Systemnahe Programmierung in Rust

- Intro -

Hubert Högl

Hochschule Augsburg / Informatik
https://hhoegl.informatik.hs-augsburg.de/hhwiki/FrontPage

2022-11-24 23:17:27

Hello Rust

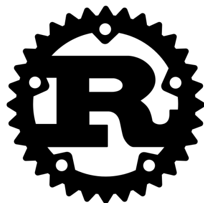

Figure 1: Logo

```rust
fn main() {
    println!("Hello, World!");
}
```

Krabbe “Ferris” als Maskottchen, siehe https://www.rustacean.net.

Was ist Systemprogrammierung?

Es gibt keine allgemeingültige verbindliche Definition für Systemprogrammierung, deshalb hier ein paar Aussagen, die für sich genommen stimmen:

- Systemprogrammierung steht im Gegensatz zur Anwendungsprogrammierung. Meist wird dabei direkt in der Sprache C mit Aufrufen der C Standardibliothek (POSIX Standard) gearbeitet. Die C Standardbibliothek fasst die Systemaufrufe (Software Interrupts) des Betriebssystems als Funktionsbibliothek zur Verfügung.
- Mit "System" meint man das Betriebssystem (Kernel + Dienstprogramme im "User Space"), d.h. alle Softwareteile die zunächst vorhanden sein müssen, um auf einem Rechner Anwendungsprogramme zu programmieren und laufen zu lassen. Die Programmierung von Betriebssystemen ist hardwarenah bzw. maschinennah.
- Systemprogramme sind auch Programme, die eng mit den Diensten und Abstraktionen des Betriebssystems zusammenarbeiten, d.h. mit Prozessen, Interprozesskommunikation, Threads, Timern, Signalen, Filesystemen, Netzwerk und so weiter.

Die folgende Abbildung zeigt grob das Zusammenspiel von Kernel und User Space.

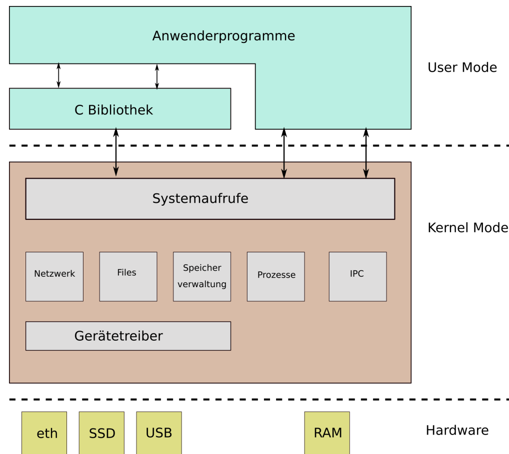

Figure 2: Kernel Architektur

Systemprogrammierung

- Systemprogrammierung wird seit gut 40 Jahren fast ausschliesslich in den Sprachen *C* (und ein wenig *C++*) und *Assembler* gemacht. Damit hat man zwar die volle Kontrolle über die Maschine, man kann aber leicht Programmierfehler machen, die vom Compiler nicht erkannt werden.

Nachteile von C

- C und C++ Programme haben oft Fehler in der Speicherverwaltung
- C und C++ Programme sind nicht typsicher (schwache Typisierung)
- "Null References: The Billion Dollar Mistake" (Tony Hoare)
- Gleichzeitigkeit ist schwer zu programmieren
- Beispiel: `mistery.c`
- Bei der Systemprogrammierung ist es oft wichtig zu wissen, in welchen Bereichen des Speichers bestimmte Teile des Programms liegen. Wir kommen auf das folgende Bild noch öfter zurück:

Speicherbereiche eines Prozesses. Die Bereiche werden auch Segmente (segments) oder Sektionen (sections) genannt.

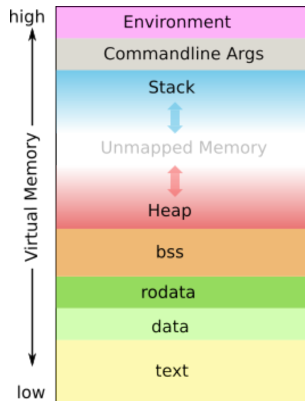

Figure 3: Speicherbild eines Prozesses

text Bereich für den Maschinencode.

data Bereich für initialisierte globale Daten (read/write).

rodata Bereich für initialisierte globale Daten (read-only).

bss Nicht-initialisierte globale Daten; wird z.B. für Puffer verwendet. Kommt von "Block Started by Symbol" aus den Anfängen der Rechnertechnik.

Heap Bereich für dynamisch allozierte Daten. Wird in C z.B. mit malloc() und free() aus der Standardbibliothek verwaltet, oder in C++ mit new() und delete().

Stack Speicher nach dem LIFO Prinzip der bei Funktionsaufrufen für die Argumentübergabe, lokale Variable, den Rückgabewert und die Rücksprungadresse verwendet wird. Wird durch die Maschinenbefehle PUSH und POP verwaltet.

(Fortsetzung)

Commandline Args Die Kommandozeilenargumente des Programms beim Aufruf.

Environment Die Umgebungsvariable des Prozesses.

Die genaue Verwendung der einzelnen Bereiche bei Rust werden wir später behandeln.

Wo wird Systemprogrammierung verwendet?

- Betriebssysteme
- Gerätetreiber für Betriebssysteme
- Betriebssystemnahe Programmierung
  - Absetzen von Systemaufrufen
  - Nutzen von Diensten der C Standardbibliothek (z.B. Speicherverwaltung, File I/O, Netzwerk, Zeit, Threads)
  - Emulatoren
  - Virtualisierung
  - Implementierung von Kommunikationsprotokollen
- Sicherheitskritische Software
- Kryptografie
- Embedded Systems ("bare-metal" Programmierung)
- Game Engines
- Signalverarbeitung von Audio- und Videodaten

Warum Rust?

(aus dem freien “Why Rust?” Report von Jim Blandy, 2015, siehe z.B. https://www.janwalter.org/Download/Books/why-rust.pdf)

- Zero-overhead principle, “Abstraction without overhead”
- Speichersicherheit (memory safety)
  - Keine Nullpointer-Dereferenzierung
  - Keine hängenden Pointer
  - Keine Pufferüberläufe
- Typsicherheit, obwohl es eine Systemprogrammiersprache ist

  “Note that being type safe is mostly independent of whether a language checks types at compile time or at run time: C checks at compile time, and is not type safe; Python checks at runtime, and is type safe. Any practical type-safe language must do at least some checks (array bounds checks, for example) at runtime.”
- Rust kompiliert statische Binaries, die sofort ausgeführt werden können. Es gibt keine Abhängigkeiten zu dynamischen Bibliotheken.
- Rust erzeugt schnelle Programme

Warum Rust? (2)

- Der generierte Maschinencode wird optimiert
- Vertrauenswürdiges Modell für parallele Programmierung

Nachteile

- Man kann beim Programmieren nicht “faul” sein. Auch kleine Nachlässigkeiten werden vom Compiler gnadenlos abgelehnt. Man brütet am Anfang die meiste Zeit über Warnungen und Fehlermeldungen des Compilers. Der Compiler gibt aber auch Hinweise aus, wie man Probleme beheben kann.
- Grosse Sprache, eher schwierig zu lernen, und auch schwierig zu vermitteln (wir werden sehen). Die Lernkurve ist steil.
- Kompilierzeit länger als bei C/C++.
- Zyklische Datenstrukturen sind nicht leicht zu programmieren (nur was für Fortgeschrittene).
- Zur Zeit (zu) viel Hype.

Anwendungsbereiche

- Ursprünglich für Systemprogrammierung entworfen, mittlerweile zur Vielzwecksprache geworden.
- Kommandozeilen-Werkzeuge. Viele bekannte Unix Tools sind in Rust neu geschrieben worden: `bat`, `fd`, `rg`, `exa`, `sd`, `dust`, `hck`, `tokei`, `procs`, ...

  Siehe meine Sammlung unter https://hhoegl.informatik.hs-augsburg.de/hhwiki/RustTools
- Verarbeitung grosser Datenmengen, z.B. Apache Arrow https://arrow.apache.org
- Applikationen in dynamischen Sprachen, z.B. Python, erweitern durch kompilierte Programme. Typischerweise wird das in C gemacht, sicherer wird es in Rust.
- Programmierung von Hardware mit wenig Resourcen (Mikrocontroller, Internet of Things).

Anwendungsbereiche (2)

- Serverseitige Programme (npm, tantivy, ...)
- Desktop-Anwendungen
- Native Applikationen auf Smartphone
- Web
  - WebAssembly
  - Kleine Web-Frameworks (ähnlich Flask): actix-web, tide, warp
  - Grosse Web-Frameworks (ähnlich Django): Rocket
- Systemprogrammierung (Fuchsia OS, Low-level Komponenten in MS Windows, AWS Bottlerocket, Teile des Linux Kernels)

- `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`

  → https://www.rust-lang.org/tools/install
- Werkzeuge `rustc`, `cargo`, `rustup`
- Rustup
  - `rustup show`
  - `rustup check`
  - `rustup update`
  - `rustup self uninstall`

rustc

- `rustc --version` gibt Version aus.
- "Hallo Welt"

```rust
// Datei "programm.rs"
fn main() {
        println!("Hallo Welt!");
}
```

- `rustc programm.rs` erzeugt ausführbare Datei `programm`. Laufen lassen mit `./programm`.
- `rustc --test programm.rs` kompiliert die Testfälle im Programm.

cargo

- `cargo new programm` erzeugt ein Package.

```
programm
       +-- Cargo.toml
       +-- src
           +-- main.rs
```

```
cd programm
cargo --version
cargo build
cargo build --release
cargo run
cargo check
cargo fmt
cargo clean
```

cargo (2)

- Cargo.toml

```
[package]
name = "hello_cargo"
version = "0.1.0"
edition = "2021"

[dependencies]
```

Clippy

Statische Codeanalyse (Lint)

- → https://github.com/rust-lang/rust-clippy#usage
- `rustup component add clippy`
- `cargo clippy`

ferris-says

https://www.rust-lang.org/learn/get-started

Cargo Package, `Cargo.toml` (Abhängigkeit von `ferris_says`)

```
use ferris_says::say; // from the previous step
use std::io::{stdout, BufWriter};

fn main() {
    let stdout = stdout();
    let message = String::from("Hello fellow Rustaceans!");
    let width = message.chars().count();

    let mut writer = BufWriter::new(stdout.lock());
    say(message.as_bytes(), width, &mut writer).unwrap();
}
```

The Book

by Steve Klabnik and Carol Nichols, with contributions from the Rust Community

DE https://rust-lang-de.github.io/rustbook-de/

EN https://doc.rust-lang.org/stable/book/

[BOOK] Kapitel 1

- Installation (Windows, Linux, Mac)
- Aufruf von rustc
- Aufruf von cargo

[BOOK] Kapitel 2

- "guessing game"

Literaturangaben

[BOOK] The Rust Programming Language, https://doc.rust-lang.org/book/