

Concurrency: Multi-core Programming & Data Processing



The Basics

Prof. P. Felber

Pascal.Felber@unine.ch

<http://iiun.unine.ch/>

Based on slides by Maurice Herlihy and Nir Shavit

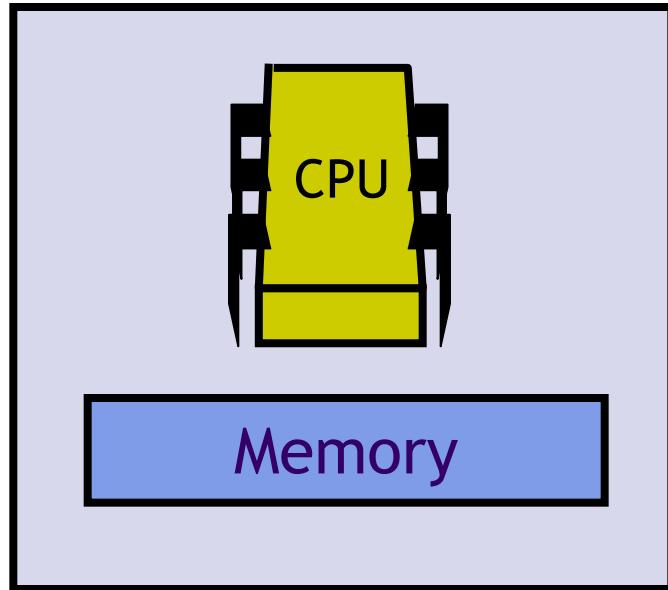


From the New York Times...

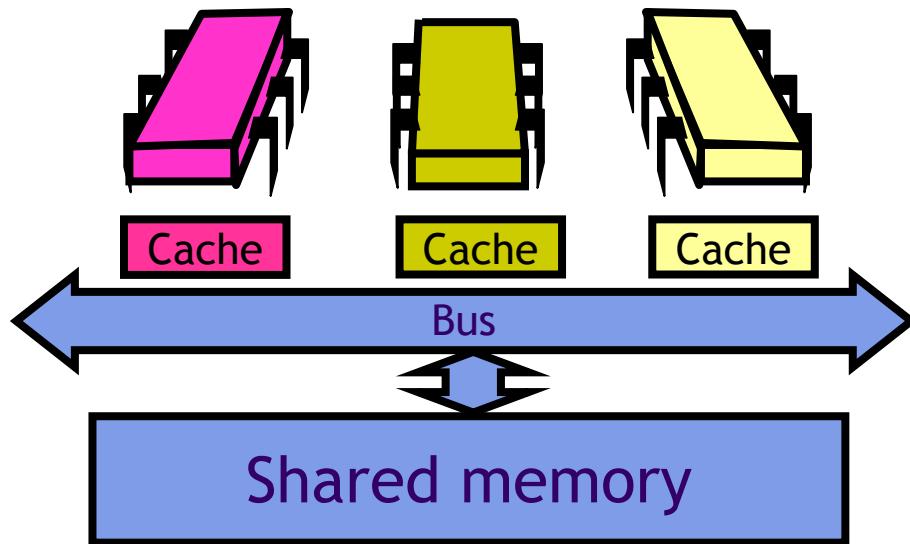
SAN FRANCISCO, May 7, 2004:

“Intel said on Friday that it was scrapping its development of two microprocessors, a move that is a shift in the company’s business strategy...”

On Your Desktop: The Uniprocessor

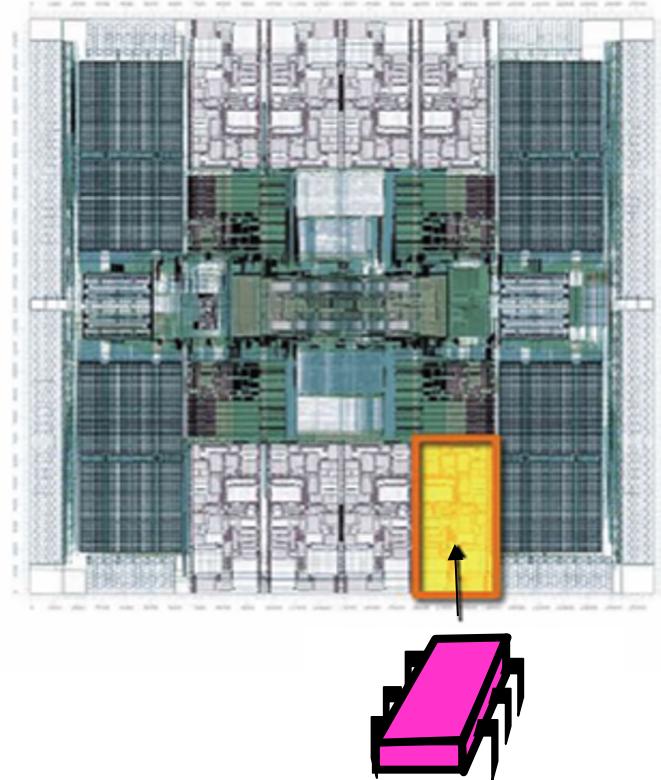


In the Enterprise: The Shared Memory Multiprocessor (SMP)



Your New Desktop: The Multicore Processor (CMP)

All on the
same chip



Sun
T2000
Niagara

Why do we Care?

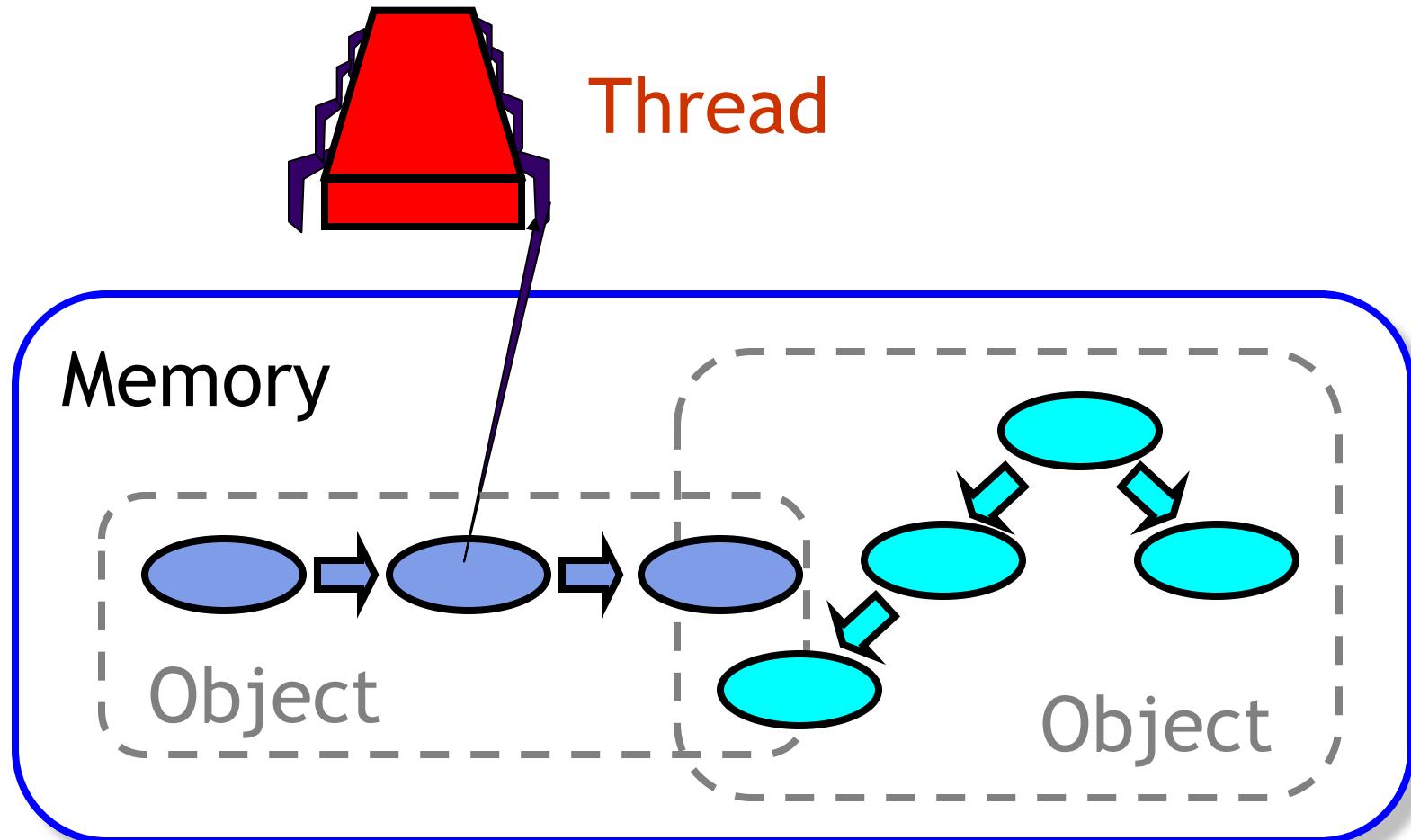
- Time no longer cures software bloat
 - The “free ride” is over
- When you double your program’s path length
 - You cannot just wait 6 months
 - Your software must somehow exploit twice as much concurrency

Multiprocessor Programming:

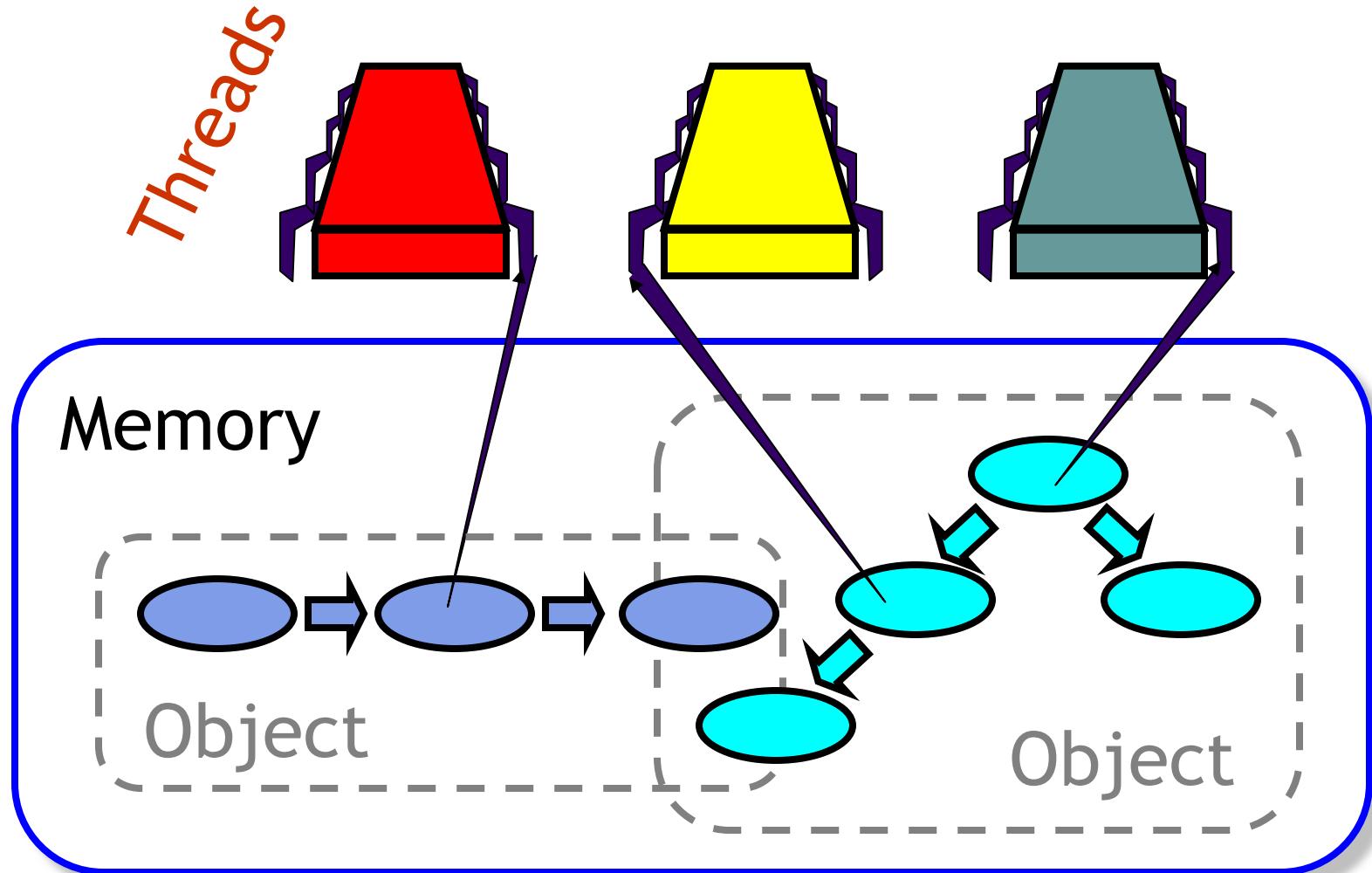
Course Overview

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

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

Road Map

- We are going to focus on **principles** first, then **practice**
 - Start with **idealized** models
 - Look at **simplistic** problems
 - Emphasize **correctness** over pragmatism

*“Correctness may be theoretical,
but incorrectness has practical impact”*

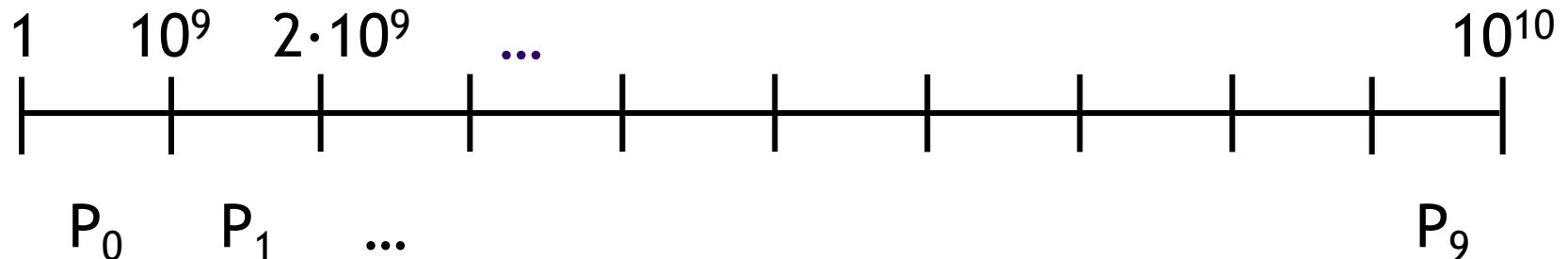
Concurrency Jargon

- Hardware
 - Processors
- Software
 - Threads, processes
- Sometimes OK to confuse these terms,
sometimes not

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

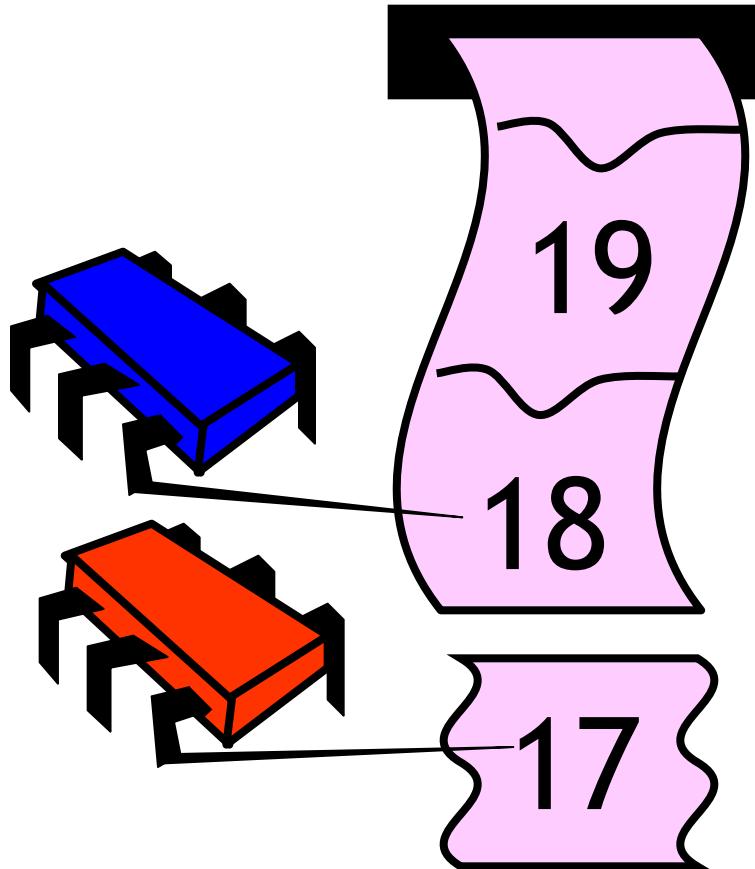
```
void primePrint {  
    int i = ThreadID.get(); // IDs in {0..9}  
    for (j = i*109+1; j <= (i+1)*109; j++) {  
        if (isPrime(j))  
            print(j);  
    }  
}
```

Issues

- Larger numbers imply fewer primes
- Yet larger numbers harder to test
- Thread workloads
 - Uneven
 - Hard to predict
- Need *dynamic* load balancing

Rejected

Shared Counter



Each thread
takes a number

Procedure for Thread i

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

void primePrint {
    while (true) {
        long j = counter.getAndIncrement();
        if (j > 1010)
            break;
        if (isPrime(j))
            print(j);
    }
}
```

Procedure for Thread *i*

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

```
void primePrint {  
    while (true) {  
        long j = counter.getAndIncrement();  
        if (j > 1010)  
            break;  
        if (isPrime(j))  
            print(j);  
    }  
}
```

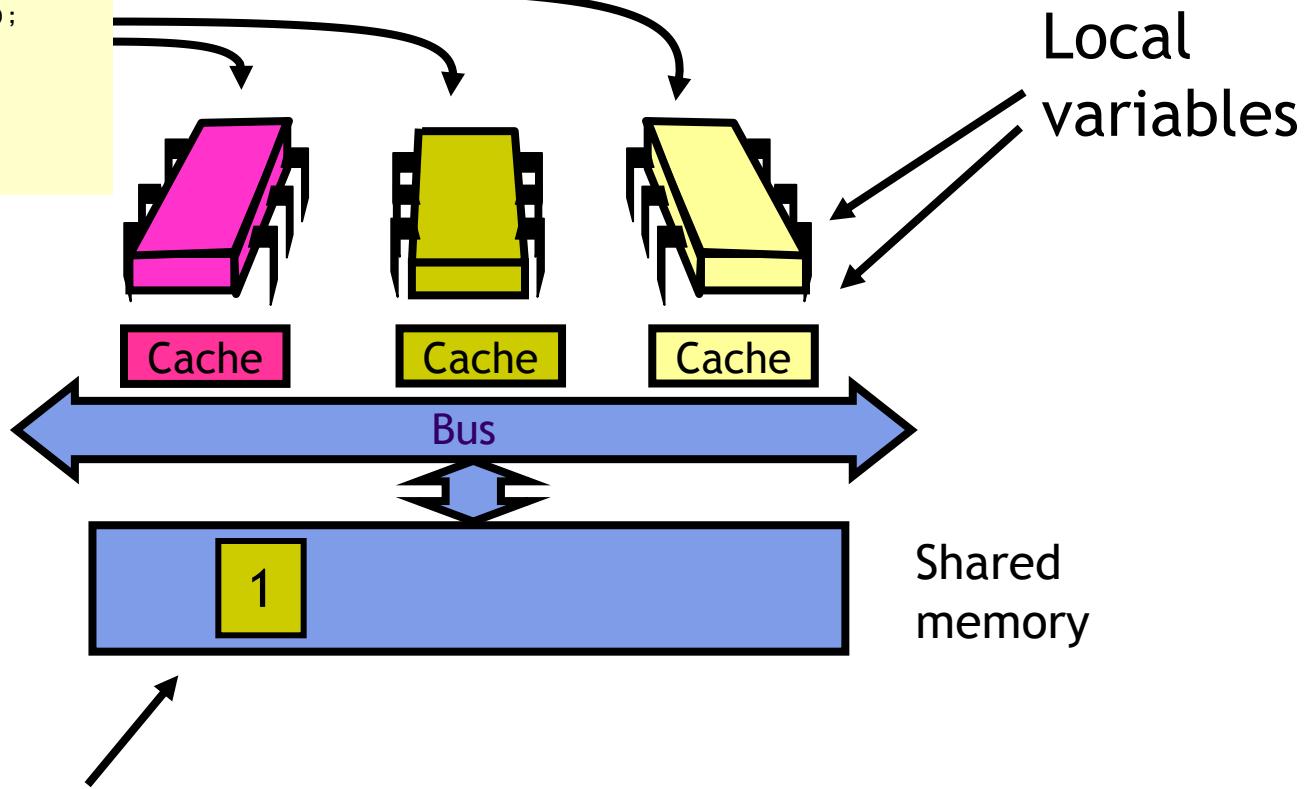
Shared counter
object

Where Things Reside

```
Counter counter = new Counter(1);
void primePrint {
    while (true) {
        long j = counter.getAndIncrement();
        if (j > 1010)
            break;
        if (isPrime(j))
            print(j);
    }
}
```

Code

Shared counter



Procedure for Thread *i*

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

void primePrint {
    while (true) {
        long j = counter.getAndIncrement();
        if (j > 1010)
            break;
        if (isPrime(j))
            print(j);
    }
}
```

Increment & return each new value

Procedure for Thread *i*

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

void primePrint {
    while (true) {
        long j = counter.getAndIncrement();
        if (j > 1010)
            break;
        if (isPrime(j))
            print(j);
    }
}
```

Stop when every value taken

Counter Implementation

```
public class Counter {  
    private long value;  
  
    public long get() {  
        return value;  
    }  
}
```

OK for uniprocessor,
not for multiprocessor

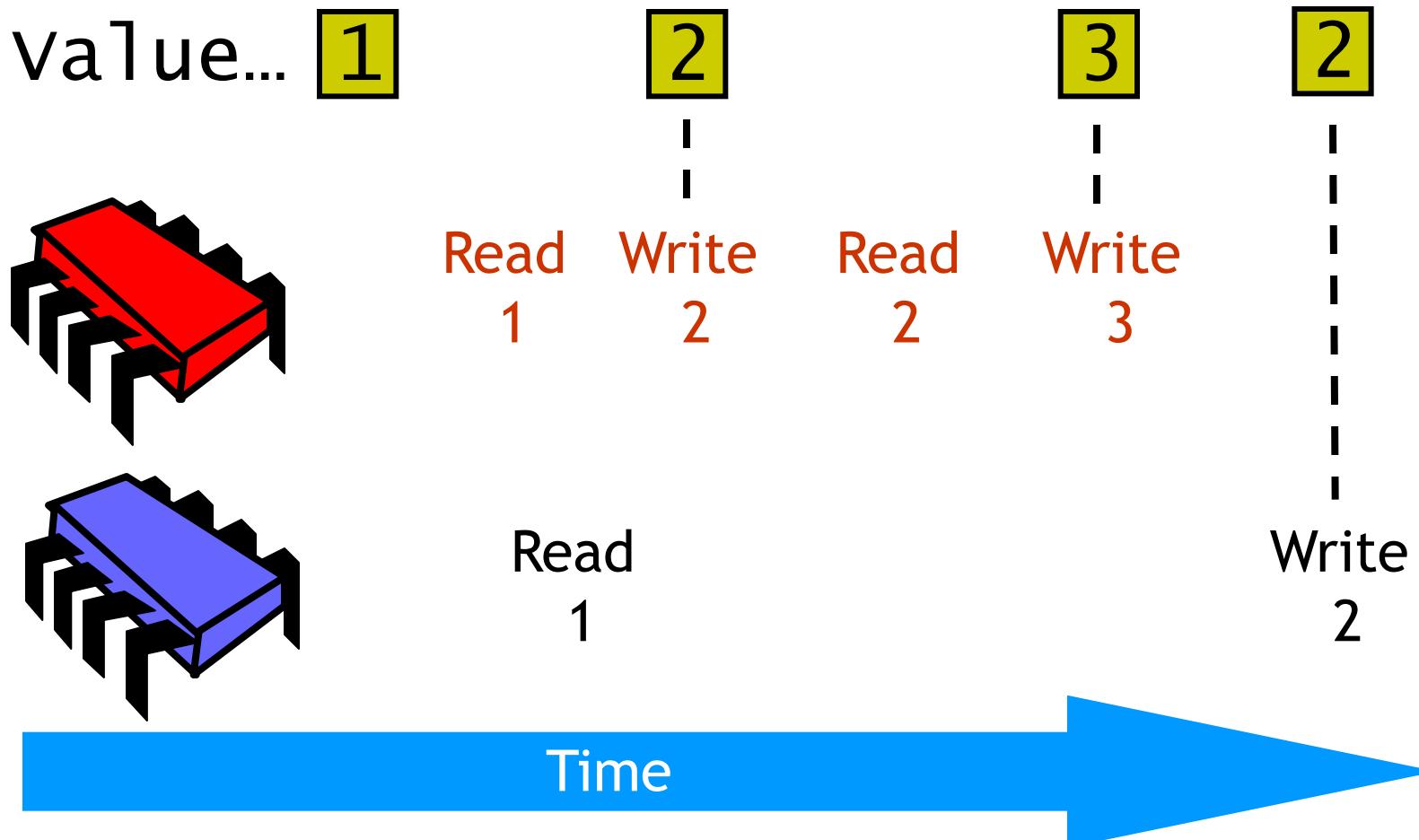
What It Means

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        return value++;  
    }  
}
```

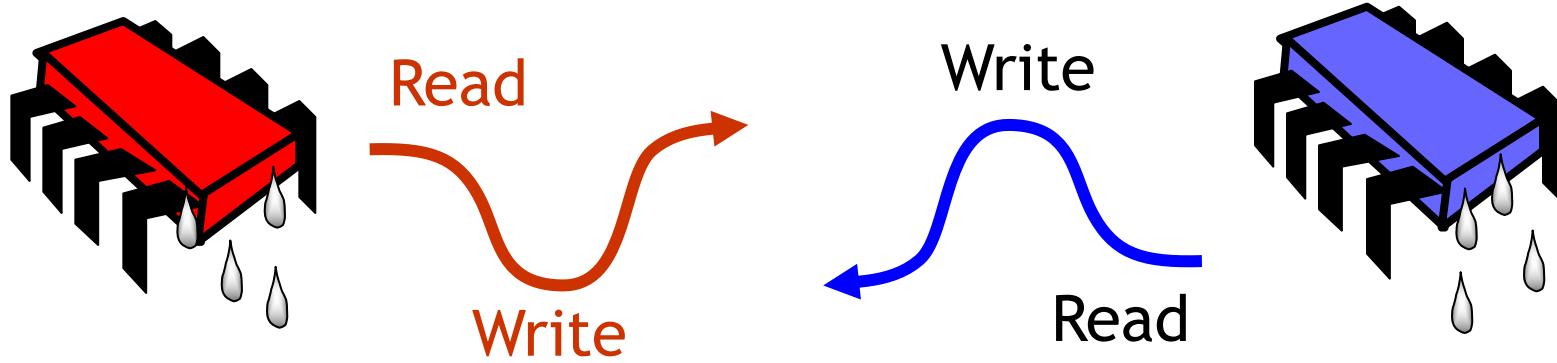
A red callout box surrounds the line `return value++;`. A red arrow points from this box to the line `long temp = value;`.

```
long temp = value;  
value = value + 1;  
return temp;
```

Not so Good...



Is this Problem Inherent?



If we could only glue reads and writes...

Challenge

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

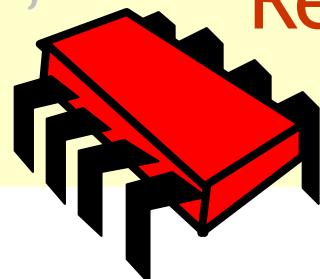
Challenge

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

Make these steps
atomic (indivisible)

Hardware Solution

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



Read-modify-write
instruction

An Aside: Java

```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        synchronized {  
            long temp = value;  
            value = temp + 1;  
        }  
        return temp;  
    }  
}
```

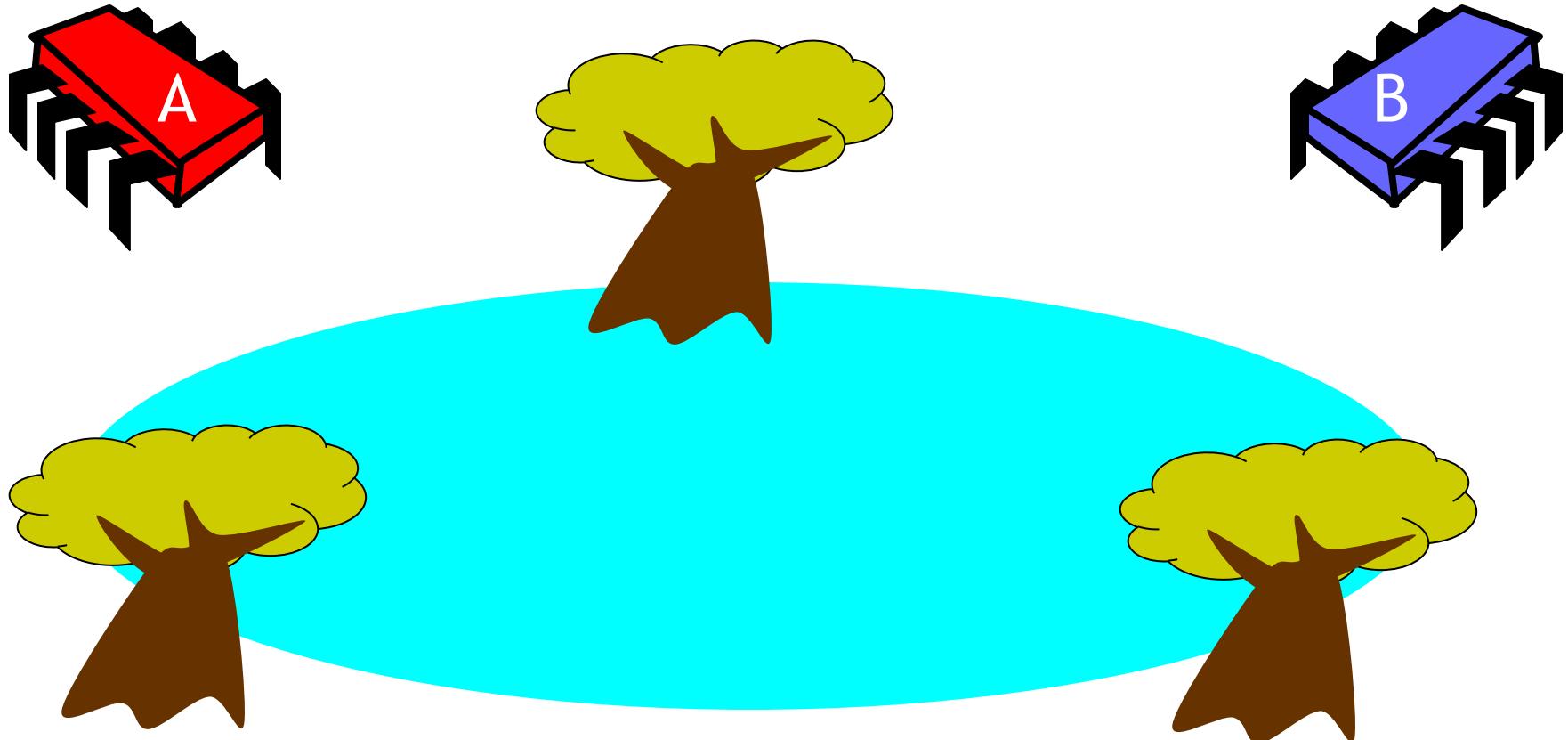
Synchronized block

An Aside: Java

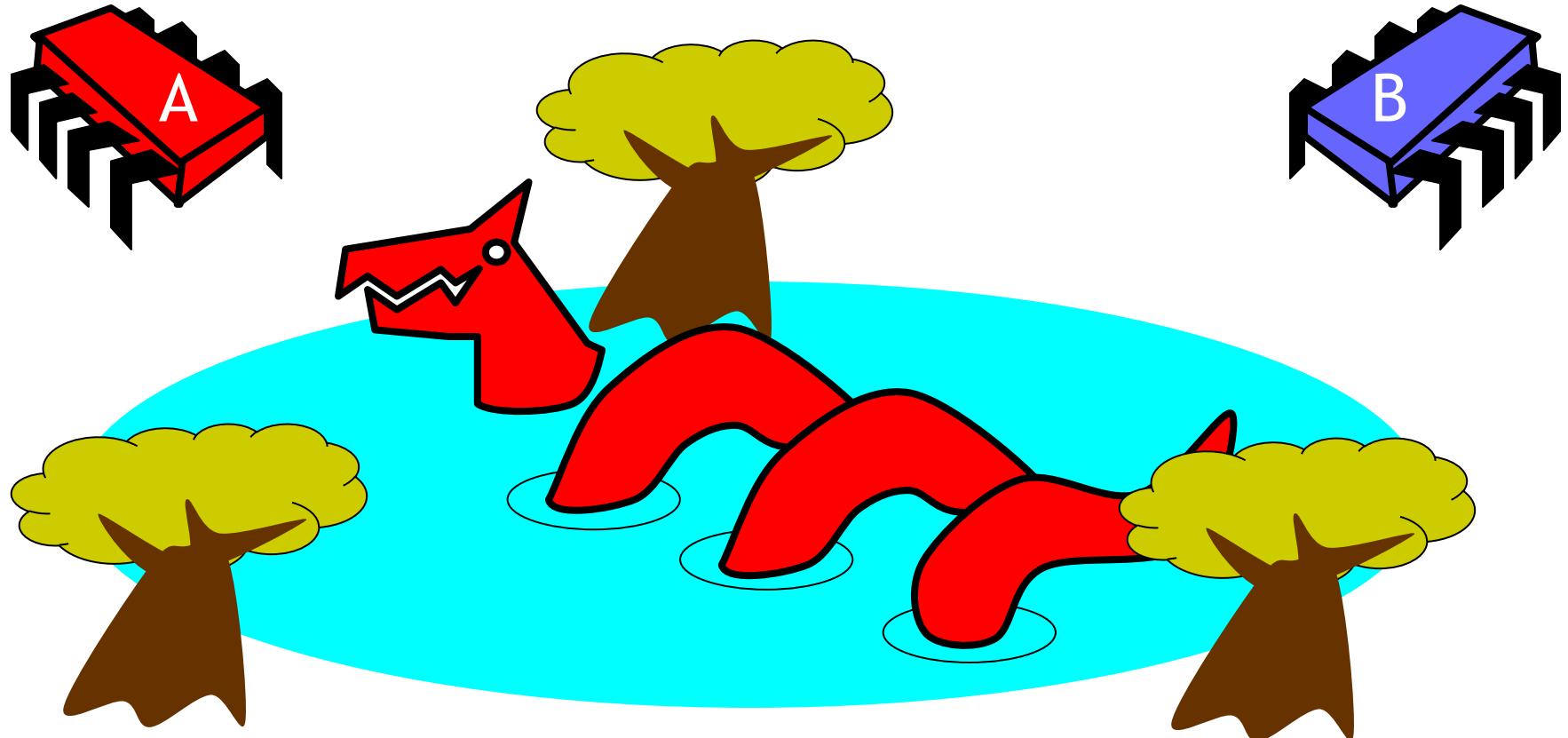
```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        synchronized {  
            long temp = value;  
            value = temp + 1;  
        }  
        return temp;  
    }  
}
```

Mutual exclusion

Mutual Exclusion or “Alice & Bob Share a Pond”



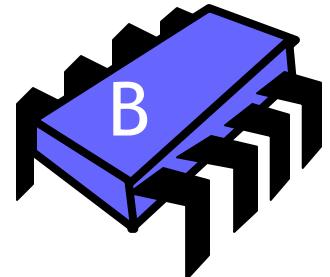
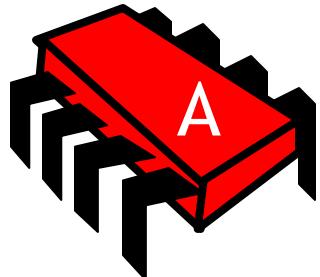
Alice has a Pet



Bob has a Pet



The Problem



The pets don't
get along



Formalizing the Problem

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

Formalizing the 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
 - Trees obscure the view

Interpretation

- Threads cannot “see” what other threads are doing
- Explicit communication required for coordination

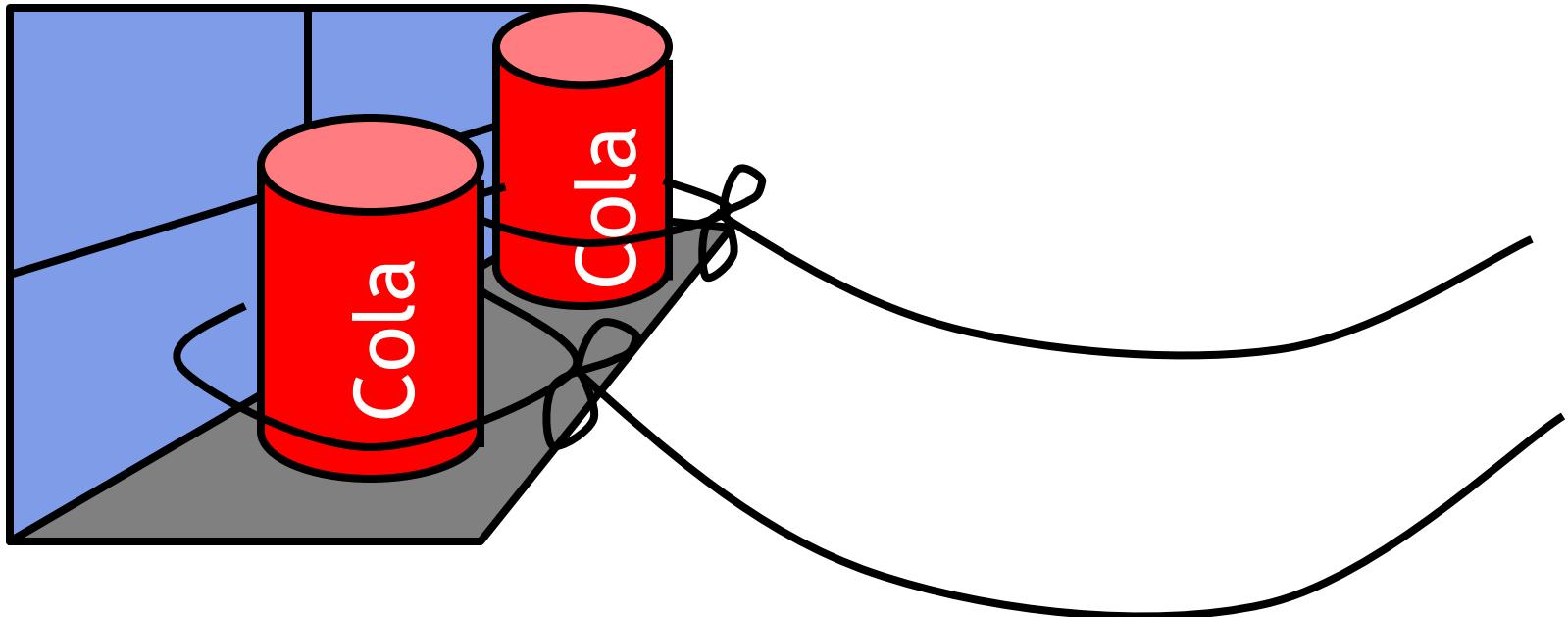
Cell Phone Protocol

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

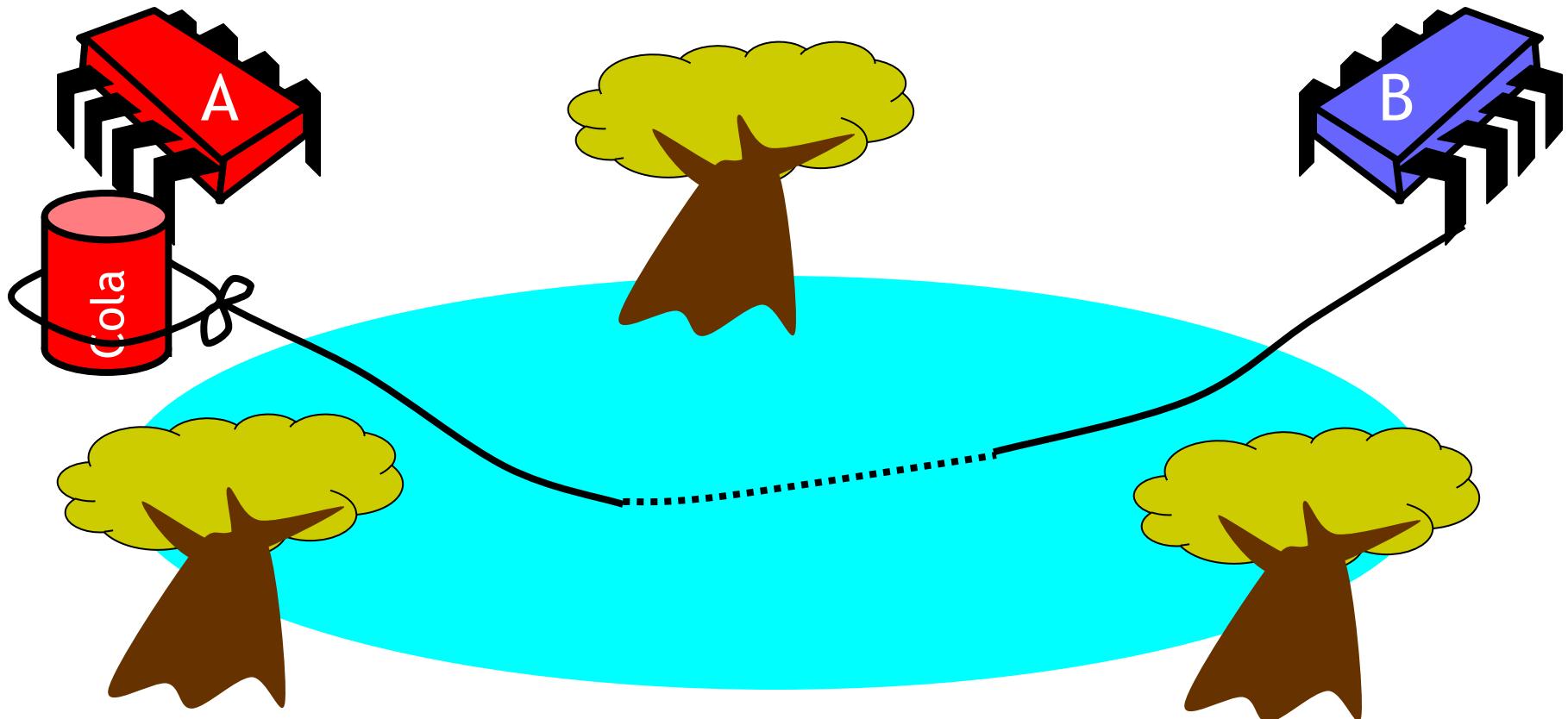
Interpretation

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

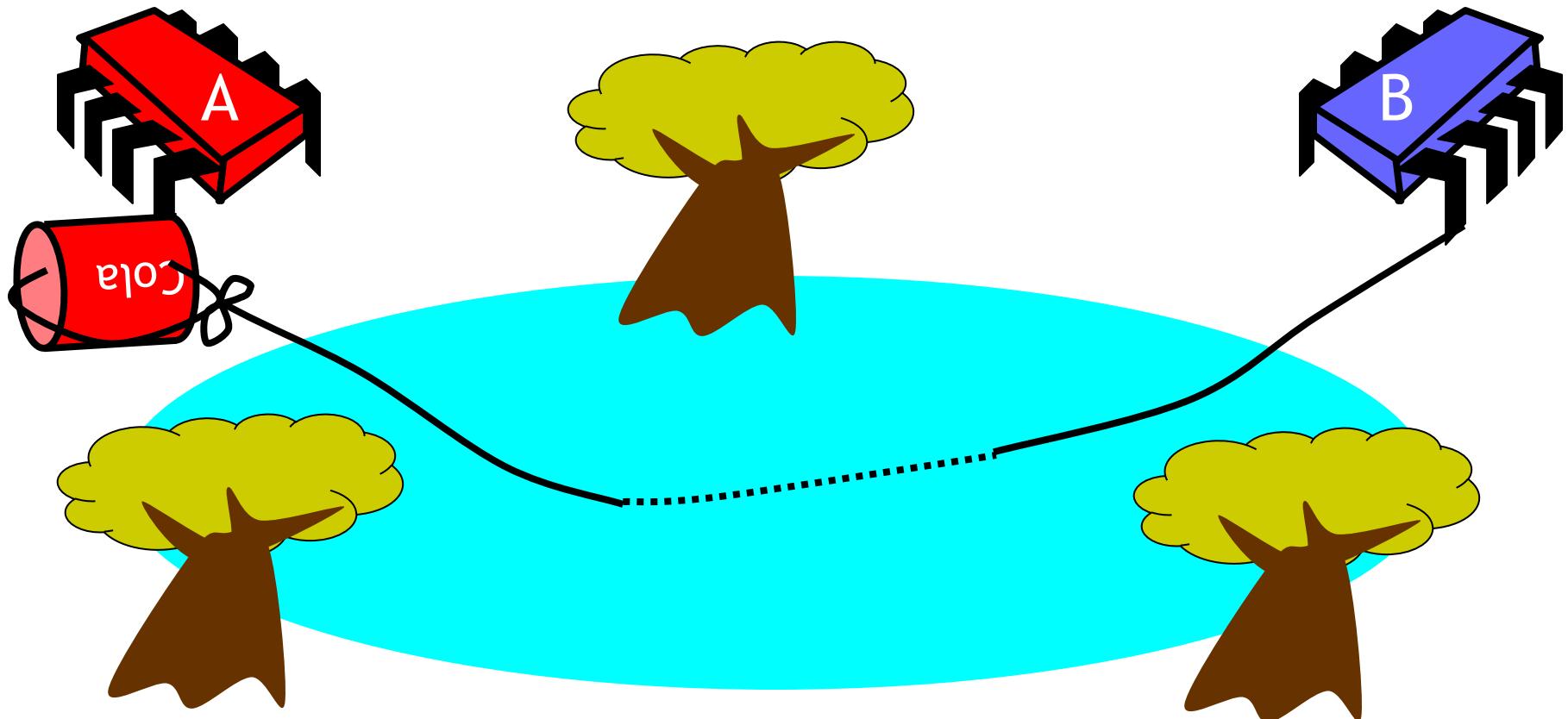
Can Protocol



Bob Conveys a Bit



Bob Conveys a Bit



Can Protocol

- Idea

- Cans on Alice's windowsill
- Strings lead to Bob's house
- Bob pulls strings, knocks over cans

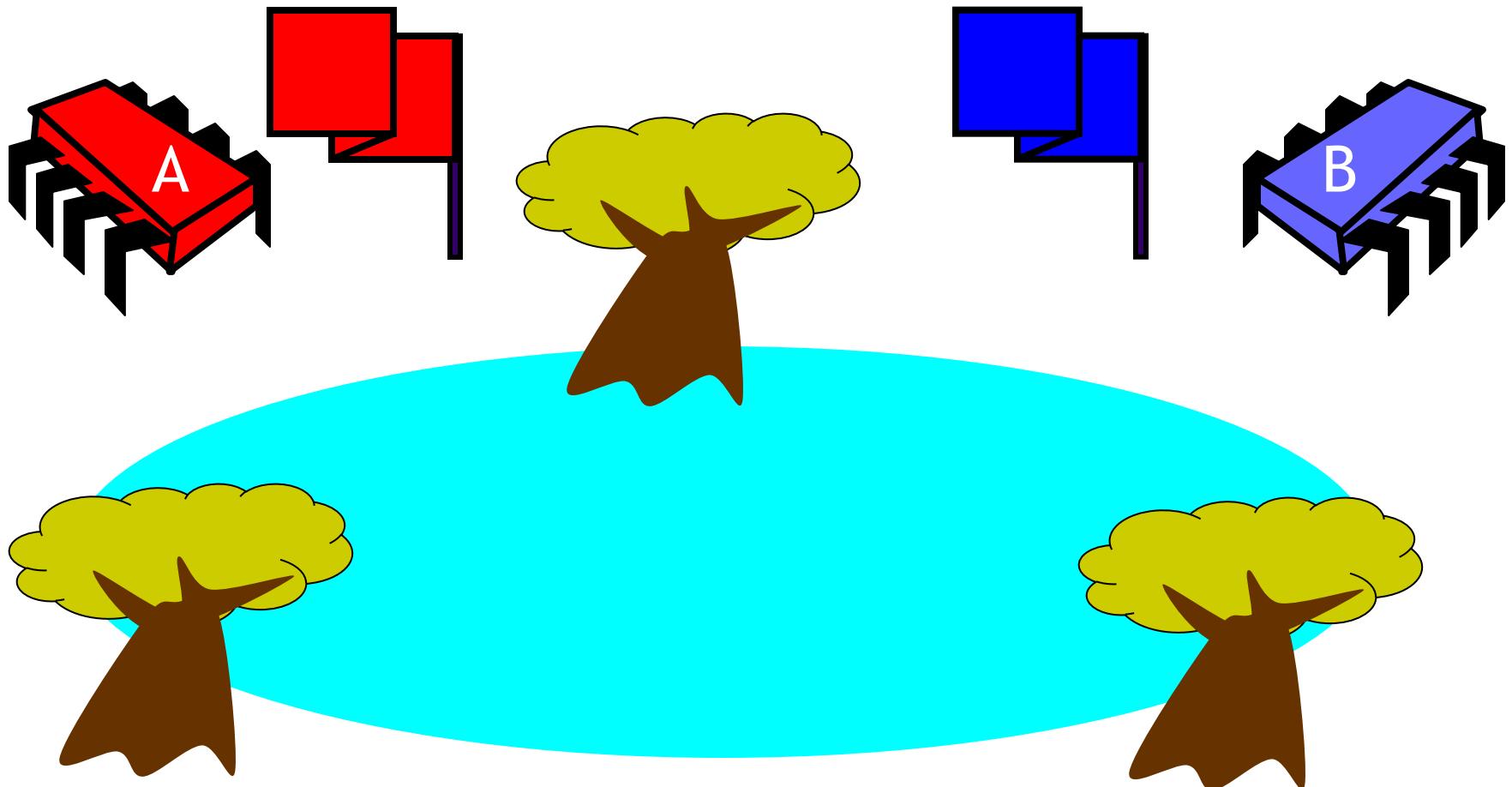
- Gotcha

- Cans cannot be reused
- Bob runs out of cans

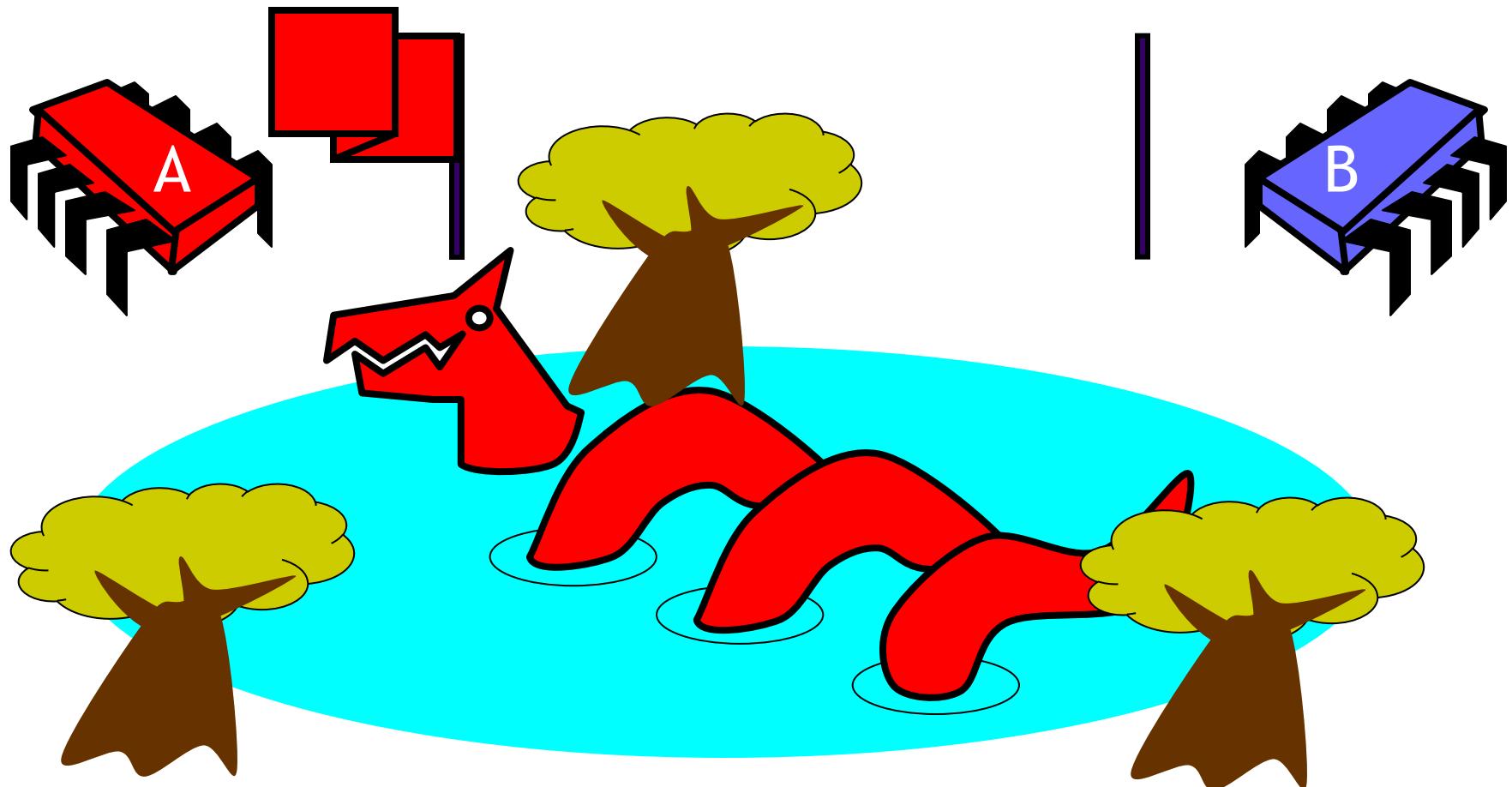
Interpretation

- 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

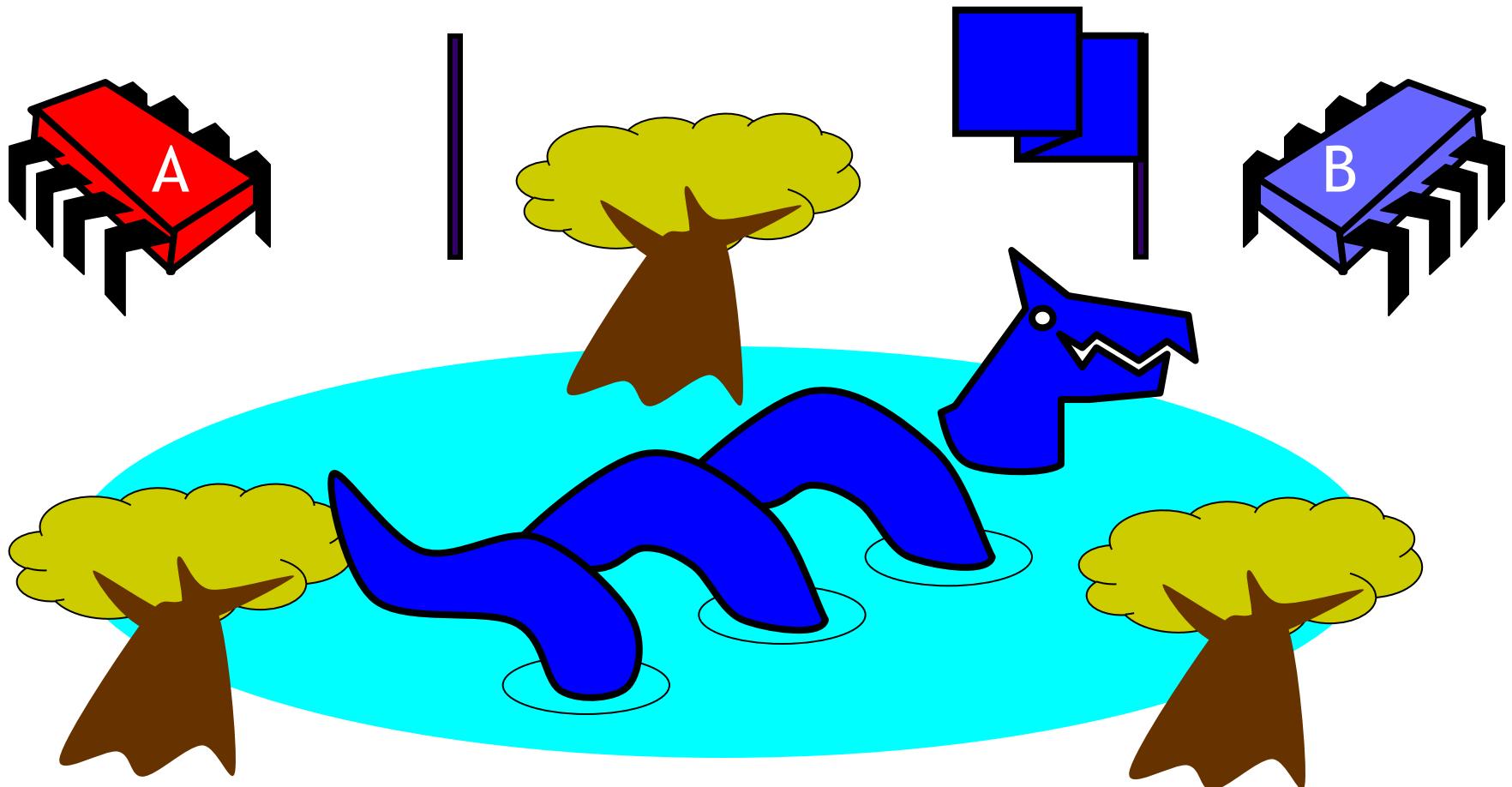
Flag Protocol



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



Bob's Protocol (2nd try)

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

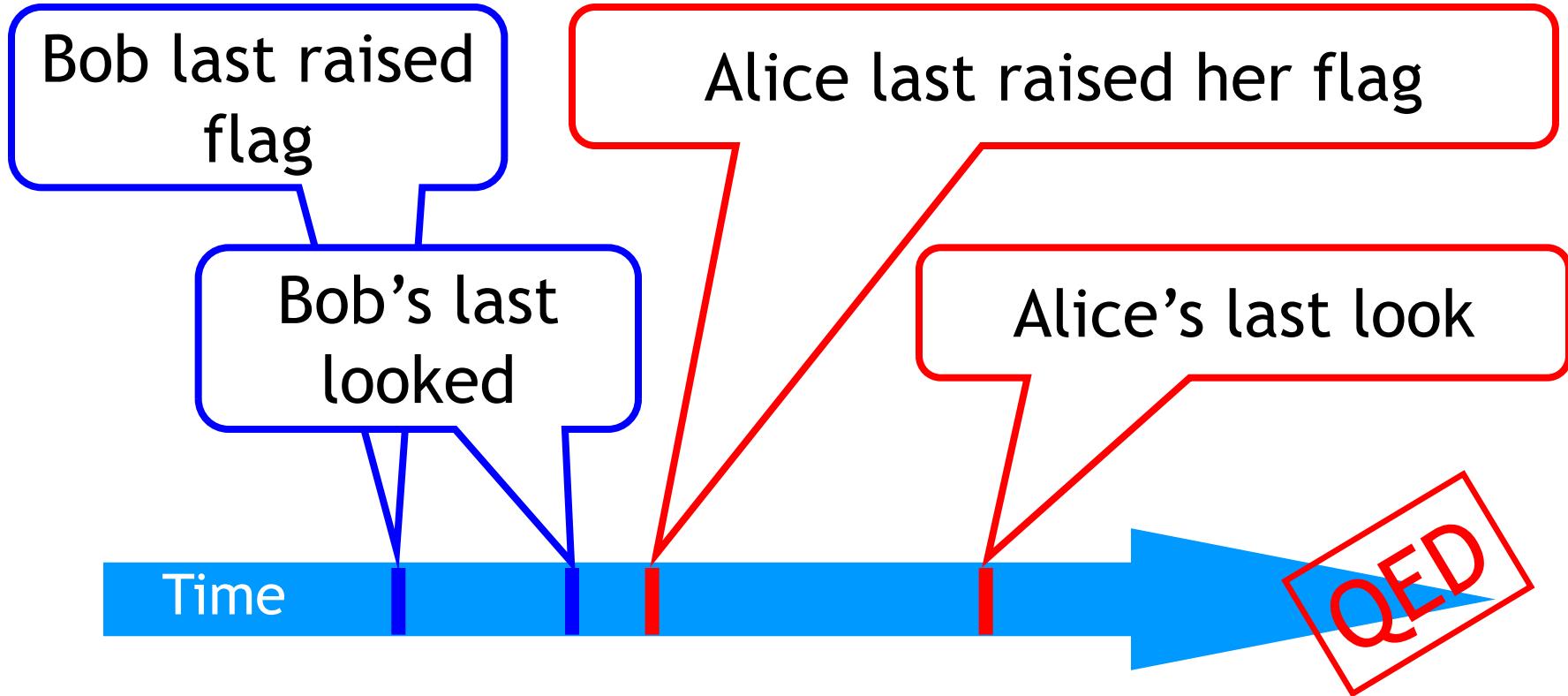
The Flag Principle

- Raise the flag
- Look at other's flag
- Flag principle
 - If each raises and looks, then...
 - 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

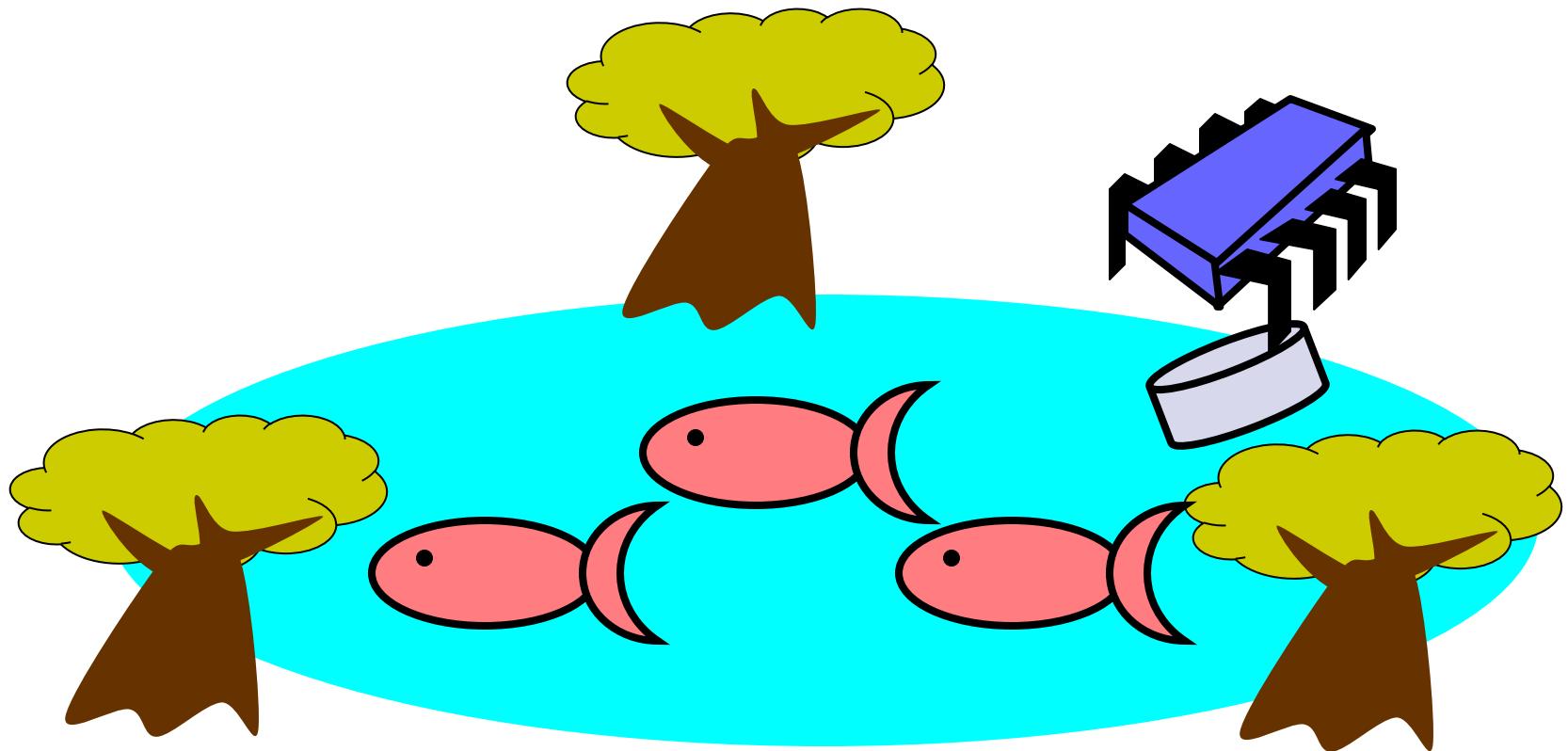
Moral of Story

- 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

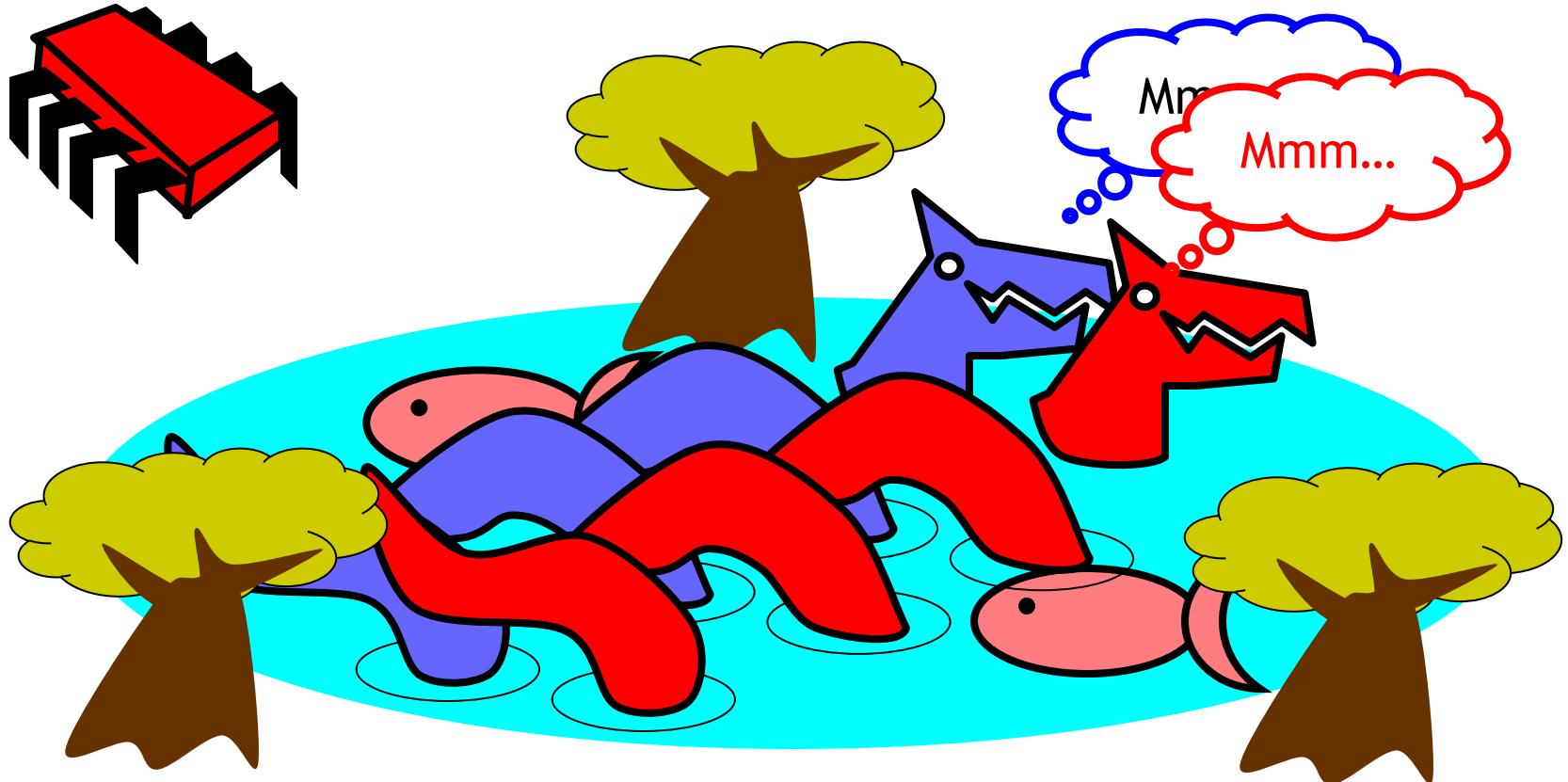
The Fable Continues

- 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

Bob Puts Food in the Pond



Alice Releases her Pets to Feed



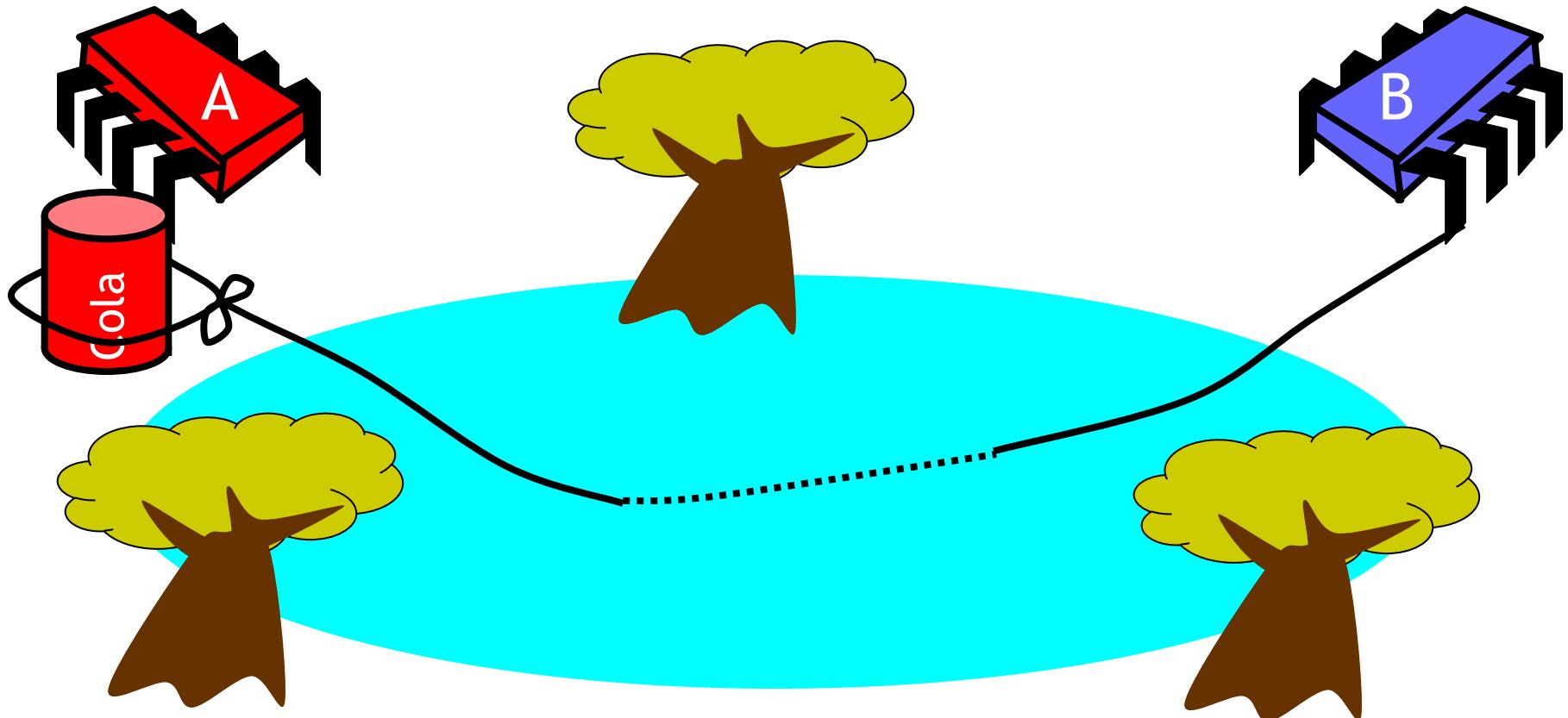
Producer/Consumer

- Alice and Bob cannot 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 is no food
 - Putting out food if uneaten food remains

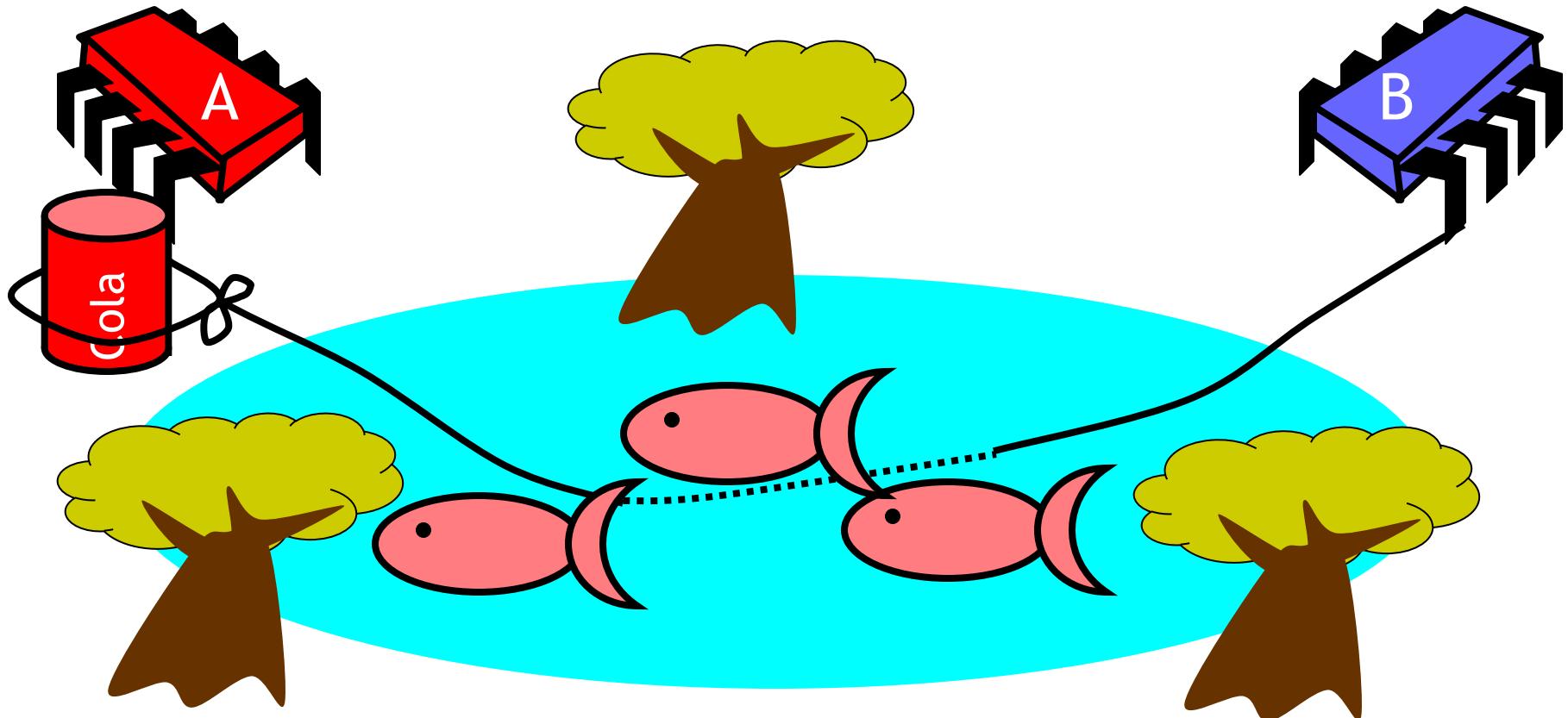
Producer/Consumer

- 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

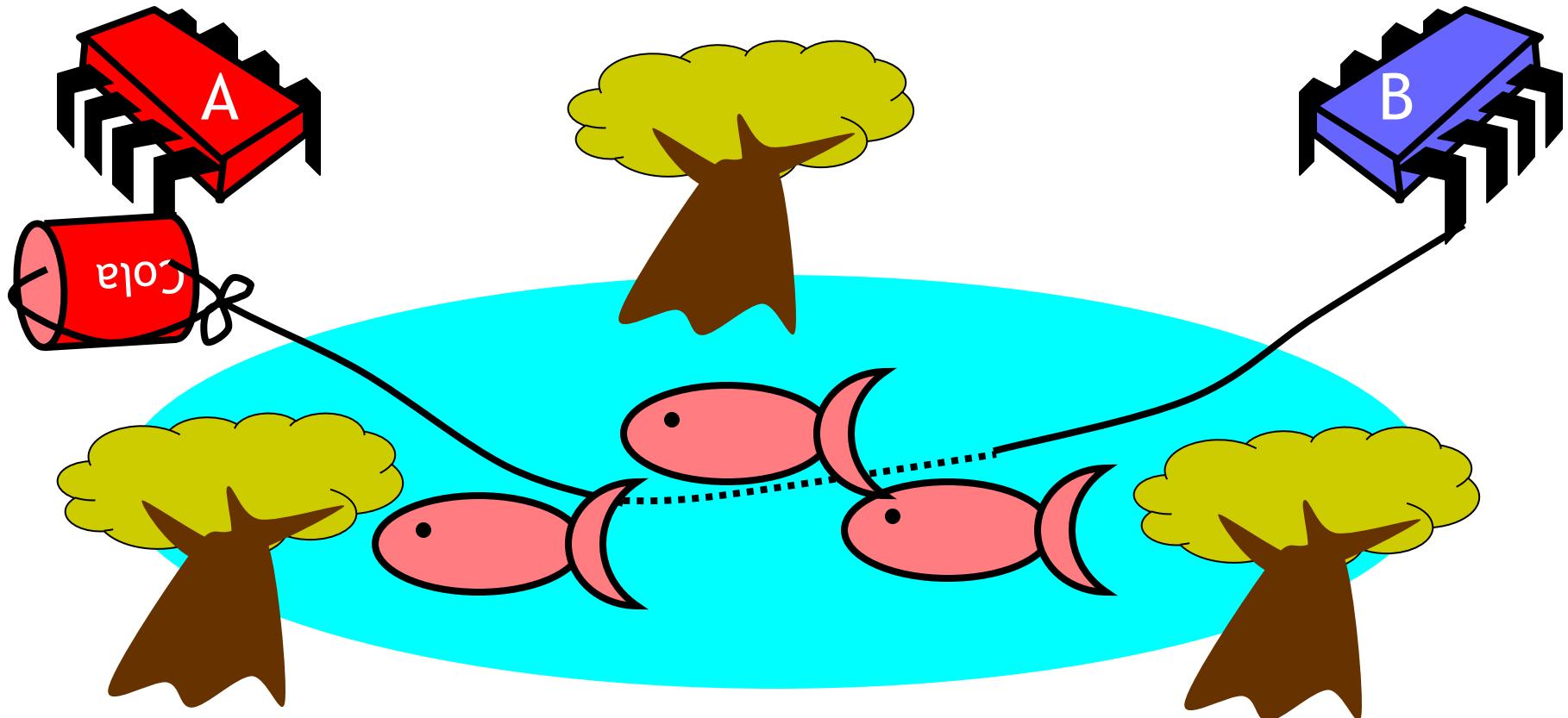
Surprise Solution



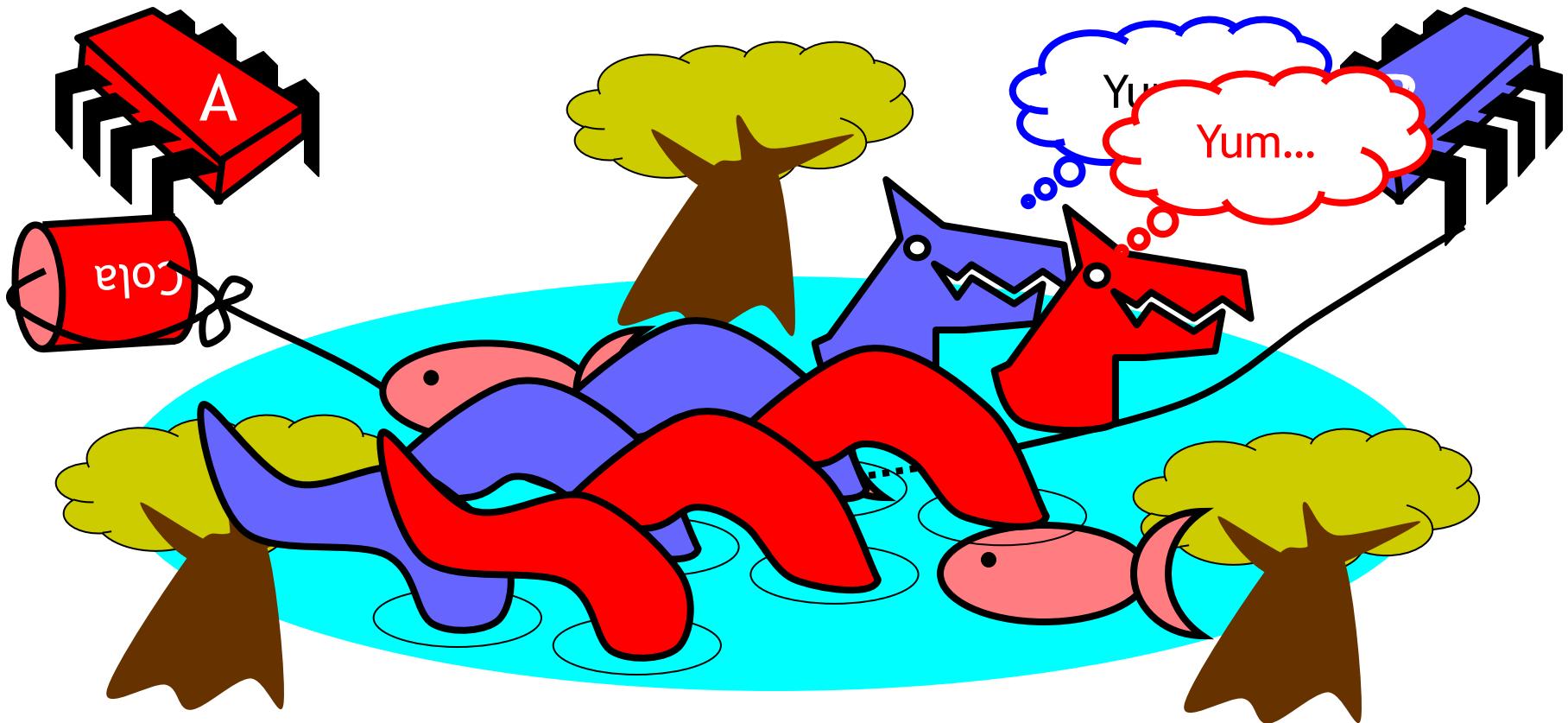
Bob Puts Food in Pond



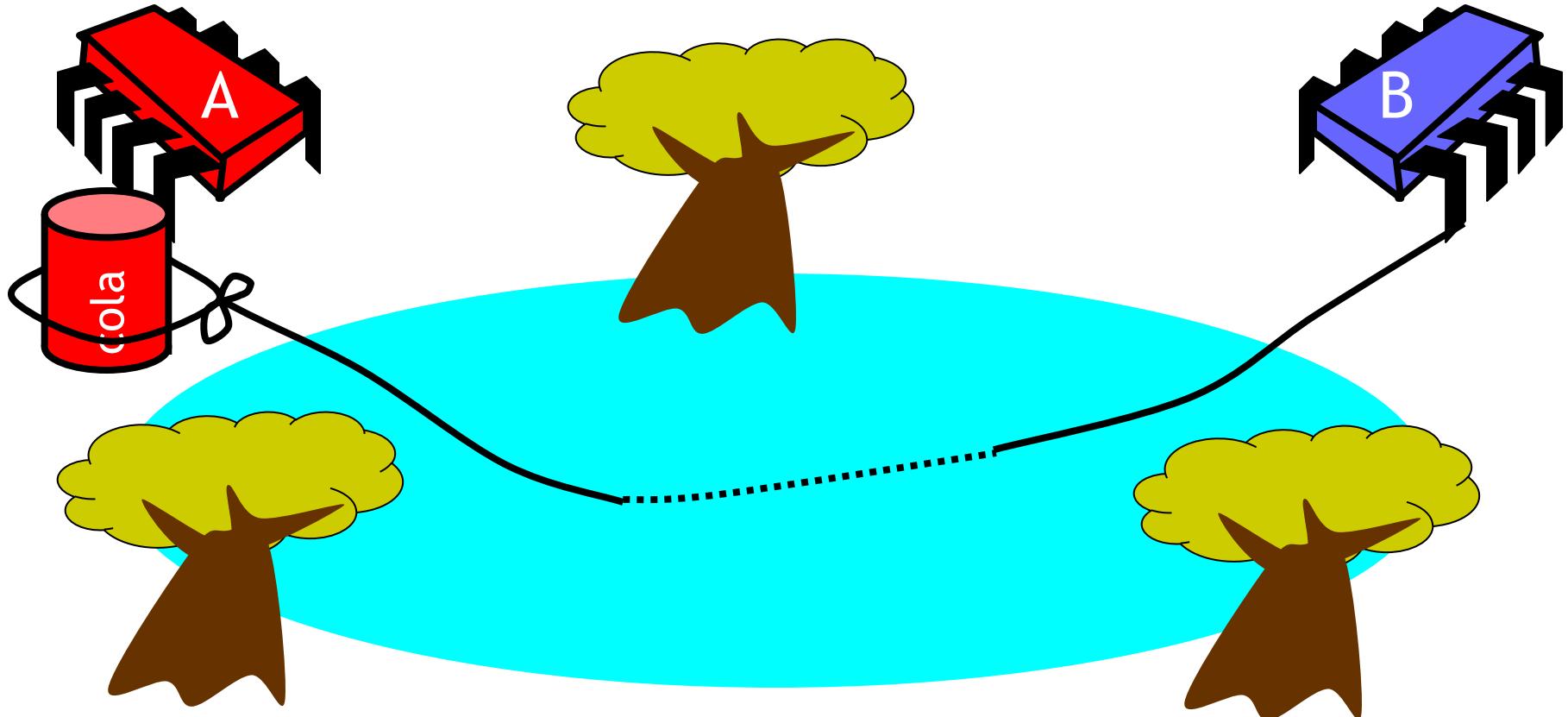
Bob Knocks over Can



Alice Releases Pets



Alice Resets Can once Pets Fed



Pseudocode

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

Alice's code

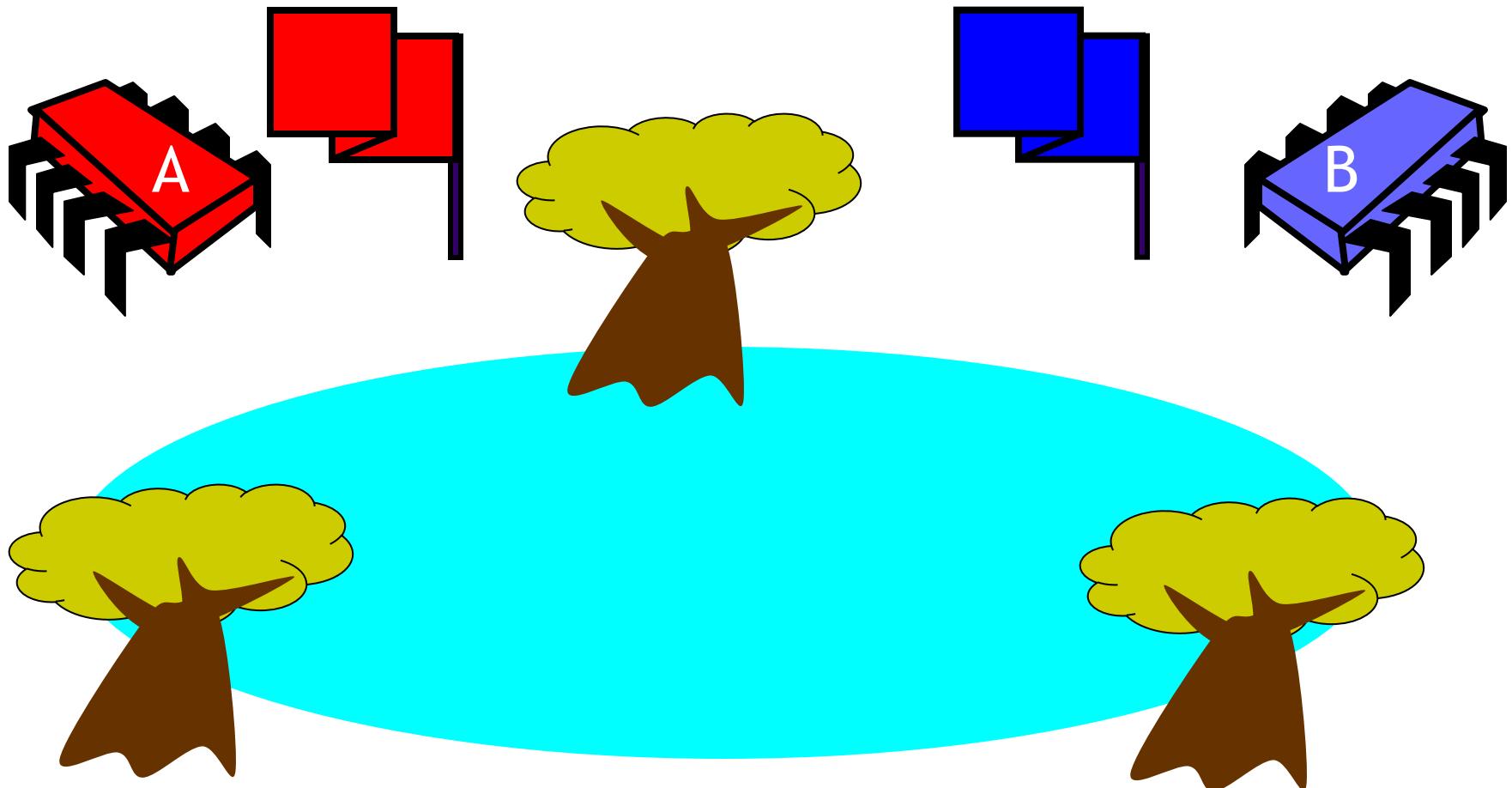
Bob's code

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

Correctness

- Mutual exclusion Safety
 - Pets and Bob never together in pond
- No starvation Liveness
 - If Bob always willing to feed, and pets always famished, then pets eat infinitely often
- Producer/consumer Safety
 - The pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food

Could also Solve using Flags



Waiting

- Both solutions use waiting

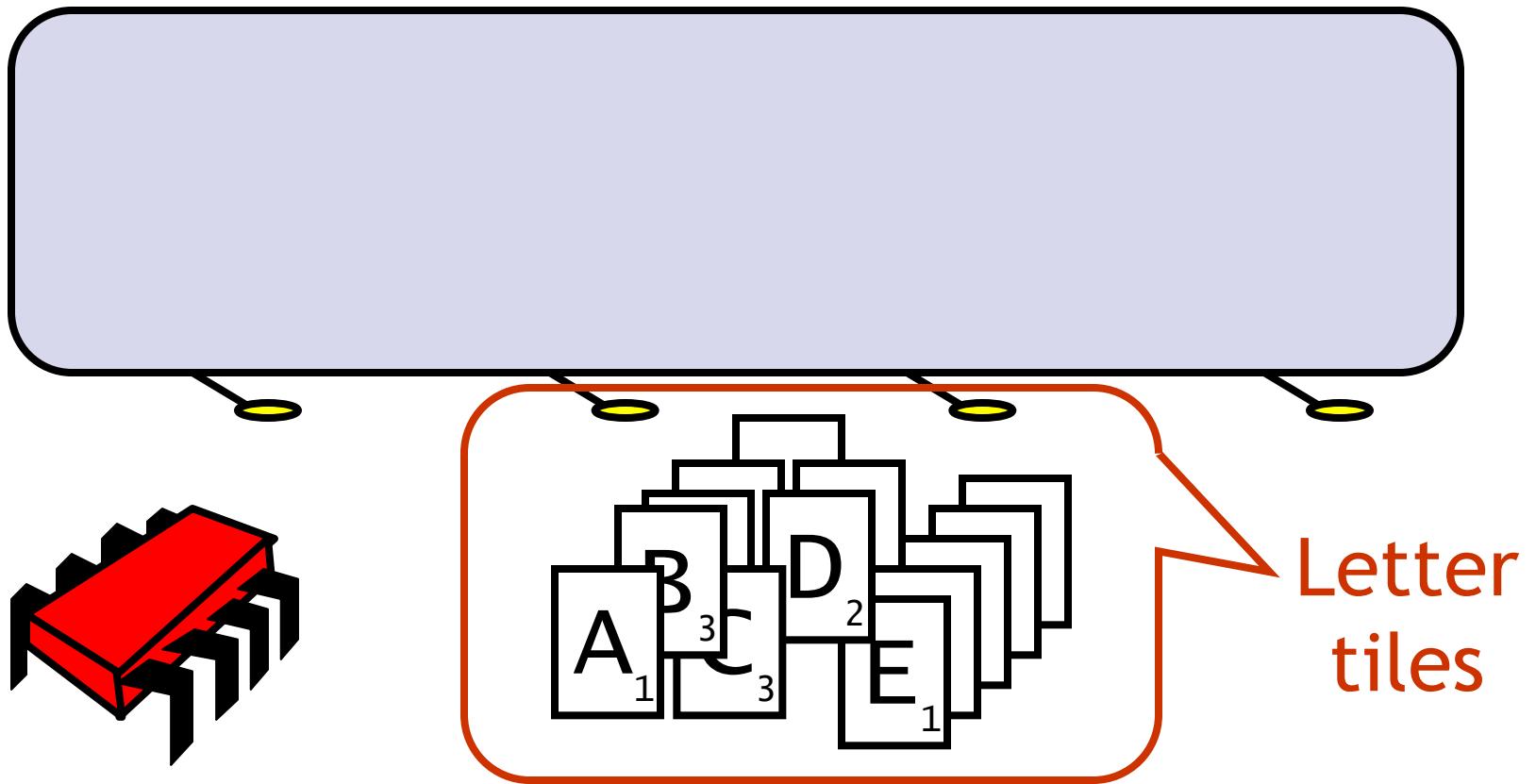
```
while (mumble) { }
```

- Waiting is *problematic*
 - If one participant is delayed
 - So is everyone else
 - But delays are common and unpredictable

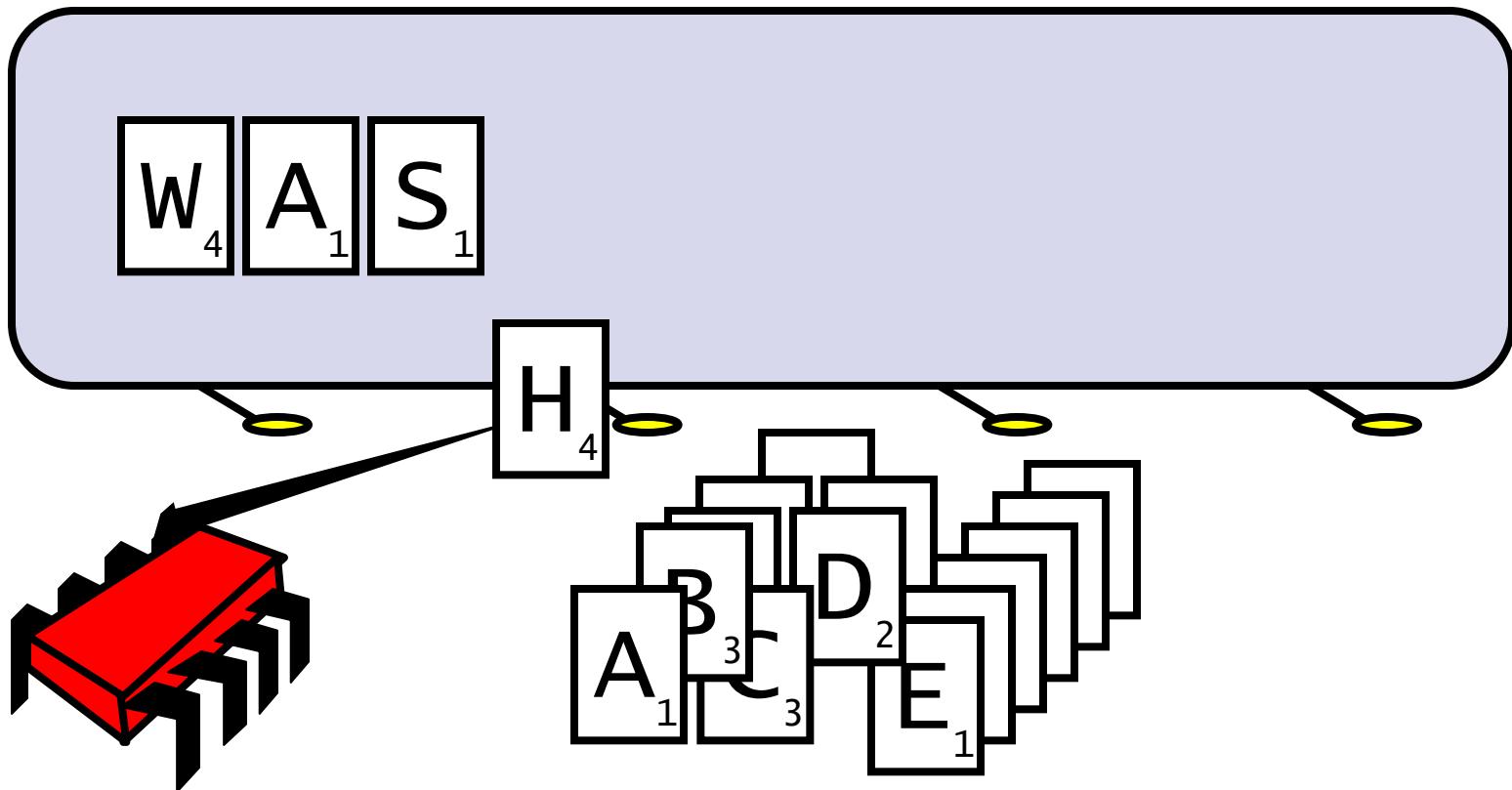
The Fable Drags on...

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

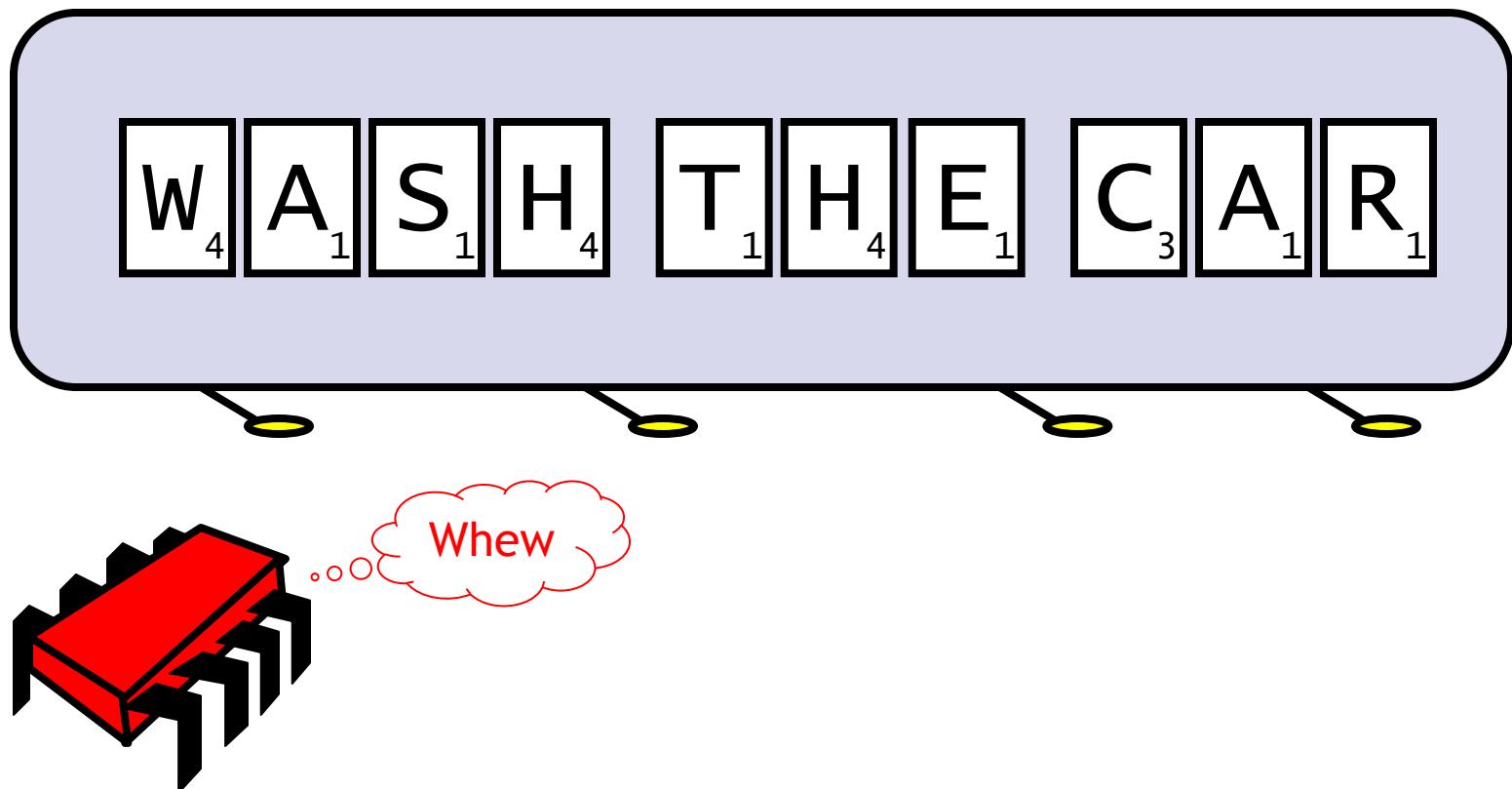
Billboards are Large



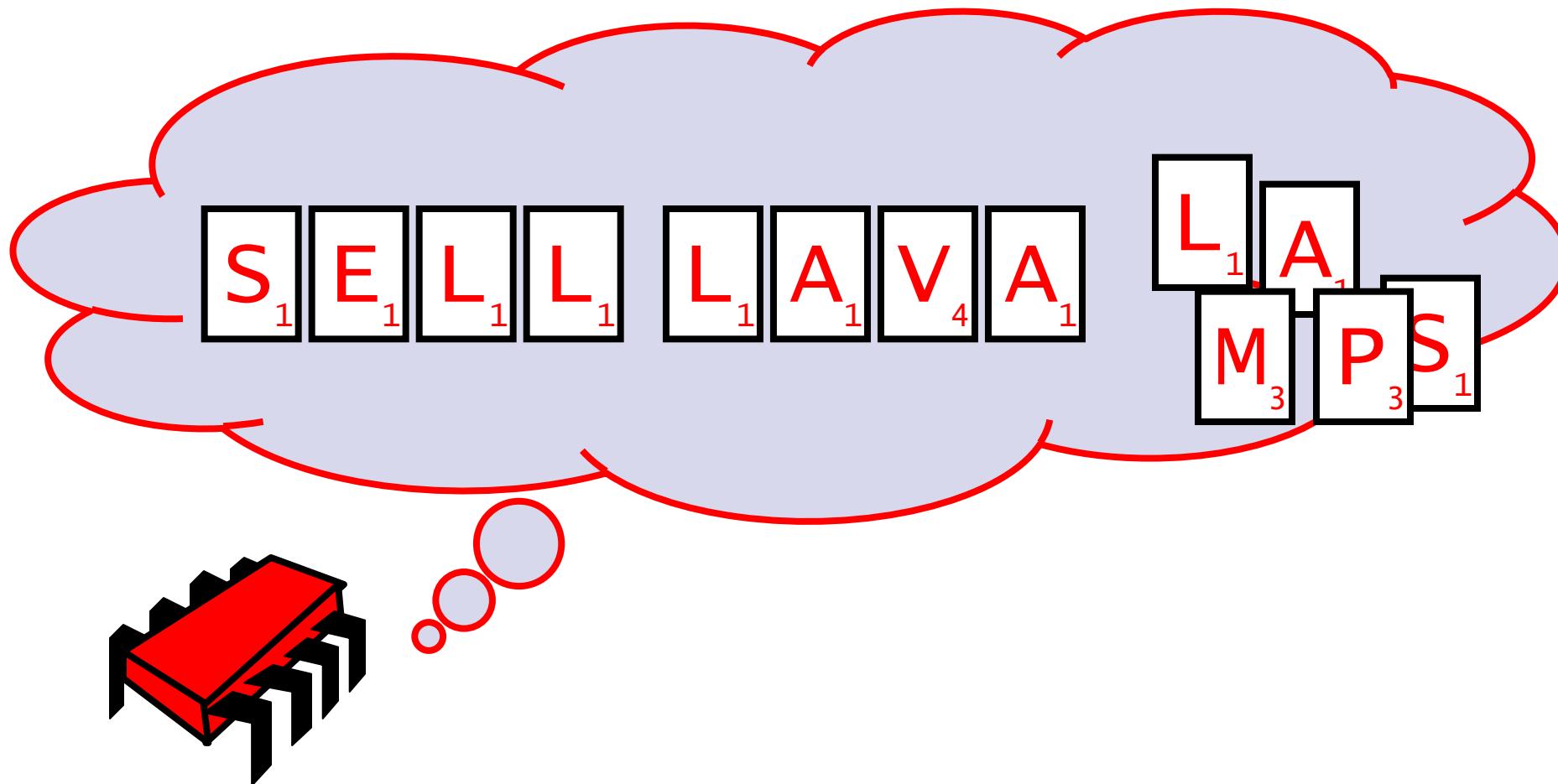
Write One Letter at a Time...



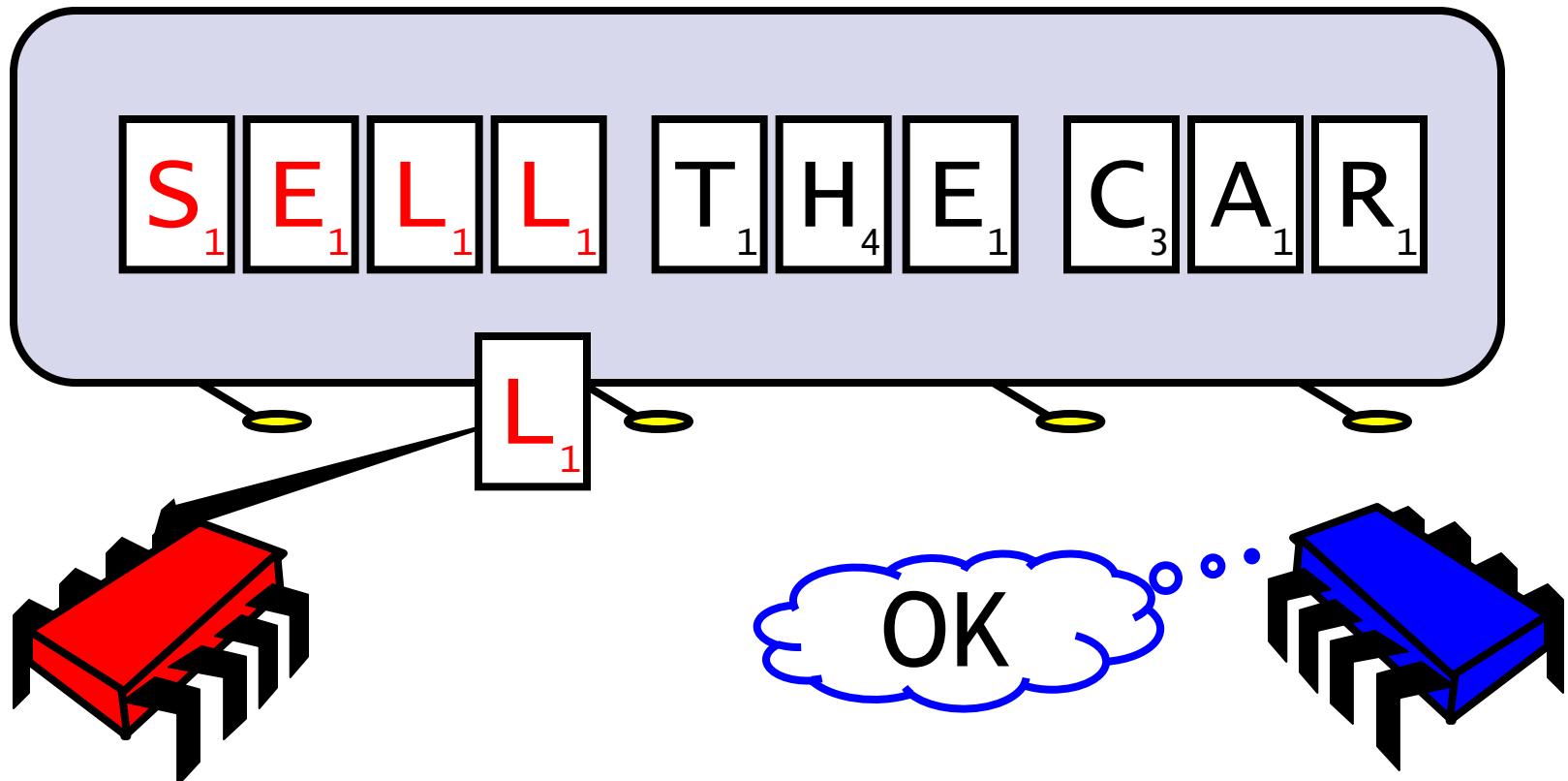
To Post a Message



Let's Send Another Message



Uh-Oh



Readers/Writers

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

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...

Amdahl's Law

$$\text{Speedup} = \frac{\text{OldExecutionTime}}{\text{NewExecutionTime}}$$

...of computation given n CPUs instead of 1

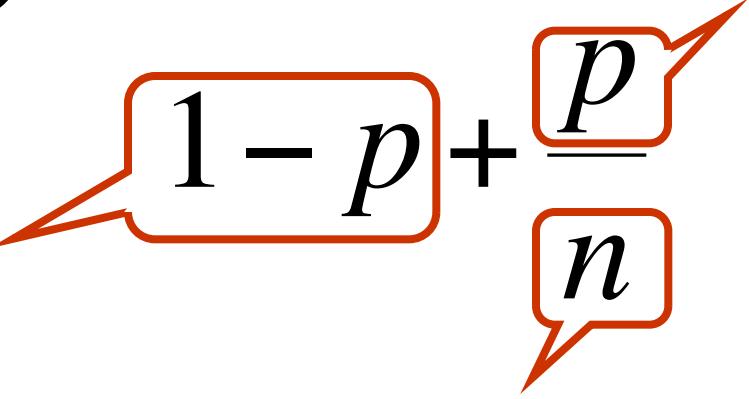
Amdahl's Law

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

Parallel fraction

Sequential fraction

Number of processors



The diagram illustrates the components of the Amdahl's Law formula. The formula is shown as a fraction: 1 divided by a sum. The first term in the sum is '1 - p', which is enclosed in a red speech bubble with an arrow pointing to it from the text 'Sequential fraction'. The second term is '+ p / n', where 'p' is in a red box with an arrow pointing to it from the text 'Parallel fraction', and 'n' is in another red box with an arrow pointing to it from the text 'Number of processors'.

Example

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

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

Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$Speedup = \frac{1}{1 - 0.8 + \frac{0.8}{10}} = 3.57$$

Example

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

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

Example

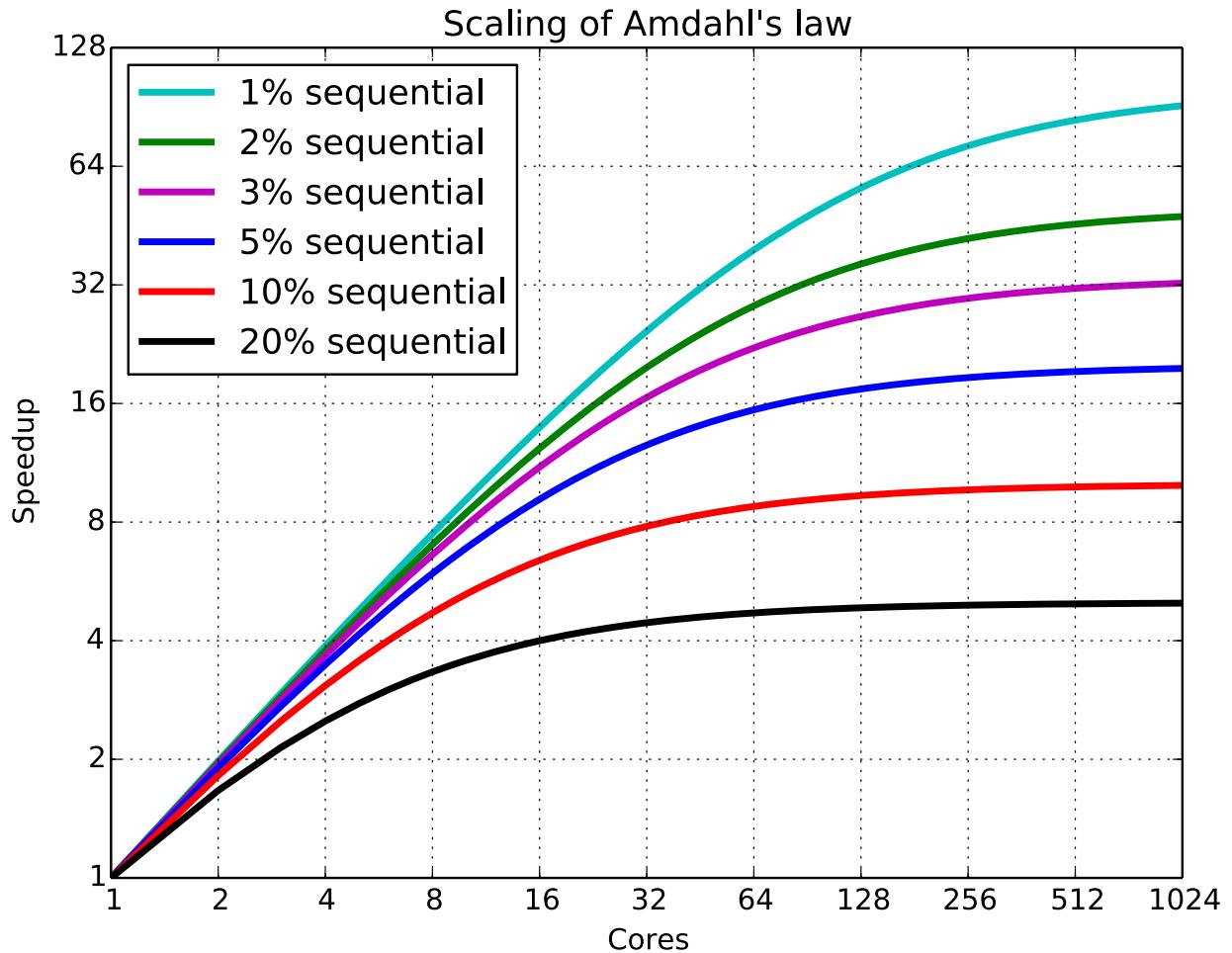
- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?

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

Scaling of Amdahl's Law

Amdahl's law

Relationship between the parallel part of a program and the speedup is not linear!



The Moral

- Making good use of our multiple processors (cores) means
- Finding ways to effectively parallelize our code
 - Minimize sequential parts
 - Reduce time in which threads **wait idle**
- This is what this course is about...
 - The % that is not easy to make concurrent yet may have a large impact on overall speedup