Lecture 7: Unsafe. Parallel computing

Alexander Stanovoy

May 10, 2022

alex.stanovoy@gmail.com

In this lecture

- Unsafe and it’s contracts
- Pointers
- Uninitialized memory
- Unsafe: when and how
- Parallel computing
- Crossbeam
- Rayon

Unsafe and it’s contracts

Unsafe

Nearly all course we were talking about Safe Rust - a Rust where we cannot make any memory safety bugs or cause undefined behaviour.

Unsafe Rust is a superset of language where we can control our code more precisily while risking the Rust's safety: incorrect usage of it results in memory unsafety and undefined behaviour just like in unsafe languages!

Unsafe

It appears in Safe Rust we cannot write all of the code we may need in practice!

Unsafe

It appears in Safe Rust we cannot write all of the code we may need in practice!

- `Vec`, since we must have a buffer with a quite dangerous invariant: prefix of initialized elements and suffix with uninitialized values!

Unsafe

It appears in Safe Rust we cannot write all of the code we may need in practice!

- `Vec`, since we must have a buffer with a quite dangerous invariant: prefix of initialized elements and suffix with uninitialized values!
- Perform some optimizations such as implementing linked lists without runtime overhead or save a little of bytes on allocations.

Unsafe

It appears in Safe Rust we cannot write all of the code we may need in practice!

- `Vec`, since we must have a buffer with a quite dangerous invariant: prefix of initialized elements and suffix with uninitialized values!
- Perform some optimizations such as implementing linked lists without runtime overhead or save a little of bytes on allocations.
- Implement functions with potentially malicious behaviour like `split_at_mut`.

Unsafe

It appears in Safe Rust we cannot write all of the code we may need in practice!

- `Vec`, since we must have a buffer with a quite dangerous invariant: prefix of initialized elements and suffix with uninitialized values!
- Perform some optimizations such as implementing linked lists without runtime overhead or save a little of bytes on allocations.
- Implement functions with potentially malicious behaviour like `split_at_mut`.
- Interfacing directly with hardware, operating systems, or other languages.

Unsafe

What exactly Unsafe Rust can do?

- Dereference raw pointers.
- Call unsafe functions (C functions, compiler intrinsics, and the raw allocator).
- Implement `unsafe` traits.
- Mutate statics.
- Access fields of unions.

And that's all! Note that it's not disabling, for instance, borrow checker: it just grants us a power which, as we'll see, comes with great responsibility.

`unsafe` keyword

That's how `get_unchecked` function is declared in `Vec`:

```
pub unsafe fn get_unchecked<I>(
    &self,
    index: I
) -> &I::Output
where
    I: SliceIndex<Self>,
{
    // Note that we're dereferencing a pointer
    // so we need 'unsafe' here!
    unsafe { &*index.get_unchecked(self) }
}
```

Every `unsafe` function upholds some contract that is usually rigorously documented! In this case, we're acquiring an element that's possibly not in the correct range.

`unsafe` keyword

That's how to use `unsafe` function in your code:

```
let v = vec![1];
unsafe {
    let x = v.get_unchecked(0);
}
```

`unsafe` block allows us to use an `unsafe` functions and dereference pointers. That's all! Technically, there are a few other things you can do, but those don't change the point.

Note that you can use such blocks everywhere, but be sure to uphold the contract!

`unsafe` keyword

For historical reasons, every `unsafe fn` contains an implicit unsafe block in Rust today. That is, if you declare an `unsafe fn`, you can always invoke any unsafe methods or primitive operations inside that `fn`.

`unsafe` keyword

For historical reasons, every `unsafe fn` contains an implicit unsafe block in Rust today. That is, if you declare an `unsafe fn`, you can always invoke any unsafe methods or primitive operations inside that `fn`.

However, that decision is now considered a mistake, and it's currently being reverted through the already accepted and implemented RFC 2585. This RFC warns about having an `unsafe fn` that performs unsafe operations without an explicit unsafe block inside it.

`unsafe` keyword

For historical reasons, every `unsafe fn` contains an implicit unsafe block in Rust today. That is, if you declare an `unsafe fn`, you can always invoke any unsafe methods or primitive operations inside that `fn`.

However, that decision is now considered a mistake, and it's currently being reverted through the already accepted and implemented RFC 2585. This RFC warns about having an `unsafe fn` that performs unsafe operations without an explicit unsafe block inside it.

The lint will also likely become a hard error in future editions of Rust. The idea is to reduce the "footgun radius" - if every `unsafe fn` is one giant unsafe block, then you might accidentally perform unsafe operations without realizing it!

`unsafe` keyword

This way we can declare `unsafe` trait in code:

```
unsafe trait TrustedOrd: Ord {}
```

Here, `unsafe` means, again, that **implementation of this trait** must uphold some contract. It's unsafe to implement it, not to use!

`unsafe` keyword

This way we can declare `unsafe` trait in code:

```
unsafe trait TrustedOrd: Ord {}
```

Here, `unsafe` means, again, that **implementation of this trait** must uphold some contract. It's unsafe to implement it, not to use!

Therefore, `unsafe` is needed to declare a contract that cannot be verified by the compiler and the programmer needs to uphold it by hand.

Soundness and Unsoundness

We already know what these words mean:

- An abstraction is called *sound* when any usage of it or the surrounding code with any arguments cannot cause memory unsafety and undefined behaviour.
- An *unsound* abstraction is an opposite to *sound* abstraction.

Soundness and Unsoundness

We already know what these words mean:

- An abstraction is called *sound* when any usage of it or the surrounding code with any arguments cannot cause memory unsafety and undefined behaviour.
- An *unsound* abstraction is an opposite to *sound* abstraction.

Any `unsafe` code is unsound by default. To make it sound, we write a safe abstraction around it, checking whether the requirements of `unsafe` are satisfied.

Soundness and Unsoundness

An example is `split_at_mut` function of slice, which creates two `&mut [T]` slices from one, that obviously violates the AXM rule, but at the same time is actually safe!

```
// Note the comment from the standard library!
pub fn split_at_mut(
    &mut self,
    mid: usize
) -> (&mut [T], &mut [T]) {
    assert!(mid <= self.len());
    // SAFETY: '[ptr; mid]' and '[mid; len]'
    // are inside 'self', which fulfills the
    // requirements of 'split_at_mut_unchecked'.
    unsafe { self.split_at_mut_unchecked(mid) }
}
```

Soundness and Unsoundness

```rust
pub unsafe fn split_at_mut_unchecked(
    &mut self,
    mid: usize
) -> (&mut [T], &mut [T]) {
    let len = self.len();
    let ptr = self.as_mut_ptr();
    // SAFETY: Caller has to check that '0 <= mid <= self.len()'.
    // '[ptr; mid]' and '[mid; len]' are not overlapping,
    // so returning a mutable reference is fine.
    unsafe {
        (
            from_raw_parts_mut(ptr, mid),
            from_raw_parts_mut(ptr.add(mid), len - mid)
        )
    }
}
```

Pointers

Pointers

As we already seen, Rust actually have pointers! To interact with them, we usually use theirs methods, `std::ptr` and `std::mem` modules.

```
use std::ptr;

let x: *mut i32 = ptr::null_mut();
if x.is_null() {
    let y: *const i32 = ptr::null();
}
let z: *mut &str = Box::into_raw(Box::new("abc"));
unsafe {
    println!("{}", &(*z)[0..2]);  // ab
}
```

It's safe to do whatewer we want with the pointers in Safe Rust until we need to dereference it.

Pointers

Question: What means `&*` ?

```
let s: *mut &str = Box::into_raw(Box::new("abc"));
let r = unsafe { &*z };
```

Pointers

Question: What means `&*` ?

```
let s: *mut &str = Box::into_raw(Box::new("abc"));
let r: &&str = unsafe { &*z };
```

It means “Dereference a pointer, then create a reference to the inner `T`”. Note that violating AXM by, for instance, *creating* two mutable references results in *instant* undefined behaviour!

Pointers and references

The core difference between pointers and references is that:

- Pointers aren't pointing to the valid data all of the time.
- References have a lifetime dependency to track whether they're outliving their parent.

Pointers are usually proved useful when you cannot check the lifetime of the object statically.

Pointer arithmetic

With pointers, you can do arbitrary pointer arithmetic, just like you can in C, by using `.add()`, and `.sub()` to move the pointer to any byte that lives *within the same allocation*:

```
let arr: [u8; 4] = [0, 1, 2, 3];
let ptr: *const u8 = arr.as_ptr();

unsafe {
    println!("{}", *ptr.add(1));  // 1
    println!("{}", *ptr.add(2));  // 2
}
```

Pointer arithmetic

Question: What means “within the same allocation”? Why do we care?

Question: What means "within the same allocation"? Why do we care?

Since such pointers are invalid, **any** usage of it is undefined behaviour! For instance, compiler is allowed to decide to eat your code and replace it with arbitrary nonsense. This is why Rust marks `add` as `unsafe`.

Question: What means “within the same allocation”? Why do we care?

Since such pointers are invalid, **any** usage of it is undefined behaviour! For instance, compiler is allowed to decide to eat your code and replace it with arbitrary nonsense. This is why Rust marks `add` as `unsafe`.

Wow! Even writing such a simple code we can shoot a leg! Safe Rust guarranties that you won’t do such a thing, but Unsafe Rust cannot.

Question: What means “within the same allocation”? Why do we care?

Since such pointers are invalid, **any** usage of it is undefined behaviour! For instance, compiler is allowed to decide to eat your code and replace it with arbitrary nonsense. This is why Rust marks `add` as `unsafe`.

Wow! Even writing such a simple code we can shoot a leg! Safe Rust guarranties that you won’t do such a thing, but Unsafe Rust cannot.

To undestand better why we don’t want to write unsafe code much even in other languages, let’s check an example where the compiler uses 3 optimizations and breaks a code complitely.

Pointer arithmetic

Here's the code our compiler wants to optimize:[1]

```
// No optimizations
char p[1], q[1] = {0};
uintptr_t ip = (uintptr_t)(p+1);
uintptr_t iq = (uintptr_t)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```

This program has two possible behaviors: either `ip` (the address one-past-the-end of `p`) and `iq` (the address of `q`) are different, and nothing is printed.

[1] Pointers Are Complicated II, or: We need better language specs

Pointer arithmetic

The first "optimization" we will perform is to exploit that if we enter the `if` body, we have `iq == ip`, so we can replace all `iq` by `ip`.

```
// 1 optimization
char p[1], q[1] = {0};
uintptr_t ip = (uintptr_t)(p+1);
uintptr_t iq = (uintptr_t)q;
if (iq == ip) {
    *(char*)(uintptr_t)(p+1) = 10;  // <- This line changed
    print(q[0]);
}
```

Pointer arithmetic

The second optimization notices that we are taking a pointer `p+1`, casting it to an integer, and casting it back, so we can remove the cast roundtrip:

```
// 2 optimizations
char p[1], q[1] = {0};
uintptr_t ip = (uintptr_t)(p+1);
uintptr_t iq = (uintptr_t)q;
if (iq == ip) {
    *(p+1) = 10;  // <- This line changed
    print(q[0]);
}
```

Pointer arithmetic

The final optimization notices that `q` is never written to, so we can replace `q[0]` by its initial value 0:

```
// 3 optimizations
char p[1], q[1] = {0};
uintptr_t ip = (uintptr_t)(p+1);
uintptr_t iq = (uintptr_t)q;
if (iq == ip) {
    *(p+1) = 10;
    print(0);  // <- This line changed
}
```

But wait, this code **never** produces the same input as original!

Pointer arithmetic

Question: One of these optimizations is incorrect. But which one is it?

Pointer arithmetic

Question: One of these optimizations is incorrect. But which one is it?

You may think of the last optimization...

Question: One of these optimizations is incorrect. But which one is it?

You may think of the last optimization...

When compiler decided to perform it, the main observation was like: “Since `q` and `p` point to different local variables, a pointer derived from `p` cannot alias `q[0]`, and hence we know that this write cannot affect the value stored at `q[0]`”.

But it appears it's a programmers fault!

```c
// No optimizations
char p[1], q[1] = {0};
uintptr_t ip = (uintptr_t)(p+1);
uintptr_t iq = (uintptr_t)q;
if (iq == ip) {
    *(char*)iq = 10;
    print(q[0]);
}
```

`p+1` is a one-past-the-end pointer, so it actually can have the same address as `q[0]`. However, LLVM IR (just like C) **does not permit memory accesses through one-past-the-end pointers**.

Pointer arithmetic

So, writing unsafe code is hard, and it's not only about writing unsafe Rust.

Are you already frightened? Me too.

mem::transmute

Sometimes, you have a type T and want to treat it as some other type U. In C, you can just cast a type. In Rust, we only know how to convert one object into another.

There's actually a way to reinterpret the bits of a value of one type as another type - `std::mem::transmute`.

```
fn foo() -> i32 {
    42
}
let pointer = foo as *const ();
let function = unsafe {
    std::mem::transmute::<*const (), fn() -> i32>(pointer)
};
assert_eq!(function(), 42);
```

`mem::transmute`

The only verification `transmute` applies is that the `T` and `U` must have the same size. There's a whole spectrum of things you should consider while transmuting memory:

- Creating an instance of any type with an invalid state is going to cause arbitrary chaos that can't really be predicted. Do not transmute `3u8` to `bool`. **Even if you never do anything with the `bool`!**

`mem::transmute`

The only verification `transmute` applies is that the `T` and `U` must have the same size. There's a whole spectrum of things you should consider while transmuting memory:

- Creating an instance of any type with an invalid state is going to cause arbitrary chaos that can't really be predicted. Do not transmute `3u8` to `bool`. **Even if you never do anything with the `bool`!**
- Transmuting an `&` to `&mut` is **always** undefined behaviour.

`mem::transmute`

The only verification `transmute` applies is that the `T` and `U` must have the same size. There's a whole spectrum of things you should consider while transmuting memory:

- Creating an instance of any type with an invalid state is going to cause arbitrary chaos that can't really be predicted. Do not transmute `3u8` to `bool`. **Even if you never do anything with the `bool`!**
- Transmuting an `&` to `&mut` is **always** undefined behaviour.

To be more specific, creating an invalid value of type `T` always results in undefined behaviour, because Rust relies on that.

`mem::transmute_copy`

`mem::transmute_copy` is even more unsafe: it doesn't check whether types have the same size! It copies `size_of<U>` bytes out of an `&T` and interprets them as a `U`.

It is undefined behavior for `U` to be larger than `T`.

Uninitialized memory

Uninitialized memory

Sometimes you need to store a value that isn't currently valid for its type.

Uninitialized memory

Sometimes you need to store a value that isn't currently valid for its type.

The most common example of this is if you want to allocate a chunk of memory for some type `T` and then read in the bytes from, for instance, the network.

Uninitialized memory

Sometimes you need to store a value that isn’t currently valid for its type.

The most common example of this is if you want to allocate a chunk of memory for some type `T` and then read in the bytes from, for instance, the network.

For this, we use the `std::mem::MaybeUninit<T>` structure. It stores exactly a `T`, but the compiler knows to make no assumptions about the validity of that `T`.

Uninitialized memory

The core methods of `MaybeUninit`:

- `uninit()` - creates a new `MaybeUninit`, or, simply speaking, a type of size of `T`. Not that you **cannot** rely on its contents, since it's uninitialized!
- `new(val: T)` - creates a new `MaybeUninit` initializing it with the contents of `T`. The compiler still make no assumptions on the contents of resulting `MaybeUninit`.
- `assume_init(self) -> T` - assumes current `MaybeUninit` to be initialized and returns it as `T`. Note that this function is `unsafe`.

Uninitialized memory

Let’s create an array of values, assuming that this array can be partly uninitialized during the process.

```rust
let array = unsafe {
    // Type inference gives
    // MaybeUninit::<[MaybeUninit<MyType>; 256]>::uninit()
    // There's also nightly feature with function 'uninit_array'
    let mut array: [MaybeUninit<MyType>; 256] =
        MaybeUninit::uninit().assume_init();

    for (i, elem) in array.iter_mut().enumerate() {
        *elem = MaybeUninit::new(calculate_elem(i));
    }

    std::mem::transmute::<_, [MyType; 256]>(array)
};
```

Uninitialized memory

When working with uninitialized memory and pointers, you should keep in mind that modifying the contents of the pointer by using dereference calls `drop`.

```
let mut b: MaybeUninit<Box<i32>> = MaybeUninit::uninit();
// 'as_mut_ptr' returns a '*mut T'
unsafe {
    // Totally wrong!
    *b.as_mut_ptr() = Box::new(42 as i32);
}
```

So, in fact, in the example above, if our type `MyType` implements `Drop`, we'll receive undefined behaviour for free!

The correct alternative, if for some reason we cannot use `MaybeUninit::new`, is to use the `std::ptr` module.

Uninitialized memory

The correct alternative, if for some reason we cannot use `MaybeUninit::new`, is to use the `std::ptr` module.

- `ptr::write(ptr, val)` takes a `val` and moves it into the address pointed to by `ptr`.

The correct alternative, if for some reason we cannot use `MaybeUninit::new`, is to use the `std::ptr` module.

- `ptr::write(ptr, val)` takes a `val` and moves it into the address pointed to by `ptr`.
- `ptr::copy(src, dest, count)` copies the bits that count `T`'s would occupy from `src` to `dest`. (this is equivalent to `memmove`)

Uninitialized memory

The correct alternative, if for some reason we cannot use `MaybeUninit::new`, is to use the `std::ptr` module.

- `ptr::write(ptr, val)` takes a `val` and moves it into the address pointed to by `ptr`.
- `ptr::copy(src, dest, count)` copies the bits that count `T`'s would occupy from `src` to `dest`. (this is equivalent to `memmove`)
- `ptr::copy_nonoverlapping(src, dest, count)` does what copy does, but a little faster on the assumption that the two ranges of memory don't overlap. (this is equivalent to `memcpy`)

Uninitialized memory

One more remark: it’s illegal to construct a reference to uninitialized data!

Instead, use pointers.

Uninitialized memory

One more remark: it’s illegal to construct a reference to uninitialized data! Instead, use pointers.

But what if we want to take a pointer to the field of the structure?

Uninitialized memory

It's illegal to construct a reference to uninitialized data! Instead, use pointers.

But what if we want to take a pointer to the field of the structure?

Ok, straightforward way:

```
&some_struct.field as *const _
```

Uninitialized memory

It's illegal to construct a reference to uninitialized data! Instead, use pointers.

But what if we want to take a pointer to the field of the structure?

Ok, straightforward way:

```
&some_struct.field as *const _
```

The problem here is not only that we're creating a reference here - imagine our struct to be `packed`. Our reference will be invalid since it is unaligned!

Uninitialized memory

To fix this, use `addr_of` and `addr_of_mut`.

```rust
struct Demo {
    field: bool,
}

let mut uninit = MaybeUninit::<Demo>::uninit();
let f1_ptr = unsafe {
    ptr::addr_of_mut!((*uninit.as_mut_ptr()).field)
};
unsafe { f1_ptr.write(true); }

let init = unsafe { uninit.assume_init() };
```

Uninitialized memory

As you may remember, Rust compiler makes optimizations depending on possible values of the type. For instance, `Option<Box<T>>` has the same size as just `Box<T>`. It's called *niche optimization*.

But when `T` is `MaybeUninit<U>`, then you cannot make any assumptions about the underlying value! That implies `Option<MaybeUninit<T>>` is not the same as `Option<T>`.

The same applies to other similar containers such as `NonZero`, `NonNull` and so on.

Uninitialized memory

```
let array = unsafe {
    let mut array: [MaybeUninit<MyType>; 256] =
        MaybeUninit::uninit().assume_init();

    for (i, elem) in array.iter_mut().enumerate() {
        *elem = MaybeUninit::new(calculate_elem(i));
    }

    std::mem::transmute::<_, [MyType; 256]>(array)
};
```

What if `MyType` needs to drop (eg. allocates a memory) and `calculate_elem` panics? We'll end up with a memory leak! You should keep it in mind while using uninitialized memory.

Unsafe: when and why

Unsafe: when and why

- Unsafe code can violate all of Rust's safety guarantees, and this is often touted as a reason why Rust's whole safety argument is a charade.

Unsafe: when and why

- Unsafe code can violate all of Rust's safety guarantees, and this is often touted as a reason why Rust's whole safety argument is a charade.
- But actually, many successiful safe languages have an unsafe superset, usually in the form of code written in C or assembly.

Unsafe: when and why

- Unsafe code can violate all of Rust's safety guarantees, and this is often touted as a reason why Rust's whole safety argument is a charade.
- But actually, many successiful safe languages have an unsafe superset, usually in the form of code written in C or assembly.
- So, you *do* need unsafe code when it comes to writing something low-level.

Unsafe: when and why

- If you’re now afraid of writing unsafe code, just remember you’ve already written in C++ and somehow managed to survive :)

Unsafe: when and why

- If you're now afraid of writing unsafe code, just remember you've already written in C++ and somehow managed to survive :)
- The most important part is to read (and write!) the documentation about unsafe functions and carefully check whether all of the invariants are satisfied all the time.

Unsafe: when and why

- If you're now afraid of writing unsafe code, just remember you've already written in C++ and somehow managed to survive :)
- The most important part is to read (and write!) the documentation about unsafe functions and carefully check whether all of the invariants are satisfied all the time.
- Try to write as less unsafe code as possible, minimize unsafe blocks size.

Unsafe: when and why

- If you’re now afraid of writing unsafe code, just remember you’ve already written in C++ and somehow managed to survive :)
- The most important part is to read (and write!) the documentation about unsafe functions and carefully check whether all of the invariants are satisfied all the time.
- Try to write as less unsafe code as possible, minimize unsafe blocks size.
- Reuse existing crates: they are always well-tested by community, and remember that soundness bugs are the most punishable bugs in the Rust community.

Unsafe: when and why

- If you're now afraid of writing unsafe code, just remember you've already written in C++ and somehow managed to survive :)
- The most important part is to read (and write!) the documentation about unsafe functions and carefully check whether all of the invariants are satisfied all the time.
- Try to write as less unsafe code as possible, minimize unsafe blocks size.
- Reuse existing crates: they are always well-tested by community, and remember that soundness bugs are the most punishable bugs in the Rust community.
- And test your work! In that, tools like Miri will help you much.

Parallel Computing

In this chapter, you're supposed to have finished Concurrency course.

It's time to make our programs multithreaded! The best way to start is to create new threads and compute something in parallel.

To do so, we use `std::thread::spawn`.

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

Example:

```
const THREAD_NUM: usize = 8;
let result: Vec<usize> = (0..THREAD_NUM)
    .map(|_| thread::spawn(move || simulate()))
    .map(|handle| handle.join().expect("thread panicked!"))
    .collect();
```

Example:

```
const THREAD_NUM: usize = 8;
let result: Vec<usize> = (0..THREAD_NUM)
    .map(|_| thread::spawn(move || simulate()))
    .map(|handle| handle.join().expect("thread panicked!"))
    .collect();
```

Question: Why do we need this `expect`?

Example:

```
const THREAD_NUM: usize = 8;
let result: Vec<usize> = (0..THREAD_NUM)
    .map(|_| thread::spawn(move || simulate()))
    .map(|handle| handle.join().expect("thread panicked!"))
    .collect();
```

Question: Why do we need this **`expect`**?

Thread can panic while executing. This panic should stop in the source thread. When joining, our **`JoinHandle`** will give us a **`Result`** with either a final value or an error with a panic value.

'static in thread::spawn

Consider the following code:

```rust
fn example() {
    let vec = vec![1, 2, 3];
    let handle = thread::spawn(|| {
        for i in vec.iter() {
            println!("{i}");
        }
    });
    handle.join();
}
```

'static in thread::spawn

```
error[E0373]: closure may outlive the current function, but it
              borrows `vec`, which is owned by the current function
 --> src/main.rs:5:31
  |
5 |     let handle = thread::spawn(|| {
  |                                ^^ may outlive borrowed value `vec`
6 |         for i in vec.iter() {
  |                  --- `vec` is borrowed here
  |
note: function requires argument type to outlive `'static`
 ...
help: to force the closure to take ownership of `vec` (and any
      other referenced variables), use the `move` keyword
  |
5 |     let handle = thread::spawn(move || {
  |                                ++++
```

`'static` in `thread::spawn`

- Rust knows nothing about a `join`.
- Moreover, even if it will know, it cannot guarrantie that we won't panic until `join`.
- And the most ridiculous: nothings stops us from *leaking* a `JoinHandle`!

So, we need to make closure `'static` to outlive any possible variable in the program. We'll use **`move`** here as the compiler suggests.

```
fn example() {
    let vec = vec![1, 2, 3];
    let guard = thread::spawn(move || {
        for i in vec.iter() {
            println!("{i}");
        }
    });
    guard.join();
}
```

The same answer applies to the question about `T` having `'static` lifetime.

'static in thread::spawn

One more program:

```
fn count_foo_bar(data: &str) -> usize {
    let t1 = thread::spawn(|| data.matches("foo").count());
    let t2 = thread::spawn(|| data.matches("bar").count());
    t1.join().unwrap() + t2.join().unwrap()
}
```

'static in thread::spawn

One more program:

```
fn count_foo_bar(data: &str) -> usize {
    let t1 = thread::spawn(|| data.matches("foo").count());
    let t2 = thread::spawn(|| data.matches("bar").count());
    t1.join().unwrap() + t2.join().unwrap()
}
```

```
error[E0621]: explicit lifetime required in the type of
               `data` --> src/lib.rs:4:14
  |
4 | let t1 = thread::spawn(|| data.matches("foo").count());
  |          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             lifetime `' static` required
```

'static in thread::spawn

```rust
fn count_foo_bar(data: Rc<str>) -> usize {
    let data_2 = data.clone();
    let t1 = thread::spawn(move || {
        data.matches("foo").count()
    });
    let t2 = thread::spawn(move || {
        data_2.matches("bar").count()
    });
    t1.join().unwrap() + t2.join().unwrap()
}
```

'static in thread::spawn

```
error[E0277]: `Rc<str>` cannot be sent between threads safely
   --> src/main.rs:7:18
    |
7   |           let t1 = thread::spawn(move || {
    |  __________________^^^^^^^^^^^^^_-
    | |                  |
    | |                  `Rc<str>` cannot be sent between
    | |                  threads safely
8   | |             data.matches("foo").count()
9   | |         });
    | |_________- within this `[closure@src/main.rs:7:32: 9:10]`
    |
    = help: within `[closure@src/main.rs:7:32: 9:10]`, the
            trait `Send` is not implemented for `Rc<str>`
```

`'static` in `thread::spawn`

Question: Can you guess what means `Send`?

Question: Can you guess what means `Send`?

If we could send `Rc` between threads, it will be possible to have 2 threads simultaneously modifying the underlying non-atomic counter!

`'static` in `thread::spawn`

Question: Can you guess what means `Send`?

If we could send `Rc` between threads, it will be possible to have 2 threads simultaneously modifying the underlying non-atomic counter!

Or, simply speaking, we'll run into a **data race**.

Send and Sync traits

Firstly, let's remember what is a data race.

Firstly, let's remember what is a data race.

The simple one which doesn't involve memory models is defined as follows:

- Two or more threads concurrently accessing a location of memory.
- One or more of them is a write.
- One or more of them is unsynchronized.

Send and Sync traits

When Rust was on it's early stages, people believed memory safety and data race safety were totally different things. *But actually, the first implies the second!*

Send and Sync traits

When Rust was on it's early stages, people believed memory safety and data race safety were totally different things. *But actually, the first implies the second!*

Data races are mostly prevented through Rust's ownership system: it's impossible to alias a mutable reference, so it's impossible to perform a data race.

Send and Sync traits

When Rust was on it's early stages, people believed memory safety and data race safety were totally different things. *But actually, the first implies the second!*

Data races are mostly prevented through Rust's ownership system: it's impossible to alias a mutable reference, so it's impossible to perform a data race.

Interior mutability makes this more complicated, which is largely why we have the `Send` and `Sync` traits.

Send and Sync traits

Send and Sync are unsafe marker traits with the following meaning:

- A type is Send if it is safe to send it to another thread.
- A type is Sync if it is safe to share between threads.

Send and Sync traits

`Send` and `Sync` are `unsafe` marker traits with the following meaning:

- A type is `Send` if it is safe to send it to another thread.
- A type is `Sync` if it is safe to share between threads. (`T` is `Sync` if and only if `&T` is `Send`)

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32`
- `Vec<i32>`
- `&str`
- `Rc<T>`
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>`
- `&str`
- `Rc<T>`
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str`
- `Rc<T>`
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>`
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>`
- `MutexGuard<'static, ()>`
- `*mut T`

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>` - only `Send`.
- `MutexGuard<'static, ()>`
- `*mut T`

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>` - only `Send`.
- `MutexGuard<'static, ()>` - only `Sync`.
- `*mut T`

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>` - only `Send`.
- `MutexGuard<'static, ()>` - only `Sync`.
- `*mut T` - not `Send` nor `Sync`.

Send and Sync traits

We need to complete a little quiz to understand what's happening. What's the types are `Sync` and `Send`?

- `i32` - `Send` and `Sync`.
- `Vec<i32>` - `Send` and `Sync`.
- `&str` - `Send` and `Sync`.
- `Rc<T>` - not `Send` nor `Sync`.
- `Cell<T>` - only `Send`.
- `MutexGuard<'static, ()>` - only `Sync`.
- `*mut T` - not `Send` nor `Sync`.

Most types are `Send` and `Sync`, but there's some exceptions.

Send and Sync traits

`Send`/`Sync` are also `auto` traits: they are implemented automatically for a type if all of its generics are `Send`/`Sync`.

Their final definition is:

```rust
pub unsafe auto trait Send {}
pub unsafe auto trait Sync {}
```

Send and Sync traits

In the incredibly rare case that a type is inappropriately automatically derived to be `Send` or `Sync`, then one can also unimplement `Send` and `Sync`.

```
#![feature(negative_impls)]

// I have some magic semantics for
// some synchronization primitive!
struct SpecialThreadToken(u8);

impl !Send for SpecialThreadToken {}
impl !Sync for SpecialThreadToken {}
```

Please note that this requires the nightly compiler! Possibly, `negative_impls` feature will land in the near future.

Send and Sync traits

In case you want to unimplement `Send` and `Sync` on the stable compiler, you can use `PhantomData`.

```
type DisableSend = PhantomData<MutexGuard<'static, ()>>;
type DisableSync = PhantomData<Cell<()>>;

struct Test {
    disable_send: DisableSend,
    disable_sync: DisableSync,
}
```

Send and Sync traits

To solve this problem, we need `Arc` - *atomic* reference counting pointer.

```
use std::sync::Arc;

fn count_foo_bar(data: Arc<str>) -> usize {
    let data_2 = data.clone();
    let t1 = thread::spawn(move || {
        data.matches("foo").count()
    });
    let t2 = thread::spawn(move || {
        data_2.matches("bar").count()
    });
    t1.join().unwrap() + t2.join().unwrap()
}
```

`std::sync`

You should know from the concurrency course that `Arc` isn't possible without atomics. Moreover, *nothing* is possible without atomics.

You should know from the concurrency course that `Arc` isn't possible without atomics. Moreover, *nothing* is possible without atomics.

To make atomics work, we need a memory model. We could just say "It's enough to have sequential consistency", but we're in Rust and want to make the fastest applications, right?

You should know from the concurrency course that `Arc` isn't possible without atomics. Moreover, *nothing* is possible without atomics.

To make atomics work, we need a memory model. We could just say "It's enough to have sequential consistency", but we're in Rust and want to make the fastest applications, right?

Rust just reuses C++20 memory model. It's not because this model is perfect (actually, everyone is pretty bad at modeling atomics), but because this model is well-studied and widely used. If there will appear a good memory model in academy, Rust will adopt it.

`std::sync` module contains the simpliest primitives of synchronizations. We'll start with the simpliest submodule - `sync::atomic::*`.

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

struct RequestHandler {
    counter: Arc<AtomicUsize>,
}
impl RequestHandler {
    fn handle_request(&self, req: ...) {
        self.counter.fetch_add(1, Ordering::SeqCst);
        /* ... */
    }
}
```

Question: Why do we need `Arc`?

Question: Can incorrect memory ordering lead to memory unsafety?

Question: Can incorrect memory ordering lead to memory unsafety?

No, until your code uses unsafe which relies on correct memory orders.

Question: Can incorrect memory ordering lead to memory unsafety?

No, until your code uses unsafe which relies on correct memory orders.

Question: Ok, data races are gone. But what about *general race conditions*?

Race conditions can happen in Rust: your program can still get deadlocked or do something nonsensical with incorrect synchronization. *Still, a race condition can't violate memory safety in a Rust program on its own!*

Question: Can incorrect memory ordering lead to memory unsafety?

No, until your code uses unsafe which relies on correct memory orders.

Question: Ok, data races are gone. But what about *general race conditions*?

Race conditions can happen in Rust: your program can still get deadlocked or do something nonsensical with incorrect synchronization. *Still, a race condition can't violate memory safety in a Rust program on its own!*

And it's the reason of existance of Rust's marketing term called "Fearless concurrency", meaning "You won't have data races, memory unsafety and undefined behaviour writing multithreaded code in Safe Rust".

std::sync

```
let data = vec![1, 2, 3, 4];
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();

thread::spawn(move || {
    other_idx.fetch_add(10, Ordering::SeqCst);
});

// Race condition!
// May panic but won't give us a memory unsafety
println!("{}", data[idx.load(Ordering::SeqCst)]);
```

```rust
let data = vec![1, 2, 3, 4];
let idx = Arc::new(AtomicUsize::new(0));
let other_idx = idx.clone();
thread::spawn(move || {
    other_idx.fetch_add(10, Ordering::SeqCst);
});

if idx.load(Ordering::SeqCst) < data.len() {
    // We can get memory unsafety here because of unsafe!
    unsafe {
        println!(
            "{}",
            data.get_unchecked(idx.load(Ordering::SeqCst))
        );
    }
}
```

You can notice that atomic is, actually, an interior mutability primitive which works in integers, exactly a multithreaded `Cell`.

Question: What is a multithreaded `RefCell`?

`std::sync`

`Mutex` is a mutual exclusion primitive used for protecting some `T`.

As you already know from the Concurrency course, `Mutex` usually protects some object. In reviews, you were probabily punished for writing mutex with name `mutex_` protecting unclear state :)

In Rust, it's also an interior mutability primitive, that is exactly a multithreaded `RefCell`.

`std::sync`

Rust have a number of other synchronization primitives:

- `Barrier`.
- `Condvar`.
- `mpsc`.
- `RwLock`.
- `Once` - Used for thread-safe, one-time initialization of a global variable.

An example of `Once`:

```
static mut VAL: usize = 0;
static INIT: Once = Once::new();

fn get_cached_val() -> usize {
    unsafe {
        INIT.call_once(|| {
            // Safety: we only mutate VAL once
            // in a synchronized fashion
            VAL = expensive_computation();
        });
        VAL
    }
}
fn expensive_computation() -> usize { /* ... */ }
```

Poisoning

Imagine our thread to panic while holding a `MutexGuard`. It means some data was partly modified when panic occured! It's not the thing that is actually violates memory safety, but we can break invariants without even noticing!

Poisoning

Imagine our thread to panic while holding a `MutexGuard`. It means some data was partly modified when panic occured! It's not the thing that is actually violates memory safety, but we can break invariants without even noticing!

Unlike C++, when locking `Mutex`, you're given `LockResult<MutexGuard<'_, T>>` which gives you a lock or `PoisonError`, meaning the mutex is poisoned, i.e thread that was holding a lock panicked!

The same applies to the thread and mpsc-queue.

Crossbeam

Crossbeam

Crossbeam is a crate with a set of tools for concurrent programming, and mainly exists to complement `std::sync`. Possibly, the parts of this crate will be moved to `std`!

Also provides some constructs useful for implementing lock-free algorithms!

crossbeam::scope

`crossbeam::scope` is used to create a scoped thread.

```rust
pub fn scope<'env, F, R>(
    f: F
) -> Result<R, Box<dyn Any + 'static + Send, Global>>
where
    F: FnOnce(&Scope<'env>) -> R;
```

crossbeam::scope

`crossbeam::scope` is used to create a scoped thread.

```
pub fn scope<'env, F, R>(
    f: F
) -> Result<R, Box<dyn Any + 'static + Send, Global>>
where
    F: FnOnce(&Scope<'env>) -> R;
```

- Creates a scope for running threads.

`crossbeam::scope` is used to create a scoped thread.

```
pub fn scope<'env, F, R>(
    f: F
) -> Result<R, Box<dyn Any + 'static + Send, Global>>
where
    F: FnOnce(&Scope<'env>) -> R;
```

- Creates a scope for running threads.
- Joins all of the threads running inside it, for which `join` was not called manually.

`crossbeam::scope`

`crossbeam::scope` is used to create a scoped thread.

```rust
pub fn scope<'env, F, R>(
    f: F
) -> Result<R, Box<dyn Any + 'static + Send, Global>>
where
    F: FnOnce(&Scope<'env>) -> R;
```

- Creates a scope for running threads.
- Joins all of the threads running inside it, for which `join` was not called manually.
- Allows threads to capture local variables. On panic, returns an error.

Usage example:

```
let people = vec![
    "Alice".to_owned(), "Bob".to_owned(), "Carol".to_owned()
];
thread::scope(|s| {
    for person in &people {
        s.spawn(move |_| {
            println!(
                "Hello from {:?}, {}!",
                std::thread::current().id(),
                person
            );
        });
    }
}).unwrap();
```

Panic example:

```
thread::scope(|s| {
    s.spawn(move |_| {
        println!("one");
        panic!("panic one");
    });
    s.spawn(move |_| {
        println!("two");
        panic!("panic two");
    });
}).map_err(|e| println!("{:?}", e));
```

```
thread '<unnamed>' panicked at 'panic two', src/main.rs:9:13
thread '<unnamed>' panicked at 'panic one', src/main.rs:5:13
```

`crossbeam::channel`

`crossbeam::channel` is an alternative to `std::sync::mpsc`.

- Message passing channels (Multi-Producer Multi-Consumer, MPMC).
- The channel may be limited by the size of the message buffer. Or unlimited.

crossbeam::channel

Usage example:

```
let (send_end, receive_end) = channel::bounded(5);
for i in 0..5 {
    send_end.send(i).unwrap();
}
// Will block!
send_end.send(5).unwrap();

let (send_end, receive_end) = channel::unbounded();
for i in 0..1000 {
    send_end.send(i).unwrap();
}
```

crossbeam::channel

Send and receive can be:

- Non-blocking.
- Blocking.
- With timeout.

```
pub fn try_send(&self, msg: T) -> Result<(), TrySendError<T>>;

pub fn send(&self, msg: T) -> Result<(), SendError<T>>;

pub fn send_timeout(
    &self,
    msg:T
    timeout: Duration
) -> Result<(), SendTimeoutError<T>>;
```

`crossbeam::channel`

- Send and receive handlers can be cloned and passed between threads. The channel remains the same - MPMC.

`crossbeam::channel`

- Send and receive handlers can be cloned and passed between threads. The channel remains the same - MPMC.
- If all handlers of one of the ends are dropped, the channel goes into the *disconnected* state.

`crossbeam::channel`

- Send and receive handlers can be cloned and passed between threads. The channel remains the same - MPMC.
- If all handlers of one of the ends are dropped, the channel goes into the *disconnected* state.
- Messages cannot be sent when channel is closed. But you can read already sent messages.

`crossbeam::channel`

- Send and receive handlers can be cloned and passed between threads. The channel remains the same - MPMC.
- If all handlers of one of the ends are dropped, the channel goes into the *disconnected* state.
- Messages cannot be sent when channel is closed. But you can read already sent messages.
- Operations on the disconnected channel aren’t blocking.

`crossbeam::channel`

- Send and receive handlers can be cloned and passed between threads. The channel remains the same - MPMC.
- If all handlers of one of the ends are dropped, the channel goes into the *disconnected* state.
- Messages cannot be sent when channel is closed. But you can read already sent messages.
- Operations on the disconnected channel aren’t blocking.
- You can use iterators on the channel: `iter()` for blocking, `try_iter()` for non-blocking.

crossbeam::select!

```rust
let (s1, r1) = channel::unbounded();
let (s2, r2) = channel::unbounded();
thread::spawn(move || assert_eq!(s1.send(10), Ok(())));
thread::spawn(move || assert_eq!(r2.recv(), Ok(20)));
select! {
    recv(r1) -> msg => assert_eq!(msg, Ok(10)),
    send(s2, 20) -> res => assert_eq!(res, Ok(())),
    default(Duration::from_secs(1)) => println!("timed out"),
}
```

crossbeam::select!

```
let (s1, r1) = channel::unbounded();
let (s2, r2) = channel::unbounded();
let mut sel = Select::new();
let oper1 = sel.recv(&r1);
let oper2 = sel.send(&s2);
let oper = sel.select_timeout(Duration::from_secs(1));
match oper {
    Ok(oper) => match oper.index() {
        i if i == oper1 => assert_eq!(oper.recv(&r1), Ok(10)),
        i if i == oper2 => assert_eq!(oper.send(&s2, 20), Ok(())),
        _ => panic!("forgotten operation"),
    },
    Err(_) => println!("timed out"),
}
```

`crossbeam::utils`

- `CachePadded` - pads the type along the cache line to disable *false sharing*.

`crossbeam::utils`

- `CachePadded` - pads the type along the cache line to disable *false sharing*.
- `ShardedLock` - sharded `RwLock`. Read captures the read lock in one shard, write captures all shards. Faster in read and slower in write than `RwLock`.

`crossbeam::utils`

- `CachePadded` - pads the type along the cache line to disable *false sharing*.
- `ShardedLock` - sharded `RwLock`. Read captures the read lock in one shard, write captures all shards. Faster in read and slower in write than `RwLock`.
- `Backoff` - exponential backoff implementation.

`crossbeam::deque`

`crossbeam::deque` - a concurrent deck that supports work stealing. Used to implement task schedulers.

`crossbeam::deque`

`crossbeam::deque` - a concurrent deck that supports work stealing. Used to implement task schedulers.

- Contains a global queue and each thread has a local queue.

`crossbeam::deque`

`crossbeam::deque` - a concurrent deck that supports work stealing. Used to implement task schedulers.

- Contains a global queue and each thread has a local queue.
- There's a `Steal` for stealing tasks from another thread's local queue.

`crossbeam::deque`

`crossbeam::deque` - a concurrent deck that supports work stealing. Used to implement task schedulers.

- Contains a global queue and each thread has a local queue.
- There's a `Steal` for stealing tasks from another thread's local queue.
- The thread first tries to take the task from its own queue, then from the global one, then it tries to steal the task from another thread.

`crossbeam::epoch`

`crossbeam::epoch` - garbage collector for lock-free algorithms.

`crossbeam::epoch` - garbage collector for lock-free algorithms.

- Solves the problem of deleting elements in lock-free algorithms.

`crossbeam::epoch`

`crossbeam::epoch` - garbage collector for lock-free algorithms.

- Solves the problem of deleting elements in lock-free algorithms.
- When deleted, the items are added to the basket corresponding to the epoch.

`crossbeam::epoch`

`crossbeam::epoch` - garbage collector for lock-free algorithms.

- Solves the problem of deleting elements in lock-free algorithms.
- When deleted, the items are added to the basket corresponding to the epoch.
- When interacting with a lock-free data structure, we increment the epoch.

`crossbeam::epoch` - garbage collector for lock-free algorithms.

- Solves the problem of deleting elements in lock-free algorithms.
- When deleted, the items are added to the basket corresponding to the epoch.
- When interacting with a lock-free data structure, we increment the epoch.
- And clean the garbage from the baskets from the last two epochs.

Rayon

Rayon is a crate for data-parallelism. It's very easy to use and lightweight! It just creates a **work-stealing** thread-pool and sends a number of tasks to it. The crate is optimized for CPU-bound tasks.

The usage is as simple as writing default Rust code!

```
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    input
        .par_iter()
        .map(|&i| i * i)
        .sum()
}
```

In short: There's no standard for building C programs. It's a non-portable mess, and a time sink. Cross-compilation of OpenMP was the last straw. Rust/Cargo is much more dependable, and enables me to support more features on more platforms.[2]

[2] Improved portability and performance (libimagequant)

Rayon

In short: There's no standard for building C programs. It's a non-portable mess, and a time sink. Cross-compilation of OpenMP was the last straw. Rust/Cargo is much more dependable, and enables me to support more features on more platforms.[2]

Additionaly, Rayon *never* had issues with memory safety. You can be sure your program is memory safe at compile time!

[2] Improved portability and performance (libimagequant)

`rayon::join`

`rayon::join` is used to run two closures.

```
pub fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
where
    A: FnOnce() -> RA + Send, B: FnOnce() -> RB + Send,
    RA: Send, RB: Send;
```

rayon::join

`rayon::join` is used to run two closures.

```
pub fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
where
    A: FnOnce() -> RA + Send, B: FnOnce() -> RB + Send,
    RA: Send, RB: Send;
```

- If used inside the thread pool, current thread executes one closure and the second is moved to the thread pool queue.

`rayon::join`

`rayon::join` is used to run two closures.

```rust
pub fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
where
    A: FnOnce() -> RA + Send, B: FnOnce() -> RB + Send,
    RA: Send, RB: Send;
```

- If used inside the thread pool, current thread executes one closure and the second is moved to the thread pool queue.
- If used outside of the thread pool, current thread will run closures sequentially.

`rayon::join`

`rayon::join` is used to run two closures.

```rust
pub fn join<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
where
    A: FnOnce() -> RA + Send, B: FnOnce() -> RB + Send,
    RA: Send, RB: Send;
```

- If used inside the thread pool, current thread executes one closure and the second is moved to the thread pool queue.
- If used outside of the thread pool, current thread will run closures sequentially.
- It is assumed that the closures are CPU-bound. If a closure blocks, it will block the thread in the thread pool and prevent us from utilizing all of the cores!

rayon::join

Usage example:

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

`rayon::scope`

`rayon::scope` is used to create a scoped thread.

```
pub fn scope<'scope, OP, R>(op: OP) -> R
where
    R: Send,
    OP: FnOnce(&Scope<'scope>) -> R + Send;
```

`rayon::scope`

`rayon::scope` is used to create a scoped thread.

```rust
pub fn scope<'scope, OP, R>(op: OP) -> R
where
    R: Send,
    OP: FnOnce(&Scope<'scope>) -> R + Send;
```

- Similar to `crossbeam::scope`, but `crossbeam::scope` starts threads in the created scope, while `rayon::scope` creates tasks that will be run in thread poll in the created scope.

`rayon::scope`

`rayon::scope` is used to create a scoped thread.

```rust
pub fn scope<'scope, OP, R>(op: OP) -> R
where
    R: Send,
    OP: FnOnce(&Scope<'scope>) -> R + Send;
```

- Similar to `crossbeam::scope`, but `crossbeam::scope` starts threads in the created scope, while `rayon::scope` creates tasks that will be run in thread poll in the created scope.
- `rayon::scope` waits for all tasks to complete.

`rayon::scope`

`rayon::scope` is used to create a scoped thread.

```
pub fn scope<'scope, OP, R>(op: OP) -> R
where
    R: Send,
    OP: FnOnce(&Scope<'scope>) -> R + Send;
```

- Similar to `crossbeam::scope`, but `crossbeam::scope` starts threads in the created scope, while `rayon::scope` creates tasks that will be run in thread poll in the created scope.
- `rayon::scope` waits for all tasks to complete.
- More flexible than `rayon::join`. But because of this, it uses the heap, where `rayon::join` can use the stack and avoid allocation.