01 Introduction

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

What is concurrency?

What is concurrency?

- We use concurrency in this course in the broad sense

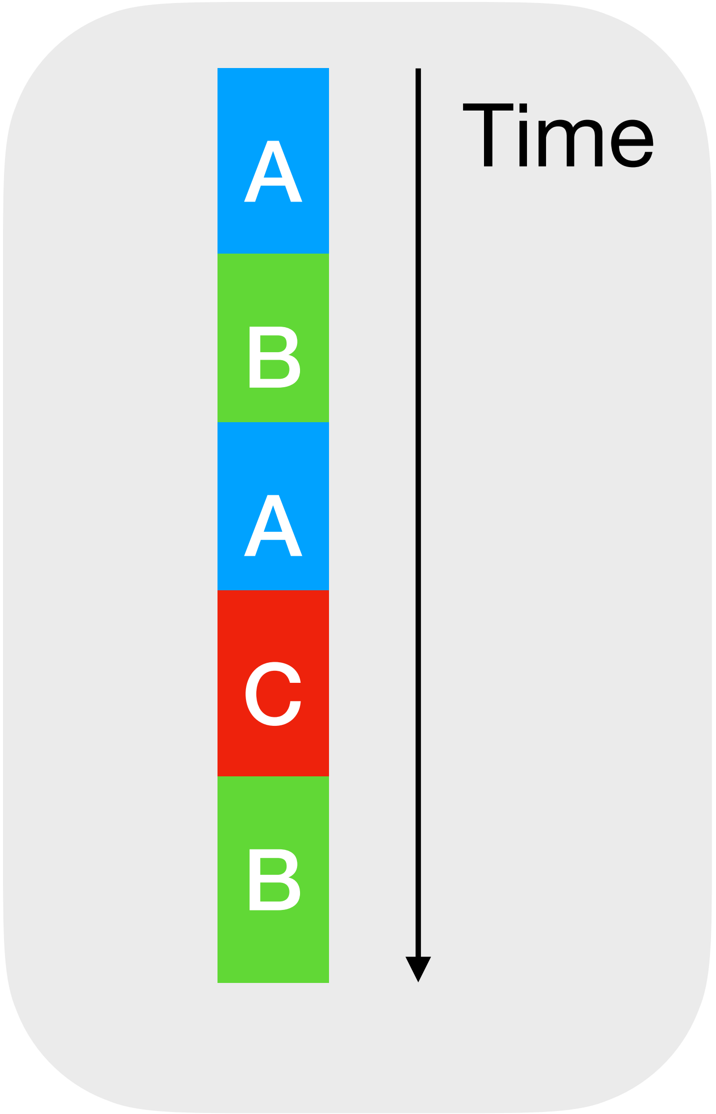
What is concurrency?

- We use concurrency in this course in the broad sense
- Encompasses multiple definitions that people use in practice
  - **“Concurrency”, Parallelism and Distribution**

Concurrency

- *Interleaved* executions of tasks in time
- **Examples** — Asynchronous I/O, GUI in JavaScript, interrupt-driven embedded systems, etc.
- ***No need for multiple cores!***
- ***Challenges***
    - non-determinism
    - Many different schedules complicate reasoning
    - efficiency, etc.

Concurrency

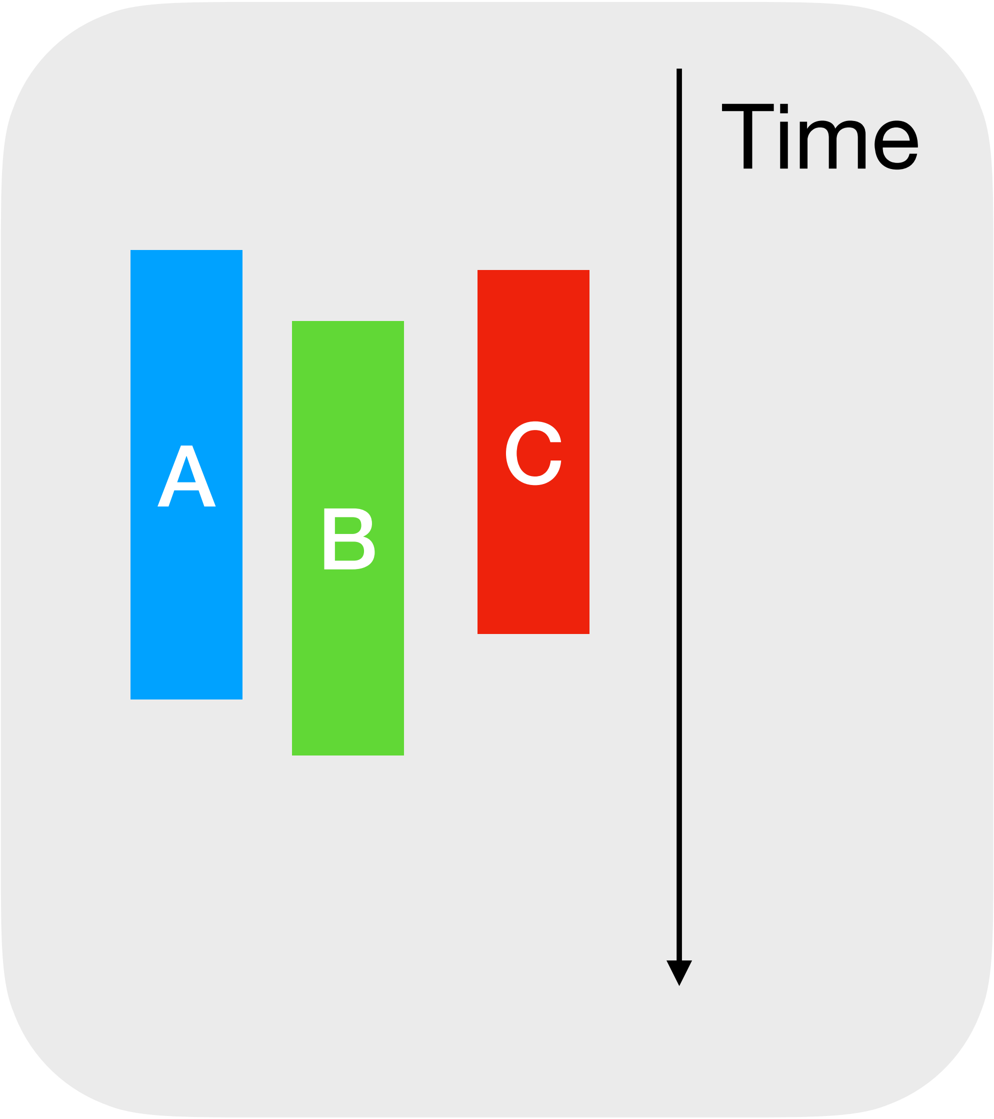


Interleaved execution

Parallelism

- Multiple ***hardware execution units*** — Multicore CPUs, GPUs, parallelism across machines (such as supercomputers, GPU farms)
- Goal is ***speed*** with correctness
- ***Challenges***
  - All the concurrency challenges
  - Relaxed memory behaviours
  - Performance scaling challenges
  - Debugging challenges

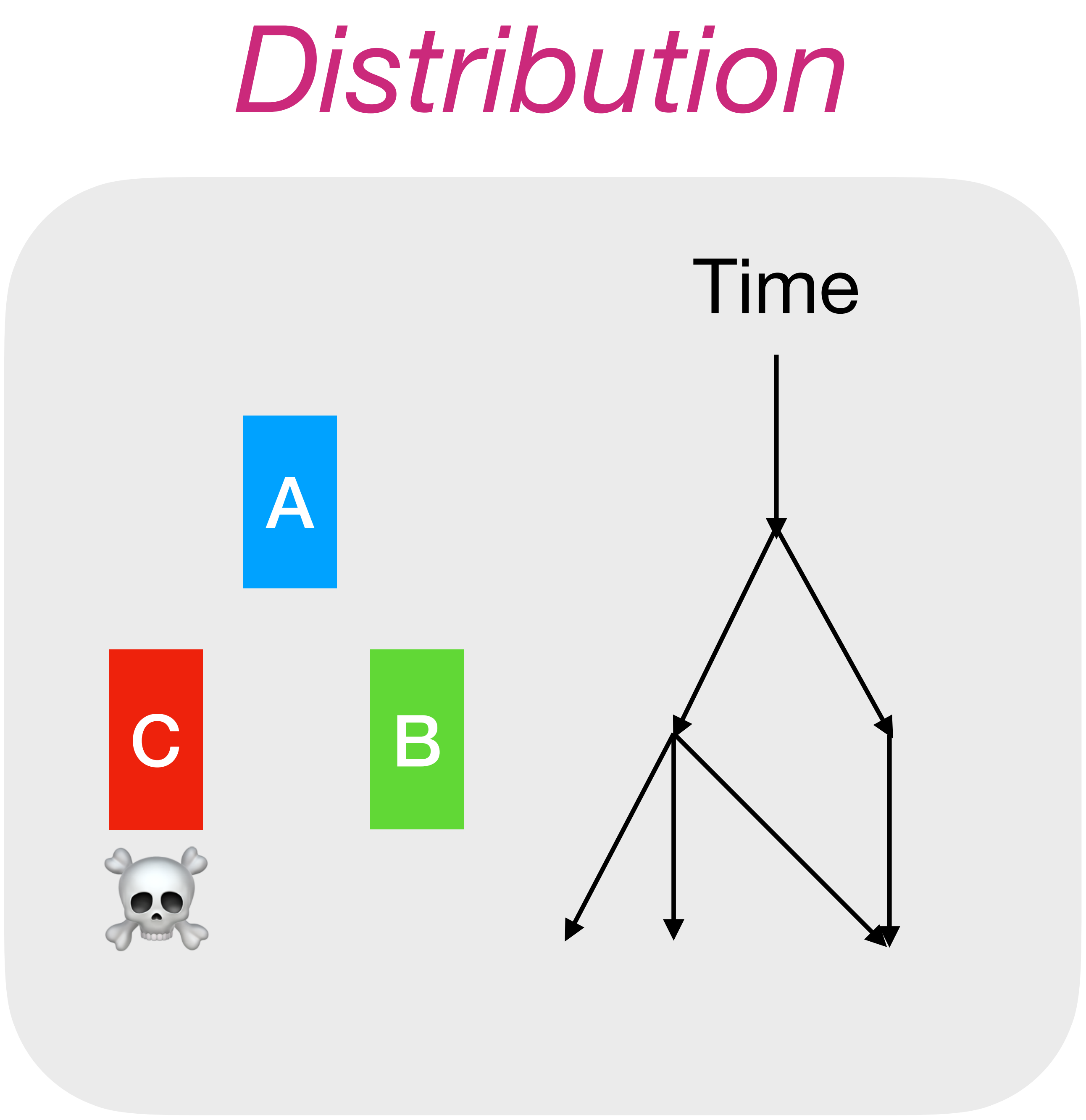
Parallelism



Simultaneous execution

Distribution

- Execution is distributed across many ***loosely-coupled*** machines
  - There is often ***no shared global clock*** across computations
  - A subset of computations may ***crash/fail.***
- **Goal** — performance, resilience in the presence of partial failures.
- Examples — the Internet, blockchains, etc.
- ***Challenges***
  - All the problems of concurrency
  - Partial failures
  - Byzantine behaviour
  - Performance scaling
  - Debugging



Why this course?

- Concurrency is ***everywhere***
  - Embedded Systems, OS, Multicore Hardware, GPUs, Reactive Systems, Cloud, …

Why this course?

- Concurrency is ***everywhere***
  - Embedded Systems, OS, Multicore Hardware, GPUs, Reactive Systems, Cloud, …
- This course equips you with the skills to ***reason*** and ***program*** under concurrency.
  - **Models:** Histories, Linearizability, Memory Models, Data races
  - **Mechanisms:** Mutual Exclusion, Atomics, Lockfree Algorithms, Strong type systems
  - **Abstractions:** Concurrency Monads, Effect Handlers, Structured Concurrency
  - **Programming:** Implementing concurrency, safety, performance

Why this course?

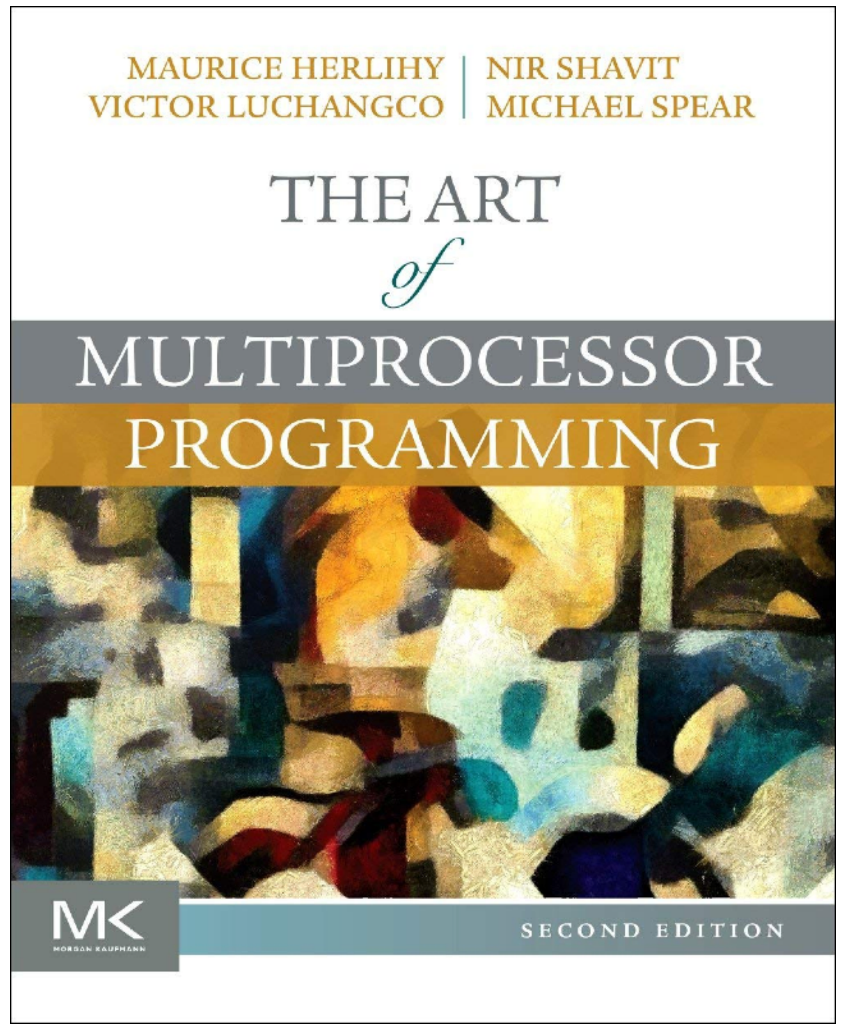
- Concurrency is ***everywhere***
  - Embedded Systems, OS, Multicore Hardware, GPUs, Reactive Systems, Cloud, …
- This course equips you with the skills to ***reason*** and ***program*** under concurrency.
  - **Models:** Histories, Linearizability, Memory Models, Data races
  - **Mechanisms:** Mutual Exclusion, Atomics, Lockfree Algorithms, Strong type systems
  - **Abstractions:** Concurrency Monads, Effect Handlers, Structured Concurrency
  - **Programming:** Implementing concurrency, safety, performance
- The course will not cover
  - Distribution
  - Performance optimisations

Course Outline

- **Parallelism**
  - Mutual Exclusion, Concurrent Objects, Relaxed Memory Models, Spin Locks, Contention, Blocking Synchronisation, Fine-grained Concurrent Data Structures

Course Outline

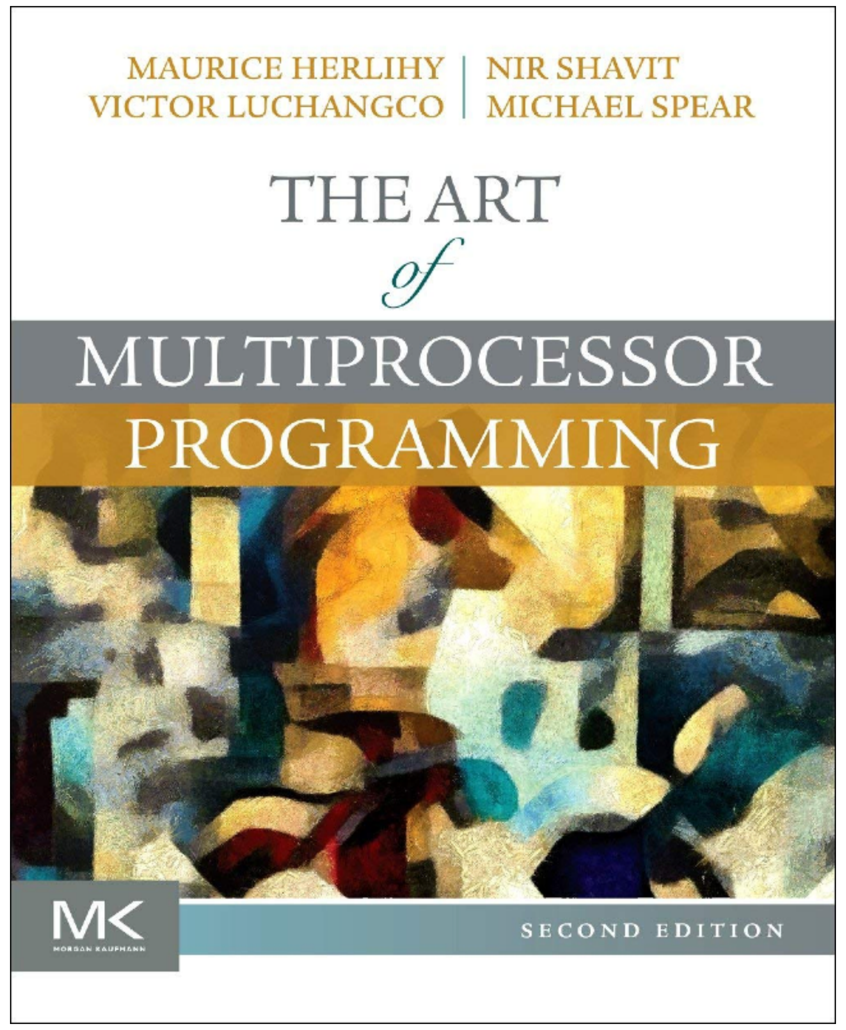
- **Parallelism**
  - Mutual Exclusion, Concurrent Objects, Relaxed Memory Models, Spin Locks, Contention, Blocking Synchronisation, Fine-grained Concurrent Data Structures



2nd Edition

Course Outline

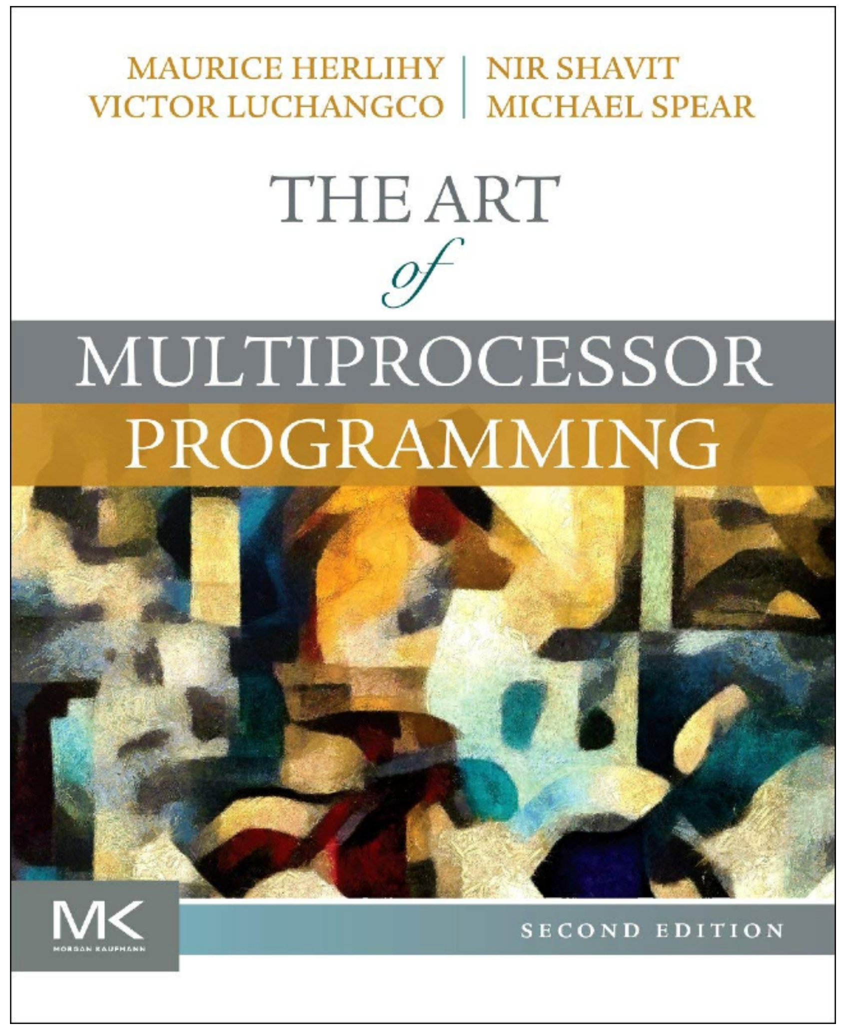
- **Parallelism**
  - Mutual Exclusion, Concurrent Objects, Relaxed Memory Models, Spin Locks, Contention, Blocking Synchronisation, Fine-grained Concurrent Data Structures
- **Concurrency**
  - Continuations, Concurrency Monads, Effect handlers, Schedulers, Concurrent data structures, Asynchronous I/O



2nd Edition

Course Outline

- **Parallelism**
    - Mutual Exclusion, Concurrent Objects, Relaxed Memory Models, Spin Locks, Contention, Blocking Synchronisation, Fine-grained Concurrent Data Structures
- **Concurrency**
    - Continuations, Concurrency Monads, Effect handlers, Schedulers, Concurrent data structures, Asynchronous I/O



2nd Edition

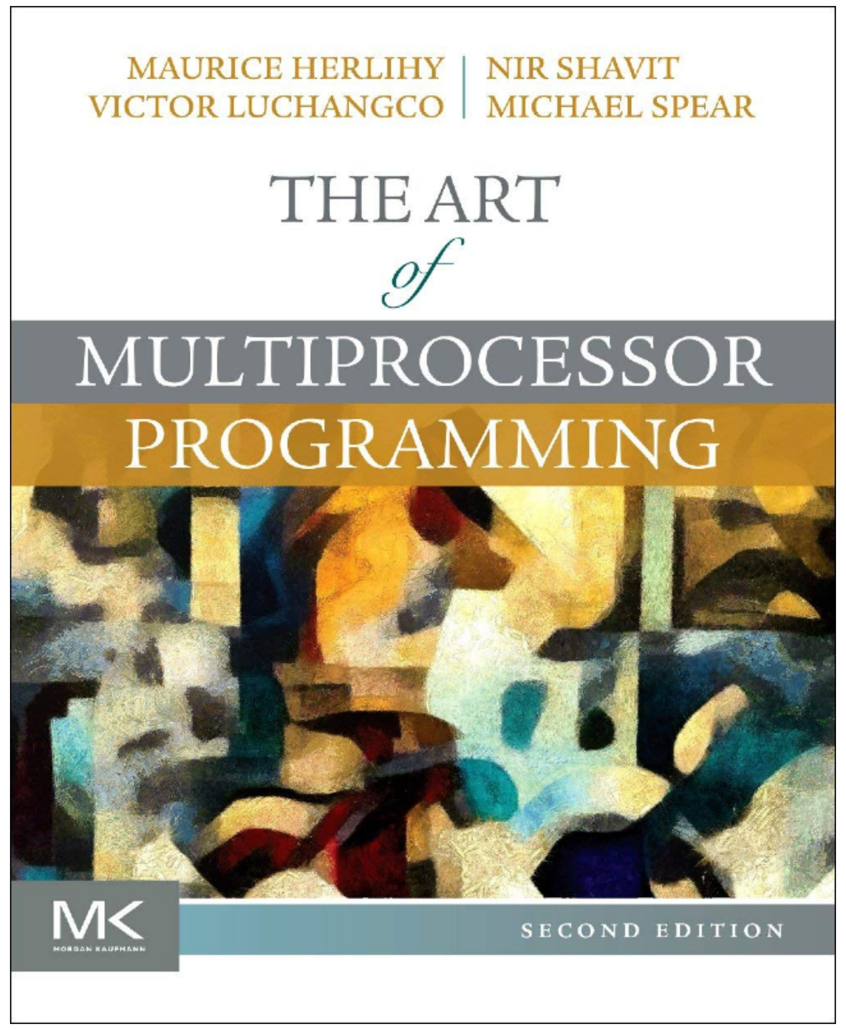
Control structures in programming languages: from goto to algebraic effects

Xavier Leroy

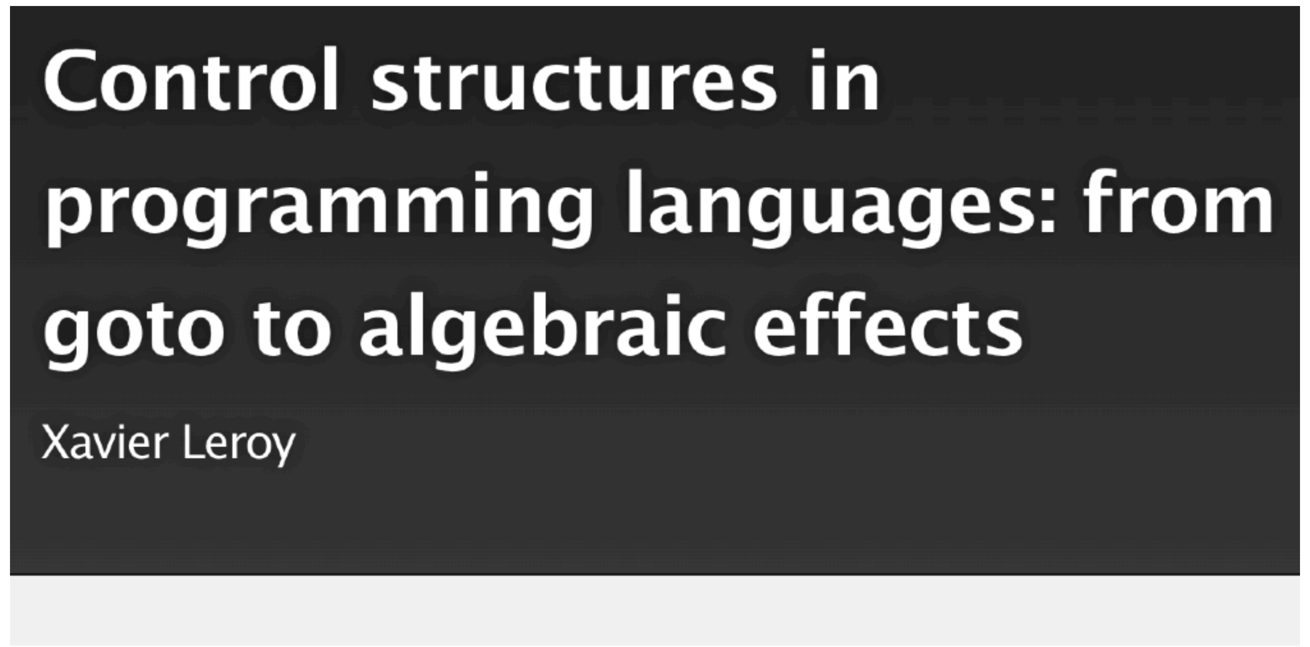
https://xavierleroy.org/control-structures/

Course Outline

- **Parallelism**
  - Mutual Exclusion, Concurrent Objects, Relaxed Memory Models, Spin Locks, Contention, Blocking Synchronisation, Fine-grained Concurrent Data Structures
- **Concurrency**
  - Continuations, Concurrency Monads, Effect handlers, Schedulers, Concurrent data structures, Asynchronous I/O
- **Safe Concurrent Programming**
  - Modes, DRF Parallel Programming



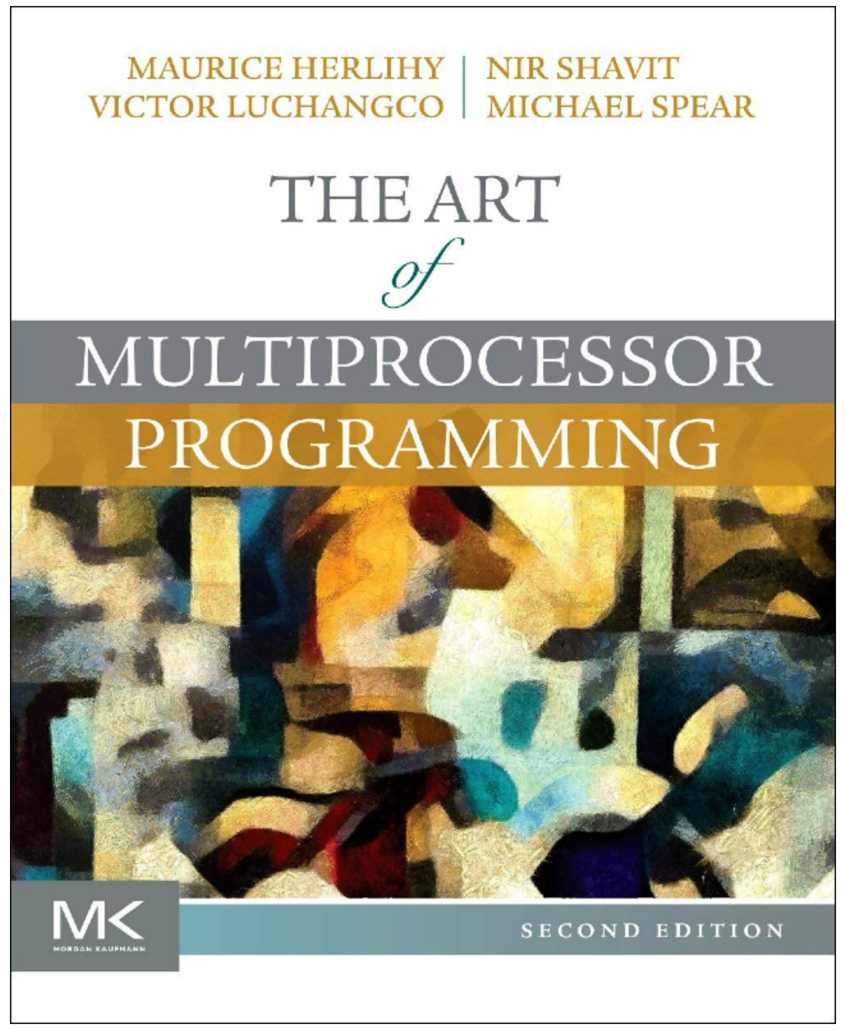
2nd Edition



https://xavierleroy.org/control-structures/

Course Outline

- **Parallelism**
    - Mutual Exclusion, Concurrent Objects, Relaxed Memory Models, Spin Locks, Contention, Blocking Synchronisation, Fine-grained Concurrent Data Structures
- **Concurrency**
    - Continuations, Concurrency Monads, Effect handlers, Schedulers, Concurrent data structures, Asynchronous I/O
- **Safe Concurrent Programming**
    - Modes, DRF Parallel Programming
- **First iteration of the course**
    - Course content neither sound nor complete



2nd Edition

Control structures in programming languages: from goto to algebraic effects

Xavier Leroy

https://xavierleroy.org/control-structures/

Course Language

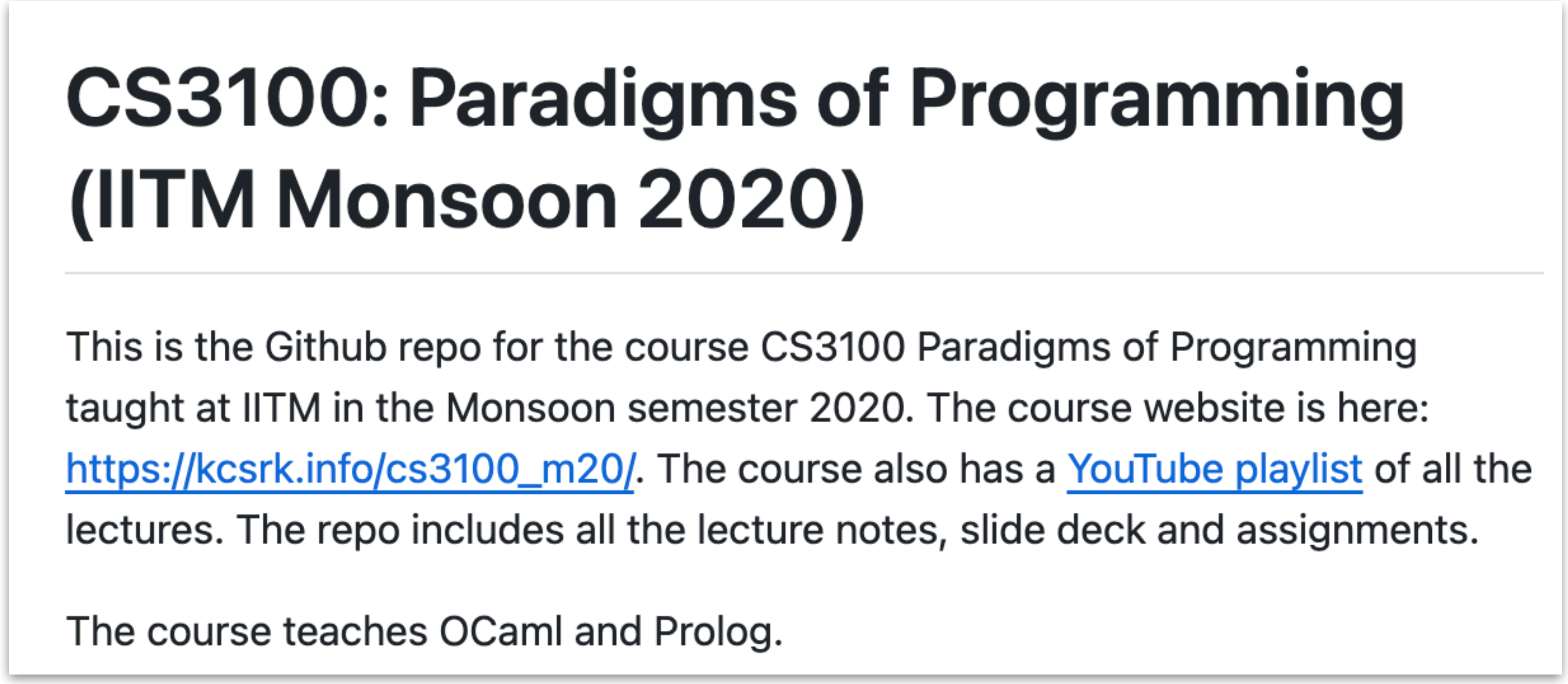
- ***OCaml*** will be the language of choice for the course

Course Language

- ***OCaml*** will be the language of choice for the course
- **Strong Pre-requisite: CS3100 OCaml Parts**
  - *I will not cover OCaml basics in this course!*

Course Language

- *OCaml* will be the language of choice for the course
- **Strong Pre-requisite: CS3100 OCaml Parts**
  - *I will not cover OCaml basics in this course!*
- OCaml Resources — CS3100 @ IITM, CS3110 book from Cornell U

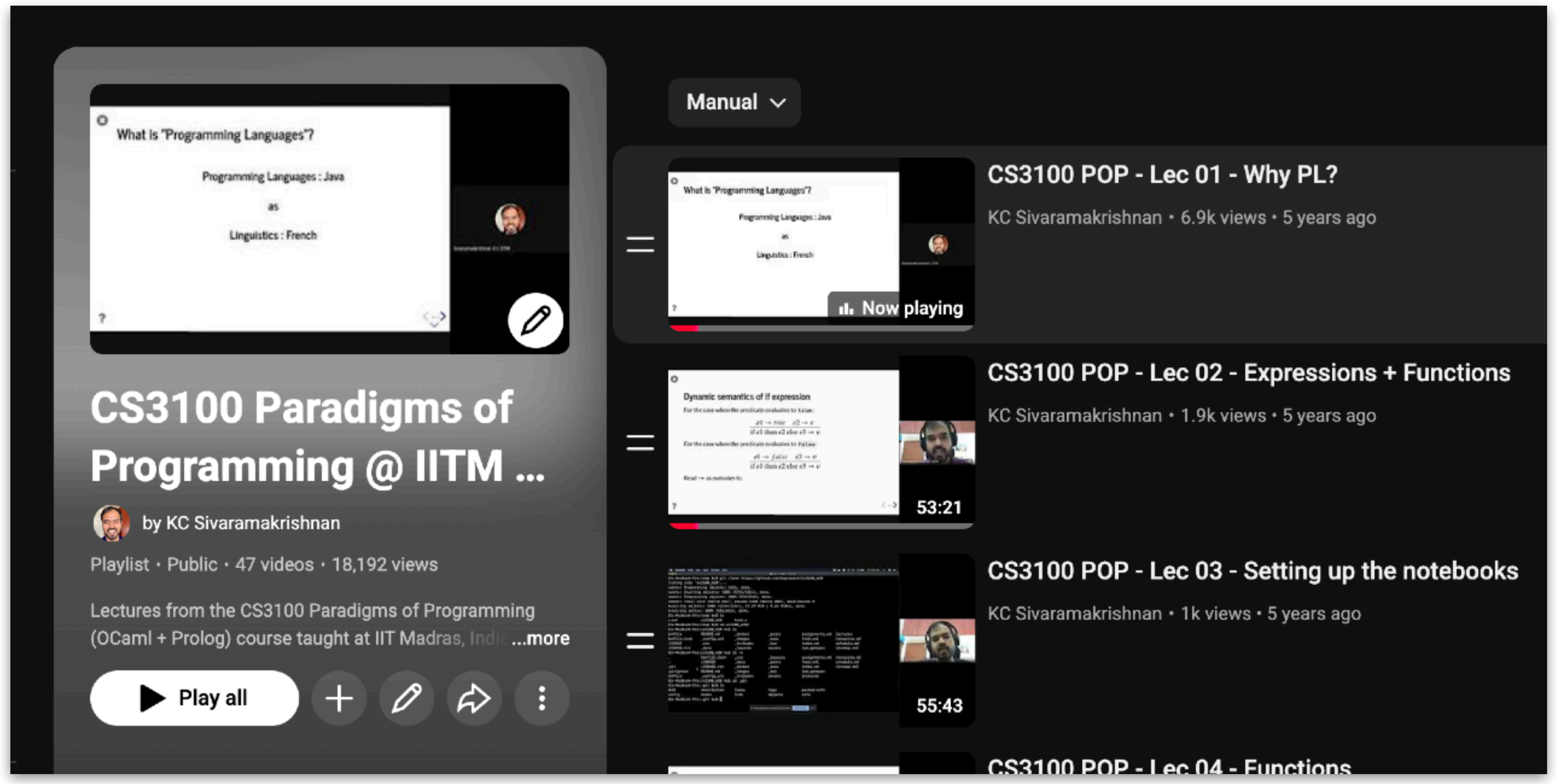


CS3100: Paradigms of Programming (IITM Monsoon 2020)

This is the Github repo for the course CS3100 Paradigms of Programming taught at IITM in the Monsoon semester 2020. The course website is here: https://kcsrk.info/cs3100_m20/. The course also has a YouTube playlist of all the lectures. The repo includes all the lecture notes, slide deck and assignments.

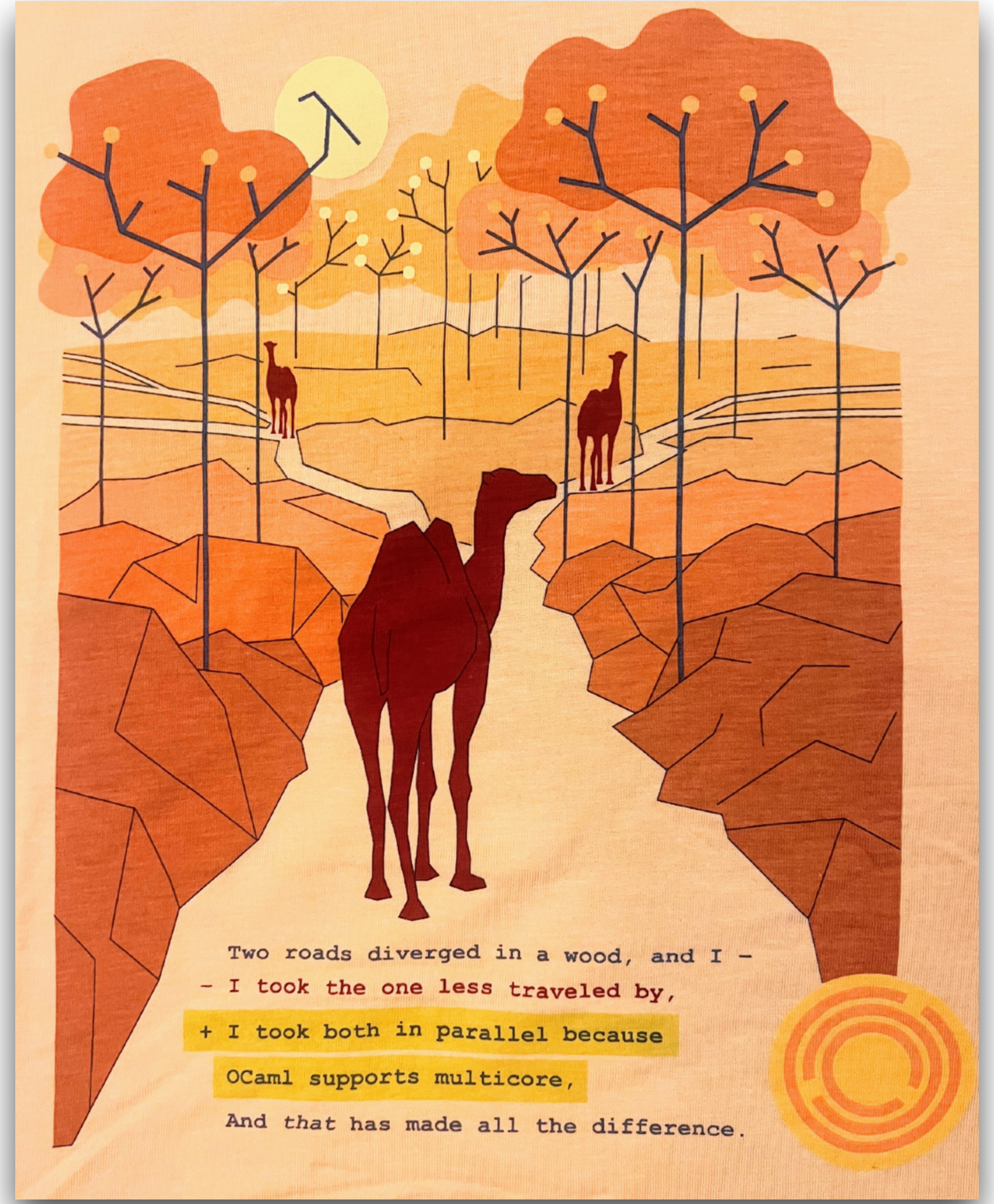
The course teaches OCaml and Prolog.

https://github.com/kayceesrk/cs3100_m20



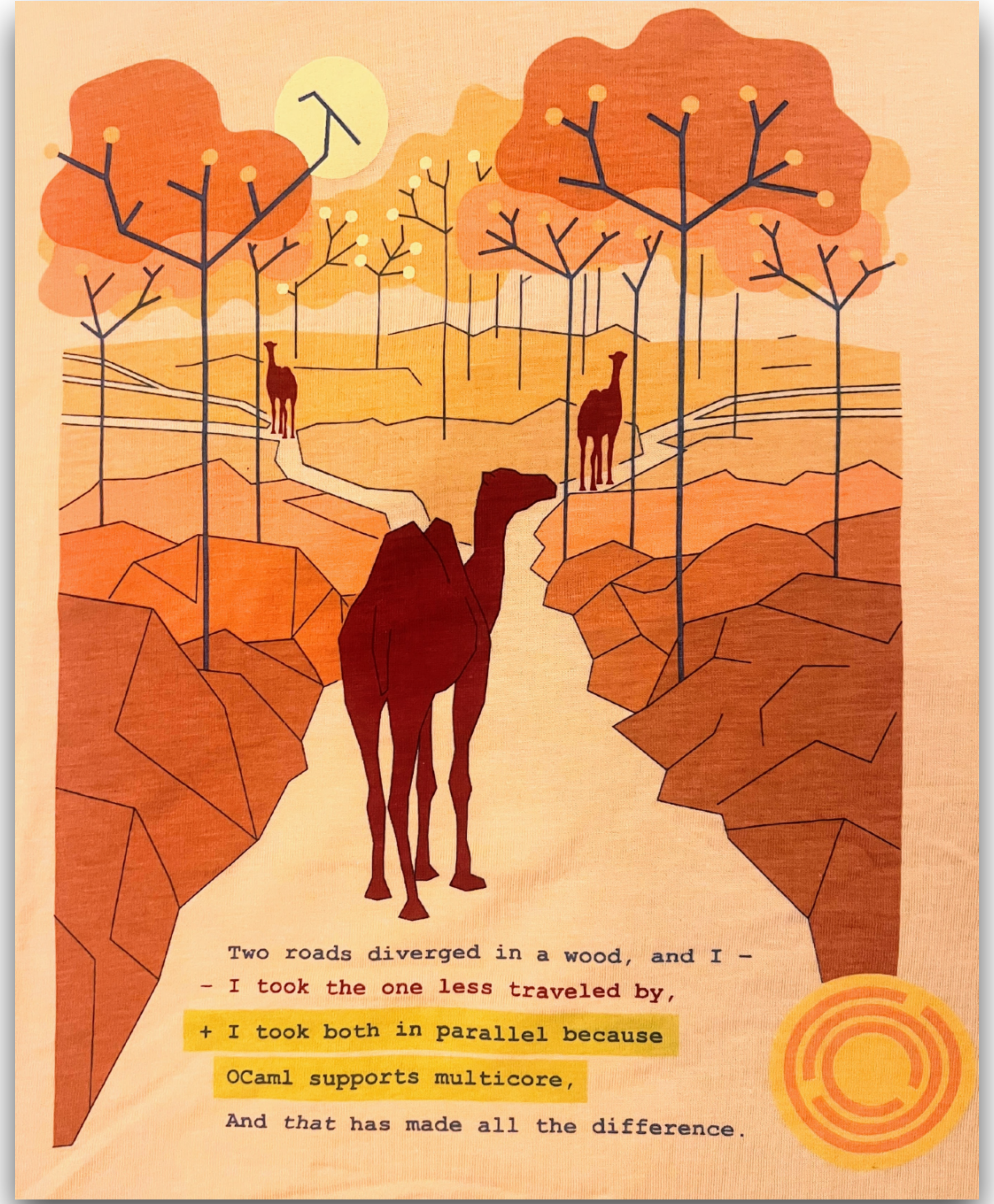
YouTube PlayList

Course Language



OCaml 5 — concurrency & parallelism

Course Language



OCaml 5 — concurrency & parallelism

OxCaml — safe, performance-oriented extension of OCaml

Evaluation Plan

- 6 in-class short quizzes **(20%)**
  - 15 minutes each, announced
  - 2 each — before quiz 1, before quiz 2, before the end sem
  - Best 5/6

** if you know what you want*

Evaluation Plan

- 6 in-class short quizzes **(20%)**
  - 15 minutes each, announced
  - 2 each — before quiz 1, before quiz 2, before the end sem
  - Best 5/6
- Mid-term **(20%)**

** if you know what you want*

Evaluation Plan

- 6 in-class short quizzes **(20%)**
  - 15 minutes each, announced
  - 2 each — before quiz 1, before quiz 2, before the end sem
  - Best 5/6
- Mid-term **(20%)**
- End Sem **(20%)**

** if you know what you want*

Evaluation Plan

- 6 in-class short quizzes **(20%)**
  - 15 minutes each, announced
  - 2 each — before quiz 1, before quiz 2, before the end sem
  - Best 5/6
- Mid-term **(20%)**
- End Sem **(20%)**
- Programming assignments x 4 **(24%)**
  - Individual

** if you know what you want*

Evaluation Plan

- 6 in-class short quizzes **(20%)**
    - 15 minutes each, announced
    - 2 each — before quiz 1, before quiz 2, before the end sem
    - Best 5/6
- Mid-term **(20%)**
- End Sem **(20%)**
- Programming assignments x 4 **(24%)**
    - Individual

Don't use LLM; they hinder learning

** if you know what you want*

Evaluation Plan

- 6 in-class short quizzes **(20%)**
  - 15 minutes each, announced
  - 2 each — before quiz 1, before quiz 2, before the end sem
  - Best 5/6
- Mid-term **(20%)**
- End Sem **(20%)**
- Programming assignments x 4 **(24%)**
  - Individual

Sign Honour Code;
Automated plagiarism check;
Institute policy on malpractice

Don't use LLM; they hinder learning

* *if you know what you want*

Evaluation Plan

- 6 in-class short quizzes **(20%)**
  - 15 minutes each, announced
  - 2 each — before quiz 1, before quiz 2, before the end sem
  - Best 5/6
- Mid-term **(20%)**
- End Sem **(20%)**
- Programming assignments x 4 **(24%)**
  - Individual
- Research mini project **(16%)**
  - Groups of 2

Sign Honour Code;
Automated plagiarism check;
Institute policy on malpractice

Don't use LLM; they hinder learning

** if you know what you want*

Evaluation Plan

- 6 in-class short quizzes **(20%)**
  - 15 minutes each, announced
  - 2 each — before quiz 1, before quiz 2, before the end sem
  - Best 5/6
- Mid-term **(20%)**
- End Sem **(20%)**
- Programming assignments x 4 **(24%)**
  - Individual
- Research mini project **(16%)**
  - Groups of 2

Sign Honour Code;
Automated plagiarism check;
Institute policy on malpractice

Don't use LLM; they hinder learning

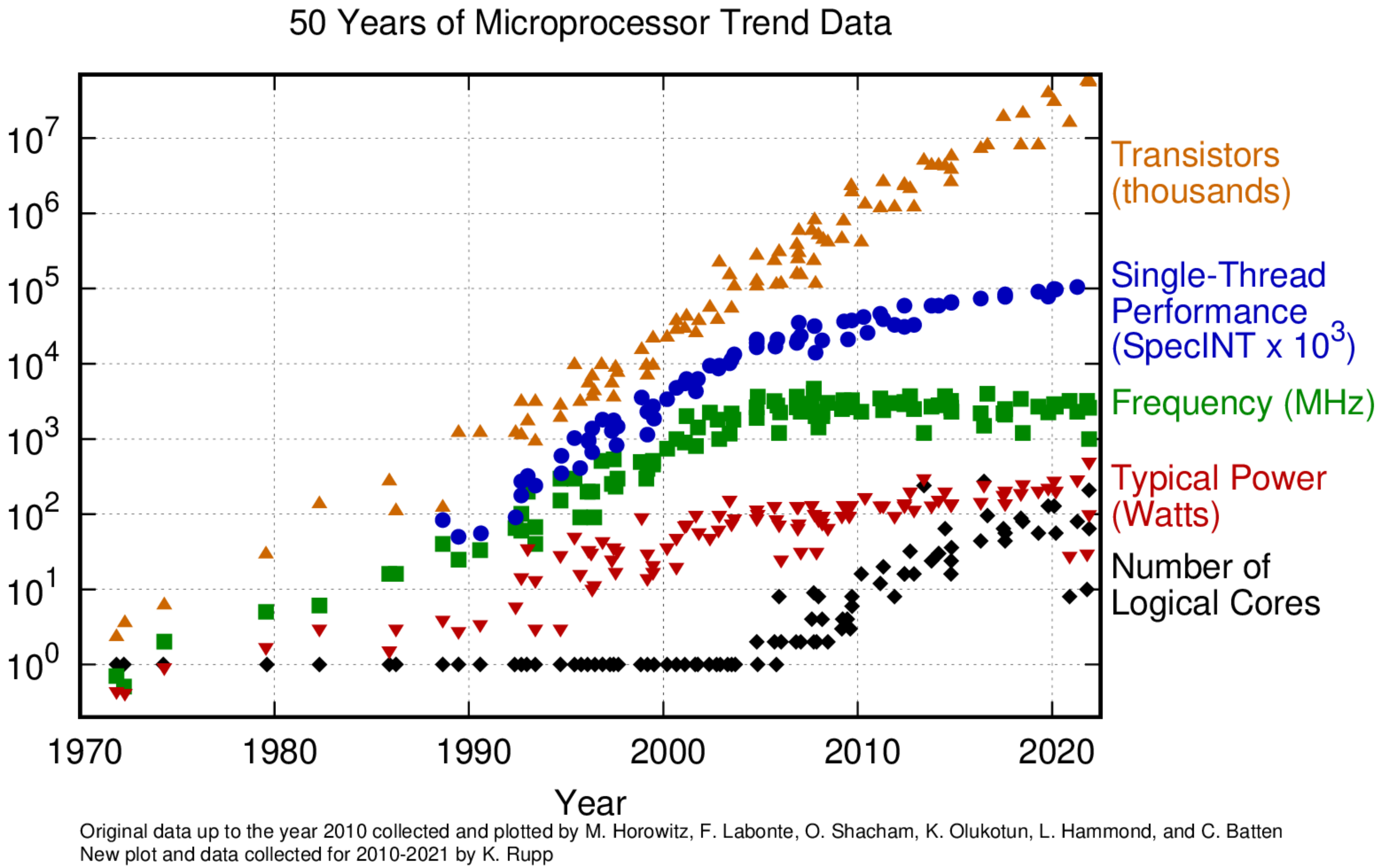
*Use LLM; they make you productive**

** if you know what you want*

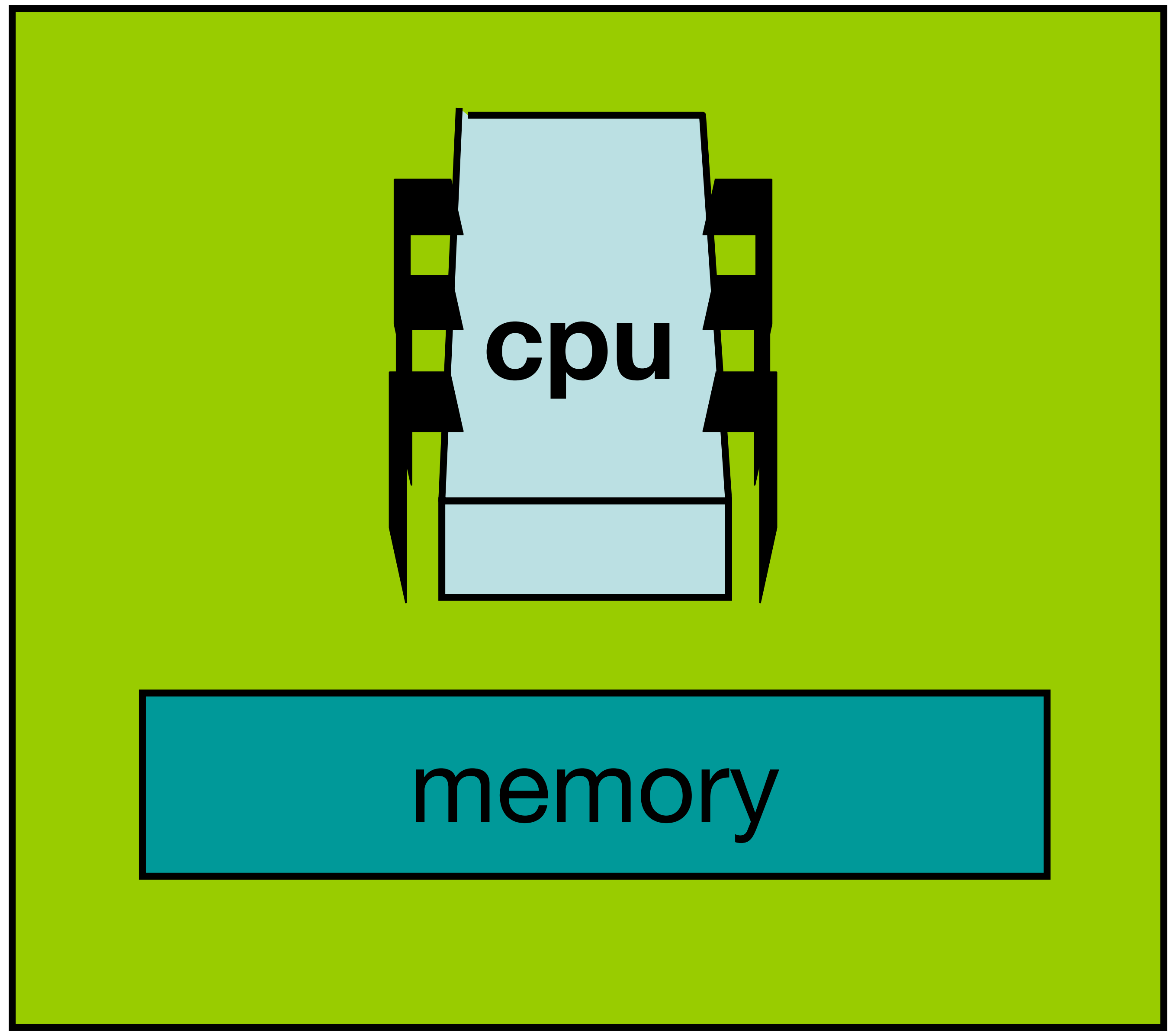
Parallelism

Moore's Law

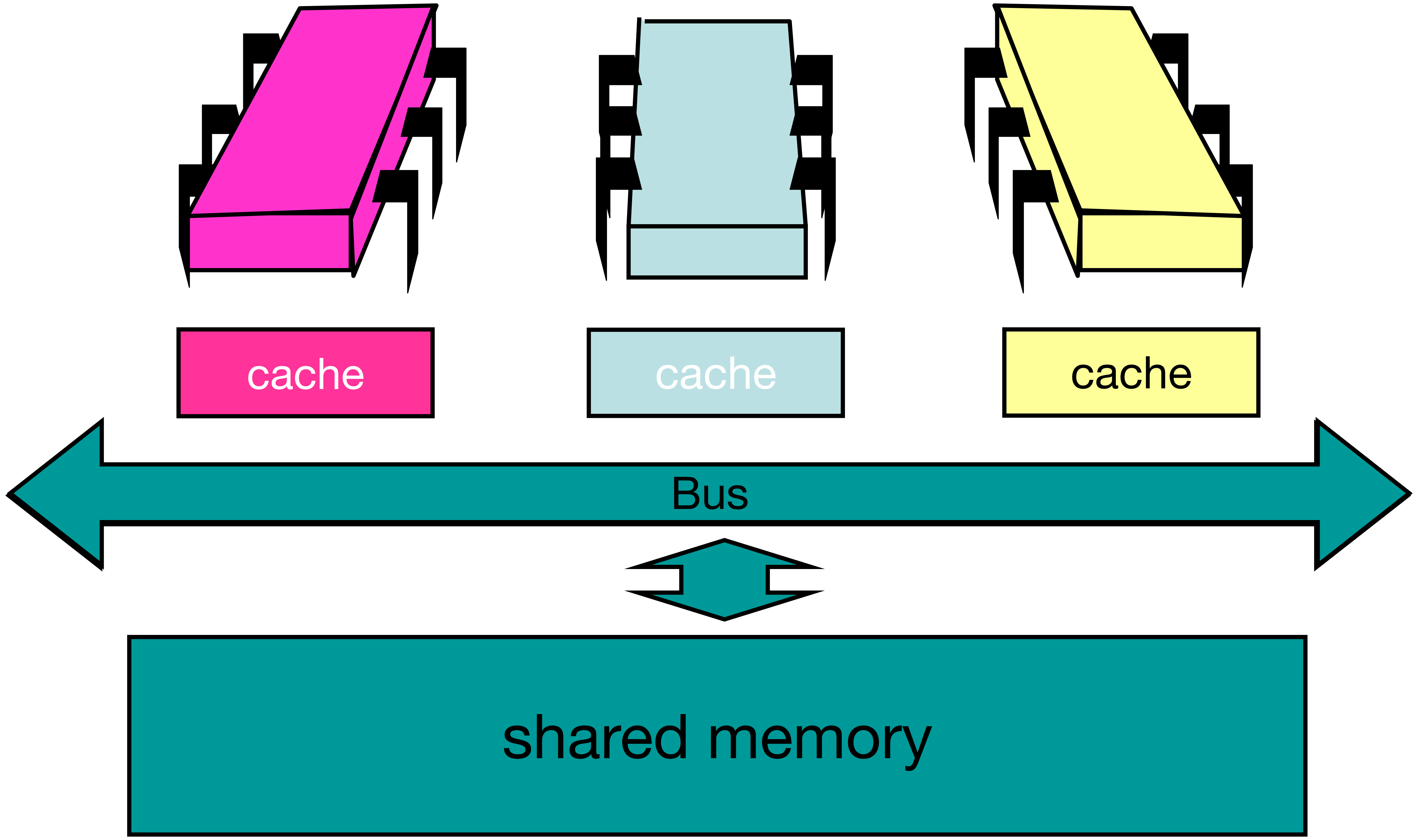
The number of transistors on an integrated circuit doubles roughly every 18 to 24 months, resulting in exponential growth in computing power.

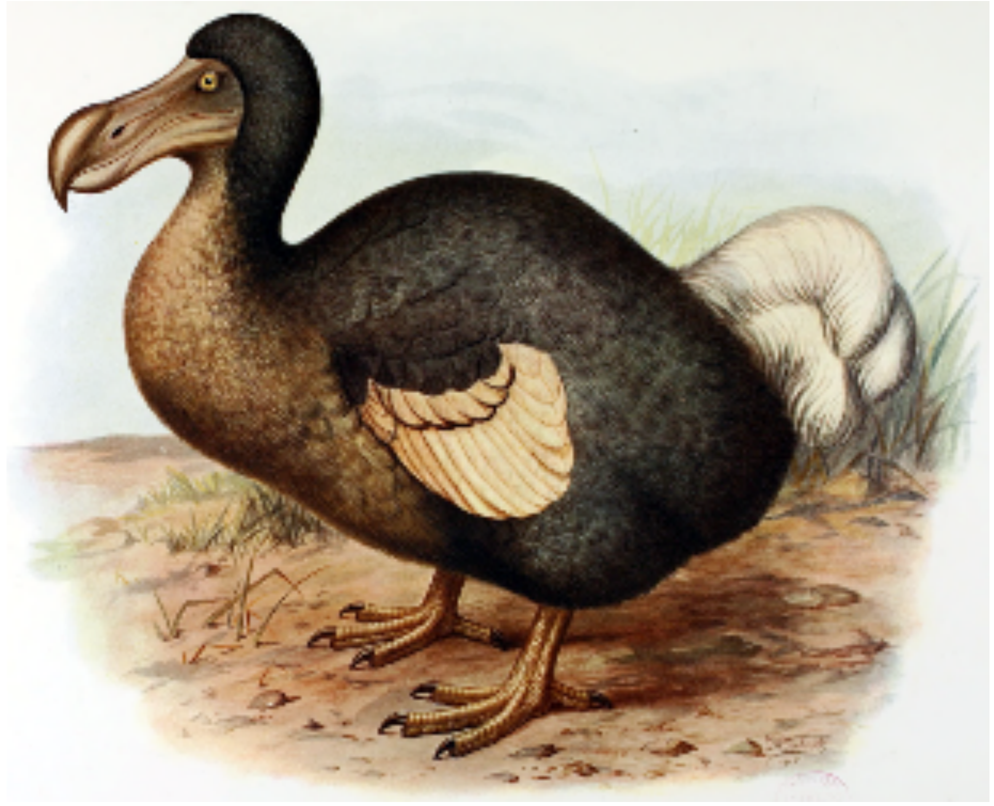


Uniprocessor (Extinct)

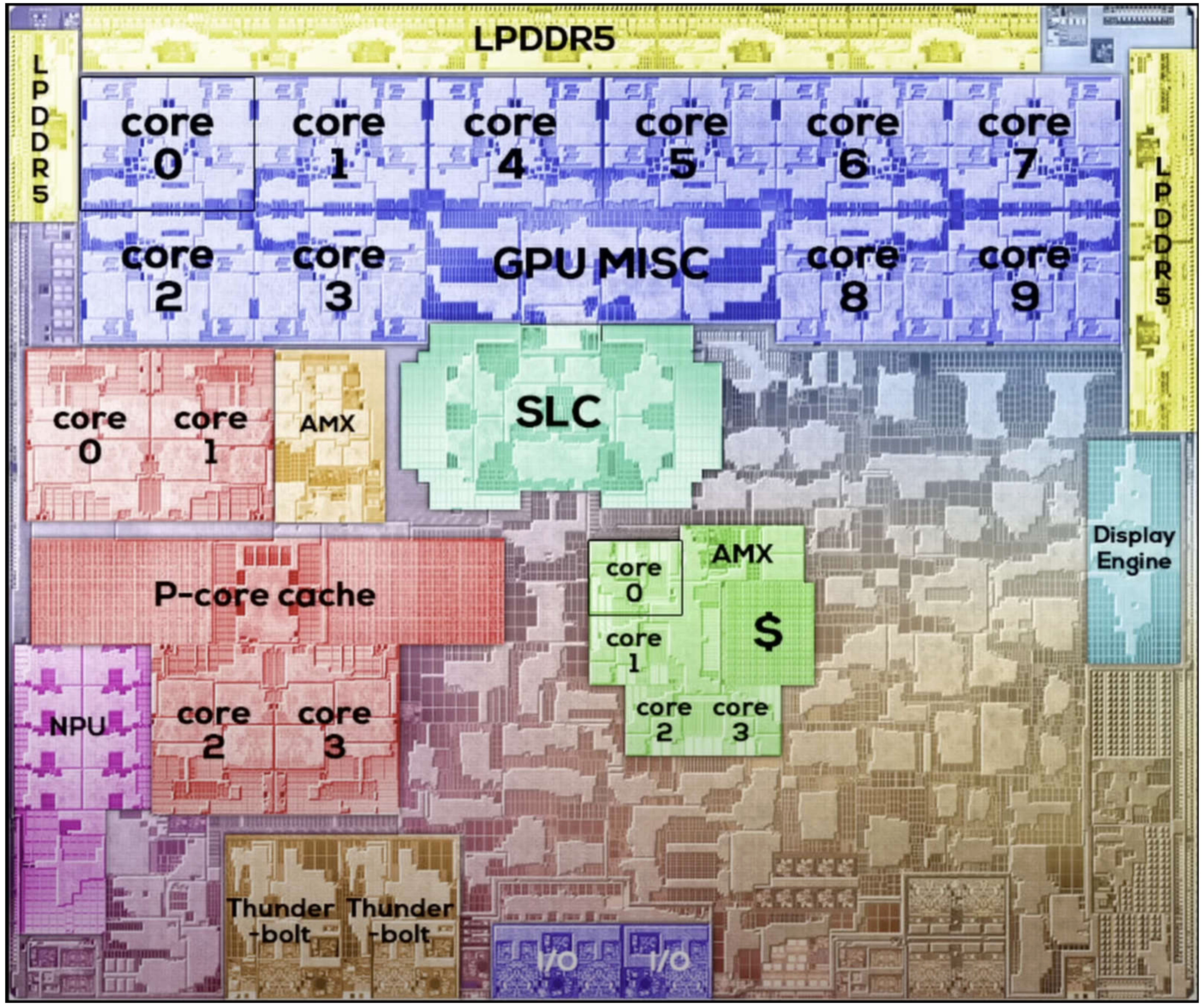


Shared Memory Multiprocessor (SMP): Extinct





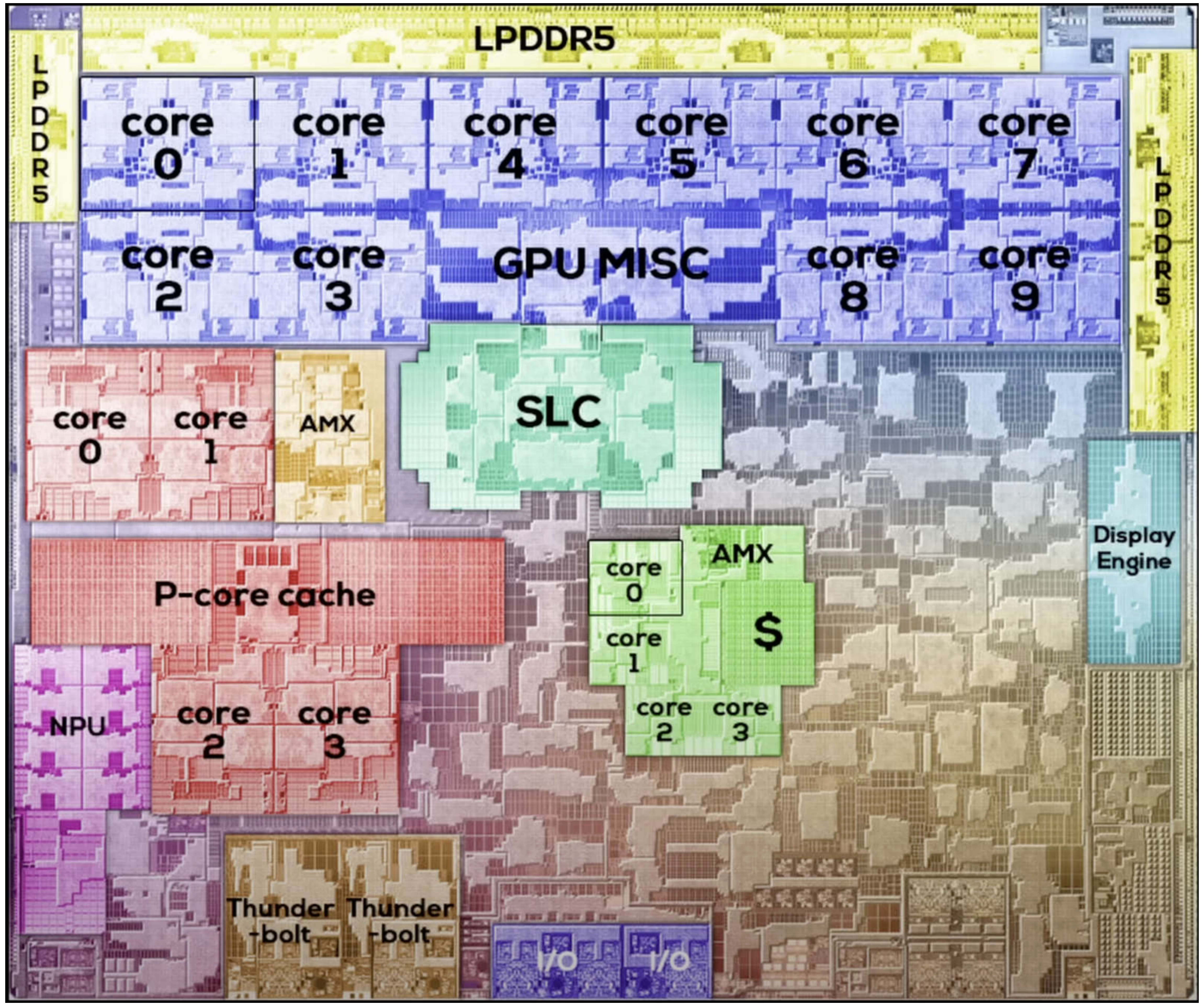
Modern Multicore Processors



Apple M3 floor plan/die shot

Modern Multicore Processors

- Various names for this
  - System-on-a-chip (SoC)
  - Chip multiprocessor (CMP)
  - A multicore machine

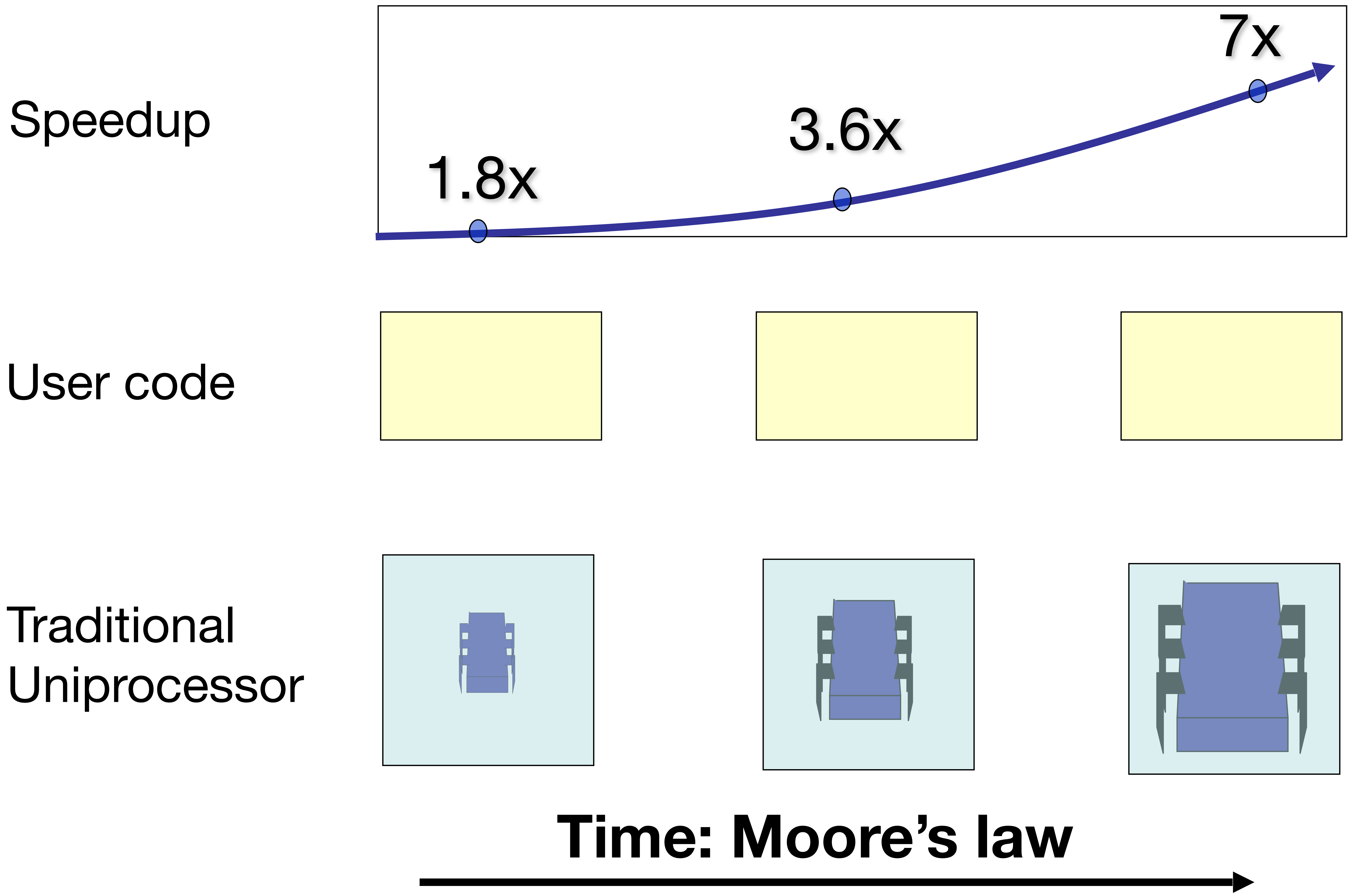


Apple M3 floor plan/die shot

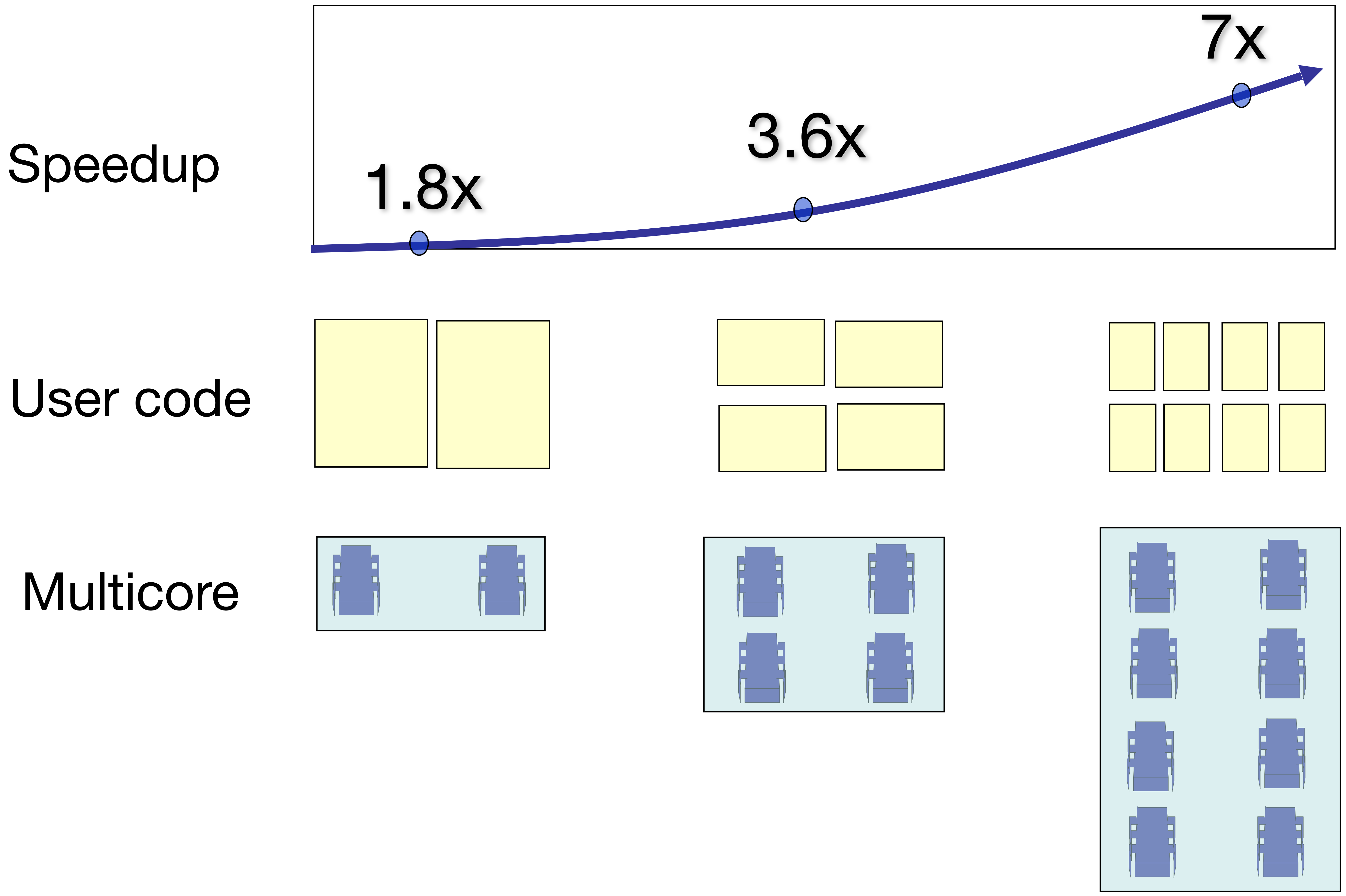
Why do we care?

- 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

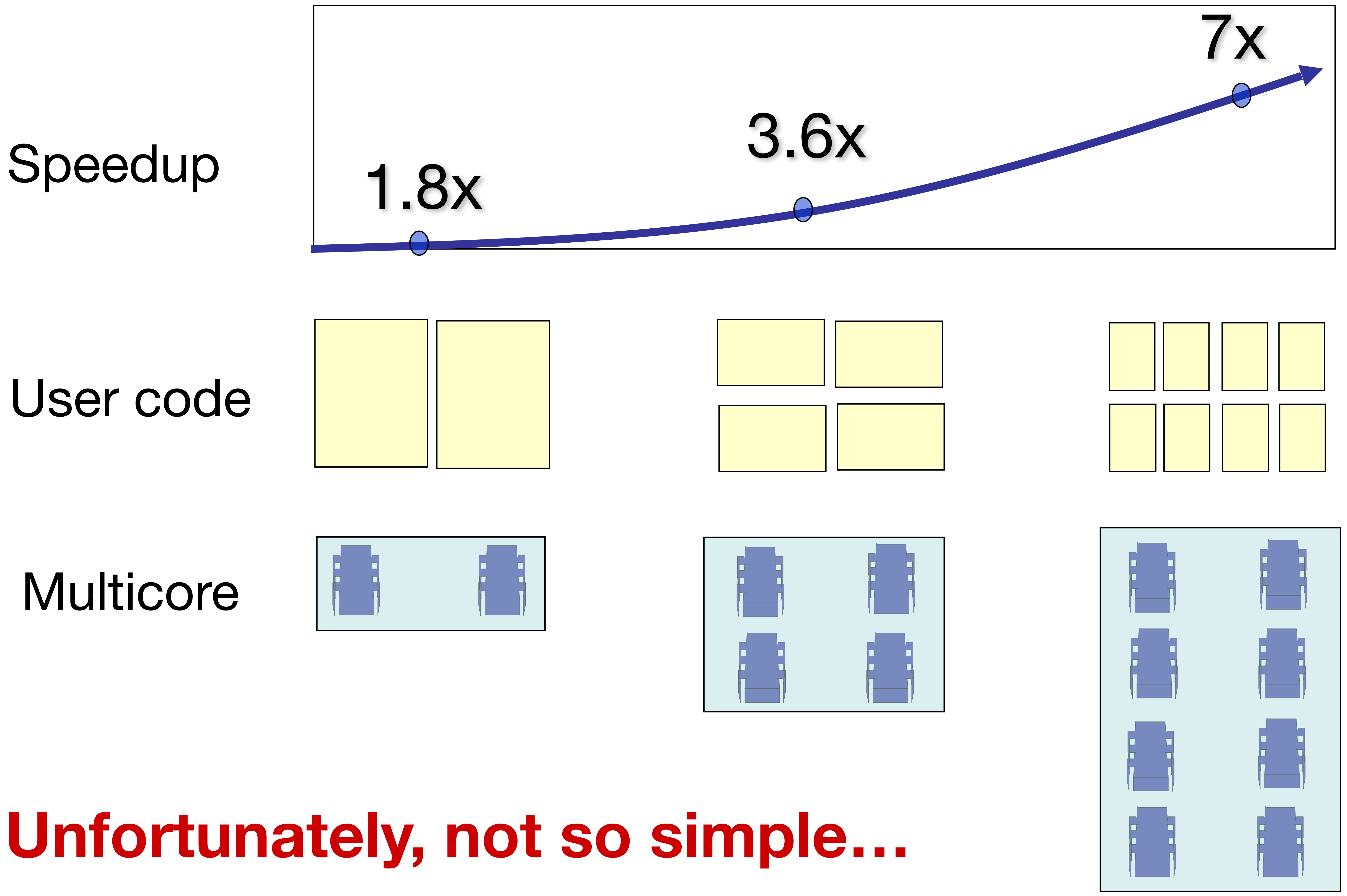
Traditional Scaling Process



Ideal Scaling Process

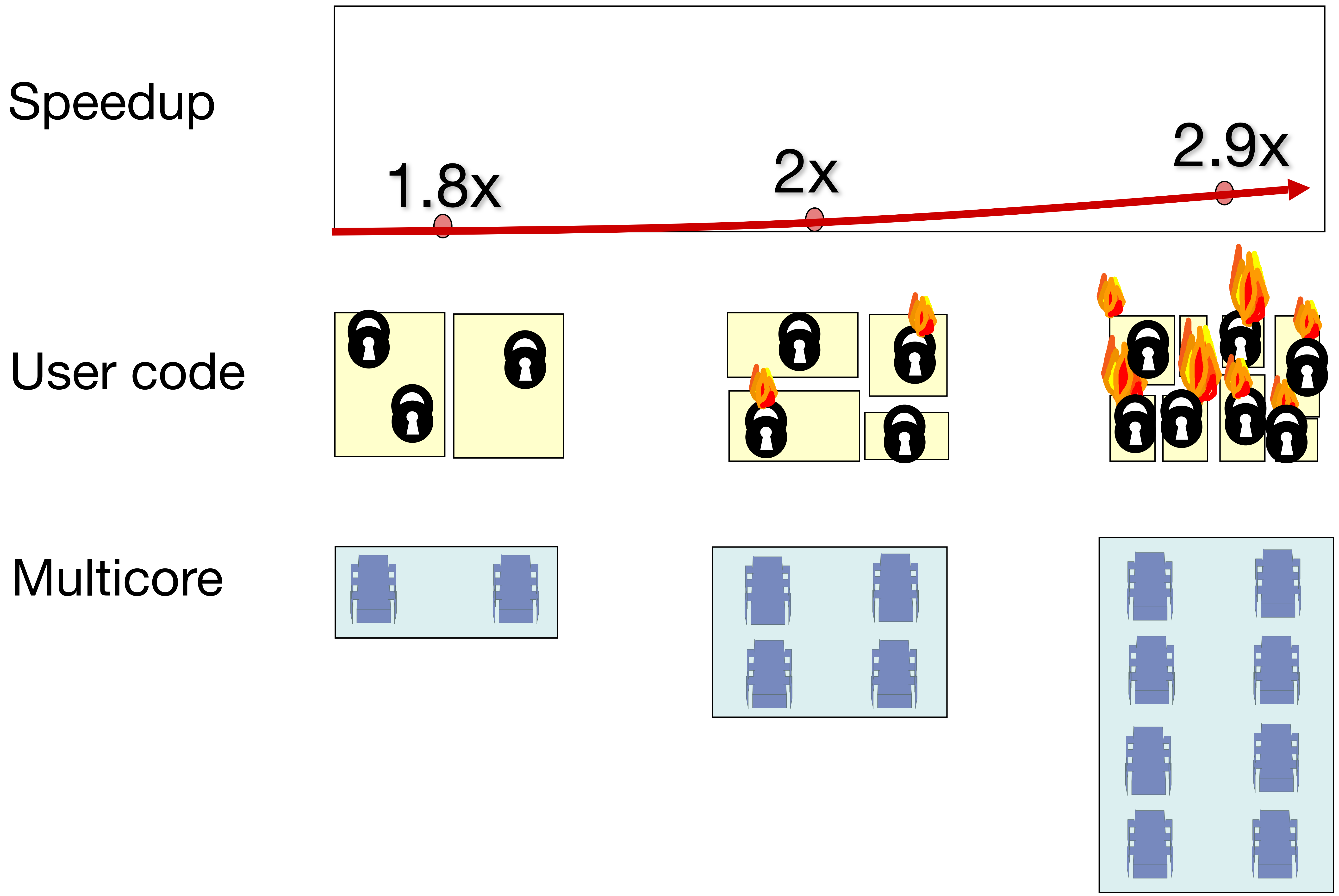


Ideal Scaling Process

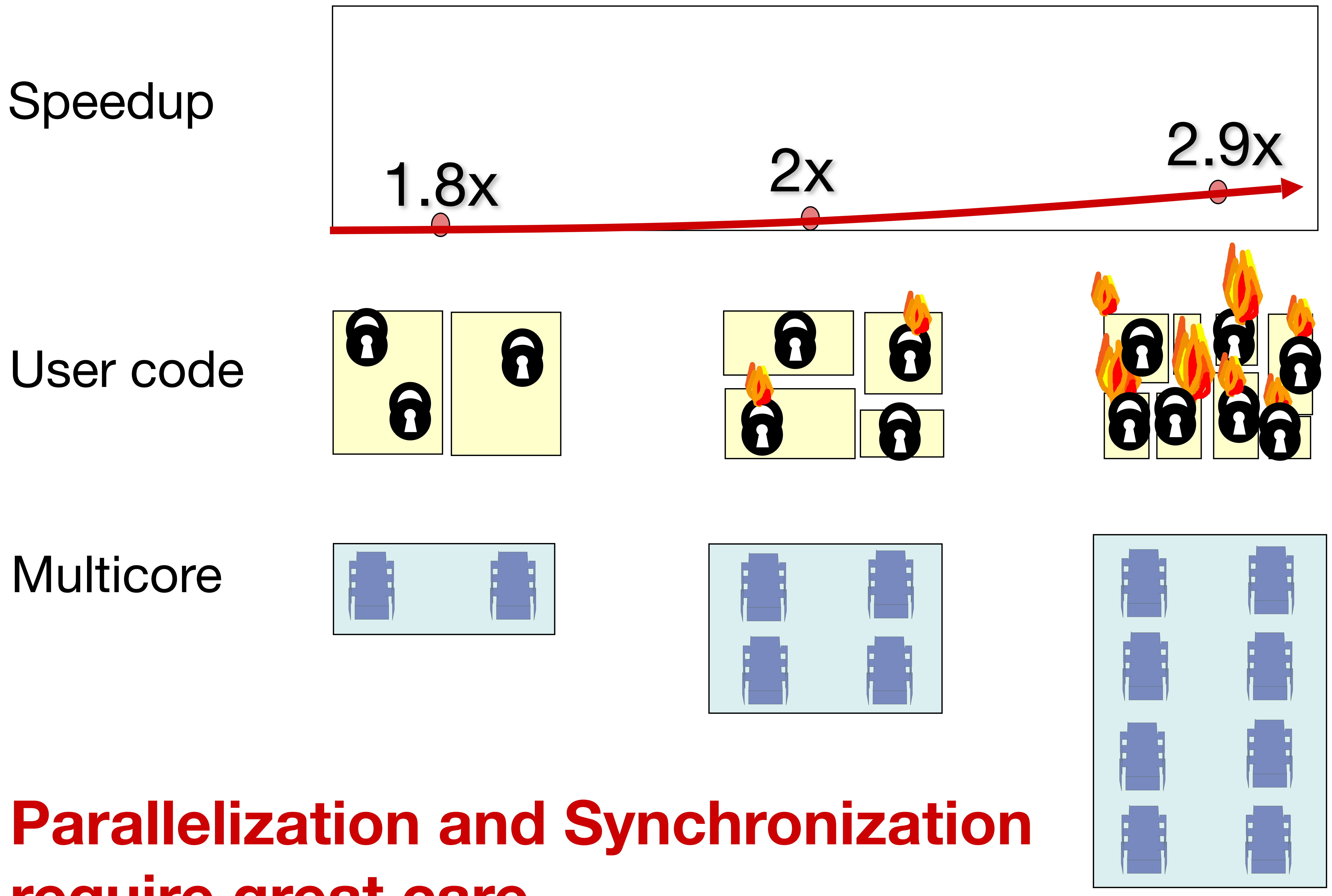


Unfortunately, not so simple…

Actual Scaling Process

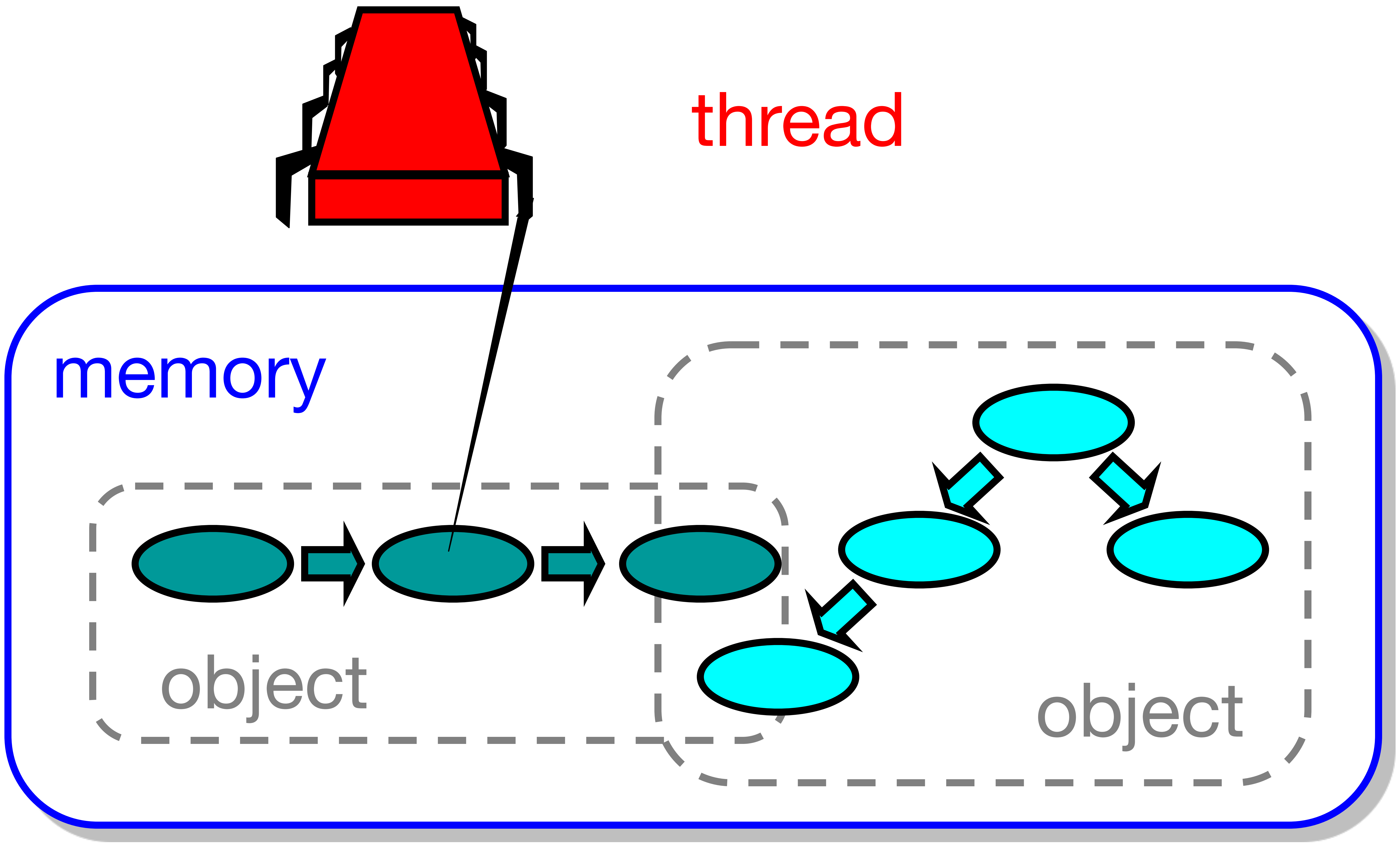


Actual Scaling Process

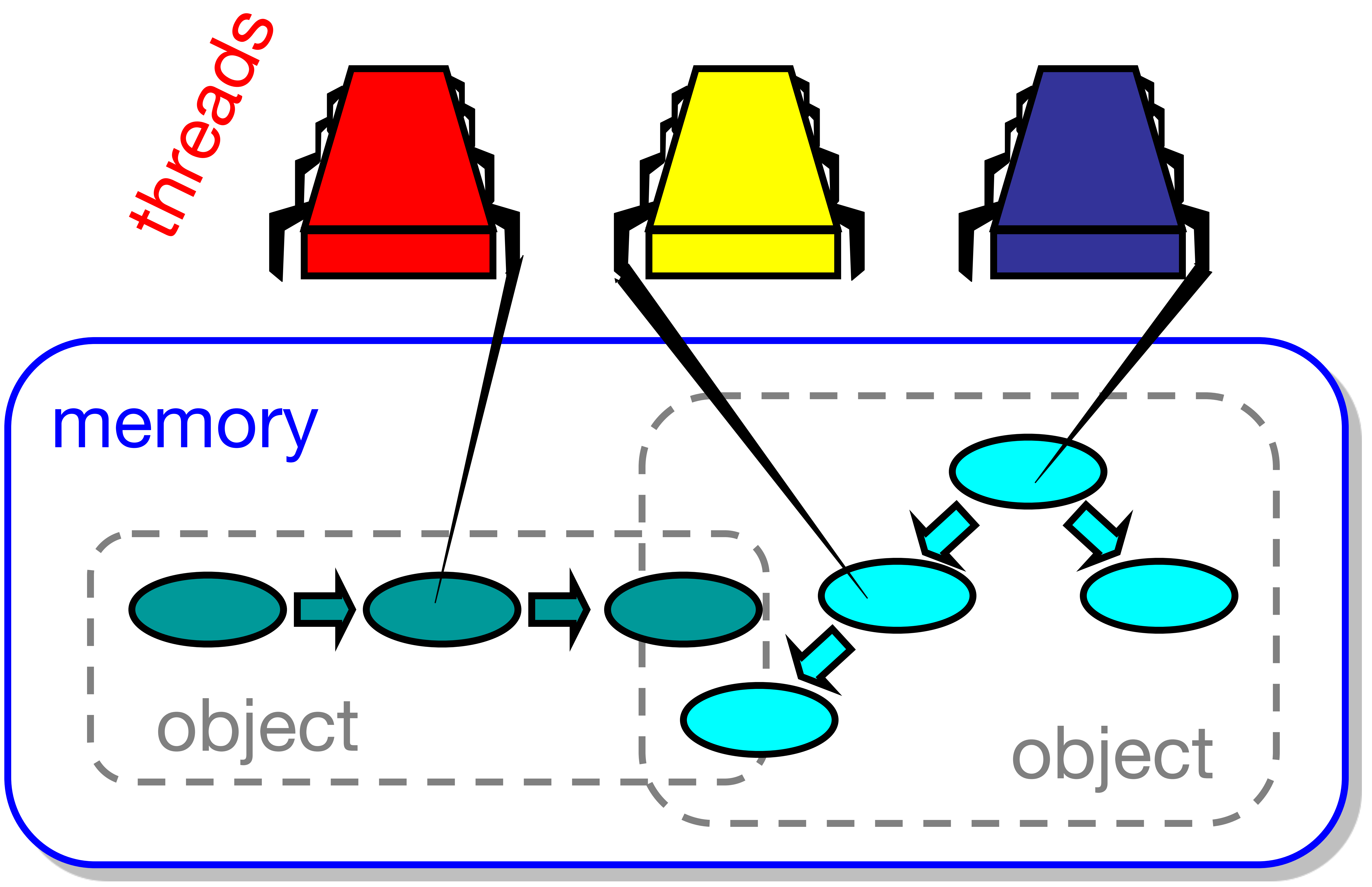


Parallelization and Synchronization require great care…

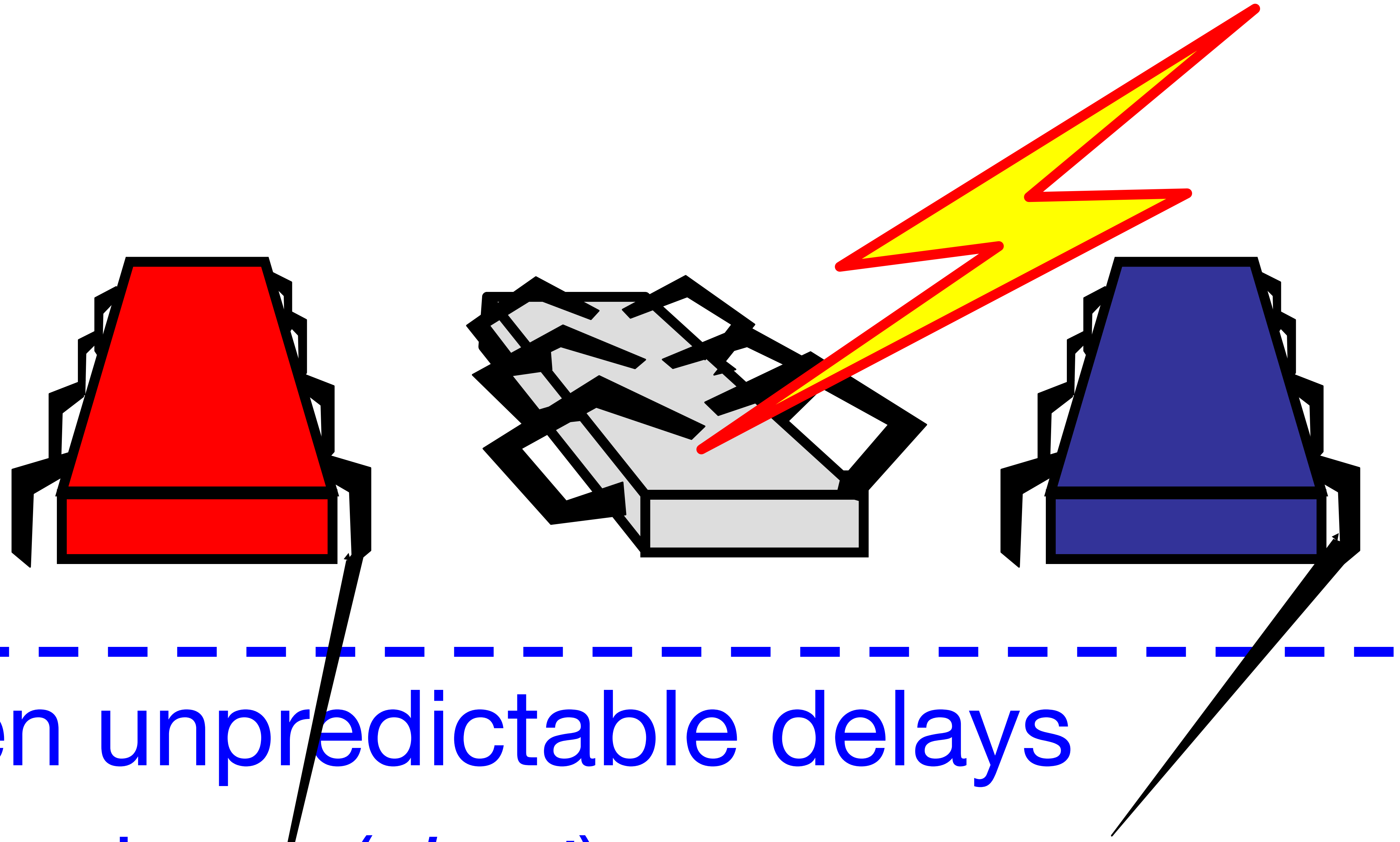
Sequential Computation



Concurrent Computation



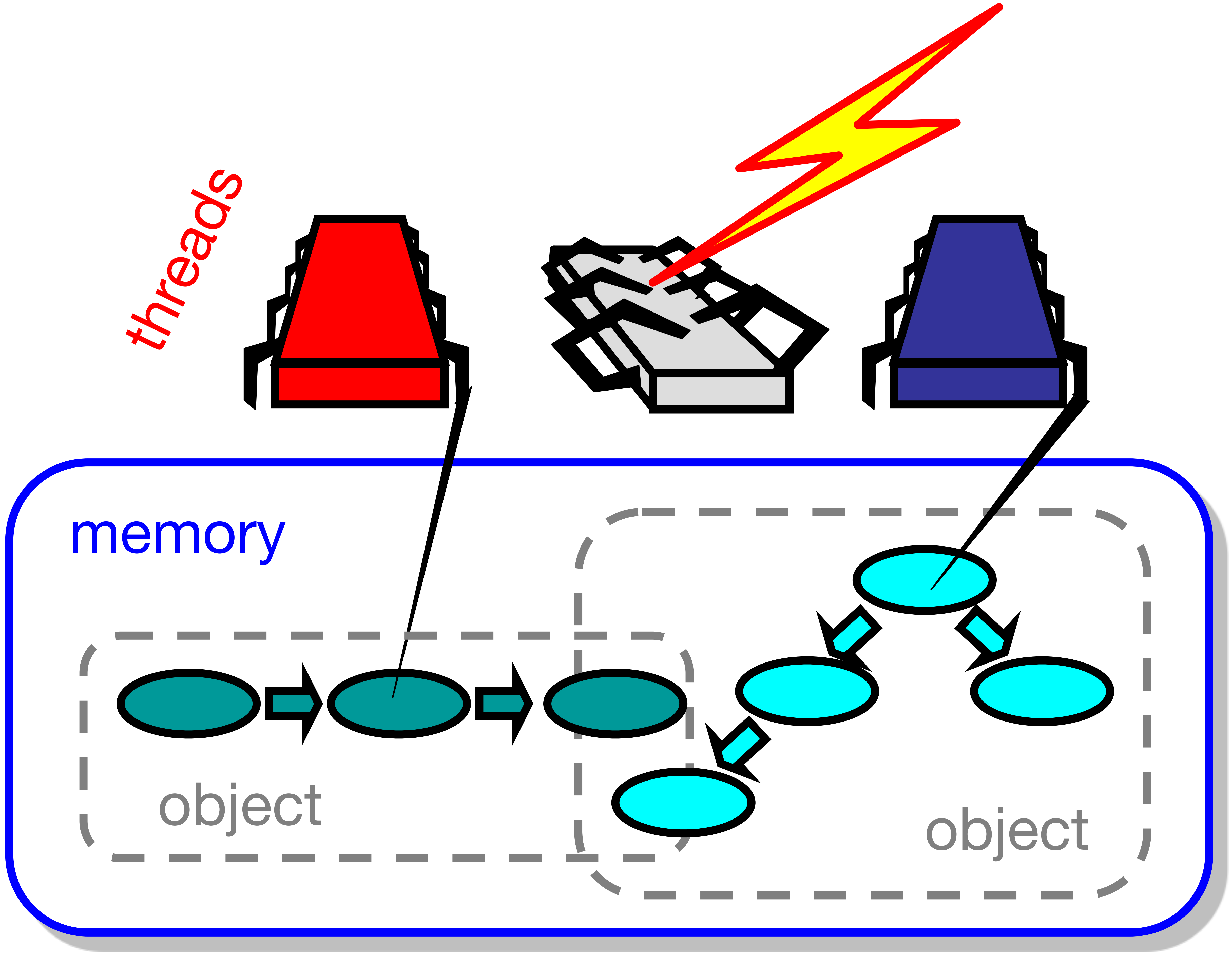
Asynchrony



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

Model Summary

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



Roadmap for the Parallelism side

Roadmap for the Parallelism side

- We are going to focus on ***principles*** first and then ***practice***
  - Start with ***idealised*** models of concurrent computations
  - Look at ***simplistic*** problems
  - Emphasise ***correctness*** over ***pragmatism***

Roadmap for the Parallelism side

- We are going to focus on ***principles*** first and then ***practice***
  - Start with ***idealised*** models of concurrent computations
  - Look at ***simplistic*** problems
  - Emphasise ***correctness*** over ***pragmatism***
- ***Principles*** will be foundational for the concurrency parts that we will study later

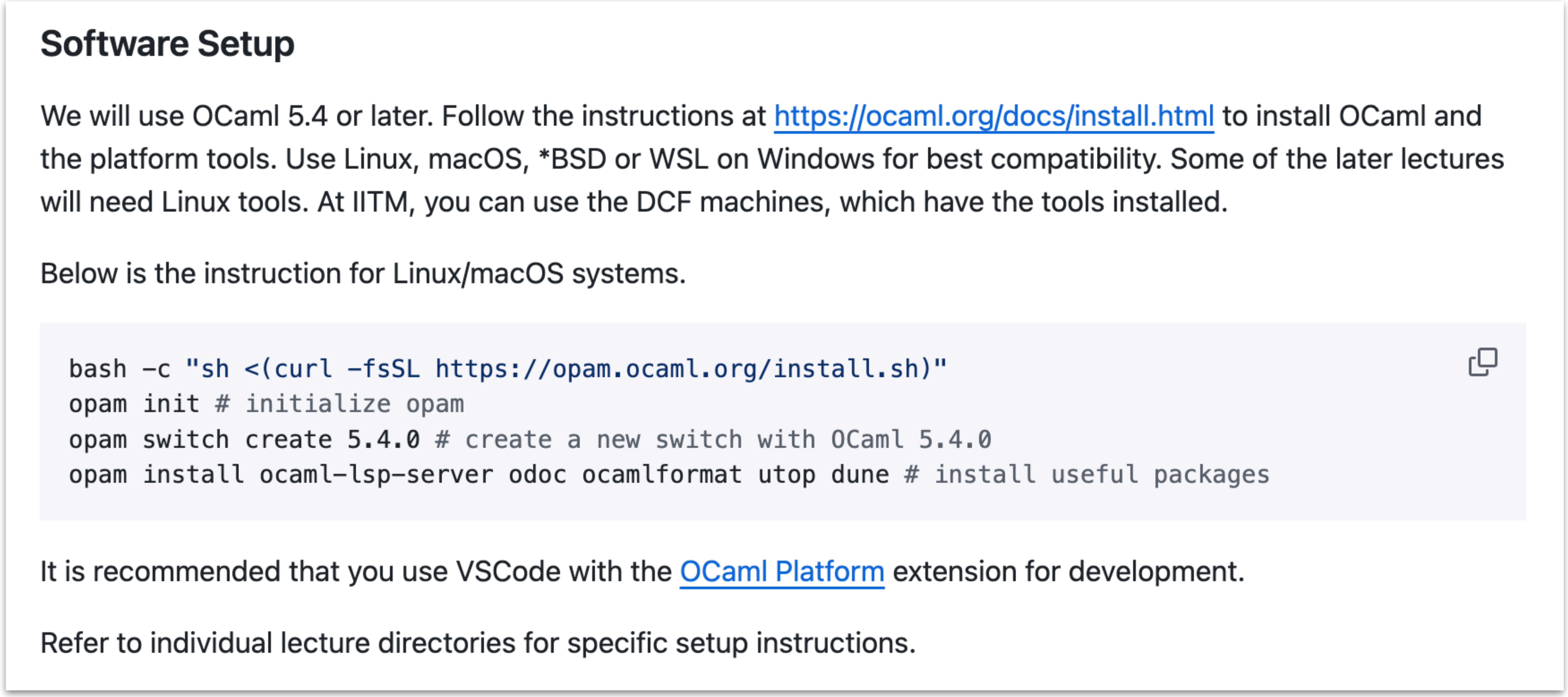
Roadmap for the Parallelism side

- We are going to focus on ***principles*** first and then ***practice***
  - Start with ***idealised*** models of concurrent computations
  - Look at ***simplistic*** problems
  - Emphasise ***correctness*** over ***pragmatism***
- ***Principles*** will be foundational for the concurrency parts that we will study later
- "Correctness may be theoretical, but incorrectness has practical impact"

Designing Concurrent Programs

Software Setup

- OCaml 5.4
- Use vscode as default
  - Comes with Vim and Emacs modes
- Use OCaml vscode platform
- More instructions on the course GitHub page



Software Setup

We will use OCaml 5.4 or later. Follow the instructions at https://ocaml.org/docs/install.html to install OCaml and the platform tools. Use Linux, macOS, *BSD or WSL on Windows for best compatibility. Some of the later lectures will need Linux tools. At IITM, you can use the DCF machines, which have the tools installed.

Below is the instruction for Linux/macOS systems.

```
bash -c "sh <(curl -fsSL https://opam.ocaml.org/install.sh)"
opam init # initialize opam
opam switch create 5.4.0 # create a new switch with OCaml 5.4.0
opam install ocaml-lsp-server odoc ocamlformat utop dune # install useful packages
```

It is recommended that you use VSCode with the OCaml Platform extension for development.

Refer to individual lecture directories for specific setup instructions.

Concurrency Jargon

- Hardware
  - Processors

Concurrency Jargon

- Hardware
  - Processors
- Software
  - Threads/Processes/“Domains”

Concurrency Jargon

- Hardware
  - Processors
- Software
  - Threads/Processes/“Domains”
- Domains are unit of parallel execution in OCaml
  - Spawn and join domains using `Domain.spawn` and `Domain.join`
  - See also
    - Domain module in OCaml Standard library https://ocaml.org/manual/5.4/api/Domain.html
    - Parallelism Chapter in OCaml manual https://ocaml.org/manual/5.1/parallelism.html

Concurrency Jargon

- Hardware
  - Processors
- Software
  - Threads/Processes/“Domains”
- Domains are unit of parallel execution in OCaml
  - Spawn and join domains using `Domain.spawn` and `Domain.join`
  - See also
    - Domain module in OCaml Standard library https://ocaml.org/manual/5.4/api/Domain.html
    - Parallelism Chapter in OCaml manual https://ocaml.org/manual/5.1/parallelism.html
- Sometimes ok to confuse them, sometimes not

Recursive Fibonacci in Parallel

- Challenge
    - Compute recursive fib(N) twice
- Embarrassingly parallel computation
    - No dependencies between the two calls
- Use Domains to parallelise the computation

Recursive Fibonacci in Parallel

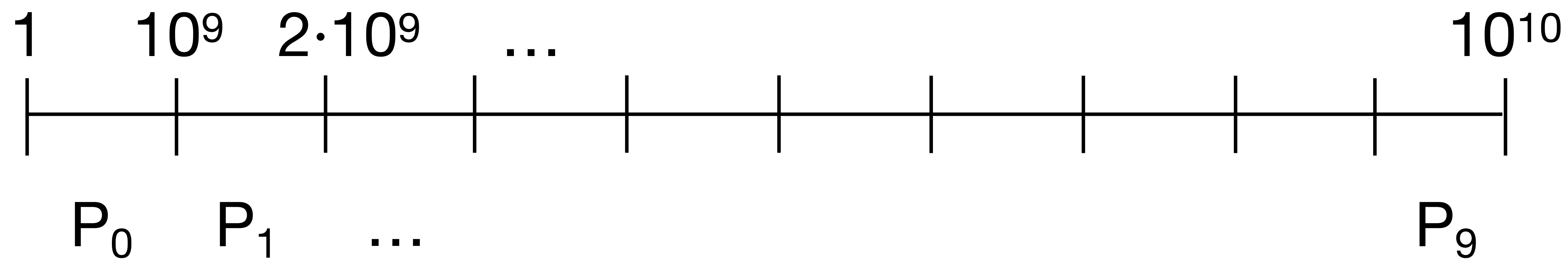
- Challenge
  - Compute recursive fib(N) twice
- Embarrassingly parallel computation
  - No dependencies between the two calls
- Use Domains to parallelise the computation

Demo

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

```
let is_prime n =
  if n <= 1 then false
  else if n = 2 then true
  else if n mod 2 = 0 then false
  else
    let rec check_divisor d =
      (* If n has a divisor d > sqrt(n), then it must
         also have a corresponding divisor n/d < sqrt(n). *)
      if d * d > n then true
      else if n mod d = 0 then false
      else check_divisor (d + 2)
    in
    check_divisor 3

let print_primes_in_range start_range end_range =
  for i = start_range to end_range do
    if is_prime i then
      Printf.printf "%d\n" i
  done
```

Procedure for thread i

```
let is_prime n =
  if n <= 1 then false
  else if n = 2 then true
  else if n mod 2 = 0 then false
  else
    let rec check_divisor d =
      (* If n has a divisor d > sqrt(n), then it must
         also have a corresponding divisor n/d < sqrt(n). *)
      if d * d > n then true
      else if n mod d = 0 then false
      else check_divisor (d + 2)
    in
    check_divisor 3

let print_primes_in_range start_range end_range =
  for i = start_range to end_range do
    if is_prime i then
      Printf.printf "%d\n" i
  done
```

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

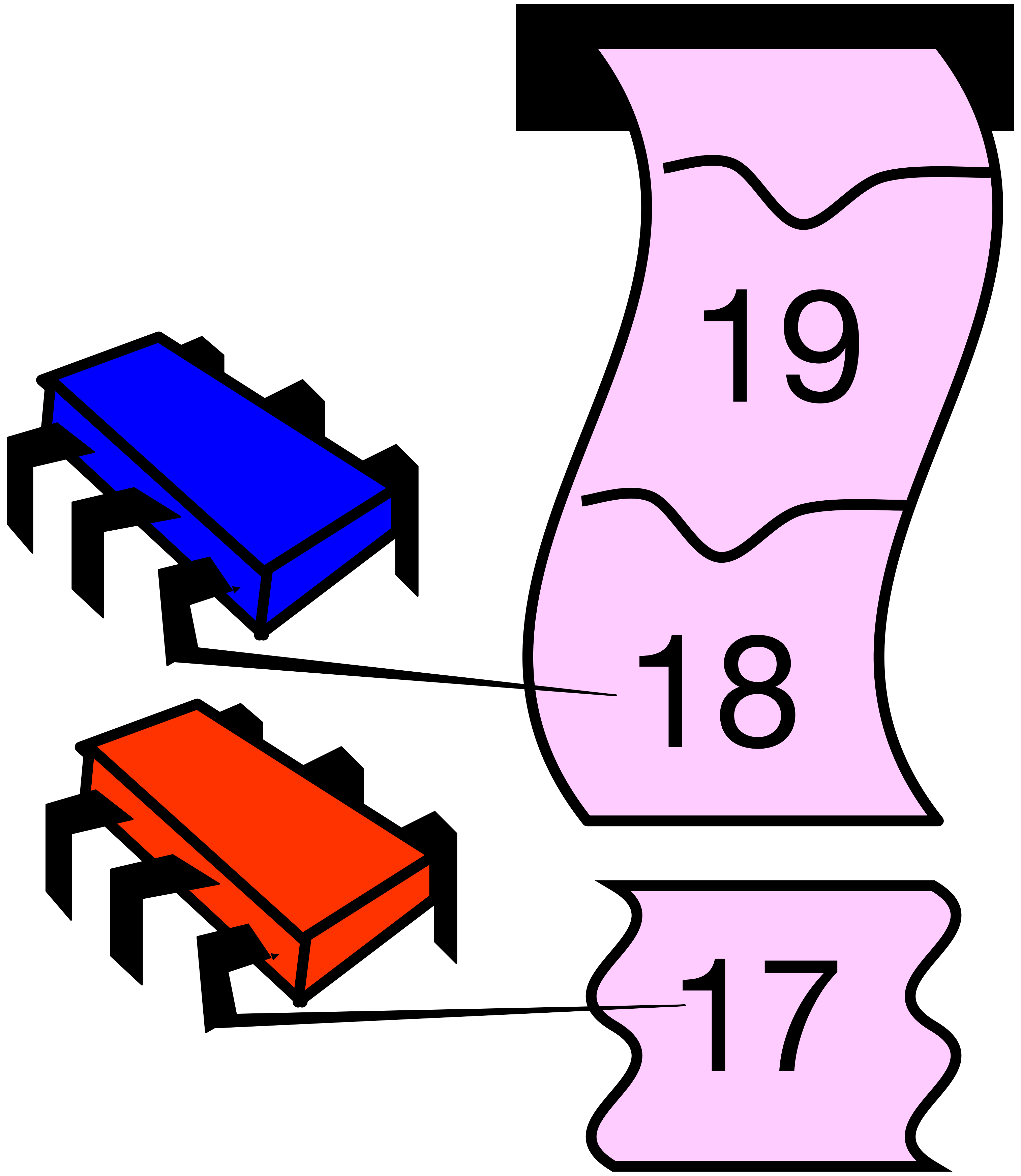
Procedure for thread i

```
let is_prime n =
  if n <= 1 then false
  else if n = 2 then true
  else if n mod 2 = 0 then false
  else
    let rec check_divisor d =
      (* If n has a divisor d > sqrt(n), then it must
         also have a corresponding divisor n/d < sqrt(n). *)
      if d * d > n then true
      else if n mod d = 0 then false
      else check_divisor (d + 2)
    in
    check_divisor 3

let print_primes_in_range start_range end_range =
  for i = start_range to end_range do
    if is_prime i then
      Printf.printf "%d\n" i
  done
```

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

Demo

Shared counter



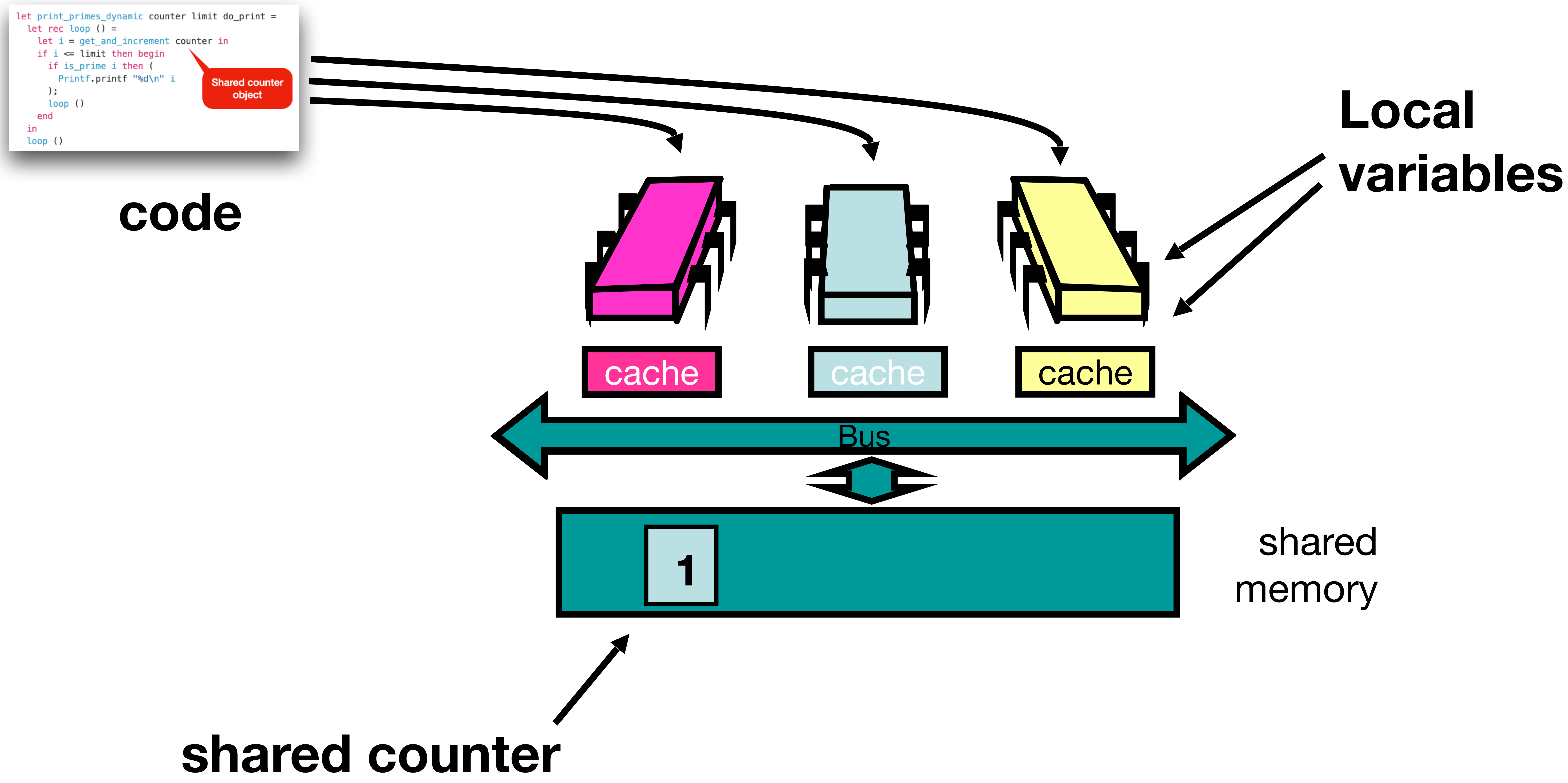
each thread
takes a number

Procedure for thread i

```
let print_primes_dynamic counter limit do_print =
  let rec loop () =
    let i = get_and_increment counter in
    if i <= limit then begin
      if is_prime i then (
        Printf.printf "%d\n" i
      );
      loop ()
    end
  in
  loop ()
```

Shared counter object

Where things reside



Procedure for thread i

```
let print_primes_dynamic counter limit do_print =
  let rec loop () =
    let i = get_and_increment counter in
    if i <= limit then begin
      if is_prime i then (
        Printf.printf "%d\n" i
      );
      loop ()
    end
  in
  loop ()
```

Stop when every value examined

Shared counter object

Counter Implementation

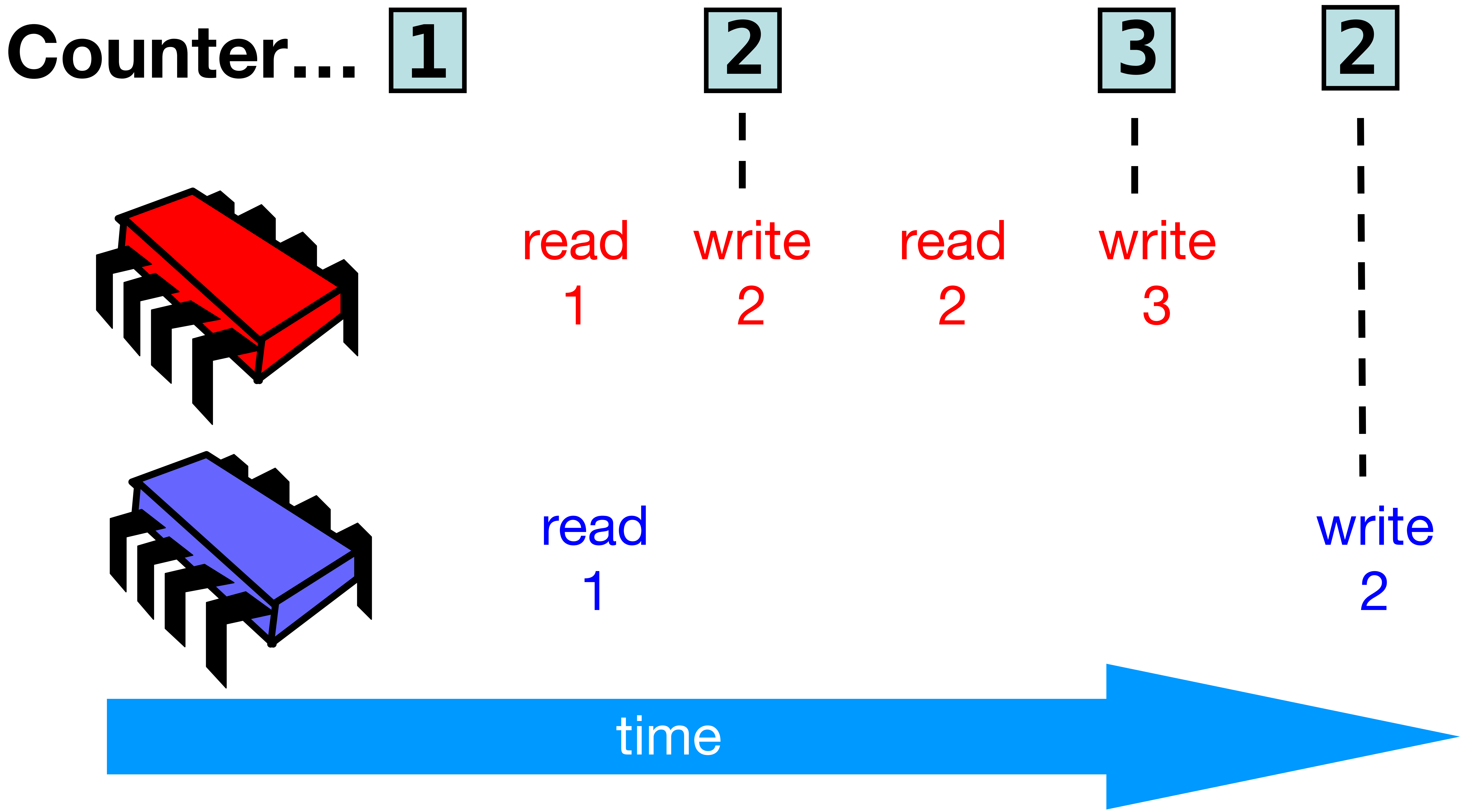
```
(* A counter is just a reference *)
let create_counter initial_value =
  ref initial_value

let get_and_increment counter =
  let v = !counter in
  counter := v + 1;
  v

let print_primes_dynamic counter limit do_print =
  let rec loop () =
    let i = get_and_increment counter in
    if i <= limit then begin
      if is_prime i then (
        Printf.printf "%d\n" i
      );
      loop ()
    end
  in
  loop ()
```

Counter Implementation

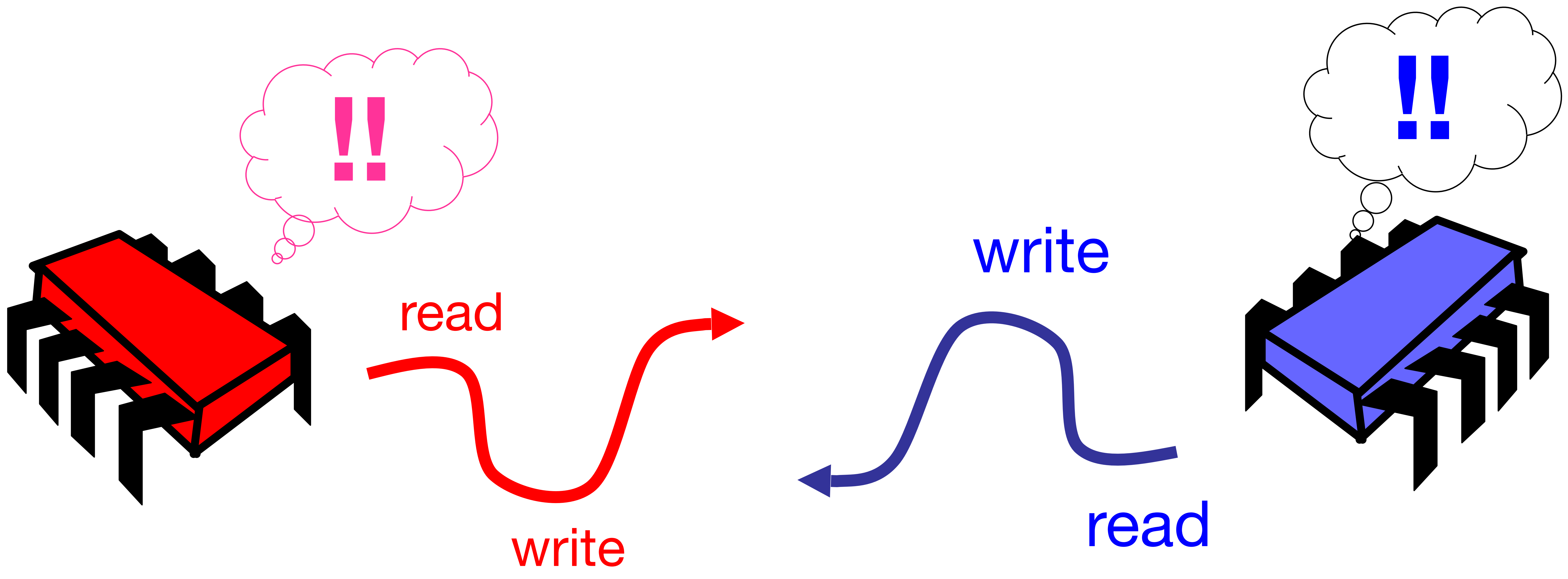
```
(* A counter is just a reference *)
let create_counter initial_value =
  ref initial_value

let get_and_increment counter =
  let v = !counter in
  counter := v + 1;
  v

let print_primes_dynamic counter limit do_print =
  let rec loop () =
    let i = get_and_increment counter in
    if i <= limit then begin
      if is_prime i then (
        Printf.printf "%d\n" i
      );
      loop ()
    end
  in
  loop ()
```

OK for single thread,
not for concurrent threads

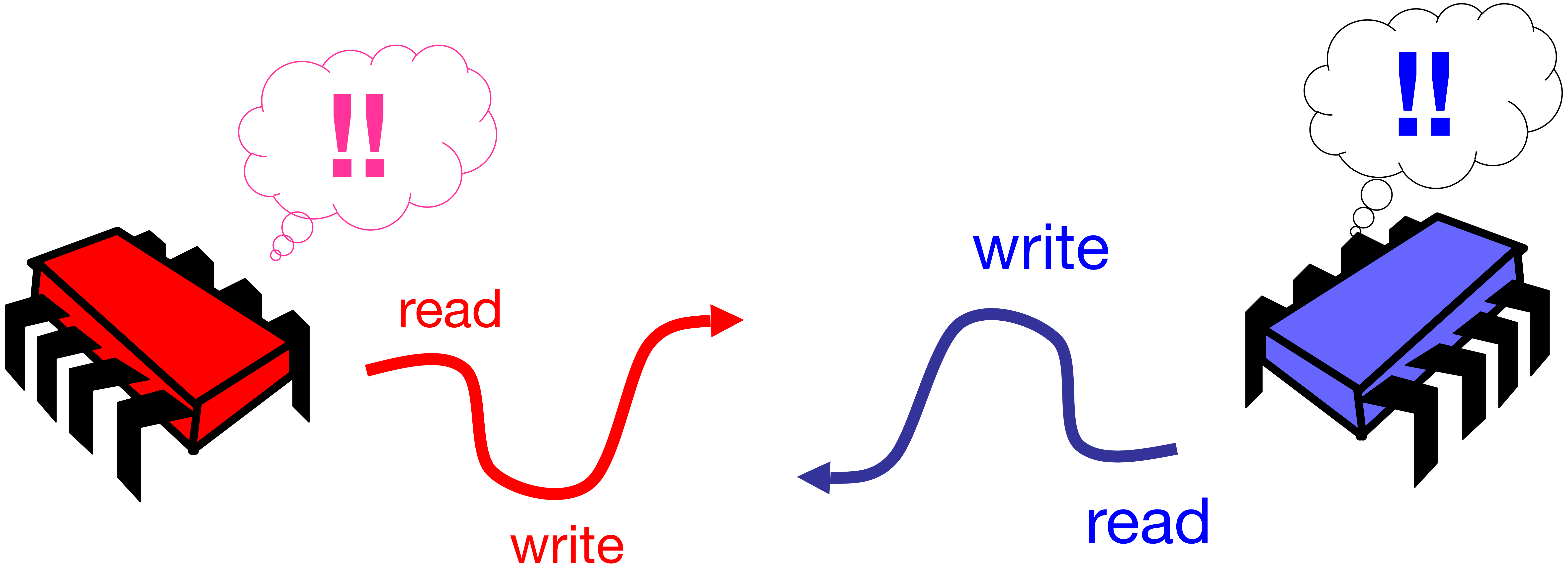
Not so good …



Is this problem inherent?



Is this problem inherent?



If we could only glue reads and writes together…

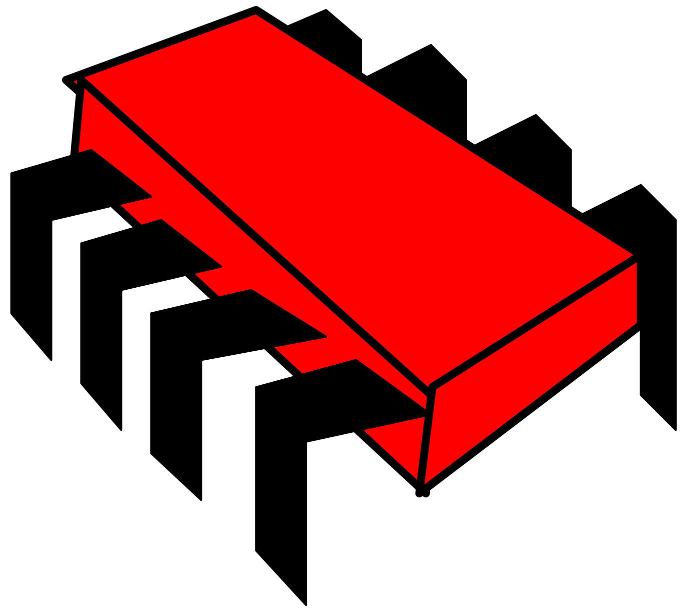
Challenge

```
(* A counter is just a reference *)
let create_counter initial_value =
  ref initial_value

let get_and_increment counter =
  let v = !counter in
  counter := v + 1;
  v
```

Make this atomic (indivisible)

Hardware solution

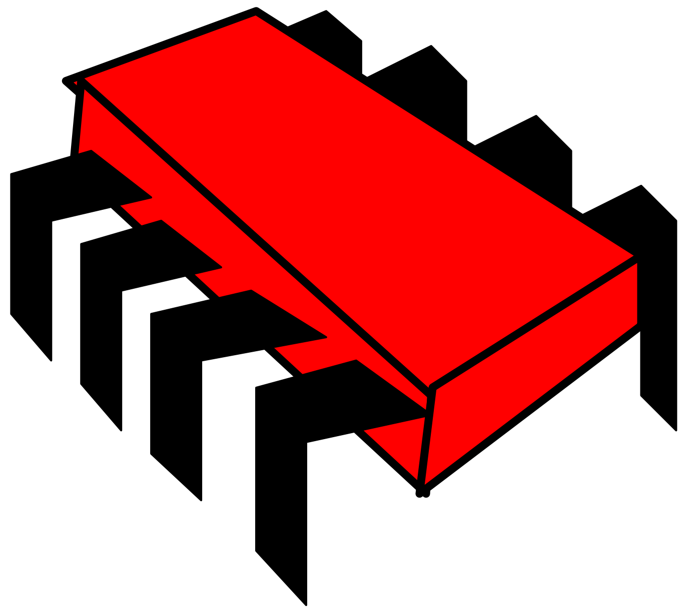
```
(* A counter is just a reference *)
let create_counter initial_value =
  ref initial_value

let get_and_increment counter =
  let v = !counter in
  counter := v + 1;
  v
```



ReadModifyWrite instruction

OCaml 5 solution

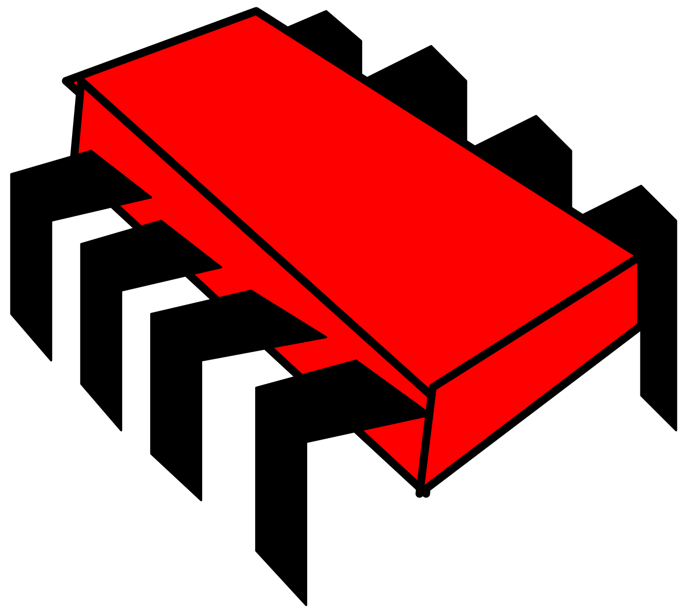
```
(* Thread-safe counter using atomic operations *)
let create_counter initial_value =
  Atomic.make initial_value

let get_and_increment counter =
  Atomic.fetch_and_add counter 1
```



ReadModifyWrite instruction

OCaml 5 solution

```
(* Thread-safe counter using atomic operations *)
let create_counter initial_value =
  Atomic.make initial_value

let get_and_increment counter =
  Atomic.fetch_and_add counter 1
```



ReadModifyWrite instruction

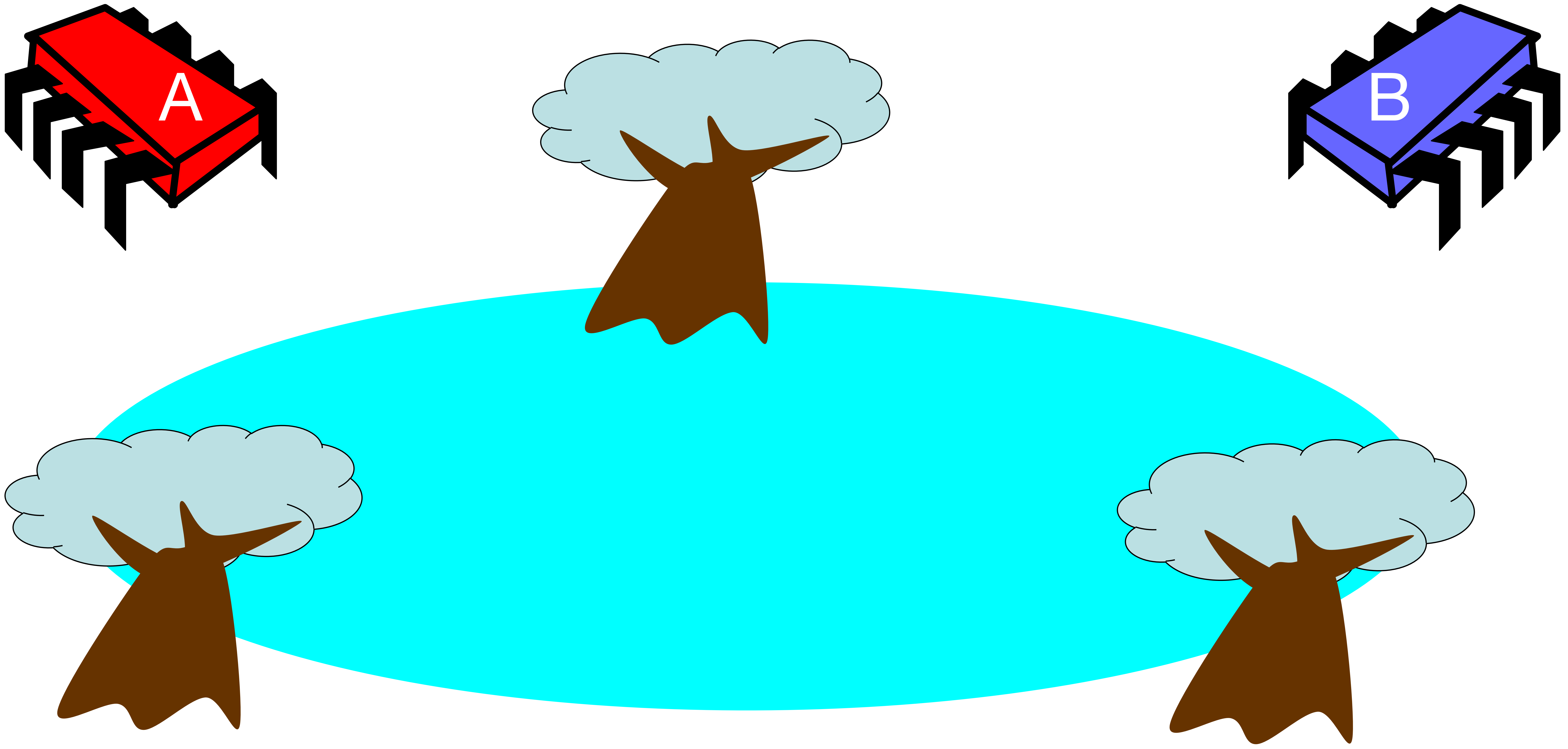
Demo

OCaml 5 solution (2)

```
let create_counter initial_value =
  (ref initial_value, Mutex.create ())

let get_and_increment (counter_ref, mutex) =
  Mutex.lock mutex;
  let value = !counter_ref in
  counter_ref := value + 1;
  Mutex.unlock mutex;
  value
```

Mutual Exclusion

Mutual Exclusion (or Alice and Bob share a pond)



Alice has a pet

Bob has a pet

The Problem

Formalising the Problem

Formalising the Problem

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

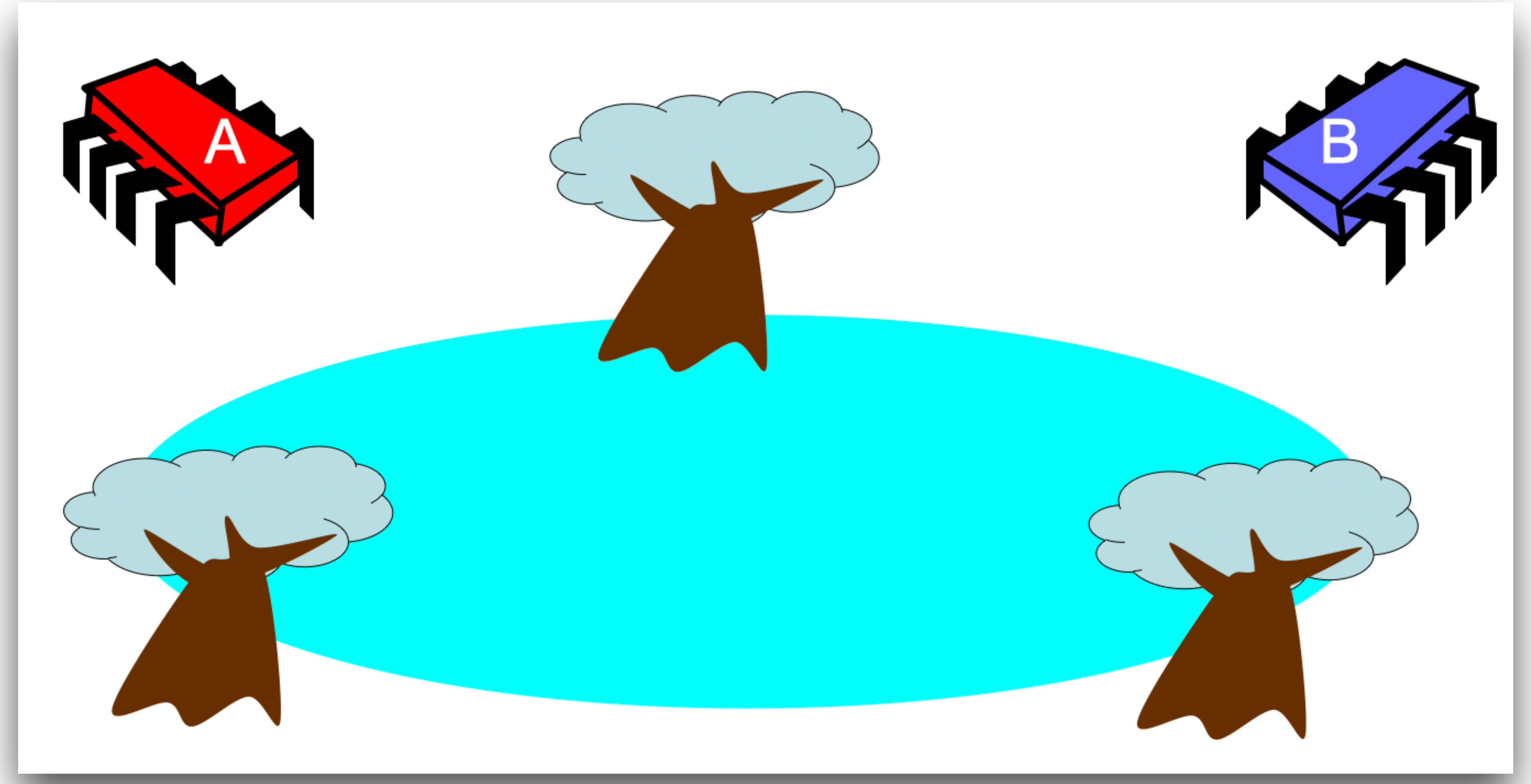
Formalising the Problem

- Two types of formal properties in asynchronous computation:
  - ***Safety*** Properties — Nothing bad happens ever
  - ***Liveness*** Properties — Something good happens eventually
- **Mutual Exclusion**
  - Both pets never in pond simultaneously
  - This is a ***safety*** property

Formalising the Problem

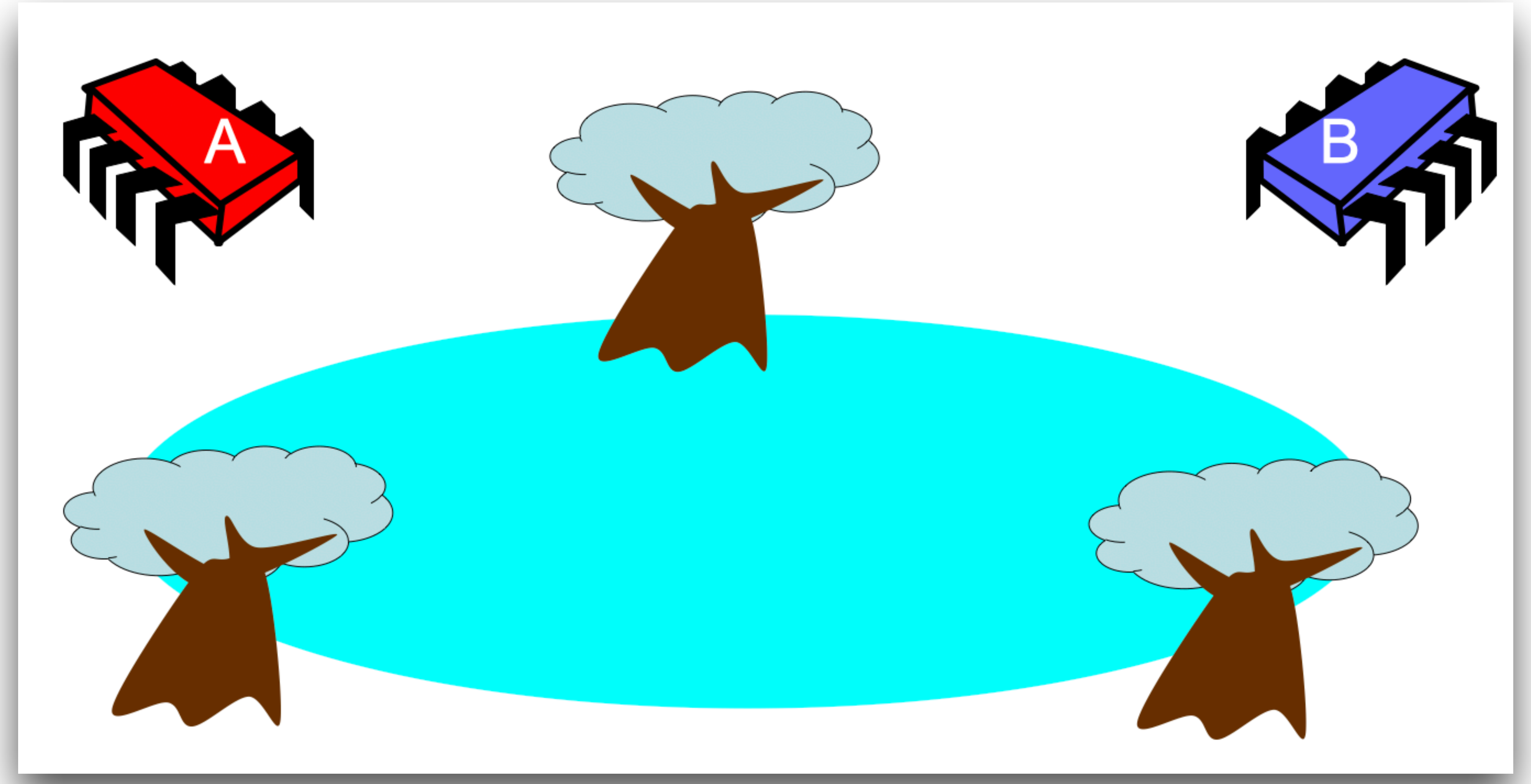
- Two types of formal properties in asynchronous computation:
  - ***Safety*** Properties — Nothing bad happens ever
  - ***Liveness*** Properties — Something good happens eventually
- **Mutual Exclusion**
  - Both pets never in pond simultaneously
  - This is a ***safety*** property
- **No Deadlock**
  - if only one wants in, it gets in
  - if both want in, one gets in.
  - This is a ***liveness*** property

Simple Protocol



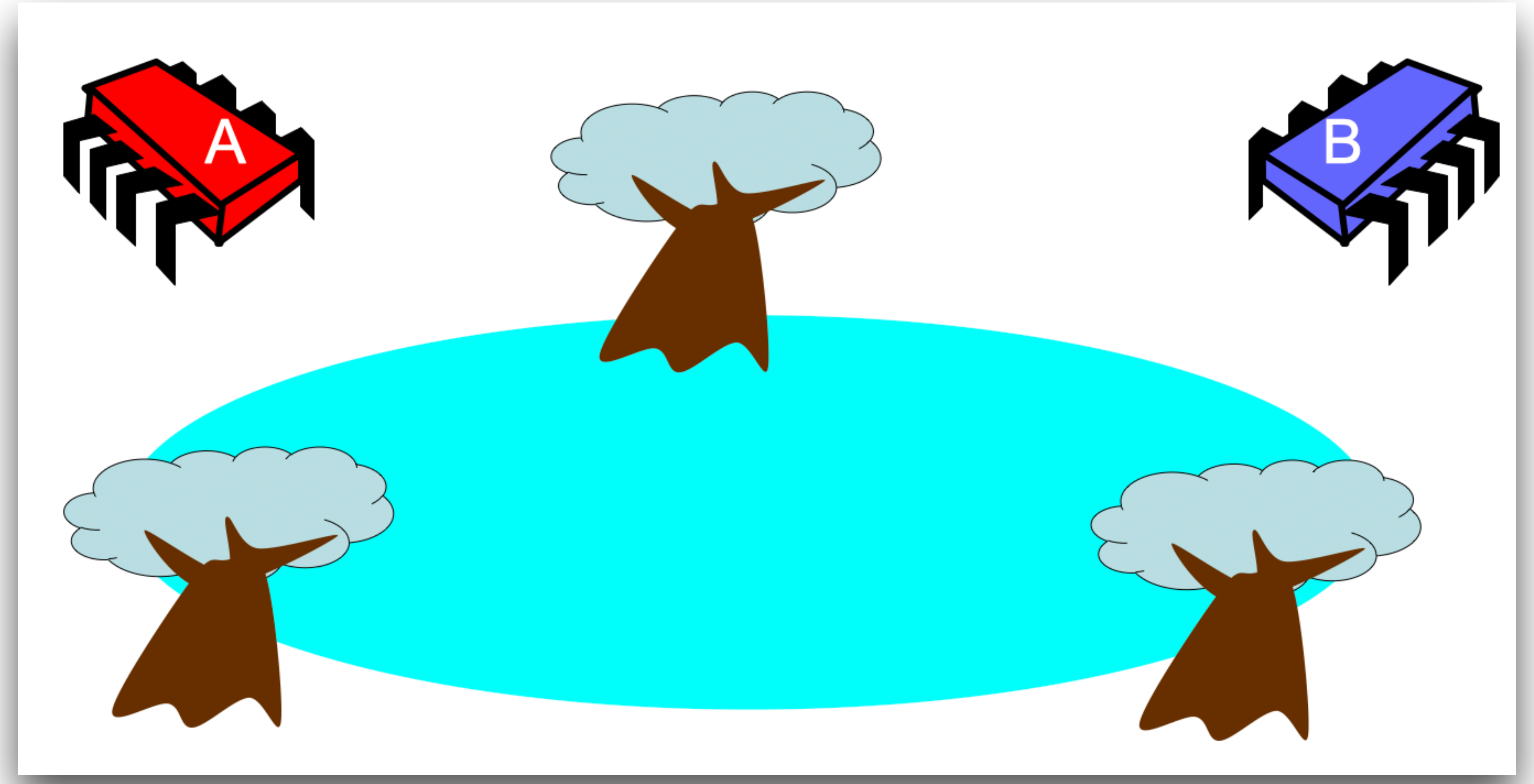
Simple Protocol

- Idea — Just look at the pond



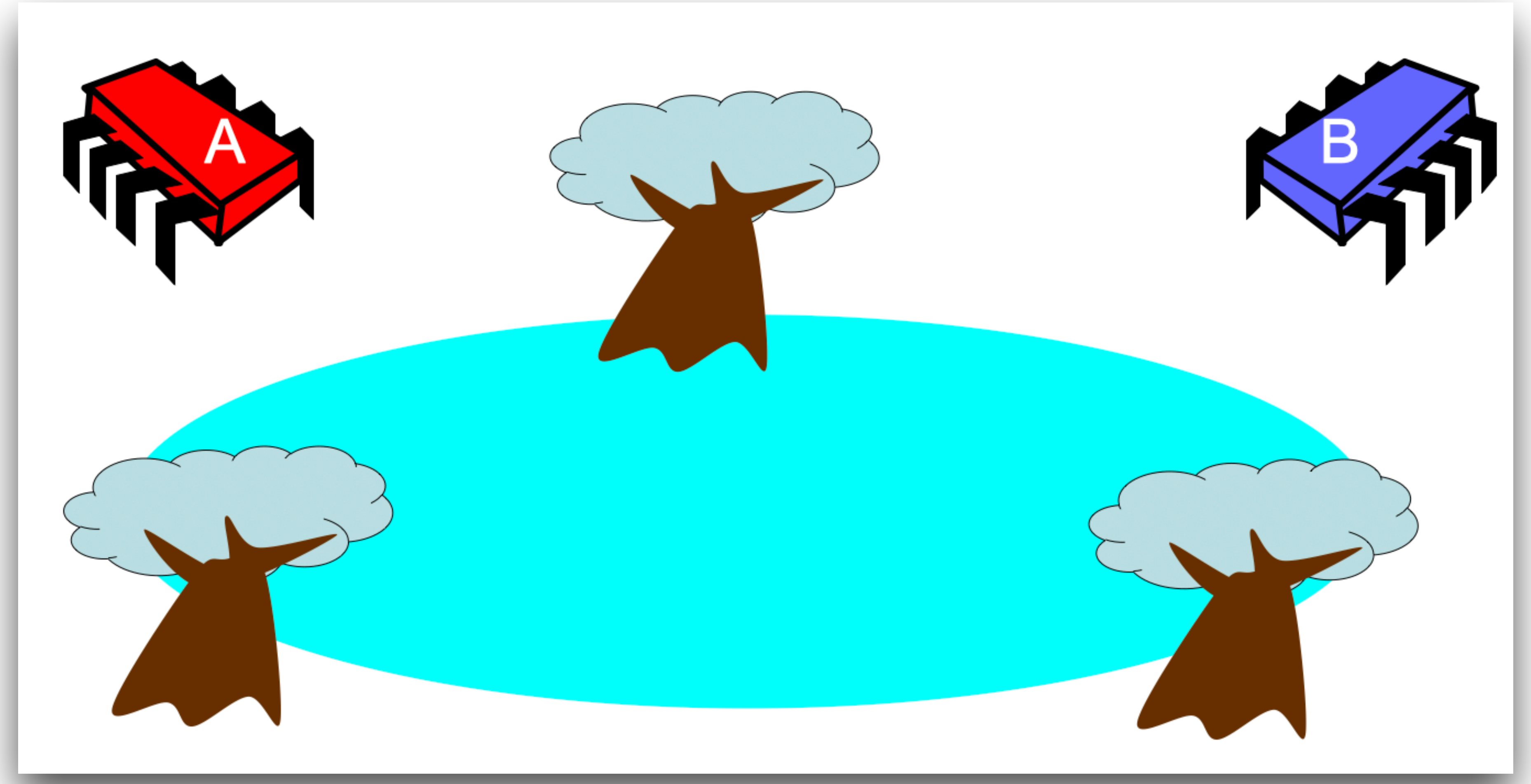
Simple Protocol

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

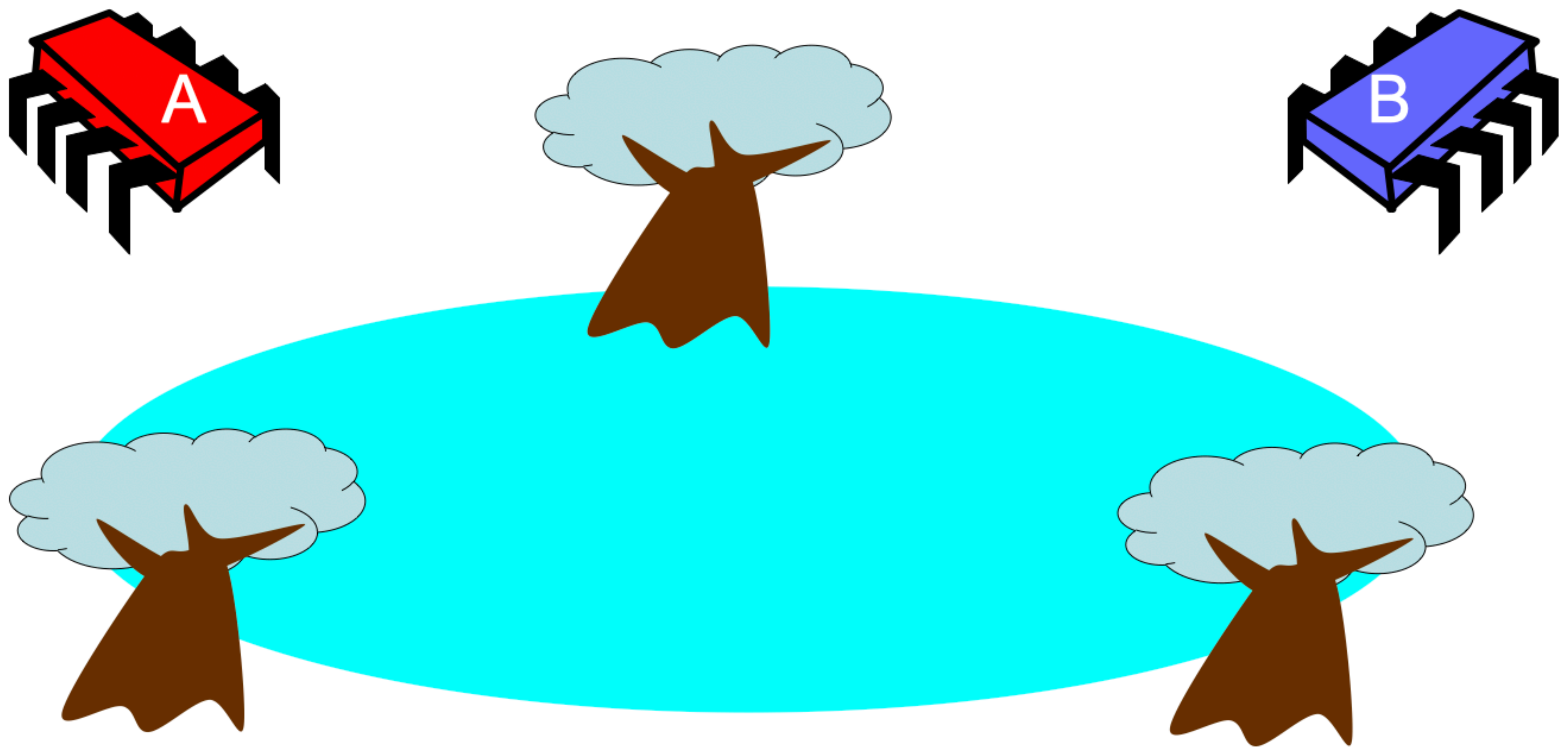


Simple Protocol

- 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

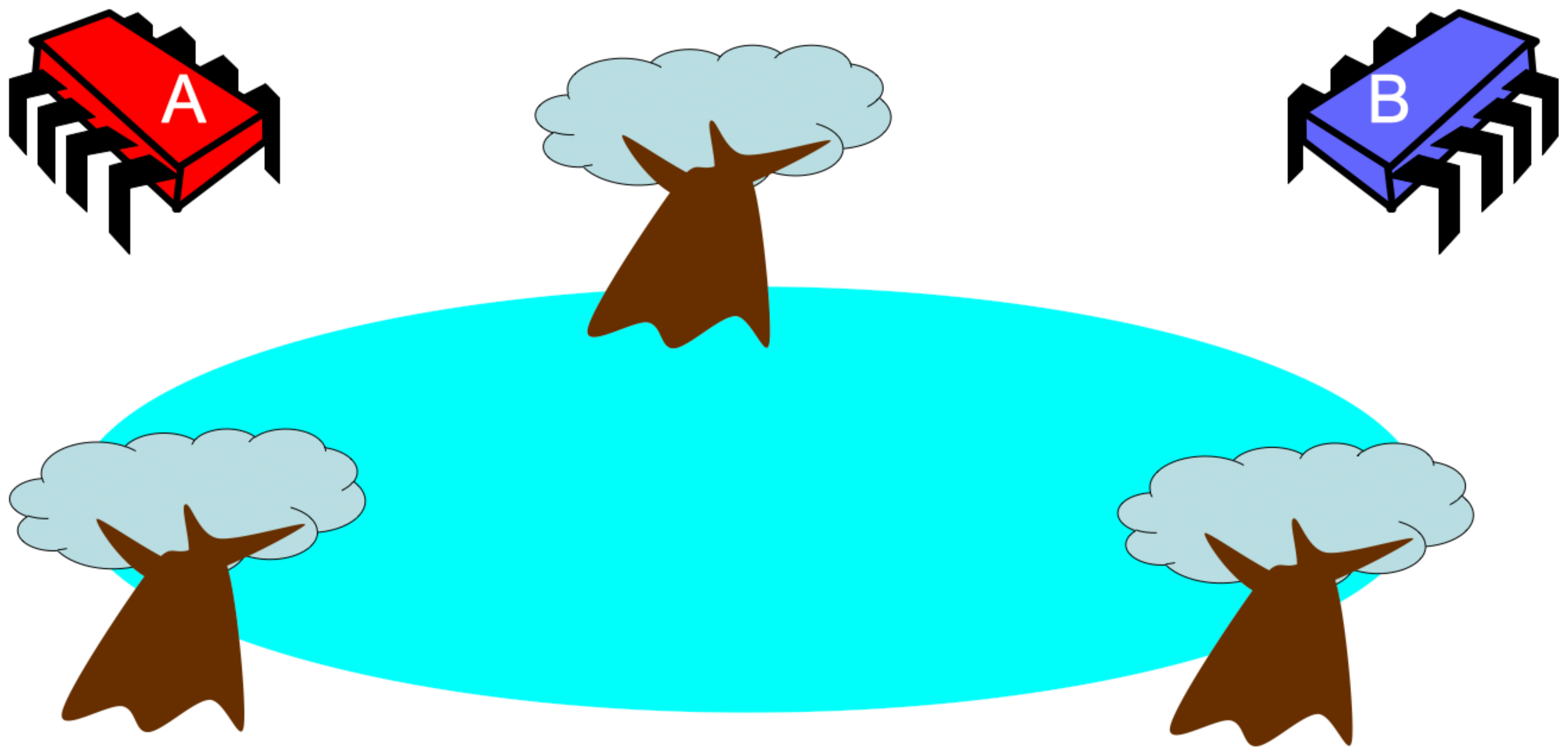


Mobile phone protocol



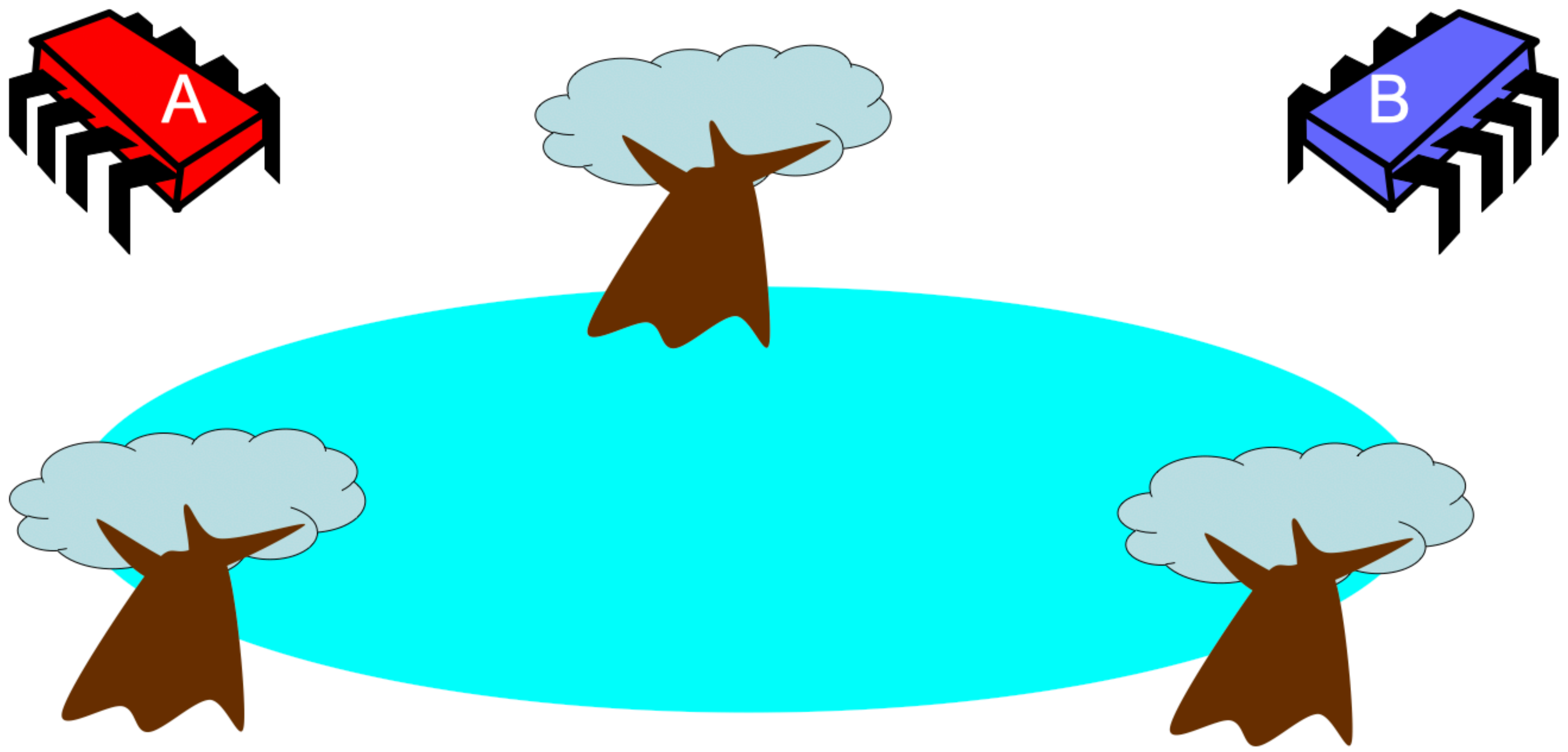
Mobile phone protocol

- Idea
  - Bob calls Alice (or vice versa)



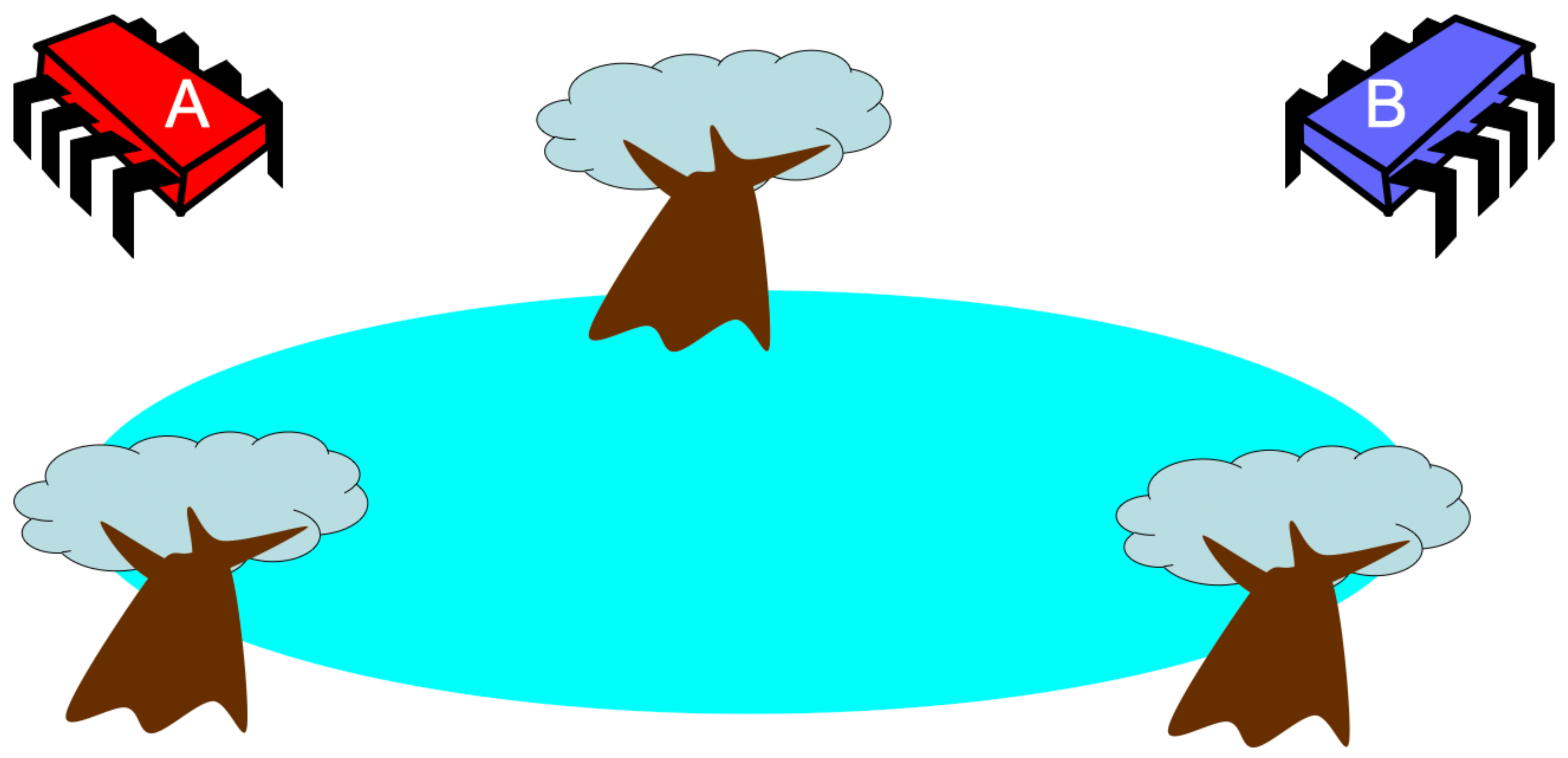
Mobile phone protocol

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

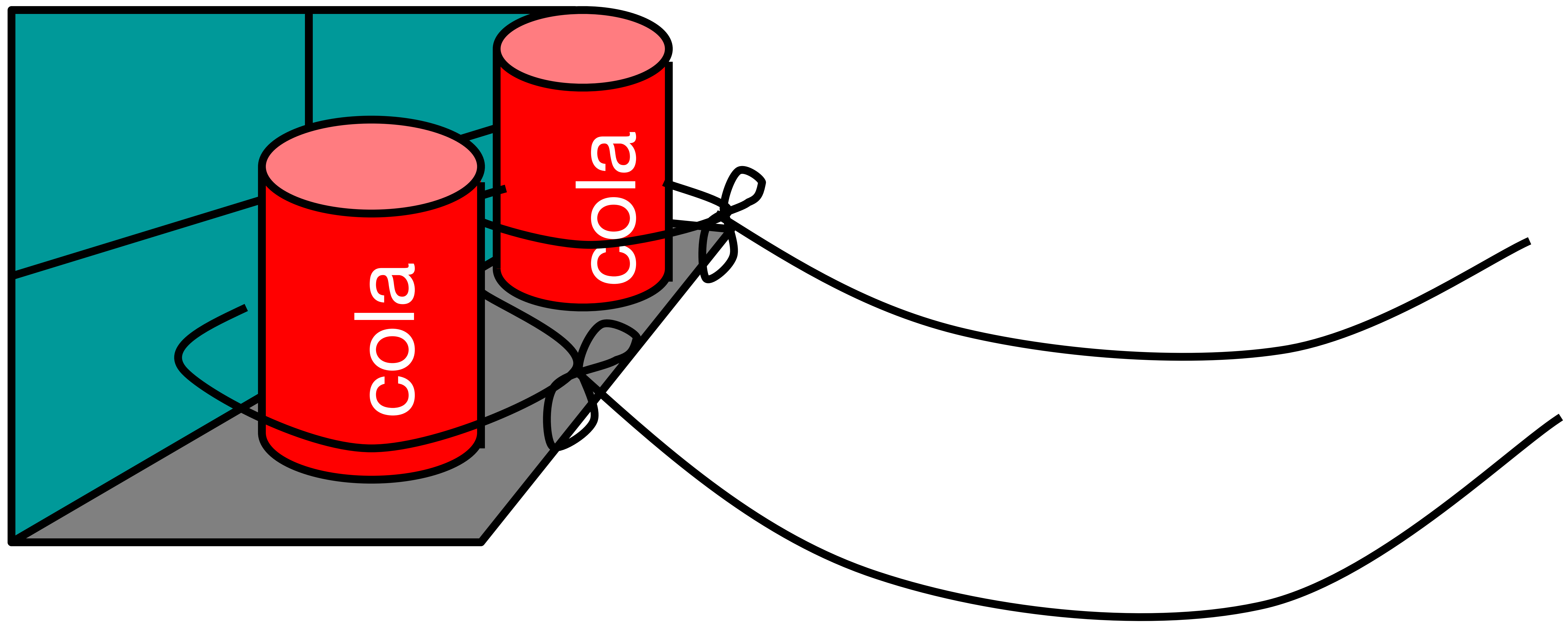


Mobile phone protocol

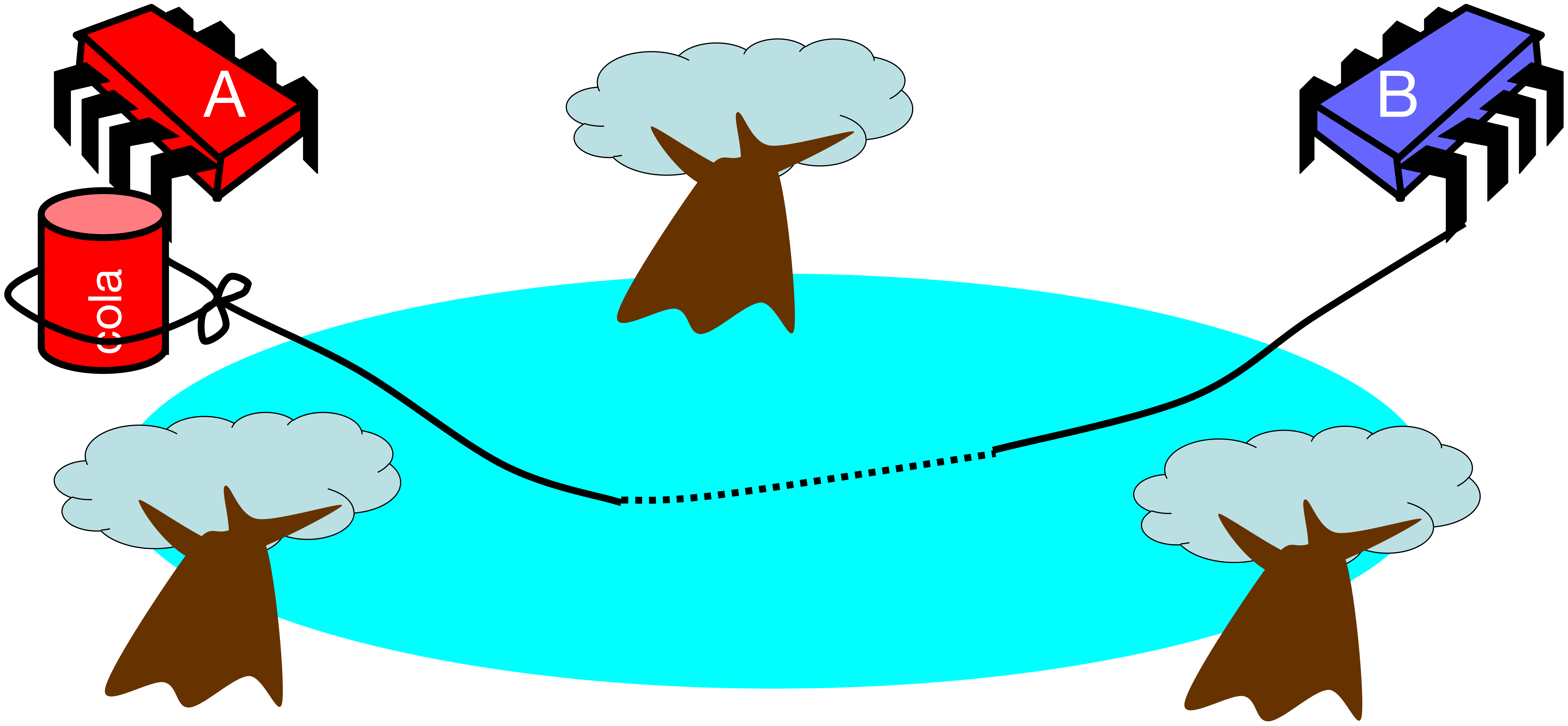
- Idea
  - Bob calls Alice (or vice versa)
- Gotcha
  - Bob takes shower
  - Alice recharges battery
  - Bob out shopping for pet food …
- Interpretation
  - Message-passing doesn't work
    - Recipient might not be listening or there at all
  - Communication must be Persistent (like writing) and not Transient (like speaking)



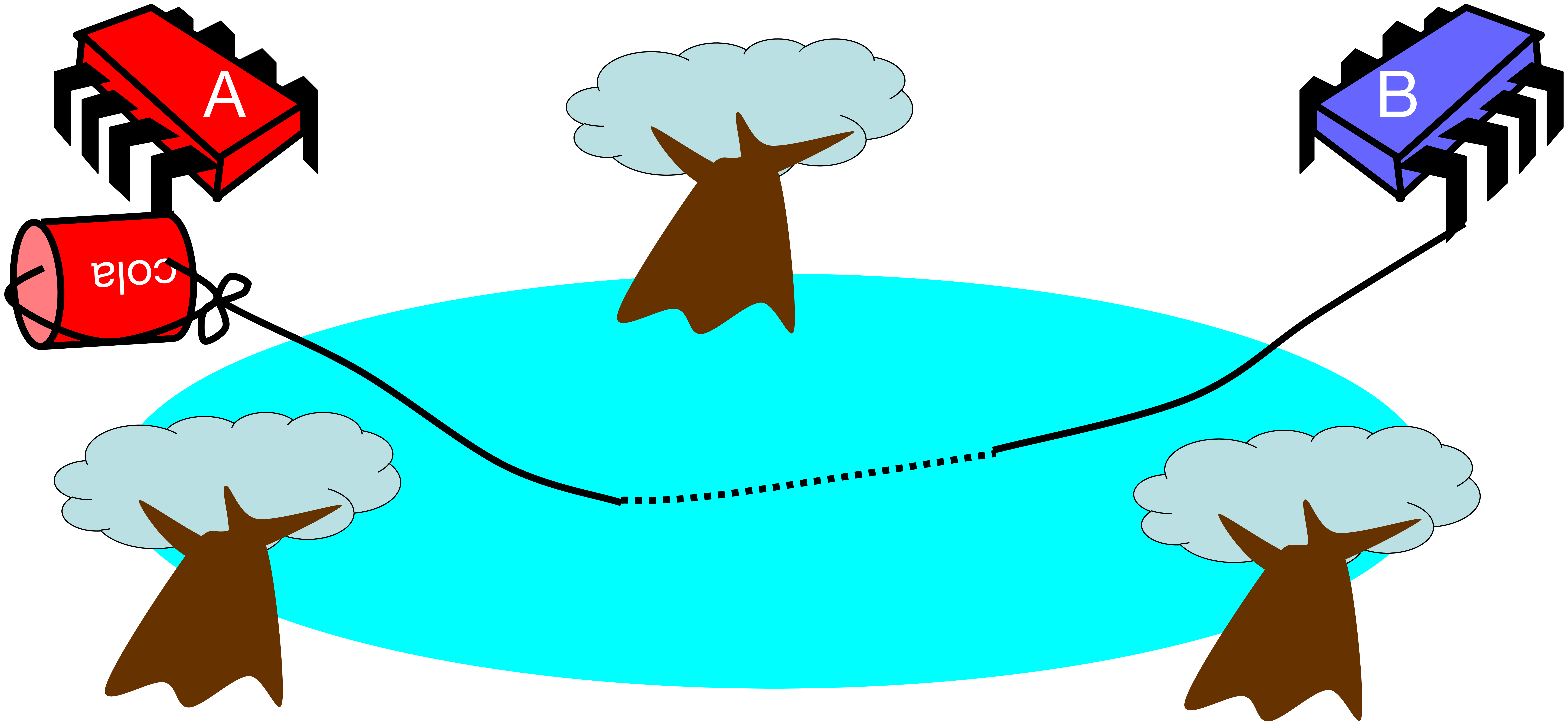
Can Protocol



Can Protocol — Bob conveys a bit



Can Protocol — Bob conveys a bit



Can Protocol

Can Protocol

- **Idea**
  - Cans on Alice’s windowsill
  - Strings lead to Bob’s house
  - Bob pulls strings, knocks over cans
    - Alice resets them

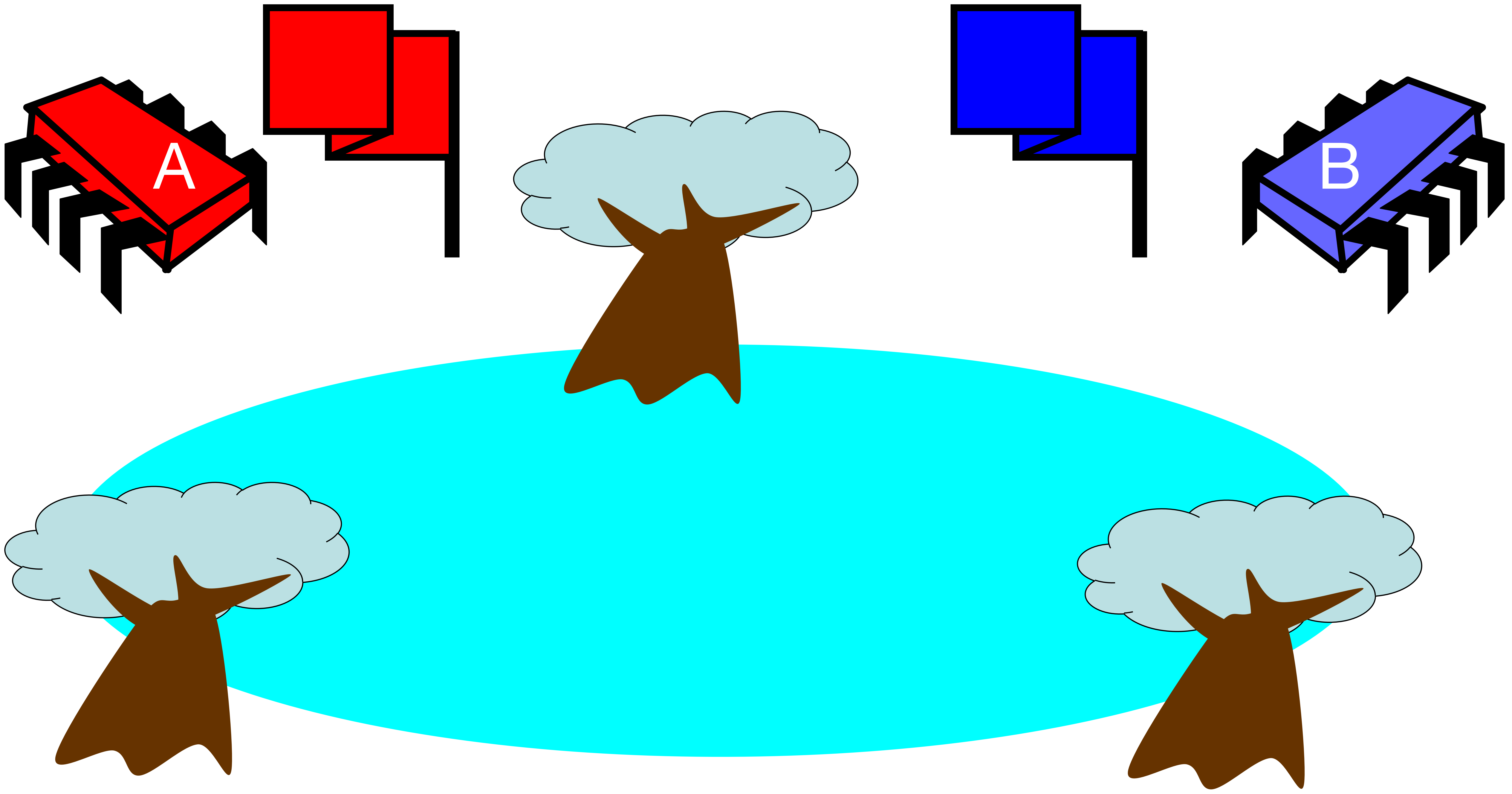
Can Protocol

- **Idea**
  - Cans on Alice’s windowsill
  - Strings lead to Bob’s house
  - Bob pulls strings, knocks over cans
    - Alice resets them
- **Gotcha**
  - Cans cannot be reused
  - Bob runs out of cans; Alice is gone on a vacation

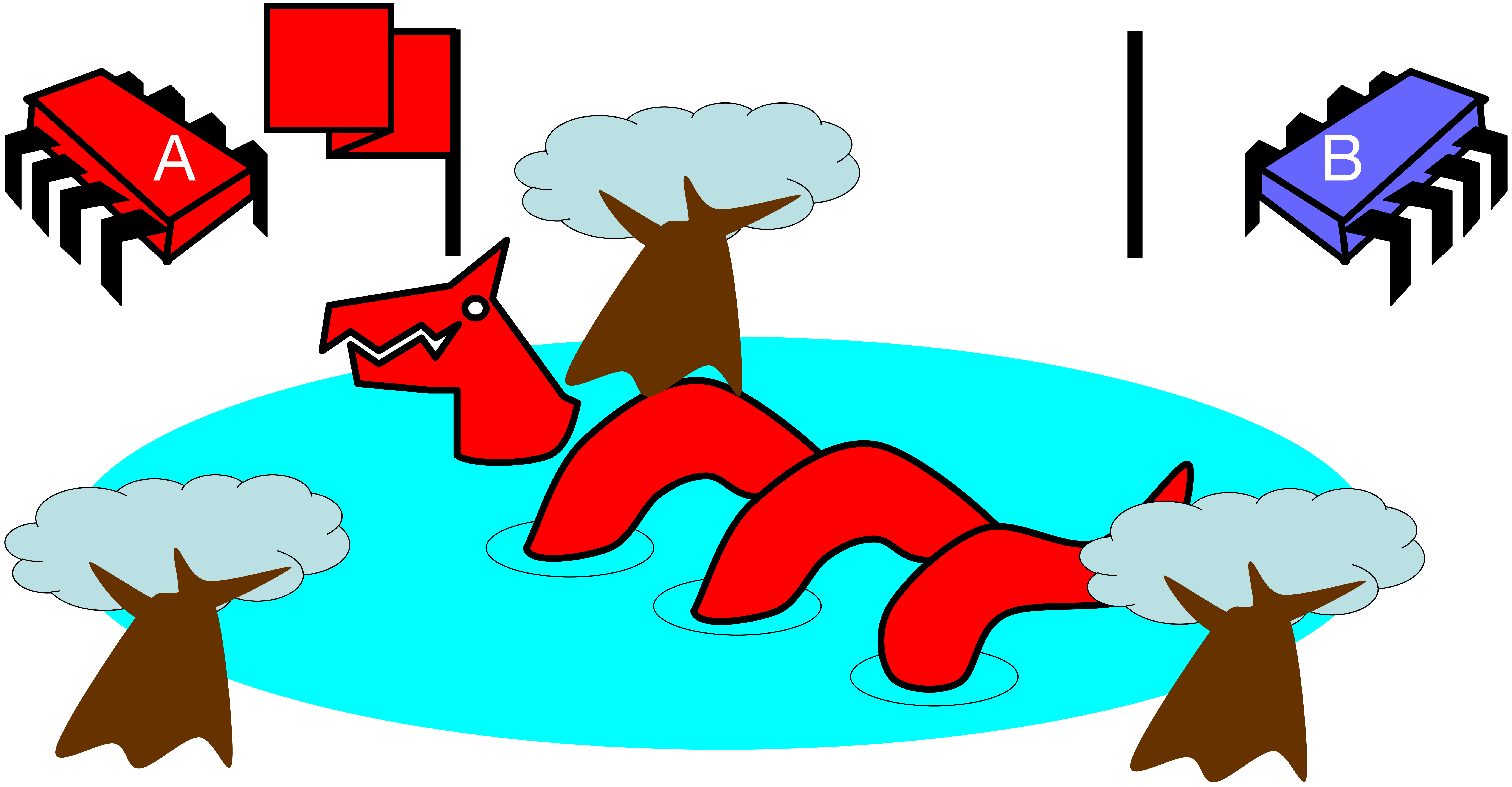
Can Protocol

- **Idea**
  - Cans on Alice's windowsill
  - Strings lead to Bob's house
  - Bob pulls strings, knocks over cans
    - Alice resets them
- **Gotcha**
  - Cans cannot be reused
  - Bob runs out of cans; Alice is gone on a vacation
- **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

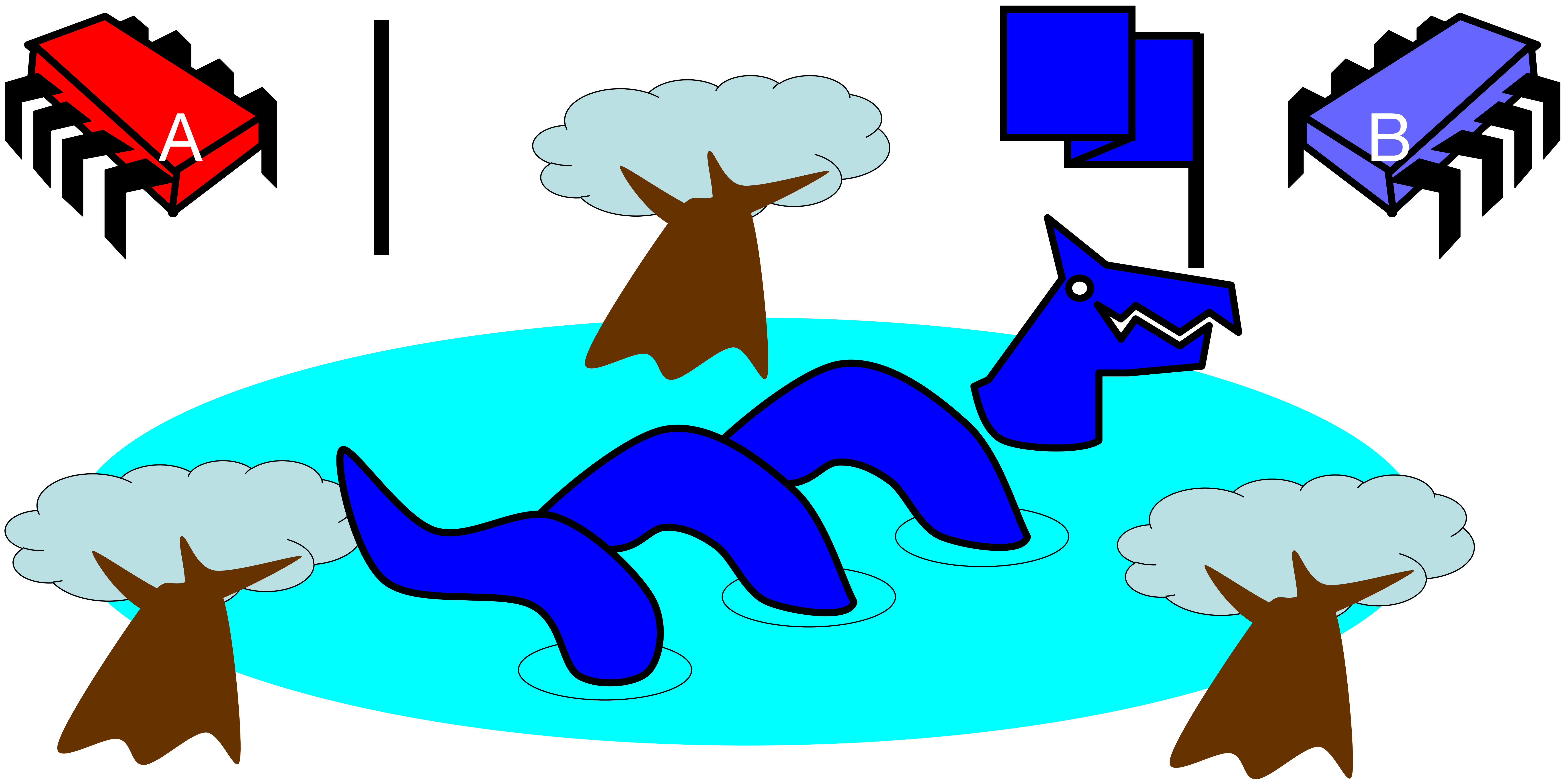
Flag Protocol



Alice's Protocol (sort of)



Bob’s Protocol (sort of)



Flag Protocol

Flag Protocol

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

Flag Protocol

- **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

Flag Protocol

- **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!

Flag Protocol

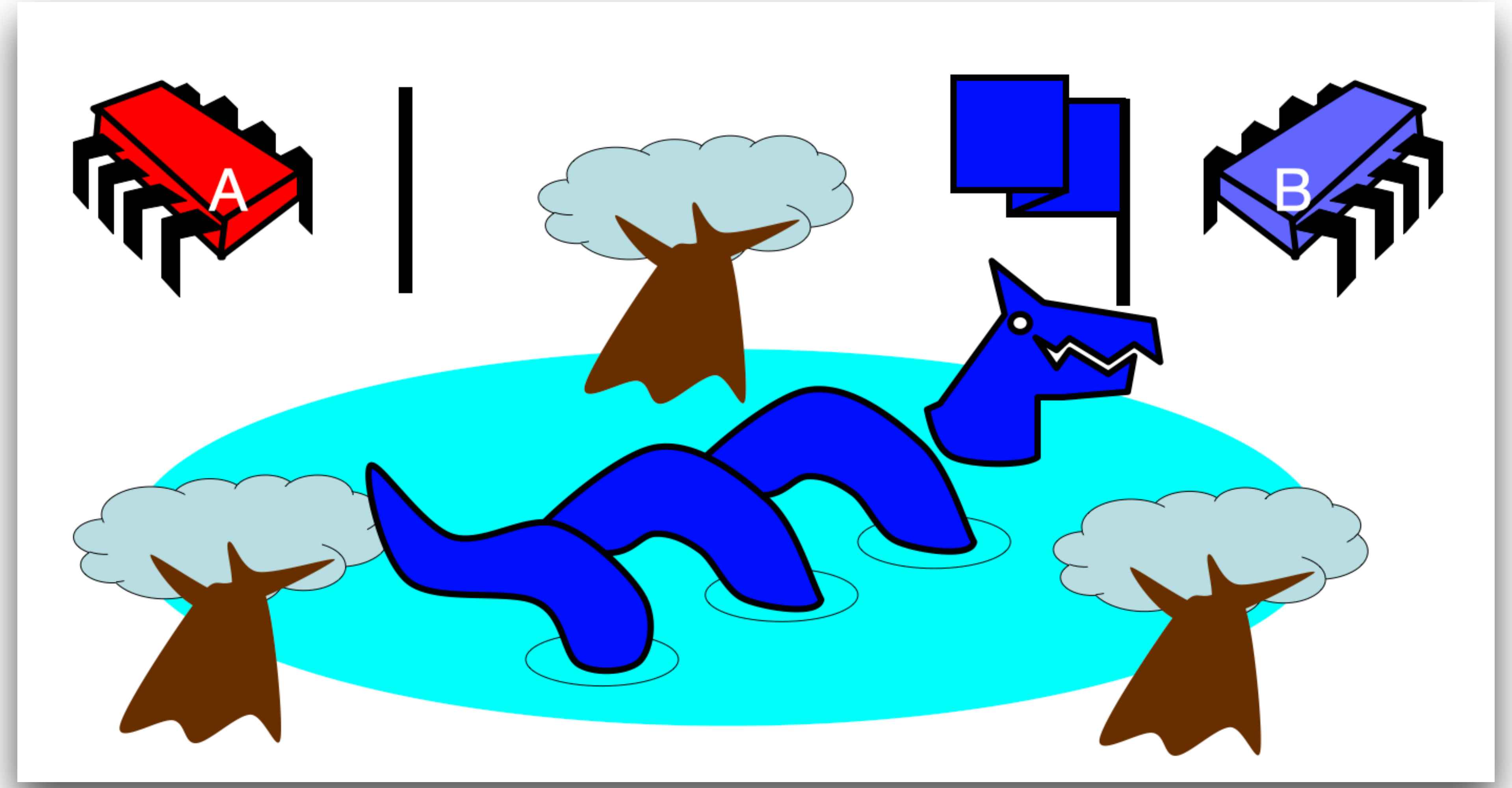
- **Alice's Protocol**
  - Raise flag
  - Wait until Bob's flag is down
  - Unleash pet
  - Lower flag when pet returns
- **Bob's Protocol (2nd try)**
  - Raise flag
  - While (Alice's flag is up) {
    - Lower flag
    - Wait for Alice's flag to go down
    - Raise flag }
  - Unleash pet
  - Lower flag when pet returns

Flag Protocol

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

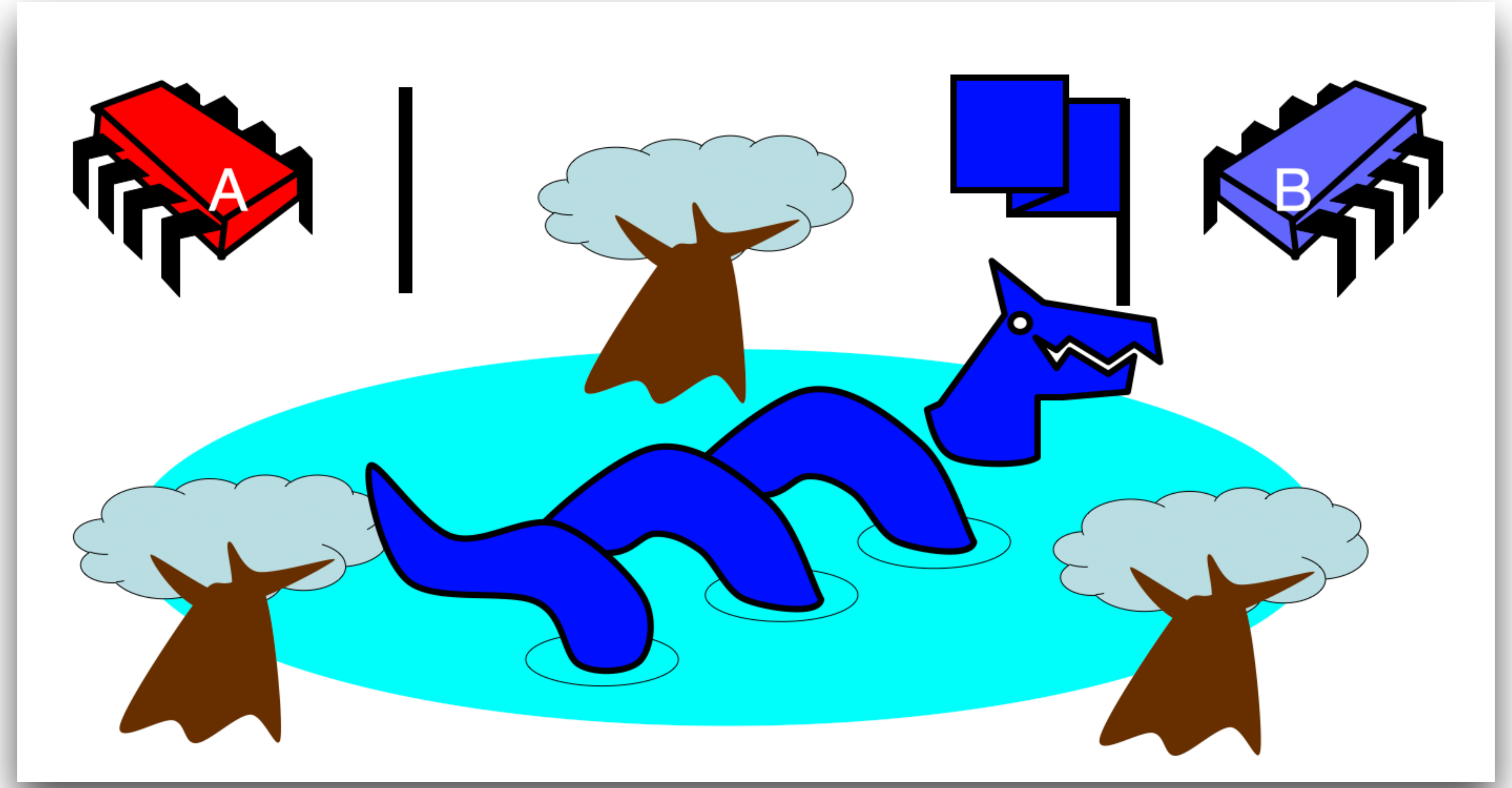
Bob defers to Alice

The Flag Principle



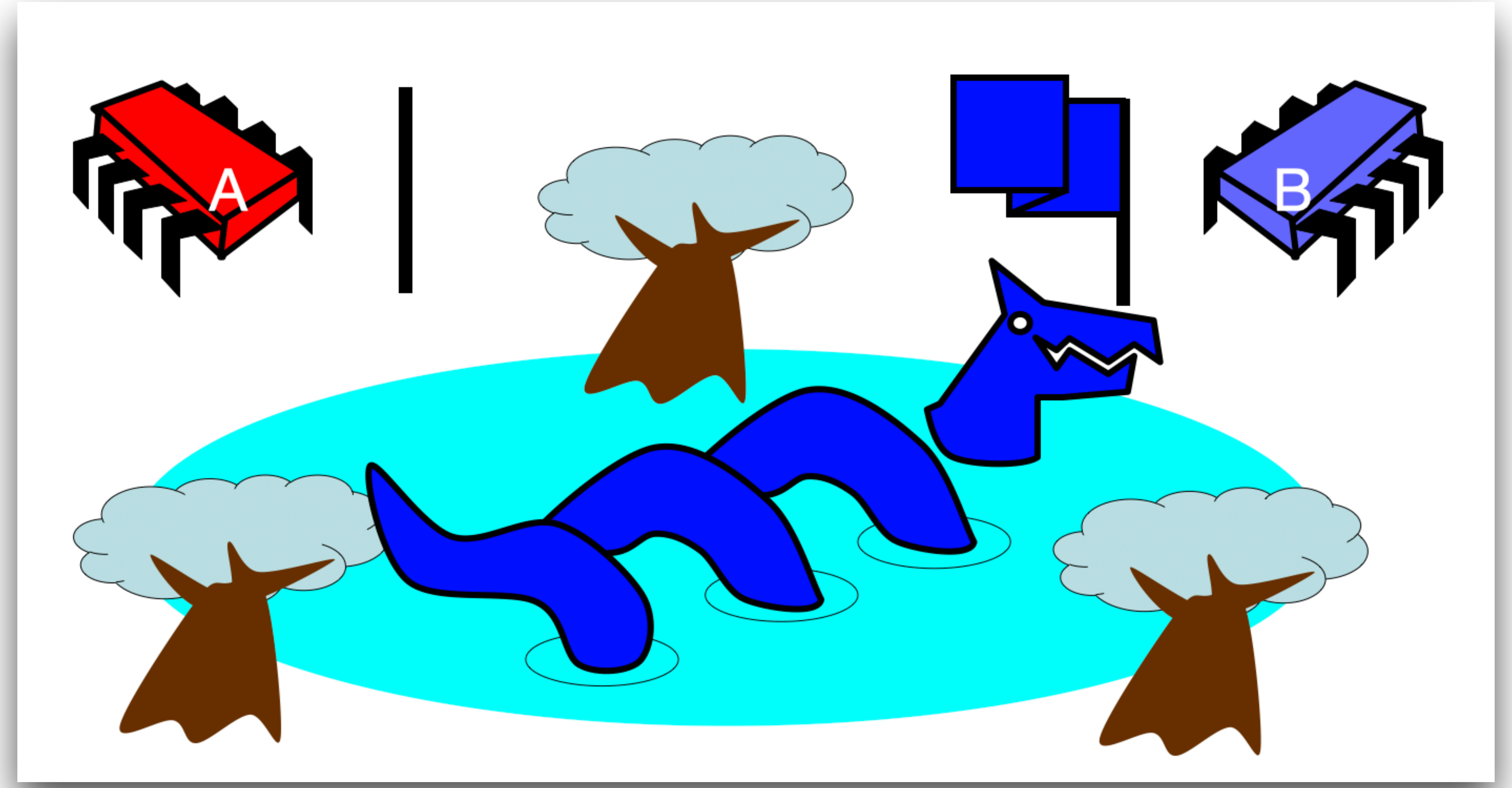
The Flag Principle

- Raise the flag



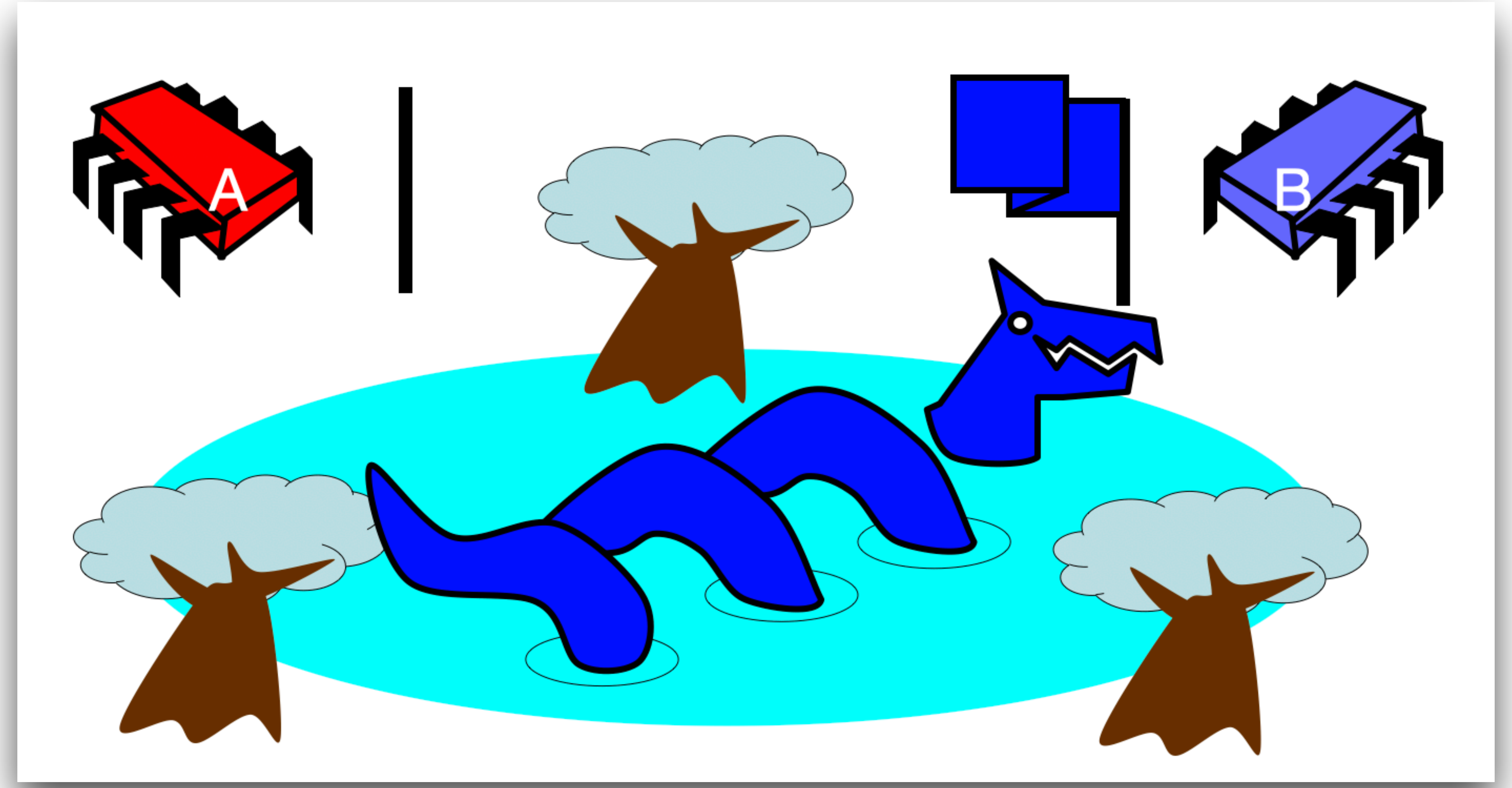
The Flag Principle

- Raise the flag
- Look at others' flag



The Flag Principle

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



Proof of Mutual Exclusion

Proof of Mutual Exclusion

- Assume both pets in pond
  - Derive a contradiction
  - By reasoning ***backwards***

Proof of Mutual Exclusion

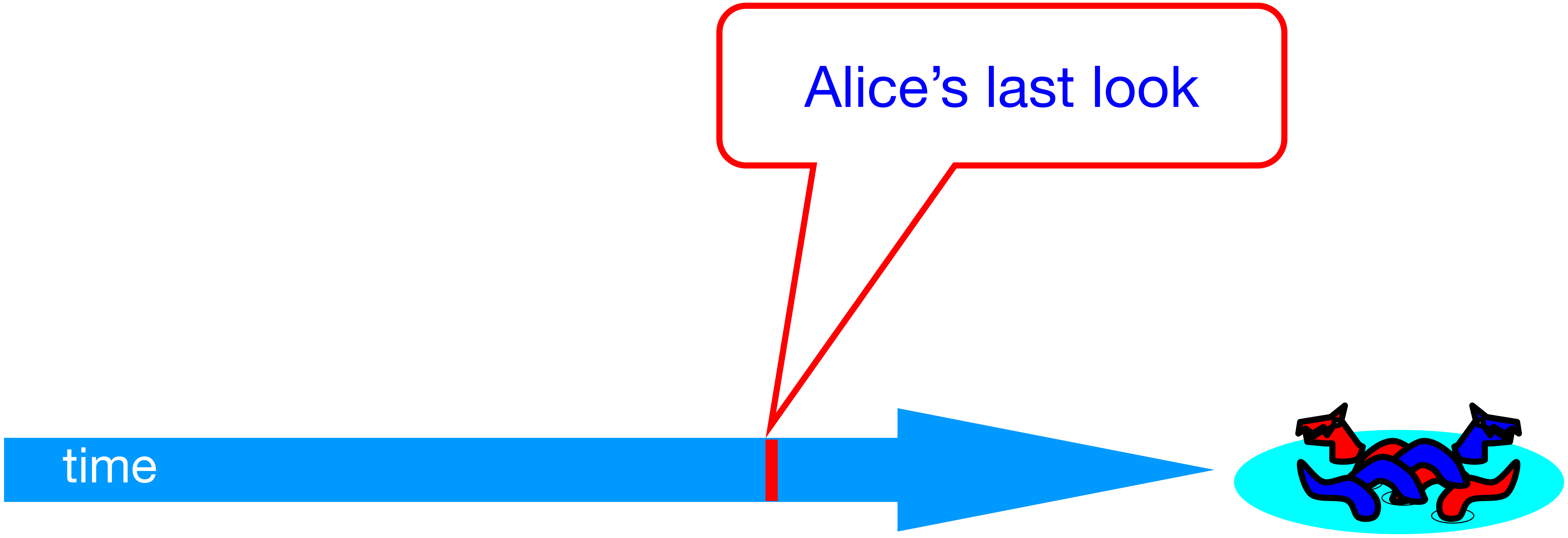
- Assume both pets in pond
  - Derive a contradiction
  - By reasoning ***backwards***
- Consider the last time Alice and Bob each looked before letting the pets in

Proof of Mutual Exclusion

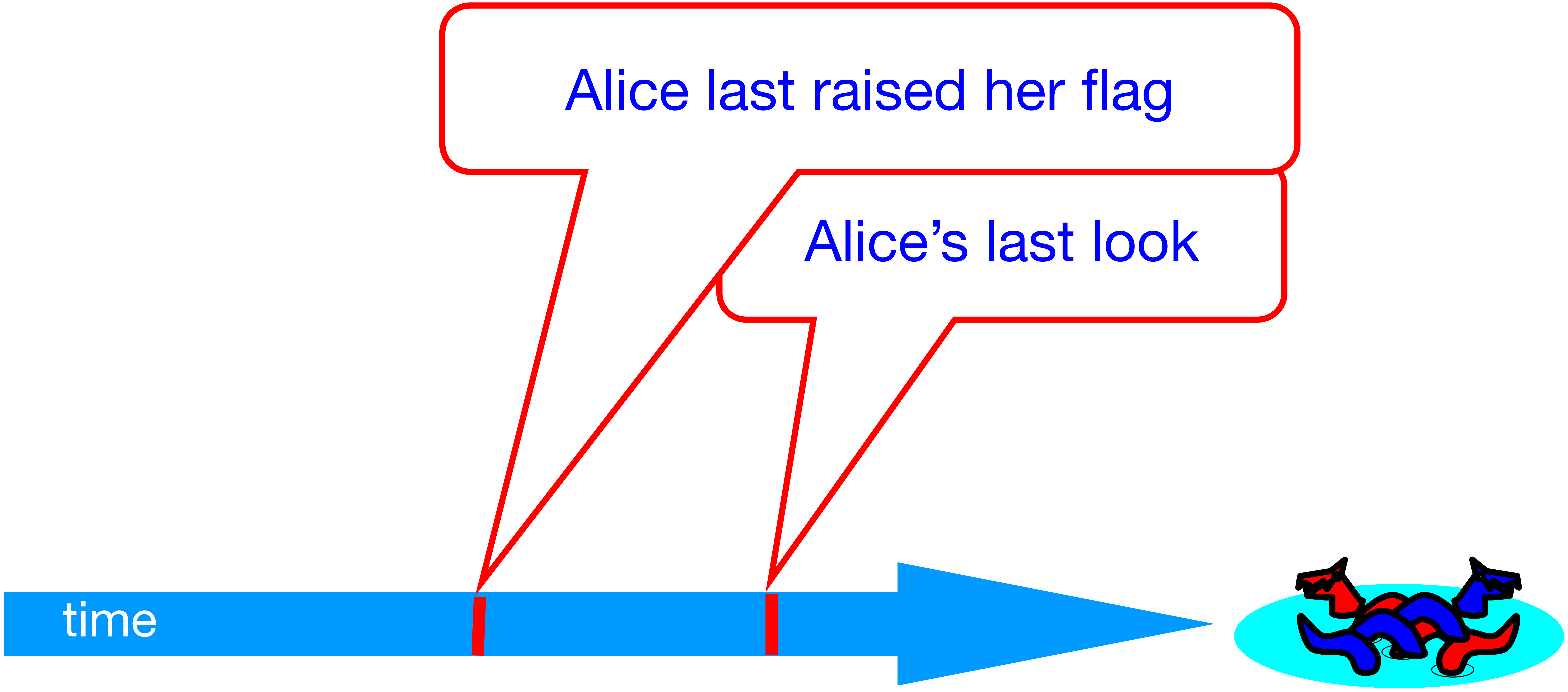
- Assume both pets in pond
  - Derive a contradiction
  - By reasoning ***backwards***
- Consider the last time Alice and Bob each looked before letting the pets in
- Without loss of generality assume Alice was the last to look…

Proof of Mutual Exclusion

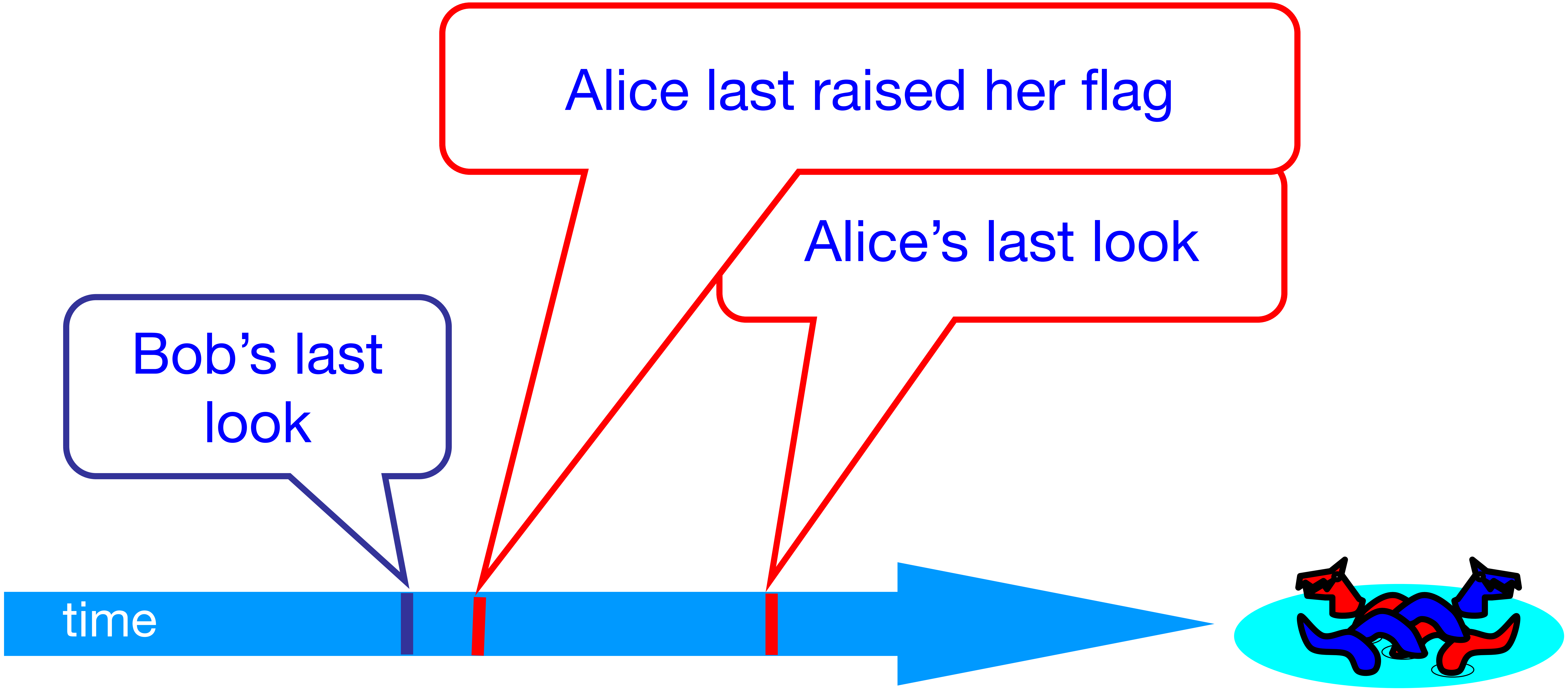
Proof of Mutual Exclusion



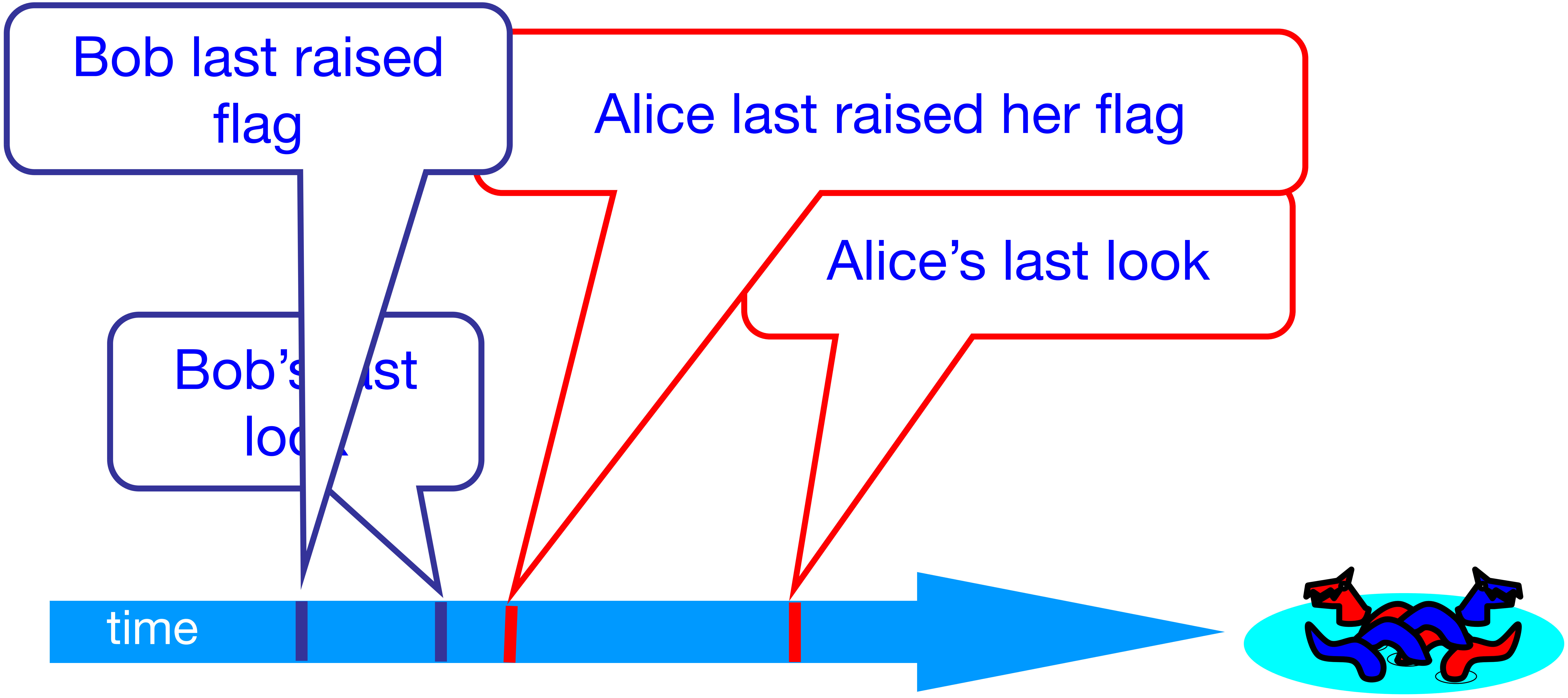
Proof of Mutual Exclusion



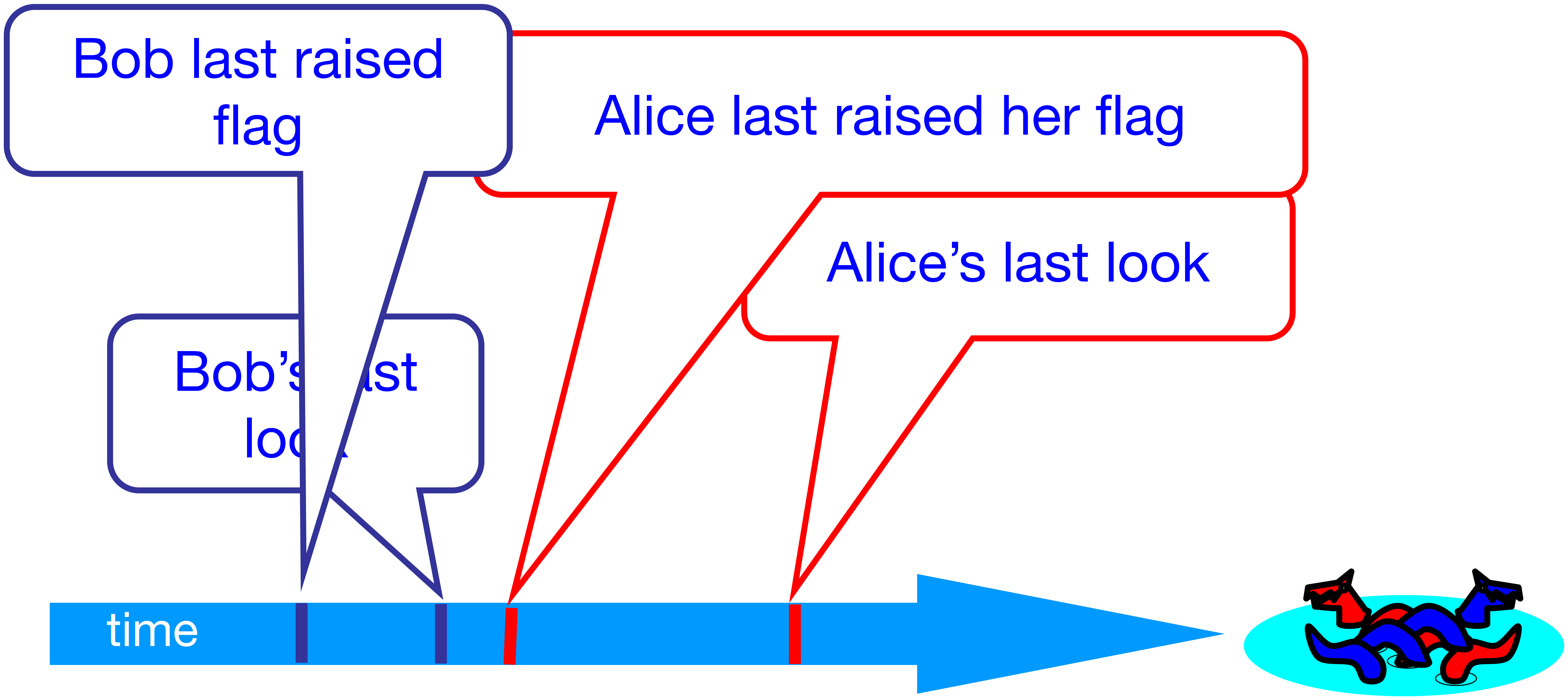
Proof of Mutual Exclusion



Proof of Mutual Exclusion

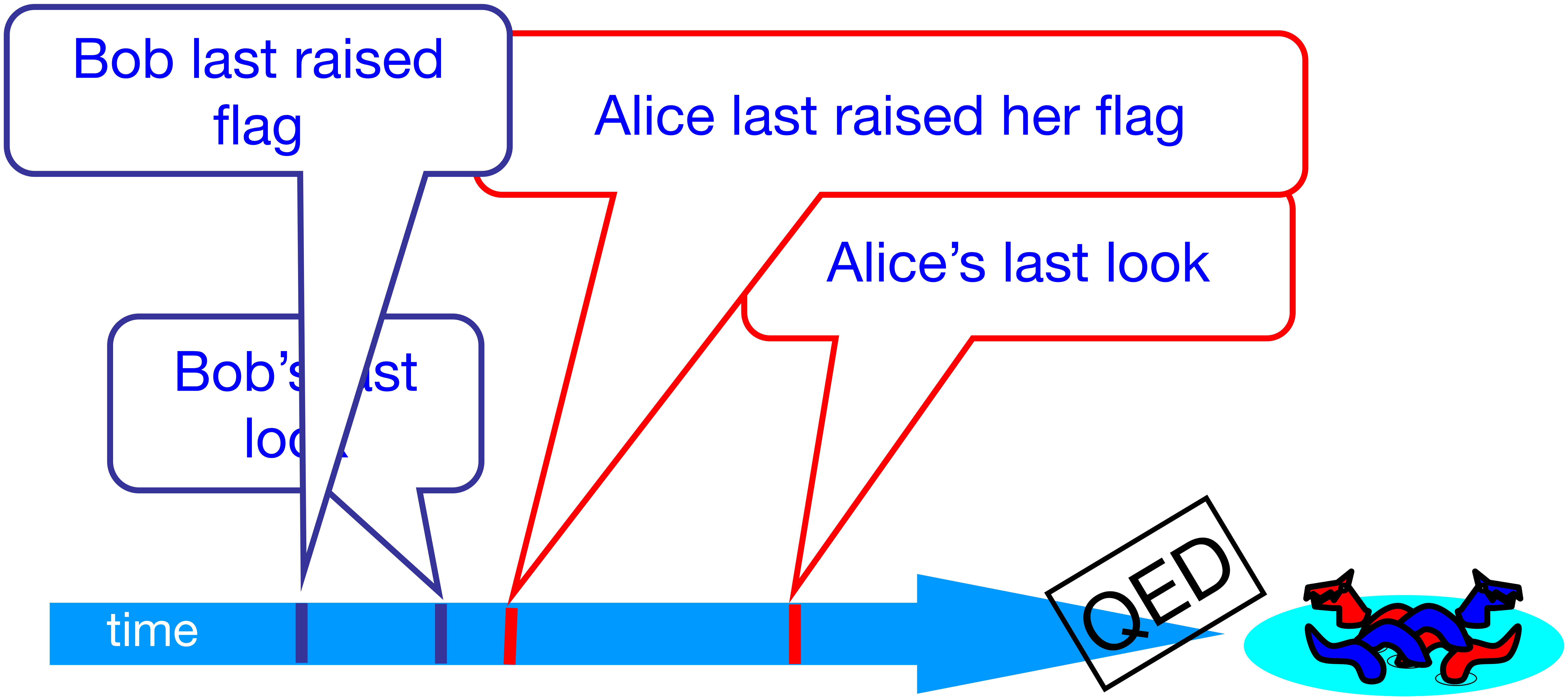


Proof of Mutual Exclusion



Alice must have seen Bob’s Flag. A Contradiction

Proof of Mutual Exclusion



Alice must have seen Bob's Flag. A Contradiction

Proof of No Deadlock

Proof of No Deadlock

- If one pet wants to get in, it gets in.

Proof of No Deadlock

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

Proof of No Deadlock

- If one pet wants to get in, it gets in.
- Deadlock requires both parties to continually try to get in.
- If Bob sees Alice’s flag, he backs off and gives her priority
  - Alice’s lexicographic privilege

Proof of No Deadlock

- If one pet wants to get in, it gets in.
- Deadlock requires both parties to continually try to get in.
- If Bob sees Alice's flag, he backs off and gives her priority
  - Alice's lexicographic privilege

QED

Remarks

Remarks

- Protocol is ***unfair***
    - Bob's pet might never get in and ***starve***

Remarks

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

Moral of the story

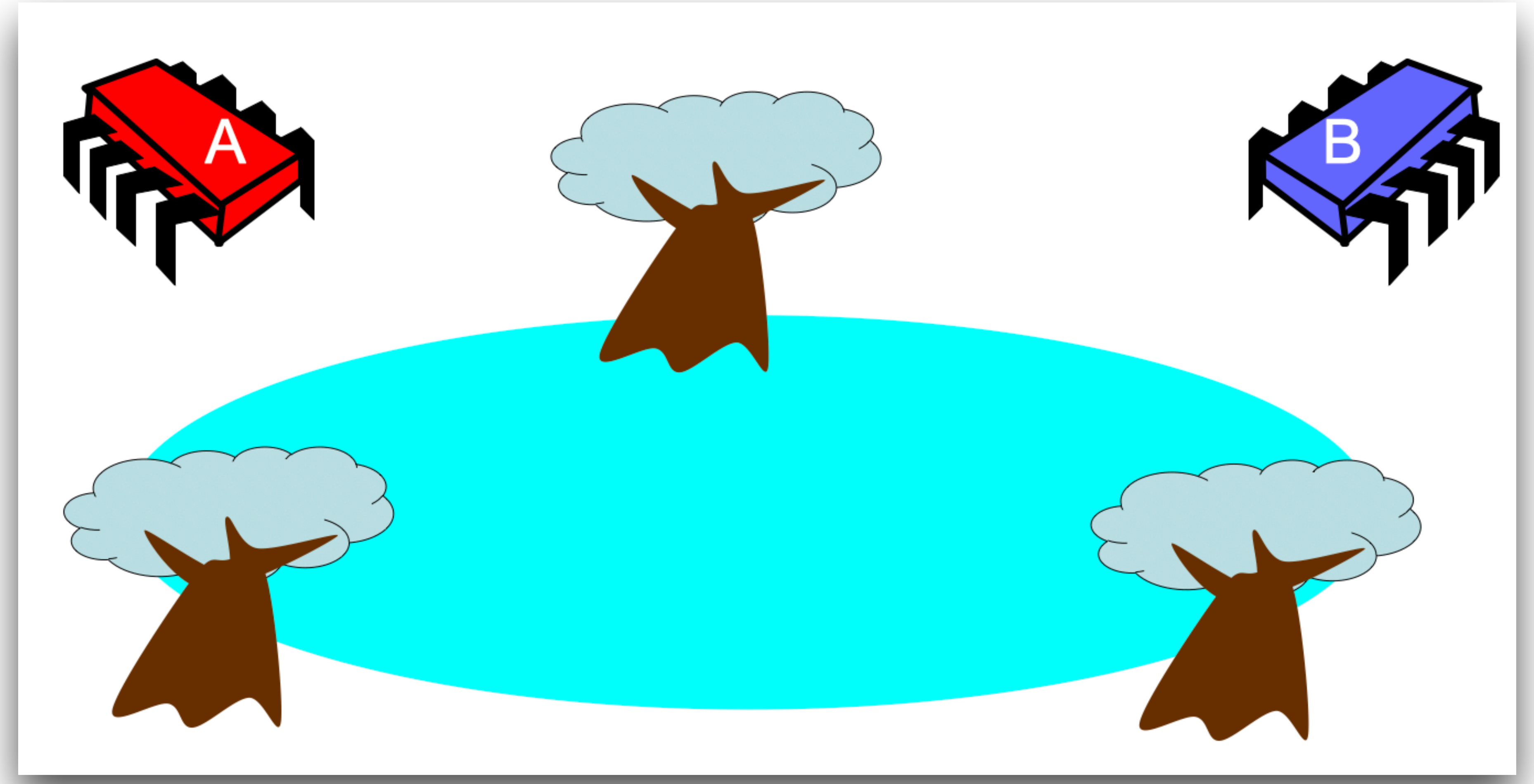
Moral of the story

- Mutual Exclusion ***cannot be solved*** by
  - transient communication (cell phones)
  - interrupts (cans)

Moral of the story

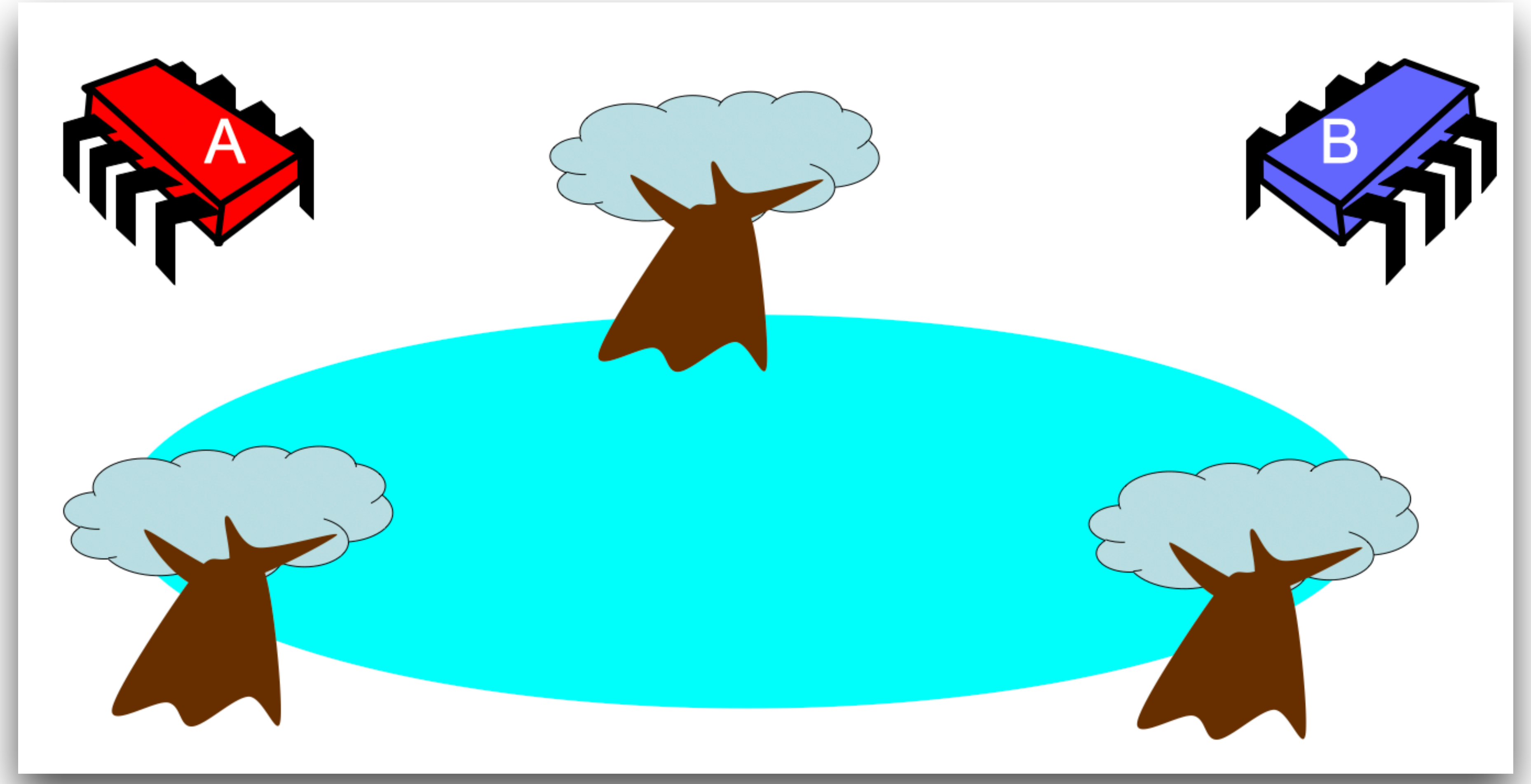
- Mutual Exclusion ***cannot be solved*** by
  - transient communication (cell phones)
  - interrupts (cans)
- It ***can be solved*** by
  - one-bit shared variables
  - that can be read or written

The fable continues



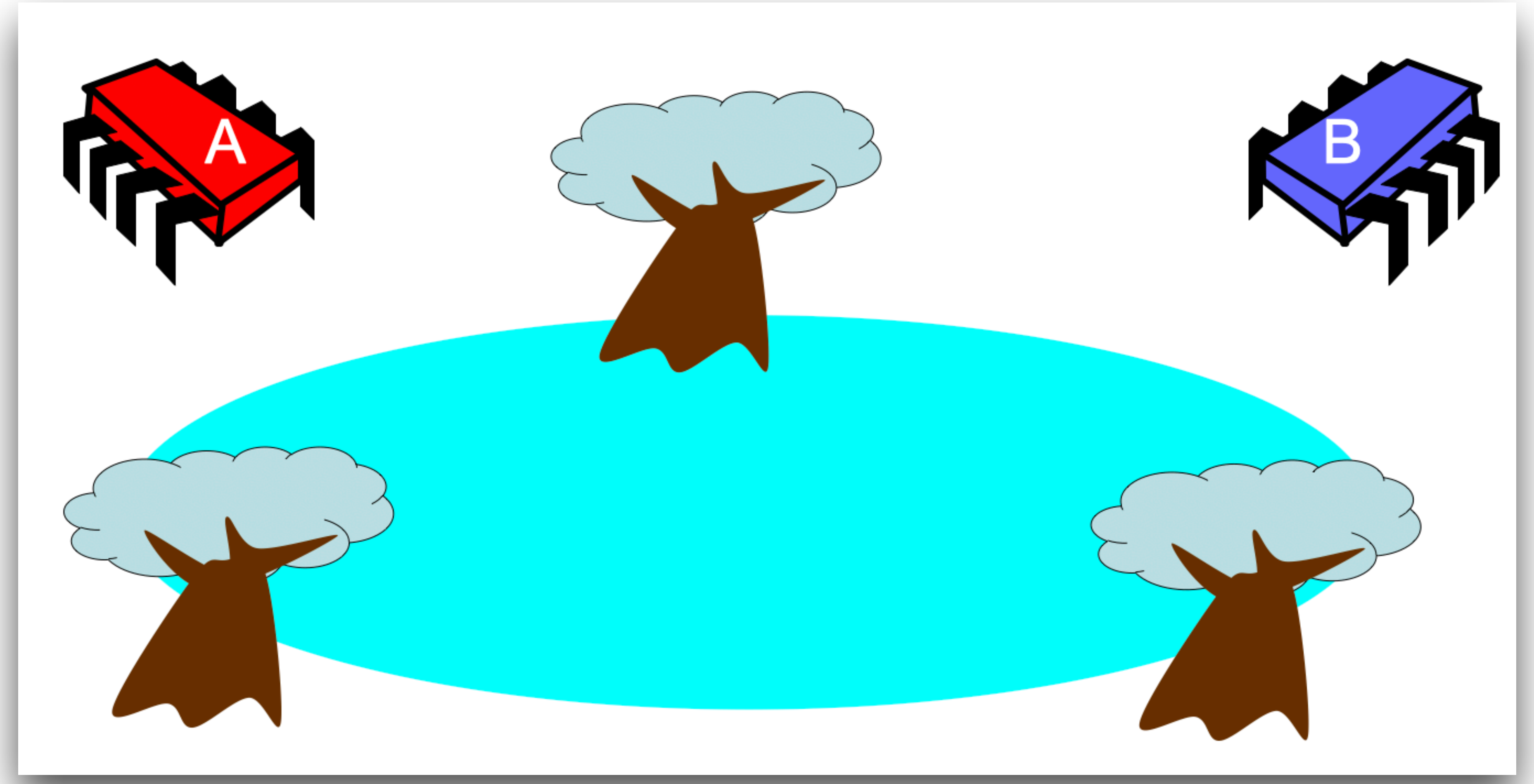
The fable continues

- Alice and Bob fall in love & marry
  - Then they fall out of love & divorce
  - She gets the pets



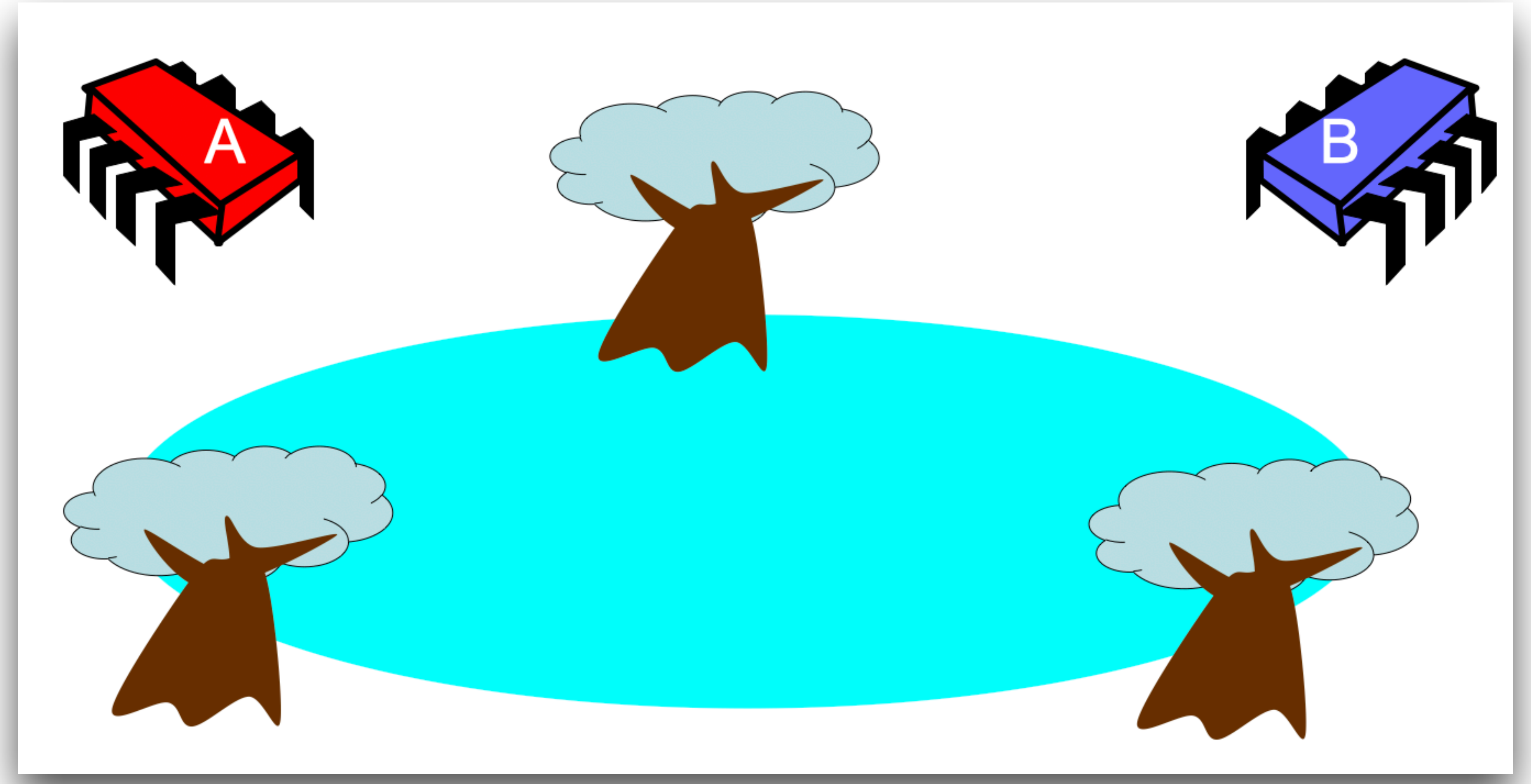
The fable continues

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

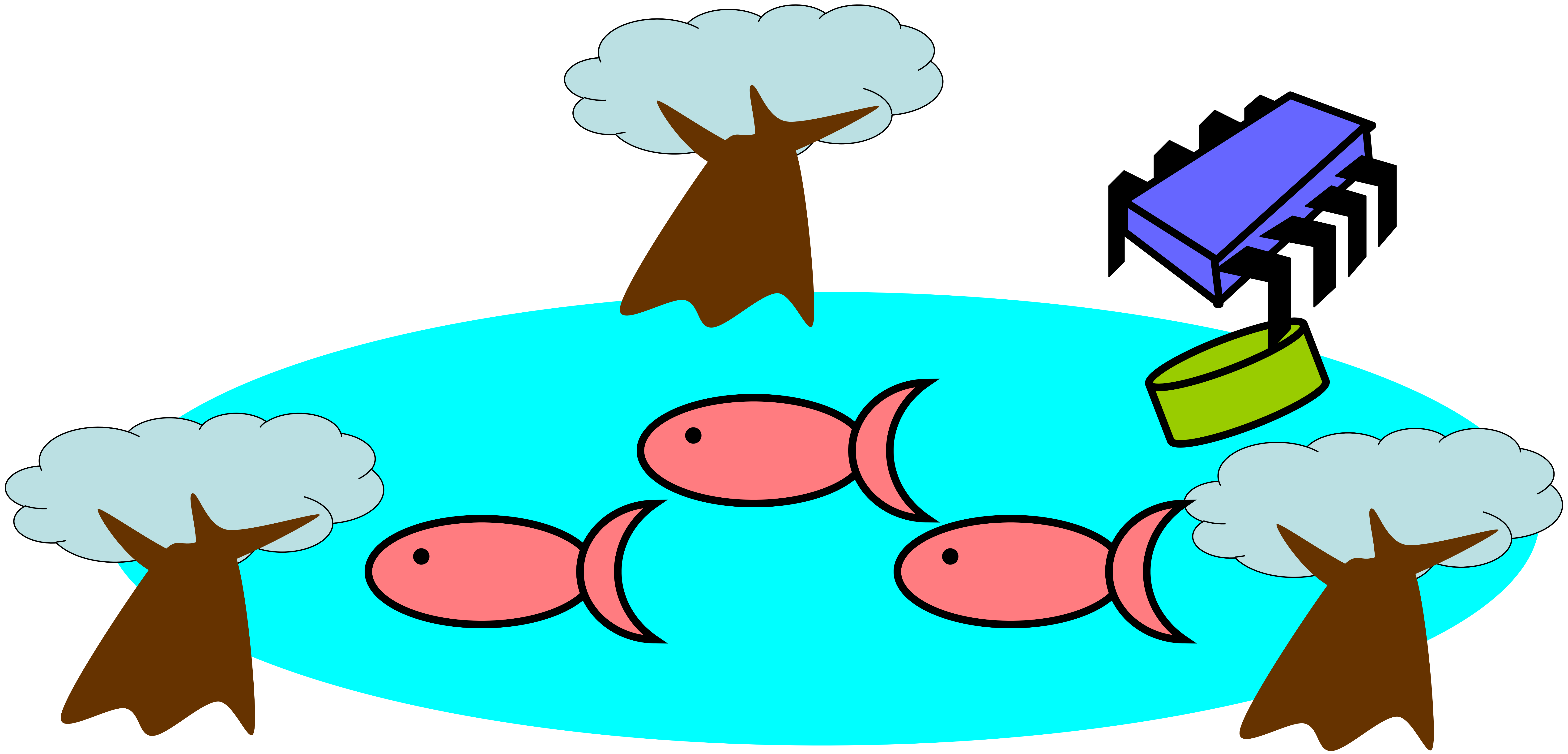


The fable continues

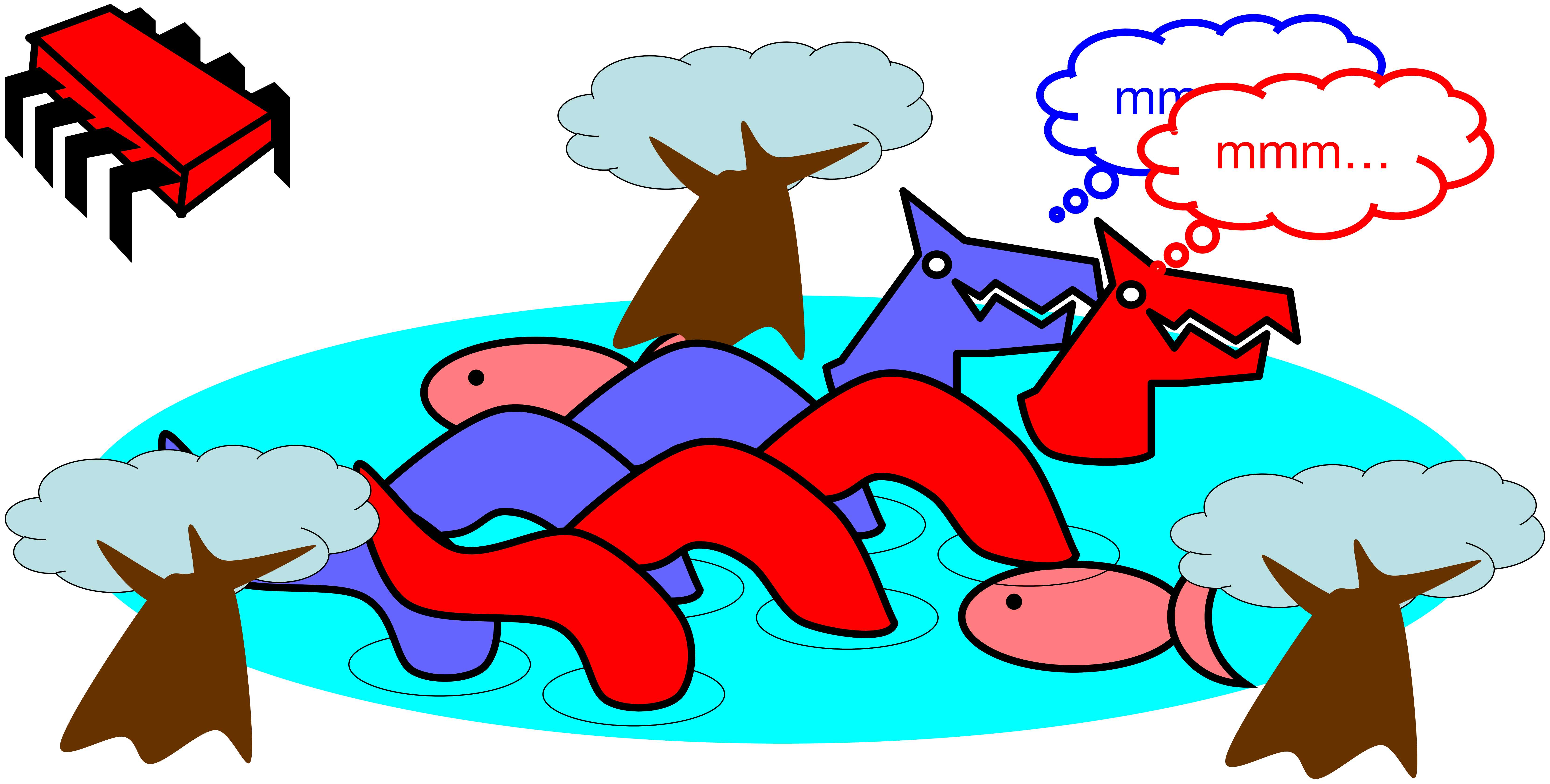
- Alice and Bob fall in love & marry
  - Then they fall out of love & divorce
  - She gets the pets
- He has to feed them
- Leading to a new coordination problem: **Producer-Consumer**



Bob Puts Food in the Pond



Alice releases her pets to feed



Producer/Consumer

Producer/Consumer

- Alice and Bob can’t meet
  - Each has a ***restraining order on the other***
  - So he puts food in the pond
  - And later, she releases the pets

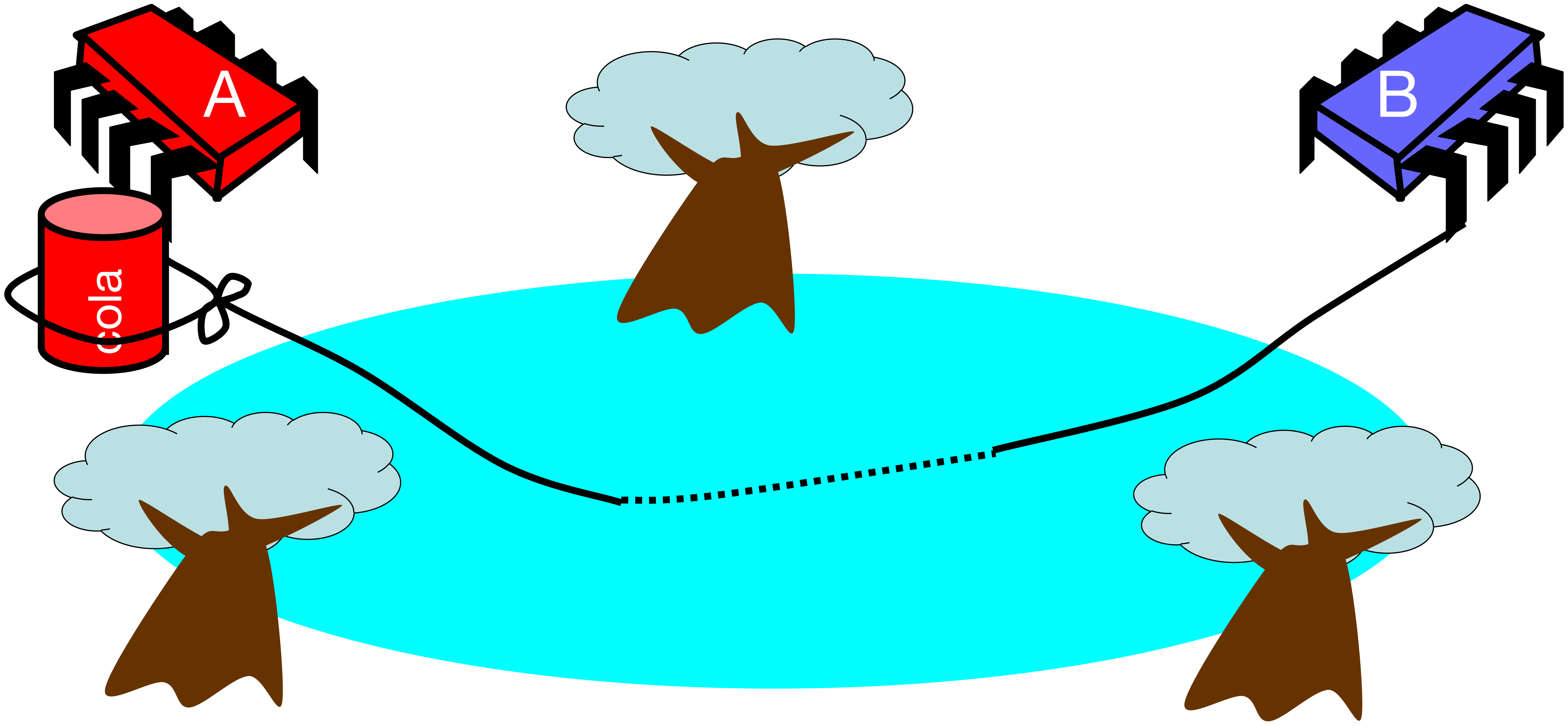
Producer/Consumer

- Alice and Bob can’t meet
  - Each has a ***restraining order on the 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

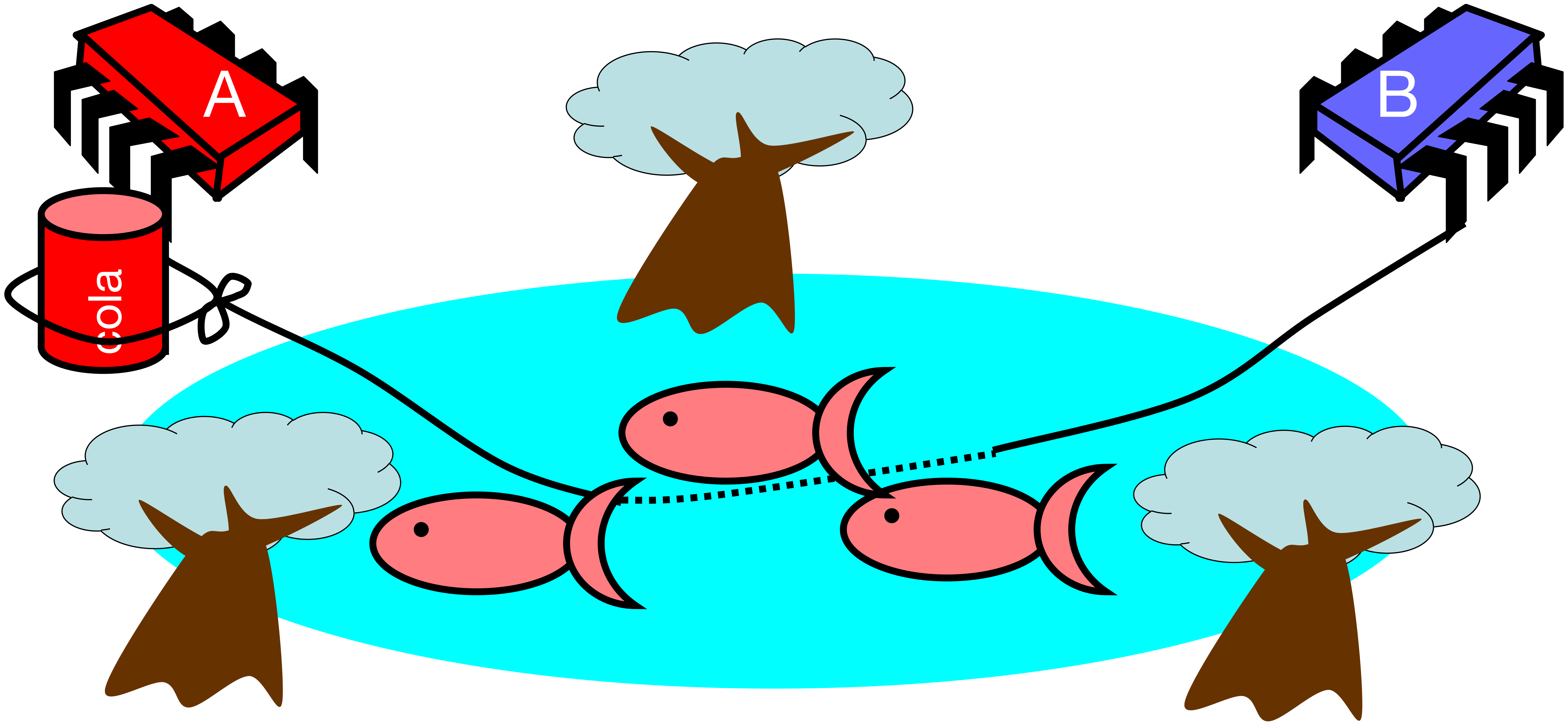
Producer/Consumer

- Alice and Bob can't meet
  - Each has a ***restraining order on the 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
- Need a mechanism so that
  - Bob lets Alice know when the food has been put out
  - Alice lets Bob know when to put out more food

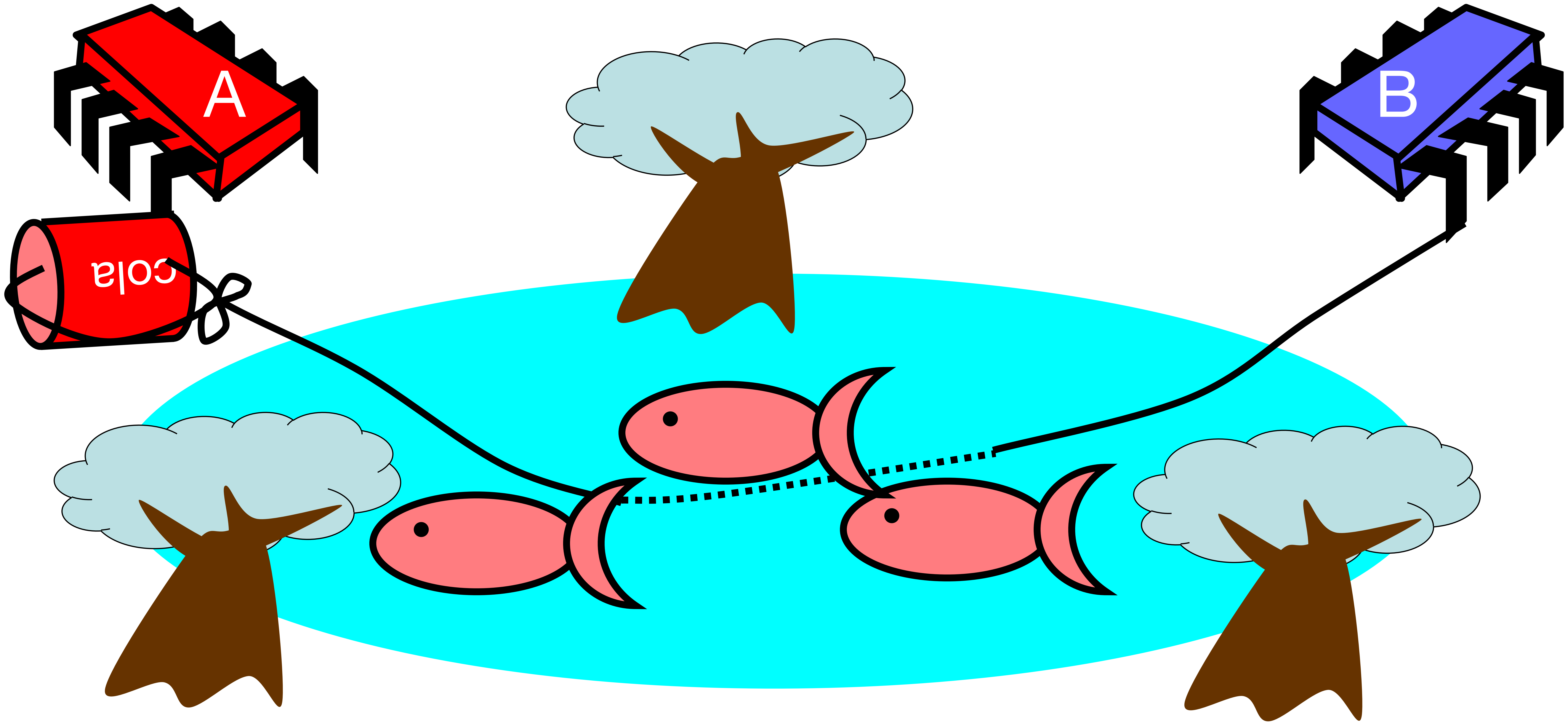
Surprise Solution



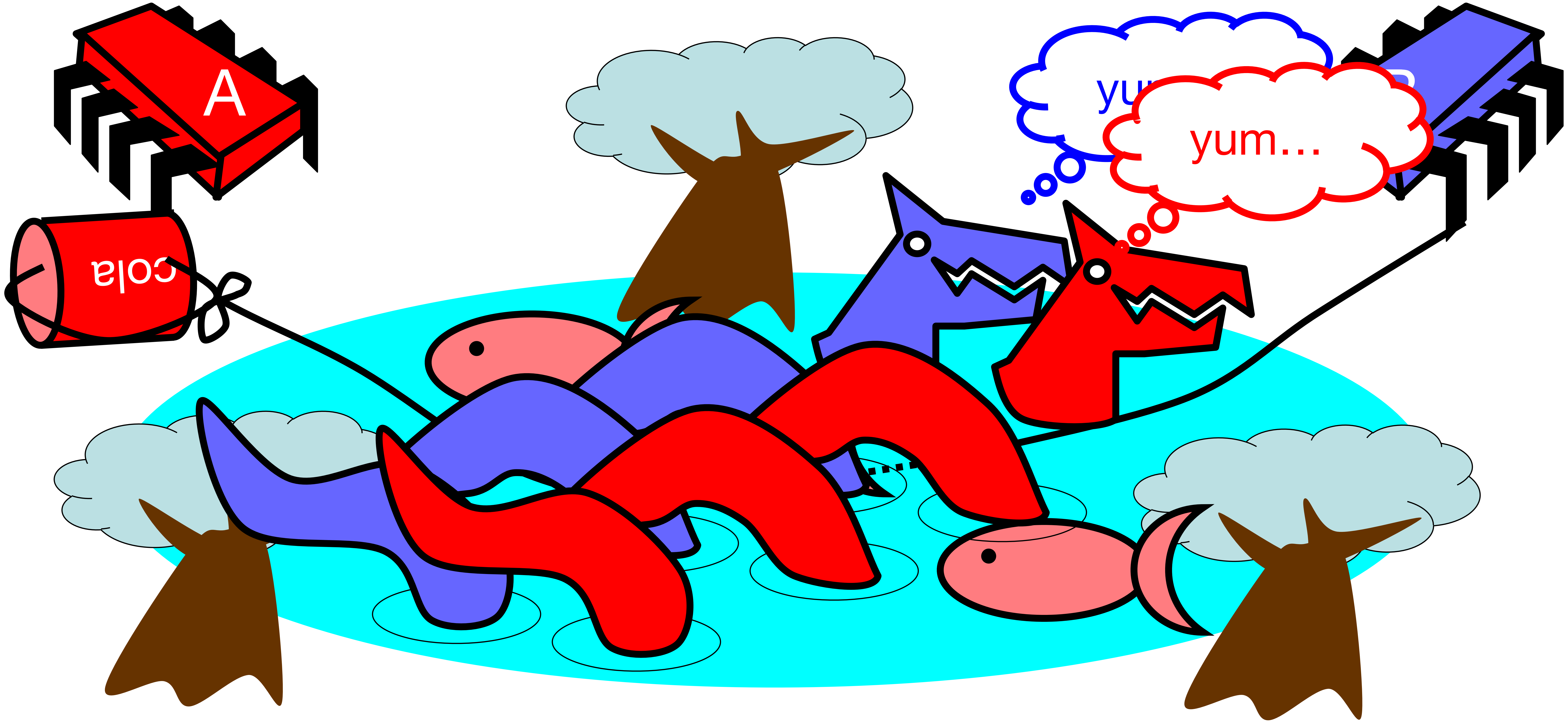
Bob puts food in the Pond



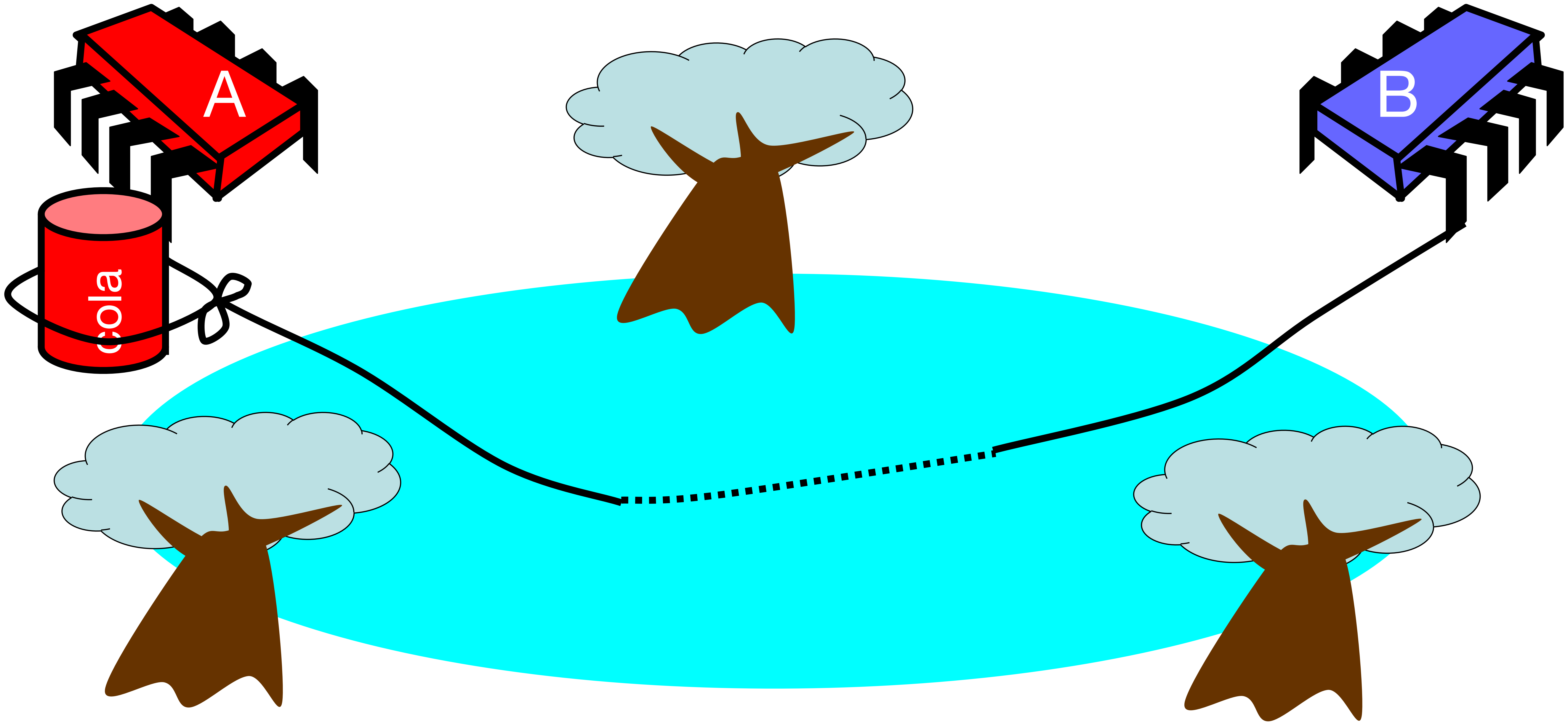
Bob knocks over can



Alice Releases Pets



Alice resets can when pets are fed



Producer/Consumer Code

```
(* Alice (Consumer) - releases pets to eat fish *)
let alice can pond =
  while true do
    while is_up can do
      ()
    done;
    (match get_food pond with
    | Some fish ->
        Printf.printf "Alice: Releasing pets to eat %s\n%!"
          (fish_to_string fish);
        Unix.sleepf 0.1;
        Printf.printf "Alice: Recapturing pets\n%!"
    | None -> failwith "impossible");
    reset can
  done
```

Producer/Consumer Code

```
(* Alice (Consumer) - releases pets to eat fish *
let alice can pond =
  while true do
    while is_up can do
      ()
    done;
    (match get_food pond with
    | Some fish ->
        Printf.printf "Alice: Releasing pets to ea
          (fish_to_string fish);
        Unix.sleepf 0.1;
        Printf.printf "Alice: Recapturing pets\n%
    | None -> failwith "impossible");
    reset can
  done
```

```
(* Bob (Producer) - stocks the pond with fish *)
let bob can pond =
  while true do
    while is_down can do
      ()
    done;
    let fish = random_fish () in
    Printf.printf "Bob: Stocking pond with %s\n%!"
      (fish_to_string fish);
    stock_pond pond fish;
    knock_over can
  done
```

Producer/Consumer Code

```
(* Alice (Consumer) - releases pets to eat fish *
let alice can pond =
  while true do
    while is_up can do
      ()
    done;
    (match get_food pond with
    | Some fish ->
        Printf.printf "Alice: Releasing pets to e
          (fish_to_string fish);
        Unix.sleepf 0.1;
        Printf.printf "Alice: Recapturing pets\n%
    | None -> failwith "impossible");
    reset can
  done
```

```
(* Bob (Producer) - stocks the pond with fish *)
let bob can pond =
  while true do
    while is_down can do
      ()
    done;
    let fish = random_fish () in
    Printf.printf "Bob: Stocking pond with %s\n%!"
     (fish_to_string fish);
    stock_pond pond fish;
    knock_over can
  done
```

Demo

Correctness

- **Mutual Exclusion**
  - Pets and Bob are never together in the pond

Correctness

- **Mutual Exclusion**
  - Pets and Bob are never together in the pond
- **No Starvation**
  - If Bob is always willing to feed, and the pets are always famished, then the pets eat infinitely often.

Correctness

- **Mutual Exclusion**
  - Pets and Bob are never together in the pond
- **No Starvation**
  - If Bob is always willing to feed, and the pets are always famished, then the pets eat infinitely often.
- **Producer/Consumer**
  - The pets never enter the pond unless there is food, and Bob never provides food if there is unconsumed food.

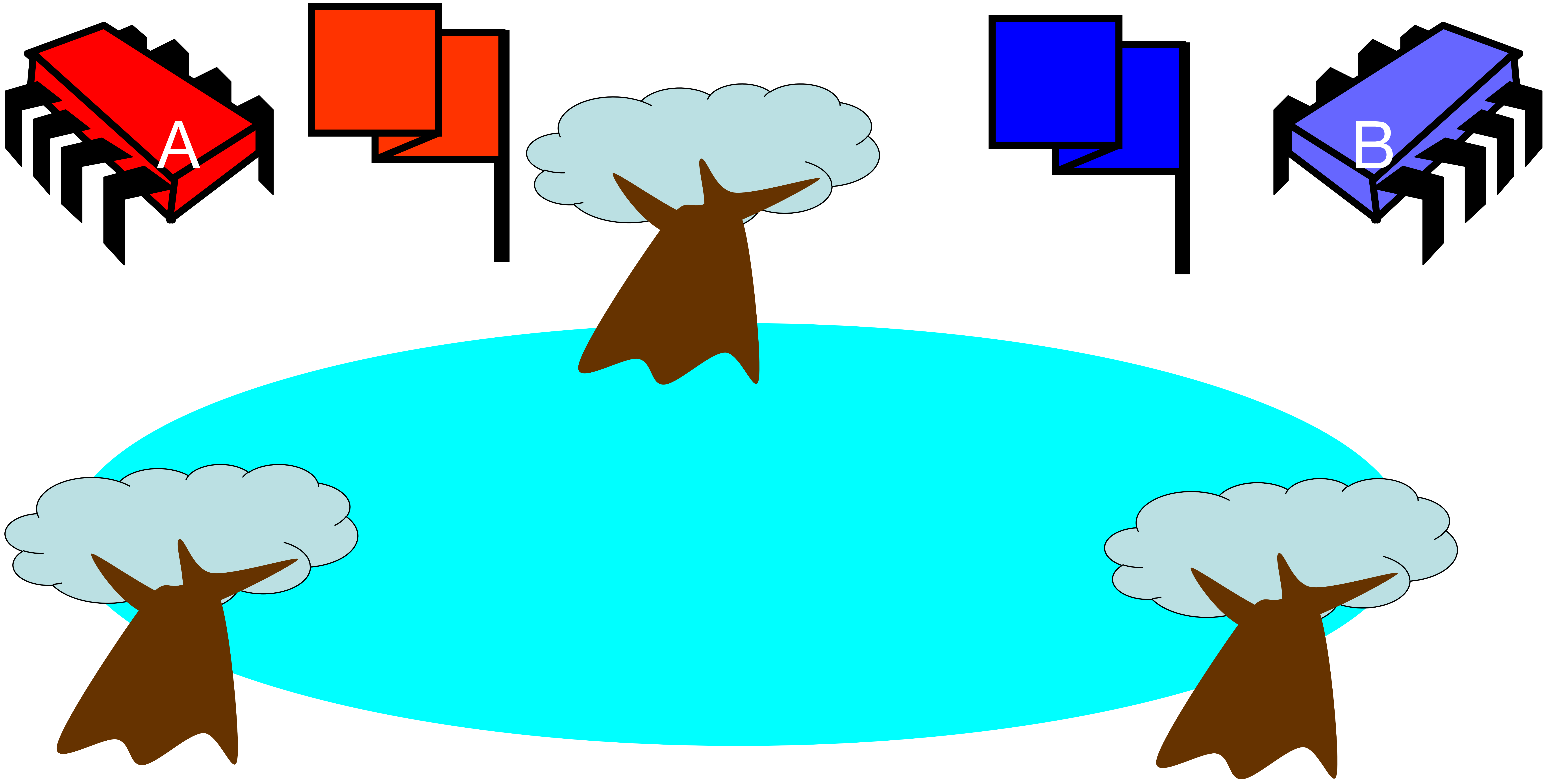
Correctness

- **Mutual Exclusion**
  - Pets and Bob are never together in the pond
- **No Starvation**
  - If Bob is always willing to feed, and the pets are always famished, then the pets eat infinitely often.
- **Producer/Consumer**
  - The pets never enter the pond unless there is food, and Bob never provides food if there is unconsumed food.
- Which of the above are ***safety*** and ***liveness*** properties?

Correctness

- **Mutual Exclusion** Safety
  - Pets and Bob are never together in the pond
- **No Starvation** Liveness
  - If Bob is always willing to feed, and the pets are always famished, then the pets eat infinitely often.
- **Producer/Consumer** Safety
  - The pets never enter the pond unless there is food, and Bob never provides food if there is unconsumed food.
- Which of the above are ***safety*** and ***liveness*** properties?

Could also solve using flags



A note on waiting

A note on waiting

- Both solutions use ***waiting***
  - `while mumble do … done`

A note on waiting

- Both solutions use ***waiting***
  - `while mumble do … done`
- In some cases waiting is ***problematic***
  - If one participant is delayed
  - So is everyone else
  - But delays are common & unpredictable

The fable drags on …

The fable drags on …

- Bob and Alice still have issues

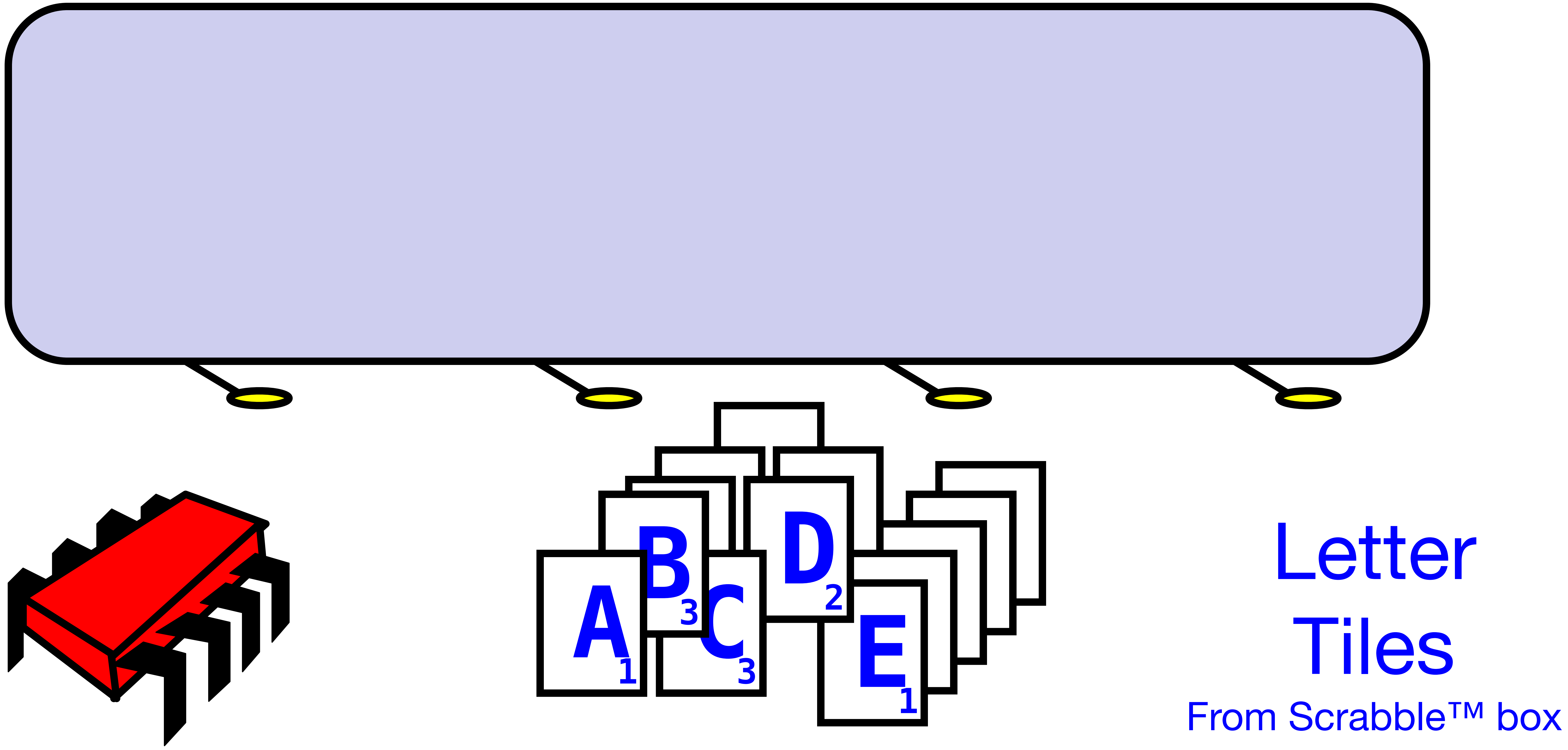
The fable drags on …

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

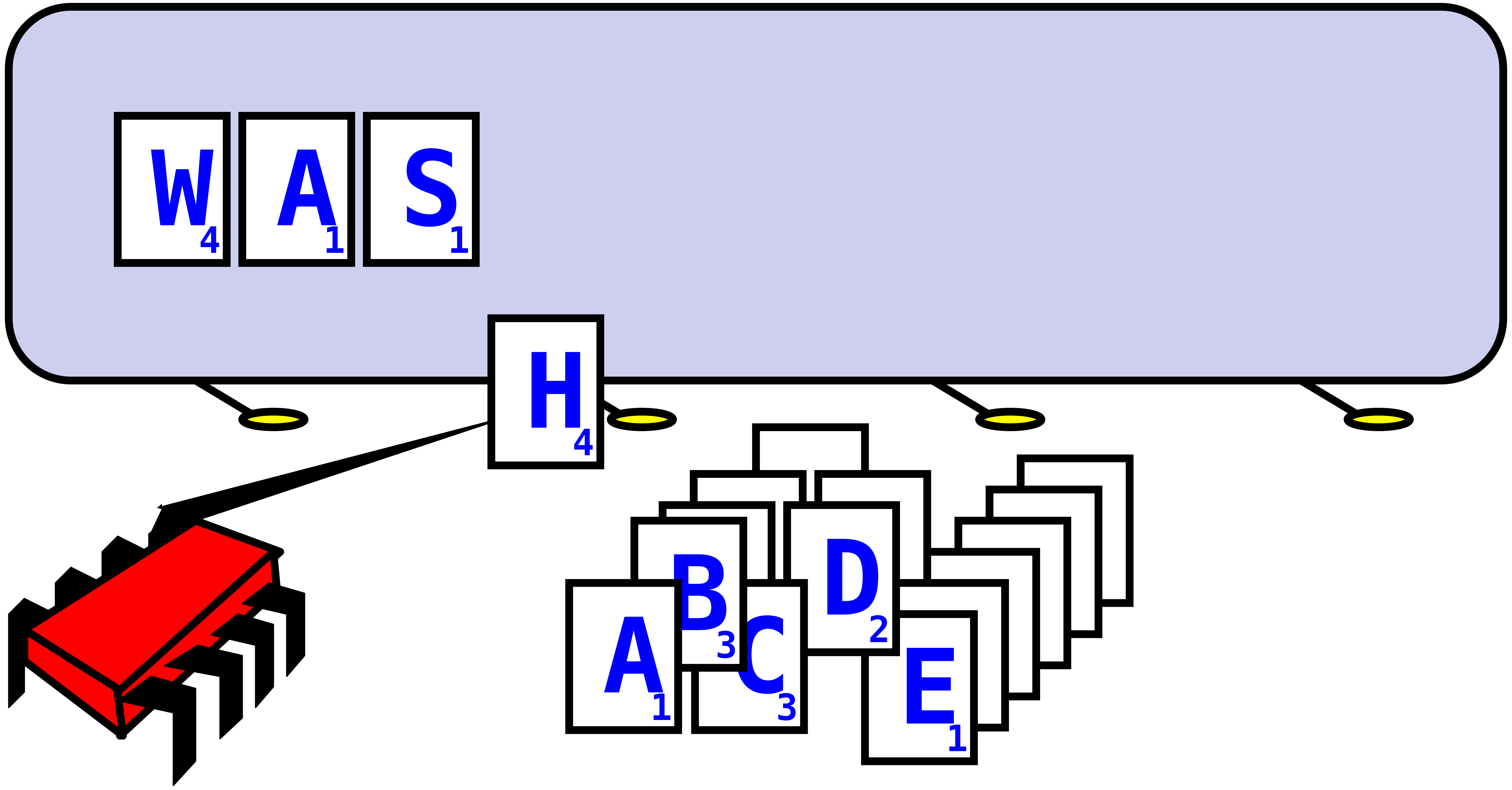
The fable drags on …

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

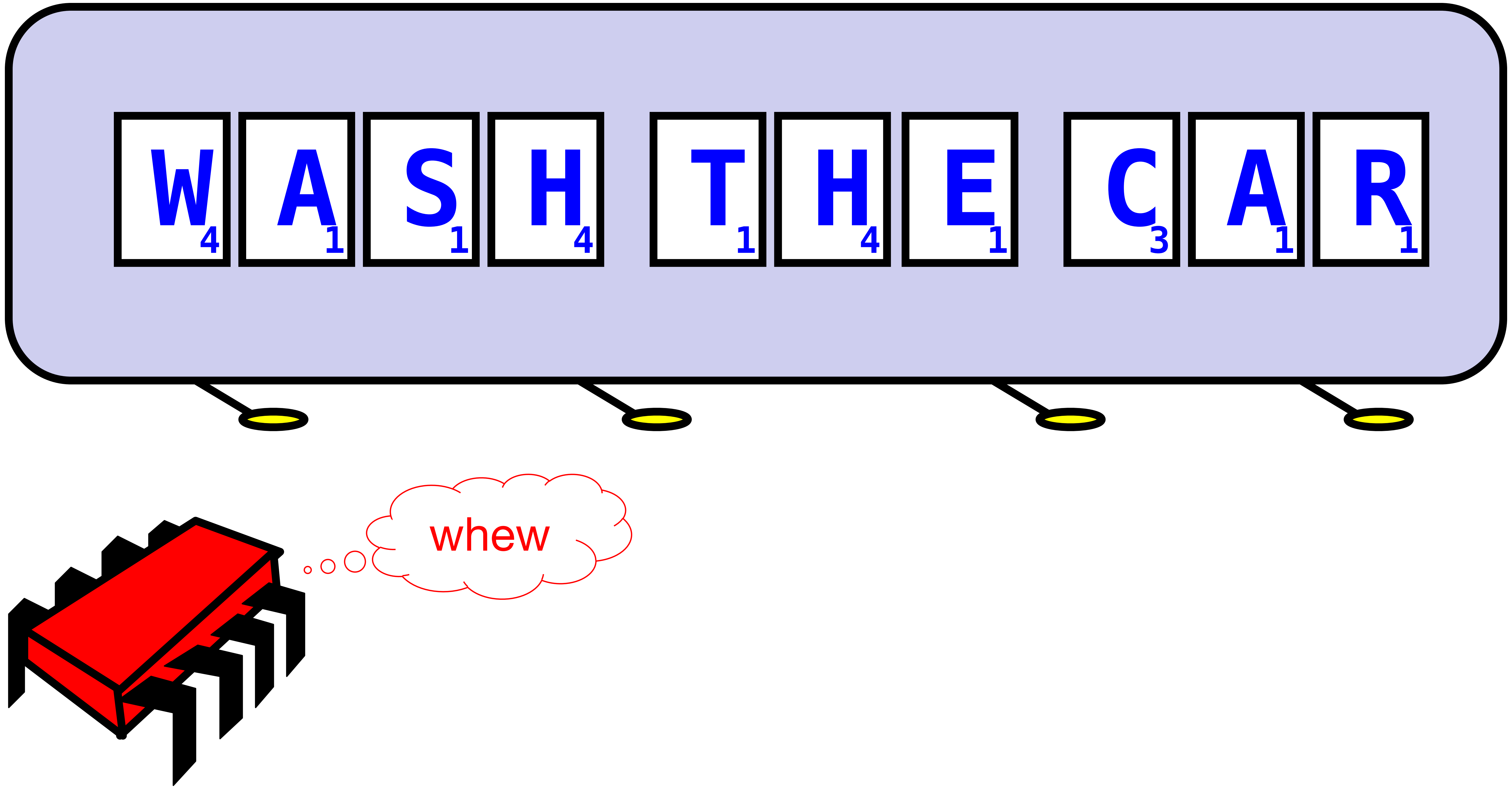
Billboards are large



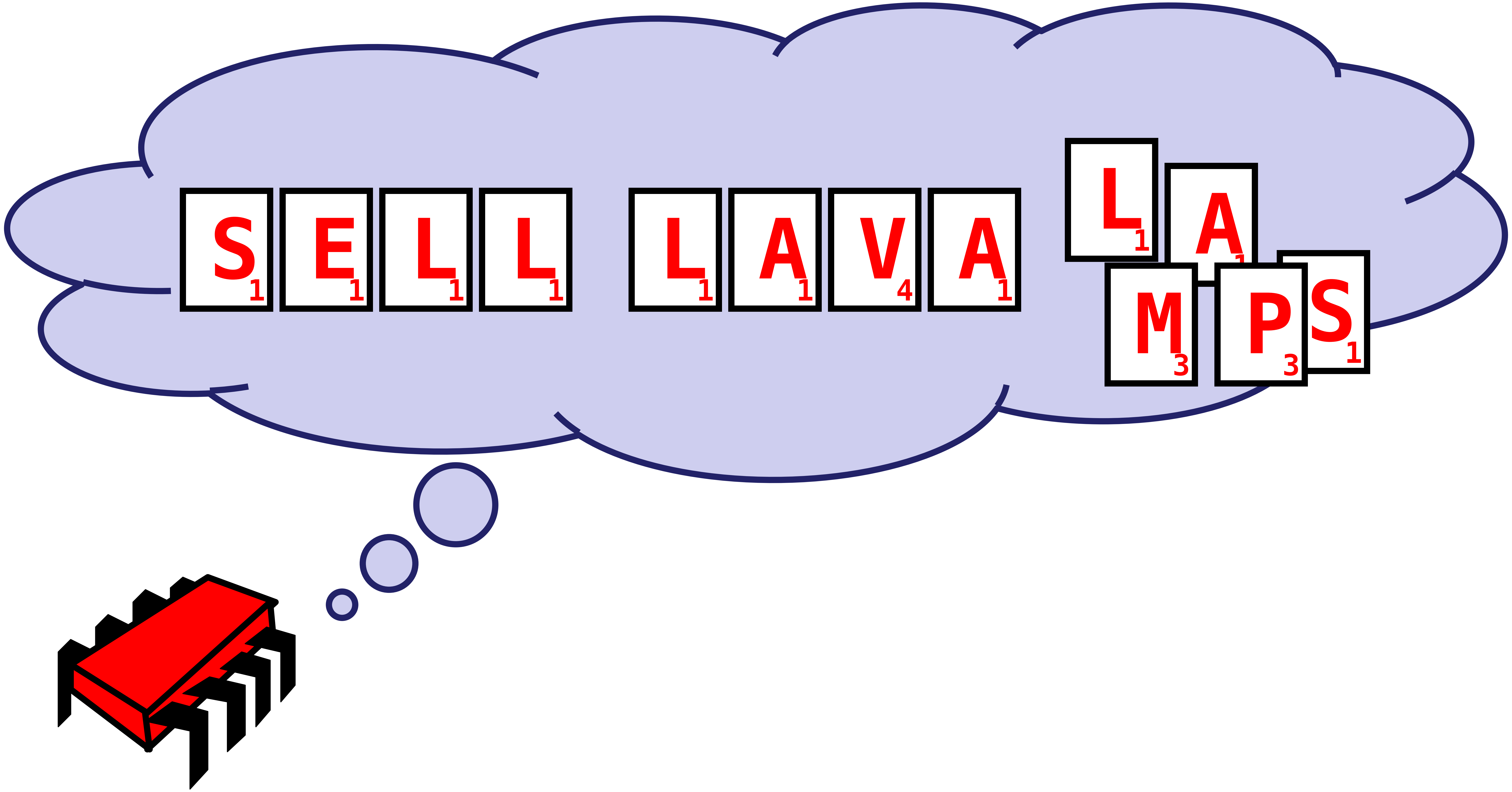
Write one letter at a time



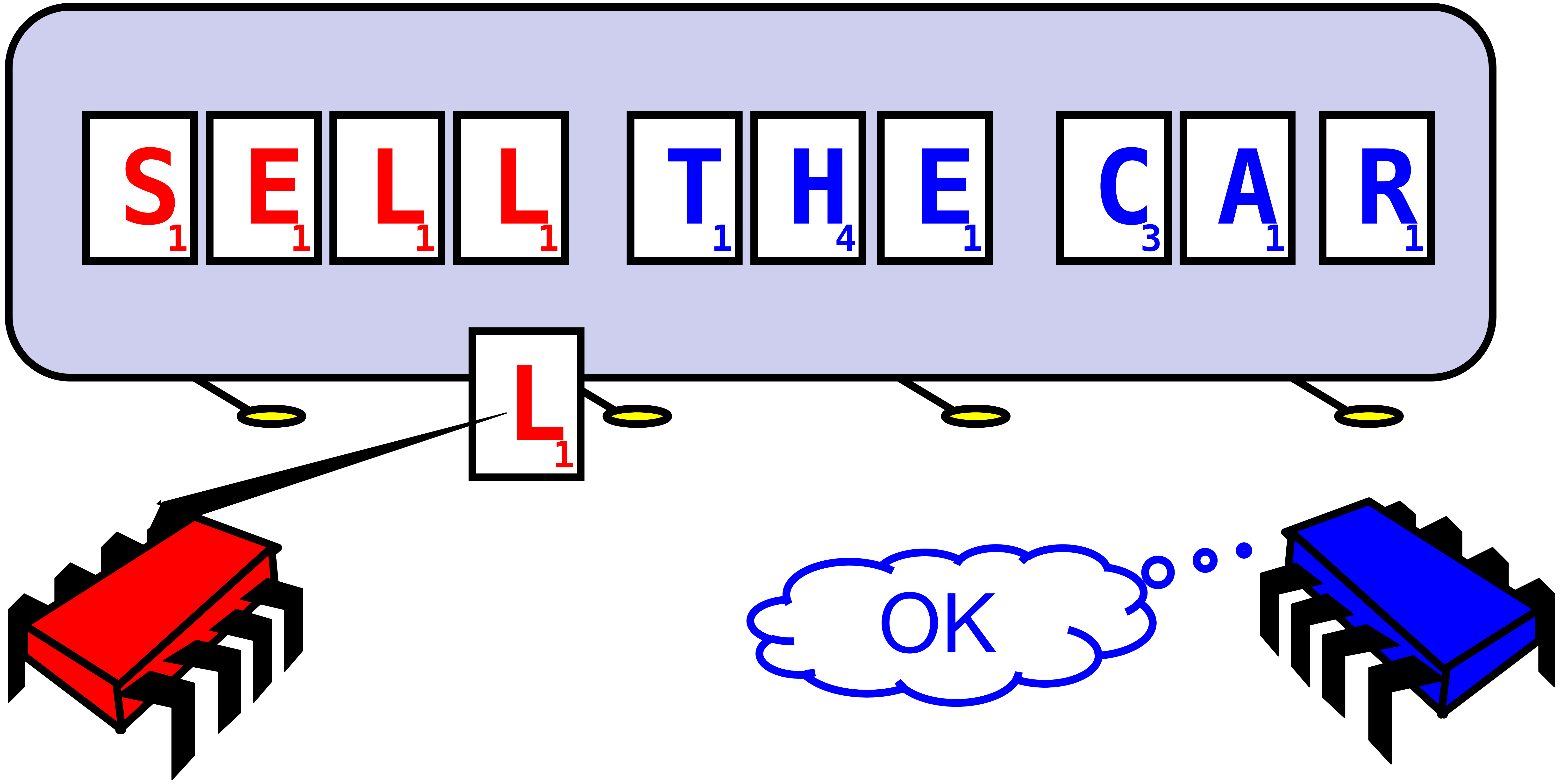
To post a message



Let’s send another message



Uh-Oh



Readers/Writers

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

- **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
- Easy with ***mutual exclusion***

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

- Easy with ***mutual exclusion***
- But mutual exclusion requires ***waiting***
  - One waits for the other
  - Everyone executes sequentially

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

- 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***
  - We will see these in later lectures

Esoteric?

Esoteric?

- Assume that you have a ***multithreading-safe, balanced, binary tree*** implementation

Esoteric?

- Assume that you have a ***multithreading-safe, balanced, binary tree*** implementation
- Multiple ***concurrent reads*** can be done without synchronisation

Esoteric?

- Assume that you have a ***multithreading-safe, balanced, binary tree*** implementation
- Multiple ***concurrent reads*** can be done without synchronisation
- Even ***concurrent update*** operations, `add` or `delete`, may modify disjoint parts of the tree
  - May be done in parallel without blocking each other

Esoteric?

- Assume that you have a ***multithreading-safe, balanced, binary tree*** implementation
- Multiple ***concurrent reads*** can be done without synchronisation
- Even ***concurrent update*** operations, `add` or `delete`, may modify disjoint parts of the tree
  - May be done in parallel without blocking each other
- You want a `size` function to return the number of elements in the tree.
  - Every update operation needs to modify `size`
  - Threads need to wait for exclusive access to the counter!

Esoteric?

- Assume that you have a ***multithreading-safe, balanced, binary tree*** implementation
- Multiple ***concurrent reads*** can be done without synchronisation
- Even ***concurrent update*** operations, `add` or `delete`, may modify disjoint parts of the tree
  - May be done in parallel without blocking each other
- You want a `size` function to return the number of elements in the tree.
  - Every update operation needs to modify `size`
  - Threads need to wait for exclusive access to the counter

performance bottleneck

`Size()` Readers/Writers Solution

- Maintain a global array `count[i]`, indexed by the thread ID
- Each thread `i` keeps a count of |add| – |remove| in the `count` array
- Function `size()` now only needs to read a ***"consistent snapshot"*** of the `count` array
  - 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

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

Amdahl's Law

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

Amdahl's Law

Parallel fraction

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

Amdahl's Law

Sequential fraction

Parallel fraction

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

Amdahl's Law

Sequential fraction

Parallel fraction

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

Number of threads

Amdal’s Law

Bad synchronization ruins everything

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

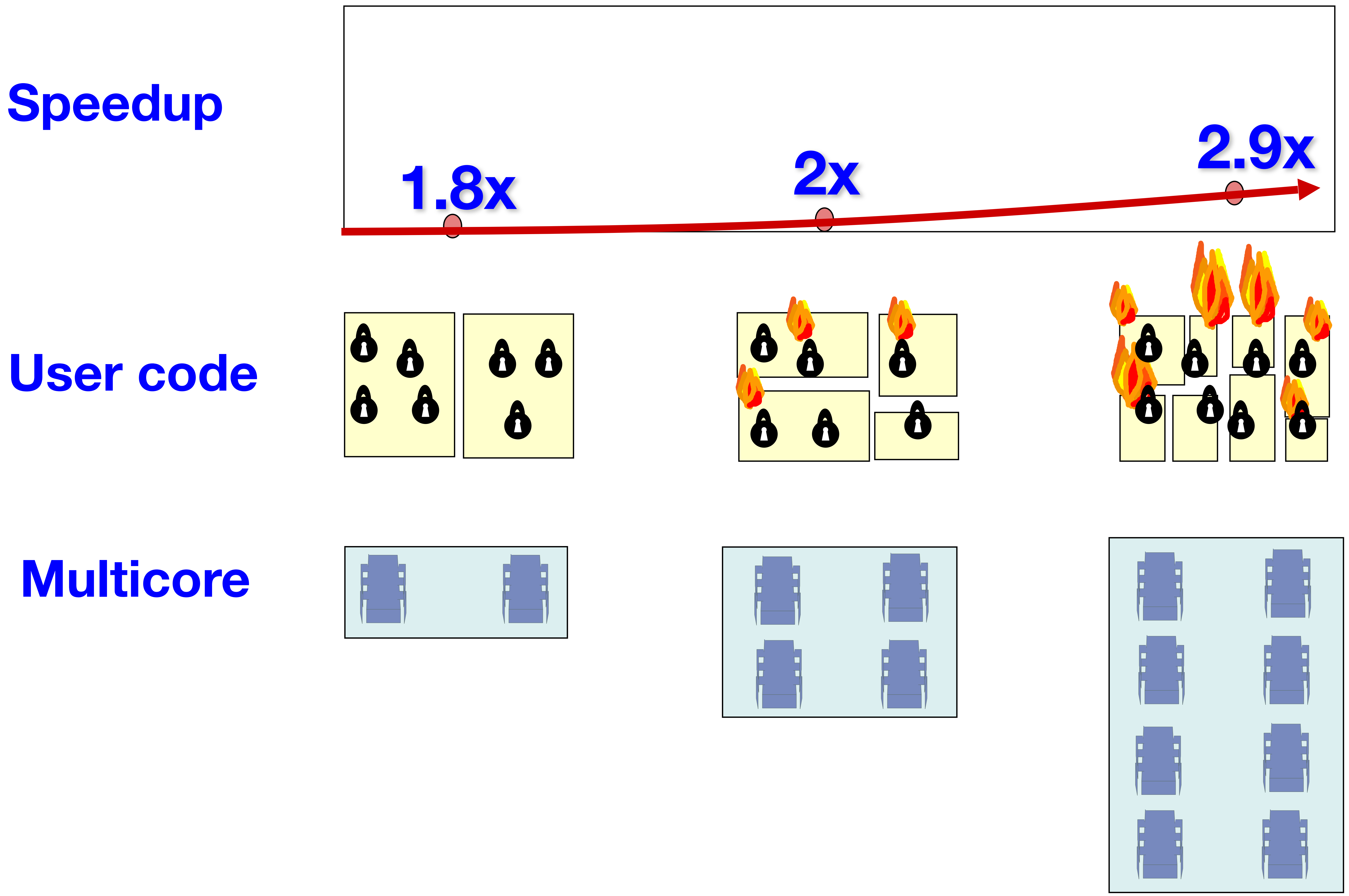
- Ten processors
- 99% concurrent, 1% sequential
- ***How close to 10-fold speedup?***

Example

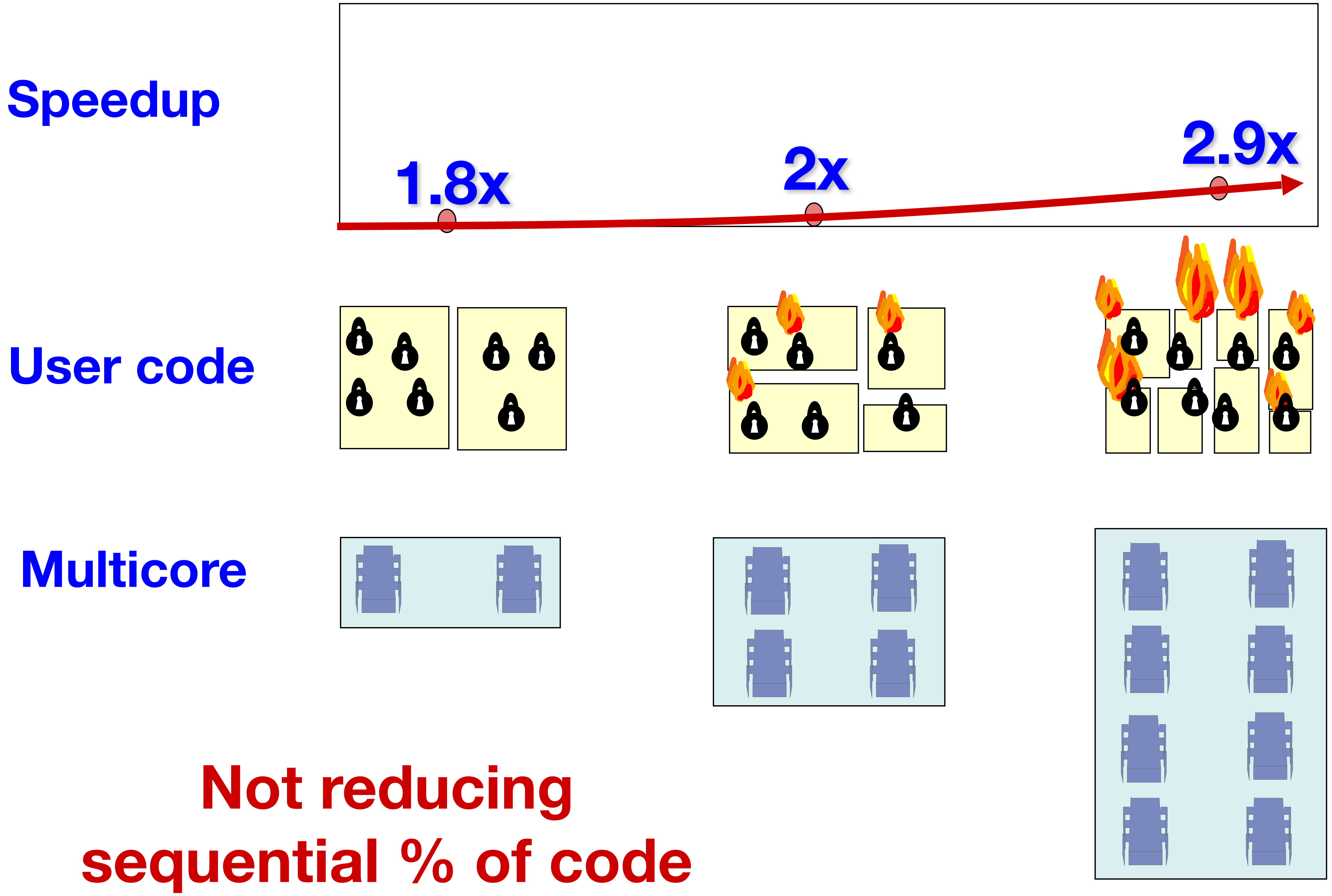
- Ten processors
- 99% concurrent, 1% sequential
- ***How close to 10-fold speedup?***

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

Back to real-world multicore scaling

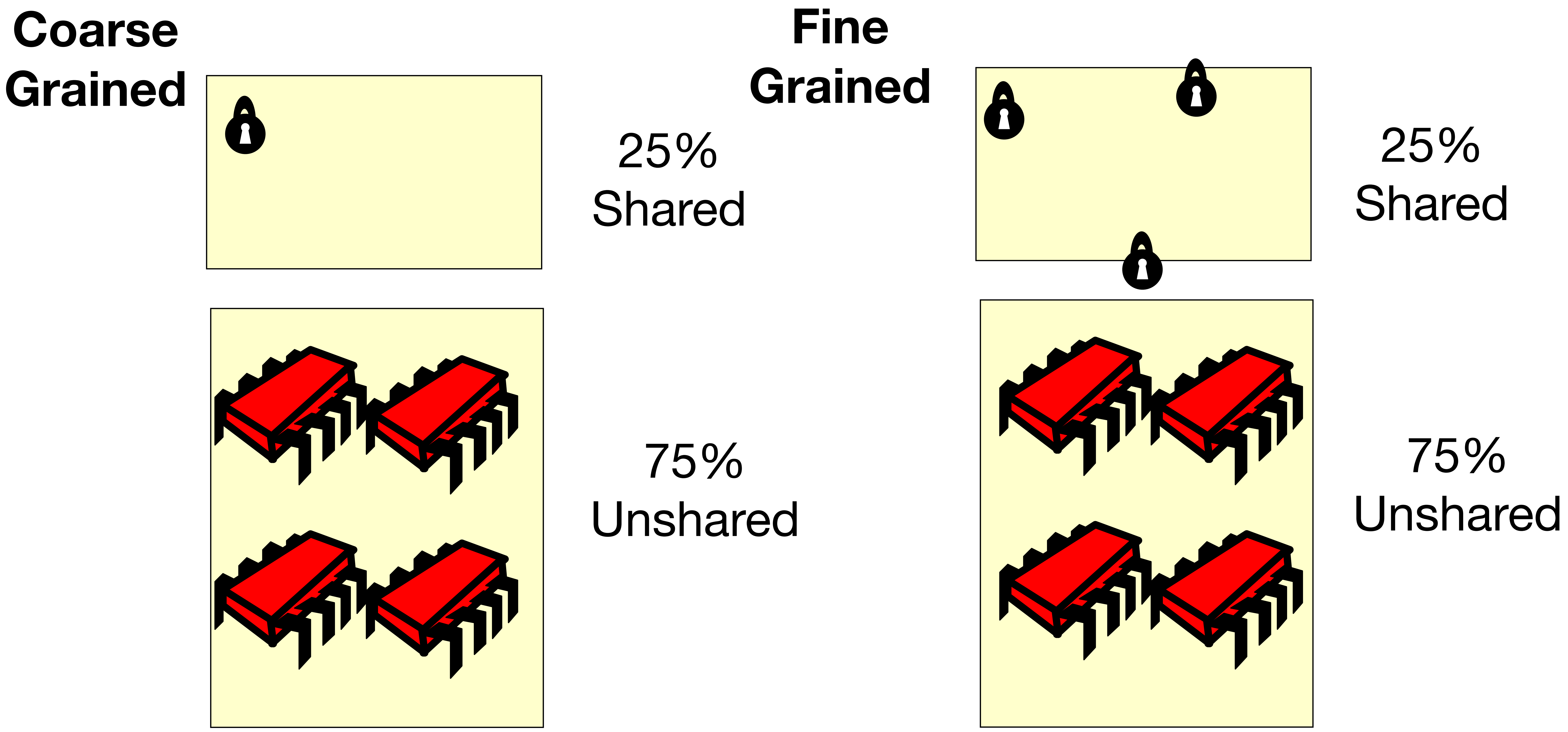


Back to real-world multicore scaling

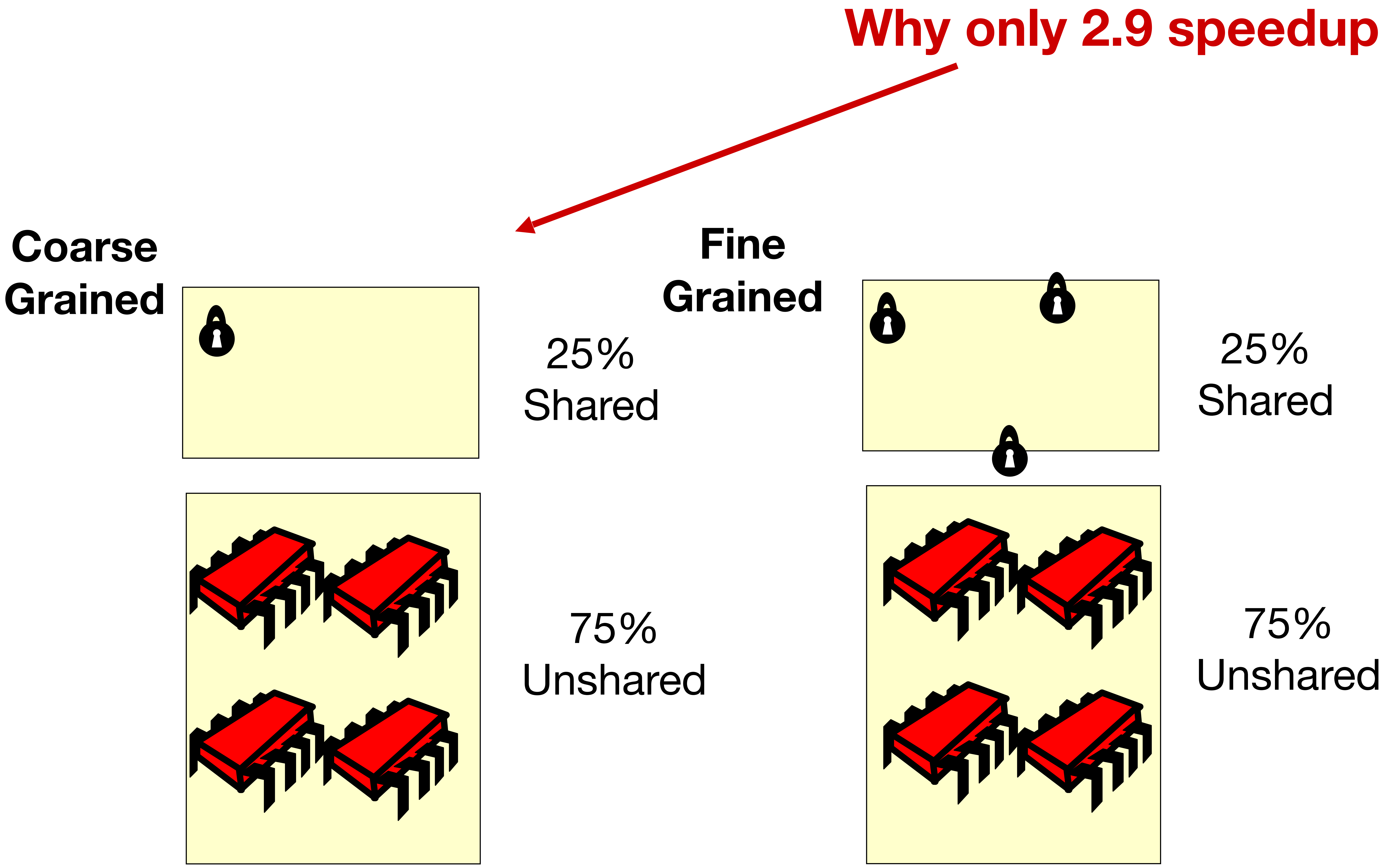


Not reducing
sequential % of code

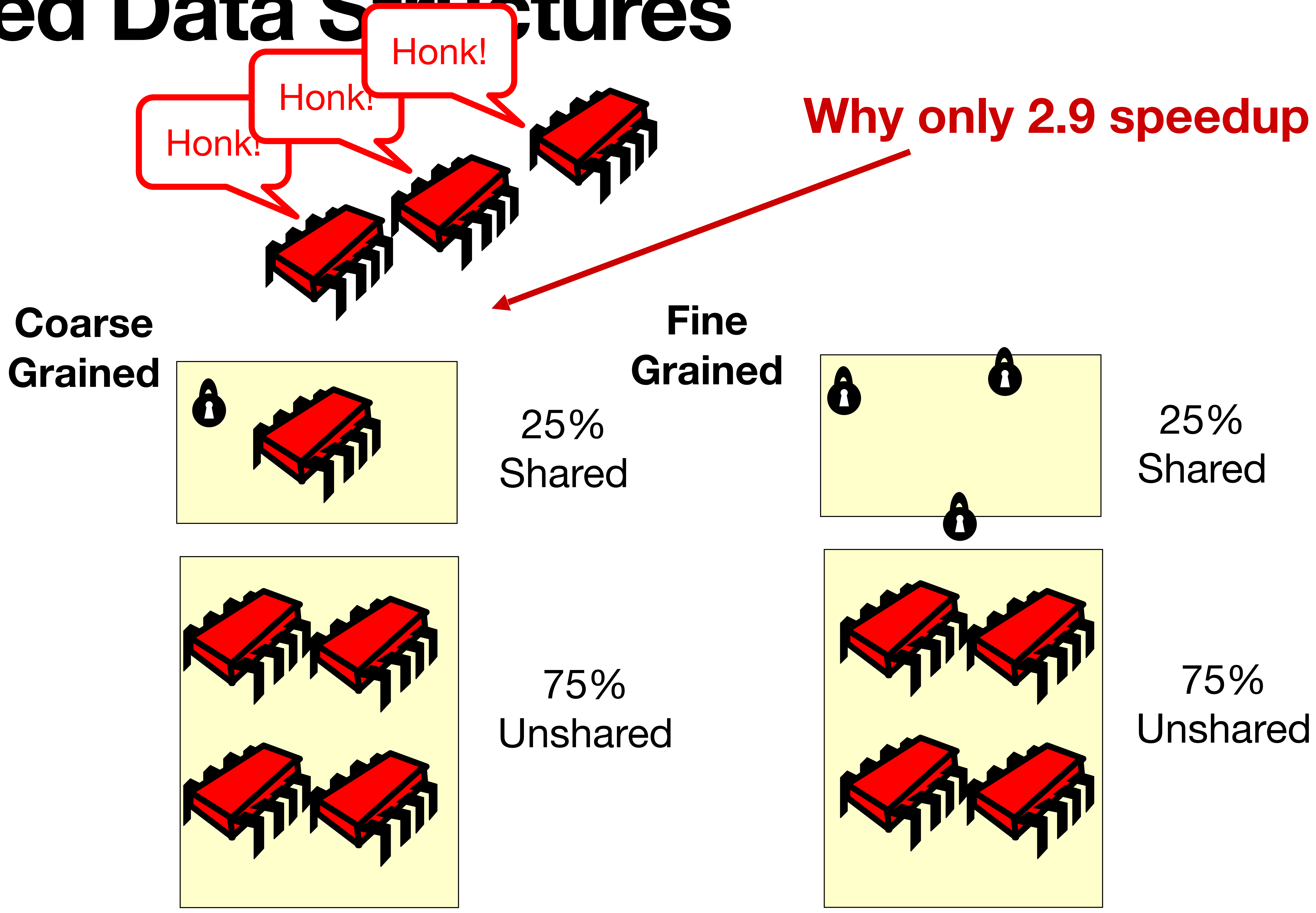
Shared Data Structures



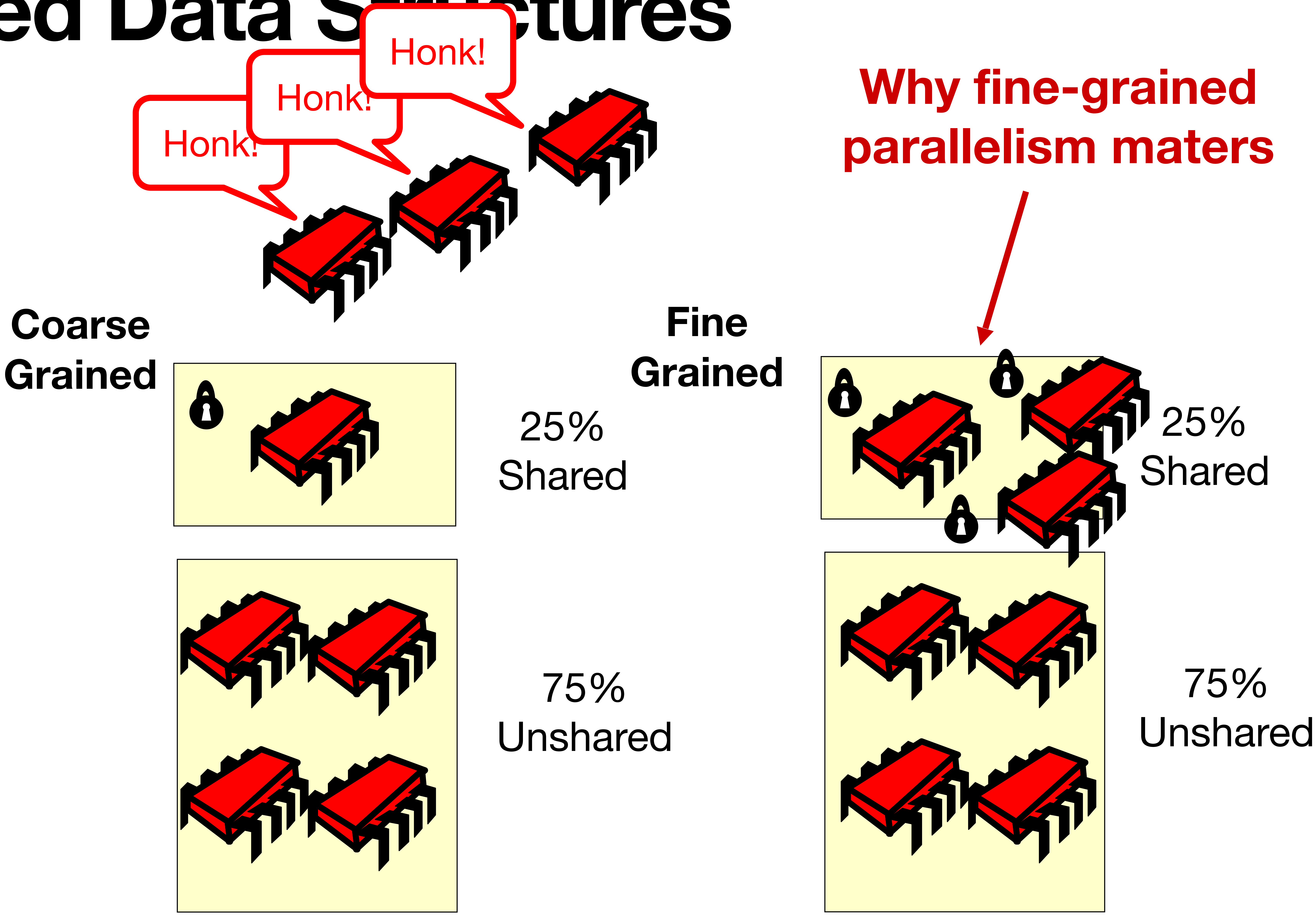
Shared Data Structures



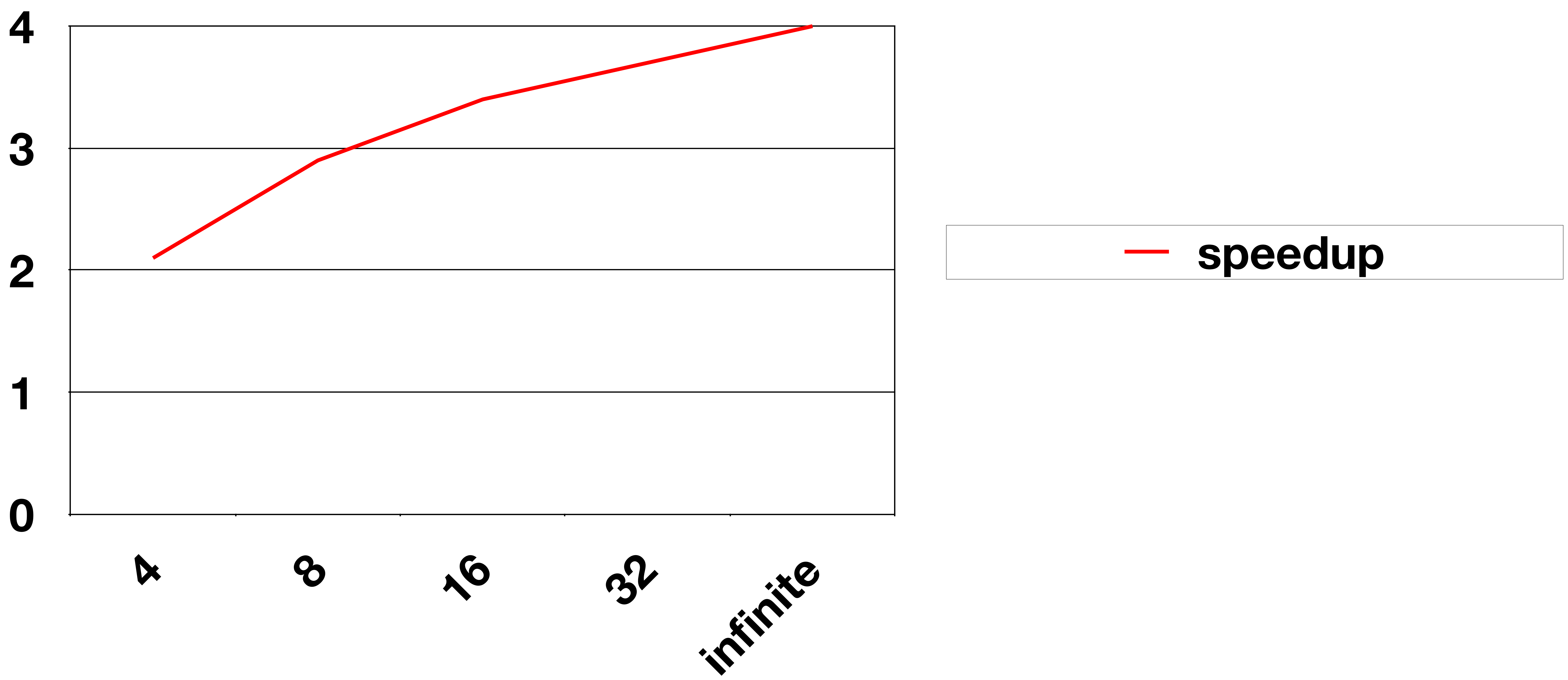
Shared Data Structures



Shared Data Structures



Diminishing returns



*This course is about the parts that are hard to make concurrent …
but still have a big influence on speedup!*

Fin