Processes & Threads

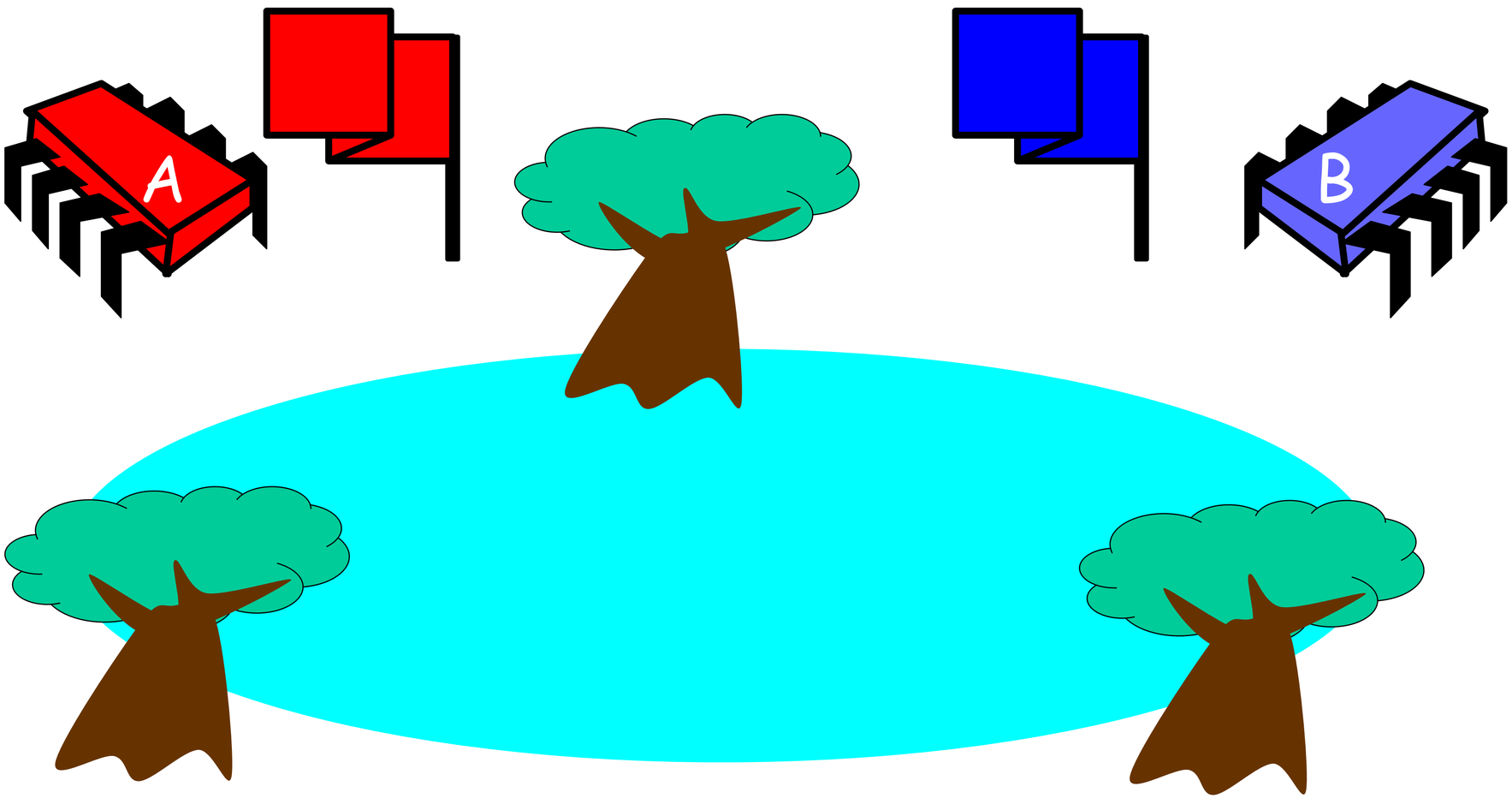
CS 475, Spring 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

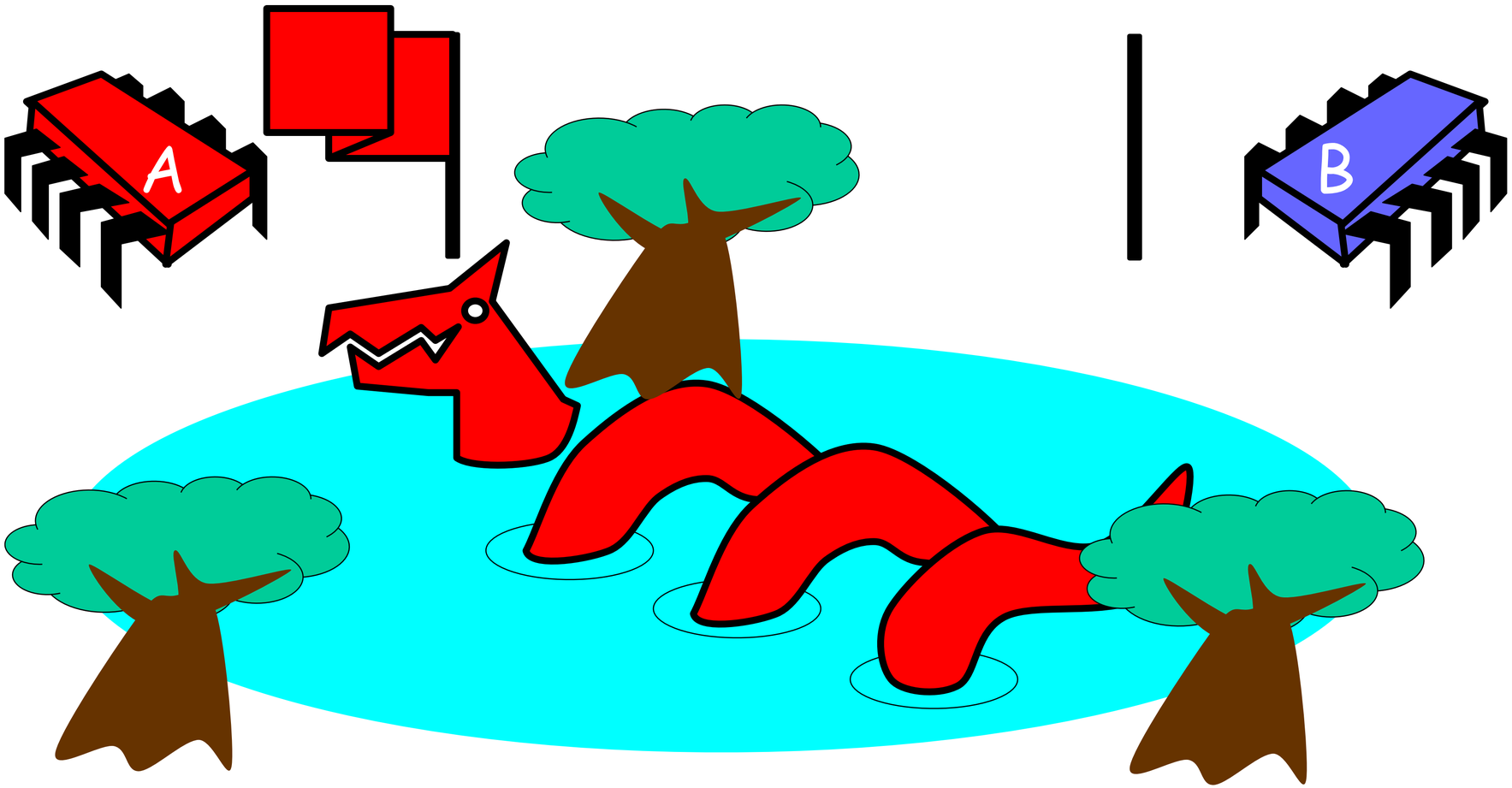
With material from Herlihy & Shavit, Art of Multiprocessor Programming

The Problem

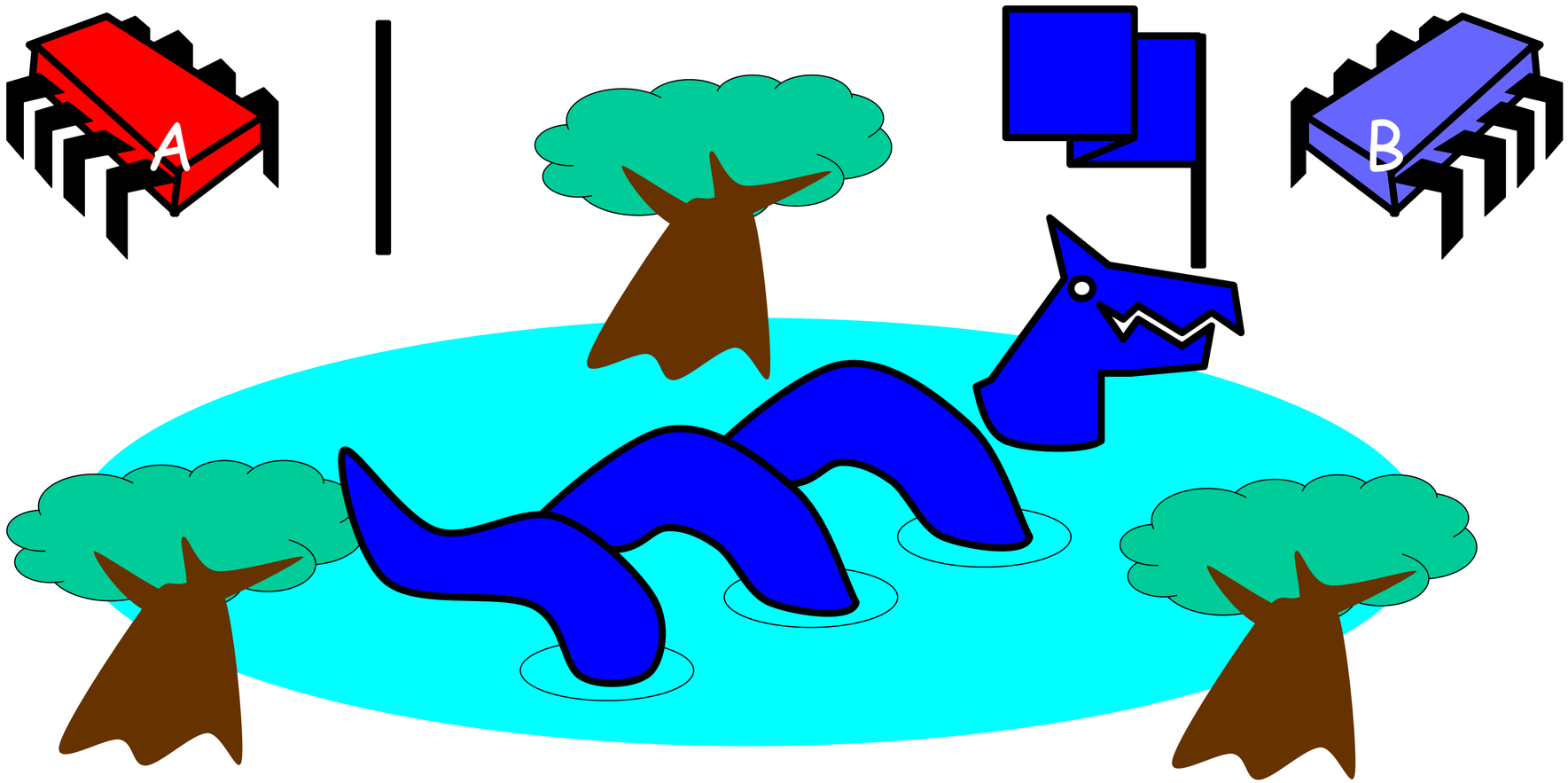
Flag Protocol



Alice's Protocol (sort of)

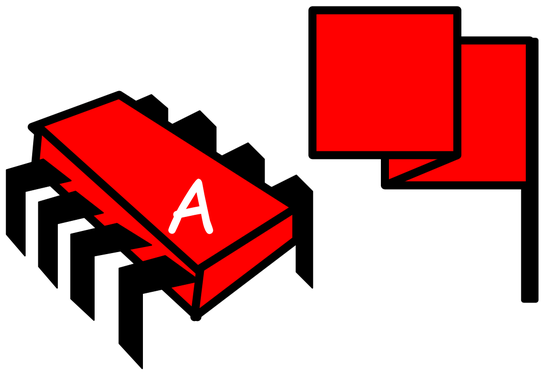


Bob's Protocol (sort of)



Alice's Protocol

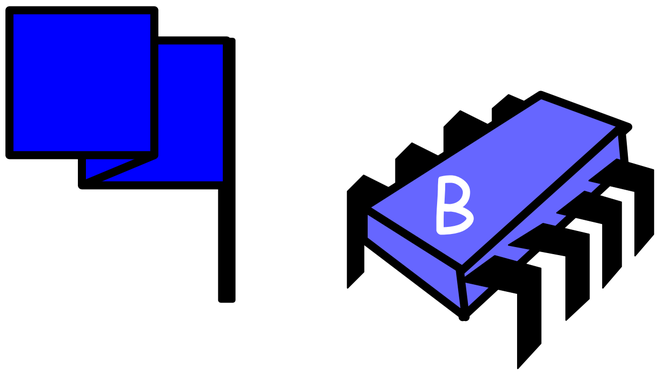
- Raise flag
- Wait until Bob's flag is down
- Unleash pet
- Lower flag when pet returns



After you!

Bob's Protocol

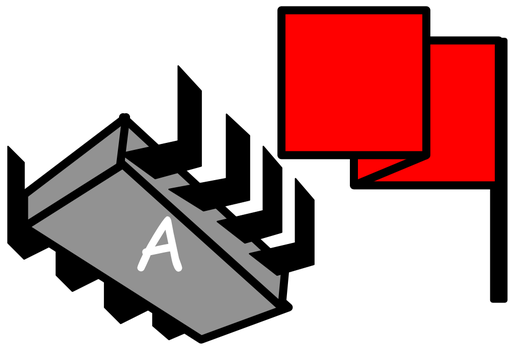
- Raise flag
- Wait until Alice's flag is down
- Unleash pet
- Lower flag when pet returns



No, no… after you!

Alice's Protocol

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

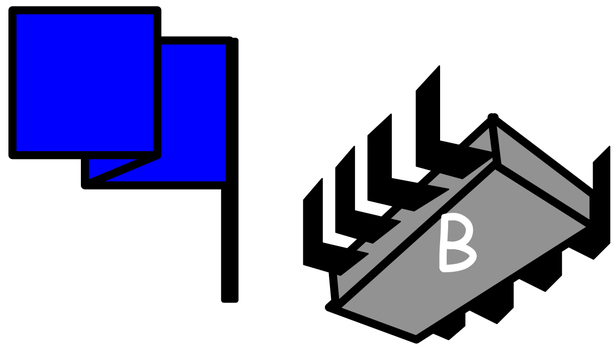


After you!

Bob's Protocol

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

danger!



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

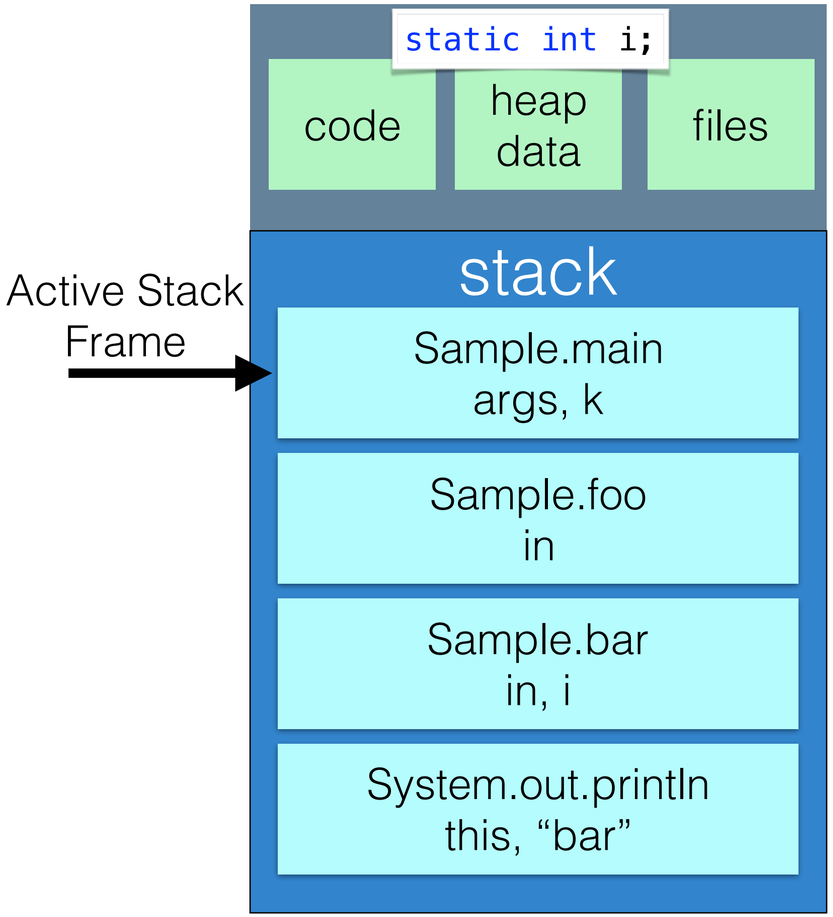
Today

- What OS abstractions do we use for concurrency and parallelism?
  - Threads
  - Processes
- A few more parables to outline more problems with concurrent computation
- Reading: H&S 1.5

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

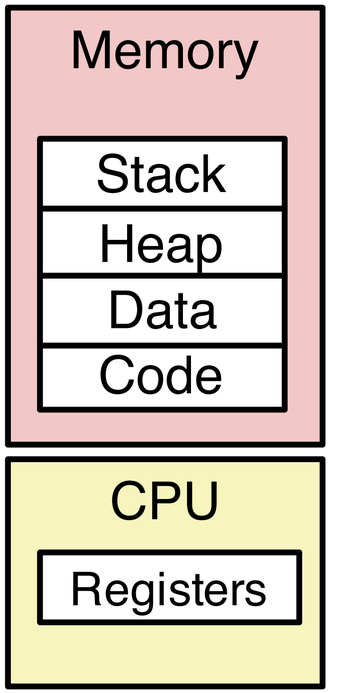
Processes



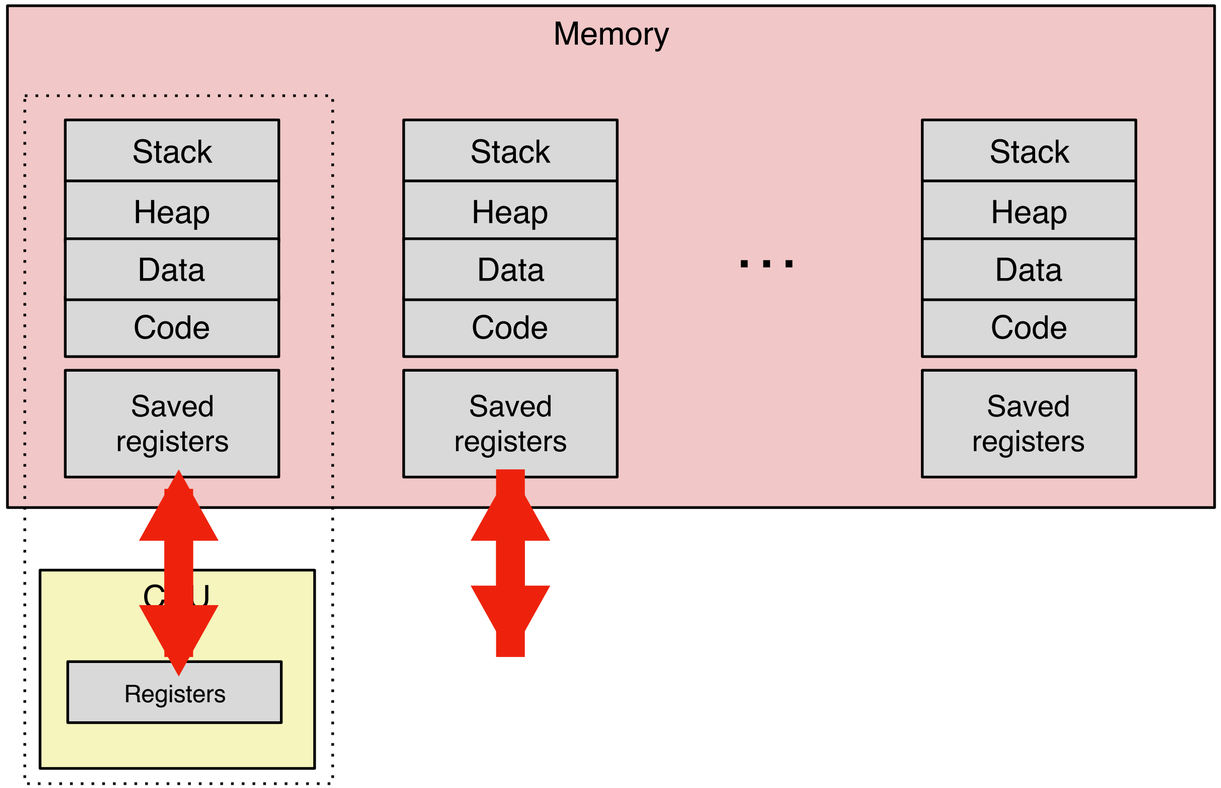

```
public class Sample
{
    static int i;
    public static void main(String[] args)
    {
        int k = 10;
        foo(k);
    }
    public static void foo(int in)
    {
        bar(in);
    }
    public static void bar(int in)
    {
        i = in;
        System.out.println("bar");
    }
}
```

Process Representation

- A process has some mapping into the physical machine (machine state)
- Provide two key abstractions to programs:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called context switching
  - Private address space
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called virtual memory



CPU Switching from Process to Process

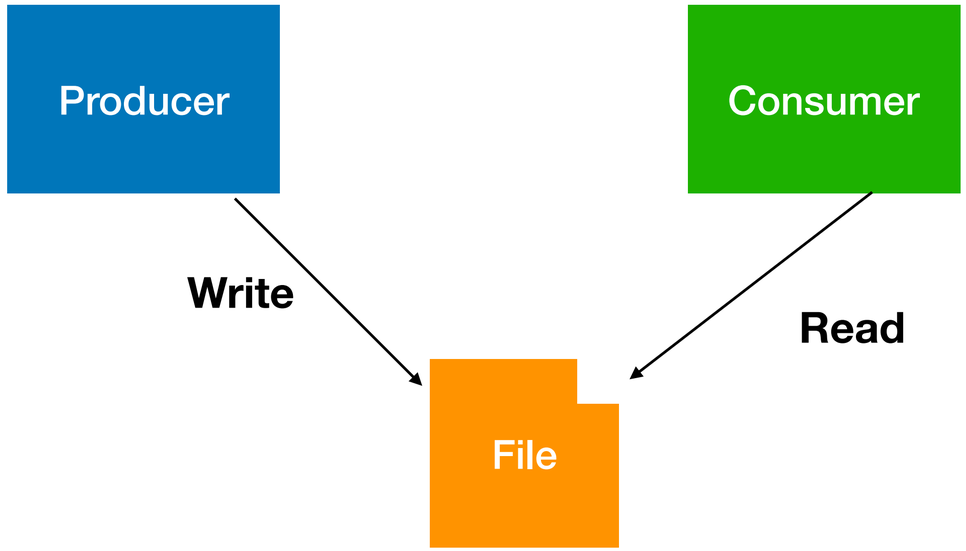


Interprocess Communication

- We might want two processes to seriously work together
- For example:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Signals are very, very NOT sufficient for these purposes
- What we need is **interprocess communication (IPC)**

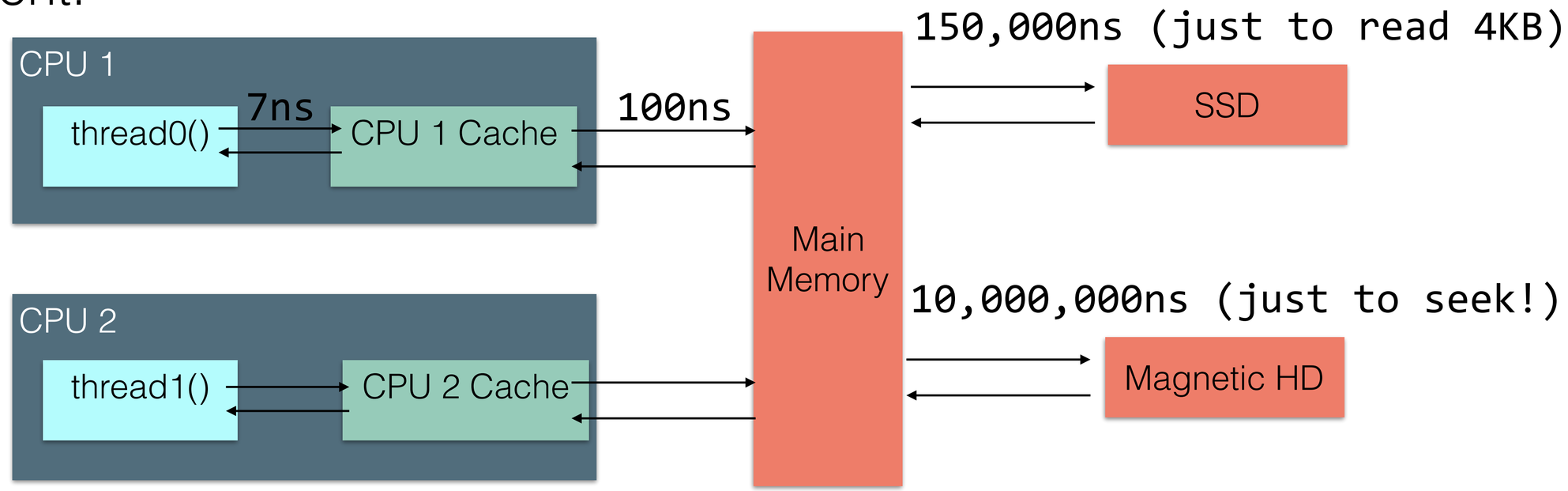
Strawman IPC

- Producer writes to a file
- Consumer reads from same file



Strawman IPC

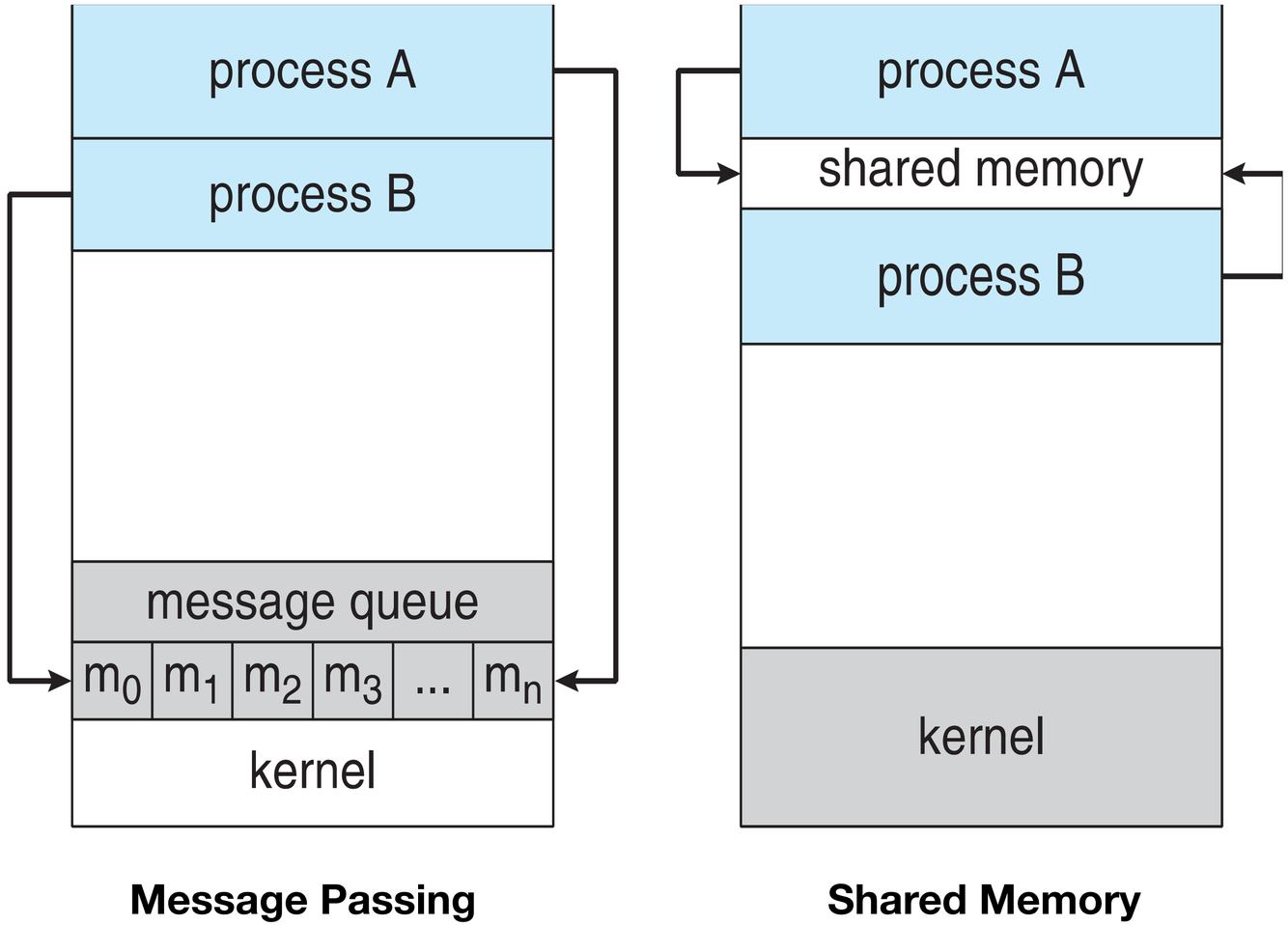
- Does it work? Yes
- Is it cumbersome (and perhaps error-prone)? Yes
  - What happens if consumer reads while producer is writing?
- Is it efficient?
  - No
  - Argument:



Improving on the Strawman

- Shared memory
  - Strawman, but the “file” is just a hunk of memory that’s shared between processes
- Message Passing
  - Abstraction on top of shared memory: producer sends messages to consumer

Message Passing & Shared Memory

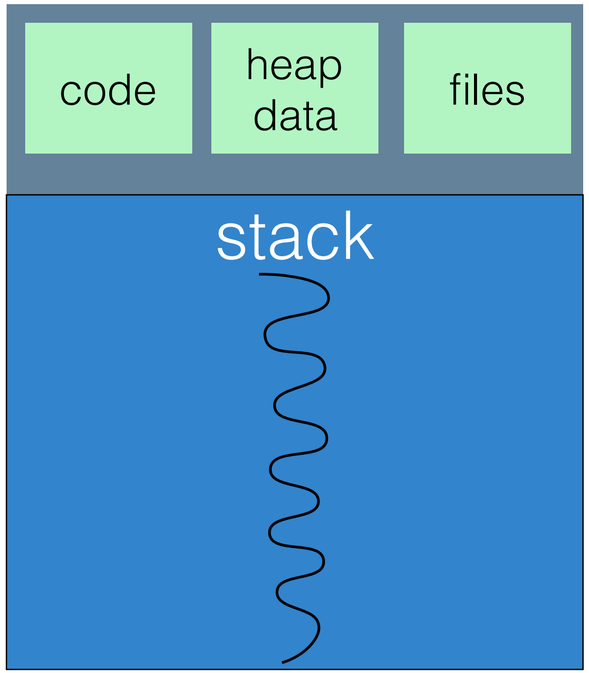


Message Passing **Shared Memory**

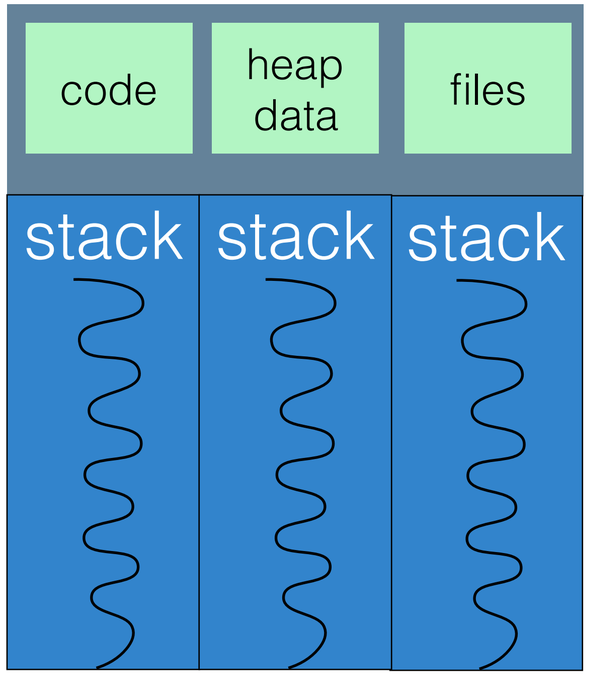
Threads

- 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)

Processes vs Threads

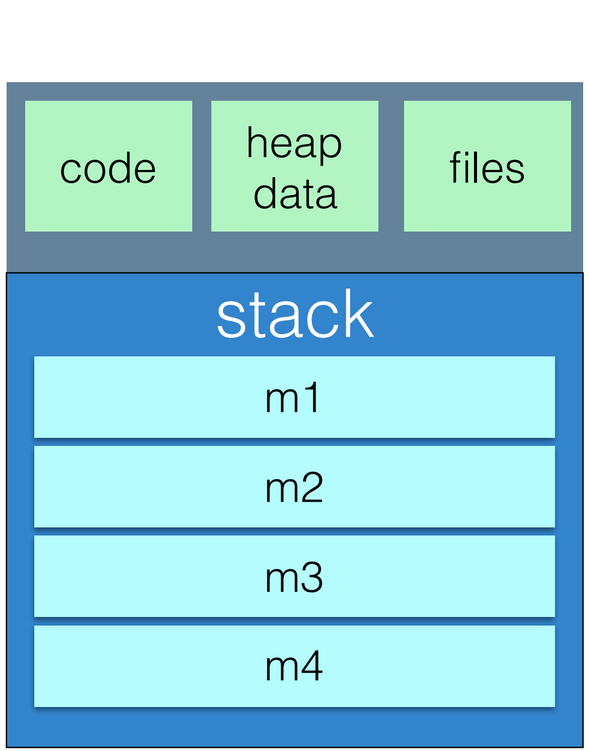


Single-Threaded Process

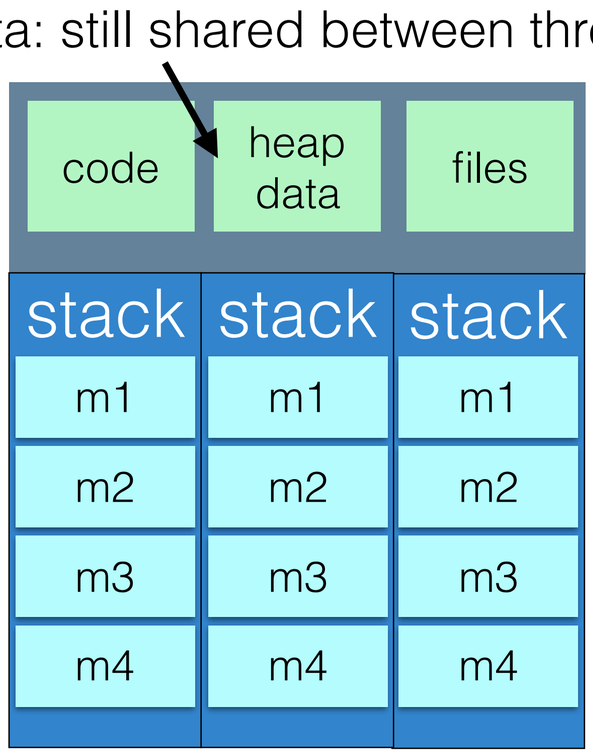


Multi-Threaded Process

Threads: Memory View



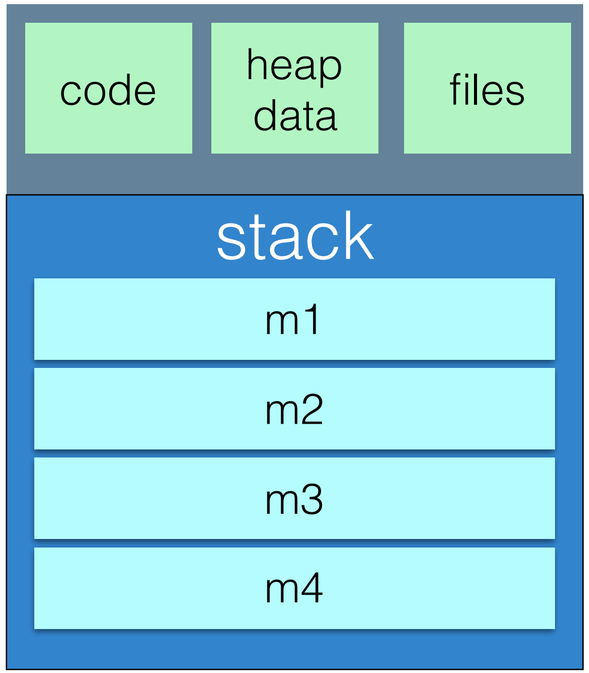
Single-Threaded Process



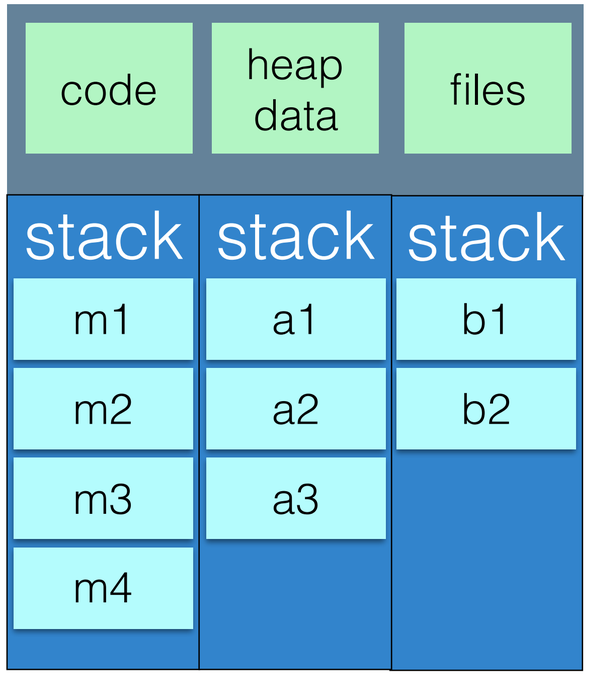
Multi-Threaded Process

Each thread might be executing the same code, but with different local variables (and hence doing different stuff)

Threads: Memory View



Single-Threaded Process



Multi-Threaded Process

Each thread might be executing totally different code, too

Processes vs Threads

- Context Switching
  - Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
  - When switching processes, **all** of that data needs to get flushed out (by the OS)
- Threads share the same address space: no need to do this switch

Processes vs Threads

- How threads and processes are similar
  - Each has its own logical control flow.
  - Each can run concurrently.
  - Each is context switched.
- How threads and processes are different
  - Threads share code and data, processes (typically) do not.
  - Threads are somewhat less expensive than processes.
  - Process control (creating and reaping) is (ballpark!) twice as expensive as thread control.

Thread Communication

- Same two high level options as processes: shared memory or message passing
- Shared memory:
  - Things are shared by default!
- Message passing:
  - Programmer manually says what to share
- We will focus on the simple shared memory approach, but keep in mind other options too

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- Specification, not implementation
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

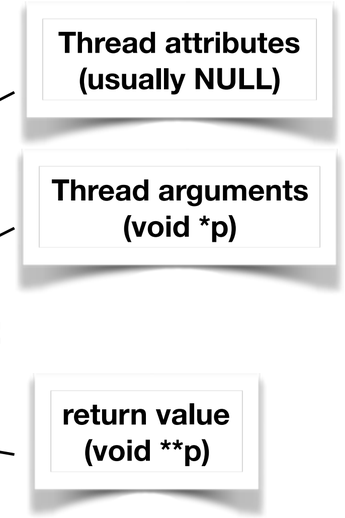
```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
    pthread_t tid;

    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
```



Threads in Java

- In Java, make a new thread by instantiating the class java.lang.Thread
- Pass it an object that implements *Runnable*
- When you call thread.start(), the run() method of your runnable is called, from a new thread
- join() waits for a thread to finish

```
Thread t = new Thread(new Runnable() {
 @Override
    public void run() {
   //This code will now run in a new thread
    }
});
t.start();
```

Threads in Java

- JVM manages threads (maybe uses Pthreads underneath)
- Each Java app gets at least one thread: `main`
  - Plus, likely a `finalizer` thread
  - Plus, the JVM itself makes a ton of threads that you can't see
    - JIT compiler, garbage collector mainly
- Fun tip: look at what threads are running in a Java app using the command-line `jstack` program

Threads in Java

```
public static void main(String[] args) throws InterruptedException {
 Thread t = new Thread(new Runnable() {
     @Override
     public void run() {
      //This code will now run in a new thread
       System.out.println("Hello from the thread!");
     }
   });
   t.start();
   System.out.println("Hello from main!”);
   t.join();
}
```

What is the output of this code?

#1 Hello from the thread!
Hello from main!

This is a race condition

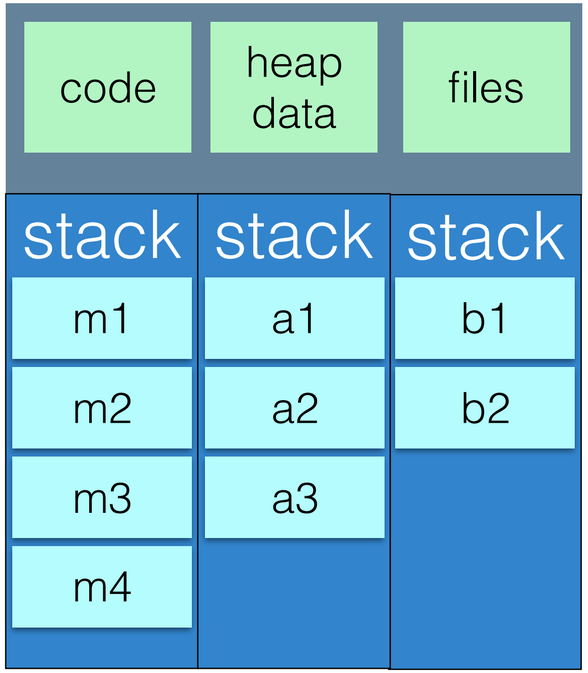
#2 Hello from main!
Hello from the thread!

Thread Communication

- Threads execute separate logical segments of code
- How do they talk to each other?

```
public static void main(String[] args) throws InterruptedException {
 Thread t = new Thread(new Runnable() {
     @Override
     public void run() {
      //This code will now run in a new thread
       System.out.println("Hello from the thread!");
     }
   });
   t.start();
   System.out.println("Hello from main!”);
   t.join();
}
```

Shared Variables in Threads



Multi-Threaded Process

Live Programming Example - Threads

The Fable Continues

- Alice and Bob fall in love & marry

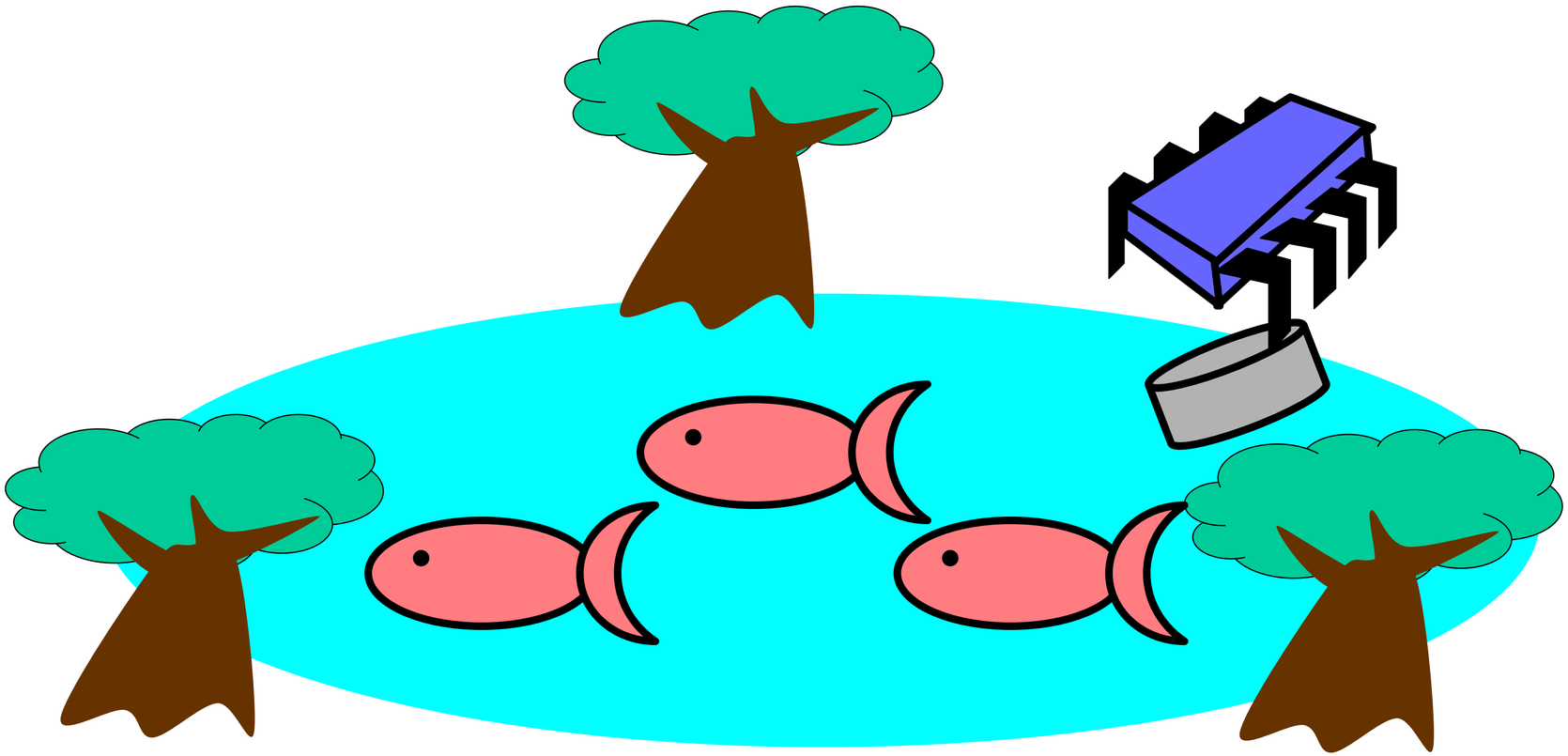
The Fable Continues

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

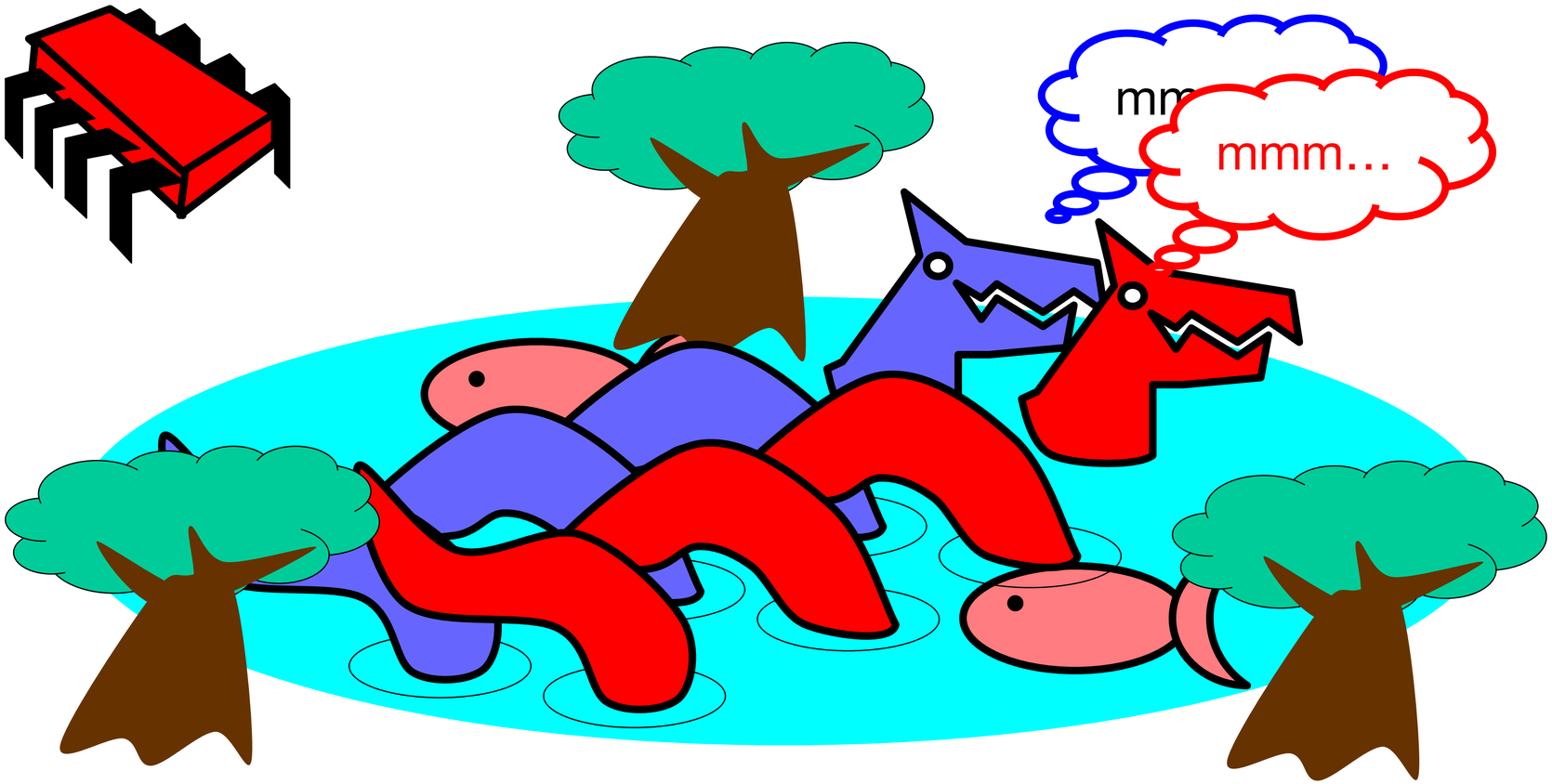
The Fable Continues

- 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



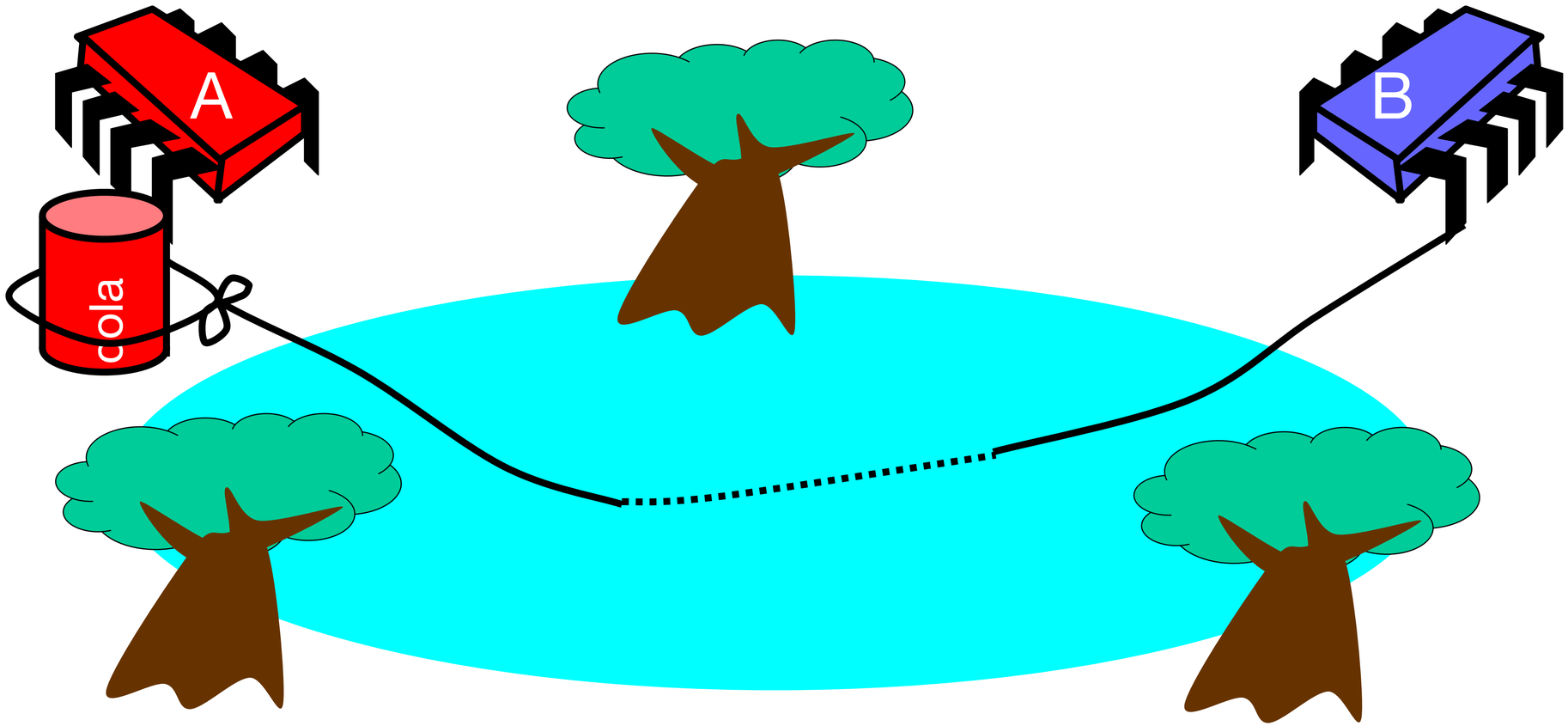
Producer/Consumer

- 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

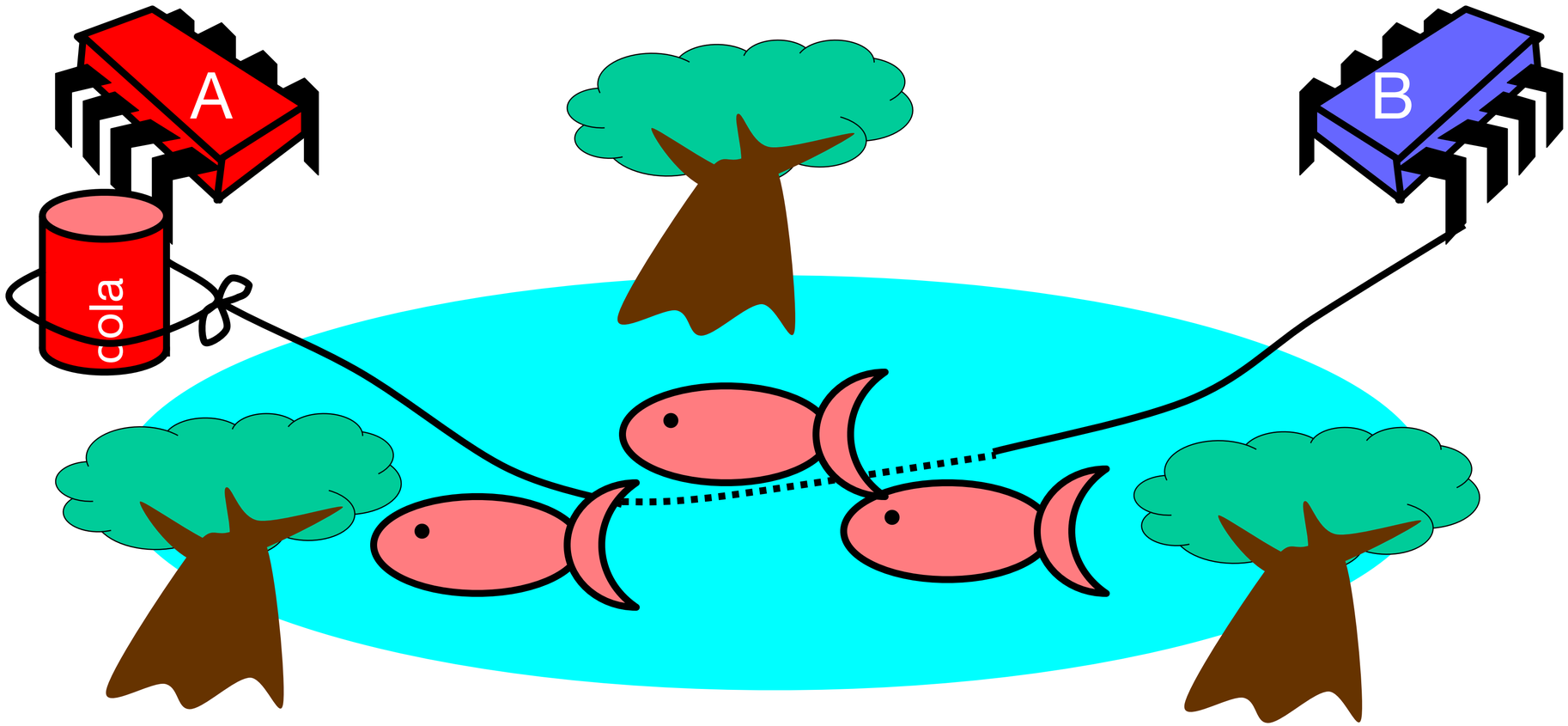
Producer/Consumer

- 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

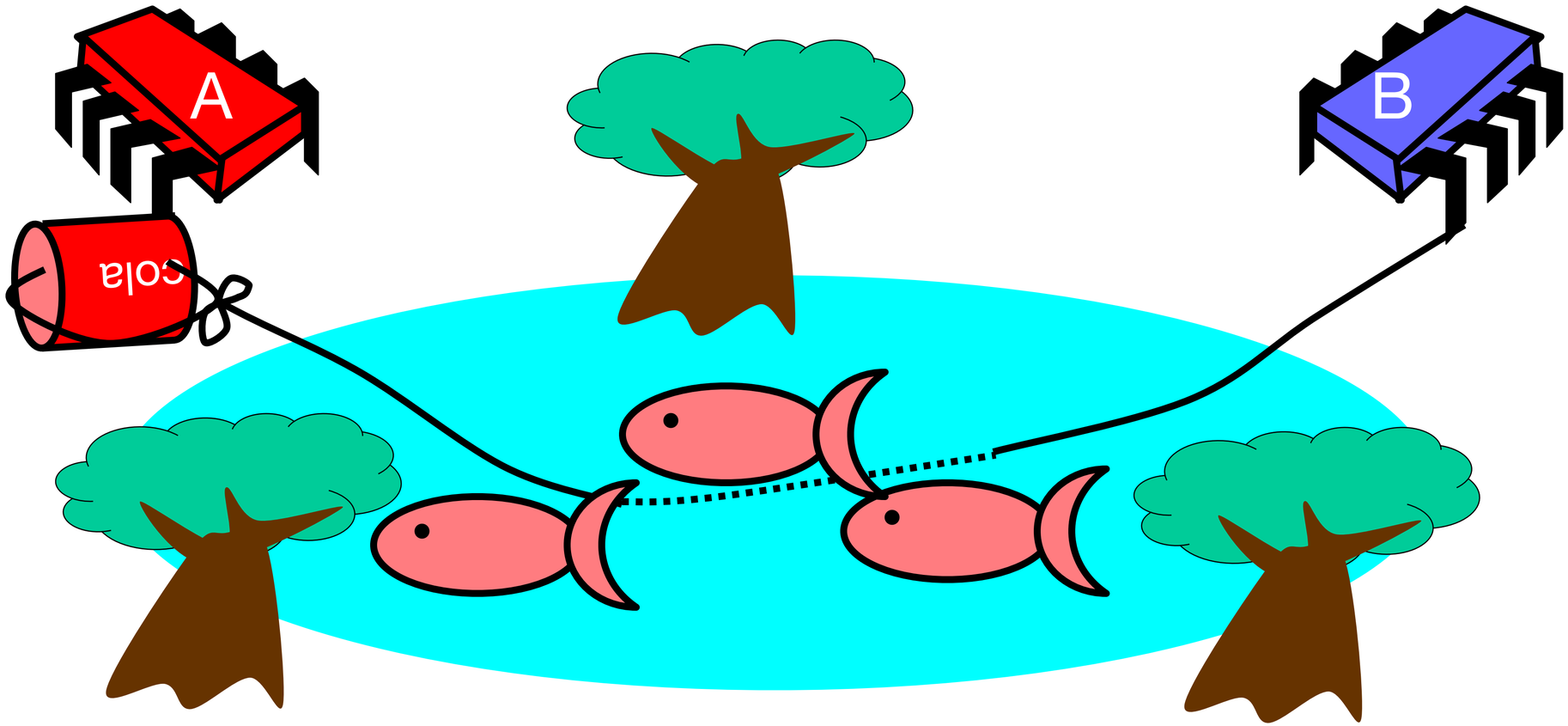
Surprise Solution



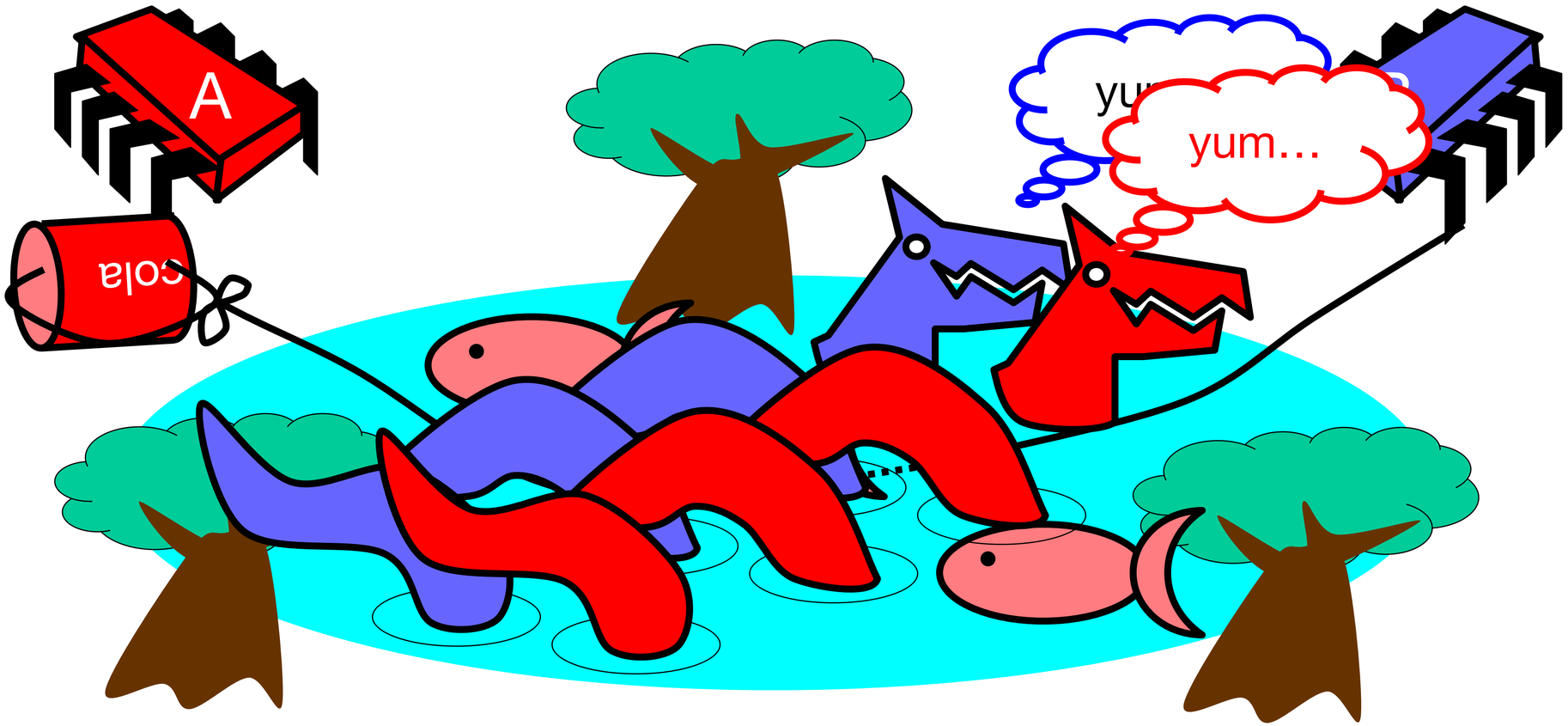
Bob puts food in Pond



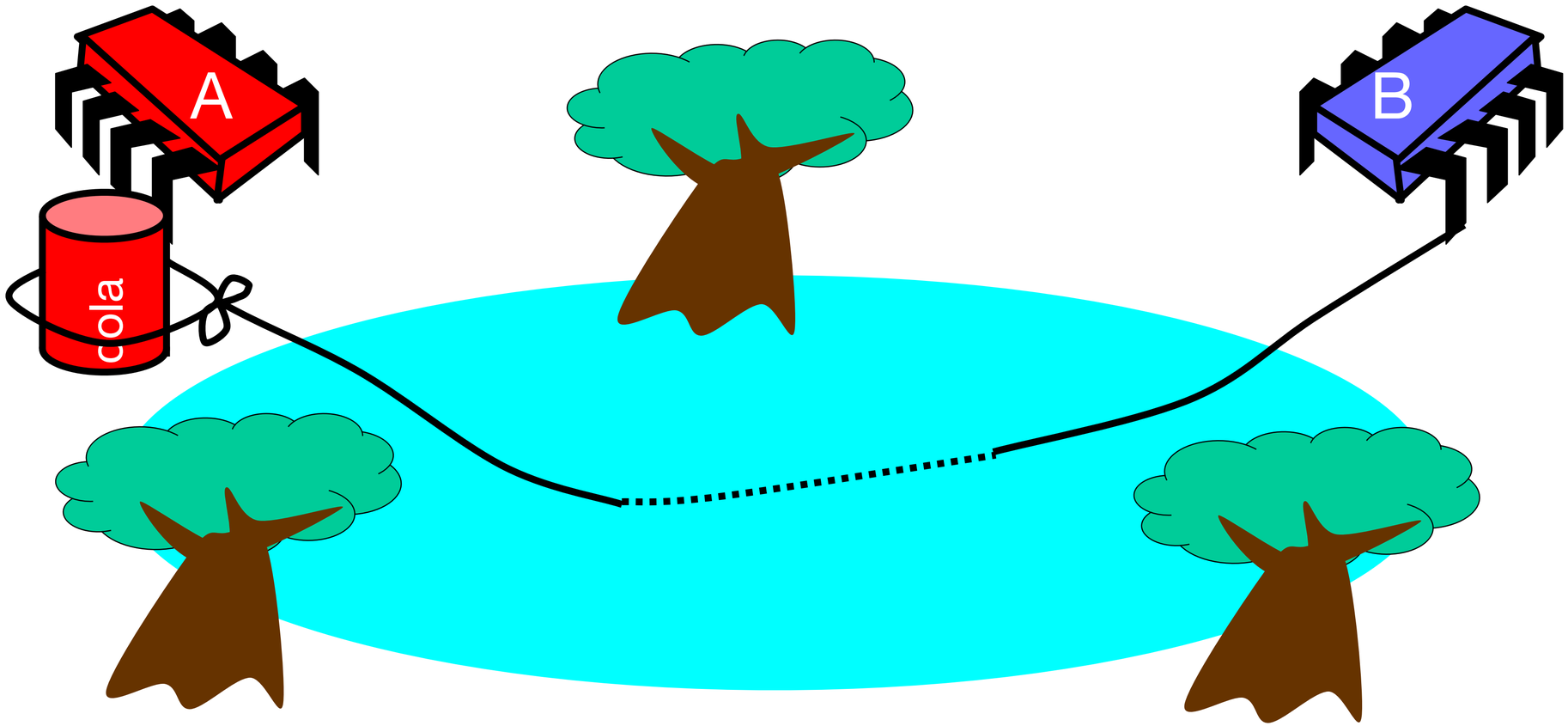
Bob knocks over Can



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

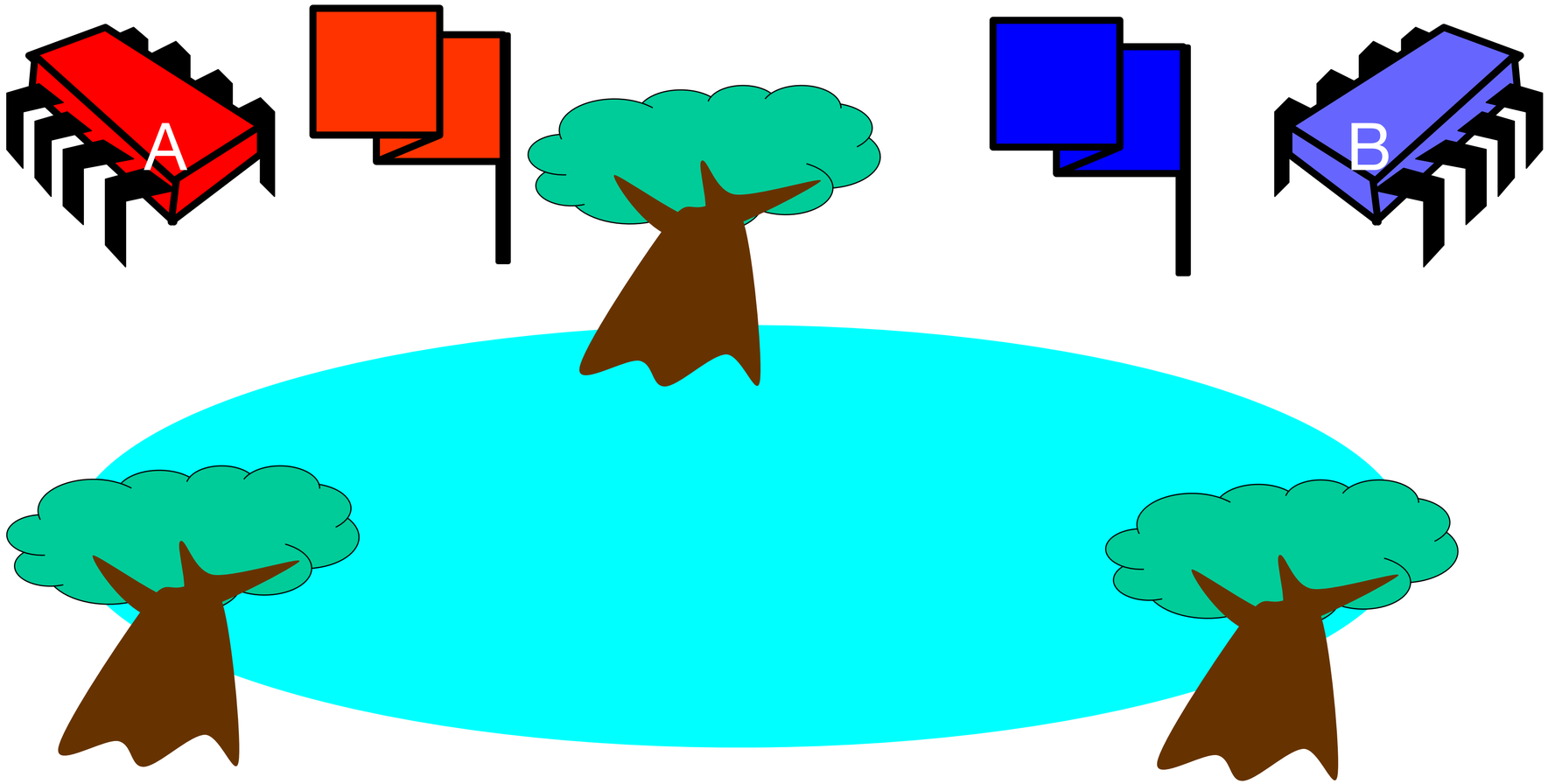
- Mutual Exclusion
  - Pets and Bob never together in pond
- No Starvation
  if Bob always willing to feed, and pets always famished, then pets eat infinitely often.
- Producer/Consumer
  The pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food.

safety

liveness

safety

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 & unpredictable**

The Fable drags on …

- Bob and Alice still have issues

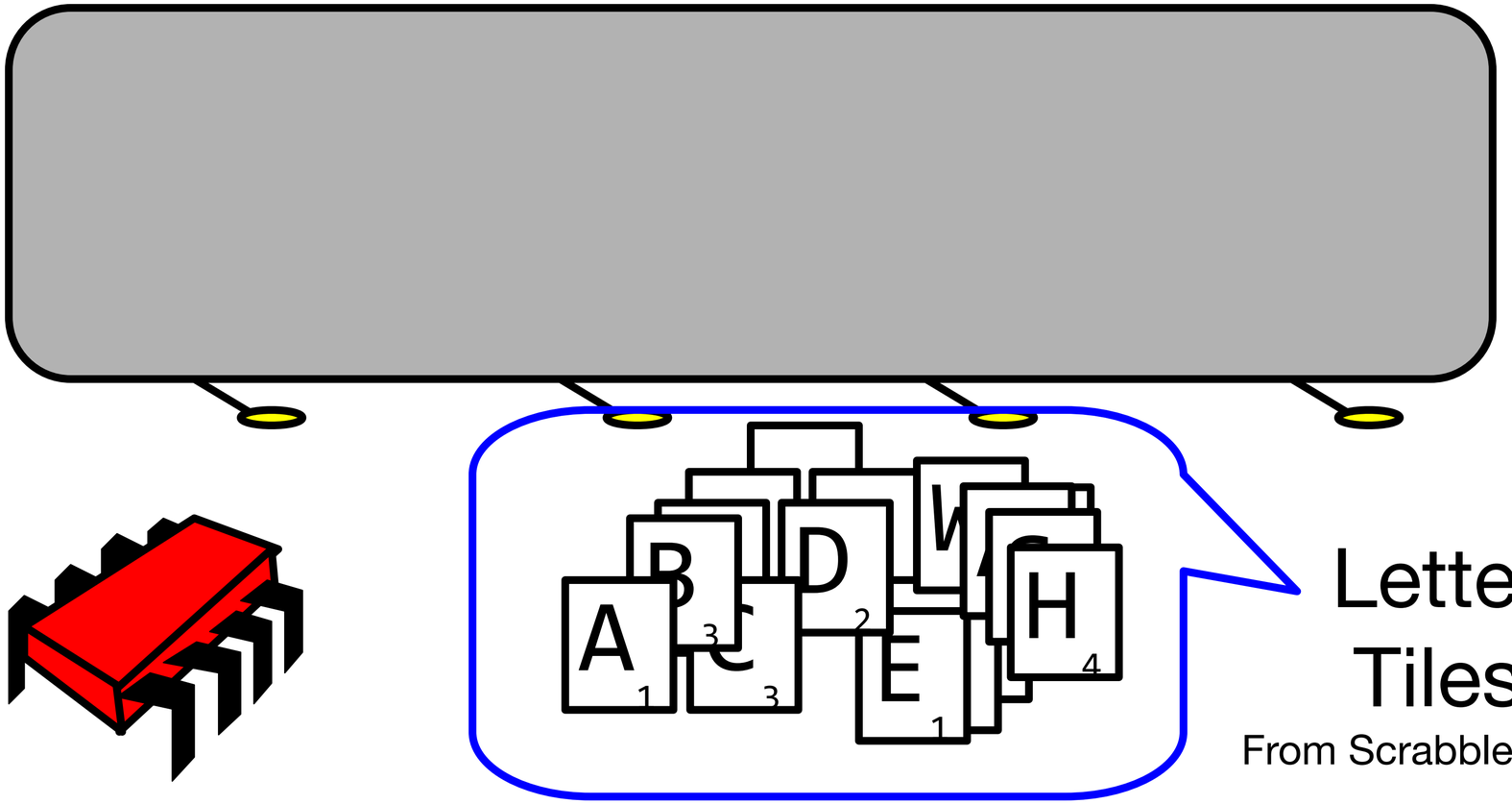
The Fable drags on …

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

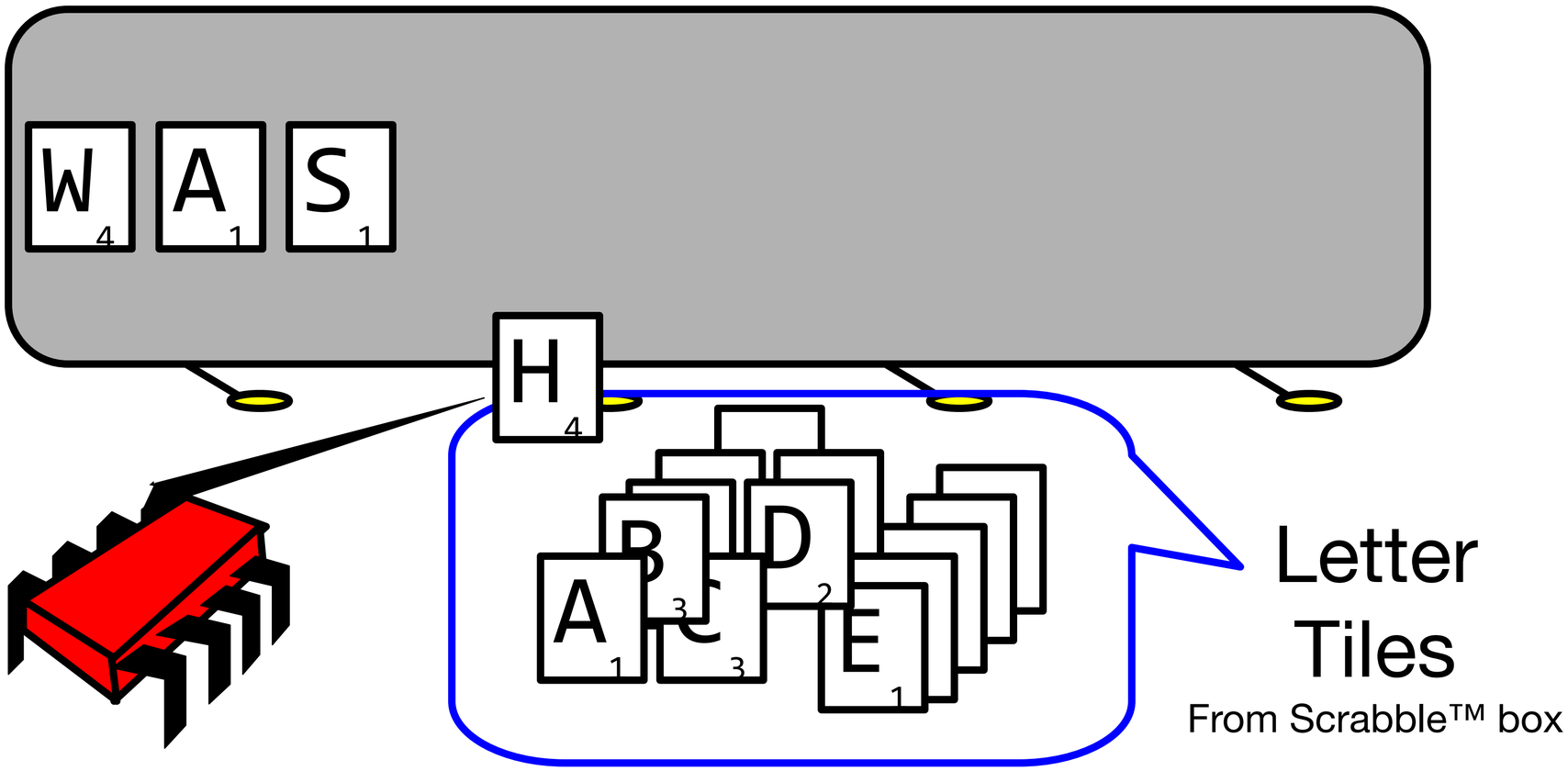
The Fable drags on …

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

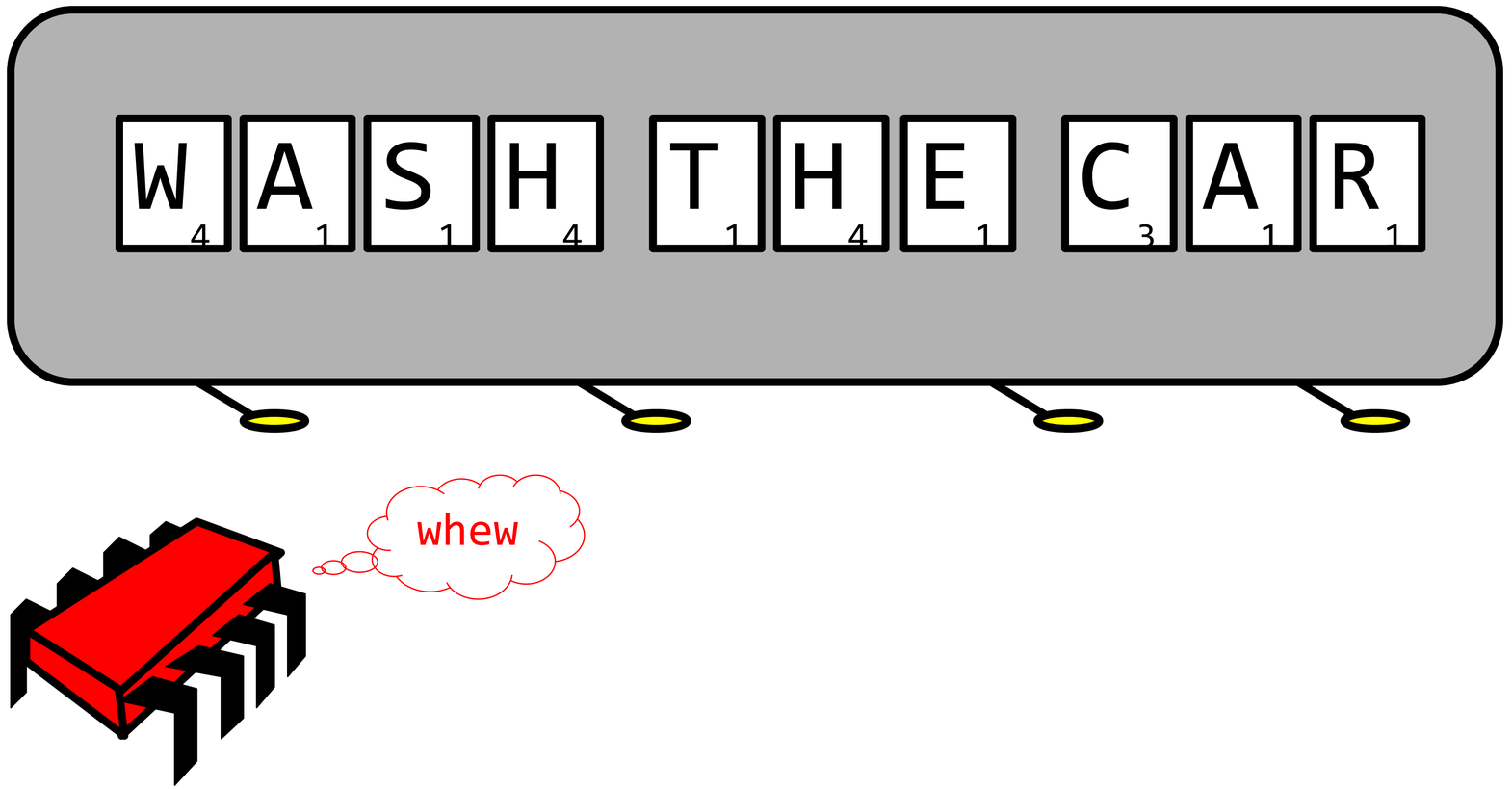
Billboards are Large



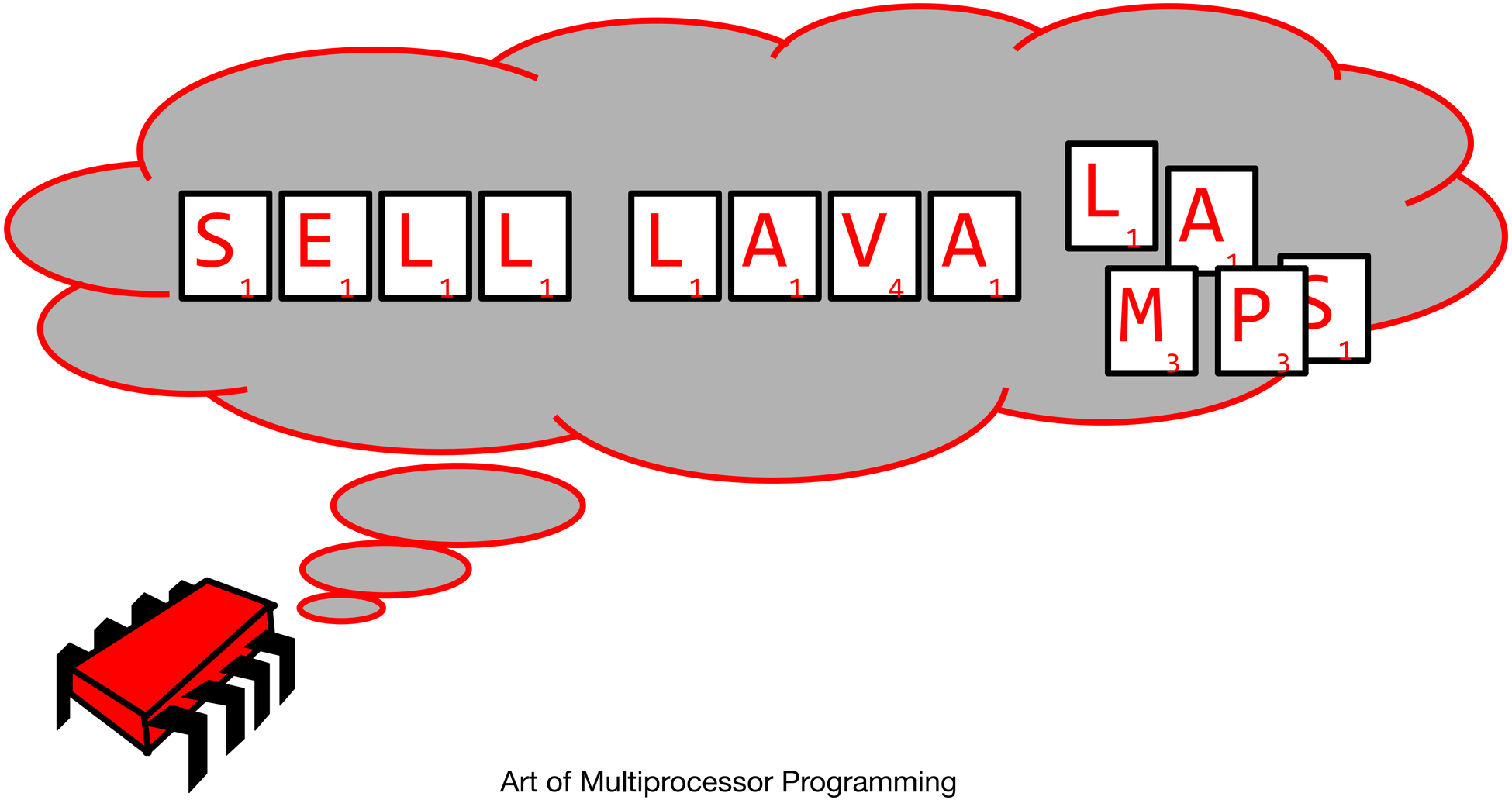
Write One Letter at a Time …



To post a message

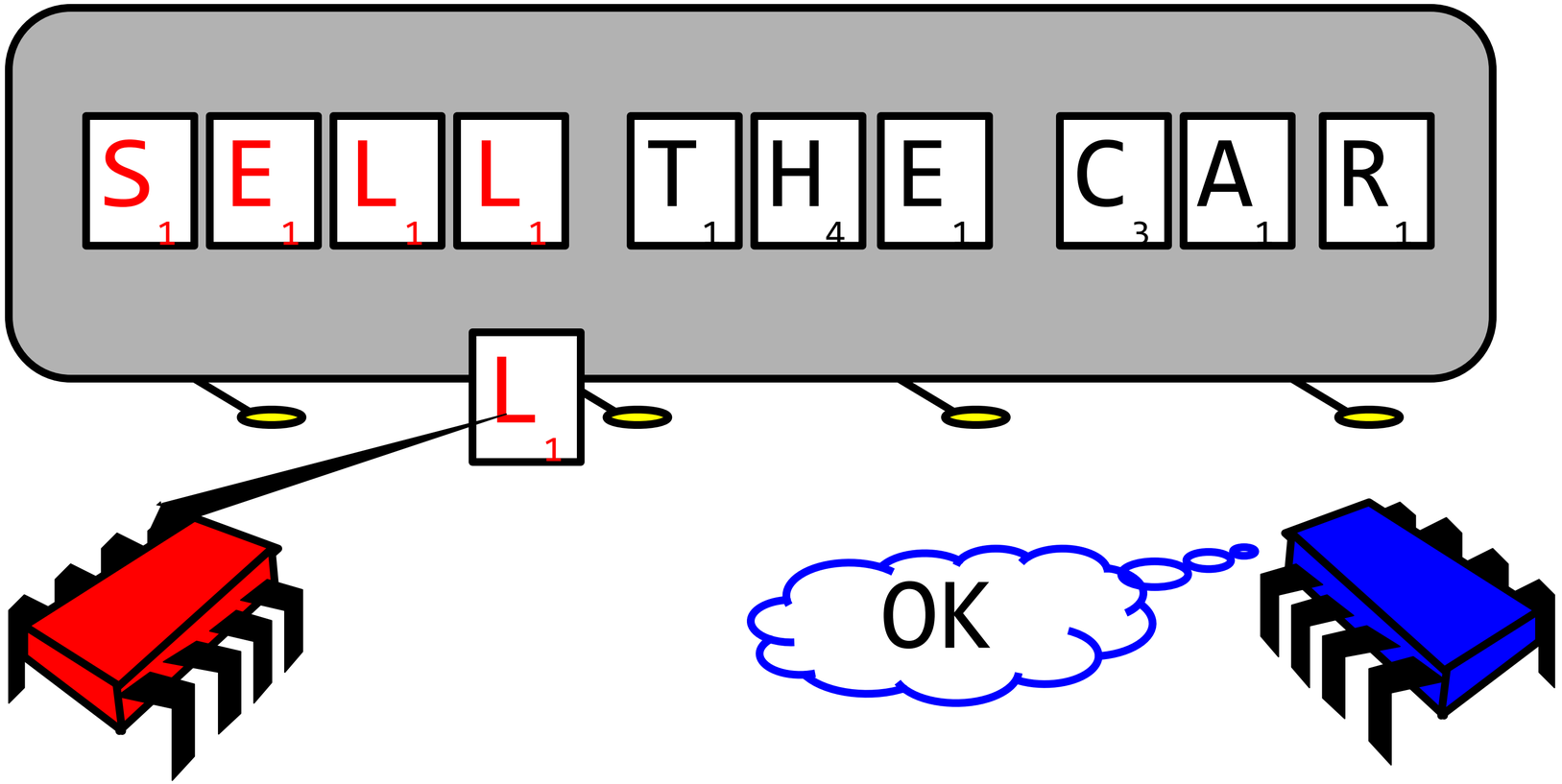


Let's send another message



Art of Multiprocessor Programming

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 (remember - the can!)

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

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S
- Serial portion of an application has disproportionate effect on performance gained by adding additional cores

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?

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

Example

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

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

Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

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

The Moral

- Making good use of our multiple processors (cores) means
- Finding ways to effectively parallelize our code
  - Minimize sequential parts
  - Reduce idle time in which threads **wait** without
- This will be a constant theme throughout the course!

Roadmap

- Weds: Mutual Exclusion - from a technical (not lochness monster) perspective
- HW1 out on Weds

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