BEAM + Rust

A Match Made in Heaven

Sonny Scroggin

scrogson

BEAM + Rust

A Match Made in Heaven

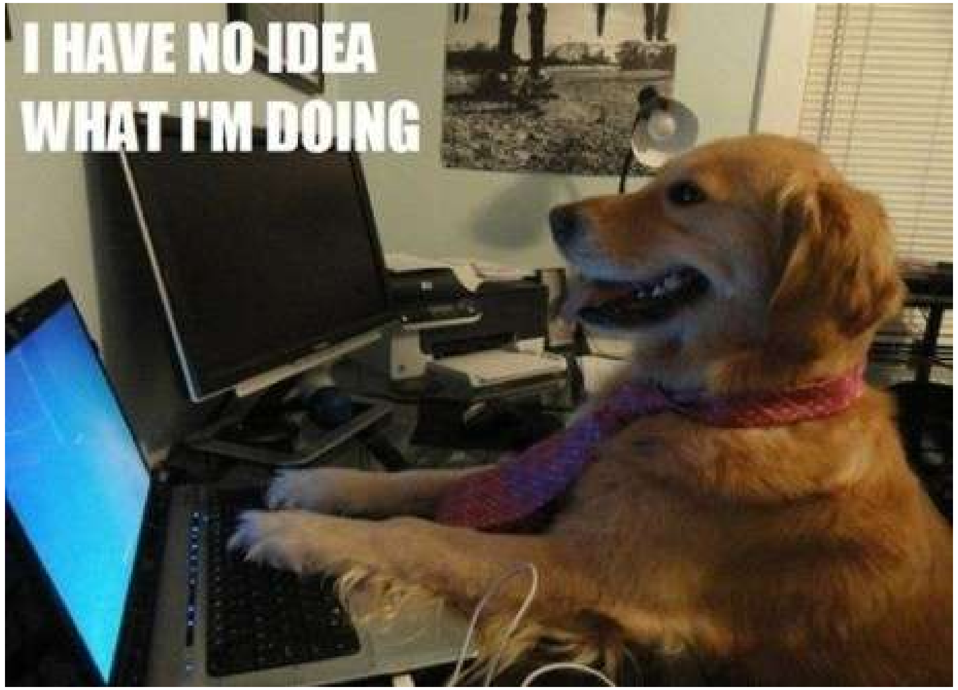
- What is Rust?
- Why Rust?
- How to Rust?
- BEAM + Rust

What is Rust?

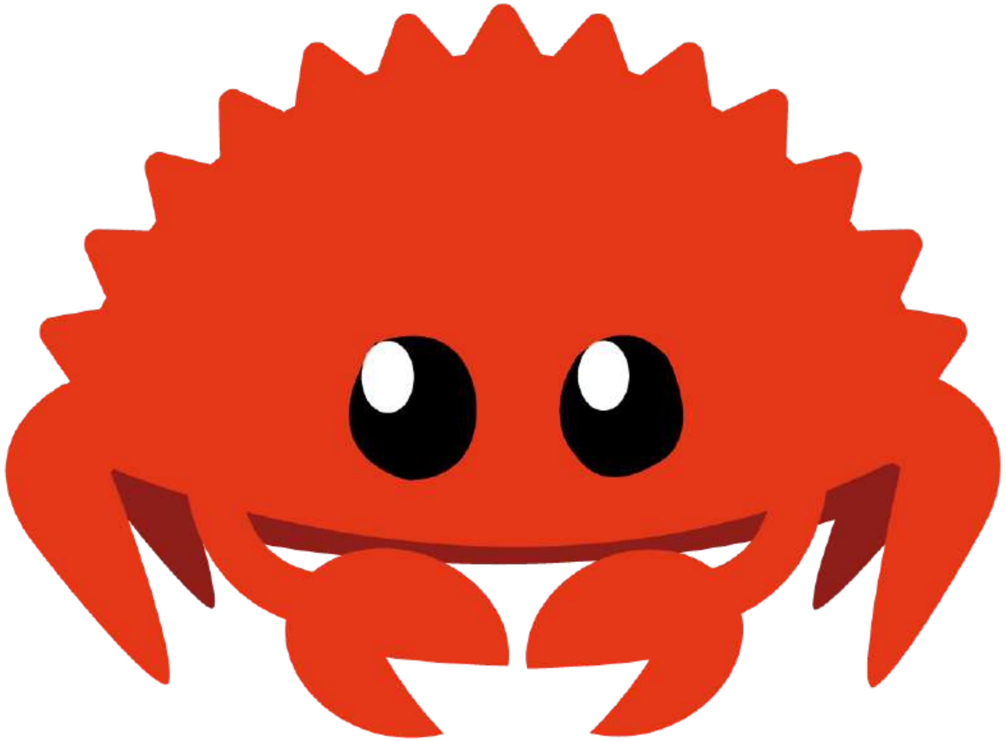
Rust is a **systems** programming language.

Empowering **everyone** to build **reliable** and **efficient** software.

When it comes to systems programming



Meh, I already know C/C++



Why Rust?

Performance

Rust is blazingly fast and memory-efficient: with **no runtime** or **garbage collector**, it can power performance-critical services, run on embedded devices, and easily **integrate with other languages**.

Safe Rust is generally fast enough without even trying.

Reliability

Rust's rich **type system** and **ownership** model guarantee **memory-safety** and **thread-safety** — and enable you to eliminate many classes of bugs at compile-time.

Productivity

Rust has great **documentation**, a **friendly compiler** with useful error messages, and **top-notch tooling**.

Rust was initially created to solve two difficult problems

- How do you make systems programming **safe**?
- How do you make **concurrency** painless?

Memory-safety

Memory-safety is a term used to describe applications that access the operating system's memory in a way that doesn't cause errors.

Common memory-safety issues

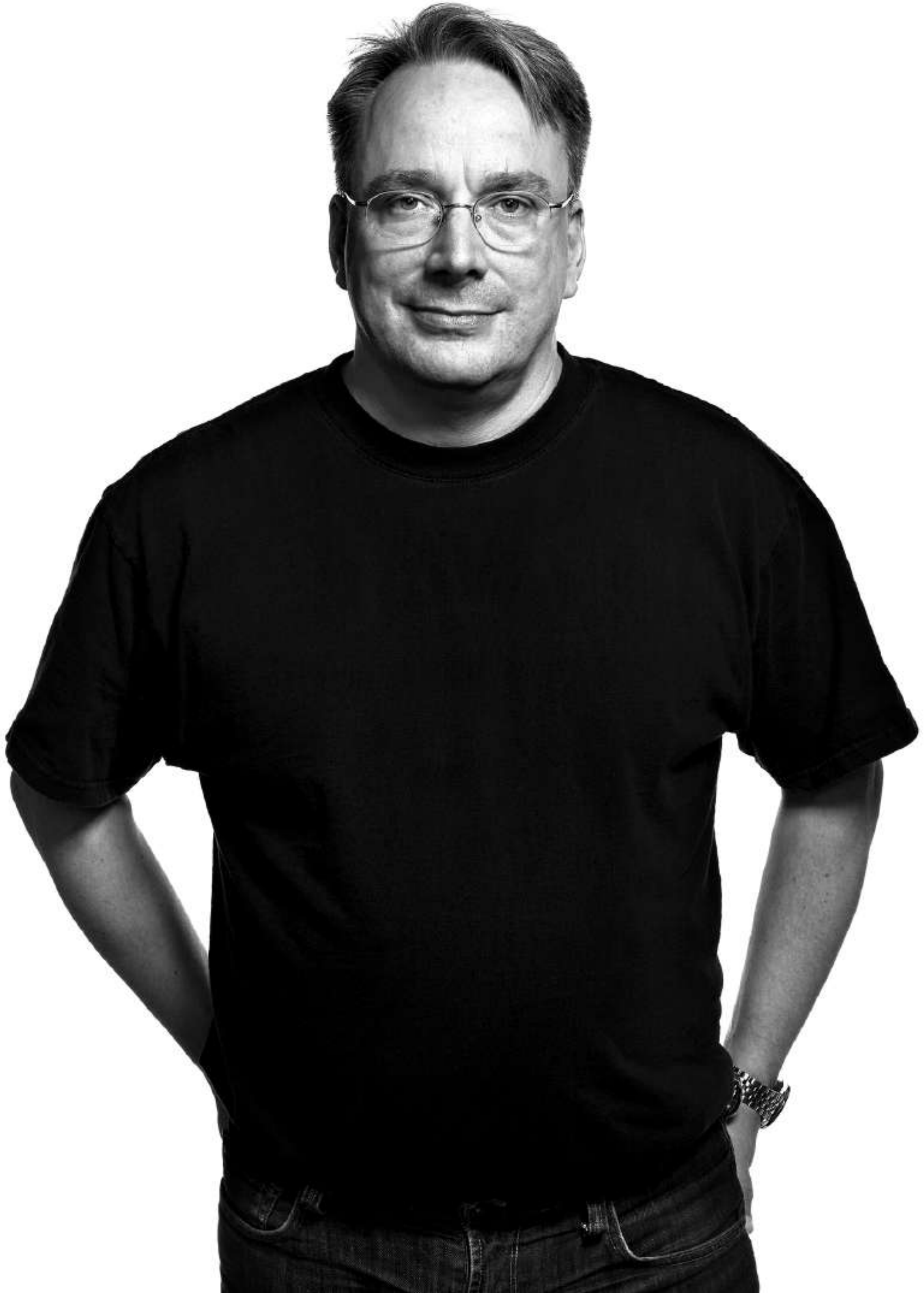
- Buffer overflows
- Race conditions
- Null pointers
- Stack/Heap exhaustion/corruption
- Use after free / double free

Memory-safety issues have been hovering at 70% for the past 12 years.

Thread-safety

Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction.

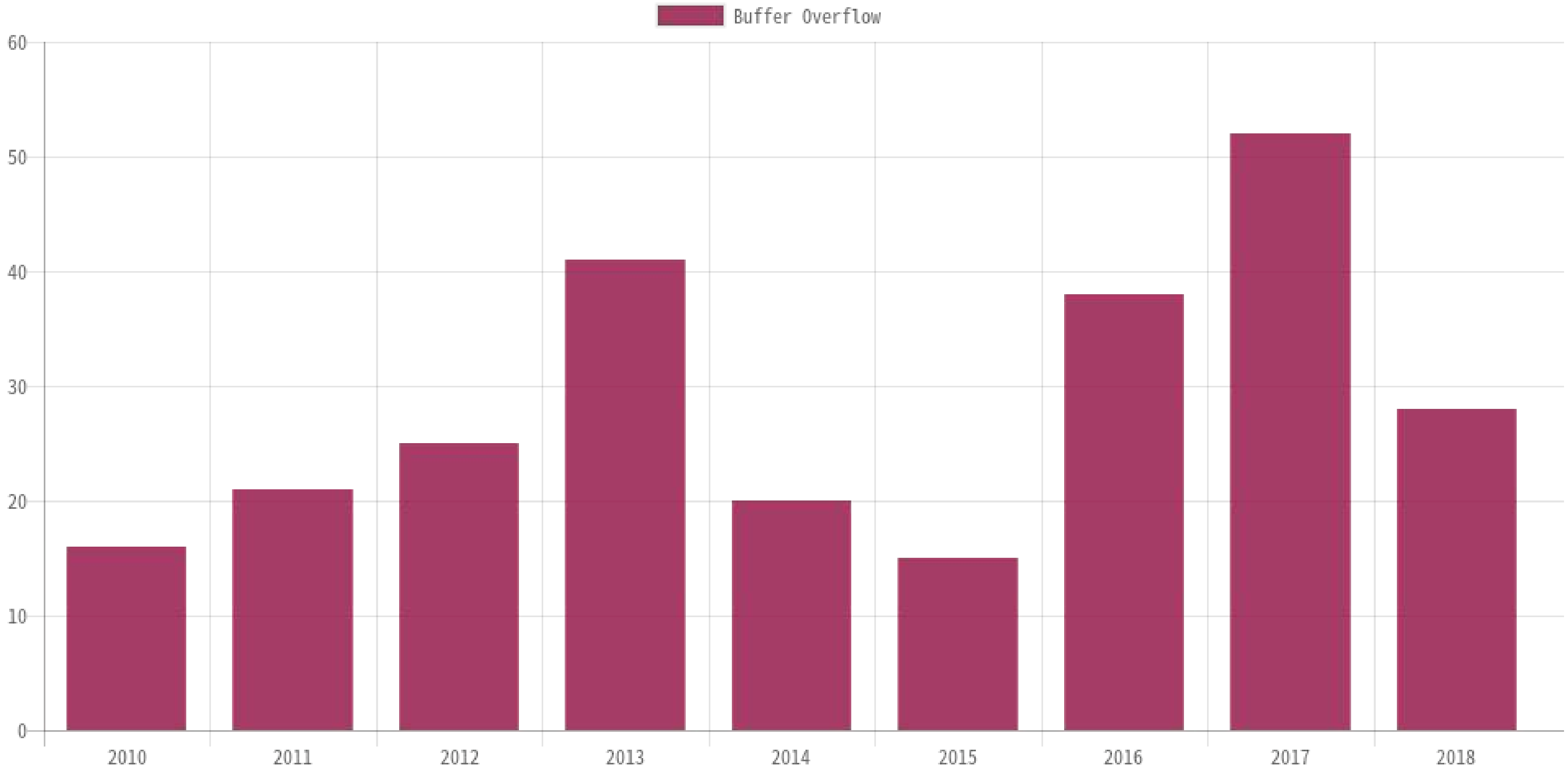
Memory and concurrency bugs often come down to code accessing data when it shouldn't.



Linus Torvalds

But I write bug-free C code, why should I bother?

Linux Kernel Vulnerabilities



Microsoft: 70 percent of all security bugs are memory safety issues

https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues

Ownership

Every value has an "owning scope"

```rust
fn new_vec() {
    let mut vec = Vec::new(); // owned by new_vec's scope
    vec.push(0);
    vec.push(1);
    // scope ends, `vec` is destroyed
}
```

```
fn new_vec() {
    let mut vec = Vec::new(); // owned by new_vec's scope
    vec.push(0);
    vec.push(1);
    // scope ends, `vec` is destroyed
}
```

Create a new vector

```
fn new_vec() {
    let mut vec = Vec::new(); // owned by new_vec's scope
    vec.push(0);
    vec.push(1);
    // scope ends, `vec` is destroyed
}
```

Push some elements

```
fn new_vec() {
    let mut vec = Vec::new(); // owned by new_vec's scope
    vec.push(0);
    vec.push(1);
    // scope ends, `vec` is destroyed
}
```

vec is now dropped

```
fn new_vec() -> Vec<i32> {
    let mut vec = Vec::new();
    vec.push(0);
    vec.push(1);
    vec
}

fn print_vec(vec: Vec<i32>) {
    for i in vec.iter() {
```

```rust
fn new_vec() -> Vec<i32> {
    let mut vec = Vec::new();
    vec.push(0);
    vec.push(1);
    vec
}

fn print_vec(vec: Vec<i32>) {
    for i in vec.iter() {
```

Create a new vec

```rust
fn new_vec() -> Vec<i32> {
    let mut vec = Vec::new();
    vec.push(0);
    vec.push(1);
    vec
}

fn print_vec(vec: Vec<i32>) {
    for i in vec.iter() {
```

Push some integers

```
fn new_vec() -> Vec<i32> {
    let mut vec = Vec::new();
    vec.push(0);
    vec.push(1);
    vec
}

fn print_vec(vec: Vec<i32>) {
    for i in vec.iter() {
```

Return the vec

```
}

fn print_vec(vec: Vec<i32>) {
    for i in vec.iter() {
        println!("{}", i)
    }
}
```

```
}

fn print_vec(vec: Vec<i32>) {
    for i in vec.iter() {
        println!("{}", i)
    }

}
```

`print_vec` takes ownership of `vec`

```rust
}

fn print_vec(vec: Vec<i32>) {
    for i in vec.iter() {
        println!("{}", i)
    }
}

fn use_vec() {
```

Interate over each entry in the vec

```
fn print_vec(vec: Vec<i32>) {
    for i in vec.iter() {
        println!("{}", i)
    }
}

fn use_vec() {
    let vec = new_vec();
```

`vec` gets deallocated here

```
    for i in vec.iter() {
        println!("{}", i)
    }
}

fn use_vec() {
    let vec = new_vec();
    print_vec(vec);
}
```

```
    for i in vec.iter() {
        println!("{}", i)
    }

}

fn use_vec() {
    let vec = new_vec();
    print_vec(vec);
}
```

Take ownership of `vec`

```
    for i in vec.iter() {
        println!("{}", i)
    }
}

fn use_vec() {
    let vec = new_vec();
    print_vec(vec);
}
```

Pass ownership to `print_vec`

```rust
fn use_vec() {
    let vec = new_vec();
    print_vec(vec);

    for i in vec.iter() {
        println!("{}", i * 2)
    }
}
```

```rust
fn use_vec() {
    let vec = new_vec();
    print_vec(vec);

    for i in vec.iter() {
        println!("{}", i * 2)
    }
}
```

Continue using `vec`

```
error: use of moved value: `vec`

for i in vec.iter() {
         ^~~
```

How can we prevent `vec` from being dropped at the end of `print_vec`?

Borrowing

&

To borrow a value, you make a reference to it

```rust
fn print_vec(vec: &Vec<i32>) {
    // `vec` is borrowed for this scope

    for i in vec.iter() {
        println!("{}", i)
    }

    // the borrow ends
}
```

```rust
fn use_vec() {
    let vec = make_vec();
    print_vec(&vec);

    for i in vec.iter() {
        println!("{}", i * 2)
    }
    // vec is destroyed here
}
```

```
fn use_vec() {
    let vec = make_vec();
    print_vec(&vec);

    for i in vec.iter() {
        println!("{}", i * 2)
    }
    // vec is destroyed here
}
```

Take ownership of `vec`

```
fn use_vec() {
    let vec = make_vec();
    print_vec(&vec);

    for i in vec.iter() {
        println!("{}", i * 2)
    }
    // vec is destroyed here
}
```

Lend `vec` to `print_vec`

```
fn use_vec() {
    let vec = make_vec();
    print_vec(&vec);

    for i in vec.iter() {
        println!("{}", i * 2)
    }
    // vec is destroyed here
}
```

Continue using `vec`

```
fn use_vec() {
    let vec = make_vec();
    print_vec(&vec);

    for i in vec.iter() {
        println!("{}", i * 2)
    }
    // vec is destroyed here
}
```

References come in two flavors

&T

Immutable references

&mut T

Mutable references

Borrowing has no runtime overhead

Rust checks these rules at compile time

```rust
fn push_all(from: &Vec<i32>, to: &mut Vec<i32>) {
    for i in from.iter() {
        to.push(*i);
    }
}
```

push_all(&vec, &mut vec)

```
error: cannot borrow `vec` as mutable because
it is also borrowed as immutable
push_all(&vec, &mut vec);
                    ^~~
```

For memory-safety, this means you can program without a garbage collector and without fear of segfaults, because Rust will catch your mistakes.

Concurrency

Channels

- Transfers ownership of the messages sent along it
- Send a pointer from one thread to another without fear of race conditions.
- Enforce thread isolation.

Locks

- A lock knows what data it protects.
- Guarantees data can only be accessed when the lock is held.
- State is never accidentally shared.
- "Lock data, not code" is enforced in Rust.

Thread-safety isn't just documentation; it's law.

- Every data type knows whether it can safely be sent between or accessed by multiple threads.
- Rust enforces this safe usage; there are no data races, even for lock-free data structures.

Message Passing

`A ! B`

Channels

```
fn channel<T>() -> (Sender<T>, Receiver<T>)
fn send(&self, t: T) -> Result<(), SendError<T>>
fn recv(&self) -> Result<T, RecvError>

impl<T: Send> Send for Sender<T> { }
```

std::sync::mpsc

This won't work

```
let mut vec = Vec::new();

sender.send(vec);

print_vec(&vec);
```

```
error[E0382]: borrow of moved value: `vec`
  --> src/main.rs:13:15
   |
8  |     let mut vec = Vec::new();
   |         ------- move occurs because `vec` has type `std::vec
...
11 |     tx.send(vec);
   |             --- value moved here
12 |
13 |     print_vec(&vec);
   |               ^^^^ value borrowed here after move
```

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

Create a channel and deconstuct the sender and receiver

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

Spawn a new thread

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

Run some computation

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

Send the result to the receiver

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

Receive the result

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

Handle the success case

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

Handle the error case

spawn, send, recv

```
let (sender, receiver) = std::sync::mpsc::channel();

std::thread::spawn(move || {
    let result = expensive_computation();
    sender.send(result).unwrap();
});

match receiver.recv() {
  Ok(msg) => println!("{:?}", msg),
  Err(err) => println!("{:?}", err)
}
```

Equivalent in Elixir

```elixir
parent = self()

spawn(fn ->
  result = expensive_computation()
  send(parent, result)
end)

receive do
  msg -> IO.inspect(msg)
end
```

For concurrency, this means you can choose from a wide variety of paradigms (message passing, shared state, lock-free, purely functional), and Rust will help you avoid common pitfalls.

BEAM + Rust

- Ports
- NIFs
- Nodes

Ports

Communicate with external programs over stdin/stdout

Sending data to a port

```
port = Port.open({:spawn_executable, "priv/binary"}, [:binary])

data = :erlang.term_to_binary(term)
len = byte_size(data)
iodata = [<<len::big-unsigned-integer-size(64)>>, data]

Port.command(port, iodata)
```

Sending data to a port

```
port = Port.open({:spawn_executable, "priv/binary"}, [:binary])

data = :erlang.term_to_binary(term)
len = byte_size(data)
iodata = [<<len::big-unsigned-integer-size(64)>>, data]

Port.command(port, iodata)
```

Open a port to an external binary

Sending data to a port

```
port = Port.open({:spawn_executable, "priv/binary"}, [:binary])

data = :erlang.term_to_binary(term)
len = byte_size(data)
iodata = [<<len::big-unsigned-integer-size(64)>>, data]

Port.command(port, iodata)
```

Serialize our data to binary

Sending data to a port

```
port = Port.open({:spawn_executable, "priv/binary"}, [:binary])

data = :erlang.term_to_binary(term)
len = byte_size(data)
iodata = [<<len::big-unsigned-integer-size(64)>>, data]

Port.command(port, iodata)
```

Get the length of the encoded data

Sending data to a port

```
port = Port.open({:spawn_executable, "priv/binary"}, [:binary])

data = :erlang.term_to_binary(term)
len = byte_size(data)
iodata = [<<len::big-unsigned-integer-size(64)>>, data]

Port.command(port, iodata)
```

Pack our length and data into an IO list

Sending data to a port

```
port = Port.open({:spawn_executable, "priv/binary"}, [:binary])

data = :erlang.term_to_binary(term)
len = byte_size(data)
iodata = [<<len::big-unsigned-integer-size(64)>>, data]

Port.command(port, iodata)
```

Send it to the port

Erlang Terms in Rust

https://crates.io/crates/eetf

A Rust implementation of Erlang External Term Format

```rust
pub enum Term {
    Atom(Atom),
    FixInteger(FixInteger),
    BigInteger(BigInteger),
    Float(Float),
    Pid(Pid),
    Port(Port),
    Reference(Reference),
    Binary(Binary),
    BitBinary(BitBinary),
    List(List),
    ImproperList(ImproperList),
    Tuple(Tuple),
    Map(Map),
}
```

```
use eetf::Term;

let (tx, rx) = std::sync::mpsc::channel::<Term>();

std::thread::spawn(move || {
    let stdin = io::stdin();
    let mut locked = stdin.lock();
    let mut term_buffer = Vec::new();

    loop {
        // ...
    }
});
```

```rust
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

```rust
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

Start an infinite loop

```
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

Create an 8-byte buffer

```
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

Read 8-bytes from stdin

```
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

Find the length of the data

```
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

Resize the term buffer to accomodate the size of the data

```rust
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

Read the data into the term buffer

```
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

Decode the binary data into a Term

```
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

Send the Term to the reciever thread

```rust
loop {
    let mut buffer = [0; 8];
    if let Err(_) = locked.read_exact(&mut buffer) {
        break;
    };

    let length = u64::from_be_bytes(buffer);
    term_buffer.resize(length as usize, 0);
    if let Err(_) = locked.read_exact(&mut term_buffer) {
        break;
    };

    let term = Term::decode(Cursor::new(&term_buffer)).unwrap();
    tx.send(term).unwrap();
}
```

```rust
while let Ok(term) = rx.recv() {
    match term {
        Term::FixInteger(FixInteger { value }) => // ...,
        Term::Atom(Atom { name }) => match name.as_ref() {
            "quit" => std::process::exit(0),
            text => println!("{}", text),
        },
        other => println!("Received {:?}", other),
    }
}
```

NIFs

Native Implemented Functions

Rustler

https://github.com/rusterlium/rustler

Example

Image processing

https://github.com/scrogson/mirage

Mirage

```
defmodule Mirage do
  use Rustler, otp_app: :mirage

  defstruct bytes: nil,
            byte_size: nil,
            extension: nil,
            height: nil,
            width: nil,
            resource: nil

  def from_bytes(_path), do: :erlang.nif_error(:nif_not_loaded)
  def resize(_resource, _width, _height), do: :erlang.nif_error(:
end
```

NIF Boilerplate

```
use rustler::schedule::SchedulerFlags::*;

mod atoms;
mod mirage;

rustler::rustler_export_nifs! {
    "Elixir.Mirage",
    [
        ("from_bytes", 1, mirage::from_bytes, DirtyIo),
        ("resize", 3, mirage::resize, DirtyCpu),
    ],
    Some(load)
}

fn load(env: rustler::Env, _info: rustler::Term) -> bool {
    mirage::load(env)
```

NIF Boilerplate

```
mod atoms;
mod mirage;

rustler::rustler_export_nifs! {
    "Elixir.Mirage",
    [
        ("from_bytes", 1, mirage::from_bytes, DirtyIo),
        ("resize", 3, mirage::resize, DirtyCpu),
    ],
    Some(load)
}

fn load(env: rustler::Env, _info: rustler::Term) -> bool {
    mirage::load(env)
}
```

Setup the mapping from BEAM MFAs to NIF functions

NIF Boilerplate

```
use rustler::schedule::SchedulerFlags::*;

mod atoms;
mod mirage;

rustler::rustler_export_nifs! {
    "Elixir.Mirage",
    [
        ("from_bytes", 1, mirage::from_bytes, DirtyIo),
        ("resize", 3, mirage::resize, DirtyCpu),
    ],
    Some(load)
}

fn load(env: rustler::Env, _info: rustler::Term) -> bool {
```

Module name

NIF Boilerplate

```rust
use rustler::schedule::SchedulerFlags::*;

mod atoms;
mod mirage;

rustler::rustler_export_nifs! {
    "Elixir.Mirage",
    [
        ("from_bytes", 1, mirage::from_bytes, DirtyIo),
        ("resize", 3, mirage::resize, DirtyCpu),
    ],
    Some(load)
}

fn load(env: rustler::Env, _info: rustler::Term) -> bool {
    mirage::load(env)
}
```

NIF Boilerplate

```rust
mod atoms;
mod mirage;

rustler::rustler_export_nifs! {
    "Elixir.Mirage",
    [
        ("from_bytes", 1, mirage::from_bytes, DirtyIo),
        ("resize", 3, mirage::resize, DirtyCpu),
    ],
    Some(load)
}

fn load(env: rustler::Env, _info: rustler::Term) -> bool {
    mirage::load(env)
}
```

NIF Boilerplate

```
mod atoms;
mod mirage;

rustler::rustler_export_nifs! {
    "Elixir.Mirage",
    [
        ("from_bytes", 1, mirage::from_bytes, DirtyIo),
        ("resize", 3, mirage::resize, DirtyCpu),
    ],
    Some(load)
}

fn load(env: rustler::Env, _info: rustler::Term) -> bool {
    mirage::load(env)
}
```

Function to call when the NIF loads

NIF Boilerplate

```
mod atoms;
mod mirage;

rustler::rustler_export_nifs! {
    "Elixir.Mirage",
    [
        ("from_bytes", 1, mirage::from_bytes, DirtyIo),
        ("resize", 3, mirage::resize, DirtyCpu),
    ],
    Some(load)
}

fn load(env: rustler::Env, _info: rustler::Term) -> bool {
    mirage::load(env)
}
```

NIF Boilerplate

```
mod atoms;
mod mirage;

rustler::rustler_export_nifs! {
    "Elixir.Mirage",
    [
        ("from_bytes", 1, mirage::from_bytes, DirtyIo),
        ("resize", 3, mirage::resize, DirtyCpu),
    ],
    Some(load)
}

fn load(env: rustler::Env, _info: rustler::Term) -> bool {
    mirage::load(env)
}
```

Setup resource objects

Resource Objects

```
struct Image {
    image: DynamicImage,
    format: ImageFormat,
}

pub fn load(env: Env) -> bool {
    rustler::resource_struct_init!(Image, env);
    true
}
```

A way to pass pointers back and forth

Elixir Struct Encoding/Decoding

```
defstruct byte_size: nil,
          extension: nil,
          height: nil,
          width: nil,
          resource: nil
```

%Mirage{}

Elixir Struct Encoding/Decoding

```
#[derive(NifStruct)]
#[module = "Mirage"]
struct Mirage {
    byte_size: usize,
    extension: Atom,
    height: u32,
    width: u32,
    resource: ResourceArc<Image>,
}
```

```rust
pub fn from_bytes<'a>(
    env: Env<'a>,
    args: &[Term<'a>]
) -> Result<Term<'a>, Error> {
    let bytes: Binary = args[0].decode()?;

    match image::load_from_memory(bytes.as_slice()) {
        // ...
    }
}
```

```rust
pub fn from_bytes<'a>(
    env: Env<'a>,
    args: &[Term<'a>]
) -> Result<Term<'a>, Error> {
    let bytes: Binary = args[0].decode()?;

    match image::load_from_memory(bytes.as_slice()) {
        // ...
    }
}
```

`<'a>` = lifetime annotations

```rust
pub fn from_bytes<'a>(
    env: Env<'a>,
    args: &[Term<'a>]
) -> Result<Term<'a>, Error> {
    let bytes: Binary = args[0].decode()?;

    match image::load_from_memory(bytes.as_slice()) {
        // ...
    }
}
```

Decode the first argument into a Binary

```
pub fn from_bytes<'a>(
    env: Env<'a>,
    args: &[Term<'a>]
) -> Result<Term<'a>, Error> {
    let bytes: Binary = args[0].decode()?;

    match image::load_from_memory(bytes.as_slice()) {
        // ...
    }
}
```

Load the bytes into an image::Image

```rust
if let Ok(format) = image::guess_format(&bytes.as_slice()) {
    let mirage = Mirage {
        byte_size: bytes.len(),
        extension: extension(format),
        width: image.width(),
        height: image.height(),
        resource: ResourceArc::new(Image { image, format }),
    };

    return Ok((ok(), mirage).encode(env));
}
return Err(rustler::Error::Atom("unsupported_image_format"));
```

```rust
if let Ok(format) = image::guess_format(&bytes.as_slice()) {
    let mirage = Mirage {
        byte_size: bytes.len(),
        extension: extension(format),
        width: image.width(),
        height: image.height(),
        resource: ResourceArc::new(Image { image, format }),
    };

    return Ok((ok(), mirage).encode(env));
}
return Err(rustler::Error::Atom("unsupported_image_format"));
```

If we have a valid image format

```
if let Ok(format) = image::guess_format(&bytes.as_slice()) {
    let mirage = Mirage {
        byte_size: bytes.len(),
        extension: extension(format),
        width: image.width(),
        height: image.height(),
        resource: ResourceArc::new(Image { image, format }),
    };

    return Ok((ok(), mirage).encode(env));
}
return Err(rustler::Error::Atom("unsupported_image_format"));
```

Build up our Mirage struct

```
if let Ok(format) = image::guess_format(&bytes.as_slice()) {
    let mirage = Mirage {
        byte_size: bytes.len(),
        extension: extension(format),
        width: image.width(),
        height: image.height(),
        resource: ResourceArc::new(Image { image, format }),
    };

    return Ok((ok(), mirage).encode(env));
}
return Err(rustler::Error::Atom("unsupported_image_format"));
```

Return `{:ok, %Mirage{}}`

```rust
if let Ok(format) = image::guess_format(&bytes.as_slice()) {
    let mirage = Mirage {
        byte_size: bytes.len(),
        extension: extension(format),
        width: image.width(),
        height: image.height(),
        resource: ResourceArc::new(Image { image, format }),
    };

    return Ok((ok(), mirage).encode(env));
}
return Err(rustler::Error::Atom("unsupported_image_format"));
```

Return `{:error, :unsupported_image_format}`

Nodes

Communicate with applications using the Erlang Distribution Protocol

erl_dist

https://crates.io/crates/erl_dist

Rust Implementation of Erlang Distribution Protocol

ei

https://crates.io/crates/ei

Rust Implementation of erl_interface

Conclusion