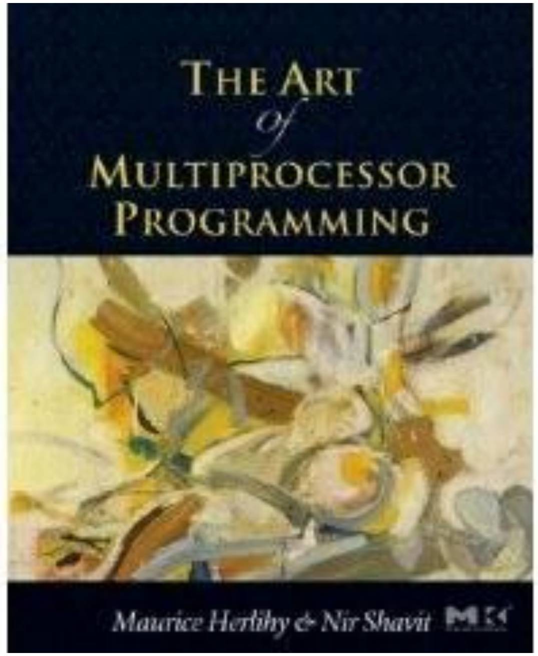
COS 226

Concurrency

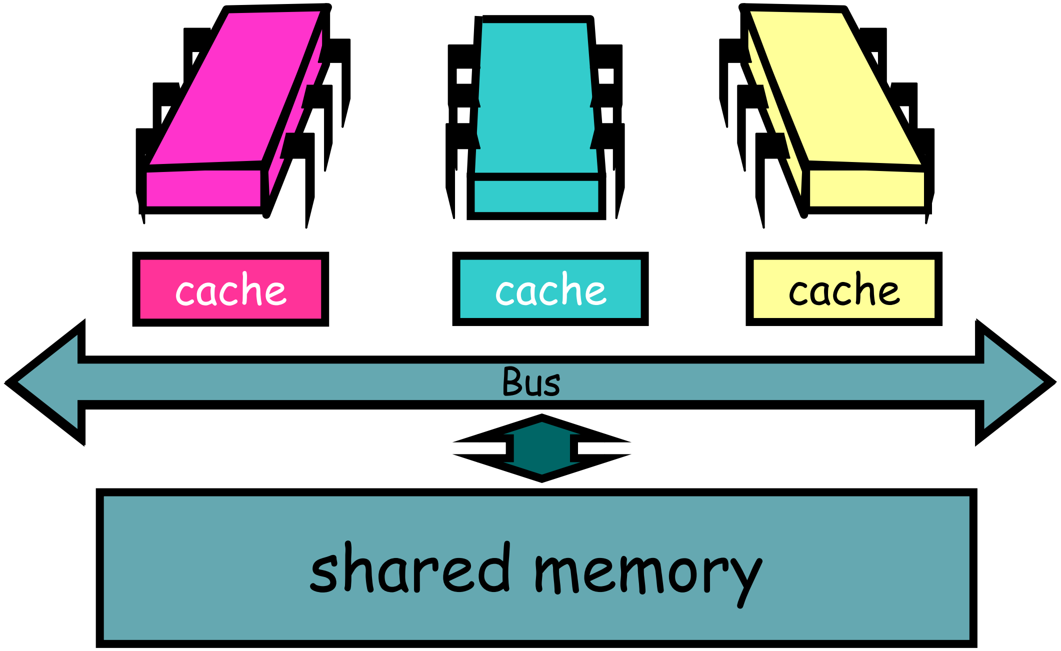
Chapter 1

Acknowledgement

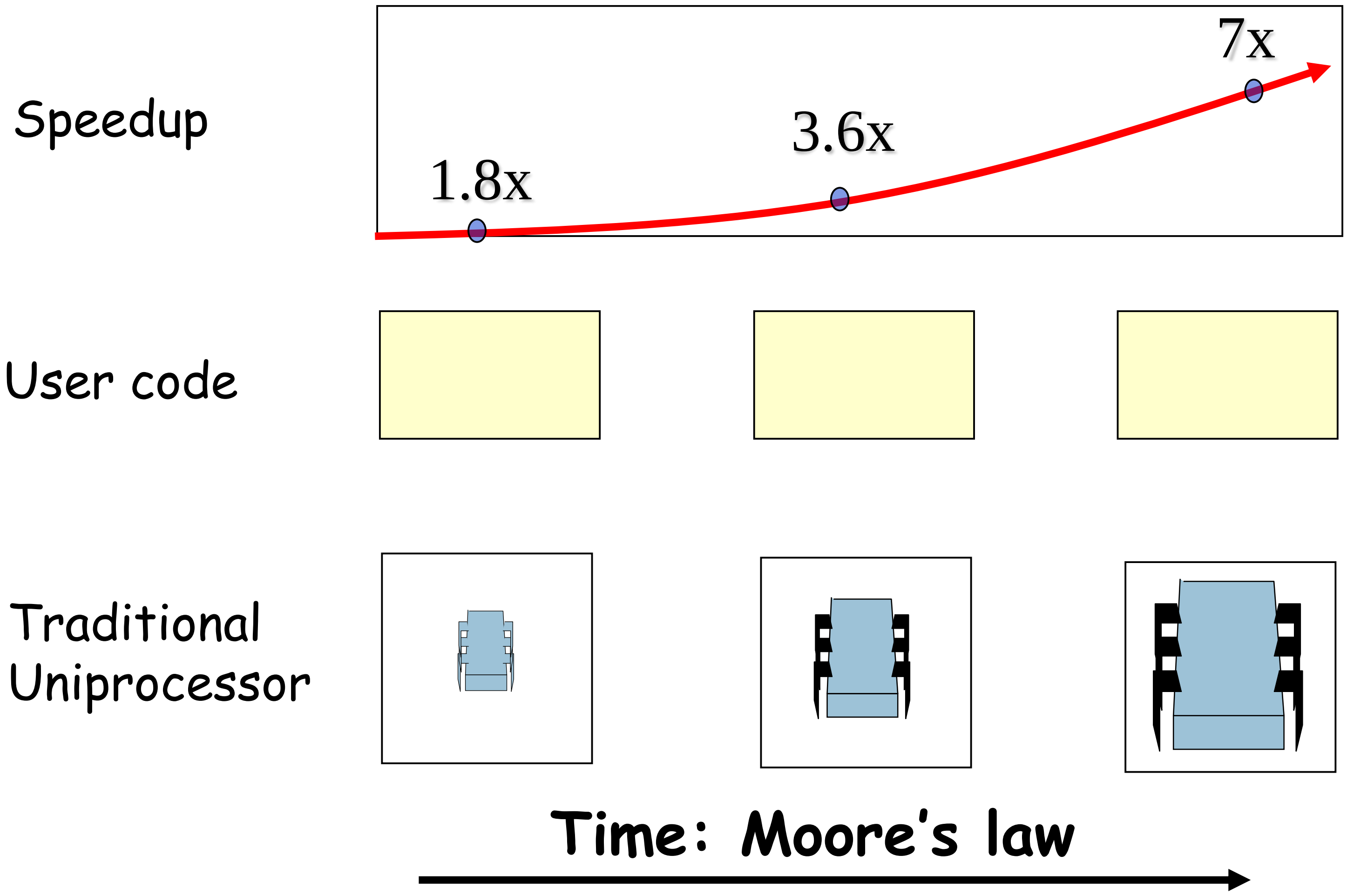


- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit

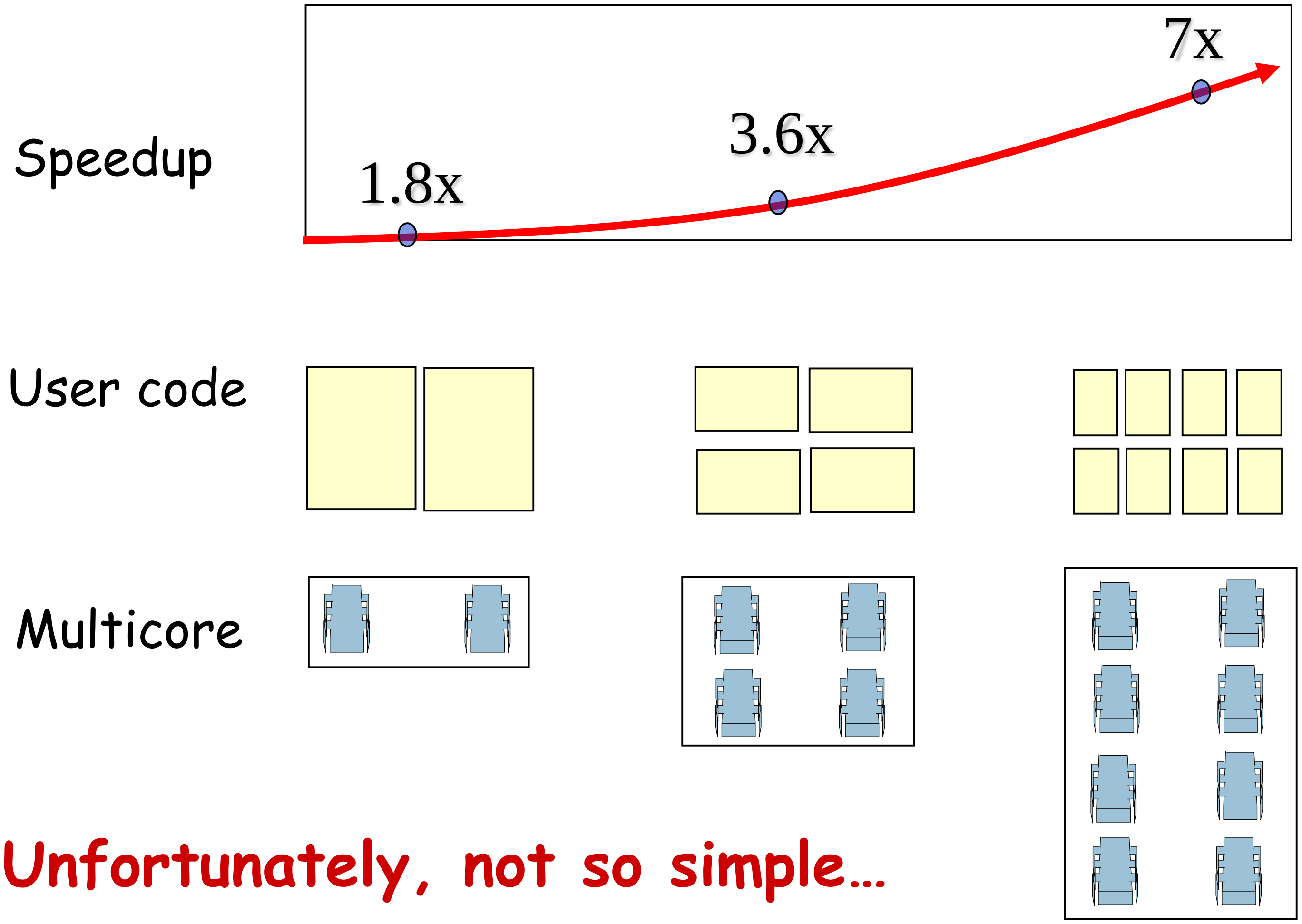
The Shared Memory Multiprocessor (SMP)



Traditional Scaling Process



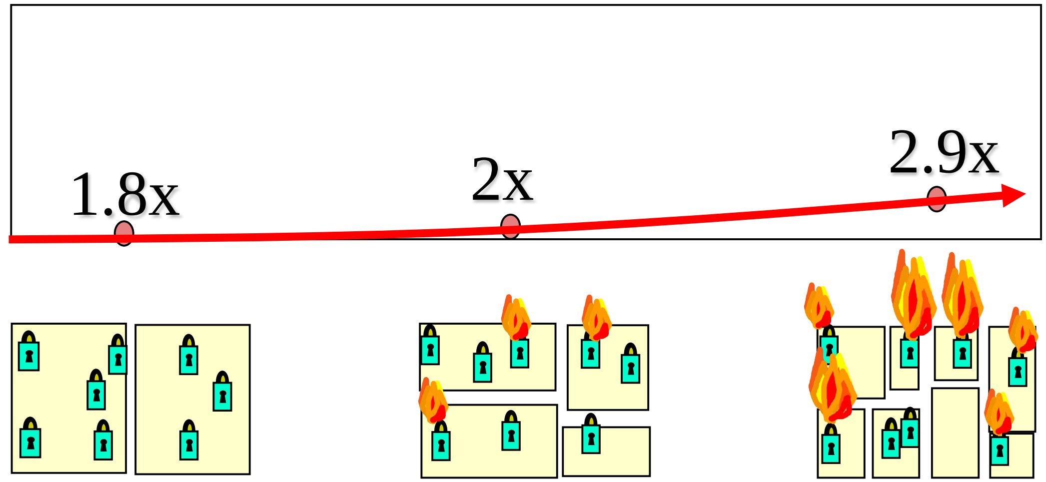
Multicore Scaling Process



Unfortunately, not so simple…

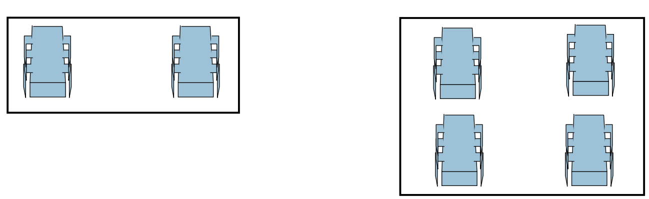
Real-World Scaling Process

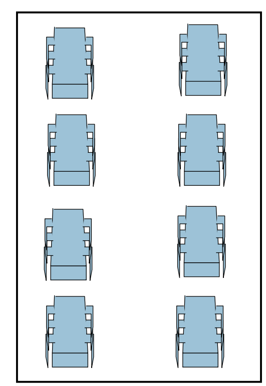
Speedup



User code

Multicore





Parallelization and Synchronization require great care…

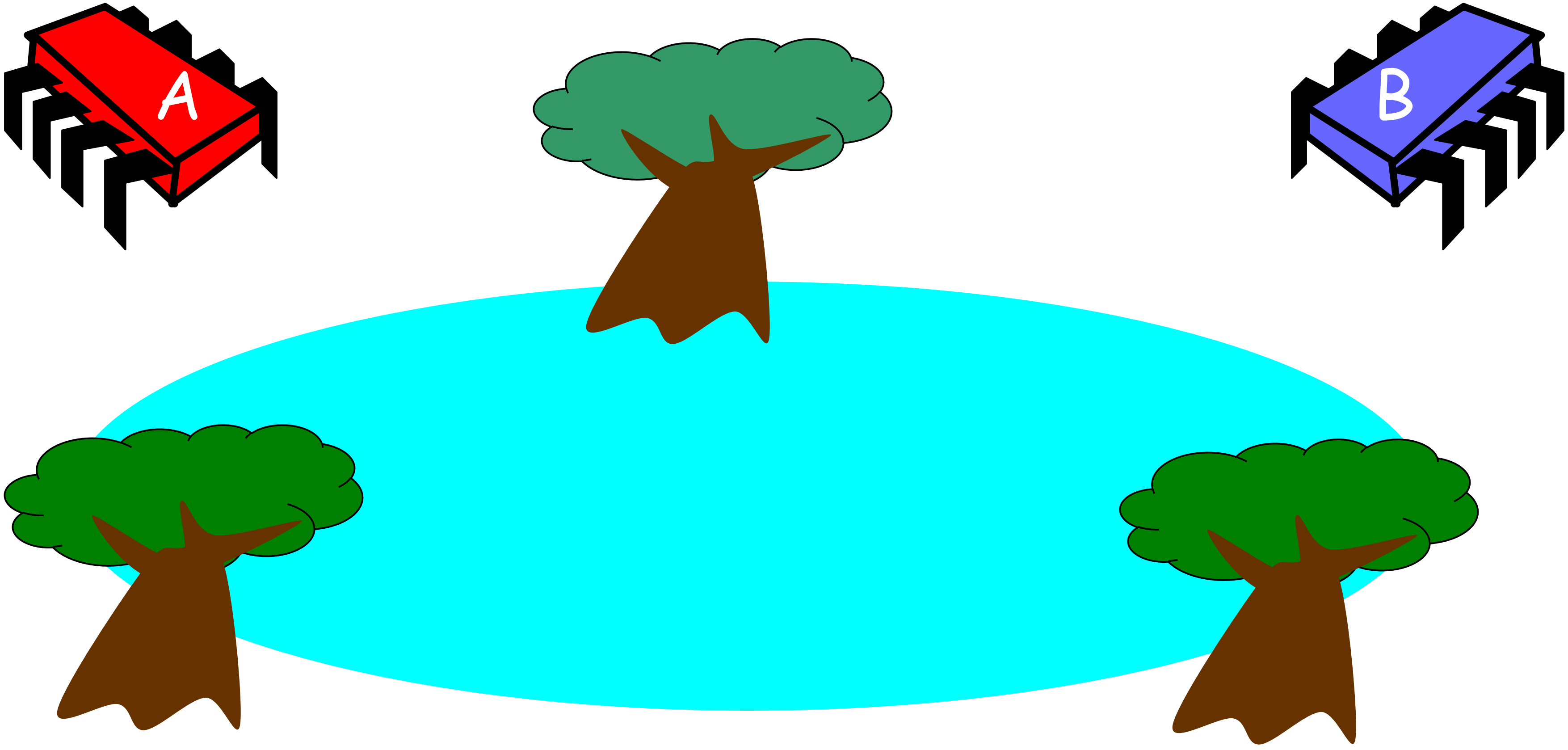
Multiprocessor programming

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

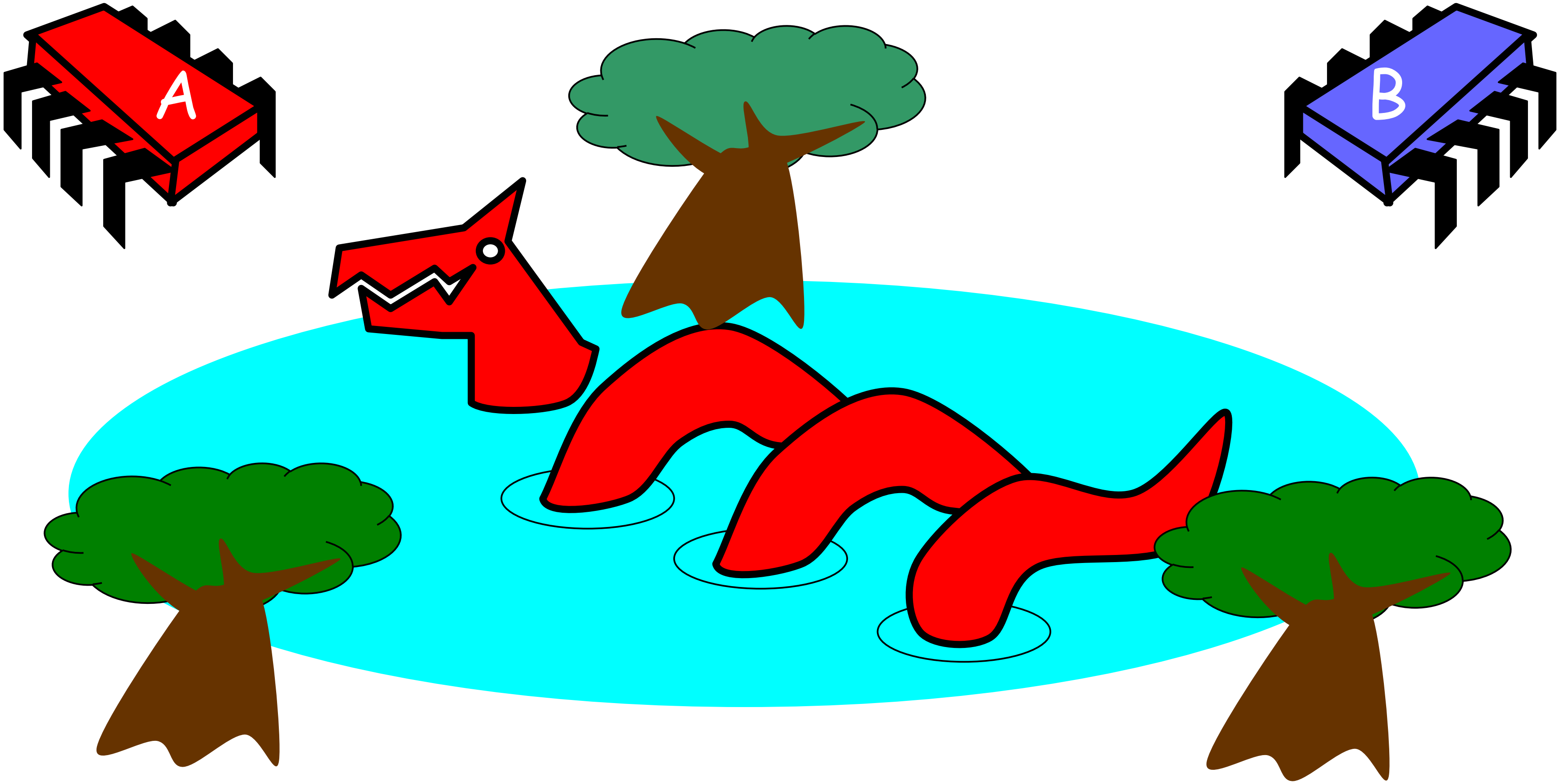
Model Summary

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

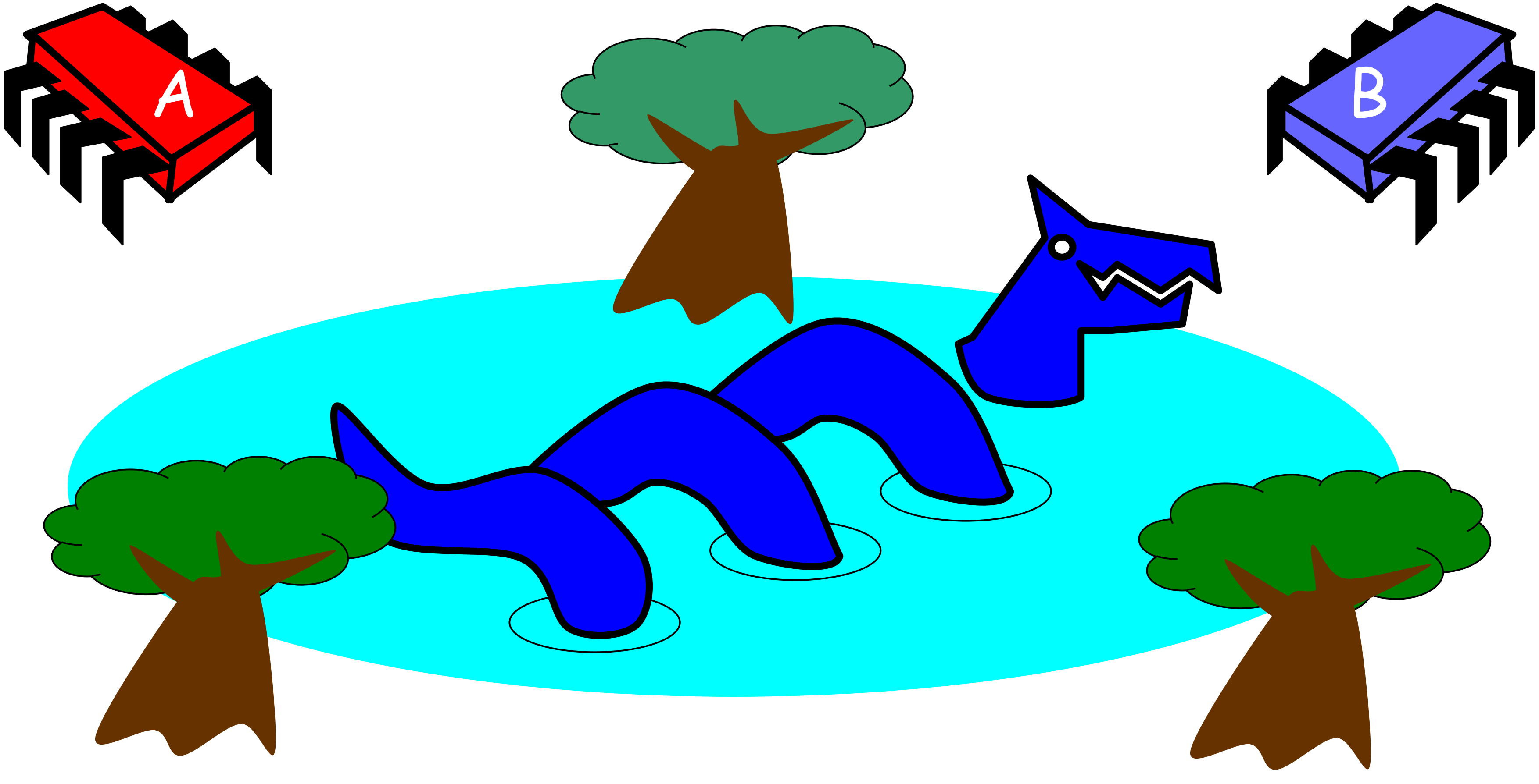
Mutual Exclusion or “Alice & Bob share a pond”



Alice has a pet



Bob has a pet



The Problem

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

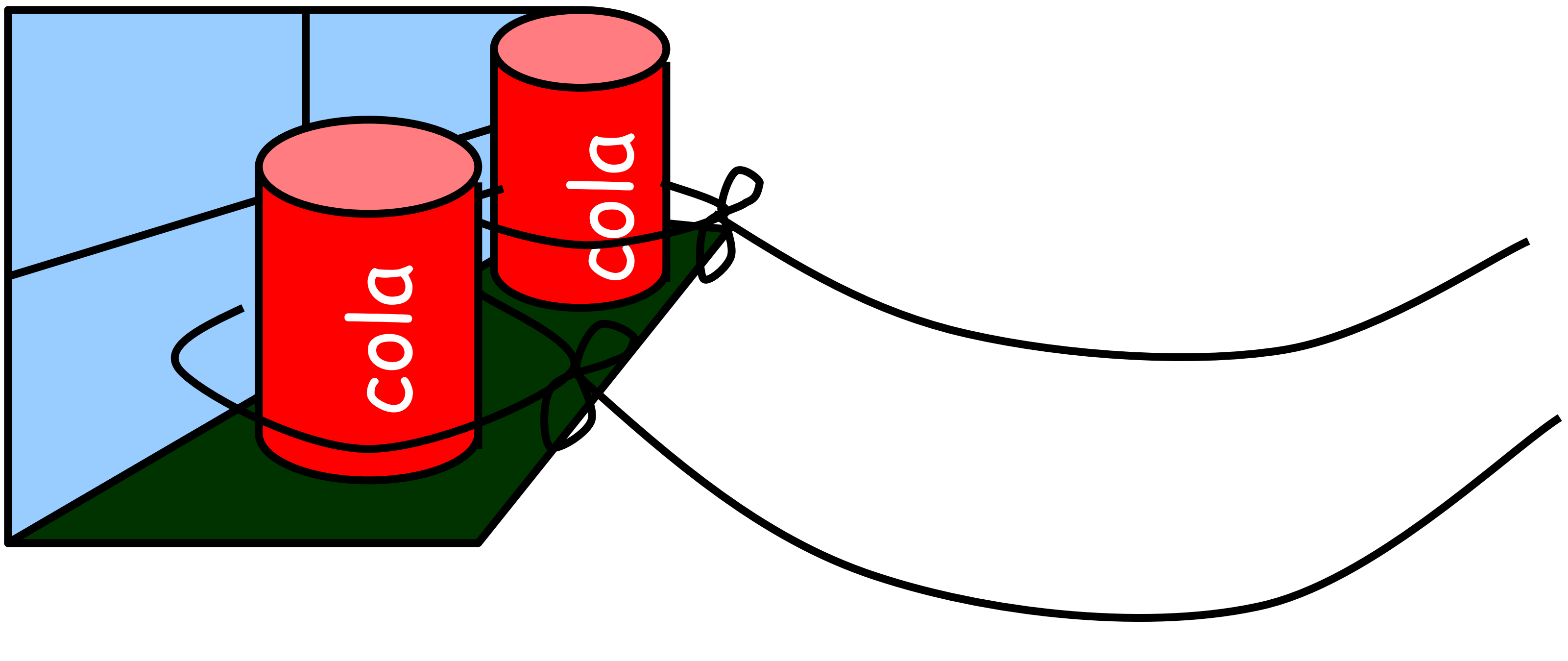
Cell Phone Protocol

- 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)

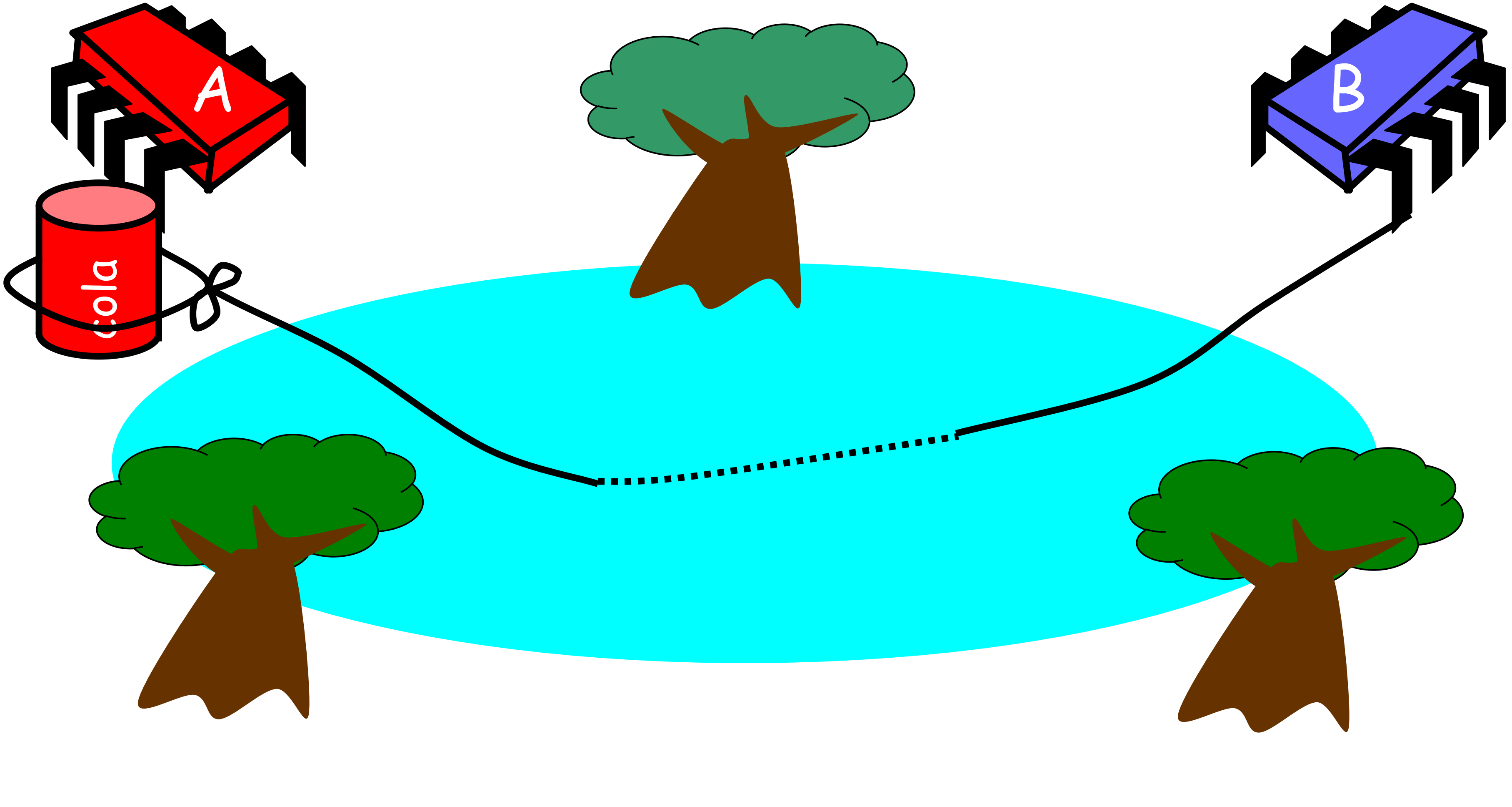
Possible solution: Can Protocol



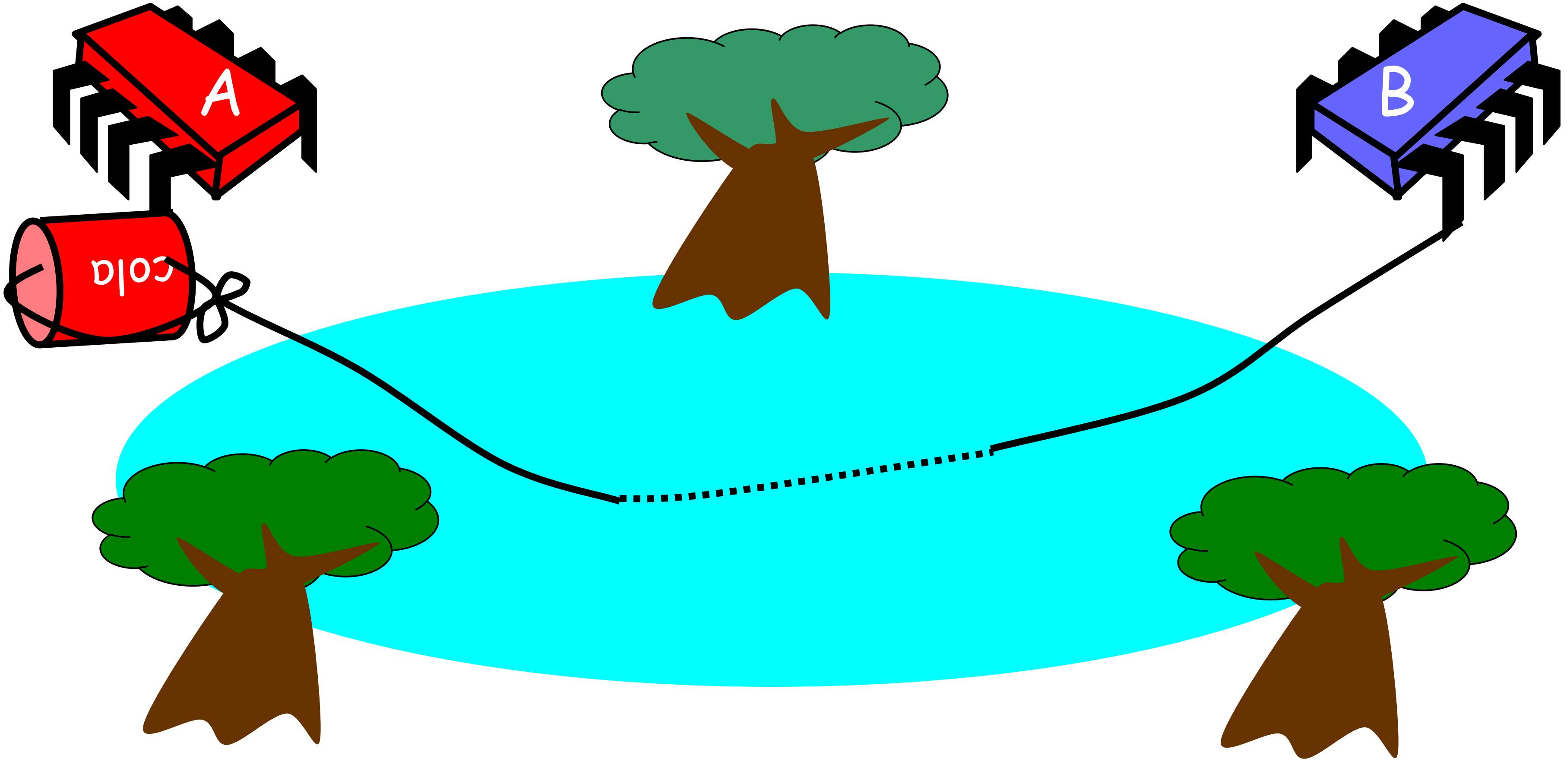
Can Protocol

- 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



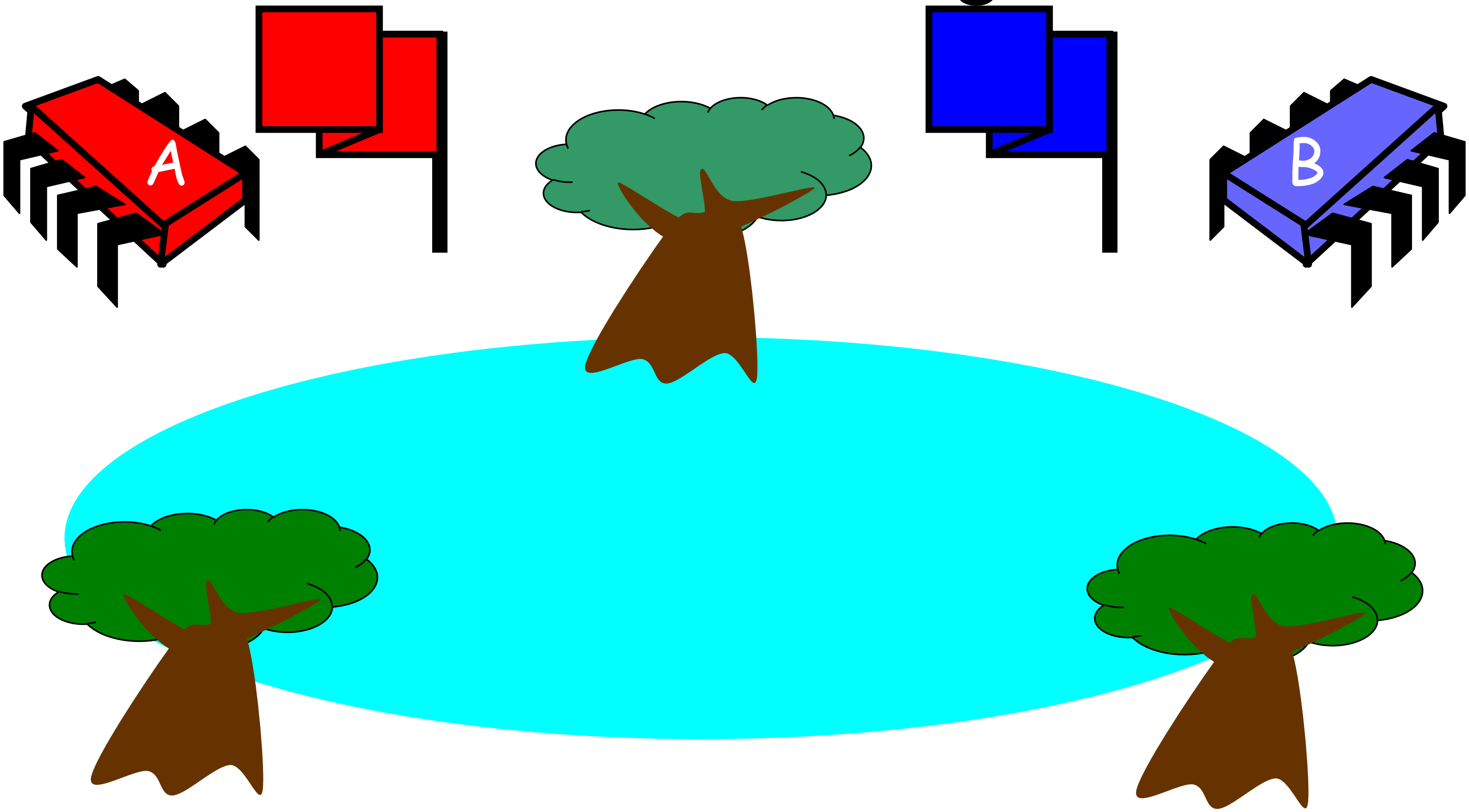
Can Protocol

- 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

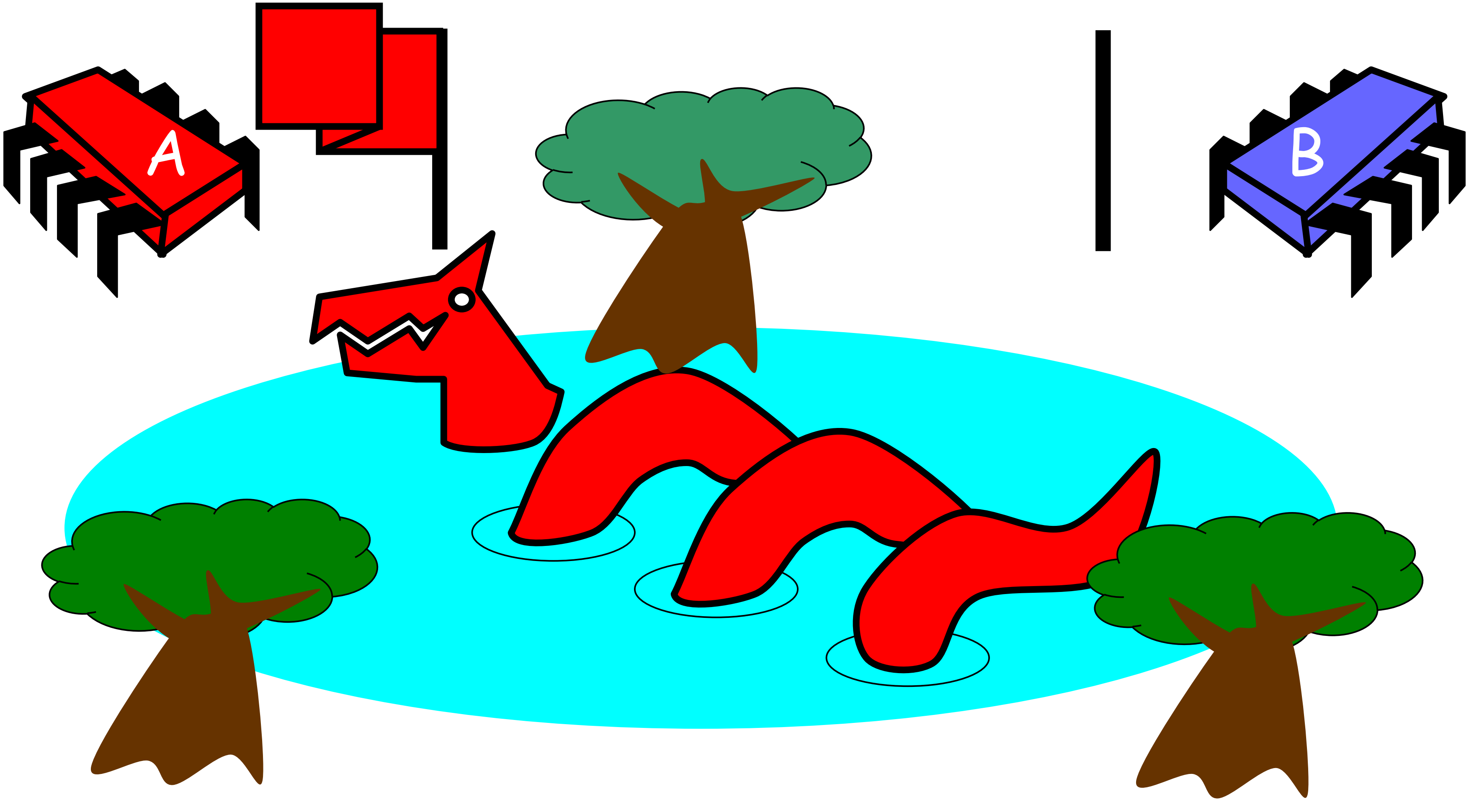
Possible solution: Flag Protocol



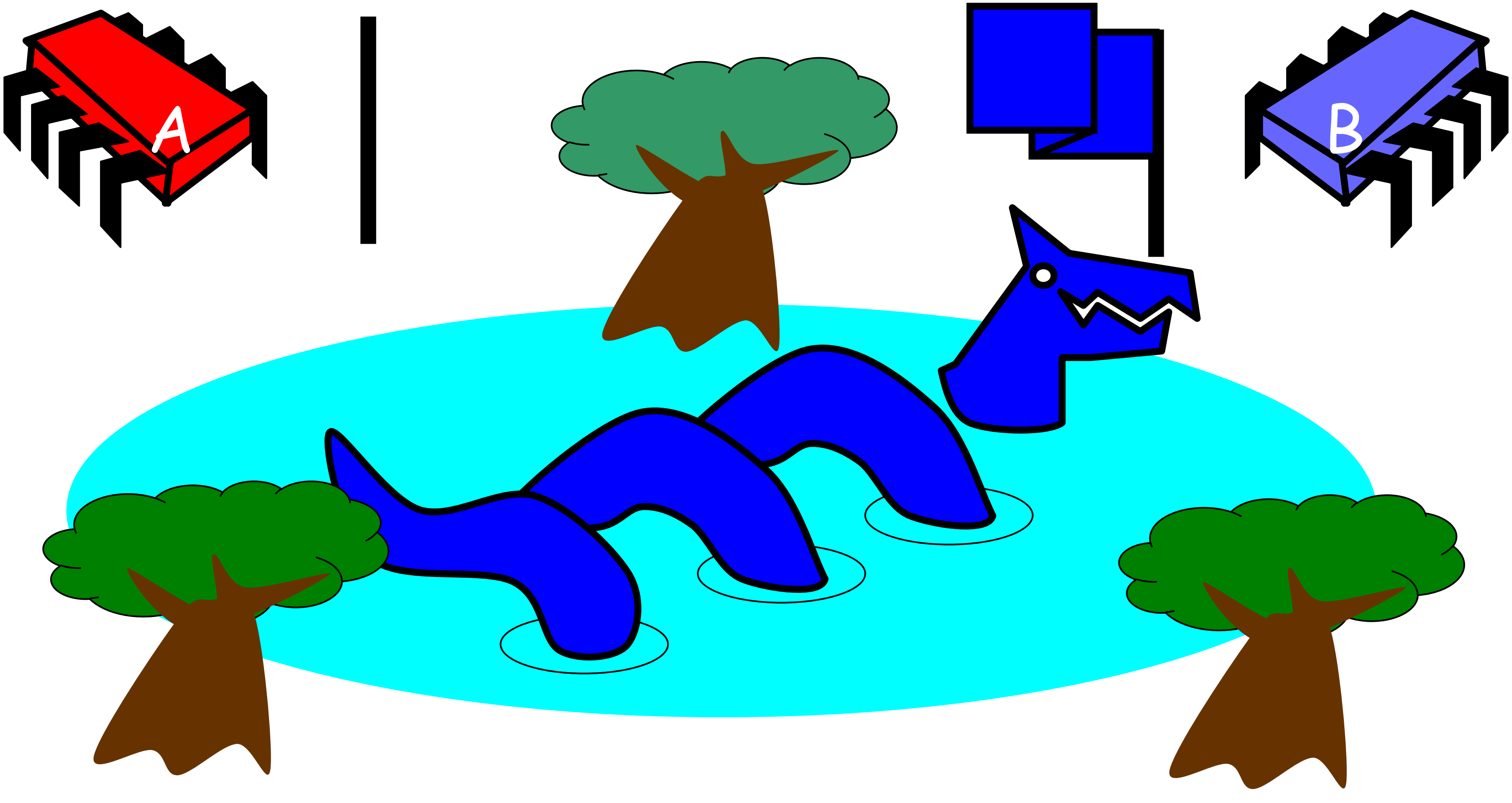
Alice’s Protocol

- 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

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

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

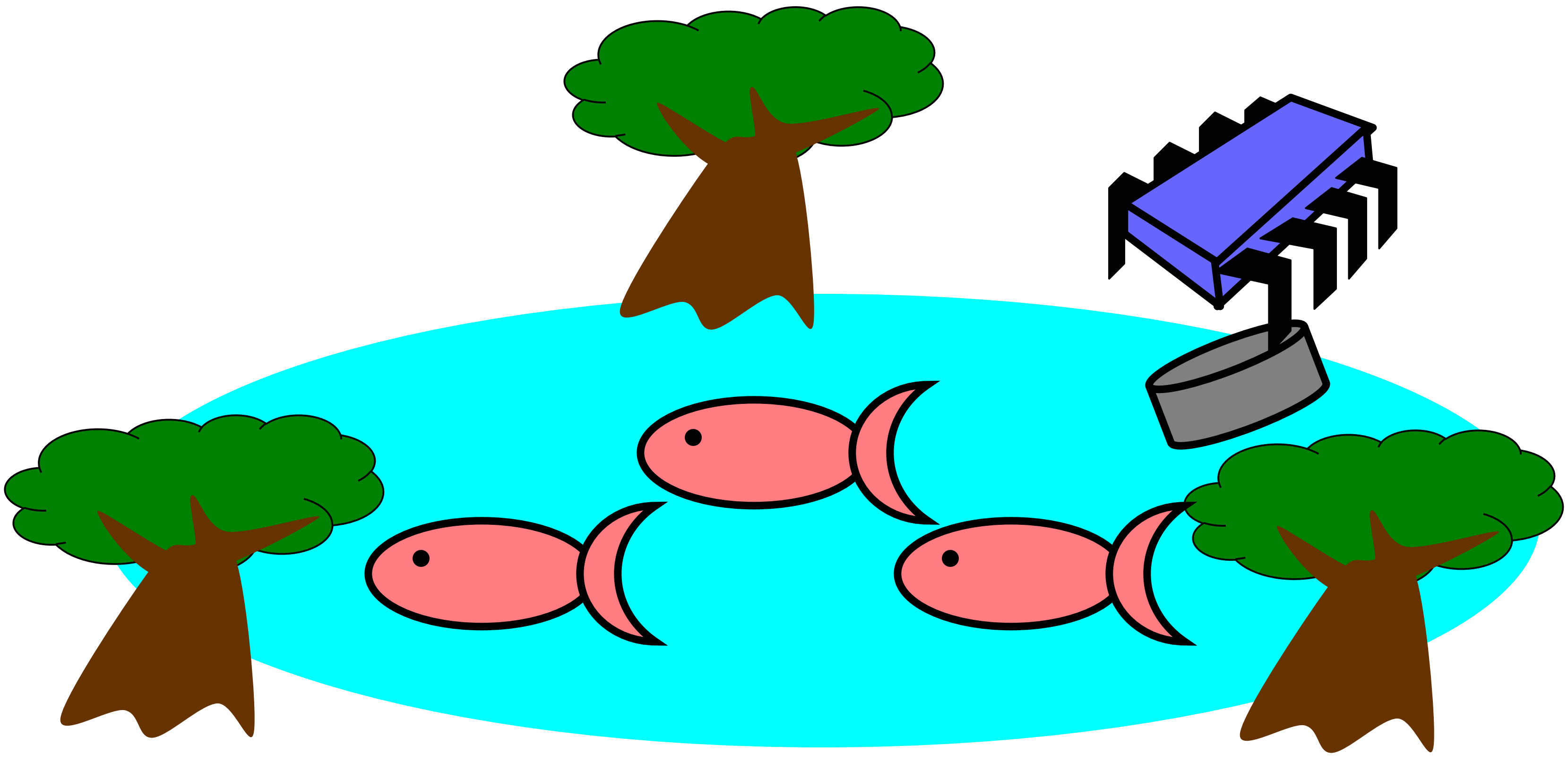
The Fable Continues

- 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

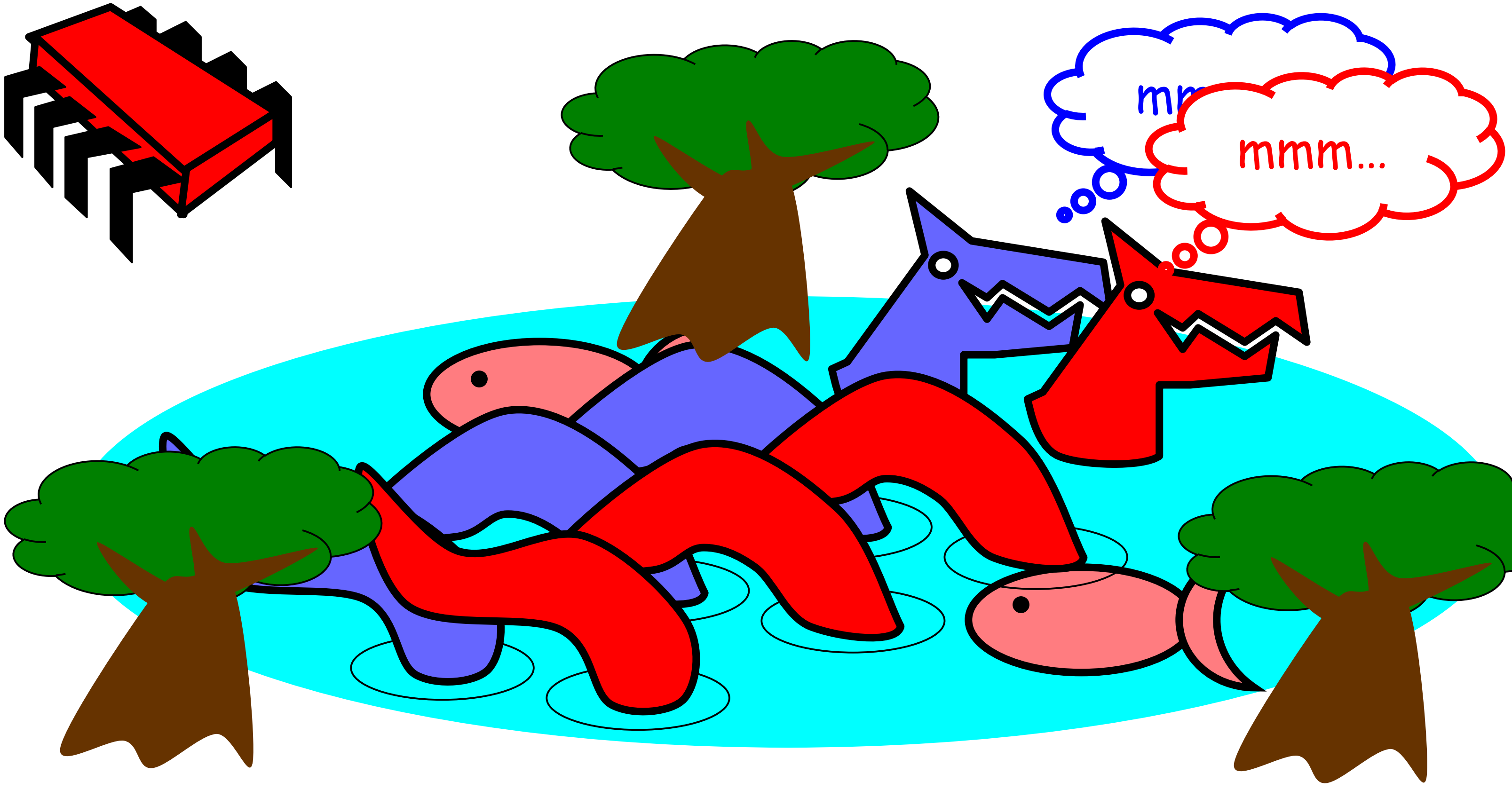
Producer-Consumer Problem

- 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

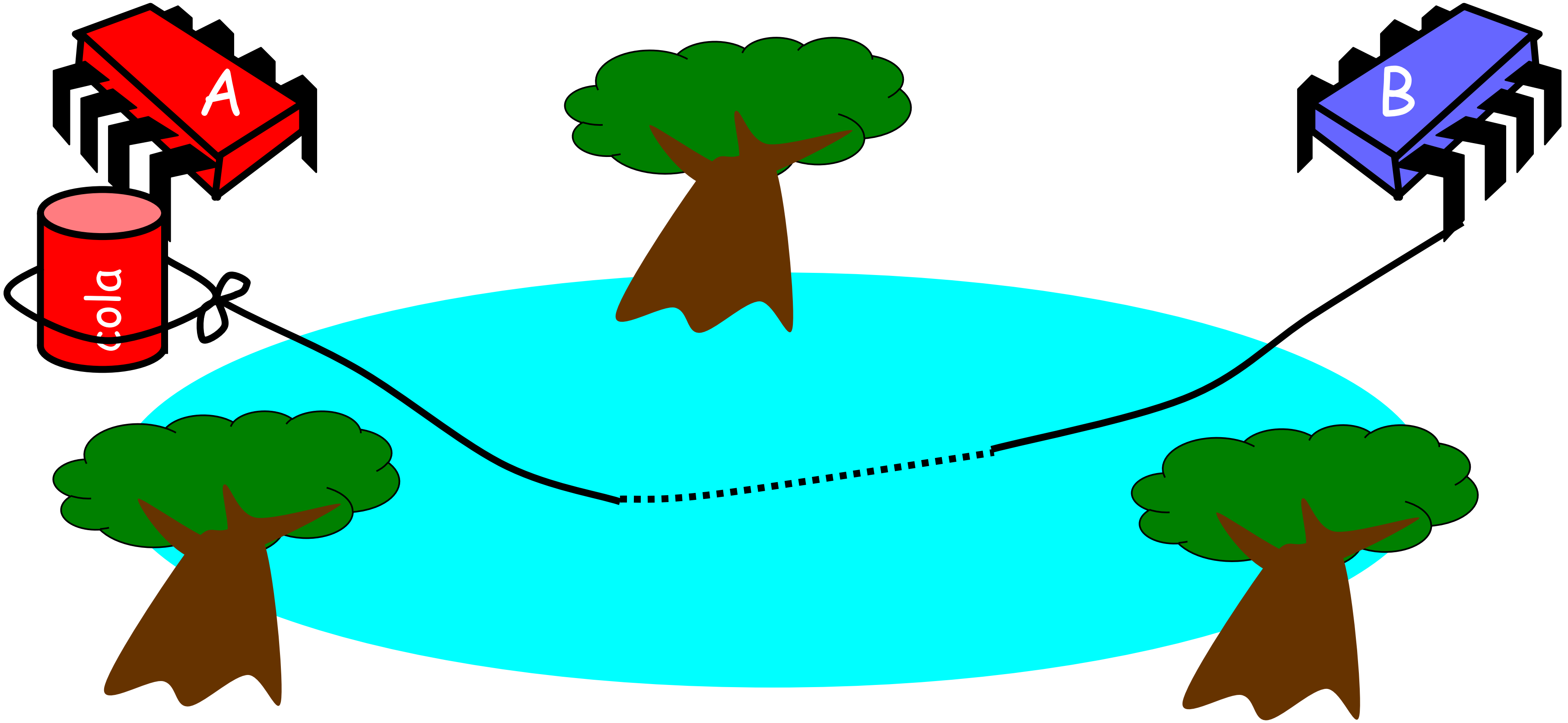
Producer/Consumer

- 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

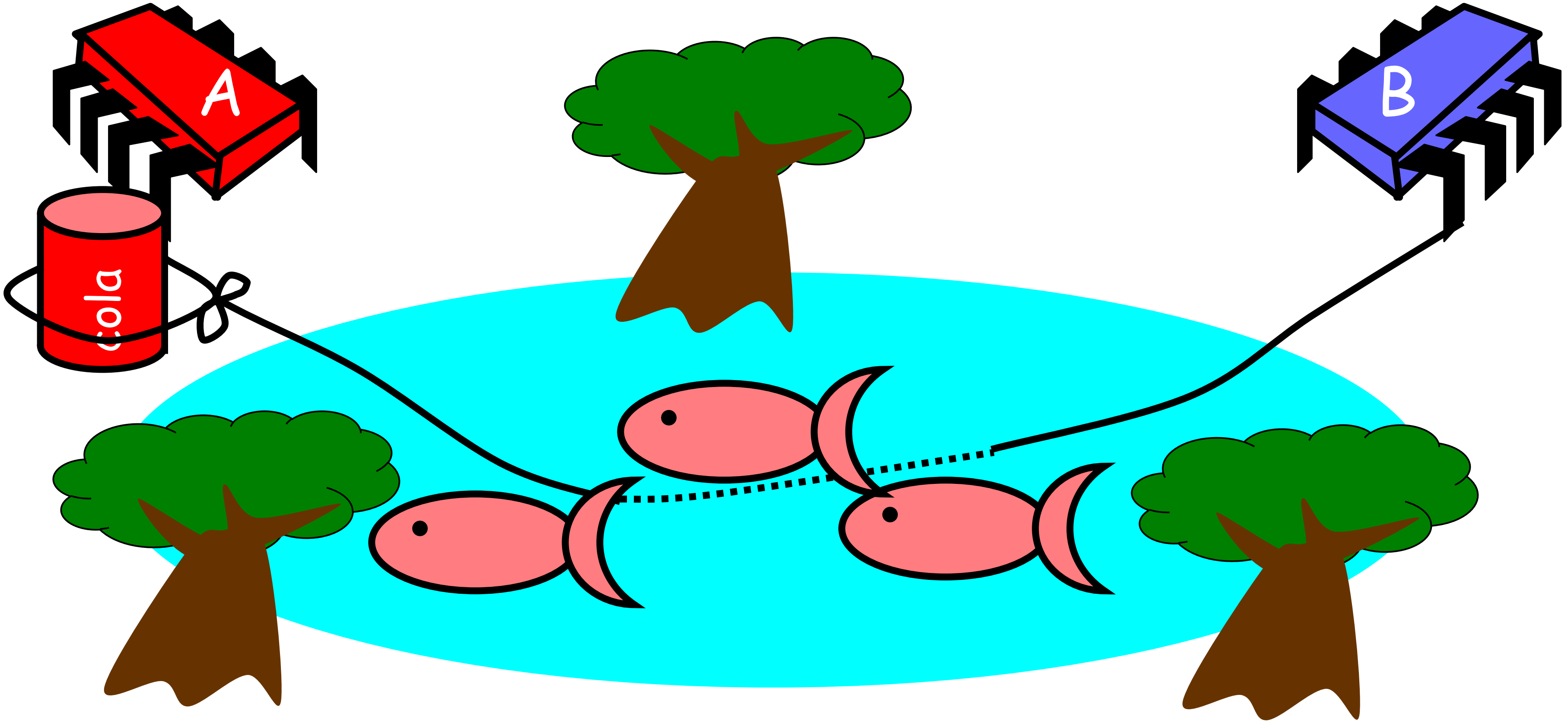
Also known as bounded buffer problem

- 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

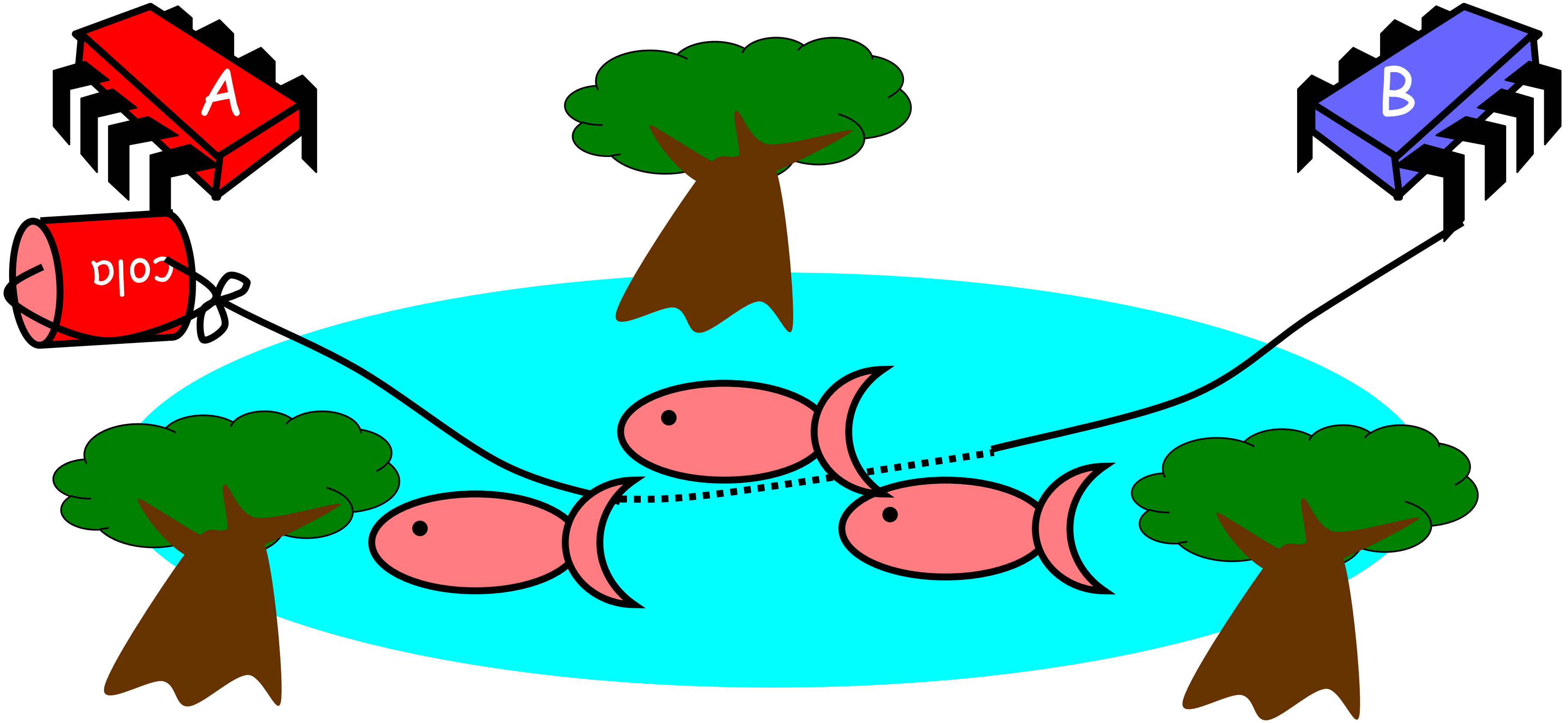
Surprise Solution



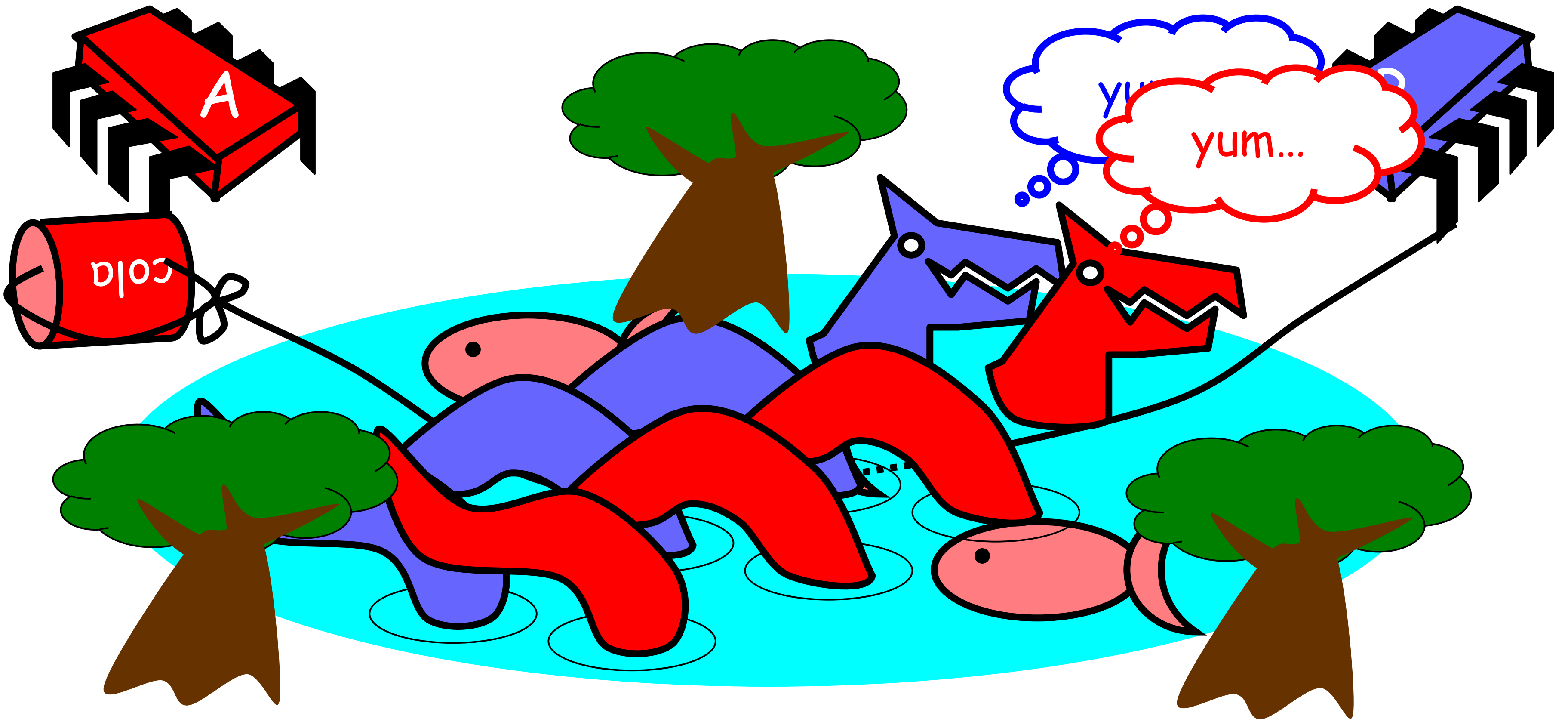
Bob puts food in Pond



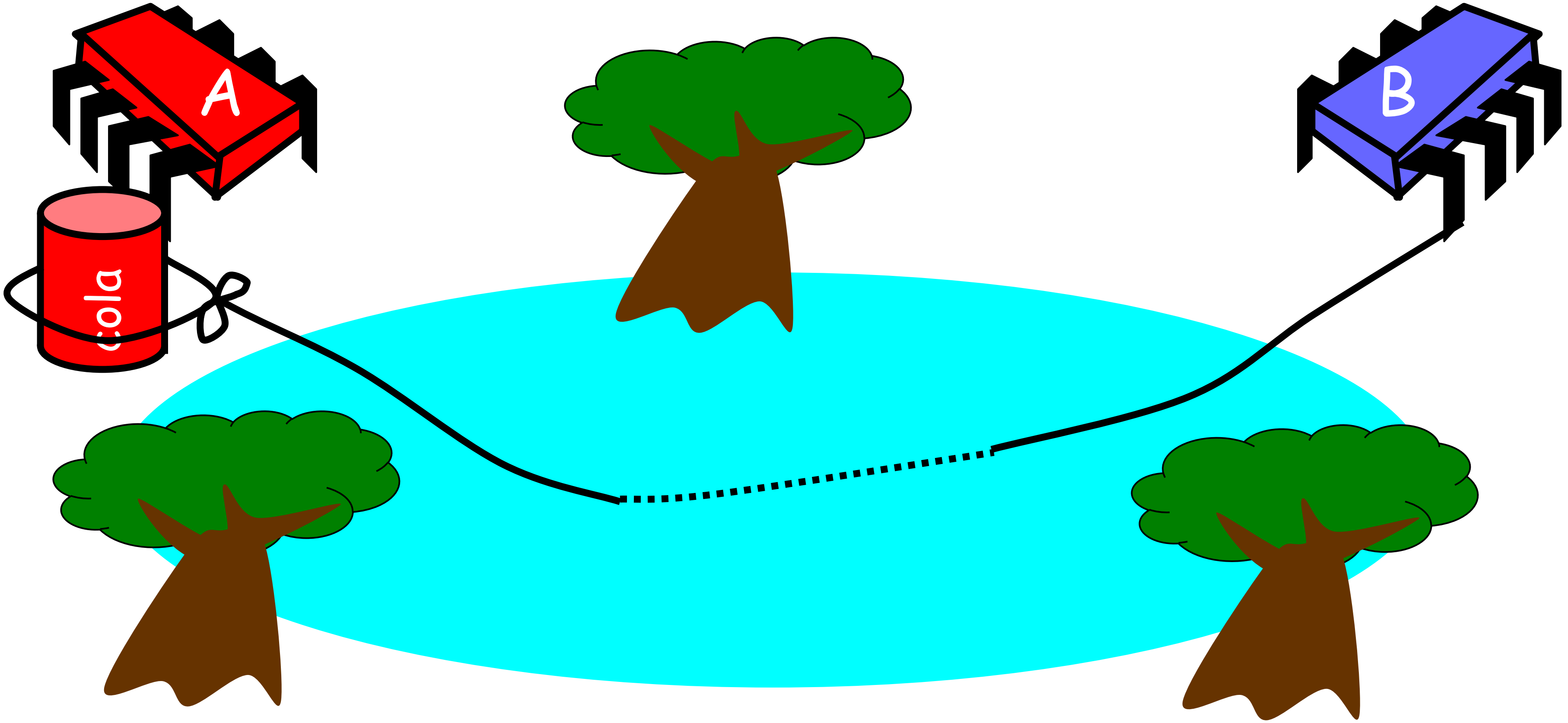
Bob knocks over Can



Alice Releases Pets



Alice Resets Can when Pets are Fed



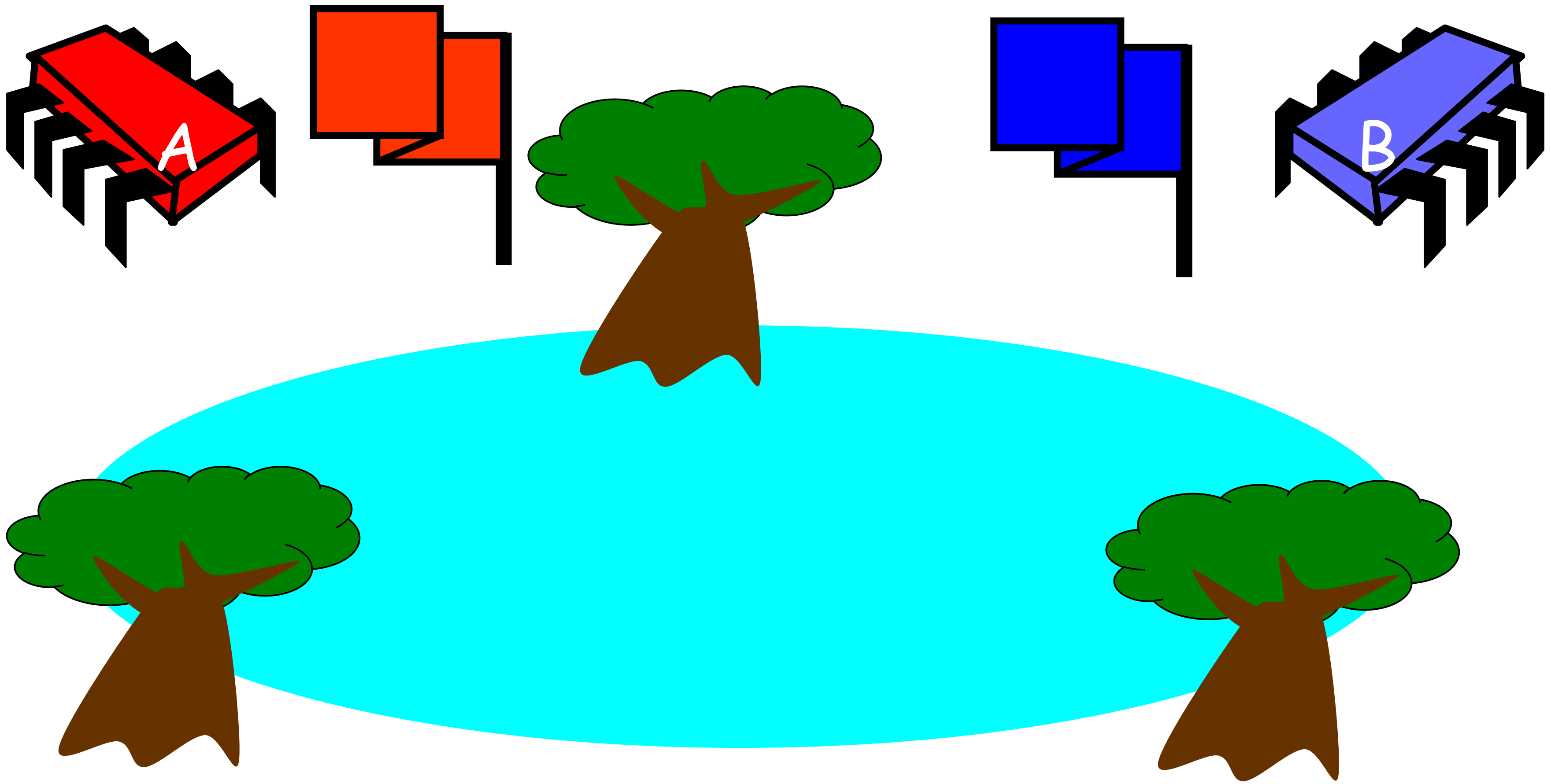
Correctness

- 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



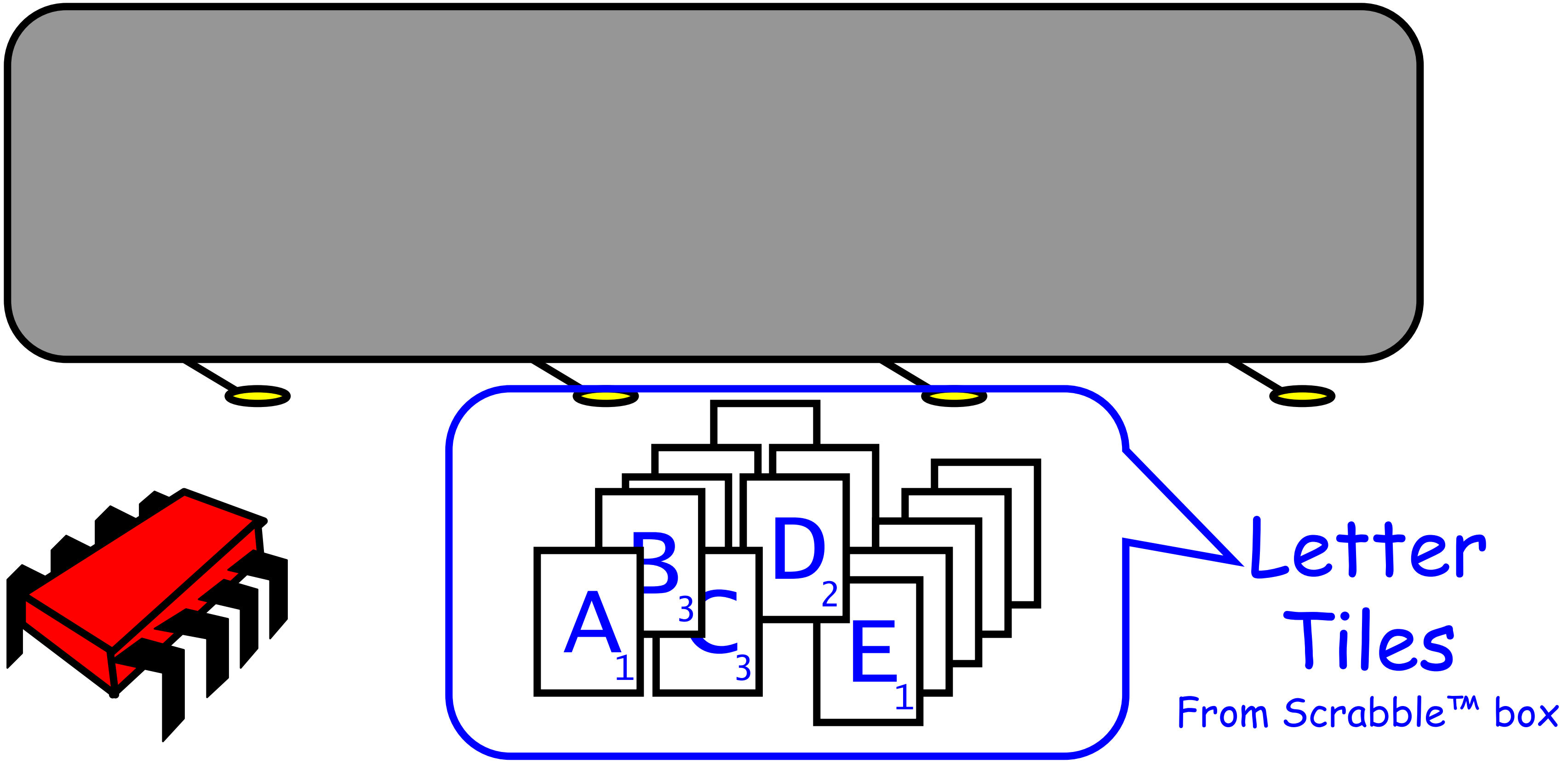
Waiting

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

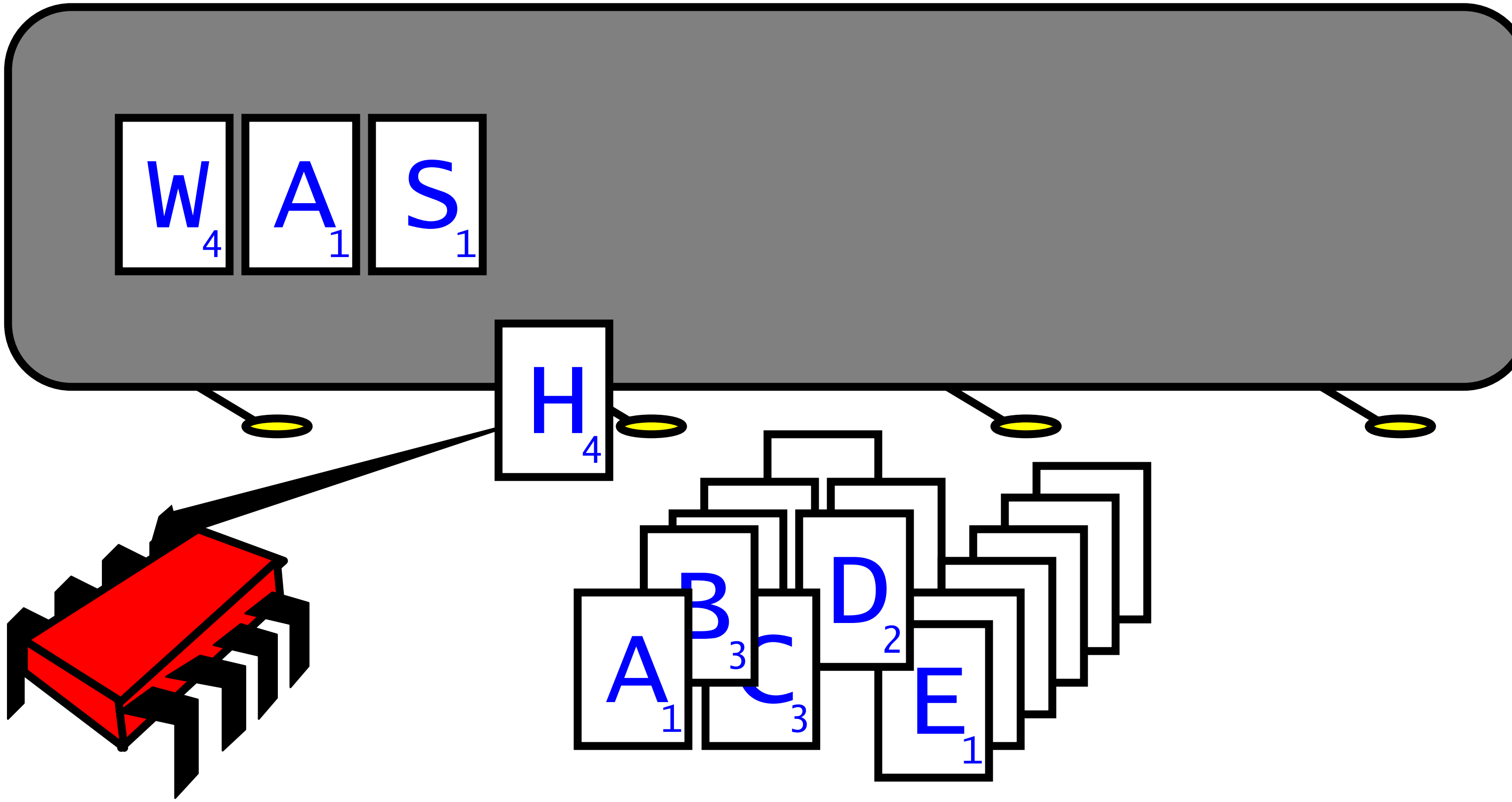
The Fable drags on …

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

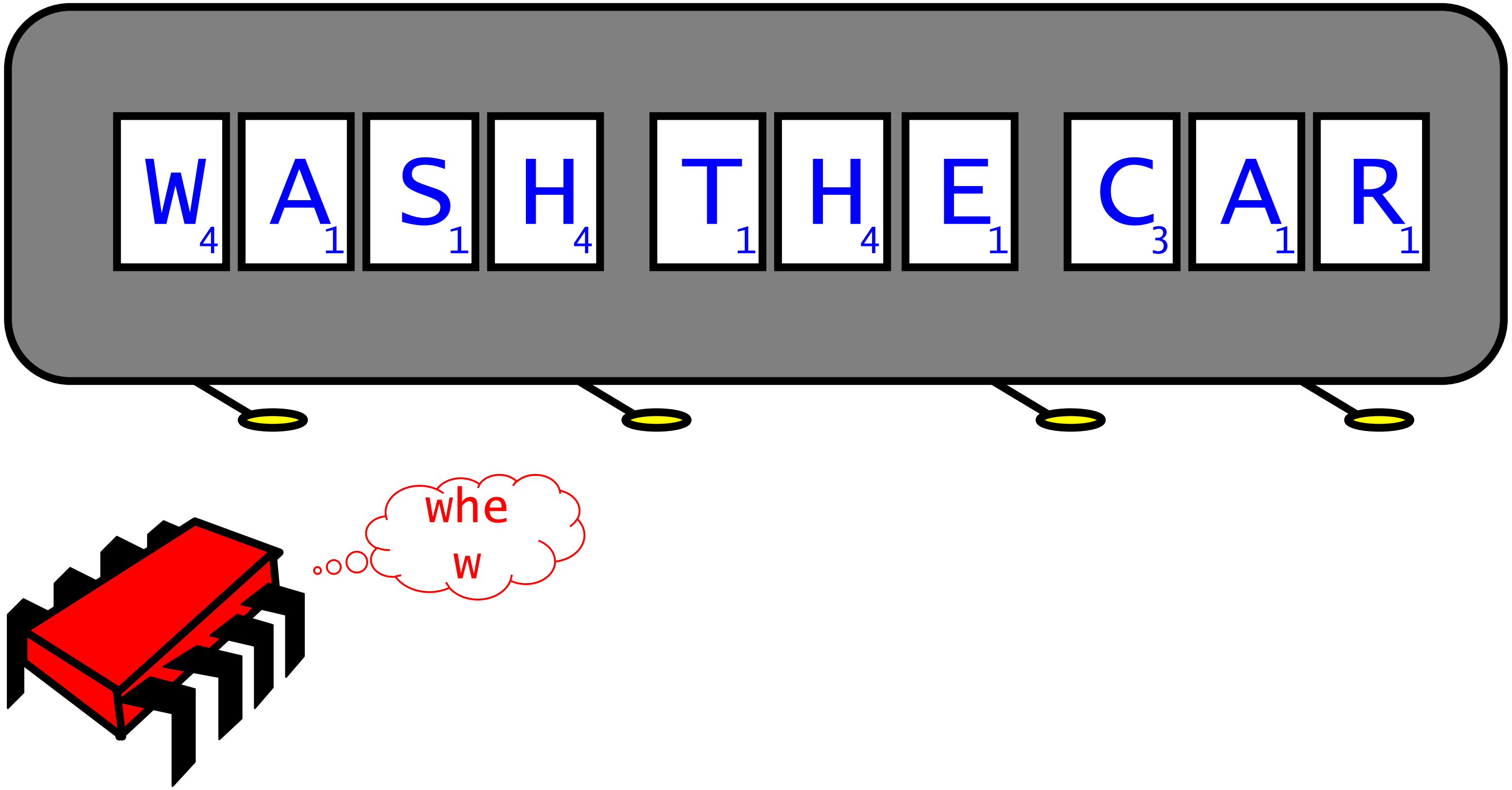
Billboards are Large



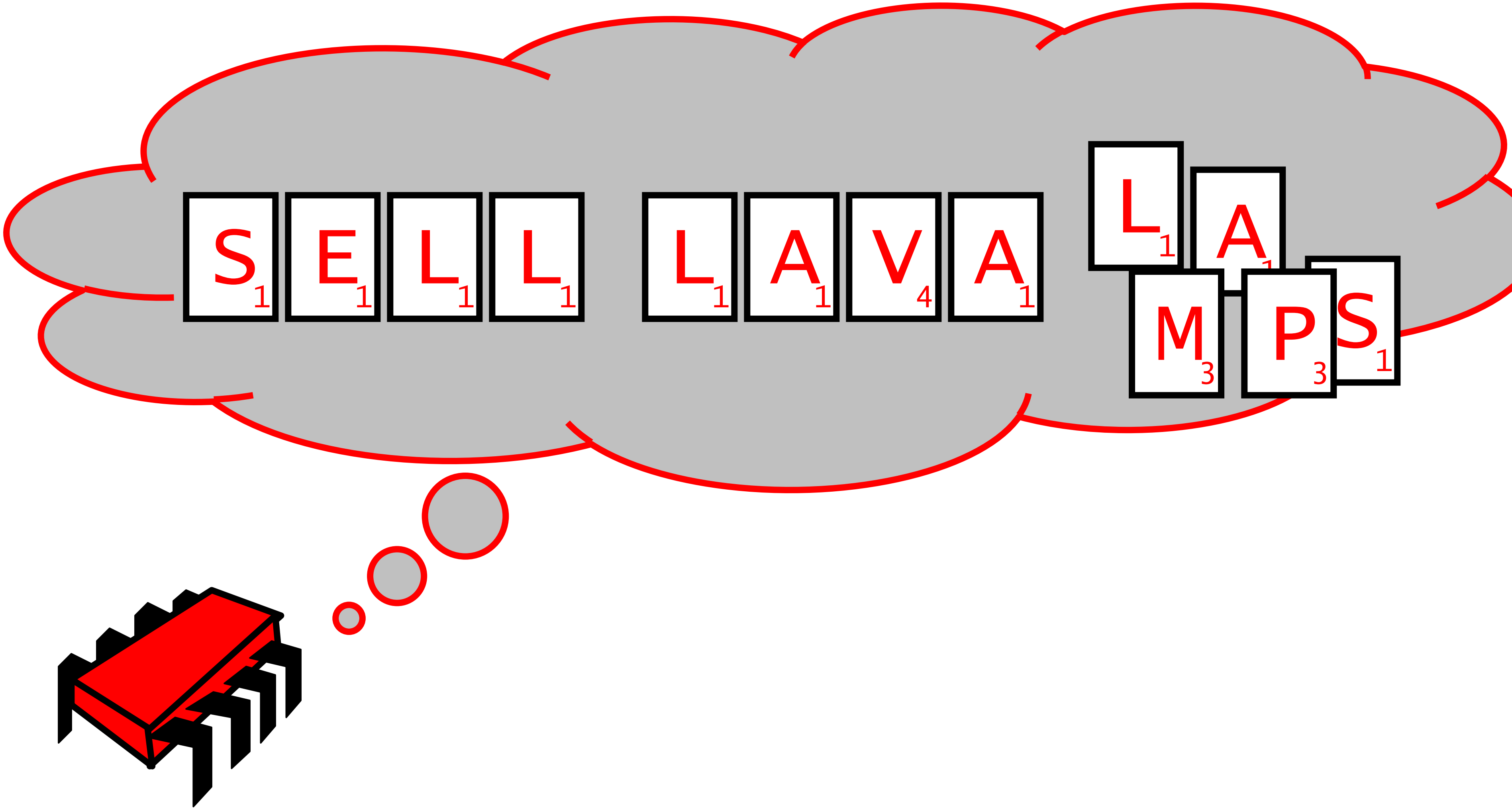
Write One Letter at a Time …



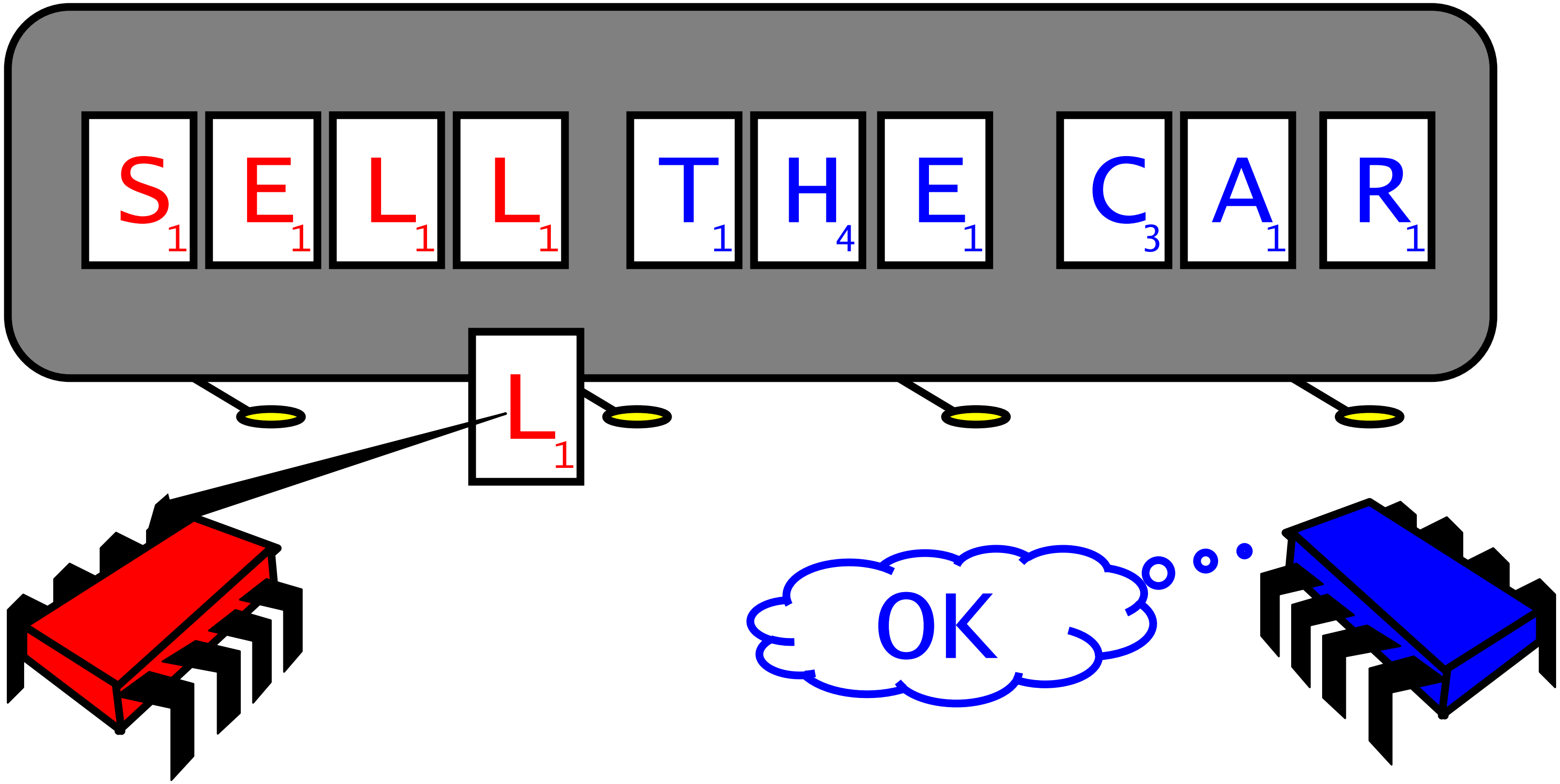
To post a message



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

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

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