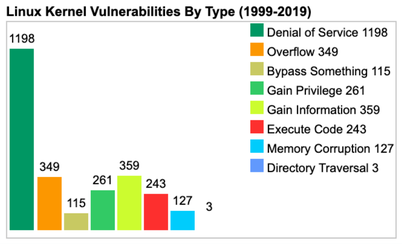
Benefits and Costs of Writing an OS kernel in Rust

PRINCETON UNIVERSITY

Narek Galstyan under the direction of Prof. Amit Levy

Motivation

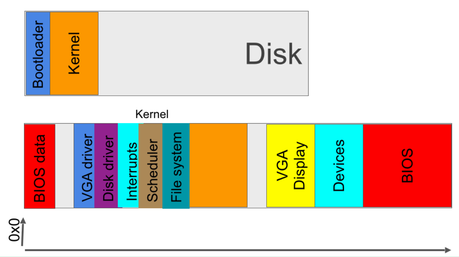
- Many of vulnerabilities found in OS kernels could be eliminated had kernels been written in more expressive programming languages
- Rust is gaining traction in the systems community as a replacement of C
- IoT ⇨ new Oses needed
- No direct evaluation of Rust for OS development



Goals

- Write a kernel in Rust
  - Implement OS kernel in Rust running directly on x86 chips
  - Model kernel components after equivalent components in xv6 pedagogical kernel and evaluate the resulting kernel against xv6
- Evaluate benefits of Rust and the cost we pay for them
  - Evaluate the advantages of Rust type system and memory management model for kernel development
  - Estimate performance over head due to Rust
  - Argue about difficulty of using Rust vs C in low level systems

The diagram *shows the disk and memory layout once the kernel is loaded into memory. Kernel components outlined in the memory layout were implemented in Rust during this project*



Related Work

1. Singularity: a Microsoft research Operating system written in Sing#, a flavor of C#
   - Came before Rust and incurred significant overhead
2. Biscuit: a POSIX compatible kernel written in Go to explore the use of higher level languages in Operating system design
   - Required a port of Go garbage collector to bare metal
   - Has significant performance overhead due to GC
3. CS140e: Stanford's experimental OS course taught in Rust

Acknowledgments

Special thanks to Professor Amit Levy for:

- Patiently explaining Rust
- Helping me navigate x86
- Valuable systems research insights
- And more (in current and next semester)

Disk IO

- Hardware provides addressable access to permanent storage at 256byte per read/write granularity
- Need to write a driver that will provide abstract streams to the rest of the system
- Hardware port IO is inherently unsafe as allows arbitrary data/memory corruption
- Rust type system allows to expose a constrained safe interface

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

#[allow(non_camel_case_types)]
#[allow(dead_code)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u8)]
enum IdeCommands {
    ATA_CMD_READ_PIO = 0x20,
    ATA_CMD_READ_PIO_EXT = 0x24,
    ATA_CMD_READ_DMA = 0xC8,
    ATA_CMD_READ_DMA_EXT = 0x25,
    ...
}
```

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

Listing 3: A function used to communicate with disk controller from Rust

Performance Cost

Rust Provides zero-cost abstractions when possible: The Rust code above compiles to assembly with no type-checking overhead

```
command_to_drive:
        pushq   %rax
        movb    %dl, %al
        movb    %sil, %cl
        movw    %di, %r8w
        movzbl  %cl, %edx
        movw    %dx, %r9w
        orw     %r9w, %r8w
        outb    %al, %r8w
        popq    %rax
        retq
```

- In other instances such as implementation of mutual exclusion, scheduling, etc. there is performance overhead due to Rust
- One could argue, in the above cases benefits of Rust are bigger as Rust:
  - Allows implementation of non-advisory, enforceable locks
  - Provides better module isolation
  - Enables fine-grained access control of memory and data structures thereby bounding impact of a bug

Future Work

- Rewrite parts of existing kernel (Linux or other C kernels) modules in Rust and integrate them into main C codebase
  - Allows to use Rust only for tasks it is best at
  - Enables to directly measure time overhead of using Rust
- Target hardware architectures other than x86
- Evaluate the development of more complex and higher level kernel modules in Rust

Safe Kernel Scheduler

- Process schedulers can be arbitrarily complex and workload specific which means they may not be written by kernel developers
- For performance reasons scheduler must run in kernel mode which in C kernels means it can cause arbitrary bugs both in the kernel and inside the processes it orchestrates
- In Rust, we can do better:

```rust
pub fn schedule(pcbs: &[PCB;NUM_PROCS], current:usize) -> usize {
    pcbs[current..].iter().enumerate()
    .filter(|(i,e)| e.in_use)
    .map(|e|e.0)
    .next().unwrap_or(current)
}
```

  - Scheduler has read only access to pcb array
  - Can make scheduling decisions based on process state
  - Rust's memory ownership system will not allow it to modify process state
  - Scheduler will still run in kernel mode thereby providing necessary performance but cannot modify arbitrary memory

Should we write Operating Systems in Rust?

- Yes!
  - Rust provides many language features such as rich macros, algebraic types, ownership tracking, integrated build system etc. that are quite useful in kernel development
- But be careful,
  - When writing kernels, sometimes even C is too high level and additional constraints of Rust may just slow you down
- And choose wisely which parts to write in Rust.
  - Based on my experience in this project, Rust features were more useful in higher level kernel structures. Rust's type system was very useful when enforcing complex protocol invariants
  - So, better to write network protocols, filesystems, schedulers of OS in Rust as opposed to lower level kernel modules.

References

- C. Cutler, M. F. Kaashoek, and R. T. Morris, "*The benefits and costs of writing a POSIX kernel in a high-level language*," in 13th USENIX (OSDI 18). Carlsbad, CA: USENIX Association, 2018, pp. 89–105.
- P. Oppermann, "Writing an os in rust (second edition)," Blog posts, 2018. Available: https://os.phil-opp.com/
- S. Benitez, "*An experimental course on operating systems (cs140e),*" University Course, 2018. Available: https://cs140e.sergio.bz/
- A. Levy et al., "*Multiprogramming a 64kb computer safely and efficiently,*" SOSP '17. New York, NY, USA: ACM, 2017, pp. 234–251. Available: http://doi.acm.org/10.1145/3132747.3132786