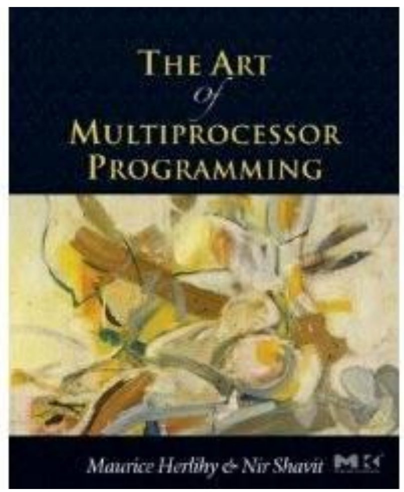
Introduction to Multiprocessor Synchronization

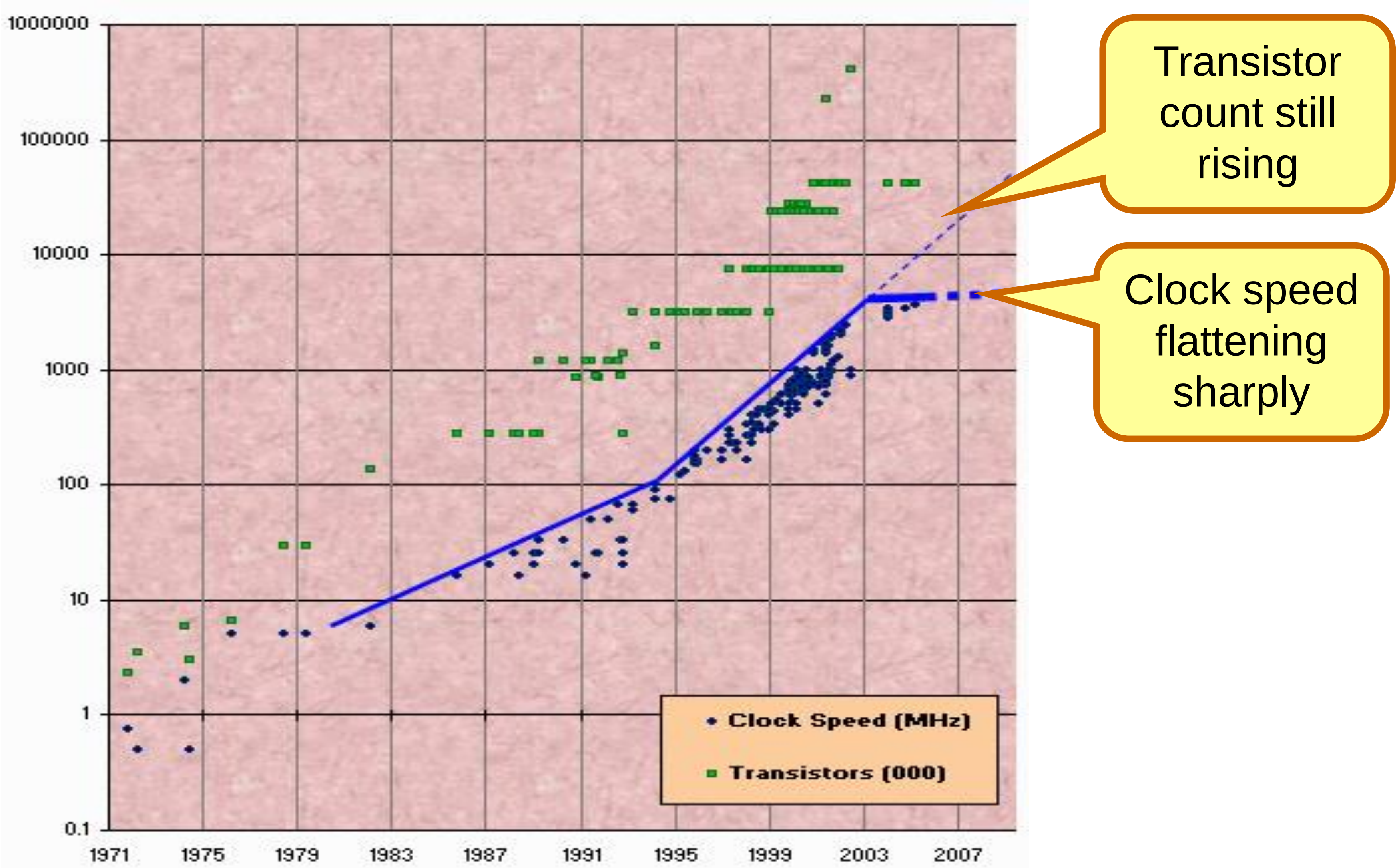


Maurice Herlihy

BROWN

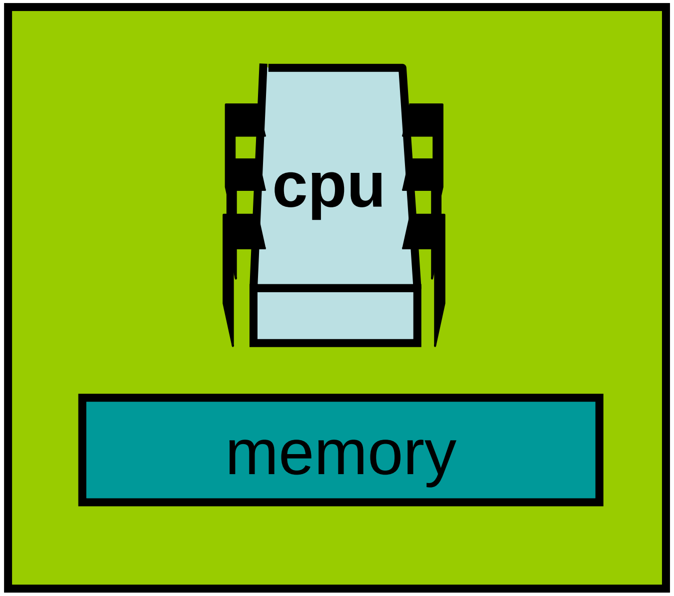
http://cs.brown.edu/courses/cs176/lectures.shtml

Moore's Law

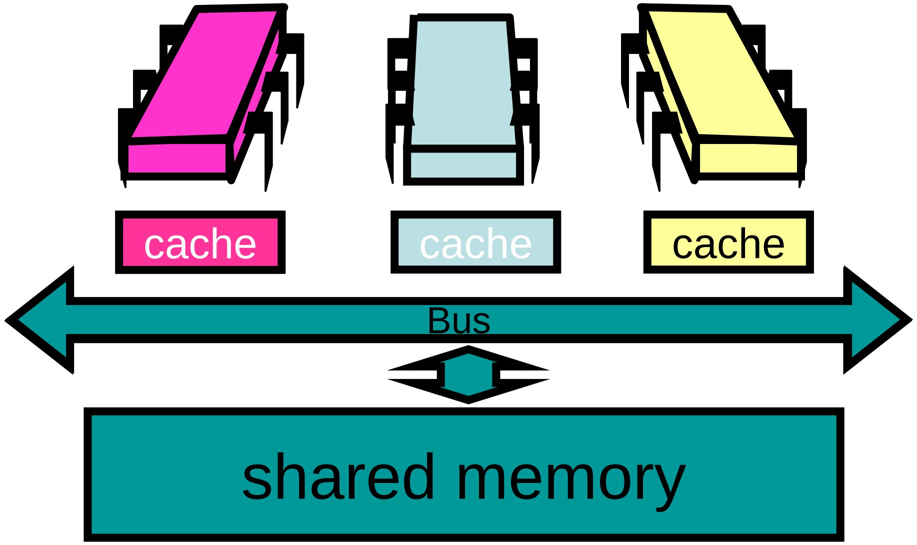


1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
Clock Speed (MHz)
Transistors (000)
Transistor count still rising
Clock speed flattening sharply

Once roamed the Earth: the Uniprocesor

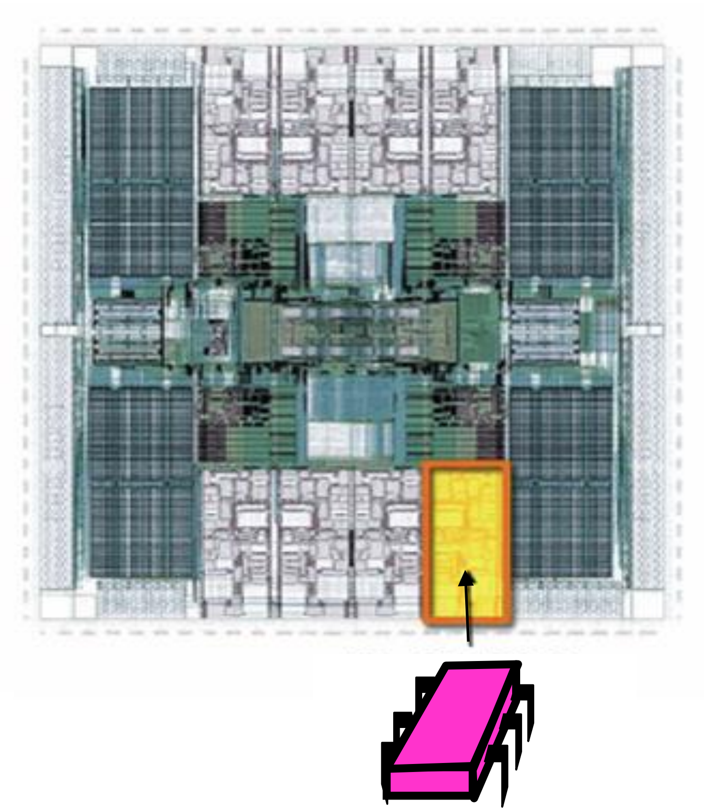


Endangered:
The Shared Memory Multiprocessor (SMP)



Meet he New Boss: The Multicore Processor (CMP)

All on the same chip

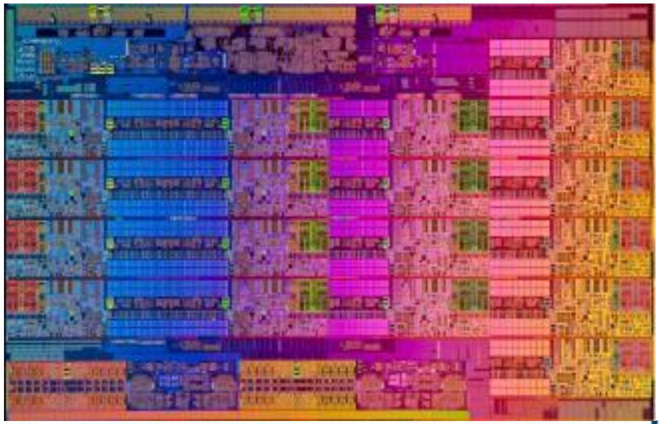


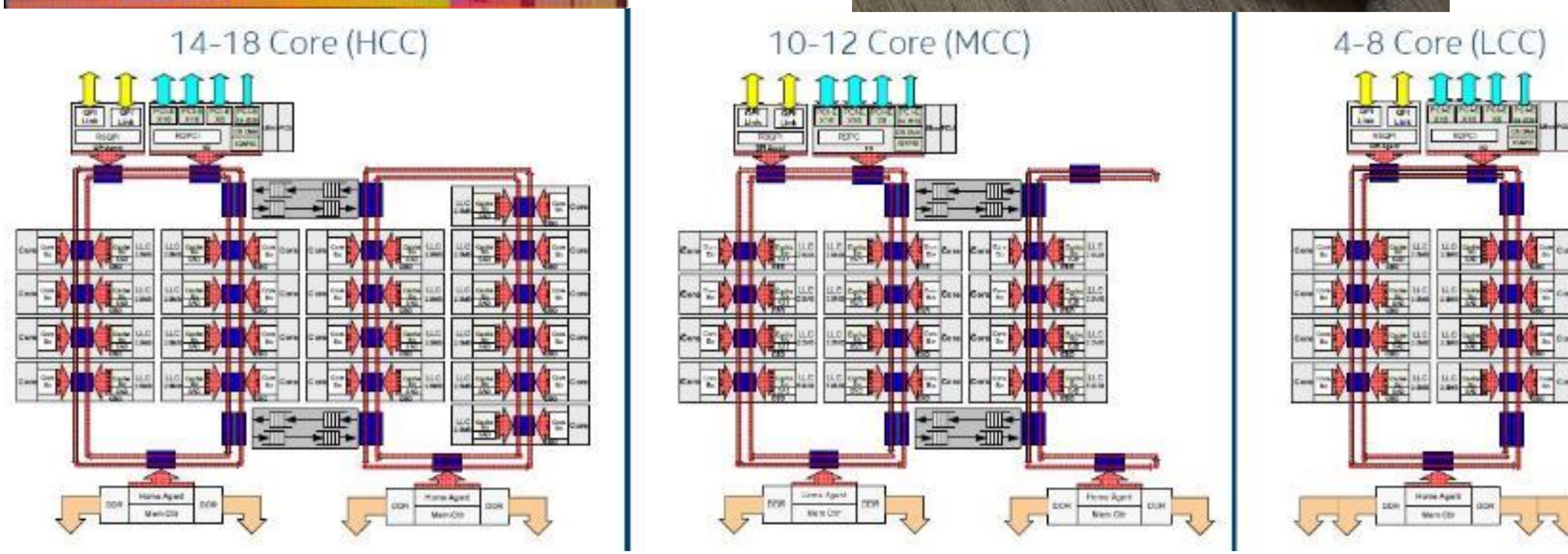
Oracle Niagara Chip

Turing Cluster

- 24 Compute Nodes in two 12 node 3U blades. Each node has **one 8-core AMD Opteron 3380 processor @ 2.6GHz**, 32GB RAM, 2TB HDD, Gigabit Ethernet port
- 1 Head Node with **one 6-core Intel Xeon E5-2620 v3 processor @ 2.40GHz**, 48GB RAM, 1+4TB HDD, Gigabit Ethernet ports
- One 24 port L2 Gigabit Ethernet switch
- Running CentOS, MPI, PBS and Apache Hadoop/Yarn
- Mounted on a 24U Rack
- http://cds.iisc.ac.in/internal-resources/computing-resources/

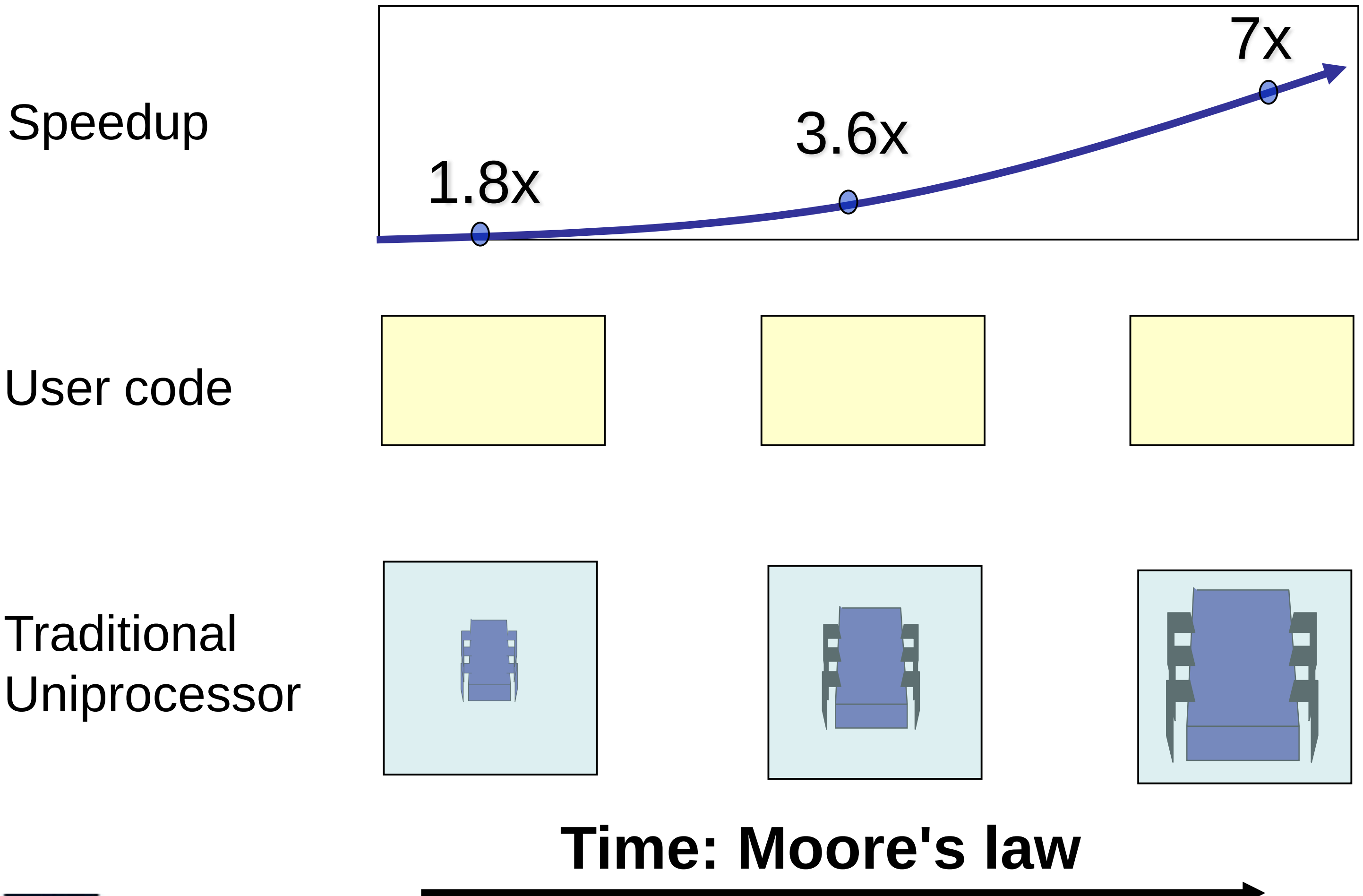
Turing Cluster: Xeon E5-2620 v3



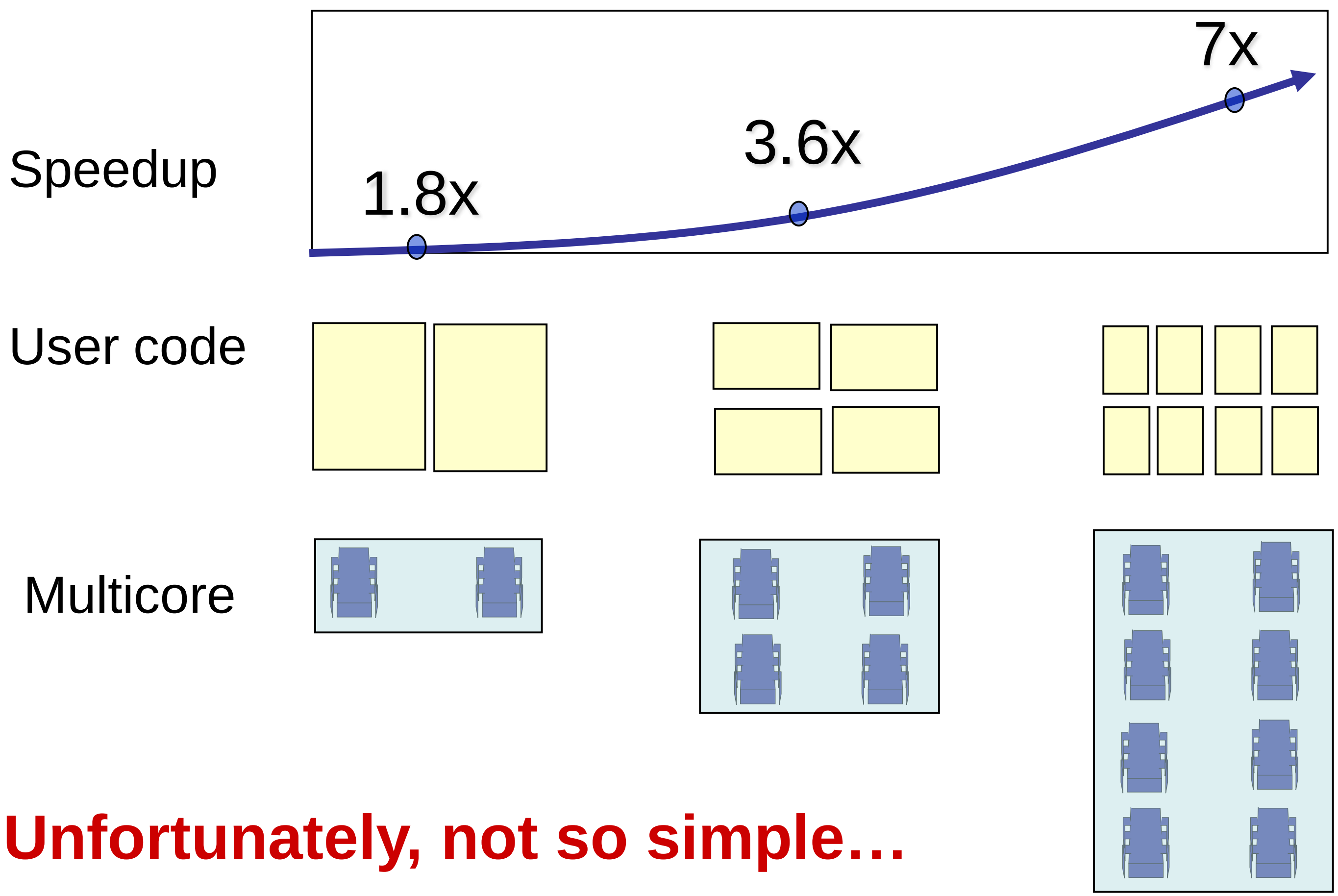


http://2eof2j3oc7is20vt9q3g7tlo5xe.wpengine.netdna-cdn.com/wp-content/uploads/2014/09/intel-xeon-e5-v3-block-diagram-detailed.jpg
http://www.enterprisetech.com/wp-content/uploads/2014/09/intel-xeon-e5-v3-die-shot.jpg

Traditional Scaling Process



Ideal Multicore Scaling Process



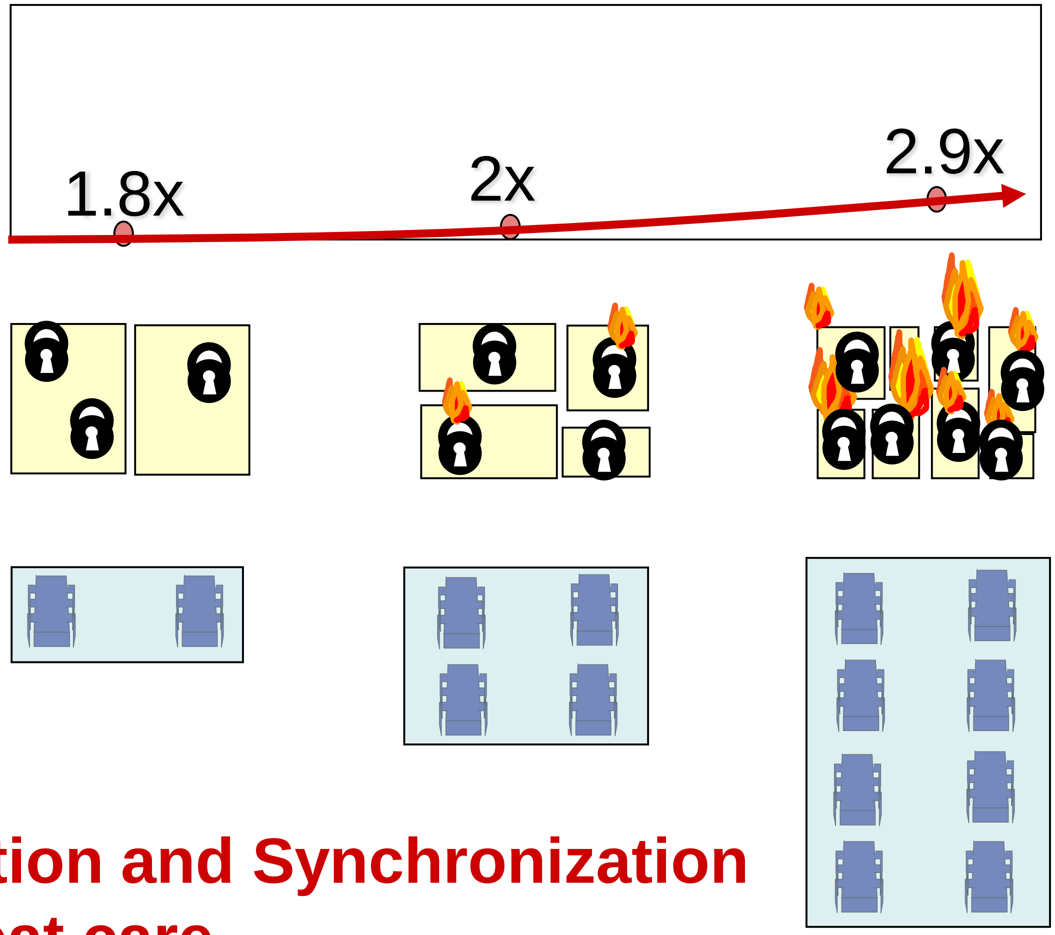
Unfortunately, not so simple…

Actual Multicore Scaling Process

Speedup

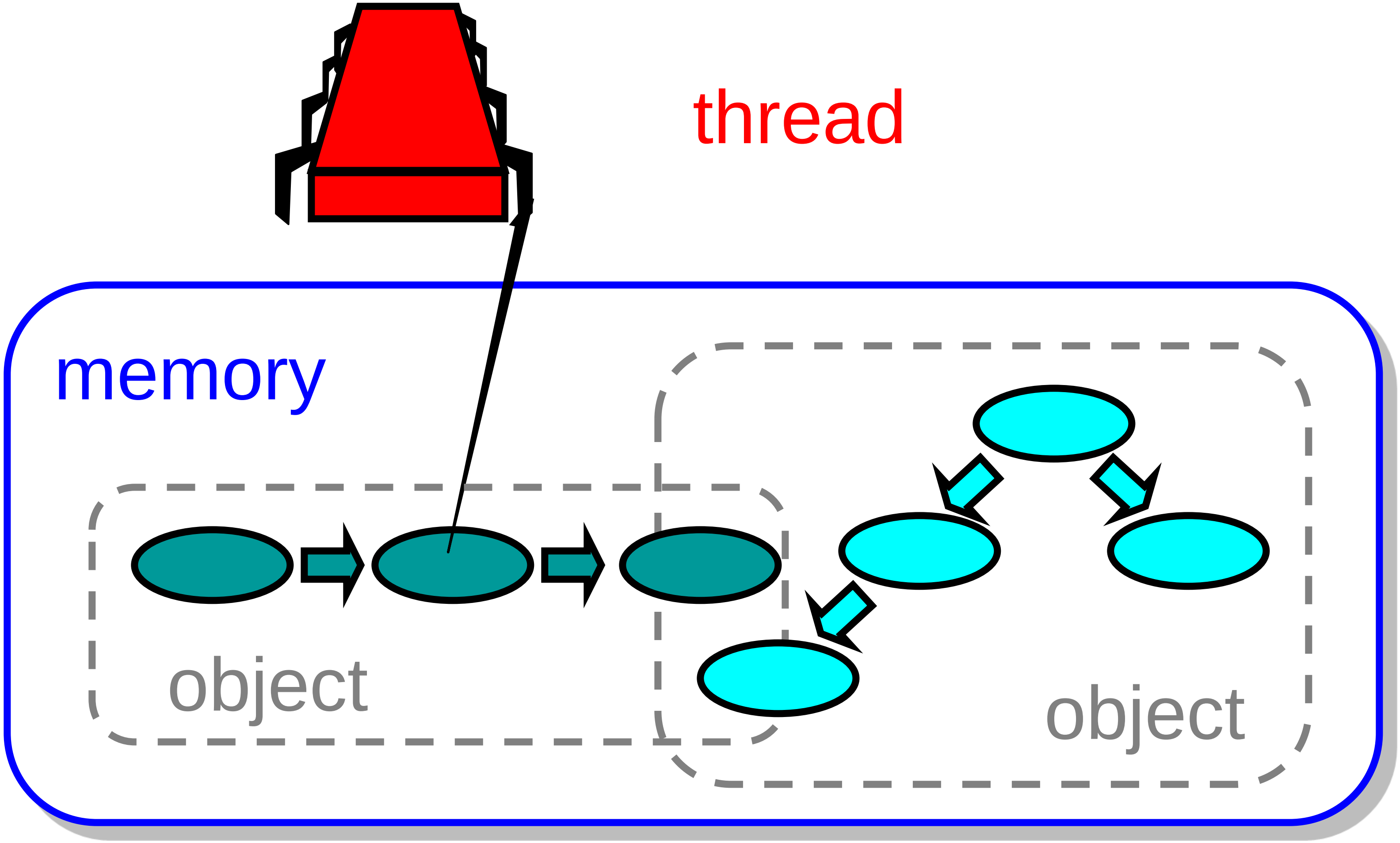
User code

Multicore

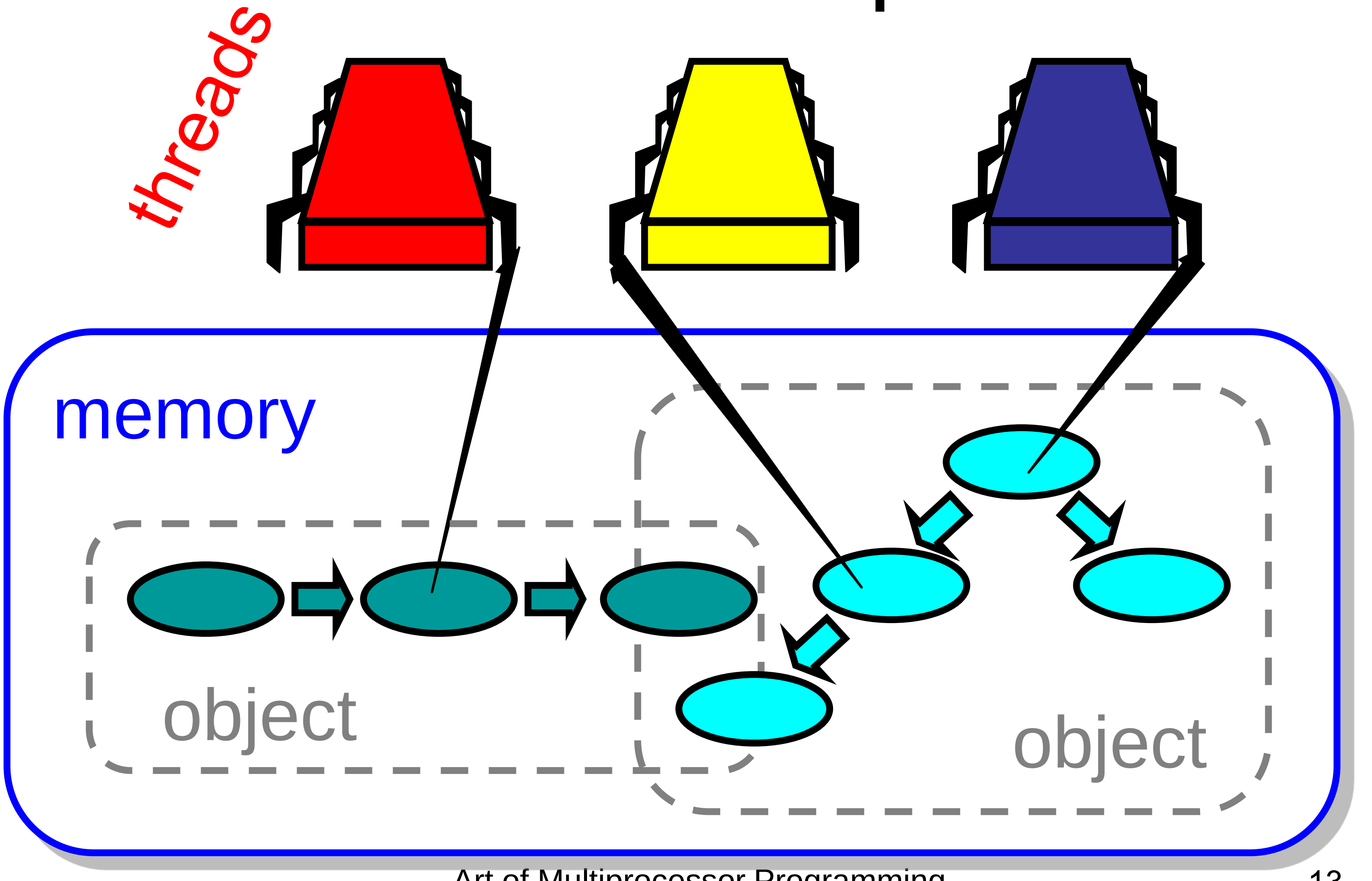


Parallelization and Synchronization require great care…

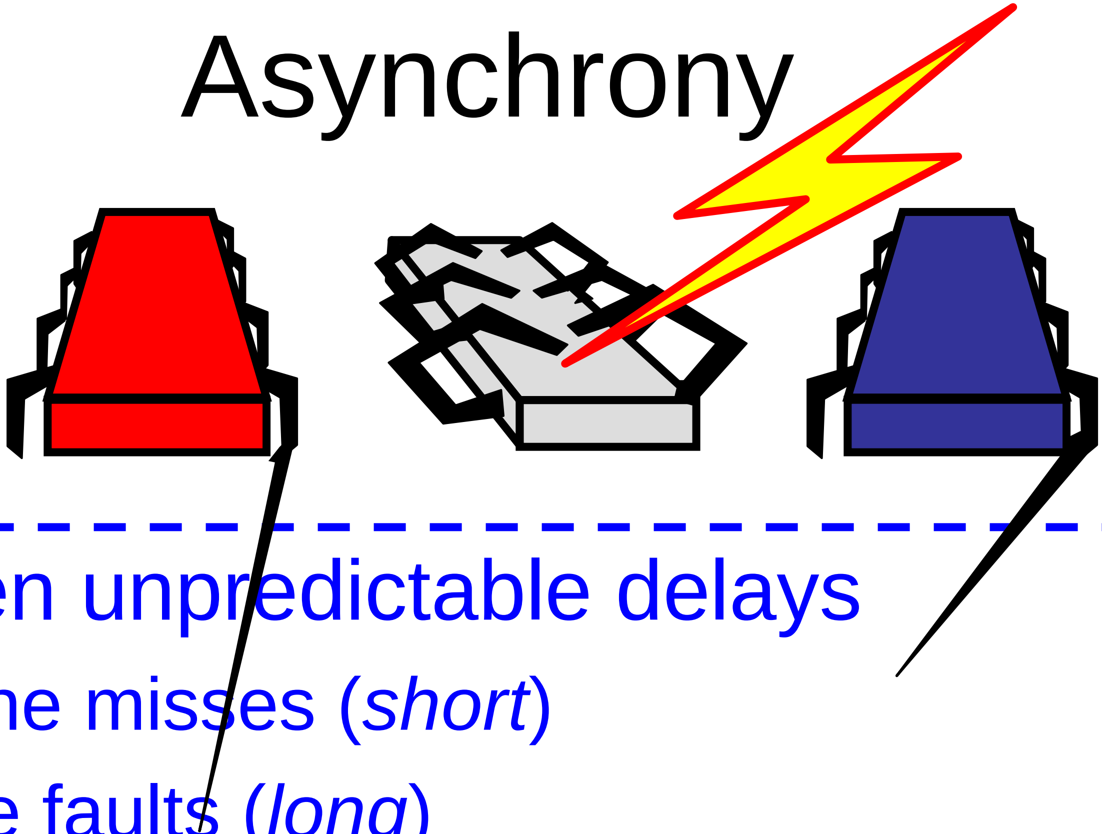
Sequential Computation



Concurrent Computation



Asynchrony



- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)

Model Summary

- Multiple *threads*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays

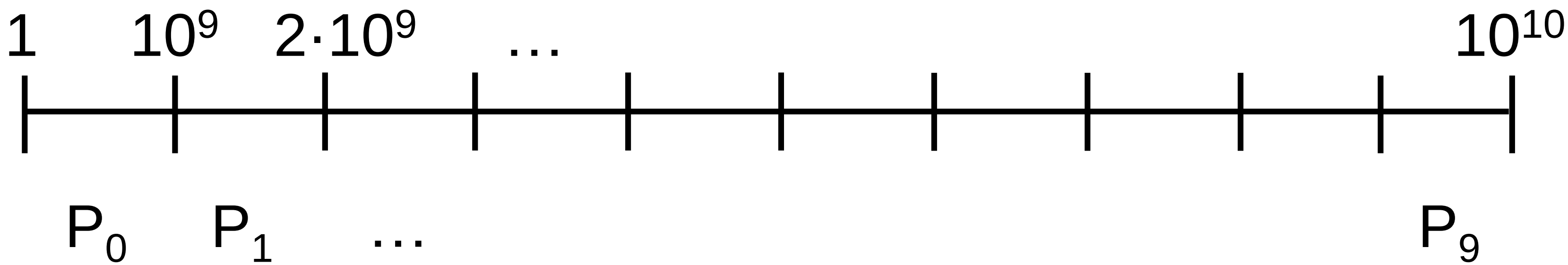
Concurrency Jargon

- Hardware
  - Processors
- Software
  - Threads, processes
- Sometimes OK to confuse them, sometimes not.

Parallel Primality Testing

- Challenge
  - Print primes from 1 to 10^{10}
- Given
  - Ten-processor multiprocessor
  - One thread per processor
- Goal
  - Get ten-fold speedup (or close)

Load Balancing



- Split the work evenly
- Each thread tests range of 10^9

Procedure for Thread *i*

```
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*10^9+1, j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

Issues

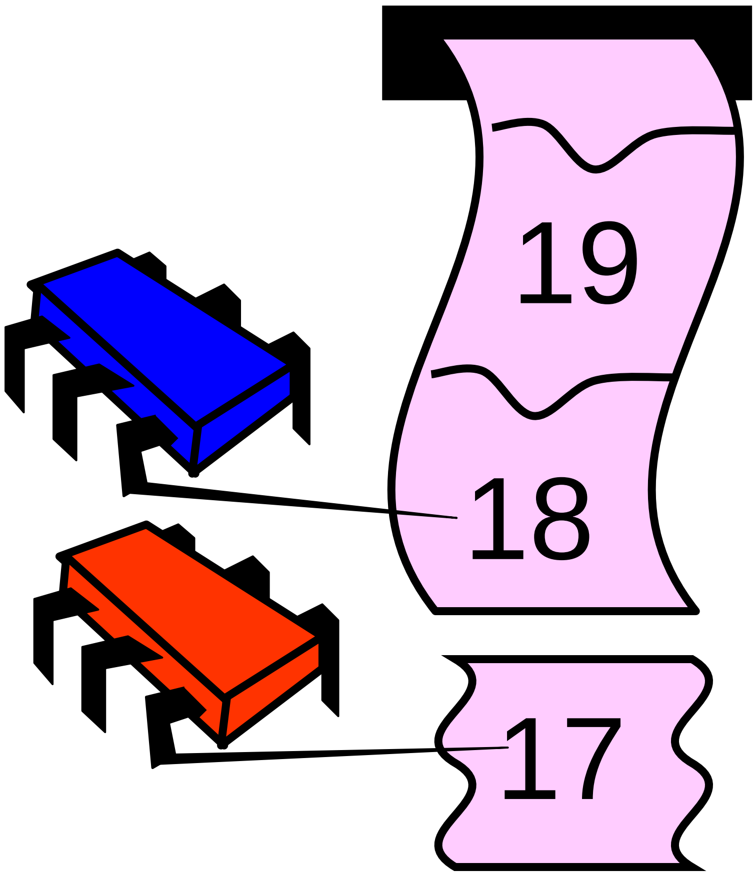
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict

Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need *dynamic* load balancing

rejected

Shared Counter



each thread takes a number

Procedure for Thread *i*

```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Procedure for Thread *i*

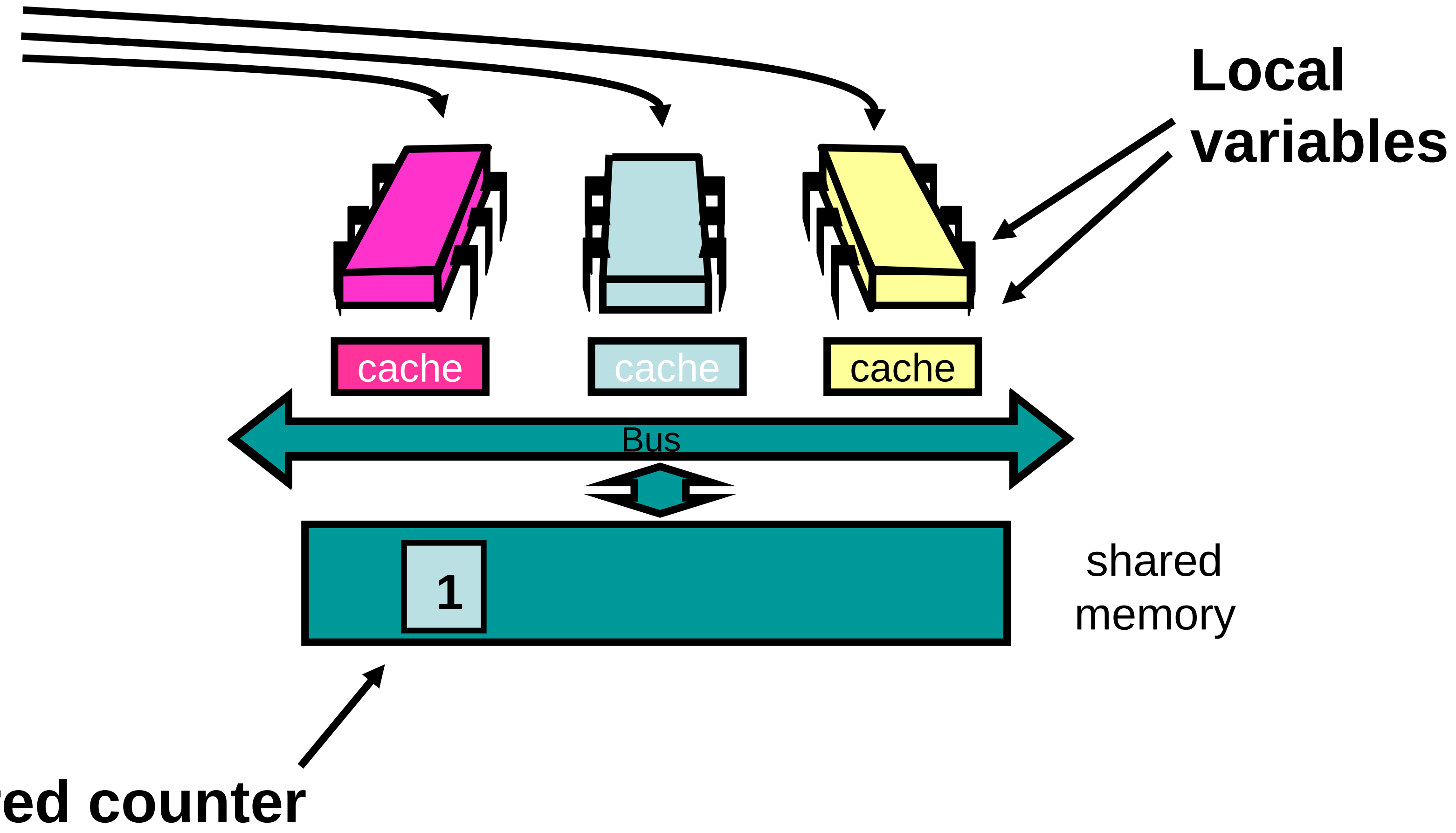
```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Shared counter object

Where Things Reside

```
void primePrint {
  int i =
ThreadID.get(); // IDs
in {0..9}
  for (j = i*10^9+1,
j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

code



Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Stop when every value taken

Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Increment & return each new value

Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

**OK for single thread,
not for concurrent threads**

What It Means

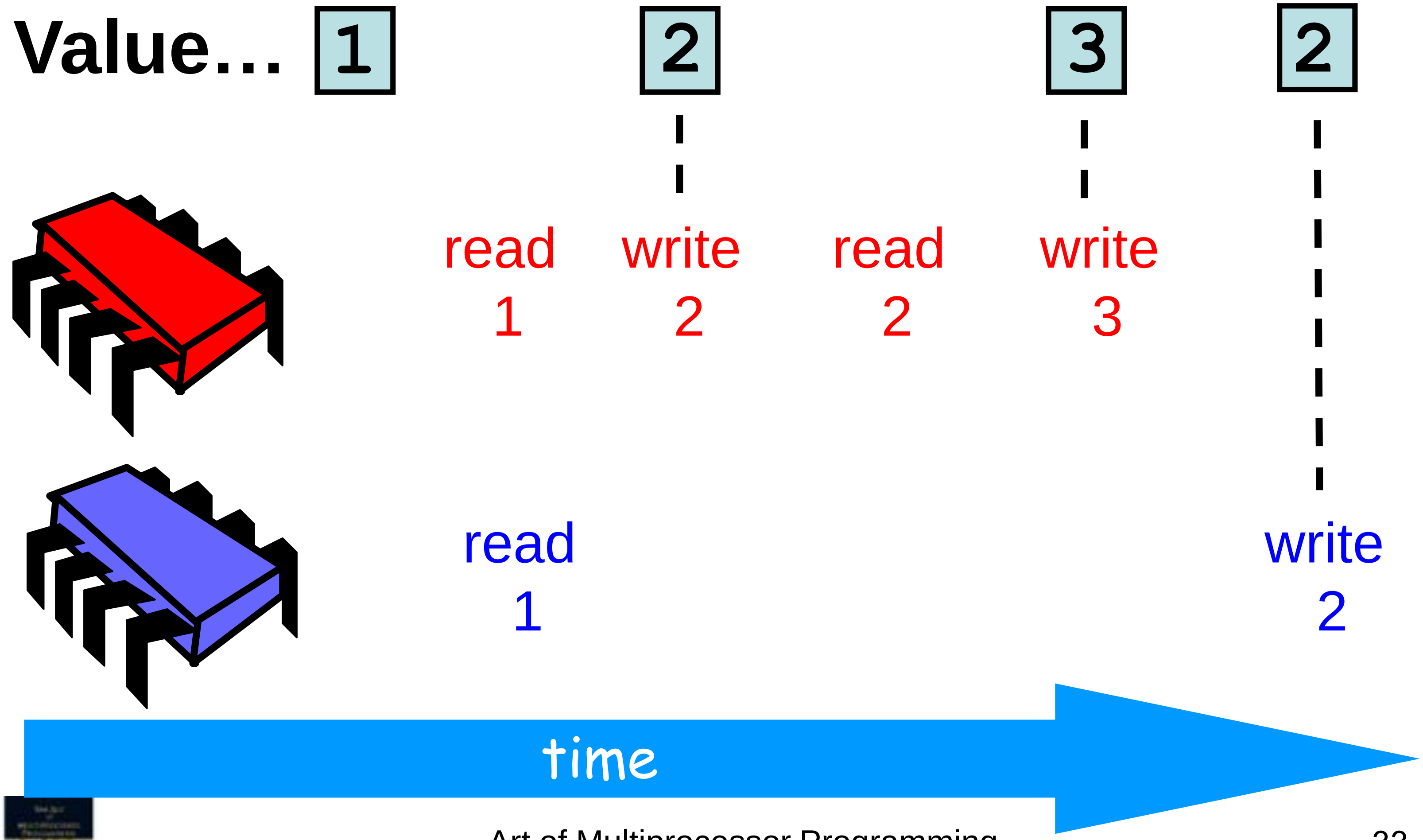
```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

What It Means

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

```
temp  = value;
value = temp + 1;
return temp;
```

Not so good…



Challenge

```
public class Counter {
  private long value;

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

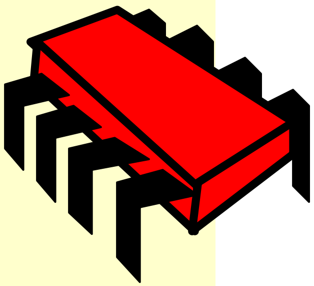
Challenge

```
public class Counter {
  private long value;

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

Make these steps *atomic* (indivisible)

Hardware Solution

```
public class Counter {
  private long value;

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



ReadModifyWrite() instruction

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

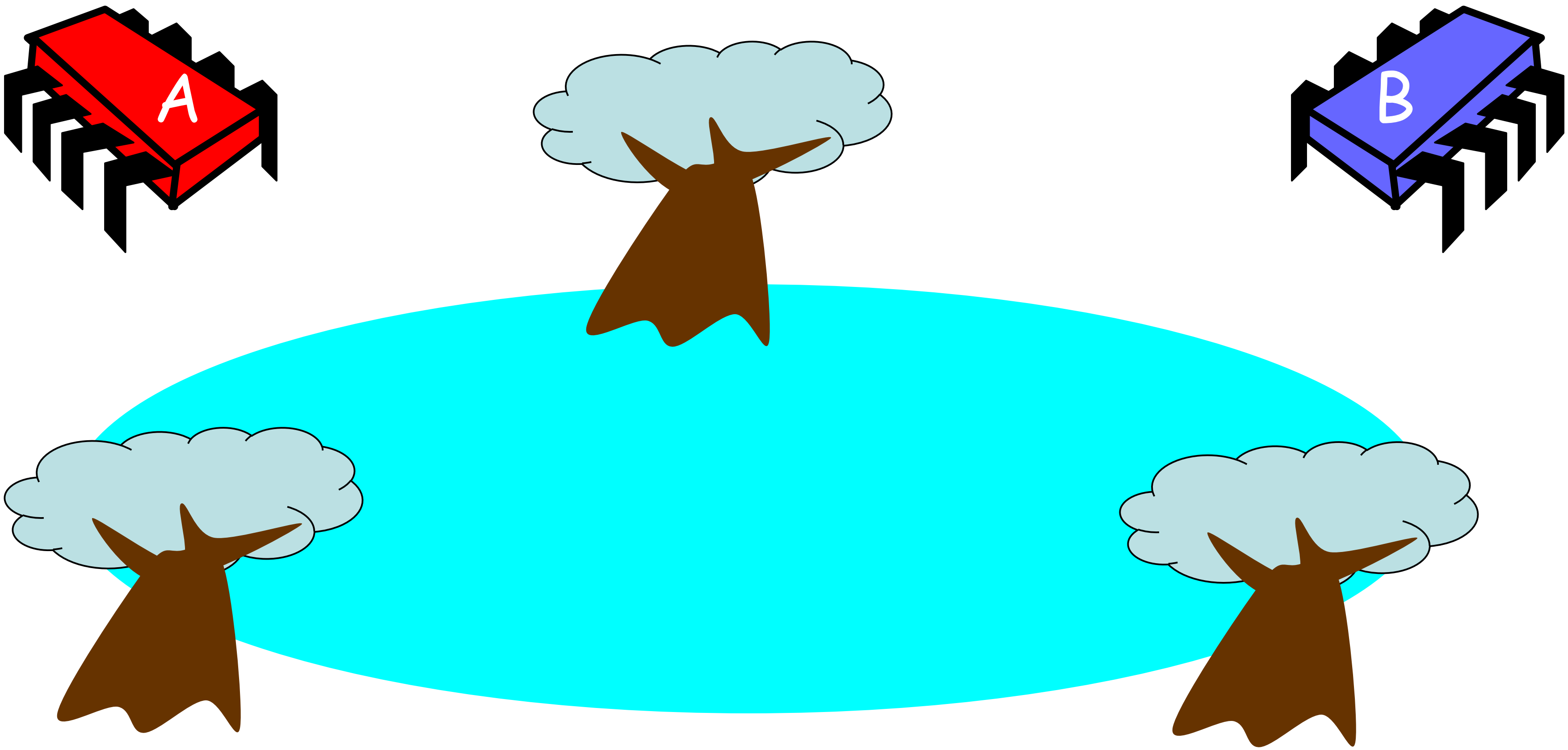
Synchronized block

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
    }
    return temp;
  }
}
```

Mutual Exclusion

Mutual Exclusion, or "Alice & Bob share a pond"



Alice has a pet

Bob has a pet

The Problem

Formalizing the Problem

- Two types of formal properties in asynchronous computation:
- Safety Properties
  - Nothing bad happens ever
- Liveness Properties
  - Something good happens eventually

Formalizing our Problem

- Mutual Exclusion
  - Both pets never in pond simultaneously
  - This is a ***safety*** property
- No Deadlock
  - if only one wants in, it gets in
  - if both want in, one gets in
  - This is a ***liveness*** property

Simple Protocol

- Idea
  - Just look at the pond
- Gotcha
  - Not atomic
  - Trees obscure the view

Interpretation

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

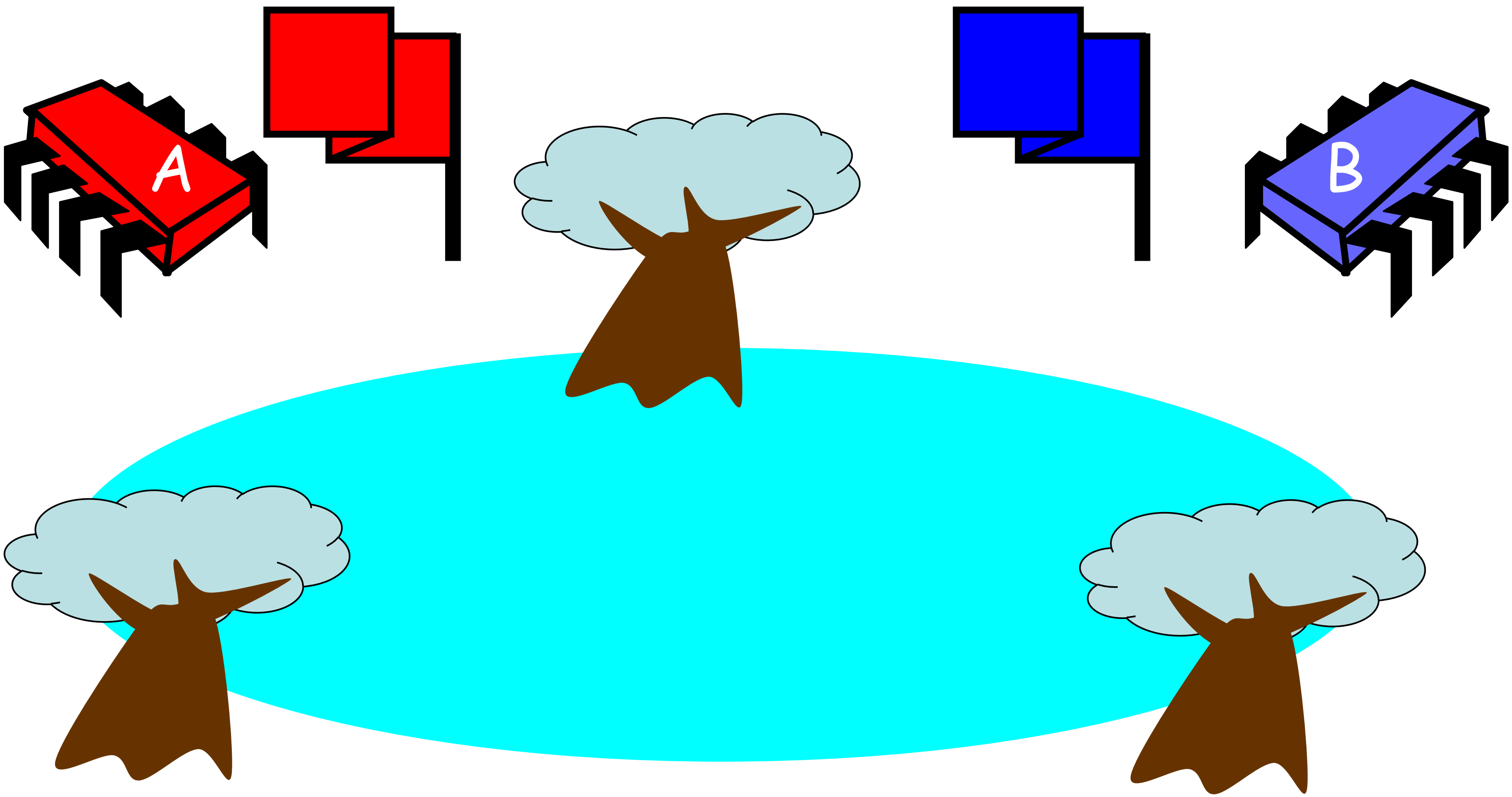
Cell Phone Protocol

- Idea
  - Bob calls Alice (or vice-versa)
- Gotcha
  - Bob takes shower
  - Alice recharges battery
  - Bob out shopping for pet food …

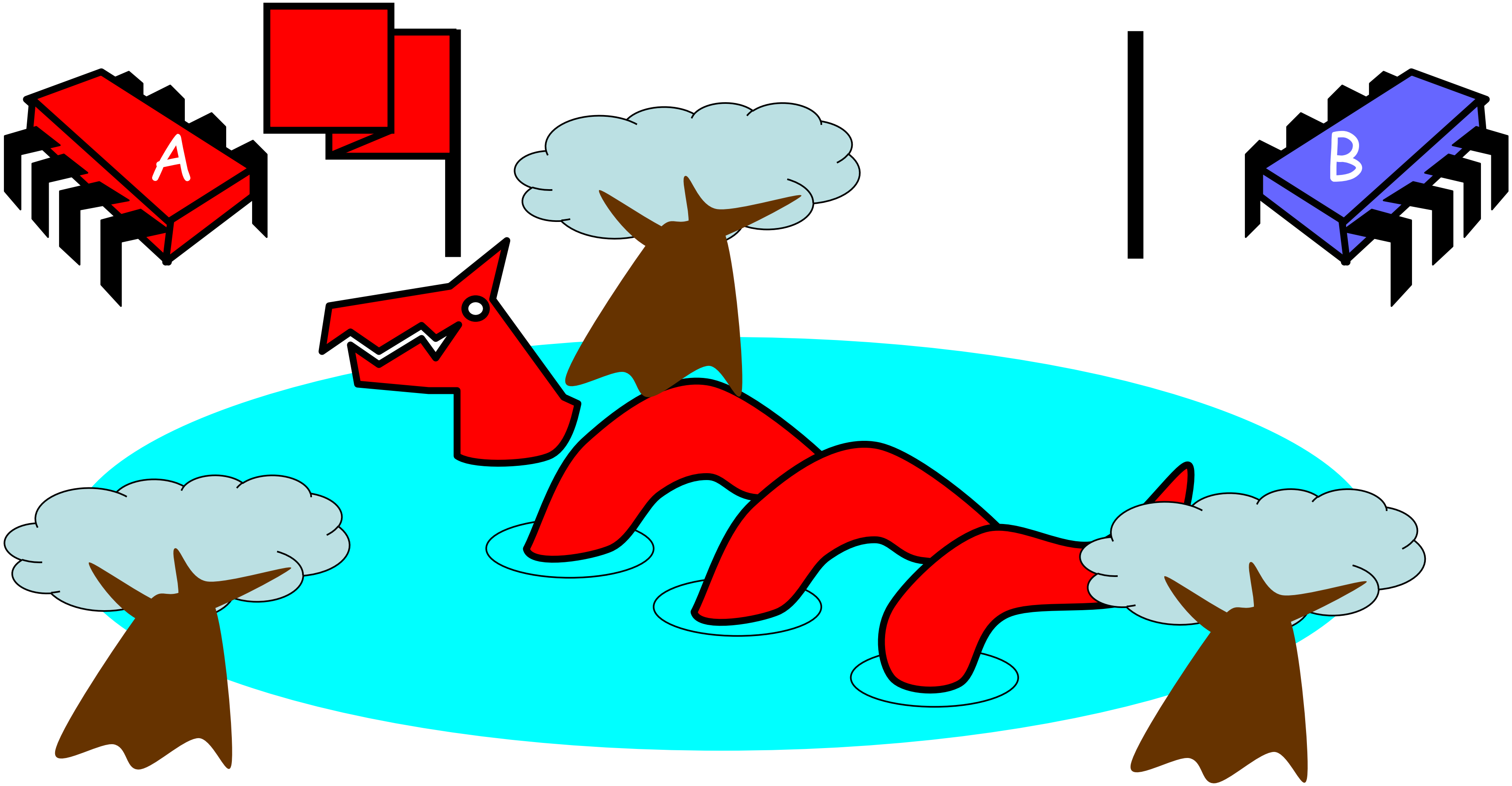
Interpretation

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

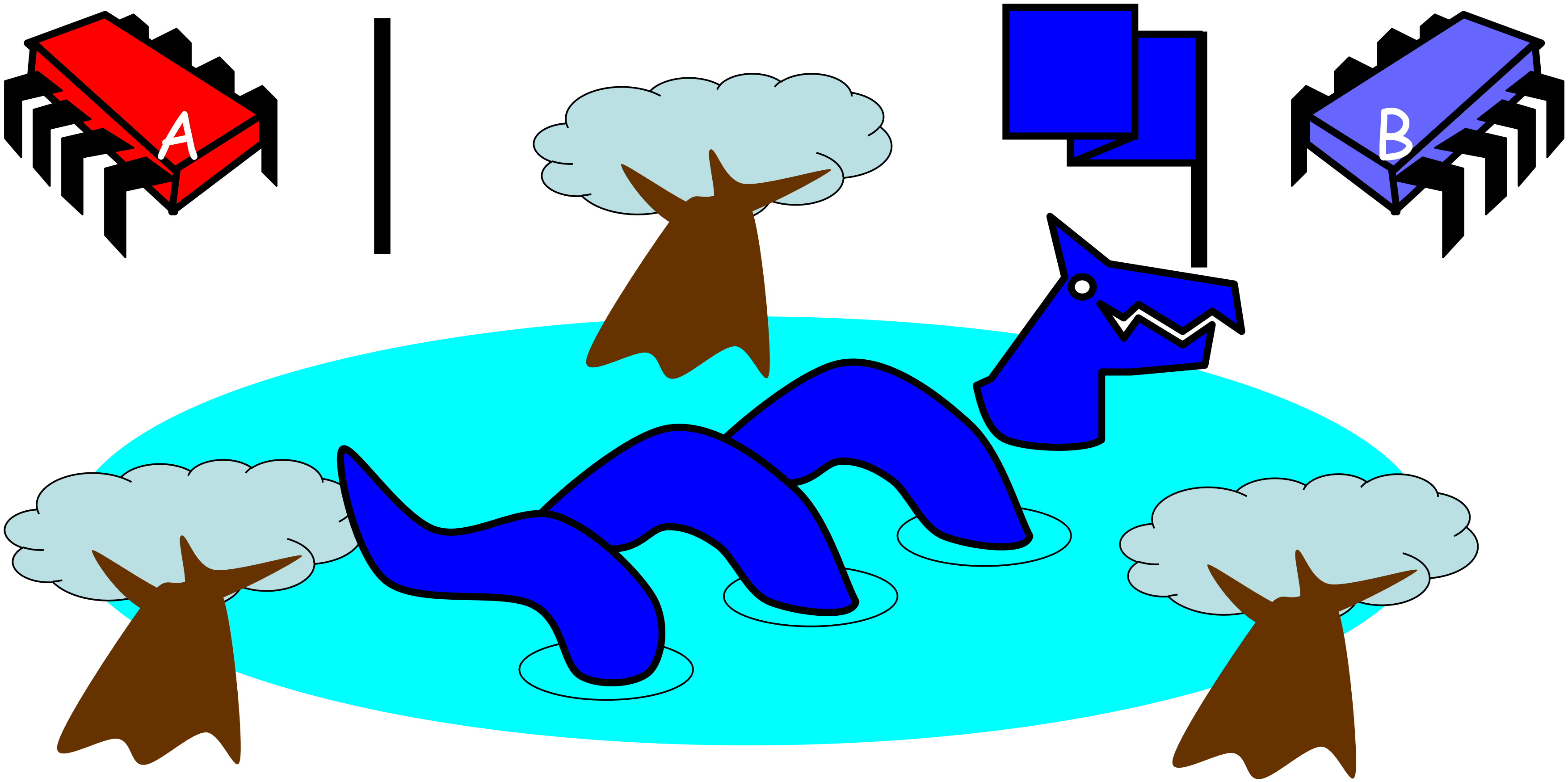
Flag Protocol



Alice's Protocol (sort of)



Bob's Protocol (sort of)



Alice's Protocol

- Raise flag
- Wait until Bob's flag is down
- Unleash pet
- Lower flag when pet returns

Bob's Protocol

- Raise flag
- Wait until Alice's flag is down
- Unleash pet
- Lower flag when pet returns

danger!

Bob's Protocol (2nd 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

Bob defers to Alice

- 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

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

Remarks

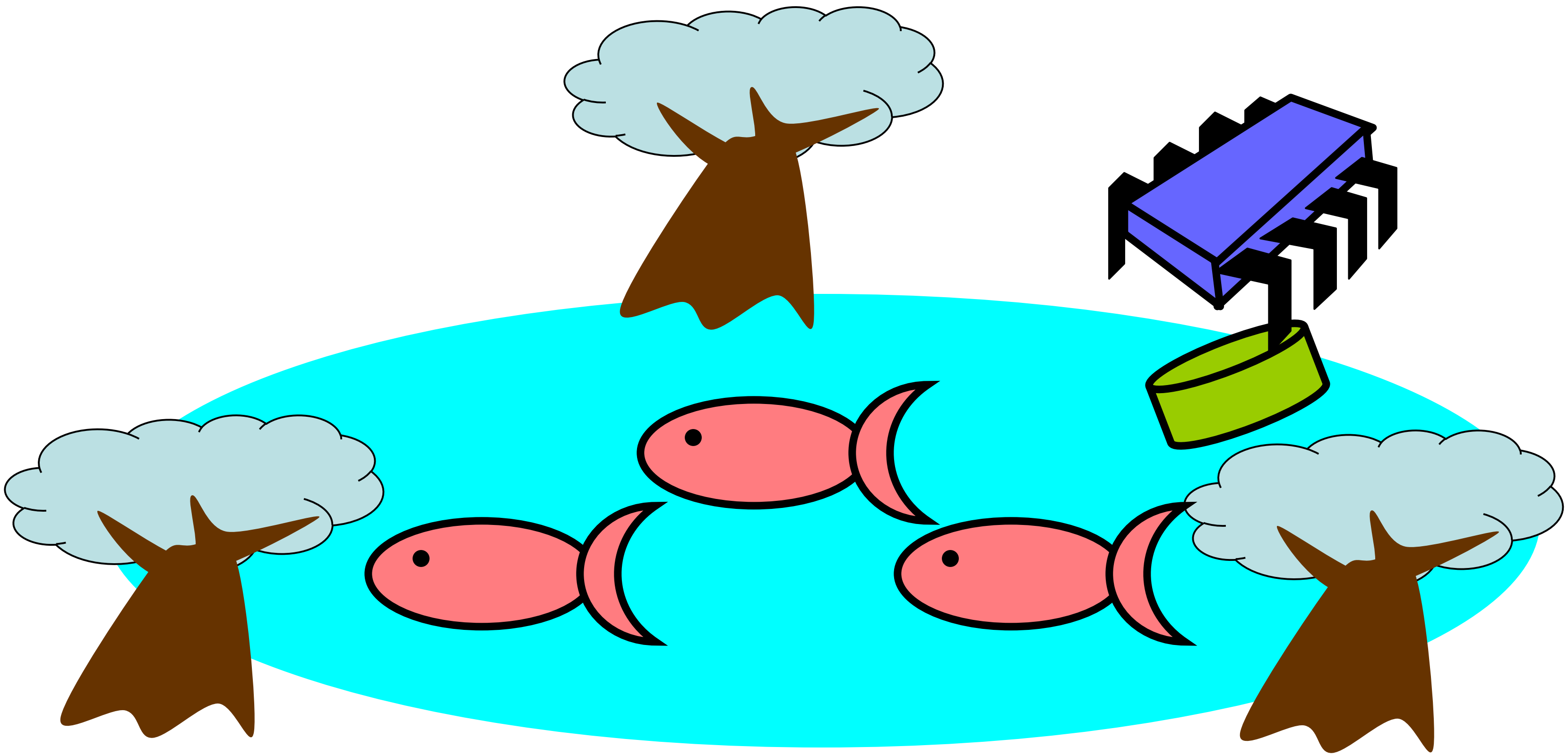
- Protocol is *unfair*
  - Bob's pet might never get in
- Protocol uses *waiting*
  - If Bob is eaten by his pet, Alice's pet might never get in

The Fable Continues

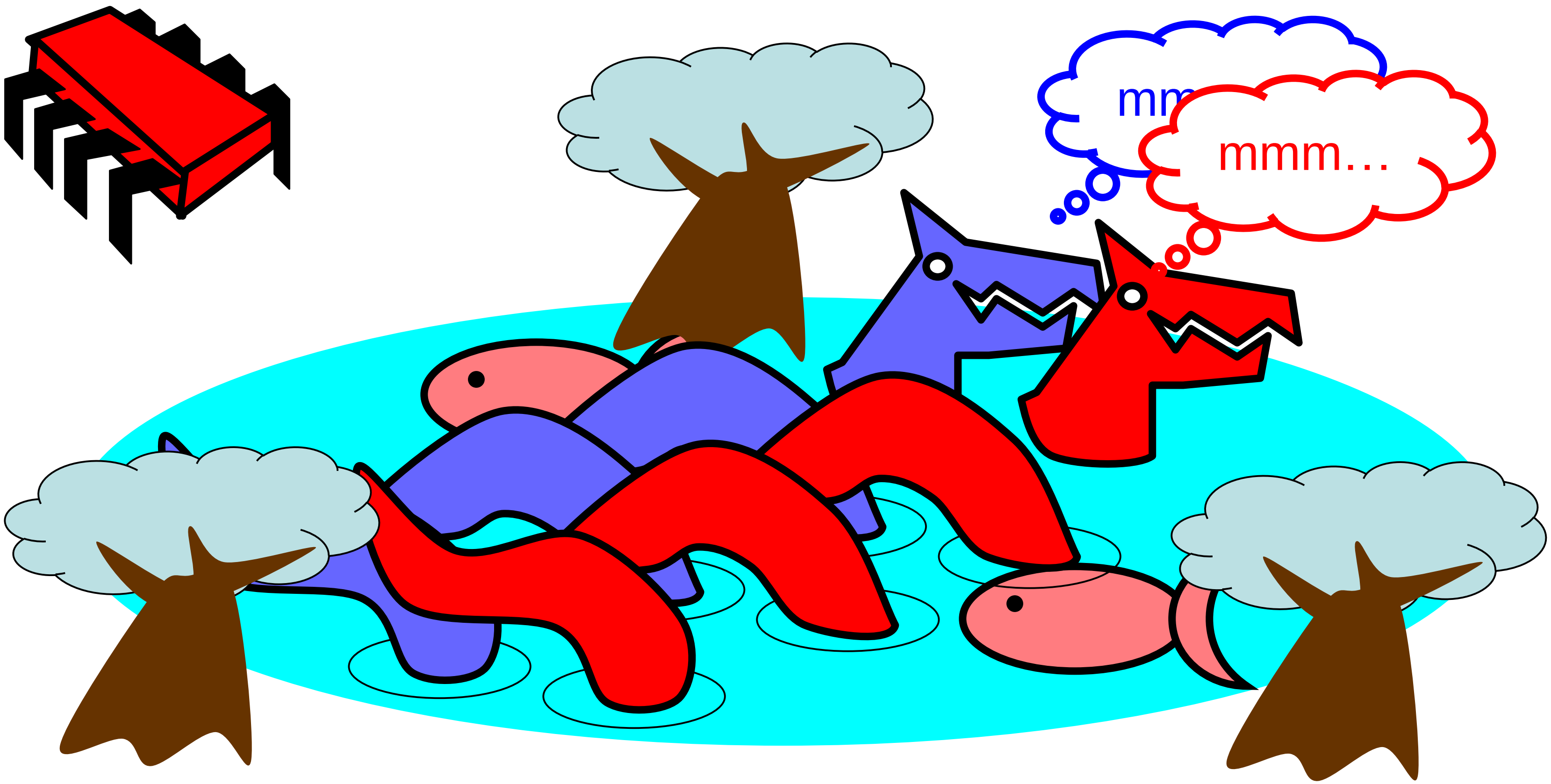
- Bob falls ill, cannot tend to the pets
- She gets the pets
  - Pets get along fine ☺
- But Bob has to feed them

- Producer-Consumer Problem

Bob Puts Food in the Pond



Alice releases her pets to Feed



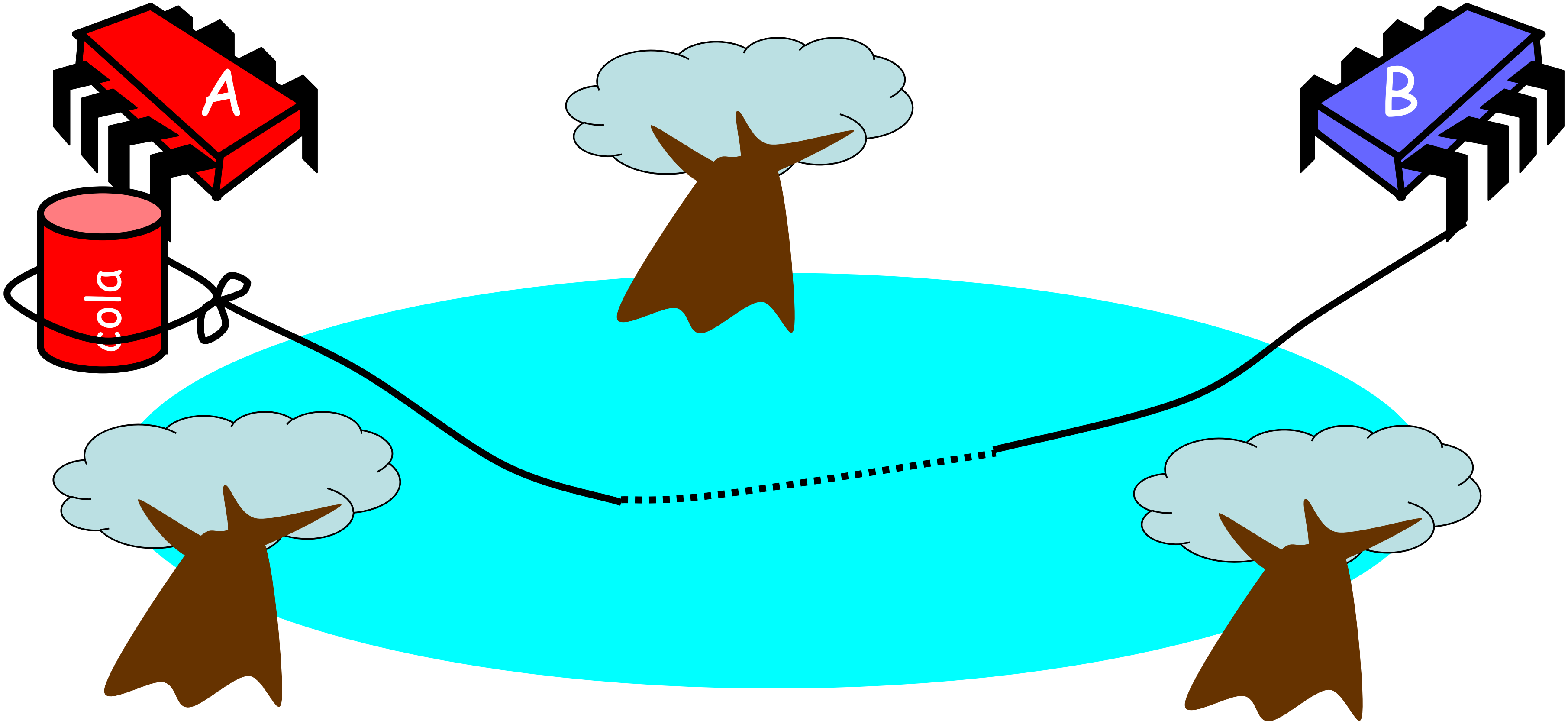
Producer/Consumer

- Alice and Bob can't meet
  - Bob’s disease is contagious
  - 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

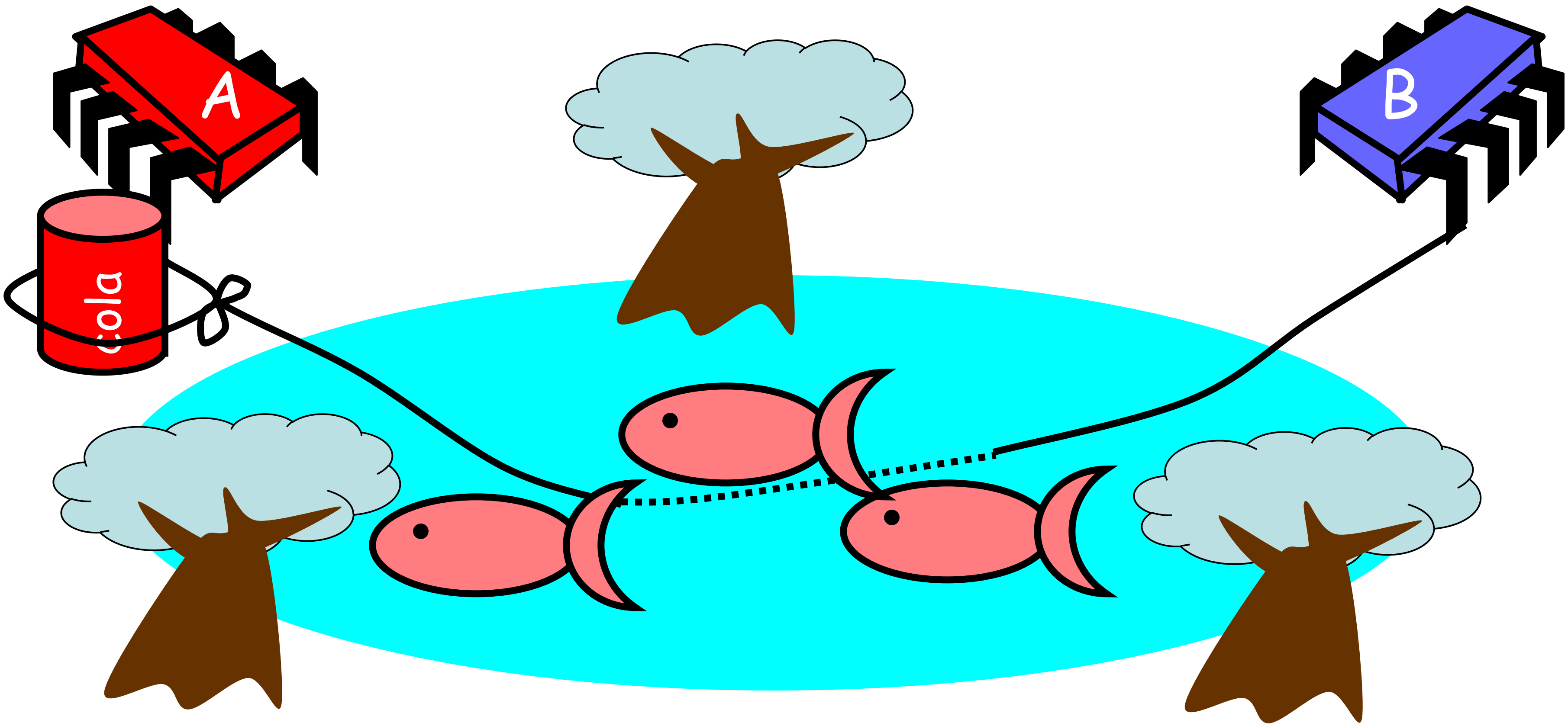
Producer/Consumer

- 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

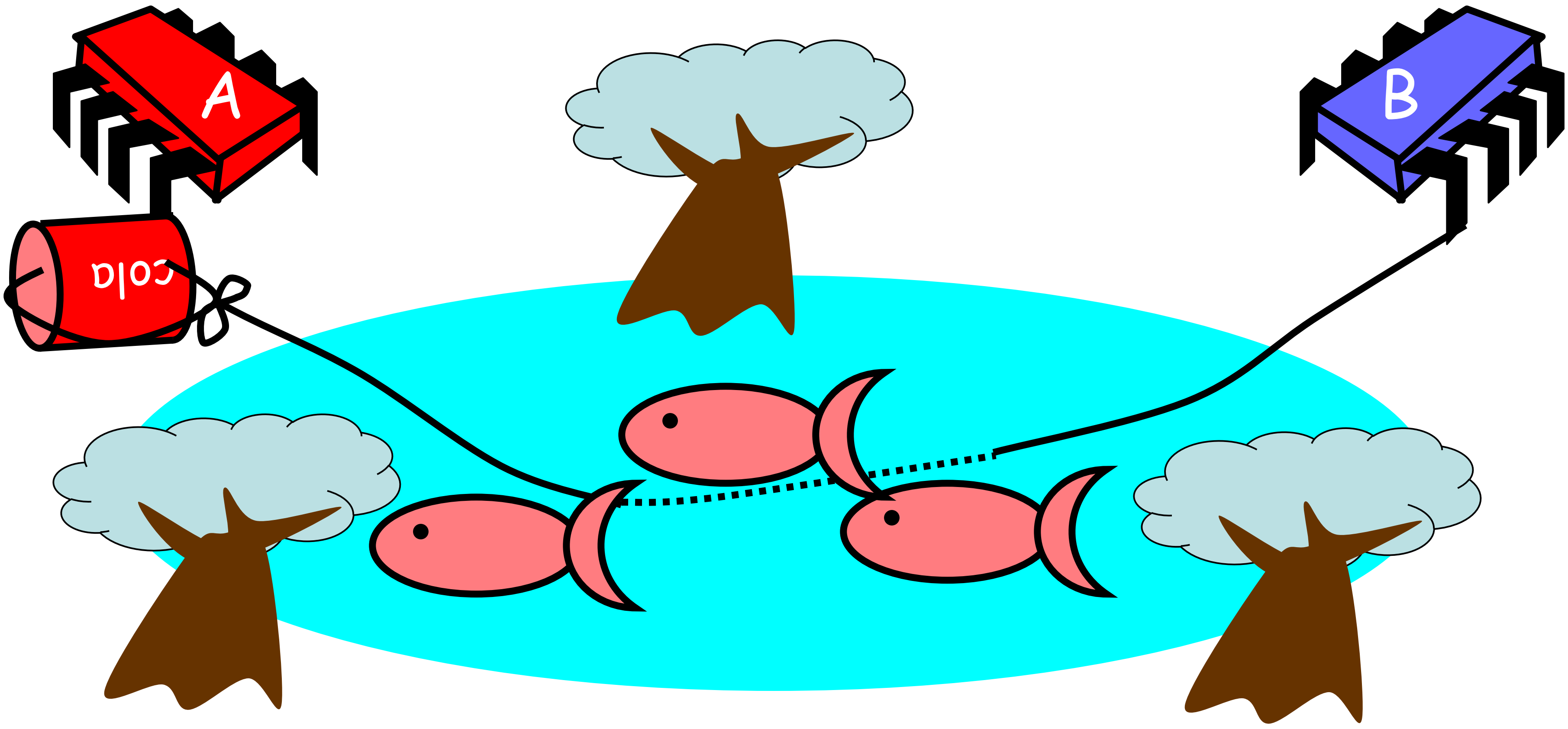
“Can” Solution



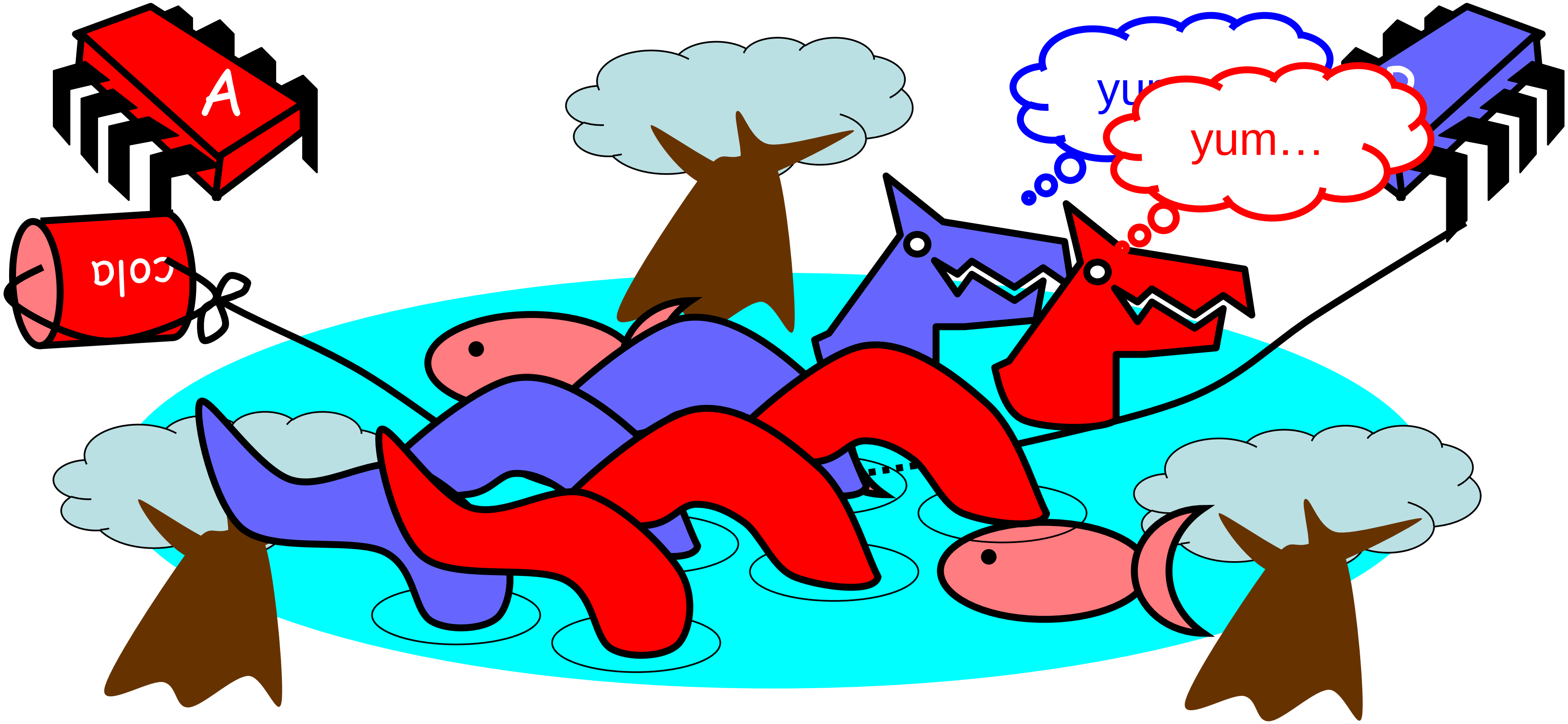
Bob puts food in Pond



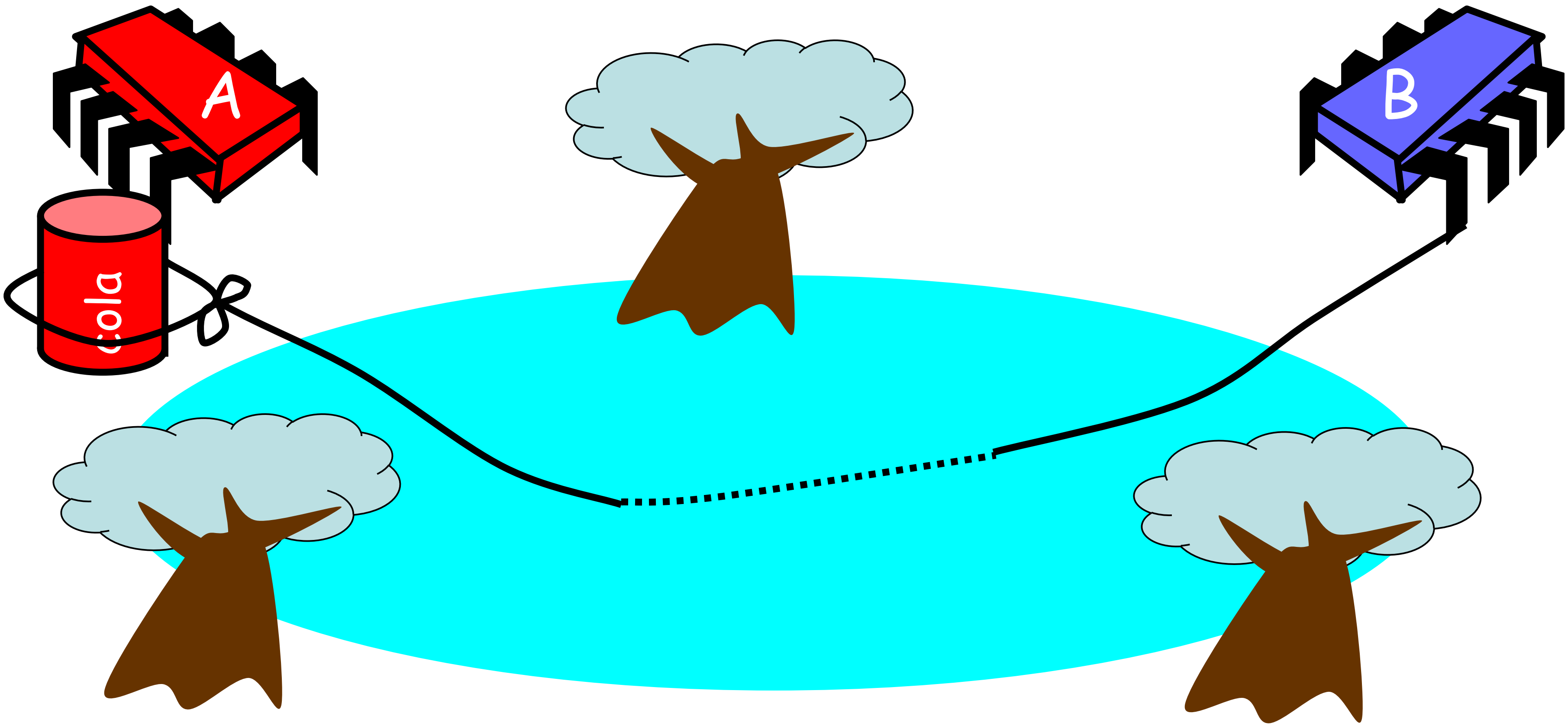
Bob knocks over Can



Alice Releases Pets



Alice Resets Can when Pets are Fed



Pseudocode

```
while (true) {
  while (can.isUp()){};
  pet.release();
  pet.recapture();
  can.reset();
}
```

Alice's code

Pseudocode

Alice's code

```
while (true) {
  while (can.isUp()){};
  pet.release();
  pet.recapture();
  can.reset();
}
```

Bob's code

```
while (true) {
  while (can.isDown()){};
  pond.stockWithFood();
  can.knockOver();
}
```

Correctness

- Mutual Exclusion
  - Pets and Bob never together in pond

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.

Correctness

- Mutual Exclusion — safety
  – Pets and Bob never together in pond
- No Starvation — liveness

  if Bob always willing to feed, and pets always famished, then pets eat infinitely often.
- Producer/Consumer — safety

  The pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food.

Spin Locks

Aside

Pseudocode

```
while (true) {
  while (can.isUp()){};
  pet.release();
  pet.recapture();
  can.reset();
}
```

Spin Lock!
Has to be protected…

```
while (true) {
  while (can.isDown()){};
  pond.stockWithFood();
  can.knockOver();
}
```

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

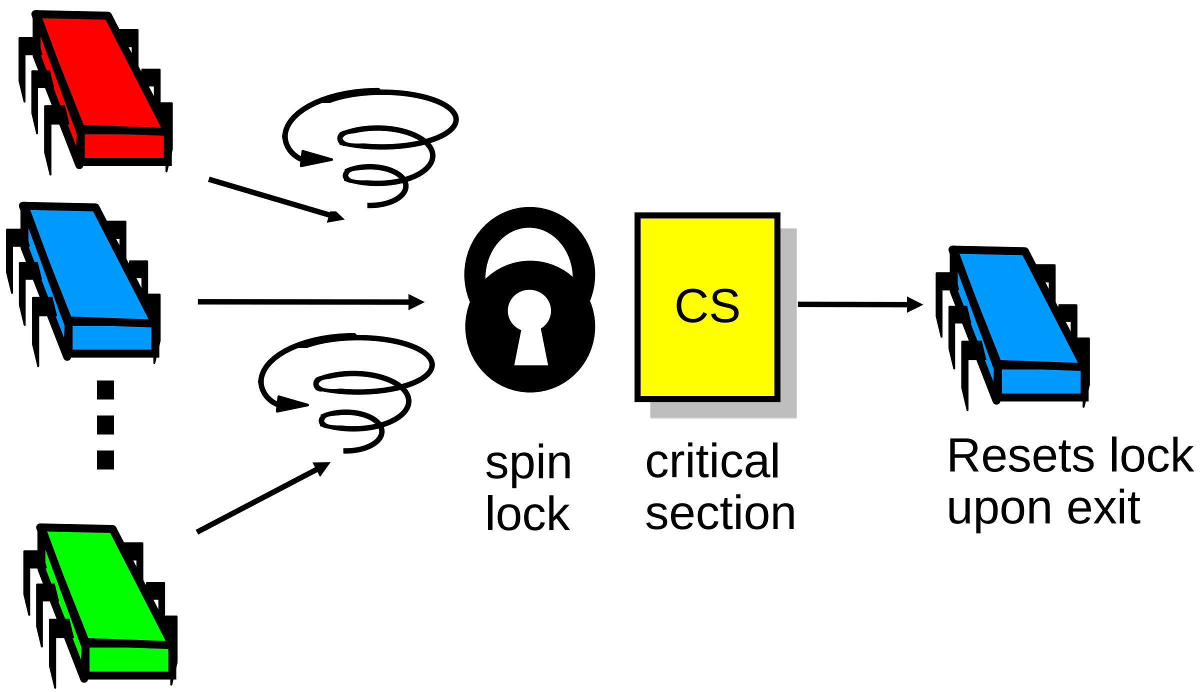
- Keep trying
  - “spin” or “busy-wait”
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

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

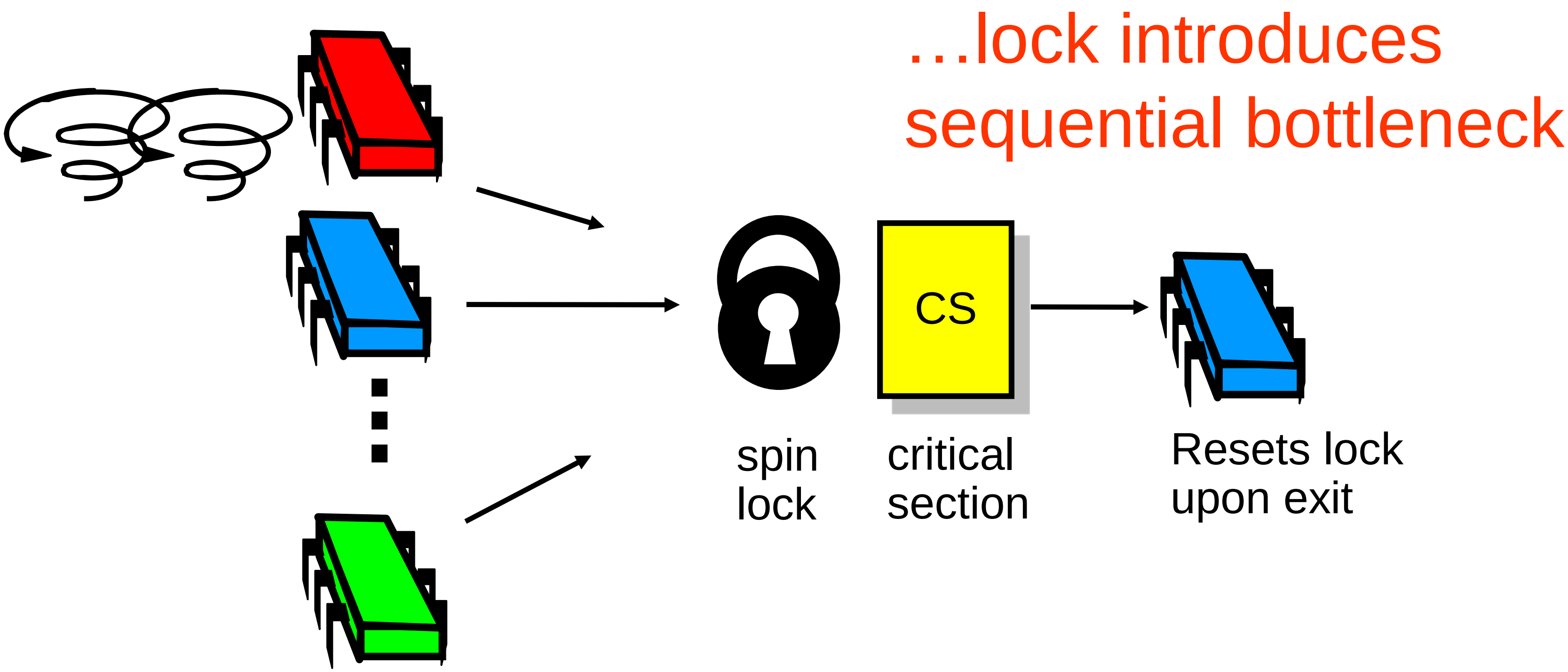
- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

our focus

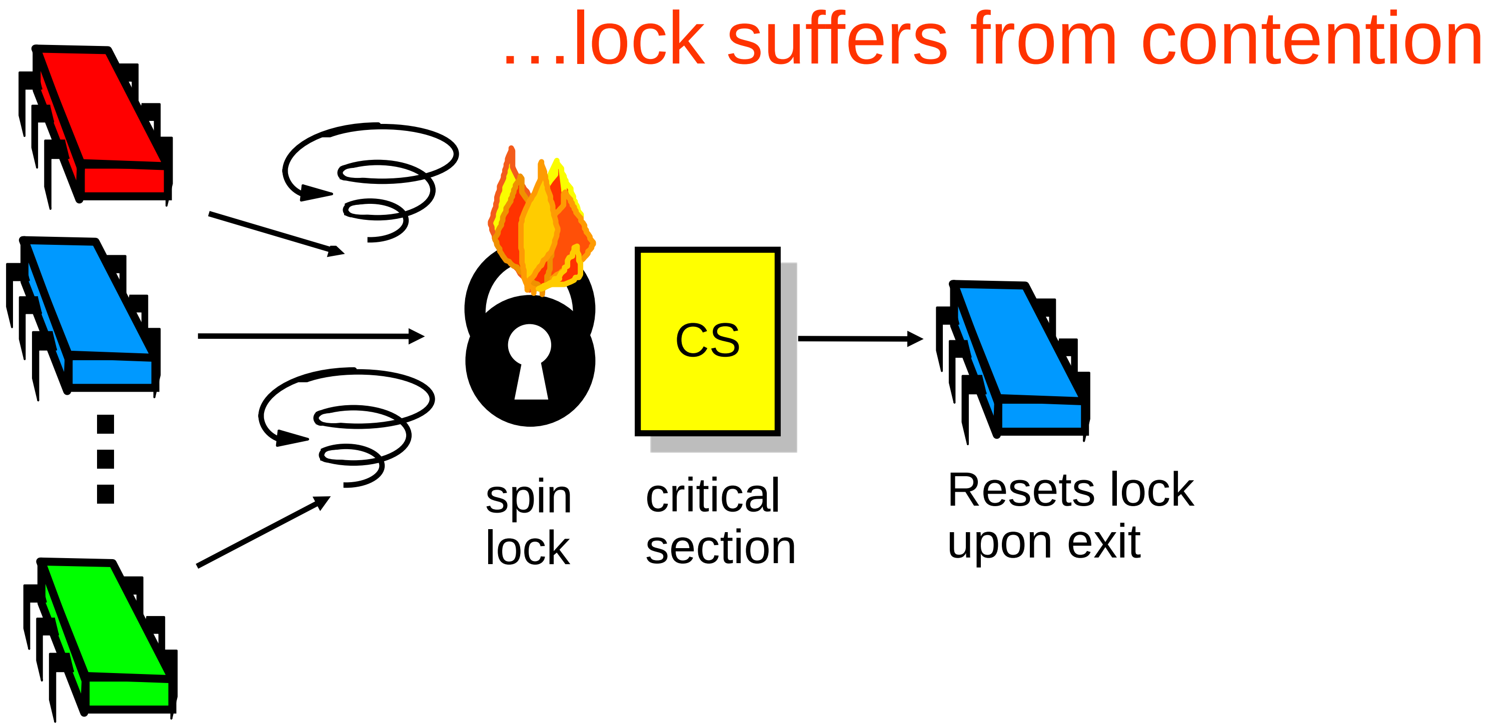
Basic Spin-Lock



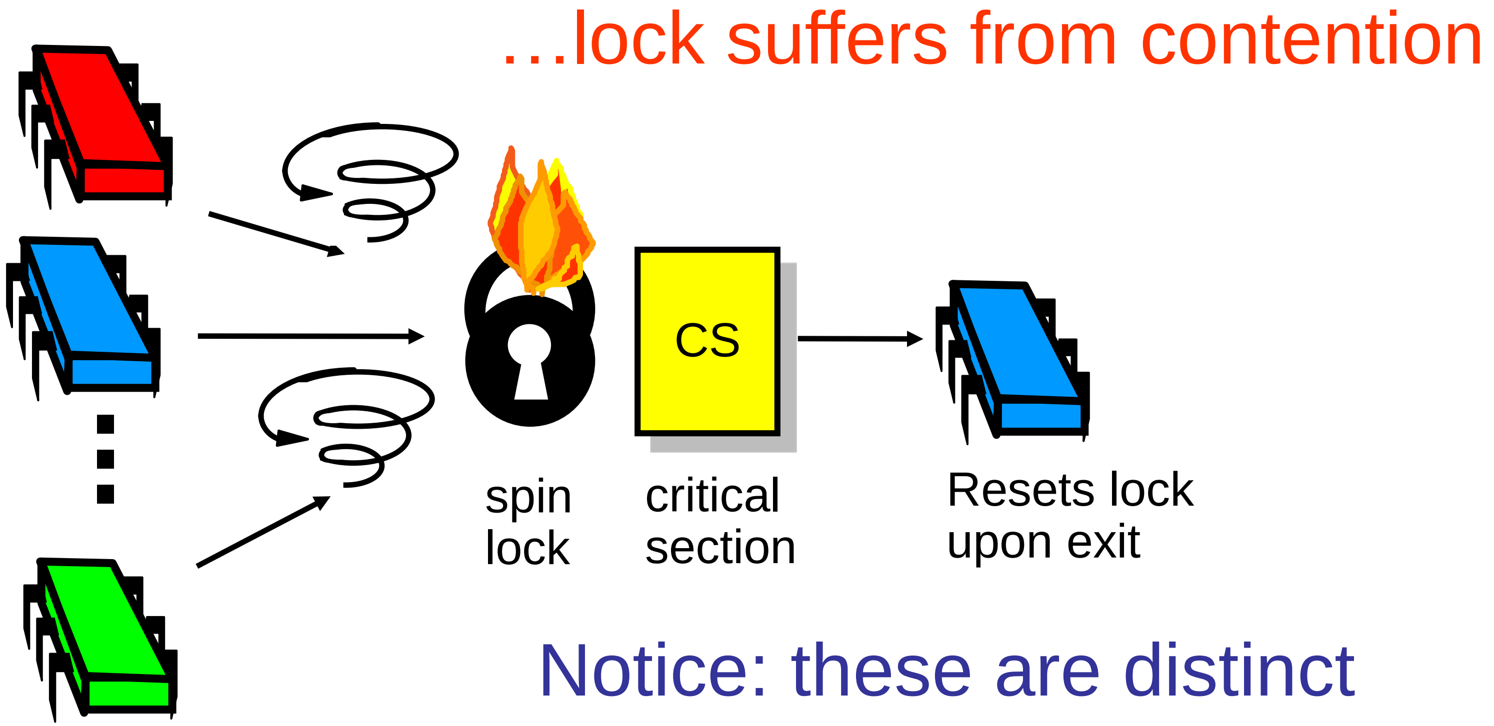
Basic Spin-Lock



Basic Spin-Lock



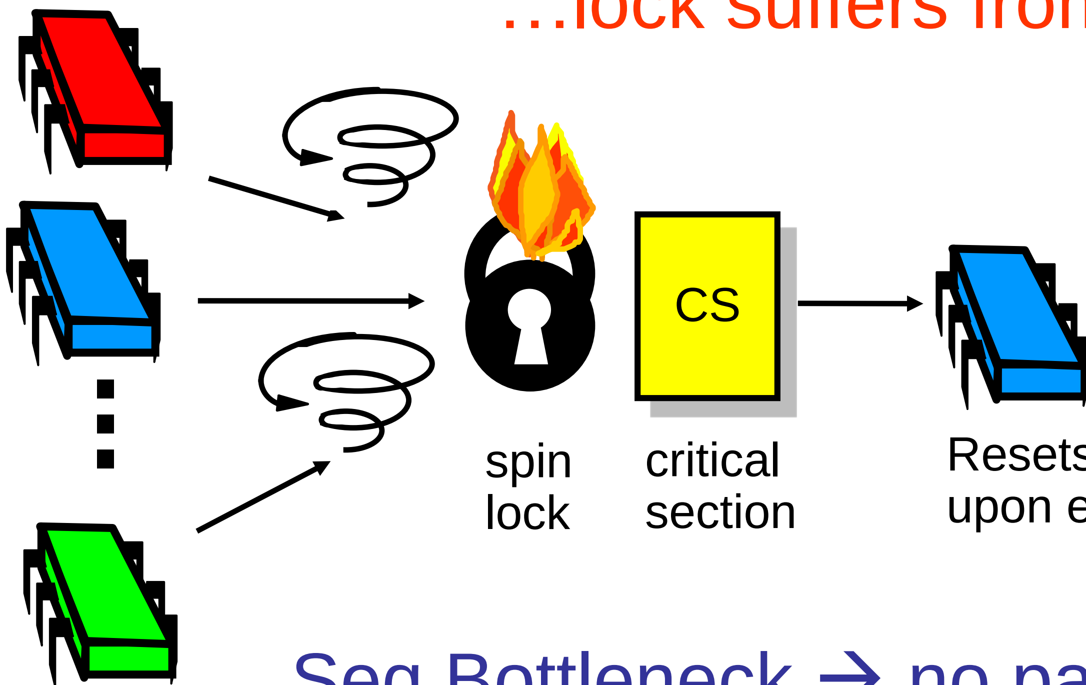
Basic Spin-Lock



Notice: these are distinct phenomena

Basic Spin-Lock

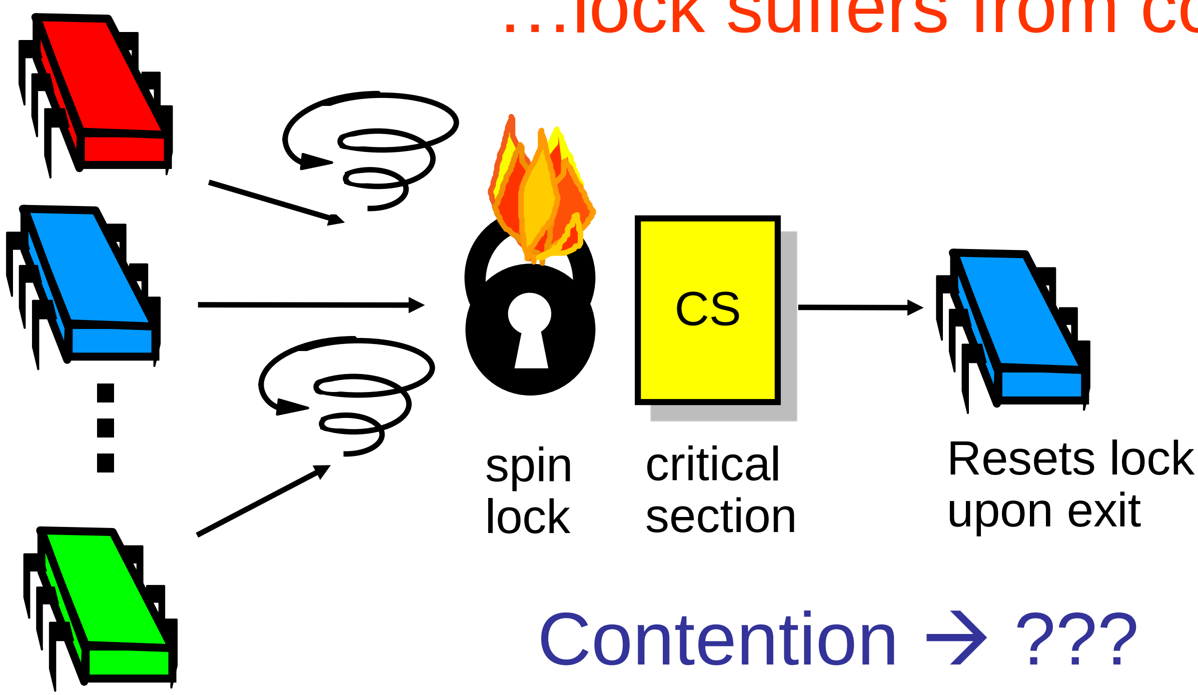
…lock suffers from contention



Seq Bottleneck → no parallelism

Basic Spin-Lock

…lock suffers from contention



Contention → ???

Review: Test-and-Set

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

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

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

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

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

Package
java.util.concurrent.atomic

Review: Test-and-Set

```
public class AtomicBoolean {
 boolean value;

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

Swap old and new values

Review: Test-and-Set

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

Review: Test-and-Set

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

Swapping in true is called "test-and-set" or TAS

Test-and-Set Locks

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

Test-and-set Lock

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

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

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

Test-and-set Lock

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

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

 void unlock() {
  state
 }}
```

Lock state is AtomicBoolean

Test-and-set Lock

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

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

 void unlock() {
  sta
 }}
```

Keep trying until lock acquired

Test-and-set Lock

```
class TA
 AtomicB
  new At

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

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

Release lock by resetting state to false

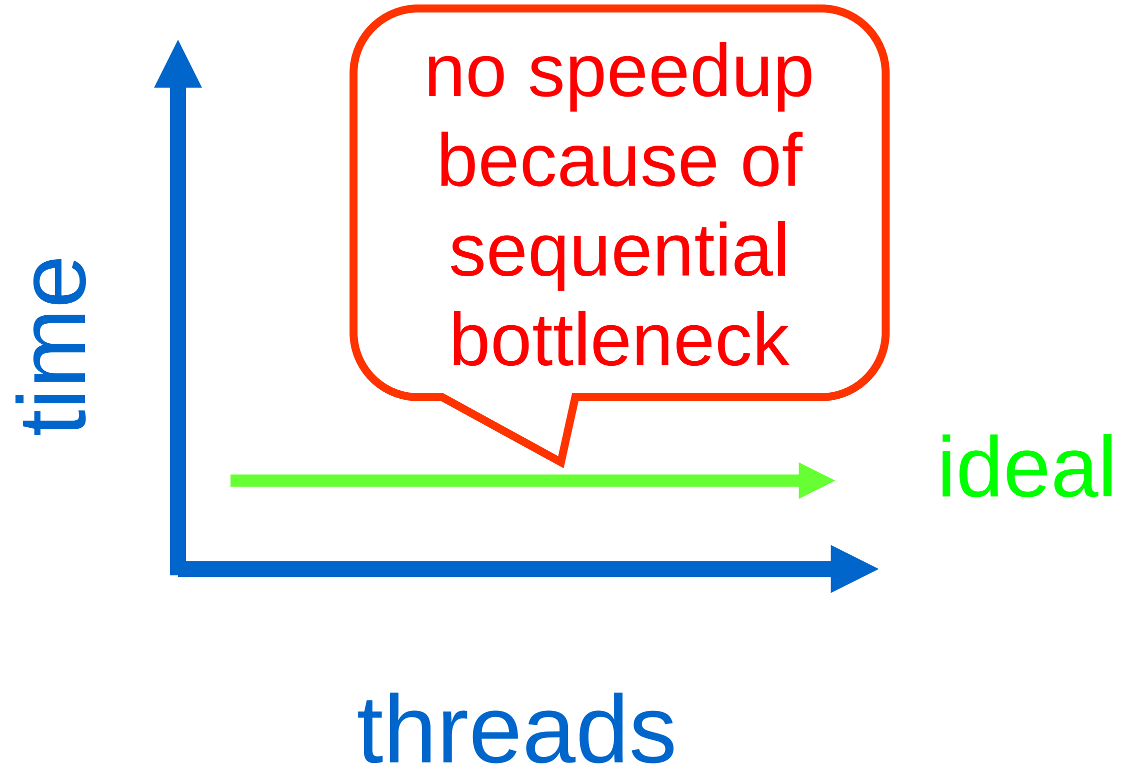
Space Complexity

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

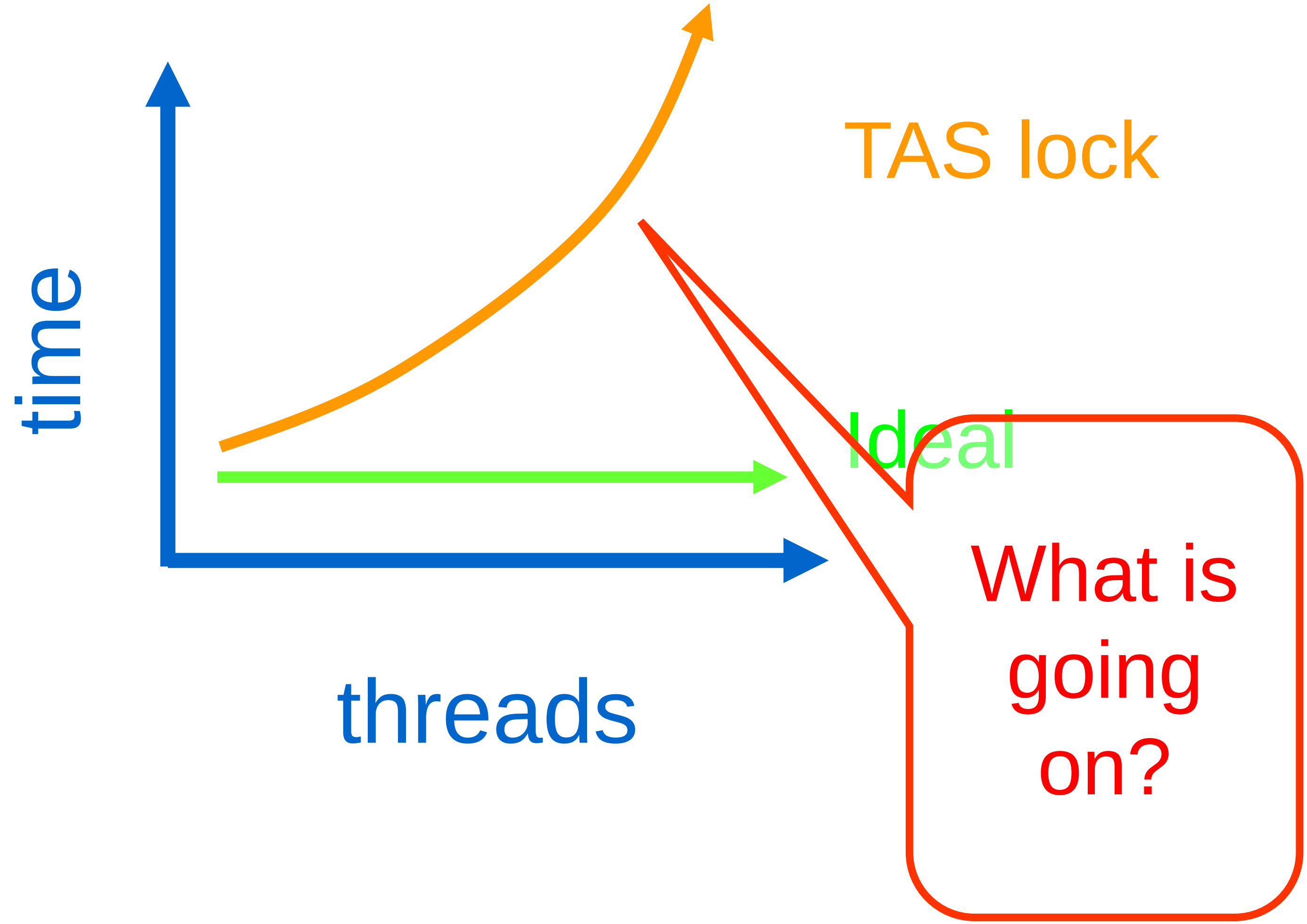
Performance

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

Graph


no speedup because of sequential bottleneck
ideal
time
threads

Mystery #1



Test-and-Test-and-Set Locks

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

Test-and-test-and-set Lock

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

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

Test-and-test-and-set Lock

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

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

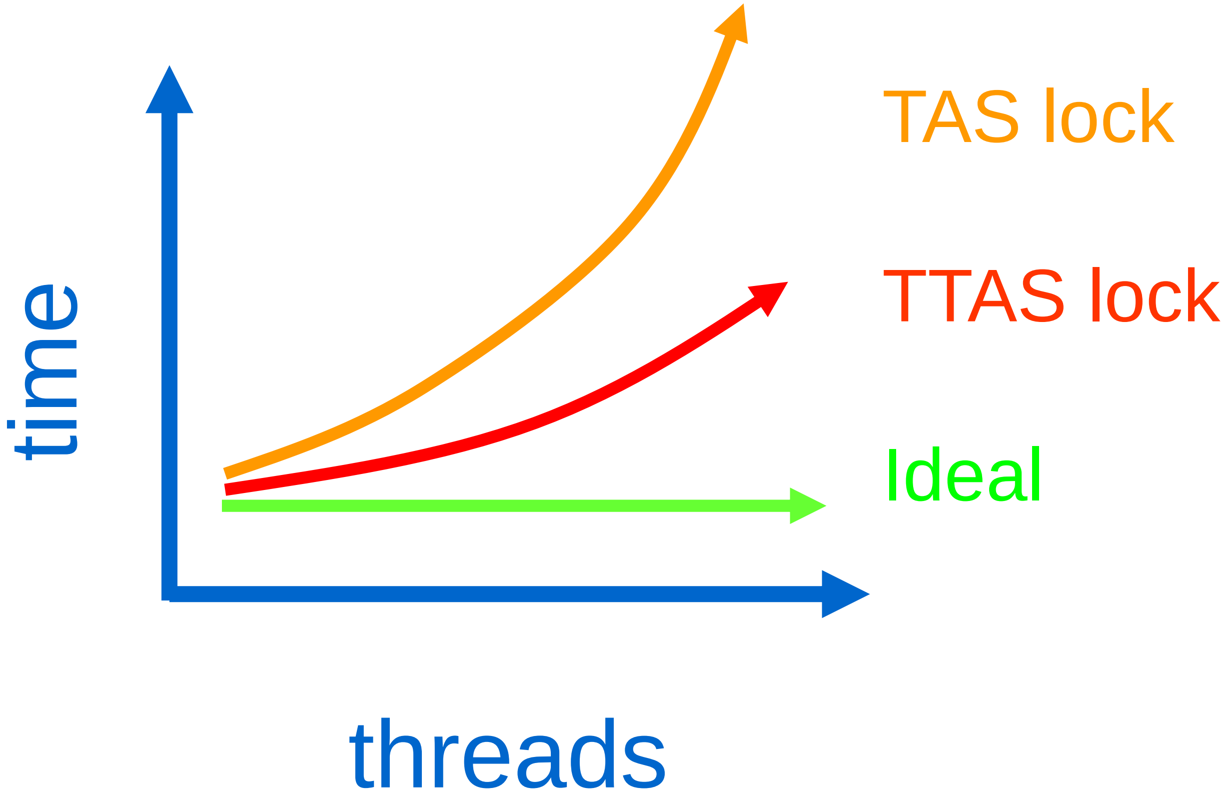
Wait until lock looks free

Test-and-test-and-set Lock

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

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

Then try to acquire it

Mystery #2



Mystery

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

Opinion

- Our memory abstraction is broken
- TAS & TTAS methods
  - Are provably the same (in our model)
  - Except they aren't (in field tests)
- Need a more detailed model …

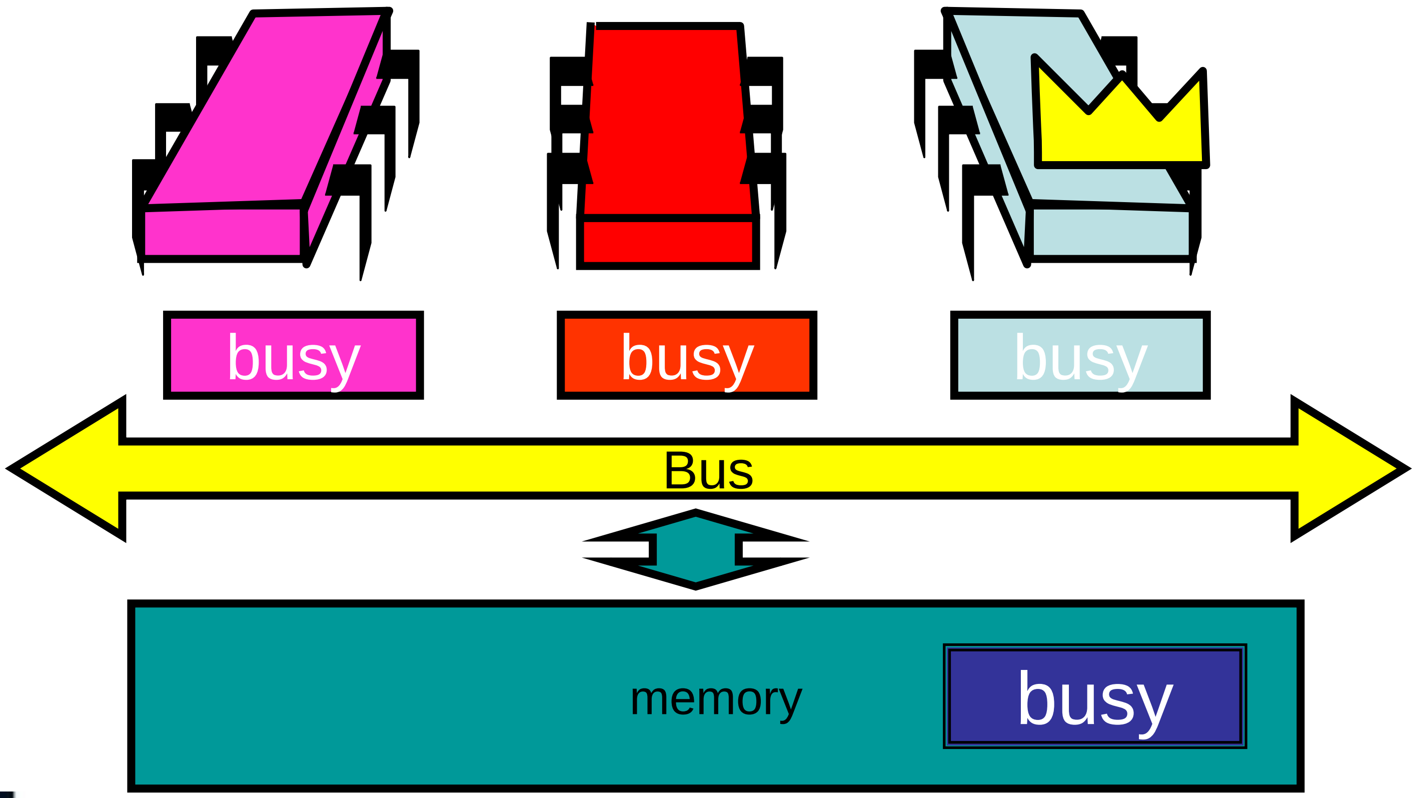
Simple TASLock

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

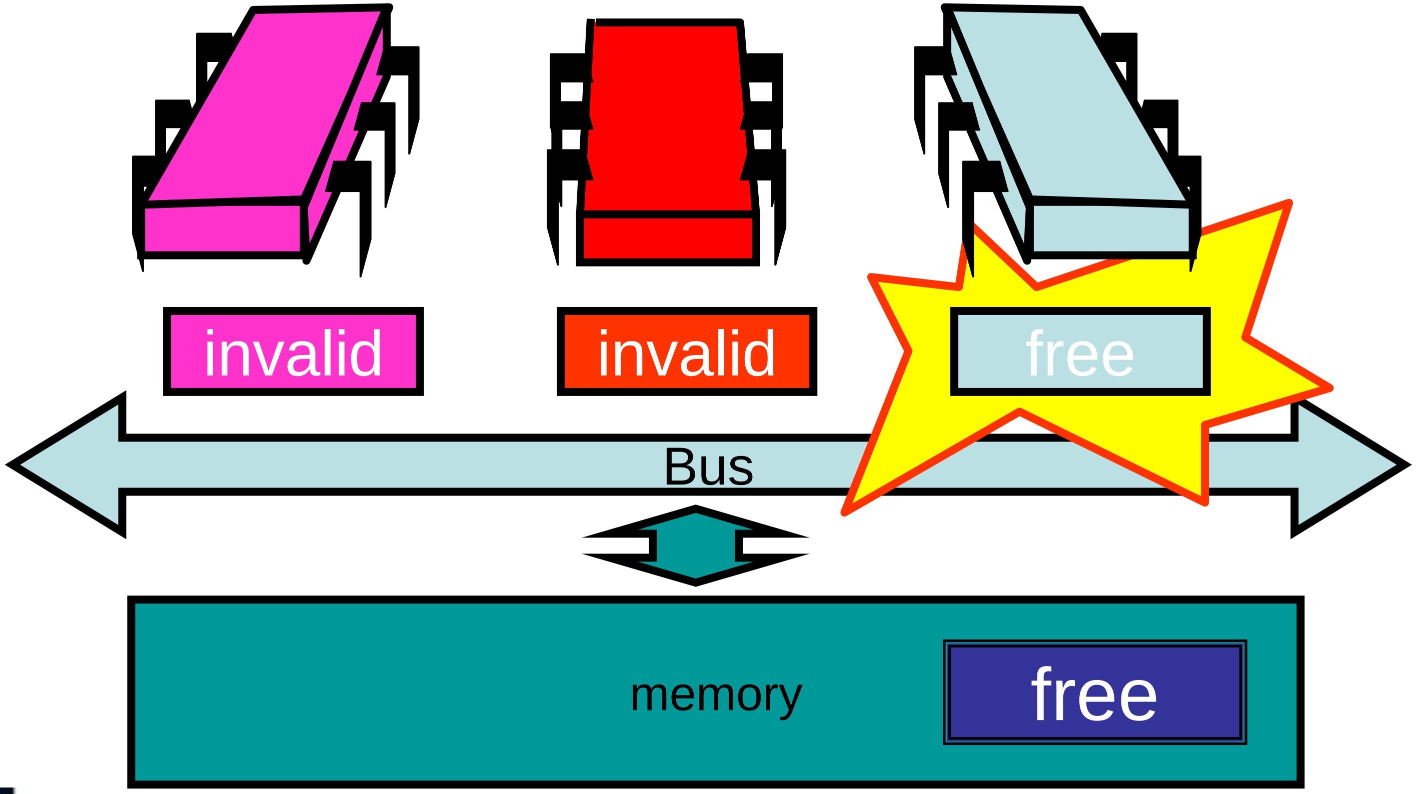
Test-and-test-and-set

- Wait until lock "looks" free
  - Spin on local cache
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm …

Local Spinning while Lock is Busy

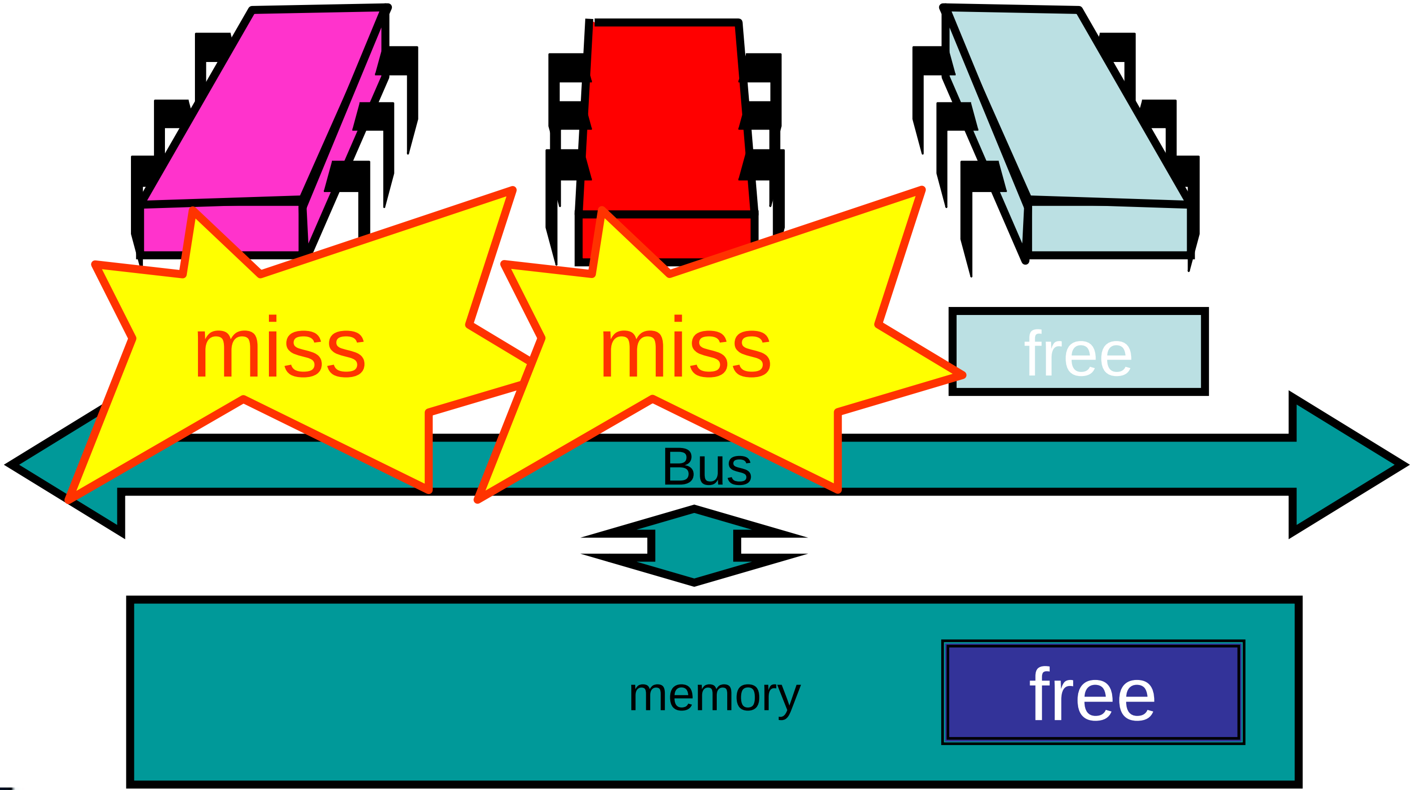


On Release



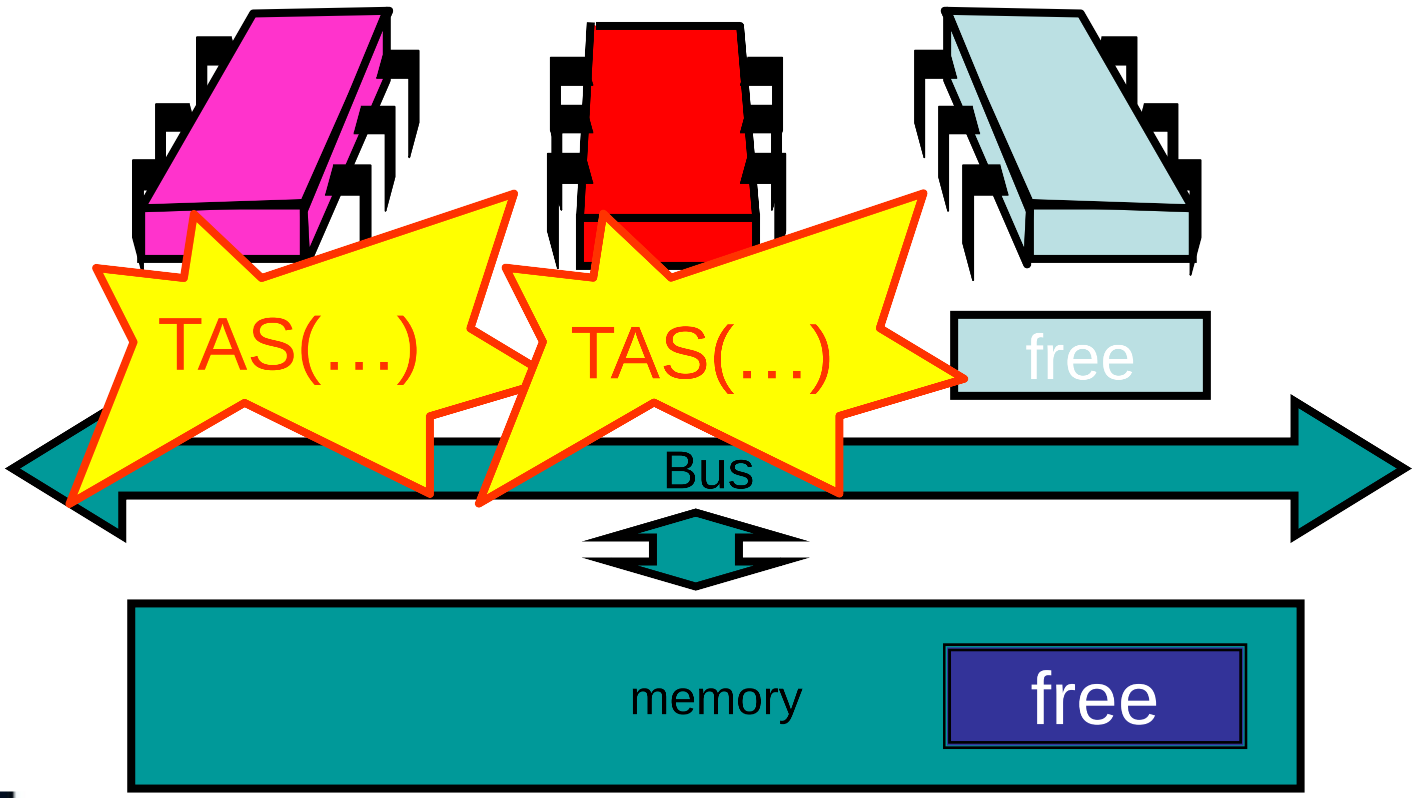
On Release

Everyone misses, rereads



On Release

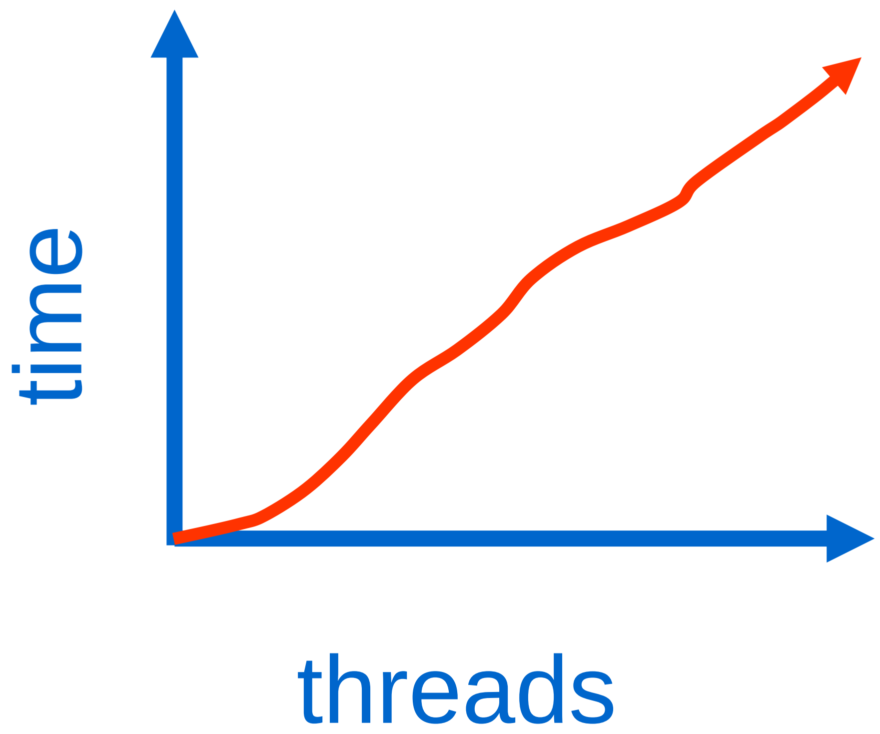
Everyone tries TAS



Problems

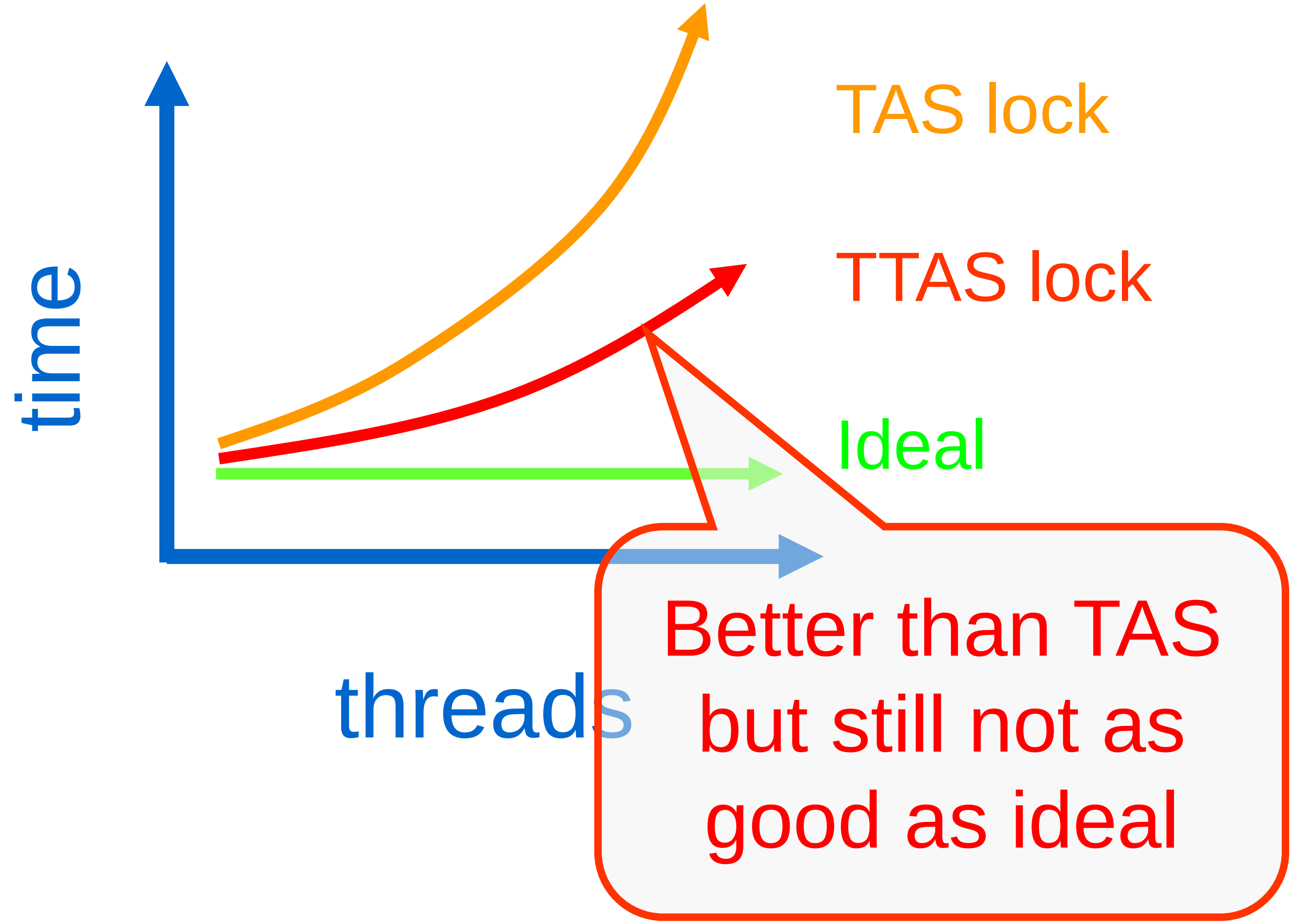
- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?

Quiescence Time



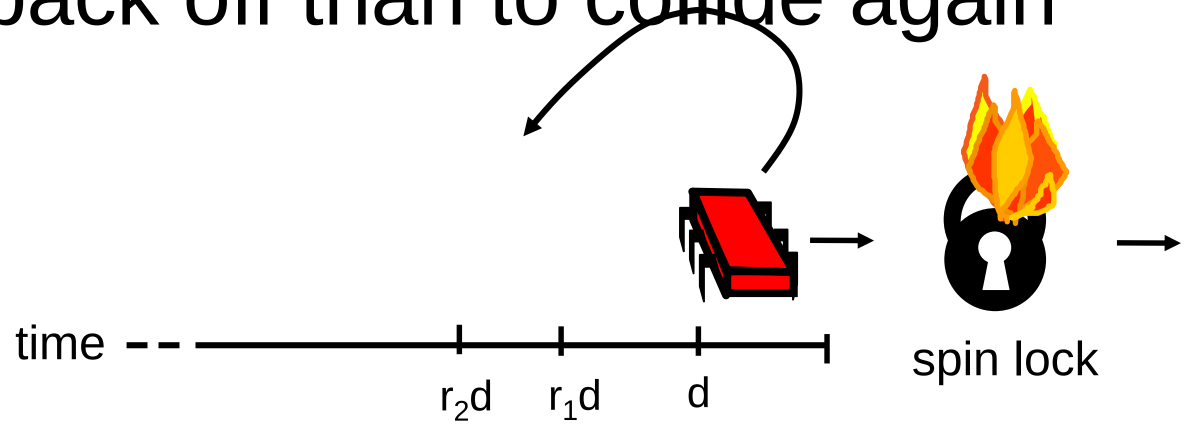
Increses linearly with the number of processors for bus architecture

Mystery Explained



Solution: Introduce Delay

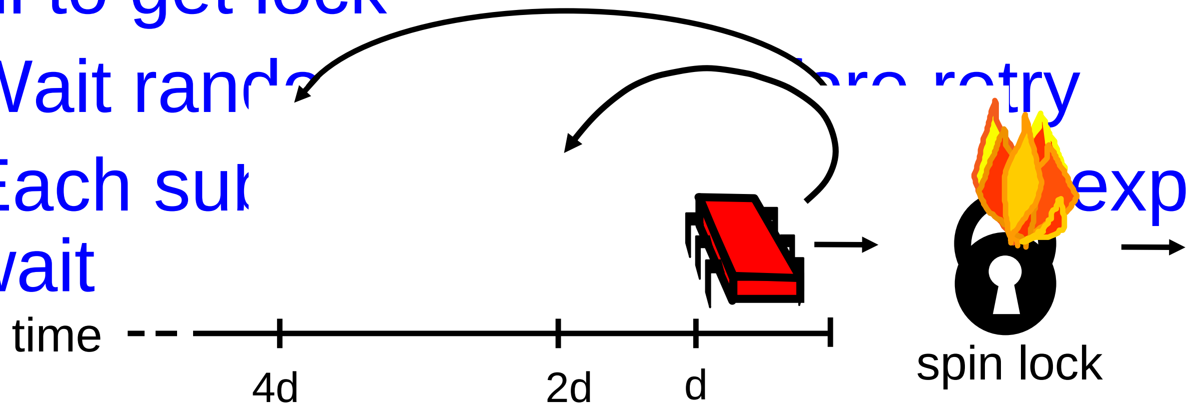
- If the lock looks free
  - But I fail to get it
- There must be contention
  - Better to back off than to collide again



Dynamic Example: Exponential Backoff

If I fail to get lock

- Wait rand… …ore retry
- Each sub… …expected wait



Concurrent Data Structures

What if you had multiple producers, consumers?

```
while (true) {
  while (a.isLocked()){};
  while (can.isUp()){};
  pet.release();
  pet.recapture();
  can.reset();
}
```

Alice & Co.

Bob & Co.

```
while (true) {
  while (b.isLocked()){};
  while (can.isDown()){};
  pond.stockWithFood();
  can.knockOver();
}
```

Does this improve performance?

- Sequential bottleneck!

Why do we care About Sequential Bottlenecks?

- We want as much of the code as possible to execute in parallel
- A larger sequential part implies reduced performance
- Amdahl's law: this relation is not linear…

Eugene Amdahl

Amdahl's Law

$$\text{Speedup} = \frac{\textit{1}\ \text{thread execution time}}{\textit{N}\ \text{thread execution time}}$$

Amdahl's Law

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

Amdahl's Law

Parallel fraction

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

Amdahl's Law

Sequential fraction

Parallel fraction

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

Amdahl's Law

Sequential fraction

Parallel fraction

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

Number of threads

Amdahl's Law (in practice)

Example

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

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?

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?

Example

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

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

Example

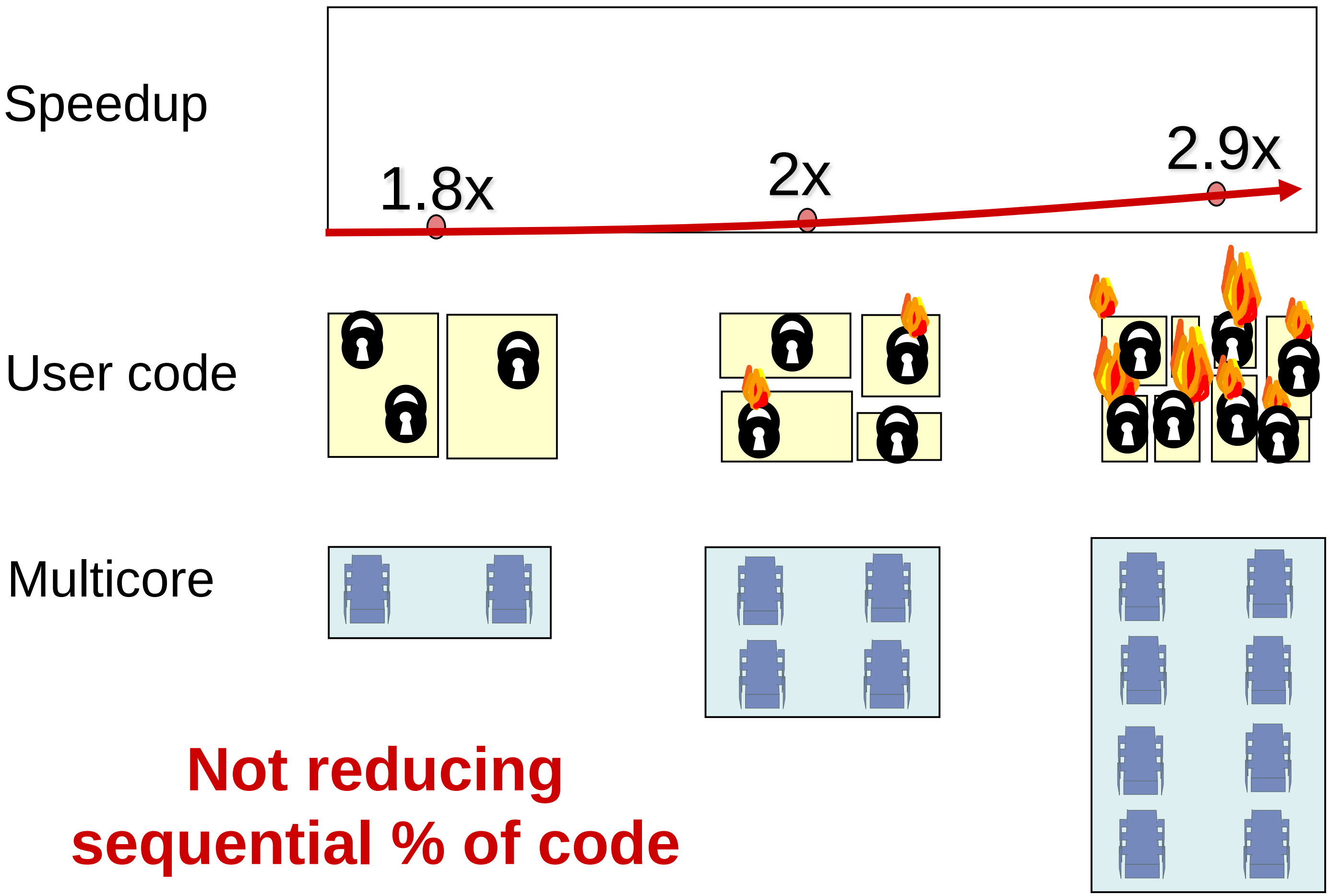
- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}$$

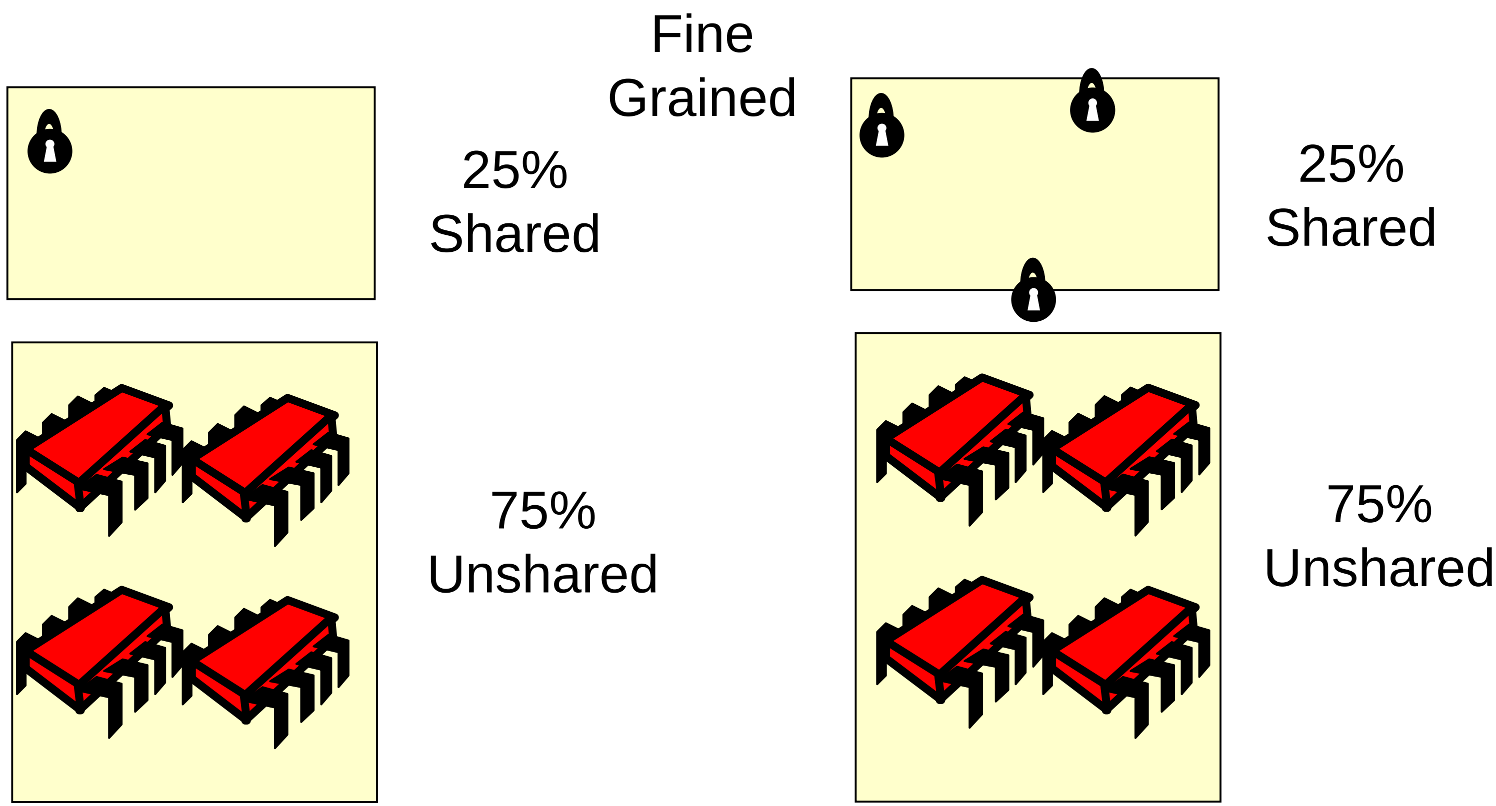
Back to Real-World Multicore Scaling



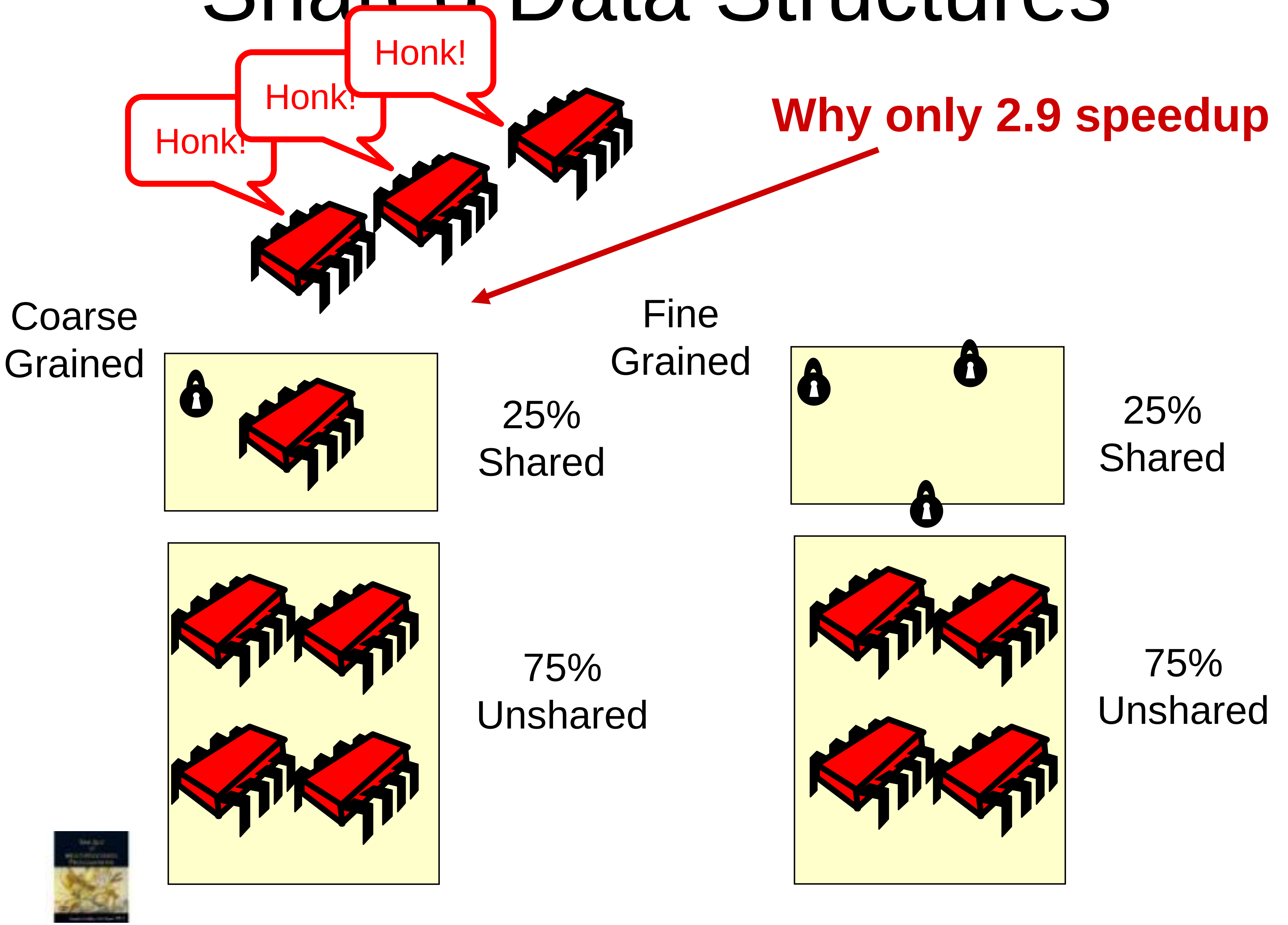
Not reducing sequential % of code

Shared Data Structures

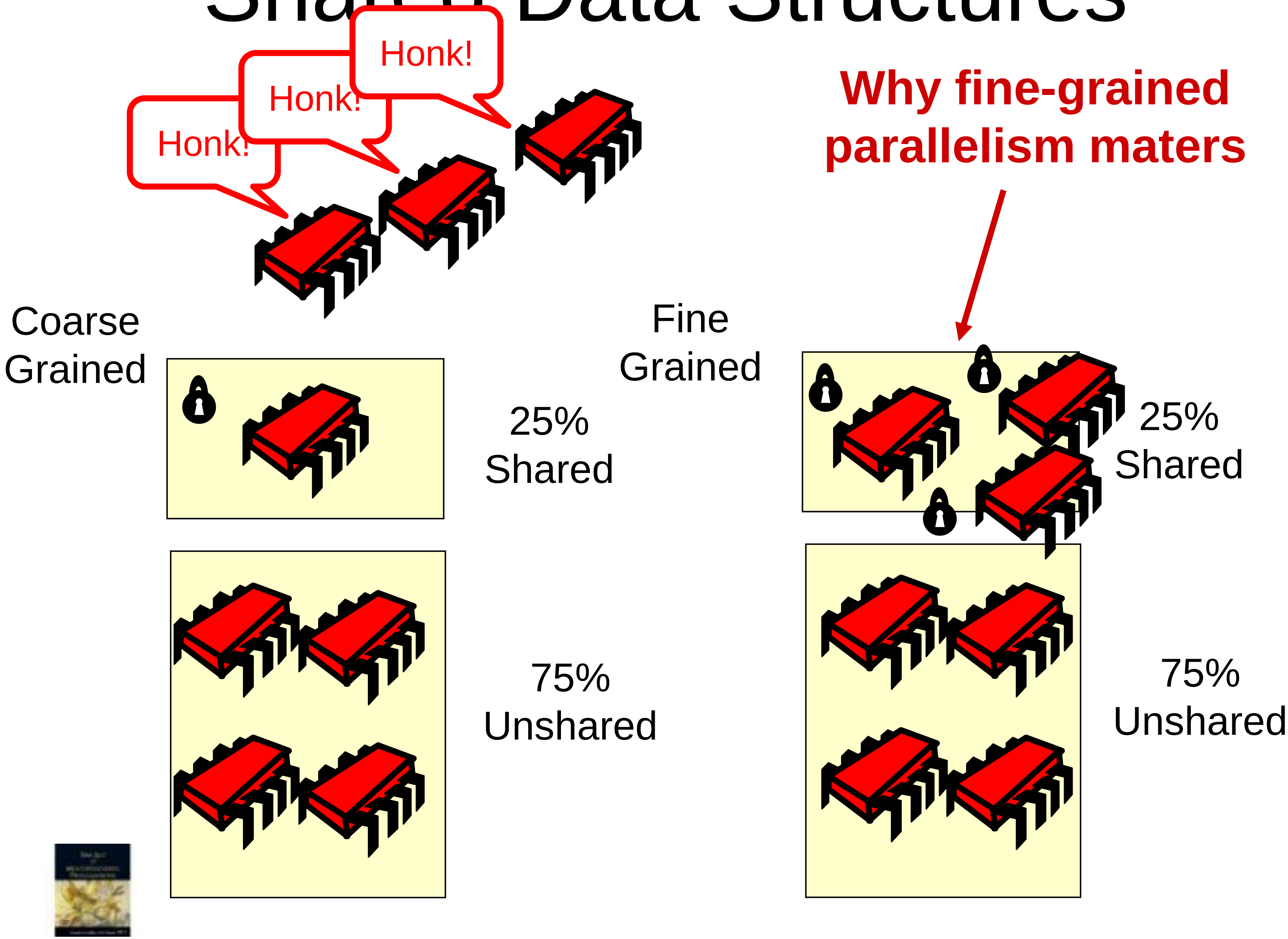
Coarse
Grained



Shared Data Structures



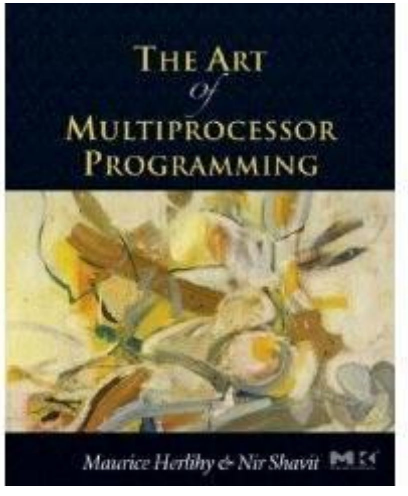
Shared Data Structures



Need for Concurrent Queues

- Avoid sequential bottleneck by introducing a buffer between the producers and consumers
- Producers add item to queue
- Consumers consume from queue
- Neither wait as long as queue is not full or empty

Concurrent Objects

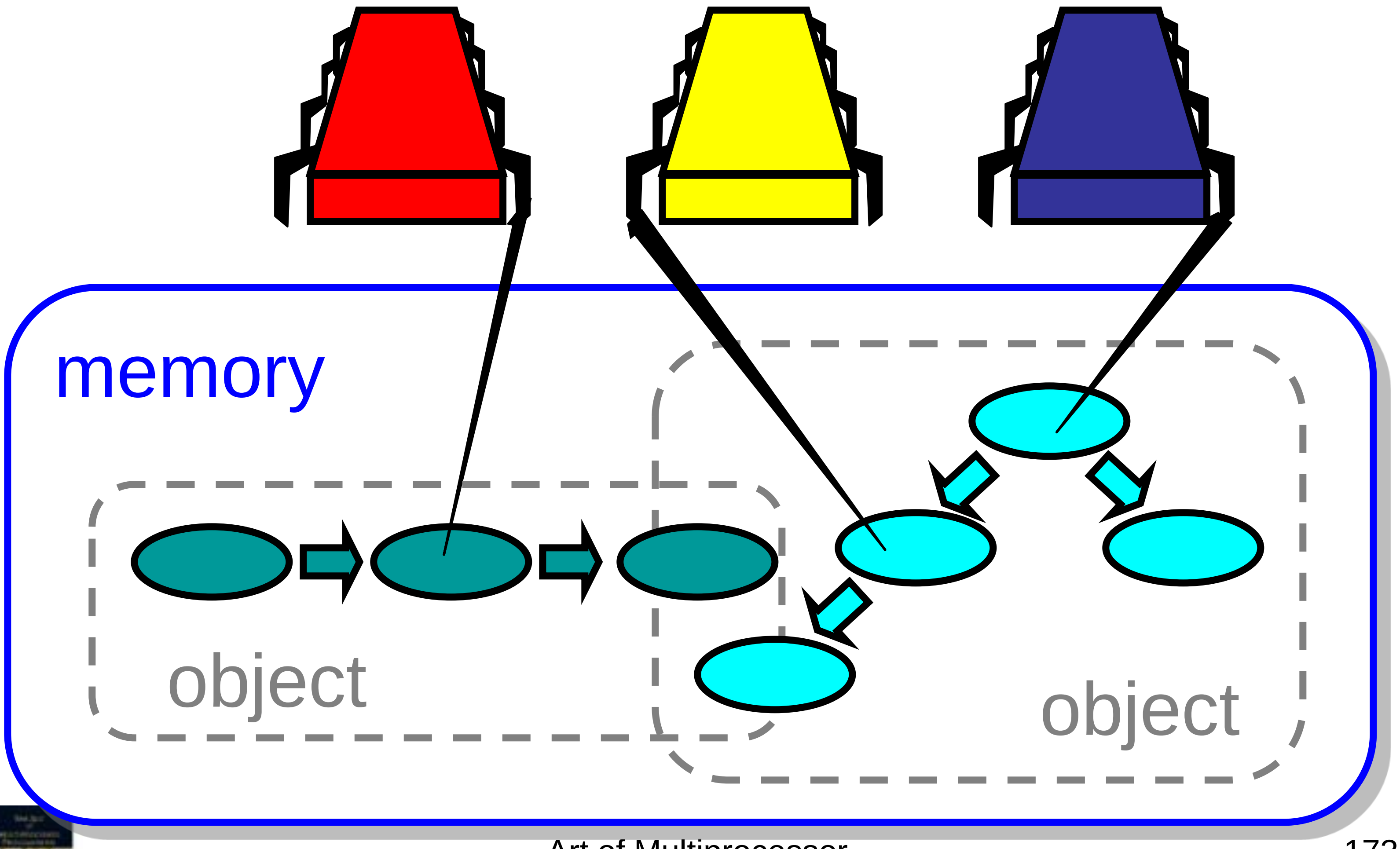


Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

Concurrent Computation



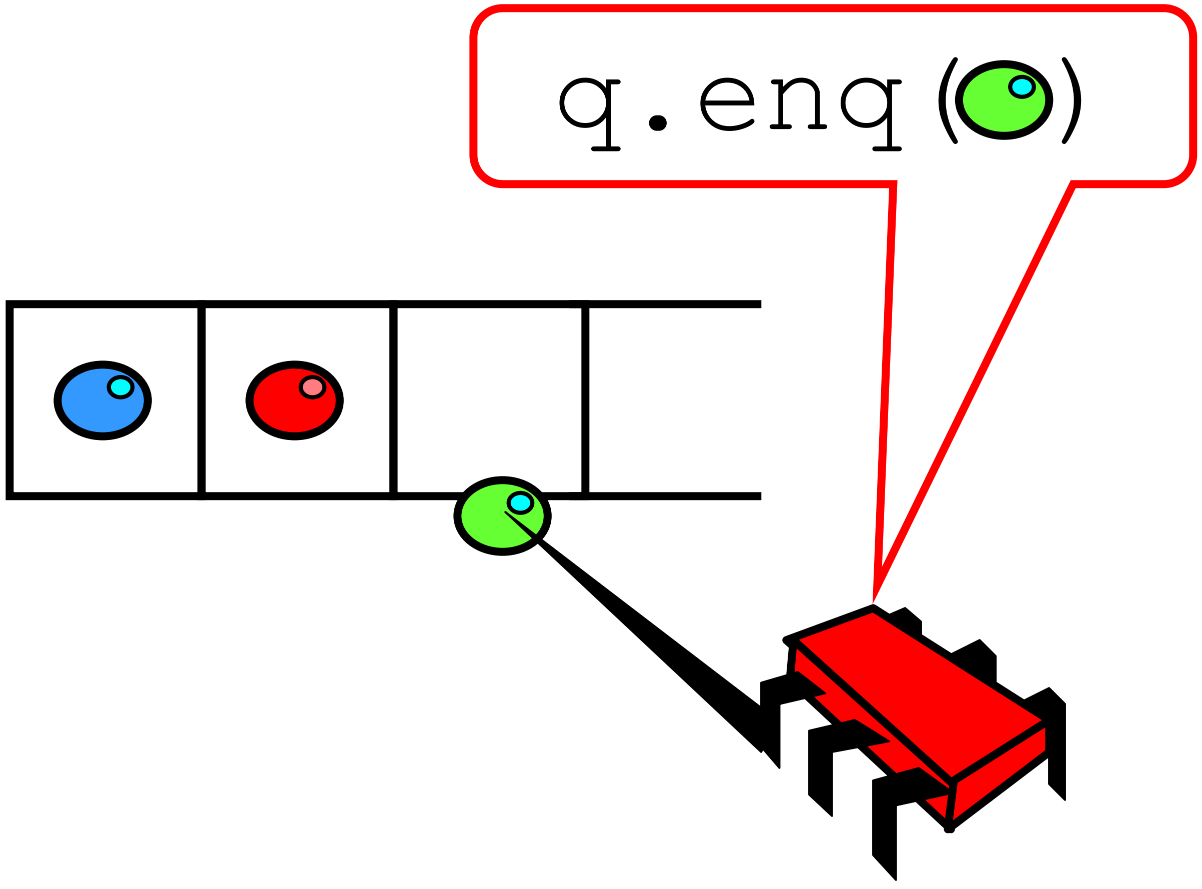
Objectivism

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

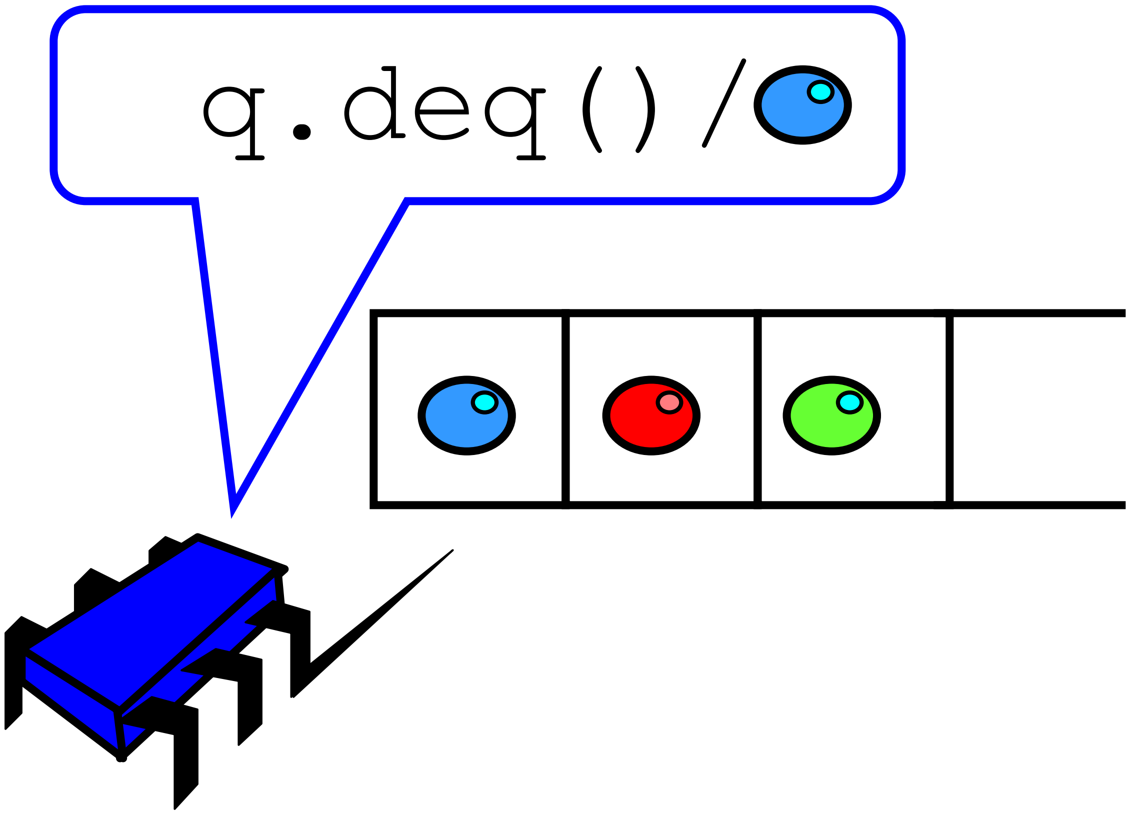
Objectivism

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

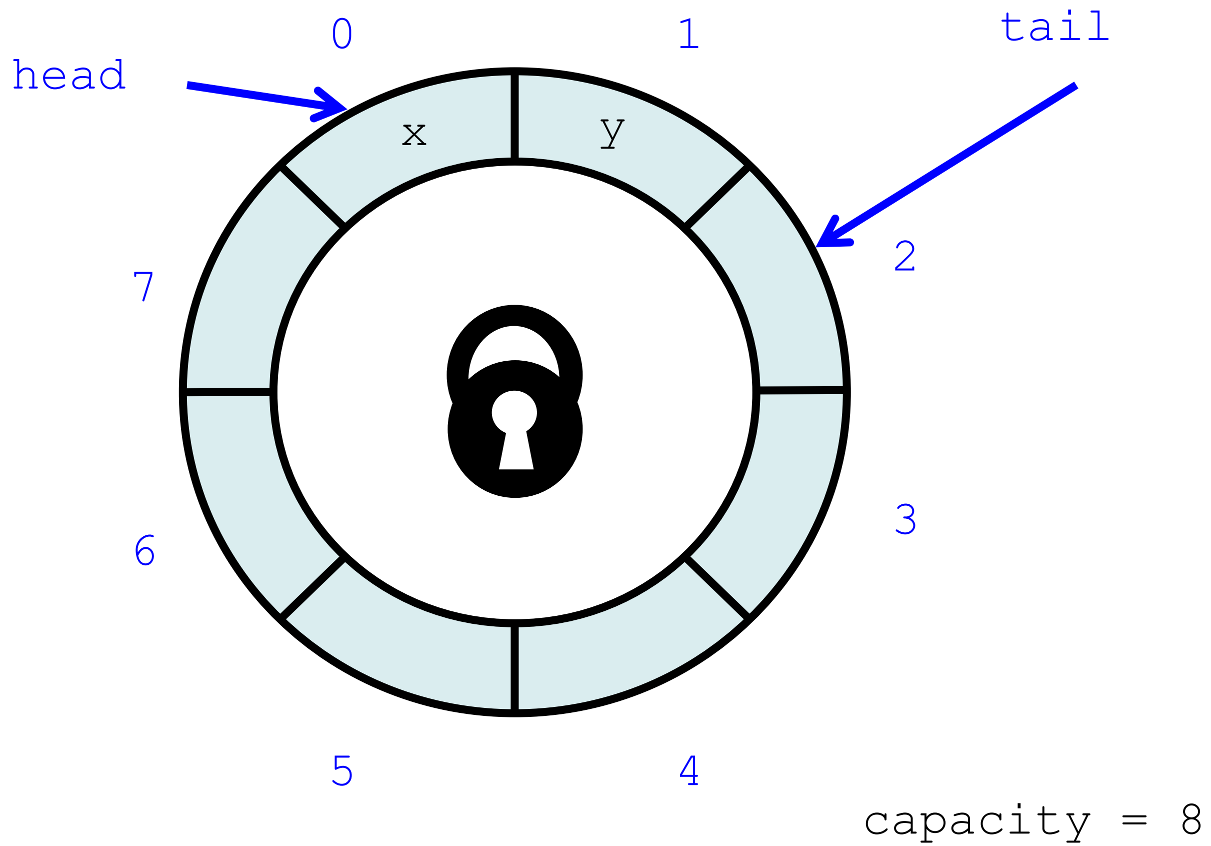
FIFO Queue: Enqueue Method



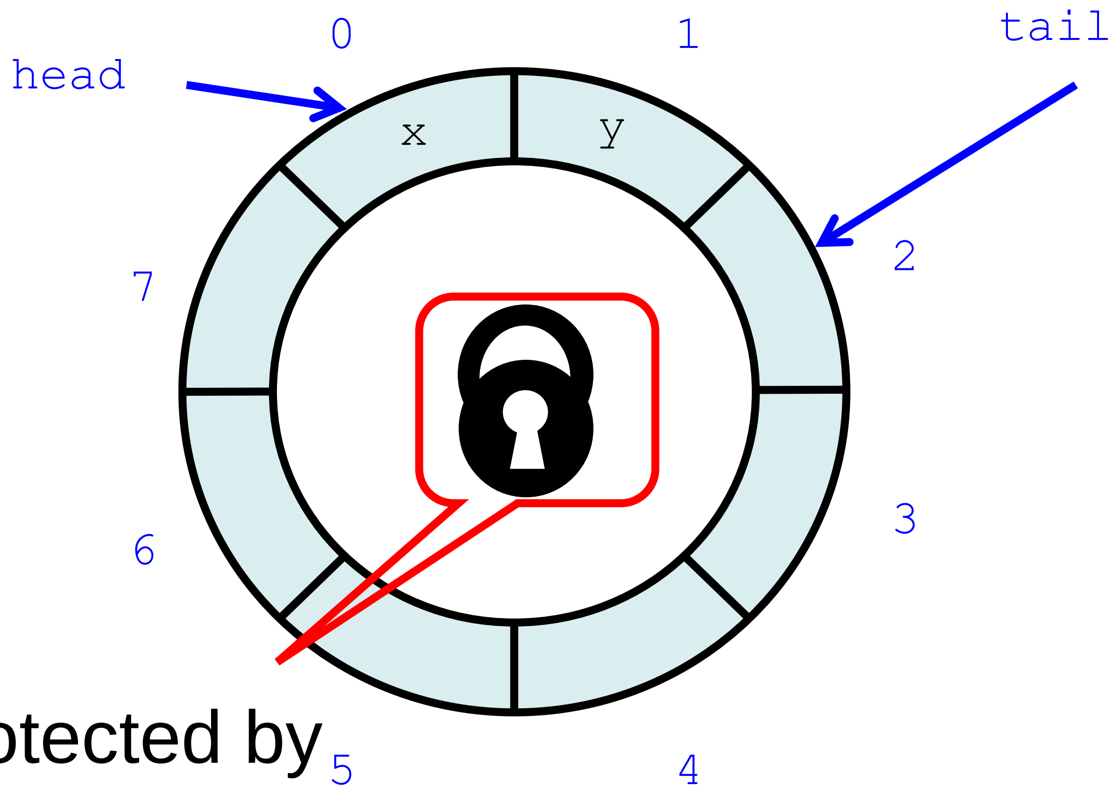
FIFO Queue: Dequeue Method



Lock-Based Queue



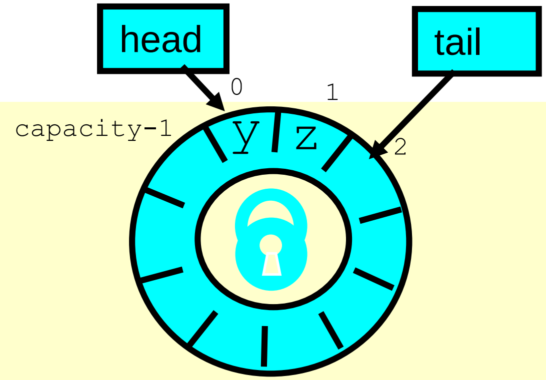
Lock-Based Queue



Fields protected by single shared lock

capacity = 8

A Lock-Based Queue

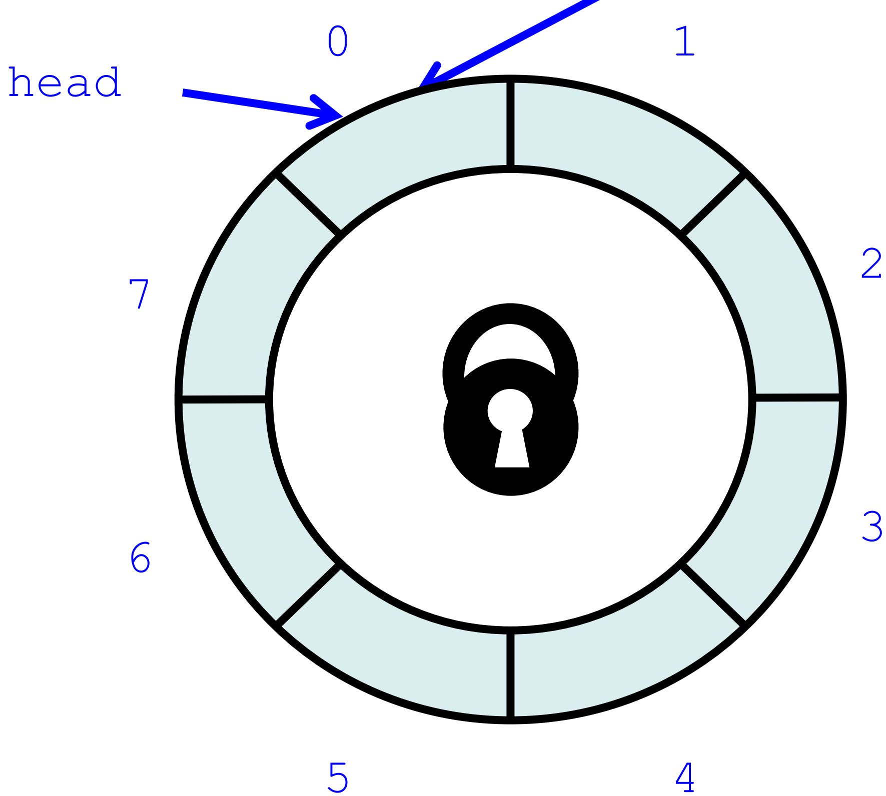



```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

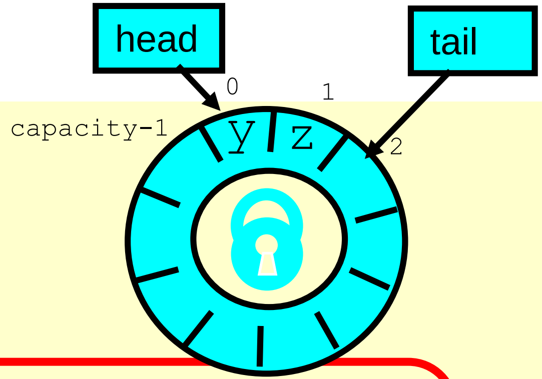
Fields protected by single shared lock

Lock-Based Queue

Initially: `head = tail`



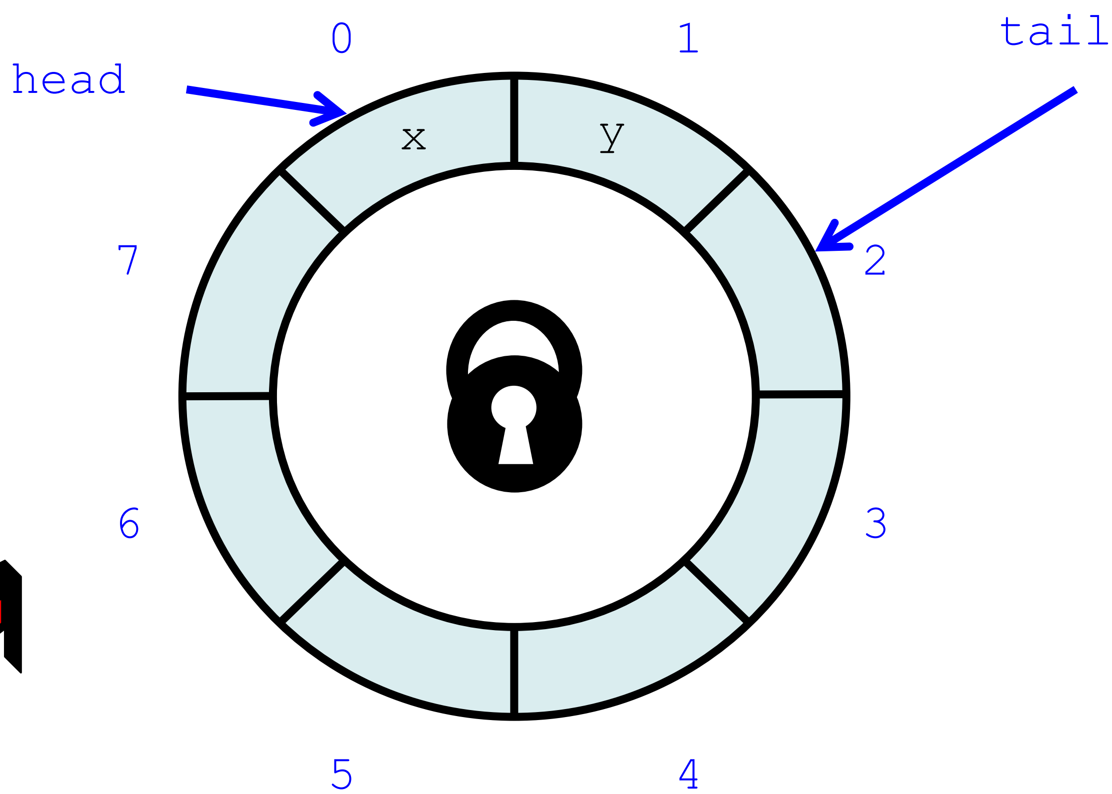
A Lock-Based Queue

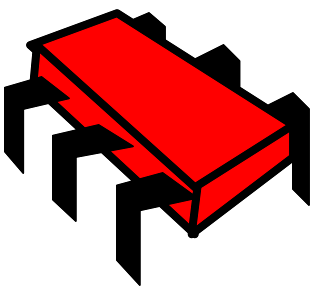



```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

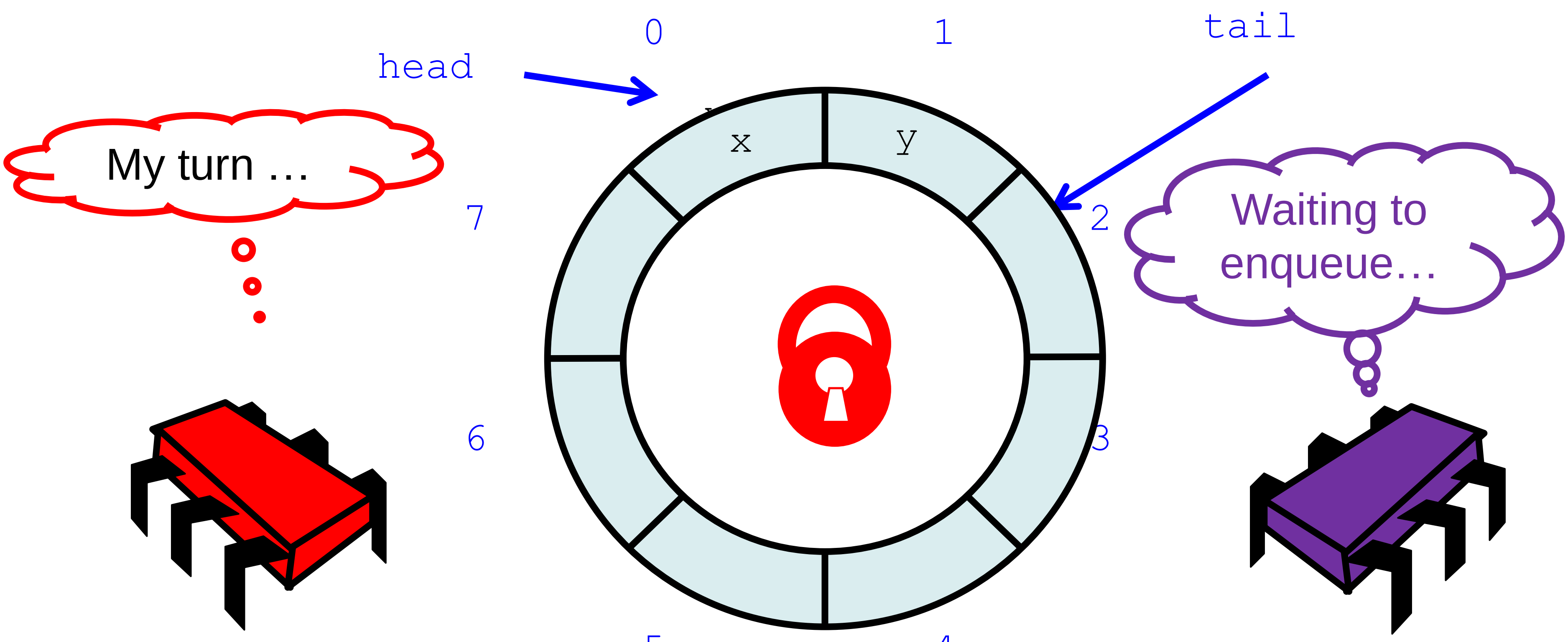
Initially `head = tail`

Lock-Based **deq()**





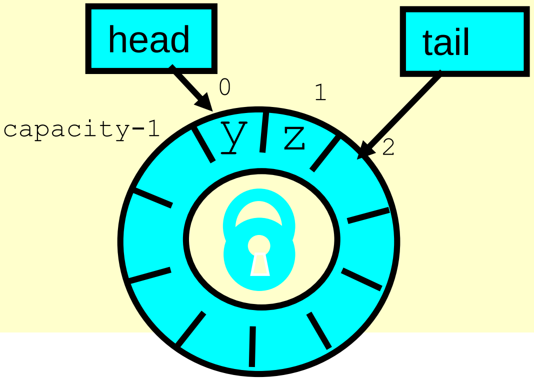
Acquire Lock



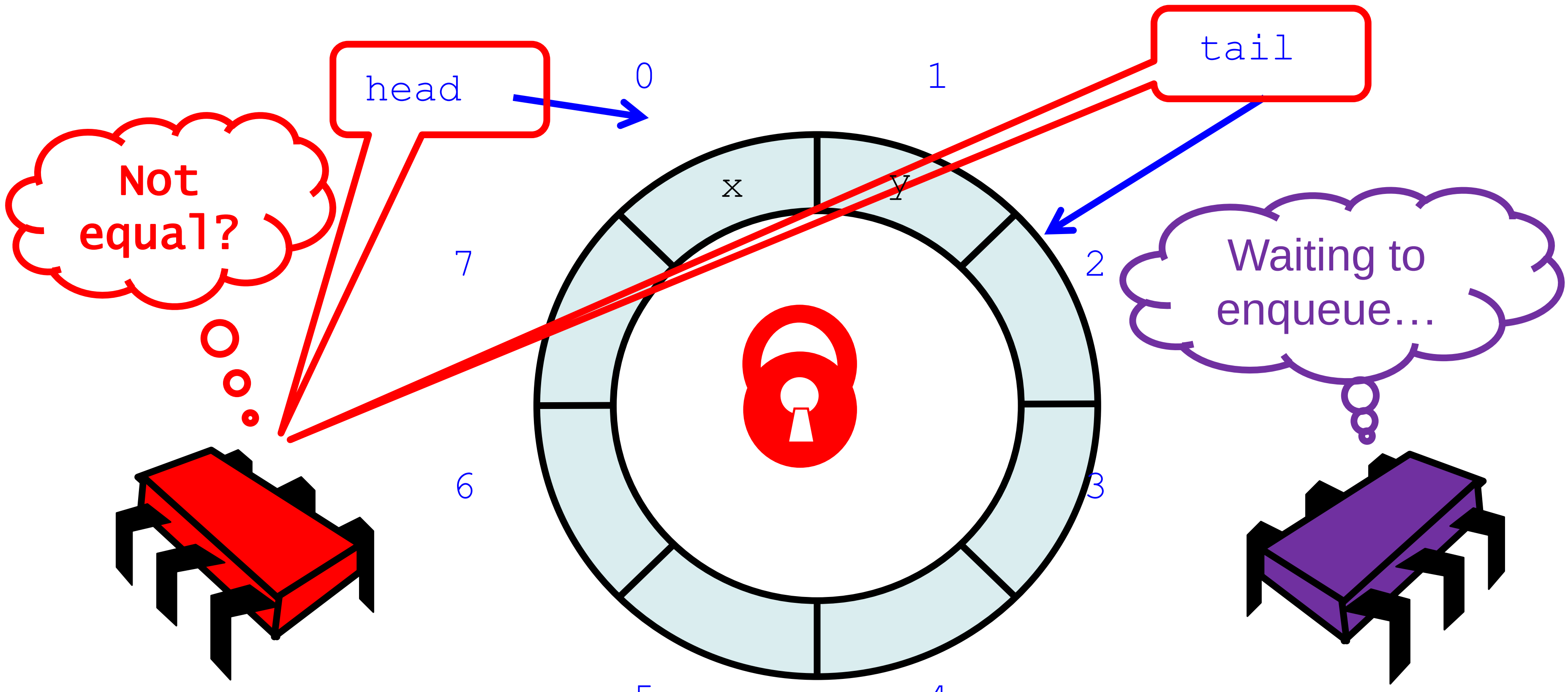
Implementation: **deq()**

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

Acquire lock at method start



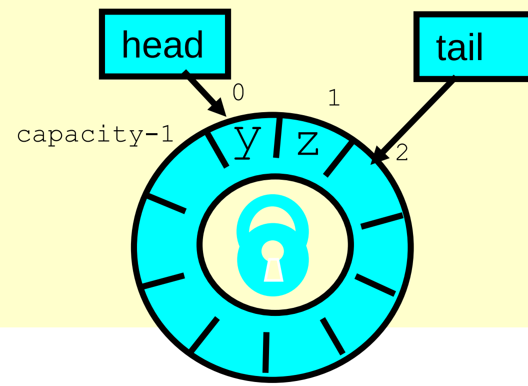
Check if Non-Empty



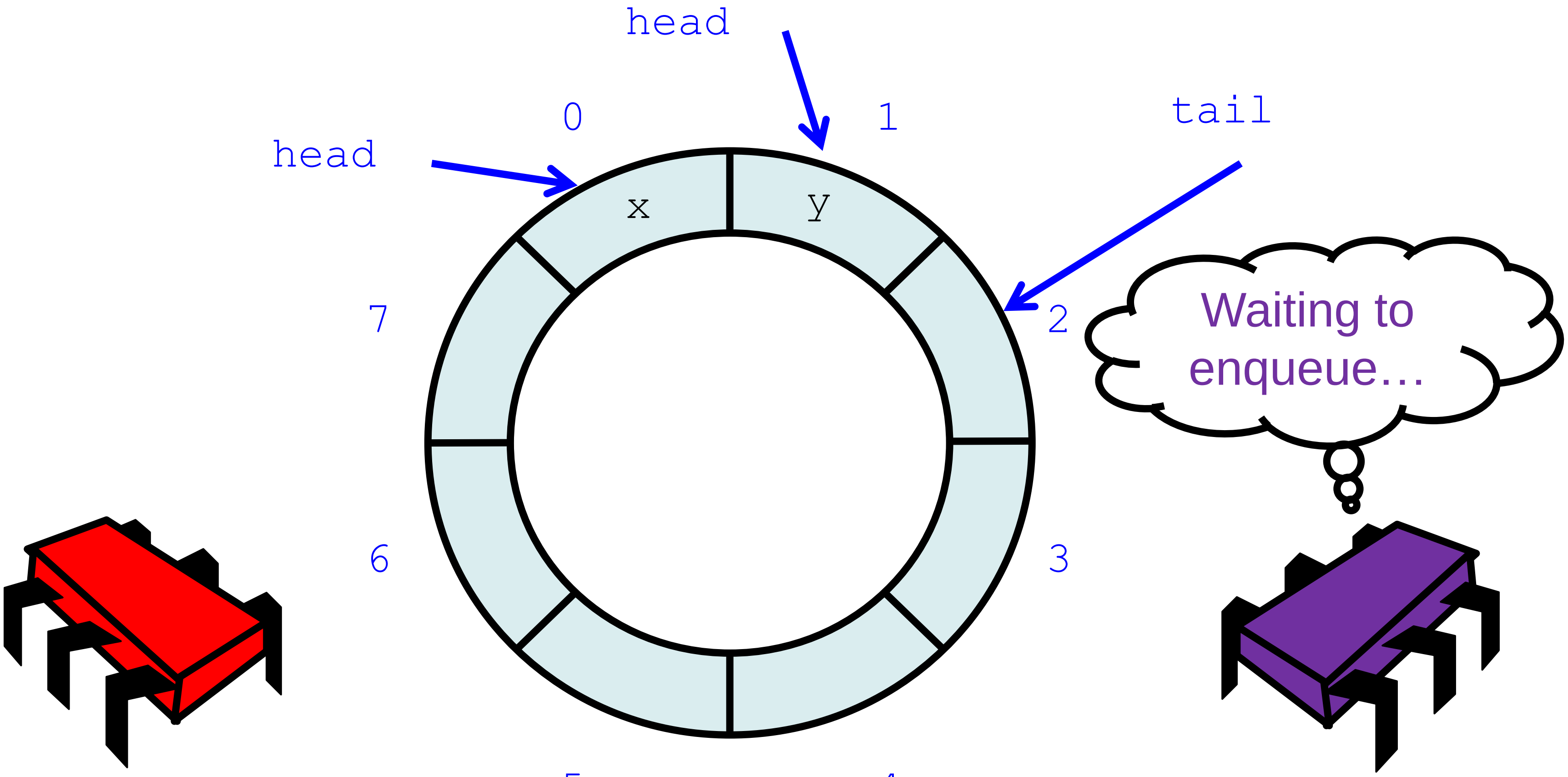
Implementation: **deq()**

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

If queue empty
throw exception



Modify the Queue

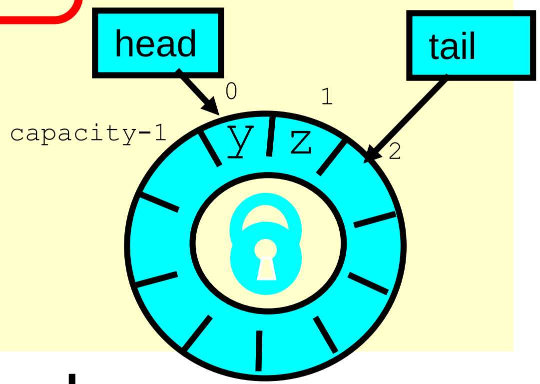


Implementation: **deq()**

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

Queue not empty?

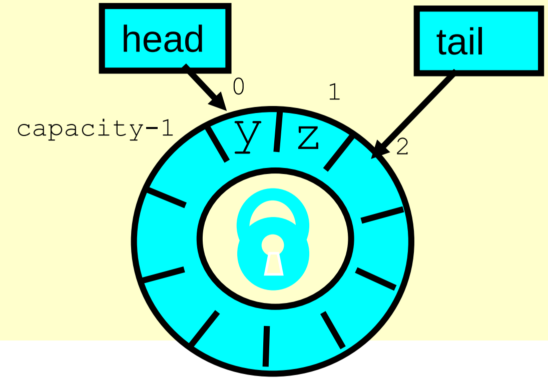
Remove item and update head



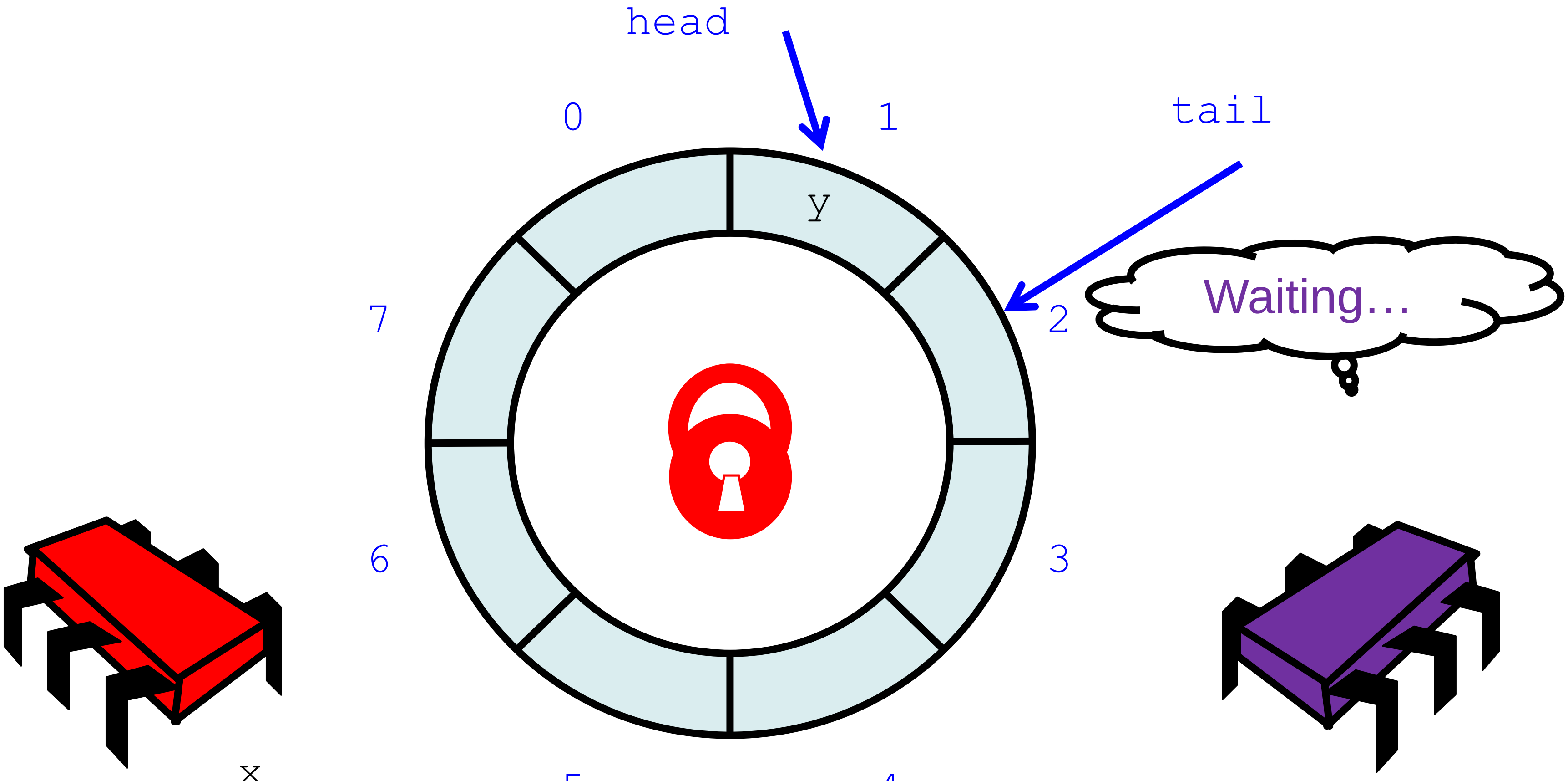
Implementation: **deq()**

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

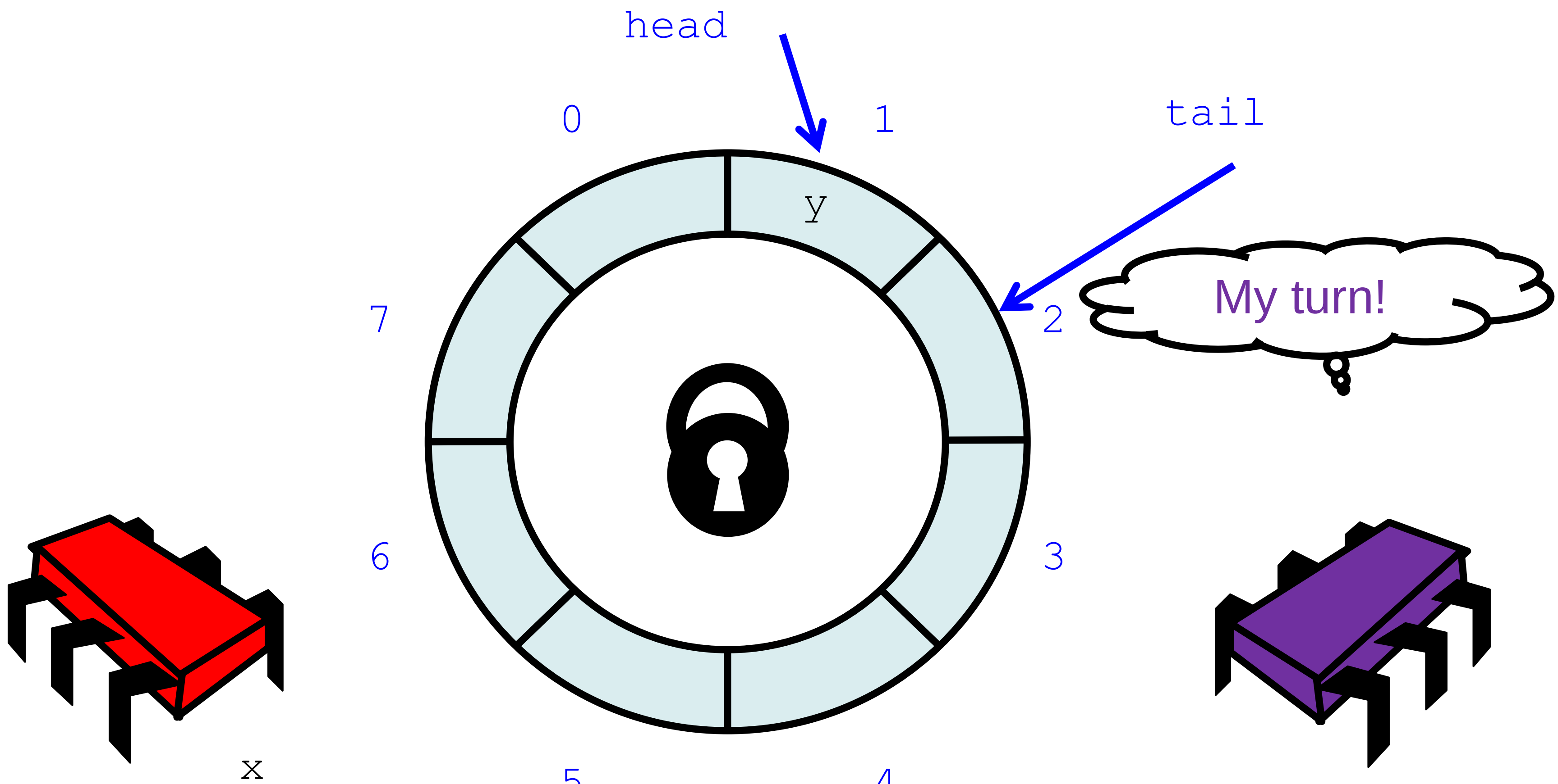
Return result



Release the Lock

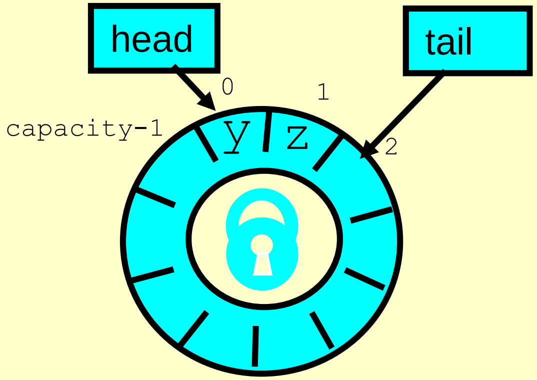


Release the Lock



Implementation: **deq()**

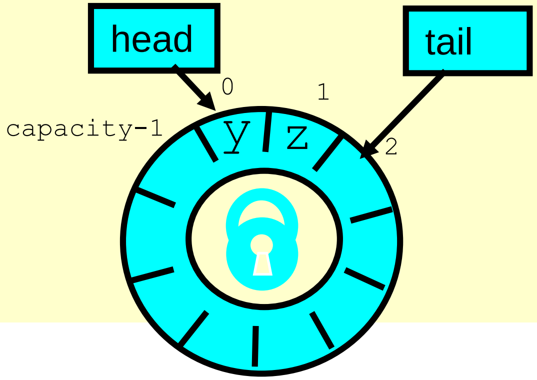
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



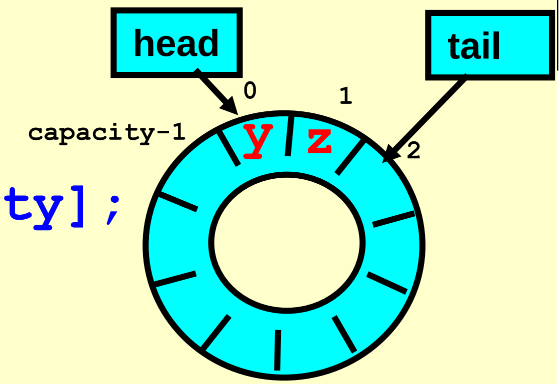
Release lock no matter what!

Implementation: **enq()**

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



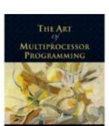
Wait-free Queue?




```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    if (tail-head == capacity) throw
         new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

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™**
- A linearizable object: one all of whose possible executions are linearizable

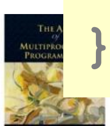


Wait-free Queue?

```
public clas   tFreeQueue {

  int          l =
  i            new Ob

           d enq(Item x) {
       ail-head == capacity) throw
        new FullException();
     items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
        new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

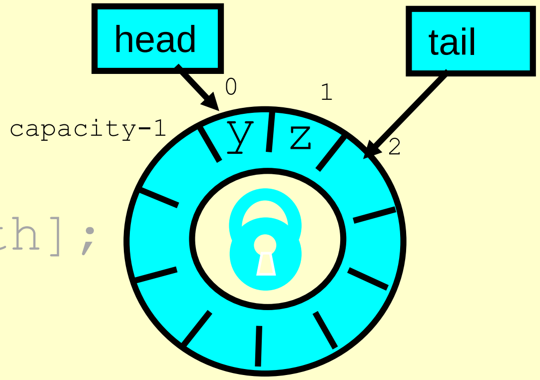
Remember that there
Is only one enqueuer
and only one dequeuer

Linearization order is order head and tail fields modified



Reasoning About Linearizability: Locking

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



Linearization points are when locks are released