Welcome to the new Reality
A/V Protocols
Rust

cs378h

Chris Rossbach

Outline

Administrivia
- Thoughts/Comments on Zoom
- Schedule Changes
- Policy Changes
- 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!

Zoominutia

Zoominutia

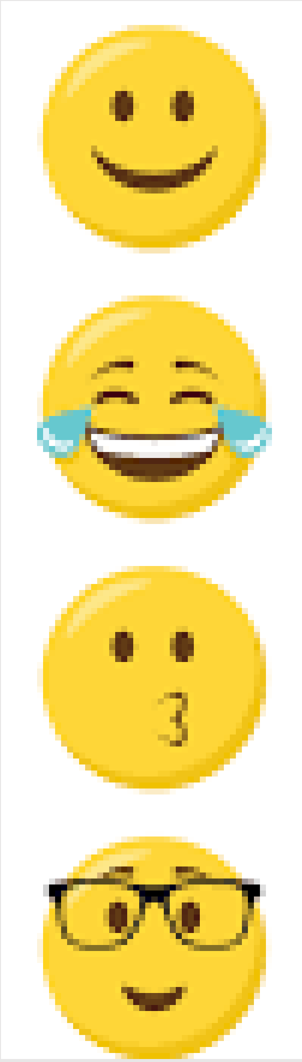
- It may take some practice

Zoominutia

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

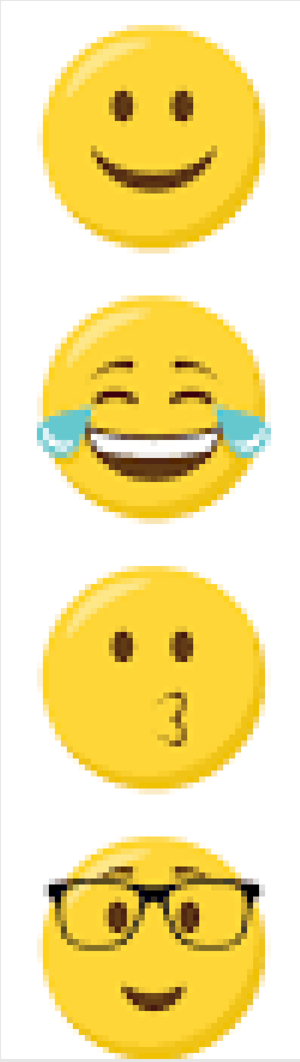
Zoominutia

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



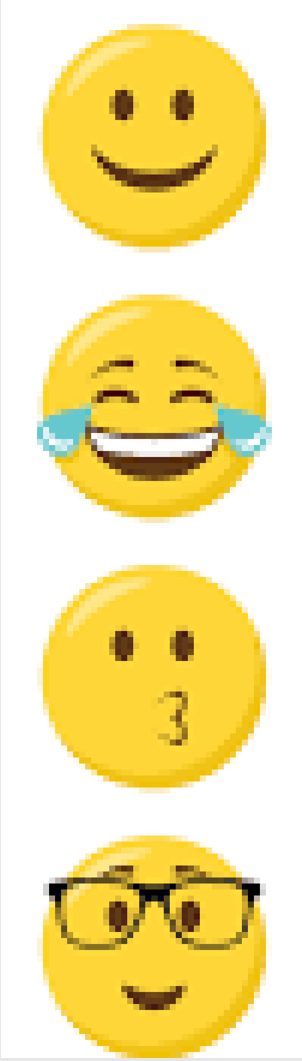
Zoominutia

- 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



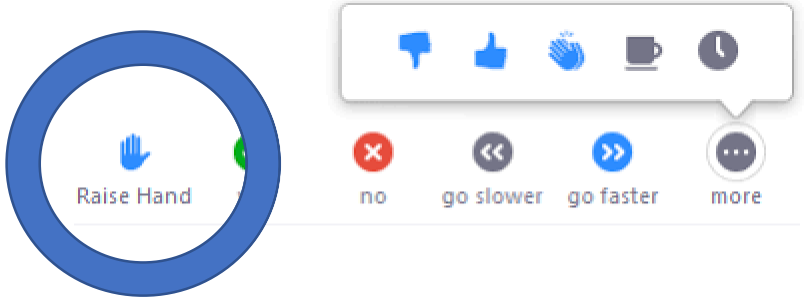
Zoominutia

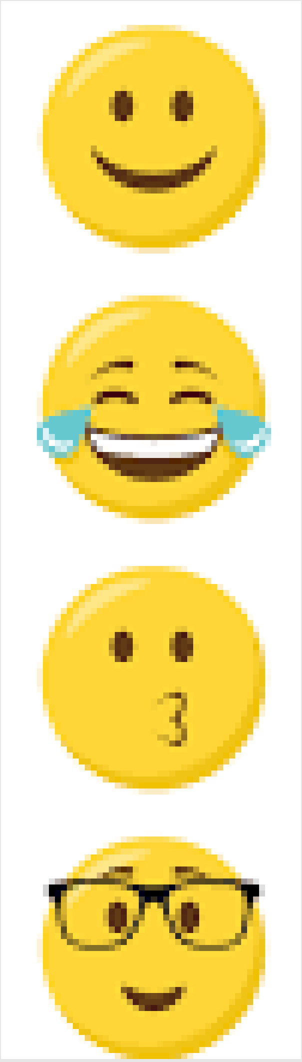
- It may take some practice
- I reserved the right side of the slides for people
- You can raise your hand
  - On participants bar/window/widget/thingy



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)





Schedule Changes

Schedule Changes

We lost a week and FPGAs can have germs. So:

Schedule Changes

We lost a week and FPGAs can have germs. So:

- FPGA lab, now optional, end of semester

Schedule Changes

We lost a week and FPGAs can have germs. So:

- FPGA lab, now optional, end of semester
- MPI lab, still optional, end of semester

Schedule Changes

We lost a week and FPGAs can have germs. So:

- FPGA lab, now optional, end of semester
- MPI lab, still optional, end of semester
- Rust Lab moved up to start…now

Schedule Changes

We lost a week and FPGAs can have germs. So:

- FPGA lab, now optional, end of semester
- MPI lab, still optional, end of semester
- Rust Lab moved up to start…now
- Exam 2: still last day of class
    - Do you want it take home (hard) or in-class (honor)?

Schedule Changes

We lost a week and FPGAs can have germs. So:

- FPGA lab, now optional, end of semester
- MPI lab, still optional, end of semester
- Rust Lab moved up to start…now
- Exam 2: still last day of class
  - Do you want it take home (hard) or in-class (honor)?
- Final Project
  - Start thinking about it soon!
  - Schedule remains the same, scope may need to shrink

Schedule Changes

We lost a week and FPGAs can have germs. So:

- FPGA lab, now optional, end of semester
- MPI lab, still optional, end of semester
- Rust Lab moved up to start…now
- Exam 2: still last day of class
  - Do you want it take home (hard) or in-class (honor)?
- Final Project
  - Start thinking about it soon!
  - Schedule remains the same, scope may need to shrink
- Lecture, office hours etc. all same times, on Zoom

Schedule Changes

We lost a week and FPGAs can have germs. So:

- FPGA lab, now optional, end of semester
- MPI lab, still optional, end of semester
- Rust Lab moved up to start…now
- Exam 2: still last day of class
    - Do you want it take home (hard) or in-class (honor)?
- Final Project
    - Start thinking about it soon!
    - Schedule remains the same, scope may need to shrink
- Lecture, office hours etc. all same times, on Zoom
- *The schedule page should look different now!*

Policy Changes

Policy Changes

- UT-mandated assessment changes
  - We will follow them scrupulously
  - We may not know them all yet
  - 378h policies will likely be *more* lenient

Policy Changes

- UT-mandated assessment changes
  - We will follow them scrupulously
  - We may not know them all yet
  - 378h policies will likely be *more* lenient
- Some thoughts on grades
  - 378h is already graded like grad school
  - Late policy will likely require relaxation
  - Probably worth remembering as you consider P/NC options

Hang in there

Hang in there

…

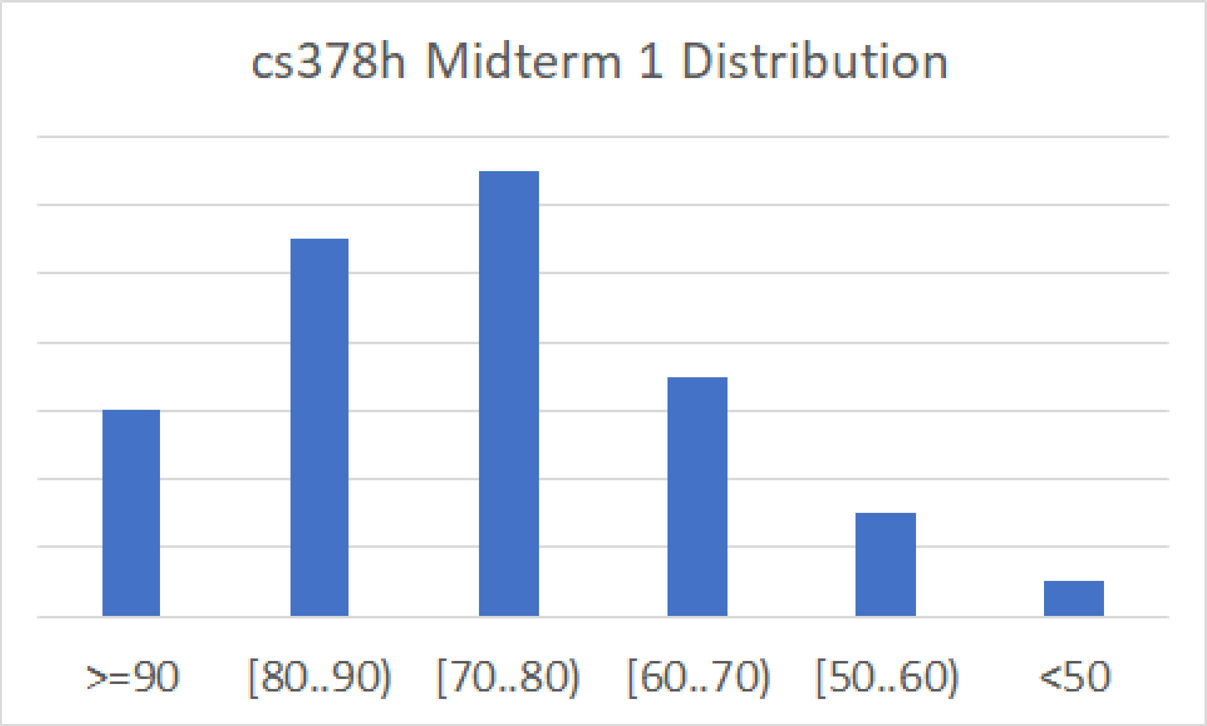
Hang in there

…

Are there Questions?

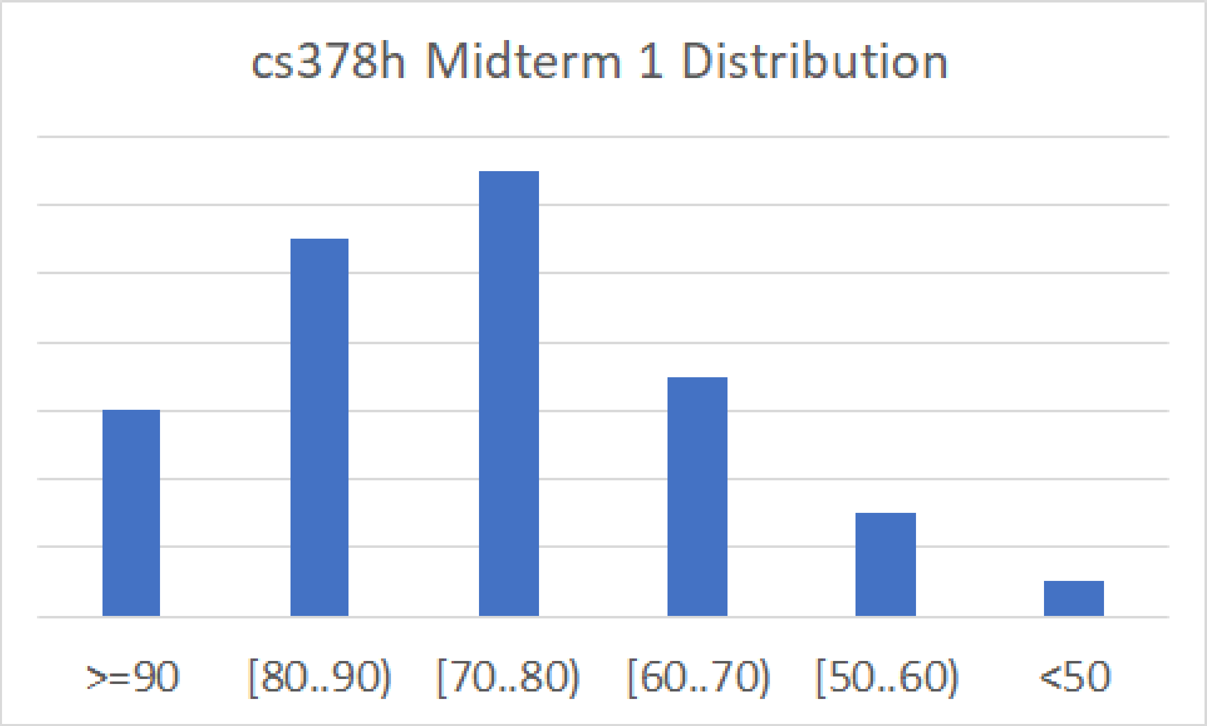
Midterm 1

Midterm 1



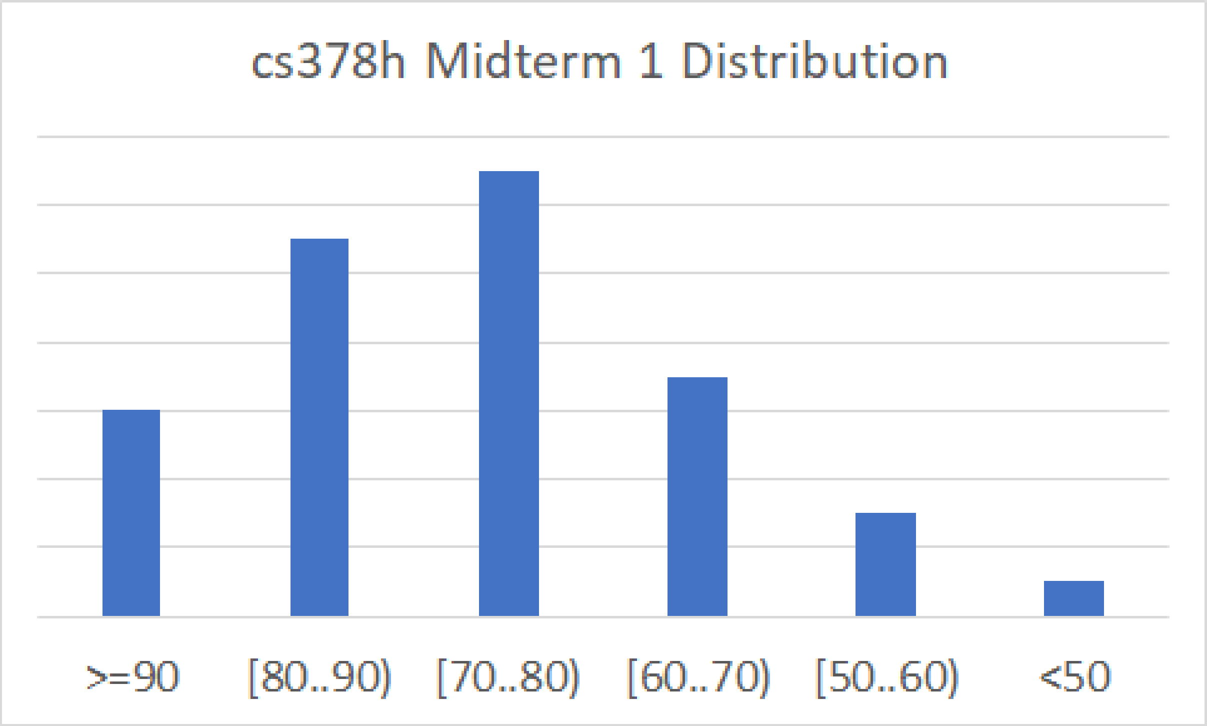
Midterm 1

- Mean: 76.3



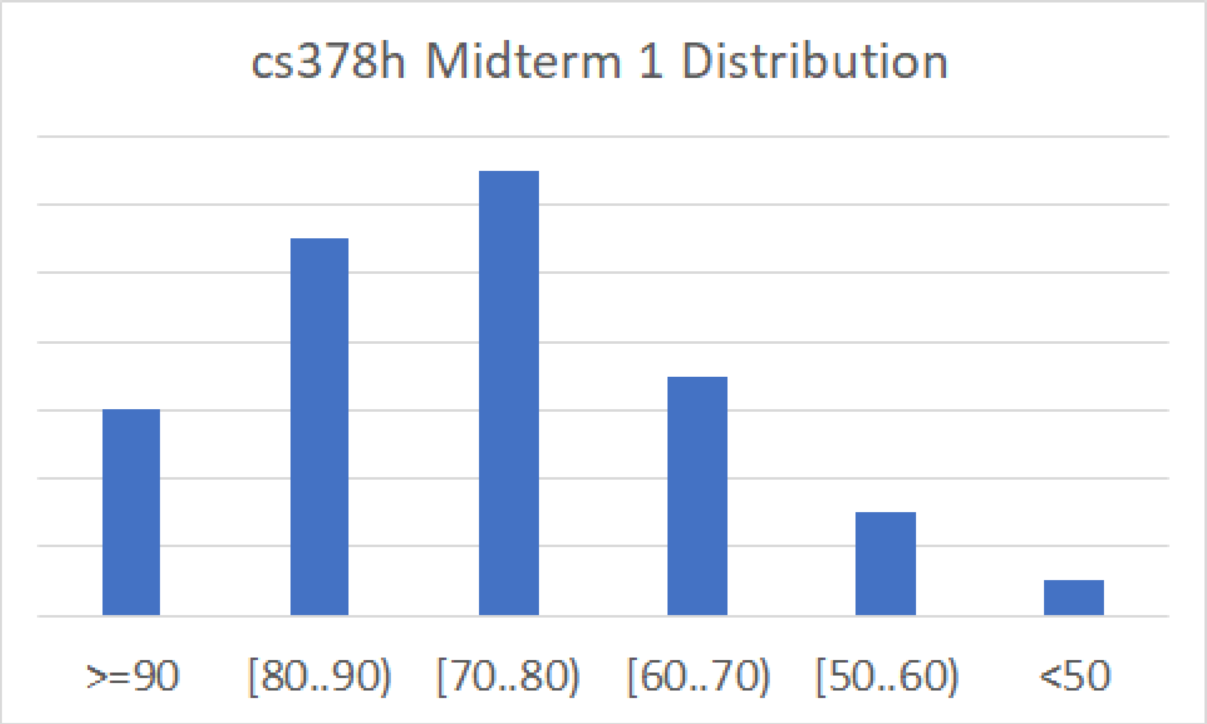
Midterm 1

- Mean: 76.3
- Median: 78



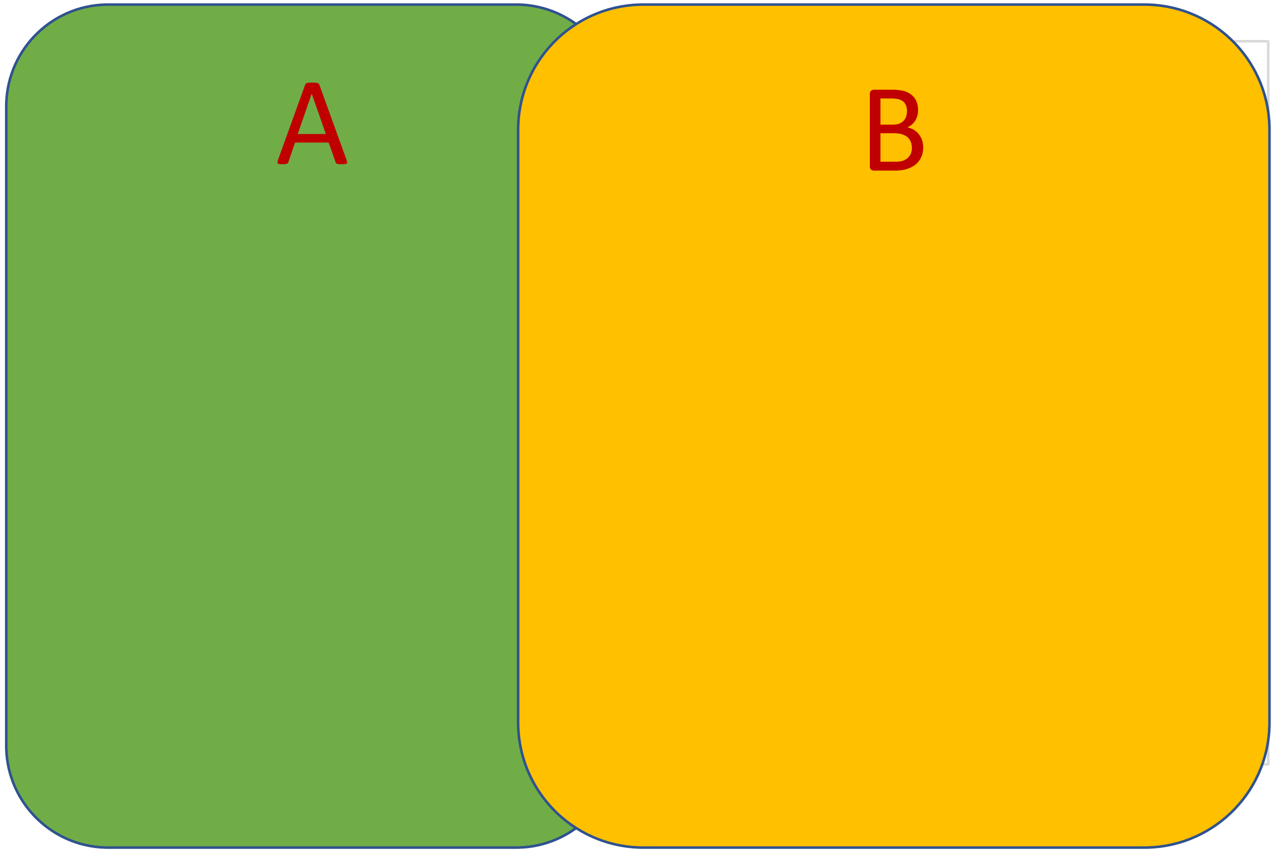
Midterm 1

- Mean: 76.3
- Median: 78
- STDEV: 11.1



Midterm 1

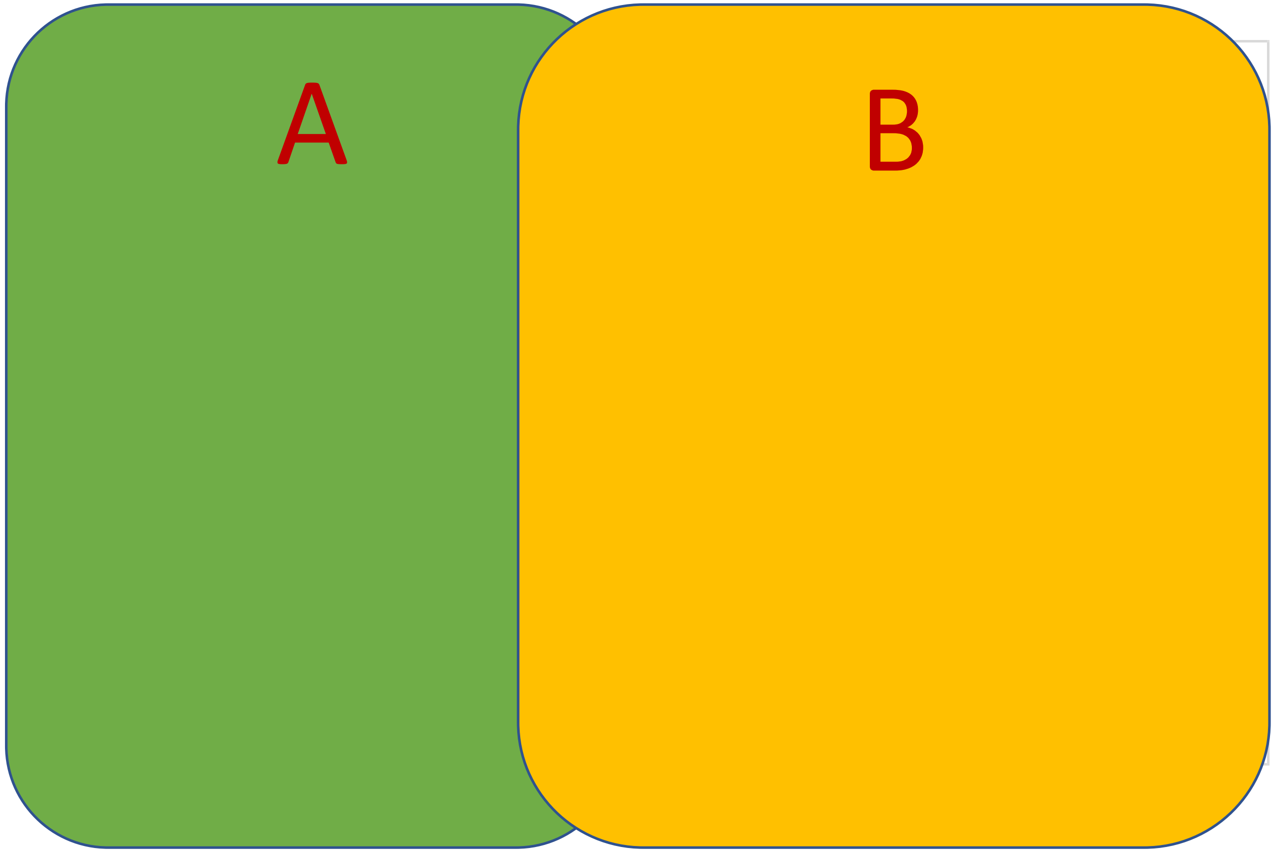
- Mean: 76.3
- Median: 78
- STDEV: 11.1



Midterm 1

Favorite subjects:
Go, GPUs
Least Favorite:
pfxsum, GPUs

- Mean: 76.3
- Median: 78
- STDEV: 11.1



Exam: 1.1.

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: 1.1.

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

   [illegible]

   (b) Spinlocks

   [illegible]

   (d) Atomic instructions

   (e) Bus locking

   (f) Processes and threads

Exam: 1.2.

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: 1.2.

2. Which of the following are true of threads?

   (a) They have their own page tables.

   (b) [illegible]

   (c) [illegible]

   (d) They must be implemented by the OS.

   (e) [illegible]

Exam: 1.4.

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: 1.4.

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) [illegible]

   (d) [illegible]

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

Exam: 1.5.

5. Barriers can be used to implement

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

Exam: 1.5.

5. Barriers can be used to implement

Exam 2.1

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: 2.4.

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: 2.4.

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: 2.4.

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: 2.4.

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 2.5

Exam 2.5

```
double atomicAdd(double *data, double val) {

    while(atomicExch(&locked, 1) != 0)
        ;  // spin

    double old = *data;
    *data = old + val;
    locked = 0;
    return old;
}
```

Exam 2.5

```
double atomicAdd(double *data, double val) {

    while(atomicExch(&locked, 1) != 0)
        ;  // spin

    double old = *data;
    *data = old + val;
    locked = 0;
    return old;
}
```

A) divergence

Exam 2.5

```
double atomicAdd(double *data, double val) {

    while(atomicExch(&locked, 1) != 0)
        ;  // spin

    double old = *data;
    *data = old + val;
    locked = 0;
    return old;
}
```

A) divergence

B) at least 1 block, N threads

Exam 2.5

```
double atomicAdd(double *data, double val) {

    while(atomicExch(&locked, 1) != 0)
        ;  // spin

    double old = *data;
    *data = old + val;
    locked = 0;
    return old;
}
```

A) divergence

B) at least 1 block, N threads

C) N blocks, 1 thread/block

Exam 2.5

```
double atomicAdd(double *data, double val) {

    while(atomicExch(&locked, 1) != 0)
        ;  // spin

    double old = *data;
    *data = old + val;
    locked = 0;
    return old;
}
```

A) divergence
B) at least 1 block, N threads
C) N blocks, 1 thread/block
D) CAS loop is OK,
- *All threads just can't get the lock!*

Exam: 3.1.

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?

Exam: 3.1.

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

Exam: 3.1.

- A) spin on local go flag
- B) some kind of blocking

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?

Exam: 3.1.

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: 3.2.

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: 3.2.

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: 3.2.

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()

Exam: 3.2.

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: 3.2.

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-Ripping

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFile(file);
        RedrawScreen();
    }
    OnOpenFile() {


        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-Ripping

```
PROGRAM MyProgram {
    TASK ReadFileAsync(name, callback) {
        ReadFileSync(name);
        Call(callback);
    }
    CALLBACK FinishOpeningFile() {
        LoadFil
        RedrawScreen();
    }
    OnOpenFile() {


        InitFileName(szName);
        EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
    }
    OnPaint();
}
```

Stack-Ripping

```
PROGRAM MyProgram {
        TASK ReadFileAsync(name, callback) {
                ReadFileSync(name);
                Call(callback);
        }
        CALLBACK FinishOpeningFile() {
                LoadFil
                RedrawScreen();
        }
        OnOpenFile() {


                InitFileName(szName);
                EnqueueTask(ReadFileAsync(szName, FinishOpeningFile));
        }
        OnPaint();
}
```

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

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…

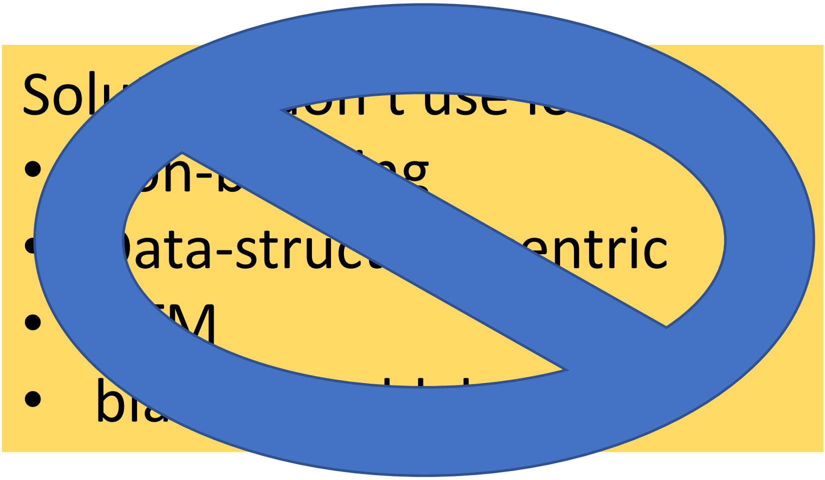
Solution: don't use locks

- non-blocking
- Data-structure-centric
- HTM
- blah, blah, blah..

Rust Motivation

Locks' litany of problems:

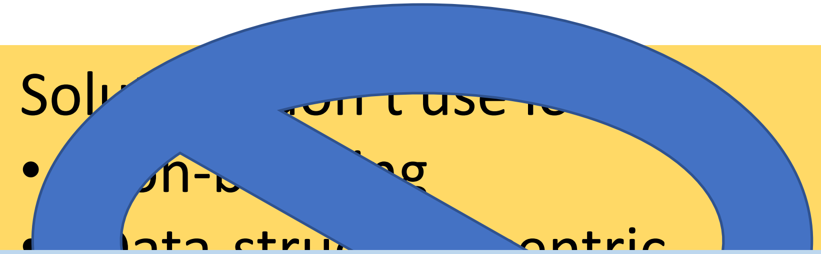
- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance
- Poor composability…



Rust Motivation

Locks' litany of problems:

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



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

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

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

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

Ownership/Borrowing

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

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);
}
```

```
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);
}
```

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

```
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
```

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?

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);
}
```

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

Shared Borrowing

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

Lend the string

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

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

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

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

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

Shared Borrowing with Concurrency

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

```
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

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

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

Clone, Move

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

```
fn helper(name: String) {
  thread::spa[illegible]|| {
    println!("{}", name);
  });
}
```

Explicitly take ownership

Clone, Move

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

Ensure concurrent owners
Work with different copies

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

Explicitly take ownership

Clone, Move

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

Ensure concurrent owners
Work with different copies

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

Explicitly take ownership

Is this better?

Clone, Move

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

Ensure concurrent owners
Work with different copies

Is this better?

```
fn helper(name: String) {
  thread::spa            || {
    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);
    }
}
```

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

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

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

Mutability

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

impl Structure {
    fn mutate(            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

```
fn main() {
```

Sharing State: Channels

```
fn main() {
  let (tx0, rx0) = channel();
```

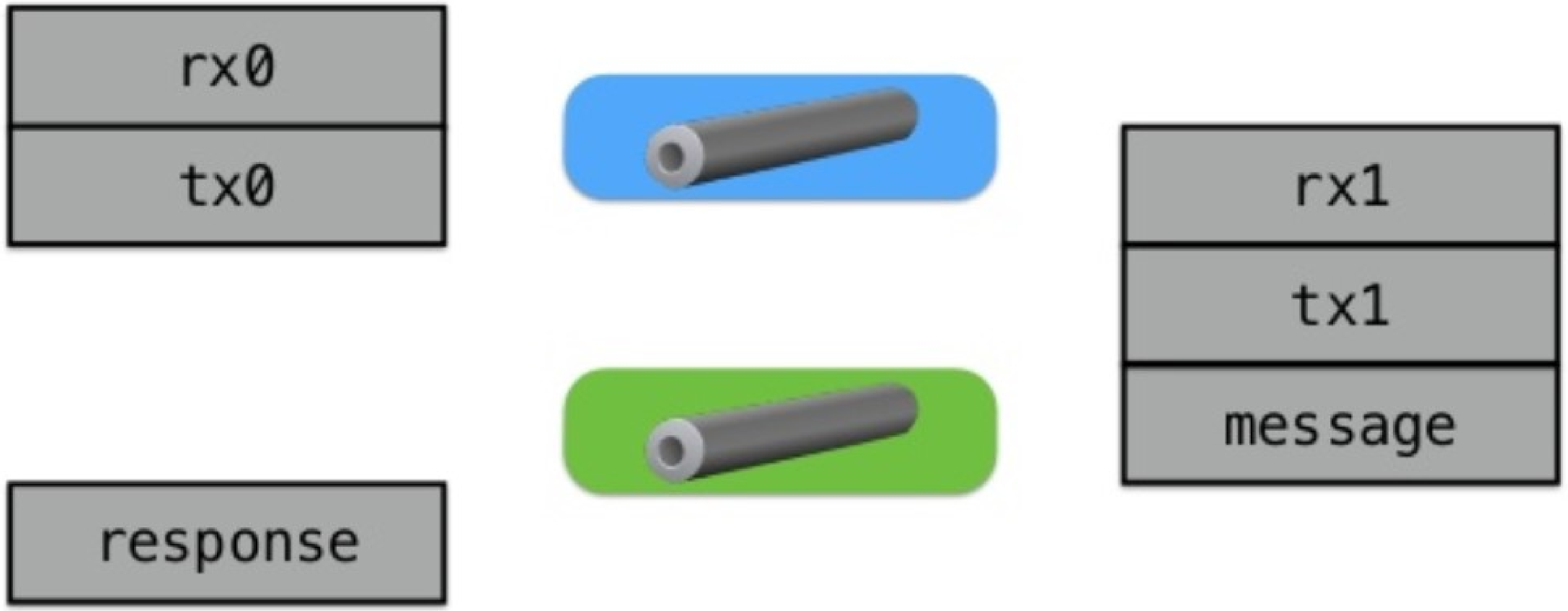
Sharing State: Channels

```
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

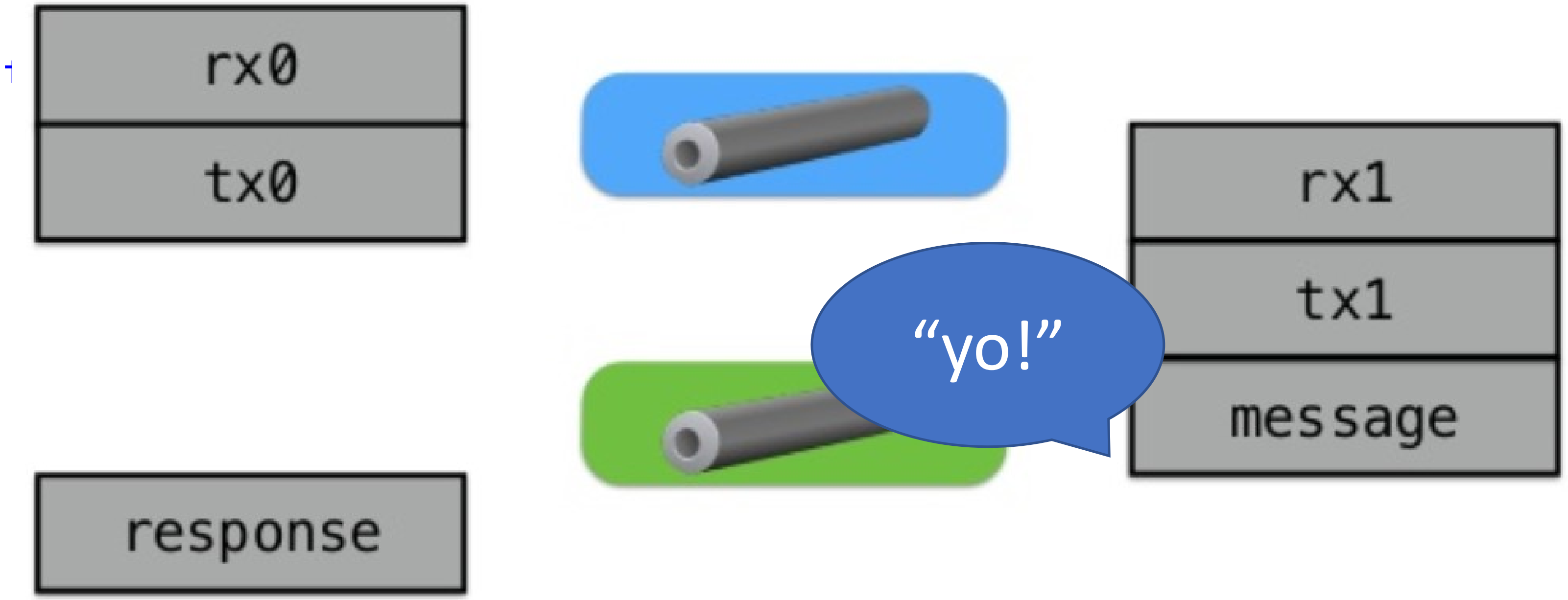
```
fn main() {
  let (tx0, rx0) = channel();
  thread::spawn(move || {
    let (tx1, rx1) = channel();
    tx0.send((format!("yo"), tx1)).unwrap();
    let response = rx1.recv().unwrap();
    println!("child got {}", response);
  });
  let (message, tx1) = rx0.recv().unwrap();
  tx1.send(format!("what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

Sharing State: Channels



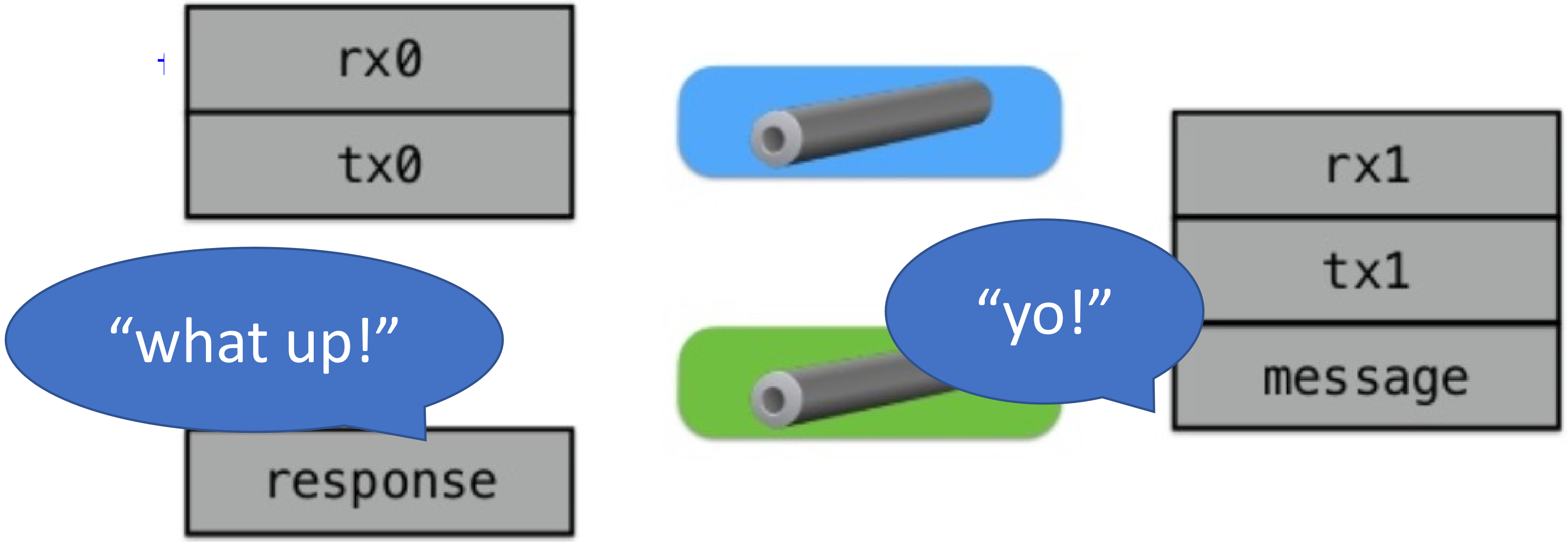

```
  let (message, tx1) = rx0.recv().unwrap();
  tx1.send(format!("what up!")).unwrap();
  println("parent received {}", message);
}
```

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

```
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

```
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);
}
```

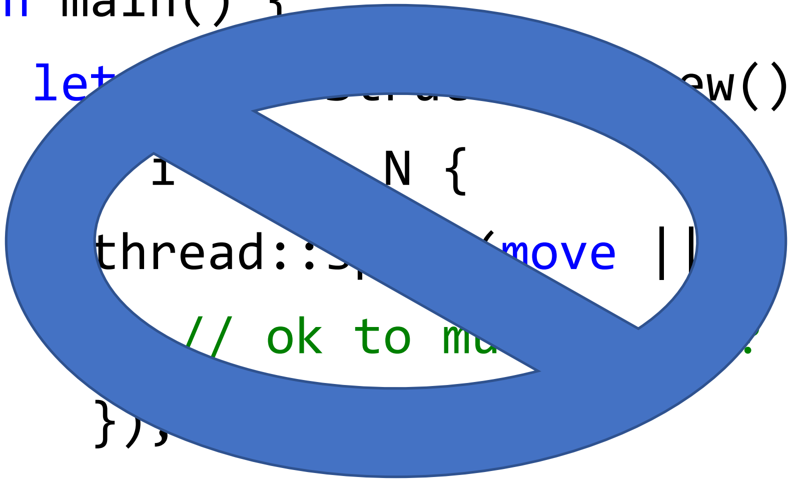
APIs return Option<T>

Sharing State

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

Sharing State

```
fn main() {
  let                    ew();
      i      N {
    thread::              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)!
    });
  }
}
```

Sharing State: Arc and Mutex

```
fn main() {
  let var = Structure::new();
  let [illegible] = 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 [illegible] 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 = A[illegible]&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 = [illegible]();
      // 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)!
    });
  }
}
```

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!