Rust

cs378

Chris Rossbach

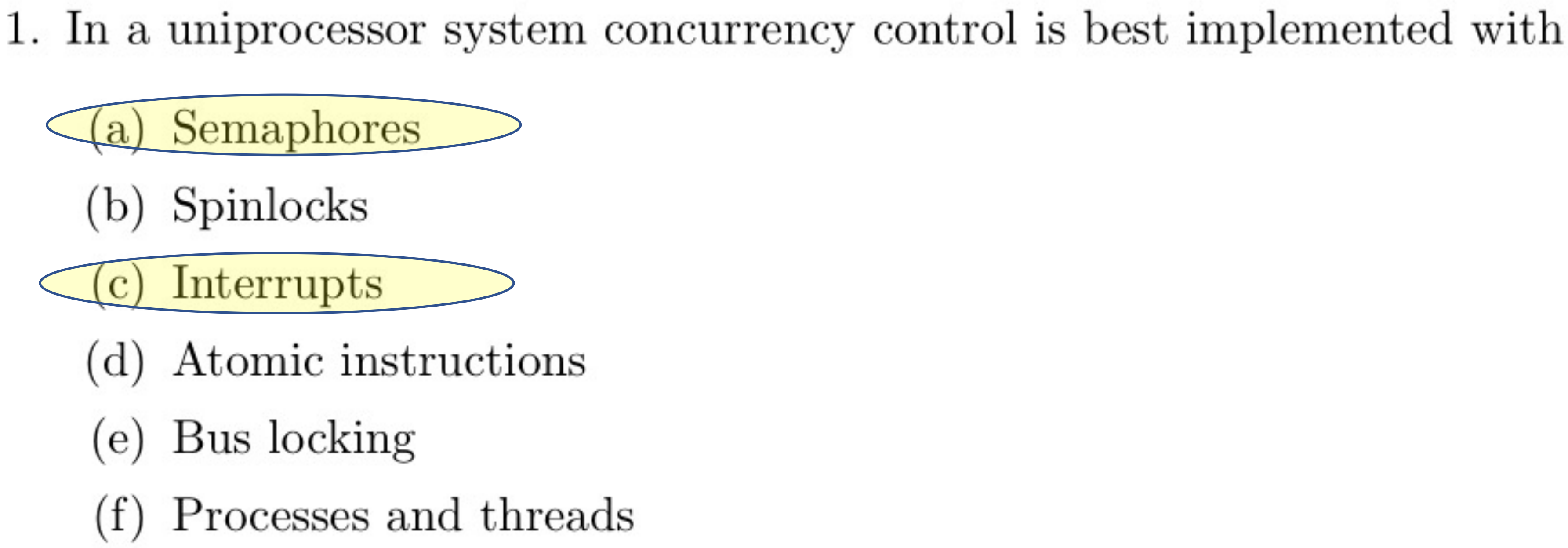
Outline

- Administrivia
  - Midterm 1 discussion
- Technical Agenda
  - Rust!
    - Overview
    - Decoupling Shared, Mutable, and State
    - Channels and Synchronization
  - Rust Lab Preview

Acknowledgements:

- https://www.slideshare.net/nikomatsakis/rust-concurrency-tutorial-2015-1202
- Thanks Nikolas Matsakis!

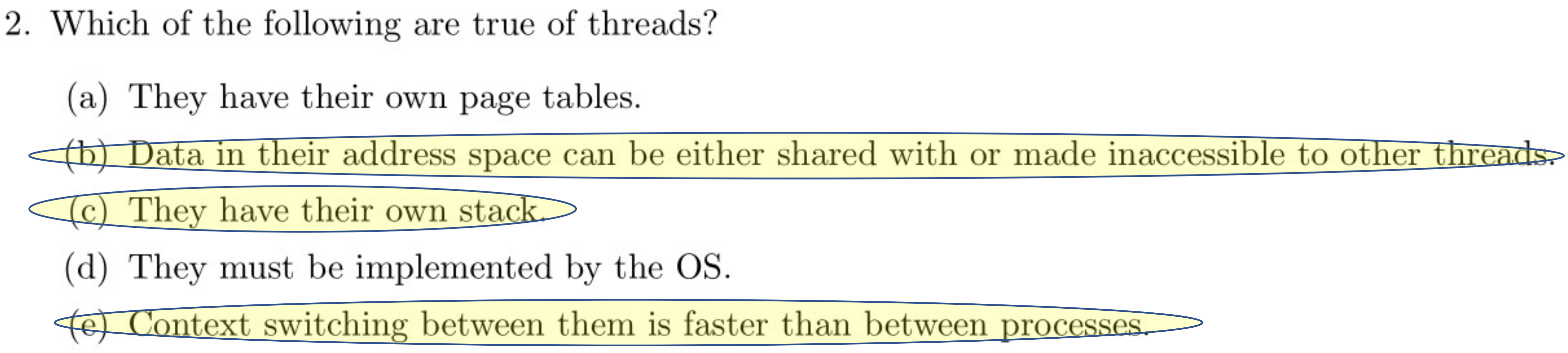
Exam Q*: Uniprocessors/Concurrency



1. In a uniprocessor system concurrency control is best implemented with

   [✓ (a) Semaphores]
   (b) Spinlocks
   [✓ (c) Interrupts]
   (d) Atomic instructions
   (e) Bus locking
   (f) Processes and threads

Exam Q*: Threads and Address Spaces



2. Which of the following are true of threads?

(a) They have their own page tables.

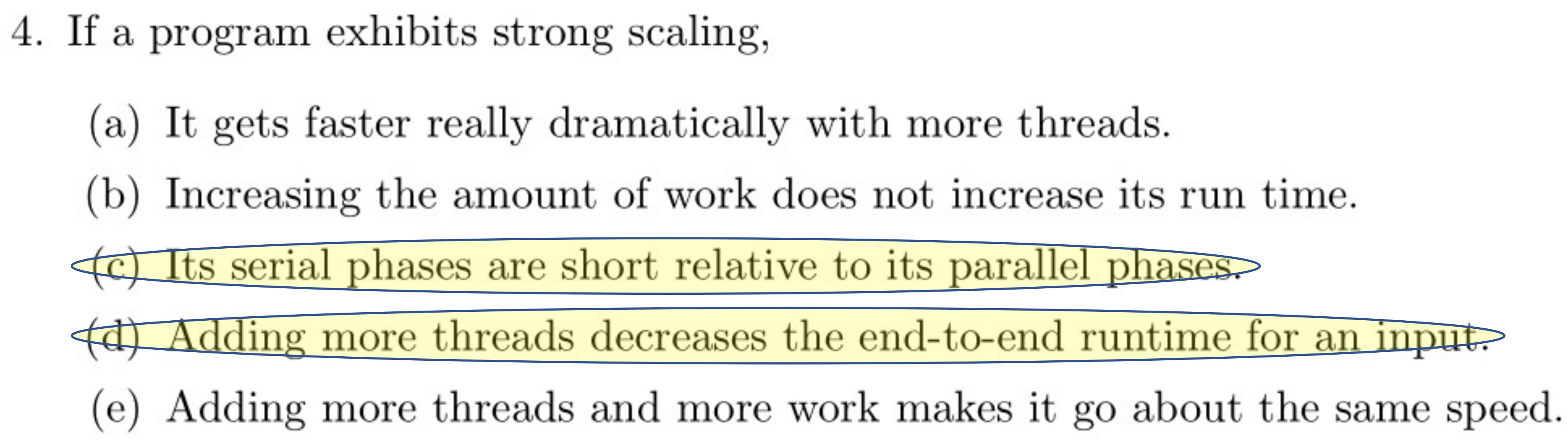
[✓ (b) Data in their address space can be either shared with or made inaccessible to other threads.]

[✓ (c) They have their own stack.]

(d) They must be implemented by the OS.

[✓ (e) Context switching between them is faster than between processes.]

Exam Q*: Scaling



4. If a program exhibits strong scaling,

   (a) It gets faster really dramatically with more threads.

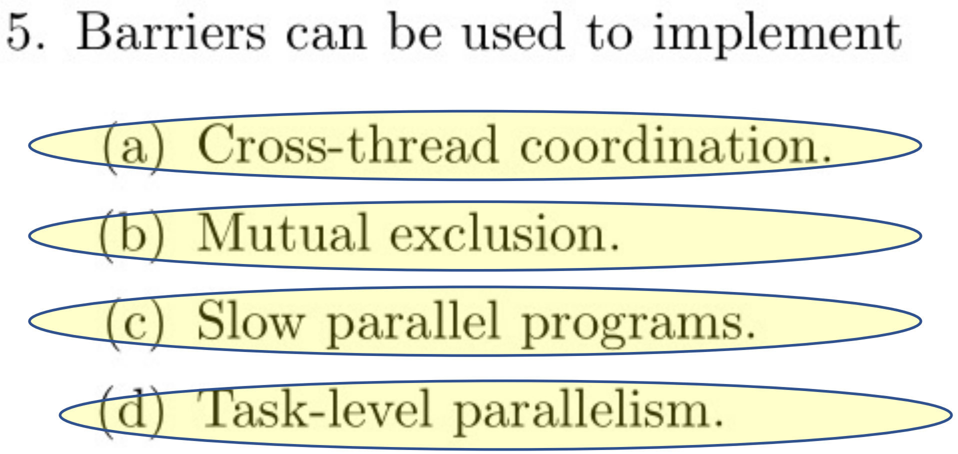
   (b) Increasing the amount of work does not increase its run time.

   [✓ (c) Its serial phases are short relative to its parallel phases.]

   [✓ (d) Adding more threads decreases the end-to-end runtime for an input.]

   (e) Adding more threads and more work makes it go about the same speed.

Exam Q*: Barrier generality



5. Barriers can be used to implement

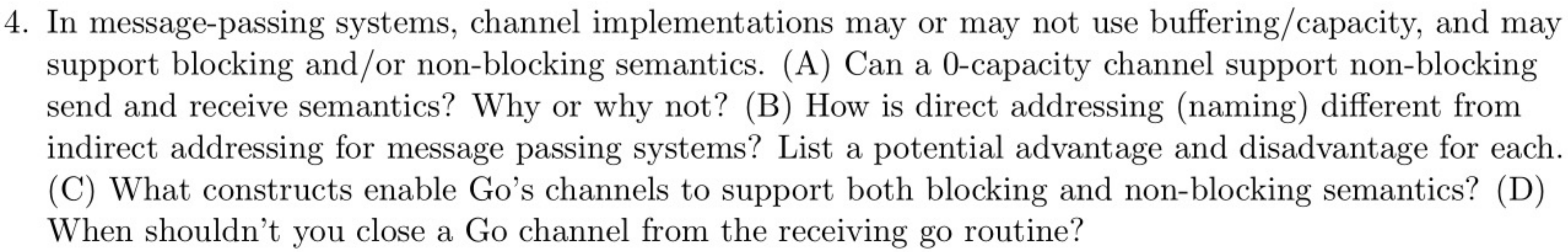
[✓ (a) Cross-thread coordination.]
[✓ (b) Mutual exclusion.]
[✓ (c) Slow parallel programs.]
[✓ (d) Task-level parallelism.]

Exam Q*: Formal properties and TM

Paraphrased: Do <safety, liveness, bounded wait, failure atomicity> suffice to define correctness for TM?

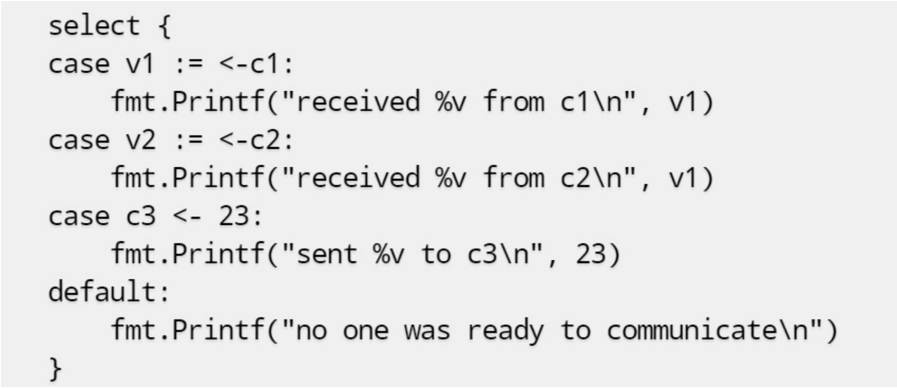
- The point: ***TM can violate single-writer invariant***
- Not the point: ***ACID***

Exam Q*: CSP models and Go



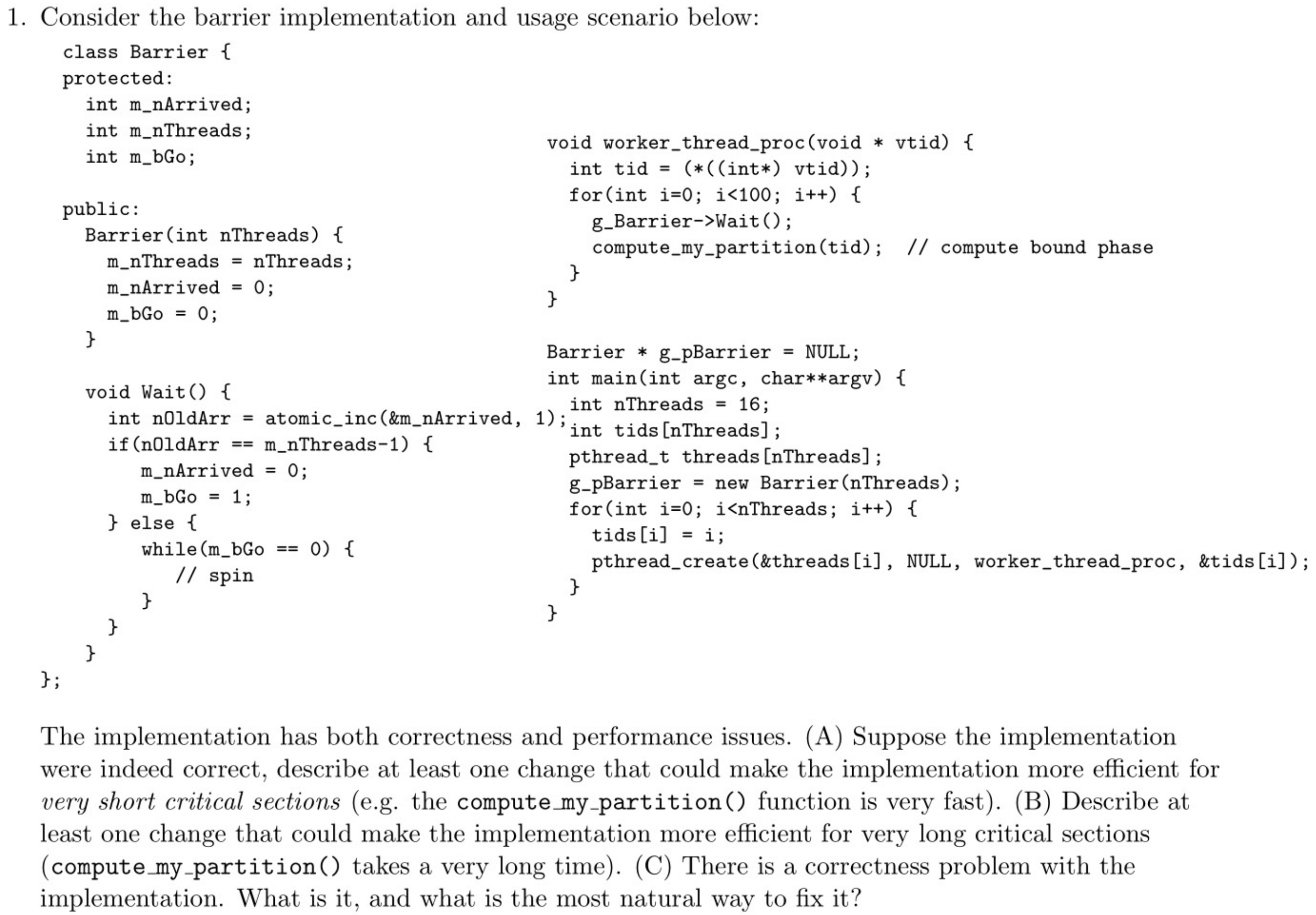
4. In message-passing systems, channel implementations may or may not use buffering/capacity, and may support blocking and/or non-blocking semantics. (A) Can a 0-capacity channel support non-blocking send and receive semantics? Why or why not? (B) How is direct addressing (naming) different from indirect addressing for message passing systems? List a potential advantage and disadvantage for each. (C) What constructs enable Go's channels to support both blocking and non-blocking semantics? (D) When shouldn't you close a Go channel from the receiving go routine?

- A) In general no, but receiver can poll
- C) Select!



```
select {
case v1 := <-c1:
    fmt.Printf("received %v from c1\n", v1)
case v2 := <-c2:
    fmt.Printf("received %v from c2\n", v1)
case c3 <- 23:
    fmt.Printf("sent %v to c3\n", 23)
default:
    fmt.Printf("no one was ready to communicate\n")
}
```

Exam Q: Barriers



1. Consider the barrier implementation and usage scenario below:

```
class Barrier {
protected:
  int m_nArrived;
  int m_nThreads;
  int m_bGo;

public:
  Barrier(int nThreads) {
    m_nThreads = nThreads;
    m_nArrived = 0;
    m_bGo = 0;
  }

  void Wait() {
    int nOldArr = atomic_inc(&m_nArrived, 1);
    if(nOldArr == m_nThreads-1) {
       m_nArrived = 0;
       m_bGo = 1;
    } else {
       while(m_bGo == 0) {
          // spin
       }
    }
  }
};
```

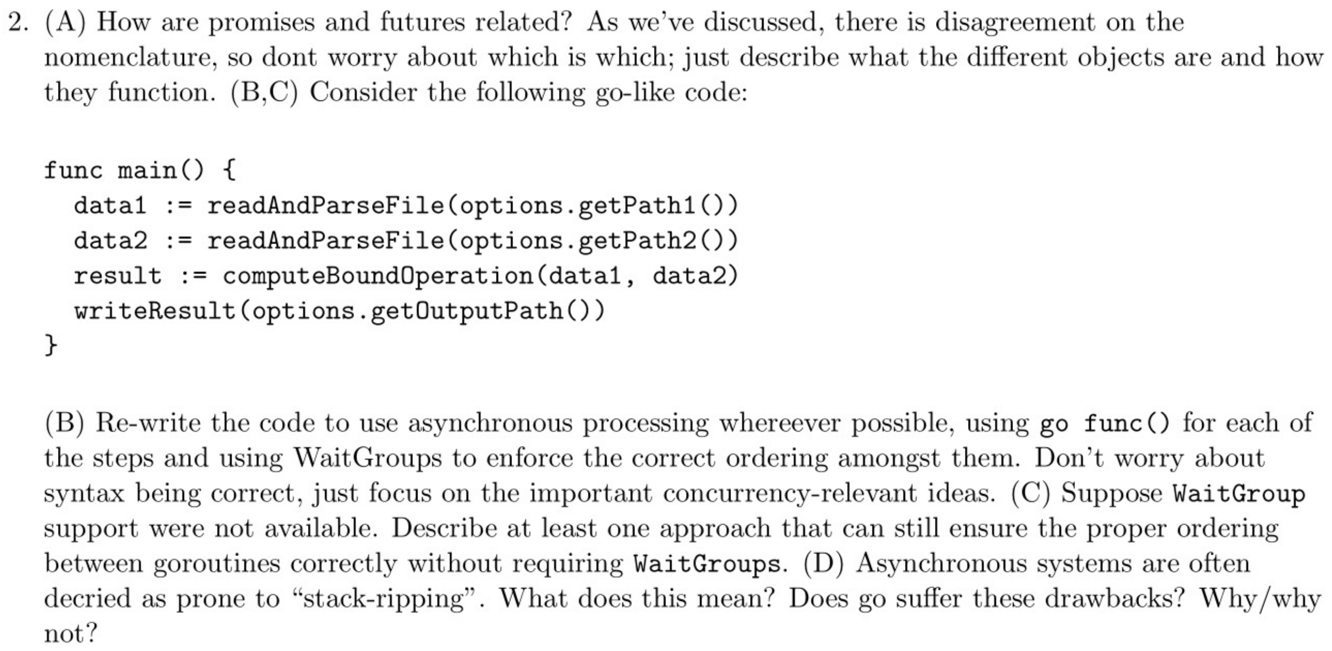
```
void worker_thread_proc(void * vtid) {
  int tid = (*((int*) vtid));
  for(int i=0; i<100; i++) {
    g_Barrier->Wait();
    compute_my_partition(tid);  // compute bound phase
  }
}

Barrier * g_pBarrier = NULL;
int main(int argc, char**argv) {
  int nThreads = 16;
  int tids[nThreads];
  pthread_t threads[nThreads];
  g_pBarrier = new Barrier(nThreads);
  for(int i=0; i<nThreads; i++) {
    tids[i] = i;
    pthread_create(&threads[i], NULL, worker_thread_proc, &tids[i]);
  }
}
```

The implementation has both correctness and performance issues. (A) Suppose the implementation were indeed correct, describe at least one change that could make the implementation more efficient for *very short critical sections* (e.g. the `compute_my_partition()` function is very fast). (B) Describe at least one change that could make the implementation more efficient for very long critical sections (`compute_my_partition()` takes a very long time). (C) There is a correctness problem with the implementation. What is it, and what is the most natural way to fix it?

- A) spin on local go flag
- B) some kind of blocking
- C) barrier doesn't reset (8), some strategy to make it reset (4)

Exam Q*: P+F



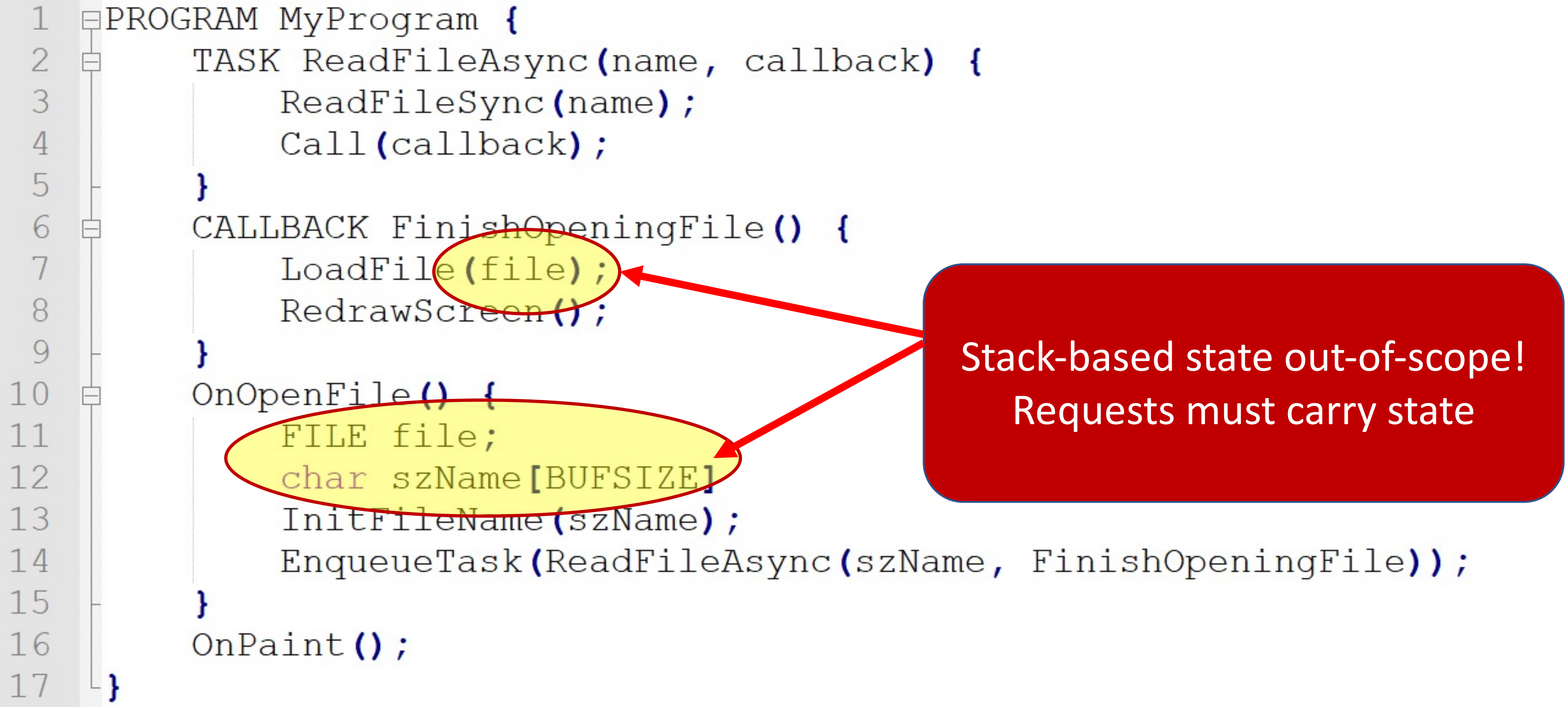
2. (A) How are promises and futures related? As we've discussed, there is disagreement on the nomenclature, so dont worry about which is which; just describe what the different objects are and how they function. (B,C) Consider the following go-like code:

```
func main() {
  data1 := readAndParseFile(options.getPath1())
  data2 := readAndParseFile(options.getPath2())
  result := computeBoundOperation(data1, data2)
  writeResult(options.getOutputPath())
}
```

(B) Re-write the code to use asynchronous processing whereever possible, using `go func()` for each of the steps and using WaitGroups to enforce the correct ordering amongst them. Don't worry about syntax being correct, just focus on the important concurrency-relevant ideas. (C) Suppose `WaitGroup` support were not available. Describe at least one approach that can still ensure the proper ordering between goroutines correctly without requiring `WaitGroups`. (D) Asynchronous systems are often decried as prone to "stack-ripping". What does this mean? Does go suffer these drawbacks? Why/why not?

- A) something about futures and promises
- B) pretty much anything with go func()
- C) Channels!
- D) Stack-ripping → some creative responses
  - (next slide)

Stack-Ripping

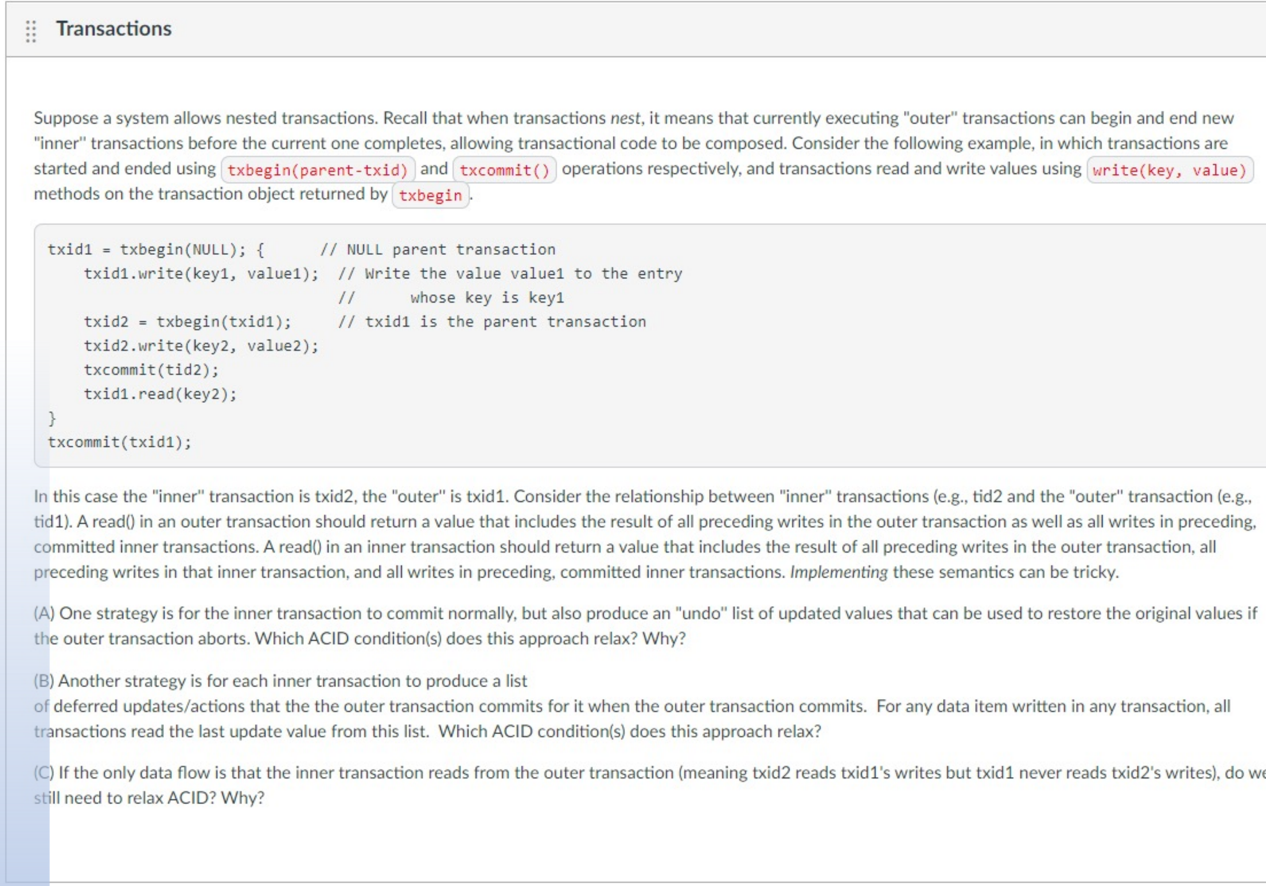


```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {
        FILE file;
        char szName[BUFSIZE]
        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-based state out-of-scope!
Requests must carry state

Exam Q*: Transactions

- A) Isolation, Atomicity, Durability
  - A) ***I***: other tx see “in-flight” state
  - A) ***A***: some of outer is available without all being available
  - A) ***D***: other tx see state that rolls back
- B) Isolation – all txs see writes of deferred actions (text is subtle)
  - B) Not ***C*** – all txs see writes in order
- C) No relaxation required
  - data only flows outer → inner
  - no uncommitted inner writes observed



Transactions

Suppose a system allows nested transactions. Recall that when transactions *nest*, it means that currently executing "outer" transactions can begin and end new "inner" transactions before the current one completes, allowing transactional code to be composed. Consider the following example, in which transactions are started and ended using `txbegin(parent-txid)` and `txcommit()` operations respectively, and transactions read and write values using `write(key, value)` methods on the transaction object returned by `txbegin`.

```
txid1 = txbegin(NULL); {      // NULL parent transaction
    txid1.write(key1, value1);  // Write the value value1 to the entry
                                //      whose key is key1
    txid2 = txbegin(txid1);     // txid1 is the parent transaction
    txid2.write(key2, value2);
    txcommit(tid2);
    txid1.read(key2);
}
txcommit(txid1);
```

In this case the "inner" transaction is txid2, the "outer" is txid1. Consider the relationship between "inner" transactions (e.g., tid2 and the "outer" transaction (e.g., tid1). A read() in an outer transaction should return a value that includes the result of all preceding writes in the outer transaction as well as all writes in preceding, committed inner transactions. A read() in an inner transaction should return a value that includes the result of all preceding writes in the outer transaction, all preceding writes in that inner transaction, and all writes in preceding, committed inner transactions. *Implementing* these semantics can be tricky.

(A) One strategy is for the inner transaction to commit normally, but also produce an "undo" list of updated values that can be used to restore the original values if the outer transaction aborts. Which ACID condition(s) does this approach relax? Why?

(B) Another strategy is for each inner transaction to produce a list
of deferred updates/actions that the the outer transaction commits for it when the outer transaction commits. For any data item written in any transaction, all transactions read the last update value from this list. Which ACID condition(s) does this approach relax?

(C) If the only data flow is that the inner transaction reads from the outer transaction (meaning txid2 reads txid1's writes but txid1 never reads txid2's writes), do we still need to relax ACID? Why?

Rust Motivation

Locks’ litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Toleranc
- Performance
- Poor composability.

Solu on't use lo
- on-b ng
- ata stru entric

Shared mutable state requires locks

- So…separate sharing and mutability
- Use type system to make concurrency safe
  - Ownership
  - Immutability
  - Careful library support for sync primitives

Rust Goals

- Multi-paradigm language modeled after C and C++
  - Functional, Imperative, Object-Oriented
- Primary Goals:
  - Safe Memory Management
  - Safe Concurrency and Concurrent Controls

Be Fast: systems programming
Be Safe: don’t crash

Memory Management

- Rust: a “safe” environment for memory
  - No Null, Dangling, or Wild Pointers
- Objects are *immutable* by default
  - User has more explicit control over mutability
- Declared variables must be initialized prior to execution
  - A bit of a pain for static/global state

Unsafe

Functions determined unsafe via specific behavior

- Deference null or raw pointers
- Data Races
- Type Inheritance

Using “unsafe” keyword → bypass compiler enforcement

- Don’t do it. Not for the lab, anyway

The user deals with the integrity of the code

Credit: http://www.skiingforever.com/ski-tricks/

Other Relevant Features

- First-Class Functions and Closures
    - Similar to Lua, Go, …
- Algebraic data types (enums)
- Class Traits
    - Similar to Java interfaces
    - Allows classes to share aspects

Hard to use/learn without awareness of these issues

Concurrency

- Tasks → Rust’s threads
- Each task → stack and a heap
  - Stack Memory Allocation – A Slot
  - Heap Memory Allocation – A Box
- Tasks can share stack (portions) with other tasks
  - These objects must be immutable
- Task States: Running, Blocked, Failing, Dead
  - Failing task: interrupted by another process
  - Dead task: only viewable by other tasks
- Scheduling
  - Each task → finite time-slice
  - If task doesn’t finish, deferred until later
  - “M:N scheduler”

Hello World

```
fn main() {
    println!("Hello, world!")
}
```

Ownership

Ownership

n. The act, state, or right of possessing something

Borrow

v. To receive something with the promise of returning it

Ownership/Borrowing →

No need for a runtime

Memory safety (GC)

Data-race freedom

MM Options:

- Managed languages: GC
- Native languages: manual management
- Rust: 3rd option: ***track ownership***

- Each value in Rust has a variable called its *owner*.
- There can only be one owner at a time.
- Owner goes out of scope→value will be dropped.

Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
}
```

```
fn helper(name: String) {
  println!(“{}”, name);
}
```

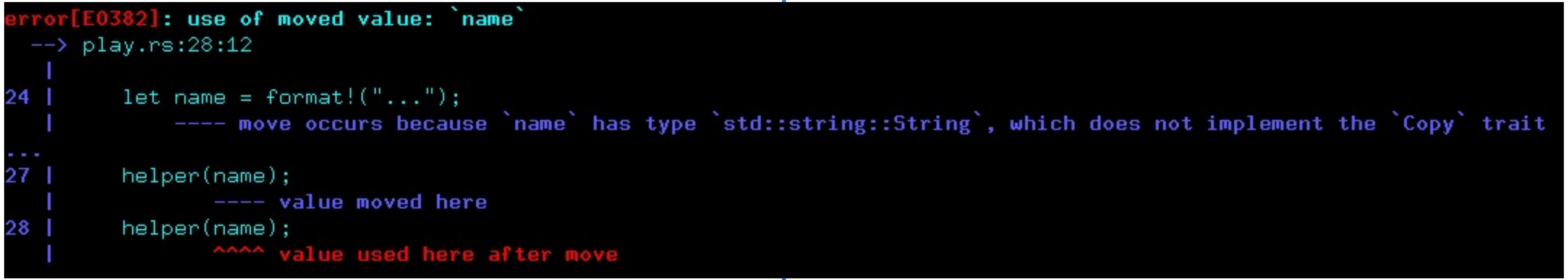
Ownership/Borrowing

```
fn main() {
  let name = format!("...");
  helper(name);
  helper(name);
}
```

Error: use of moved value: `name`

```
fn helper(name: String) {
  println!("{}", name);
}
```

Take ownership of a String



```
error[E0382]: use of moved value: `name`
  --> play.rs:28:12
   |
24 |     let name = format!("...");
   |         ---- move occurs because `name` has type `std::string::String`, which does not implement the `Copy` trait
...
27 |     helper(name);
   |            ---- value moved here
28 |     helper(name);
   |            ^^^^ value used here after move
```

What kinds of problems might this prevent?

Pass by reference takes “ownership implicitly” in other languages like Java

Shared Borrowing

```
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

Lend the string

```
fn helper(name: &String) {
  println!(“{}”, name);
}
```

Take a reference to a String

Why does this fix the problem?

Shared Borrowing with Concurrency

```rust
fn main() {
  let name = format!("...");
  helper(&name);
  helper(&name);
}
```

```rust
fn helper(name: &String) {
  thread::spawn(||{
    println!("{}", name);
  });
}
```

Lifetime `static` required

```
error[E0621]: explicit lifetime required in the type of `name`
  --> play.rs:11:18
   |
10 | fn helper(name: &String) -> thread::JoinHandle<()> {
   |                 ------- help: add explicit lifetime `'static` to the type of `name`: `&'static std::string::String`
11 |     let handle = thread::spawn(move ||{
   |                  ^^^^^^^^^^^^^^ lifetime `'static` required
```

Does this prevent the exact same class of problems?

Clone, Move

```
fn main() {
  let name = format!("...");
  helper(name.clone());
  helper(name);
}
```

Ensure concurrent owners
Work with different copies

Is this better?

```
fn helper(name: String) {
  thread::spawn(move || {
    println!("{}", name);
  });
}
```

Copy versus Clone:

Default: Types cannot be copied

- Values move from place to place
- E.g. file descriptor

Clone: Type is expensive to copy

- Make it explicit with clone call
- e.g. Hashtable

Copy: type implicitly copy-able

- e.g. u32, i32, f32, …

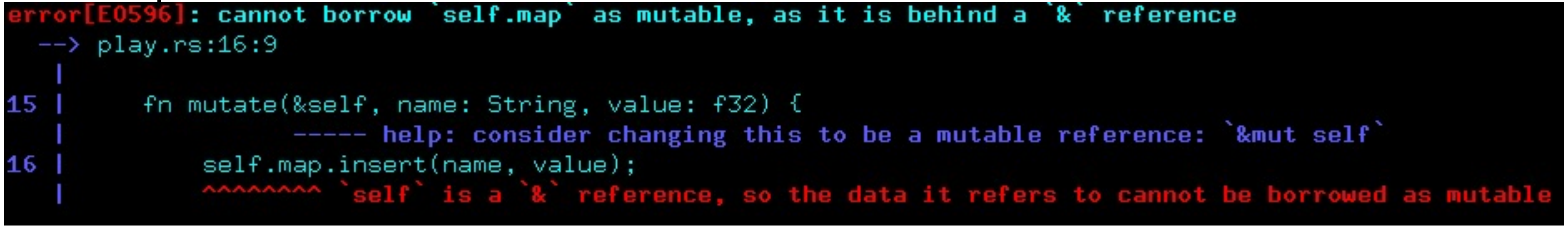
`#[derive(Clone, Debug)]`

Mutability

```
struct Structure {
    id: i32,
    map: HashMap<String, f32>,
}

impl Structure {
    fn mutate(&self, name: String, value: f32) {
        self.map.insert(name, value);
    }
}
```

Error: cannot be borrowed as mutable



```
error[E0596]: cannot borrow `self.map` as mutable, as it is behind a `&` reference
  --> play.rs:16:9
   |
15 |     fn mutate(&self, name: String, value: f32) {
   |               ----- help: consider changing this to be a mutable reference: `&mut self`
16 |         self.map.insert(name, value);
   |         ^^^^^^^^ `self` is a `&` reference, so the data it refers to cannot be borrowed as mutable
```

Mutability

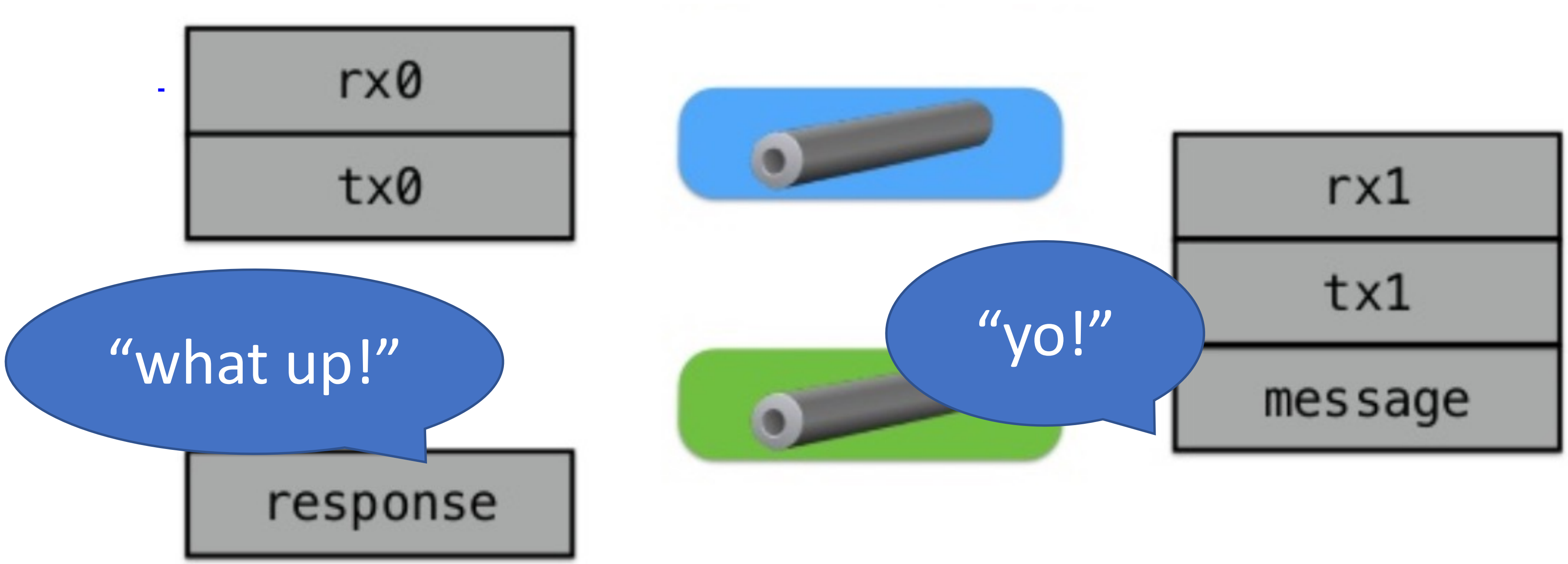
```
struct Structure {
    id: i32,
    map: HashMap<String, f32>,
}

impl Structure {
    fn mutate(&mut self, name: String, value: f32){
        self.map.insert(name, value);
    }
}
```

Key idea:

- Force mutation and ownership to be explicit
- Fixes MM *and* concurrency in fell swoop!

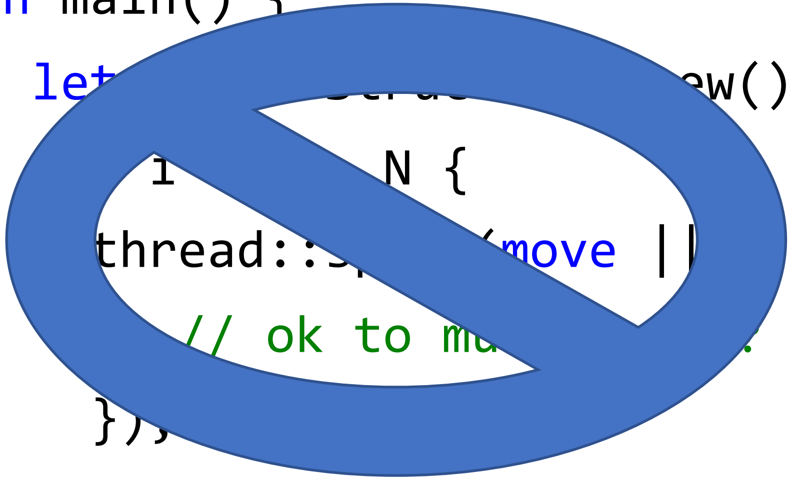
Sharing State: Channels



```
  let (message, tx1) = rx0.recv().unwrap();
  tx1.send(format!("what up!")).unwrap();
  println("parent received {}", message);
}
```

APIs return Option<T>

Sharing State



```
fn main() {
  let                          ew();
      i        N {
    thread::S         move ||
        // ok to mu
    });
  }
}
```

Sharing State: Arc and Mutex

```
fn main() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

Key ideas:

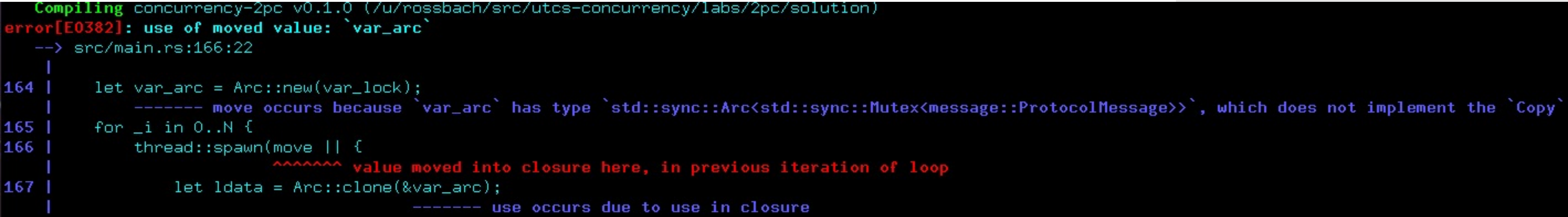
- Use reference counting wrapper to pass refs
- Use scoped lock for mutual exclusion
- Actually compiles → works 1st time!

Sharing State: Arc and Mutex, *really*

```rust
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

Why doesn't "&" fix it?
(&var_arc, instead of just var_arc)

Would cloning var_arc fix it?



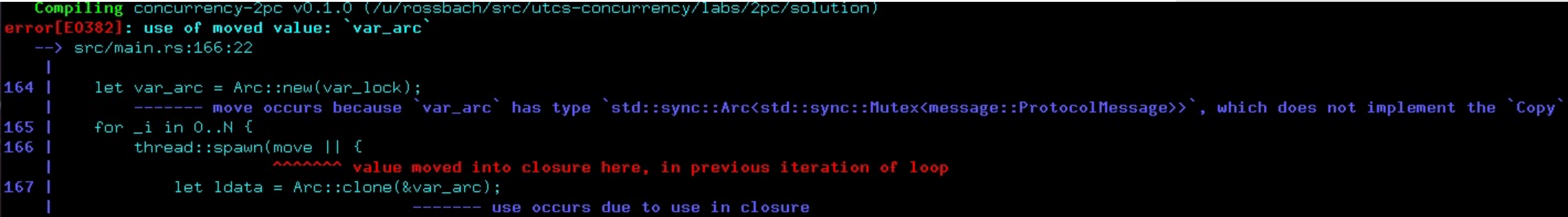
```
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0382]: use of moved value: `var_arc`
   --> src/main.rs:166:22
    |
164 |     let var_arc = Arc::new(var_lock);
    |         ------- move occurs because `var_arc` has type `std::sync::Arc<std::sync::Mutex<message::ProtocolMessage>>`, which does not implement the `Copy`
165 |     for _i in 0..N {
166 |         thread::spawn(move || {
    |                       ^^^^^^^ value moved into closure here, in previous iteration of loop
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- use occurs due to use in closure
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(move || {
      let ldata = Arc::clone(&var_arc.clone());
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

Same problem!

What if we just don't ***move?***

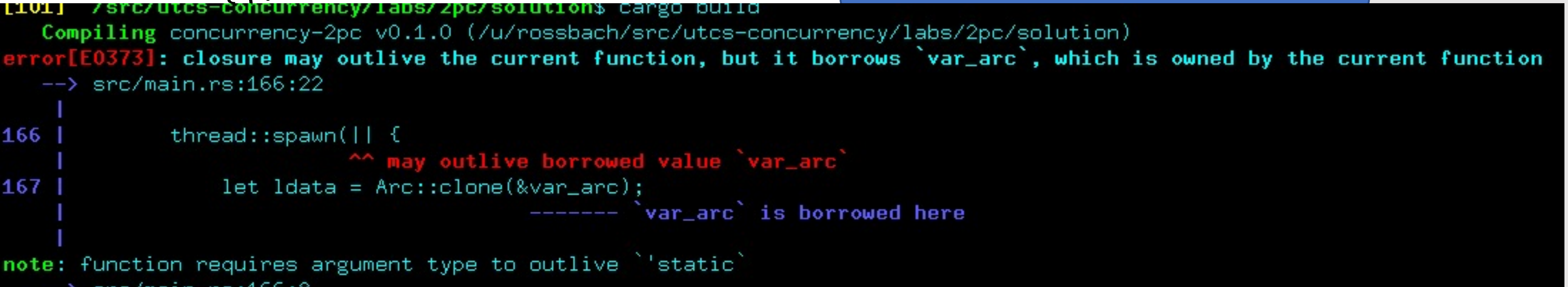


```
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0382]: use of moved value: `var_arc`
   --> src/main.rs:166:22
    |
164 |     let var_arc = Arc::new(var_lock);
    |         ------- move occurs because `var_arc` has type `std::sync::Arc<std::sync::Mutex<message::ProtocolMessage>>`, which does not implement the `Copy`
165 |     for _i in 0..N {
166 |         thread::spawn(move || {
    |                       ^^^^^^^ value moved into closure here, in previous iteration of loop
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- use occurs due to use in closure
    |
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    thread::spawn(|| {
      let ldata = Arc::clone(&var_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
```

What's the actual fix?



```
[101] /src/utcs-concurrency/labs/2pc/solution$ cargo build
   Compiling concurrency-2pc v0.1.0 (/u/rossbach/src/utcs-concurrency/labs/2pc/solution)
error[E0373]: closure may outlive the current function, but it borrows `var_arc`, which is owned by the current function
   --> src/main.rs:166:22
    |
166 |         thread::spawn(|| {
    |                       ^^ may outlive borrowed value `var_arc`
167 |             let ldata = Arc::clone(&var_arc);
    |                                     ------- `var_arc` is borrowed here
    |
note: function requires argument type to outlive `'static`
```

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lock);
  for i in 0..N {
    let clone_arc = var_arc.clone();
    thread::spawn(move || {
      let ldata = Arc::clone(&clone_arc);
      let vdata = ldata.lock();
      // ok to mutate var (vdata)!
    });
  }
}
```

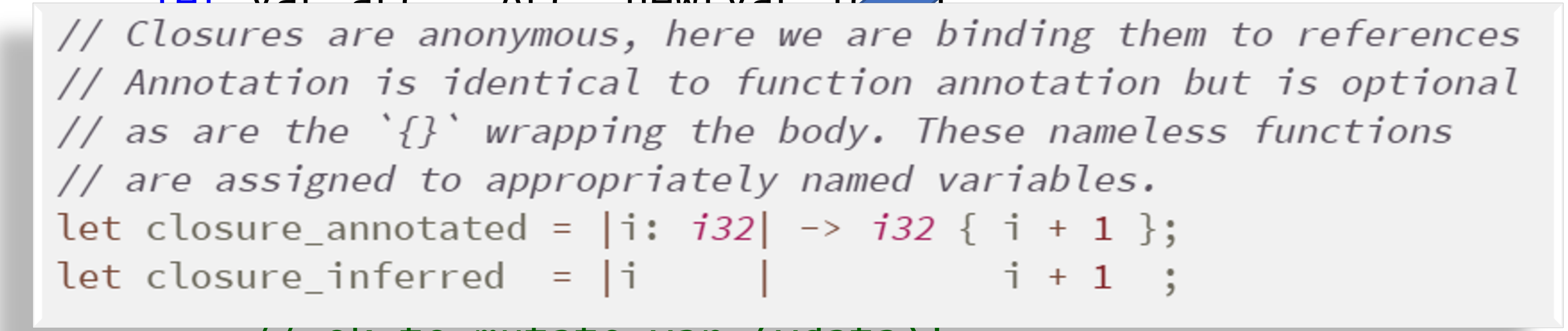
Compiles! Yay!

Other fixes?

Sharing State: Arc and Mutex, *really*

```
fn test() {
  let var = Structure::new();
  let var_lock = Mutex::new(var);
  let var_arc = Arc::new(var_lo   );
```

Parameters!



```
// Closures are anonymous, here we are binding them to references
// Annotation is identical to function annotation but is optional
// as are the `{}` wrapping the body. These nameless functions
// are assigned to appropriately named variables.
let closure_annotated = |i: i32| -> i32 { i + 1 };
let closure_inferred  = |i     |          i + 1  ;
```

```
      // ok to mutate var (vdata)!
    ~~});~~
  }
  for i in 0..N { join(); }
}
```

Why does this compile?

Could we use a vec of JoinHandle to keep var_arc in scope?

What if I need my lambda to own some things and borrow others?

Discussion

GC lambdas, Rust C++

- This is pretty nuanced:
- Stack closures, owned closures, managed closures, exchg heaps

Ownership and Macros

  Macros use regexp and expand to closures

Summary

Rust: best of both worlds

  systems vs productivity language

Separate sharing, mutability, concurrency

Type safety solves MM and concurrency

Have fun with the lab!