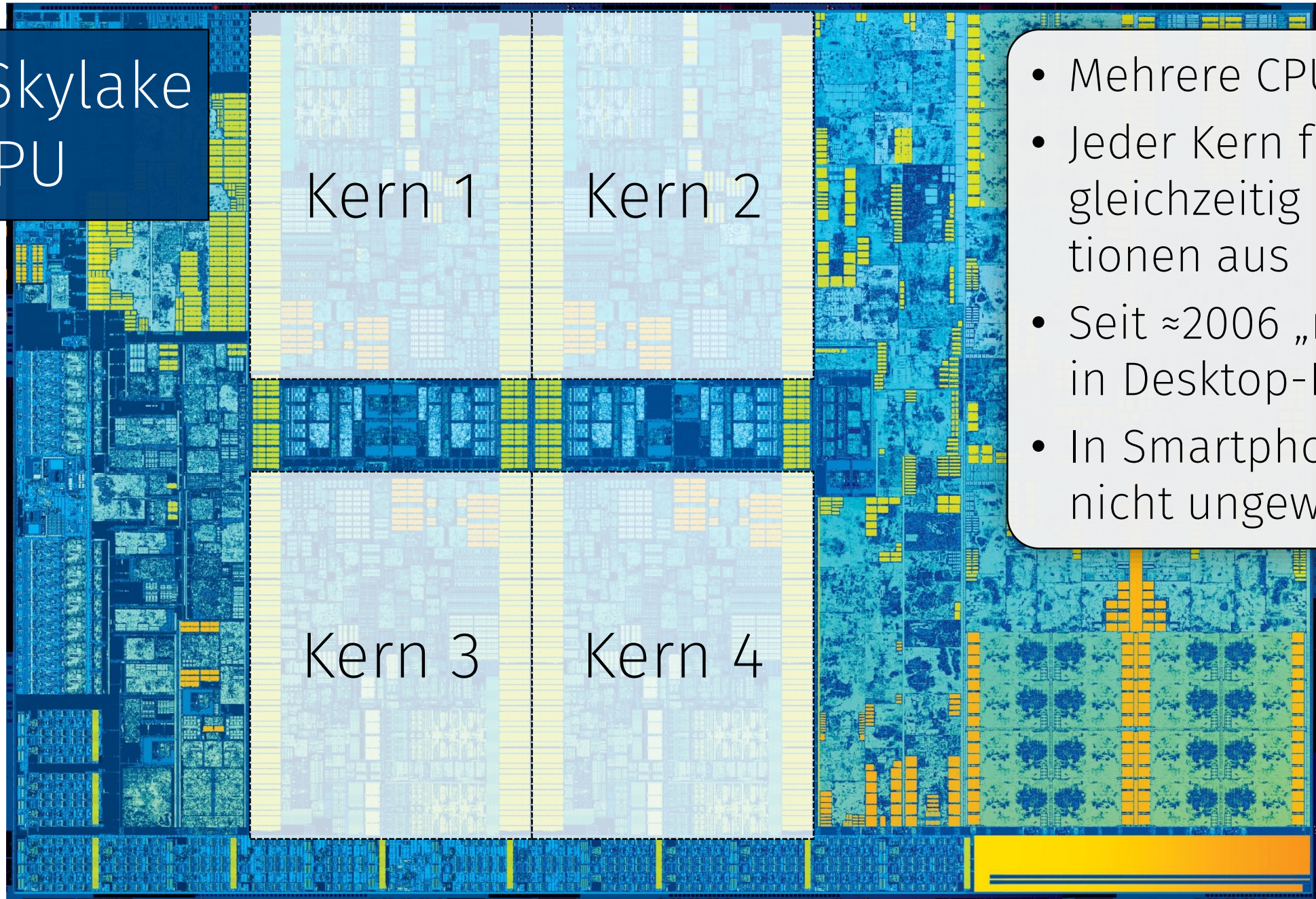


18.

Concurrency & Multithreading

Intel Skylake CPU

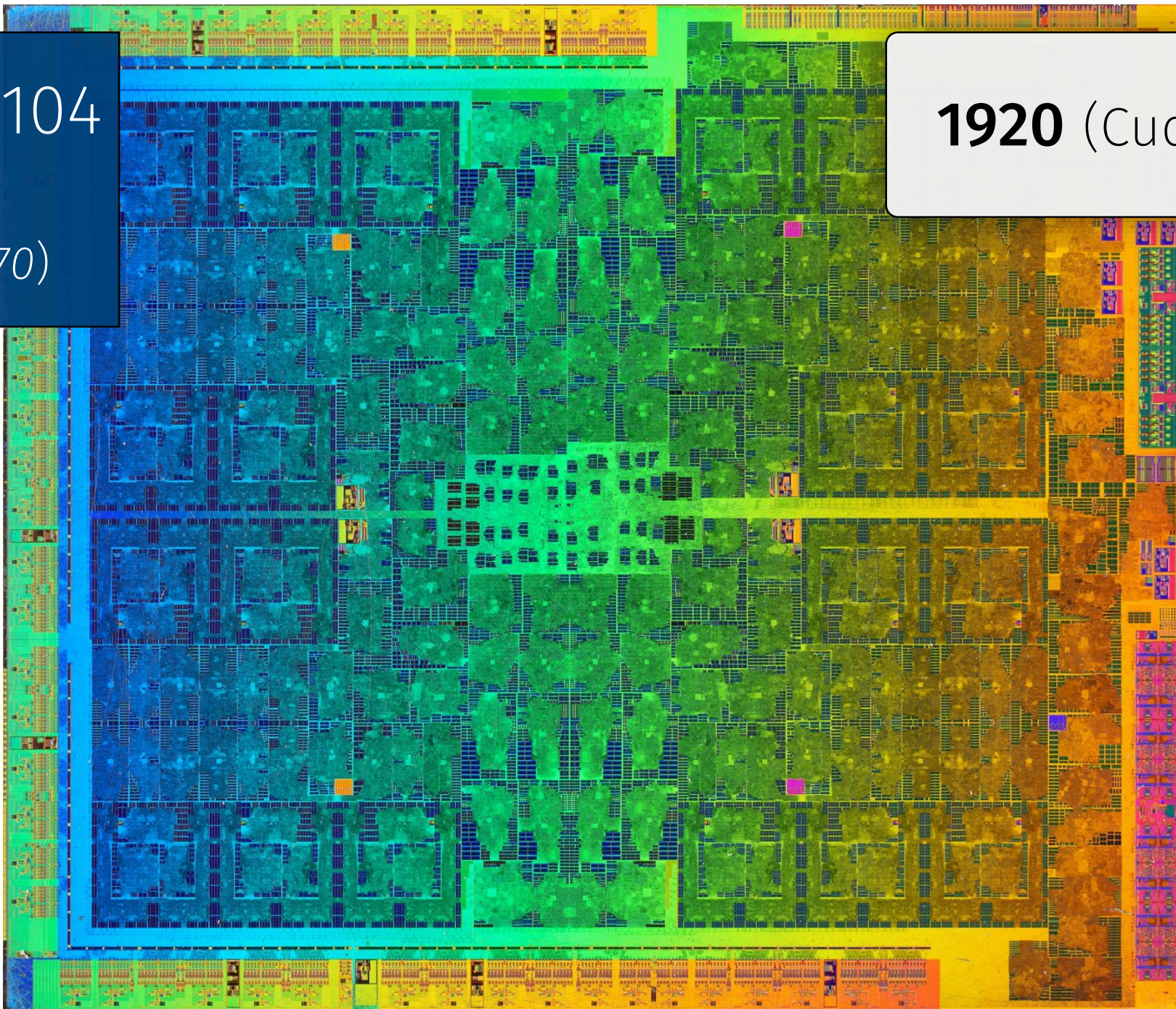


- Mehrere CPU-Kerne
- Jeder Kern führt gleichzeitig Instruktionen aus
- Seit ≈2006 „normal“ in Desktop-PCs
- In Smartphones nicht ungewöhnlich

[Quelle](#)

Nvidia GP104
GPU
(aus GTX 1070)

1920 (Cuda-) Kerne!



Quelle



- **299 008** CPU-Kerne
- **50 233 344** GPU-Kerne

Titan Supercomputer

[Quelle](#)

Begriffe & Motivation

- [Parallelism ≠ Concurrency](#)
- Concurrency = Nebenläufigkeit (Verzahnung unabhängiger Prozesse)

Warum Multithreading?

„Doing a lot of things at once“

- Hardware bietet mehrere „Kerne“
- Höchste Leistung: Alle Kerne gleichzeitig nutzen

Warum Concurrency?

„Dealing with a lot of things at once“

- Gewisse Prozesse konzeptuell unabhängig voneinander
- Sollte in Programmiersprache entsprechend abgebildet werden

Thread & Prozess

- Im Betriebssystem: mehrere *Prozesse*
 - Eigener Speicher (virtuelle Adressen)
 - „Wir sind der einzige Prozess“
 - Eigene {Datei, Socket, Gerät, ...}-Handles
 - Stark von anderen *Prozessen* isoliert → Erstellung dauert lange
- Jeder *Prozess*: möglicherweise mehrere *Threads*
 - Teilen sich: Handles, Speicher, ...
 - Eigener Stack (im gleichen virtuellen Adressbereich), eigene Register, ...
 - Schwach isoliert → Können schnell erstellt werden

- *Scheduler* des Betriebssystems teilt Threads auf physikalische CPUs auf
- Mehr Threads als Kerne möglich!
- Jeder Thread bekommt bestimmte „CPU-Zeit“

Thread in Rust starten

```
use std::thread;

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

- Nimmt Funktionsding ohne Argumente
- Funktionsding wird in neuem Thread ausgeführt

```
$ rustc hello.rs
$ ./hello
$
```

Keine Ausgabe

Warum?



Alle Threads werden beendet, wenn sich der „*main thread*“ beendet

Thread in Rust starten.. und warten

```
use std::thread;

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

    // waits for thread to finish
    handle.join();
}
```

```
$ rustc hello.rs
$ ./hello
Hello from another
thread!
$
```

- `spawn()` gibt `JoinHandle` zurück
- `join()` wartet auf Thread
- Wird normalerweise *detached*

Neuer Thread kann länger leben
als „Eltern-Thread“!
(Ausnahme: Eltern-Thread = *main thread*)



Länger leben als Eltern

```
fn main() {
    thread::spawn(t1);
    thread::sleep(Duration::from_millis(500));
}

fn t1() {
    thread::spawn(t2);
    println!("Bye T1 ❤️");
}

fn t2() {
    thread::sleep(Duration::from_millis(100));
    println!("Bye T2 💔");
}
```

```
$ rustc hello.rs
$ ./hello
Bye T1 ❤️
Bye T2 💔
$
```

- `sleep()` lässt aktuellen Thread warten
 - Nie `sleep()` statt `join()`!

Ergebnis zurückgeben

Kommunikation in der Praxis oft anders als hier!

```
let handle = thread::spawn(|| {
    // expensive operation here...
    // return result from closure:
    1 + 1
});

// another expensive operation...
let result_a = 1 + 2;

// collect result from other thread
let result_b = handle.join().unwrap();

// We executed two expensive operations
// in parallel!
```

- Funktionsding kann etwas zurückgeben
- Ergebnis durch **join()** erhalten

```
impl<T> JoinHandle<T> {
    fn join(self) -> Result<T> { ... }
}
```

Wann **Err(...)**?

- ➔ Wenn Thread panic't
- Panics beenden nur Thread, nicht Prozess

Variablen von außen

- Kann nicht Eltern-Stackframe referenzieren
- Ownership übernehmen!

```
let input = read_from_user();
let handle = thread::spawn(|| {
    is_prime(input)
});

// Doing stuff in the meantime...

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

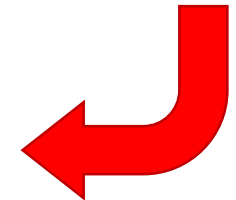
```
error[E0373]: closure may outlive the current function, but it borrows
`input`, which is owned by the current function
```

```
--> t.rs:7:32
```

```
7 |     let handle = thread::spawn(|| {
8 |         is_prime(input)
      |         ^^^^^ `input` is borrowed here
      |         ^^ may outlive borrowed value `input`
```

```
help: to force the closure to take ownership of `input` (and any other
referenced variables), use the `move` keyword, as shown:
```

```
|     let handle = thread::spawn(move || {
```



Lösung

Wie schafft der Compiler das?

```
fn foo<F>(f: F)
  where F: FnOnce()
{}

fn bar<F>(f: F)
  where F: FnOnce() + 'static
{}

let local_var = 3;
// works:
foo(|| println!("{}", local_var));
// error:
bar(|| println!("{}", local_var));
```

Aus letztem Kapitel:

```
fn foo<T: 'static>(...) { ... }
```

- Variable vom Typ **T** ist für immer gültig
 - Referenziert nichts, was nicht für immer lebt
-
- Funktionsding muss für immer leben können!

Variable von außen teilen

```
let large_string = read_from_user();  
  
let handle = thread::spawn(|| {  
    number_of_words(&large_string)  
});  
  
let a = number_of_sentences(&large_string);  
  
// print both results...
```

Fehler

- Aus beiden Thread nutzen
- Wir wollen String nicht klonen
- String erst nach letztem Thread droppen!



Reference Counted Smart Pointer!

→ **Arc<T>** ←

Variable von außen teilen

```
let large_string = read_from_user();  
  
// Move string into Arc. `for_me` is a handle  
// to the string (which lives on the heap now).  
let for_me = Arc::new(large_string);  
  
// We only clone the handle, not the string.  
// This second handle is moved into the thread.  
let for_thread = for_me.clone();  
  
// Both threads can access the string immutably  
let handle = thread::spawn(move || {  
    number_of_words(&for_thread)  
});  
let a = number_of_sentences(&for_me);
```

- **Arc<T>** garantiert, dass String noch lebt, solange noch Handles leben
- Zugriff über **Deref**-Trait

Warum eigentlich nicht **Rc<T>**?

Data Race

Rust verhindert
Data Races!

Heisenbug:

“A bug that seems to disappear when one attempts to study it”

- Definition:
 - Zwei oder mehr Threads greifen gleichzeitig auf *eine Speicherstelle* zu
 - Mindestens ein Zugriff ist ein **Schreibzugriff**
 - Mindestens ein Zugriff ist **nicht synchronisiert**

- Bugs durch Data Races höchst indeterministisch
- Schlecht zu debuggen

```
peter.money += amount;
```

```
; amount is in rbx  
mov rax, [peter_location]  
add rax, rbx  
mov [peter_location], rax
```

```
// money = 10, T1.amount = 3, T2.amount = 4
```

```
T1: load [T1.rax = 10]
```

```
T1: add in rax [T1.rax = 13]
```

```
T2: load [T2.rax = 10]
```

```
T2: add in rax [T2.rax = 14]
```

```
T2: store [money = 14]
```

```
T1: store [money = 13]
```

Einfach mal Rc benutzen

```
// ...  
let for_me = Rc::new(large_string);  
// ...
```



- Problem:
 - RefCount wird von mehreren Threads gleichzeitig verändert
 - **Arc** synchronisiert Veränderung

```
error[E0277]: the trait bound `Rc<String>: std::marker::Send` is not satisfied  
--> t.rs:12:18  
   |  
12 |     let handle = thread::spawn(move || {  
   |                               ^^^^^^^^^^^^^^^^^ the trait `Send` is not implemented for `Rc<String>`  
   |  
= note: `Rc<String>` cannot be sent between threads safely  
= note: required because it appears within the type `[closure@t.rs:12:32: 14:6  
         for_thread:Rc<String>]`  
= note: required by `std::thread::spawn`
```

Das **Send**-Trait

“Types that can be transferred across thread boundaries safely”

- Marker Trait
- Wird automatisch für Typen implementiert, die nur aus **Send**-Typen bestehen
 - Nicht via `#[derive(...)]`, sondern ganz automatisch
- Für alle primitiven Typen implementiert
- *Nicht* für Raw-Pointer implementiert!
- Versenden zwischen Threads: Nur ein Thread besitzt Wert zu einem gegebenen Zeitpunkt!

Wer implementiert **Send**?

- u8 ✓
- Vec<T> ✓ where T: **Send**
- Rc<T> ✗
- &T ✓ where T: **Sync**

Das **Sync**-Trait

“Types for which it is safe to share references between threads”

- Auch Marker Trait
- Wird auch automatisch für Typen implementiert, die nur aus **Sync**-Typen bestehen
- Auch für alle primitiven Typen implementiert
- Auch *nicht* für Raw-Pointer implementiert!
- „T ist **Sync**, genau dann wenn **&T Send** ist“

Send & Sync

- Ein Typ **T** implementiert *nicht Sync*, wenn:
 - **T** unsynchronisierte, interior Mutability (via **&T**) zulässt
- Ein Typ **T** implementiert *nicht Send*, wenn
 - **T** sich Speicher mit einem anderen **T**-Objekt teilt
 - **T** unsynchronisierte Veränderung (via **&mut T** oder **&T**) auf diesen gemeinsamen Speicher zulässt

Wer implementiert was?

	Send	Sync
• u8	✓	✓
• Vec<T>	✓ where T: Send	✓ where T: Sync
• Rc<T>	✗	✗
• &T	✓ where T: Sync	✓ where T: Sync
• &mut T	✓ where T: Send	✓ where T: Sync
• RefCell<T>	✓ where T: Send	✗
• Arc<T>	✓ where T: Send + Sync	✓ where T: Send + Sync


Signatur von spawn()

```
fn spawn<F, T>(f: F) -> JoinHandle<T>  
    where F: FnOnce() -> T + Send + 'static,  
          T: Send + 'static
```

- Funktionsding muss von Eltern-Thread zu Kind-Thread geschickt werden
- Ergebnis muss von Kind-Thread zu Eltern-Thread geschickt werden
- Beide müssen für immer leben

Mutex

(mutual exclusion)

- Implementiert **Send** wenn T: **Send**
- Implementiert **Sync** wenn T: **Send** 
- Erlaubt interior Mutability, **aber** synchronisiert!
 - Also keine Data Races!
- Durch **lock()** exklusiven Zugriff auf inneren Wert
 - Funktion kommt erst zurück, sobald Zugriff auf Wert gesichert wurde
 - Mit **try_lock()** testen, ob exklusiver Zugriff zurzeit möglich
 - Wird nie blocken
- Guards nutzen RAII-Pattern

Mutex Beispiel

```
let m = Mutex::new(vec![]);

// Send mutex to other threads (probably inside of an `Arc`).
{
    // We lock the mutex here: `lock()` will wait/block
    // until the lock has been acquired.
    let guard = m.lock().unwrap();

    // While we have the `LockGuard` in scope, we know that we are
    // the only one with access to the underlying value.
    guard.push(27);
    guard[0] += 9;
}

// The guard was dropped: now other threads can access the value again.
```

RwLock

- Wie **Mutex**, erlaubt aber genaueren Zugriff
- Durch **write()** exklusiven Zugriff (wie **Mutex**)
- Durch **read()** Lesezugriff
 - Weitere Lesezugriffe gleichzeitig möglich!
- Sinnvoll, wenn oft nur gelesen wird

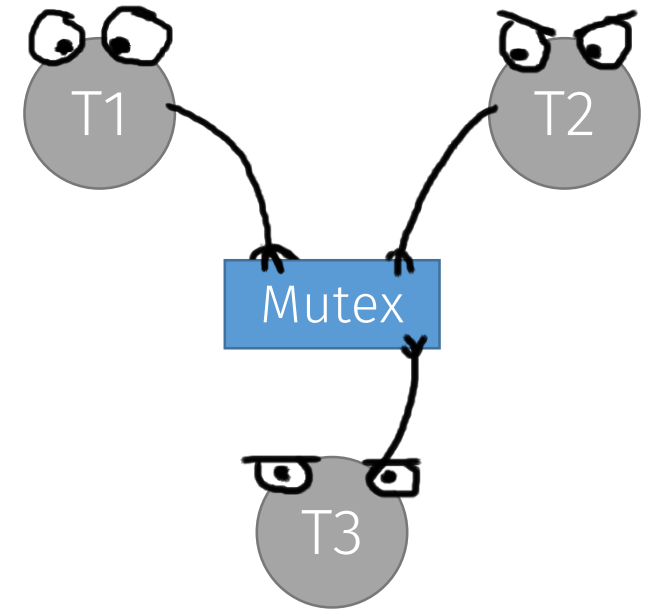
```
let rwlock = RwLock::new(vec![]);
{
    let guard = rwlock.read().unwrap();
    println!("{}", *guard);
    // other threads could be reading right now, too
}
```

Poisoning

Warum **Result**?

```
fn lock(&self) -> Result<LockGuard<T>, ...>  
// [...]  
fn read(&self) -> Result<RwLockReadGuard<T>, ...>  
fn write(&self) -> Result<RwLockWriteGuard<T>, ...>
```

- Thread könnte panic'en während er den Lock besitzt
- Panic: Unerwarteter Fehler/Bug!
 - ➔ Geteilte Daten eventuell korrupt/inkonsistent
- Mutex wird „vergiftet“, **lock()** führt zu **Err**
- Daten können trotzdem durch Umwege wiederhergestellt werden
- **unwrap()**: Panic „weiterleiten“, oft OK



Poisoning

Warum **Result**?

```
fn lock(&self) -> Result<LockGuard<T>, ...>  
// [...]  
fn read(&self) -> Result<RwLockReadGuard<T>, ...>  
fn write(&self) -> Result<RwLockWriteGuard<T>, ...>
```

- Thread könnte panic'en während er den Lock besitzt
- Panic: Unerwarteter Fehler/Bug!
 - ➔ Geteilte Daten eventuell korrupt/inkonsistent
- Mutex wird „vergiftet“, **lock()** führt zu **Err**
- Daten können trotzdem durch Umwege wiederhergestellt werden
- **unwrap()**: Panic „weiterleiten“, oft OK



Threads isolieren!

Atomics

- Interior Mutability, ähnlich zu `Cell<T>`
 - Lesen mit `load()`
 - Schreiben mit `store()`
 - Und weitere nützliche Operationen...
- Nur für wenige primitive Typen
 - `AtomicBool`, `AtomicUsize`, `AtomicIsize`, ...
- Schneller als `Mutex/RwLock`
 - Ist oft Grundbaustein für `Mutex/...`
 - `Arc<T>` nutzt `AtomicUsize`

Moment mal...
Wozu `AtomicBool`?

Atomic Ordering

```
fn load(&self, order: Ordering) -> usize
fn store(&self, val: usize, order: Ordering)
```

- Alle Operationen mit **Ordering** Parameter
- Legt fest, wie stark der Compiler/die CPU Instruktionen umordnen darf
 - Ohne Umordnen u.ä. wäre **AtomicBool = bool**
- [Versteht niemand](#), erstmal lieber **SeqCst** nutzen!
 - Das „langsamste“
 - **Relaxed, Acquire** und **Release** schneller, aber „gefährlich“

Channel

Bis jetzt:

Objekte konzeptuell
geteilt



Oft besser:

Objekte konzeptuell zwischen
Threads *hin und her schicken*

- Objekte über Kanal schicken
- Nur ein Thread besitzt Objekt zu einem Zeitpunkt
- Oft:
 - Arbeitsaufträge schicken
 - Ergebnisse empfangen

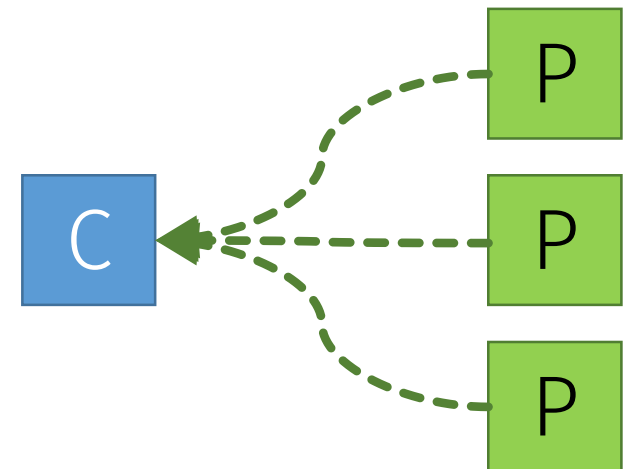
```
// (transmitting & receiving end)  
let (tx, rx) = channel();  
thread::spawn(move || {  
    // `tx` was moved to this thread  
    tx.send(10).unwrap();  
});  
// `rx` stayed outside to receive  
assert_eq!(rx.recv().unwrap(), 10);
```

Channel

```
fn channel<T>() -> (Sender<T>, Receiver<T>)
```

- In **std**: *MPSC*
 - „Multi Producer, Single Consumer“
 - **Sender** implementiert **Clone**, **Receiver** nicht
 - FIFO
- **send()** und **recv()** geben **Result** zurück
 - **Err** bei **send()** wenn Receiver zerstört wurde
 - **Err** bei **recv()** wenn alle Sender zerstört wurden
 - Falls unerwartet, mit **unwrap()** Fehler weiterleiten
- **Receiver** kann als Iterator genutzt werden

```
fn recv(&self)  
    -> Result<T, RecvError>  
  
fn send(&self)  
    -> Result<T, SendError<T>>
```



Channel Beispiel

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

// Start a few threads
for _ in 0..8 {
    // one clone for each thread
    let tx = tx.clone();
    thread::spawn(move || {
        ...
    });
}

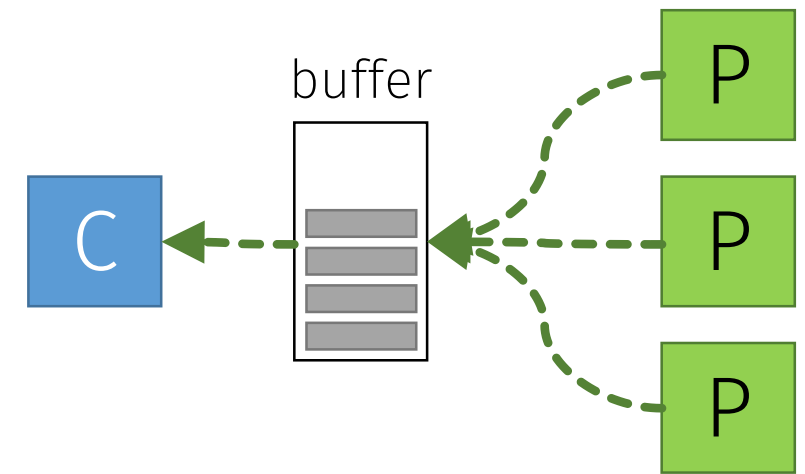
// Receiver can be used as iterator! :)
for prime in rx {
    println!("found one: {}", prime);
}
```

```
loop {
    let n = random();
    if is_prime(n) {
        // Send prime number to the
        // main thread. If the main
        // thread dropped the receiver
        // we will just panic.
        tx.send(n).unwrap();

        // Slow doooooown!
        thread::sleep(...);
    }
}
```

[Code @ Playground](#)

Blocking & sync channel



- `recv()` blockt: wartet auf Daten
- `Sender::send()` blockt nie: Theoretisch unendlicher Buffer
- Channel mit begrenztem Buffer: `sync_channel(buffer_size)`
 - `SyncSender::send()` blockt wenn Buffer voll
- Wenn `buffer_size == 0` → „rendezvous channel“
 - Beide Seiten müssen gleichzeitig „anwesend“ sein, um Daten auszutauschen
- `try_{recv(), send()}` testen, ob Operation möglich: Blocken nie

Weitere Anwendungsbeispiele

- Spezieller Fall: Einfache Synchronisierung von Threads
 - Senden von () durch den Kanal
 - Sender kann Empfänger Signal (ohne weitere Informationen) schicken
 - Empfänger wartet auf Signal, wacht dann auf
- Aus dem Computergrafikpraktikum 2016:
 - Spiel ähnlich zu Minecraft
 - Welt muss generiert werden (kann einige Zeit kosten)
 - Hauptthread (u.a. fürs Rendering zuständig) schickt Aufträge
 - Generierungsthread nimmt Aufträge an und generiert Chunks
 - Fertige Chunks werden zurück an den Hauptthread geschickt

Rust's Philosophie bei Concurrency


- Kernsprache möglichst mächtig/ausdrucksstark [\[Talk Empfehlung\]](#)
 - Alles weitere als Library implementieren
 - Bezogen auf alle Bereiche, nicht nur Concurrency
- Früher: *Green Threads* in Kernsprache (\approx wie in Go)
 - Lightweight Threads, zusätzlicher Scheduler in User-Space (d.h.: Runtime)
 - Wurden entfernt, da Runtime für Systemprogrammierung nicht akzeptabel
- Stattdessen:
 - Externe Crates: [crossbeam](#), [chan](#), [rayon](#), ...
 - Bald: Async IO & Futures

Crossbeam

- Scoped Thread API
 - Erlaubt Zugriff auf Stack des Elternthreads
- Non-blocking Datenstrukturen
 - Deutlich schneller als **Mutex<Vec<...>>** in stark parallelen Situationen
- Work Stealing Queue
 - Vergleichbar mit *Single Producer, Multiple Consumer* Channel

```
// Look ma, no Arcs!  
let big_string = read_input();  
crossbeam::scope(|scope| {  
    scope.spawn(|| {  
        count_words(&big_string);  
    });  
    count_sentences(&big_string);  
});
```

```
// Back then in Rust's std  
let big_string = read_input();  
let handle = thread::scoped(|| {  
    count_words(&big_string);  
});  
count_sentences(&big_string);  
// handle is dropped: joining
```



Rayon

A data parallelism library

Parallele Iteratoren

- Iterator Chains einfach parallel ausführen

join() Funktion:

- Divide&Conquer Probleme parallel ausführen

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {  
    paths.iter()  
        .map(|path| Image::load(path))  
        .collect()  
}
```



```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {  
    paths.par_iter()  
        .map(|path| Image::load(path))  
        .collect()  
}
```

Methoden paralleler Iteratoren?

- Viele Iterator-Methoden sind sequentiell definiert
 - Funktionieren nicht ohne Weiteres parallel
- `count()` → Kann funktionieren
- `min()` → Kann funktionieren
- `collect()` → Kann funktionieren
- `fold()` → Problematisch
 - Wird zu: `reduce()`

```
[(0, 1), (5, 6)].par_iter().cloned().reduce(  
    || (0, 0), // start value  
    |a, b| (a.0 + b.0, a.1 + b.1)  
);
```


join(): Mögliche Parallelität

- `join()` nimmt zwei Funktionsdinger
 - Führt beide *möglicherweise* parallel aus
 - Arbeit wird optimal auf verfügbare physikalische CPUs aufgeteilt

```
fn quick_sort<T: PartialOrd + Send>(v: &mut [T]) {  
    if v.len() <= 1 {  
        return;  
    }  
  
    let mid = partition(v);  
    let (lo, hi) = v.split_at_mut(mid);  
    rayon::join(|| quick_sort(lo), || quick_sort(hi));  
}
```

Zusammenfassung

- Kernsprache bietet mächtige Möglichkeiten
 - **Send** und **Sync** ermöglichen *Date Race Freedom*

