APPS@UCU

Rust #4: Smart pointers, Concurrency and Async

Sultanov Andriy

APPLIED SCIENCES FACULTY

Contents

Smart pointers

Smart pointers
General

Rust standard library has a few useful smart pointers, which modify the possibilities of operating with the value they hold:

- Box<T>
- Rc<T>
- Cell<T>
- RefCell<T>

Smart pointers

Deref trait

Implementing Deref allows Rust to treat smart pointers as references to the values they hold.

You can dereference types impementing Deref explicitly using the * operator, or you can omit the dereferencing and let Rust implicitly coerce your type to the type you need.

Thus, when calling a method on a type that implements Deref and does not implement the method, Rust will recursively dereference this type until it finds the appropriate method.

So, let's say we want to have this kind of OOP behavior in Rust:

```
class Foo {
    void m() { ... }
}

class Bar extends Foo {}

public static void main(String[] args) {
    Bar b = new Bar();
    b.m();
}
```

Smart pointers
Deref trait

We can model it like this:

```rust
use std::ops::Deref;

struct Foo {}

impl Foo {
    fn m(&self) {
        //..
    }
}

struct Bar {
    f: Foo,
}

impl Deref for Bar {
    type Target = Foo;
    fn deref(&self) -> &Foo {
        &self.f
    }
}
```

And then easily use it:

```
fn main() {
    let b = Bar { f: Foo {} };
    b.m();
}
```

This allows you to easily operate with Rust’s smart pointers.

Smart pointers
Deref trait

Rust will coerce types according to these rules:

- From &T to &U when T: Deref<Target=U>
- From &mut T to &mut U when T: DerefMut<Target=U>
- From &mut T to &U when T: Deref<Target=U>

Box<T>

Boxes are Rust’s way to store data on the heap.

They are useful when you want to handle values whose sizes can not be known at compile time as if their size was known. This works since the size of the pointer to the heap that the Box keeps on the stack is known.

Box<T>

An example of moving a type that is typically stored on the stack onto the heap:

```
let val: u8 = 42;
let boxed: Box<u8> = Box::new(val);

// We can easily dereference the box
println!("{}", boxed); // prints 42
```

The value is going to be deallocated once it goes out scope, calling Box’s drop implementation.

Box<T>

We could have just as well explicitly dereferenced the Box:

```rust
let val: u8 = 42;
let boxed: Box<u8> = Box::new(val);

// We can easily dereference the box
println!("{}", boxed); // prints 42

// And explicitly like this:
println!("{}", *boxed);
```

Box<T>

If you want to create an unsized data structure, for example a recursive cons list, you can just use Box:

```
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

// Creating a new list
let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
println!("{:?}", list); // prints Cons(1, Cons(2, Nil))
```

What will end up being stored is an i32 and a pointer usize to another List on the heap.

Rc<T>

While Rust's ownership system is pretty strict, there are some ways to have multiple ownership. Rc<T> is a reference counter that keeps track of the number of pointers to a certain value, and drops it once nobody uses it anymore.

Rc<T>

Rc<T> does not allow mutability, and basically changes the ownership system so that the control over the lifetime of the value is done during runtime, not compile time.

Rc<T>

```
let text = "Rc examples".to_string();
{
    // Reference count is 1
    let rc_a: Rc<String> = Rc::new(text);
    {
        // Reference count is 2
        let rc_b: Rc<String> = Rc::clone(&rc_a);

        rc_a.len();
        println!("{}", rc_b);

        // rc_b is dropped, reference count is 1
    }
    // rc_a is dropped, reference count is 0, the value is dropped too
}
// Error, rc_examples was dropped!
println!("rc_examples: {}", rc_examples);
```

Others

Rust also provides several other standard smart pointers, like Cell<T> and RefCell<T>, which allow mutating the immutable value in various cases.

We are not going to cover them in these lectures, but it's important to understand how all of these smart pointers work under the hood, and the way they allow us to operate with the values they hold - through Deref.

Concurrency

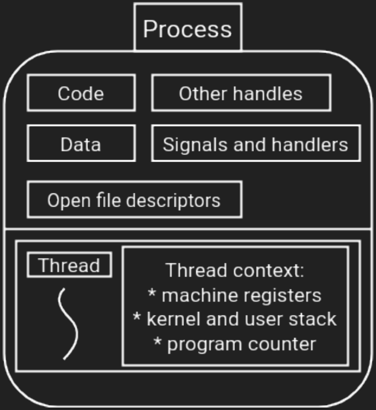
Concurrency in general

We've talked about how computers work in general before, but we've always implicitly assumed that they only run one thing at a time - that is, we've talked about a single-core CPU.

How are user programs represented for that CPU? Through the abstraction of a process - an instance of a program in execution. It might be better to think of the process as the collection of data structures that fully describes the state of the execution of a certain program.

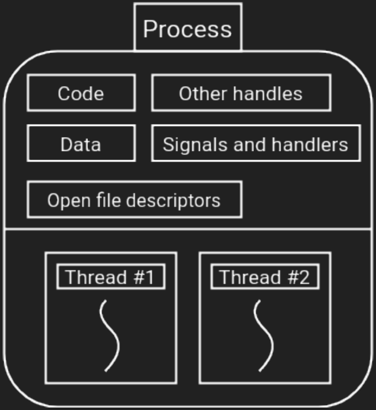
Concurrency in general

Thus, a process can roughly be represented as this, a collection of data structures that describe it, and a single thread of execution that is actually scheduled for execution and handles the state of the program.



Concurrency in general

One process can contain multiple threads of execution though, which are going to have the same shared memory and code through the process, but are going to be running different code simultaneously.



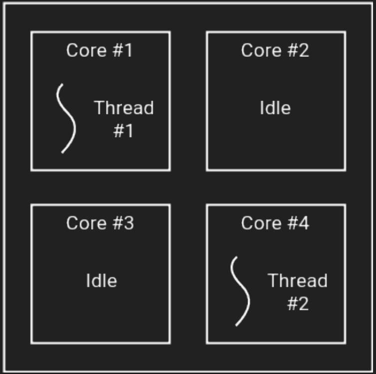
Concurrency in general

The task of scheduling and prioritizing these threads falls on the operating system, which allows creating these structures in the first place, and maintains a list of them, periodically switching the CPUs to run this or that thread.

We are not going to talk about multithreading from the OS perspective, but it’s important to understand what happens with our CPU once we have multiple cores and are able to execute several threads at a time.

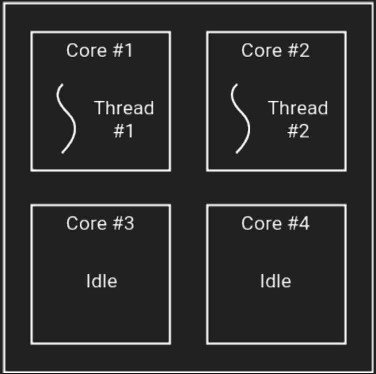
Concurrency in general

The operating system and the CPU handle all of the minor details of the process of scheduling and actually executing the threads on several cores, we are just going to assume that all of this works nicely (it almost always does)!



Concurrency in general

The operating system and the CPU handle all of the minor details of the process of scheduling and actually executing the threads on several cores, we are just going to assume that all of this works nicely (it almost always does)!



Synchronization

We've already talked about how a safe system would look like, where nobody will violate the memory safety of others:

- There can be many readers with no writers at the same time
- There can be only one writer with no readers at the same time
- Values can be used only as long as they still exist

We've seen how Rust ensures all of these rules are satisfied in a single-thread scenario, but what if there are several threads running at the same time? How do we make sure they don't write into each other's memory?

Synchronization

There are several kinds of problems which can arise in parallel multithreaded programs, where we can rarely be sure which thread runs first, and what data is accessible to each of them.

Without proper synchronization, you can get:

- Race conditions - when threads access shared data in an undefined order.
- Deadlocks and livelocks - when multiple threads can not properly agree on their behavior.

These might only arise in rare conditions and can be hard to reproduce.

Synchronization and ownership

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

fn main() {
    for i in 1..6 {
        thread::spawn(|| {
            for j in 1..10 {
                println!("thread {} - {}", i, j);
                thread::sleep(Duration::from_millis(1));
            }
        });
    }

    for i in 1..5 {
        println!("main thread - {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Why doesn’t this compile?

Synchronization and ownership

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

fn main() {
    for i in 1..6 {
        thread::spawn(move || {
            for j in 1..10 {
                println!("thread {} - {}", i, j);
                thread::sleep(Duration::from_millis(1));
            }
        });
    }

    for i in 1..5 {
        println!("main thread - {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

How do we make sure these threads finish running?

Synchronization and ownership

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

fn main() {
    let mut vec = Vec::new();

    for i in 1..6 {
        vec.push(thread::spawn(move || {
            for j in 1..10 {
                println!("thread {} - {}", i, j);
                thread::sleep(Duration::from_millis(1));
            }
        }));
    }

    for thread in vec {
        thread.join().unwrap();
    }

    for i in 1..5 {
        println!("main thread - {}", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Synchronization and ownership

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

fn main() {
    let mut vec = Vec::new();

    for i in 1..6 {
        vec.push(thread::spawn(move || {
            for j in 1..10 {
                println!("thread {} - {}", i, j);
                thread::sleep(Duration::from_millis(1));
            }
        }));
    }

    for i in 1..5 {
        println!("main thread - {}", i);
        thread::sleep(Duration::from_millis(1));
    }

    for thread in vec {
        thread.join().unwrap();
    }
}
```

Synchronization and ownership

Alright, let’s say we want our threads to modify a single variable? We can not use move closures, since this will force threads to take the variable by value and not by mutable reference.

Rust ownership model prevents us from having several mutable references to a value at the same time, and this works with threads too.

Synchronization and ownership

There are several ways of making sure that different threads are synchronized:

- Locks - before modifying the variable, just make sure you are the only thread doing so.
- Atomics - make sure nobody can interfere while you are in the middle of modifying a value.

Locking

Rust provides us with several standard synchronization primitives that can be used for this, we'll only look at several most important ones.

Mutex is a standard way of locking variables behind a lock, which requires you to acquire the lock before getting access to the value inside.

Locking

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

fn main() {
    let a = Mutex::new(42);

    {
        let mut data = a.lock().unwrap();
        *data += 1;
    }

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

Locking

Let’s try using it with several threads:

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

fn main() {
    let mut vec = Vec::new();
    let mut a = Mutex::new(42);

    for _ in 1..6 {
        vec.push(thread::spawn(move || {
            for _ in 1..10 {
                let mut data = a.lock().unwrap();
                *data += 1;
            }
        }));
    }

    for thread in vec {
        thread.join().unwrap();
    }
}
```

Locking

Remember we've talked about a smart pointer that allows multiple ownership at runtime? Let's use it!

```
use std::{rc::Rc, sync::Mutex, thread};

fn main() {
    let mut vec = Vec::new();
    let mut a = Rc::new(Mutex::new(42));

    for _ in 1..6 {
        vec.push(thread::spawn(move || {
            for _ in 1..10 {
                let mut data = a.lock().unwrap();
                *data += 1;
            }
        }));
    }

    for thread in vec {
        thread.join().unwrap();
    }
}
```

Locking

One last trick up Rust's sleeve - Send and Sync traits.

The Send trait is implemented for types the ownership of which is safe to transfer between threads. Rc<T> is not one of these, it's only implemented for single-threaded programs.

Sync instead means that it is safe for a type to be referenced from different threads, which is not okay with Rc<T>, Cell<T>, but is okay with Mutex<T>.

Locking

Let’s instead use a thread-safe reference counter, an atomic reference counter Arc<T>:

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

fn main() {
    let mut vec = Vec::new();
    let mut a = Arc::new(Mutex::new(42));

    for _ in 1..6 {
        let counter = Arc::clone(&a);
        vec.push(thread::spawn(move || {
            for _ in 1..10 {
                let mut data = counter.lock().unwrap();
                *data += 1;
            }
        }));
    }

    for thread in vec {
        thread.join().unwrap();
    }
}
```

Atomics

Rust also provides atomic variants of the primitive types: AtomicBool, AtomicU16 and so on.

These guarantee that an operation is not interrupted by a different thread, and expose a different interface from the non-atomic single-thread primitives.

Ensuring that a variable is atomic, that an operation on it either happens or doesn't at all, with no states in-between is pretty expensive, so use these types only when you need them!

Atomics

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

fn main() {
    let a = Arc::new(AtomicUsize::new(1));

    let clone = Arc::clone(&a);
    let thread = thread::spawn(move || {
        clone.fetch_add(1, Ordering::Relaxed);
    });

    if let Err(panic) = thread.join() {
        println!("Thread had an error: {:?}", panic);
    }
}
```

Channels

There are, of course, easier ways of communicating between threads. Rust provides a standard implementation of a multiple-producer single-consumer thread-safe channel with which threads can communicate with each other.

Channels

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

// MPSC Channel returns two ends:
// * tx - a transmitter, which can be cloned
// * rx - a receiver, which can't be cloned
let (tx, rx) = channel();

thread::spawn(move|| {
    tx.send(10).unwrap();
});

println!("{}", rx.recv().unwrap());
```

Channels

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

let (tx, rx) = channel();

for i in 0..10 {
    let tx = tx.clone();
    thread::spawn(move|| {
        tx.send(i).unwrap();
    });
}

for _ in 0..10 {
    let j = rx.recv().unwrap();
    println!("{}", j);
}
```

Looking back

Looking back

Over the course of four weeks we’ve covered some general topics, which I hope you will be able to apply everywhere, not only in Rust:

- Machine memory model
- Safe mutability rules
- Safe ownership and lifetime rules
- Multithreading

Looking back

And we’ve also looked at quite a lot of Rust-specific stuff:

- Rust’s ownership/lifetime/mutability system
- Cargo ecosystem
- Algebraic types and error handling
- Declarative macros
- Traits
- Iterators and closures
- Multithreading basics

And ahead...

Looking ahead

There is quite a lot of stuff in Rust that we haven't covered. If you want to dive deeper yourself, here is a short list of some of the most important things you can look into:

- Async Rust
- Procedural macros
- Testing
- FFI and bindings

And many more...

Looking ahead

But, most importantly, practice makes perfect!

I hope that this short tour of Rust has shown you the possibilities of modern languages and ecosystems, and taught you a few important things in general!

Thank you!