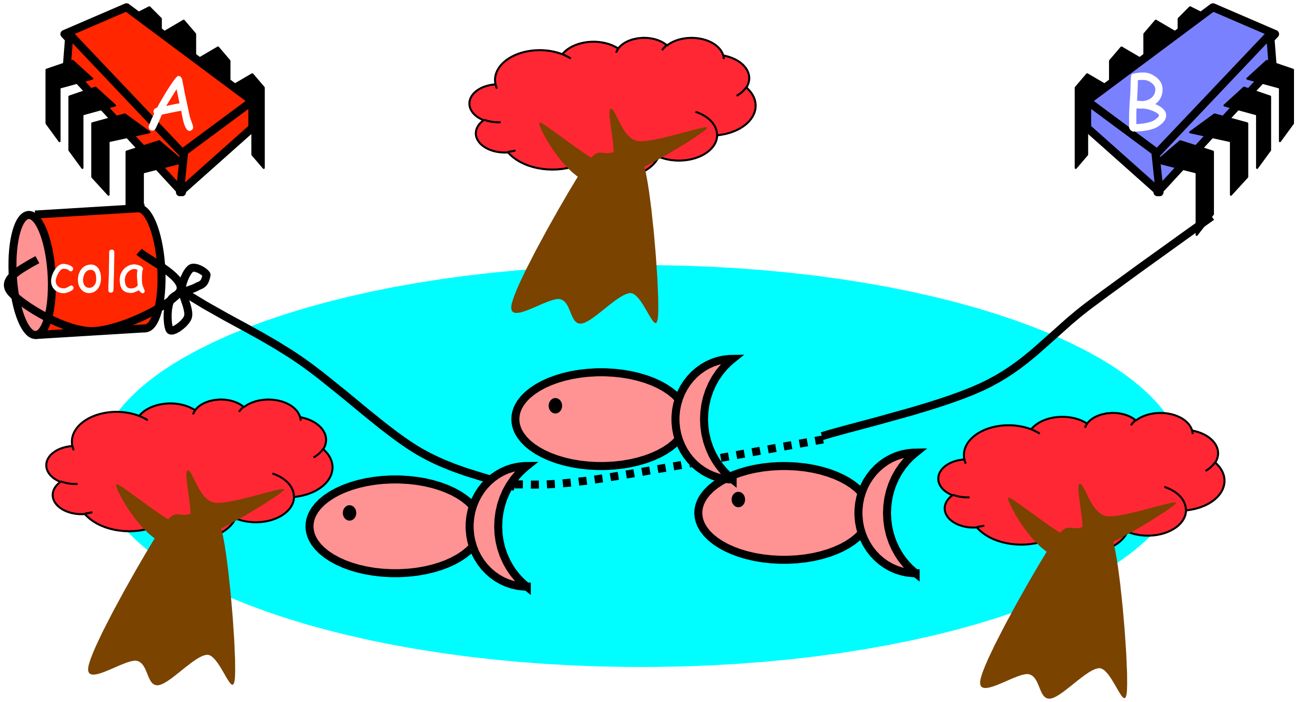
CS-206 Concurrency

Lecture 4
Mutual Exclusion



Spring 2015
Prof. Babak Falsafi
parsa.epfl.ch/courses/cs206/

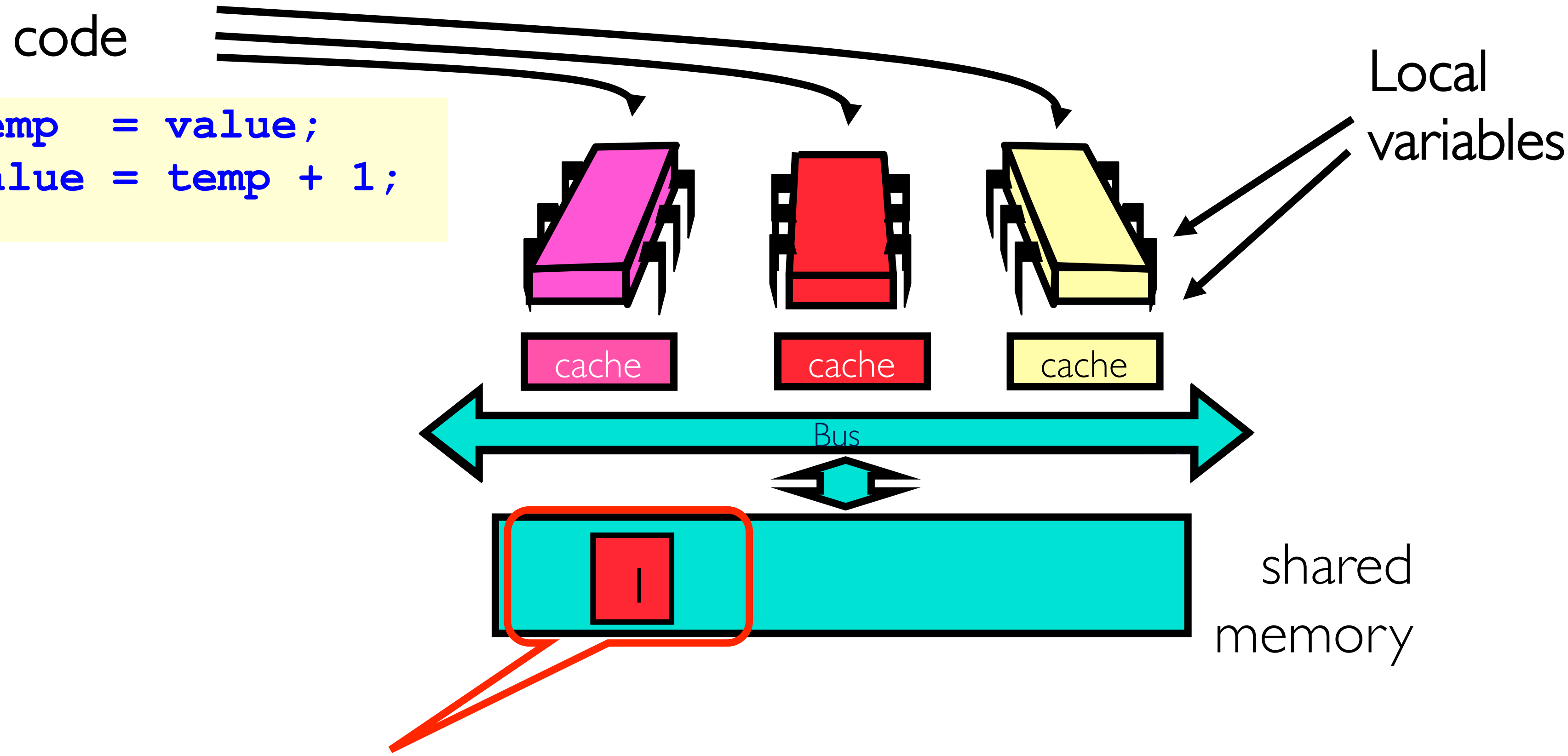
Where are We?

M	T	W (Lecture & Lab)	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar
9-Mar	[illegible]	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May	5-May	6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

- Mutual Exclusion in practice
  - How do we share?
  - How do we share well?
    - Protocols
    - Properties
- Next week
  - Mutual Exclusion 2
    - Formal definiton

Recall: Counter in Shared Memory

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

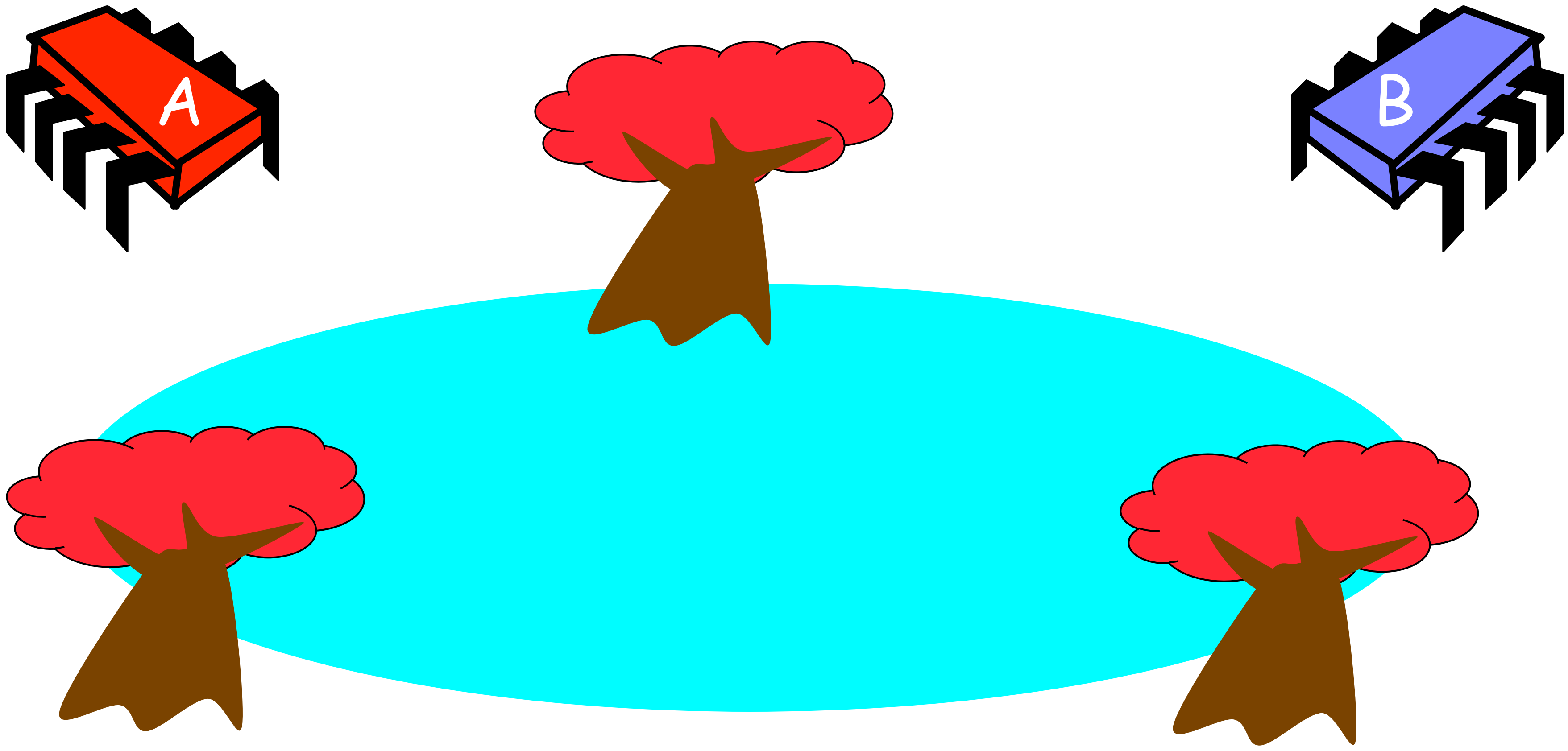


Need Mutual Exclusion

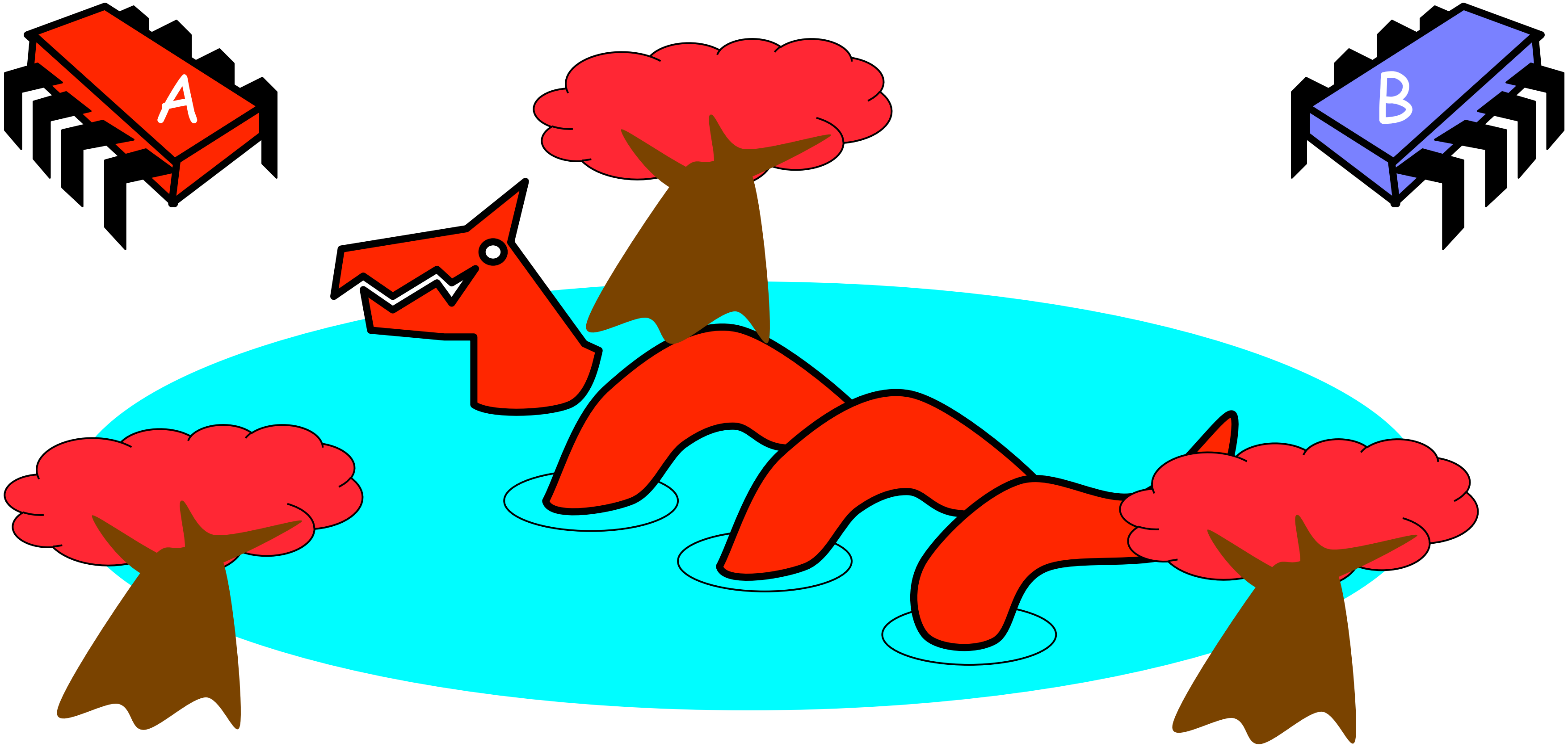
Sharing requires coordination

- Illustrated through a fable
- Our Fable
  - Alice & Bob have pets: a dog and a cat
  - The pets like to play in a pond
  - But, they don't get a long
  - So, need a protocol to share the pond
    - Agree on guidelines as to how to share
  - The protocol would guarantee mutual exclusion

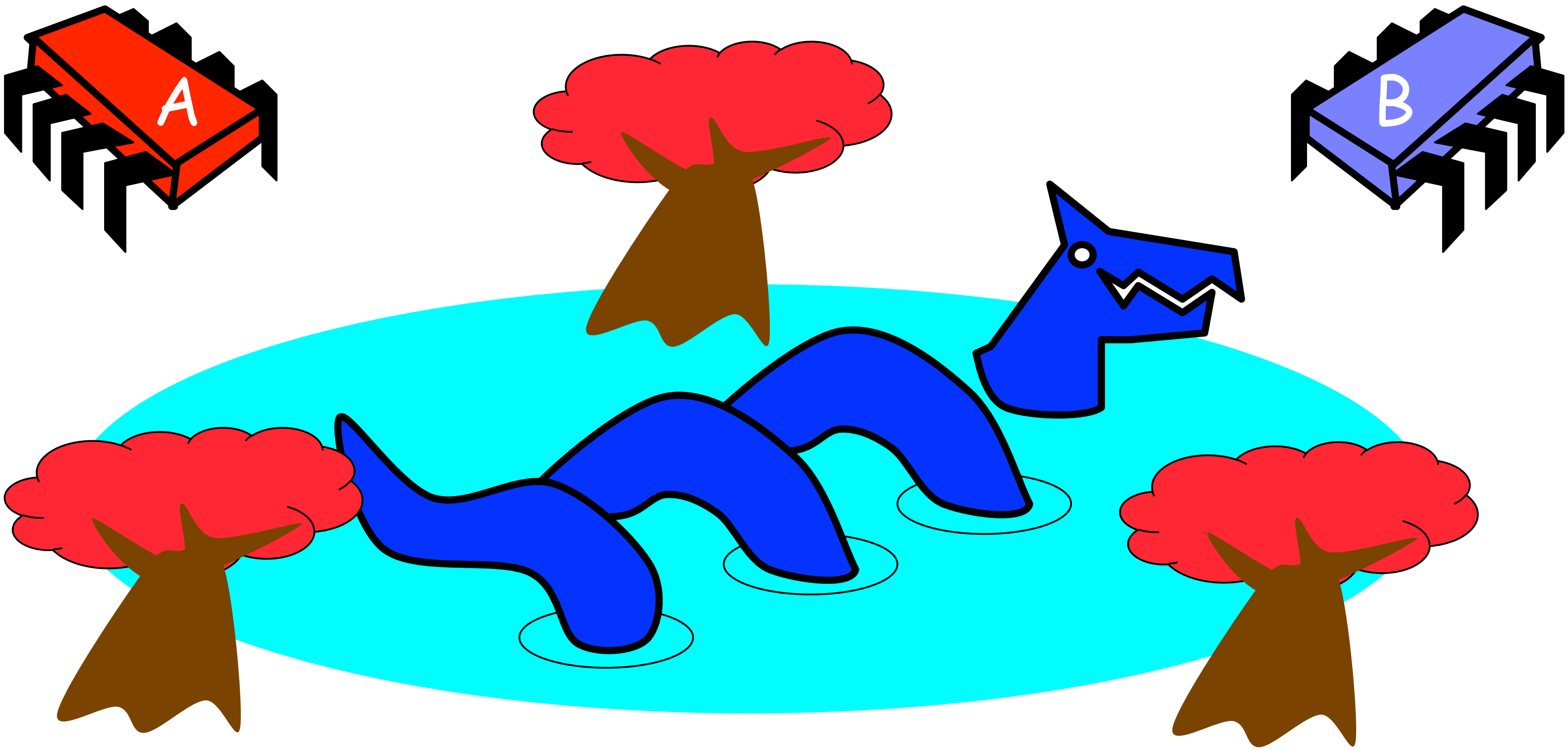
Mutual Exclusion, or "Alice & Bob share a pond"



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 (means "potential problems")
  - 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
  - Alice calls Bob (or vice-versa)
- Gotcha
  - Bob takes shower
  - Alice may need to recharge battery
  - Bob out of cell phone tower reach…

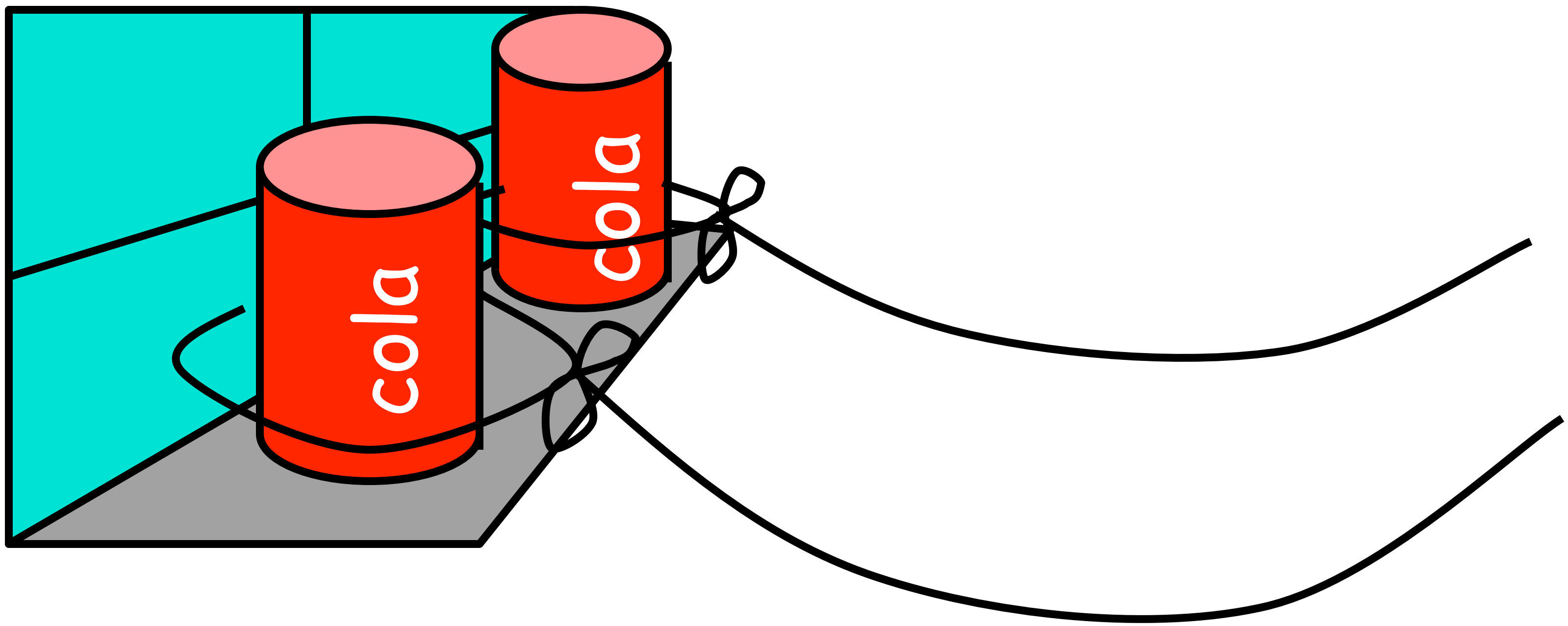
Interpretation

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

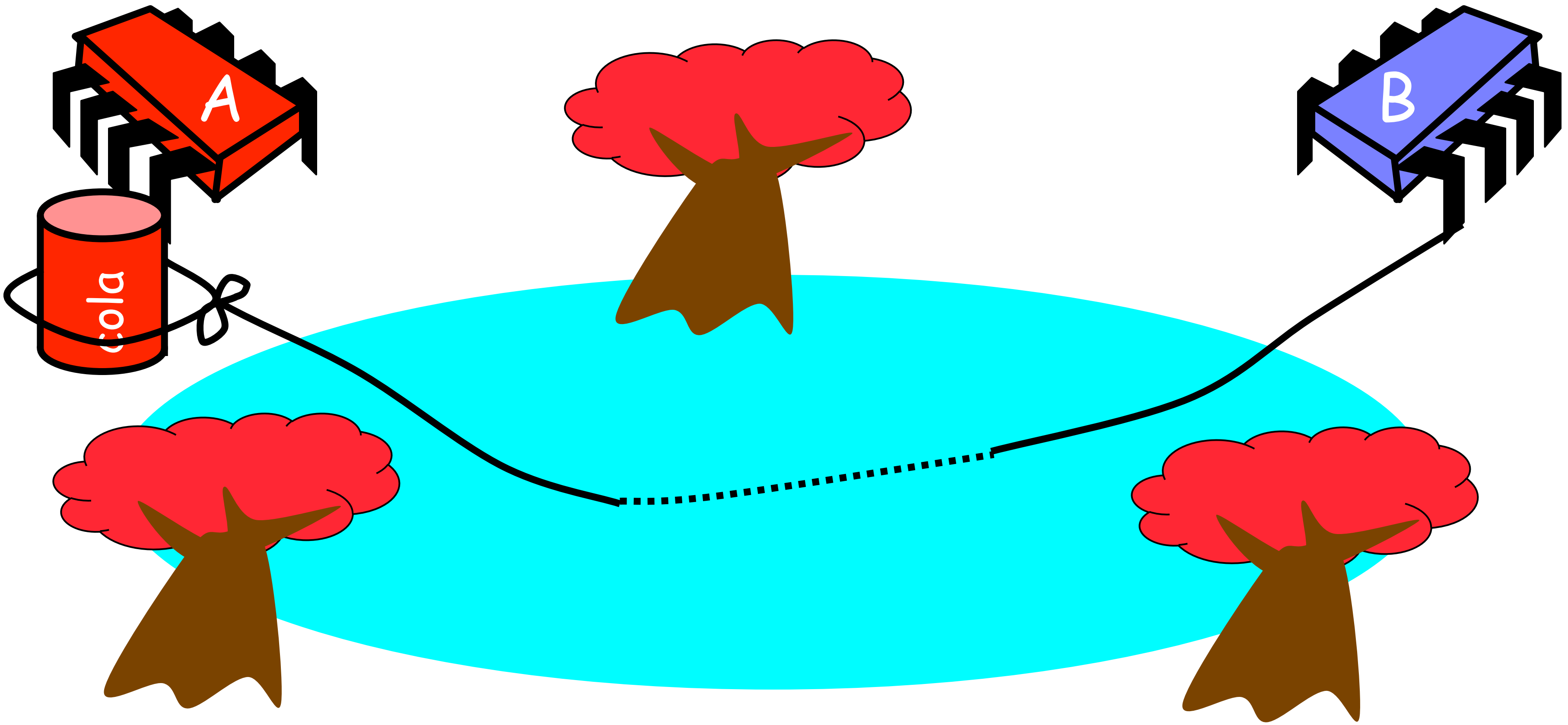
Alice comes up with a clever solution

- Beer/Cola cans!
  - Used to send a signal
  - Sort of like conveying a "ready" bit
  - Fallen can = ready, upright can = not ready
- One or more beer cans by Alice's window (on windowsill)
- Each with string running to Bob's house
- To send a signal, Bob pulls string
- …and knock the can over

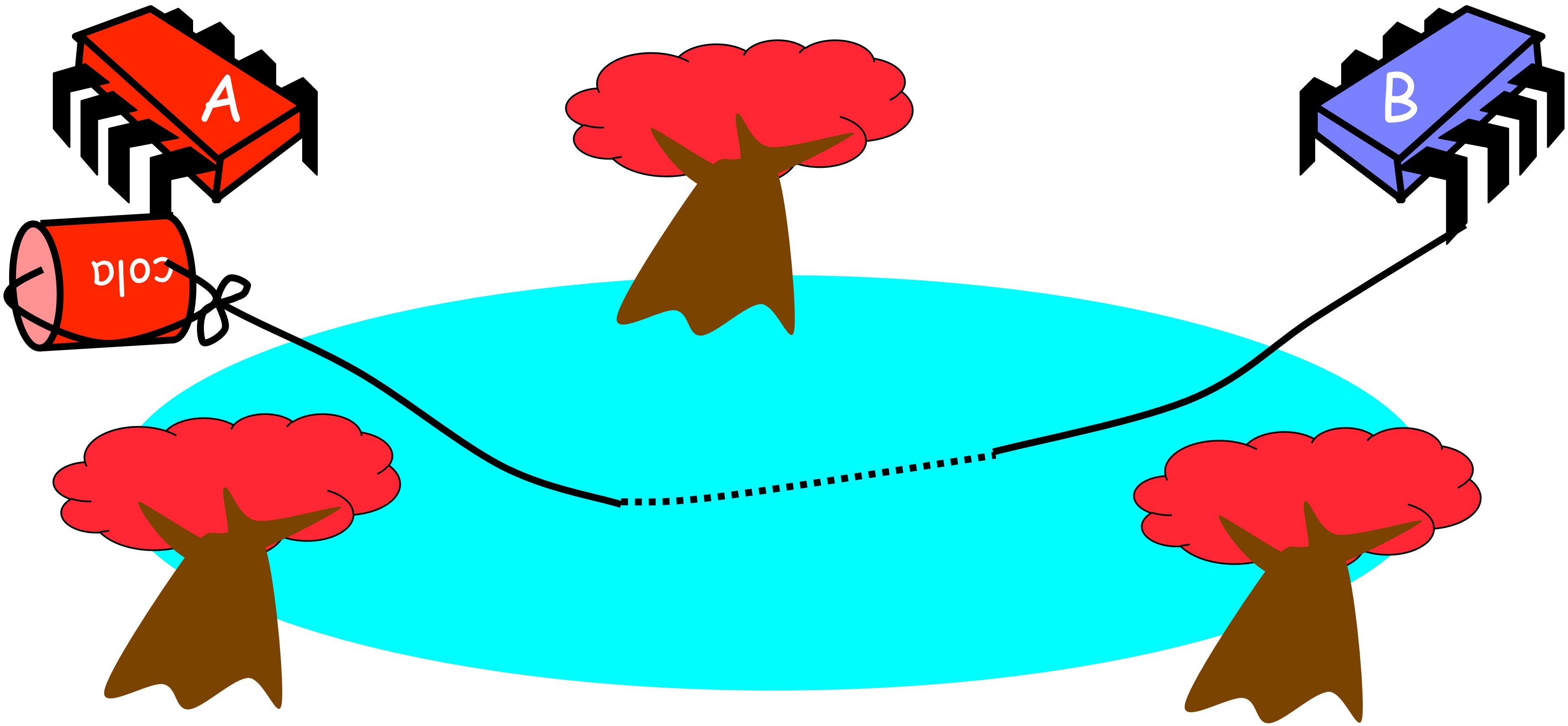
Can Protocol



Bob conveys a bit



Bob conveys a bit



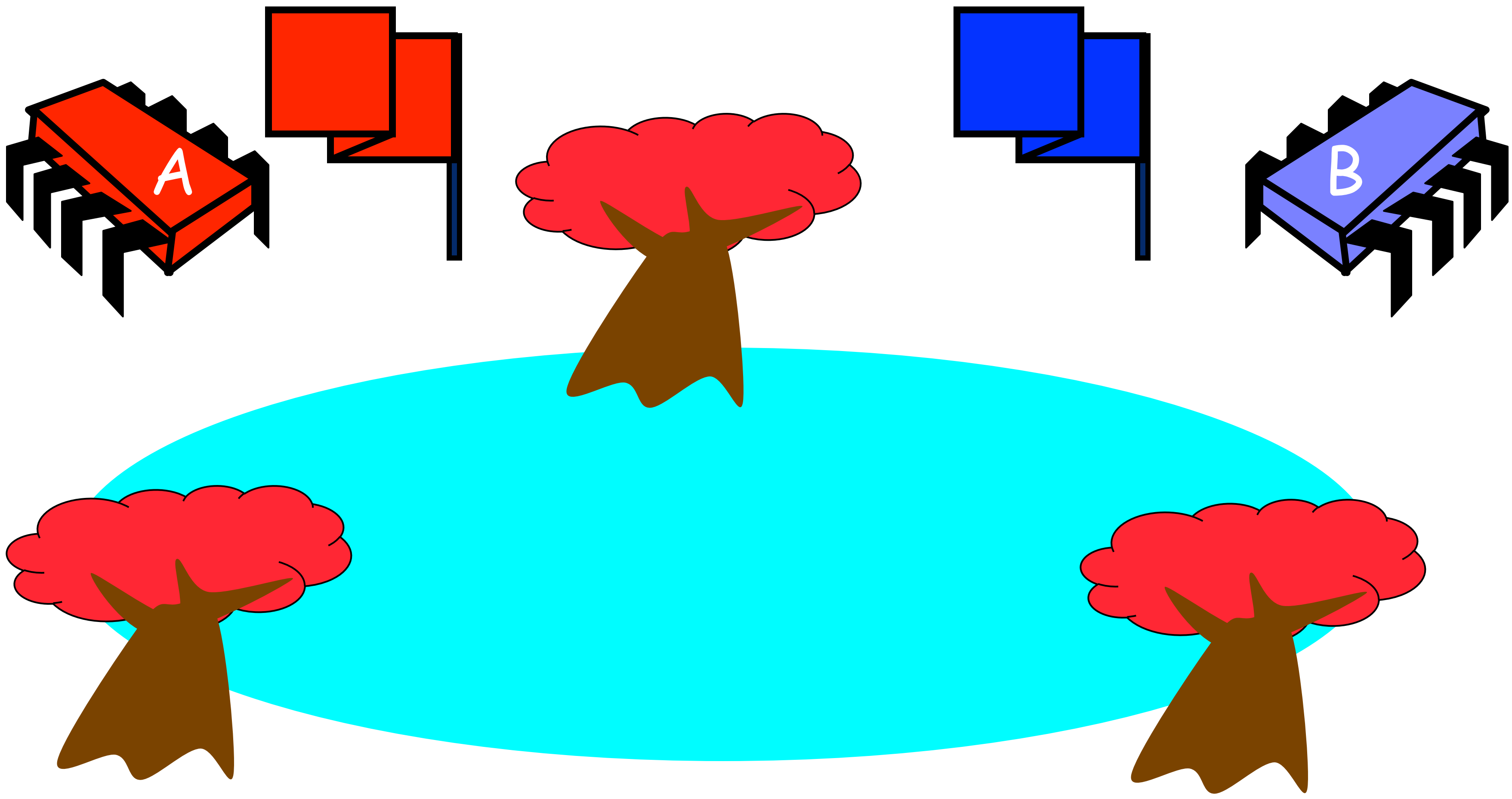
Can Protocol

- Idea
  - ▷ Cans by Alice's window
  - ▷ Strings lead to Bob's house
  - ▷ Bob pulls strings, knocks over cans
- Gotcha
  - ▷ Cans cannot be reused
  - ▷ Bob runs out of cans
  - ▷ What if Alice takes a long time to reset (goes on vacation)?

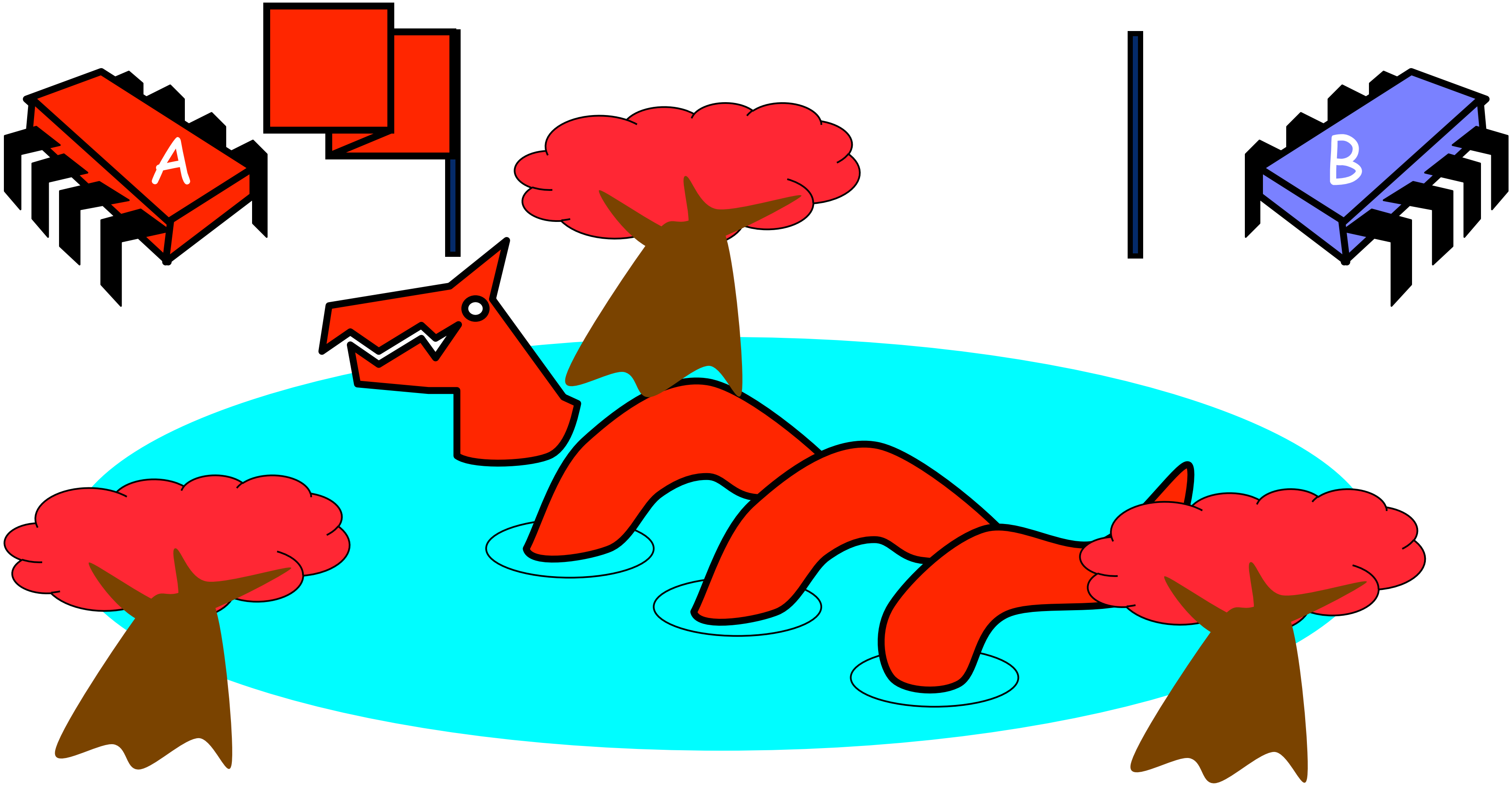
Interpretation

- Sending a signal (interrupt to set a bit)
  - Does not solve mutual exclusion
  - Sender sets fixed bit in receiver's space
  - Receiver resets bit when ready
  - Requires unbounded number of interrupt bits

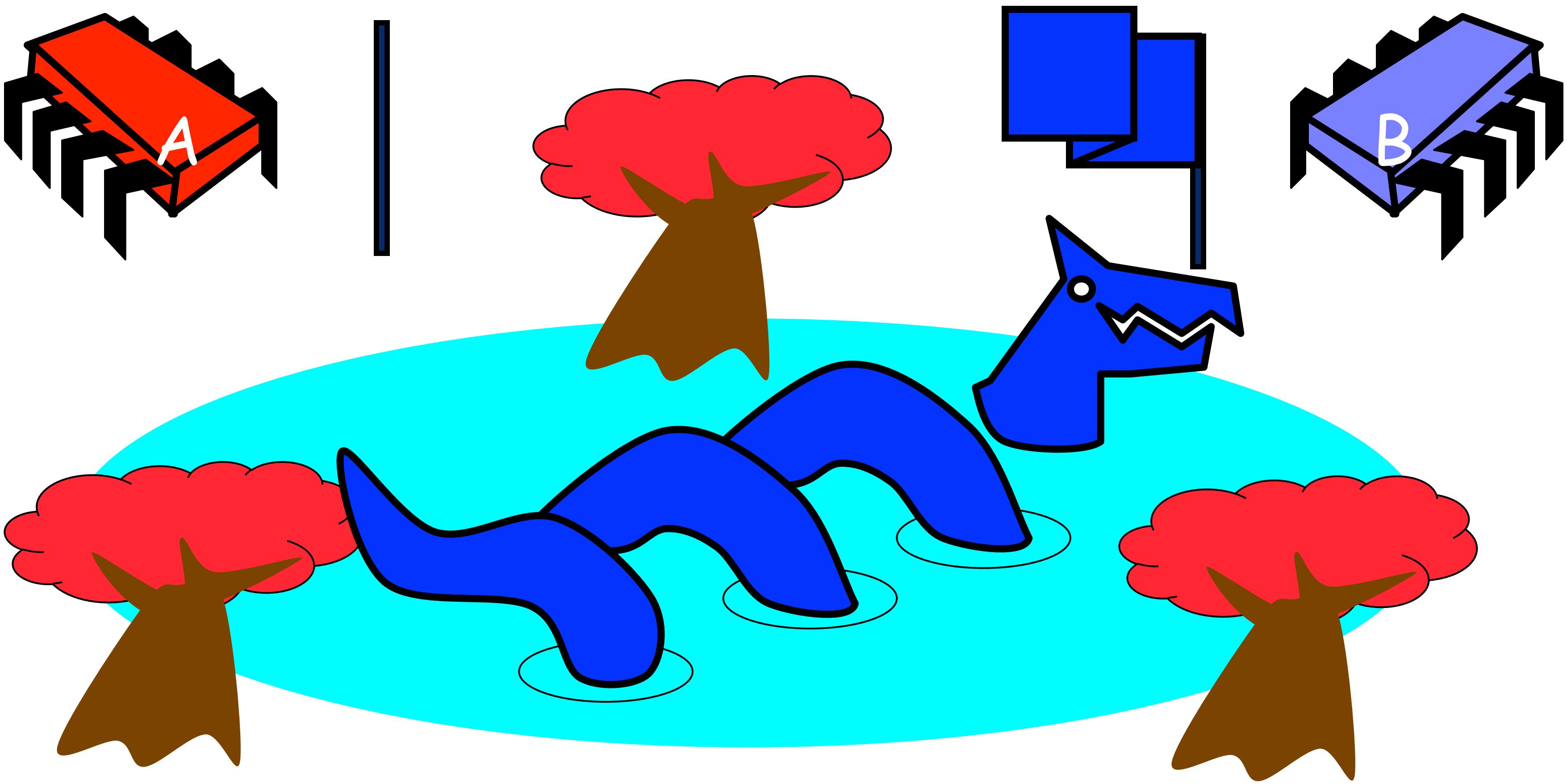
Flag Protocol



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

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

Bob defers to Alice

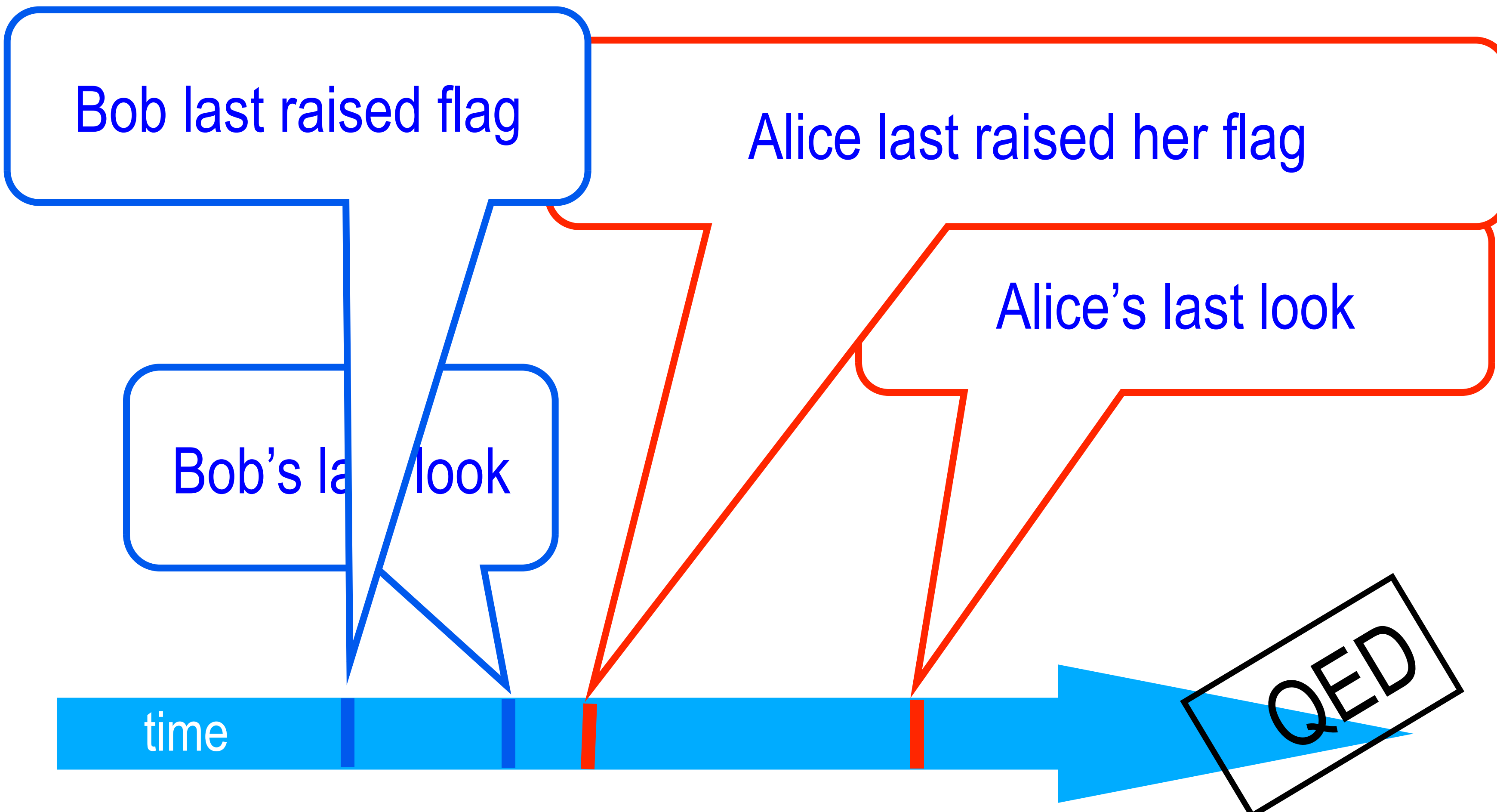
The Flag Principle

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

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
- 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

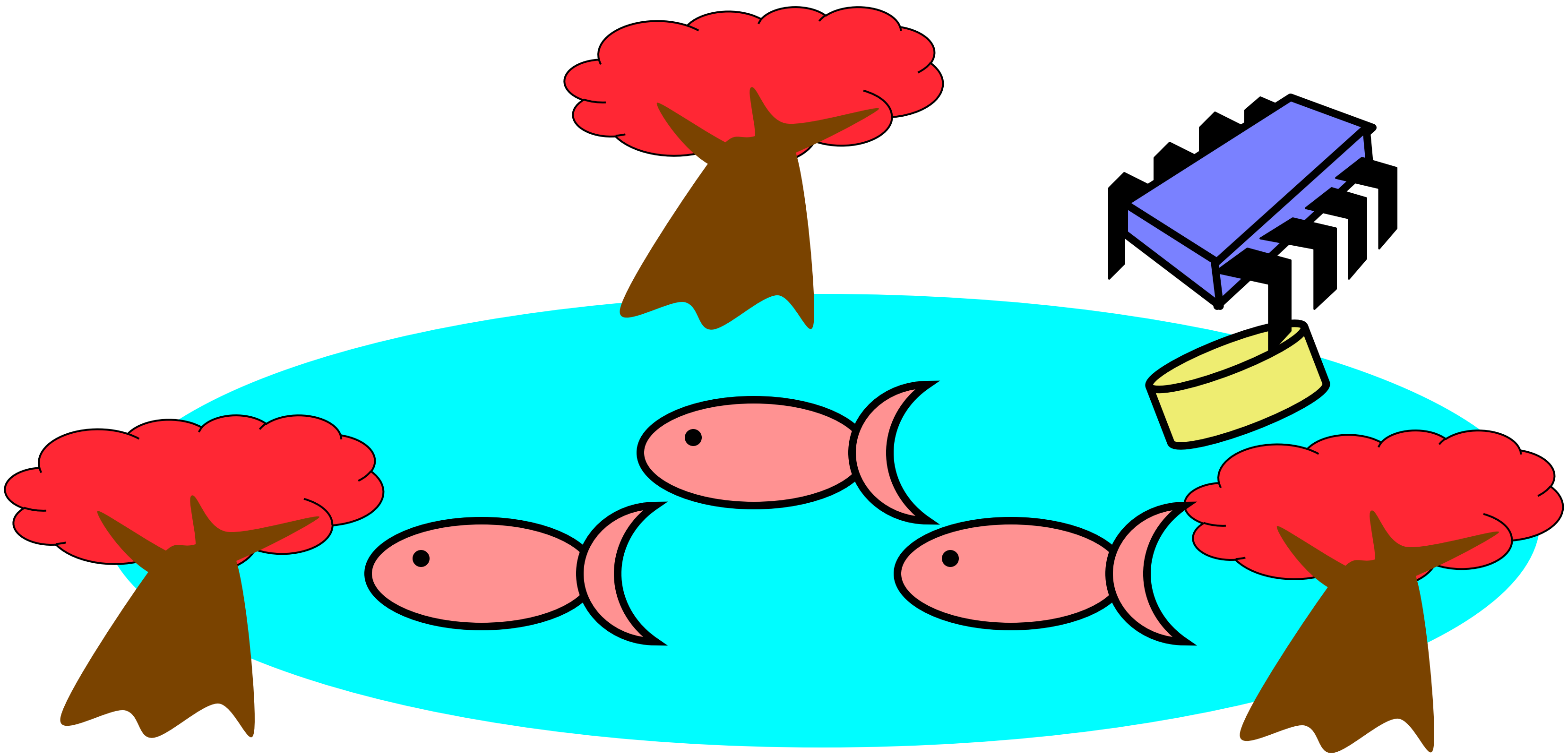
Moral of Story

- 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

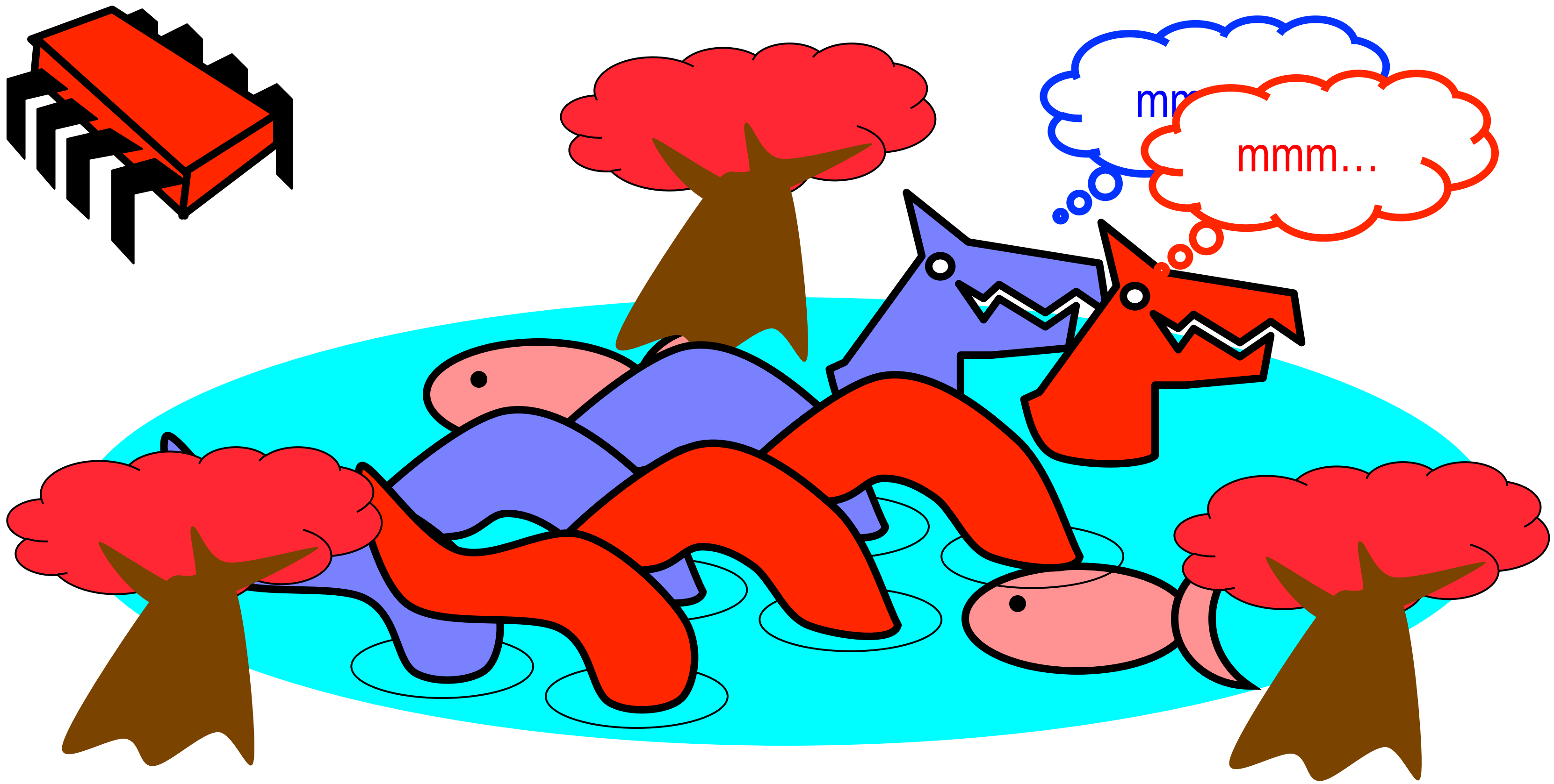
The Fable Continues

- 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
  - Used everywhere in parallel computing/distributed systems
  - E.g., network buffers (shared between processor & network)
  - E.g., OS scheduler (queues holding active threads)

Bob Puts Food in the Pond



Alice releases her pets to Feed



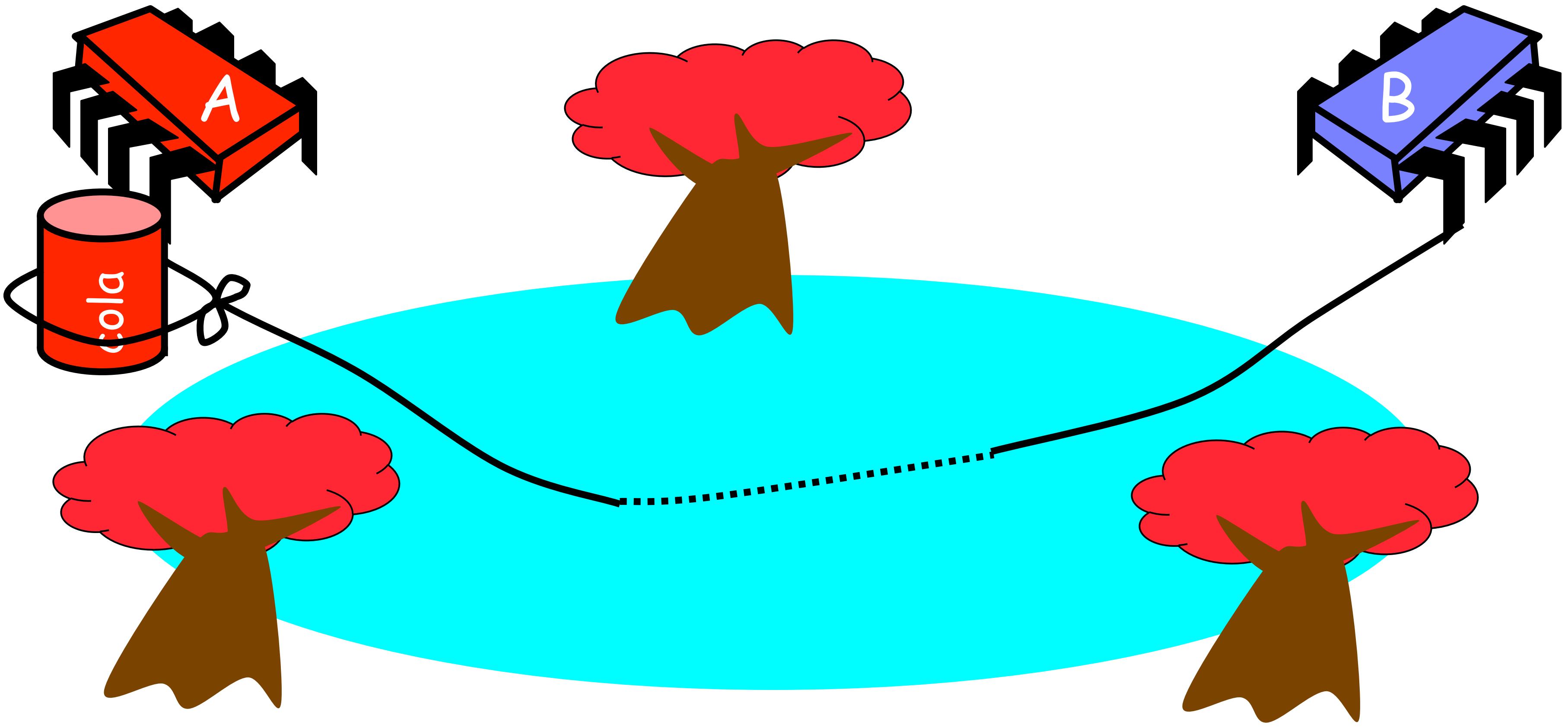
Producer/Consumer

- Alice and Bob can't meet
  - Each has restraining order (by court) 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

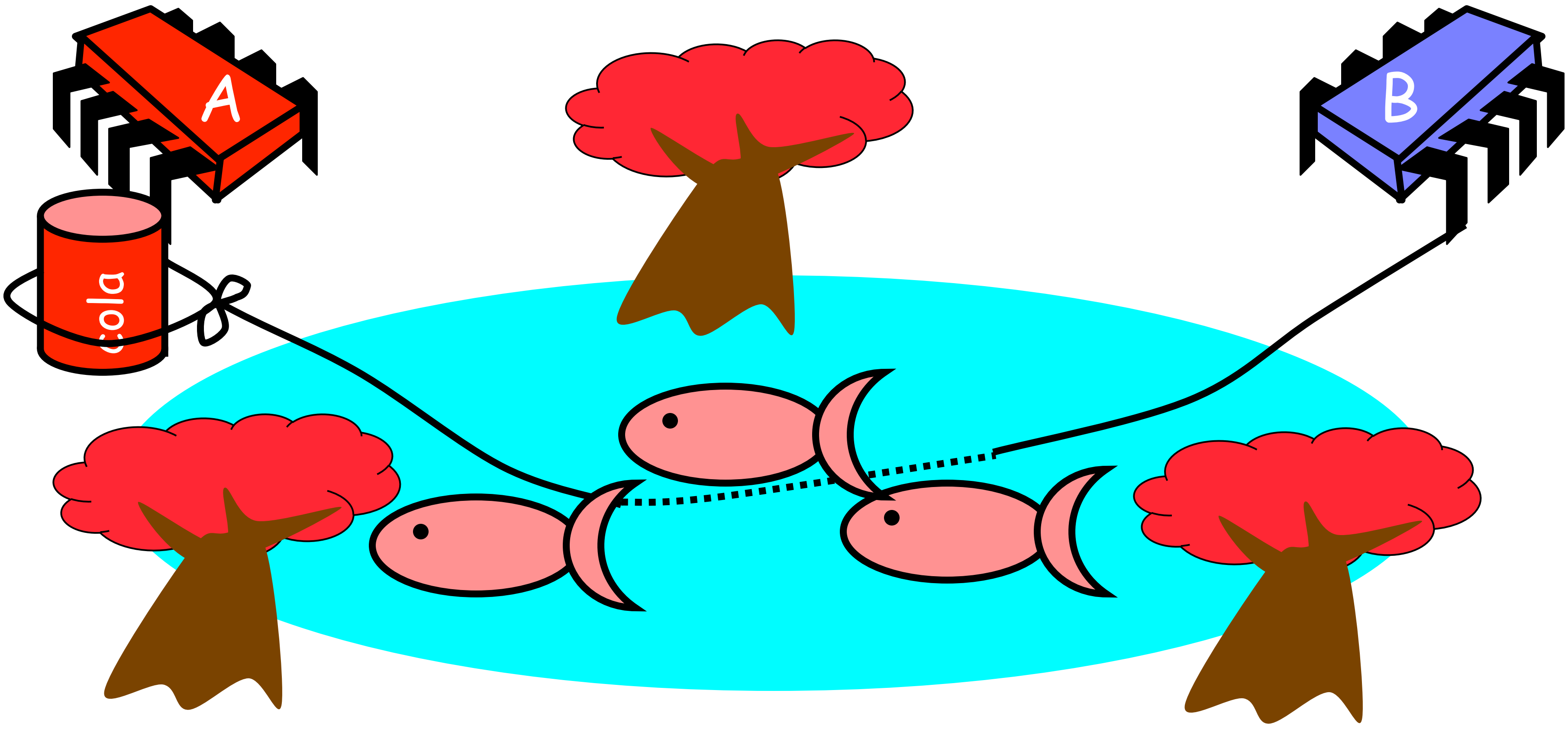
Producer/Consumer

- 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

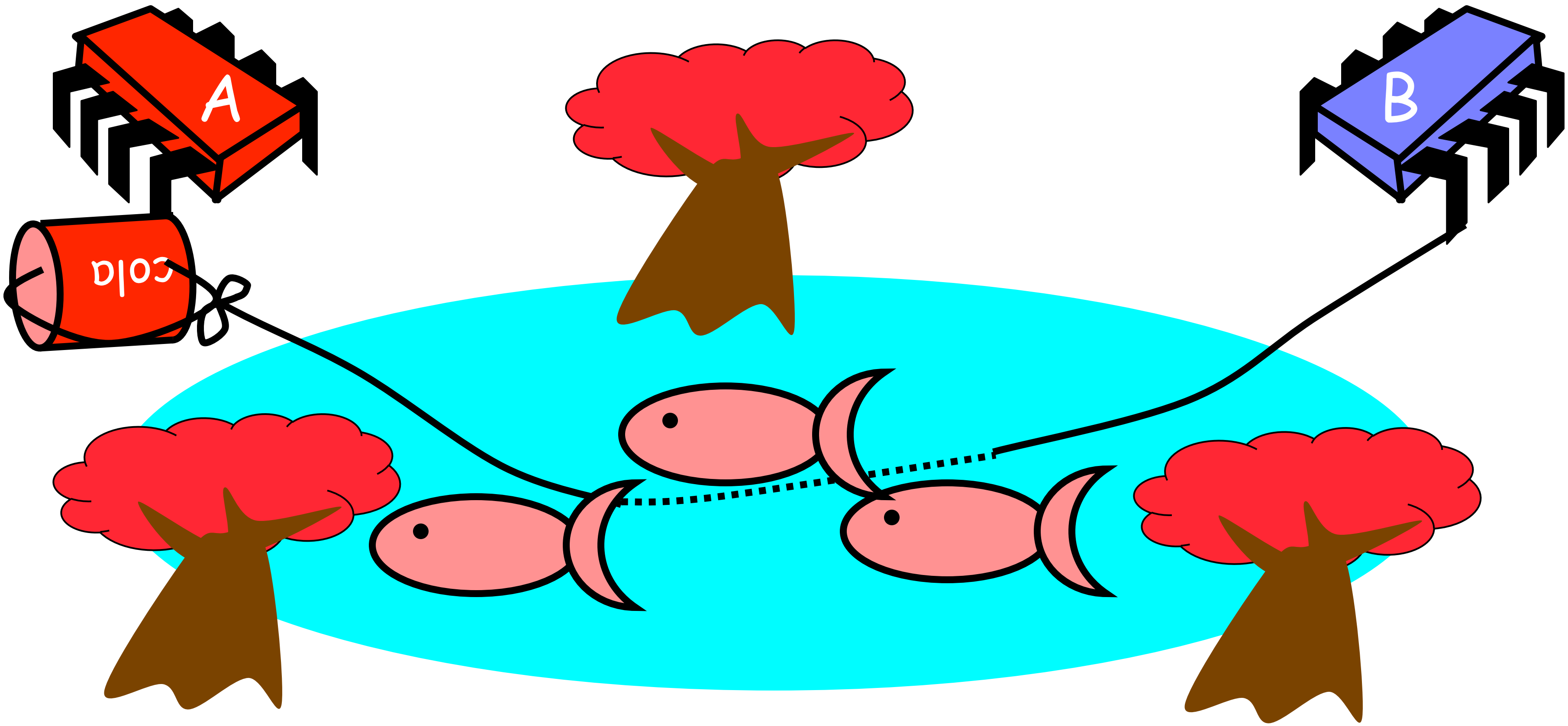
Surprise Solution



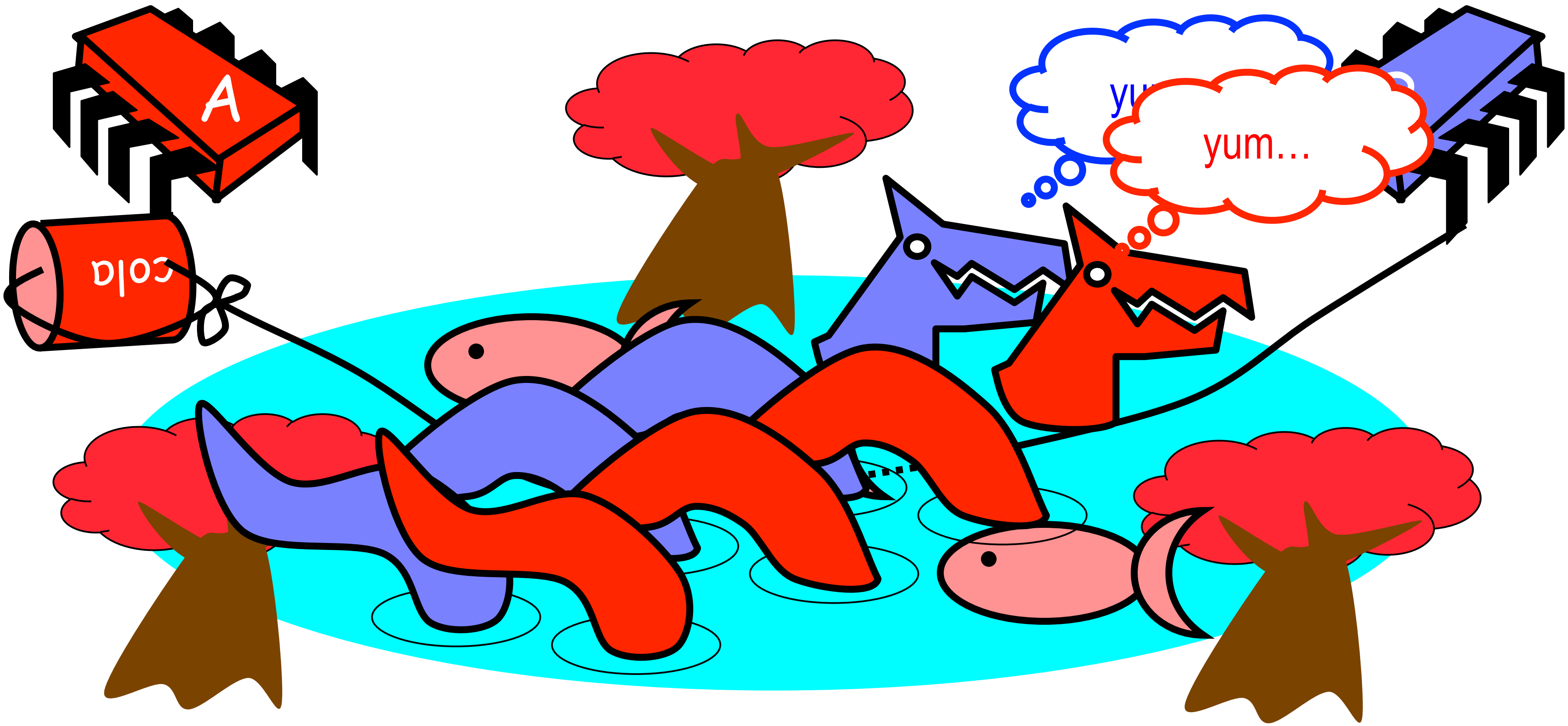
Bob puts food in Pond



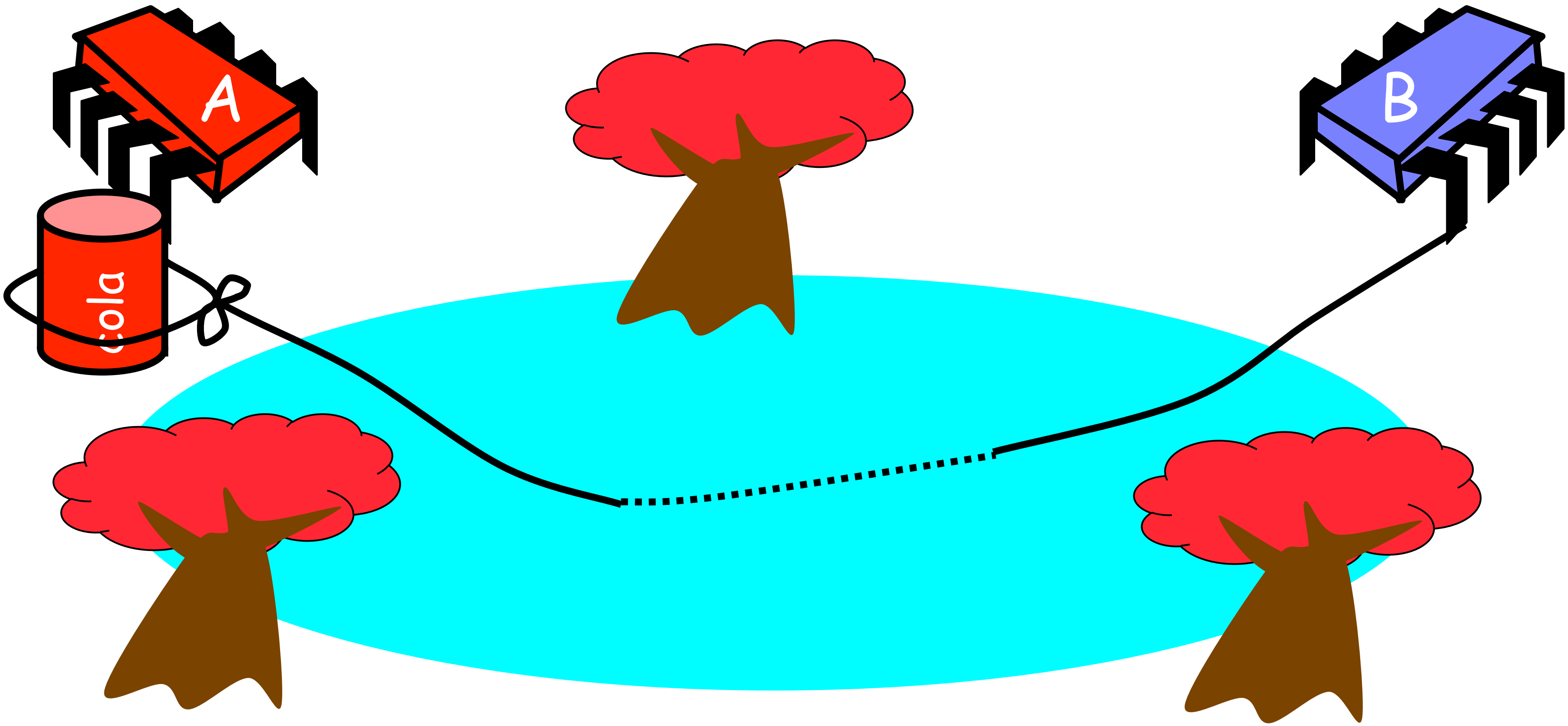
Bob knocks over Can



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

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

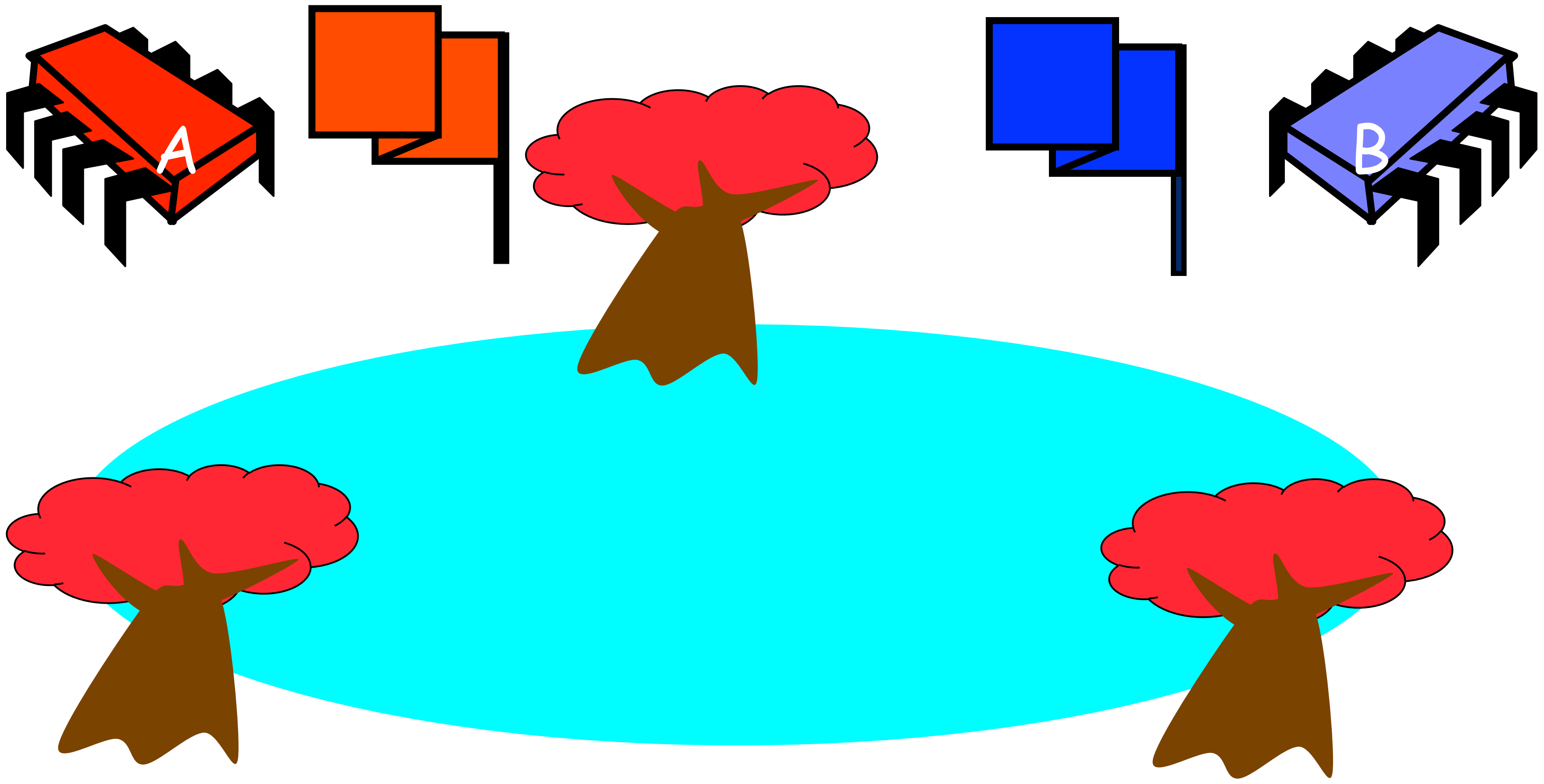
- Mutual Exclusion
  - Pets and Bob never together in pond
- No Starvation

  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.

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

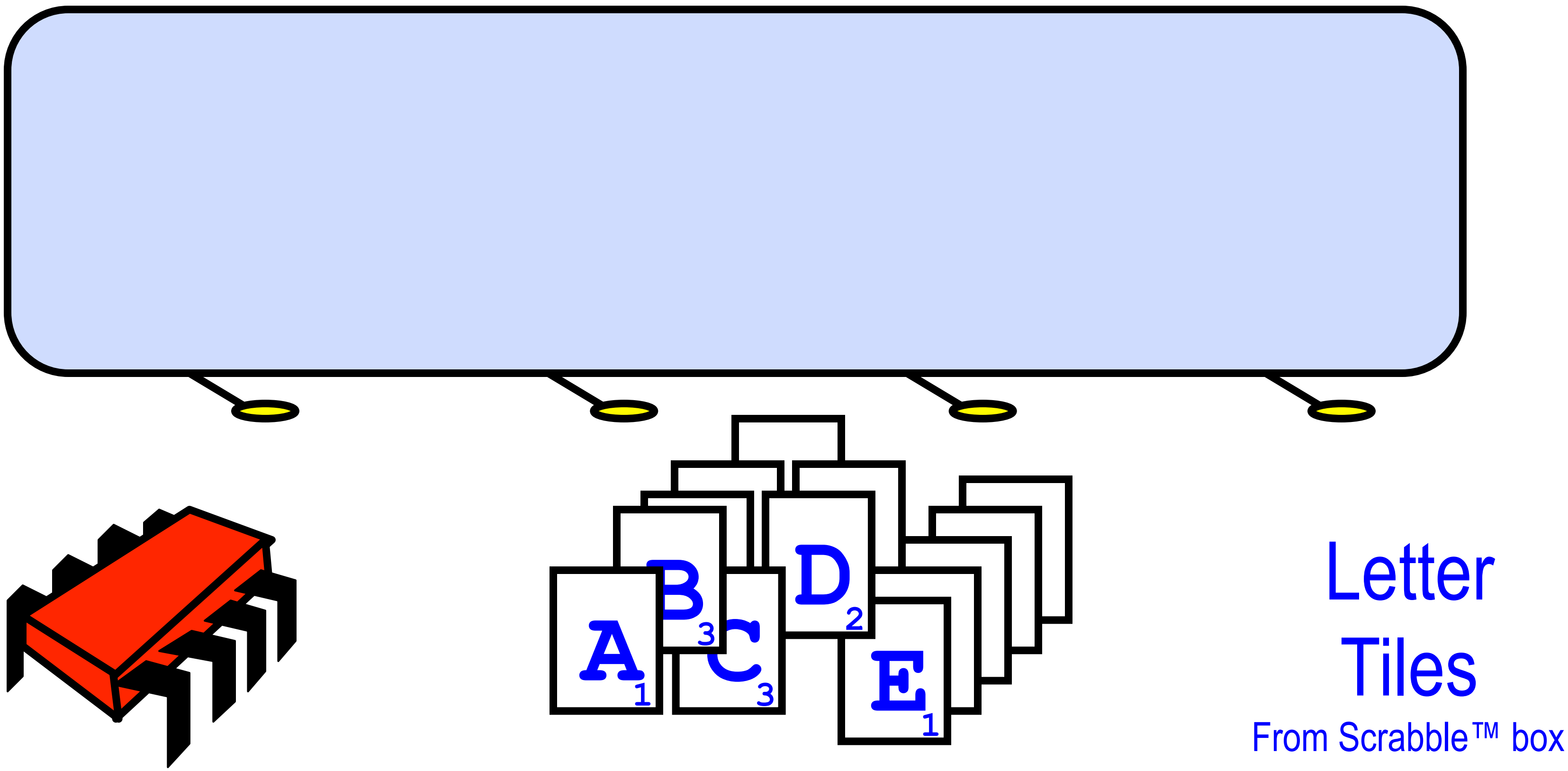
The Fable drags on …

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

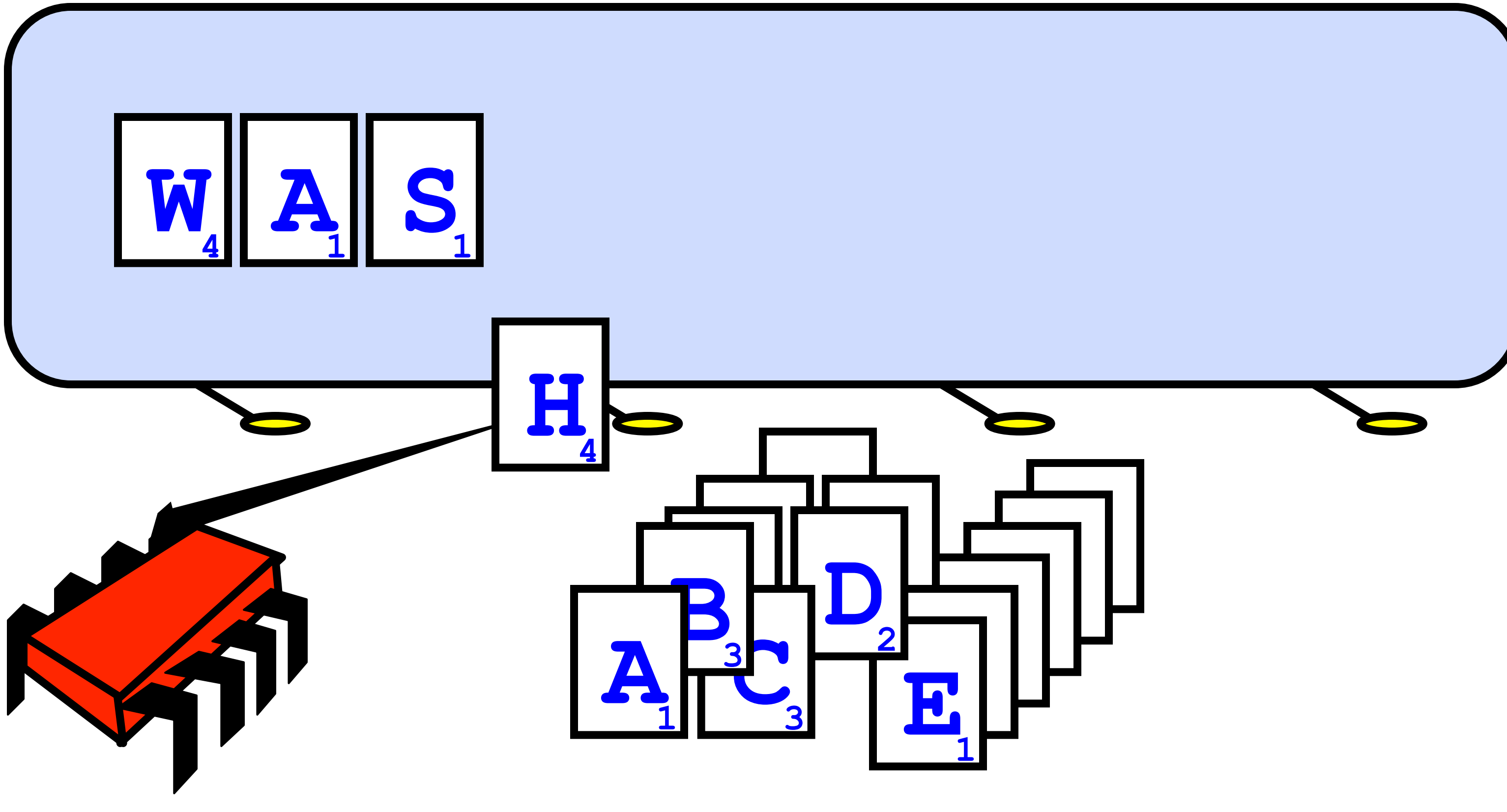
Readers/Writers

- Alice & Bob agree to write messages on a billboard
- Alice starts writing one letter at a time
  - First she writes "wash the car"
  - Then writes "sell the house"
  - What does Bob read?

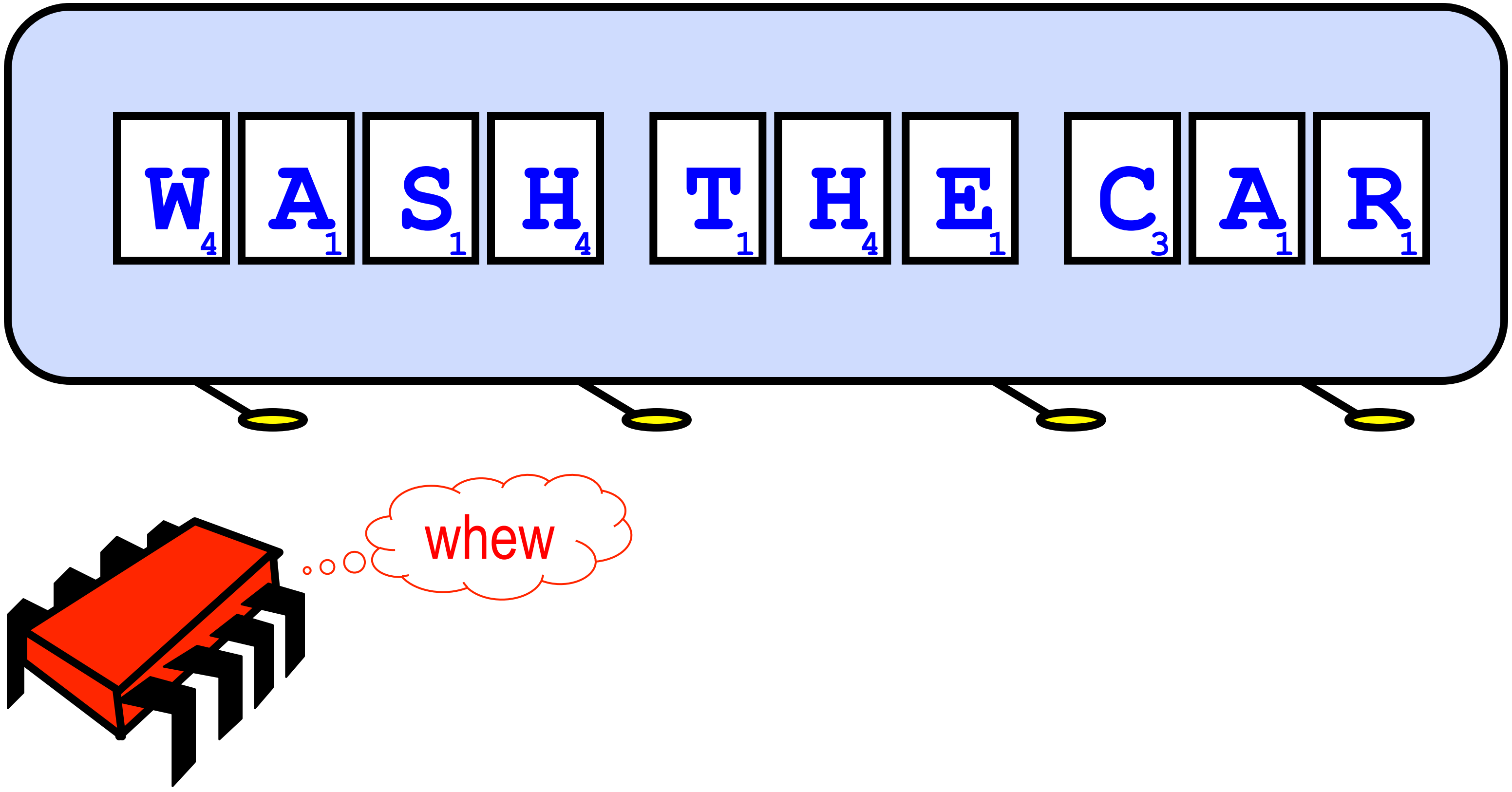
Billboards are Large



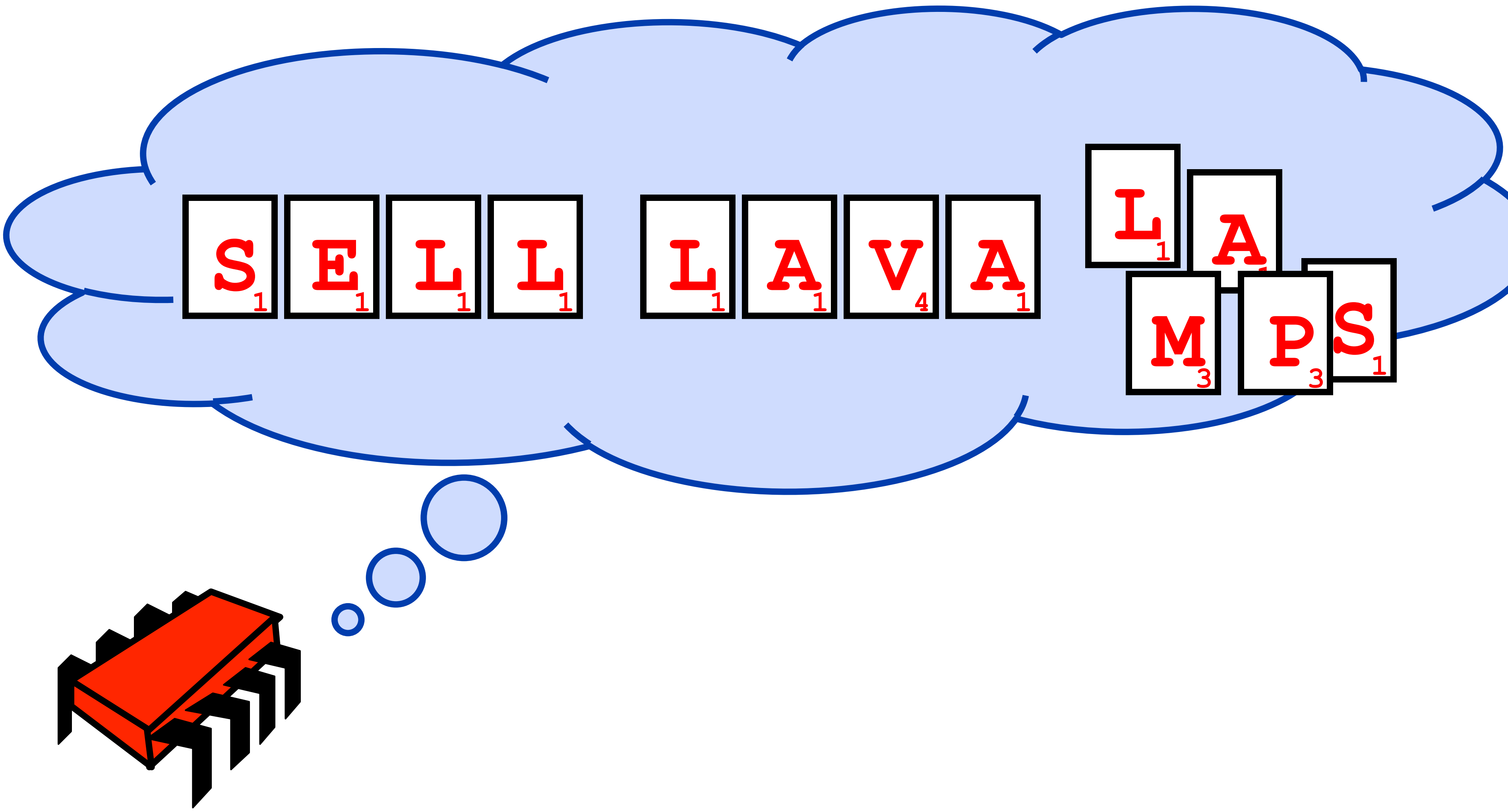
Write One Letter at a Time …



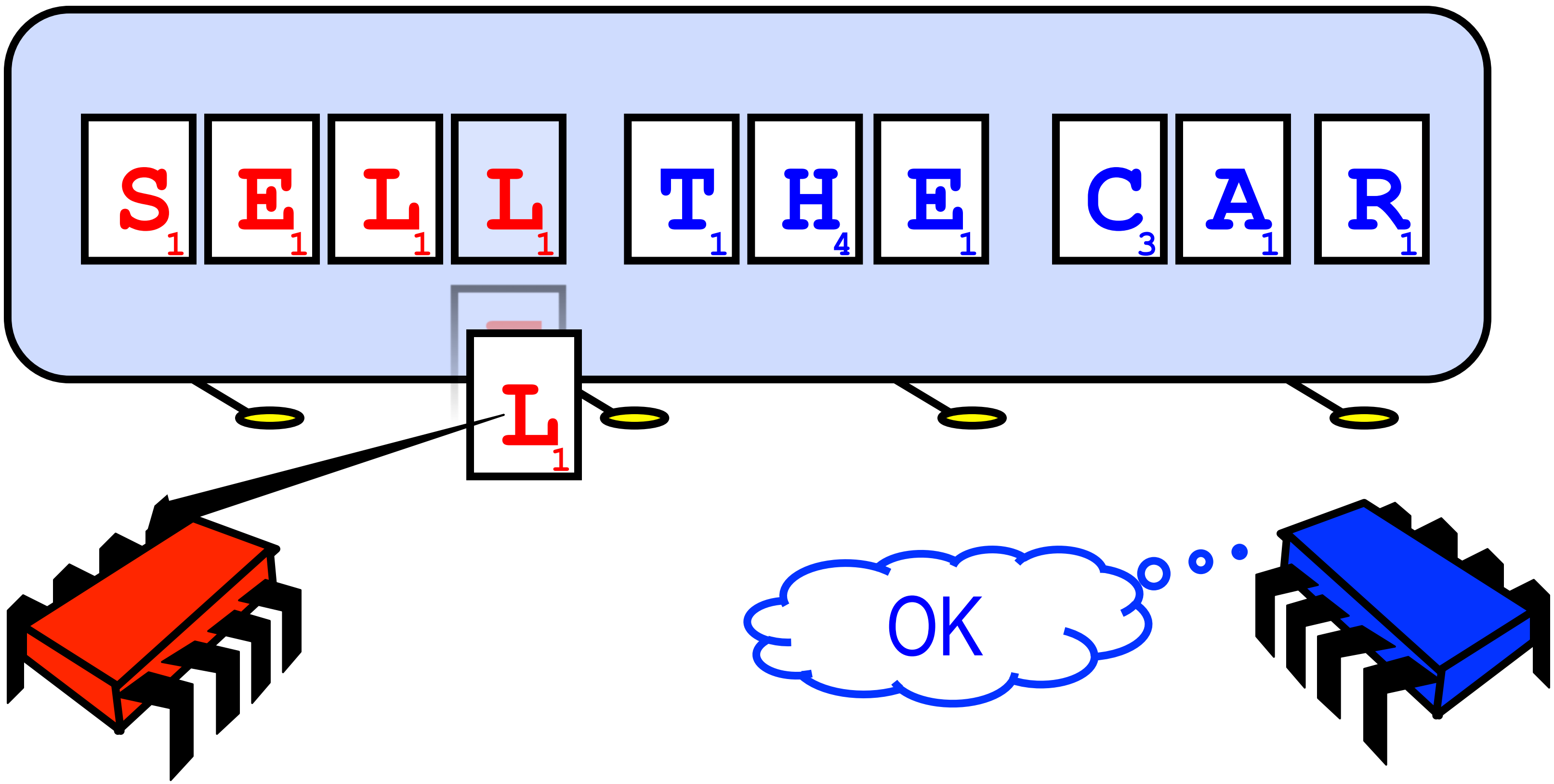
To post a message



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 "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

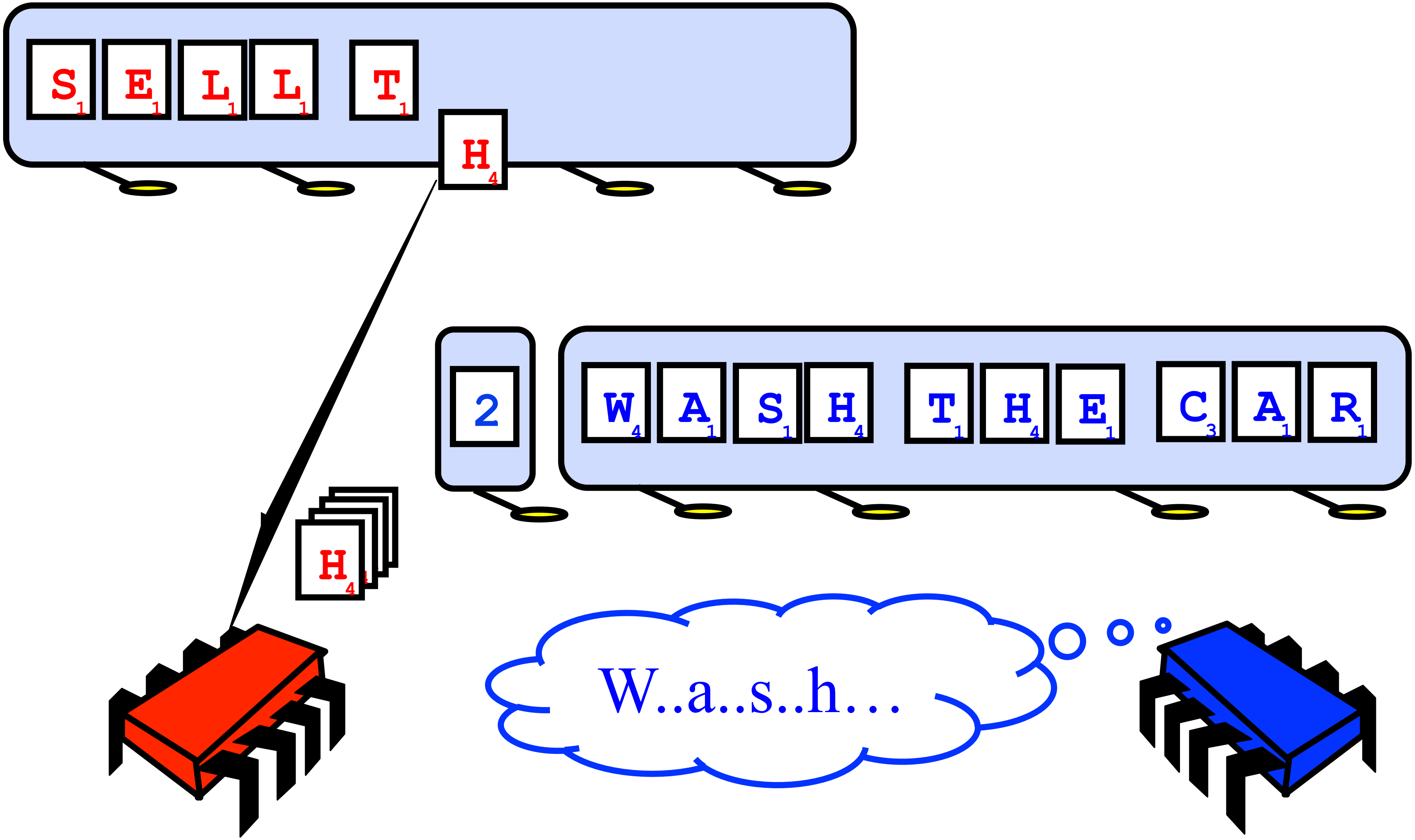
Why do we care?

- 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…

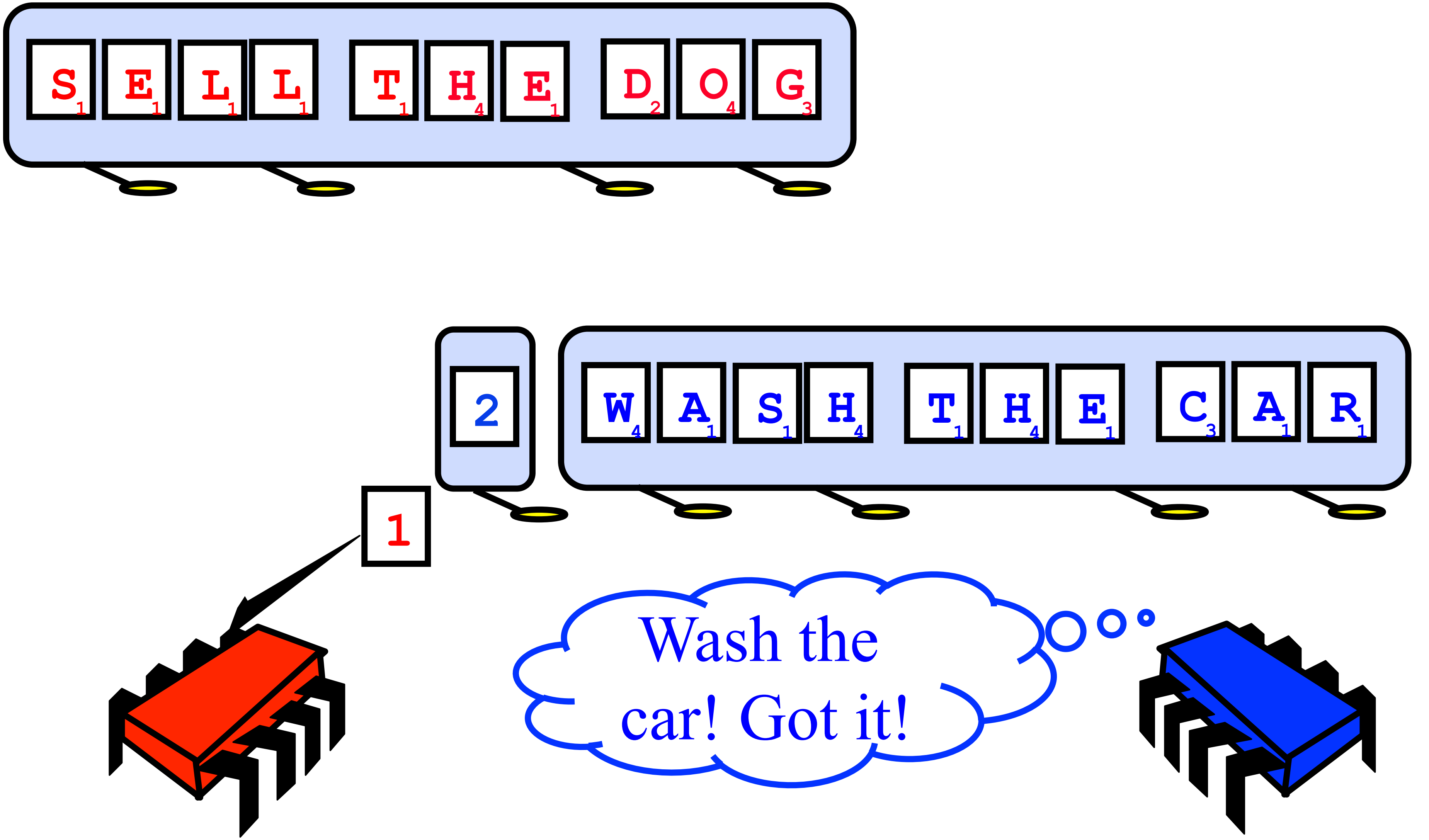
Parallelizing Readers/Writers

- Use two billboards
  - While Bob reads from one…
  - Alice writes to the other
- How do they know where to read/write?
  - Third billboard
  - Tells Bob which board to read

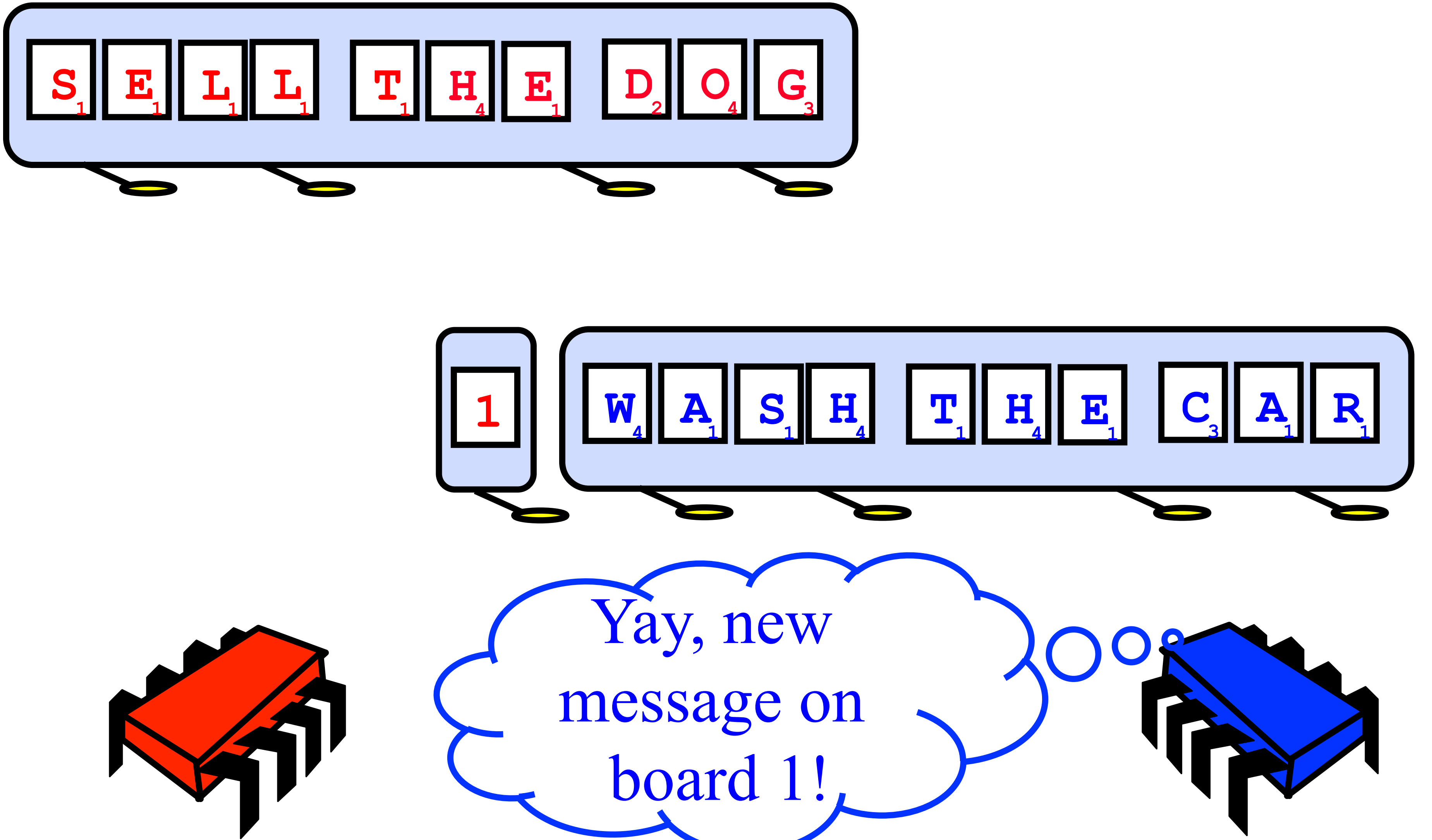
Wait-free protocol



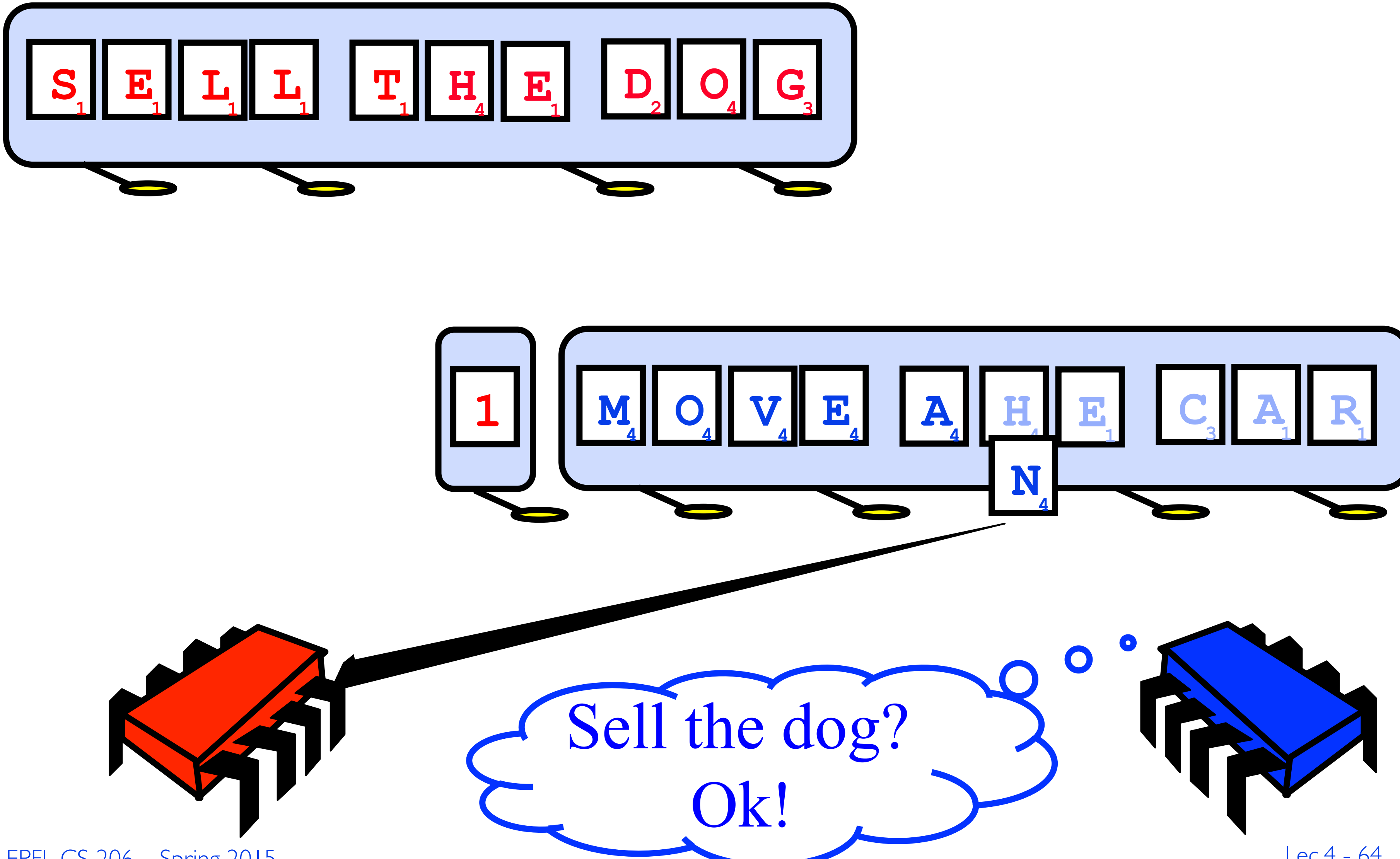
Wait-free protocol



Wait-free protocol



Wait-free protocol



Wait-free protocol

- Is this protocol entirely wait-free?
- Is the protocol correct?
- How do you fix it?

Summary

- Need protocols to coordinate
- Mutual exclusion properties
  - Safety
  - Liveness
  - Fairness
- Producer/Consumer sharing
- Reducing sequential execution/waiting