第 6 讲：The Programming Languages of OS

第五节：Writing kernel in Rust?

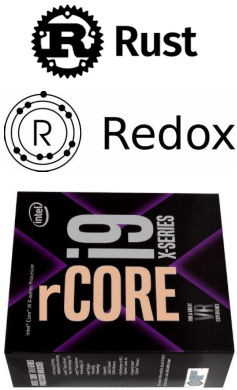
陈渝

清华大学计算机系

yuchen@tsinghua.edu.cn

2020 年 3 月 22 日

Introduction – OS in Rust

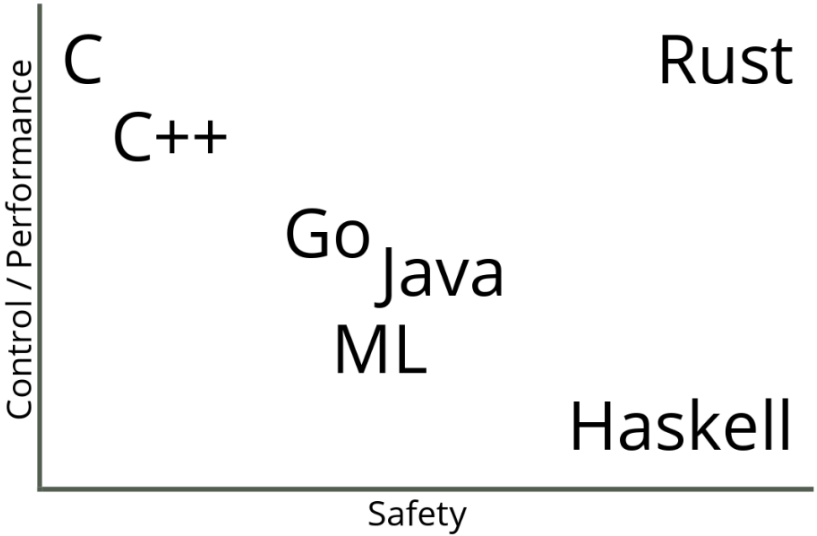


OS in Rust?

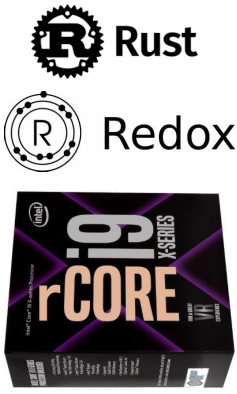
Rust

Redox

Firecracker



Introduction – OS in Rust – Go vs Rust



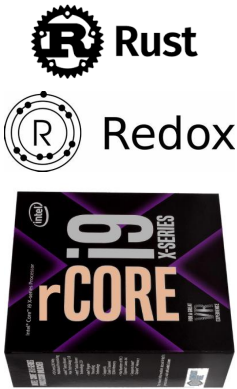
	Go	Rust
Birth	Announced Nov 2009, 1.0 March 2012	Announced 2010, 1.0 May 2015
Popularity	TIOBE # 17 @ 0.996%, SO Most Loved #5	TIOBE #36 @ 0.267%. SO Most Loved #1 ~3yrs
Sponsor	Google	Mozilla
Type System	Strong, Static, Inferred	Strong, Static, Inferred
Syntax Similarity	C	C, C++, ML
Features	Imperative, sort of OO, Procedural, Reflective, Event Driven, Concurrent	Imperative, OO, Functional, Procedural, Generic, Concurrent.
Perf Benchmarks	2-20x slower than C	2-20x faster than Go

Introduction – OS in Rust

Go and Rust Similarities

- Strongly typed.
- Prefer composition over inheritance.
- Errors are values.
- Lightweight, performant, cross platform, systems programming.
- Great tooling: IDE support, formatter, LSP.
- Integrated testing and documentation.

Introduction – OS in Rust



Rust > Go

- Functional features and higher abstractions with no run time cost.
- FFI to C code.
- Errors are values.
- Generally more performant.
- Compile time memory and thread safety guarantees.
- Package management support via Cargo.

Introduction – OS in Rust

Rust

Redox

Go > Rust

- Simple, clear syntax and language features.
- Fast compile.
- Easy cross compilation.
- Batteries included std lib.
- Learning curve and productivity.

Firecracker

Introduction – OS in Rust

OS in Rust?

- Rust is a systems software programming language designed around safety, parallelism, and speed
- Rust has a novel system of ownership, whereby it can statically determine when a memory object is no longer in use
- This allows for the power of a garbage-collected language, but with the performance of manual memory management
- This is important because —unlike C —Rust is highly composable, allowing for more sophisticated (and higher performing!) primitives

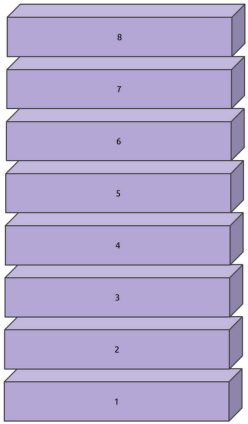
Rust

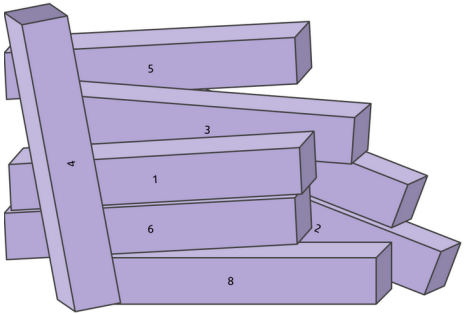
Redox

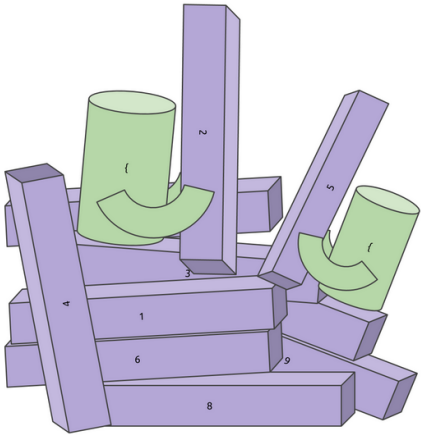
Firecracker

Rust's central feature is ownership.

- The Stack and the Heap

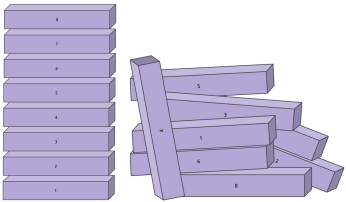






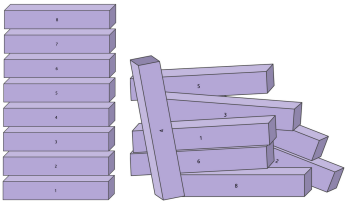
Rust's central feature is ownership. Keep these rules in mind :

- Each value in Rust has a variable that' s called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.



Rust's central feature is ownership.

```
1 fn main() {
2     let hello = "Hello, World!";
3     println!("{}", hello);
4 } // variable `hello` is now invalid
```

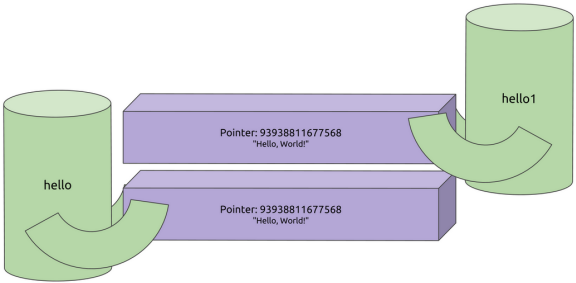


Rust's central feature is ownership.

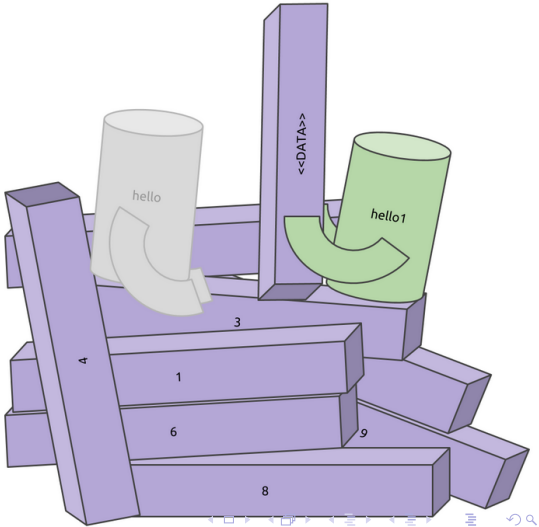
Copy vs Move

```rust
fn main() {
    let hello = "Hello, World!"; // string literal
    let hello1 = hello; // copy the value of hello and bind it to hello1
    println!("{}", hello); // this works!

    let hello = String::from("Hello, World!"); // String type
    let hello1 = hello; // move the data of hello into hello1
    println!("{}", hello); // error[E0382]: use of moved value: `hello`
}
```

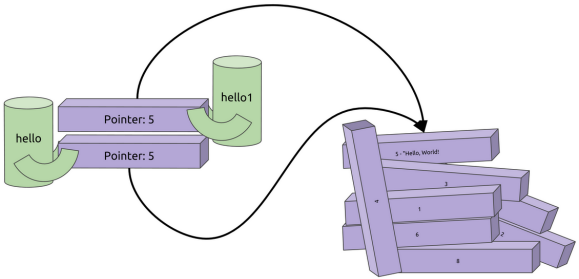
Copy trait when using &str



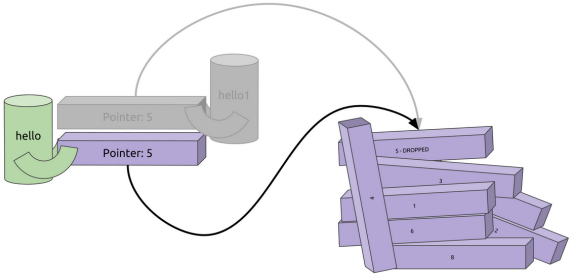
Move trait when using the heap



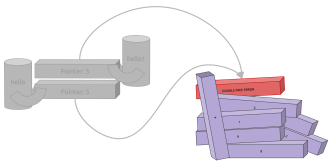
A rough sketch of copying String type data



Dropping hello1



Double Free Error



Deep copy

```rust
fn main() {
    let hello = String::from("Hello, World!"); // String type
    let hello1 = hello.clone(); // clone data from hello into hello1
    println!("{}", hello); // #⇒ "Hello, World!"
}
```

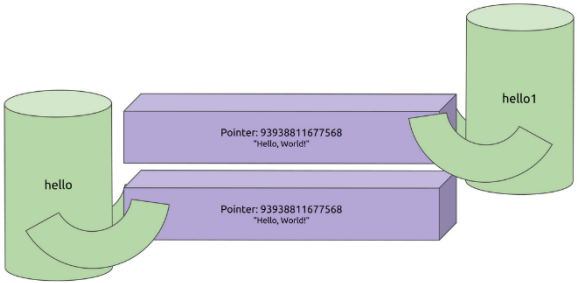
Ownership and Functions

```rust
fn main() {
    let string = "Hello, World!";
    println!("{:p}", string); // 0x5652d704aa80
    foo(string);
}

fn foo(string: &str) {
    println!("{:p}", string); // 0x5652d704aa80
}
```

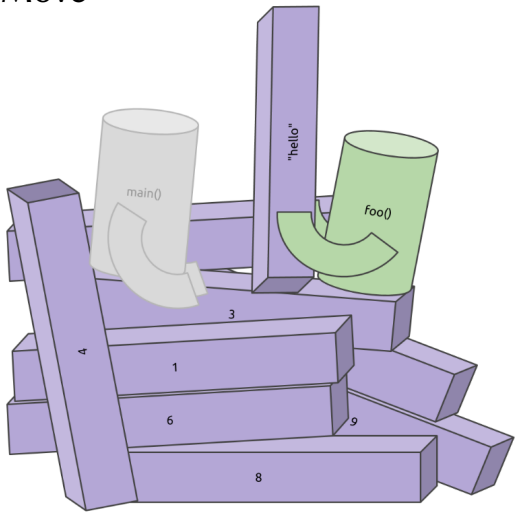
```rust
fn main() {
  let string = String::from("hello");
  println!("{:p}", string.as_ptr()); // 0x7efced01c010
  foo(string);
}

fn foo(string: String){
  println!("{:p}", string.as_ptr()); // 0x7efced01c010
}
```

Copy versus Move

Copy


hello
hello1
Pointer: 93938811677568
"Hello, World!"
Pointer: 93938811677568
"Hello, World!"

Move


main()
"hello"
foo()
3
4
1
6
9
8

Introduction – OS in Rust – Why Rust?

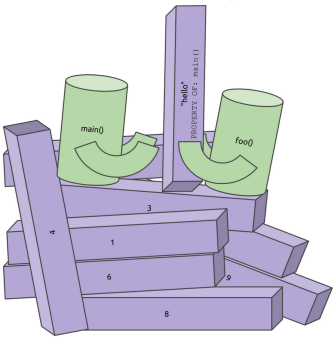
```rust
fn main() {
  let string = String::from("hello");
  println!("{:p}", string.as_ptr()); // 0x7efced01c010
  foo(string.clone());
}

fn foo(string: String){
  println!("{:p}", string.as_ptr()); // 0x7f89f841c028
}
```

clone()

```rust
fn main() {
  let string = foo();
  println!("{:p}", string.as_ptr()); // 0x7fc98be1c010
}

fn foo() -> String {
  let string = String::from("hello");
  println!("{:p}", string.as_ptr()); // 0x7fc98be1c010
  return string;
}
```

Giving ownership

borrowing



Passing a reference/borrowing

```rust
fn main() {
  let string = String::from("hello");
  println!("{:p}", string.as_ptr()); // 0x7f819641c010
  foo(&string);
  println!("{:p}", string.as_ptr()); // 0x7f819641c010
}

fn foo(string: &String) {
  println!("{:p}", string.as_ptr()); // 0x7f819641c010
}
```

Introduction – OS in Rust – Why Rust?

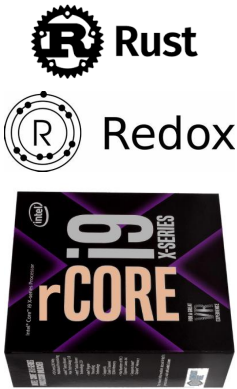
Passing a mutable reference between functions

```rust
fn main() {
  let mut string = String::from("hello");
  println!("{:p}", string.as_ptr()); // 0x7fc98be1c010
  foo(&mut string);
  println!("{:p}", string.as_ptr()); // 0x7fc98be1d000
}

fn foo(string: &mut String) {
  println!("{:p}", string.as_ptr()); // 0x7fc98be1c010
  string.push_str(".");
}
```

Dangling Reference, ERR!

```rust
fn main() {
  let string = foo();
}

fn foo() -> &String {
  let string = String::from("hello");
  println!("{}", string);
  &string
}
```

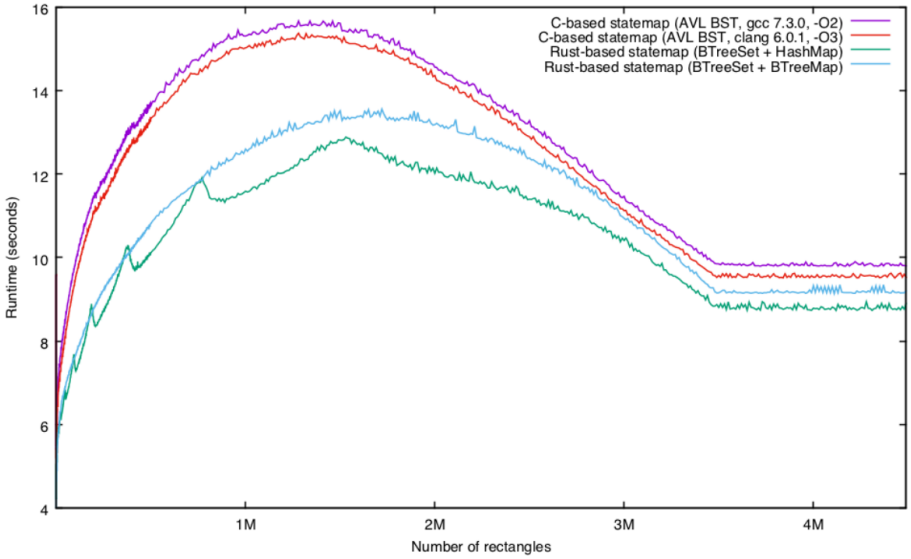
Introduction – OS in Rust



Is it time to rewrite the operating system in Rust?
Rust &OS in the 2010s

- First attempt at an operating system kernel in Rust seems to be Alex Light's Reenix, ca. 2015: a re-implementation of a teaching operating system in Rust as an undergrad thesis
- Since Reenix's first efforts, there have been quite a few small systems in Rust, e.g.: Redox, Tifflin, Tock, intermezzOS, RustOS/QuiltOS, Rux, and Philipp Oppermann's Blog OS, and rcore.
- Some of these are teaching systems (intermezzOS, Blog OS), some are unikernels (QuiltOS) and/or targeted at IoT (Tock)

Introduction – OS in Rust



Source: http://dtrace.org/blogs/bmc/2018/09/28/the-relative-performance-of-c-and-rust/

Introduction – OS in Rust

OS in Rust?

Rust has a number of other features that make it highly compelling for systems software implementation:

- Algebraic types allow robust, concise error handling
- Hygienic macros allow for safe syntax extensions
- Foreign function interface allows for full-duplex integration with C without sacrificing performance
- “unsafe” keyword allows for some safety guarantees to be surgically overruled (though with obvious peril)
- active community and ecosystem, etc.

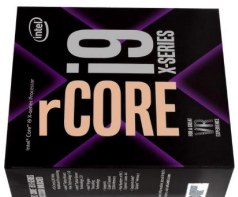
Introduction – OS in Rust

OS in Rust?

Rust has a number of other features that make it highly compelling for systems software implementation:

- If the history of operating systems implementation teaches us anything, it's that runtime characteristics trump development challenges!
- Structured languages (broadly) replaced assembly because they performed as well
- every operating system retains some assembly for reasons of performance!
- With its focus on performance and zero-cost abstractions, Rust does represent a real, new candidate programming language for operating systems implementation

Introduction – OS in Rust



Challenges of OS in Rust?

- While Rust's advantages are themselves clear, it's less clear what the advantage is when replacing otherwise working code
- For in-kernel code in particular, the safety argument for Rust carries less weight: in-kernel C tends to be de facto safe
- Rust does, however, presents new challenges for kernel development, esp. with respect to multiply-owned structures
- An OS kernel - despite its historic appeal and superficial fit for Rust - may represent more challenge than its worth

But what of hybrid/other approaches?

Introduction – OS in Rust

Hybrid approach I: Rust in-kernel components

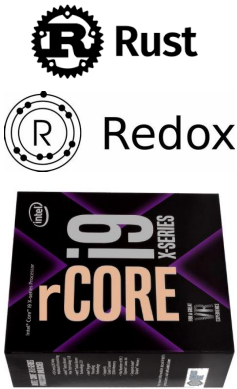
- One appeal of Rust is its ability to interoperate with C
- One hybrid approach to explore would be to retain a C-/assembly-based kernel while allowing for Rust-based in-kernel components like device drivers and filesystems
- This would allow for an incremental approach —and instead of rewriting, Rust can be used for new development
- An There is a prototype example of this in FreeBSD/Linux; others are presumably possible

Introduction – OS in Rust

Hybrid approach II: Rust OS components

- An operating system is not just a kernel!
- Operating systems have significant functionality at user-level: utilities, daemons, service-/device-/fault- management facilities, debuggers, etc.
- If anything, the definition of the OS is expanding to distributed system that represents a multi-computer control plane —that itself includes many components
- These components are much more prone to run-time failure.
- Many of these are an excellent candidate for Rust.

Introduction – OS in Rust



Hybrid approach III: Rust-based firmware/hypervisor

- Below the operating system lurks hardware-facing special- purpose software: firmware/hypervisor
- Firmware/hypervisor is a sewer of unobservable software with a long history of infamous quality problems
- Firmware has some of the same challenges as kernel development (e.g., dealing with allocation failures), but may otherwise be more amenable to Rust
- This is especially true when/where firmware is in user-space and is network-facing! (e.g., OpenBMC/OpenSBI)

References

- Is it time to rewrite the operating system in Rust? Bryan Cantrill, tech talk, 2018
- Ownership in Rust, Thomas Countz, 2018
- rust vs go https://jackyzhen.github.io/rust-vs-go-slides/