CS 344: Operating Systems I

03.13: Part IV – Rust

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab

Announcement

- Upcoming deadlines
  - 3/15: Programming assignment V
  - 3/17: Extra credit opportunity (Linus Torvalds article)
  - 3/20: Midterm quiz IV
  - 3/20: The other three extra credit opportunities
  - 3/22: Late submissions for programming assignments only

Topics for today

- Rust
  - Motivation
    - Problem: control vs. safety
    - Solution: Rust
  - Core concepts
    - Ownership and borrowing
    - Concurrency
    - Unsafe code
  - Benefits
    - No need for a runtime
    - Memory safety
    - Data-race freedom
  - Example practice
    - Multi-threaded map-reduce

Motivation

- **Popular(?) programming languages**
  - C
  - C++
  - Java
  - JavaScript (JS)
  - Python
  - Go
  - Perl
  - Scala
  - Lua
  - …

Motivation: A trade off between control and safety


```c
…
#define  BUFSIZE          20

int main(void) {
   char *buf;
   char *str = "Hello world!";

   // initialize the memory space
   buf = (char *) malloc( sizeof(char) * BUFSIZE );

   // copy the string to the buffer
   strncpy(buf, str, BUFSIZE);

   // print the string
   printf("Buffer contains: %s.\n", buf);

   return 0;
}
```

```python
…import

if __main__ == "__main__":
   buf = ""
   str = "Hello world!"

   // copy the string
   buf += str

   // print out it
   print ("{}".format(buf))
   # done.
```

Example:

- **C:** More control over mem. allocation, but less safe
- **Python:** Less control, but more safe

Motivation: a trade off between control and safety

- **Example: C has more control, but care must be taken**

```
…
#define  BUFSIZE            20

int main(void) {
   char *buf;
   char *str = "Hello world!";

   // initialize the memory space
   buf = (char *) malloc( sizeof(char) * BUFSIZE );

   // copy the string to the buffer
   strncpy(buf, str, BUFSIZE);

   // free the buffer
   free(buf);

   // print the string
   printf("Buffer contains: %s.\n", buf);

   return 0;
}
```

- Allocate 20 bytes
- "buf" points the first char of "Hello world!"
- "buf" points "NULL"
- "buf" is used in the printf statement (Note: **use-after-free** vulnerability – link)

C (example):
- We can control the memory allocations
- We must be careful when we allocate (safety)

Example scenario
- Programs run on the OS for satellites
- Programs run on the NASA's Curiosity

Motivation: a trade off between control and safety

- **Example: Python doesn't need mem. control, but often less efficient**

```
...import

if __main__ == "__main__":
    buf = ""
    str = "Hello world!"

    // copy the string
    buf += str

    // nullify the string
    str = ""

    // print out it
    print ("{}".format(buf))
    # done.
```

- Python interpreter allocates 20 bytes
- The interpreter allocates 20 bytes
- "str" releases the string, but we **do not know** if the mem is de-allocated after this
- "buf" is used in the print statement

Python (example):

- We cannot control the memory allocations
- We do not need to care the mem. de-allocations [Garbage collector (GC) will do this management, but it requires ++computations and ++memory]

Example scenario

- Programs run on your laptop
- Programs run on the clusters (or in the cloud)

A solution: Rust!

- Rust
  - A programming language designed for (memory) safety and performance
  - Try this example (link)!
    - Write a Rust program (hello.rs)
    - Compile and run the program (rustc hello.rs)

- Rust addresses
  - Runtime performance (unlike Python or Java, Rust does not use GC)
  - Memory leaks (no explicit allocation/de-allocation)
  - No data-race condition

Rust example: hello world

- **Hello-world**

```
fn main() {
   println! ("Hello world! ");
}
```

Rust type: we can explicitly/implicitly set a variable type

- **Hello-world**

```
fn main() {
   println! ("Hello world! ");
}
```

- **Types supported**

```
fn main() {
   let logical: bool = true;
   let a_float: f64 = 1.0;
   let default_float = 3.0;              // f64
   let default_integer = 7;              // i32
   let default_unsigned64: usize = 100;  // u64

   let mut inferred_type = 12;
   inferred_type = 4294967296i64;

   let mut mutable = 12; mutable = 21;
   mutable = true;

   let mutable = true;
}
```

Initialize variables:

- Line 1: we can set it to “bool”
- Line 2: we can set it to “f64” (64-bit float: double)
- Line 3: it can automatically define it to “f64” (3.0)
- Line 4: it can automatically define it to “i32” (7)
- Line 5: we can use “usize” to define “u64” (64-bit)

Rust type: fixed variables and mutable variables

- **Hello-world**

```
fn main() {
    println! ("Hello world! ");
}
```

- **Types supported**

```
fn main() {
    let logical: bool = true;
    let a_float: f64 = 1.0;
    let default_float = 3.0;                // f64
    let default_integer = 7;                // i32
    let default_unsigned64: usize = 100;    // u64

    let mut inferred_type = 12;
    inferred_type = 4294967296i64;

    let mut mutable = 12; mutable = 21;
    mutable = true;

    let mutable = true;
}
```

Initialize variables:

- Line 1: we can set it to "bool"
- Line 2: we can set it to "f64" (64-bit float: double)
- Line 3: it can automatically define it to "f64" (3.0)
- Line 4: it can automatically define it to "i32" (7)
- Line 5: we can use "usize" to define "u64" (64-bit)

Variable types can be inferred from context:

- Line 1: we can set the var. to a **mutable** (mut)
- Line 2: it will automatically set the var to "i64"

Rust type: fixed variables and mutable variables – cont'd

- **Hello-world**

```
fn main() {
    println! ("Hello world! ");
}
```

- **Types supported**

```
fn main() {
    let logical: bool = true;
    let a_float: f64 = 1.0;
    let default_float = 3.0;                // f64
    let default_integer = 7;                // i32
    let default_unsigned64: usize = 100;    // u64

    let mut inferred_type = 12;
    inferred_type = 4294967296i64;

    let mut mutable = 12; mutable = 21;
    mutable = true;

    let mutable = true;
}
```

Initialize variables:

- Line 1: we can set it to "bool"
- Line 2: we can set it to "f64" (64-bit float: double)
- Line 3: it can automatically define it to "f64" (3.0)
- Line 4: it can automatically define it to "i32" (7)
- Line 5: we can use "usize" to define "u64" (64-bit)

Variable types can be inferred from context:

- Line 1: we can set the var. to a **mutable** (mut)
- Line 2: it will automatically set the var to "i64"

Mutable variables:

- Line 1: we can update the value of the mutable var.
- Line 2: but we cannot change the type of it

Rust type: variable shadowing

- **Hello-world**

```
fn main() {
    println! ("Hello world! ");
}
```

- **Types supported**

```
fn main() {
    let logical: bool = true;
    let a_float: f64 = 1.0;
    let default_float = 3.0;                // f64
    let default_integer = 7;                // i32
    let default_unsigned64: usize = 100;    // u64

    let mut inferred_type = 12;
    inferred_type = 4294967296i64;

    let mut mutable = 12; mutable = 21;
    mutable = true;

    let mutable = true;
}
```

Initialize variables:
- Line 1: we can set it to "bool"
- Line 2: we can set it to "f64" (64-bit float: double)
- Line 3: it can automatically define it to "f64" (3.0)
- Line 4: it can automatically define it to "i32" (7)
- Line 5: we can use "usize" to define "u64" (64-bit)

Variable types can be inferred from context:
- Line 1: we can set the var. to a **mutable** (mut)
- Line 2: it will automatically set the var to "i64"

Mutable variables:
- Line 1: we can update the value of the mutable var.
- Line 2: but we cannot change the type of it

Shadowing:
- Line 1: we can override the variable (variable shadowing: link)

Rust example: array, indexing, for-loop, and if statements

- **Example I**

```
fn main() {
    let xs: [i32; 5] = [1, 2, 3, 4, 5];
    let ys: [i32; 10] = [0; 10];

    println! ("The first element: {}", xs[0]);
    println! ("Elements from the first to the fourth: {}", xs[0 .. 3]);
}
```

Initialize arrays:
- Line 1: we can create an array "i32"; the len is 5
- Line 2: we can initialize with all 0s

Indexing:
- Line 1: we can access an element by the index
- Line 2: we can access multiple elements

Rust example: array, indexing, for-loop, and if statements

- **Example I**

```
fn main() {
    let xs: [i32; 5] = [1, 2, 3, 4, 5];
    let ys: [i32; 10] = [0; 10];

    println! ("The first element: {}", xs[0]);
    println! ("Elements from the first to the fourth: {}", xs[0 .. 3]);
}
```

Initialize arrays:
- Line 1: we can create an array "i32"; the len is 5
- Line 2: we can initialize with all 0s

Indexing:
- Line 1: we can access an element by the index
- Line 2: we can access multiple elements

- **Example II**

```
fn main() {

    for n in 1…101 {
        if n < 10 && n % 5 == 0 {
            println!("The number smaller than 10 and divisible by 5: {}", n);
        } else {
            println!("The number is {}", n);
        }
    }

    println!("The final number will be {}", n);
}
```

For loop:
- Line 1: it iterates from 1 to 100 (*i.e.*, 101 – 1)
  (alternative: for n in 1..=100)

If … else:
- Line 1: we can use && for the "and" condition
  ("or" is || / "not" is ! / "not eq" is !=)

Rust example: function

- **Function calls**

```
fn compute(x: u32, y: u32) -> u32 {
    if x == 0 {
        return 0;
    }

    let z = x.pow(y);
    z
}

fn main() {
    let val;

    val = compute(3, 4);
    println! ("Result: {}", val);
}
```

Rust function:
- Line 1: we receive two arguments x, y
  (both x, y are "u32" and returns "u32")
- Line 2: if "x == 0" then return 0
  (we need "return" if we exit the fn early)
- Line 3: compute x^y and store it to z
- Line 4: return z
  (no explicit return statement is required)

Rust function "call":
- Line 1: create "val" variable
- Line 2: call the "compute" function with 3 and 4
- Line 3: store the result to "val"
  (Note: won't work if we "let val = 0;" in Line 1)

Topics for today

- Rust
  - Motivation
    - Problem: control vs. safety
    - Solution: Rust
  - Core concepts
    - Ownership and borrowing
    - Concurrency
    - Unsafe code
  - Benefits
    - No need for a runtime
    - Memory safety
    - Data-race freedom
  - Example practice
    - Multi-threaded map-reduce

Rust core concepts

- Core concepts
  - Ownership and borrowing
  - Concurrency
  - Unsafe code

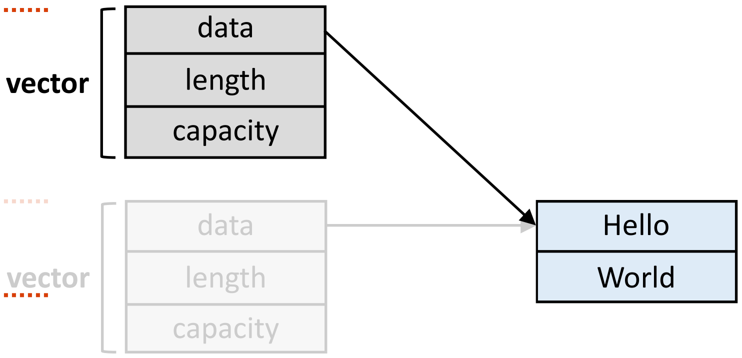
Rust ownership

- Ownership
  - **Definition:** a set of rules how a Rust program manages memory
  - Rust rules:
    - Each value in Rust has a variable “owner”
    - There can be only one owner at a time
    - If the owner goes out of scope, the value will disappear
  - Ownership example:

```
fn take(vec: Vec<String>){
    println!("{:?}", vec);
}

fn main() {
    let mut vec = Vec::new();
    vec.push(String::from("Hello "));
    vec.push(String::from("World "));
    take(vec);

    vec.push(String::from("from the other side!"))
}
```



Rust ownership

- Ownership
  - **Definition:** a set of rules how a Rust program manages memory
  - Rust rules:
    - Each value in Rust has a variable “owner”
    - There can be only one owner at a time
    - If the owner goes out of scope, the value will disappear
  - Ownership example:

```
fn take(vec: Vec<String>){
    println!("{:?}", vec);
}

fn main() {
    let mut vec = Vec::new();
    vec.push(String::from("Hello "));
    vec.push(String::from("World "));
    take(vec);

    vec.push(String::from("from the other side!"))
}
```

But Sometimes, We Need “vec” again in main!

Note:
The last line will cause an error! No “vec”
Ownership is *forced* by the Rust compiler

It prevents:
Use-after-free vulnerability
(dangling pointers)

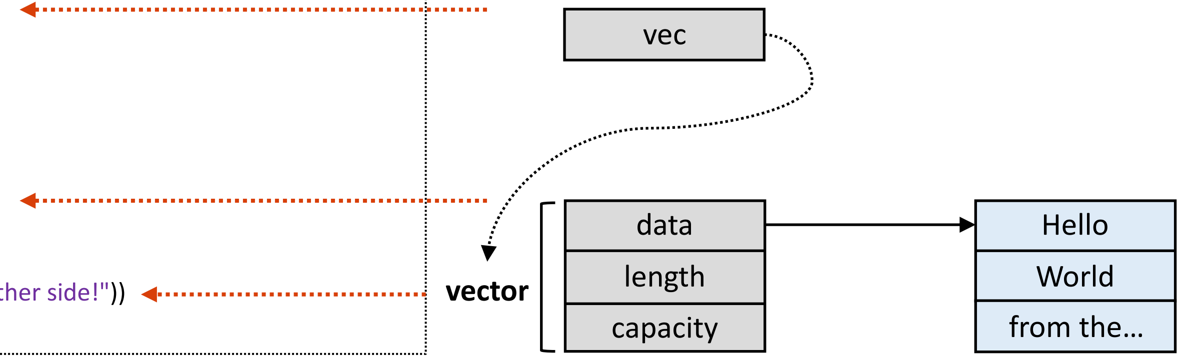
Rust borrowing

- Borrowing
  - **Definition:** a way to access data without taking ownership over it
  - Borrowing example:

```
fn borrow(vec: &Vec<String>){
    println!("{:?}", vec);
}

fn main() {
    let mut vec = Vec::new();
    vec.push(String::from("Hello "));
    vec.push(String::from("World "));
    borrow(&vec);

    vec.push(String::from("from the other side!"))
}
```



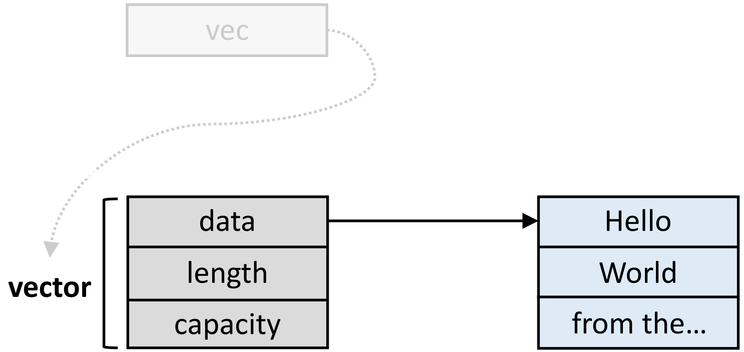
Rust borrowing

- Borrowing
  - **Definition:** a way to access data without taking ownership over it
  - Borrowing example:

```
fn borrow(vec: &Vec<String>){
    println!("{:?}", vec);
}

fn main() {
    let mut vec = Vec::new();
    vec.push(String::from("Hello "));
    vec.push(String::from("World "));
    borrow(&vec);

    vec.push(String::from("from the other side!"))
}
```



But "vec" Is Immutable in "borrow"!

Note:
The "borrow" fn uses a shared reference "vec"
The "vec" disappears if the function ends
The "vec" in main still is alive

Rust concurrency

- Concurrency
  - Shared **read-only** accesses
  - Concurrency example:

Deposit thread:
- Line 1: read the balance and make it mutable
- Line 2: increase the balance by 100
- Line 3: print out the balance

Withdrawal thread:
- Line 1: read the balance and make it mutable
- Line 2: decrease the balance by 300
- Line 3: print out the balance

Thread join:
- Line 1: wait for the threads to join
- Line 2: print out the balance value

```
use std::thread;

fn main() {
    let mut balance = 200;
    let mut threads = vec![];

    // deposit thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance += 100;
        println!("Increase the balance {}", new_balance);
    }));

    // withdrawal thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance -= 300;
        println!("Decrease the balance {}", new_balance);
    }));

    for thread in threads {
        let _ = thread.join();
    }
    println!("Final balance {}", balance);
}
```

Rust concurrency

- Concurrency
  - Shared **read-only** accesses
  - Concurrency example:

Results:

```
$ ./main
Decrease the balance -100
Increase the balance 300
Final balance 200
```

Note:

"balance" is a read-only shared variable
"new_balance" only exists in each thread
No effect on the actual "balance" in main

```rust
use std::thread;

fn main() {
    let mut balance = 200;
    let mut threads = vec![];

    // deposit thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance += 100;
        println!("Increase the balance {}", new_balance);
    }));

    // withdrawal thread
    threads.push(thread::spawn(move || {
        let mut new_balance = balance;
        new_balance -= 300;
        println!("Decrease the balance {}", new_balance);
    }));

    for thread in threads {
        let _ = thread.join();
    }
    println!("Final balance {}", balance);
}
```

Rust concurrency

- Concurrency
  - Shared **read-only** accesses
  - Shared **mutable** accesses
  - Concurrency example:

Mutable by threads:
- Mutex: mutable if we lock() the variable
- Arc : send-able to multiple threads

Deposit thread:
- Line 1: clone the Arc instance; point to the same.
- Line 2: lock and get the balance value
- Line 3: increase 100 (cf. access with *)

Withdrawal thread:
- The same as the deposit thread
– Decrease the balance by $300

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

fn main() {
  let balance = Arc::new(Mutex::new(200));
  let mut threads = vec![];

  // deposit thread
  let balance4deposit = Arc::clone(&balance);
  threads.push(thread::spawn(move || {
     let mut new_balance = balance4deposit.lock().unwrap();
     *new_balance += 100;
     println!("Increase the balance {}", new_balance);
  }));

  // withdrawal thread
  let balance4withdrawal = Arc::clone(&balance);
  threads.push(thread::spawn(move || {
     let mut new_balance = balance4withdrawal.lock().unwrap();
     *new_balance -= 300;
     println!("Decrease the balance {}", new_balance);
  }));

  for thread in threads {
     let _ = thread.join();
  }

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

Rust Concurrency

- Concurrency
  - Shared **read-only** accesses
  - Shared **mutable** accesses
  - Concurrency example:

Results:
$ ./main
Increase the balance 300
Decrease the balance 0
Final balance 0

Note:
"balance" is a mutable shared variable
"new_balance" points to the mutable variable
Require to wrap with Arc for sending to threads
Modify the value is only available after lock()

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

fn main() {
   let balance = Arc::new(Mutex::new(200));
   let mut threads = vec![];

   // deposit thread
   let balance4deposit = Arc::clone(&balance);
   threads.push(thread::spawn(move || {
      let mut new_balance = balance4deposit.lock().unwrap();
      *new_balance += 100;
      println!("Increase the balance {}", new_balance);
   }));

   // withdrawal thread
   let balance4withdrawal = Arc::clone(&balance);
   threads.push(thread::spawn(move || {
      let mut new_balance = balance4withdrawal.lock().unwrap();
      *new_balance -= 300;
      println!("Decrease the balance {}", new_balance);
   }));

   for thread in threads {
      let _ = thread.join();
   }

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

Unsafe code in Rust

- Safety that Rust offers:
  - **Memory safety**
    - Cannot mutate an immutable variable
    - To modify a mutable variable in a function:
      - The function should own the variable (ownership)
      - The function that just borrows the variable cannot mutate it (borrowing)
  - **Data-race freedom**
    - Threads cannot mutate a shared variable without “locking”

- Safety that is “out-of-scope”:
  - Deadlocks (not the data-race)
  - …

Unsafe code in Rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Unsafe code in Rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static variable:
- "anumber" can be accessible in any code in this file

Create 10 threads:
- Each thread prints the thread index and "anumber"

Results:

```
$ ./main
Thread 0: anumber is 10
Thread 4: anumber is 10
Thread 5: anumber is 10
Thread 2: anumber is 10
Thread 8: anumber is 10
…
```

```rust
use std::thread;

static anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      println!("Thread {}: anumber is {}", tidx, anumber);
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }
}
```

Unsafe code in Rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static variable:
- "anumber" can be accessible in any code in this file

Create 10 threads:
- It will return a Rust **compilation error**
- Rust prevents us from directly modifying static mut
- Rust prohibits us from even just accessing it

```
use std::thread;

static mut anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      println!("Thread {}: anumber is {}", tidx, anumber);
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }
}
```

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Static (mutable) variable:
- We want "anumber" can be **modified** in any code

Create 10 threads:
- Use "unsafe" keyword if we modify "anumber"
- "unsafe" means we understand the consequences
- Now each thread will increase "anumber" by 10

Print out the static mutable:
- Use "unsafe" even for just printing out

```
use std::thread;

static mut anumber: i32 = 10;

fn main() {
   let mut threads = vec![];

   for tidx in 0..10 {
      threads.push(thread::spawn(move || {
         unsafe {
            anumber += 1;
            println!("Thread {}: anumber is {}", tidx, anumber);
         }
      }));
   }

   for thread in threads {
      let _ = thread.join();
   }

   unsafe {
      println!("The final anumber is {}", anumber);
   }
}
```

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Results:

```
$ ./main
Thread 0: anumber is 20
Thread 2: anumber is 30
Thread 3: anumber is 40
Thread 4: anumber is 50
Thread 5: anumber is 60
Thread 7: anumber is 70
Thread 1: anumber is 80
Thread 6: anumber is 90
Thread 8: anumber is 100
Thread 9: anumber is 110
The final anumber is 110
```

```rust
use std::thread;

static mut anumber: i32 = 10;

fn main() {
  let mut threads = vec![];

  for tidx in 0..10 {
    threads.push(thread::spawn(move || {
      unsafe {
        anumber += 1;
        println!("Thread {}: anumber is {}", tidx, anumber);
      }
    }));
  }

  for thread in threads {
    let _ = thread.join();
  }

  unsafe {
    println!("The final anumber is {}", anumber);
  }
}
```

Unsafe code in rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

A variable:
- "s" contains the address of the string "123"

A (pointer) variable:
- "ptr" is the pointer for the string "123"
- "ptr" is "constant" and the type of "u8"

Dereference the pointer values:
- "ptr.offset(#)" is the same as *(ptr + 1) in C
- "as char" converts the output of "ptr.offset" as char
- It causes a **compilation error (Rust prevents this)**

```
fn main() {
    let s: &str = "123";
    let ptr: *const u8 = s.as_ptr();

    println!("{}", *ptr.offset(1) as char);
    println!("{}", *ptr.offset(2) as char);
}
```

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

A variable:
- "s" contains the address of the string "123"

A (pointer) variable:
- "ptr" is the pointer for the string "123"
- "ptr" is "constant" and the type of "u8"

Access the pointer values:
- Use "unsafe" to do the pointer arithmetic
- "unsafe" means we understand the consequences
- It causes a **compilation error (Rust prevents this)**

```
fn main() {
    let s: &str = "123";
    let ptr: *const u8 = s.as_ptr();

    unsafe {
        println!("{}", *ptr.offset(1) as char);
        println!("{}", *ptr.offset(2) as char);
    }
}
```

Results:
```
$ ./main
2
3
```

What Does It Mean by "Understanding the Consequences"?

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

Access the out-of-bound values:
- "*ptr.offset(3)" accesses the 4th character [?!]

Results:
```
$ ./main
2
3
10
```

```
fn main() {
    let s: &str = "123";
    let ptr: *const u8 = s.as_ptr();

    unsafe {
        println!("{}", *ptr.offset(1) as char);
        println!("{}", *ptr.offset(2) as char);
        println!("{}", *ptr.offset(3));
    }
}
```

Unsafe code in rust

- What can be "unsafe" in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

An external function:
- The function "abs" is defined in C (not in Rust)

Use of the external function:
- A **compilation error** (cannot call "abs" *directly*)
- Not sure whether the abs implementation is safe

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    println!("Absolute value of -3 according to C: {}", abs(-3));
}
```

Unsafe code in Rust

- Allow "unsafe" code in Rust:
  - Mutate a static mutable variable
  - Dereference a raw pointer
  - Call external functions (not defined with Rust)

An external function:
- The function "abs" is defined in C (not in Rust)

Use of the external function:
- Use "unsafe" to call the "abs" function
- Not sure whether the abs implementation is safe

Results:

```
$ ./main
Absolute value of -3 according to C: 3
```

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

Topics for today

- Rust
  - Motivation
    - Problem: control vs. safety
    - Solution: Rust
  - Core concepts
    - Ownership and borrowing
    - Concurrency
    - Unsafe code
  - Benefits
    - No need for a runtime
    - Memory safety
    - Data-race freedom
  - Example practice
    - Multi-threaded map-reduce

Rust advantages

- Rust addresses these problems:
  - Runtime check and performance
    - Rust does not require to use GC
    - Rust users (who write the code) consider memory allocations
    - Rust performs compilation time checks
  - Memory safety (no *explicit* allocation/de-allocation)
    - Memory allocations are handled by “ownerships” and “borrowing”
    - Only one “owner” exists at a time; “ownership” transfers if we pass the variable to fn
    - “borrowing” allows to access data without “own”ing it
  - No data-race condition
    - Shared data have two types: “read-only” and “mutable”
    - “read-only” data can only be read by others (*e.g.*, threads that access it)
    - “mutable” data can only be read after the lock()

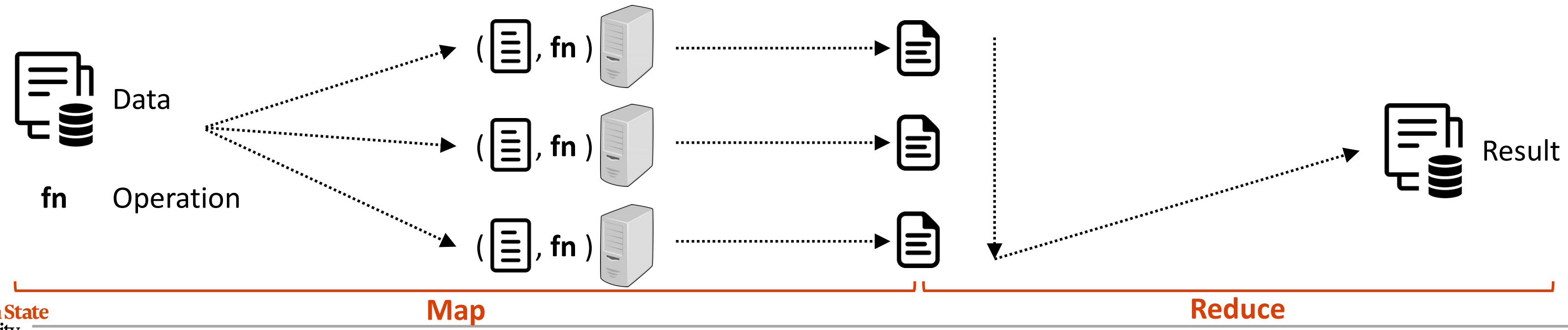
Topics for today

- Rust
  - Motivation
    - Problem: control vs. safety
    - Solution: Rust
  - Core concepts
    - Ownership and borrowing
    - Concurrency
    - Unsafe code
  - Benefits
    - No need for a runtime
    - Memory safety
    - Data-race freedom
  - Example practice
    - Multi-threaded map-reduce

Background: map-reduce

- Map-reduce:
  - **Definition:** a programming model to process large-scale datasets in parallel on a cluster
  - **TL; DR:** *Map* large data to multiple machines, run in parallel, and *reduce* the results

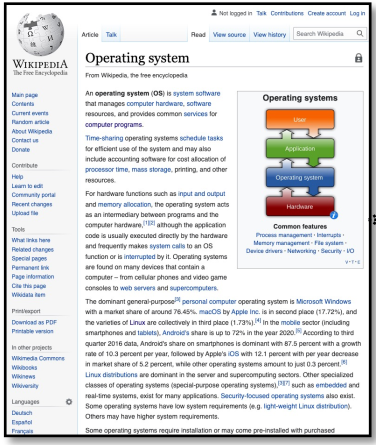
- Procedure:
  - Define a set of operations required to run on the entire data
  - Split the data into multiple chunks (and send them to multiple machines)
  - **Map** the operations to each split and compute intermediate results in parallel
  - **Reduce** the intermediate results into a final output



Background: map-reduce

- Data abstraction:
  - Key/value pairs
  - ex. in word-counts, (“cs344”, 5) as (key, value)

- Word count example:



Entire Wiki articles

Map

(Wikipedia, 1)
(OS, 10)
(operating, 3)
(task, 5)
(abstraction, 2)
…

……

(storage, 1)
(systems, 45)
(Linux, 7)
(task, 5)
…

Intermediate results

Reduce

(Wikipedia, 1)
(OS, **13**)
(operating, 3)
(task, **8**)
(abstraction, 2)
(storage, 1)
(systems, 45)
(Linux, 7)
…

Final results

ECO I: MULTI-THREADED MAP REDUCE IN RUST

- Goal
  - Compute the sum of integers in an array in a map-reduce manner

- Runtime outputs:

./main <# partition> <# of integers>

```
$ ./main 5 150
Number of partitions = 2
        size of partition 0 = 75
        size of partition 1 = 75
Intermediate sums = [2775, 8400]
Sum = 11175
Number of partitions = 5
        size of partition 0 = 30
        size of partition 1 = 30
        size of partition 2 = 30
        size of partition 3 = 30
        size of partition 4 = 30
        Intermediate sums = [435, 1335, 2235, 3135, 4035]
Sum = 11175
```

Output from the sample code
- Compute the sum of 150 numbers with 2 partitions
- Use this part for sanity-checking

Output that is required to implement
- Compute the sum of 150 numbers with 5 partitions

ECO I: MULTI-THREADED MAP REDUCE IN RUST

- Goal
  - Compute the sum of integers in an array in a map-reduce manner

- Runtime outputs:

./main <# partition> <# of integers>

```
$ ./main 5 150
Number of partitions = 2
        size of partition 0 = 75
        size of partition 1 = 75
Intermediate sums = [2775, 8400]
Sum = 11175
Number of partitions = 5
        size of partition 0 = 30
        size of partition 1 = 30
        size of partition 2 = 30
        size of partition 3 = 30
        size of partition 4 = 30
        Intermediate sums = [435, 1335, 2235, 3135, 4035]

Sum = 11175
```

Note:
- Each partition contains the same number of elements
- **Map:** we divide 150 into 2 x 75 elements
  each thread computes each partition
- Intermediate sums contain the sum from each partition

Note:
- **Reduce:** compute the sum of the intermediate sums

ECO I: MULTI-THREADED MAP REDUCE IN RUST

- Plan of attack
  - **Must:** start by reading the description on Canvas
  - **Must:** understand the sample program provided and compile+run it
  - **Must:**
    - Implement “partition_data” function
    - **Map:** create # threads (= # partitions) that compute the intermediate sums
    - Store the intermediate sums returned from each thread
    - **Reduce:** run “reduce_data” and print out the final sum

Topics for today

- Rust
  - Motivation
    - Problem: control vs. safety
    - Solution: Rust
  - Core concepts
    - Ownership and borrowing
    - Concurrency
    - Unsafe code
  - Benefits
    - No need for a runtime
    - Memory safety
    - Data-race freedom
  - Example practice
    - Multi-threaded map-reduce

Thank You!

Mon/Wed 12:00 – 1:50 PM

Sanghyun Hong

sanghyun.hong@oregonstate.edu

Oregon State University

SAIL
Secure AI Systems Lab