

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

# Rust and Multithreading

Jacques-Henri Jourdan

February 7th, 2023

## Preliminaries

## Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex&lt;T&gt;

# Abstract from last weeks

During the two last weeks, we learned:

- Basics of the Rust language.
- Type system enforces: “mutation XOR aliasing”,
  - with the use of lifetimes.
- Traits: an abstraction mechanism, sometimes at zero-cost.
- Unsafe blocks/functions: workaround strong static type-checking constraints,
  - but correct unsafe code: very subtle.
- Encapsulation: clients can safely use libraries written with unsafe code.
- Interior mutability (the ability to mutate through shared borrows): a typical example of well-encapsulated unsafe code.

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

## How to write correct and efficient code with shared-memory concurrency in Rust?

## 1 Preliminaries

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

## 2 Threads in Rust

- Type of spawn?
- Send and Sync
- Type of spawn

Inter-thread  
communication

Mutex<T>

## 3 Inter-thread communication

- Mutex<T>

# A note on vocabulary

What is concurrency? What is parallelism?

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

**Preliminaries****Threads in Rust**

Type of spawn?

Send and Sync

Type of spawn

**Inter-thread  
communication**

Mutex&lt;T&gt;

# A note on vocabulary

What is concurrency? What is parallelism?

## Concurrency

When **several tasks** are executed at the same time, sharing a **common resource**.

- The focus is on organizing the use of the common resource
  - Example: scheduling, mutexes, ...

## Parallelism

When **several computations units** are used to execute one or several tasks.

- The focus is on the improvement of **performances**.

## Questions:

- Example of a parallel but not concurrent program?
- Example of a concurrent but not parallel program?

# Threads

A **OS abstraction** allowing a program to create several tasks:

- running at the same time,
- sharing the same memory space.

Is this a concurrency or a parallelism concept?

# Threads

A **OS abstraction** allowing a program to create several tasks:

- running at the same time,
- sharing the same memory space.

A **concurrency** concept, providing parallelism.

- Initially used to design UIs (not for performances).

# Threads

A **OS abstraction** allowing a program to create several tasks:

- running at the same time,
- sharing the same memory space.

A **concurrency** concept, providing parallelism.

- Initially used to design UIs (not for performances).

A threading library traditionally consists in:

- A way to create new threads, by providing a function to run (i.e., `pthread_create`).
- Synchronization primitives: organize the sharing of common resources (such as memory)
  - E.g., mutexes, atomics, ....

A OS abstraction allowing a program to create several tasks:

- running at the same time,
- sharing the same memory space.

A concurrency concept.

Note: there also are language-level abstractions for concurrency.

- Initially used in C/C++ (e.g., `fork()`, `pthread_create()`)
- (In Rust: futures)

They do not provide (multicore) parallelism on their own.

This is not the scope of this course.

A threading library.

- A way to create threads (e.g., `fork()`, `pthread_create()`).
- Synchronization primitives: organize the sharing of common resources (such as memory)
  - E.g., mutexes, atomics, ....

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread

communication

Mutex<T>

# Programming with threads

Rust and  
Multithreading

Jacques-Henri  
Jourdan

Why is efficiently programming with threads difficult?

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

# Programming with threads

Rust and  
Multithreading

Jacques-Henri  
Jourdan

In C++:

```
int main() {
    int x = 0;
    std::thread t([&](){x += 1;});
    x += 1;
    t.join();
}
```

This is a **data race**:

- two **potentially simultaneous** memory accesses,
- using an **ordinary variable** (i.e., non-atomic),
- in **two different threads**,
- one of them is a **write**.

This is **undefined behavior**. The program can do anything.  
Common bug, subtle to find.

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

## 1 Preliminaries

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

## 2 Threads in Rust

- Type of spawn?
- Send and Sync
- Type of spawn

## 3 Inter-thread communication

- Mutex<T>

# A thread spawning function

Could you propose a type for a thread spawning function?

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

# A thread spawning function

Rust and  
Multithreading

Jacques-Henri  
Jourdan

Proposal 1:

```
fn spawn<F>(f: F) where F: FnOnce() -> ()
```

This is **unsound**. Why?

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

# A thread spawning function

Proposal 1:

```
fn spawn<F>(f: F) where F: FnOnce() -> ()
```

This is **unsound** because **F** could contain references to the current stack frame:

```
fn f() {
    let mut x = 0;
    spawn(||{
        let dt = std::time::Duration::from_millis(100);
        std::thread::sleep(dt);

        // The closure contains a borrow to x in the spawner's stack frame.
        // So this results in dereferencing a dangling borrow!
        x += 1
    });
}
```

# A thread spawning function

Proposal 2:

```
fn spawn<F>(f: F) where F: FnOnce() -> (),  
    F: 'static
```

- The constraint `F: 'static` means the *type* of the closure `F` should outlive `'static`.
- `'static` is the never ending lifetime.
- So `F: 'static` means that the type `F` should be always valid.
- In particular, `F` cannot contain any “true” borrow.

In order to prevent the compiler to use borrows in closures: `move` keyword:

```
spawn(move ||{ .... })
```

Captured variables are moved/copied in the closure instead of being borrowed.

# A thread spawning function

Proposal 2:

```
fn spawn<F>(f: F) where F: FnOnce() -> (),  
    F: 'static
```

Is this sound?

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

# A thread spawning function

Proposal 2:

```
fn spawn<F>(f: F) where F: FnOnce() -> (),  
    F: 'static
```

This is **not sound**.

Because of interior mutability, we can create data races:

```
fn f() {  
    let x = Rc::new(Cell::new(42));  
    let y = x.clone();  
    // x and y are pointers referring to the same Cell  
    std::thread::spawn(move ||{  
        y.set(12);  
    });  
    x.set(13);  
}
```

We should not be allowed to share pointers to `Cell` across threads!

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

[Preliminaries](#)[Threads in Rust](#)[Type of spawn?](#)[Send and Sync](#)[Type of spawn](#)[Inter-thread  
communication](#)[Mutex<T>](#)

# The Sync trait

Sharing pointers (e.g., `&T` or `Rc<T>`) to a type across threads is not benign.

It should **not** be allowed for some types (e.g., `Cell`).

# The Sync trait

Sharing pointers (e.g., `&T` or `Rc<T>`) to a type across threads is not benign.  
It should **not** be allowed for some types (e.g., `Cell`).

There is a **type trait** for this: `Sync`.

- `T: Sync` means that it is safe to share a pointer to `T` across threads.
- It is an **unsafe trait**: it is unsafe to implement it by hand.
- It is an **auto trait**: when this makes sense, Rust implements it automatically.
  - Example: `struct S<T, U> { t: T, u: U }` is `Sync` iff both `T` and `U` are `Sync`.
  - Closures are `Sync` iff their components are `Sync`.

Examples?

# The Sync trait

Sharing pointers (e.g., `&T` or `Rc<T>`) to a type across threads is not benign.  
It should **not** be allowed for some types (e.g., `Cell`).

There is a **type trait** for this: `Sync`.

- `T: Sync` means that it is safe to share a pointer to `T` across threads.
- It is an **unsafe trait**: it is unsafe to implement it by hand.
- It is an **auto trait**: when this makes sense, Rust implements it automatically.
  - Example: `struct S<T, U> { t: T, u: U }` is `Sync` iff both `T` and `U` are `Sync`.
  - Closures are `Sync` iff their components are `Sync`.

Examples:

- `i32`, `Box<i32>`, `&mut i32` are `Sync`.
  - A shared pointer to them only provides read access.
- `Cell<i32>` is **not Sync**.
  - A shared pointer can be used for writing.
- `Rc<i32>` and `RefCell<i32>` are **not Sync**.
  - Accesses to the internal counter would be racy.

# The Send trait

Similarly, it is not always safe to **move** a value to another threads.

There is a type trait for this: **Send**.

- **T: Send** means it is safe to move a value of type **T** to another thread.
- As **Sync**, this is an **auto** and **unsafe** trait.

Examples?

# The Send trait

Similarly, it is not always safe to **move** a value to another threads.

There is a type trait for this: **Send**.

- **T: Send** means it is safe to move a value of type **T** to another thread.
- As **Sync**, this is an **auto** and **unsafe** trait.

Examples:

- We have **T: Sync** iff **&T: Send**.
  - Specific instance of **Send** for **&T** in standard library.
  - So **&Cell<T>** is not **Send**.
- **i32**, **Box<i32>**, **&mut i32** are **Send**.
  - They don't have anything specific to one thread.
- **Rc<T>** is **never Send**.
  - Access to the internal counter would be a data race.
- **Cell<T>** and **RefCell<T>** are **Send** when **T: Send**.
  - When we have full ownership, nobody can access the counter.

# Send and Sync: recap

Rust and  
Multithreading  
Jacques-Henri  
Jourdan

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

Inter-thread  
communication

Mutex<T>

T: `Send` means we can move an object of type T to another thread.

T: `Sync` means we can share an object of type T with another thread.

- This is equivalent to &T: `Send`.

# A thread spawning function

Proposal 3:

```
fn spawn<F>(f: F) where F: FnOnce() -> (),  
    F: Send + 'static
```

We require the closure to be `Send` and `'static` so that the captured values:

- cannot refer to the spawner's call frame;
- cannot contain possibly-terminating borrows;
- can be safely moved to another thread.
  - In particular, `Rc<RefCell<i32>>` is not `Send`!

# A thread spawning function

Proposal 3:

```
fn spawn<F>(f: F) where F: FnOnce() -> (),  
    F: Send + 'static
```

We require the closure to be `Send` and `'static` so that the captured values:

- cannot refer to the spawner's call frame;
- cannot contain possibly-terminating borrows;
- can be safely moved to another thread.
  - In particular, `Rc<RefCell<i32>>` is not `Send`!

This is **sound**, but misses an important feature: get the result of concurrent computation.

```
struct JoinHandle<T> { ... }

pub fn spawn<F, T>(f: F) -> JoinHandle<T> where F: FnOnce() -> T,
                                              F: Send + 'static,
                                              T: Send + 'static { ... }

impl<T> JoinHandle<T> {
    /* Result<T> is like Option<T>. An error is returned if the thread panics. */
    pub fn join(self) -> Result<T>
    { ... }
}
```

## 1 Preliminaries

Preliminaries

Threads in Rust

Type of spawn?

Send and Sync

Type of spawn

## 2 Threads in Rust

- Type of spawn?
- Send and Sync
- Type of spawn

Inter-thread  
communication

`Mutex<T>`

## 3 Inter-thread communication

- `Mutex<T>`

# How can threads communicate?

So far, communication is limited to creation and termination of a thread.  
But Rust also provides means of communication:

- concurrent FIFO queues,
- [Mutex](#), [RwLock](#),
- [Arc](#): a thread-safe variant of [Rc](#),
- atomic shared variables for numeric types,
- ...

Most of them use interior mutability.

# Mutex<T>: exclusive locks

Mutex<T> is a lock, explicitly protecting a variable of type T.

In many programming languages, mutexes are not explicitly associated with the resource they protect.

- In Rust, a Mutex<T> owns a value of type T.
- Another example of interior mutability, because after taking the lock we can mutate the protected value.

```
pub struct Mutex<T> { ... }
pub struct MutexGuard<'a, T> { ... }

impl<T> Mutex<T> {
    pub fn new(t: T) -> Mutex<T> { ... }

    // Taking a lock, receiving a MutexGuard
    pub fn lock<'a>(&'a self) -> MutexGuard<'a, T> { ... }

    // These functions allow accessing the content of the mutex without locking.
    // This is safe because here we know we are the unique owner.
    pub fn into_inner(self) -> T { ... }
    pub fn get_mut(&mut self) -> &mut T { ... }
}

unsafe impl<T: Send> Send for Mutex<T> {}
unsafe impl<T: Send> Sync for Mutex<T> {}
```

Small lie: a mutex can be “poisonned” if a thread panicked while holding the lock... This technicality adds noise in the API, we erase it here.

```
// Guards can be used as a shared or mutable borrow
impl<'a, T> Deref for MutexGuard<'a, T> {
    type Target = T;
    fn deref(&self) -> &T { ... }
}
impl<'a, T> DerefMut for MutexGuard<'a, T> {
    fn deref_mut(&mut self) -> &mut T { ... }
}

// Dropping the guards unlocks the mutex
impl<'a, T> Drop for MutexGuard<'a, T> {
    fn drop(&mut self) { ... }
}

unsafe impl<'a, T: Sync> Sync for MutexGuard<'a, T> {}
// MutexGuard is not Send: cannot unlock in a different thread
```

# Conclusion

Multithreading in Rust is mostly a **library concern**.

Nothing in the language is designed explicitly for concurrency.

- Except perhaps the fact that `Send` and `Sync` are auto-trait. But this is a reusable mechanism.

Yet, Rust is particularly well-suited for multithreading!

# Next week

Next week, we will get an overview of how one can prove the **soundness of the type system** of Rust.

We will use a **logical relation** approach.

Get ready:

- revise logical relations,
- think about how to formalize the type system,
- think about how you would design the logical relation.