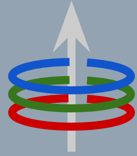


CoE 164

Computing Platforms

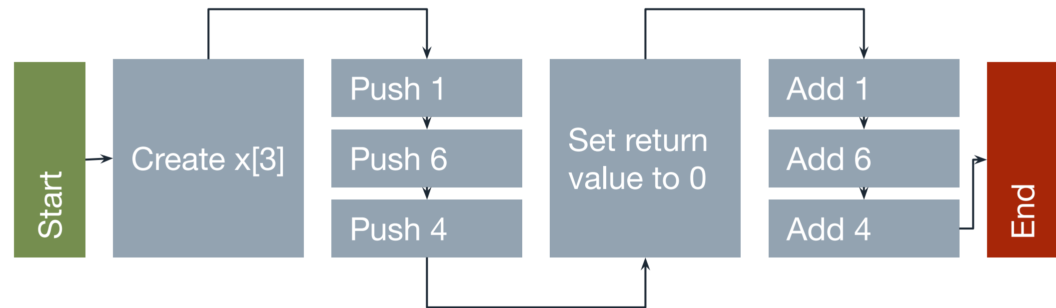
12a: Rust Concurrency

SEQUENTIAL PROGRAMMING

So far, you have been taught that each line of your code is executed *sequentially*. It’s like a series of commands the computer just executes one after another.

SEQUENTIAL PROGRAMMING

```
fn main() {
    let x[i64 ;3] = [1, 6, 4];
    println!("{}", x[0] + x[1] + x[2]);
}
```



BEYOND SEQUENTIAL PROGRAMMING

It takes time to execute a long list of instructions especially if there is only a single CPU working on it!

What if we split our instructions such that we can execute them faster?

If we are limited to *only one* CPU, is there a way that we can maximize the time it runs our instructions?

PARALLEL PROGRAMMING

In comparison to sequential programming, *parallel programming* uses multiple computing modules to solve a problem.

It saves time because it can now execute tasks at the same time - *multitasking*!

It enables *concurrency*!

CONCURRENCY AND PARALLELISM

Parallelism

- Execute instructions at the same time
- Split tasks and run them simultaneously

Concurrency

- Execute and suspend instructions
- Split tasks and run them according to schedule

PROCESSES AND THREADS

An executed program's code is run in a *process*. Multiple subprograms that run independently called *threads* can be spawned by the program.

THREADS

A program can spawn a thread using the `thread::spawn` function with its only argument being the closure containing the subroutine to run in the thread.

Note that the calling program is also a thread on its own.

```
use std::thread;


thread::spawn(|| {
    println!("hello from thread!");
});
```

Example

THREADS: SAMPLE CODE

Example

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

fn main() {
    thread::spawn(|| {
        for i in 1..=10 {
            println!("<spwn_t> {}!", i);
            thread::sleep(Duration::from_millis( 1));
        }
    });

    for i in 1..=5 {
        println!("<main_t> {}!", i);
        thread::sleep(Duration::from_millis( 1));
    }
}
```

THREADS: SAMPLE CODE

Running the code snippet previously will display interleaved prints from both the main and spawned threads similar to the one on the right.

This happens because the two threads run independently.

If the main thread finishes executing, all of its spawned threads are also terminated.

Example

```
<spwn_t> 1!
<main_t> 1!
<spwn_t> 2!
<spwn_t> 3!
<main_t> 2!
<spwn_t> 4!
<spwn_t> 5!
<main_t> 3!
<main_t> 4!
<main_t> 5!
<spwn_t> 6!
```

THREADS: JOINING

The `thread::spawn` function returns a `JoinHandle`. We can use it to force the main thread to wait for the thread to finish executing before moving on.

Note that the main thread *blocks* while joining.

The thread panicked if the `JoinHandle` returns an `Err`.

```
let t = thread::spawn(|| {
    println!("hello from thread!");
});

if let Err(e) = t.join() {
    println!("Error during wait!");
}
```

Example

THREADS: DATA

In Rust, *only one thread* should have a copy of a piece of data. To transfer ownership of data to a spawned thread, the `move` keyword should be added before the closure.

Example

```
let v = vec![1, 6, 4];
let h = thread::spawn(move || {
    println!("vec [{:?}]", v);
});


// Compile error below
println!("vec [{:?}]", v);
```

THREAD COMMUNICATION

There are times when we need threads to share or transfer data. During this time, both threads need to be *synchronized*.

Rust usually treats thread synchronization in terms of *process interaction*.

PROCESS INTERACTION

Programs can be further divided into different categories:

- Message passing
  - Like postal mail
- Shared space
  - Like bulletin boards

MESSAGE PASSING

The `std::sync::mpsc` library provides for creation and management of a *multiple-producer single-consumer* channel where threads can send and receive data.

The `channel()` function returns a tuple with the "transmitter" and "receiver", respectively. Channels can only send and receive data of a specific type.

```
use std::sync::mpsc;

let my_str = String::from("hello!");
let (tx, rx) = mpsc::channel();
tx.send(my_str).unwrap();
```

Example

MESSAGE PASSING: SAMPLE CODE

Example

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

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hello!");
        tx.send(val).unwrap();
    });

    if let Ok(x) = rx.recv() {
        println!("Received \"{x}\" from child thread!");
    }
}
```

MESSAGE PASSING: SAMPLE CODE

Running the sample code will let the spawned thread send a string to the main thread.

Note that data is *moved* once it is sent through a channel.

Example

```
thread::spawn(move || {
    let val = String::from(
        "Hello!"
    );
    tx.send(val).unwrap();

    // compile error!
    println!("{}", val);
});
```

MESSAGE PASSING: MULTIPLE PRODUCERS

Multiple threads can "share" a single transmitter by cloning it and transferring each clone to their respective threads.

Additionally, we can receive all possible data from the channel by iterating through the receiver. The loop stops if the channel is closed.

The order of the data received is arbitrary.

Example

```
let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
    let val = String::from("1!");
    tx1.send(val).unwrap();
});

thread::spawn(move || {
    let val = String::from("2!");
    tx.send(val).unwrap();
});

for each_recv in rx {
    println!("> {}", each_recv);
}
```

SHARED SPACE: MUTEX

The `std::sync::Mutex` struct hides a data under a *mutual exclusion (mutex)* lock. The `lock()` *blocking* method should be called to "unwrap" the data it is hiding. Data inside a mutex lock can be edited when a lock has been acquired.

Sometimes, a mutex should be dropped manually after use using to `drop()` function to unlock it.

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

let my_str = Mutex::new(String::from("hello!"));

if let Ok(mut d) = my_str.lock() {
    *d = String::from("world!");
}
```

Example

SHARED SPACE: MUTEX

Mutexes are more useful when used across several threads. For this more common case, the mutex should be wrapped inside an *atomically referenced counted (Arc)* smart pointer.

An Arc-wrapped mutex should be cloned before sending it to a thread that will use that mutex. The Arc pointer will automatically count how many threads use the mutex.

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

let my_str = Arc::new(Mutex::new( String::from("hello!")));

if let Ok(mut d) = my_str.lock() {
    *d = String::from("world!");
}
```

Example

SHARED SPACE: SAMPLE CODE

Example

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

fn main() {
    let count_lock = Arc::new(Mutex::new(0));

    for _ in 0..10 {
        let count_lock_a = Arc::clone(&count_lock);
        thread::spawn(move || {
            if let Ok(mut i) = count_lock_a.lock() {
                *i += 1;
                println!("Count: {i}");
            }
```

SHARED SPACE: SAMPLE CODE

Example

```
            thread::sleep(Duration::from_millis(10));
        });
    }

    while let Ok(i) = count_lock.lock() {
        if *i >= 10 {
            println!("Final count: {i}");
            break;
        }
    }
}
```

SHARED SPACE: SAMPLE CODE

Running the code snippet previously will display prints from all of the 10 spawned threads *in ascending order* plus the main thread. Each thread adds one to a shared counter locked behind a mutex.

Example

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Count: 6
Count: 7
Count: 8
Count: 9
Count: 10
Final Count: 10
```

SHARED SPACE: POISONED MUTEX

A mutex is **poisoned** if one of the threads holding it panics. In this case, other threads waiting for the mutex to be open will throw an `Err` instead and will not be able to access the data in it. A poisoned mutex may mean that some invariant in the program is not held.

Data in a poisoned mutex may still be accessed, but one should be reminded that the data may be in an unexpected state.

THREAD SAFETY

Due to the nature of threads, some problems can arise while using them.

- Race conditions
- Deadlocks
- Unreproducible bugs

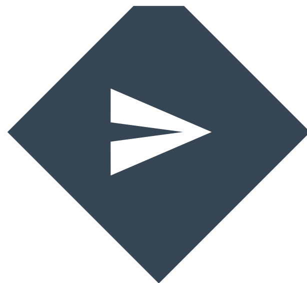
THREAD SAFETY

Rust subjects threads under the same ownership rules as data. This solves the most common problems regarding threading.

The following traits are applied to data that can be sent or received across threads

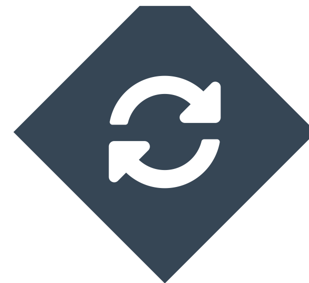
- `Send`
- `Sync`

SEND AND SYNC TRAITS



Send

- Ownership can be transferred across threads
- Almost all Rust data types have this trait

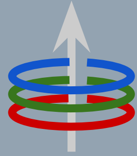


Sync

- Reference can be transferred across threads
- Multiple threads can safely reference the data

RESOURCES

- The Rust Book



CoE 164

Computing Platforms

12a: Rust Concurrency