Systemnahe Programmierung in Rust

- "The Book" / Nebenläufigkeit / Kap. 16 -

Hubert Högl

Technische Hochschule Augsburg / Informatik
https://tha.de/~hhoegl

2024-10-05 13:23:33

Furchtlose Nebenläufigkeit

Kap. 16: https://rust-lang-de.github.io/rustbook-de/ch16-00-concurrency.html

Nebenläufige Programmierung (*concurrent programming*) $\leftrightarrow$ Parallele Programmierung

Nebenläufig = verschiedene Teile eines Programmes werden unabhängig voneinander ausgeführt.

Parallel = verschiedene Teile eines Programmes werden gleichzeitig ausgeführt.

"Nebenläufig" im Buch bedeutet immer nebenläufig *und/oder* gleichzeitig.

Eigentümerschaft und Rust-Typen helfen nicht nur bei Speichersicherheit, sondern auch bei Nebenläufigkeitsproblemen.

Threads (16.1)

Threads (Stränge)

Probleme, die durch nebeneinander laufende Threads erzeugt werden:

- Wettlaufsituationen (race conditions), bei denen Stränge auf Daten oder Ressourcen in einer inkonsistenten Reihenfolge zugreifen.
- Deadlocks, bei denen zwei Stränge auf den jeweils anderen warten, sodass beide Stränge nicht fortgesetzt werden können.
- Fehler, die nur in bestimmten Situationen auftreten und schwer zu reproduzieren und zu beheben sind.

In Rust 1:1 Modell: 1 OS Thread : 1 Rust Thread

Threads (2)

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Hallo Zahl {} aus dem erzeugten Strang!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    // handle.join().unwrap();  // oder hier

    for i in 1..5 {
        println!("Hallo Zahl {} aus dem Hauptstrang!", i);
        thread::sleep(Duration::from_millis(1));
    }
    handle.join().unwrap();
}
```

Threads (3)

Der Vektor `v` wird in der Closure als Referenz ausgeliehen, da dies für `println!` reicht. Allerdings kann das Hauptprogramm jederzeit den Vektor entfernen (`drop`) und somit ist das Programm nicht gültig.

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Hier ist ein Vektor: {:?}", v);
    });

    handle.join().unwrap();
}
```

Lösung: `move`

```
    let handle = thread::spawn(move || {
        println!("Hier ist ein Vektor: {:?}", v);
    });
```

Nachrichtenaustausch (16.2)

Go Slogan: *Do not communicate by sharing memory; instead, share memory by communicating.*

https://go.dev/doc/effective_go#concurrency

Kanäle in der Standardbibliothek (`mpsc::channel;`)

- Sender (stromaufwärts)
- Empfänger (stromabwärts)
- Kanal schliessen durch `drop` von Empfänger oder Sender

Nachrichtenaustausch (2)

- recv() blockiert
- try_recv() blockiert nicht

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hallo");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Erhalten: {}", received);
}
```

Nachrichtenaustausch (3)

`send()` übernimmt die Eigentümerschaft am Argument und verschiebt sie zum Empfänger.

Nun senden zwei Threads, einer in `tx1`, der andere in `tx`. Das Hauptprogramm empfängt in `rx`. Die Nachrichten kommen beim Empfänger in zufälliger Reihenfolge an.

```
let (tx, rx) = mpsc::channel();
let tx1 = tx.clone();

...
```

Nebenläufigkeit auf gemeinsamen Daten (18.3)

Gemeinsam genutzter Hauptspeicher ist wie Mehrfacheigentum.

Mehrfacheigentum ist möglich, siehe Smart Pointer (Kap. 15)

`Mutex` = *mutual exclusion* (Metapher: Podium, Diskussion, nur ein Mikrofon)

- Du musst versuchen, die Sperre zu erwerben, bevor du die Daten verwendest.
- Wenn du mit den Daten, die der Mutex schützt, fertig bist, musst du die Daten entsperren, damit andere Stränge die Sperre übernehmen können.

Traditionell: Mutexe sind schwierig, Kanäle sind einfach (deshalb verwendet Go z.B. Kanäle).

Rust: Durch Typsystem und Eigentumsregeln werden Mutexe einfach zu verwenden (deshalb u.a. “furchtlose Nebenläufigkeit”).

Nebenläufigkeit auf gemeinsamen Daten (2)

`Mutex<T>` API

```
// single threaded
let m = Mutex::new(5);
let mut num = m.lock().unwrap();
*num = 6;
```

`m.lock()` gibt Smart Pointer `MutexGuard` zurück, der in `LockResult` gewickelt ist. `MutexGuard` implementiert `Deref`. Die Sperre wird aufgehoben, wenn Smart Pointer den Gültigkeitsbereich verlässt (`drop`).

Nebenläufigkeit auf gemeinsamen Daten (3)

Nun 10 Threads, `counter` mit Mutex, der von jedem Thread hochgezählt wird und an Vektor `handles` angehängt wird.

Kompilierfehler: Problem, dass counter nicht in mehrere Threads verschoben werden kann.

```
fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
```

Nebenläufigkeit auf gemeinsamen Daten (4)

Lösung: Ähnlich wie in Kap. 15: Mehrere Eigentümer mit `Rc<T>`, jedoch ist dieser nicht sicher bei mehreren Threads. Verwende statt dessen **Arc**.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }
```

Nebenläufigkeit auf gemeinsamen Daten (5)

Für einfachere Operationen gibt es statt `Mutex<T>` in der Standardbibliothek `std::sync::atomic` (sicheren, gleichzeitigen, atomaren Zugriff auf primitive Typen).

Ähnlichkeit:

- `RefCell<T>` / `Rc<T>`
- `Mutex<T>` / `Arc<T>`

Achtung:

- Mit `Mutex<T>` kann man Deadlocks machen

Erweiterbare Nebenläufigkeit mit `Sync` und `Send` (16.4)

XXX to do

Noch ein paar Hinweise

- Code: `material/Beispiele/fearless`
- The Book Kap. 20 “Final Project: Building a Multithreaded Web Server”
- Jens Breitbart, Stefan Lankes, Wettlaufoperationen, iX Developer 2019 - Moderne Softwareentwicklung.
  https://hhoegl.informatik.hs-augsburg.de/sysprog/Rust/iX-Developer-Rust-2019.pdf
  Code: `material/Beispiele/ix-parallel/`
- Folien von Lukas Kalbertodt (2016) - siehe `18-Concurrency-Multithreading.pdf`
  https://github.com/LukasKalbertodt/programmieren-in-rust/tree/master/slides
- Buch Programming Rust, Kap. 2 “A Tour of Rust” (Mandelbrot-Set) und Kap. 19 “Concurrency”
  https://learning.oreilly.com/library/view/programming-rust-2nd/9781492052586