
# Parallelism and multithreading

NET7212 — Safe System Programming (in Rust)

Samuel Tardieu  Stefano Zacchiroli

2023-11-14


# Introduction

## What is parallelism?

*Parallelism*, or *parallel computing*, is the process of carrying out several computations or performing actions *simultaneously*.

For example, on a multitasking operating system (Unix, Microsoft Windows), several processes execute concurrently:

- The processes may themselves be made of multiple *threads* executing concurrently.
- Executing threads are mapped onto *processor cores*.
- Even on one core, things happen simultaneously through *time-sharing* since computations and actions are not *atomic*.

## Why parallelism?

Moore's Law (1965) is a prediction that the number of transistors on a microchip would double approximately every two years, leading to a significant increase in computing power.

However:

- The physical limitations of materials and manufacturing processes make it difficult to continue increasing the clock speed of processors at the same rate as in the past.
- As the size of transistors approaches the atomic scale, it becomes increasingly difficult to manufacture them with the required precision and reliability.
- Increasing transistor density leads to increased power consumption and heat generation, limiting performance and lifespan.

There is a trade-off between transistor density and power consumption, making it challenging to continue doubling the number of transistors on a chip every two years without sacrificing performance or reliability.

Parallelism favors the use of multiple processor cores working next to another over increasing the performances of a single core.

## When is parallelism useful?

Parallelism allows for the simultaneous execution of multiple tasks, which can improve performance and efficiency in a variety of contexts:

- It is particularly useful for computationally intensive tasks that can be split into smaller subtasks that can be executed independently and concurrently.
- It can be used to take advantage of the processing power of multiple cores or processors in a single machine, or across multiple machines in a distributed system.
- It can also be used to increase the throughput of I/O-bound tasks, such as network communication or disk access, by allowing multiple tasks to be executed concurrently.
- In some cases, it can reduce energy consumption by reducing the overall computation time and allowing larger periods of system sleep.

Common examples of parallel computing include scientific simulations, machine learning, data analytics, and web server applications that handle high volumes of requests.

However, parallelism also introduces new challenges, such as managing shared resources, ensuring consistency across parallel tasks, and dealing with communication overhead.

## Some challenges caused by parallelism

Parallelism comes with its set of new challenges:

- **Communication overhead**: communication between parallel processes can introduce latency and overhead.
- **Load balancing**: uneven distribution of workload among processing units can lead to underutilization or overutilization.
- **Memory access patterns**: poor memory access patterns can lead to contention and performance degradation.
- **Race conditions**: incorrect results due to multiple processes accessing a shared resource without proper synchronization.
- **Synchronization**: proper synchronization is critical to ensure correct and consistent results in parallel programs.

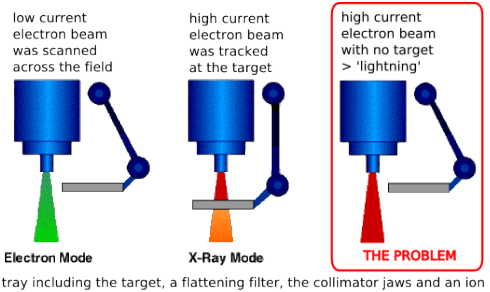
*Race conditions* and *synchronization* will constitute the main topic of this “Safe system programming” class.

# The Therac-25 incident

The Therac-25 incident was a series of accidents in the 1980s caused by a software error in a medical radiation therapy machine that resulted in several patients receiving massive overdoses of radiation, leading to severe injuries and deaths.

The machine was controlled by software and had two modes: electron mode, and X-ray mode.

*(credits AECL)*




tray including the target, a flattening filter, the collimator jaws and an ion chamber was moved OUT for "electron" mode, and IN for "photon" mode.

*(credits wikidot)*

## The Therac-25 incident breakdown

- The Therac-25 incident was caused by a race condition in the machine's software.
- The race condition occurred when the operator quickly changed between different treatment modes, causing the software to skip safety checks and deliver the maximum dose of radiation to the patient.
- The incident caused several patients to suffer severe radiation burns and injuries, and several died as a result of the overdoses.
- The incident highlighted the importance of thorough software testing and safety checks, particularly in safety-critical systems such as medical devices.

## The Northeast Blackout of 2003

A race condition occurred in the alarm processing software of an energy management system during the Northeast Blackout of 2003:

- The race condition occurred when two processes accessed a shared flag variable that represented whether certain alarms had been inhibited (temporarily disabled).
- The flag variable was not properly synchronized, which led to an incorrect state in the system and caused it to misinterpret the state of the power grid.
- The incorrect state triggered a chain reaction of failures in the power grid that eventually led to a widespread blackout affecting over 50 million people in the northeastern United States and Canada.

The incident highlighted the importance of proper software testing and quality assurance in critical infrastructure systems, as well as the need for better communication and coordination among stakeholders.

# Parallelism in practice

## A powerful accounting system in C

Let's simulate two bank accounts `a` and `b`, with a function `transfers()` which:

- transfers 10 units from `a` to `b` if possible,
- then transfers back 10 units from `b` to `a` if possible.
- 10,000 times in a row (we honor our clients wishes, however weird they may be)

```c
#include <assert.h>

static volatile int a = 100;
static volatile int b = 0;

static void transfers() {
  for (int i = 0; i < 10000; i++) {
    if (a >= 10) { a -= 10; assert(a >= 0); b += 10; }
    if (b >= 10) { b -= 10; assert(b >= 0); a += 10; }
  }
}

int main() {
  transfers();
}
```

The main program will run forever: no assertion will ever fail.

Our new accounting software is so great, we should make ten transactions in parallel, that should work as well and take less time!

We are coding in C, so let us use *pthreads*.

# Let's make our accounting software parallel

```c
#include <assert.h>

#include <pthread.h>

static volatile int a = 100;
static volatile int b = 0;

static void* transfers(void *ignored) {
  for (int i = 0; i < 1000; i++) {
    if (a >= 10) { a -= 10; assert(a >= 0); b += 10; }
    if (b >= 10) { b -= 10; assert(b >= 0); a += 10; }
  }
}

int main() {
  pthread_t threads[10];
  for (int i = 0; i < 10; i++)
    pthread_create(&threads[i], NULL, transfers, NULL);
  for (int i = 0; i < 10; i++) // Wait for threads end
    pthread_join(threads[i], NULL);
}
```

10 threads execute in parallel:

- `pthread_create()` creates a new thread which starts executing concurrently.
- Everything should be fine: we always take 10 units from an account if it is there.

⚠ Sometimes, but not always, we get something like:

```
accounting: main.c:10: transfers:
  Assertion `b >= 0' failed.
```

Any of the two assertions may trigger randomly. So far for the future of our accounting business.

# Why did it fail?

How is it possible to get

```
accounting: main.c:10: transfers:
  Assertion `b >= 0' failed.
```

when this line

```
11      if (b >= 10) { b -= 10; assert(b >= 0); a += 10; }
```

first checks that `b` is at least 10 before decreasing it?

Threads might be suspended by the operating systems at arbitrary times. If `b` contains 10, two threads executing simultaneously might successfully check that `b >= 10`. They will both execute `b -= 10`; if the execution of this operation happens first in one thread then in the second one, `b` will contain `-10`.

We need to wrap the test `b >= 10` and the operation `b -= 10` in a critical section.

## Critical section

By defining critical sections, protected by a *mutex*, we can serialize code paths and prevent their concurrent execution:

```
6  static pthread_mutex_t mutex =
7    PTHREAD_MUTEX_INITIALIZER;
8
9  static void* transfers(void *ignored) {
10   for (int i = 0; i < 1000; i++) {
11     pthread_mutex_lock(&mutex);
12     if (a >= 10) { a -= 10; assert(a >= 0); b += 10; }
13     pthread_mutex_unlock(&mutex);
14
15     pthread_mutex_lock(&mutex);
16     if (b >= 10) { b -= 10; assert(b >= 0); a += 10; }
17     pthread_mutex_unlock(&mutex);
18   }
19 }
```

The `mutex` variable enforces *mutual exclusion*.

⚠ The outcome of the program (the value in each account `a` and `b`) is likely to be different from the version without threads, as threads will progress independently at potentially different speeds.

Let's try to rewrite our accounting system in Rust.

# Basic accounting software in Rust

Here is the same program with only the main thread in Rust:

```
1  fn transfers(a: &mut u64, b: &mut u64) {
2      for _ in 0..10_000 {
3          if *a >= 10 { *a -= 10; *b += 10; }
4          if *b >= 10 { *b -= 10; *a += 10; }
5      }
6  }
7
8  fn main() {
9      let (mut a, mut b) = (100, 0);
10     transfers(&mut a, &mut b);
11 }
```

- The variables cannot be global as using mutable static variables is unsafe in Rust.
- Assertions are not needed as the subtraction will fail if `a` or `b` ever attempt to go out of the `u64` range.

## Maybe we could have used unsigned types in C instead of a mutex?

What if we had used `unsigned int` instead of `int` in C? Could we get rid of the mutex?

```c
#include <assert.h>
#include <pthread.h>

static volatile unsigned int a = 100;
static volatile unsigned int b = 0;

static void* transfers(void *ignored) {
  for (int i = 0; i < 1000; i++) {
    if (a >= 10) { a -= 10; assert(a >= 0); b += 10; }
    if (b >= 10) { b -= 10; assert(b >= 0); a += 10; }
  }
}

int main() {
  pthread_t threads[10];
  for (int i = 0; i < 10; i++)
    pthread_create(&threads[i], NULL, transfers, NULL);
  for (int i = 0; i < 10; i++) // Wait for threads end
    pthread_join(threads[i], NULL);
}
```

The program would not crash anymore, as `a` and `b` cannot be negative. However, they will underflow and get an extremely large value. We are going to get ruined when the client comes and cash their account.

⚠ This is a known attack scenario, getting goods by underflowing a counter.

This was not a good idea. Back to Rust.

## Threads in Rust

Rust has two kind of threads:

- **regular threads,** similar to pthreads, which live their own lifes after they are started;
- **scoped threads,** which are started from a scope, and must terminate before their creator can live this scope.

We will start with regular threads:

- `std::thread::spawn()` takes a parameterless function returning a `T` and executes it on a thread;
- it immediately returns a `std::thread::JoinHandle<T>`;
- the handle can be used to wait for the thread termination and retrieve its computed value, or to enquiry about the thread status.

If you do not care about the thread once you have created it, you can forget about the `JoinHandle`.

# Rust threads: an example

The following program creates 5 threads which all display their number after some time has passed:

```rust
use std::{thread, time::Duration};

fn main() {
    let mut handles = vec![];
    for i in 1..=5 {
        let handle = thread::spawn(move || {
            thread::sleep(Duration::from_secs(6 - i));
            println!("In thread number {i}");
        });
        handles.push(handle);
    }
    println!("Waiting for the threads to terminate");
    handles.into_iter().for_each(|h| h.join().unwrap());
    println!("All threads have terminated");
}
```

```
Waiting for the threads to terminate
In thread number 5
In thread number 4
In thread number 3
In thread number 2
In thread number 1
All threads have terminated
```

Why did we have to use `move` in the closure?

Because `i` is referenced from the closure and has long ceased to exist when it is time to display it.

## Closures and captures

Closures in Rust capture from their environment:

- every variable used inside the closure if ownership is required;
- a non-mutable reference to every variable used by reference inside the closure;
- a mutable reference to every variable used as a mutable reference inside the closure.

```
1 let (a, b, c) = (String::from("A"), &10, &mut 42);
2 let f = || { *c += *b; if *c > 100 { let s = a; println!("s = {s}")}};
```

- `a` is moved to `s` inside the closure, this requires ownership of `a`, so `a` is captured in `f` and unavailable after the definition of `f`.
- `b` is dereferenced, so `&b` is captured (`b` is borrowed); `f` lifetime cannot be longer than the one of `b`.
- `c` is mutably dereferenced, so `&mut c` is captured (`c` is mutably borrowed); `f` lifetime cannot be longer than the one of `c`, and `c` is unusable outside `f` as long as `f` is alive.

Using `move` in front of the closure forces it to capture all the referenced variables (here it would be `a`, `b` and `c`).

## Closures and captures (cont.)

How is `i` captured in our example?

- `6 - i` uses `i32::sub(self, other: &i32)`, so `i` is used by reference;
- `println!()` is a macro and does not consume its arguments, it uses them by reference.

```
5       for i in 1..=5 {
6           let handle = thread::spawn(move || {
7               thread::sleep(Duration::from_secs(6 - i));
8               println!("In thread number {i}");
9           });
```

`i` would be captured by reference. However, a regular thread can run potentially forever, so its main function (the closure here) needs to have a `'static` lifetime. `i`'s lifetime ends at the end of the current loop iteration, so the closure cannot live longer.

Using `move` forces the closure to capture `i`. As `i32` implements `Copy`, a copy of the current value of `i` is captured as `i` in the closure. The closure's lifetime is now `'static` as it doesn't borrow any variable.

## Closures and captures (cont.)

A closure automatically implements certain traits, depending on what is captured:

- If all captures are `Clone` (resp. `Copy`), the closure implement `Clone` (resp. `Copy`). Note that immutable references implement both.
- If all captures implement `Send` (resp. `Sync`), the closure implements `Send` (resp. `Sync`). Note that immutable references to `Sync` types are `Send` and `Sync` themselves.

To be able to be used on another thread than the one which creates it, a closure must implement `Send`. But what are `Send` and `Sync` exactly?

## More about `Send` and `Sync`

`Send` and `Sync` are two marker (without functions) auto (automatically derived) traits:

- Implementing `Send` means that it is safe to send an object of this type to the stack of another thread and use it from there.
- Implementing `Sync` means that it is safe to reference an object of this type from another thread.

`T` implements `Sync` if and only if `&T` implements `Send`. A new reference to an object is necessarily created from the thread where the object is. Sending the reference to another thread (`&T: Send`) means that this reference can be used there (`T: Sync`).

`Send` and `Sync` are not automatically implemented for types containing raw pointers or non `Send`/`Sync` fields. You can however implement them yourself, but this is an unsafe operation:

```
unsafe impl Send for MyTypeContainingRawPointers {}
```

## Threads and lifetimes

What are exactly the constraints on the `std::thread::spawn()` argument?

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

- The main function of the new thread is of type `F`.
- `F` implements `FnOnce()`: this represents parameterless functions returning `()` that will be called only once. Indeed, this function is the main thread function, `spawn()` will only call it once.
- A function of type `F` takes no argument and returns a value of type `T` (possibly `()`).
- `F` instances' lifetime is `'static`: the thread can live forever.
- `F` instances are `Send` and can be sent to another thread: of course, they will be sent to a new thread to be executed there!

The result of type `T` must be sent to the thread calling `.join()` on the join handle at a later point in the future, so `T` must be both `'static` and `Send`.

## Interlude: function traits

Three function traits are automatically implemented on closures if they fit:

- `Fn` designates all the closures that don't consume a captured variable and don't mutably reference a variable from the environment. They can only read from their environment and can be called several times. They implement `FnOnce` and `FnMut` automatically.
- `FnMut` designates all the closures that don't consume their captured variables, so they can be called several times. They implement `FnOnce` automatically.
- `FnOnce` designates closures that can be called only once by their owner (they will be consumed by the call). They can consume their captured variables.

What function traits do `a`, `b` and `c` implement?

```
1 let mut s = String::from("Hello, world");
2 let a = || s.push('!');
3 let b = || println!("s = {s}");
4 let c = || std::mem::drop(s);
```

- `a` uses `&mut s`: `FnMut` and `FnOnce`.
- `b` uses `&s` in `println!()`: `Fn`, `FnMut` and `FnOnce`.
- `c` consumes `s`: `FnOnce` only.

## Interlude: function traits (cont.)

The trait definition describes the parameters and return type (if any) of the function. For example, in the `Fn` family:

- `Fn()` receives no parameter and returns `()`
- `Fn(&'static str) -> (u32, String)` is implemented on closures taking a reference to a `str` with a `'static` lifetime and returning a couple of an unsigned 32 bit integer and an owned string

There also exist function pointer types to describe regular functions defined with the `fn` keywords. For example, `fn(&str) -> usize` is the type of `str::len(&self)`.

All `fn` functions implement `Fn` and thus `FnMut` and `FnOnce`; `str::len` can be passed when a `FnOnce(&str) -> u32` is expected for example.

# Interlude: function traits (cont.)

What if we want to build a `foreach()` function that calls a given function on every element of a vector?

```
1  fn foreach<T, F: Fn(T)>(v: Vec<T>, f: F) {
2      for e in v {
3          f(e);
4      }
5  }
6
7  fn main() {
8      foreach(vec![1, 2, 3], |e| println!("- {e}"));
9  }
```

will print all three elements of the vector.

But what if we want to print the partial sum of the elements (1, 3, and 6)?

## Interlude: function traits (cont.)

To print the partial sum, we need to capture a variable in which we will store the partial sum:

```
1  fn foreach<T, F: Fn(T)>(v: Vec<T>, f: F) {
2      for e in v {
3          f(e);
4      }
5  }
6
7  fn main() {
8      let mut total = 0;
9      foreach(vec![1, 2, 3], |e| {
10         total += e;    // i32::add_assign(&mut self, other: i32)
11         println!("- {total}")
12     });
13 }
```

However, it will not compile: our closure captures `total` by mutable reference and implements only `FnMut(i32)` and `FnOnce(i32)`, not `Fn(i32)`.

`foreach()` needs to accept a `FnMut(T)`.

# Interlude: function traits (cont.)

`f` also needs to be mutable to be able to mutate its environment (here: the `total` variable defined in `main`):

```
1  fn foreach<T, F: FnMut(T)>(v: Vec<T>, mut f: F) {
2      for e in v {
3          f(e);
4      }
5  }
6
7  fn main() {
8      let mut total = 0;
9      foreach(vec![1, 2, 3], |e| {
10         total += e;
11         println!("- {total}")
12     });
13     // The closure has ceased to exist at this point, so it no longer
14     // mutably borrows total, we can use it again.
15     println!("The final total is {total}");
16 }
```

```
- 1
- 3
- 6
The final total is 6
```

But what if we add `move`?

```
move |e| {
    total += e;
    println!("- {total}");
}
```

```
- 1
- 3
- 6
The final total is 0
```

## Back to the accounting software

We are now equipped to attempt to parallelize our accounting software.

```
fn transfers(a: &mut u64, b: &mut u64) {
    for _ in 0..10_000 {
        if *a >= 10 { *a -= 10; *b += 10; }
        if *b >= 10 { *b -= 10; *a += 10; }
    }
}

fn main() {
    let (mut a, mut b) = (100, 0);
    transfers(&mut a, &mut b);
}
```

From what we have seen so far, we already know that something like the following will not work as a and b don't have a static lifetime and cannot be mutably borrowed by every thread main function:

```
    for _ in 0..10 {
        std::thread::spawn(|| transfers(&mut a, &mut b));
    }
```

## Solutions to explore

We could try the following things:

- Put `a` and `b` in the same structure. It is a good start, but will not be enough, as we would have to borrow the structure mutably from every thread.
- Protect the structure with a mutex. Surely Rust must have one? Yes: `std::sync::Mutex`.
- Put the structure inside a special clonable smart pointer type that can give us access from multiple thread. That would be `std::sync::Arc` (*Atomically Reference Counted*).

By combining all those components, we should be able to implement our parallel accounting software.

Let's start by looking at the mutex.

## std::sync::Mutex

In C, pthread's mutex needs to be locked and unlocked to implement a critical section. This is unsafe by itself for several reasons:

- If one forgets to unlock the mutex on any code path, it will stay locked forever.
- If one thread crashes while the mutex is locked, others waiting for the mutex will wait forever.
- If one modifies the variables protected by the mutex without locking it first, the mutex becomes useless (i.e., pthread's mutexes implement *advisory locking* rather than *mandatory locking*).

In Rust:

- The `std::sync::Mutex` instance becomes the owner of the protected data. The data is no longer available except by locking the mutex (= mandatory locking).
- Locking the mutex returns a `MutexGuard`: this is a smart pointer implementing `Deref` and `DerefMut` for the protected data.
- Dropping the `MutexGuard` unlocks the mutex.
- Panicking while holding the `MutexGuard` makes it enter a *poisoned* state: other threads waiting for the mutex will be made aware of this state.

## std::sync::Mutex: examples

We can use a mutex to protect our grouped accounts:

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

struct Accounts { a: u64, b: u64 }

fn transfer_a_to_b(accounts: &Mutex<Accounts>) {  // Note: no need for &mut Mutex<Accounts>
    let mut accounts = accounts.lock().unwrap();
    // Now accounts is a MutexGuard<Accounts>
    accounts.a -= 10;
    accounts.b += 10;
    // Here the MutexGuard is dropped, and the mutex is unlocked
}

fn main() {
    let accounts = Mutex::new(Accounts { a: 100, b: 0 });
    transfer_a_to_b(&accounts);
    // Lock the mutex only for the duration of the statement:
    assert_eq!(accounts.lock().unwrap().a, 90);
}
```

We need to make the mutex `'static`: enter `std::sync::Arc`.

## std::sync::Arc

`Arc<T>` is a smart pointer:

- It and its clones *together* own an object of type `T`.
- It implements `Deref` towards `T` (but not `DerefMut`).
- It can be cloned: this will increment the number of references (starting at 1 at `Arc` creation).
- It implements `Drop`, decrementing the number of references upon drop. When the number of references reaches 0, its content is dropped as well.
- If `T` implements `Send` and `Sync`, `Arc<T>` does as well and can be sent to other threads or references from there.

Remember that `lock()` on a `Mutex` only requires an immutable reference (`&self`): `Arc<Mutex<T>>` is an ideal choice for sharing access to an object of type `T` while keeping it synchronized.

# The main program

The `main()` function is in charge of building the `Arc<Mutex<Accounts>>` and clones it (incrementing the reference count) for every thread:

```
1  use std::{sync::{Arc, Mutex}, thread};
2
3  struct Accounts { a: u64, b: u64 }
4
5  // Batch of transfers from a to b and b to a
6  fn transfers(accounts: &Mutex<Accounts>) {
7      todo!()
8  }
9
10 fn main() {
11     let accounts = Arc::new(Mutex::new(Accounts { a: 100, b: 0 }));
12     let handles: Vec<_> = (0..10).map(|_| {
13         let accounts = Arc::clone(&accounts);        // Increment reference count by cloning
14         thread::spawn(move || transfers(&accounts))  // Capture the cloned accounts variable
15     }).collect();
16     handles.into_iter().for_each(|h| h.join().unwrap());
17 }
```

## The transfers function

The `transfers()` function makes transactions in a loop if there are enough monetary units in the source account:

```
1  // Batch of transfers from a to b and b to a
2  fn transfers(accounts: &Mutex<Accounts>) {
3      for _ in 0..1_000 {
4          {
5              let mut accounts = accounts.lock().unwrap();
6              if accounts.a >= 10 { accounts.a -= 10; accounts.b += 10; }
7          } // unlock
8          {
9              let mut accounts = accounts.lock().unwrap();
10             if accounts.b >= 10 { accounts.b -= 10; accounts.a += 10; }
11         } // unlock
12     }
13 }
```

Note that `transfers()` never notices that the `Mutex<Account>` comes from an `Arc`, thanks to the automatic coercion done by `Deref` when calling `transfers(&accounts)`.

## Reflections on threads

In our code, some of the issues came from Rust's requirement that functions given to `std::thread::spawn()` have a `'static` lifetime. Such requirement exists because threads may run indefinitely, continuing well after the captured local variables have been destroyed:

```
1  fn main() {
2      let accounts = Arc::new(Mutex::new(Accounts { a: 100, b: 0 }));
3      let handles: Vec<_> = (0..10).map(|_| {
4          let accounts = Arc::clone(&accounts);        // Increment reference count by cloning
5          thread::spawn(move || transfers(&accounts))  // Capture the cloned accounts variable
6      }).collect();
7      handles.into_iter().for_each(|h| h.join().unwrap());
8  }
```

However, we *know* that our threads terminate before our current function (`main()`) and we would like the threads to necessitate a shorter lifetime for their closure.

This is a job for the *scoped threads*.

## Scoped threads

Scoped threads, a recent addition to Rust, are different from regular threads:

- They are launched by a method of a local `Scope` object created by `std::thread::scope()`.
- The lifetime of their closure needs to be at least the scope object's lifetime instead of `'static`.
- When the scope object is dropped, its destructor waits for all the scoped threads which have not been joined to terminate.

Let's enjoin the disappearance of the `Arc`:

```
1  fn main() {
2      let accounts = Mutex::new(Accounts { a: 100, b: 0 });
3      thread::scope(|s| {
4          for _ in 0..10 {
5              s.spawn(|| transfers(&accounts));
6          }
7      });  // Automatically wait for the termination of all 10 scoped threads
8  }
```

Scoped threads are handy for short-lived parallelization, but regular threads are more appropriate for creating long-lived threads.

## What we have learned so far

- C threads offer no guarantees that data synchronization is used.
- pthreads mutexes commonly used in C may be left in a locked state forever and require extra attention.
- Rust makes it impossible to concurrently access mutable data from different threads in safe mode.
- Rust allows you to create long-lived threads, or short-lived threads with the guarantee that the whole scope stack will stay valid.
- Even with proper synchronization, the outcome of a parallel program may not be the same as a sequential program performing the same operations in a different order. This is true in any language.

However, we are only at the beginning of what Rust has to offer in terms of concurrency safety.

# Rust tools for a concurrent world

## Getting a thread outcome

So far we have ignored the value returned by our threads. Let's illustrate the use of the `JoinHandle` to get back the result:

```
1  fn main() {
2      let h1 = std::thread::spawn(|| fibo(20));
3      let h2 = std::thread::spawn(|| fibo(30));
4      let h3 = std::thread::spawn(|| fibo(40));
5      let total = h1.join().unwrap() + h2.join().unwrap() + h3.join().unwrap();
6      println!("fibo(20) + fibo(30) + fibo(40) = {total}");
7  }
```

All three computations happen in parallel on their own thread (which may or may not be mapped to a different processor core, depending on the availability).

# Catching a panic

A panic in Rust may, depending on the configuration of the compiled program:

- stop the current thread and unwind its call stack, calling destructors as needed;
- stop the whole program immediately without unwinding.

In the first (and most common) case, a thread panicking will return an `Err()` variant when joined:

```
1 fn main() {
2     let handle = std::thread::spawn(|| panic!("Panicking hard"));
3     if handle.join().is_err() {
4         println!("The thread has panicked");
5     }
6 }
```

## Multiple producer single consumer (MPSC) channels

A `std::sync::mpsc::Channel<T>` is an infinite channel made of two parts:

- a `Sender<T>` which can be cloned and implements `Send` if `T` does;
- a blocking `Receiver<T>` which cannot be cloned but implements `Send` if `T` does.

This can be used to exchange data between threads (if `T: Send`). Here every thread sends two values:

```
1  fn main() {
2      let (tx, rx) = std::sync::mpsc::channel::<i32>();
3      for i in 1..=3 {
4          // Clone the sender side to capture it
5          let tx = tx.clone();
6          std::thread::spawn(move || {
7              tx.send(i).unwrap(); tx.send(i*10).unwrap();
8          });
9      }
10     // rx.recv() blocks while waiting for data
11     while let Ok(x) = rx.recv() {
12         println!("- {x}");
13     }
14 }
```

```
- 1
- 2
- 10
- 20
- 3
- 30
```

- Dropping the receiver makes `tx.send()` return an error
- Dropping all the senders make `rx.recv()` return an error instead of blocking

## When a Mutex is too costly: RwLock

A `Mutex` forces all access to the data to be serialized. In situations where data is more often read than written, this prevents threads from doing concurrent reads.

`std::sync::RwLock<T>` provides two methods to access the data:

- `.read()` returns a smart pointer which implements `Deref` with `T` as a target. This may block until a write locks exist.
- `.write()` returns a smart pointer implementing `DerefMut` with `T` as a target. This may block until it can get exclusive access.

Depending on the underlying operating system, `RwLock<T>` may be susceptible to starvation issues, for example if readers keep arriving while a writer is waiting.

## When an Arc is too costly: Rc

An `Arc<T>` (*Atomically Reference Counted*) allows you to access an object of type `T` from different places, by keeping a count of the existing clones of the `Arc`.

Internally, `Arc<T>` uses synchronized memory operations to ensure an accurate reference count even when used concurrently from several processor cores. This may be overkill if no simultaneous clones or drops can happen because all uses come from the same thread.

In this case, `std::rc::Rc<T>` (*Reference Counted*) offers the same services as `Arc<T>` but does not implement `Send` or `Sync`: it can only be used on the thread where it has been created.

Just like `Arc<T>`, `Rc<T>` implements `Deref` with `T` as a target (but not `DerefMut`). Does that mean that `Rc<T>` can never be used to modify a `T`, since we can only get a `&T` and no `&mut T`?

It is time to introduce *interior mutability*.

## Interior mutability

Interior mutability is a concept which allows modifying data for which we only have an immutable reference.

Wait, what? Isn't it Rust goal to ensure that an immutable reference cannot be used to alter the pointed data? Indeed, but…

Interior mutability is done through special types such as `std::cell::RefCell<T>`:

- The `.borrow()` method returns a smart-pointer implementing `Deref` which can be used to access the data.
- The `.borrow_mut()` method returns a smart-pointer implementing `DerefMut` which can be used to modify the data.
- Both those methods panic if a mutable smart-pointer is obtained at the same time as any of those smart-pointer.

When using `RefCell`, the verification is done at run-time instead of at compile-time. At no point a mutable reference to the data will exist at the same time as any other reference.

## Combining Rc and RefCell

Using Rust basic blocks such as `Rc` and `RefCell`, one can build powerful and safe data structures. In this example, a `TreeNode` is reachable through its parent or its children:

```
1  struct TreeNode<T> {
2      value: T,
3      children: Vec<TreeNodePtr<T>>,
4      parent: Option<TreeNodePtr<T>>,
5  }
6
7  type TreeNodePtr<T> = Rc<RefCell<TreeNode<T>>>;
8
9  fn add_value<T>(parent: &TreeNodePtr<T>, value: T) { // Add a new child node to the parent
10     let child = Rc::new(RefCell::new(TreeNode {
11         value, children: vec![], parent: Some(Rc::clone(parent))
12     }));
13     parent.borrow_mut().children.push(child);
14 }
15
16 fn remove_value<T>(parent: &TreeNodePtr<T>, nth: usize) { // Remove the nth child from the parent
17     let child = parent.borrow_mut().children.remove(nth);
18     child.borrow_mut().parent = None;
19 }
```

## More on interior mutability

We have encountered interior mutability already:

- Through a `&Mutex<T>`, you are able to obtain a `&mut T` by locking the mutex.
- `Arc<T>` and `Rc<T>` only give you a `&T`, but they increment their `counter` field when cloned (through a `&Arc<T>` or `&Rc<T>`).

In addition to `std::cell::RefCell<T>`, other types exist to implement interior mutability:

- `std::cell::Cell<T>` lets you atomically exchange the owned data with another of the same type through a `&Cell<T>`.
- `std::cell::UnsafeCell<T>` is the basic unsafe building block for building `Cell<T>` and `RefCell<T>`. It gives you a `*mut T` to the owner data through its `.get()` method.

`Cell` and `RefCell` are the safe ways of implementing interior mutability in your own types. However they only work on a single thread as they do not implement `Send`.

## Efficient interior mutability in a multithreaded environment

In a multithreaded environment, one can use raw mutable pointers to implement interior mutability.
Here is an example of a reference counted shared data similar to `Arc`:

```
struct Shared<T> {
    inner: *mut Inner<T>,
    phantom: PhantomData<Inner<T>>,  // Not explained here: dropping a Shared<T> might drop a <T>
}

impl<T> Clone for Shared<T> { … }                  // Increment the counter
impl<T> Drop for Shared<T> { … }                   // Decrement the counter
unsafe impl<T: Send + Sync> Send for Shared<T> {} // Allow sending a Shared<T> onto another thread

impl<T> Shared<T> {
    /// Unsafety: two &mut T must never exist at the same time
    unsafe fn get_mut(&self) -> &mut T { todo!() }
}

struct Inner<T> {
    data: T,                                       // The owned data
    counter: std::sync::atomic::AtomicUsize,       // The reference count
}
```

## Alternatives to multithreading in Rust

Multithreading is not the only form of parallel execution supported by Rust. Those will be studied later in the course.

### Multiprocessing

By using the services of the underlying operating system, Rust can spawn processes and make them communicate. Those processes will not share memory, except for specially designated areas.

### Asynchronous programming

Asynchronous programming allows multiple concurrent executions to happen on a single thread. This is appropriate when the program is I/O bound: using a thread for every I/O communication channel will cause having many threads waiting on I/O, while a few threads would be sufficient to check on the various channel readyness.

## Conclusions

Rust has many tools to let you benefit from multithreading in a safe way (regular and scoped threads, `Send`, `Sync`, `Arc<T>`, `Mutex<T>`, `RwLock<T>`).

When data is shared only within a single thread, specialized tools allow you not to pay the cost of inter-thread synchronization (`Rc<T>`, `Cell<T>`, `RefCell<T>`).

Unsafe constructs let you build hand-designed tools that suit your needs (raw pointers, `UnsafeCell<T>`, atomic data types).

You can do everything you would be able to do in C with pthreads and pointers by going `unsafe`, but usually staying in the safe world is enough and performances will live up to your expectations as the overhead has been kept minimal.