Rust in Depth

Lunch session @Junyang

The Genshin of Programming Languages

[***An almost religious case for Rust***] (https://medium.com/@siberianguy/an-almost-religious-case-for-rust-e4c4764acd8)

[***The worst thing about Rust is easily the community***] (https://news.ycombinator.com/item?id=36240704)

[***Linux developers argue over Rust in kernel***] (https://www.techzine.eu/news/devops/128931/linux-developers-argue-over-rust-in-kernel/)

The Genshin of Programming Languages

The Genshin of Programming Languages

OSDI '25 #336

Review #336A
Review #336B
Review #336C
Review #336D
Rebuttal Response
Review #336E
Comment

locking protocol for the same state. Like, could you (in principle) retrofit this onto Linux's VMA?

I'm not convinced that Rust actually buys us much in this world (and I say this as an unabashed Rust evangelist). The modularity argument is tough to make when the whole MMU hardware is a global resource, but I guess the point is that the absence of `unsafe` means most code can't interact with that (no inline assembly in safe code)? Figure 2 is a classic concurrency bug that Rust doesn't protect against, and isn't really specific to this problem, so I with the paper was more direct. I also wish there was some discussion/examples of bugs that Rust *did* protect from.

Dictionary

Definitions from Oxford Languages · Learn more

e·van·ge·list

/əˈvanjələst/

noun

1. a person who seeks to convert others to the Christian faith, especially by public preaching.
   "an American television evangelist"

   Similar: preacher missionary gospeler proselytizer converter

The Genshin of Programming Languages

- [OSDI'24] *SquirrelFS: using the Rust compiler to check file-system crash consistency*
- [OSDI'24] *DRust: Language-Guided Distributed Shared Memory with Fine Granularity, Full Transparency, and Ultra Efficiency*

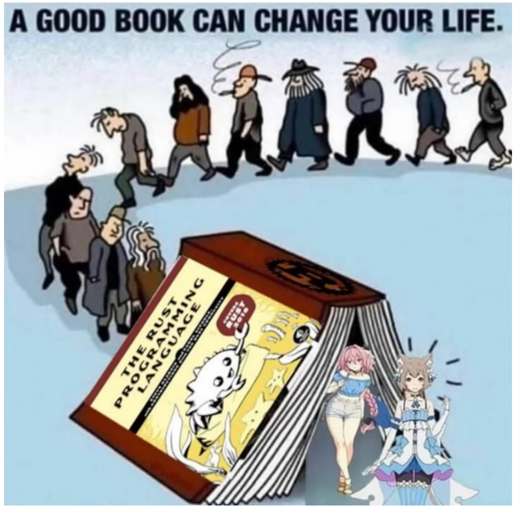
- [OSDI'20] *Theseus: an Experiment in Operating System Structure and State Management*
- [OSDI'20] *RedLeaf: Isolation and Communication in a Safe Operating System*

- [NSDI'25] *Beehive: A Scalable Disaggregated Memory Runtime Exploiting Asynchrony of Multithreaded Programs*
- [ATC'25] *ASTERINAS: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB*

What do we have today

1. A “short” tutorial on Rust
2. Rust in the lab

Awesome resources for Rust learning



Tools:

[***The Rust Playground***](https://play.rust-lang.org/)
[***Compiler Explorer***](https://godbolt.org/)

Book:

[***The Rust Programming Language (with quiz!)***](https://rust-book.cs.brown.edu/)
[***The Rustonomicon***](https://doc.rust-lang.org/nomicon/)

Tutorial topics

1. The Rust type system
2. The Rust borrow checker
3. Unsafe code

The Rust type system

Recall your C++ knowledge for:

	C++	Rust
Public/private class fields	`class Foo { public: int bar; };`	`struct Foo { pub bar: i32 }`
Template programming	`template <typename T>` `T foo(std::shared_ptr<T> bar) {}`	`fn foo<T>(bar: Arc<T>) {}`

Constructors and destructors	`class Foo {`
	`    Foo() {}`
	`    ~Foo() {}`
	`};`
		`struct Foo {}`
		`impl Foo {`
		`    fn new() -> Self {}`
		`}`
		`impl Drop for Foo {`
		`    fn drop(&mut self) {}`
		`}`

Yes, you have them in Rust. (Also, variable scope, closure, operator overloading, …)

The Rust type system

What's different from C++?

- No inheritance/method overloading (use trait instead);
- No raw pointers in safe Rust;
- No reinterpretation cast in safe Rust;
- No implicit copies;
- …

The Rust type system—key idea

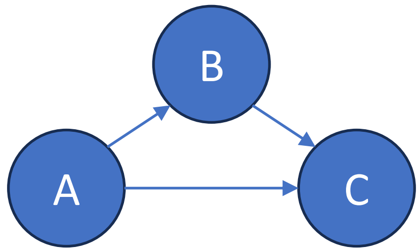
What's different from C++?

1. Strict typing: one memory location (variable) must have only one type;
2. Ownership: a variable must have a unique owner (ultimately the stack or static).

The Rust type system—type state

Exploit 1. strict typing.

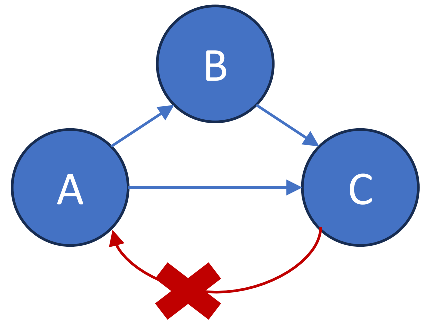
Think of a state machine.



The Rust type system—type state

Exploit 1. strict typing.

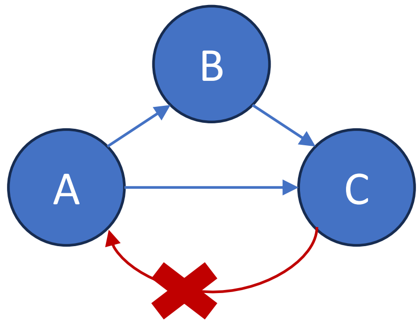
Think of a state machine.
You want the compiler to check
that C->A never happens.



The Rust type system—type state

Exploit 1. strict typing.

Think of a state machine.
You want the compiler to check that C->A never happens.



Write states as types, and don't write C->A conversion.

[Rust Playground Link]()

The Rust type system—type state

Exploit 1. strict typing.

Rust Playground Link

Quiz1: how to break this?

```rust
trait State: Sized {
    fn instance() -> Self;
}
trait Transition<Target: State>: State {
    fn step(self) -> Target {
        Target::instance()
    }
}

struct A;
impl State for A { fn instance() -> Self { A } }
struct B;
impl State for B { fn instance() -> Self { B } }
struct C;
impl State for C { fn instance() -> Self { C } }

impl Transition<B> for A {}
impl Transition<C> for A {}
impl Transition<C> for B {}

pub fn main() {
    let s1: A = A;
    let s2: B = s1.step();
    let s3: C = s2.step();
}
```

The Rust type system—type state

Exploit 1. strict typing.

[Rust Playground Link]

Quiz1: how to break this?

Quiz2: how to prevent such a hole?

```rust
trait State: Sized {
    fn instance() -> Self;
}
trait Transition<Target: State>: State {
    fn step(self) -> Target {
        Target::instance()
    }
}

struct A;
impl State for A { fn instance() -> Self { A } }
struct B;
impl State for B { fn instance() -> Self { B } }
struct C;
impl State for C { fn instance() -> Self { C } }

impl Transition<B> for A {}
impl Transition<C> for A {}
impl Transition<C> for B {}

pub fn main() {
    let s1: A = A;
    let s2: B = s1.step();
    let s3: C = s2.step();
}
```

The Rust type system—type state

Exploit 1. strict typing.

Rust Playground Link

Possible answers:

Quiz1: how to break this?

- Modify the trait bond/add malicious trait implementation;
- Unsafe type casting.

Quiz2: how to prevent such a hole?

- Use a module to encapsulate TCB;
- #[deny(unsafe_code)]

```
trait State: Sized {
    fn instance() -> Self;
}
trait Transition<Target: State>: State {
    fn step(self) -> Target {
        Target::instance()
    }
}

struct A;
impl State for A { fn instance() -> Self { A } }
struct B;
impl State for B { fn instance() -> Self { B } }
struct C;
impl State for C { fn instance() -> Self { C } }

impl Transition<B> for A {}
impl Transition<C> for A {}
impl Transition<C> for B {}

pub fn main() {
    let s1: A = A;
    let s2: B = s1.step();
    let s3: C = s2.step();
}
```

The Rust type system–access control

Exploit 2. ownership.

Think of an OS, the trusted entity assigns resources to users.

The Rust type system–access control

Exploit 2. ownership.

Think of an OS, the trusted entity assigns resources to users.

What's "users"?

The Rust type system–access control

Exploit 2. ownership.

Think of an OS, the trusted entity assigns resources to users.

What’s “users”?

An owner can be:
- A module;
- A thread;
- A CPU;
- Global;
- … any other owners.

The Rust type system—access control

Exploit 2. ownership.

Think of an OS, the trusted entity assigns resources to users.

What's "users"?

An owner can be:
- A module;
- A thread;
- A CPU;
- Global;
- … any other owners.

```rust
mod owner1 {
    static OWNED_RES: SpinLock<Option<IrqLine>> = SpinLock::new(None);
    pub fn yeild_ownership() -> IrqLine {
        OWNED_RES.lock().take().expect("No IRQ line owned")
    }
    pub fn take_ownership(irq: IrqLine) {
        OWNED_RES.lock().replace(irq);
    }
}
mod owner2 {
    static OWNED_RES: SpinLock<Option<IrqLine>> = SpinLock::new(None);
        pub fn yeild_ownership() -> IrqLine {
        OWNED_RES.lock().take().expect("No IRQ line owned")
    }
    pub fn take_ownership(irq: IrqLine) {
        OWNED_RES.lock().replace(irq);
    }
}
```

The Rust type system–access control

Exploit 2. ownership.

Think of an OS, the trusted entity assigns resources to users.

What's “users”?

An owner can be:

- A module;
- A thread;
- A CPU;
- Global;
- … any other owners.

```
let irq_line = IrqLine::alloc().unwrap();
let task = Task::spawn(move || {
    let mut irq_line = irq_line;
    irq_line.on_active(|| { println!("IRQ!"); });
});
```

The Rust type system—access control

Exploit 2. ownership.

Think of an OS, the trusted entity assigns resources to users.

What's "users"?

An owner can be:

- A module;
- A thread;
- A CPU;
- Global;
- … any other owners.

```
cpu_local! {
    pub static OWNED_RES: Option<IrqLine> = None;
}
```

The Rust type system–access control

Exploit 2. ownership.

Think of an OS, the trusted entity assigns resources to users.

What's "users"?

An owner can be:

- A module;
- A thread;
- A CPU;
- Global;
- … any other owners.

Quiz3: How to declare a globally owned resouce?

The Rust borrow checker

Too restrictive if only the owner can access a variable.

Others can borrow a variable mutably/immutably.

The Rust borrow checker

Too restrictive if only the owner can access a variable.

Others can borrow a variable mutably/immutably.

- The current thread borrows a global resource;

```rust
static TOTAL_MEMORY_CAP: usize = 0x1000;
fn thread_fn() {
    let cap = &TOTAL_MEMORY_CAP;
    println!("Memory capacity: 0x{:x}", cap);
}
```

The Rust borrow checker

Too restrictive if only the owner can access a variable.

Others can borrow a variable mutably/immutably.

- The current thread borrows a global resource;

```
static TOTAL_MEMORY_CAP: usize = 0x1000;
fn thread_fn() {
    let cap = &TOTAL_MEMORY_CAP;
    println!("Memory capacity: 0x{:x}", cap);
}
```

Quiz4: borrow it mutably?

The Rust borrow checker

Too restrictive if only the owner can access a variable.

Others can borrow a variable mutably/immutably.

- The current thread borrows a global resource;
- One variable borrows a field from another variable;

```
struct Foo {
    bar: u32,
}
impl Foo {
    fn borrow_bar(&mut self) -> &mut u32 {
        &mut self.bar
    }
}
struct Boo<'a> {
    bar: &'a mut u32,
}
```

```
fn main() {
    let mut foo = Foo { bar: 1 };
    let mut boo = Boo {
        bar: foo.borrow_bar(),
    };
    *boo.bar = 2;
    assert_eq!(foo.bar, 2);
}
```

The Rust borrow checker

Too restrictive if only the owner can access a variable.

Others can borrow a variable mutably/immutably.

Borrow checker checks:

- Alias XOR mutability (SWMR);
- Variables outlive borrows (lifetime).

The Rust borrow checker

Too restrictive if only the owner can access a variable.

Others can borrow a variable mutably/immutably.

Borrow checker checks:
- Alias XOR mutability (SWMR);
- Variables outlive borrows (lifetime).

Borrow checker is (always) deterministic!
Borrow checker is (sometimes) overzealous!
(you may understand why in the following section)

The Rust borrow checker–advanced lifetimes

You have:

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

The Rust borrow checker—advanced lifetimes

You have:

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

The compiler fills details into it so that it looks like:

```
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b is as big as we need this borrow to be
        // (just need to get to `println!`)
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // Temporary scope because we don't need the
            // &mut to last any longer.
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

The Rust borrow checker—advanced lifetimes

You have:

```rust
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x);
```

The compiler fills details into it so that it looks like:

```rust
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {
        // 'b is as big as we need this borrow to be
        // (just need to get to `println!`)
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {
            // Temporary scope because we don't need the
            // &mut to last any longer.
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

mutable borrow of `data`

while a borrow to `data` (x) lives for `‘b`

... and the borrow checker may not be so happy about it.

The Rust borrow checker–advanced lifetimes

Lifetimes are type parameters.

```
struct IntRef<'a> { // A type for lifetimes that lives for at least `'a`
    value: &'a i32,
}
let x = 42;
let y = IntRef { value: &x };
```

Goes to:

```
'a: {
    let x = 42;
    'b: {
        let y: IntRef<'b> = IntRef<'b> { value: &'b x };
    }
}
```

The Rust borrow checker–advanced lifetimes

Writing lifetimes bonds.

```rust
fn lock<'a, 'rcu>(page: &'a PtPage, _guard: &'rcu dyn InAtomicMode)
    -> PageTableGuard<'rcu, E, C>
    where 'a: 'rcu // `'a` is a subtype of `'rcu`, so `'a` outlives `'rcu`
    { /* ... */ }
```

The Rust borrow checker–advanced lifetimes

Writing lifetimes bonds.

```
fn lock<'a, 'rcu>(page: &'a PtPage, _guard: &'rcu dyn InAtomicMode)
    -> PageTableGuard<'rcu, E, C>
    where 'a: 'rcu // `'a` is a subtype of `'rcu`, so `'a` outlives `'rcu`
    { /* ... */ }
```

Wait wait wait wait wait, when should you write lifetimes bonds?

- Manually assigning lifetimes to pointers when encapsulating unsafe code;
- Perhaps write (linear programming) rules that are not statically checked in prior works yet?

The Rust borrow checker–advanced lifetimes

Quiz5: for the concrete lifetimes in the given code,

```
'a: {
    let x = 42;
        'b: {
        let y: &'b x = &'b x;
    }
}
```

which of the following holds?

1. ‘a: ‘b
2. ‘b: ‘a
3. Neither
4. Nondeterministic

Unsafe Rust

Most importantly, unsafe/safe interaction.

Unsafe Rust

Most importantly, unsafe/safe interaction.

Writing solely unsafe code is fine.

Unsafe Rust

Most importantly, unsafe/safe interaction.

Writing solely unsafe code is fine.
Mixing safe/unsafe will blow up all your safe code.

Unsafe Rust

A type is a set of valid values with a specific memory layout.

Breaking type safety with unsafe Rust? Easy.

```rust
struct IntRef<'a> {
    value: &'a usize,
}
impl<'a> IntRef<'a> {
    fn new(value: &'a usize) -> Self {
        IntRef { value }
    }
    fn read_value(&self) -> usize {
        *self.value
    }
}
```

```rust
fn main() {
    let x = 42;
    let int_ref = IntRef::new(&x);
    unsafe {
        (&int_ref as *const _ as *const usize)
            .cast_mut().write(100);
    }
    println!("{}", int_ref.read_value()); // ???
}
```

Unsafe Rust

When do you have to write unsafe?

Unsafe Rust

When do you have to write unsafe?

Unfortunately, a lot (when building systems).

For low-level functionalities:

- Implementing locks;
- Implementing data structures: trees, linked lists, even `Vec`;
- Inline assembly;
- Foreign Language Interface (FFI), etc.

For performance:

- Zero-copy (you have to cast between types via `core::mem::transmute`);
- Manual memory management (like RCU);
- Uninitialized memory, etc.

Unsafe Rust

When do you have to write unsafe?

Unfortunately, a lot (when building systems).

20% crates at crates.io uses unsafe code:
“We write unsafe code, so you don’t have to.”

But you are a systems researcher, you have to.

Unsafe Rust

How to write unsafe code:

When using unsafe functions, state why the requirements hold.

```
impl<'a> From<&'a [u8]> for VmReader<'a, Infallible> {
    fn from(slice: &'a [u8]) -> Self {
        // SAFETY:
        // - The validity for read accesses are met because the pointer is converted
        //   from an immutable reference that outlives the lifetime `'a`.
        // - The type, i.e., the `u8` slice, is plain-old-data.
        unsafe { Self::from_kernel_space(slice.as_ptr(), slice.len()) }
    }
}
```

When writing unsafe functions/traits, state what's the requirement to be safe.

```
impl<'a> VmReader<'a, Infallible> {
    /// Constructs a `VmReader` from a pointer and a length,
    /// which represents a memory range in kernel space.
    /// # Safety
    /// `ptr` must be [valid] for reads of `len` bytes during
    /// the entire lifetime `a`.
    /// [valid]: crate::mm::io#safety
    pub unsafe fn from_kernel_space(ptr: *const u8, len: usize) -> Self {
```

Unsafe Rust

Advanced safety topics:

- Type safety (coercions);
- Panic safety;
- Concurrency (`Send` & `Sync`);
- FFI/ABI.

Go read the nomicon. Discuss with me if can’t understand.

Rust in the lab

Let's scrutinize 🔍 this paper:

[OSDI'24] *IntOS: Persistent Embedded Operating System and Language Support for Multi-threaded Intermittent Computing*

(@Yonghao suggested to do it, so great!)

Rust in the lab

Try think about how Rust can be used to enforce these rules in Chaper 6:

Rule 1: Persistent objects should not be accessed (both write and read) outside the transaction and their update inside the transaction must be logged. Modifications on persistent objects outside transactions are untracked. Therefore

Rule 2: References/Pointers to persistent objects should not escape a transaction as a return value. Rule 2 further enforces Rule 1. Allowing the return of references would

Rule 3: Persistent objects should not contain references to volatile objects. Volatile objects are susceptible to data loss during power failures. Storing their references in persistent

Rule 4: System calls (excluding Locks) should only be made within transactions. There is, in theory, no fundamental restriction against using a system call outside a transaction

Rule 5: Locks should not be used inside transactions. A critical section, defined by locks, should be larger than a trans-

Enforcement IntOS employs Rust's robust type system to uphold the aforementioned rules, akin to [33] that statically prevents common persistent memory programming errors

Rust in the lab–Rule 1

Rule 1: Persistent objects should not be accessed (both write and read) outside the transaction and their update inside the transaction must be logged.

Rust in the lab–Rule 1

Rule 1: Persistent objects should not be accessed (both write and read) outside the transaction and their update inside the transaction must be logged.

Recall that transactions APIs are carried out by closures:

```
let _ = transaction::run(|j: JournalHandle, t: SyscallToken| {
    // ... body of the transaction
})
```

Rust in the lab–Rule 1

Rule 1: Persistent objects should not be accessed (both write and read) outside the transaction and their update inside the transaction must be logged.

Recall that transactions APIs are carried out by closures:

```
let _ = transaction::run(|j: JournalHandle, t: SyscallToken| {
    let stats = PBox::new(Stats::new(), j);
    // ... body of the transaction
})
```

The rule are enforced if:

- ✅ A `JournalHandle` can only be provided by a transaction;
- ✅ The `JournalHandle` is needed to create a persistent object (`PBox`);
- ✅ Updates to `PBox`es are logged.

Rust in the lab–Rule 2

Rule 2: References/Pointers to persistent objects should not escape a transaction as a return value.

Rust in the lab–Rule 2

Rule 2: References/Pointers to persistent objects should not escape a transaction as a return value.

```
let _ = transaction::run(|j: JournalHandle, t: SyscallToken| {
    let stats = PBox::new(Stats::new(), j);
    // need to use a journal handle to borrow persistent objects
    let stats_ref = stats.as_ref(j);
    // ... body of the transaction
})
```

Rust in the lab–Rule 2

Rule 2: References/Pointers to persistent objects should not escape a transaction as a return value.

```
let _ = transaction::run(|j: JournalHandle, t: SyscallToken| {
    let stats = PBox::new(Stats::new(), j);
    // need to use a journal handle to borrow persistent objects
    let stats_ref = stats.as_ref(j);
    // ... body of the transaction
})
```

What if:

```
let stats_ref = transaction::run(|j: JournalHandle, t: SyscallToken| {
    let stats = PBox::new(Stats::new(), j);
    // need to use a journal handle to borrow persistent objects
    let stats_ref = stats.as_ref(j);
    // ... body of the transaction
    return stats_ref; // return it out of the transaction?
})
```

Rust in the lab—Rule 2

Rule 2: References/Pointers to persistent objects should not escape a transaction as a return value.

```
let _ = transaction::run(|j: JournalHandle, t: SyscallToken| {
    let stats = PBox::new(Stats::new(), j);
    // need to use a journal handle to borrow persistent objects
    let stats_ref = stats.as_ref(j);
    // ... body of the transaction
})
```

What if:

Reference to `j` outlive `j`!

```
let stats_ref = transaction::run(|j: JournalHandle, t: SyscallToken| {
    let stats = PBox::new(Stats::new(), j);
    // need to use a journal handle to borrow persistent objects
    let stats_ref = stats.as_ref(j);
    // ... body of the transaction
    return stats_ref; // return it out of the transaction?
})
```

Rust in the lab–Rule 3

Rule 3: Persistent objects should not contain references to volatile objects.

Rust in the lab—Rule 3

Rule 3: Persistent objects should not contain references to volatile objects.

```rust
/// A trait for types that can be safely stored in persistent memory.
pub unsafe auto trait PSafe {}

// DON'T DO:
// unsafe impl<'a, T> PSafe for &'a T {}

impl<T: PSafe> PBox<T> {
    pub fn new(x: T, t: SyscallToken) -> Self {
        todo!()
    }
}
```

Rust in the lab–Rule 4

Rule 4: System calls (excluding Locks) should only be made within transactions.

Rust in the lab–Rule 4

Rule 4: System calls (excluding Locks) should only be made within transactions.

```
let _ = transaction::run(|j: JournalHandle, t: SyscallToken| {
    // need a system call token to do system calls
    let q = sys_queue_create::<Result>(Q_SZ, t).unwrap();
    // ... body of the transaction
})
```

Rust in the lab–Rule 5

Rule 5: Locks should not be used inside transactions.

Rust in the lab–Rule 5

Rule 5: Locks should not be used inside transactions.

Enforcement INTOS employs Rust's robust type system to uphold the aforementioned rules, akin to [33] that statically prevents common persistent memory programming errors

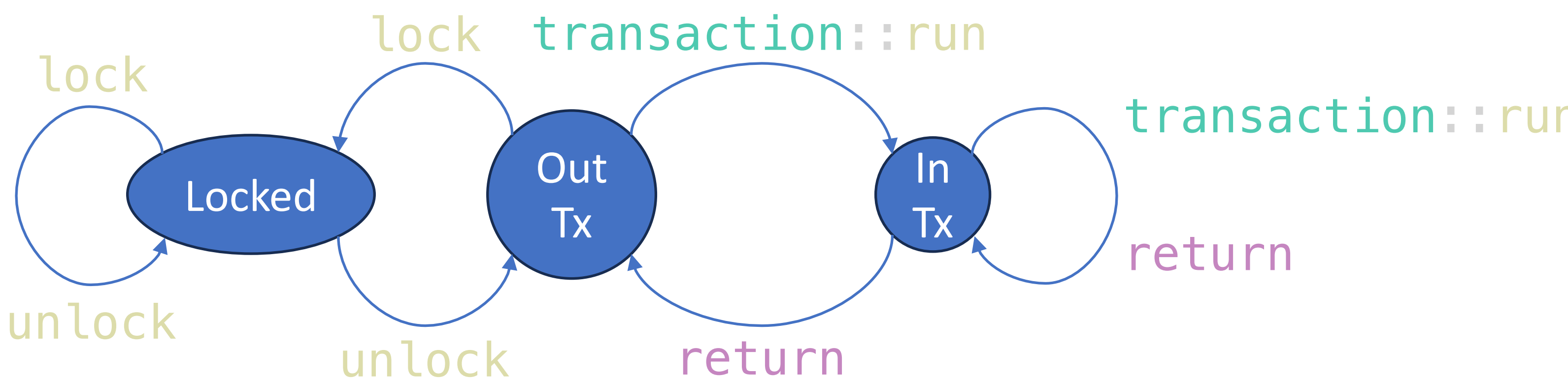
https://github.com/yiluwusbu/IntOS/blob/master/src/syscalls.rs#L502

Fraud…?

Rust in the lab—Rule 5

Rule 5: Locks should not be used inside transactions.

But now you and I know how to do it.



Thanks for listening

Lunch session @Junyang