Why Write OS In Rust?

v2.0

Presented by RC

Outline

- Language Basics
- Memory Safety
- Expression
- Interoperability
- Performance
- Usability
- Drawbacks
- Summary

Outline

- **Language Basics ◁**
- Memory Safety
- Expression
- Interoperability
- Performance
- Usability
- Drawbacks
- Summary

Variables

- Immutable by default
- Shadowing

```
let x = 0;
x = 42; // error: cannot assign twice to immutable variable `x`

let mut y = 0;
y = 42; // ok
```

```
let x = foo();
let x = x.unwrap(); // this `x` is different from the above one
```

Ownership

- Move semantics

```rust
let x = String::from("foo");
let y = x;
println!("{}", x); // error: borrow of moved value: `x`
```

```cpp
// C++: similar but different
auto x = std::string("foo");
auto y = std::move(x);
std::cout << x << std::endl; // ok, `x` is in valid but unspecified state
```

Ownership

```
#[derive(Debug)]
struct MyString(String);

fn foo1(s: String) { println!("len: {}", s.len()); }
fn foo2(s: String) -> MyString { println!("len: {}", s.len()); MyString(s) }

pub fn main() {
    let x = String::from("foo");
    foo1(x);
    let x = String::from("bar");
    let x = foo2(x);
    println!("{:?}", x);
}
```

Borrowing

- Multiple immutable references, or
- Single mutable reference

```
let x = String::from("foo");
let y = &x;
let z = &x;
println!("{}, {}, {}", x, y, z); // ok
```

```
let mut x = String::from("foo");
let y = &mut x;
let z = &mut x; // error: cannot borrow `x` as mutable more than once at a time
y; z;
```

Borrowing

```
let mut x = 0;
let y = &x;
x = 42; // error: cannot assign to `x` because it is borrowed
// *(&mut x) = 42; // basically the same thing
y;
```

```
let mut x = 0;
let y = &mut x;
*y = 42; // ok
x; // error: cannot use `x` because it was mutably borrowed
// *(&x); // basically the same thing
y;
```

Smart Pointers

- `Box<T>`: data on the heap

```rust
let x = Box::new(42);
let mut y = x;
*y += 1;
// there is no `y = nullptr` in Rust
```

```cpp
// C++
auto x = std::make_unique<int>(42);
auto y = std::move(x);
*y += 1;
y = nullptr;
auto z = *y; // runtime error: segmentation fault
```

Smart Pointers

- `Rc<T>`: multiple ownership (reference count)

```
fn foo(val: Rc<String>) { val; /* consumes val */ }

let x = Rc::new(String::from("foo"));
foo(x.clone()); // pass a clone of Rc, instead of moving `x` in
println!("{}", x); // ok
```

Smart Pointers

- `Cell<T>`: interior mutability
- `RefCell<T>`: interior mutability + runtime borrow checking

```
let x = Cell::new(42);
let y = x.replace(0);
assert_eq!(x.get(), 0);
assert_eq!(y, 42);
```

```
let x = RefCell::new(42);
let ref1 = x.borrow();
let ref2 = x.borrow();
let ref3 = x.borrow_mut(); // runtime error: already borrowed
```

Doubly Linked List

- Why it's not easy to implement?
  - Because you have to think in Rust.
- How to implement?

```
struct Node<T> {
    pub data: T,
    pub prev: Option<Weak<RefCell<Node<T>>>>,
    pub next: Option<Rc<RefCell<Node<T>>>>,
}
```

  - Full example

Unsafe

In Safe Rust, the **compiler** ensures soundness;

In Unsafe Rust, **programmers** ensure soundness.

Read The Rustonomicon before writing unsafe code.

Outline

- Language Basics
- **Memory Safety ◁**
- Expression
- Interoperability
- Performance
- Usability
- Drawbacks
- Summary

Memory Safety

- Microsoft: 70% of all vulnerabilities in their products over the past decade have been caused by a lack of memory safety.
- Chromium: 70% of serious security bugs are memory safety problems.
- Android: 90% of their vulnerabilities are memory safety issues.
- Mozilla: 74% of Firefox's security bugs in its style component are memory safety bugs.

Memory Safety

- Spatial memory safety
- Temporal memory safety
- Thread safety

Spatial Memory Safety

How C sucks:

```
int someos_file_write(int fd, void *buf, size_t count) {
    int ret, cnt, remain = count; // implicit conversion
    // ...
    while (remain > 0) {
        fr->count /* ssize_t */ = cnt = MIN(remain, FS_BUF_SIZE);
        memcpy(fr->buf, buf, cnt);
        // write to file, ret is: 1) < 0 for failure; 2) >= 0 for bytes written
        ret = ipc_call(fr);
        buf = (char *)buf + ret; remain -= ret;
        if (ret != cnt) break;
    }
    return count - remain;
}
```

Spatial Memory Safety

At least 4 bugs in the above C code!

1. `int fd`: `fd` can be negative or zero
2. `void *buf`: `buf` can be null or have unmatched size
3. `remain = count`: `remain` can be negative or very large due to unmatched type
4. `ret = ipc_call(fr);`: no error checking for `ret`

Spatial Memory Safety

What features of Rust can help?

- Null safety
- Type safety
- Boundary check
- Error handling

Spatial Memory Safety

Same thing implemented in Rust:

```
fn someos_file_write(fd: FileDescriptor, buf: &mut [u8]) -> SomeOsResult<usize> {
    let count = buf.len();
    let mut written = 0usize;
    while written < count {
        let buf_slice = &mut buf[written..min(written + FS_BUF_SIZE, count)];
        let n = ipc_call(buf_slice)?;
        written += n;
        if n < buf_slice.len() {
            break;
        }
    }
    Ok(written)
}
```

Spatial Memory Safety

What if forgot `min(written + FS_BUF_SIZE, count)`?

```
let buf_slice = &mut buf[written..written + FS_BUF_SIZE];
```

If `written + FS_BUF_SIZE` is larger than `buf.len()`, it'll panic:

```
thread 'main' panicked at 'range end index 1024 out of range for slice of length 512'
```

Temporal Memory Safety

A bad example in C:

```c
void usb_request_async(int req, void *buffer, void (*callback)(int, void *)) {
    global_urb->req = req;
    global_urb->buffer = buffer;
    global_urb->callback = callback;
    usb_send_urb_async(global_urb);
}

void usb_request_cb(int req, void *buffer) { /* access buffer */ }
void func() {
    unsigned char buf[4096];
    usb_request_async(USB_REQ_GET_XXX, buf, usb_request_cb);
}
```

Temporal Memory Safety

What features of Rust can help?

- Lifetime
- Ownership

Temporal Memory Safety

Try the same in Rust (consider only immutable buffer for simplicity):

```
fn usb_request_async<'a>(
    req: UsbRequest,
    buffer: &'a [u8],
    callback: Box<dyn Fn(UsbRequest, &[u8]) + Send + 'static>,
) {
    std::thread::spawn(move || { // simulate async usb request callback
        callback(req, buffer);
    });
}
```

```
error: `buffer` has lifetime `'a` but it needs to satisfy a `'static` lifetime requirement
```

Temporal Memory Safety

Change a little bit according to the error message:

```
fn usb_request_async(
    req: UsbRequest,
    buffer: &'static [u8],
    callback: Box<dyn Fn(UsbRequest, &[u8]) + Send + 'static>,
) {
    std::thread::spawn(move || { // simulate async usb request callback
        callback(req, buffer);
    });
}
```

Temporal Memory Safety

Then use `usb_request_async`:

```
fn usb_request_cb(req: UsbRequest, buffer: &[u8]) {
    println!("{:?}", buffer);
}

fn func() {
    let buffer: [u8; 1024] = [0; 1024];
    usb_request_async(UsbRequest::GetXxx, &buffer, Box::new(usb_request_cb));
}
```

```
error: `buffer` does not live long enough
```

Temporal Memory Safety

Possibly a better design:

```
struct Urb {
    req: UsbRequest,
    buffer: Vec<u8>,
    callback: Box<dyn Fn(UsbRequest, &[u8]) + Send + 'static>,
}

impl Urb {
    fn new(req: UsbRequest, buf_size: usize,
           callback: impl Fn(UsbRequest, &[u8]) + Send + 'static) -> Self {
        Self { req, buffer: vec![0; buf_size], callback: Box::new(callback) }
    }
}
```

Temporal Memory Safety

```
fn usb_send_urb_async(urb: Urb) {
    std::thread::spawn(move || { // simulate async usb request callback
        let cb = urb.callback;
        cb(urb.req, &urb.buffer);
    });
}

fn usb_request_cb(req: UsbRequest, buffer: &[u8]) {
    println!("{:?}", buffer);
}

fn func() {
    let urb = Urb::new(UsbRequest::GetXxx, 1024, usb_request_cb);
    usb_send_urb_async(urb);
}
```

Thread Safety

Mutable static variables can't be accessed **safely**:

```
static mut BALANCE: u32 = 1;

fn func() {
    std::thread::spawn(|| BALANCE += 2);
    BALANCE -= 1;
}
```

```
error: use of mutable static is unsafe and requires unsafe function or block
```

Thread Safety

So you may want interior mutability:

```
static BALANCE: RefCell<u32> = RefCell::new(1);

fn func() {
    std::thread::spawn(|| BALANCE.replace_with(|&mut old| old + 2));
    BALANCE.replace_with(|&mut old| old - 1);
}
```

```
error: `RefCell<u32>` cannot be shared between threads safely
```

Thread Safety

Variables shared between threads must impl `Sync` trait:

- `AtomicXXX`, `Mutex`, Crossbeam, etc.

```
static BALANCE: AtomicU32 = AtomicU32::new(1);

fn func() {
    std::thread::spawn(|| BALANCE.fetch_add(2, Ordering::Relaxed));
    BALANCE.fetch_sub(1, Ordering::Relaxed);
}
```

Thread Safety

Another way to share - multiple ownership:

```
fn func() {
    let balance = Rc::new(AtomicU32::new(1));
    let balance_clone = balance.clone();
    std::thread::spawn(move || balance_clone.fetch_add(2, Ordering::Relaxed));
    balance.fetch_sub(1, Ordering::Relaxed);
}
```

```
error: `Rc<AtomicU32>` cannot be sent between threads safely
```

Thread Safety

Variables sent across threads must impl `Send` trait:

```
fn func() {
    let balance = Arc::new(AtomicU32::new(1));
    let balance_clone = balance.clone();
    std::thread::spawn(move || balance_clone.fetch_add(2, Ordering::Relaxed));
    balance.fetch_sub(1, Ordering::Relaxed);
}
```

Outline

- Language Basics
- Memory Safety
- **Expression ◁**
- Interoperability
- Performance
- Usability
- Drawbacks
- Summary

Expression

Can Rust express everything that C can?

In theory, YES, both are turing-complete.

In practice, ALMOST, with some `unsafe` code. When you can't implement some logic in Rust, you probably should rethink the design.

Low Level Programming

Booting:

```
global_asm!(include_str!("entry.asm"));

#[no_mangle]
#[link_section = ".text.boot"]
extern "C" fn _start_rust() -> ! {
    #[link_section = ".data.boot"]
    static START: spin::Once<()> = spin::Once::new();
    START.call_once(|| { clear_bss(); memory::create_boot_pt(); });
    memory::enable_mmu();
    unsafe { _start_kernel() }
}
```

Low Level Programming

Accessing system registers:

```
#[link_section = ".text.boot"]
pub fn enable_mmu() {
    // set memory attribute indirection
    MAIR_EL1.write(...);
    // configure stage 1 of the EL1 translation regime
    TCR_EL1.write(...);
    // set both TTBR0_EL1 and TTBR1_EL1
    let pgd_frame = PhysFrame::<Size4KiB>::of_addr(unsafe { &BOOT_PGD as *const PageTable } as u64);
    translation::ttbr_el1_write(0, pgd_frame);
    translation::ttbr_el1_write(1, pgd_frame);
    translation::local_invalidate_tlb_all();
    // enable MMU
    SCTLR_EL1.modify(SCTLR_EL1::M::Enable + SCTLR_EL1::C::Cacheable + SCTLR_EL1::I::Cacheable);
    // ensure that MMU is enabled before the next instruction
    unsafe { barrier::isb() }
}
```

Low Level Programming

Driver:

```
fn handle_irq(&self) {
    loop {
        let irqstat = self.read_cpu_reg(GICC_IAR); /* MMIO inside */
        let irqnr = irqstat & GICC_IAR_INT_ID_MASK;
        match irqnr {
            0..=15 => crate::irq::handle_inter_processor(irqnr),
            16..=31 => crate::irq::handle_local(irqnr - 16),
            32..=1020 => crate::irq::handle_shared(irqnr - 32),
            _ => break,
        }
        self.write_cpu_reg(GICC_EOIR, irqstat);
    }
}
```

More Features

- Crate & Module: better modularity
- Ownership & Lifetime: memory safety
- Trait: generic programming
- Result & Error: better error handling
- Closure & Iterator: functional programming
- Macro: meta programming

More Features

Traits in drivers:

```
pub trait IrqChip {
    fn enable_local_irq(&self, ppi: u32);
    // ...
    fn handle_irq(&self);
}

pub struct Gic { /* ... */ }

impl IrqChip for Gic {
    fn enable_local_irq(&self, ppi: u32) { /* ... */ }
    // ...
    fn handle_irq(&self) { /* ... */ }
}
```

More Features

Traits in memory management:

```
pub trait HardwareMemory {
    fn memory_get_available_ram() -> Vec<Range<PhysAddr>>;

    type PageTable: PageTable;
}

pub trait PageTable {
    fn new(is_kernel: bool) -> Self;
    unsafe fn map_kernel(&mut self) -> KernResult<()>;
    unsafe fn map(&mut self, va_range: Range<VirtAddr>, pa: PhysAddr, ...) -> KernResult<()>;
}
```

More Features

```
impl HardwareMemory for hw {
    fn memory_get_available_ram() -> Vec<Range<PhysAddr>> { /* ... */ }

    type PageTable = PageTableImpl;
}

pub struct PageTableImpl { /* ... */ }

impl PageTable for PageTableImpl { /* ... */ }
```

More Features

Functional programming:

```
pub(super) fn init() {
    let mut allocator = FRAME_ALLOCATOR.lock();
    hw::memory_get_available_ram().iter().for_each(|range| {
        let start_pfn = phys_to_pfn(align_up(range.start));
        let end_pfn = phys_to_pfn(align_down(range.end));
        allocator.add_frame(start_pfn, end_pfn);
    })
}

pub unsafe fn alloc_frame() -> Option<PhysAddr> {
    FRAME_ALLOCATOR.lock().alloc(1).map(pfn_to_phys)
}
```

More Features

DSL supported by macro:

```
register_bitfields! {u64,
    pub SCTLR_EL1 [
        I   OFFSET(12) NUMBITS(1) [ NonCacheable = 0, Cacheable = 1 ],
        NAA OFFSET(6)  NUMBITS(1) [ Disable = 0, Enable = 1 ],
        SA0 OFFSET(4)  NUMBITS(1) [ Disable = 0, Enable = 1 ],
        SA  OFFSET(3)  NUMBITS(1) [ Disable = 0, Enable = 1 ],
        C   OFFSET(2)  NUMBITS(1) [ NonCacheable = 0, Cacheable = 1 ],
        // ...
    ]
}
```

Outline

- Language Basics
- Memory Safety
- Expression
- **Interoperability** ◁
- Performance
- Usability
- Drawbacks
- Summary

Interoperability

Call C from Rust:

```
extern "C" {
    fn some_function_in_c(buf: *mut u8, n: usize) -> usize;
}

fn func() {
    let mut buf = [0u8; 1024];
    let buf_raw_ptr = &mut buf[0] as *mut u8;
    let ret = unsafe { some_function_in_c(buf_raw_ptr, buf.len()) };
}
```

Interoperability

Call Rust from C:

```rust
#[no_mangle]
extern "C" fn some_function_in_rust(buf: *mut u8, n: usize) -> usize {
    let buf = unsafe { core::slice::from_raw_parts_mut(buf, n) };
    println!("{:?}", buf); buf.len()
}
```

```c
extern size_t some_function_in_rust(unsigned char *buf, size_t n);
void func() {
    unsigned char buf[1024];
    some_function_in_rust(buf, 1024);
}
```

Interoperability

- Chromium: Rust and C++ interoperability
- Firefox: Integrating Rust and C++ in Firefox
- Android: Rust/C++ interop in the Android Platform
- Linux: Supporting Linux kernel development in Rust

Outline

- Language Basics
- Memory Safety
- Expression
- Interoperability
- **Performance** ◁
- Usability
- Drawbacks
- Summary

Performance

According to The Computer Language Benchmarks Game, Rust achieves same or even better performance than C and C++.

Features that make Rust fast:

- Cross-language LTO
- Zero-cost abstractions, e.g. Iterators
- Faster impl for single-threaded context, e.g. Rc vs Arc
- Carefully implemented builtin data structures, e.g. BTreeMap
- ……

Outline

- Language Basics
- Memory Safety
- Expression
- Interoperability
- Performance
- **Usability ◁**
- Drawbacks
- Summary

Usability

Tools:

- Rustup makes toolchain management easy
- Cargo makes dependency management easy
- rust-analyzer is an excellent language server
- rustc produces comprehensible error messages
- rustfmt formats Rust code according to style guide
- rust-clippy can catch common mistakes in Rust code
- bindgen helps generate Rust FFI bindings to C and C++ libraries
- ……

Usability

Dependency management:

```
[dependencies]
spin = "0.7.0"
buddy_system_allocator = { git = "https://github.com/richardchien/buddy_system_allocator.git" }
log = "0.4.0"
lazy_static = { version = "1.4.0", features = ["spin_no_std"] }
bitflags = "1.2.1"

[target.'cfg(target_arch = "aarch64")'.dependencies]
aarch64 = { git = "https://github.com/richardchien/aarch64", rev = "5dc2a13" }
```

Usability

`Debug` trait for easy debugging:

```
#[repr(u8)]
#[derive(Debug)]
enum IntType {
    SyncEl1t = 1,
    IrqEl1t = 2,
    // ...
}

#[no_mangle]
extern "C" fn _handle_interrupt(int_type: IntType) {
    println!("Interrupt occurred, type: {:?}", int_type);
}
```

Outline

- Language Basics
- Memory Safety
- Expression
- Interoperability
- Performance
- Usability
- **Drawbacks ◁**
- Summary

Drawbacks

- Steep learning curve
- Many unstable features
- Low speed compilation
- Small ecosystem (but growing quickly)

Outline

- Language Basics
- Memory Safety
- Expression
- Interoperability
- Performance
- Usability
- Drawbacks
- **Summary ◁**

Summary

- Safe by default
- Rich language features without need for `std`
- Harder to write, but easier to be correct once compiled
- Easy-to-use tools
- Backed by big companies: Google, Microsoft, Huawei, Mozilla, ...
- Future seems bright

Special thanks to Alex Chi for reviewing the slides.

References

- The Rust Programming Language & The Rustonomicon
- Considering Rust by Jon Gjengset
- Writing Linux Kernel Modules in Safe Rust by Geoffrey Thomas & Alex Gaynor
- Implications of Rewriting a Browser Component in Rust by Diane Hosfelt
- Speed of Rust vs C by kornelski
- C++ Is Faster and Safer Than Rust: Benchmarked by Yandex by Roman Proskuryakov

OSes Written in Rust

- Redox: A Unix-like OS with a modern microkernel
- Tock: An embedded OS for low-memory and low-power microcontrollers
- RedLeaf: A new OS aimed at leveraging Rust language features for developing safe systems
- Theseus: A new OS for experimenting with novel OS structure, better state management, and how to shift OS responsibilities like resource management into compiler
- rCore: A teaching OS based on BlogOS, providing a subset of Linux syscalls