Rust

cs378h

Chris Rossbach

Outline

Administrivia

    Midterm 1 results

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!

Memory Lane: Zoominutia

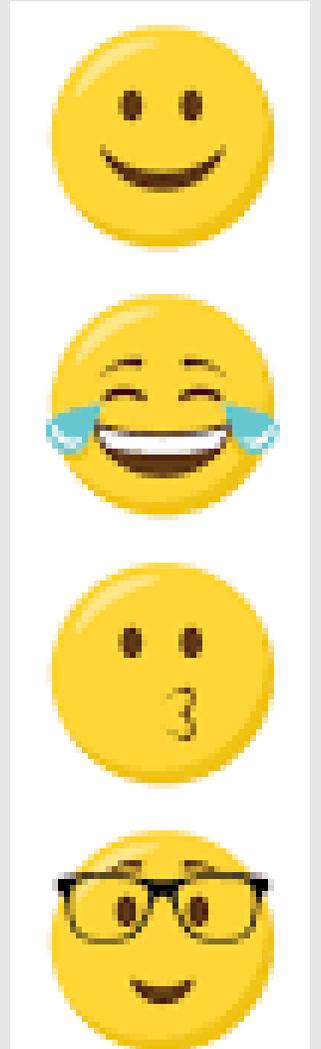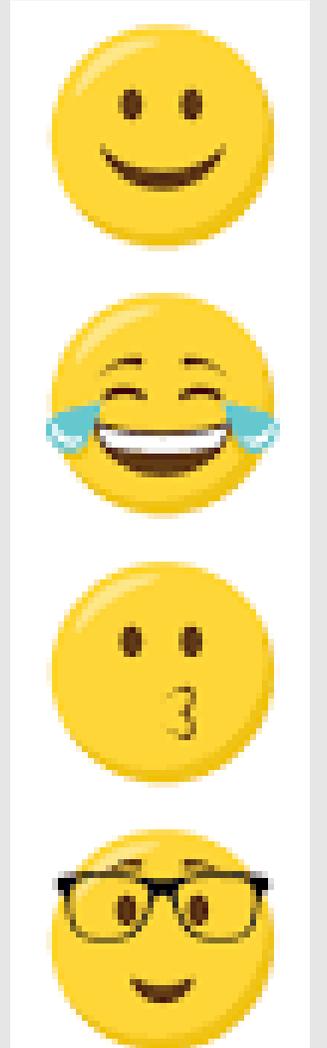# Memory Lane: Zoominutia

- It may take some practice

Memory Lane: Zoominutia

- It may take some practice
- I reserved the right side of the slides for people

Memory Lane: Zoominutia

- It may take some practice
- I reserved the right side of the slides for people

Memory Lane: Zoominutia

- It may take some practice
- I reserved the right side of the slides for people
- You can raise your hand
  - On participants bar/window/widget/thingy

Memory Lane: Zoominutia

- It may take some practice
- I reserved the right side of the slides for people
- You can raise your hand
  - On participants bar/window/widget/thingy

Memory Lane: Zoominutia

- It may take some practice

- I reserved the right side of the slides for people

- You can raise your hand
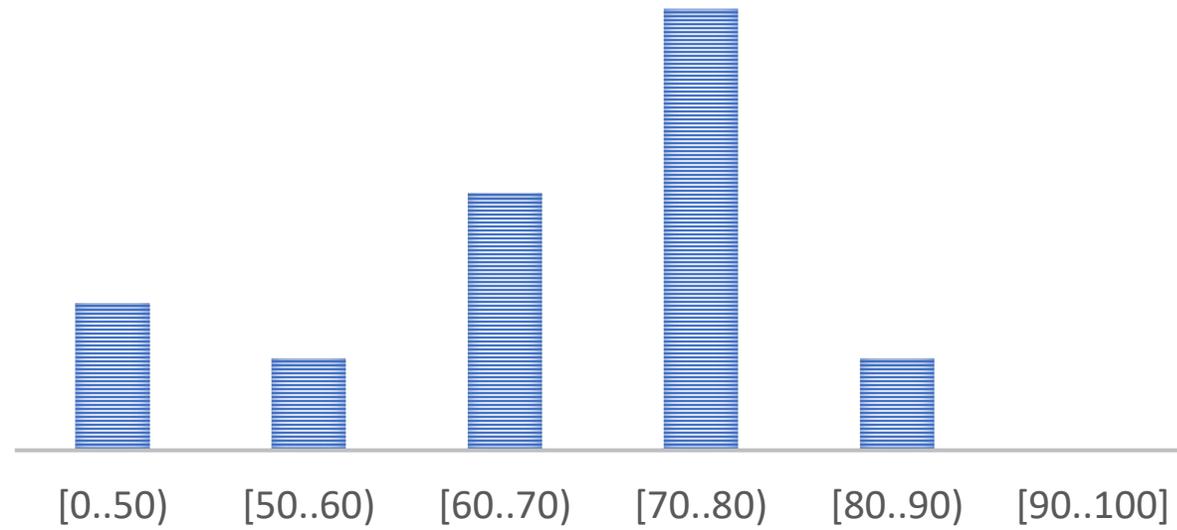 - On participants bar/window/widget/thingy
  - But I might not hear it
  - OK to just speak up (I hope)
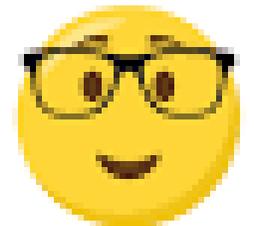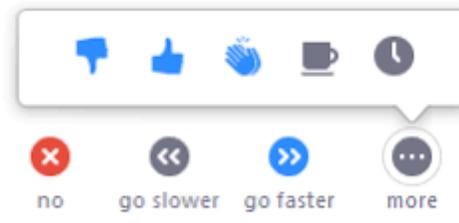
Midterm Statistics

Midterm Statistics
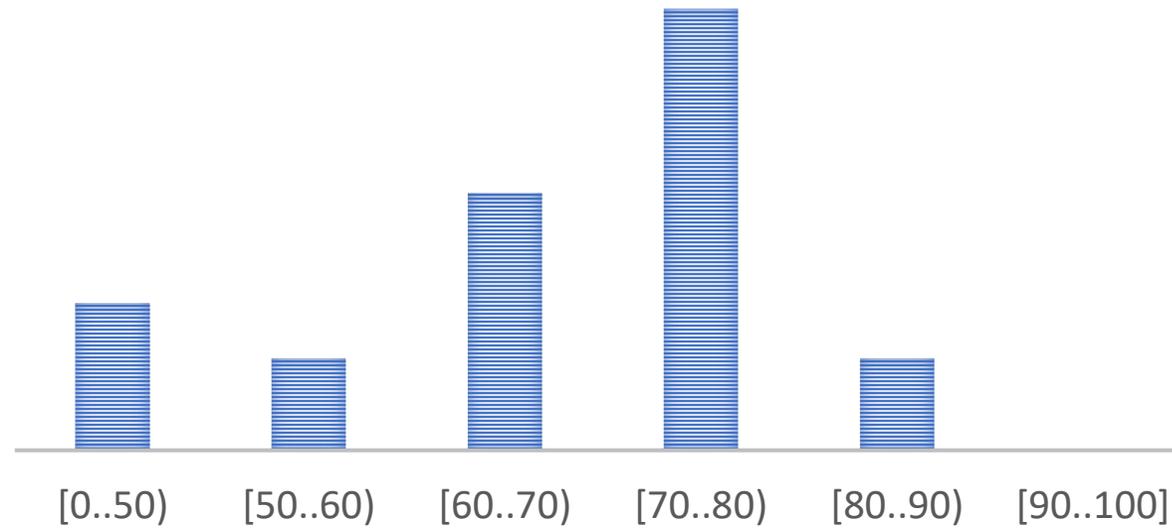
- Mean: 65

Midterm Statistics
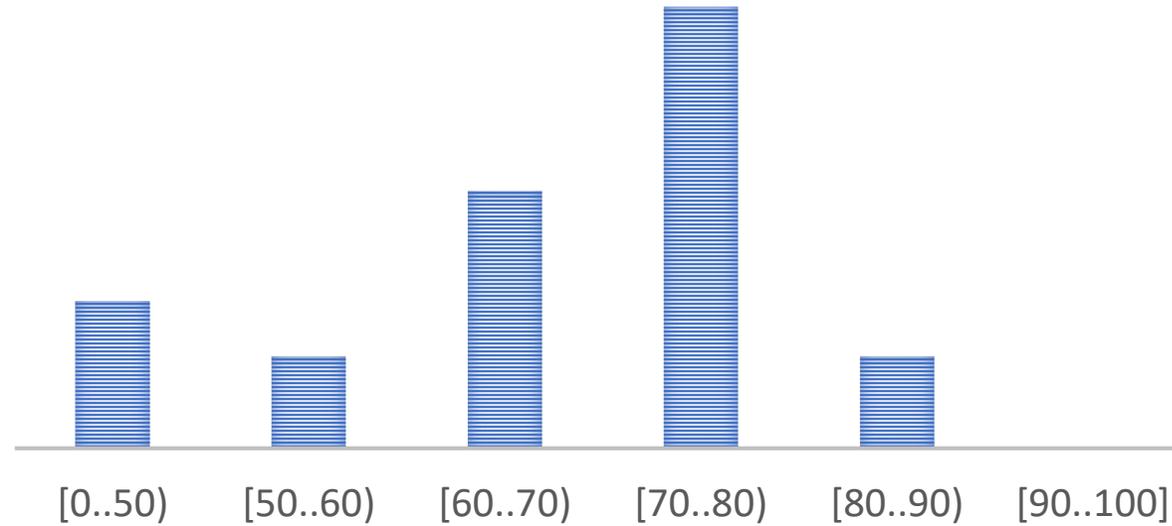
- Mean: 65
- Median: 71

Midterm Statistics
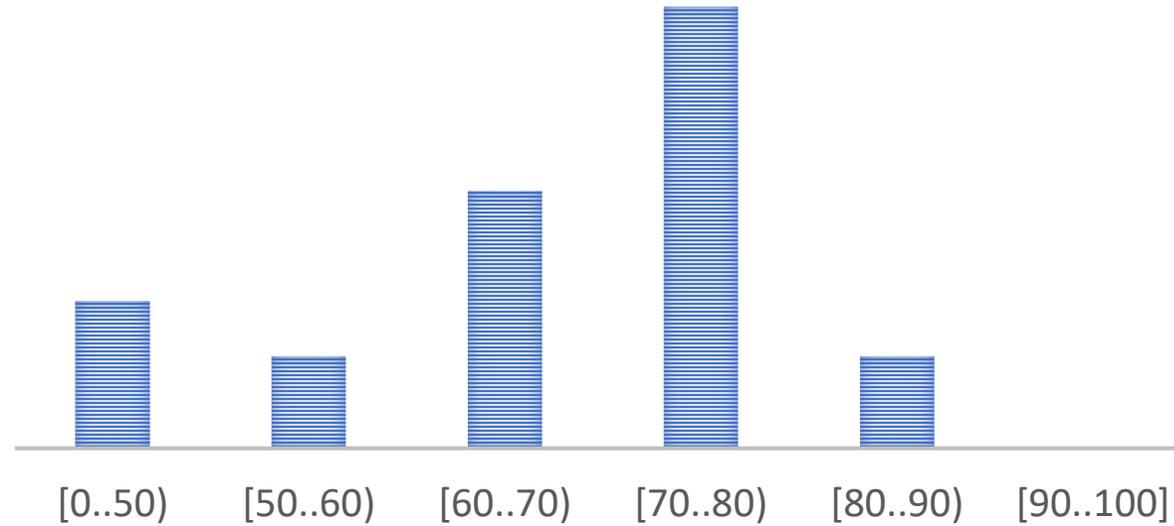
- Mean: 65
- Median: 71
- STDEV: 18

Midterm Statistics
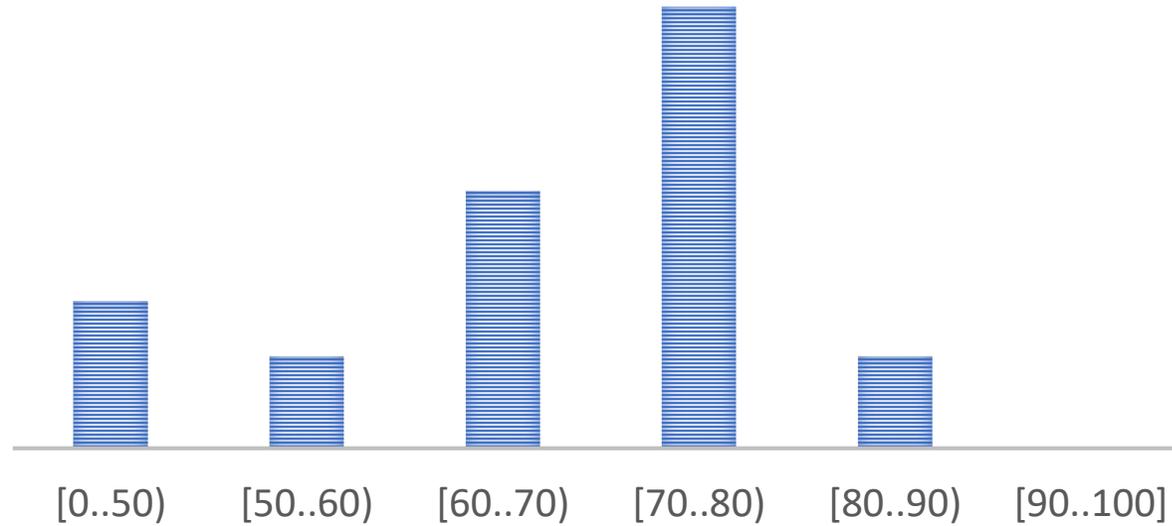
- Mean: 65
- Median: 71
- STDEV: 18
- Range: [28..82]

| [0..50) | [50..60) | [60..70) | [70..80) | [80..90) | [90..100] |
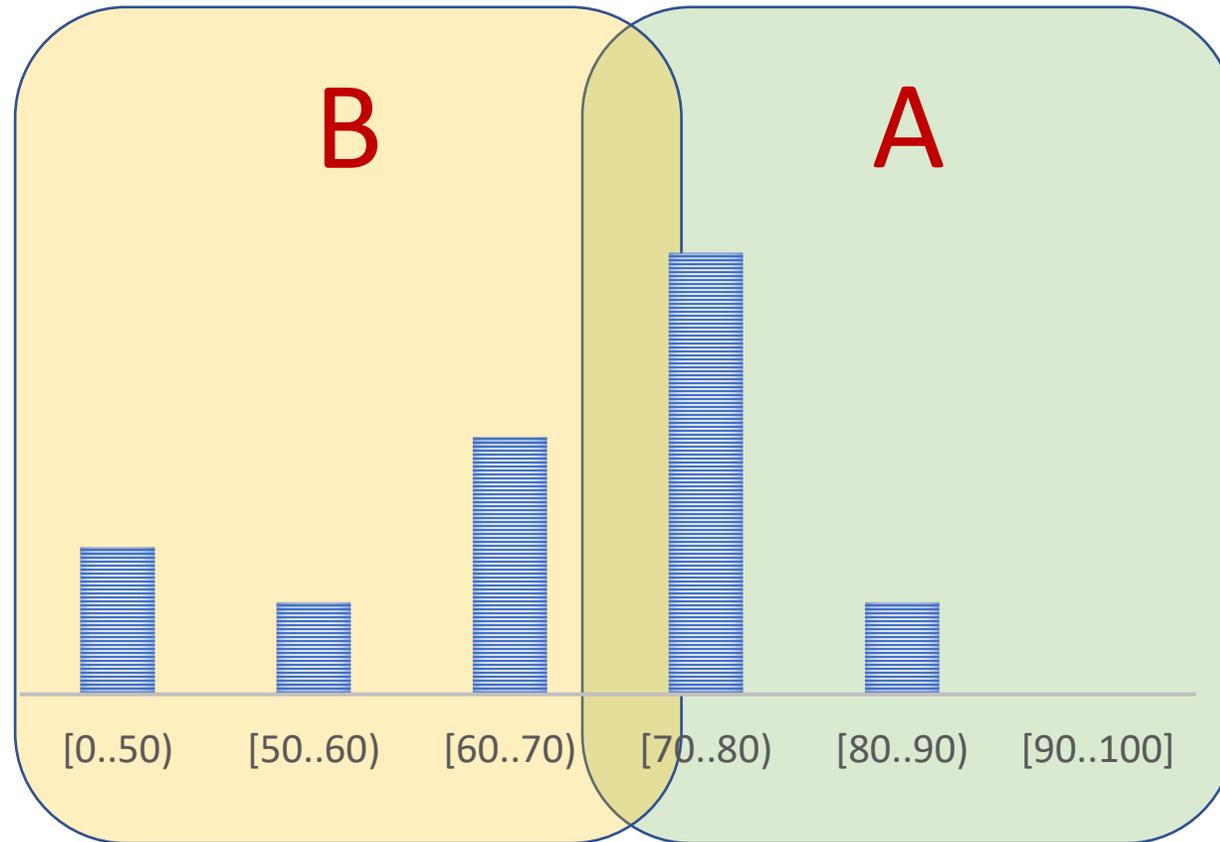
Midterm Statistics
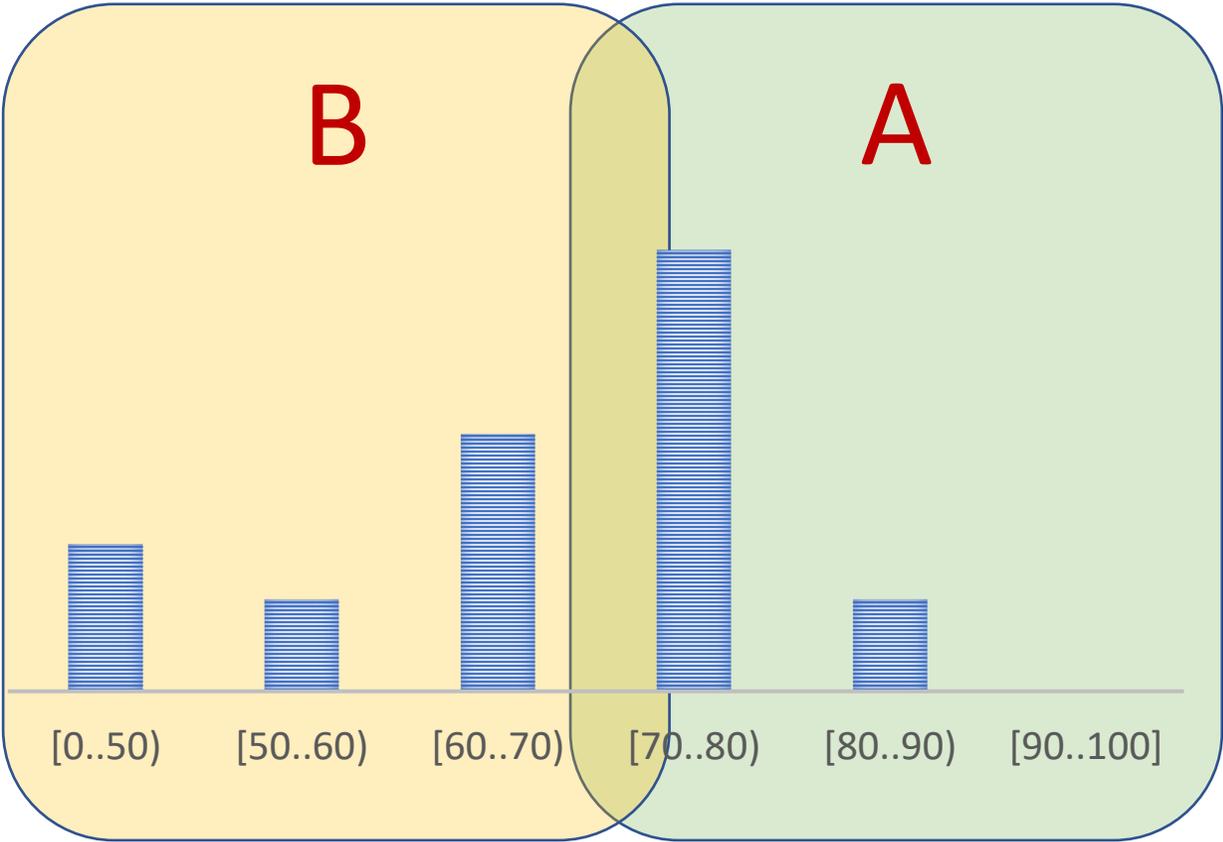
- Mean: 65
- Median: 71
- STDEV: 18
- Range: [28..82]

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*: 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*: 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*: 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*: 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?

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

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!

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*: 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?

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

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:

```go
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()

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!

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

```
 1  PROGRAM MyProgram {
 2      TASK ReadFileAsync(name, callback) {
 3          ReadFileSync(name);
 4          Call(callback);
 5      }
 6      CALLBACK FinishOpeningFile() {
 7          LoadFile(file);
 8          RedrawScreen();
 9      }
10      OnOpenFile() {
11          FILE file;
12          char szName[BUFSIZE]
13          InitFileName(szName);
14          EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15      }
16      OnPaint();
17  }
```

Stack-Ripping

```
 1  PROGRAM MyProgram {
 2      TASK ReadFileAsync(name, callback) {
 3          ReadFileSync(name);
 4          Call(callback);
 5      }
 6      CALLBACK FinishOpeningFile() {
 7          LoadFile(file);
 8          RedrawScreen();
 9      }
10      OnOpenFile() {
11          FILE file;
12          char szName[BUFSIZE]
13          InitFileName(szName);
14          EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15      }
16      OnPaint();
17  }
```

Stack-Ripping

```
1   PROGRAM MyProgram {
2       TASK ReadFileAsync(name, callback) {
3           ReadFileSync(name);
4           Call(callback);
5       }
6       CALLBACK FinishOpeningFile() {
7           LoadFile(file);
8           RedrawScreen();
9       }
10      OnOpenFile() {
11          FILE file;
12          char szName[BUFSIZE]
13          InitFileName(szName);
14          EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15      }
16      OnPaint();
17  }
```

Stack-Ripping

```
1   PROGRAM MyProgram {
2       TASK ReadFileAsync(name, callback) {
3           ReadFileSync(name);
4           Call(callback);
5       }
6       CALLBACK FinishOpeningFile() {
7           LoadFile(file);
8           RedrawScreen();
9       }
10      OnOpenFile() {
11          FILE file;
12          char szName[BUFSIZE]
13          InitFileName(szName);
14          EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
15      }
16      OnPaint();
17  }
```

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

Exam Q*: Transactions

Exam Q*: 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);
```

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., l1). 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, mmitted 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 eceding writes in that inner transaction, and all writes in preceding, committed inner transactions. *Implementing* these semantics can be tricky.

) 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 e outer transaction aborts. Which ACID condition(s) does this approach relax? Why?

) Another strategy is for each inner transaction to produce a list 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 ansactions read the last update value from this list. Which ACID condition(s) does this approach relax?

) 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 ll need to relax ACID? Why?

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

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);
```

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., l1). 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, mmitted 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 eceding writes in that inner transaction, and all writes in preceding, committed inner transactions. *Implementing* these semantics can be tricky.

) 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 e outer transaction aborts. Which ACID condition(s) does this approach relax? Why?

) Another strategy is for each inner transaction to produce a list 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 ansactions read the last update value from this list. Which ACID condition(s) does this approach relax?

) 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 ll need to relax ACID? Why?

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

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);
```

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., 11). 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, mmitted 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 eceding writes in that inner transaction, and all writes in preceding, committed inner transactions. *Implementing* these semantics can be tricky.

) 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 e outer transaction aborts. Which ACID condition(s) does this approach relax? Why?

) Another strategy is for each inner transaction to produce a list 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 ansactions read the last update value from this list. Which ACID condition(s) does this approach relax?

) 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 ll need to relax ACID? Why?

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);
```

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., l1). 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, mmitted 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 eceding writes in that inner transaction, and all writes in preceding, committed inner transactions. *Implementing* these semantics can be tricky.

) 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 e outer transaction aborts. Which ACID condition(s) does this approach relax? Why?

) Another strategy is for each inner transaction to produce a list 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 ansactions read the last update value from this list. Which ACID condition(s) does this approach relax?

) 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 ll need to relax ACID? Why?

Rust Motivation

Rust Motivation

Locks' litany of problems:

Rust Motivation

Locks' litany of problems:

- Deadlock

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance
- Poor composability…

Rust Motivation

Locks' litany of problems:

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

Solution: don't use locks
- non-blocking
- Data-structure-centric
- HTM
- blah, blah, blah..

Rust Motivation

Locks' litany of problems:

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance
- Poor composability…

Soluon't use l
- on-bng
- ata-structentric
- M
- blal

Rust Motivation

Locks' litany of problems:

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

Solution: don't use locks
- Non-blocking
- Data structure-centric

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

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

Memory Management

Rust: a "safe" environment for memory

No Null, Dangling, or Wild Pointers

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

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

Unsafe

Functions determined unsafe via specific behavior

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

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

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

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

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

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

Concurrency

Tasks → Rust's threads

Concurrency

Tasks → Rust's threads

Each task → stack and a heap

      Stack Memory Allocation – A Slot
      Heap Memory Allocation – A Box

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

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

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

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

Ownership

Ownership

Ownership

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

Ownership

Ownership

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

Borrow

v. To receive something with the promise of returning it

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

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***

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

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

Ownership/Borrowing

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

Ownership/Borrowing

```rust
fn main() {
    let name = format!("...");
    helper(name);
}
```
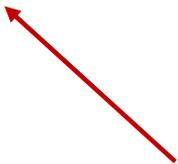
```rust
fn helper(name: String) {
    println!("{}", name);
}
```

Ownership/Borrowing

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

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

Ownership/Borrowing

```rust
fn main() {
    let name = format!("...");
    helper(name);
    helper(name);
}
```
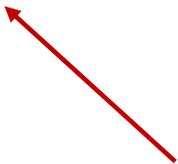
Error: use of moved value: `name`

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

Ownership/Borrowing

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

Error: use of moved value: `name`
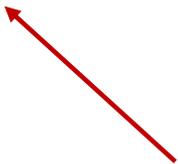
```rust
fn helper(name: String) {
    println!("{}", name);
}
```

Take ownership of a String

Ownership/Borrowing

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

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

Take ownership of a String

Error: use of moved value: `name`

```
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
```

Ownership/Borrowing

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

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

Take ownership of a String

Error: use of moved value: `name`

```
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?

Ownership/Borrowing

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

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

Take ownership of a String

Error: use of moved value: `name`

```
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

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

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

Shared Borrowing

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

Lend the string

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

Shared Borrowing

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

Lend the string

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

Take a reference to a String

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);
    });
}
```

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

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
```

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

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

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

Clone, Move

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

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

Explicitly take ownership

Clone, Move

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

Ensure concurrent owners
Work with different copies

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

Explicitly take ownership

Clone, Move

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

Ensure concurrent owners
Work with different copies

Is this better?

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

Explicitly take ownership

Clone, Move

```rust
fn main() {

    let name = format!("...");

    helper(name.clone());

    helper(name);

}
```

Ensure concurrent owners
Work with different copies

Is this better?

```rust
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

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

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

Mutability

```rust
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

Mutability

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


impl Structure {
    fn mutate(&self, name: String, value: f32) {
        self.map.insert(name, value);
    }
}
```
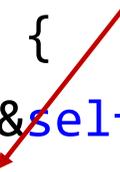
Error: cannot be borrowed as mutable

```
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

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

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

Mutability

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

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

Mutability

```rust
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

Sharing State: Channels

```rust
fn main() {
```

Sharing State: Channels

```rust
fn main() {
    let (tx0, rx0) = channel();
```
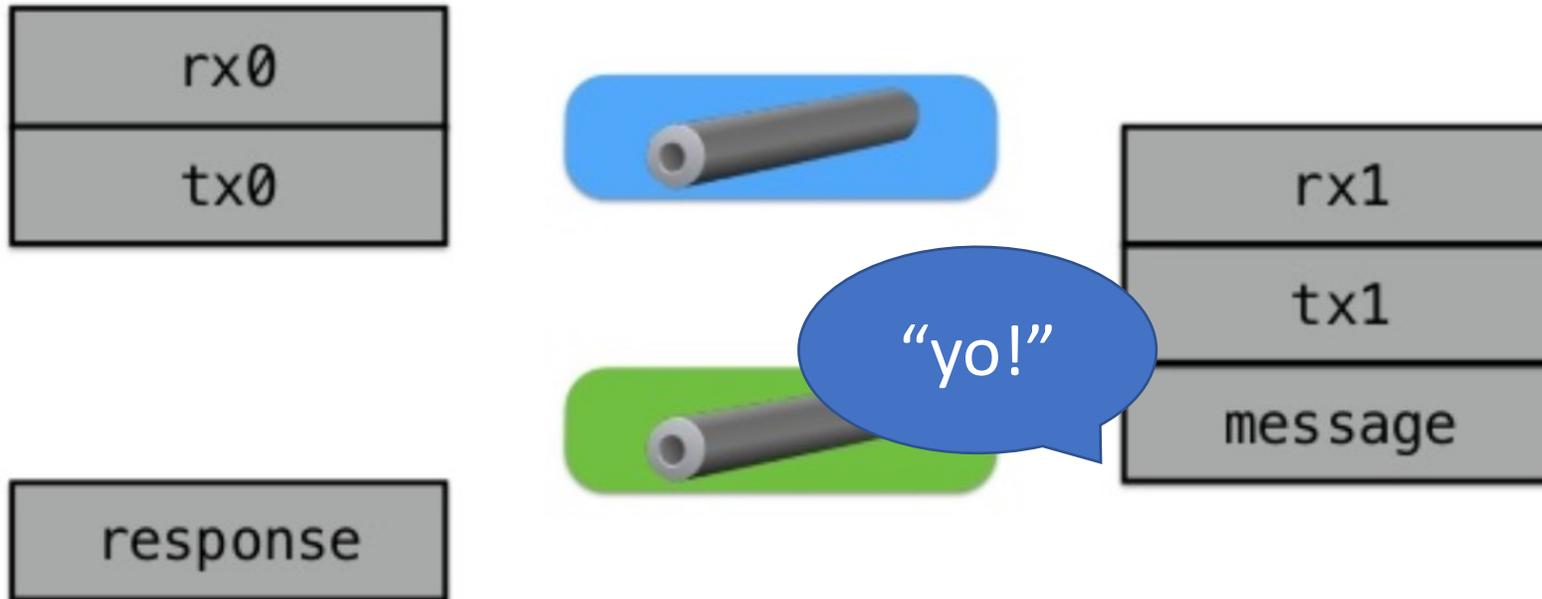
Sharing State: Channels

```rust
fn main() {
  let (tx0, rx0) = channel();
  thread::spawn(move || {
    let (tx1, rx1) = channel();
    tx0.send((format!("yo"), tx1)).unwrap();
    let response = rx1.recv().unwrap();
    println!("child got {}", response);
  });
```

Sharing State: Channels

```rust
fn main() {
    let (tx0, rx0) = channel();
    thread::spawn(move || {
        let (tx1, rx1) = channel();
        tx0.send((format!("yo"), tx1)).unwrap();
        let response = rx1.recv().unwrap();
        println!("child got {}", response);
    });
    let (message, tx1) = rx0.recv().unwrap();
    tx1.send(format!("what up!")).unwrap();
    println("parent received {}", message);
}
```
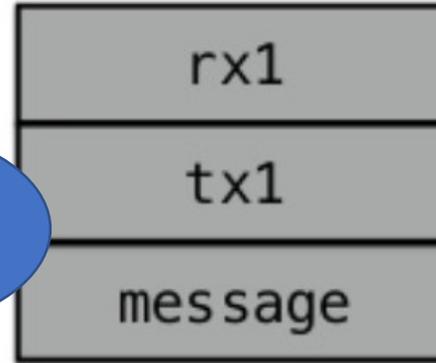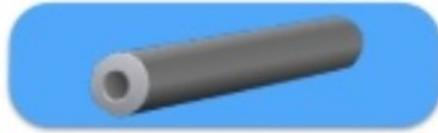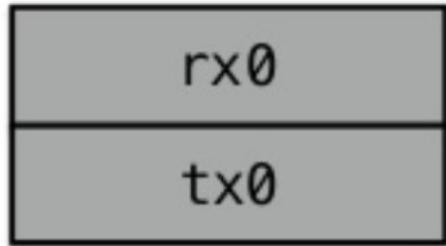
Sharing State: Channels


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

Sharing State: Channels


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

Sharing State: Channels


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

Sharing State: Channels

```rust
fn main() {
    let (tx0, rx0) = channel();
    thread::spawn(move || {
        let (tx1, rx1) = channel();
        tx0.send((format!("yo"), tx1)).unwrap();
        let response = rx1.recv().unwrap();
        println!("child got {}", response);
    });
    let (message, tx1) = rx0.recv().unwrap();
    tx1.send(format!("what up!")).unwrap();
    println("parent received {}", message);
}
```

Sharing State: Channels

```rust
fn main() {
    let (tx0, rx0) = channel();
    thread::spawn(move || {
        let (tx1, rx1) = channel();
        tx0.send((format!("yo"), tx1)).unwrap();
        let response = rx1.recv().unwrap();
        println!("child got {}", response);
    });
    let (message, tx1) = rx0.recv().unwrap();
    tx1.send(format!("what up!")).unwrap();
    println("parent received {}", message);
}
```
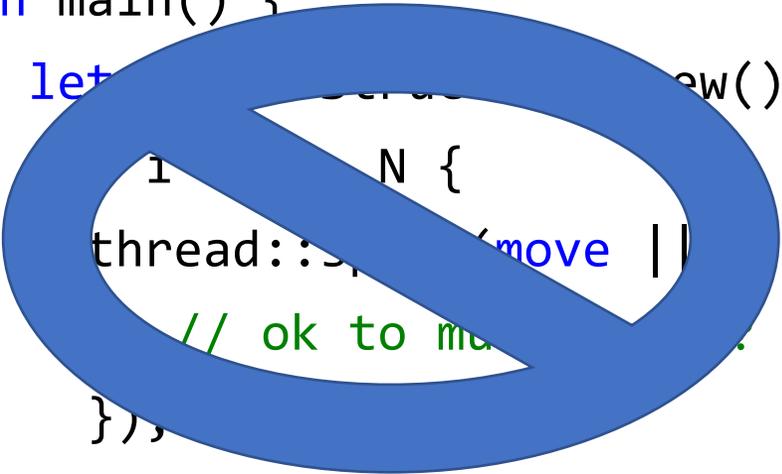
APIs return Option<T>

Sharing State

```rust
fn main() {
    let var = Structure::new();
    for i in 0..N {
        thread::spawn(move || {
            // ok to mutate var?
        });
    }
}
```

Sharing State

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

Sharing State: Arc and Mutex

```rust
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)!
        });
    }
}
```

Sharing State: Arc and Mutex

```rust
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)!
        });
    }
}
```

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)!
        });
    }
}
```

Sharing State: Arc and Mutex

```rust
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)!
        });
    }
}
```

Sharing State: Arc and Mutex

```rust
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)!
        });
    }
}
```

Sharing State: Arc and Mutex

```rust
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!

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!