

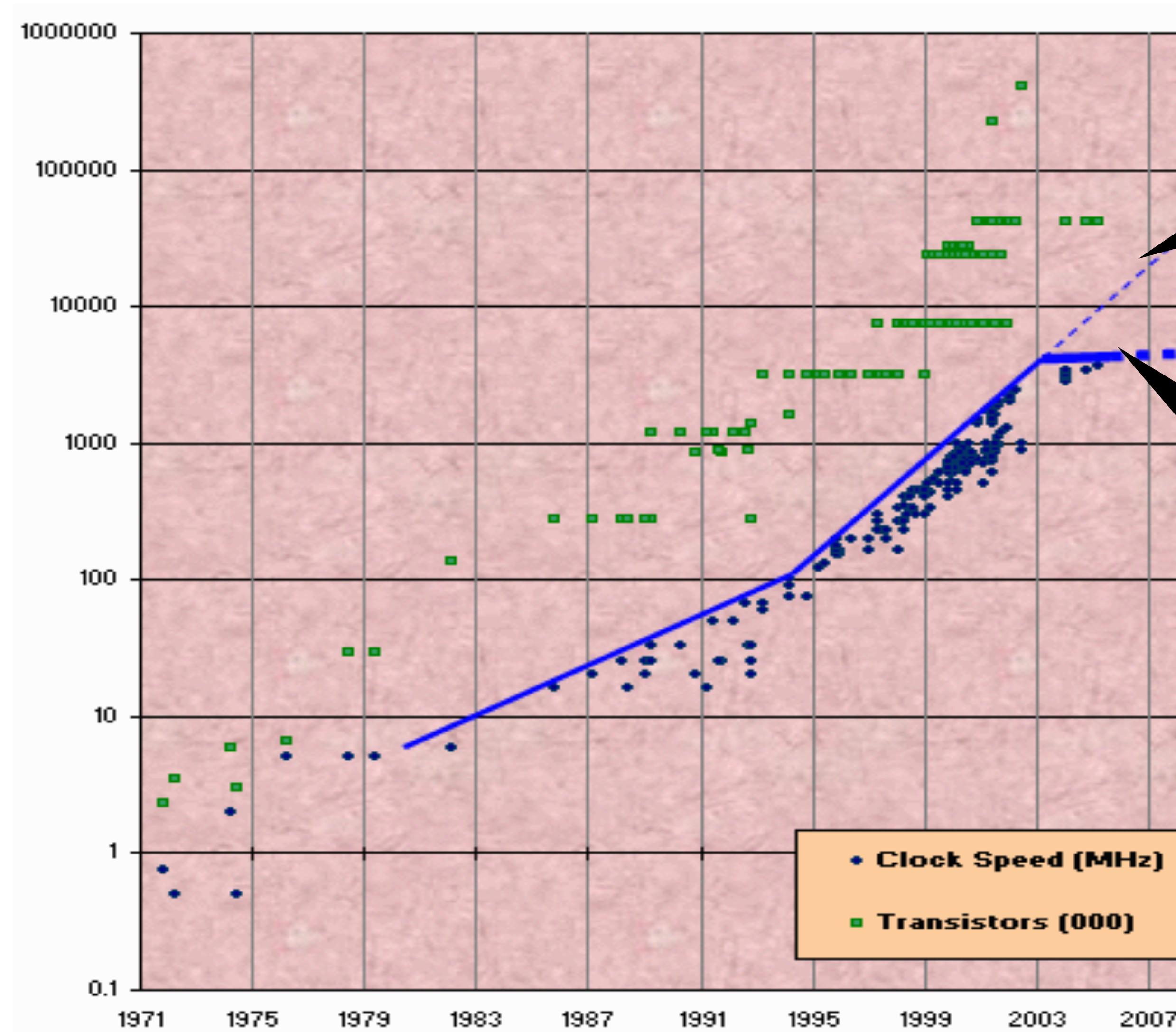
# YSC4231: Parallel, Concurrent and Distributed Programming

Ilya Sergey

[ilya.sergey@yale-nus.edu.sg](mailto:ilya.sergey@yale-nus.edu.sg)

<https://ilyasergey.net/YSC4231>

# Moore's Law



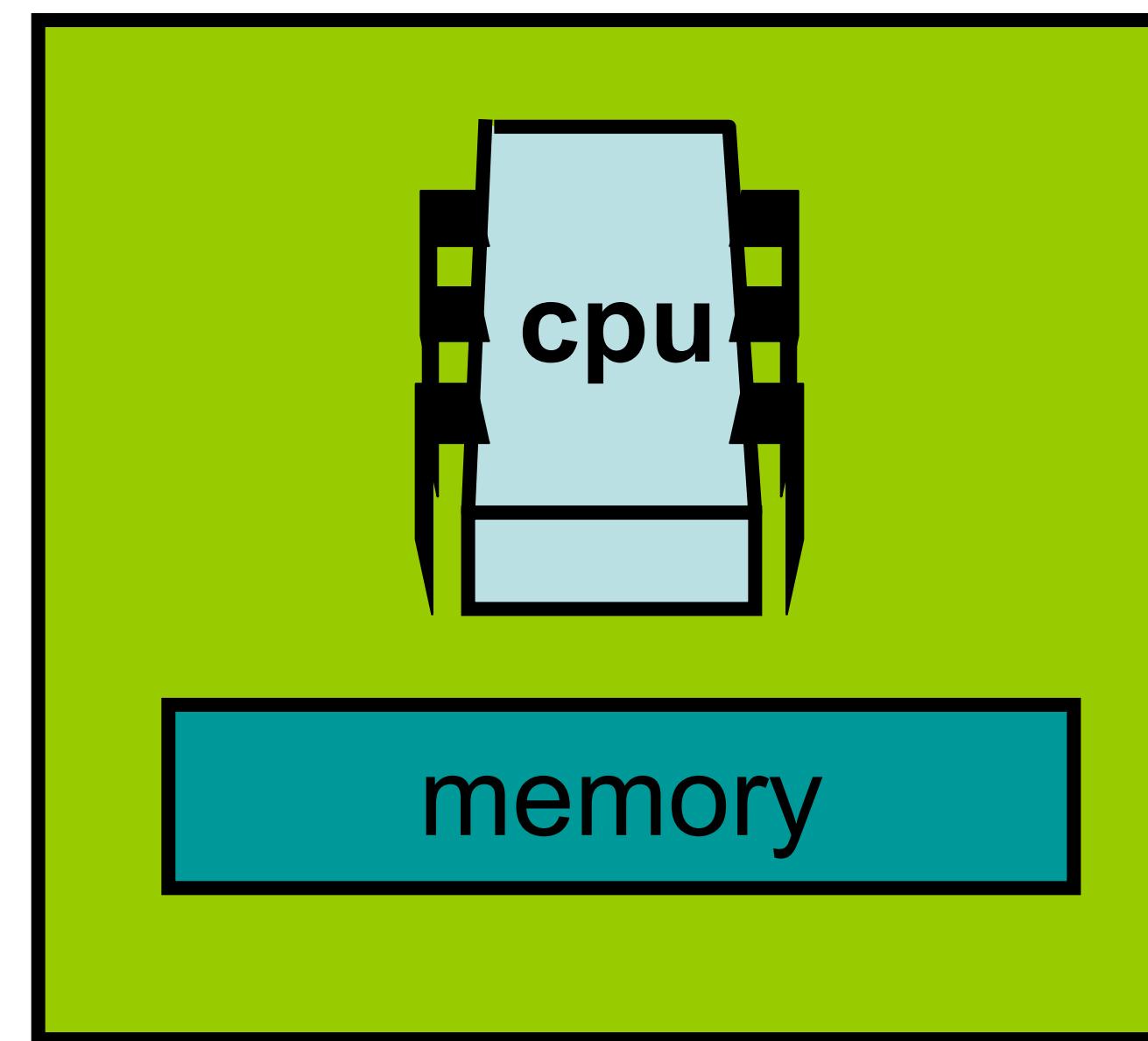
Transistor  
count still  
rising

Clock  
speed  
flattening  
sharply

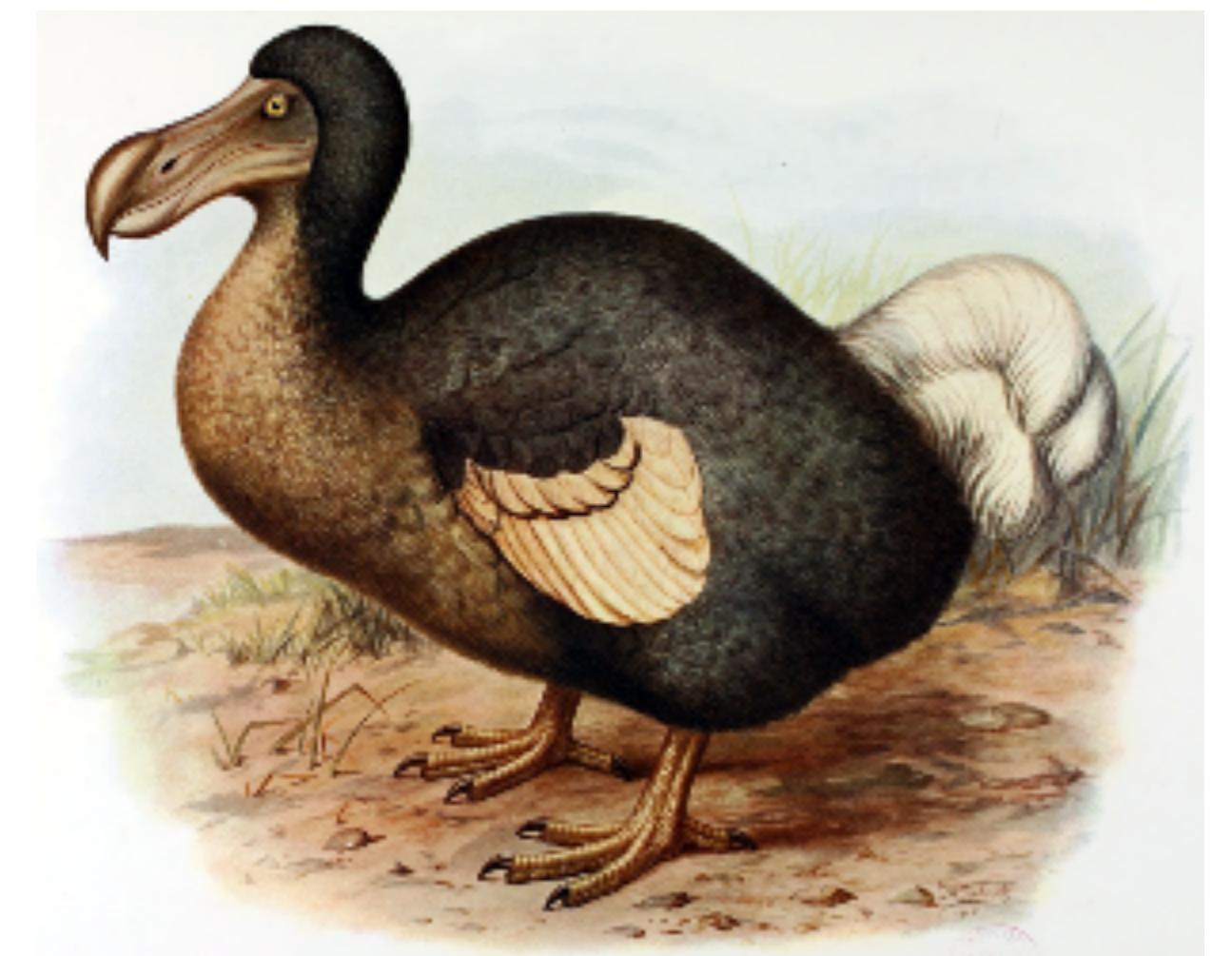
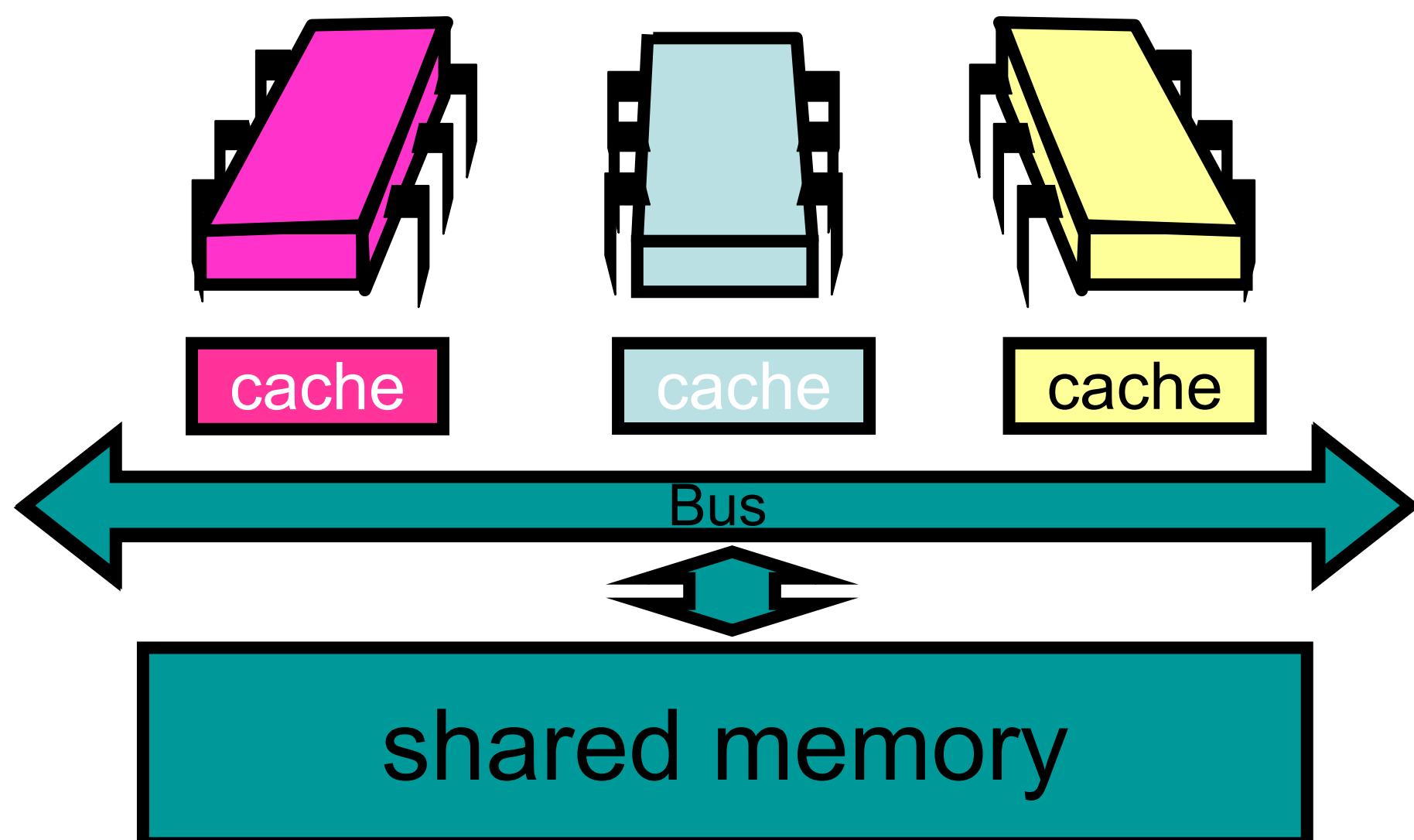
# Moore's Law (in practice)



# Extinct: the Uniprocessor

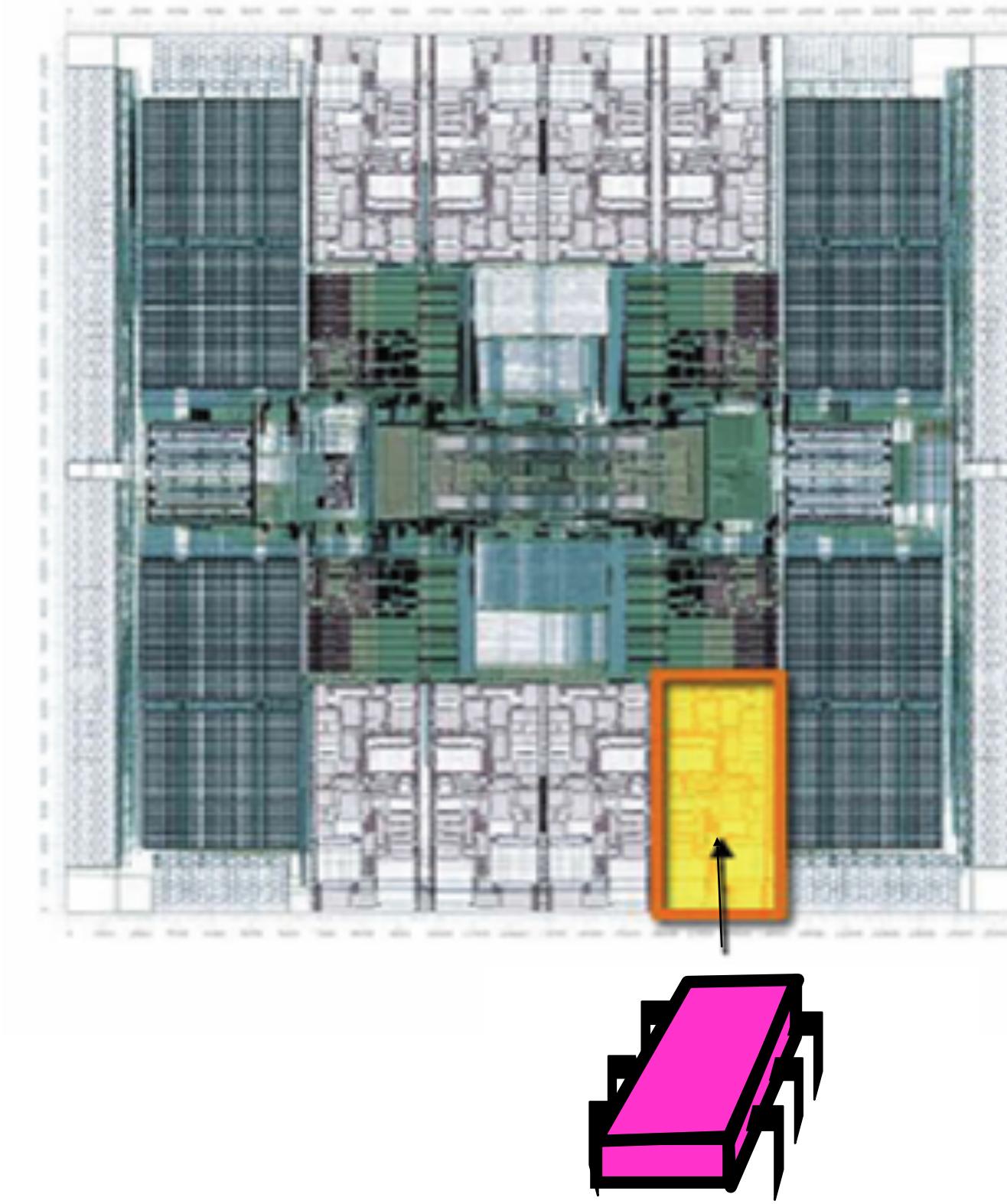


# Extinct: The Shared Memory Multiprocessor (SMP)



# The New Boss: The Multicore Processor (CMP)

All on the  
same chip

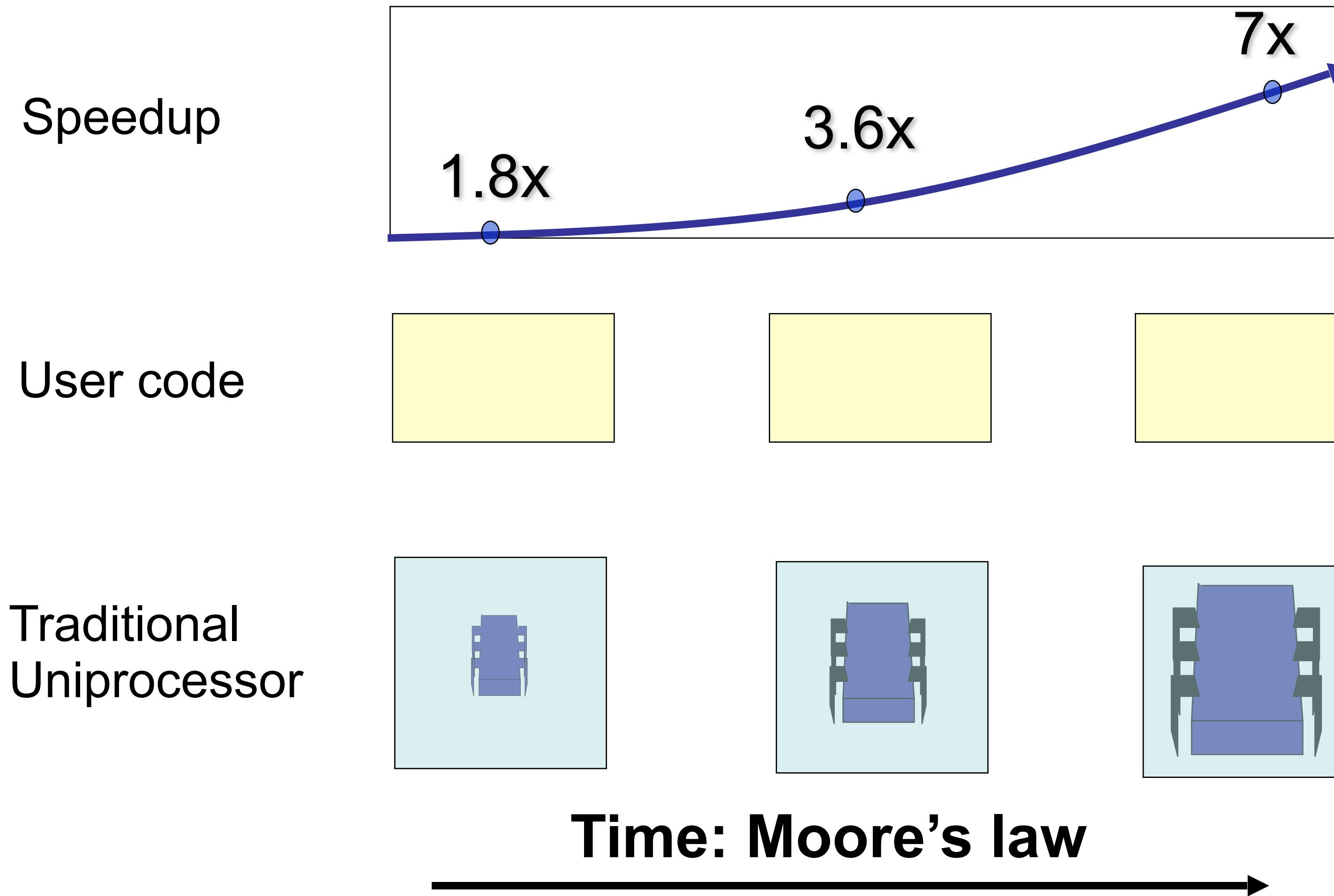


**Sun  
T2000  
Niagara**

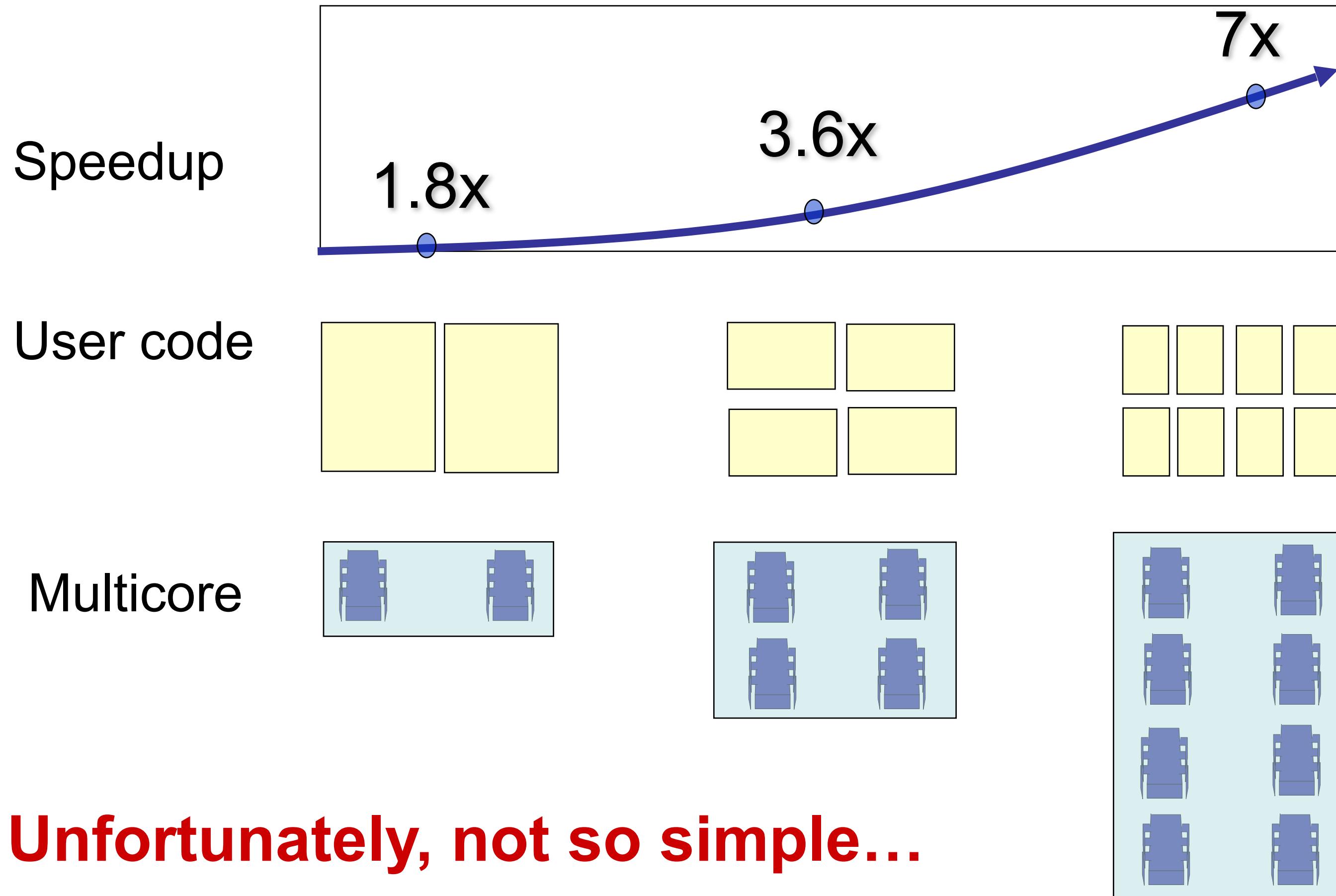
# Why do we care?

- 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

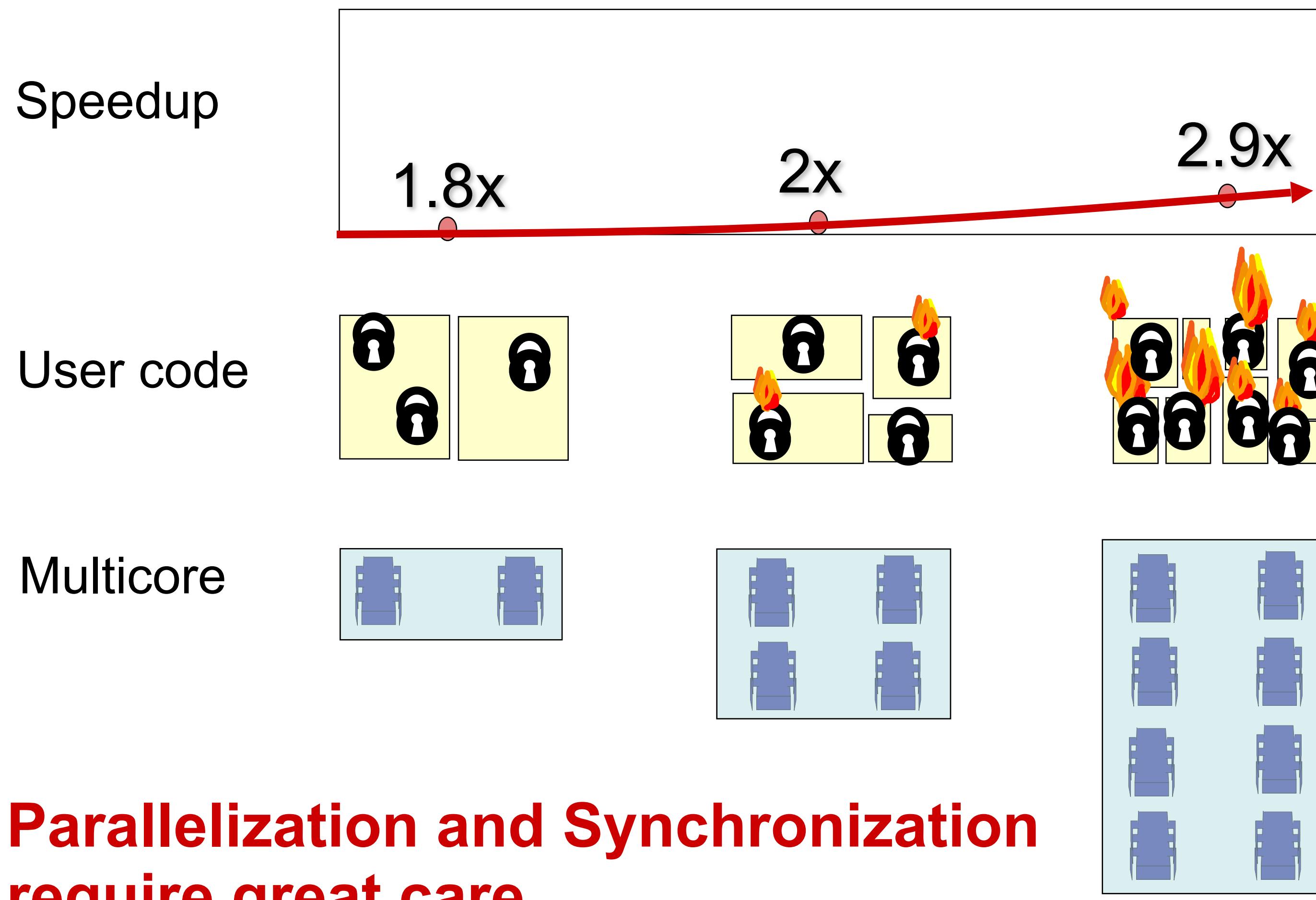
# Traditional Scaling Process



# Ideal Scaling Process

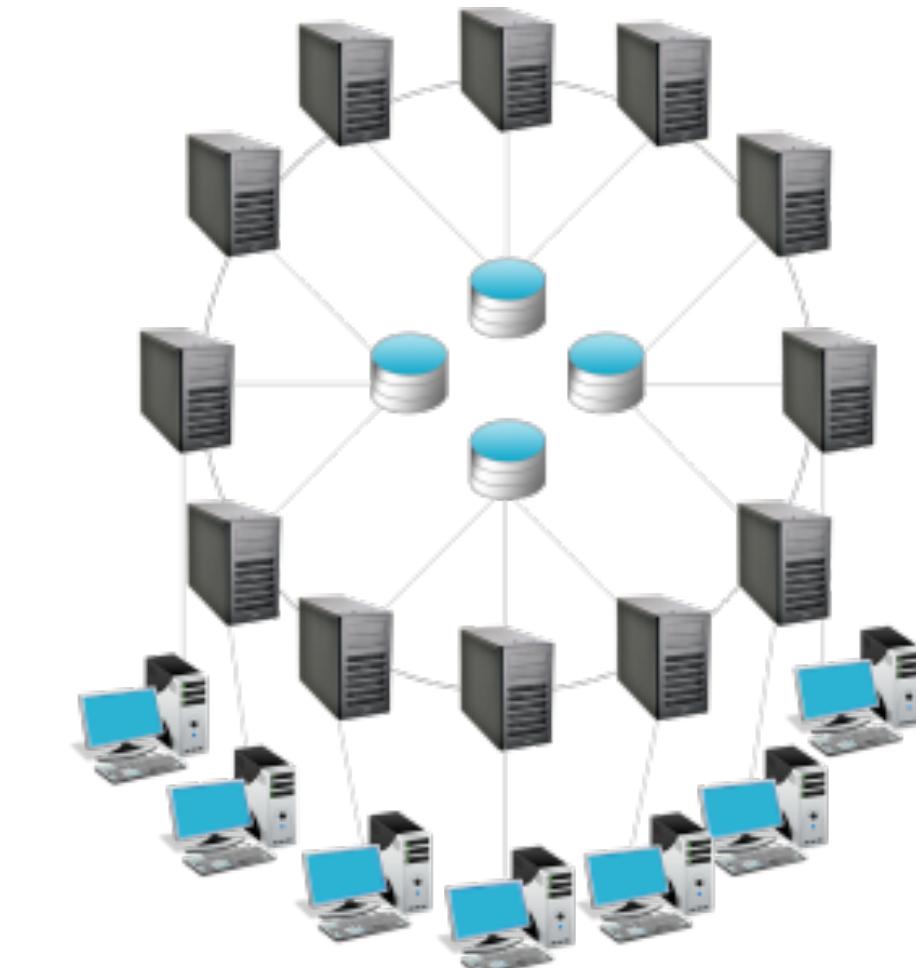


# Actual Scaling Process



# What this course is about?

- Writing *efficient* code by exploiting the *parallelism* offered by modern multiprocessors by means of writing *concurrent* programs
- Designing *concurrent* algorithms and data structures (executing on the same computer, possibly in parallel)
- Avoiding common mistakes when writing *concurrent* code; *formally reasoning* about its *correctness*.
- Basics of *distributed computing* (over multiple computers) in the presence of *communication faults*.



# Programming Language

- A mix of functional and object-oriented programming  
*(suitable for both OCaml and Java/C++ hackers)*
- Supports almost all styles of concurrency  
(shared-memory, message-passing, transactional memory, etc.)
- Type-safe, garbage-collected.
- Interoperability with Java, compiling into JVM (Java Virtual Machine)
- Great IDE support (we'll be using IntelliJ IDEA with Scala plugin)



# Grading

- 3 Written Theory Assignments: 18%
- 7 Programming Assignments: 42%
- 1 Research Mini-project (groups of 2): 6%
- Mid-Term Project: 15%
- Final Project: 15%
- In-class participation: 4%

# Homework

- Two types: theoretical and programming assignments
- Complete *individually*
- Deliverables:
  - a PDF with typeset answers (theory) and occasionally some code
  - a link to a tagged GitHub release (programming)
- Each assignment is graded out of 20 points

# Collaboration

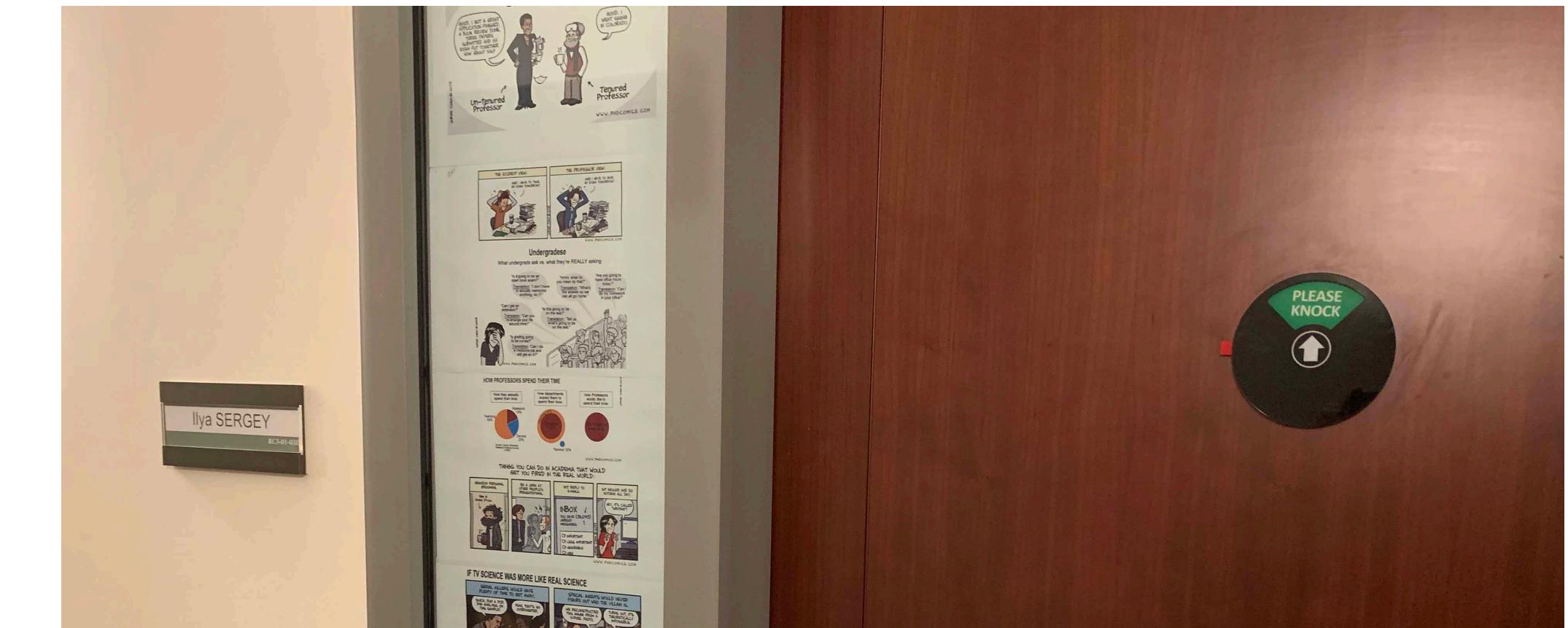
- **Permitted:**
  - Talking about the homework problems with the peer tutor
  - Using other textbooks
  - Using the Internet for documentation on Scala and Java.
- **Not permitted:**
  - Obtaining the answer directly from anyone or anything else in any form
  - Adapting a solution from a similar one found on the internet
  - “Copying with understanding” from other resources
  - 1st strike: 0 points for assignment
  - 2nd strike: F for the module, the case is passed to the Acad. Integrity Committee

# Late Hand-ins and Resubmissions

- Projects that **don't compile** will get no credit and can't be resubmitted.
- Work submitted *before* the deadline and receiving <18 points and  $\geq 6$  points can be resubmitted *within one week* after the grades are posted on Canvas.
  - The amended grade is left at the instructor's discretion.
  - Work that scored less than 6 points can't be resubmitted
- Late submissions will be penalised by subtracting  $(2 + \text{full days after deadline})$  points from the maximal score (20).
  - Late submissions cannot be resubmitted.
- No resubmissions for the **midterm** and **final** project.

# Getting Help

- Office Hours (#RC3-01-03E, Cendana):  
Mondays, 3:30pm-5:00pm;  
Wednesdays, 5:15pm-6:30pm  
**Please, email upfront!**



- **E-mail policy:** questions about homework assignments sent less than 24 hours before submission deadline **won't be answered**.
- **Exception:** bug reports.

# Peer Tutor

Mark Yuen

[mark.yuen@u.yale-nus.edu.sg](mailto:mark.yuen@u.yale-nus.edu.sg)

- Tutoring Hours:

by appointment



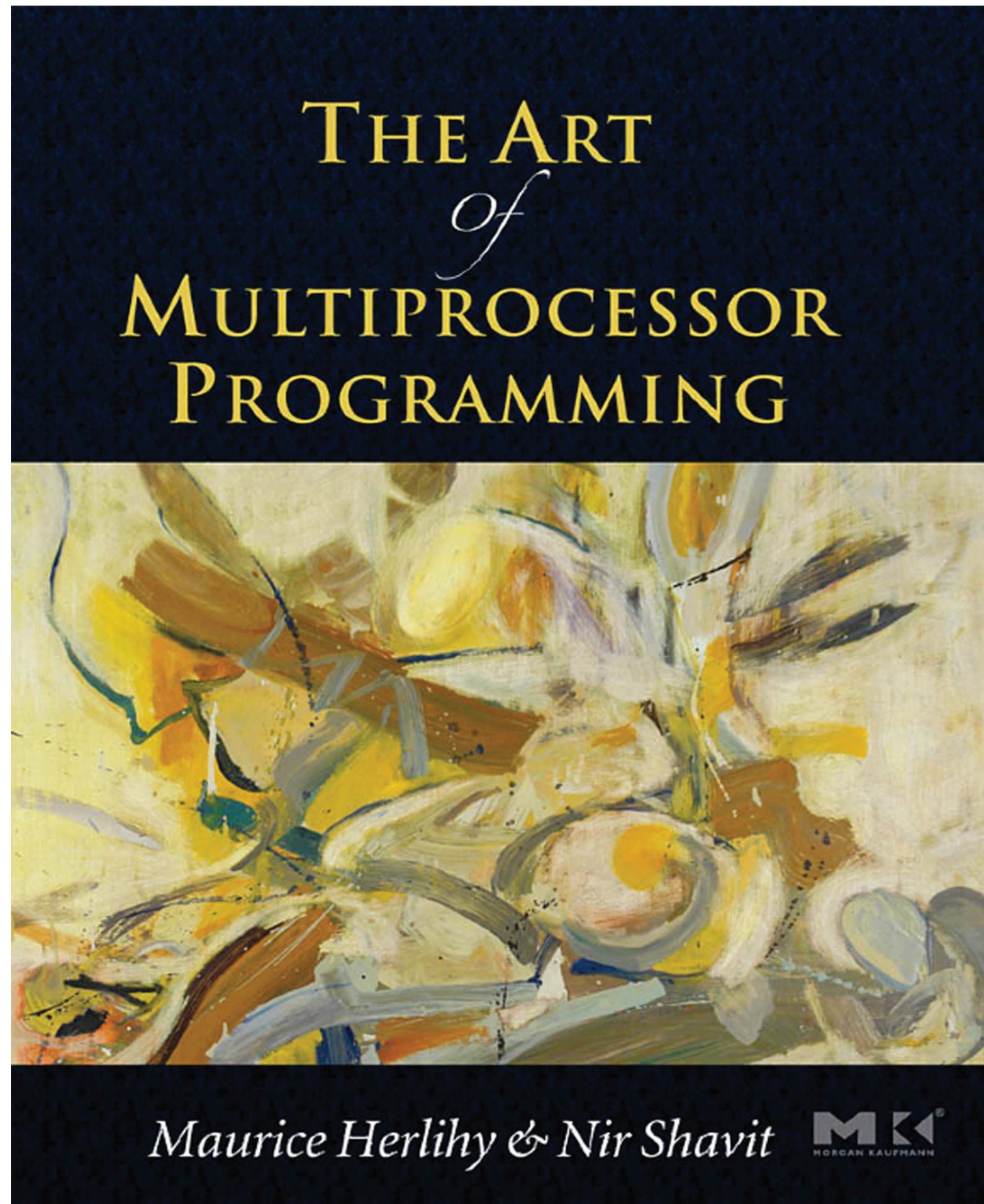
What's in this course.

# Most of this course: Multicore Programming

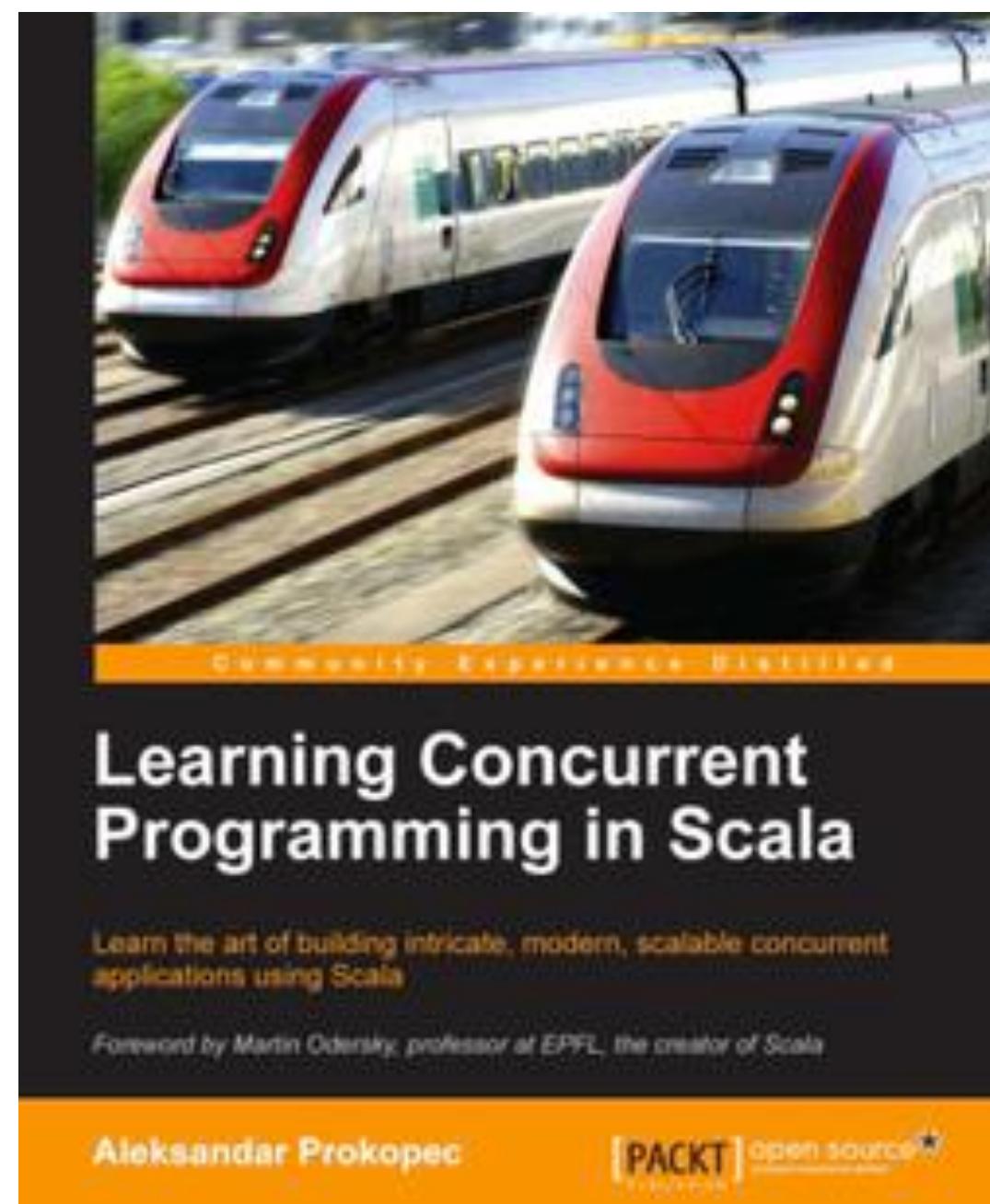
- Fundamentals
  - Models, algorithms, impossibility
- Real-World programming
  - Architectures
  - Techniques

# Resources

About 50% of the material



about 30%

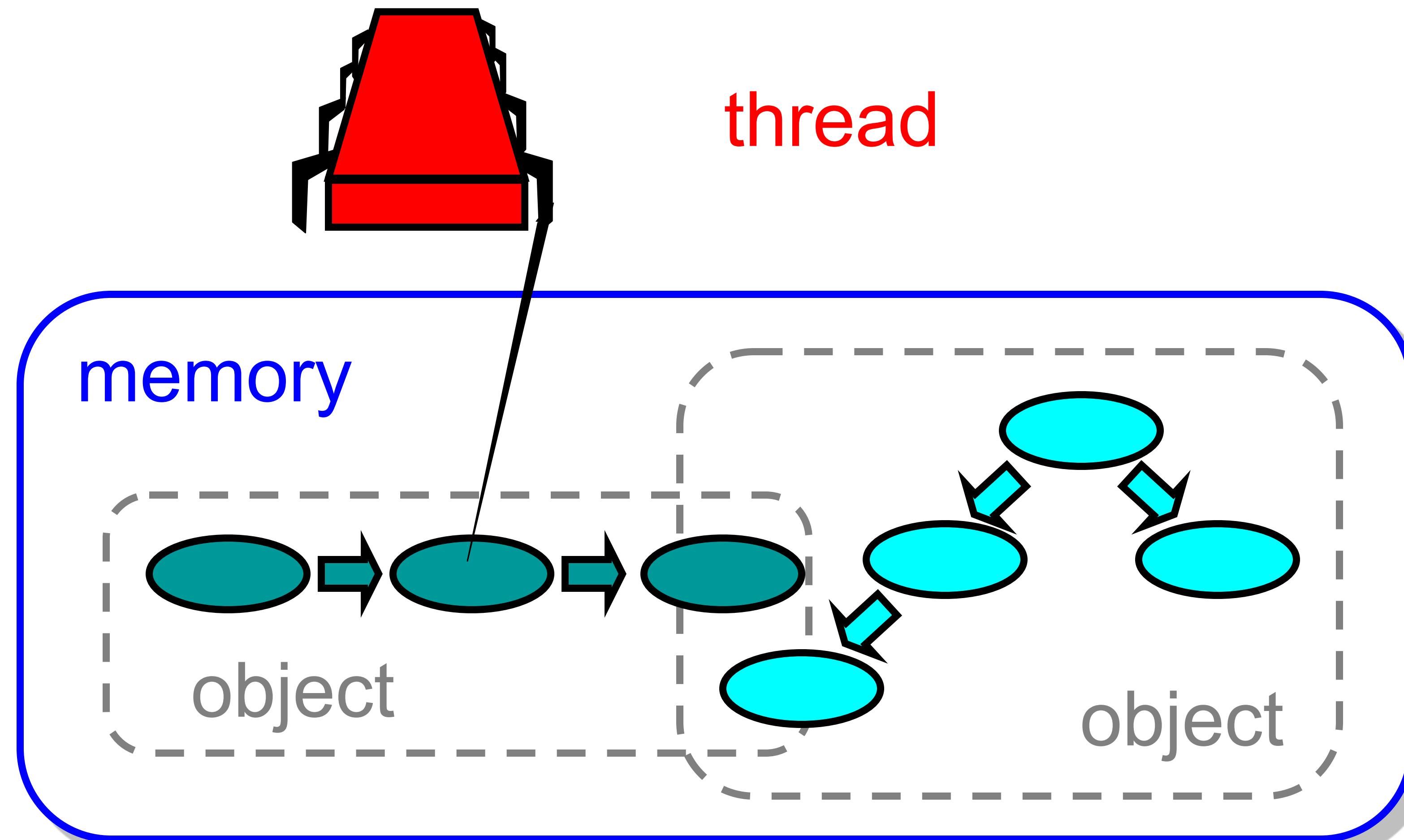


The rest

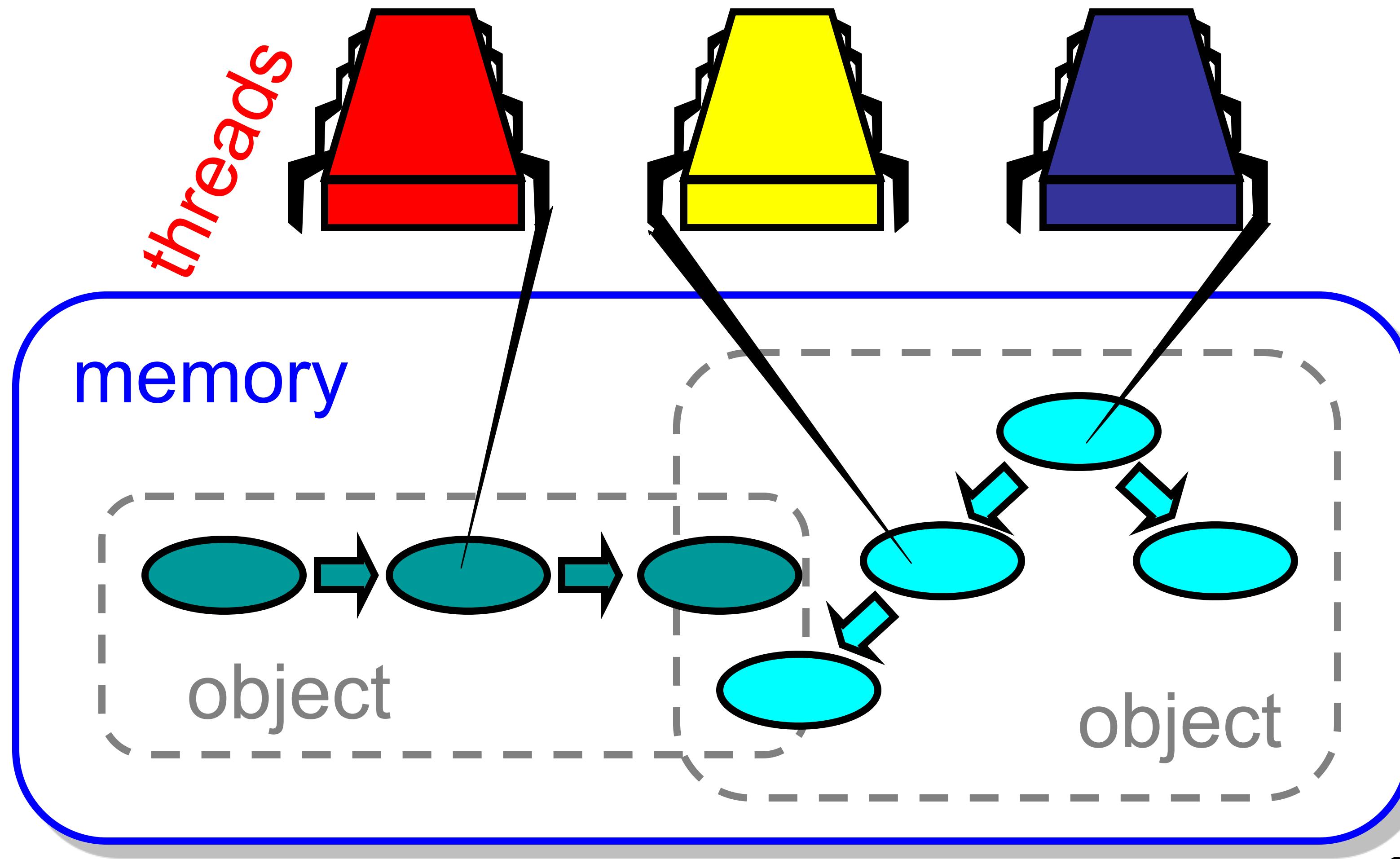
- Lecture slides
- Lecture notes
- The Code

# Thinking concurrently

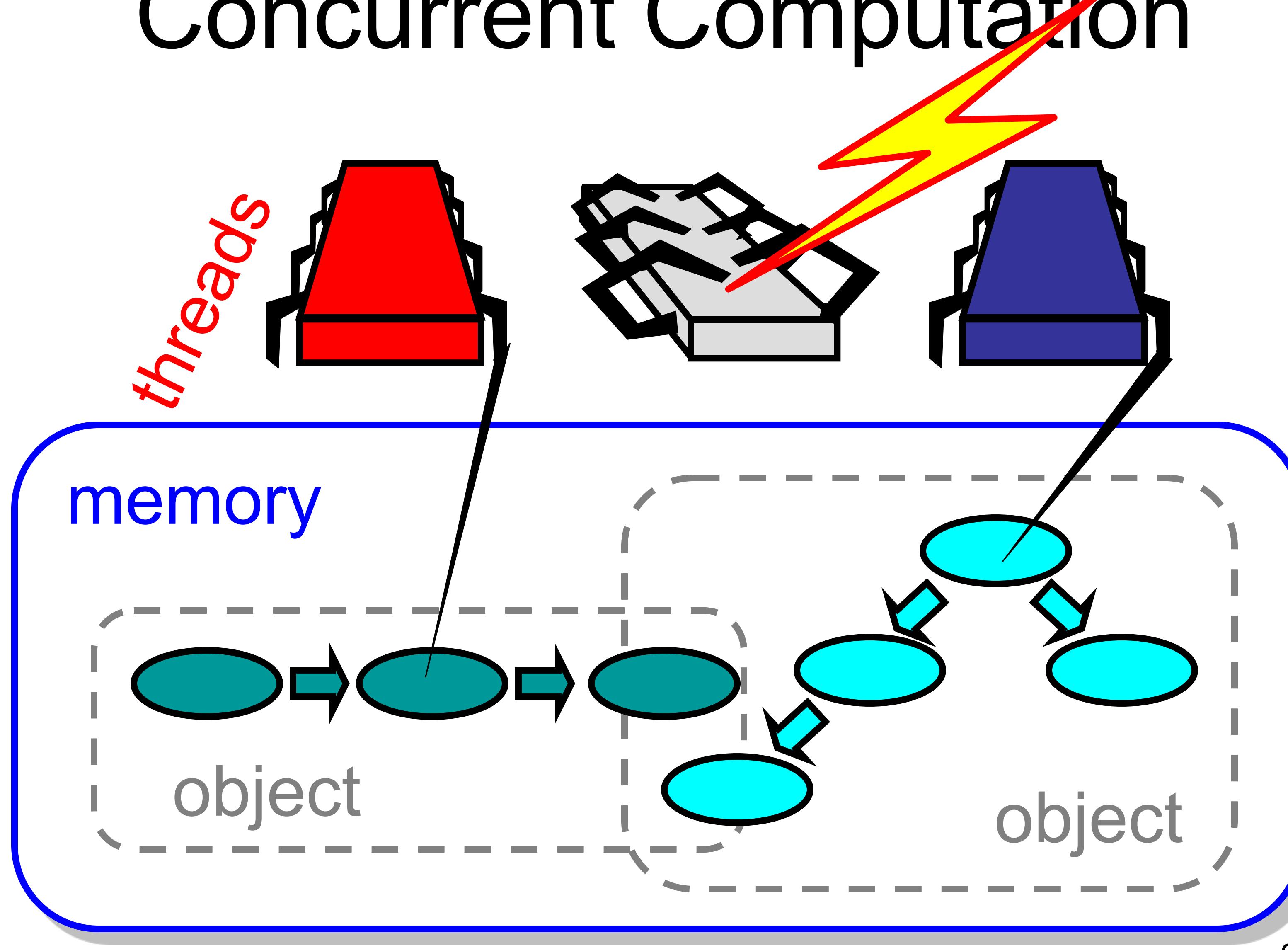
# Sequential Computation



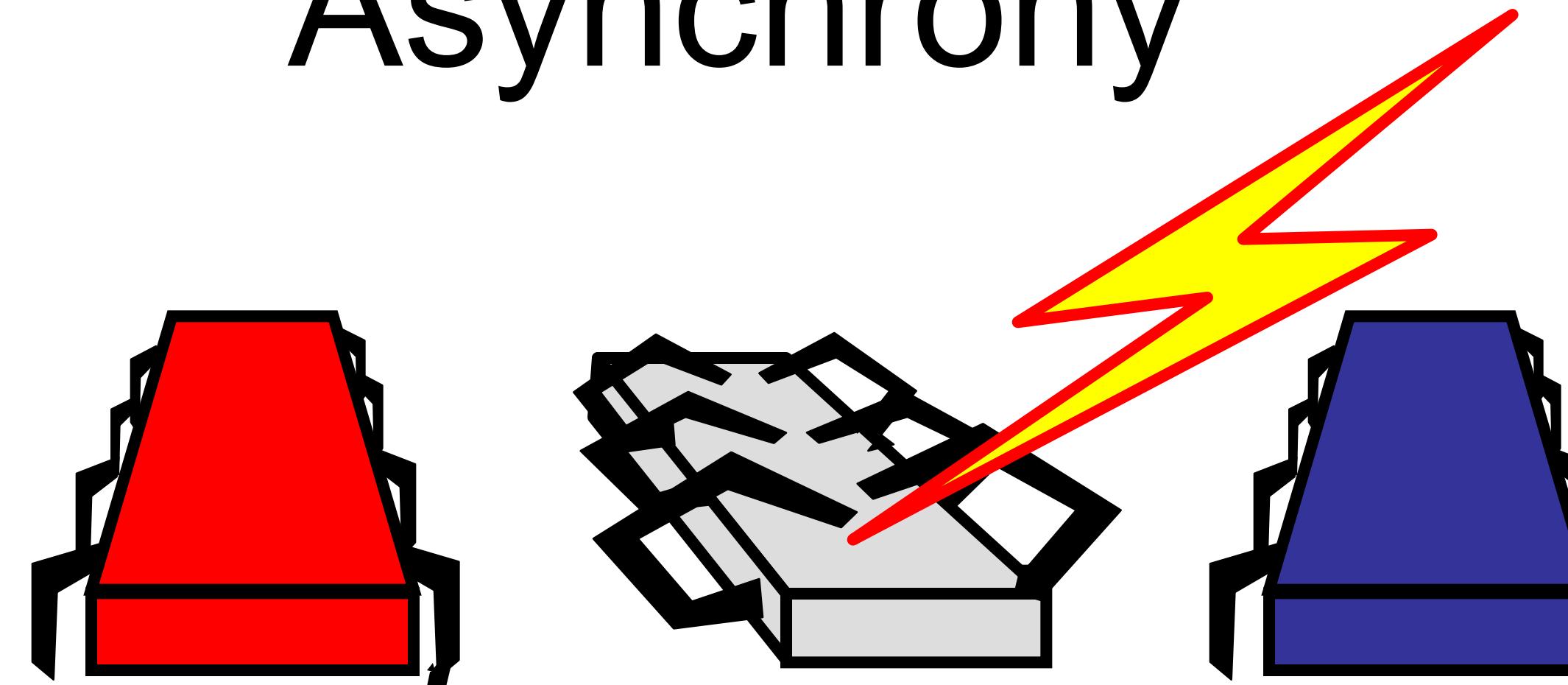
# Concurrent Computation



# Concurrent Computation

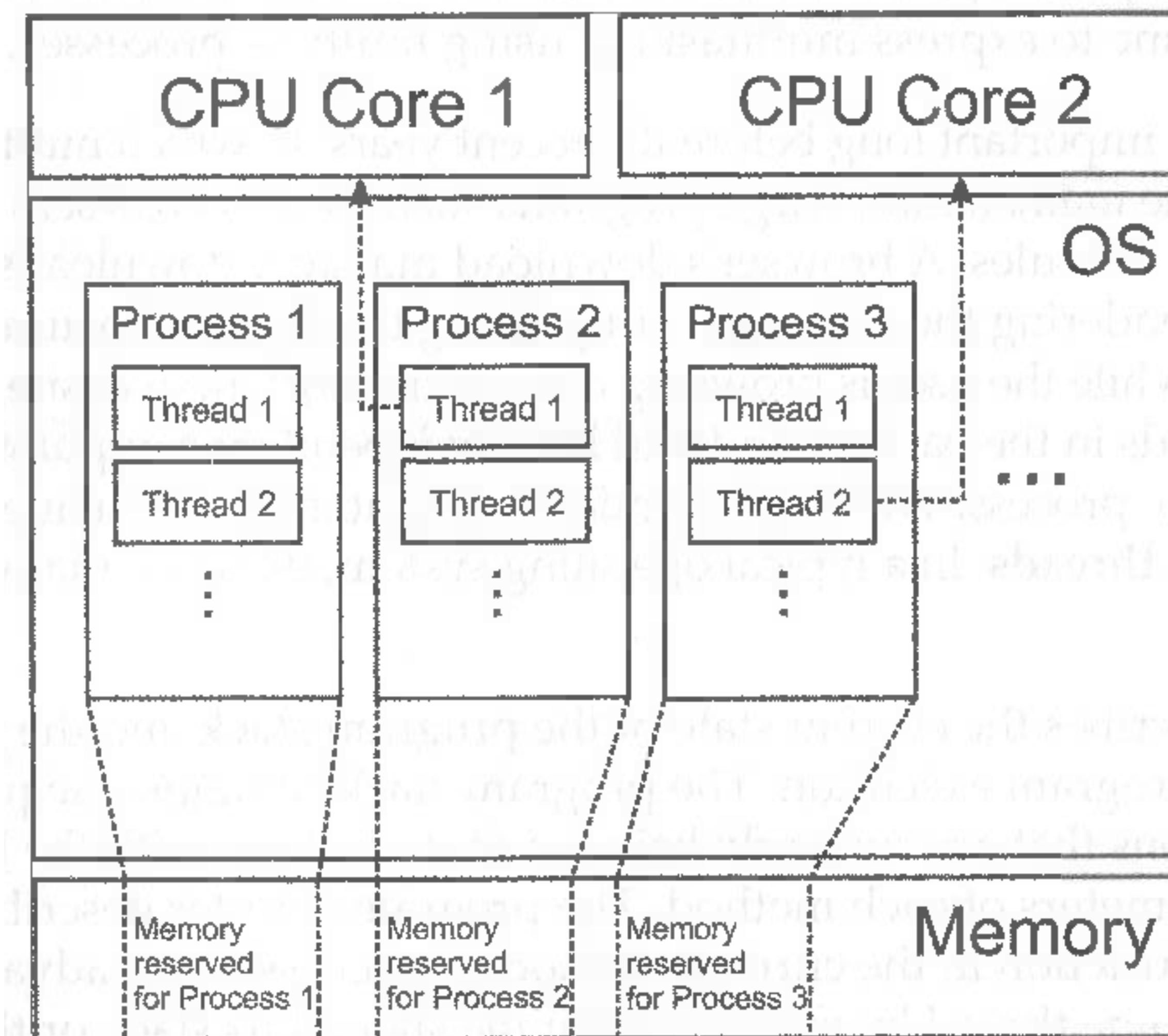


# Asynchrony



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

# Threads, Processes and Processors



# Model Summary

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

# Road Map

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

# Concurrency Jargon

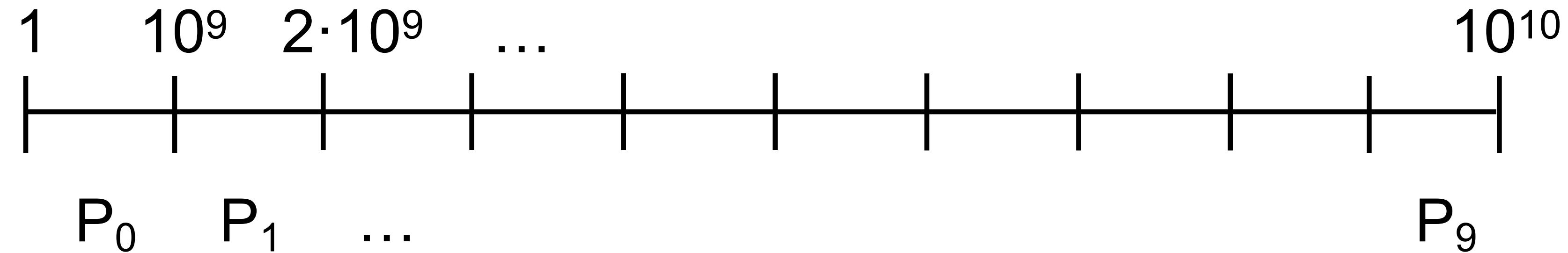
- Hardware
  - Processors
- Software
  - Threads, processes  
(one process may have several threads)
- Sometimes OK to confuse them, sometimes not.

# Designing Concurrent Programs

# 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 $i$

```
def primePrint(): Unit = {
    val i = ThreadID.get // Thread IDs in 0..9
    val block = math.pow(10, 109)
    for (j <- (i * block) + 1 to (i + 1) * block) {
        if (isPrime(j)) {
            println(j)
        }
    }
}
```

# Issues (?)

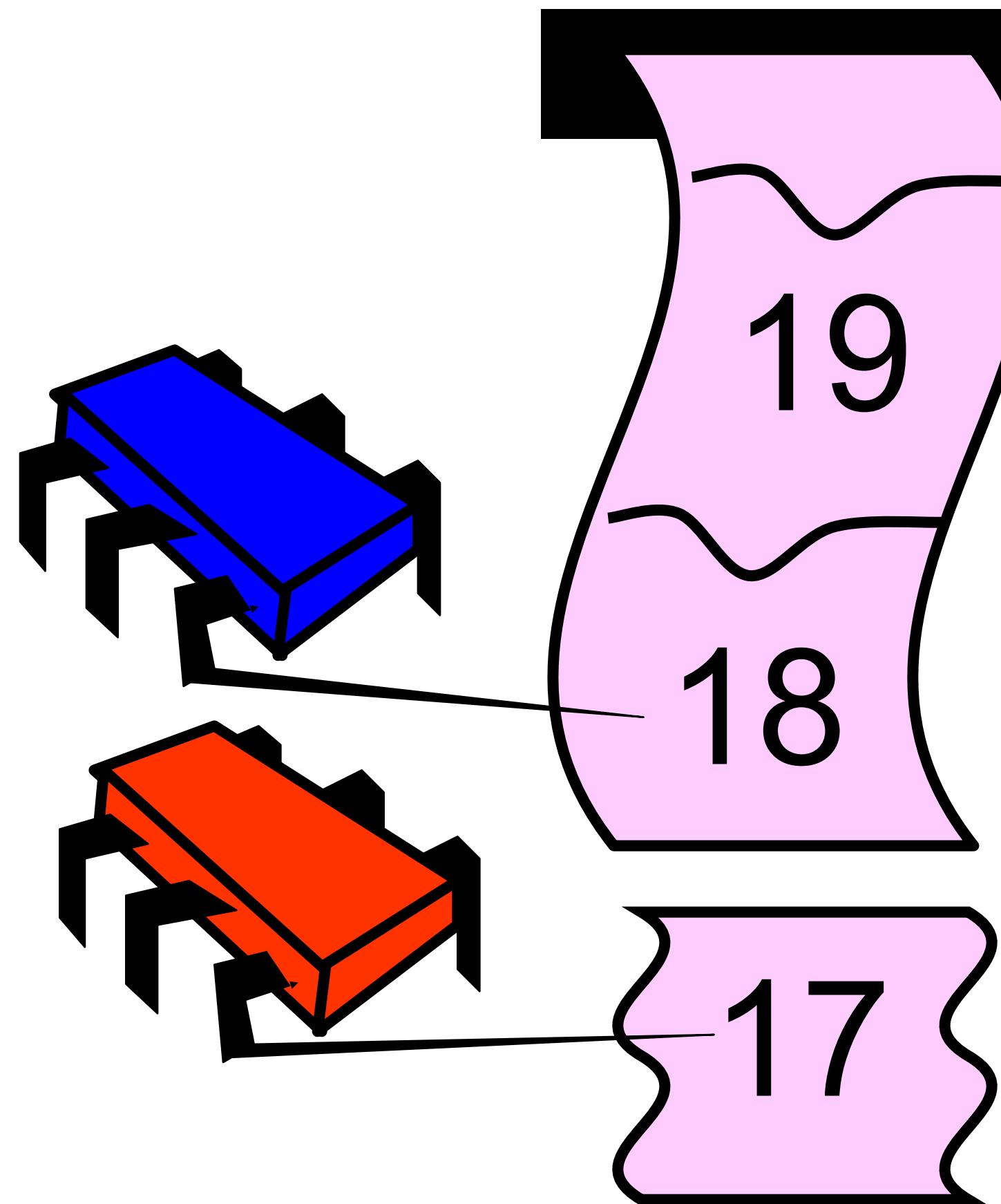
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict

# Issues

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

rejected

# Shared Counter



each thread  
takes a number

# Procedure for Thread $i$

```
val counter = new Counter

def primePrint(): Unit = {
    var i: Int = 1
    val limit = math.pow(10, 9).intValue
    while (i < limit) {
        i = counter.getAndIncrement
        if (isPrime(i)) {
            println(i)
        }
    }
}
```

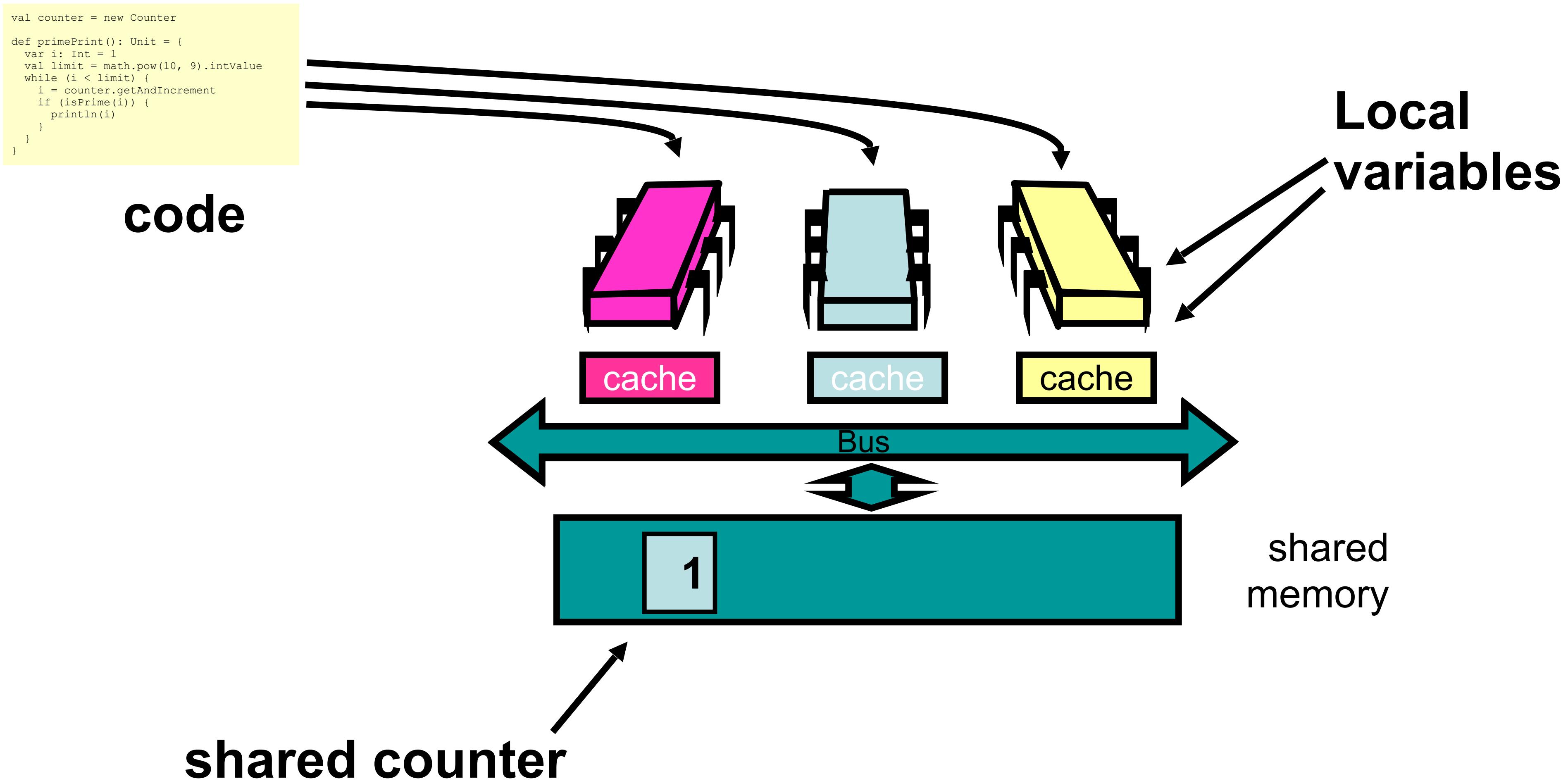
# Procedure for Thread *i*

```
val counter = new Counter

def primePrint(): Unit = {
    var i: Int = 1
    val limit = math.pow(10, 9).intValue
    while (i < limit) {
        i = counter.getAndIncrement
        if (isPrime(i)) {
            println(i)
        }
    }
}
```

Shared counter  
object

# Where Things Reside



# Procedure for Thread *i*

```
val counter = new Counter

def primePrint(): Unit = {
    var i: Int = 1
    val limit = math.pow(10, 9).intValue
    while (i < limit) {
        i = counter.getAndIncrement
        if (isPrime(i)) {
            println(i)
        }
    }
}
```

**Stop when every value taken**

# Procedure for Thread *i*

```
val counter = new Counter

def primePrint(): Unit = {
    var i: Int = 1
    val limit = math.pow(10, 9).intValue
    while (i < limit) {
        i = counter.getAndIncrement
        if (isPrime(i)) {
            println(i)
        }
    }
}
```

Increment & return each  
new value

# Demo

# Counter Implementation

```
class Counter {  
    private var count = 0  
  
    def getAndIncrement: Int = {  
        val tmp = count  
        count = tmp + 1  
        tmp  
    }  
}
```

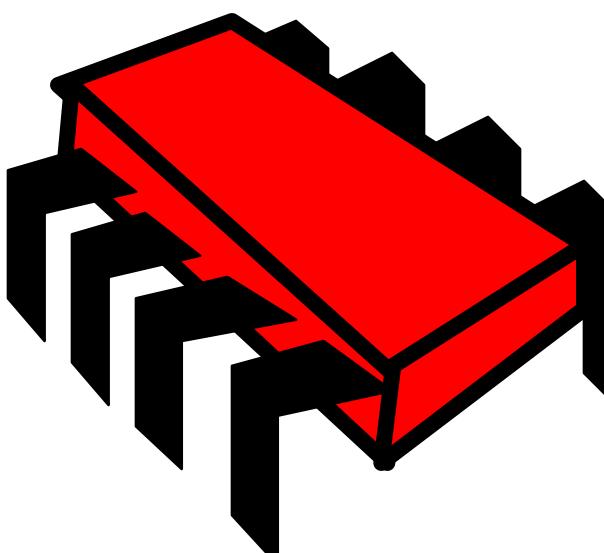
# Counter Implementation

```
class Counter {  
    private var count = 0  
  
    def getAndIncrement: Int = {  
        val tmp = count  
        count = tmp + 1  
        tmp  
    }  
}
```

OK for single thread,  
not for concurrent threads

# Not so good...

**Value... 1**



read write read write  
1 2 2 3

# read

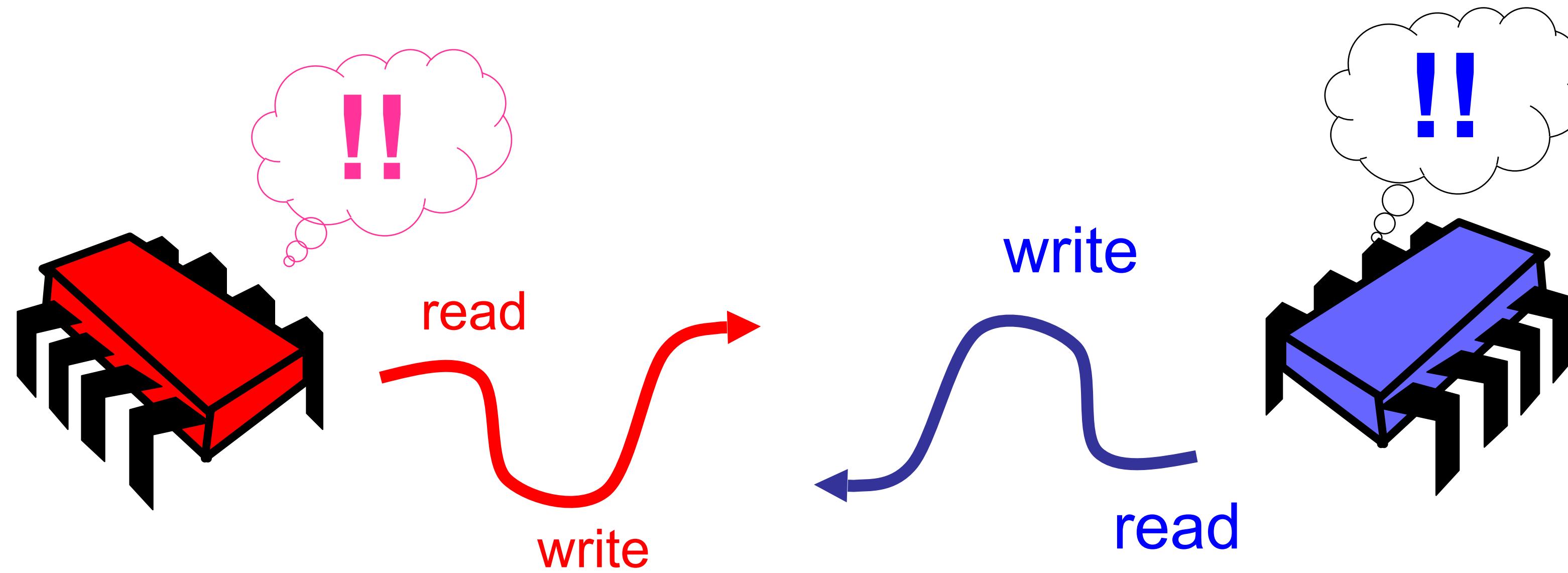
## 1

# write

# 2

# time

# Is this problem inherent?



If we could only glue reads and writes together...

# Challenge

```
class Counter {  
    private var count = 0  
  
    def getAndIncrement: Int = {  
        val tmp = count  
        count = tmp + 1  
        tmp  
    }  
}
```

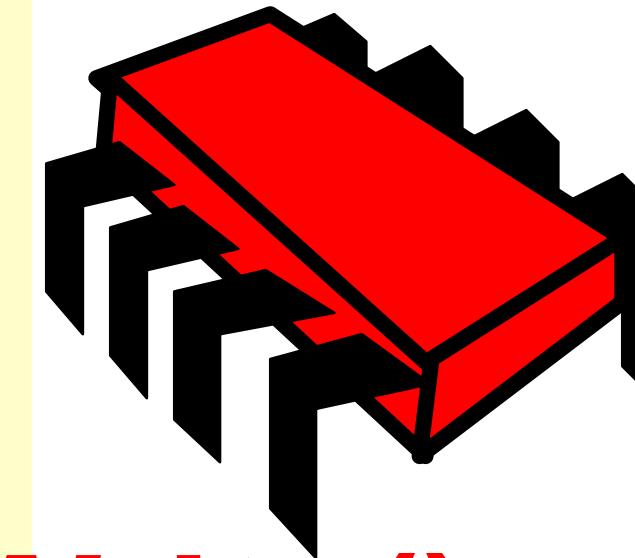
# Challenge

```
class Counter {  
    private var count = 0  
  
    def getAndIncrement: Int = {  
        val tmp = count  
        count = tmp + 1  
        tmp  
    }  
}
```

Make these steps  
*atomic (indivisible)*

# Hardware Solution

```
class Counter {  
    private var count = 0  
  
    def getAndIncrement: Int = {  
        val tmp = count  
        count = tmp + 1  
        tmp  
    }  
}
```



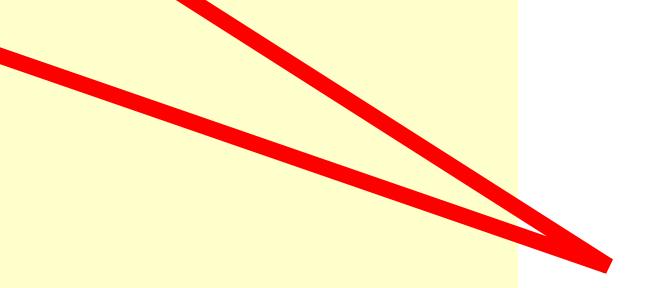
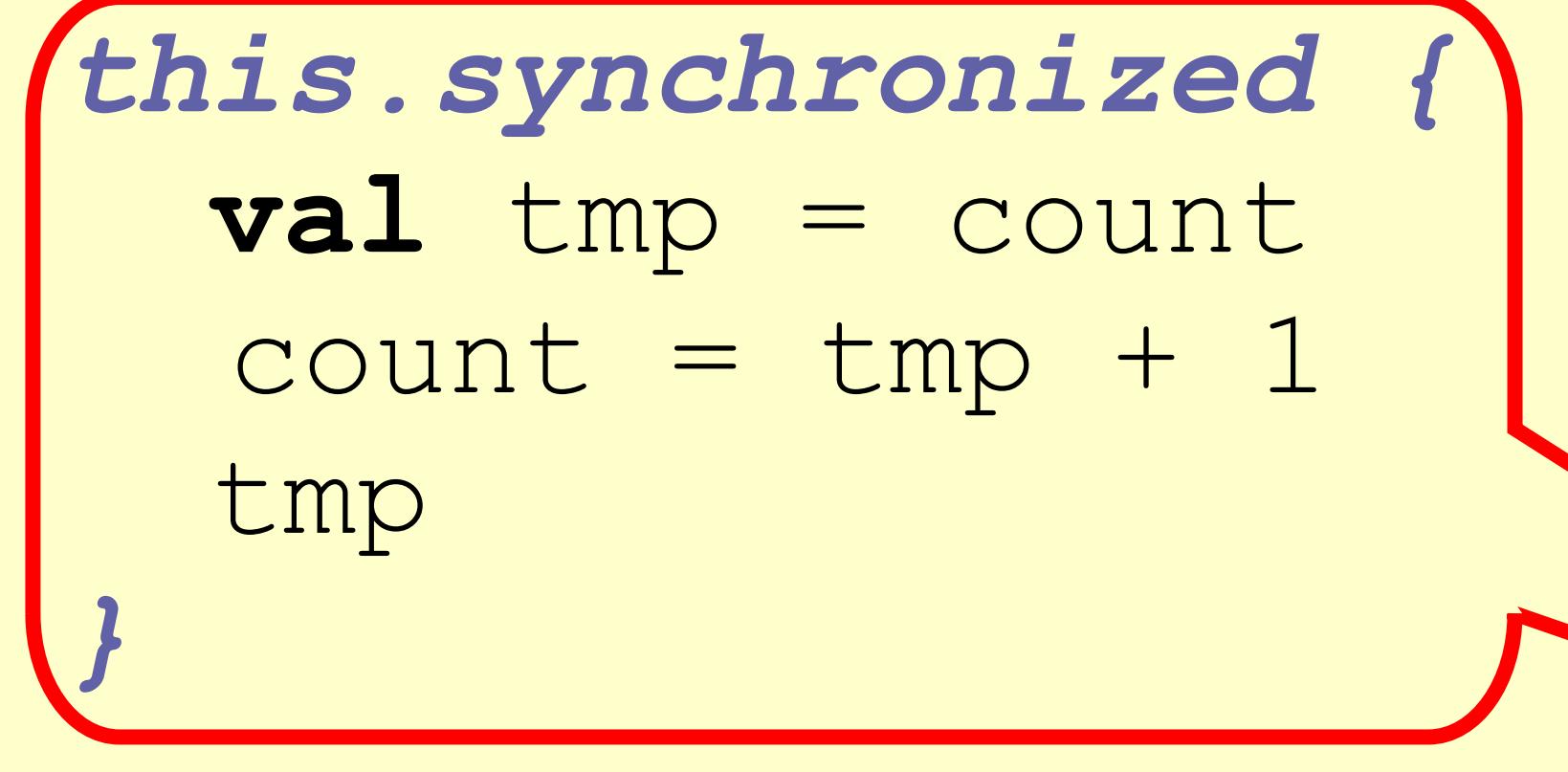
**ReadModifyWrite()  
instruction**

# Java / Scala solution

```
class Counter {  
    private var count = 0  
  
    def getAndIncrement: Int = {  
        this.synchronized {  
            val tmp = count  
            count = tmp + 1  
            tmp  
        }  
    }  
}
```

# Java / Scala solution

```
class Counter {  
    private var count = 0  
  
    def getAndIncrement: Int = {  
        this.synchronized {  
            val tmp = count  
            count = tmp + 1  
            tmp  
        }  
    }  
}
```



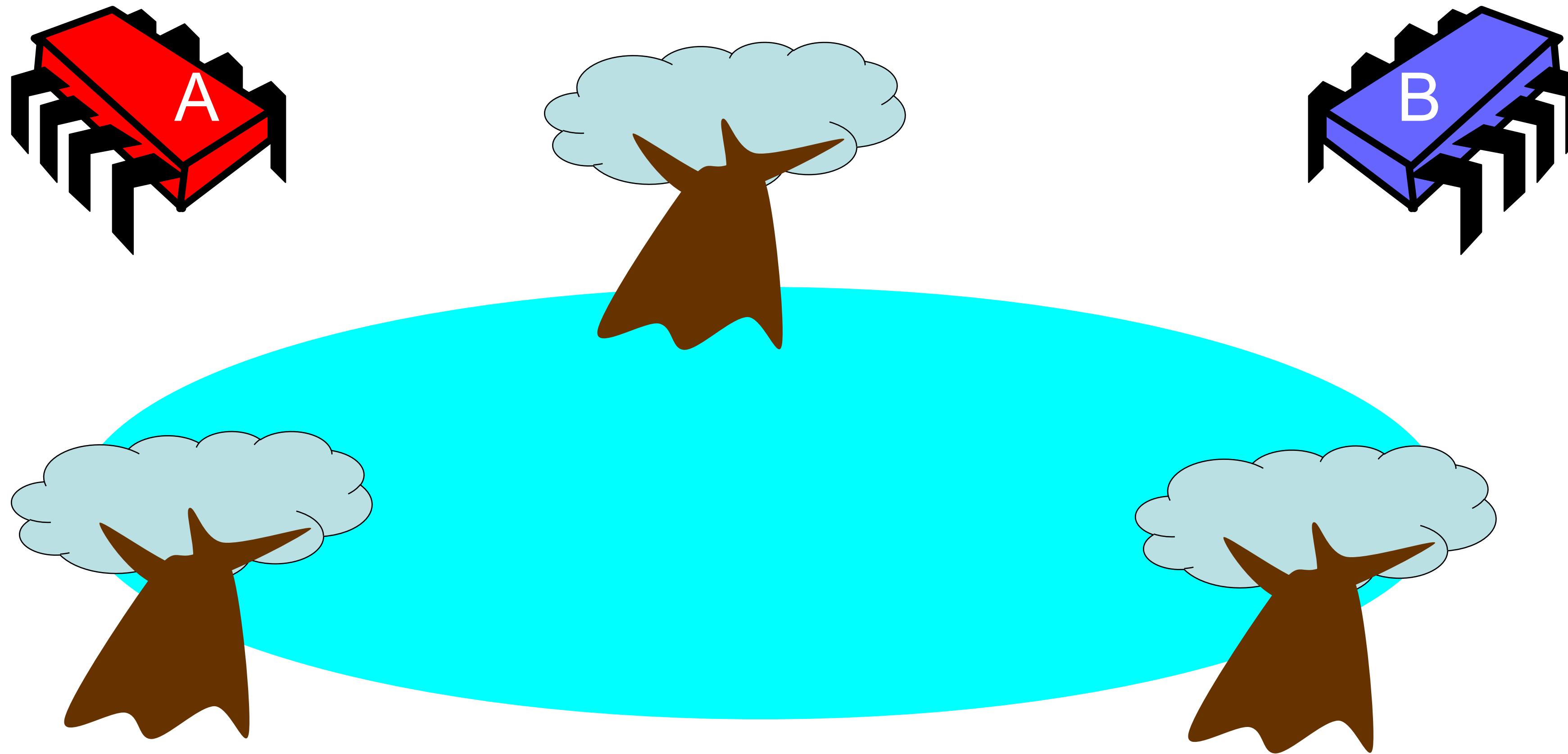
Synchronized block

# Java / Scala solution

```
class Counter {  
    private var count = 0  
  
    def getAndIncrement: Int = {  
        this.synchronized {  
            val tmp = count  
            count = tmp + 1  
            tmp  
        }  
    }  
}
```

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
  - If is violated, this is done by a *finite* computation
- Liveness Properties
  - Something good happens eventually
  - Cannot be violated by a finite computation  
(intuition we can always run longer and see what happens)

# 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
- Problems?
- Gotcha
  - Not atomic
  - Trees obscure the view

# Interpretation

- Threads can't "see" what other threads are doing
- Explicit communication required for coordination

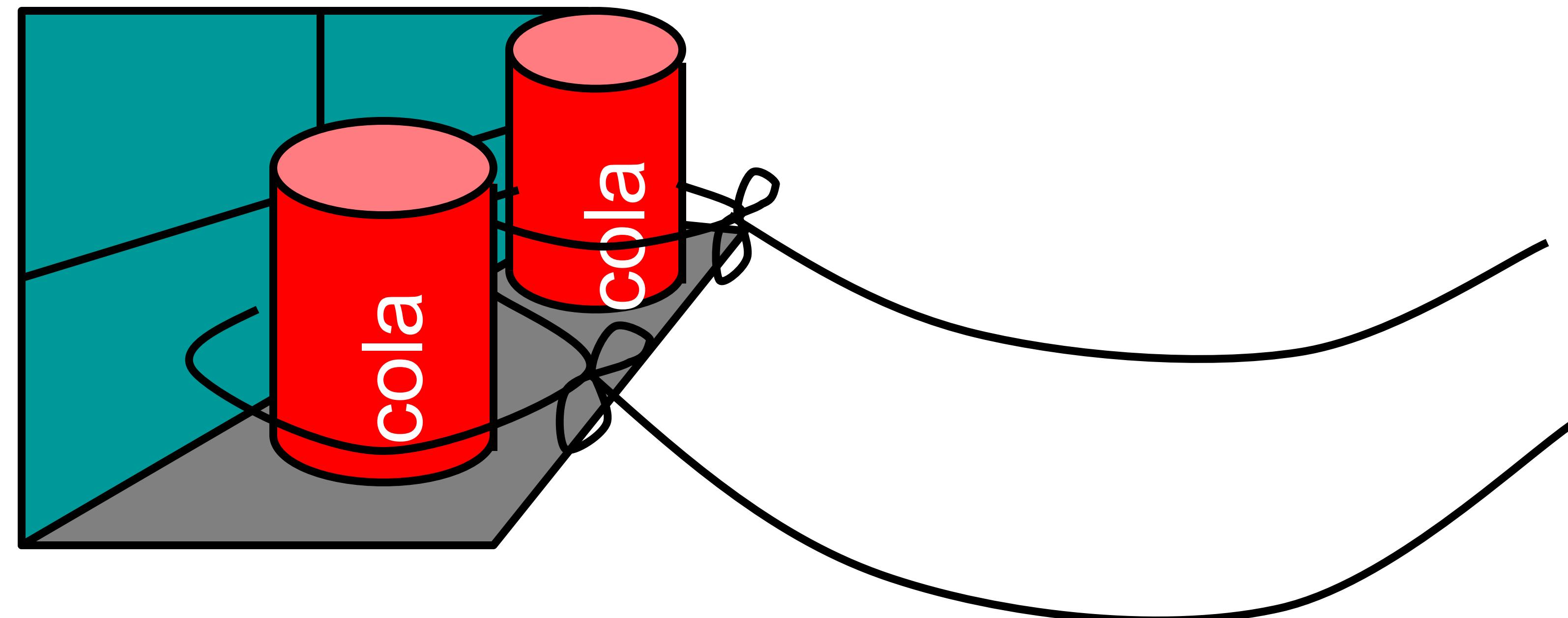
# Cell Phone Protocol

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

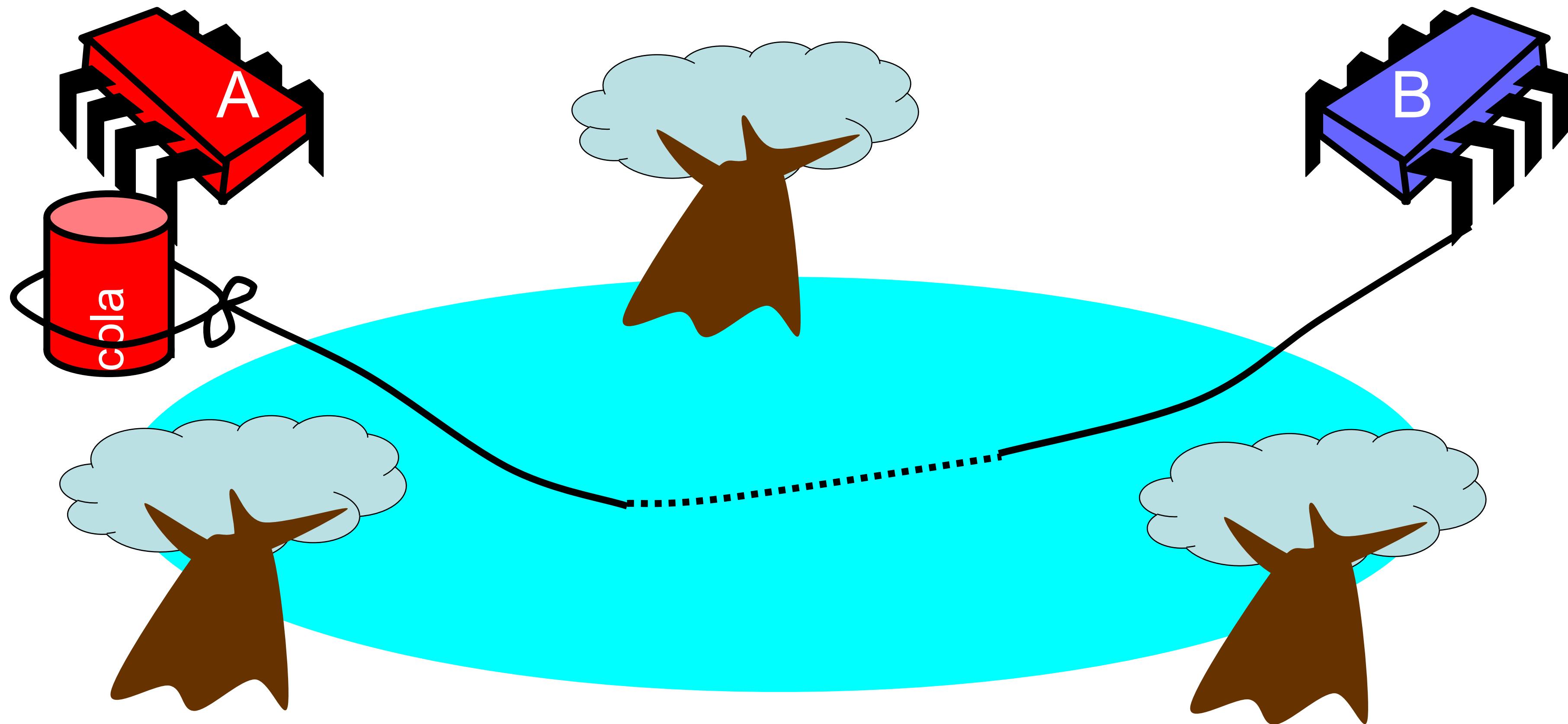
# Interpretation

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

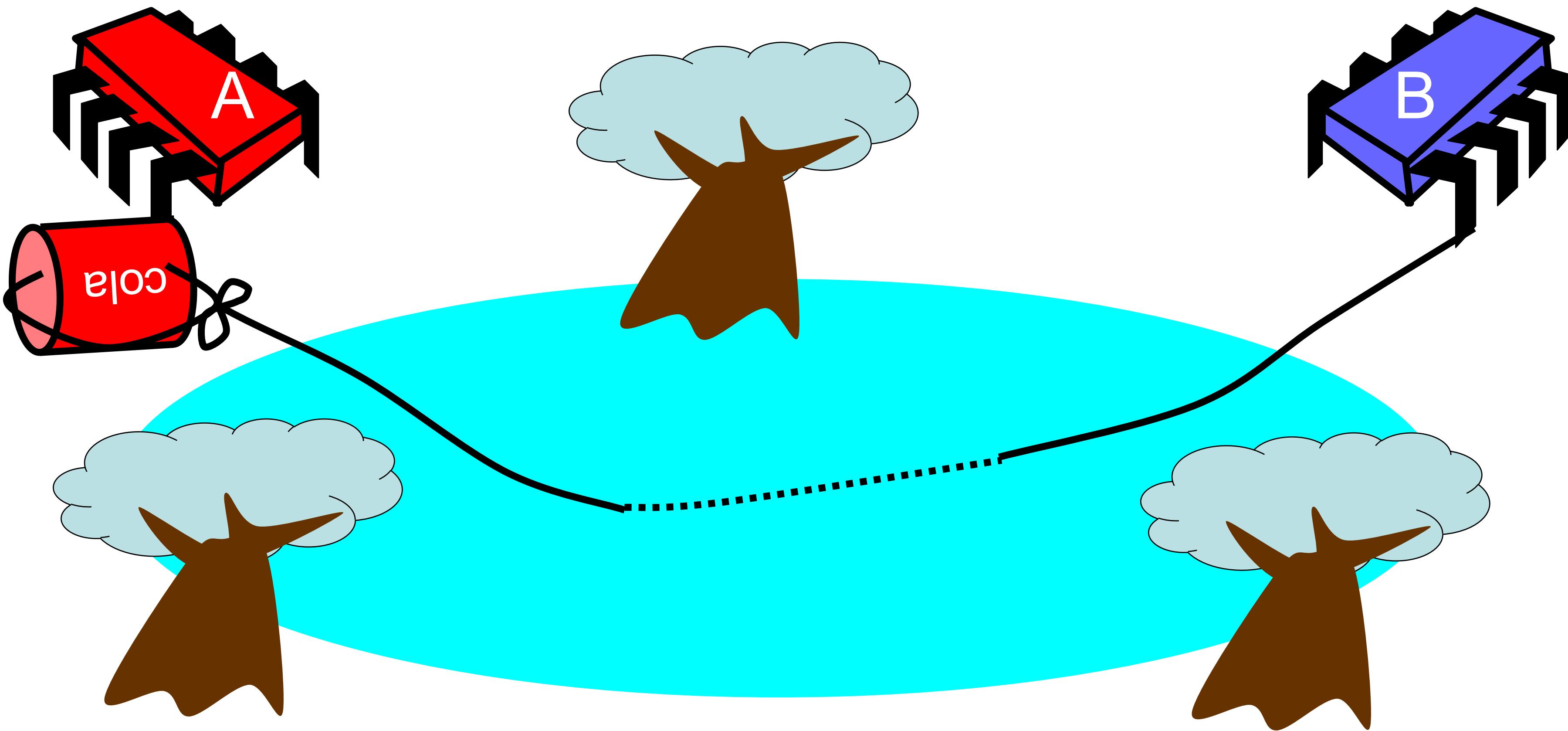
# Can Protocol



# Bob conveys a bit



# Bob conveys a bit



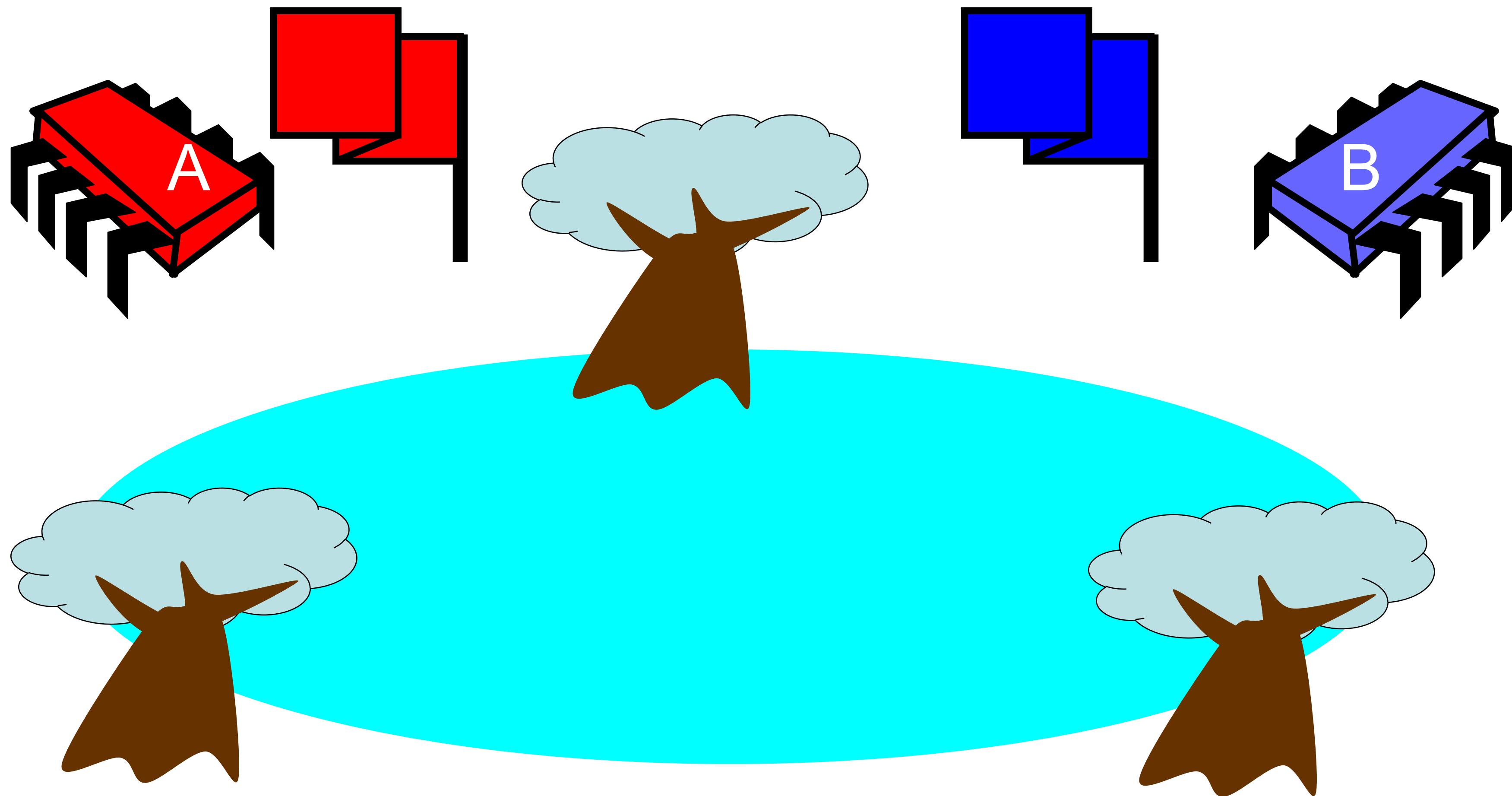
# Can Protocol

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

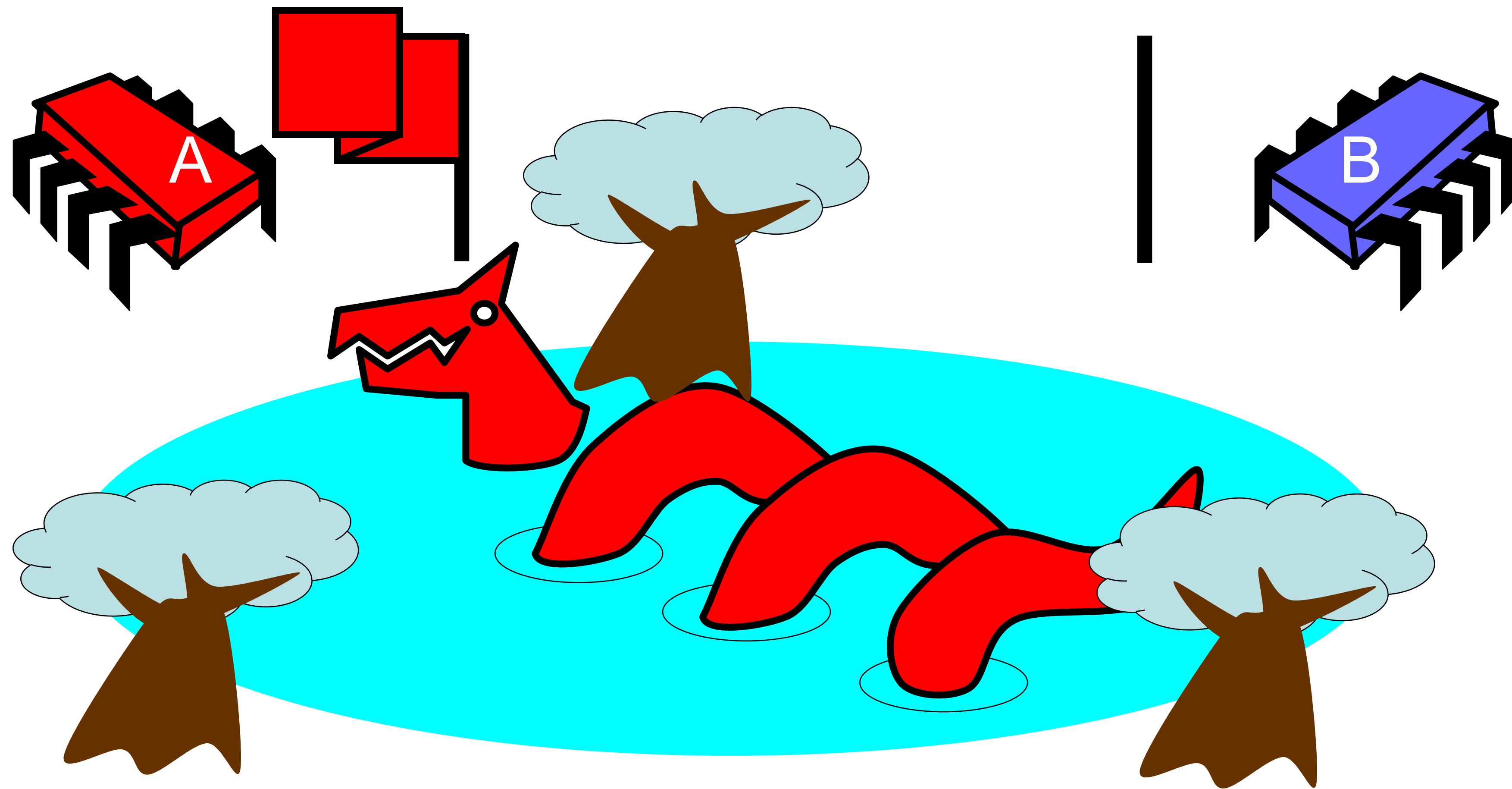
# Interpretation

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

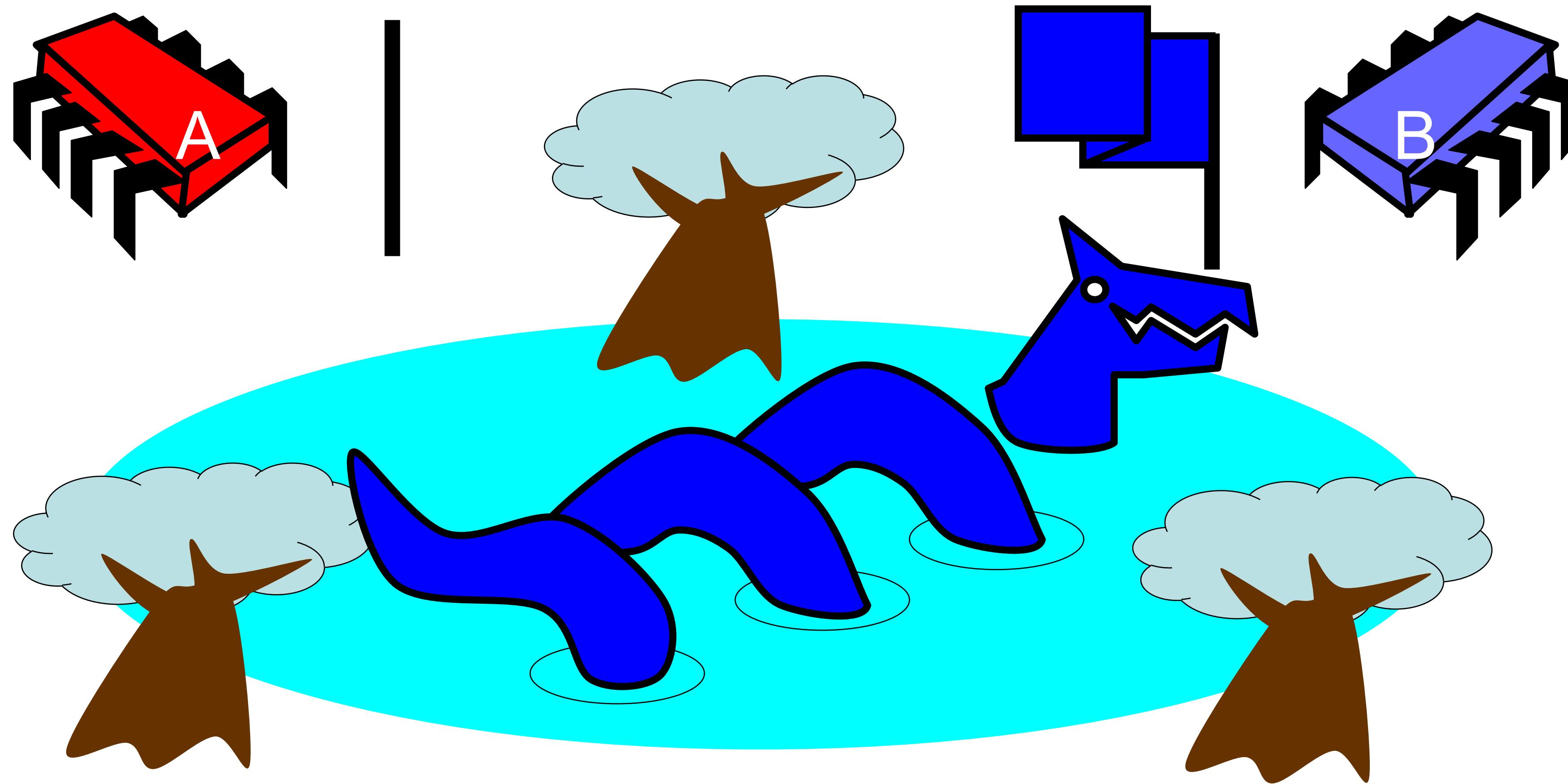
# Flag Protocol



# Alice's Protocol (sort of)



# Bob's Protocol (sort of)



# Alice's Protocol

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

# Bob's Protocol

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

Problems with this 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: deadlock!

# Bob's Protocol (2<sup>nd</sup> 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**

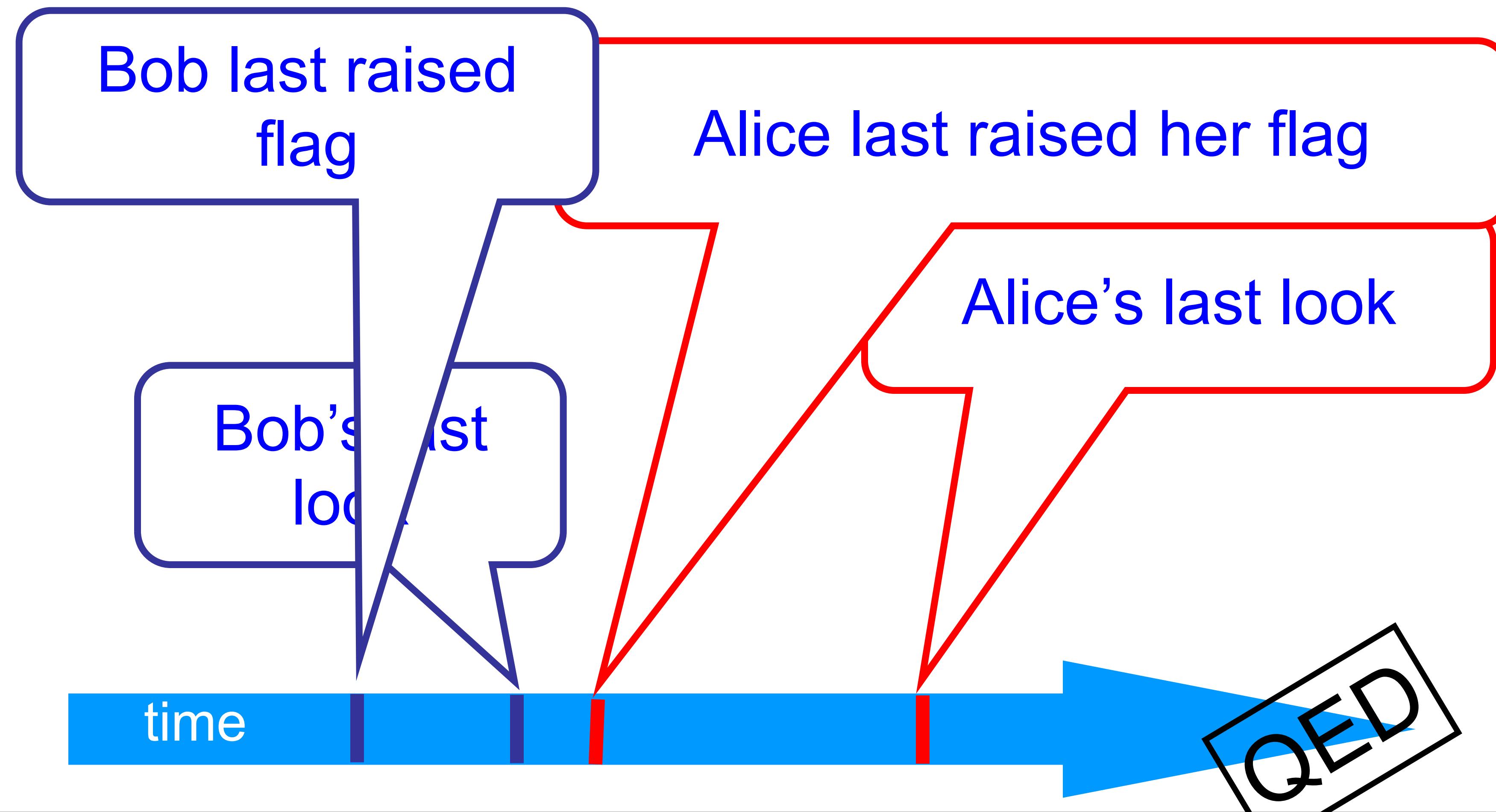
# The Flag Principle

- 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.

# Proof of No Deadlock

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

# Proof of No Deadlock

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



# Remarks

- Protocol is *unfair* (why?)
  - 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

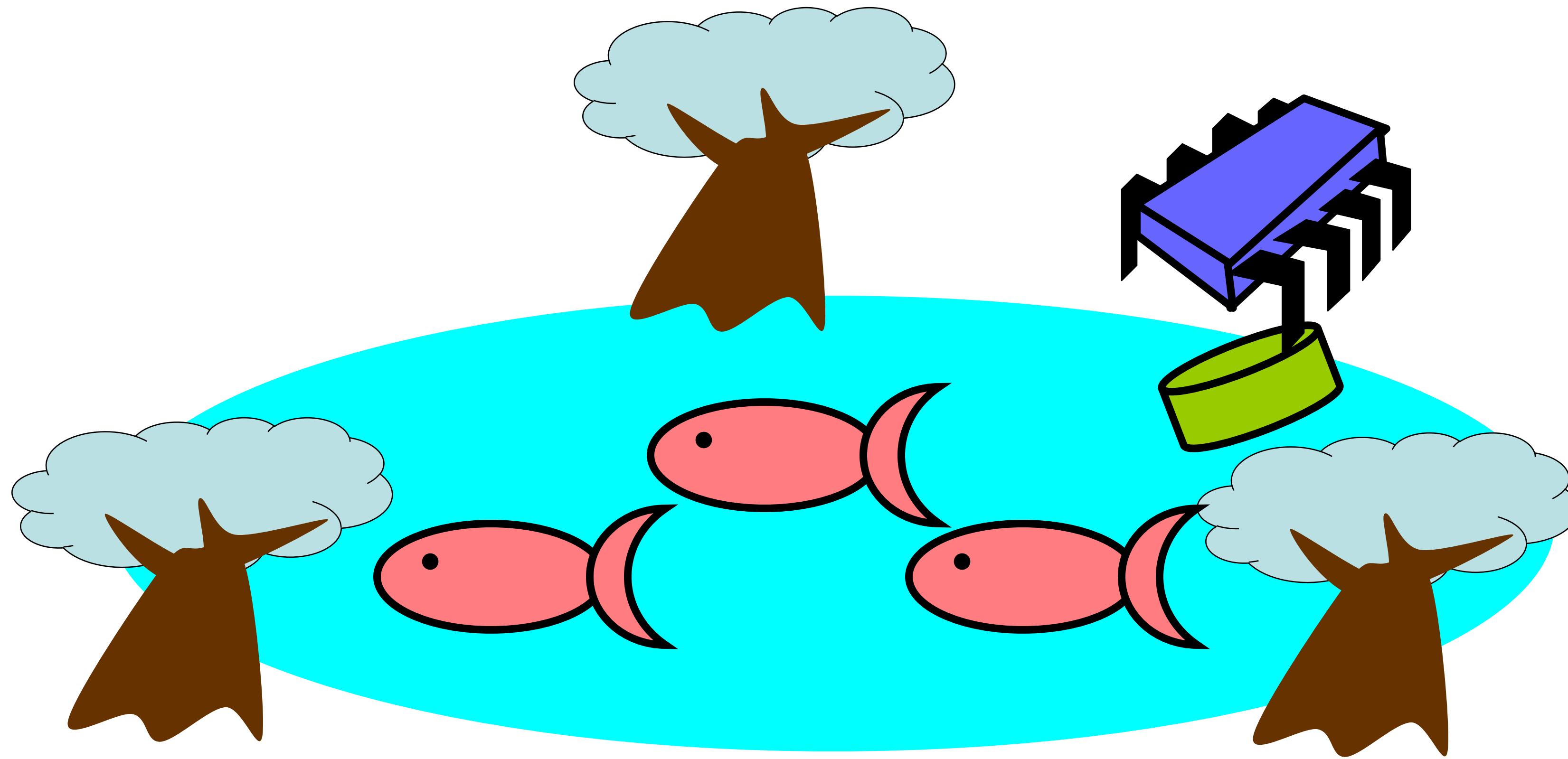
# The Fable Continues

- Alice and Bob fall in love & marry
- Then they fall out of love & divorce
  - After a coin flip, she gets the pets
  - He has to feed them

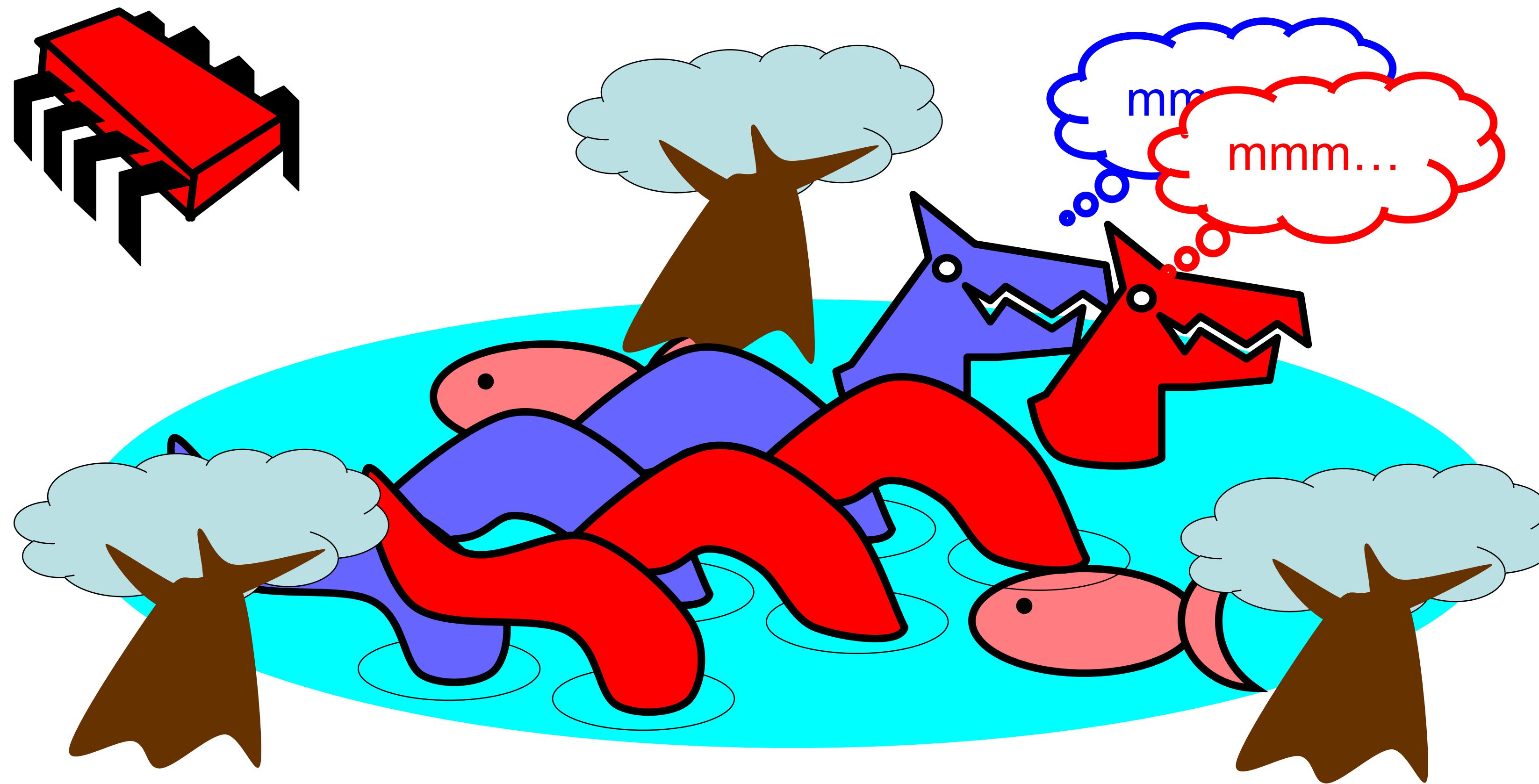
# The Fable Continues

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

# Bob Puts Food in the Pond



# Alice releases her pets to Feed



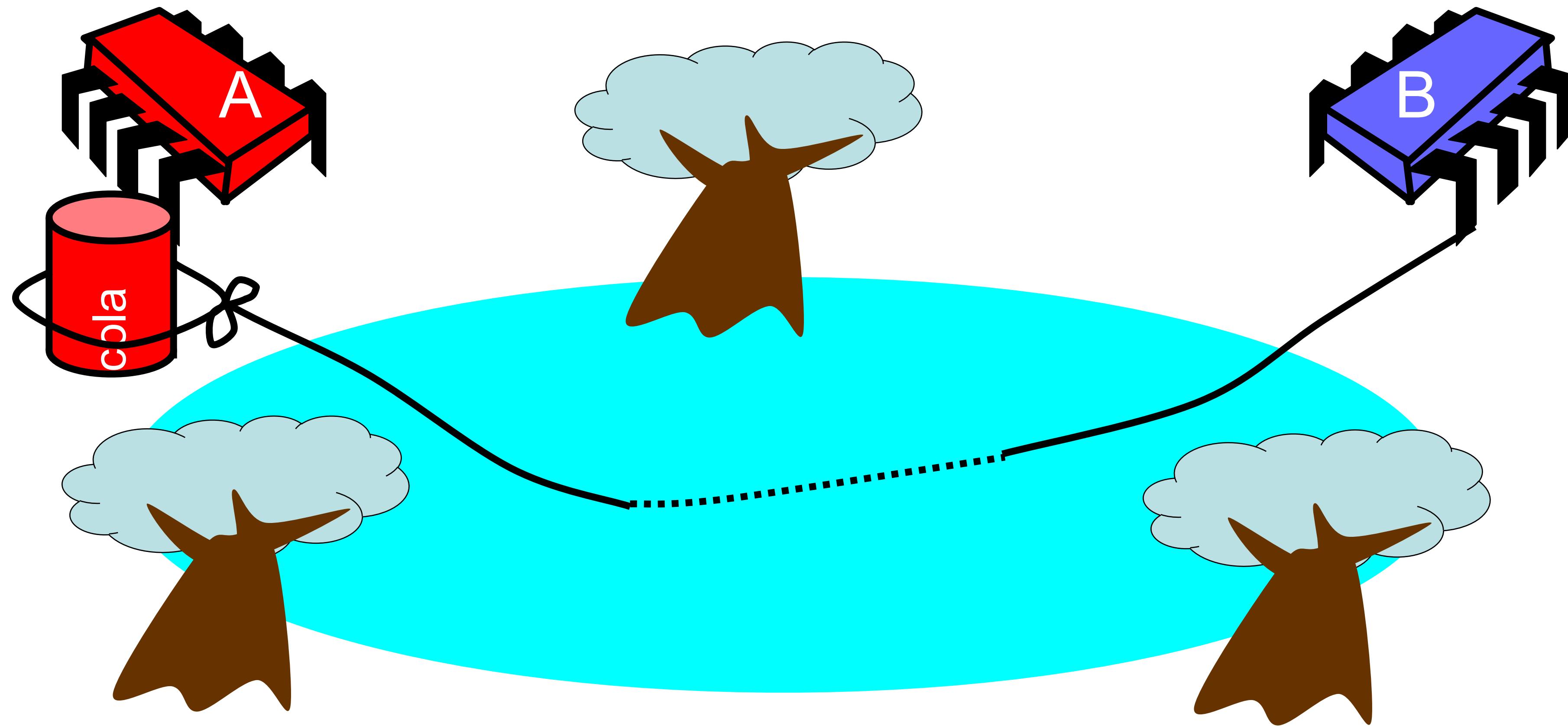
# Producer/Consumer

- Alice and Bob can't meet
  - Each has restraining order on other
  - So he puts food in the pond
  - And later, she releases the pets
- Avoid
  - Releasing pets when there's no food
  - Putting out food if uneaten food remains

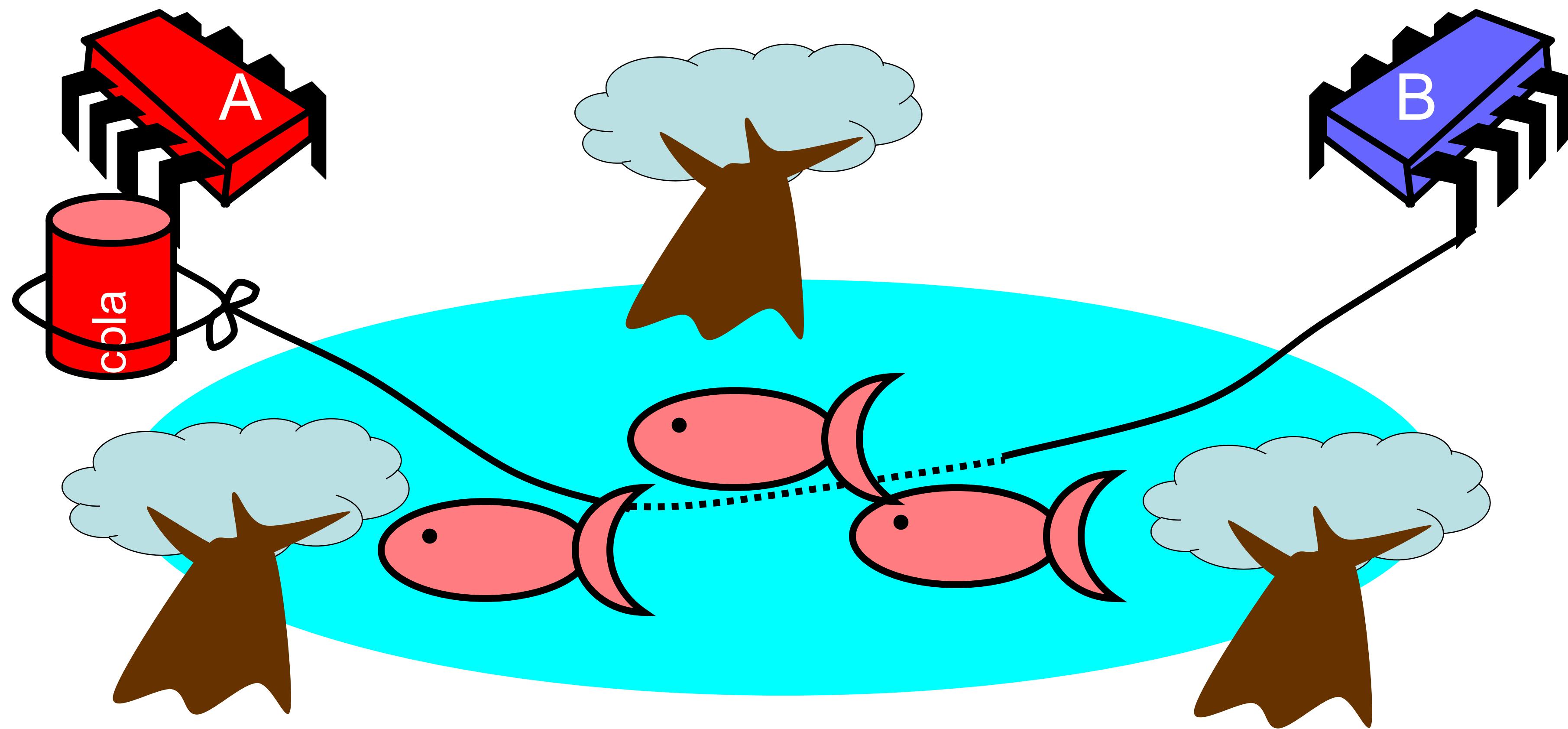
# Producer/Consumer

- Need a mechanism so that
  - Bob lets Alice know when food has been put out
  - Alice lets Bob know when to put out more food

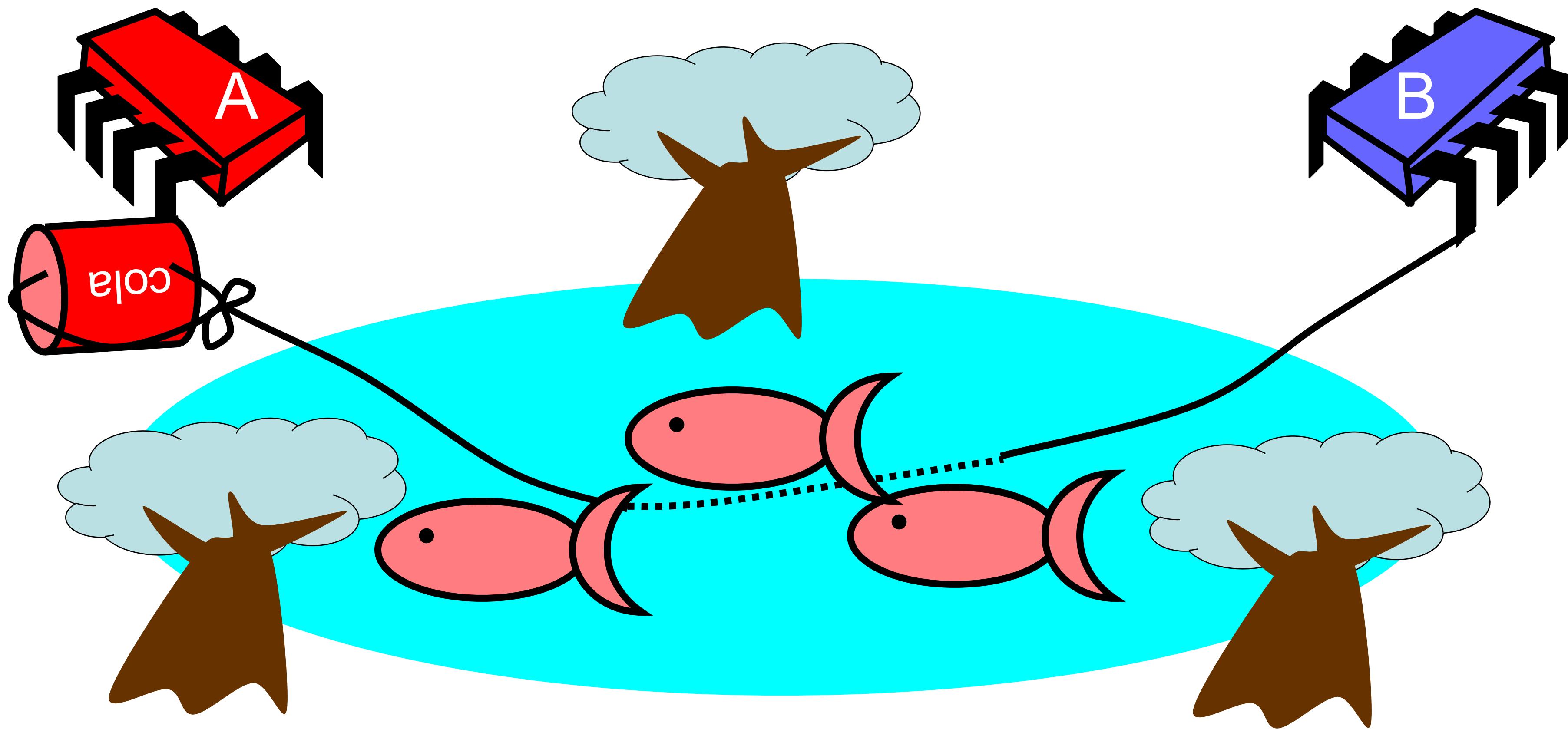
# Surprise Solution



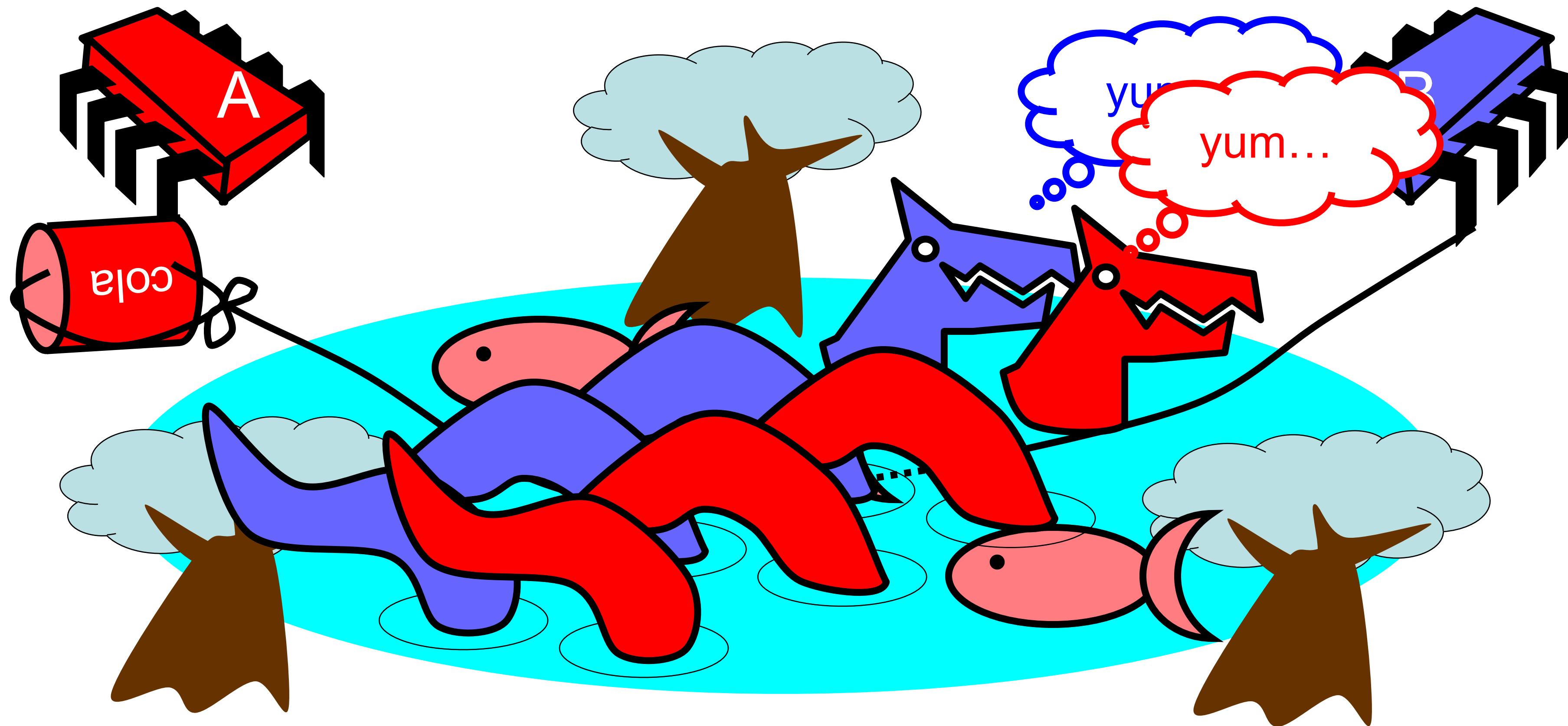
# Bob puts food in Pond



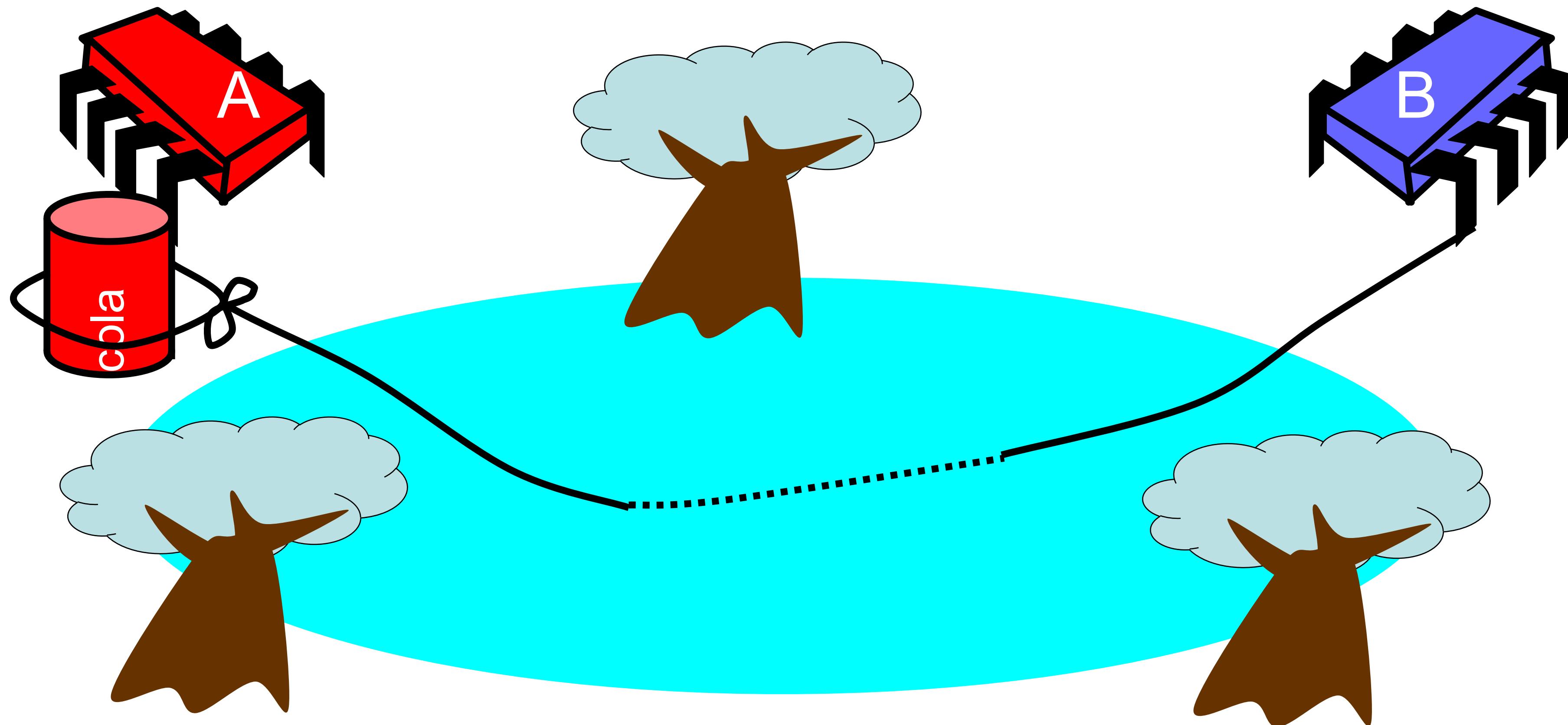
# Bob knocks over Can



# Alice Releases Pets



# Alice Resets Can when Pets are Fed



# Pseudocode

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

Alice's code

# Pseudocode

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

Alice's code

Bob's code

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

# Correctness

- Mutual Exclusion
  - Pets and Bob never together in pond

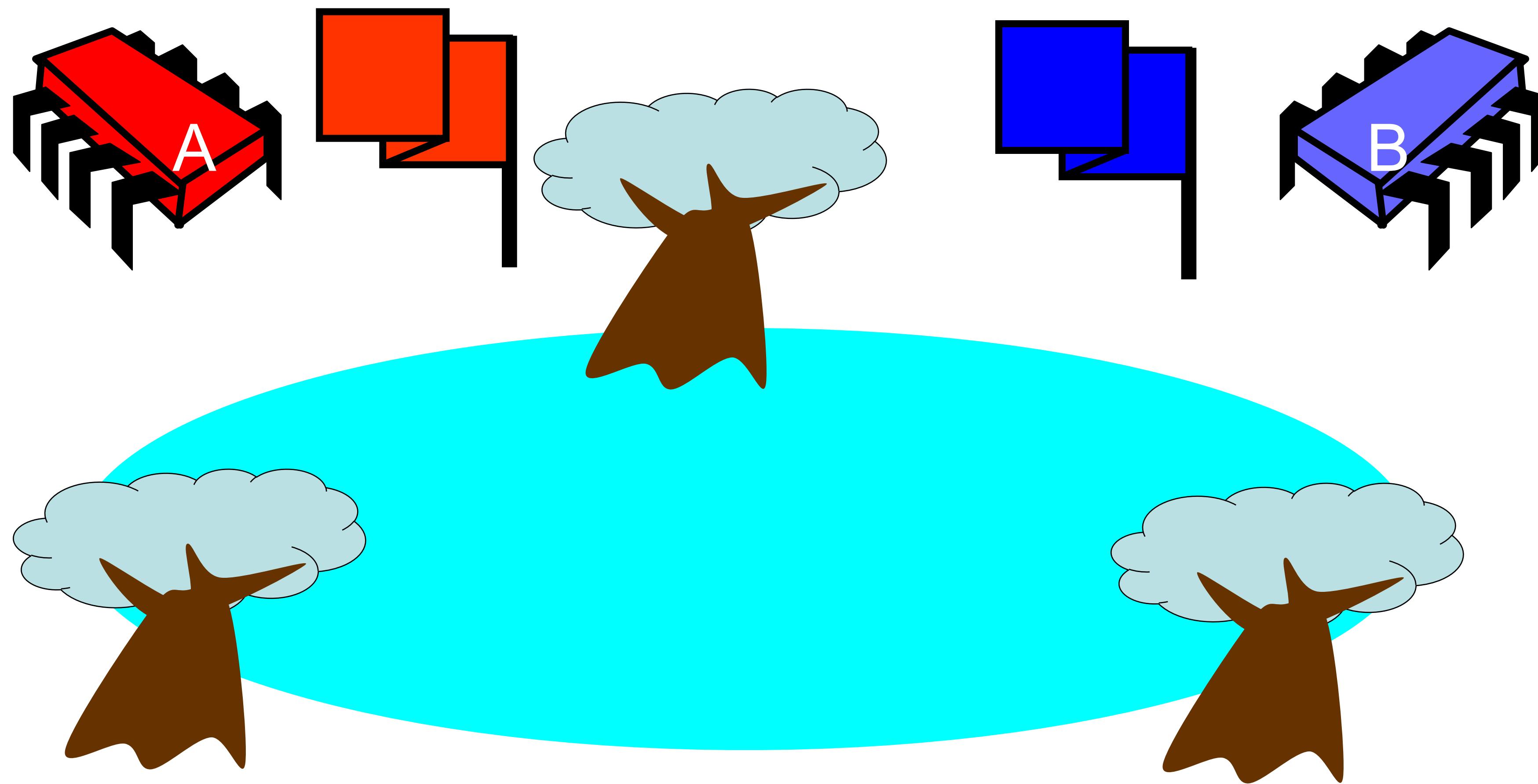
# Correctness

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

# Correctness

- **Mutual Exclusion** 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

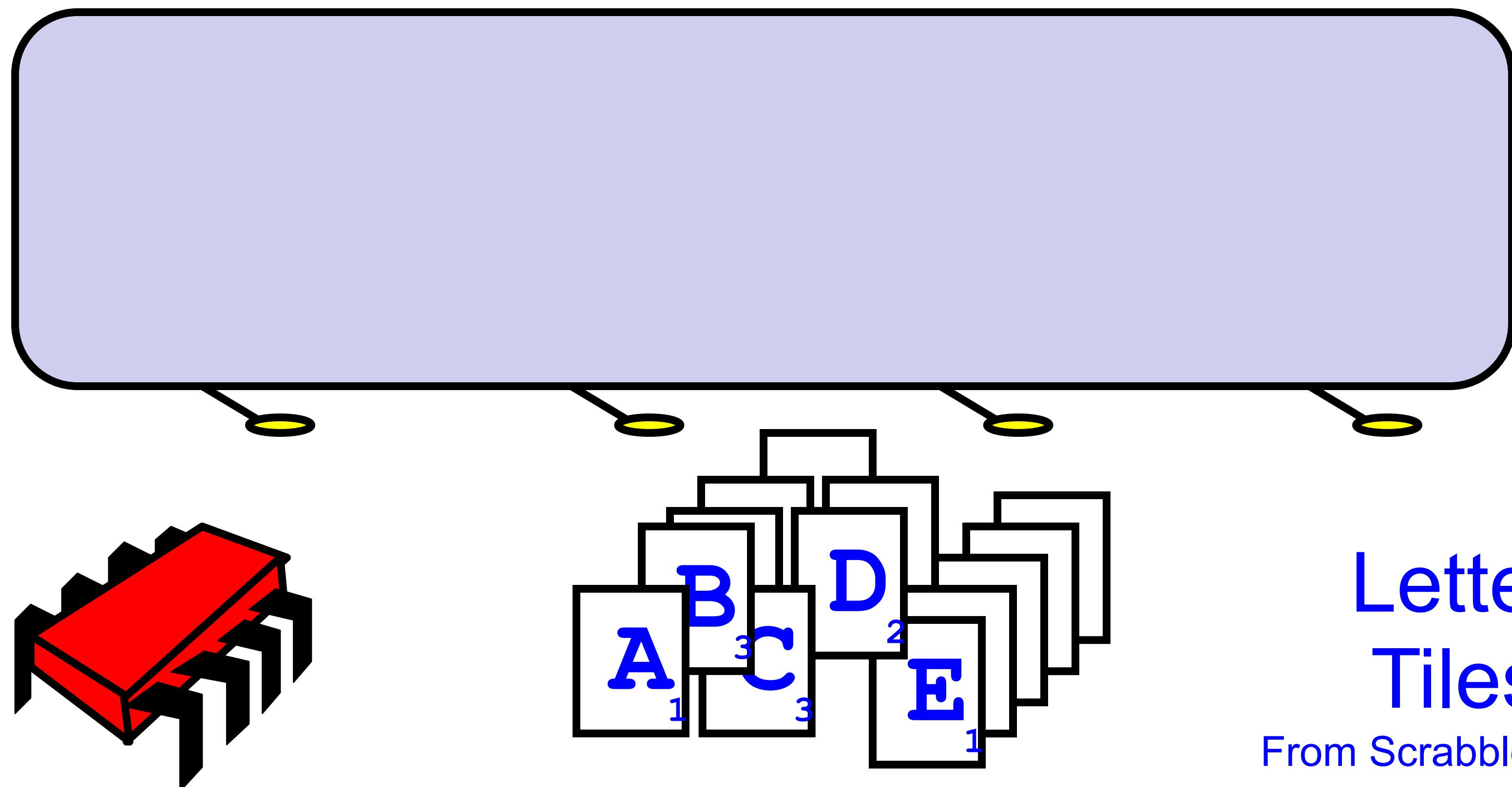
# The Fable drags on ...

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

# The Fable drags on ...

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

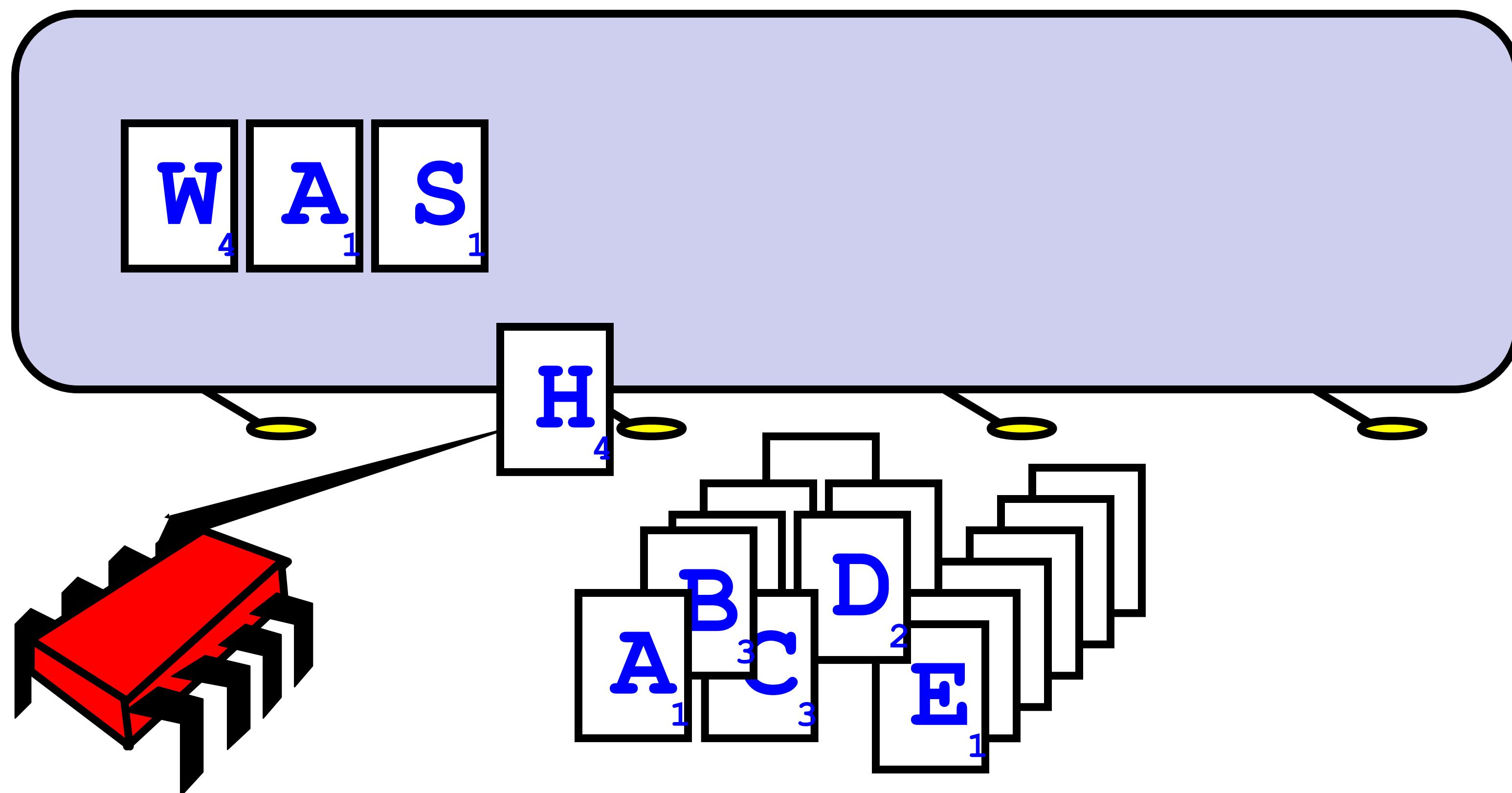
# Billboards are Large



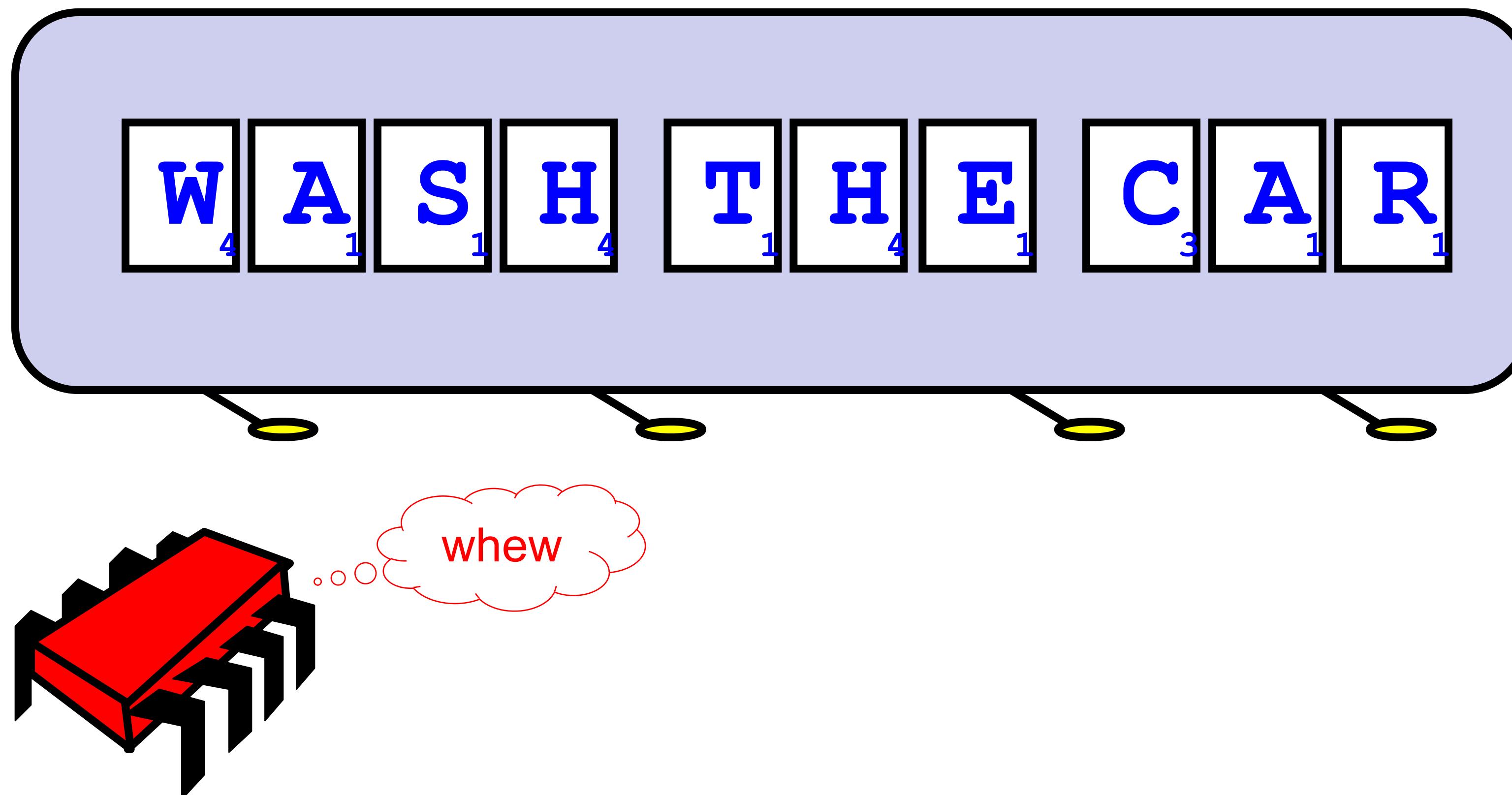
Letter  
Tiles

From Scrabble™ box

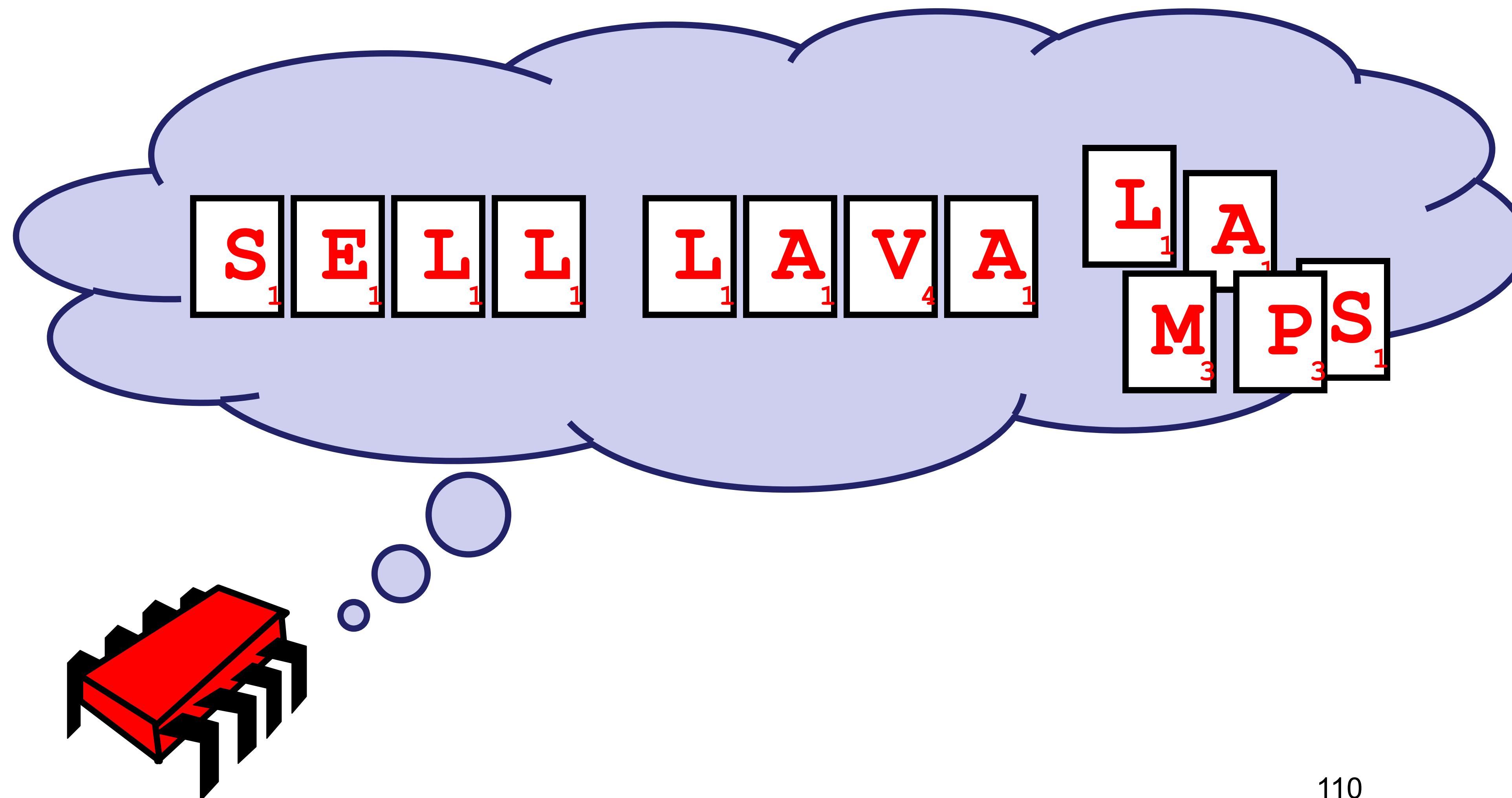
# Write One Letter at a Time ...



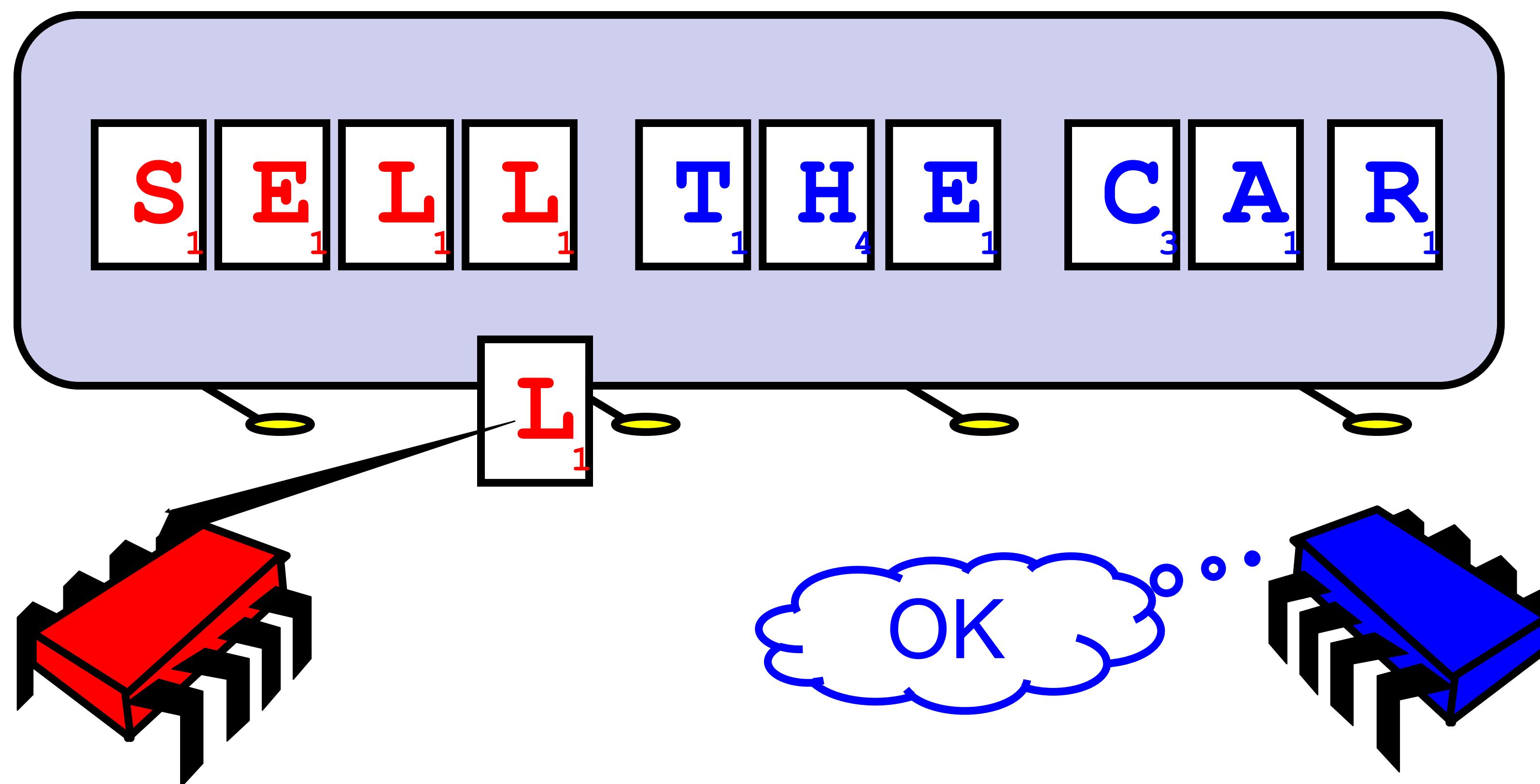
# To post a message



# Let's send another message



# Uh-Oh



# Readers/Writers

- Devise a protocol so that
  - Writer writes one letter at a time
  - Reader reads one letter at a time
  - Reader sees “snapshot”
    - Old message or new message
    - No mixed messages

# Readers/Writers (continued)

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

# Esoteric?

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

performance bottleneck

# Readers/Writers Solution

- Each thread  $i$  has  $\text{size}[i]$  counter
  - only it increments or decrements.
- To get object's size, a thread reads a “snapshot” of all counters
- This eliminates the bottleneck

# Concurrency and Mutual Exclusion

**Mutual Exclusion = Sequential Execution**

# 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

Speedup =

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

# Amdahl's Law

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

# Amdahl's Law

Speedup =

$$\frac{1}{1 - p + \frac{p}{n}}$$

**Parallel fraction**

# Amdahl's Law

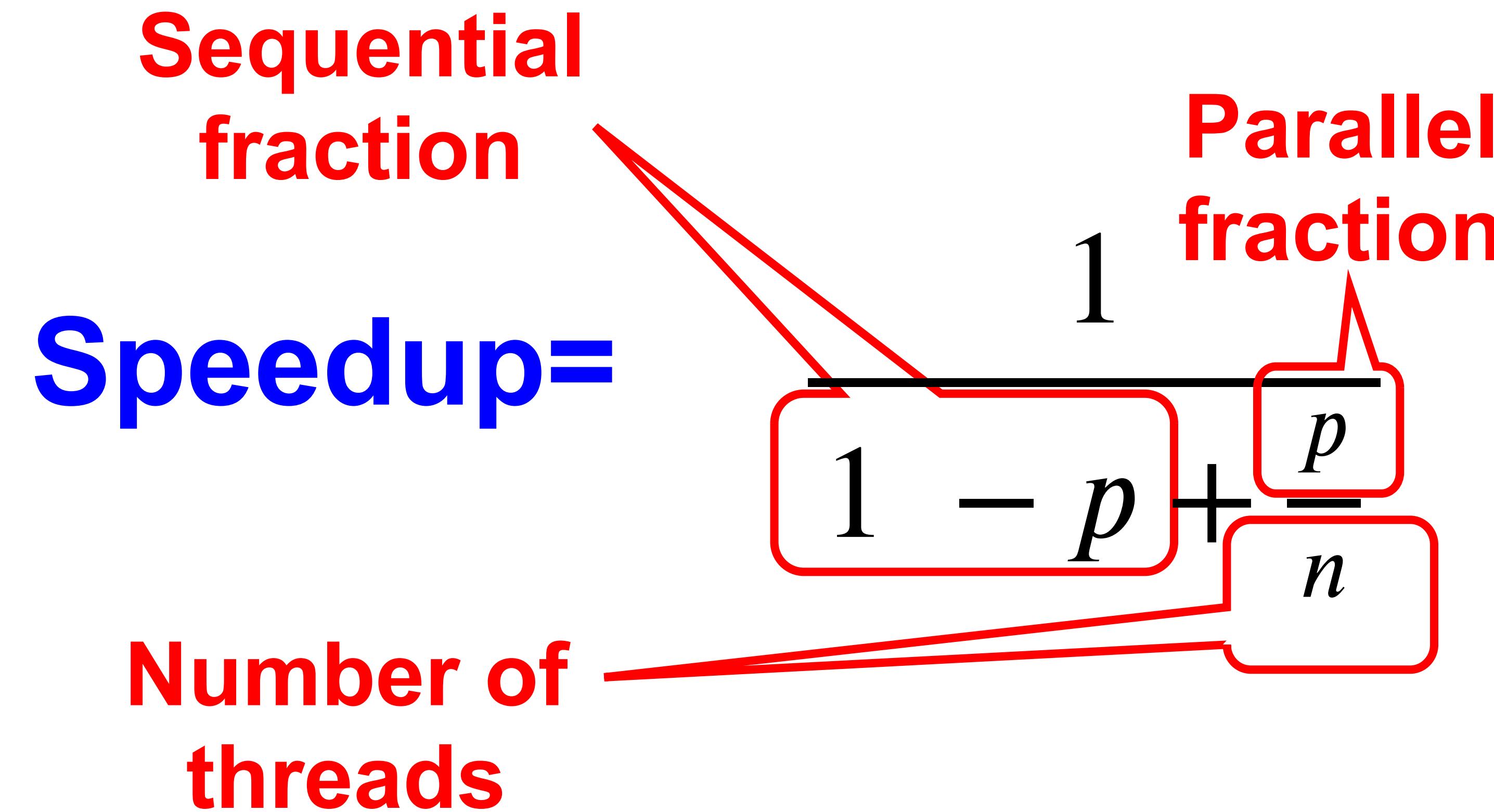
Sequential fraction

Parallel fraction

Speedup =

$$\frac{1}{(1 - p) + \frac{p}{n}}$$

# Amdahl's Law



# 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, 01% sequential
- How close to 10-fold speedup?

# Example

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

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

# Next Week

- Basics of Scala programming
- *Formal model* for thinking about concurrency
- Algorithms for *mutual exclusion*

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.

