Rust programming

Module C: concurrency and parallelism

Who am i?

- I’m Folkert
- work on Network Time Protocol and other systems programming things
- work on the Roc compiler (and other low-level shenanigans)

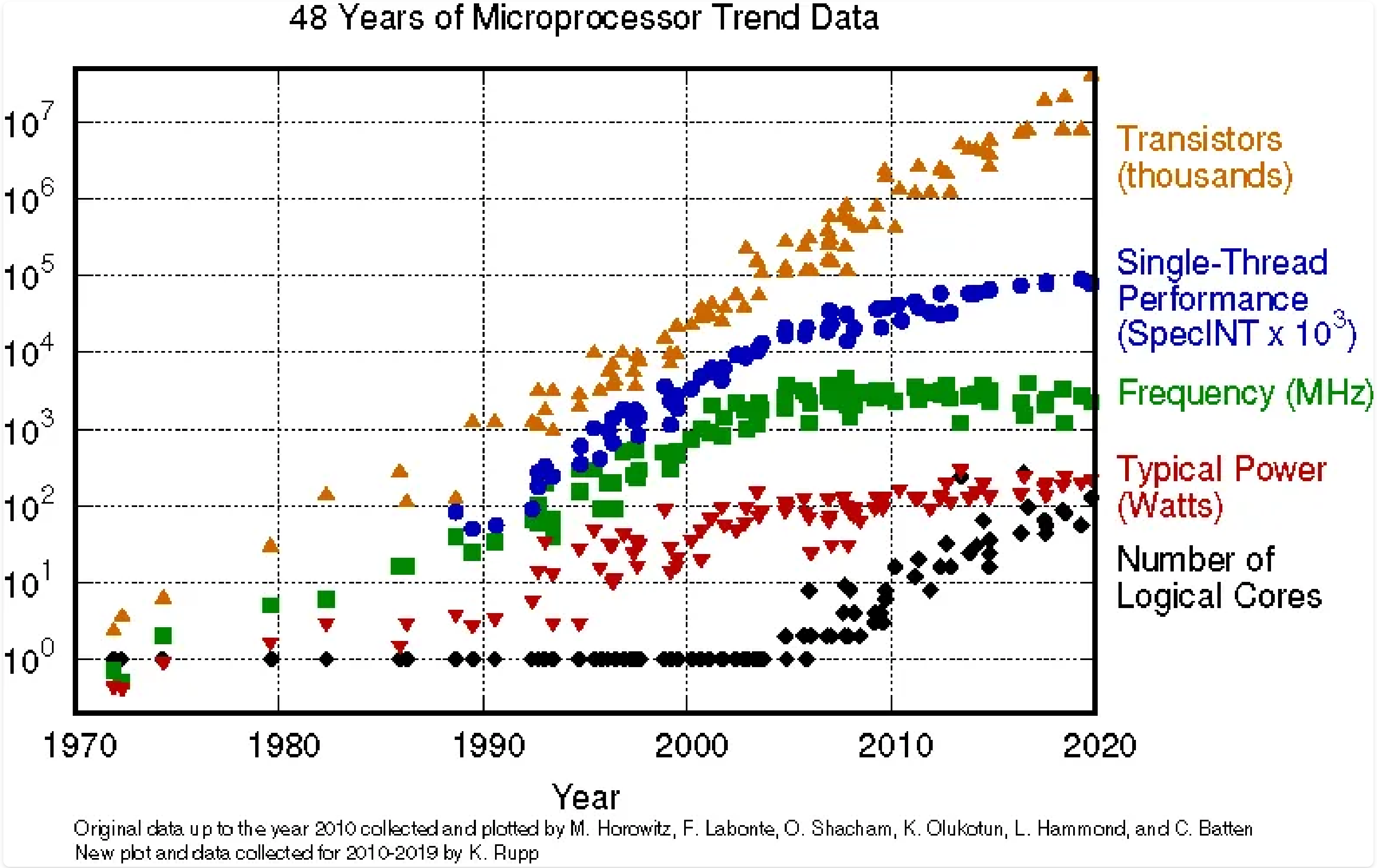
Last time...

- Cargo and dependencies
- Creating a nice API
- Testing and benchmarking
- Setting up your own project

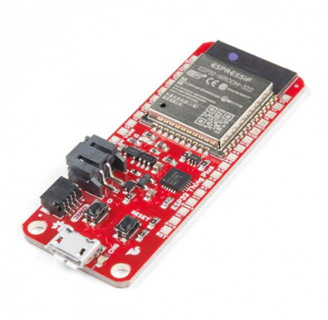
Learning objectives

after this lecture + exercises, you can:

- parallelize a program with Rayon
- work with threads in rust
- reason about exclusive access
- implement a basic Mutex


48 Years of Microprocessor Trend Data
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
Year
1970
1980
1990
2000
2010
2020
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Concurrency vs. Parallelism

Concurrency	Parallelism
Interleaves work	Parallelizes work
1 or more cores	2 or more cores
	
Waiting for events	Waiting for computation

Parallelism with Rayon

solving Pleasantly Parallel Problems

TF–IDF

An algorithm for searching in a big collection of text documents

- term frequency–inverse document frequency
- TF: "how often does a word occur in a particular document"
- IDF: "how rare is the word across all documents"

Problem:

- how do we aggregate the results?

TF–IDF in Rayon

```
use std::collections::HashMap;
use rayon::prelude::*;

fn document_frequency(documents: &[&str]) -> HashMap<&str, usize> {
    documents
        .par_iter()
        .map(|document| term_occurence(document))
        .reduce(HashMap::default, combine_occurences);
}

/// Map each word in the document to the value 1
fn term_occurence(document: &str) -> HashMap<&str, usize> {
    todo!()
}

/// combine the counts from maps a and b.
fn combine_occurences<'a>(
    a: HashMap<&'a str, usize>,
    b: HashMap<&'a str, usize>,
) -> HashMap<&'a str, usize> {
    todo!()
}
```

Combining results

The `combine_documents` function has several useful properties

- our operation is associative `a • (b • c) = (a • b) • c`
- our operation has a neutral value `HashMap::default()`: `0 • x = x • 0 = x`
- therefore we can split the computation `a • b • c • d = (0 • a • b) • (0 • c • d)`
- an associative operation with a neutral value is called a "monoid"

```
// for each word, how often it occurs across all documents
documents
    .par_iter()
    .map(|document| count_words(document))
    .reduce(HashMap::default, combine_documents);
```

- this idea means each thread can start accumulating values

Intermezzo: Closures

- Closures are anonymous (unnamed) functions
- they can capture ("close over") values in their scope
- they are first-class values

```
fn foo() -> impl Fn(i64, i64) -> i64 {
    z = 42;
    |x, y| x + y + z
}

fn bar() -> i64 {
    // construct the closure
    let f = foo();

    // evaluate the closure
    f(1, 2)
}
```

- very useful when working with iterators, `` `Option` `` and `` `Result` ``.

```
let evens: Vec<_> = some_iterator.filter(|x| x % 2 == 0).collect();
```

So far

- Closures are unnamed inline functions
- Rayon makes data-parallel programming in rust extremely convenient

Fearless concurrency

thread-based concurrency in rust

Fearless concurrency

```rust
use std::thread;

fn main() {
    thread::spawn(f);
    thread::spawn(f);

    println!("Hello from the main thread.");
}

fn f() {
    println!("Hello from another thread!");

    let id = thread::current().id();
    println!("This is my thread id: {id:?}");
}
```

- A process can spawn multiple threads of execution. These run concurrently (and may run in parallel)
- Question: what is the output of this program?

Expected output

maybe

```
Hello from another thread!
This is my thread id: ThreadId(411)
Hello from another thread!
This is my thread id: ThreadId(412)
Hello from the main thread.
```

or

```
Hello from another thread!
Hello from another thread!
This is my thread id: ThreadId(411)
This is my thread id: ThreadId(412)
Hello from the main thread.
```

Expected output

but most likely

```
Hello from the main thread.
```

The process exits when the main thread is done!

- `.join()` can be used to block the main thread until the child is done

```
fn main() {
    let t1 = thread::spawn(f);
    let t2 = thread::spawn(f);

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

    t1.join().unwrap();
    t2.join().unwrap();
}
```

- `.join()` turns a panic in the thread into an `Err`

Thread lifetime

- a more typical example

```rust
let numbers = Vec::from_iter(0..=1000);

let t = thread::spawn(move || {
    let len = numbers.len();
    let sum = numbers.iter().sum::<usize>();
    sum / len
});

let average = t.join().unwrap();

println!("average: {average}");
```

- `numbers` must be `move`d into the closure!

Thread lifetime

- otherwise `numbers` might be dropped while the thread is still using it!

```rust
let numbers = Vec::from_iter(0..=1000);

let t = thread::spawn(|| {
    let len = numbers.len();
    let sum = numbers.iter().sum::<usize>();
    sum / len
});

drop(numbers); // compile error: would create a dangling reference

let average = t.join().unwrap();

println!("average: {average}");
```

Thread lifetime: make it known

```rust
let numbers = Vec::from_iter(0..=1000);

let average = thread::scope(|spawner| {
    spawner.spawn(|| {
        let len = numbers.len();
        let sum = numbers.iter().sum::<usize>();
        sum / len
    }).join().unwrap()
});

println!("average: {average:?}");
```

- explicitly bound the lifetime with a scope
- threads are always joined at the end of that scope
- makes immutable references just work

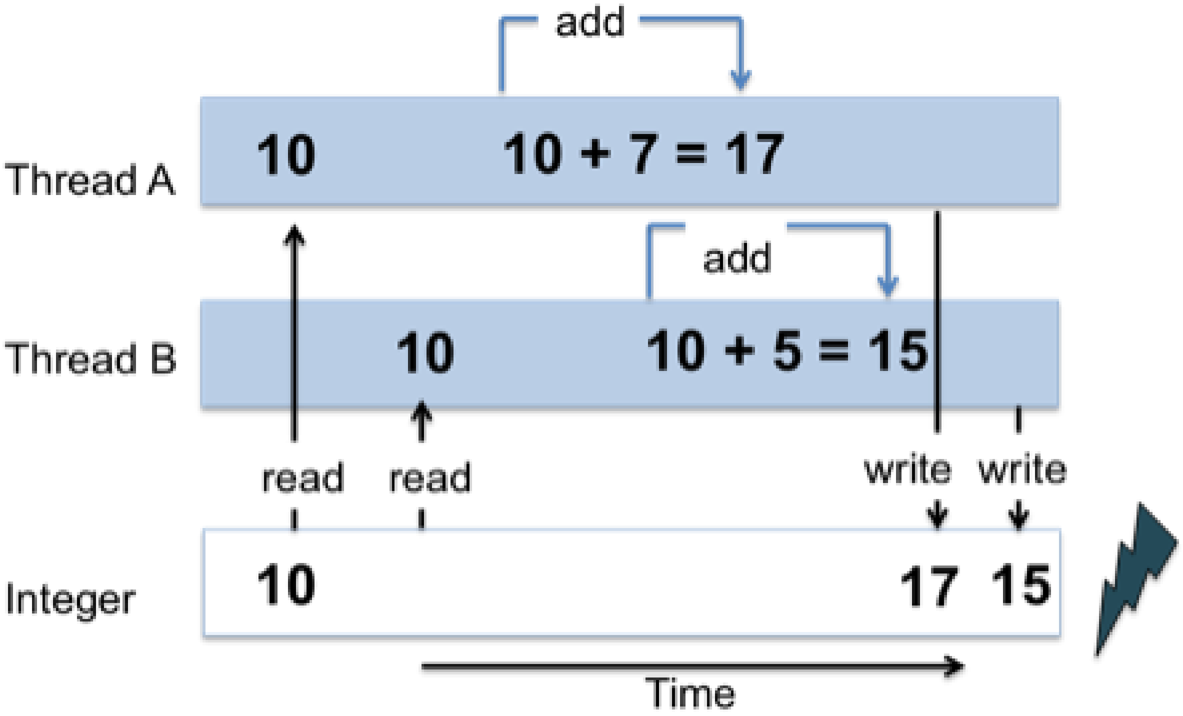
- but mutable borrowing rules still apply:

```
let mut count = 0;
let counter = &mut count;

std::thread::scope(|s| {
    s.spawn(|| { *counter = *counter + 1; });
    s.spawn(|| { *counter = *counter + 1; });
});
```

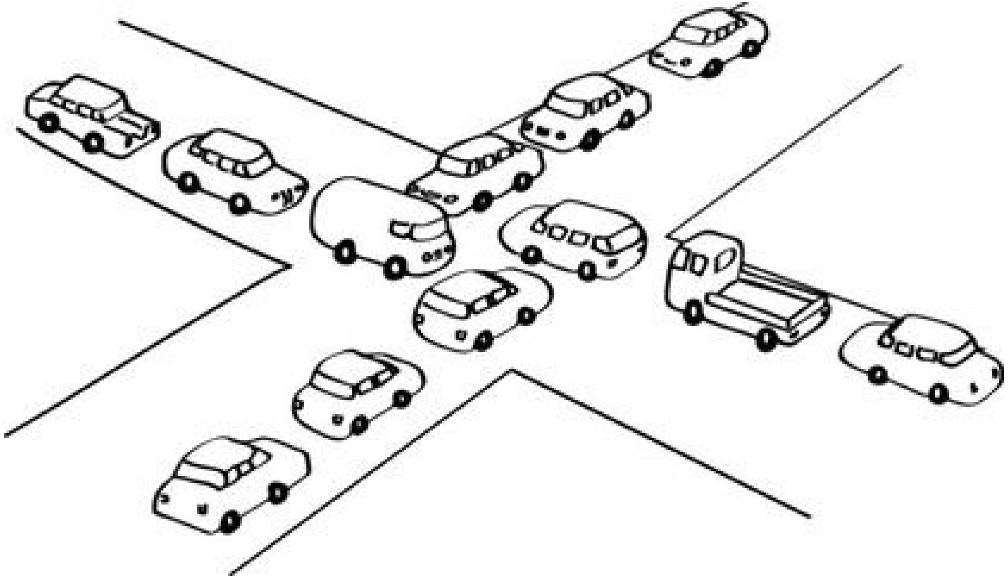
```
error[E0499]: cannot borrow `*counter` as mutable more than once at a time
6 |     thread::scope(|s| {
  |                    - has type `&'1 Scope<'1, '_>`
7 |         s.spawn(|| { *counter = *counter + 1; });
  |         ----------------------------------------
  |         |       |    |
  |         |       |    first borrow occurs due to use of `*counter` in closure
  |         |       first mutable borrow occurs here
  |         argument requires that `*counter` is borrowed for `'1`
8 |         s.spawn(|| { *counter = *counter + 1; });
  |                 ^^   -------- second borrow occurs due to use of `*counter` in closure
  |                 |
  |                 second mutable borrow occurs here
```

Race Conditions

- if multiple mutable borrows were allowed, this could happen ...



Fearless concurrency



- borrowing rules prevent data races & deadlocks
- but also any shared mutable state between threads
- many correct, useful programs are disallowed!

Re-defining references

- `&T`: (possibly) shared reference
- `&mut T`: exclusive reference

for safe mutation, we need exclusive *access*, which we can get in multiple ways:

- we have an exclusive reference to the value
- we own the value (we can exclusively borrow from ourselves)
- access is inherently exclusive (atomic operations)

Atomics

- atomic operations are indivisible, but relatively expensive

```rust
use std::sync::atomic::{AtomicU32, Ordering};

let foo = AtomicU32::new(0);
assert_eq!(foo.fetch_add(10, Ordering::SeqCst), 0);
assert_eq!(foo.load(Ordering::SeqCst), 10);
```

- no risk of a race condition: another thread cannot read the value while an atomic operation is ongoing

```rust
pub fn fetch_add(&self, val: u32, order: Ordering) -> u32
```

Mutual Exclusion

- `Mutex` allows mutation of a `T` through a shared `&Mutex<T>` reference

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

fn main() {
    let n = Mutex::new(String::from("foo"));
    thread::scope(|s| {

        s.spawn(|| { n.lock().unwrap().push_str("bar"); });

        s.spawn(|| { n.lock().unwrap().push_str("baz"); });

    });

    println!("{}", n.into_inner().unwrap());
}
```

- threads lock the mutex, but there is no `unlock` ?!

Sharing ownership between threads

```rust
impl<T> Mutex<T> {
    pub fn lock<'a>(&'a self) -> LockResult<MutexGuard<'a, T>> {
        ...
    }
}
```

- Acquires a mutex, blocking the current thread until it is able to do so
- Returns a `PoisonError` if a thread panicked while holding the lock
- Returns a `MutexGuard`, proof to the type checker that we hold the lock
- `MutexGuard<'a, T>` implements `DerefMut<Target = T>`, so we can use it like a mutable reference

```rust
impl<'a, T> DerefMut for MutexGuard<'a, T> {
    fn deref_mut(&mut self) -> &mut T {
        // ...
    }
}
```

- dropping the `MutexGuard` unlocks the mutex

Moving ownership between threads

- Some values should never be shared or moved between threads

The `Send` and `Sync` marker traits enforce this:

```
pub unsafe auto trait Send { /* no method */ }
pub unsafe auto trait Sync { /* no method */ }
```

- `Send`: A type is Send if it can be sent to another thread. In other words, if ownership of a value of that type can be transferred to another thread
- `Sync`: A type is Sync if it can be shared with another thread. In other words, a type T is Sync if and only if a shared reference to that type `&T` is Send

`Send`

- A type is Send if it can be sent to another thread. In other words, if ownership of a value of that type can be transferred to another thread

```
impl<T: ?Sized> !Send for MutexGuard<'_, T>
impl<T: ?Sized + Sync> Sync for MutexGuard<'_, T>
```

- On certain OS's, only the thread that locked a mutex may unlock it again!

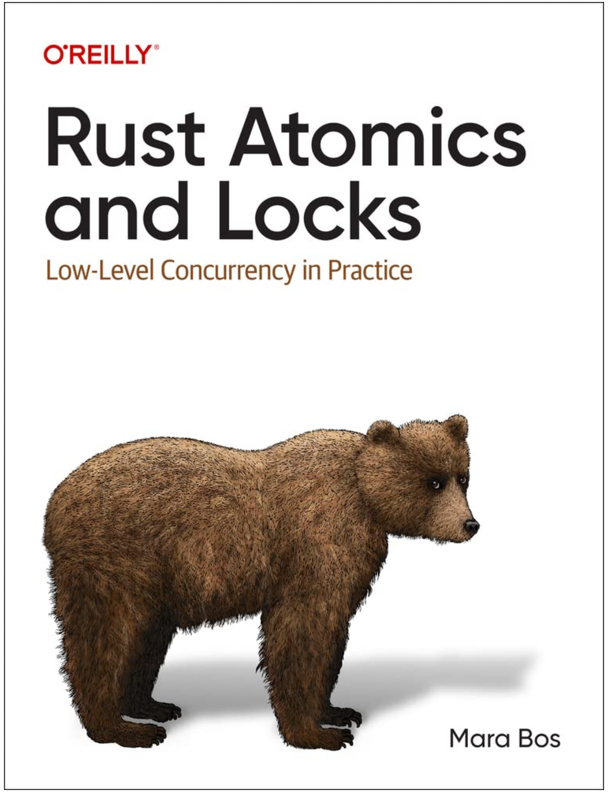
MPSC: many producer single consumer

```rust
fn main() {
    let (tx, rx) = std::sync::mpsc::channel();

    std::thread::scope(|s| {
        for (i, tx) in std::iter::repeat(tx).take(10).enumerate() {
            s.spawn(move || { tx.send(i).unwrap(); });
        }

        s.spawn(move || {
            while let Ok(msg) = rx.recv() {
                println!("{msg}");
            }
        });
    });
}
```

where the `Receiver` is:

```rust
impl<T: Send> Send for Receiver<T>
impl<T> !Sync for Receiver<T>
```

Further reading



- read for free at https://marabos.nl/atomics/

Summary

- Rayon makes parallel computation easy
- Scoped threads allow borrowing into threads
- Mutation requires exclusive access
- Some data structures guarantee exclusive access (even through a shared reference)
- The borrow checker, `Send` and `Sync` prevent many common problems