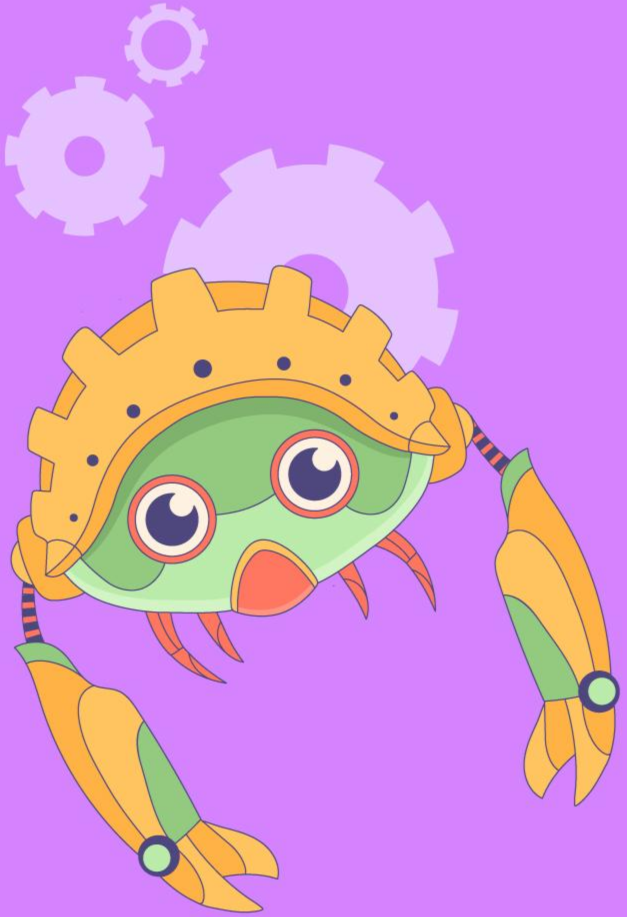
Alice Ryhl

Software Engineer @ Google

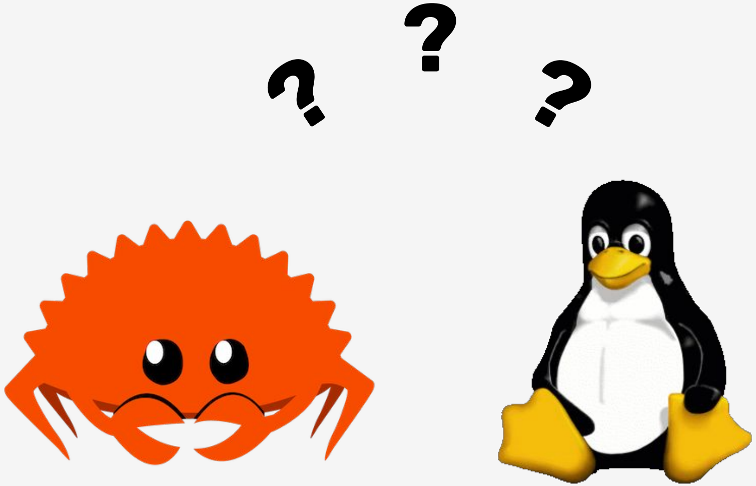
Rust in the Linux Kernel



INTRODUCTION

- Rust for Linux was started in 2020.
- The first Rust code was merged into Linux in December 2022.
- There are several Rust drivers, but none have been merged yet.

Why use Rust in the Kernel?



Why use Rust in the Kernel?

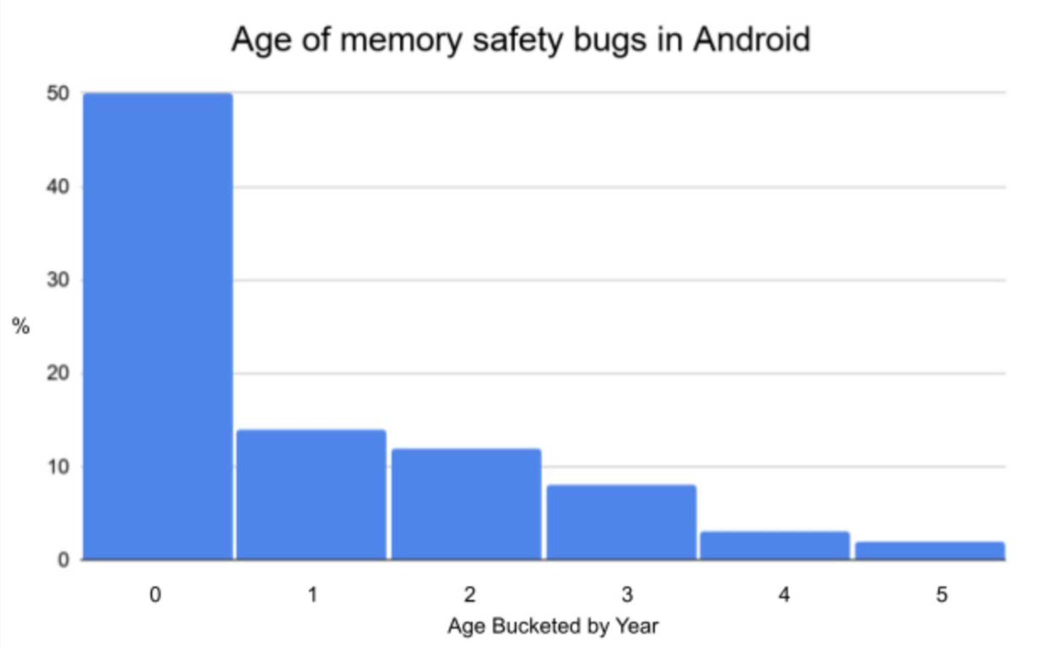
> If you have a very large (millions of lines of code) codebase, written in a memory-unsafe programming language (such as C or C++), you can expect at least 65% of your security vulnerabilities to be caused by memory unsafety.
>
> — Alex Gaynor

Why use Rust in the Kernel?

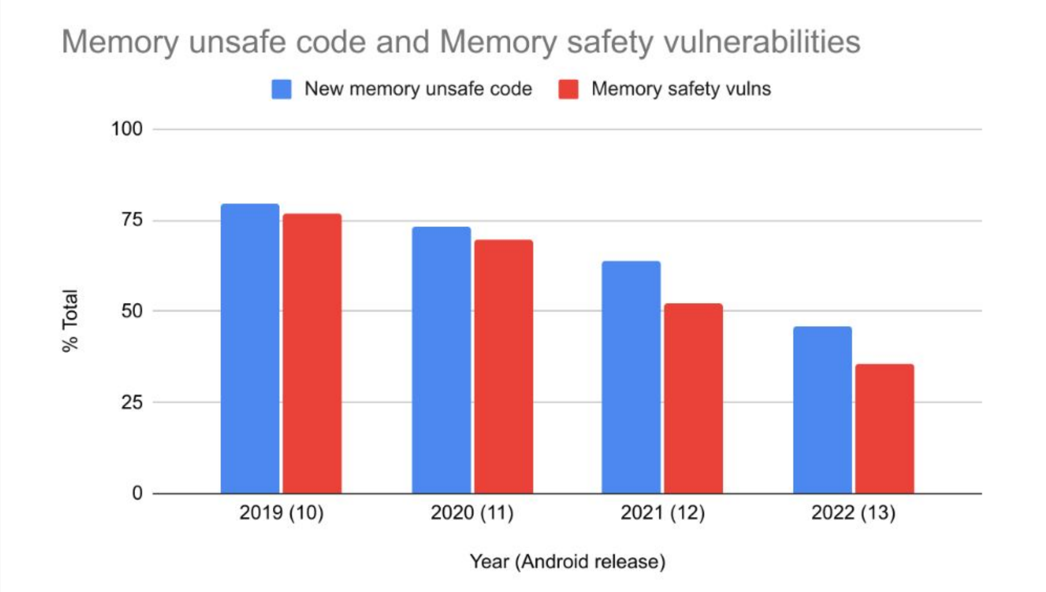
Also true for the Kernel:

“65% of CVEs behind the last six months of Ubuntu security updates to the Linux kernel have been memory unsafety.”

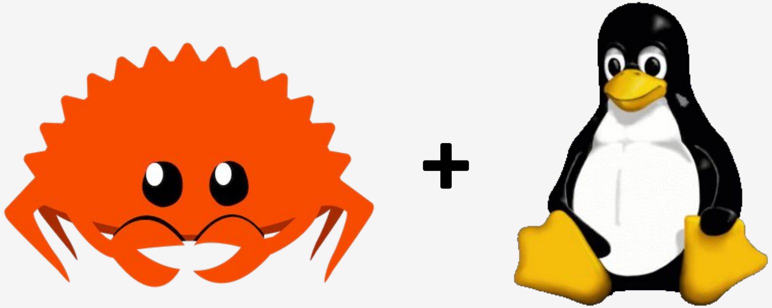
Most vulnerabilities are in new code



Empirical evidence that Rust makes a difference



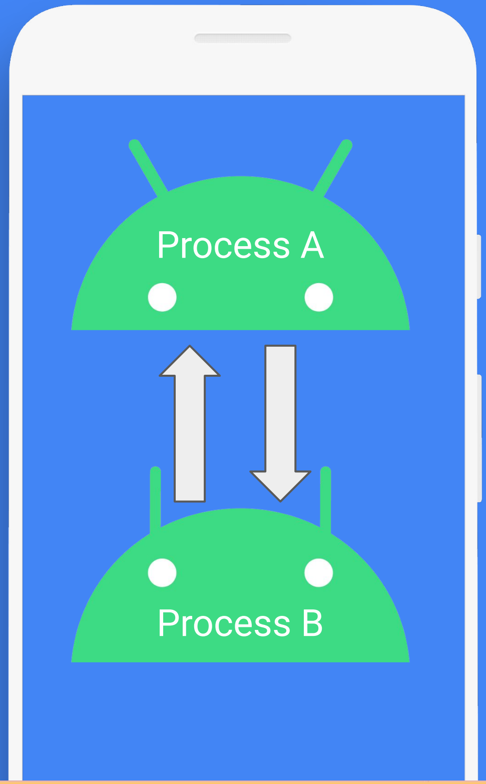
Rust projects in the Kernel



Rust projects in the Kernel

- Android Binder driver
- PuzzleFS and TarFS
- Asahi Linux GPU driver
- NVMe and Null block driver
- Asix PHY ethernet driver

Android Binder driver



Challenges that Binder faces

High complexity	Accumulated technical debt	Security issues

High complexity makes it difficult to resolve tech debt without causing security issues.

Security issues in Binder

1

High vulnerability density

Binder's density is around 3.1 vulnerabilities per kLOC.

2

Not getting better

Binder has averaged ~3 high/critical severity vulnerabilities per year over the past 6 years.

3

Risk is not theoretical

We are aware of exploits for about half of the vulnerabilities in Binder.

4

Security critical

Even Android's most de-privileged sandboxes have direct access to Binder.

Rust Binder

Feature parity

Implements all features in C Binder.

(except for some debugging facilities)

Passes tests

Passes all Binder tests in AOSP.

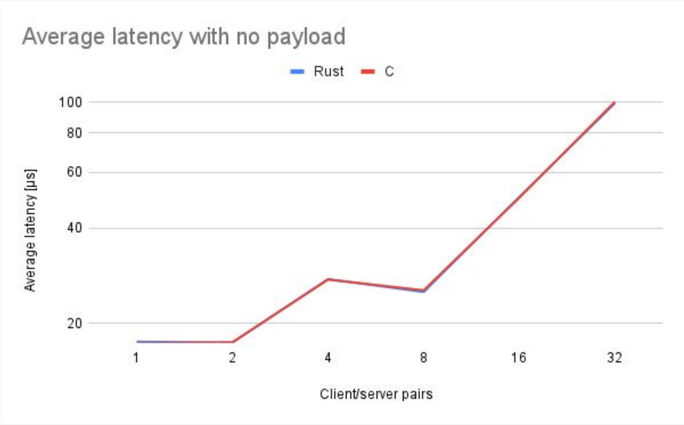
Can boot a device and run a variety of apps without issues.

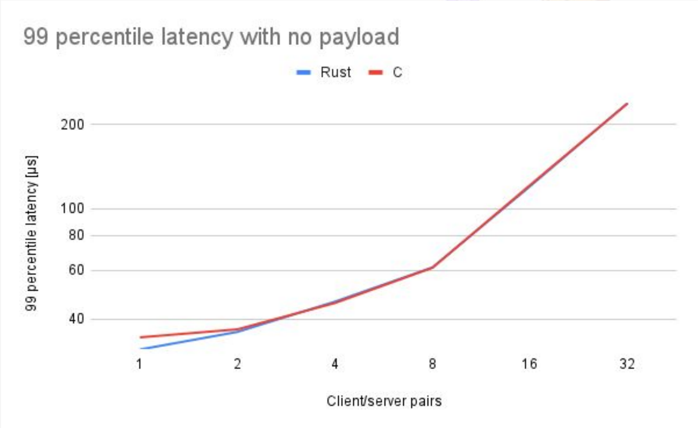
Promising performance

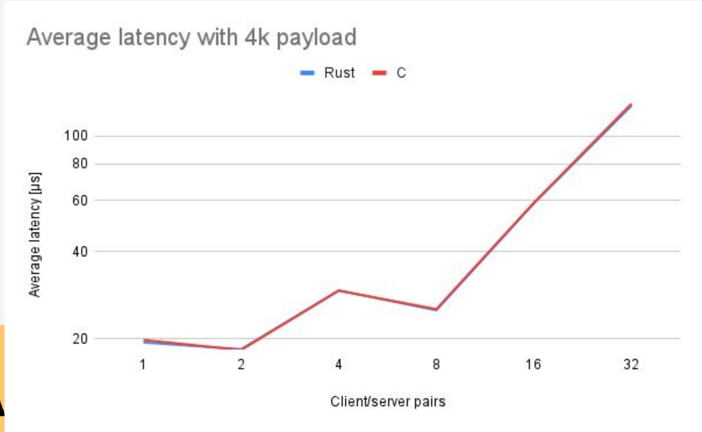
On a simple benchmark, drivers have similar performance.

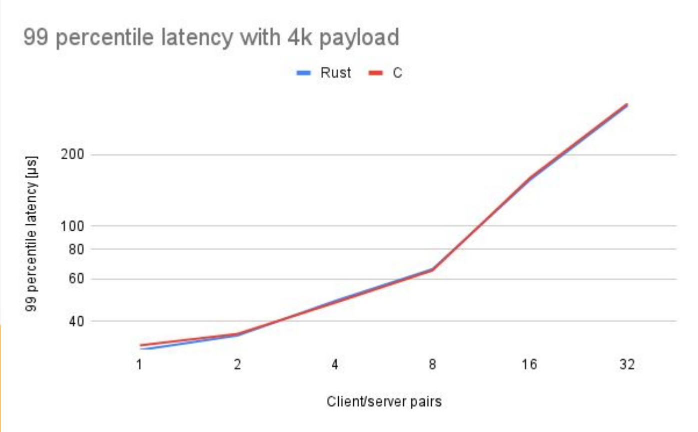
Still a lot of work to do.

Rust Binder benchmarks

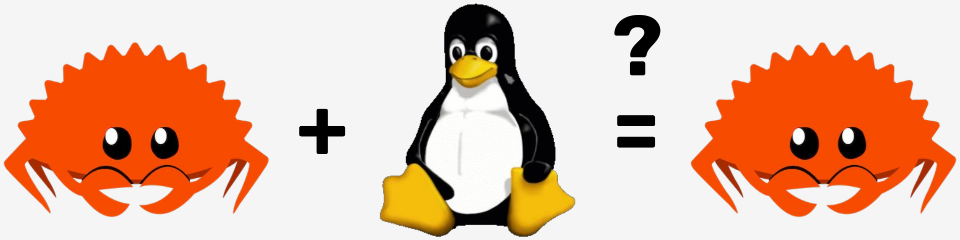




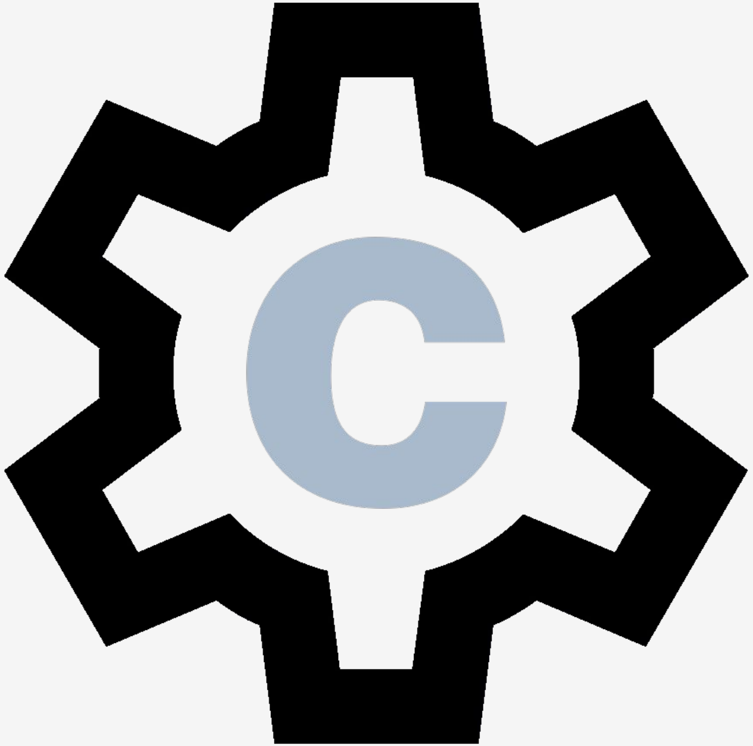




How is Kernel Rust different?



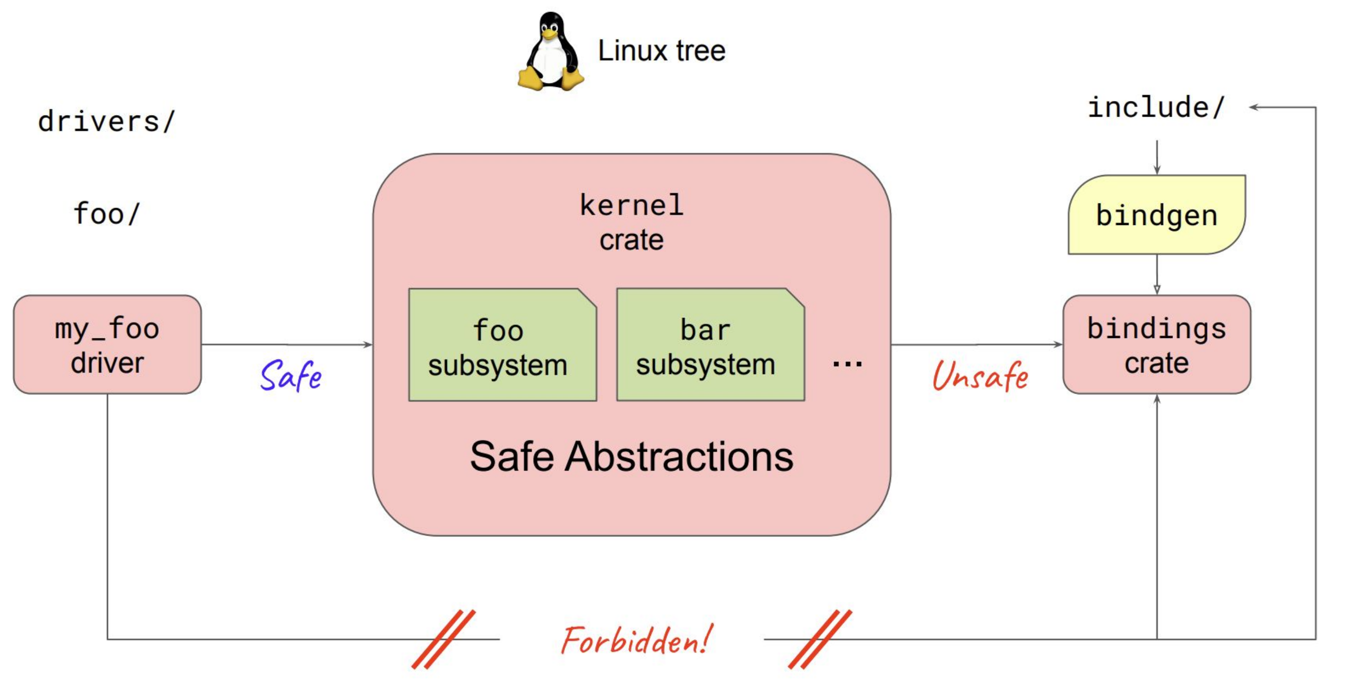
Wrapping C



Wrapping C

- Kernel drivers need to access many different C apis.
- For now, the driver author must write the C wrapper.
- Requires a good understanding of unsafe Rust.

Wrapping C


Linux tree
drivers/
foo/
include/
kernel
crate
bindgen
my_foo
driver
Safe
foo
subsystem
bar
subsystem
...
Unsafe
bindings
crate
Safe Abstractions
Forbidden!

Workqueue example in Binder

```
impl workqueue::WorkItem for Process {
    type Pointer = Arc<Process>;

    fn run(me: Arc<Self>) {
        let defer;
        {
            let mut inner = me.inner.lock();
            defer = inner.defer work;
            inner.defer_work = 0;
        }

        if defer & PROC DEFER FLUSH != 0 {
            me.deferred_flush();
        }
        if defer & PROC DEFER RELEASE != 0 {
            me.deferred_release();
        }
    }
}
```

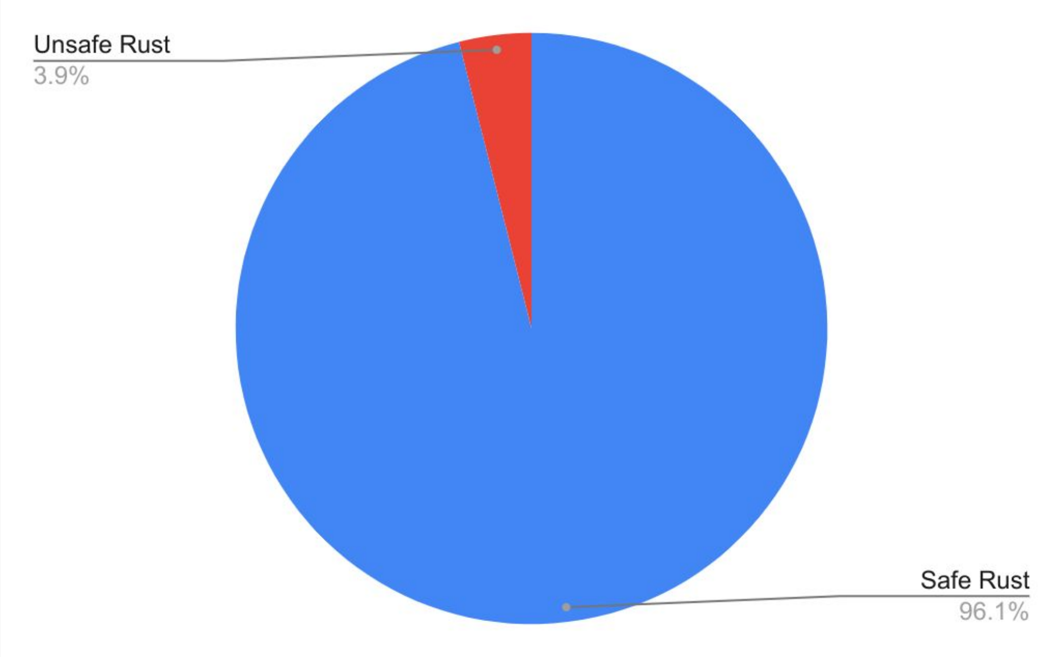
No unsafe needed in Binder!

C wrappers needed by Binder

- Collections: Linked List, red/black tree, xarray.
- Synchronization: Mutex, SpinLock, CondVar.
- Memory management: Page manipulation.
- Files: Manipulation of open files.
- Workqueue: Execute code in the background.

Unsafe code in Binder

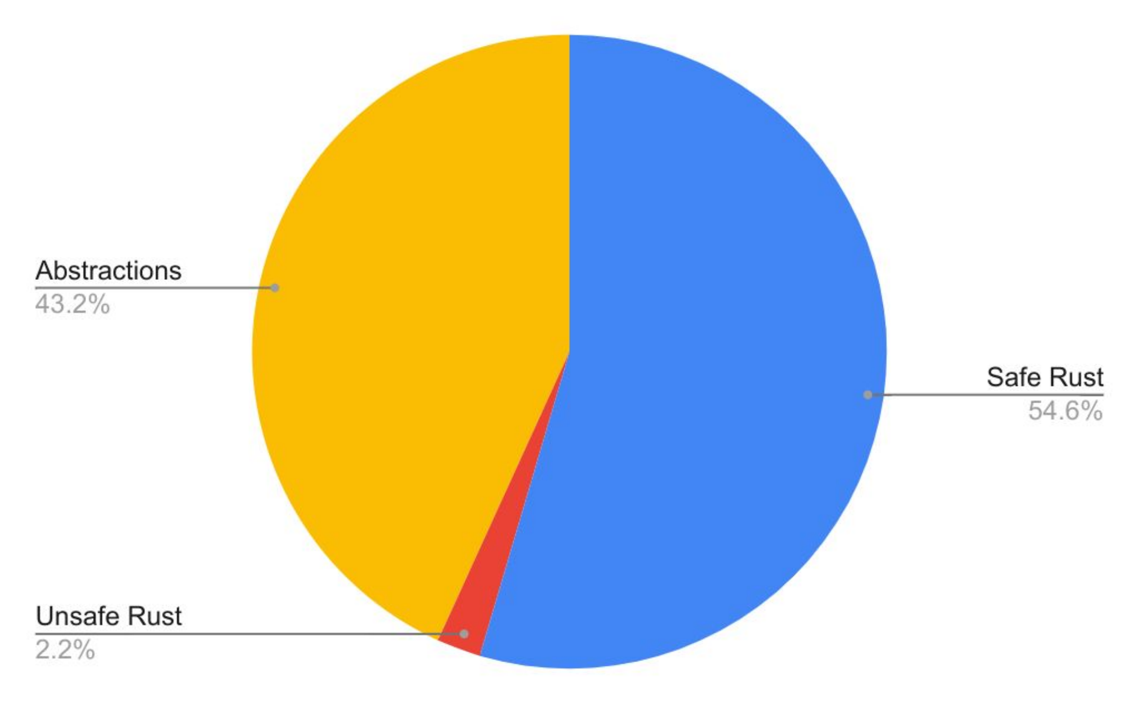
- Safe Rust in Binder
- Unsafe Rust in Binder



What about C wrappers?

You only have to get them right once, across all drivers.

- Safe Rust in Binder
- Unsafe Rust in Binder
- Abstractions



Fallible allocations

We can’t just crash if we run out of memory!

```
Box::try_new(value)?
```

Fallible allocations

- **Must be careful to clean up on allocation failure.**
- To make operations infallible, we allocate memory before we need it.
- Linked list > Vec

Fallible allocations

- Must be careful to clean up on allocation failure.
- **To make operations infallible, we allocate memory before we need it.**

CREATE → **DESTROY**

Fallible
Allocate here

Not fallible!
Use allocation here

Fallible allocations example in Binder

```
struct Allocation<T> {
    is_oneway: bool,
    pid: Pid,
    data: Option<T>,
    free_res: RBTreeNodeReservation<FreeKey, ()>,
}
```

Fallible allocations

- Must be careful to clean up on allocation failure.
- To make operations infallible, we allocate memory before we need it.
- **Linked list > Vec**

You can't always sleep

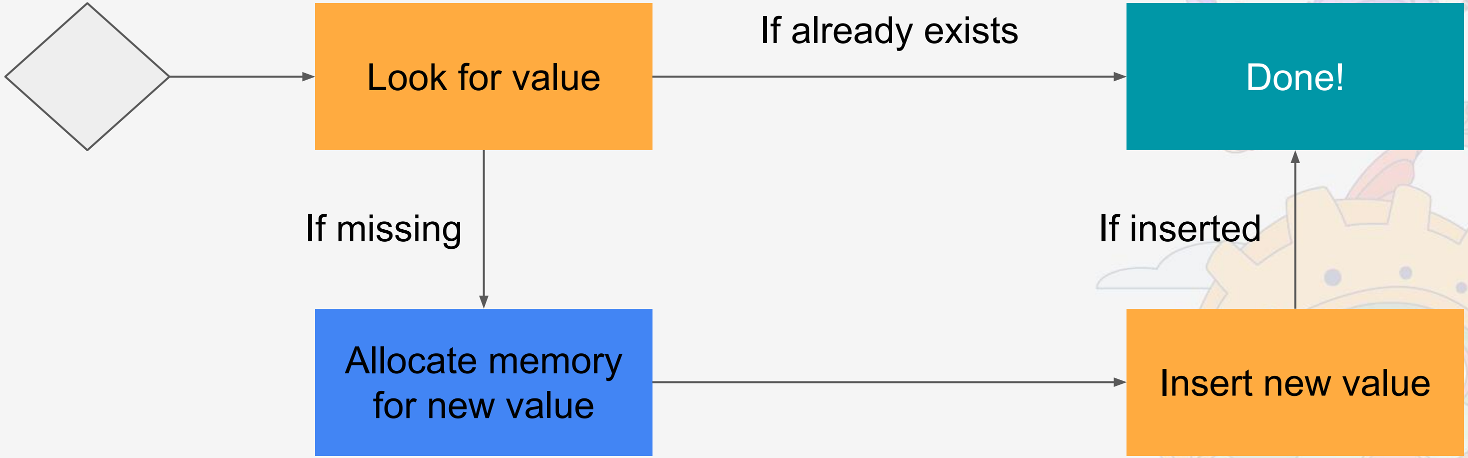
And allocating memory might sleep!

You can't always sleep

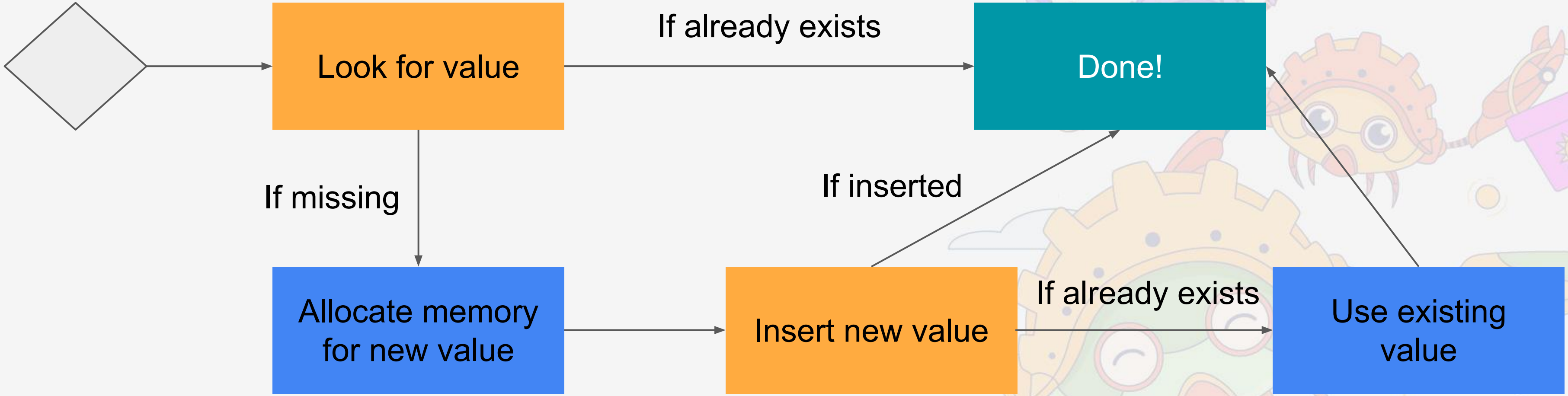
Two types of mutex:

- Mutex
  - lock() will sleep, allows sleeping
- Spinlock
  - lock() will not sleep, does not allow sleeping

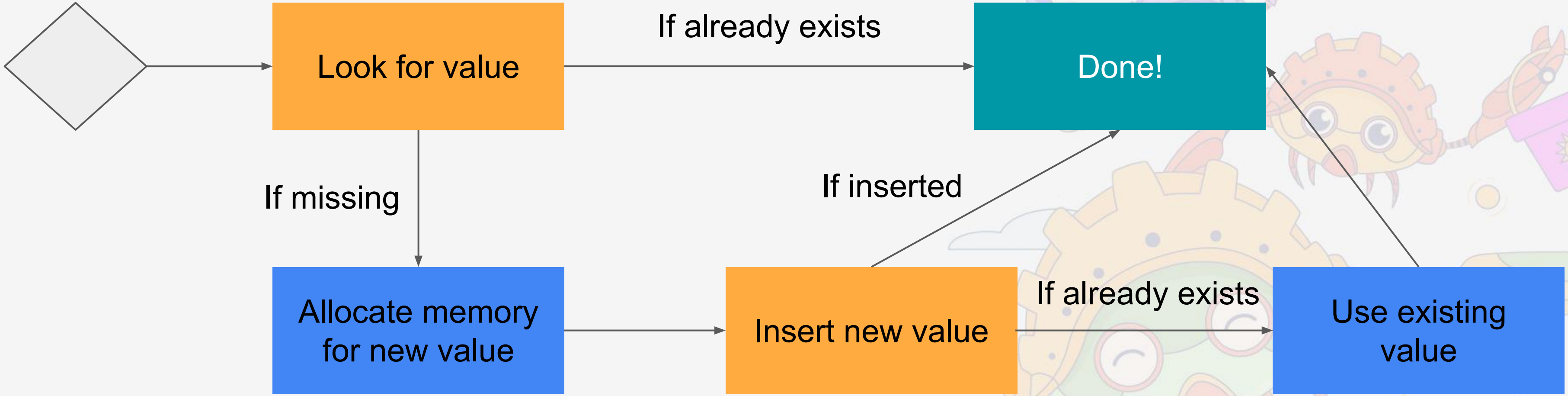
Get or insert new



Get or insert new



Get or insert new



In Binder, I had to write this many times!

You can't always sleep example in Binder

```
fn get_thread(self: ArcBorrow<'_, Self>, id: i32) -> Result<Arc<Thread>> {
    {
        let inner = self.inner.lock();
        if let Some(thread) = inner.threads.get(&id) {
            return Ok(thread.clone());
        }
    }

    // Allocate a new `Thread` without holding any locks.
    let ta = Thread::new(id, self.into())?;
    let node = RBTree::try_allocate_node(id, ta.clone())?;

    let mut inner = self.inner.lock();

    // Recheck. It's possible the thread was created while we were not holding the lock.
    if let Some(thread) = inner.threads.get(&id) {
        return Ok(thread.clone());
    }

    inner.threads.insert(node);
    Ok(ta)
}
```

You can’t always sleep example in Binder

```
fn get_thread(self: ArcBorrow<'_, Self>, id: i32) -> Result<Arc<Thread>> {
    {
        let inner = self.inner.lock();
        if let Some(thread) = inner.threads.get(&id) {
            return Ok(thread.clone());
        }
    }

    // Allocate a new `Thread` without holding any locks.
    let ta = Thread::new(id, self.into())?;
    let node = RBTree::try_allocate_node(id, ta.clone())?;

    let mut inner = self.inner.lock();

    // Recheck. It's possible the thread was created while we were not holding the lock.
    if let Some(thread) = inner.threads.get(&id) {
        return Ok(thread.clone());
    }

    inner.threads.insert(node);
    Ok(ta)
}
```

You can't always sleep example in Binder

```
fn get_thread(self: ArcBorrow<'_, Self>, id: i32) -> Result<Arc<Thread>> {
    {
        let inner = self.inner.lock();
        if let Some(thread) = inner.threads.get(&id) {
            return Ok(thread.clone());
        }
    }

    // Allocate a new `Thread` without holding any locks.
    let ta = Thread::new(id, self.into())?;
    let node = RBTree::try_allocate_node(id, ta.clone())?;

    let mut inner = self.inner.lock();

    // Recheck. It's possible the thread was created while we were not holding the lock.
    if let Some(thread) = inner.threads.get(&id) {
        return Ok(thread.clone());
    }

    inner.threads.insert(node);
    Ok(ta)
}
```

You can't always sleep example in Binder

```
fn get_thread(self: ArcBorrow<'_, Self>, id: i32) -> Result<Arc<Thread>> {
    {
        let inner = self.inner.lock();
        if let Some(thread) = inner.threads.get(&id) {
            return Ok(thread.clone());
        }
    }

    // Allocate a new `Thread` without holding any locks.
    let ta = Thread::new(id, self.into())?;
    let node = RBTree::try_allocate_node(id, ta.clone())?;

    let mut inner = self.inner.lock();

    // Recheck. It's possible the thread was created while we were not holding the lock.
    if let Some(thread) = inner.threads.get(&id) {
        return Ok(thread.clone());
    }

    inner.threads.insert(node);
    Ok(ta)
}
```

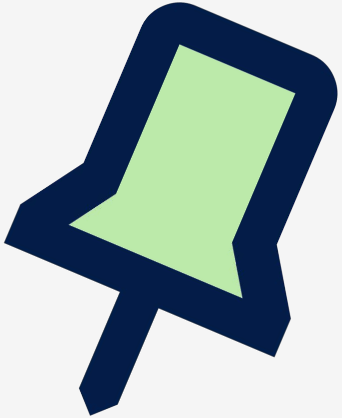
You can't always sleep example in Binder

```
fn get_thread(self: ArcBorrow<'_, Self>, id: i32) -> Result<Arc<Thread>> {
    {
        let inner = self.inner.lock();
        if let Some(thread) = inner.threads.get(&id) {
            return Ok(thread.clone());
        }
    }

    // Allocate a new `Thread` without holding any locks.
    let ta = Thread::new(id, self.into())?;
    let node = RBTree::try_allocate_node(id, ta.clone())?;

    let mut inner = self.inner.lock();

    // Recheck. It's possible the thread was created while we were not holding the lock.
    if let Some(thread) = inner.threads.get(&id) {
        return Ok(thread.clone());
    }

    inner.threads.insert(node);
    Ok(ta)
}
```

You can’t always sleep

We have a custom linter for catching sleeps in atomic contexts.

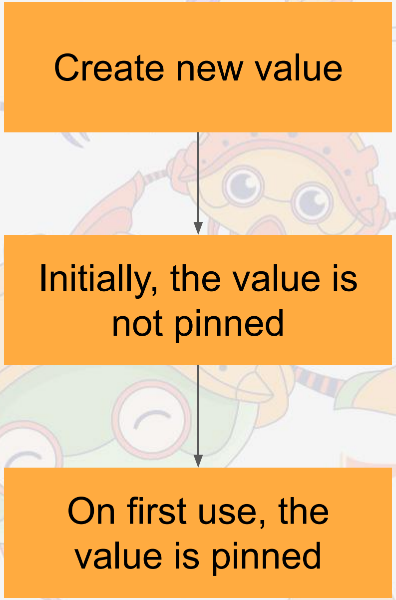
Pinning is not enough



Pinning is not enough

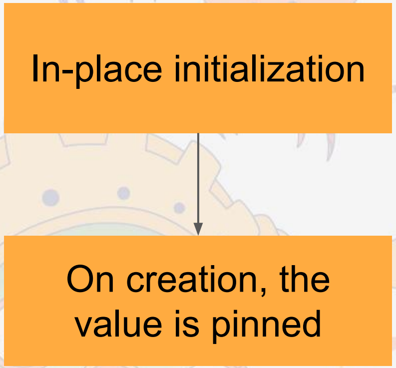
Normal pinning:

- Before first use, value may move around.
- Values are pinned on first use.



Pinning is not enough

- We use C types defined by the Kernel
- Those C types require the value to be pinned immediately.
- Done using special macro.



Pin-init example in Binder

```
Arc::pin_init(pin_init!(Thread {
    id,
    process,
    inner <- kernel::new_spinlock!(ThreadInner::new()),
    work_condvar <- kernel::new_poll_condvar!(),
    links <- ListLinks::new(),
    links_track <- AtomicListArcTracker::new(),
}))
```

Pin-init example in Binder

```
Arc::pin_init(pin_init!(Thread {
    id,
    process,
    inner <- kernel::new_spinlock!(ThreadInner::new()),
    work_condvar <- kernel::new_poll_condvar!(),
    links <- ListLinks::new(),
    links_track <- AtomicListArcTracker::new(),
}))
```

Pin-init example in Binder

```
Arc::pin_init(pin_init!(Thread {
    id,
    process,
    inner <- kernel::new_spinlock!(ThreadInner::new()),
    work_condvar <- kernel::new_poll_condvar!(),
    links <- ListLinks::new(),
    links_track <- AtomicListArcTracker::new(),
}))
```

Unstable compiler features

```
#![feature(allocator_api)]
#![feature(coerce_unsized)]
#![feature(dispatch_from_dyn)]
#![feature(new_uninit)]
#![feature(offset_of)]
#![feature(ptr_metadata)]
#![feature(receiver_trait)]
#![feature(unsize)]
```

Custom Arc

You cannot implement your own Arc in stable Rust.

Custom Arc

Why use a custom Arc?

- Uses the Kernel's refcounting logic.
    - **Don't abort on overflow!**
- No weak references.
- All Arcs are pinned.

Unstable is also needed for:

- Custom Arc
- Fallible allocations
- Const evaluation
- offset_of! macro

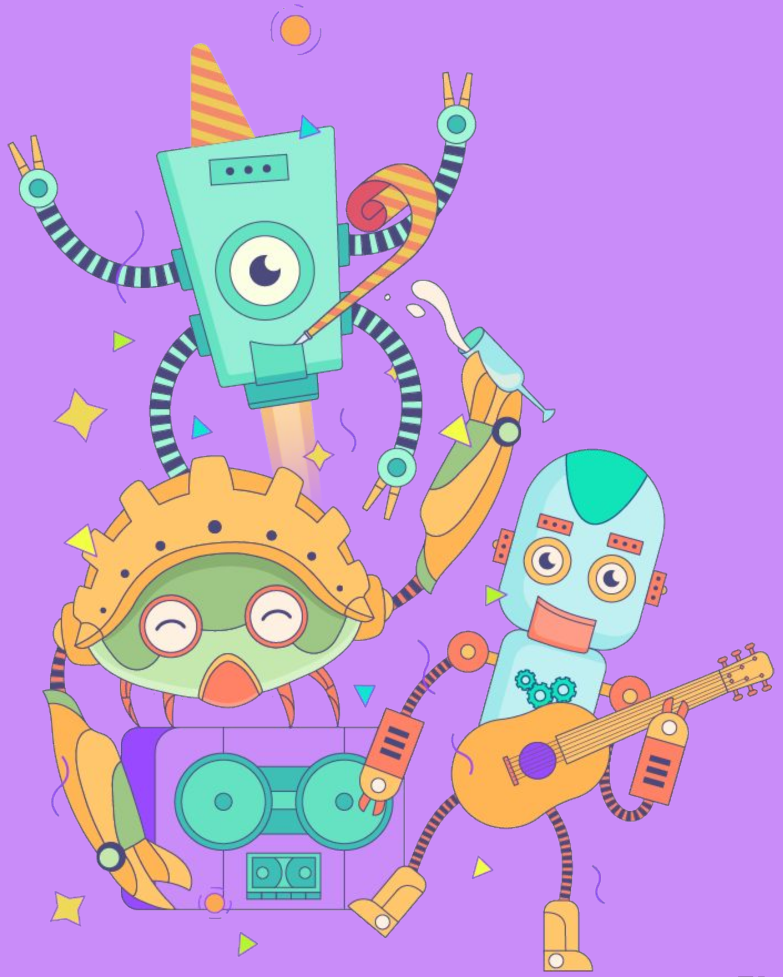
Unstable compiler features

Unstable features is a problem for all embedded Rust code.

Call to action

Let’s get embedded Rust off nightly Rust.

Thank you for listening



Alice Ryhl

alice@ryhl.io