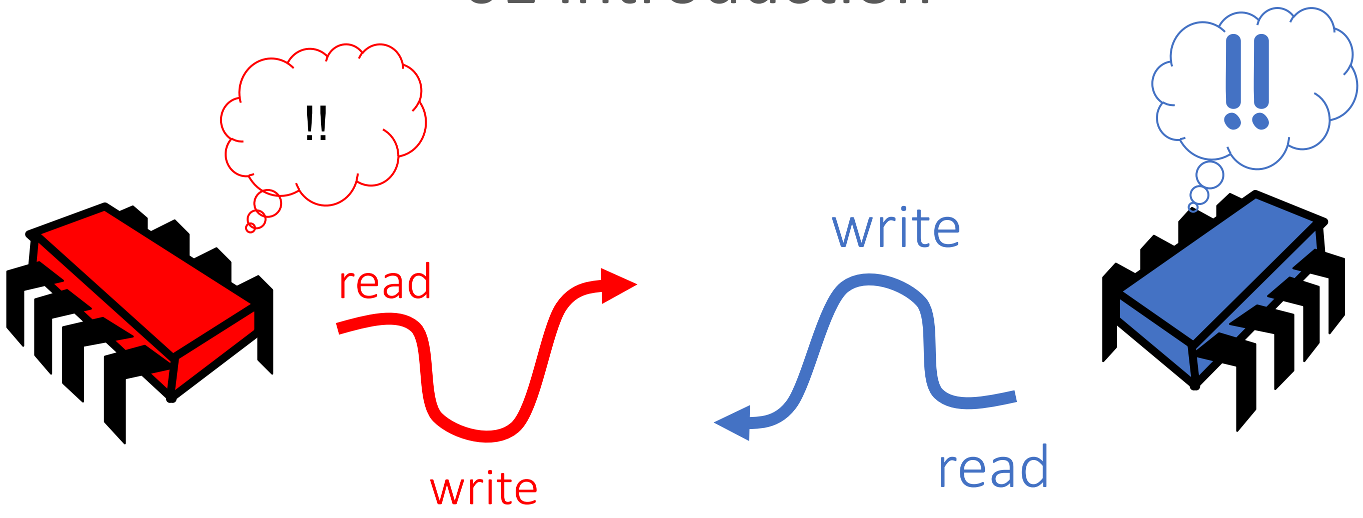
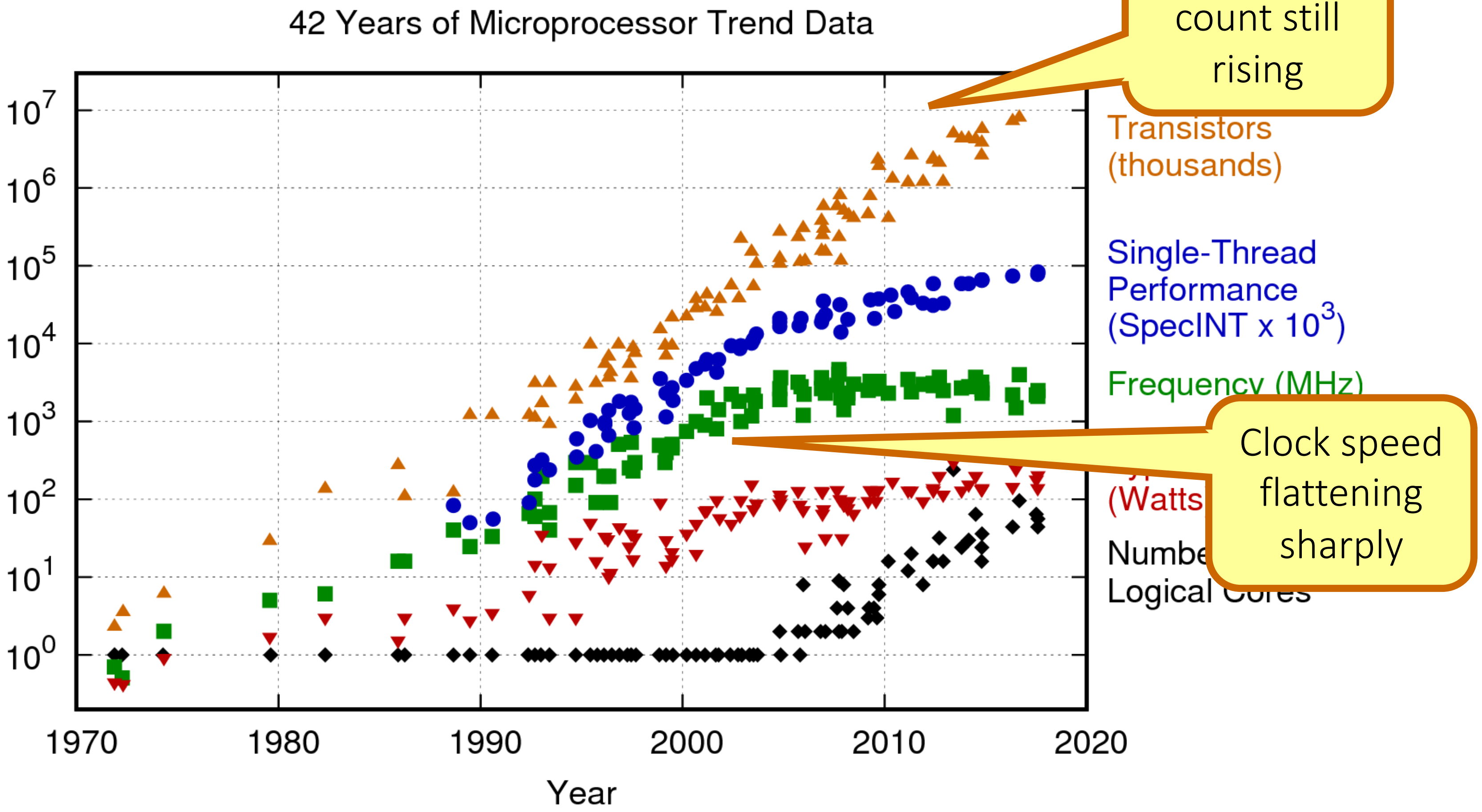
Replication and Consistency

01 Introduction



Dr. Annette Bieniusa
AG Software Technology + AG Programming Languages
TU Kaiserslautern

Moore’s Law

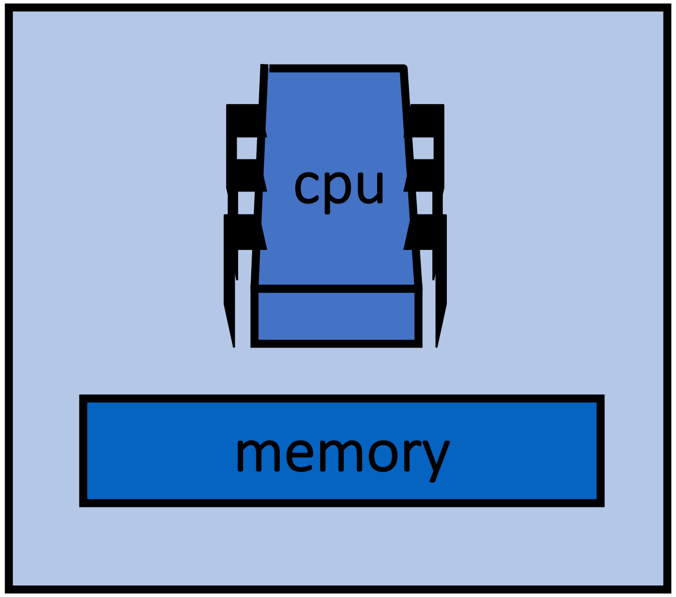


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

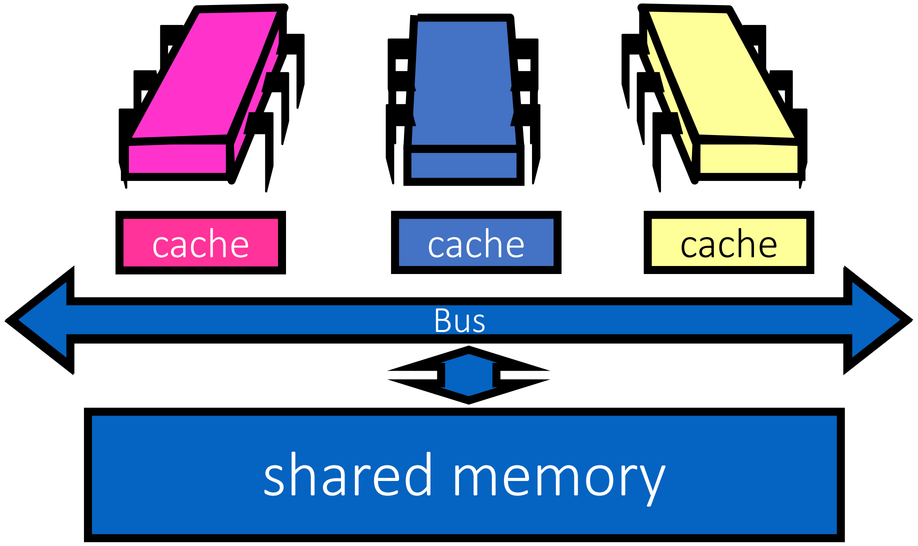
Source: https://github.com/karlrupp/microprocessor-trend-data

Moore’s Law (in practice)

Nearly Extinct: the Uniprocesor

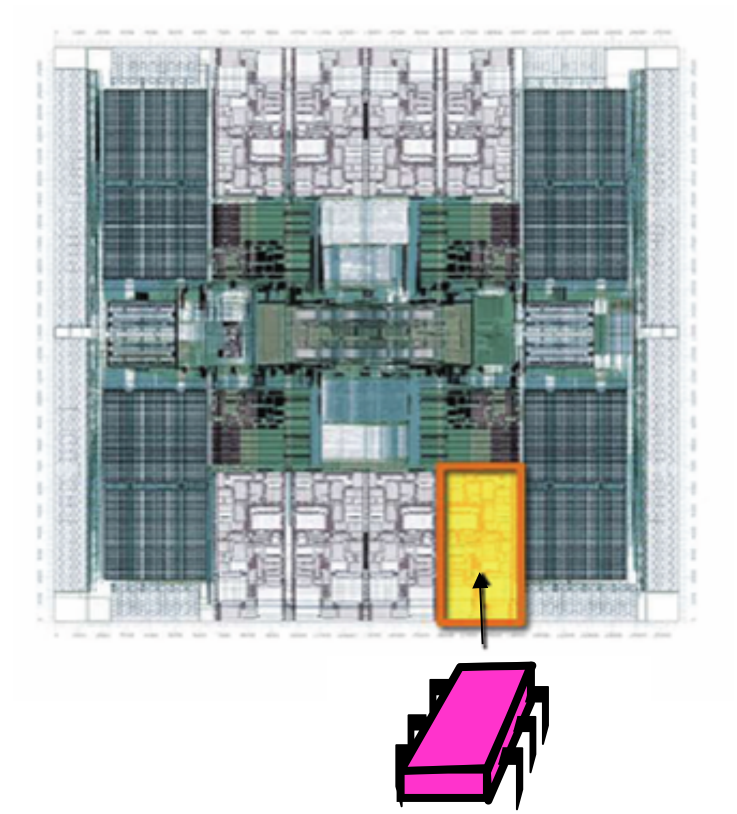


Endangered: The Shared Memory Multiprocessor (SMP)



The New Boss: The Multicore Processor (CMP)

All on the same chip

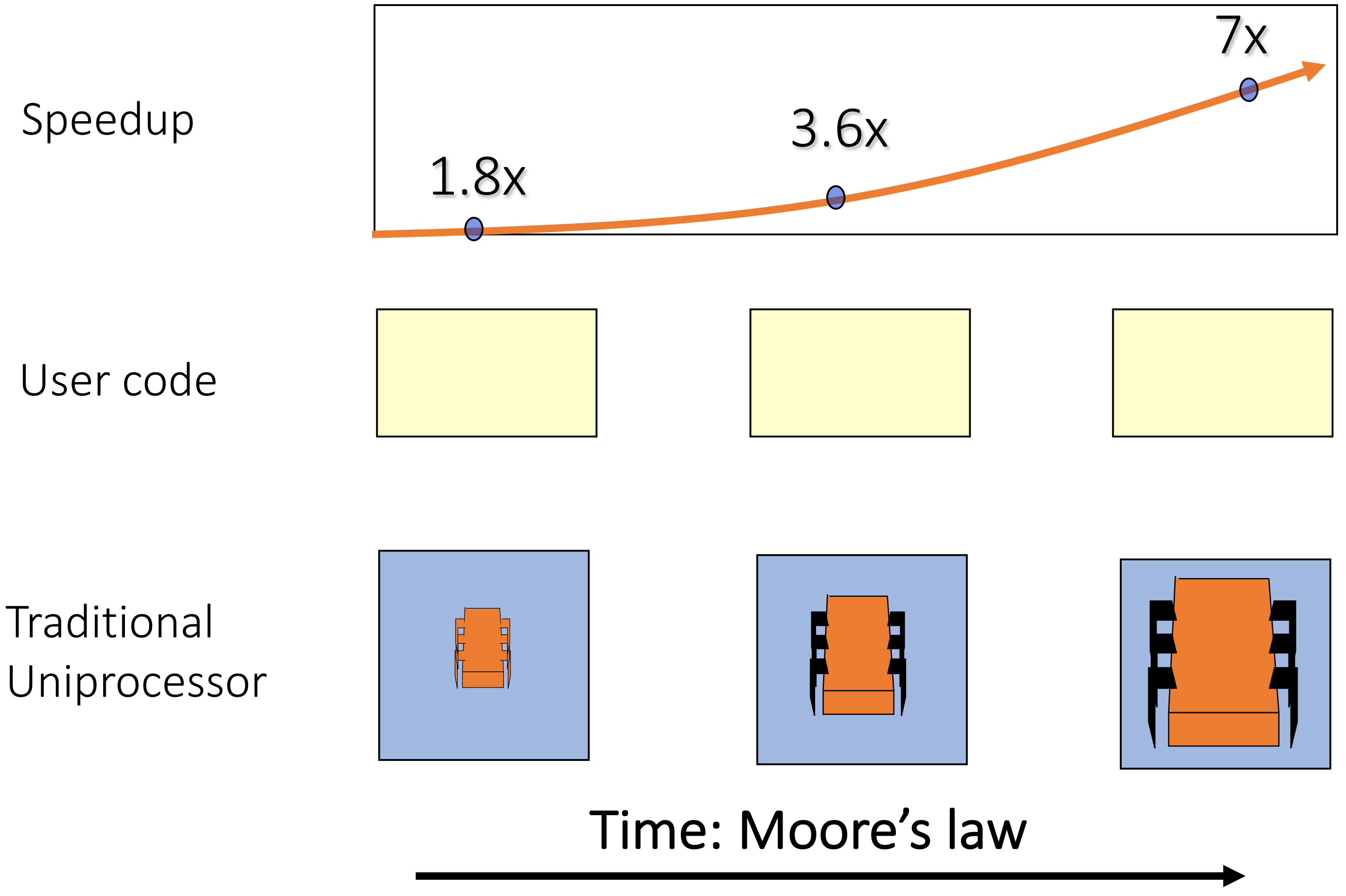


Sun T2000 Niagara

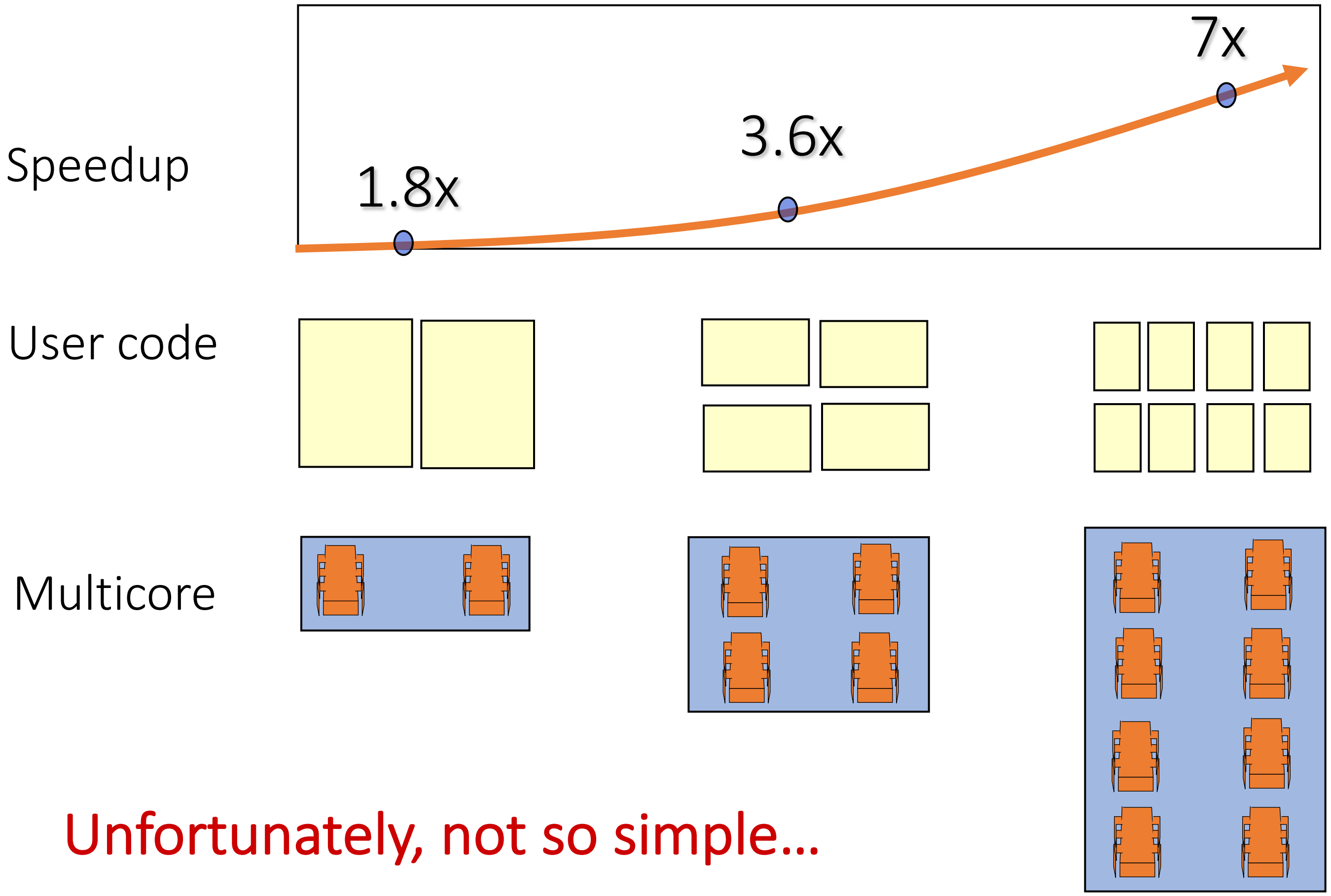
Why do we care?

- Time no longer cures software bloat
    - The “free-ride” is over
- When you double your program’s path length
    - You can’t just wait 6 months
    - Your software must somehow exploit twice as much concurrency

Traditional Scaling Process

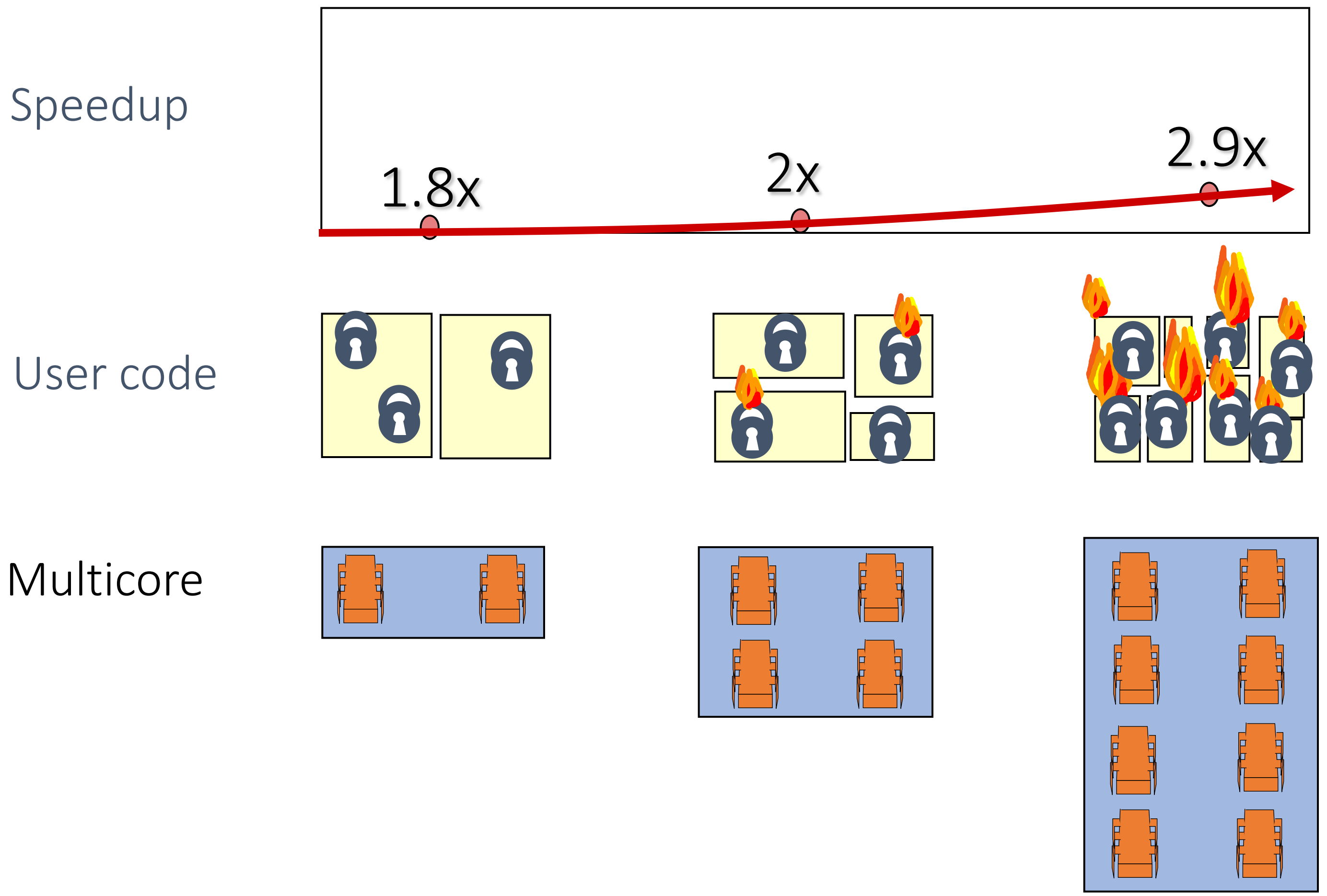


Ideal Scaling Process



Unfortunately, not so simple…

Actual Scaling Process



Amdahl's Law

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

$$= \frac{1}{\underbrace{\boxed{1-p}}_{\text{Sequential fraction}} + \frac{\overbrace{\boxed{p}}^{\text{Parallel fraction}}}{n}}$$

Example 1

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

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

Example 2

- 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 3

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

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

Example 4

- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?

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

Amdahl’s Law (in practice)

Parallelization and Synchronization require great care…

And this is the topic of our course!

Course overview

- Fundamentals
  - Synchronization primitives for Mutual Exclusion and Cooperation
  - Models for Shared Memory: TSO + PSO
    (Guest lecturer: Victor Vafeiadis, MPI-SWS)
  - Correctness notions for Concurrent Objects
  - Impossibility results
- Real-World programming
  - Spin locking
  - Concurrent data structures: Lists, Queues, Stacks

Course objectives

You will be able to

- understand and explain the underlying mechanisms of classical concurrent and replicated data structures,
- explain and elaborate the limitations of non-blocking synchronization mechanisms, and
- formally describe and compare standard memory and consistency models for concurrent systems.

Organization

- Lecturer: Annette Bieniusa
- Tutor: 🥺

Lectures: Wed 10:00 – 11:30

Check homepage for new room!

(Bi-weekly) Exercise session
Please participate in vote here:

https://terminplaner4.dfn.de/AaeSjkdJzXuPDikl

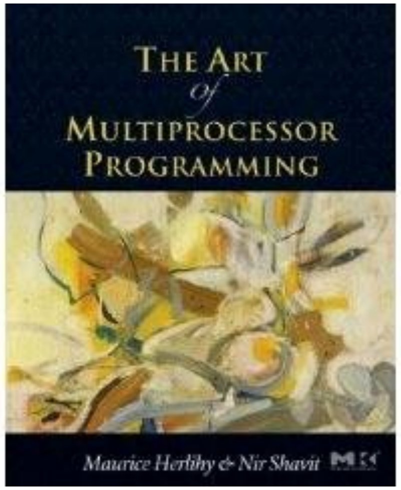
(deadline Friday, Nov 08, 12:00)

Exam and admission

- Oral exam
- Please register with our secretary Judith Stengel for a time slot (and ofc with the examination office)!

- Admission to exam:
  One mandatory mini-project in January

Reading material

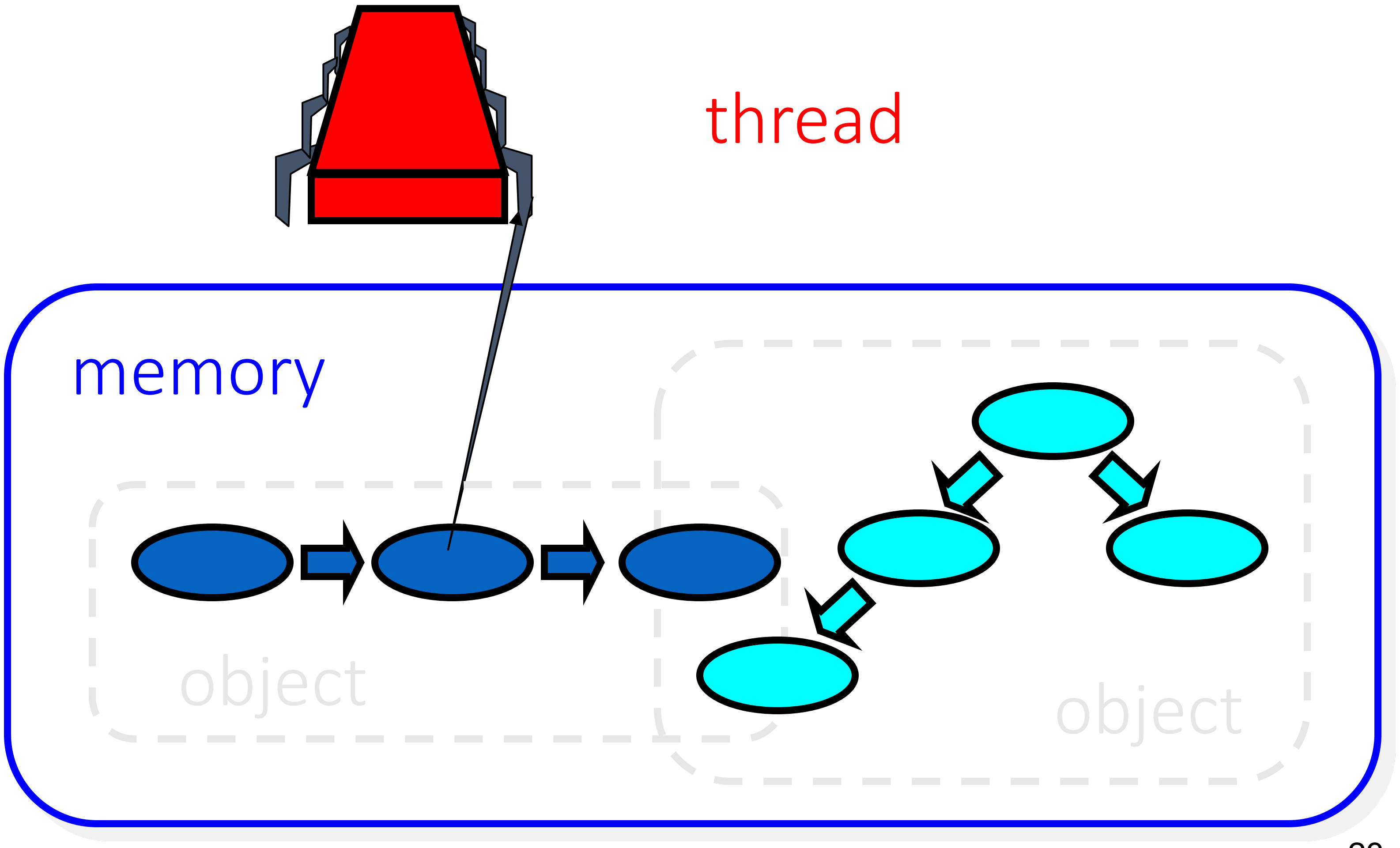

THE ART of MULTIPROCESSOR PROGRAMMING
Maurice Herlihy & Nir Shavit

The Art of Multiprocessor Programming

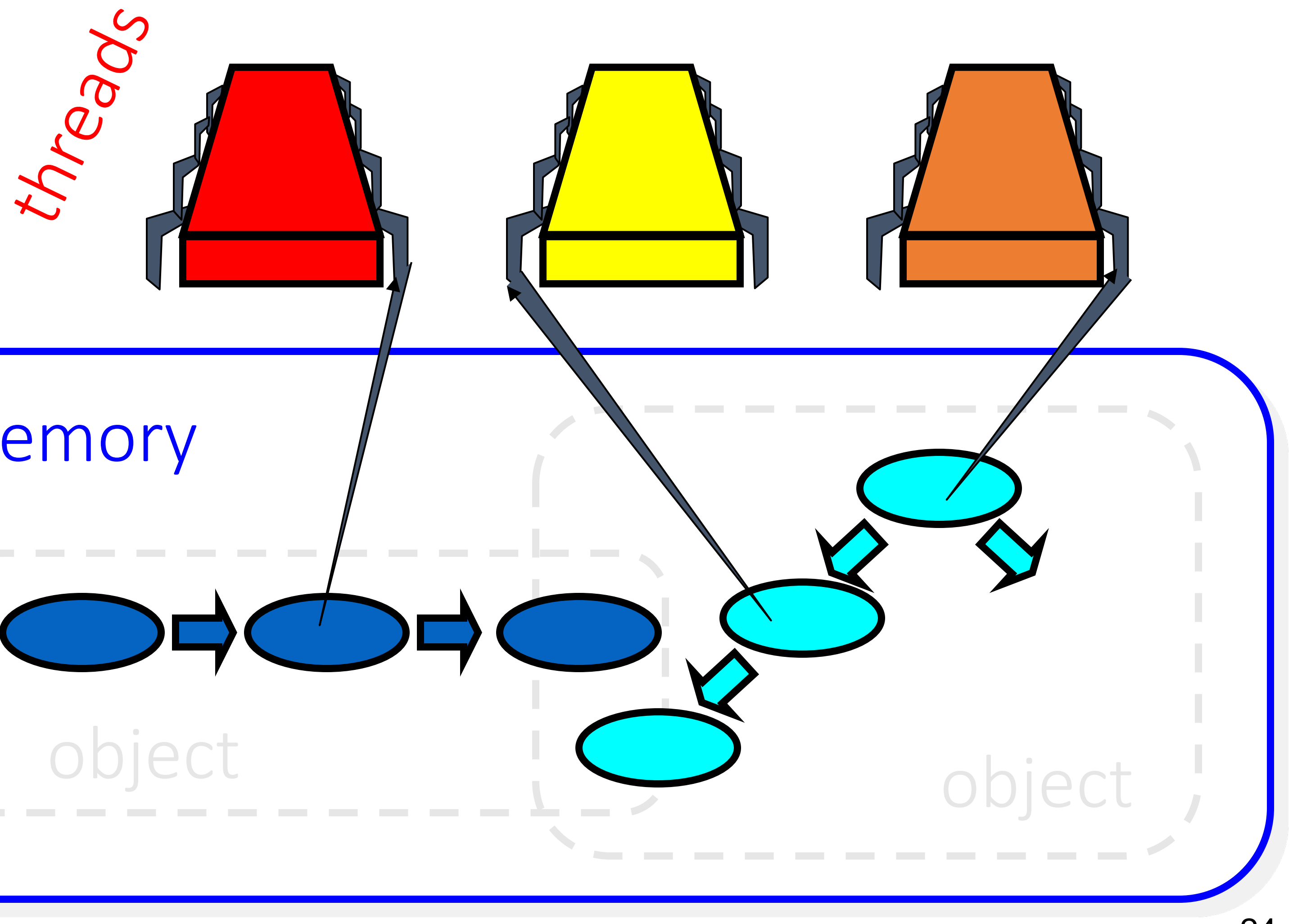
by Maurice Herlihy & Nir Shavit

(Morgan Kaufman, 2012, Revised first edition)

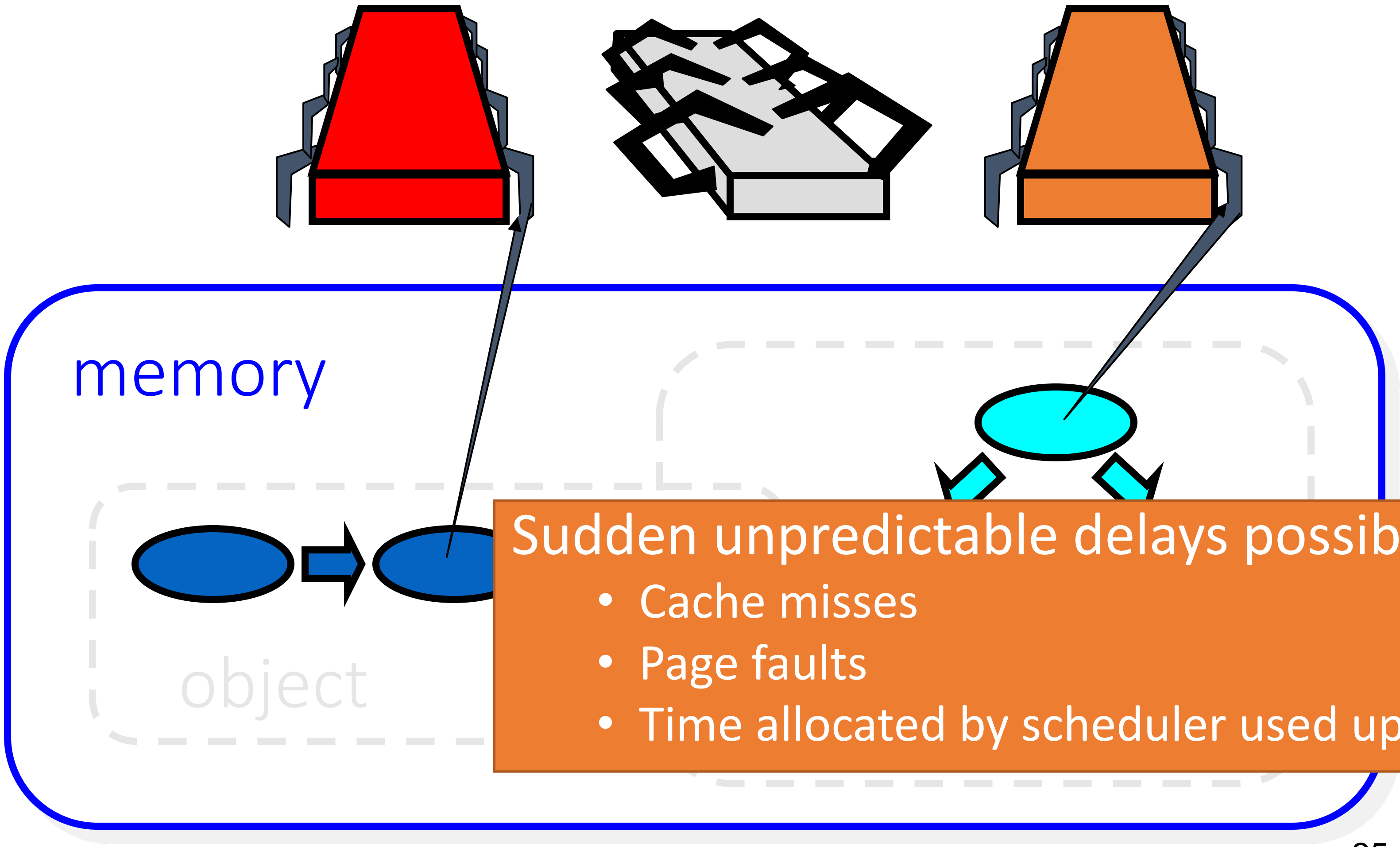
Sequential Computation



Concurrent Computation



Asynchrony



Sudden unpredictable delays possible

- Cache misses
- Page faults
- Time allocated by scheduler used up

Model Summary

- N *threads*
  - Sometimes called *processes*
  - Uniquely identified by id number (for simplicity: 1, …, n)
- Single shared *memory*
  - More complex memory models later!
- *Objects* reside in memory
  - Communication via reading and writing of memory locations (sometimes called shared registers)
- Unpredictable asynchronous delays
  - I.e. no assumptions about relative speed of processors

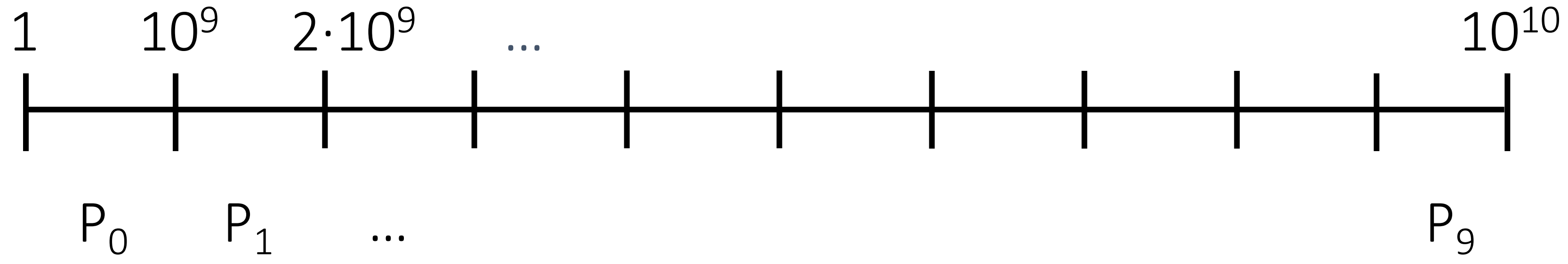
Road Map

- 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”

Task: Parallel Primality Testing

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

Idea: Load Balancing



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

Procedure for Thread *i*

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

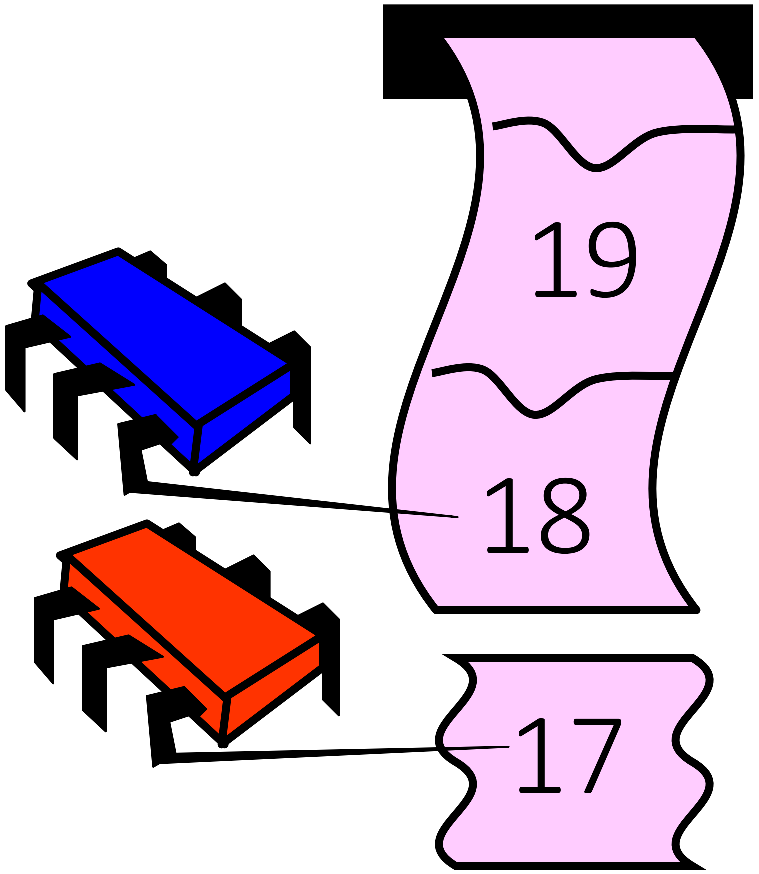
Issues

- Higher ranges have fewer primes
- Yet, larger numbers harder to test
- As a consequence, thread workloads are
  - uneven
  - hard to predict

rejected

We want *dynamic* load balancing!

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*

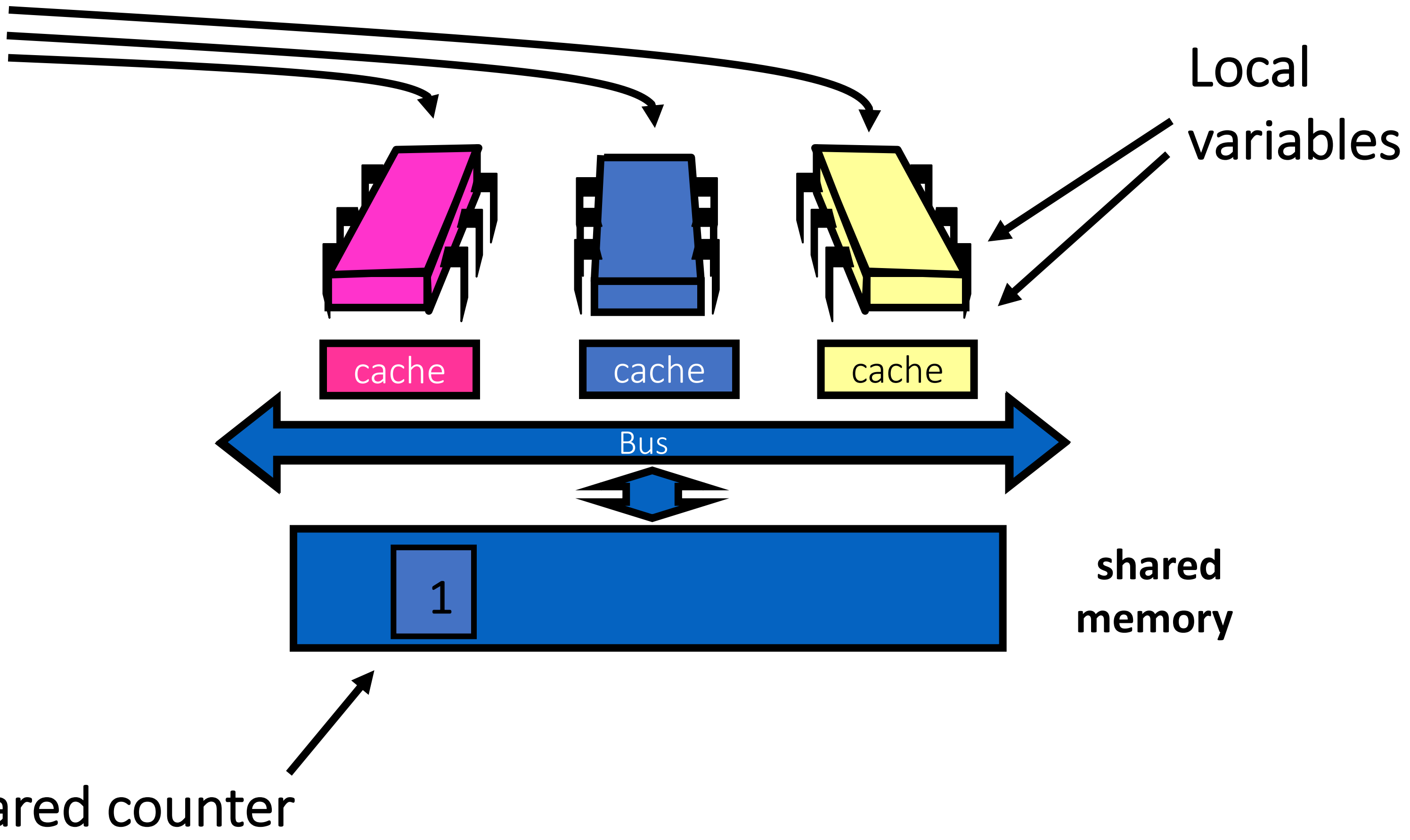
```
Counter counter = new Counter(1);

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++;
  }
}
```

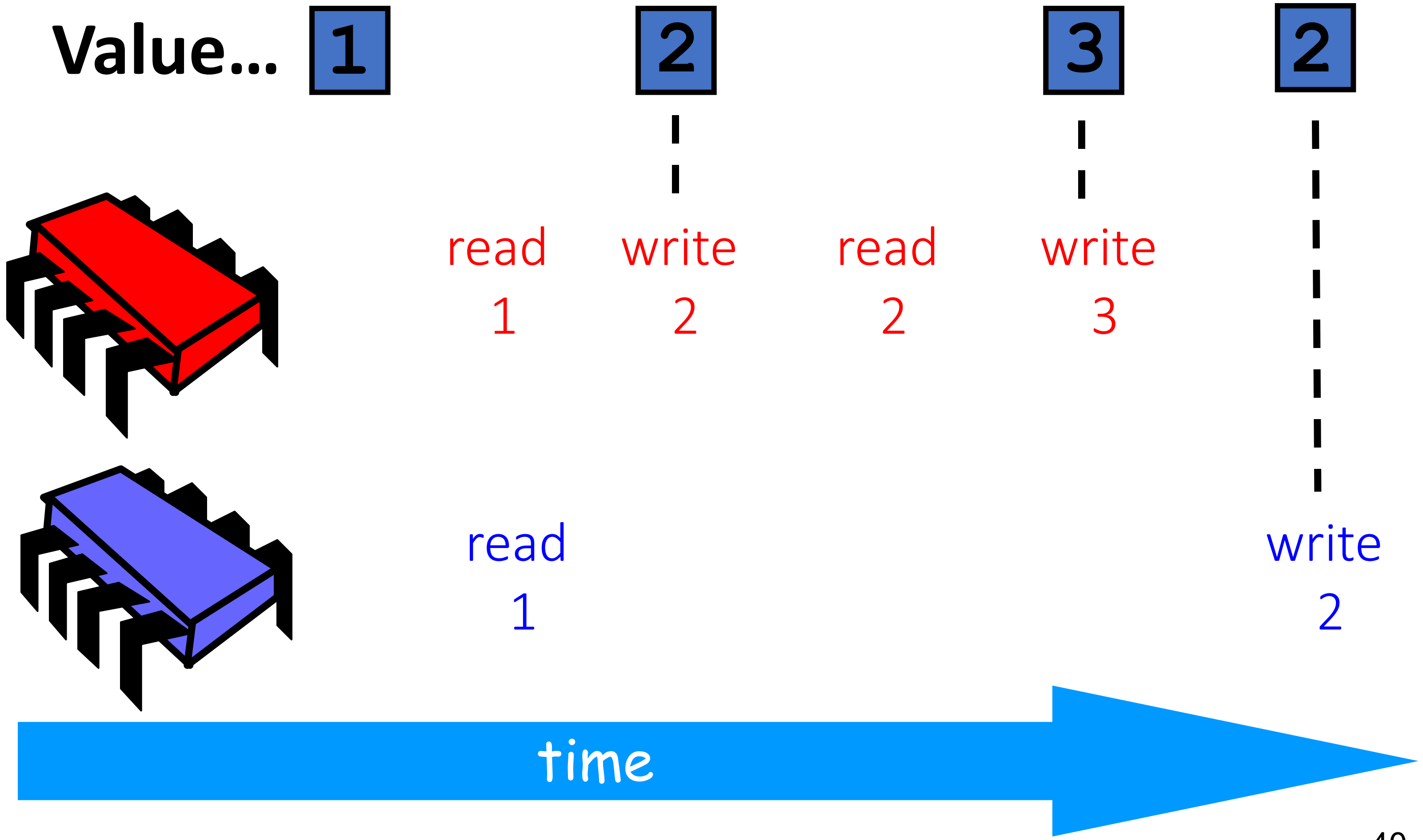
OK for single thread,
not for concurrent threads

What It Means

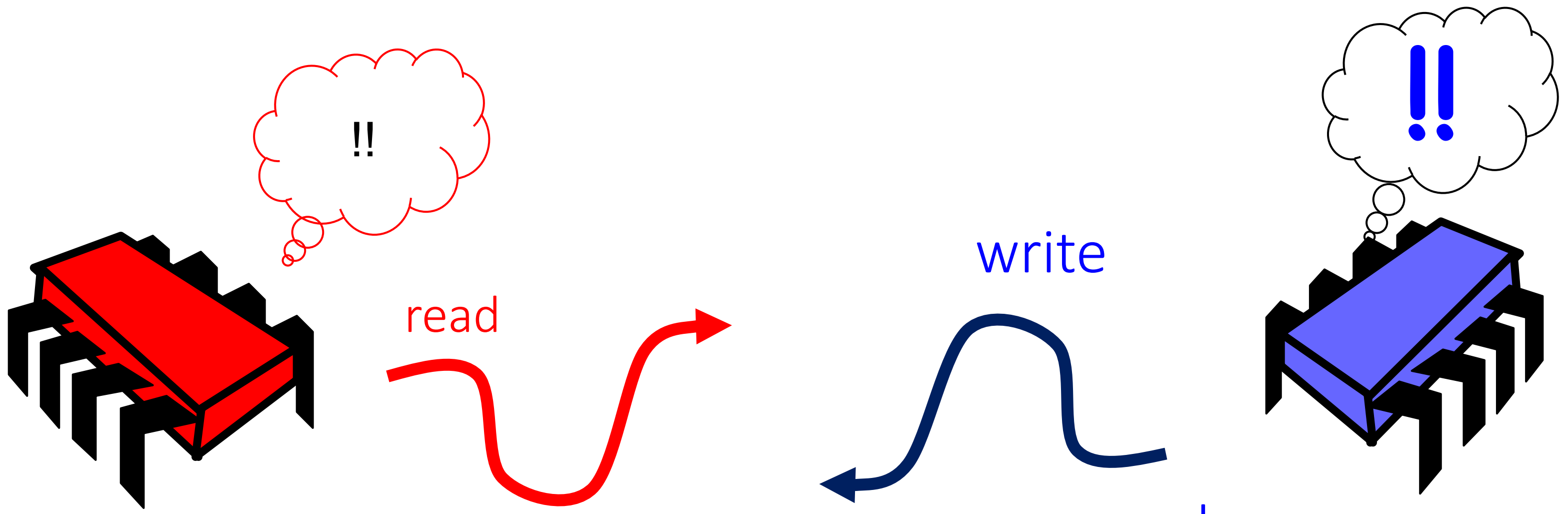
```
public class Counter {
  private long value;

  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…

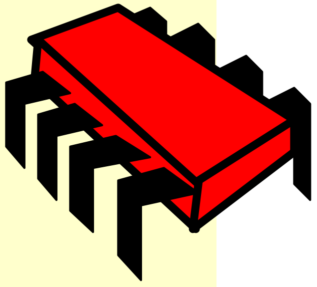
Challenge

```
public class Counter {
  private long value;

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

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;
  }
}
```

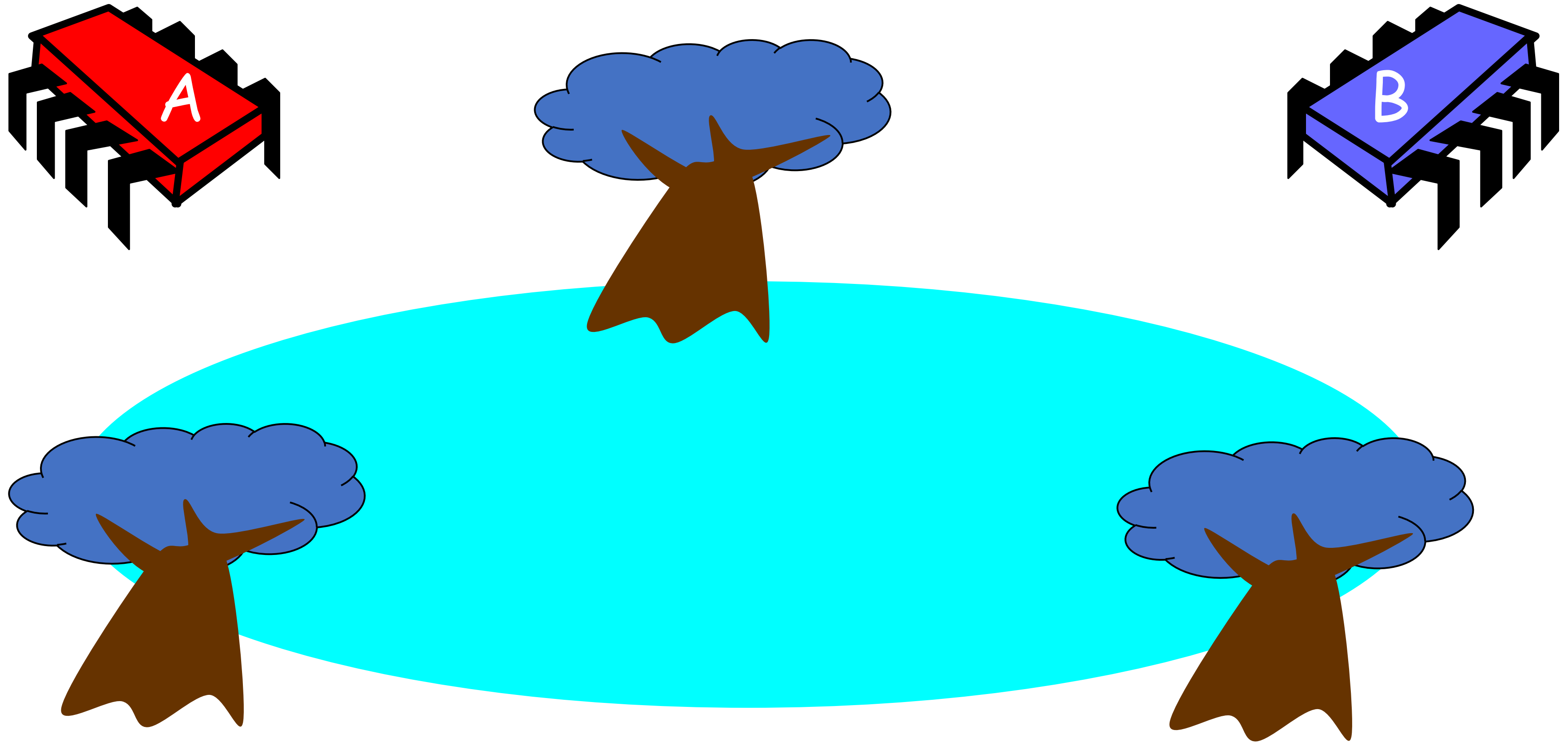
Synchronized block

An Aside: Java™

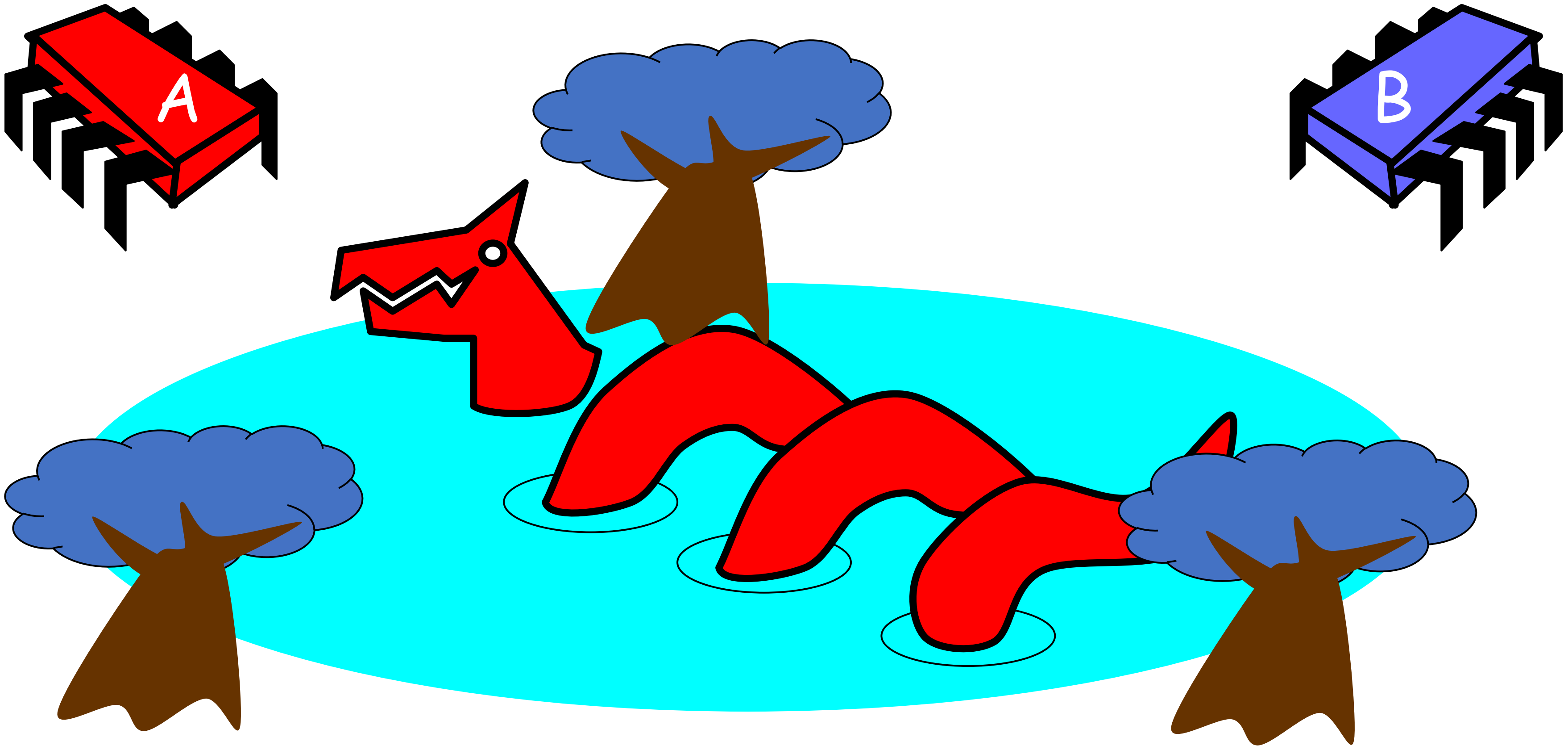
Mutual Exclusion

```
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

The pets don't get along

Formalizing the Problem

Two types of formal properties in asynchronous computation:

- Safety Properties
  - “Nothing bad happens ever”
- Liveness Properties
  - “Something good happens eventually”

Formalizing the Pet 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

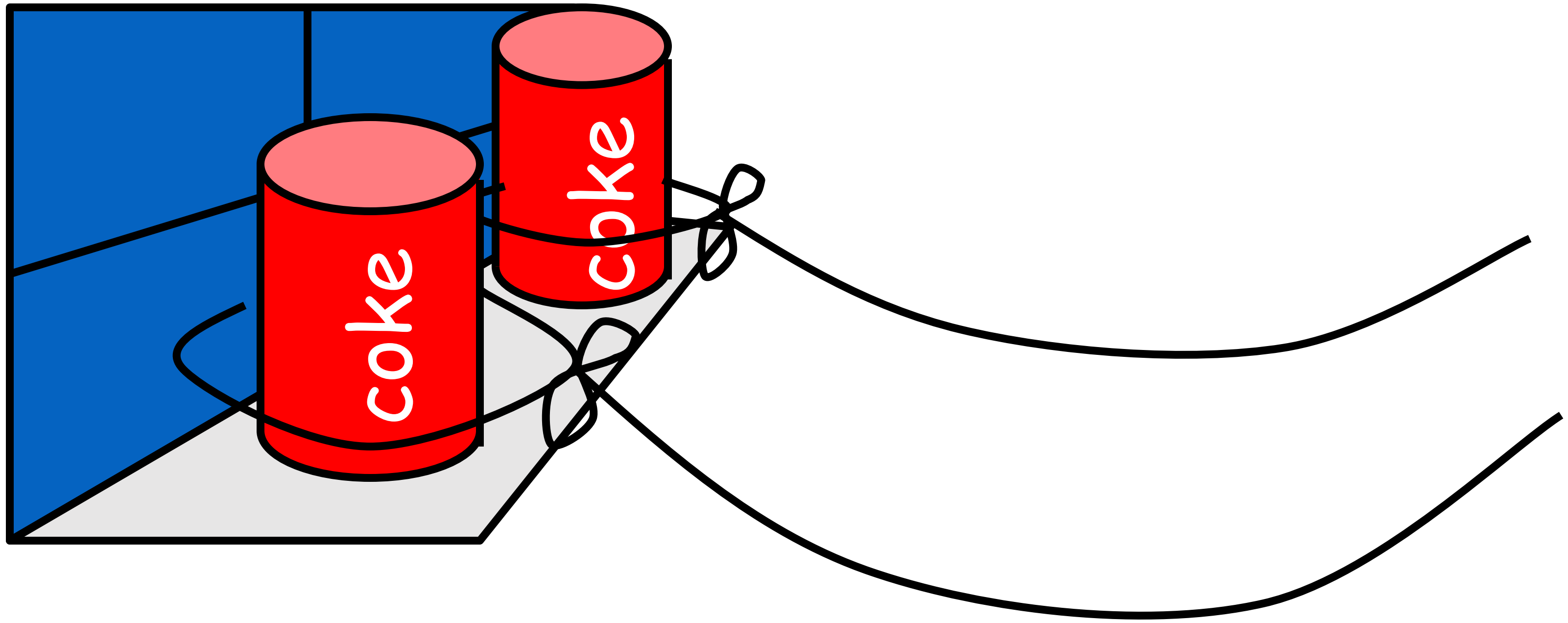
Cell Phone Protocol

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

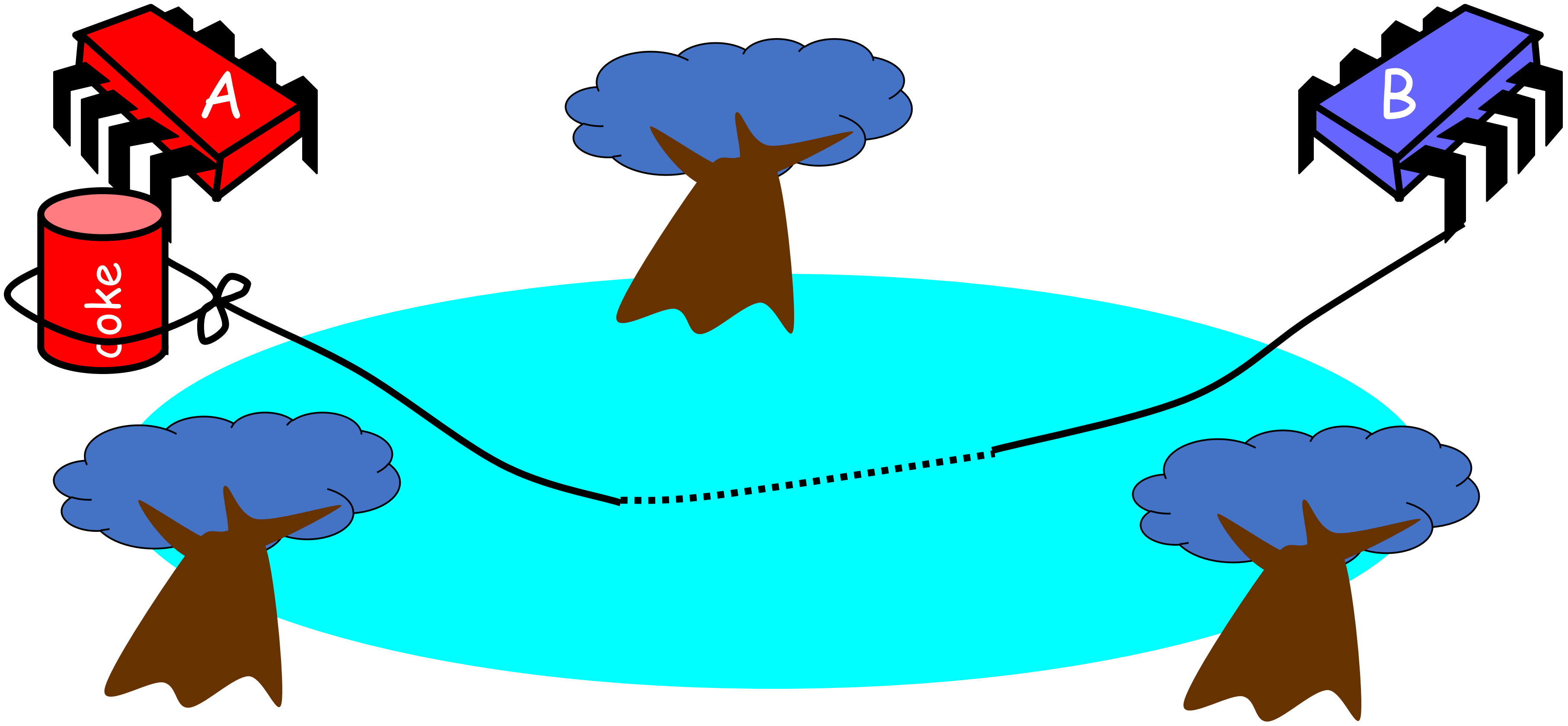
Interpretation

- Message-passing doesn’t work
- Recipient might not be
  - listening
  - there at all
- Communication must be
  - persistent (like writing)
  - not transient (like speaking)

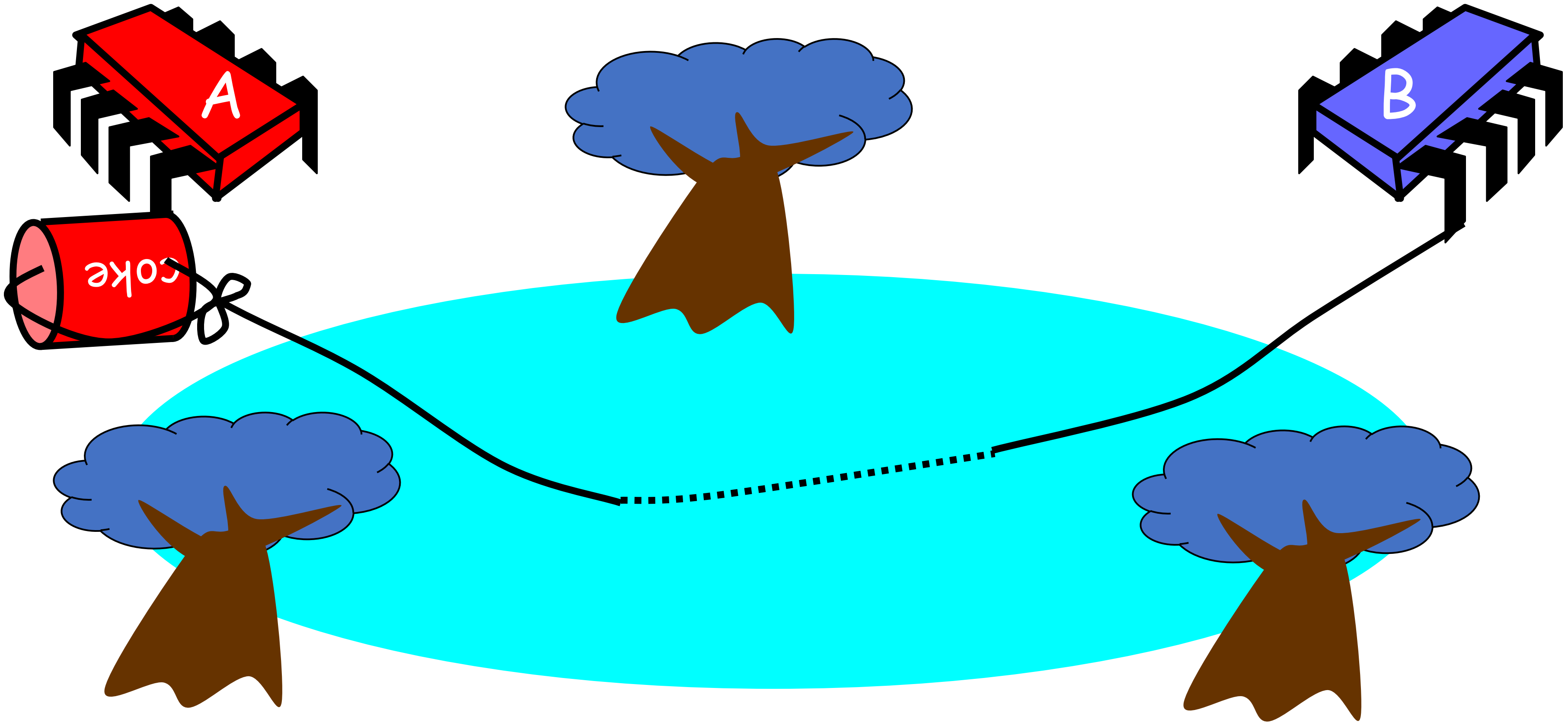
Can Protocol



Bob conveys a bit



Bob conveys a bit



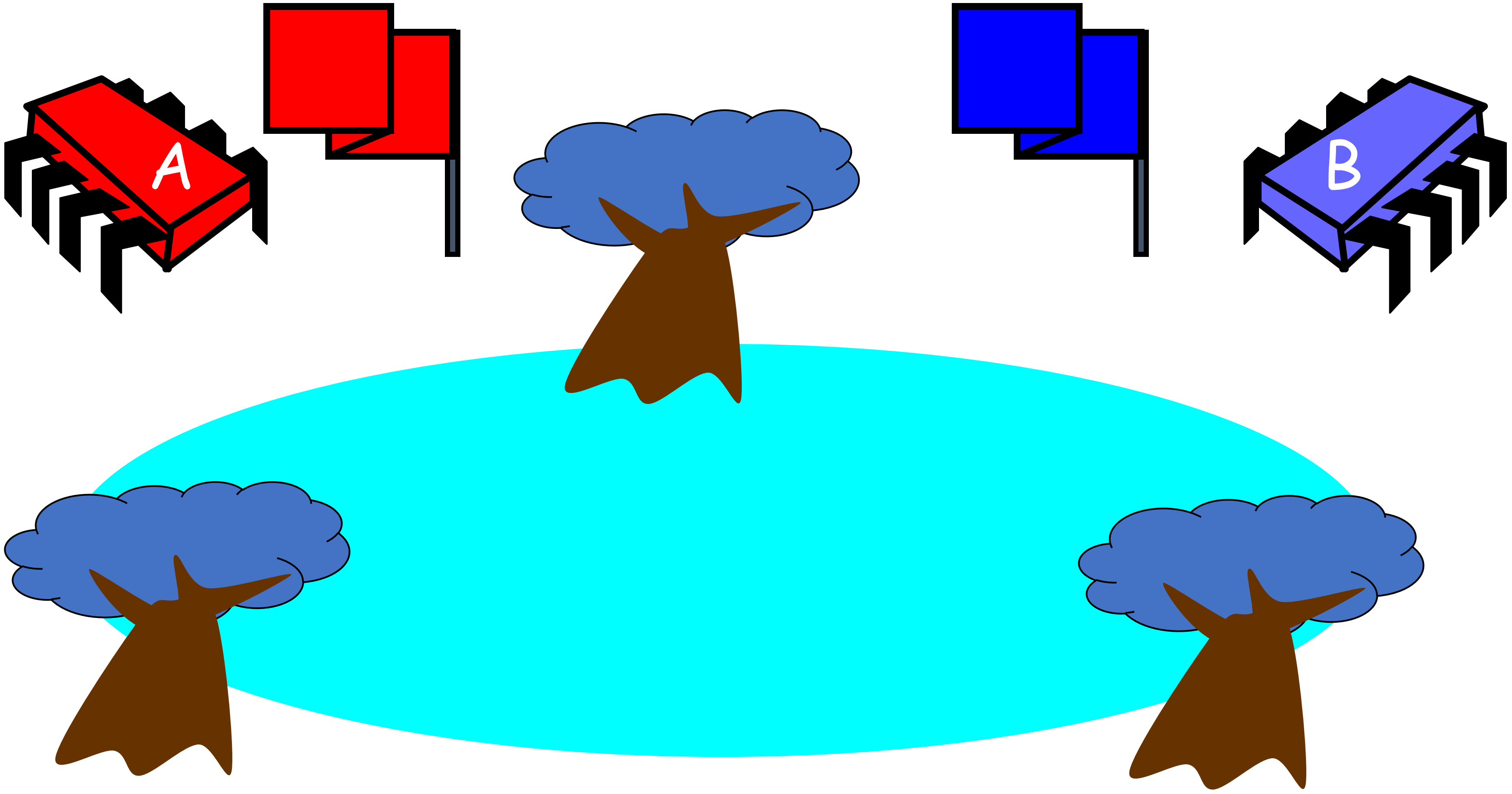
Can Protocol

- Idea
  - Cans on Alice’s windowsill
  - Strings lead to Bob’s house
  - Bob pulls strings, knocks over cans
- Gotcha
  - Cans cannot be reused unless Alice sets them up again
  - Bob runs out of cans

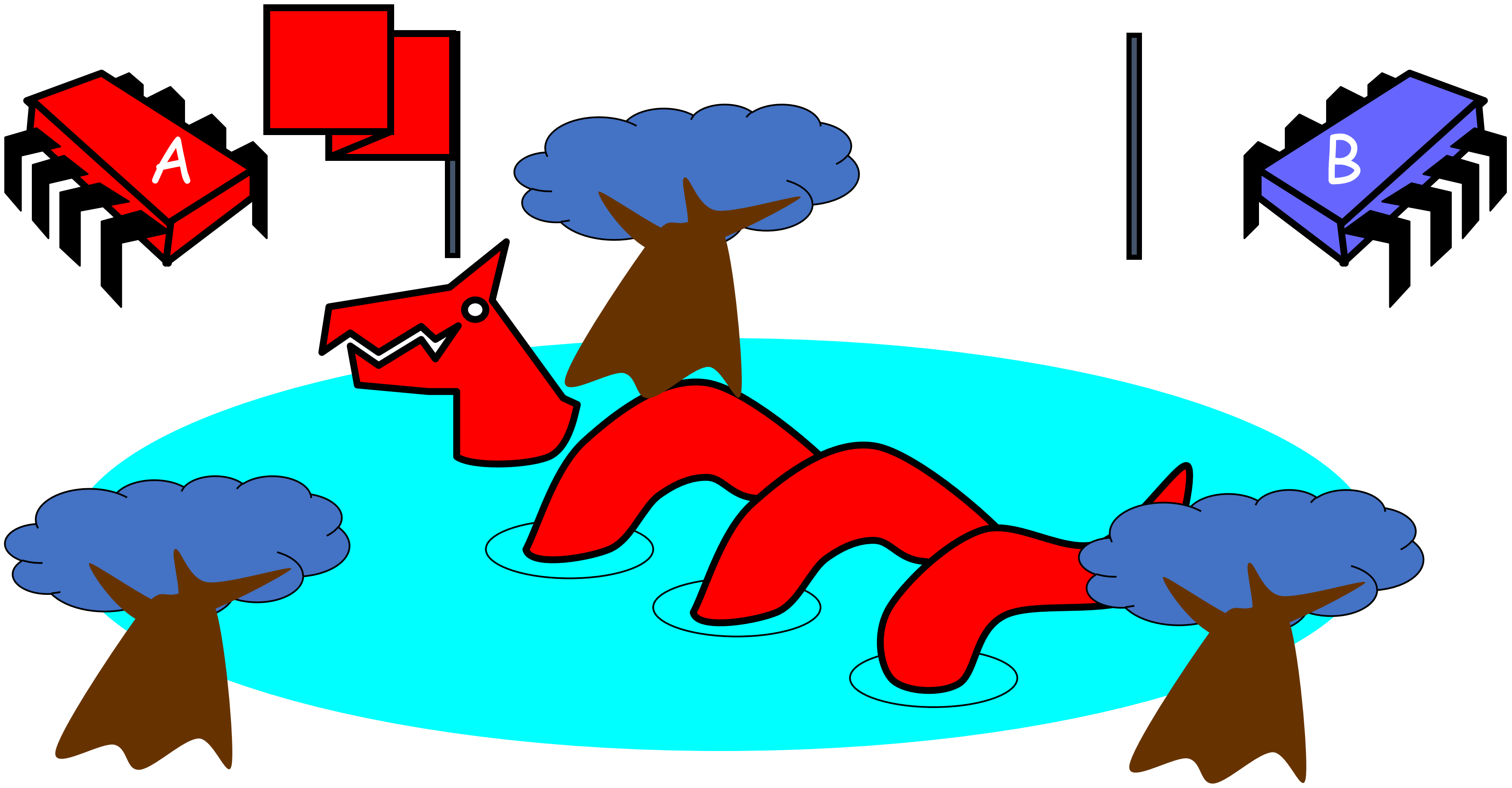
Interpretation

- 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

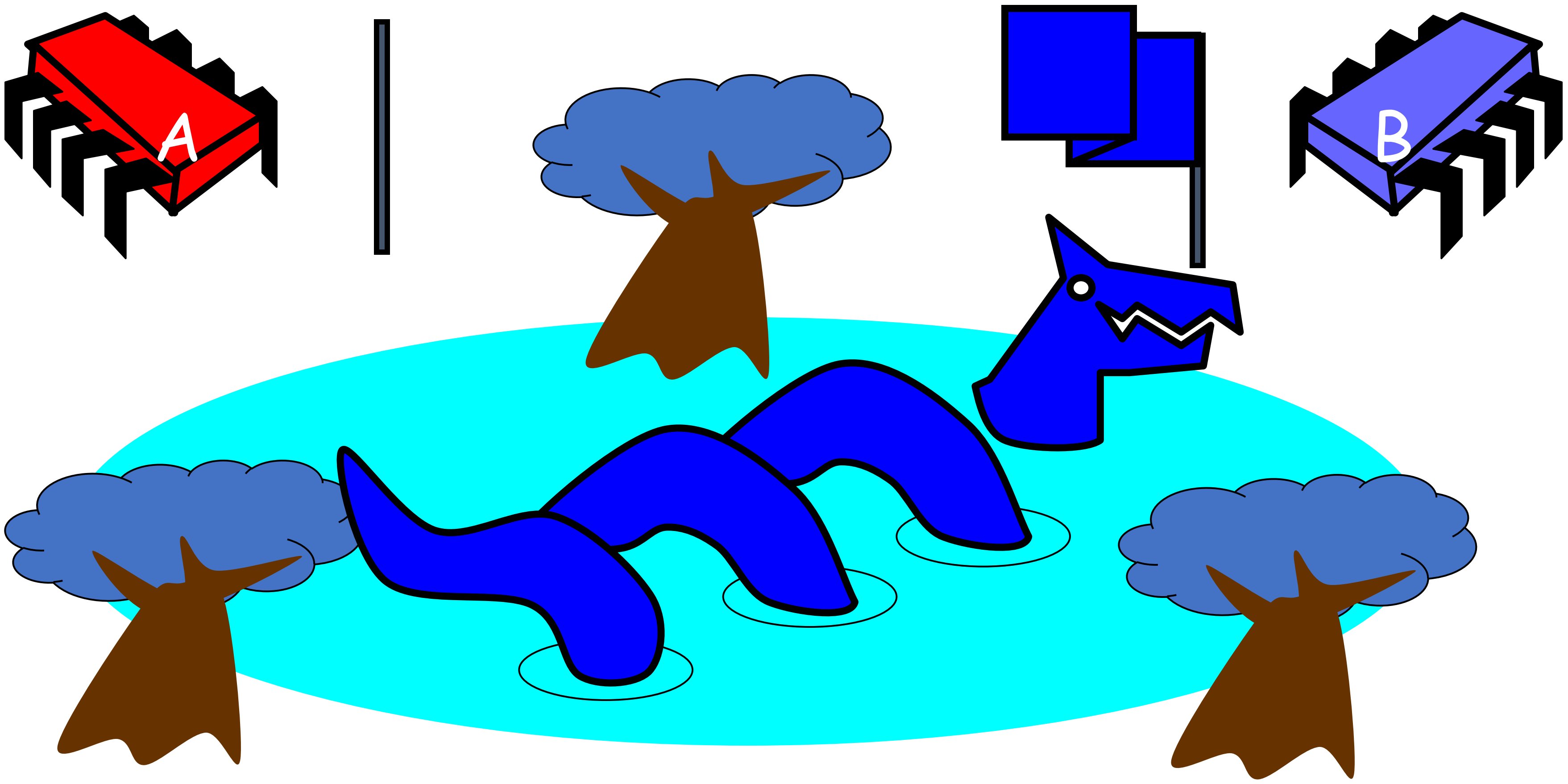
Flag Protocol



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 defers to Alice

The Flag Principle

- Raise the flag
- Look at other’s flag

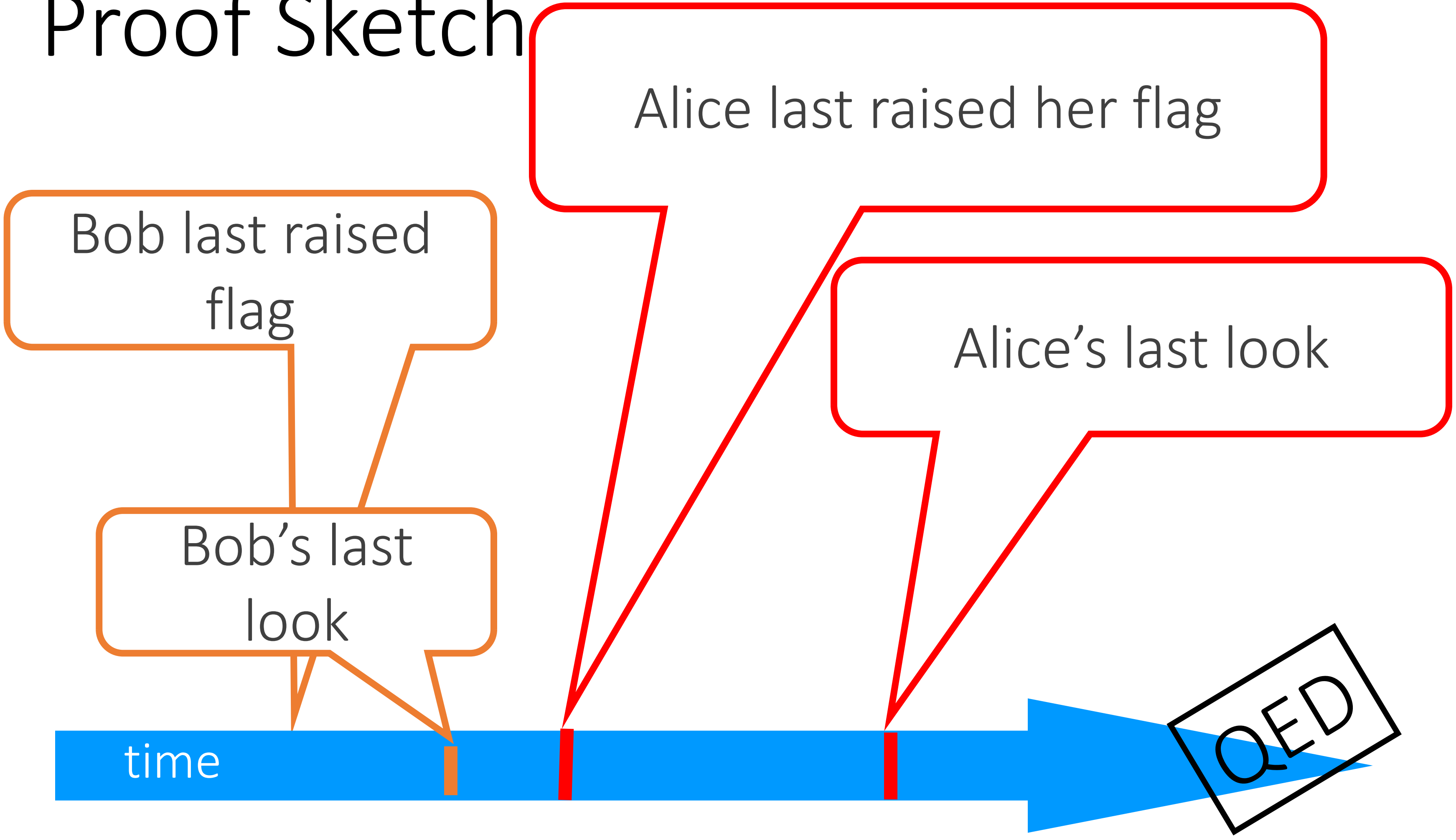
Flag Principle:

- If each raises and looks, then the 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 Sketch



Alice must have seen Bob's Flag → Contradiction

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

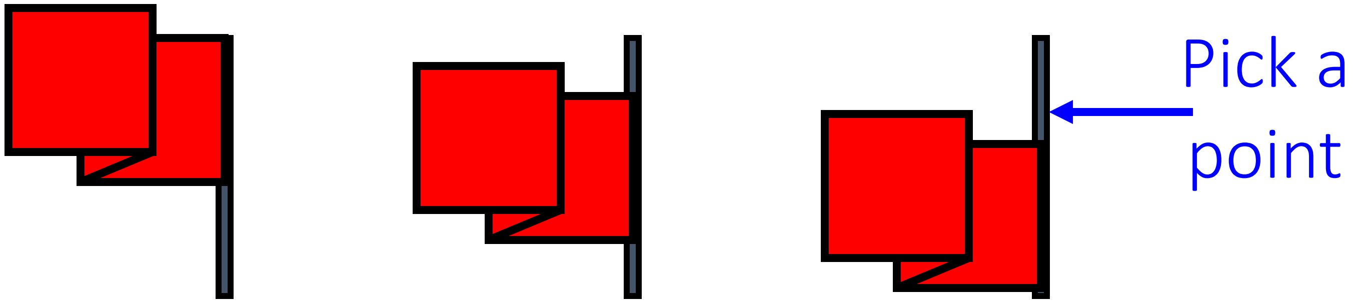
Remarks

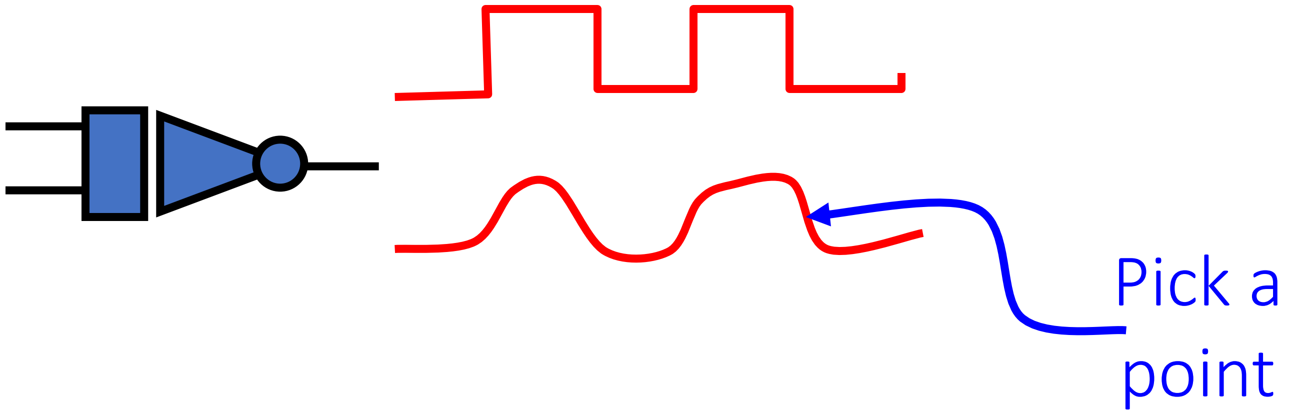
- Protocol is *unfair*
  - Bob’s pet might never get in
- Protocol uses *waiting*
  - If Bob is eaten by his pet, Alice’s pet might never get in

Moral of Story

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

The Arbiter Problem (an aside)

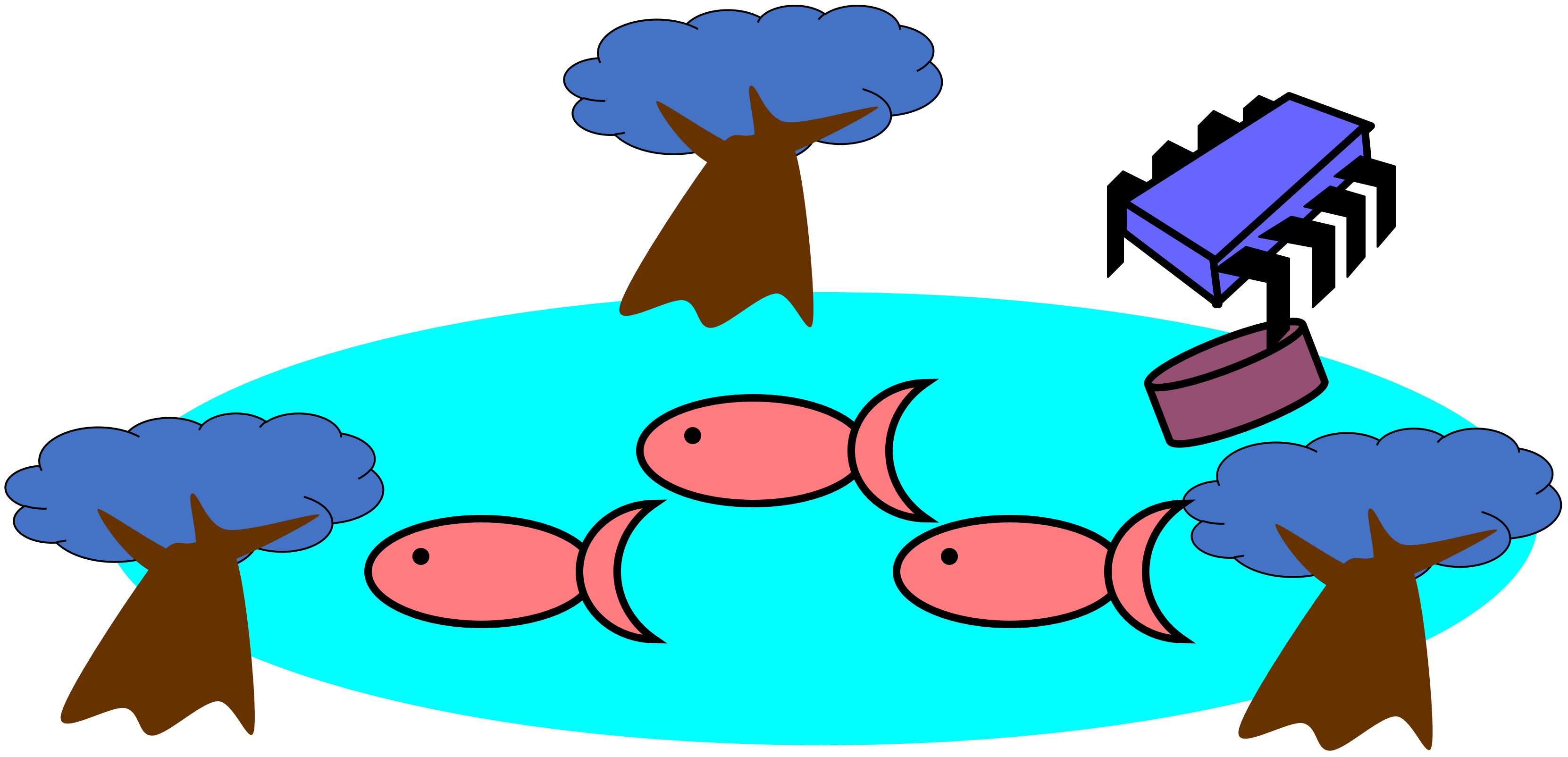




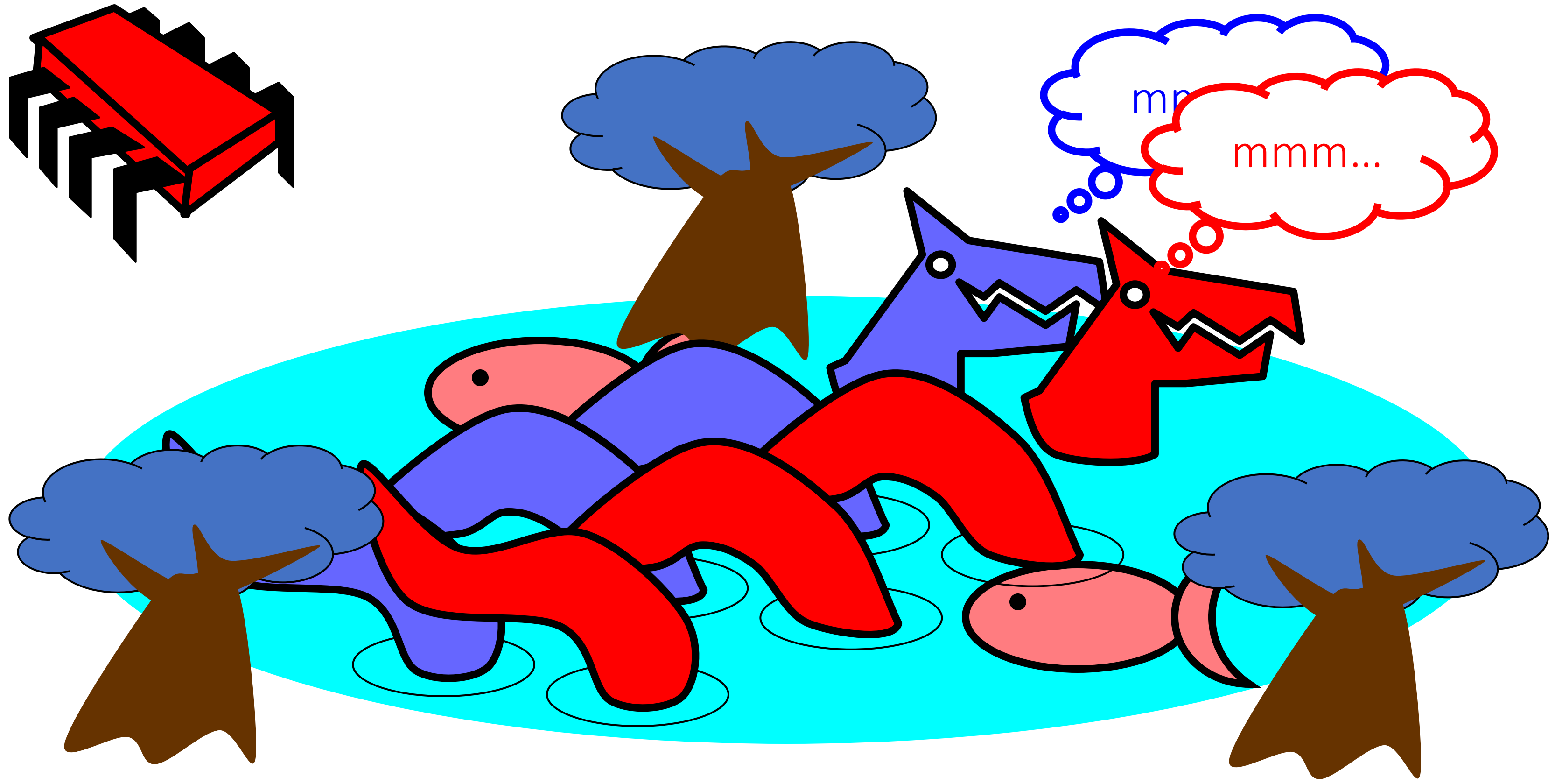
The Fable Continues

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

Bob puts food in the pond



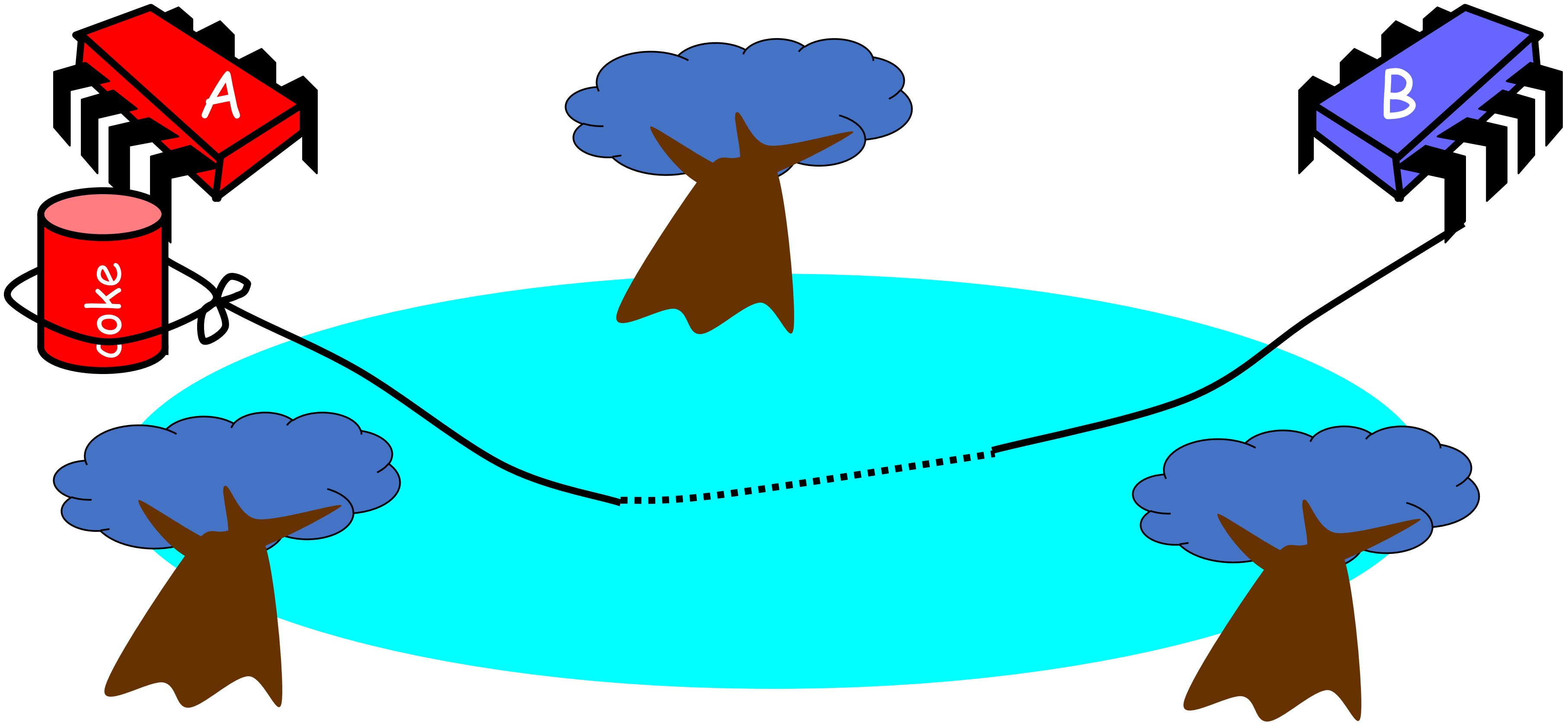
Alice releases her pets to feed



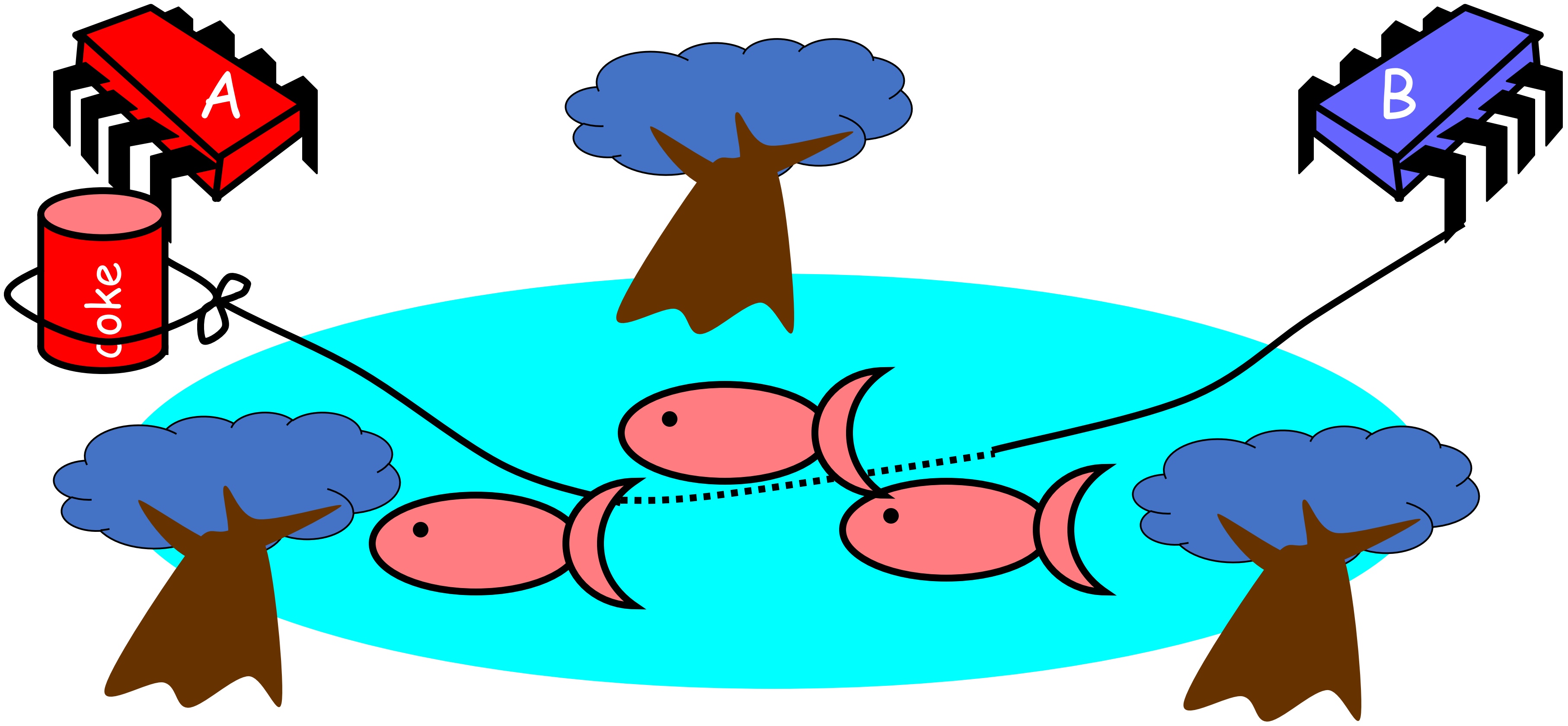
The Producer/Consumer Problem

- 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
- They must avoid
  - Releasing pets when there’s no food
  - Putting out food if uneaten food remains
- 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

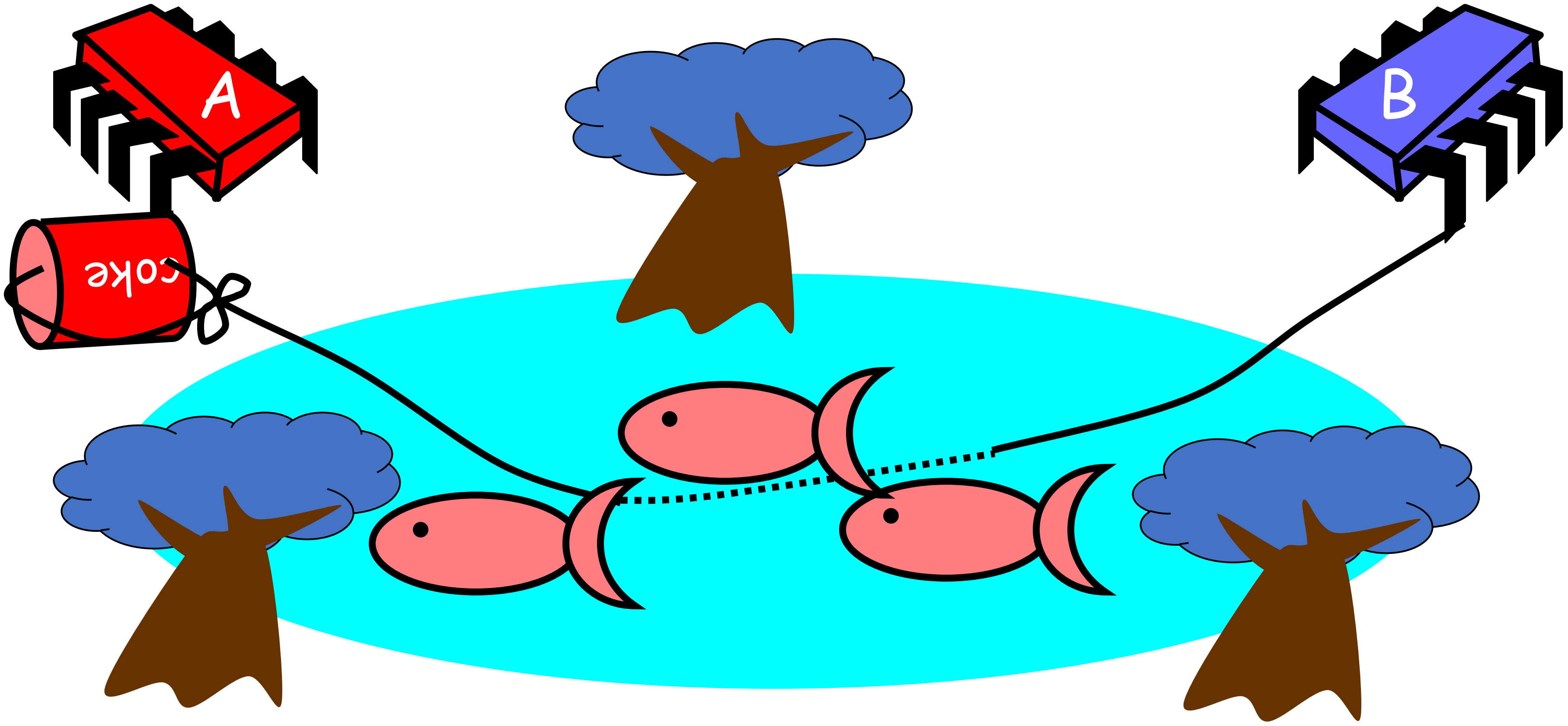
Surprise Solution



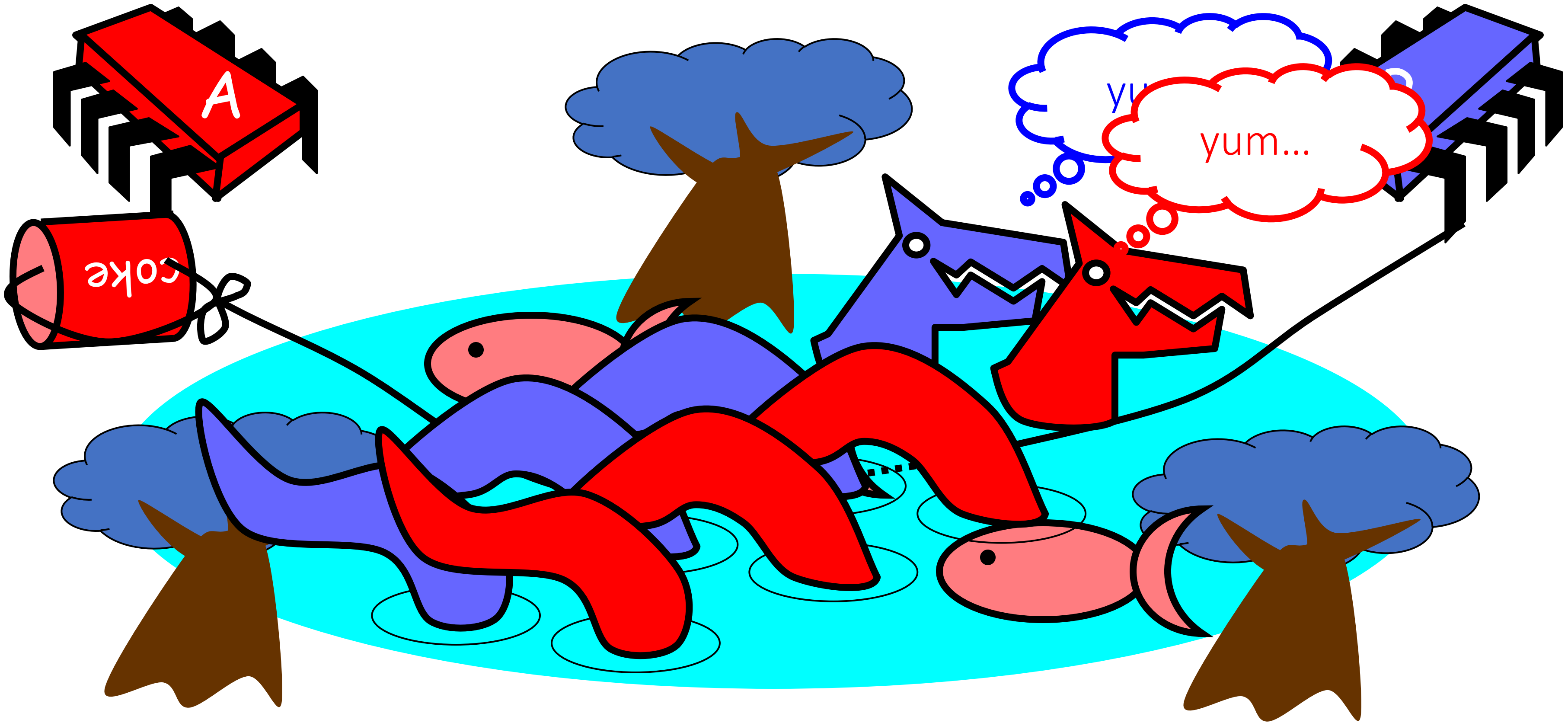
Bob puts food in pond



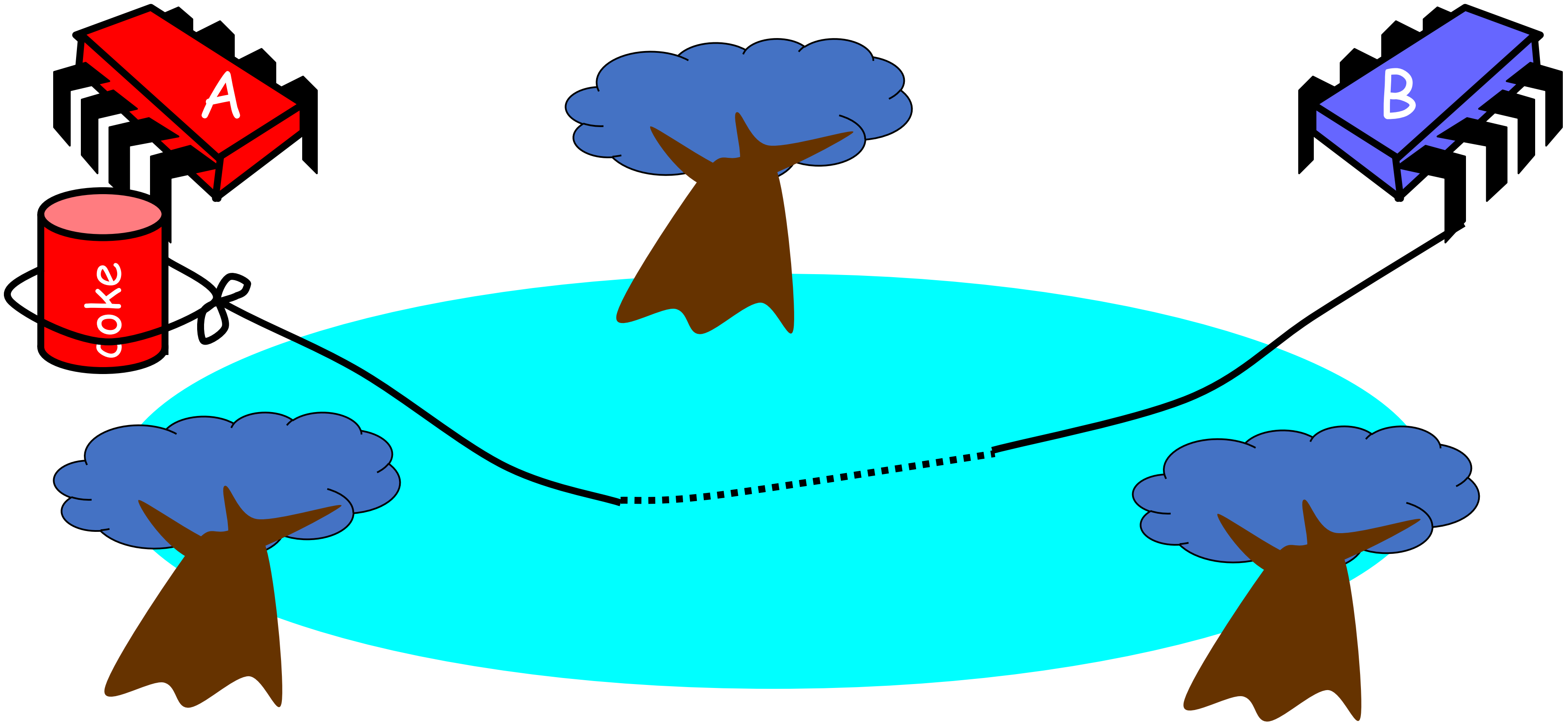
Bob knocks over Can



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

Bob's code

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

Correctness

- Mutual Exclusion

  safety

  Pets and Bob never together in pond

- No Starvation

  liveness

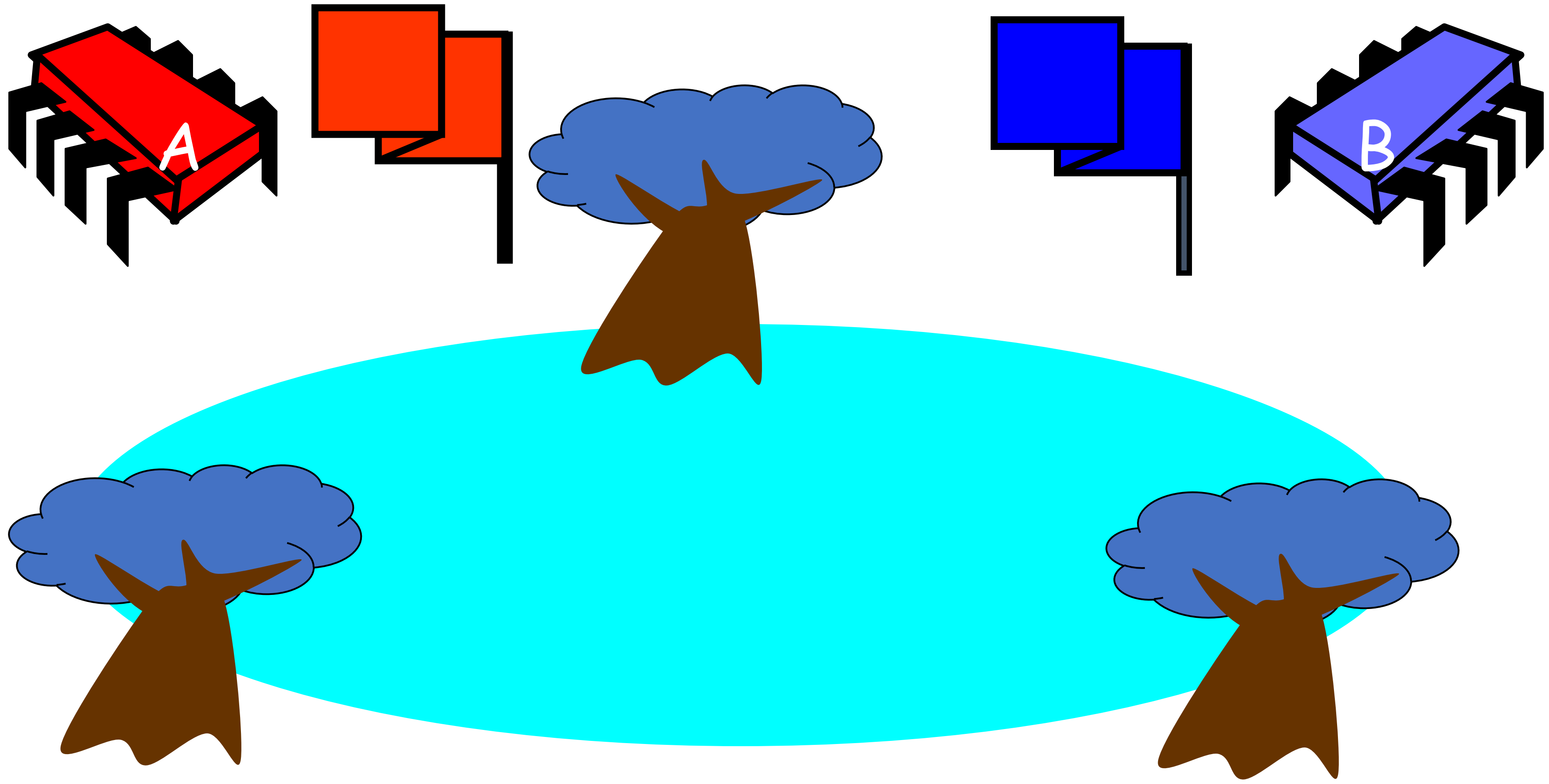
  If Bob always willing to feed, and pets always famished, then pets eat infinitely often.

- Producer/Consumer

  safety

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

Could Also Solve Using Flags



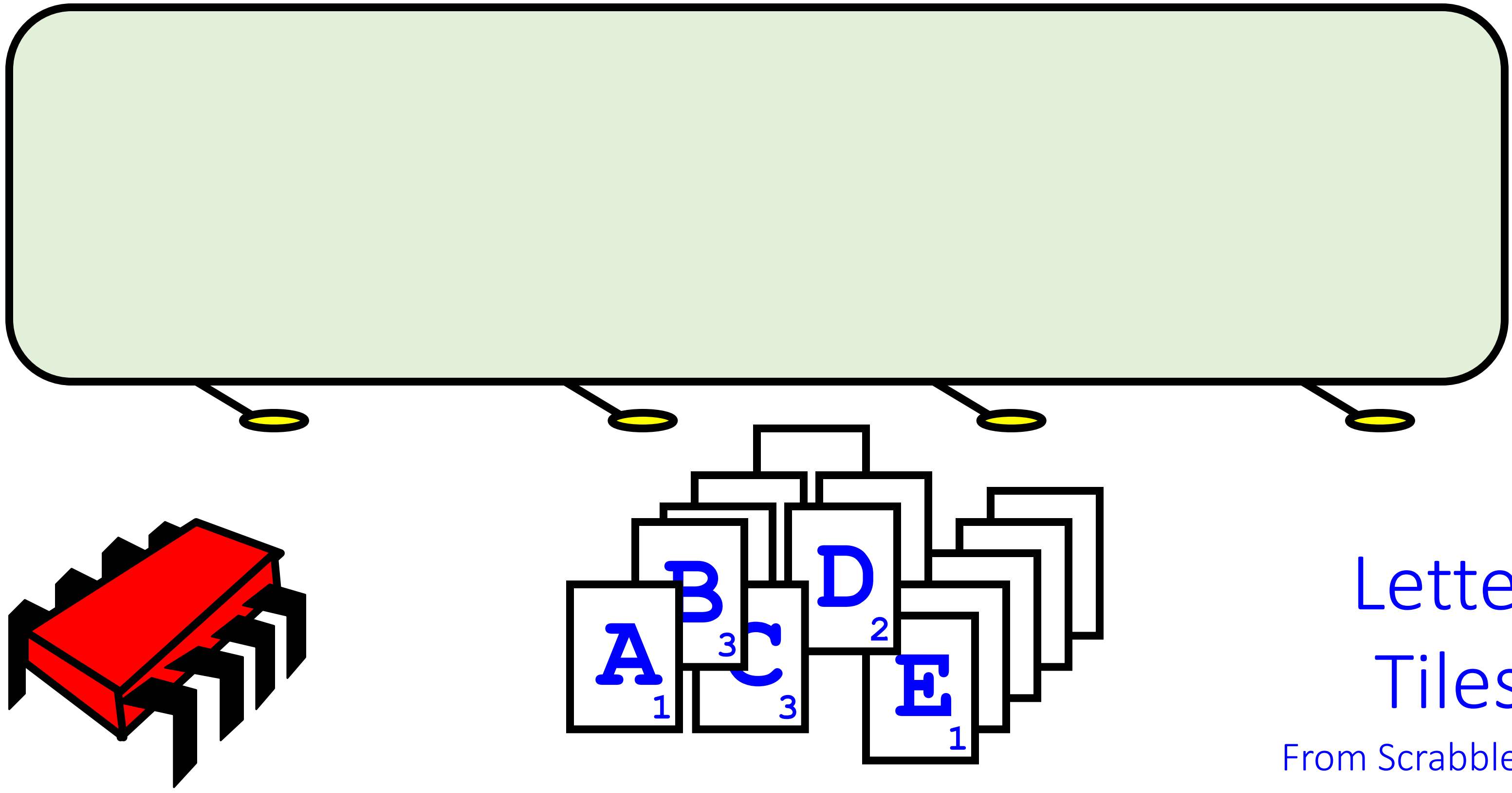
Waiting

- Both solutions use waiting
  - **`while(mumble){}`**
- In some cases, waiting is ***problematic***
  - If one participant is delayed, so is everyone else
  - Delays are common & unpredictable

The Fable drags on …

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

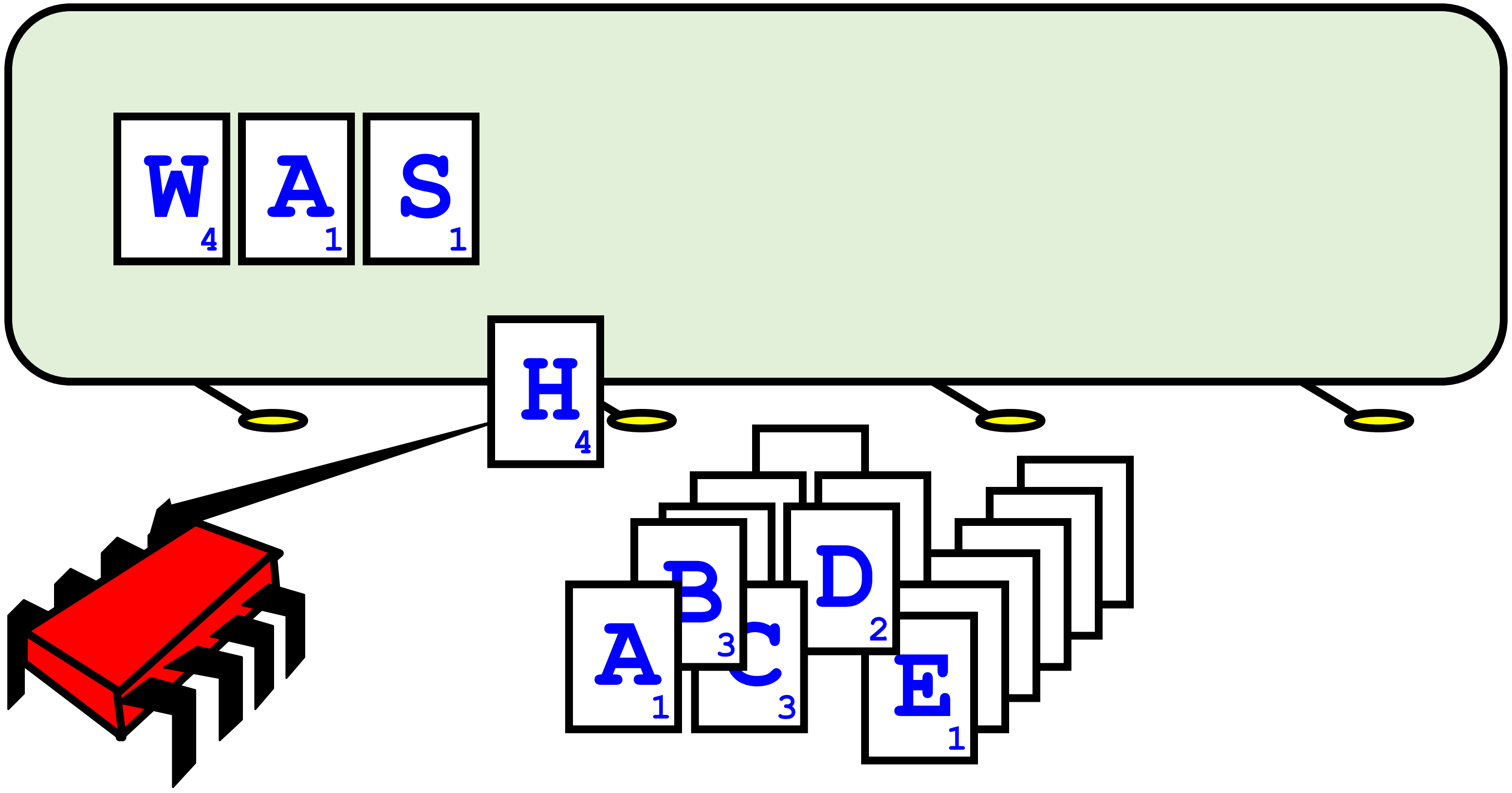
Billboards are Large



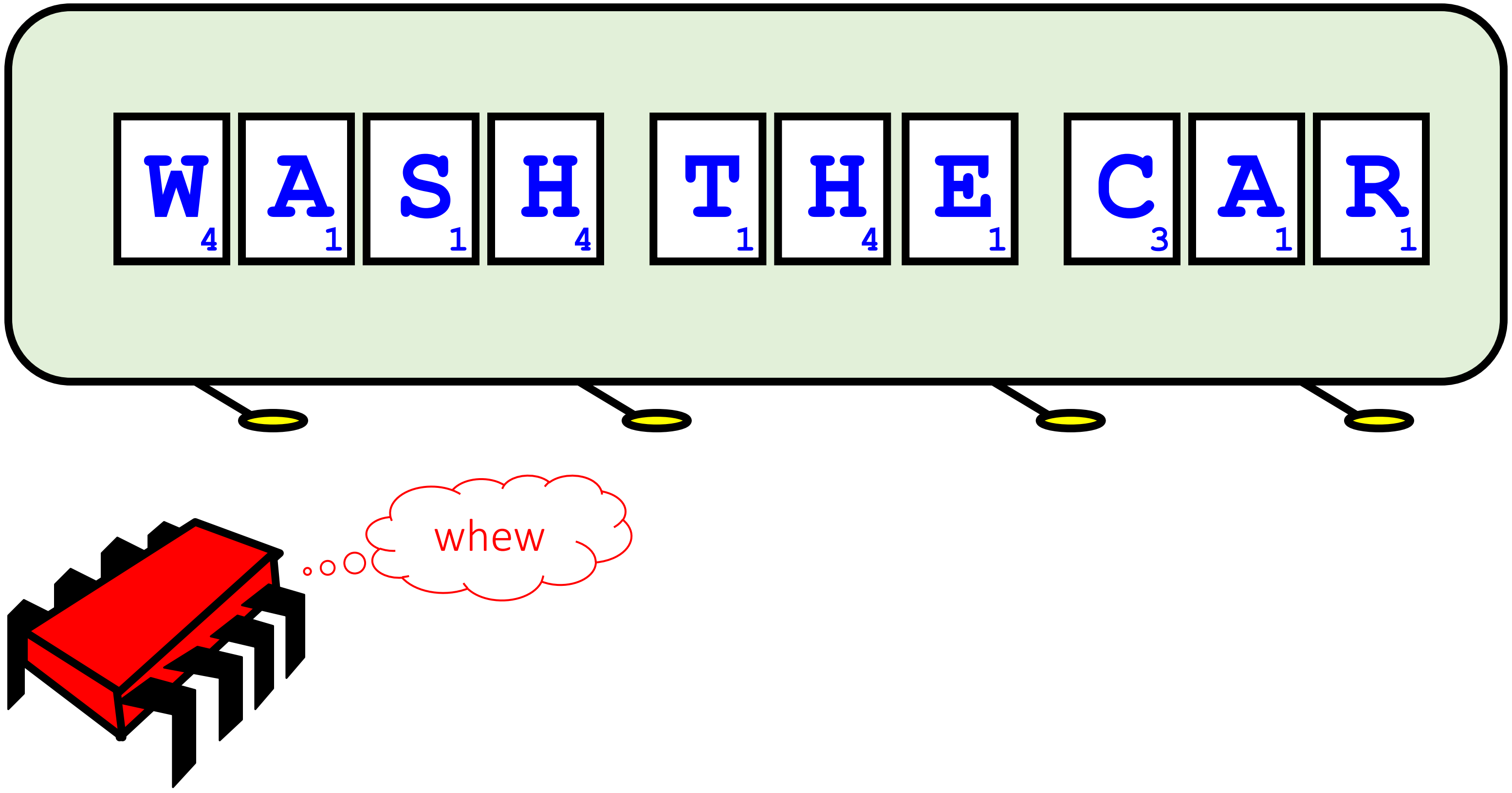
Letter Tiles

From Scrabble™ box

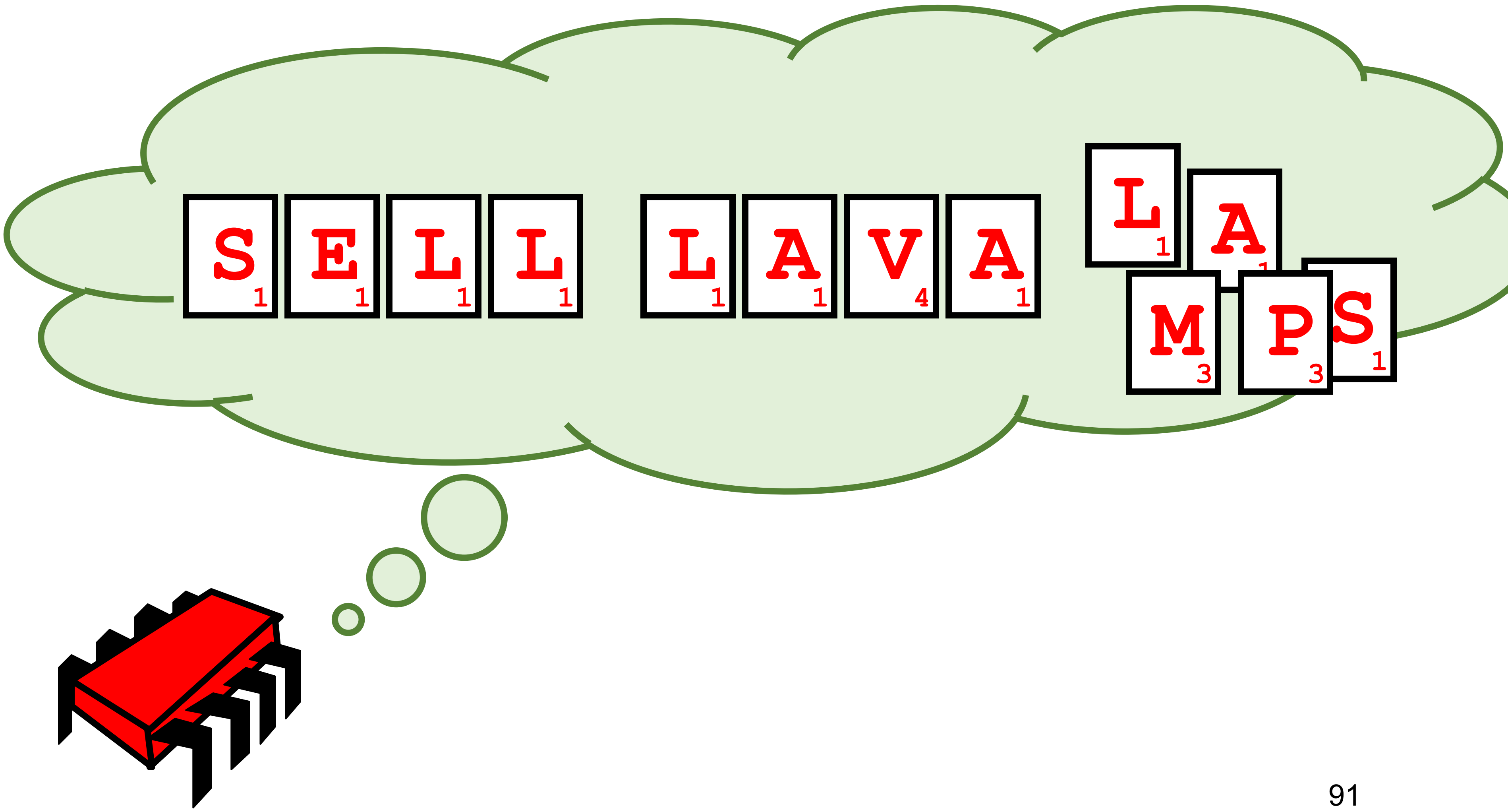
Write One Letter at a Time …



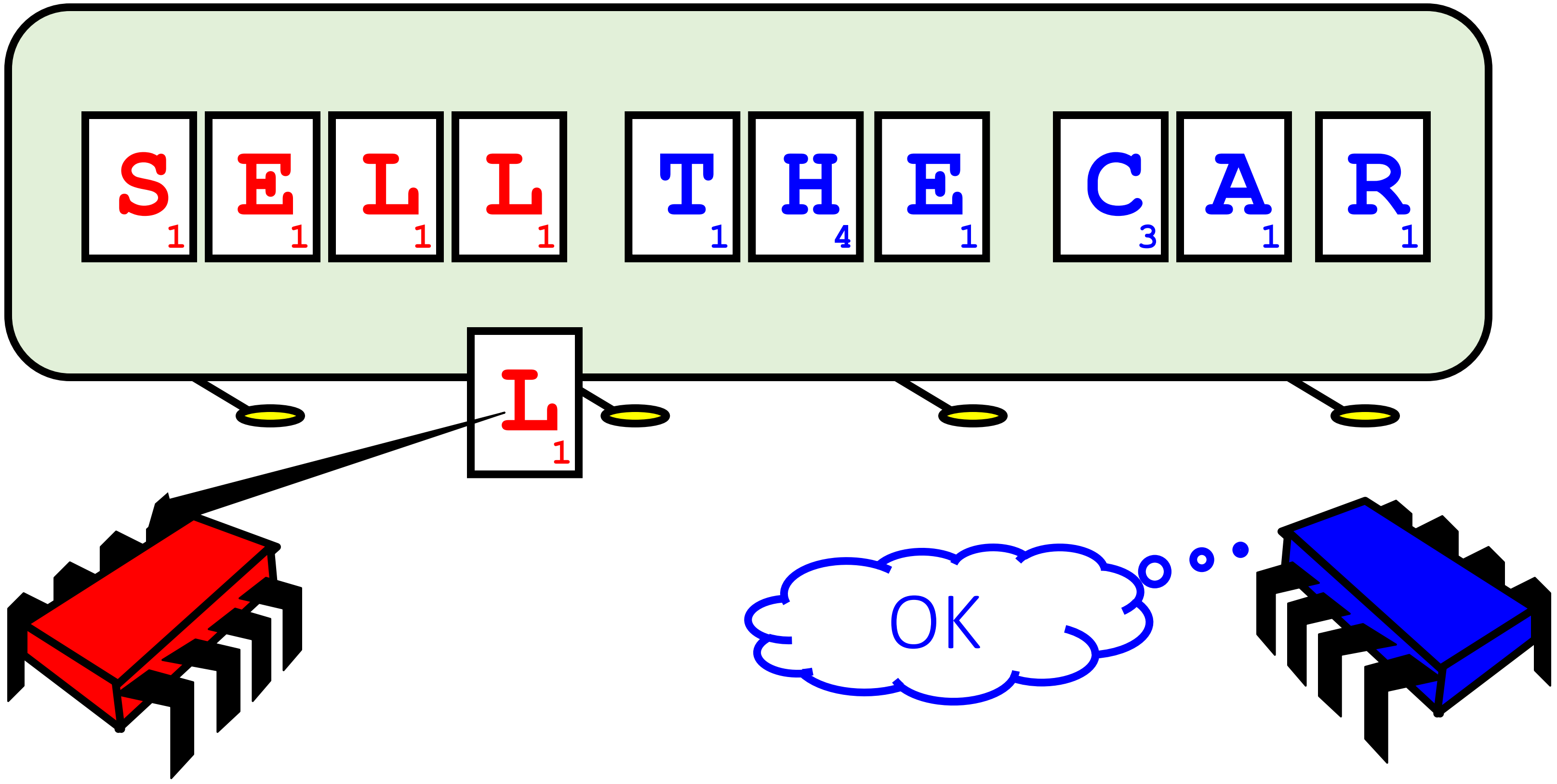
… to post a message



Let’s send another message



Uh-Oh



The Readers/Writers Problem

- 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

The Readers/Writers Problem (continued)

- Easily solvable with mutual exclusion
- But mutual exclusion requires waiting
  - One waits for the other
  - Everyone executes sequentially

We can solve R/W without mutual exclusion!

Esoteric?

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

performance bottleneck

Readers/Writers Solution

- Each thread i has **`size[i]`** counter
  - only this thread increments or decrements.
- To get object’s size, a thread reads a “snapshot” of all counters
- This eliminates the bottleneck!

Summary

- 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…
- Synchronization: Contention or Collaboration
- Classical Problems
  - Mutual Exclusion
  - Producer-Consumer
  - Reader-Writer

Homework: Reading

Leslie Lamport : **Solved Problems, Unsolved Problems and NonProblems in Concurrency**
Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing (August, 1984) 1-11.

https://lamport.azurewebsites.net/pubs/solved-and-unsolved.pdf

„This is the invited address I gave at the 1983 PODC conference, which I transcribed from a tape recording of my presentation. The first few minutes of the talk were not taped, so I had to reinvent the beginning. This talk is notable because it marked the rediscovery by the computer science community of Dijkstra's 1974 CACM paper that introduced the concept of self-stabilization. A self-stabilizing system is one that, when started in any state, eventually "rights itself" and operates correctly. The importance of self-stabilization to fault tolerance was obvious to me and a handful of people, but went completely over the head of most readers. Dijkstra's paper gave little indication of the practical significance of the problem, and few people understood its importance. So, this gem of a paper had disappeared without a trace by 1983. My talk brought Dijkstra's paper to the attention of the PODC community, and now self-stabilization is a regular subfield of distributed computing. I regard the resurrection of Dijkstra's brilliant work on self-stabilization to be one of my greatest contributions to computer science.
The paper contains one figure--copied directly from a transparency--with an obviously bogus algorithm. I tried to recreate an algorithm from memory and wrote complete nonsense. It's easy to make such a mistake when drawing a transparency, and I probably didn't bother to look at it when I prepared the paper. To my knowledge, it is the only incorrect algorithm I have published. "

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

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