# Processes & Threads

CS 475, Spring 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

*With material from Herlihy & Shavit, Art of Multiprocessor Programming*

# 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!**

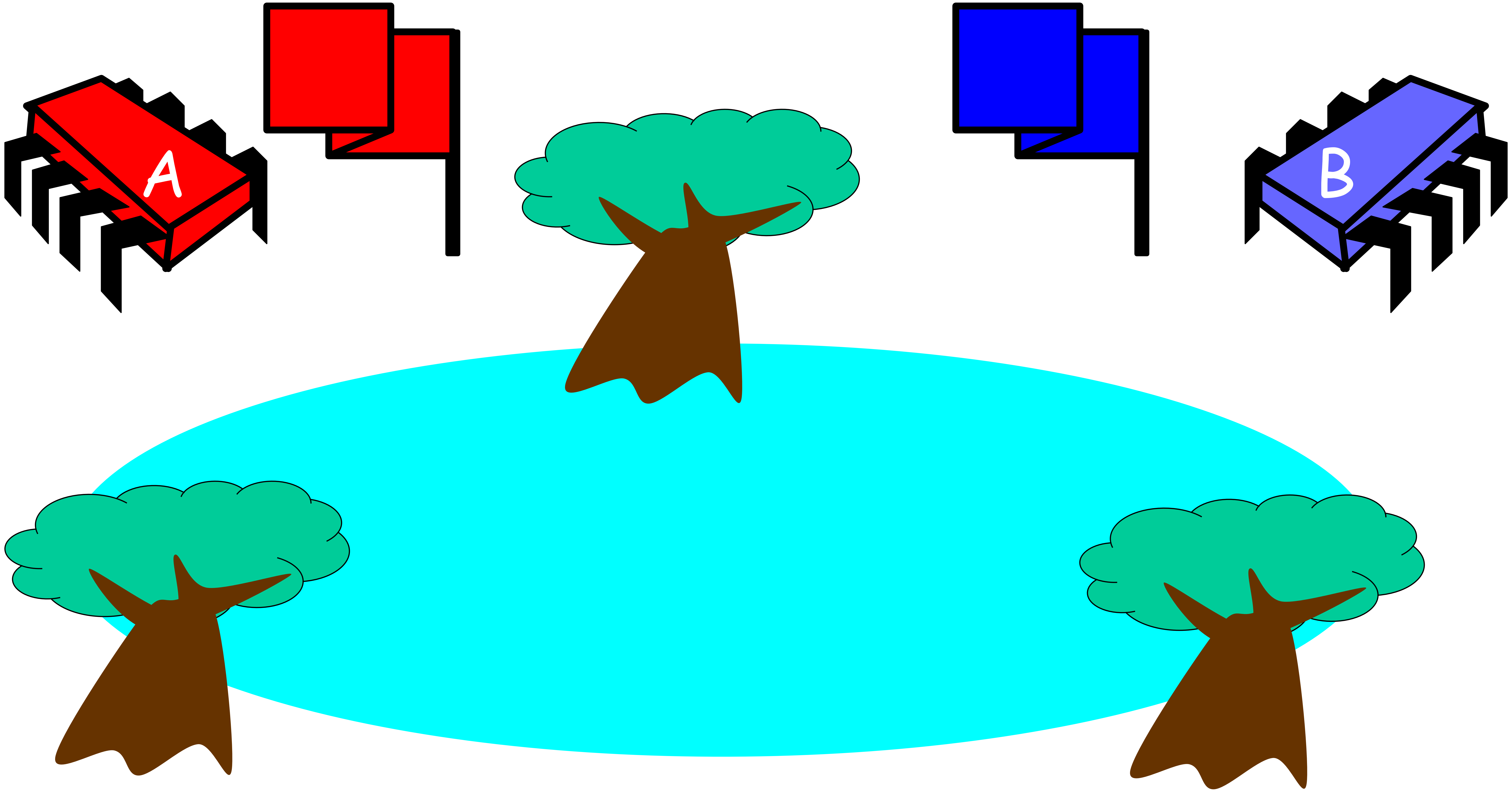
| Thread 1 | Thread 2 |
|---|---|
| increment() | increment() |
| read i = 0 | |
| write i = 1 | read i = 0 |
| | write i =1 |

**This is one possible interleaving**

# The Problem

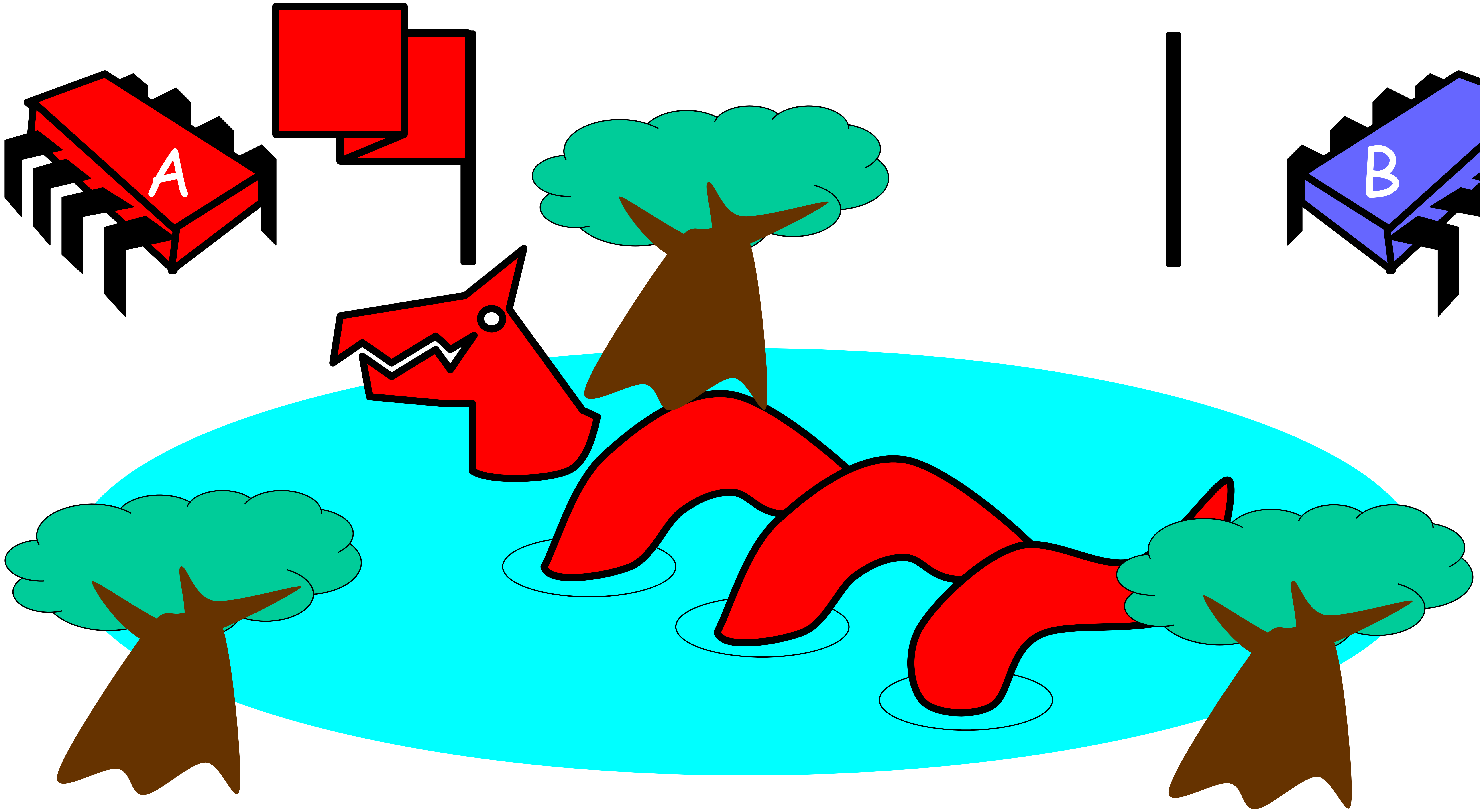

Art of Multiprocessor Programming

# Flag Protocol



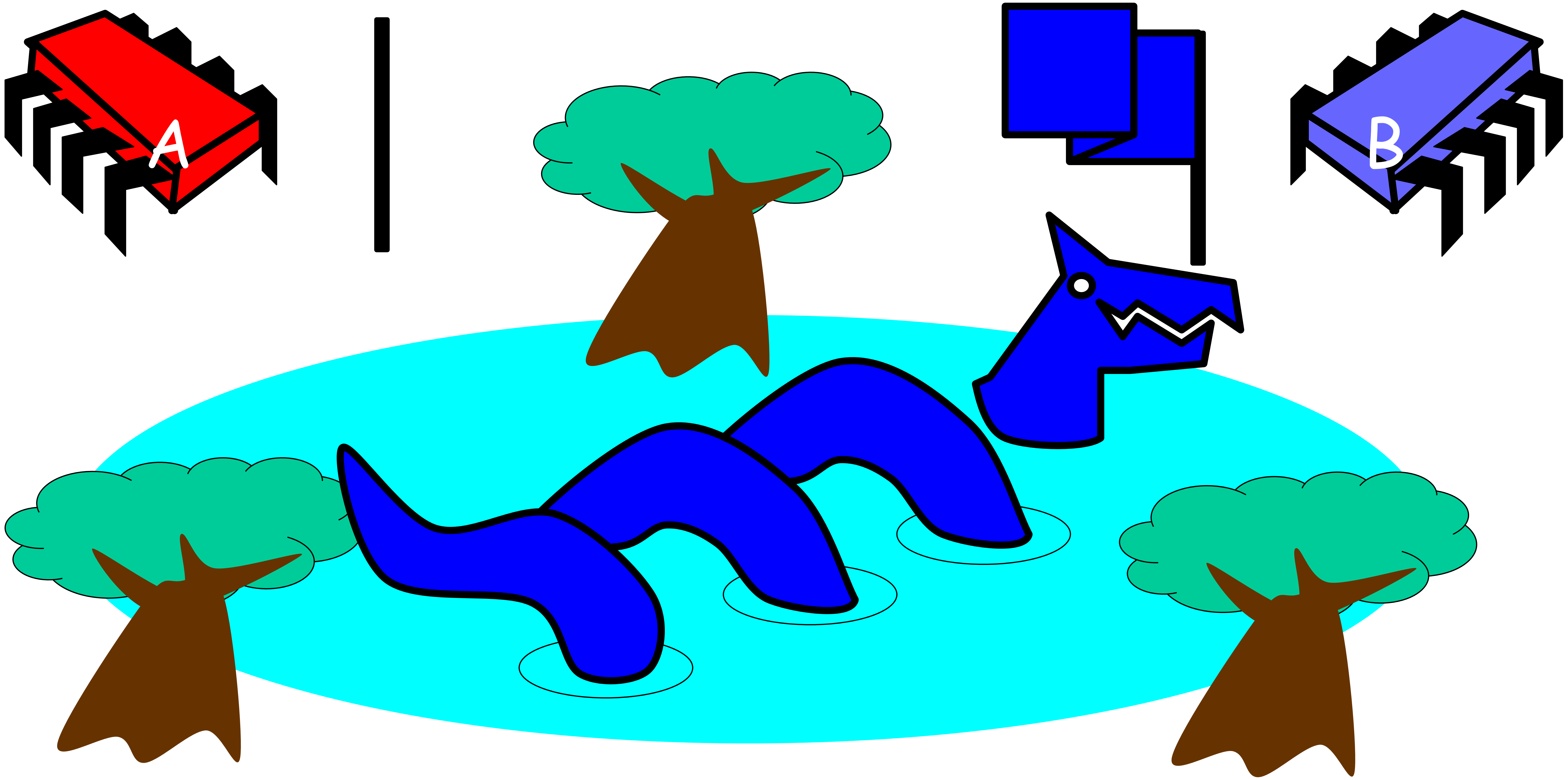

Art of Multiprocessor Programming

# Alice's Protocol (sort of)




Art of Multiprocessor Programming

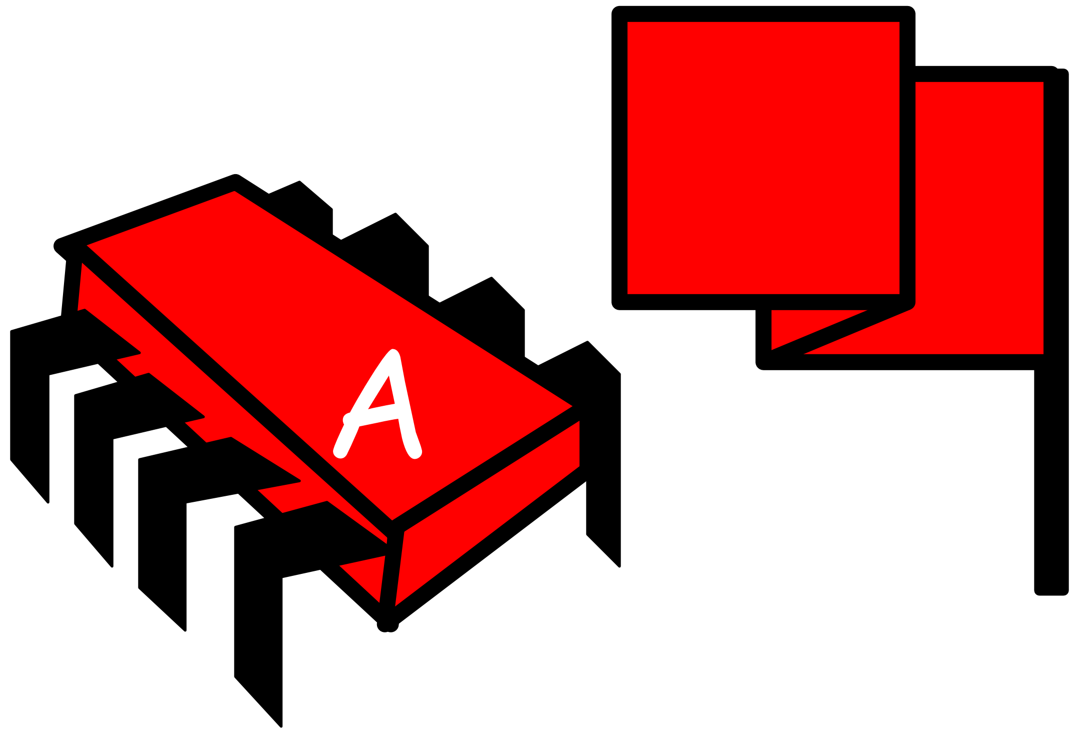
# Bob's Protocol (sort of)




Art of Multiprocessor Programming

# Alice's Protocol

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

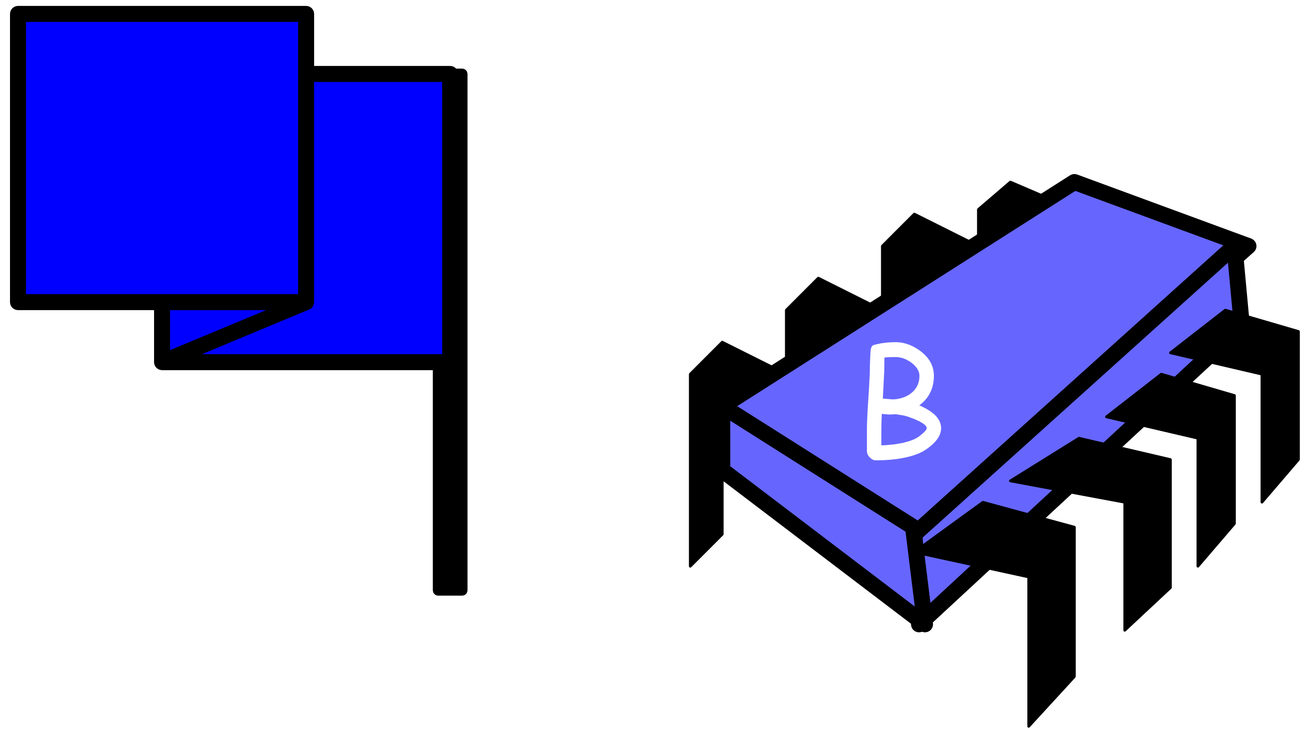



**After you!**

# Bob's Protocol

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

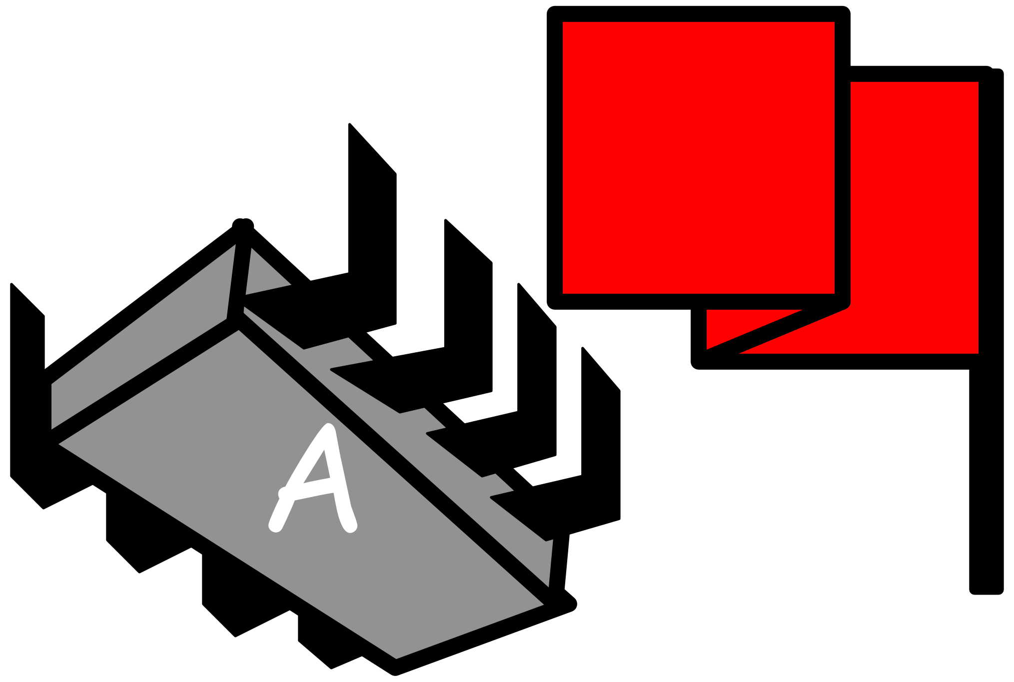
danger!




**No, no… after you!**

# Alice's Protocol

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

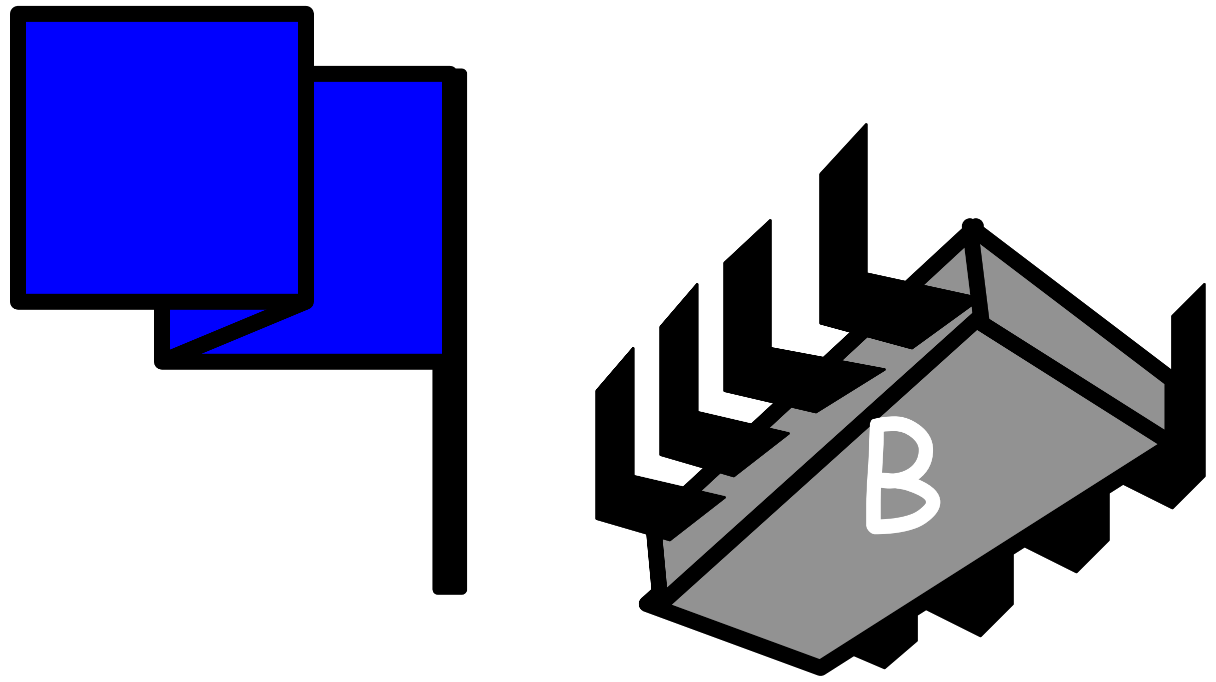



**After you!**

# Bob's Protocol

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

danger!




**No, no… after you!**

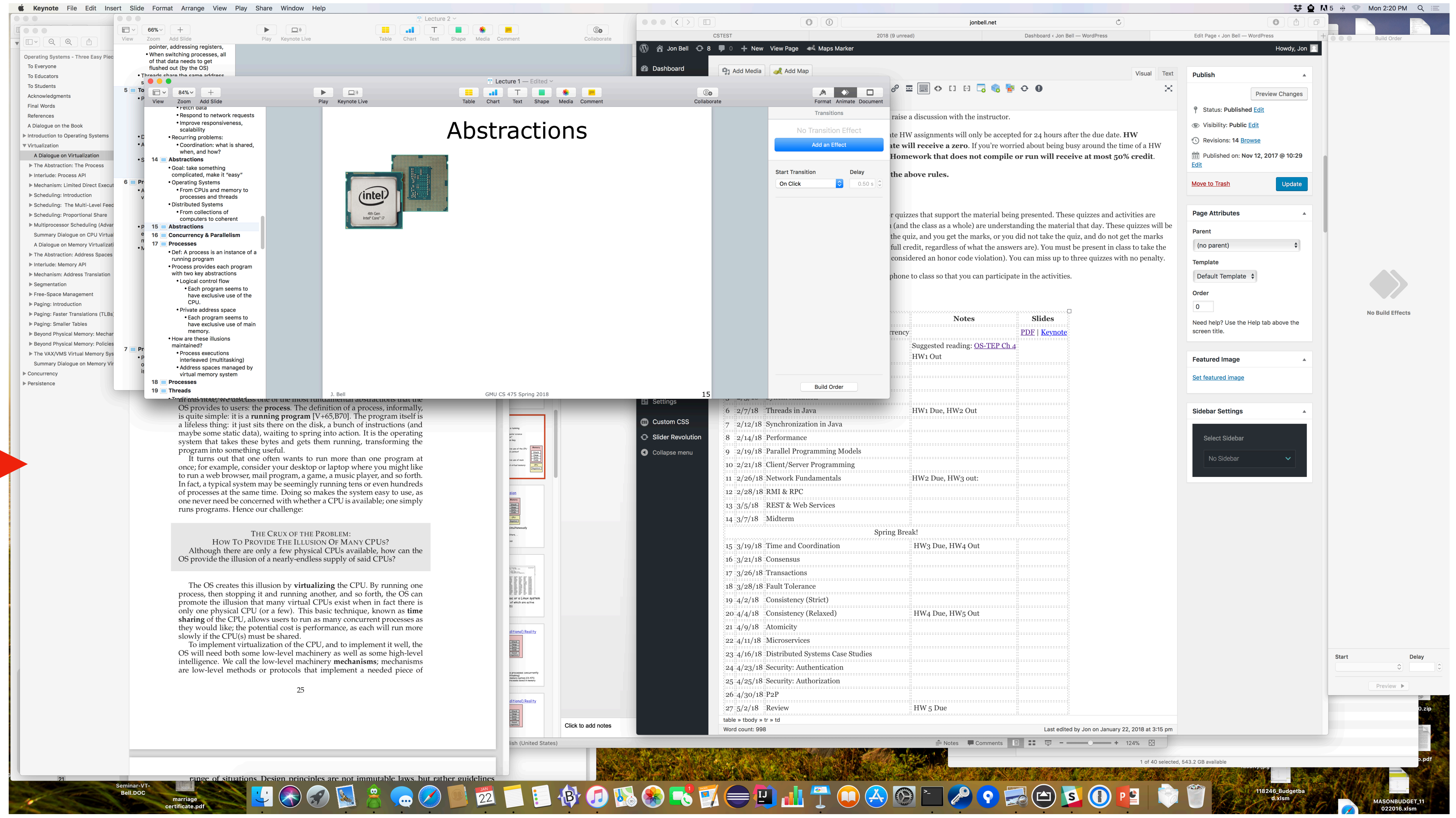
# Alice's Protocol

- 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

# Review: Abstractions

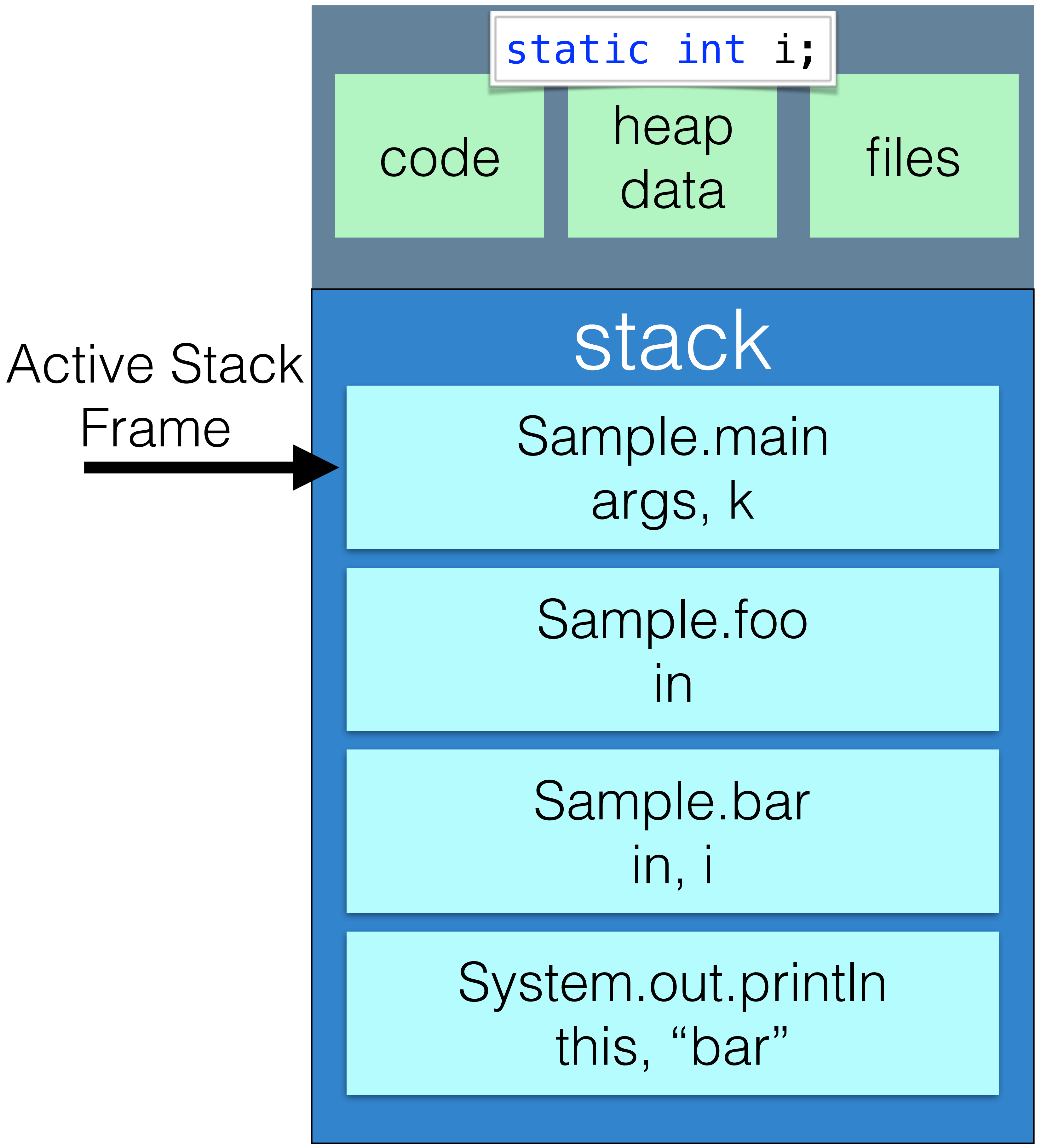
# Today

- What OS abstractions do we use for concurrency and parallelism?
  - Threads
  - Processes
- Reading: H&S 1.5
- Note: HW1 posted: https://www.jonbell.net/gmu-cs-475-spring-2019/homework-1/

# 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

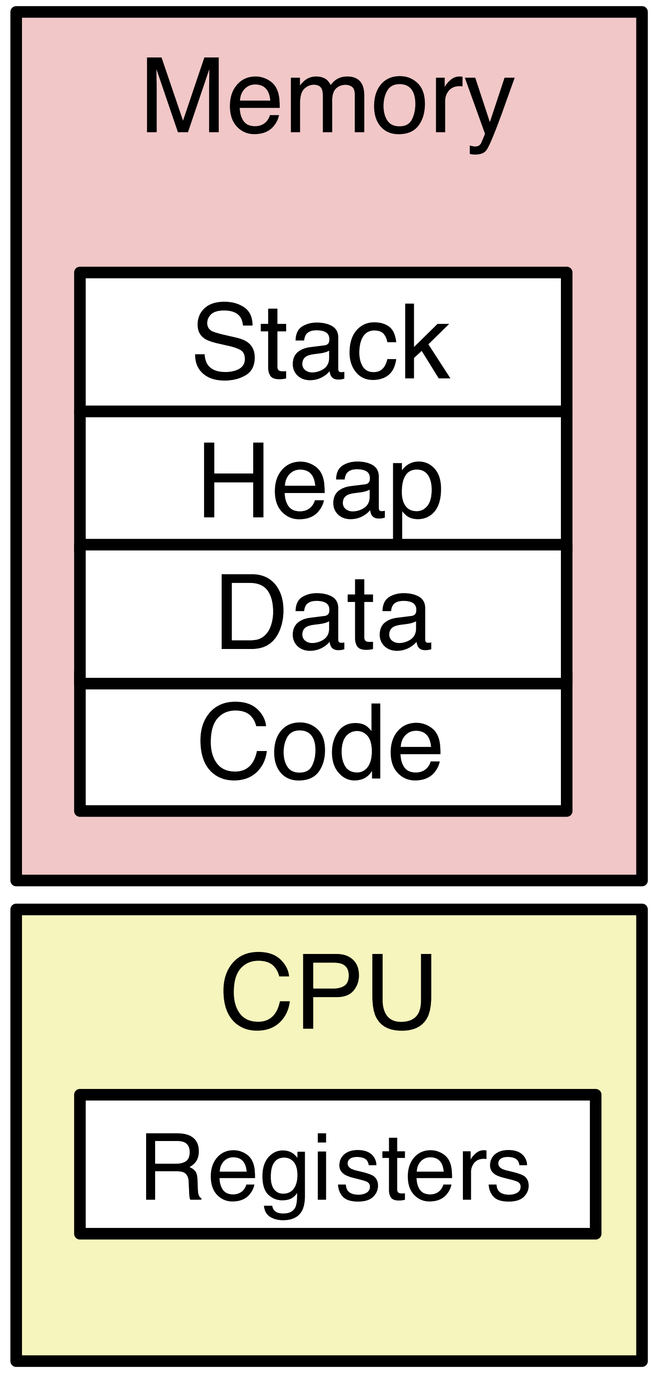
# Processes




```java
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");
    }
}
```

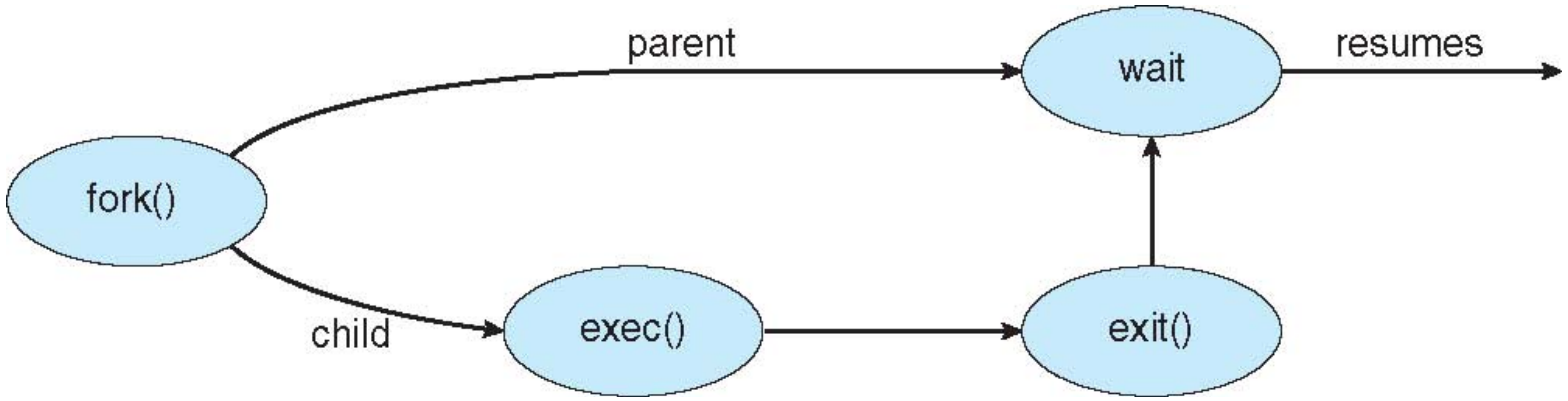
# Process Representation

- 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



# Creating Processes

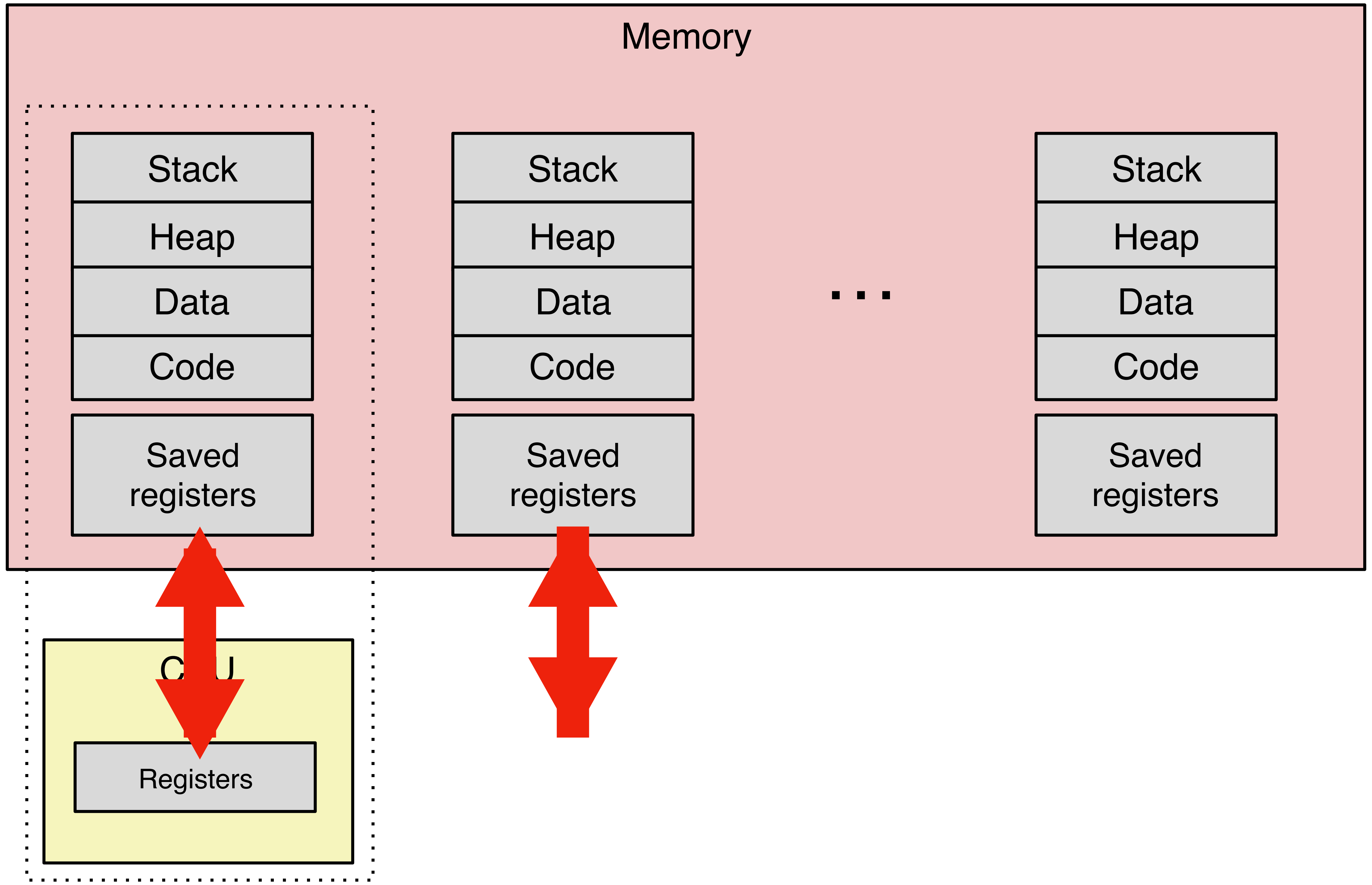
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program



# Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

# CPU Switching from Process to Process



# Interprocess Communication

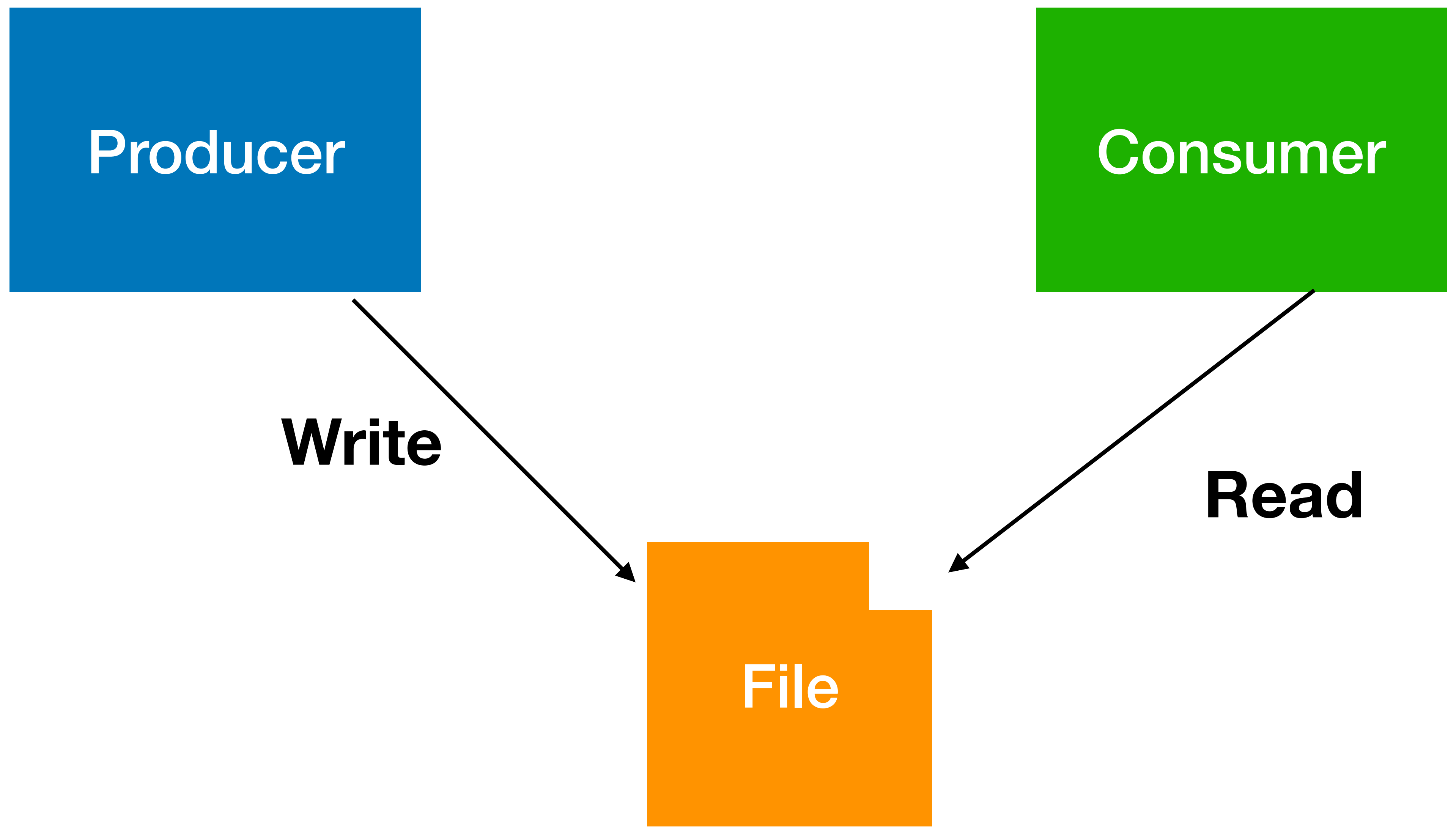
- We might want two processes to seriously work together
- For example:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Signals are very, very NOT sufficient for these purposes
- What we need is **interprocess communication (IPC)**

# Producer-Consumer Model

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size
- Producer writes to a buffer, consumer reads
- Buffer is just a chunk of memory

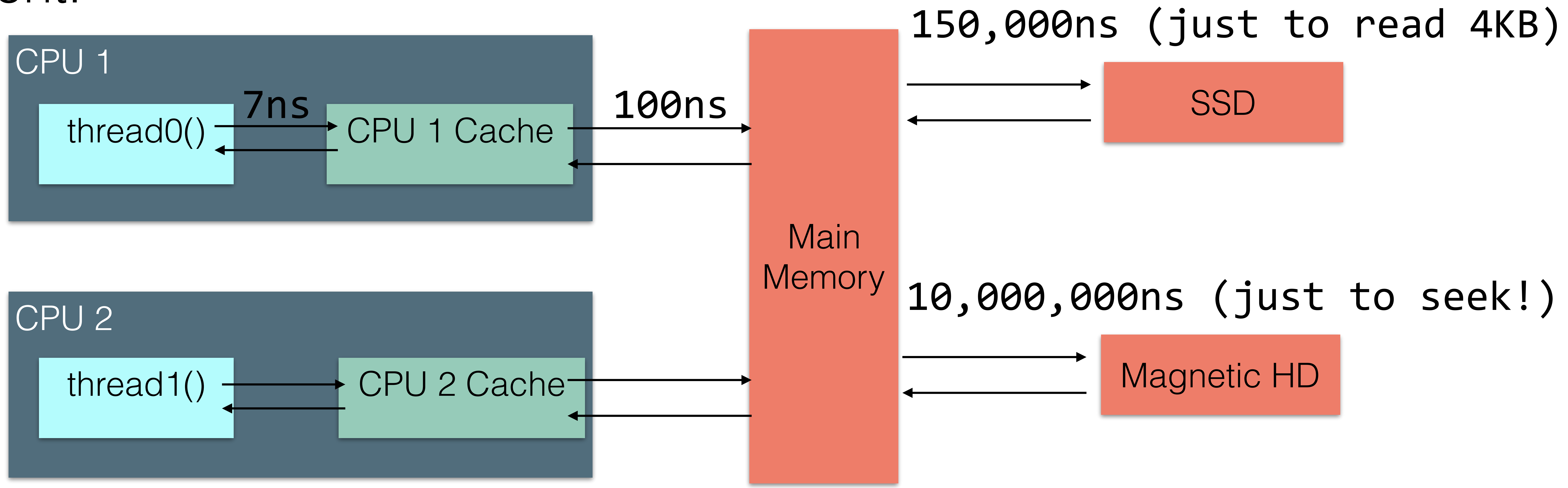
# Strawman IPC

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



# Strawman IPC

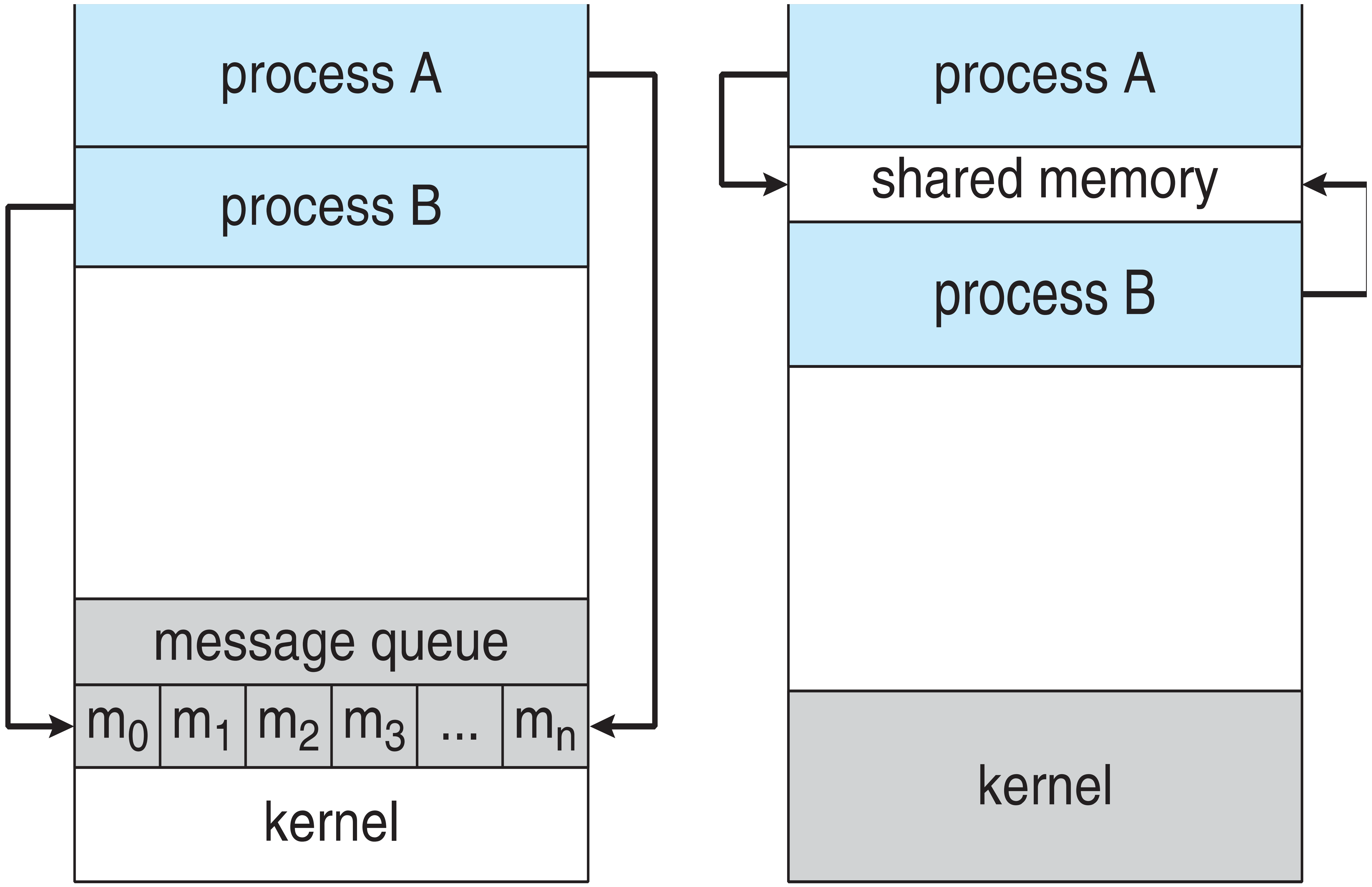
- Does it work? Yes
- Is it cumbersome (and perhaps error-prone)? Yes
  - What happens if consumer reads while producer is writing?
- Is it efficient?
  - No
  - Argument:



# 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**


Fig © Silberschatz, Galvin and Gagne

# Shared Memory

- As high performance as you can get
  - Each process directly reads/writes memory, which happens to be shared
- Can become confusing to program (correctly)
  - Which variables exactly are shared?
  - What happens if I copy a pointer to (non-shared) memory into shared memory?
  - What happens if producer/consumer read/write simultaneously?

# Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message)
  - receive(message)
- The message size is either fixed or variable
- Messaging system can be arbitrarily complex, adding additional features

# Message Passing

- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive
- On a single machine, this is usually done by creating a named mailbox (or "port")
- Key implementation questions:
  - Are sending and/or receiving blocking, or non-blocking?
  - Is there a message queue?

# Syncronous and Asynchronous

- Message passing may be either blocking or non-blocking
- **Blocking** is considered synchronous
  - Blocking send -- the sender is blocked until the message is received
  - Blocking receive -- the receiver is  blocked until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send -- the sender sends the message and continue
  - Non-blocking receive -- the receiver receives:
    - A valid message,  or
    - Null message
- Different combinations possible
  - E.g. both send and receive are blocking, only one, neither

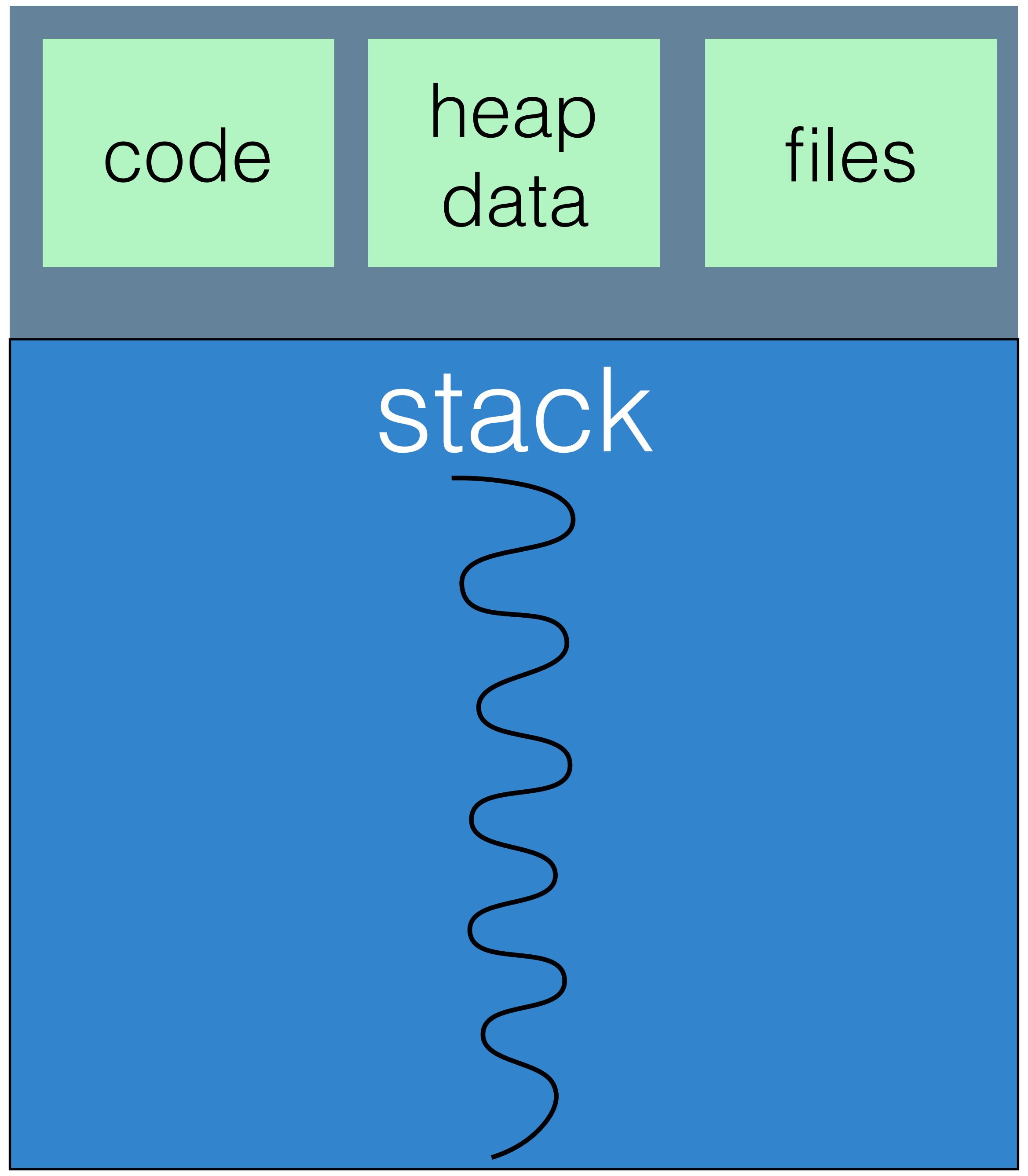
# Blocking Send (Synchronous)

# Non Blocking Send (Asynchronous)

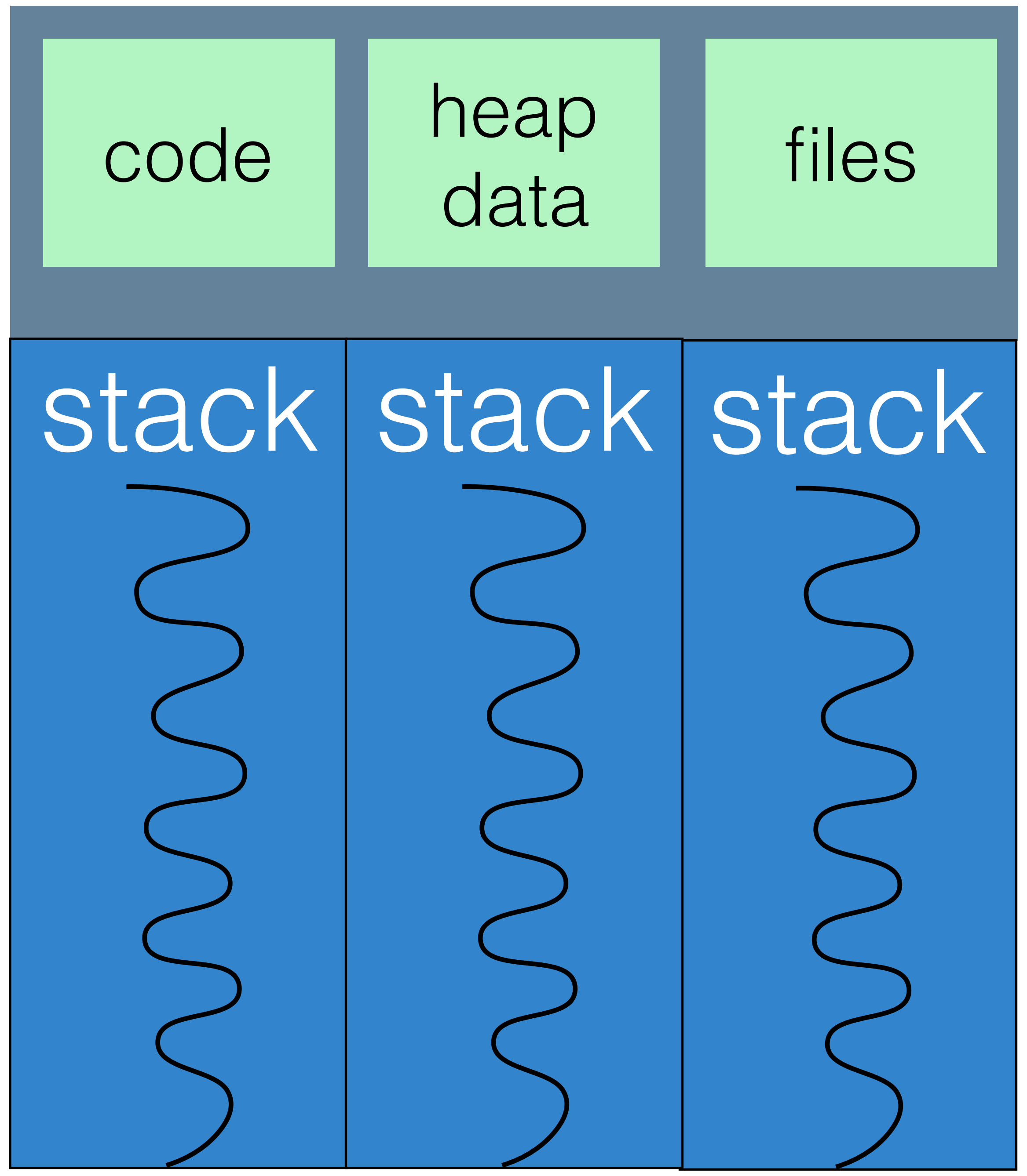
# Threads

- 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

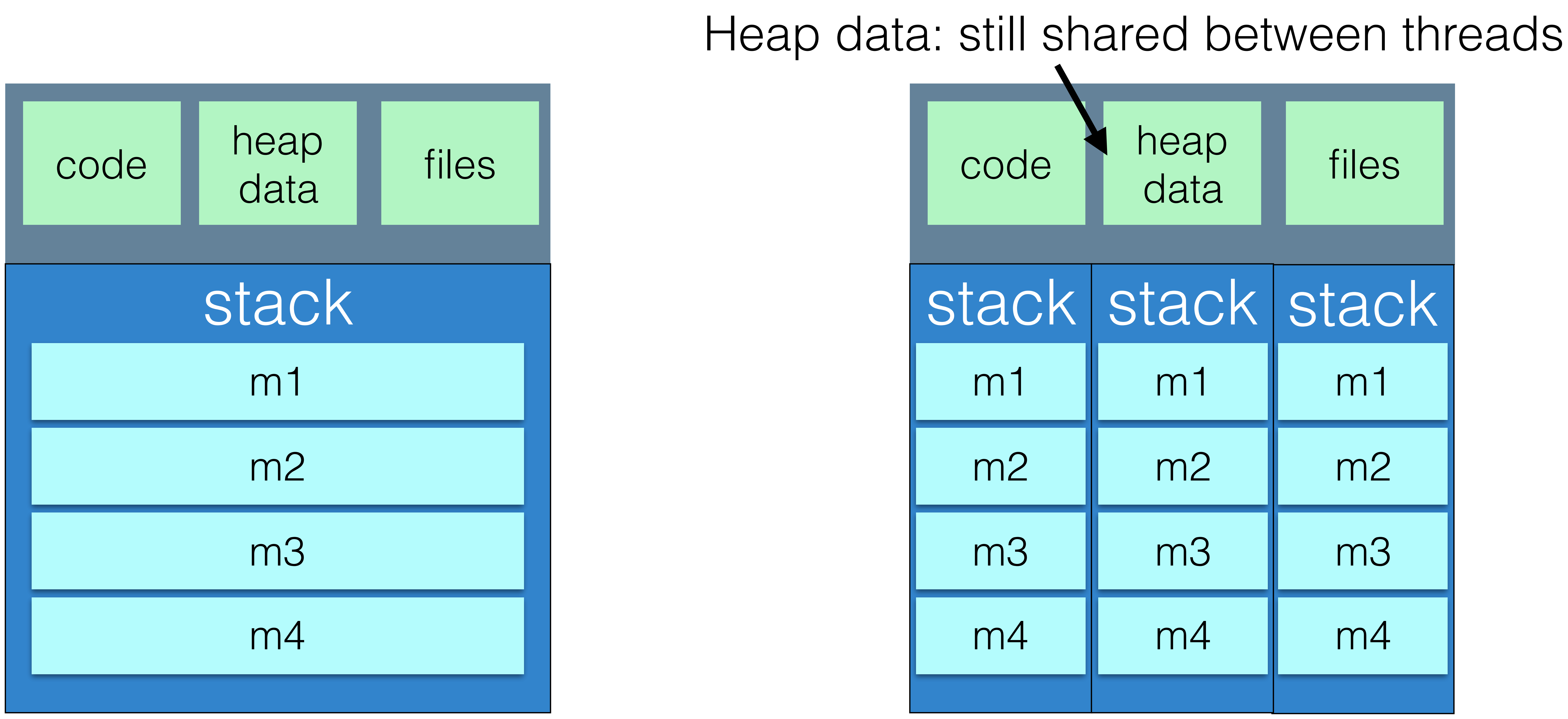



Multi-Threaded Process

# What do we use threads for?

- Run multiple tasks seemingly at once
  - Update UI
  - Fetch data
  - Respond to network requests
- Process creation: heavyweight, thread creation: lightweight
- Improve responsiveness, scalability
- Concurrency + Parallelism

# Threads: Memory View

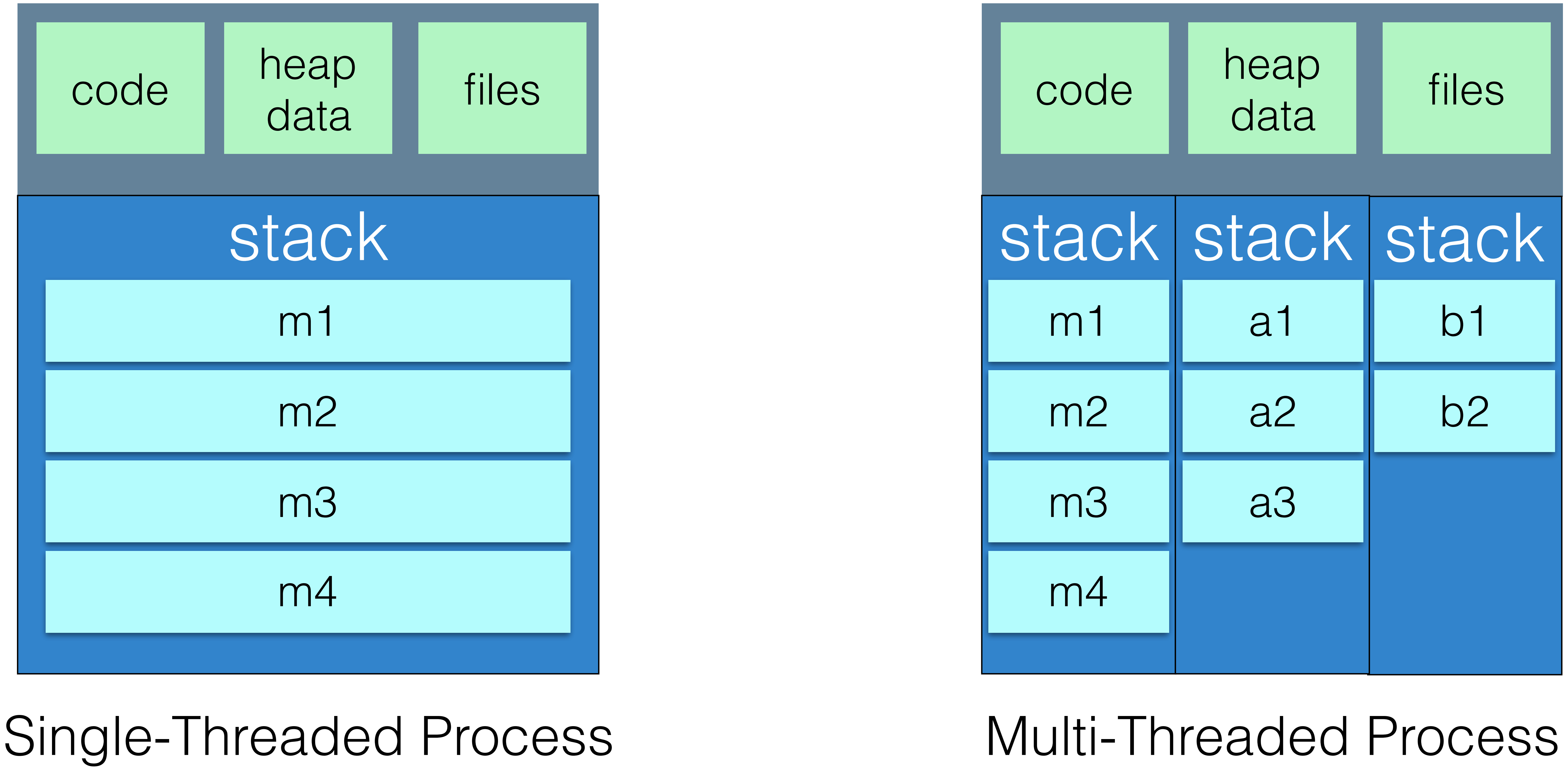



Single-Threaded Process

Multi-Threaded Process

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

# Threads: Memory View




**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;
}
```

**Thread attributes (usually NULL)**

**Thread arguments (void *p)**

**return value (void **p)**

# 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**

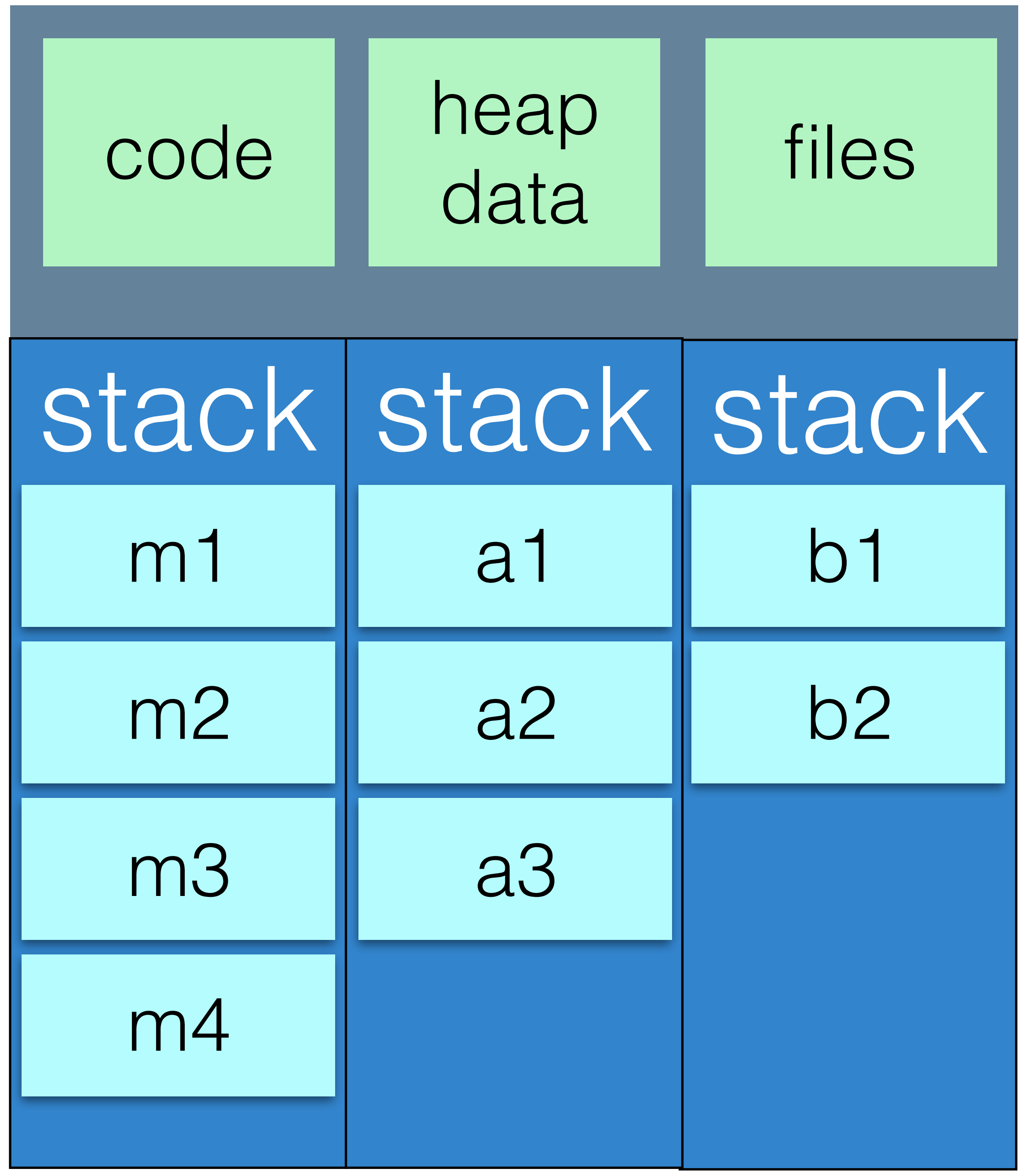
#2 Hello from main!
Hello from the thread!

# 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

# Splitting up the work

- The problem: What if we have thousands of tasks to do simultaneously, should we make a new thread for each?
  - No (lots of overhead, probably too many threads)
- The answer: think about work as **tasks** and not **threads**
  - Threads will magically appear to do your tasks
  - Tasks -> **Runnable** and **Callable** objects
  - **ExecutorService** handles taking tasks and running them

# Live Programming Example - ExecutorService

# Locking in Java

- Most locks are *reentrant*: if you hold it, and ask for it again, you don't have to wait (because you already have it)
- Basic primitives:
  - `synchronized{}`
  - `wait`
  - `notify`
- Plus…
  - `Lock` API… `lock.lock()`, `lock.unlock()`
  - The *preferred* way

# Synchronized methods in Java

```java
public synchronized static void increment()
{
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`

# Synchronized methods in Java

```java
public synchronized static void increment()
{
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`

```java
public synchronized static void incrementOther()
{
    j = j + 1;
}
```

Result: Before entering `incrementOther()`, thread gets a lock on the Class object of `incrementOther()`

**Problem?**

# Synchronized blocks in Java

- Can also use *any* object as that monitor

```
static Object someObject = new Object();
public static void increment()
{
    synchronized(someObject){
        i = i + 1;
    }
}
static Object someOtherObject = new Object();
public static void incrementOther()
{
    synchronized(someOtherObject){
        j = j + 1;
    }
}
```

Now, two different threads could call `increment()` and `incrementOther()` at the same time

# Java Lock API

- `Synchronized` gets messy: what happens when you need to synchronize many operations? What if we want more complicated locking?
- `ReentrantLock`: same semantics as synchronized

```java
static ReentrantLock lock = new ReentrantLock();
public static void increment()
{
    lock.lock();
    try{
        i = i + 1;
    } finally{
        lock.unlock();
    }
}
```

# Locking Granularity

- BIG design question in writing concurrent programs: how many locks should you have?
- Example: Distributed filesystem
  - It would be *correct* to block all clients from reading *any* file, when one client writes a file
  - However, this would not be performant at all!
  - It would be much better to instead lock on *individual files*
- *More locks -> more complicated semantics and tricky to avoid deadlocks, races*

# 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?

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

# 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
- Reminder: HW1 Out
  - https://www.jonbell.net/gmu-cs-475-spring-2019/homework-1