



Intro to Rust Lang Parallelism

Today: Parallelism

- Parallelism
 - Workers, Processes, and Threads
 - Multithreading
- Mutual Exclusion and `Mutex<T>`
- Atomics
- Message Passing
- `thread::scope`
- `Arc<T>`

Parallelism vs Concurrency

Concurrency

Problem of handling many tasks at once

Parallelism

Solution of working on multiple tasks at the same time

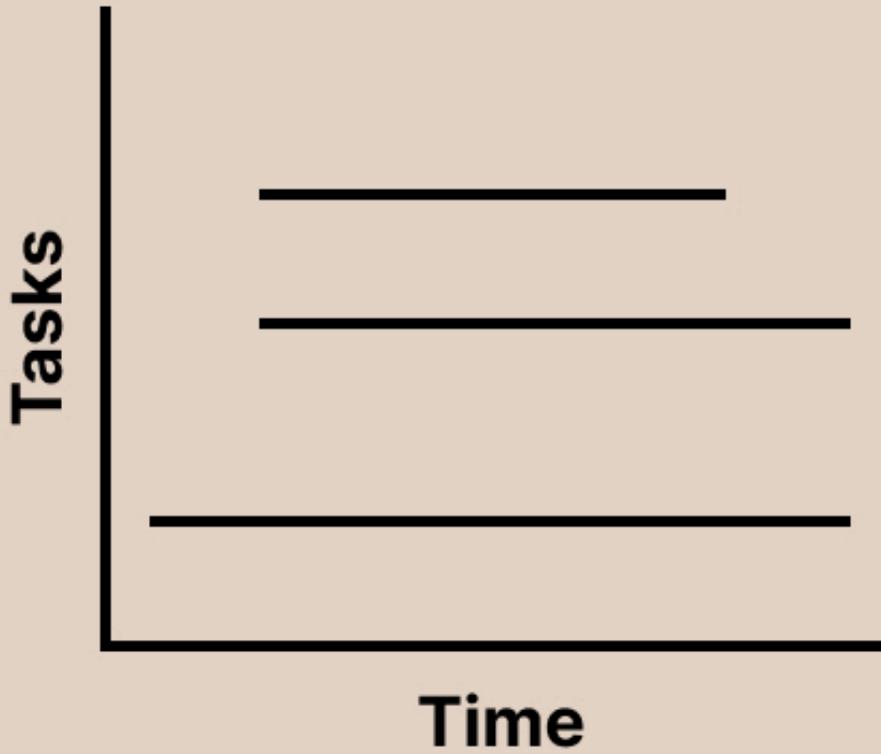
- **Key difference:** Parallelism utilizes **multiple** workers

Workers

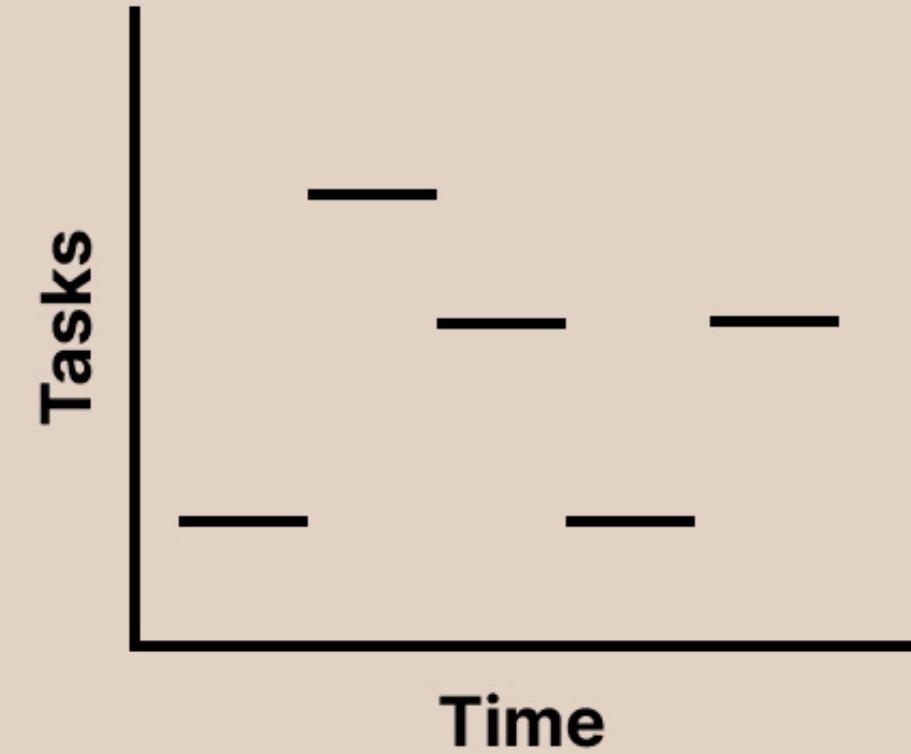
Parallelism divides tasks among workers.

- In hardware, we call these workers **processors** and **cores**
- In software, workers are abstracted as **processes** and **threads**
 - Processes contain many threads
- Parallelism requires **multiple** workers
 - Concurrency models can have any number of workers!

Parallelism



Concurrency



Designing Parallel Solutions

There are two important questions we need to ask:

- Division of labour
 - Who are the workers and how do we divide the work?
- Thread communication
 - What information needs to be shared and how?
 - Approach 1: Shared Memory
 - Approach 2: Message Passing

Multithreading

For today, we will simplify terminology (that tends to get overloaded).

- Our workers are *threads*
- You can think of threads as a "stream of instructions"

The Main Thread

The thread that runs by default is the main thread.

```
fn main() {
    for i in 1..3 {
        println!("Main thread says: Hello {i}!");
        thread::sleep(Duration::from_millis(1));
    }
}
```

- So far, we have only been running code on the main thread!

Spawning a Thread

We can create (spawn) more threads using `std::thread::spawn`.

```
fn main() {
    let child_handle = thread::spawn(|| {
        for i in 1..=8 {
            println!("Child thread says: Hello {i}!");
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..=3 {
        println!("Main thread says: Hello {i}!");
        thread::sleep(Duration::from_millis(1));
    }
}
```



Spawning a Thread

```
let child_handle = thread::spawn(|| {
    for i in 1..=8 {
        println!("Child thread says: Hello {i}!");
        thread::sleep(Duration::from_millis(1));
    }
});  
  
for i in 1..=3 {
    println!("Main thread says: Hello {i}!");
    thread::sleep(Duration::from_millis(1));
}
```

- `thread::spawn` takes an `FnOnce` closure
 - The closure will be run on the newly-created thread
- Question: What should this program's output be?



Possible Output 1

Here is one possible program output:

```
Main thread says: Hello 1!
Child thread says: Hello 1!
Child thread says: Hello 2!
Main thread says: Hello 2!
Child thread says: Hello 3!
Main thread says: Hello 3!
Child thread says: Hello 4!
Child thread says: Hello 5!
```

Possible Output 2

Here is another possible program output:

```
Main thread says: Hello 1!  
Child thread says: Hello 1!  
Main thread says: Hello 2!  
Main thread says: Hello 3
```

Possible Output 3

And here is yet another possible program output!

```
Main thread says: Hello 1!  
Child thread says: Hello 1!  
Main thread says: Hello 2!  
Child thread says: Hello 2!  
Main thread says: Hello 3!  
Child thread says: Hello 3!
```

- What's going on here?

Multithreaded Code is Non-Deterministic

- Most of the code we have written this semester has been deterministic
- The only counterexamples have been when we've used random number generators or interacted with I/O
- The execution of multi-threaded code can interleave with each other in undeterminable ways!

Multithreaded Code is Non-Deterministic

In this example, the child thread can interleave its prints with the main thread.

```
let child_handle = thread::spawn(|| {
    for i in 1..=8 {
        println!("Child thread says: Hello {i}!");
        thread::sleep(Duration::from_millis(1));
    }
}) ;

for i in 1..=3 {
    println!("Main thread says: Hello {i}!");
    thread::sleep(Duration::from_millis(1));
}
```

- Why doesn't the child thread print "Hello" 8 times?

Process Exit Kills Threads

- When the main thread finishes execution, the process it belongs to exits
- Once the process exits, all threads in the process are killed
- If we want to let our child threads finish, we have to wait for them!

Joining Threads

We can `join` a thread when we want to wait for it to finish.

```
// Spawn the child thread.  
let child_handle = thread::spawn(|| time-consuming_function());  
  
// Run some code on the main thread.  
main_thread_foo();  
  
// Wait for the child thread to finish running.  
child_handle.join().unwrap();
```

- `join` will block the calling thread until the child thread finishes
 - In this example, the calling thread is the main thread

Joining Threads

If we go back to our original example and add a `join` on the child's handle at the end of the program, then the child thread can run to completion!

```
Main thread says: Hello 1!
Child thread says: Hello 1!
Main thread says: Hello 2!
Child thread says: Hello 2!
Child thread says: Hello 3!
Main thread says: Hello 3!
Child thread says: Hello 4!
Child thread says: Hello 5!
Child thread says: Hello 6!
Child thread says: Hello 7!
Child thread says: Hello 8!
```

Example: Multithreaded Drawing

Suppose we're painting an image to the screen, and we have eight threads.

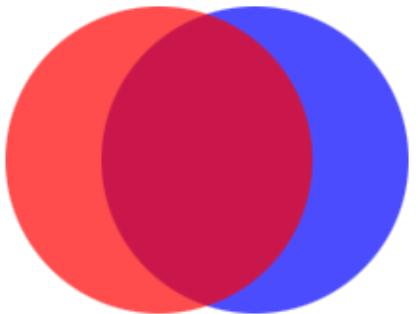
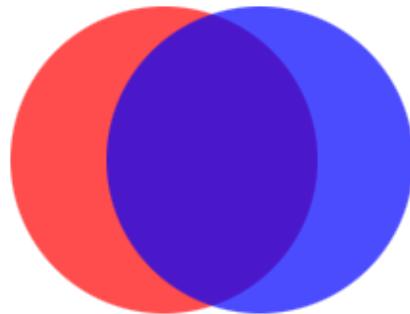
How should we divide up the work between the threads?

- Divide image into eight regions
- Assign each thread to paint one region
- Easy! "Embarrassingly parallel"
 - Threads don't need to keep tabs on each other

Example: Multithreaded Drawing

What if our image is more complex?

- We could be painting semi-transparent circles
- Circles could overlap and/or could be constantly moving
- The *order* in which we paint circles changes what we need to draw



The Case for Communication

If we want consistent drawings, we need a way for our threads to talk / communicate with each other!

- For each pixel:
 - Which color circle should it draw?
 - How many circles have been drawn on this pixel?
 - Do we need to wait for other circles to finish being drawn?
- Simplification for today: each pixel tracks how many circles it has to draw

Motivating Communication

Problem: How do threads communicate?

Solutions: We'll discuss two approaches...

- Approach 1: Shared Memory
- Approach 2: Message Passing

Approach 1: Shared Memory

For each pixel, create a shared variable `x` that represents the number of circles that overlap on this pixel:

```
static int x = 0; // One per pixel.
```

- Note that this is C pseudocode: we'll explain the Rust way soon
- When a thread touches a pixel, increment the pixel's associated `x`
- Now each thread knows how many layers of paint there are on that pixel

Approach 1: Shared Memory

Are we done?

- Not quite...
- Shared memory is prone to **data races**

Shared Memory: Data Races

Step 1: `x` is in shared memory, and `x` must uphold some invariant to be correct.

```
// Invariant: `x` is total number of times **any** thread has called `update_x`.
static int x = 0;

static void update_x(void) {
    x += 1;
}
```

Shared Memory: Data Races

Step 2: `x` temporarily becomes incorrect (mid-update).

```
// Invariant: `x` is total number of times **any** thread has called `update_x`.
static int x = 0;

static void update_x(void) {
    int temp = x;    // `x` lags the number of times `update_x` has been called by 1.
    temp += 1;        // `x` is still incorrect...
    x = temp;        // `x` is now correct!
}
```

- We don't actually write code like this
 - This is how it gets compiled to machine instructions

Shared Memory: Data Races

Step 3: Multiple threads update  at the same time...

```
// Invariant: `x` is total number of times **any** thread has called `update_x`.
static int x = 0;

static void update_x(void) {
    x += 1;
}

// <!-- snip -->

for (int i = 0; i < 20; ++i) {
    spawn_thread(update_x);
}
```

Shared Memory: Data Races

Step 3: When multiple threads update x at the same time... they interleave!

Thread 1	Thread 2
temp = x	
	temp = x
temp += 1	
	temp += 1
x = temp	
	x = temp

Shared Memory: Data Races

We want $x = 2$, but we could get $x = 1$!

Thread 1	Thread 2
Read $\text{temp} = 0$	
	Read $\text{temp} = 0$
Set $\text{temp} = 1$	
	Set $\text{temp} = 1$
Set $x = 1$	
	Set $x = 1$

Shared Memory: Atomicity

We want the update to be **atomic**. That is, other threads cannot cut in mid-update.

Not Atomic

Thread 1	Thread 2
temp = x	
	temp = x
temp += 1	
	temp += 1
x = temp	
	x = temp

Atomic

Thread 1	Thread 2
temp = x	
temp += 1	
x = temp	
	temp = x
	temp += 1
	x = temp

Fixing a Data Race

We must eliminate one of the following:

1. x is in shared memory
2. x temporarily becomes incorrect (mid-update)
3. Unsynchronized updates (parties can "cut in" mid-update)

Mutual Exclusion

Approach 1: Mutual Exclusion

Take turns! No "cutting in" mid-update.

1. x is in shared memory
2. x temporarily becomes incorrect (mid-update)
3. ~~Unsynchronized updates (parties can "cut in" mid-update)~~

Approach 1: Mutual Exclusion

We need to establish *mutual exclusion*.

- You can think of mutual exclusion as "Only one thread at a time"
 - Mutual exclusion means threads don't interfere with each other
- A common tool for this is a `mutex` lock

Mutual Exclusion in C

Here is how you would use a `mutex` in C:

```
static int x = 0;
static mutex_t x_lock;

static void run_thread(void) {
    mtx_lock(&x_lock);
    x += 1;
    mtx_unlock(&x_lock);
}
```

- Only one thread can hold the mutex lock (`mutex_t`) at a time
- Other threads block / wait until they get their turn to hold the lock
- Each thread gets "mutual exclusion" over `x`

Mutual Exclusion in Rust

In Rust, the `Mutex` exists in the standard library!

```
use std::sync::Mutex;

// Details incoming in a few slides...
fn main() {

    let m: Mutex<i32> = Mutex::new(42);

    {
        let mut num = m.lock().unwrap();
        *num += 1;
    }

    println!("m = {:?}", m);
}
```

Mutex<T>

Rust's `Mutex` is a smart pointer!

- `Mutex<T>` owns the data it protects
 - In C, you as the programmer have to ensure the mutex is locked correctly
 - In Rust, you cannot access the data unless you hold the lock!
- Rust can achieve this using a `MutexGuard<'a, T>`
 - You can think of a `Mutex` as the lock provider and the `MutexGuard` as the lock itself

MutexGuard<'a, T>

The `MutexGuard<'a, T>` created from the `Mutex<T>` is also a smart pointer!

```
let m: Mutex<i32> = Mutex::new(42);
{
    // Create a `MutexGuard` that gives the current thread exclusive access.
    let guard1 = m.lock().expect("lock was somehow poisoned");

    // Dereference the guard to get to the data.
    let num = *guard1;
    println!("{}{}", num);
}
// At the end of the scope, `guard1` gets dropped and the lock is released.

// Since the first guard was dropped, we can take the lock again!
let guard2 = m.lock().expect("lock was somehow poisoned");
```

- Ask us after lecture if you're interested in why there is a *lifetime* there...

Mutex::lock

Here is the signature for `Mutex::lock`:

```
pub fn lock(&self) -> LockResult<MutexGuard<'_, T>> {}
```

- Do you notice anything strange about this?
- Remember that you are able to *mutate* the data once you have the guard

Interior Mutability

```
pub fn lock(&self) -> LockResult<MutexGuard<'_, T>> {}
```

- This allows multiple workers to access the lock without a mutable reference
- Even though `lock` takes an *immutable* reference, we are allowed to *mutate* the data that the guard is protecting
- We call this *interior mutability*
- More experienced rustaceans call references "shared" and "exclusive" instead of "immutable" and "mutable" for this exact reason



Atomics

Approach 2: Atomics

One airtight, *atomic* update! Cannot be "temporarily incorrect" mid-update.

1. x is in shared memory
2. * ~~temporarily becomes incorrect (mid-update)~~
3. Unsynchronized updates (parties can "cut in" mid-update)

Code to Machine Instructions

Recall that the compiler will usually translate the following operation...

```
x += 1;
```

...into the machine instruction equivalent of this code:

```
int temp = x;  
temp += 1;  
x = temp;
```

Atomics

However, we can use an atomic operation like this:

```
__sync_fetch_and_add(&x, 1); // syntax depends on library
```

...which is implemented in hardware with just one instruction:

```
x += 1;
```

- `fetch_and_add` : performs the operation suggested by the name, and returns the value that was previously in memory
 - Also `fetch_and_sub` , `fetch_and_or` , `fetch_and_and` , ...

Aside: `compare_and_swap`

Another common atomic operation is `compare_and_swap`

- If the current value matches an old value, update the value
 - Returns whether or not the value was updated
- You can do "lock-free" programming with just CAS
 - No locks! Just `compare_and_swap` until we successfully write new value
 - *Not necessarily more performant than lock-based solutions*

Atomics

These atomic operations are also implemented in the Rust standard library.

```
use std::sync::atomic::{AtomicI32, Ordering};

let x = AtomicI32::new(0);

x.fetch_add(10, Ordering::Relaxed);
x.fetch_sub(2, Ordering::AcqRel);

println!("Atomic Output: {}", x.load(Ordering::SeqCst));
```

```
Atomic Output: 8!
```



Atomics

```
let x = AtomicI32::new(0);

x.fetch_add(10, Ordering::Relaxed);
x.fetch_sub(2, Ordering::AcqRel);

println!("Atomic Output: {}!", x.load(Ordering::SeqCst));
```

- The API is largely identical to C++20 atomics
- If you're interested in what the Ordering is, look up "memory ordering"

Atomic Action

Here is an example of incrementing an atomic counter from multiple threads!

```
static counter: AtomicUsize = AtomicUsize::new(0);

fn main() {
    // Spawn 100 threads that each increment the counter by 1.
    let handles: Vec<JoinHandle<_>> = (0..100)
        .map(|_| thread::spawn(|| counter.fetch_add(1, Ordering::Relaxed)))
        .collect();

    // Wait for all threads to finish.
    handles.into_iter().for_each(|handle| { handle.join().unwrap(); });

    assert_eq!(counter.load(Ordering::Relaxed), 100);
}
```

Message Passing

Eliminating Shared Memory

If we eliminate shared memory...

1. ~~* is shared memory~~
2. x becomes incorrect mid-update
3. Unsynchronized updates

Eliminating Shared Memory

If we eliminate shared memory... any data races are trivially gone.

1. ~~* is shared~~
2. ~~* becomes incorrect mid-update~~
3. ~~Unsynchronized updates~~

Message Passing

Now we'll talk about the second approach to communication, message passing!

- Approach 1: Shared Memory
- Approach 2: Message Passing
 - **Eliminates shared memory**

Message Passing

- Threads communicate via *channels*
 - You can think of a sending thread sending a message down a river
 - Receiving thread picks up the message downstream
- Popular mechanism for concurrency and synchronization in the Go language

Creating channels

Channels consist of both a transmitter and a receiver.

```
let (tx, rx) = mpsc::channel();
```

- **Transmitter:** Connor writes "Review the ZFOD PR" and sends it down a river
- **Receiver:** Ben finds the duck downstream and reads the message
- Each channel can only transmit/receive one type
- Communication is only one-way
- This is an `mpsc`-flavored channel...

Message Passing Example

Here's an example of a child thread sending a message to the main thread.

```
let (tx, rx) = mpsc::channel();

thread::spawn(move || { // Takes ownership of `tx`.
    let val = String::from("review the ZFOD PR!");
    tx.send(val).unwrap(); // Send `val` through the transmitter.
});

let received = rx.recv().unwrap(); // Receive `val` through the receiver.
println!("I am too busy to {}!", received);
```

- After we send `val`, we no longer have ownership of it

Message Passing Iterator

We can also use receivers as iterators!

```
let (tx, rx) = mpsc::channel();

thread::spawn(move || { // Takes ownership of `tx`.
    let val = String::from("review the ZFOD PR!");

    tx.send(val).unwrap(); // Send `val` through the transmitter.
    tx.send("buy Connor lunch").into().unwrap(); // Send another message!
});

for msg in rx { // Iterate through messages until all transmitters are dropped.
    println!("I am too busy to {}!", msg);
}
```

- Wait, what does `mpsc` stand for?

mpsc \Rightarrow Multiple Producer, Single Consumer

This means we can `clone` the transmitter to get *multiple message producers*.

```
let (tx, rx) = mpsc::channel();

let tx_clone = tx.clone();
thread::spawn(move || { // Takes ownership of `tx_clone`.
    tx_clone.send("yo".into()).unwrap();
    thread::sleep(Duration::from_secs(1));
} );

thread::spawn(move || { // Takes ownership of `tx`.
    tx.send("hello".into()).unwrap();
    thread::sleep(Duration::from_secs(1));
} );
```



Parallelism in Rust

Threads in Rust

Rust uniquely provides some nice guarantees for parallel code, and at the same time introduces a few complications...

- Rust's typechecker guarantees an absence of *any data races*
 - Unless you use `unsafe`
 - This is a super powerful guarantee!
- General race conditions are not prevented (non-determinism)
 - Deadlocks are still possible

Creating Threads: Detailed

Recall that threads can be spawned with `thread::spawn`.

```
let child_handle = thread::spawn(|| {
    for i in 1..=8 {
        println!("Child thread says: Hello {i}!");
        thread::sleep(Duration::from_millis(1));
    }
});
```

- `thread::spawn` takes in a closure, implementing the `FnOnce` and `Send` traits
 - `FnOnce` implies we cannot spawn multiple threads of the same closure
 - `Send` is a marker trait that says the type is safe to send to another thread
 - You can read more about `Send` [here in the Rustonomicon](#)

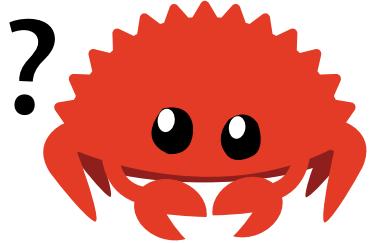


Capturing Values in Threads

What if we want to use a value from outside the closure?

```
let v = vec![1, 2, 3];

thread::spawn(|| {
    println!("Here's a vector: {:?}", v);
});
```



Capturing Values in Threads

```
error[E0373]: closure may outlive the current function, but it borrows `v`, which is owned by the current function
--> src/main.rs:6:19
6 |     thread::spawn(|| {
   |         ^^^ may outlive borrowed value `v`
7 |         println!("Here's a vector: {:?}", v);
   |                     - `v` is borrowed here
|
note: function requires argument type to outlive ``static``
--> src/main.rs:6:5
6 | /     thread::spawn(|| {
7 | |         println!("Here's a vector: {:?}", v);
8 | |     );
   | |____^
help: to force the closure to take ownership of `v` (and any other referenced variables), use the `move` keyword
6 |     thread::spawn(move || {
   |             +---
```

- In other words, what if `v` goes out of scope while the thread is still running?



Moving Values into Threads

To solve this, we can take ownership of values by *moving* them into the closure.

```
let v = vec![1, 2, 3];

let handle = thread::spawn(move || {
    println!("Here's a vector: {:?}", v);
});
```

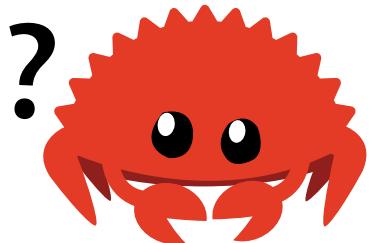
Access from Different Threads

What if we want `v` to remain accessible in the main thread?

```
fn main() {
    let v = vec![1, 2, 3];

    thread::spawn(move || {
        println!("Vector from child: {:?}", v);
    });

    // THIS DOESN'T COMPILE!
    println!("Vector from parent: {:?}", v);
}
```



Access from Different Threads

- We could clone `v` to keep it accessible in the main thread
 - But cloning can be expensive
 - And what if we actually want to share `v` between the threads?
- There are at least two alternatives to cloning here:
 - Approach 1: `thread::scope`
 - Approach 2: `Arc` and `Mutex`

Local Data?

Suppose we're writing a function to process a large array in parallel:

```
fn main() {
    let mut data = [1, 2, 3, 4, 5, 6]; // Pretend this is much larger...
    compute_squares(data);
}
```

- The array is local to the function (stack-allocated)
 - We don't want to move ownership
 - We don't want to allocate it on the heap unnecessarily

Approach 1: `thread::scope`

`thread::scope` creates a scope for spawning threads that *borrow*s local data.

```
fn compute_squares(numbers: &mut [i32]) {
    thread::scope(|s| {
        let mid = numbers.len() / 2;
        let (left, right) = numbers.split_at_mut(mid);

        let t1 = s.spawn(/* do stuff with `left` */);
        let t2 = s.spawn(/* do stuff with `right` */);
    });
}
```

- `thread::scope`'s closure takes a `Scope` object `s`
 - You use this `s` to spawn threads with `s.spawn`

Approach 1: `thread::scope`

Threads are joined automatically when the scope exits

```
fn compute_squares(numbers: &mut [i32]) {
    thread::scope(|s| {
        let mid = numbers.len() / 2;
        let (left, right) = numbers.split_at_mut(mid);

        let t1 = s.spawn(/* do stuff with `left` */);
        let t2 = s.spawn(/* do stuff with `right` */);
    });
}
```

- No explicit `join` is needed
 - Very clean!

Approach 1: `thread::scope`

The Rust compiler ensures that the borrowed data (`numbers`) outlives all threads.

```
fn compute_squares(numbers: &mut [i32]) {
    thread::scope(|s| {
        let mid = numbers.len() / 2;
        let (left, right) = numbers.split_at_mut(mid);

        let t1 = s.spawn(/* do stuff with `left` */);
        let t2 = s.spawn(/* do stuff with `right` */);
    });

    // Can still access `numbers`!
    println!("Finished computing squares: {:?}", numbers);
}
```



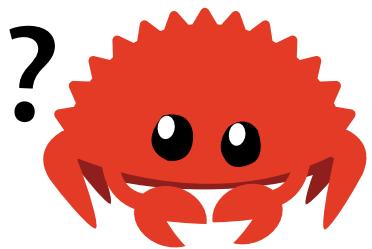
Multiple Thread Access

- With `thread::scope`, we can share local data across threads
- What if we actually wanted different threads to access and own the data?

Multiple Thread Access

Let's say we want two threads to access a mutex-protected vector.

```
let data = Mutex::new(vec![2, 3, 4, 5]);  
  
let t1 = thread::spawn(|| {  
    let mut data_guard = data.lock().unwrap();  
    data_guard.insert(0, 1);  
});  
let t2 = thread::spawn(|| {  
    let mut data_guard = data.lock().unwrap();  
    data_guard.push(6);  
});
```



Multiple Thread Access

```
error[E0373]: closure may outlive the current function, but it borrows  
    `data`, which is owned by the current function
```

```
--> src/main.rs:7:28
```

```
7 |     let t1 = thread::spawn(|| {  
8 |         ^^^ may outlive borrowed value `data`  
9 |         let mut data_guard = data.lock().unwrap();  
10 |             ----- `data` is borrowed here
```

```
help: to force the closure to take ownership of `data` (and any other  
referenced variables), use the `move` keyword
```

```
7 |     let t1 = thread::spawn(move || {  
8 |             +---
```

Multiple Thread Access

- We can't access the local Mutex from the child threads because the parent thread might finish first
- We also cannot `move` the mutex into both threads (can only move into one)
- What we really want is for both threads to *own* the data
 - This should remind you of `Rc<T>` !

Remember `Rc<T>` ?

Recall the `Rc<T>` (Reference Counted) Smart Pointer.

- `Rc<T>` points to a heap-allocated value
 - Keeps track of the number of references to the value (refcount)
 - The data is dropped when the reference count hits zero
- Provides **shared ownership** of the value

Rc<T>

But, `Rc<T>` is not thread-safe... updates to the reference count are not atomic!

Thread 1	Thread 2
<code>temp = refcount</code>	
	<code>temp = refcount</code>
<code>temp += 1</code>	
	<code>temp += 1</code>
<code>refcount = temp</code>	
	<code>refcount = temp</code>

Arc<T>

Rc<T> has an equivalent atomic version Arc<T> .

- "Atomically Reference Counted"
- Same functionality as Rc<T> , but thread-safe
 - Reference count is atomically updated
- Use Rc<T> for single-threaded operations, Arc<T> for multi-threaded

Approach 2: Arc<Mutex<T>>

Here is the other approach, using `Arc<T>` and `Mutex<T>`.

```
let data = Arc::new(Mutex::new(vec![2, 3, 4, 5]));
let data_clone = Arc::clone(&data);

let t1 = thread::spawn(move || {
    let mut data_guard = data.lock().unwrap();
    data_guard.insert(0, 1);
});
let t2 = thread::spawn(move || {
    let mut data_guard = data_clone.lock().unwrap();
    data_guard.push(6);
});
```

- Both threads own the mutex!



Approach 2: Arc<Mutex<T>>

Here is a more practical application of `Arc<Mutex<T>>` :

```
let data = Arc::new(Mutex::new(vec![]));

for i in 0..10 {
    let data = data.clone();
    handles.push(thread::spawn(move || {
        let mut guard = data.lock().expect("lock was somehow poisoned");
        guard.push(i);
    }));
}

// Code for joining handles and printing omitted.
```

```
[1, 0, 3, 6, 5, 4, 2, 7, 9, 8]
```



Send and Sync

- Send and Sync : **marker traits** used to enforce safety with multiple threads
 - Send : indicates that T can be safely sent between threads
 - Sync : indicates that T can be safely referenced between threads
 - *Only if &T implements Send

Other Synchronization Primitives

There are shared state **primitives** other than `Arc<T>` and `Mutex<T>` :

- `RwLock<T>` : a lock which can have concurrent *readers*
- `CondVar<T>` : blocks a thread until a condition is met
- `Barrier` : memory barrier where threads can wait for others to reach it
- **and more...**

Review: "Fearless Concurrency"

Today's content is referred to as "fearless concurrency" in the Rust community:

- By leveraging Rust's type system, we can move entire classes of concurrency and parallelism bugs to compile-time
- Rather than choosing a restrictive "dogmatic" approach to concurrency, Rust supports many different approaches, all of which are completely safe
- Some people believe that this may be the *best reason* to use Rust



Final Homework: The Billion Row Challenge

- Process a billion rows as fast as possible!
- The key objective is to learn to optimize a program by applying parallelism in Rust
- Feel free to use 3rd-party crates ([Rayon](#) should be a useful one)
- Have fun 😊

Next Lecture: Concurrency

Thanks for coming!

Slides created by:

Connor Tsui, Benjamin Owad, David Rudo,
Jessica Ruan, Fiona Fisher, Terrance Chen

