RuStuBS

Rust in der Betriebssystemlehre

2024-03-15

Paul Bergmann

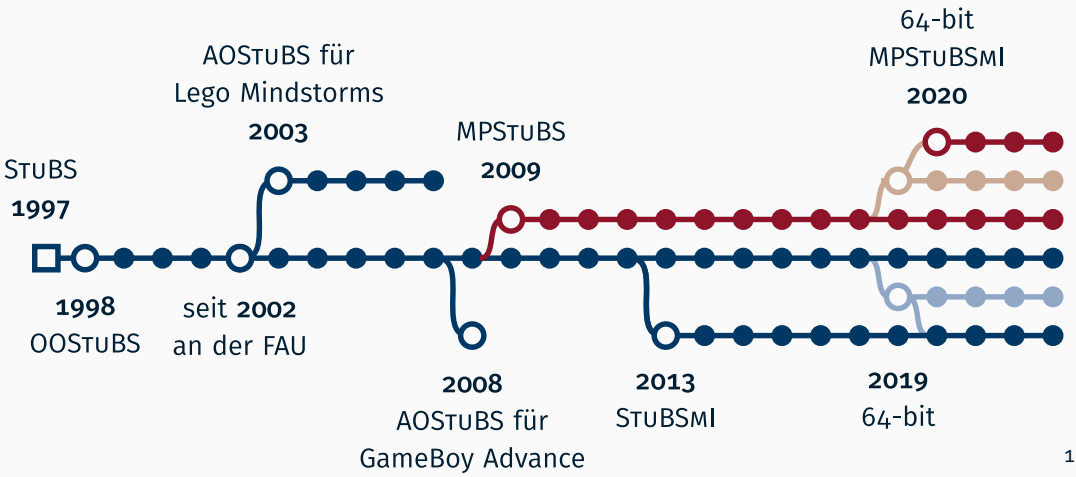
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

Friedrich-Alexander-Universität
Technische Fakultät

Einleitung

Was ist StuBS



[1]

[1] Bernhard Heinloth. ***Übung zu Betriebssysteme - Organisation.***

Was ist Rust

Eine seit 2010 entwickelte, systemnahe Programmiersprache

> *Rust's rich type system and ownership model guarantee memory-safety and thread-safety* [2]

Unterteilung in *sichere* und *unsichere* Programmteile

[2] Rust Foundation. ***The Rust Programming Language.*** URL: `https://www.rust-lang.org/` (besucht am 14.02.2024).

Zielsetzung

- MPStuBS-Aufgaben 1-6 lösen

Zielsetzung

- MPStuBS-Aufgaben 1-6 lösen
  - Interruptbehandlung
  - Softwareinterrupts
  - Präemptives Multithreading
  - Passives Warten

Zielsetzung

- MPStuBS-Aufgaben 1-6 lösen
  - Interruptbehandlung
  - Softwareinterrupts
  - Präemptives Multithreading
  - Passives Warten
- *Ähnlich genug* zu C++ StuBS um an Lehre anzuknüpfen

Zielsetzung

- MPStuBS-Aufgaben 1-6 lösen
  - Interruptbehandlung
  - Softwareinterrupts
  - Präemptives Multithreading
  - Passives Warten
- *Ähnlich genug* zu C++ StuBS um an Lehre anzuknüpfen
- Wenig `unsafe` Code

Übersicht

1. Einleitung
2. Übersetzungsprozess
3. Typische Studentische Fehler
4. Fazit

Übersetzungsprozess

Viel Code lässt sich direkt übersetzen

```
bool Watch::windup(uint32_t us) {
    uint64_t tmp_ticks = (LAPIC::TICKS_PER_MS * us) / 1000;
    uint32_t tmp32_ticks = static_cast<uint32_t>(tmp_ticks);

    // Calculate LAPIC timer divider
    for (int divider = 1; divider <= 128; divider *= 2) {
        // Check for overflow
        if (tmp_ticks == static_cast<uint64_t>(tmp32_ticks)) {
            // ...
            return true;
        }
        tmp_ticks /= 2;
        tmp32_ticks = static_cast<uint32_t>(tmp_ticks);
    }

    return false;
}
```

Viel Code lässt sich direkt übersetzen

```
bool Watch::windup(uint32_t us) {
    uint64_t tmp_ticks = (LAPIC::TICKS_PER_MS * us) / 1000;
    uint32_t tmp32_ticks = static_cast<uint32_t>(tmp_ticks);

    // Calculate LAPIC timer divider
    for (int divider = 1; divider <= 128; divider *= 2) {
        // Check for overflow
        if (tmp_ticks == static_cast<uint64_t>(tmp32_ticks)) {
            // ...
            return true;
        }
        tmp_ticks /= 2;
        tmp32_ticks = static_cast<uint32_t>(tmp_ticks);
    }

    return false;
}
```

Viel Code lässt sich direkt übersetzen

```
bool Watch::windup(uint32_t us) {
    uint64_t tmp_ticks = (LAPIC::TICKS_PER_MS * us) / 1000;
    uint32_t tmp32_ticks = static_cast<uint32_t>(tmp_ticks);

    // Calculate LAPIC timer divider
    for (int divider = 1; divider <= 128; divider *= 2) {
        // Check for overflow
        if (tmp_ticks == static_cast<uint64_t>(tmp32_ticks)) {
            // ...
            return true;
        }
        tmp_ticks /= 2;
        tmp32_ticks = static_cast<uint32_t>(tmp_ticks);
    }

    return false;
}
```

Viel Code lässt sich direkt übersetzen

```rust
pub fn windup(period: Duration) -> Result<(), WindupError> {
    let ticks_ms = timer::ticks_per_ms() as u128;
    let mut tmp_ticks = (ticks_ms * period.as_micros()) / 1000;

    let mut divider_u8 = 1u8;
    while let Some(divider) = timer::Divider::try_get(divider_u8) {
        // Check for overflow
        if let Ok(tmp_ticks_u32) = u32::try_from(tmp_ticks) {
            // ...
            return Ok(());
        }
        tmp_ticks /= 2;
        divider_u8 *= 2;
    }

    Err(WindupError::PeriodTooLarge)
}
```

Viel Code lässt sich direkt übersetzen

```rust
pub fn windup(period: Duration) -> Result<(), WindupError> {
    let ticks_ms = timer::ticks_per_ms() as u128;
    let mut tmp_ticks = (ticks_ms * period.as_micros()) / 1000;

    let mut divider_u8 = 1u8;
    while let Some(divider) = timer::Divider::try_get(divider_u8) {
        // Check for overflow
        if let Ok(tmp_ticks_u32) = u32::try_from(tmp_ticks) {
            // ...
            return Ok(());
        }
        tmp_ticks /= 2;
        divider_u8 *= 2;
    }

    Err(WindupError::PeriodTooLarge)
}
```

Viel Code lässt sich direkt übersetzen

```rust
pub fn windup(period: Duration) -> Result<(), WindupError> {
    let ticks_ms = timer::ticks_per_ms() as u128;
    let mut tmp_ticks = (ticks_ms * period.as_micros()) / 1000;

    let mut divider_u8 = 1u8;
    while let Some(divider) = timer::Divider::try_get(divider_u8) {
        // Check for overflow
        if let Ok(tmp_ticks_u32) = u32::try_from(tmp_ticks) {
            // ...
            return Ok(());
        }
        tmp_ticks /= 2;
        divider_u8 *= 2;
    }

    Err(WindupError::PeriodTooLarge)
}
```

Manche Konstrukte lassen sich nicht einfach übersetzen

Interruptbehandlung in C++ StuBS

Manche Konstrukte lassen sich nicht einfach übersetzen

Interruptbehandlung in C++ StuBS

```
class Gate { virtual void trigger(); };
class Keyboard : Gate {
    char buffer;
    void trigger() overwrite { buffer = fetch_keyboard(); }
};
```

Manche Konstrukte lassen sich nicht einfach übersetzen

Interruptbehandlung in C++ StuBS

```
class Gate { virtual void trigger(); };
class Keyboard : Gate {
    char buffer;
    void trigger() overwrite { buffer = fetch_keyboard(); }
};

static Gate* plugbox[NUM_VECTORS];

void assign(uint8_t vector, Gate* g) { plugbox[vector] = g; }
Gate* report(uint8_t vector) { return plugbox[vector]; }
```

Manche Konstrukte lassen sich nicht einfach übersetzen

Interruptbehandlung in C++ StuBS

```
class Gate { virtual void trigger(); };
class Keyboard : Gate {
    char buffer;
    void trigger() overwrite { buffer = fetch_keyboard(); }
};

static Gate* plugbox[NUM_VECTORS];

void assign(uint8_t vector, Gate* g) { plugbox[vector] = g; }
Gate* report(uint8_t vector) { return plugbox[vector]; }

void handle(uint8_t vector) { report(vector)->trigger(); }
```

Lesen und Schreiben von globalem Zustand

```
static Gate* plugbox[NUM_VECTORS];

void assign(uint8_t vector, Gate* g) { plugbox[vector] = g; }
Gate* report(uint8_t vector) { return plugbox[vector]; }
```

In Rust so nicht möglich!

Lesen und Schreiben von globalem Zustand

```
static Gate* plugbox[NUM_VECTORS];

void assign(uint8_t vector, Gate* g) { plugbox[vector] = g; }
Gate* report(uint8_t vector) { return plugbox[vector]; }
```

In Rust so nicht möglich! Stattdessen:

- Schreiben während Initialisierungsphase

Lesen und Schreiben von globalem Zustand

```
static Gate* plugbox[NUM_VECTORS];

void assign(uint8_t vector, Gate* g) { plugbox[vector] = g; }
Gate* report(uint8_t vector) { return plugbox[vector]; }
```

In Rust so nicht möglich! Stattdessen:

- Schreiben während Initialisierungsphase
- Danach Read-Only

Lesen und Schreiben von globalem Zustand

```
static Gate* plugbox[NUM_VECTORS];

void assign(uint8_t vector, Gate* g) { plugbox[vector] = g; }
Gate* report(uint8_t vector) { return plugbox[vector]; }
```

In Rust so nicht möglich! Stattdessen:

- Schreiben während Initialisierungsphase
- Danach Read-Only
- Eigenschaft **statisch** bekannt machen

Schreibender Zugriff auf Treiberobjekt

```
class Gate { virtual void trigger(); };
class Keyboard : Gate {
    char buffer;
    void trigger() overwrite { buffer = fetch_keyboard(); }
};

void handle(uint8_t vector) { report(vector)->trigger(); }
```

Was passiert bei parallel ausgelösten Interrupts?

Schreibender Zugriff auf Treiberobjekt

```
class Gate { virtual void trigger(); };
class Keyboard : Gate {
    char buffer;
    void trigger() overwrite { buffer = fetch_keyboard(); }
};

void handle(uint8_t vector) { report(vector)->trigger(); }
```

Was passiert bei parallel ausgelösten Interrupts?

- Mehrere `&mut Keyboard` auf gleiches Objekt (**UB** in Rust)

Schreibender Zugriff auf Treiberobjekt

```
class Gate { virtual void trigger(); };
class Keyboard : Gate {
    char buffer;
    void trigger() overwrite { buffer = fetch_keyboard(); }
};

void handle(uint8_t vector) { report(vector)->trigger(); }
```

Was passiert bei parallel ausgelösten Interrupts?

- Mehrere `&mut Keyboard` auf gleiches Objekt (**UB** in Rust)
- Geräteabhängig ob parallel oder nicht

Schreibender Zugriff auf Treiberobjekt

```
class Gate { virtual void trigger(); };
class Keyboard : Gate {
    char buffer;
    void trigger() overwrite { buffer = fetch_keyboard(); }
};

void handle(uint8_t vector) { report(vector)->trigger(); }
```

Was passiert bei parallel ausgelösten Interrupts?

- Mehrere `&mut Keyboard` auf gleiches Objekt (**UB** in Rust)
- Geräteabhängig ob parallel oder nicht
- Treiber kümmern sich stattdessen selber um **interior mutability**

Verwendung von `unsafe`

- `#[forbid(unsafe_code)]` wird nichts

Verwendung von `unsafe`

- `#[forbid(unsafe_code)]` wird nichts
- 12.5% aller Expressions sind `unsafe`

Verwendung von `unsafe`

- `#[forbid(unsafe_code)]` wird nichts
- 12.5% aller Expressions sind `unsafe`
- Treibercode (`machine`) liegt leicht darüber mit 16%

Verwendung von `unsafe`

- `#[forbid(unsafe_code)]` wird nichts
- 12.5% aller Expressions sind `unsafe`
- Treibercode (`machine`) liegt leicht darüber mit 16%
- Verglichen mit 100% in C++ ;)

Verwendung von `unsafe`

- `#[forbid(unsafe_code)]` wird nichts
- 12.5% aller Expressions sind `unsafe`
- Treibercode (`machine`) liegt leicht darüber mit 16%
- Verglichen mit 100% in C++ ;)
- Minimierung führt manchmal zu Abstraktionsschachtelung
  z.B. `Bootstrapped<PerCore<RefCell<Option<Window>>>>`

Typische Studentische Fehler

Rust ist defensiv

```
int foo = *ptr;
asm volatile("mov %0, %%cr3" : : "r"(pml4_addr));
int bar = *ptr;
```

Rust ist defensiv

```
int foo = *ptr;
int bar = *ptr;
asm volatile("mov %0, %%cr3" : : "r"(pml4_addr));
```

Rust ist defensiv

```
int foo = *ptr;
asm volatile("mov %0, %%cr3" : : "r"(pml4_addr) : "memory");
int bar = *ptr;
```

Rust ist defensiv

```
int foo = *ptr;
asm volatile("mov %0, %%cr3" : : "r"(pml4_addr) : "memory");
int bar = *ptr;
```

Rust macht standardmäßig pessimistischere Annahmen

Rust ist defensiv

```
int foo = *ptr;
asm volatile("mov %0, %%cr3" : : "r"(pml4_addr) : "memory");
int bar = *ptr;
```

Rust macht standardmäßig pessimistischere Annahmen:

- nomem versus memory Clobber bei Inline Assembly

Rust ist defensiv

```
int foo = *ptr;
asm volatile("mov %0, %%cr3" : : "r"(pml4_addr) : "memory");
int bar = *ptr;
```

Rust macht standardmäßig pessimistischere Annahmen:

- nomem versus memory Clobber bei Inline Assembly
- Bounds-Checks bei Arrayzugriffen

Rust ist defensiv

```
int foo = *ptr;
asm volatile("mov %0, %%cr3" : : "r"(pml4_addr) : "memory");
int bar = *ptr;
```

Rust macht standardmäßig pessimistischere Annahmen:

- nomem versus memory Clobber bei Inline Assembly
- Bounds-Checks bei Arrayzugriffen
- Overflow-Checks bei Arithmetik (in Debugbuilds)

1. Fehlerquelle

```
void Dispatcher::dispatch(Thread* next) {
    Thread* current = active();
    setActive(next);
    current->resume(next);
}
```

1. Fehlerquelle: nullptr

```
void Dispatcher::dispatch(Thread* next) {
    Thread* current = active();

    assert(current != nullptr);
    assert(next != nullptr);

    setActive(next);
    current->resume(next);
}
```

1. Fehlerquelle: `nullptr`

```rust
mod dispatcher {
    static ACTIVE: PerCore<Cell<Option<&'static ThreadControlBlock>>> = ...;

    fn dispatch(next: &'static ThreadControlBlock) {
        let current = ACTIVE
            .get()
            .expect("dispatch called when no thread was previously dispatched");

        ACTIVE.set(Some(next));
        current.resume(next);
    }
}
```

2. Fehlerquelle

```
void Keyboard::plugin() {
    Plugbox::assign(Vector::KEYBOARD, this);
}

void main() {
    Keyboard{}.plugin();
    Core::Interrupt::enable();
    while (true);
}
```

2. Fehlerquelle: Use After Free

```
void Keyboard::plugin() {
    Plugbox::assign(Vector::KEYBOARD, this);
}

void main() {
    Keyboard{}.plugin();
    Core::Interrupt::enable();
    while (true);
}
```

2. Fehlerquelle: Use After Free

In Rust: Borrow Checker saves the day!

```
error[E0716]: temporary value dropped while borrowed
  --> src/main.rs:67:5
   |
67 |     Keyboard::new().plugin();
   |     ^^^^^^^^^^^^^^^^----------- temporary value is freed at the end of this statement
   |     |
   |     creates a temporary value which is freed while still in use
   |     argument requires that borrow lasts for `'static`
```

3. Fehlerquelle

```
fn main() {
    let app = Application::new();
    black_box(app);
}
```

3. Fehlerquelle: Stack Overflows

```rust
fn main() {
    // App hat eigenen Stack -> Stack Overflow
    let app = Application::new();
    black_box(app);
}
```

Hier hilft Rust nicht wirklich...

Fazit

Rust ist eine ergonomische Sprache für Betriebssysteme...

+ Moderne Sprachfeatures

Rust ist eine ergonomische Sprache für Betriebssysteme…

- \+ Moderne Sprachfeatures
- \+ Beugt vielen Fehlern vor
  - `nullptr`-Zugriffe
  - Use-After-Free

Rust ist eine ergonomische Sprache für Betriebssysteme…

- \+ Moderne Sprachfeatures
- \+ Beugt vielen Fehlern vor
  - `nullptr`-Zugriffe
  - Use-After-Free
- \+ (Meist) Elegante Abstraktionen

…mit einigen verbleibenden Kanten

+ Moderne Sprachfeatures
+ Beugt vielen Fehlern vor
  - `nullptr`-Zugriffe
  - Use-After-Free
+ (Meist) Elegante Abstraktionen

– Fehlende Sprachspezifikation

…mit einigen verbleibenden Kanten

- \+ Moderne Sprachfeatures
- \+ Beugt vielen Fehlern vor
  - `nullptr`-Zugriffe
  - Use-After-Free
- \+ (Meist) Elegante Abstraktionen

- – Fehlende Sprachspezifikation
- – Teilweise schwer zu debuggen

…mit einigen verbleibenden Kanten

+ Moderne Sprachfeatures
+ Beugt vielen Fehlern vor
  - `nullptr`-Zugriffe
  - Use-After-Free
+ (Meist) Elegante Abstraktionen

– Fehlende Sprachspezifikation

– Teilweise schwer zu debuggen

– Manchmal nicht so elegante Abstraktionen

Danke für die Aufmerksamkeit!
Fragen?

3. Fehlerquelle: Data Races

- In der Lehrveranstaltung meist schnell identifiziert

3. Fehlerquelle: Data Races

- In der Lehrveranstaltung meist schnell identifiziert
- Rust: nur threadsichere (Sync) Objekte können geteilt werden...

```
error[E0277]: `RefCell<i32>` cannot be shared between threads safely
  --> src/main.rs:59:24
   |
59 | static NOT_THREADSAFE: RefCell<i32> = RefCell::new(42);
   |                        ^^^^^^^^^^^^ `RefCell<i32>` cannot be shared between threads
   |
   = help: the trait `Sync` is not implemented for `RefCell<i32>`
   = note: shared static variables must have a type that implements `Sync`
```

3. Fehlerquelle: Data Races

- In der Lehrveranstaltung meist schnell identifiziert
- Rust: nur threadsichere (Sync) Objekte können geteilt werden...
- ...aber wann markiere ich etwas als Sync?

```
unsafe impl Sync for RefCell<i32> {} // <- Inkorrekt
```

Initialisierung von globalen Objekten

- C++ packt globale Konstruktoren in `.init_array`-Sektion
- Wir rufen sie manuell auf

Initialisierung von globalen Objekten

- C++ packt globale Konstruktoren in `.init_array`-Sektion
- Wir rufen sie manuell auf
- Rust verwendet keine `.init`-Sektion
- Verspätete Initialisierung stattdessen per Hand nachgebaut

4. Fehlerquelle: Stack Overflows

Hier hilft Rust nicht wirklich...

4. Fehlerquelle: Stack Overflows

Hier hilft Rust nicht wirklich...

```rust
const CPU_CORE_STACK_SIZE: usize = 16 * 4096;
```

(verglichen mit 4096 Byte in C++)

4. Fehlerquelle: Stack Overflows

Hier hilft Rust nicht wirklich...

```
const CPU_CORE_STACK_SIZE: usize = 16 * 4096;
```

(verglichen mit 4096 Byte in C++)

- Extrem hoher Stackverbrauch in Debug-Builds

4. Fehlerquelle: Stack Overflows

Hier hilft Rust nicht wirklich...

```
const CPU_CORE_STACK_SIZE: usize = 16 * 4096;
```

(verglichen mit 4096 Byte in C++)

- Extrem hoher Stackverbrauch in Debug-Builds
- Bei naiver Implementierung werden Variablen auf dem Stack angelegt und dann in statischen Bereich kopiert

4. Fehlerquelle: Stack Overflows

Hier hilft Rust nicht wirklich...

```
const CPU_CORE_STACK_SIZE: usize = 16 * 4096;
```

(verglichen mit 4096 Byte in C++)

- Extrem hoher Stackverbrauch in Debug-Builds
- Bei naiver Implementierung werden Variablen auf dem Stack angelegt und dann in statischen Bereich kopiert
- Keine einfach nutzbare Stack Usage Analysis (vgl. `-Wstack-usage=N`)