

Betriebssystementwicklung in Rust

oder

wie ich lernte das Rust-Typsystem zu lieben

Flo Rommel

Rust-Meetup

09.07.2019

"Rust is a systems programming language"

– rust-lang.org (bis vor kurzem)

Was ist eine Systemprogrammiersprache?

→ Eine Sprache um Systemsoftware zu schreiben ¹

Systemsoftware ↔ **Anwendungssoftware**

Beispiele:

- Betriebssystem(-kernel), Gerätetreiber
- Dienstprogramme, Middleware
- Browser-Engine
- ...

¹ https://en.wikipedia.org/wiki/System_programming_language

Agenda



- Besonderheiten des Rust-Typsystems
- Betriebssystementwicklung in Rust
 - Grundlegendes
 - Das Betriebssystem StuBS
 - RuStuBS: StuBS in Rust
- Ausblick

Move-Semantik



```
let s = String::from("hello");  
let t = s;
```

```
println!("{}", t);  
println!("{}", s);
```



```
fn myfunction(x: String) {  
    // nothing here  
}
```

...

```
let s = String::from("hello");
```

```
myfunction(s);
```

```
println!("{}", s);
```



Aber:

```
let s = 35;  
let t = s;
```

```
println!("{}", t);  
println!("{}", s);
```



References & Lifetimes



```
let r;  
  
let s = String::from("hello");  
r = &s;  
  
println!("{}", r);
```



Für jede Variable kann der Compiler eine statische Laufzeit ermitteln.

Referenzen leben immer nur so lange wie das referenzierte Objekt.

```
let r;  
  
{  
    let s = String::from("hello");  
    r = &s;  
}  
  
println!("{}", r);
```



References & Mutability



```
let mut s = String::from("hello");  
  
let r1 = &s;  
let r2 = &s;  
  
println!("r1: {}, r2: {}", r1, r2);
```



```
let mut s = String::from("hello");  
  
let r1 = &mut s;  
let r2 = &mut s;  
  
my_function(r1, r2);
```



```
let mut s = String::from("hello");  
  
let r1 = &s;  
let r2 = &mut s;  
  
my_function2(r1, r2);
```



Regel:

Entweder

beliebig viele immutable Referenzen


oder

eine mutable Referenz


Unsafe Code



```
let r;  
  
{  
    let s = String::from("hello");  
    r = &s;  
}  
  
println!("{}", r);
```



```
let r;  
  
{  
    let s = String::from("hello");  
    r = &s as *const String;  
}  
  
unsafe {  
    println!("{}", *r);  
}
```



Ausgabe:

hP>☒



Undefiniertes Verhalten!



Betriebssystementwicklung in Rust

Grundlegendes



Hier: **Betriebssystem := Kernel**

→ Unterste Softwareschicht
stellt grundlegende Abstraktionen bereit

Aufgaben eines Betriebssystemkernels:

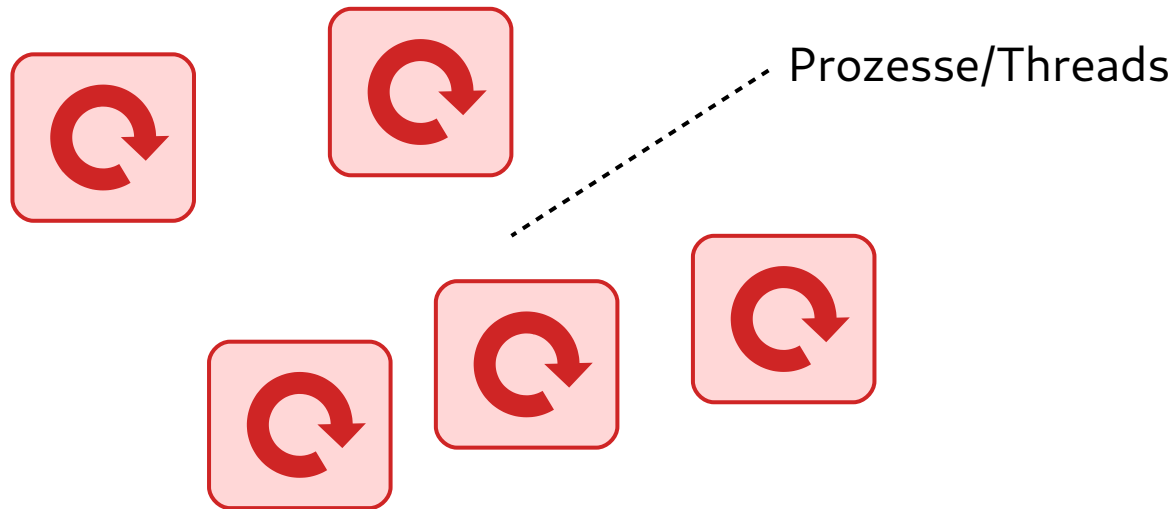
- Prozessverwaltung
- Speicherverwaltung
- Geräteverwaltung (Treiber)
- Dateisysteme

Beispiel: Prozessverwaltung

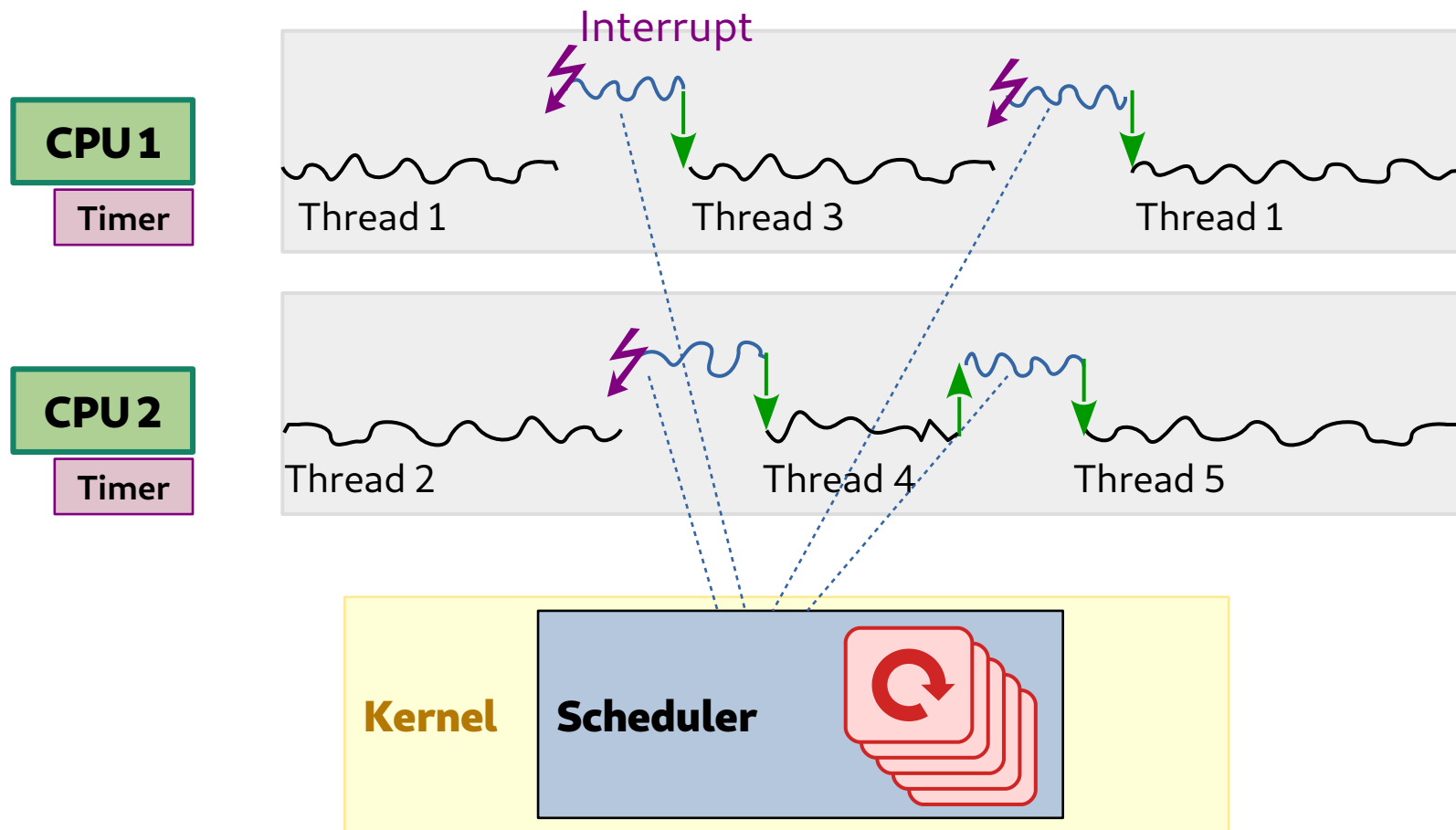


CPU 1

CPU 2



Beispiel: Prozessverwaltung



Das Betriebssystem StuBS



(OO)StuBS: (Objektorientiertes) Studenten-Betriebssystem

Lehr-Betriebssystem-Kernel in C++ für x86

- Präemptives Multitasking
- SMP-Support [\[MPStuBS\]](#)
- Isolation (Kernelspace ↔ Userspace) mit Adressraumverwaltung
→ Paging, Prozesse [\[StuBSml\]](#)
- Einfache Treiber für IO (Bildschirm CGA/VGA, Tastatur)



- Implementierung von StuBS in Rust → **RuStuBS** (2015)
- Ziel: Ausnutzung von Rusts Typssystems bei der Betriebssystementwicklung



Sichere, maschinenunabhängige Module

Modul **machine**
(Treiber/Hardwareinteraktion)
maschinenabhängig

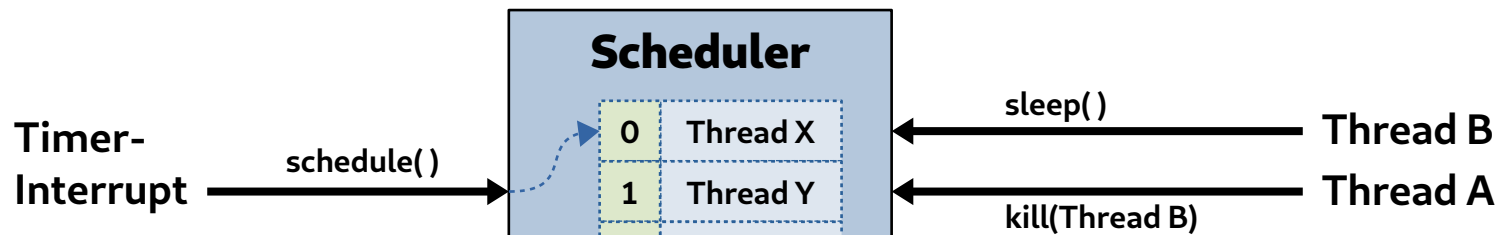
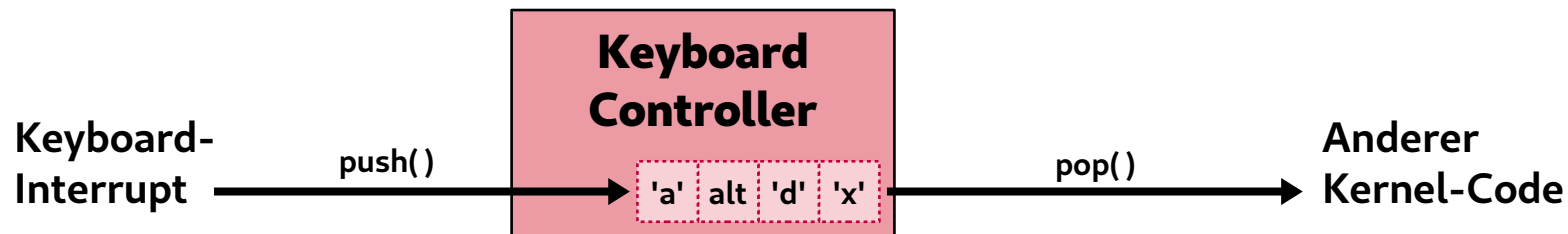
Modul **guard**
(Synchronisation)
maschinenunabhängig

unsafe

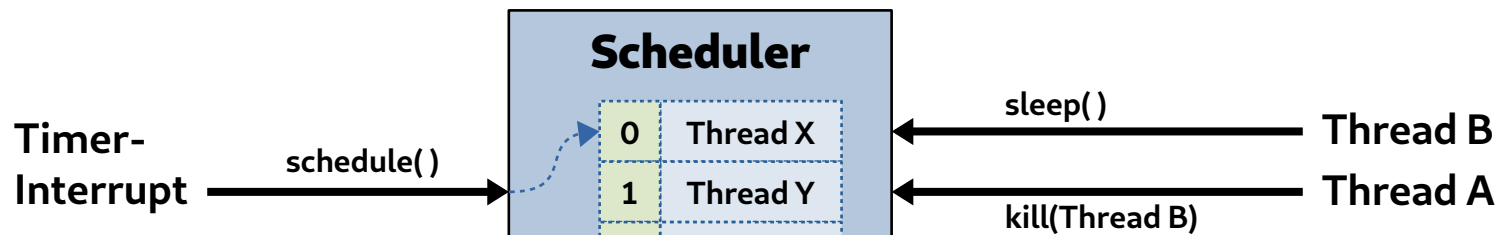
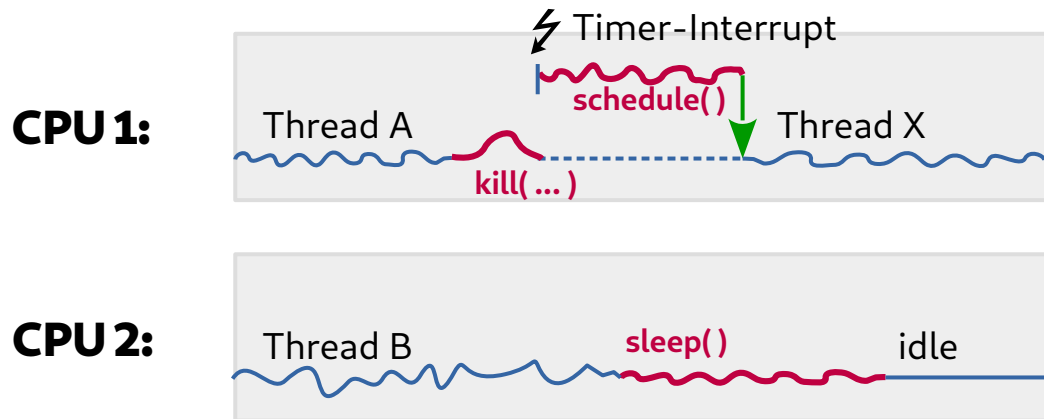


Beispiel: Synchronisation

Geteilte Systemressourcen in StuBS



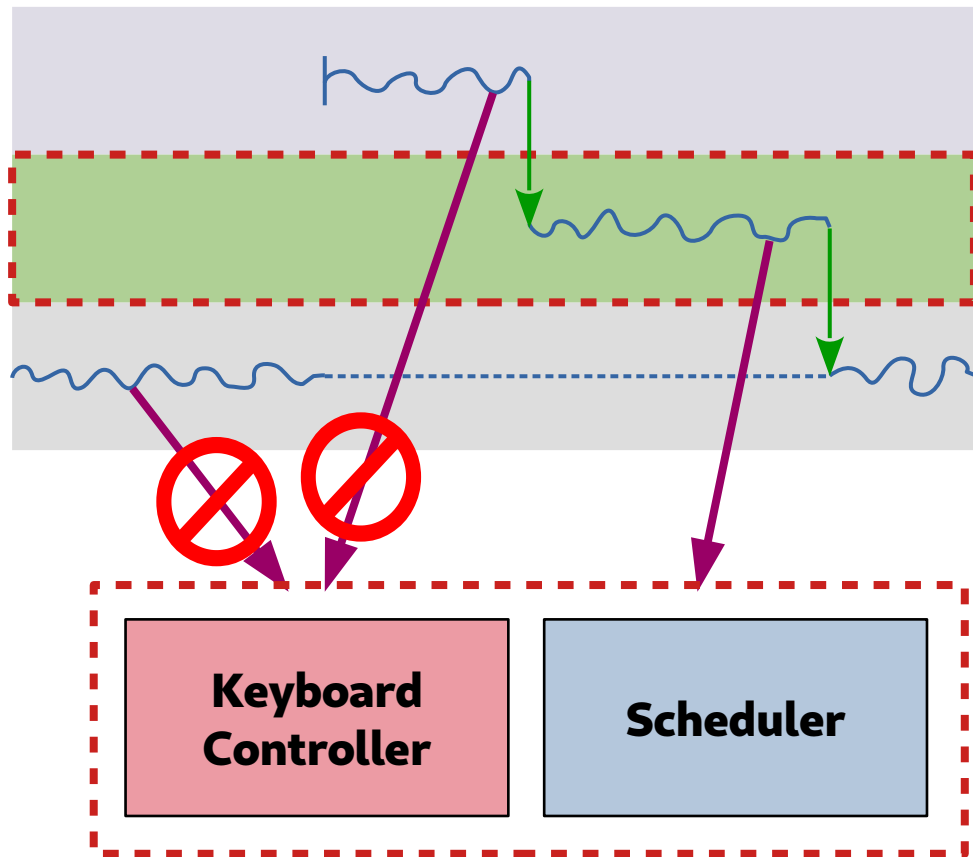
Geteilte Systemressourcen in StuBS



Prioritätsebenen



Big Kernel Lock



Interrupts (Prolog)

Geschützte Ebene (Epilog)

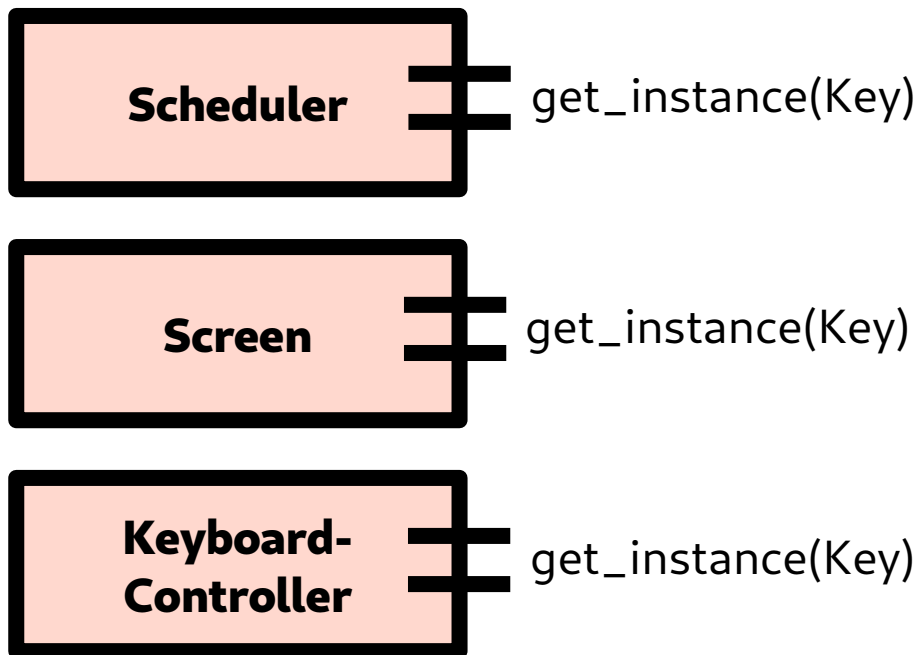
Threads (normaler Code)

Immer nur eine einzige CPU ist zu einem Zeitpunkt in der geschützten Ebene.

→ Zutrittsfunktionen

`Guard::enter()`, `Guard::exit()`

Prioritätsebenen in Rust – Idee



Prioritätsebenen in Rust – Code



```
pub struct Guard {
    phantom: ()
}

pub struct GuardKey {
    phantom: ()
}

impl Guard {
    pub fn lock(self) -> GuardKey {
        unsafe {
            guarded_env().enter();
        }
        GuardKey::new()
    }
}

impl Drop for GuardKey {
    fn drop(&mut self) {
        unsafe {
            guarded_env().leave();
        }
    }
}
```

```
fn some_function(...) {
    // ...

    let second_key;

    {
        let key = guard.lock();

        let sched =
            Scheduler::get_instance(&key);

        second_key = guard.lock();

        // key is dropped here
    }

    // ...
}
```



Ausblick

Typestate Pattern¹



Verallgemeinerung des Prinzips:

Informationen über den Laufzeit-Zustand werden durch das Typsystem zur Compilezeit codiert

Einfaches Beispiel aus der Standardbibliothek:

```
let file = std::fs::File::open("myfile.txt"?;

// use file

drop(file);

// cannot use file or any references to file here anymore
```

¹ <http://cliffle.com/blog/rust-typestate/>

Typestate Pattern¹



Verallgemeinerung des Prinzips:

Informationen über den Laufzeit-Zustand werden durch das Typsystem zur Compilezeit codiert

Weitere Beispiele:

- RAII-Guards (std): Mutex, RwLock, RefCell
- Serde (siehe ¹)

¹ <http://cliffle.com/blog/rust-typestate/>



Rust bietet durch sein Typssystem die Möglichkeit der Implementierung von sicheren und statisch überprüften Zustandsübergängen

Voraussetzungen dafür sind:

- Destruktoren & deterministische Zerstörung von Objekten
- Move-Semantik
(Einhaltung wird vom Compiler sichergestellt)
- Statisch geprüfte Lebenszeiten von Variablen