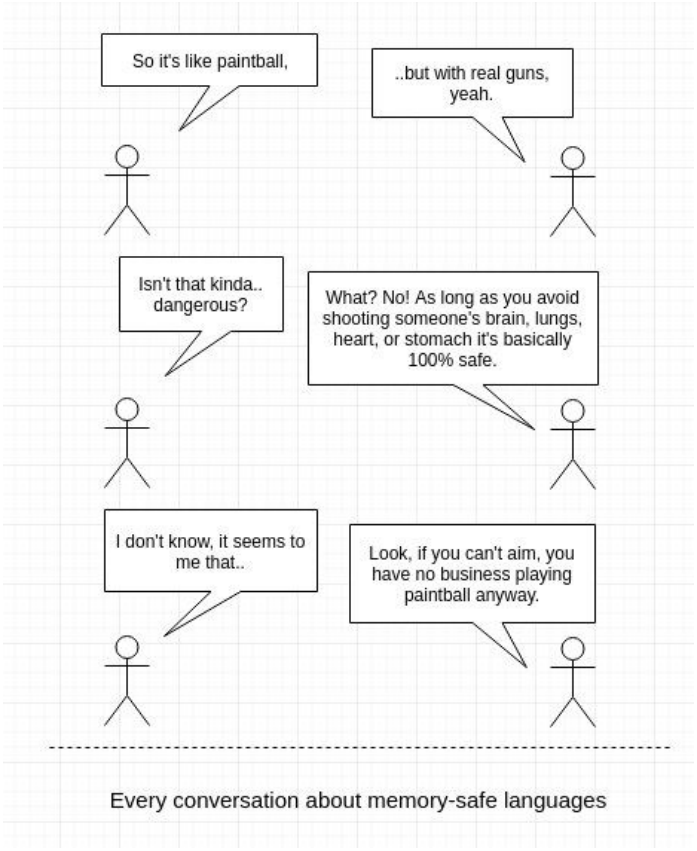
KernelModule  ust>

Recap: Was machen wir?

- Erste Idee: Ein beliebiges Kernel-Modul in Rust schreiben
- Zweite Idee: ext2 nach Rust portieren
- Finale Idee: ramfs in Rust nachimplementieren

- Vereinfachende Annahmen:
  - Wir unterstützen nur Systeme mit MMU
  - Wir unterstützen nur x86_64

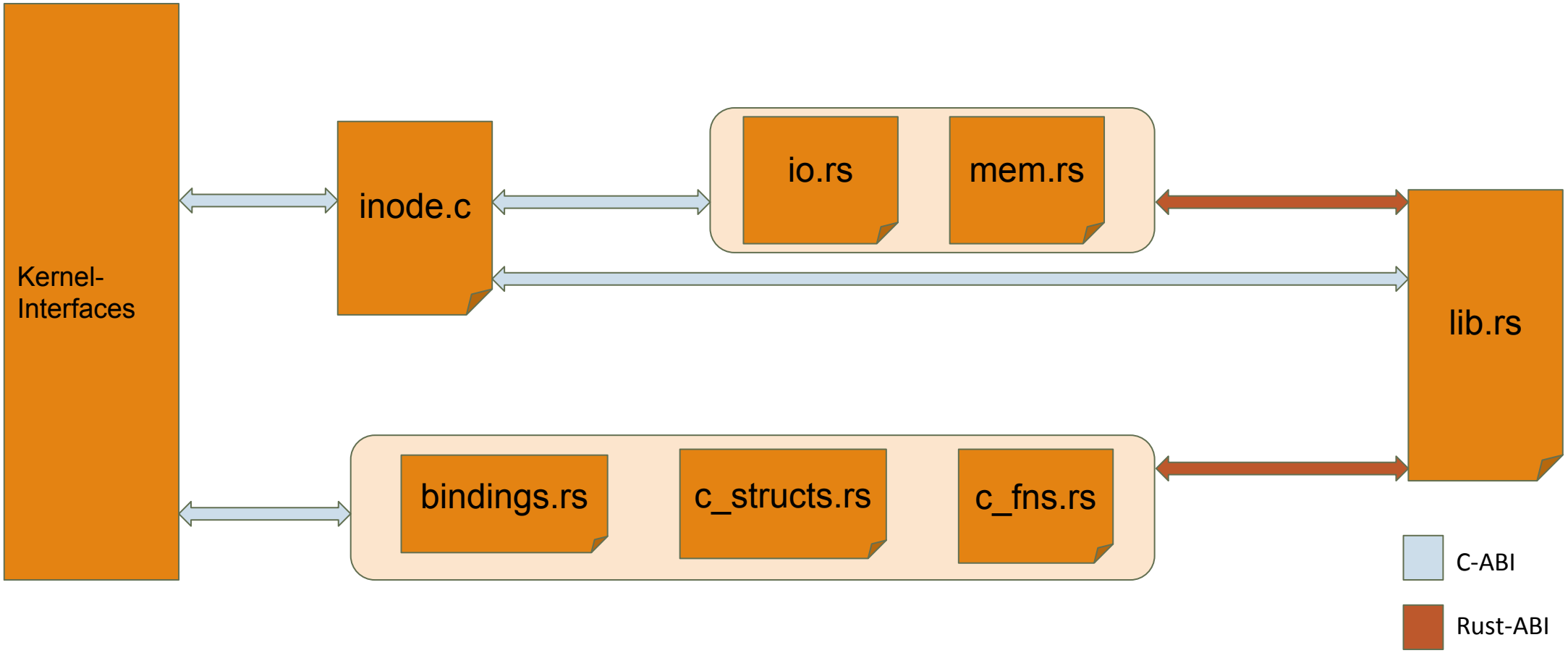
Motivation



- C ist zwar sicherer und einfacher als Assembly, lässt aber immer noch viel Raum für Speicherfehler
- Rust ist im Gegensatz zu z.B. Go überhaupt eine Option, weil es keine Runtime benötigt
- Safety Checks in Rust sind statisch zur Compile-Zeit
  → kein Performanceverlust

Bildquelle: https://fasterthanli.me/series/reading-files-the-hard-way/part-1

Architektur



Grundlagen: Speicher und IO

- Speicherverwaltung in Rust normalerweise aus libc
  ⟶ Heap-allozierte Typen wie Box in no_std-Umgebung nicht ohne weiteres möglich
- Gebündelte Crate alloc erlaubt Definition eines GlobalAllocator
  ⟶ alloc() und dealloc() auf krealloc() und kfree() abbilden
- IO: print! Makro nutzt auch libc ⟶ Makro selbst mit kprintf() neu definieren
- IO und Speicherverwaltung konnten wir aus vorherigen Rust-Kernelmodulen übernehmen

Rust-Wrapper um C-Funktionen

- Aufruf von C-Funktionen per FFI ist unsafe
  ⟶ Rust-Compiler kann Sicherheit von fremdem Code nicht garantieren
- Wir wissen, dass Kernel-Interfaces (bei richtiger Benutzung) sicher sind
- Safe Wrappers um unsafen Code schreiben, die nur sichere Benutzung erlauben

Vorgehensweise

1. Wrapper-Funktion mit Namenskonvention (z.B. “rs_<Funktionsname>”) anlegen
2. unsafen Aufruf der Original-Funktion einfügen
3. Return-Type anpassen mit
   a. Result, falls ursprünglicher Return-Type ein Integer ist
   b. Option, falls ursprünglicher Return-Type ein Pointer ist
   c. Nichts, falls die Funktion keinen Wert zurückgibt
4. Original-Rückgabewert entsprechend Schritt 3 wrappen

Unsere Beobachtungen bzw. Methoden beruhen auf Standard-C-Semantik. Es gibt keine Garantie, dass Funktionen Fehler genau so darstellen!

Automatisierte Generierung

- Erstellen von Funktionswrappern weitgehend “mechanisch”
- Wenn wir Elemente einer Funktionsdeklaration (Name, Parameter, Return-Type) auslesen können ⟶ automatisierbar
- Wir haben dazu das Tool “wrapgen” geschrieben
  - nutzbar als Standalone-Tool
  - oder im Rust-Buildprozess einbindbar
- Noch sehr frühe Version ⟶ Contributions erwünscht
- Nicht perfekt: Kann keine Funktionssemantik erkennen, würde z.B. Read- und Write-Funktionen falsch wrappen

Rust-Wrapper um C-Structs

- Problem: Zugriff auf raw pointer ist fast immer unsafe
- Erste Lösungsidee: Eigenen Trait definieren, der *mut c_type zu c_type dereferenziert
  - Nicht umsetzbar, weil es Ownership verletzt
  - Außerdem: C-Interface benötigen wieder Pointer
- Neuer Ansatz: “Leichtgewichtige” Wrapper-Structs
  - leichtgewichtig, weil zu wrappender Pointer als einziges Feld
  - Zugriff über associated functions

Trait: Sammlung von Funktionen, die ein Struct implementieren soll (vergleichbar mit Interfaces in OOP)

Vorgehensweise

1. Initiales Struct mit from_ptr() und get_ptr() erstellen
2. Besondere Konstruktoren erstellen
3. Accessor-Funktionen für Felder schreiben
4. Funktionen mit Pointer als Argument zu Associated Functions umschreiben
5. Eigene Erweiterungen als Extension Trait implementieren

Beispiel: Inode - Schritt 1

1. Initiales Struct mit from_ptr() und get_ptr() erstellen
2. Besondere Konstruktoren erstellen
3. Accessor-Funktionen für Felder schreiben
4. Funktionen mit Pointer als Argument zu Associated Functions umschreiben
5. Eigene Erweiterungen als Extension Trait implementieren

```
struct Inode {
    ptr: *mut inode,
}

impl Inode {
    pub fn from_ptr(ptr: *mut inode) → Self {
        Self { ptr }
    }

    pub fn get_ptr(&self) → *mut inode {
        self.ptr
    }
}
```

Beispiel: Inode - Schritt 2

1. Initiales Struct mit from_ptr() und get_ptr() erstellen
2. Besondere Konstruktoren erstellen
3. Accessor-Funktionen für Felder schreiben
4. Funktionen mit Pointer als Argument zu Associated Functions umschreiben
5. Eigene Erweiterungen als Extension Trait implementieren

```
impl Inode {
    pub fn new(sb: SuperBlock) → Self {
        Self::from_ptr(rs_new_inode(sb))
    }

    pub fn null() → Self {
        Self { ptr: core::ptr::null_mut() }
    }
}
```

Iterativer Prozess: Wenn neue Wrapper geschrieben werden, alte Stellen anpassen

Beispiel: Inode - Schritt 3

1. Initiales Struct mit from_ptr() und get_ptr() erstellen
2. Besondere Konstruktoren erstellen
3. Accessor-Funktionen für Felder schreiben
4. Funktionen mit Pointer als Argument zu Associated Functions umschreiben
5. Eigene Erweiterungen als Extension Trait implementieren

```
impl Inode {
    pub fn set_ino(&self) {
        unsafe {
        (*self.ptr).i_ino = get_next_ino().into()
        }
    }

    pub fn get_sb(self) → SuperBlock {
        SuperBlock::from_ptr_unchecked(
            unsafe { (*self.ptr).i_sb }
        )
    }
}
```

Beispiel: Inode - Schritt 4

1. Initiales Struct mit from_ptr() und get_ptr() erstellen
2. Besondere Konstruktoren erstellen
3. Accessor-Funktionen für Felder schreiben
4. Funktionen mit Pointer als Argument zu Associated Functions umschreiben
5. Eigene Erweiterungen als Extension Trait implementieren

```
impl Inode {
    pub fn inc_nlink(&self) {
        unsafe { inc_nlink(self.ptr) }
    }
}
```

Beispiel: Inode - Schritt 5

1. Initiales Struct mit from_ptr() und get_ptr() erstellen
2. Besondere Konstruktoren erstellen
3. Accessor-Funktionen für Felder schreiben
4. Funktionen mit Pointer als Argument zu Associated Functions umschreiben
5. Eigene Erweiterungen als Extension Trait implementieren

```
pub trait RamfsInodeOps {
    fn ramfs_set_inode_ops(&self);
    fn is_in_debug_mode(&self) → bool;
}

impl RamfsInodeOps for Inode {
    fn ramfs_set_inode_ops(&self) {
        unsafe { ramfs_set_inode_ops(self.ptr) }
    }

    fn is_in_debug_mode(&self) → bool {
        self.get_sb().is_in_debug_mode()
    }
}
```

Beispiel: Option Parsing - C

```c
static int ramfs_parse_options(char *data, struct
ramfs_mount_opts *opts)
{
	substring_t args[MAX_OPT_ARGS];
	int option;
	int token;
	char *p;

	opts->mode = RAMFS_DEFAULT_MODE;

	while ((p = strsep(&data, ",")) != NULL) {
		if (!*p)
			continue;
		token = match_token(p, tokens, args);
		switch (token) {
		case Opt_mode:
			if (match_octal(&args[0], &option))
				return -EINVAL;
			opts->mode = option & S_IALLUGO;
			break;
		}
	}

	return 0;
}
```

Überprüfung der Optionen

Aufsplitten des Strings

Modus setzen

Beispiel: Option Parsing - Rust

```
fn ramfs_parse_options(
    data: &str,
    opts: &mut RamfsMountOpts,
) {
    for substr in data.split_terminator(","){
        match substr{
            _ if substr.starts_with("mode=") ⇒ opts.mode =
                                parse_octal(substr.split_at(substr.find("=").unwrap()).1).unwrap(),
            "debug" ⇒ opts.debug = true,
            _ ⇒ {}
        }
    }
}
```

Aufsplitten des Strings

Überprüfung der Optionen

Modus setzen

zusätzlicher debug-Modus

Können wir rsramfs fest in den Kernel kompilieren?

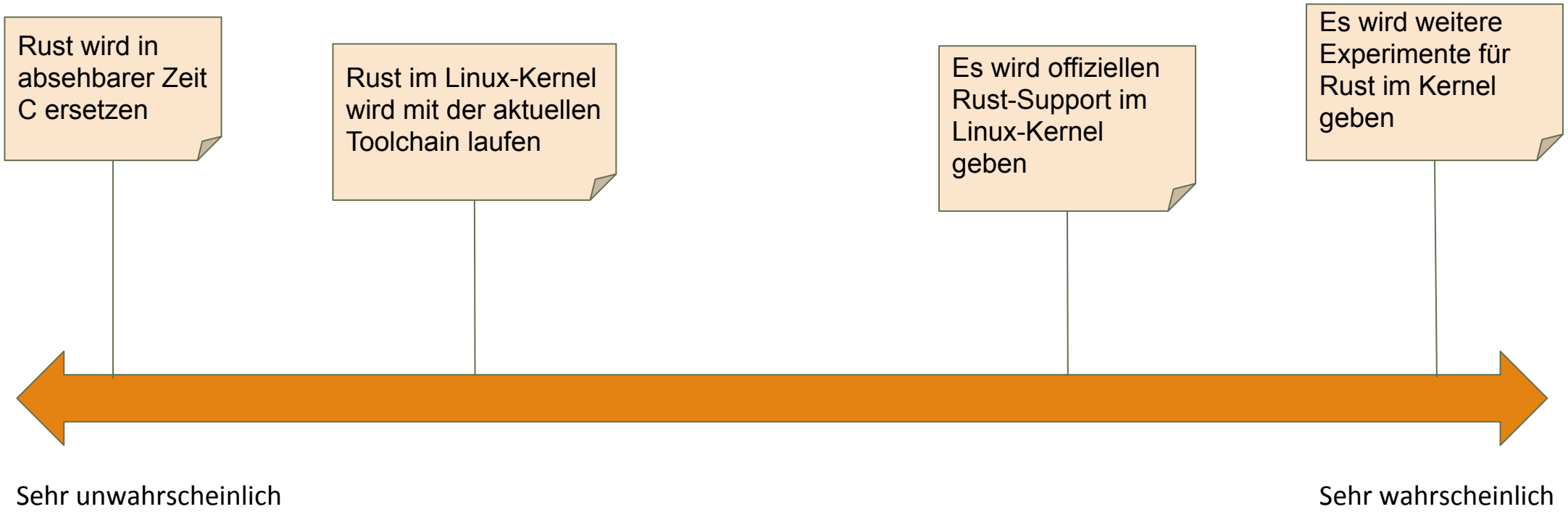
- Testumgebung: WSL 2 → Kernel-Image lässt sich einfach über Config File ändern
- Anpassungen im Build-Prozess sowie in der Initialisierung notwendig
  - Kein eigener Build-Ordner
  - Als obj-y statt obj-m kompilieren
  - Initialisierung nicht mehr als Modul, sondern mit fs_initcall
- Aktuell noch sehr “hacky”
  - Doppelte Symbole in lib/string.o wegen Rust-Crate compiler_builtins
- Aber: Experimentell läuft es stabil

Lohnt sich ein Kernel-Projekt in Rust?

+ Rust-Code allgemein einfacher für Entwickler mit wenig Kernel-Erfahrung zu verstehen
+ Memory Safety erlaubt mehr Vertrauen in eigenen Code

- Keine einheitlichen Bindings → Deutlich mehr Zeit zum Portieren von Interfaces als Implementierung unseres Moduls
- Noch an vielen Stellen Workarounds notwendig
- Größtenteils “terra incognita” → noch weniger Informationen online als zu Kernel-Entwicklung in C

Ausblick


Rust wird in absehbarer Zeit C ersetzen
Rust im Linux-Kernel wird mit der aktuellen Toolchain laufen
Es wird offiziellen Rust-Support im Linux-Kernel geben
Es wird weitere Experimente für Rust im Kernel geben
Sehr unwahrscheinlich
Sehr wahrscheinlich

Was könnten wir in Zukunft machen?

- Ein komplexeres Dateisystem portieren/selbst bauen
- Treiber für andere Geräte (z.B. spezielle HID-Geräte) portieren
- Wrapper für komplexere Kernel-Features (z.B. Mutexe)
- Die Kernel-Bindings in eine eigene Crate auslagern
- Uns weiter mit Testing befassen
  - Aktuell ein “Integration Test” via Bash-Script
  - Können wir z.B. KUnit mit Rust verwenden?
  - Können wir eine CI für Kernel-Module aufsetzen