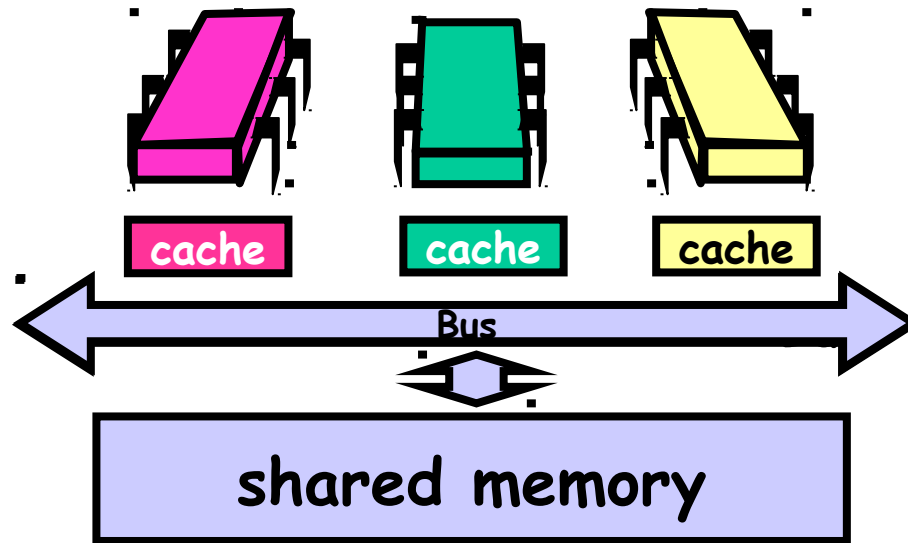


Introduction

Companion slides for
The Art of Multiprocessor
Programming
by Maurice Herlihy & Nir Shavit

(Abridged version. Original at <http://booksite.elsevier.com/9780123705914/?ISBN=9780123705914>)

In the Enterprise: The Shared Memory Multiprocessor (SMP)



Model Summary

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

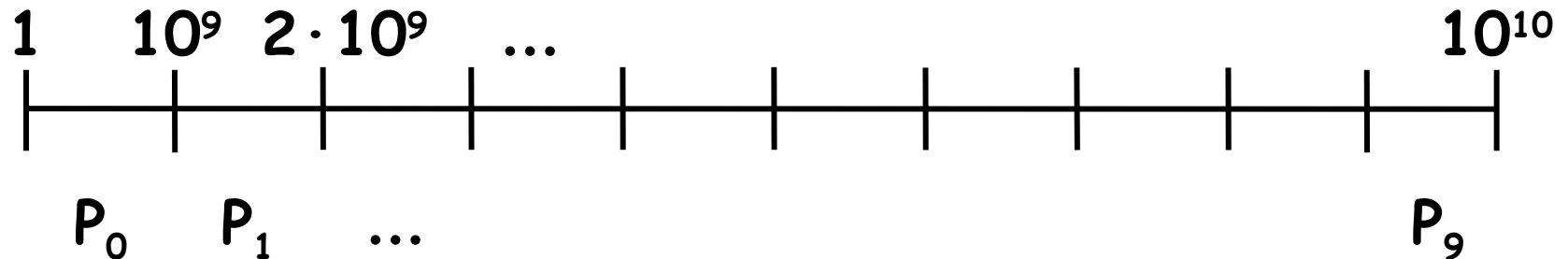
Concurrency Jargon

- Hardware
 - Processors
- Software
 - Threads, processes
- Sometimes OK to confuse them, 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

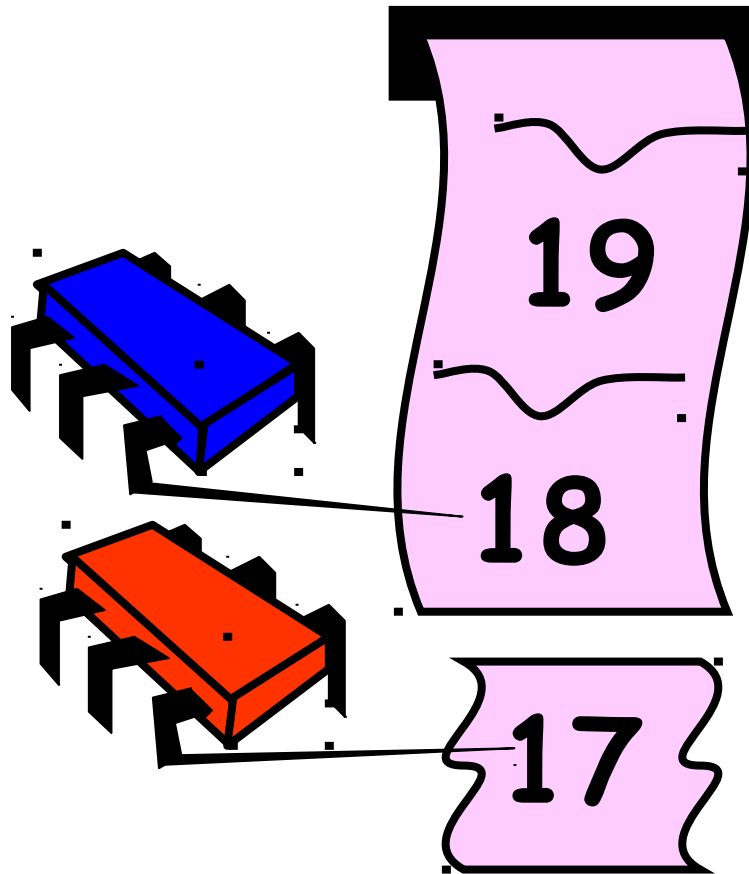
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
 - Uneven
 - Hard to predict

Issues

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

rejected

Shared Counter



each thread
takes a
number

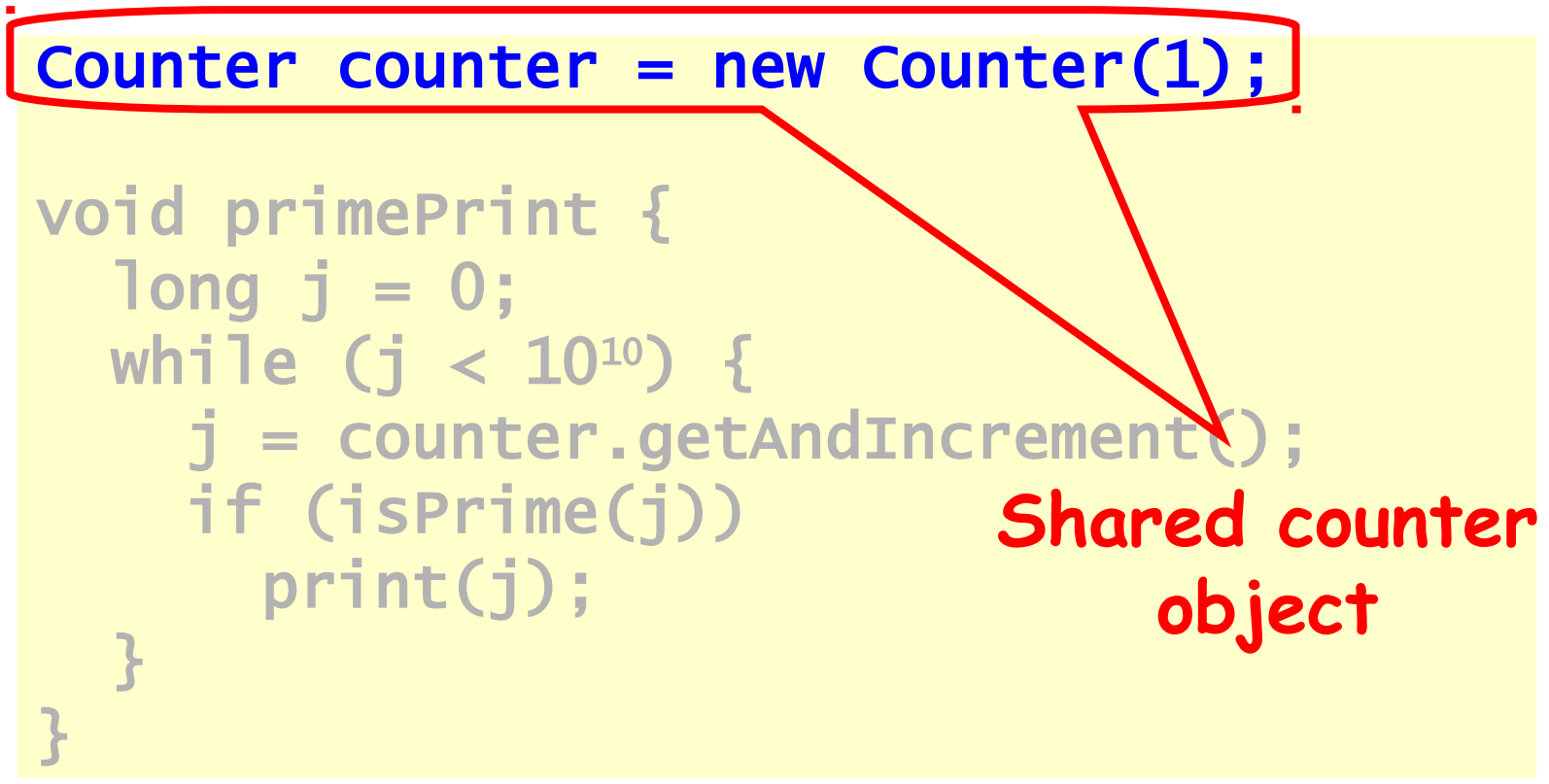
Procedure for Thread *i*

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

void primePrint {
    long j = 0;
    while (j < 1010) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
```

Procedure for Thread *i*

Counter counter = new Counter(1);



```
void primePrint {  
    long j = 0;  
    while (j < 1010) {  
        j = counter.getAndIncrement();  
        if (isPrime(j))  
            print(j);  
    }  
}
```

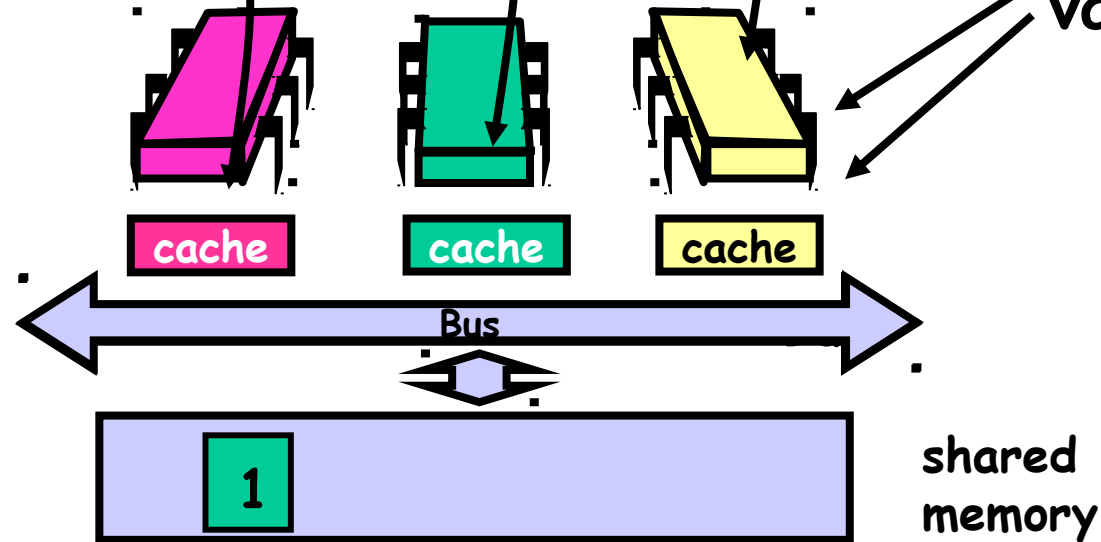
**Shared counter
object**

Where Things Reside

```
void primePrint {  
    int i =  
    ThreadID.get(); // IDs  
    in {0..9}  
    for (j = i*10+1,  
        j<(i+1)*10; j++) {  
        if (isPrime(j))  
            print(j);  
    }  
}
```

code

Local
variables



shared counter

Procedure for Thread i

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

```
void primePrint {
```

```
    long j = 0;
```

```
    while (j <  $10^{10}$ ) {
```

```
        j = counter.getAndIncrement();
```

```
        if (isPrime(j))
```

```
            print(j);
```

```
    }
```

```
}
```

Stop when every
value taken

Procedure for Thread *i*

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

```
void primePrint {
```

```
    long j = 0;
```

```
    while (j < 1010) {
```

```
        j = counter.getAndIncrement();
```

```
        if (isPrime(j))
```

```
            print(j);
```

```
    }
```

```
}
```

Increment &
return each new
value

Counter Implementation

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


Counter Implementation

```
public class Counter {  
    private long value;
```

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

```
}
```

OK for single thread,
not for concurrent threads

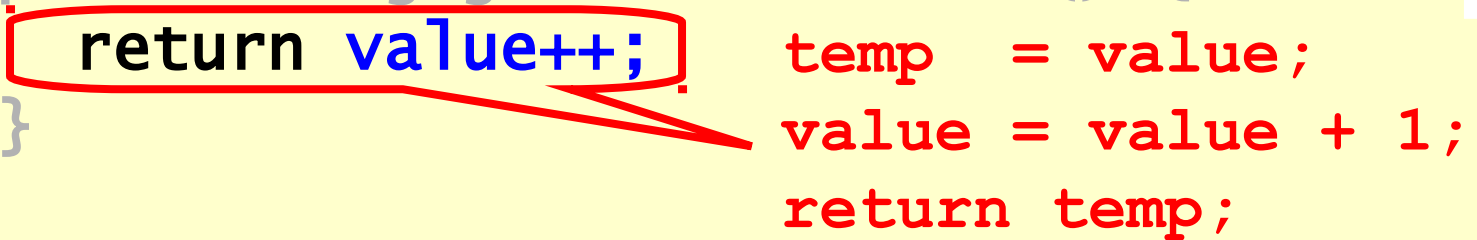
What It Means

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

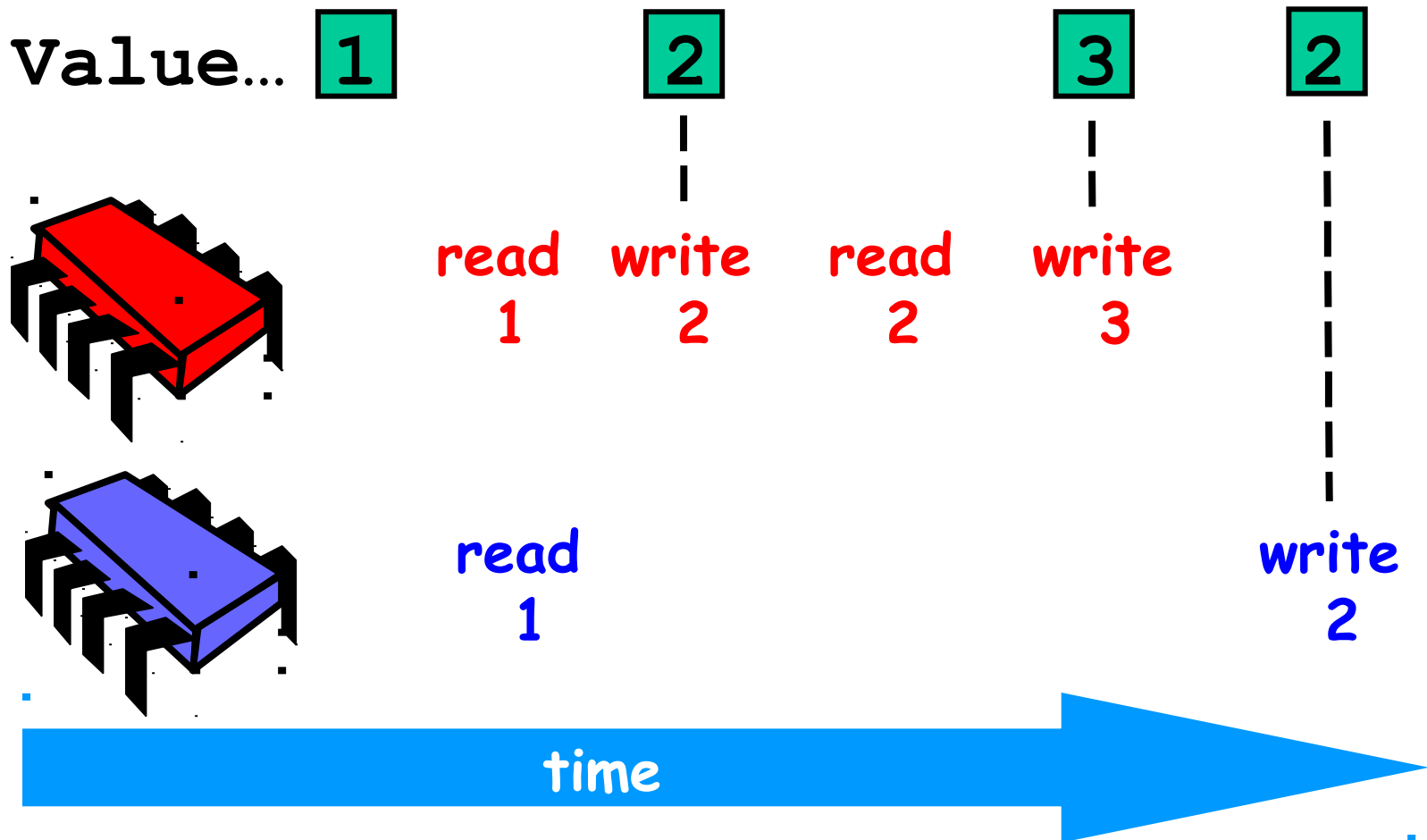
What It Means

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

*temp = value;
value = value + 1;
return temp;*



Not so good...



Challenge

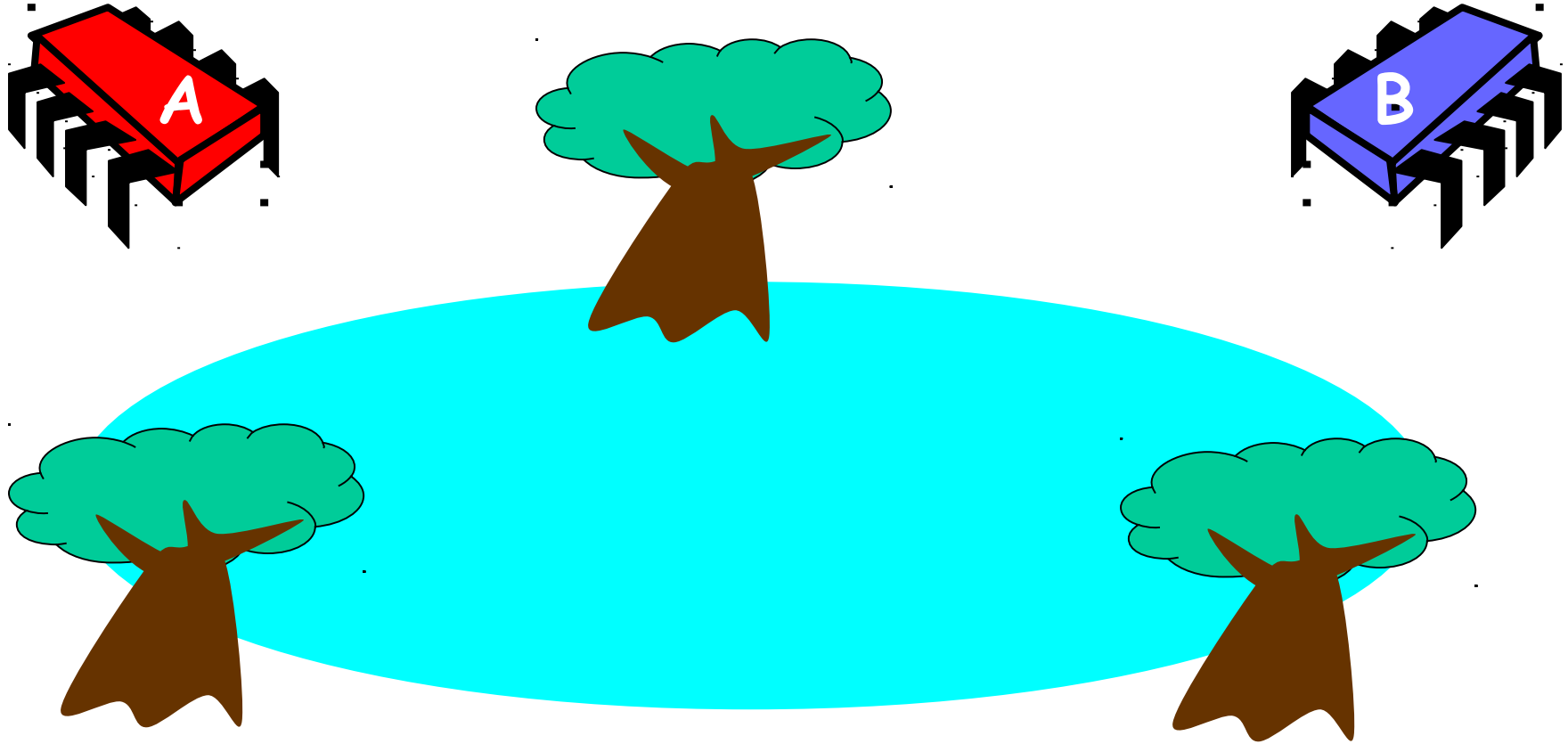
```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

Challenge

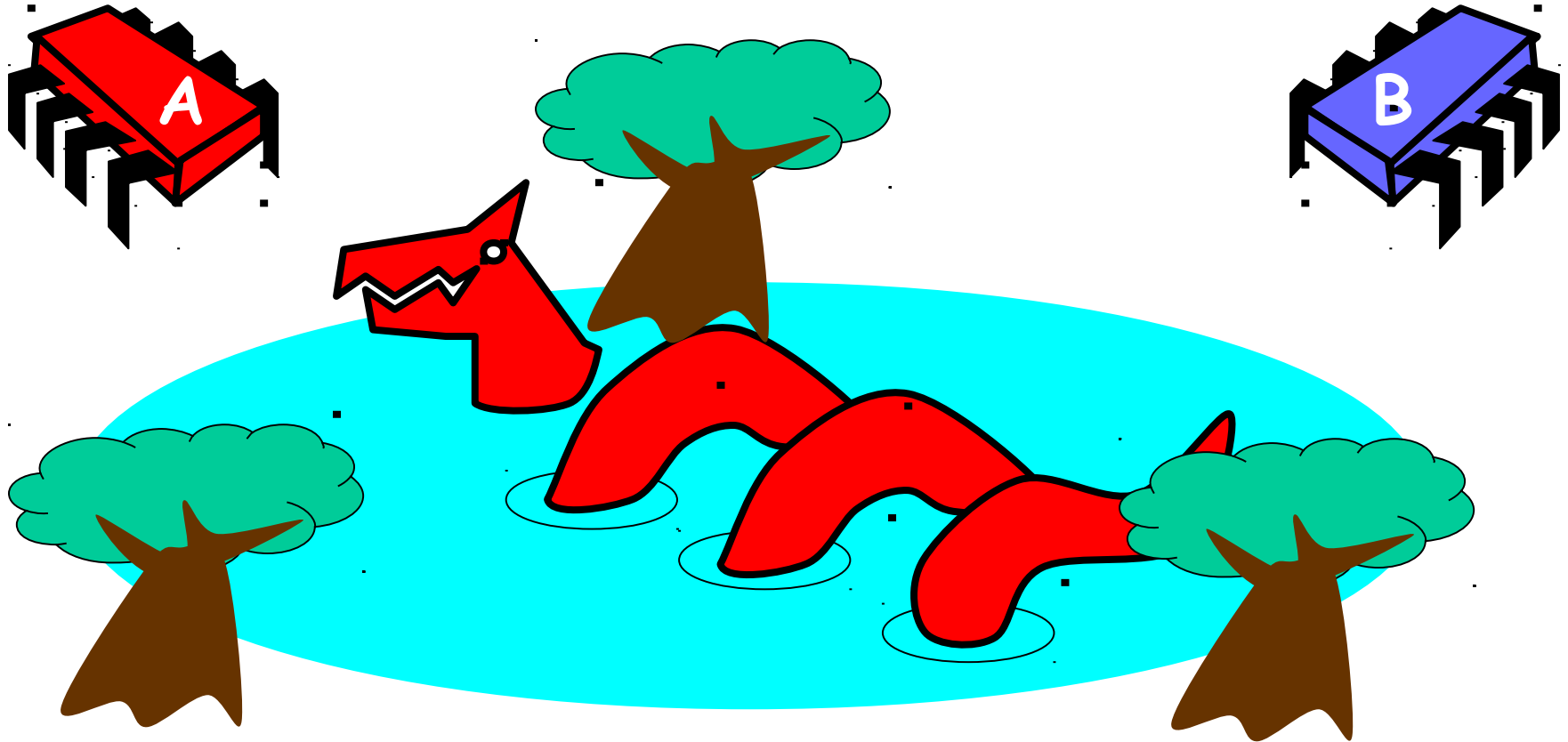
```
public class Counter {  
    private long value;  
  
    public long getAndIncrement() {  
        temp = value;  
        value = temp + 1;  
        return temp;  
    }  
}
```

**Make these steps
atomic (indivisible)**

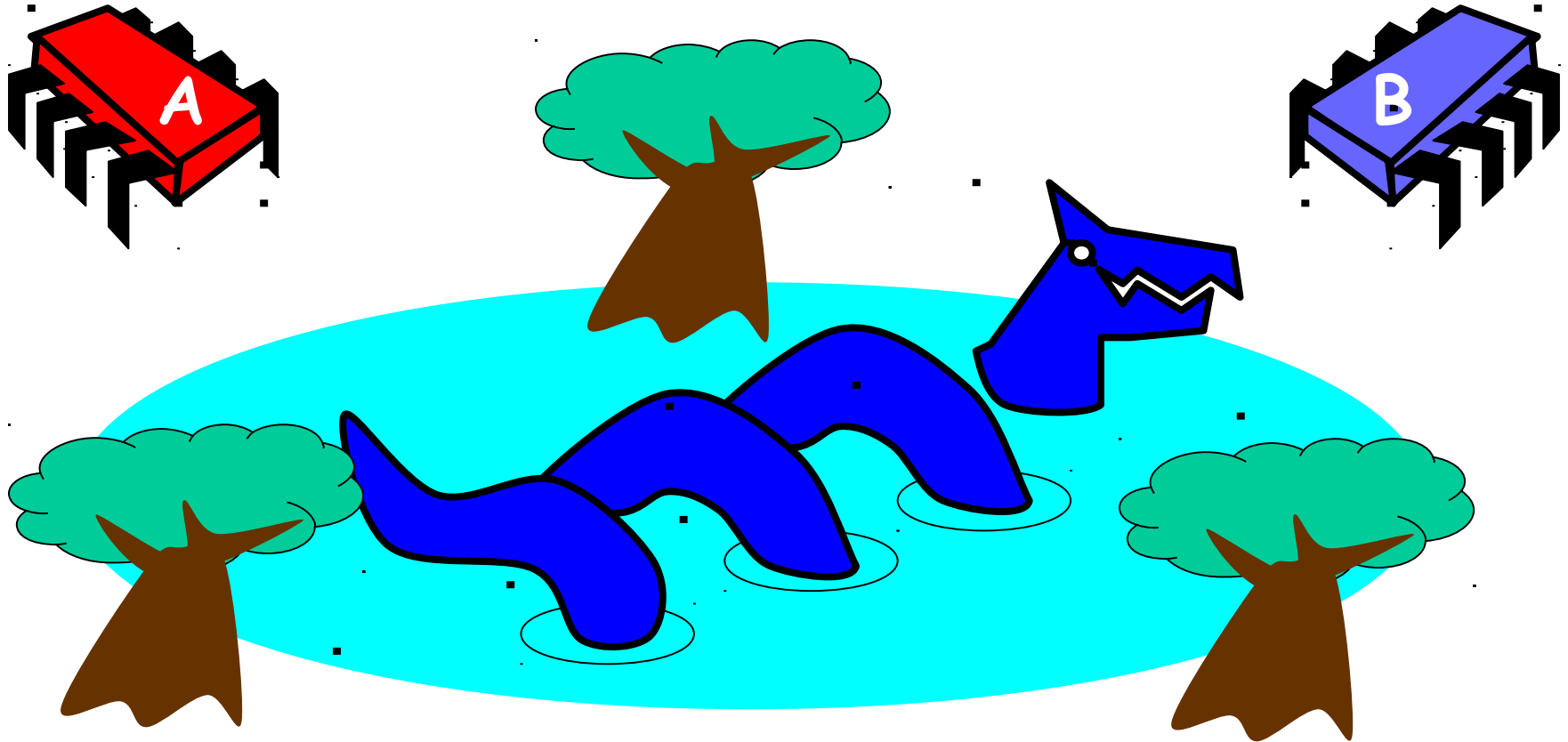
Mutual Exclusion or "Alice & Bob share a pond"



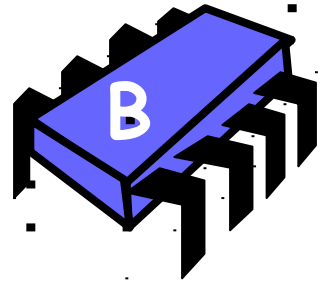
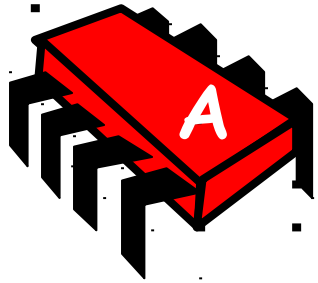
Alice has a pet



Bob has a pet



The Problem



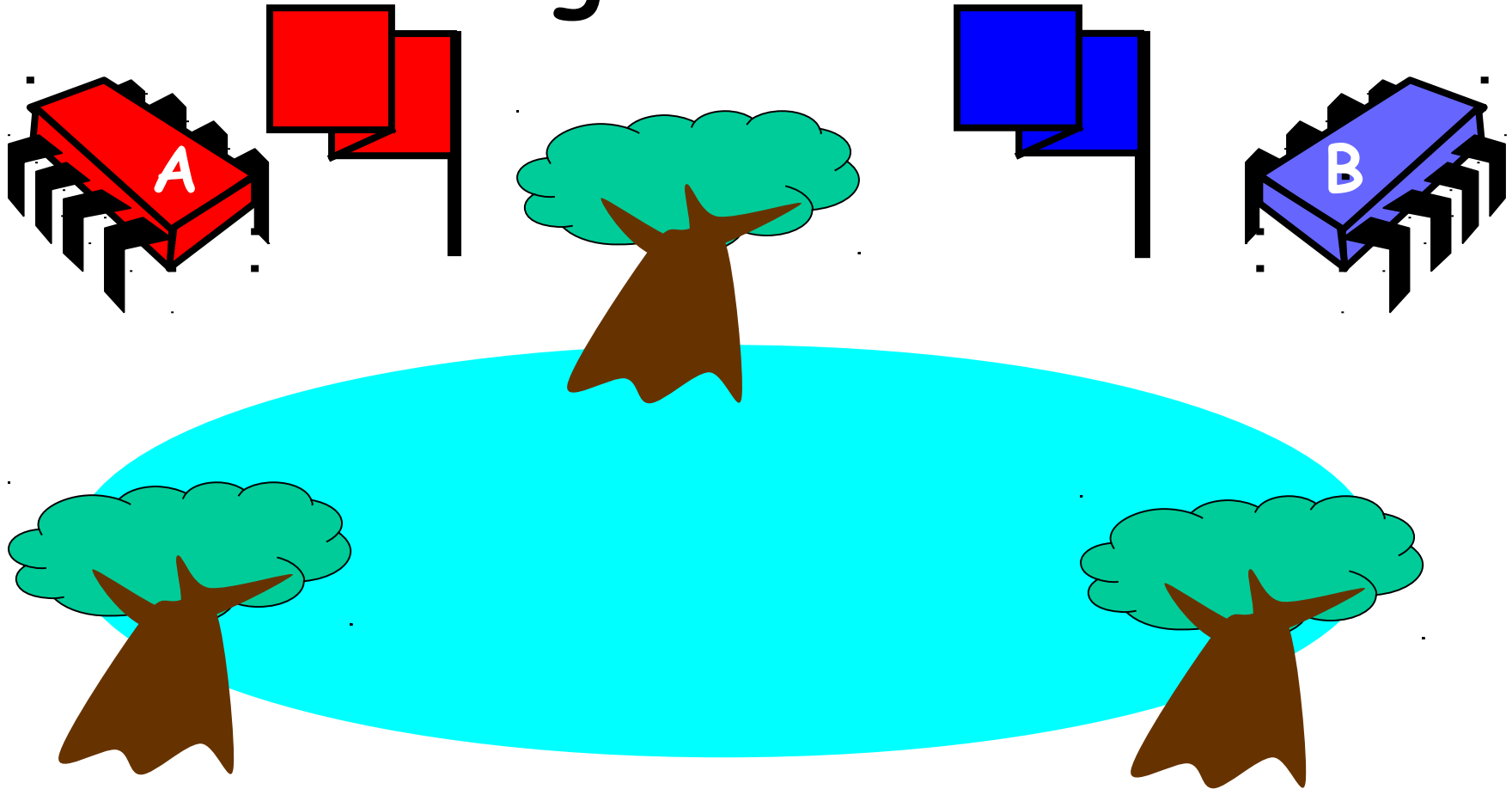
Formalizing the Problem

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

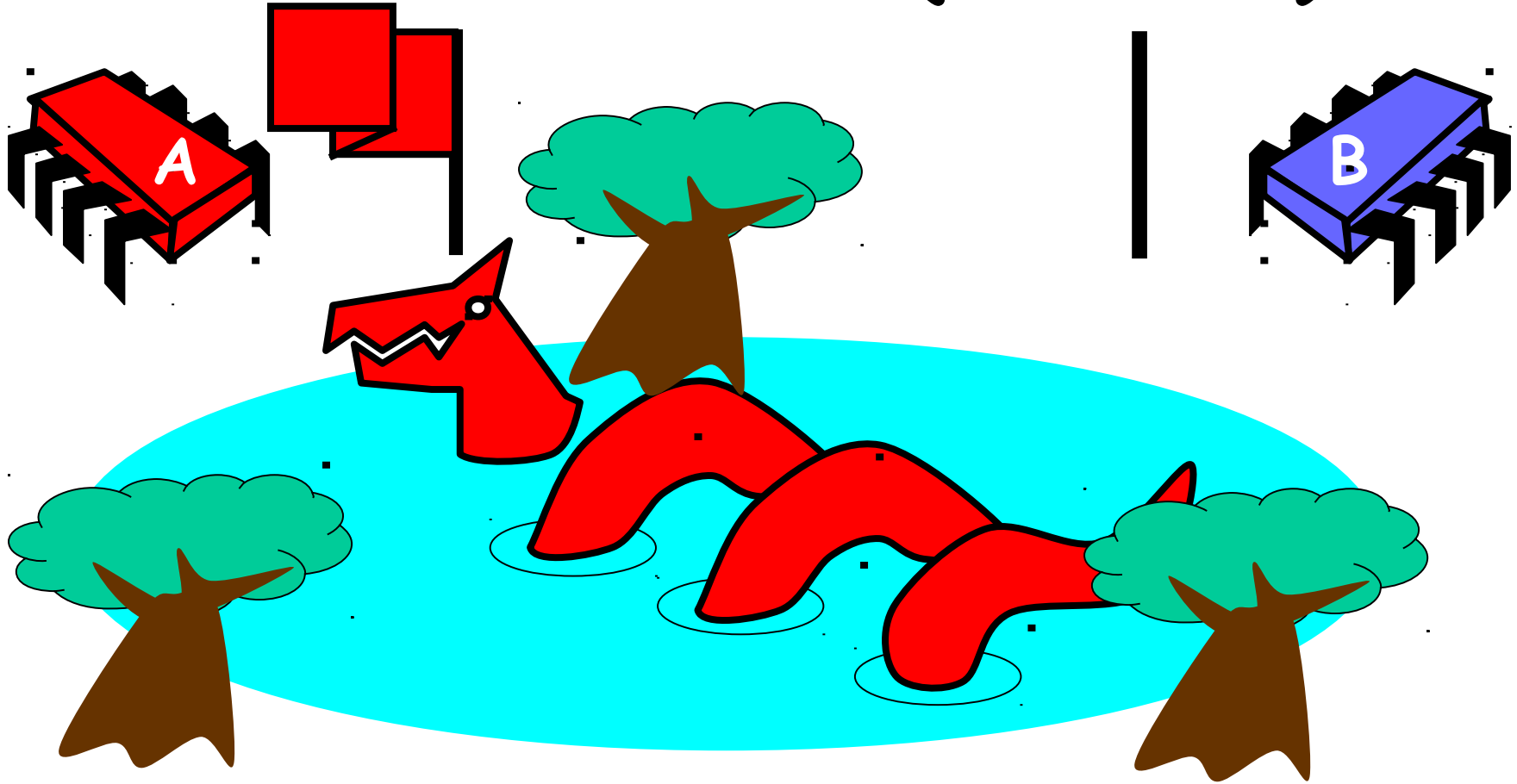
Formalizing our Problem

- Mutual Exclusion
 - Both pets never in pond simultaneously
 - This is a *safety* property
- No Deadlock
 - if only one wants in, it gets in
 - if both want in, one gets in.
 - This is a *liveness* property

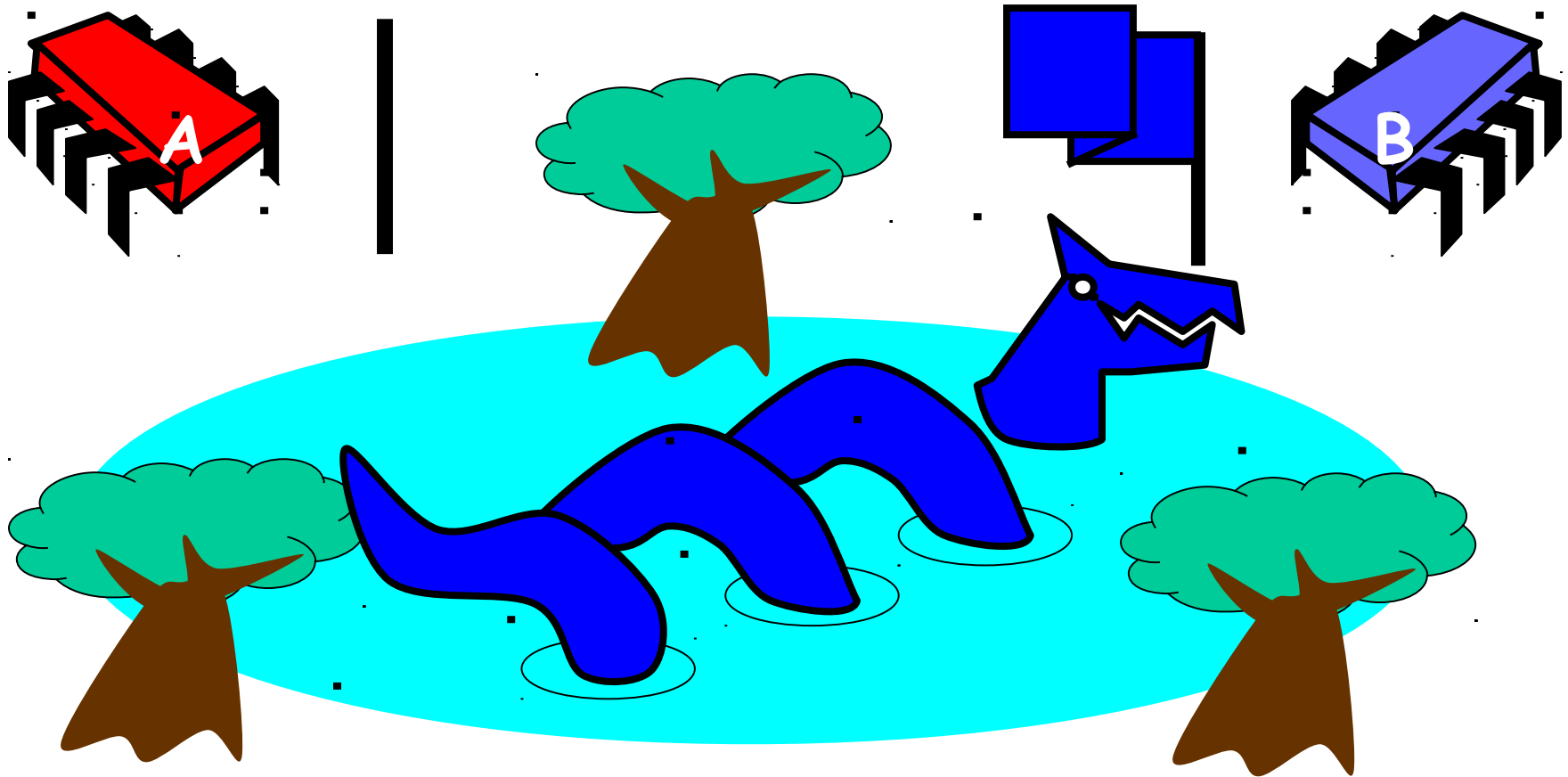
Flag Protocol



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

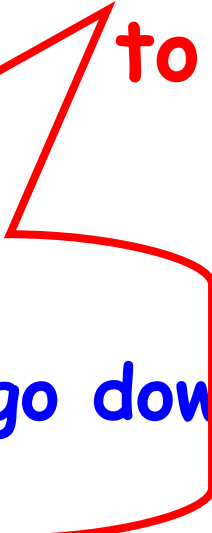
- Raise flag
- Wait until Alice's flag is down
- Unleash pet
- Lower flag when pet returns



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'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
- 

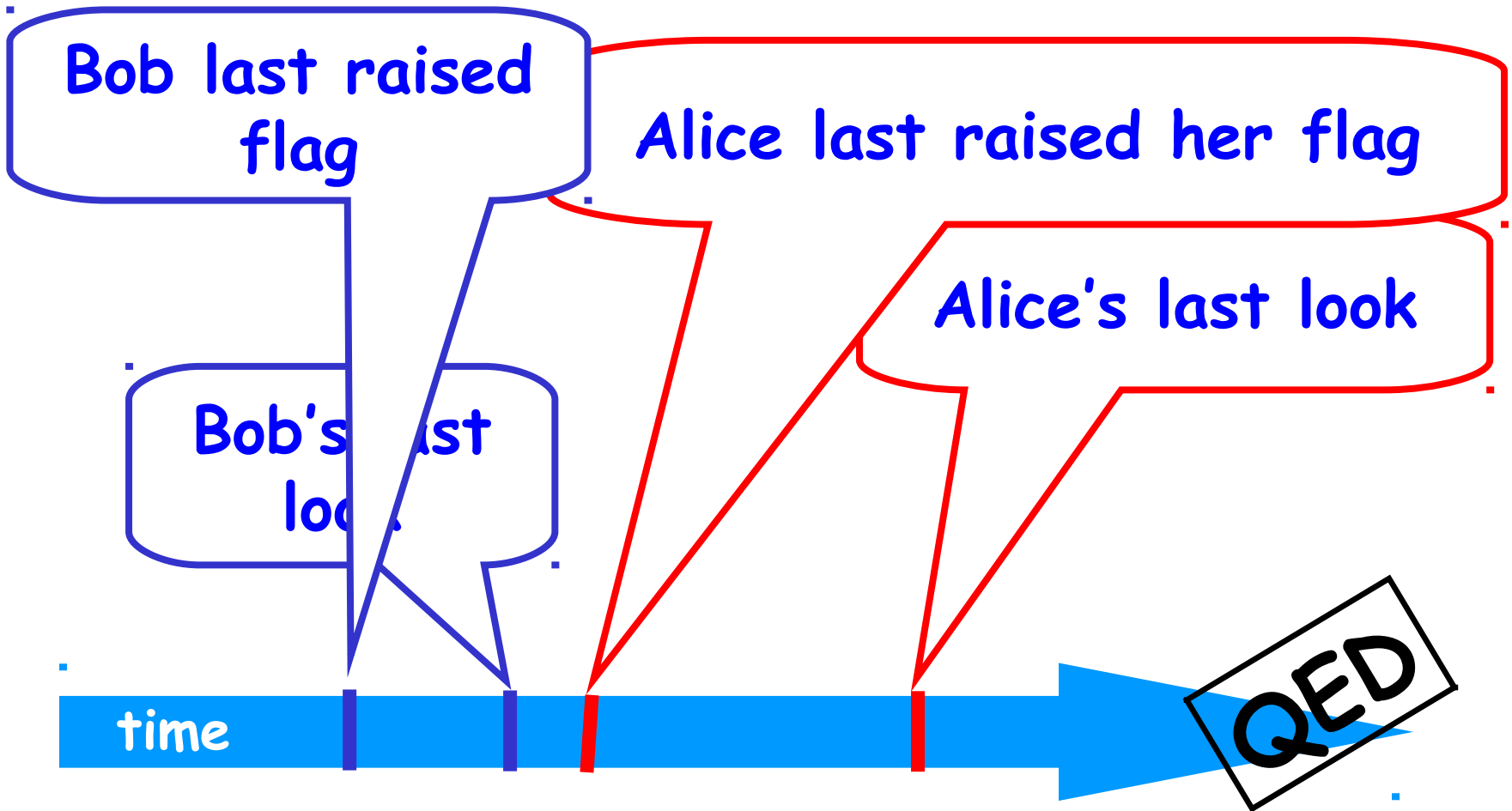
The Flag Principle

- 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 gives her priority (a gentleman...)



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

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.