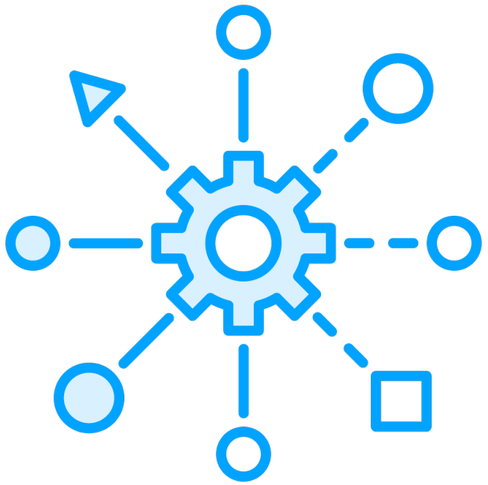
Concurrency and Rust

Zachary Bennett
Lead Software Developer

@z_bennett_ | zachbennettcodes.com

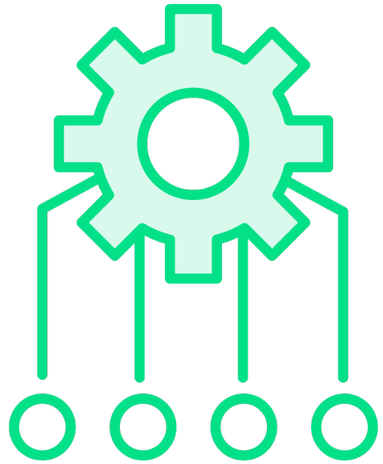
Different parts of a program running independently

Concurrency vs. Parallelism



Concurrency

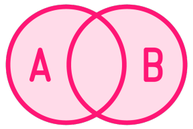
A programming environment where multiple parts of a program run independent of one another



Parallelism

A programming environment where multiple parts of a program run at the same time.

Rust's Concurrency Goals



Unify safe memory handling with preventing concurrency bugs

Catch concurrency bugs at compile time – not runtime

Make writing concurrent code more simple

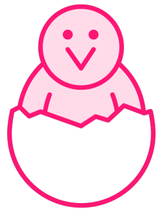
“Fearless Concurrency”

Using Threads

Thread

You can think of a thread as the smallest independently executable portion of a program.

Thread Basics



Spawning new threads

Joining thread handles

Moving values between threads

Spawning Threads

```
use std::thread;

thread::spawn(|| {
  println!(“This is a log coming from within a separate thread!”);
});

println!(“This is a log coming from within the main thread!”);
```

Waiting for Threads to Finish

```
use std::thread;

let thread_handle = thread::spawn(|| {
  for num in 1..30 {
    println!(“This is a log coming from within a separate thread!”);
    thread::sleep(Duration::from_millis(1));
  }
});


thread_handle.join().unwrap();
```

Transferring Values to Threads Using Move Closures

```
let my_str = String::from(“Hello World!”);

let thread_handle = thread::spawn(move || {
  println!(“This thread now owns this string: {}”, my_str);
});

thread_handle.join().unwrap();
```

Message Passing Using Channels

A way of sending data from one thread to another

Message Passing Between Threads Using Channels

```
use std::sync::mpsc;
use std::thread;

// MPSC: Multiple Producers, Single Consumer
let (tx, rx) = mpsc::channel();

thread::spawn(move || {
  let my_str = String::from(“Hello Other Thread!”);
  tx.send(my_str).unwrap();
});

let received_msg = rx.recv().unwrap();
println!(“Got this message from a different thread: {}”, received_msg);
```

Safe Concurrency With Channels and Move Closures

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

thread::spawn(move || {
  let my_str = String::from(“Hello Other Thread!”);
  tx.send(my_str).unwrap();

  // THIS LINE DOES NOT COMPILE! Ownership of “my_str” has moved
  println!(“We sent this message: {}”, my_str);
});

let received_msg = rx.recv().unwrap();
println!(“Got this message from a different thread: {}”, received_msg);
```

Shared-state Concurrency

Mutex

Short for “mutual exclusion”, mutexes make accessing shared data safe. Mutexes ensure that pieces of shared data are locked while being accessed by other threads. This makes your writes and reads consistent.

Shared-state Concurrency Using Mutexes

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

let my_mutex = Mutex::new(42);

{
  let mut custom_data_ref = my_mutex.lock().unwrap();
  *custom_data_ref = 12;
}

// The lock release happens automatically when the mutex goes out of scope!
```

Demo

Thread API

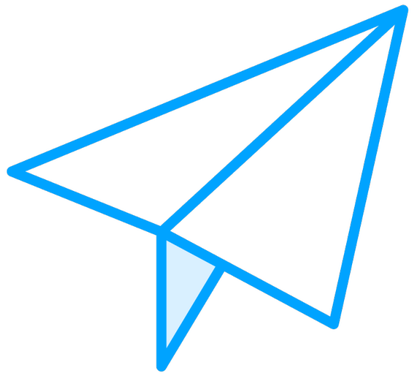
- Spawning threads
- Join handles

Channels

Shared-state Concurrency and Mutexes

Concurrency Traits

Concurrency Traits



Send

This trait allows transfer of ownership between threads

Sync

This trait marks a type has safely usable between multiple threads

You don’t have to manually implement these traits!

Demo

Send and Sync traits

- Use data in the “Send” fashion
- Use data in the “Sync” fashion