# Why Write OS In Rust?

## ChCoreCon #7, 3/1/2021

Presented by [RC]

# Outline

- Safety
- Expression
- Interoperability
- Performance
- Usability
- Drawbacks
- Summary

# Outline

- **Safety** 👉
- Expression
- Interoperability
- Performance
- Usability
- Drawbacks
- Summary

# Safety

- Spatial memory safety
- Temporal memory safety
- Thread safety

According to the MSRC blog, **70%** of the security issues that it assigns a CVE to are memory safety issues!

# Spatial memory safety

How C sucks:

```c
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
3. `remain = count`: `remain` can be negative due to unmatched type
4. `buf = (char *)buf + ret`: `ret` can be negative, causing out-of-bound access

# 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', src/main.rs:23:30
```

# 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[E0759]: `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`:

```rust
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[E0597]: `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

Features of Rust to prevent from data races:

- Global static variables can't be modified *safely*

```rust
static mut GLOBAL_INT: u32 = 1;

fn func() {
    // bad practice
    std::thread::spawn(|| unsafe { GLOBAL_INT = 3 });
    unsafe { GLOBAL_INT = 2 }
}
```

# Thread safety

- Variables that are shared between threads must impl `Sync`

```
// static GLOBAL_INT_BAD: RefCell<u32> = RefCell::new(1);
lazy_static! {
    static ref GLOBAL_INT: Mutex<u32> = Mutex::new(1);
}

fn func() {
    std::thread::spawn(|| *GLOBAL_INT.lock().unwrap() = 3);
    *GLOBAL_INT.lock().unwrap() = 2;
}
```

# Thread safety

- Variables that are sent across threads must impl `Send`

```
fn func() {
    let shared_int = Arc::new(Mutex::new(1));
    let shared_int_clone = shared_int.clone();

    std::thread::spawn(move || *shared_int_clone.lock().unwrap() = 3);
    *shared_int.lock().unwrap() = 2;
}
```

# Outline

- 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.

# Expression

Example for booting:

```
#[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() }
}
```

# Expression

Example for driver:

```
fn handle_irq(&self) {
    loop {
        let irqstat = self.read_cpu_reg(GICC_IAR);
        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);
    }
}
```

# Expression

Much better than C:

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

# Expression

Compared to C++:

- Similar funtionality
- Safe by default
- `core` and `alloc` crates without need for `std`

# Outline

- 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:

```
#[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()
}
```

```
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);
}
```

# Outline

- 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
- ……

# Performance

Open source projects (re)written in Rust that're proved to be fast(er):

- ripgrep
- alacritty
- fd
- actix-web
- serde
- servo
- ……

# Outline

- 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

Example for 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

- Safety
- Expression
- Interoperability
- Performance
- Usability
- **Drawbacks** 👉
- Summary

# Drawbacks

- Steep learning curve
- Many unstable features
- Slow compile speed
- Small ecosystem (but growing quickly)

# Outline

- 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 some big companies: Google, Microsoft, 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
- Why Rust for safe systems programming by MSRC
- 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