

Introduction to Concurrency

CS 475, Spring 2018
Concurrent & Distributed Systems

Today

- Distributed & Concurrent Systems: high level overview and key concepts
- Relevant links:
 - Syllabus: <http://www.jonbell.net/gmu-cs-475-spring-2019/>
 - Piazza: <https://piazza.com/class/jqzcb36wlqz249>

Course Topics

- This course will teach you **how** and **why** to build distributed systems
- Distributed System is “a collection of independent computers that appears to its users as a single coherent system”
- This course will give you theoretical knowledge of the tradeoffs that you’ll face when building distributed systems

Course Staff

- Prof Jonathan Bell (me)
 - Office hour: ENGR 4422 Mon & Weds 1:30-2:15 pm or by appointment
 - Areas of research: Software Engineering, Program Analysis, Software Systems

Two hobbies: cycling, ice cream



Course Staff

- GTA: Abhijeet Mishra
 - Office Hours: TBA
- Please, **no emails** to instructor or TAs about the class: use Piazza

Grading

- 55% Homework
 - 4 assignments + final project, ~2 weeks to do each, all done individually
 - Your code will be autograded; you can resubmit and view your score
 - Also graded by hand for some non-functional issues
- 10% Checkpoint quizzes
 - Pass/fail (Pass if you are in class and submit a quiz, fail if you don't)
 - Use laptop or phone to complete the quiz in class (please write your name and answers on a piece of paper and bring to me after class if you lost/broke/etc your smart phone or laptop)
- 15% Midterm Exam, 20% Final Exam



But, seriously

- They may be unlike any assignments you have done so far
- By the end of the semester, you will have built a sizable and complicated, real, usable distributed system, using standard technologies like RMI and ZooKeeper
- Assignments are mostly out for 2 weeks: it will take 2 weeks to do the assignment
 - If you start the day before, there will not be enough hours in the day to complete the assignment
- Assignments are graded on functionality, with clear cut-offs for partial functionality. Focus on building incremental functionality (some, but very few points for trying to get everything and succeeding at nothing)
- First assignment out Monday

Policies

- My promises to you:
 - Quiz results will be available instantaneously in class; we will discuss quiz in real time
 - Homework will be graded within 3 days of submission
 - Exams will be graded within a week

Policies

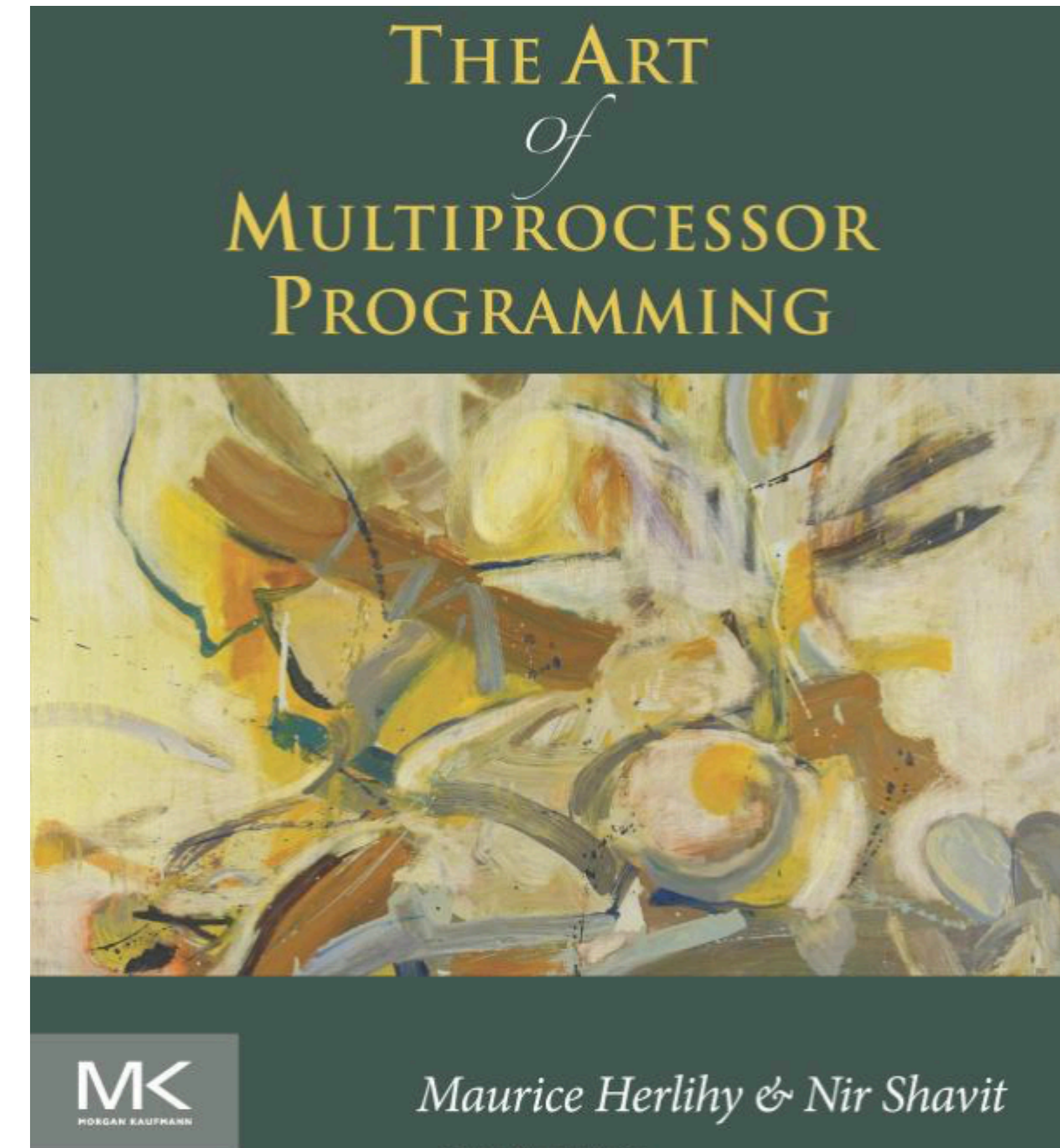
- Lateness on homework:
 - 10% penalty if submitted UP TO 24 hours after deadline
 - No assignments will be accepted more than 24 hours late
 - Out of fairness: **no exceptions**
- Attendance & Quizzes:
 - You can miss up to 3 with no penalty
 - Again, out of fairness: **no exceptions** beyond this

Honor Code

- Refresh yourself of the department honor code
- Homeworks are 100% individual
 - Discussing assignments at high level: ok, sharing code: not ok
 - If in doubt, ask the instructor
 - If you copy code, we WILL notice (see some of my recent research results in “code relatives”)
- Online activities/checkpoints/quizzes must be completed by you, and while in class
 - Nobody leaves the room until all responses are accounted

Readings

- Good news: new (to this class) book!
- The Art of Multiprocessor Programming, Herlihy and Shavit
- Also recommended as a reference (free): Distributed Systems 3rd Edition (van Steen and Tanenbaum)
<https://www.distributed-systems.net/index.php/books/distributed-systems-3rd-edition-2017/>

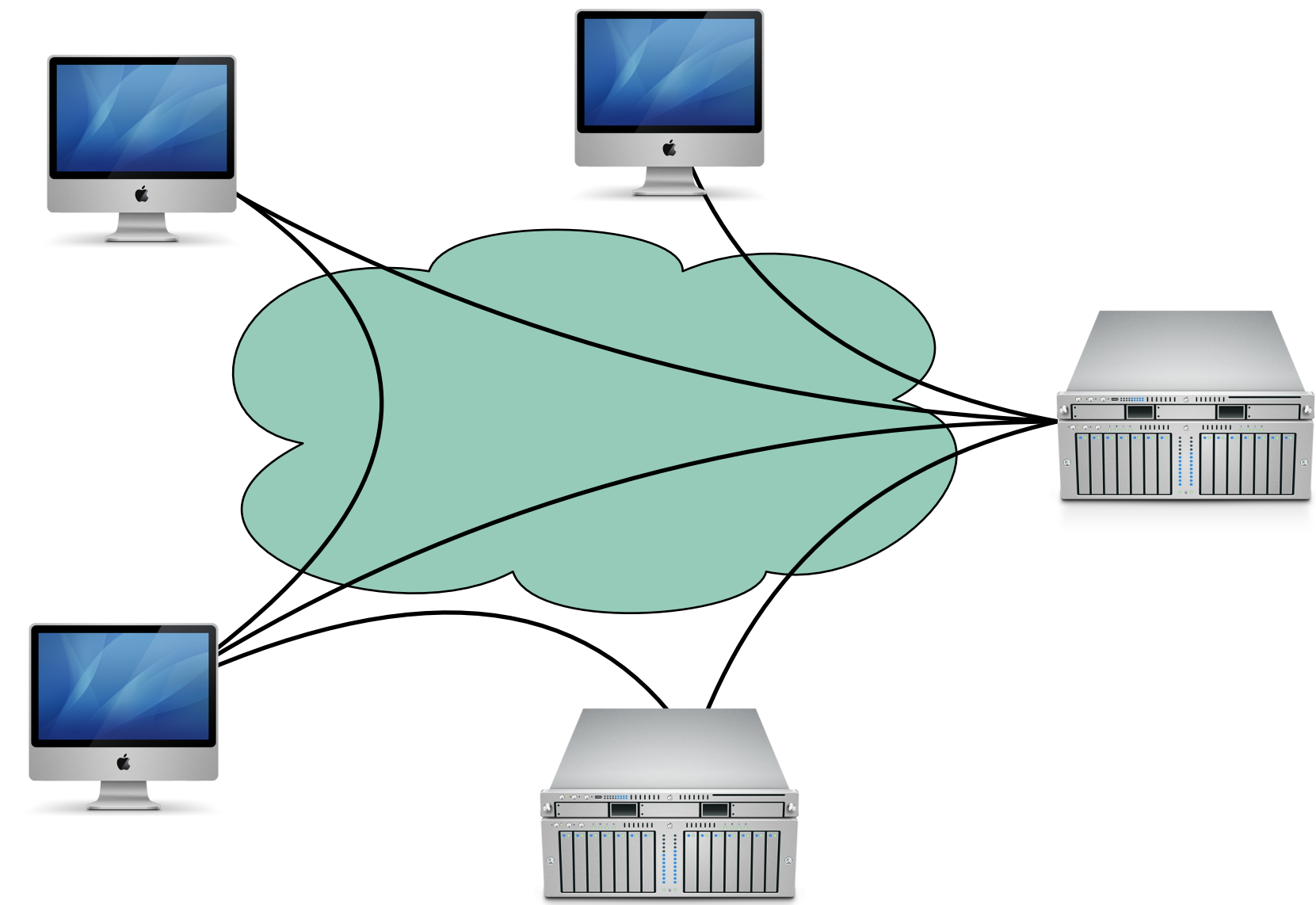


Course Topics



How do I run multiple things at once on my computer?

Concurrency, first half of course



How do I run a big task across many computers?

Distributed Systems, second half of course

Layers

- From hardware
- To OS
- To programming languages
- To networks
- To libraries and middleware
- To developers

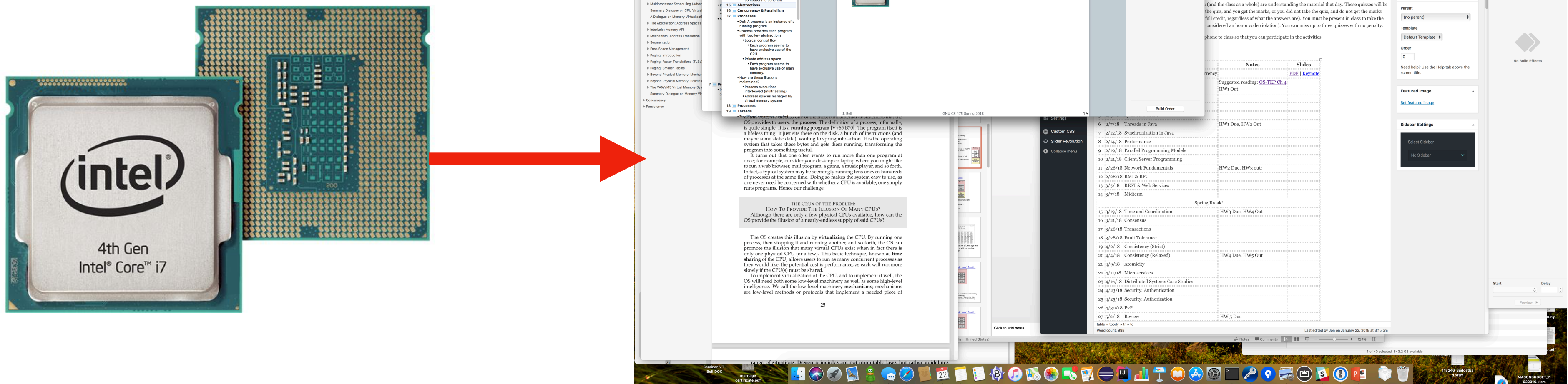
Concurrency

- Goal: do multiple things, at once, coordinated, on one computer
 - Update UI
 - Fetch data
 - Respond to network requests
 - Improve responsiveness, scalability
- Recurring problems:
 - Coordination: what is shared, when, and how?

Abstractions

- Goal: take something complicated, make it “easy”
- Operating Systems
 - From CPUs and memory to processes and threads
- Distributed Systems
 - From collections of computers to coherent applications

Abstractions



What are the abstractions that sit between the CPU and my multitasking operating system?

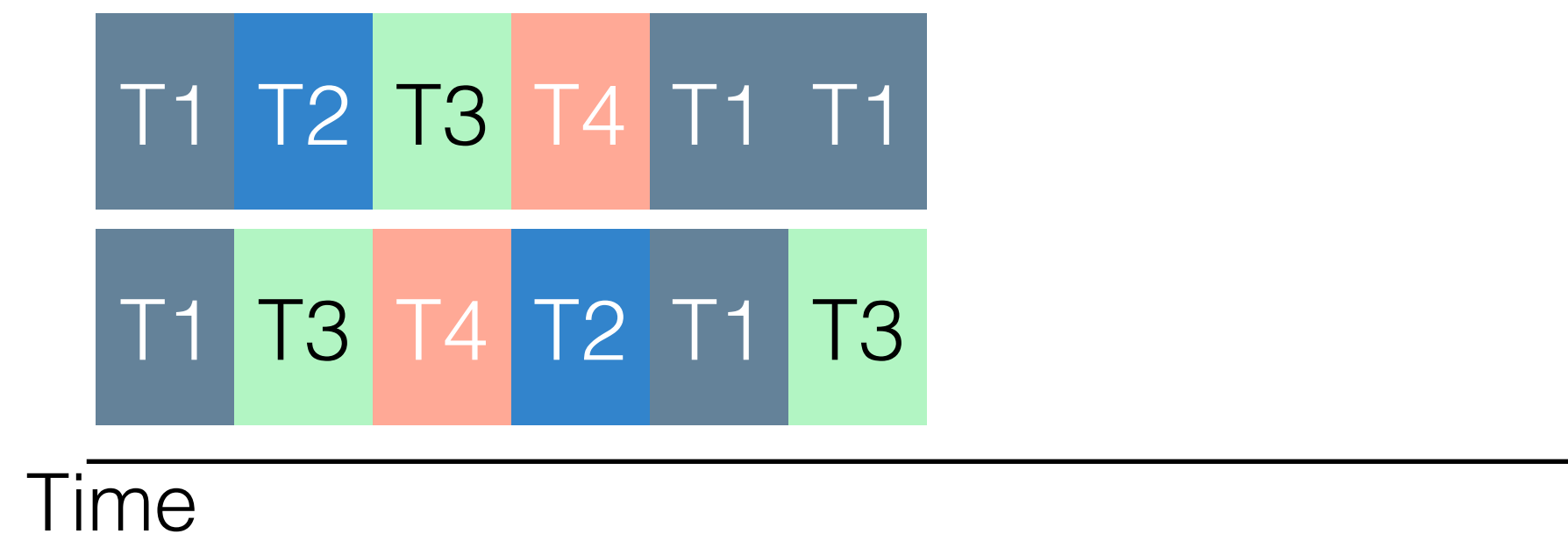
Concurrency & Parallelism

4 different things: T1 T2 T3 T4

Concurrency:
(1 processor)



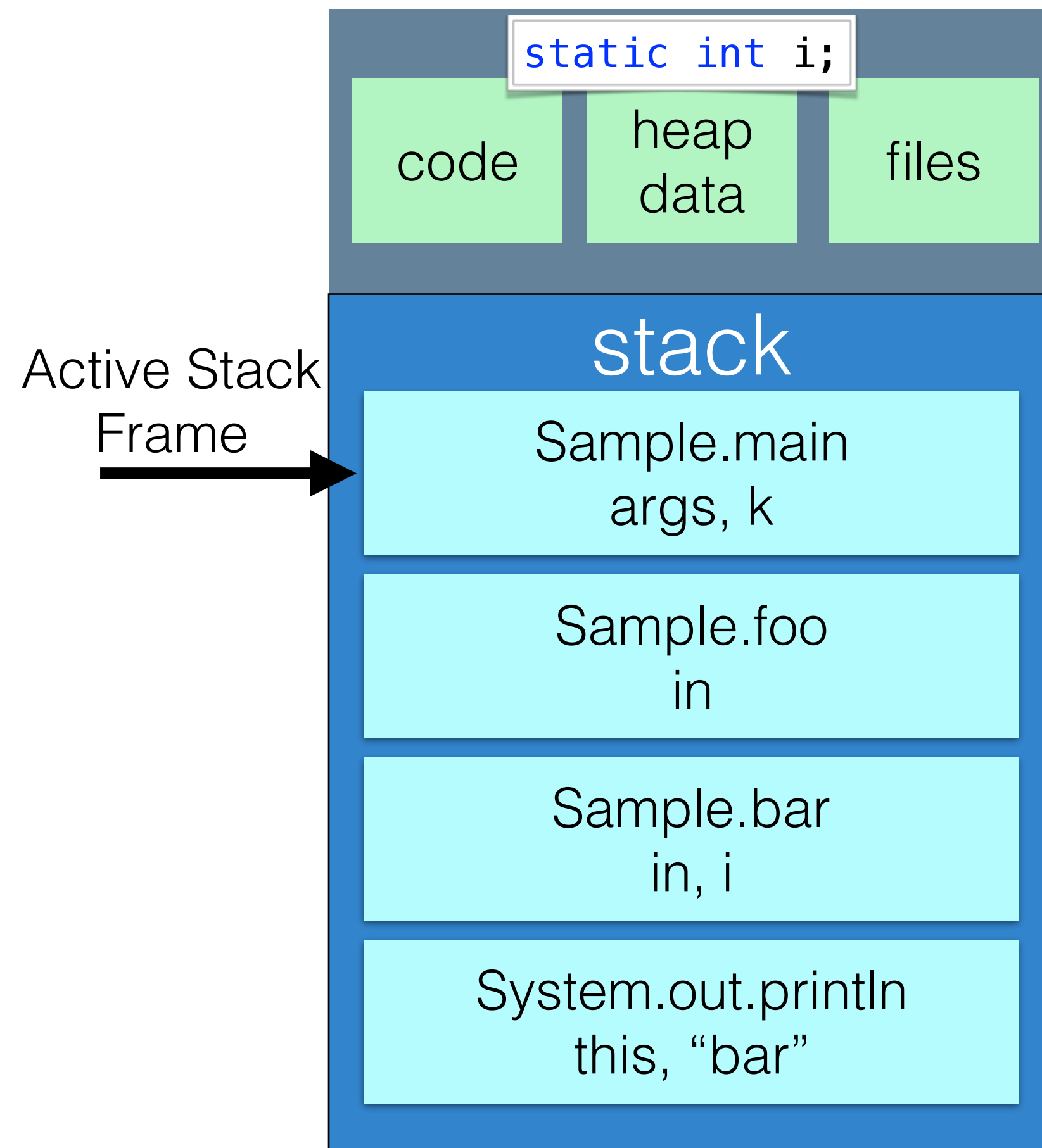
Parallelism:
(2 processors)



Processes

- Def: A process is an instance of a running program
- Process provides each program with two key abstractions
 - Logical control flow
 - Each program seems to have exclusive use of the CPU.
 - Private address space
 - Each program seems to have exclusive use of main memory.
- How are these illusions maintained?
 - Process executions interleaved (multitasking)
 - Address spaces managed by virtual memory system

Processes



```
public class Sample
{
    static int i;
    public static void main(String[] args)
    {
        int k = 10;
        foo(k);
    }
    public static void foo(int in)
    {
        bar(in);
    }
    public static void bar(int in)
    {
        i = in;
        System.out.println("bar");
    }
}
```

Threads

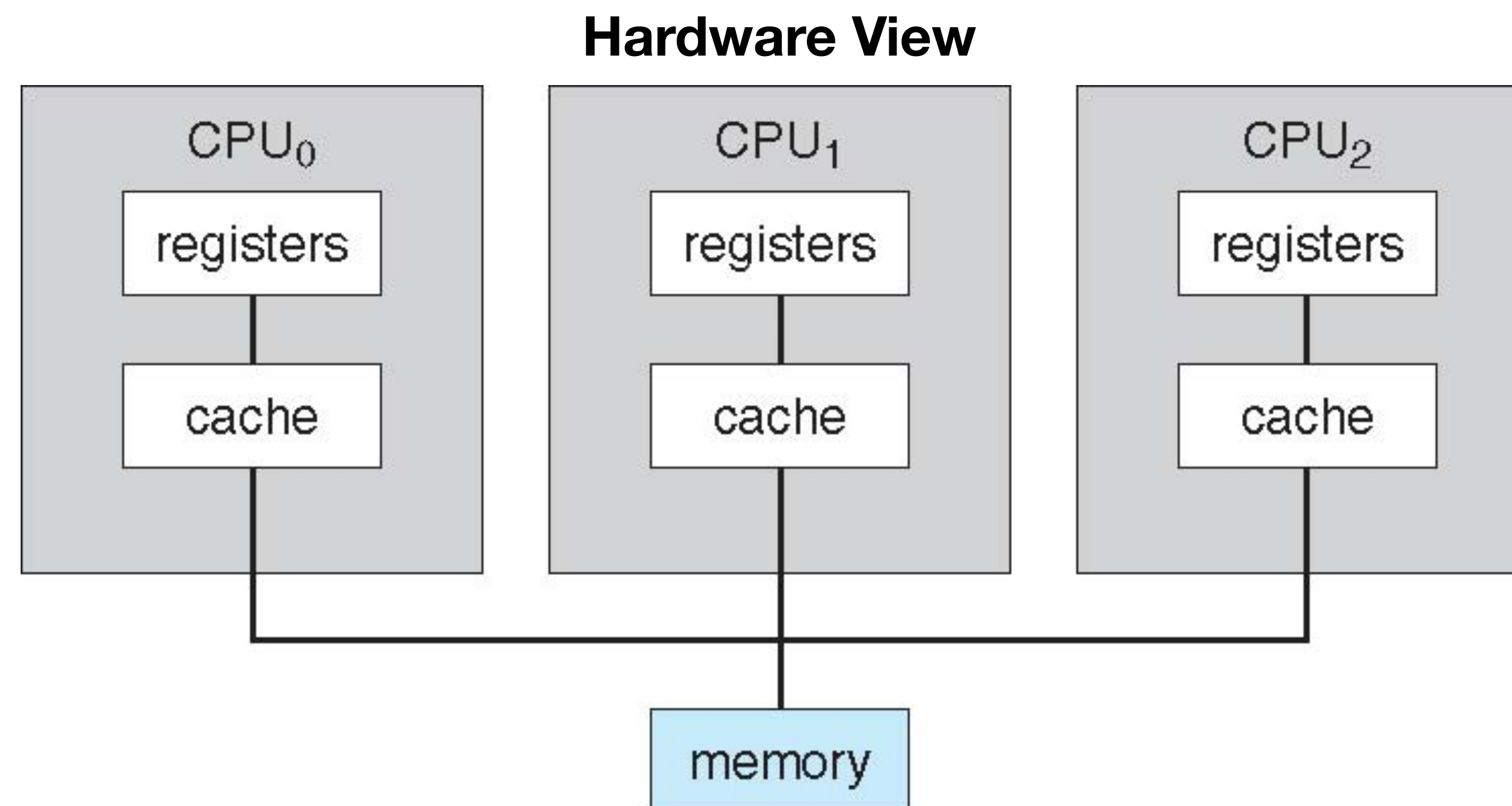
- Traditional processes created and managed by the OS kernel
- Process creation expensive - fork system call in UNIX
- Context switching expensive
- Cooperating processes - no need for memory protection (separate address spaces)

More Abstractions

- Process + Thread -> one computer
- How can we abstract many computers working together?
- What does that even look like?

Leaky Abstractions

- Completely hiding the underlying complexity is never possible, usually not desirable
- Example: our first two abstractions (concurrency) - process and thread



More machines, more problems

- Say there's a 1% chance of having some hardware failure occur to a machine (power supply burns out, hard disk crashes, etc)
- Now I have 10 machines
 - Probability(at least one fails) = $1 - \text{Probability}(\text{no machine fails}) = 1 - (1 - .01)^{10} = 10\%$
- 100 machines -> 63%
- 200 machines -> 87%
- So obviously just adding more machines doesn't solve fault tolerance

How much to hide?

- Completely hiding how distributed a system is may be too much:
 - Communication latencies can't be hidden (pesky speed of light!)
 - Completely hiding failures is **impossible** (we will prove this later in the semester)
 - Can never distinguish a slow computer from one that is crashed
- Hiding more adds performance costs

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”
- First principle (today): Mutual Exclusion

Online activity

Go to socrative.com and select “Student Login” (works well on laptop, tablet or phone)

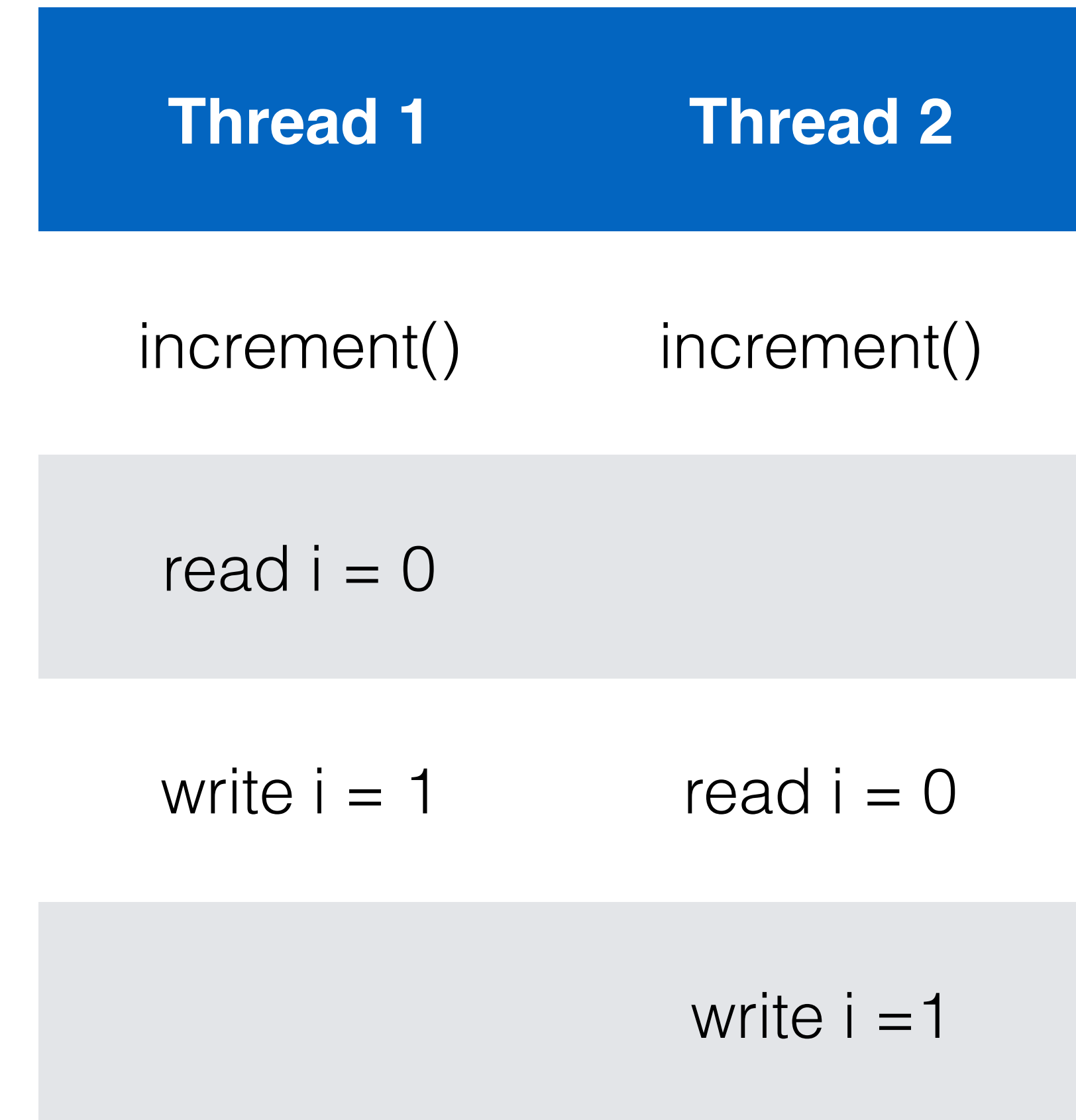
Room Name: CS475
ID is your [@gmu.edu](mailto:yourname@gmu.edu) email

Mutual Exclusion

If two threads run the same code (at once), what is the value of i at the end?

```
static int i = 0;
public static void increment()
{
    i = i + 1;
}
```

Is it guaranteed to be 2? No - it can also be 1 at the end!



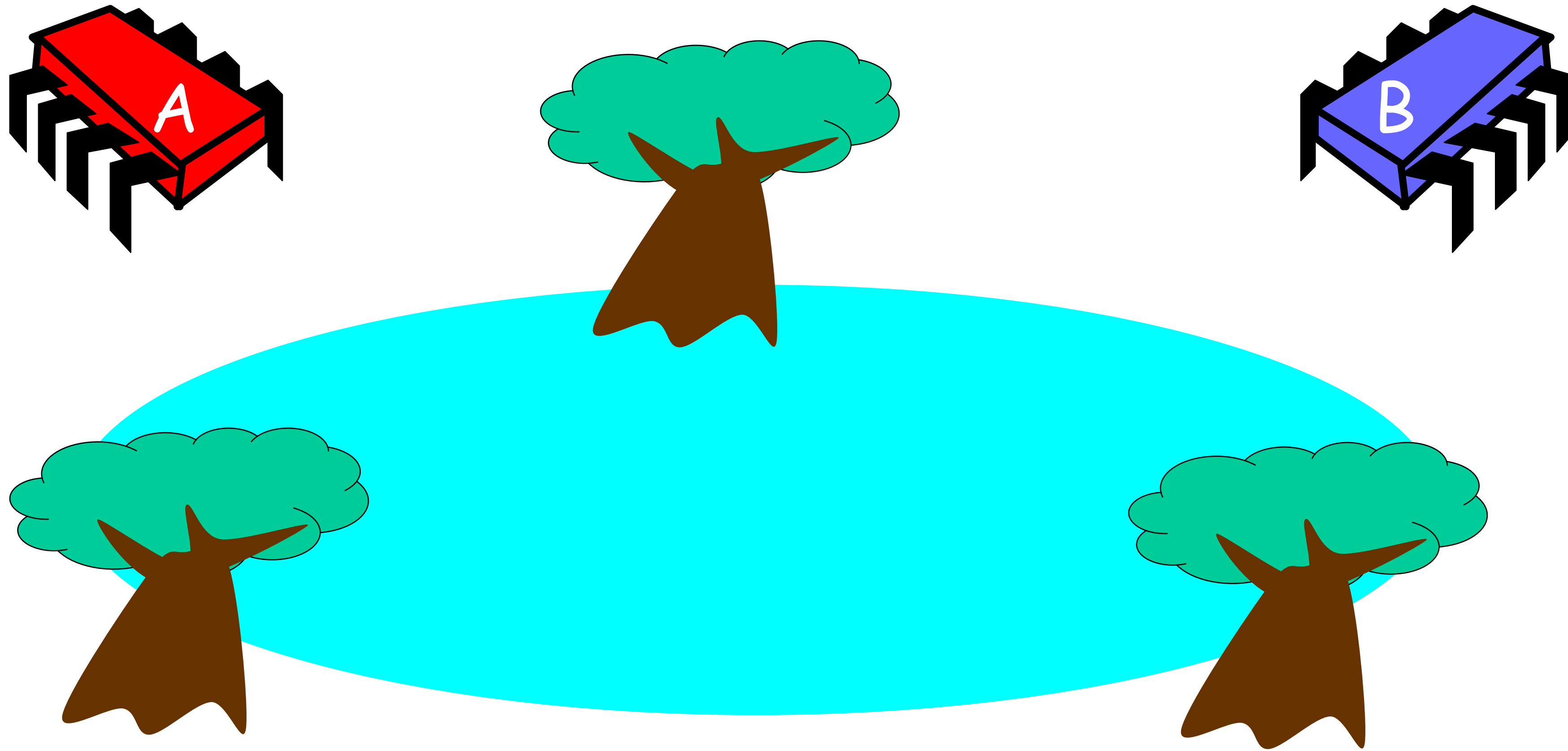
This is one possible interleaving

Mutual Exclusion

- Mutual exclusion: how can we guarantee that multiple threads do **not** enter the same *critical region* at the same time

```
static int i = 0;  
public static void increment()  
{  
    Critical Region → i = i + 1;  
}
```

Mutual Exclusion or “Alice & Bob share a pond”



Art of Multiprocessor
Programming

Alice has a pet



Art of Multiprocessor
Programming

Bob has a pet



Art of Multiprocessor
Programming

The Problem



(the pond is the critical section)

Art of Multiprocessor
Programming

Formalizing the Problem

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

Formalizing our Problem

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

Simple Protocol

- Idea
 - Just look at the pond
- Gotcha
 - Trees obscure the view

Interpretation

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

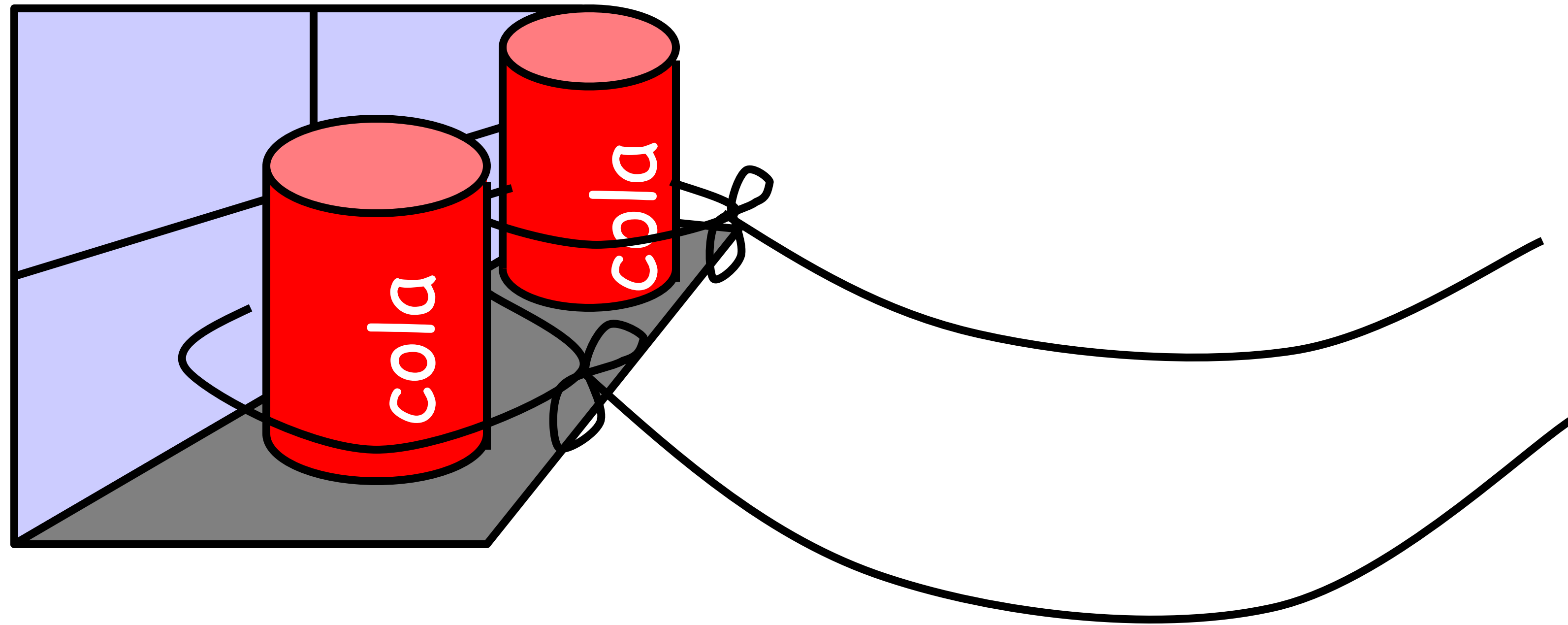
Cell Phone Protocol

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

Interpretation

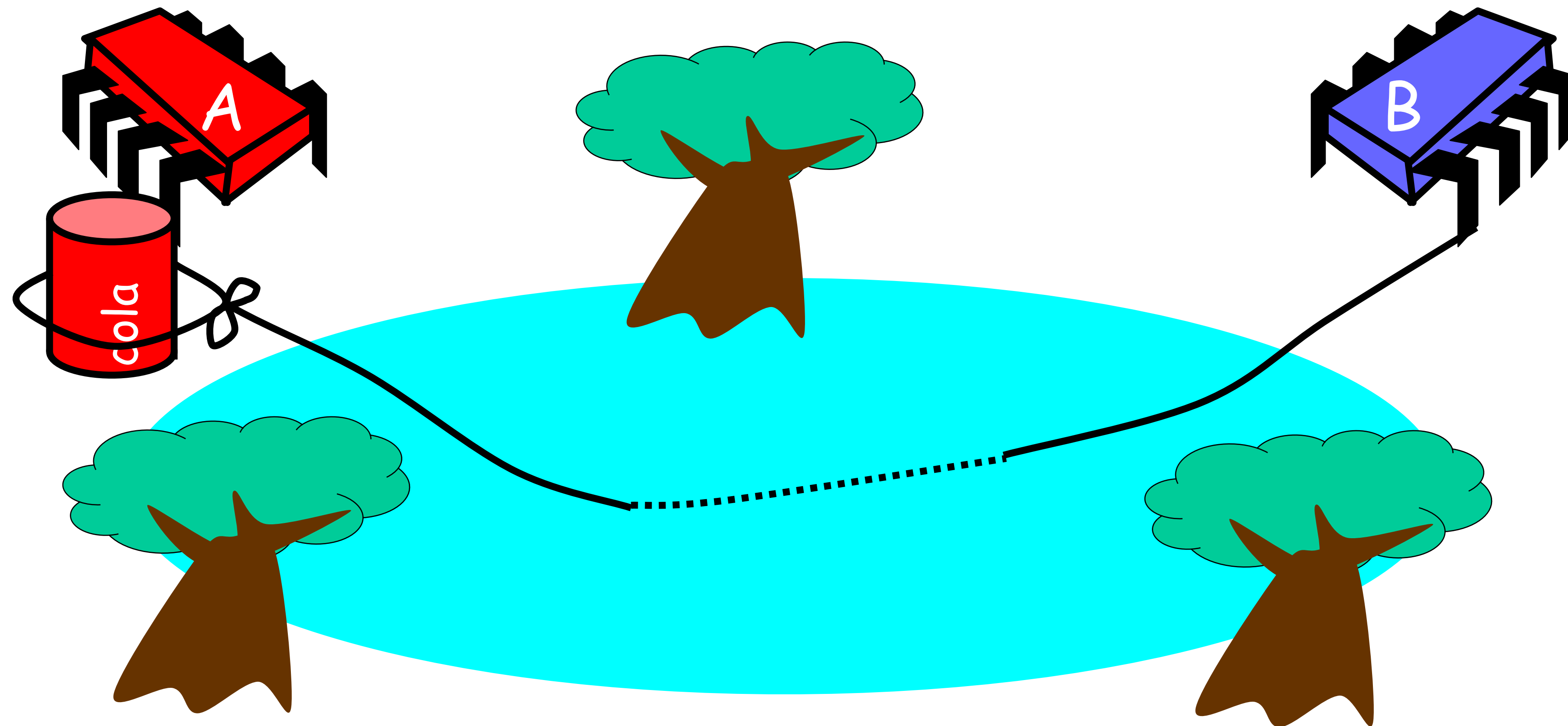
- Message-passing doesn't work
- Recipient might not be
 - Listening
 - There at all
- Communication must be
 - Persistent (like writing)
 - Not transient (like speaking)

Can Protocol



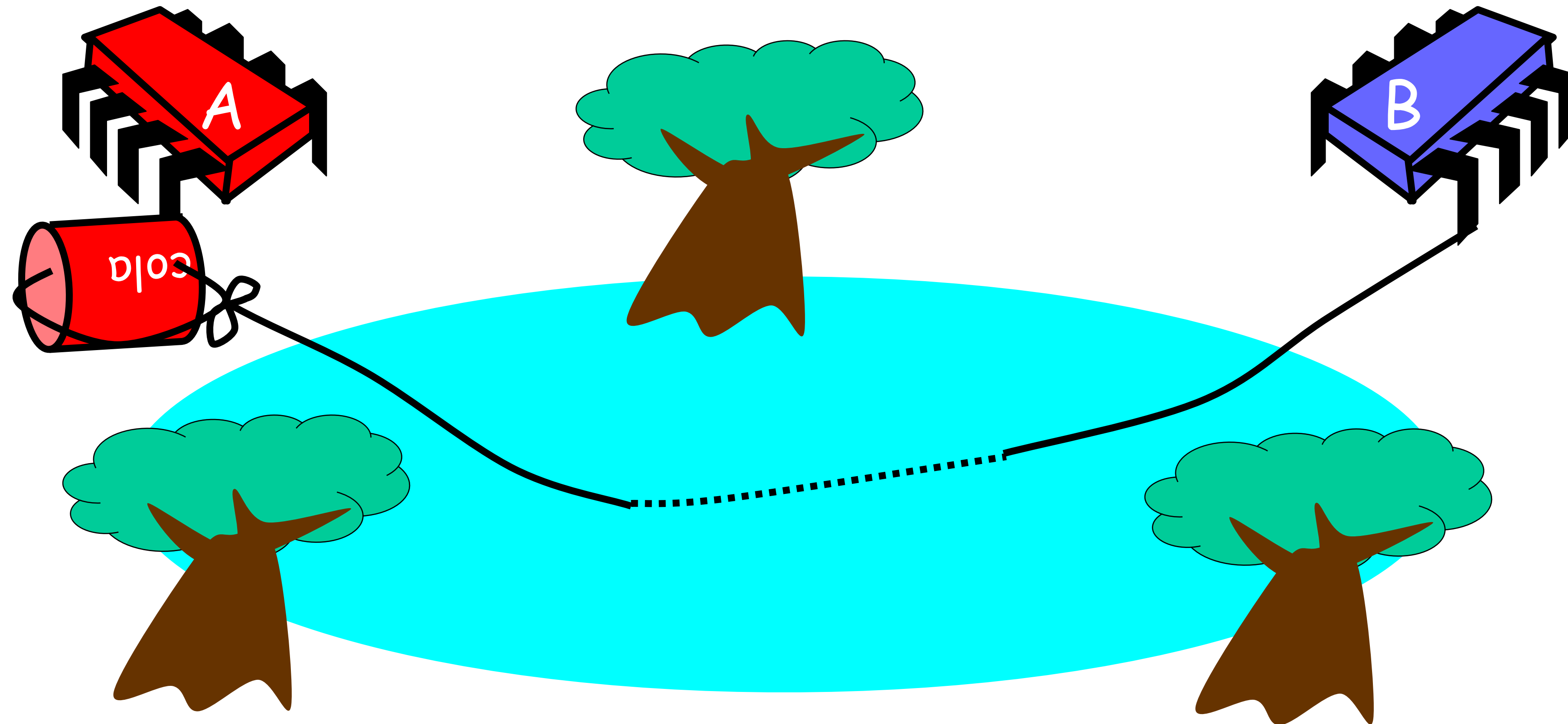
Art of Multiprocessor
Programming

Bob conveys a bit



Art of Multiprocessor
Programming

Bob conveys a bit



Art of Multiprocessor
Programming

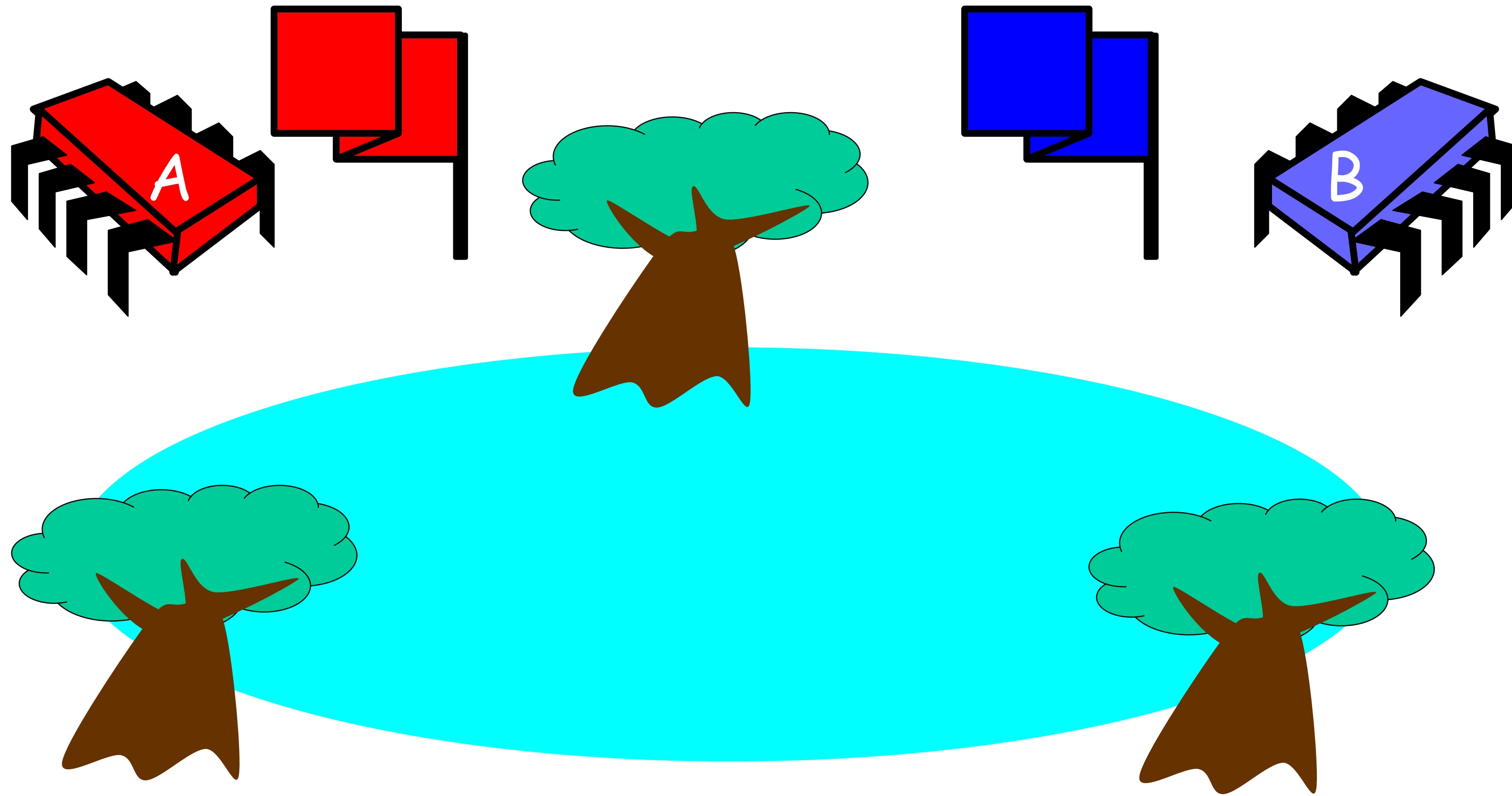
Can Protocol

- 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

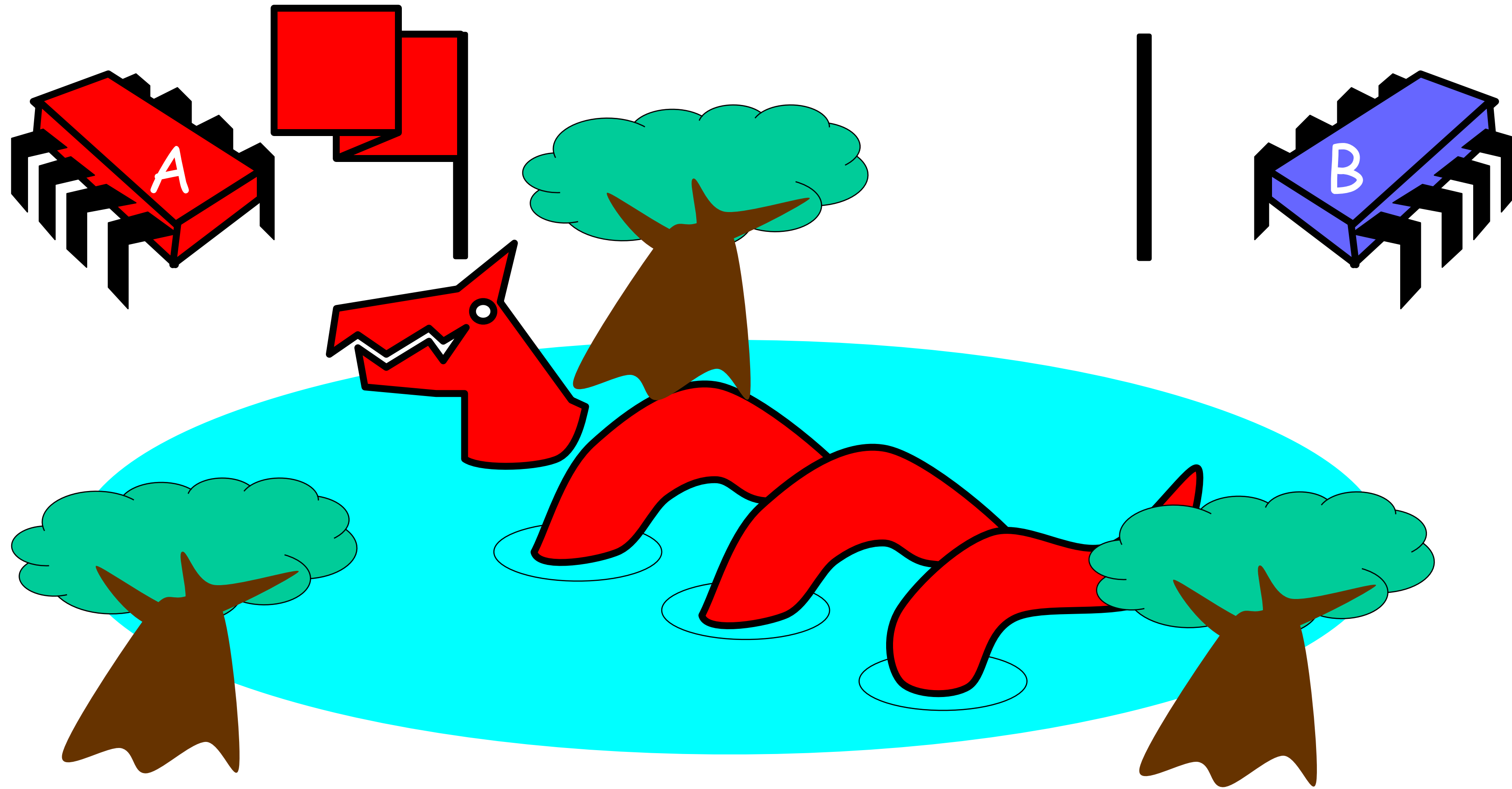
- Cannot solve mutual exclusion with interrupts
 - Sender sets fixed bit in receiver's space
 - Receiver resets bit when ready
 - Requires unbounded number of interrupt bits

Flag Protocol



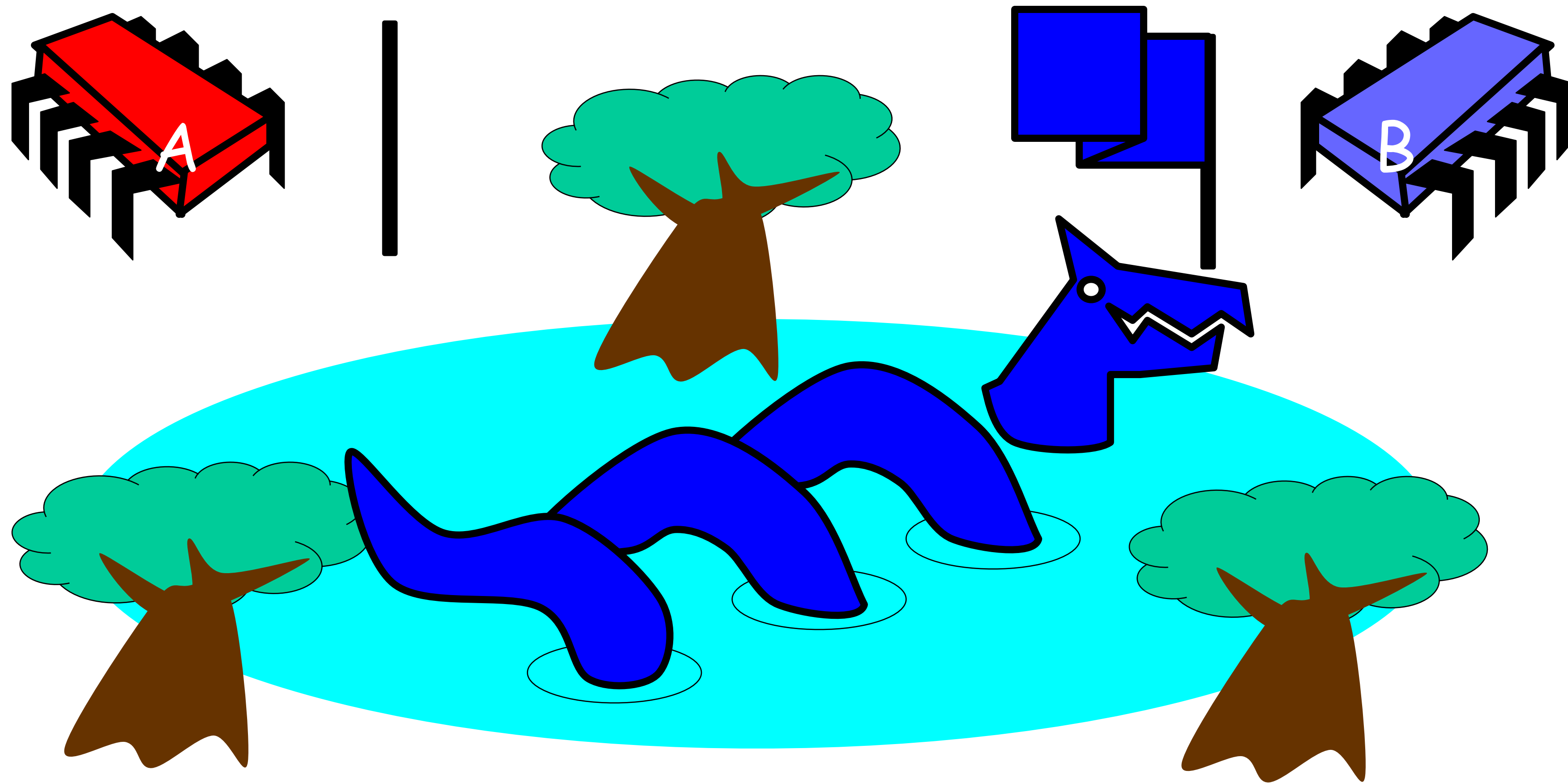
Art of Multiprocessor
Programming

Alice's Protocol (sort of)



Art of Multiprocessor
Programming

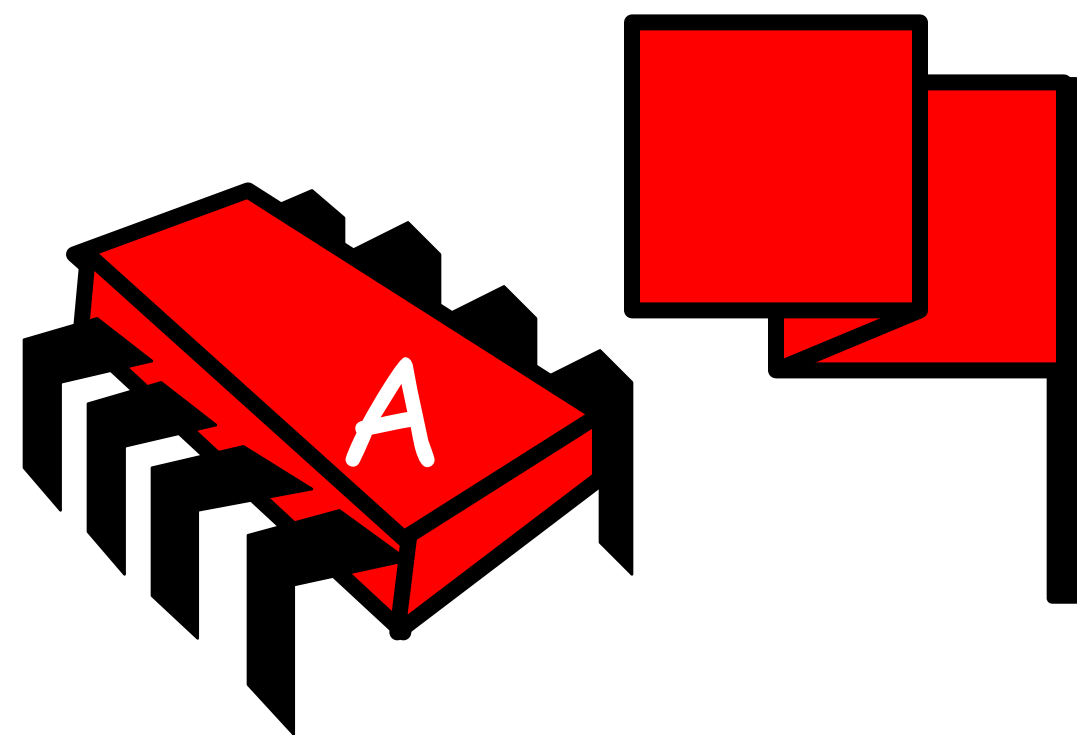
Bob's Protocol (sort of)



Art of Multiprocessor
Programming

Alice's Protocol

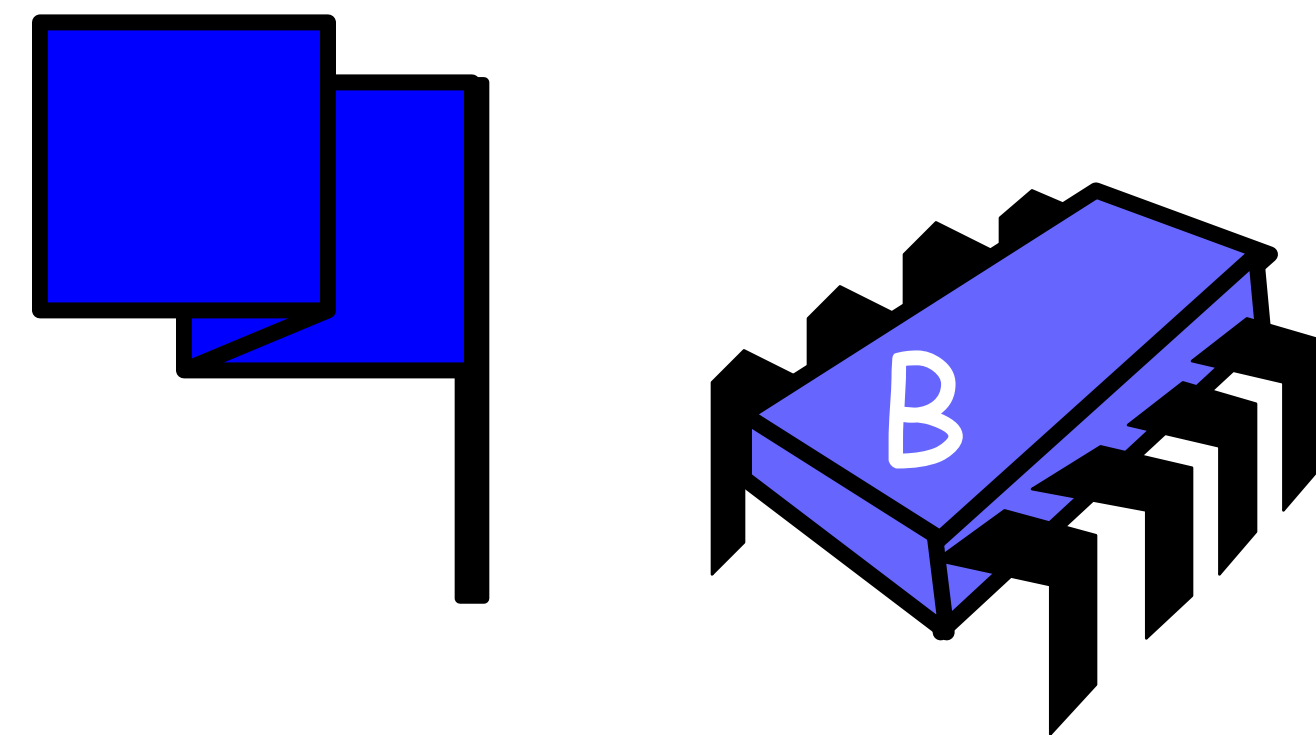
- Raise flag
- Wait until Bob's flag is down
- Unleash pet
- Lower flag when pet returns



After you!

Bob's Protocol

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

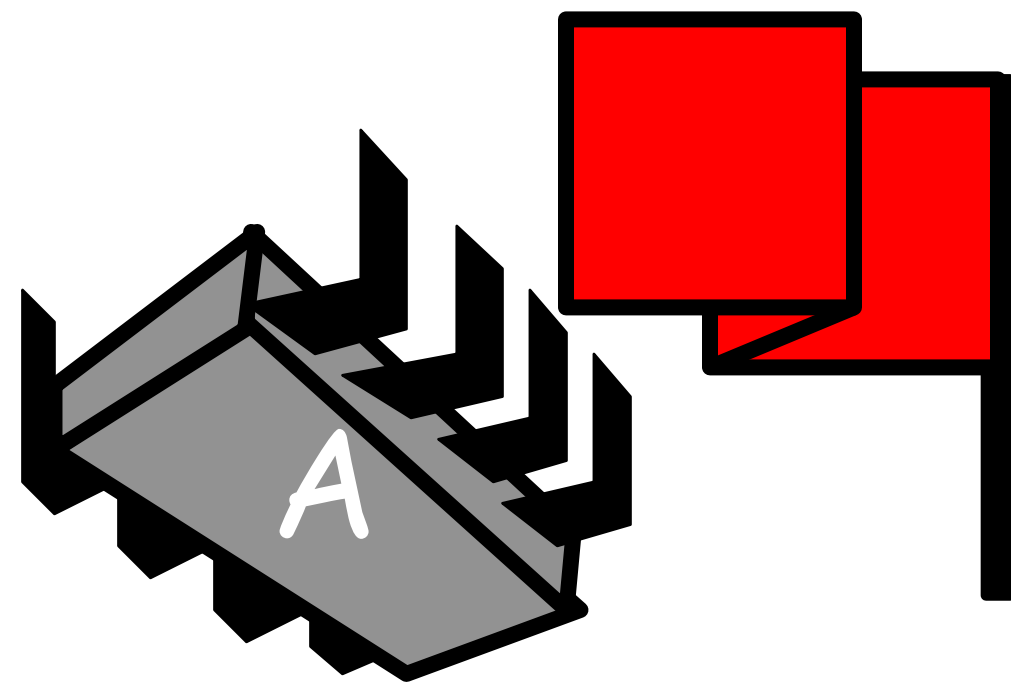


No, no... after you!



Alice's Protocol

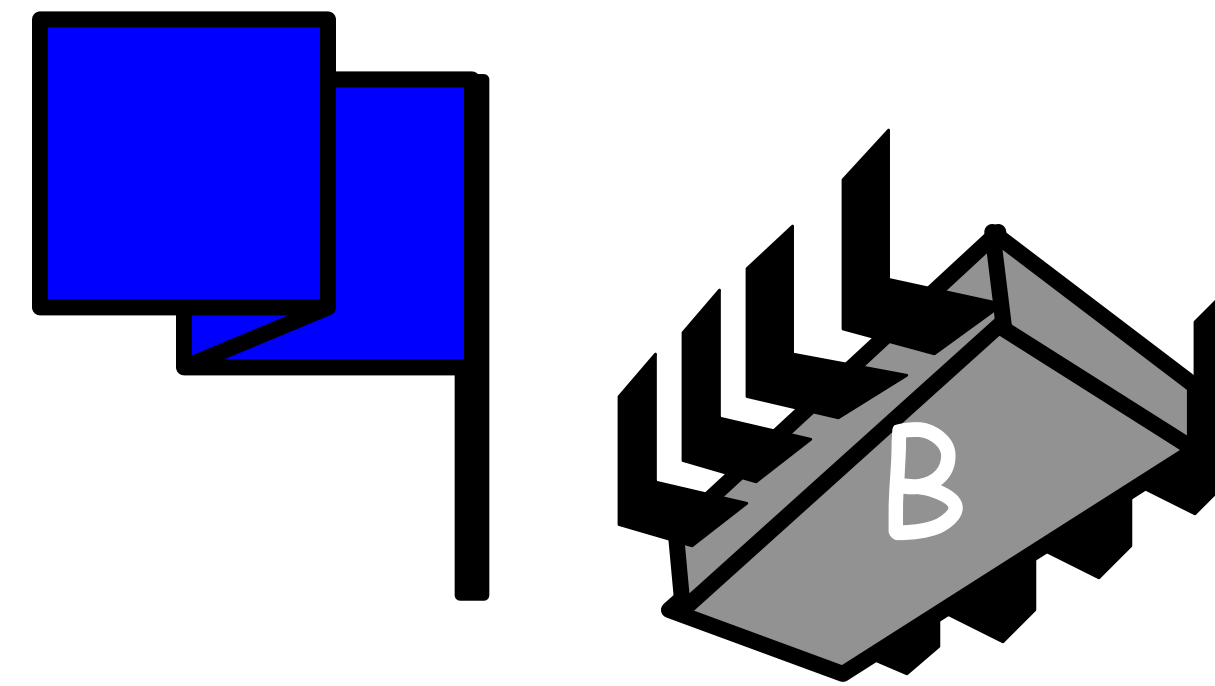
- Raise flag
- Wait until Bob's flag is down
- Unleash pet
- Lower flag when pet returns



After you!

Bob's Protocol

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



No, no... after you!




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'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**
- 
- A red callout bubble with a pointed tail pointing towards the 'While Alice's flag is up' section of the protocol. The bubble contains the text 'Bob defers to Alice' in red.

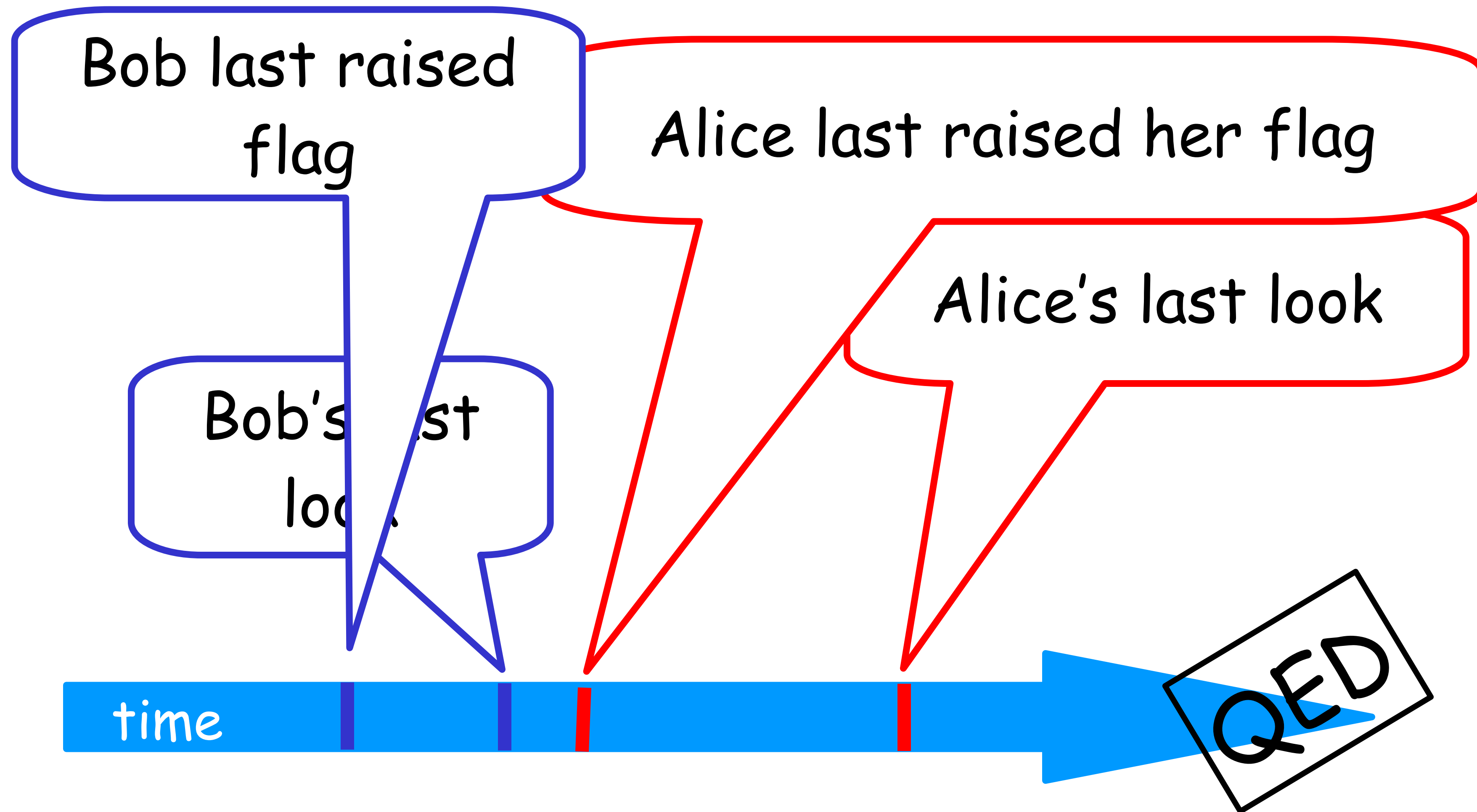
The Flag Principle

- 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

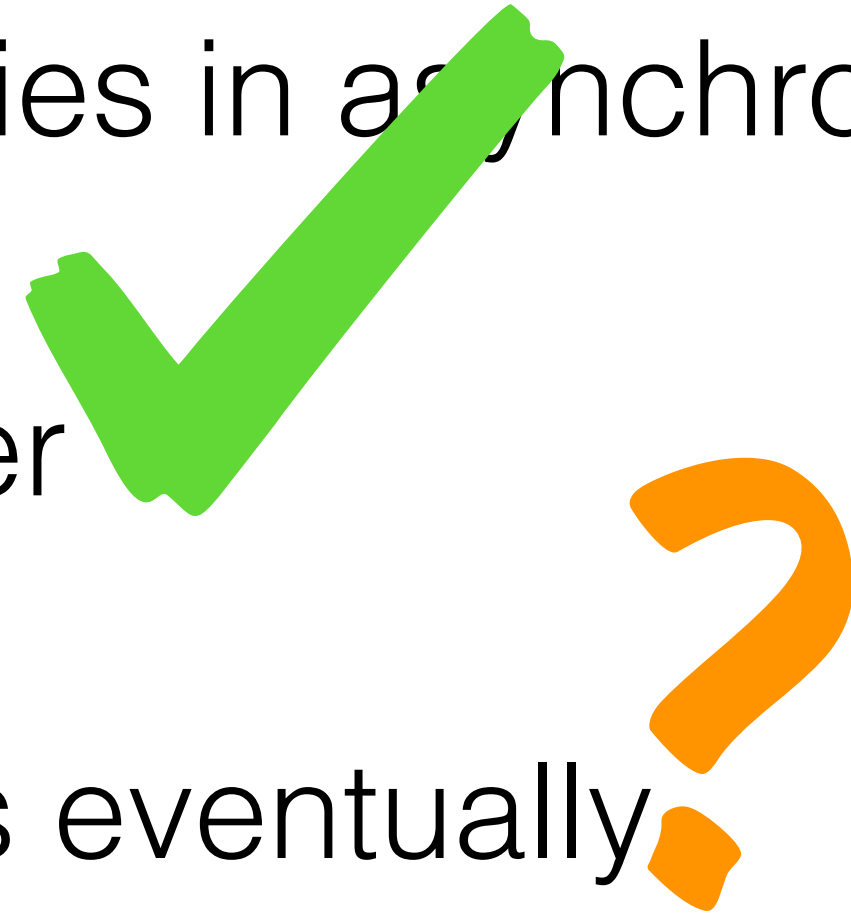
Proof of No Deadlock

- 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

Formalizing the Problem

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



Remarks

- Protocol is unfair
 - Bob's pet might never get in (**starvation**)
- 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 (flags)

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”
- HW 1 will be posted Monday

This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>
- You are free to:
 - Share — copy and redistribute the material in any medium or format
 - Adapt — remix, transform, and build upon the material
 - for any purpose, even commercially.
- Under the following terms:
 - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
 - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.