# Fearless Concurrency

## Programmieren in Rust

S. Perren, F. Mayer, F. Klopfer

24. Mai 2019

# Gliederung

# Verwendung

- Ermöglicht Teile eines Programms **nebenläufig** auszuführen.

  → **Erhöhte Performance**
  → **Erhöhte Programmkomplexität**

- Da Threads nebenläufig sind, können wir keine Aussage über deren Ausführungsreihenfolge treffen.

  → **Probleme**:
  - Race Conditions
  - Deadlocks
  - Programmfehler, welche sich nur sehr schwer repoduzieren und beheben lassen

# Verwendung

- Rust versucht die negativen Auswirkungen, durch die Verwendung von Threads zu minimieren:

  → Ownershipsystem
  → Typüberprüfungen

- Dadurch sind viele Fehler die bei der nebenläufigen Programmierung auftreten können, bereits zur **Compile-Zeit** bekannt.

# Erzeugen von Threads

- Ein neuer Thread lässt sich mittels **`spawn()`** erzeugen:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
```

  → nimmt eine **`Closure`** entgegen
  → liefert **`JoinHandle`** zurück.

- Alle Threads eines Programms werden beendet, wenn der **main-Thread** beendet wird!

# Erzeugen von Threads

- Beispiel:

```rust
// hello.rs
use std::thread;

fn main() {
    thread::spawn(f: || {
        println!("Hello from another thread!");
    });

    println!("Hello from main thread!");
}
```

Welche Ausgabe liefert das Programm?

# Erzeugen von Threads

- Ausgabe:

```
$ ./hello
Hello from main thread!
$
```

→ Erzeugter Thread wird sofort beendet.
→ Damit wir die Ausgabe des erzeugten Threads sehen können, muss der main-Thread auf dessen Beendigung **warten**.

# Warten auf Threads

- Mittels `join()` kann man auf einen bereits gestarteten Thread warten:

```
pub fn join(self) -> Result<T>
```

→ liefert `Result` zurück:
  - `Ok(T)` wenn Thread sich beendet hat.
  - `Err` wenn Thread panicked.

# Warten auf Threads

- Beispiel von oben mit Warten:

```rust
// hello.rs
use std::thread;

fn main() {
    let handle :JoinHandle<?> = thread::spawn( f: || {
        println!("Hello from another thread!");
    });

    println!("Hello from main thread!");

    match handle.join() {
        Ok(_) => {},
        Err(_) => {println!("Child thread panicked!")}
    }
}
```

- Ausgabe:

```
$ ./hello
Hello from main thread!
Hello from another thread!
$
```

# Thread schlafen lassen

- Mittels **sleep()** kann man einen Thread für eine gewisse Zeit schlafen lassen:

```
pub fn sleep(dur: Duration)
```

- Niemals mit Hilfe von **sleep()** auf Beendigung eines Threads warten! Dafür verwenden wir **join()**.

# Thread schlafen lassen

- Beispiel:

```rust
// sleep.rs
use std::thread::sleep;
use std::time::Duration;
use std::thread;

fn main() {
    println!("Hello from main thread!\n");
    thread::spawn( f: || {
        println!("Hello from another thread!");
        println!("Doing an expensive operation...");

        let a :i32 = 1;
        let b :i32 = 1;
        let res :i32 = a + b;

        println!("Finished... res: {}", res);
    });

    sleep( dur: Duration::from_secs( secs: 1));
}
```

- Ausgabe:

```
$ ./sleep
Hello from main thread!

Hello from another thread!
Doing an expensive operation...
Finished... res: 2
$
```

# Länger leben als Elternthread

- Threads können **länger** leben als Elternthread:

```rust
// live_longer.rs
use std::thread;
use std::thread::sleep;
use std::time::Duration;

fn main() {
    println!("Hello from main thread!");
    thread::spawn( f: t1);
    sleep( dur: Duration::from_secs( secs: 1));
    println!("Good bye from main thread!");
}

fn t1() {
    println!("Hello from t1!");
    thread::spawn( f: t2);
    println!("Good bye from t1!");
}

fn t2() {
    println!("Hello from t2!");
    thread::sleep( dur: Duration::from_millis( millis: 500));
    println!("Good bye from t2!");
}
```

- Ausgabe:

```
$ ./sleep
Hello from main thread!
Hello from t1!
Good bye from t1!
Hello from t2!
Good bye from t2!
Good bye from main thread!
$
```

- Ausnahme: **main-Thread**.

# Ergebnis eines Threads zurückgeben

- Threads können ein Ergebnis zurückgeben:

```rust
// add_forty_one.rs
use std::thread;
use std::io::stdin;

fn main() {
    let handle :JoinHandle<i32> = thread::spawn( f: || {
        println!("Please enter a number: ");

        let mut user_input :String = String::new();
        stdin().read_line( buf: &mut user_input).unwrap();

        let num: i32 = user_input.trim().parse().unwrap();

        num + 41
    });

    let res :i32 = handle.join().unwrap();
    println!("res: {}", res);
}
```

- Ausgabe:

```
$ ./add_forty_one
Please enter a number:
1
res: 42
$
```

# Variablen von außen verwenden

- Variablen können einem Thread von außen mitgegeben werden:

```rust
// is_even.rs
use std::io::stdin;
use std::thread;

fn main() {
    println!("Please enter a number:");

    // read user input
    let mut user_input :String = String::new();
    stdin().read_line( buf: &mut user_input);

    // parse user input to i32
    let num: i32 = user_input.trim().parse().unwrap();

    // check if num is even
    let handle :JoinHandle<bool> = thread::spawn( f: || is_even(num));

    // do something useful in the meantime

    println!("res: {}", handle.join().unwrap());
}
```

```rust
fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        true
    } else {
        false
    }
}
```

# Variablen von außen verwenden

- Compiler-Output:

```
error[E0373]: closure may outlive the current function, but it borrows `num`, which is owned by the current function
  --> src/main.rs:17:32
   |
17 |     let handle = thread::spawn(|| is_even(num));
   |                                ^^          --- `num` is borrowed here
   |                                |
   |                                may outlive borrowed value `num`
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:17:18
   |
17 |     let handle = thread::spawn(|| is_even(num));
   |                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
help: to force the closure to take ownership of `num` (and any other referenced variables), use the `move` keyword
   |
17 |     let handle = thread::spawn(move || is_even(num));
   |                                ^^^^^^^
```

- Woher weiß der Compiler das?

# Variablen von außen verwenden

- Da es sich um eine `FnOnce()` -Closure mit `'static` Lieftime handelt und Rust versucht den Wert von `num` zu borrowen, ist nicht sicher gestellt, dass der main-Thread länger als der erzeugte Thread lebt.
- Daher verbietet uns der Compiler so vorzugehen.
- Abhilfe durch `move`-Closure:
  → Erlaubt uns Daten eines Threads in einem anderen Thread zu verwenden.
  → Ownership wird dabei an den Thread übergeben

# Variablen von außen verwenden

- Angepasstes Programm mit **`move`**-Closure:

```rust
// is_even.rs
use std::io::stdin;
use std::thread;

fn main() {
    println!("Please enter a number:");

    // read user input
    let mut user_input :String = String::new();
    stdin().read_line( buf: &mut user_input);

    // parse user input to i32
    let num: i32 = user_input.trim().parse().unwrap();

    // check if num is even
    let handle :JoinHandle<bool> = thread::spawn( f: move || is_even(num));

    // do something useful in the meantime

    println!("res: {}", handle.join().unwrap());
}
```

```rust
fn is_even(num: i32) -> bool {
    if num % 2 == 0 {
        true
    } else {
        false
    }
}
```

- Ausgabe:

```
$ ./is_even
Please enter a number:
42
res: true
$
```

# Transferieren von Daten zwischen Threads

- Davor: Thread Syntax & Semantik
- Jetzt: Channels
- Benutzung von Rusts standart Bibliothek für mpsc Channels
- Wie sieht es aus mit mpmc?
- Außerdem: Die einzigen beiden Concurrency Konzepte der Sprache Rust selbst

# Idee und Syntax

Idee: "Do not communicate by sharing memory; instead, share memory by communicating."
- Go language documentation

Lösung: `let (tx, rx) = mpsc::channel();`

wobei `tx` der Produzent und `rx` der Konsument ist

- Ein Channel ist geschlossen sobald ein Ende gedroppt wurde

# Der Produzent

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

- Der erstellte Thread benötigt die Ownership für den Produzenten → `move` Closure
- `send()`:
  - übernimmt Ownership des übergebenen Datums,
  - liefert ein `Result<T, E>` für den Fall, dass es keinen Empfänger gibt, oder dieser bereits gedroppt wurde

# Der Konsument

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

Der `Receiver` stellt zwei Methoden zum Empfangen von Daten zur Verfügung:

- `recv()`: blockiert den Thread und wartet auf ein Datum
  - Liefert ein `Result<T, E>` sobald ein Datum gesendet wurde
  - Das `Result<T, E>` hält einen `Err` falls die sendende Seite des Channels geschlossen wurde
- `try_recv()`: blockiert nicht und liefert sofort ein `Result<T, E>`

# Mehrere Produzenten

- Wie kommen wir zum `m` in `mpsc`?
- Der idiomatische Weg ist `clone` für den Sender aufzurufen
- Anschließend kann die Ownership jedes Klones an einen Thread übergeben werden

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

# mpsc Beispiel

```rust
let (tx, rx) = mpsc::channel();

let tx1 = mpsc::Sender::clone(&tx);
thread::spawn(move || {
    let vals = vec![
        String::from("Hello"),
        String::from("there!"),
    ];
    for val in vals {
        tx1.send(val).unwrap();
    }
});
```

```rust
thread::spawn(move || {
    let vals = vec![
        String::from("General"),
        String::from("Kenobi.."),
    ];
    thread::sleep(Duration::from_secs(1));
    for val in vals {
        tx.send(val).unwrap();
    }
});

for received in rx {
    println!("{}", received);
}
```

# Was ist mit `mpmc`?

Channels sind in Go zu einem beliebten Synchronisierung-Tool geworden während Rusts standart Implementierungen einige Nachteile aufweisen:

- `Sender` ist nicht `Sync` (`&Sender` ist nicht `Send`),
- `Receiver` kann nicht geklont werden,
- `select!` macro ist immer noch nicht stable
- Bounded channels sind einfach Deques umgeben von Mutexen und damit eher langsam

# Community to the rescue

Daher entschied sich die Community ihre eigene Library für Concurrency und Parallelität zu schreiben: `Crossbeam`

- Im Grunde die selbe Idee wie `std::sync`, nur größer
- Soll Low-Level Werkzeuge bereit stellen auf welchen Libraries wie Rayon und Tokio aufbauen können
- Problem ist Lock-freie und damit schnelle Strukturen sind schwer mit manuellem Memory-Management vereinbar
- Ein Lösungsansatz wurde in der sog. Epochen-basierten Garbage Collection gefunden: `aturon.github.io/blog/2015/08/27/epoch/` (Blockeintrag von Aaron Turon, Mitarbeiter bei Mozilla und Rust-Entwickler)

# Crossbeam

Überblick über das `crossbeam-channel` Crate:

- `Sender` und `Receiver` können geklont und zwischen Threads geteilt werden
- `select!` Macro kann über mehrere, dynamisch aufgebaute Channel-operationen blockieren
- Sehr wenige Locks für gute Performance (Mozillas und Samsungs Servo Projekt hat bereits von `mpsc` auf `crossbeam-channel` umgestellt)

`github.com/crossbeam-rs/crossbeam`

# Send & Sync

- Die meißten Concurrency Features sind Teil der standart Bibliothek von Rust
- Nur zwei Konzepte sind tatsächlich in der Sprache selbst verbaut
- Die `std::marker` Traits `Sync` und `Send`

# Send & Sync

`Send`

- Zeigt an, dass der Typ zwischen Threads übertragen werden kann
- Wird automatisch implementiert wenn der Compiler es für angemessen hält
  (z.B. bei Typen welche ausschließlich aus `Send` Typen bestehen)

`Sync`

- Zeigt an, dass eine Referenz auf einen Typ zwischen Threads übertragen werden kann
- Ein Typ `T` ist genau dann `Sync`, wenn `&T` `Send` ist

Note: `Send` und `Sync` selbst zu implementieren bedeutet unsafe Rust Code zu schreiben!

# Shared-State Concurrency

- Bisher: Channels, also Single Ownership
- Jetzt: Shared Memory, also Multiple Ownership.
- Mehrere threads können auf die gleiche Speicherstelle zur gleichen Zeit zugreifen
- MutExe garantieren Konsistenz
- Arc aka Atomic Reference Counter macht Multiple Ownership möglich

# Mutex aka Mutual Exclusion

- erlauben einem Thread gleichzeitig auf die Speicherstelle zu zugreifen
- Datentype der geschützt werden soll wird in den Mutex gepackt
- Vor dem Zugriff muss der Mutex `gelockt` werden
- Freigegeben wird implizit durch `Drop`
- Deadlocks sind trotzdem möglich $\Rightarrow$ `try_lock` und der Bankier-Algo oder z.B. wait/wound

# Die Mutex<T> API I

- `try_lock:  Mutex` → `MutexGuard` oder `PoisonError`
- `lock` funktioniert ähnlich aber blockt den thread (Deadlock!)
- MutexGuard ist ein smart pointer & gibt das lock in drop() frei
- OS spezifischer Code
- Cross-platform Wrapper
- High-level Mutexes

# Die Mutex<T> API II

*Ein pthread Mutex muss statisch an der gleichen Speicherstelle stehen. Wenn man einen solchen realloziert, verliert man die thread-Sicherheit*

```
pub struct Mutex<T: ?Sized> {
    // Note that this mutex is in a *box*,
    //not inlined into the struct itself.
    // The Rust Mutex can be safely moved at any time.
    // To ensure native mutex is used correctly
    // we box inner mutex to assign a const. addr.
    inner: Box<sys::Mutex>,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

# Die Mutex<T> API III

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

# Poisoning

Sollte nur mit gutem Grund genutzt werden!

- Normalerweise: `lock().unwrap()` um zu propagieren, falls ein thread `panic()`t
- Erholung möglich: `PoisonError.into_inner()` → `MutexGuard`

```
let mut guard = match lock.lock() {
     Ok(guard) => guard,
     Err(poisoned) => poisoned.into_inner(),
 };
 *guard += 1;
```

# Atomics

- OS, Compiler, Processor induzieren non-determinism durch Interrupts, Optimierungen und Cache Nutzung
  ⇒Keine garantie über die Ausführungsordnung oder ob überhaupt alles ausgeführt wird
- Atomic Instruktion: "Führe diese Instruktionen ungestört aus"
- x86 garantiert strong ordering (locking), ARM weak ordering (exclusive monitor aka load-linked/store-conditional) Garantien
- std::sync::atomic::Ordering geben die möglichkeit die Garantien zu steuern

# Die Arc<T> API I

- Atomic Reference Counter: Multiple ownership, Thread-safe durch die nutzung von `AtomicUsize`
- `clone: &Arc -> Arc`
- Nur Lese-Zugriff möglich
- Thread-safe Reference != Thread-safe data

# Die Arc<T> API II

```
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::thread;

let val = Arc::new(AtomicUsize::new(5));

for _ in 0..10 {
    let val = Arc::clone(&val);

    thread::spawn(move || {
        let v = val.fetch_add(1, Ordering::SeqCst);
        println!("{:?}", v);
    });
}
```

# Die Arc<T> API III

```rust
use std::sync::{Mutex, Arc};
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);
    }

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

    println!("Result: {}", *counter.lock().unwrap());
}
```

# Andere Mechanismen aus std::sync

- Barrier: Ensures multiple threads will wait to reach a point and resume execution all together
- CondVar: Blocking a thread while waiting for an event to occur
- RWLock: Multiple Reader, one writer at a time
- Weak: Arc with non-owning references

# Futures

RFC: Futures, Async & Await
Async Buch
Wird am 3. Juno gemerget!

- Async fn: Schicke eine Anfrage und warte auf die Antwort
- Futures: Ein noch nicht erhaltenes Result
- `await: Future` $\rightarrow$ `Result`

Wozu braucht man den Spaß?

# Why Async?

- Jeder gespawnte thread verbraucht Ressourcen und muss Verwaltet, insbesondere geschedulet werden
- Stattdessen nutzt man Worker threads oder einen threadpool.
  $\Rightarrow$ Weniger Overhead für den Kernel, einfache Verwaltung der Tasks, weil asynchron, nicht blockierend
- Sprich: Blockt eine async fn, wird sie wieder in die queue eingereiht und ein anderer Task wird ausgeführt

# Anwendungen

- I/O: mio
- Netzwerk-orientiert: tokio
- Http Server: hyper
- Rechenintensive Aufgaben: rayon
- RPC: tarpc
- DB queries: tokio_postgres
- Generelle Werkzeuge für Executors z.B. TaskScheduling, Thread-sfae Datenstrukturen, MPMC, garbage collector: crossbeam

# async/await!

- async transformiert eine Funktion in einen endlichen Automaten, der das `Future`-Trait implementiert
- `executor::block_on(future)` um die asynchrone Funktion auszuführen (nicht Teil des RFC/std)
- await!() um auf das `Result` zu warten, join!(f1, f2) um auf mehrere zu warten

Nur `async`, `await!` und `Waker` werden gemerget! Die Runtime (z.B. tokio) soll nicht Teil von Rust's `std` werden

# Pseudo-Example

```
async fn learn_song() -> Song { ... }
async fn sing_song(song: Song) { ... }
async fn dance() { ... }

async fn learn_and_sing() {
    // Wait until the song has been learned before singing it.
    // We use `await!` here rather than `block_on` to prevent blocking the
    // thread, which makes it possible to `dance` at the same time.
    let song = await!(learn_song());
    await!(sing_song(song));
}

async fn async_main() {
    let f1 = learn_and_sing();
    let f2 = dance();

    // `join!` is like `await!` but can wait for multiple futures concurrently.
    // If we're temporarily blocked in the `learn_and_sing` future, the `dance`
    // future will take over the current thread. If `dance` becomes blocked,
    // `learn_and_sing` can take back over. If both futures are blocked, then
    // `async_main` is blocked and will yield to the executor.
    join!(f1, f2)
}

fn main() {
    block_on(async_main()); // async_main is run on an executor
}
```

# References I


S. Klabnik, C. Nichols, and R. Community, *The Rust Programming Language*. No Starch Press, Incorporated, 2018, ISBN: 9781593278281. [Online]. Available: https://books.google.de/books?id=jgHoAQAACAAJ.


(May 14, 2019). The rustnomicon, [Online]. Available: https://doc.rust-lang.org/nomicon/concurrency.html (visited on 05/17/2019).


(May 20, 2019). Effective go, [Online]. Available: https://golang.org/doc/effective_go.html (visited on 05/20/2019).

# References II


(Aug. 27, 2015). Lock-freedom without garbage collection, [Online]. Available: https://aturon.github.io/blog/2015/08/27/epoch/ (visited on 05/20/2019).


(Jan. 29, 2019). Lock-free rust: Crossbeam in 2019, [Online]. Available: https://stjepang.github.io/2019/01/29/lock-free-rust-crossbeam-in-2019.html (visited on 05/23/2019).


(). Crossbeam, [Online]. Available: https://github.com/crossbeam-rs/crossbeam (visited on 05/23/2019).

# References III


(May 14, 2019). University of pennnsylvania, cis198, slide sets 10 & 11, [Online]. Available: https://github.com/cis198-2016f/slides/blob/gh-pages/10/content.md (visited on 05/17/2019).


(May 14, 2019). Std::sync - rust, [Online]. Available: https://doc.rust-lang.org/std/sync/ (visited on 05/17/2019).


(Mar. 28, 2019). Getting started - rust async book, [Online]. Available: https://rust-lang.github.io/async-book/ (visited on 05/17/2019).


A. Turon. (Feb. 26, 2019), Zero-cost futures in rust · aaron turon, [Online]. Available: https://aturon.github.io/2016/08/11/futures/ (visited on 05/17/2019).

# References IV


(May 14, 2019). The rust embedded book - concurrency, [Online]. Available: https://docs.rust-embedded.org/book/concurrency/ (visited on 05/17/2019).


(Jan. 30, 2019). Concurrency multithreading, [Online]. Available: https://github.com/LukasKalbertodt/programmieren-in-rust/blob/master/slides/18-Concurrency-Multithreading.pdf (visited on 05/22/2019).


(May 22, 2019). Std::thread - rust, [Online]. Available: https://doc.rust-lang.org/std/thread/ (visited on 05/22/2019).