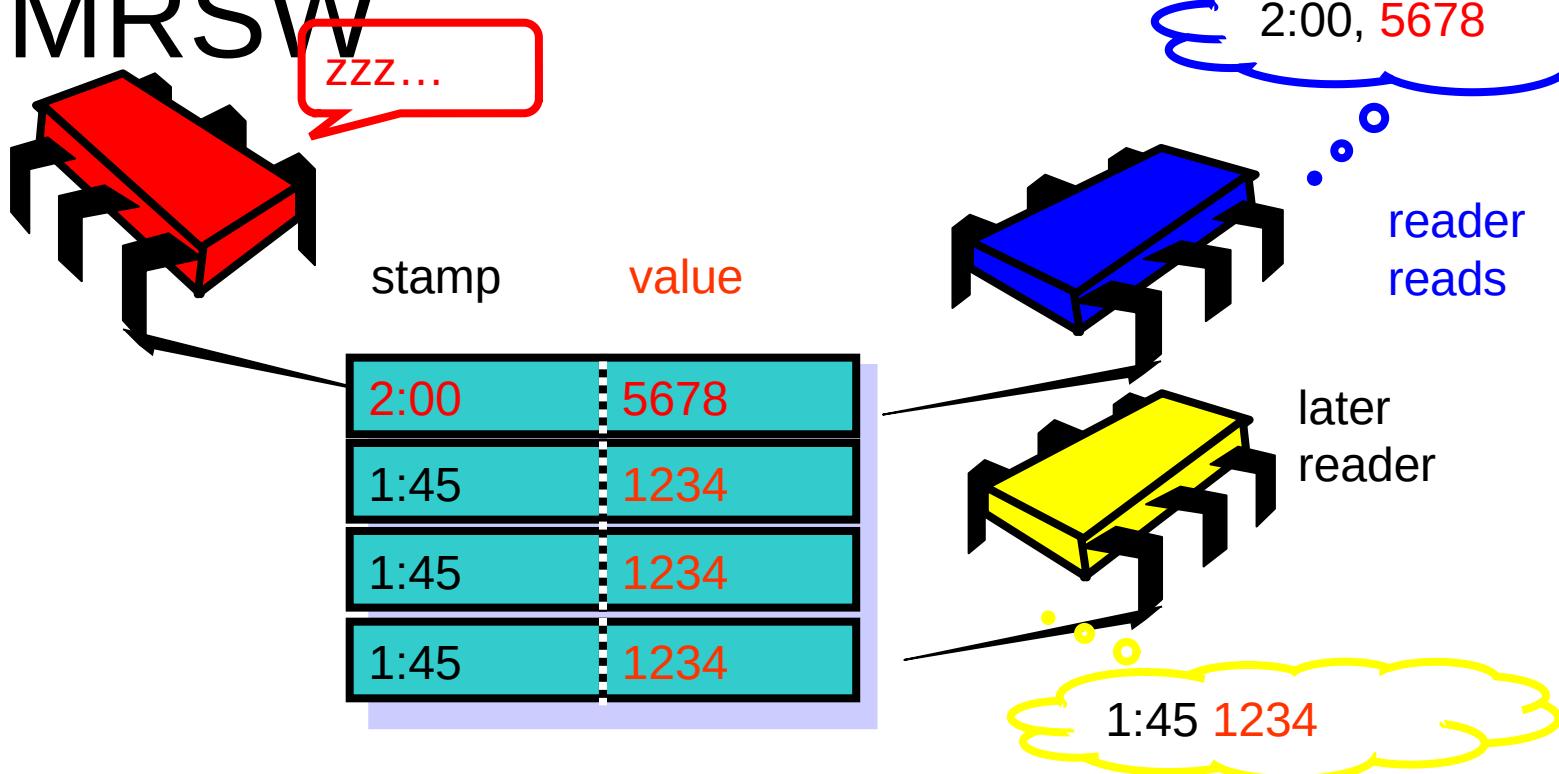
# Atomic Single Reader $\sqsubseteq$ Atomic Multi-Reader



# Atomic MRSW $\sqsubseteq$ Atomic SRSW



# Atomic SRSW $\equiv$ Atomic MRSW



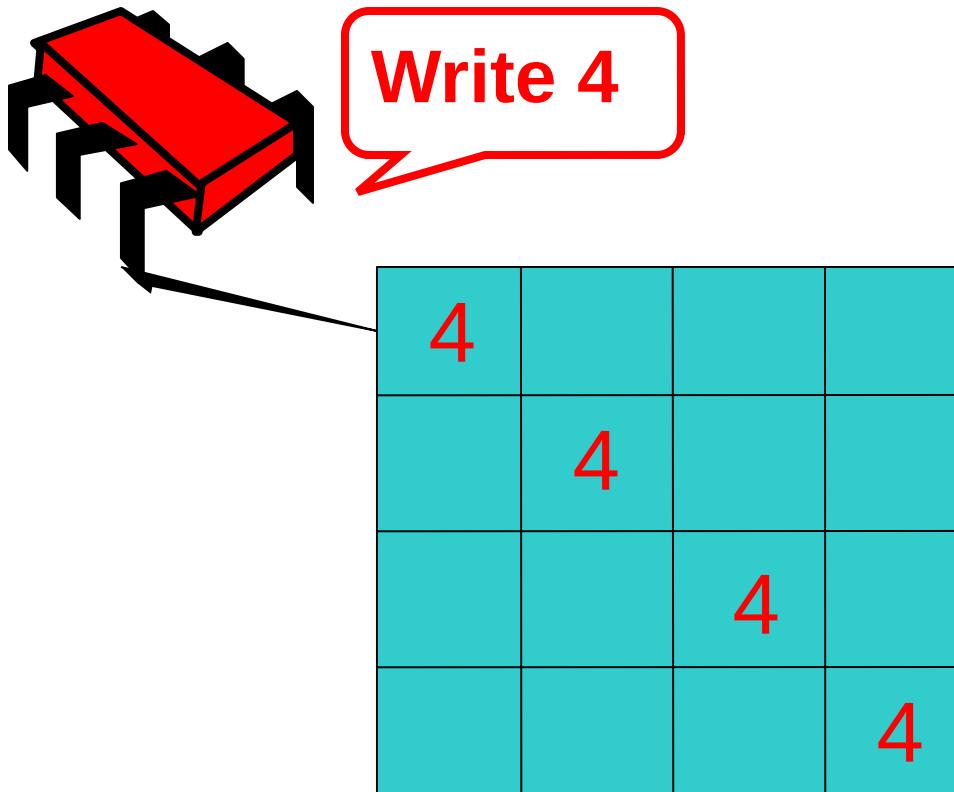
Yellow was completely after Blue but read earlier value...  
not linearizable!

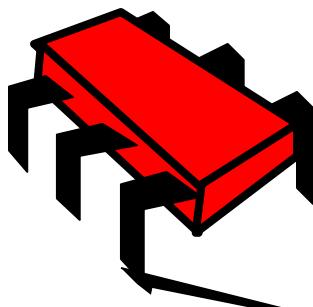
# Atomic MRSW

- We address this problem by having earlier reader threads help out later threads, by telling them which value they read

# Atomic MRSW

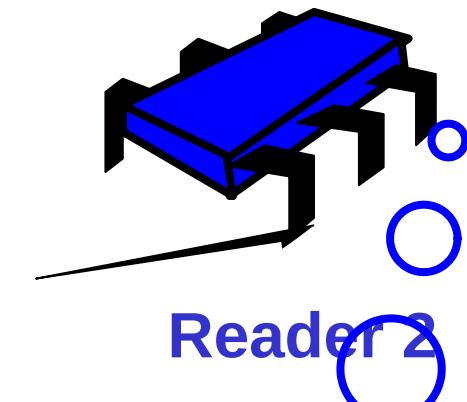
- n-threads share a n-by-n array of stamped values
- Read() calls determine latest threads by timestamps
- Similar to the Safe MRSW Register implementation, the writer writes the new values to the array, but only on the diagonals





ZZZ...

4			
	4		
		4	



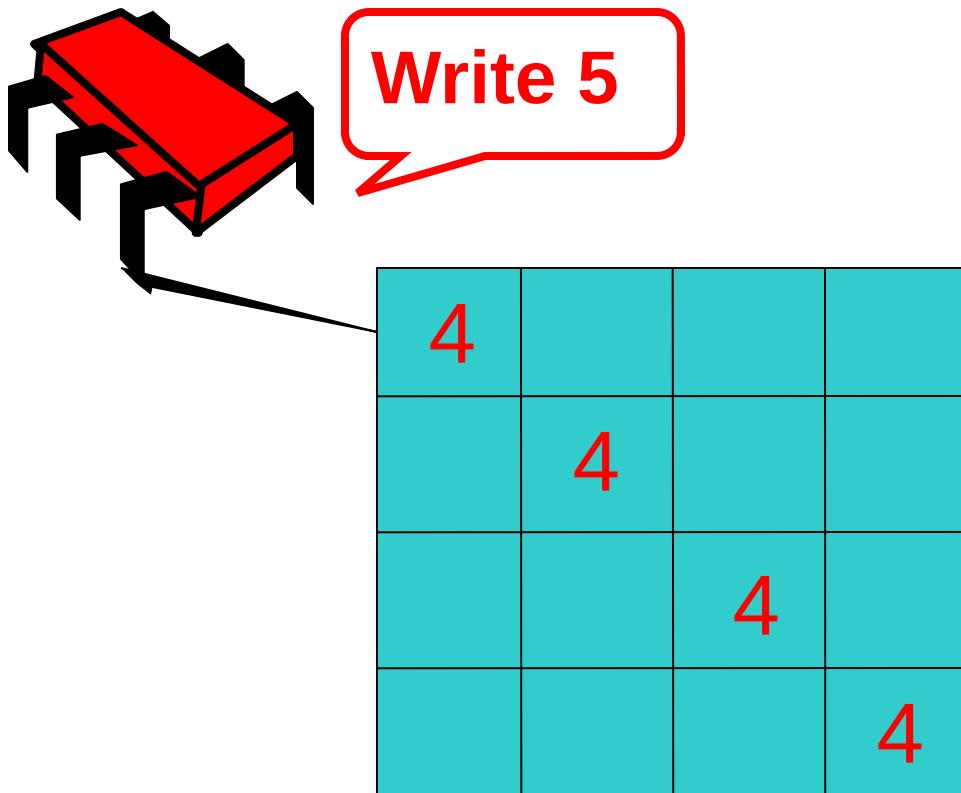
Reader 2

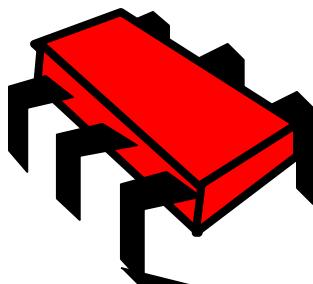
Is there a

value in this  
column

with a  
higher  
timestamp?

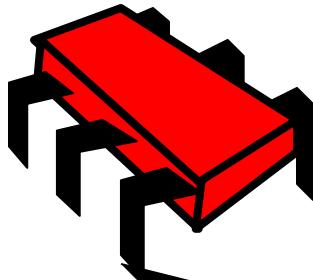
If not then the current value is the latest





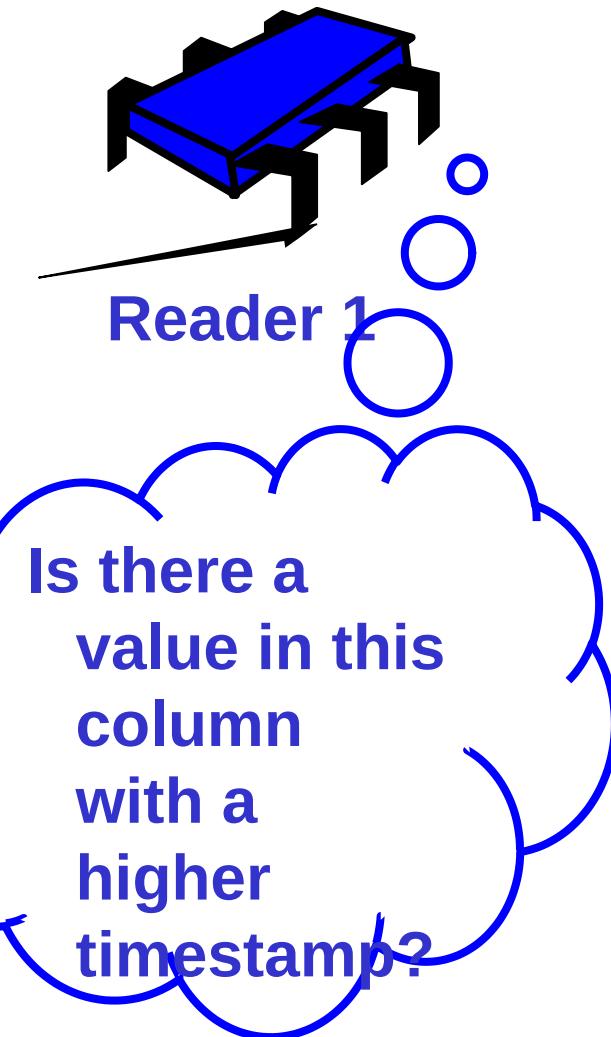
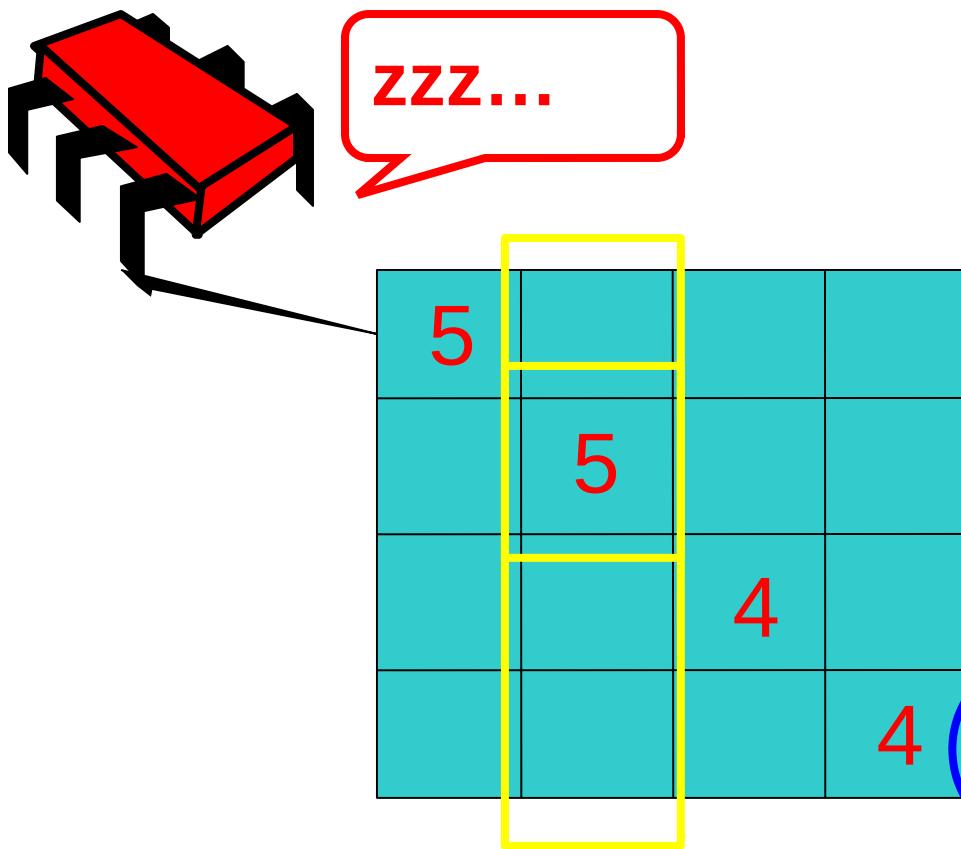
Write 5

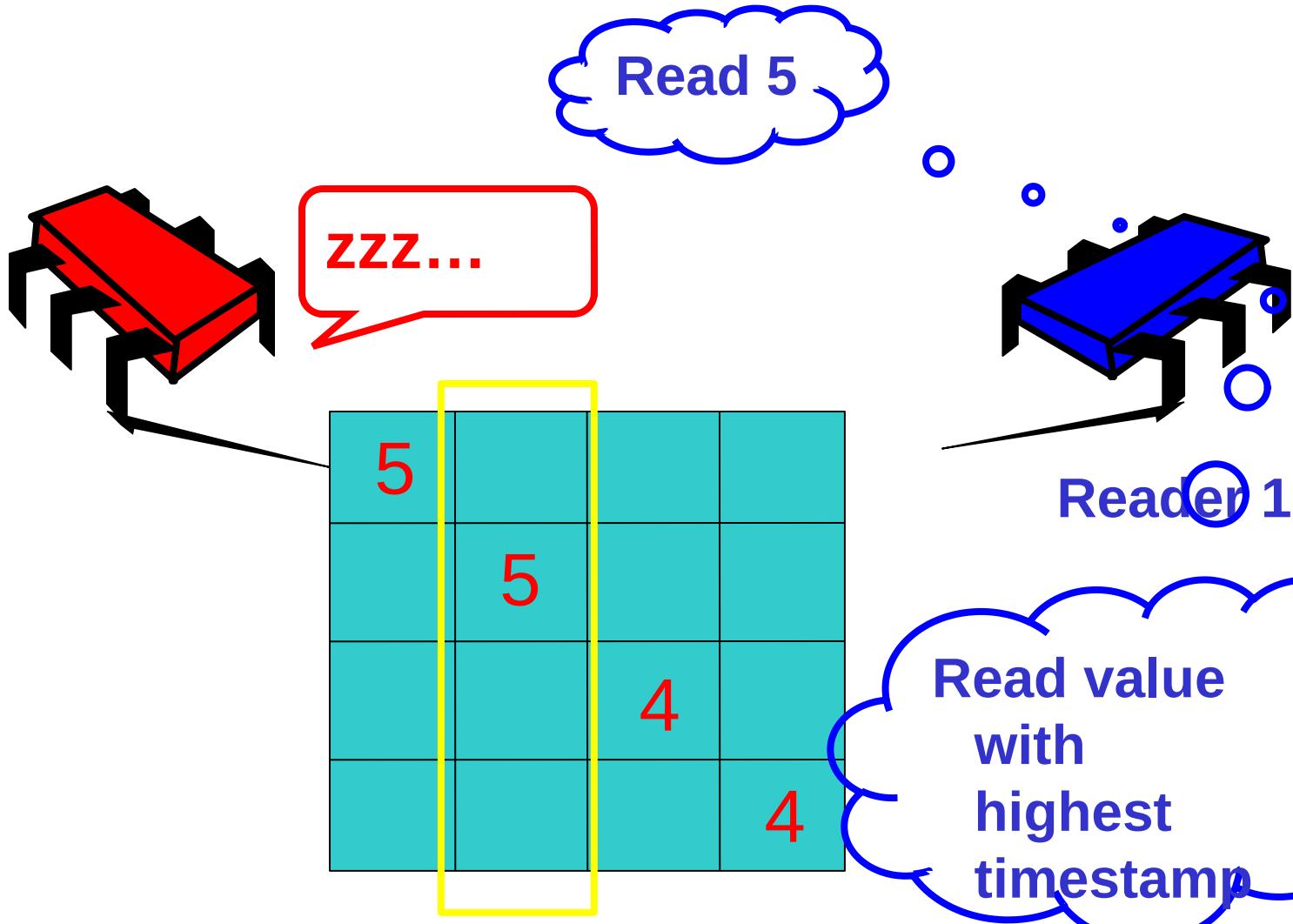
5			
	5		
		4	
			4

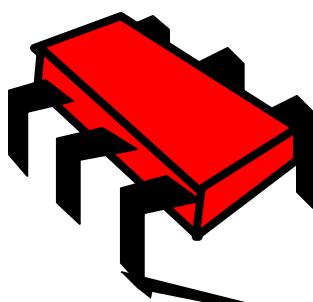


zzz...

5			
	5		
		4	
			4

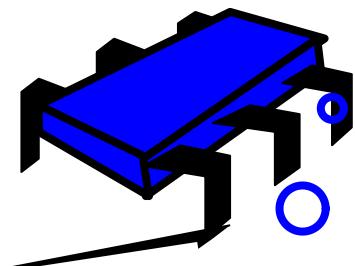






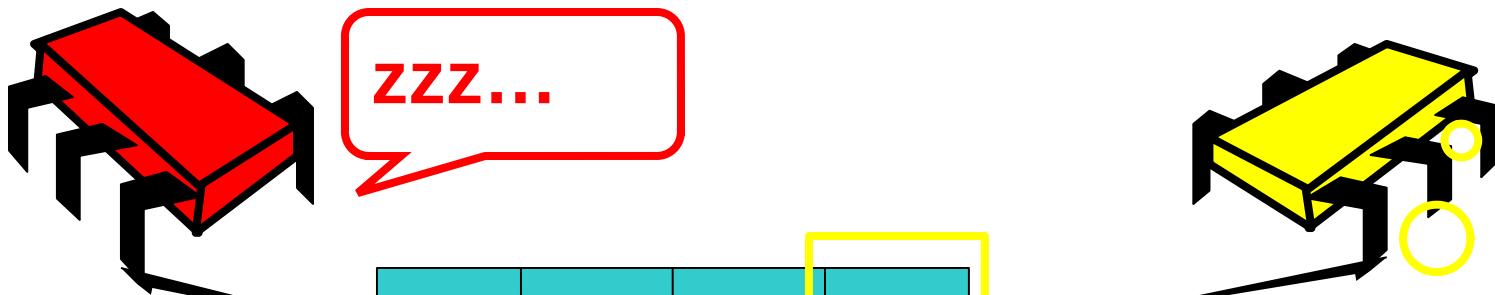
ZZZ...

5			
5	5	5	5
		4	
			4



Reader 1

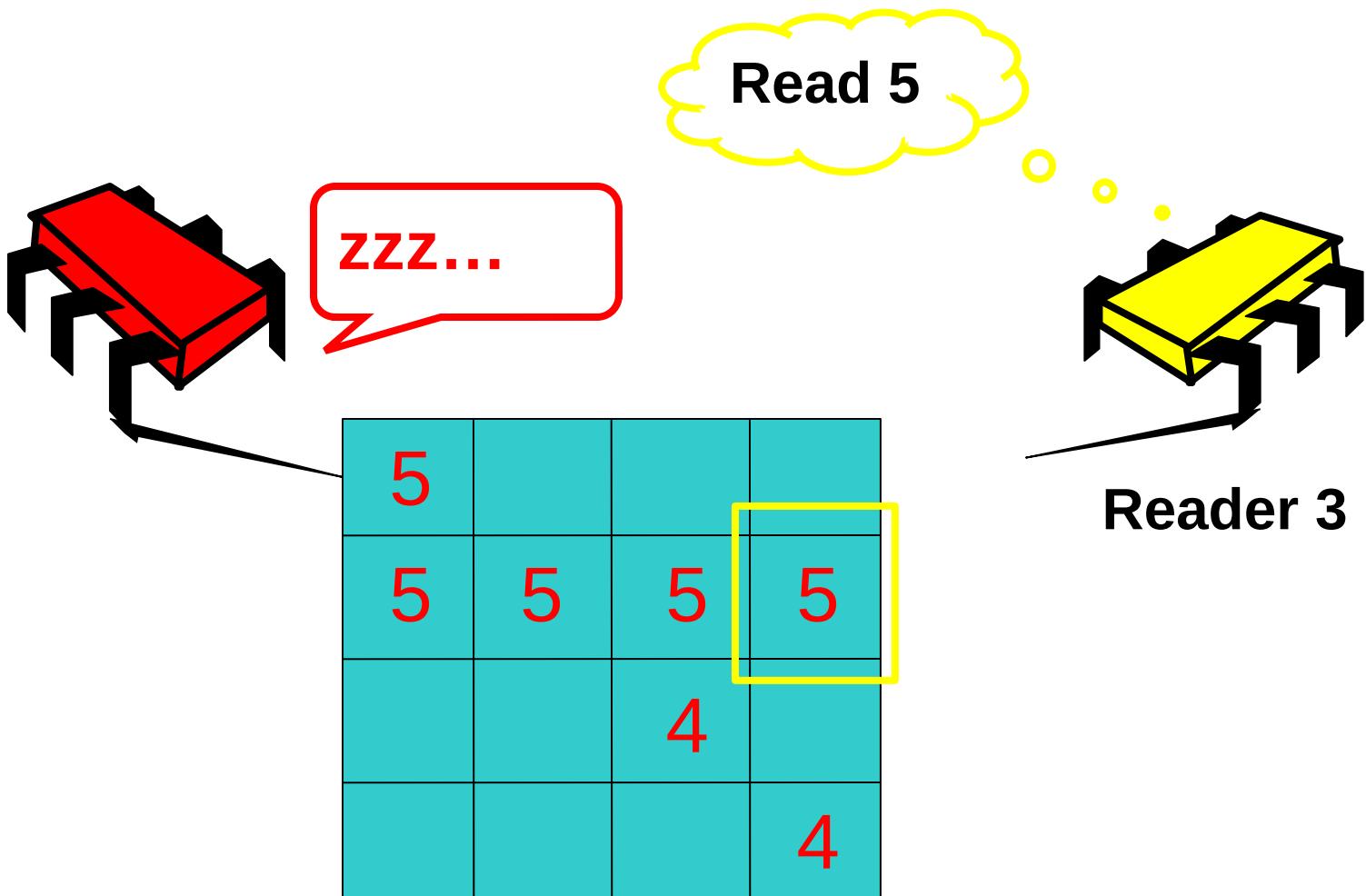
Write highest timestamp to row



5			
5	5	5	5
		4	
			4

Reader 3

Is there a  
value in this  
column  
with a  
higher  
timestamp?



# MRSW Atomic

```
public class AtomicMRSWRegister implements Register{
    long lastStamp;
    StampedValue<T>[][] a_table;

    public T read()  {
        int me = ThreadID.get();
        StampedValue<T> value = a_table[me][me];
        for (int i = 0; i < n; i++)
            value = StampedValue.max(value,
                                      a_table[i][me]);
        for (int i = 0; i < n; i++)
            a_table[me][i] = value;
    }
}
```

*Matrix of  
StampedValues*

# MRSW Atomic

```
public class AtomicMRSWRegister implements Register{
    long lastStamp;
    StampedValue<T>[][] a_table;

    public T read()  {
        int me = ThreadID.get();
        StampedValue<T> value = a_table[me][me];
        for (int i = 0; i < n; i++)
            value = StampedValue.max(value,
                                      a_table[i][me]);
        for (int i = 0; i < n; i++)
            a_table[me][i] = value;
    }
}
```

*Check column for maximum*

# MRSW Atomic

```
public class AtomicMRSWRegister implements Register{
    long lastStamp;
    StampedValue<T>[][] a_table;
```

*Write maximum to  
row*

```
public T read()  {
    int me = ThreadID.get();
    StampedValue<T> value = a_table[me][me];
    for (int i = 0; i < n; i++)
        value = StampedValue.max(value,
                                  a_table[i][me]);
    for (int i = 0; i < n; i++)
        a_table[me][i] = value;
}
```

# MRSW Atomic

```
public void write(T v) {  
    long stamp = lastStamp + 1;  
    lastStamp = stamp;  
    StampedValue<T> value = new StampedValue<T>(stamp,  
    v);  
    for (int i = 0; i < n; i++)  
        a_table[i][i] = value;  
}
```

*Write to diagonal*

# Can't Yellow Miss Blue's Update? ... Only if Readers Overlap...

1:45  
1234

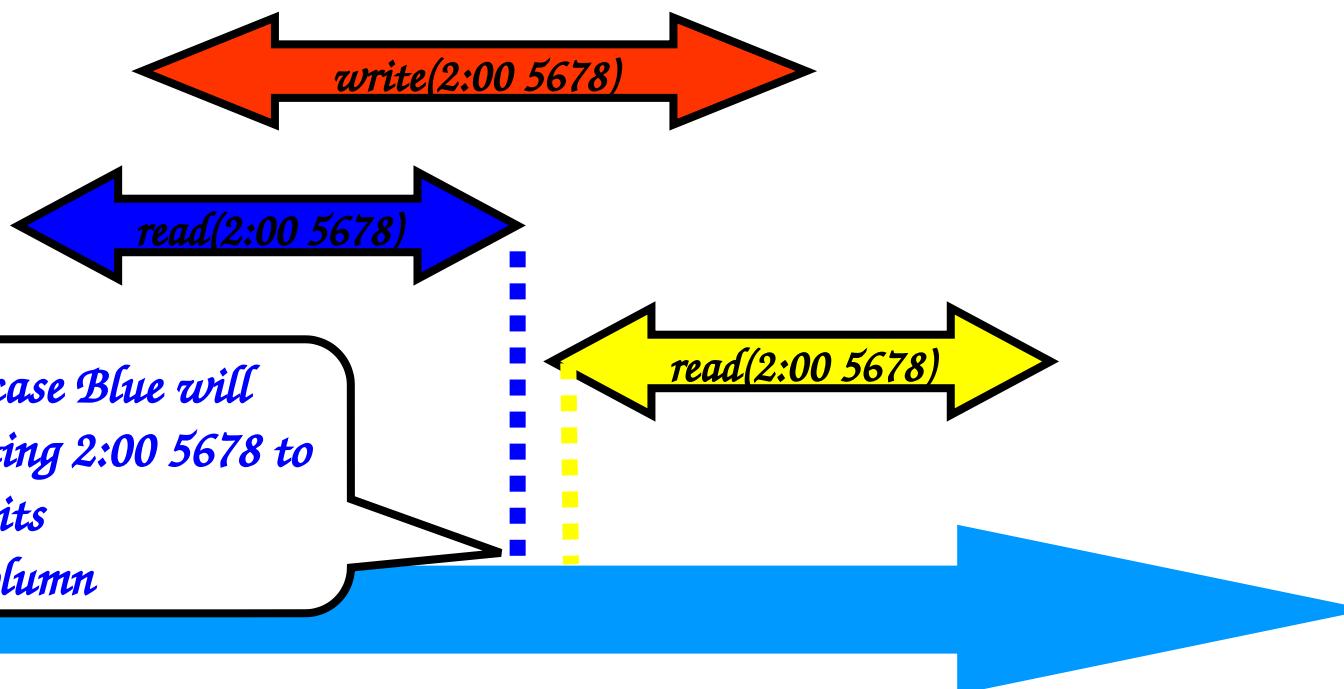


In which case its OK  
to read 1234



# Bad Case Only When Readers Don't Overlap

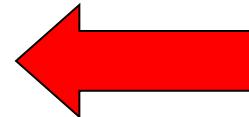
1:45  
1234





# Road Map

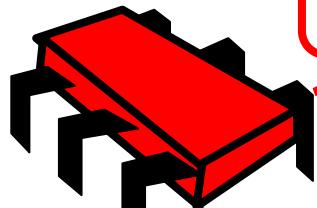
- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic



*Next*

# Multi-Writer Atomic From Multi-Reader Atomic

Writer 1

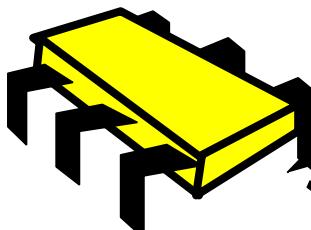


Write 5678

stamp      value

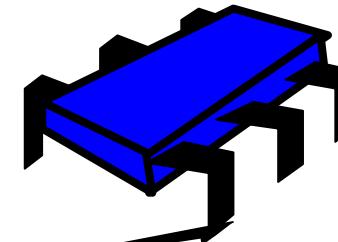
Each writer reads all then writes Max+1 to its register

1:45	1234
2:00	5678
1:45	1234
2:15	X Y Z W



Write X Y Z W

Read



Readers read all and take max (Lexicographic like Bakery)

Max is 2:15, return X Y Z W

# MRMW Atomic

```
public class AtomicMRMWRegister implements Register{  
    StampedValue<T>[] a_table;  
  
    public void write (T value) {  
        int me = ThreadID.get();  
        StampedValue<T> max = StampedValue.MIN;  
        for (int i = 0; i < n; i++)  
            max = StampedValue.max(max,  
                a_table[i]);  
        a_table[me] = new StampedValue(max.stamp + 1,  
value);  
    }  
}
```

*Array of  
StampedValues*

# MRMW Atomic

```
public class AtomicMRMWRegister implements Register{  
    StampedValue<T>[] a_table;  
  
    public void write (T value)  {  
        int me = ThreadID.get();  
        StampedValue<T> max = StampedValue.MIN;  
        for (int i = 0; i < n; i++)  
            max = StampedValue.max(max,  
                a_table[i]);  
        a_table[me] = new StampedValue(max.stamp + 1,  
value);  
    }  
}
```

*Find highest  
timestamp*

# MRMW Atomic

```
public class AtomicMRMWRegister implements Register{  
    StampedValue<T>[] a_table;  
  
    public void write (T value)  {  
        int me = ThreadID.get();  
        StampedValue<T> max = StampedValue.MIN;  
        for (int i = 0; i < n; i++)  
            max = StampedValue.max(max,  
                a_table[i]);  
        a_table[me] = new StampedValue(max.stamp + 1,  
value);  
    }  
}
```

*Write new value to  
array*

# MRMW Atomic

```
public T read() {  
    StampedValue<T> max = StampedValue.MIN;  
    for (int i = 0; i < n; i++)  
        max = StampedValue.max(max, a_table[i]);  
    return max.value;  
}
```

*Find highest  
timestamp*



# Conclusion

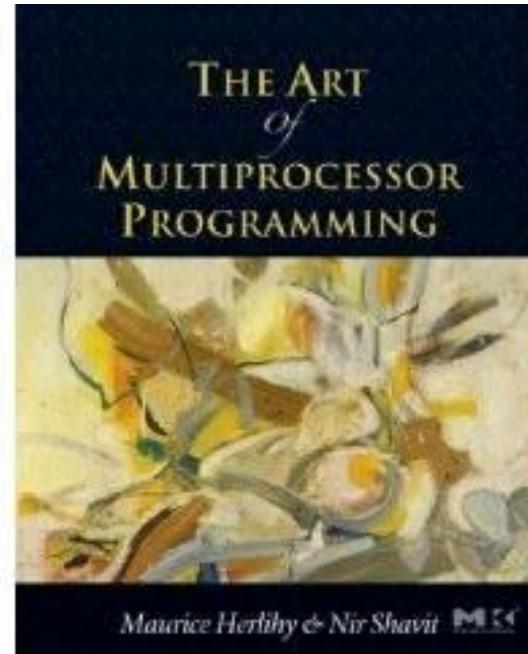
- One can construct a wait-free MRMW atomic register from SRSW Safe Boolean registers

# COS 226

## Chapter 5

### The Relative Power of Primitive Synchronization Operations

# Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



# Background

- Which atomic instructions would you include when designing a new multiprocessor?
- Supporting them all would be inefficient

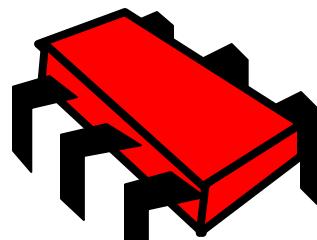
# Wait-Free Implementation

- Every method call completes in finite number of steps
- Implies no mutual exclusion

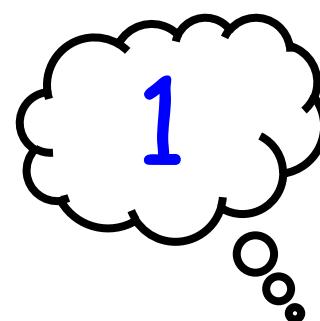


# From Weakest Register

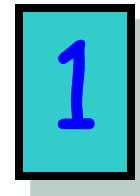
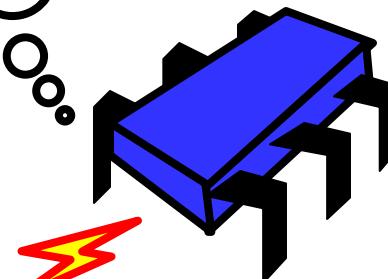
Single writer



0

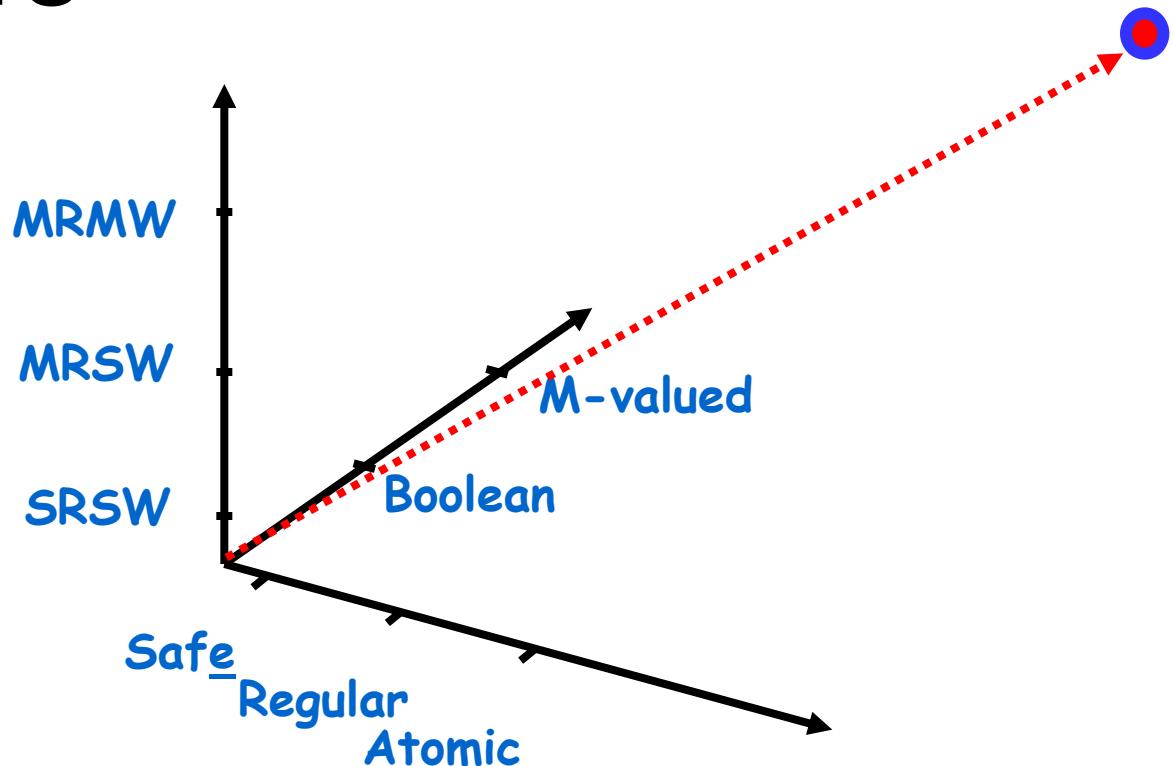


Single reader



Safe Boolean register

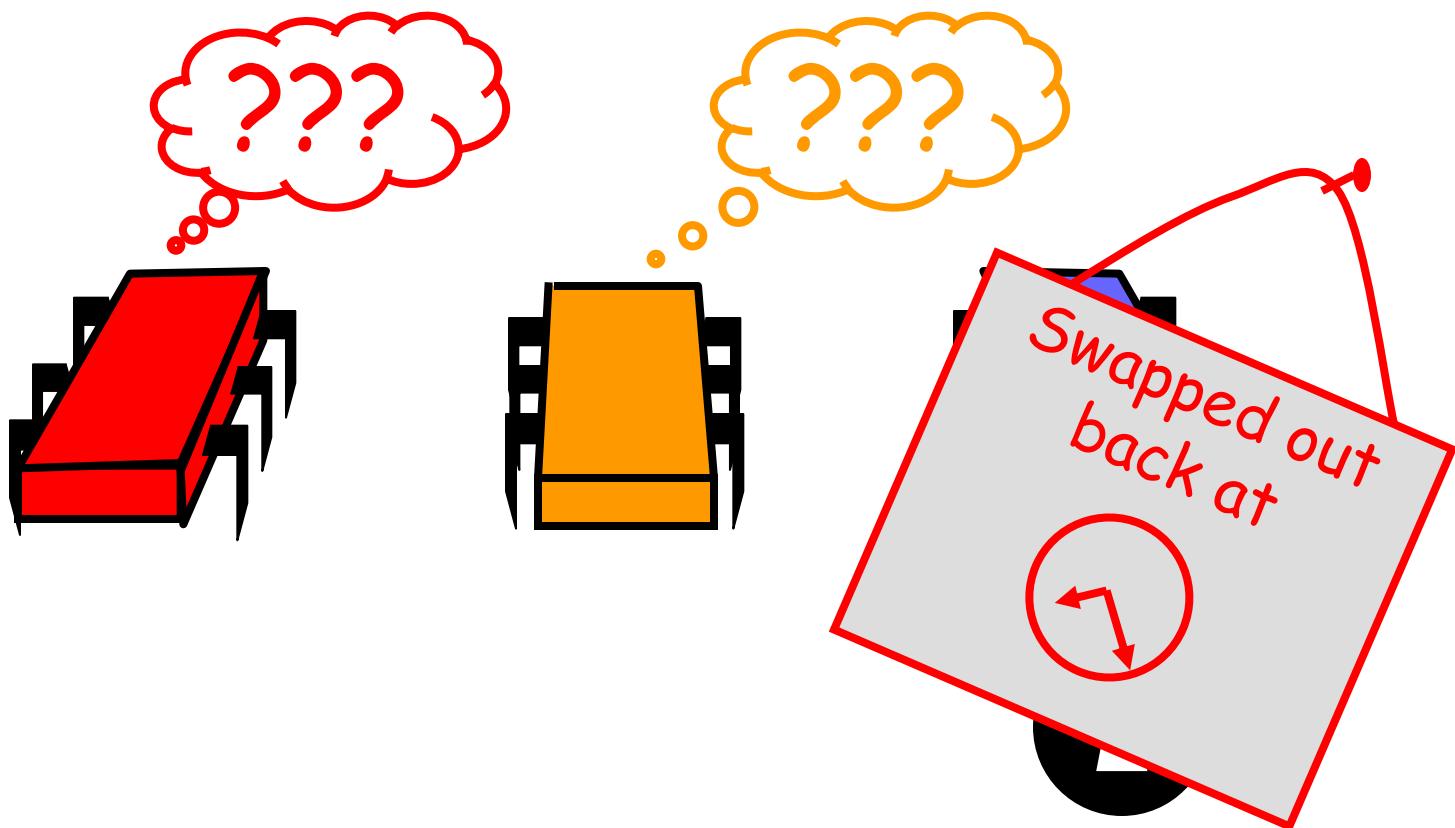
# All the way to a Wait-free Implementation of Atomic Registers



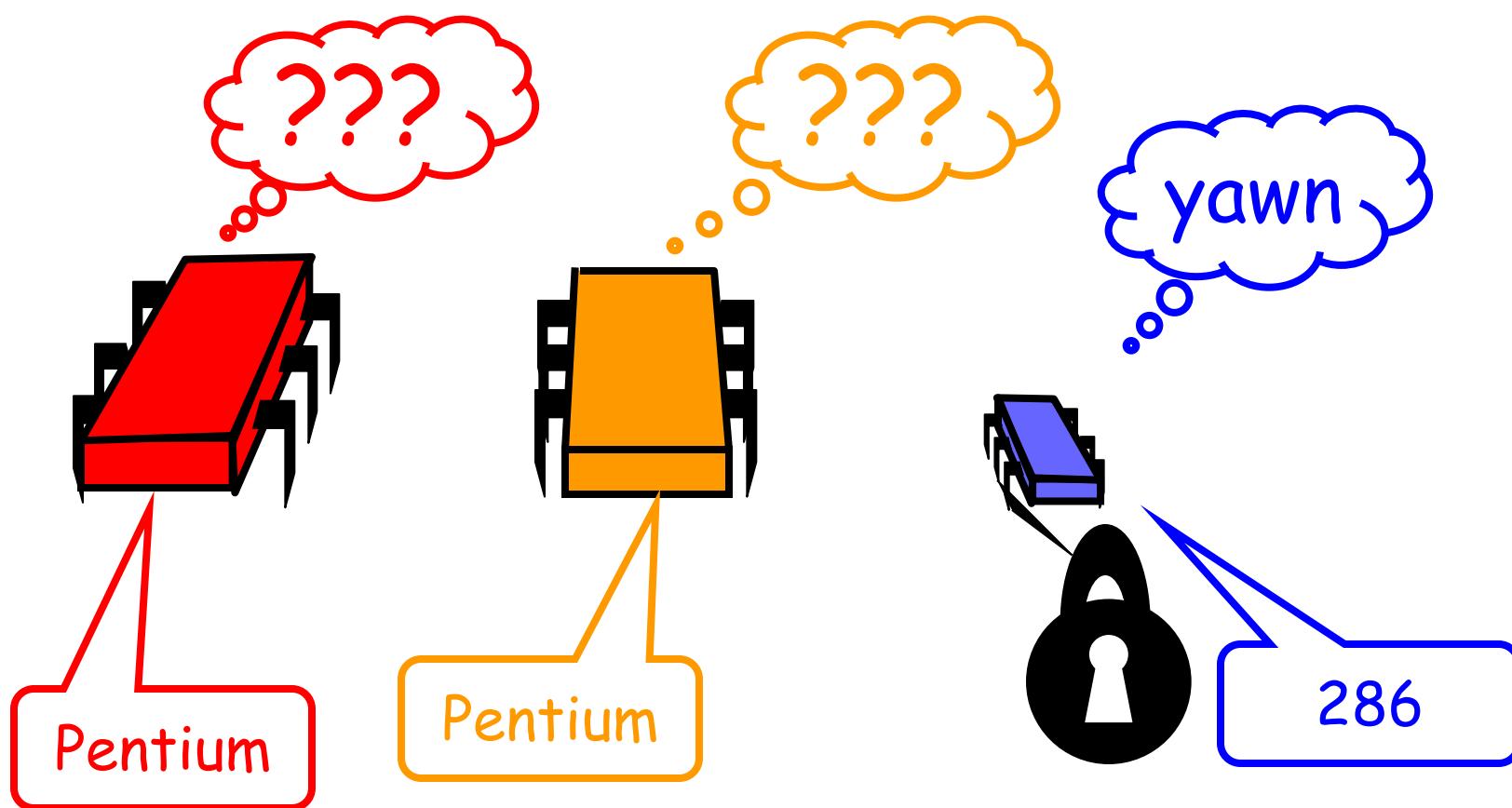


# **Why is Mutual Exclusion so wrong?**

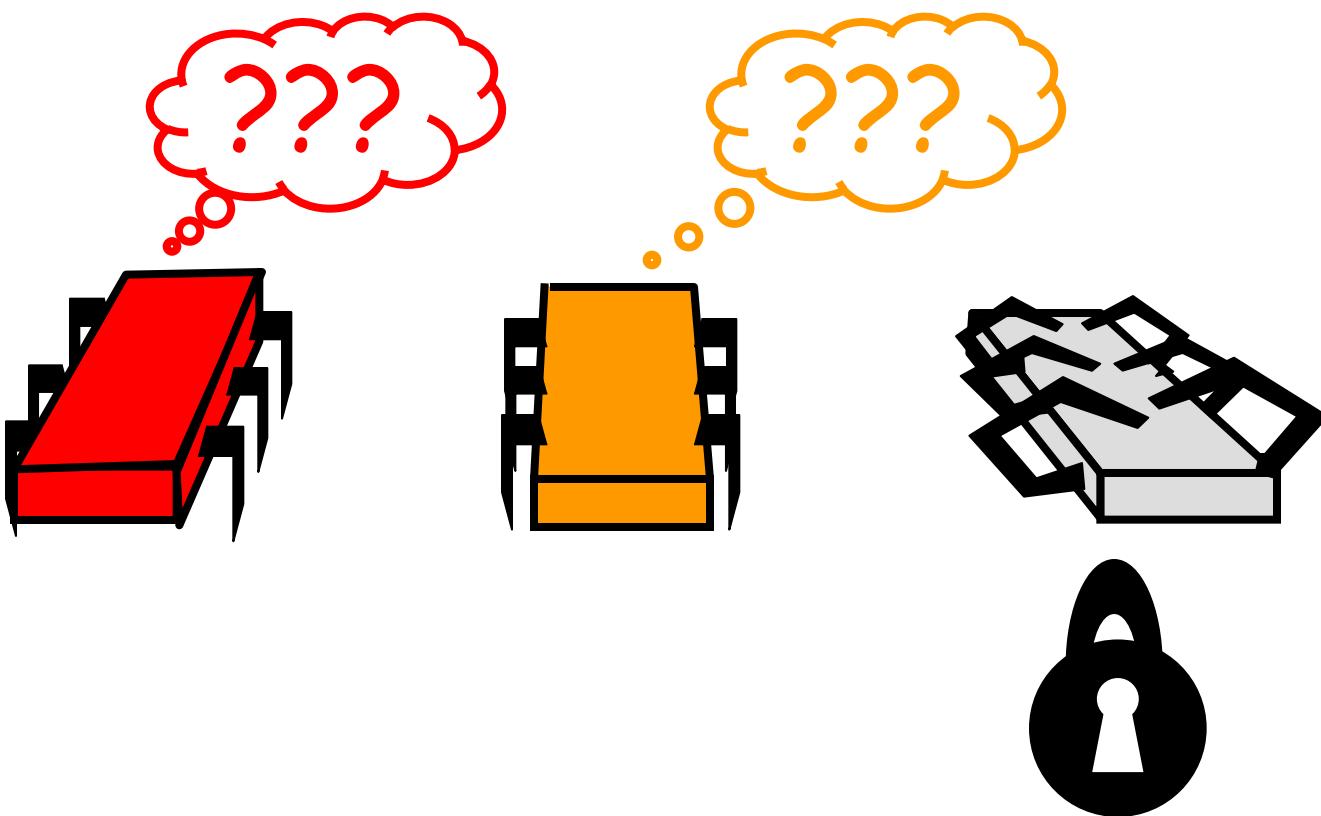
# Asynchronous Interrupts



# Heterogeneous Processors



# Fault-tolerance





# Goal

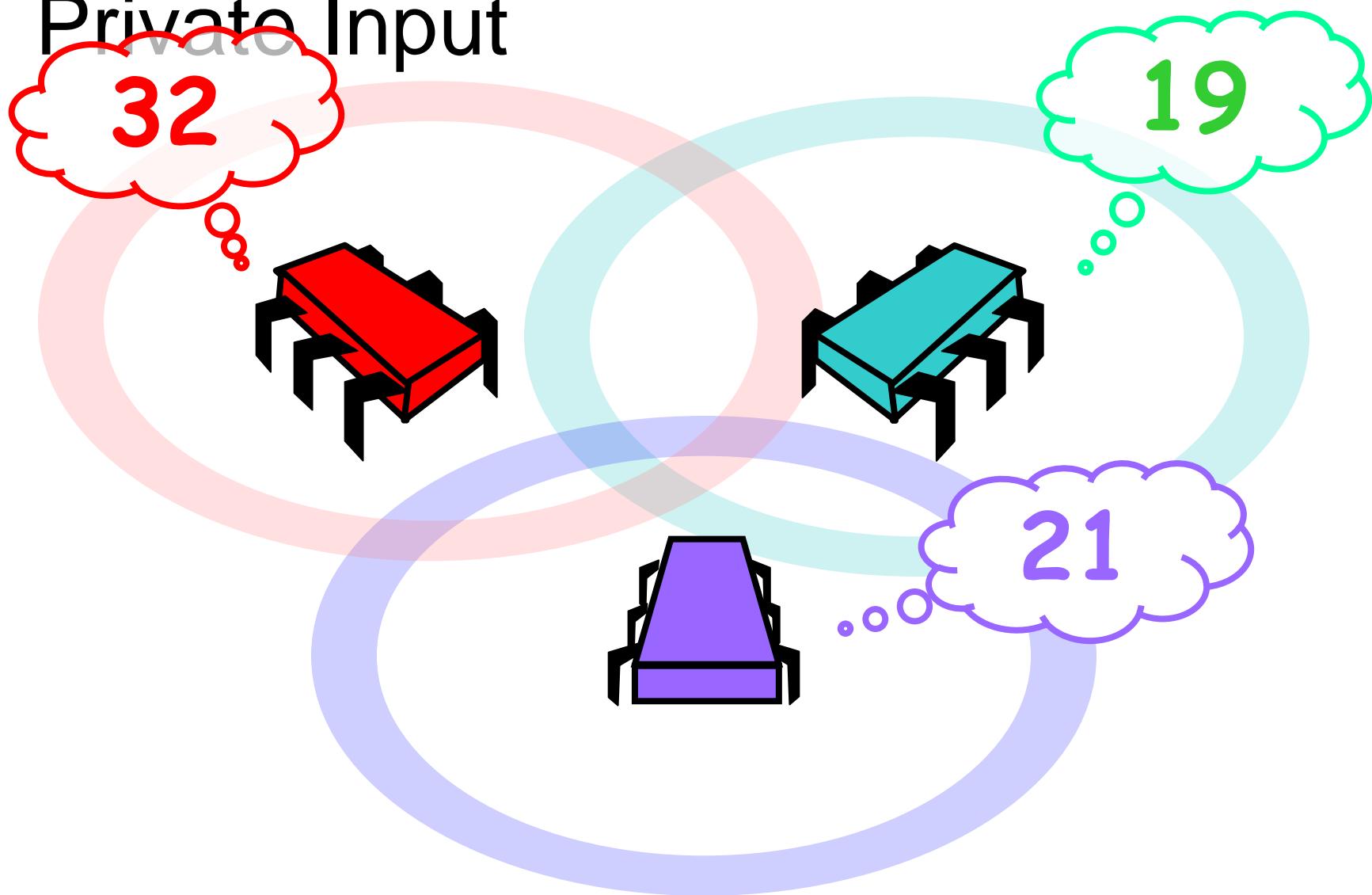
- To identify a set of primitive synchronization operations powerful enough to solve synchronization problems likely to arise in practise
- To do this we need a way to evaluate the *power* of various synchronization primitives
  - What can they solve and how efficiently?



# Consensus

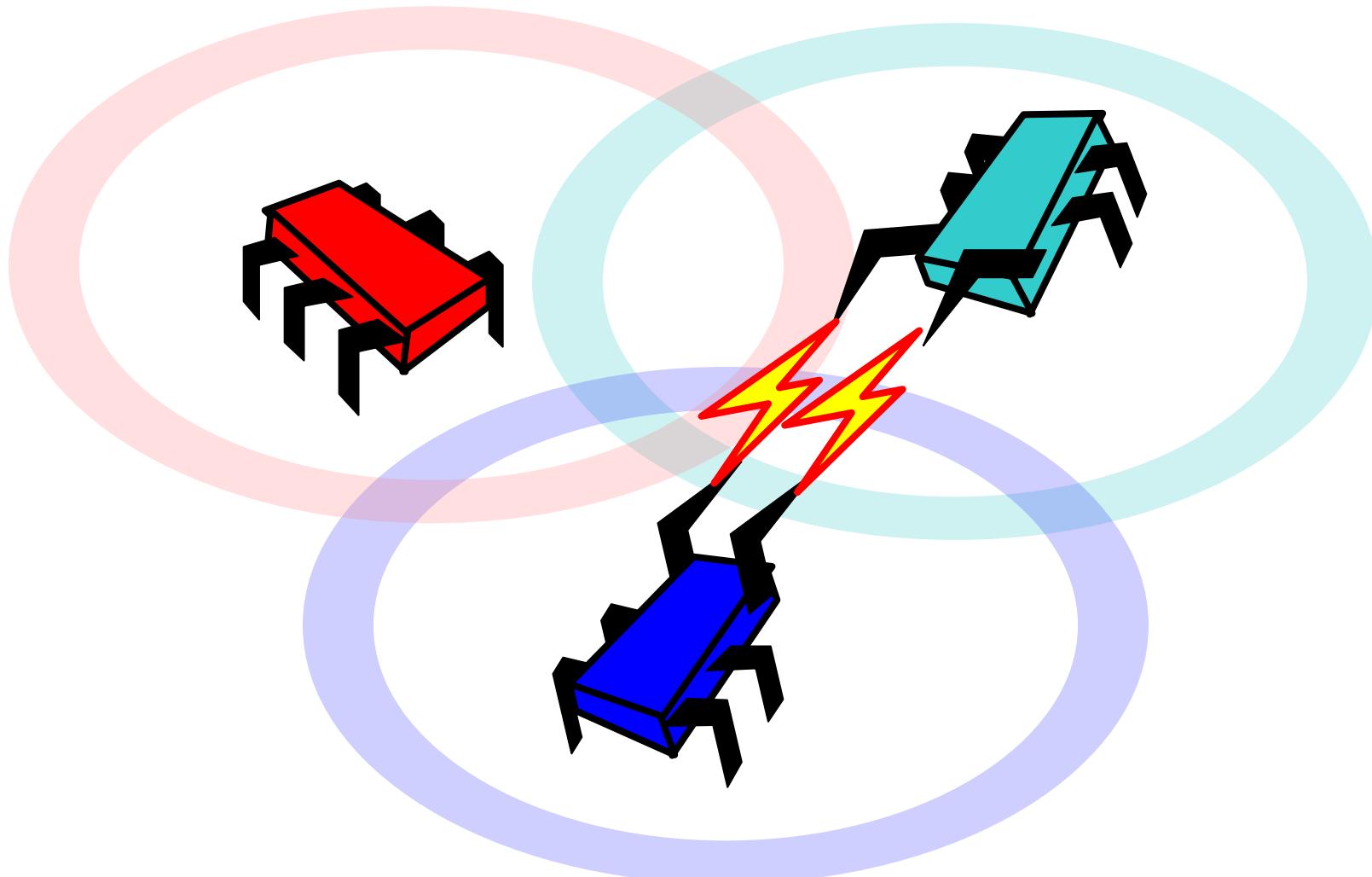
- Not all synchronization instructions are created equal
- A hierarchy of synchronization primitives exist
- We are going to use the principle of **consensus** to determine the *power* of synchronization primitives

# Consensus: Each Thread has a Private Input



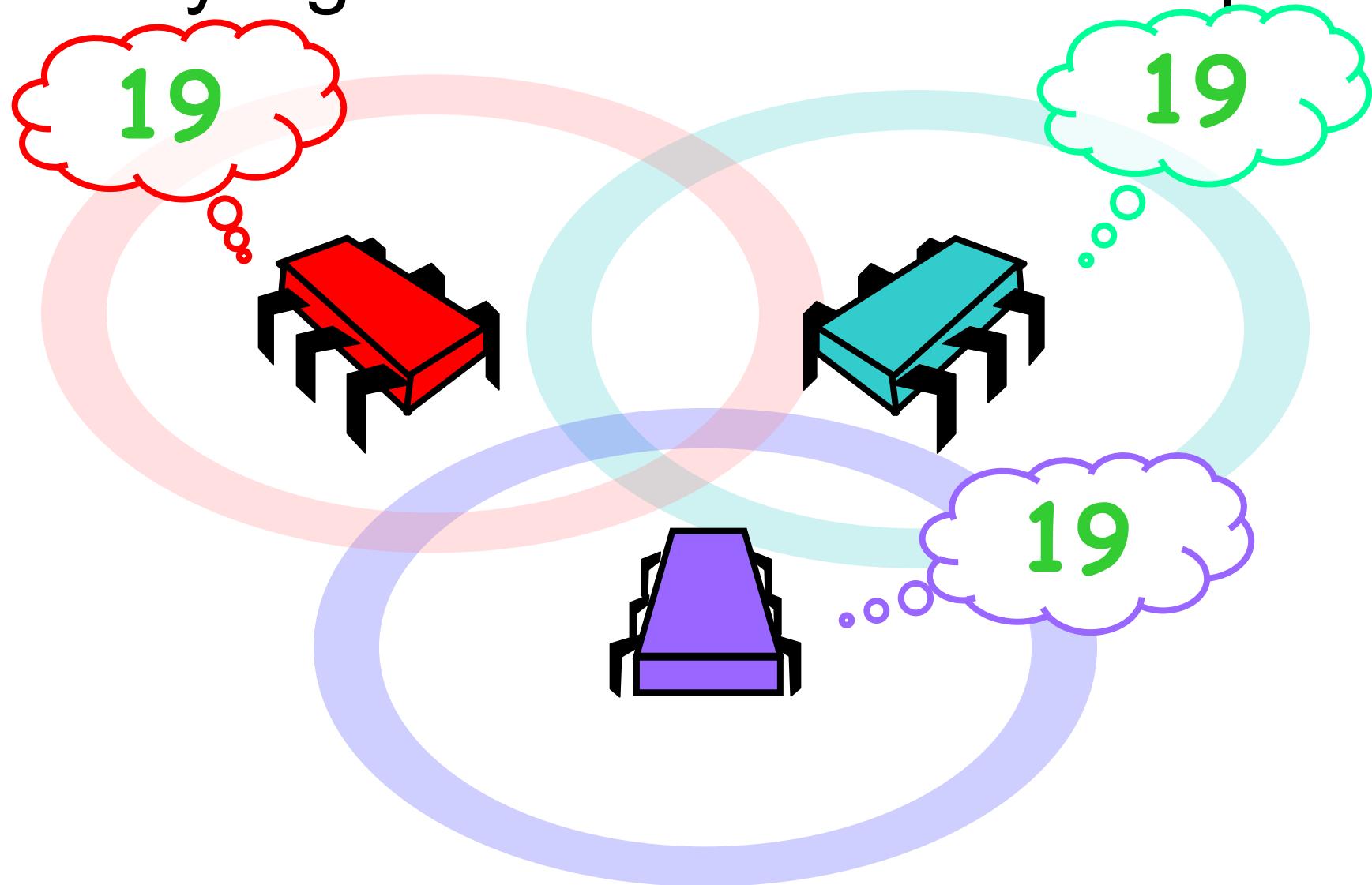


# They Communicate





# They Agree on One Thread's Input



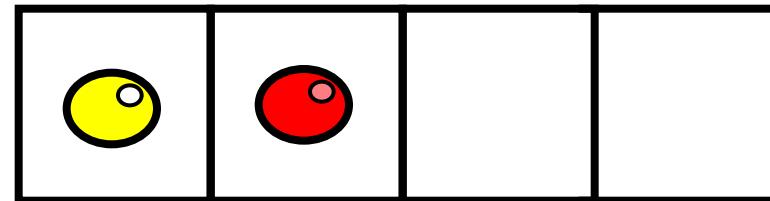


# Consensus

- Basic idea:
  - Each class in hierarchy has an associated consensus number
    - Maximum number of threads for which objects can solve consensus

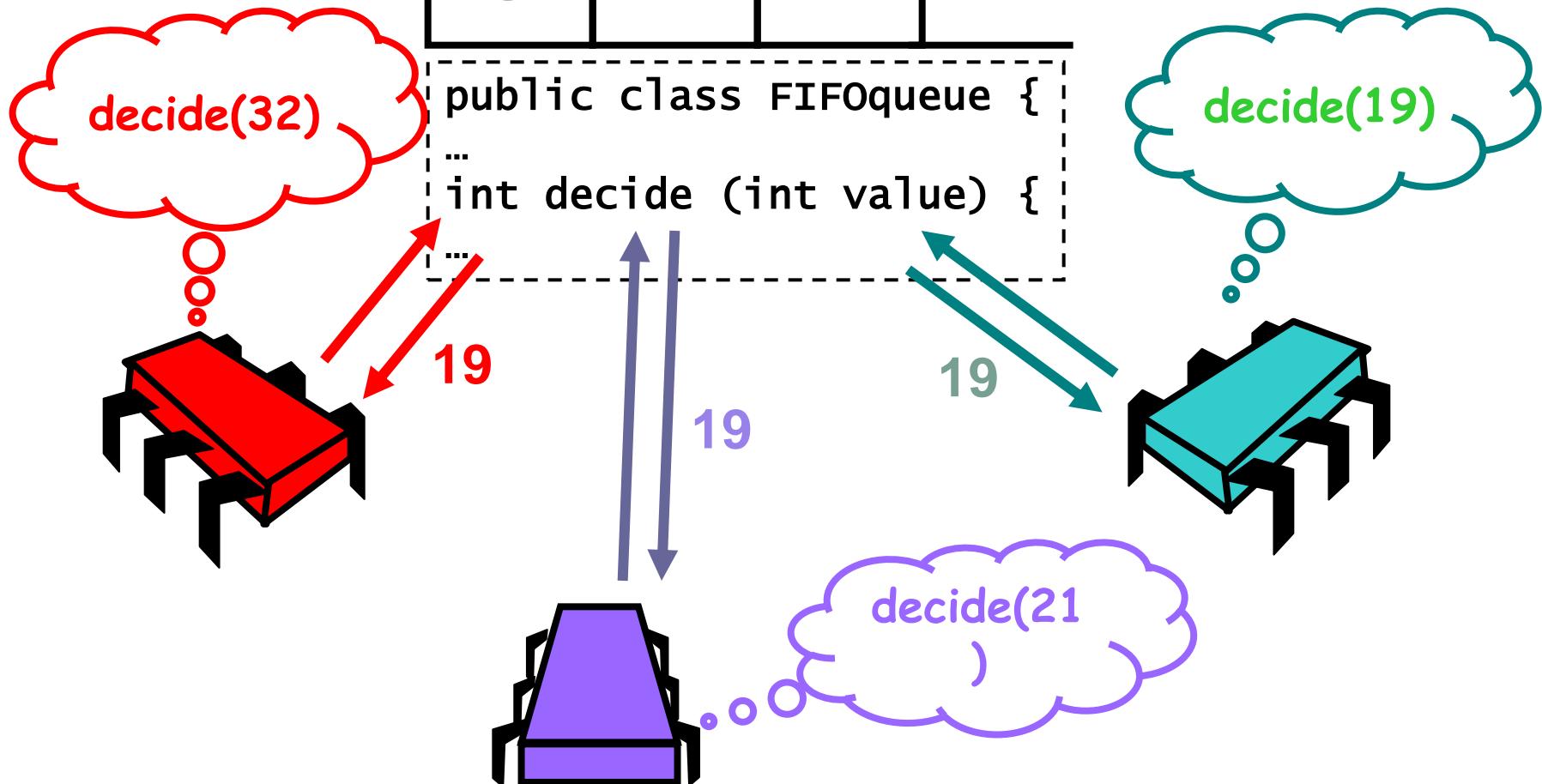
# Formally: Consensus

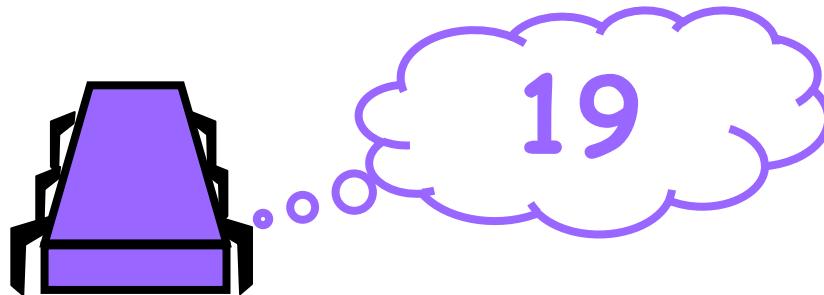
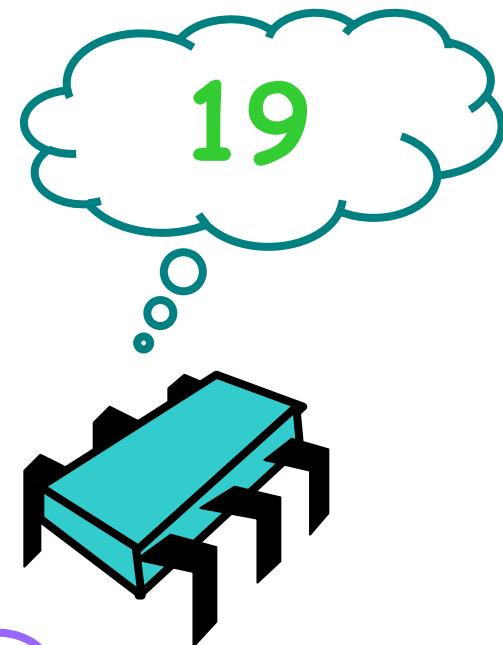
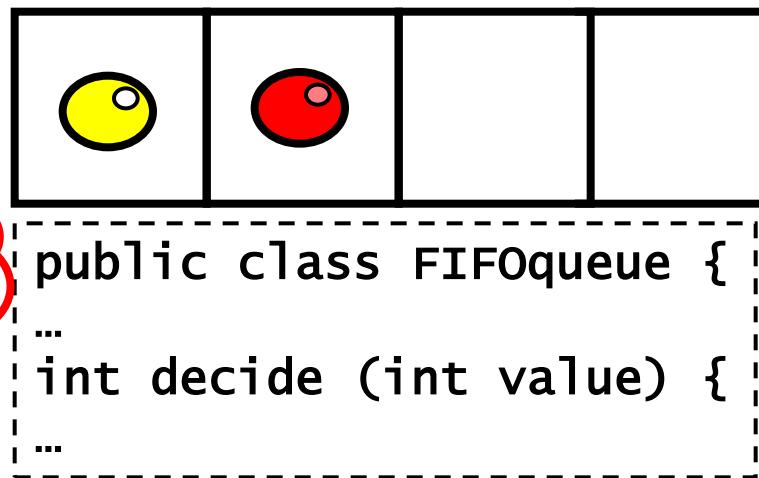
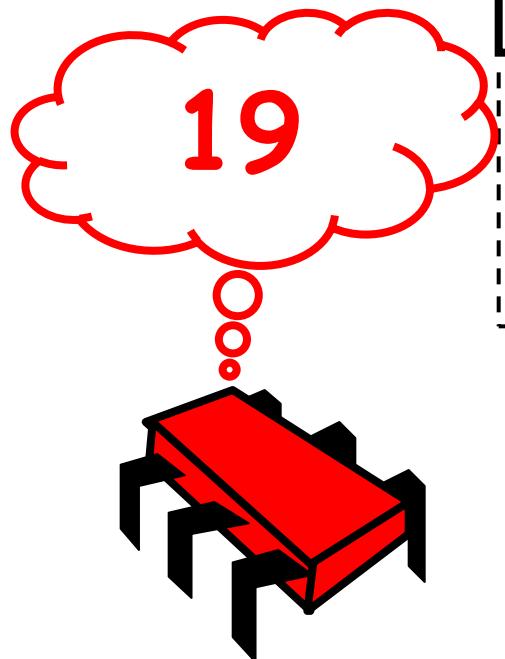
- A consensus object has a `decide()` method
- Each thread calls the `decide()` method with its input at most once
- `decide()` returns a value with the following conditions:
  - Consistent:
    - all threads decide the same value
  - Valid:
    - the common decision value is some thread's input



```
public class FIFOqueue {
```

```
    int decide (int value) {
```







# Consensus

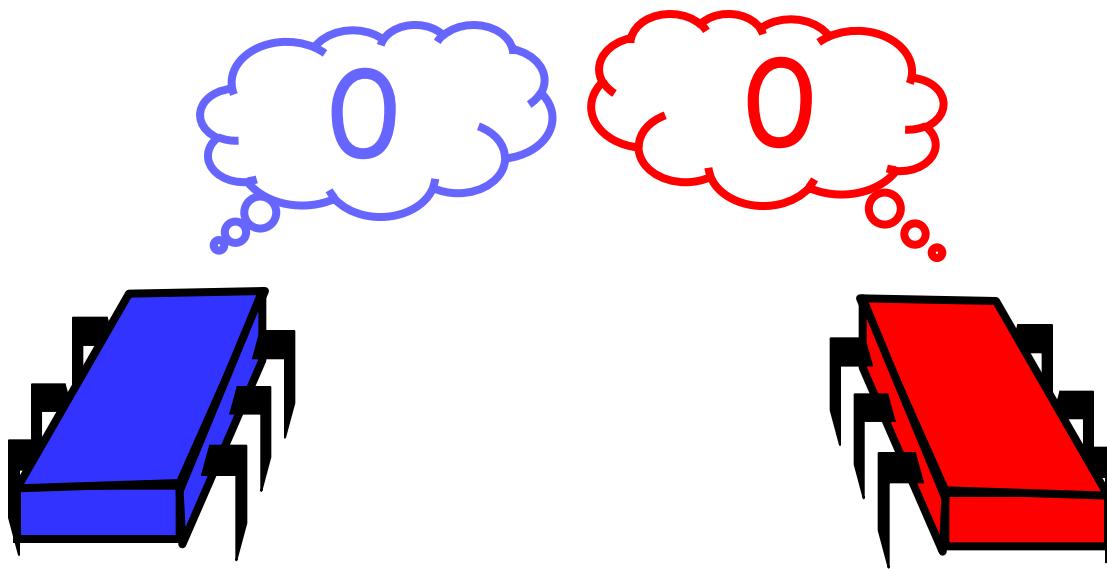
- We are now going to look at different concurrent object classes and see whether they solve consensus



# Consensus

- Can consensus be reached using atomic registers?

# Both Inputs 0



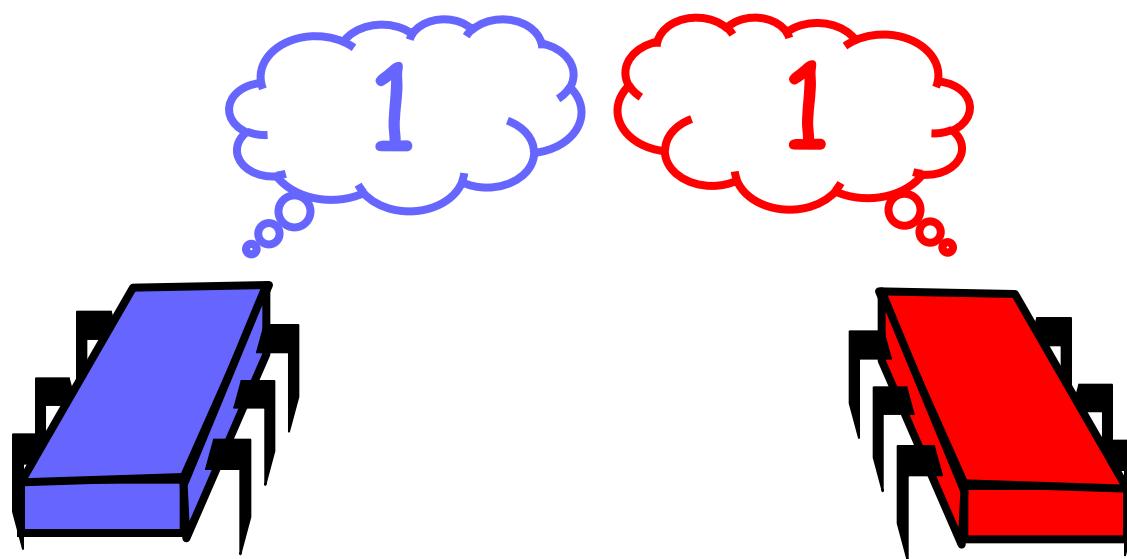
Univalent: all executions must decide 0

# Both Inputs 0



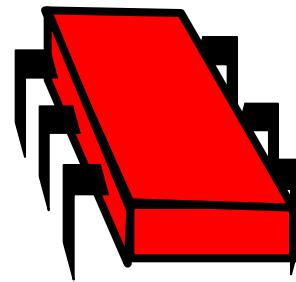
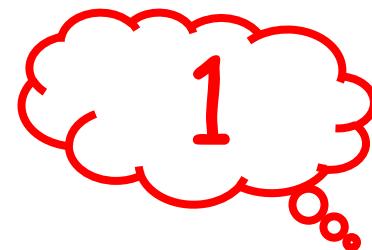
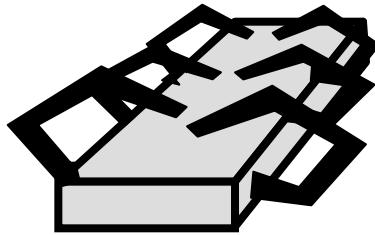
Including this solo execution by A

# Both Inputs 1



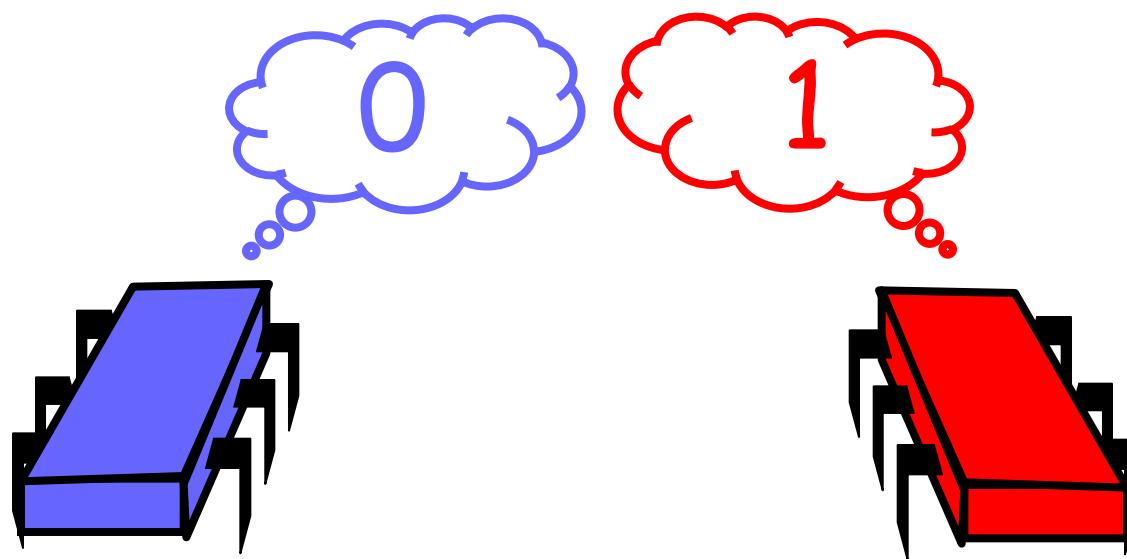
Univalent: all executions must decide 1

# Both Inputs 1

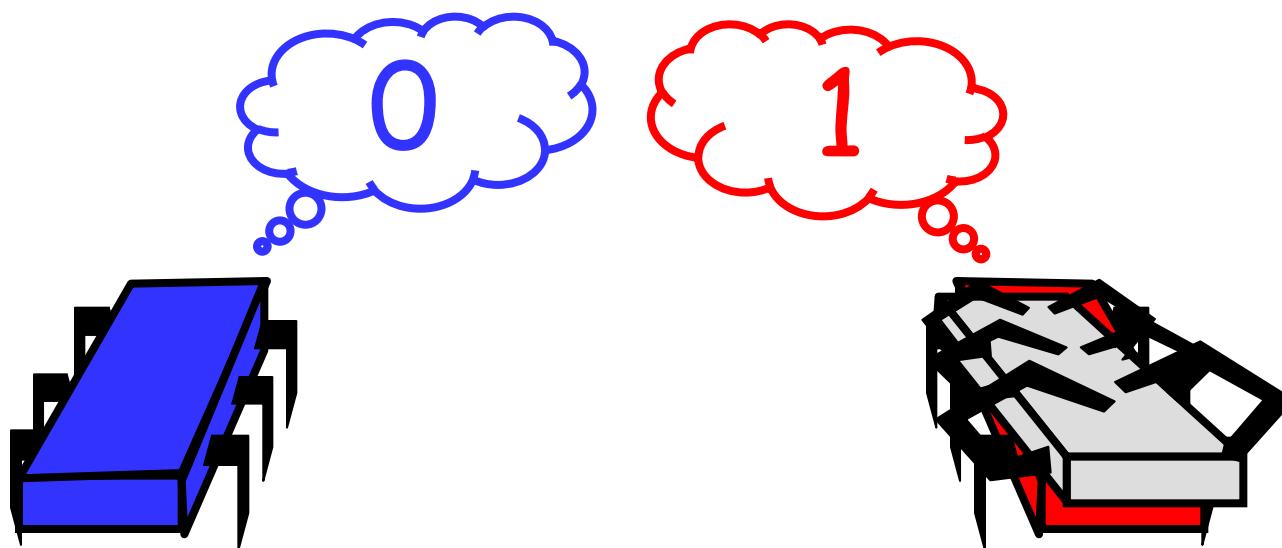


Including this solo execution by B

# What if inputs differ?

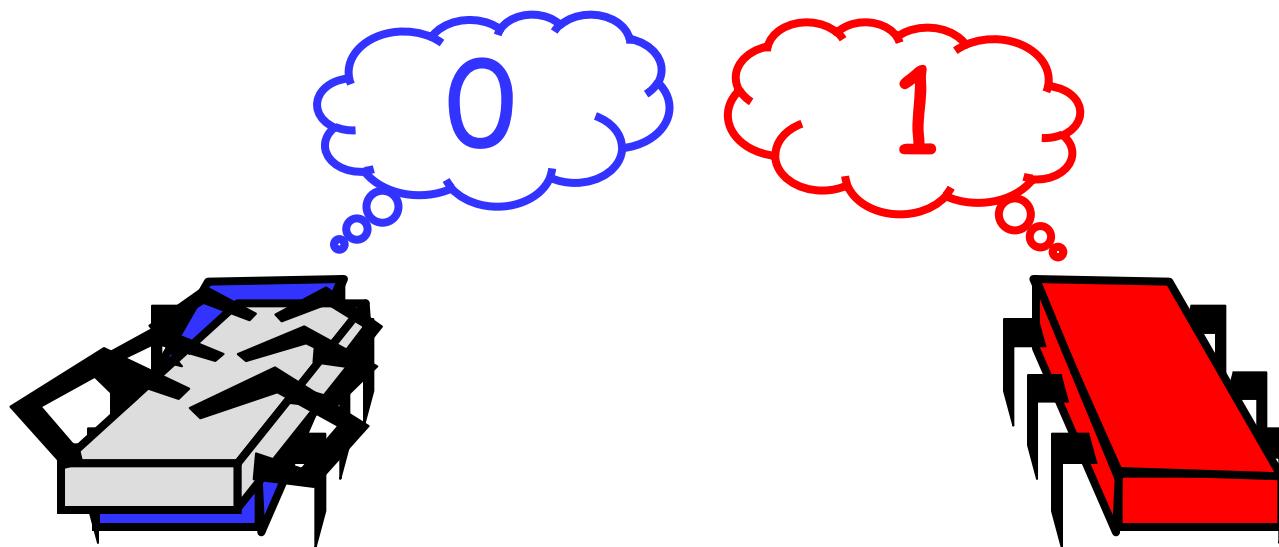


# The Possible Executions



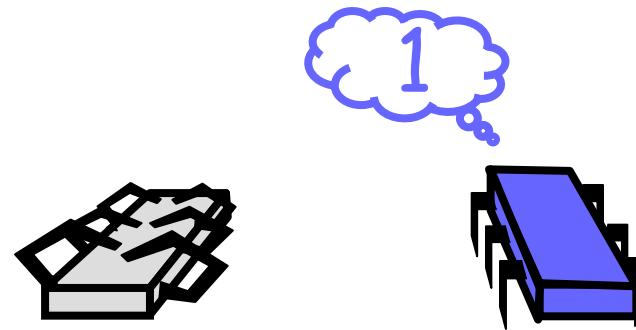
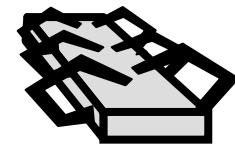
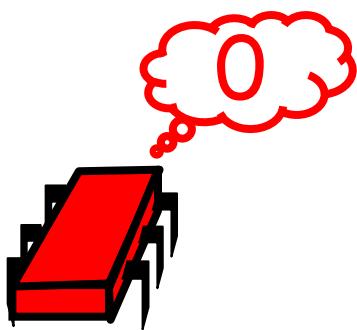
Include the solo execution by A  
that decides 0

# The Possible Executions



Also include the solo execution by B  
which decides 1

# Possible Executions Include



- Solo execution by A must decide 0
- Solo execution by B must decide 1



# Atomic Registers

- Method calls:
  - One of the threads reads from the register
  - Both threads write to separate registers, or
  - Both threads write to the same registers
- The proofs show that in each case two threads cannot reach consensus on two values using atomic registers

# Theorem 5.2.1

- Atomic registers have consensus number 1.
- Consensus numbers:
  - The consensus number of a class is the largest number of threads that can solve consensus using that class.
  - Consensus number 1 means only sequential.



# Consensus Object

```
public interface Consensus {  
    Object decide(Object value);  
}
```

# Generic Consensus Protocol

```
abstract class ConsensusProtocol<T>
    implements Consensus {
    protected T[] proposed = new T[N];

    protected void propose(T value) {
        proposed[ThreadID.get()] = value;
    }

    abstract public T decide(T value);
}
```

# Generic Consensus Protocol

```
abstract class ConsensusProtocol<T>
    implements Consensus {
    protected T[] proposed = new T[N];
```

```
    protected void propose(T value) {
        proposed[ThreadID.get()] = value;
    }
```

```
    abstract public T d
}
```

Each thread's  
proposed value

# Generic Consensus Protocol

```
abstract class ConsensusProtocol<T>
    implements Consensus {
    protected T[] proposed = new T[N];

    protected void propose(T value) {
        proposed[ThreadID.get()] = value;
    }

    abstract public T decide(T value);
}
```

**Propose a value**

# Generic Consensus Protocol

at

Decide a value: abstract method  
means subclass does the heavy lifting  
(real work)

```
protected void propose(T value) {  
    proposed[ThreadID.get()] = value;  
}
```

abstract public T decide(T value);



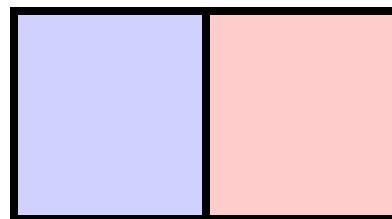
# Can a FIFO Queue Implement Consensus?



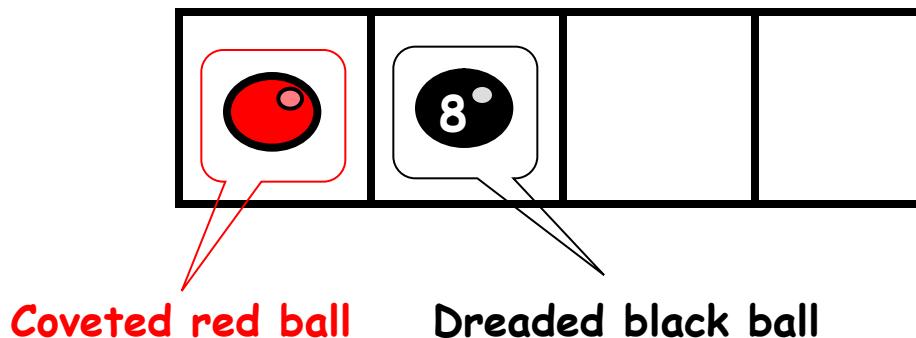
# FIFO Queue

- Let's start with 2-threads...

# FIFO Consensus



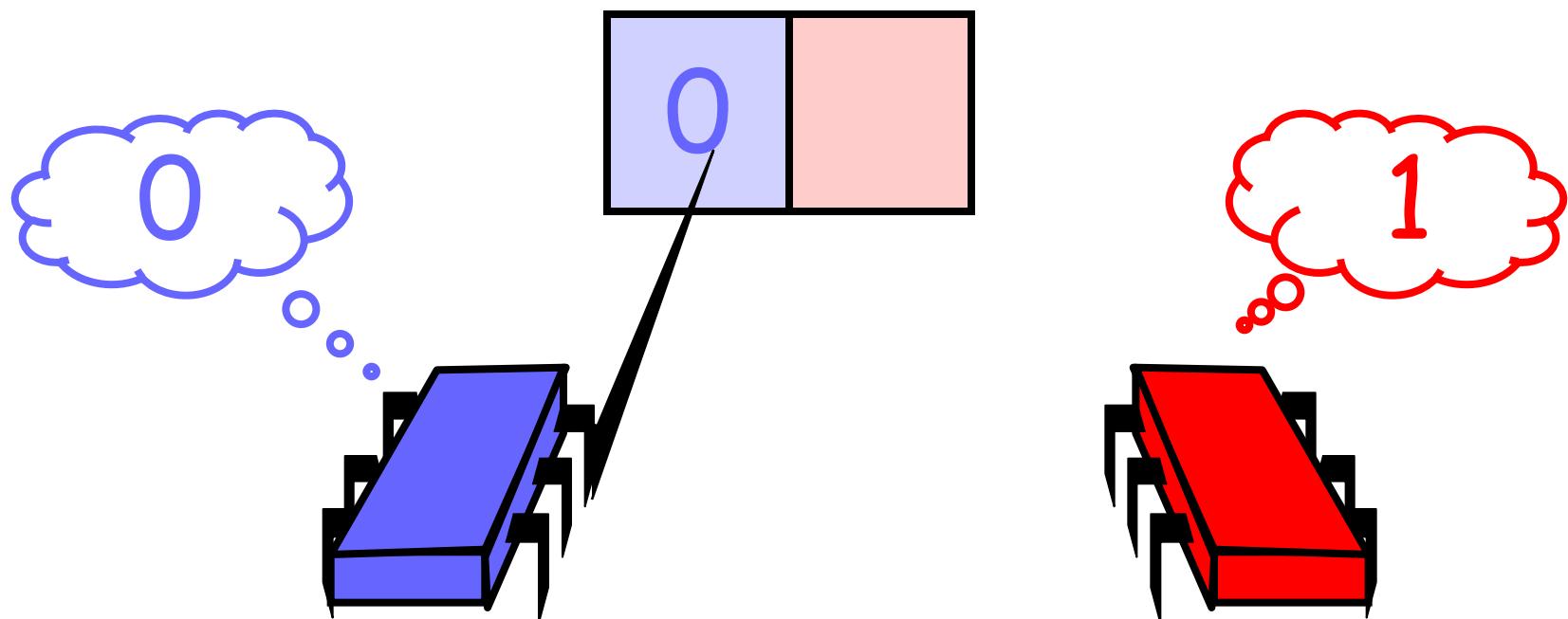
proposed array



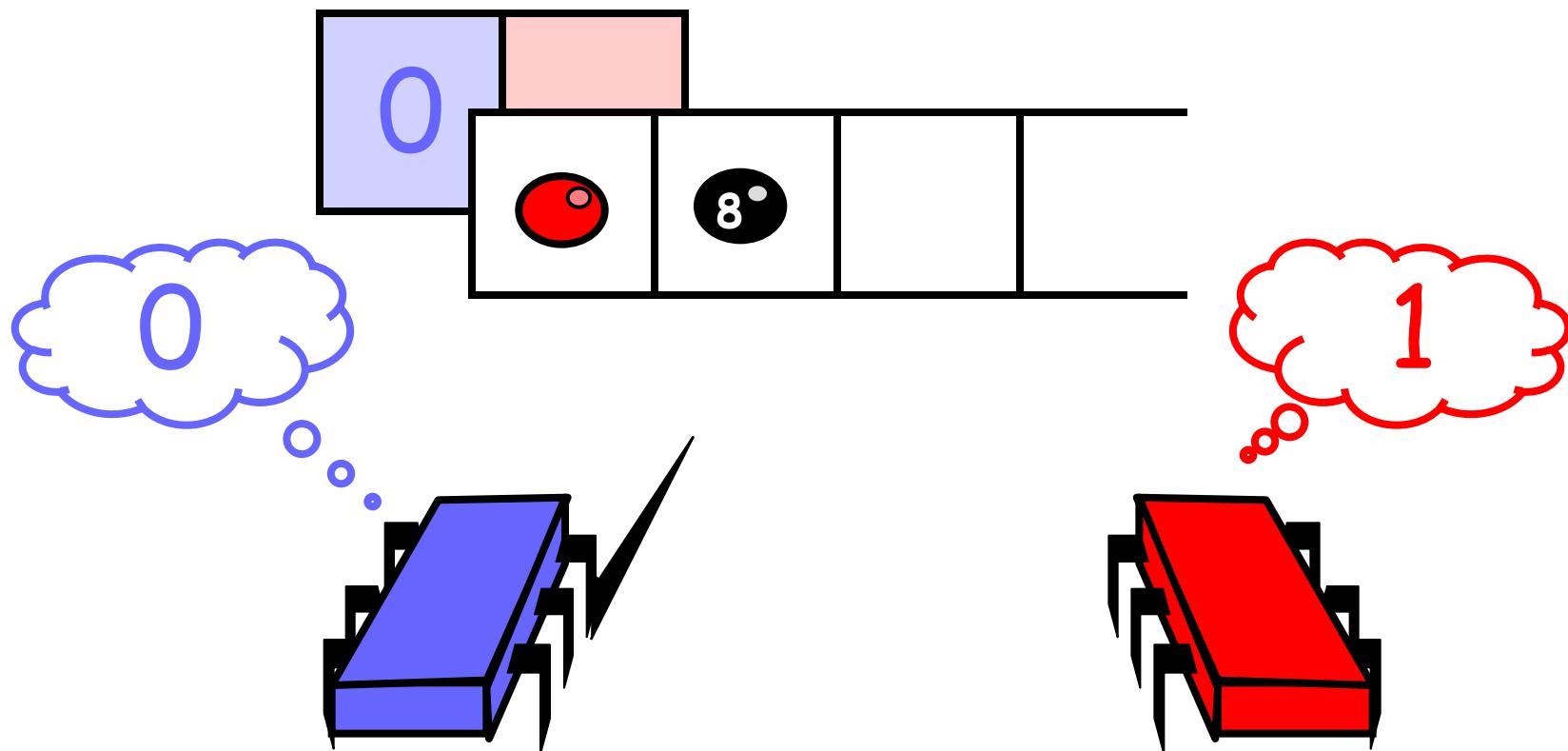
FIFO Queue  
with red and  
black balls

# Protocol: Write Value to Array

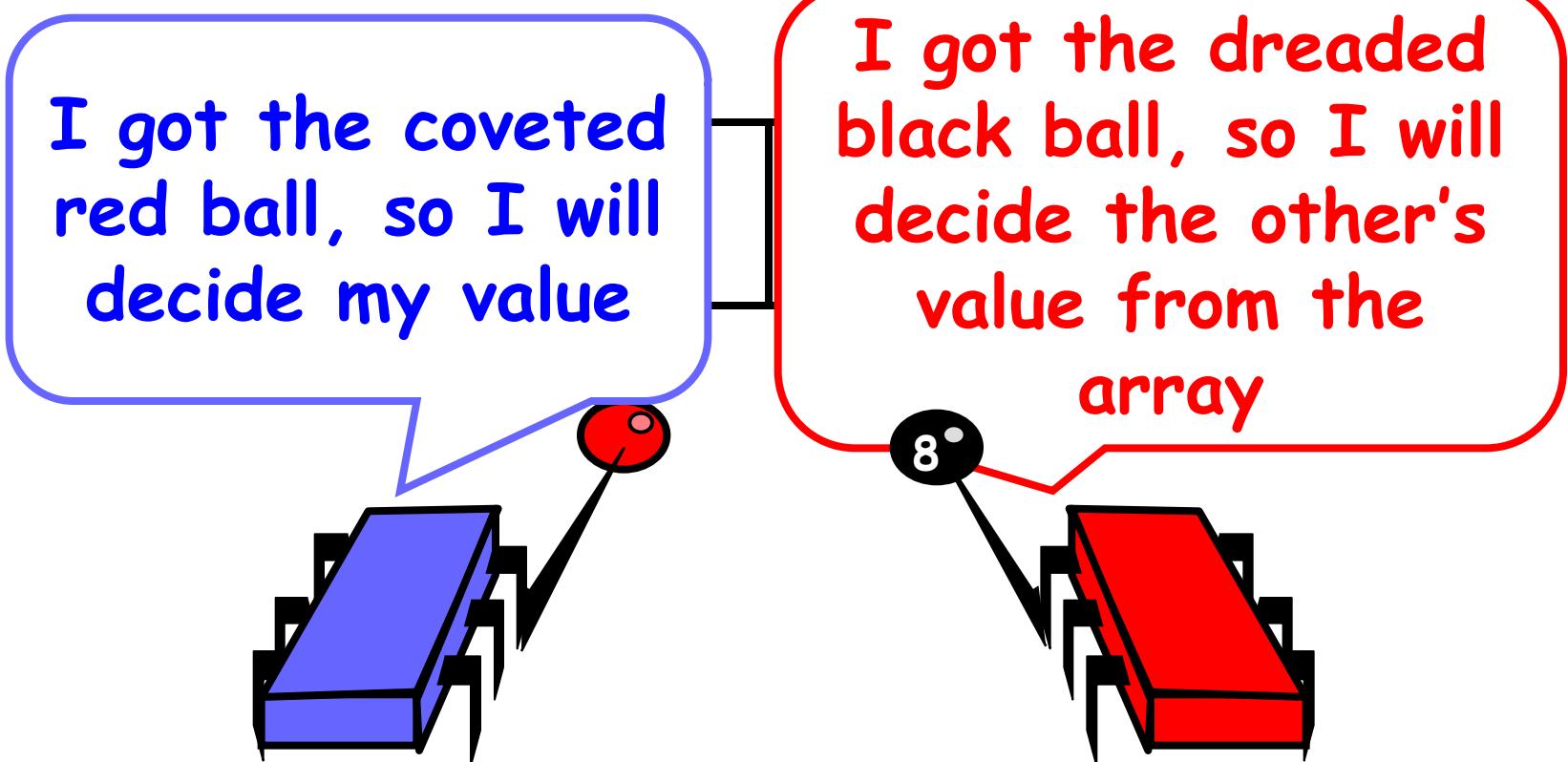
2-Threads attempt to enqueue a value at the same time



# Protocol: Take Next Item from Queue



# Protocol: Take Next Item from Queue

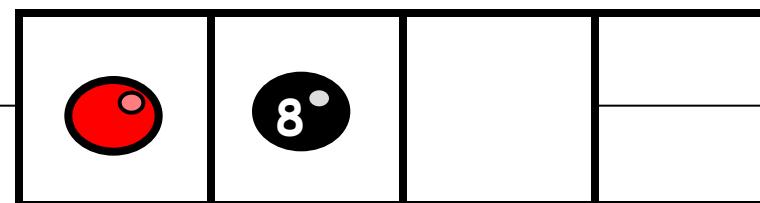


# Consensus Using FIFO Queue

```
public class QueueConsensus  
    extends ConsensusProtocol {  
    private Queue queue;  
    public QueueConsensus() {  
        queue = new Queue();  
        queue.enq(Ball.RED);  
        queue.enq(Ball.BLACK);  
    }  
    ...  
}
```

# Initialize Queue

```
public class QueueConsensus  
    extends ConsensusProtocol {  
    private Queue queue;  
    public QueueConsensus() {  
        this.queue = new Queue();  
        this.queue.enq(Ball.RED);  
        this.queue.enq(Ball.BLACK);  
    }  
    ...  
}
```



# Who Won?

```
public class QueueConsensus
    extends ConsensusProtocol {
    private Queue queue;
    ...
    public decide(object value) {
        propose(value);
        Ball ball = this.queue.deq();
        if (ball == Ball.RED)
            return proposed[i];
        else
            return proposed[1-i];
    }
}
```

# Who Won?

```
public class QueueConsensus  
    extends ConsensusProtocol {  
    private Queue queue;  
  
    ...  
    public decide(object value) {  
        propose(value);  
        Ball ball = this.queue.deq();  
        if (ball == Ball.RED)  
            return proposed[i];  
        else  
            return proposed[1-ij];  
    }  
}
```

Race to dequeue  
first queue item

# Who Won?

```
public class QueueConsensus  
    extends ConsensusProtocol {  
    private Queue queue;  
  
    ...  
    public decide(object value) {  
        propose(value);  
        Ball ball = this.queue.deq();  
        if (ball == Ball.RED)  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```

i = ThreadID.get();  
I win if I was  
first

# Who Won?

```
public class QueueConsensus  
    extends ConsensusProtocol {  
    private Queue queue;  
    ...  
    public decide(object  
        propose(value);  
        Ball ball = this.queue.deq();  
        if (ball == Ball.RED)  
            return proposed[i];  
        else  
            return proposed[1-i];  
    }  
}
```

*Other thread wins if I was second*

# FIFO Queues

- Although FIFO queues solve two-thread consensus, they cannot solve 3-thread consensus.

# Theorem 5.4.1

- FIFO queues have consensus number 2.



# Read-modify-write operations

- Many synchronization operations can be described as read-modify-write (RMW) operations
- In object form: read-modify-write registers

# RMW Methods

- A method is an RMW for the function set  $F$  if it atomically replaces register value  $v$  with  $f(v)$  for some  $f \in F$  and returns  $v$

# RMW Methods

- From `java.util.concurrent`:
  - `getAndSet(x)`:
    - Replaces current value with  $x$  and returns prior value
    - $f_v(x) = v$
  - `getAndIncrement()`
    - Atomically adds 1 to the current value and returns the old value
    - $f_v(x) = v + 1$

# RMW Methods

## □ getAndAdd(k)

- Atomically adds  $k$  to the current value and returns the prior value
- $f_k(x) = x + k$

## □ get()

- Returns the register's value

# The Exception

- compareAndSet()
  - Takes 2 values – expected value  $e$  and update value  $u$
  - If value is equal to  $e$ , it replaces it with  $u$  otherwise it remains unchanged
  - Returns a Boolean value to indicate whether value was changed

# Read-Modify-Write

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndMumble() {  
            int prior = this.value;  
            this.value = mumble(this.value);  
            return prior;  
        }  
}
```

# Read-Modify-Write

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndMumble() {  
            int prior = this.value;  
            this.value = mumble(this.value);  
            return prior;  
        }  
}
```

**Return prior value**

# Read-Modify-Write

```
public abstract class RMWRegister {  
    private int value;  
  
    public int synchronized  
        getAndMumble() {  
            int prior = this.value;  
            this.value = mumble(this.value);  
            return prior;  
        }  
}
```

**Apply function to current value**

# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                      int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }
```

# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                      int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }
```

If value is what was expected, ...

# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                      int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }
```

... replace it

# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                      int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }
```

Report success

# compareAndSet

```
public abstract class RMWRegister {  
    private int value;  
    public boolean synchronized  
        compareAndSet(int expected,  
                      int update) {  
        int prior = this.value;  
        if (this.value==expected) {  
            this.value = update; return true;  
        }  
        return false;  
    } ... }
```

Otherwise report failure

# Definition

- A RMW method
  - With function `mumble(x)`
  - is non-trivial if there exists a value  $v$
  - Such that  $v \neq \text{mumble}(v)$

# Par Example

- `Identity(x) = x`
  - is trivial
- `getAndIncrement(x) = x+1`
  - is non-trivial



# Theorem

- Any non-trivial RMW object has consensus number of 2

# Reminder

- Subclasses of **consensus** have
  - **propose(x)** method
    - which just stores x into **proposed[i]**
    - built-in method
  - **decide(object value)** method
    - which determines winning value
    - customized, class-specific method

# Implementation

```
public class RMWConsensus
    extends ConsensusProtocol {
private RMWRegister r = new
RMWRegister(v);

public Object decide(Object value) {
    propose(value);
    if (r.getAndMumble() == v)
        return proposed[i];
    else
        return proposed[j];
}}
```

# Implementation

```
public class RMWConsensus
    extends ConsensusProtocol {
private RMWRegister r = new
RMWRegister(v);
public Object decide(Object value) {
    propose(value);
    if (r.getAndMumble() == v)
        return proposed[i];
    else
        return proposed[j];
}}
```

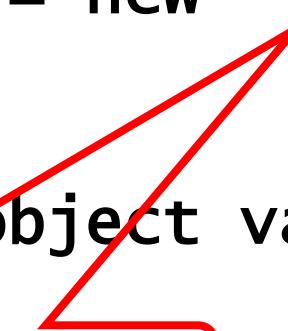
Initialized to v

# Implementation

```
public class RMWConsensus
    extends Consensus {
private RMWRegister r = new RMWRegister(v);

public Object decide(Object value) {
    propose(value);
    if (r.getAndMumble() == v)
        return proposed[1];
    else
        return proposed[j];
}}
```

Am I first?



# Implementation

```
public class RMWConsensus
    extends ConsensusProtocol {
private RMWRegister r = new
RMWRegister(v);
public Object decide(Object value) {
    propose(value);
    if (r.getAndMumble() == v)
        return proposed[i];
    else
        return proposed[j];
}}
```

Yes, return my  
input

# Implementation

```
public class RMWConsensus
    extends ConsensusProtocol {
private RMWRegister r = new
RMWRegister(v);

public Object decide(Object value) {
    propose(value);
    if (r.getAndMumble() == v)
        return proposed[i];
    else
        return proposed[j];
}}
```

No, return  
other's input



# Common2 RMW Operations

- Let  $F$  be a set of functions such that for all  $f_i$  and  $f_j$ , either
  - Commute:  $f_i(f_j(v)) = f_j(f_i(v))$
  - Overwrite:  $f_i(f_j(v)) = f_i(v)$
- Claim: Any set of RMW objects that commutes or overwrites has consensus number exactly 2



# Examples

- `getAndSet()`
  - Overwrite
- `getAndAdd()`
  - Commute
- `getAndIncrement()`
  - Commute



# Common2 RMW Registers

- Theorem:
  - Any RMW register in Common2 has consensus number of 2.

# compareAndSet()

- A register providing compareAndSet() and get() methods has an infinite consensus number

# Implementation

```
public class CASConsensus
    extends ConsensusProtocol {
private final int FIRST = -1;
private AtomicInteger r = new
    AtomicInteger(FIRST);

public Object decide(object value) {
    propose(value);
    int i = ThreadID.get();
    if (r.compareAndSet(FIRST, i))
        return proposed[i];
    else
        return proposed[j];
}}
```

# Implementation

```
public class CASConsensus
    extends ConsensusProtocol {
private final int FIRST = -1;
private AtomicInteger r = new
    AtomicInteger(FIRST);
```

Use Atomic  
Register

```
public Object decide(object value) {
    propose(value);
    int i = ThreadID.get();
    if (r.compareAndSet(FIRST, i))
        return proposed[i];
    else
        return proposed[j];
}}
```

# Implementation

```
public class CASConsensus
    extends ConsensusProtocol {
private final int FIRST = -1;
private AtomicInteger r = new
    AtomicInteger(FIRST);

public Object decide(object value) {
    propose(value);
    int i = ThreadID.get();
    if (r.compareAndSet(FIRST, i))
        return proposed[i];
    else
        return proposed[j];
}}
```

**Add value to proposed array**

# Implementation

```
public class CASConsensus
    extends ConsensusProtocol {
private final int FIRST = -1;
private AtomicInteger r = new
    AtomicInteger(FIRST);

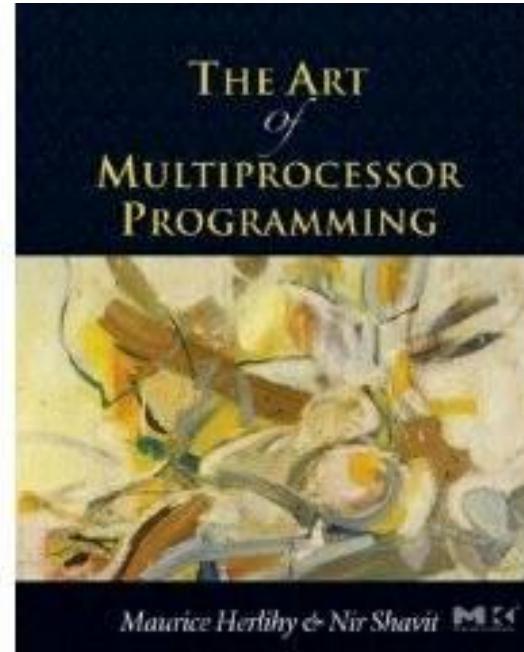
public Object decide(object value) {
    propose(value);
    int i = ThreadID.get();
    if (r.compareAndSet(FIRST, i))
        return proposed[i];
    else
        return proposed[j];
}}
```

If I am the first  
thread to access  
the register

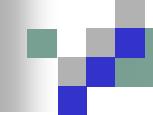
COS 226

Chapter 7  
Spin Locks and Contention

# Acknowledgement

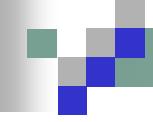


- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



# Concurrency issues

- Memory contention:
  - Not all processors can access the same memory at the same time and if they try they will have to queue
- Contention for communication medium:
  - If everyone wants to communicate at the same time, some of them will have to wait
- Communication latency:
  - It takes more time for a processor to communicate with memory or with another processor.



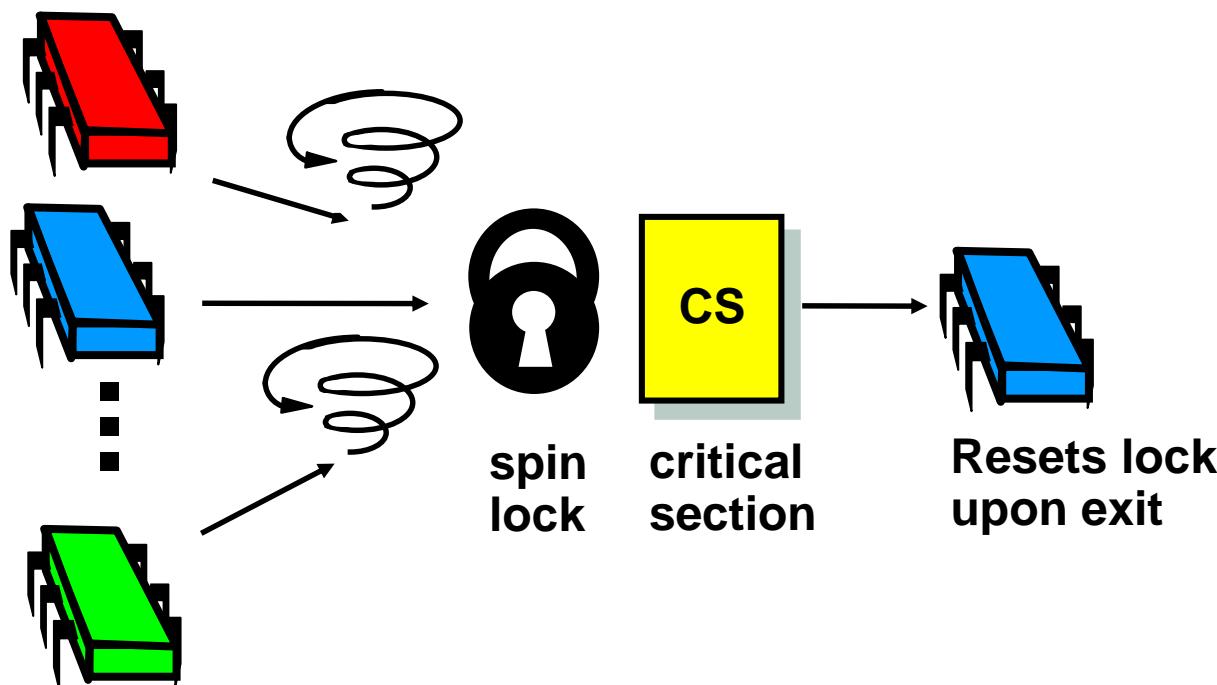
# New goals

- Think of performance, not just correctness and progress
- Understand the underlying architecture
- Understand how the architecture affects performance
- Start with Mutual Exclusion

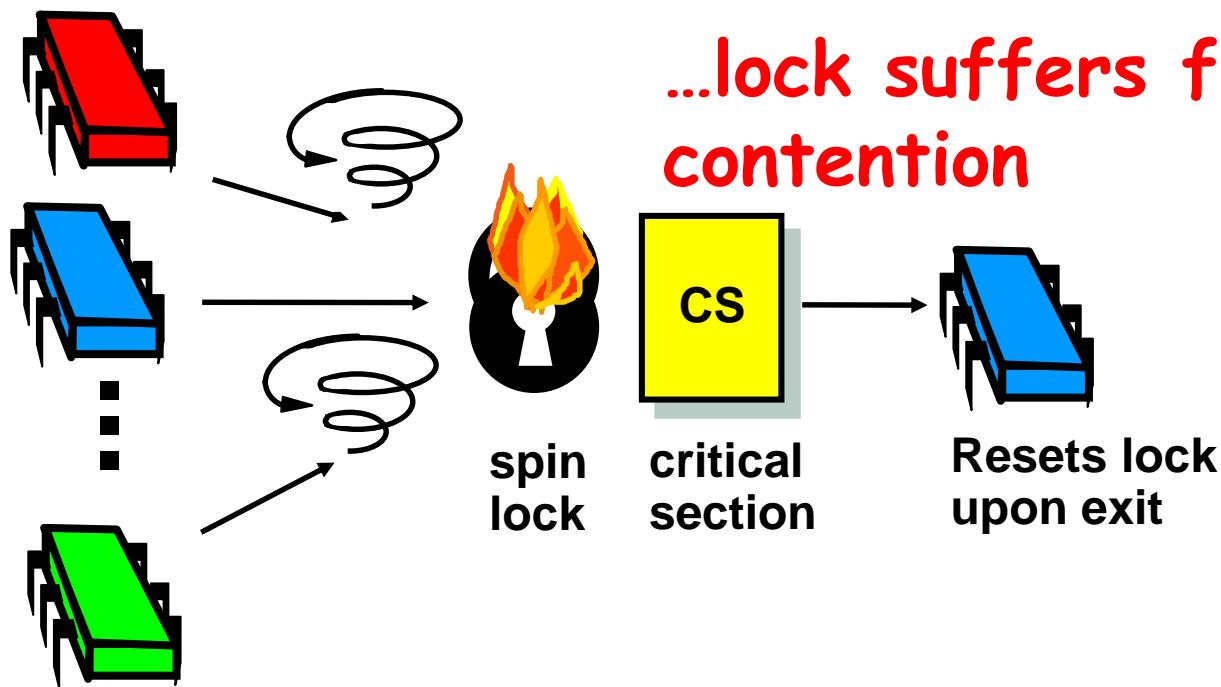
# What should you do if you can't get a lock?

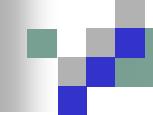
- Keep trying
    - “spin” or “busy-wait”
    - Good if delays are short
  - Give up the processor
    - Suspend yourself and ask the schedule to create another thread on your processor
    - Good if delays are long
    - Always good on uniprocessor
- our focus*

# Basic Spin-Lock



# Basic Spin-Lock





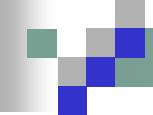
# Contention

- Contention:
  - When multiple threads try to acquire a lock at the same time
- High contention:
  - There are many such threads
- Low contention:
  - The opposite

# Welcome to the real world

- Java Lock interface
  - `java.util.concurrent.locks` package

```
Lock mutex = new LockImpl (...);  
...  
mutex.lock();  
try {  
    ...  
} finally {  
    mutex.unlock();  
}
```

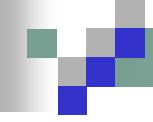


# Why don't we just use the Filter or Bakery Locks?

- The principal drawback is the need to read and write  $n$  distinct locations where  $n$  is the number of concurrent threads
- This means that the locks require space linear in  $n$

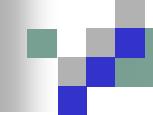
# What about the Peterson lock?

```
class Peterson implements Lock {  
    private boolean[] flag = new boolean[2];  
    private int victim;  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;  
        victim = i;  
        while (flag[j] && victim == i) {};  
    }  
}
```



# Peterson lock?

- It is not our logic that fails, but our assumptions about the real world
- We assumed that read and write operations are atomic
- Our proof relied on the assumption that any two memory accesses by the same thread, even to different variables, take place in program order

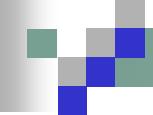


# Why does it not take place in program order?

- Modern multiprocessors do not guarantee program order
- Due to:
  - Compilers
    - reorder instructions to enhance performance
  - Multiprocessor hardware itself
    - writes to multiprocessor memory do not necessarily take effect when they are issued
    - writes to shared memory are buffered and written to memory only when needed

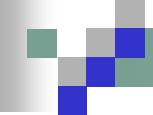
# What about the Peterson lock?

```
class Peterson implements Lock {  
    private boolean[] flag = new boolean[2];  
    private int victim;  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;           Important that these  
        victim = i;              steps take place in  
        while (flag[j] && victim == i) {};  
    }  
}
```



# How can one fix this?

- Memory barriers (or memory fences) can be used to force outstanding operations to take effect
- It is the programmer's responsibility to know when to insert a memory barrier
- However, memory barriers are expensive



# Memory barriers

- Synchronization instructions such as `getAndSet()` or `compareAndSet()` often include memory barriers
- As do reads and writes to **volatile** fields

# 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

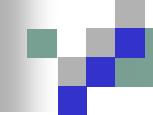
```
AtomicBoolean lock  
= new AtomicBoolean(false)  
...  
boolean prior = lock.getAndSet(true)
```

# 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

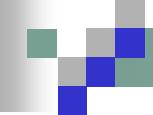
```
class TASLock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
  
    void unlock() {  
        state.set(false);  
    }  
}
```

*Keep trying until  
lock acquired*

# Test-and-set Lock

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

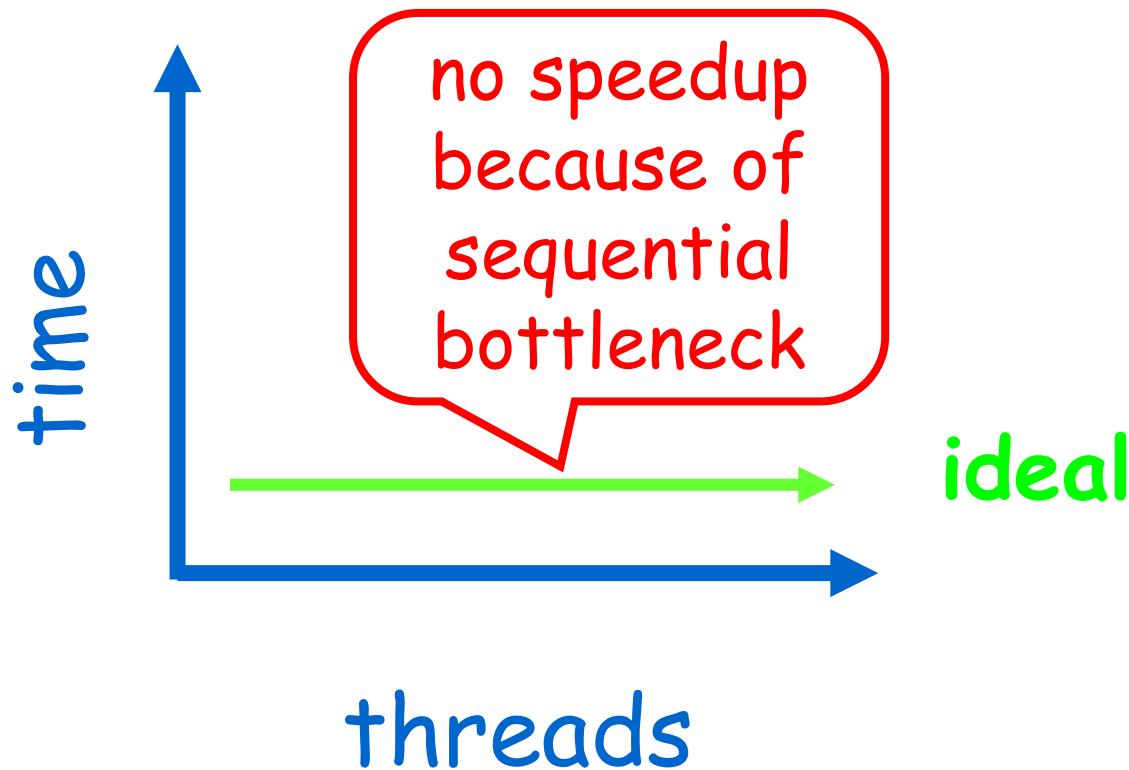
*Release lock by  
resetting state to  
false*



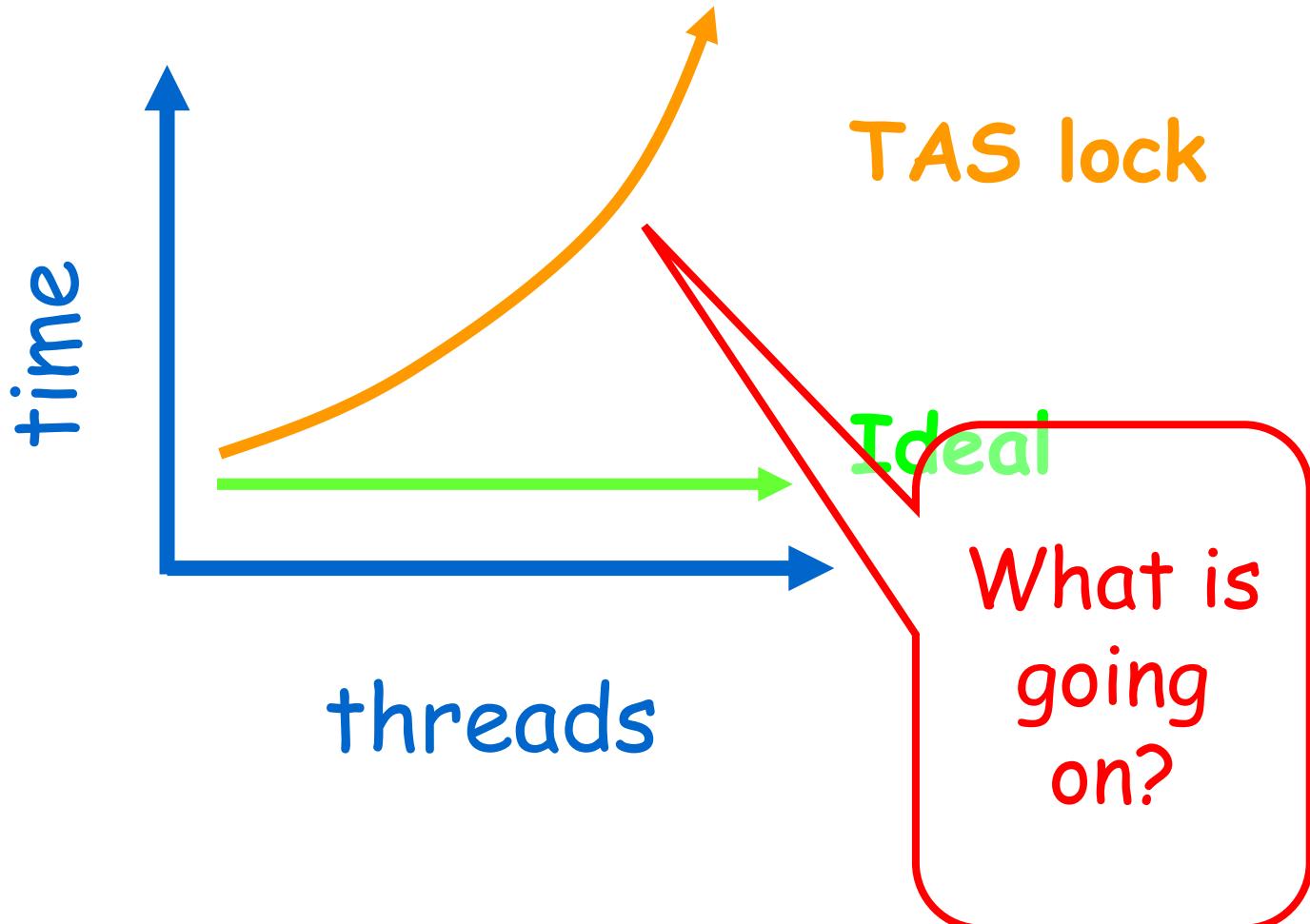
# Performance

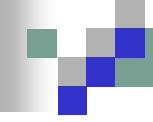
- Experiment
  - n threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?

# Graph



# Mystery #1

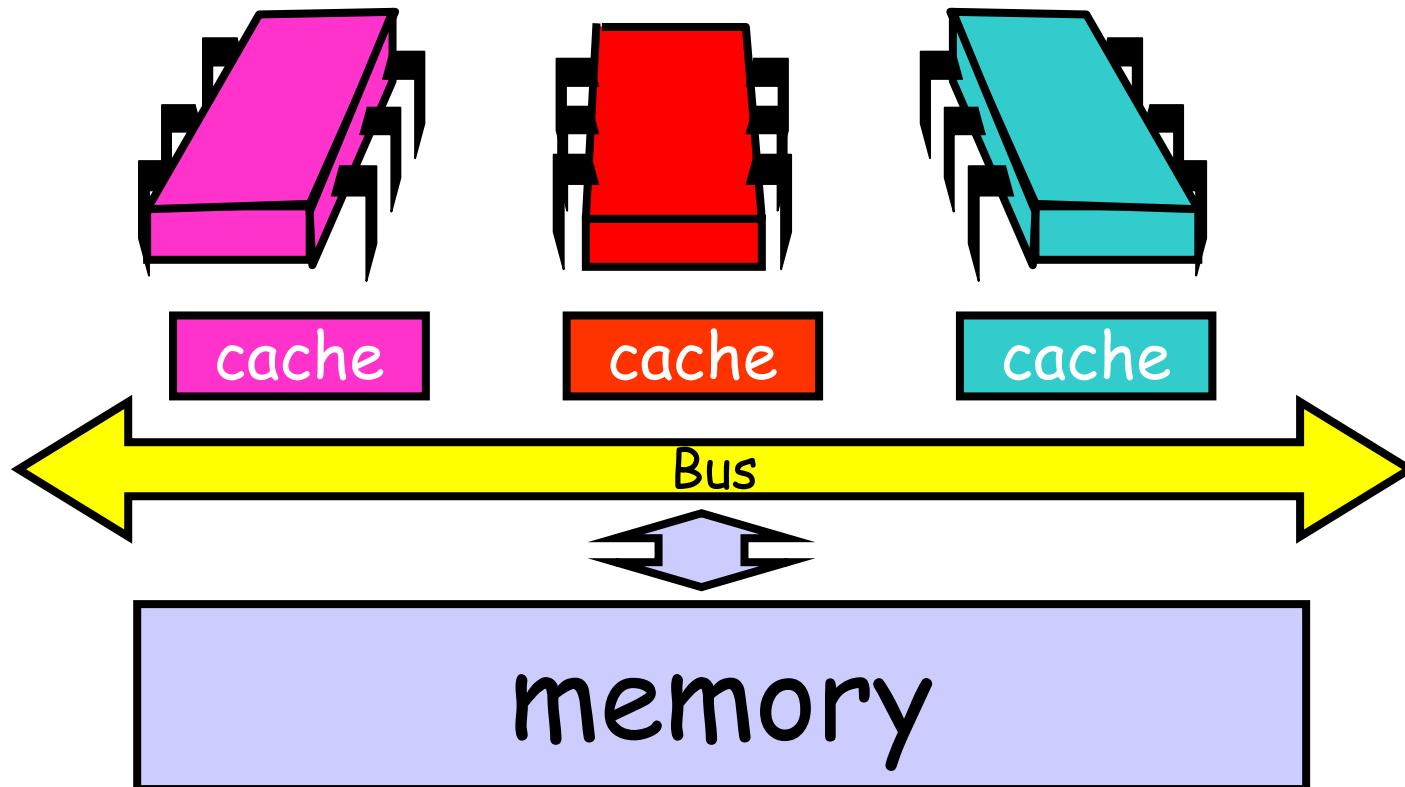




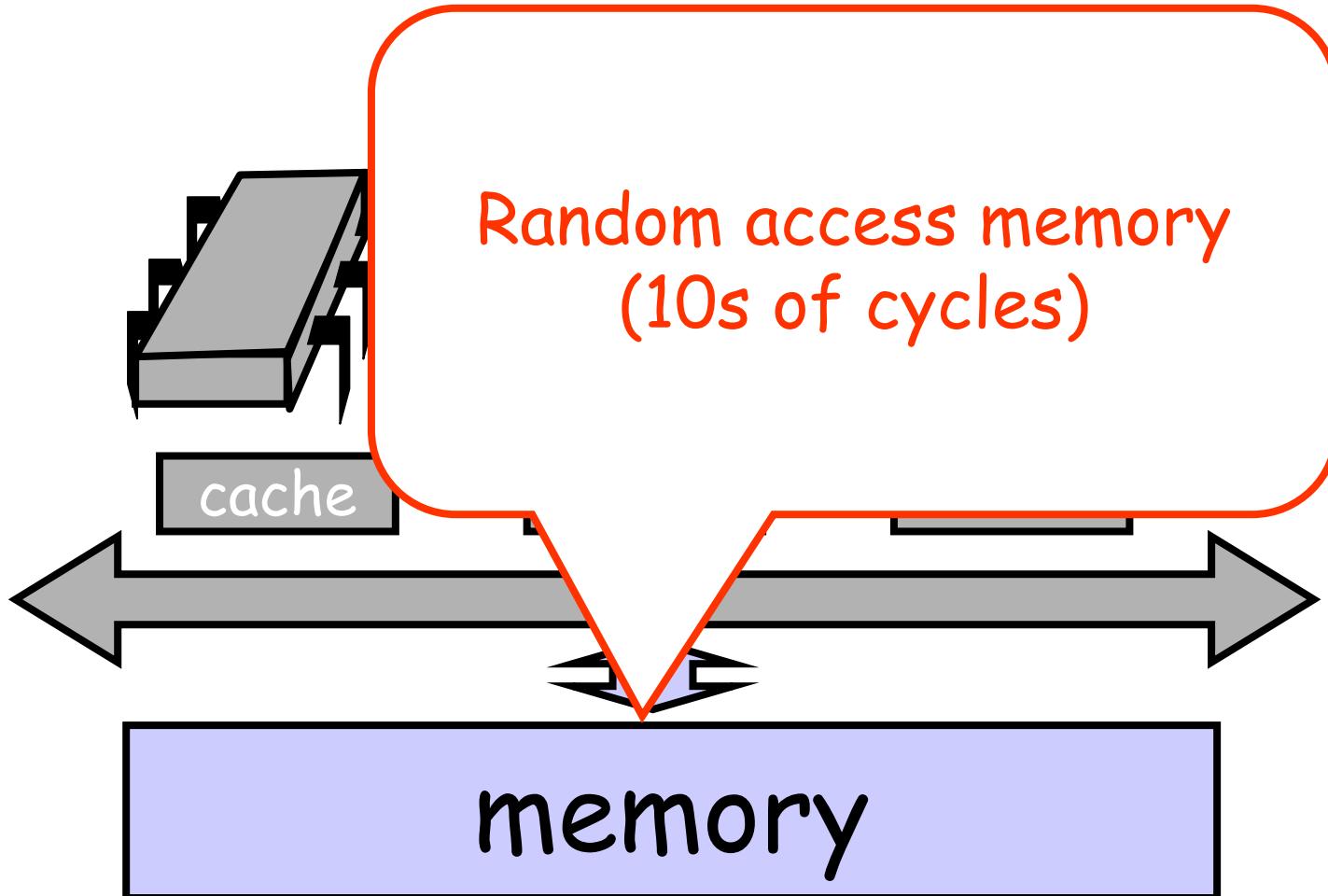
# Questions

- Why is TAS so bad (so much worse than ideal)?

# Bus-Based Architectures



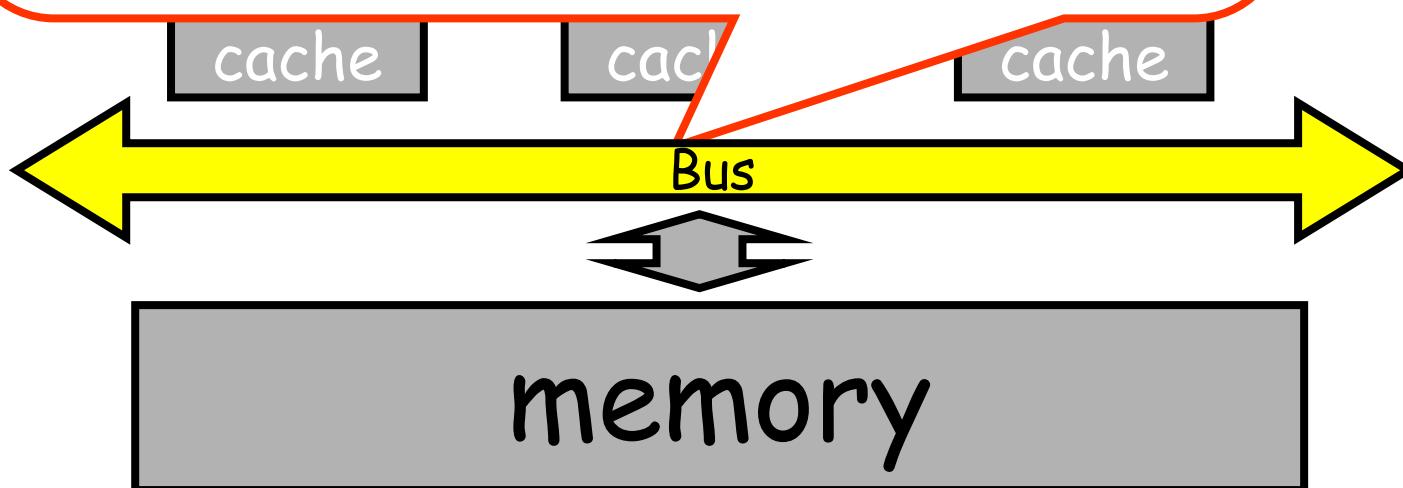
# Bus-Based Architectures



# Bus-Based Architectures

## Shared Bus

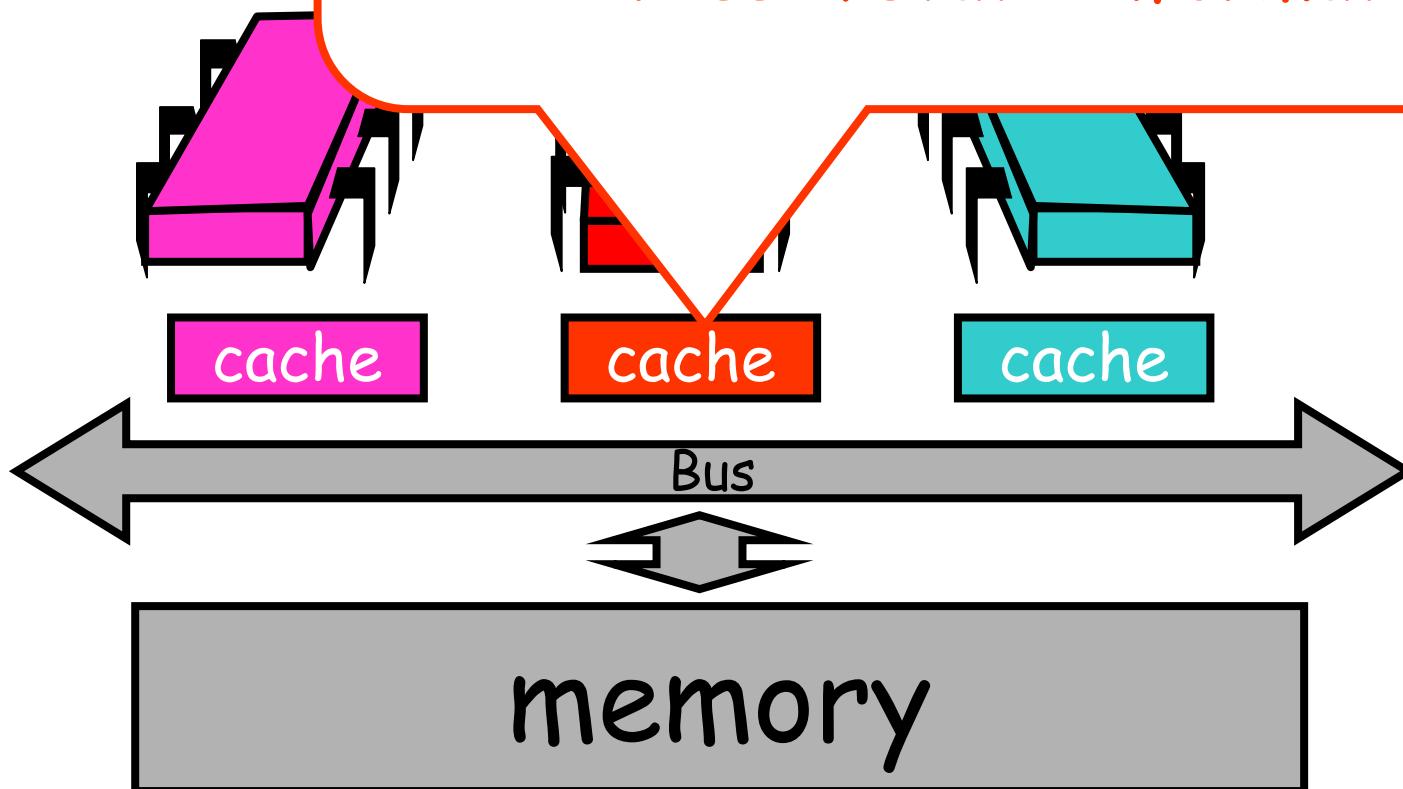
- Broadcast medium
- One broadcaster at a time
- Processors and memory all "snoop"

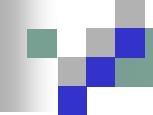


# Bus-Based

## Per-Processor Caches

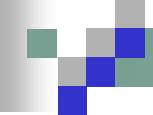
- Small
- Fast: 1 or 2 cycles
- Address & state information





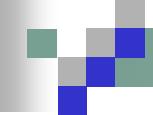
# Jargon Watch

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



# Jargon Watch

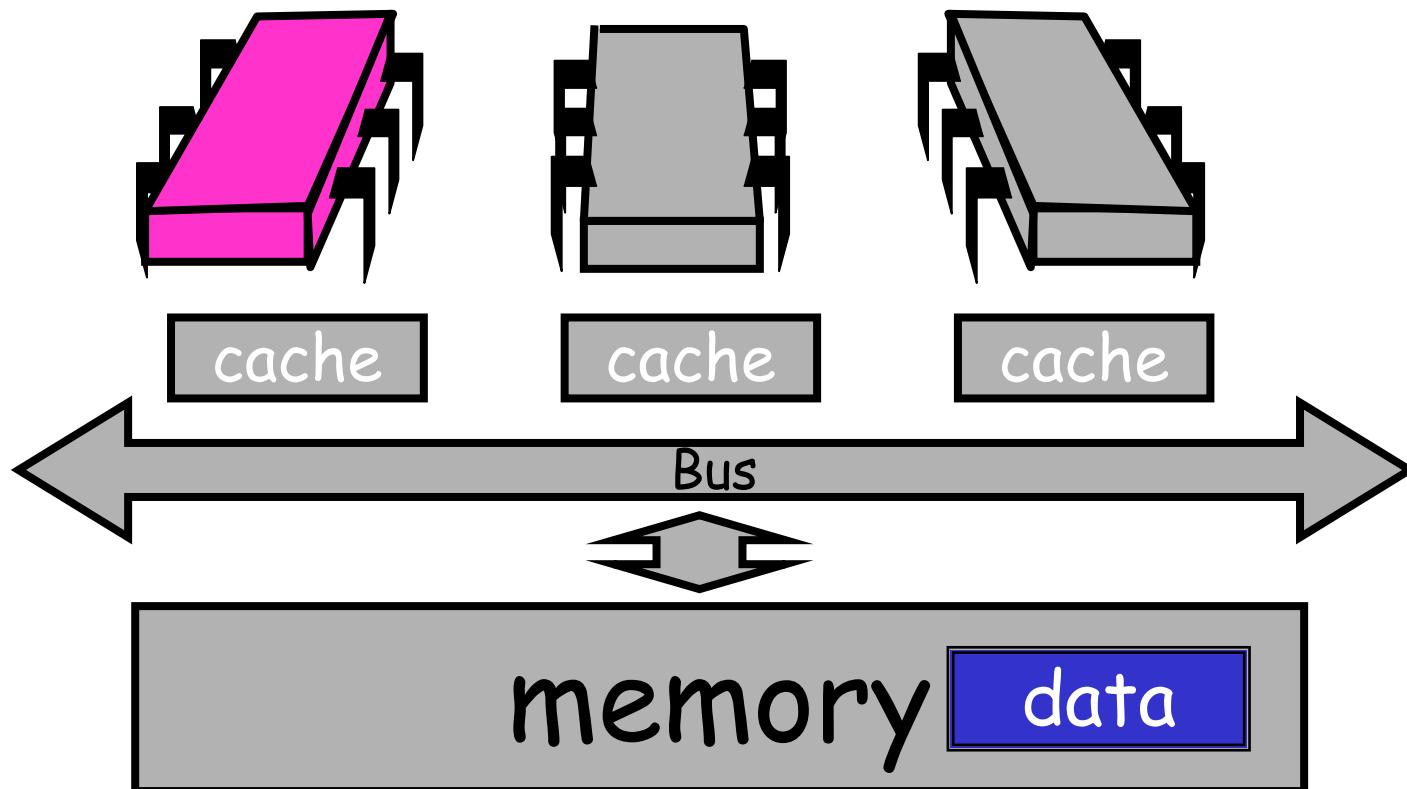
- Cache hit
  - “I found what I wanted in my cache”
  - Good Thing™
- Cache miss
  - “I had to shlep all the way to memory for that data”
  - Bad Thing™



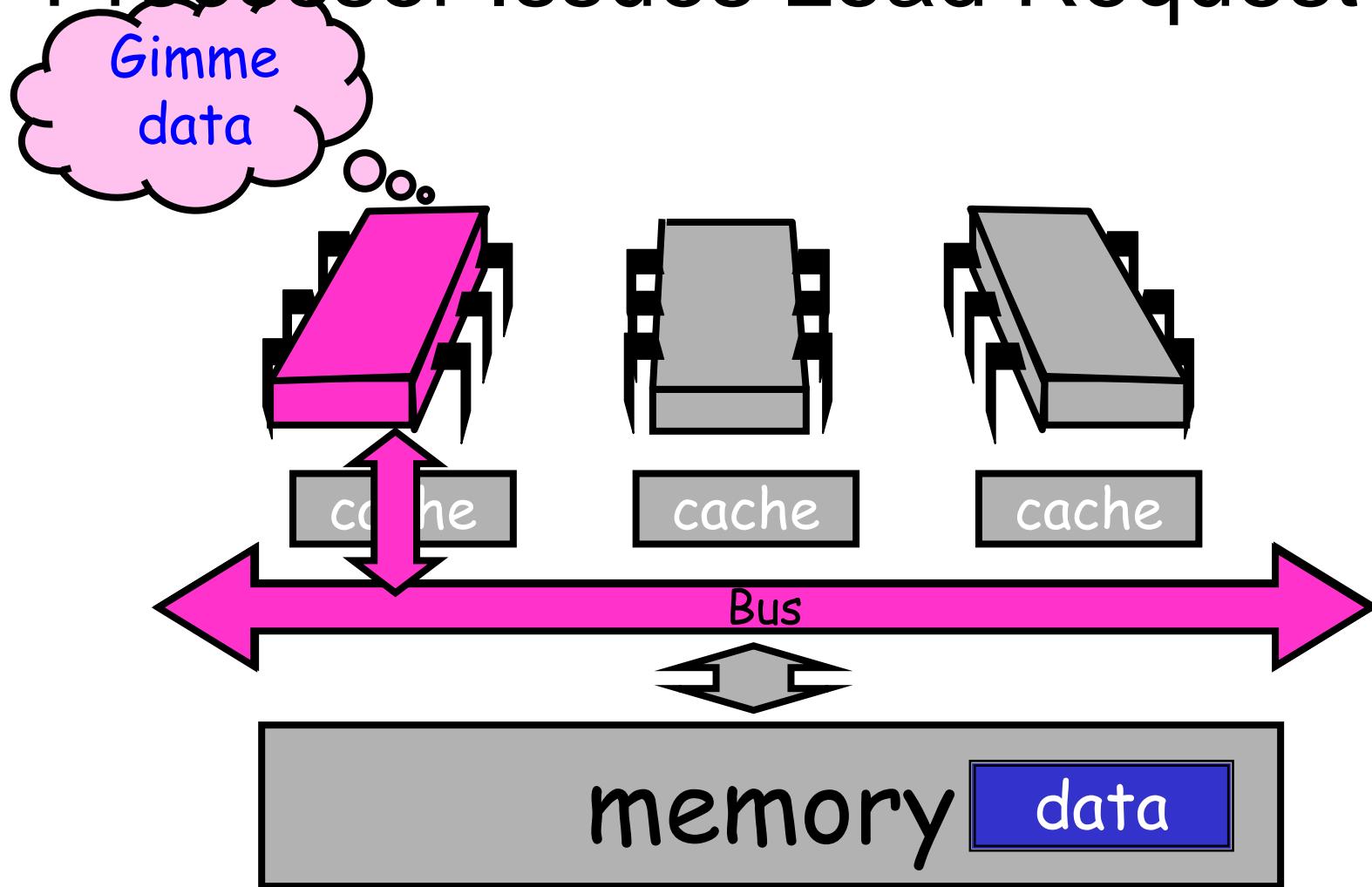
# Cave Canem

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

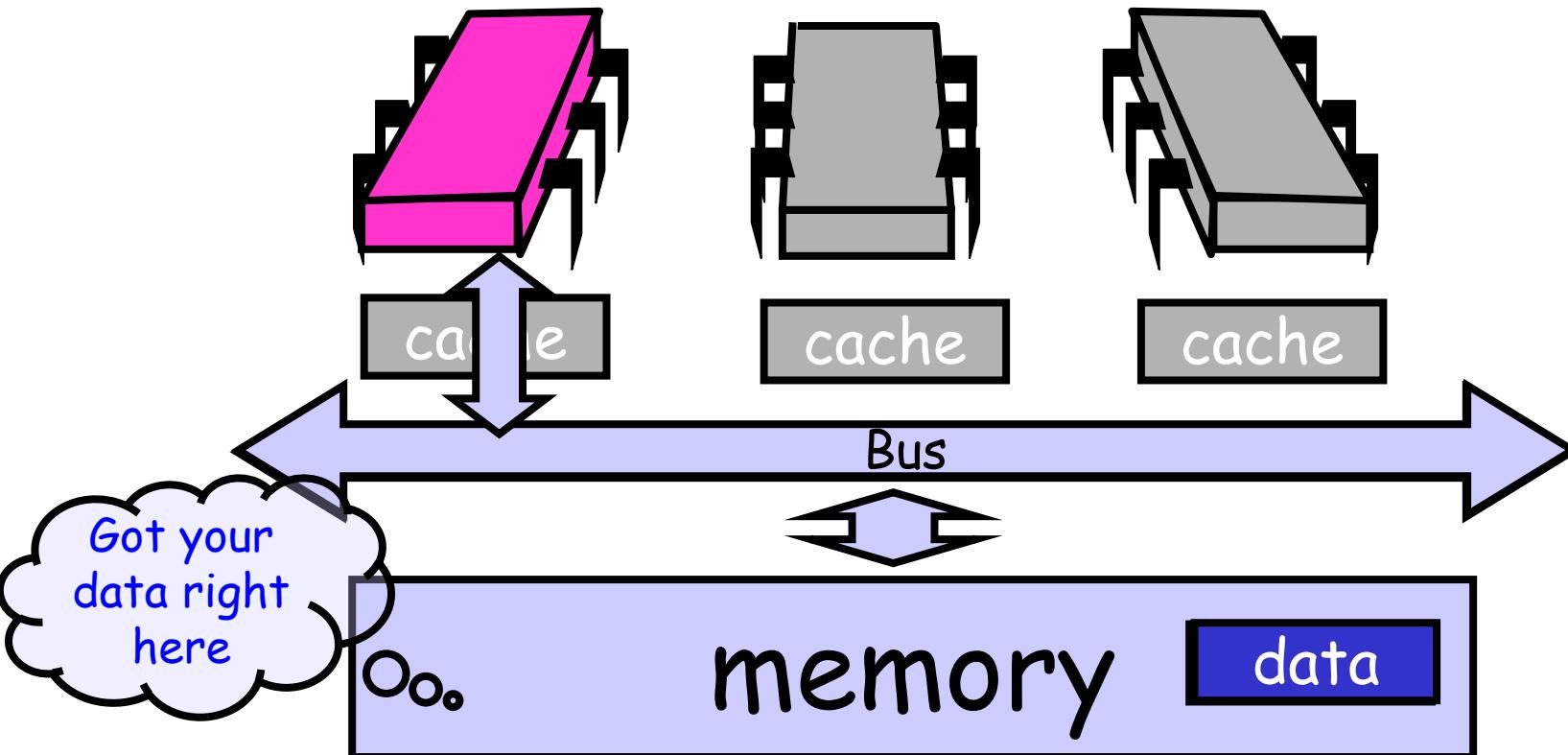
# Processor Issues Load Request



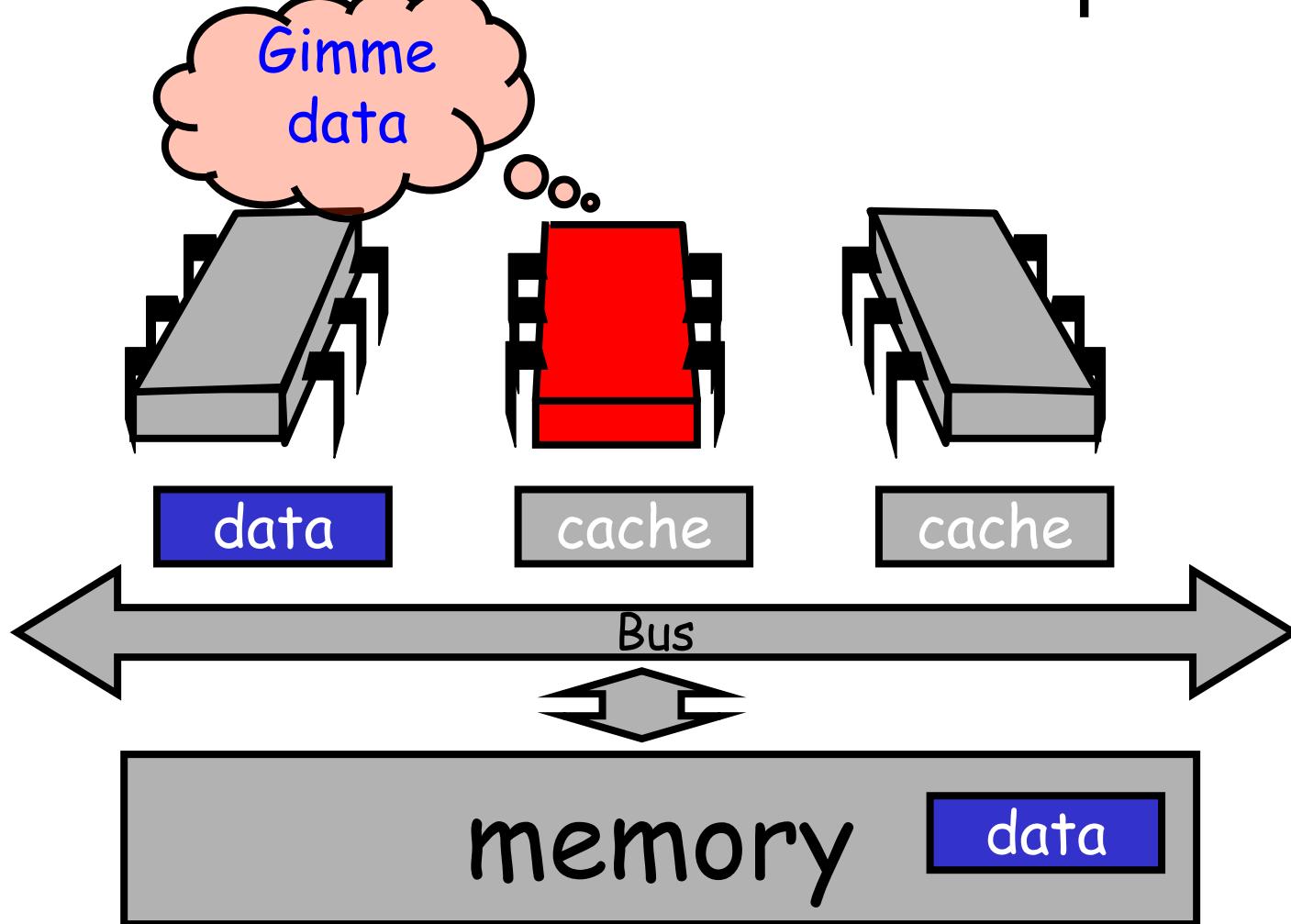
# Processor Issues Load Request



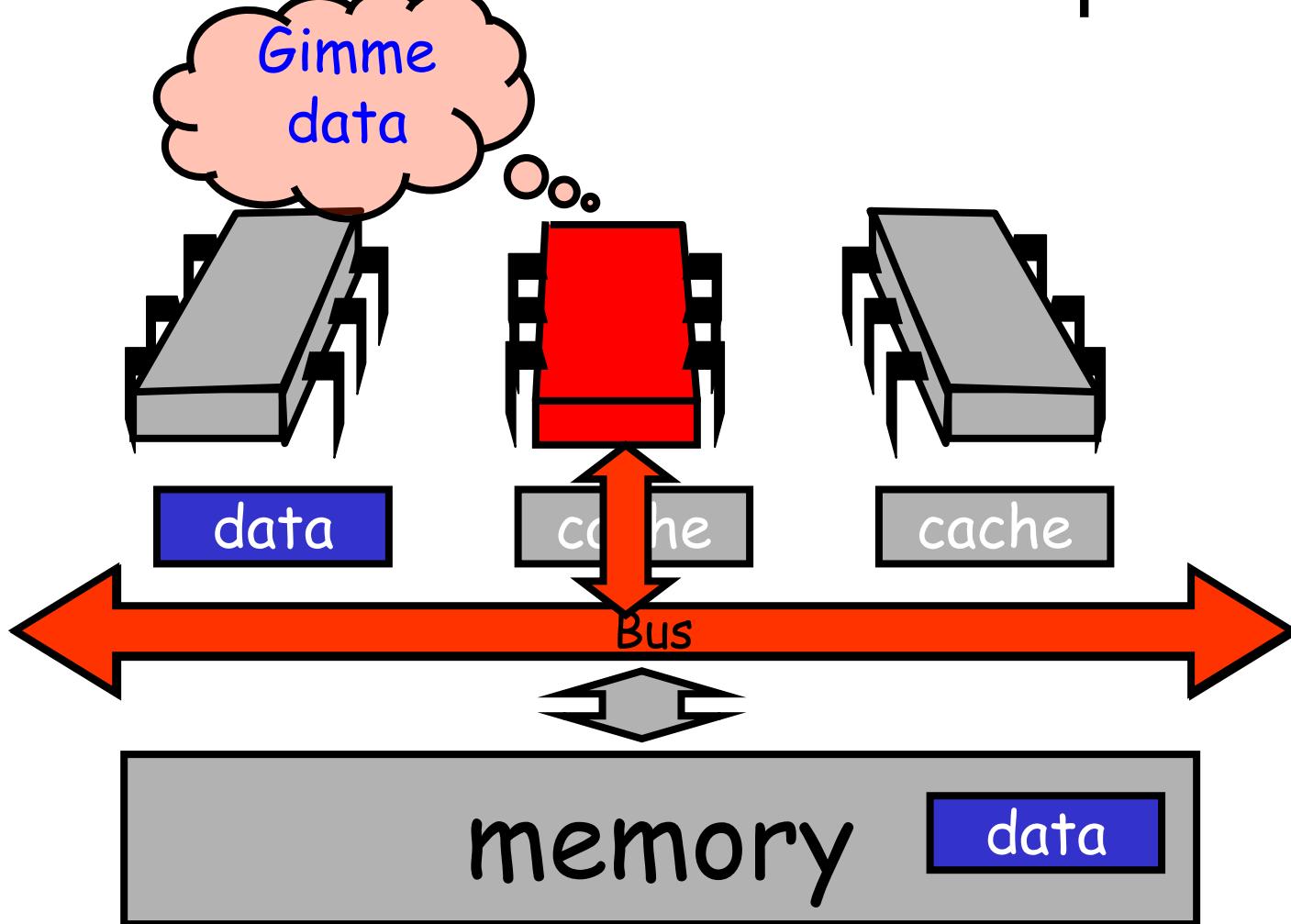
# Memory Responds



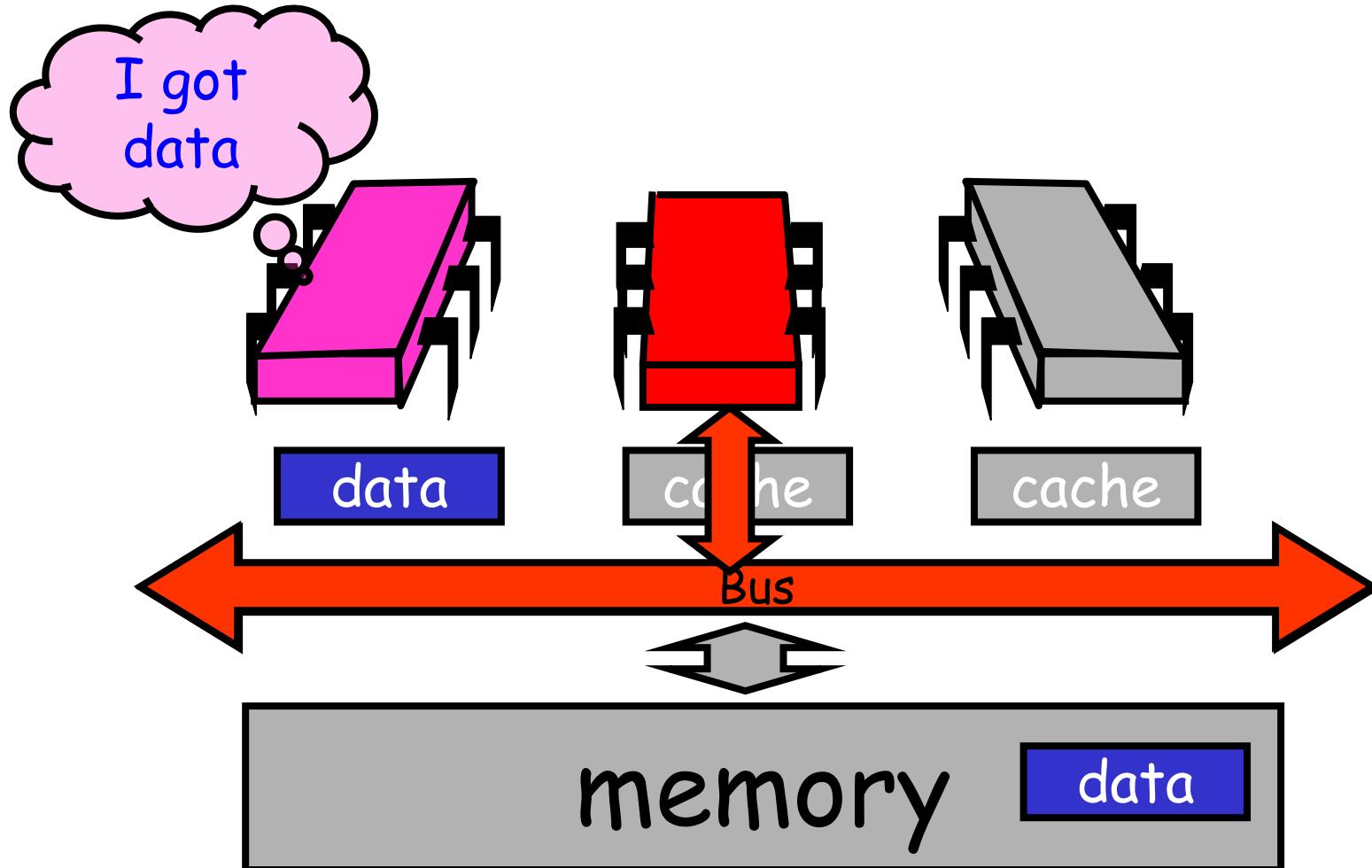
# Processor Issues Load Request



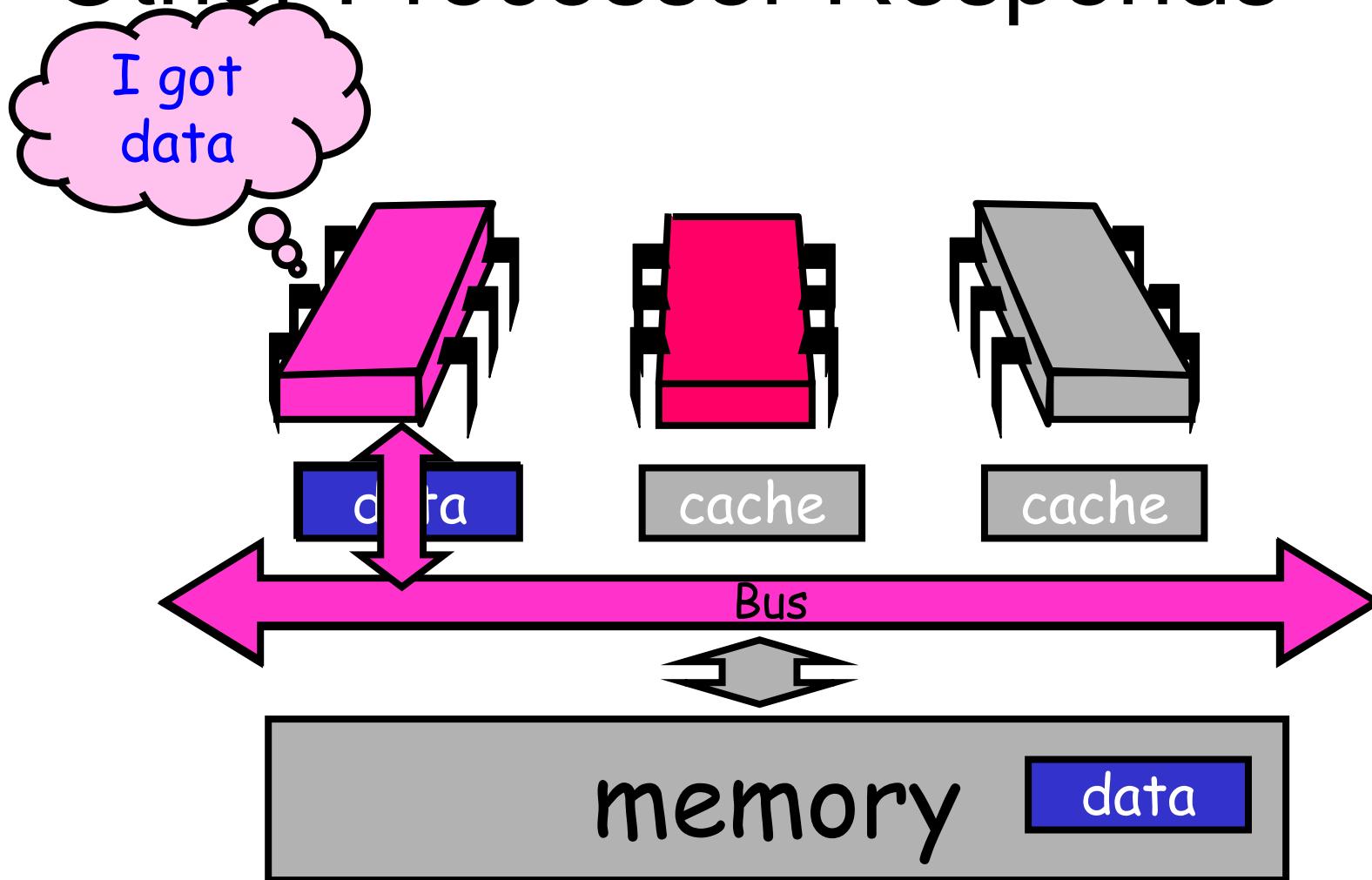
# Processor Issues Load Request



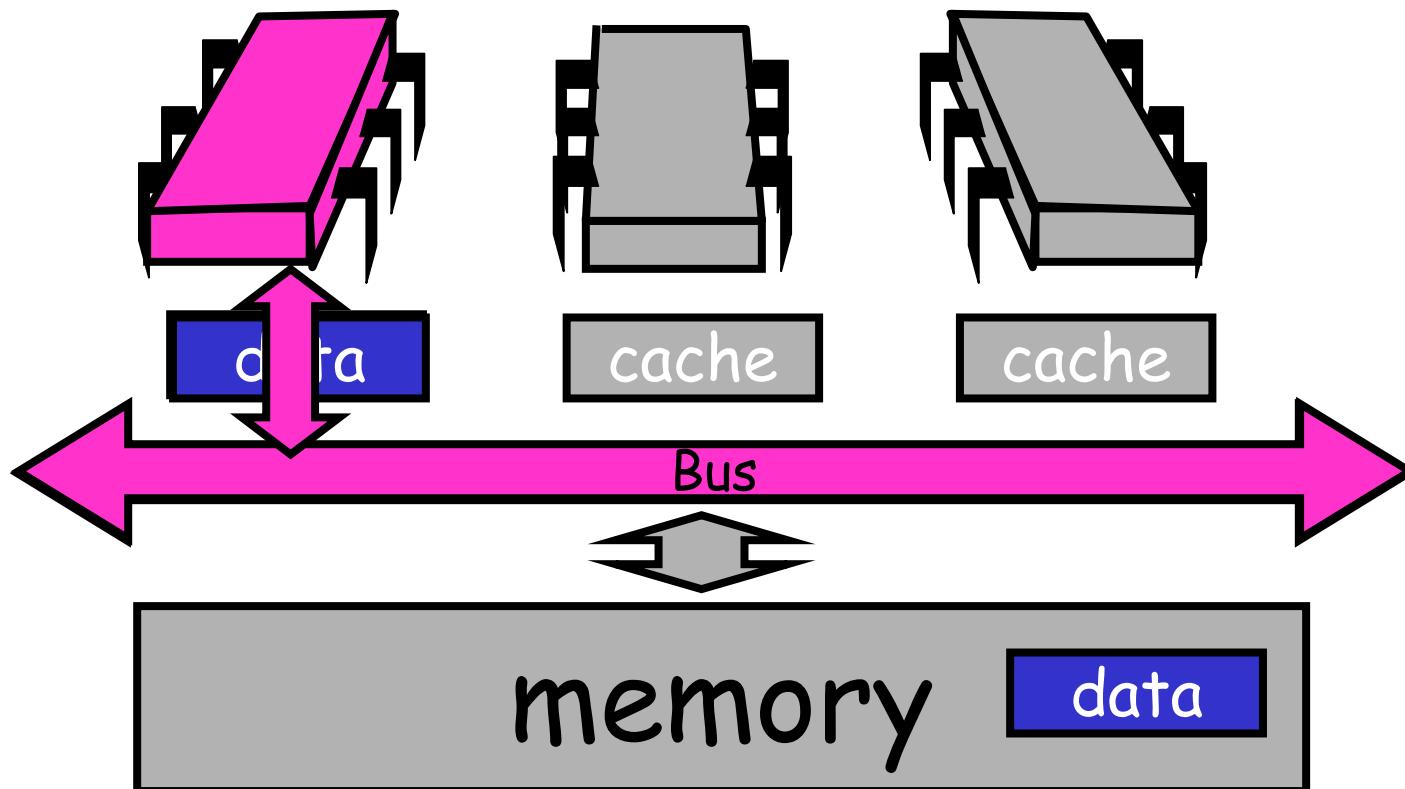
# Processor Issues Load Request



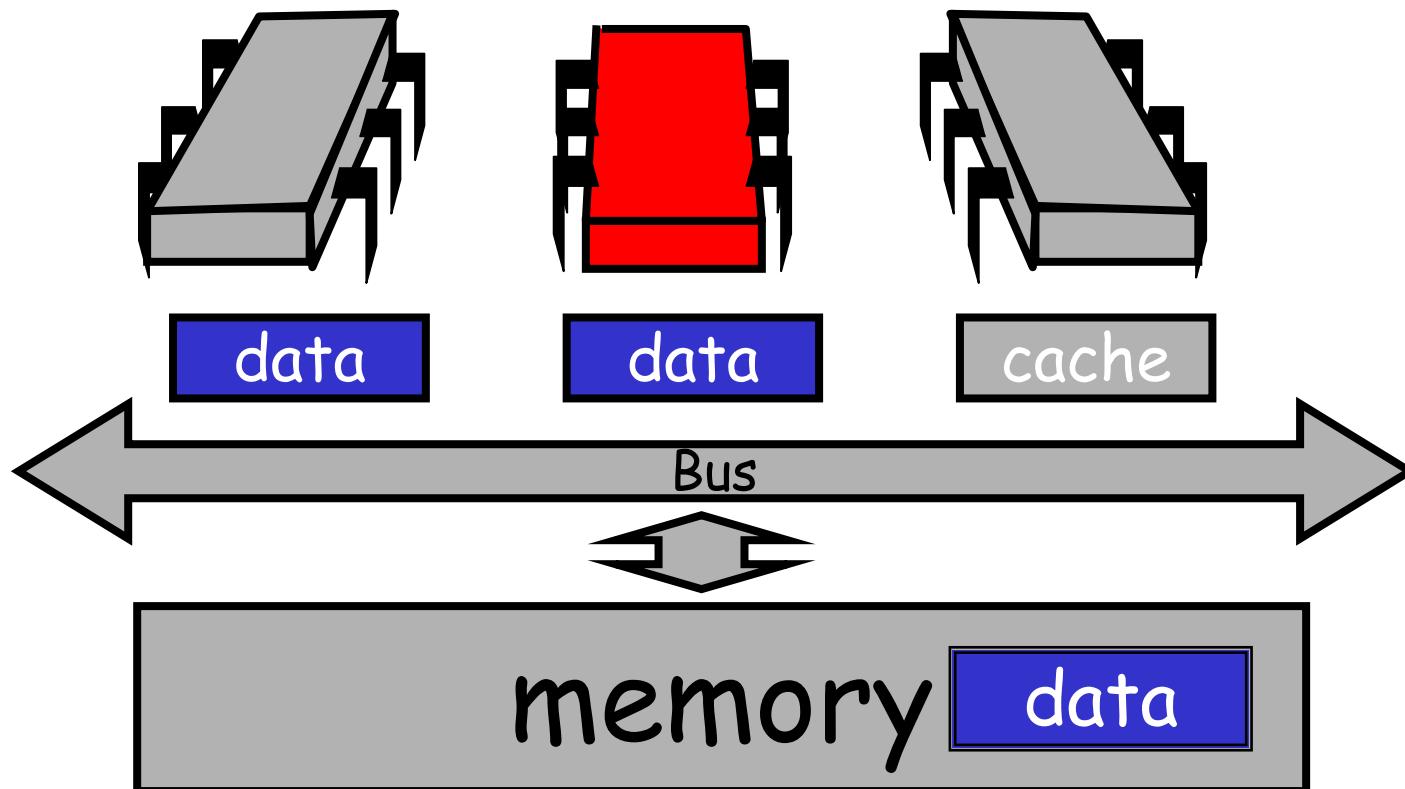
# Other Processor Responds



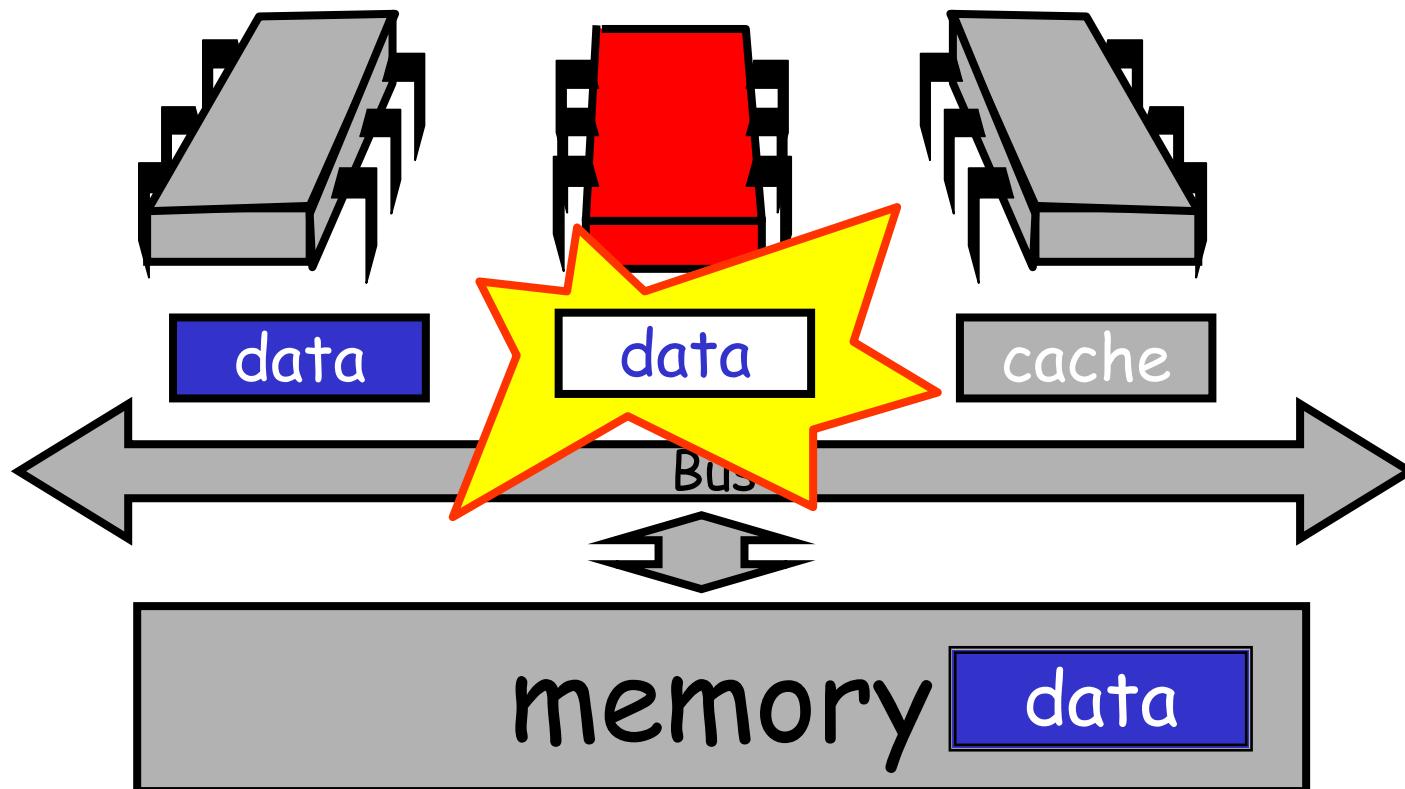
# Other Processor Responds



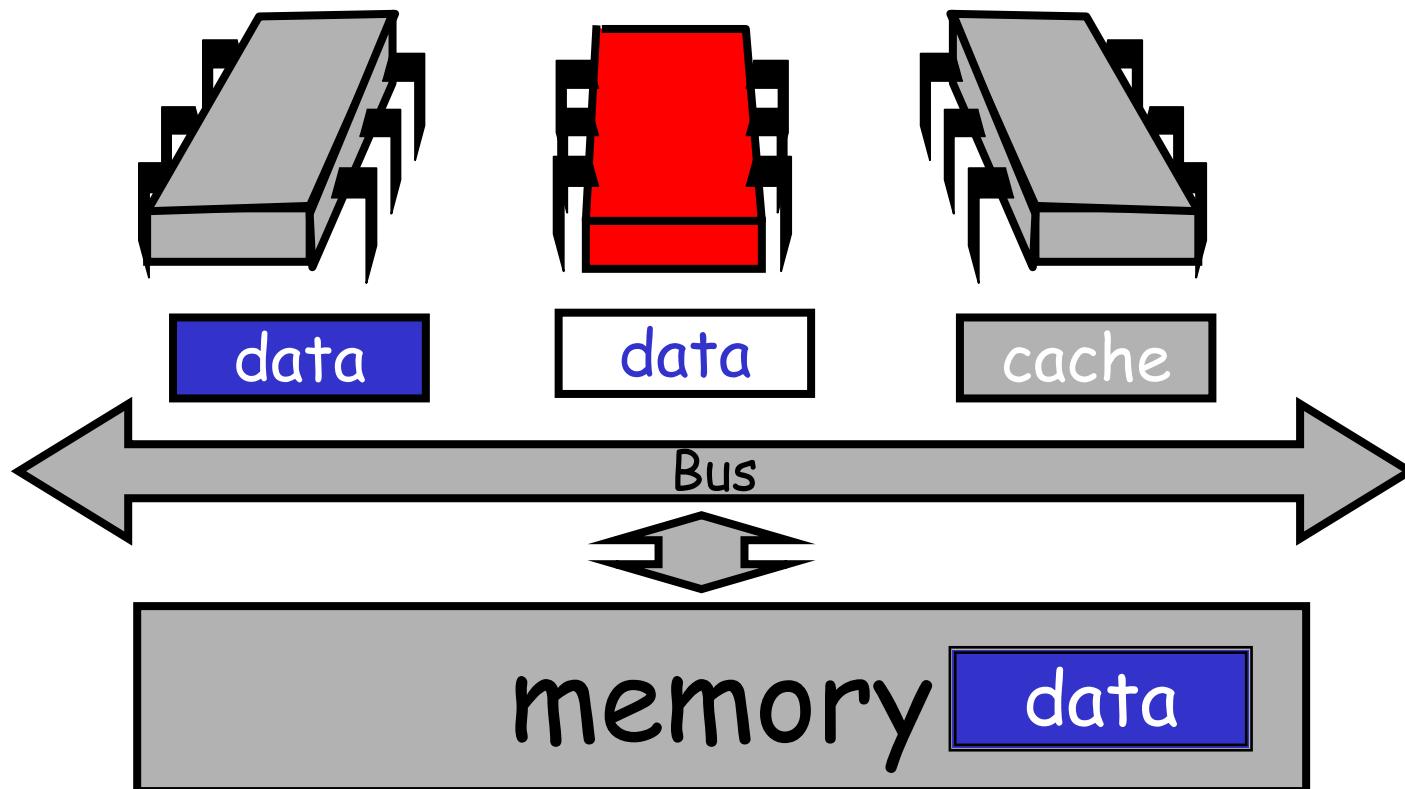
# Modify Cached Data



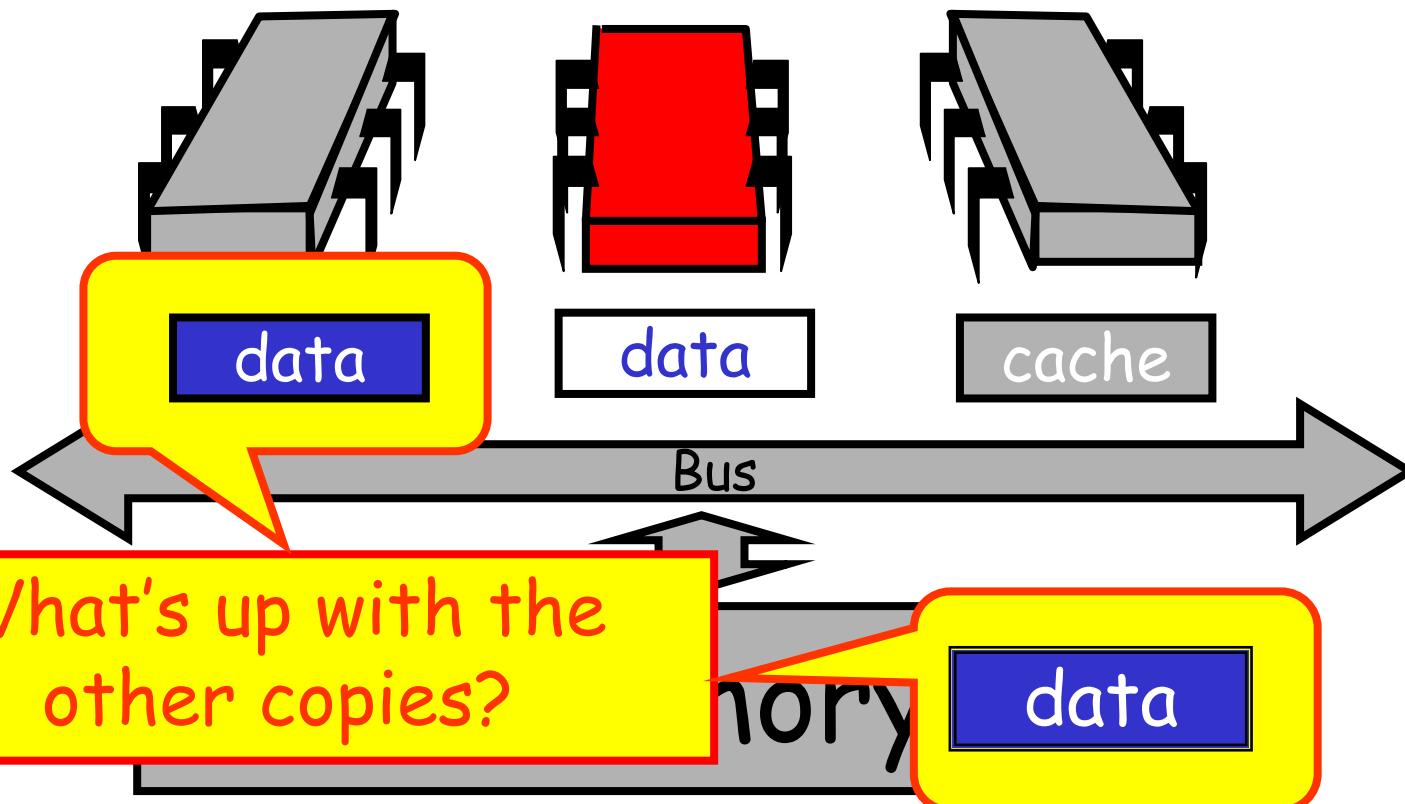
# Modify Cached Data

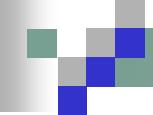


# Modify Cached Data



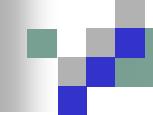
# Modify Cached Data





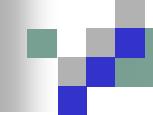
# Cache Coherence

- 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 Cache Coherence Protocol

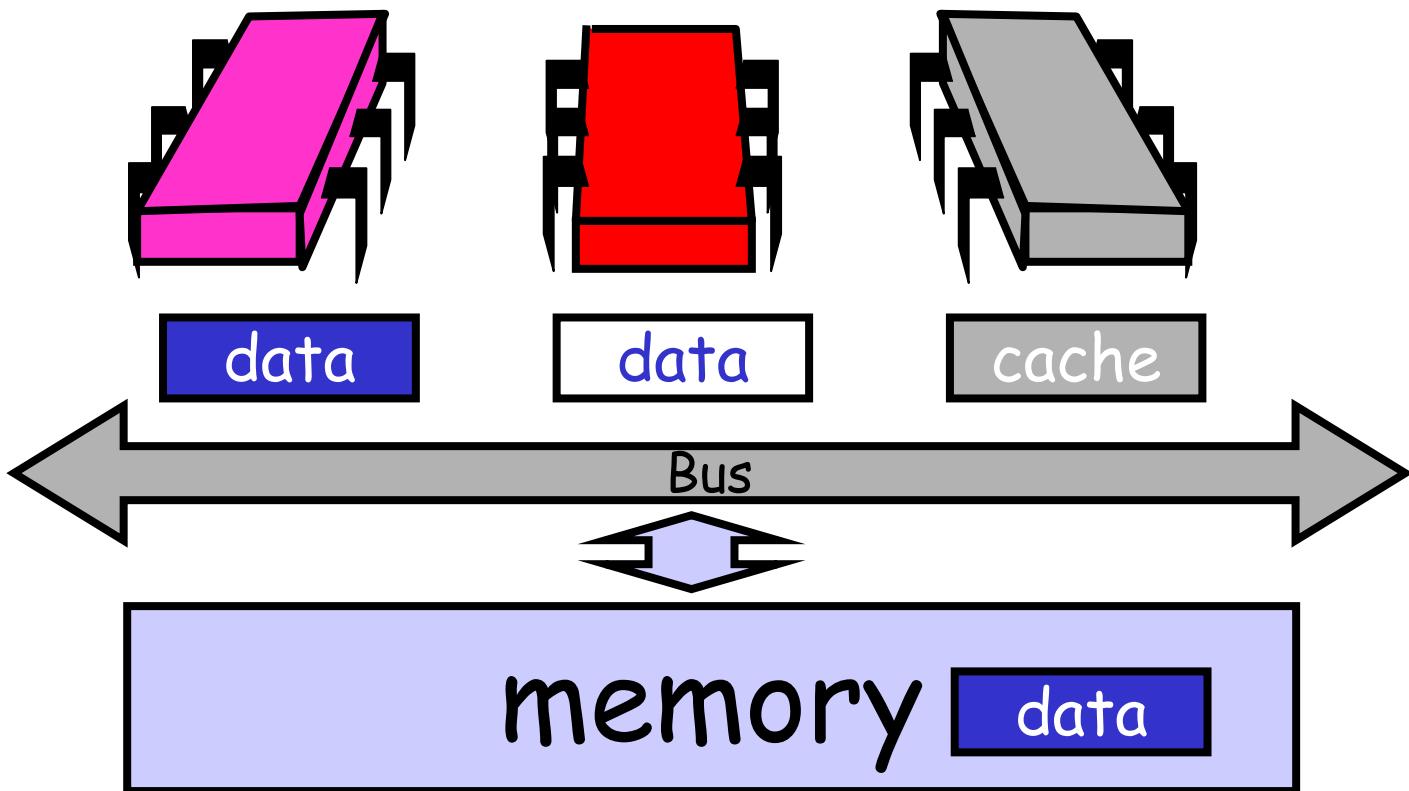
- Accumulate changes in cache
- Write back when needed
  - Need the cache for something else
  - Another processor wants it
- Write-back coherence protocol:
  - Invalidate other entries
  - Requires non-trivial protocol ...



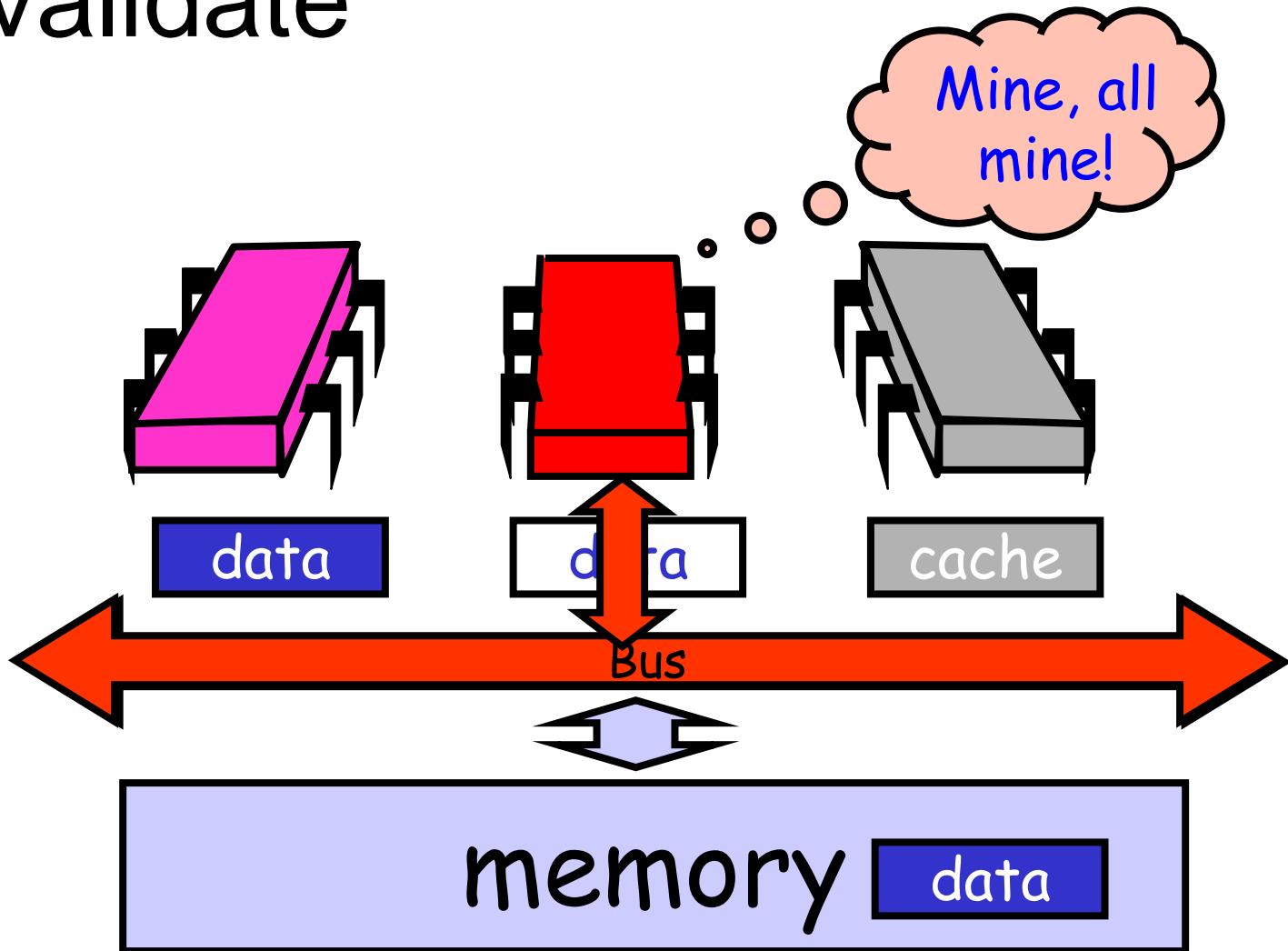
# Write-Back Caches

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

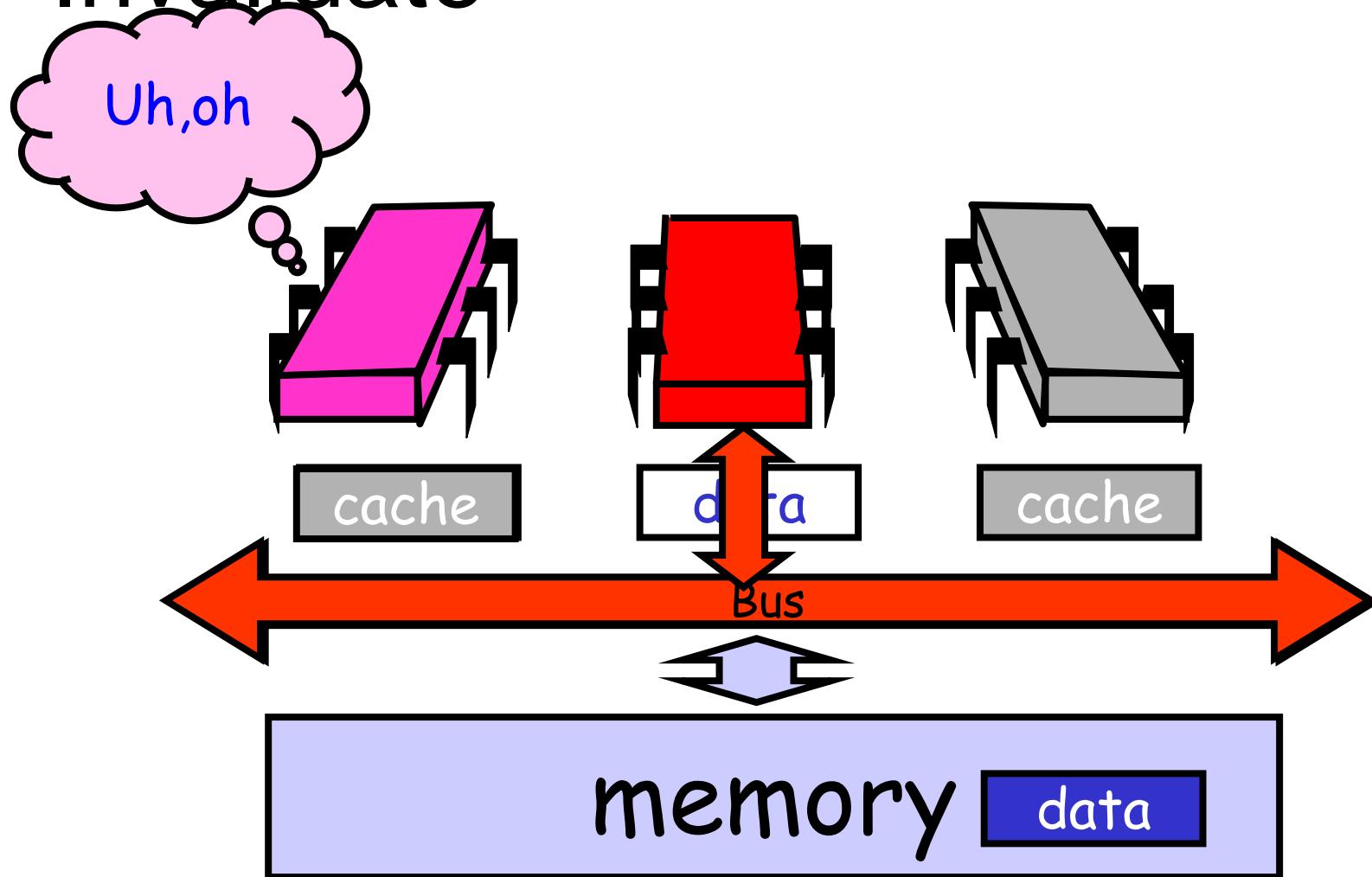
# Invalidate



# Invalidate

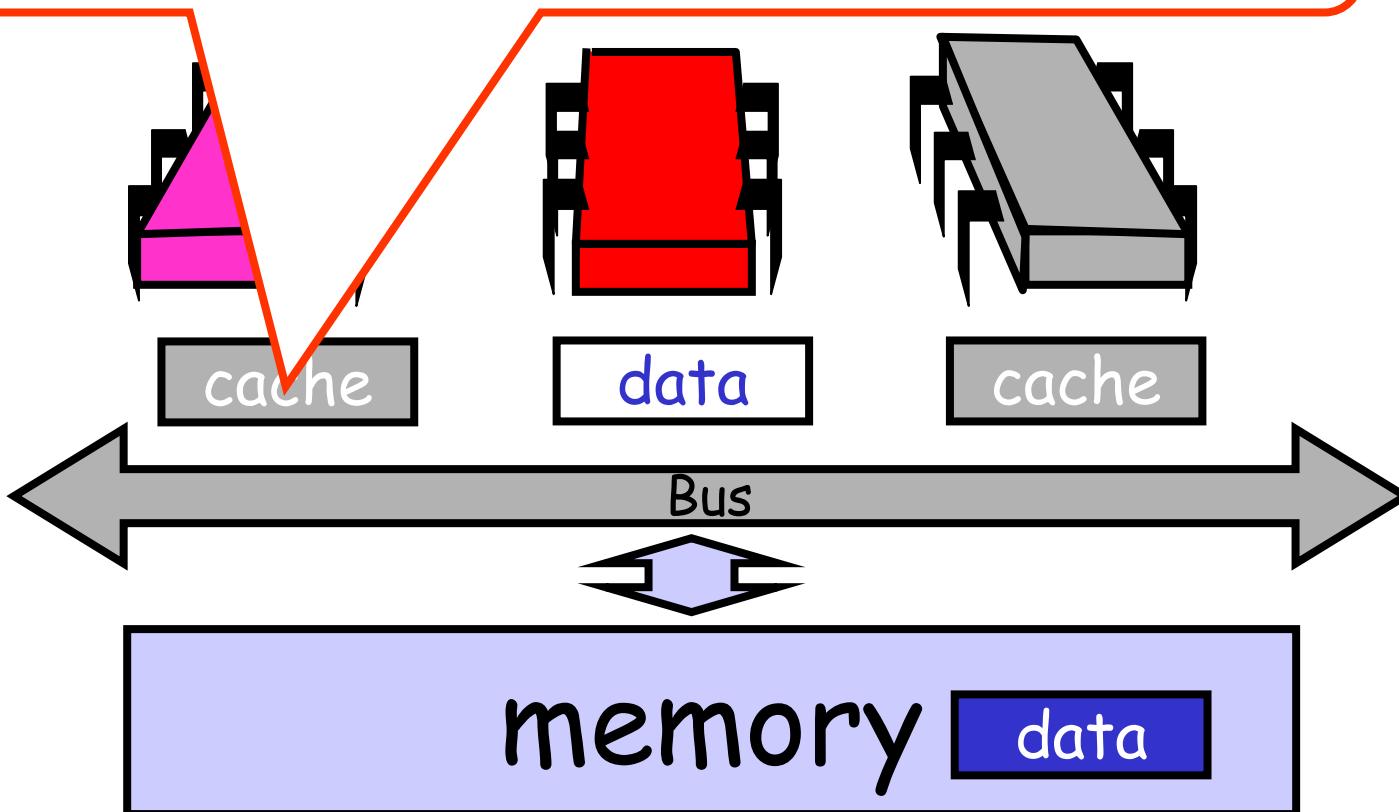


# Invalidate



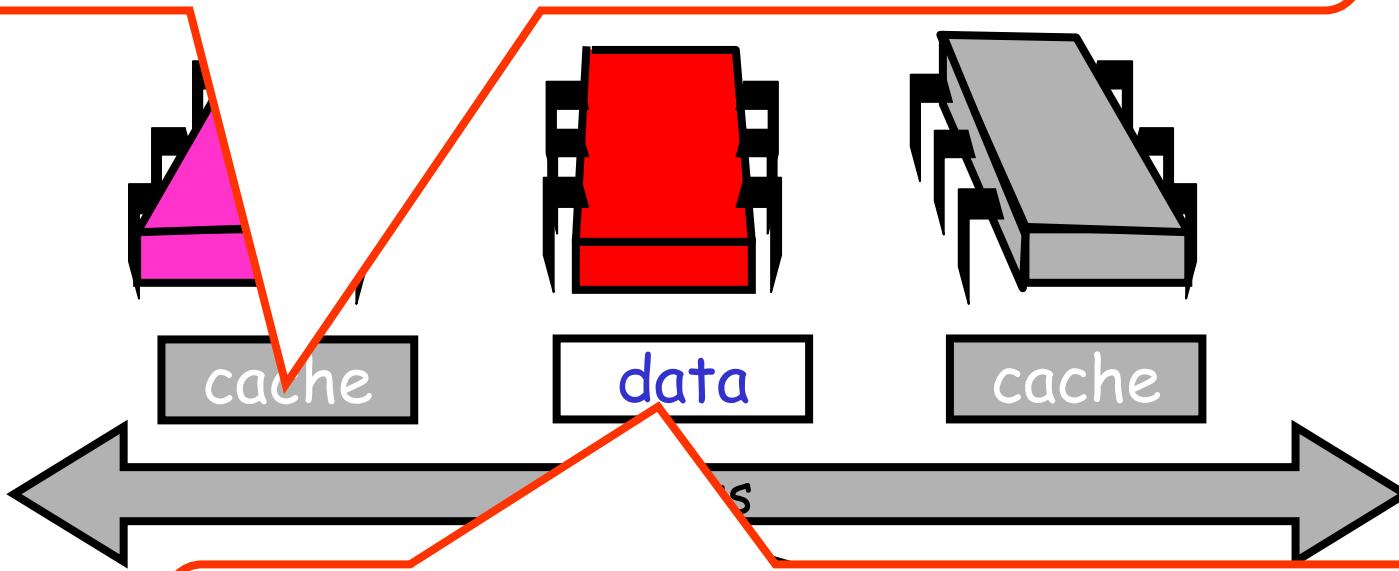
# Invalidate

Other caches lose read permission



# Invalidate

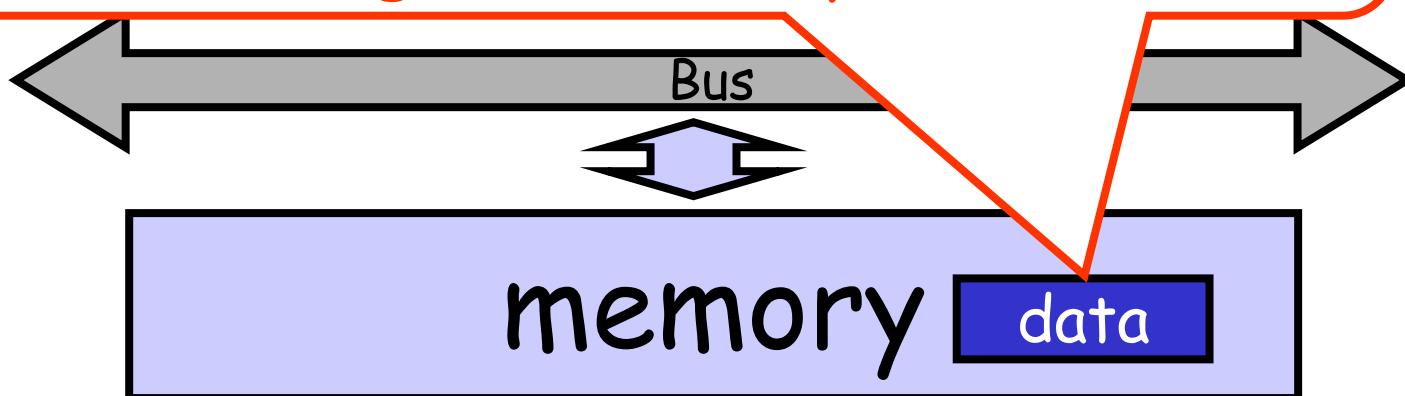
Other caches lose read permission



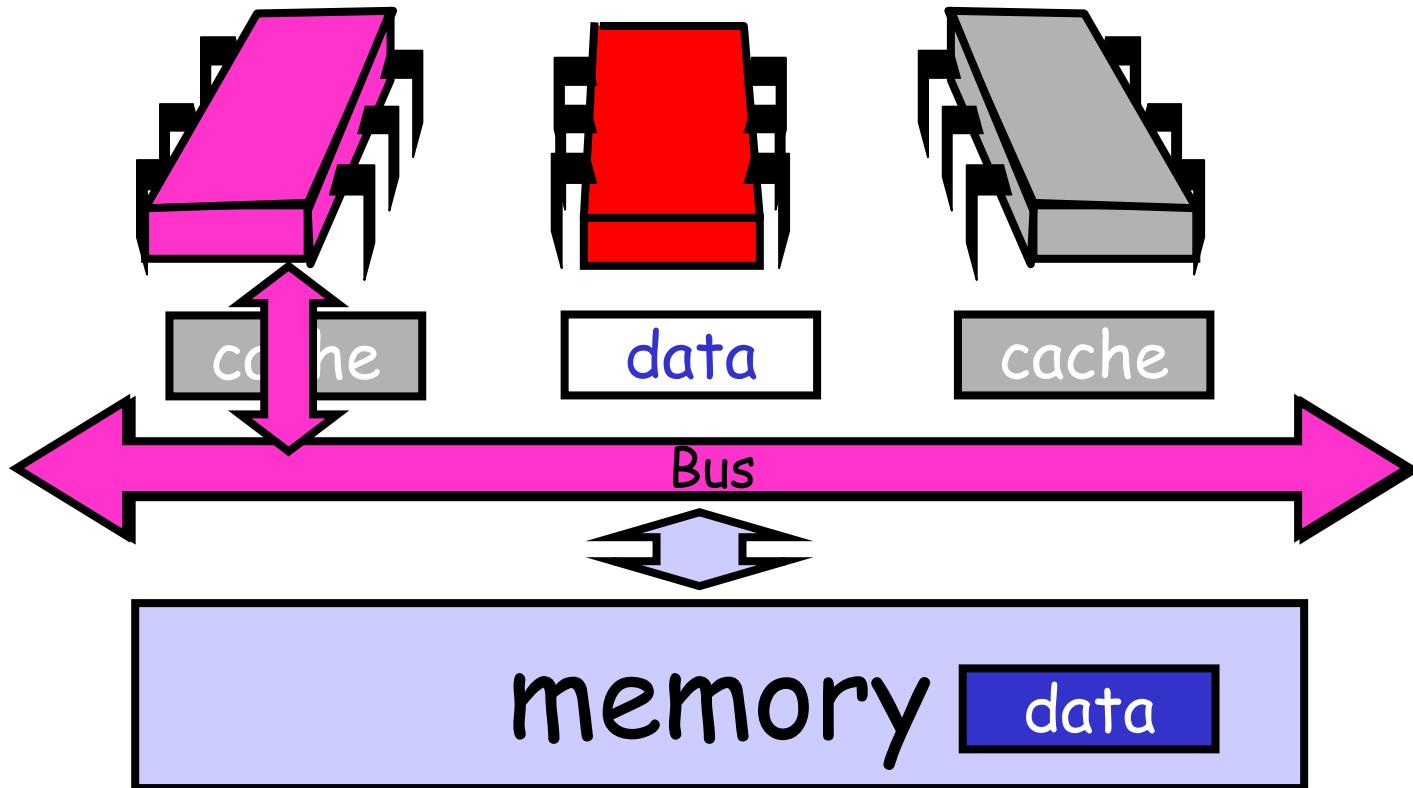
This cache acquires write permission

# Invalidate

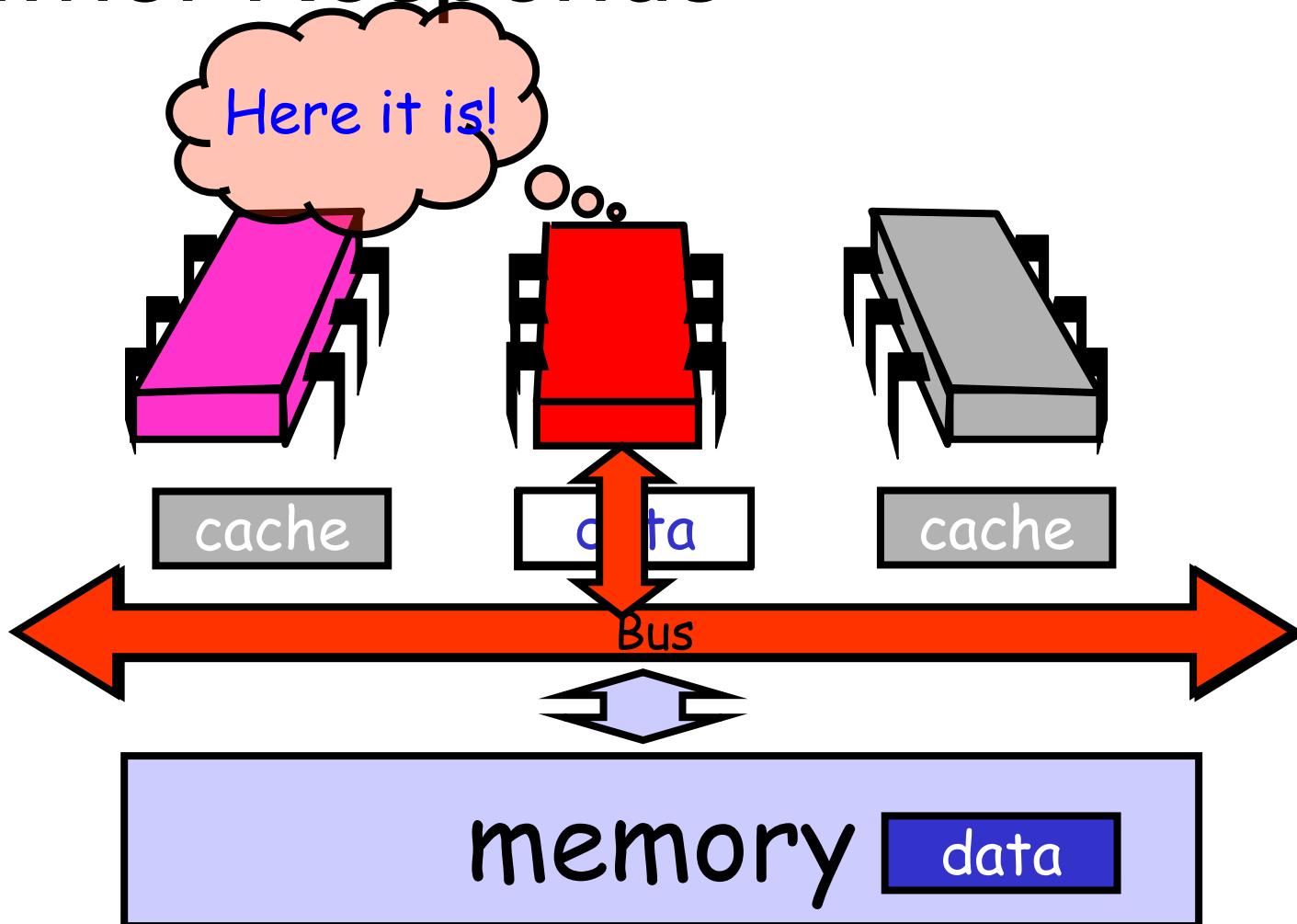
Memory provides data only if not present in any cache, so no need to change it now (expensive)



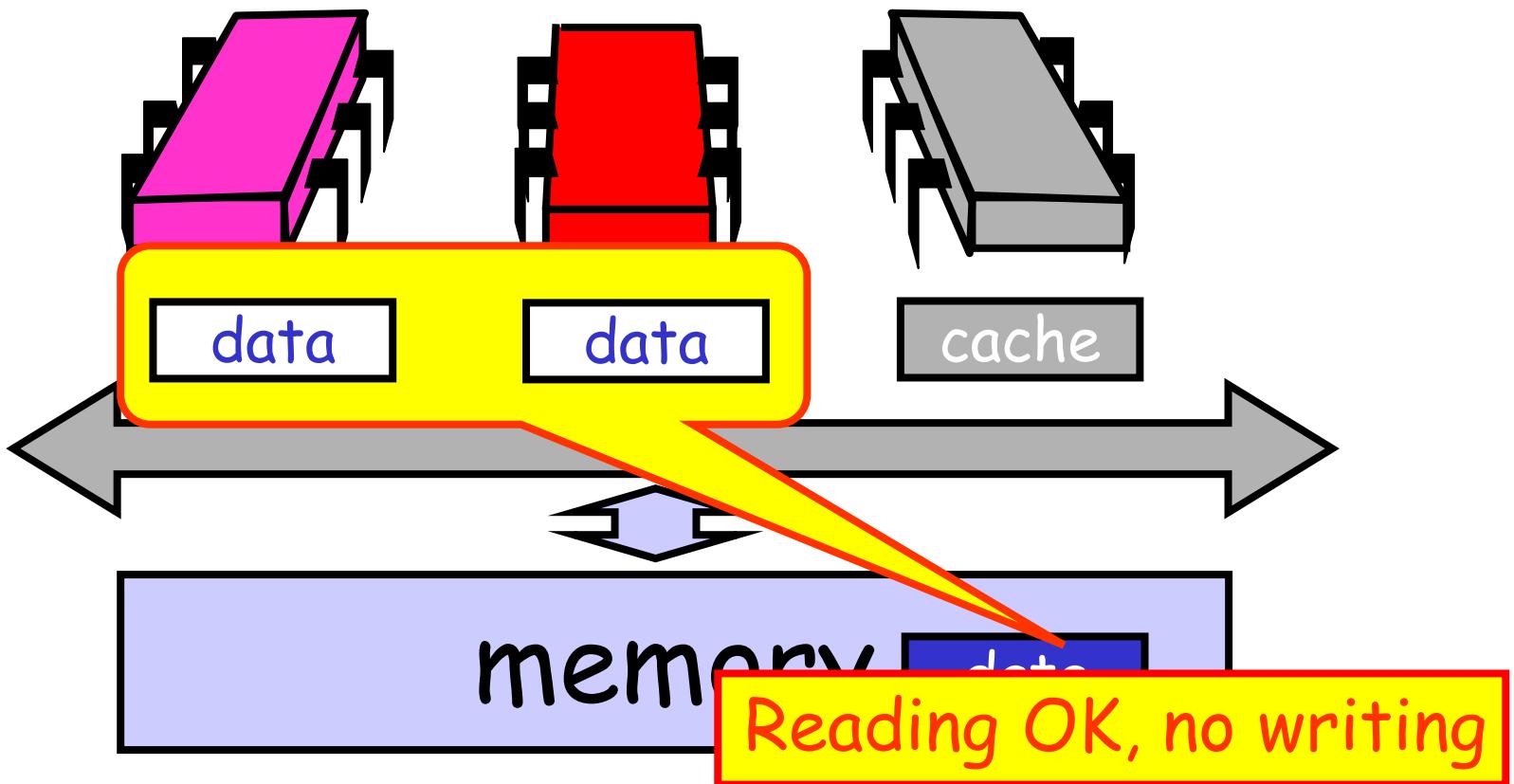
# Another Processor Asks for Data



# Owner Responds



# End of the Day ...



# Test-and-set Lock

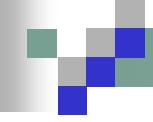
```
class TASLock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (state.getAndSet(true)) {}  
    }  
}
```

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

Has to change to  
current value while  
spinning

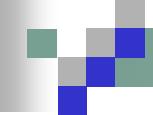
# Back to TASLocks

- How does a TASLock perform on a write-back shared-bus architecture?
  - Because it uses the bus, each getAndSet() call delays all the other threads
    - Even those not waiting for the lock
  - The getAndSet() call forces the other processors to discard their own cached copies – resulting in a cache miss every time
  - They must then use the bus to fetch the new, but unchanged value



# TASLock

- When the thread wants to release the lock it may be delayed because the bus is being monopolized by the spinners



# 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;  
    }  
}
```

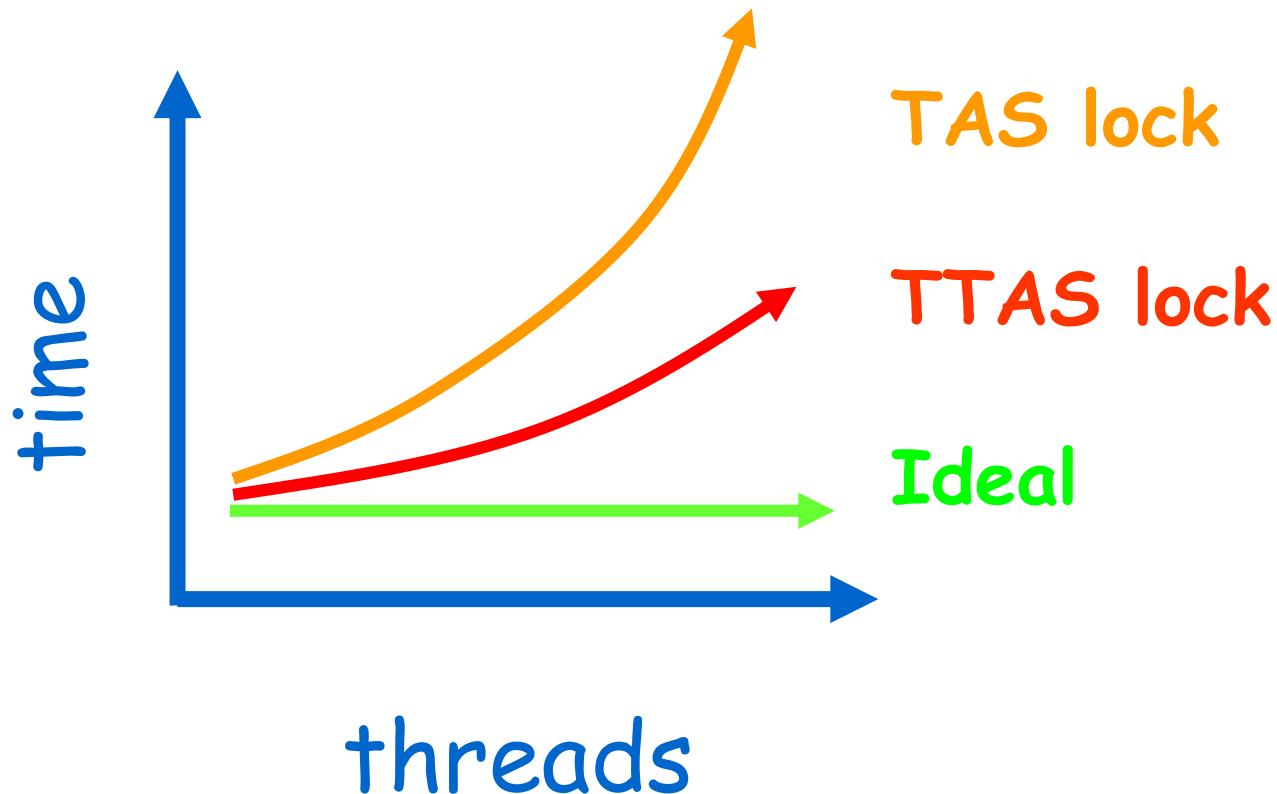
Wait until lock looks free

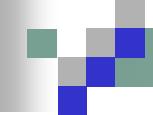
# Test-and-test-and-set Lock

```
class TTASLock {  
    AtomicBoolean state =  
        new AtomicBoolean(false);  
  
    void lock() {  
        while (true) {  
            while (state.get()) {}  
            if (!state.getAndSet(true))  
                return;  
    }  
}
```

Then try to  
acquire it

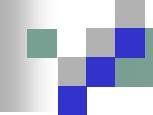
# Mystery #2





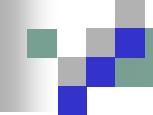
# What about the TTASLock?

- Suppose thread A acquires the lock.
- The first time thread B reads the lock it takes a cache miss and has to use the bus to fetch the new value
- As long as A holds the lock however, B repeatedly rereads the value – resulting in a cache hit every time
- B thus produces no extra traffic



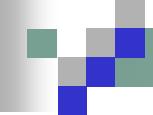
# What about the TTASLock?

- However when A releases the lock:
  - A writes false to the lock variable
  - The spinner's cached copies are invalidated
  - Each one takes a cache miss
  - They all use the bus to read a new value
  - They all call getAndSet() to acquire the lock
  - The first one to acquire the lock invalidates the others who must then reread the value
  - Storm of traffic



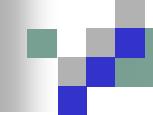
# Local spinning

- Threads repeatedly reread cached values instead of repeatedly using the bus



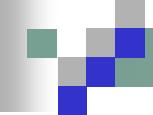
# Exponential Backoff

- Recall that in the TTASLock, the thread first reads the lock and if it appears to be free it attempts to acquire the lock
- “If I see that the lock is free, but then another thread acquires it before I can, then there must be high contention for that lock”
- Better to back off and try again later



# For how long should a thread back off?

- Rule of thumb:
  - The larger number of unsuccessful tries, the higher the contention, the longer the thread should back off.
- Approach:
  - Whenever the thread sees the lock has become free, but fails to acquire it, it backs off before retrying



# What about lock-step?

- What happens if all the threads backs off the same amount of time?
- Instead the threads should back off for a random amount of time
- Each time the thread tries and fails to get the lock, it doubles the back-off time, up to a fixed maximum.

# Exponential Backoff Lock

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

# Exponential Backoff Lock

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

Fix minimum delay

# Exponential Backoff Lock

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

# Exponential Backoff Lock

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

If we win, return

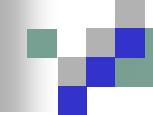
# Exponential Backoff Lock

```
public class Backoff implements Lock {  
    public void lock() {  
        int delay = MIN_DELAY; Otherwise back off for random duration  
        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;  
        }  
    }  
}
```

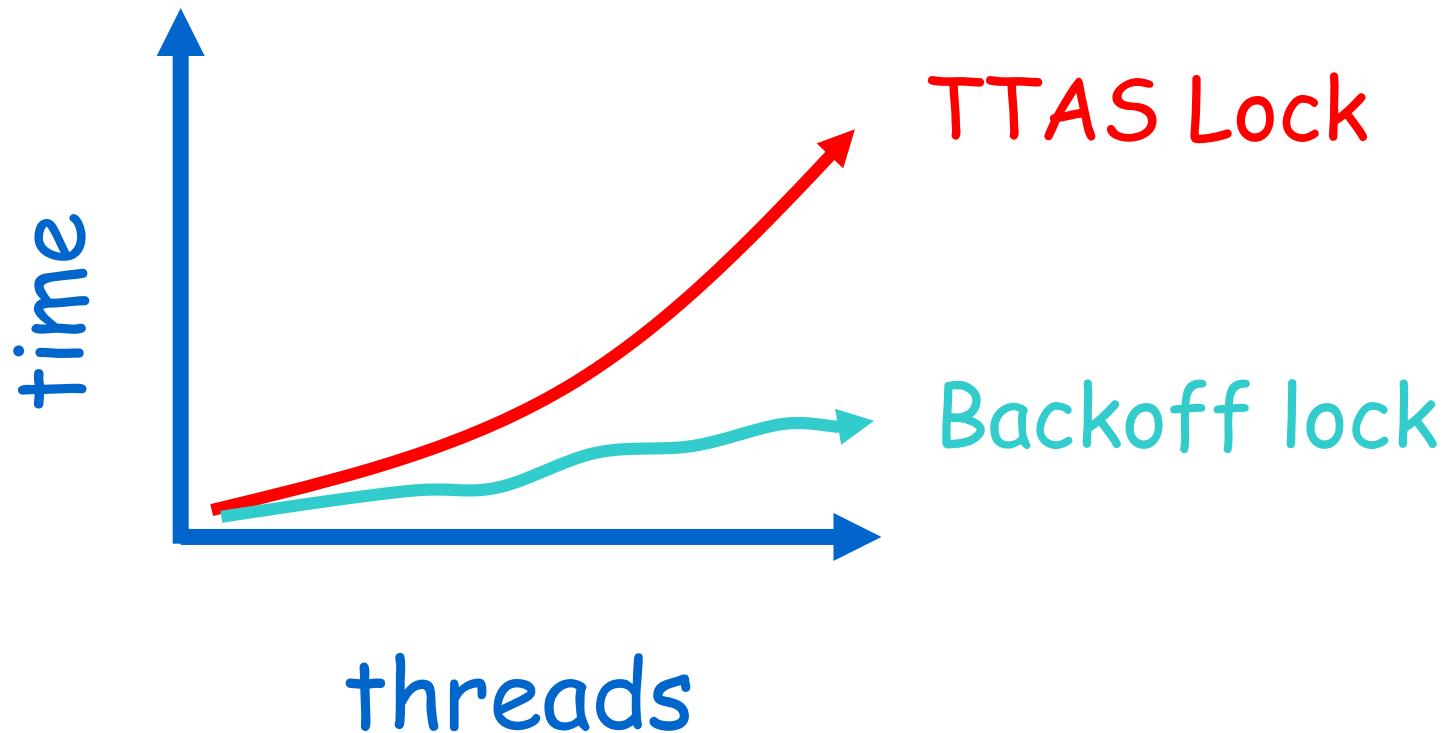
*Double max delay,  
within reason*

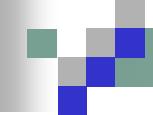


# Exponential Backoff Lock

- Important to note that a thread backs off only when it fails to acquire a lock that was just available, but is not available anymore
- Observing that a lock is held by another thread does not imply backoff

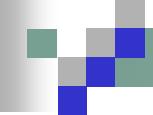
# Spin-Waiting Overhead





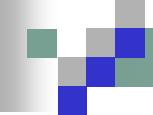
# Backoff: Other Issues

- Good
  - Easy to implement
  - Beats TTAS lock
- Bad
  - Must choose parameters carefully
    - Sensitive to choice of minimum and maximum delays
  - Sensitive to number of processors and their speed
    - Cannot have a general solution for all platforms and machines



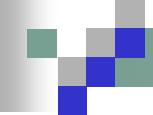
# BackoffLock drawbacks

- Cache-coherence Traffic:
  - All threads spin on the same location
- Critical Section Underutilization:
  - Threads delay longer than necessary



# Idea

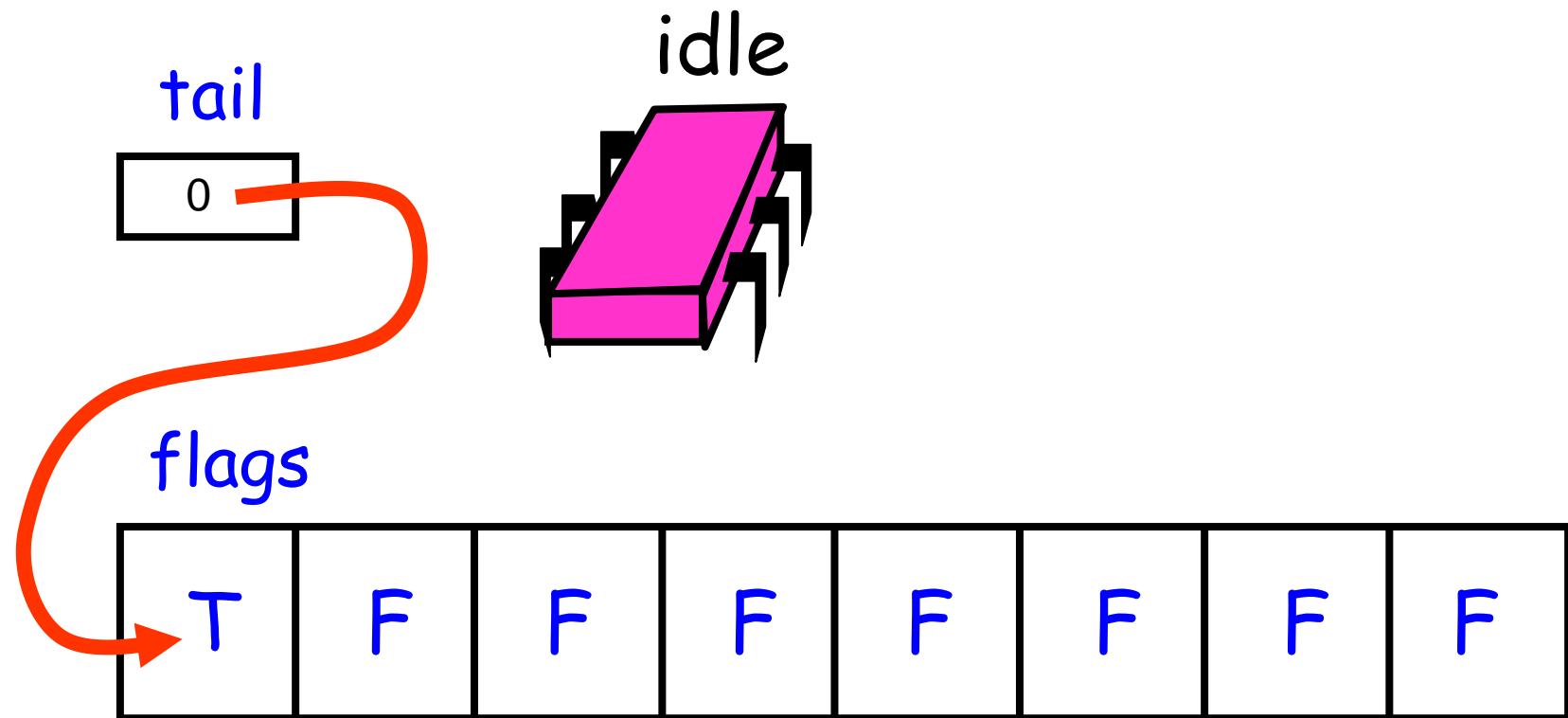
- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others



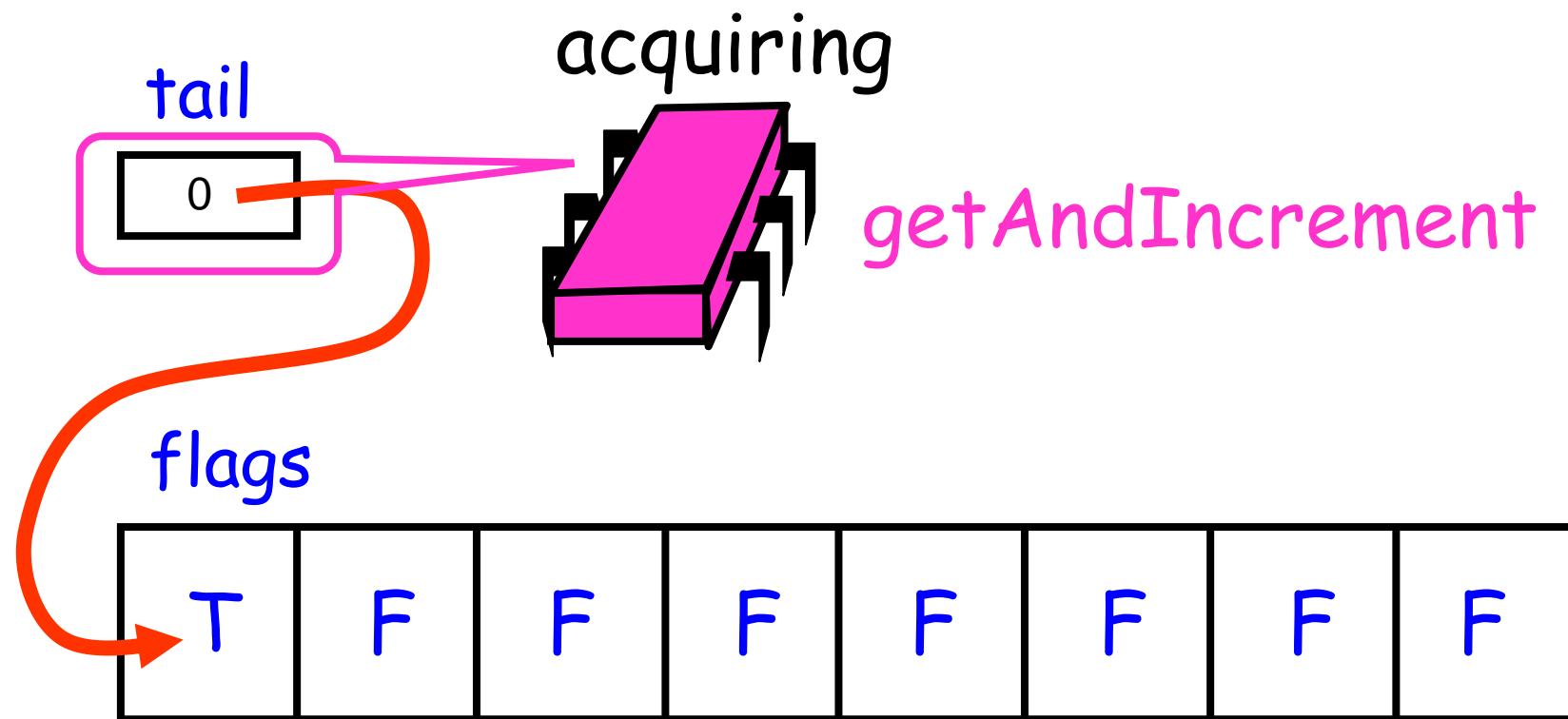
# Queue Locks

- Cache-coherence traffic is reduced since each thread spins on a different location
- No need to guess when to attempt to access lock – increase critical section utilization
- First-come-first-served fairness

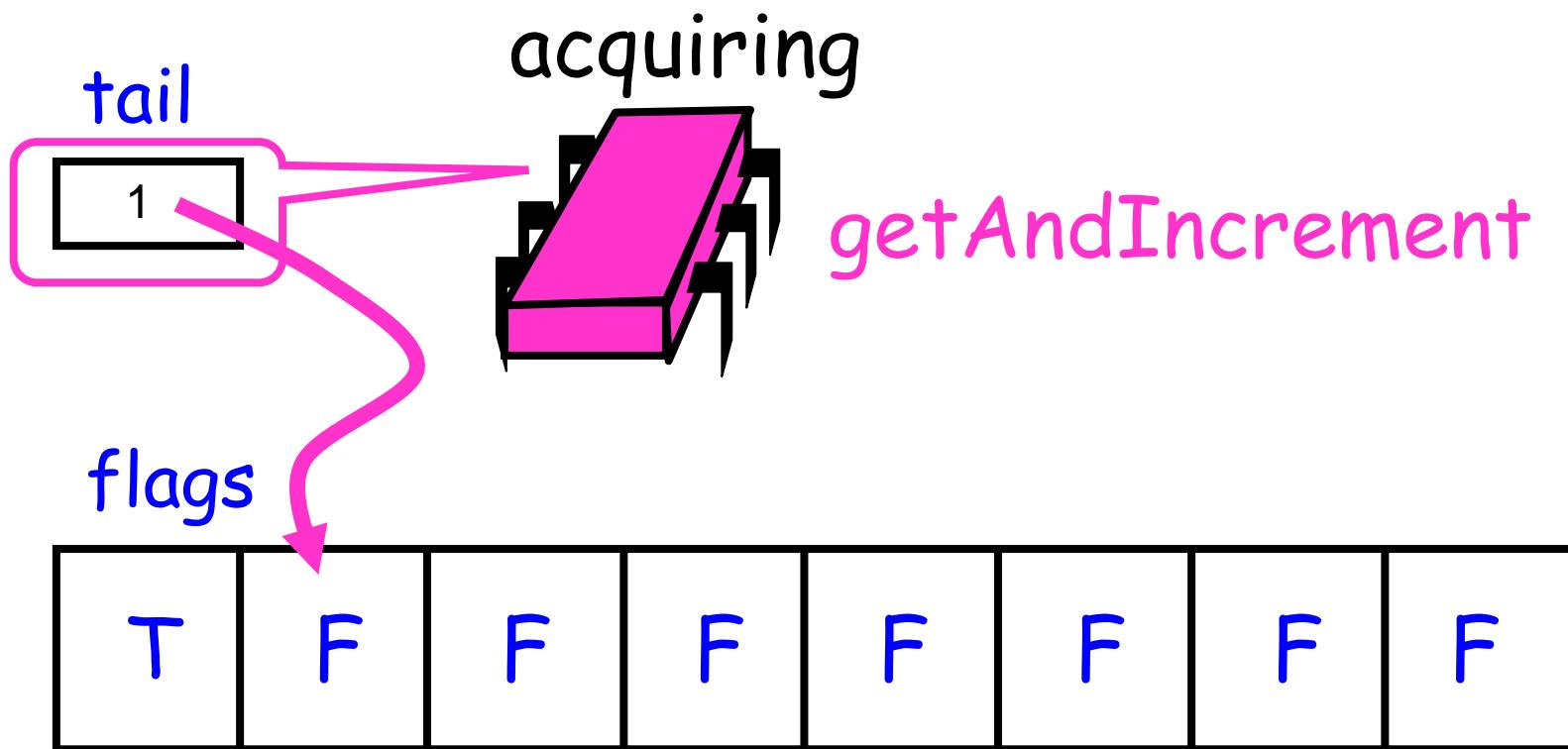
# Anderson Queue Lock



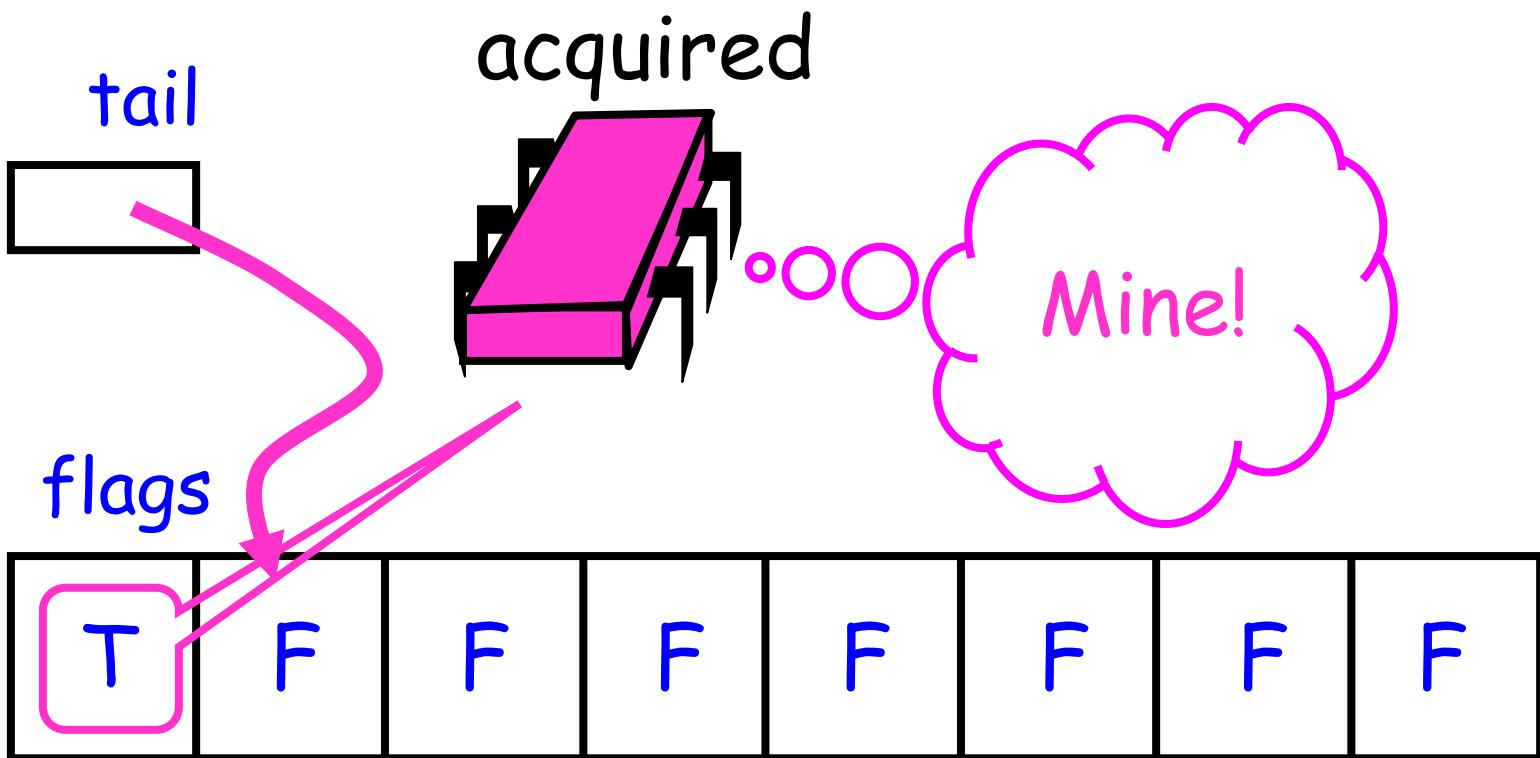
# Anderson Queue Lock



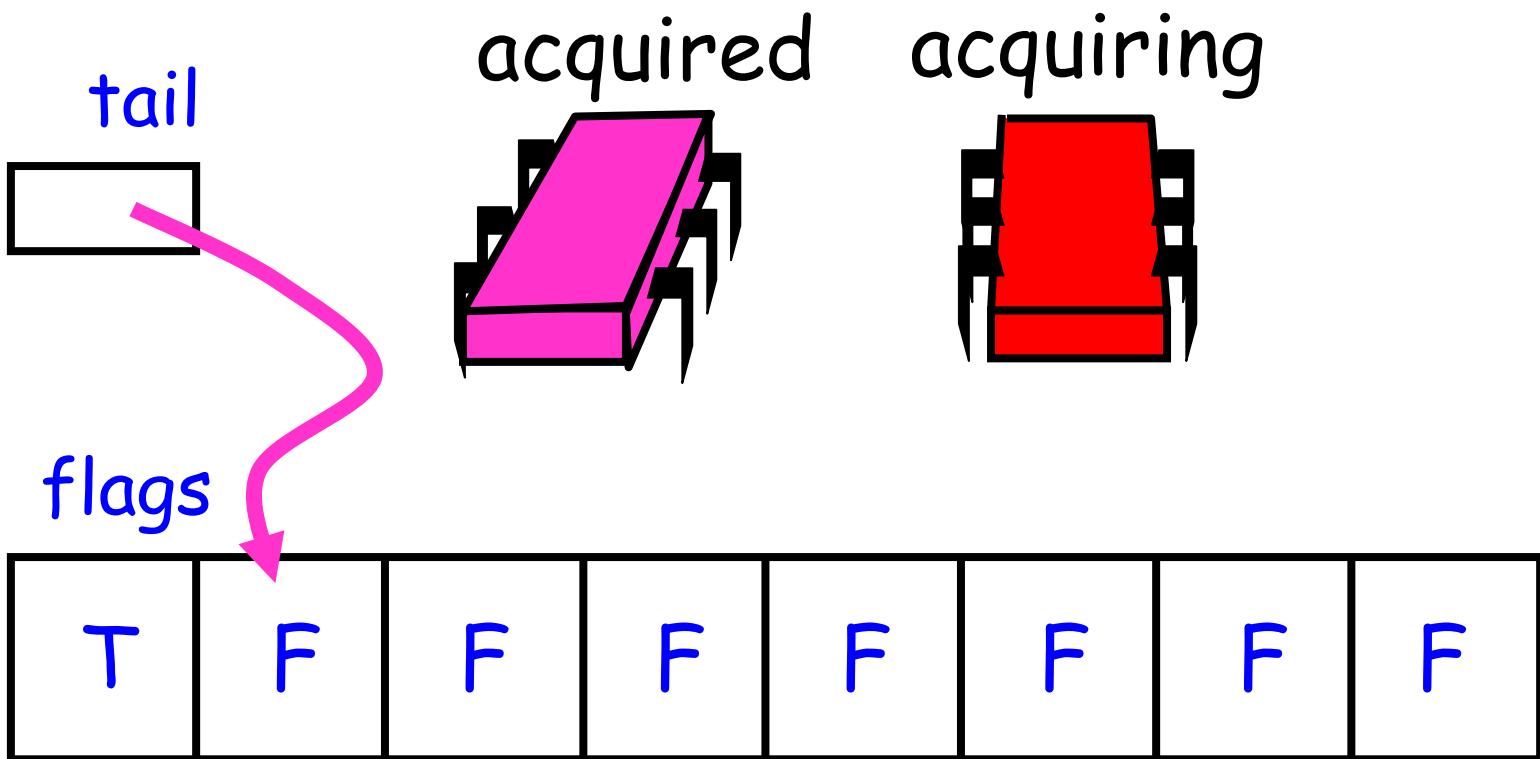
# Anderson Queue Lock



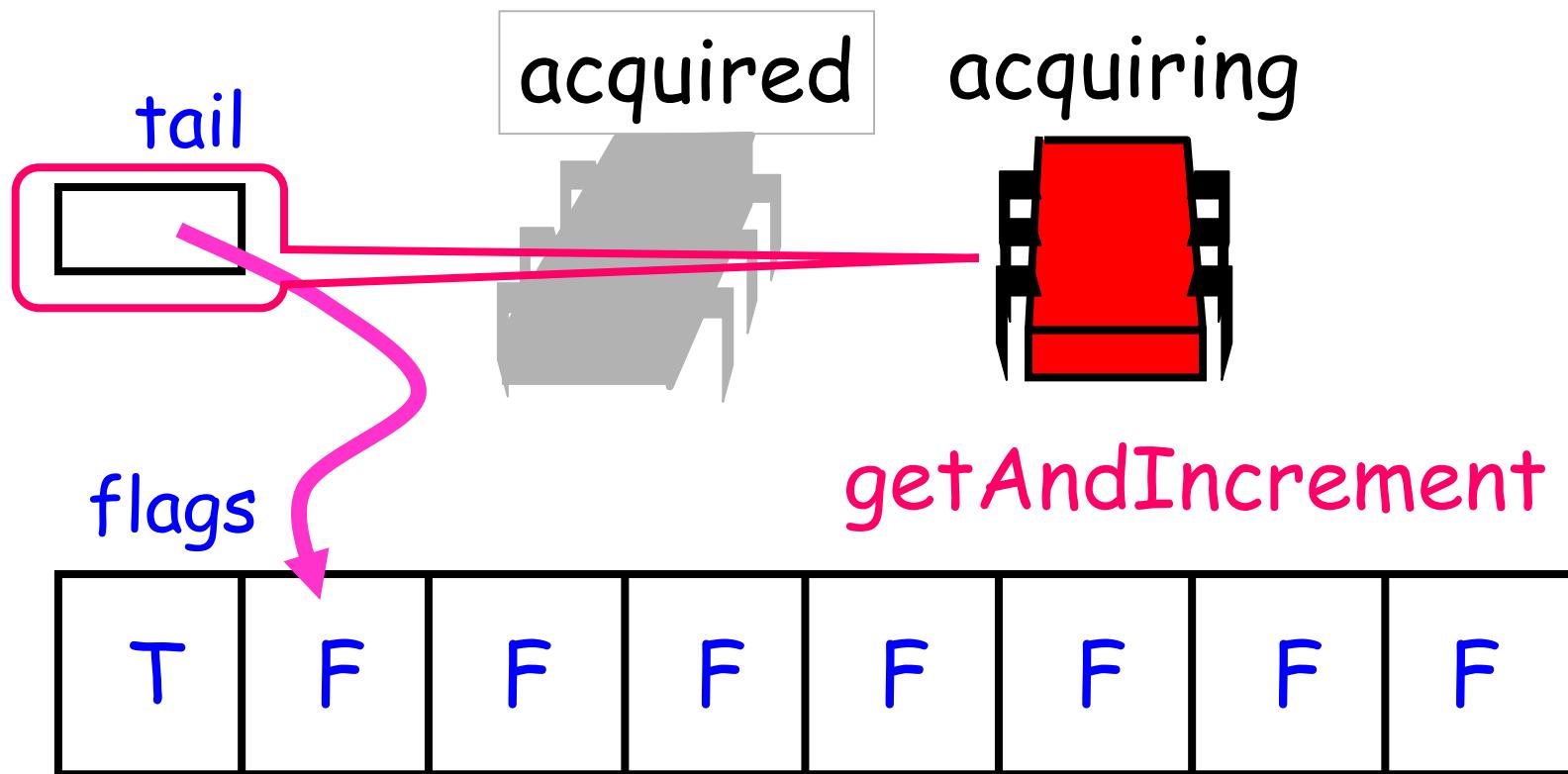
# Anderson Queue Lock



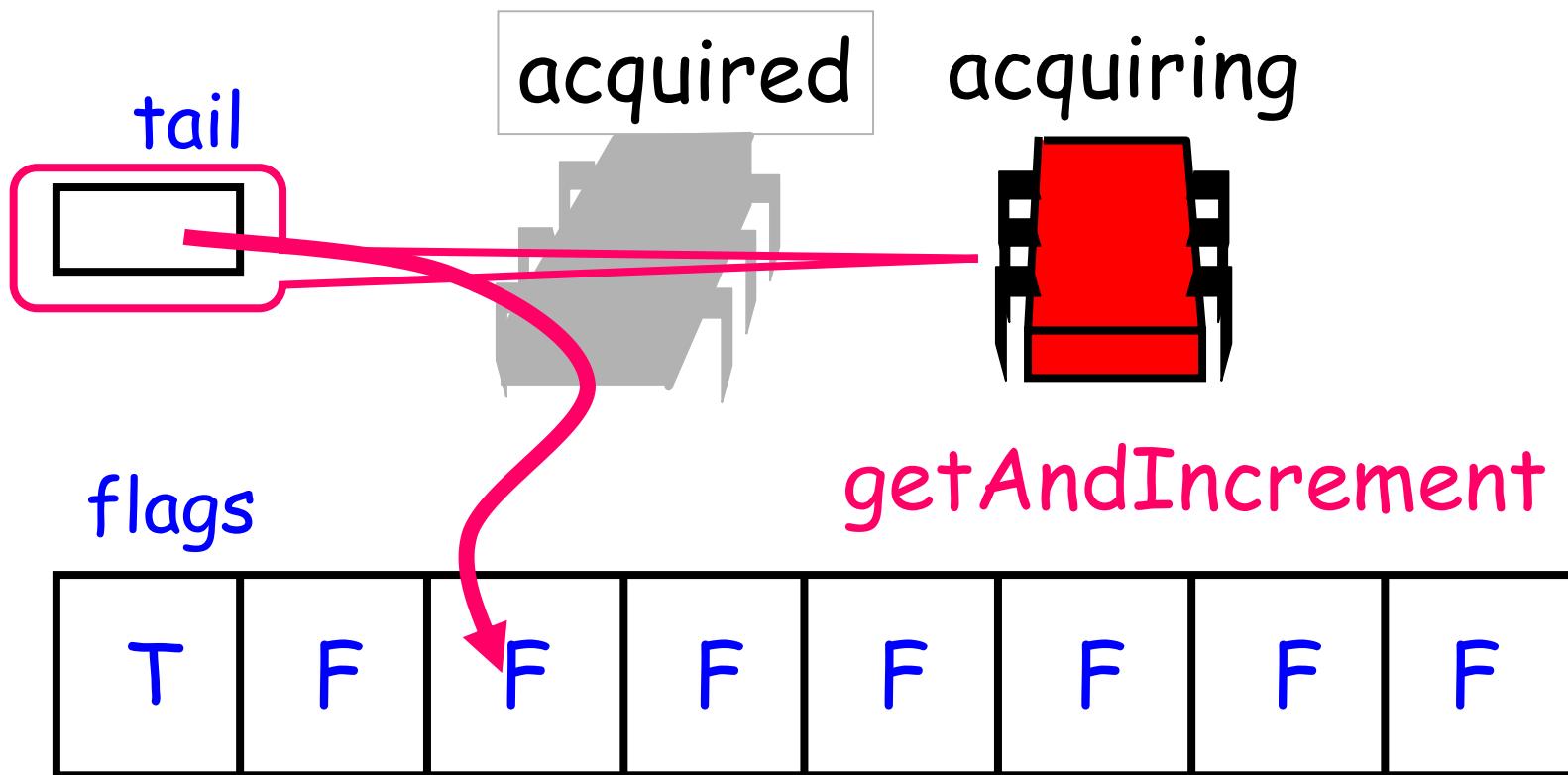
# Anderson Queue Lock



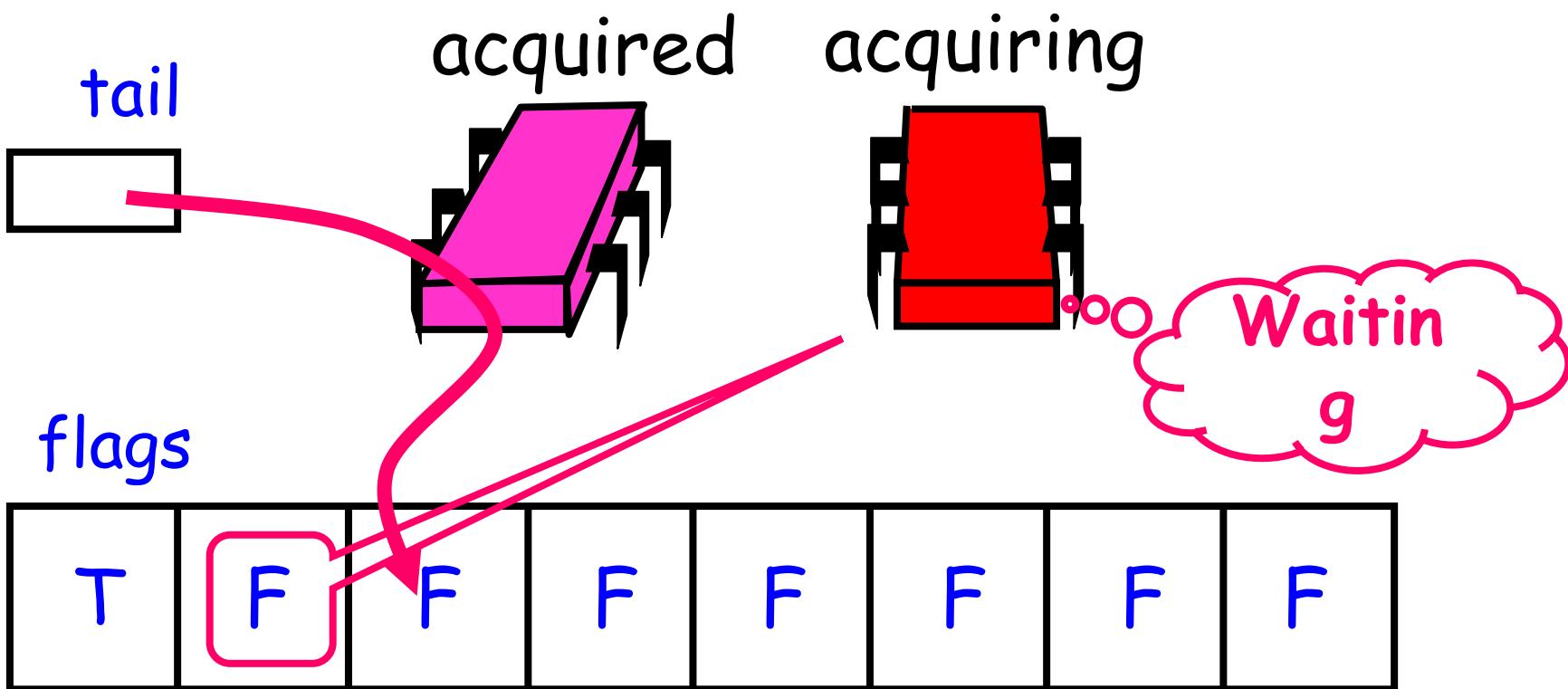
# Anderson Queue Lock



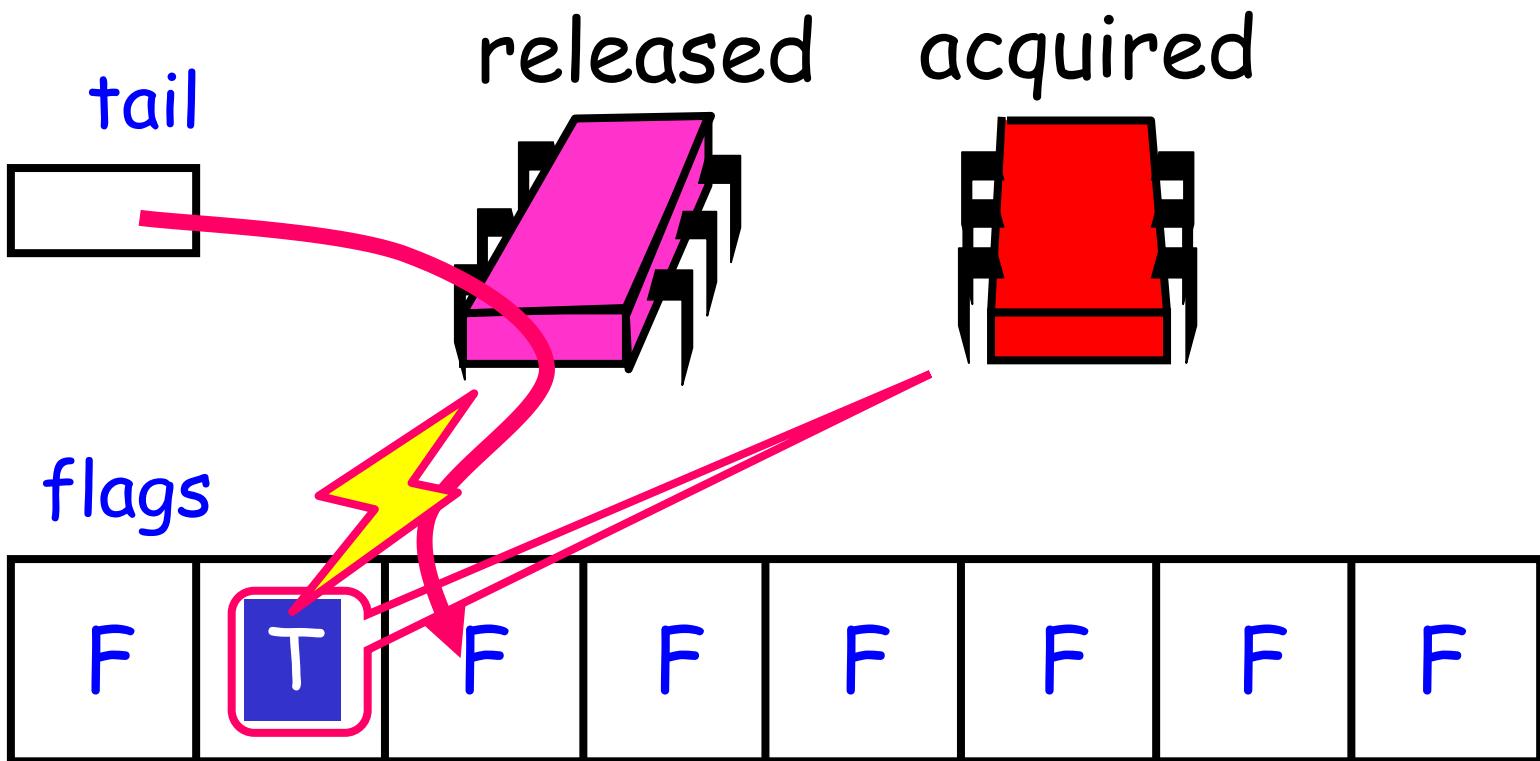
# Anderson Queue Lock



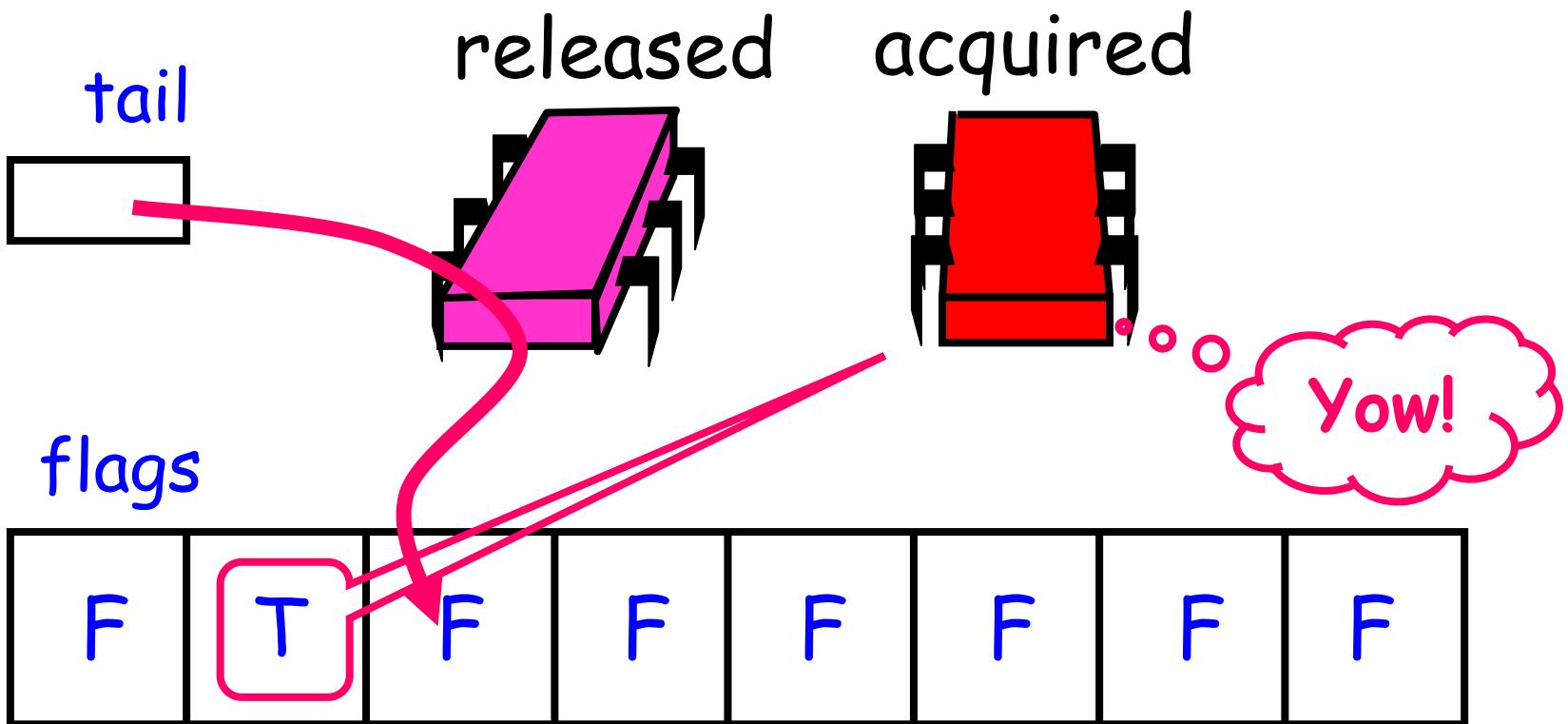
# Anderson Queue Lock



# Anderson Queue Lock



# Anderson Queue Lock



# Anderson Queue Lock

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

# Anderson Queue Lock

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

One flag per thread

# Anderson Queue Lock

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

Next flag to use

# Anderson Queue Lock

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

Thread-local variable

# Anderson Queue Lock

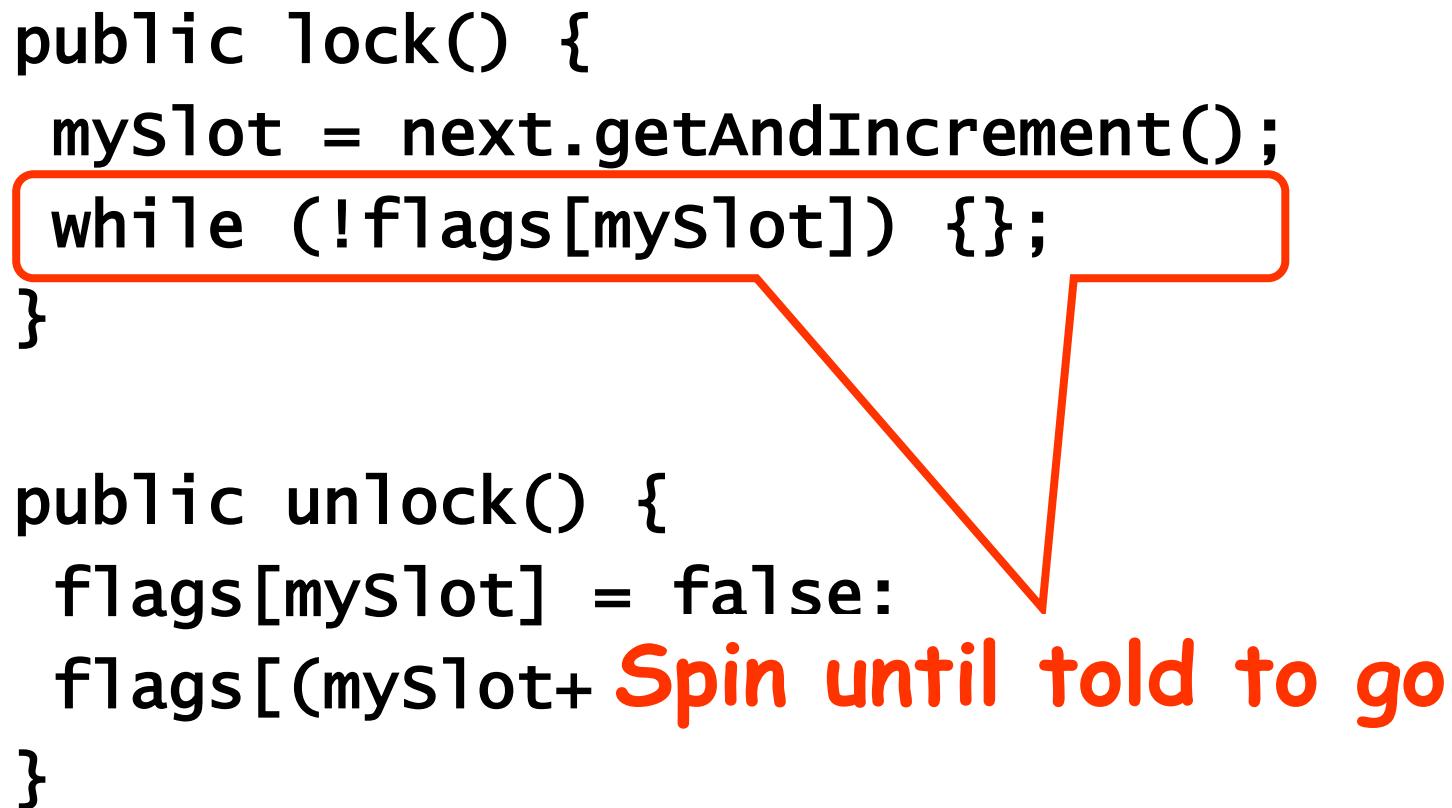
```
public lock() {  
    mySlot = tail.getAndIncrement();  
    while (!flags[mySlot]) {};  
}  
  
public unlock() {  
    flags[mySlot] = false;  
    flags[(mySlot+1)] = true;  
}
```

# Anderson Queue Lock

```
public lock() {  
    mySlot = tail.getAndIncrement();  
    while (!flags[mySlot]) {};  
}  
  
public unlock() {  
    flags[mySlot] = false  
    flags[(mySlot+1)] = Take next slot  
}
```

# Anderson Queue Lock

```
public lock() {  
    mySlot = next.getAndIncrement();  
    while (!flags[mySlot]) {};  
}  
  
public unlock() {  
    flags[mySlot] = false;  
    flags[(mySlot+ Spin until told to go  
}]
```



# Anderson Queue Lock

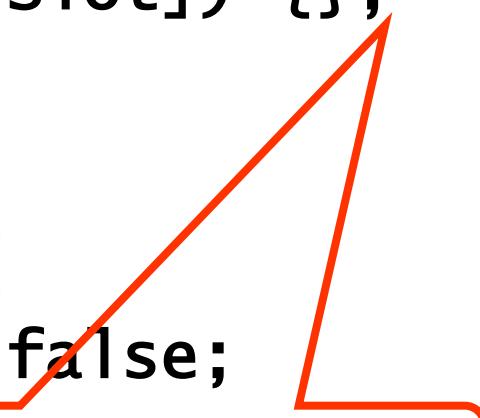
```
public lock() {  
    myslot = next.getAndIncrement();  
    while (!flags[myslot]) {};  
}  
  
public unlock() {  
    flags[myslot] = false;  
    flags[(myslot+1)] = true;  
}
```

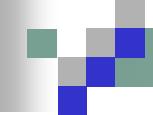
Prepare slot for re-use

# Anderson Queue Lock

```
public lock() {  
    mySlot = ne  
    while (!flags[mySlot]) ;  
}  
  
public unlock() {  
    flags[mySlot] = false;  
    flags[(mySlot+1)] = true;  
}
```

**Tell next thread to go**

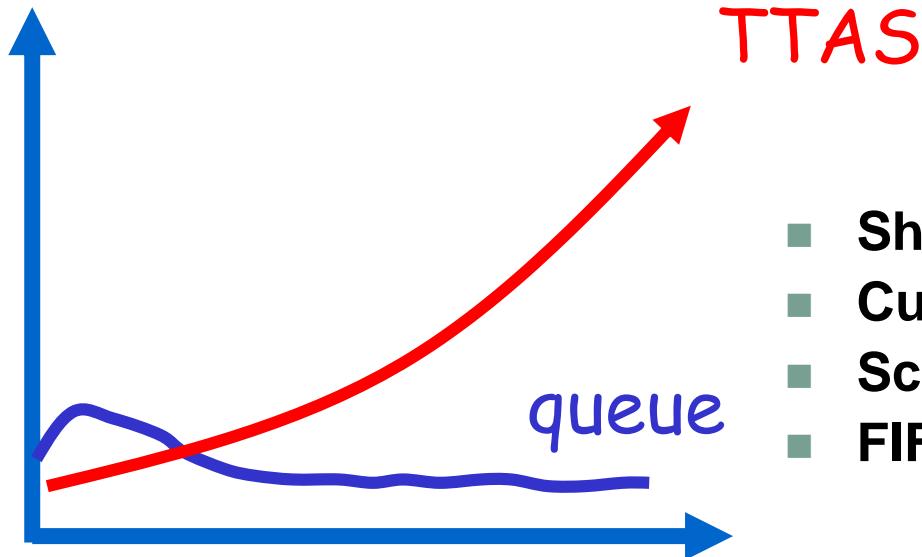




# Anderson Queue Lock

- Although the flags [] array is shared, contention on the array locations are minimised since each thread spins on its own locally cached copy of a single array location

# Performance



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

# Anderson Queue Lock

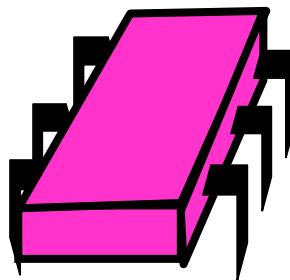
- Good
  - First truly scalable lock
  - Simple, easy to implement
- Bad
  - Not space efficient
  - One bit per thread
    - Unknown number of threads?
    - Small number of actual contenders?

# CLH Queue Lock

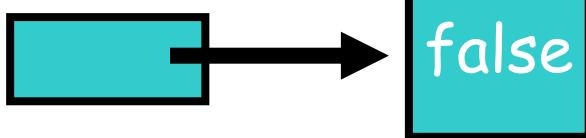
- Virtual Linked List keeps track of the queue
- Each thread's status is saved in its node:
  - True – has acquired the lock or wants to acquire the lock
  - False – is finished with the lock and has released it
- Each node keeps track of its predecessors status

# Initially

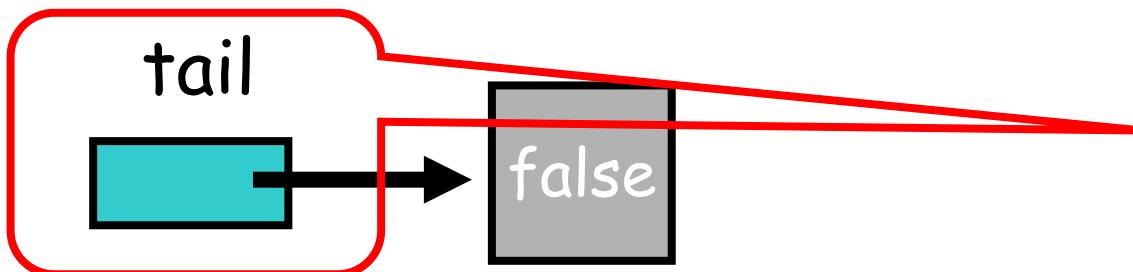
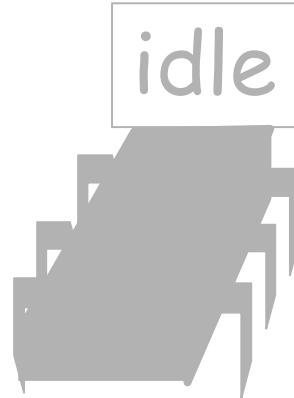
idle



tail

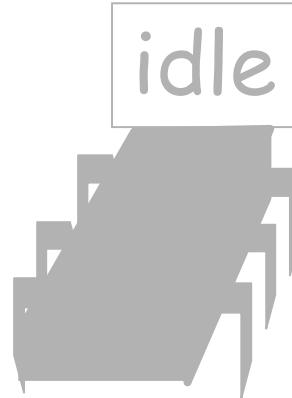


# Initially

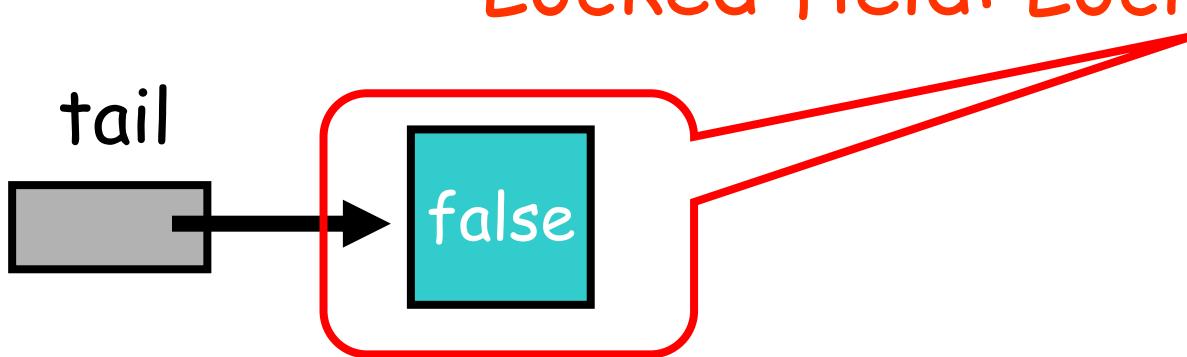


Queue tail

# Initially

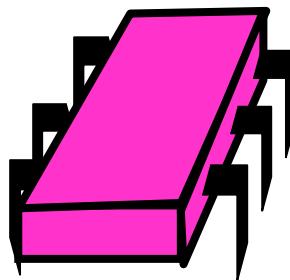


Locked field: Lock is free

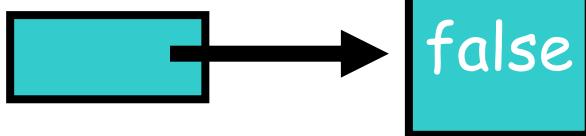


# Initially

idle

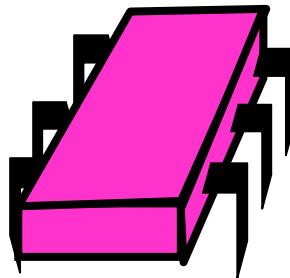


tail

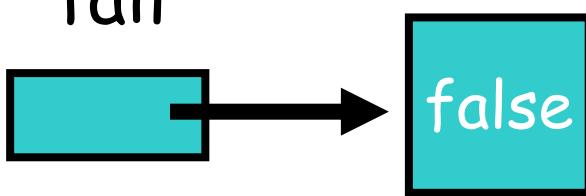


# Purple Wants the Lock

acquiring

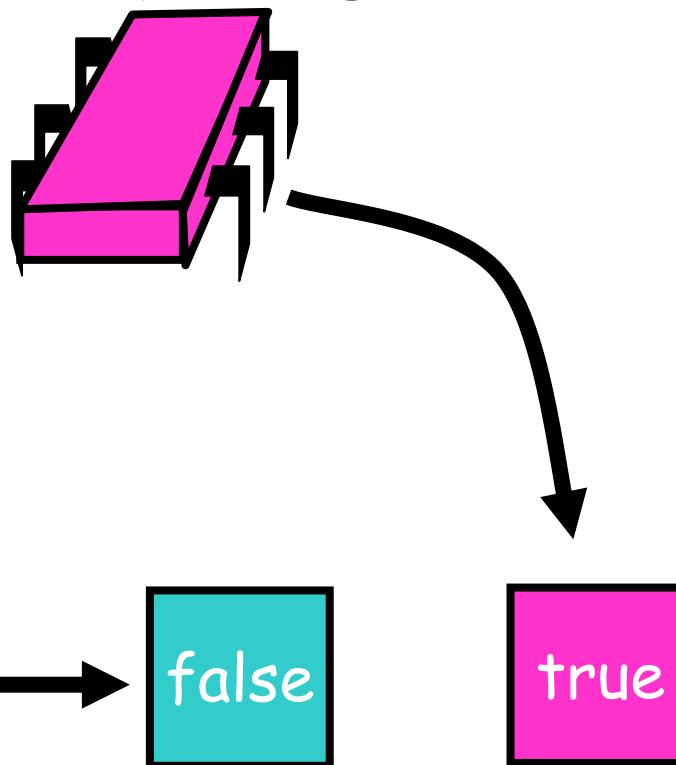


tail

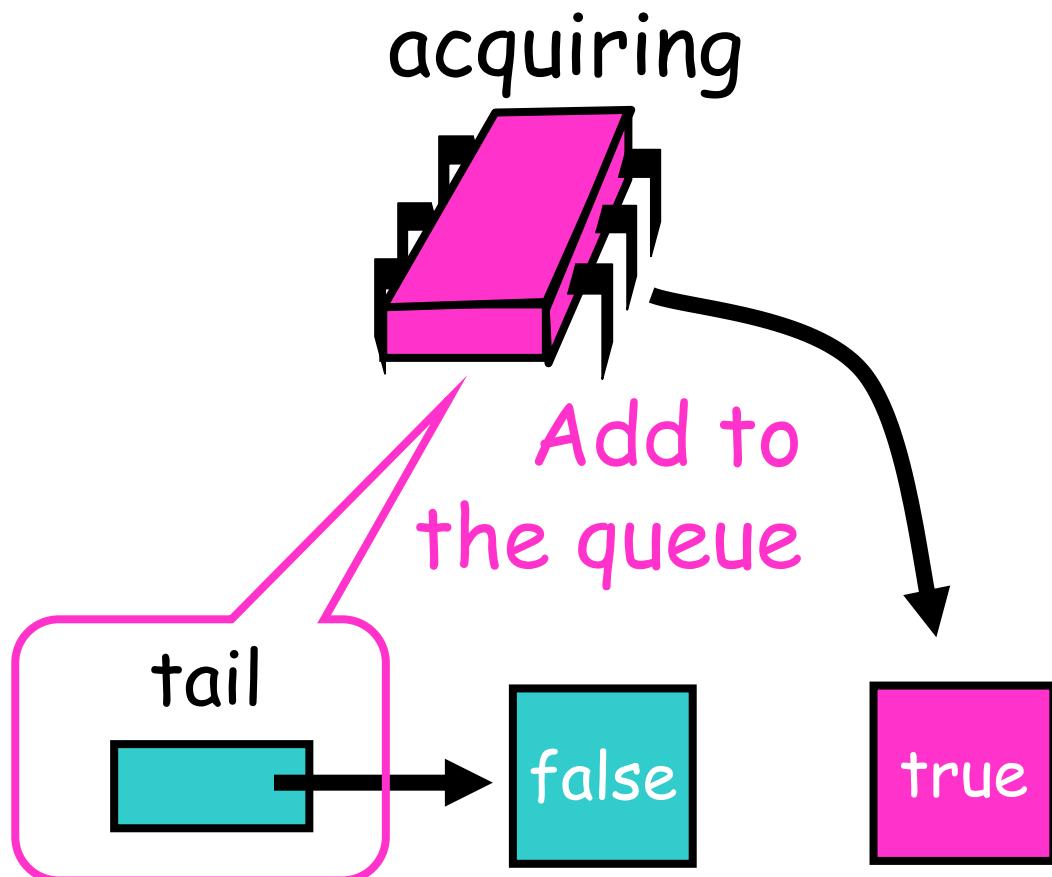


# Purple Wants the Lock

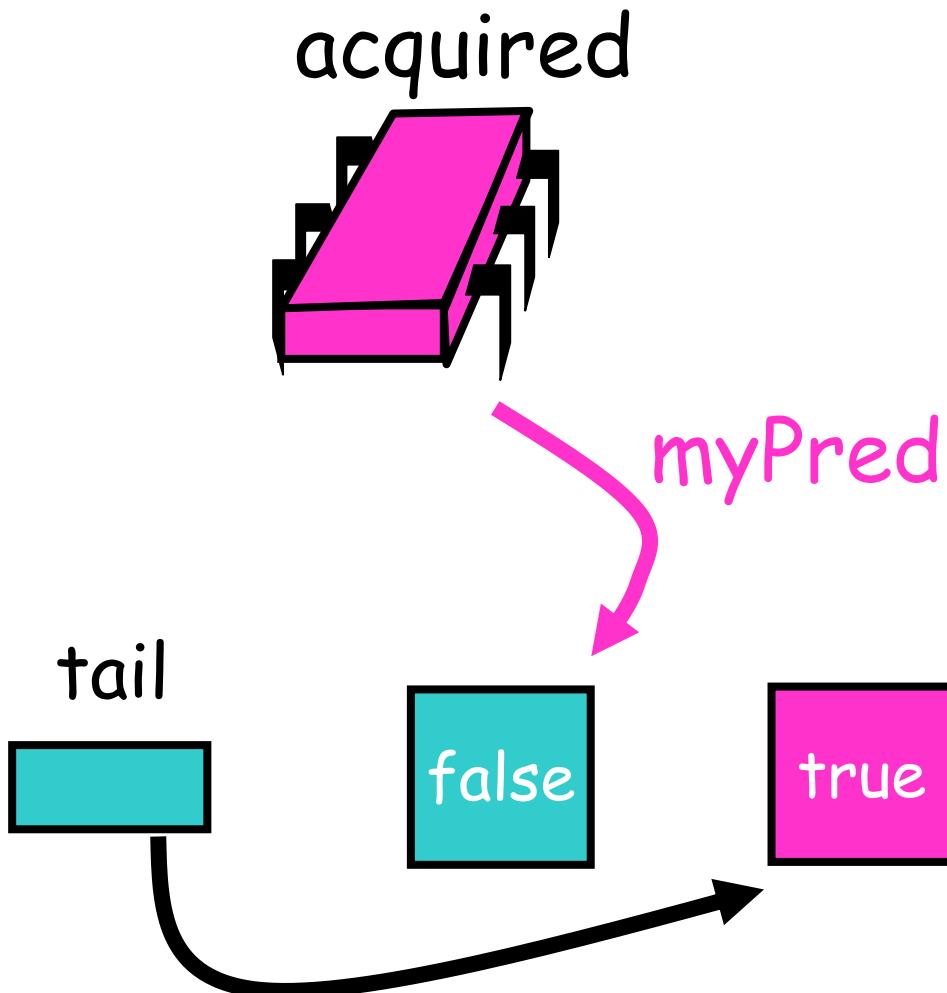
acquiring



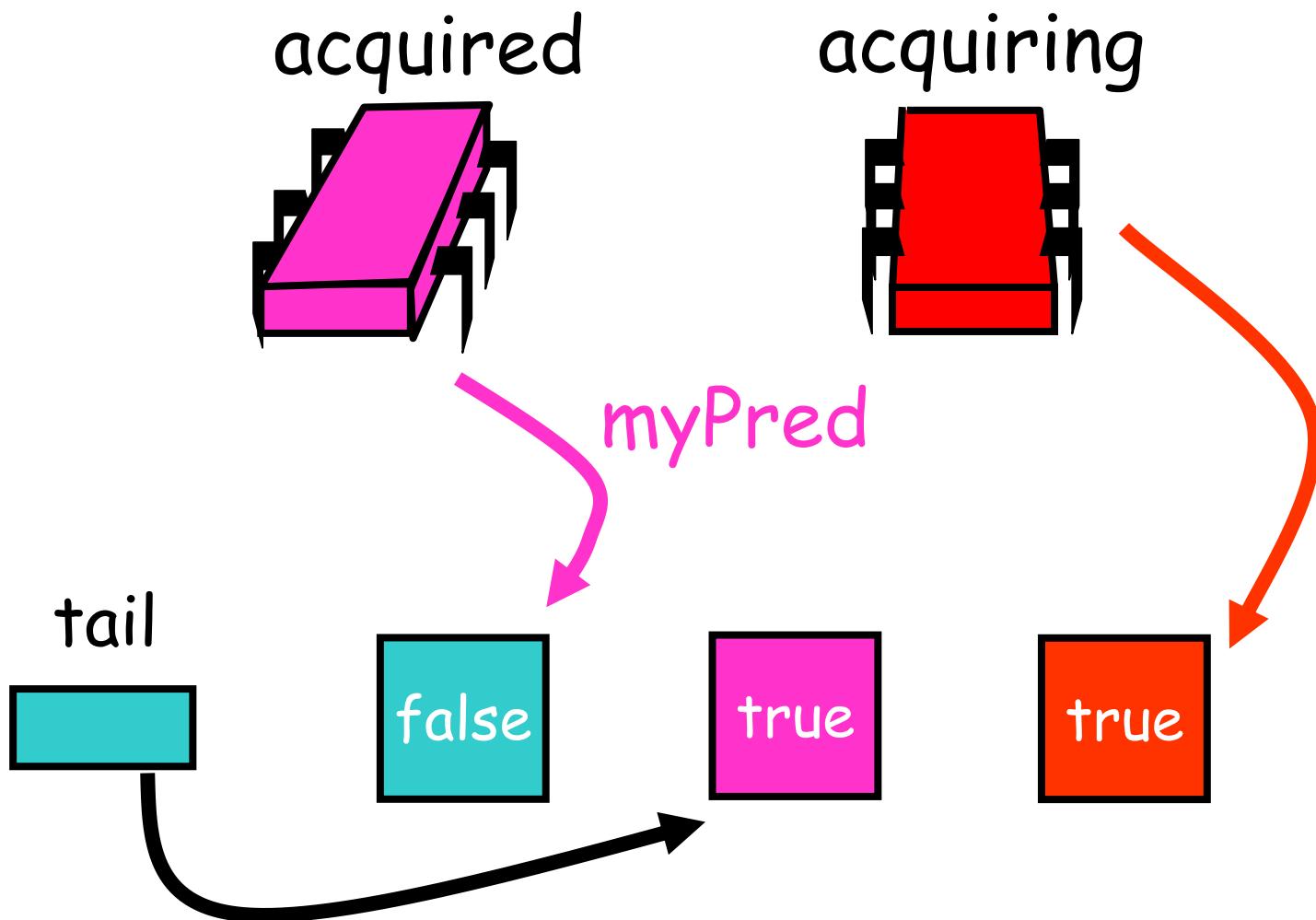
# Purple Wants the Lock



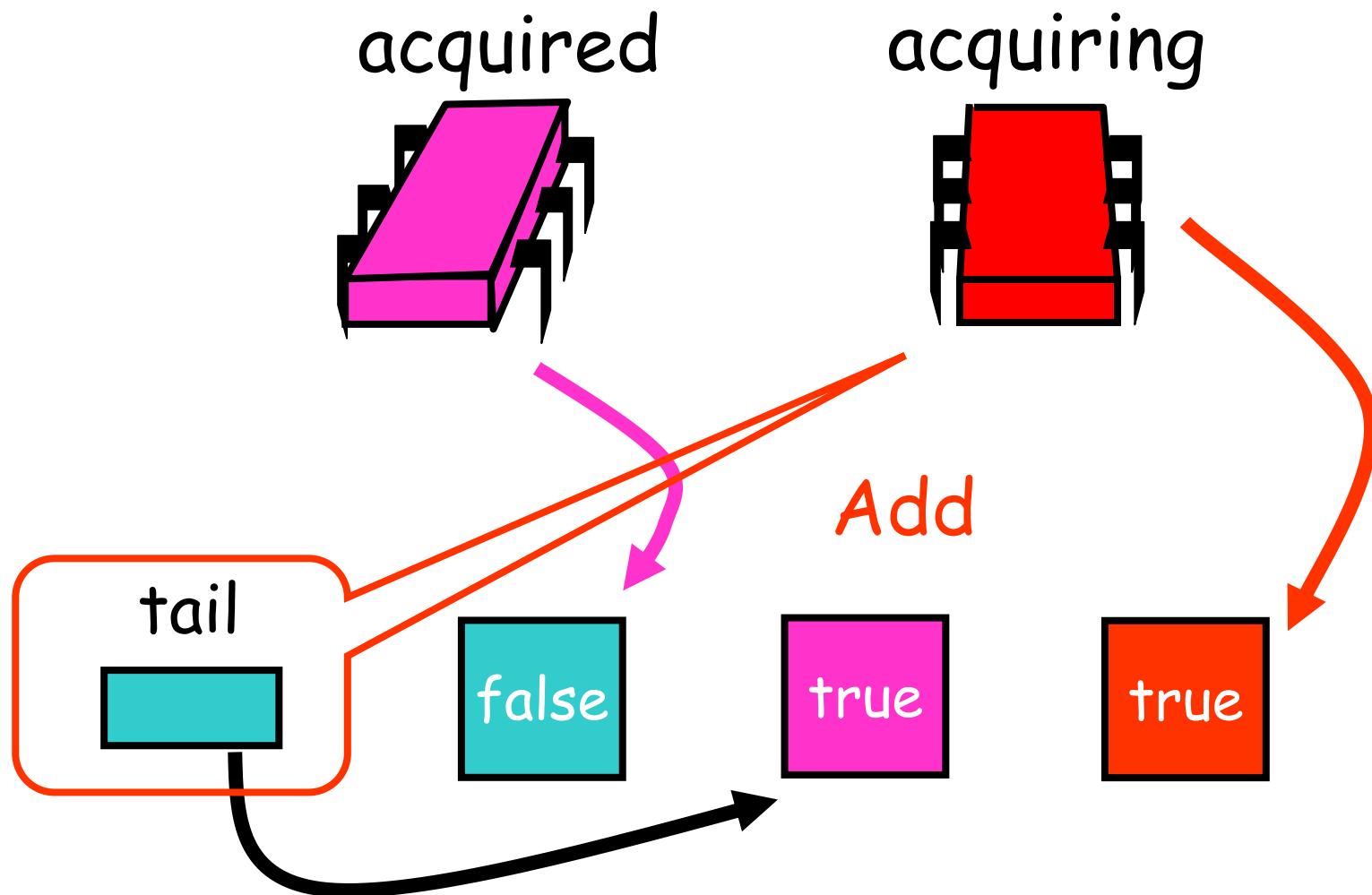
# Purple Has the Lock



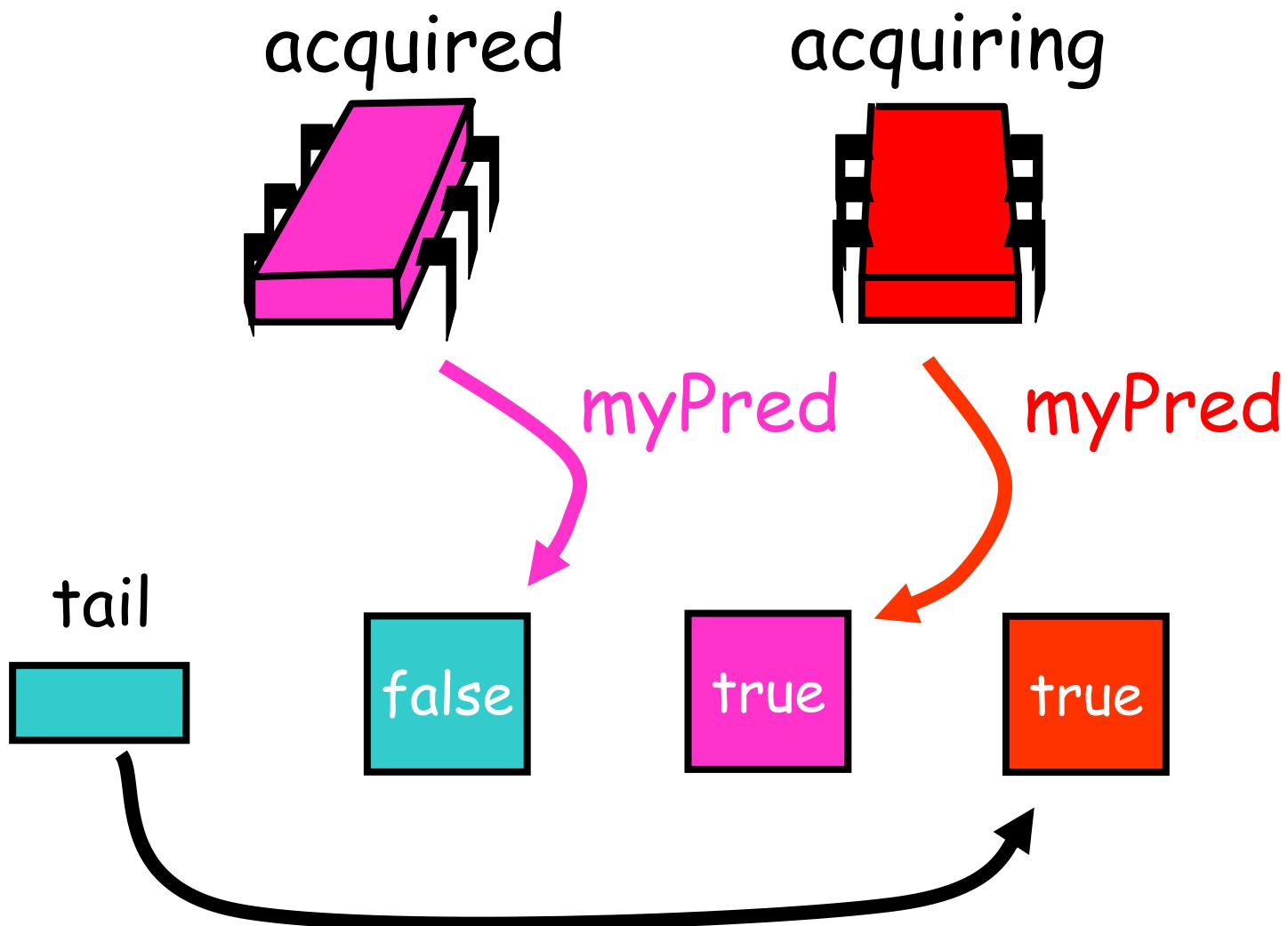
# Red Wants the Lock



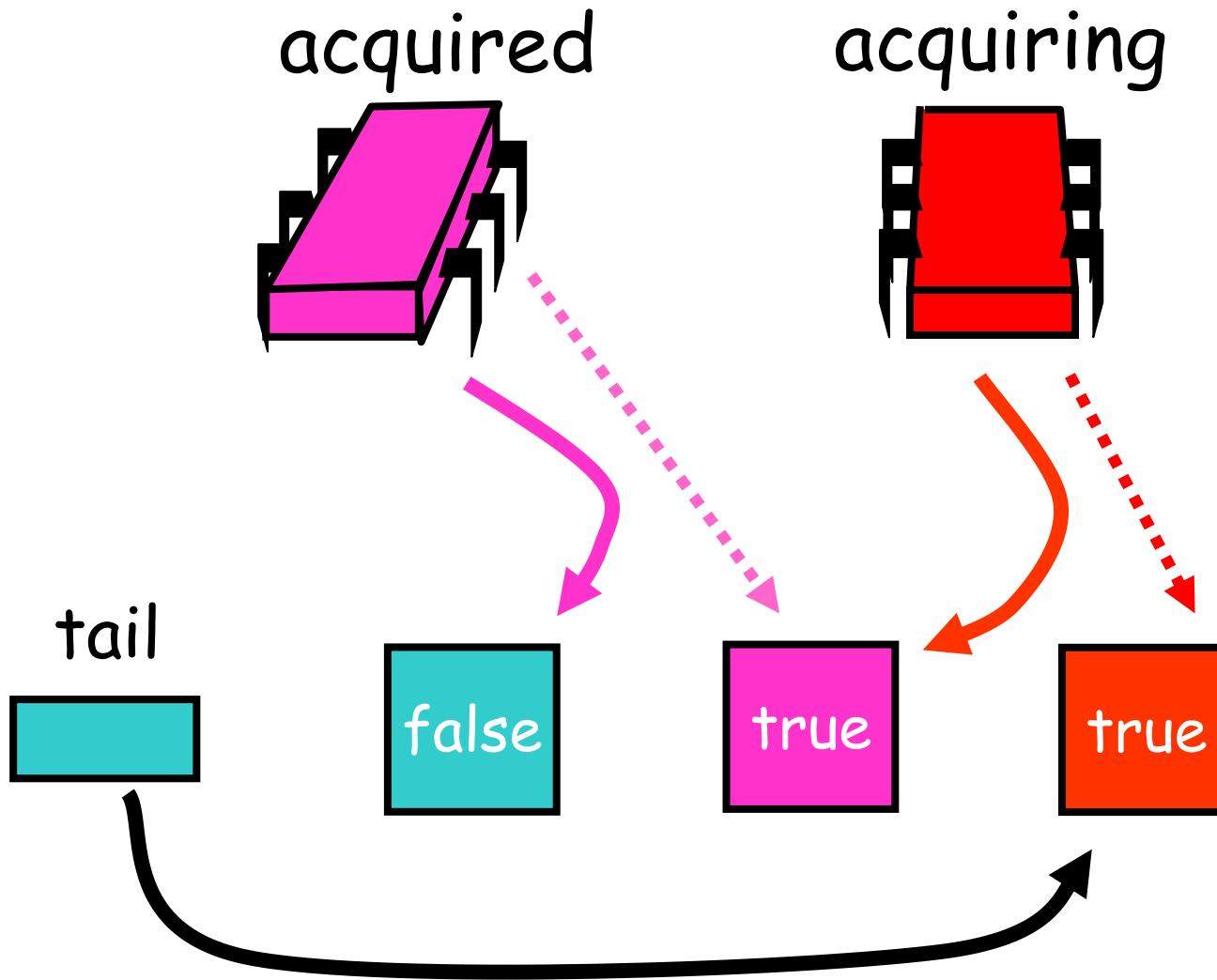
# Red Wants the Lock



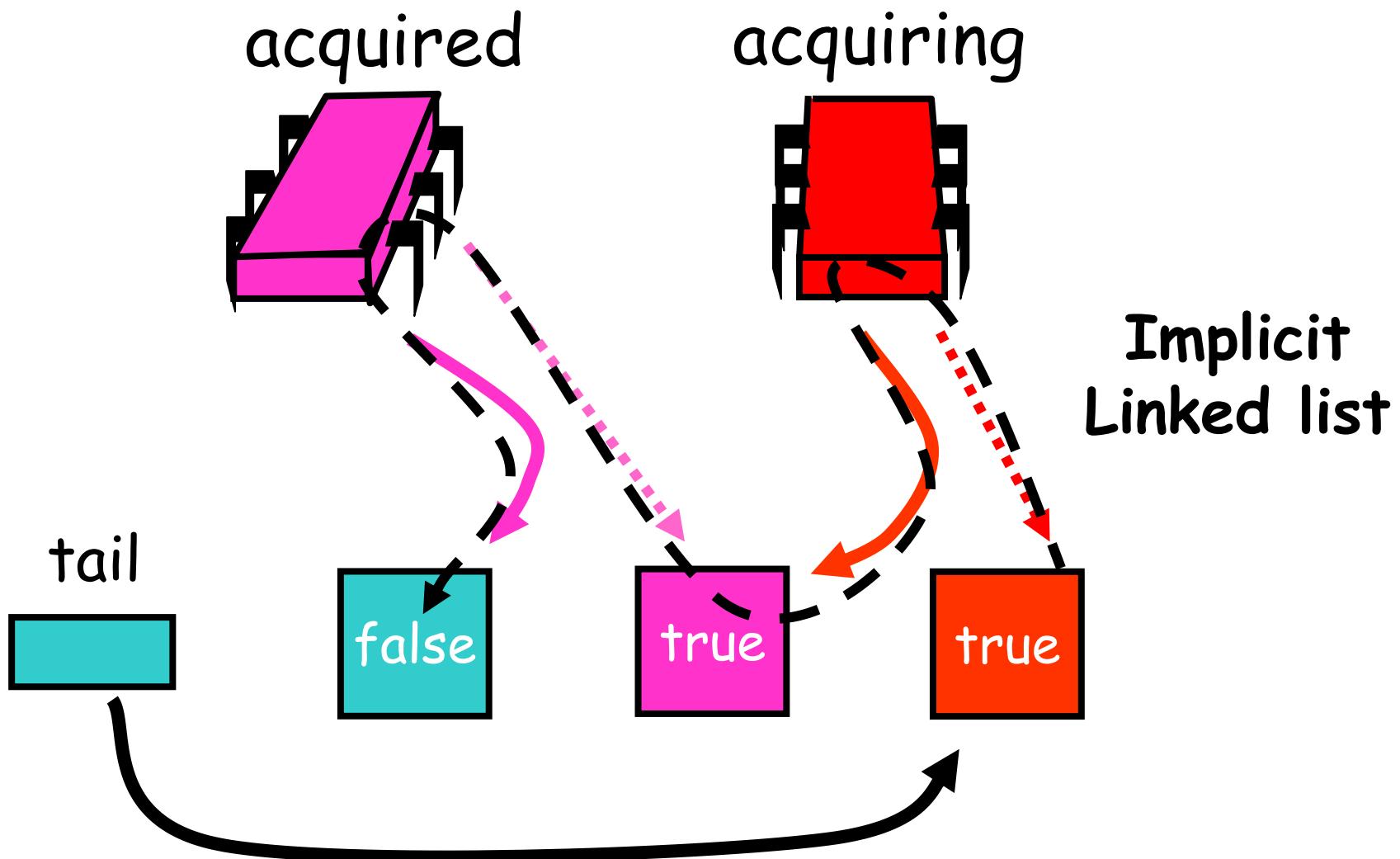
# Red Wants the Lock



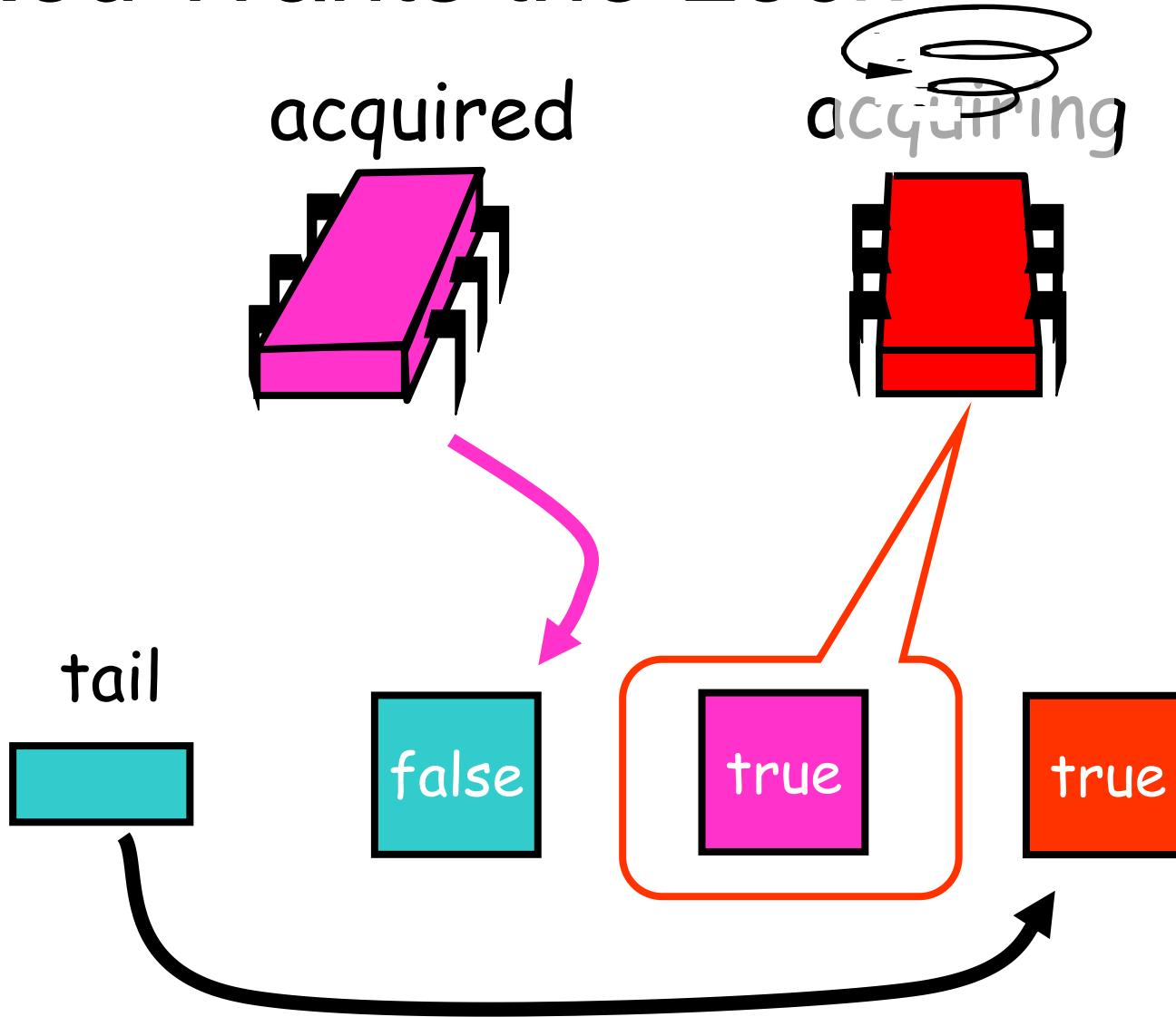
# Red Wants the Lock



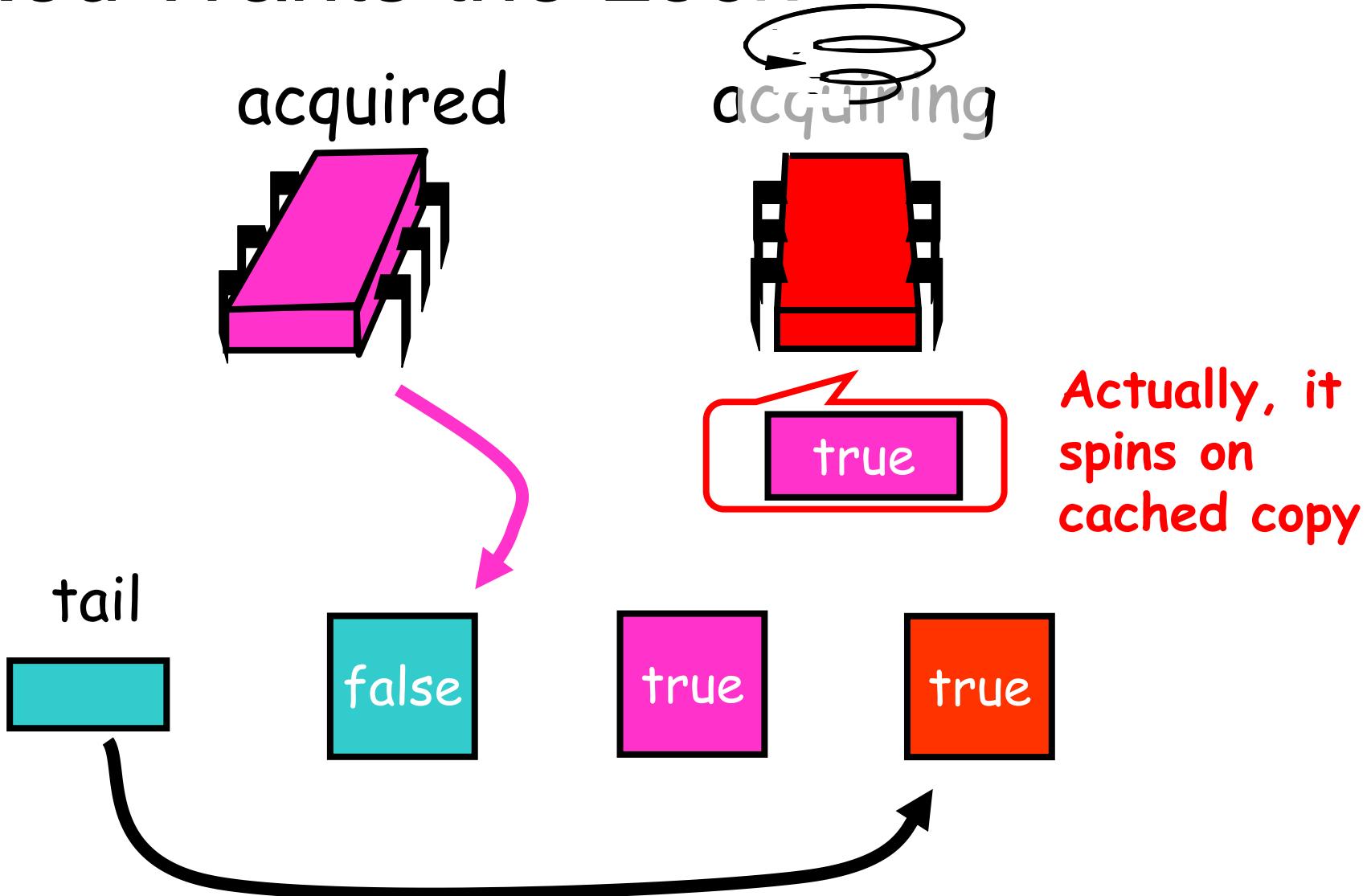
# Red Wants the Lock



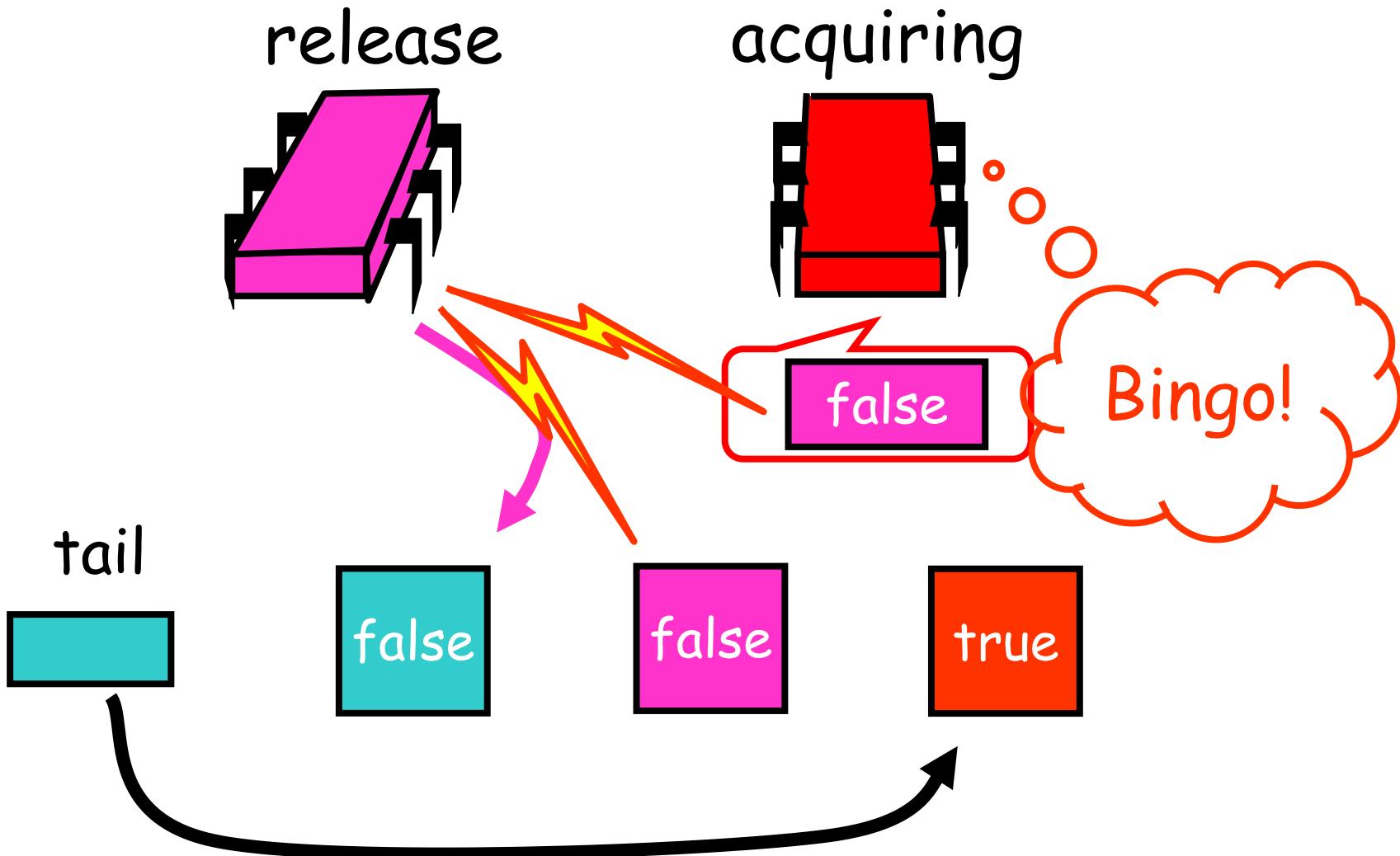
# Red Wants the Lock



# Red Wants the Lock

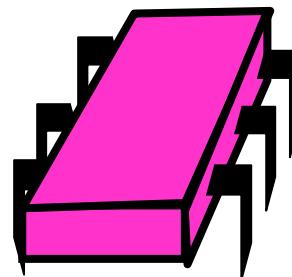


# Purple Releases

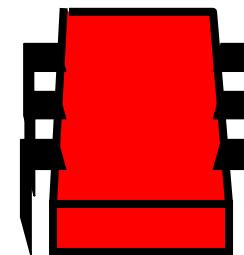


# Purple Releases

released



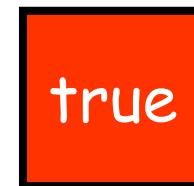
acquired



tail



true



# CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        myNode.locked = true;  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

# CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        myNode.locked = true;  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

Queue tail

# CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        myNode.locked = true;  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) ;  
    }  
}
```

Thread-local Qnode

# CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();  
    public void lock() {  
        myNode.locked = true;  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

Add my node

# CLH Queue Lock

```
class CLHLock implements Lock {  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode  
        = new Qnode();           Spin until predecessor  
    public void lock() {          releases lock  
        myNode.locked = true;  
        Qnode pred  
            = tail.getAndSet(myNode);  
        while (pred.locked) {}  
    }  
}
```

# CLH Queue Lock

```
class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

# CLH Queue Lock

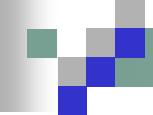
```
class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

Notify successor

# CLH Queue Lock

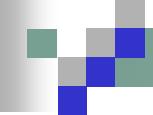
```
class CLHLock implements Lock {  
    ...  
    public void unlock() {  
        myNode.locked.set(false);  
        myNode = pred;  
    }  
}
```

Recycle  
predecessor's node



# CLH Lock

- Good
  - Lock release affects predecessor only
  - Small, constant-sized space



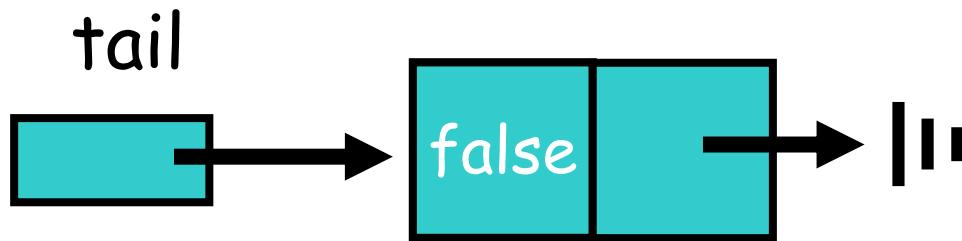
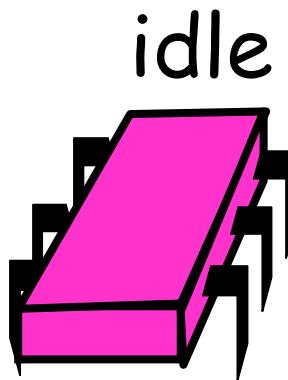
# MCS Lock

- FIFO order
- Spin on local memory only
- Small, Constant-size overhead

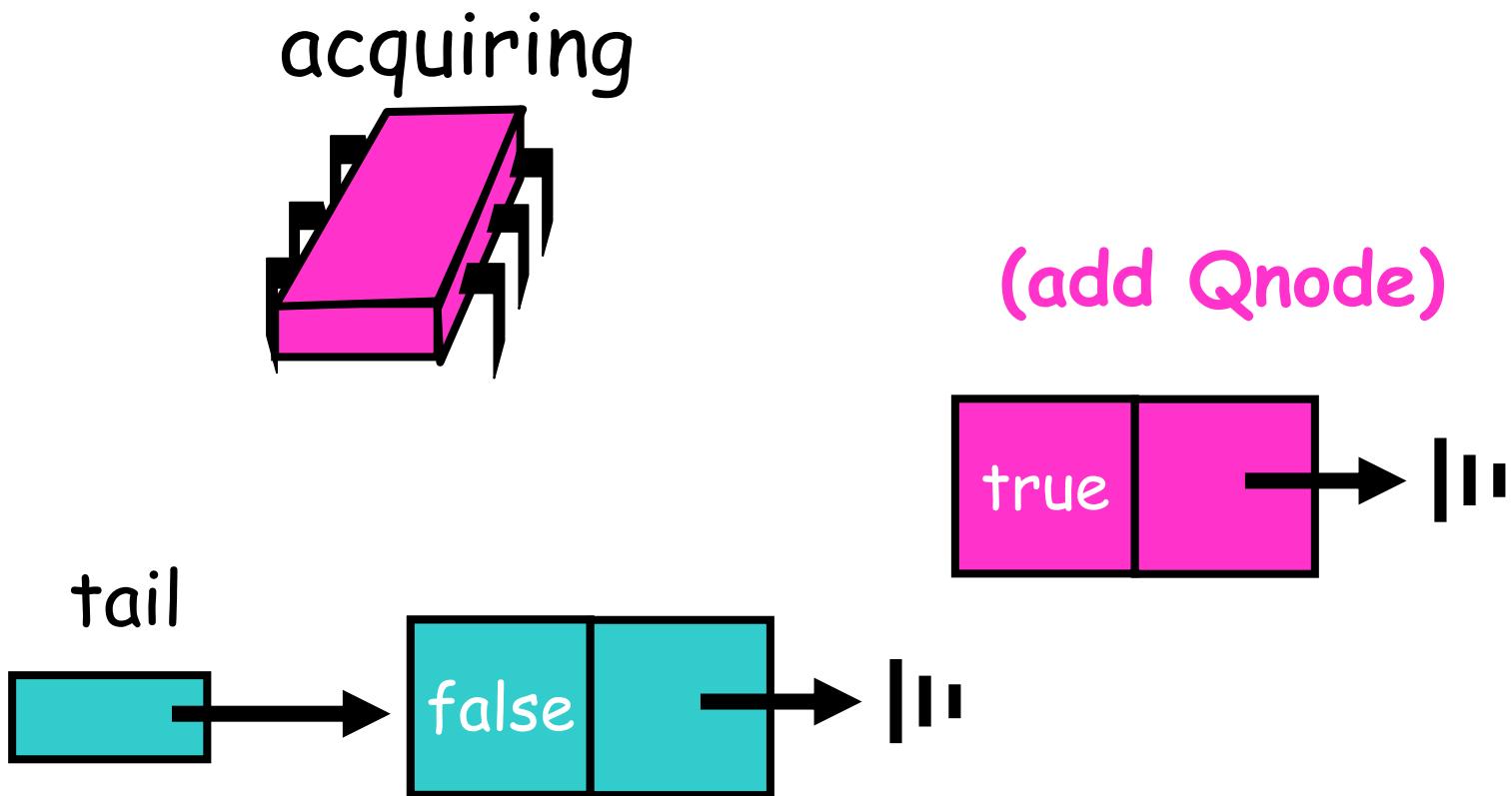
# MCS Queue Lock

- Similar to CLHLock, but the linked list is explicit instead of implicit
- Each node in the Queue has a next field

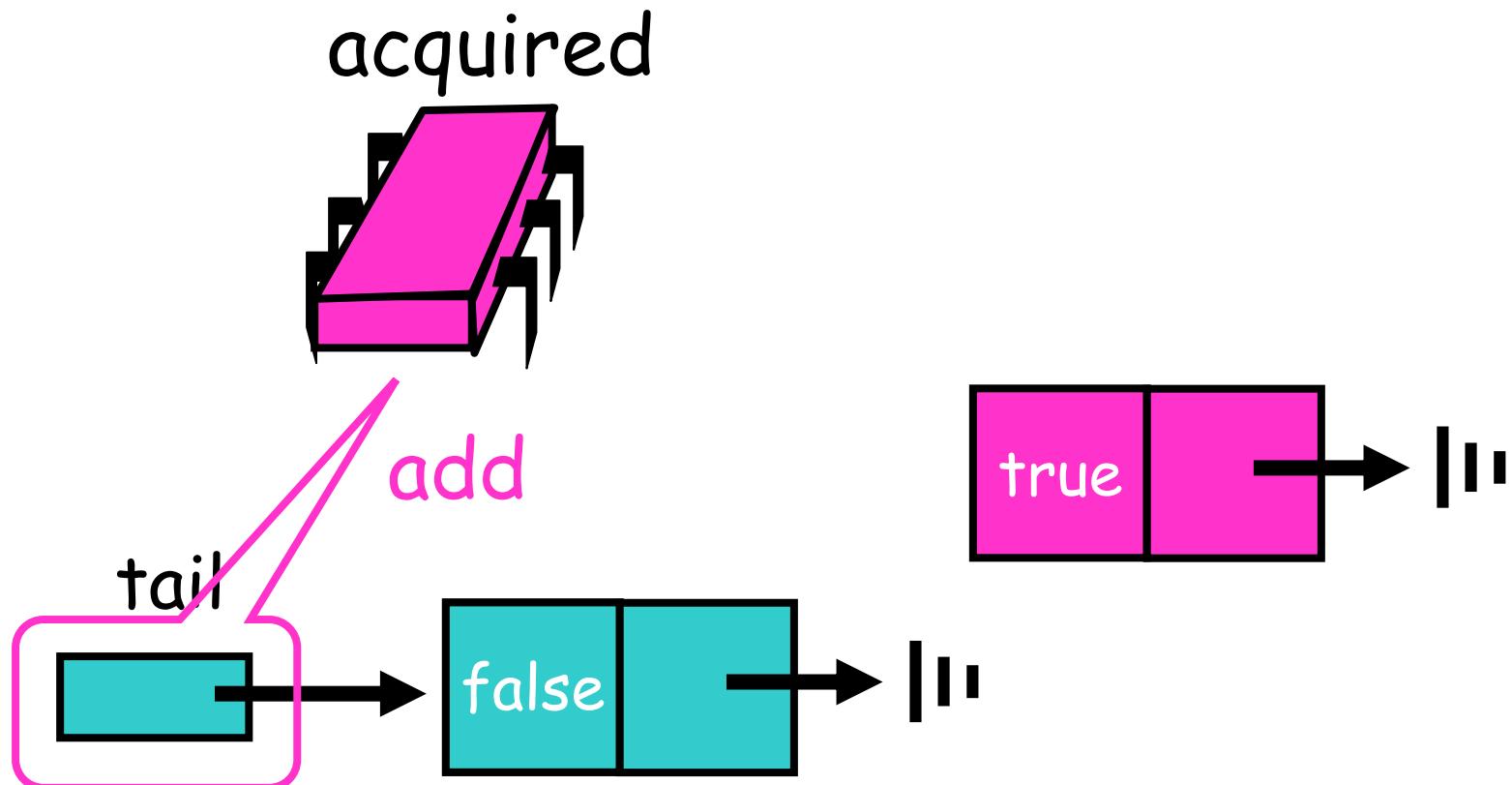
# Initially



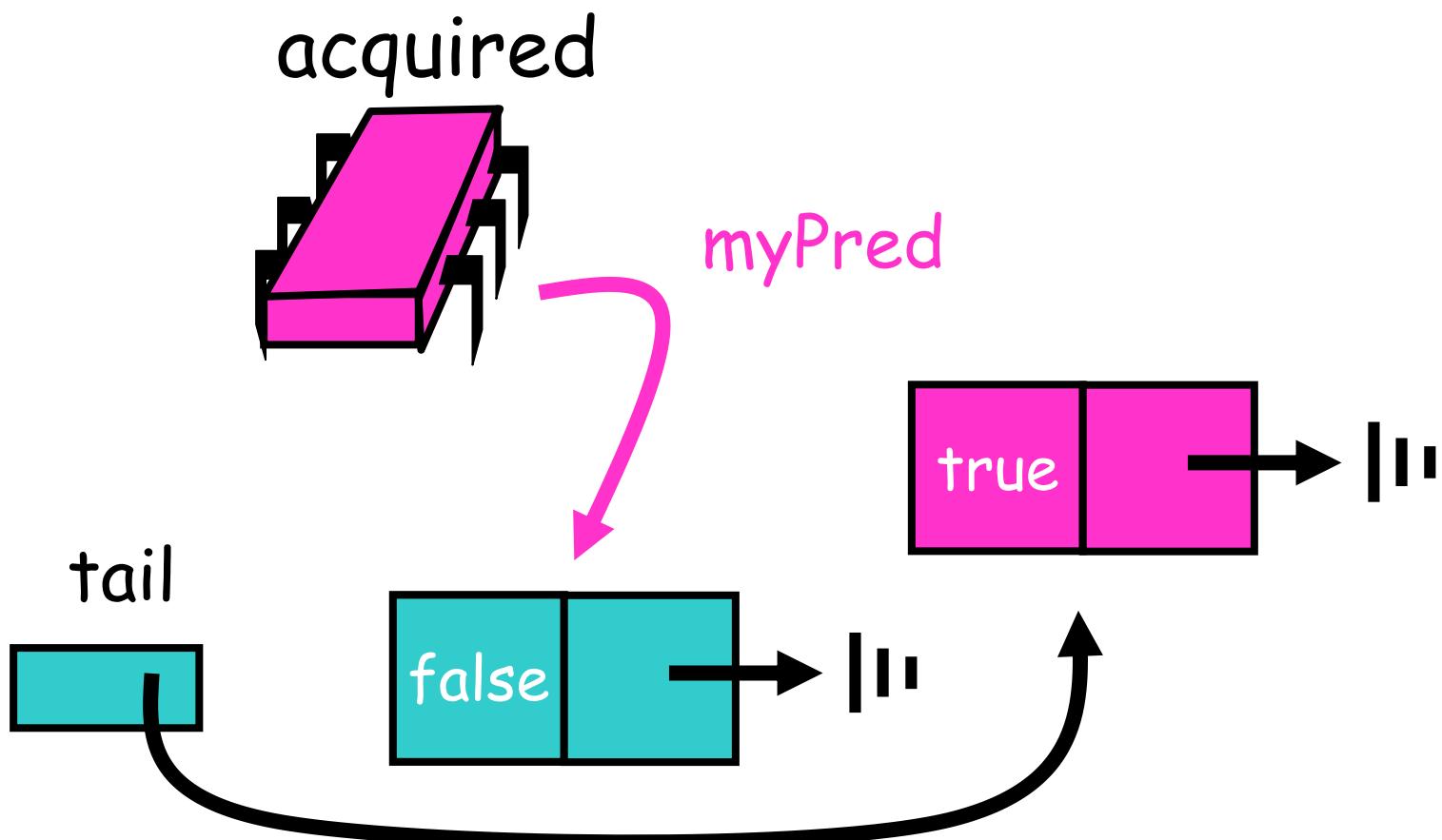
# Acquiring



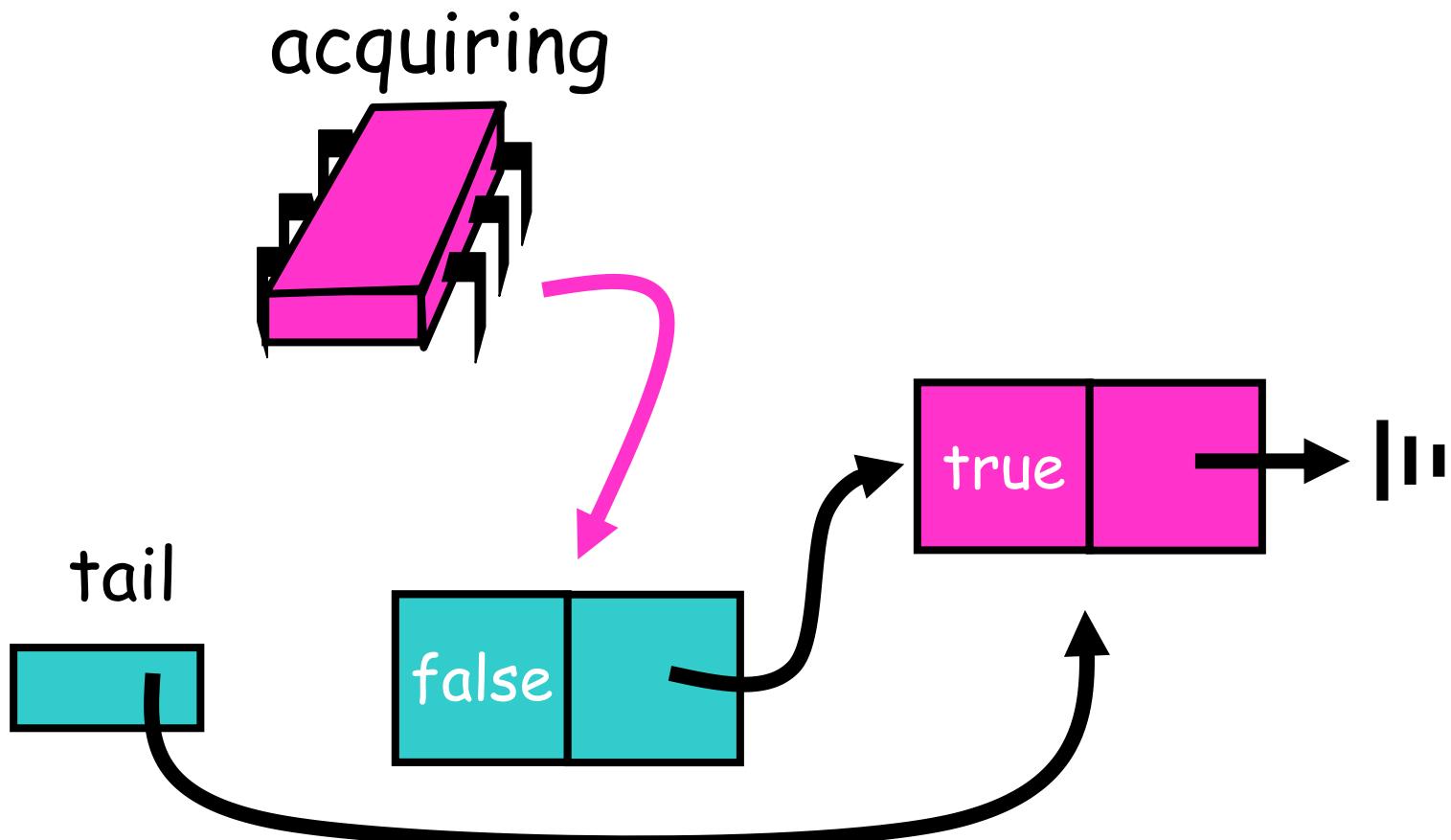
# Acquiring



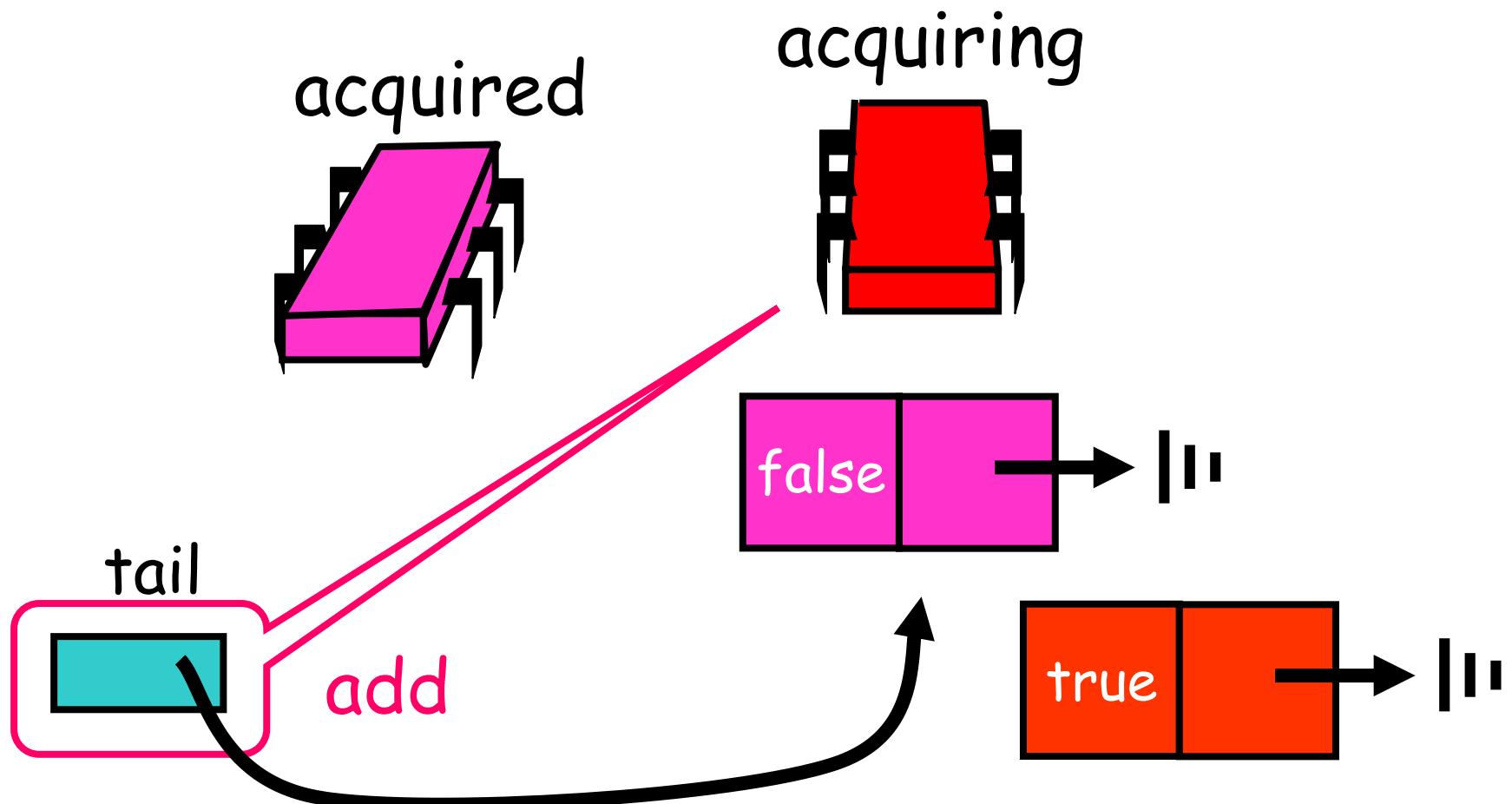
# Acquiring



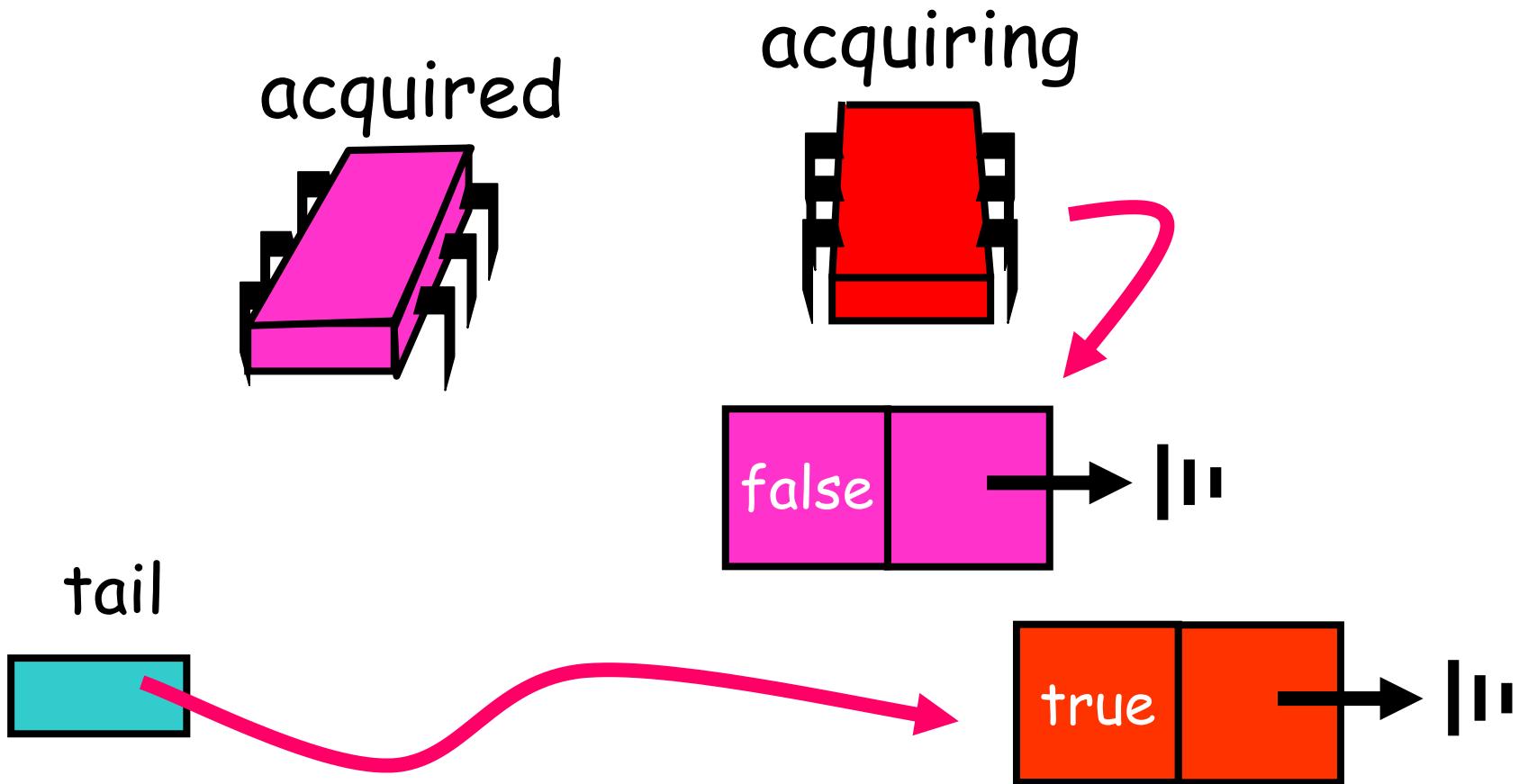
# Acquiring



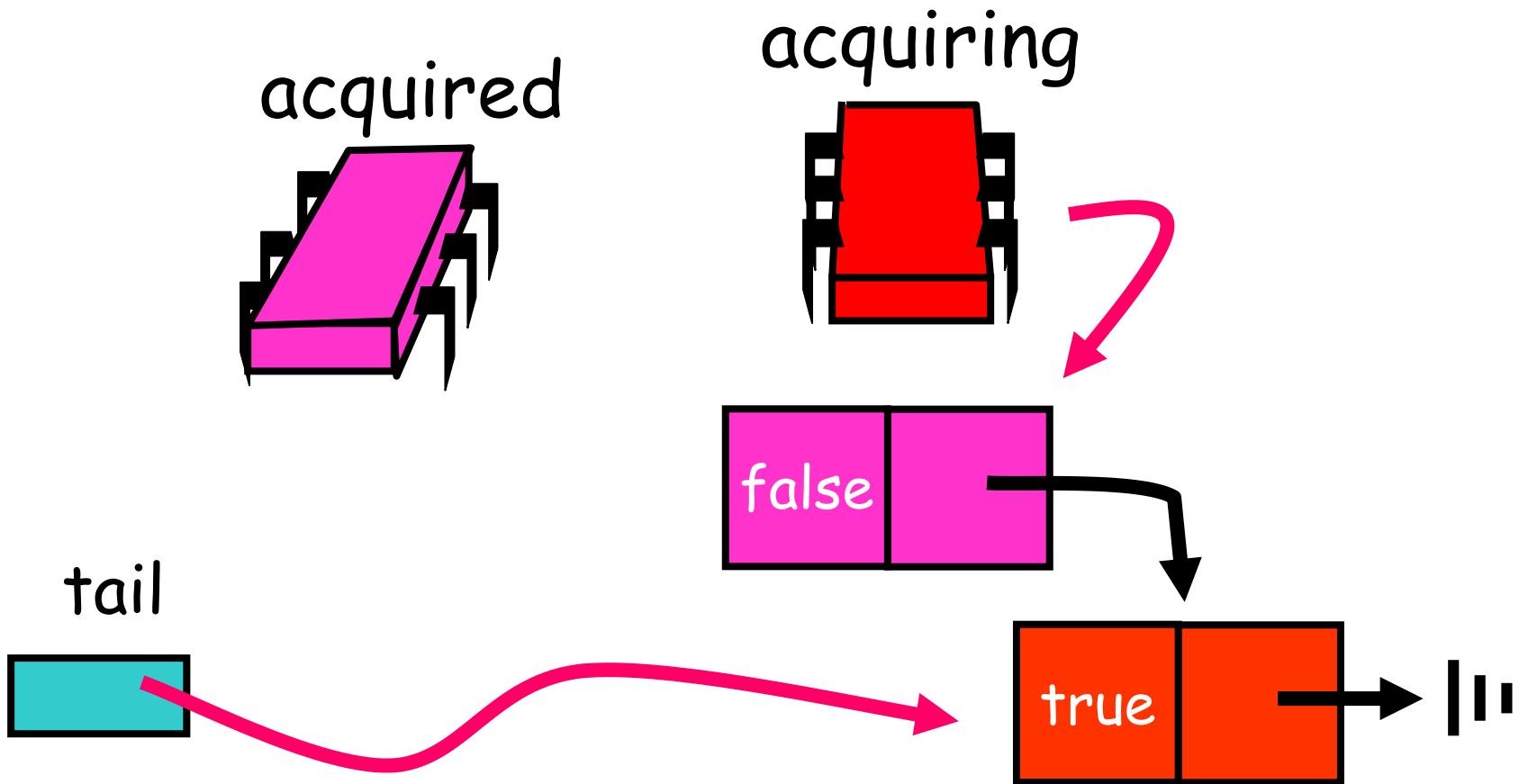
# Acquiring



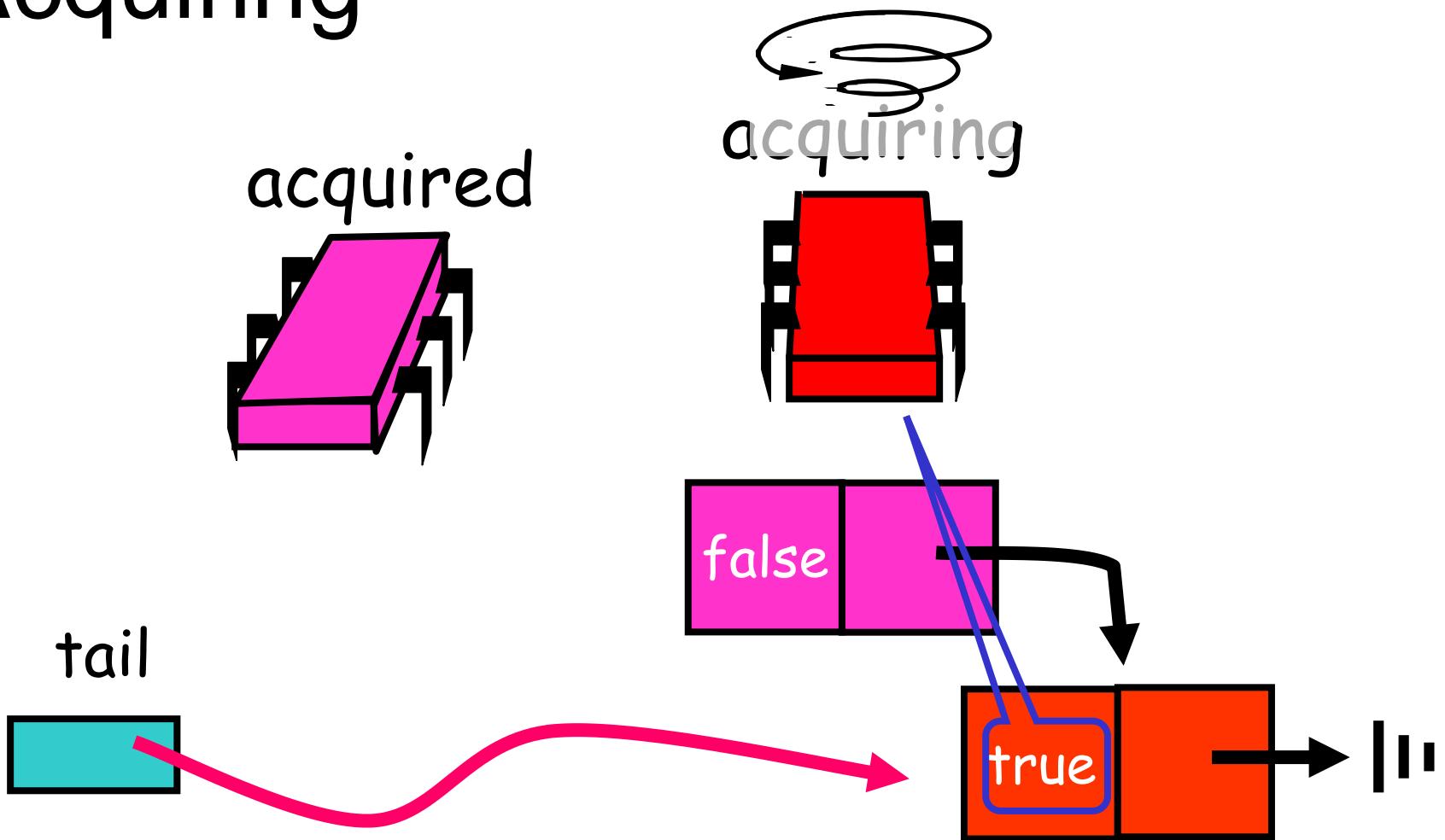
# Acquiring



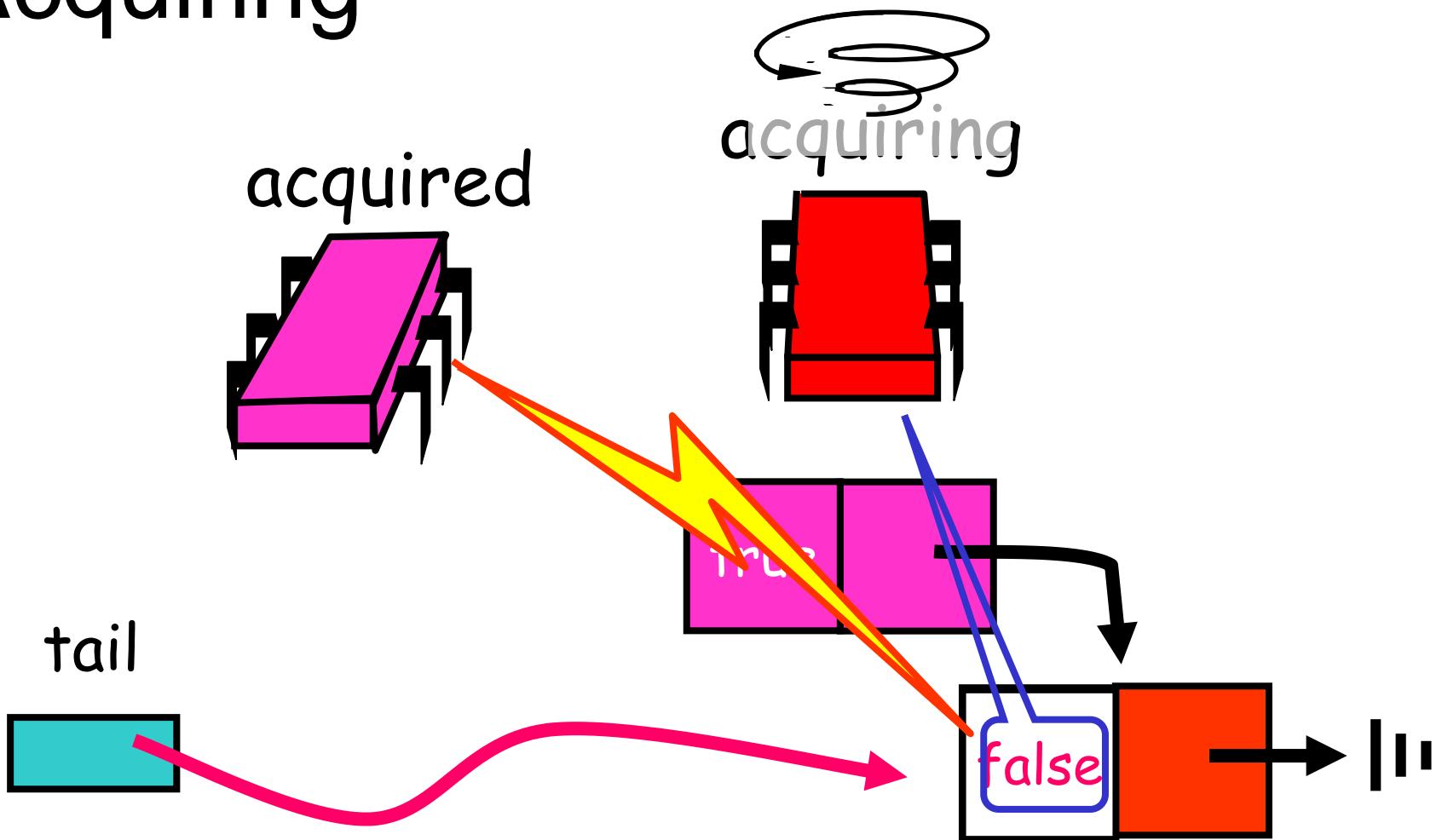
# Acquiring



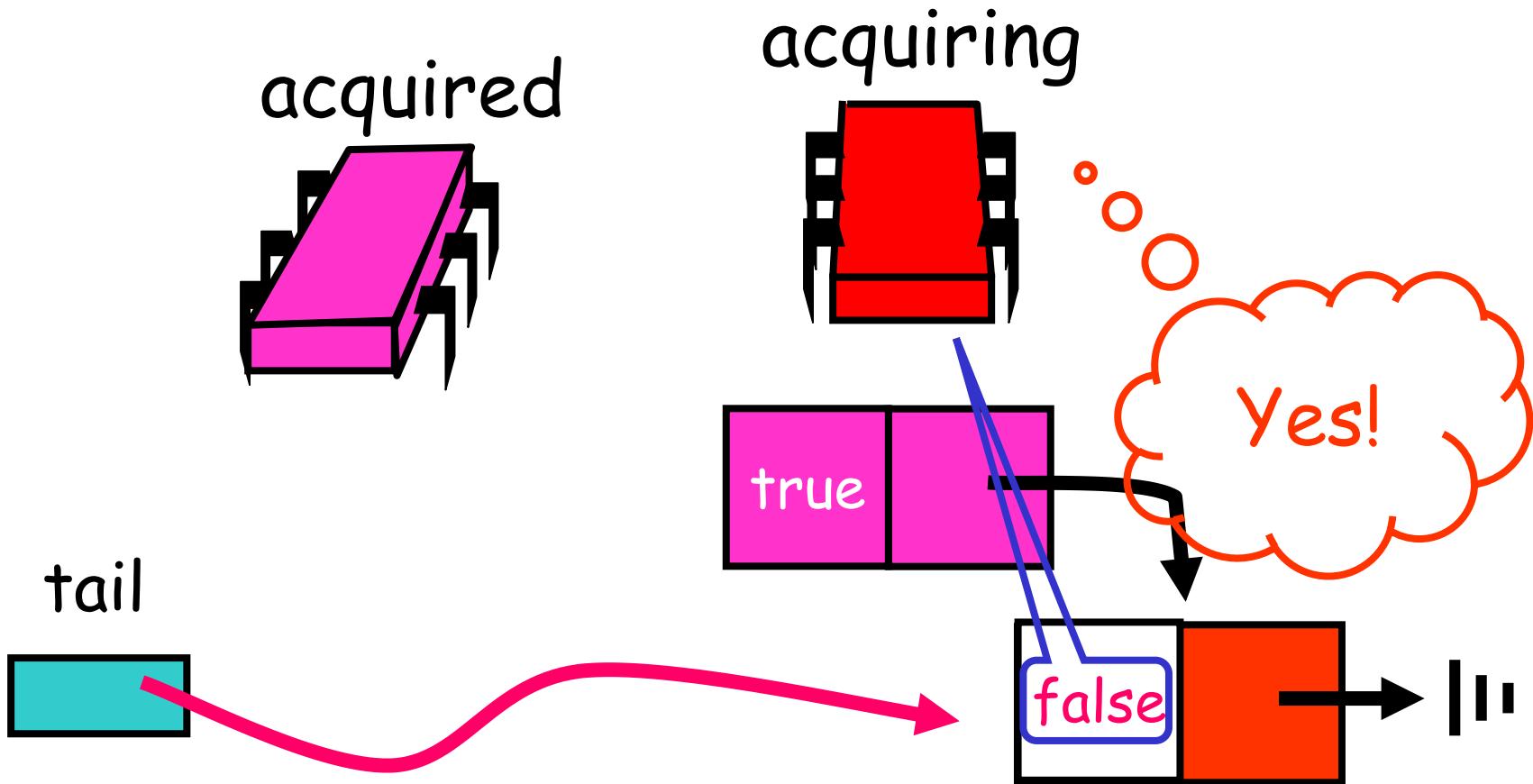
# Acquiring



# Acquiring



# Acquiring



# MCS Queue Lock

```
class Qnode {  
    boolean locked = false;  
    Qnode next = null;  
}
```

# MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        }  
    }  
}
```

# MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        }  
    }  
}
```

Make a  
QNode

# MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void lock() {  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true; add my Node to  
            pred.next = qnode;      the tail of  
            while (qnode.locked) {} queue  
        }  
    }  
}
```

# MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference tail;          Fix if queue  
    public void lock() {           was non-empty  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        }  
    }  
}
```

# MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference tail;      Wait until  
    public void lock() {        unlocked  
        Qnode qnode = new Qnode();  
        Qnode pred = tail.getAndSet(qnode);  
        if (pred != null) {  
            qnode.locked = true;  
            pred.next = qnode;  
            while (qnode.locked) {}  
        }  
    }  
}
```

# MCS Queue Unlock

```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```

# MCS Queue Lock

```
class MCSLock implements Lock {  
    AtomicReference tail;  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null)  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false; } }  
Missing  
successor?
```

# MCS Queue Lock

If really no successor, :k {  
    return;

```
public void unlock() {
    if (qnode.next == null) {
        if (tail.CompareAndSet(qnode,
null)
            return;
        while (qnode.next == null) {}
    }
    qnode.next.locked = false;
}}
```

# MCS Queue Lock

Otherwise wait for successor to catch up

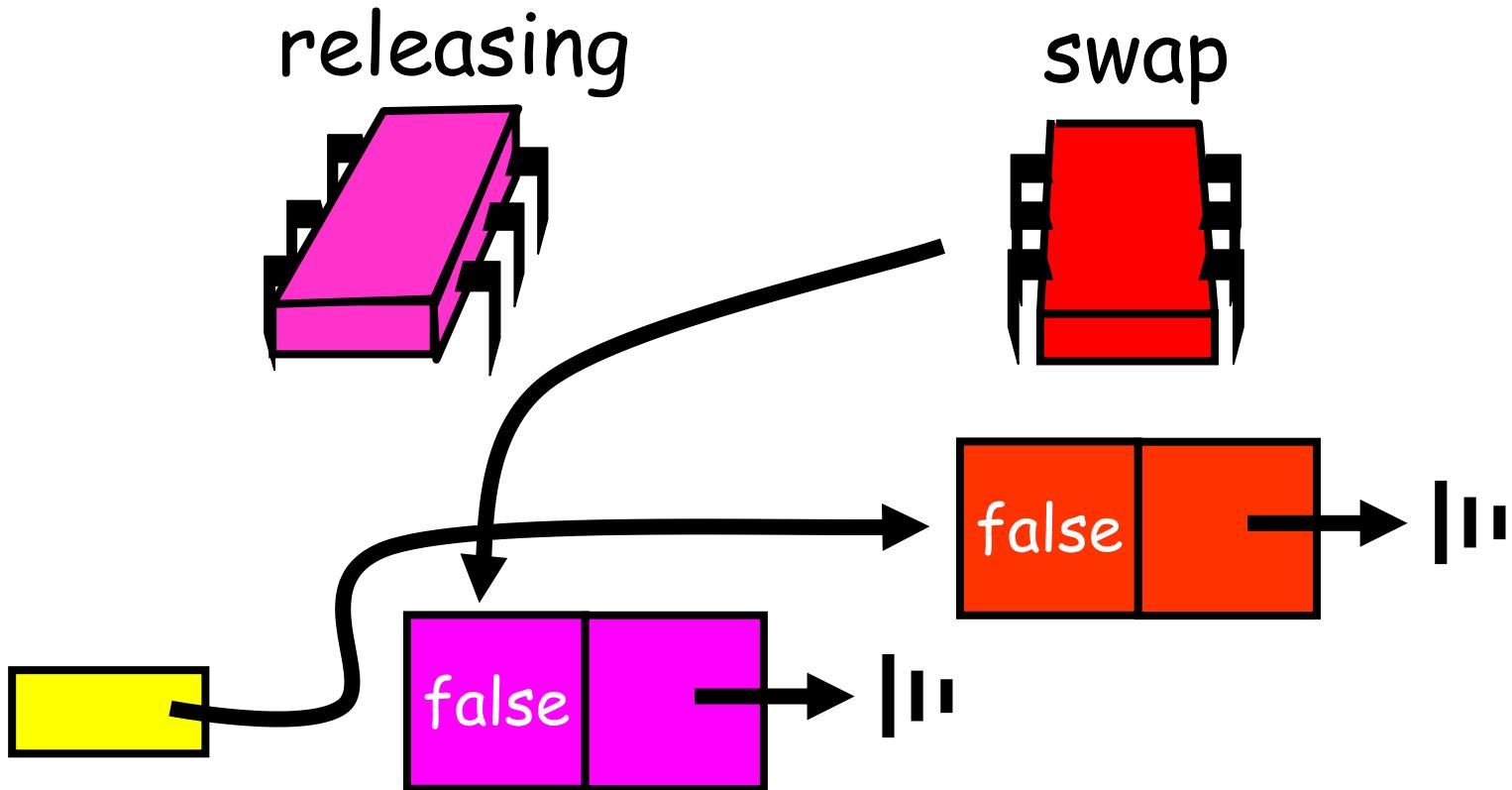
```
public void unlock() {
    if (qnode.next == null) {
        if (tail.CAS(qnode, null)
            return;
    while (qnode.next == null) {}
}
qnode.next.locked = false;
}}
```

# MCS Queue Lock

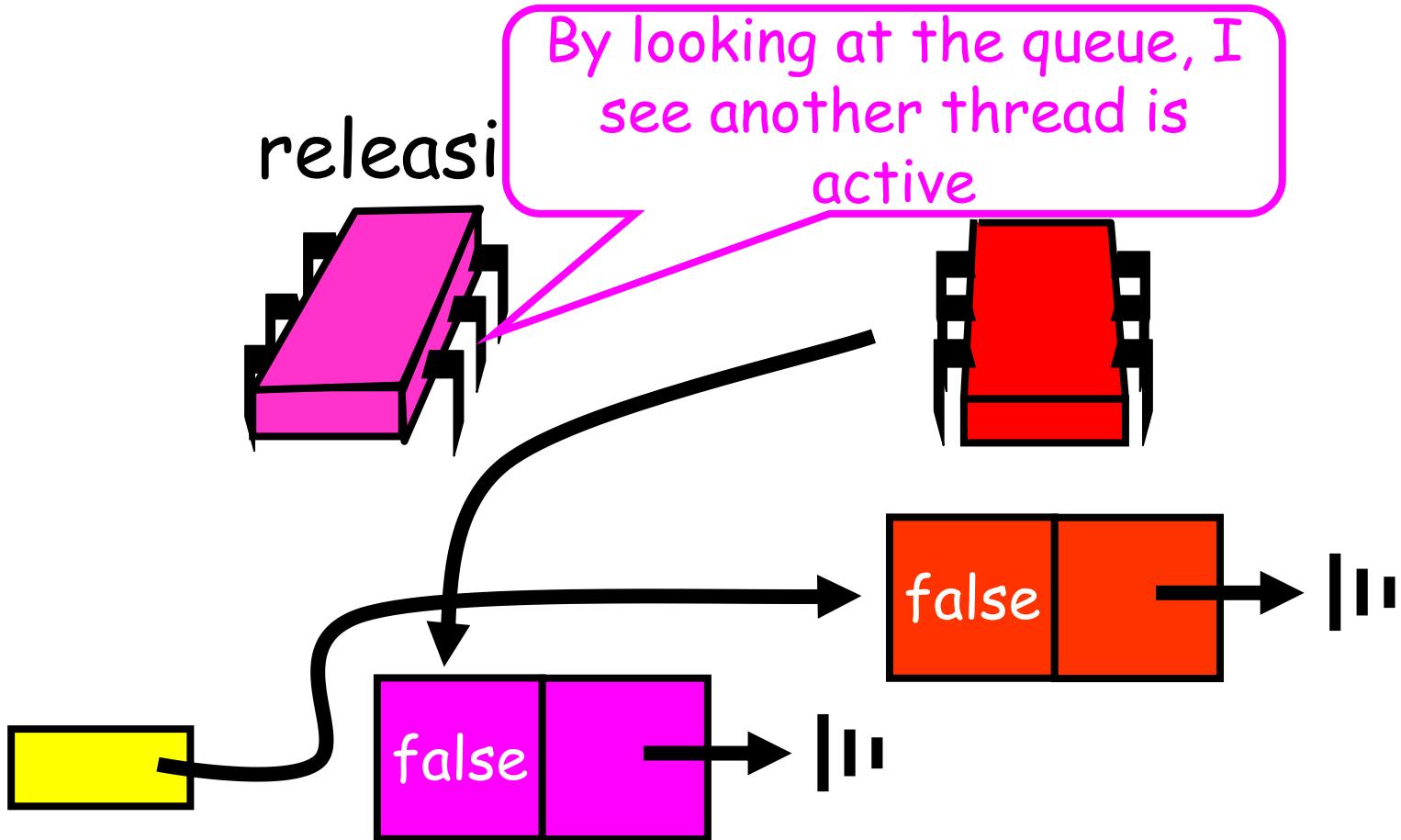
```
class MCSLock implements Lock {  
    AtomicReference<QNode> tail = new AtomicReference<QNode>(new QNode());  
    public void unlock() {  
        if (qnode.next == null) {  
            if (tail.CAS(qnode, null))  
                return;  
            while (qnode.next == null) {}  
        }  
        qnode.next.locked = false;  
    }  
}
```

Pass lock to successor

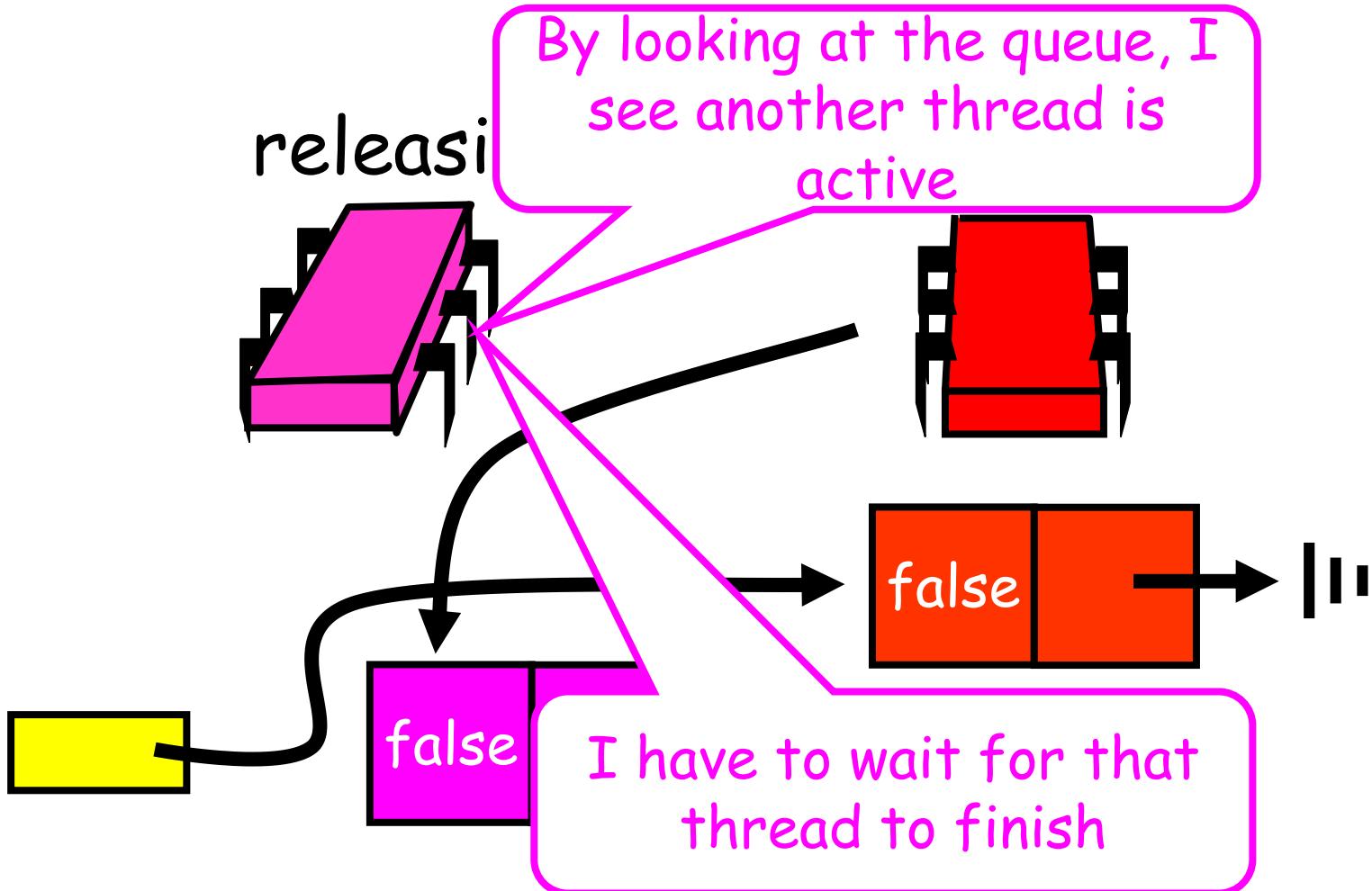
# Purple Release



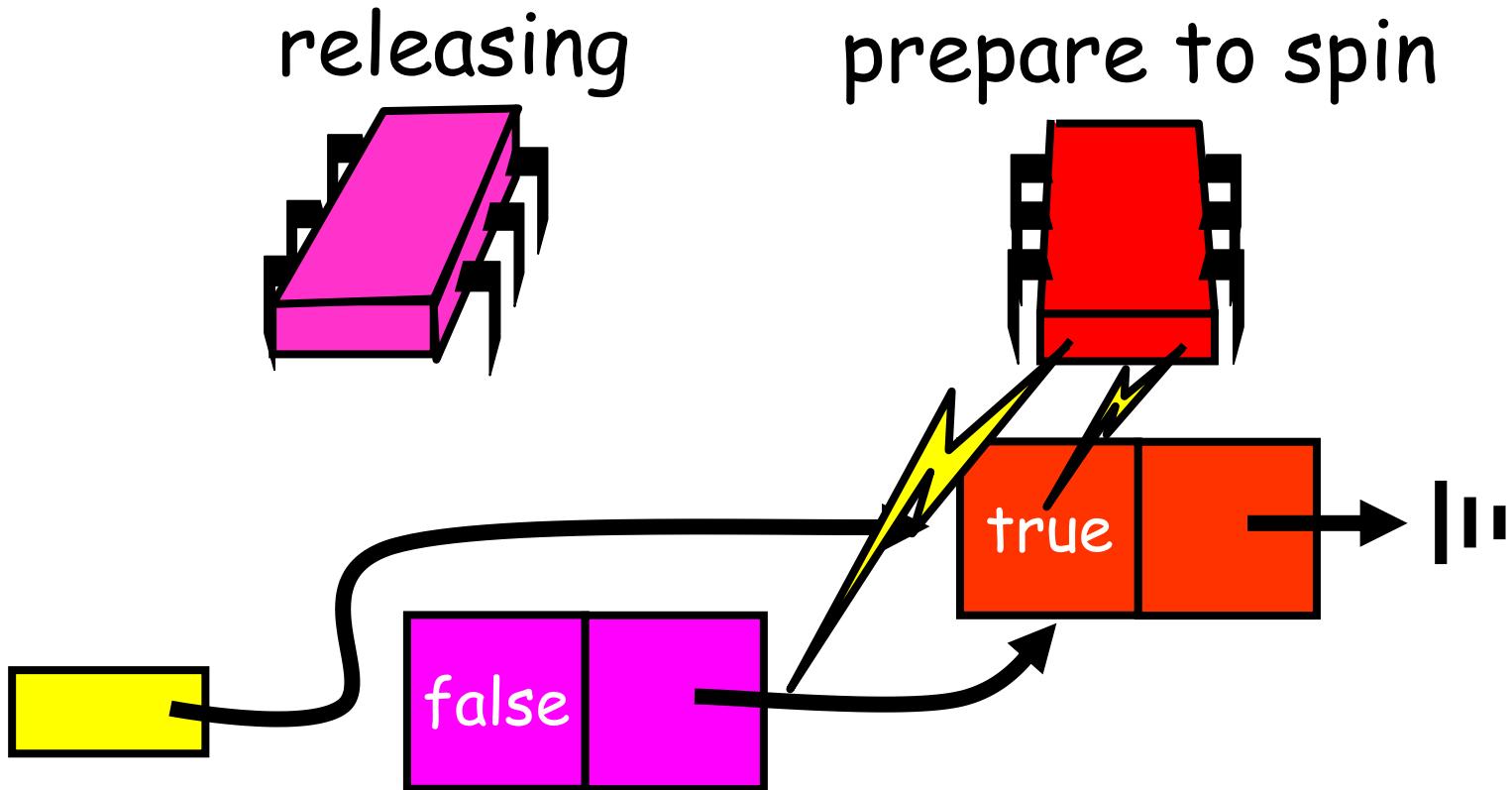
# Purple Release



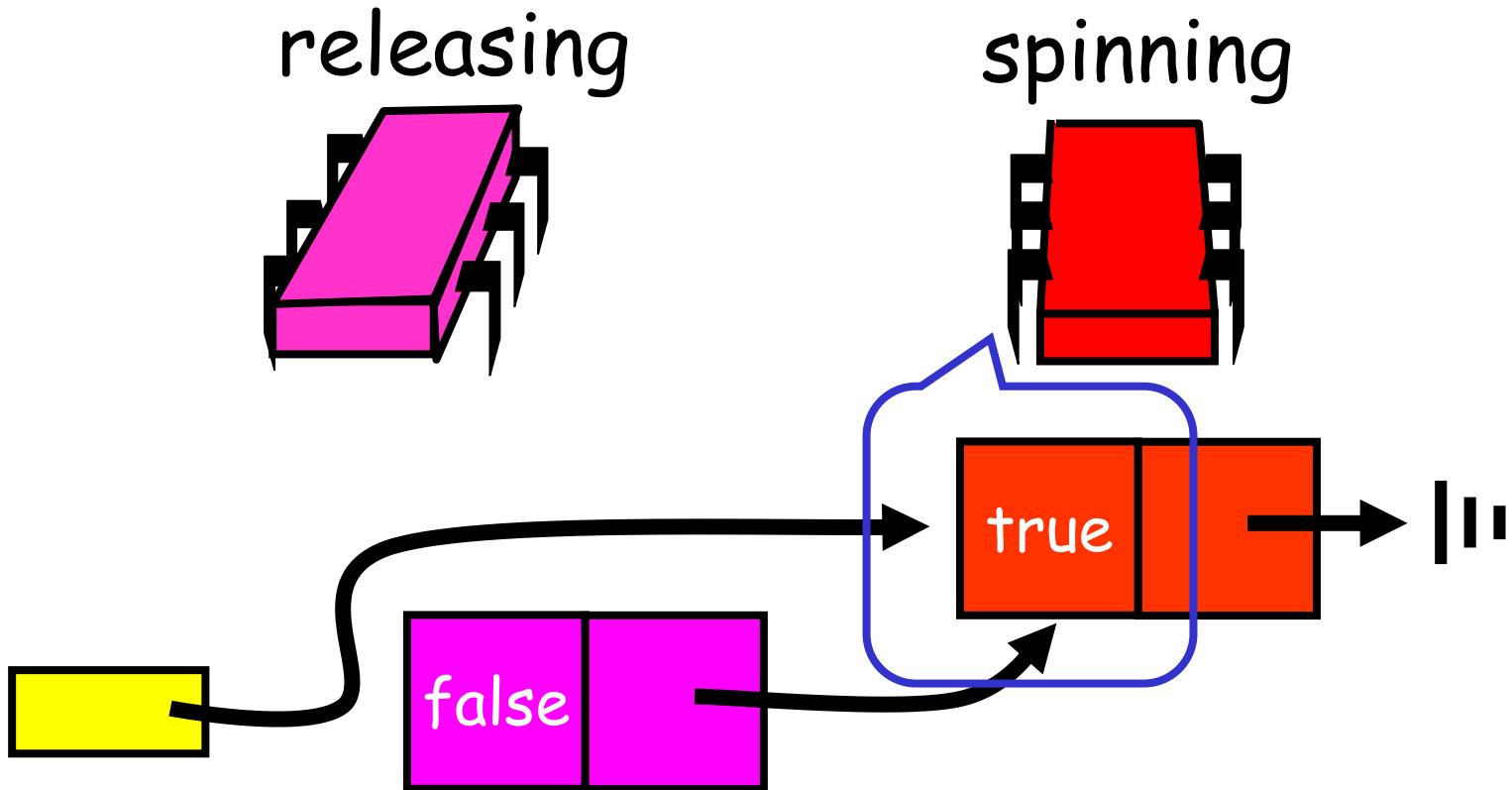
# Purple Release



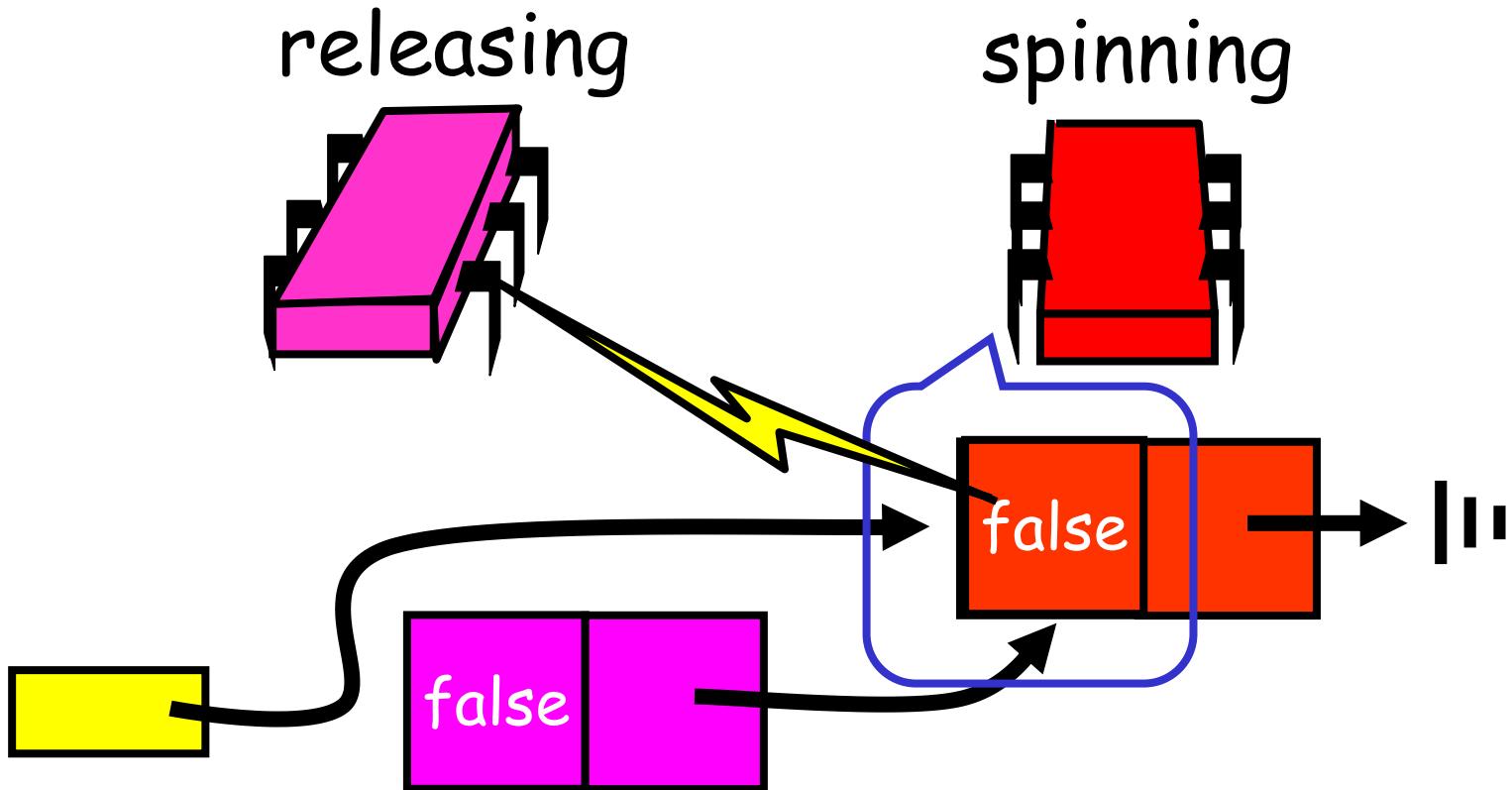
# Purple Release



# Purple Release

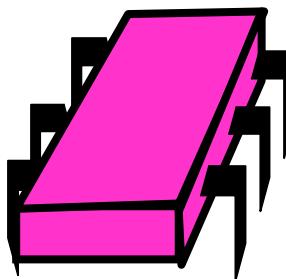


# Purple Release

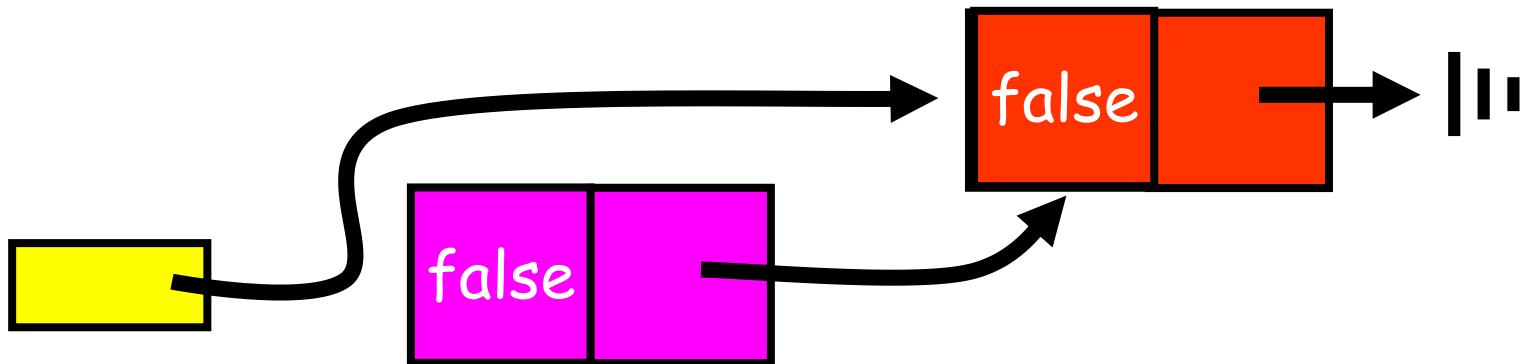
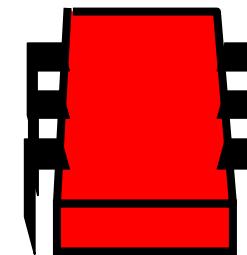


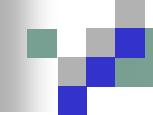
# Purple Release

releasing



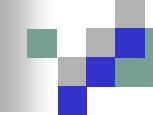
Acquired lock





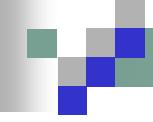
# Lock with timeout

- Java interface includes a tryLock() method to specify a maximum duration the thread is willing to wait to acquire the lock
- Should the thread not acquire the lock in the designated time, the thread will timeout



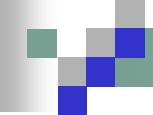
# Abortable Locks

- What if you want to give up waiting for a lock?
- For example
  - Timeout
  - Database transaction aborted by user



# Lock with timeout

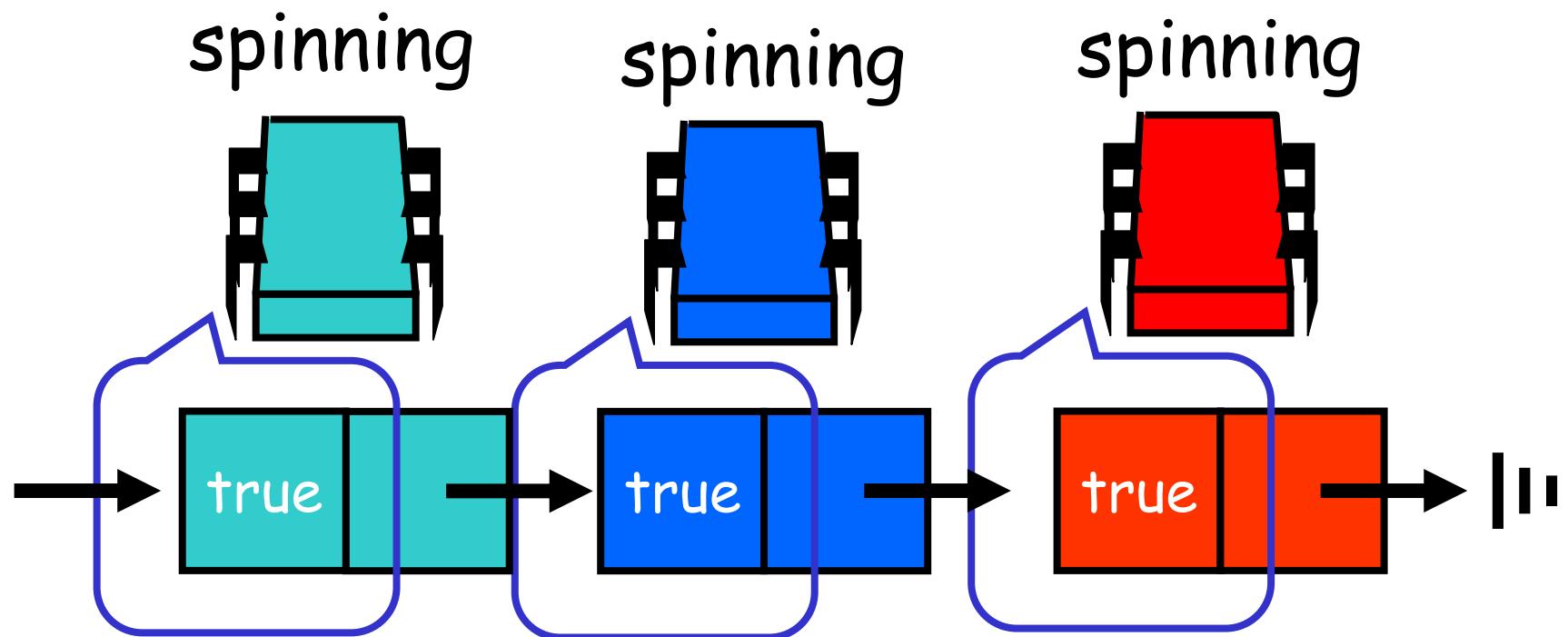
- What happens to other threads when you timeout?



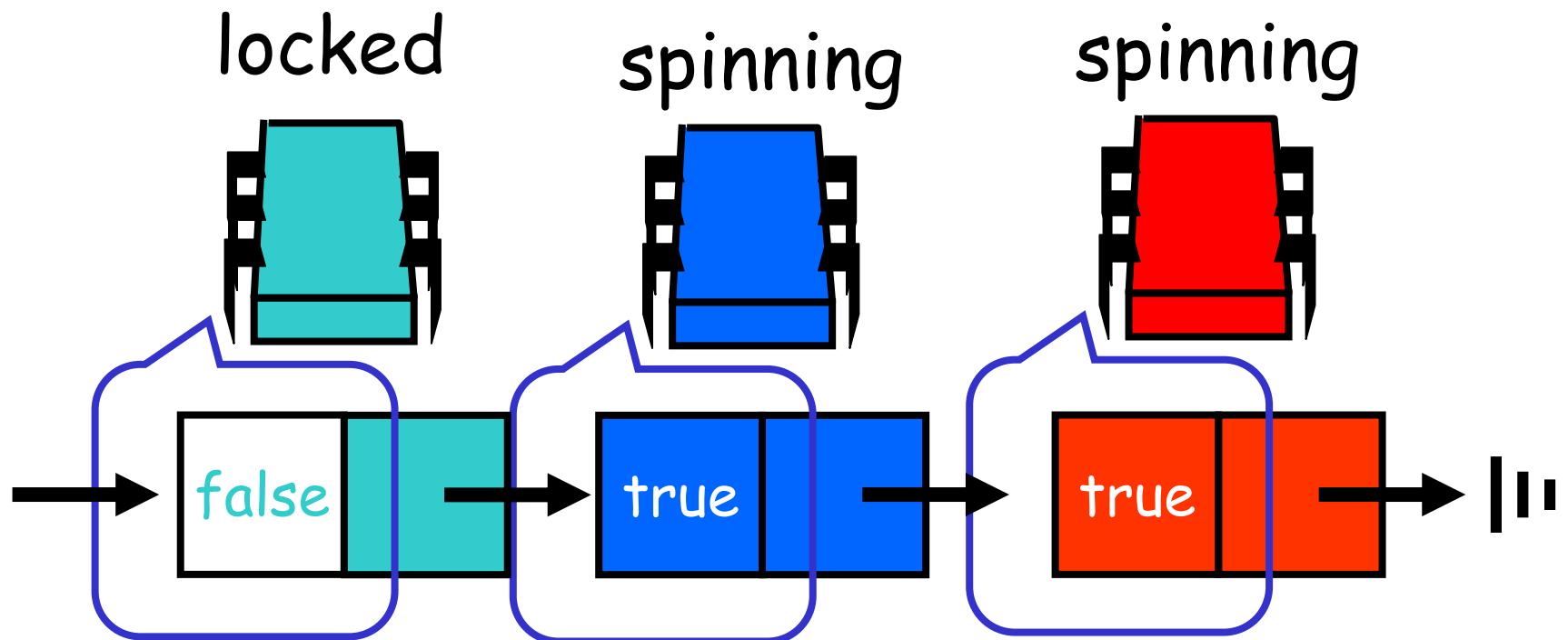
# Back-off Lock

- Aborting is trivial
  - Just return from lock() call
- Extra benefit:
  - No cleaning up
  - Wait-free
  - Immediate return

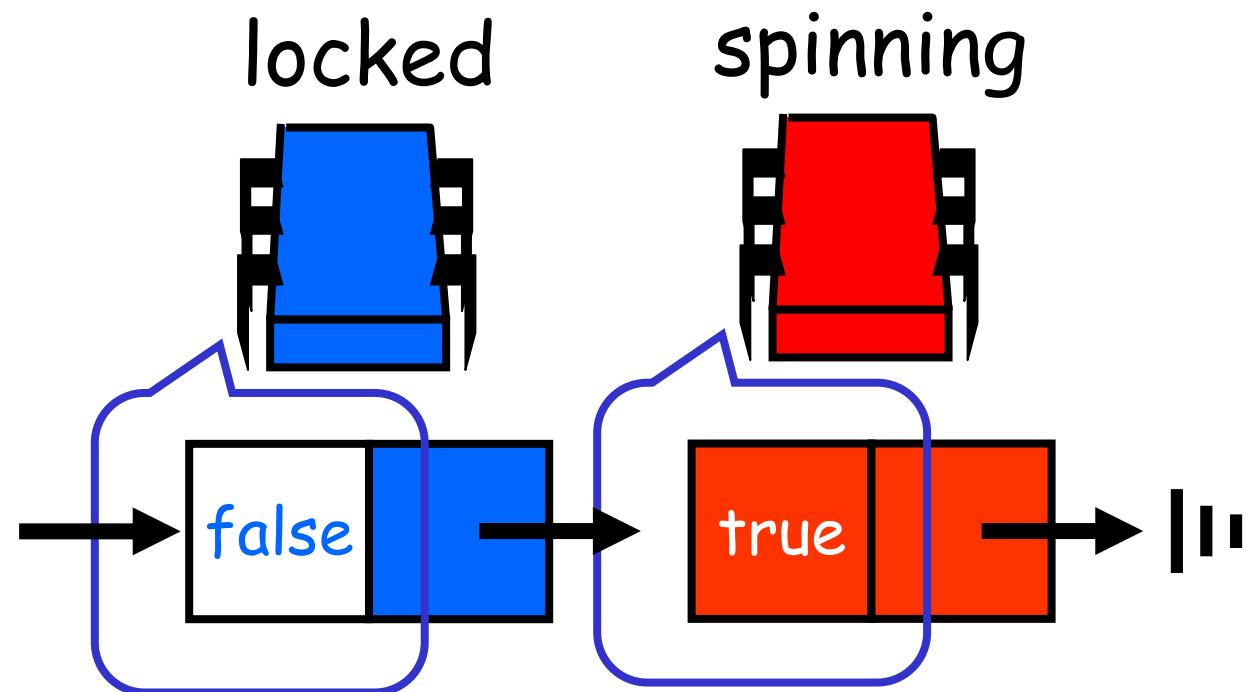
# MCS Queue Locks



# MCS Queue Locks

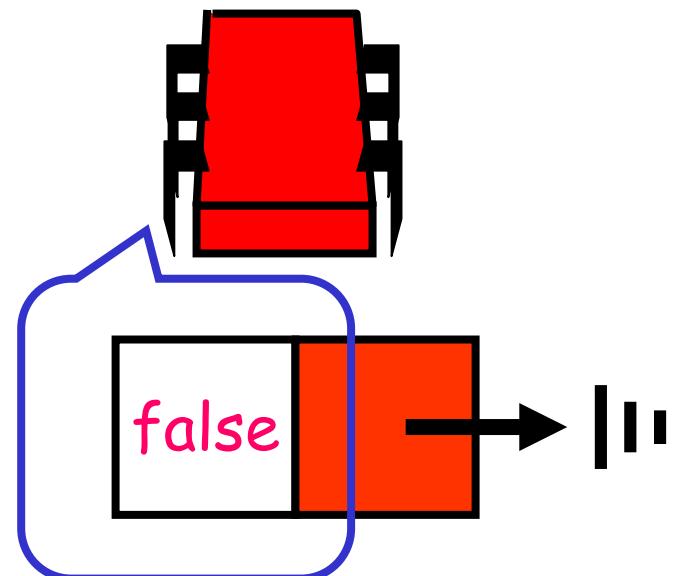


# MCS Queue Locks

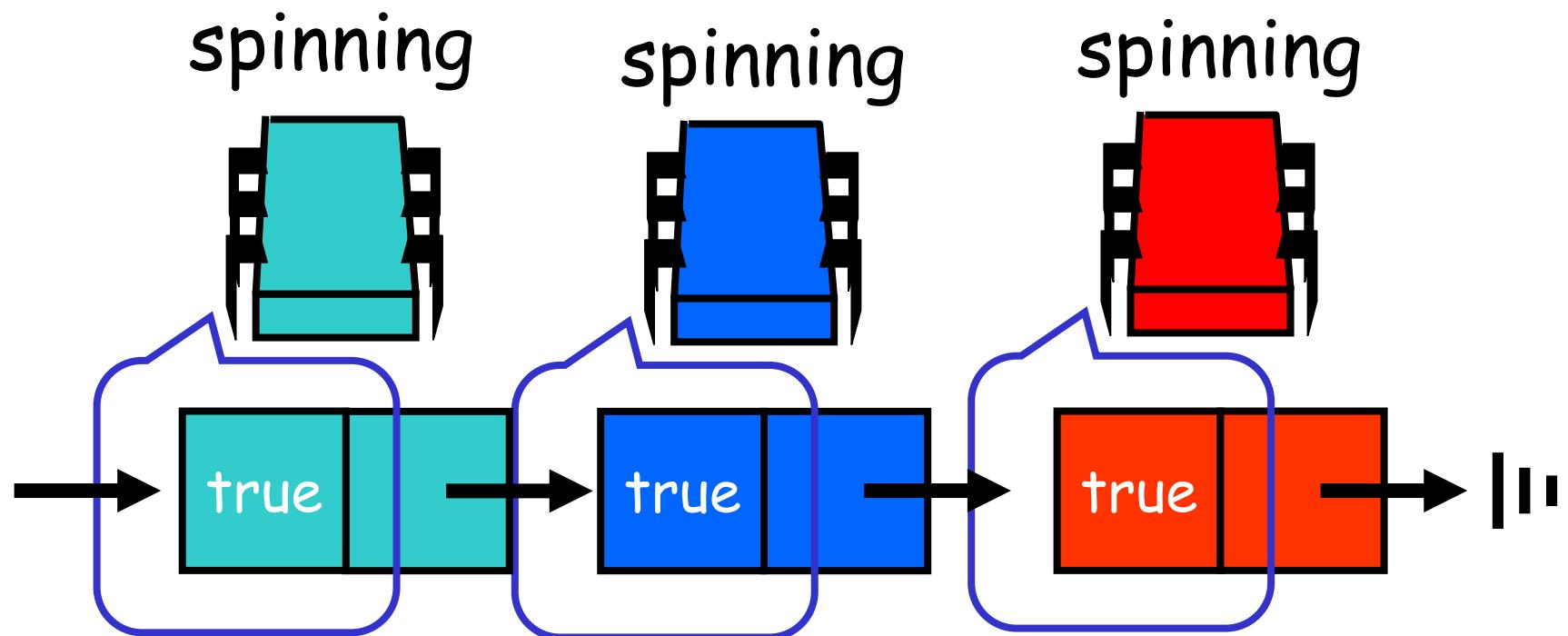


# MCS Queue Locks

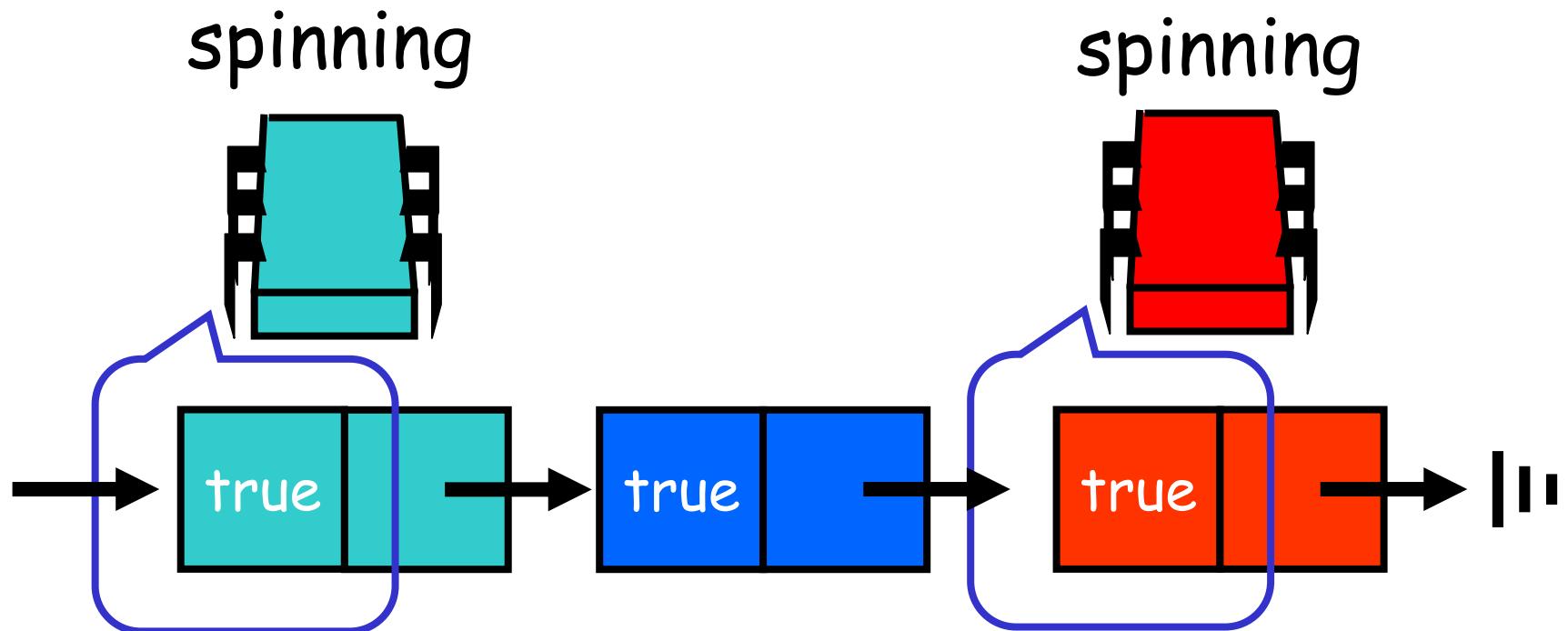
locked



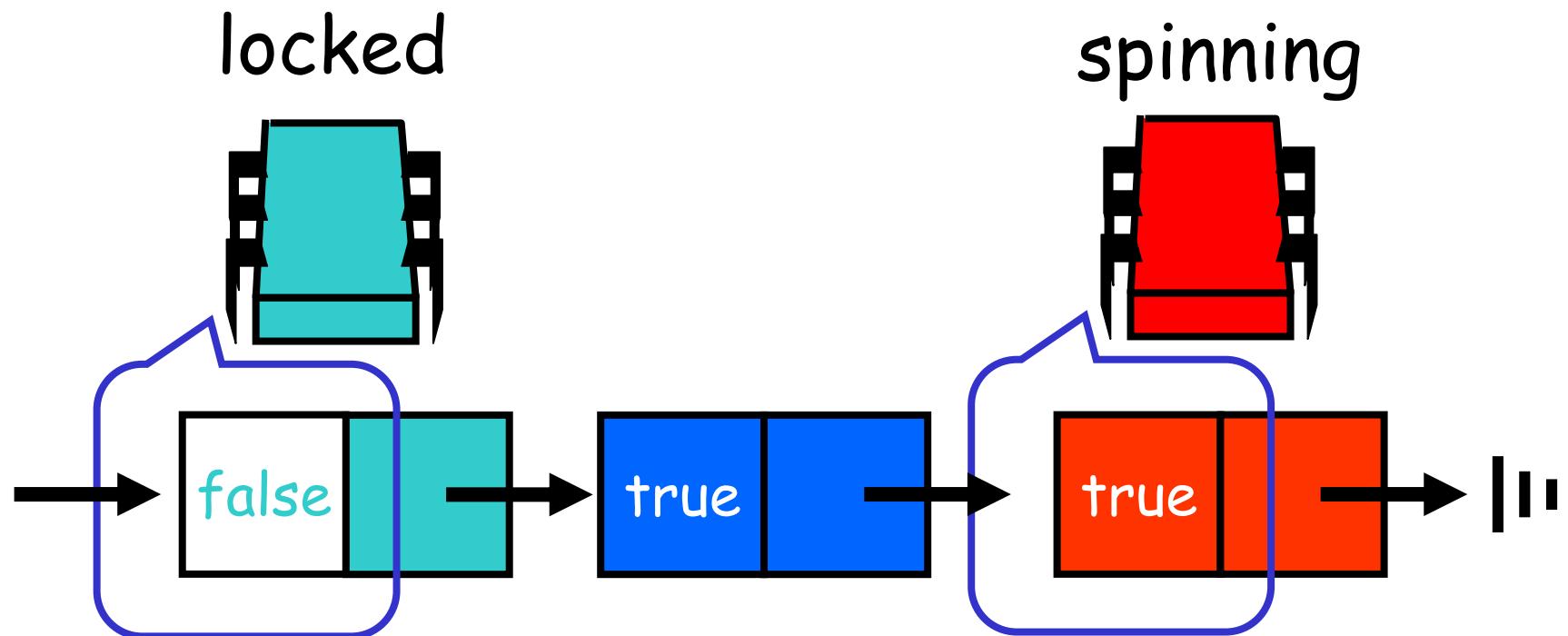
# MCS Queue Locks



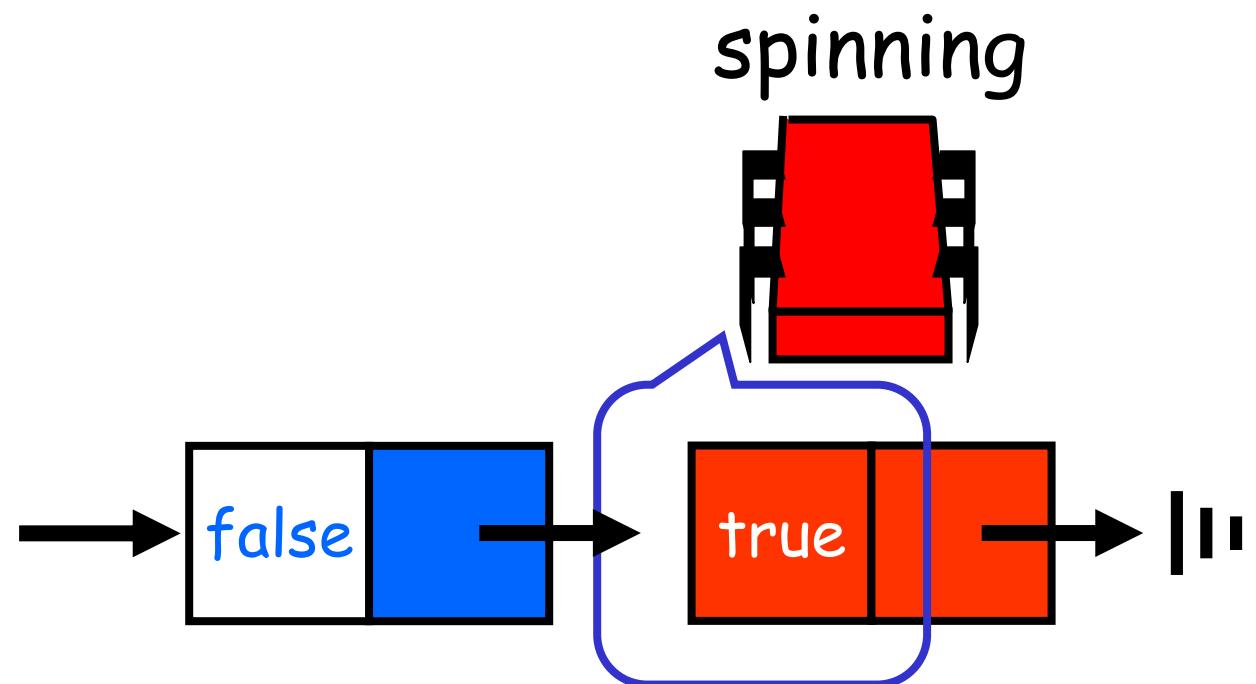
# MCS Queue Locks



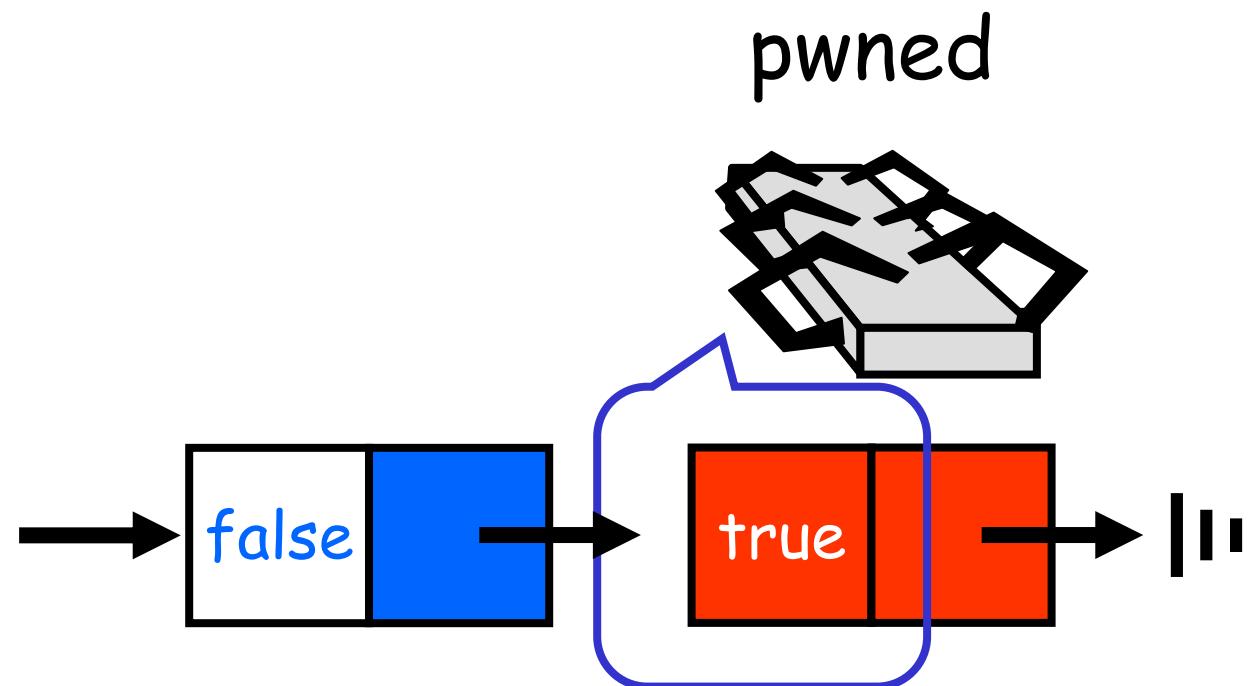
# MCS Queue Locks



# MCS Queue Locks

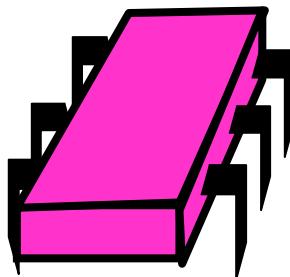


# MCS Queue Locks

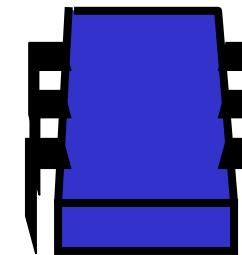
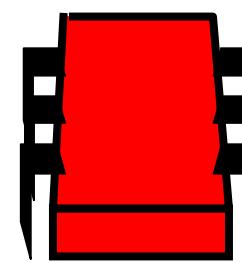


# CLH Queue Lock

acquired



acquiring acquiring



tail

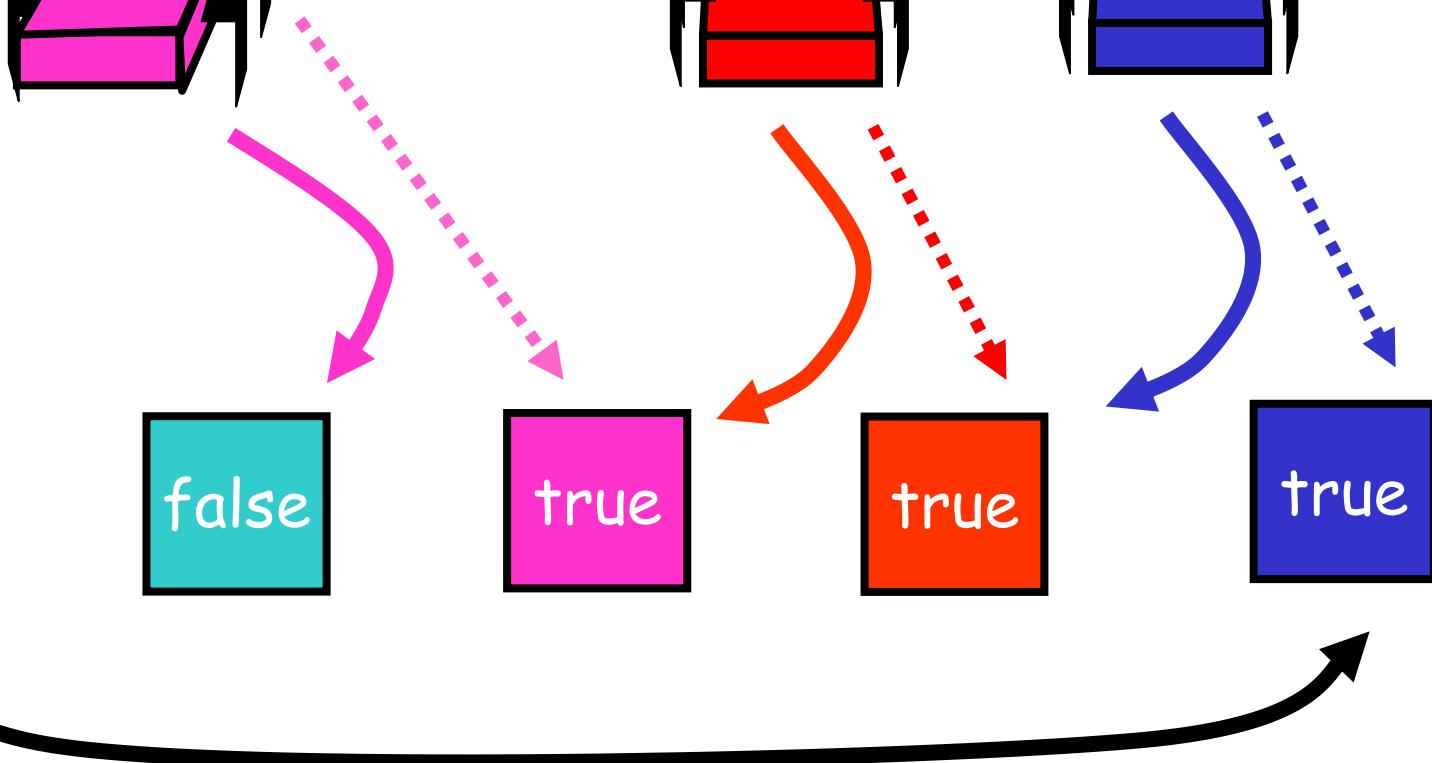


false

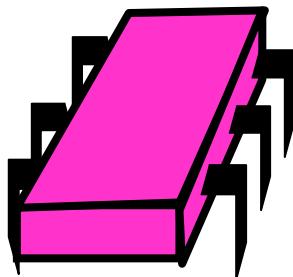
true

true

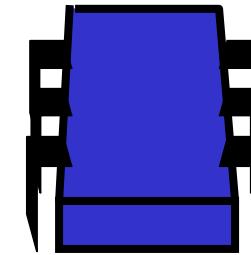
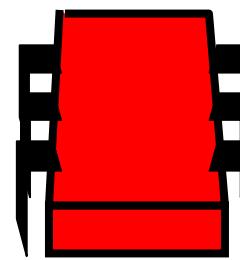
true



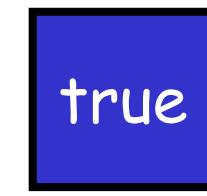
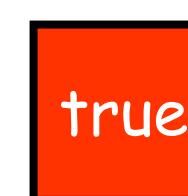
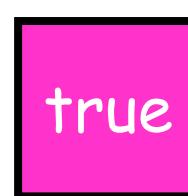
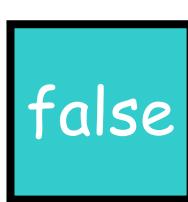
acquired



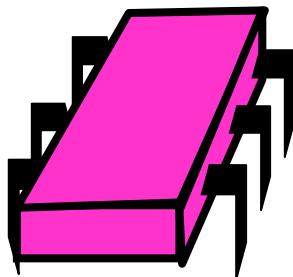
acquiring acquiring



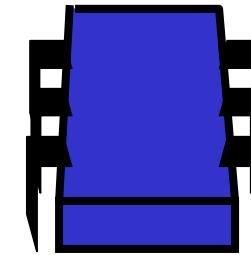
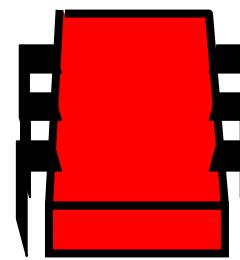
tail



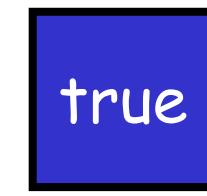
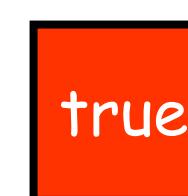
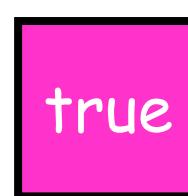
acquired



acquiring acquiring



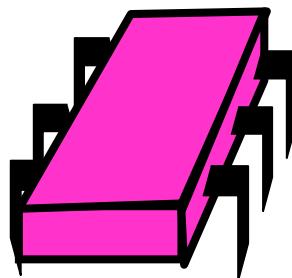
tail





Finished

release



tail



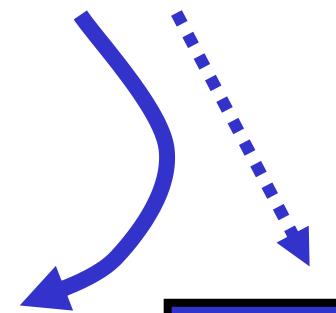
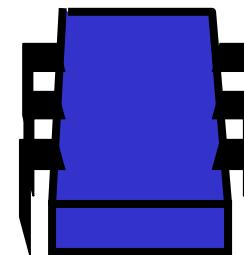
false

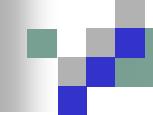
false

true

true

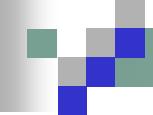
acquiring





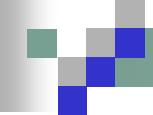
# Queue Locks

- Can't just quit
  - Thread in line behind will starve
- Need a graceful way out



# Abortable CLH Lock

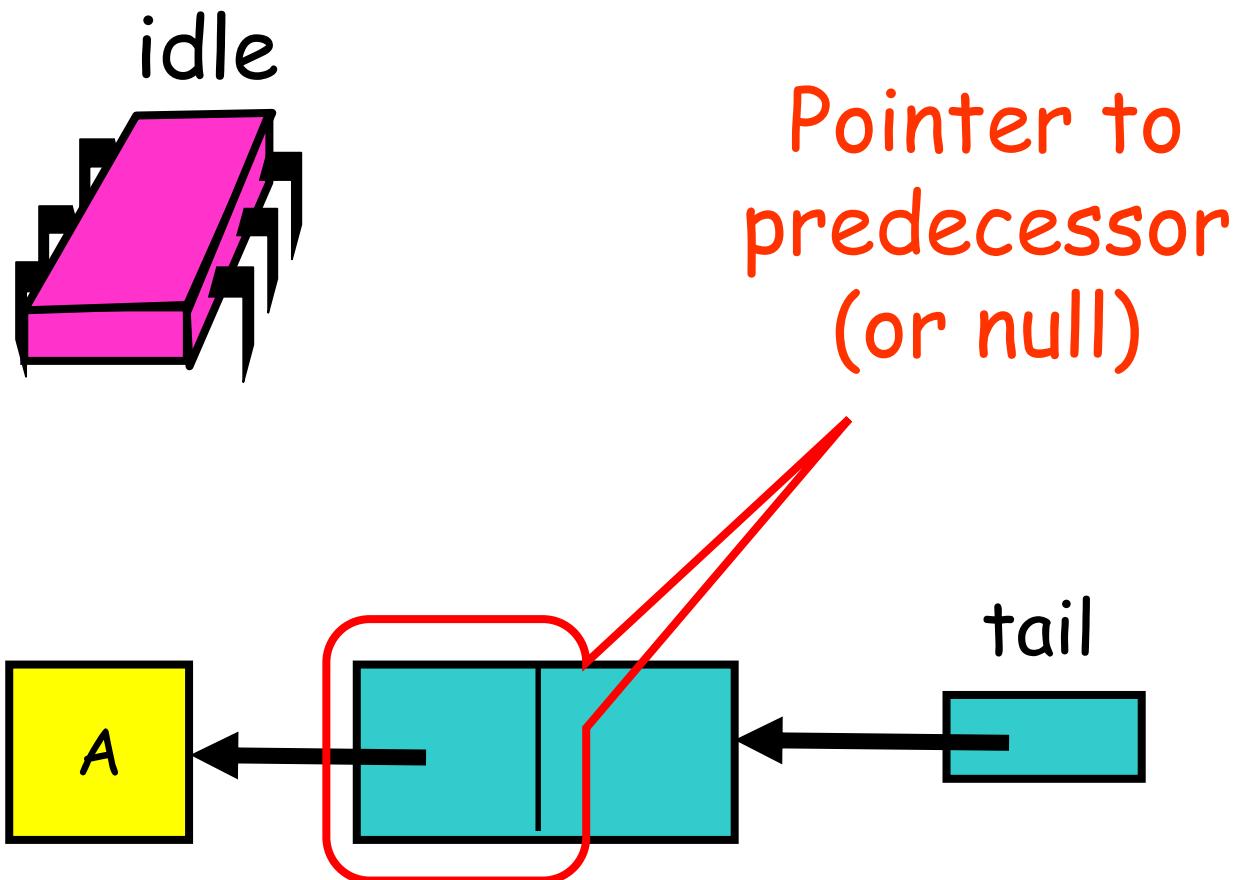
- When a thread gives up
  - Removing node from queue in a wait-free way is hard
- Idea for lazy approach:
  - let successor deal with it.



# A queue lock with timeouts

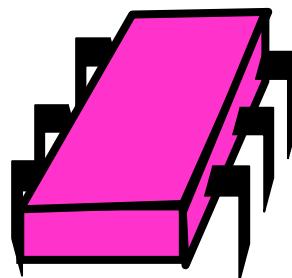
- When a thread times out, it marks its node as abandoned
- The successor in the queue notices that the node has been abandoned
- Successor starts spinning on the abandoned node's predecessors

# Initially

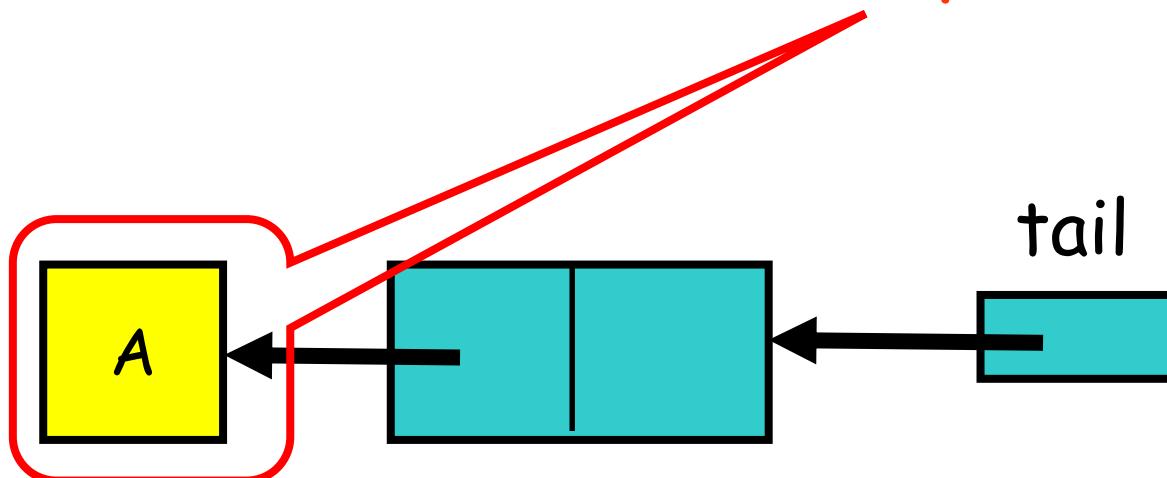


# Acquiring

idle

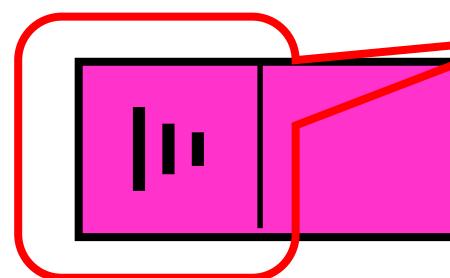
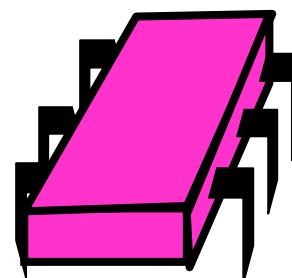


Distinguished  
available **node**  
means lock is  
free

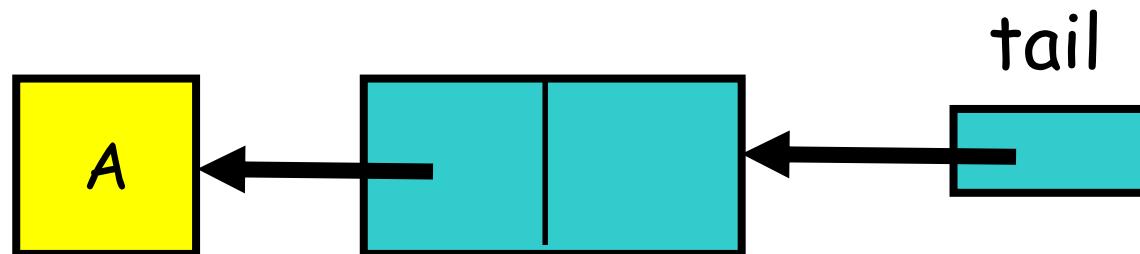


# Acquiring

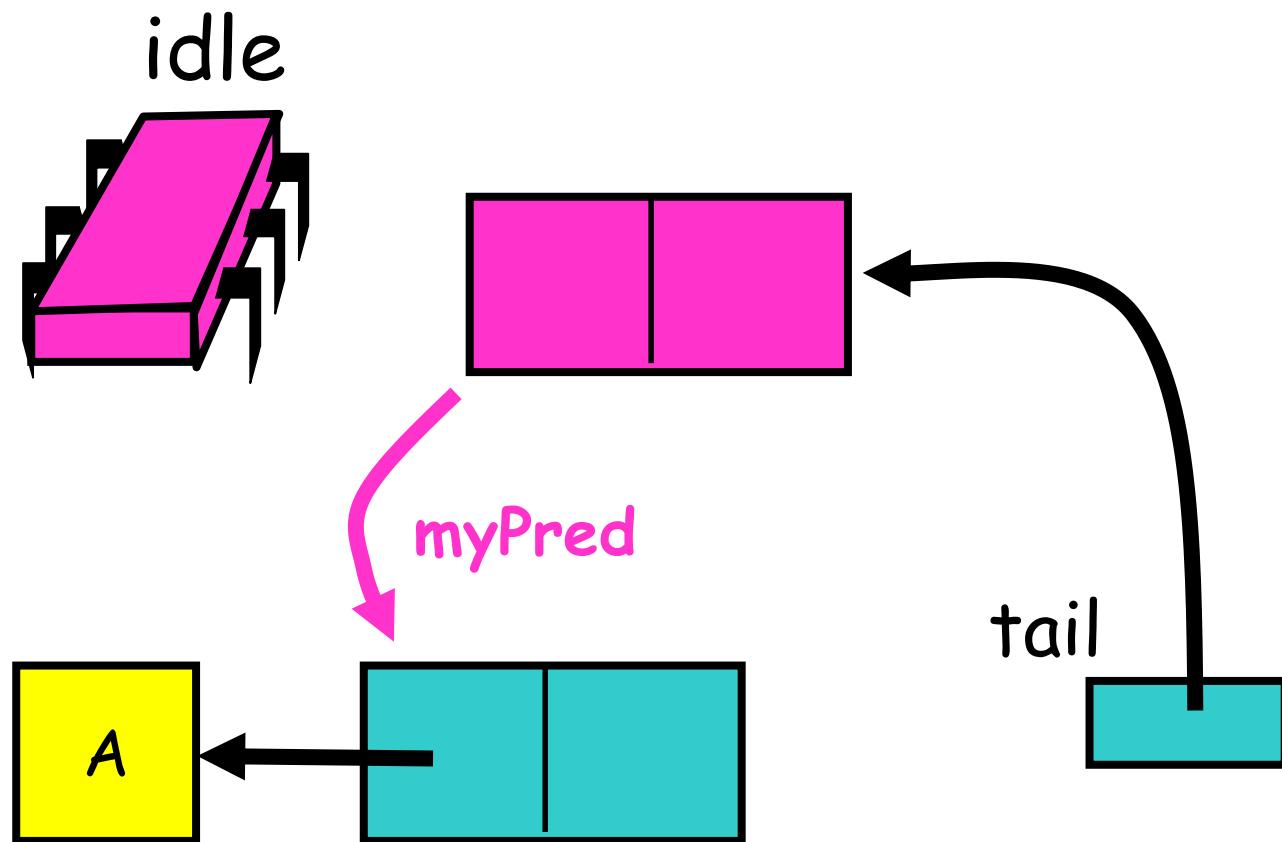
idle



Null predecessor  
means lock not  
released or  
aborted

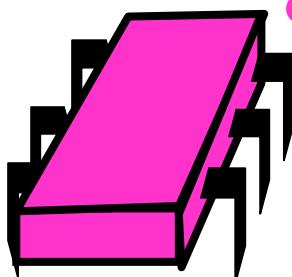


# Acquiring

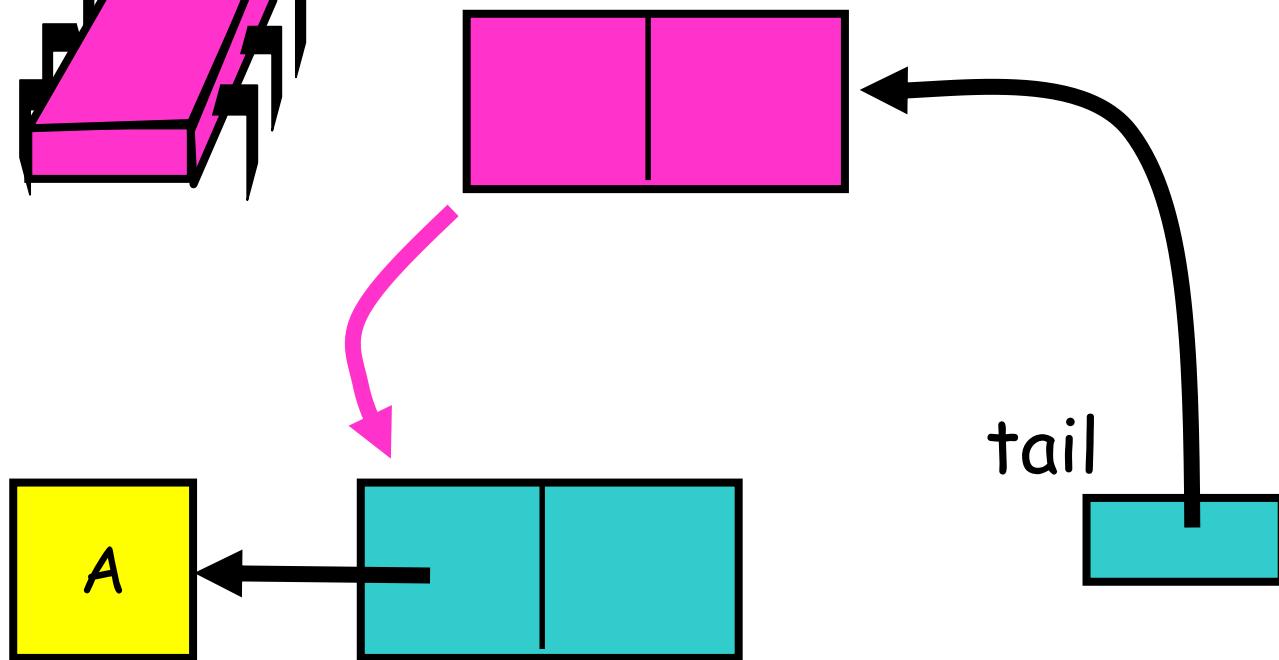


# Acquired

idle



Pointer to  
AVAILABLE means  
lock is free.



# Time-out Lock

```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

# Time-out Lock

```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

Distinguished node to  
signify free lock

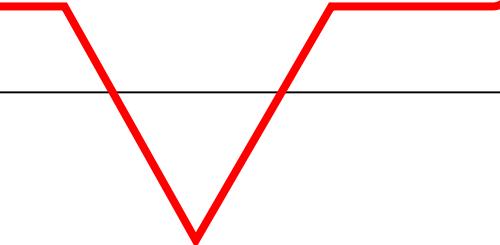
# Time-out Lock

```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```

Tail of the queue

# Time-out Lock

```
public class TOLock implements Lock {  
    static Qnode AVAILABLE  
        = new Qnode();  
    AtomicReference<Qnode> tail;  
    ThreadLocal<Qnode> myNode;
```



Remember my node ...

# Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred =  
tail.getAndSet(qnode);  
if (myPred== null  
    || myPred.prev == AVAILABLE) {  
    return true;  
}  
...  
}
```

# Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred =  
        tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
}
```

Create & initialize node

# Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred =  
        tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
    Swap with tail
```

# Time-out Lock

```
public boolean lock(long timeout) {  
    Qnode qnode = new Qnode();  
    myNode.set(qnode);  
    qnode.prev = null;  
    Qnode myPred =  
        tail.getAndSet(qnode);  
    if (myPred == null  
        || myPred.prev == AVAILABLE) {  
        return true;  
    }  
    ...  
}
```

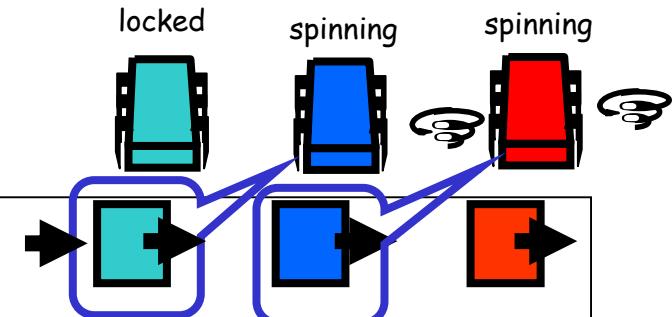
If predecessor absent or released, we are done

# Time-out Lock

...

```
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}
```

...



# Time-out Lock

```
...  
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}  
...  
  
Keep trying for a while ...
```

# Time-out Lock

...

```
long start = now();  
while (now() - start < timeout) {  
    Qnode predPred = myPred.prev;  
    if (predPred == AVAILABLE) {  
        return true;  
    } else if (predPred != null) {  
        myPred = predPred;  
    }  
}
```

...

*Spin on predecessor's  
prev field*

# Time-out Lock

```
...
long start = now();
while (now() - start < timeout) {
    Qnode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }
}
...

```

Predecessor released lock

# Time-out Lock

```
...
long start = now();
while (now() - start < timeout) {
    Qnode predPred = myPred.prev;
    if (predPred == AVAILABLE) {
        return true;
    } else if (predPred != null) {
        myPred = predPred;
    }
}
...

```

Predecessor aborted,  
advance one

# Time-out Lock

```
...
if (!tail.compareAndSet(qnode,
    myPred))
    qnode.prev = myPred;
return false;
}
}
```

What do I do when I time out?

# Time-out Lock

```
...
if (!tail.compareAndSet(qnode,
    myPred))
    qnode.prev = myPred;
return false;
}
```

Do I have a successor? If CAS fails: I do have a successor, tell it about myPred

# Time-out Lock

```
...
if (!tail.compareAndSet(qnode,
    myPred))
    qnode.prev = myPred;
return false;
{
}
```

If CAS succeeds: no successor,  
simply return false

# Time-Out Unlock

```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode,  
                           null))  
        qnode.prev = AVAILABLE;  
}
```

# Time-out Unlock

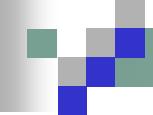
```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode,  
                           null))  
        qnode.prev = AVAILABLE;  
}
```

If CAS failed: exists  
successor, notify successor  
it can enter

# Timing-out Lock

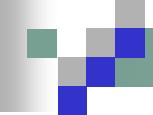
```
public void unlock() {  
    Qnode qnode = myNode.get();  
    if (!tail.compareAndSet(qnode,  
                           null))  
        qnode.prev = AVAILABLE;  
}
```

*CAS successful: set tail to  
null, no clean up since no  
successor waiting*



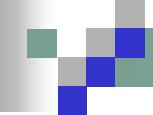
# Composite Locks

- Spin lock algorithms impose trade-offs
- Most spin locks have advantages and disadvantages



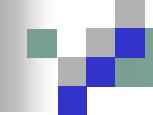
# Composite Locks

- Backoff Lock:
  - + provide trivial timeout protocols
  - are not scalable
  - may have critical section underutilization if timeout parameters are not well-tuned.



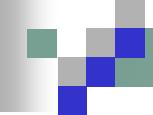
# Composite Locks

- Queue locks:
  - + provide first-come-first-served fairness
  - + fast lock release
  - nontrivial protocols for abandoned nodes



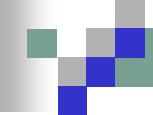
# Composite Locks

- Composite locks combine the best of both approached



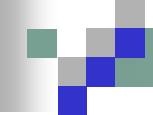
# Composite Locks

- Idea:
  - In a queue only the threads at the front of the queue require lock handoffs
- Solution:
  - Keep a small number of waiting threads in a queue and have the rest use exponential backoff



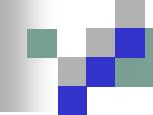
# Composite Locks

- Keep a short, fixed-sized array of lock nodes
- Each thread that tries to acquire the lock selects a node in the array at random
- If the node is in use, the thread backs off and tries again



# Composite Locks

- Once the thread acquires a node, it enqueues that node in a TOLock-style queue
- The thread spins on the preceding node and when that node's owner signals it is done, the thread enters the critical section

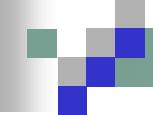


# Composite Locks

- Every node in the array has a state:
  - FREE, WAITING, RELEASED, ABORTED
- Actions depend on the state of the randomly selected node

# Composite Lock

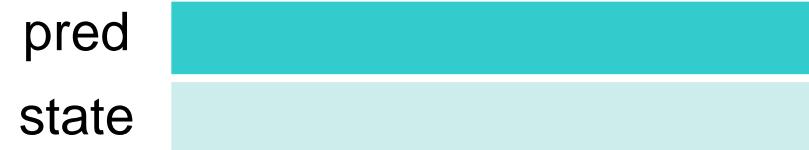
- A FREE node is available for other threads to acquire
- A WAITING node is linked into the queue and either in the critical section or waiting to enter
- A node becomes RELEASED when the owner leaves the critical section
- If a node is enqueued but the owner has quit, the node is ABORTED



# Composite Lock

- A thread acquires the lock in three steps:
  - Acquires a node in the waiting array
  - Enqueues the node in the queue
  - Waits until the node reaches the head of the queue

Qnode:

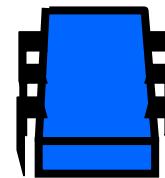


myPred

Tail:

null	null	null	null
FREE	FREE	WAITING	FREE

myPred = null



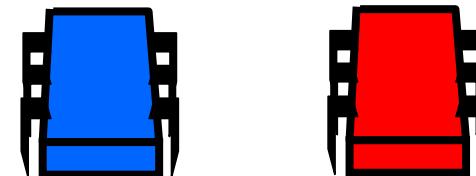
In CS

Randomly  
select array  
location

Tail:

null	null	null	null
FREE	FREE	WAITING	WAITING

myPred = null      myPred

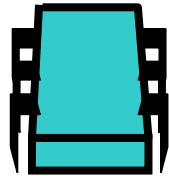


In CS

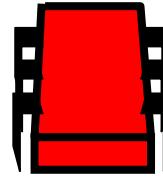
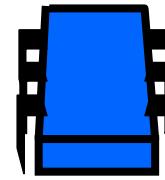
Tail:

null	null	null	null
WAITING	FREE	WAITING	WAITING

myPred



myPred = null

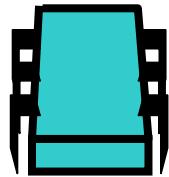


In CS

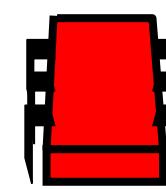
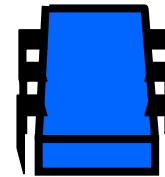
Tail:

null	null	null	
WAITING	FREE	WAITING	ABORTED

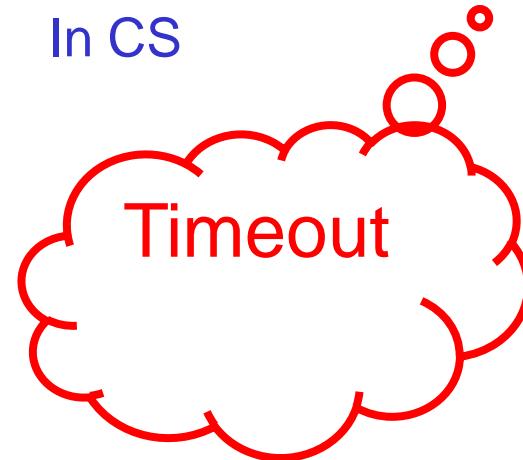
myPred



myPred = null



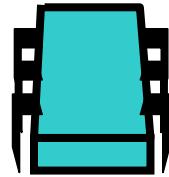
In CS



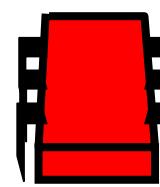
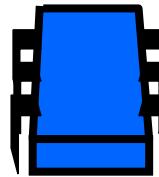
Tail:

null	null	null	
WAITING	FREE	WAITING	FREE

myPred



myPred = null



In CS

Better  
clean this  
up

# Composite Lock

```
public class CompositeLock implements  
Lock {  
    AtomicReference<Qnode> tail;  
    Qnode [] waiting;  
    ...
```

Tail field is either null or  
points to last value entered  
into queue (waiting array)

# Composite Lock

```
public class CompositeLock implements  
    Lock {  
    AtomicReference<Qnode> tail;  
    Qnode [] waiting;  
    ...  
}
```

Short queue of waiting  
threads - implemented as  
array

# Composite Lock

```
enum State {FREE, WAITING, RELEASED,  
ABORTED}  
class Qnode {  
    AtomicReference<State> state;  
    Qnode pred;  
  
    public Qnode() {  
        state = new  
AtomicReference<State>  
(State.FREE);  
    }  
}
```

# Composite Lock

```
enum State {FREE, WAITING, RELEASED,  
ABORTED}  
class Qnode {  
    AtomicReference<State> state;  
    Qnode pred;  Starting states for all  
                threads is FREE  
    public Qnode() {  
        state = new  
AtomicReference<State>  
(State.FREE);  
    }  
}
```

# Composite Lock

```
public boolean tryLock() {  
    Qnode node = acquireNode();  
    Qnode pred = spliceNode(node);  
    waitForPredecessor(pred, node);  
}
```

# Composite Lock

```
public boolean tryLock() {  
    Qnode node = acquireNode();  
    Qnode pred = spliceNode(node);  
    waitForPredecessor(pred, node);  
}
```

Randomly select node from  
waiting array and return  
available node

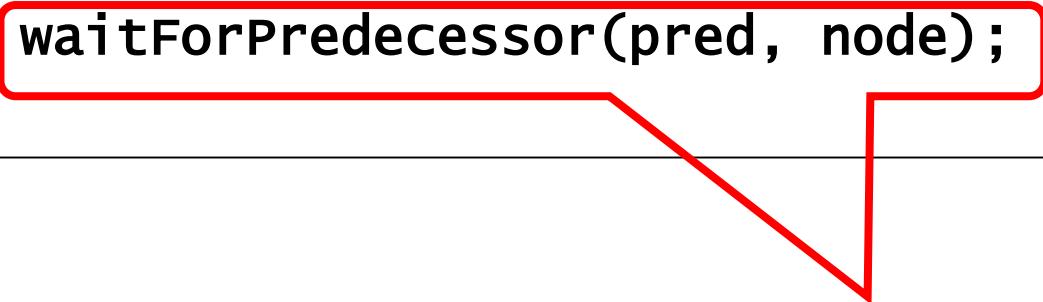
# Composite Lock

```
public boolean tryLock() {  
    Qnode node = acquireNode();  
    Qnode pred = spliceNode(node);  
    waitForPredecessor(pred, node);  
}
```

Add selected node to queue

# Composite Lock

```
public boolean tryLock() {  
    Qnode node = acquireNode();  
    Qnode pred = spliceNode(node);  
    waitForPredecessor(pred, node);  
}
```



Add selected node to queue

# acquireNode()

- Thread selects a node at random
- Tries to acquire the node by setting its state from FREE to WAITING
- If it fails it examines the state
  - If ABORTED or RELEASED it recycles the node

# Composite Lock

```
private Qnode acquireNode() {  
    Qnode node = waiting[random.nextInt()];  
    while (true) {  
        if (state.compareAndSet(FREE, WAITING))  
            return node;
```

# Composite Lock

Choose random location  
in waiting array

```
private Qnode acquireNode() {  
    Qnode node = waiting[random.nextInt()];  
    while (true) {  
        if (state.compareAndSet(FREE, WAITING))  
            return node;
```

# Composite Lock

If state was FREE then  
no problem

```
private Qnode acquireNode() {  
    Qnode node = waiting[random.nextInt()];  
    while (true) {  
        if (state.compareAndSet(FREE, WAITING))  
            return node;  
    }  
}
```

# Composite Lock

If state ABORTED or  
RELEASED

```
currTail = tail;  
if (state == ABORTED || state ==  
RELEASED) {  
    if (node == currTail) {  
        Qnode myPred = null;  
        if (state == ABORTED)  
            myPred = node.pred;  
        if (tail.compareAndSet(  
            currTail, myPred)  
        node.state.set(WAITING);  
        return node;  
    }  
}
```

# Composite Lock

Can only be recycled if last node in queue

```
currTail = tail;  
if (state == ABORTED || state ==  
RELEASED) {  
    if (node == currTail) {  
        Qnode myPred = null;  
        if (state == ABORTED)  
            myPred = node.pred;  
        if (tail.compareAndSet(  
            currTail, myPred)  
        node.state.set(WAITING);  
        return node;  
    }  
}
```

# Composite Lock

If last node is ABORTED tail is set to predecessor otherwise

```
currTail = tail;
if (state == ABORTED || state ==
RELEASED) {
    if (node == currTail) {
        Qnode myPred = null;
        if (state == ABORTED)
            myPred = node.pred;
        if (tail.compareAndSet(
            currTail, myPred)
            node.state.set(WAITING);
        return node;
    }
}
```

Successfully recycle  
node and set to  
**WAITING**

# Composite Lock

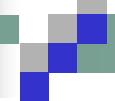
```
currTail = tail;  
if (state == ABORTED || state ==  
RELEASED) {  
    if (node == currTail) {  
        Qnode myPred = null;  
        if (state == ABORTED)  
            myPred = node.pred;  
        if (tail.compareAndSet(  
            currTail, myPred)  
            node.state.set(WAITING);  
        return node;  
    }  
}
```

# spliceNode()

- Once node is acquired, the node is spliced into the queue
- Thread tries to set tail to node
- If timeout, set node to FREE and throw TimeoutException
- If succeeds return prior value of tail (node's predecessor)

# Composite Lock

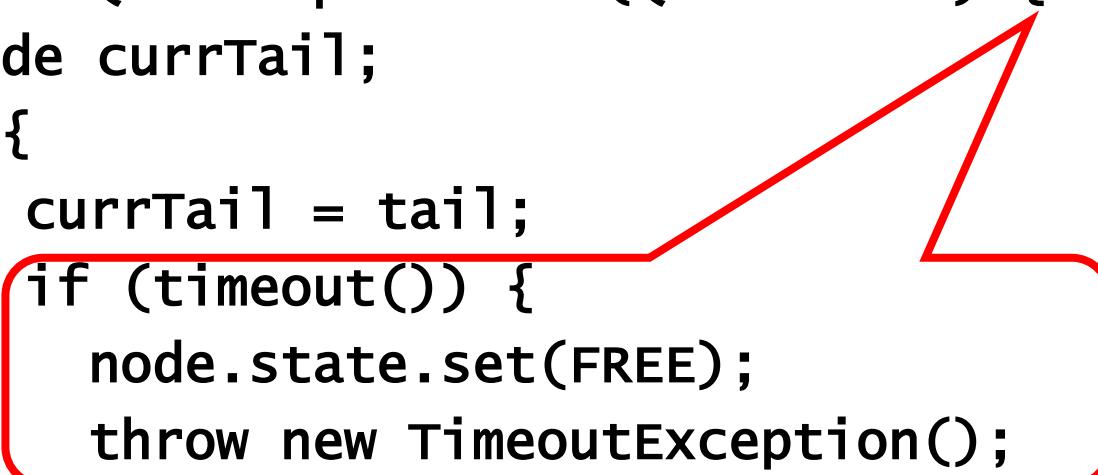
```
private Qnode spliceNode(Qnode node) {  
    Qnode currTail;  
    do {  
        currTail = tail;  
        if (timeout()) {  
            node.state.set(FREE);  
            throw new TimeoutException();  
        }  
    } while (!tail.compareAndSet(currTail,  
        node));  
    return currTail;  
}
```



If waited too long...

# Composite Lock

```
private Qnode spliceNode(Qnode node) {  
    Qnode currTail;  
    do {  
        currTail = tail;  
        if (timeout()) {  
            node.state.set(FREE);  
            throw new TimeoutException();  
        }  
    } while (!tail.compareAndSet(currTail,  
        node));  
    return currTail;  
}
```



# Composite Lock

```
private Qnode spliceNode(Qnode node) {  
    Qnode currTail;  
    do {  
        currTail = tail;  
        if (timeout()) {  
            node.state.set(FREE);  
            throw new TimeoutException();  
        }  
    } while (!tail.compareAndSet(currTail,  
        node));  
    return currTail;  
}
```

Try to set tail to new  
node

# Composite Lock

- Finally thread has to wait its turn
- If predecessor is null thread can enter CS
- If predecessor is not RELEASED, check if it is ABORTED
  - If so, mark as FREE and get ABORTED node's predecessor
- If predecessor is RELEASED enter CS

# Composite Lock

```
private void waitForPredecessor {  
    if (pred == null)  
        return;  
    State predState = pred.state.get();  
    while (predState != RELEASED) {  
        if (predState == ABORTED) {  
            Qnode temp = pred;  
            pred = pred.pred;  
            temp.state.set(FREE);  
        }  
    }  
}
```

# Composite Lock

```
private void waitForPredecessor {  
    if (pred == null)  
        return;  
    State predState = pred.state.get();  
    while (predState != RELEASED) {  
        if (predState == ABORTED) {  
            Qnode temp = pred;  
            pred = pred.pred;  
            temp.state.set(FREE);  
        }  
    }
```

If I am first in line

# Composite Lock

```
private void waitForPredecessor {  
    if (pred == null)  
        return;  
  
    State predState = pred.state.get();  
    while (predState != RELEASED) {  
        if (predState == ABORTED) {  
            Qnode temp = pred;  
            pred = pred.pred;  
            temp.state.set(FREE);  
        }  
    }  
}
```

Spin on predecessor's state

# Composite Lock

```
private void waitForPredecessor {  
    if (pred == null)  
        return;  
    State predState = pred.state.get();  
    while (predState != RELEASED) {  
        if (predState == ABORTED) {  
            Qnode temp = pred;  
            pred = pred.pred;  
            temp.state.set(FREE);  
        }  
    }  
}
```

If predecessor ABORTED set to  
FREE and spin on predecessors  
predecessor

# Composite Lock

```
if (timeout()) {  
    node.pred = pred;  
    node.state.set(ABORTED);  
} throw new TimeoutException();  
predState = pred.state.get();  
}  
pred.state.set(FREE);  
return;  
}
```

If I have waited long enough...

# Composite Lock

```
if (timeout()) {  
    node.pred = pred;  
    node.state.set(ABORTED);  
} throw new TimeoutException();  
predState = pred.state.get();  
}  
  
pred.state.set(FREE);  
return;  
}
```

When predecessor RELEASED

# Composite Lock

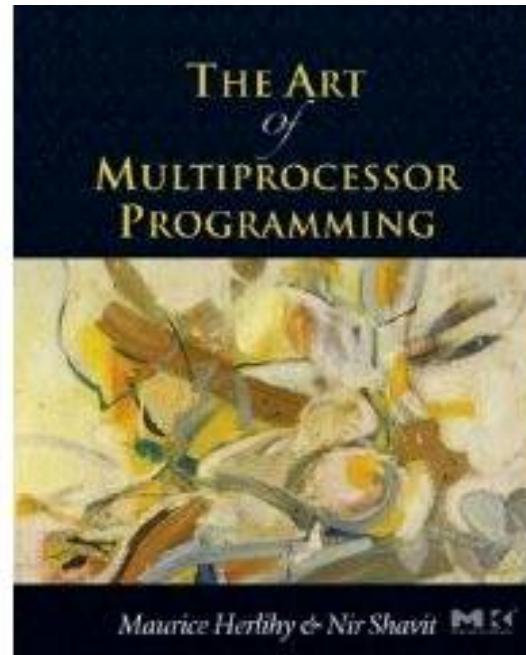
```
public void unlock() {  
    myNode.state.set(RELEASED);  
}
```

COS 226

# Chapter 8

## Monitors and Blocking Synchronization

# Acknowledgement

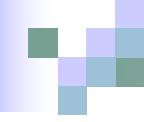


- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



# Monitors

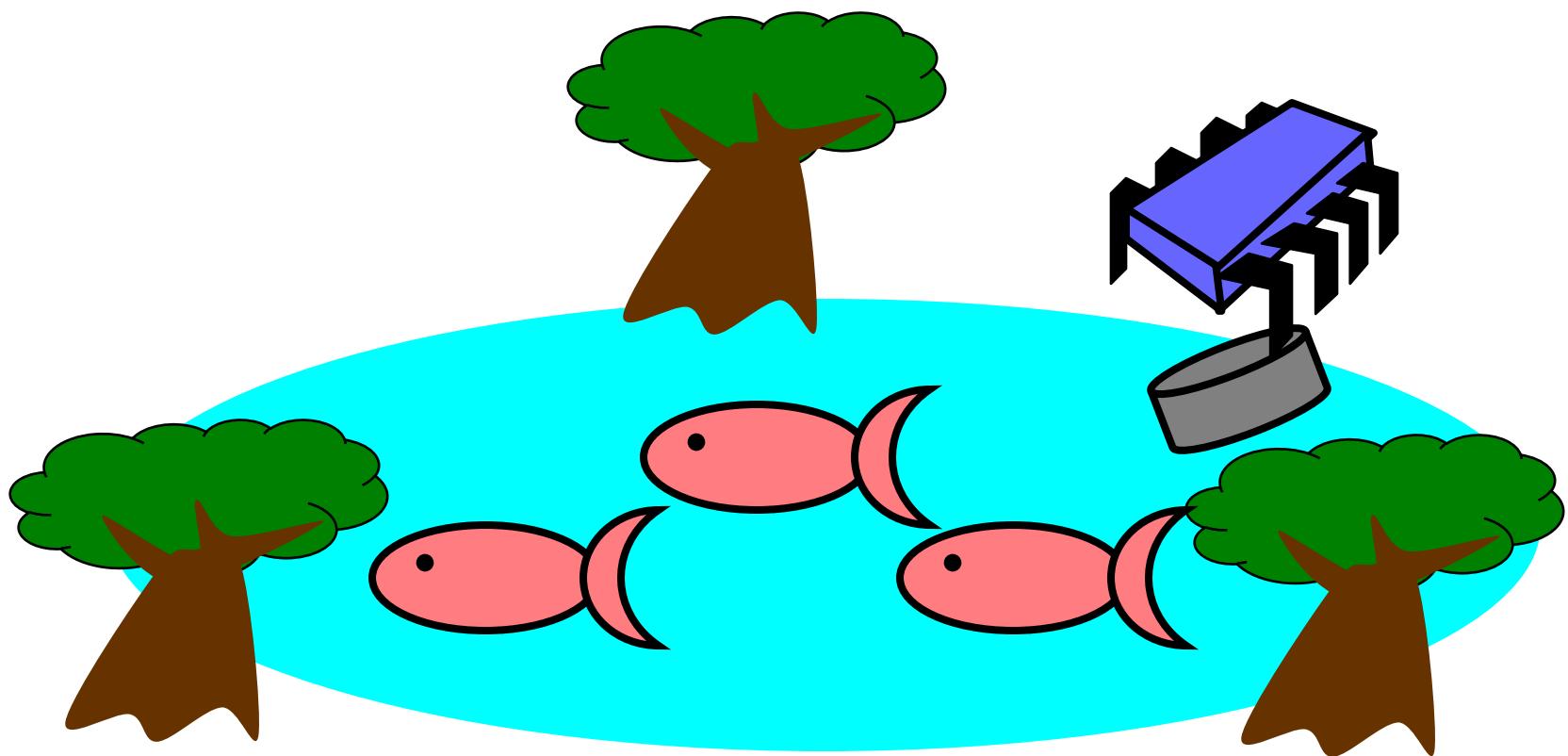
- Structured way of combining synchronization and data
- Combines data, methods and synchronization in a single modular package



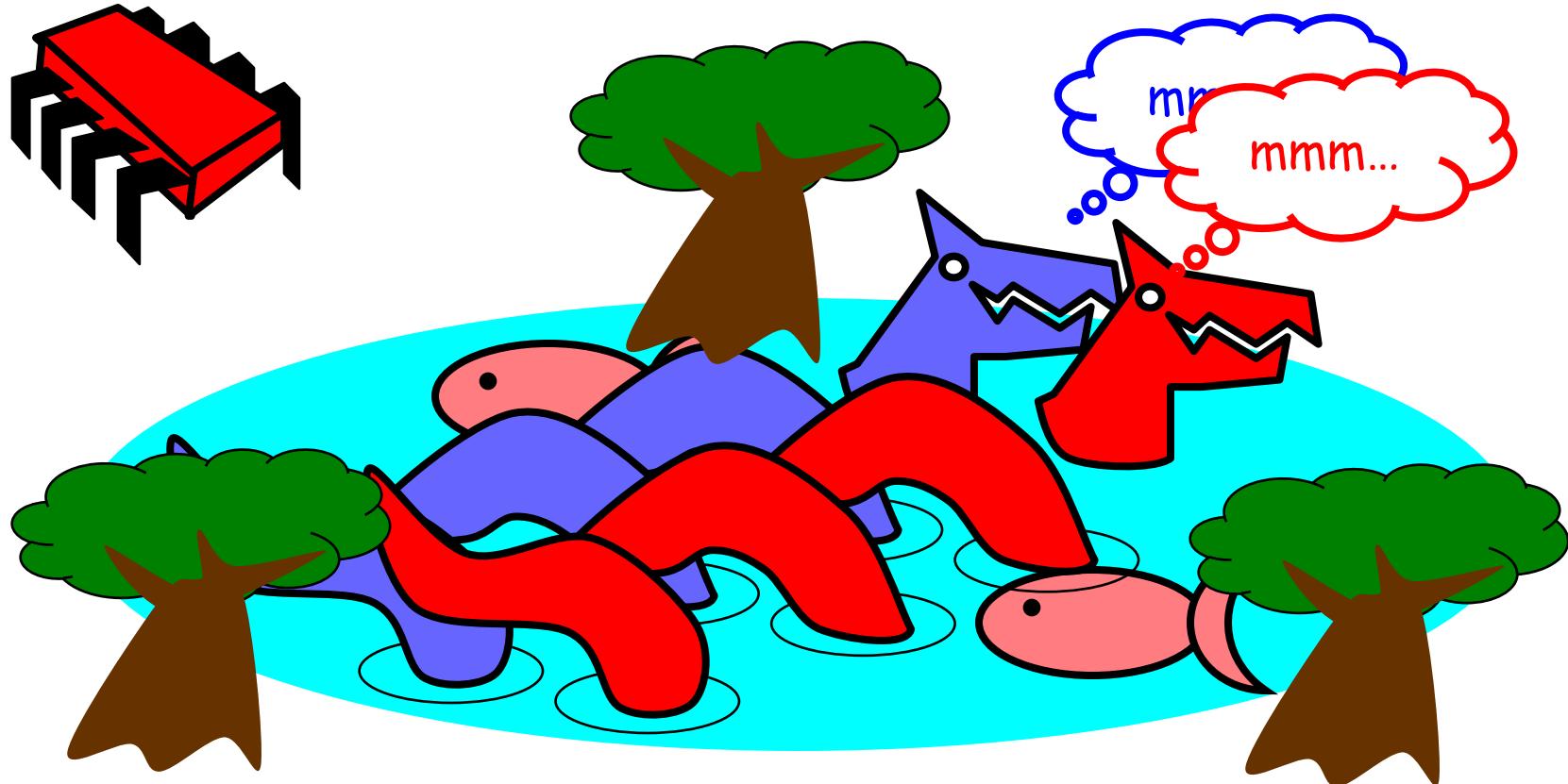
# Producer-Consumer Problem

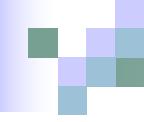
- Synchronization problem

# Bob Puts Food in the Pond



# Alice releases her pets to Feed





# Producer/Consumer

- Alice and Bob can't meet
  - Each has restraining order on other
  - So he puts food in the pond
  - And later, she releases the pets
- Avoid
  - Releasing pets when there's no food
  - Putting out food if uneaten food remains

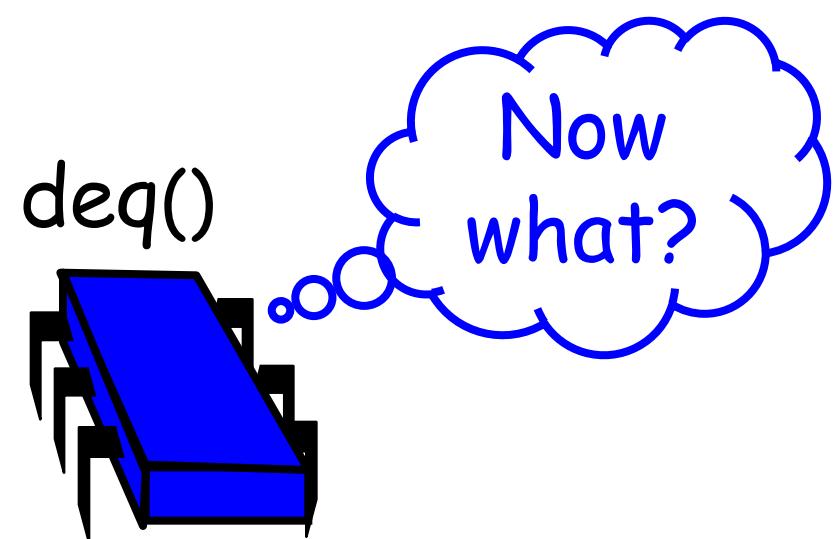
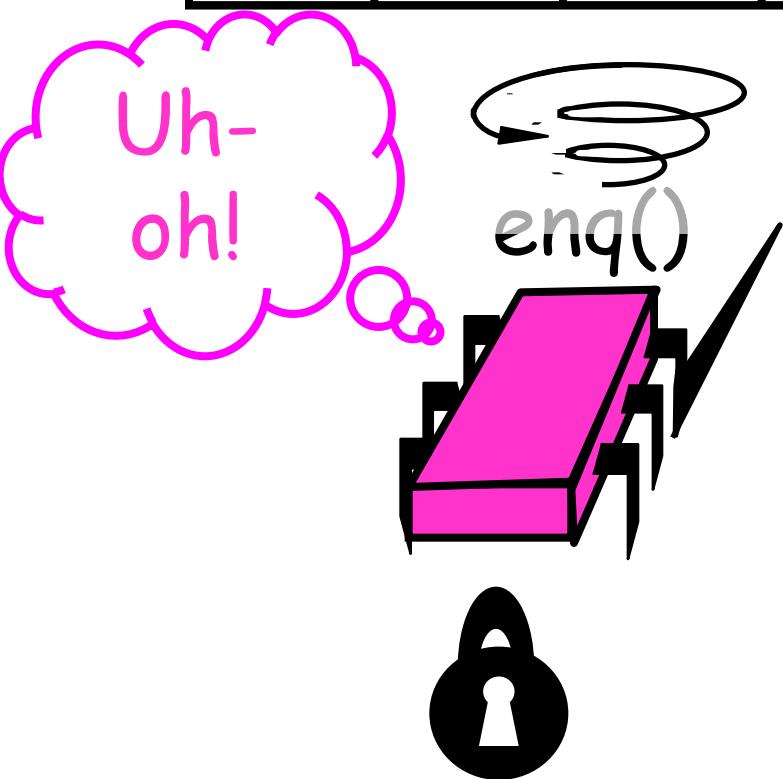
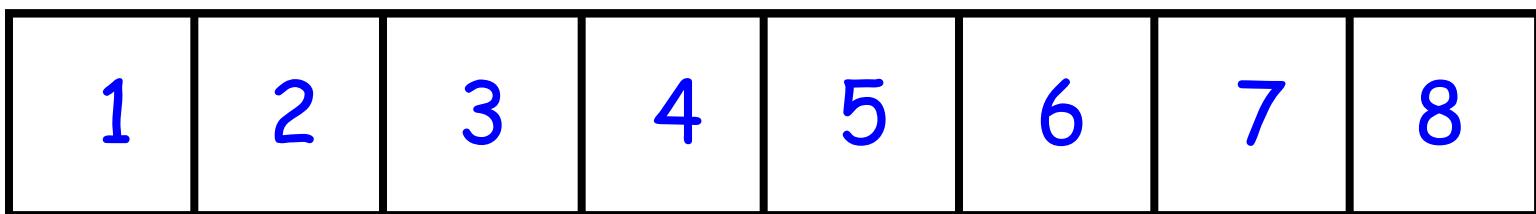
# In real life...

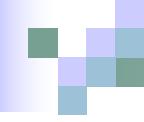
- Imagine application with two threads – producer and consumer
- Two threads communicate through a shared FIFO queue
- Principles:
  - The producer generates data, puts it into the queue and start again
  - At the same time the consumer, consumes the data one piece at a time

# Producer

```
mutex.lock()  
try {  
    queue.enq();  
} finally {  
    mutex.unlock();  
}
```

# Producer-Consumer





# Problems

- What happens when the queue is full?
- Problem:
  - Producer should not try to add data if the buffer is full
  - Consumer should not try to remove data from an empty buffer
- Where should this be managed?



# Sensible approach

- Allow each queue to manage its own synchronization
- The queue itself has its own internal lock, acquired by methods and released when it returns
  - For example, lock() can be called inside the enq() method
- If a thread then tries to enq() an item onto a full queue, the enq() method itself can detect the problem



# Monitor Locks

- Only one thread at a time can *hold* a lock
- A thread *acquires* a lock when it starts to hold the lock
- A thread *releases* the lock when it stops holding the lock
- A monitor has methods each which acquires the lock when it is called and releases the lock when it returns



# Monitor Locks

- When a thread cannot immediately acquire a lock it either:
  - Spins – repeatedly testing whether it is available
  - Blocks – suspends thread and creates new thread
- Spinning = short time
- Blocking = long time



# Spinning vs. Blocking

- For example:
  - A thread waiting for another thread to release the lock should spin if lock is held briefly
  - A consumer thread waiting to dequeue an item from a empty queue should block
- Often spinning and blocking are combined
- Spinning does not work on uniprocessors



# Producer-Consumer

- Need a way for a thread that has acquired a lock to release the lock for a while and then to reacquire it and try again



# Conditions

- In java concurrency package
- Condition object provides ability to release a lock temporarily
- A Condition is related to a lock and is created by lock's newCondition() method

# Conditions

```
Condition condition = mutex.newCondition();  
...  
mutex.lock();  
try {  
    while (!property) {  
        condition.await();  
    } catch (InterruptedException e) {...}  
...  
}
```

# Conditions

```
Condition condition = mutex.newCondition();
```

```
...
```

```
mutex.lock();
```

```
try {
```

```
    while (!property) {
```

```
        condition.await();
```

```
    } catch (InterruptedException e) {...}
```

```
...
```

```
}
```

Create Condition object that is related to lock

# Conditions

```
Condition condition = mutex.newCondition();
```

```
...
mutex.lock();
try {
    while (!property) {
        condition.await();
    } catch (InterruptedException e) {...}
}
```

Acquires lock

# Conditions

```
Condition condition = mutex.newCondition();  
...  
mutex.lock();  
try {  
    while (!property) {  
        condition.await();  
    } catch (InterruptedException e) {...}  
...  
}  
  
Tests whether property holds
```

# Conditions

```
Condition condition = mutex.newCondition();  
...  
mutex.lock();  
try {  
    while (!property) {  
        condition.await();  
    } catch (InterruptedException e) {...}  
...  
}  
  
Releases the lock and suspends itself
```

# Conditions

```
Condition condition = mutex.newCondition();  
...  
mutex.lock();  
try {  
    while (!property) {  
        condition.await();  
    } catch (InterruptedException e) {...}  
...  
} Suspension can be interrupted by other thread
```

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Condition to check  
whether queue is  
full

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Array of elements  
with some max  
amount

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Attempt to add item  
to queue

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock(); // Acquires lock  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Check whether queue is full



# Producer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            ...  
        } catch (InterruptedException e) {  
            Thread.currentThread().interrupt();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Sleep while queue is full



# Conditions

- How does the Producer know when the queue is not full anymore?
- Options:
  - Can awaken on time constraints
  - Another thread (Consumer) will have to wake it.

# Conditions

```
public interface Condition {  
    void await()  
    boolean await (long time)  
    boolean awaitUntil (Date)  
    long awaitNanos (long nanoSec)
```

- All of these methods uses time constraints

# Conditions

- Another thread can also use signal() to notify threads that it has changed a certain property.

```
void signal()  
void signalAll()
```



# Conditions

- However, there is no guarantee that when the threads awakens after a specified time, the property will hold
- Thus the thread must retest the property when it awakes



# Conditions

- When threads are woken up, they could still have to compete for the lock



# Producer-Consumer queue

- If the Producer thread has been suspended because of a full queue, when should it be woken up again?
- And what about checking whether the queue is empty or not?

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            items[tail] = x;  
            ...  
            isEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            items[tail] = x;  
            ...  
            isEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Second condition to test whether queue is empty

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            items[tail] = x;  
            ...  
            isEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

When thread wakes up it can access the CS – add an item

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public void enq(T x) {  
        lock.lock();  
        try {  
            while (items.length == max)  
                isFull.await();  
            items[tail] = x;  
            ...  
            isEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Since an item has been added it is not empty – alert Consumer

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public T deq() {  
        lock.lock();  
        try {  
            while (items.length == 0)  
                isEmpty.await();  
            T x = items[head]  
            ...  
            isFull.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public T deq() {  
        lock.lock(); // First acquire lock  
        try {  
            while (items.length == 0)  
                isEmpty.await();  
            T x = items[head];  
            ...  
            isFull.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

First acquire lock

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public T deq() {  
        lock.lock();  
        try {  
            while (items.length == 0)  
                isEmpty.await();  
            T x = items[head]  
            ...  
            isFull.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Check whether  
queue is empty

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public T deq() {  
        lock.lock();  
        try {  
            while (items.length == 0)  
                isEmpty.await();  
            T x = items[head]  
            ...  
            isFull.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

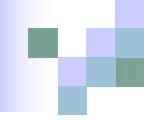
Release and sleep  
if queue is empty

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public T deq() {  
        lock.lock();  
        try {  
            while (items.length == 0)  
                isEmpty.await();  
            T x = items[head];  
            ...  
            isFull.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Remove item from  
queue

```
class LockedQueue {  
    Lock lock = new ReentrantLock();  
    Condition isFull = lock.newCondition();  
    Condition isEmpty = lock.newCondition();  
    T[] items;  
    ...  
    public T deq() {  
        lock.lock();  
        try {  
            while (items.length == 0)  
                isEmpty.await();  
            T x = items[head];  
            ...  
            isFull.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Let Producer know  
that queue is not full  
anymore



# Monitor

- The combination of methods, mutual exclusion locks and condition objects is called a **monitor**



# Lost-Wakeup Problem

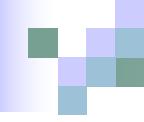
- Just like locks are vulnerable to deadlock, conditions are vulnerable to lost wakeups
- Lost wakeup:
  - One or more threads wait forever without realizing that the property for which they have been waiting has become true

```
public void enq(T x) {  
    lock.lock();  
    try {  
        while (count == items.length)  
            isFull.await();  
        items[tail] = x;  
        ++count;  
        if (count == 1)  
            isEmpty.signal();  
    }  
    } finally {  
        lock.unlock();  
    }  
}
```



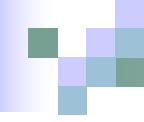
# Lost-Wakeup Problem

- Always make sure that all necessary threads are signalled
- When using multiple consumer and producer threads (for example)
  - Make sure that you signal all processes waiting on a property and
  - Specify a timeout when waiting



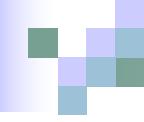
# Readers-Writers Locks

- Many shared objects have the property of having readers that only access the object without modifying it and writers that modify the object



# Readers-Writers Locks

- Should readers be synchronized?
- Should writers be synchronized?
- How is synchronization achieved?

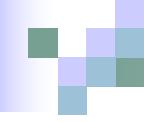


# Readers-Writers Locks

- A readers-writers lock should allow multiple readers or a single writer to enter the critical section

# Readers-Writers Locks Interface

```
public interface ReadWriteLock {  
    Lock readLock();  
    Lock writeLock();  
}
```



# Readers-Writers Locks

- The interface should satisfy the following properties:
  - No thread can acquire the write lock while any thread holds either the write lock or the read lock
  - No thread can acquire the read lock while any other thread holds the write lock



# Questions

- How can we make sure that all other readers are finished before acquiring a write lock?
- How can we make sure that a writer is not currently holding a lock?
- When should the threads block?
- When should the threads wake up?



# Solutions

- We need a way of specifying how many readers and writers are currently contending for the lock

# SimpleReadWriteLock

```
public class SimpleReadWriteLock  
    implements ReadWriteLock {  
    boolean writer;  
    int readers;  
    Lock lock;  
    Condition condition;  
    Lock readLock, writeLock;
```

# SimpleReadWriteLock

```
public SimpleReadWriteLock() {  
    writer = false; No initial writers  
    readers = 0;  
    lock = new ReentrantLock();  
    readLock = new ReadLock();  
    writeLock = new WriteLock();  
    condition = lock.newCondition();
```

# SimpleReadWriteLock

```
public SimpleReadWriteLock() {  
    writer = false;  
    readers = 0; No initial readers  
    lock = new ReentrantLock();  
    readLock = new ReadLock();  
    writeLock = new WriteLock();  
    condition = lock.newCondition();
```

# SimpleReadWriteLock

```
public SimpleReadWriteLock() {  
    writer = false;          Separate locks for  
    readers = 0;             readers and writers  
    lock = new ReentrantLock();  
    readLock = new ReadLock();  
    writeLock = new WriteLock();  
    condition = lock.newCondition();
```

# Reader Lock

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

# Reader Lock

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

# Reader Lock

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

Suspend while there is a writer

# Reader Lock

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

Increase number  
of readers

# Reader Lock

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



Acquire Lock

# Reader Lock

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

Decrease number  
of readers

# Reader Lock

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

If there are no more readers, wake the writer

# Writer Lock

```
class writeLock implements Lock {  
    public void lock() {  
        lock.lock();  
        try {  
            while (readers > 0)  
                condition.await();  
            writer = true;  
        } finally {  
            lock.unlock();  
        }  
    }
```

# Writer Lock

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



# Readers-Writers Lock

- The SimpleReadWriteLock is correct, but is it fair?
- If readers are much more frequent than writers, the writers can be locked out for a long time



# Fair Readers-Writers Lock

- A fair implementation would imply that no readers are allowed to acquire the lock after the writer has acquired it

# FIFOReadWriteLock

```
public class FIFOReadWriteLock  
    implements ReadwriteLock {  
    boolean writer;  
    int readAcquires, readReleases;  
    Lock lock;  
    Condition condition;  
    Lock readLock, writeLock;
```

# FIFOReadWriteLock

```
public class FIFOReadWriteLock  
    implements ReadwriteLock {  
    boolean writer;  
    int readAcquires, readReleases;  
    Lock lock;  
    Condition condition;  
    Lock readLock, writeLock;
```

Number of readers that have acquired the lock

# FIFOReadWriteLock

```
public class FIFOReadWriteLock  
    implements ReadwriteLock {  
    boolean writer;  
    int readAcquires, readReleases;  
    Lock lock;  
    Condition condition;  
    Lock readLock, writeLock;
```

Number of readers that have released the lock

# FIFO Reader Lock

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

# FIFO Reader Lock

```
public void unlock() {  
    lock.lock();  
    try {  
        readReleases++;  
        if (readAcquires == readReleases)  
            condition.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```

# FIFO Reader Lock

```
public void unlock() {  
    lock.lock();  
    try {  
        readReleases++;  
        if (readAcquires == readReleases)  
            condition.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```

If all the readers that have acquired the lock released the lock

# FIFO Writer Lock

```
class writeLock implements Lock {  
    public void lock() {  
        lock.lock();  
        try {  
            writer = true;  
            while (readAcquires !=  
                readReleases)  
                condition.await();  
        } finally {  
            lock.unlock();  
        }  
    }
```

# FIFO Writer Lock

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



# Locks

- Using the locks described in Chapter 2 and 7, a thread that attempts to reacquire a lock it already holds will deadlock with itself
- This situation can arise if a method that acquires a lock makes a nested call to another method that acquires the same lock



# ReentrantLock

- A lock is reentrant if it can be acquired multiple times by the same thread
- `java.util.concurrent.locks` package provides ReentrantLocks



# ReentrantLock

- A ReentrantLock is owned by the last thread who successfully locked it, but not yet unlocked it
- A thread will successfully hold the lock when the lock is not owned by another thread
- The lock will return immediately if the current thread already holds the lock



# Our own ReentrantLock

```
public class SimpleReentrantLock  
    implements Lock {  
    Lock lock;  
    Condition condition;  
    int owner;  
    int holdCount;  
    ...}
```

# Our own ReentrantLock

```
public class SimpleReentrantLock  
    implements Lock {  
    Lock lock;  
    Condition condition;  
    int owner;  
    int holdCount;  
    ...  
}
```

The ID of the last thread to acquire the lock

# Our own ReentrantLock

```
public class SimpleReentrantLock  
    implements Lock {  
    Lock lock;  
    Condition condition;  
    int owner;  
    int holdCount; Incremented  
each time lock is  
acquired  
    ...
```

# Our own ReentrantLock

```
public void lock() {  
    int me = ThreadID.get();  
    lock.lock();  
    if (owner == me) {  
        holdCount++;  
        return;  
    }  
    while (holdCount != 0)  
        condition.await;  
    owner = me;  
    holdCount = 1;
```

# Our own ReentrantLock

```
public void lock() {  
    int me = ThreadID.get();  
    lock.lock();  
    if (owner == me) {  
        holdCount++;  
        return;  
    }  
    while (holdCount != 0)  
        condition.await;  
    owner = me;  
    holdCount = 1;
```

*Do I already hold  
the lock? - then  
I can access the  
lock*

# Our own ReentrantLock

```
public void lock() {  
    int me = ThreadID.get();  
    lock.lock();  
    if (owner == me) {  
        holdCount++;  
        return;  
    }  
    while (holdCount != 0)  
        condition.await;  
    owner = me;  
    holdCount = 1;
```

Otherwise, if the holdCount is not 0 then another thread is holding the lock

# Our own ReentrantLock

```
public void unlock() {  
    lock.lock();  
    try {  
        if (holdCount == 0 || owner !=  
            ThreadID.get())  
            throw new  
                IllegalMonitorStateException();  
        holdCount--;  
        if (holdCount == 0)  
            condition.signal();  
    } finally {  
        lock.unlock();  
    }  
}
```

You cannot  
unlock a lock that  
is not yours



# Mutual exclusion locks

- A mutual exclusion lock guarantees that only one thread can enter the critical section
- If another thread wants to enter the critical section while it is occupied, it suspends itself until the other thread notifies it to try again



# Semaphore

- A generalization of a mutual exclusion lock
- Instead of allowing only one thread into the CS, it allows at most  $c$  – where  $c$  is the capacity

# Semaphore

```
public class Semaphore {  
    Lock lock;  
    Condition condition;  
    int capacity;  
    int state;  
    ...
```

The max amount  
of threads  
allowed in the CS

# Semaphore

```
public class Semaphore {  
    Lock lock;  
    Condition condition;  
    int capacity;  
    int state;  
    ...
```

The current  
number of  
threads in the  
CS

# Semaphores

```
public void lock() {  
    lock.lock();  
    try {  
        while (state == capacity)  
            condition.await;  
        state++;  
    } finally {  
        lock.unlock();  
    }  
}
```

# Semaphores

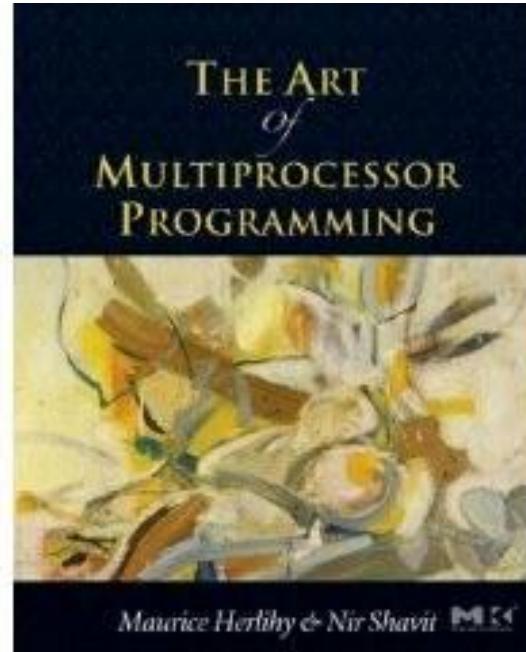
```
public void unlock() {  
    lock.lock();  
    try {  
        state--;  
        condition.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```

COS 226

# Chapter 9

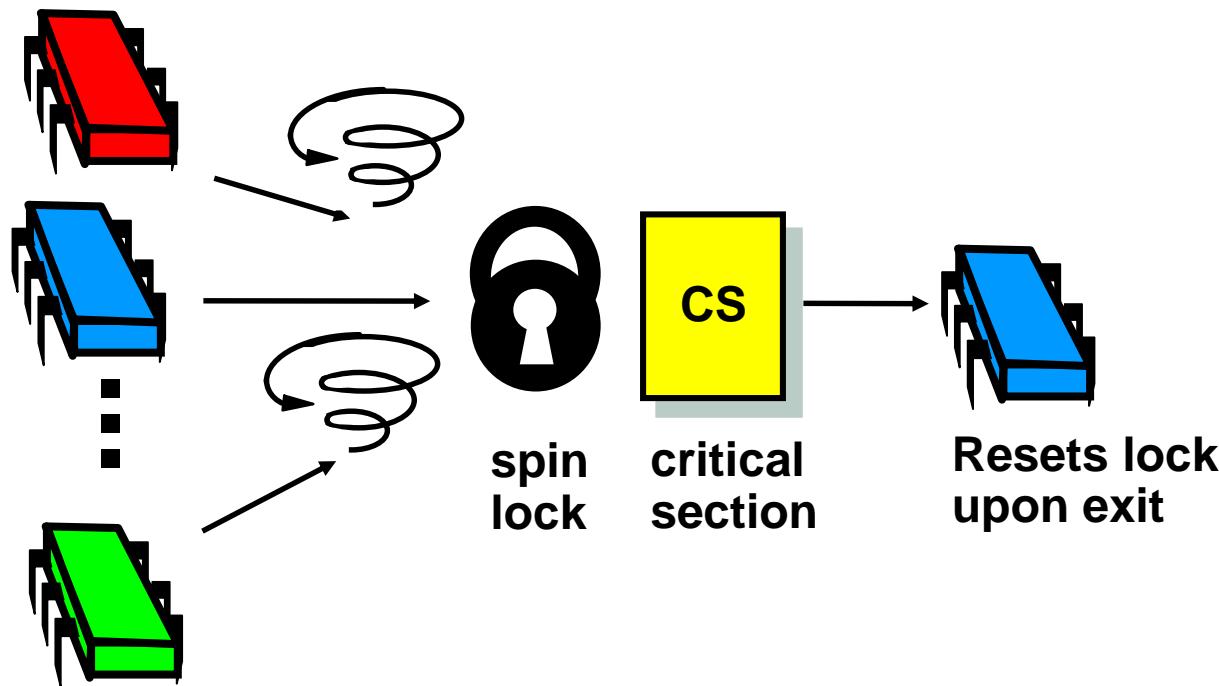
## Linked Lists: The Role of Locking

# Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

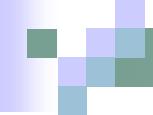
# Last Lecture: Spin-Locks





# Spin locks

- In Chapter 7 we saw how to build scalable spin locks that provide mutual exclusion efficiently

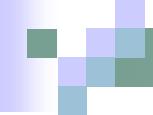


# So, how do we construct a scalable concurrent data structure?

- The most obvious solution would be to take a sequential implementation of the class, add a scalable lock and make sure that every method call acquires and releases the lock
- = coarse-grained synchronization
- What is the potential problem with this?

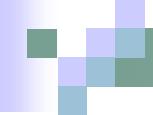
# Problem

- A class that uses a single lock to mediate all its method calls is not always scalable
- Coarse-grained synchronization works well when the level of concurrency is low
- However when too many threads tries to acquire the lock, it forms a bottleneck



# This Chapter

- Introduce four “patterns”
  - Bag of tricks ...
  - Methods that work more than once ...
- For highly-concurrent objects
  - Concurrent access
  - More threads, more throughput



# First: Fine-Grained Synchronization

- Instead of using a single lock ...
- Split object into
  - Independently-synchronized components
- Methods calls interfere only when the access
  - The same component ...
  - At the same time



# Second: Optimistic Synchronization

- Search without locking ...
- If you find it, lock and check ...
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking, but
  - Mistakes are expensive



# Third: Lazy Synchronization

- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done
- Lazy synchronization splits it into these two removal phases



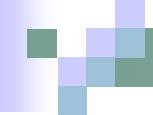
# Fourth: Lock-Free Synchronization

- Don't use locks at all
  - Use compareAndSet() & relatives ...
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead



# Linked List

- Illustrate these patterns ...
- Using a list-based Set
  - Common application
  - Building block for other apps



# Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - **add(x)** put x in set
  - **remove(x)** take x out of set
  - **contains(x)** tests if x in set

# List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
    public boolean remove(T x);  
    public boolean contains(T x);  
}
```

# List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
    public boolean remove(T x);  
    public boolean contains(T x);  
}
```

Returns true if  $x$  was not  
already in set

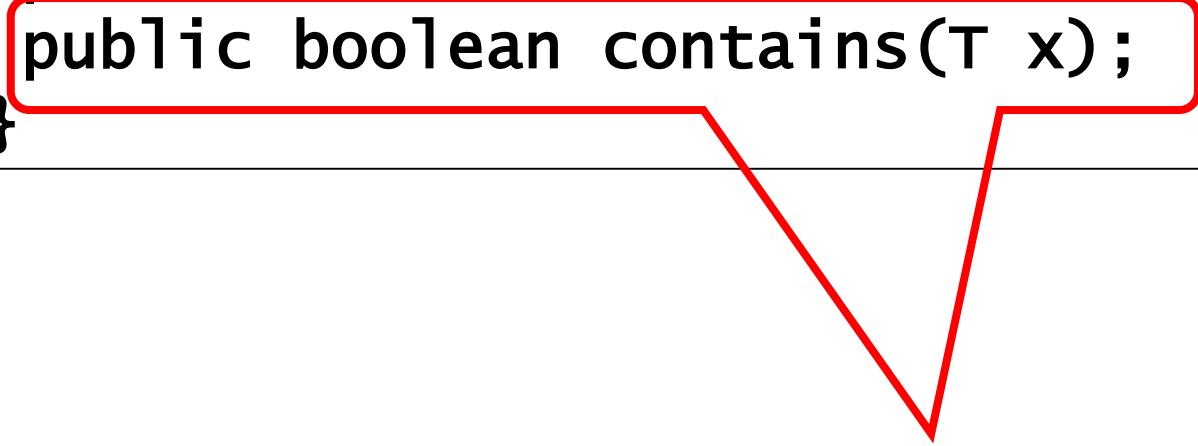
# List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
    public boolean remove(T x);  
    public boolean contains(Tt x);  
}
```

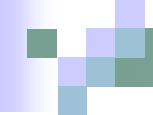
Returns true if  $x$  was in set

# List-Based Sets

```
public interface Set<T> {  
    public boolean add(T x);  
    public boolean remove(T x);  
    public boolean contains(T x);  
}
```



Returns true if  $x$  was in set

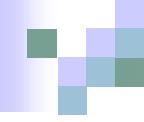


# List-Based Sets

- A set is implemented as a linked list of nodes
- Node<T> has three fields:
  - Item – actual item
  - Key – item's hash code, nodes are sorted according to key
  - Next – reference to next node in list

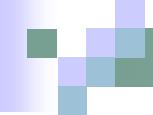
# List Node

```
public class Node {  
    public T item;  
    public int key;  
    public Node next;  
}
```



# List-Based Sets

- Lists has two types of nodes:
  - Regular nodes – hold items
  - Sentinel nodes – head and tail
- Each thread that traverses through the list use:
  - curr – a “pointer” to the current node
  - pred – a “pointer” to the node’s predecessor

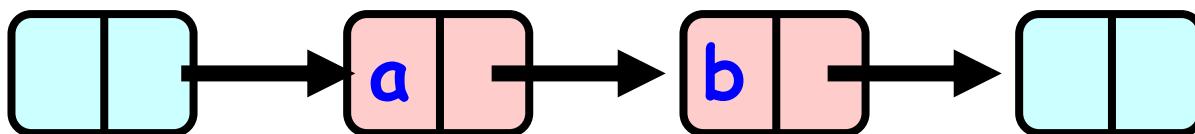


# Freedom from interference

- We assume that `add()`, `remove()` and `contains()` are the only methods that can modify nodes
- We also assume that sentinel nodes cannot be added or removed
- And nodes are sorted by keys and keys are unique

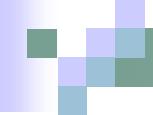
# Reasoning about Concurrent objects

- Concrete representation:



- Abstract Value:

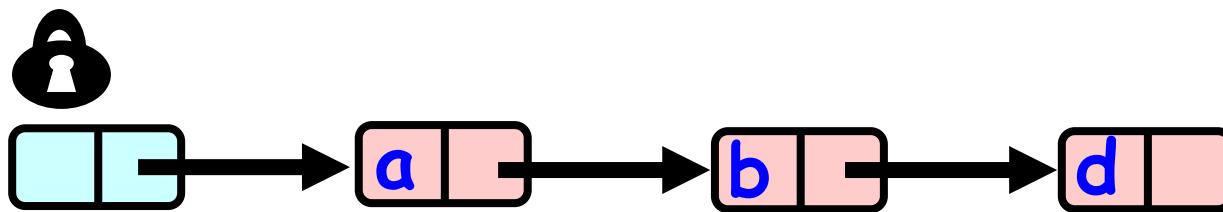
$\square \{a, b\}$



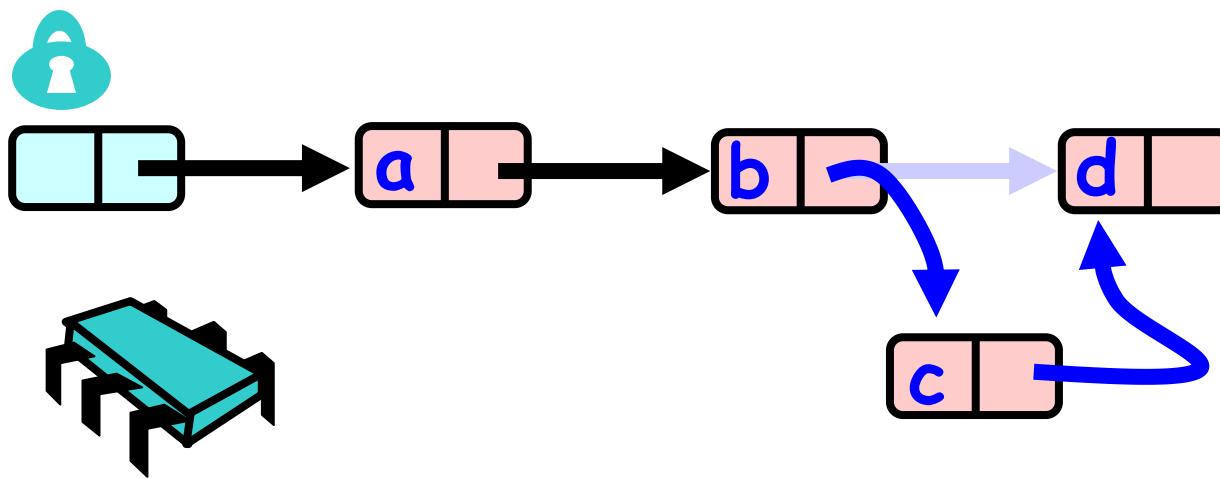
# Safety and Liveness?

- Safety:
  - Linearizability
- Liveness
  - Deadlock-free
  - Starvation-free
  - Nonblocking?

# Coarse Grained Locking



# Coarse Grained Locking





# Coarse-grained synchronization

- One concurrent data structure
- One lock
  - Method acquires and releases lock with each access
- Multiple threads

# Coarse-grained synchronization

```
public class CoarseList<T> {  
    private Node head;  
    private Lock lock = new ReentrantLock();  
    public CoarseList() {  
        head = new Node(Integer.MIN_VALUE);  
        head.next = new  
        Node(Integer.MAX_VALUE);  
    }  
}
```

```
public boolean add(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key)  
            return false;  
        else {  
            Node node = new Node(item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

```
public boolean add(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key)  
            return false;  
        else {  
            Node node = new Node(item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

Acquire lock

```
public boolean add(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key)  
            return false;  
        else {  
            Node node = new Node(item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

Starting positions  
for pred and  
curr

```
public boolean add(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key)  
            return false;  
        else {  
            Node node = new Node(item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

Traverse through list to  
find correct position

```
public boolean add(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key)  
            return false;  
        else {  
            Node node = new Node(item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

If element already exists  
return false

```
public boolean add(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key)  
            return false;  
        else {  
            Node node = new Node(item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

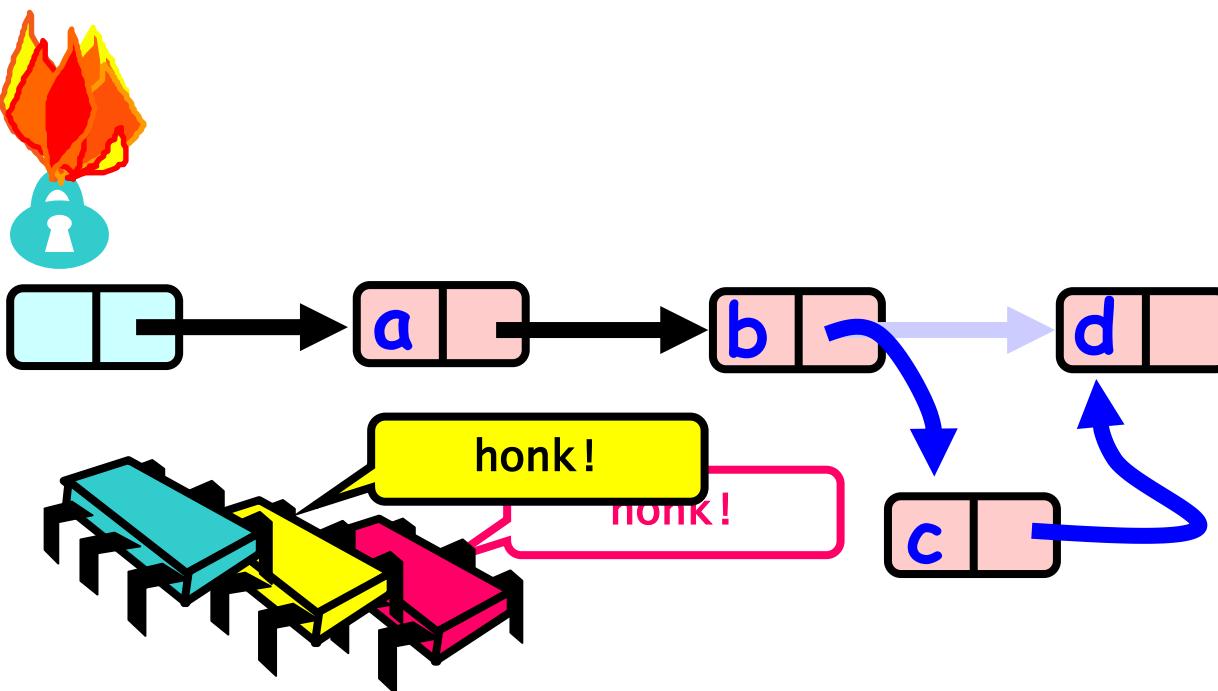
Create node and  
insert in list

```
public boolean add(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key)  
            return false;  
        else {  
            Node node = new Node(item);  
            node.next = curr;  
            pred.next = node;  
            return true;  
        }  
    } finally {  
        lock.unlock();  
    }  
}
```

**Release lock**

```
public boolean remove(T item) {  
    Node pred, curr;  
    int key = item.hashCode();  
    lock.lock();  
    try {  
        pred = head;  
        curr = pred.next;  
        while (curr.key < key) {  
            pred = curr;  
            curr = curr.next;  
        }  
        if (key == curr.key) {  
            pred.next = curr.next;  
            return true;  
        } else  
            return false;  
    } finally {  
        lock.unlock();  
    }  
}
```

# Coarse Grained Locking

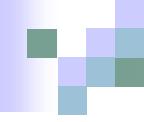


Simple but hotspot + bottleneck



# Coarse-grained synchronization

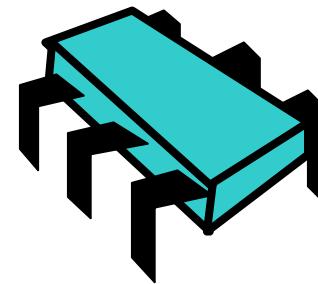
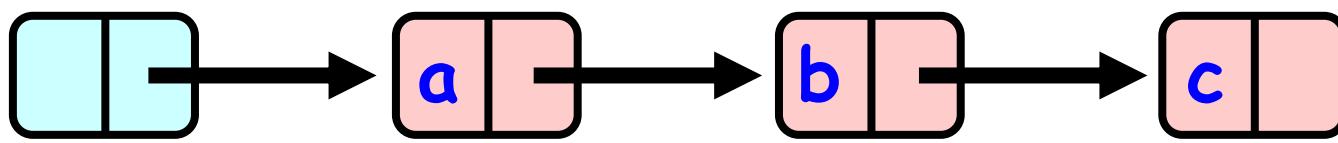
- Easy to implement
- Simple, clear and correct
  - Deserves respect!
- But, works poorly with high contention
  - Queue locks help
  - But bottlenecks still an issue



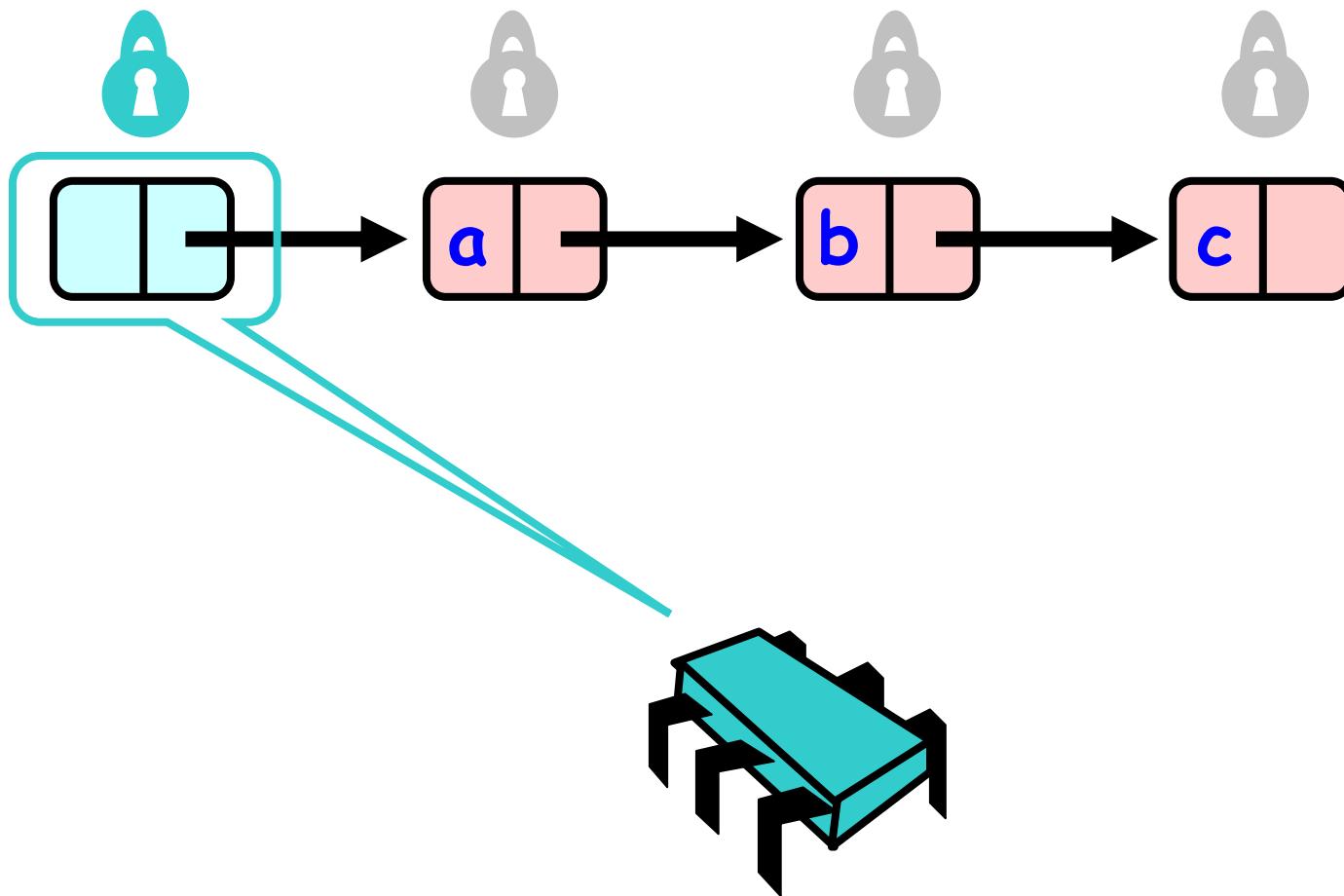
# Fine-grained synchronization

- Instead of locking the list as a whole, place a lock on each entry
- Split object into pieces
  - Each piece has own lock
  - As thread traverses list, he locks each entry with its first visit and unlocks it later
- Concurrent threads can now traverse the list together

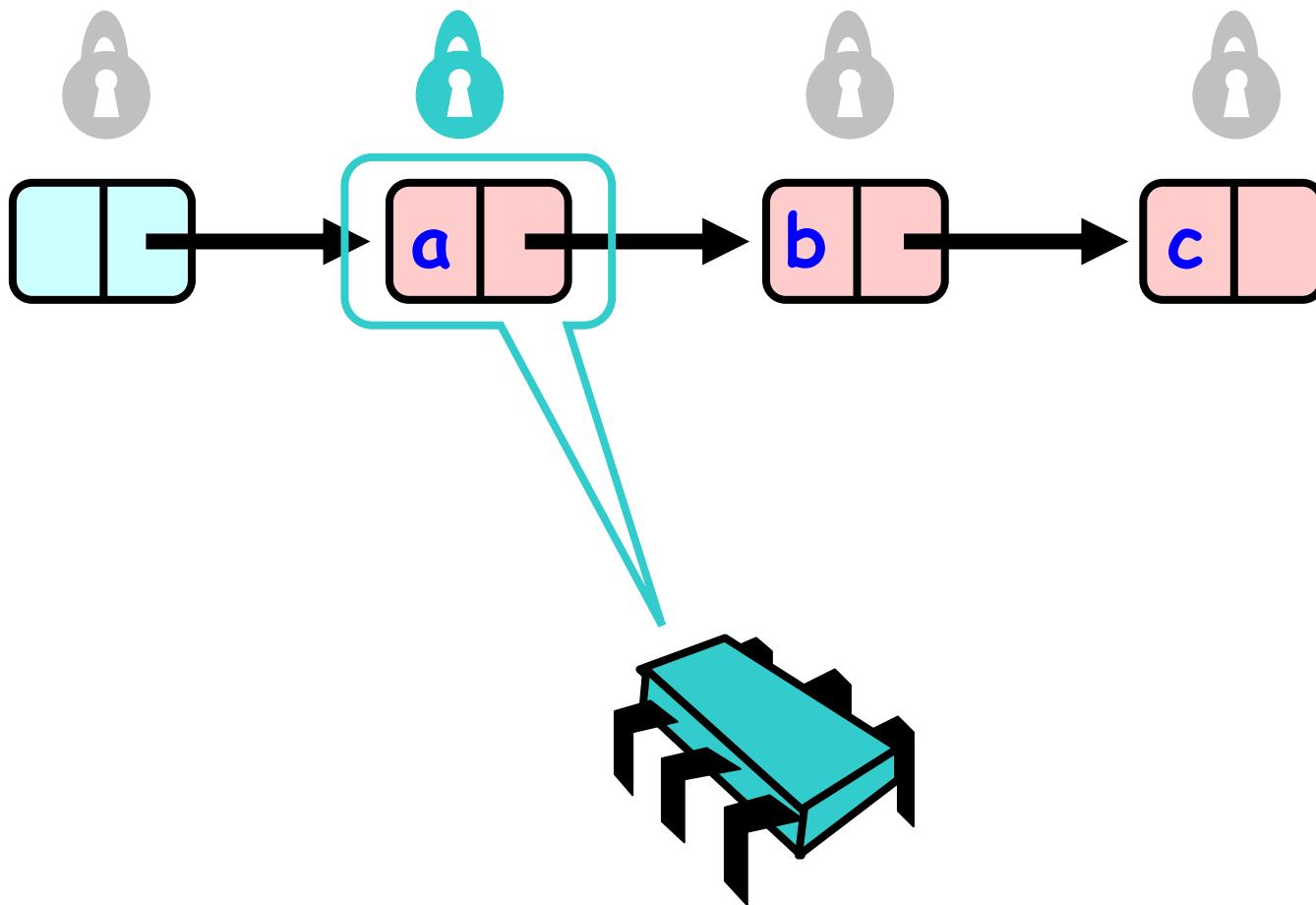
# Fine-grained synchronization



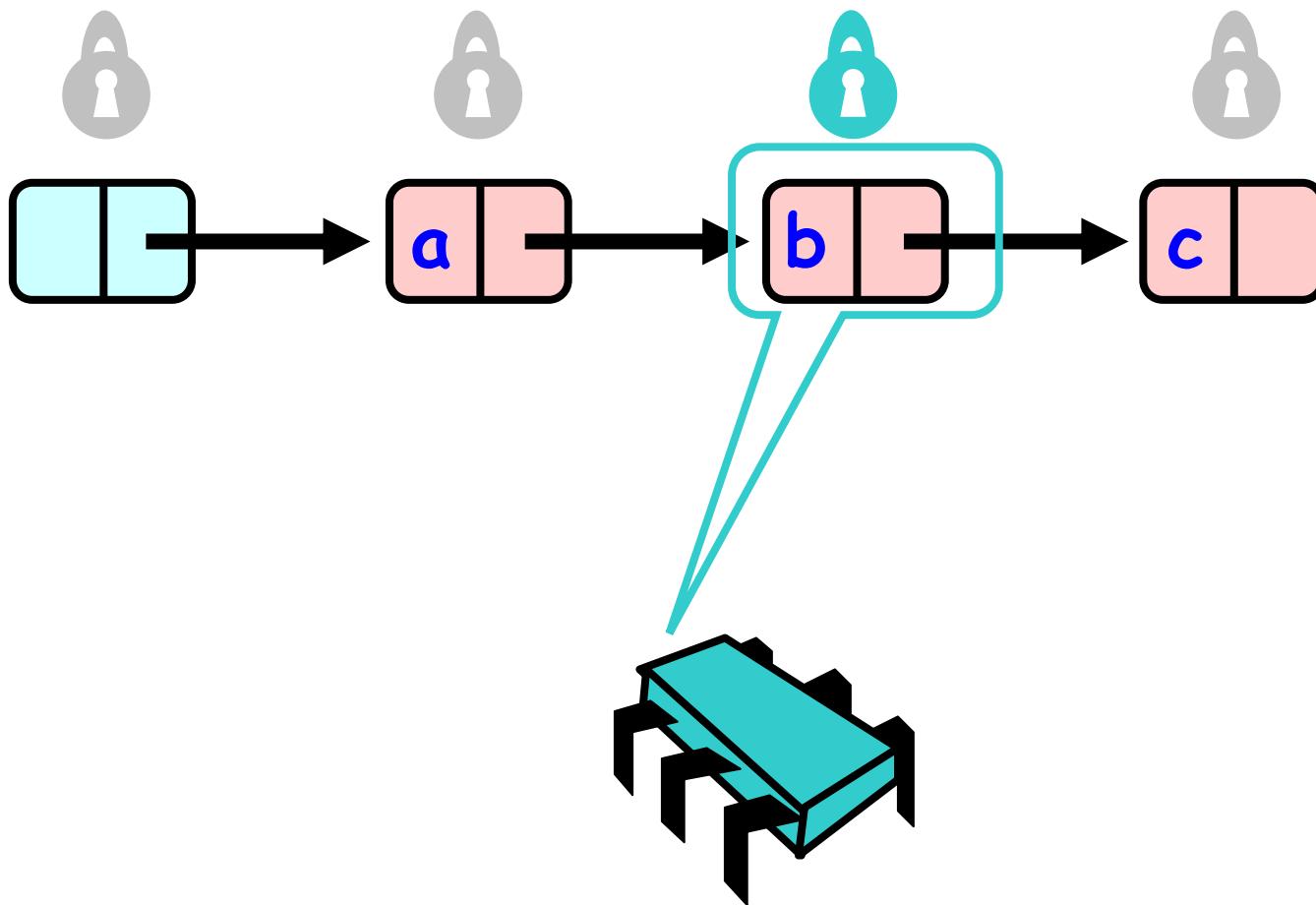
# Fine grained synchronization



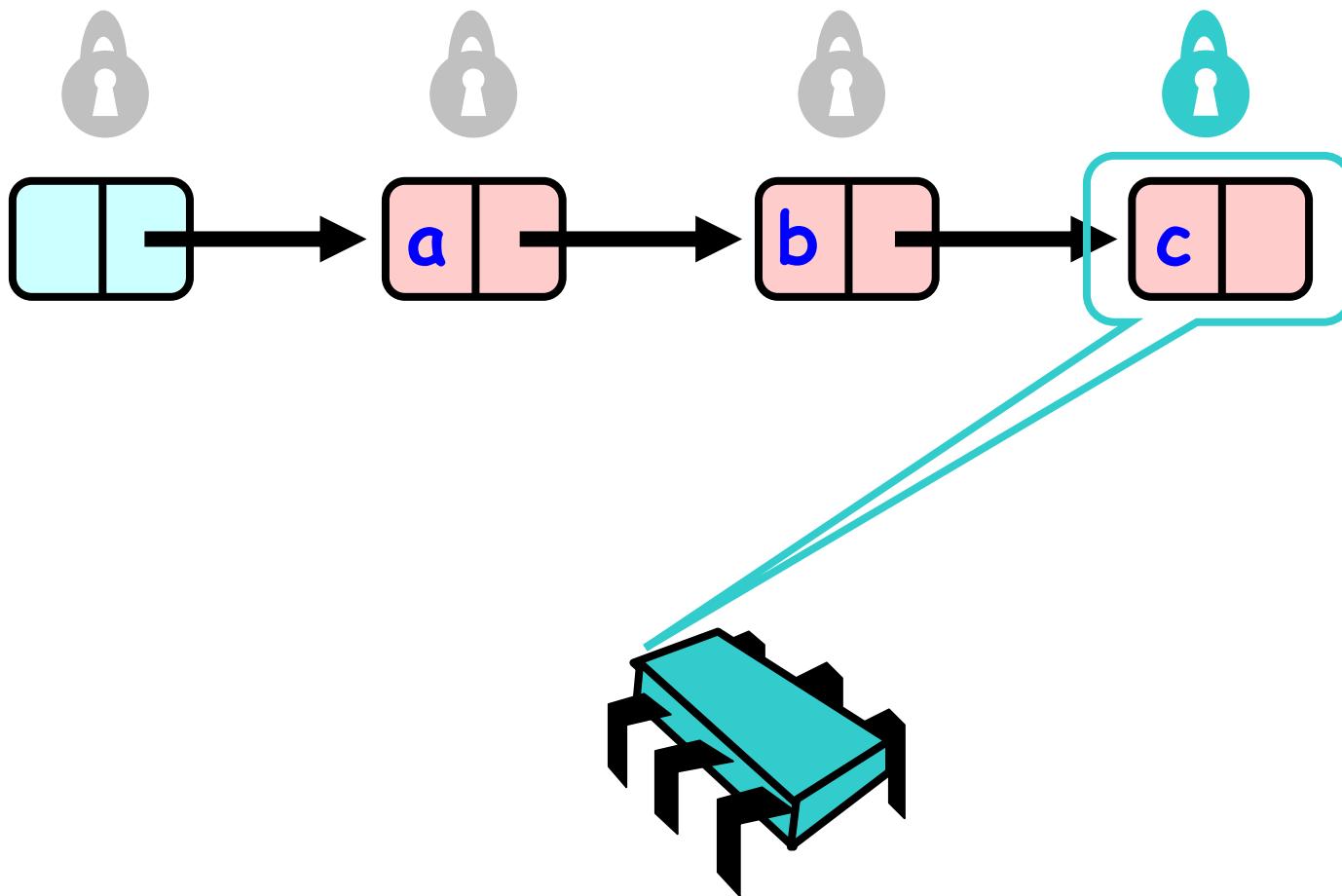
# Hand-over-Hand locking



# Fine-grained synchronization



# Fine-grained synchronization

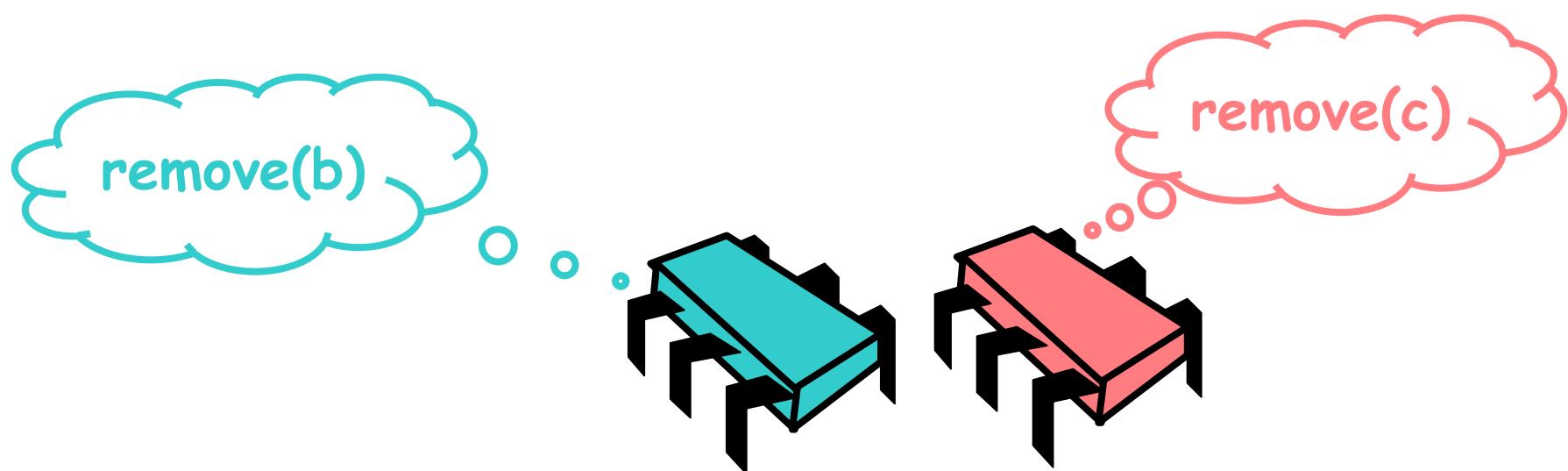
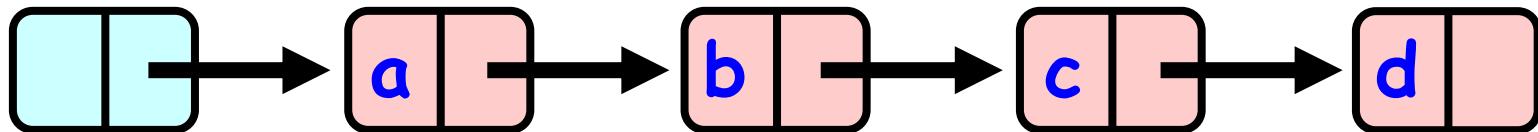




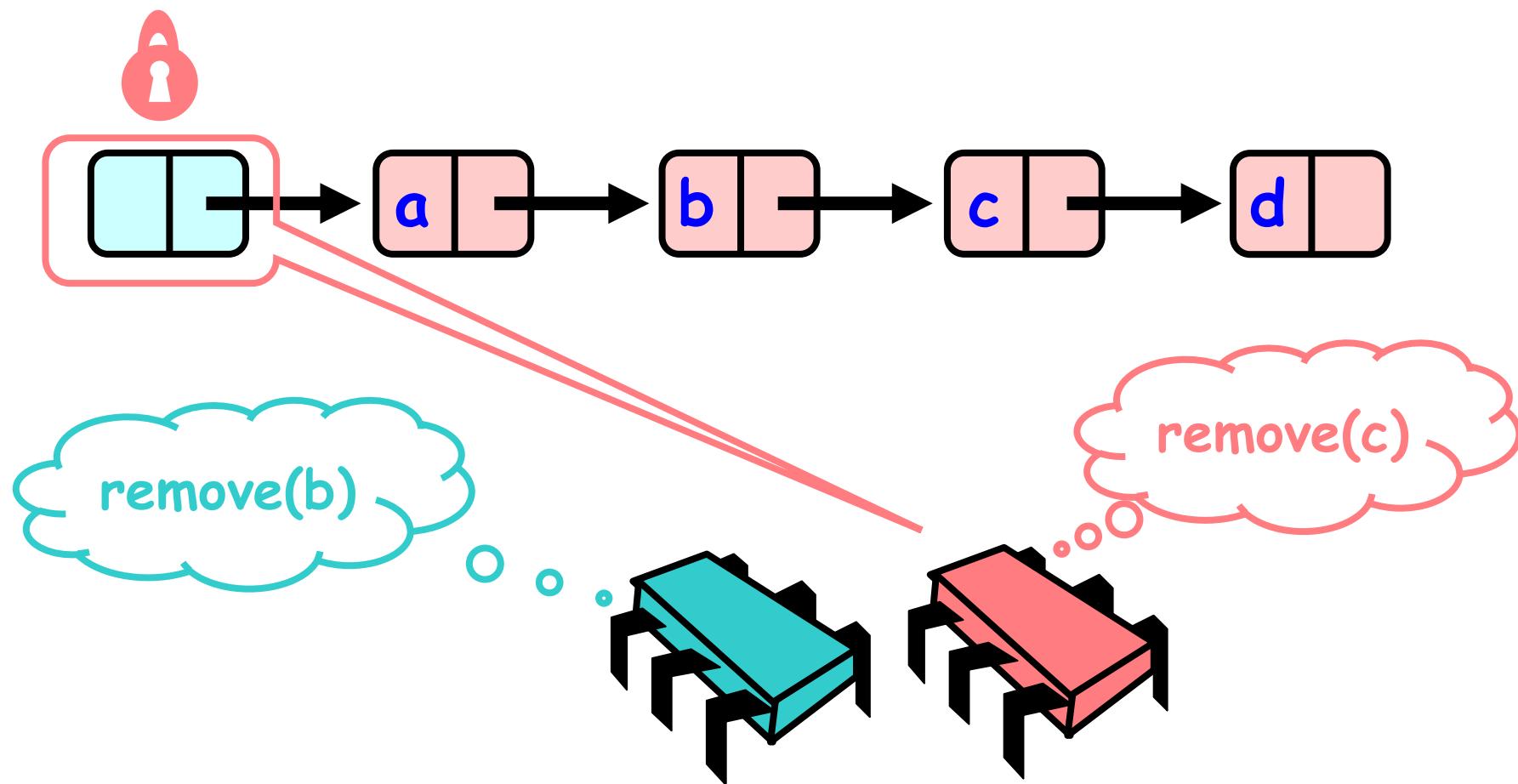
# Fine-grained synchronization

- However, it is unsafe to unlock a before locking b

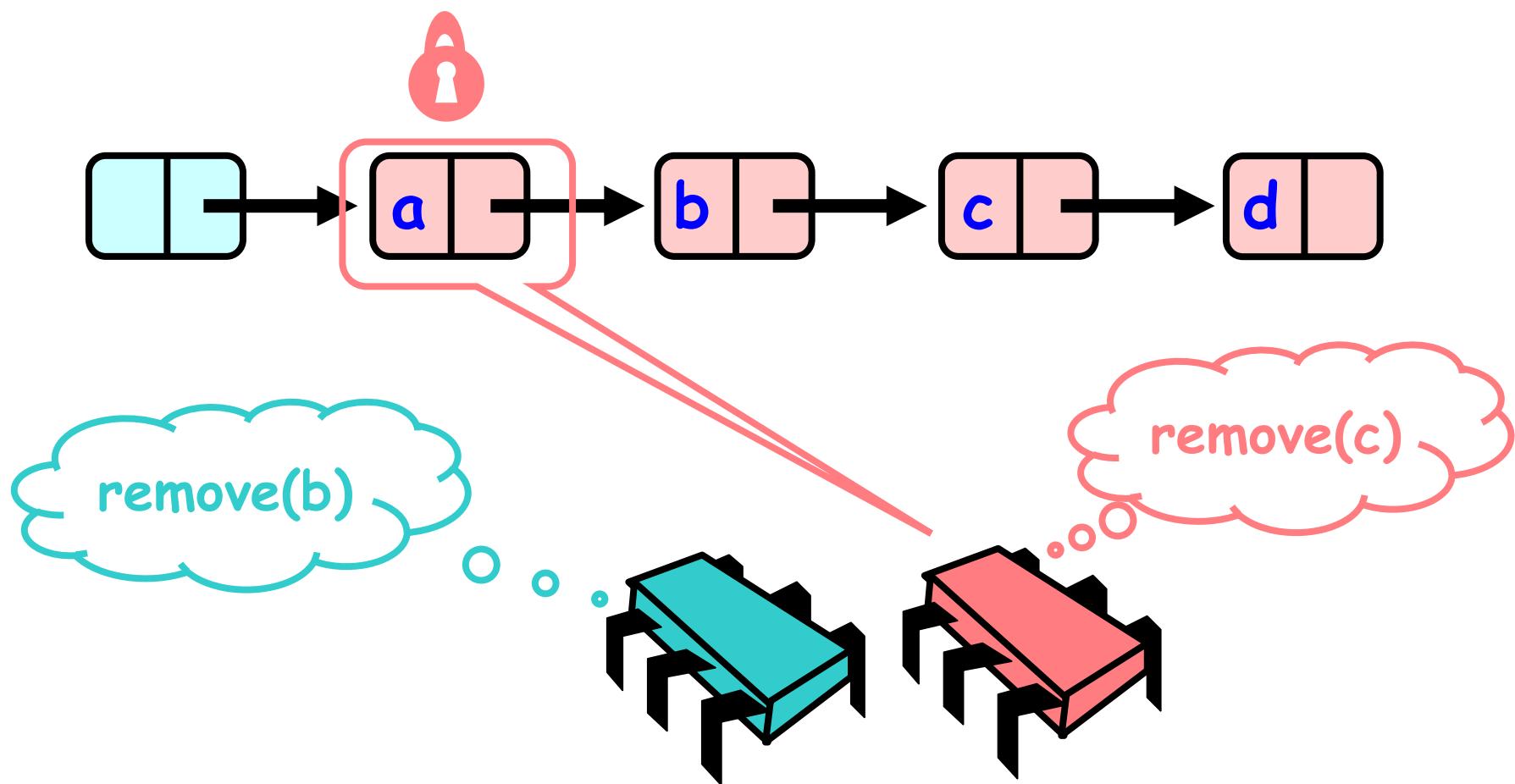
# Concurrent Removes



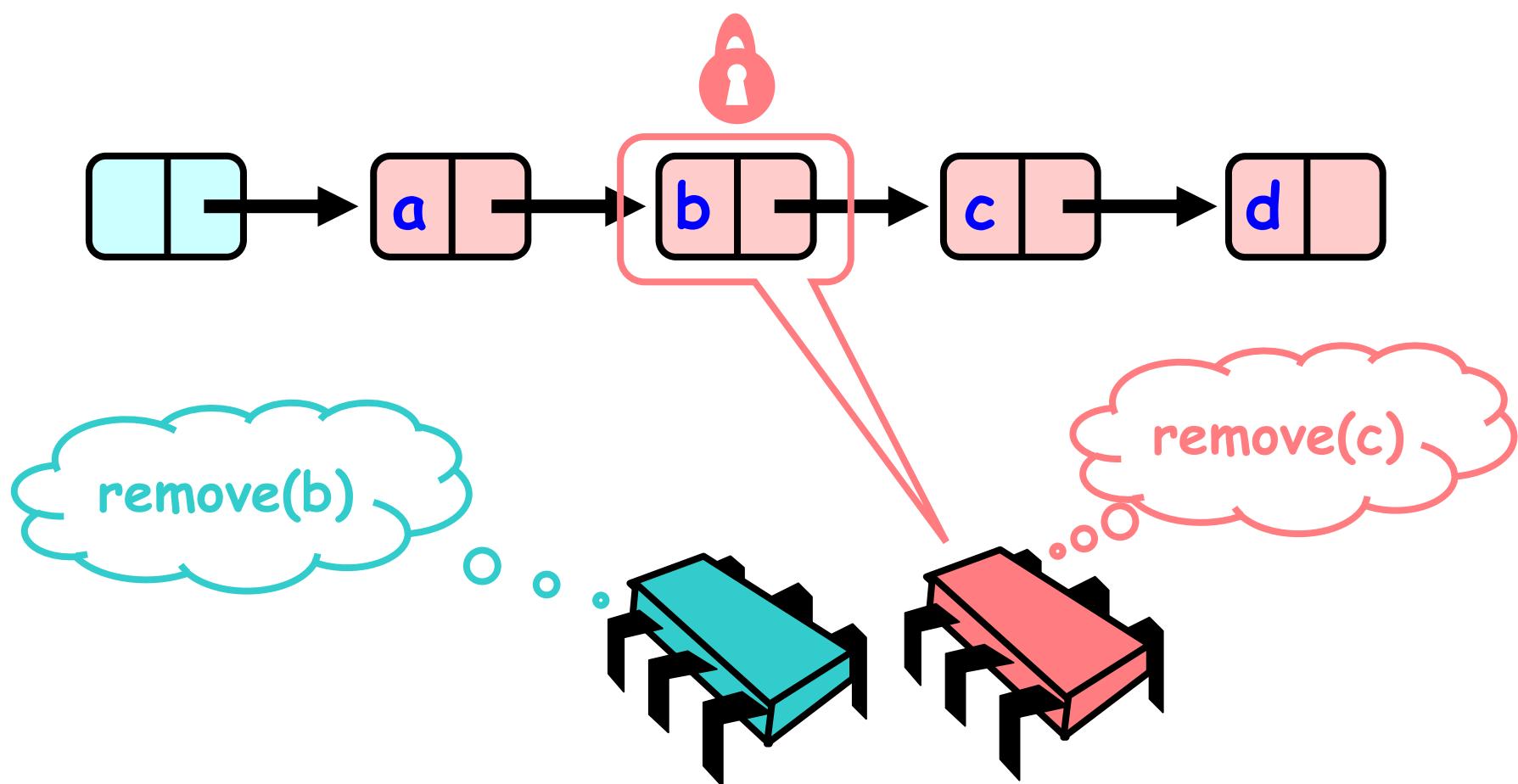
# Concurrent Removes



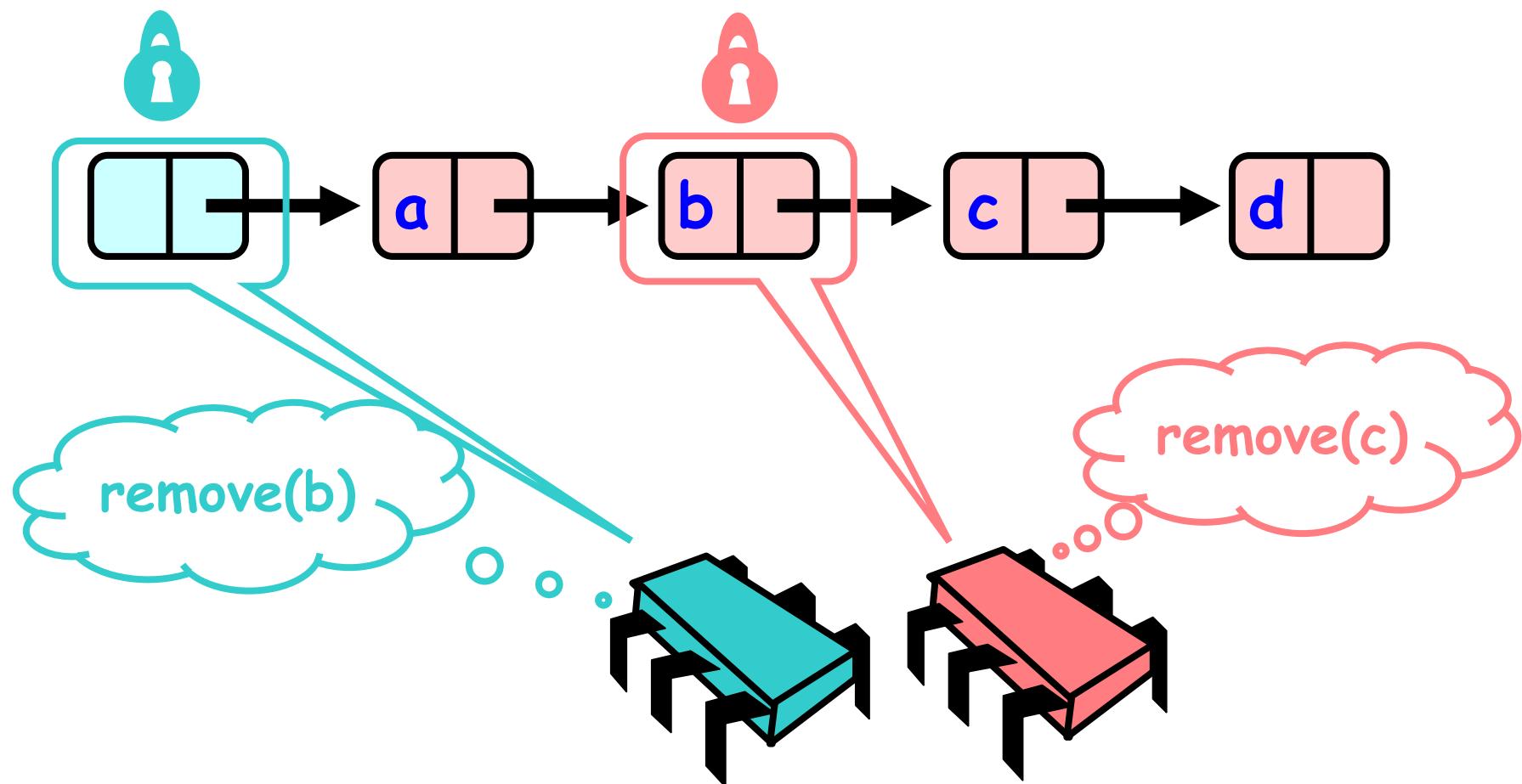
# Concurrent Removes



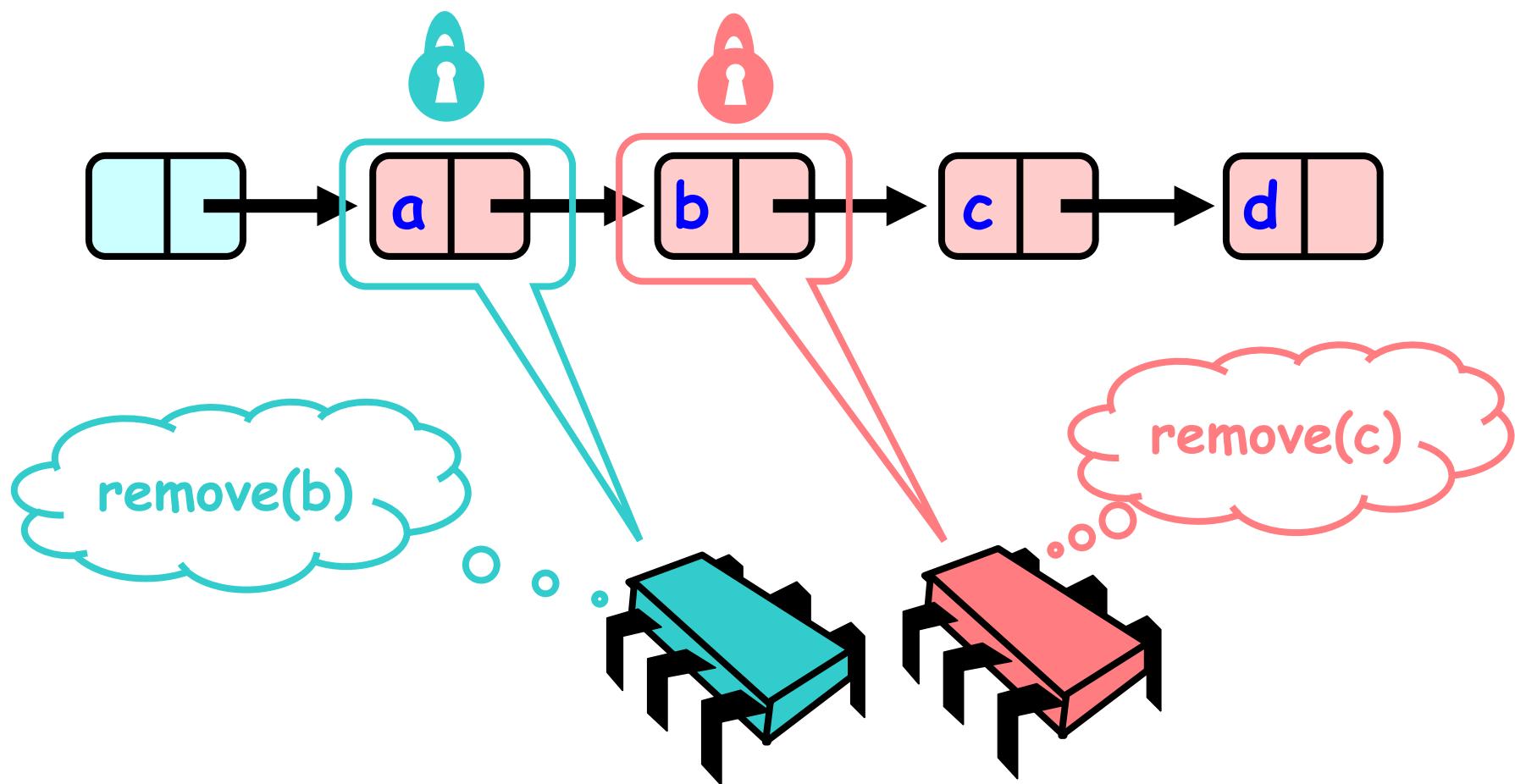
# Concurrent Removes



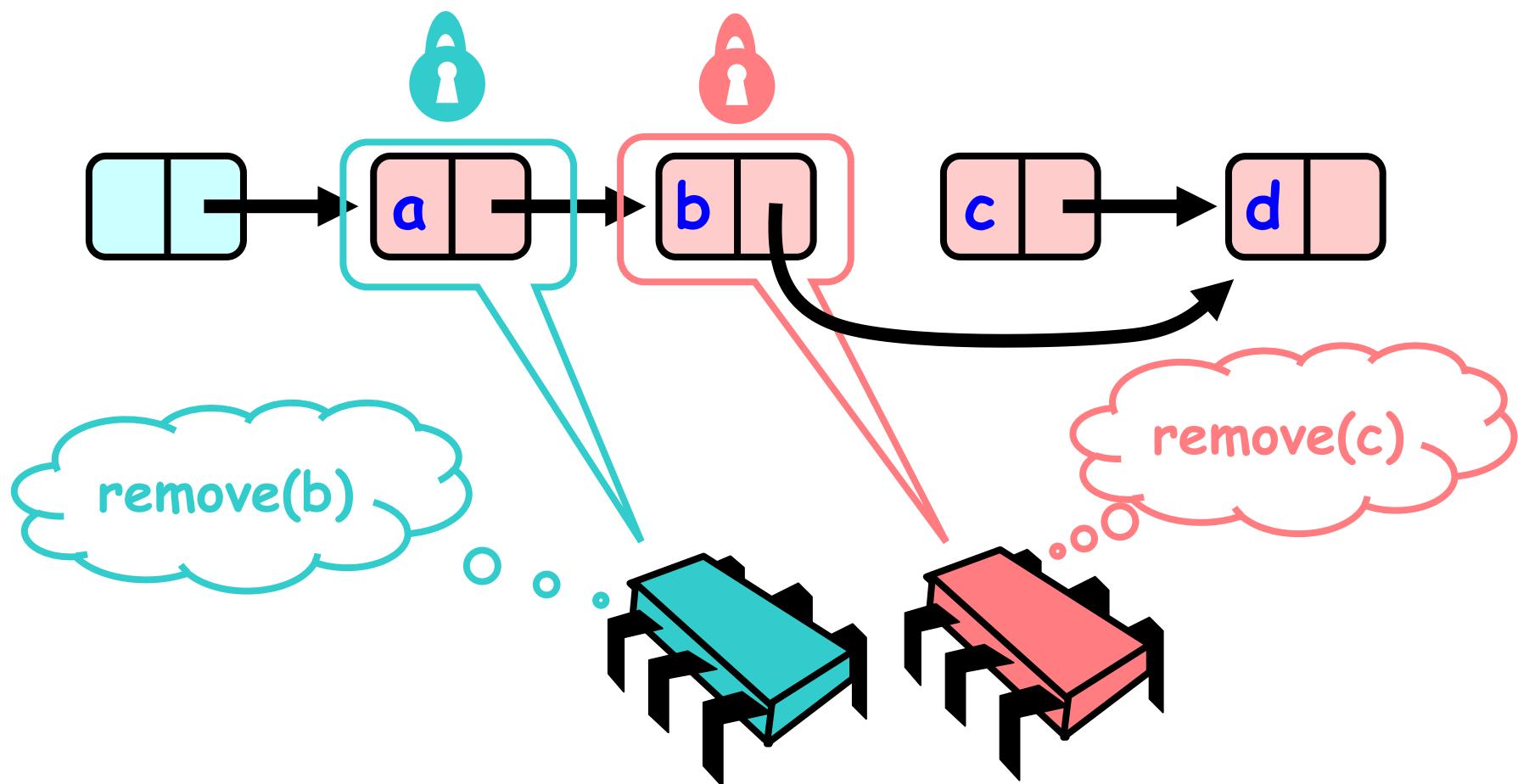
# Concurrent Removes



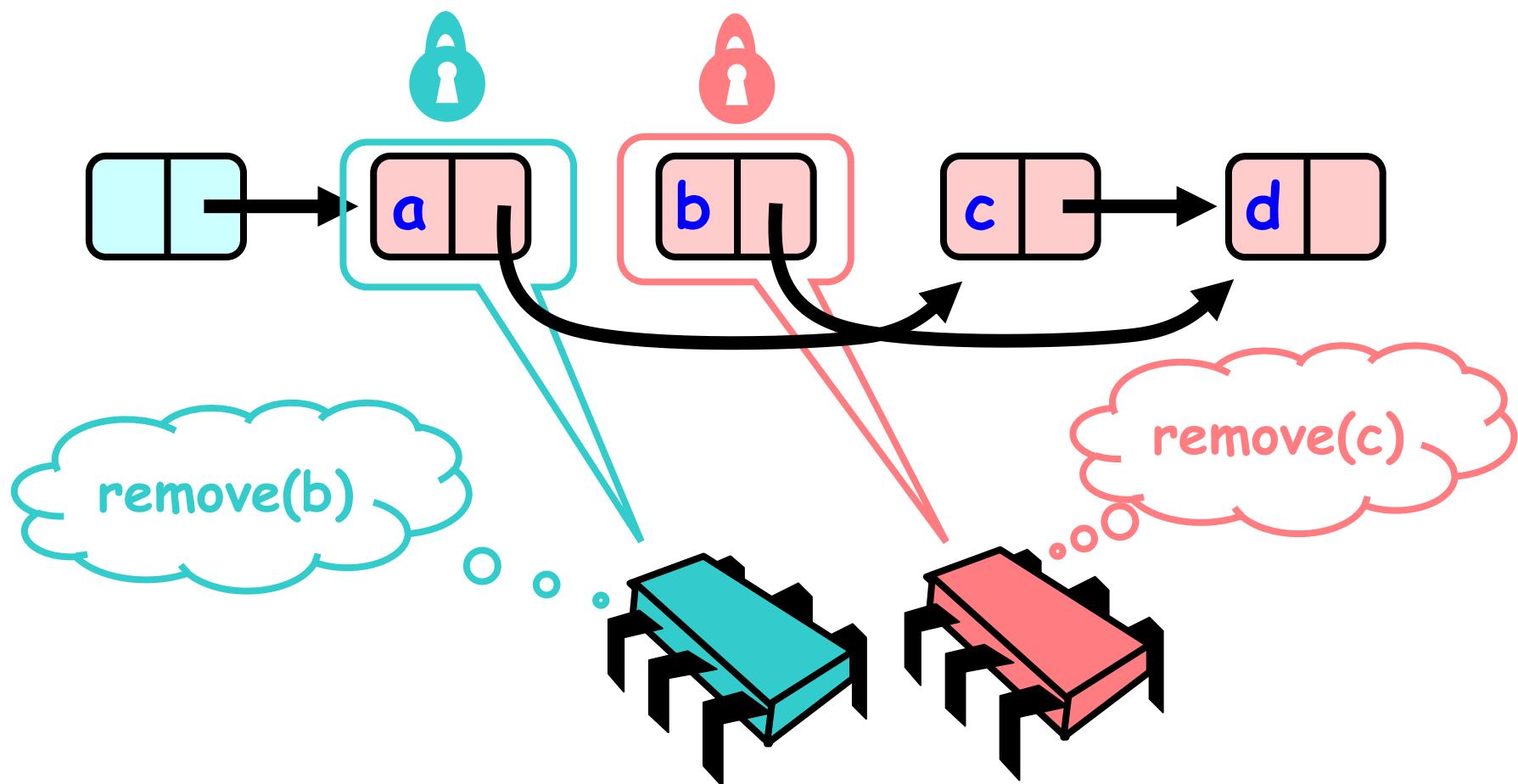
# Concurrent Removes



# Concurrent Removes



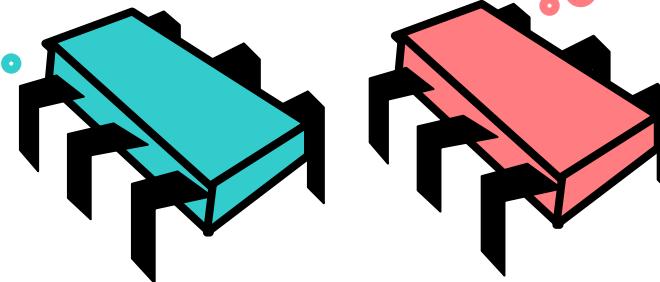
# Concurrent Removes



# Uh, Oh



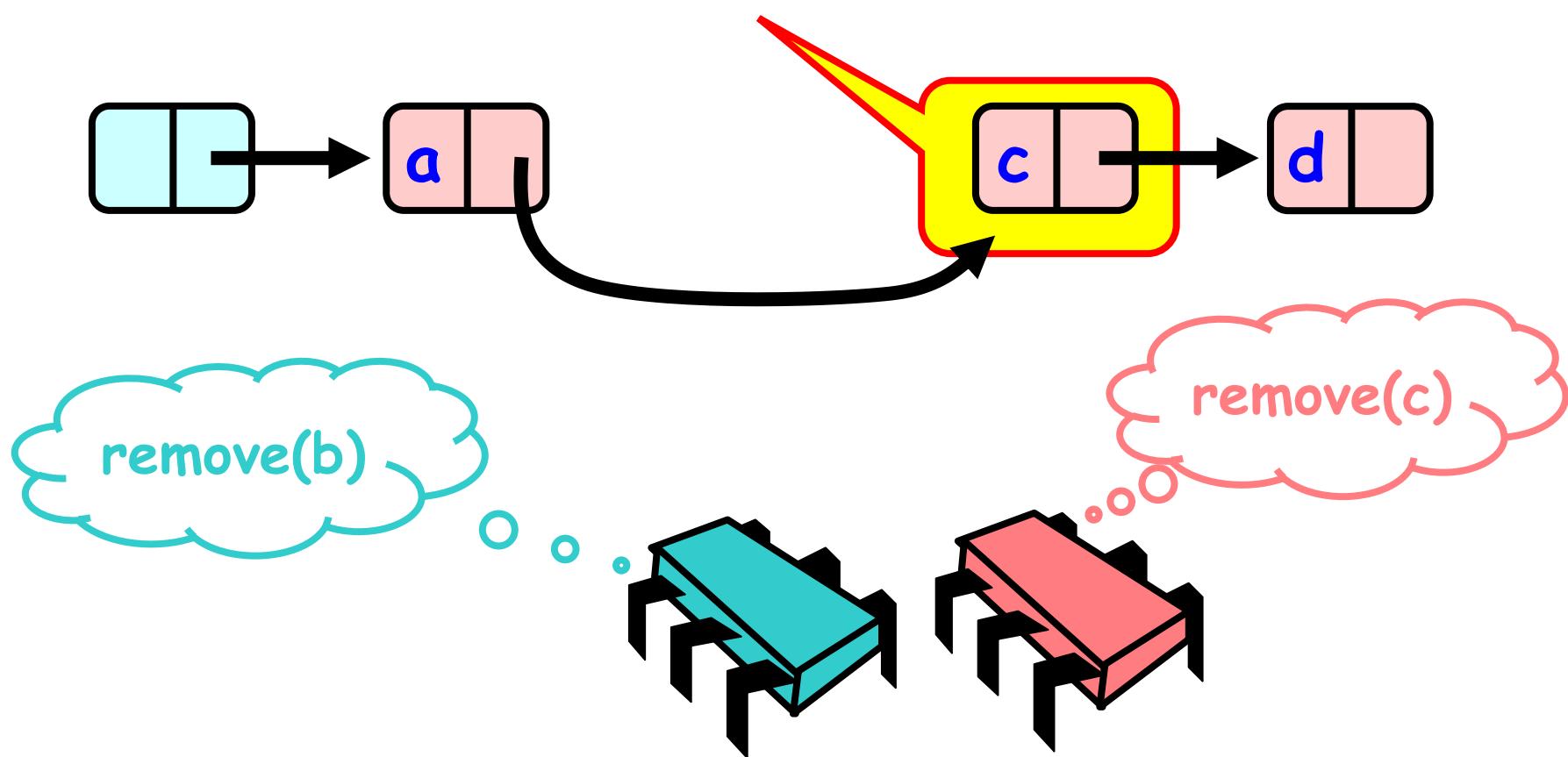
remove(b)



remove(c)

# Uh, Oh

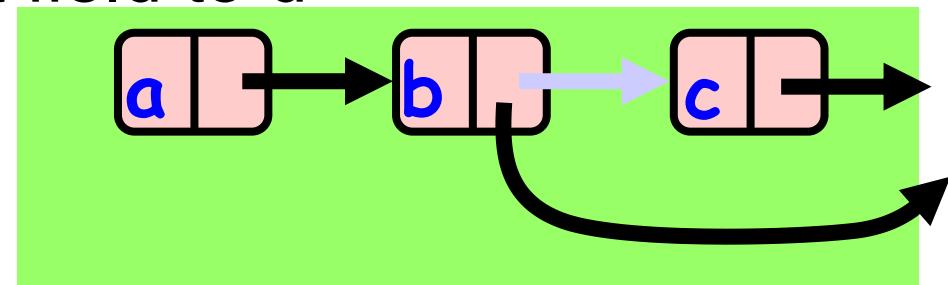
Bad news, c not removed



# Problem

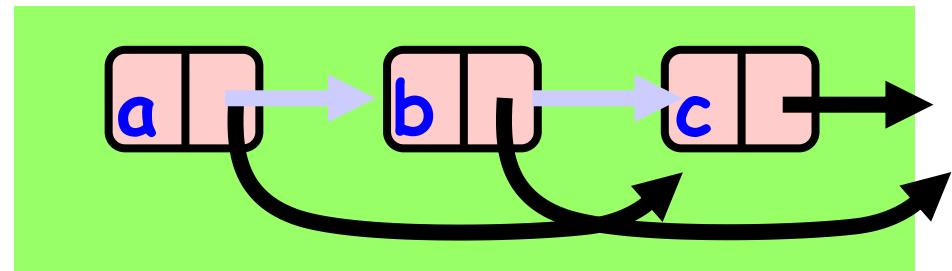
- To delete node c

- Swing node b's next field to d



- Problem is,

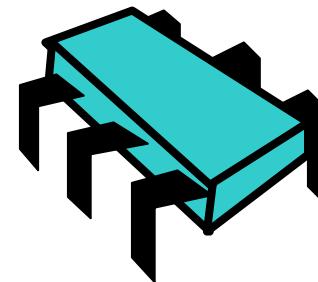
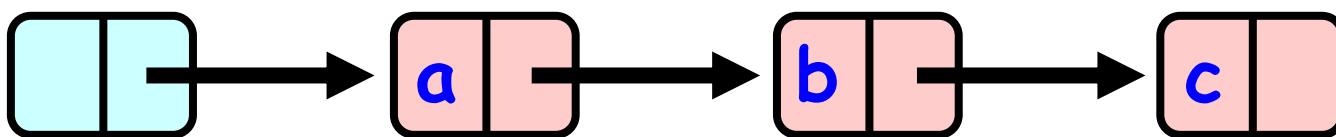
- Someone deleting b concurrently could direct a pointer to c



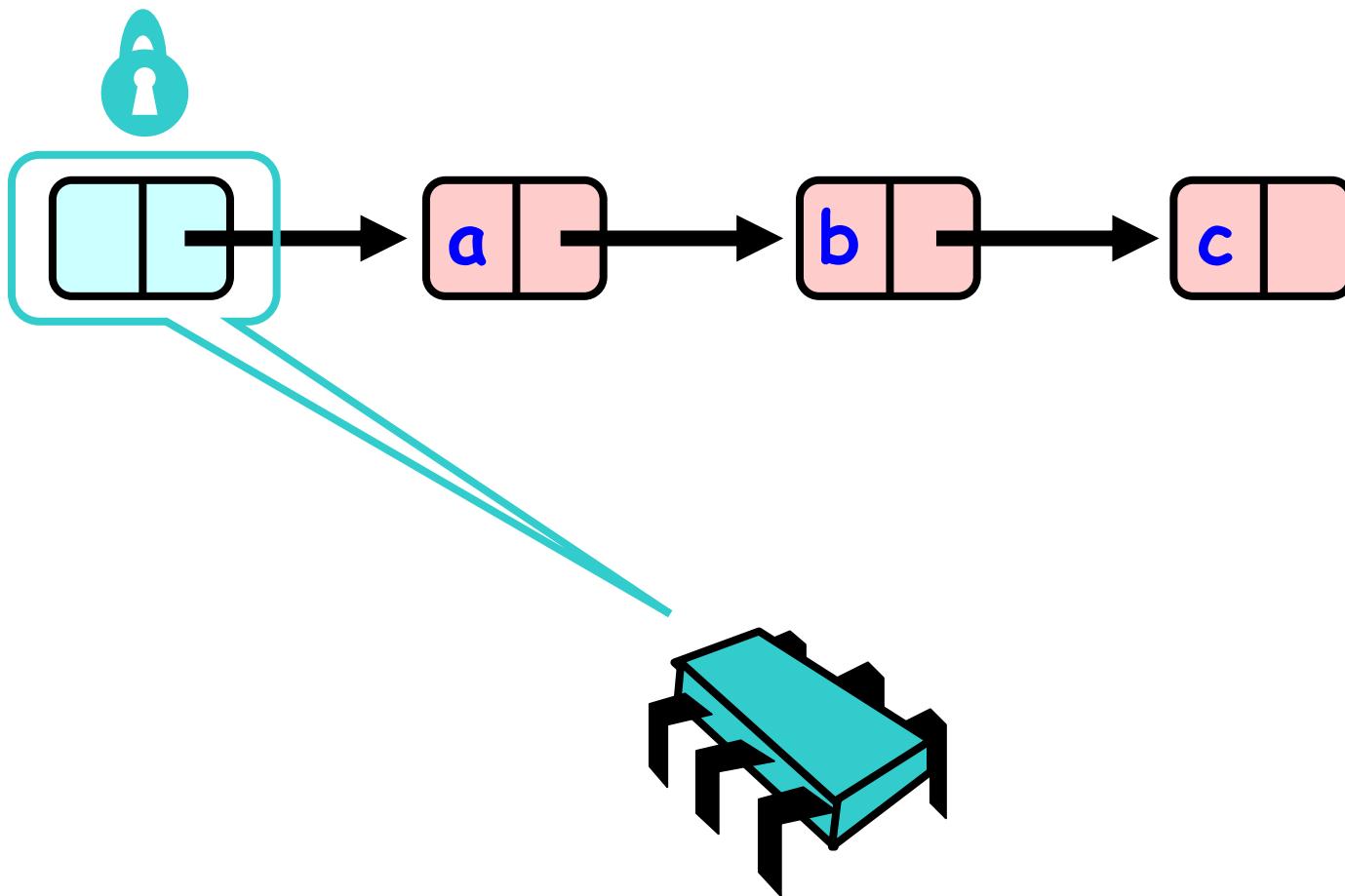
# Solution

- Hand-over-hand locking
  - Except for the initial head sentinel node, acquire the next lock while holding the previous lock
  - In other words, hold two locks at a time

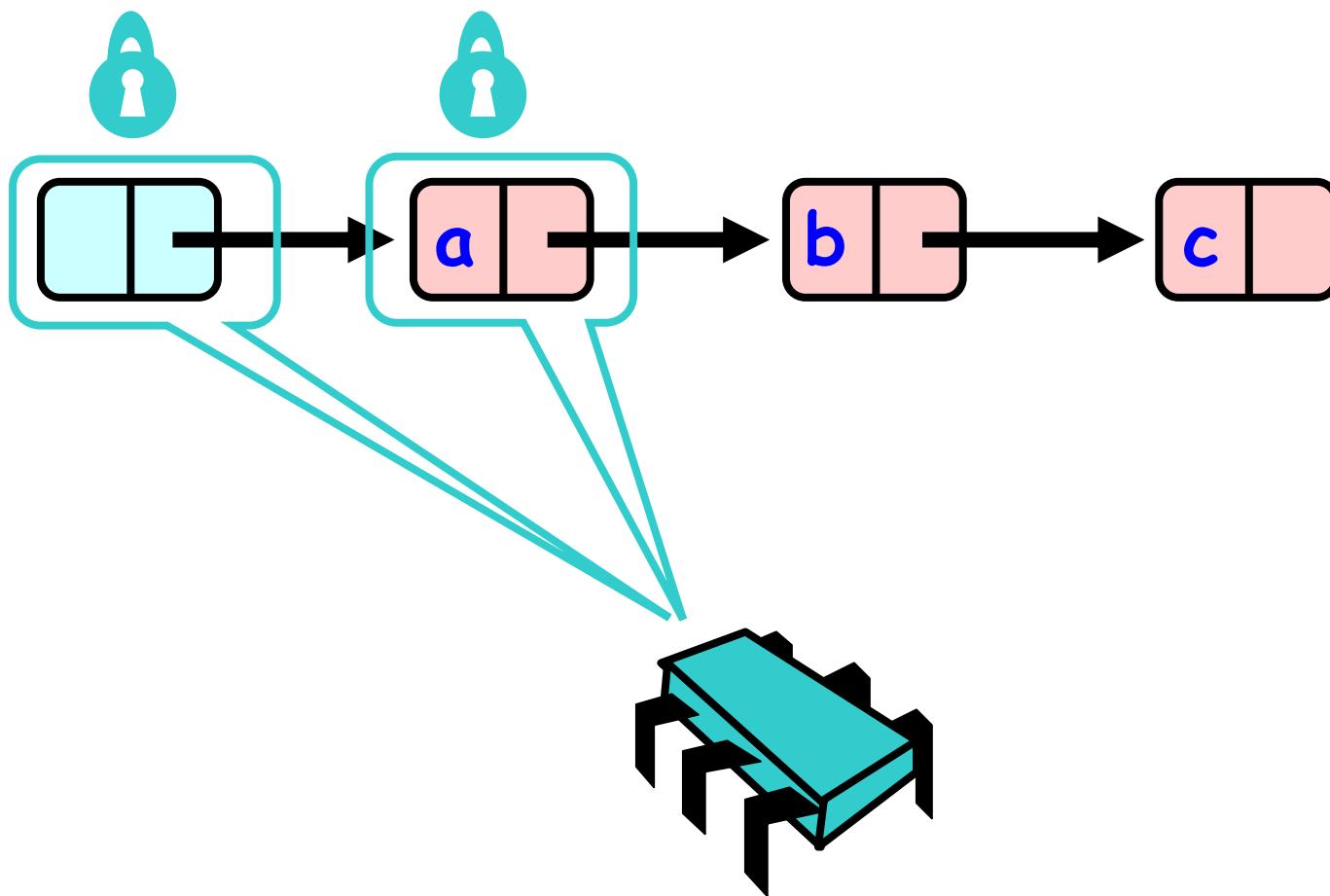
# Hand-over-Hand locking



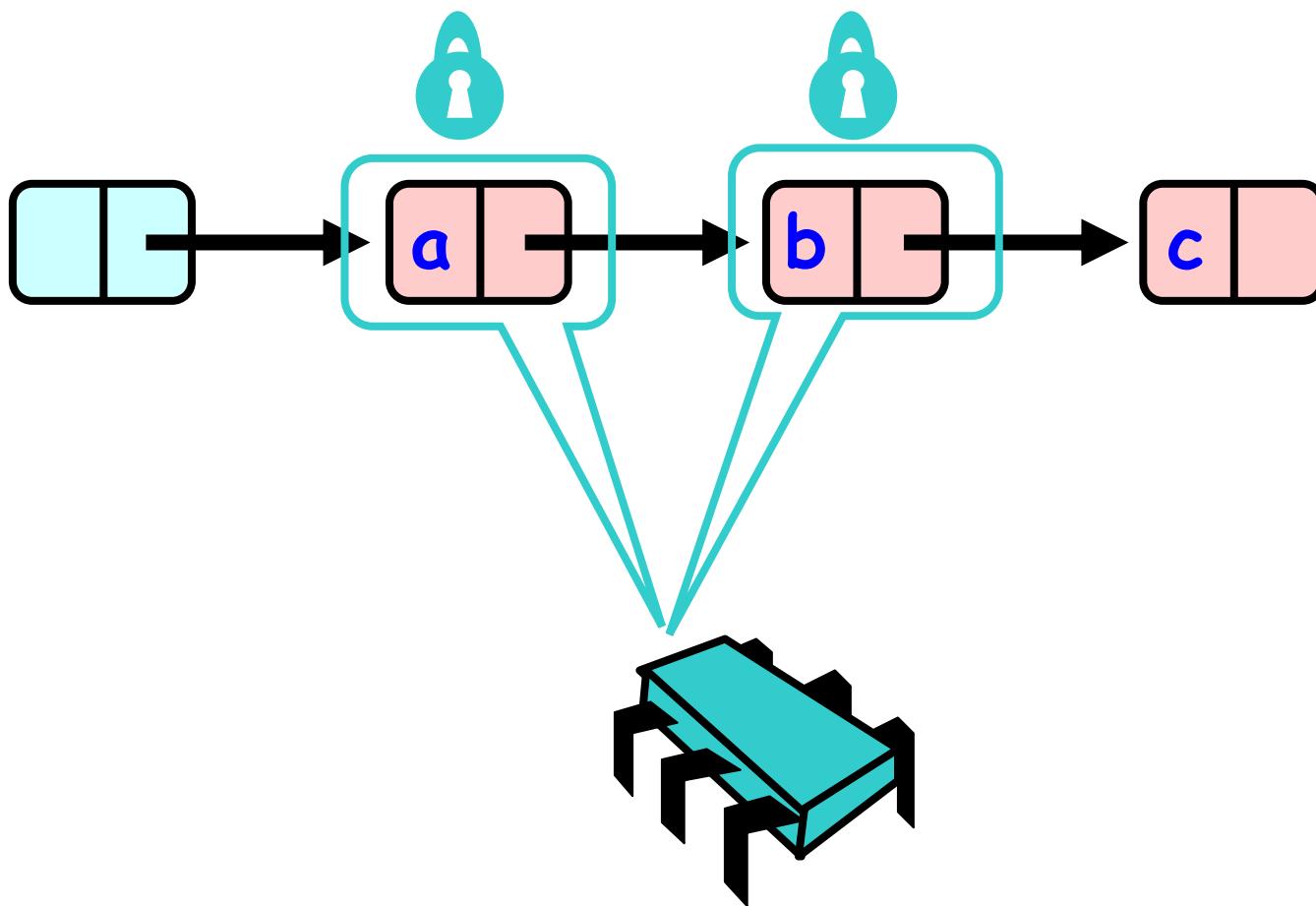
# Hand-over-Hand locking



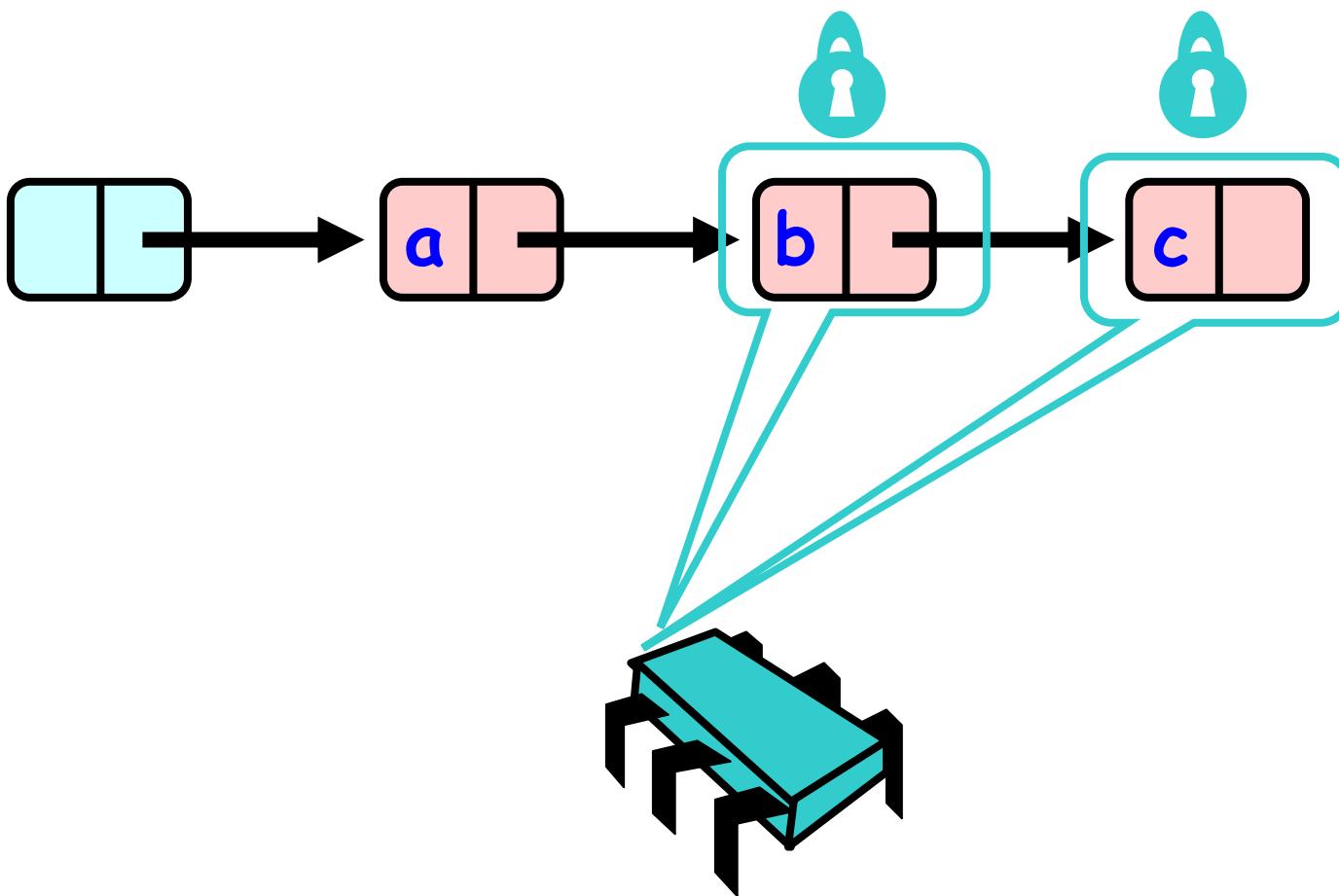
# Hand-over-Hand locking



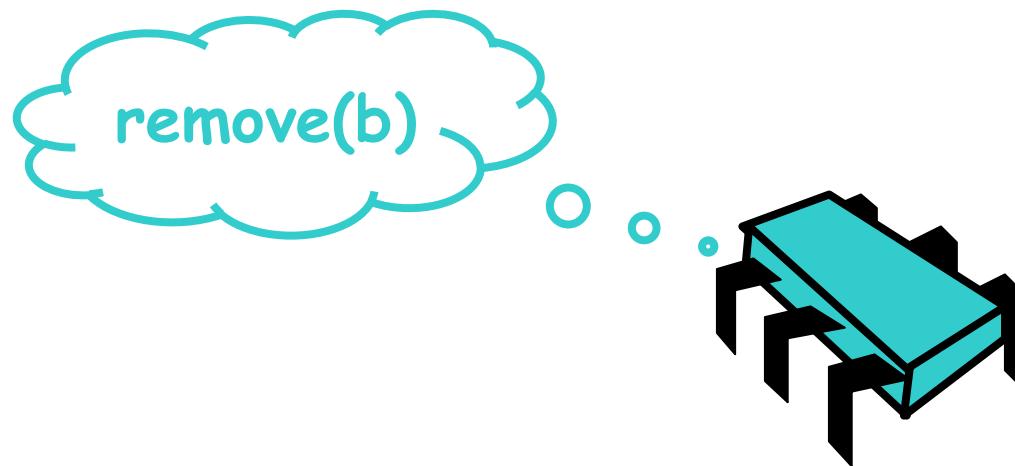
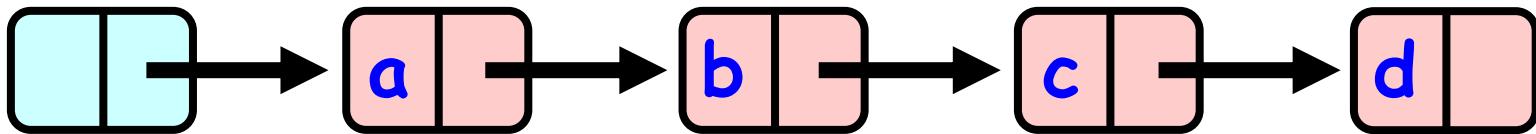
# Hand-over-Hand locking



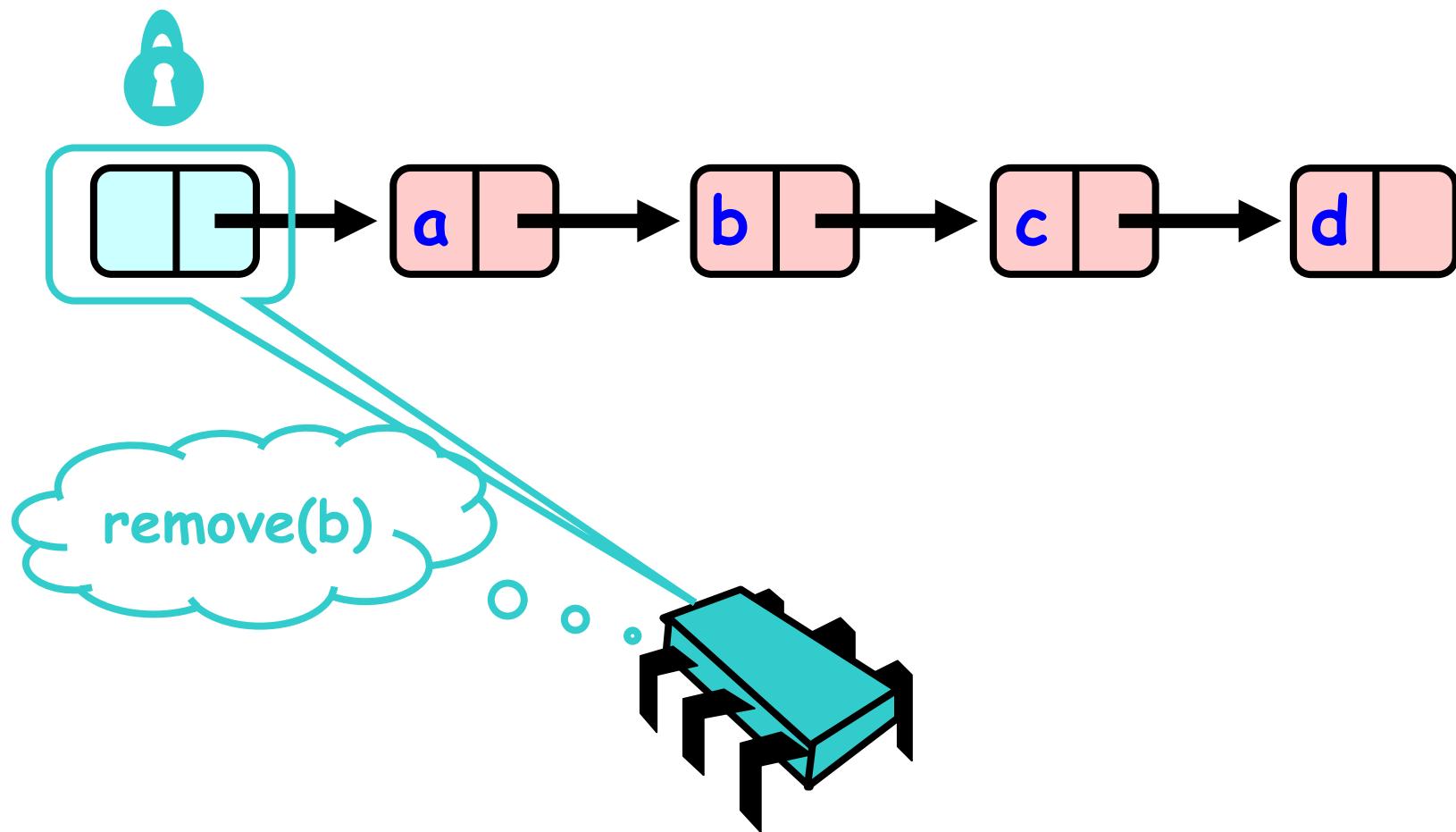
# Hand-over-Hand locking



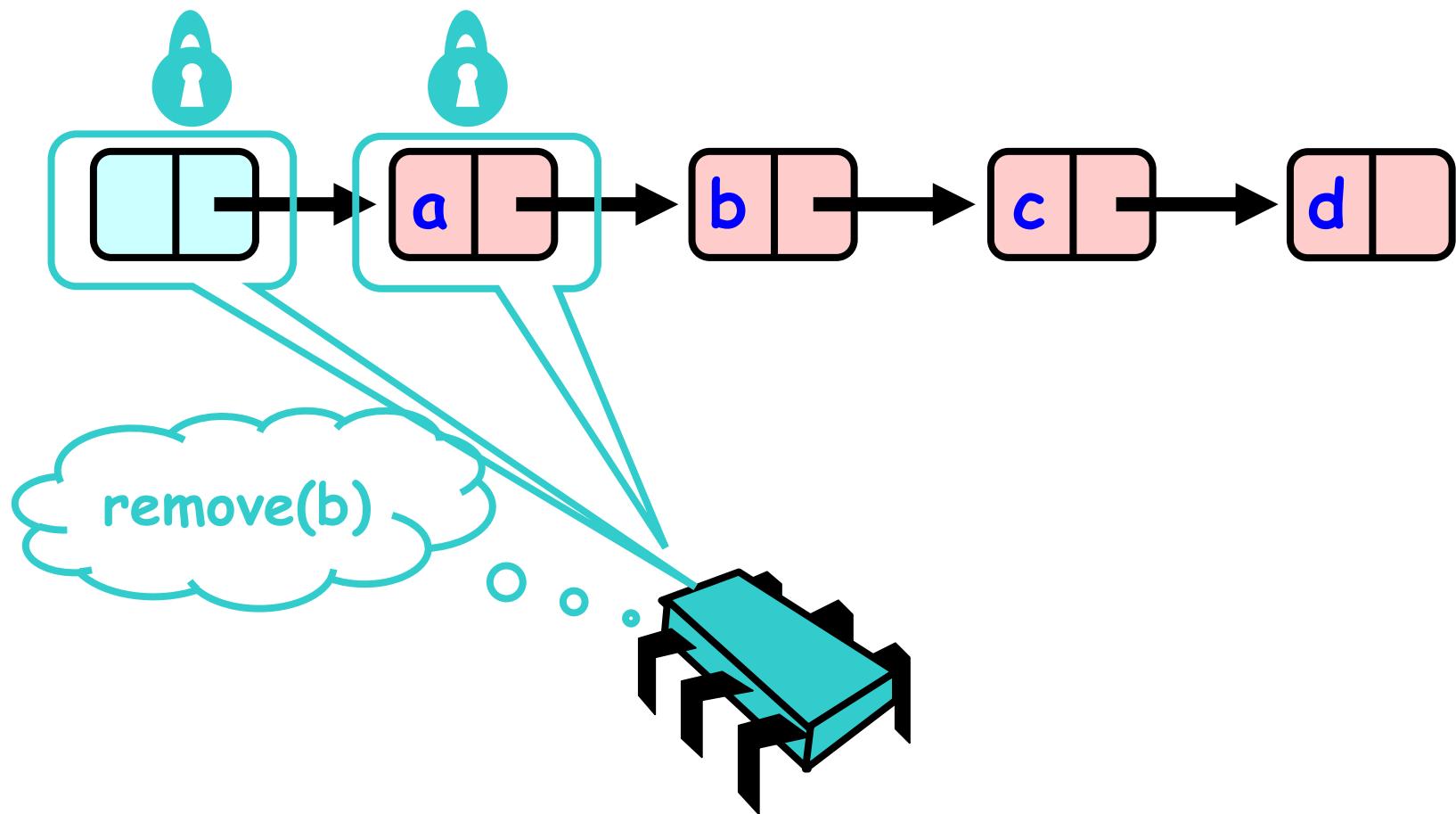
# Removing a Node



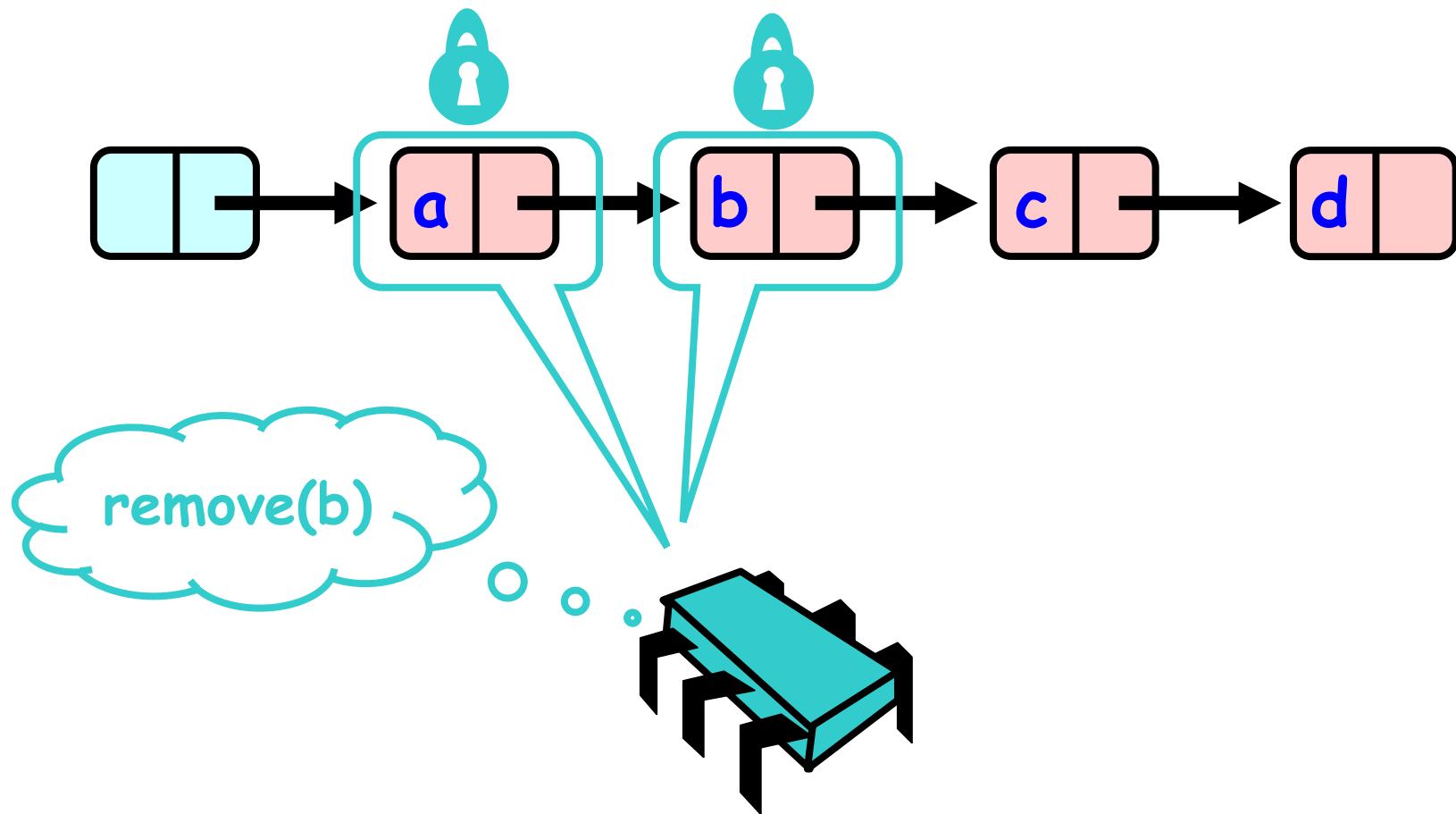
# Removing a Node



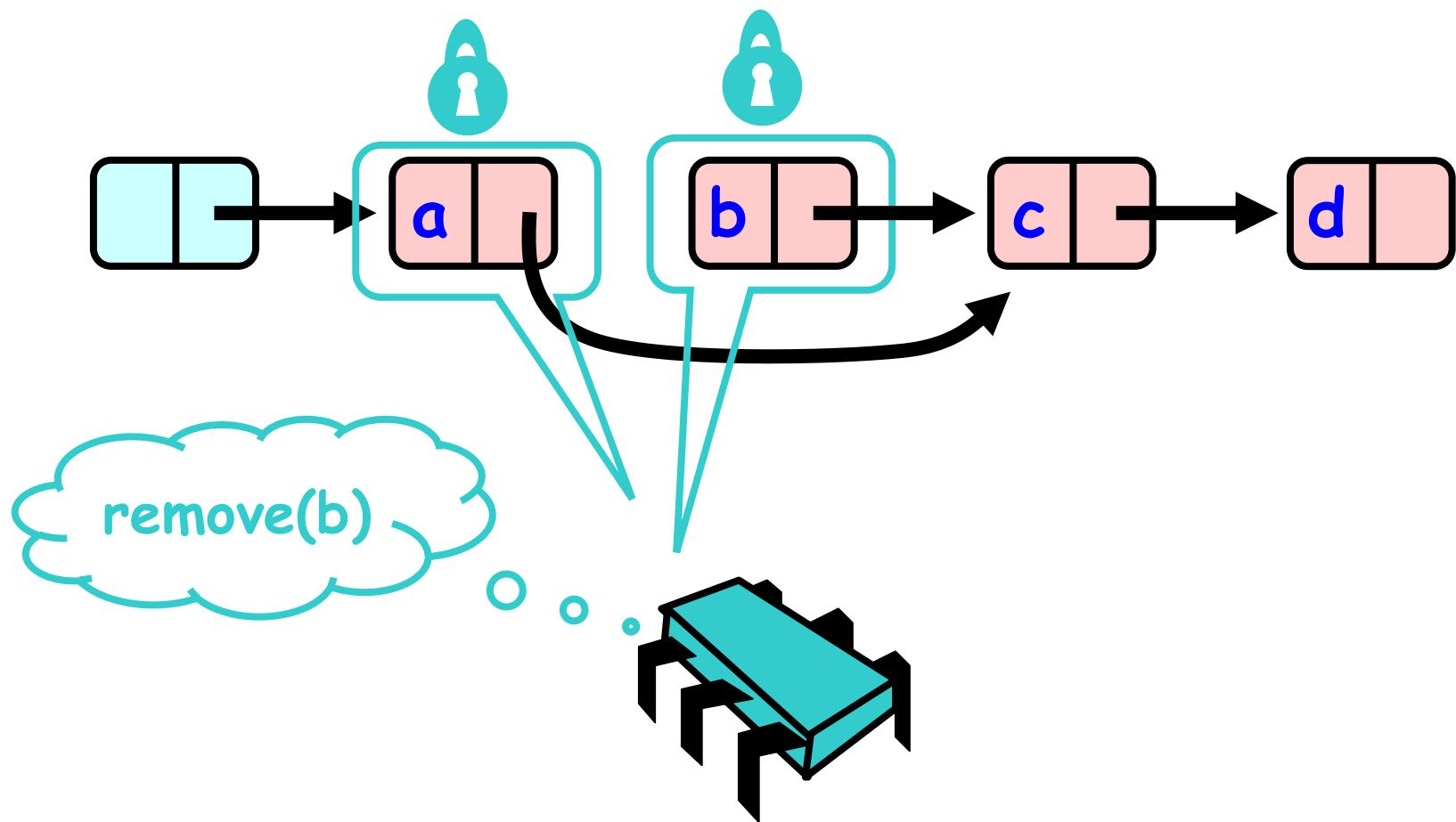
# Removing a Node



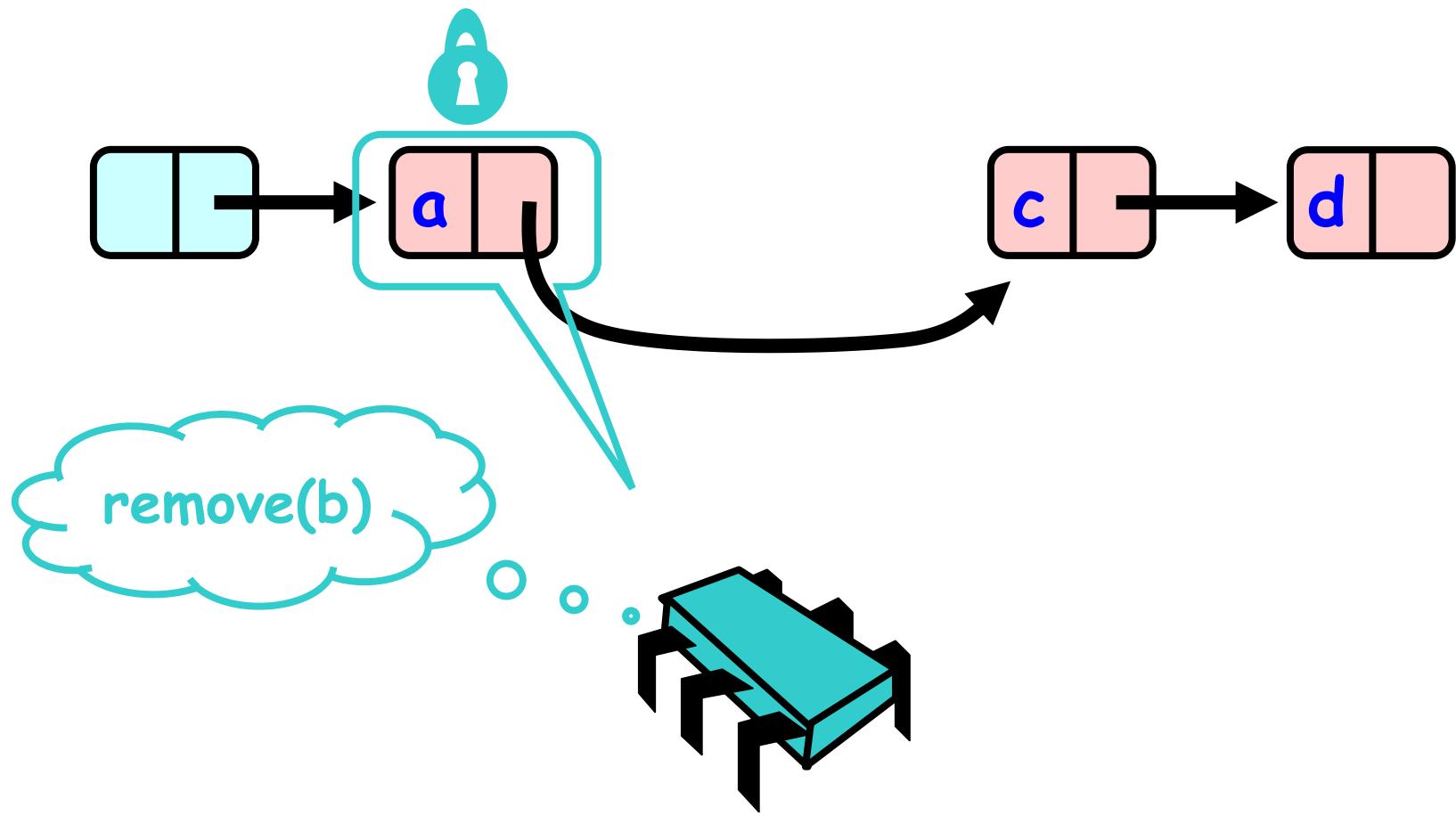
# Removing a Node



# Removing a Node



# Removing a Node

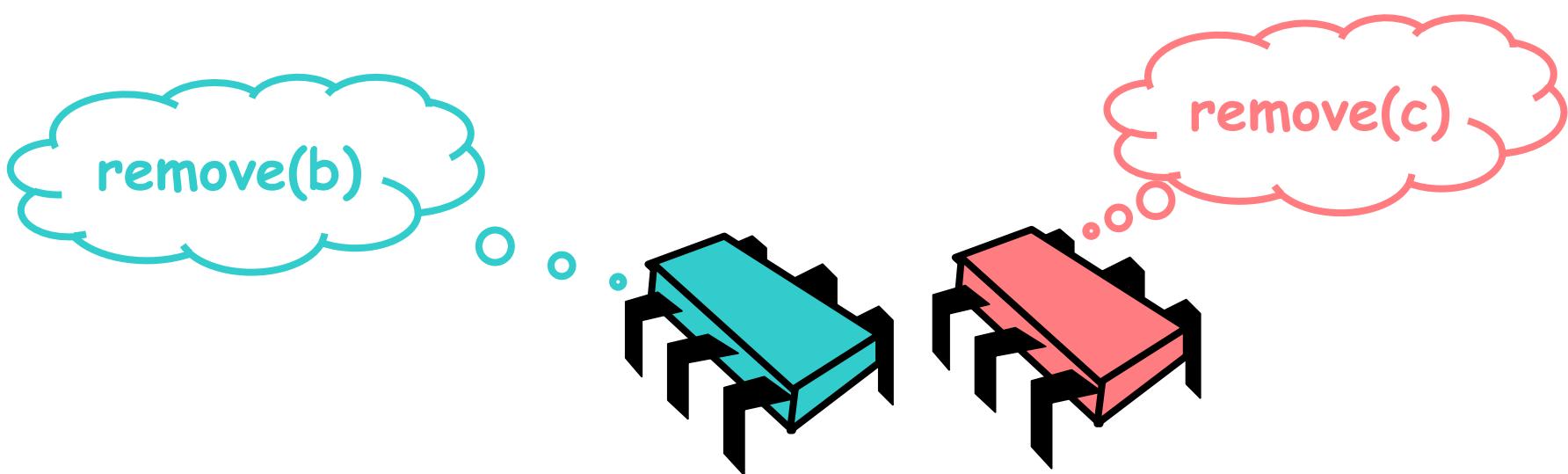
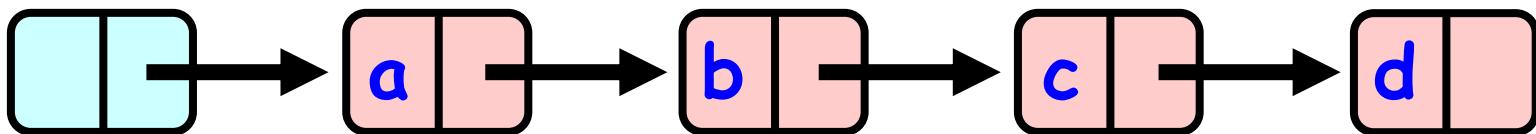




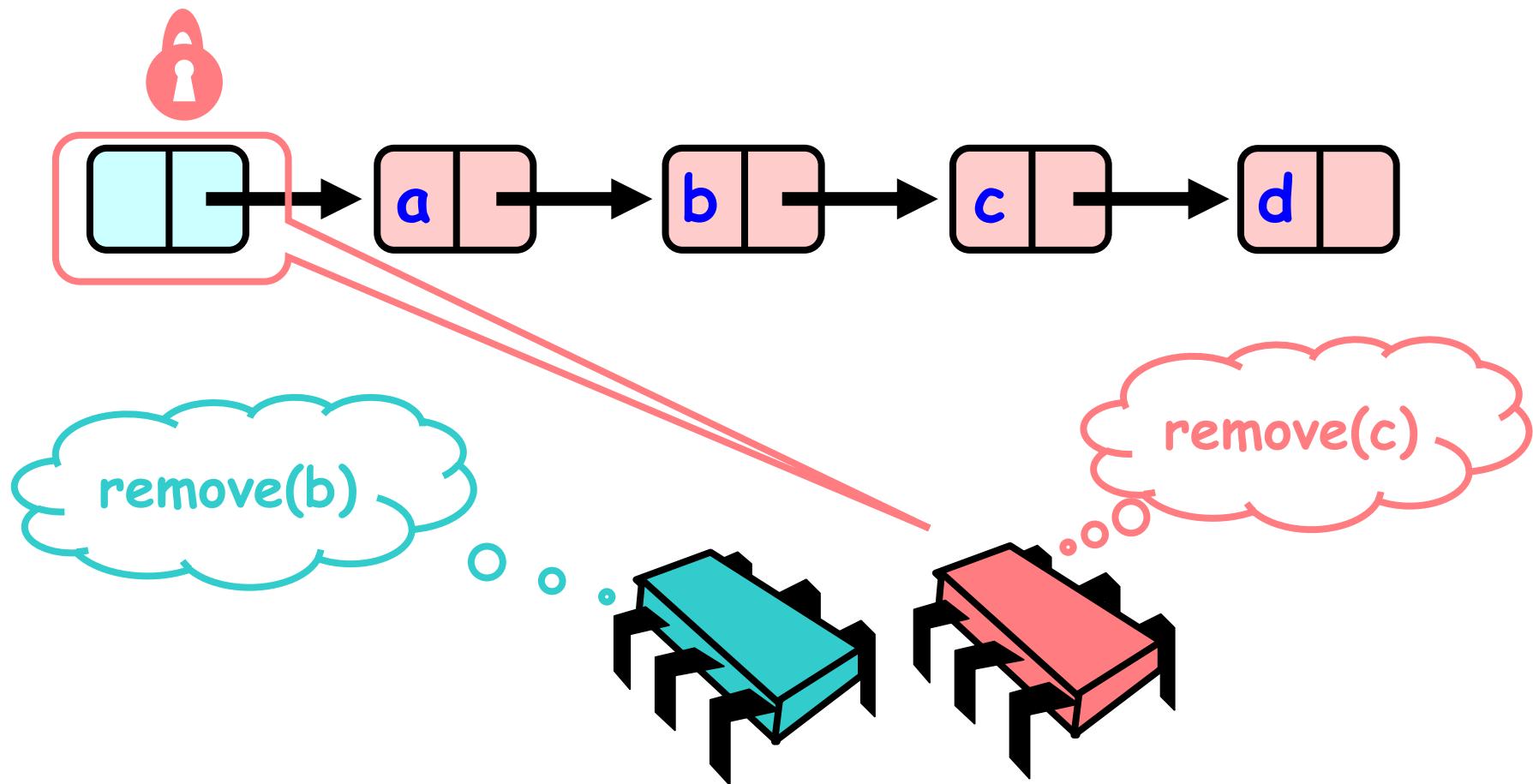
# Hand-over-hand

- Does it solve our problem?

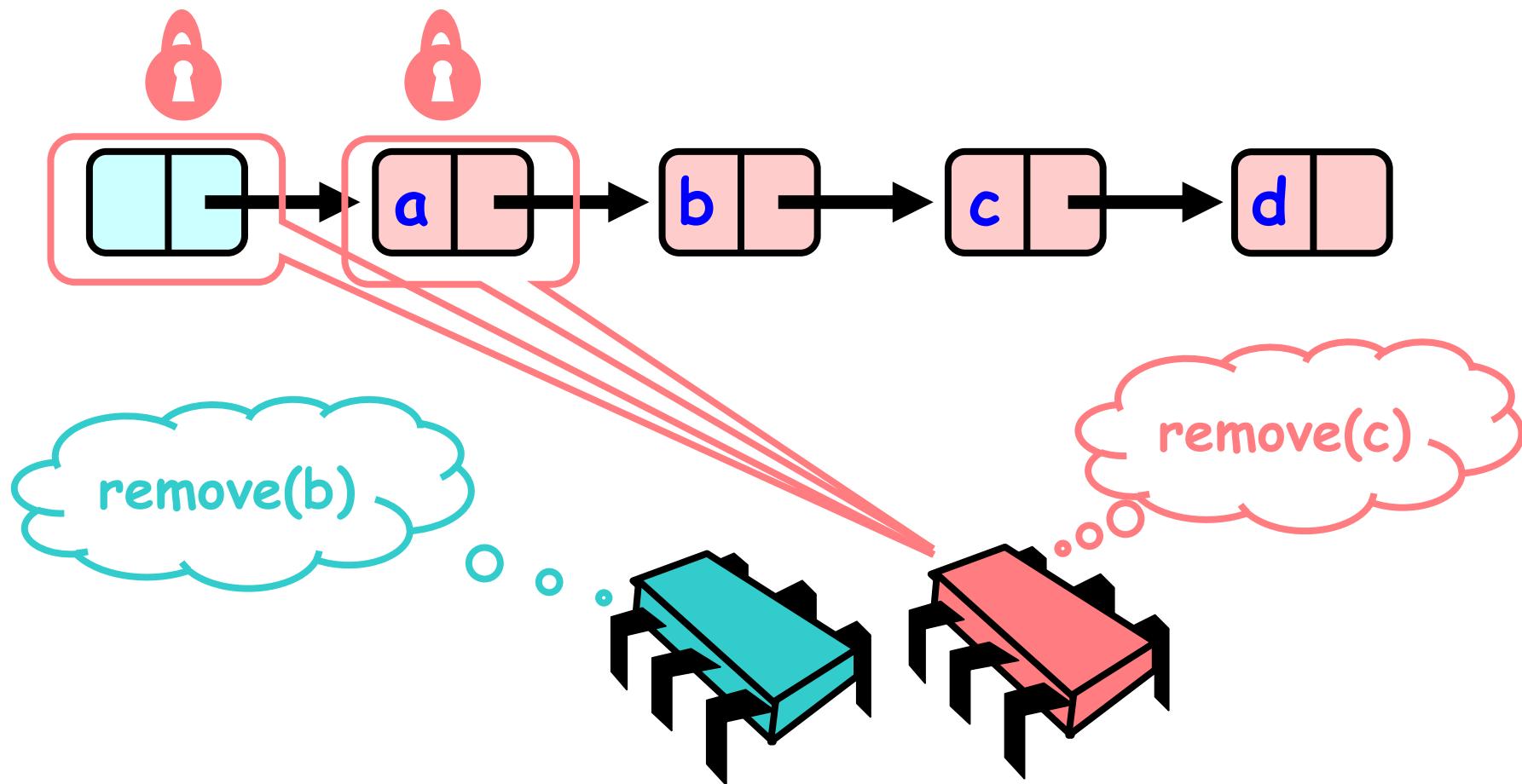
# Removing a Node



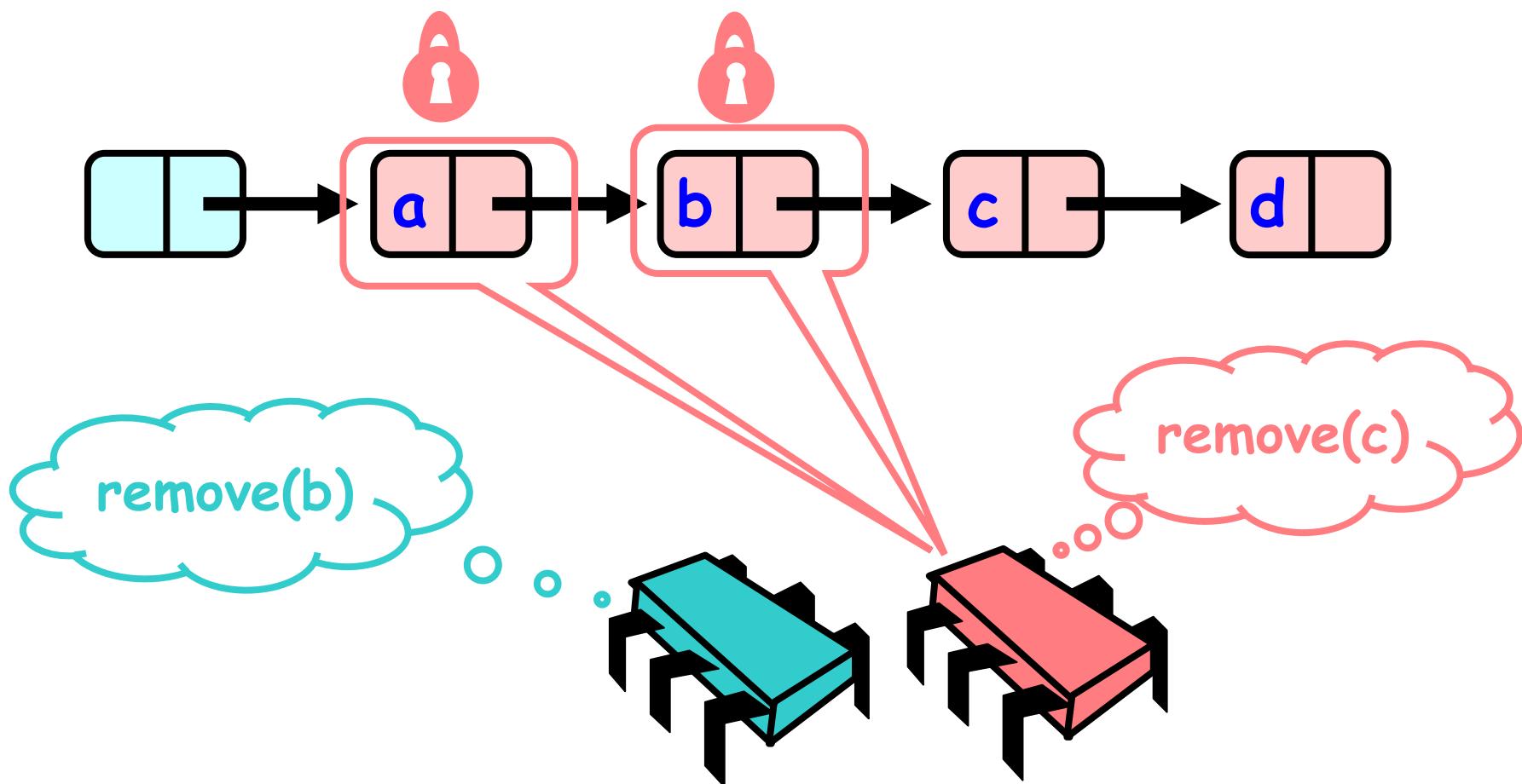
# Removing a Node



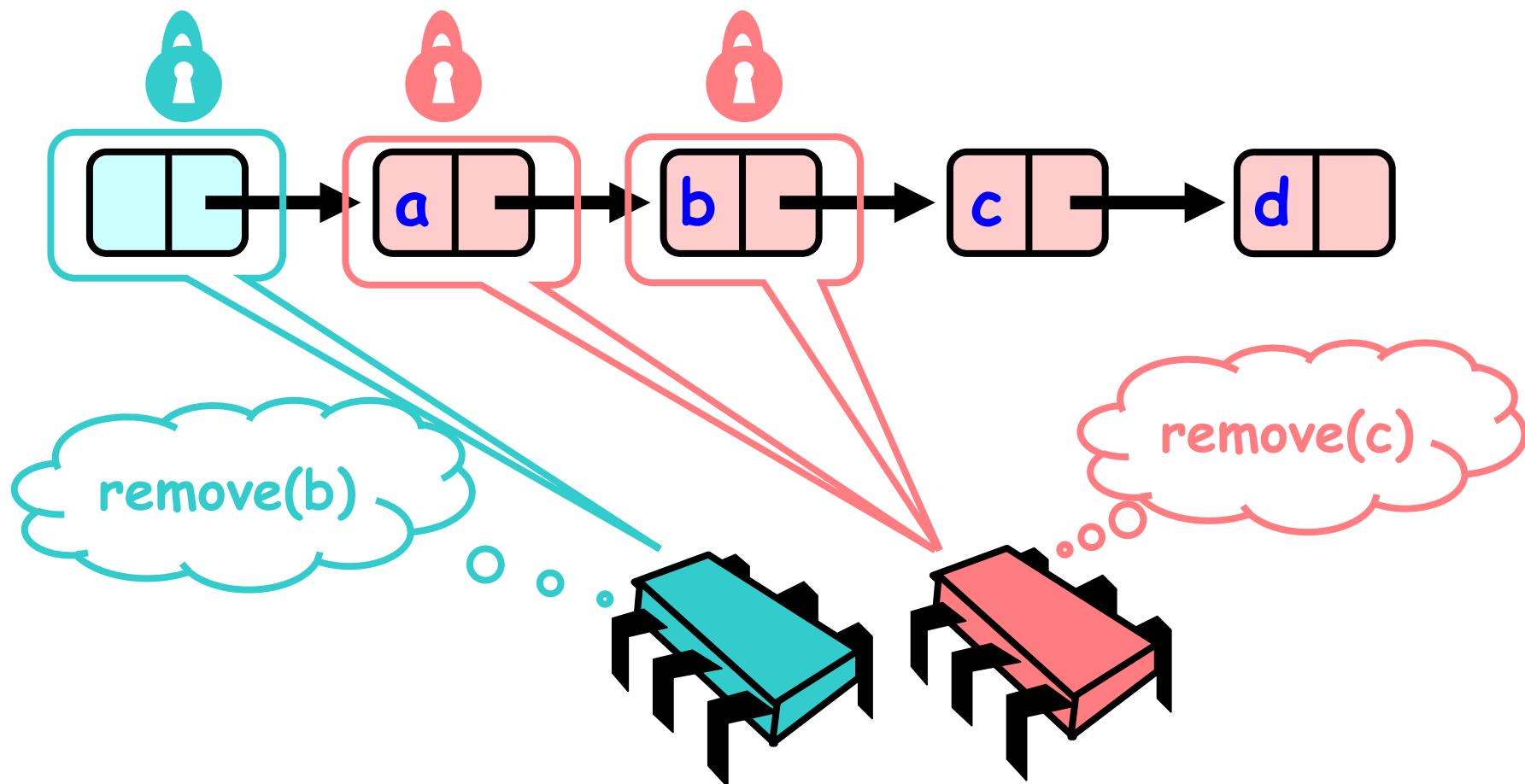
# Removing a Node



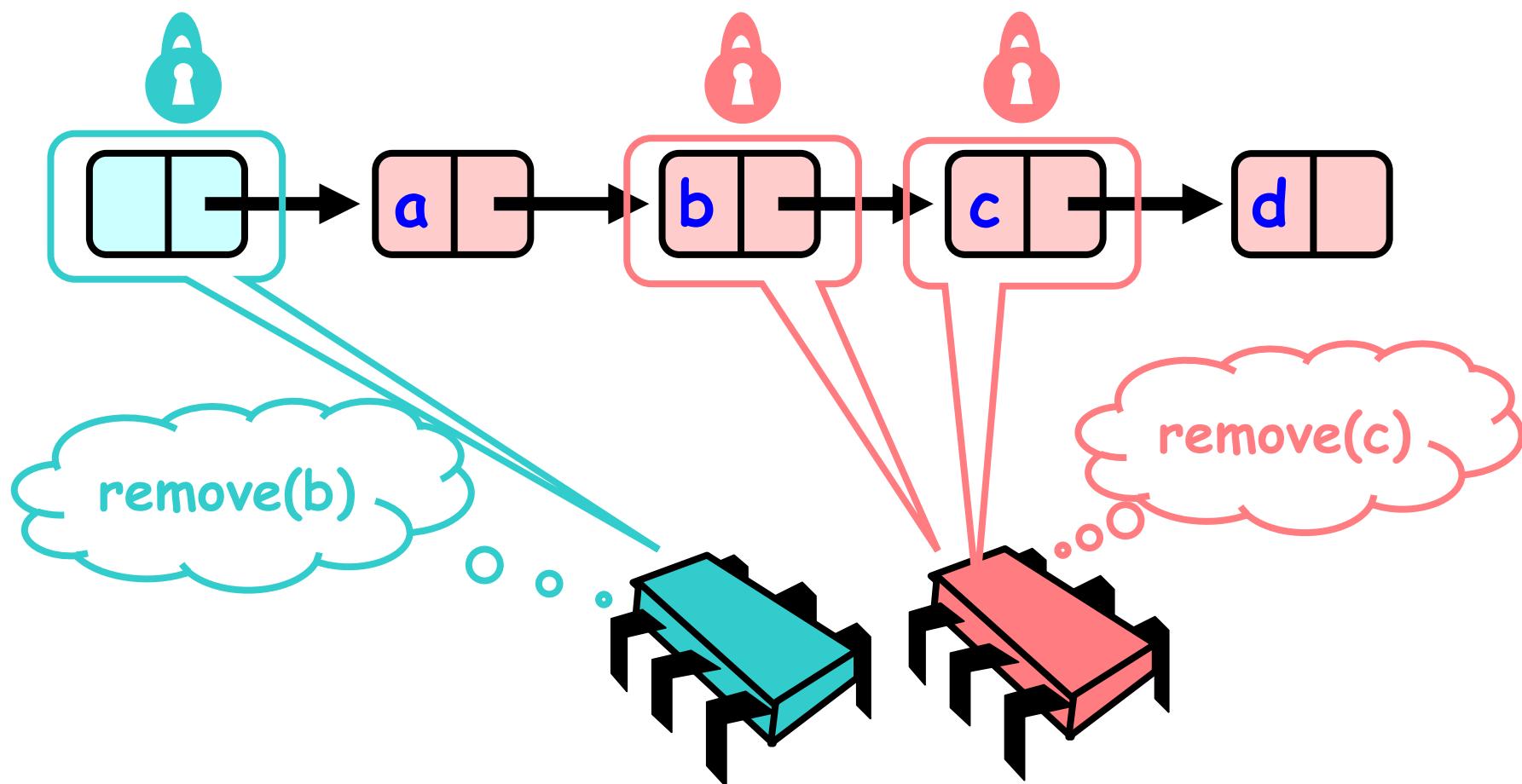
# Removing a Node



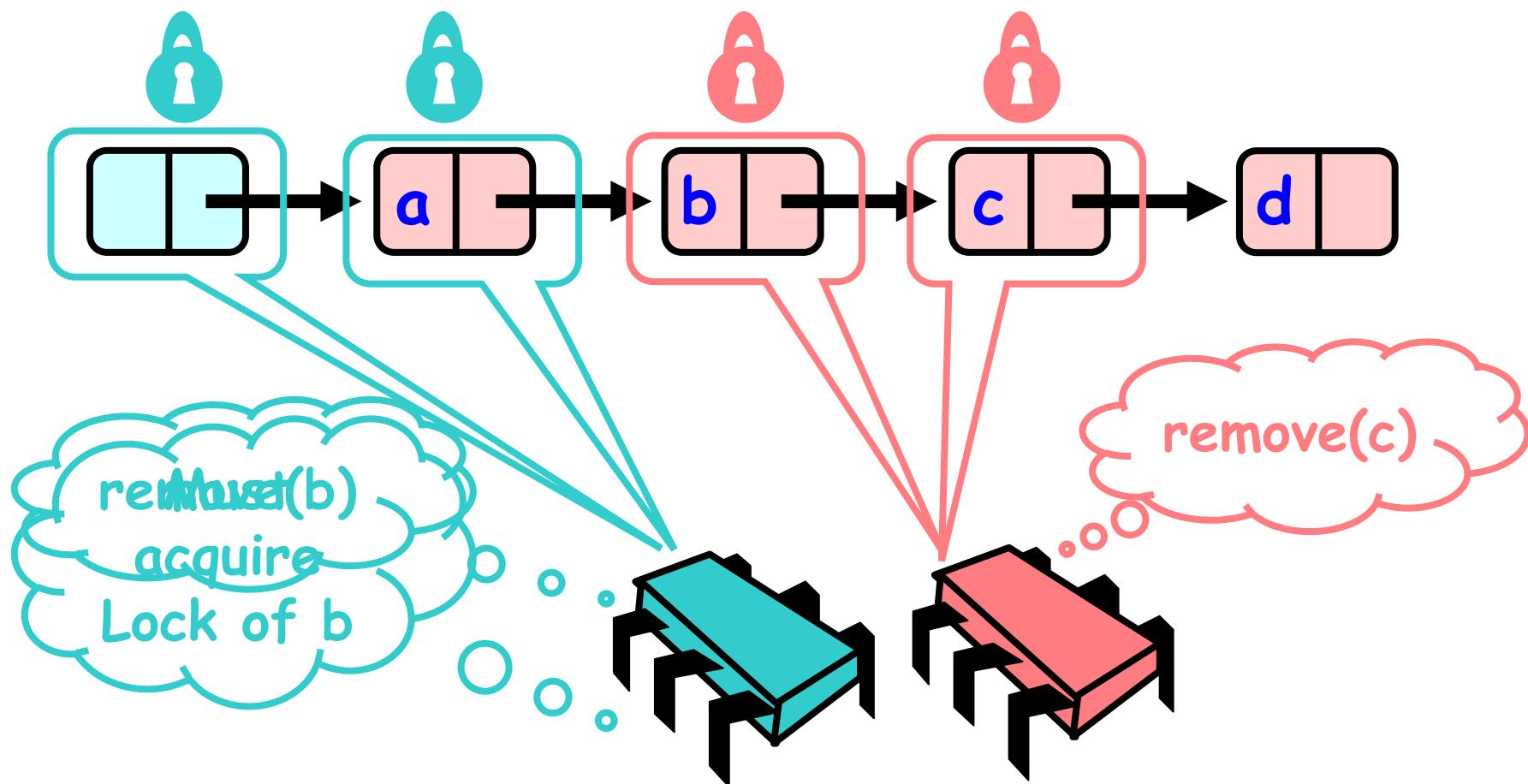
# Removing a Node



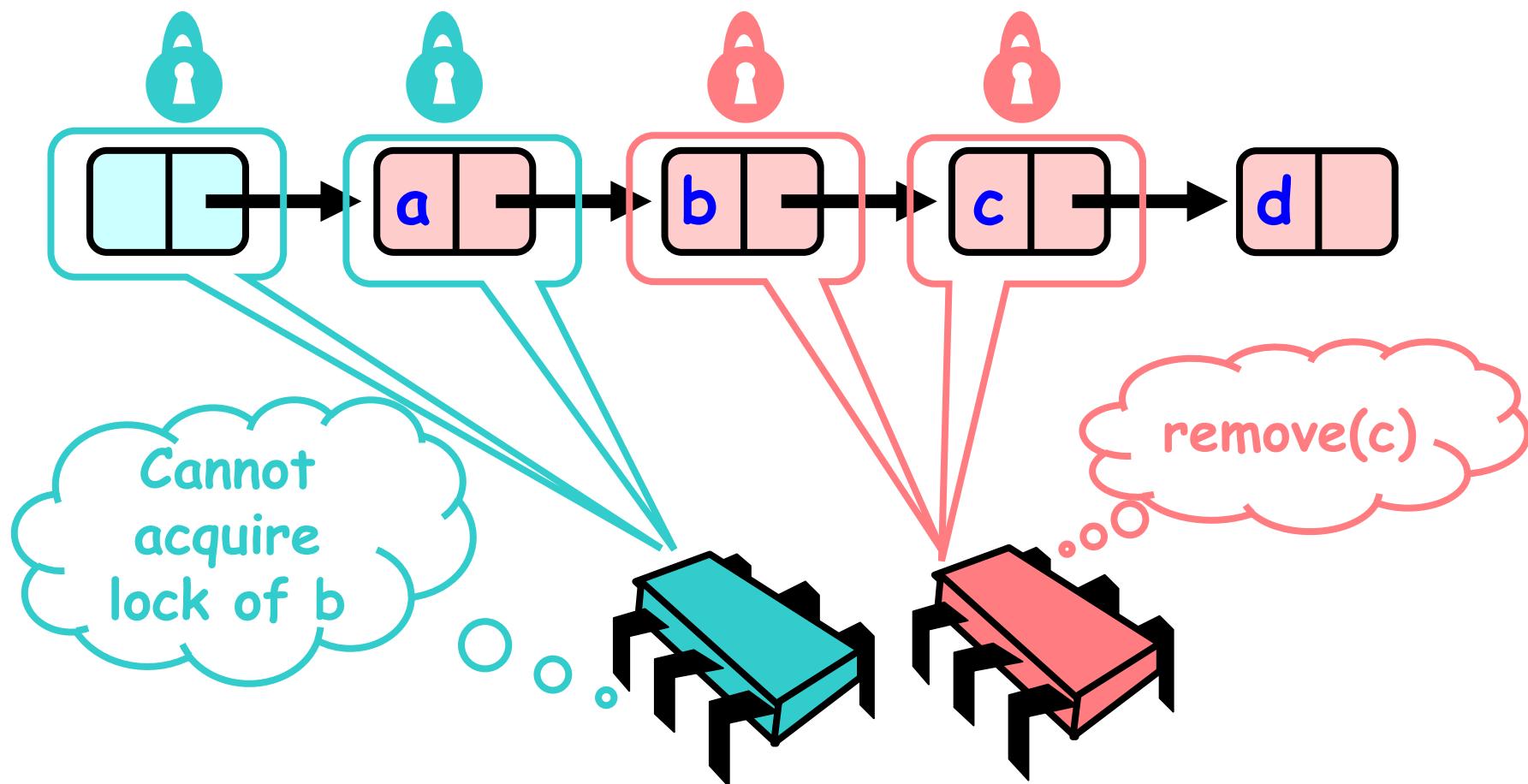
# Removing a Node



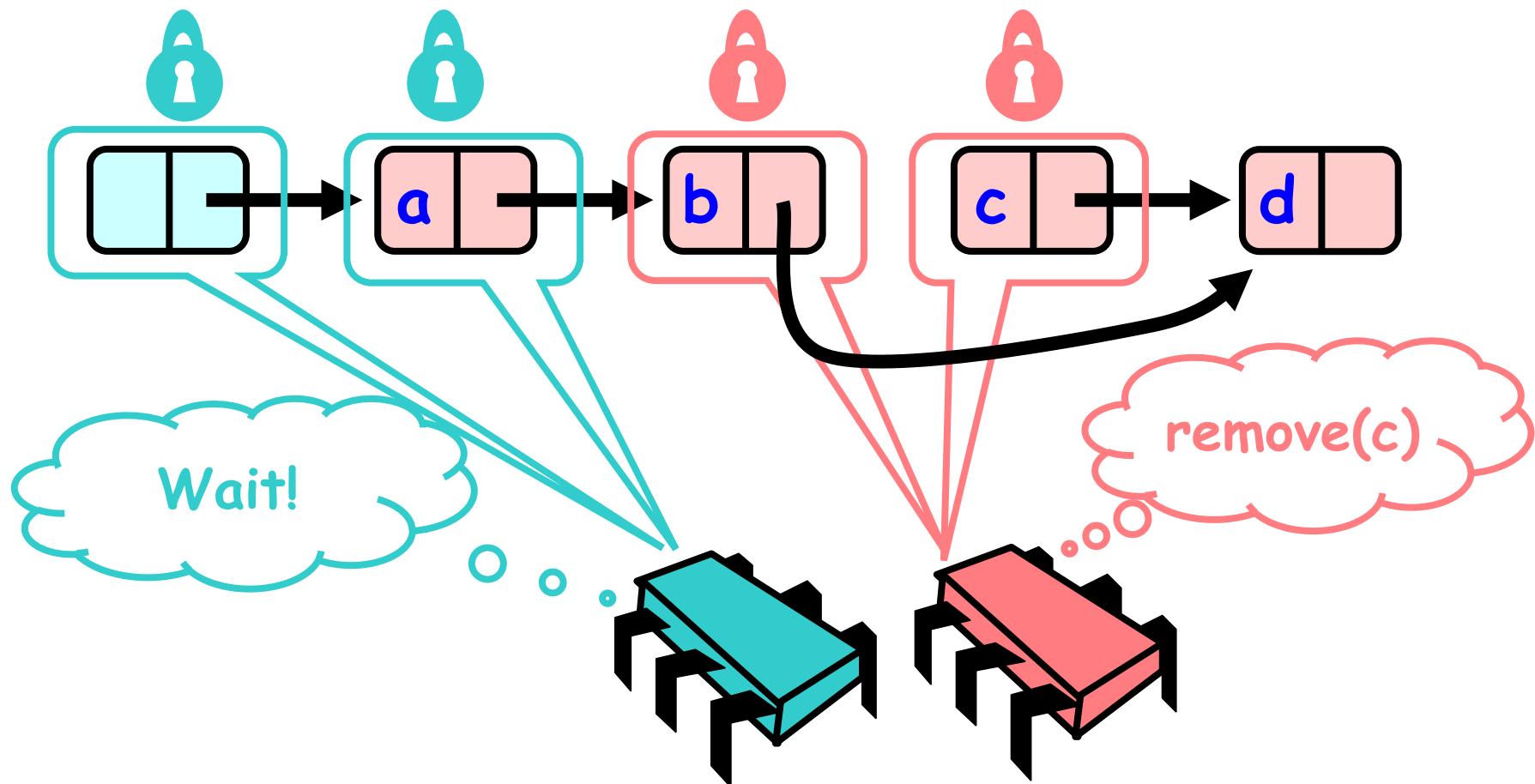
# Removing a Node



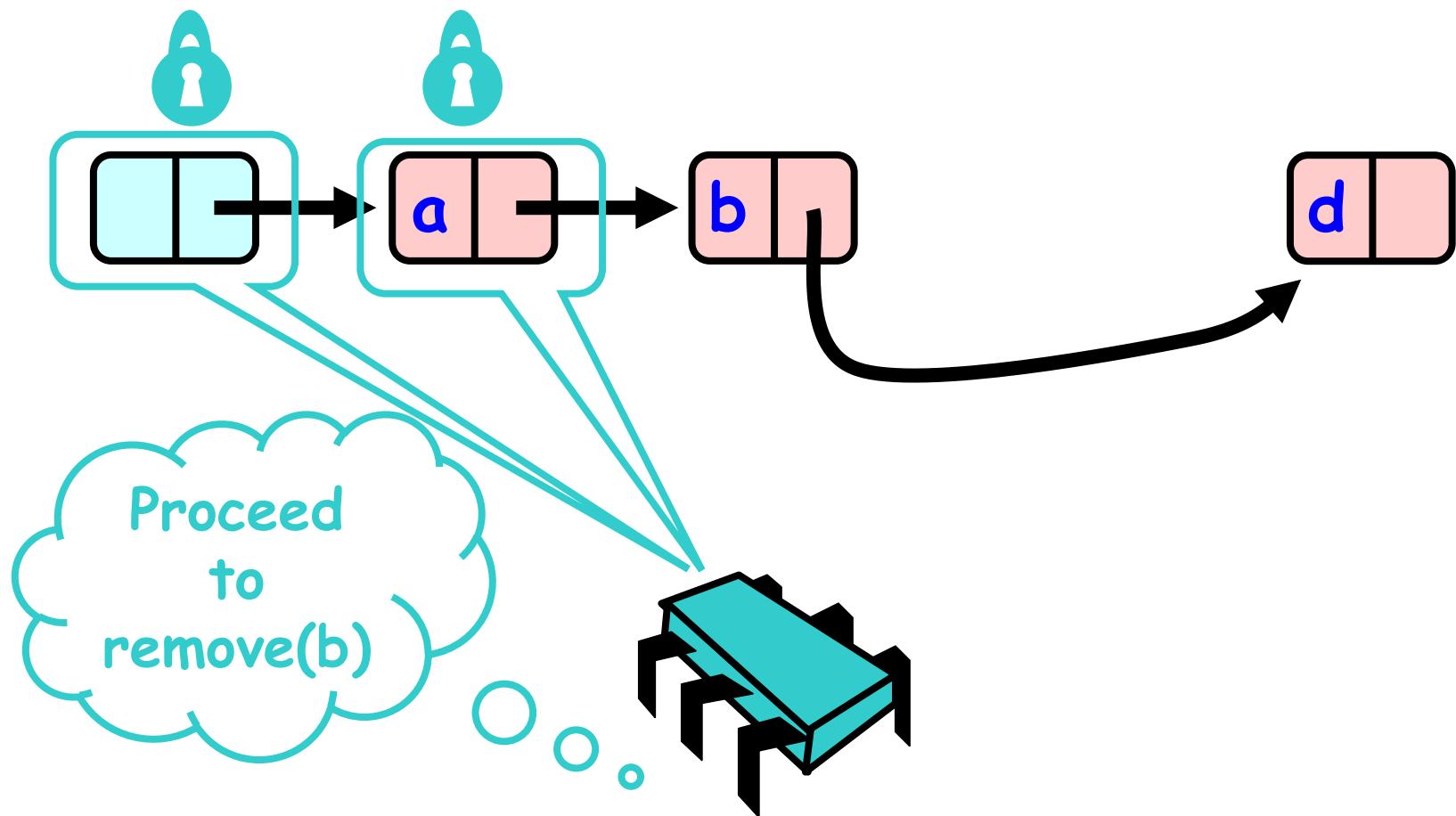
# Removing a Node



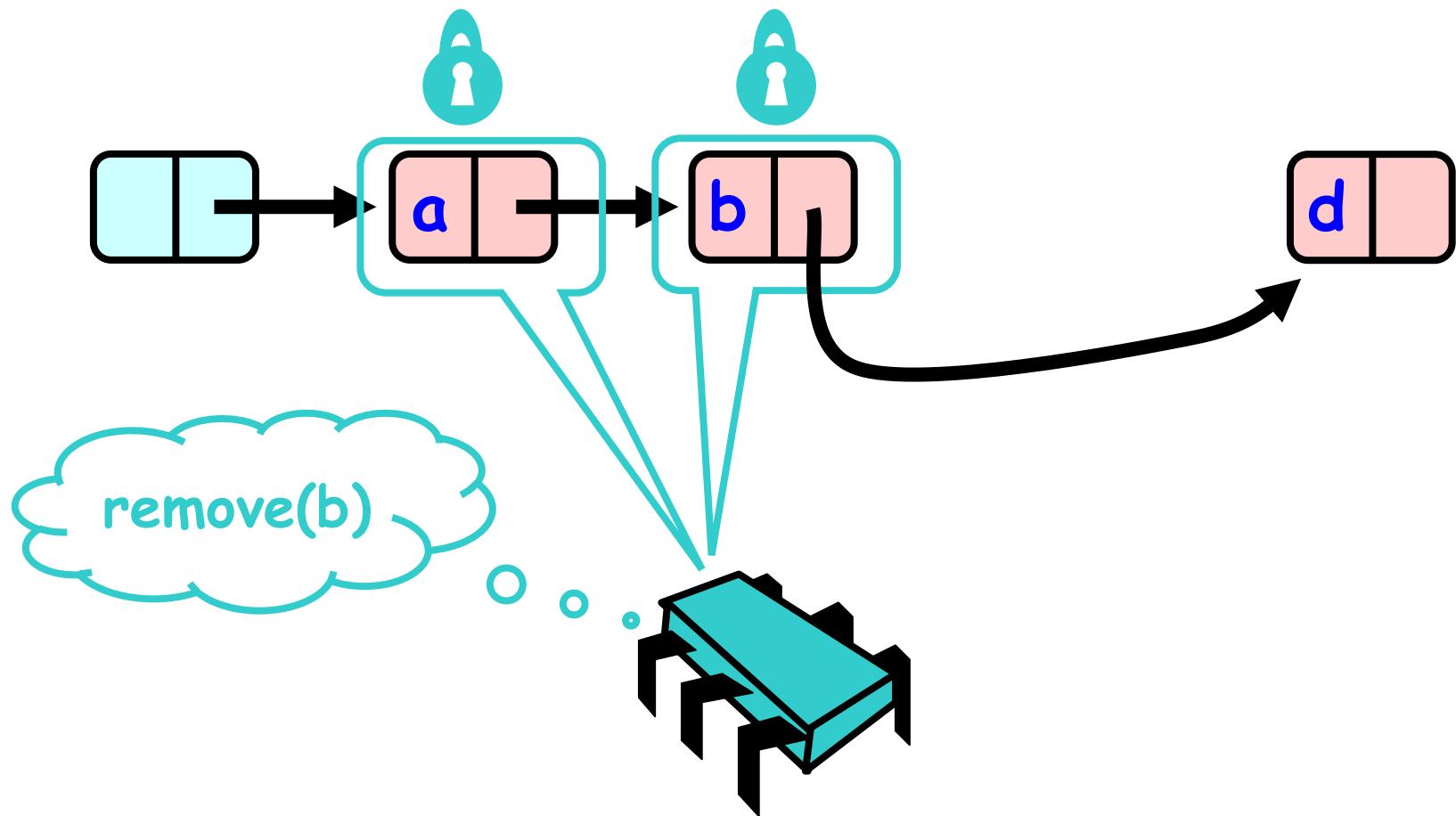
# Removing a Node



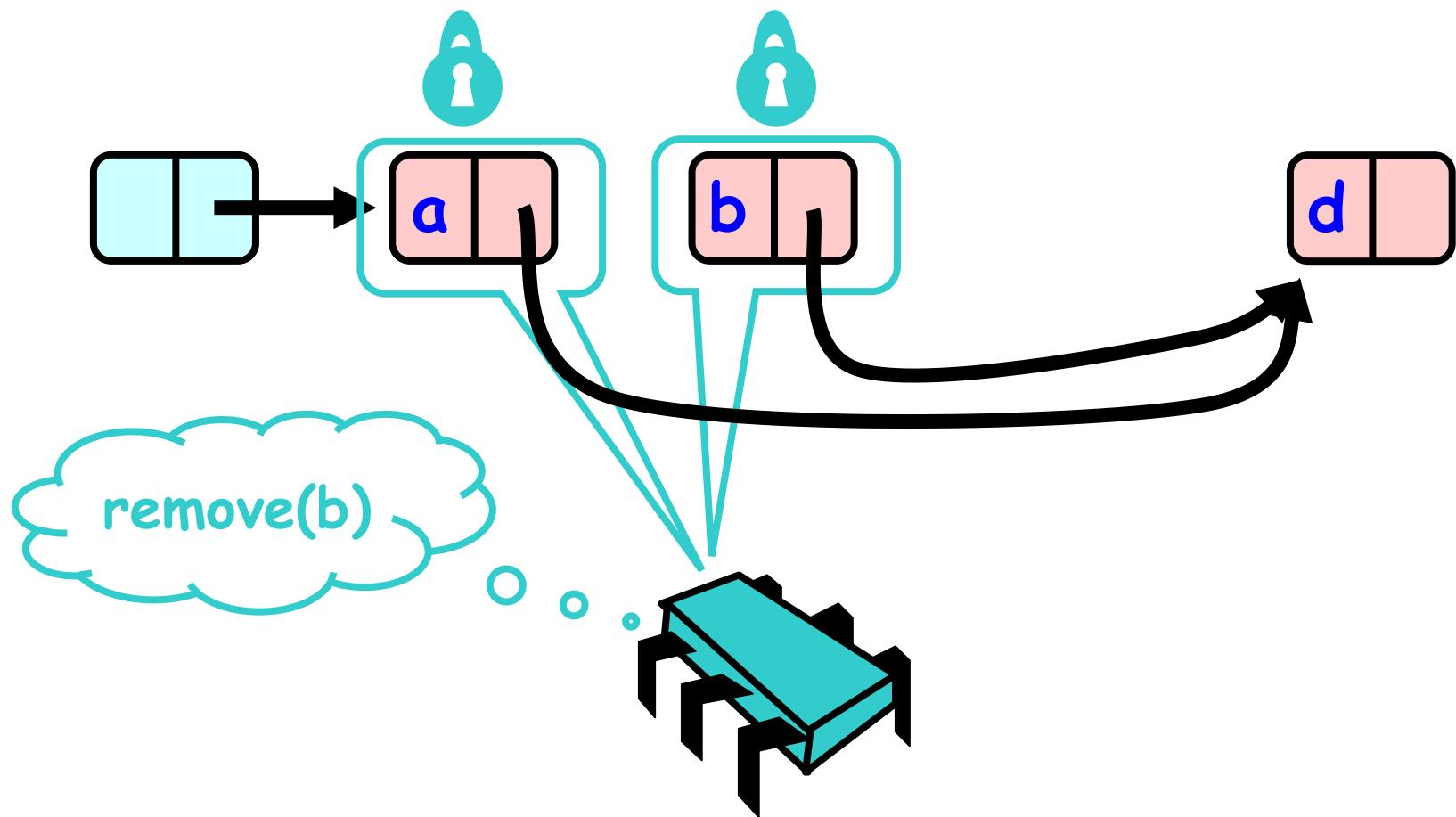
# Removing a Node



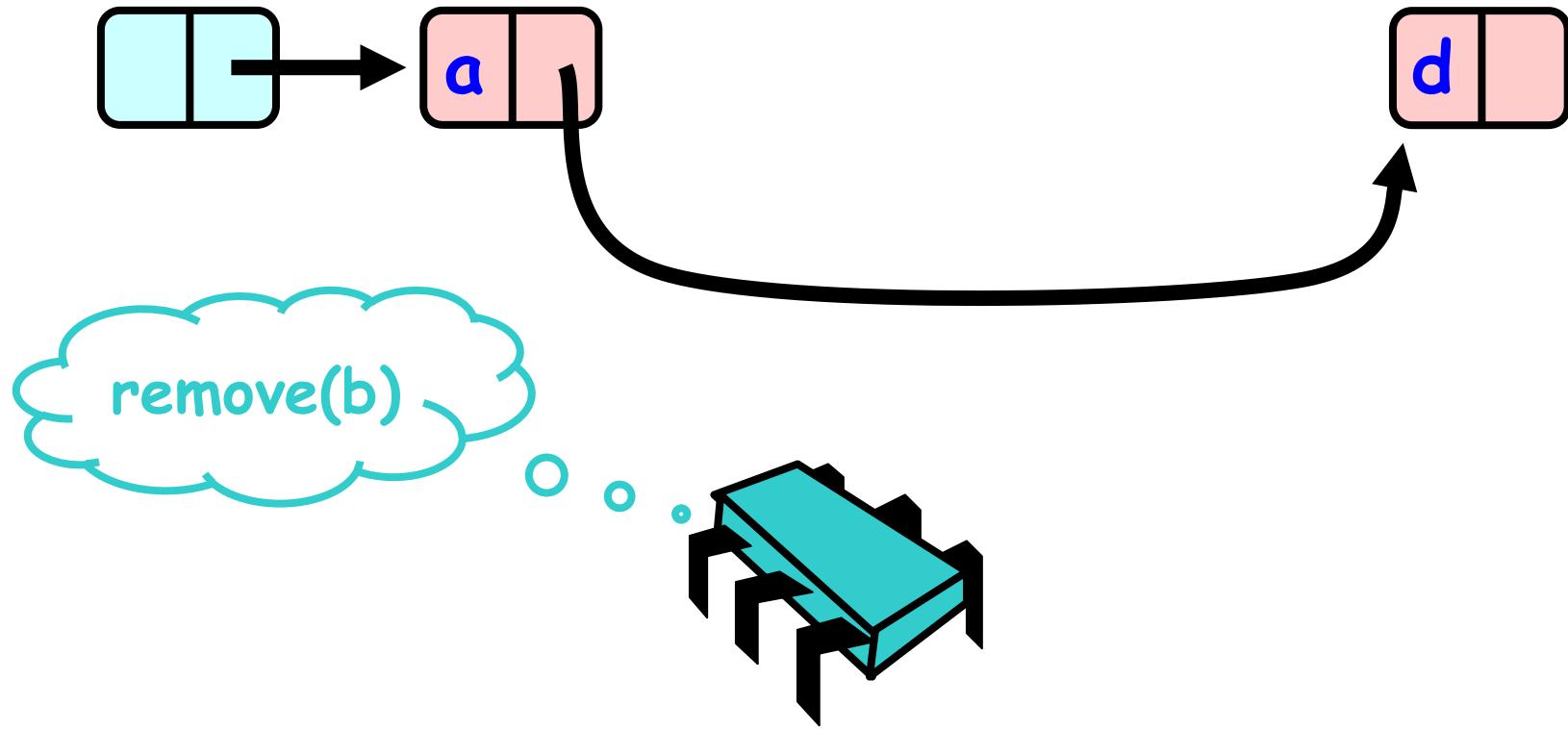
# Removing a Node



# Removing a Node



# Removing a Node



# Removing a Node



# Remove method

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred, curr;  
    try {  
        ...  
    } finally {  
        curr.unlock();  
        pred.unlock();  
    }  
}
```

# Remove method

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred, curr;  
    try {  
        ...  
    } finally {  
        curr.unlock();  
        pred.unlock();  
    }  
}
```

Key used to order node

# Remove method

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred, curr;  
    try {  
        ...  
    } finally {  
        currNode.unlock();  
        predNode.unlock();  
    }  
}
```

Predecessor and current nodes

# Remove method

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred, curr;  
    try {  
        ...  
    } finally {  
        curr.unlock();  
        pred.unlock();  
    }  
}
```

Make sure  
locks released

# Remove method

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred, curr;  
    try {  
        ...  
    } finally {  
        curr.unlock();  
        pred.unlock();  
    }  
}
```

Everything else

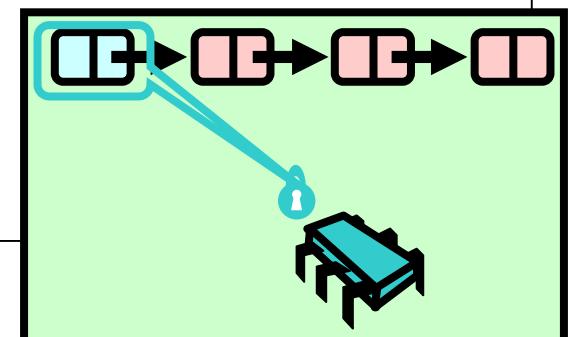
# Remove method

```
try {  
    pred = head;  
    pred.lock();  
    curr = pred.next;  
    curr.lock();  
    ...  
} finally { ... }
```

# Remove method

```
try {  
    pred = head;  
    pred.lock();  
    curr = pred.next;  
    curr.lock();  
    ...  
} finally { ... }
```

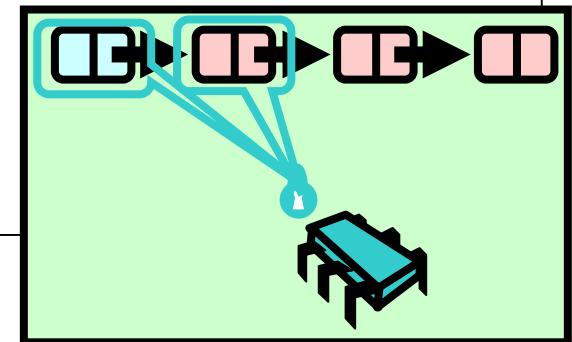
lock pred == head



# Remove method

```
try {  
    pred = head;  
    pred.lock();  
    curr = pred.next;  
    curr.lock();  
    ...  
} finally { ... }
```

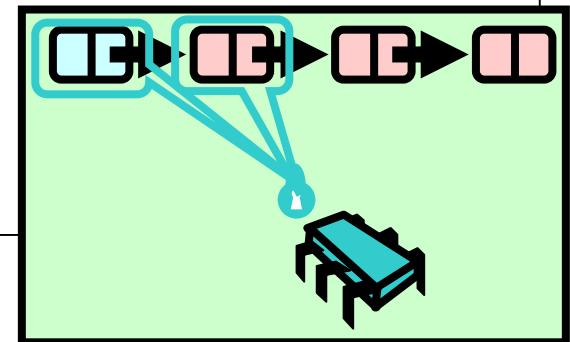
*Lock current*



# Remove method

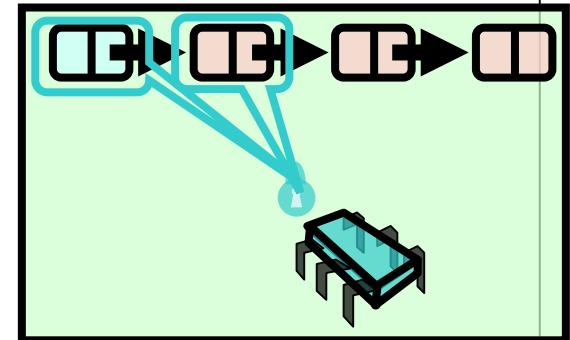
```
try {  
    pred = this.head;  
    pred.lock();  
    curr = pred.next;  
    curr.lock();  
    ...  
} finally { ... }
```

**Traversing list**



# Remove: searching

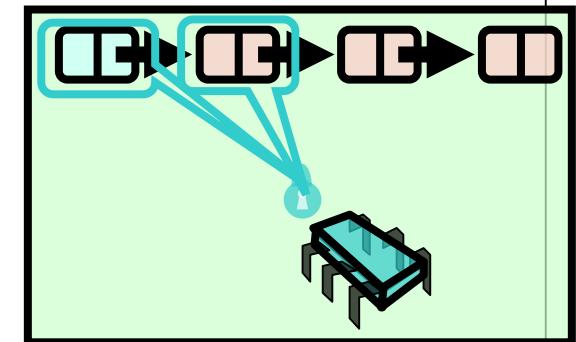
```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        return true;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}  
return false;
```



# Remove: searching

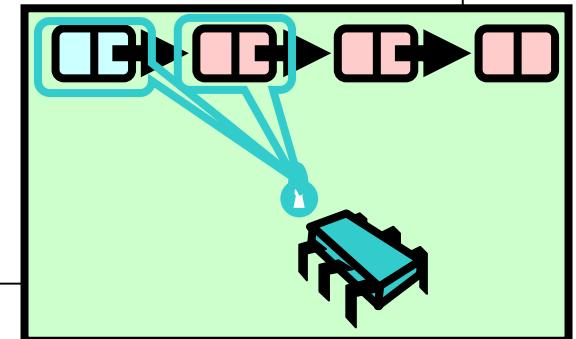
```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        return true;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}  
return false;
```

Search key range



# Remove: searching

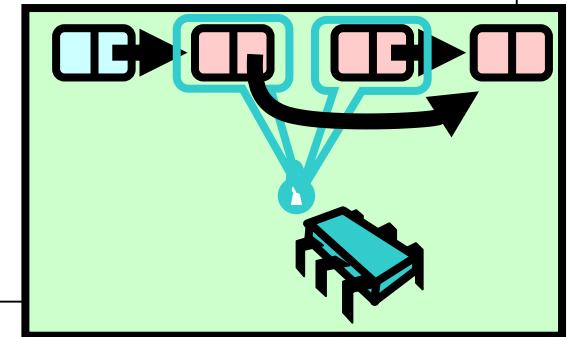
```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        return true;  
    }  
    pred.unlock(); At start of each loop:  
    pred = curr;      curr and pred locked  
    curr = curr.next;  
    curr.lock();  
}  
return false;
```



# Remove: searching

```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        return true;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}  
return false;
```

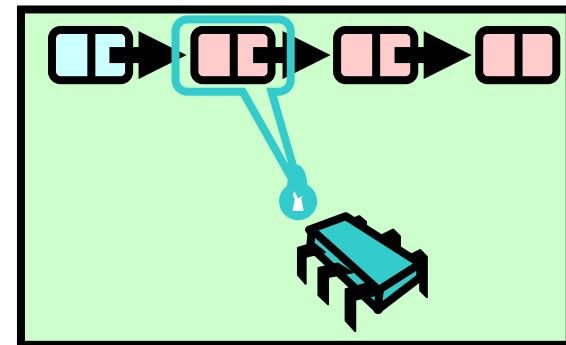
If item found, remove node



# Remove: searching

Unlock predecessor

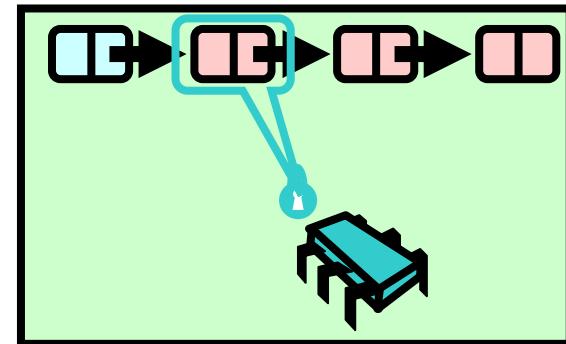
```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        return true;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}  
return false;
```



# Remove: searching

```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        return true;  
    }  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}  
return false;
```

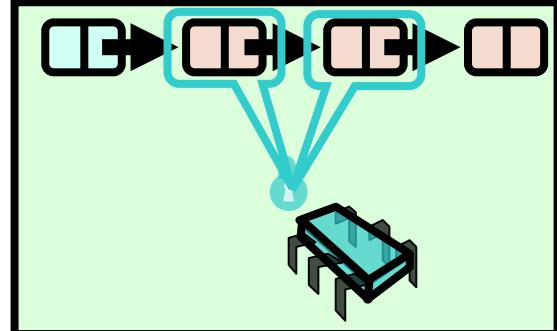
**Promote predecessor**



# Remove: searching

```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        return true;  
    }  
    pred.unlock();  
    pred = currNode;  
    curr = curr.next;  
    curr.lock();  
}  
  
return false;
```

**Find and lock new current**



# Remove: searching

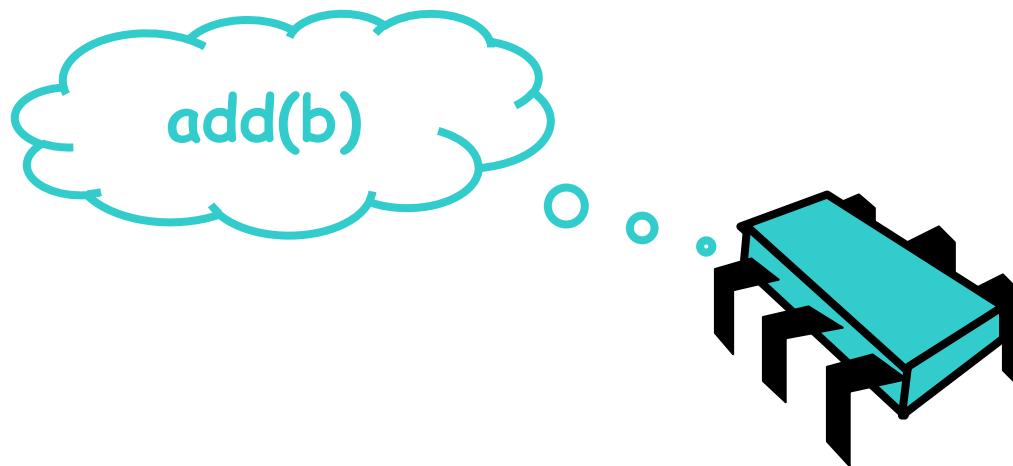
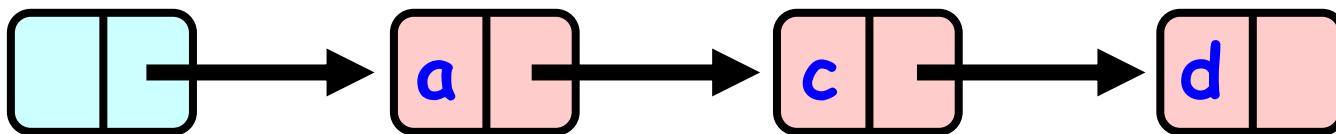
```
while (curr.key <= key) {  
    if (item == curr.item) {  
        pred.next = curr.next;  
        return true;  
    } Otherwise, not present  
    pred.unlock();  
    pred = curr;  
    curr = curr.next;  
    curr.lock();  
}  
  
return false;
```



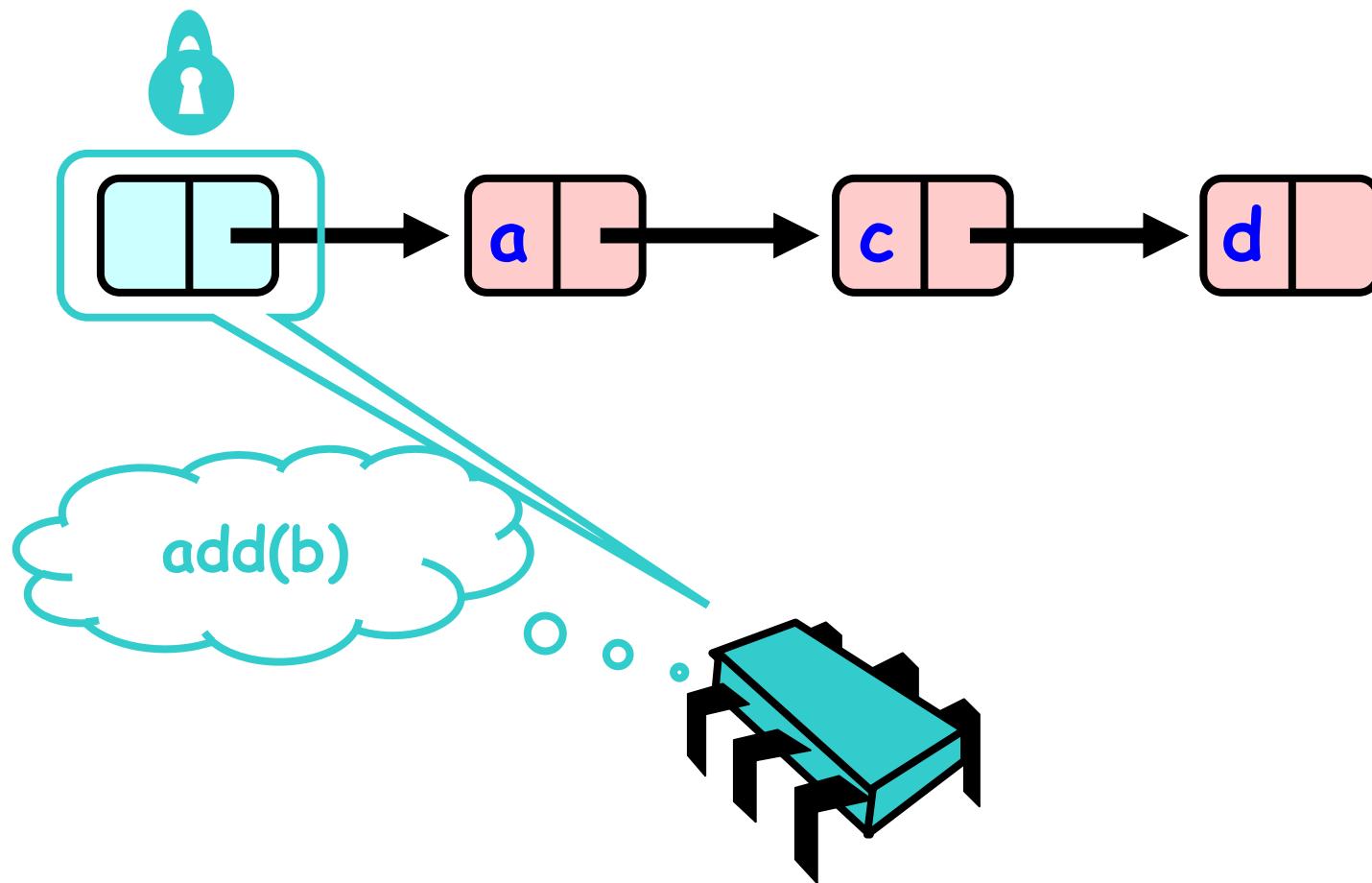
# Why does this work?

- To remove node e
  - Must lock e
  - Must lock e's predecessor
- Therefore, if you lock a node
  - It can't be removed
  - And neither can its successor

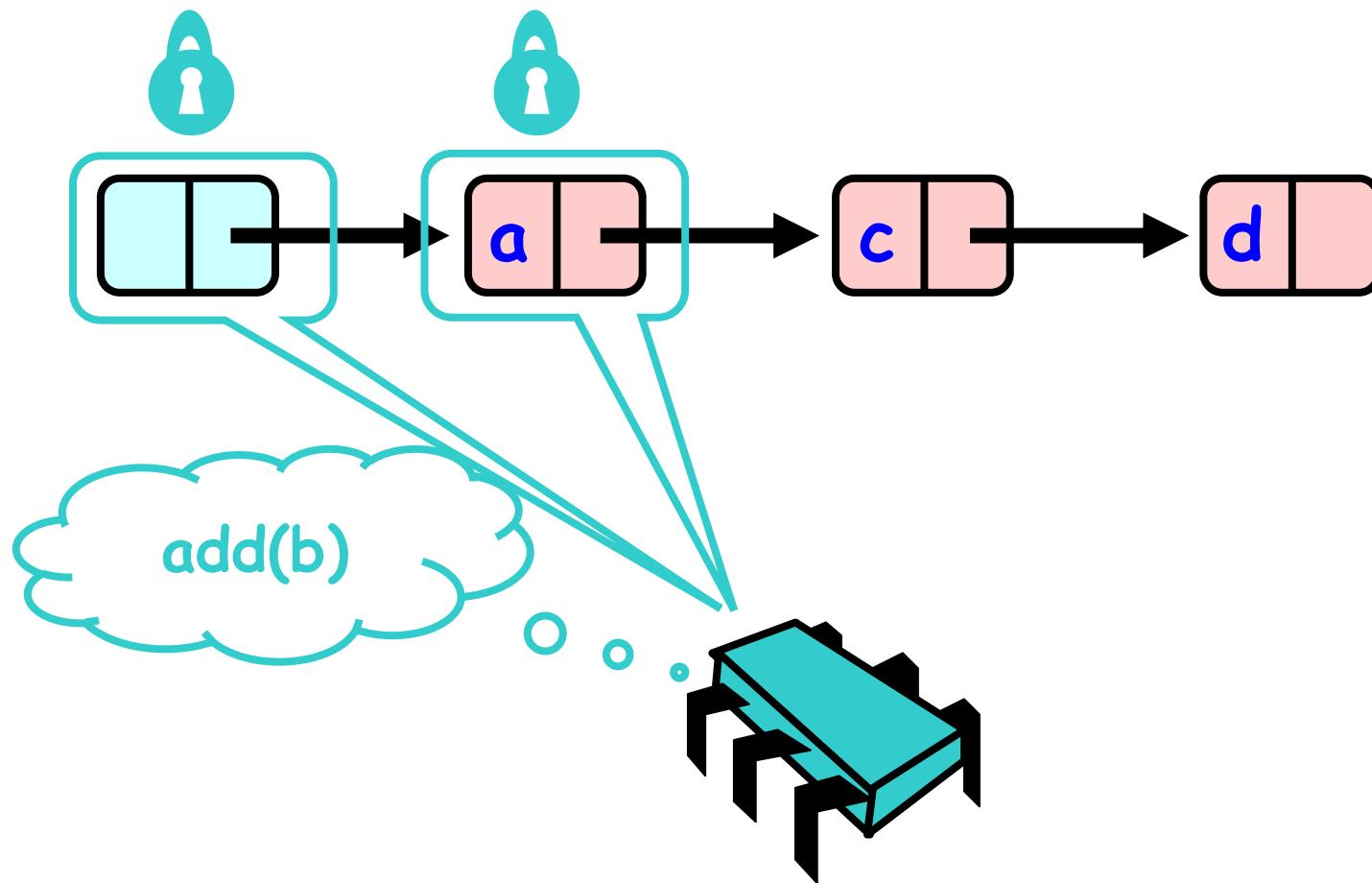
# Adding a node



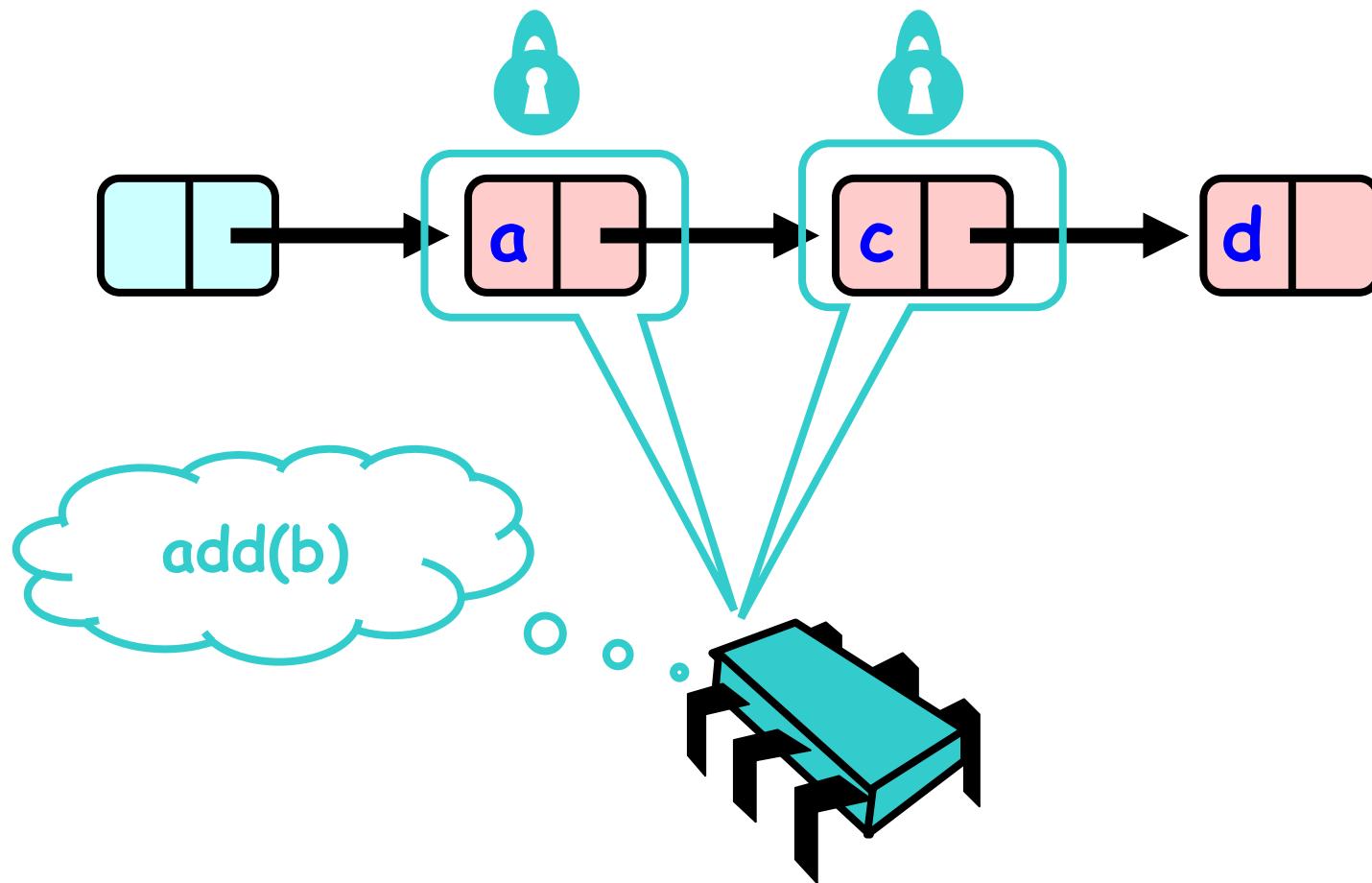
# Adding a node



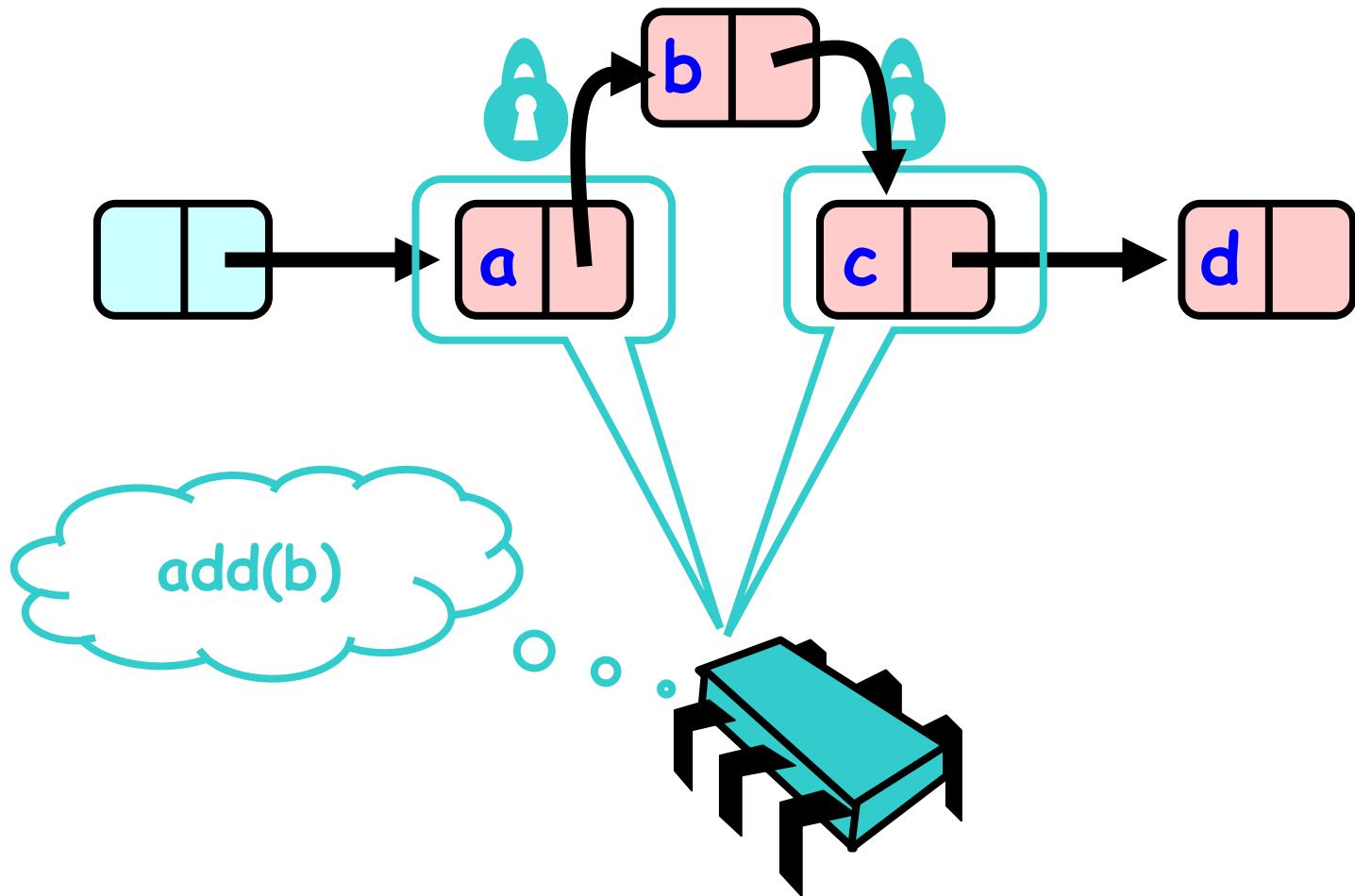
# Adding a node



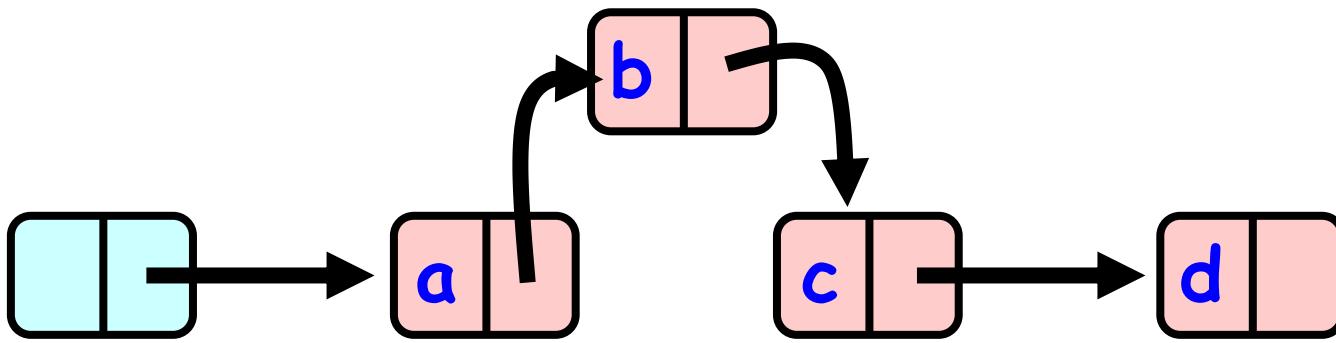
# Adding a node

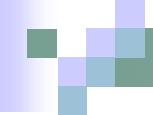


# Adding a node



# Adding a node

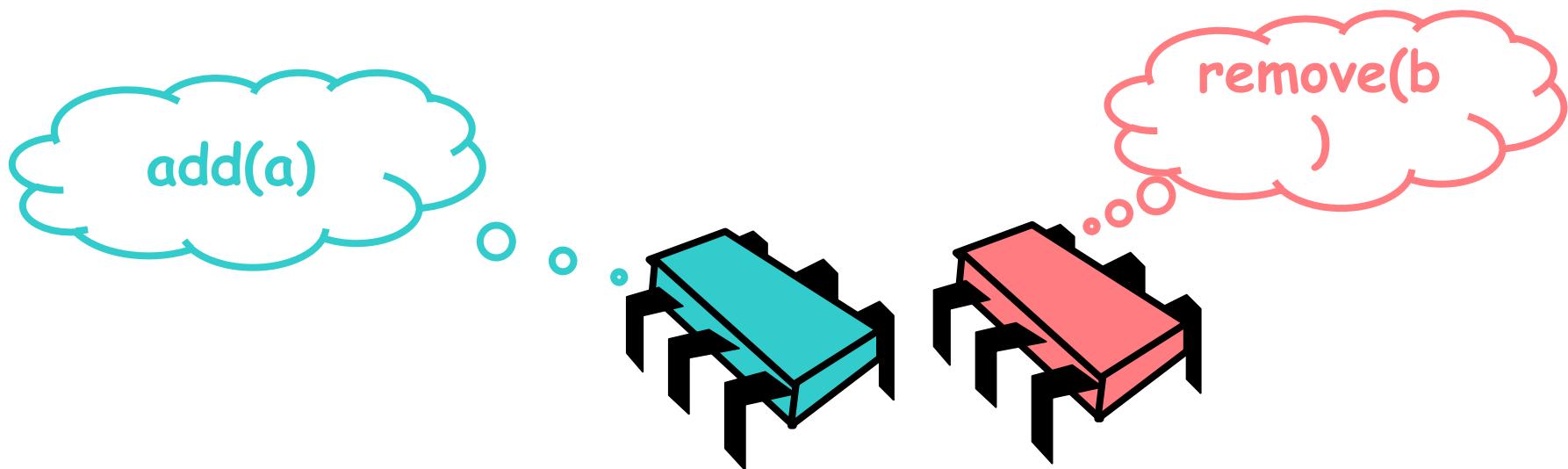
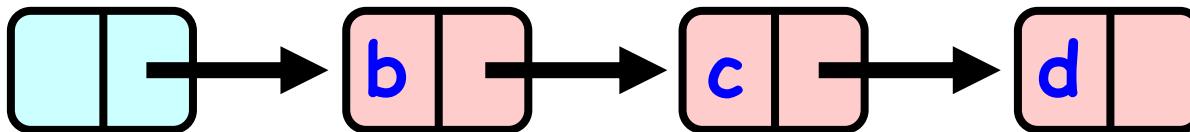




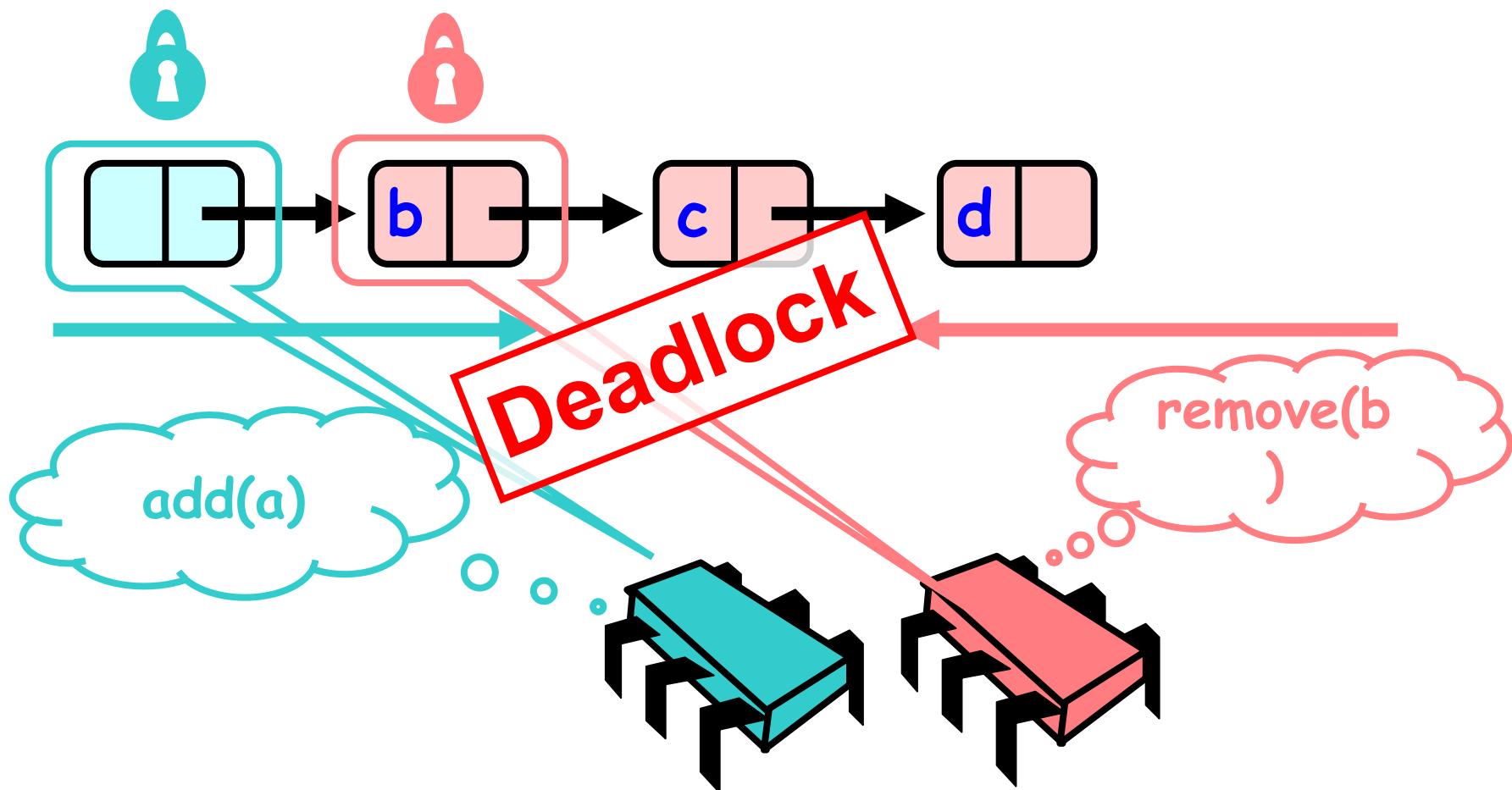
# Hand-over-hand locking

- Does it matter whether threads acquire locks in the same order?
- What happens when it does not happen in the same order?

# Hand-over-hand locking



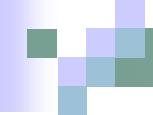
# Hand-over-hand locking





# Fine-grained synchronization

- Although fine-grained synchronization is an improvement over coarse-grained synchronization it is still a potentially long sequence of locks acquisitions and releases
- The algorithm is blocking



# Fine-grained synchronization

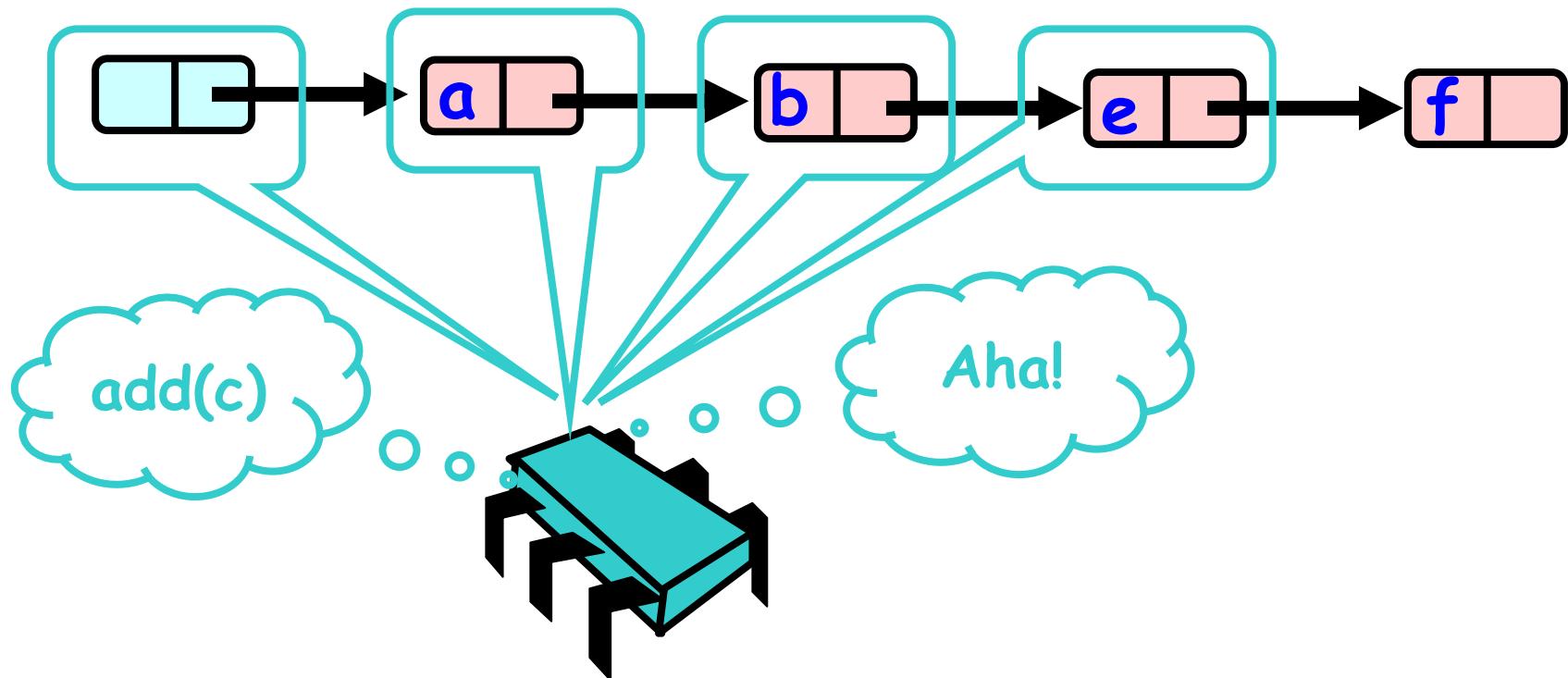
- For example:
  - A thread removing the second item in the list still blocks all concurrent threads searching for later nodes
- Possible solution:
  - To take a chance



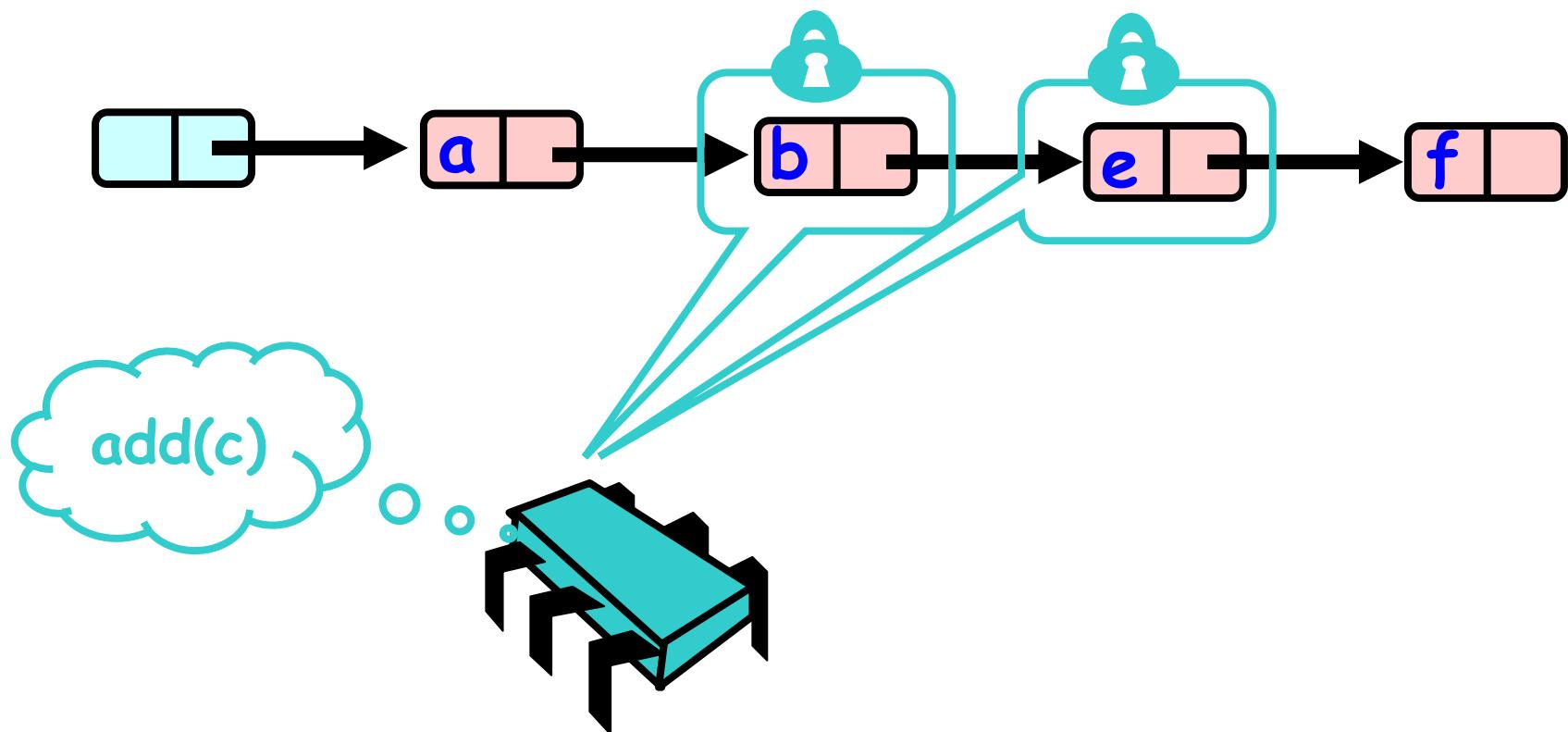
# Optimistic synchronization

- Search without acquiring locks
- Lock the nodes found
- Confirm that the locked nodes are correct
- If a synchronization error caused the wrong nodes to be locked, start again

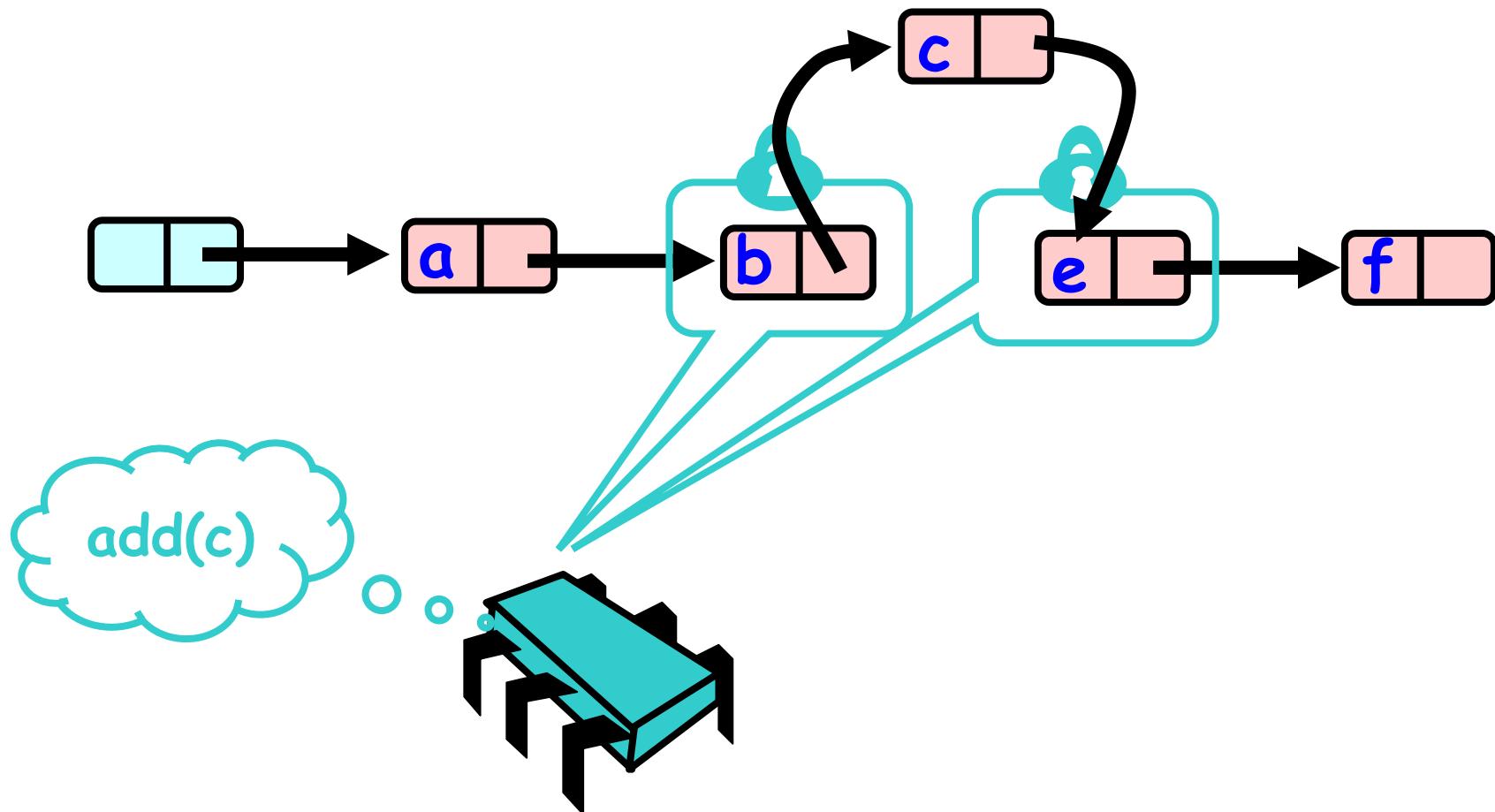
# Optimistic: Traverse without Locking



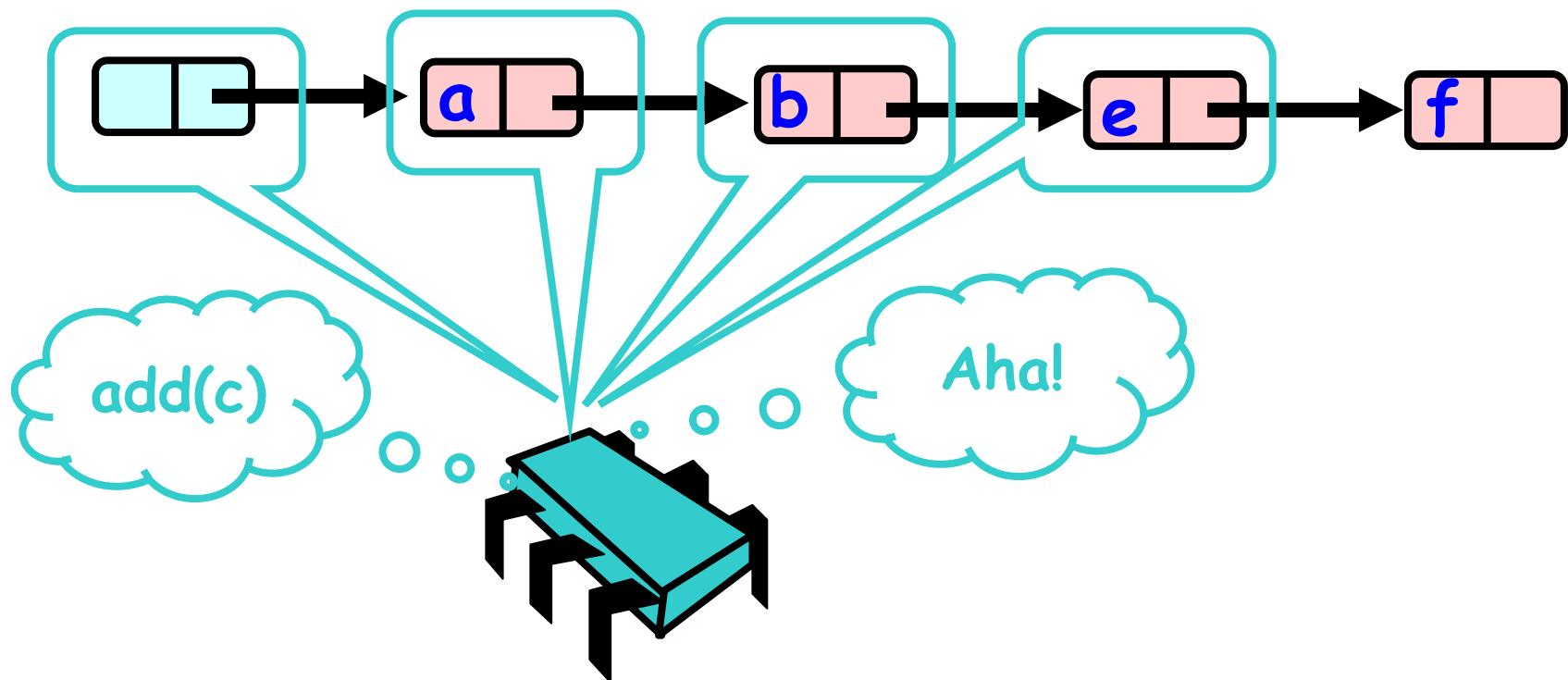
# Optimistic: Lock and Load



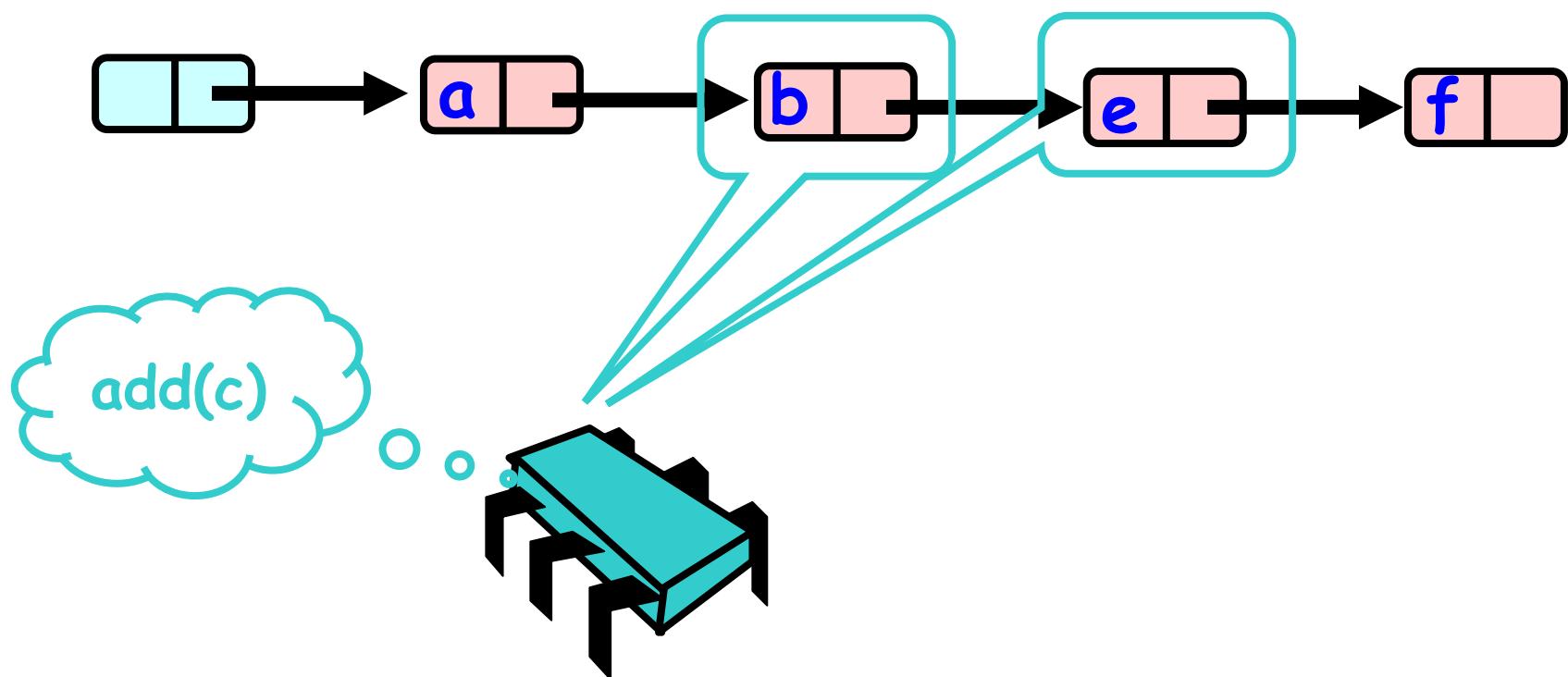
# Optimistic: Lock and Load



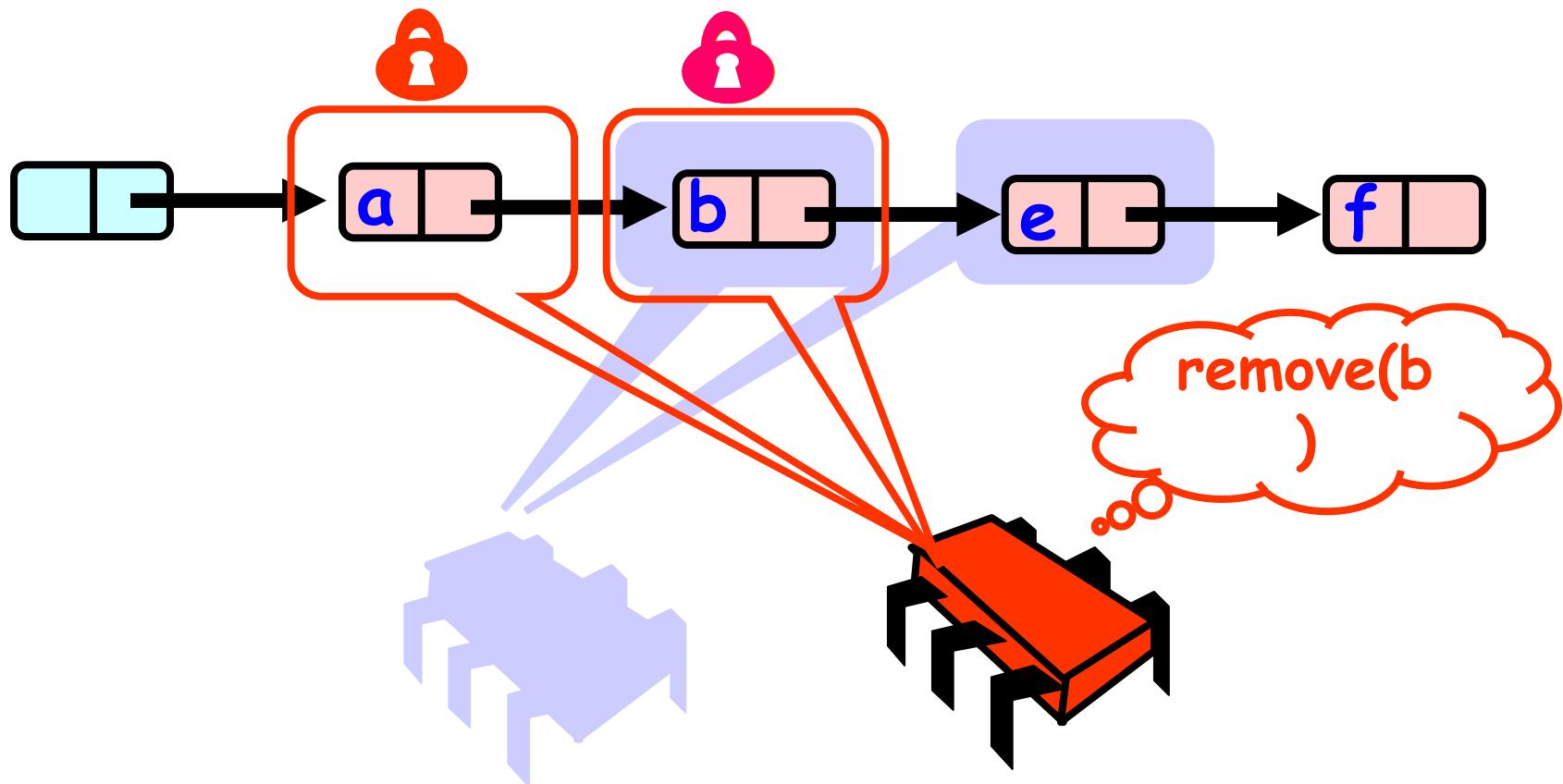
# What could go wrong?



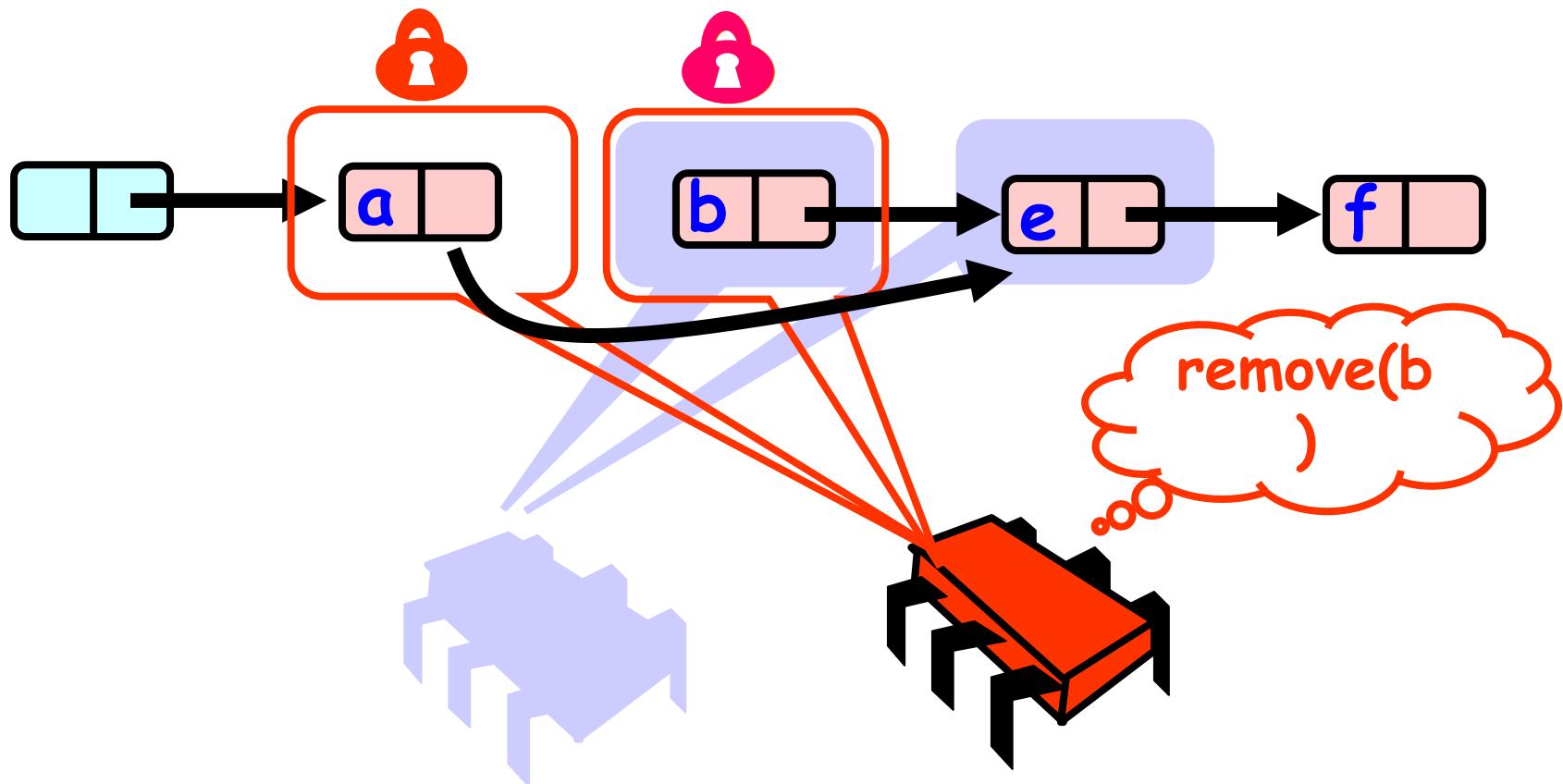
# What could go wrong?



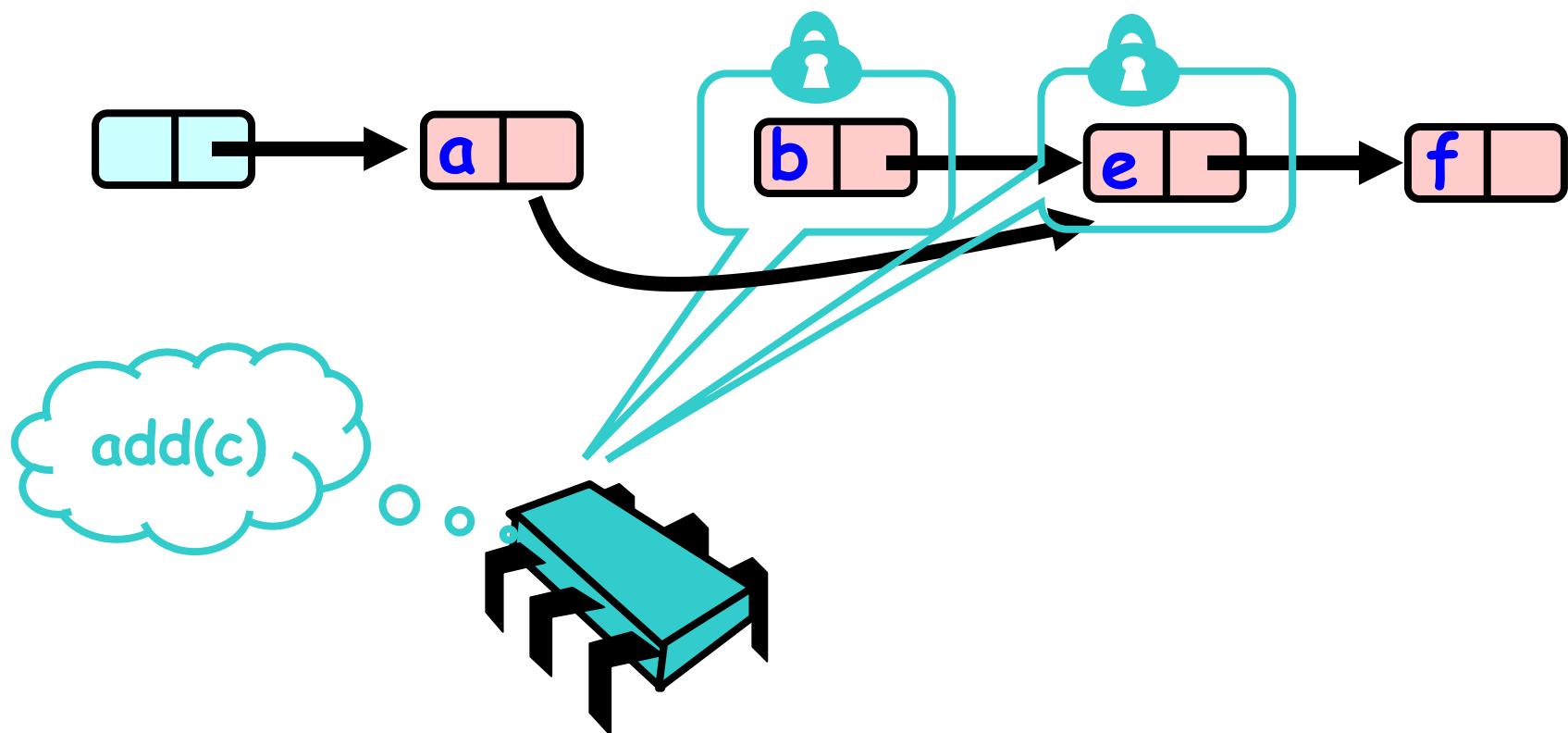
# What could go wrong?



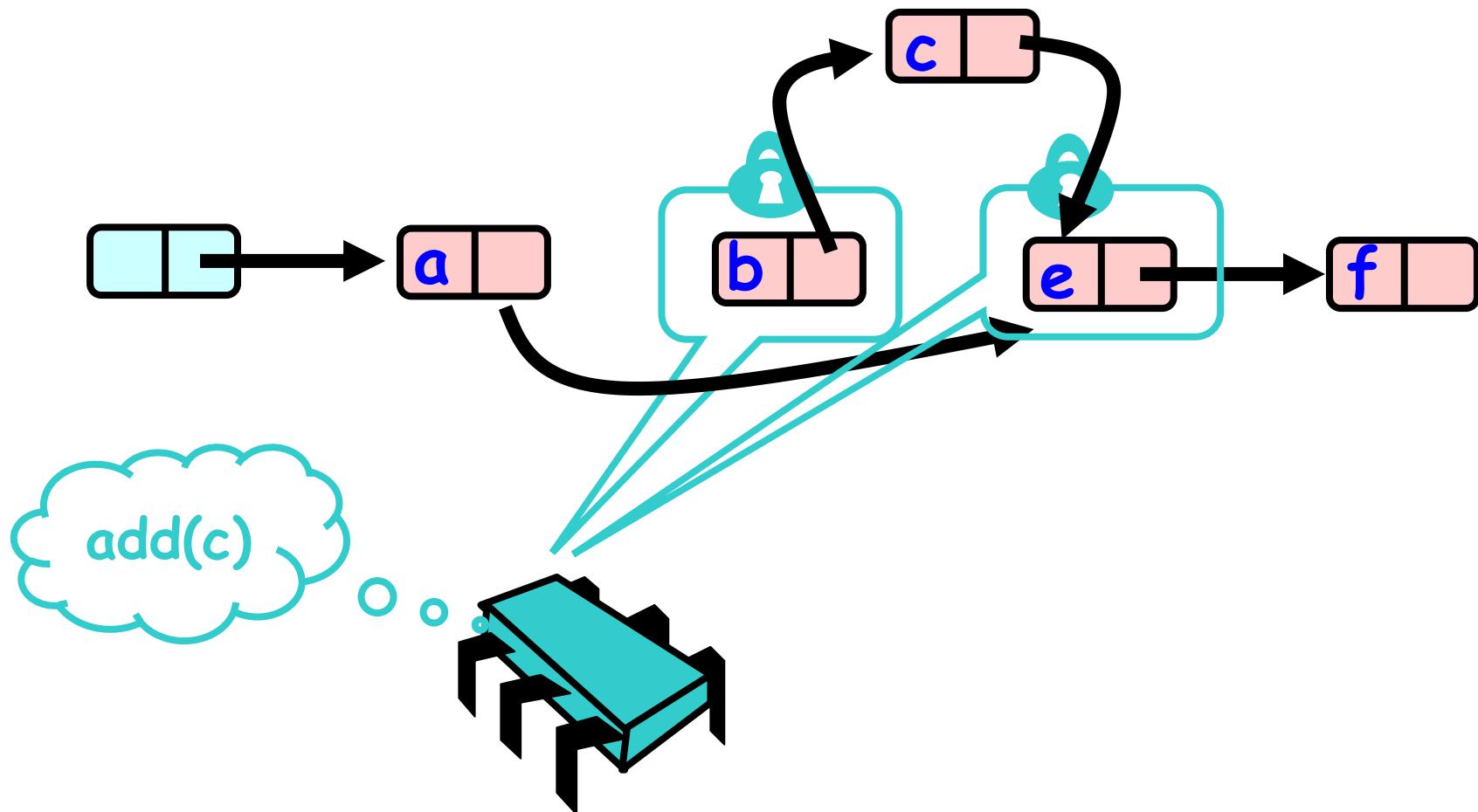
# What could go wrong?



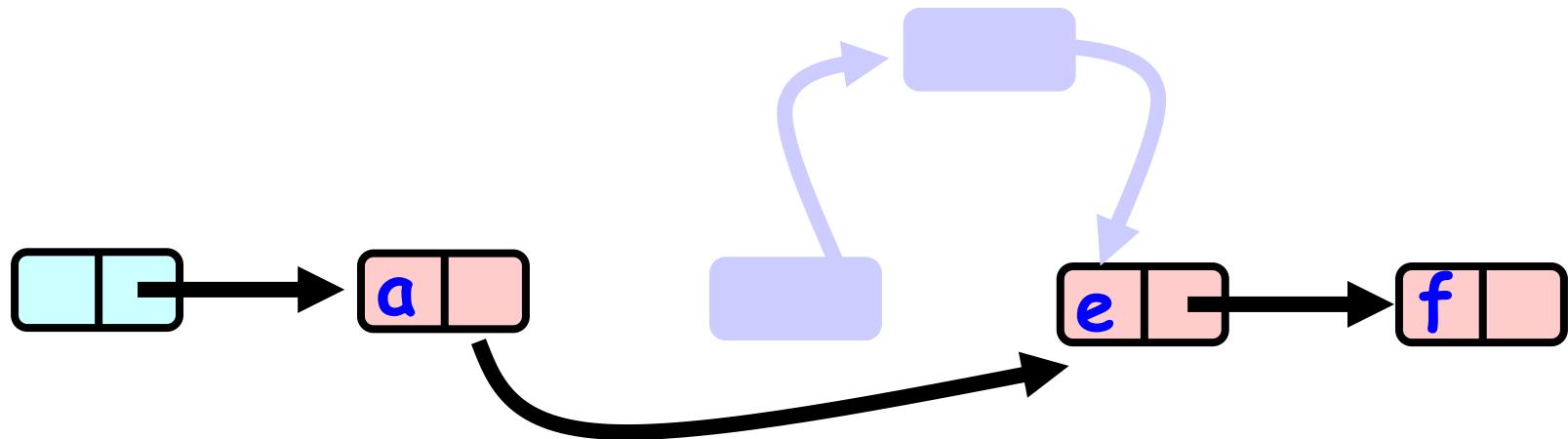
# What could go wrong?



# What could go wrong?

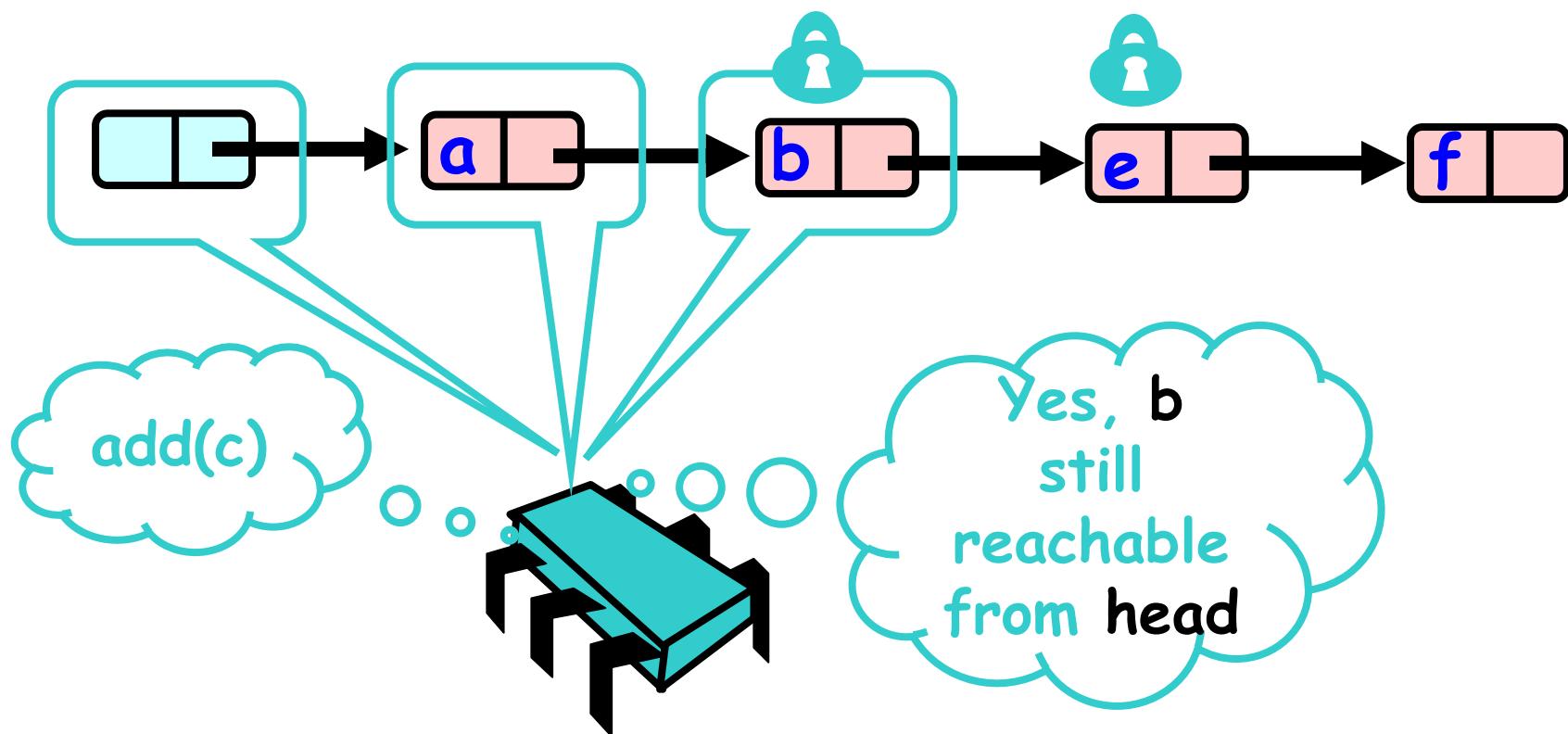


# What could go wrong?

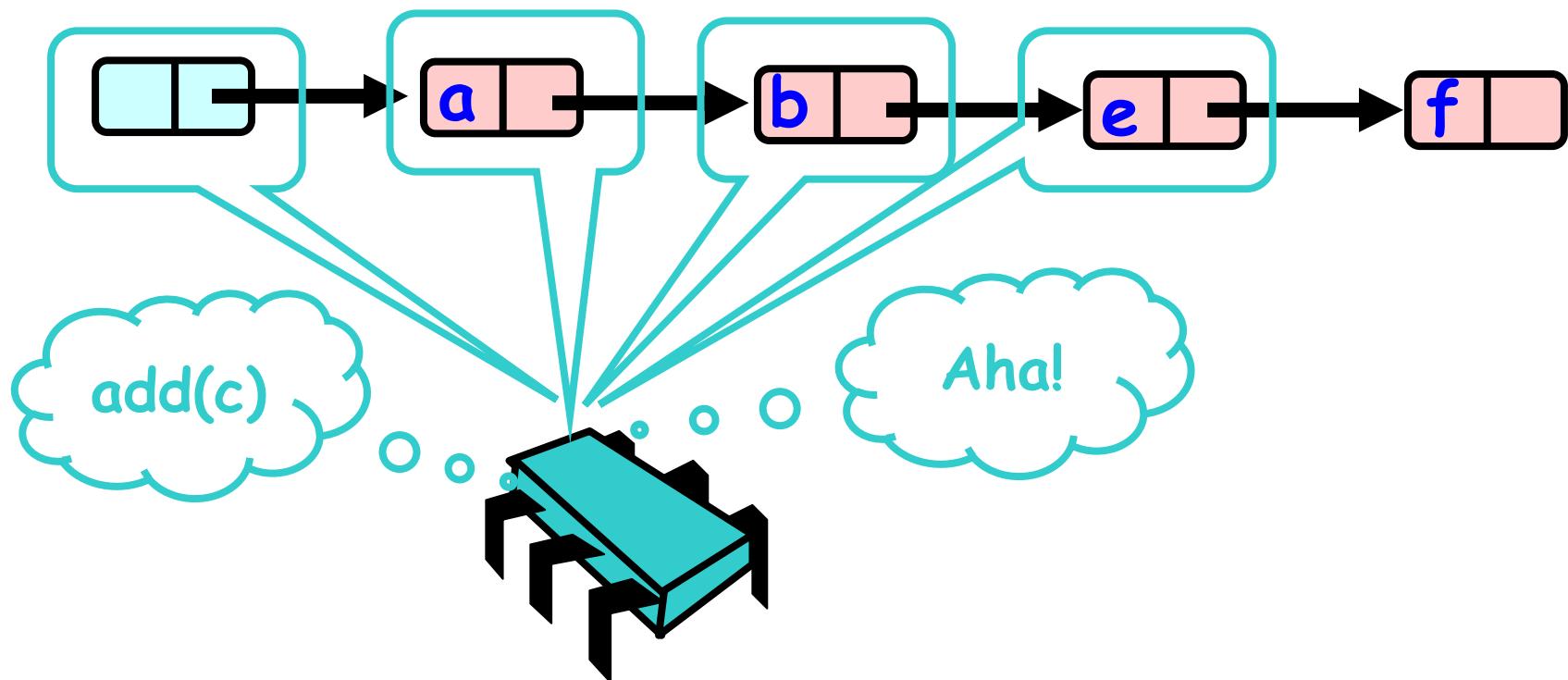


Uh-oh

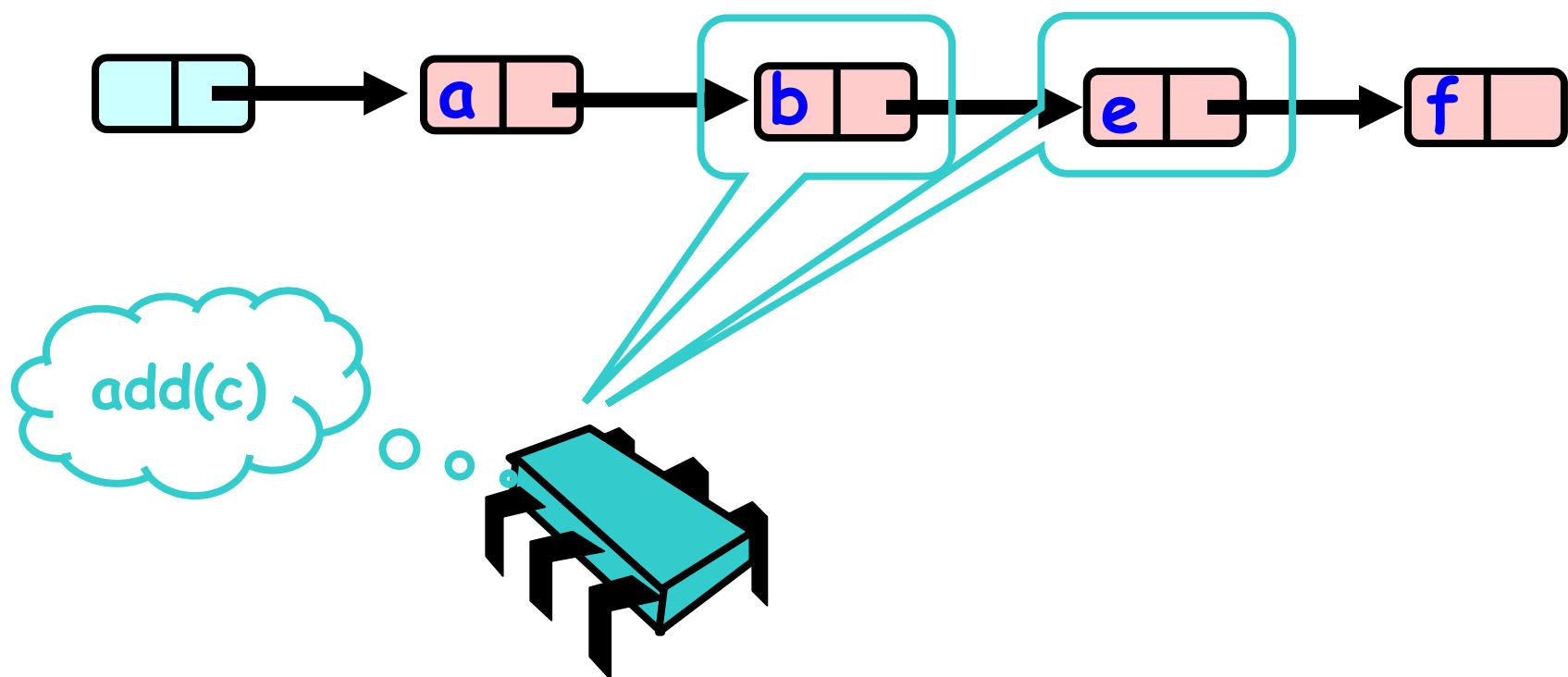
# Validate – Part 1



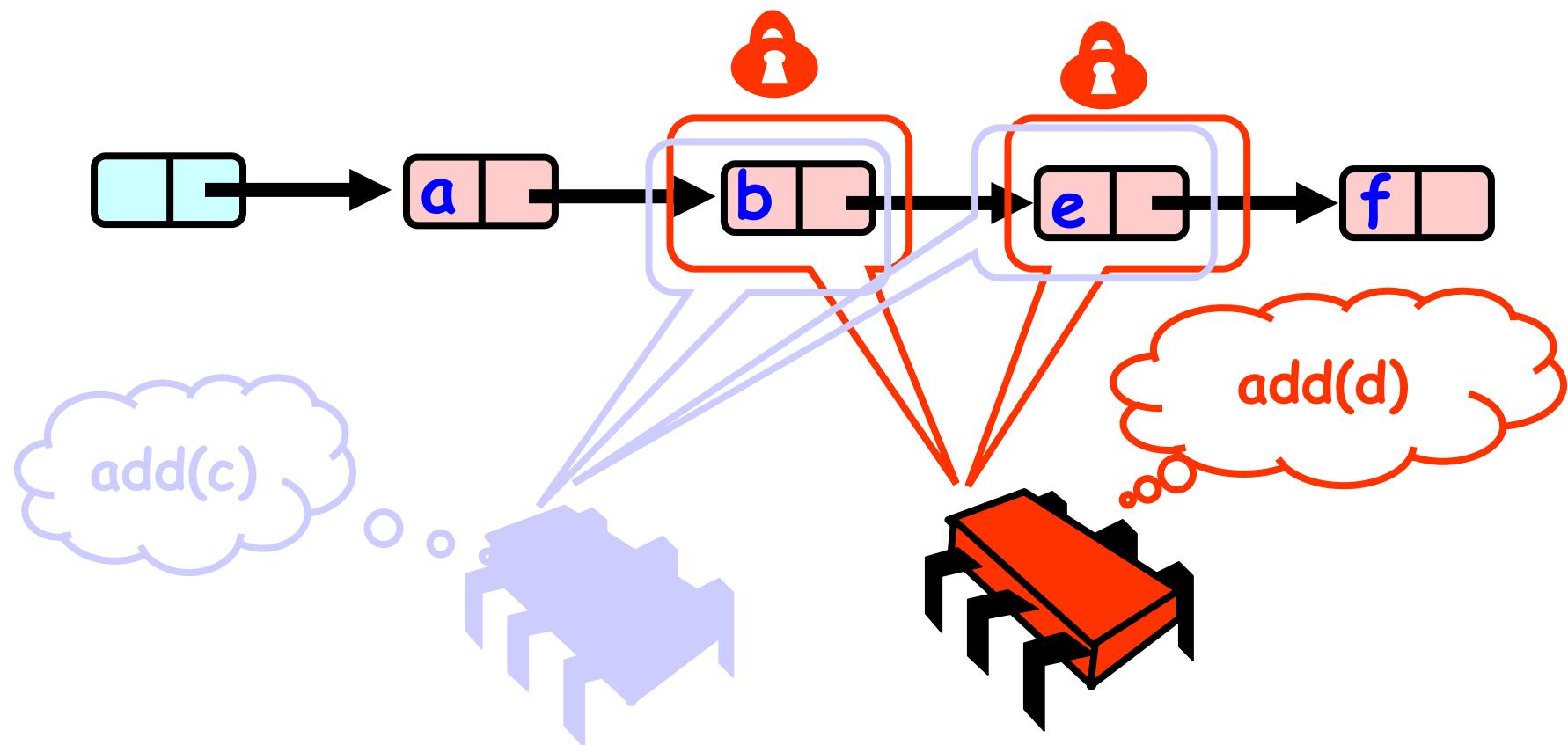
# What else could go wrong?



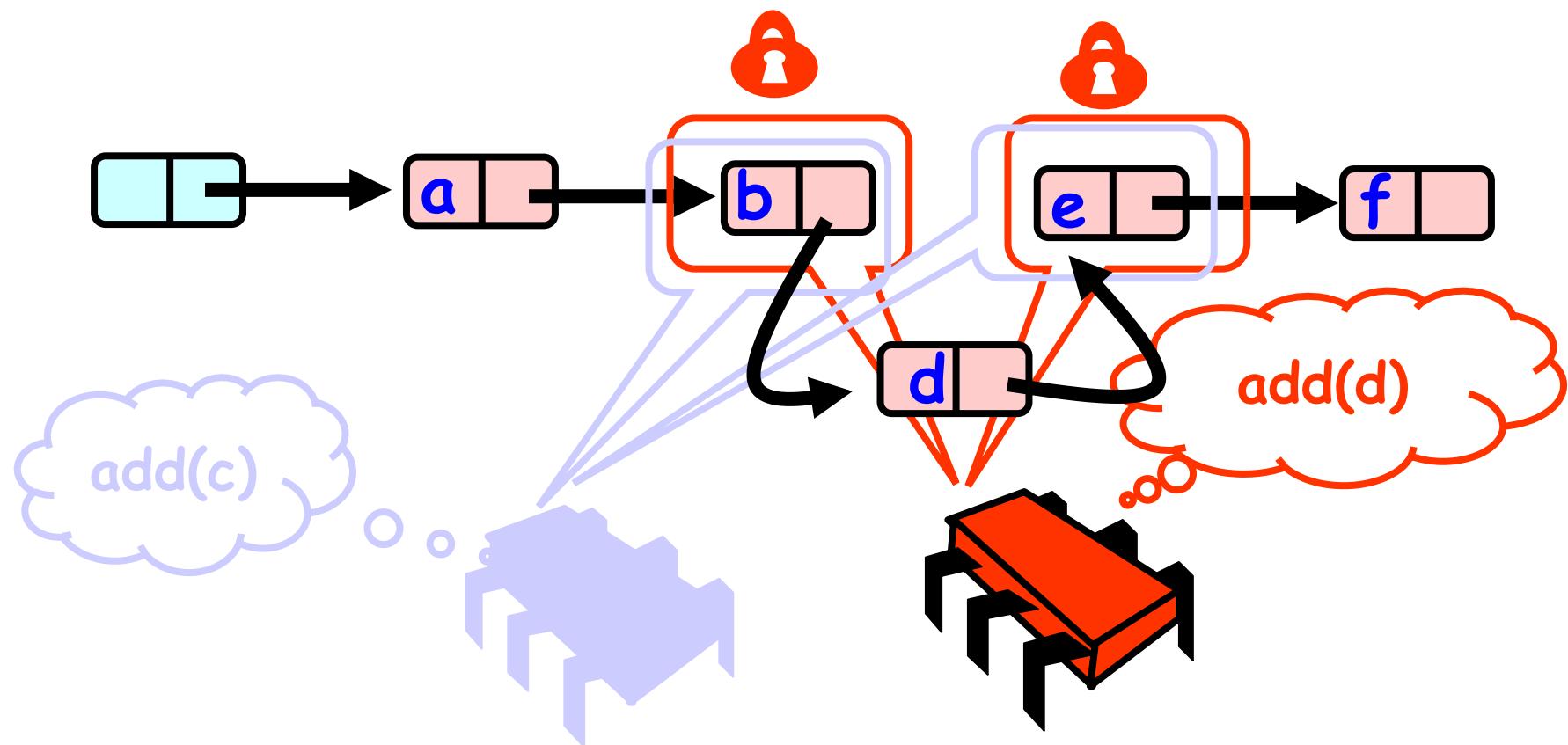
# What else could go wrong?



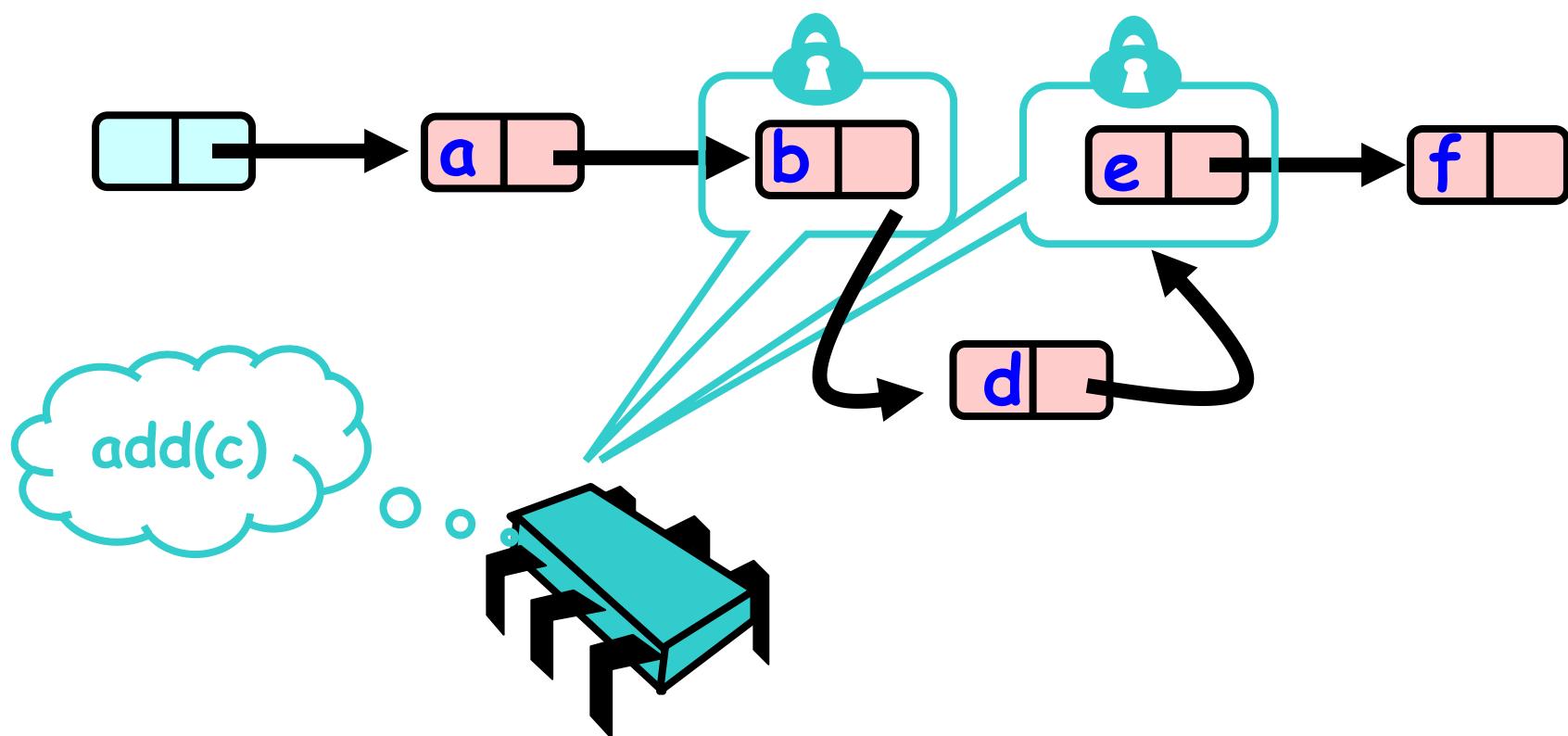
# What else could go wrong?



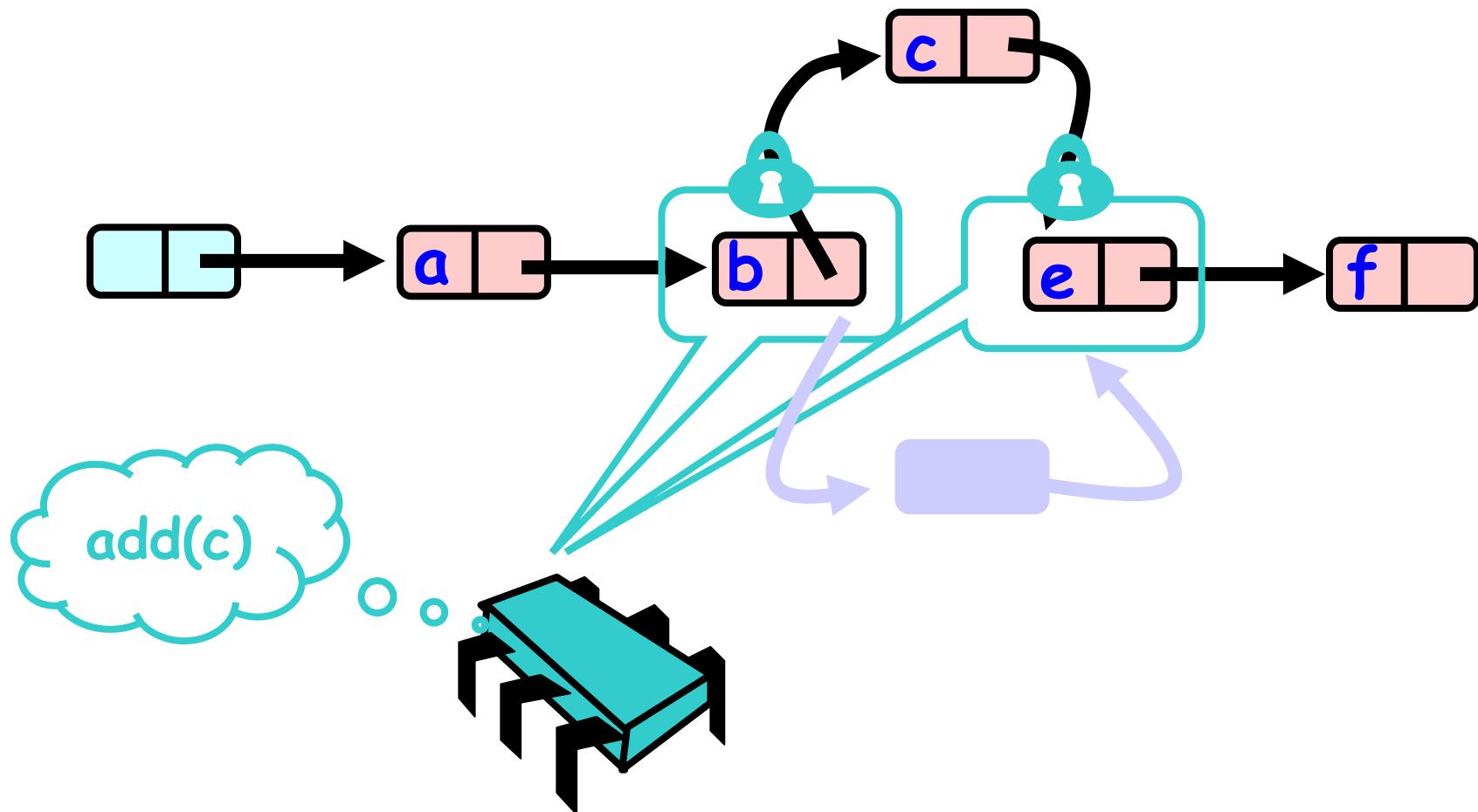
# What else could go wrong?



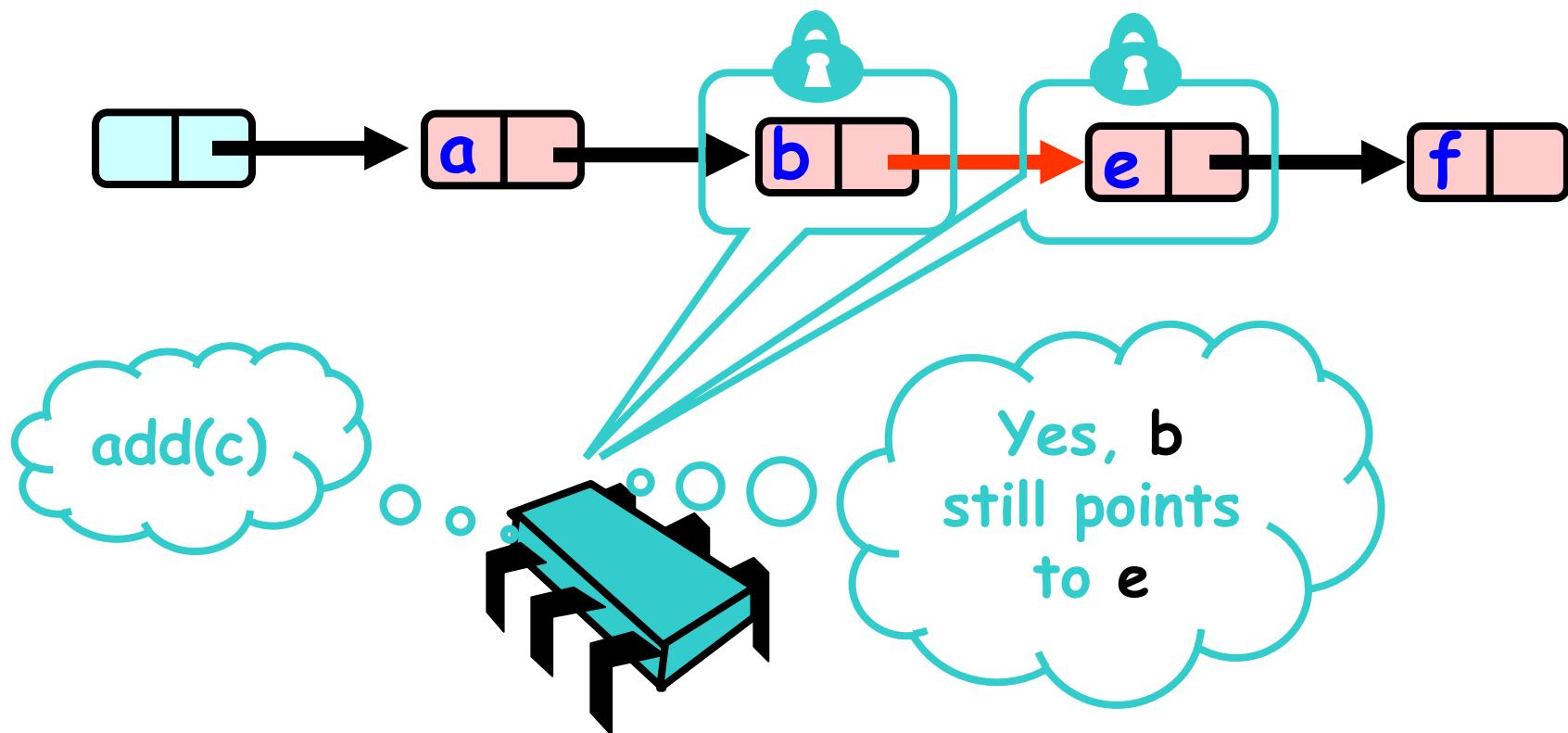
# What else could go wrong?



# What else could go wrong?



# Validate Part 2 (while holding locks)



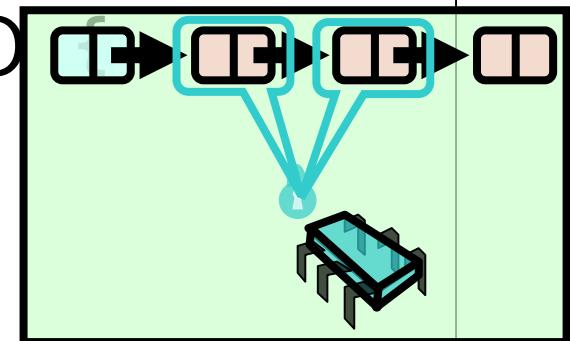
# Validation

```
private boolean  
validate(Node pred,  
        Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

# Validation

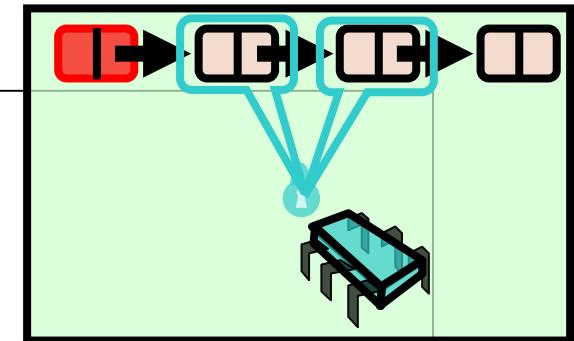
```
private boolean validate(Node pred,  
    Node curr) {  
  
    Node node = head;  
    while (node.key <= pred.key)  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

Predecessor &  
current nodes



# Validation

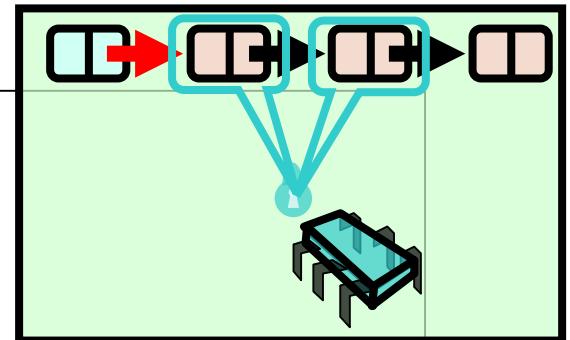
```
private boolean validate(Node pred,  
                       Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```



Begin at the  
beginning

# Validation

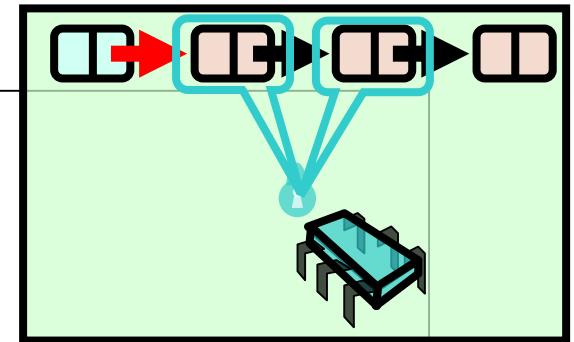
```
private boolean validate(Node pred,  
                       Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```



Search range of keys

# Validation

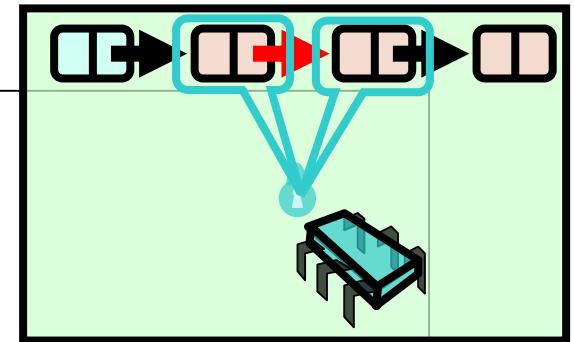
```
private boolean validate(Node pred,  
                       Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```



Predecessor reachable

# Validation

```
private boolean validate(Node pred,  
                        Node curr) {  
    Node node = head;  
    while (node.key <= pred.key) {  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

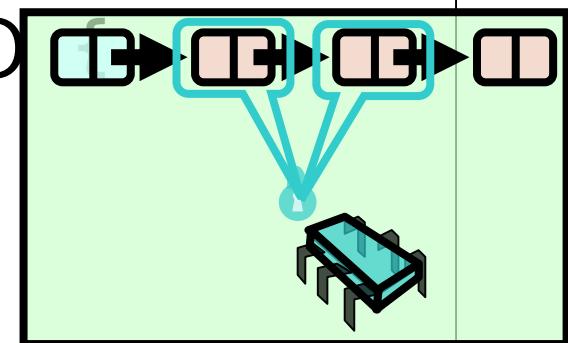


Is current node next?

# Validation

```
private boolean validate(Node pred,  
                       Node curr) {  
    Node node = head;  
    while (node.key <= pred.key)  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
    return false;  
}
```

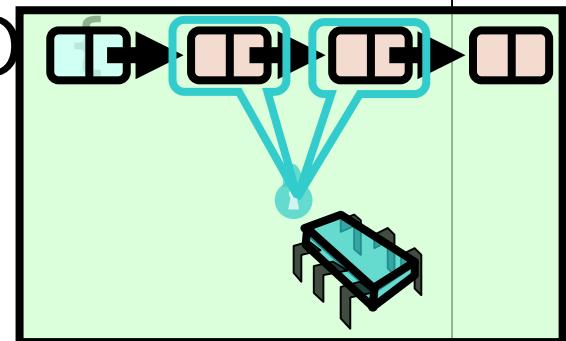
*Otherwise move on*



# Validation

```
private boolean validate(Node pred,  
                       Node curr) {  
    Node node = head;  
    while (node.key <= pred.key)  
        if (node == pred)  
            return pred.next == curr;  
        node = node.next;  
    }  
  
    return false;  
}
```

**Predecessor not reachable**



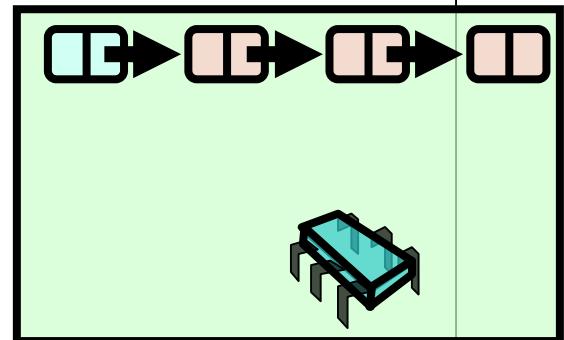
# Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key <= key) {  
        if (item == curr.item)  
            break;  
        pred = curr;  
        curr = curr.next;  
    } ...
```

# Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key <= key) {  
        if (item == curr.item)  
            break;  
        pred = curr;  
        curr = curr.next;  
    } ...
```

Search key

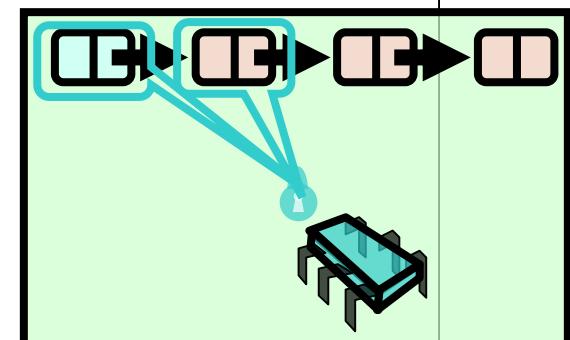


# Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key <= key) {  
        if (item == curr.item)  
            break;  
    }  
}
```

Predecessor and current  
nodes

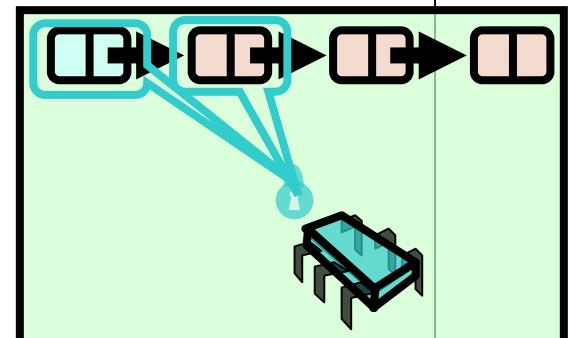
, ...



# Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key <= key) {  
        if (item == curr.item)  
            break;  
        pred = curr;  
        curr = curr.next;  
    } ...
```

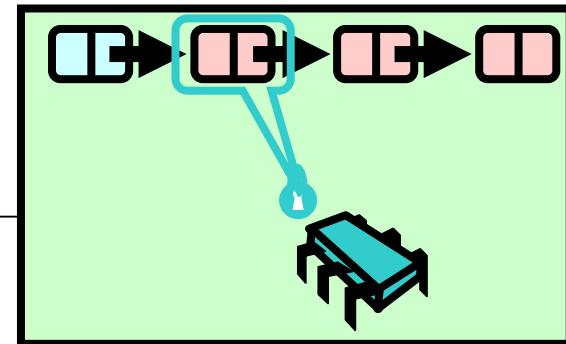
Search by key



# Remove: searching

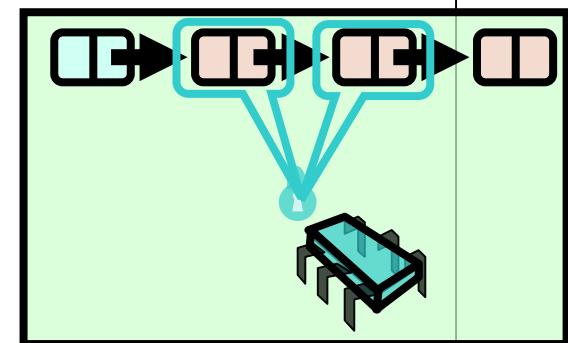
```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key <= key) {  
        if (item == curr.item)  
            break;  
        pred = curr;  
        curr = curr.next;  
    } ...  
}
```

Stop if we find item

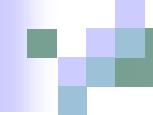


# Remove: searching

```
public boolean remove(Item item) {  
    int key = item.hashCode();  
    Node pred = this.head;  
    Node curr = pred.next;  
    while (curr.key <= key) {  
        if (item == curr.item)  
            break;  
        pred = curr;  
        curr = curr.next;  
    } ...  
}
```



Move along



# On Exit from Loop

- If item is present
  - curr holds item
  - pred just before curr
- If item is absent
  - curr has first higher key
  - pred just before curr
- Assuming no synchronization problems

# Remove Method

```
try {
    pred.lock(); curr.lock();
    if (validate(pred,curr) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

# Remove Method

```
try {  
    pred.lock(); curr.lock();  
    if (validate(pred, curr) {  
        if (curr.item == item) {  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    } finally {  
        pred.unlock();  
        curr.unlock();  
    }  
}
```

Always unlock

# Remove Method

```
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    }
} finally {
    pred.unlock();
    curr.unlock();
}
```

Lock both nodes

# Remove Method

```
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

*Check for synchronization conflicts*

# Remove Method

```
try {  
    pred.lock(); curr.lock();  
    if (validate(pred, curr)) {  
        if (curr.item == item) {  
            pred.next = curr.next;  
            return true;  
        } else {  
            return false;  
        }  
    }  
} finally {  
    pred.unlock();  
    curr.unlock();  
}
```

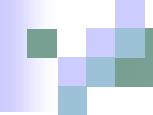
target found,  
remove node

# Remove Method

```
try {
    pred.lock(); curr.lock();
    if (validate(pred,curr) {
        if (curr.item == item) {
            pred.next = curr.next;
            return true;
        } else {
            return false; target not found
        }
    } } finally {
    pred.unlock();
    curr.unlock();
} }
```

# Optimistic List

- Limited hot-spots
  - Targets of add() & remove()
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought ...



# What about contains()?

- Contains() imply that the item is in the list, if and only if it is reachable
- Coarse-grained synchronization?
- Fine-grained synchronization?
- Optimistic synchronization?



# Coarse-grained synchronization

- Works much the same as add() and remove()
- Thread acquires lock, searches through the list, returns true/false, releases the lock



# Fine-grained synchronization

- Also works much the same as add() and remove()
- Threads that search through the list, acquire and release the pred and curr lock until the item is found or it reaches the end of the list



# Optimistic synchronization

- Thread traverses through the list without locking until items are found or the end of the list is reached
- Does this mean that the item is reachable however?
- Nodes are then locked and determined if they are reachable

# Optimistic synchronization

```
public boolean contains(T item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next;  
    }  
    try {  
        pred.lock(); curr.lock();  
        if (validate(pred, curr))  
            return (curr.key == key)  
    } finally {  
        pred.unlock(); curr.unlock();  
    }  
}
```

# Optimistic synchronization

```
public boolean contains(T item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next;  
    }  
    try {  
        pred.lock(); curr.lock();  
        if (validate(pred, curr))  
            return (curr.key == key)  
    } finally {  
        pred.unlock(); curr.unlock();  
    }  
}
```

**Search for item**

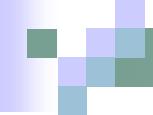
# Optimistic synchronization

```
public boolean contains(T item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next; Acquire lock  
    }  
    try {  
        pred.lock(); curr.lock();  
        if (validate(pred, curr))  
            return (curr.key == key)  
    } finally {  
        pred.unlock(); curr.unlock();  
    }  
}
```

# Optimistic synchronization

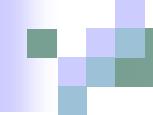
```
public boolean contains(T item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key < key) {  
        pred = curr; curr = curr.next;  
    }  
    try {  
        pred.lock(); curr.lock();  
        if (validate(pred, curr))  
            return (curr.key == key)  
    } finally {  
        pred.unlock(); curr.unlock();  
    }  
}
```

Is item  
reachable?



# Optimistic synchronization

- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - `contains()` method still acquires locks



# Lazy synchronization

- `contains()` calls are likely to be made more often than calls to other methods
- Idea of lazy synchronization is to refine optimistic synchronization so that `contains()` calls are wait-free and `add()` and `remove()` calls traverse the list only once (in the absence of contention)



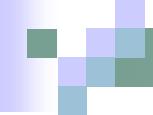
# Lazy synchronization

- Each node has an additional boolean field called **marked**
- **marked** indicates whether the node is in the list (reachable)
- Now there is no need to validate if the node is reachable – every unmarked node is reachable



# Lazy synchronization

- If a thread does not find a node, or finds it marked, that item is not in the list
- As a result contains() needs only one wait-free traversal



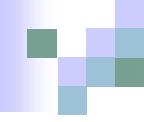
# Lazy synchronization

- Like optimistic, except
  - Scan once
  - `contains(x)` never locks ...
- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”



# Lazy synchronization

- All methods traverse the list ignoring locks
- add() and remove() lock pred and curr as with Optimistic
- Validation however does not require a traversal through the list to determine if a node is reachable



# Validation

- No need to rescan list!
- Check that pred is not marked
- Check that curr is not marked
- Check that pred points to curr

# Validation

```
private boolean  
    validate(Node pred, Node curr) {  
return  
    !pred.marked &&  
    !curr.marked &&  
    pred.next == curr);  
}
```

# List Validate Method

```
private boolean  
    validate(Node pred, Node curr) {  
return  
    !pred.marked &&  
    !curr.marked &&  
    pred.next == curr);  
}
```

Predecessor not  
Logically removed

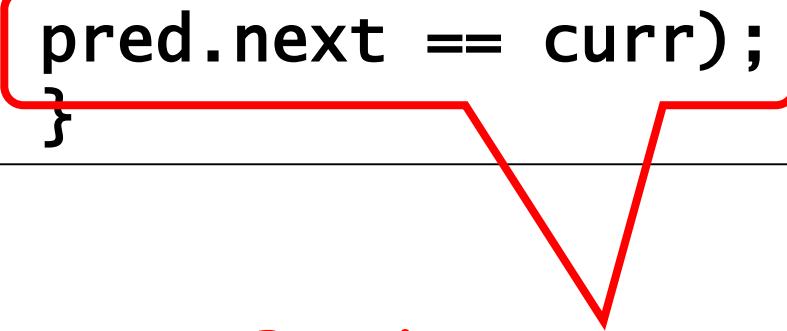
# List Validate Method

```
private boolean  
    validate(Node pred, Node curr) {  
    return  
        !pred.marked &&  
        !curr.marked &&  
        pred.next == curr);  
}
```

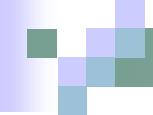
Current not  
Logically removed

# List Validate Method

```
private boolean  
    validate(Node pred, Node curr) {  
    return  
        !pred.marked &&  
        !curr.marked &&  
        pred.next == curr);  
    }
```



Predecessor still  
Points to current



# Contains() method

- Thread traverse through list
- Instead of locking pred and curr, the **marked** field of the target is checked
- Contains() is wait-free

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Start at the head

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Search key range

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Traverse without locking  
(nodes may have been removed)

# Contains

```
public boolean contains(Item item) {  
    int key = item.hashCode();  
    Node curr = this.head;  
    while (curr.key < key) {  
        curr = curr.next;  
    }  
    return curr.key == key && !curr.marked;  
}
```

Present and undeleted?



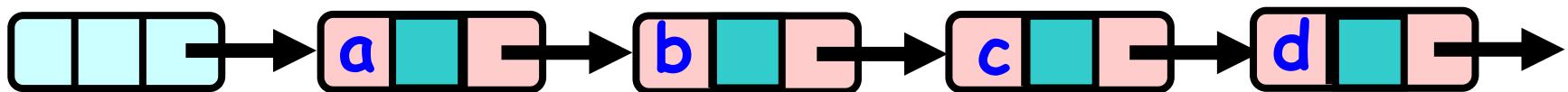
# Add() method

- Same as Optimistic synchronization
- Validate() method differs

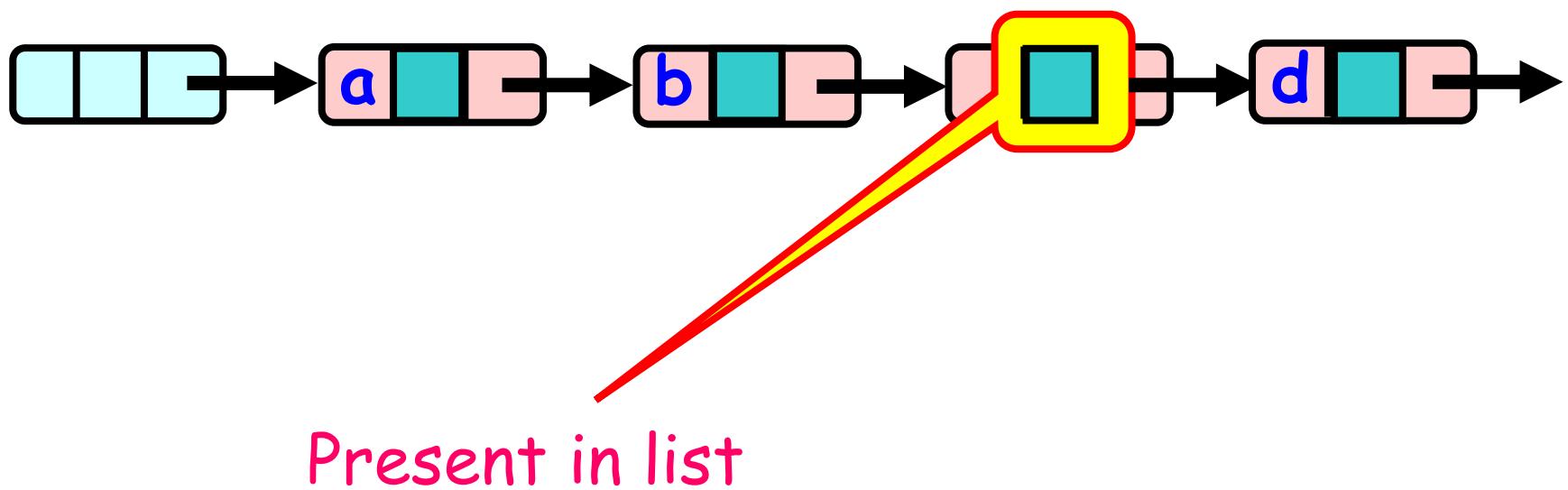
# Remove() method

- Divided into:
  - Logical removal – set the node's **marked** field
  - Physical removal – change the links to remove node from linked list
- Thread traverses through list without locks
- When item is found, acquire locks, validate and remove

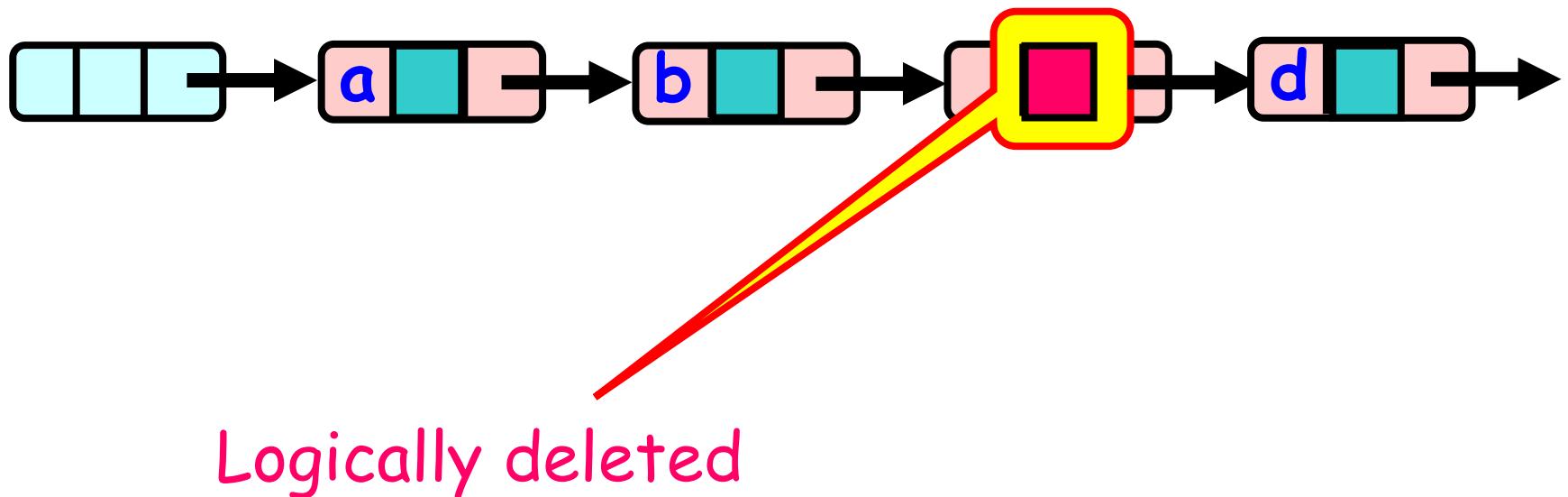
# Lazy Removal



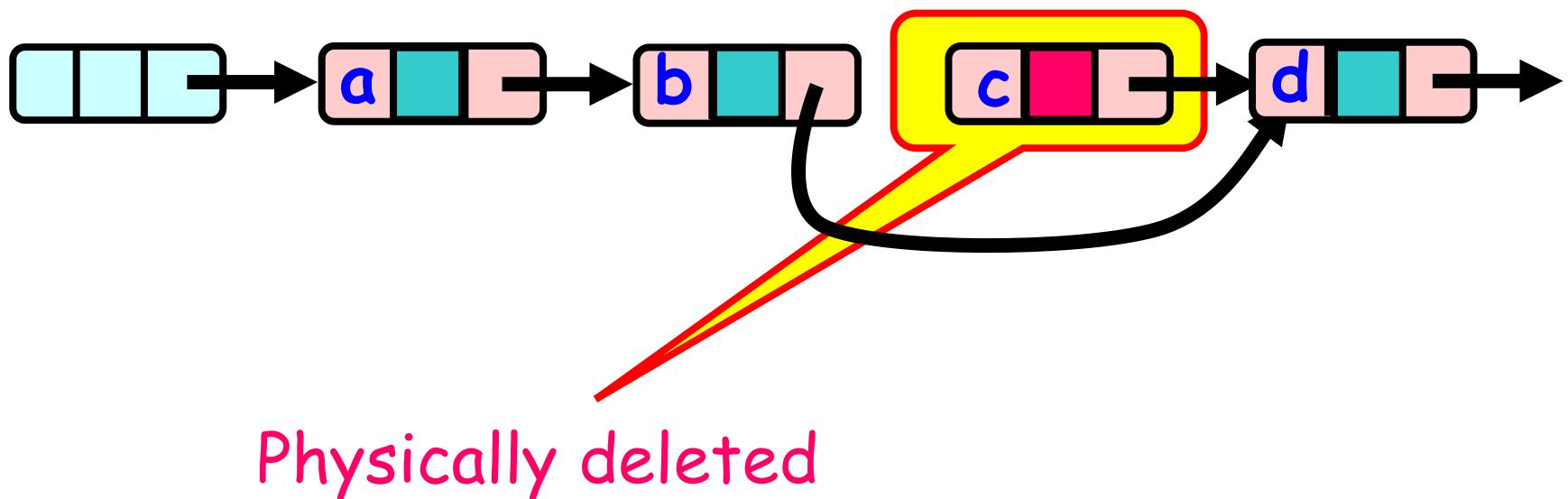
# Lazy Removal



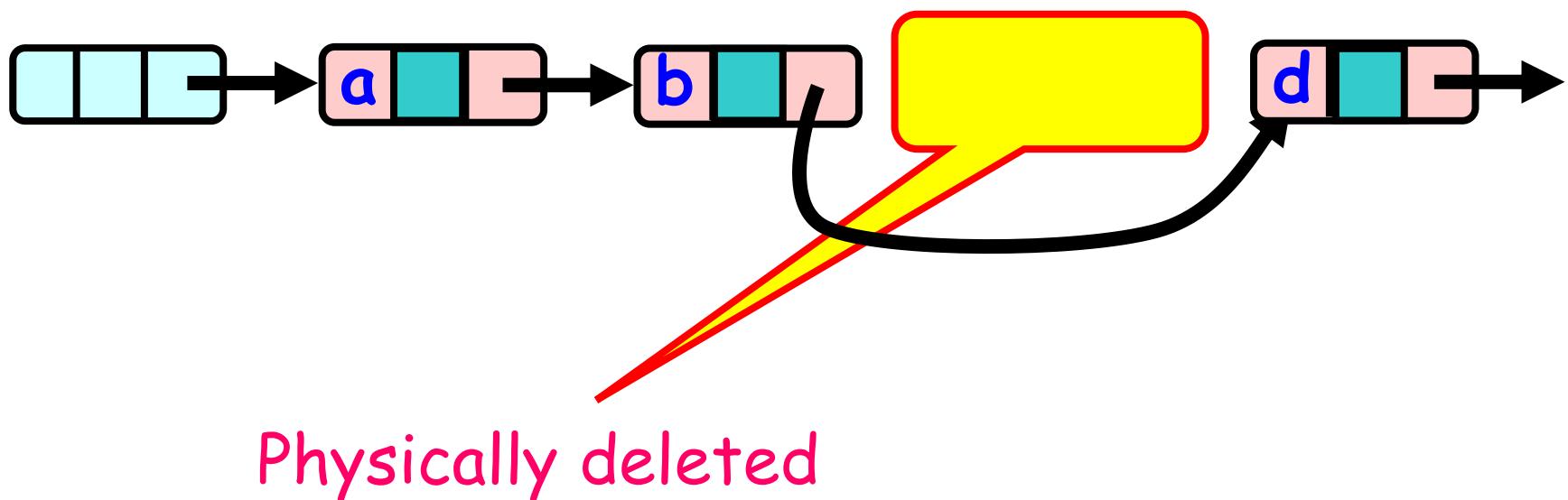
# Lazy Removal



# Lazy Removal



# Lazy Removal



# Remove

```
public boolean remove(T item) {  
    int key = item.hashCode();  
    Node pred = head;  
    Node curr = pred.next;  
    while (curr.key < key) {  
        pred = curr;  
        curr = curr.next;  
    }  
    ...
```

# Remove

```
try {
    pred.lock(); curr.lock();
    if (validate(pred,curr) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

# Remove

```
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    }
} finally {
    pred.unlock();
    curr.unlock();
}
```

Validate as before

# Remove

```
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    }
} finally {
    pred.unlock();
    curr.unlock();
}
```

Key found

# Remove

```
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } finally {
        pred.unlock();
        curr.unlock();
    }
}
```

Logical remove

# Remove

```
try {
    pred.lock(); curr.lock();
    if (validate(pred, curr)) {
        if (curr.key == key) {
            curr.marked = true;
            pred.next = curr.next;
            return true;
        } else {
            return false;
        }
    } finally {
        pred.unlock(); physical remove
        curr.unlock();
    }
}
```

# Evaluation of Lazy Synchronization

- Good:
  - contains() doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays

# Difference between Optimistic and Lazy Synchronization

	Optimistic	Lazy
contains()	Ignores locks, then locks pred and curr and validates before returning true/false	Ignores locks and returns true/false based on <b>marked</b> field
validate(pred, curr)	Traverses through list and validates if node is reachable from head and if curr follows on pred	Does not traverse but validates on <b>marked</b> fields of pred and curr and if curr follows on pred
add()	Ignores locks, then locks pred and curr and validates before adding	Ignores locks, then locks pred and curr and validates before adding
remove()	Ignores locks, then locks pred and curr and validates before removing	Ignores locks, then locks pred and curr, validates, changes <b>marked</b> field and then removes

# Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin”
    - Cache miss, page fault, descheduled ...
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler....

# Reminder: Lock-Free Data Structures

- No matter what ...
  - Guarantees minimal progress in any execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can't use locks



# Lock-free Synchronization

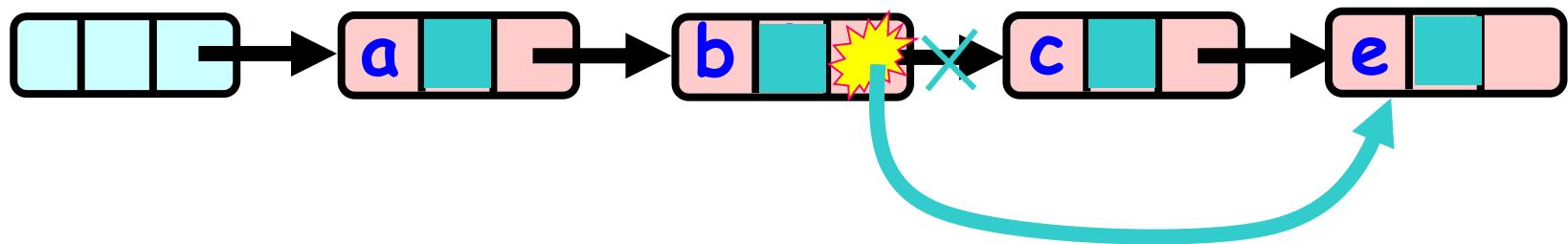
- We already have wait-free contains()
- Next logical step
  - lock-free add() and remove()
- Solution:
  - Use compareAndSet()



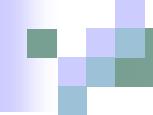
# Lock-Free synchronization

- Make use of compareAndSet() to change the next links when items are added or removed
- Since compareAndSet() is atomic, mutual exclusion is enforced
- What could go wrong?

# Remove Using CAS



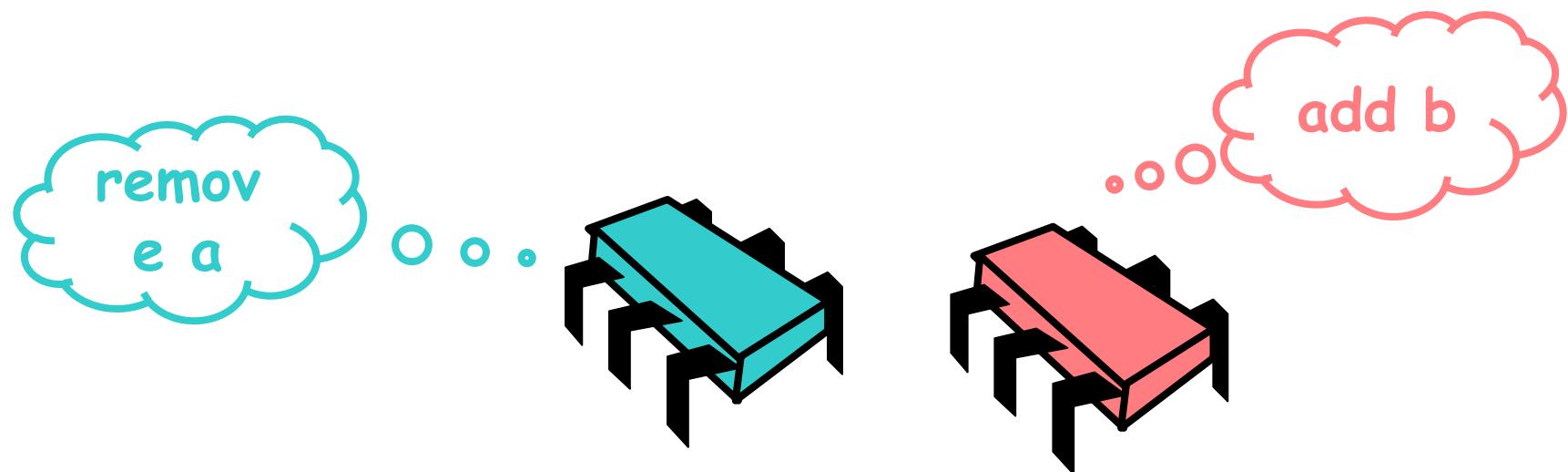
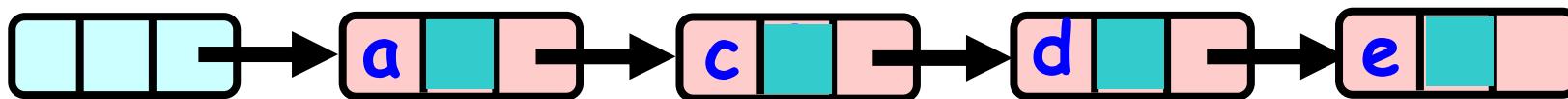
- `remove(c)`
- Use `compareAndSet()` to set b's next field to point to e



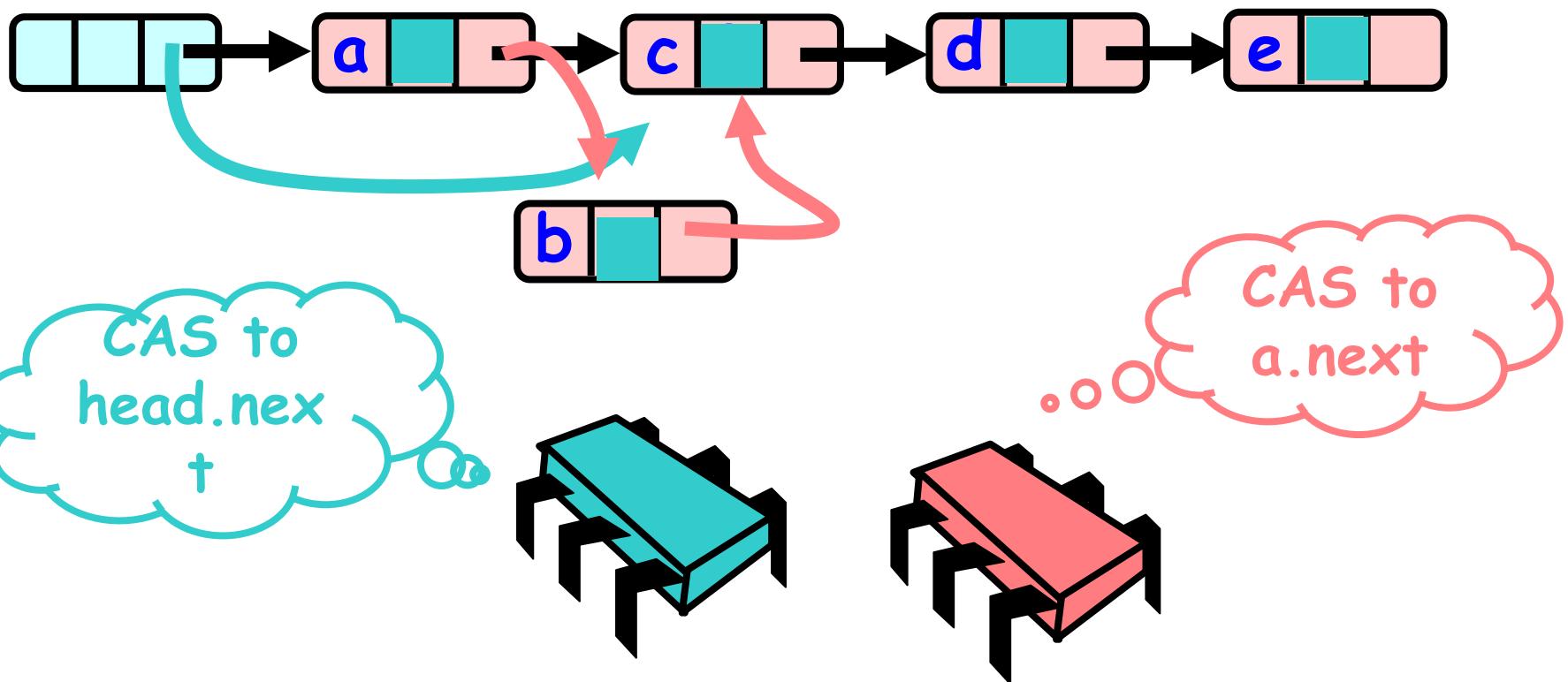
# Remove using CAS

- Unfortunately this idea does not work

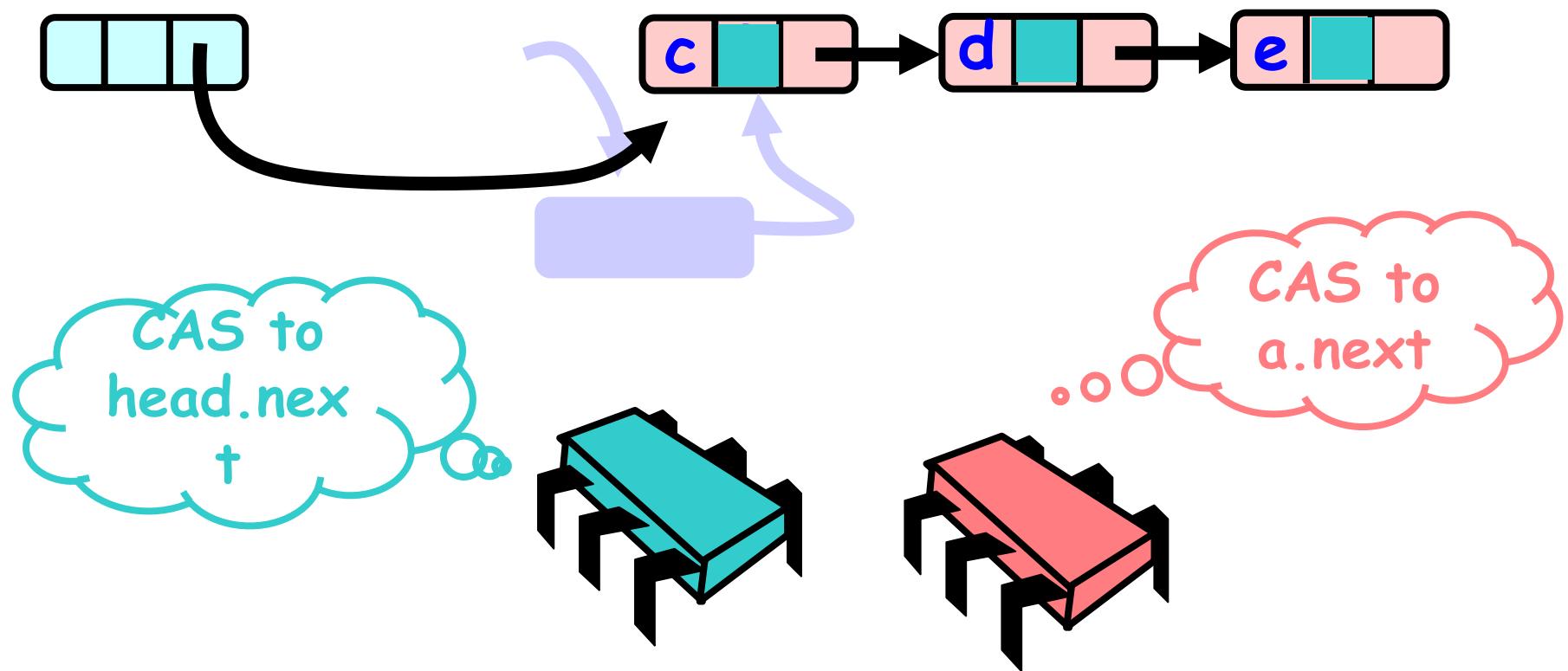
# Remove using CAS

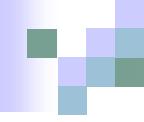


# Remove using CAS



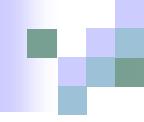
# Remove using CAS





# Non-blocking synchronization

- We need a way to ensure that a node's fields cannot be updated after that node has been logically/physically deleted from the list
- Use a **marked** field
- Any attempt to update the next field when the **marked** field is true will fail



# Non-blocking synchronization

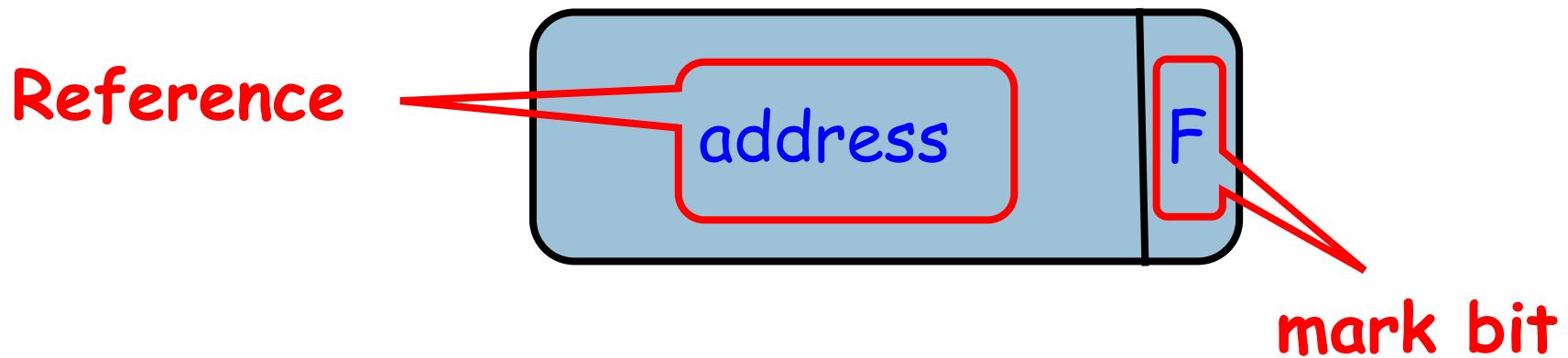
- Our approach:
  - To treat the next and marked fields as a single unit

# Solution

- Use AtomicMarkableReference
- Atomically
  - Swing reference and
  - Update flag
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer

# Marking a Node

- **AtomicMarkableReference** class
  - Java.util.concurrent.atomic package



# Changing State

```
Public boolean compareAndSet(  
    Object expectedRef,  
    Object updateRef,  
    boolean expectedMark,  
    boolean updateMark);
```

# Changing State

If this is the current reference ...

```
Public boolean compareAndSet(  
    Object expectedRef,  
    Object updateRef,  
    boolean expectedMark,  
    boolean updateMark);
```

And this is the current mark ...

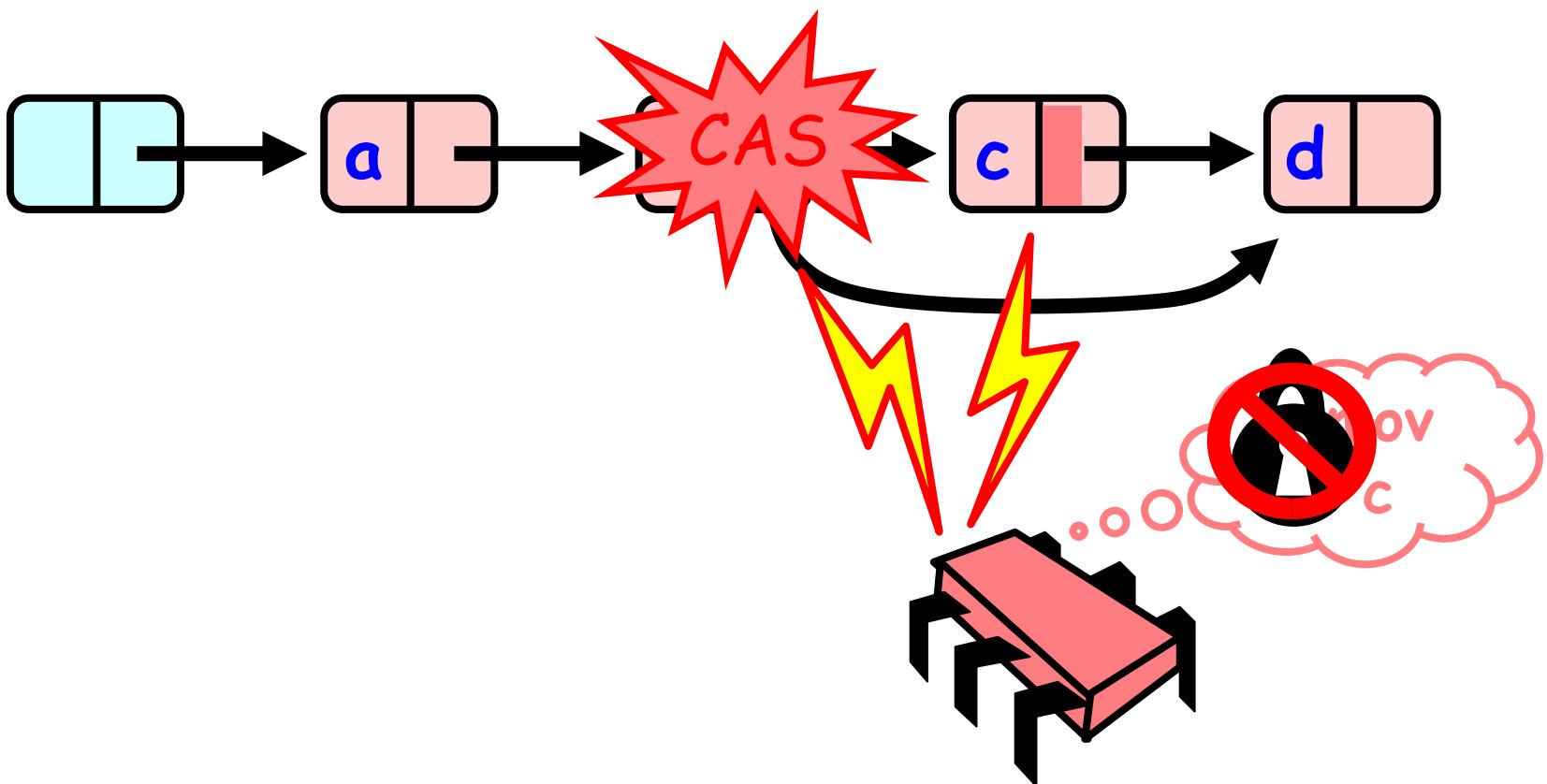
# Changing State

*...then change to this  
new reference ...*

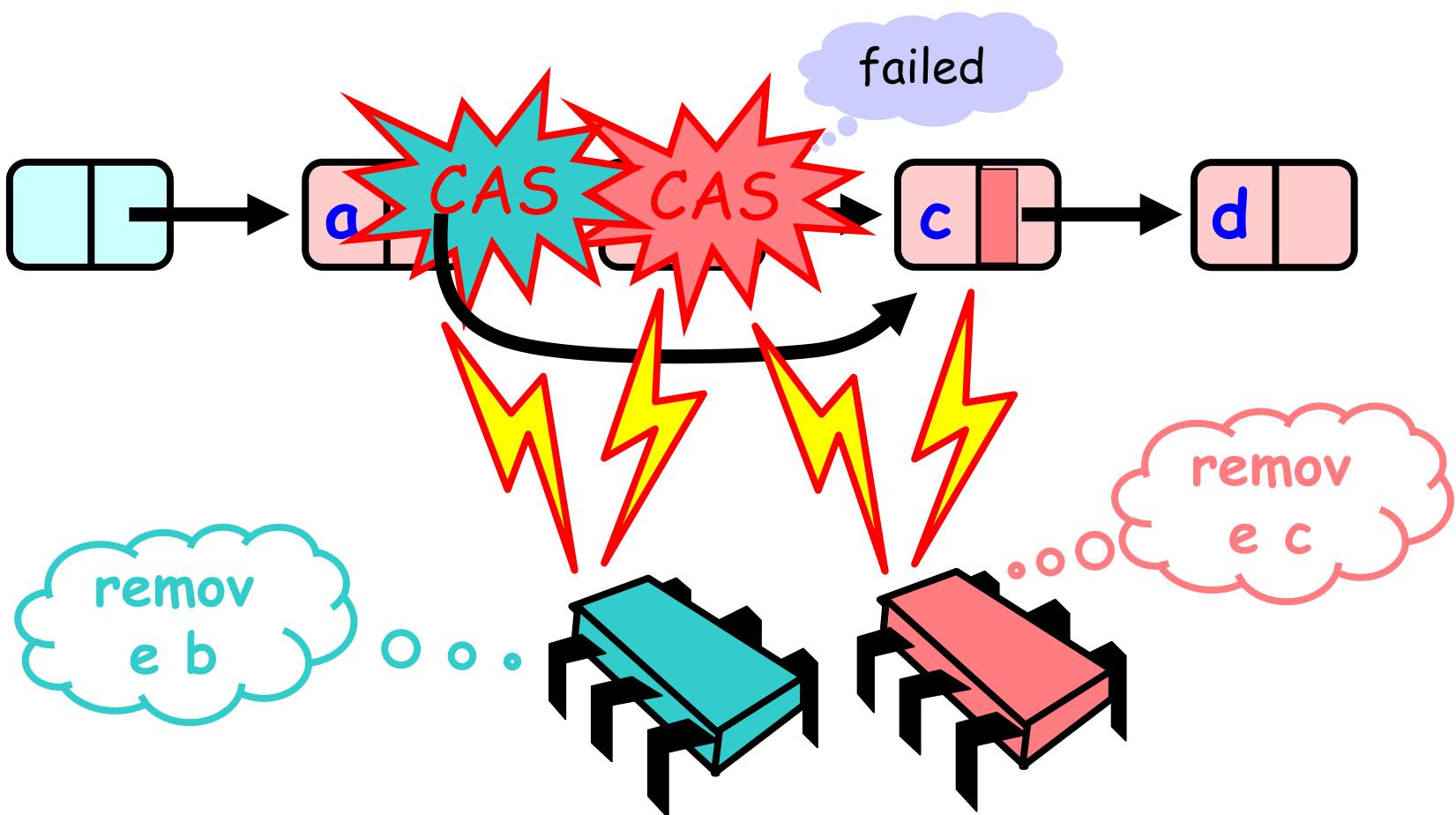
```
Public boolean compareAndSet(  
    Object expectedRef,  
    Object updateRef,  
    boolean expectedMark,  
    boolean updateMark);
```

*... and this new  
mark*

# Removing a Node

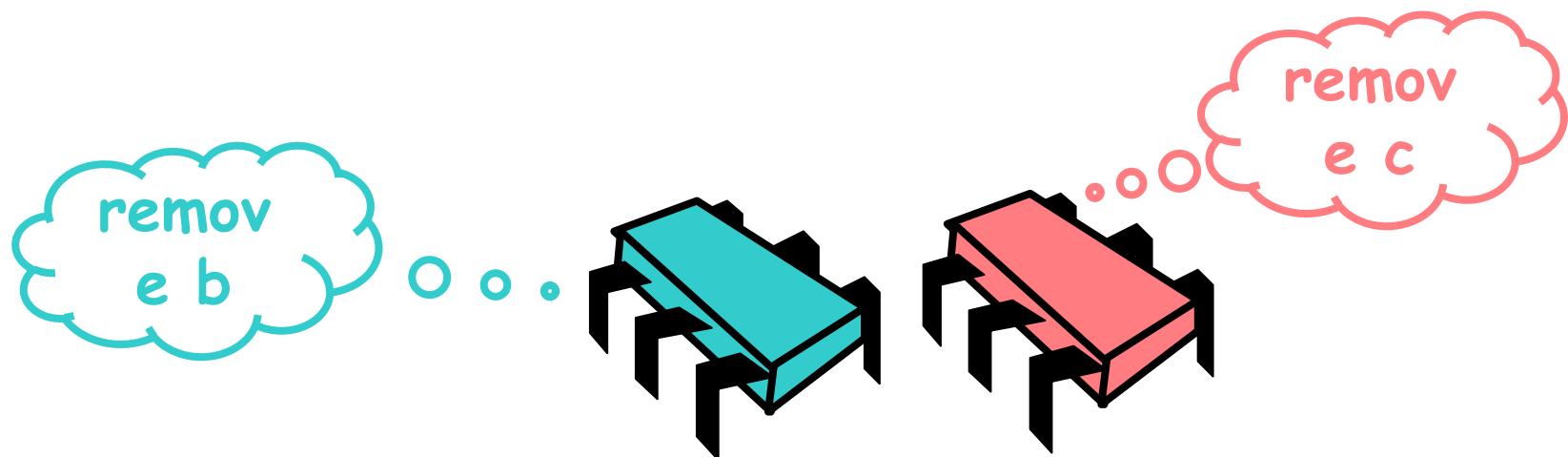
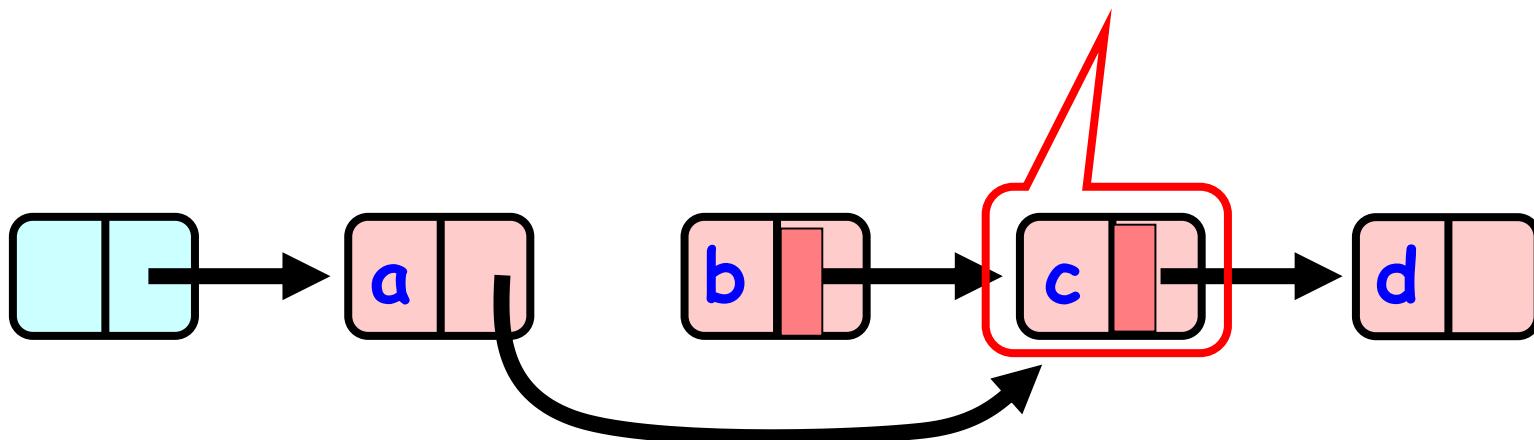


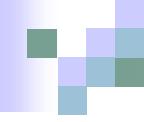
# Removing a Node



# Removing a Node

But c is still in  
the list...  
Is it?

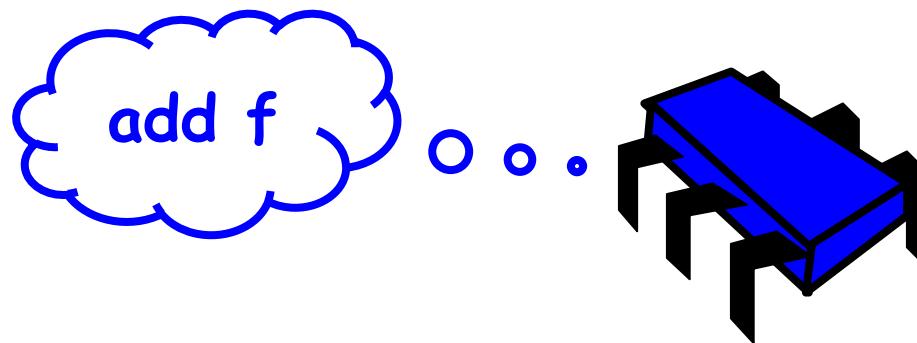




# Non-blocking synchronization

- The physical removal is shared by all threads calling add() or remove()
  - As each thread traverses the list, it cleans up the list by physically removing any marked nodes it encounters
- Contains does not remove any nodes but traverses all nodes whether they are marked or not

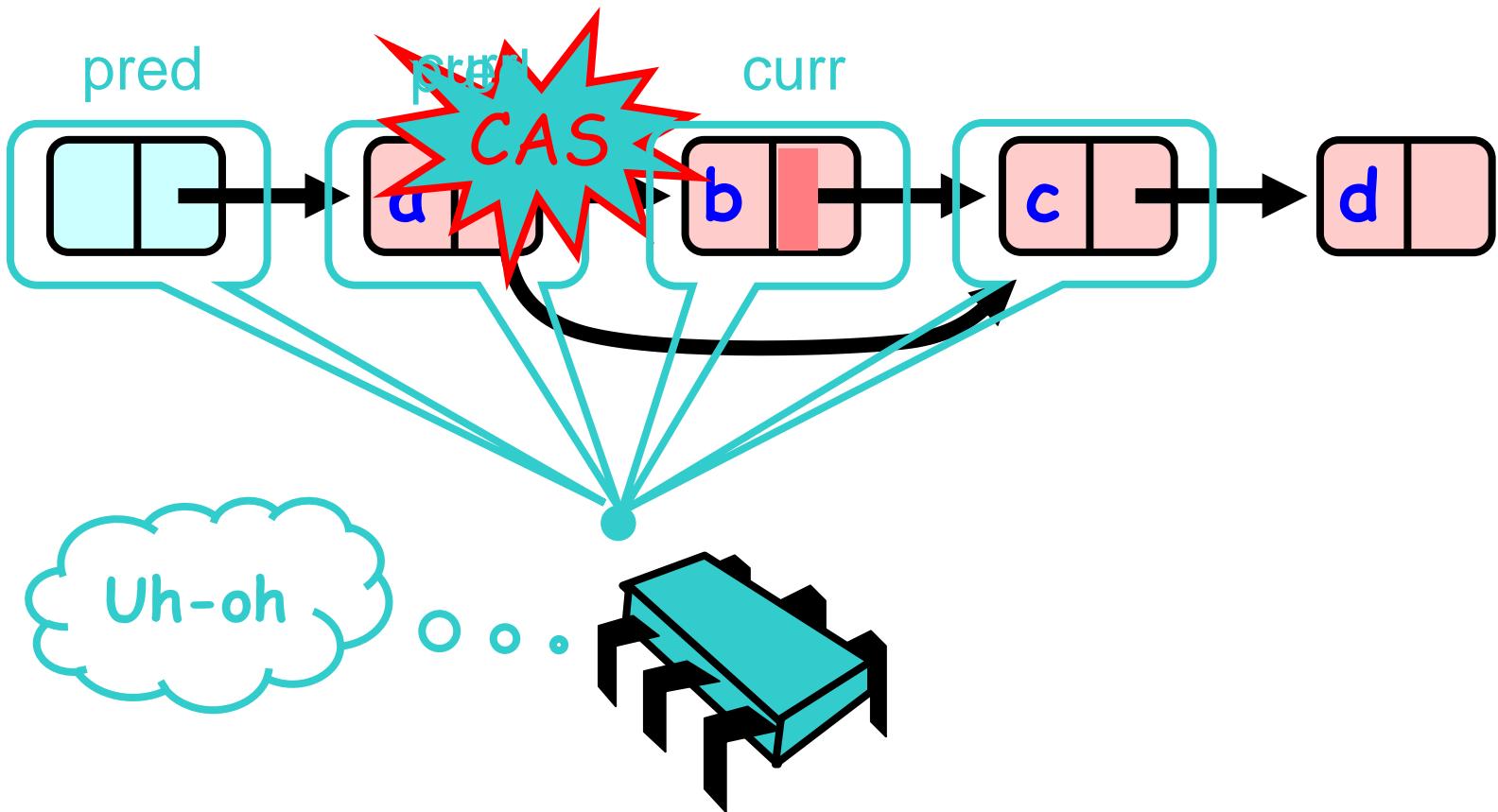
# Removing a Node

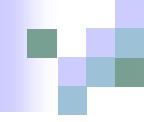


# Removing a Node



# Lock-Free Traversal (only Add and Remove)





# Non-blocking synchronization

- Why can other threads not simply traverse the list without removing marked nodes?
- Why are nodes not removed directly after marking them?

# The Window find() method

```
public window find(Node head, int key)
{
    boolean[] marked = {false};
    boolean snip;

    retry: while (true) {
        pred = head;
        curr = pred.next.getReference();
        while (true) {
            succ = curr.next.get(marked);
            while (marked[0]) {
                ...
            }
        }
    }
}
```

# The Window find() method

```
    ...
    snip = pred.next.CAS(curr, succ,
false, false);
if (!snip)
    continue retry;
curr = succ;
succ = curr.next.get(marked);
}
if (curr.key >= key)
    return new window(pred, curr);
pred = curr;
curr = succ;
}
```

# add() method

```
public boolean add(T item) {  
    int key = item.hashCode();  
  
    while (true) {  
        Window window = find(head, key);  
        Node pred = window.pred;  
        Node curr = window.curr;  
        ...  
    }  
}
```

# add() method

```
public boolean add(T item) {  
    int key = item.hashCode();  
  
    while (true) {  
        Window window = find(head, key);  
        Node pred = window.pred;  
        Node curr = window.curr;  
  
        ...  
    }  
}
```

Traverses list and  
physical remove items till  
position found

# add() method

```
if (curr.key == key)
    return false;
else {
    Node node = new Node(item);
    node.next = new
AtomicMarkableReference(curr, false);

    if (pred.next.compareAndSet(curr,
node, false, false))
        return true;
}
```

# add() method

```
if (curr.key == key)
    return false;
else {
    Node node = new Node(item);
    node.next = new
AtomicMarkableReference(curr, false);

    if (pred.next.compareAndSet(curr,
node, false, false))
        return true;
} }
```

Item is already in list

# add() method

```
if (curr.key == key)
    return false;
else {
    Node node = new Node(item);
    node.next = new
AtomicMarkableReference(curr, false);

    if (pred.next.compareAndSet(curr,
node, false, false))
        return true;
}
```

Create new node

# add() method

```
if (curr.key == key)
    return false;
else {
    Node node = new Node(item);
    node.next = new
AtomicMarkableReference(curr, false);

    if (pred.next.compareAndSet(curr,
node, false, false))
        return true;
}
```

Add item using CAS

# remove() method

```
public boolean remove(T item) {  
    int key = item.hashCode();  
    boolean snip;  
  
    while (true) {  
        Window window = find(head, key);  
        Node pred = window.pred;  
        Node curr = window.curr;  
        ...
```

# remove() method

```
public boolean remove(T item) {  
    int key = item.hashCode();  
    boolean snip;  
  
    while (true) {  
        Window window = find(head, key);  
        Node pred = window.pred;  
        Node curr = window.curr;  
  
        ...  
    }  
}
```

Traverses list and  
physical remove items till  
position found

# remove() method

```
if (curr.key != key)
    return false;
else {
    Node succ =
        curr.next.getReference();
    snip = curr.next.attemptMark(succ,
        true);

    if (!snip) continue;
    pred.next.compareAndSet(curr,
        succ, false, false);
    return true;
}
```

# remove() method

```
if (curr.key != key)
    return false;
else {
    Node succ =
        curr.next.getReference();
    snip = curr.next.attemptMark(succ,
        true);

    if (!snip) continue;
    pred.next.compareAndSet(curr,
        succ, false, false);
    return true;
}
```

Item is not in list

# remove() method

```
if (curr.key != key)
    return false;
else {
    Node succ =
        curr.next.getReference();
    snip = curr.next.attemptMark(succ,
        true);

    if (!snip) continue;
    pred.next.compareAndSet(curr,
        succ, false, false);
    return true;
}
```

Mark item as removed with CAS

# remove() method

```
if (curr.key != key)
    return false;
else {
    Node succ =
        curr.next.getReference();
    snip = curr.next.attemptMark(succ,
        true);

    if (!snip) continue;
    pred.next.compareAndSet(curr,
        succ, false, false);
    return true; Try again if CAS failed
}
```

# remove() method

```
if (curr.key != key)
    return false; Attempt physical removal
else {
    Node succ =
        curr.next.getReference();
    snip = curr.next.attemptMark(succ,
        true);

    if (!snip) continue;
    pred.next.compareAndSet(curr,
        succ, false, false);
    return true;
}
```

# Performance

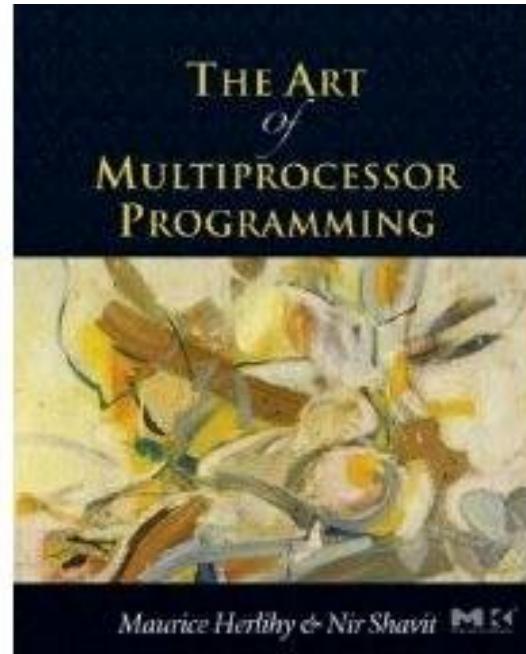
- Non-blocking synchronization guarantees progress in the face of arbitrary delays
- At what cost?
  - Support of atomic modification of a reference and a boolean mark has added performance cost
  - As add() and remove() traverse the list they have to do additional cleanup

# COS 226

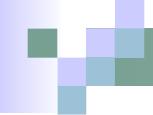
## Chapter 10

### Concurrent Queues and the ABA Problem

# Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



# Pools

- Similar to Set class of Chapter 9 except:
  - Pool object does not necessarily have a contains() method
  - Same item can appear more than once



# Bounded vs Unbounded

- Bounded
  - Fixed capacity
  - Good when resources an issue
- Unbounded
  - Holds any number of objects



# Total, partial or synchronous

## ■ Total:

- A method is total if calls do not wait for conditions to become true
- For example:
  - An attempt to dequeue from an empty queue throws an exception or failure code



# Total, partial or synchronous

- Partial:
  - A method may wait for conditions to hold
  - For example:
    - Thread that tries to dequeue from an empty queue blocks (suspends) until item becomes available



# Total, partial or synchronous

- Synchronous:
  - A method is synchronous if it waits for another method to overlap its call interval
  - For example:
    - A method call that adds an item to the pool is blocked until that item is removed by another method call
    - Used by CSP and Ada for threads to rendezvous and exchange values

# Blocking vs Non-Blocking

- Problem cases:
  - Removing from empty pool
  - Adding to full (bounded) pool
- Blocking
  - Caller waits until state changes
- Non-Blocking
  - Method throws exception



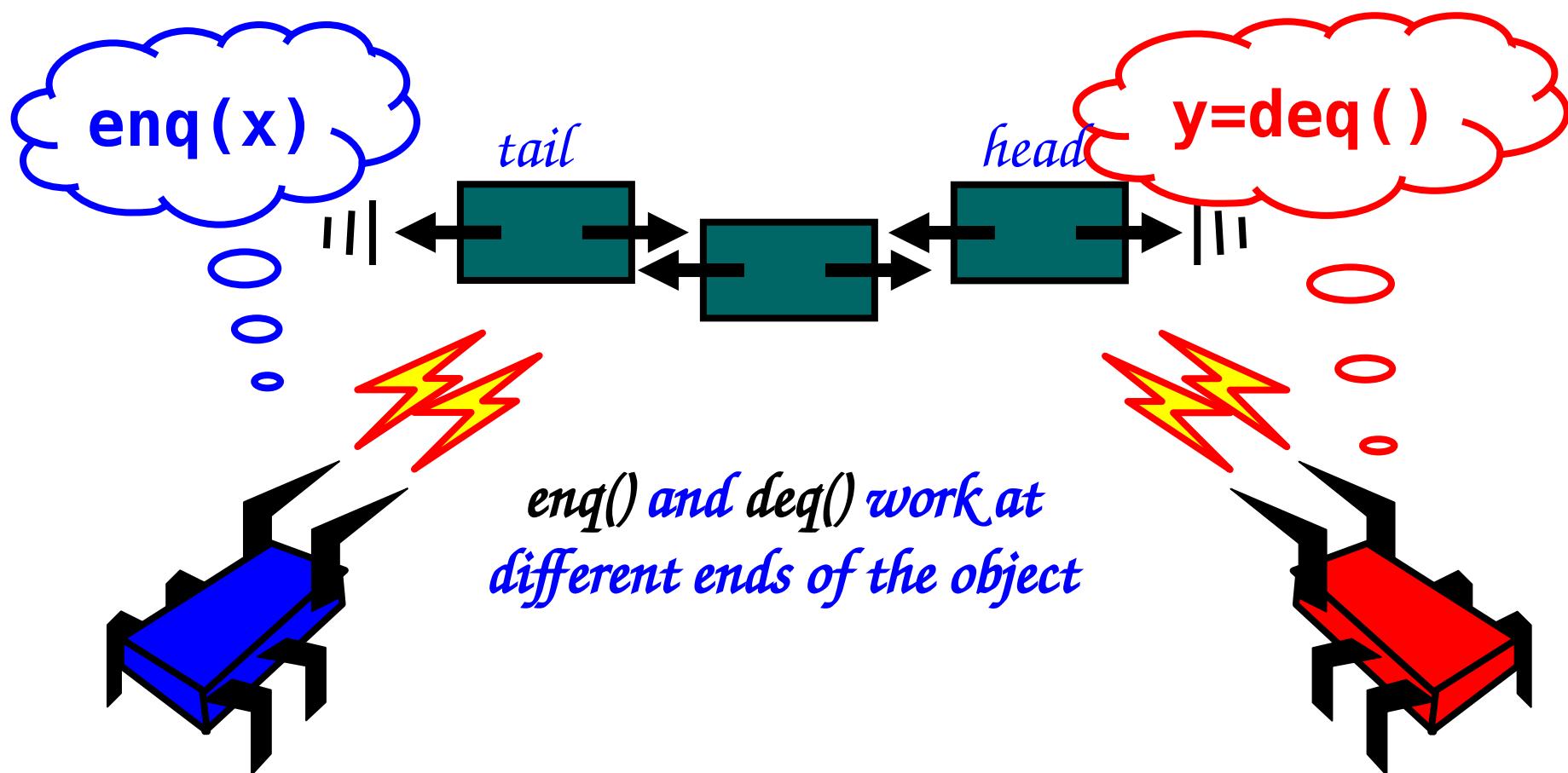
# Bounded Queues

- How much concurrency can we expect a bounded queue implementation with multiple concurrent enqueueers and dequeuers to have?

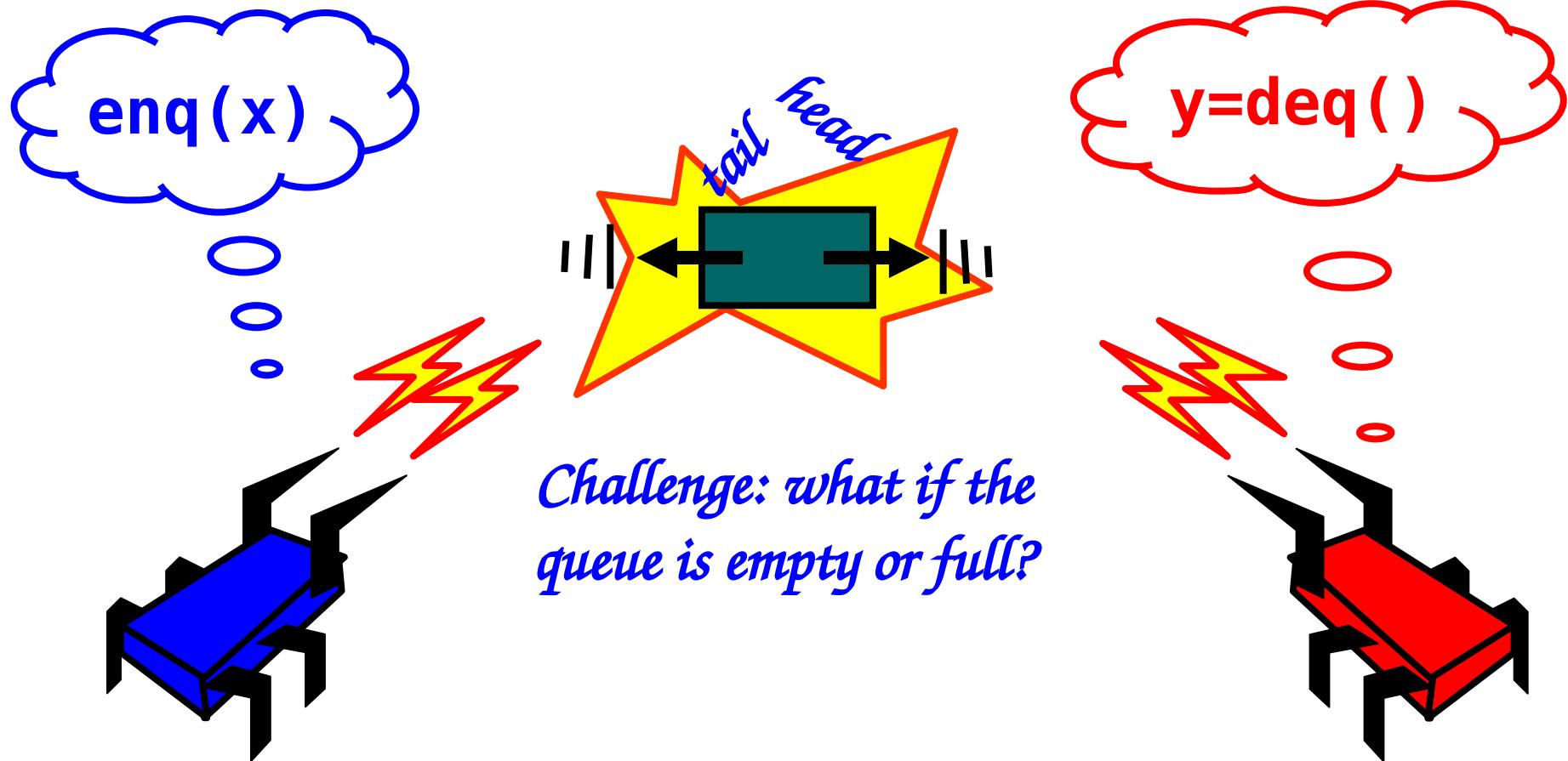
# Bounded Queues

- Informally:
  - enq() and deq() methods operate on opposite ends of the queue
  - So an enq() call and a deq() call should be able to proceed concurrently without interference
  - Concurrent deq() calls and concurrent enq() calls will however interfere

# Queue: Concurrency



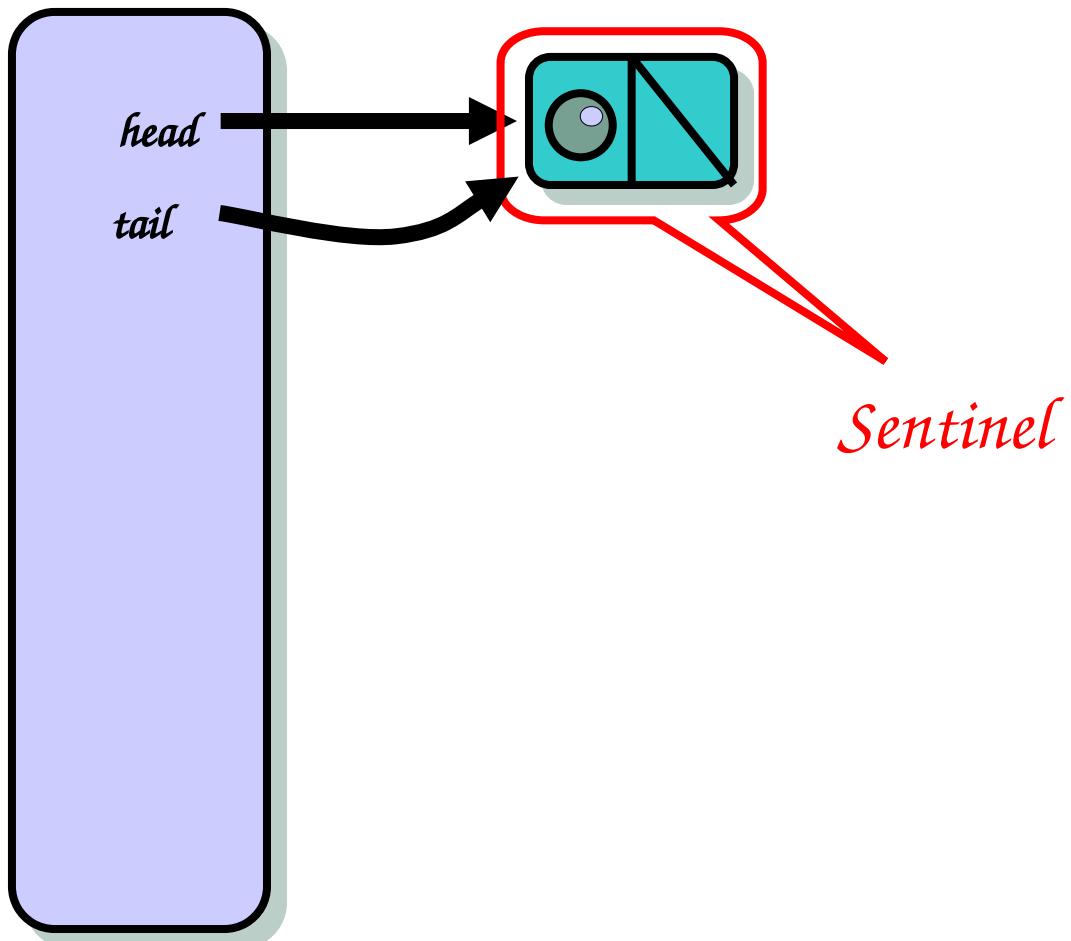
# Concurrency



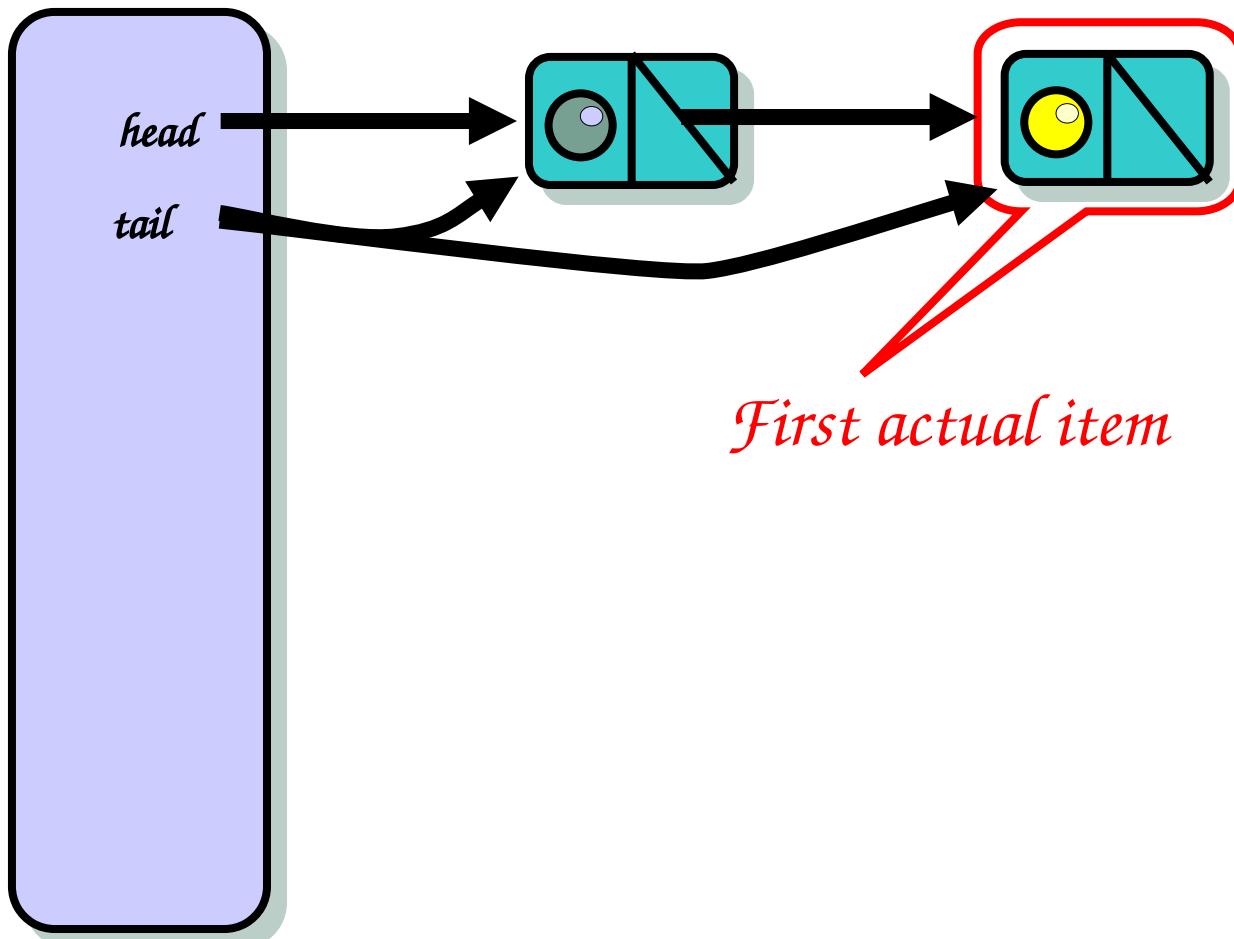
# Bounded lock-based queue

- Implement queue as linked list
- Queue has head and tail fields that refer to first and last nodes in list
- The queue contains a sentinel node – a place-holder

# Bounded Queue



# Bounded Queue





# Bounded Queue

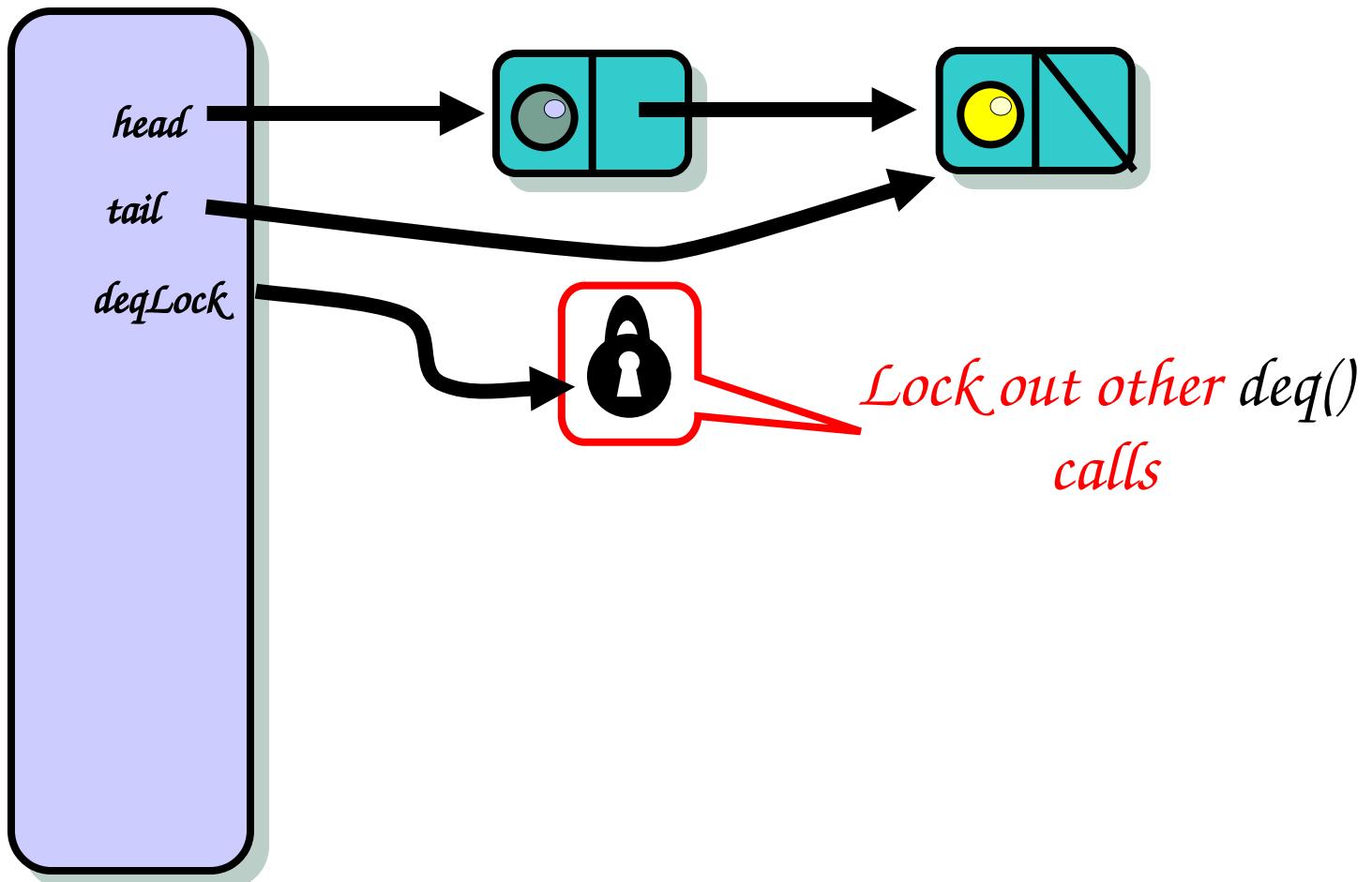
- Two distinct locks are used:
  - `deqlock` – to lock the front of the queue to ensure that there is not more than one dequeuer at a time
  - `enqlock` – to lock the end of the queue to ensure that there is not more than one enqueueuer at a time



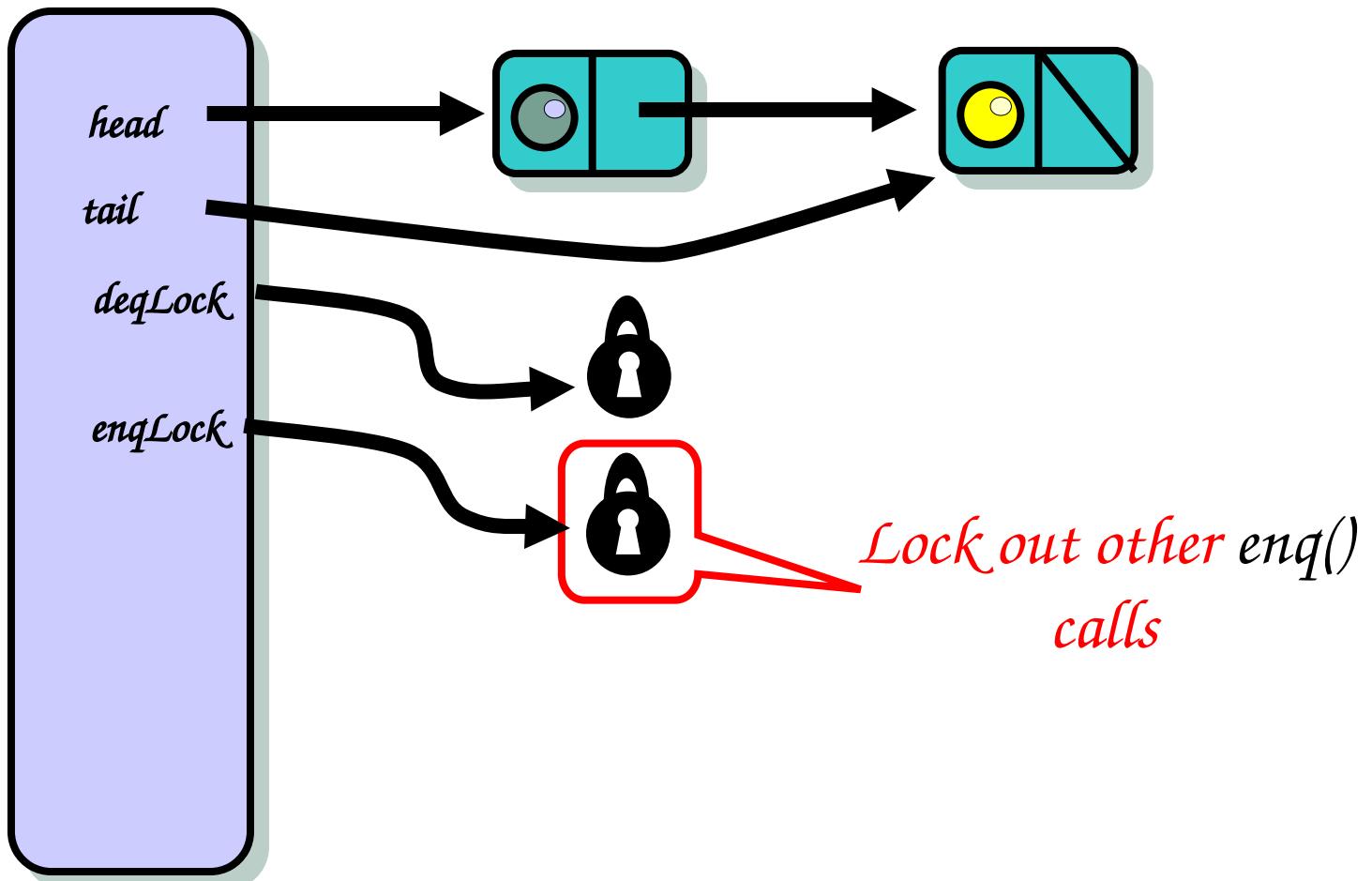
# Bounded Queue

- Using two locks instead of one means that enqueueers and dequeuers can operate independently and does not lock one another out unnecessarily

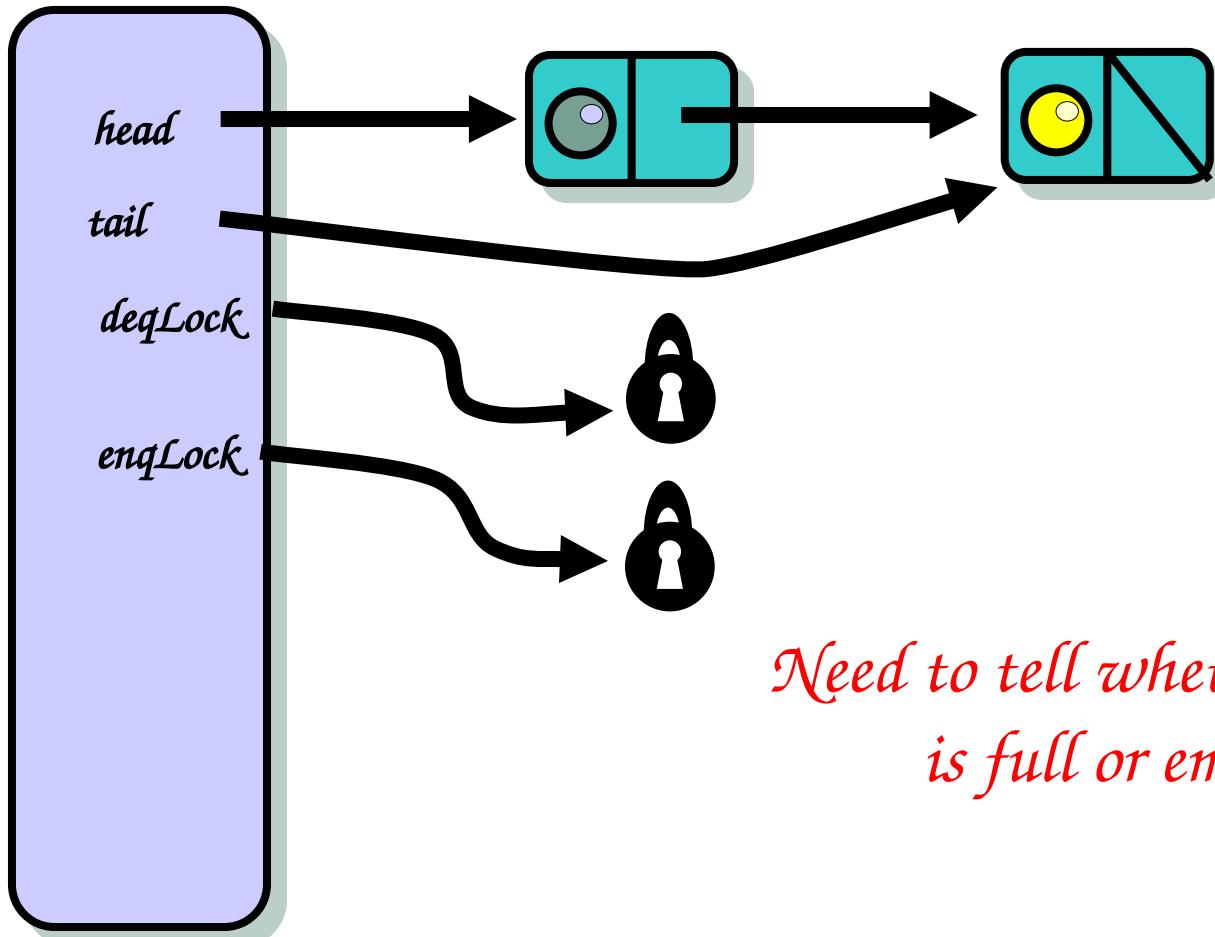
# Bounded Queue



# Bounded Queue



# Not Done Yet





# Bounded Queues

- `enq()` calls cannot continue while list is full
- `deq()` calls cannot continue while list is empty

# Bounded Queue

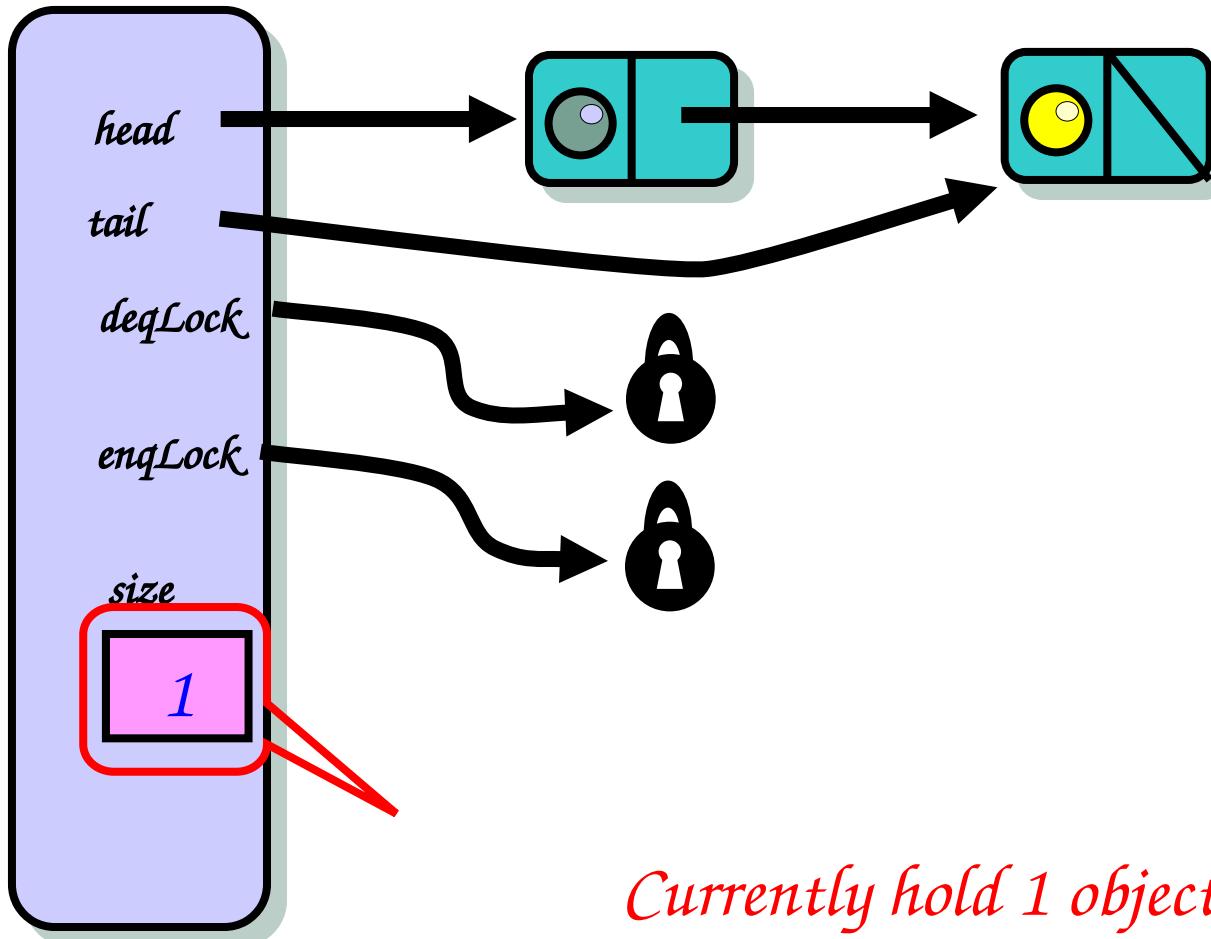
- Each lock has an associated condition field:
  - enqlock is associated with the notFullCondition
    - Notifies enqueueurs when list is not full anymore
  - deqlock is associated with the notEmptyCondition
    - Notifies dequeuers when list is not empty anymore



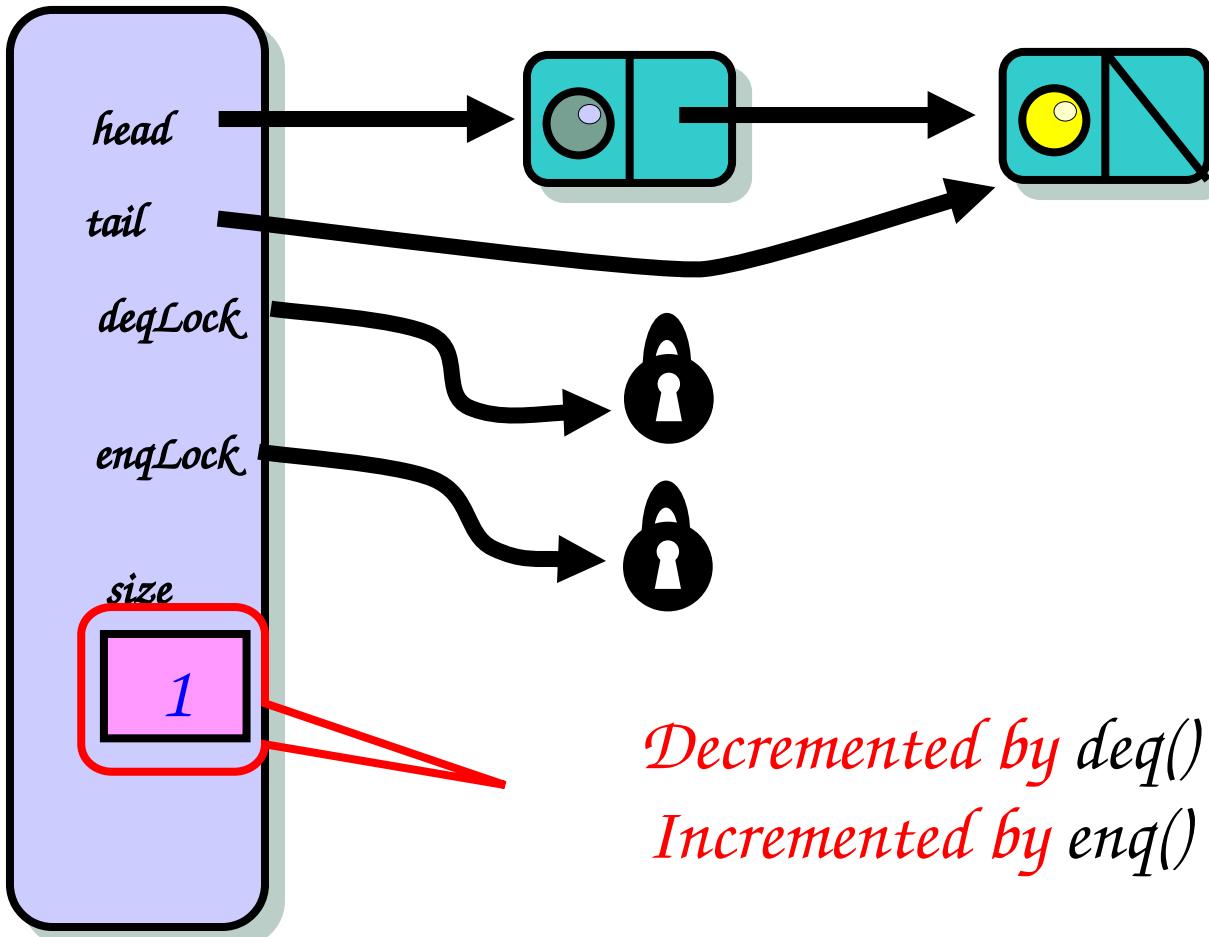
# Bounded Queue

- Since the queue is bounded we must also keep track of the number of empty/available slots
- The size field keeps track of the number of objects currently in the queue
- The size field is an AtomicInteger

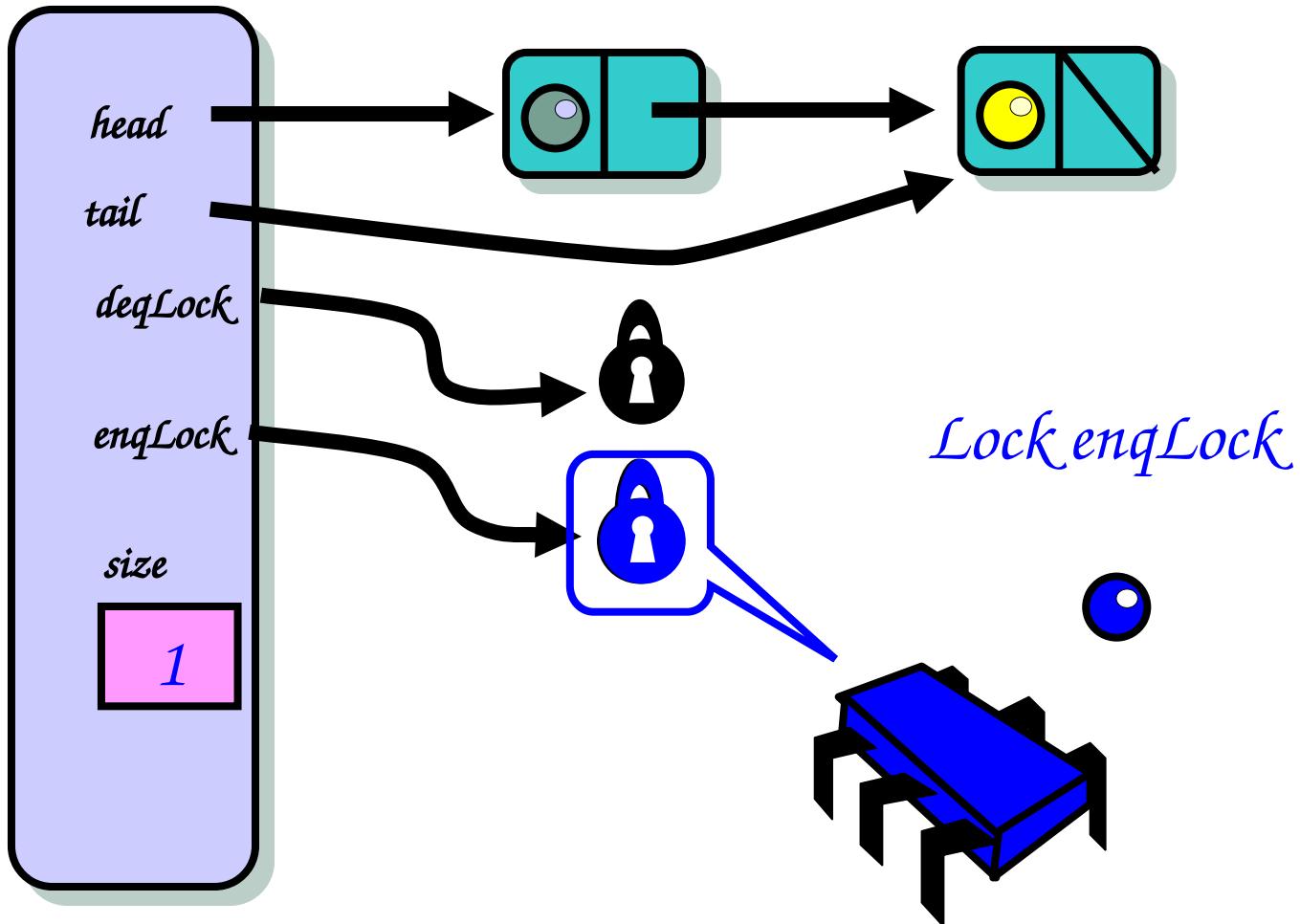
# Not Done Yet



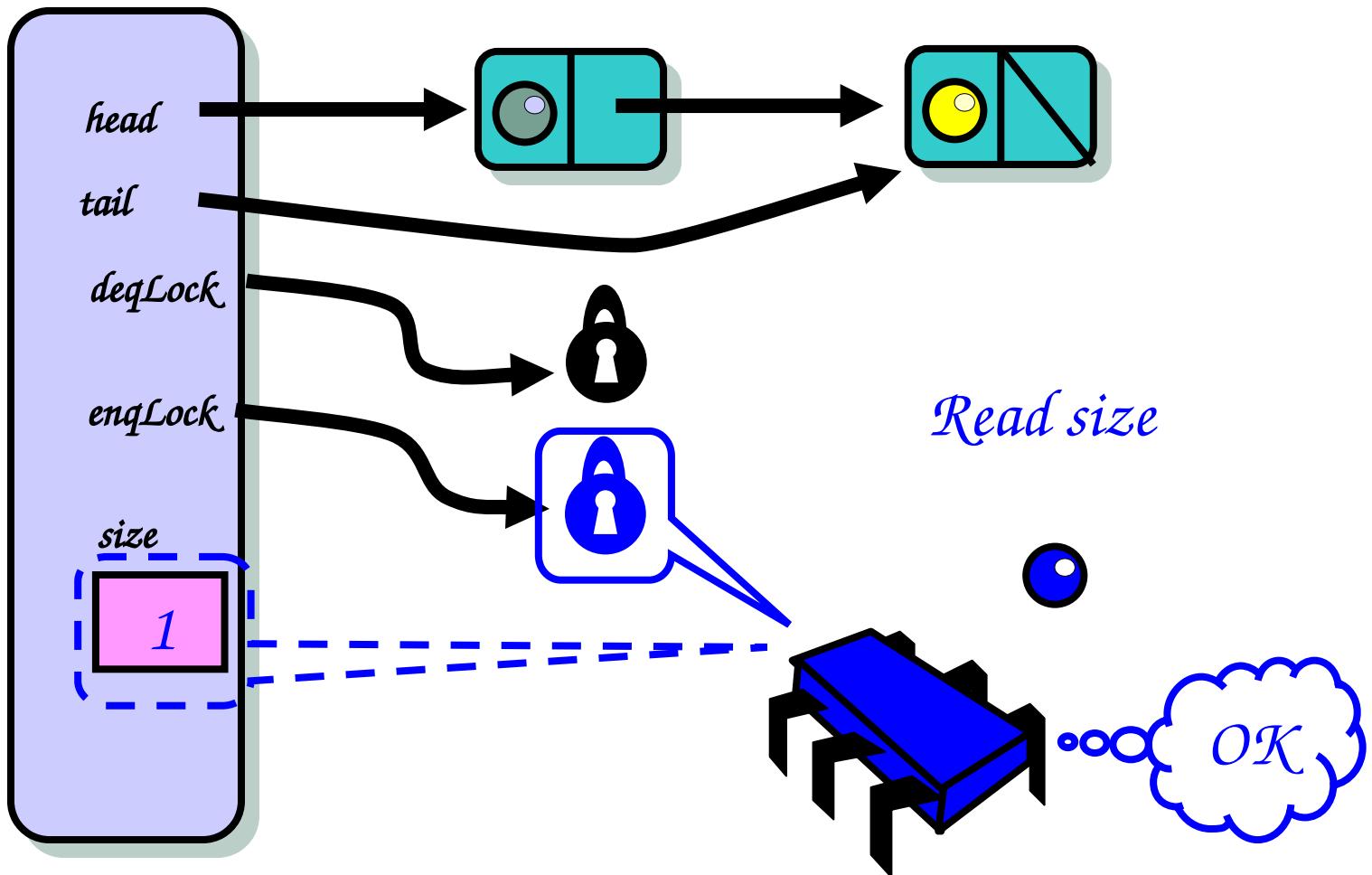
# Not Done Yet



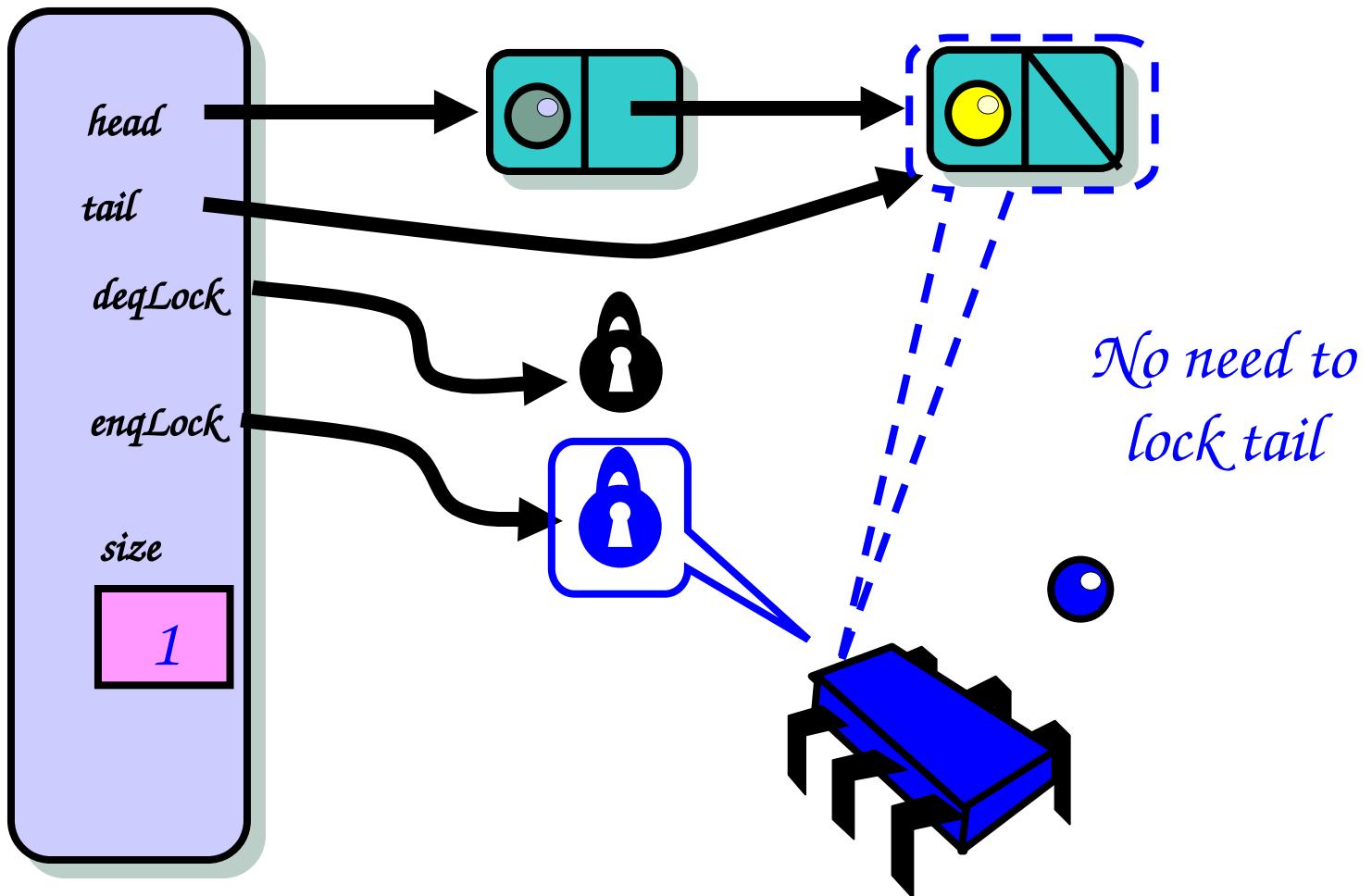
# Enqueuer



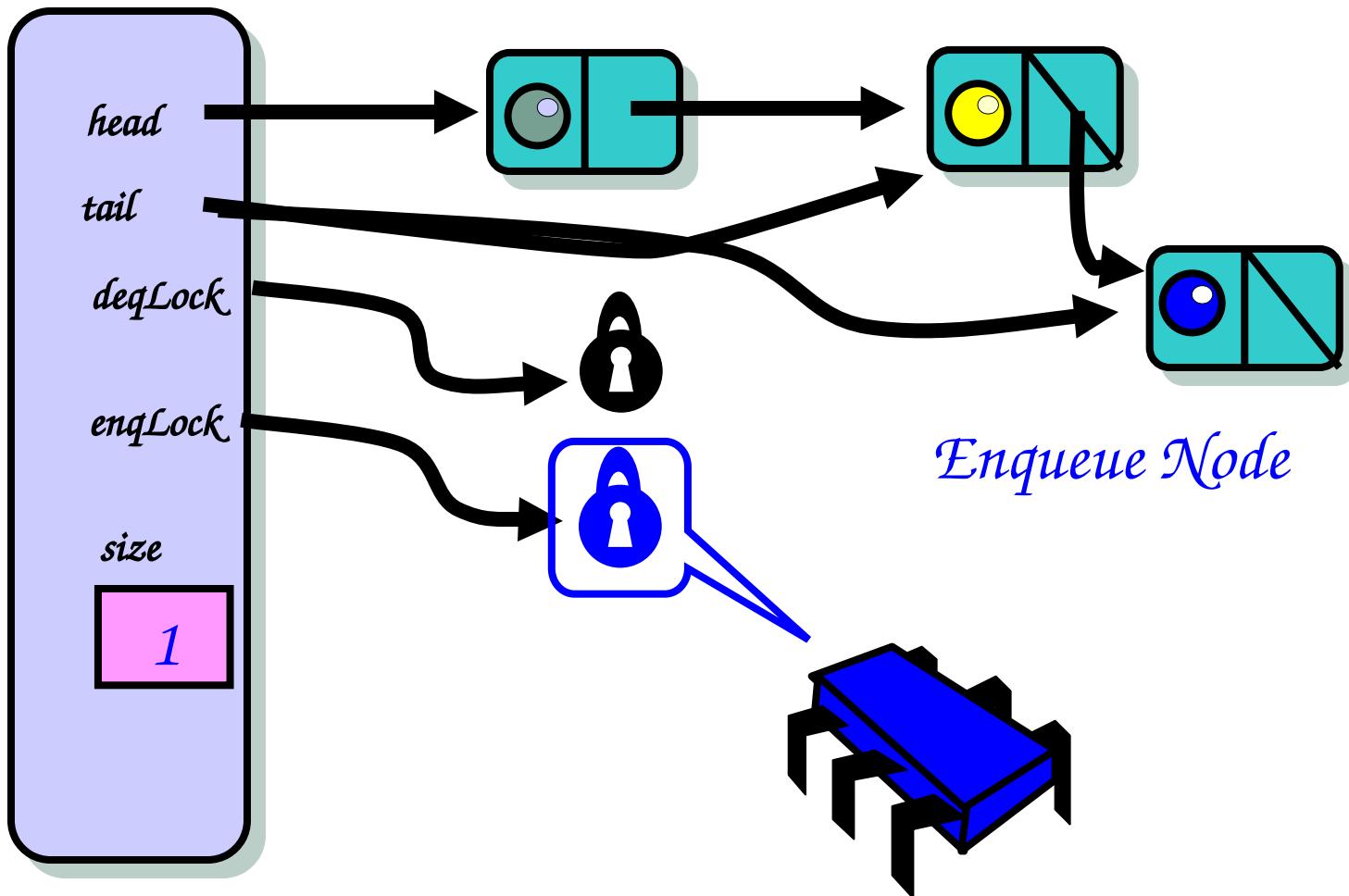
# Enqueuer



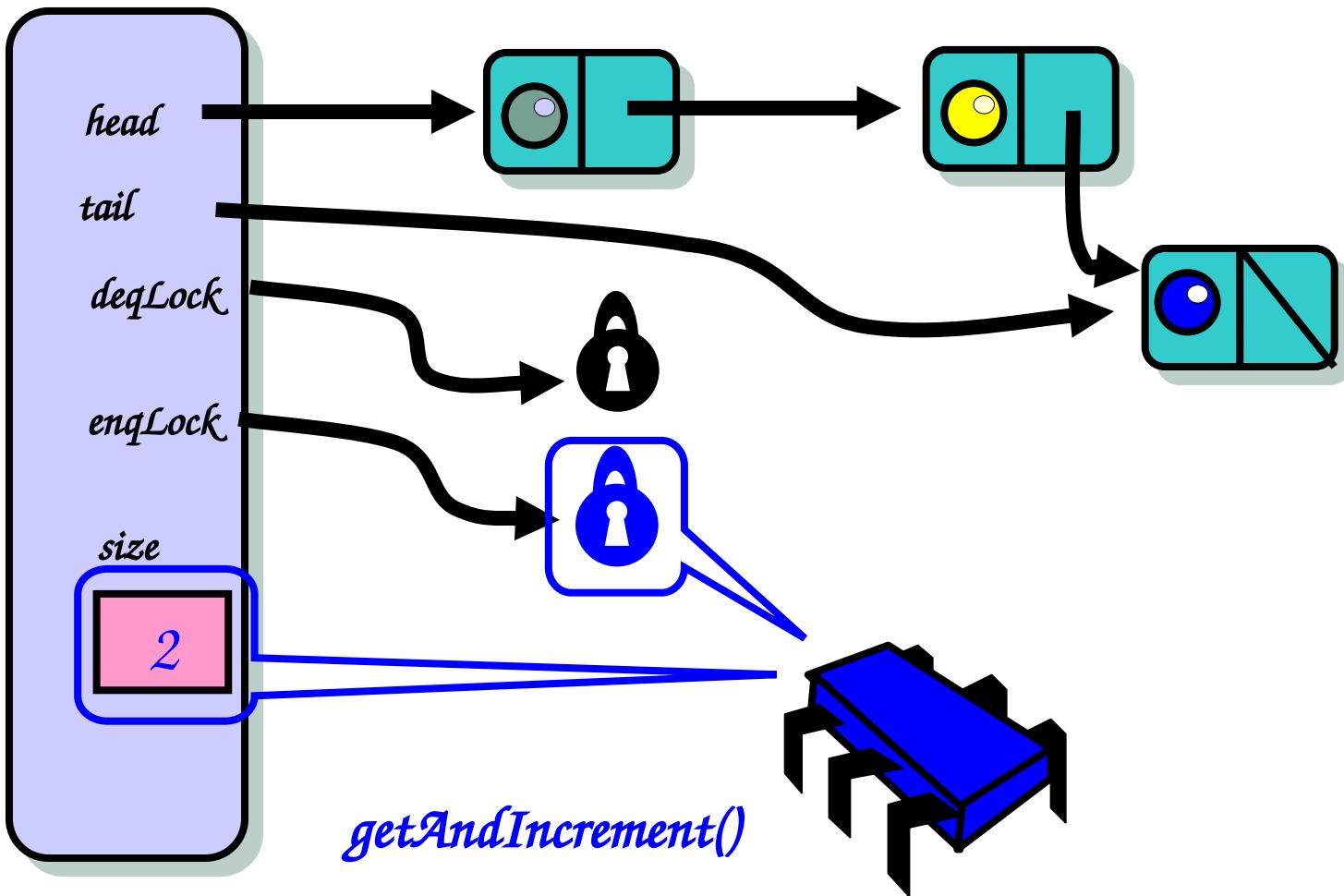
# Enqueuer



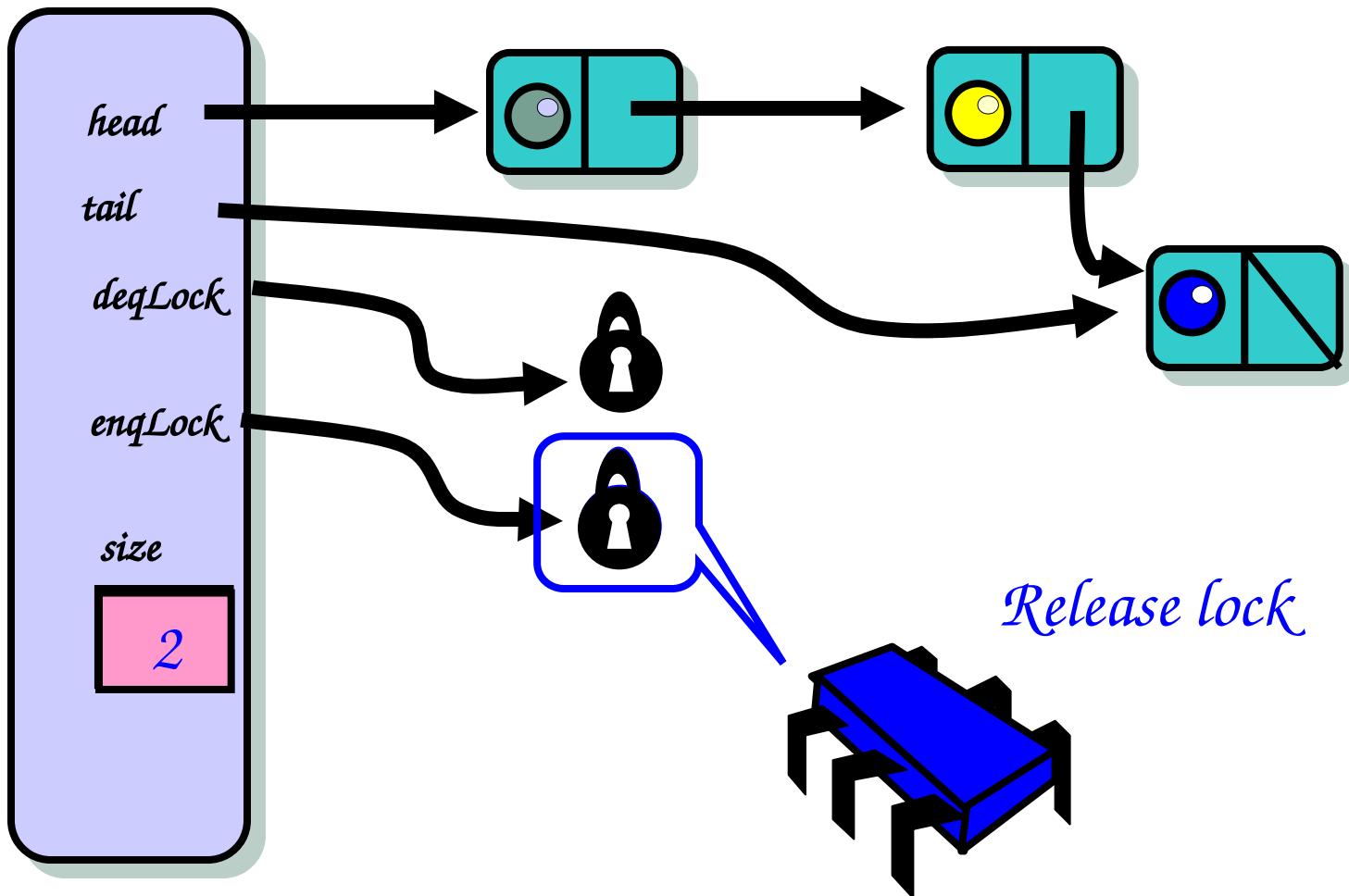
# Enqueuer



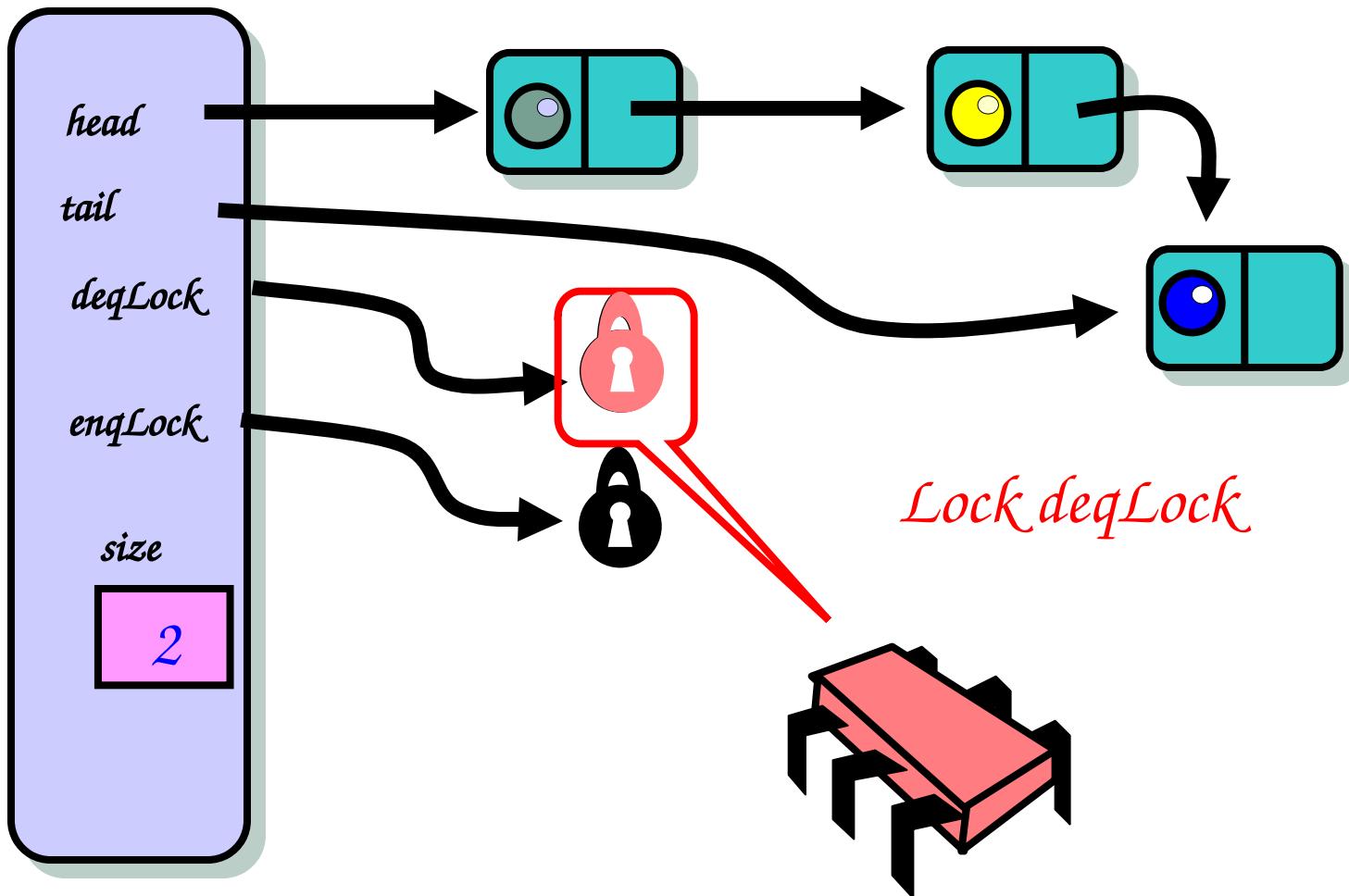
# Enqueuer



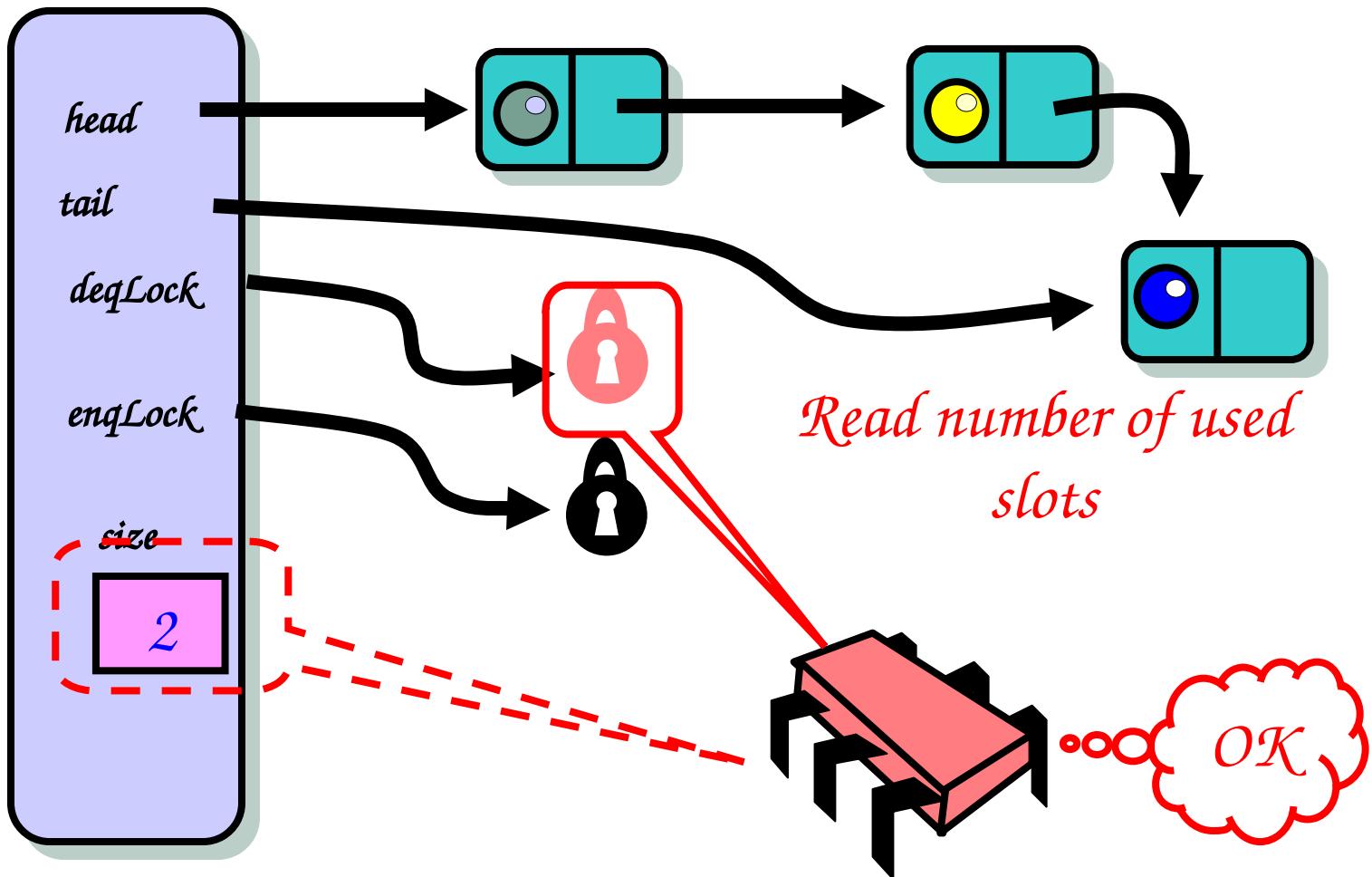
# Enqueuer



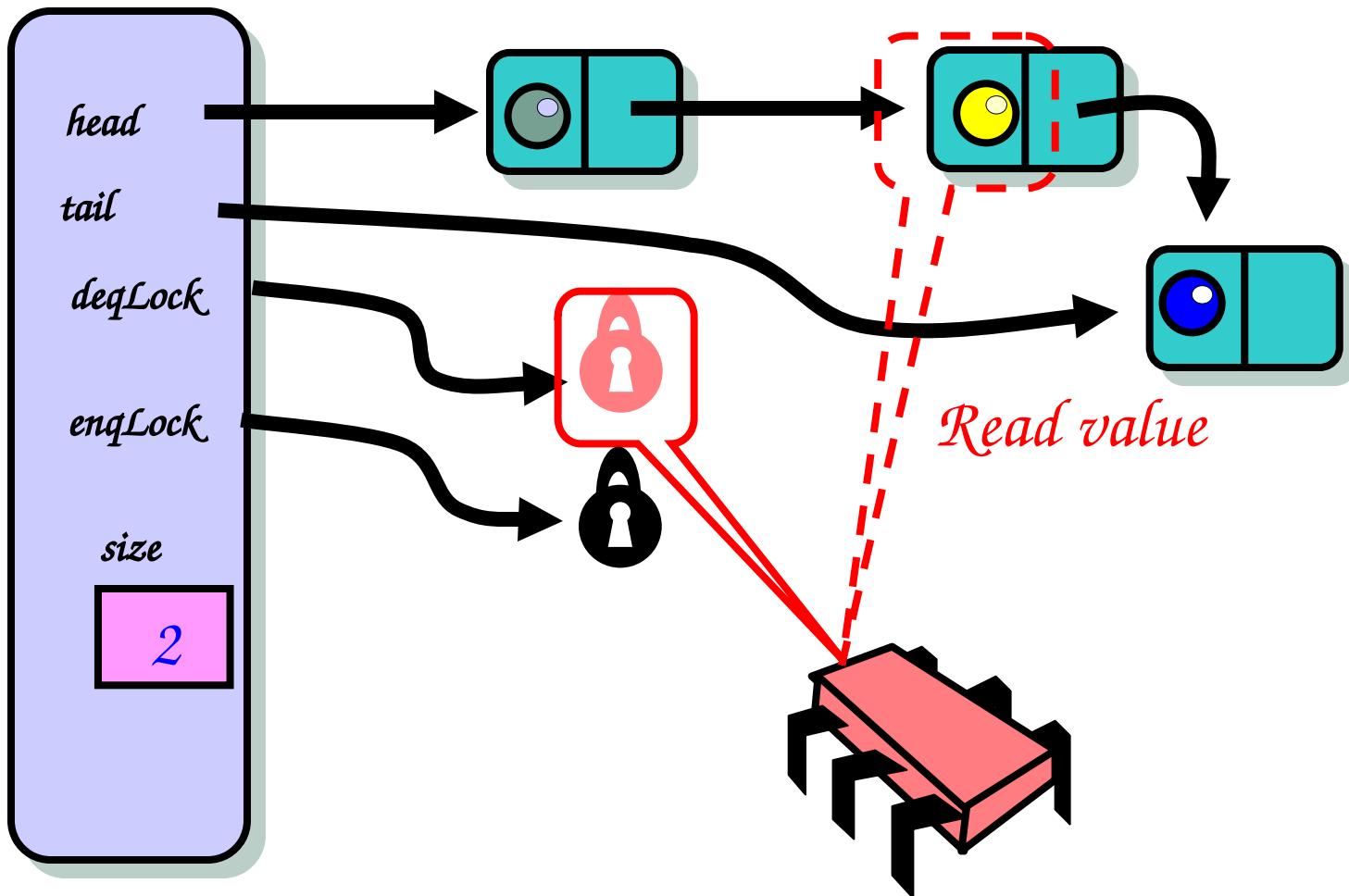
# Dequeuer



# Dequeuer

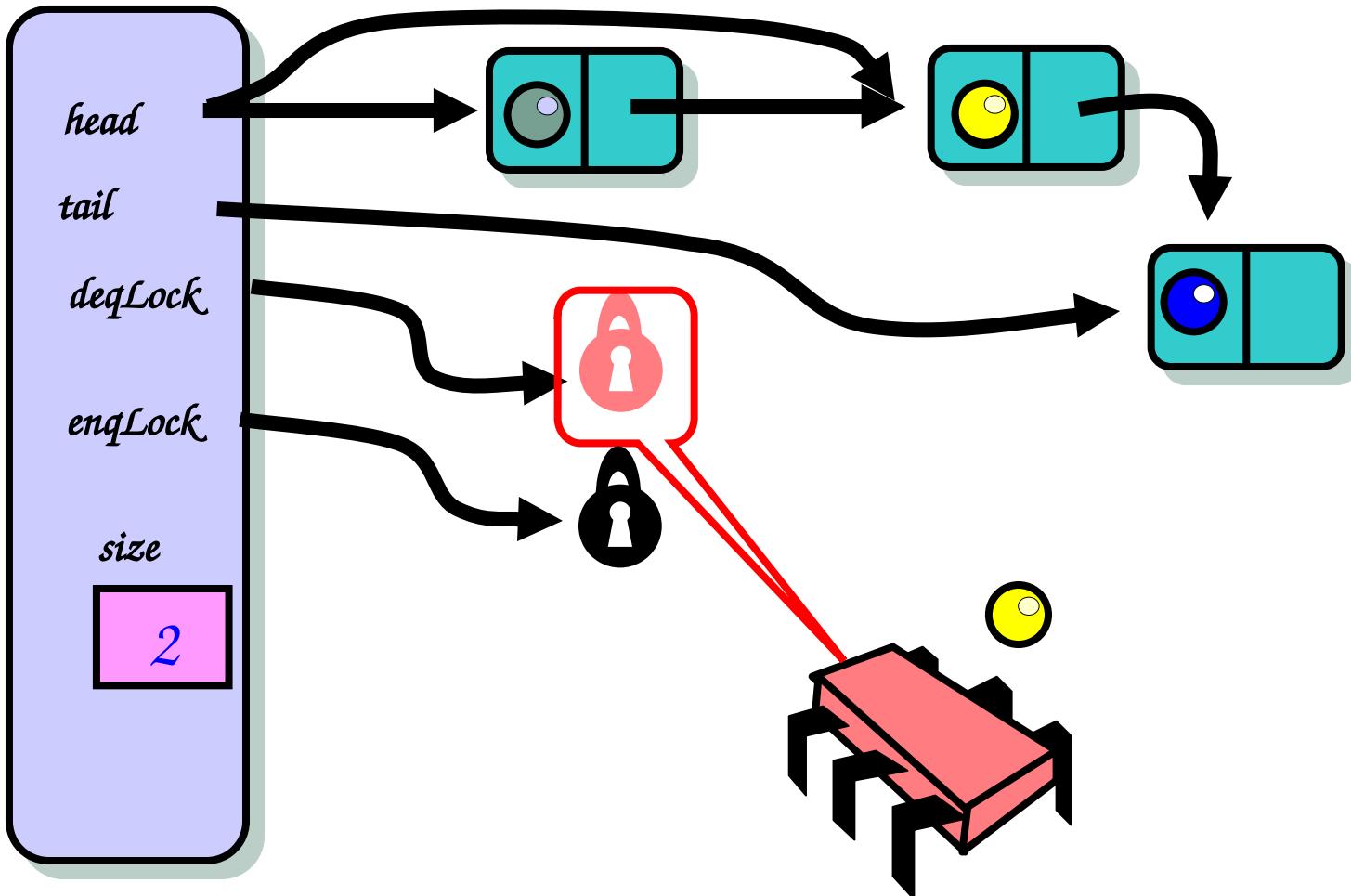


# Dequeuer

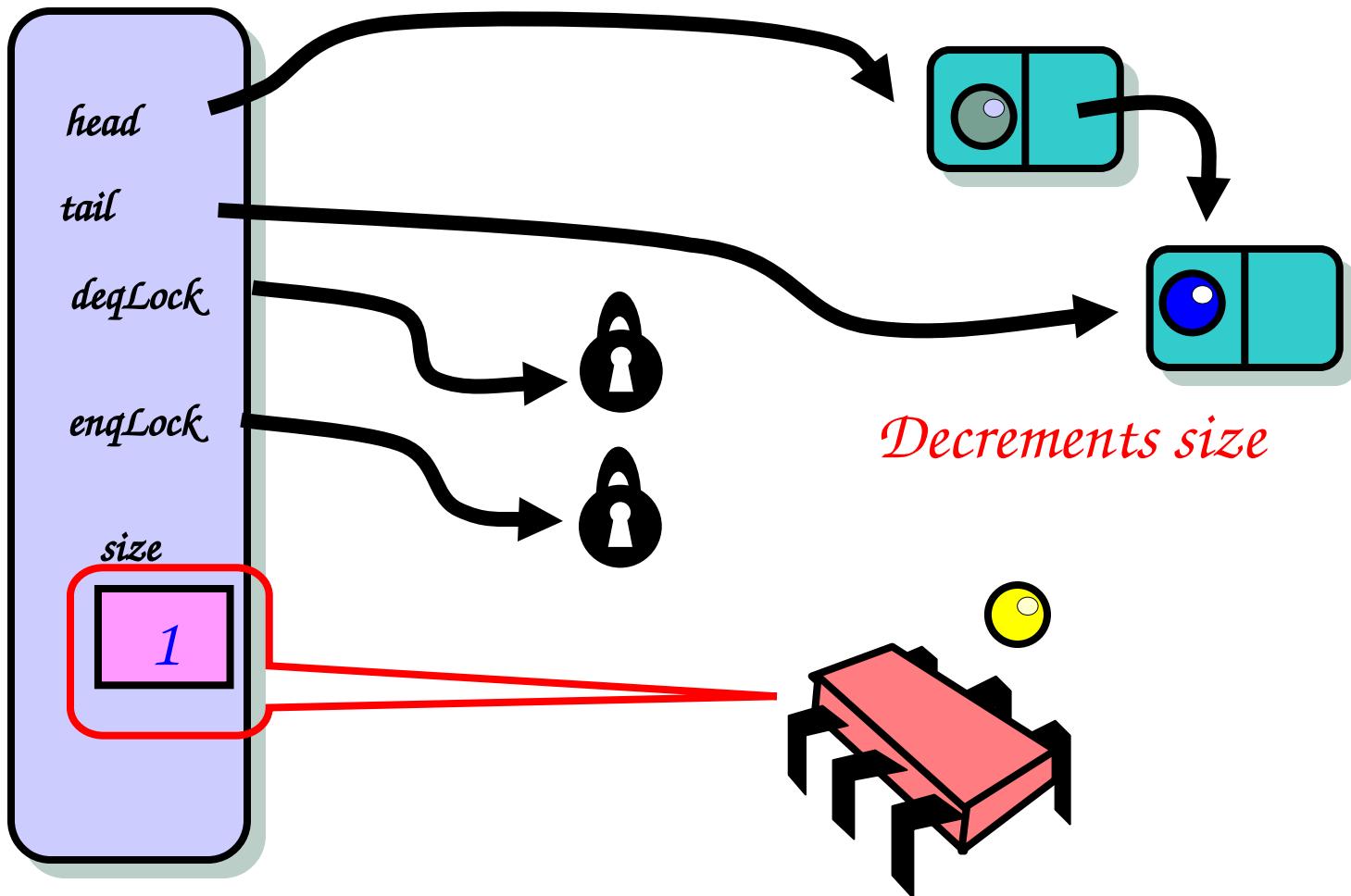


*Make first Node new sentinel*

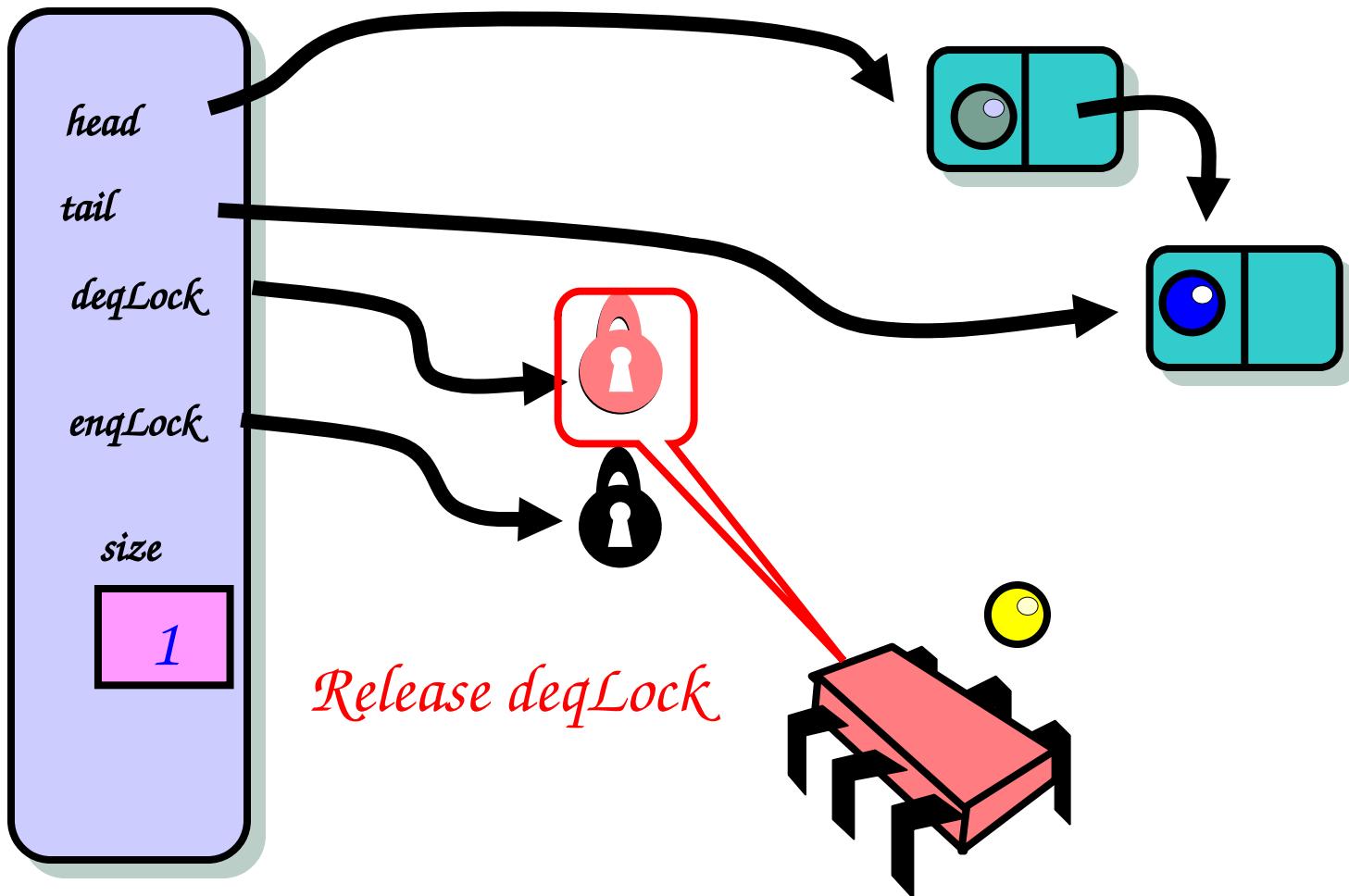
# Dequeuer



# Dequeuer



# Dequeuer



# Bounded Queue

```
public class BoundedQueue<T> {  
    ReentrantLock enqLock, deqLock;  
    Condition notEmptyCondition, notFullCondition;  
    AtomicInteger size;  
    Node head;  
    Node tail;  
    int capacity;  
    enqLock = new ReentrantLock();  
    notFullCondition = enqLock.newCondition();  
    deqLock = new ReentrantLock();  
    notEmptyCondition = deqLock.newCondition();  
}
```

# Bounded Queue

```
public class BoundedQueue<T> {  
    ReentrantLock enqLock, deqLock;  
    Condition notEmptyCondition, notFullCondition;  
    AtomicInteger size;  
    Node head;  
    Node tail;  
    int capacity;  
    enqLock = new ReentrantLock();  
    notFullCondition = enqLock.newCondition();  
    deqLock = new ReentrantLock();  
    notEmptyCondition = deqLock.newCondition();  
}
```

*Enq & deq locks*

# Bounded Queue

```
public class BoundedQueue<T> {  
    ReentrantLock enqLock, dequeLock;  
    Condition notEmptyCondition, notFullCondition;  
    AtomicInteger size;  
    Node head;  
    Node tail;  
    int capacity;  
    enqLock = new ReentrantLock();  
    notFullCondition = enqLock.newCondition();  
    dequeLock = new ReentrantLock();  
    notEmptyCondition = dequeLock.newCondition();  
}
```

*Conditions*

# Bounded Queue

```
public class BoundedQueue<T> {  
    ReentrantLock enqLock, deqLock;  
    Condition notEmptyCondition, notFullCondition;  
    AtomicInteger size;  
    Node head;  
    Node tail;  
    int capacity;  
    enqLock = new ReentrantLock();  
    notFullCondition = enqLock.newCondition();  
    deqLock = new ReentrantLock();  
    notEmptyCondition = deqLock.newCondition();  
}
```

*Associate conditions with locks*

# Enq Method Part One

```
public void enq(T x) {  
    boolean mustWakeDequeueuers = false;  
    enqLock.lock();  
    try {  
        while (size.get() == capacity)  
            notFullCondition.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = tail.next;  
        if (size.getAndIncrement() == 0)  
            mustWakeDequeueuers = true;  
    } finally {  
        enqLock.unlock();  
    }  
    ...  
}
```

# Enq Method Part One

```
public void enq(T x) {  
    boolean mustWakeDequeueuers = false;  
    enqLock.lock();  
    try {  
        while (size.get() == capacity)  
            notFullCondition.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = tail.next;  
        if (size.getAndIncrement() == 0)  
            mustWakeDequeueuers = true;  
    } finally {  
        enqLock.unlock();  
    }  
    ...  
}
```

*Lock and unlock enq lock*

# Enq Method Part One

```
public void enq(T x) {  
    boolean mustWakeDequeueuers = false;  
    enqLock.lock();  
    try {  
        while (size.get() == capacity)  
            notFullCondition.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = tail.next;  
        if (size.getAndIncrement() == 0)  
            mustWakeDequeueuers = true;  
    } finally {  
        enqLock.unlock();  
    }  
    ...  
}
```

*If queue is full, patiently await further  
instructions ...*

# Enq Method Part One

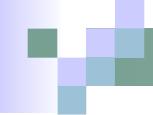
```
public void enq(T x) {  
    boolean mustWakeDequeueuers = false;  
    enqLock.lock();  
    try {  
        while (size.get() == capacity)  
            notFullCondition.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = tail.next;  
        if (size.getAndIncrement() == 0)  
            mustWakeDequeueuers = true;  
    } finally {  
        enqLock.unlock();  
    }  
    ...  
}
```

*Add new node*

# Enq Method Part One

```
public void enq(T x) {  
    boolean mustWakeDequeuers = false;  
    enqLock.lock();  
    try {  
        while (size.get() == capacity)  
            notFullCondition.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = tail.next;  
        if (size.getAndIncrement() == 0)  
            mustWakeDequeuers = true;  
    } finally {  
        enqLock.unlock();  
    }  
    ...  
}
```

*If queue was empty, allows dequeuers to be woken*

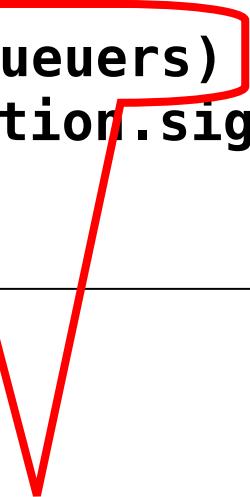


# Enq Method Part Deux

```
public void enq(T x) {  
    ...  
    if (mustWakeDequeueurs)  
        notEmptyCondition.signalAll();  
    }  
}
```

# Enq Method Part Deux

```
public void enq(T x) {  
    ...  
    if (mustWakeDequeueuers) {  
        notEmptyCondition.signalAll();  
    }  
}
```



*Are there dequeuers to be signaled?*

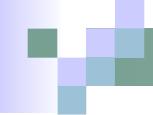
# Enq Method Part Deux

```
public void enq(T x) {  
    ...  
    if (mustWakeDequeueuers) {  
        notEmptyCondition.signalAll();  
    }  
}
```

*Signal dequeuers that queue no longer empty*

# Deq Method Part One

```
public T deq() {  
    T result;  
    boolean mustWakeEnqueuers = false;  
    deqLock.lock();  
    try {  
        while (size.get() == 0)  
            notEmptyCondition.await();  
        result = head.next.value;  
        head = head.next;  
        if (size.getAndDecrement() == capacity)  
            mustWakeEnqueuers = true;  
    } finally {  
        deqLock.unlock();  
    }  
    ...  
}
```



# Deq Method Part Deux

```
public T deq() {  
    ...  
    if (mustWakeEnqueuers) {  
        notFullCondition.signalAll();  
    }  
    return result;  
}
```

# The Enq() & Deq() Methods

- Share no locks
  - That's good
- But do share an atomic counter
  - Accessed on every method call
  - That's not so good
- Can we alleviate this bottleneck?



# Split the Counter into two counters

- enqSideSize
  - Decremented by deq()
- deqSideSize
  - Incremented by enq()

# Two counters

- The enq() method:
  - Checks enqSideSize and proceeds as long as less than capacity
  - Cares only if value is capacity
- The deq() method:
  - Checks deqSideSize and proceeds as long as greater than zero
  - Cares only if value is zero



# Unbounded Queues

- Queue can hold unbounded number of items:
  - enq() always enqueues its item
  - deq() throws an EmptyException if there is no item to dequeue
- Same representation as bounded queue, only no need to count number of items

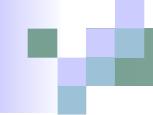
# Unbounded queue enq()

```
public void enq(T x) {  
    enqLock.lock();  
    try {  
        Node e = new Node(x);  
        tail.next = e;  
        tail = tail.next;  
    } finally {  
        enqLock.unlock();  
    }  
}
```

# Unbounded queue deq()

```
public T deq() throws EmptyException {  
    T result;  
    deqLock.lock();  
    try {  
        if (head.next == null)  
            throw new EmptyException();  
        result = head.next.value;  
        head = head.next;  
    } finally {  
        deqLock.unlock();  
    }  
    return result;  
}
```

*Only need to check that queue  
is not empty*



# Now an unbounded lock-free queue

- Natural extension of the unbounded queue
- However it prevents method calls from starving by having quicker threads help slower threads



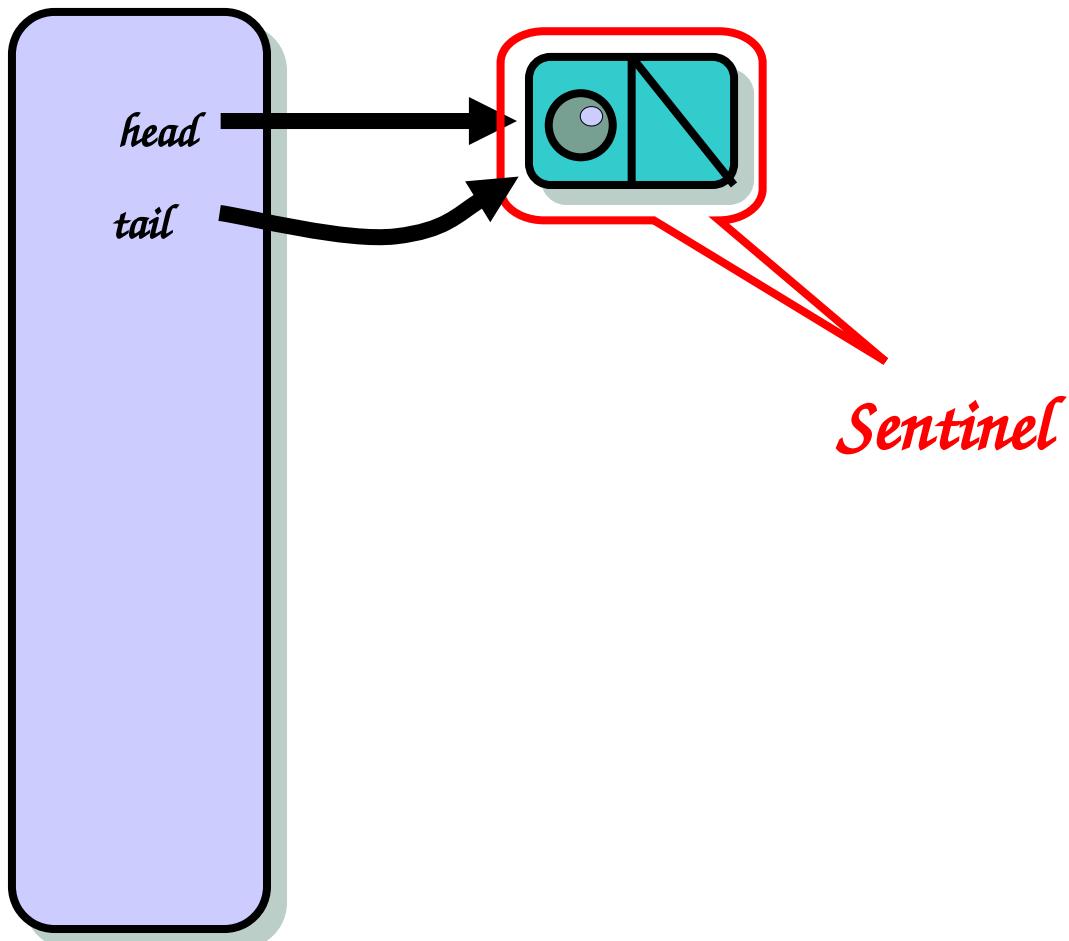
# Unbounded Lock-free Queue

- Again represented as a Linked List
- However each node's next field is now an AtomicReference (to facilitate lock-free operations)

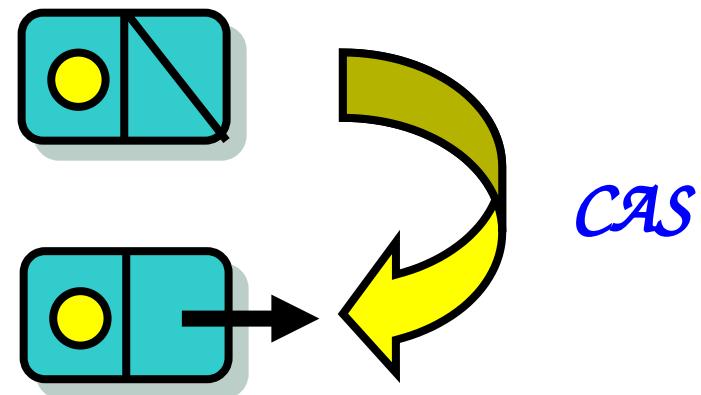
# Lock-free queue Node class

```
public class Node {  
    public T value;  
    public AtomicReference<Node> next;  
  
    public Node(T value) {  
        value = value;  
        next = new AtomicReference<Node>(null);  
    }  
}
```

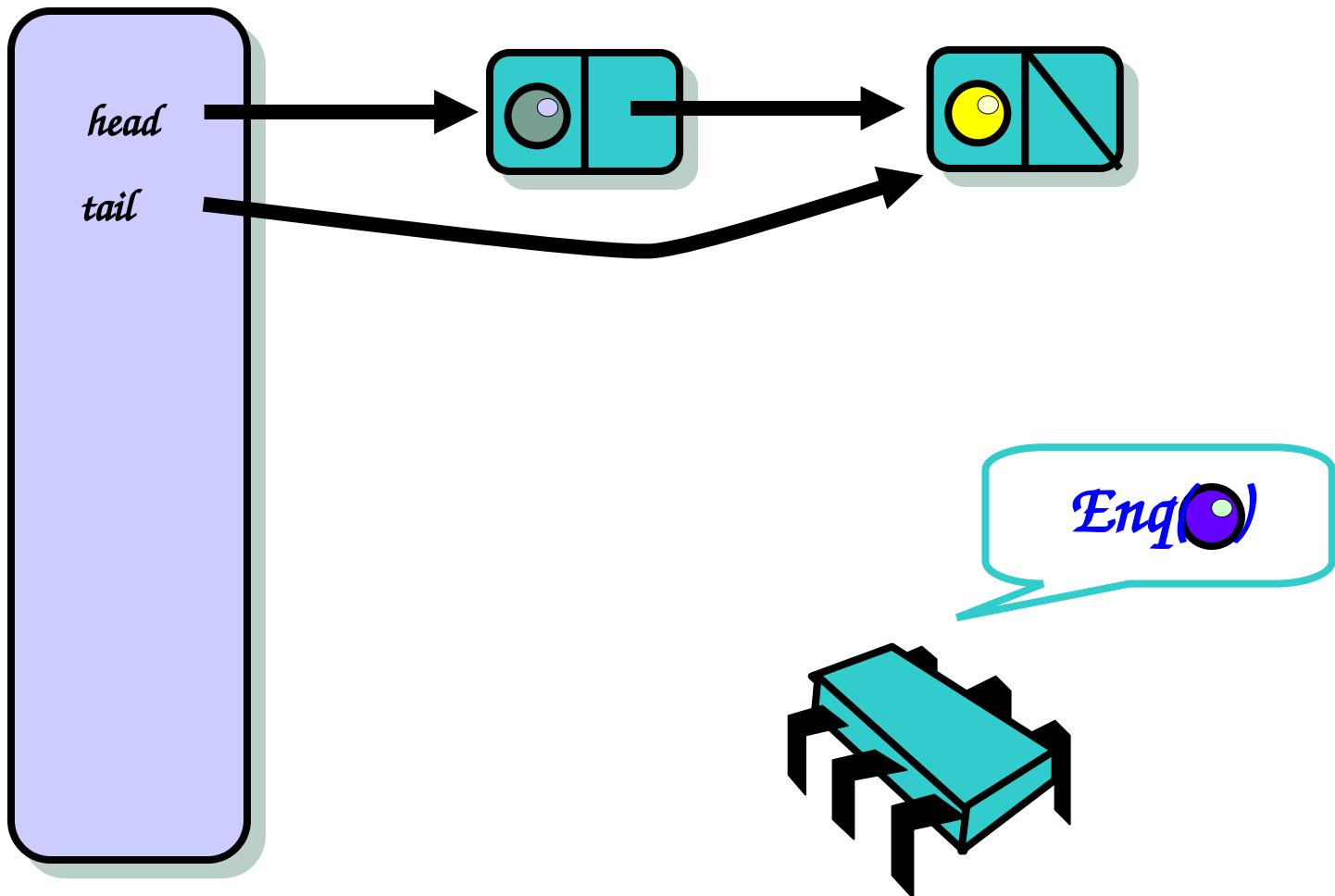
# A Lock-Free Queue



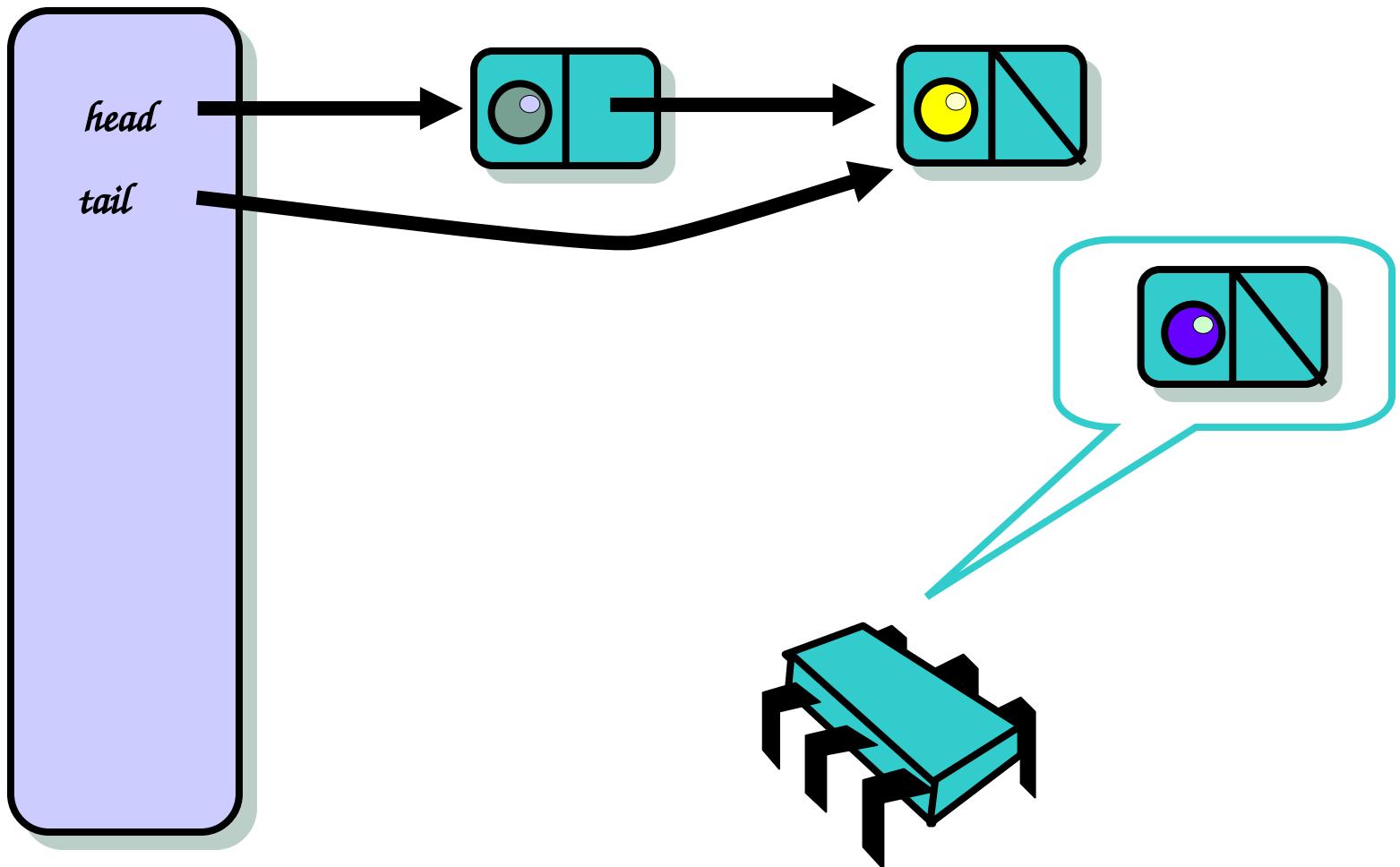
# Compare and Set



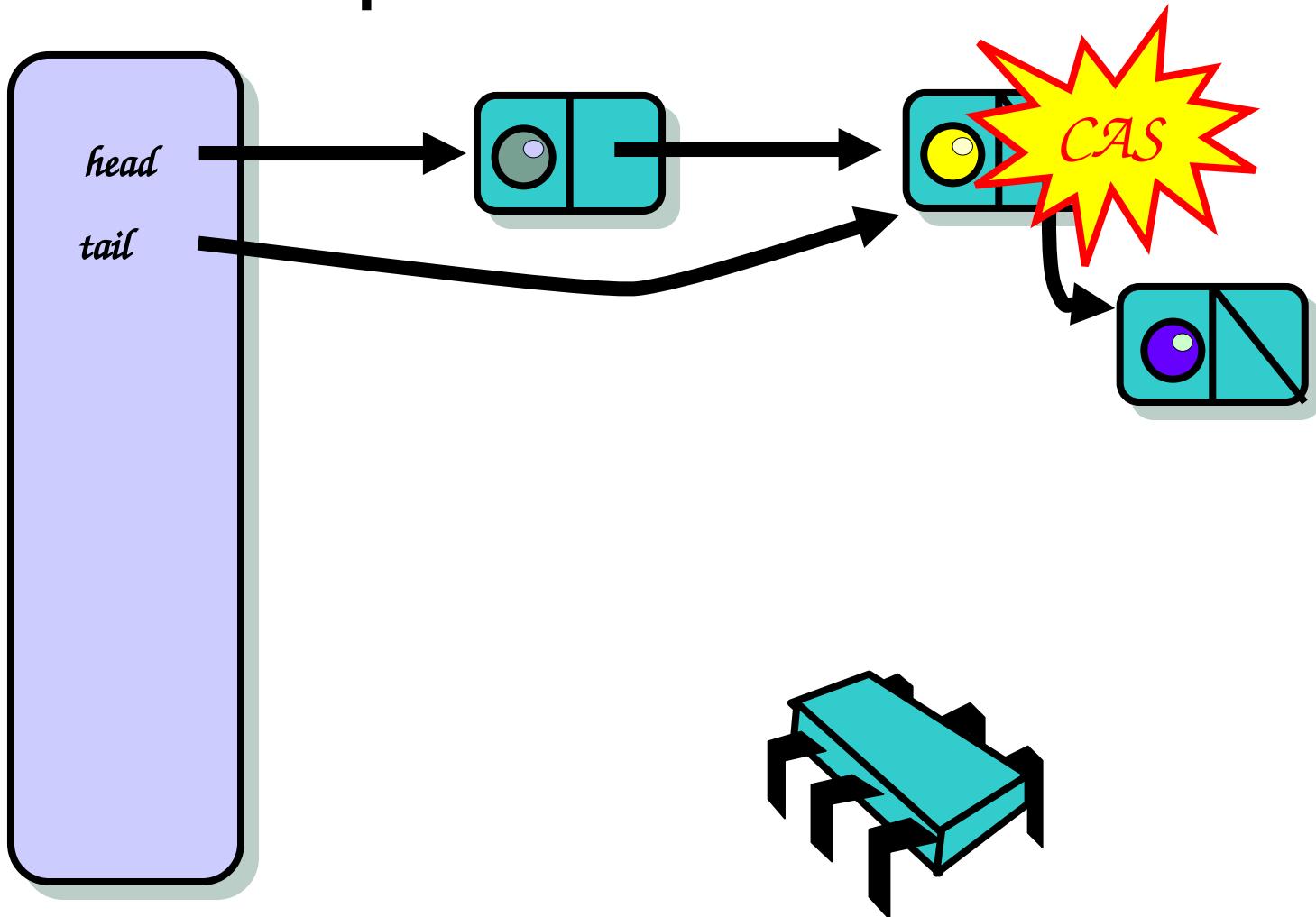
# Enqueue



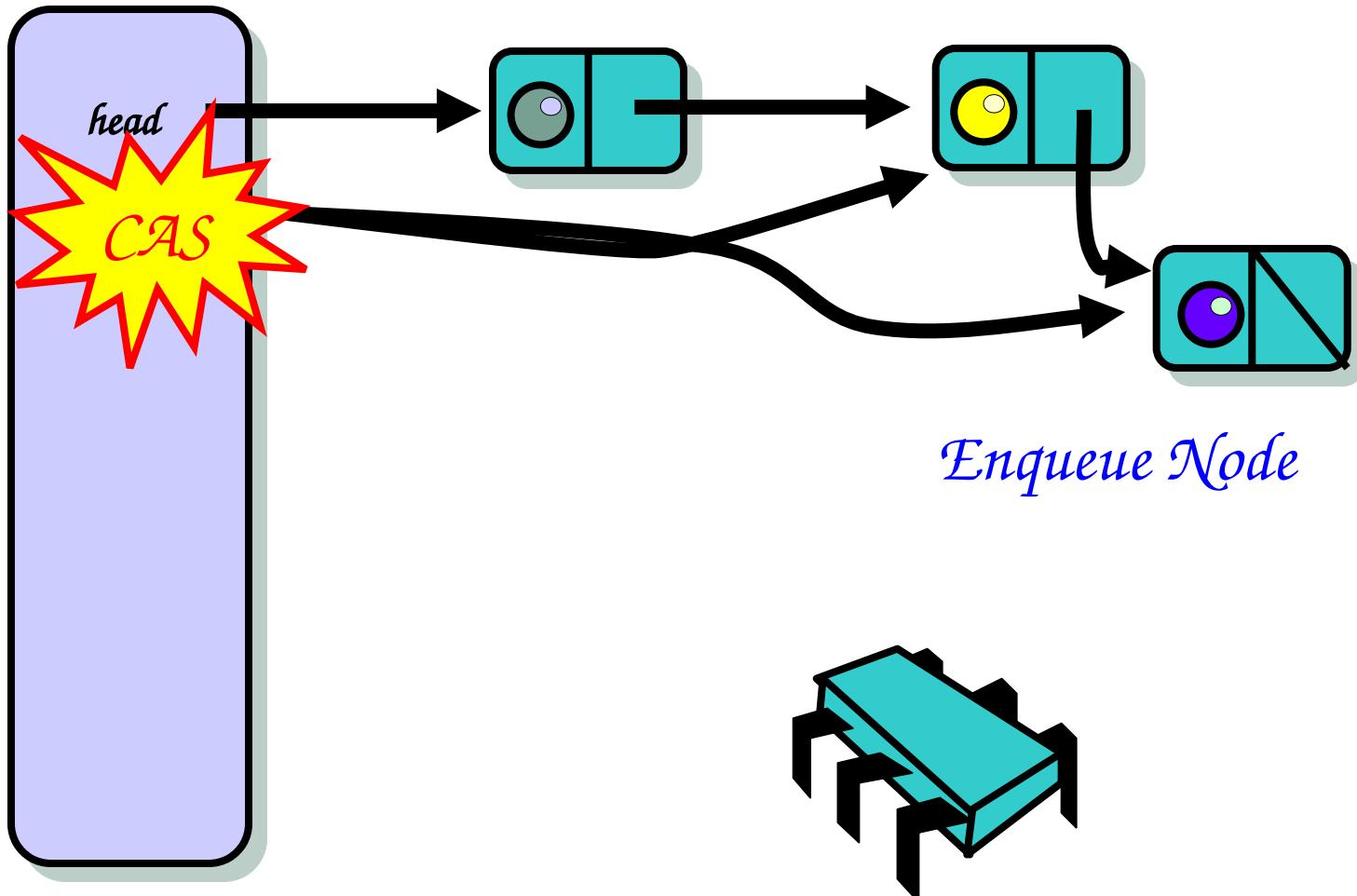
# Enqueue



# Logical Enqueue



# Physical Enqueue



# Enqueue

- The enq() method is lazy
- Enqueue consists of two steps:
  - Logical enqueue – appends the node to the linked list
  - Physical enqueue – changes the tail field to point to the new node
- These two steps are not atomic



# Enqueue

- It is thus possible that a new item is reachable by other threads before it actually becomes the tail
- The tail field can thus refer to either
  - The actual last node or
  - The penultimate node



# Enqueue

- When enqueueing you have to make sure that you are adding an item to the actual end of the queue
- So check whether the tail you are working with has a successor

# Enqueue

- What do you do if you find
  - Tail pointing to the wrong value?
- Stop and help fix it
  - If **tail** node has non-*null* next field
  - CAS the queue's **tail** field to **tail.next**

# Lock-free enqueue method

```
public void enq(T value) {  
    Node node = new Node(value);  
    while (true) {  
        Node last = tail.get();  
        Node next = tail.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next, node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                } else  
                    tail.compareAndSet(last, last.next);  
            }  
        }  
    }  
}
```

# Lock-free enqueue method

```
public void enq(T value) {  
    Node node = new Node(value);  
    while (true) {  
        Node last = tail.get();  
        Node next = tail.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next, node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                } else  
                    tail.compareAndSet(last, last.next);  
            }  
        }  
    }  
}
```

*What does tail point to*

# Lock-free enqueue method

```
public void enq(T value) {  
    Node node = new Node(value);  
    while (true) {  
        Node last = tail.get();  
        Node next = tail.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next, node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                } else  
                    tail.compareAndSet(last, last.next);  
            }  
        }  
    }  
}
```

*What does tail's next point to*

# Lock-free enqueue method

```
public void enq(T value) {  
    Node node = new Node(value);  
    while (true) {  
        Node last = tail.get();  
        Node next = tail.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next, node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                } else  
                    tail.compareAndSet(last, last.next);  
            }  
        }  
    }  
}
```

*If the value pointed to by tail has not changed...*

# Lock-free enqueue method

```
public void enq(T value) {  
    Node node = new Node(value); ...and tail's next points  
    while (true) {  
        Node last = tail.get();  
        Node next = tail.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next, node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                } else  
                    tail.compareAndSet(last, last.next);  
            }  
        }  
    }  
}
```

*...and tail's next points  
to null...*

# Lock-free enqueue method

```
public void enq(T value) {  
    Node node = new Node(value);  
    while (true) {  
        Node last = tail.get();  
        Node next = tail.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next, node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                } else  
                    tail.compareAndSet(last, last.next);  
            }  
        }  
    }  
}
```

*Logical enqueue*

# Lock-free enqueue method

```
public void enq(T value) {  
    Node node = new Node(value);  
    while (true) {  
        Node last = tail.get();  
        Node next = tail.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next, node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                } else  
                    tail.compareAndSet(last, last.next);  
            }  
        }  
    }  
}
```

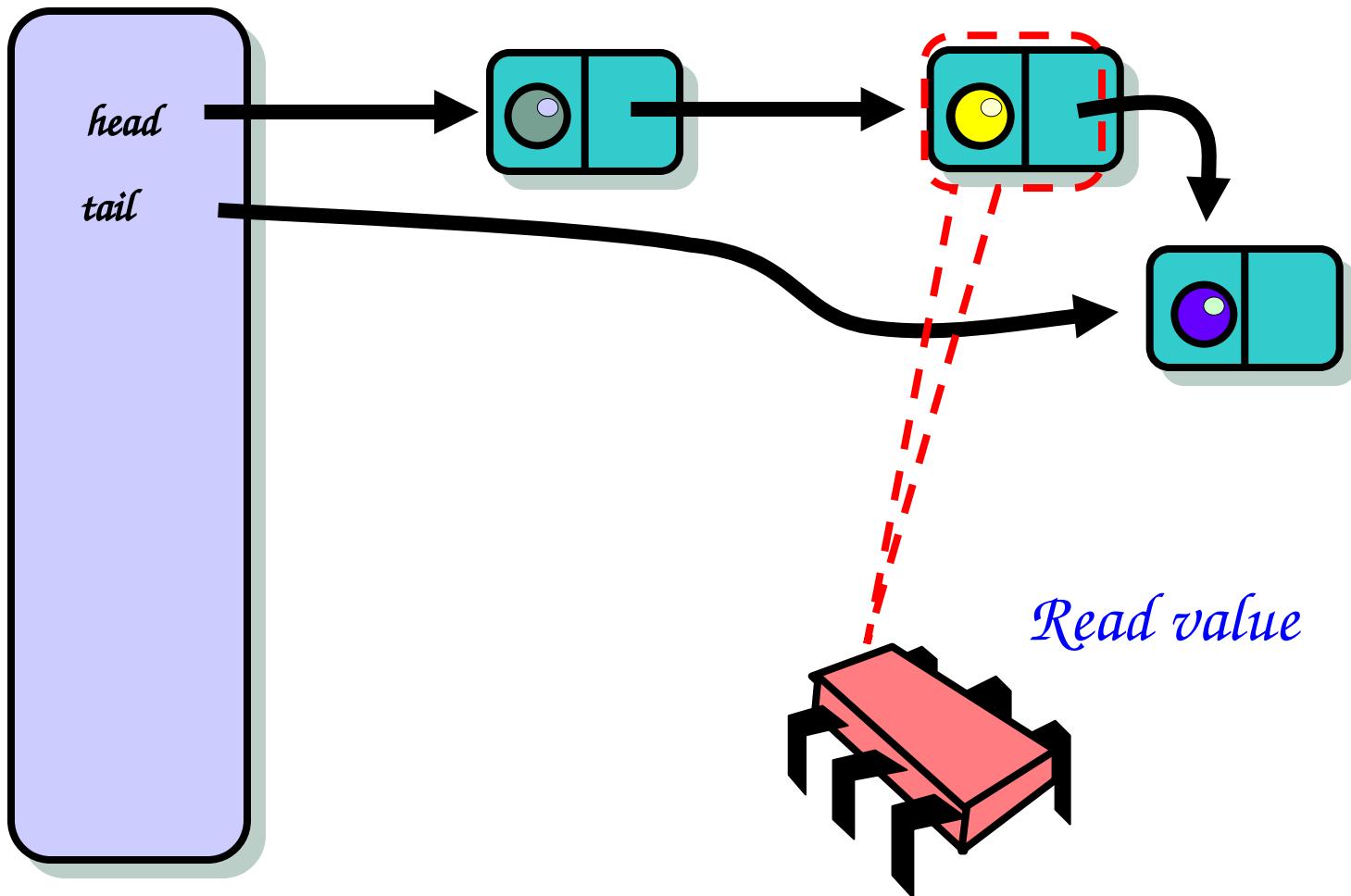
*Physical enqueue*

# Lock-free enqueue method

```
public void enq(T value) {  
    Node node = new Node(value);  
    while (true) {  
        Node last = tail.get();  
        Node next = tail.next.get();  
        if (last == tail.get()) {  
            if (next == null) {  
                if (last.next.compareAndSet(next, node)) {  
                    tail.compareAndSet(last, node);  
                    return;  
                } else  
                    tail.compareAndSet(last, last.next);  
            }  
        }  
    }  
}
```

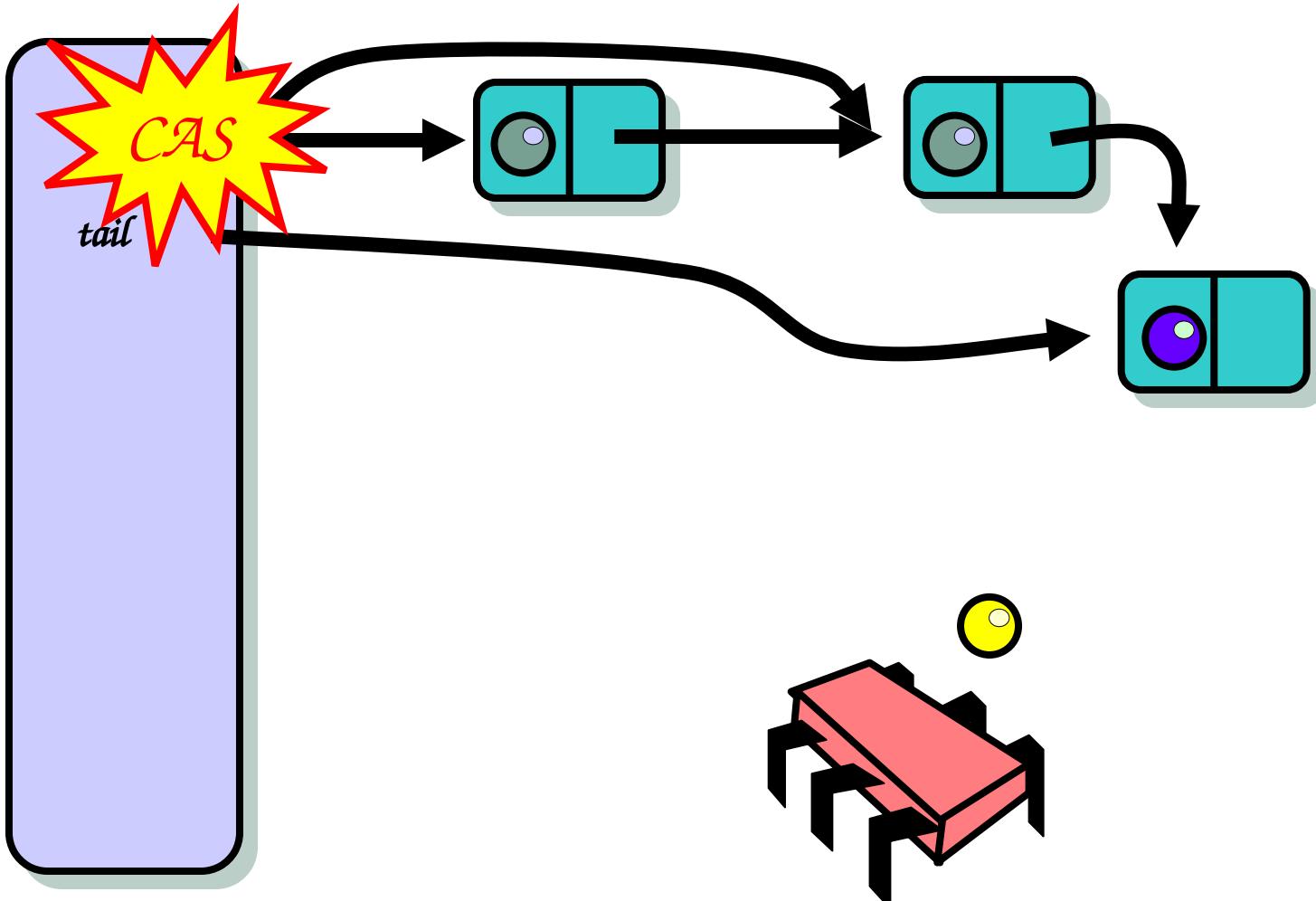
If tail has a successor it 'helps' other nodes by advancing tail to refer to the successor

# Dequeuer



*Make first Node new  
sentinel*

# Dequeueuer





# Dequeuer

- Although queue is unbounded, you always have to check that queue is not empty
- Without counter, queue is empty when  $\text{head} = \text{tail} = \text{sentinel}$

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

*What does head point to*

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

*What does tail point to*

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

*Is there a node after head?*

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

*Has the value of head changed?*

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

*Is the queue empty?*

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

*Is the queue still empty?*

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

*If a new value node  
have arrived, 'help'  
others by advancing tail*

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

*If the queue is not empty*

# Lock-free dequeue method

```
public T deq() throws EmptyException {  
    while (true) {  
        Node first = head.get();  
        Node last = tail.get();  
        Node next = first.next.get();  
        if (first == head.get()) {  
            if (first == last) {  
                if (next == null) {  
                    throw new EmptyException();  
                }  
                tail.compareAndSet(last, next);  
            } else {  
                T value = next.value;  
                if (head.compareAndSet(first, next))  
                    return value;  
            }  
        }  
    }  
}
```

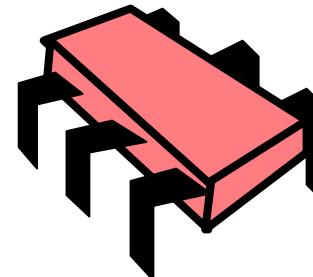
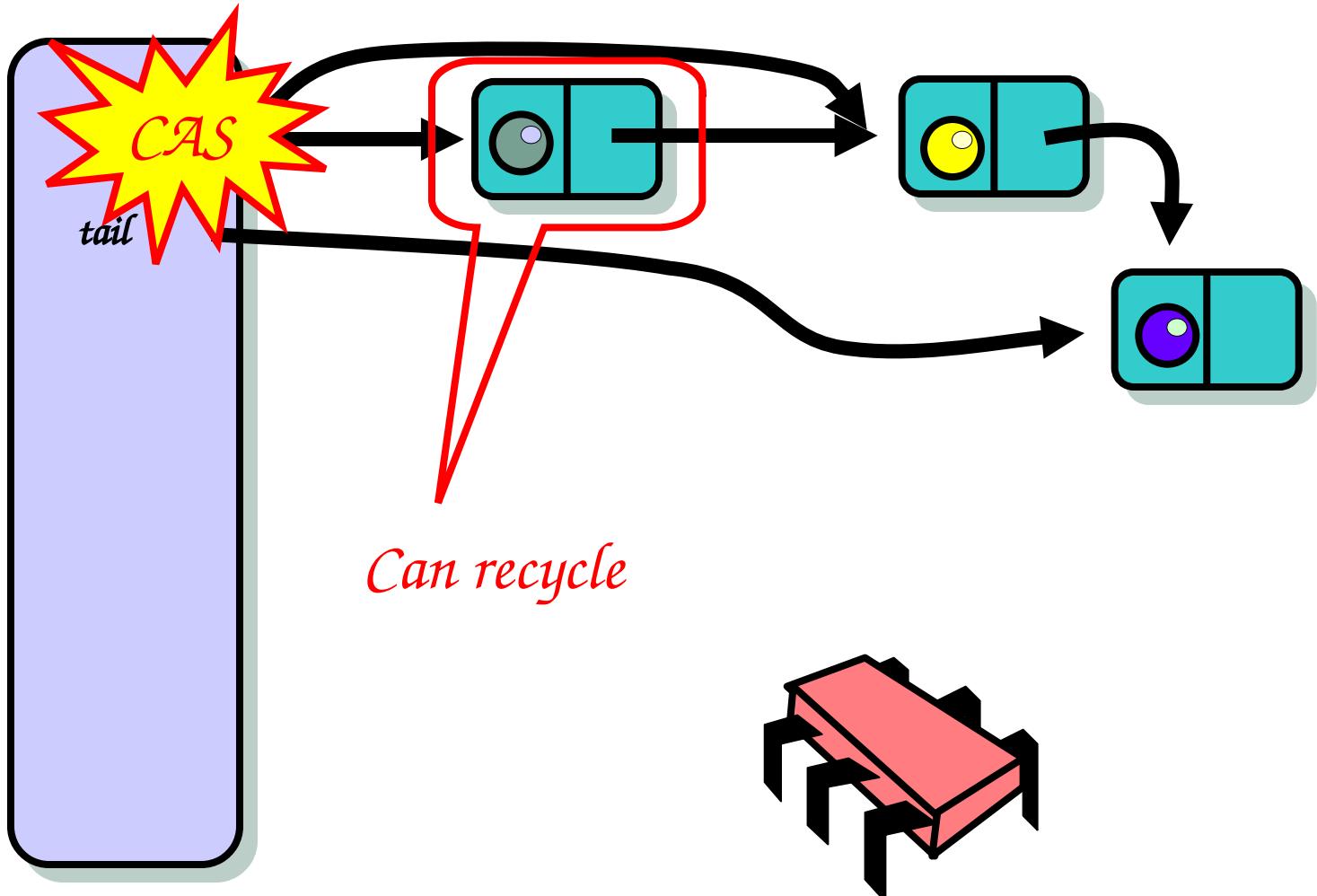
*Try to remove the first  
value from the queue*



# Memory reuse

- What do we do with nodes after we dequeue them?
- With current implementation nodes are unlinked from linked list, but we depend on built-in garbage collection to remove them from memory
- Suppose there is no garbage collector?

# Dequeuer

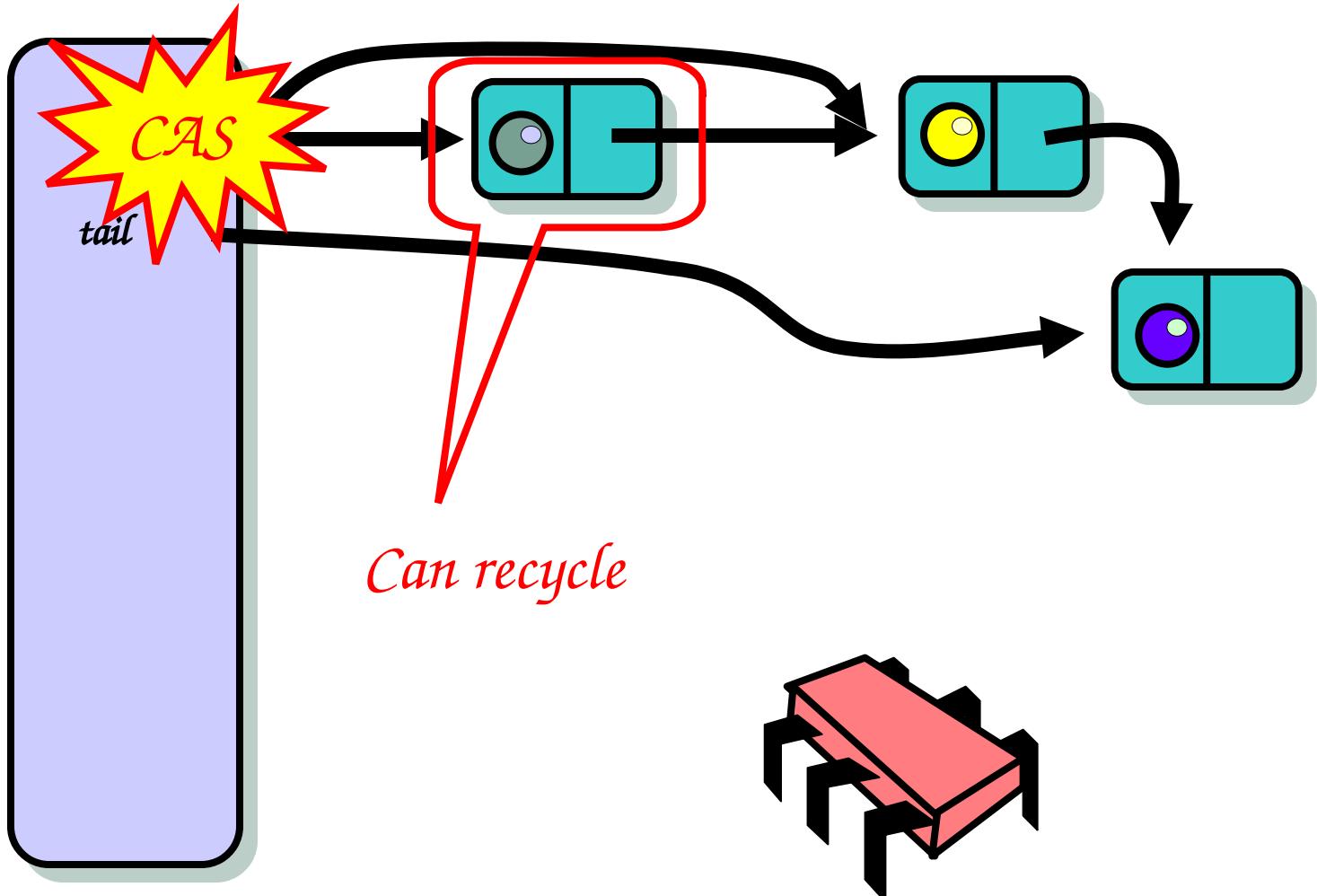




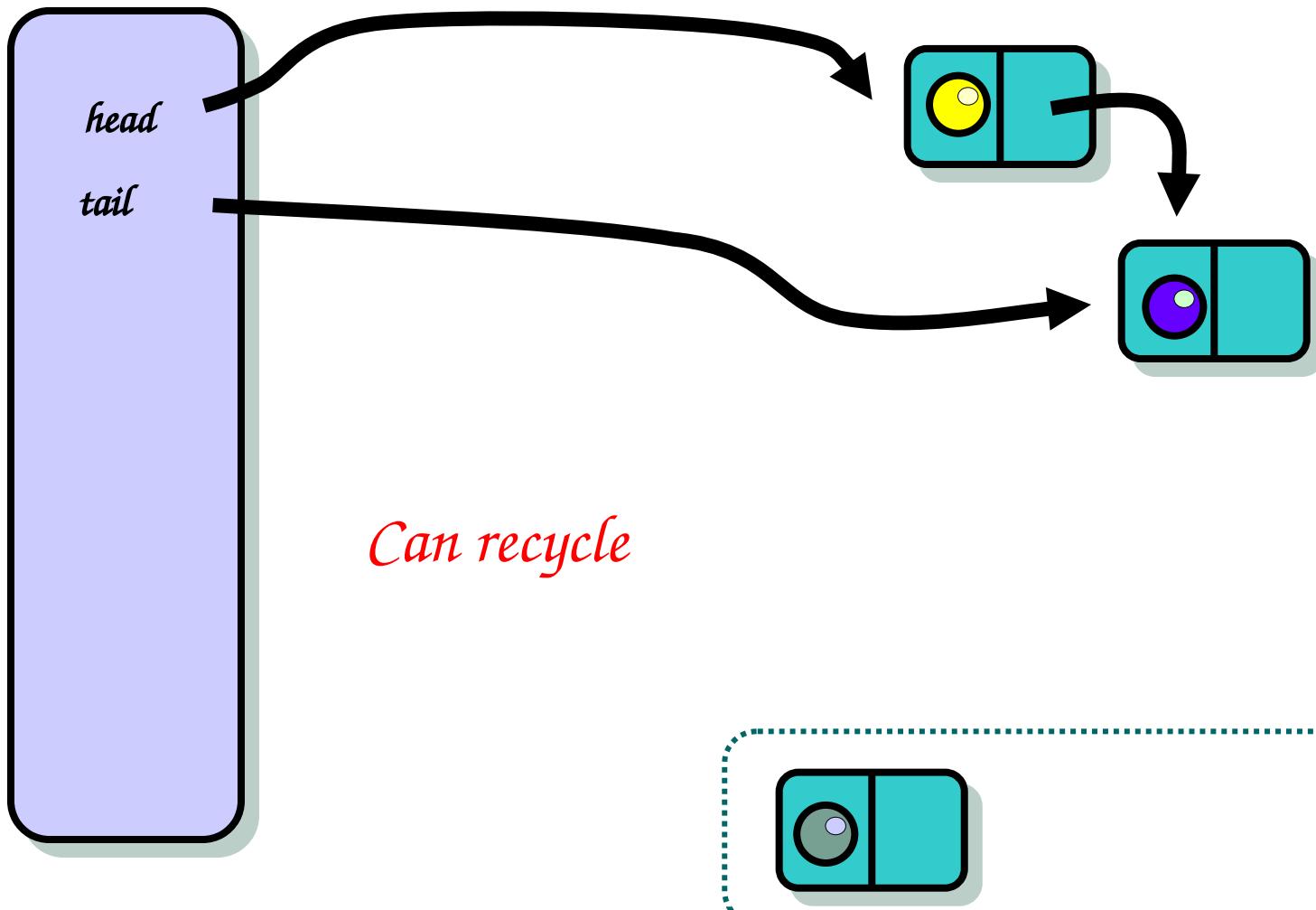
# Simple solution

- Each thread maintains its own private free-list of unused queue entries
  - When enqueue – get a node from the free-list
  - When dequeue – add removed node to free-list
- If free-list is empty, simply allocate new node through dynamic memory allocation

# Dequeuer



# Dequeuer

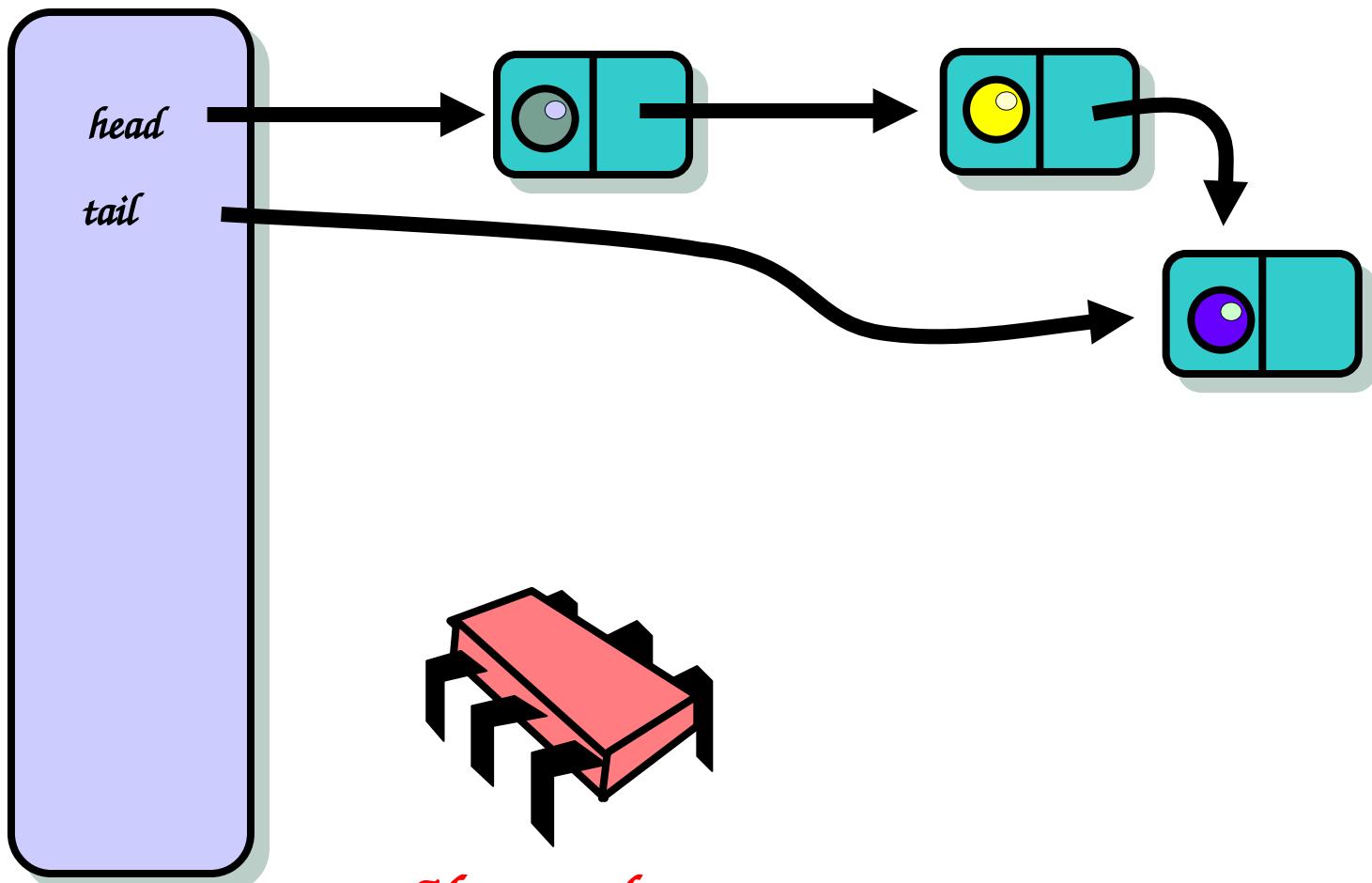




# Potential problem

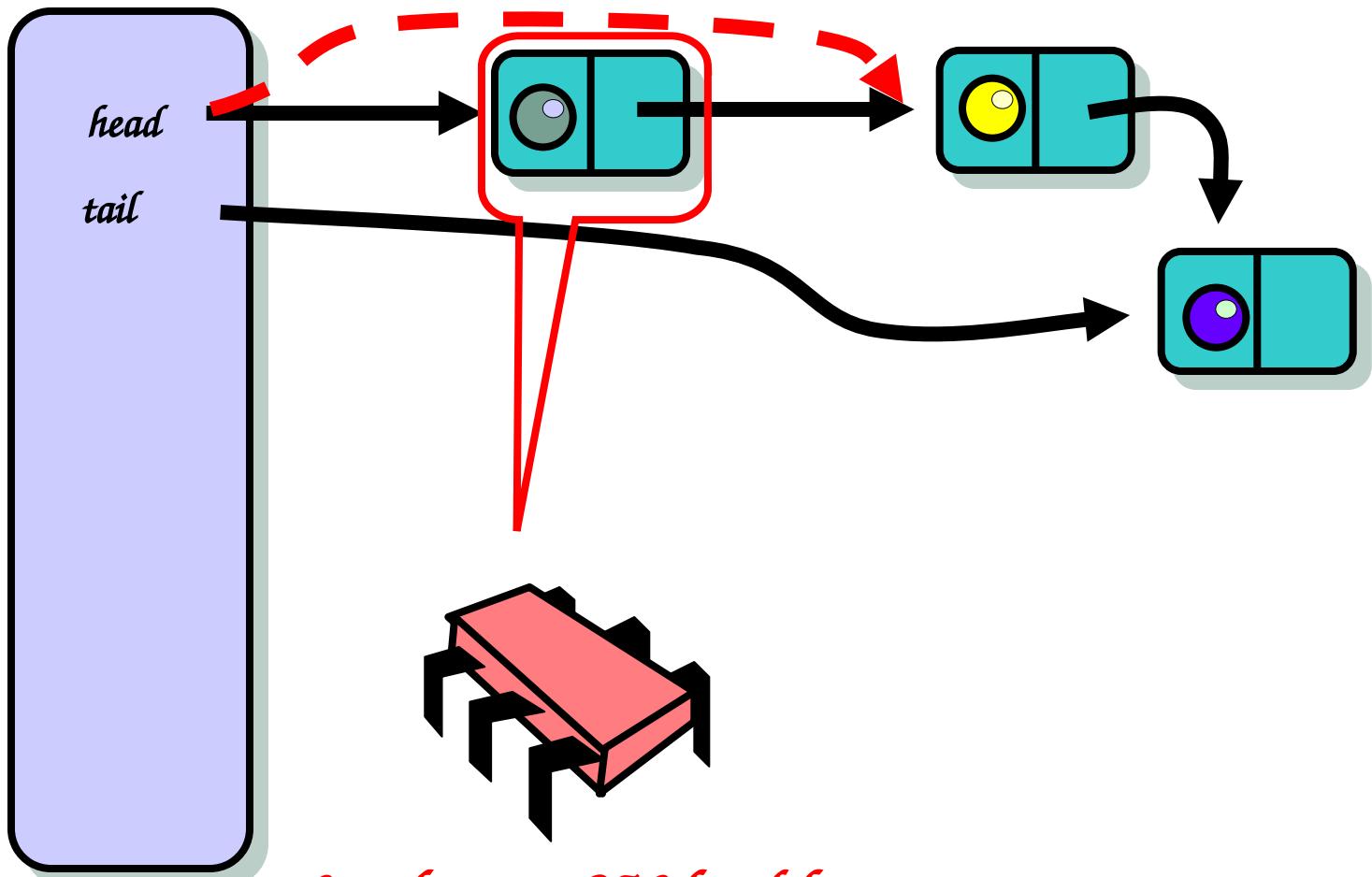
- When nodes are recycled – removed from the queue and later added again
  - Dreaded ABA problem

# Dequeuer



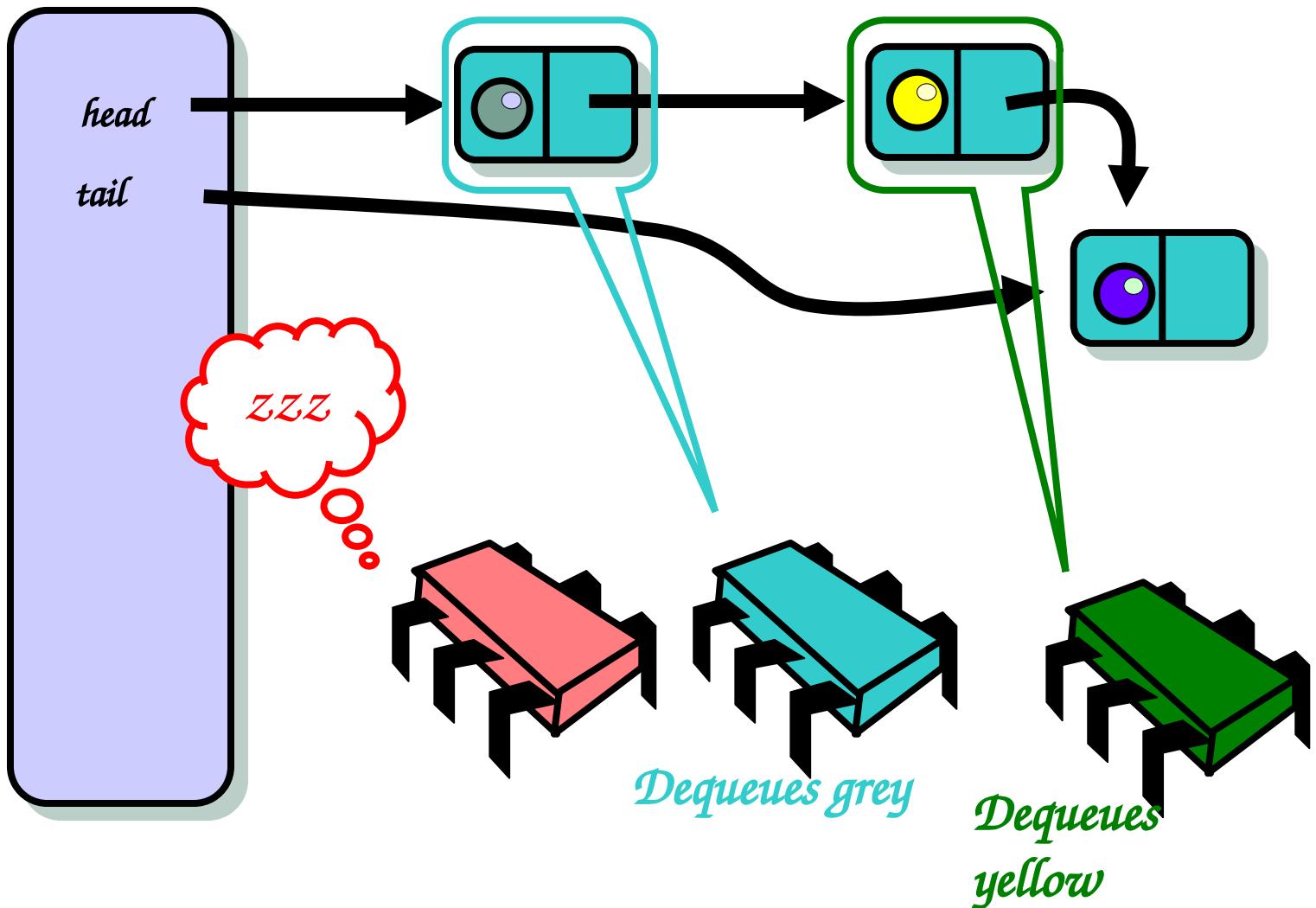
*About to dequeue grey*

# Dequeuer

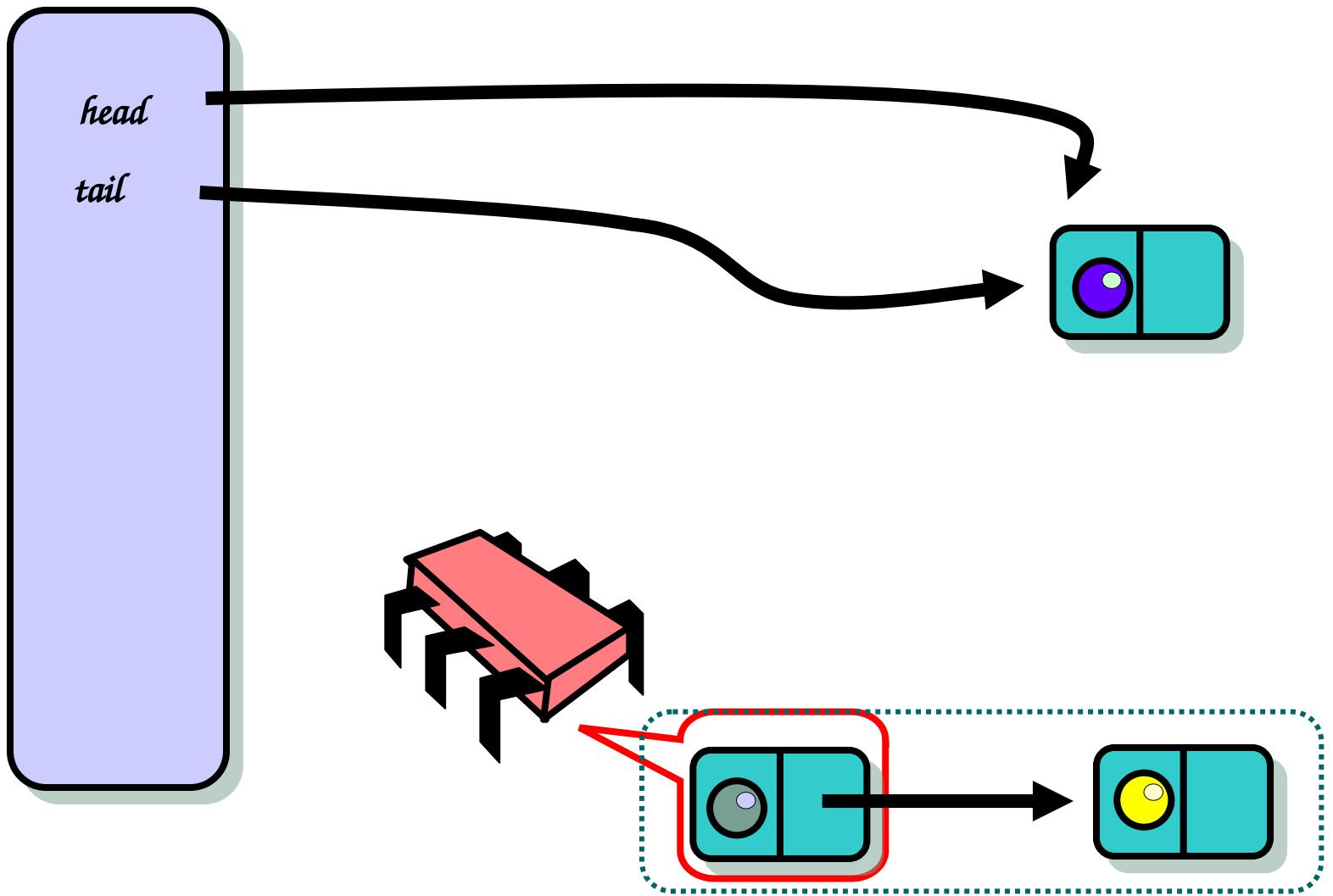


*So, about to CAS head from grey to yellow*

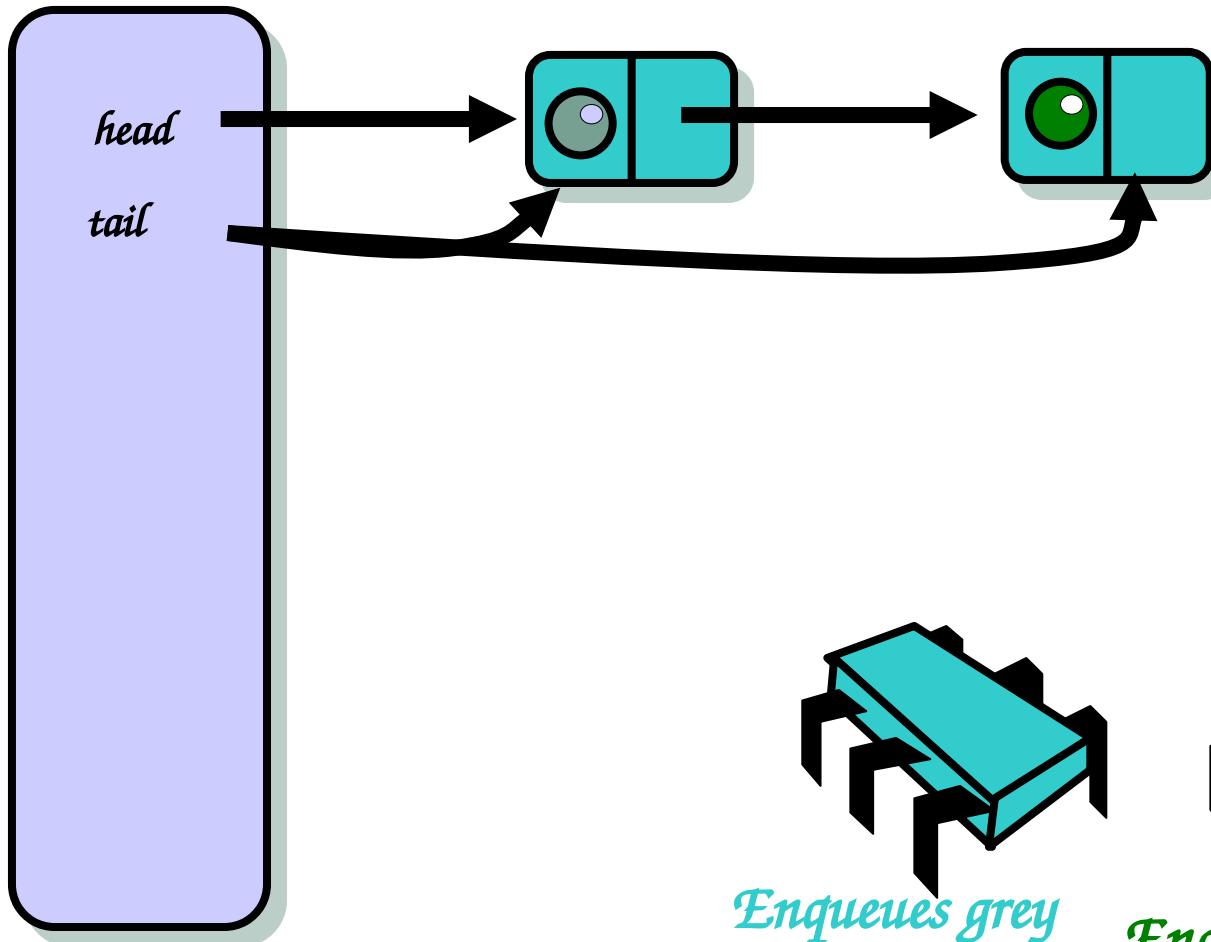
# Dequeuer



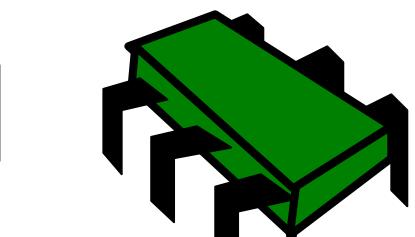
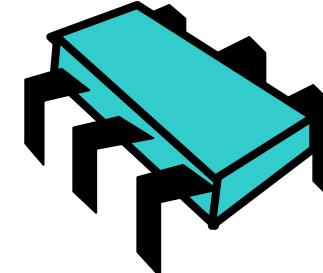
# Dequeuer



# Dequeuer

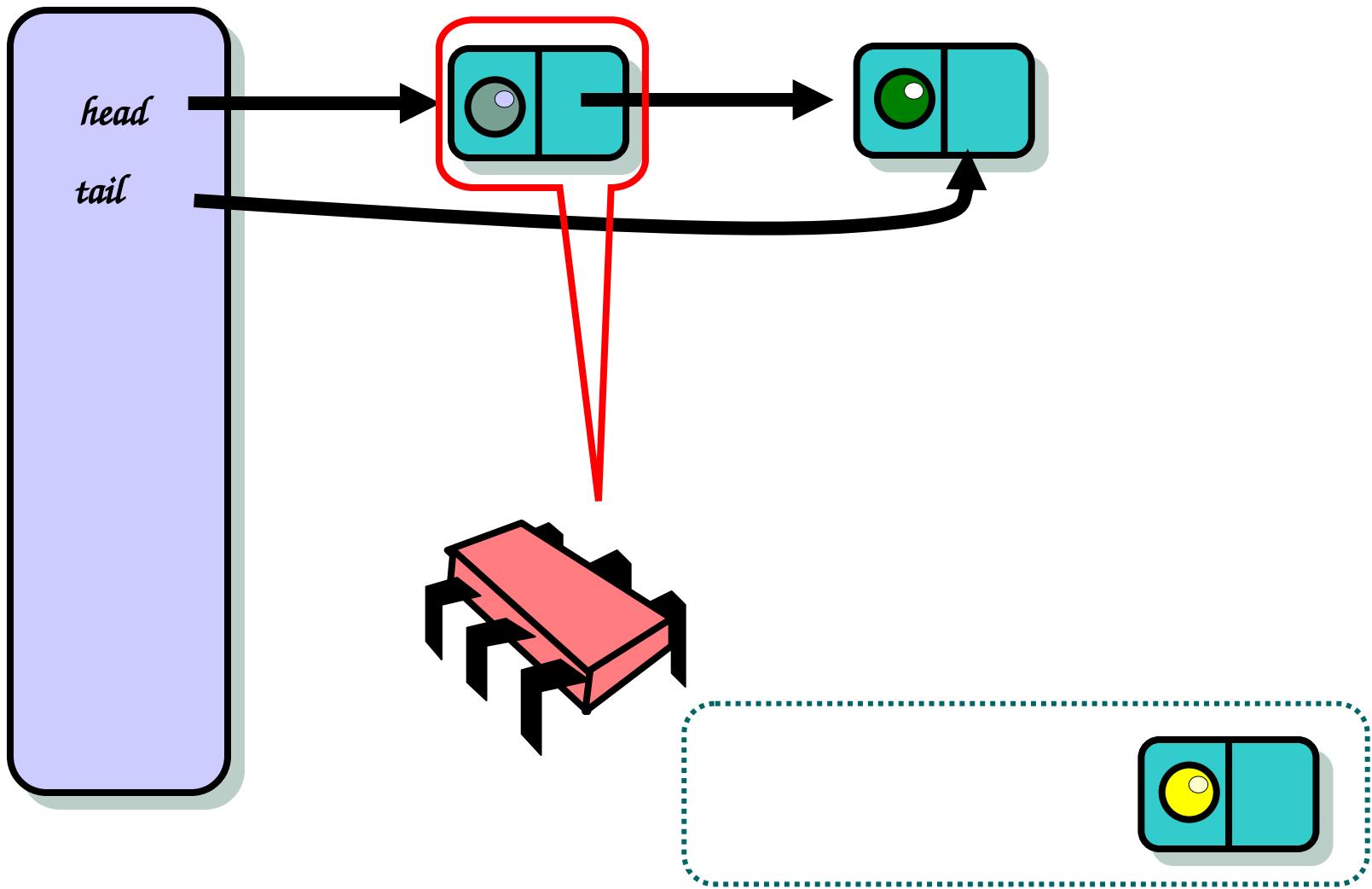


*Enqueues grey  
Recycles*

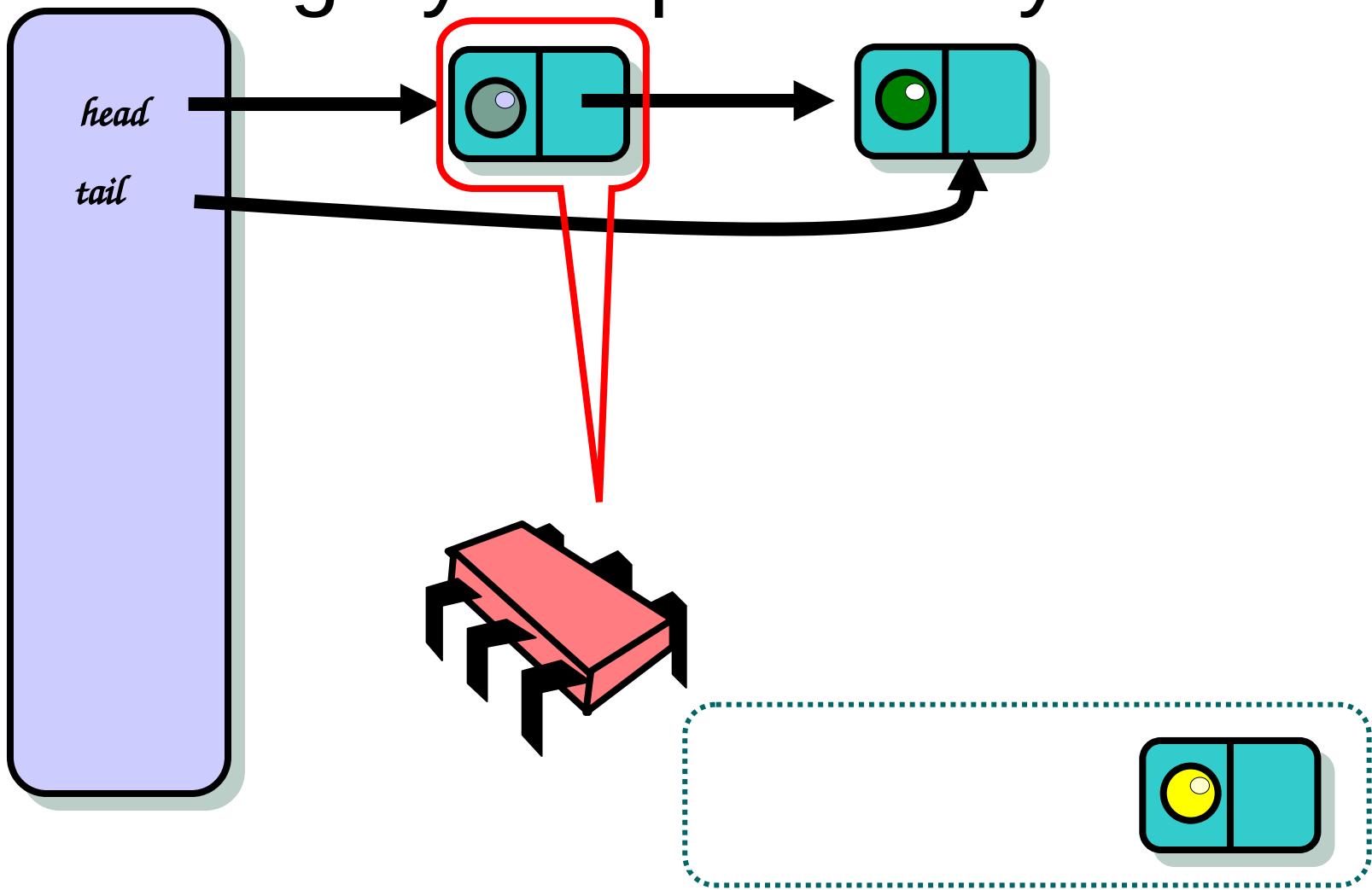


*Enqueues green*

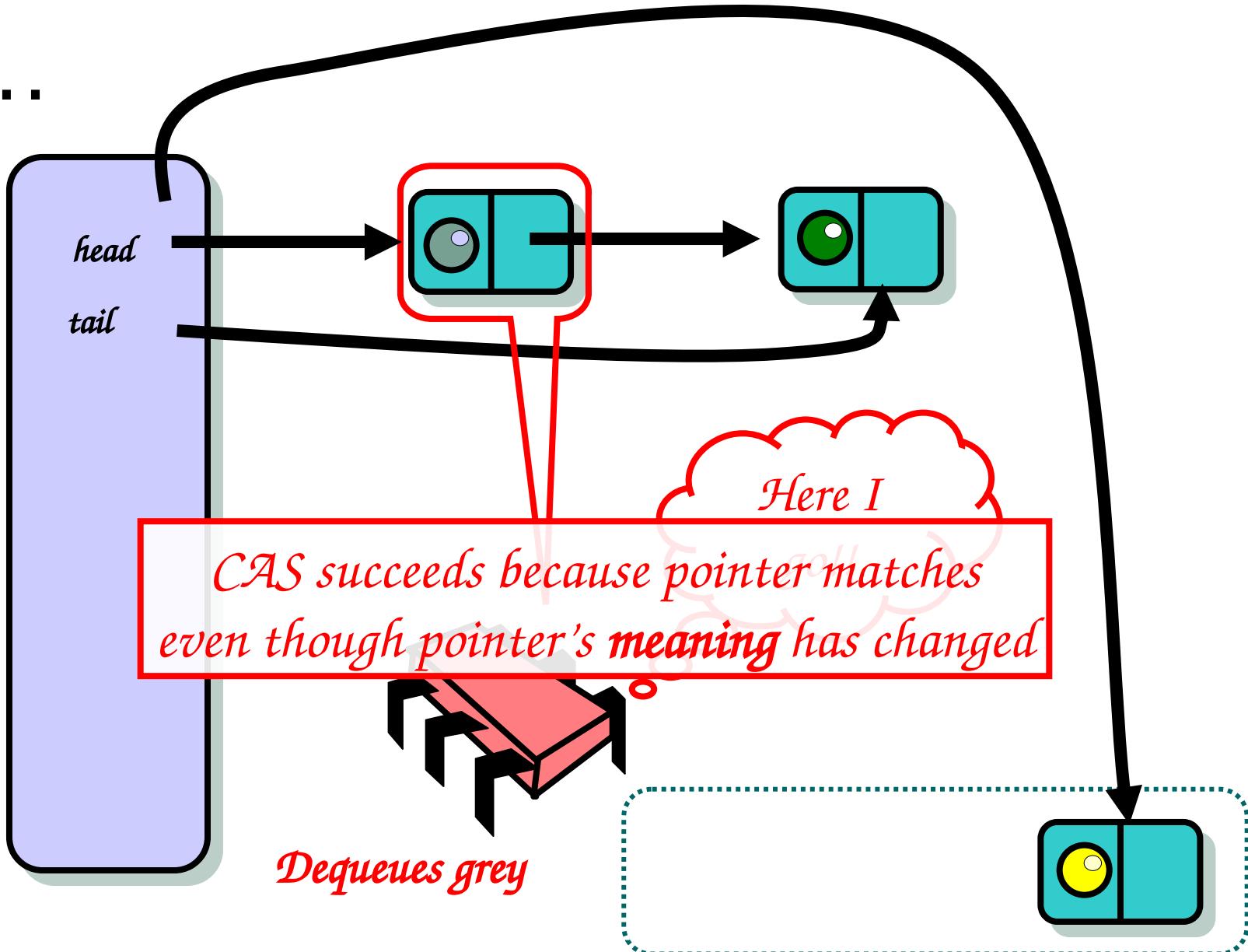
But the red thread still had grey marked



And according to its previous calculation grey still points to yellow



So...





# ABA problem

- Head ends up pointing to a node that is not in the queue anymore, but has been added to the free-list



# Dreaded ABA – A Solution

- Tag each pointer with a counter
- Unique over lifetime of node
- Pointer size vs word size issues
- Overflow?
  - Don't worry be happy?
  - Bounded tags?
- AtomicStampedReference class

# Dual Data Structures

- To reduce the synchronization overhead of the synchronous queue we split enq() and deq() into two steps

# deq() method

- If a dequeuer tries to remove an item from an empty queue:
  - It puts a reservation object in the queue to indicate that the deq() is waiting for an enq() to rendezvous with
  - deq() spins on flag in reservation



# enq() method

- When enq() sees reservation:
  - Fulfills the reservation by depositing an item
  - Notifies deq() by setting object's flag

# Dual Data Structure

- An enq() can also add its own reservation to the queue and spin on the flag waiting for a deq()
- At any time the queue contains either enq() reservations, deq() reservations or is empty



# Dual Data Structure

- Methods take effect in two stages:
  - Reservation and fulfillment



# Dual Data structures

- Good things:
  - Waiting threads spin on a locally cached flag
  - Ensures fairness
  - Linearizable

# Node class

```
private enum NodeType {ITEM, RESERVATION};  
private class Node {  
    volatile NodeType type;  
    volatile AtomicReference<T> item;  
    volatile AtomicReference<Node> next;  
  
    Node (T myItem, NodeType myType) {  
        item = new AtomicReference<T> (myItem);  
        next = new AtomicReference<Node>(null);  
        type = myType;  
    }  
}
```

# enq()

```
public void enq(T e) {  
    Node offer = new Node (e, ITEM);  
    while (true) {  
        Node t = tail.get(), h = head.get();  
        if (h == t || t.type == ITEM) {  
            Node n = t.next.get();  
            if (t == tail.get()) {  
                if (n != null) {  
                    tail.compareAndSet(t, n);  
                } else if (t.next.CAS(n, offer)) {  
                    tail.compareAndSet(t, offer);  
                    while (offer.item.get() == e) {}  
                    h = head.get();  
                    if (offer == h.next.get())  
                        head.compareAndSet(h, offer);  
                }  
            }  
        }  
    }  
    return; }}
```

# enq()

*Create new node*

```
public void enq(T e) {  
    Node offer = new Node (e, ITEM);  
    while (true) {  
        Node t = tail.get(), h = head.get();  
        if (h == t || t.type == ITEM) {  
            Node n = t.next.get();  
            if (t == tail.get()) {  
                if (n != null) {  
                    tail.compareAndSet(t, n);  
                } else if (t.next.CAS(n, offer)) {  
                    tail.compareAndSet(t, offer);  
                    while (offer.item.get() == e) {}  
                    h = head.get();  
                    if (offer == h.next.get())  
                        head.compareAndSet(h, offer);  
                }  
            }  
        }  
    }  
    return; }}
```

# enq()

If the queue is empty  
OR if it contains only  
**ITEMS...**

```
public void enq(T e) {  
    Node offer = new Node (e, ITEM);  
    while (true) {  
        Node t = tail.get(), h = head.get();  
        if (h == t || t.type == ITEM) {  
            Node n = t.next.get();  
            if (t == tail.get()) {  
                if (n != null) {  
                    tail.compareAndSet(t, n);  
                } else if (t.next.CAS(n, offer)) {  
                    tail.compareAndSet(t, offer);  
                    while (offer.item.get() == e) {}  
                    h = head.get();  
                    if (offer == h.next.get())  
                        head.compareAndSet(h, offer);  
                }  
            }  
        }  
    }  
    return; }}
```

*...then no reservations  
only add ITEM*

# enq()

```
public void enq(T e) {  
    Node offer = new Node (e, ITEM);  
    while (true) {  
        Node t = tail.get(), h = head.get();  
        if (h == t || t.type == ITEM) {  
            Node n = t.next.get();  
            if (t == tail.get()) {  
                if (n != null) {  
                    tail.compareAndSet(t, n);  
                } else if (t.next.CAS(n, offer)) {  
                    tail.compareAndSet(t, offer);  
                    while (offer.item.get() == e) {}  
                    h = head.get();  
                    if (offer == h.next.get())  
                        head.compareAndSet(h, offer);  
                }  
            }  
        }  
    }  
    return; }}
```

*If tail not last value  
move it up*

# enq()

```
public void enq(T e) {  
    Node offer = new Node (e, ITEM);  
    while (true) {  
        Node t = tail.get(), h = head.get();  
        if (h == t || t.type == ITEM) {  
            Node n = t.next.get();  
            if (t == tail.get()) {  
                if (n != null) {  
                    tail.compareAndSet(t, n);  
                } else if (t.next.CAS(n, offer)) {  
                    tail.compareAndSet(t, offer);  
                    while (offer.item.get() == e) {}  
                    h = head.get();  
                    if (offer == h.next.get())  
                        head.compareAndSet(h, offer);  
                }  
            }  
        }  
    }  
    return; }}
```

*Else add new ITEM  
node to queue*

# enq()

```
public void enq(T e) {  
    Node offer = new Node (e, ITEM);  
    while (true) {  
        Node t = tail.get(), h = head.get();  
        if (h == t || t.type == ITEM) {  
            Node n = t.next.get();  
            if (t == tail.get()) {  
                if (n != null) {  
                    tail.compareAndSet(t, n);  
                } else if (t.next.CAS(n, offer)) {  
                    tail.compareAndSet(t, offer);  
                    while (offer.item.get() == e) {}  
                    h = head.get();  
                    if (offer == h.next.get())  
                        head.compareAndSet(h, offer);  
                }  
            }  
        }  
    }  
}
```

*Now we wait until it  
gets dequeued*

# enq()

```
public void enq(T e) {  
    Node offer = new Node (e, ITEM);  
    while (true) {  
        Node t = tail.get(), h = head.get();  
        if (h == t || t.type == ITEM) {  
            Node n = t.next.get();  
            if (t == tail.get()) {  
                if (n != null) {  
                    tail.compareAndSet(t, n);  
                } else if (t.next.CAS(n, offer)) {  
                    tail.compareAndSet(t, offer);  
                    while (offer.item.get() == e) {}  
                    h = head.get();  
                    if (offer == h.next.get())  
                        head.compareAndSet(h, offer);  
                }  
            }  
        }  
    }  
}
```

# enq()

*After it is dequeued  
remove node from queue*

```
public void enq(T e) {  
    Node offer = new Node (e, ITEM);  
    while (true) {  
        Node t = tail.get(), h = head.get();  
        if (h == t || t.type == ITEM) {  
            Node n = t.next.get();  
            if (t == tail.get()) {  
                if (n != null) {  
                    tail.compareAndSet(t, n);  
                } else if (t.next.CAS(n, offer)) {  
                    tail.compareAndSet(t, offer);  
                    while (offer.item.get() == e) {}  
                    h = head.get();  
                    if (offer == h.next.get())  
                        head.compareAndSet(h, offer);  
                }  
            }  
        }  
    }  
}
```

# enq()

```
...
} else {
    Node n = h.next.get();
    if (t != tail.get() || h != head.get() || n ==
        null) {
        continue;
    }
    boolean success = n.item.compareAndSet(null,e);
    head.compareAndSet(h, n);
    if (success)
        return;
}}
```

*Otherwise fulfill  
reservation*

# enq()

```
...
} else {
    Node n = h.next.get();
    if (t != tail.get() || h != head.get() || n ==
        null) {
        continue;
    }
    boolean success = n.item.compareAndSet(null,e);
    head.compareAndSet(h, n);
    if (success)
        return;
}}
```

*Make sure nothing has changed*

# enq()

```
...
} else {
    Node n = h.next.get();
    if (t != tail.get() || h != head.get() || n ==
        null) {
        continue;
    }
    boolean success = n.item.compareAndSet(null, e);
    head.compareAndSet(h, n);
    if (success)
        return;
}}
```

# enq()

*Change item in  
reservation node to enq  
item*

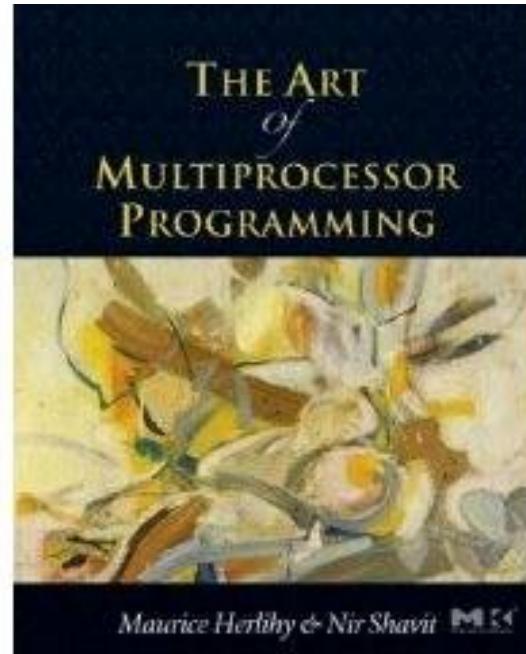
```
...
} else {
    Node n = h.next.get();
    if (t != tail.get() || h != head.get() || n ==
        null) {
        continue;
    }
    boolean success = n.item.compareAndSet(null, e);
    head.compareAndSet(h, n);
    if (success)
        return;
}}
```

COS 226

# Chapter 11

## Concurrent Stacks and Elimination

# Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



# Stacks

- Methods
  - `pop()`
  - `push(x)`
- Last-in-First-out (LIFO) order



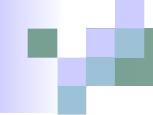
# Concurrent Stacks

- Do stacks provide opportunity for concurrency?
  - push() and pop() calls access only the top of the stack
- However, stacks are not inherently sequential



# Lock-Free Stacks

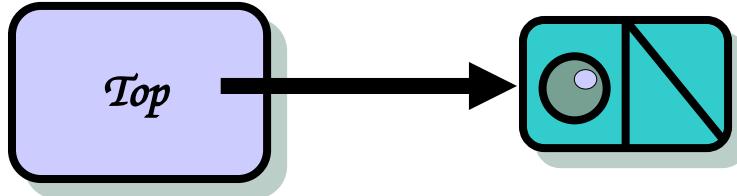
- Linked-list implementation
- top field points to the first node
- pop() from an empty list throws an exception
- push() forms a new node



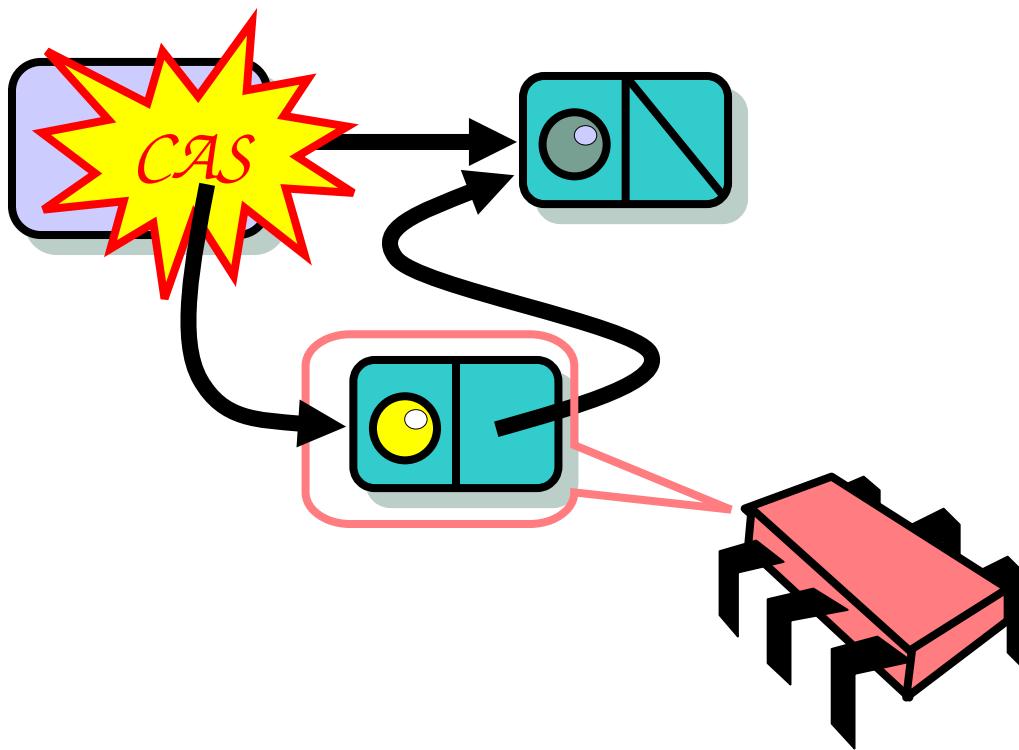
# Lock-free Stacks

- Lock-free = atomic
- Use compareAndSet methods to change the link values in the list

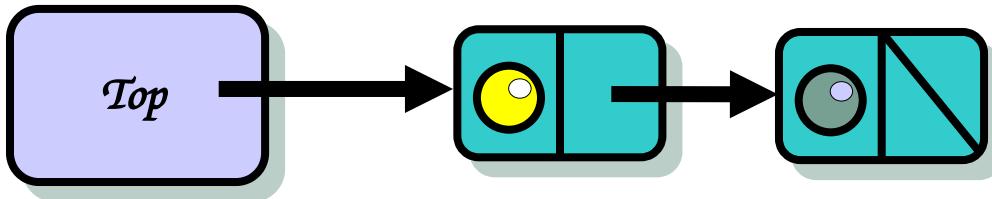
# Empty Stack



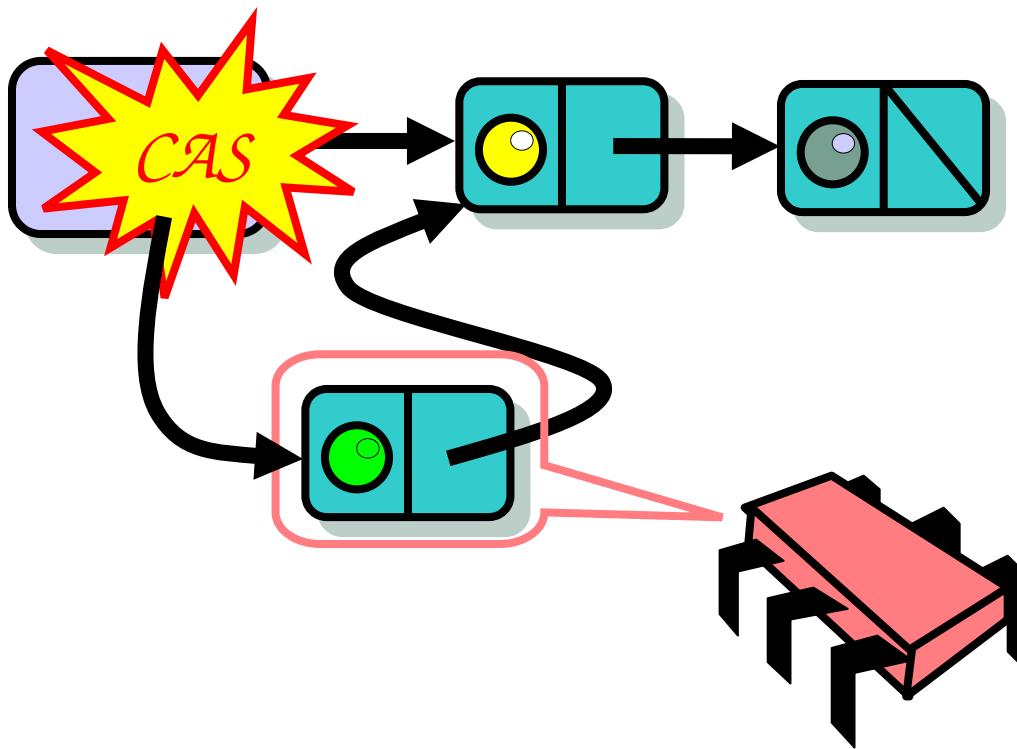
# Push



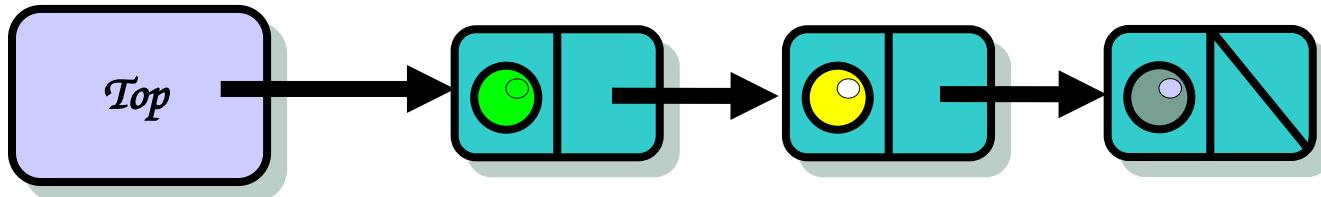
# Push



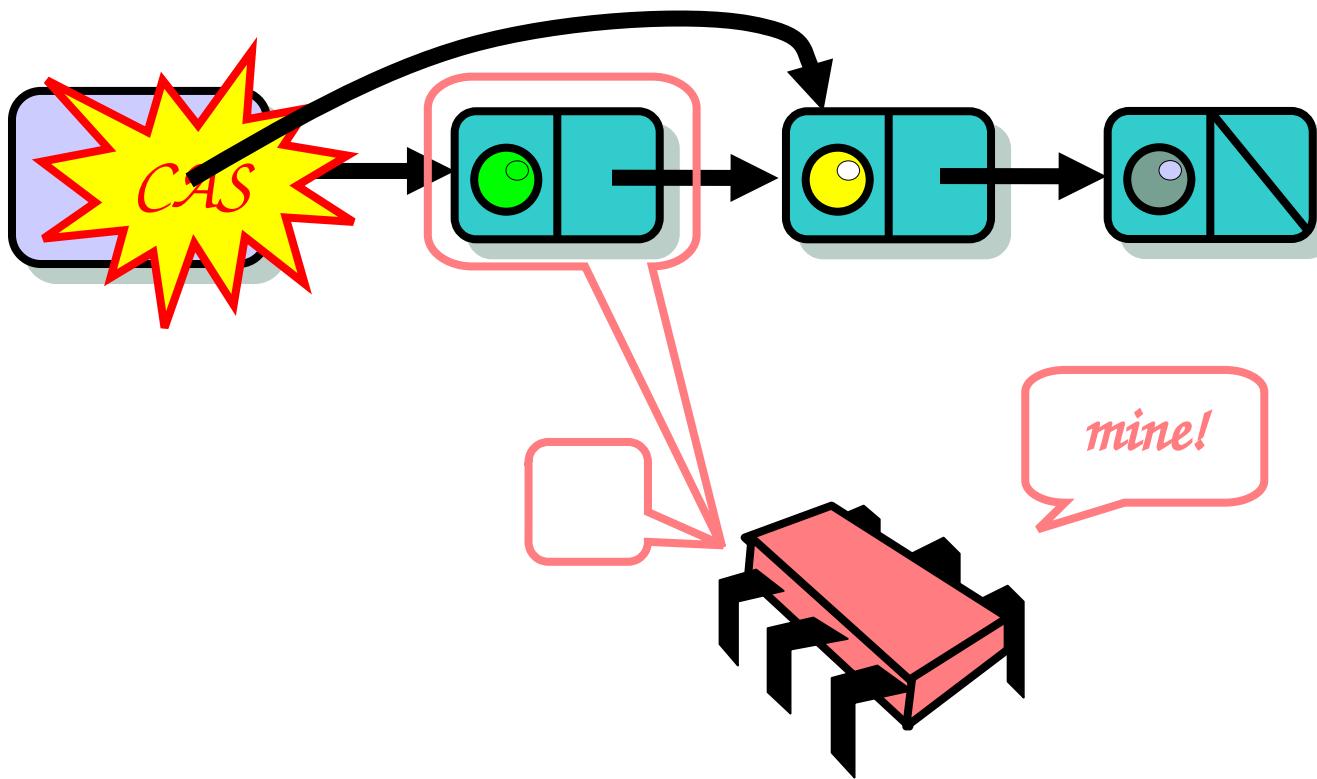
# Push



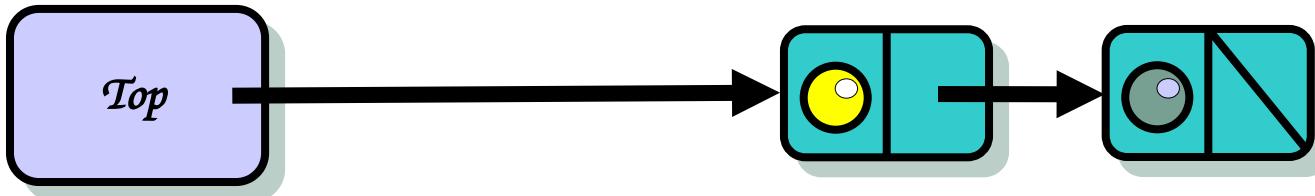
# Stack



# Pop



# Pop



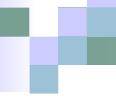
# Lock-free Stack

```
public boolean tryPush(Node node){  
    Node oldTop = top.get();  
    node.next = oldTop;  
    return (top.compareAndSet(oldTop, node))  
}
```

In what scenario will the tryPush return false?

# Lock-free Stack

- If the compareAndSet returns a false it means that there is high contention
- In scenarios where there are high contention it is more effective to backoff and try again later



# Lock-free Stack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else backoff.backoff();  
    }  
}
```

# Lock-free Stack

```
public T pop() throws EmptyException {  
    while (true) {  
        Node returnNode = tryPop();  
        if (returnNode != null)  
            return returnNode.value;  
        else  
            backoff.backoff();  
    }  
}  
  
protected Node tryPop() throws EmptyException {  
    Node oldTop = top.get();  
    if (oldTop == null)  
        throw new EmptyException()  
    Node newTop = oldTop.next;  
    if (top.compareAndSet(oldTop, newTop))  
        return oldTop;  
    else  
        return null;  
}
```



# Lock-free Stack

- Good
  - No locking
- Bad
  - Even with backoff, huge contention at top
  - In any case, no real parallelism



# Lock-free Stack

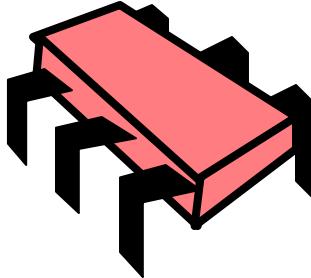
- The LockFreeStack implementation scales poorly
  - Stack's top field is a source of contention
  - Sequential bottleneck
    - Method calls can proceed only one after the other
- Exponential Backoff alleviates contention, but does nothing for bottleneck



# Observation

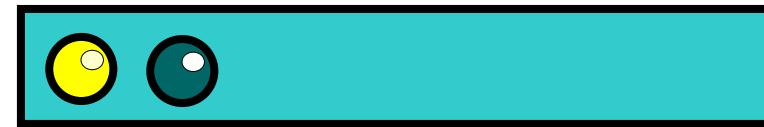
- If a `push()` is immediately followed by a `pop()` the two operations cancel out and the stack does not change
- We can exploit this observation

# Observation

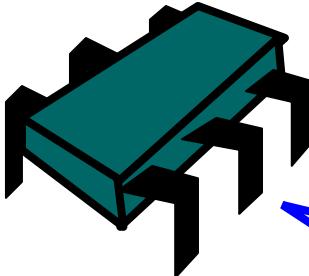


*Push()*

*linearizable stack*



*Pop()*



*Yes!*

*After an equal number  
of pushes and pops,  
stack stays the same*

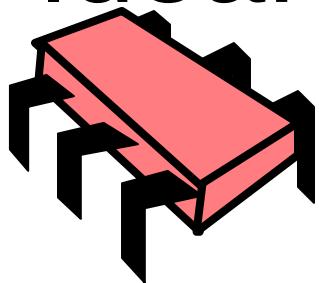
# Elimination

- Cancel out concurrent pairs of pop() and push() method calls
- Thread calling push() exchanges value with thread calling pop()
- No modification to the stack needed
- The two calls *eliminate* one another

# Elimination

- Idea:
  - Threads eliminate one another through an EliminationArray
  - Threads pick random array entries to try to meet complementary calls
  - Pairs of pop() and push() method calls that pick the same location exchange values and returns

# Idea: Elimination Array

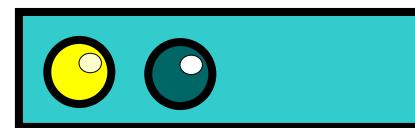


*Push()*

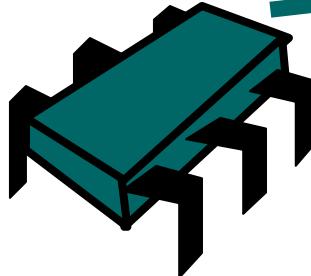
*Pick at random*



*stack*



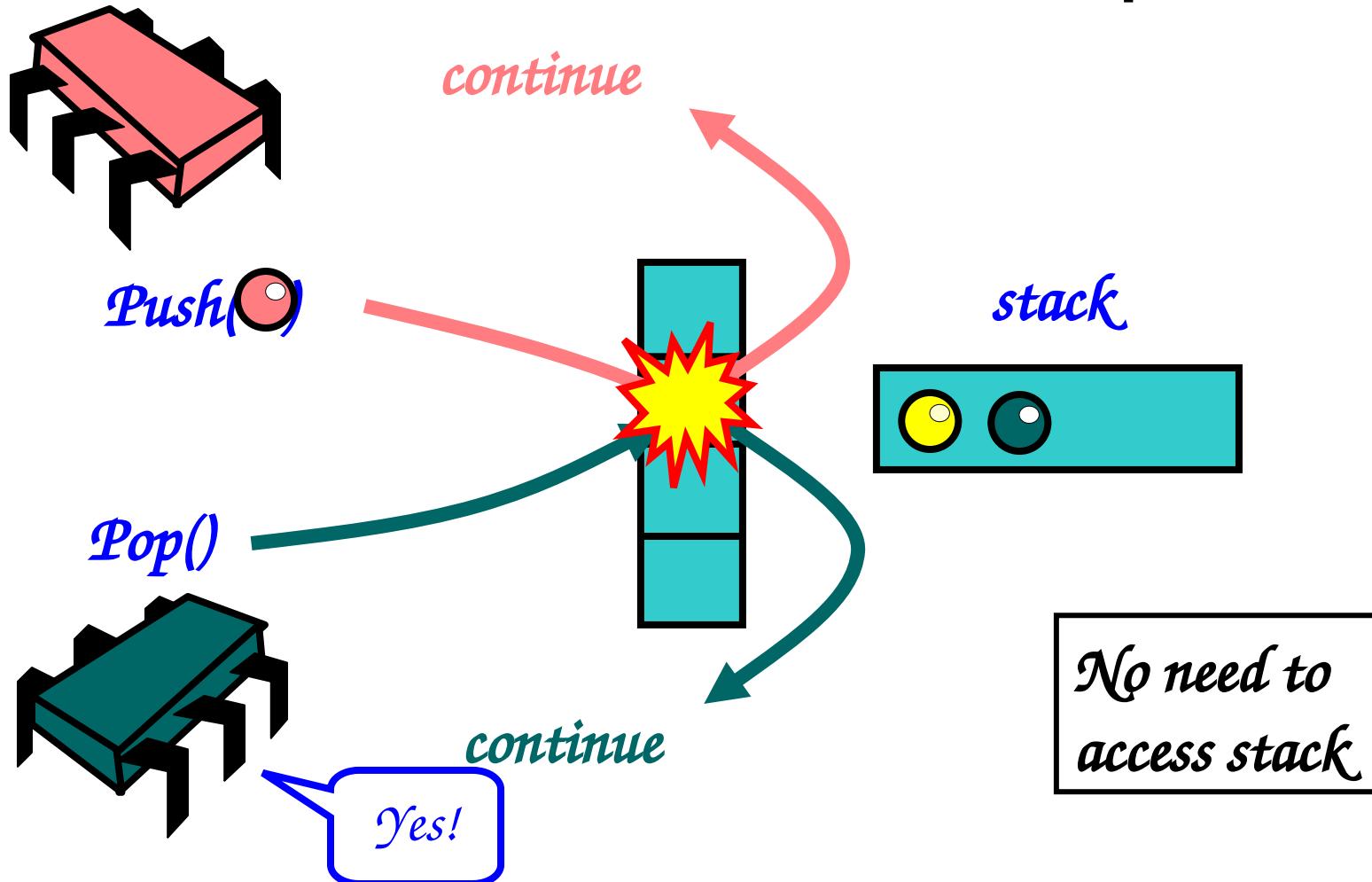
*Pop()*



*Pick at random*

*Elimination  
Array*

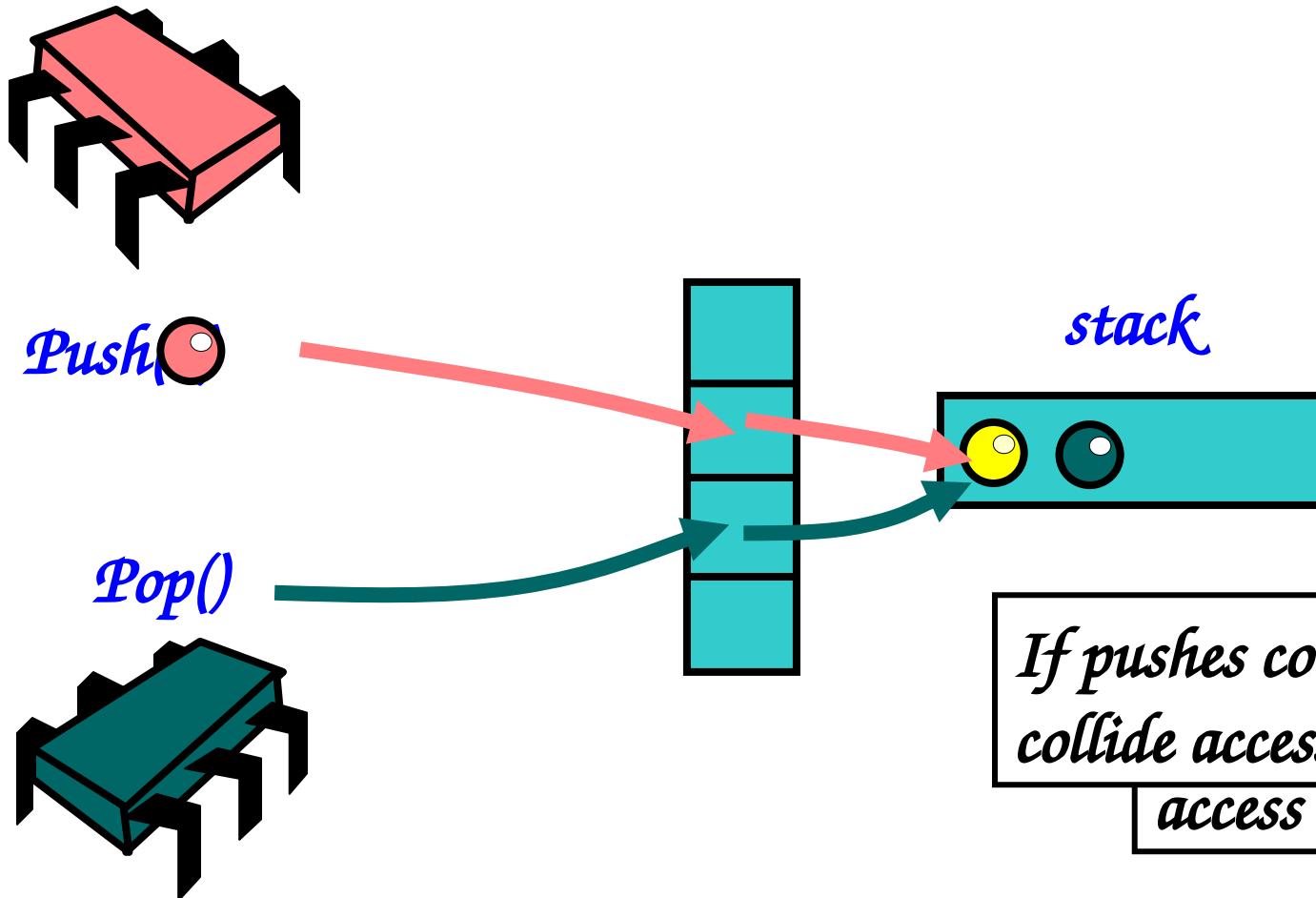
# Push Collides With Pop



# Elimination

- A thread whose call cannot be eliminated
  - Because it failed to find a partner (picked a wrong location) or
  - Picked a partner of the wrong type (pop() meeting a pop())
- Can either try again to eliminate at a new location or access the stack

# No Collision



# Elimination

- The EliminationArray can be used as a backoff scheme:
  - Each thread first access the Stack
  - If it fails (compareAndSet fails) it attempts to eliminate its call using the array instead of simply backing off
  - If it fails to eliminate, it access the Stack again
  - And so on...
- Called an EliminationBackoffStack



# Elimination Backoff Stack

- Threads that push and pop should be allowed to cancel out
- However we must avoid a situation in which a thread can make an offer to more than one thread



# Exchangers

- Object that allow exactly two threads (and no more) to rendezvous and exchange values
- The EliminationArray can be implemented as an array of Exchangers

# LockFreeExchanger

- A LockFreeExchanger<T> object allows two threads to exchange values of type T
- If A calls exchange() with a and B calls exchange() with b, then A should return b and vice versa

# LockFreeExchanger

- On a high level:
  - First thread arrives to write its values and spins until a second thread arrives
  - The second detects that the first is waiting, reads it value and signals the exchange
- The first thread may timeout if a second does not show up

# LockFreeExchanger

- The exchanger has three possible states:
  - EMPTY
  - BUSY
  - WAITING
- AtomicStampedReference records the exchanger's state as an int stamp

# LockFreeExhanger

- The exchanger reads the state of the AtomicStampedReference slot and proceeds as follows:

# If EMPTY

- Thread tries to place item in slot and set state to WAITING
- If it was successful it spins waiting for another thread
- If another thread shows up, it will take the item, replace it with its own and set the state to BUSY
- The waiting state will take the item and reset the state to EMPTY

# LockFreeExchanger

```
public class LockFreeExchanger {  
    public T exchange(T myItem) {  
        int[] stampholder = {EMPTY};  
        while (true) {  
            T yrItem = slot.get(stampHolder);  
            int stamp = stampholder[0];  
            switch(stamp) {
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
            if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
        } else {
            yrItem = slot.get(stampHolder);
            slot.set(null, EMPTY);
            return yrItem;
        }
    } break;
```

# If EMPTY

```
case FEMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
    WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
            if (slot.compareAndSet(myItem, null, WAITING,
            EMPTY)){throw new TimeoutException();
                Slot is free, try to insert myItem and change
                state to WAITING
                return yrItem;
            }
        } break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
            if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
Loop while still time left to attempt exchange
            slot.set(null, EMPTY);
            return yrItem;
        }
    } break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
            if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
Get item from slot and check if state changed  
to BUSY
                return yrItem;
            }
        } break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
            if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
        }
        If successful reset slot state to EMPTY
        slot.set(null, EMPTY);
        return yrItem;
    }
} break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
    } else {
        yrItem = slot.get(stampHolder);
        slot.set(null, EMPTY);
        return yrItem;
    }
} break;
```

and return item found in slot

# If EMPTY

*Otherwise we ran out of time, try to  
reset state to EMPTY, if successful  
time out*

```
    em, myItem, EMPTY,  
    timeBound){  
        stampHolder[0] = stampHolder[1];  
        if (stampHolder[0] == BUSY) {  
            slot.set(null, EMPTY);  
            return yrItem;  
        }  
        if (slot.compareAndSet(myItem, null, WAITING,  
EMPTY)){throw new TimeoutException();}  
        } else {  
            yrItem = slot.get(stampHolder);  
            slot.set(null, EMPTY);  
            return yrItem;  
        }  
    } break;
```

# If EMPTY

*' If reset failed, someone showed up after all, take  
\\ that item*

```
while (System.nanoTime() < timeBound){  
    yrItem = slot.get(stampHolder);  
    if (stampHolder[0] == BUSY) {  
        slot.set(null, EMPTY);  
        return yrItem;  
    }  
    if (slot.compareAndSet(myItem, null, WAITING,  
EMPTY)){throw new TimeoutException();}  
    } else {  
        yrItem = slot.get(stampHolder);  
        slot.set(null, EMPTY);  
        return yrItem;  
    }  
} break;
```

# If EMPTY

*Set slot to EMPTY and return item found*

```
WAITING)) {  
    while (System.nanoTime() < timeBound){  
        yrItem = slot.get(stampHolder);  
        if (stampHolder[0] == BUSY) {  
            slot.set(null, EMPTY);  
            return yrItem;  
        }  
        if (slot.compareAndSet(myItem, null, WAITING,  
EMPTY)){throw new TimeoutException();  
    } else {  
        yrItem = slot.get(stampHolder);  
        slot.set(null, EMPTY);  
        return yrItem;  
    }  
} break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null)
                return yrItem
            }
            if (slot.compareAndSet(EMPTY, WAITING)) {
                throw new TimeoutException();
            } else {
                yrItem = slot.get(stampHolder);
                slot.set(null, EMPTY);
                return yrItem;
            }
        }
    } break;
```

*If initial CAS failed then someone else changed state from EMPTY to WAITING so retry from start*

# If WAITING

- Some thread is waiting and the slot contains the item
- The thread takes the item and tries to replace it with its own by changing state from WAITING to BUSY
- It may fail if another thread succeeds or if the initial thread timeout
- If it does succeed it can return the item

# If WAITING

```
case WAITING:  
if (slot.compareAndSet(yrItem, myItem, WAITING,  
BUSY))  
    return yrItem;  
break;
```

- Someone is waiting for an exchange, to try to CAS my item in and change state to BUSY
- If successful return that item, otherwise another thread got it

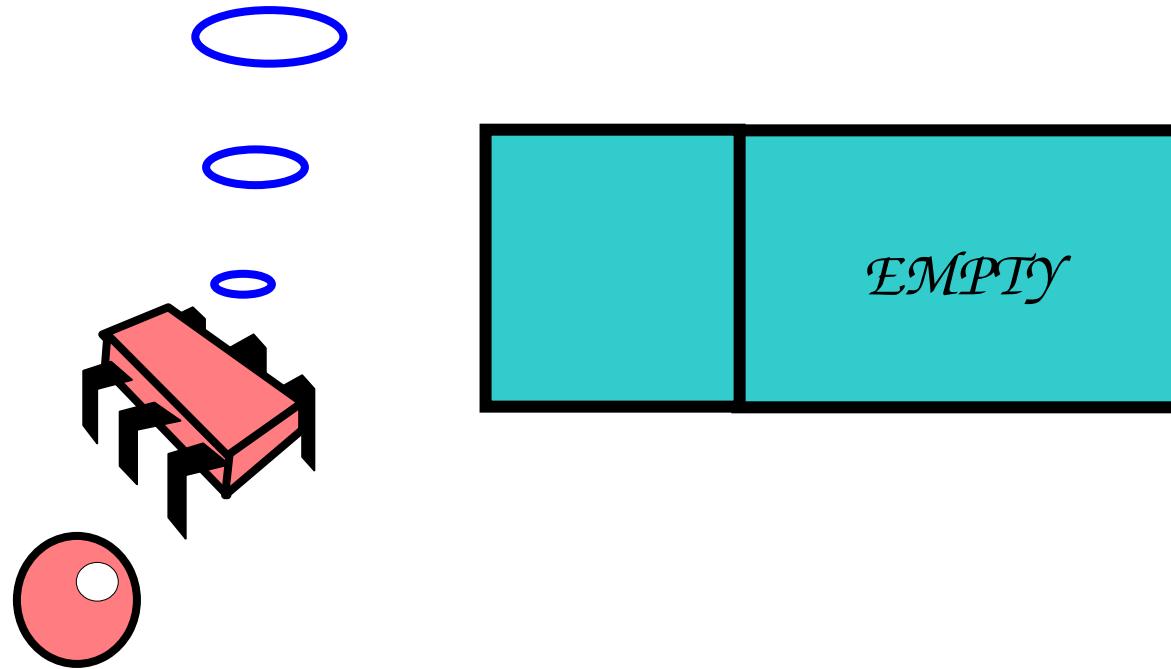
# If BUSY

- If the state is BUSY then two other threads are currently using the slot for an exchange

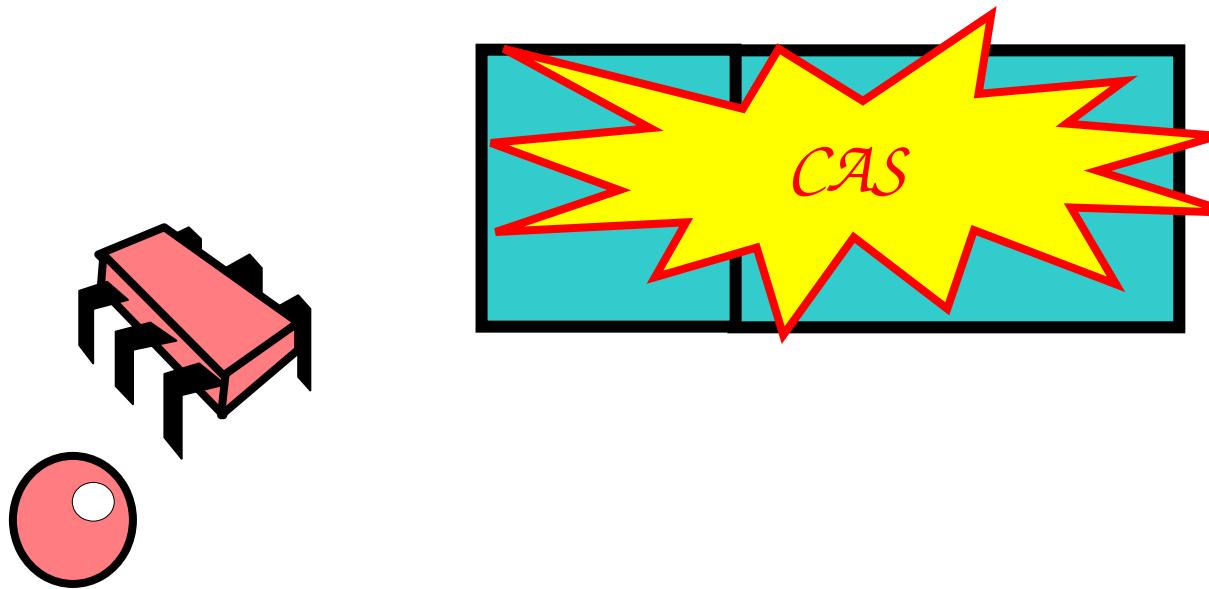
```
case BUSY:  
    break;
```

# Lock-free Exchanger

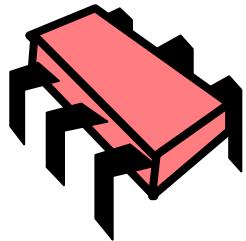
*Sees that slot is empty*



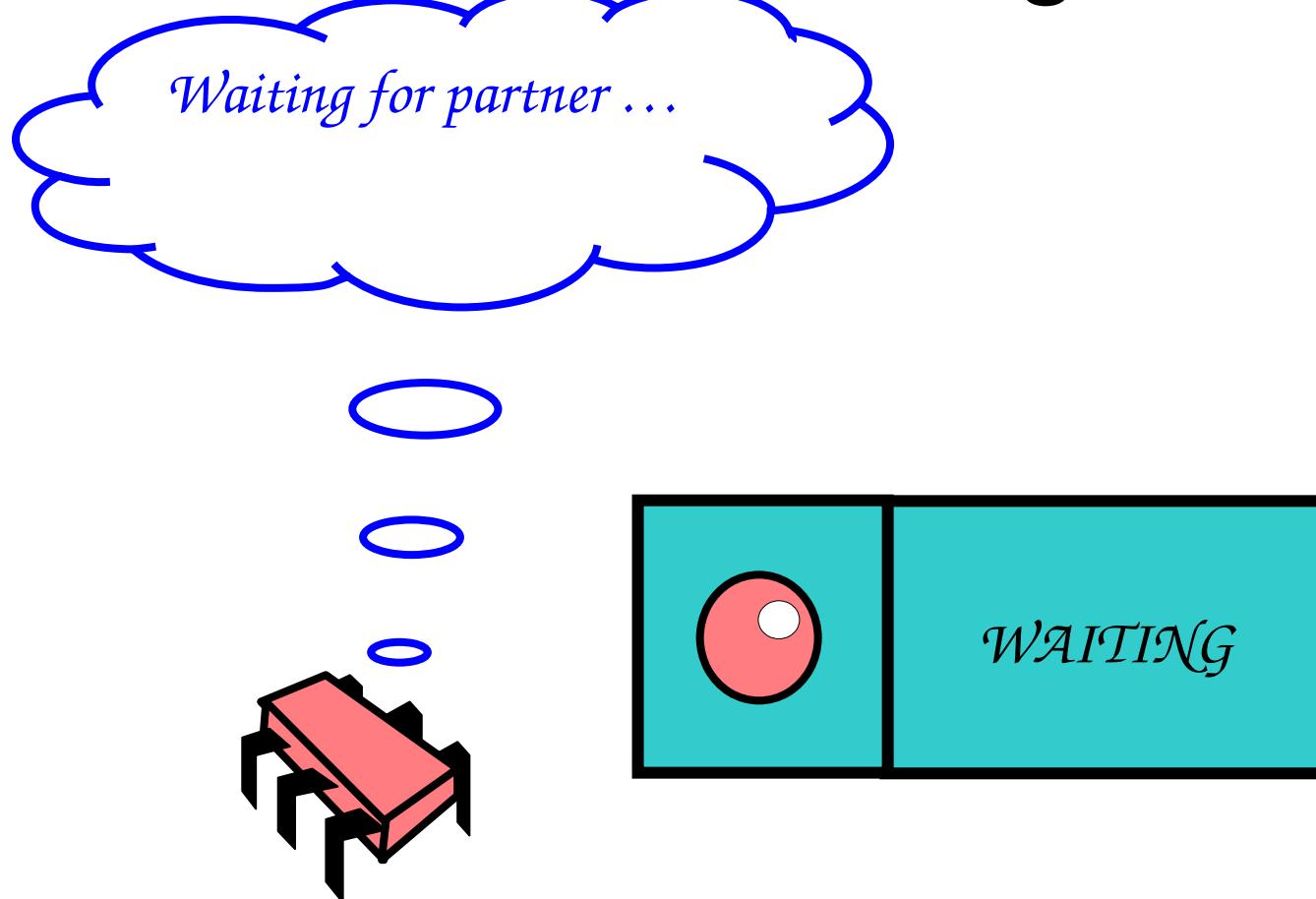
# Lock-free Exchanger



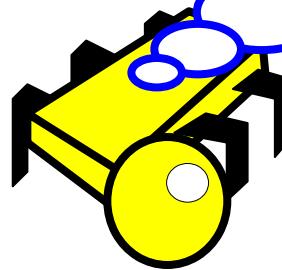
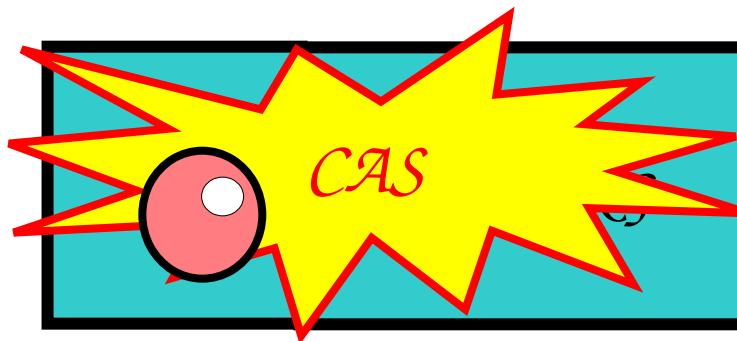
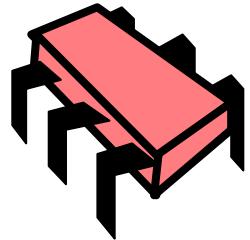
# Lock-free Exchanger



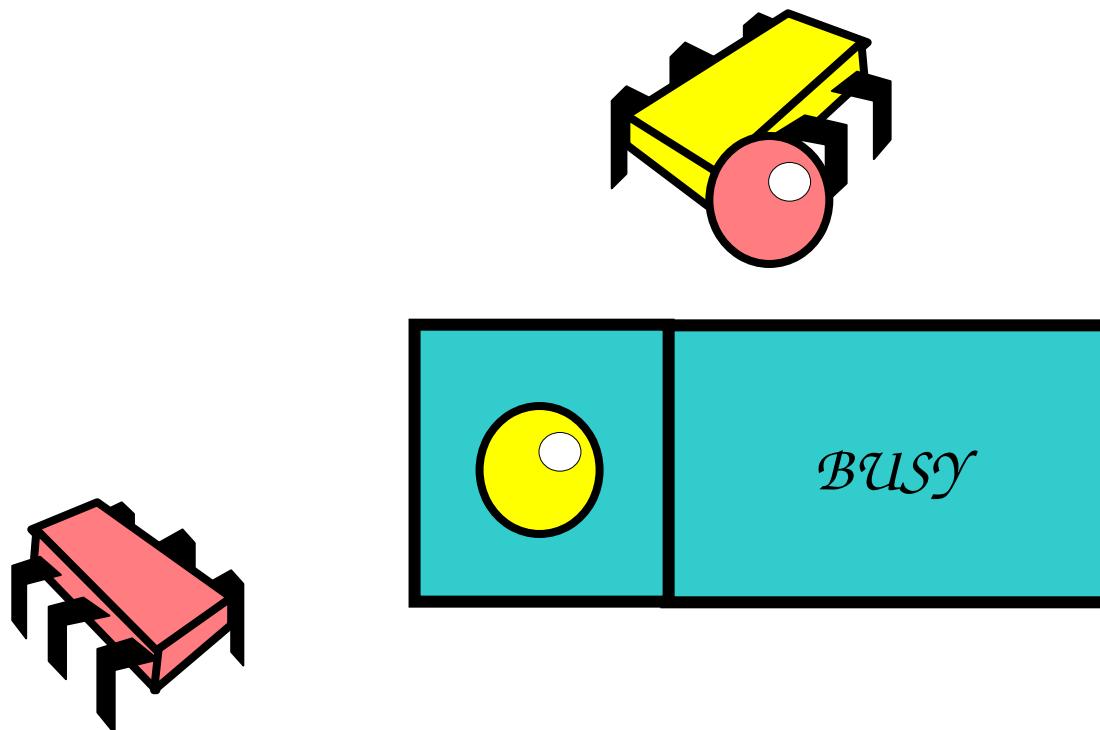
# Lock-free Exchanger



# Lock-free Exchange

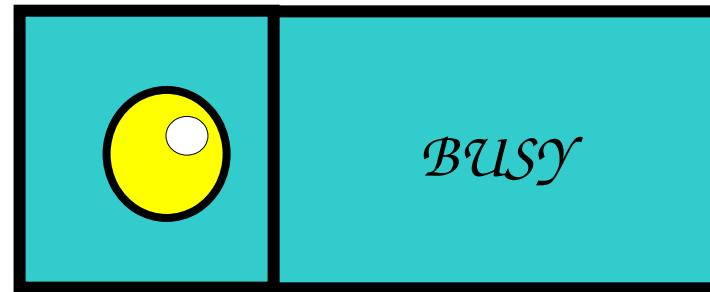
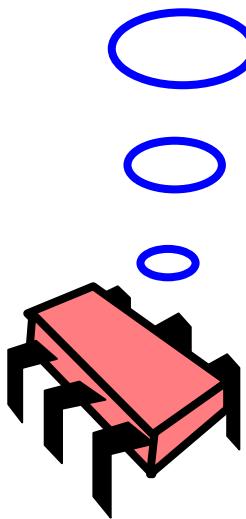


# Lock-free Exchanger



# Lock-free Exchanger

*Partner showed up,  
take item and reset to  
EMPTY*



# Back to EliminationBackoffStack

- An EliminationArray is implemented as an array of Exchanger objects
- A thread attempting to perform an exchange picks an array entry at random and call that entry's exchange() method

# EliminationBackoffStack

- Each thread first access the Stack
- If it fails, it attempts to eliminate its call using the array
  - It chooses a random array entry and calls exchange
- If it fails to eliminate due to:
  - Choosing an entry where the state is BUSY
  - Colliding with an unsuitable method call
- It accesses the Stack again
- And so on...

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

*First try to push element onto stack*

# Same method as Lock-free Stack

```
public boolean tryPush(Node node){  
    Node oldTop = top.get();  
    node.next = oldTop;  
    return (top.compareAndSet(oldTop, node))  
}
```

*Will return false if another thread succeeded = high contention*

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

*If high contention visit  
Elimination Array and  
try to eliminate call*

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

*If push method returns null, exchange was successful*

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

*If not, try again*

# EliminationBackoffStack

```
public T pop() throws EmptyException {  
    while (true) {  
        Node returnNode = tryPop();  
        if (returnNode != null)  
            return returnNode.value;  
        else try {  
            T otherValue =  
                eliminationArray.visit(null);  
            if (otherValue != null)  
                return otherValue;  
        } catch (TimeoutException e) {}  
    }  
}
```

# EliminationBackoffStack

```
public T pop() throws EmptyException {  
    while (true) {  
        Node returnNode = tryPop();  
        if (returnNode != null)  
            return returnNode.value;  
        else try {  
            T otherValue =  
                eliminationArray.visit(null);  
            if (otherValue != null)  
                return otherValue;  
        } catch (TimeoutException e) {}  
    }  
}
```

*First try to pop element from stack*

# Same method as Lock-free Stack

```
protected Node tryPop() throws EmptyException
{
    Node oldTop = top.get();
    if (oldTop == null)
        throw new EmptyException();
    Node newTop = oldTop.next;
    if (top.compareAndSet(oldTop, newTop))
        return oldTop;
    else
        return null;
}
```

*Return null if failed due to high contention*

# EliminationBackoffStack

```
public T pop() throws EmptyException {  
    while (true) {  
        Node returnNode = tryPop();  
        if (returnNode != null)  
            return returnNode.value;  
        else try {  
            T otherValue =  
                eliminationArray.visit(null);  
            if (otherValue != null)  
                return otherValue;  
        } catch (TimeoutException e) {}  
    }  
}
```

*Attempt to eliminate – add null to elimination array*

# EliminationBackoffStack

```
public T pop() throws EmptyException {  
    while (true) {  
        Node returnNode = tryPop();  
        if (returnNode != null)  
            return returnNode.value;  
        else try {  
            T otherValue =  
                eliminationArray.visit(null);  
            if (otherValue != null)  
                return otherValue;  
        } catch (TimeoutException e) {}  
    }  
}
```

*Successful exchange for pop method if non-null value returned*

# Performance

- At low levels of contention the performance of EliminationBackoffStack is comparable to LockFreeStack
- However at high contention the number of successful eliminations will increase, allowing more operations to complete in parallel