CS3211 Tutorial 8+9

Safety and Concurrency in Rust

Asynchronous Programming in Rust

(AY 25/26 Semester 2)

April 9, 2026

(Compiled by Benson, thanks to all past and current TAs!)

Contents

Safety in Rust

Revisiting Concurrent Counter
- Threads and Scoped Threads
- `Mutex<T>`
- Reference counting with `Arc<T>`
- Atomics

Safe Data Parallelism with Rayon

Line echo server
- Threaded version
- `tokio` version: async/await

Revisiting H2O Problem

Welcome to the Rust World!

C++ (50%)

- Threads
- Synchronization
- Atomics
- Memory Ordering
- Debugging
- Lock-Free Programming

Go (25%)

- Lightweight Co-routines (Goroutines)
- Channels and Message Passing
- Etc

Rust (25%)

- Compile-time safety checking
- Safe futures / async
- Safe data parallelism
- Etc

The Guarantees of Rust

- Safe Rust $\Rightarrow$ No Undefined Behavior
  - No use-after-free, double-free.
  - Bounds checks, panics.
  - References are always valid and variables are initialized before use.
  - No data races.
- All of these are still possible!
  - Deadlock, livelock, etc.
  - Memory leaks.
- How to achieve this? Compiler performs (possibly too restrictive) checks!

Contents

Safety in Rust

Revisiting Concurrent Counter
- Threads and Scoped Threads
- `Mutex<T>`
- Reference counting with `Arc<T>`
- Atomics

Safe Data Parallelism with Rayon

Line echo server
- Threaded version
- `tokio` version: async/await

Revisiting H2O Problem

Ownership and Borrowing

```rust
fn f(s: String) {
  // similar to free(s) in c++
  drop(s)
}

fn main() {
  let s: String = String::from("Hello World!");
  f(s);
  print!("{}", s);
}
```

Rust Playground Link

`f(s)` **passes the ownership** of `s` to the function `f`.

Ownership and Borrowing

```rust
fn f(s: String) -> String {
  s
}

fn main() {
  let s: String = String::from("Hello World!");
  let t = f(s);
  print!("{}", t);
}
```

Rust Playground Link

Ownership can be passed back!

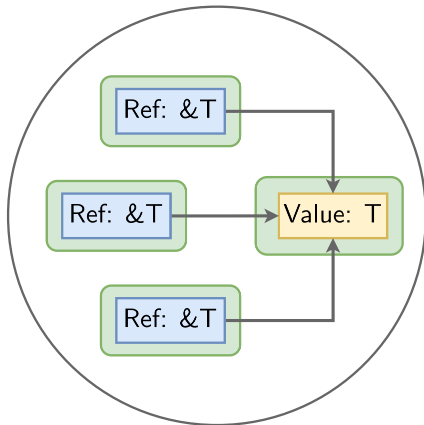
Ownership and Borrowing

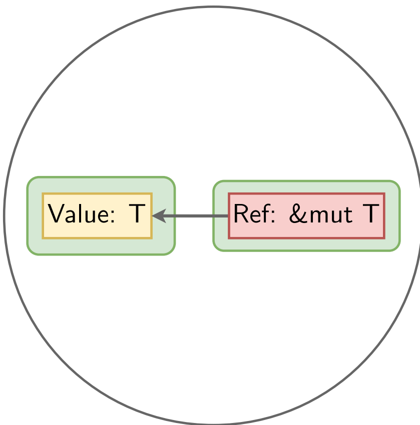
```
1  fn f(s: &String) {
2    println!("{}", s);
3  }
4
5  fn g(s: &mut String) {
6    *s = String::from("Bye!!!!");
7  }
8
9  fn main() {
10   let mut s: String = String::from("Hello World!");
11   f(&s);
12   g(&mut s);
13   f(&s);
14 }
```

Rust Playground Link

Note: References in Rust $\approx$ Pointers in C++.

Mutability or Sharing?





(Exclusivity guaranteed at compile time)

Lifetime

```
1 fn main() {
2   let mut x = 5;
3   let y = &mut x;
4   x = 6;
5   println!("{y}");
6 }
```

Rust Playground Link

Does this compile?

- **Lifetime** of `x`: The scope.
- **Lifetime** of the reference: Line 3 - 5.

Lifetime

```rust
fn longest(x: &str, y: &str) -> &str {
  if x.len() > y.len() {
    x
  } else {
    y
  }
}

fn main() {
  let string1 = "abcd";
  let result;
  {
    let string2 = "xyz";
    result = longest(string1, string2);
  }
  println!("{}", result);
}
```

Rust Playground Link

`string1` and `string2` have different lifetimes! What is the lifetime of the return value?

Lifetime

Lifetime annotations:

```
1  fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
2    if x.len() > y.len() {
3      x
4    } else {
5      y
6    }
7  }
```

```
1  fn longest_print_z<'a, 'b>(x: &'a str, y: &'a str, z: 'b str) -> &'a str {
2    println!("{}", z);
3    if x.len() > y.len() {
4      x
5    } else {
6      y
7    }
8  }
```

Checkpoint

Why do we need these rules in Rust?

1. There can only be one owner at a time: Prevents ____double free____.
2. There can only be either one mutable reference or any number of immutable references: Prevents ____data races____.
3. Every reference has a lifetime: Prevents ____use after free____.

Contents

Safety in Rust

Revisiting Concurrent Counter
- Threads and Scoped Threads
- `Mutex<T>`
- Reference counting with `Arc<T>`
- Atomics

Safe Data Parallelism with Rayon

Line echo server
- Threaded version
- `tokio` version: async/await

Revisiting H2O Problem

Concurrent Counter with Threads

```rust
use std::thread;

fn main() {
  let mut counter = 0;

  let t0 = thread::spawn(|| { counter += 1; });

  t0.join();

  println!("{}", counter);
}
```

Rust Playground Link

Issues:

1. `t0` might outlive the `main` thread (use-after-free).
2. There could be a data race between the threads `main` and `t0`.

Concurrent Counter with Scoped Threads

```
1   use std::thread;
2
3   fn main() {
4     let mut counter = 0;
5
6     thread::scope(|s| {
7       s.spawn(|| { counter += 1; });
8     });
9
10    println!("{}", counter);
11  }
```

Rust Playground Link

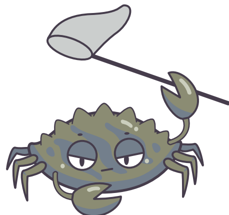
Exercise: Can you add one more thread?

Concurrent Counter with Scoped Threads

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

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

  thread::scope(|s| {
    s.spawn(|| { *counter.lock().unwrap() += 1; });
    s.spawn(|| { *counter.lock().unwrap() += 1; });
  });

  println!("{}", *counter.lock().unwrap());
}
```

Rust Playground Link

Concurrent Counter with Scoped Threads



(A thread panics while holding the lock ⇒ "poisoned")

`Result<MutexGuard<'_, {integer}>, PoisonError<...>>`

`*counter.lock().unwrap()` ⇒ `{integer}`

`std::sync::Mutex<{integer}>`

Interior Mutability

Why can we mutate the inner `T` with just `&Mutex<T>`?

```
pub struct Mutex<T: ?Sized> {
  inner: sys::Mutex,     // mutex
  poison: poison::Flag, // panic
  data: UnsafeCell<T>,  // value
}
pub struct UnsafeCell<T: ?Sized> {
  value: T,
}
```

```
impl<T: ?Sized> Deref for MutexGuard<'_, T> {
  type target = T;

  fn deref(&self) -> &T {
    unsafe { &*self.lock.data.get() }
  }
}
```

```
pub const fn get(&self) -> *mut T {
  self as *const UnsafeCell<T>
       as *const T
       as *mut T
}
```

The compiler disables some checks, but the library is provably correct.

Reference counting with Arc<T>

Arc: "Atomically Reference Counted" (aka shared_ptr in C++)

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

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

  let t0 = {
    let counter = counter.clone();
    thread::spawn(move || {
      *counter.lock().unwrap() += 1;
    })
  };
  // do the same for t1

  t0.join.unwrap();

  println!("{}", *counter.lock().unwrap());
}
```

Rust Playground Link

Atomics

```rust
use std::sync::atomic::{AtomicI32, Ordering};
use std::thread;

fn main() {
  let counter = AtomicI32::new(0);

  thread::scope(|s| {
    s.spawn(|| {
      counter.fetch_add(1, Ordering::Relaxed);
    });
    s.spawn(|| {
      counter.fetch_add(1, Ordering::Relaxed);
    });
  });

  println!("{}", counter.load(Ordering::Relaxed));
}
```

Rust Playground Link

Interesting Complexities with Scopes

Extra Slide

This program does not compile... try to find out the problem and fix it.

```
use std::thread;

fn main() {
  let arr = vec![String::new(); 10];
  thread::scope(|s| {
    for i in 0..10 {
      s.spawn(|| println!("{}", &arr[i]));
    }
  })
}
```

Rust Playground Link

Interesting Complexities with Scopes

Extra Slide

Solution.

1. The thread needs the value `i` but the for loop might end earlier!
   - We must move `i` into the thread.
   - Put a `move` for `s.spawn`.
2. We run into another compile error – we cannot move `arr` into the thread!
   - Instead of moving `arr`, move (i.e. borrow) a reference to `arr` instead.

```
use std::thread;

fn main() {
  let arr = vec![String::new(); 10];
  thread::scope(|s| {
    let borrowed_arr = &arr;
    for i in 0..10 {
      s.spawn(move || println!("{}", borrowed_arr[i]));
    }
  })
}
```

Contents

Safety in Rust

Revisiting Concurrent Counter
- Threads and Scoped Threads
- `Mutex<T>`
- Reference counting with `Arc<T>`
- Atomics

Safe Data Parallelism with Rayon

Line echo server
- Threaded version
- `tokio` version: async/await

Revisiting H2O Problem

Safe Data Parallelism with Rayon

```
1  fn magic_sum(from: u128, to: u128) -> u128 {
2    (from..to).filter(|i| i % 7 == i % 5).sum()
3  }
4
5  fn main() {
6    println!("{}", magic_sum(0, 100000000));
7  }
```

Rust Playground Link

This program looks trivially parallelizable...

Parallelizing with Rayon:

```
1  use rayon::prelude::*;
2
3  fn magic_sum(from: u128, to: u128) -> u128 {
4    (from..to).into_par_iter().filter(i i % 7 == i % 5).sum()
5  }
```

Contents

Safety in Rust

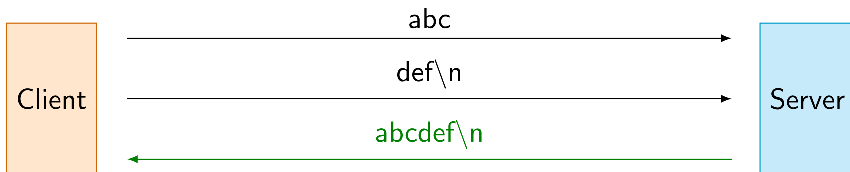
Revisiting Concurrent Counter
- Threads and Scoped Threads
- `Mutex<T>`
- Reference counting with `Arc<T>`
- Atomics

Safe Data Parallelism with Rayon

Line echo server
- Threaded version
- `tokio` version: async/await

Revisiting H2O Problem

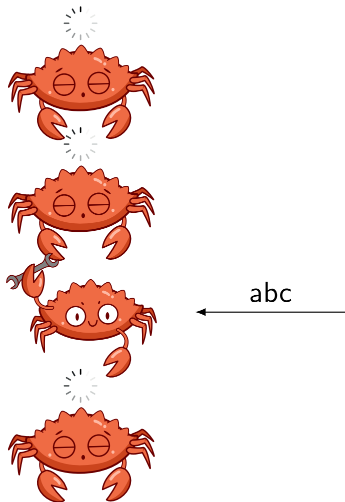
Line echo server



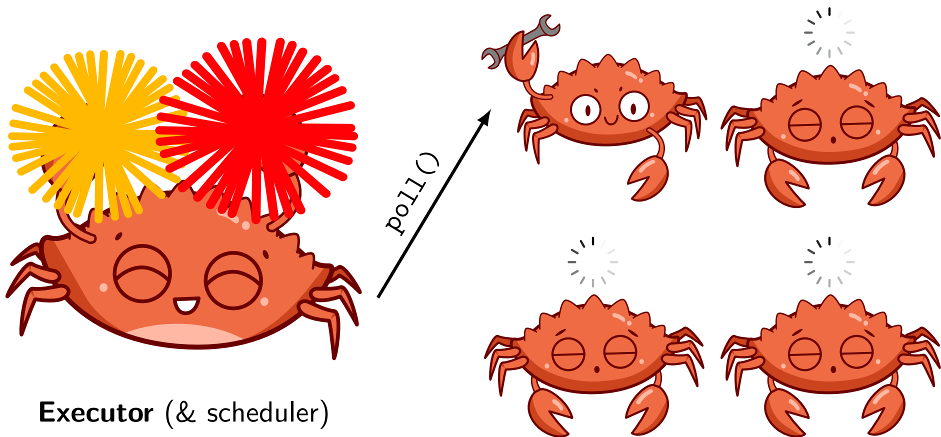
Threaded version

```rust
fn main() -> std::io::Result<()> {
  let port = std::env::args()
    .nth(1)
    .map(|s| s.parse().unwrap())
    .unwrap_or(50000u16);
  let listener = TcpListener::bind(
     SocketAddr::from(([127, 0, 0, 1], port)))?;
  loop {
    let (socket, _) = listener.accept()?;
    thread::spawn(move || {
      eprintln!("Accepted connection");
      std::mem::drop(handle_client(socket));
      eprintln!("Connection ended");
    });
  }
}
```

Issue. We're wasting resources on threads that are **mostly just sleeping**!



Async Functions – Tokio Runtime



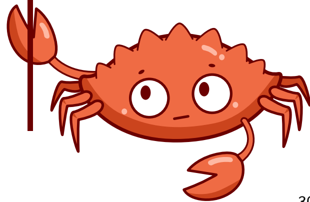
Writing Async Functions

```
1  async fn add_to_inbox(...) -> Result<(), Error> {
2    let msg = load_message(email)?;
3    ...
4  }
```

Async functions return a **Future**!

await: Wait for the future and get its value.

```
1  async fn add_to_inbox(...) -> Result<(), Error> {
2    let msg = load_message(email).await?;
3    ...
4  }
```



tokio version of Line echo server

```
1  fn handle_client(stream: TcpStream)
2      -> std::io::Result<()> {
3    let mut reader = BufReader::new(stream);
4    let mut buf: Vec<u8> = Vec::new();
5    loop {
6      let size = reader.read_until(b'\n', &mut buf)?;
7      if size == 0 || buf[size - 1] != b'\n' {
8        break;
9      }
10     reader.get_mut().write_all(&buf[..size])?;
11     buf.clear();
12   }
13   Ok(())
14 }
```

Rust Playground Link

Let's make the handler async!

tokio version of Line echo server

```
1  #[tokio::main]
2  async fn main() -> std::io::Result<()> {
3    let port = std::env::args()
4      .nth(1)
5      .map(|s| s.parse().unwrap())
6      .unwrap_or(50000u16);
7    let listener = TcpListener::bind(
8      SocketAddr::from(([127, 0, 0, 1], port))).await?;
9    loop {
10     let (socket, _) = listener.accept().await?;
11     handle_client(socket).await?;
12   }
13 }
```

Main program: Do you see the issue?

tokio version of Line echo server

```
1   #[tokio::main]
2   async fn main() -> std::io::Result<()> {
3     let port = std::env::args()
4       .nth(1)
5       .map(|s| s.parse().unwrap())
6       .unwrap_or(50000u16);
7     let listener = TcpListener::bind(
8         SocketAddr::from(([127, 0, 0, 1], port))).await?;
9     loop {
10      let (socket, _) = listener.accept().await?;
11      tokio::spawn(async move {
12        std::mem::drop(handle_client(socket).await);
13      });
14    }
15  }
```

Now it's correct!

`tokio` version of Line echo server

We're done!

Async codes are **not too different** from sync codes!

- The compiler handles much of the heavy lifting on our behalf.

Async Functions – State Machine

Extra Slide

```
1 async fn add_to_inbox(...) -> Result<(), Error> {
2   let msg = load_message(email).await?;
3   let user = get_user(id).await?;
4   user.verify_has_space(&msg)?;
5   user.add_to_inbox(msg).await
6 }
```

How to allow context switching?

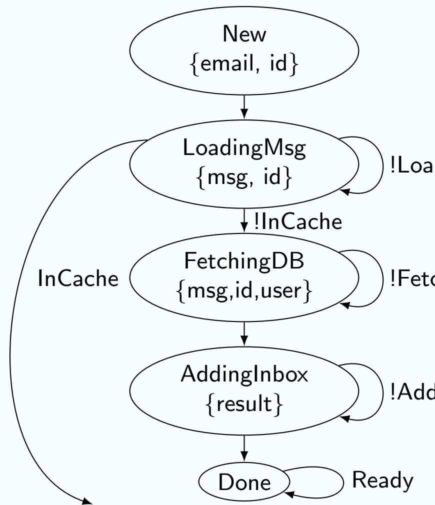
- Too slow to copy registers, program counter, stack pointer, etc.

Async Functions – State Machine

Extra Slide

```
1  async fn add_to_inbox(...) -> Result<(), Error> {
2    let msg = load_message(email).await?;
3    let user = get_user(id).await?;
4    user.verify_has_space(&msg)?;
5    user.add_to_inbox(msg).await
6  }
```

```
1  async fn get_user(id: u64) -> u64 {
2    if let Some(user) = cache.get(&id) {
3      return *user;
4    }
5    let user = fetch_db(id).await;
6    cache.put(id, user);
7    user
8  }
```

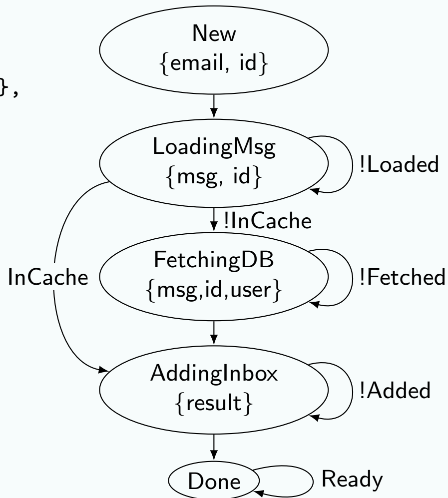


Async Functions – State Machine

Extra Slide

```rust
enum AddToInboxState {
  New          {email: String, id: u64},
  LoadingMsg  {msg: Option<String>, id: u64},
  FetchingDB  {msg: ..., id: u64, user: Option<u64>},
  AddingInbox {result: Option<Result>},
}
```

```rust
fn poll(..., cx) -> Poll<Self::Output> {
  match self.state {
    ...
    AddingInbox => {
      if let Some(res) = self.result {
        cx.wake(); // wake my caller!
        Poll::Ready(res);
      } else {
        Poll::Pending
      }
    }
  }
}
```



Contents

Safety in Rust

Revisiting Concurrent Counter

- Threads and Scoped Threads
- `Mutex<T>`
- Reference counting with `Arc<T>`
- Atomics

Safe Data Parallelism with Rayon

Line echo server

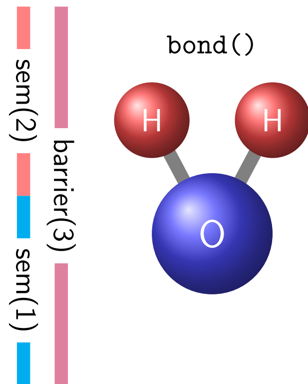
- Threaded version
- `tokio` version: async/await

Revisiting H2O Problem

Recap: H2O Problem

```cpp
std::counting_semaphore<> oxygenSem{1};
std::counting_semaphore<> hydrogenSem{2};
std::barrier<> barrier{3};

void oxygen(void (*bond)()) {
  oxygenSem.acquire();
  barrier.arrive_and_wait();
  bond();
  oxygenSem.release();
}

void hydrogen(void (*bond)()) {
  hydrogenSem.acquire();
  barrier.arrive_and_wait();
  bond();
  hydrogenSem.release();
}
```

Link to godbolt



Rusty H2O: Implementation

```
1  use tokio::sync::{Barrier, Semaphore};
2  struct WaterFactory {
3    o_sem: Semaphore,
4    h_sem: Semaphore,
5    barrier: Barrier,
6  }
7
8  fn oxygen(&self, bond: impl FnOnce()) {
9    let _permit = self.o_sem.acquire(); // RAII
10   self.barrier.wait();
11   bond();
12 }
13
14 // inside main
15 let f = Arc::new(WaterFactory::new());
16 let hs = (0..n * 2).map(|i| {
17   let f = f.clone();
18   tokio::spawn(async move {
19     f.hydrogen(|| bond(&format!("h{i}"))).await;
20   })
21 });
```

Rust Playground Link

Summary

- (Safe) Rust comes with many useful guarantees.
  - No undefined behaviour, memory safety, type safety...

- Async-await is a great way to have many minimal-overhead tasks that exploit maximum concurrency.
  - Rust efficiently compiles async functions into state machines.
  - Tokio executes these async functions as tasks.

- Good News: ~~Assignment 3 (coding part) should be doable in $<$ 5 mins if you really understood this tutorial!~~ No longer the case :((

Attendance

Session ID: 1940161
CS3211 Tutorial 9
9 April 2026 14:00-16:00
BIZ2-02-02 - SEMINAR ROOM 2-2

That's it!

Anonymous Feedback (throughout the semester):

https://forms.gle/6a9T4t88wNwYxG4G8

The link to the tutorial slides will be posted in telegram.