Embedded Linux Conference 2017

Jonathan Creekmore <jonathan@thecreekmores.org>

Star Lab Corp.

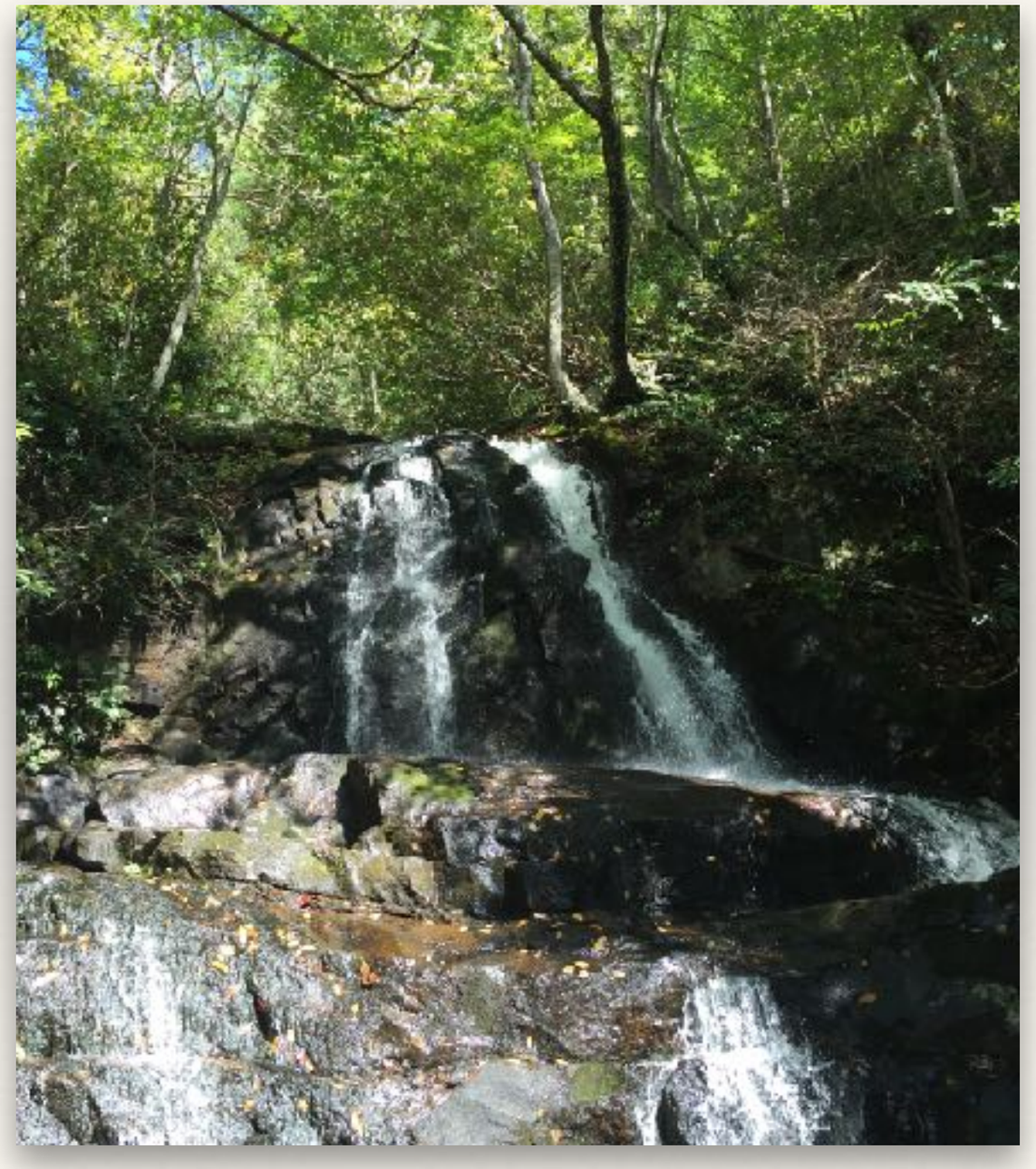
@jcreekmore

Rust

Removing the Sharp Edges from Systems Programming

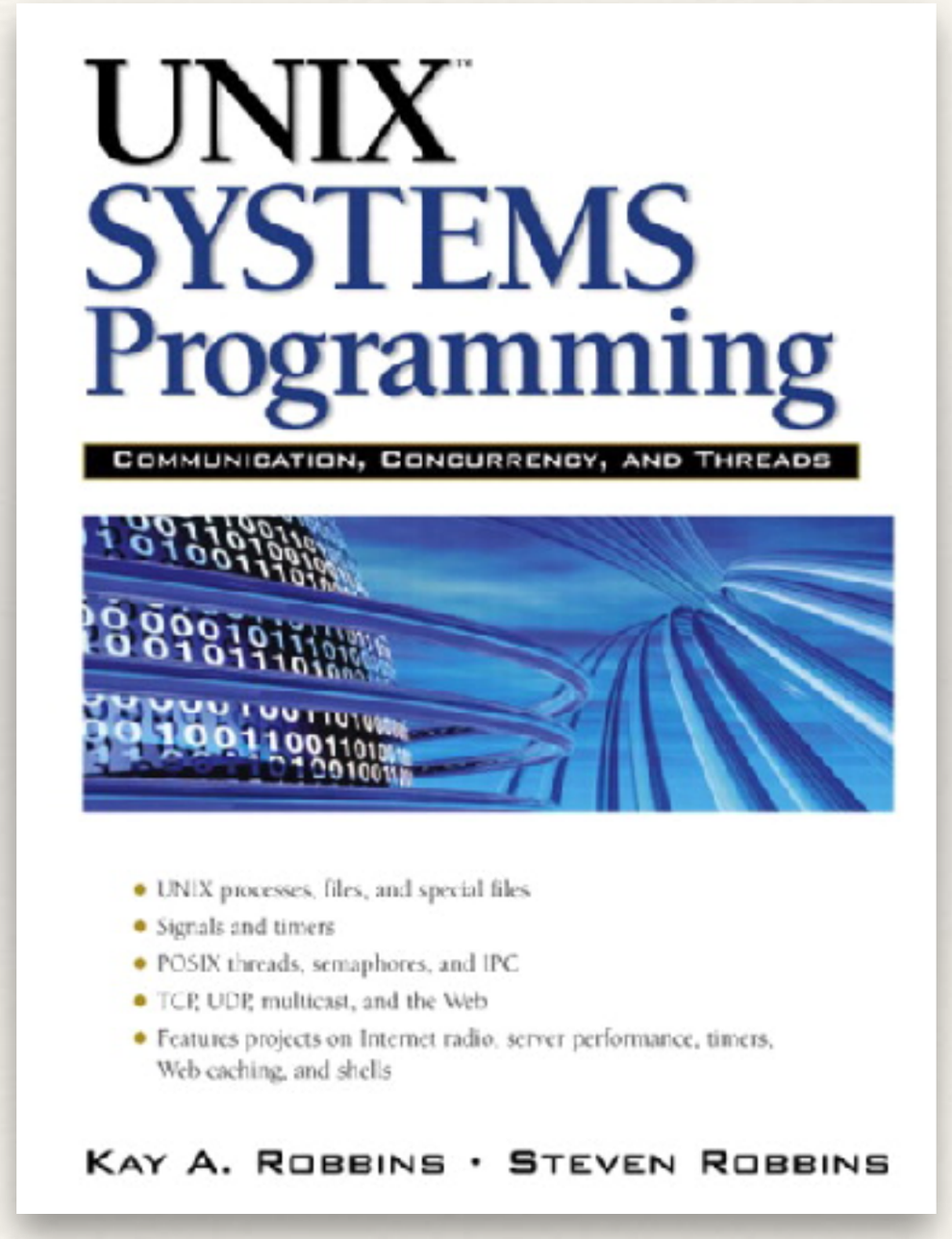
Who is this guy?

- Systems programmer for over 15 years
- Specialities:
  - Kernel & Driver development
  - Software security
- Activities that bring me joy:
  - Programming Language Theory
  - Mathematics
  - History
  - Hiking
  - Camping



What is Systems Programming?

- Provides services to other software
  - e.g. kernels, libraries, daemons
- Typically resource constrained
  - memory or cpu constrained
  - special access to hardware
- Used to build abstractions for application programmers

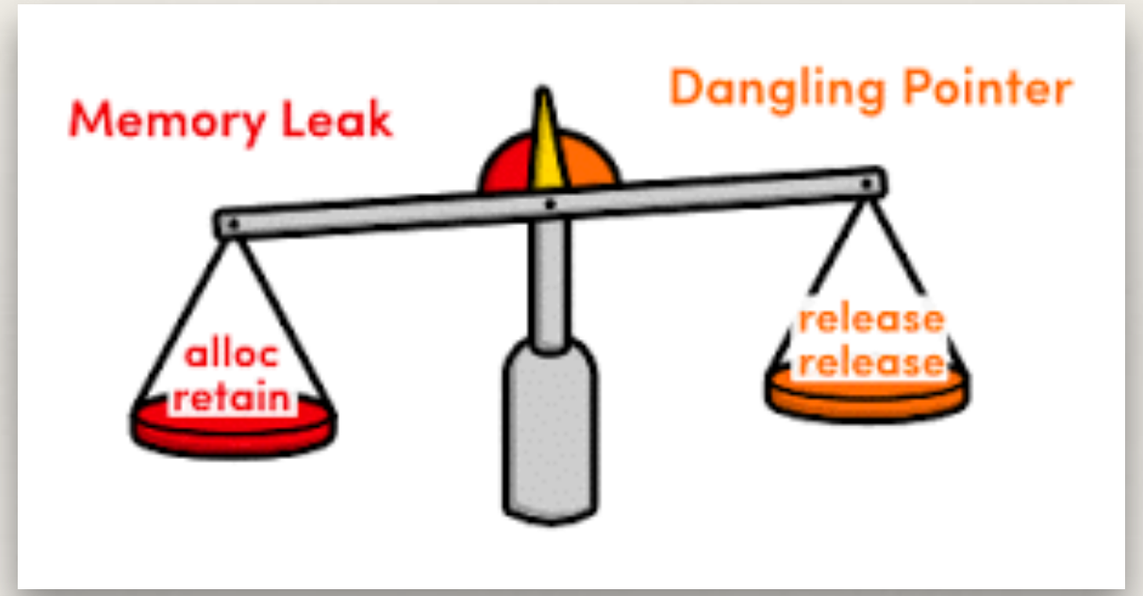


Sharp Edges: Manual Memory Management

- "Modern" apps programming languages (C#, Java, JavaScript, etc.) usually provide a garbage collector
- Garbage collection introduces time/space inefficiencies
  - Little to no runtime overhead desired
- Shared by many types of environments (no common GC available)
- Used to build programs that may need specific control over memory layout and allocation

Sharp Edges: Manual Memory Management

- Programmer managed:
  - Scope of allocated memory
  - Data races by writes to non-exclusive access to memory

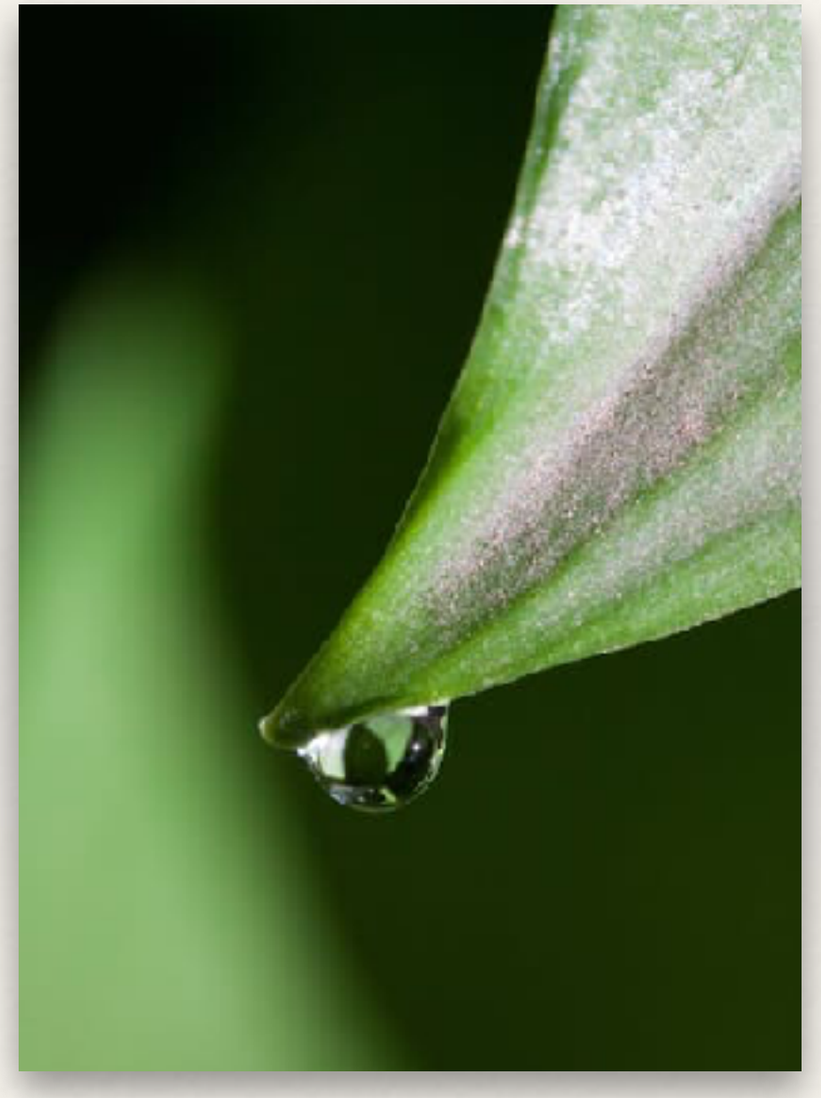


Speedy tour through Rust

A Helping Hand

- Strong, static typing
- Encodes ownership and lifetimes into the type system
- Data is immutable by default
- Ownership is transferred by default
  - Programmer can choose to borrow data rather than transfer ownership

Memory leak issues in C



```c
char *upcase(const char *in) {
    size_t len = strlen(in);
    char *out = (char *)malloc(len + 1);
    if (!out)
        return NULL;
    for (size_t i = 0; i < len; i++) {
        out[i] = toupper(in[[i]);
    }
}

void test() {
    char *test = strdup("Hello world");
    test = upcase(test);
}
```

Leak issue in Rust: Ownership transfer or borrow

```
// ownership transfer
fn upcase(input: String) -> String {
    let mut out = String::new();

    for c in input {
        out.push(toupper(c));
    }

    out
}

// borrowing
fn upcase2(input: &String) -> String {
    let mut out = String::new();

    for c in input {
        out.push(toupper(c));
    }

    out
}
```

Lifetime issue in C++

```
void test() {
    // Create a new BigObject
    BigObject *foo = new BigObject;

    // Get a reference to the object stored in
    // BigObject
    Object &bar = &foo->bar;

    // Some function consumes foo
    consume(foo);
    foo = NULL;

    // Use the bar reference we acquired earlier
    bar.doit();
}
```

Lifetime issue in Rust: Compile Time Error

```
fn consume(_: BigObject) {

}

fn test() {
    let foo = BigObject::new();
    let bar = &foo.bar;
    consume(foo);
    bar.doit();
}
```

```
error: cannot move out of `foo` because it is borrowed [--explain E0505]
  --> <anon>:26:13
   |>
25 |>     let bar = &foo.bar;
   |>               ------- borrow of `foo.bar` occurs here
26 |>     consume(foo);
   |>             ^^^ move out of `foo` occurs here

error: aborting due to previous error
```

Data Race in C++

```
void test(std::deque<int> &in) {
    for (std::deque<int>::iterator it = in.begin(); it != in.end(); ++it) {
        if (*it % 2 == 0) {
            // If erasure happens anywhere* in the deque,
            // all iterators, pointers and references
            // related to the container are invalidated.
            in.erase(it);
        }
}
```

Data Race in Rust: Compile Time Error

```
fn test(input: &mut Vec<usize>) {
    for (i, x) in input.iter().enumerate() {
        if x % 2 == 0 {
            input.remove(i);
        }
    }
}
```

```
error: cannot borrow `*input` as mutable because it is also borrowed as immutable [--explain E0502]
 --> <anon>:4:13
  |>
2 |>     for (i, x) in input.iter().enumerate() {
  |>                   ----- immutable borrow occurs here
3 |>         if x % 2 == 0 {
4 |>             input.remove(i);
  |>             ^^^^^ mutable borrow occurs here
5 |>         }
6 |>     }
  |>     - immutable borrow ends here
```

Exciting Features

Algebraic Data Types

Sum Types

```
enum Sum {
    Foo,
    Bar(usize, String),
    Baz { x: usize,
          y: String },
}
```

Product Types

```
type ProductTuple =
    (usize, String);

struct ProductStruct {
    x: usize,
    y: String,
}
```

Pattern Matching

```rust
pub enum Sum {
    Foo,
    Bar(usize, String),
    Baz { x: usize, y: String },
}

fn test() {
    let foo = Sum::Baz { x: 42, y: "foo".into() };

    let value = match foo {
        Sum::Foo => 0,
        Sum::Bar(x, _) => x,
        Sum::Baz { x, .. } => x,
    };
}
```

Traits and Generics

```
trait Truthiness {
    fn is_truthy(&self) -> bool;
}

impl Truthiness for usize {
    fn is_truthy(&self) -> bool {
        match *self {
            0 => false,
            _ => true,
        }
    }
}

impl Truthiness for String {
    fn is_truthy(&self) -> bool {
        match self.as_ref() {
            "" => false,
            _ => true,
        }
    }
}
```

```
fn print_truthy<T>(value: T)
    where T: Debug + Truthiness
{
    println!("Is {:?} truthy? {}",
                    &value,
                    value.is_truthy());
}

fn main() {
    print_truthy(0);
    print_truthy(42);

    let empty = String::from("");
    let greet =
        String::from("Hello!");
    print_truthy(empty);
    print_truthy(greet);
}
```

Traditional Error Handling

- “I call it my billion dollar mistake. It was the invention of the null reference in 1965” — Tony Hoare
- Dangerous because nothing is explicitly required to check for NULL (in C/C++).
  - Best practices and some static checkers look for it.
  - Failure to check causes `SEGFAULT` in best case, undefined behavior in worst case
- Common practice in C/C++ to overload return type with errors

Option type

- `Option<T>` is a sum type providing two constructors:
  - `Some<T>`
  - `None`

- Type system forces you to handle the error case
- Chaining methods allow code to execute only in success case:
  - `Some(42).map(|x| x + 8) => Some(50)`
  - `Some(42).and_then(|x| Some(x + 8)) => Some(50)`
  - `None.map(|x| x + 8) => None`

Result type

- Result<T, E> is a sum type providing two constructors:
  - Ok<T>
  - Err<E>
- Type system again forces handling of error cases
- Same chaining methods available as Option<T>
  - Provides a `Result<T, E>::map_err(U) -> Result<T, U>` method
- Both Option<T> and Result<T, E> provide ways to convert between each other

Other features in brief

- Unsafe code
  - Break safety features in a delimited scope
- Foreign function interface
  - Call out to C code and wrap existing libraries
- Hygienic macros
  - Brings safety to generated code

Building Applications

Cargo

- Build tool and dependency manager for Rust
  - Builds packages called “crates”
  - Downloads and manages the dependency graph
    - Test integration!
    - Doc tests!
- Ties into crates.io, the community crate host
- See the Cargo documentation for a good Getting Started guide (http://doc.crates.io/index.html)

meta-rust

- Yocto layer for building Rust binaries
  - https://github.com/meta-rust/meta-rust
- Support for:

Yocto Release	Legacy version	Default version
krogoth	Rust 1.10	Rust 1.12.1
morty	Rust 1.12.1	Rust 1.14
pyro	Rust 1.14	Rust 1.16/17

cargo bitbake

- Tool for auto-generating a BitBake file from a Cargo.toml file
  - https://github.com/cardoe/cargo-bitbake
- Target BitBake file uses the `meta-rust` crate fetcher to download dependencies
- Cargo is then used to build the target inside the Yocto build process

Rough Edges

Rough Edges

- Fighting the borrow checker
  - Takes a while to wrap your head around ownership
  - Eventually, it does click
- Stable vs. unstable features
  - Useful APIs and syntax are unstable-only
- Many useful libraries are immature
  - async-I/O is a big one
- Cargo locks you in to its build methodology (partially mitigated by cargo bitbake!)

Rust is a young language

- Stable 1.0 version only hit in May 2015
  - 6-week release schedule brings us to 1.15 as of February 2017
- More APIs continue to stabilize over time
  - Compiler changes take longer
- Active community writing libraries
  - Libraries tend to be in flux, though.

Want to give Rust a try?

https://www.rustup.rs