Concurrency and Parallelism I

Threads and Synchronisation

Cooper Pierce & Jack Duvall

Carnegie Mellon University

Outline

Review: What are threads?

Threads allow our program to have multiple instruction streams executing concurrently (and perhaps, in parallel). Talking about 1:1 threads here:

What are some resources threads share?

... and some they don't?

Review: What are threads?

Threads allow our program to have multiple instruction streams executing concurrently (and perhaps, in parallel). Talking about 1:1 threads here:

What are some resources threads share?

- an address space

… and some they don't?

Review: What are threads?

Threads allow our program to have multiple instruction streams executing concurrently (and perhaps, in parallel). Talking about 1:1 threads here:

What are some resources threads share?

- an address space
- `static` variables

… and some they don't?

Review: What are threads?

Threads allow our program to have multiple instruction streams executing concurrently (and perhaps, in parallel). Talking about 1:1 threads here:

What are some resources threads share?

- an address space
- `static` variables
- signals, to an extent

... and some they don't?

Review: What are threads?

Threads allow our program to have multiple instruction streams executing concurrently (and perhaps, in parallel). Talking about 1:1 threads here:

What are some resources threads share?

- an address space
- `static` variables
- signals, to an extent

... and some they don't?

- registers

Review: What are threads?

Threads allow our program to have multiple instruction streams executing concurrently (and perhaps, in parallel). Talking about 1:1 threads here:

What are some resources threads share?

- an address space
- `static` variables
- signals, to an extent

... and some they don't?

- registers
- stack space (can still access other threads' stacks!)

Review: What are threads?

Threads allow our program to have multiple instruction streams executing concurrently (and perhaps, in parallel). Talking about 1:1 threads here:

What are some resource threads share?

- an address space
- `static` variables
- signals, to an extent

... and some they don't?

- registers
- stack space (can still access other threads' stacks!)
- `_Thread_local` variables

Review: What are threads?

Threads allow our program to have multiple instruction streams executing concurrently (and perhaps, in parallel). Talking about 1:1 threads here:

What are some resource threads share?

- an address space
- `static` variables
- signals, to an extent

… and some they don't?

- registers
- stack space (can still access other threads' stacks!)
- `_Thread_local` variables

Other alternatives to 1:1 threads include M:N threads ("green threads") and processes.

Counting in C

```c
int incr(void *x) {
        ++(*(int *)x);
        return 0;
}

int main() {
        int x = 0;
        thrd_t threads[NUM_THREADS];

        for (size_t i = 0; i < NUM_THREADS; ++i) {
                if (thrd_create(&threads[i], incr, &x) != thrd_success) {
                        fprintf(stderr, "Issue creating thread\n");
                        return 1;
                }
        }

        for (size_t i = 0; i < NUM_THREADS; ++i) {
                thrd_join(threads[i], NULL);
        }

        printf("After incrementing x %zu times, it is now %d\n",
                        NUM_THREADS, x);
        return 0;
}
```

Any issues?

Yes! Lots! Our program doesn't even work:

```
[16:54] laptop:lectures | ./a.out
After incrementing x 1000 times, it is now 1000
[16:55] laptop:lectures | ./a.out
After incrementing x 1000 times, it is now 1000
[16:55] laptop:lectures | ./a.out
After incrementing x 1000 times, it is now 1000
[16:55] laptop:lectures | ./a.out
After incrementing x 1000 times, it is now 999
[16:55] laptop:lectures | ./a.out
After incrementing x 1000 times, it is now 999
[16:55] laptop:lectures | ./a.out
After incrementing x 1000 times, it is now 999
```

and it isn't even consistently wrong. What happened?

Race Conditions

Our program is violating one of the things Rust's system of only[1] allowing mutation through exclusive borrows (`&mut T`) is designed to prevent: two different threads might try and modify the same value at the same time. There's at least one potential issue with this:

[1]mostly

Race Conditions

Our program is violating one of the things Rust's system of only[1] allowing mutation through exclusive borrows (`&mut T`) is designed to prevent: two different threads might try and modify the same value at the same time. There's at least one potential issue with this:

Thread A

```
tmp = *x

tmp = tmp + 1

*x = tmp
```

Thread B

```

tmp = *x
tmp = tmp + 1
*x = tmp
```

[1] mostly

Synchronisation

We need a way to communicate between threads, so they can coordinate working through a *critical section*: part of the code we want to ensure only one thread is executing at a time.

Luckily for us, there are generally a lot of ways exposed as part of thread APIs which let us do this:

Synchronisation

We need a way to communicate between threads, so they can coordinate working through a *critical section*: part of the code we want to ensure only one thread is executing at a time.

Luckily for us, there are generally a lot of ways exposed as part of thread APIs which let us do this:

- semaphores

Synchronisation

We need a way to communicate between threads, so they can coordinate working through a *critical section*: part of the code we want to ensure only one thread is executing at a time.

Luckily for us, there are generally a lot of ways exposed as part of thread APIs which let us do this:

- semaphores
- mutexes

Synchronisation

We need a way to communicate between threads, so they can coordinate working through a *critical section*: part of the code we want to ensure only one thread is executing at a time.

Luckily for us, there are generally a lot of ways exposed as part of thread APIs which let us do this:

- semaphores
- mutexes
- reader-writer locks

Synchronisation

We need a way to communicate between threads, so they can coordinate working through a *critical section*: part of the code we want to ensure only one thread is executing at a time.

Luckily for us, there are generally a lot of ways exposed as part of thread APIs which let us do this:

- semaphores
- mutexes
- reader-writer locks
- condvars

Synchronisation

We need a way to communicate between threads, so they can coordinate working through a *critical section*: part of the code we want to ensure only one thread is executing at a time.

Luckily for us, there are generally a lot of ways exposed as part of thread APIs which let us do this:

- semaphores
- mutexes
- reader-writer locks
- condvars
- spinlocks (almost guarateed to be the wrong choice unless you're a kernel)

Synchronisation

We need a way to communicate between threads, so they can coordinate working through a *critical section*: part of the code we want to ensure only one thread is executing at a time.

Luckily for us, there are generally a lot of ways exposed as part of thread APIs which let us do this:

- semaphores
- mutexes
- reader-writer locks
- condvars
- spinlocks (almost guarateed to be the wrong choice unless you're a kernel)
- various asm instructions used to implement any of the above

Counting in C: Electric Boogaloo

```c
int main() {
        int x = 0;
        mtx_init(&mutex, mtx_plain);
        thrd_t threads[NUM_THREADS];

        for (size_t i = 0; i < NUM_THREADS; ++i) {
                if (thrd_create(&threads[i], incr, &x) != thrd_success) {
                        fprintf(stderr, "Issue creating thread\n");
                        return 1;
                }
        }

        for (size_t i = 0; i < NUM_THREADS; ++i) {
                thrd_join(threads[i], NULL);
        }
        mtx_destroy(&mutex);

        printf("After incrementing x %zu times, it is now %d\n", NUM_THREADS,
               x);
        return 0;
}
```

```c
static mtx_t mutex;

int incr(void *x) {
        mtx_lock(&mutex);
        ++(*(int *)x);
        return 0;
}
```

Counting in C: This Time For Real

```c
int main() {
        int x = 0;
        mtx_init(&mutex, mtx_plain);
        thrd_t threads[NUM_THREADS];

        for (size_t i = 0; i < NUM_THREADS; ++i) {
                if (thrd_create(&threads[i], incr, &x) != thrd_success) {
                        fprintf(stderr, "Issue creating thread\n");
                        return 1;
                }
        }

        for (size_t i = 0; i < NUM_THREADS; ++i) {
                thrd_join(threads[i], NULL);
        }
        mtx_destroy(&mutex);

        printf("After incrementing x %zu times, it is now %d\n", NUM_THREADS,
               x);

        return 0;
}
```

```c
static mtx_t mutex;

int incr(void *x) {
        mtx_lock(&mutex);
        ++(*(int *)x);
        mtx_unlock(&mutex);
        return 0;
}
```

Other Issues?

Now that we've finalised on this version, are there any other issues we have to consider? How confident are we in the correctness of this?

Other Issues?

Now that we've finalised on this version, are there any other issues we have to consider? How confident are we in the correctness of this?

One tool we might use (had I written this with pthreads, instead of C11 threads) is ThreadSanitizer which is a pretty good *dynamic* checker. Note that this can't catch everything, and it'll only be as good as your test cases!

Other Issues?

Now that we've finalised on this version, are there any other issues we have to consider? How confident are we in the correctness of this?

One tool we might use (had I written this with pthreads, instead of C11 threads) is ThreadSanitizer which is a pretty good *dynamic* checker. Note that this can't catch everything, and it'll only be as good as your test cases!

For instance, we probably missed that if the thread running `main` terminates early, we're accessing a invalid stack value!

Outline

Toward a Better API

Let's begin by taking a look at the type of `thrd_create` (`pthread_create` is a little different, but for our purpose, close enough to just consider one of them):

```c
int thrd_create(thrd_t *thr, int (*func)(void *), void *arg)
```

How could we refactor this into something better, if we had more tools in our type system?

Toward a Better API

Let's begin by taking a look at the type of `thrd_create` (`pthread_create` is a little different, but for our purpose, close enough to just consider one of them):

```
int thrd_create(thrd_t *thr, int (*func)(void *), void *arg)
```

How could we refactor this into something better, if we had more tools in our type system?
Some possibilities:

```
result<thr, thrd_error> thrd_create(int (*func)(void *), void *arg)
```

Toward a Better API

Let's begin by taking a look at the type of `thrd_create` (`pthread_create` is a little different, but for our purpose, close enough to just consider one of them):

```
int thrd_create(thrd_t *thr, int (*func)(void *), void *arg)
```

How could we refactor this into something better, if we had more tools in our type system?
Some possibilities:

```
result<thr, thrd_error> thrd_create(int (*func)(void *), void *arg)
```

```
result<thr, thrd_error> thrd_create<T>(int (*func)(T), T arg)
```

Toward a Better API

Let's begin by taking a look at the type of `thrd_create` (`pthread_create` is a little different, but for our purpose, close enough to just consider one of them):

```
int thrd_create(thrd_t *thr, int (*func)(void *), void *arg)
```

How could we refactor this into something better, if we had more tools in our type system?
Some possibilities:

```
result<thr, thrd_error> thrd_create(int (*func)(void *), void *arg)
```

```
result<thr, thrd_error> thrd_create<T>(int (*func)(T), T arg)
```

```
result<handle<U>, thrd_error> thrd_create<T, U>(U (*func)(T), T arg)
```

Toward a Better API: the Function Itself

That last one seems like a pretty nice improvement, but is there anything else we might want to iterate on?

```
result<handle<U>, thrd_error> thrd_create<T, U>(U (*func)(T), T arg)
```

Toward a Better API: the Function Itself

That last one seems like a pretty nice improvement, but is there anything else we might want to iterate on?

```
result<handle<U>, thrd_error> thrd_create<T, U>(U (*func)(T), T arg)
```

We're still letting this just take any old function in! Supposing we had the ability to do so, what are some constraints we might want here?

Toward a Better API: the Function Itself

That last one seems like a pretty nice improvement, but is there anything else we might want to iterate on?

```
result<handle<U>, thrd_error> thrd_create<T, U>(U (*func)(T), T arg)
```

We're still letting this just take any old function in! Supposing we had the ability to do so, what are some constraints we might want here?

- probably don't want to let it take things by value if the type references a location which might be reachable through another pointer (e.g., a pointer, a reference counted pointer)

Toward a Better API: the Function Itself

That last one seems like a pretty nice improvement, but is there anything else we might want to iterate on?

```
result<handle<U>, thrd_error> thrd_create<T, U>(U (*func)(T), T arg)
```

We're still letting this just take any old function in! Supposing we had the ability to do so, what are some constraints we might want here?

- probably don't want to let it take things by value if the type references a location which might be reachable through another pointer (e.g., a pointer, a reference counted pointer)
- probably don't want to let it take a reference to a type if we could cause a write to occur using the reference (e.g., a reference to something fulfilling the first bullet point)

Toward a Better API: the Function Itself

That last one seems like a pretty nice improvement, but is there anything else we might want to iterate on?

```
result<handle<U>, thrd_error> thrd_create<T, U>(U (*func)(T), T arg)
```

We're still letting this just take any old function in! Supposing we had the ability to do so, what are some constraints we might want here?

- probably don't want to let it take things by value if the type references a location which might be reachable through another pointer (e.g., a pointer, a reference counted pointer)
- probably don't want to let it take a reference to a type if we could cause a write to occur using the reference (e.g., a reference to something fulfilling the first bullet point)
- any reference we pass has to last at least as long as the new thread will (potentially, for the life of the program)

Rust: `std::thread::spawn`

Here's what Rust gives us:

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

Rust: `std::thread::spawn`

Here's what Rust gives us:

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

So we've got some old favourites, and some new things to talk about:

Rust: `std::thread::spawn`

Here's what Rust gives us:

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

So we've got some old favourites, and some new things to talk about:

- Closure traits return, so we get a little bit more flexibility than function pointers

Rust: `std::thread::spawn`

Here's what Rust gives us:

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

So we've got some old favourites, and some new things to talk about:

- Closure traits return, so we get a little bit more flexibility than function pointers
- There's a new marker trait, `Send`—we'll talk about this in a second

Rust: `std::thread::spawn`

Here's what Rust gives us:

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

So we've got some old favourites, and some new things to talk about:

- Closure traits return, so we get a little bit more flexibility than function pointers
- There's a new marker trait, `Send`—we'll talk about this in a second
- We also have a *lifetime bound* on our types: this means that any references the bounded type contains needs to live for at least as long as the bounding lifetime—in the case of `'static`, for the life of the program.

Scoped threads: `std::thread::scope`

What about the case where we know we join a thread before some value it's borrowing is dropped? Does our current API support this well?

```rust
let mut x = 0;
let t = std::thread::spawn(|| x += 1);
t.join();
println!("{x}");
```

Scoped threads: `std::thread::scope`

What about the case where we know we join a thread before some value it's borrowing is dropped? Does our current API support this well?

```rust
let mut x = 0;
let t = std::thread::spawn(|| x += 1);
t.join();
println!("{x}");
```

No! This won't compile: playground.

Scoped threads: `std::thread::scope`

What about the case where we know we join a thread before some value it's borrowing is dropped? Does our current API support this well?

```rust
let mut x = 0;
let t = std::thread::spawn(|| x += 1);
t.join();
println!("{x}");
```

No! This won't compile: playground.
What would we need to know for this to be safe? How could we prove that threads were joined by a certain point?

Scoped threads: `std::thread::scope`

```
pub struct Scope<'scope, 'env: 'scope> { /* private fields */ }

pub fn scope<'env, F, T>(f: F) -> T
where
    F: for<'scope> FnOnce(&'scope Scope<'scope, 'env>) -> T,


impl<'scope, 'env> Scope<'scope, 'env> {
    pub fn spawn<F, T>(&'scope self, f: F)
        -> ScopedJoinHandle<'scope, T>
    where
        F: FnOnce() -> T + Send + 'scope,
        T: Send + 'scope,
}
```

Scoped threads: `std::thread::scope`

So we have a `Scope` which can use to spawn threads, and this scope has a lifetime `'scope`, which is the most threads spawned from it can live. In turn, the threads might borrow data for `'env`, the lifetime of the values in the captured environment, which is at least as long as `'scope`.

Scoped threads: `std::thread::scope`

So we have a `Scope` which can use to spawn threads, and this scope has a lifetime `'scope`, which is the most threads spawned from it can live. In turn, the threads might borrow data for `'env`, the lifetime of the values in the captured environment, which is at least as long as `'scope`.

So now, revisiting our example from before, we could write:

```
let mut x = 0;
std::thread::scope(|s| {
    s.spawn(|| x += 1);
});
println!("{x}");
```

`Sync`

`Sync` is a trait implemented by types if it is safe to share a borrow (i.e., `&T`) across threads. This is most things:

[2]in safe Rust; with `unsafe`, it's UB

`Sync`

`Sync` is a trait implemented by types if it is safe to share a borrow (i.e., `&T`) across threads. This is most things:

- `u8`, `i32`, etc..

[2] in safe Rust; with `unsafe`, it's UB

`Sync`

`Sync` is a trait implemented by types if it is safe to share a borrow (i.e., `&T`) across threads. This is most things:

- `u8`, `i32`, etc..
- `bool`,

[2] in safe Rust; with `unsafe`, it's UB

Sync

Sync is a trait implemented by types if it is safe to share a borrow (i.e., &T) across threads. This is most things:

- u8, i32, etc..
- bool,
- Vec<T>, [T] (when T is Sync)

[2]in safe Rust; with unsafe, it's UB

Sync

Sync is a trait implemented by types if it is safe to share a borrow (i.e., &T) across threads. This is most things:

- u8, i32, etc..
- bool,
- Vec<T>, [T] (when T is Sync)

So what is it not?

[2] in safe Rust; with unsafe, it's UB

`Sync`

`Sync` is a trait implemented by types if it is safe to share a borrow (i.e., `&T`) across threads. This is most things:

- `u8`, `i32`, etc..
- `bool`,
- `Vec<T>`, `[T]` (when `T` is `Sync`)

So what is it not?

- `Rc<T>`—a *non-atomic* reference counted pointer to `T`. If we had a `&Rc<T>`, we could clone the pointer and then modify `T`!

[2] in safe Rust; with `unsafe`, it's UB

Sync

Sync is a trait implemented by types if it is safe to share a borrow (i.e., &T) across threads. This is most things:

- u8, i32, etc..
- bool,
- Vec<T>, [T] (when T is Sync)

So what is it not?

- Rc<T>—a *non-atomic* reference counted pointer to T. If we had a &Rc<T>, we could clone the pointer and then modify T!
- Other runtime checked types which allow mutation via shared borrows like Cell<T> (because they don't synchronise!)

[2] in safe Rust; with **unsafe**, it's UB

`Sync`

`Sync` is a trait implemented by types if it is safe to share a borrow (i.e., `&T`) across threads. This is most things:

- `u8`, `i32`, etc..
- `bool`,
- `Vec<T>`, `[T]` (when `T` is `Sync`)

So what is it not?

- `Rc<T>`—a *non-atomic* reference counted pointer to `T`. If we had a `&Rc<T>`, we could clone the pointer and then modify `T`!
- Other runtime checked types which allow mutation via shared borrows like `Cell<T>` (because they don't synchronise!)

Keep in mind it would still be bad to share an exclusive borrow (`&mut T`) across threads! There's just no way for us to even construct overlapping exclusive borrows to begin with[2].

[2] in safe Rust; with `unsafe`, it's UB

Send

`Send` is a trait implemented by types if it is safe to move its value (i.e., `T`) across threads. Again, this is most things.

In fact, if a type `T` is `Sync`, then `&T` is `Send` (and vice-versa).

Essentially, this is the class of things we can transfer across thread boundaries because they either: (1) don't allow for mutable access to the same location as reachable from elsewhere or (2) ensure such access is protected from occuring in two different execution contexts (think threads) at the same time.

Mutex<T>

The most commonly used method of synchronisation is probably a mutex. One of the biggest difference between Rust and other languages is that Mutex<T> is a container—the data protected by the mutex is inexorably tied to it as part of the type. Let's take a look at the API (some minor simplifications for the slides):

```rust
pub struct Mutex<T> { /* fields omitted */ }

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

    pub fn lock(&self) -> LockResult<MutexGuard<'_, T>>
    pub fn try_lock(&self) -> TryLockResult<MutexGuard<'_, T>>
    pub fn get_mut(&mut self) -> LockResult<&mut T>
}
```

Anything missing?

`Arc<T>`

Frequently `Mutex<T>` will be contained in `Arc`, an atomically incrementing reference counted pointer.

Why might this be?

`Arc<T>`

Frequently `Mutex<T>` will be contained in `Arc`, an atomically incrementing reference counted pointer.
Why might this be? If we want to store a mutex on the stack, lifetimes can become an issue: how do we know how long the mutex lives?

Arc<T>

Frequently Mutex<T> will be contained in Arc, an atomically incrementing reference counted pointer.
Why might this be? If we want to store a mutex on the stack, lifetimes can become an issue: how do we know how long the mutex lives? If we allocate space for this, and know when we're free to get rid of it via reference counting, we can avoid this.
(alternatives include statics, or just leaking memory)

```rust
pub struct Arc<T> { /* fields omitted */ }

impl<T> Arc<T> {
    pub fn new(data: T) -> Arc<T>
    pub fn get_mut(this: &mut Arc<T>) -> Option<&mut T>
    pub fn pin(data: T) -> Pin<Arc<T>> // For next week!
}
```

importantly this also implements Clone and Deref<Target = T>.

The `std::sync` module

Alongside these, the rest of the `std::sync` module has some important types and modules for writing concurrent code:

The `std::sync` module

Alongside these, the rest of the `std::sync` module has some important types and modules for writing concurrent code:

- `RwLock<T>`—platform-agnostic reader-writer locks

The `std::sync` module

Alongside these, the rest of the `std::sync` module has some important types and modules for writing concurrent code:

- `RwLock<T>`—platform-agnostic reader-writer locks
- `Condvar`—platform-agnostic condition variables

The `std::sync` module

Alongside these, the rest of the `std::sync` module has some important types and modules for writing concurrent code:

- `RwLock<T>`—platform-agnostic reader-writer locks
- `Condvar`—platform-agnostic condition variables
- `Barrier`—implements memory barriers, a way for multiple threads to wait at a certain point until all relevant threads reach that point

The `std::sync` module

Alongside these, the rest of the `std::sync` module has some important types and modules for writing concurrent code:

- `RwLock<T>`—platform-agnostic reader-writer locks
- `Condvar`—platform-agnostic condition variables
- `Barrier`—implements memory barriers, a way for multiple threads to wait at a certain point until all relevant threads reach that point
- `Once`—provides thread-safe one-time initialisation for globals/static variables

The `std::sync` module

Alongside these, the rest of the `std::sync` module has some important types and modules for writing concurrent code:

- `RwLock<T>`—platform-agnostic reader-writer locks
- `Condvar`—platform-agnostic condition variables
- `Barrier`—implements memory barriers, a way for multiple threads to wait at a certain point until all relevant threads reach that point
- `Once`—provides thread-safe one-time initialisation for globals/static variables
- `mpsc`—a module with Multi-producer, single-consumer queues for message passing between threads.

The `std::sync` module

Alongside these, the rest of the `std::sync` module has some important types and modules for writing concurrent code:

- `RwLock<T>`—platform-agnostic reader-writer locks
- `Condvar`—platform-agnostic condition variables
- `Barrier`—implements memory barriers, a way for multiple threads to wait at a certain point until all relevant threads reach that point
- `Once`—provides thread-safe one-time initialisation for globals/static variables
- `mpsc`—a module with Multi-producer, single-consumer queues for message passing between threads.
- `atomic`

Counting in Rust

```rust
use std::thread;

const NUM_THREADS: usize = 1000;

fn main() {
    let mut x: i32 = 0;
    let mut threads = Vec::with_capacity(NUM_THREADS);

    for _ in 0..NUM_THREADS {
        threads.push(thread::spawn(|| {
            x += 1;
        }));
    }

    for thread in threads {
        if let Err(e) = thread.join() {
            std::panic::resume_unwind(e);
        }
    }

    println!("After incrementing x {NUM_THREADS} times, it is now {x}");
}
```

Counting in Rust

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

const NUM_THREADS: usize = 1000;

fn main() {
    let x = Mutex::new(0);
    let mut threads = Vec::with_capacity(NUM_THREADS);

    for _ in 0..NUM_THREADS {
        threads.push(thread::spawn(|| match x.lock() {
            Ok(mut x) => *x += 1,
            Err(_) => unreachable!("This function cannot panic, so the mutex cannot be poisoned"),
        }));
    }

    for thread in threads {
        if let Err(e) = thread.join() {
            std::panic::resume_unwind(e);
        }
    }

    let x = x.lock().expect("Can't be poisoned");
    println!("After incrementing x {NUM_THREADS} times, it is now {x}");
}
```

Counting in Rust

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

const NUM_THREADS: usize = 1000;

fn main() {
    let x = Arc::new(Mutex::new(0));
    let mut threads = Vec::with_capacity(NUM_THREADS);

    for _ in 0..NUM_THREADS {
        let x = x.clone();
        threads.push(thread::spawn(move || match x.lock() {
            Ok(mut x) => *x += 1,
            Err(_) => unreachable!("This function cannot panic, so the mutex cannot be poisoned"),
        }));
    }

    for thread in threads {
        if let Err(e) = thread.join() {
            std::panic::resume_unwind(e);
        }
    }

    let x = x.lock().expect("Can't be poisoned");
    println!("After incrementing x {NUM_THREADS} times, it is now {x}");
}
```