Software Systems

An introduction to Software Systems and Concurrency in Rust

Vivian Roest
Jonathan Dönszelmann

Delft University of Technology, The Netherlands

November 16, 2024

Today's Lecture

Course Information

- Course overview
- Staff
- Deadlines and Grading
- Group work

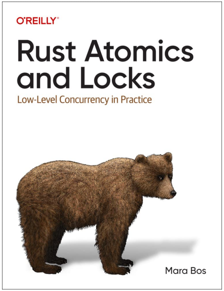
Mutability and Concurrency

- Globals
- Interior Mutability
- Send + Sync
- Concurrency

Software Systems

- Continuation of Software Fundamentals[a]
- Two parts
  1. Rust for embedded systems
  2. Model based software development,

[a] The homologation course for Electrical Engineers, that was focused on learning programming (in Rust). If you haven't done this course: there are notes available on the course website and at `https://cese.ewi.tudelft.nl`



`https://marabos.nl/atomics/`

Staff

- Part 1:
  - Vivian Roest
  - George Hellouin de Menibus (Head TA)
  - & TAs
- Part 2:
  - Rosilde Corvino
  - Guohao Lan
  - & TAs

Deadlines

- Wednesday Week 3 (27 November)
  Assignment 1 about concurrency
- Wednesday Week 4 (4 December)
  Assignment 2 about performance
- Wednesday Week 6 (18 December)
  Assignment 3 about embedded development

Programs, processes and threads

Question:

What's the difference between a program, a process and a thread?

Programs, processes and threads

Question:

What's the difference between a program, a process and a thread?

Question:

Is multithreading possible on a single core system?

Programs, processes and threads

- Programs: Binaries and Scripts
- Process: An instantiation of a program
- Subprocess: A copy of a process without shared memory
- Thread: Subprocess with shared memory

Concurrency vs Parallelism

Question:

Is multithreading possible on a single core system?

- Parallelism: two things are happening at exactly the same time
- Concurrency: two things happen intertwined.
- Multithreading doesn't need to be parallel, it can be concurrent

Scheduler

- More threads/processes than cores
- Concurrency: the illusion of parallelism
- Priorities and `niceness`
- Generally:
  - Processes get time slices
  - Processes can yield their remaining time
  - The scheduler can cooperate with needs of processes

Scheduler

```
std::thread::yield_now();

// approximately the same as
std::thread::sleep(Duration::from_secs(0));

```

- Make sure another thread is temporarily allowed to execute

Blocking

Question:

What does **blocking** mean in the context of concurrent execution?

Blocking

- When waiting for a lock to unlock
- Another thread is allowed to run
- The blocking thread is restarted when it can make progress again
- More efficient than waiting in an infinite loop

Memory Safety

- Buffer Overflow
- Use After Free
- Double Free
- Race Conditions
- Null Pointer Dereference

Why does Rust work like it does?

Concurrency bugs in C

```c
int v = 0;

void count(int* delta) {
    for (int i = 0; i < 100000; i++) v += *delta;
}

int main() {
    thrd_t t1, t2;
    int d1 = 1, d2 = -1;

    thrd_create(&t1, count, &d1);
    thrd_create(&t2, count, &d2);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("%d\n", v);
}

```

Aliasing

- One memory location
- Accessed through multiple pointers

```rust
fn update(a: &mut u64) {
    *a += 1;
}
```

```
update:
    push    rbp
    mov     rbp,rsp
    mov     eax,DWORD PTR [rip+0x2f18]
    add     eax,0x1
    mov     DWORD PTR [rip+0x2f0f],eax
    mov     eax,0x0
    pop     rbp
    ret
```

Mutexes

```c
int v = 0, n = 100000;
mtx_t m;

void count(int* delta) {
    for (int i=0; i<n; i++) {
        mtx_lock(&m);
        v += *delta;
        mtx_unlock(&m);
    }
}

int main() {
    mtx_init(&m, mtx_plain);

    // spawn and stop threads

    printf("%d\n", v);
}
```

Atomics

```
#include<stdatomic.h>

_Atomic int v = 0;
int n = 100000;

void count(int* delta) {
    for (int i=0; i<n; i++) {
        atomic_fetch_add_explicit(
            &v,
            *(int *)delta,
            memory_order_relaxed
        );
    }
}

int main() {
    // spawn and stop threads
    printf("%d\n", v);
}
```

Ordering

Atomics require ordering[1]

```rust
pub enum Ordering {
    Relaxed,
    Release,
    Acquire,
    AcqRel,
    SeqCst, // Always correct
}

```

[1] More info: https://marabos.nl/atomics/memory-ordering.html

Concurrency

```rust
use std::thread::spawn;
static v: i32 = 0;

fn count(delta: i32) {
    for _ in 0..100_000 {
        v += delta;
    }
}

fn main() {
    let t1 = spawn(|| count(1));
    let t2 = spawn(|| count(-1));

    t1.join().unwrap();
    t2.join().unwrap();

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

Concurrency

```
use std::thread::spawn;
static mut v: i32 = 0;

fn count(delta: i32) {
    for _ in 0..100_000 {
        v += delta;
    }
}

fn main() {
    let t1 = spawn(|| count(1));
    let t2 = spawn(|| count(-1));

    t1.join().unwrap();
    t2.join().unwrap();

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

Concurrency

```
error[E0133]: use of mutable static is unsafe and requires unsafe
↪  function or block
 --> src/main.rs:7:9
  |
7 |         v += delta;
  |         ^^^^^^^^^^ use of mutable static
  |
  = note: mutable statics can be mutated by multiple threads:
  aliasing violations or data races will cause undefined behaviour
```

Intermezzo: Unsafe

- Relaxes some of Rust's strict guarantees
- Puts the programmer in charge of creating **sound** programs.
- More in Lecture 3.

Concurrency

```
static mut v: i32 = 0;

fn count(delta: i32) {
    for _ in 0..100_000 {
        unsafe{v += delta};
    }
}

fn main() {
    let t1 = thread::spawn(|| count(1));
    let t2 = thread::spawn(|| count(-1));

    t1.join().unwrap();
    t2.join().unwrap();

    println!("{}", unsafe{v});
}
```

Rust Playground

Data Sharing Rules

- Sharing data immutably is okay
- Moving values to other threads is okay
- Sharing data mutably is not okay.

Notice that these rules are the same as those enforced by the borrow checker!

Mutex as a Monad

```c
int v = 0, n = 100000;
mtx_t m;

void count(int* delta) {
    for (int i=0; i<n; i++) {
        mtx_lock(&m);
        v += *delta;
        mtx_unlock(&m);
    }
}

int main() {
    mtx_init(&m, mtx_plain);

    // spawn and stop threads

    printf("%d\n", v);
}

```

Mutex as a Monad

Mutexes work like options, they wrap the value. Providing added context to the data, guarding it.

```rust
static v: Mutex<i32> = Mutex::new(0);

fn count(delta: i32) {
    for _ in 0..100_000 {
        *v.lock() += delta;
    }
}
```

Mutex as a Monad

What does a Mutex look like?

```
pub struct Mutex<T: ?Sized> {
    inner: sys::MovableMutex,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

- UnsafeCell: Interior mutability
- Allows mutations even when not mutable
- Unsafe to use, like mutable statics
- Mutex has OS help

Removing globals

What would happen if we made the value a local variable:

```
fn count(delta: i32, v: &Mutex<i32>) {
    for i in 0..100_000 {
        *v.lock() += delta
    }
}

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

    let t1 = thread::spawn(|| count(1, &v));
    let t2 = thread::spawn(|| count(-1, &v));

    t1.join().unwrap();
    t2.join().unwrap();

    println!("{}", v.lock());
}
```

Removing globals

```
  error[E0373]: closure may outlive the current function, but it
   ↪  borrows `v`, which is owned by the current function
  --> src/main.rs:14:28
   |
14 |     let t1 = thread::spawn(|| count(1, &v));
   |                            ^^            - `v` is borrowed here
   |                            |
   |                            may outlive borrowed value `v`
   |
note: function requires argument type to outlive `'static`
  --> src/main.rs:14:14
   |
14 |     let t1 = thread::spawn(|| count(1, &v));
   |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Removing globals

- The spawned threads *could* run for longer than the main thread
- `v` is deallocated at the end of main
- the threads may need `v` for longer

Removing globals

- The spawned threads *could* run for longer than the main thread
- `v` is deallocated at the end of main
- the threads may need `v` for longer
- Solution: Throw `v` on the heap so it could live longer!

Boxing it up

```
fn count(delta: i32, v: Box<Mutex<i32>>) {
    for i in 0..100_000 {
        *v.lock() += delta
    }
}

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

    let t1 = thread::spawn(|| count(1, v));
    let t2 = thread::spawn(|| count(-1, v));

    t1.join().unwrap();
    t2.join().unwrap();

    println!("{}", v.lock());
}
```

Boxing it up

We try to access the same `Box` at different locations.

```
error[E0382]: use of moved value: `v`
12 |     let v = Box::new(Mutex::new(0));
   |         - move occurs because `v` has type `Box<_>`,
   |           which does not implement the `Copy` trait
13 |
14 |     let t1 = thread::spawn(|| count(1, v));
   |                            --          - variable moved
   |                            |             due to use in closure
   |                            |
   |                            value moved into closure here
   |
15 |     let t2 = thread::spawn(|| count(-1, v));
   |                            ^^           - use occurs
   |                            |              due to use in closure
   |                            |
   |                            value used here after move
```

So when do we deallocate?

Reference Counting

- Keep track of the number of owners
- When it reaches 0 → deallocate
- When we clone the reference, increment the reference count
- Rc doesn't implement Copy like other references, because Copying them is *not* trivial
- Like C++ `std::shared_ptr`

Reference Counting

```rust
fn create_vectors() -> (Rc<Vec<i32>>, Rc<Vec<i32>>) {
    // one vector on the heap. rc=1
    let a = Rc::new(vec![1, 2, 3]);

    // two *extra* references to it. rc=3
    let ref_1 = a.clone(); // doesn't clone the vec! only the ref!
    let ref_2 = a.clone(); // doesn't clone the vec! only the ref!

    (ref_1, ref_2) // return both. rc=2
}

fn main() {
    let (a, b) = create_vectors(); // Both are the same vector
    println!("{}", a);
    println!("{}", b);
    // rc finally drops to 0
}
```

Rc is clonable even if the internal value is not.

Reference Counting

```
error[E0277]: `Rc<Mutex<i32>>` cannot be shared between threads safely
   --> src/main.rs:15:14
    |
15  |     let t1 = thread::spawn(|| count(1, v.clone()));
    |              ^^^^^^^^^^^^^^ `Rc<Mutex<i32>>` cannot be shared
    |                             between threads safely
    |
    = help: the trait `Sync` is not implemented for `Rc<Mutex<i32>>`
    = note: required because of the requirements on the
            impl of `Send` for `&Rc<Mutex<i32>>`
```

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

Sharing and Reference Counting

Question:

Why can't we share an `Rc` between two threads?

Sharing and Reference Counting

Question:

Why can't we share an `Rc` between two threads?

- Keeping track of ownership means updating the reference count
- Updating the reference count needs mutability (on clone and drop)
- Within one thread that's safe

So apparently, it is not safe to send or share some types between threads.

T: Send + Sync

`Send`: It is safe to send `T` to another thread

`Sync`: It is safe to share a `T` with another thread (`T` is `Sync` iff `&T` is `Send`)

Reference Counting, Send and Sync

```
error[E0277]: `Rc<Mutex<i32>>` cannot be shared between threads safely
   --> src/main.rs:15:14
    |
15  |     let t1 = thread::spawn(|| count(1, v.clone()));
    |              ^^^^^^^^^^^^^ `Rc<Mutex<i32>>` cannot be shared
    |                             between threads safely
    |
    = help: the trait `Sync` is not implemented for `Rc<Mutex<i32>>`
    = note: required because of the requirements on the
            impl of `Send` for `&Rc<Mutex<i32>>`
```

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

```

Reference Counting, Send and Sync

```
impl<T: ?Sized> !Send for Rc<T> {}
impl<T: ?Sized> !Sync for Rc<T> {}
```

Atomic Reference Counter (Arc)

- We could use a `Mutex` again
- Atomics
- `Arc<T>: Send + Sync`
- So we can use it to share references between threads

Fixing the code with Arcs

```
fn count(delta: i32, v: Arc<Mutex<i32>>) {
    for i in 0..100_000 {
        *v.lock() += delta
    }
}

fn main() {
    let v = Arc::new(Mutex::new(0));
    let (v1, v2) = (v.clone(), v.clone());

    let t1 = thread::spawn(|| count(1, v1));
    let t2 = thread::spawn(|| count(-1, v2));

    t1.join().unwrap();
    t2.join().unwrap();

    println!("{}", v.lock());
}
```

Concurrency in other languages

- Python: Doesn't really have it
- Java: Every object has a "thin lock"
- C or C++: You're responsible
- Go: You're responsible
- Haskell: No mutability
- JavaScript/TypeScript: relies on async/await
- Rust: The Compiler is responsible

Break

- See you back in 15 minutes
- Crash course concurrent programming.

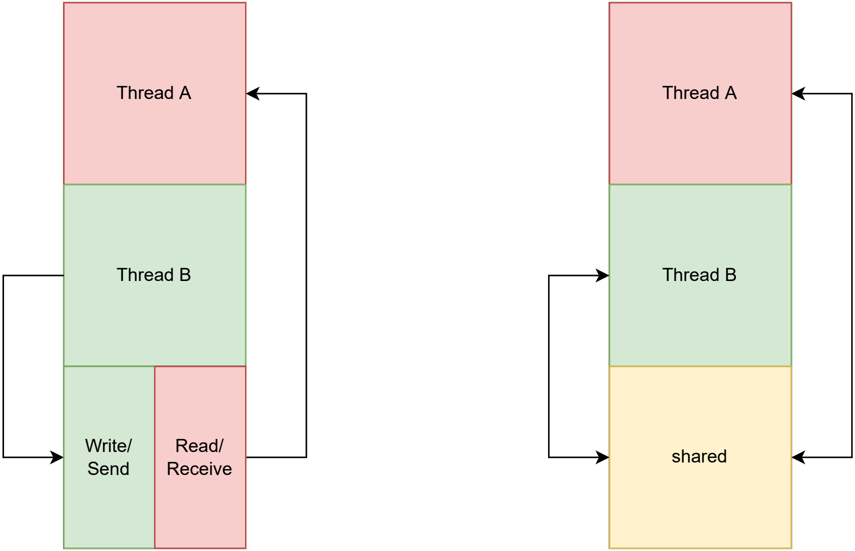
Communicating between threads

1. Communicate by sharing memory
   - Make memory mutable from multiple threads
   - Needs locking to avoid data races
   - Can be quite slow when there is lots of communication
   - Useful for exchanging large amounts of data
   - More complex logic

Communicating between threads

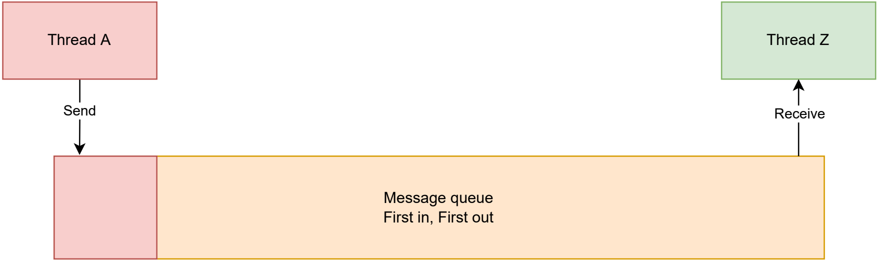
1. Communicate by sharing memory
   - Make memory mutable from multiple threads
   - Needs locking to avoid data races
   - Can be quite slow when there is lots of communication
   - Useful for exchanging large amounts of data
   - More complex logic
2. Communicating by sending messages
   - No need for explicit locking
   - Easy to reason about
   - Safe by design: No mutability problems

Communicating between threads



Channels

- Like a queue, first in, first out
- One end in one thread, other end in another thread
- Lock-free insertion (magic / atomics depending on who you ask)
- Receivers **block** and wait when there are no messages



Channels

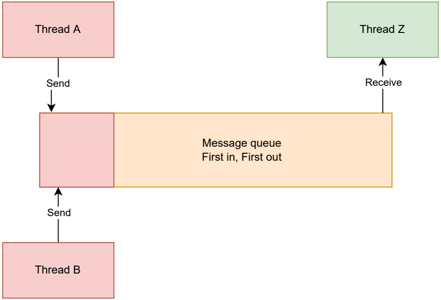
A simple example

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

fn main() {
    // tx: Sender<i32>, rx: Receiver<i32>
    let (tx, rx) = channel();

    spawn(move || {
        while let Ok(i) = rx.recv() {
            println!("hello, {i}")
        }
    });

    tx.send(1).unwrap();
    tx.send(2).unwrap();
    tx.send(3).unwrap();
}
```

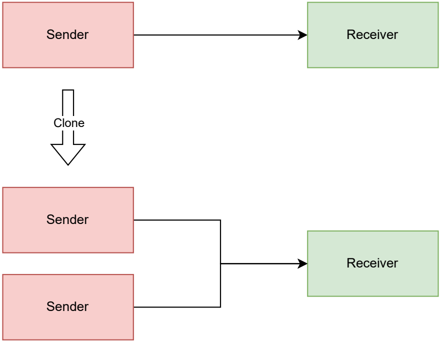
Channels: mpsc

- Multi Producer Single Consumer
- Senders can be cloned, receivers can not

Multiple Producers



Cloning



Channels: Cloning Senders

```rust
fn main() {
    let (tx, rx) = channel();

    spawn(move || {
        while let Ok(i) = rx.recv() {
            println!("hello, {i}")
        }
    });

    // clone senders
    let tx1 = tx.clone();
    let tx2 = tx.clone();

    spawn(move || for i in 0..5  {tx1.send(i).unwrap()});
    spawn(move || for i in 5..10 {tx2.send(i).unwrap()});
}
```

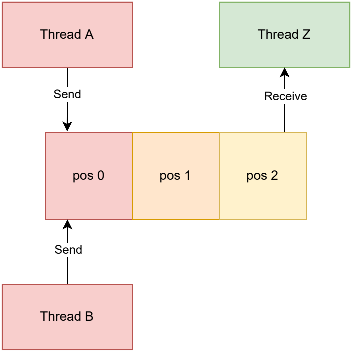
Channels: multiple receivers?

What are the semantics?

- Send messages to all receivers
- Send messages to one receiver
- Various libraries implement both

Bounded Channels

- By default, channels are unbounded
- Problem: senders can get ahead of receivers
- Solution: Buffer with a certain size: Bounded channel / sync channel
- Senders can block



Bounded Channels Example

```rust
fn main() {
    let (tx, rx) = sync_channel(3);

    spawn(move || {
        while let Ok(i) = rx.recv() {
            println!("hello, {i}")
        }
    });

    // clone senders 40 times
    for i in 0..40 {
        let tx_clone = tx.clone();
        spawn(move || for j in (i*10)..(i*10+10) {
            tx_clone.send(j).unwrap()
        });
    }
}
```

Bounded Channels: 0 size

Question:

What happens when the size is 0?

Bounded Channels: 0 size

- Sender blocks until a receiver is ready

```rust
fn main() {
    let (tx, rx) = sync_channel(0);

    spawn(move || {
        while let Ok(recv_msg) = rx.recv() {
            println!("hello, {i}")
        }
    });

    // clone senders 40 times
    for thread_count in 0..40 {
        let tx_clone = tx.clone();
        spawn(move || for i in (thread_count*10)..(thread_count*10+10) {
            tx_clone.send(i).unwrap()
        });
    }
}
```

Thread Management

The lifecycle of a thread:

```
let handle = spawn(|| {...});

...

handle.join();

```

Thread Management

The lifecycle of multiple threads:

```
let mut handles = Vec::new();

for ... {
    handles.push(spawn(|| {...}));
}

...

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

```

Question:

How many threads should we spawn?

Thread Management

The lifecycle of multiple threads:

```
let mut handles = Vec::new();

for ... {
    handles.push(spawn(|| {...}));
}

...

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

```

Insight

It is not always trivial to determine the amount of threads to spawn.

Thread Management

```rust
fn task_runner(rx: Receiver<Task>) {
    while let Some(task) = rx.recv() { task() }
}

let mut txs = Vec::new();
for _ in num_cores {
    let (rx, tx) = channel();
    spawn(|| task_runner(rx));
    txs.push(tx);
}

// Execute task(s) on a random thread
txs.choose(thread_rng()).send(some_task)
```

Note: Task could for example be a function pointer

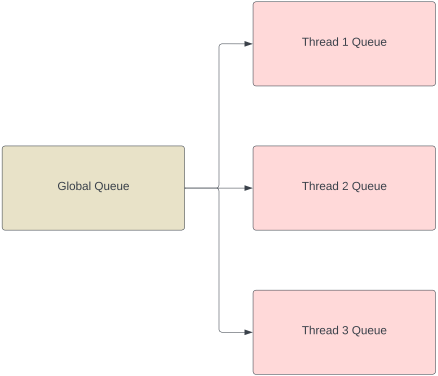
Thread Management

```rust
fn task_runner(rx: Receiver<Task>) {
    while let Some(task) = rx.recv() { task() }
}

let mut txs = Vec::new();
for _ in num_cores {
    let (rx, tx) = channel();
    spawn(|| task_runner(rx));
    txs.push(tx);
}

// Execute task(s) on a random thread
txs.choose(thread_rng()).send(some_task);
```

Question:

Does this ensure an equal distribution of work?

Thread Management: work stealing



Thread 1 queue and global queue empty? Steal from another thread!

Thread Management: Work Stealing

```
use rayon::ParallelIterator;

let tasks: [Task; 3] = [taska, taskb, taskc];
tasks
    .into_par_iter()
    .for_each(|x| x())
```

Waiting for IO

Situation:

1. A webserver is handling many connections.
2. Each connection has an associated TCP socket
3. Every time we receive a packet, we have to process it for 0.1ms
4. Every time we send a packet, it takes about 10 seconds to get a response

Question:

How many threads do we need for maximum utilization?

Waiting for IO

But the model of ‘tasks’ is really useful!

```
fn task() {
    let conn = start_connection();
    while let Some(packet) = conn.recv() {
        conn.send(process(packet));
    }
}

for i in 0..=many {
    spawn(task);
}
```

Take scheduling into our own hands!

Don't let the OS schedule threads

- Make our own lightweight 'task'
- Make our own scheduler
- Tasks tell the scheduler what event they're waiting for
- The scheduler wakes up a task when the OS says the event has happened

Examples:

- Java, Elixir: Green Threads
- Go: Goroutines
- Python, Javascript, C++, Rust: Async/IO

Take scheduling into our own hands!

Problem:

How do we preempt a task?

- ‘Easy’ for interpreted languages
- Go inserts extra runtime checks

Take scheduling into our own hands!

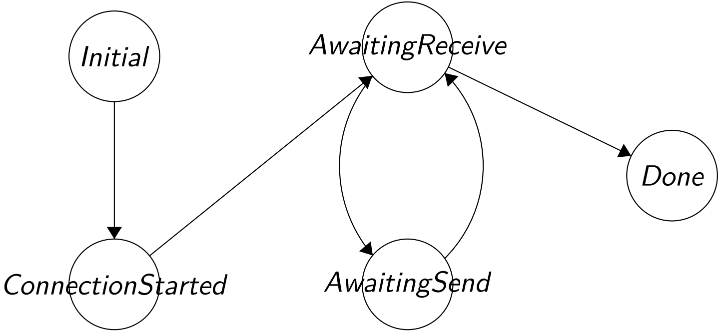
Find the places where we could start doing something else.
Modelling I/O as a state machine.

```
let conn = start_connection();
while let Some(packet) = conn.recv() {
    conn.send(process(packet));
}
```

States:

- `Initial`
- `ConnectionStarted{conn}`
- `AwaitingReceive{conn}`
- `AwaitingSend{conn, packet}`
- `Done`

Take scheduling into our own hands!



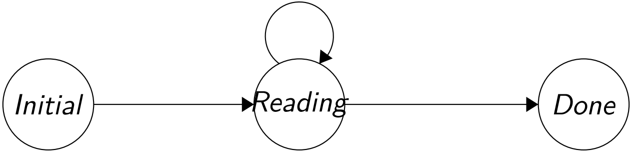
Reading a file

```
fn read_file(file: &Path) -> Vec<u8> {
    let f = File::open(path);
    let buffer = Vec::new();

    loop {
        let mut read_buffer = [0; 2048];
        let num_read = f.read(&mut read_buffer);

        if num_read == 0 { break; }
        buffer.extend(&read_buffer[0..num_read]);
    }
    buffer
}
```

Question:

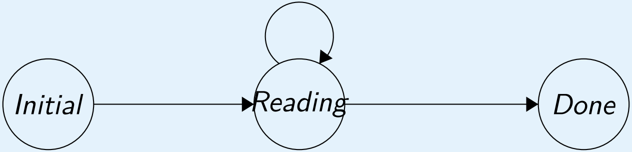
What are the states?

Reading a file

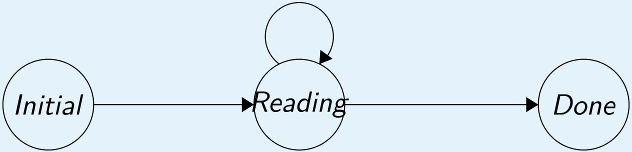


Reading two files simultaneously

Task 1



Task 2



Try to make progress in both: Polling

Asynchronous IO

- https://man7.org/linux/man-pages/man7/aio.7.html
- aio_read like read, but returns immediately
- aio_error see the status (error, ok, or in progress) of a read request
- aio_return get the return value of read, how many bytes were read?

Note: this is *one* option under linux. There is also io_uring for Linux, and similar APIs on Windows and MacOS.

Asynchronous IO

```
pub async fn read_file(path: &Path) -> Result<Vec<u8>, io::Error> {
    let mut file = File::open(path).await?;
    let mut buffer = Vec::new();

    loop {
        let mut read_buffer = Vec::new();
        let num_bytes = file.read(&mut read_buffer).await?;
        if num_bytes == 0 {
            break;
        }

        buffer.extend(&read_buffer[0..num_bytes]);
    }

    Ok(buffer)
}
```

Asynchronous IO

In Rust `.await`

- Sets up an (asynchronous) request to the operating system
- Goes to the next state in the state machine
- Yields to poll another state machine
- When we get back to polling this state machine:
  - ask the OS if we made progress
  - continue executing code

Summary

- Achieving fearless concurrency in Rust
- Using channels to your advantage
- Be conscious of your thread count
- Not all cases call for threads
- Async I/O internals and its state machines

Rounding up

- First lab next thursday
- First assignment online today:
  - creating a concurrent `grep` clone

Lifetimes

- Every value has a lifetime
- After the end of a lifetime, a value is dropped
- Some types have a lifetime associated with them

```rust
fn main() {
    let i = 3; // Lifetime for `i` starts.

    {
        let borrow1 = &i; // `borrow1` lifetime starts.

        println!("borrow1: {}", borrow1); //
    } // `borrow1 ends.


    {
        let borrow2 = &i; // `borrow2` lifetime starts.

        println!("borrow2: {}", borrow2); //
    } // `borrow2` ends.

}   // Lifetime ends.
```

Lifetimes

In functions, you can be explicit about lifetimes

```
fn print_ref<'a>(x: &'a i32) {
    println!("print_ref: x is {}", x);
}
```

In structs, you **have** to be explicit about lifetimes

```
struct Example<'a> {
    field: &'a i32,
}
```

Lifetimes

Sometimes, logic requires two lifetimes to be the same

```
fn longest(a: &str, b: &str) -> &str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}
```

Lifetimes

Sometimes, logic requires two lifetimes to be the same

```
fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() {
        a
    } else {
        b
    }
}
```