Qualitative and quantitative evaluation of writing an OS kernel in Rust.

Narek Galstyan ‘20
Advisor: Amit Levy
Why OS in Rust?
Why OS in Rust?

• No dangling pointers, buffer overflows
Why OS in Rust?

- No dangling pointers, buffer overflows
- No Garbage collection, safety enforced through lifetimes
Why OS in Rust?

- No dangling pointers, buffer overflows
- No Garbage collection, safety enforced through lifetimes
- Algebraic types (will see later)
Why OS in Rust?

- No dangling pointers, buffer overflows
- No Garbage collection, safety enforced through lifetimes
- Algebraic types (will see later)
- Most of type checking and memory management is planned compile time and so minimal overhead after it is compiled
Motivation and Goal

• Kernel is responsible for every program in the computer. Insecure kernel → insecure computer regardless of the application level security
Motivation and Goal

- Kernel is responsible for every program in the computer. Insecure kernel $\rightarrow$ insecure computer regardless of the application level security.

- Drivers comprise over 70% of kernel code. They do not need all privileges given to kernel mode but usually run in Ring 0 for efficiency.
Motivation and Goal

- Kernel is responsible for every program in the computer. Insecure kernel $\rightarrow$ insecure computer regardless of the application level security.

- Drivers comprise over 70% of kernel code. They do not need all privileges given to kernel mode but usually run in Ring 0 for efficiency.
  - Rust can limit the scope of potential damage caused by drivers.
Motivation and Goal

• Kernel is responsible for every program in the computer. Insecure kernel → insecure computer regardless of the application level security.

• Drivers comprise over 70% of kernel code. They do not need all privileges given to kernel mode but usually run in Ring 0 for efficiency.
  
  • Rust can limit the scope of potential damage caused by drivers

• Goal: Write a kernel in Rust to evaluate its benefits for kernel programming and the cost we pay
Problem Background and Related Work

- Cody Cutler, M. Frans Kaashoek, and Robert T. Morris: **Writing Kernel in Go**
  - Similar motivation and goals
  - They had to port GC to bare metal and most of their issues were connected with this

- Stanford’s experimental OS course in Rust
  - Ran only one semester, built for ARM

- Tock

- Philip Opperman’s Blog OS
Approach

• Build a kernel modeled after xv6 but still in idiomatic Rust

• Pause and evaluate advantages of C and advantages of Rust during development whenever there is an opportunity (in following slides)

• Quantify
  • developer performance cost of satisfying Rust requirements
  • the hardware performance cost of abstraction
Implementation

- OS in a nutshell
Implementation

• OS in a nutshell
Implementation

- OS in a nutshell

Diagram:
- Disk
- Memory
- Bootloader
- Kernel

0x0
Implementation

- OS in a nutshell

![Diagram showing OS components and disk layout]

- BIOS data
- VGA Display
- Devices
- BIOS
Implementation

• OS in a nutshell

![Diagram showing OS components and boot process]

- BIOS data
- Bootloader
- VGA display
- Devices
- BIOS

Disk

Kernel

VGA

Display

0x0
Implementation

- OS in a nutshell
OS in a nutshell:

- BIOS data
- Kernel
- VGA Display
- Devices
- BIOS

Disk

Implementation

0x0
• OS implementation

- BIOS data
- VGA driver
- Kernel
- VGA display
- Devices
- BIOS
Implementation

OS in a nutshell

0x0

BIOS data
VGA driver
Disk driver
VGA Display
Devices
BIOS

Disk

BIOS

VGA

Display

Kernel

Bootloader

Kernel

Disk driver

Bootloader

VGA driver

BIOS data
OS in a nutshell:

- BIOS
- Bootloader
- Kernel
- Memory
- Disk
- VGA
- Display
- Devices
- BIOS

0x0
• OS in a nutshell

- BIOS data
- VGA driver
- Disk driver
- Interrupts
- Scheduler
- VGA Display
- Devices
- BIOS
• OS in a nutshell

- BIOS data
- VGA driver
- Disk driver
- Interrupts
- Scheduler
- File system
- VGA Display
- Devices
- BIOS
Implementation: Lock
Implementation: Lock

- There are multiple threads of execution even in single-threaded kernels
Implementation: Lock

- There are multiple threads of execution even in single-threaded kernels
- Task: provide necessary data access isolation primitives
Implementation: Lock

- There are multiple threads of execution even in single-threaded kernels.
- Task: provide necessary data access isolation primitives.
- Advantage of Rust: Language guarantees that locks are acquired before data usage.

**Rust with spin lock**

```rust
f.lock().read()
```

**C**

```c
int fread(struct file *f)
{
    int r;
    ilock(f->ip);
    if((r = read(f->ip)) > 0)
        iunlock(f->ip);
    return r;
}
```
Implementation: x86_64 interface

• Hardware primitives specified by x86_64 architecture that are necessary in other parts of the kernel

• Task: provide Rust api to the primitives

• Advantage of Rust: None
Implementation: x86_64 interface

- Hardware primitives specified by x86_64 architecture that are necessary in other parts of the kernel
- Task: provide Rust api to the primitives
- Advantage of Rust: None
Implementation: DiskIO

- Hardware provides byte level sequential disk controllers through I/O in and out assembly instructions
- Task: Build a safe data streaming API on top of this
- Advantage of Rust: Has enough expressive power that allows to safely expose inherently unsafe I/O functionality
Implementation: DiskI/O

```rust
fn command_to_drive(ctrl: IdeController, port: IdePortArgs, value: u8) {
    unsafe {
        outb(<port: ctrl as u16 | port as u16, value>);
    }
}
```
Implementation: DiskIO

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u16)]
enum IdeController {
    Primary = 0x1f0,
    Secondary = 0x170,
}
```

```rust
enum IdePortArgs {
    ATA_REG_DATA = 0x00,
    ATA_REG_LBA3 = 0x09,
    ATA_REG_CONTROL = 0x0C,
}
```

```rust
fn command_to_drive(ctrl: IdeController, port: IdePortArgs, value: u8) {
    unsafe {
        outb(port: ctrl as u16 | port as u16, value);
    }
}
```
Analysis: Zero Cost Abstractions

- Load args
- Command logic
- Return

```
example::command_to_drive:
4
pushq %rax
movb %dl, %al
movb %sil, %cl
movw %di, %r8w
movzbl %cl, %edx
movw %dx, %r9w
orw %r9w, %r8w
movzwl %r8w, %edi
movzbl %al, %esi
callq *example::outb@GOTPCREL(%rip)
popq %rax
retq
```
Conclusion: So should we rewrite OS in Rust?

• Yes
  • Macros inside language semantics
  • Compiler helps a lot and makes the code easier to maintain
  • Language protection is very useful when hardware protection is not available
Conclusion: So should we rewrite OS in Rust?
Conclusion: So should we rewrite OS in Rust?

• Maybe Not
Conclusion: So should we rewrite OS in Rust?

• Maybe Not

  • Too high level to be productive, sometimes even C is too high level
Conclusion: So should we rewrite OS in Rust?

• Maybe Not

• Too high level to be productive, sometimes even C is too high level

```c
void encodeGdtEntry(uint8_t *target, struct GDT source)
{
    // Check the limit to make sure that it can be encoded
    if ((source.limit > 65536) &&
        (source.limit & 0xFFF) != 0xFFF) {
        kerror("You can't do that!");
    }
    if (source.limit > 65536) {
        // Adjust granularity if required
        source.limit = source.limit >> 12;
        target[6] = 0xC0;
    } else {
        target[6] = 0x40;
    }
    ...
}
```
Conclusion: So should we rewrite OS in Rust?

- Maybe Not

  - Too high level to be productive, sometimes even C is too high level
Conclusion: So should we rewrite OS in Rust?

• Maybe Not
  • Too high level to be productive, sometimes even C is too high level
  • Benefits of rewriting existing monolithic OSes are marginal
Conclusion: So should we rewrite OS in Rust?

• Maybe Not

  • Too high level to be productive, sometimes even C is too high level
  • Benefits of rewriting existing monolithic OSes are marginal
  • Qualitatively it is harder to write an OS in Rust
Bibliography


• *Writing an OS in Rust (Second Edition)* Philipp Oppermann's blog, https://os.phil-opp.com/

• CS140e (Winter 2018) An Experimental Course on OSes, https://cs140e.sergio.bz/

• *Is It Time to Rewrite the Operating System in Rust?* Qcon 2019 talk by Bryan Cantrill, https://www.infoq.com/presentations/os-rust
Appendix - Ownership

// will not work
struct Node {
    next: &mut Node,
    previous: &mut Node,
    data: Foo,
}

// will work, requires unsafe
struct Node_raw {
    next: *mut Node_raw,
    previous: *mut Node_raw,
    data: Foo,
}
Appendix - Lifetimes

```c
int *danger() {
    int a = 4;
    return &a;
}
```

```c
int *evenMoreDanger() {
    int *a = malloc(sizeof(int));
    if (a) *a = 4;
    return a
}
```
Implementation: VGA Driver

- Hardware provides memory mapped video graphics array
- Task: Implement safe API around unsafe memory accesses
- Advantages of Rust:
  - Type checked arguments
  - Bound checked arrays

```rust
#[repr(u8)]
pub enum Color {
    Black = 0,
    Blue = 1,
    Green = 2,
    /**/
    White = 15,
}
```
Implementation: Safe Scheduler

- Rust detects memory safety and concurrency issues
- These are often not very crucial in low level kernel programming
- When writing more complex conceptually higher level algorithmic code (like a scheduler) it is good to
Implementation: Safe Scheduler

• When writing complex conceptually high level algorithmic code (like a scheduler) it is good to guarantee local scope for any bug
  • Round Robin, First-Come First-Served, Multi-level queues, etc
• Would be great to be able to write a scheduler in safe Rust without significant runtime overhead
Implementation: Safe Scheduler

```rust
#[no_mangle]
pub fn schedulepcbMutex: &Mutex<[PCB;NUM_PROCS]>, current: usize) -> usize {
    /* scheduling logic:
    // e.g.
    (current + 1) % NUM_PROCS
    */

```