Concurrency testing with Loom

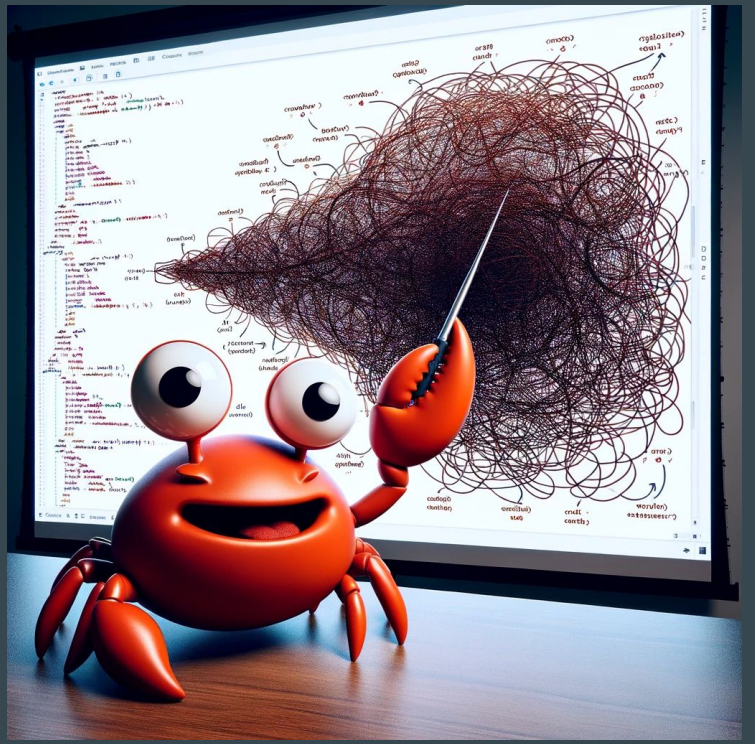
Anton Velikanov

whoami

- using Rust in production since 2018
- studying databases internals in practice
- previously security researcher (audits/bug hunting/ctfs participant)
- ❤️ Linux & OpenSource:
  - https://github.com/bondifuzz
  - I use  btw
  - ...

Agenda

1. Fearless concurrency in Rust
2. Concurrency fundamentals recap
3. Example of tricky concurrent code
4. What is Loom? How to use?
5. Loom usage example
6. Loom limitations
7. Conclusion & references



Fearless concurrency

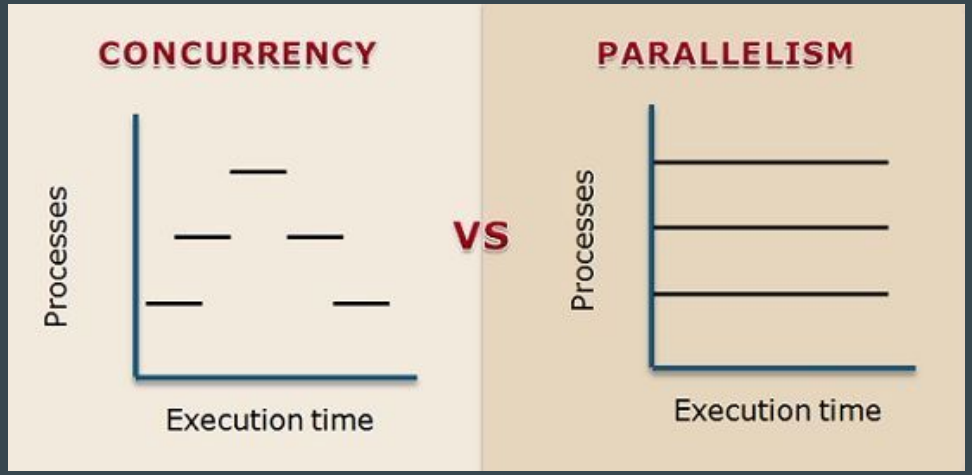
- Send
- Sync
- std::sync::{Mutex, RwLock, Arc, mpsc, atomic}

Concurrency fundamentals recap

- why ?

Concurrency fundamentals recap

- why ?
- concurrency != parallelism
  - single-thread concurrency
  - single-core multithreaded concurrency
  - multicore concurrency



Concurrency fundamentals recap

- why ?
- concurrency != parallelism
- compiler reordering:
  - result stays the same
  - does not take other threads into account
  - require ordering for operations with atomics

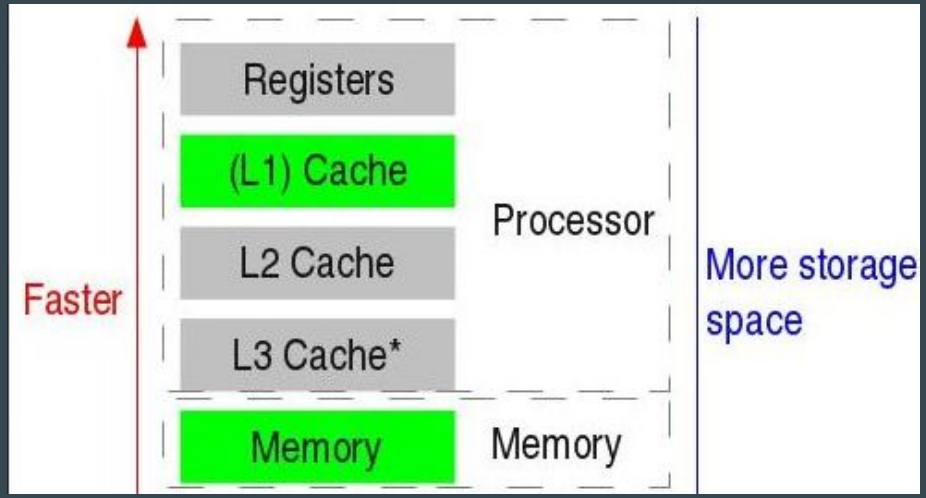
```
fn f(a: &mut i32, b: &mut i32)
{
    *a += 1;
    *b += 1;
    *a += 1;
}
```



```
fn f(a: &mut i32, b: &mut i32)
{
    *a += 2;
    *b += 1;
}
```

Concurrency fundamentals recap

- why ?
- concurrency != parallelism
- compiler reordering
- hardware reordering:
  - CPU registers
  - CPU caches
  - RAM



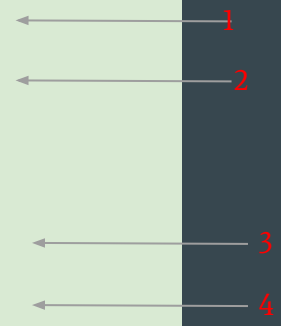
Concurrency fundamentals recap

- memory model
  - architecture agnostic
  - happens-before relationship
  - ignore machine instructions, caches, buffers, timing, instruction reordering, compiler optimizations, etc...
- Rust memory model copied from C++ (almost)

```
pub enum Ordering {
    Relaxed,
    Release,
    Acquire,
    AcqRel,
    SeqCst,
}
```

How reordering looks like?

```
static X: AtomicBool = AtomicBool::new(false);

static Y: AtomicBool = AtomicBool::new(false);


let t1 = spawn(|| {
    let r1 = Y.load(Ordering::Relaxed);
    X.store(r1, Ordering::Relaxed);
});
let t2 = spawn(|| {
    let r2 = X.load(Ordering::Relaxed);
    Y.store(true, Ordering::Relaxed);
});
```



r2 == true ?

How reordering looks like?

```rust
static X: AtomicBool = AtomicBool::new(false);

static Y: AtomicBool = AtomicBool::new(false);


let t1 = spawn(|| {
    let r1 = Y.load(Ordering::Relaxed);
    X.store(r1, Ordering::Relaxed);
});
let t2 = spawn(|| {
    let r2 = X.load(Ordering::Relaxed);
    Y.store(true, Ordering::Relaxed);
});
```

1

2

3

4

r2 == true ?

Valid execution order: 4 1 2 3

What is Loom?

- concurrency testing tool
- part of Tokio project
- model based testing
- track of all cross-thread interactions
- provides replacement for std::sync::*
  and std::thread::*

How to use Loom?

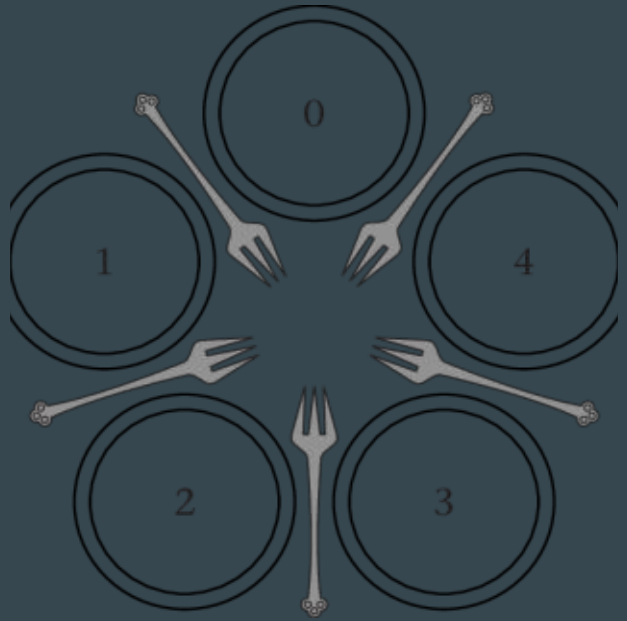
```rust
use std::sync::Arc;
use std::sync::atomic::AtomicUsize;
use std::sync::atomic::Ordering::SeqCst;
use std::thread;

#[test]
fn test_concurrent_logic() {
    let v1 = Arc::new(AtomicUsize::new(0));
    let v2 = v1.clone();

    thread::spawn(move || {
        v1.store(1, SeqCst);
    });

    assert_eq!(0, v2.load(SeqCst));
}
```

```rust
use loom::sync::atomic::AtomicUsize;
use loom::thread;

use std::sync::Arc;
use std::sync::atomic::Ordering::SeqCst;

#[test]
fn test_concurrent_logic() {
    loom::model(|| {
        let v1 = Arc::new(AtomicUsize::new(0));
        let v2 = v1.clone();

        thread::spawn(move || {
            v1.store(1, SeqCst);
        });

        assert_eq!(0, v2.load(SeqCst));
    });
}
```

Example: Dining philosophers problem

Five philosophers dine together at the same table.

- Each philosopher has their own place at the table.
- There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks.
- Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right fork.
- Thus two forks will only be available when their two nearest neighbors are thinking, not eating.
- After an individual philosopher finishes eating, they will put down both forks.



Example: Dining philosophers problem

```
struct Fork;
struct Philosopher {
    name: String,
    // left_fork: ...
    // right_fork: ...
    // thoughts: ...
}
impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        // Pick up forks...
        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Hypatia", "Plato", "Aristotle", "Pythagoras"];

fn main() {
    // Create forks

    // Create philosophers

    // Make each of them think and eat 100 times

    // Output their thoughts
}
```

Example: Dining philosophers problem

```
struct Philosopher {
    name: String,
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
}
impl Philosopher {
    fn eat(&self) {
        let _left = self.left_fork.lock().unwrap() ;
        let _right = self.right_fork.lock().unwrap() ;
        println!("{} is eating...", &self.name);
        std::thread::sleep(Duration::from_millis(10));
    }
}
fn solution() {
    let forks = (0..PHILOSOPHERS.len())
        .map(|_| Arc::new(Mutex::new(Fork)))
        .collect::<Vec<_>>();
    // Create philosophers
    let mut handles = vec![];
    for philosopher in philosophers {
        handles.push(thread::spawn(move || {
            for _ in 0..100 {
                philosopher.eat() ;
            }
        }));
    }
    handles.into_iter().for_each( |h| h.join().unwrap()) ;
}
```

```
let philosophers = PHILOSOPHERS
        .iter()
        .enumerate()
        .map(|(i, name)| {
            let left_fork = Arc::clone(&forks[i]);
            let right_fork = Arc::clone(&forks[(i + 1) %
PHILOSOPHERS.len()]) ;
            Philosopher::new(name.to_string() , left_fork,
right_fork)
        })
        .collect::<Vec<_>>();
```

Changes:

1. `use std::sync::* -> use loom::sync::*`
2. `loom::model(||solution())`

Example: Dining philosophers problem

```
struct Philosopher {
    name: String,
    left_fork: Arc<Mutex<Fork>>,
    right_fork: Arc<Mutex<Fork>>,
}
impl Philosopher {
    fn eat(&self) {
        let _left = self.left_fork.lock().unwrap() ;
        let _right = self.right_fork.lock().unwrap() ;
        println!("{} is eating...", &self.name);
        std::thread::sleep(Duration::from_millis(10));
    }
}
fn solution() {
    let forks = (0..PHILOSOPHERS.len())
        .map(|_| Arc::new(Mutex::new(Fork)))
        .collect::<Vec<_>>();
    // Create philosophers
    let mut handles = vec![];
    for philosopher in philosophers {
        handles.push(thread::spawn(move || {
            for _ in 0..100 {
                philosopher.eat() ;
            }
        }));
    }
    handles.into_iter().for_each( |h| h.join().unwrap()) ;
}
```

```
let philosophers = PHILOSOPHERS
        .iter()
        .enumerate()
        .map(|(i, name)| {
            let left_fork = Arc::clone(&forks[i]);
            let right_fork = Arc::clone(&forks[(i + 1) %
PHILOSOPHERS.len()]) ;
            Philosopher::new(name.to_string() , left_fork,
right_fork)
        })
        .collect::<Vec<_>>();
```

2 minutes later

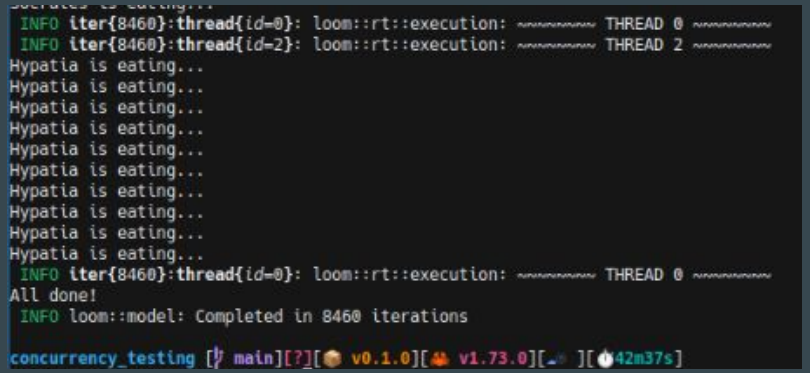
deadlock; threads = [(Id(0), Blocked(Location(Some(Location { file: "src/main.rs", line: 104, col: 40 })))), (Id(1), Blocked(Location(Some(Location { file: "src/main.rs", line: 48, col: 38 })))), (Id(2), Blocked(Locat
ion(Some(Location { file: "src/main.rs", line: 48, col: 38 })))), (Id(3), Blocked(Location(Some(Location { file: "src/main.rs", line: 48, col: 38 }))))]

Example: Dining philosophers problem

```
let philosophers = PHILOSOPHERS
    .iter()
    .enumerate()
    .map(|(i, name)| {
        let mut left_fork = Arc::clone(&forks[i]);
        let mut right_fork = Arc::clone(&forks[(i + 1) % PHILOSOPHERS.len()]);
        if i == forks.len() - 1 {
            std::mem::swap(&mut left_fork, &mut right_fork);
        }
        Philosopher::new(name.to_string(), left_fork, right_fork)
    })
    .collect::<Vec<_>>();
```

Loom limitations

- working time is too long
- combinatorial explosion with many threads
- not cover all executions with Ordering::Relaxed
- several types from std still not implemented



Conclusion

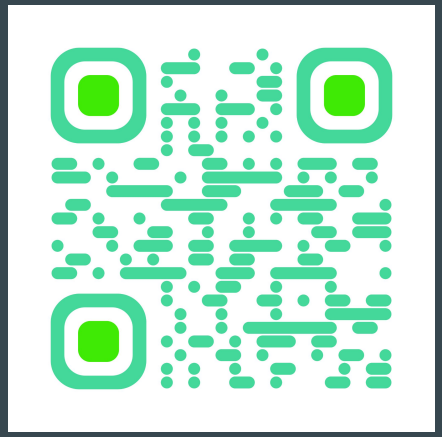
1. Testing of concurrent code is hard, but doable
2. Shuttle by awslabs
3. ThreadSanitizer by Google/LLVM

References:

1. Rustonomicon
2. Jon Gjengset - Rust for rustaceans / YT: Crust of rust
3. Mara Bos - Rust Atomics and Locks
4. Google - Comprehensive Rust course

Questions? Ask!

LinkedIn



Github