Fearless Concurrency

S. Perren, F. Mayer, F. Klopfer

24. Mai, 2019

Threads

- **Verwendung:**
  Ermöglicht Programmteile **nebenläufig** auszuführen.
  → Erhöt **Programmpeformance** sowie **Programmkomplexität**
  → Man kann **keine** Aussage über die Ausführungsreihenfolge der Threads treffen.

- **Erzeugen von Threads**:

```rust
use std::thread;

let handle = thread::spawn(|| {
    // put your code here
});
```

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

- **Warten auf Threads:**

```rust
// thread was previously spawned
handle.join()
```

- **Thread schlafen lassen:**

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

// let thread sleep for one sec
thread::sleep(Duration::from_secs(1));
```

  Niemals mittels `thread::sleep()` auf die Beendigung eines Threads warten!
  Threads können **länger** leben als der Elternthread. **Ausnahme:** main-Thread.

- **Ergebnis eines Threads zurück geben:**

```rust
use std::thread;
use std::io::stdin;

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

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

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

        num + 41
    });

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

- **Variable von außen in Thread verwenden:**

```rust
use std::thread;

fn main() {
    let vec = vec![0, 8, 15];

    let handle = thread::spawn(move || {
        println!("vec: {:?}", vec);
    });

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

  Beachte: Hier wird das `move`-Keyword benötigt, damit die Closure die Ownership der verwendeten Varibalen erhält. Anderenfalls würde versucht werden, den Wert der Closure zu borrowen, was der Compiler nicht zulässt.

Channels

- **Verwendung:**

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

Erstellt einen untypisierten Channel.

- **Produzent:**

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

  - übernimmt Ownership des übergebenen Datums,
  - liefert ein `Result<T, E>` für den Fall, dass es keinen Empfänger gibt, oder dieser bereits gedroppt wurde

- **Konsument:**

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

  - `recv()`: blockiert den Thread und wartet auf ein Datum
  - `try_recv()`: blockiert nicht und liefert sofort ein `Result<T, E>`

- **Mehrere Produzenten:**

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

- **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

Shared Memory

- Mehrere Threads sollen gleichzeitig auf die gleiche Speicherstelle zugreifen
- Atomics sind Hardware-unterstützte thredsichere Datentypen

```
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);
    });
}
```

- Mutex "schützt" die Daten:

```
let m = Mutex::new(0);
{
    let mut val = m.lock().unwrap();
    *m = 42;
}
```

- Arc macht Multiple Ownership mit mehreren Threads möglich durch interne Verwendung von Atomics:

```
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();
}
```

- Andere Mechanismen:
  - 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

Aufgabenstellung:

Schreiben Sie ein Programm in dem ein `int`-Counter erstellt wird. Der `main`-Thread soll dann `N`-viele Threads starten, wobei jeder Thread den Wert des Counters um 1 erhöht. Der `main`-Thread soll mit Hilfe eines MPSC-Channels auf Beendigung der Threads warten, d.h. es soll kein `join()` verwendet werden.

Tipp: Man kann auch den `unit`-Type:`()` verschicken.