Rust for Linux: Code Documentation & Tests

Miguel Ojeda
ojeda@kernel.org
Recap
Rust in the kernel

Driver — Safe — Subsystem 1, Subsystem 2, ... — Unsafe — C APIs

Safe Abstractions

Forbidden!
**Safe and Unsafe (functions)**

**Safe function**: a function that does not trigger undefined behavior in any context and/or for any possible inputs.

That is, it does not have any precondition (regarding undefined behavior).

In other words, whatever a caller does, it does not produce undefined behavior.

**Unsafe function**: a function that is not **safe**, prefixed with the **unsafe** keyword.

This means it has safety preconditions.

Callers have to declare they are upholding the contract.
Safe and Unsafe (code)

Unsafe code: code inside an unsafe block.

It has access to all operations.

Safe code: code that is outside an unsafe block (i.e. the default).

It cannot perform some operations (e.g. calling unsafe functions or dereferencing raw pointers).

Unsafe block: a block of code prefixed with the unsafe keyword.
Convention

= Safe function

= Unsafe function

= Safe code

= Unsafe code
The deny(unsafe_op_in_unsafe_fn) “dialect”

Kernel code is written in this “dialect”:

```rust
pub unsafe fn f() {
    g();       // The default in Rust: unsafe function implies unsafe block.
}

pub unsafe fn f() {
    unsafe { g(); } // Dialect: an explicit unsafe block is needed.
}
```

This may become the default in a future Rust edition.
Documenting a Rust function
Documenting a Rust function

Let us migrate this C function to Rust and document it:

```rust
i32 load(const i32 * p) {
    return *p;
}
```
In Rust, one may start writing:

```rust
pub fn load(p: *const i32) -> i32 {
    *p
}
```
Documenting a Rust function

We require that all public “items” (functions, types, modules...) must be documented.

```rust
/// Loads a 32-bit signed integer.
pub fn load(p: *const i32) -> i32 {
    *p
}
```
Documenting a Rust function

This code does not compile:

```rust
/// Loads a 32-bit signed integer.
pub fn load(p: *const i32) -> i32 {
    *p
}
```

error[E0133]: dereference of raw pointer is unsafe and requires unsafe block

```rust
2 | *p
   | ^^^ dereference of raw pointer
```

= note: raw pointers may be null, dangling or unaligned; they can violate aliasing rules and cause data races: all of these are undefined behavior
Documenting a Rust function

A raw pointer dereference may introduce undefined behavior.

Rust requires an `unsafe` block.

/// Loads a 32-bit signed integer.
pub fn load(p: *const i32) -> i32 {
    unsafe {
        *p
    }
}
Documenting a Rust function

We require to justify why there is no undefined behavior.

For this, we use a // SAFETY: comment.

```rust
/// Loads a 32-bit signed integer.
pub fn load(p: *const i32) -> i32 {
    // SAFETY: ...
    unsafe {
        *p
    }
}
```
One way to justify it is to ask the callers to uphold a precondition.

```rust
/// Loads a 32-bit signed integer.
///
/// `p` must be valid, aligned and point to initialized memory.
pub fn load(p: *const i32) -> i32 {
    // SAFETY: ...
    unsafe { *p }
}
```
Documenting a Rust function

Now we have a precondition that is required to preserve safety.

Functions with safety preconditions are `unsafe` and must be marked as such.

```rust
/// Loads a 32-bit signed integer.
///
/// `p` must be valid, aligned and point to initialized memory.
pub unsafe fn load(p: *const i32) -> i32 {
    // SAFETY: ...
    unsafe { *p }
}
```
We require safety preconditions to be written in a `# Safety` section.

```rust
/// Loads a 32-bit signed integer.
///
/// # Safety
///
/// `p` must be valid, aligned and point to initialized memory.
pub unsafe fn load(p: *const i32) -> i32 {
    // SAFETY: ...
    unsafe { *p }
}
```
Documenting a Rust function

Now we can use the safety precondition to justify the `unsafe` block.

```rust
/// Loads a 32-bit signed integer.
///
/// # Safety
///
/// `p` must be valid, aligned and point to initialized memory.
pub unsafe fn load(p: *const i32) -> i32 {
    // SAFETY: The safety requirements of the function ensure we can
dereference and produce a valid value.
    unsafe { *p }
}
```
The /// # Safety section

Unsafe functions have safety preconditions that the caller must uphold.

The # Safety section in the documentation of an unsafe function describes those safety preconditions (requirements).

For instance, a function that takes a raw pointer and requires it to be valid must state so in this section. Callers must comply with it.

They should not be confused with the // SAFETY: comments.

It is also used in unsafe traits to describe requirements for implementers.
The // SAFETY: comments

A // SAFETY: comment must precede every unsafe block.

The comment must explain why the unsafe block does not invoke undefined behavior.

For instance, there must be a // SAFETY: comment above a raw pointer dereference justifying why the raw pointer is valid and aligned.
Other ways of justifying an `unsafe` block

For instance, consider:

```rust
/// Returns a string representing the error, if one exists.
pub fn name(&self) -> Option<&'static CStr> {
    // SAFETY: Just an FFI call, there are no extra safety requirements.
    let ptr = unsafe {
        bindings::errname(-self.0)
    };
    if ptr.is_null() {
        None
    } else {
        // SAFETY: The string returned by `errname` is static and `NUL`-terminated.
        Some(unsafe {
            CStr::from_char_ptr(ptr)
        })
    }
}
```
Type invariants
Type invariants

Assume there exists a `print_error()` C function.

Let us consider how to justify the following `unsafe` block:

```rust
pub fn print_error(code: i32) {
    // SAFETY: ...
    unsafe { bindings::print_error(code) }
}
```
Type invariants

First, we create a new type:

```rust
/// Represents an error code.
pub struct ErrorCode(i32);
```
Type invariants

First, we create a new type:

```rust
/// Represents an error code.
pub struct ErrorCode(i32);

impl ErrorCode {
    // Methods go here.
}
```
Type invariants

Then, we add a constructor:

```rust
/// Represents an error code.
pub struct ErrorCode(i32);

impl ErrorCode {
    /// Creates an [ErrorCode] from an error code.
    pub fn from_code(code: i32) -> Result<Self, ()> {
        if code > MAX_ERROR_CODE {
            return Err(());
        }
        Ok(Self(code))
    }
}
```
Type invariants

If an `ErrorCode` exists, then the code has to be valid by construction.

```rust
/// Represents an error code.
pub struct ErrorCode(i32);

impl ErrorCode {
    /// Creates an ['ErrorCode'] from an error code.
    pub fn from_code(code: i32) -> Result<Self, ()> {
        if code > MAX_ERROR_CODE {
            return Err(());
        }
        Ok(Self(code))
    }
}
```
Type invariants

We add a method to get back the raw error code:

```rust
/// Represents an error code.
pub struct ErrorCode(i32);

impl ErrorCode {
    // ...

    /// Returns the contained error code.
    pub fn to_code(&self) -> i32 {
        self.0
    }
}
```
We add a method to get back the raw error code:

```rust
/// Represents an error code.
pub struct ErrorCode(i32);

impl ErrorCode {
    /// Returns the contained error code.
    /// The result is guaranteed to be a valid error code.
    pub fn to_code(&self) -> i32 {
        self.0
    }
}
```
Now we can modify our original function to take an `ErrorCode` instead:

```rust
code: i32) {
     // SAFETY: ...
     unsafe { bindings::print_error(code) }
}
```

```rust
code: ErrorCode) { // SAFETY: ...
     unsafe { bindings::print_error(ec.to_code()) }
}
```
Type invariants

And write the justification for the `unsafe` block.

```rust
pub fn print_error(code: i32) {
    // SAFETY: ...
    unsafe {
        bindings::print_error(code)
    }
}

pub fn print_error(ec: ErrorCode) {
    // SAFETY: The error code returned by `to_code()` is always valid.
    unsafe {
        bindings::print_error(ec.to_code())
    }
}
```
Type invariants

We say the `ErrorCode` type maintains an invariant.

We document type invariants in an `# Invariants` section:

```rust
/// Represents a valid error code.
pub struct ErrorCode(i32);
```
Type invariants

We say the `ErrorCode` type maintains an invariant.

We document type invariants in an `# Invariants` section:

```rust
/// Represents a valid error code.
///
/// # Invariants
///
/// The error code is within the interval of valid error codes, as defined
/// by specification X.
pub struct ErrorCode(i32);
```
Type invariants

We also document why invariants are upheld when we create or mutate the state:

```rust
impl ErrorCode {
    /// Creates an `[ErrorCode]` from an error code.
    pub fn from_code(code: i32) -> Result<Self, ()> {
        if code > MAX_ERROR_CODE {
            return Err(());
        }
        Ok(Self(code))
    }
}
```
Type invariants

We also document why invariants are upheld when we create or mutate the state:

```rust
impl ErrorCode {
    /// Creates an [`ErrorCode`\[\] from an error code.
    pub fn from_code(code: i32) -> Result<Self, ()> {
        if code > MAX_ERROR_CODE {
            return Err(());
        }

        // INVARIANT: The check above ensures the type invariant holds.
        Ok(Self(code))
    }
}
```
The /// # Invariants section

Types may have invariants, i.e. properties all objects of that type satisfy.

The # Invariants section in the documentation of a type describes those invariants.

For instance, a type that wraps a pointer in a way that it kept always valid should document it in this section. Other code can rely on that.

They should not be confused with the // INARIANT: comments.
The // INARIANT: comments

An // INARIANT: comment should be used when object state related to the invariant is mutated (including construction).

The comment explains why the type invariant is still preserved.

For instance, when a type with an invariant is constructed, an // INARIANT: comment should precede the statement, explaining why the invariant holds.

It may be used for loop invariants as well.
/// An owned string that is guaranteed to have exactly one `\`NUL\` byte, which is at the end.
/// /// Used for interoperability with kernel APIs that take C strings.
/// /// # Invariants
/// /// The string is always `\`NUL\`-terminated and contains no other `\`NUL\` bytes.
pub struct CString {
    buf: Vec<u8>,
}

impl CString {
    /// Creates an instance of [\`CString\`] from the given formatted arguments.
    pub fn try_from_fmt(args: fmt::Arguments<_>) -> Result<Self, Error> {
        // ... 

        // INVARIANT: We wrote the `\`NUL\` terminator and checked above that no
        // other `\`NUL\` bytes exist in the buffer.
        Ok(Self { buf })
    }
}
An example from the kernel

```rust
impl Deref for CString {
    type Target = CStr;

    fn deref(&self) -> &Self::Target {
        // SAFETY: The type invariants guarantee that the string is
        // `NUL`-terminated and that no other `NUL` bytes exist.
        unsafe { CStr::from_bytes_with_nul_unchecked(self.buf.as_slice()) }
    }
}
```
Writing examples
Writing examples

For instance, consider a data structure:

```rust
/// A red-black tree with owned nodes.
///
/// It is backed by the kernel C red-black trees.
///
/// # Invariants
///
/// Non-null parent/children pointers stored in instances of the `rb_node`
/// C struct are always valid, and pointing to a field of our internal
/// representation of a node.
pub struct RBTree<K, V> {
    // ...
}
```
Writing examples

We want to show how to use the data structure. For that, we add an # Examples section:

```rust
/// ...  
/// 
/// # Examples  
/// 
/// In the example below we do several operations on a tree.  
/// We note that insertions may fail if the system is out of memory.  
/// 
/// # use kernel::prelude::*;  
/// use kernel::rbtree::RBTree;  
/// fn rbtest() -> Result {  
/// // Create a new tree.  
/// let mut tree = RBTree::new();  
/// 
/// // Insert three elements.  
/// tree.try_insert(20, 200)?;  
/// tree.try_insert(10, 100)?;  
/// tree.try_insert(30, 300)?;  
/// 
/// ... 
```
Writing examples

We can write assertions:

```rust
/// ... 
/// 
/// // Check the nodes we just inserted. 
/// { 
///     let mut iter = tree.iter();
///     assert_eq!(iter.next().unwrap(), (&10, &100));
///     assert_eq!(iter.next().unwrap(), (&20, &200));
///     assert_eq!(iter.next().unwrap(), (&30, &300));
///     assert!(iter.next().is_none());
/// } 
/// ... 
```
Writing examples

We can write several independent examples:

```plaintext
/// # Examples
///
/// In the example below we do several operations on a tree. We note that insertions may fail if
/// the system is out of memory.
///
/// First example...
///
/// In the example below, we first allocate a node, acquire a spinlock, then insert the node into
/// the tree. This is useful when the insertion context does not allow sleeping, for example, when
/// holding a spinlock.
///
/// Second example...
///
/// In the example below, we reuse an existing node allocation from an element we removed.
///
/// Third example...
```
The /// # Examples section

Function, type, trait, module and crate documentation should contain an # Examples section with examples as needed to showcase and clarify their usage.

For instance, a module should provide examples showing the most common use cases of its APIs.

It is also useful to show common pitfalls.

Examples double as tests: they are compiled and run (when enabled).
How it looks like
Function core::ptr::read

pub unsafe fn read<T>(src: *const T) -> T

- Reads the value from src without moving it. This leaves the memory in src unchanged.

Safety

Behavior is undefined if any of the following conditions are violated:

- src must be valid for reads.
- src must be properly aligned. Use read_unaligned if this is not the case.
- src must point to a properly initialized value of type T.

Note that even if T has size 0, the pointer must be non-null and properly aligned.

Examples

Basic usage:

```rust
let x = 12;
let y = &x as *const i32;

unsafe {
    assert_eq!(std::ptr::read(y), 12);
}
```

Manually implement mem::swap:

```rust
unsafe {
    mem::swap(y, std::ptr::read(y));
}
```
Function core::ptr::read

```
pub unsafe fn read<T>(src: *const T) -> T
```

- Reads the value from `src` without moving it. This leaves the memory in `src` unchanged.

### Safety

Behavior is undefined if any of the following conditions are violated:

- `src` must be valid for reads.
- `src` must be properly aligned. Use `read_unaligned` if this is not the case.
- `src` must point to a properly initialized value of type `T`.

Note that even if `T` has size 0, the pointer must be non-null and properly aligned.

### Examples

Basic usage:

```
let x = 12;
let y = &x as *const 132;

unsafe {
    assert_eq!(std::ptr::read(y), 12);
}
```

Manually implement `mem::swap`:
### Safety

Behavior is undefined if any of the following conditions are violated:

- `src` must be valid for reads.
- `src` must be properly aligned. Use `read_unaligned` if this is not the case.
- `src` must point to a properly initialized value of type `T`.

Note that even if `T` has size 0, the pointer must be non-null and properly aligned.

### Examples

**Basic usage:**

```rust
let x = 12;
let y = x as *const i32;

unsafe {
    assert_eq!(std::ptr::read(y), 12);
}
```

Manually implement `mem::swap:`
Function core::ptr::read

```rust
pub unsafe fn read<T>(src: *const T) -> T
```

Reads the value from `src` without moving it. This leaves the memory in `src` unchanged.

**Safety**

Behavior is undefined if any of the following conditions are violated:

- `src` must be valid for reads.
- `src` must be properly aligned. Use `read_unaligned` if this is not the case.
- `src` must point to a properly initialized value of type `T`.

Note that even if `T` has size 0, the pointer must be non-null and properly aligned.

**Examples**

Basic usage:

```rust
let x = 12;
let y = &x as *const 132;

unsafe {
    assert_eq!(std::ptr::read(y), 12);
}
```

Manually implement `mem::swap`:
**Function core::ptr::read**

```rust
pub unsafe fn read<T>(src: *const T) -> T
```

- Reads the value from `src` without moving it. This leaves the memory in `src` unchanged.

**Safety**

Behavior is undefined if any of the following conditions are violated:

- `src` must be `valid` for reads.
- `src` must be properly aligned. Use `read_unaligned` if this is not the case.
- `src` must point to a properly initialized value of type `T`.

Note that even if `T` has size 0, the pointer must be non-null and properly aligned.

**Examples**

Basic usage:

```rust
let x = 12;
let y = &x as *const 132;

unsafe {
    assert_eq!(std::ptr::read(y), 12);
}
```

Manually implement `mem::swap`:
Struct kernel::rbtree::RBTree

```rust
pub struct RBTree<K, V> { /* fields omitted */ }
```

[-] A red-black tree with owned nodes.
It is backed by the kernel C red-black trees.

**Invariants**

Non-null parent/children pointers stored in instances of the `rb_node` C struct are always valid, and pointing to a field of our internal representation of a node.

**Examples**

In the example below we do several operations on a tree. We note that insertions may fail if the system is out of memory.

```rust
use kernel::rbtree::RBTree;

fn rbtest() -> Result {
    // Create a new tree.
    let mut tree = RBTree::new();

    // Insert three elements.
    tree.try_insert(20, 200)?;
    tree.try_insert(10, 100)?;
    tree.try_insert(30, 300)?;

    // Check the nodes we just inserted.
    {
        let mut iter = tree.iter();
        assert_eq!(iter.next().unwrap(), (&10, &100));
    }
}
```
Other notes
Documenting modules

If a module is written in its own file, then we document it inside it using `//!`:

```rust
// SPDX-License-Identifier: GPL-2.0

//! Red-black trees.
//!
//! C header: [`include/linux/rbtree.h`](../../../../include/linux/rbtree.h)
//!

use crate::{bindings, Result};
use alloc::boxed::Box;
// ...
```
Other sections

In the standard Rust library, which we use, other fairly common sections are:

- # Errors section.
- # Panics section.

We may start using these, or new ones, too, as we explore how to best document consistently the code.
Writing prose

Feel free to document any item extensively, including writing prose divided with extra sections that are not the “standard” ones.

Typically, this applies to modules and types.

For instance, a good example is the documentation for Vec from the standard library.

These docs may be enough to replace the usual Documentation/ ones.
Indexing

The `Vec` type allows to access values by index, because it implements the `Index` trait. An example will be more explicit:

```rust
let v = vec![0, 2, 4, 6];
println!("{}", v[1]); // it will display '2'
```

However be careful: if you try to access an index which isn't in the `Vec`, your software will panic! You cannot do this:

```rust
let v = vec![0, 2, 4, 6];
println!("{}", v[6]); // it will panic!
```

Use `get` and `get_mut` if you want to check whether the index is in the `Vec`.

Slicing

A `Vec` can be mutable. On the other hand, slices are read-only objects. To get a `slice`, use `&`. Example:

```rust
fn read_slice(slice: &[usize]) {
    // ...
}

let v = vec![0, 1];
read_slice(&v);
```

// ... and that's all!
// you can also do it like this:
Vec

Methods
allocator
append
as_mut_ptr
as_mut_slice
as_ptr
as_slice
capacity
clear
dedup
dedup_by
dedup_by_key
drain
drain_filter
extend_from_slice
extend_from_within
from_raw_parts
from_raw_parts_in
insert
into_boxed_slice
into_raw_parts
into_raw_parts_with_alloc
is_empty

**capacity** - A Vec will allocate it and only if `mem::size_of::<T>() * capacity() > 0`. In general, Vec’s allocation details are very subtle — if you intend to allocate memory using a Vec and use it for something else (either to pass to unsafe code, or to build your own memory-backed collection), be sure to deallocate this memory by using `from_raw_parts` to recover the Vec and then dropping it.

If a Vec has allocated memory, then the memory it points to is on the heap (as defined by the allocator Rust is configured to use by default), and its pointer points to `len` initialized, contiguous elements in order (what you would see if you coerced it to a slice), followed by `capacity - len` logically unitialized, contiguous elements.

A vector containing the elements 'a' and 'b' with capacity 4 can be visualized as below. The top part is the Vec struct, it contains a pointer to the head of the allocation in the heap, length and capacity. The bottom part is the allocation on the heap, a contiguous memory block.

```
+-----------------+-
| ptr | len | capacity |
+-----------------+-
| 0x0123 | 2 | 4 |
+-----------------+-

Heap
+-----------------+-
| 'a' | 'b' | uninit | uninit |
+-----------------+-
```

- **uninit** represents memory that is not initialized, see `MaybeUninit`.
- Note: the ABI is not stable and Vec makes no guarantees about its memory layout (including the order of fields).

Vec will never perform a “small optimization” where elements are actually stored on the stack for two reasons:

- It would make it more difficult for unsafe code to correctly manipulate a Vec. The contents of a Vec wouldn’t have a stable address if it were only moved, and it would be more difficult to determine if a Vec had actually allocated memory.
- It would penalize the general case, incurring an additional branch on every access.

Vec will never automatically shrink itself, even if completely empty. This ensures no unnecessary allocations or deallocations occur. Emptying a Vec and then filling it back up to the same `len` should incur no calls to the allocator. If you wish to free up unused memory, use `shrink_to_fit` or `shrink_to`. 
Other coding guidelines

No direct access to C bindings.
Rust 2021 edition & idioms.
No undocumented public APIs.
No unneeded panics.
No infallible allocations.
Clippy linting enabled.
Automatic formatting enforced.

...
Useful references

rustdoc guide on writing documentation:

https://doc.rust-lang.org/rustdoc/how-to-write-documentation.html

rustdoc guide on writing documentation tests:

https://doc.rust-lang.org/rustdoc/documentation-tests.html

Rust for Linux coding guidelines:

Tests
Kinds of tests

In Rust projects, there are usually 3 kinds of tests:

Unit tests (#[test] in the source code).

Documentation tests (e.g. # Examples section).

Integration tests (in a tests/ folder).

We would like to adapt those for the kernel:

We are working on integrating these with KUnit.
Unit tests

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_cstr_to_str() {
        let good_bytes = b"\xf0\x9f\xa680\0";
        let checked_cstr = CStr::from_bytes_with_nul(good_bytes).unwrap();
        let checked_str = checked_cstr.to_str().unwrap();
        assert_eq!(checked_str, "🦀");
    }

    #[test]
    #[should_panic]
    fn test_cstr_to_str_panic() {
        let bad_bytes = b"\xc3\x28\0";
        let checked_cstr = CStr::from_bytes_with_nul(bad_bytes).unwrap();
        checked_cstr.to_str().unwrap();
    }
}
The tests are compiled and run in the Rust for Linux CI before merging new code. Currently, this only tests a few configurations. Build testing with other configurations was recently added to the 0-DAY CI Kernel Test Service.
Conclusions
Conclusions

Rust requires `unsafe` blocks around potentially-UB operations.

It also provides a way to mark functions that may trigger UB.

On top of that, in Rust for Linux:

- **Unsafe** functions do not imply an `unsafe` block (`unsafe_op_in_unsafe_fn`).
- **Unsafe** blocks must be justified (`// SAFETY`).
- **Unsafe** functions must be marked as such (`unsafe`).
- Safety preconditions must be documented (`# Safety`).
Conclusions

Type invariants are a key tool to develop APIs with less safety preconditions.

Code documentation does not change the behavior of the code, but:

They make reviewing the soundness of a module easier.

The # Safety sections are critical for users to understand the preconditions.

Writing examples is useful to document, test and make sure the documentation stays aligned with the code.

We are working on integrating the testing support with the kernel.
Rust for Linux: Code Documentation & Tests

Miguel Ojeda
ojeda@kernel.org
We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The LF Mentoring Program is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- Outreachy remote internships program supports diversity in open source and free software.
- Linux Foundation Training offers a wide range of free courses, webinars, tutorials and publications to help you explore the open source technology landscape.
- Linux Foundation Events also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at events.linuxfoundation.org.

Thank you for joining us today!
Backup slides
Rust tree

library/

- core crate
- alloc crate

- builtins crate

Linux tree

rust/

- alloc crate
- kernel crate
- macros crate

- exports
- helpers

include/

- bindgen
- bindings crate
(Un)safe functions vs. (un)safe code

<table>
<thead>
<tr>
<th>Safe function with only safe code</th>
<th>Unsafe function with only safe code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe function with unsafe code</td>
<td>Unsafe function with unsafe code</td>
</tr>
</tbody>
</table>
What happens if we make a mistake?

If a \texttt{safe} function is not actually \texttt{safe}, then it is called \texttt{unsound}.

This is considered a bug.

In the standard library, a CVE is assigned.

\begin{verbatim}
fn f(p: *const i32) -> i32 {
    unsafe { *p }
}
\end{verbatim}