PURDUE UNIVERSITY

Principles of Concurrency

Lecture 14
Rust

Motivation

- Fundamental tension in concurrency between safety and performance/control.
  - Safety: statically identify bugs; data encapsulation
  - Performance/control: manipulate low-level representation without penalty

- Java, Haskell, OCaml, etc. give strong safety guarantees at the expense of control
- C/C++ give control but at the expense of safety

Rust

- Industry-supported language designed to overcome this tension
- Main idea: use a strong type-system to prohibit important kinds of unwanted behavior, namely those involving mutation of shared state.
  - Approach allows many kinds of systems programming errors (e.g, data races, use-after-free) to be detected statically
  - For data structures that cannot be checked in this way, Rust allows them to encapsulated as "unsafe" within otherwise safe APIs

Sustainability

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

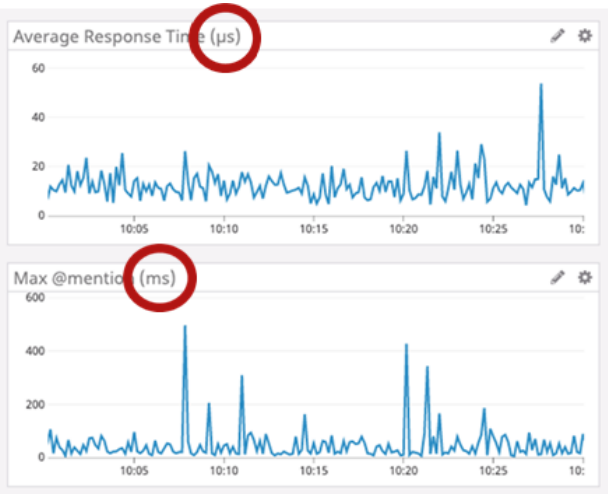
	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

Low energy footprint

with low latencies and faster response times

Go

Rust



Discord server

The Problem Being Addressed


```
gift := Gift { .. }
channel <- gift;
gift.open(); // oops
```

```
// The other goroutine
gift := <- channel
gift.open()
```

In Go

```
let gift = Gift::new();
channel.send(gift);

// ERROR: gift was already moved
gift.open();
```

```
let gift = channel.recv();
gift.open();
```

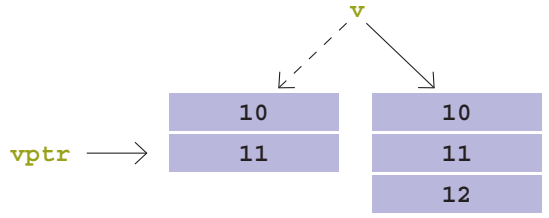
In Rust

Data Race

Use-after-free

```
1  std::vector<int> v { 10, 11 };
2  int *vptr = &v[1]; // Points *into* v.
3  v.push_back(12);
4  std::cout << *vptr; // Bug (use-after-free)
```

C++




```
1  let mut v = vec![10, 11];
2  let vptr = &mut v[1]; // Points *into* 'v'.
3  v.push(12);
4  println!("{}", *vptr); // Compiler error
```

Rust

Key Innovations: Ownership and Borrowing


```
1  fn consume(w: Vec<i32>) {
2    drop(w); // deallocate vector
3  }
4  let v = vec![10, 11];
5  consume(v);
6  v.push(12); // Compiler error
```

Could have been automatically inserted by the compiler

Ownership

pass-by-value

track lifetimes

```
1  fn add_something(v: &mut Vec<i32>) {
2    v.push(11);
3  }
4  let mut v = vec![10];
5  add_something(&mut v);
6  v.push(12); // Ok!
7  // v.push(12) is syntactic sugar for Vec::push(&mut v, 12)
```

Borrowing

pass-by-reference

Ownership

- A variable binding takes ownership of its data. [lifetimes]
  A piece of data can only have one owner at a time.
- When a binding goes out of scope, the bound data is released automatically.
  For heap-allocated data, this means de-allocation.
- Data must be guaranteed to outlive its references.

```
fn foo() {
    // Creates a Vec object.
    // Gives ownership of the Vec object to v1.
    let mut v1 = vec![1, 2, 3];
    v1.pop();
    v1.push(4);
    // At the end of the scope, v1 goes out of scope.
    // v1 still owns the Vec object, so it can be cleaned up.
}
```

CIS 198: Rust Programming, University of Pennsylvania, Spring 2016,
`http://cis198-2016s.github.io/slides/01/`

Move Semantics


```
let v1 = vec![1, 2, 3];
let v2 = v1;   // Ownership of the Vec object moves to v2.
println!("", v1[2]); // error: use of moved value ‘v1‘
```

- `let v2 = v1;`
  - We don't want to copy the data, since that's expensive.
  - The data cannot have multiple owners.
  - **Solution:** move the Vec's ownership into v2, and declare v1 invalid.
- `println!("{}", v1[2]);`
  - We know that v1 is no longer a valid variable binding, ∴ error!
  - Rust can reason about this at compile time, ∴ compiler error.
  - Moving ownership is a compile-time semantic.
    It doesn't involve moving data during your program.

- Rust would be a pain to write if we were forced to explicitly move ownership back and forth.

```
fn vector_length(v: Vec<i32>) -> Vec<i32> {
    // Do whatever here,
    // then return ownership of ‘v‘ back to the caller
}
```

- The more variables you had to hand back (think 5+), the longer your return type would be!

CIS 198: Rust Programming, University of Pennsylvania, Spring 2016,
`http://cis198-2016s.github.io/slides/01/`

Borrowing

- In place of transferring ownership, we can borrow data.
- A variable's data can be borrowed by taking a reference to the variable (i.e., aliasing); ownership doesn't change.
- When a reference goes out of scope, the borrow is over.
- The original variable retains ownership throughout.

```
let v = vec![1, 2, 3];
let v_ref = &v;  // v_ref is a reference to v.
assert_eq!(v[1], v_ref[1]); // use v_ref to access the data
                            // in the vector v.
// BUT!
let v_new = v;  // Error, cannot transfer ownership
                // while references exist to it.
```

CIS 198: Rust Programming, University of Pennsylvania, Spring 2016,
http://cis198-2016s.github.io/slides/01/

Borrowing


```
// ‘length‘ only needs ‘vector‘ temporarily, so it is borrowed.
fn length(vec_ref: &Vec<i32>) -> usize {
    // vec_ref is auto-dereferenced when you call methods on it.
    vec_ref.len()
}
fn main() {
    let vector = vec![];
    length(&vector);
    println!("{:?}", vector); // this is fine
}
```

- References, like bindings, are immutable by default.
- The borrow is over after the reference goes out of scope (at the end of `length`).
- (`usize` =The pointer-sized unsigned integer type.)

CIS 198: Rust Programming, University of Pennsylvania, Spring 2016,
`http://cis198-2016s.github.io/slides/01/`

Borrowing


```
// ‘push‘ needs to modify ‘vector‘ so it is borrowed mutably.
fn push(vec_ref: &mut Vec<i32>, x: i32) {
    vec_ref.push(x);
}
fn main() {
    let mut vector: Vec<i32> = vec![];
    let vector_ref: &mut Vec<i32> = &mut vector;
    push(vector_ref, 4);
}
```

- Variables can be borrowed by mutable reference: `&mut vec_ref`.
  - `vec_ref` is a reference to a mutable `Vec`.
  - The type is `&mut Vec<i32>`, not `&Vec<i32>`.
- Different from a reference which is variable.
- You can have exactly one mutable borrow at a time.
  Also you cannot dereference borrows (changes ownership).

Summary

1. You can’t keep borrowing something after it stops existing.
2. One object may have many immutable references to it `(&T)`.
3. **OR** exactly one mutable reference `(&mut T)` (not both).

CIS 198: Rust Programming, University of Pennsylvania, Spring 2016,
`http://cis198-2016s.github.io/slides/01/`

Consequences

Valid in C, C++,...

```
let y: &i32;
{
    let x = 5;
    y = &x; // error: ‘x‘ does not live long enough
}
println!("{}", *y);
```

This eliminates vast numbers of memory safety bugs *at compile time!*

Under the standard ownership rules:

- You **cannot** implement doubly-linked lists (and circular structures in general).
- You **cannot** call C libraries.
- ...

Unsafe Rust

```
unsafe {
   ...
}
```

- Relaxes some of the checking rules.
- Allows C libraries calls.
- Allows access to raw pointers.
- Allows you to implement language extensions, e.g., doubly-linked lists, garbage-collected pointers, etc.

CIS 198: Rust Programming, University of Pennsylvania, Spring 2016, `http://cis198-2016s.github.io/slides/01/`

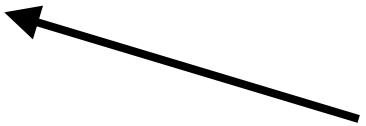
Concurrency


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

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap();
}
```



Synchronization point: main thread waits for child thread to complete
What happens if the join moves above the for loop?

Concurrency


```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Why doesn't this compile?

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector:
{:?}", v);
    });

    drop(v); // oh no!

    handle.join().unwrap();
}
```

Reference to v in spawned thread would no longer be valid

Concurrency

Rust's ownership rules require (among other things):

- there may exist at most one mutable reference to a value
- there may be any number of immutable references to a value

These two rules enforce a multiple-reader, single-writer discipline

```
int main() {
  std::string msg = "Hello";
  std::thread t1([&]() {
    std::cout << msg << std::endl;
  });
   msg += ",world";
   t1.join();
   return 0;
}
```

```
fn main() {
  let mut msg = "Hello".to_string();
  let handle = thread::spawn(|| {
       println!("{}, &msg);
  });
  msg.push_str(",world");
  handle.join().unwrap();
}
```

Rust generates two errors:

- cannot borrow 'msg' as mutable; why?
- msg does not live long enough? why?

To fix these errors - modify msg and then invoke the thread

Concurrency


```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Move gives ownership to the spawned thread; the parent no longer has access to v

Shared Memory


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

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    println!("m = {:?}", m);
}
```

Unlock automatically is injected once mut goes out of scope; the value protected by the mutex (num) is a mutable reference

Why does this not compile?

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

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

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

Sharing and Concurrency


```
1  let v = vec![10,11];
2  let vptr = &v[1];
3  join( || println!("v[1] = {}", *vptr),
4        || println!("v[1] = {}", *vptr));
5  v.push(12);
```

What happens if `v.push(12)` is moved into one of the threads?

```
1  let mutex_v = Mutex::new(vec![10, 11]);
2  join(
3    || { let mut v = mutex_v.lock().unwrap();
4         v.push(12); },
5    || { let v = mutex_v.lock().unwrap();
6         println!("{:?}", *v) });
```

`&'a mut T`:

lock's type is: `fn(&'a Mutex<T>) -> MutexGuard <'a, T>`

lock can be called with a shared reference to mutex - allowing lock to be invoked by multiple threads. What would happen if lock took a mutable reference?

Shared Memory

Atomic reference counting smart pointer

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

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

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