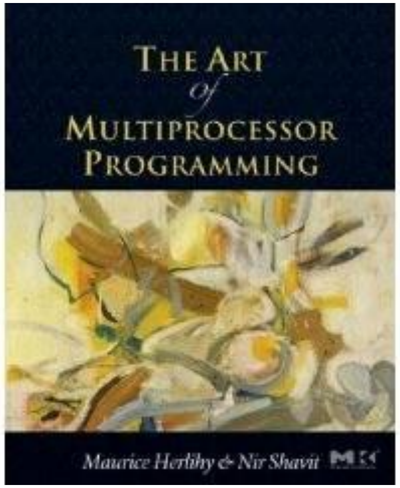
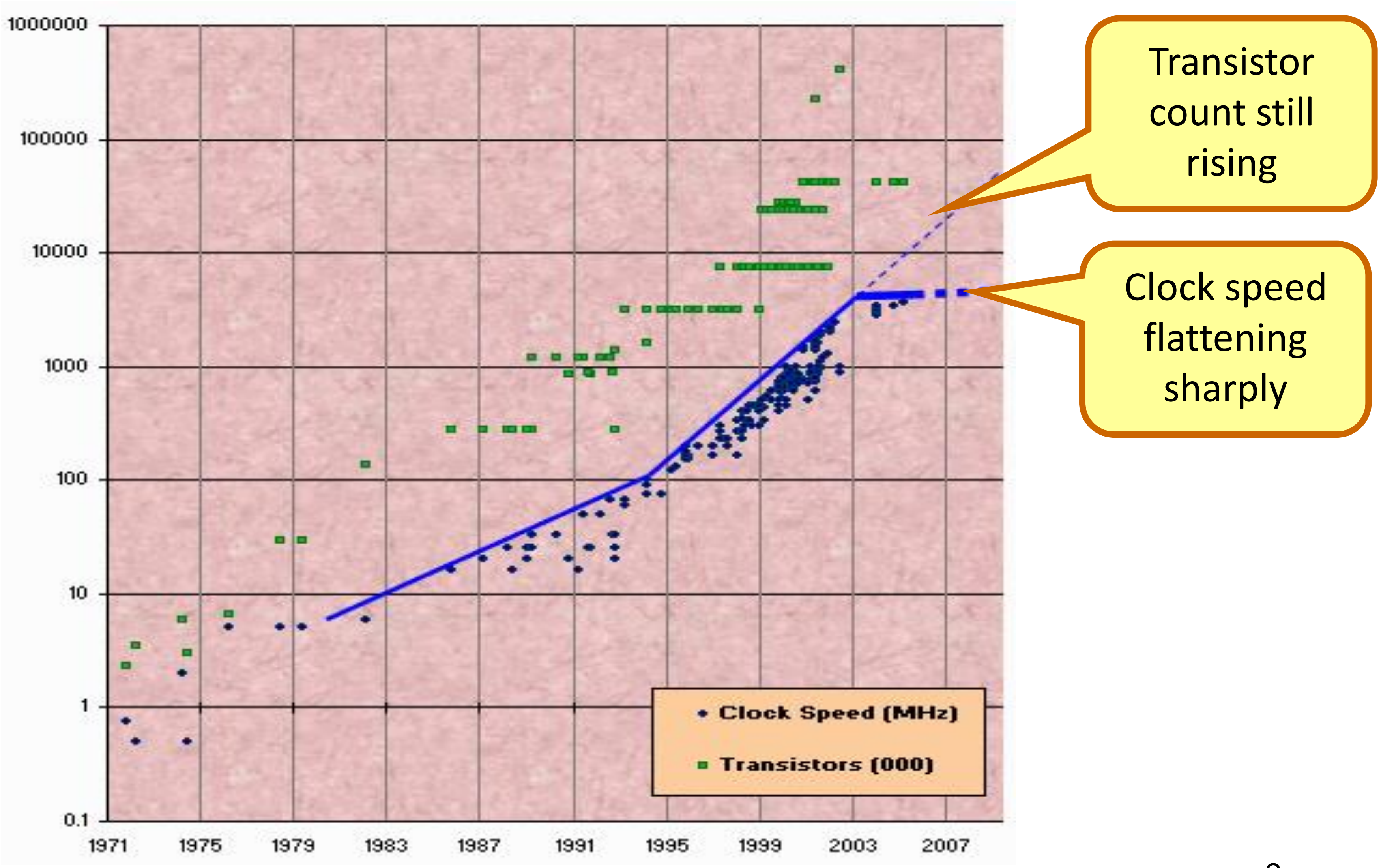
# Programming Paradigms for Concurrency
# Introduction




Based on companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

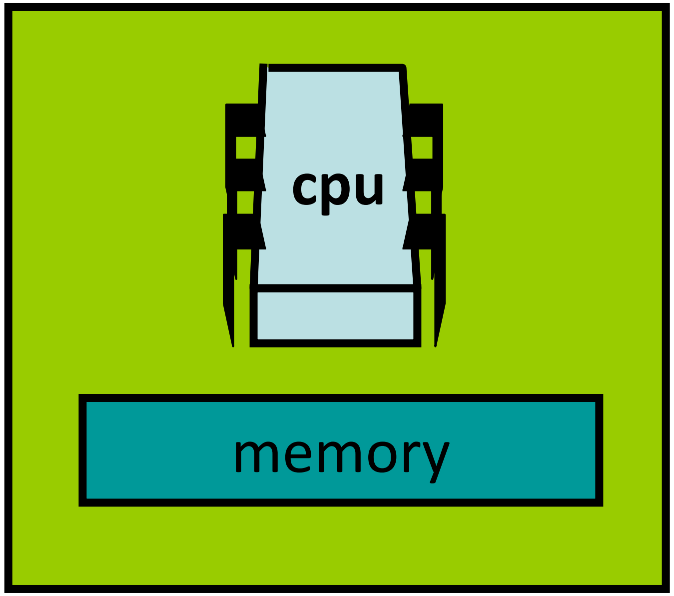
Modified by
Thomas Wies
New York University

# Moore's Law

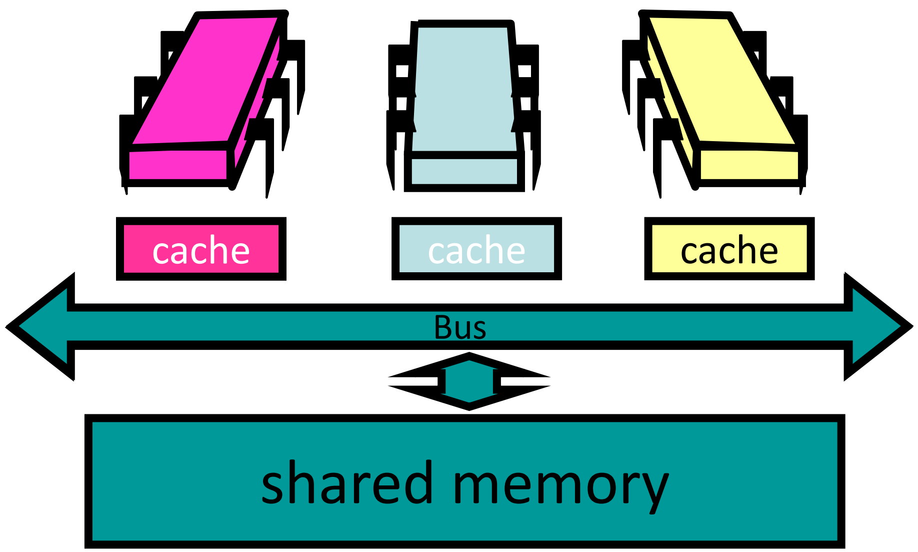

1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
Clock Speed (MHz)
Transistors (000)
Transistor count still rising
Clock speed flattening sharply

# Moore’s Law (in practice)

# Nearly Extinct: the Uniprocesor

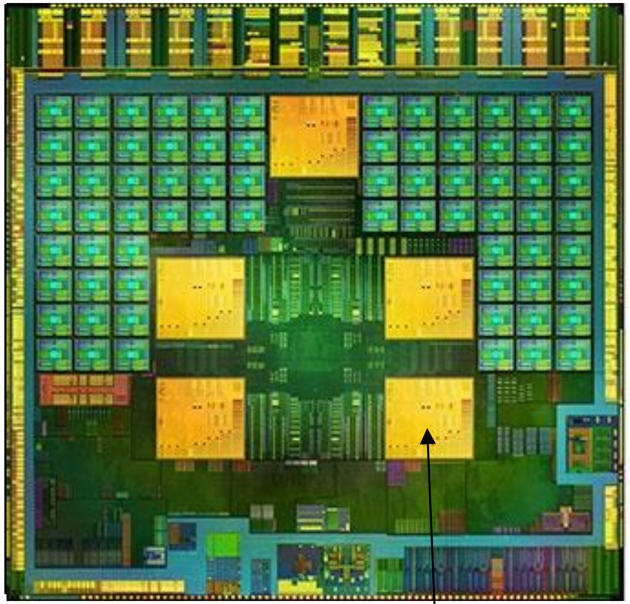


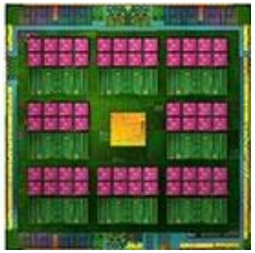
# Endangered: The Shared Memory Multiprocessor (SMP)

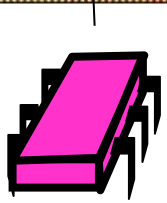


# The New Boss: The Multicore Processor (CMP)

All on the same chip

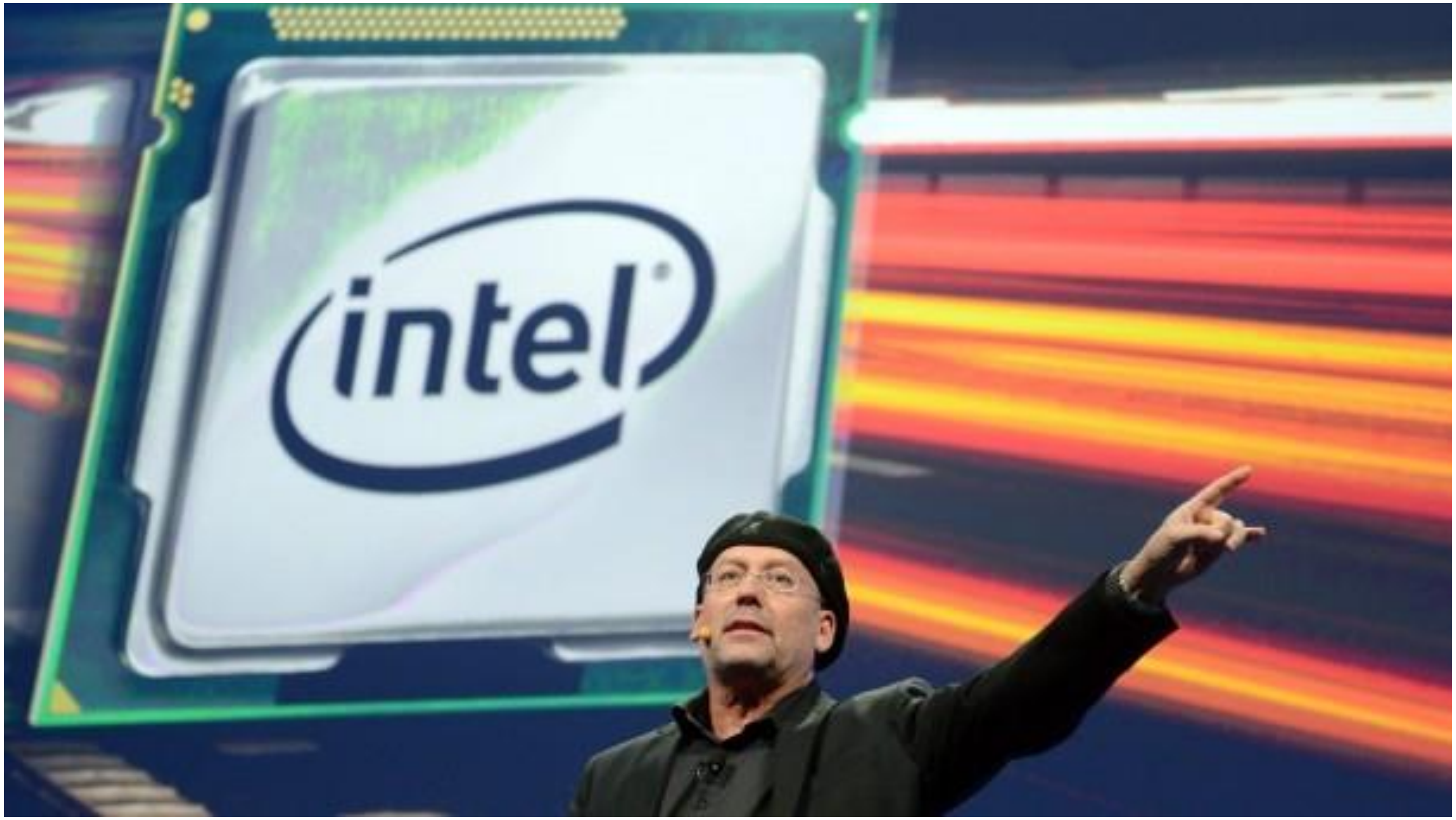




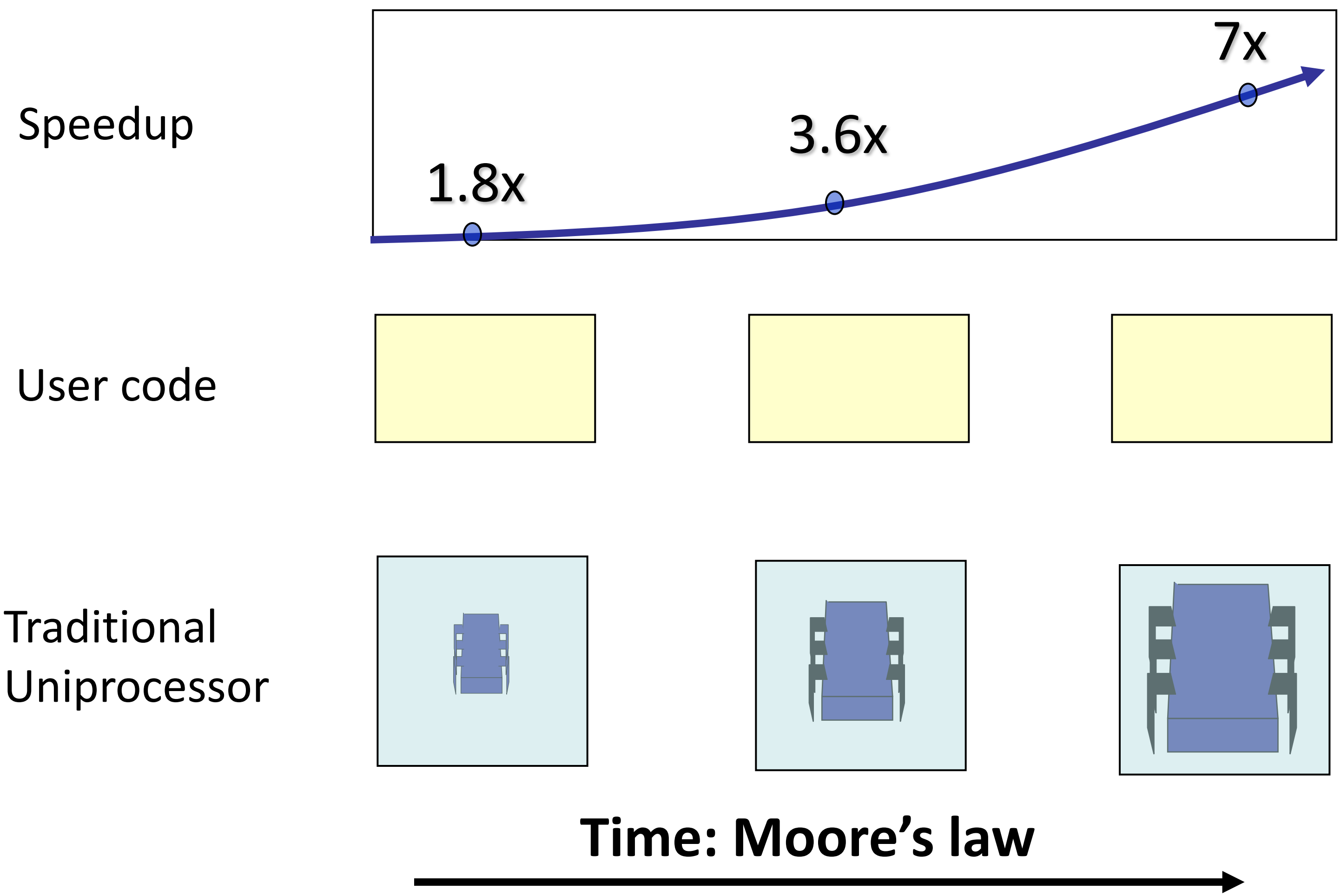


**NVidia Tegra 4 with ARM Cortex-A15**

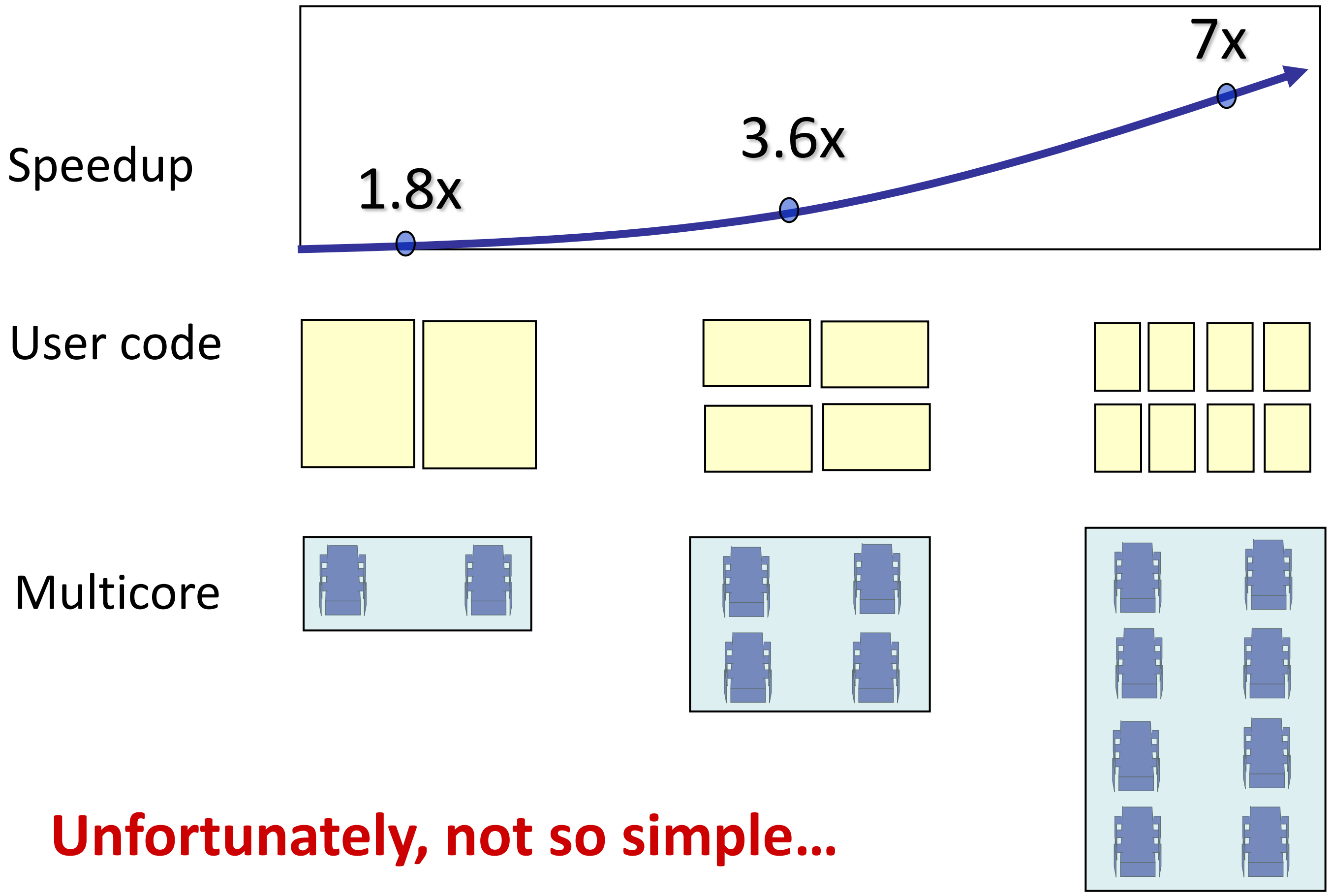
# Why is Mooly Eden Smiling?



# Traditional Scaling Process

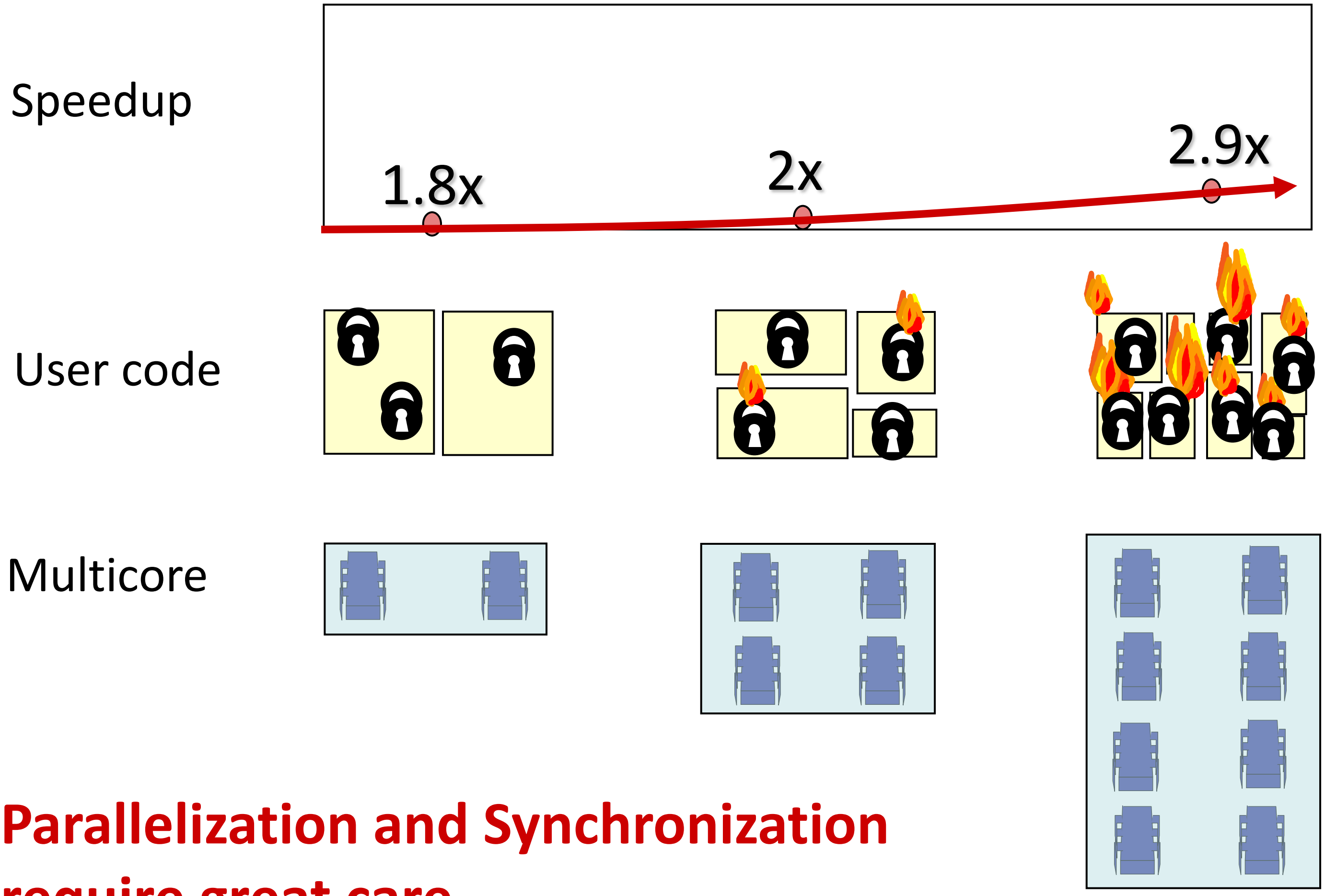


# Ideal Scaling Process




**Unfortunately, not so simple…**

# Actual Scaling Process




**Parallelization and Synchronization require great care…**

# Course Overview

- Fundamentals
  - models, algorithms, impossibility
- Real-World programming
  - architectures
  - synchronization primitives
    - spin-locks, monitors, barriers
  - paradigms and techniques
    - shared memory concurrency
    - transactional memories
    - message passing

# Languages Used

- Java
  - shared memory concurrency
- Scala
  - software transactional memory
  - message passing (actors)

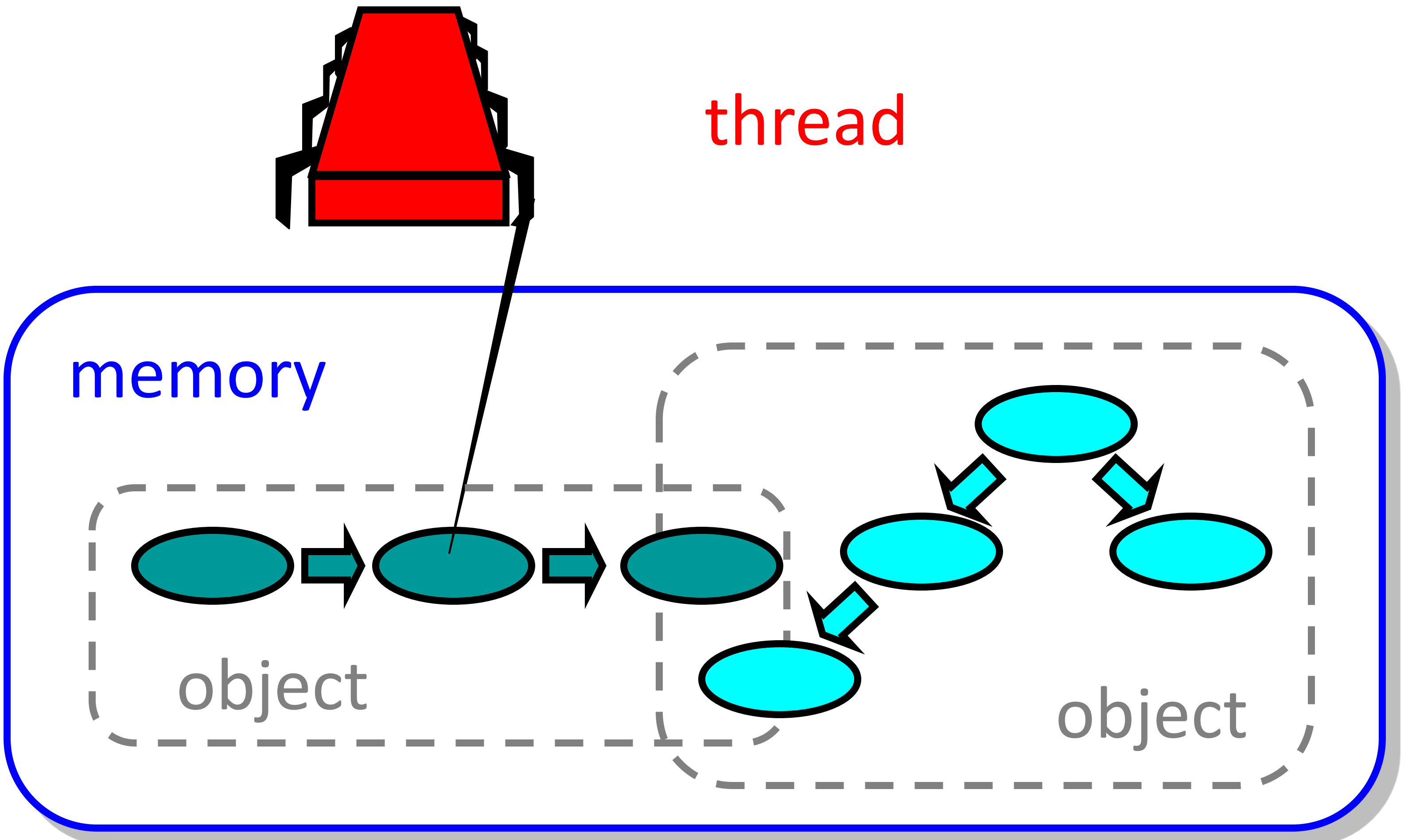
# Administrative Issues

- Office Hours: Tue 3-4pm, or by appointment
- Office: CIWW 407
- Course web site:
  http://cs.nyu.edu/wies/teaching/ppc-14
- Mailing list:
  csci_ga_3033_014_sp14@cs.nyu.edu

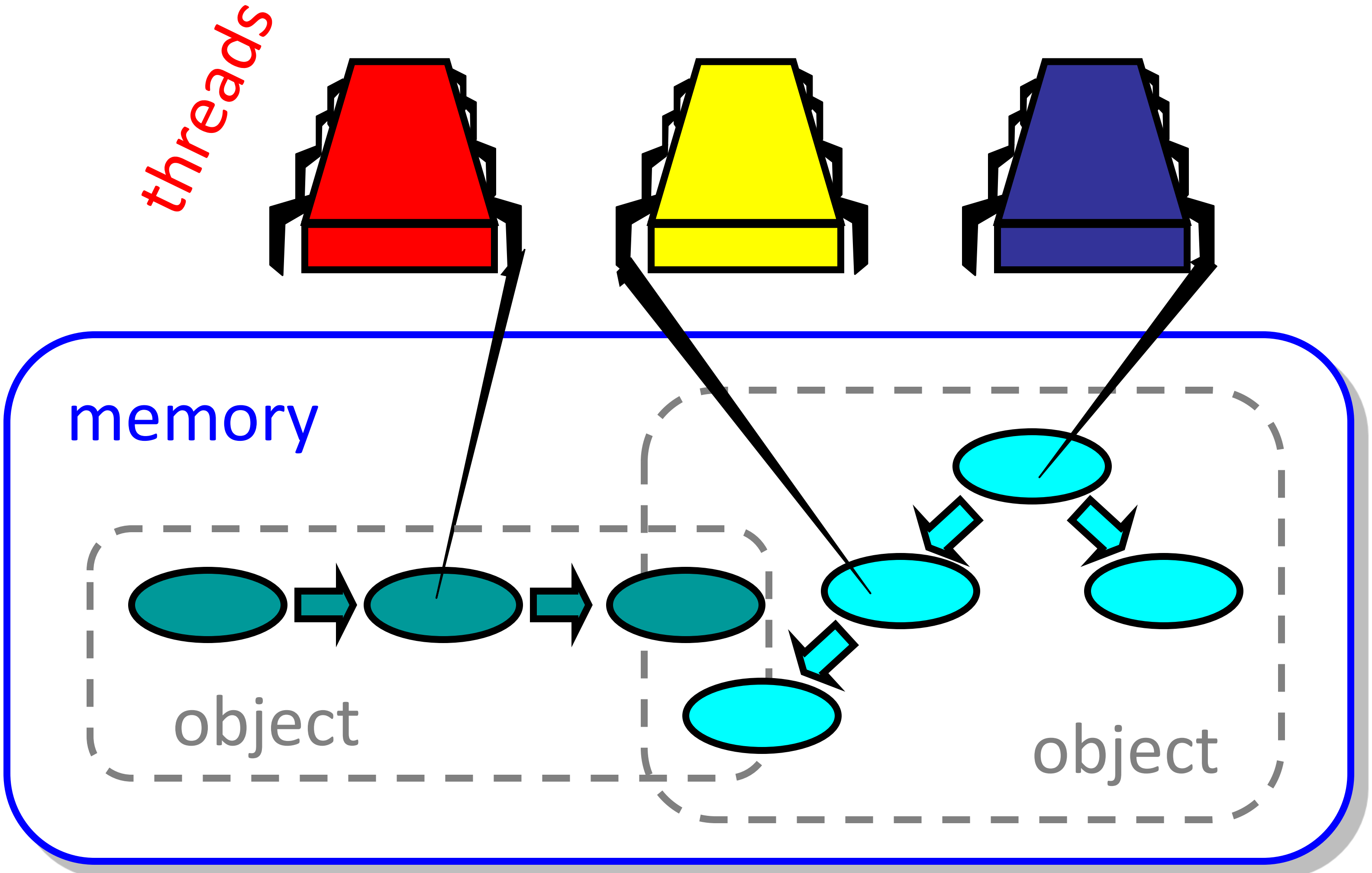
# Grading

- Weekly Assignments: 30%
- Term Project: 30%
- Final Exam: 40%

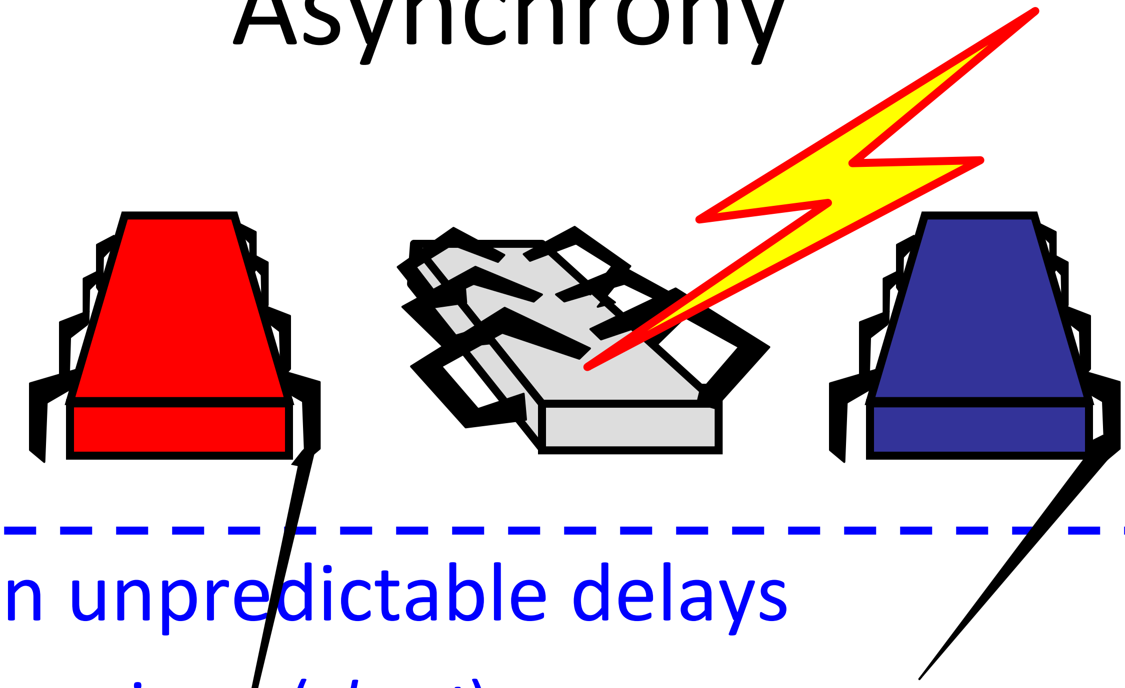
# Sequential Computation



# Concurrent Computation



# Asynchrony



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

# Model Summary

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

# Road Map

- We are going to focus on principles first, then practice
  - Start with idealized models
  - Look at simplistic problems
  - Emphasize correctness over pragmatism
  - “Correctness may be theoretical, but incorrectness has practical impact”

# Toyota Unintended Acceleration Incidents (2009-2010)

- at least one fatal accident
- more than 9 million vehicles recalled
- In a 2013 court case, unprotected critical variables in Toyota's real-time OS were linked to the incidents

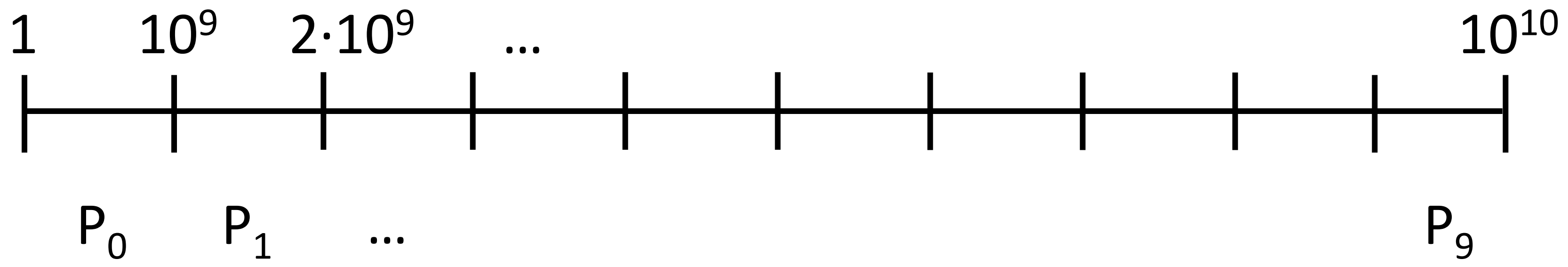
# Concurrency Jargon

- 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

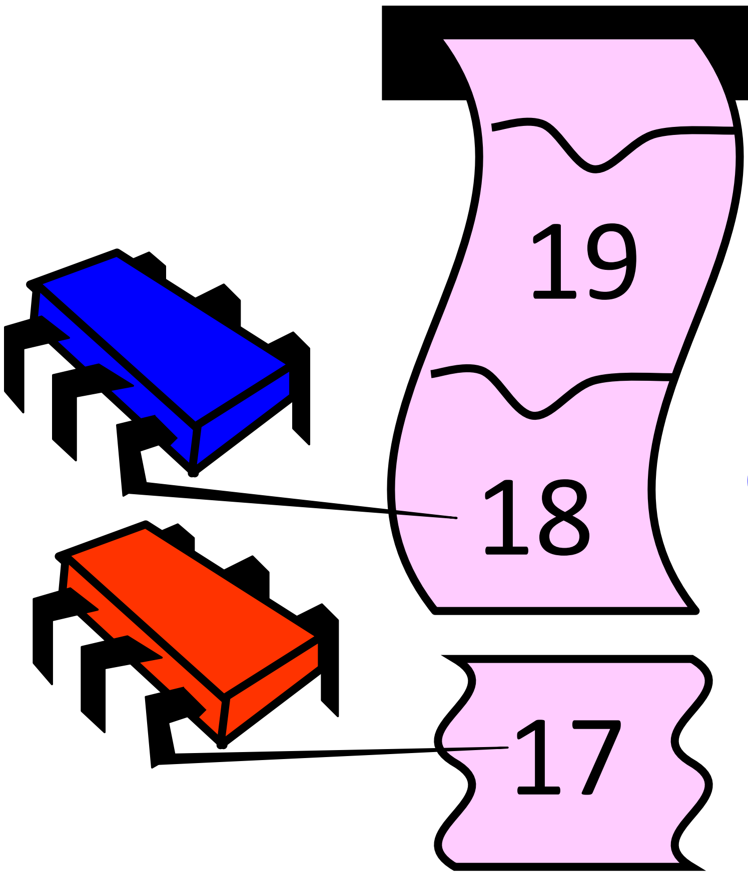
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict

# 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*

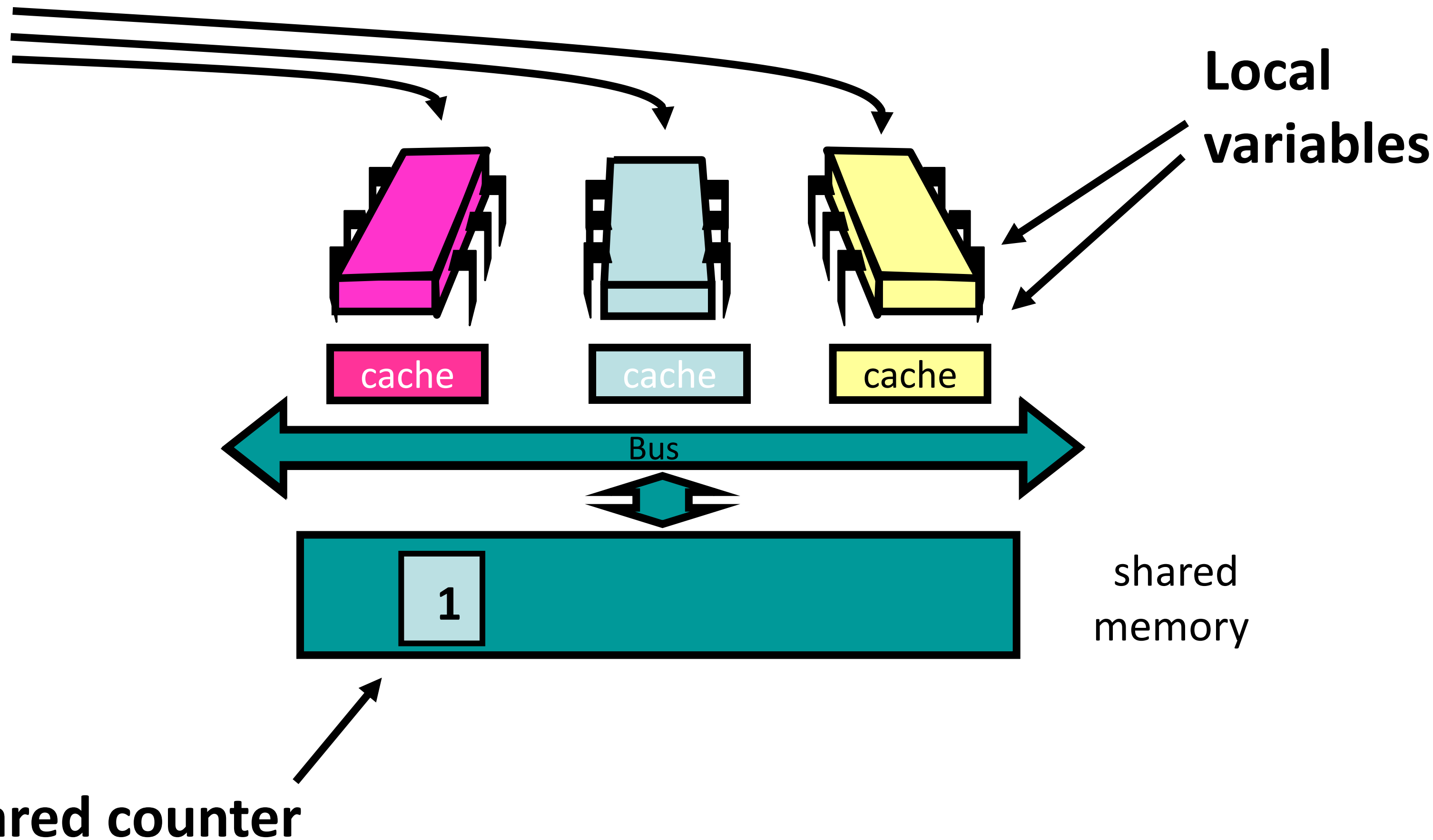
```
Counter counter = new Counter(1);

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*10⁹+1,
j<(i+1)*10⁹; 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

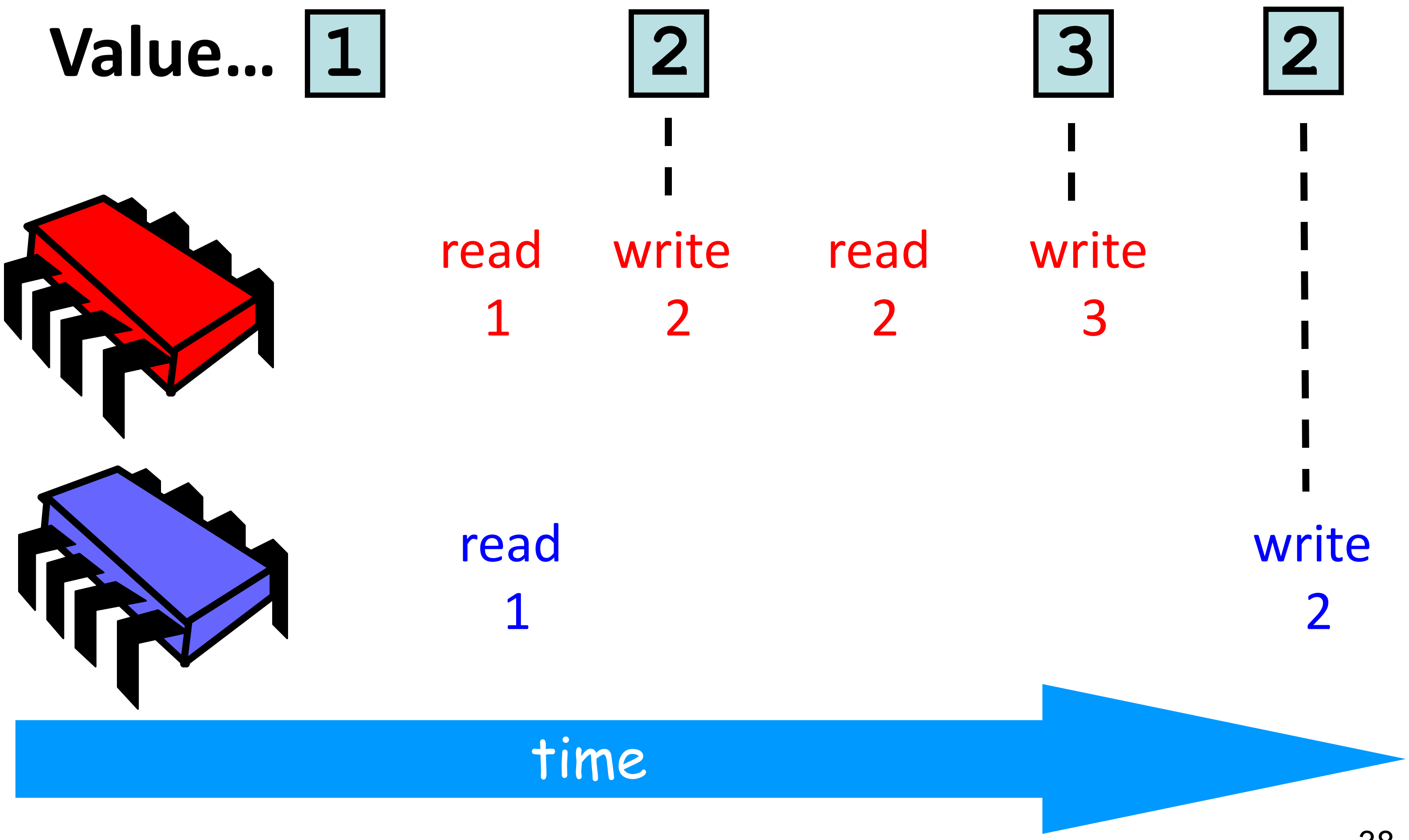
```
public class Counter {
  private long value;

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

# What It Means

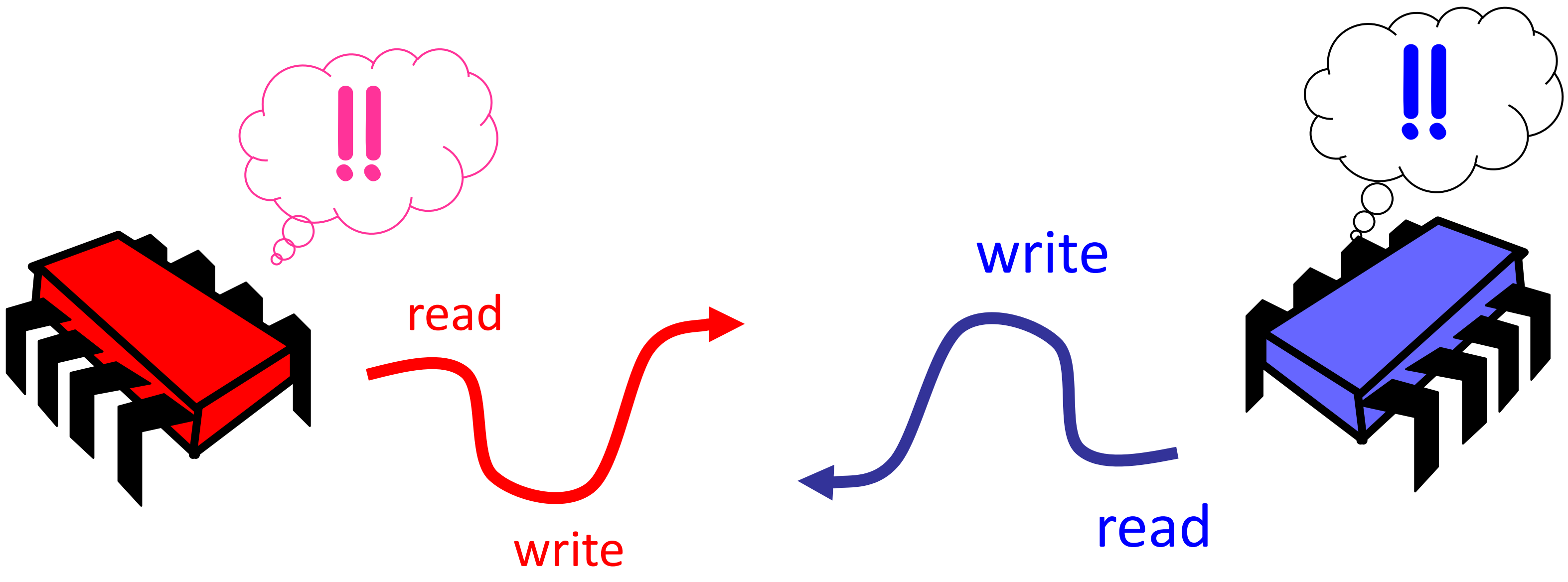
```
public class Counter {
  private long value;

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

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

# Not so good…



# Is this problem inherent?




If we could only glue reads and writes together…

# Challenge

```
public class Counter {
  private long value;

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

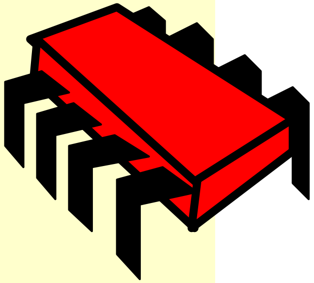
# Challenge

```
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;
  }
}
```

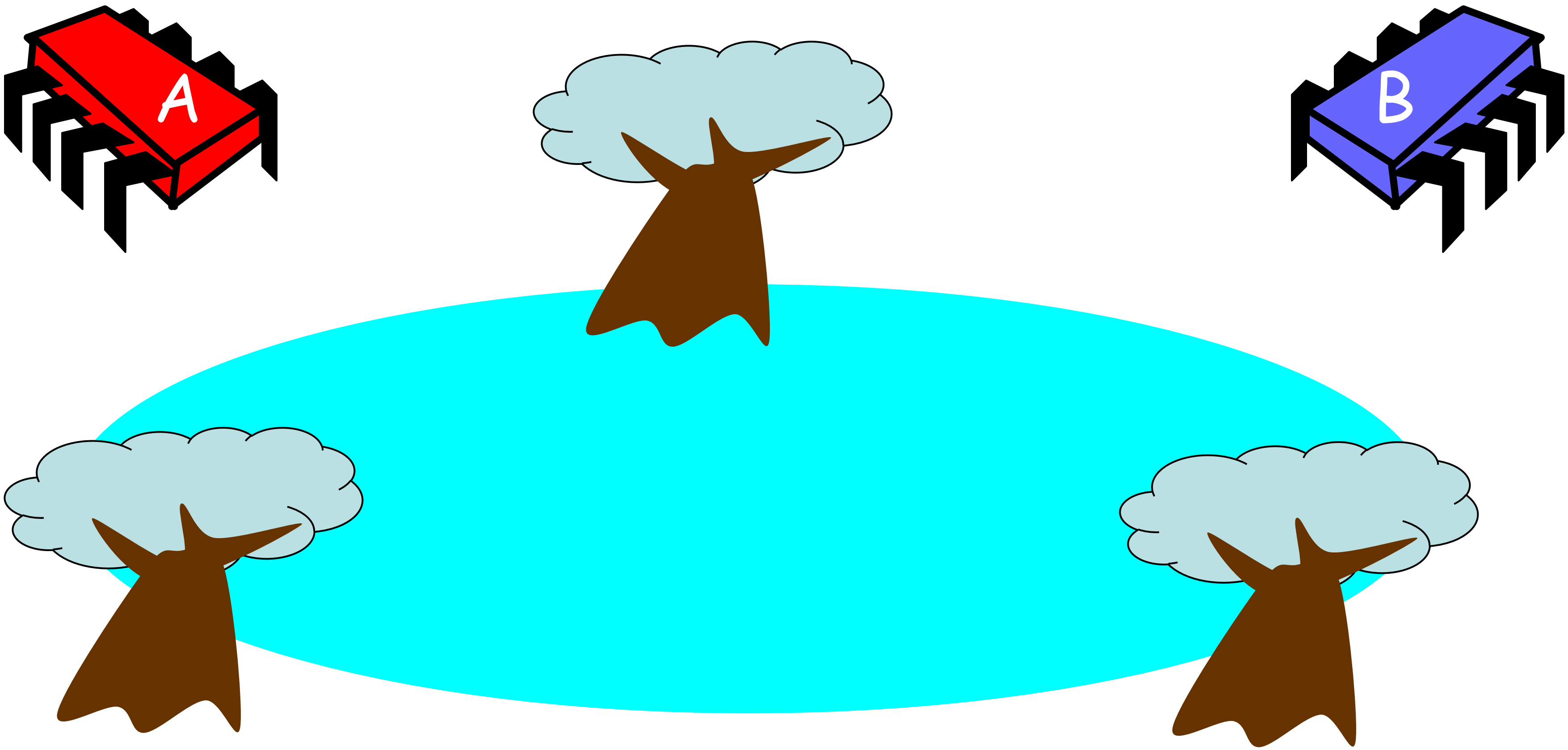
Synchronized block

# An Aside: Java™

Mutual Exclusion

```
public class Counter {
  private long value;

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

# 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

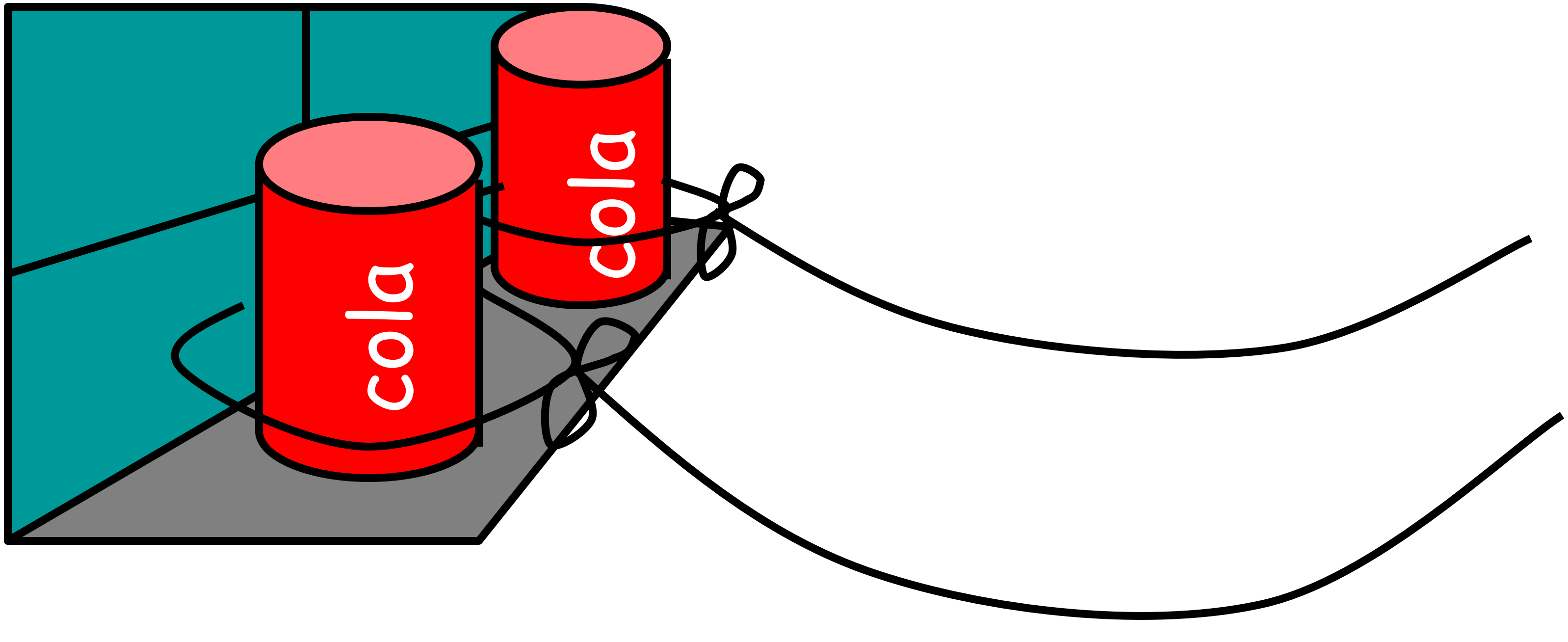
# Cell Phone Protocol

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

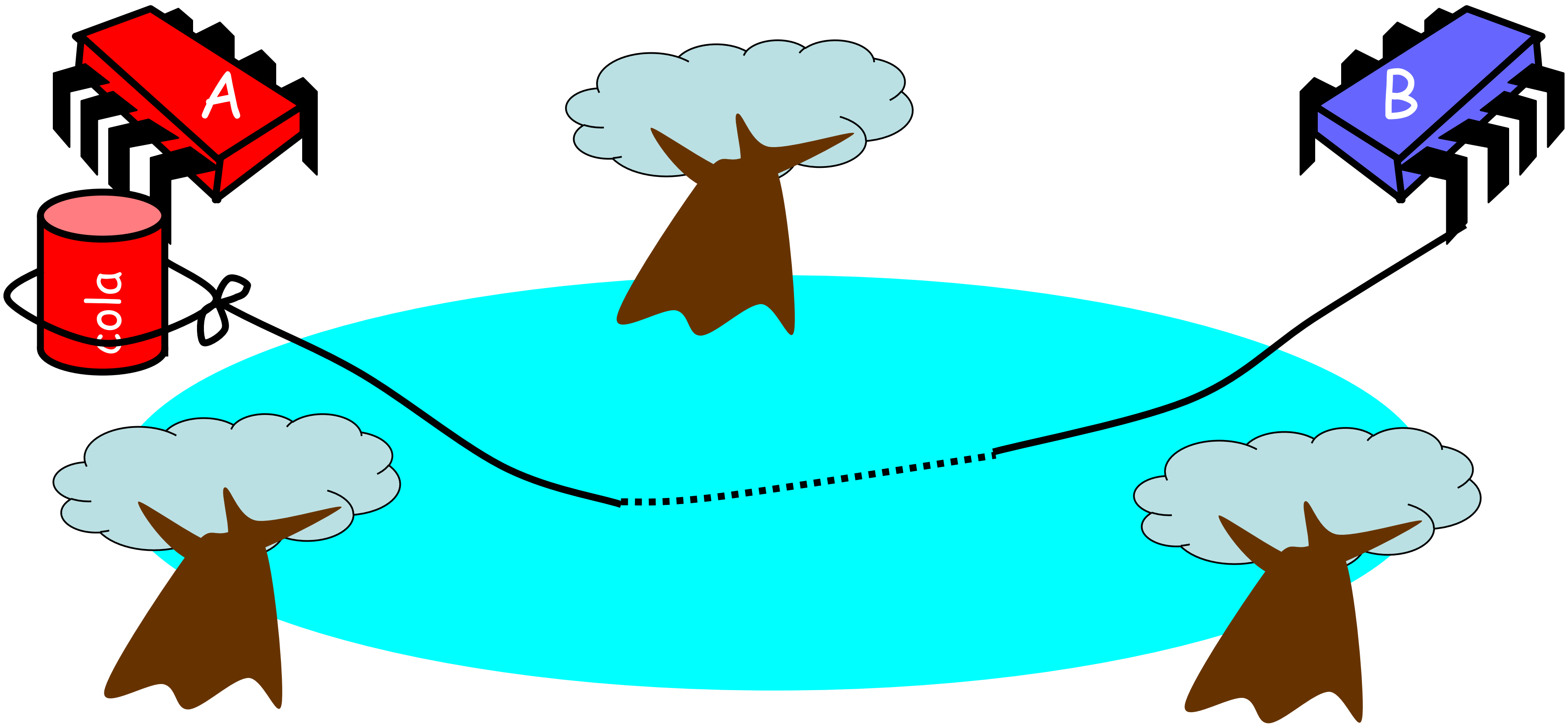
# Interpretation

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

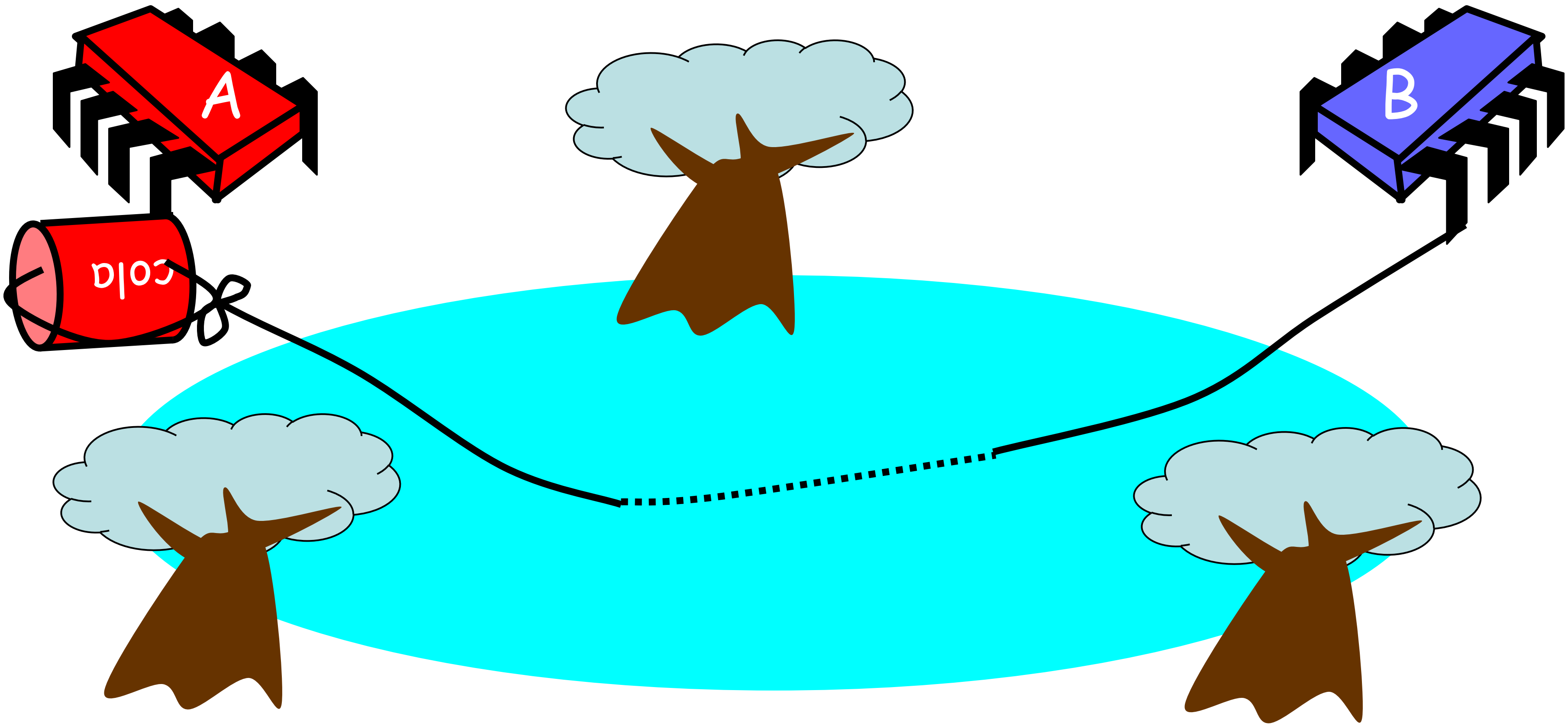
# Can Protocol



# Bob conveys a bit



# Bob conveys a bit



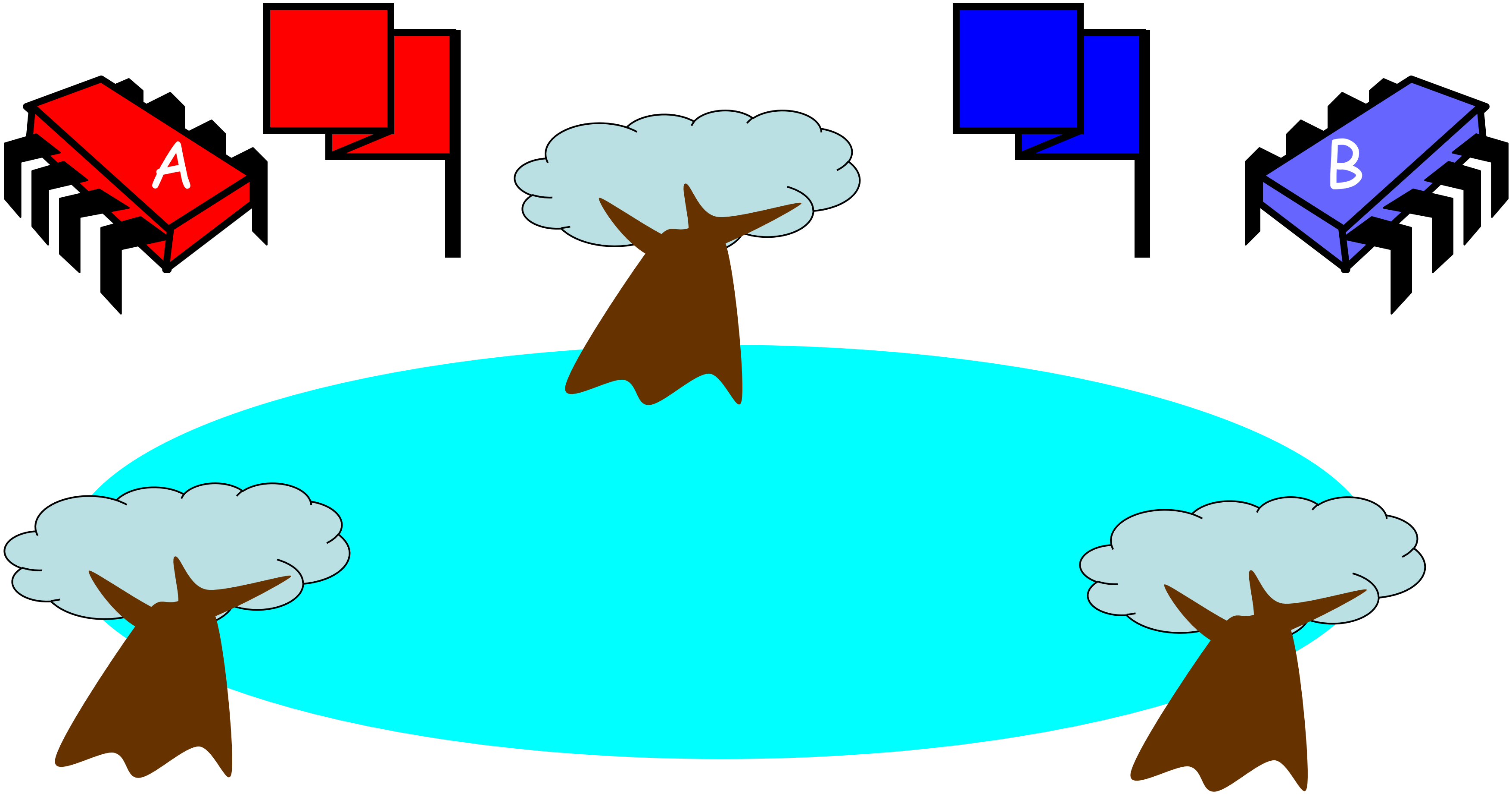
# Can Protocol

- Idea
  - Cans on Alice's windowsill
  - Strings lead to Bob's house
  - Bob pulls strings, knocks over cans
- Gotcha
  - Cans cannot be reused
  - Bob runs out of cans

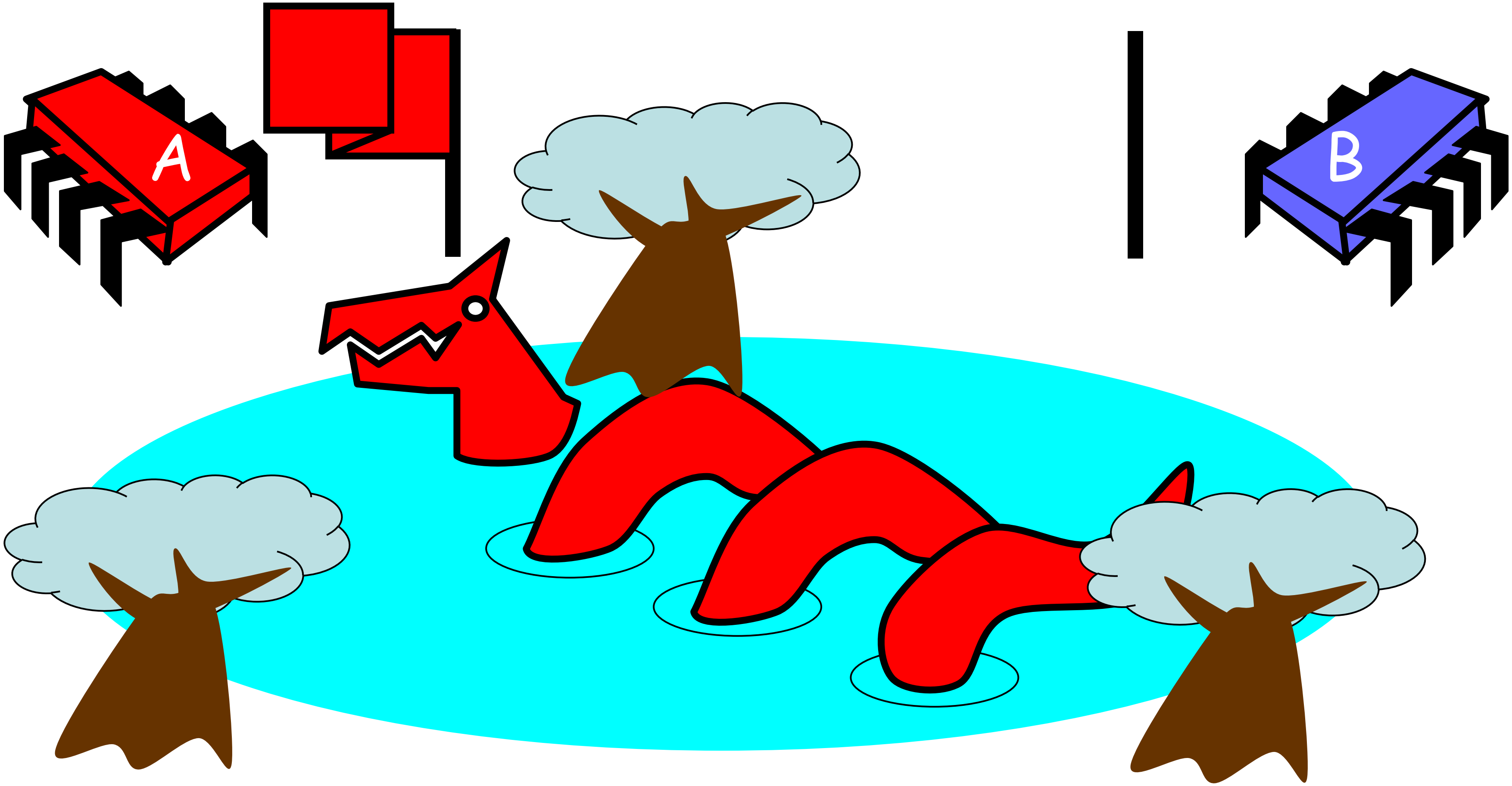
# Interpretation

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

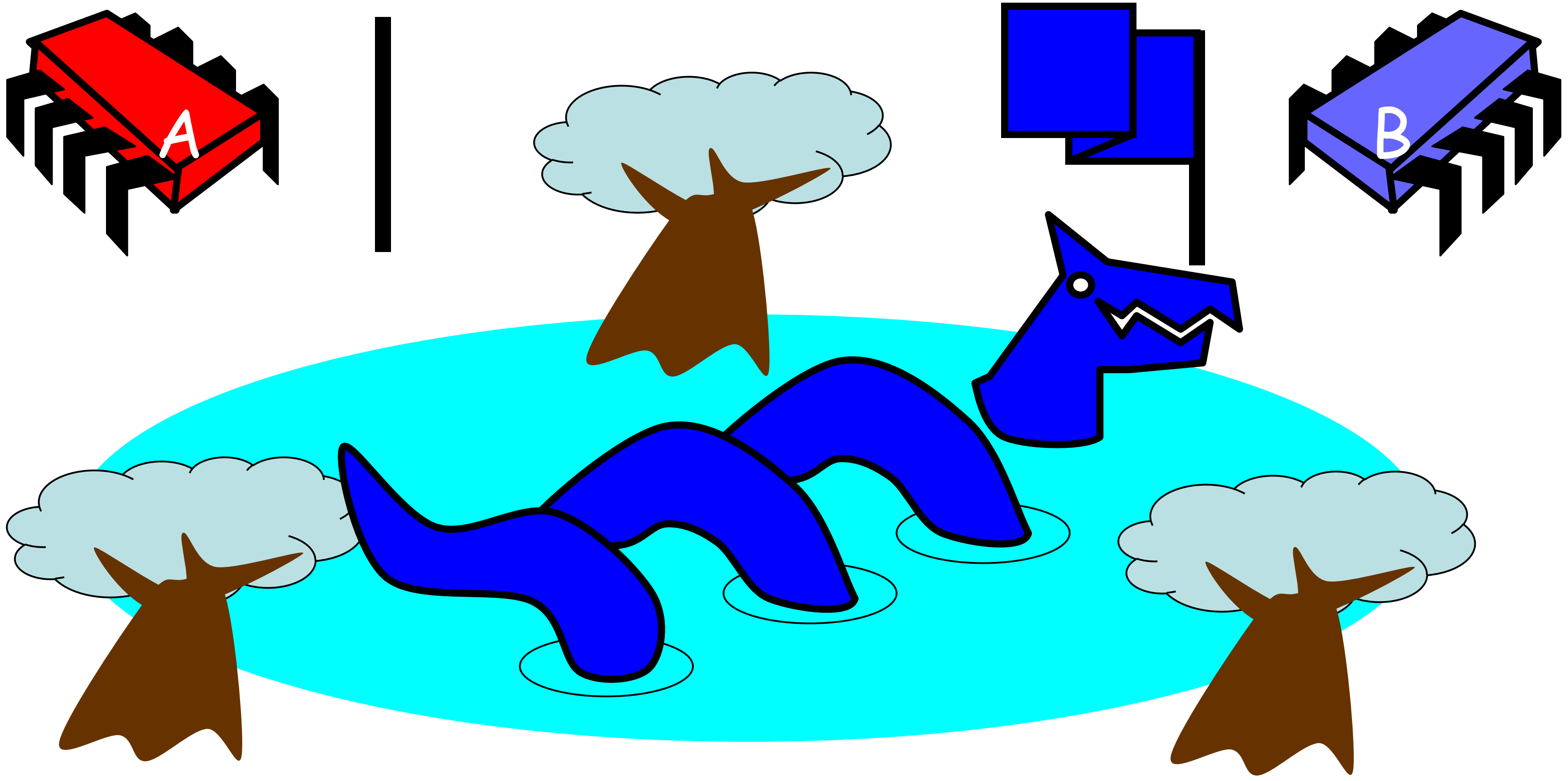
# Flag Protocol



# 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

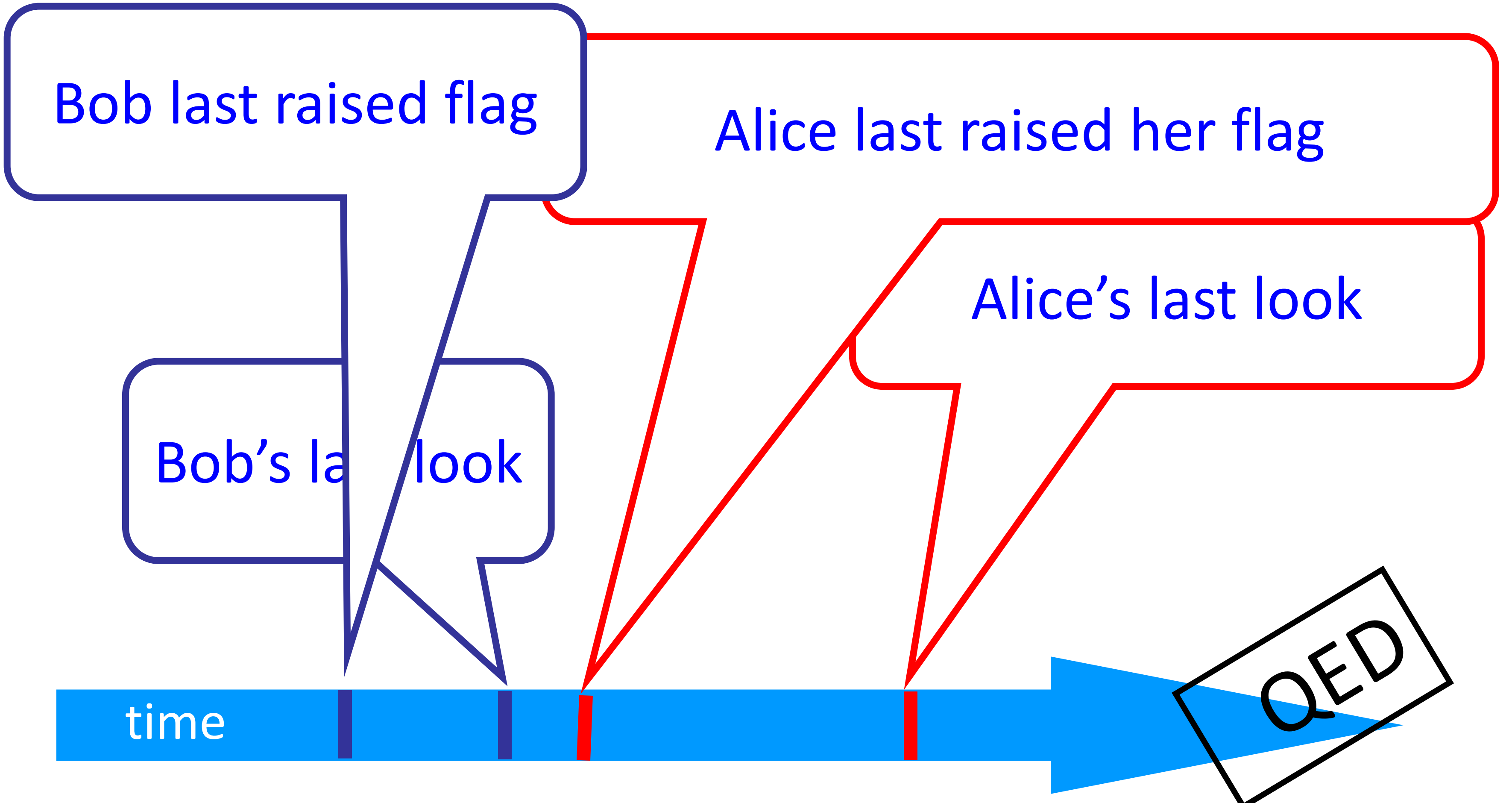
# The Flag Principle

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

# Proof of Mutual Exclusion

- Assume both pets in pond
  - Derive a contradiction
  - By reasoning **backwards**
- Consider the last time Alice and Bob each looked before letting the pets in
- Without loss of generality assume Alice was the last to look…

# Proof




**Alice must have seen Bob’s Flag. A Contradiction**

# Proof of No Deadlock

- If only one pet wants in, it gets in.

# Proof of No Deadlock

- If only one pet wants in, it gets in.
- Deadlock requires both continually trying to get in.

# Proof of No Deadlock

- If only one pet wants in, it gets in.
- Deadlock requires both continually trying to get in.
- If Bob sees Alice’s flag, he gives her priority (a gentleman…)

QED

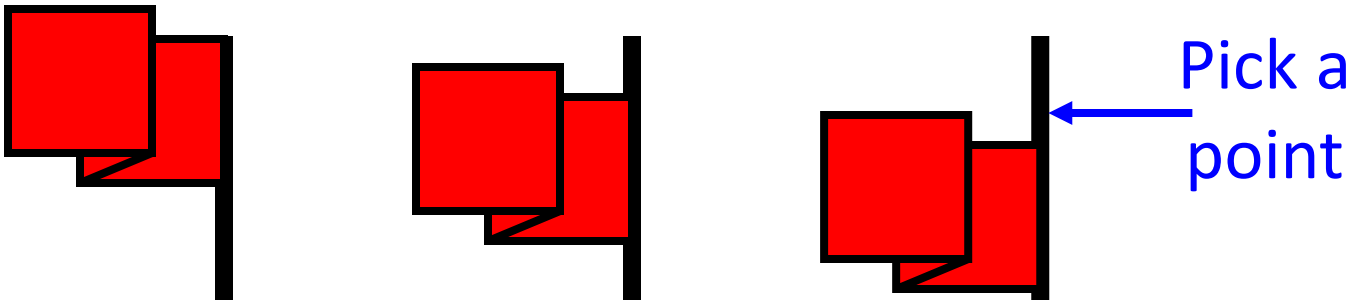
# Remarks

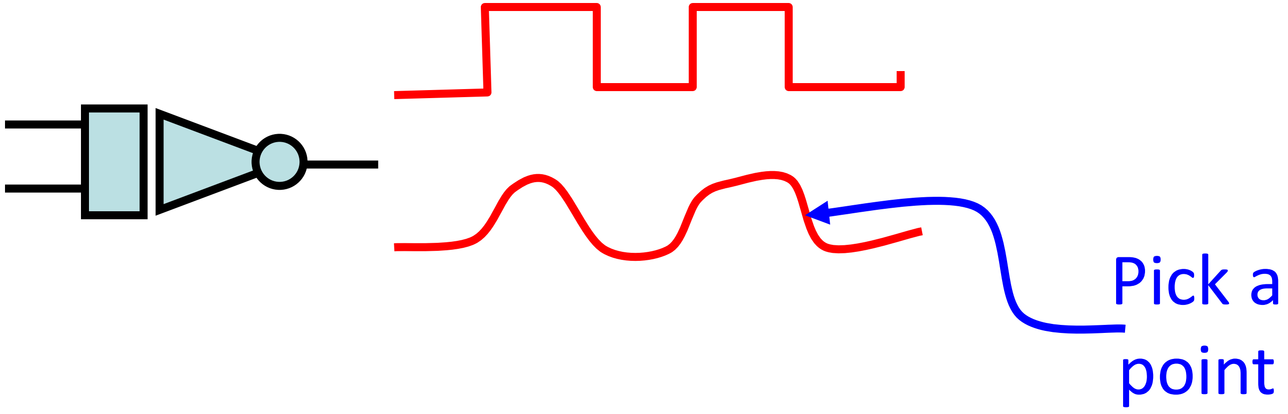
- Protocol is *unfair*
  - Bob's pet might never get in
- Protocol uses *waiting*
  - If Bob is eaten by his pet, Alice's pet might never get in

# Moral of Story

- Mutual Exclusion cannot be solved by
  - transient communication (cell phones)
  - interrupts (cans)
- It can be solved by
  - one-bit shared variables
  - that can be read or written

# The Arbiter Problem (an aside)



# The Fable Continues

- Alice and Bob fall in love & marry

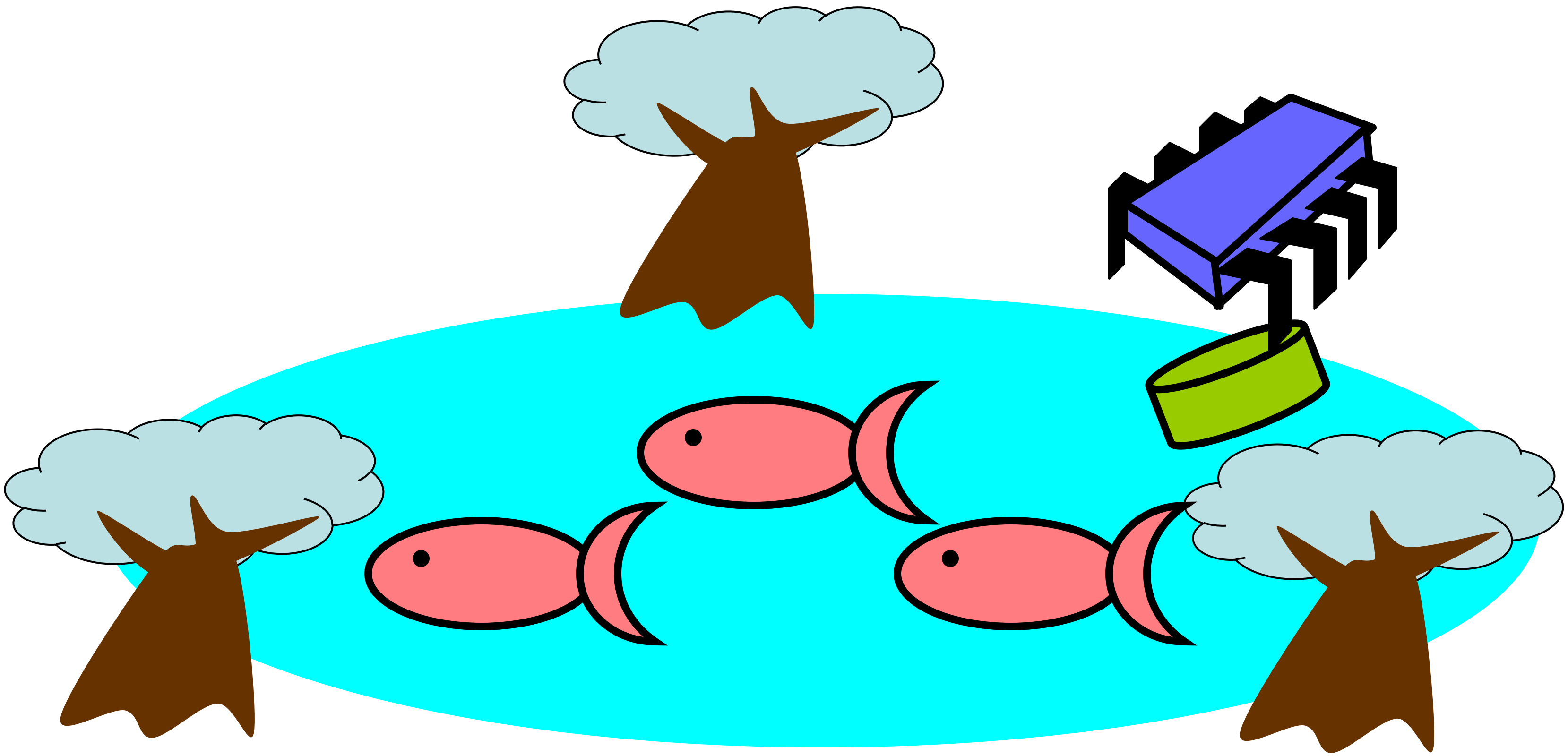
# The Fable Continues

- Alice and Bob fall in love & marry
- Then they fall out of love & divorce
  - She gets the pets
  - He has to feed them

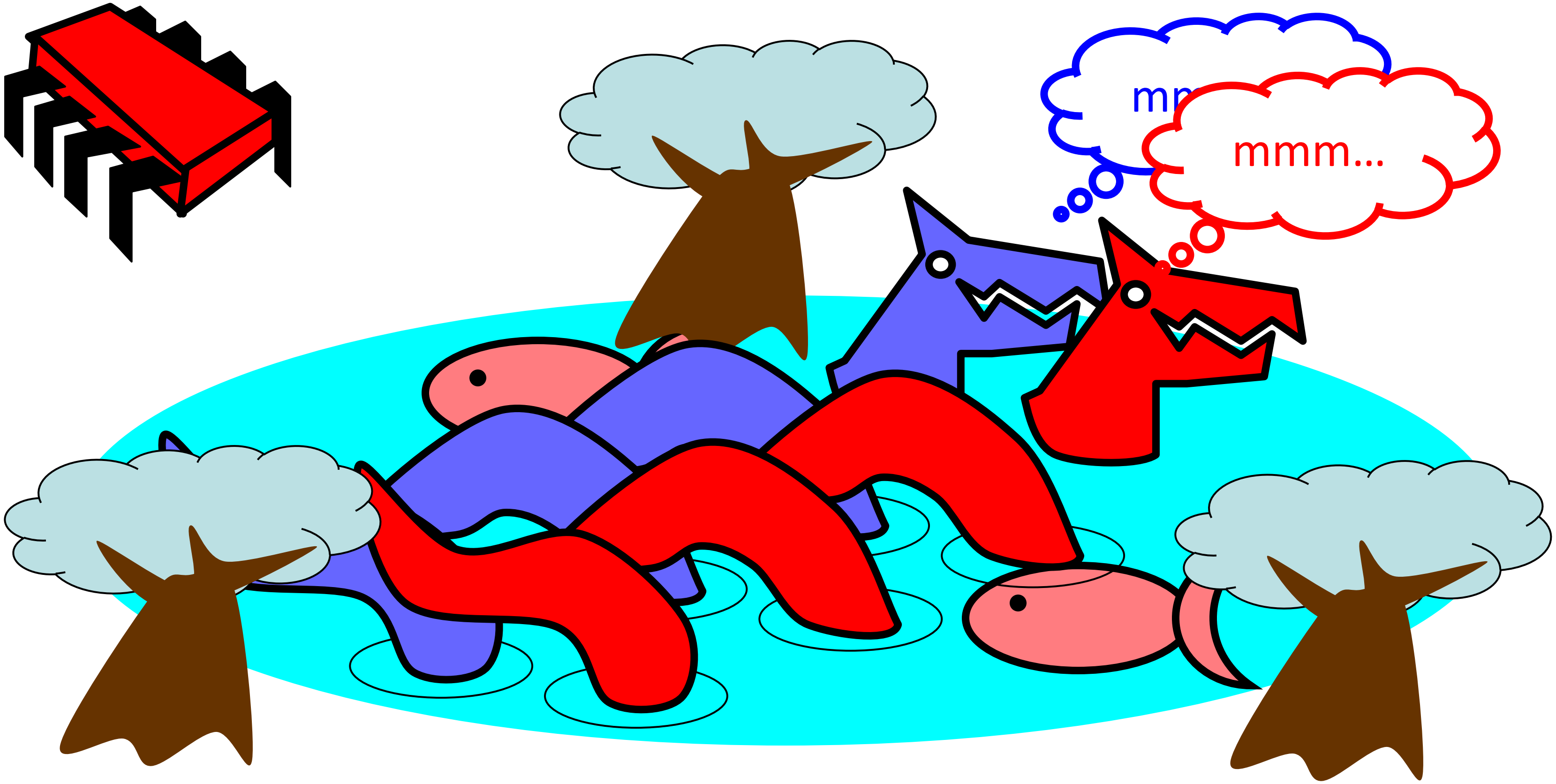
# The Fable Continues

- Alice and Bob fall in love & marry
- Then they fall out of love & divorce
  - She gets the pets
  - He has to feed them
- Leading to a new coordination problem: Producer-Consumer

# Bob Puts Food in the Pond



# Alice releases her pets to Feed



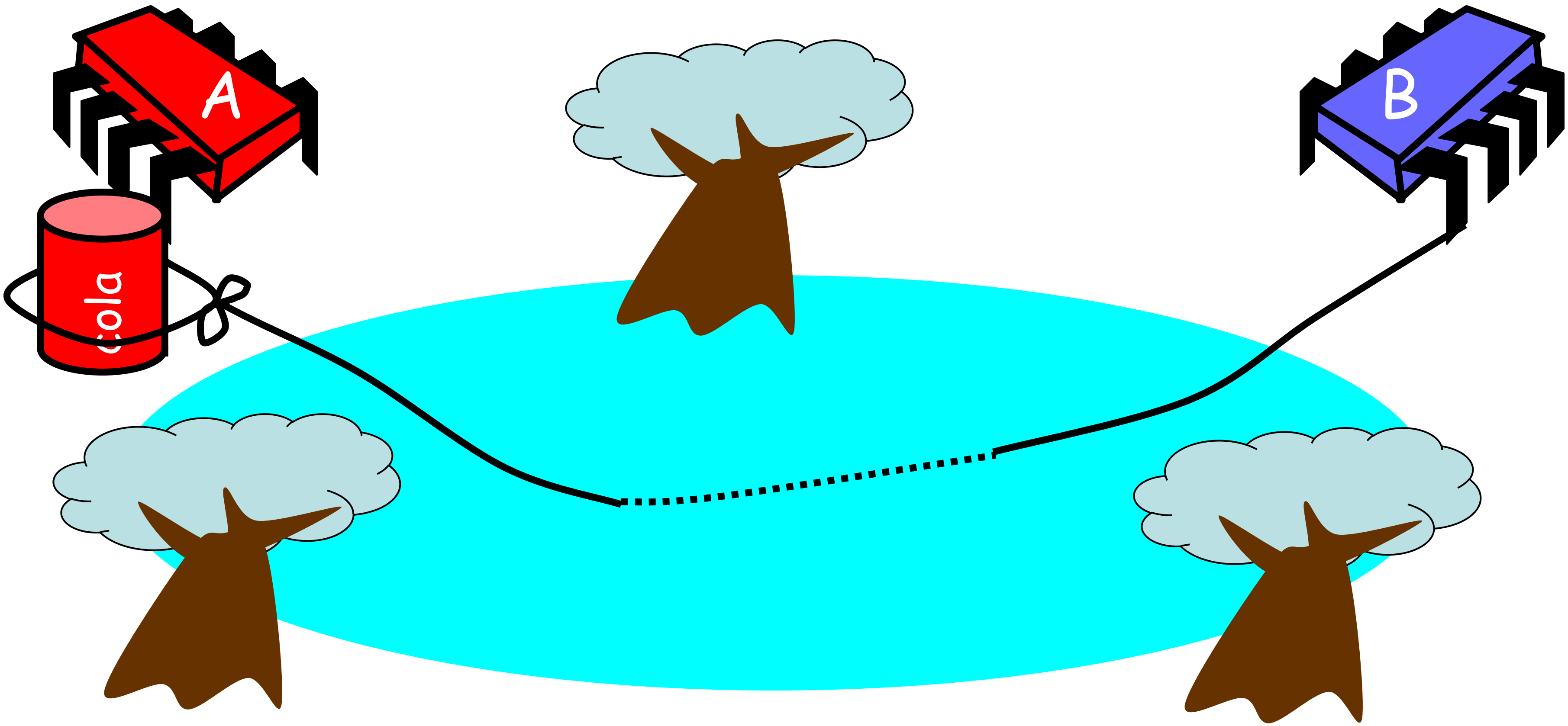
# Producer/Consumer

- 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

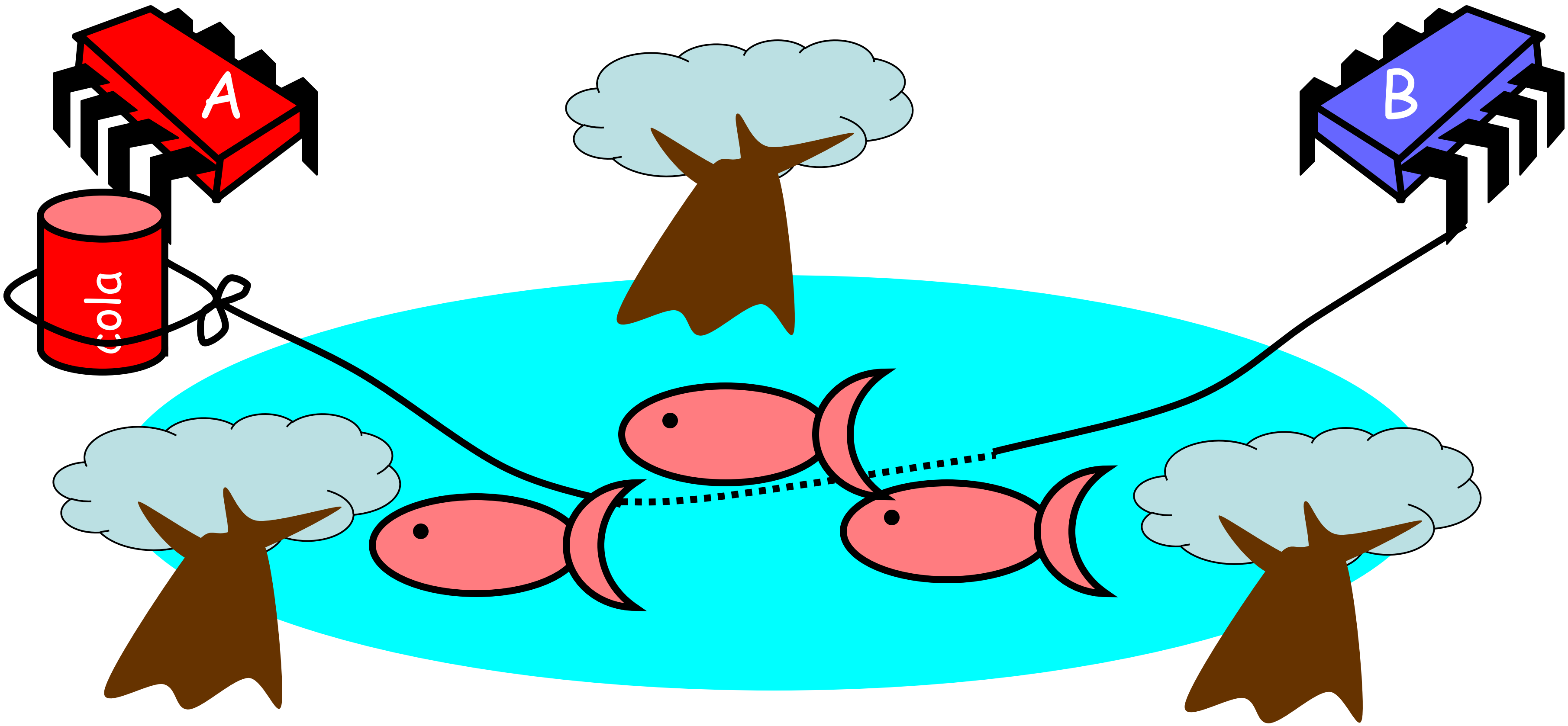
# Producer/Consumer

- 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

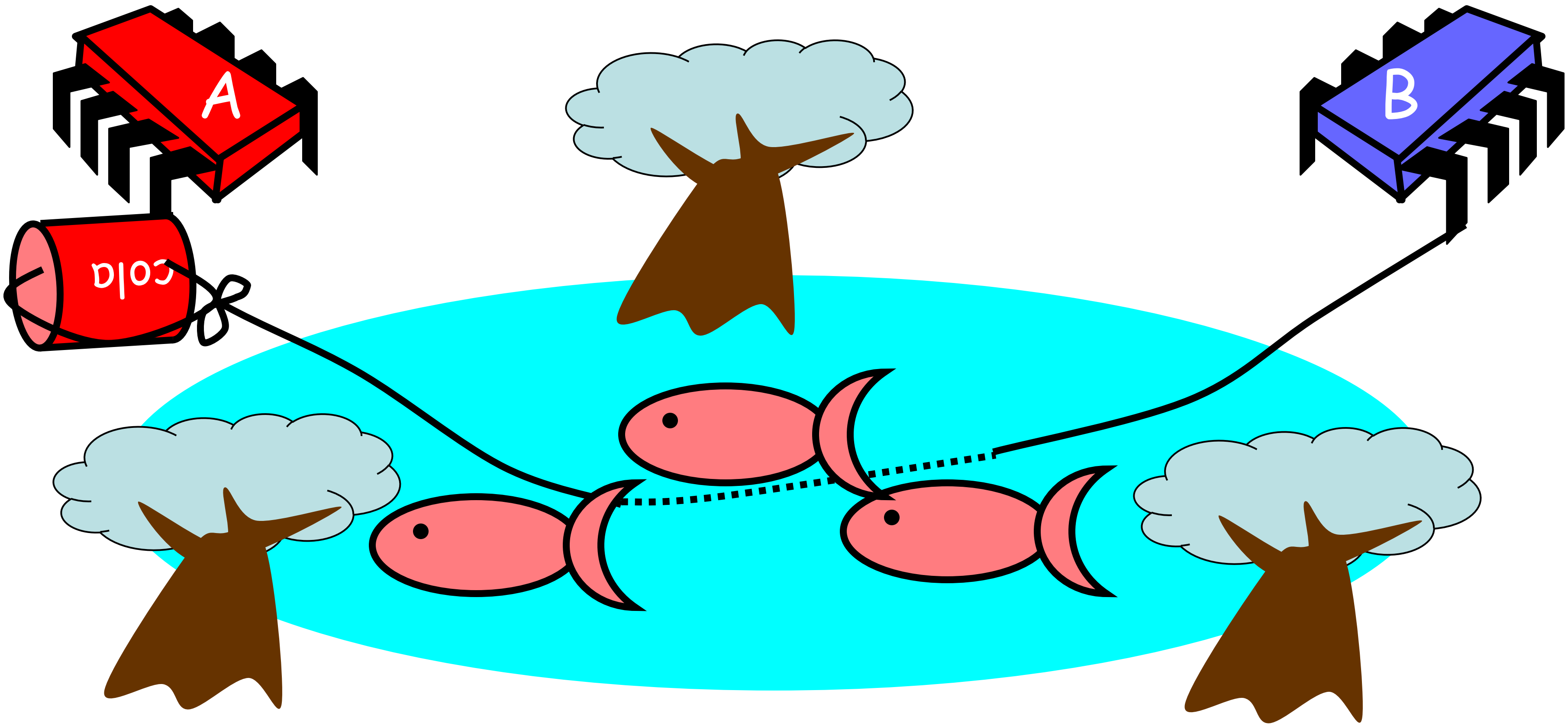
# Surprise Solution



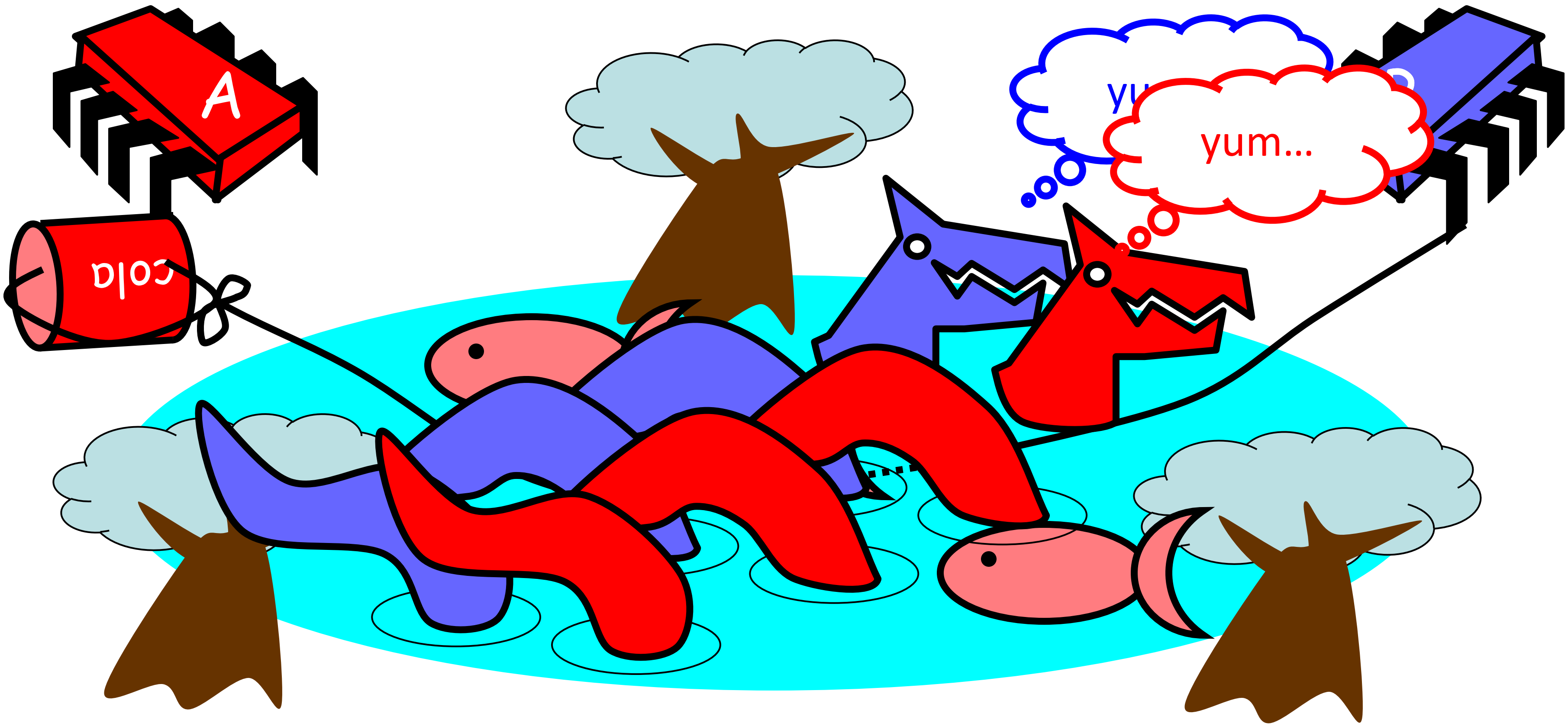
# Bob puts food in Pond



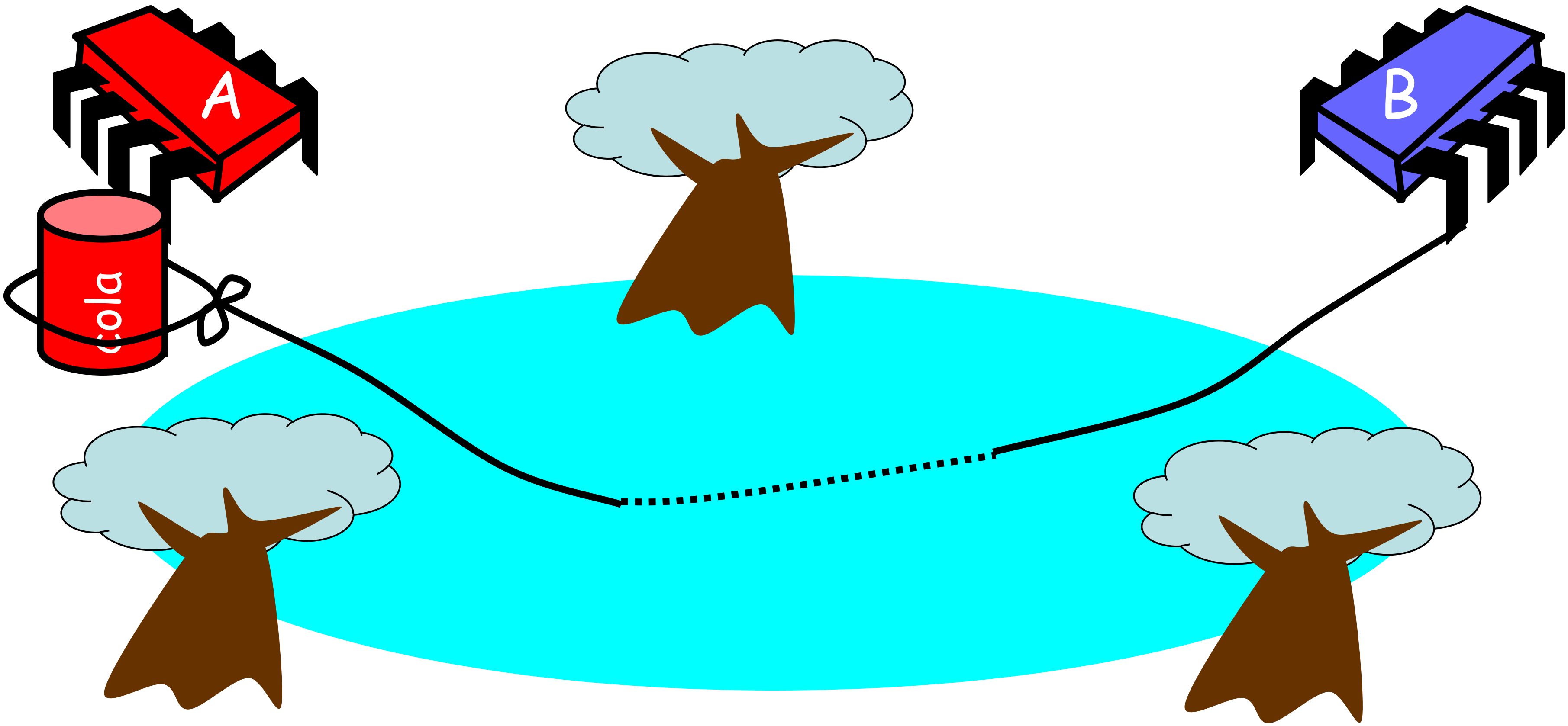
# Bob knocks over Can



# 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

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

Alice's code

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
  - Pets and Bob never together in pond

safety

- No Starvation

  if Bob always willing to feed, and pets always famished, then pets eat infinitely often.

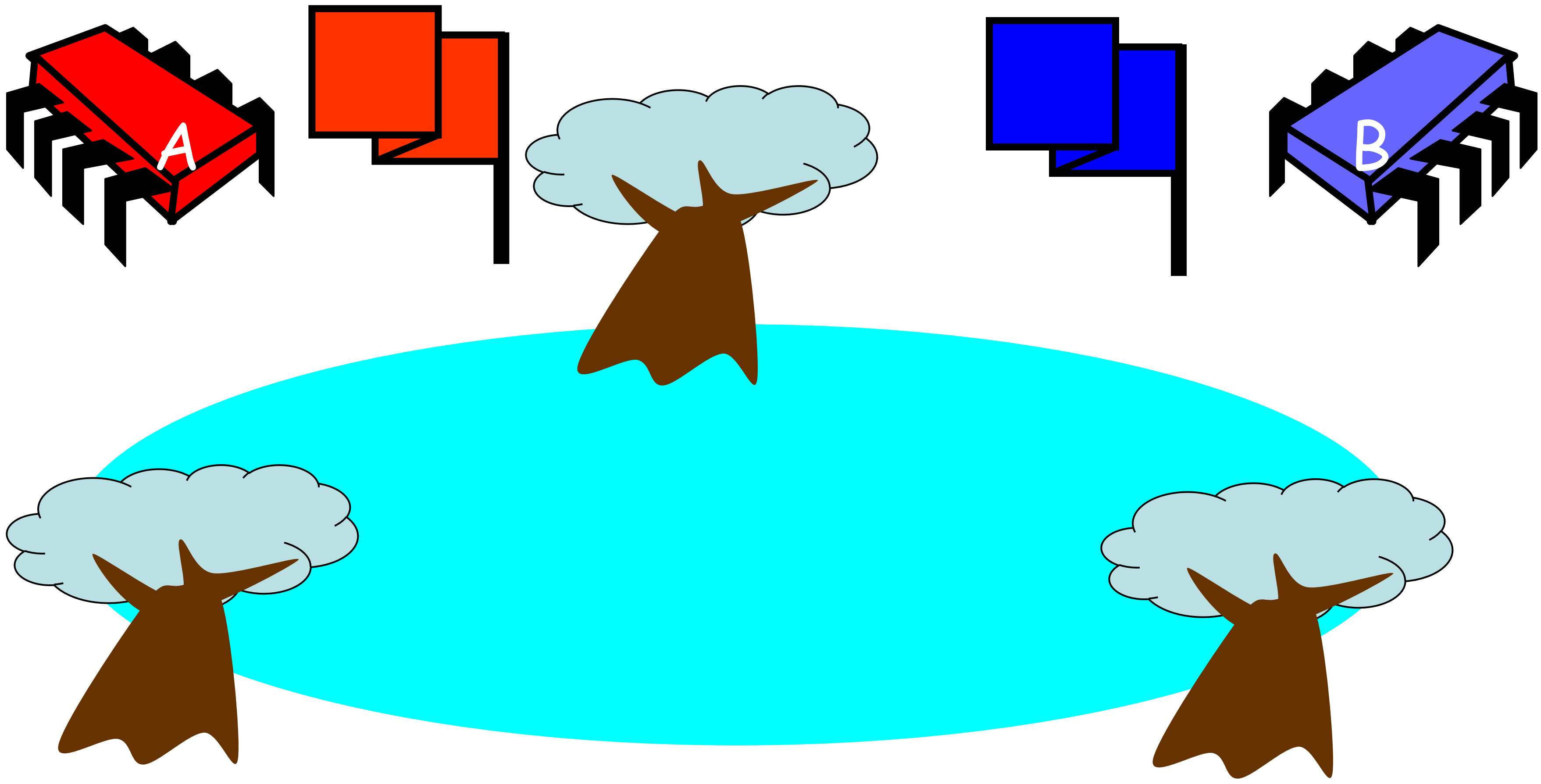
liveness

- Producer/Consumer

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

safety

# Could Also Solve Using Flags



# Waiting

- Both solutions use waiting
  - `while(mumble){}`
- In some cases 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

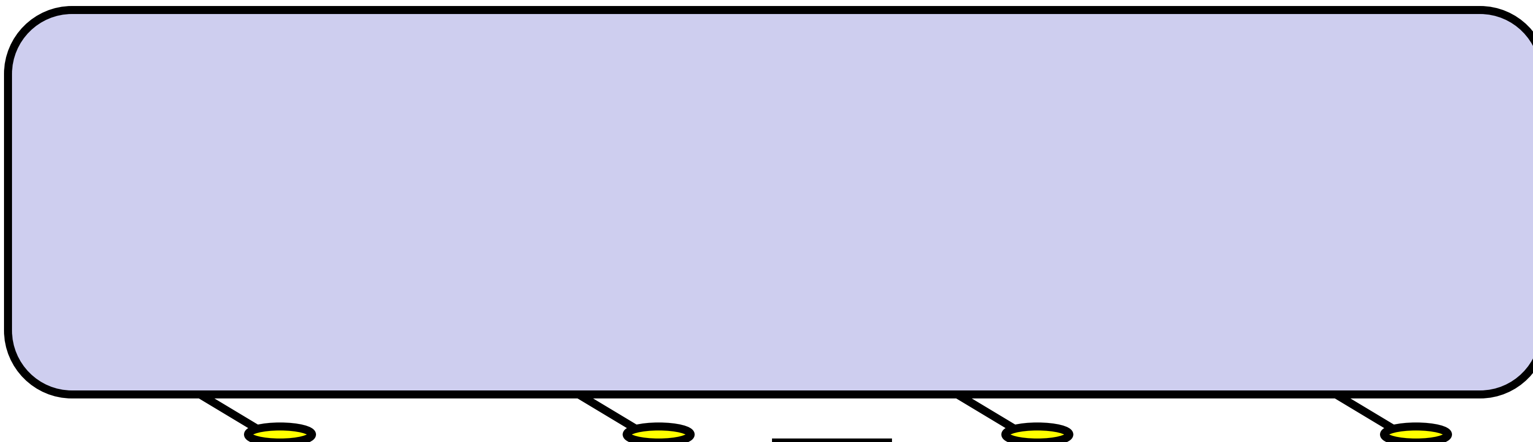
# The Fable drags on …

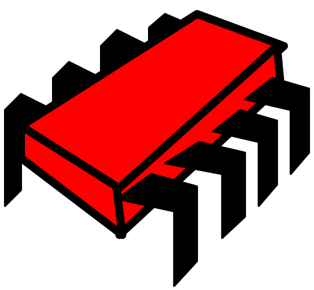
- Bob and Alice still have issues
- So they need to communicate

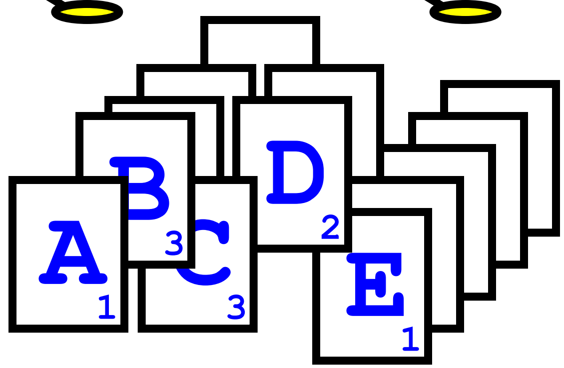
# The Fable drags on …

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

# Billboards are Large



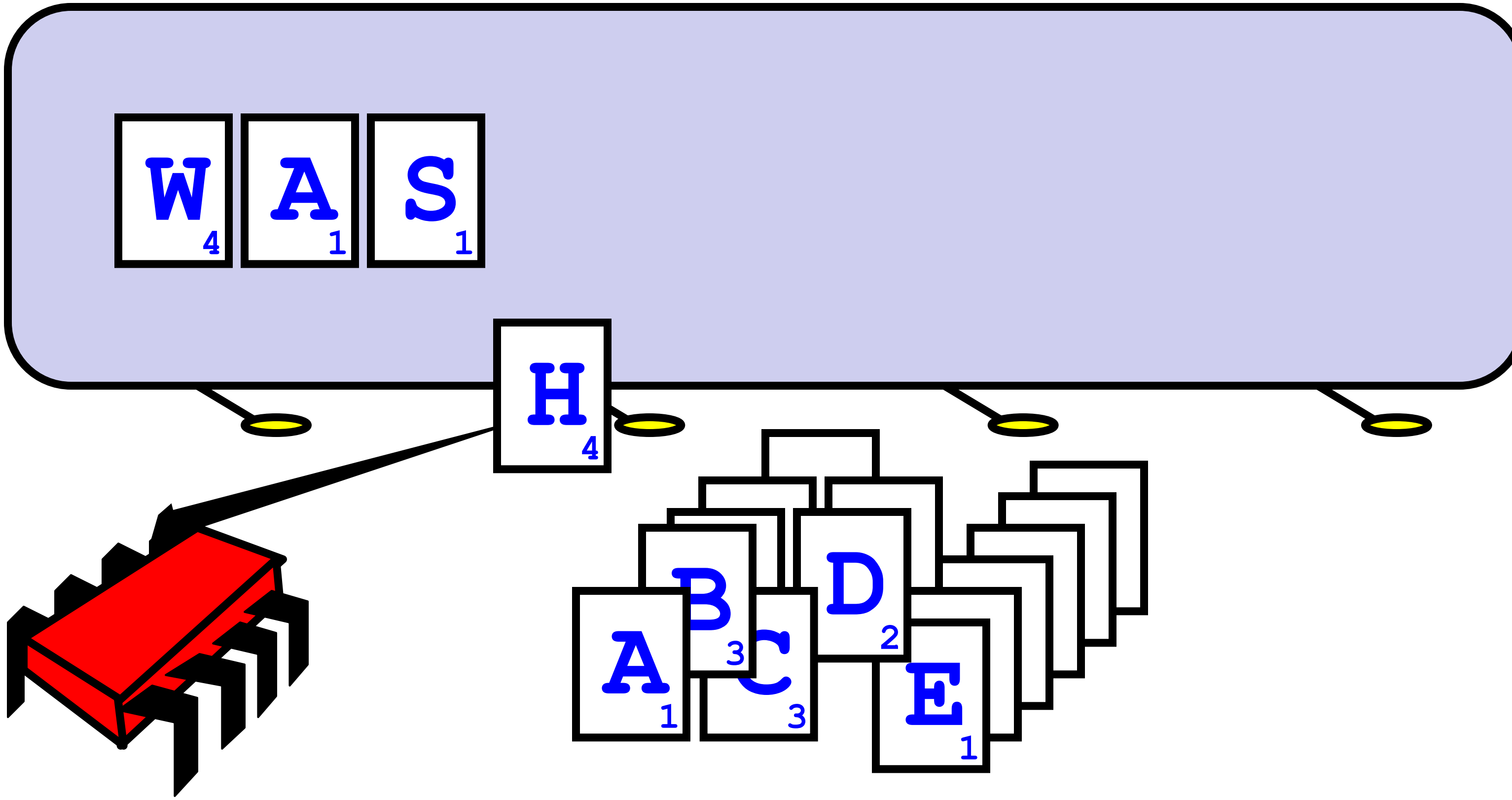




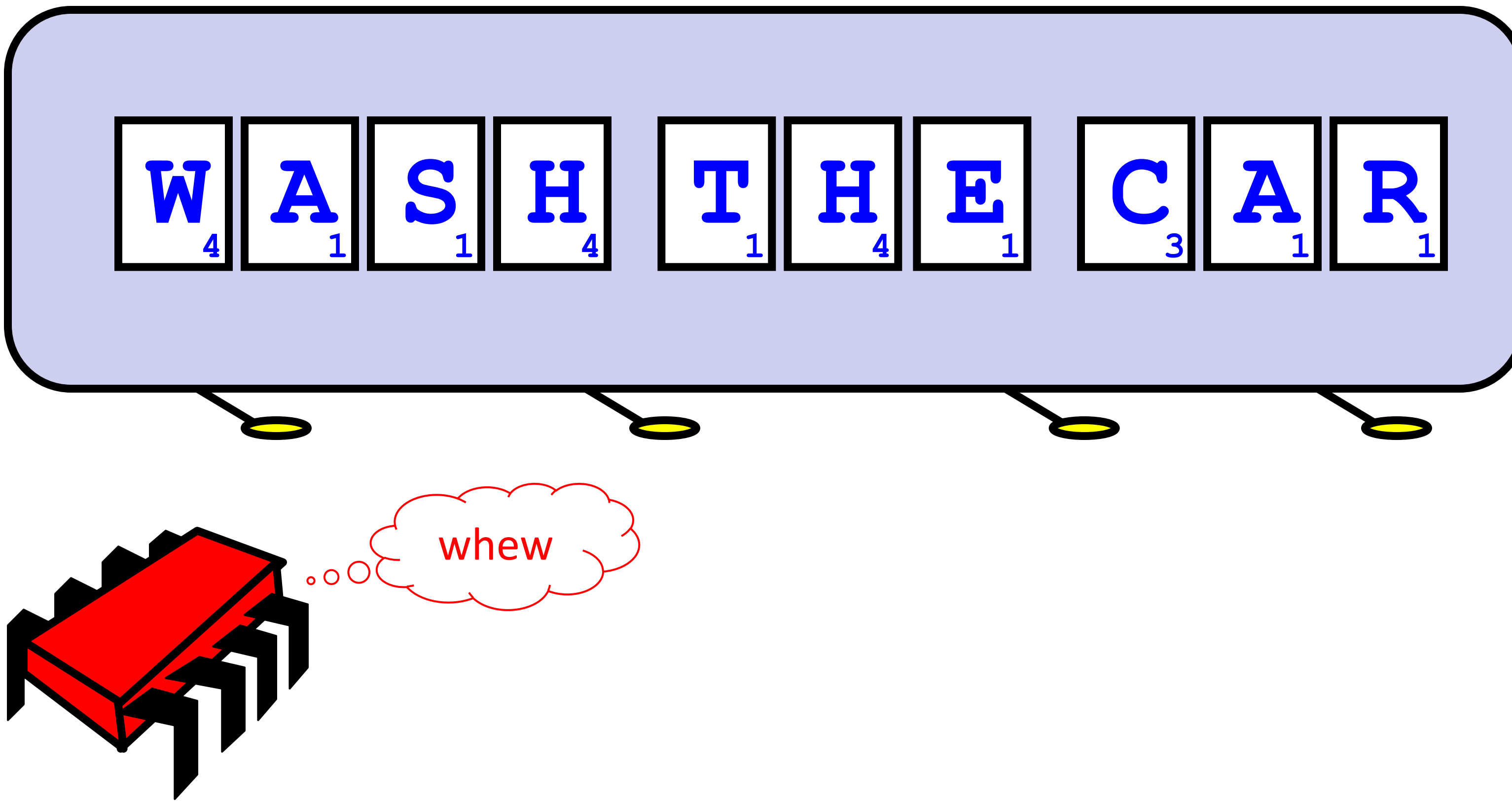

Letter Tiles

From Scrabble™ box

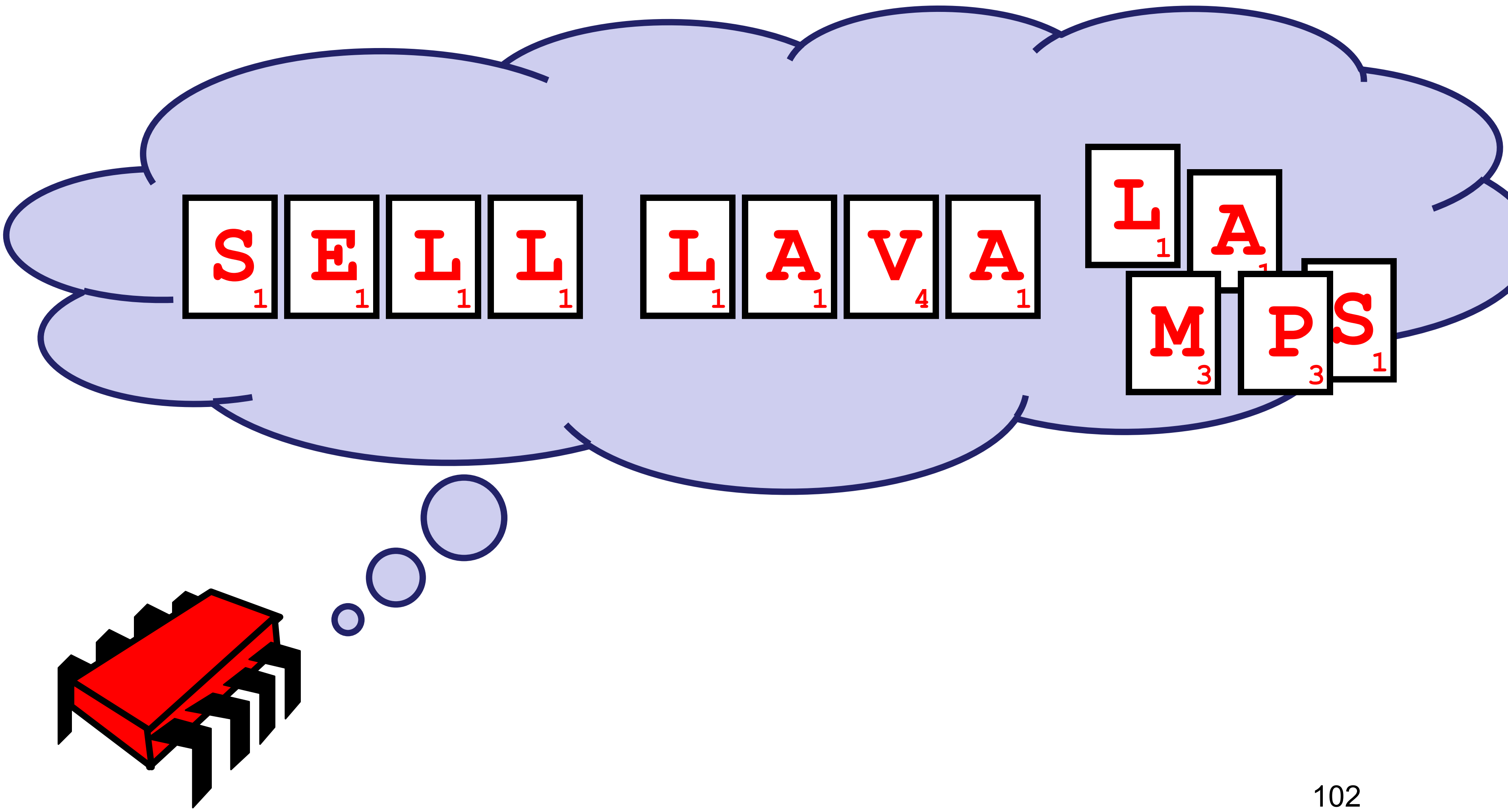
# Write One Letter at a Time …



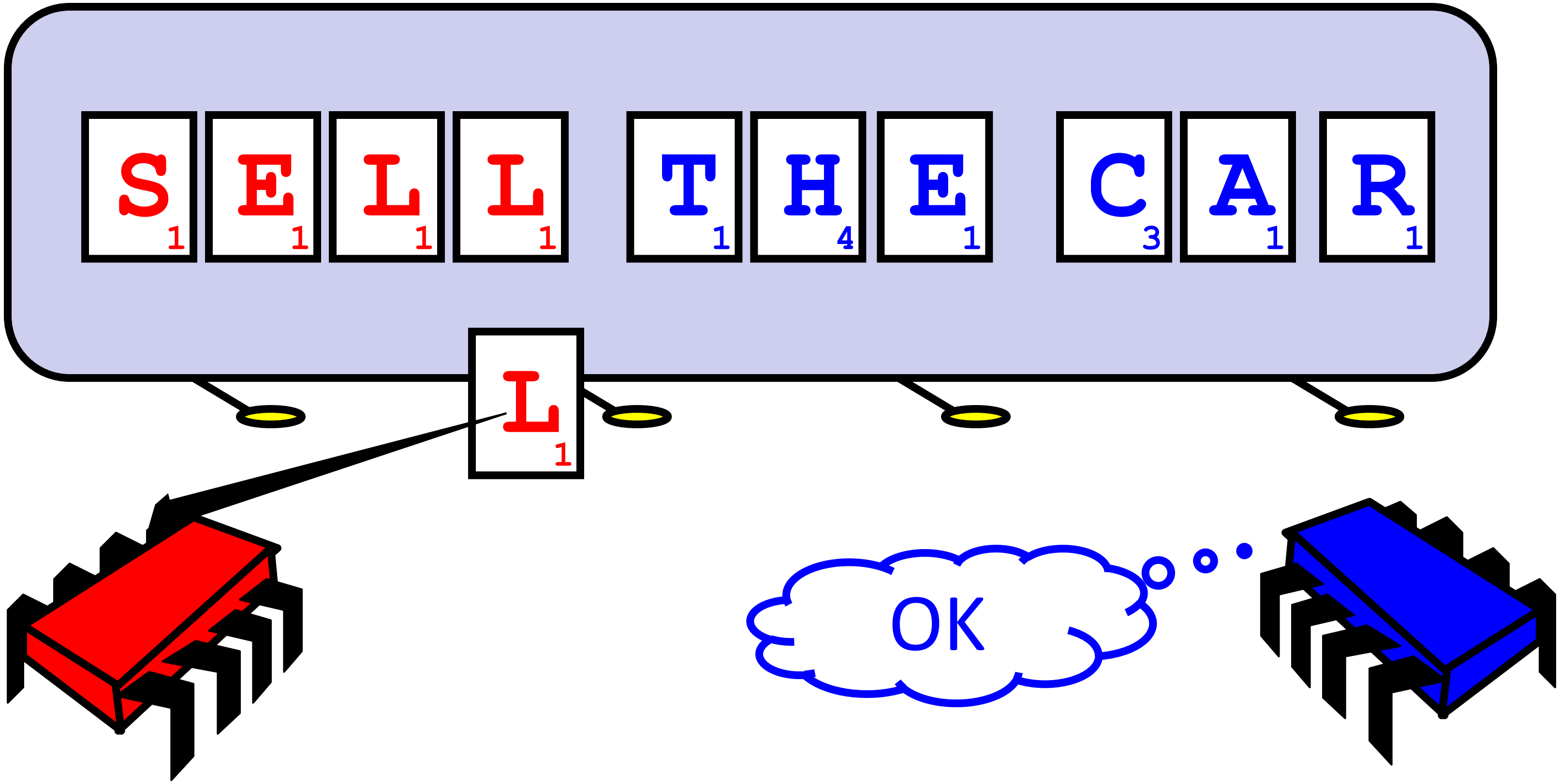
# To post a message



# 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 “snapshot”
    - Old message or new message
    - No mixed messages

# Readers/Writers (continued)

- Easy with mutual exclusion
- But mutual exclusion requires waiting
  - One waits for the other
  - Everyone executes sequentially
- Remarkably
  - We can solve R/W without mutual exclusion

# Esoteric?

- Java container **`size()`** method
- Single shared counter?
  - incremented with each **`add()`** and
  - decremented with each **`remove()`**
- Threads wait to exclusively access cou

**performance bottleneck**

# Readers/Writers Solution

- Each thread i has `size[i]` counter
  - only it increments or decrements.
- To get object's size, a thread reads a "snapshot" of all counters
- This eliminates the bottleneck

# Why do we care?

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

# Amdahl's Law

$$\text{Speedup} = \frac{1\text{-thread execution time}}{n\text{-thread execution time}}$$

# Amdahl's Law

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

# Amdahl's Law

Parallel fraction

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

# Amdahl's Law

Sequential fraction

Parallel fraction

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

# Amdahl's Law

Sequential fraction

Parallel fraction

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

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

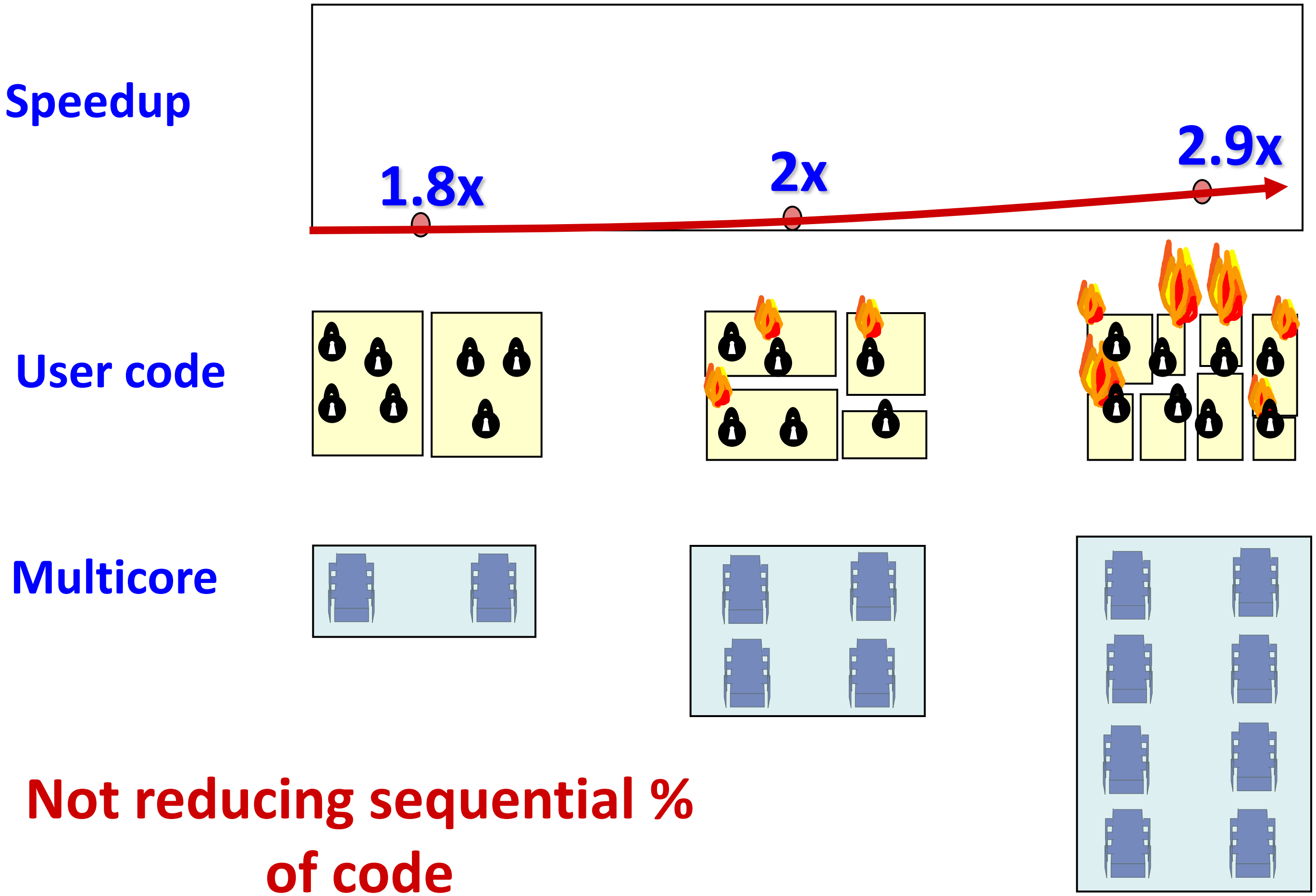
- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

# Example

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

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

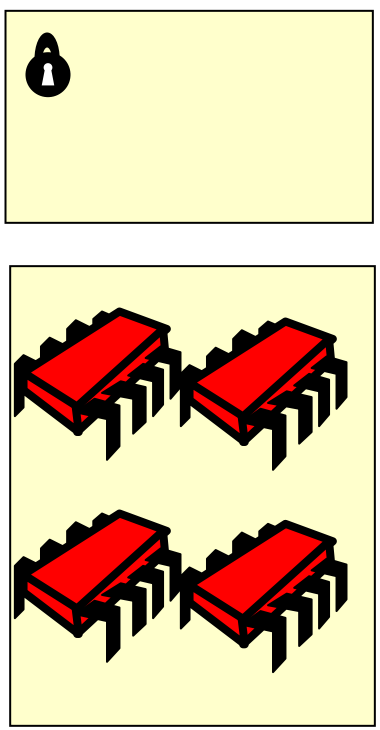
# Back to Real-World Multicore Scaling




**Not reducing sequential % of code**

# Shared Data Structures

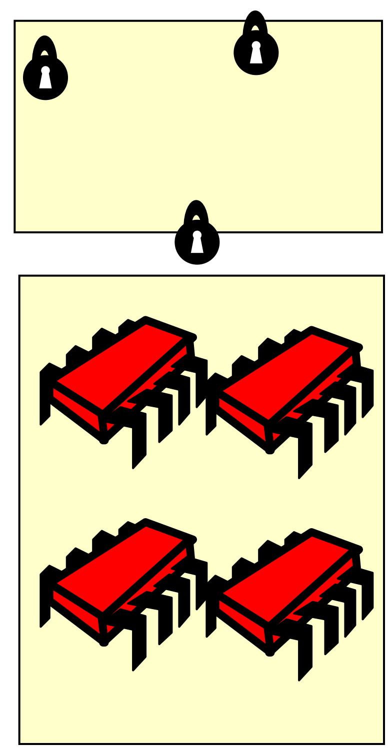
Coarse Grained



25% Shared

75% Unshared

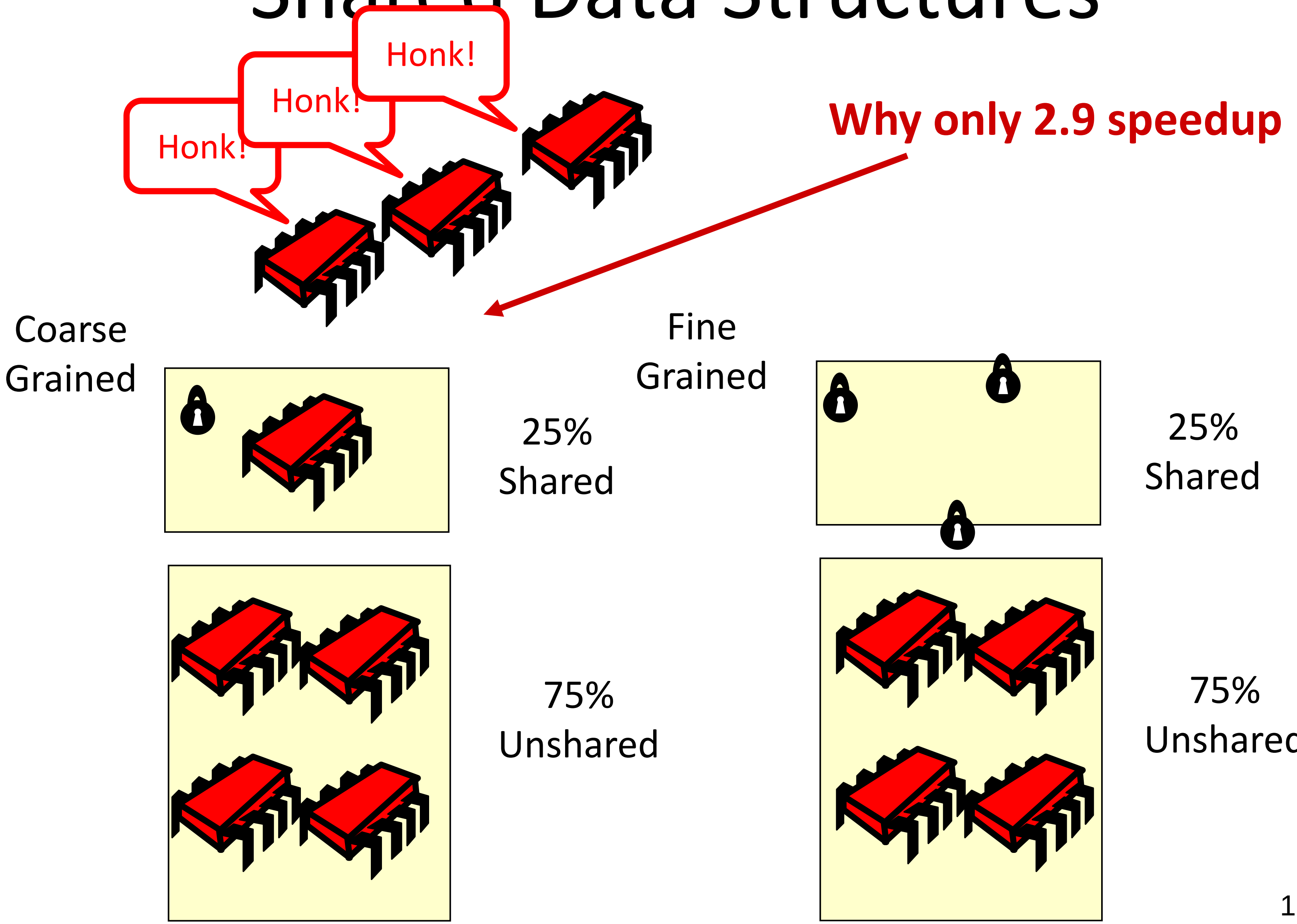
Fine Grained



25% Shared

75% Unshared

# Shared Data Structures



# Shared Data Structures