The Rust Way of OS Development

Philipp Oppermann

May 30, 2018 HTWG Konstanz

About Me

- Computer science student at KIT (Karlsruhe)
- *“Writing an OS in Rust”* blog series (os.phil-opp.com)
- Embedded Rust development

Contact:

- phil-opp on GitHub
- Email: hello@phil-opp.com

Rust

- 3 year old programming language
- Memory safety without garbage collection
- Used by Mozilla, Dropbox, Cloudflare, …

```rust
enum Event {
    Load,
    KeyPress(char),
    Click { x: i64, y: i64 }
}

fn print_event(event: Event) {
    match event {
        Event::Load => println!("Loaded"),
        Event::KeyPress(c) => println!("Key {} pressed", c),
        Event::Click {x, y} => println!("Clicked at x={}, y={}", x, y),
    }
}
```

OS Development

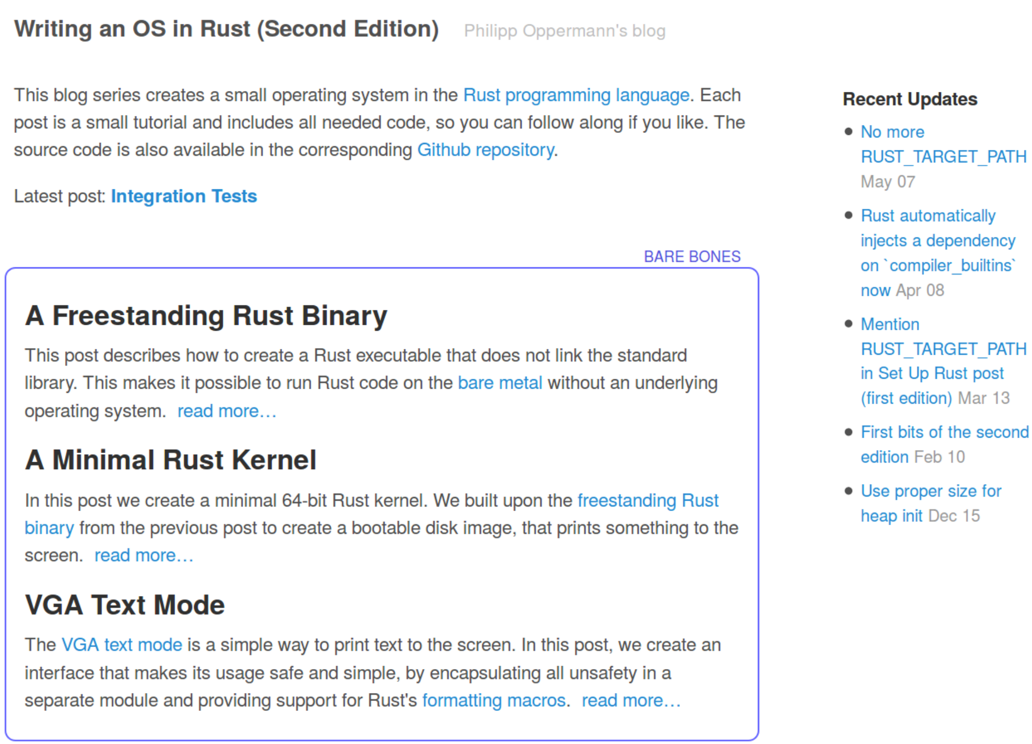
- "Bare metal" environment
  - No underlying operating system
  - No processes, threads, files, heap, …

Goals

- Abstractions
  - For hardware devices (drivers, files, …)
  - For concurrency (threads, synchronization primitives, …)
- Isolation (processes, address spaces, …)
- Security

OS Development in Rust

- **Writing an OS in Rust**: Tutorials for basic functionality
  - Booting, testing, CPU exceptions, page tables
  - No C dependencies
  - Works on Linux, Windows, macOS



Writing an OS in Rust (Second Edition) Philipp Oppermann's blog

This blog series creates a small operating system in the Rust programming language. Each post is a small tutorial and includes all needed code, so you can follow along if you like. The source code is also available in the corresponding Github repository.

Latest post: **Integration Tests**

BARE BONES

A Freestanding Rust Binary

This post describes how to create a Rust executable that does not link the standard library. This makes it possible to run Rust code on the bare metal without an underlying operating system. read more…

A Minimal Rust Kernel

In this post we create a minimal 64-bit Rust kernel. We built upon the freestanding Rust binary from the previous post to create a bootable disk image, that prints something to the screen. read more…

VGA Text Mode

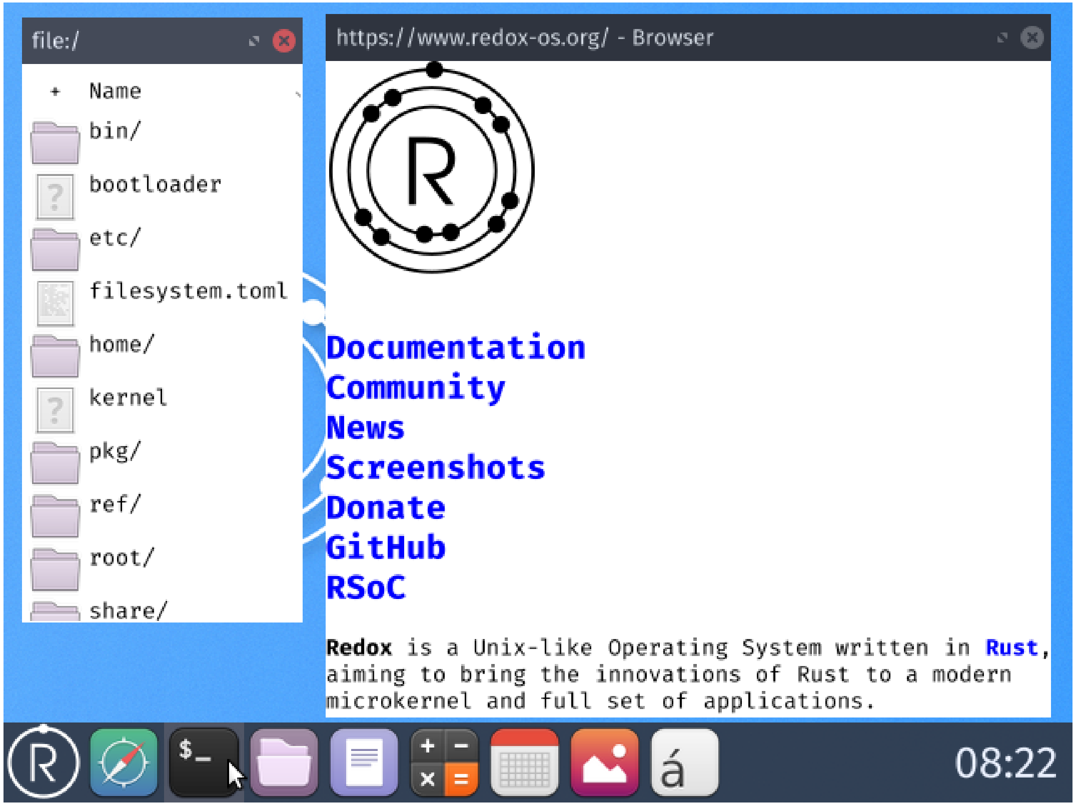
The VGA text mode is a simple way to print text to the screen. In this post, we create an interface that makes its usage safe and simple, by encapsulating all unsafety in a separate module and providing support for Rust's formatting macros. read more…

Recent Updates

- No more RUST_TARGET_PATH May 07
- Rust automatically injects a dependency on `compiler_builtins` now Apr 08
- Mention RUST_TARGET_PATH in Set Up Rust post (first edition) Mar 13
- First bits of the second edition Feb 10
- Use proper size for heap init Dec 15

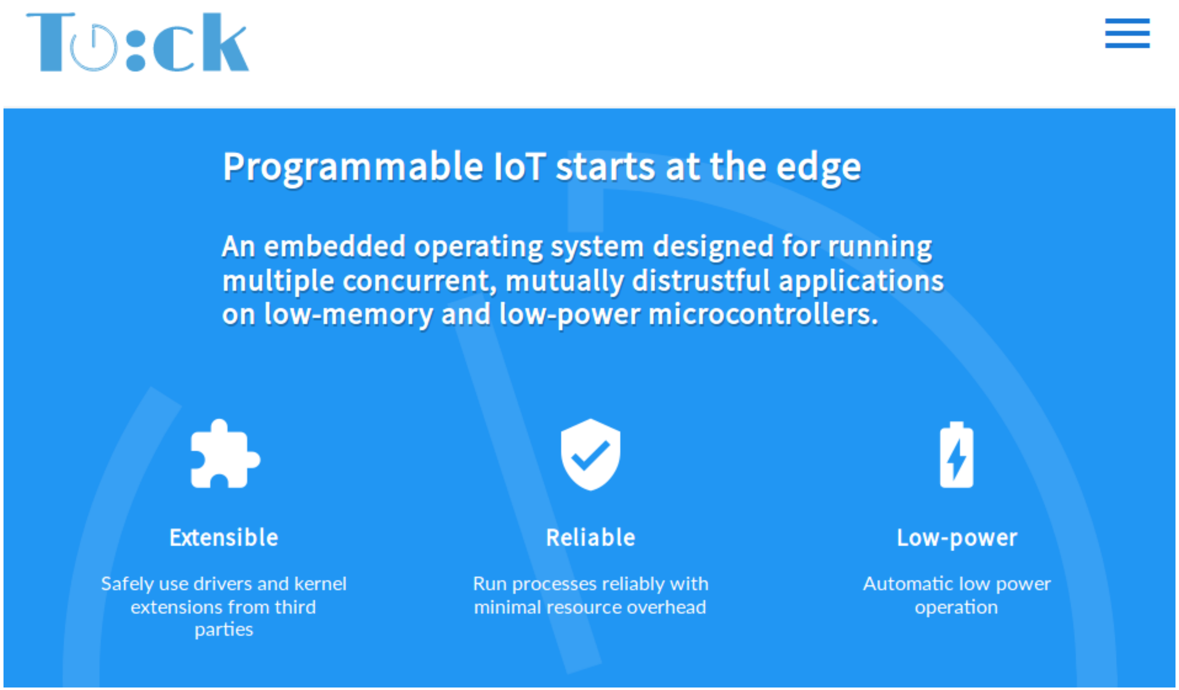
OS Development in Rust

- **Writing an OS in Rust**: Tutorials for basic functionality
- **Redox OS**: Most complete Rust OS, microkernel design



OS Development in Rust

- **Writing an OS in Rust**: Tutorials for basic functionality
- **Redox OS**: Most complete Rust OS, microkernel design
- **Tock**: Operating system for embedded systems



OS Development in Rust

- **Writing an OS in Rust**: Tutorials for basic functionality
- **Redox OS**: Most complete Rust OS, microkernel design
- **Tock**: Operating system for embedded systems
- **Nebulet**: Experimental WebAssembly kernel
    - WebAssembly is a binary format for executable code in web pages
    - Idea: Run wasm applications instead of native binaries
    - Wasm is sandboxed, so it can safely run in kernel address space
    - A bit slower than native code
    - But no expensive context switches or syscalls

OS Development in Rust

- **Writing an OS in Rust**: Tutorials for basic functionality
- **Redox OS**: Most complete Rust OS, microkernel design
- **Tock**: Operating system for embedded systems
- **Nebulet**: Experimental WebAssembly kernel

What does using Rust mean for OS development?

Rust means...

Memory Safety

Memory Safety

- No invalid memory accesses
  - No buffer overflows
  - No dangling pointers
  - No data races
- Guaranteed by Rust's ownership system
  - At compile time

Memory Safety

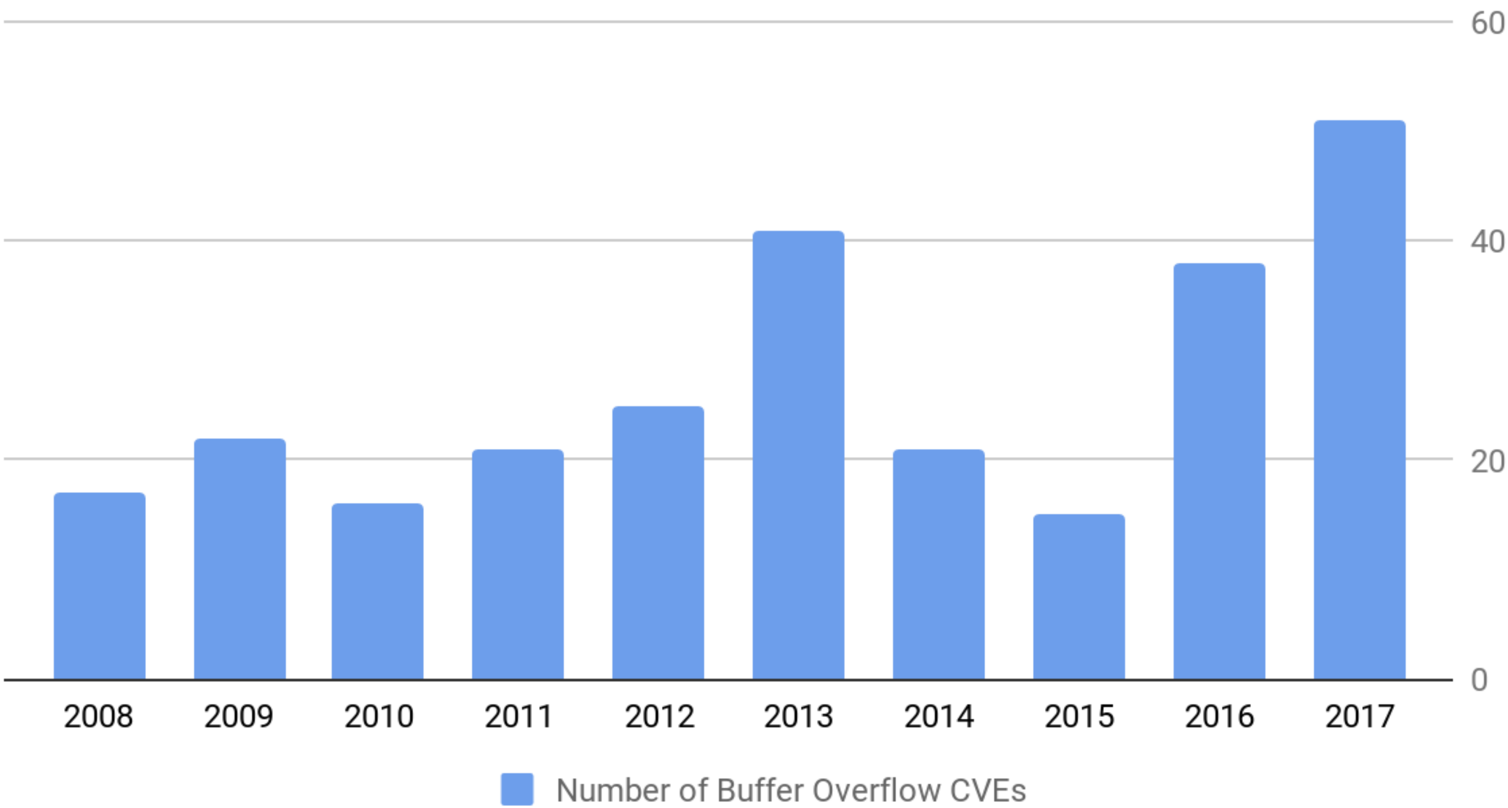
- No invalid memory accesses
  - No buffer overflows
  - No dangling pointers
  - No data races
- Guaranteed by Rust's ownership system
  - At compile time

In C:

- Array capacity is not checked on access
  - Easy to get buffer overflows
- Every `malloc` needs exactly one `free`
  - Easy to get *use-after-free* or *double-free* bugs
- Vulnerabilities caused by memory unsafety are still common

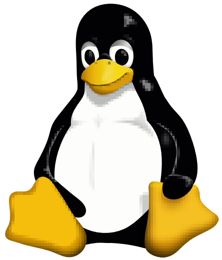
Memory Safety

Buffer Overflow Vulnerabilities in Linux

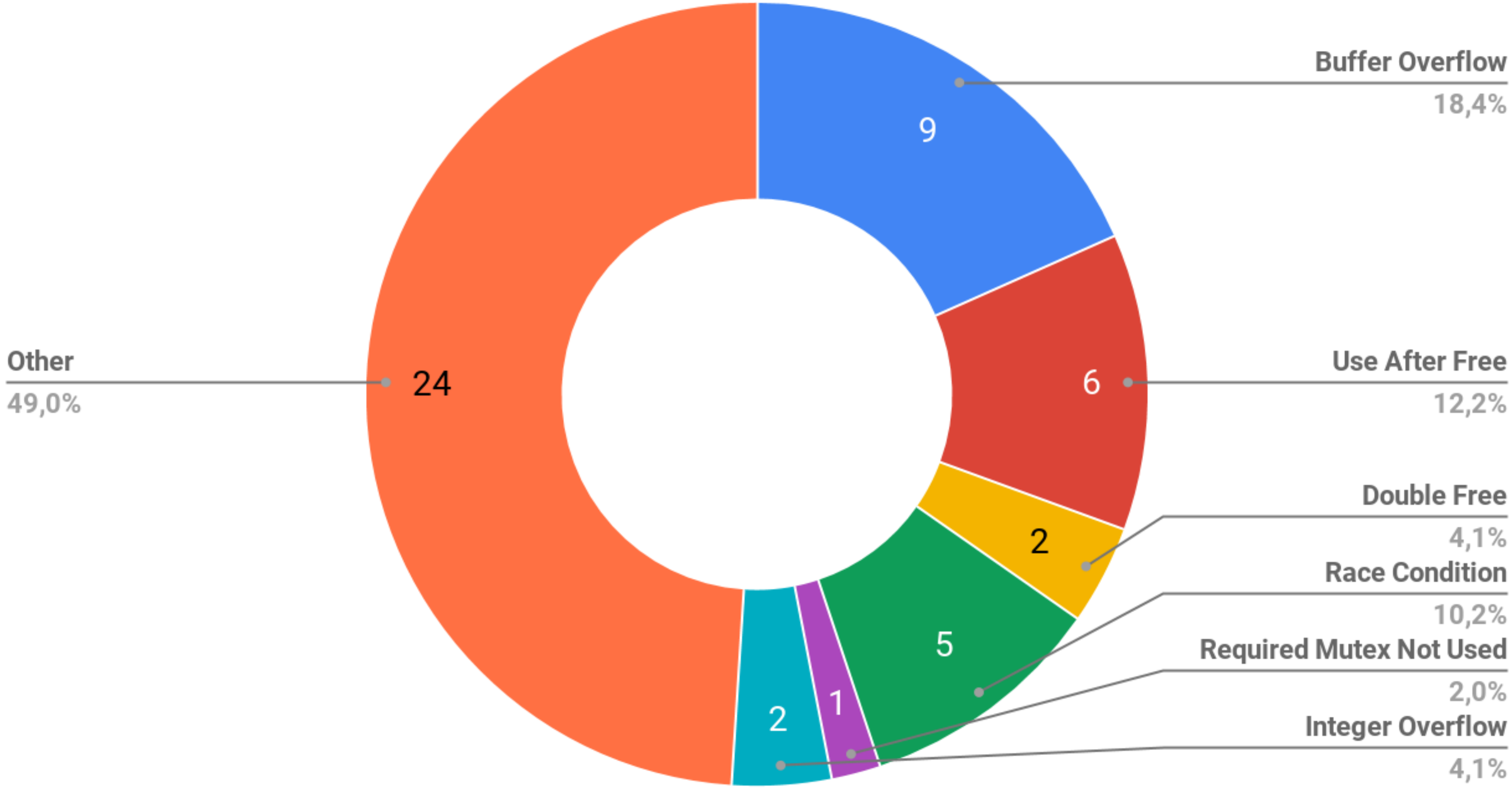


Source: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

Memory Safety



Linux CVEs in 2018 (Jan – Apr)



Source: https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2018/Linux-Linux-Kernel.html

Memory Safety: A Strict Compiler

- It can take some time until your program compiles
- Lifetimes can be complicated
  - “error: `x` does not live long enough”

However:

- “If it compiles, it usually works”
- Far less debugging
  - No data races!
- Refactoring is safe and painless

Encapsulating Unsafety

Encapsulating Unsafety

- Not everything can be verified at compile time
- Sometimes you need `unsafe` in a kernel
  - Writing to the VGA text buffer at `0xb8000`
  - Modifying CPU configuration registers
  - Switching the address space (reloading `CR3`)

Encapsulating Unsafety

- Not everything can be verified at compile time
- Sometimes you need `unsafe` in a kernel
  - Writing to the VGA text buffer at `0xb8000`
  - Modifying CPU configuration registers
  - Switching the address space (reloading `CR3`)

- Rust has `unsafe` blocks that allow to
  - Dereference raw pointers
  - Call `unsafe` functions
  - Access mutable statics
  - Implement `unsafe` traits

- Goal: Provide safe abstractions that encapsulate unsafety
  - Like hardware abstractions in an OS

Encapsulating Unsafety: Example

```rust
/// Read current page table
pub fn read_cr3() -> PhysFrame { … }
/// Load a new page table
pub unsafe fn write_cr3(frame: PhysFrame) { … }

/// Invalidate the TLB completely by reloading the CR3 register.
pub fn flush_tlb() { // safe interface
    let frame = read_cr3();
    unsafe { write_cr3(frame) }
}
```

- The `CR3` register holds the root page table address
  - Reading is safe
  - Writing is `unsafe` (because it changes the address mapping)
- The `flush_tlb` function provides a safe interface
  - It can't be used in an `unsafe` way

A Powerful Type System

A Powerful Type System: Mutexes

C++	Rust

`std::vector data = {1, 2, 3};`
`// mutex is unrelated to data`
`std::mutex mutex;`
 	
`// unsynchronized access possible`
`data.push_back(4);`
 	
`mutex.lock();`
`data.push_back(5);`
`mutex.unlock();`
	`let data = vec![1, 2, 3];`
	`// mutex owns data`
	`let mutex = Mutex::new(data);`
	
	`// compilation error: data was moved`
	`data.push(4);`
	
	`let mut d = mutex.lock().unwrap();`
	`d.push(5);`
	`// released at end of scope`

⇒ Rust ensures that Mutex is locked before accessing data

A Powerful Type System: Page Table Methods

Add a page table mapping:

```
fn map_to<S: PageSize>(
    &mut PageTable,
    page: Page<S>,              // map this page
    frame: PhysFrame<S>,        // to this frame
    flags: PageTableFlags,
) {…}

impl PageSize for Size4KB {…} // standard page
impl PageSize for Size2MB {…} // “huge” 2MB page
impl PageSize for Size1GB {…} // “giant” 1GB page (only on some architectures)
```

A Powerful Type System: Page Table Methods

Add a page table mapping:

```
fn map_to<S: PageSize>(
    &mut PageTable,
    page: Page<S>,               // map this page
    frame: PhysFrame<S>,         // to this frame
    flags: PageTableFlags,
) {…}

impl PageSize for Size4KB {…} // standard page
impl PageSize for Size2MB {…} // “huge” 2MB page
impl PageSize for Size1GB {…} // “giant” 1GB page (only on some architectures)
```

- Generic over the page size
  - 4KB, 2MB or 1GB
- Page and frame must have the same size

A Powerful Type System

Allows to:

- Make misuse impossible
  - Impossible to access data behind a Mutex without locking
- Represent contracts in code instead of documentation
  - Page size of page and frame parameters must match in `map_to`

Everything happens at compile time ⇒ *No run-time cost!*

Easy Dependency Management

Easy Dependency Management

- Over 15000 crates on **crates.io**
- Simply specify the desired version
  - Add single line to `Cargo.toml`
- Cargo takes care of the rest
  - Downloading, building, linking

Easy Dependency Management

- Over 15000 crates on **crates.io**
- Simply specify the desired version
  - Add single line to `Cargo.toml`
- Cargo takes care of the rest
  - Downloading, building, linking

- It works the same for OS kernels
  - Crates need to be `no_std`
  - Useful crates: `bitflags`, `spin`, `arrayvec`, `x86_64`, ...

Easy Dependency Management

- **bitflags**: A macro for generating structures with single-bit flags

```
#[macro_use] extern crate bitflags;

bitflags! {
    pub struct PageTableFlags: u64 {
        const PRESENT =         1 << 0;     // bit 0
        const WRITABLE =        1 << 1;     // bit 1
        const HUGE_PAGE =       1 << 7;     // bit 7
        …
    }
}

fn main() {
    let stack_flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE;
    assert_eq!(stack_flags.bits(), 0b11);
}
```

Easy Dependency Management

- **bitflags**: A macro for generating structures with single-bit flags
- **spin**: Spinning synchronization primitives such as spinlocks

Easy Dependency Management

- **bitflags**: A macro for generating structures with single-bit flags
- **spin**: Spinning synchronization primitives such as spinlocks
- **arrayvec**: Stack-based vectors backed by a fixed sized array

```rust
use arrayvec::ArrayVec;

let mut vec = ArrayVec::<[i32; 16]>::new();
vec.push(1);
vec.push(2);
assert_eq!(vec.len(), 2);
assert_eq!(vec.as_slice(), &[1, 2]);
```

Easy Dependency Management

- **bitflags**: A macro for generating structures with single-bit flags
- **spin**: Spinning synchronization primitives such as spinlocks
- **arrayvec**: Stack-based vectors backed by a fixed sized array
- **x86_64**: Structures, registers, and instructions specific to `x86_64`
    - Control registers
    - I/O ports
    - Page Tables
    - Interrupt Descriptor Tables
    - …

Easy Dependency Management

- **bitflags**: A macro for generating structures with single-bit flags
- **spin**: Spinning synchronization primitives such as spinlocks
- **arrayvec**: Stack-based vectors backed by a fixed sized array
- **x86_64**: Structures, registers, and instructions specific to `x86_64`

- Over 350 crates in the no_std category
  - Many more can be trivially made `no_std`

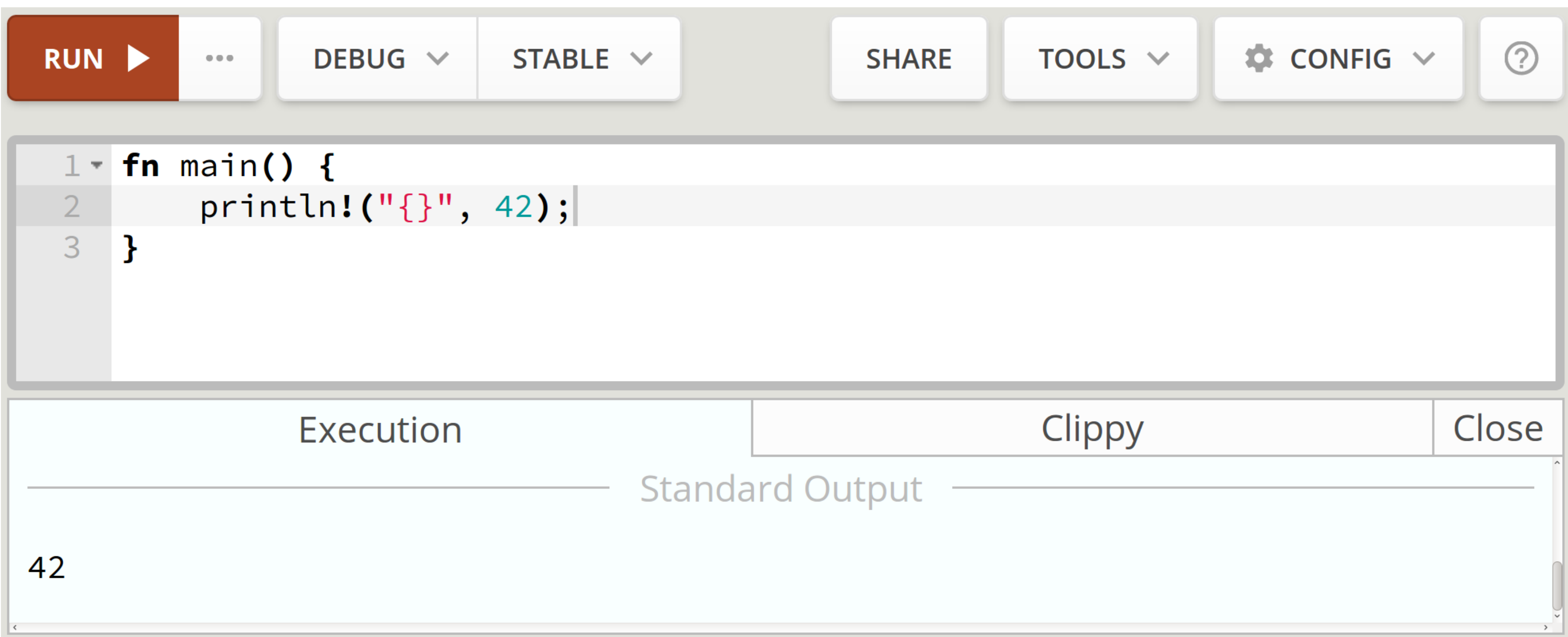
Great Tooling

Great Tooling

- **rustup**: Use multiple Rust versions for different directories
- **cargo**: Automatically download, build, and link dependencies
- **rustfmt**: Format Rust code according to style guidelines

Great Tooling

- **rustup**: Use multiple Rust versions for different directories
- **cargo**: Automatically download, build, and link dependencies
- **rustfmt**: Format Rust code according to style guidelines
- **Rust Playground**: Run and share code snippets in your browser



Great Tooling

- **clippy**: Additional warnings for dangerous or unidiomatic code

```
fn equal(x: f32, y: f32) -> bool {
    if x == y { true } else { false }
}
```

Great Tooling

- **clippy**: Additional warnings for dangerous or unidiomatic code

```rust
fn equal(x: f32, y: f32) -> bool {
    if x == y { true } else { false }
}
```

```
error: strict comparison of f32 or f64
 --> src/main.rs:2:8
  |
2 | if x == y { true } else { false }
  |    ^^^^^^ help: consider comparing them within some error: (x - y).abs() < err

warning: this if-then-else expression returns a bool literal
 --> src/main.rs:2:5
  |
2 | if x == y { true } else { false }
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ help: you can reduce it to: x == y
```

Great Tooling

- **proptest**: A property testing framework

```
fn parse_date(s: &str) -> Option<(u32, u32, u32)> {
    // […] check if valid YYYY-MM-DD format
    let year = &s[0..4];
    let month = &s[6..7]; // BUG: should be 5..7
    let day = &s[8..10];
    convert_to_u32(year, month, date)
}
proptest! {
    #[test]
    fn parse_date(y in 0u32..10000, m in 1u32..13, d in 1u32..32) {
        let date_str = format!("{:04}-{:02}-{:02}", y, m, d);
        let (y2, m2, d2) = parse_date(&date_str).unwrap();
        prop_assert_eq!((y, m, d), (y2, m2, d2));
    }
}
```

```
- try random values                        y = 2497, m = 8, d = 27     passes
    |                                      y = 9641, m = 8, d = 18     passes
    | (failed test case found)             y = 7360, m = 12, d = 20     fails

- reduce y to find simpler case            y = 3680, m = 12, d = 20    fails
    |                                      y = 1840, m = 12, d = 20    fails
    |                                      y = 920, m = 12, d = 20     fails
    |                                      y = 460, m = 12, d = 20     fails
    |                                      y = 230, m = 12, d = 20     fails
    |                                      y = 115, m = 12, d = 20     fails
    |                                      y = 57, m = 12, d = 20      fails
    |                                      y = 28, m = 12, d = 20      fails
    |                                      y = 14, m = 12, d = 20      fails
    |                                      y = 7, m = 12, d = 20       fails
    |                                      y = 3, m = 12, d = 20       fails
    |                                      y = 1, m = 12, d = 20       fails
    | (simplest y case still fails)        y = 0, m = 12, d = 20        fails

- reduce m to find simpler case            y = 0, m = 6, d = 20        passes
    |                                      y = 0, m = 9, d = 20        passes
    |                                      y = 0, m = 11, d = 20       fails
    | (minimum failure value found)        y = 0, m = 10, d = 20        fails

- reduce d to find simpler case            y = 0, m = 10, d = 10       fails
    |                                      y = 0, m = 10, d = 5        fails
    |                                      y = 0, m = 10, d = 3        fails
    |                                      y = 0, m = 10, d = 2        fails
    | (reduced test case found)            y = 0, m = 10, d = 1         fails
```

See https://github.com/altsysrq/proptest

Great Tooling for OS Development

In C:

- First step is to build a **cross compiler**
  - A `gcc` that compiles for a bare-metal target system
  - Lots of build dependencies
- On Windows, you have to use **cygwin**
  - Required for using the GNU build tools (e.g. `make`)
  - The *Windows Subsystem for Linux* might also work

In Rust:

- Rust works natively on Linux, Windows, and macOS
- The Rust compiler `rustc` is already a cross-compiler
- For linking, we can use the cross-platform **`lld`** linker
  - By the LLVM project

Great Tooling for OS Development

bootimage: Create a bootable disk image from a Rust kernel

- Cross-platform, no C dependencies
- Automatically downloads and compiles a bootloader
- **bootloader**: A x86 bootloader written in Rust and inline assembly

Goals:

- Make building your kernel as easy as possible
- Let beginners dive immediately into OS programming
  - No hours-long toolchain setup
- Remove platform-specific differences
  - You shouldn't need Linux to do OS development

See https://os.phil-opp.com/news/2018-03-09-pure-rust/

Great Tooling for OS Development

In development: **bootimage test**

- Basic integration test framework
- Runs each test executable in an isolated QEMU instance
  - Tests are completely independent
  - Results are reported through the serial port
- Allows testing in target environment

Testing on real hardware?

An Awesome Community

An Awesome Community

- Code of Conduct from the beginning
  - “We are committed to providing a **friendly, safe and welcoming environment** for all […]”
  - “We will exclude you from interaction if you insult, demean or harass anyone”
  - Followed on GitHub, IRC, the Rust subreddit, etc.

An Awesome Community

- Code of Conduct from the beginning
  - “We are committed to providing a **friendly, safe and welcoming environment** for all [...]”
  - “We will exclude you from interaction if you insult, demean or harass anyone”
  - Followed on GitHub, IRC, the Rust subreddit, etc.
- It works!
  - No inappropriate comments for “Writing an OS in Rust” so far
  - Focused technical discussions

An Awesome Community

- Code of Conduct from the beginning
    - "We are committed to providing a **friendly, safe and welcoming environment** for all [...]"
    - "We will exclude you from interaction if you insult, demean or harass anyone"
    - Followed on GitHub, IRC, the Rust subreddit, etc.
- It works!
    - No inappropriate comments for "Writing an OS in Rust" so far
    - Focused technical discussions

vs:

> "So this patch is utter and absolute garbage, and should be shot in the head and buried very very deep." Linus Torvalds on 14 Aug 2017

No Elitism

No Elitism

Typical elitism in OS development:

> “A **decade of programming**, including a few years of low-level coding in assembly language and/or a systems language such as C, is **pretty much the minimum necessary** to even understand the topic well enough to work in it.”

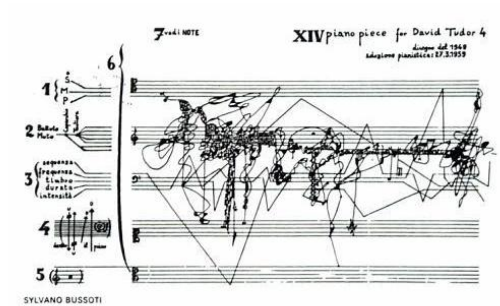
From wiki.osdev.org/Beginner_Mistakes

Most Rust Projects:

- It doesn't matter where you come from
  - C, C++, Java, Python, JavaScript, …
- It's fine to ask questions
  - People are happy to help

No Elitism

- **IntermezzOS**: “People who have not done low-level programming before are a specific target of this book“



intermezz**OS**, (a little OS)

- **Writing an OS in Rust**: Deliberately no particular target audience
  - People are able to decide themselves
  - Provide links for things not explained on the blog
    - E.g. for advanced Rust and OS concepts

Exciting New Features

Exciting New Features

- **Impl Trait**: Return closures from functions
- **Non-Lexical Lifetimes**: A more intelligent borrow checker
- **WebAssembly**: Run Rust in browsers

Exciting New Features

- **Impl Trait**: Return closures from functions
- **Non-Lexical Lifetimes**: A more intelligent borrow checker
- **WebAssembly**: Run Rust in browsers

In development: **Futures and async / await**

- Simple and fast asynchronous code

- How does it work?
- What does it mean for OS development?

Futures

Result of an asynchronous computation:

```
trait Future {
    type Item;
    type Error;
    fn poll(&mut self, cx: &mut Context) -> Result<Async<Self::Item>,
                                                   Self::Error>;
}

enum Async<T> {
    Ready(T),
    NotReady,
}
```

- Instead of blocking, `Async::NotReady` is returned

Futures: Implementation Details

- Futures do nothing until polled
- An `Executor` is used for polling multiple futures until completion
  - Like a scheduler
- If future is not ready when polled, a `Waker` is created
  - Notifies the `Executor` when the future becomes ready
  - Avoids continuous polling

Combinators

- Transform a future without polling it (similar to iterators)
- Examples
  - `future.map(|v| v + 1)`: Applies a function to the result
  - `future_a.join(future_b)`: Wait for both futures
  - `future.and_then(|v| some_future(v))`: Chain dependent futures

Async / Await

Traditional synchronous code:

```
fn get_user_from_database(user_id: u64) -> Result<User> {…}

fn handle_request(request: Request) -> Result<Response> {
    let user = get_user_from_database(request.user_id)?;
    generate_response(user)
}
```

- Thread blocked until database read finished
  - Complete thread stack unusable
- Number of threads limits number of concurrent requests

Async / Await

Asynchronous variant:

```
async fn get_user_from_database(user_id: u64) -> Result<User> {…}

async fn handle_request(request: Request) -> Result<Response> {
    let future = get_user_from_database(request.user_id);
    let user = await!(future)?;
    generate_response(user)
}
```

- Async functions return `Future<Item=T>` instead of `T`
- No blocking occurs
  - Stack can be reused for handling other requests
- Thousands of concurrent requests possible

How does `await` work?

Async / Await: Generators

- Functions that can suspend themselves via `yield`:

```
fn main() {
    let mut generator = || {
        println!("2");
        yield;
        println!("4");
    };

    println!("1");
    unsafe { generator.resume() };
    println!("3");
    unsafe { generator.resume() };
    println!("5");
}
```

Async / Await: Generators

- Functions that can suspend themselves via `yield`:

```
fn main() {
    let mut generator = || {
        println!("2");
        yield;
        println!("4");
    };

    println!("1");
    unsafe { generator.resume() };
    println!("3");
    unsafe { generator.resume() };
    println!("5");
}
```

- Compiled as state machines

```rust
let mut generator = {
    enum Generator { Start, Yield1, Done, }

    impl Generator {
        unsafe fn resume(&mut self) {
            match self {
                Generator::Start => {
                    println!("2");
                    *self = Generator::Yield1;
                }
                Generator::Yield1 => {
                    println!("4");
                    *self = Generator::Done;
                }
                Generator::Done => panic!("generator resumed after completion")
            }
        }
    }
    Generator::Start
};
```

Async / Await: Generators

- Generators can keep state:

```
fn main() {
    let mut generator = || {
        let number = 42;
        let ret = "foo";

        yield number; // yield can return values
        return ret
    };

    unsafe { generator.resume() };
    unsafe { generator.resume() };
}
```

Where are `number` and `ret` stored between `resume` calls?

```rust
let mut generator = {
    enum Generator {
        Start(i32, &'static str),
        Yield1(&'static str),
        Done,
    }
    impl Generator {
        unsafe fn resume(&mut self) -> GeneratorState<i32, &'static str> {
            match self {
                Generator::Start(i, s) => {
                    *self = Generator::Yield1(s); GeneratorState::Yielded(i)
                }
                Generator::Yield1(s) => {
                    *self = Generator::Done; GeneratorState::Complete(s)
                }
                Generator::Done => panic!("generator resumed after completion")
            }
        }
    }
    Generator::Start(42, "foo")
};
```

Async / Await: Implementation

```
async fn handle_request(request: Request) -> Result<Response> {
    let future = get_user_from_database(request.user_id);
    let user = await!(future)?;
    generate_response(user)
}
```

Compiles roughly to:

```
async fn handle_request(request: Request) -> Result<Response> { GenFuture(|| {
    let future = get_user_from_database(request.user_id);
    let user = loop { match future.poll() {
        Ok(Async::Ready(u)) => break Ok(u),
        Err(e) => break Err(e),
        Ok(Async::NotReady) => yield,
    }}?;
    generate_response(user)
})}
```

Async / Await: Implementation

Transform Generator into Future:

```
struct GenFuture<T>(T);

impl<T: Generator> Future for GenFuture<T> {
    fn poll(&mut self, cx: &mut Context) -> Result<Async<T::Item>, T::Error> {
        match unsafe { self.0.resume() } {
            GeneratorStatus::Complete(Ok(result)) => Ok(Async::Ready(result)),
            GeneratorStatus::Complete(Err(e)) => Err(e),
            GeneratorStatus::Yielded => Ok(Async::NotReady),
        }
    }
}
```

Async / Await: For OS Development?

- Everything happens at compile time
  - Can be used in OS kernels and on embedded devices
- Makes asynchronous code simpler

Use Case: **Cooperative multitasking**

- Yield when waiting for I/O
- Executor then polls another future
- Interrupt handler notifies `Waker`
- Only a single thread is needed
  - Devices with limited memory

Async / Await: An OS Without Blocking?

A blocking thread makes its whole stack unusable

- Makes threads heavy-weight
- Limits the number of threads in the system

What if blocking was not allowed?

Async / Await: An OS Without Blocking?

A blocking thread makes its whole stack unusable

- Makes threads heavy-weight
- Limits the number of threads in the system

What if blocking was not allowed?

- Threads would return futures instead of blocking
- Scheduler would schedule futures instead of threads
- Stacks could be reused for different threads

- Only a few stacks are needed for many, many futures
- Task-based instead of thread-based concurrency
    - Fine grained concurrency at the OS level

Summary

Rust means:

- **Memory Safety** no overflows, no invalid pointers, no data races
- **Encapsulating Unsafety** creating safe interfaces
- **A Powerful Type System** make misuse impossible

- **Easy Dependency Management** cargo, crates.io
- **Great Tooling** clippy, proptest, bootimage

- **An Awesome Community** code of conduct
- **No Elitism** asking questions is fine, no minimum requirements

- **Exciting New Features** futures, async / await

Slides are available at https://os.phil-opp.com/talks

Summary

Rust means:

- **Memory Safety** no overflows, no invalid pointers, no data races
- **Encapsulating Unsafety** creating safe interfaces
- **A Powerful Type System** make misuse impossible

- **Easy Dependency Management** cargo, crates.io
- **Great Tooling** clippy, proptest, bootimage

- **An Awesome Community** code of conduct
- **No Elitism** asking questions is fine, no minimum requirements

- **Exciting New Features** futures, async / await

Slides are available at https://os.phil-opp.com/talks

Extra Slides

Encapsulating Unsafety

Not possible in all cases:

```
/// Write a new root table address into the CR3 register.
pub fn write_cr3(page_table_frame: PhysFrame, flags: Cr3Flags) {
    let addr = page_table_frame.start_address();
    let value = addr.as_u64() | flags.bits();
    unsafe { asm!("mov $0, %cr3" :: "r" (value) : "memory"); }
}
```

Encapsulating Unsafety

Not possible in all cases:

```
/// Write a new root table address into the CR3 register.
pub fn write_cr3(page_table_frame: PhysFrame, flags: Cr3Flags) {
    let addr = page_table_frame.start_address();
    let value = addr.as_u64() | flags.bits();
    unsafe { asm!("mov $0, %cr3" :: "r" (value) : "memory"); }
}
```

Problem: Passing an invalid `PhysFrame` could break memory safety!

- A frame that is no page table
- A page table that maps all pages to the same frame
- A page table that maps two random pages to the same frame

Encapsulating Unsafety

Not possible in all cases:

```
/// Write a new root table address into the CR3 register.
pub unsafe fn write_cr3(page_table_frame: PhysFrame, flags: Cr3Flags) {
    let addr = page_table_frame.start_address();
    let value = addr.as_u64() | flags.bits();
    asm!("mov $0, %cr3" :: "r" (value) : "memory");
}
```

Problem: Passing an invalid `PhysFrame` could break memory safety!

- A frame that is no page table
- A page table that maps all pages to the same frame
- A page table that maps two random pages to the same frame

⇒ Function needs to be `unsafe` because it depends on valid input

Encapsulating Unsafety

Edge Cases: Functions that...

- ... disable paging?

Encapsulating Unsafety

Edge Cases: Functions that...

- ... disable paging? `unsafe`
- ... disable CPU interrupts?

Encapsulating Unsafety

Edge Cases: Functions that…

- … disable paging? `unsafe`
- … disable CPU interrupts? `safe`
- … might cause CPU exceptions?

Encapsulating Unsafety

Edge Cases: Functions that…

- … disable paging? `unsafe`
- … disable CPU interrupts? `safe`
- … might cause CPU exceptions? `safe`
- … can be only called from privileged mode?

Encapsulating Unsafety

Edge Cases: Functions that…

- … disable paging? `unsafe`
- … disable CPU interrupts? `safe`
- … might cause CPU exceptions? `safe`
- … can be only called from privileged mode? `safe`
- … assume certain things about the hardware?
  - E.g. there is a VGA text buffer at `0xb8000`

Encapsulating Unsafety

Edge Cases: Functions that...

- ... disable paging? `unsafe`
- ... disable CPU interrupts? `safe`
- ... might cause CPU exceptions? `safe`
- ... can be only called from privileged mode? `safe`
- ... assume certain things about the hardware? `depends`
    - E.g. there is a VGA text buffer at `0xb8000`

Async / Await: Generators

- Why is `resume` unsafe?

```
fn main() {
    let mut generator = move || {
        let foo = 42;
        let bar = &foo;
        yield;
        return bar
    };
    unsafe { generator.resume() };
    let heap_generator = Box::new(generator);
    unsafe { heap_generator.resume() };
}
```

```
enum Generator {
    Start,
    Yield1(i32, &i32),
    Done,
}
```

Async / Await: Generators

- Why is `resume` unsafe?

```rust
fn main() {
    let mut generator = move || {
        let foo = 42;
        let bar = &foo;
        yield;
        return bar
    };
    unsafe { generator.resume() };
    let heap_generator = Box::new(generator);
    unsafe { heap_generator.resume() };
}
```

```rust
enum Generator {
    Start,
    Yield1(i32, &i32),
    Done,
}
```

- Generator contains reference to itself
  - No longer valid when moved to the heap ⇒ undefined behavior
  - Must not be moved after first `resume`

Await: Just Syntactic Sugar?

Is `await` just syntactic sugar for the `and_then` combinator?

```
async fn handle_request(request: Request) -> Result<Response> {
    let user = await!(get_user_from_database(request.user_id))?;
    generate_response(user)
}
```

```
async fn handle_request(request: Request) -> Result<Response> {
    get_user_from_database(request.user_id).and_then(|user| {
        generate_response(user)
    })
}
```

In this case, both variants work.

Await: Not Just Syntactic Sugar!

```
fn read_info_buf(socket: &mut Socket) -> [u8; 1024]
    -> impl Future<Item = [0; 1024], Error = io::Error> + 'static
{
    let mut buf = [0; 1024];
    let mut cursor = 0;
    while cursor < 1024 {
        cursor += await!(socket.read(&mut buf[cursor..]))?;
    };
    buf
}
```

- We don't know how many `and_then` we need
  - But each one is their own type -> boxed trait objects required
- `buf` is a local stack variable, but the returned future is `'static`
  - Not possible with `and_then`
  - *Pinned types* allow it for `await`