Welcome to Rust for Linux

The Linux Kernel is currently working to add support for using the Rust language.

This involves considerations on both sides of the equation - Linux and Rust.

Rust must conform to the non-negotiable requirements Linux imposes on compile- and run-time behavior of its core and modules. Meanwhile, Linux must expose its existing functionality in ways that Rust can access as efficiently and safely as possible.

We'll look at the general background, get our hands dirty with some existing Rust code in Linux, and then explore particular integration challenges, both resolved and ongoing.

Schedule

Including 10 minute breaks, this session should take about 0 minutes. It contains:

Segment	Duration

Basic Interoperation Requirements

To use Rust code in Linux, we can start by comparing this situation with C/Rust interop in userspace.

In userspace, the most common setup is to use Cargo to compile our Rust and later integrate into a C build system if needed. Meanwhile, the Linux Kernel compiles its C code with its custom Kbuild build system. In Rust for Linux, the kernel build system invokes the Rust compiler directly, without Cargo.

Unlike typical usage of Rust in userspace, which makes use of the rust standard library through the `std` crate, Rust in the kernel does not run atop an operating system, so kernel Rust will have to eschew the standard library.

Much code in the kernel is compiled into kernel modules rather than as part of the core kernel. To write kernel modules in Rust we'll need to be able to match the ABI of kernel modules.

To reap the benefits of Rust, we want to be able to write as much code as possible in safe Rust. This means that we want safe wrappers for as much kernel functionality as possible.

When writing these wrappers, we'll need to refer to the data types of values passed to and from existing kernel functions in C. Unlike userspace C, the kernel uses its own set of primitive types rather than those provided by the C standard. We'll have to map back and forth between those kernel types and compatible Rust ones when doing foreign calls.

Finally, even the core Rust library assumes a basic level of functionality that includes some costly operations (e.g. unicode processing) for which the kernel does not want to pay implementation costs. To use Rust in the kernel we'll need a way to disable this functionality.

Building Kernel Modules

To build kernel modules in Rust, we need to build `.ko` shared objects that link against the rest of the kernel.

In C, kernel modules use the `module_init` and `module_exit` macros to specify how to initialize and deinitialize the module. For Rust, we'll need some equivalent of these macros. Ultimately, these macros specify the values of two fields in a `struct this_module` which is placed in the `.gnu.linkonce.this_module` section of the kernel module object file.

We can achieve this in Rust with by defining an equivalent struct as a `static` item and using the `#[unsafe(link_section = ".gnu.linkonce.this_module")]` attribute. The `.init` and `.exit` fields of our struct will need to be pointers to the appropriate functions, which leads to our next question: how do we define Rust functions with the types and calling convention expected here?

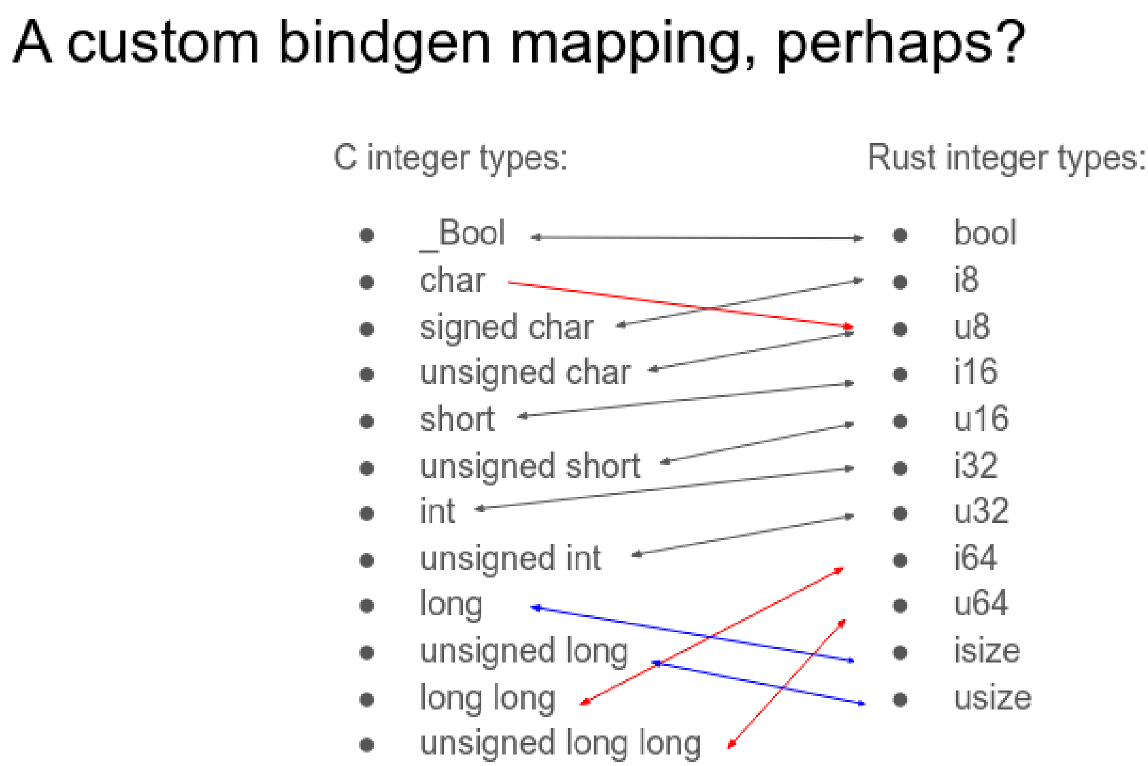
In practice, Rust for Linux has a convenient and safe wrapper around this pattern which we'll see when we look at a real-world Rust kernel module.

Type Mapping

The kernel uses slightly different conventions for its types than other C code. As such, the normal use of `bindgen` to generate bindings can introduce some problems when applied to the kernel.

The explicit `{u,s}{8,16,32,64}` types map in the obvious way to Rust `{u,i}{8,16,32,64}` types, but the potential signedness of `char` as well as the kernel's requirement that `long` s can hold pointers introduce some deviations from the expectations of C as implemented by bindgen.

As such, an alternative bindgen mapping is used for the kernel[1]:



1 https://lwn.net/Articles/993163/

Bindings and Safe Interfaces

`bindgen` is used to generate low-level, unsafe bindings for C interfaces.

But to reap the benefits of Rust, we want to use safe, foolproof interfaces to unsafe functionality.

Subsystems are expected to implement safe interfaces on top of the low-level generated bindings. These safe interfaces are exposed as top-level modules within the `kernel` crate. The top-level `bindings` module holds the unsafe `bindgen`-generated bindings, which are generated from the C headers included by `rust/bindings/bindings_helper.h`.

In Rust for Linux, unsafe `bindgen`-generated bindings should not be used outside the `kernel` crate. Drivers and other subsystems will make use of the safe abstractions from this crate.

Only a subset of Linux subsystems currently have such abstractions.

It's worth browsing the list of modules exposed by the `kernel` crate to see what exists currently. Many of these subsystems have only partial bindings based on the needs of consumers so far.

Adding a Module

To add a module for some subsystem, first its header must be added to `bindings_helper.h`. It may be necessary to write some custom code to wrap macros or `inline` functions that are not automatically handled by `bindgen`; this code lives in the `rust/helpers/` directory.

Then we need to write a safe abstraction using these bindings and exposing them to the rest of kernel Rust.

Some commits from work-in-progress bindings and abstractions can provide an idea of what it looks like to expose new kernel functionality:

- GPIO Consumer: https://github.com/Fabo/linux/commit/fecb4bd73f06bb2cac8e16aca7ef0e2f1b6acb50
- Regmap: https://github.com/Fabo/linux/commit/ec0b740ac5ab299e4c86011a0002919e5bbe5c2d
- I2C: https://github.com/Fabo/linux/commit/70ed30fcdf8ec62fa91485c3c0a161a9d0194668

Guidelines for Abstractions

Abstractions may not be perfectly safe, but should try to be as safe as possible. Unsafe functionality exposed should have its safety conditions documented so that users have guidance on how to use the functionality and justify such use.

Abstractions should also attempt to present relatively idiomatic Rust in their interfaces:

- Follow Rust naming/capitalization conventions while remaining unsurprising to kernel developers.

- Use RAII instead of manual resource management where possible.

- Avoid raw pointers to bound kernel objects in favor of safer, more limited interfaces.

  When exposing types from generated bindings, code should make use of the `Opaque<T>` type along with native Rust references and the `ARef<T>` type for types that are inherently reference-counted. This type links types' built-in reference count operations to the `Clone` and `Drop` traits.

Submitting the cyclic dependency

We already know that drivers should not use unsafe bindings directly. But subsystem maintainers may balk if they see patches submitted that add Rust abstractions without motivation or consumers. But drivers and subsystem abstractions may have to be submitted separately to different maintainers due to the distributed nature of Linux development.

So how should a developer submit a driver that requires bindings/abstractions for a subsystem not yet exposed to Rust?

There are two main approaches[1]:

1. Submit the driver as an RFC before submitting the abstractions it relies upon while referencing the RFC as a potential consumer.
2. Submit a stub driver and fill out non-stub functionality as subsystem abstractions land.

[1] https://rust-for-linux.zulipchat.com/#narrow/channel/288089-General/topic/Upstreaming.20a.20driver.20with.20unsave.20C.20API.20calls.3F/near/471677707

Removing Bloat

Rust for Linux makes use of `libcore` to avoid reimplementing all functionality of the Rust standard library. But even `libcore` has some functionality built-in that is not portable to all targets the kernel would like to support or that is not necessary for the kernel while occupying valuable code space.

This includes[1]:

- Support for math with 128-bit integers
- String formatting for floating-point numbers
- Unicode support for strings

Work is ongoing to make these features optional. In the meantime, the `libcore` used by Rust for Linux is larger and less portable than it could be.

[1] https://github.com/Rust-for-Linux/linux/issues/514

Hands-on With Kernel Rust

We’ve talked about the general requirements for using Rust in the Linux kernel.

Now let’s dig into the code as it stands and see how to work with the present state of Rust for Linux.

Rust for Linux

First, we want a checkout of Linux with Rust support. The basics have been upstream since

Then, we can follow the instructions from the Rust for Linux quick-start guide:

1. Install a Rust toolchain, including standard library sources:

```
rustup override set $(scripts/min-tool-version.sh rustc)
rustup component add rust-src rustfmt clippy
```

2. install bindgen

```
cargo install --locked --version $(scripts/min-tool-version.sh
bindgen) bindgen
```

2. Enable `CONFIG_RUST` in your kernel build configuration.

3. When building the kernel, use an LLVM toolchain:

```
make LLVM=1
```

`rust-analyzer` Setup

The `rust-analyzer` LSP server provides IDE support for working with Rust.

First, we install rust-analyzer normally:

```
rustup component add rust-analyzer
```

To use it with Rust for Linux, we need to generate a configuration file for `rust-analyzer`[1]:

```
make -C .../linux-with-rust-support M=$PWD rust-analyzer
```

Then, opening our editor in the directory where the `rust-project.json` file was created should run the language server with the appropriate settings.

[1] https://github.com/Rust-for-Linux/linux/blob/rust/Documentation/rust/quick-start.rst#rust-analyzer

Macros

The `kernel` crate exposes some kernel functionality through macros, and provides other macros to facilitate definitions that follow kernel patterns.

Printing macros

The kernel provides `pr_info!` and `dev_info!`, which correspond to the identically-named kernel macros in C. These support string formatting compatible with Rust's `std::print!`.

```
pr_info!("hello {}\n", "there");
```

Conditional Compilation

In C, conditional compilation is done with the preprocessor using `#if` / `#ifdef`.

Rust in the kernel may want to perform conditional compilation (which is done with `#[cfg]` attributes in Rust) based on the same `CONFIG_FOO` macros as C code might consider.

The kernel build system exports these as `cfg` s, so they can be used as shown below[1]:

```
#[cfg(CONFIG_X)]       // Enabled               (`y` or `m`)
#[cfg(CONFIG_X="y")]   // Enabled as a built-in (`y`)
#[cfg(CONFIG_X="m")]   // Enabled as a module   (`m`)
#[cfg(not(CONFIG_X))]  // Disabled
```

Kernel Vtables

The kernel has slightly different requirements for its vtables than Rust traits provide. For kernel vtables, unimplemented functions are represented by `NULL` function pointers, while Rust traits always implement all methods.

This mismatch is resolved by providing Rust with the `vtable!` attribute macro.

This macro is placed above trait definitions and impls and provides constant `bool HAS_METHODNAME` members for each method.

The `module!` macro

Kernel modules are defined with the `macro!` module, which we'll examine on its own.

[1] https://docs.kernel.org/rust/general-information.html#conditional-compilation

A Rust Kernel Module

A minimal Rust kernel module looks like the below (from `samples/rust/rust_minimal.rs` in the Rust for Linux tree):

```rust
use kernel::prelude::*;

module! {
    type: RustMinimal,
    name: "rust_minimal",
    author: "Rust for Linux Contributors",
    description: "Rust minimal sample",
    license: "GPL",
}

struct RustMinimal {
    numbers: KVec<i32>,
}

impl kernel::Module for RustMinimal {
    fn init(_module: &'static ThisModule) -> Result<Self> {
        pr_info!("Rust minimal sample (init)\n");
        pr_info!("Am I built-in? {}\n", !cfg!(MODULE));

        let mut numbers = KVec::new();
        numbers.push(72, GFP_KERNEL)?;
        numbers.push(108, GFP_KERNEL)?;
        numbers.push(200, GFP_KERNEL)?;

        Ok(RustMinimal { numbers })
    }
}

impl Drop for RustMinimal {
    fn drop(&mut self) {
        pr_info!("My numbers are {:?}\n", self.numbers);
        pr_info!("Rust minimal sample (exit)\n");
    }
}
```

We’ll examine each part of the module definition in the following slides.

▼ *Speaker Notes*

It is also possible to build Rust kernel modules out-of-tree.

The `module!` Macro

A kernel module itself is declared with the `module!` macro.

Here we specify the type for the module, upon which we will implement the `kernel::Module` trait, as well as metadata like the module's name and description.

```
module! {
    type: RustMinimal,
    name: "rust_minimal",
    author: "Rust for Linux Contributors",
    description: "Rust minimal sample",
    license: "GPL",
}

struct RustMinimal;
```

Module Setup and Teardown

Our module implements the `kernel::Module` trait to specify its entrypoint and perform any necessary set-up:

```
pub trait Module: Sized + Sync {
    fn init(name: &'static CStr, module: &'static ThisModule) ->
Result<Self>;
}
```

If some setup fails (e.g. finding device tree nodes or acquiring needed resources), the `init` method can return `Err`.

Drop impl

By implementing `Drop` on our module struct, we can perform any necessary cleanup and teardown.

```
impl Drop for MyModule {
    fn drop(&mut self) {
        // ...
    }
}
```

Module Parameters

Support for defining and accessing module parameters from Rust has not yet landed in mainline Linux.

However, there is outstanding work toward supporting module parameters[1].

In the meantime, it may be preferable to configure modules through sysfs.

[1] https://lore.kernel.org/rust-for-linux/20240819133345.3438739-1-nmi@metaspace.dk/

Using Abstractions

Now that we've seen a trivial driver, let's look at a real one for the `Asix ax88796b network PHY` .

Here we'll see what it looks like to register a driver for a particular subsystem and implement the needed functionality using abstractions from a subsystem.

Complications and Conflicts

There are a number of subtleties and unresolved conflicts between the Rust paradigm and the kernel one.

Some of these will require additional research and development before Rust for Linux is ready for the prime-time, while others merely require some additional learning and attention on behalf of aspiring Rust for Linux developers.

One consequence of the current state of affairs is that a nightly version of the Rust compiler is required to build Rust for Linux. This does not present a problem for development, but it does for distributions that want to ship kernels using Rust while avoiding depending on a bleeding-edge compiler toolchain. Being able to build the project with a stable compiler is a significant goal of the Rust for Linux project; the issues blocking this are tracked specifically[1].

We'll visit the most significant of these in the interest of being aware of challenges we may encounter when trying to implement kernel functionality in Rust.

[1] https://github.com/rust-lang/rust-project-goals/issues/116

`Pin` and Self-Reference

The Linux kernel pervasively relies on intrusive data structures and programming patterns that rely on the stability of data structures' addresses.

In C, these patterns show up in places like `struct list_head` and the `container_of` macro.

The programming rules for these data structures require being careful about where instances are allocated and how they are linked into and removed from their data structures.

Moves

In Rust, however, instances of data types may change their addresses any time they are moved, and the compiler is relied on to be aware of any outstanding references that would prevent moving them. The most common pattern for constructing values in Rust even involves a move– simply returning the value from a constructor function.

This paradigm does not work for values that must be constructed "in-place" to avoid moves, but the C approach of writing into a blob of uninitialized memory until fully initialized is also an anathema in Rust: it would force us into writing unsafe code any place we wanted to construct an instance of our type.

`Pin`

A similar concern already exists in Rust for compiler-generated types that internally contain self references; these can be occur in the state machines generated by the compiler for `async` functions.

The `Pin<T>` wrapper type exists to wrap an indirection (such as `&mut T` or `Box<T>`) in such a way that an `&mut T` cannot be created to the underlying `T` (as this would allow using a function like `mem::swap` that would effectively change its address).

Field projection

`Pin<T>` also has the effect of requiring a choice for each field of the pinned type: will it be accessed through a `Pin<&mut Field>` or simply through `&mut Field`? Either may be acceptable, depending on the semantics of the type, but the two options must not coexist for a single field as that would allow the `Pin<&mut Field>` to be moved via `mem::swap` on the `&mut Field`.

The boilerplate for exposing access to each field of a pinned struct ("projecting" the field) via only one of `Pin<_>` or directly is handled by the `pin-project` crate in userspace Rust.

Unfortunately, this crate uses procedural macros to parse Rust code and these in turn have heavy dependencies that the Rust for Linux project does not want to take on.

Instead, Rust for Linux has its own solution to pinned initialization and pin projection.

`pinned-init`

The solution employed for these concerns in Rust for Linux is the `pinned-init` crate. Using this crate looks like the following:

```rust
use kernel::{prelude::*, sync::Mutex, new_mutex};
#[pin_data]
struct Foo {
    #[pin]
    a: Mutex<usize>,
    b: u32,
}

let foo = pin_init!(Foo {
    a <- new_mutex!(42, "Foo::a"),
    b: 24,
});

// `foo` now is of the type `impl PinInit<Foo>`.
// We can now use any smart pointer that we like (or just the stack)
to actually initialize a Foo:

let foo: Result<Pin<Box<Foo>>> = Box::pin_init(foo);
```

Further reading

- https://rust-for-linux.com/the-safe-pinned-initialization-problem
- https://github.com/Rust-for-Linux/pinned-init

The Kernel Rust Safety Model

Soundness

Safety in normal, userspace Rust is already a subtle topic. The verification boundary for `unsafe` code is not the unsafe block or even the containing function, but the privacy boundary of the public interface of the containing module. And the guarantees that unsafe code can rely on depend on a combination of the semantics of regular Rust along with the behavior of the underlying compiler, operating system, and hardware.

In kernel Rust, things are even more complicated. The golden standard for Rust code making use of `unsafe` is that it must be impossible for any consumer of the code to trigger undefined behavior through safe interfaces. But there are many parts of the Linux kernel in which we might want to use Rust that cannot be fully compartmentalized from the rest of the kernel by a safe, water-tight API.

Many tasks performed by the kernel are only understandable outside the model of C or Rust language semantics: for example, writing to CPU registers that control paging or DMA may alter the meaning of pointers, but models of language semantics do not include notions of the underlying architecture's paging or memory-management system. Tools like miri cannot analyze programs that perform low-level operations like these, and static analysis tools similarly lack models of their effects. So we're forced to live with a less thorough notion of safety than we might have in userspace Rust.

For now, some kernel components will be suitable for writing fully safe Rust interfaces (perhaps those with limited interactions with the rest of the system, such as GPIOs), while others can only offer limited safety.

This is an area where Rust for Linux is pushing the boundaries of what Rust's paradigm of memory safety can achieve.

Limitations of Type and Nemory Safety

Rust’s guarantees of memory safety provide a baseline that can raise our confidence in Rust code head and shoulders above the status quo writing other low-level languages. But some desirable properties are difficult or impossible to guarantee through Rust’s type system.

For example, because variables can always be dropped, it’s difficult to guarantee liveness properties.

Similarly, because Rust type- and borrow-checking are local analyses, they cannot be used to ensure global properties like lock ordering.

Other tools can perform useful static analyses for Rust code similar to those that might be performed with standalone C static analysis packages or gcc compiler plugins. `Clippy` is the most common static analysis tool for Rust code, but for kernel-specific analyses the `klint` tool also exists.

Atomic/Task Contexts and Sleep

One of the safety conditions that the Rust type system does not help us establish is freedom from deadlocks. In the Linux kernel, a related concern is only sleeping in contexts where doing so is allowed. In particular, code executing in a task context may sleep, but code executing in an atomic context (for example, within an interrupt handlers, while holding a spinlock, or in an RCU critical section) may not.

Sleeping in the wrong place may lead to kernel hangs, but in the context of RCU, it can even threaten memory safety: if a CPU sleeps in an RCU read-side critical section, it will be mistakenly considered to have exited that critical section, potentially leading to use-after-free[^1].

Existing C code in the kernel relies on `might_sleep` and similar annotations which facilitate debugging via runtime tracking when `CONFIG_DEBUG_ATOMIC_SLEEP` is enabled.

Because of the need to forbid sleep, it is not sufficient to simply use RAII to model RCU in Rust as we might intuitively want to do. We need some additional checking to ensure that while RCU guards exist, no sleeps or context switches are performed.

`klint`

The `klint` tool performs static analysis on kernel Rust code and addresses this problem by tracking preemption count across all functions at compile-time.

It does so based on annotations added to our functions that specify:

- the expected range of preemption counts when calling the function
- the adjustment performed to the preemption count after the function returns

If a function is called from a context where preemption count may be outside the function's expectation, `klint` will emit an error message.

Recursive functions, generics, and function pointers complicate this analysis, so it is not foolproof, and conditional control flow around also means `klint`'s analysis is approximate. But this still catches obvious mistakes in straightforward code, and `klint` is only likely to improve its analyses.

[^1] https://www.memorysafety.org/blog/gary-guo-klint-rust-tools/

Memory Models: LKMM vs. Rust (C11) Memory Model

Memory models are their own complex topic which we will not cover in depth, but to summarize:

- The Linux Kernel and the Rust language itself use different memory models, which specify what guarantees are made when different threads interact through shared memory and low-level synchronization primitives.
    - The kernel has its own memory model (Linux Kernel Memory Model or “LKMM”).

      This is because it predates standardized formal memory models for concurrency and needs high performance for synchronization as used in RCU and elsewhere.
    - Rust inherits the semantics promised by LLVM - from the C++11 specification (and adopted by the C11 spec). So Rust essentially uses the C11 MM.
    - LKMM relies on orderings provided by address, data, and control dependencies.
    - The C11 MM does not provide all of these, so it isn’t simple to express the LKMM in terms of the C11 MM.
        - LKMM relies on semantics not guaranteed by the C spec but merely by compiler behavior.

          This means that conforming to the C standard is not sufficient for an arbitrary compiler to compile a working kernel. In practice, the kernel is only compiled with GCC or Clang, which both implement the desired semantics, so this is fine.

- Because Rust atomics and Linux kernel atomics do not necessarily provide the same guarantees, using them together could have very surprising results.
- Instead, Kernel Rust should probably re-implement corresponding atomics the same way the kernel does in C[1].
    - This should allow Rust for Linux to interoperate with the rest of the kernel in an understandable way, but could subtly alter the behavior of other crates that use atomics if used in the kernel atop kernel atomics.

▼ *Speaker Notes*

See these links for more background:

- https://rust-for-linux.zulipchat.com/#narrow/channel/288089-General/topic/Status.20of.20the.20Linux-kernel.20memory.20model.20support.20in.20Rust
- https://rust-lang.zulipchat.com/#narrow/channel/136281-t-opsem/topic/.E2.9C.94.20Rust.20and.20the.20Linux.20Kernel.20Memory.20Model
- https://lwn.net/Articles/967049/
- https://lwn.net/Articles/993785/

[1] https://github.com/rust-lang/unsafe-code-guidelines/issues/348#issuecomment-1221376388

Separate Compilation and Linking

One hiccup integrating Rust into the kernel compilation process is that C is designed for full separate compilation, where each source file can be compiled into an object file, and then these object files are linked into a single loadable archive by the C toolchain's linker. Rust, however, expects to compile its programs at the granularity of individual crates and control the linking process.

In C, the compiler is not responsible for safety, so the correctness of linking C built with different flags or compilers is left up to the user. But compiled Rust code has no stable ABI, and so the compiler must be careful not to link together two libraries compiled with different versions of the Rust compiler, or with different code-generation flags.

Target modifiers

In cases where two crates are linked together, the Rust compiler will attempt to verify that they have been compiled by the same version of the compiler to ensure that no ABI incompatibility will undermine the memory safety of their composition.

However, if one crate was compiled with modifications to its effective ABI relative to the other (such as forbidding usage of a register, like the `-ffixed-x18` flag does), then it may not be valid to conclude that the resulting program will behave as intended.

The Rust compiler currently avoids this situation primarily by treating each compiler configuration as an entirely separate target; crates compiled for different targets may not be linked together. But defining a fully custom target when running the compiler is a feature only exposed by the unstable nightly version of the compiler, which Rust for Linux does not want to commit to doing indefinitely.

The way out is a proposal[1] to create “target modifiers”, a stable way of specifying variants of standard targets at compile-time. Compiled crates will be stamped with the target variant so that the Rust compiler can ensure the target modifiers match at link-time, but users will not be required to create an entirely new compilation target.

▼ *Speaker Notes*

[1] https://github.com/rust-lang/rfcs/pull/3716

Fallible Allocation

Allocation in Rust is assumed to be infallible:

```
let x = Box::new(5);
```

In the Linux kernel, memory allocation is much more complex.

```
void * kmalloc(size_t size, int flags)
```

`flags` is one of `GFP_KERNEL`, `GFP_NOWAIT`, `GFP_ATOMIC`, etc.[1]

The return value must be checked against `NULL` to see whether allocation succeeded.

In Rust for Linux, rather than using the infallible allocation APIs provided by `liballoc`, the kernel library has its own allocation interfaces:

KBox

```
let b = KBox::new(24_u64, GFP_KERNEL)?;
assert_eq!(*b, 24_u64);
```

`KBox::new` returns a `Result<Self, AllocError>`. Here we propagate this error with the `?` operator.

KVec

Similarly, `KVec` presents a similar API to the standard `Vec`, but where operations that may allocate take a flags parameter:

```
let mut v = KVec::new();
v.push(1, GFP_KERNEL)?;
assert_eq!(&v, &[1]);
```

FromIterator

Because the standard `FromIterator` trait also involves making new collections often involving memory allocation, the `.collect()` method on iterators is not available in Rust for Linux in its original form. Work is ongoing to design an equivalent API[2], but for now we do without its convenience.

[1] https://docs.kernel.org/core-api/memory-allocation.html

[2] https://rust-for-linux.zulipchat.com/#narrow/channel/288089-General/topic/flat_map.20collecting.20with.20Kvec

Code Size

One pitfall when writing Rust code can be the multiplicative increase in generated machine code when using generics.

For the Linux kernel, which must be suitable for space-limited embedded environments, keeping code size low is a significant concern.

Experiments with Rust in the kernel so far have shown that Rust code can be of similar code size to C, but may also be larger in some cases[1].

Assessing Bloat

Tools exist to help analyze different source code's contribution to the size of compiled code, such as `cargo-bloat`.

Shrinking Code Size

The reasons for code bloat vary and are not generally specific to Linux kernel usage of Rust. The most common causes for code bloat are excessive use of generics and forced inlining. In general, generics should be prefered over trait objects when writing abstractions that are expected to "compile out" or where generating separate code for different types is critical for performance (e.g. inner loops or arithmetic on values of a generic type).

In other situations, trait objects should be prefered to allow reusing definitions without machine-code duplication, which may closer mirror patterns that would be most natural in C.

When accepting generic parameters that get converted to a concrete type before use, follow the pattern of defining an inner monomorphic function that can be shared[2]:

```rust
pub fn read_to_string<P: AsRef<Path>>(path: P) -> io::Result<String> {
    fn inner(path: &Path) -> io::Result<String> {
        let mut file = File::open(path)?;
        let size = file.metadata().map(|m| m.len() as usize).ok();
        let mut string = String::with_capacity(size.unwrap_or(0));
        io::default_read_to_string(&mut file, &mut string, size)?;
        Ok(string)
    }
    inner(path.as_ref())
}
```

[1] https://www.usenix.org/system/files/atc24-li-hongyu.pdf

[2] https://github.com/rust-lang/rust/blob/ae612bedcbfc7098d1711eb35bc7ca994eb17a4c/library/std/src/fs.rs#L295-L304

Documentation

Documentation in Rust for Linux is built with the `rustdoc` tool just like for regular Rust code.

Running rustdoc on the kernel is done with the `rustdoc` Make target:

```
make LLVM=1 rustdoc
```

after which generated docs can be viewed by opening `Documentation/output/rust/rustdoc/kernel/index.html`.

Pre-generated documentation for the current kernel release is available at:

https://rust.docs.kernel.org/kernel/

More information

https://docs.kernel.org/rust/general-information.html#code-documentation

Security Mitigations

Even though Rust is memory-safe, larger systems using Rust are not necessarily memory-safe. The kernel is no exception. The kernel is often compiled with various security mitigations and hardening flags, and to avoid undermining these (e.g. by providing gadgets or running afoul of CPU errata), Rust code compiled into the kernel should also be built with the same set of mitigations.

Many of these mitigations are already supported by the Rust compiler, which merely needs to expose the same underlying LLVM functionality offered by Clang.

Speculative execution (Meltdown/Spectre) mitigations

Recent CPU side-channel vulnerabilities in particular require changes to compilers' code generation ("retpolines", etc.) in order to prevent userspace access to kernel data. Support for these code-generation changes is still pending in `rustc`[1].

[1] https://github.com/Rust-for-Linux/linux/issues/355

Async

The kernel performs many operations concurrently and involves significant amounts of interaction between CPU cores and other devices. For this reason, it would be no surprise to see that async Rust would be a fundamental requirement for using Rust in the kernel. But the kernel is central arbitrer of most synchronization and is currently written in regular, synchronous C.

Rust code making use of `async` mostly exists to write composable code that will run atop event loops, but the Linux kernel is not really organized as an event loop: user tasks call directly into the kernel; control flow for interrupts is handled by hardware.

As such, `async` support is not critical for most kernel programming tasks. However, it is possible to view some components of the kernel as async executors, and some work has been done in this direction. Wedson Almeida Filho implemented both workqueue-based[1] and single-threaded async executors as proofs of concept.

There is not a fundamental incompatibility between Rust-for-Linux and Rust `async`, which is a similar situation to the amenability of `async` to use in embedded Rust programming (e.g. the Embassy project).

Nonetheless, no killer application of `async` in Rust for Linux has made it a priority.

▼ *Speaker Notes*

[1] https://github.com/Rust-for-Linux/linux/tree/rust/rust/kernel/kasync

An example of an async server using the kernel async executor may be found here.

Next Steps

The Linux documentation has a number of pointers to further resources on using Rust in the kernel.

In addition, it’s well worthwhile to join the Rust for Linux Zulip.