Benjamin Owad, David Rudo, and Connor Tsui

Parallelism vs. Concurrency

Parallelism

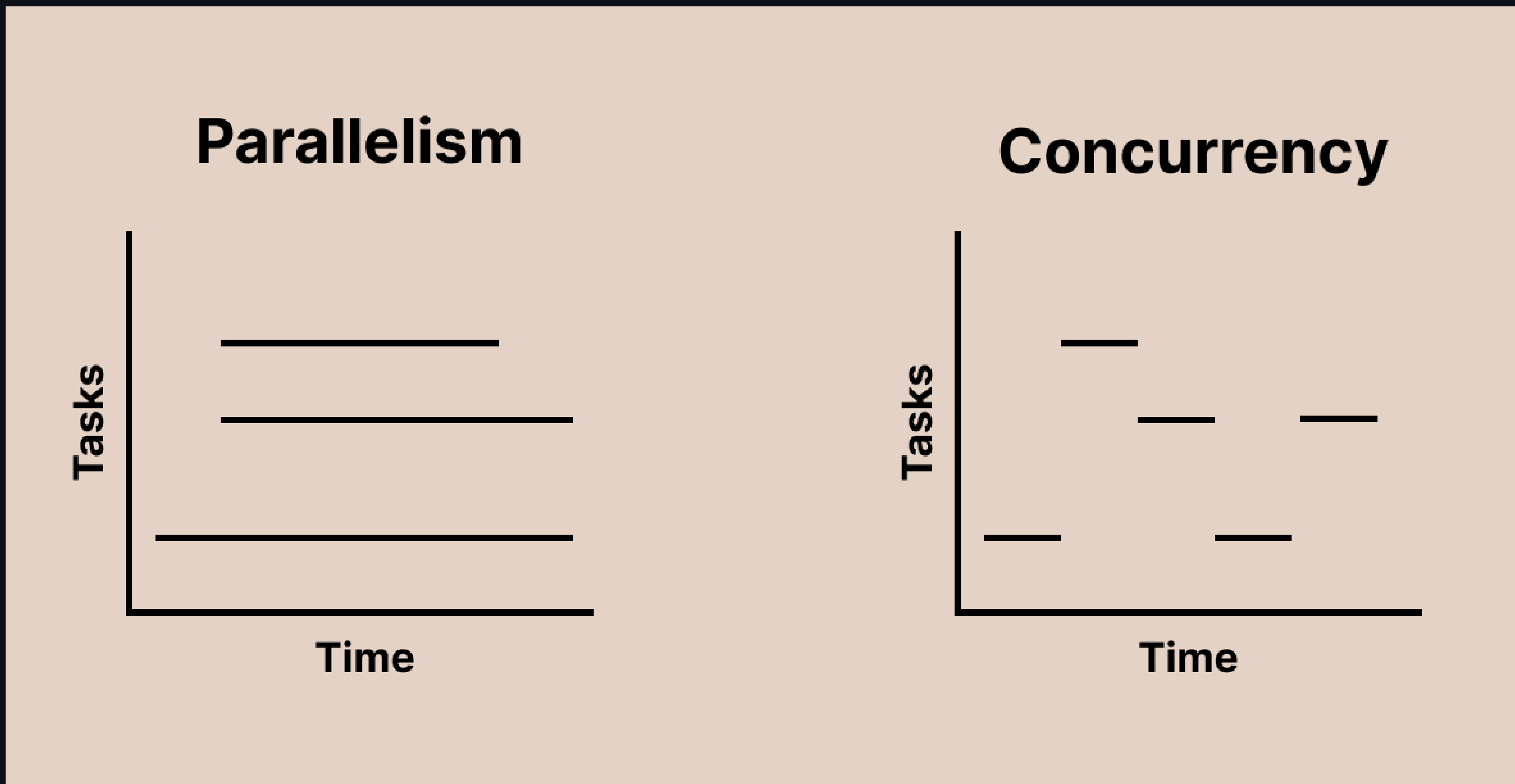
- Work on multiple tasks at the same time
- Utilizes multiple processors/cores

Concurrency

- Manage multiple tasks, but only do one thing at a time.
- Better utilizes a single processor/core

- These terms are used (and abused) interchangably

Parallelism vs. Concurrency



Parallelism vs. Concurrency

Parallelism

Concurrency

Parallelism vs. Concurrency (Examples)

Parallelism

- Have one processor work on loading the webpage, while another updates the progress bar
- Often used to divide tasks into smaller units that can run at the same time
  - e.g. Processing 100x100px regions of an image on each core
  - "Divide and conquer"

Concurrency

- As we load a webpage, take a break sometimes to update the loading progress bar
- Often used to do other things while we wait for blocking I/O operations
  - e.g. Running garbage collection while we wait for a response over the network

Today: Parallelism

- Threads
- Synchronization
- Message Passing
- `Send` and `Sync`
- More Synchronization

Terminology: Threads

- Dangerously overloaded term—can mean one of many things
- For this lecture, we define it as a "stream of instructions"
- In Rust, language threads are 1:1 with OS threads
- **Key point:** Threads share the same resources

Sharing Resources

```
static int x = 0;

static void thread(void) {
  int temp = x;
  temp += 1;
  x = temp;
}
// <!-- snip -->
for (int i = 0; i < 20; ++i) {
  create_thread(thread); // helper function not shown
}
```

- What is the value of `x` after we join on all 20 threads?
  - What is the next slide's title going to be?

Race Conditions

When multiple threads have access to the same data, things get complicated…

- Specifically, this is about *data races*

The Bad Slide

Thread 1	Thread 2
temp = x (temp = 0)	
	temp = x (temp = 0)
temp += 1 (temp = 0 + 1)	
	temp += 1 (temp = 0 + 1)
x = temp (x = 1)	
	x = temp (x = 1)

- Uh oh...

Synchronization

To make sure instructions happen in a reasonable order, we need to establish *mutual exclusion*, so that threads don't interfere with each other.

- Mutual exclusion means "Only one thread can do something at a time"
- A common tool for this is a mutex lock

Sharing Resources With Mutual Exclusion

```c
static int x = 0;
static mtx_t x_lock;

static void thread(void) {
  mtx_lock(&x_lock);
  int temp = x;
  temp += 1;
  x = temp;
  mtx_unlock(&x_lock);
}
// <!-- snip -->
```

- Only one thread can hold the mutex lock at a time
- This provides *mutual exclusion*--only one thread may access `x` at the same time.

Threads in Rust

Threads in Rust

- Rust's typechecker guarantees an absence of *data races*
  - (Unless you use unsafe)
- General race conditions are not prevented
- Deadlocks are still allowed

Creating Threads

Threads can be created/spawned using `thread::spawn`.

```
let handle = thread::spawn(|| {
    for i in 1..10 {
        println!("hi number {} from the spawned thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
});
```

- `thread::spawn` takes in a function, implementing the `FnOnce` and `Send` traits.
    - Closures are often used to allow capturing values, but functions work as well
    - More on the `Send` trait later...
- Returns a `JoinHandle` type

Joining Threads

To wait for a thread to complete, we *join* on it.

```
let handle = thread::spawn(|| {
    for i in 1..10 {
        println!("hi number {} from the spawned thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
});

handle.join().unwrap();
```

- Execution of the main thread is halted until the spawned thread finishes

Capturing Values in Threads

We often want to use things outside of the the closure, but borrowing them can be problematic.

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

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

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

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

Capturing Values in Threads

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

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

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

- `v` is no longer accessible in the main thread
- You could clone `v` to solve this problem
  - But, what if we *wanted* to share `v`?

Multiple Owners

Recall `Rc<T>` from last lecture.

- `Rc<T>` works like `Box<T>`, providing a (spiritually) heap-allocated value.
- The difference is, `Rc<T>` has an internal reference count, and the heap allocated value will only be dropped when the reference count reaches zero.
- The only problem is, `Rc<T>` is not thread safe...

`Arc<T>`

"Arc" stands for "Atomically Reference Counted". This means, it is thread-safe, at the cost of slightly slower operations.

- General advice: default to using `Rc<T>`, and switch to `Arc<T>` if you need to share ownership across threads
  - The compiler will not let you use `Rc` across threads

Sharing Resources in Rust

We can give the vector multiple owners by using an `Arc`.

```
let v = Arc::new(vec![1, 2, 3]);

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

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

handle.join().unwrap();
```

- `v` and `v_copy` both point to the same value
- When both are dropped, only then will the underlying vector be dropped
- Is this a data race?
    - No, because we are only performing reads

Sharing Mutable Resources in Rust

If we attempt to mutate the vector, we will indeed encounter an error

```
let v = Arc::new(vec![1, 2, 3]);

let v_copy = v.clone();
let handle = thread::spawn(move || {
    v_copy.push(4);
    println!("Here's a vector: {:?}", v_copy);
});

v.push(5);
println!("Here's a vector: {:?}", v);

handle.join().unwrap();
```

- This prevents a data race
    - If we allowed this, it would violate one of the rules—only one mutable reference at a time

Sharing Mutable Resources in Rust

```rust
let v = Arc::new(vec![1, 2, 3]);

let v_copy = v.clone();
let handle = thread::spawn(move || {
    v_copy.push(4);
    println!("Here's a vector: {:?}", v_copy);
});

v.push(5);
println!("Here's a vector: {:?}", v);

handle.join().unwrap();
```

```
cannot borrow data in an Arc as mutable
<!-- snip -->
help: trait DerefMut is required to modify through a dereference,
but it is not implemented for Arc<Vec<i32>>
```

Sharing Mutable Resources in Rust

The solution to this is actually the same as in C—we introduce a mutex.

Mutexes in Rust

Unlike in C, mutexes in Rust actually *wrap* values.

```
let x = Mutex::new(0);
let x_data = x.lock().unwrap();
```

- This allows the typechecker to verify that the lock is acquired before accessing a value (and eliminates a class of bugs)
  - If we know this, our multiple mutable references rule is not broken!
- `x_data` is a `MutexGuard` type.
  - It has deref coercion, so one can operate on it just like it was the actual data
- When `x_data` is dropped, the mutex will be unlocked.
- `lock` may return an error if another thread panics

Sharing Mutable Resources in Rust

```rust
let v = Arc::new(Mutex::new(vec![1, 2, 3]));

let v_copy = v.clone();
let handle = thread::spawn(move || {
    v_copy.lock().unwrap().push(4);
    println!("Here's a vector: {:?}", v_copy);
});

v.lock().unwrap().push(5);
println!("Here's a vector: {:?}", v);

handle.join().unwrap();
```

- The other thread cannot access the mutex until it is dropped (unlocked)
- This prevents multiple mutable references, and the data race, by providing mutual exclusion!

C to Rust Example

C to Rust Example

Here's the C code from before, turned into Rust directly.

```rust
let mut x = 0;

for _ in 0..20 {
    thread::spawn(|| {
        x += 1;
    });
}
```

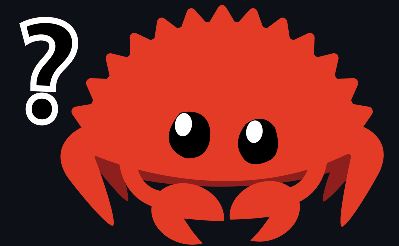
- A sea of errors ensues of course, but the key idea is that this violates one of our rules.
  - We can't have multiple mutable references at the same time!

C to Rust Example (with Mutexes)

Here's our code from before, with mutexes incorporated

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

for _ in 0..20 {
    thread::spawn(|| {
        let mut data = x.lock().unwrap();
        *data += 1;
    });
}
```



- What is wrong now?
    - What if the main function ends? It owns `x`, so the thread references to `x` will be invalid...
- How can we have multiple owners?

C to Rust Example (with Multiple Ownership)

```rust
let x = Arc::new(Mutex::new(0));

for _ in 0..20 {
    let x_clone = Arc::clone(&x);
    thread::spawn(move || {
        let mut data = x_clone.lock().unwrap();
        *data += 1;
    });
}
```

- Notice that we `move` each clone of `x` into each thread, taking ownership of it
- Each thread has a pointer to the mutex
    - The mutex is not deallocated until all of the `Arc`s pointing to it are dropped (and the reference count is zero)

The Good Slide

```rust
let x = Arc::new(Mutex::new(0));
let mut handles = vec![];

for _ in 0..20 {
    let x_clone = Arc::clone(&x);
    handles.push(thread::spawn(move || {
        let mut data = x_clone.lock().unwrap();
        *data += 1;
    }));
}

for handle in handles { handle.join().unwrap(); } // Wait for all threads
println!("Final value of x: {}", *x.lock().unwrap());
```

- `x` is 20, *every time*.
  - And it is illegal for it to be anything else in safe Rust.

Parallelism Checkpoint

Up until now, we have been talking about parallelism with *shared state*. Let's shift gears and talk about *message passing*.

Message Passing

Rather than sharing state between threads, an increasingly popular approach to safe concurrency is message passing.

- In this approach, threads communicate with each other through channels
- Golang famously utilizes this approach

Message Passing Example

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

- Channels have two halves, a transmitter and a receiver
- Connor writes "Review the ZFOD PR" on a rubber duck and it floats down the river (transmitter)
  - Ben finds the duck downstream, and reads the message (receiver)
- Note that communication is one-way here
- Note also that each channel can only transmit/receive one type
  - e.g. `Sender<String>`, `Receiver<String>` can't transmit integers

Message Passing Example

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

thread::spawn(move || { // Take 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);
```

- Note that, after we send `val`, we no longer have ownership of it!

Message Passing Example

We can also use receivers as iterators!

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

thread::spawn(move || { // Take 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();
});

for msg in rx {
  println!("I am too busy to {}!", msg);
}
```

- Wait, what does `mpsc` stand for?

`mpsc` ⟹ Multiple Producer, Single Consumer

This means we can `clone` the transmitter end of the channel, and have *multiple producers*.

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

let tx1 = tx.clone();
thread::spawn(move || { // owns tx1
      tx1.send("yo".into()).unwrap();
      thread::sleep(Duration::from_secs(1));
});

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

for received in rx {
    println!("Got: {}", received);
}
```

`Send` and `Sync`

`Send` and `Sync`

Everything we have gone over so far is a *standard library* feature. The language itself provides two marker traits to enforce safety when dealing with multiple threads, `Send` and `Sync`.

`Send` vs. `Sync`

`Send`

- Indicates that the type is safe to *send* between threads.
- `Rc<T>` does not implement this trait, because it is not thread safe.

`Sync`

- Indicates that the type implementing `Send` can be referenced from multiple threads
- For example, `RefCell<T>` from last lecture implements `Send` but not `Sync`
- `Rc<T>` does not implement `Sync` either

Using `Send` and `Sync`

- It is generally rare that you would implement these traits yourself
  - Structs containing all `Send` / `Sync` types automatically derive `Send` / `Sync`
  - Explicitly implementing either one requires using `unsafe`
- This would be an example of a trait you might want to *unimplement*
  - e.g. If you are doing something with `unsafe` that is not thread-safe
  - `impl !Send for CoolType<T> {}`

More Shared State Primitives

`RwLock<T>` (Reader-Writer Lock)

A reader-writer lock is like a mutex, except it allows concurrent access between readers (not writers).

- We can acquire a read lock (or shared lock)
  - Can be held by multiple readers at once
  - No writers can hold the lock
- We can acquire a write lock (or exclusive lock),
  - Can be held by only one writer
  - No readers can hold the lock

`RwLock<T>` Example

```
let shared_data = Arc::new(RwLock::new(Vec::<i32>::new()));

// All of the readers can hold the read lock simultaneously
for _ in 0..5 {
    let shared_data_clone = Arc::clone(&shared_data);
    thread::spawn(move || {
        let data = shared_data_clone.read().unwrap();
        println!("Reader: {:?}", *data);
    });
}

// The writer has to be the only one with the lock
let shared_data_clone = Arc::clone(&shared_data);
thread::spawn(move || {
    let mut data = shared_data_clone.write().unwrap();
    data.push(42);
    println!("Writer: {:?}", *data);
});
```

Even More Primitives

- `CondVar<T>`—release a mutex and atomically wait to be signaled to re-acquire it
- `Barrier`—Memory barrier, allows multiple threads to wait at a certain point, until all relevant threads reach that point
- `Weak<T>`—downgraded version of `Rc` or `Arc` that holds a pointer, but does not count as an owner.
  - Retrieving the value can fail, if it has been deallocated already.

One more thing...

`std::sync::atomic`

Rust provides atomic primitive types, like `AtomicBool`, `AtomicI8`, `AtomicIsize`, etc.

- Safe to share between threads (implementing `Sync`), providing ways to access the values atomically from any thread
- 100% lock free, using bespoke assembly instructions
- Highly performant, but very difficult to use
- Requires an understanding of *memory ordering*—one of the most difficult topics in computer systems
- We won't cover it further in this course, but the API is largely 1:1 with the C++20 atomics.

Review: "Fearless Concurrency"

What we have gone over today is referred to as "fearless concurrency" in the rust community.

- By leveraging the ownership system, we can move entire classes of concurrency bugs to compile-time
- Rather than choosing a restrictive "dogmatic" approach to concurrency, Rust supports many approaches, *safely*
- Subjectively, this may be the single best reason to use this language
- Both parallelism and concurrency, as introduced in this lecture, benefit from these guarantees

Next Lecture: Concurrency

- Including `async` / `await` !
- Thank you for coming!