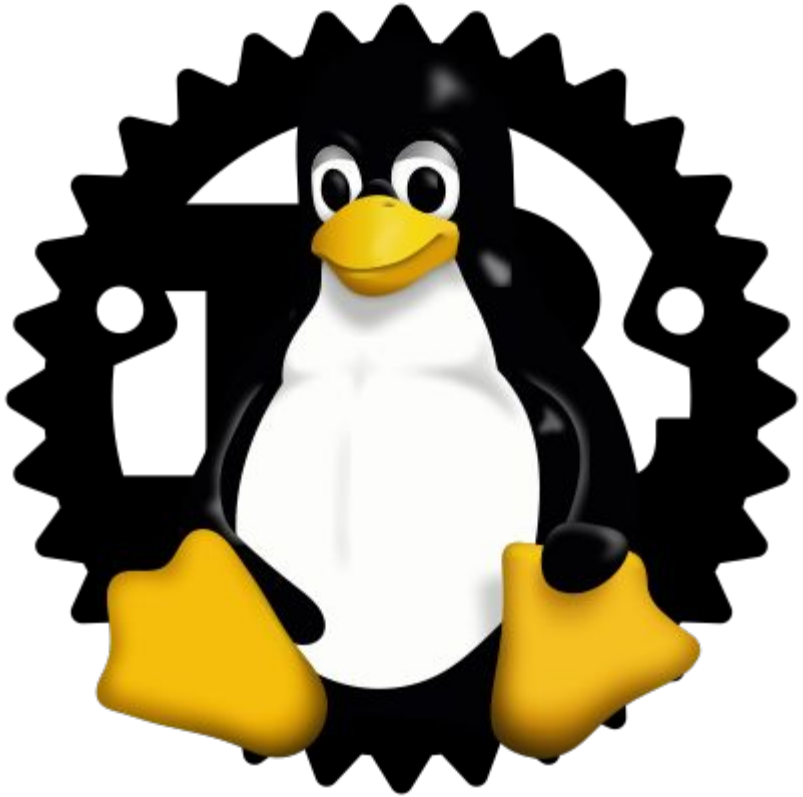


Rust for Linux

Miguel Ojeda
Wedson Almeida Filho
Alex Gaynor

Introduction

Rust for Linux aims to add Rust support to the Linux kernel.

We believe Rust offers key improvements over C in this domain.

We hope this talk results in the improvement of Rust for everyone!

Language

Library

Tooling

Nightly features

Language
Library
Tooling

A general priority is to stabilize them (or to find alternatives):

https://github.com/Rust-for-Linux/linux/issues/2

The kernel should build without `RUSTC_BOOTSTRAP`.

The rest of this talk focuses on things that are *not* in nightly.

Nightly features

Language
Library
Tooling

```
feature(allocator_api)
feature(associated_type_defaults)
feature(bench_black_box)
feature(coerce_unsized)
feature(concat_idents)
feature(const_fn_trait_bound)
feature(const_mut_refs)
feature(core_panic)
feature(dispatch_from_dyn)
feature(doc_cfg)
feature(generic_associated_types)
feature(global_asm)
feature(ptr_metadata)
feature(receiver_trait)
feature(unsize)
```

```
cfg(no_fp_fmt_parse)
cfg(no_global_oom_handling)
cfg(no_rc)
cfg(no_sync)
```

```
-Zbinary_dep_depinfo=y
-Zbuild-std
-Zsymbol-mangling-version=v0
```

Pinning: init workaround example

Language

Library

```
impl<T> Mutex<T> {
    /// Constructs a new mutex.
    ///
    /// # Safety
    ///
    /// The caller must call [`Mutex::init_lock`] before using the mutex.
    pub unsafe fn new(t: T) -> Self {
        todo!()
    }
}

impl<T> CreatableLock for Mutex<T> {
    unsafe fn init_lock(
        self: Pin<&mut Self>,
        name: &'static CStr,
        key: *mut bindings::lock_class_key,
    ) {
        todo!()
    }
}
```

`Mutex::new`: is unsafe, requires `init_lock` call.

C mutex init happens here, now that address is stable.

Synchronisation primitive initialisation: C example

Language

Library

```
/**
 * mutex_init - initialize the mutex
 * @mutex: the mutex to be initialized
 *
 * Initialize the mutex to unlocked state.
 *
 * It is not allowed to initialize an already locked mutex.
 */
#define mutex_init(mutex)                                   \
do {                                                        \
    static struct lock_class_key __key;                     \
                                                            \
    __mutex_init((mutex), #mutex, &__key);                  \
} while (0)
```

New static variable for each init call.

Mutex name is derived from the expression.

Pinning: usage example

Language

Library

```
fn new(ctx: Ref<Context>) -> Result<Ref<Self>> {
    let mut process = Pin::from(UniqueRef::try_new(Self {
        ctx,
        task: Task::current().group_leader().clone(),
        // SAFETY: `inner` is initialised in the call to `mutex_init` below.
        inner: unsafe { Mutex::new(ProcessInner::new()) },
        // SAFETY: `node_refs` is initialised in the call to `mutex_init` below.
        node_refs: unsafe { Mutex::new(ProcessNodeRefs::new()) },
    })?);

    // SAFETY: `inner` is pinned when `Process` is.
    let pinned = unsafe { process.as_mut().map_unchecked_mut(|p| &mut p.inner) };
    mutex_init!(pinned, "Process::inner");

    // SAFETY: `node_refs` is pinned when `Process` is.
    let pinned = unsafe { process.as_mut().map_unchecked_mut(|p| &mut p.node_refs) };
    mutex_init!(pinned, "Process::node_refs");

    Ok(process.into())
}
```

Unsafe `Mutex::new`: requires `init_lock` call before use.

`Self` is pinned, but projections aren't necessarily.

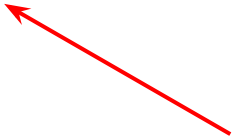
Names are manually specified.

Pinning: ideal ergonomics

Language

Library

```
fn new(ctx: Ref<Context>) -> Result<Ref<Self>> {
    Ref::try_new(Self {
        ctx,
        task: Task::current().group_leader().clone(),
        inner: Mutex::new(ProcessInner::new()),
        node_refs: Mutex::new(ProcessNodeRefs::new()),
    })
}
```



Magic would allow `Mutex::new` to initialise in-place, define a new lock-class static variable, know that `Self::inner` and `Self::node_refs` are also pinned, infer the name, and call `__mutex_init`.

Modularization of `core` and `alloc`

Language

Library

The kernel does not need a range of `core`/`alloc` features.

Configuring them off is important for correctness, code size…

Likely useful for other domains too:

- Embedded.
- Safety-critical.

Modularization of `core` and `alloc`

Language

Library

Floating-point:

```
cfg(no_fp_fmt_parse)
```

128-bit integers.

Unicode.

Infallible allocation APIs:

```
cfg(no_global_oom_handling).
```

Some `try_*` methods still missing.

Types (e.g. `Rc`) implemented in terms of existing kernel facilities:

```
cfg(no_rc)
cfg(no_sync)
```

Language

Memory model: current status, example

```
struct Example {
    a: AtomicU32,
    b: AtomicU32,
}

fn get_sum_with_guard(
     v: &SeqLock<SpinLock<Example>>
) -> u32 {
    loop {
        let guard = v.read();
        let sum =
            guard.a.load(Ordering::Relaxed) +
            guard.b.load(Ordering::Relaxed);
        if !guard.need_retry() {
            break sum;
        }
    }
}
```

```
403fd4:  14000002   b       403fdc
403fd8:  d503203f   yield
403fdc:  b9400808   ldr     w8, [x0, #8]
403fe0:  3707ffc8   tbnz    w8, #0, 403fd8
403fe4:  d50339bf   dmb     ishld
403fe8:  b9400c09   ldr     w9, [x0, #12]
403fec:  b940100a   ldr     w10, [x0, #16]
403ff0:  d50339bf   dmb     ishld
403ff4:  b940080b   ldr     w11, [x0, #8]
403ff8:  6b08017f   cmp     w11, w8
403ffc:  54ffff01   b.ne    403fdc
404000:  0b090148   add     w8, w10, w9
```

Language

Memory model: future potential

Unified/Compatible Linux kernel and Rust memory models:

- No need to use inline assembly to define a new memory model.

Language-supported address and control dependencies:

- No fragility when implementing high-performance concurrent code.

Rust would be *better* than C:

- Wouldn't require workarounds that affect performance.

Avoid assuming Cargo

Tooling

Cargo is a great package manager & build system.

However, the kernel does not use packages and has its own build system.

Thus tooling that may only be used through Cargo is problematic, e.g.:

`build-std`

: Used for the current test support hack.

Miri

: Not used yet, but we would like to.
See https://github.com/rust-lang/miri/issues/1835.

Const support: device id tables, C example

```
struct amba_id {
    unsigned int        id;
    unsigned int        mask;
    void                *data;
};

static const struct amba_id pl061_ids[] = {
    {
        .id     = 0x00041061,
        .mask   = 0x000fffff,
    },
    { 0, 0 },
};
MODULE_DEVICE_TABLE(amba, pl061_ids);
```

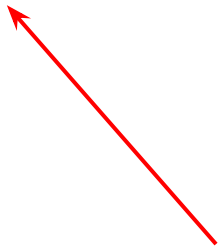
May contain device-specific untyped data.

Must be zero-terminated.

Used to build a table in final binary that can be read by tools.

Const support: device id tables, Rust example

```
impl amba::Driver for PL061Device {
    type Data = Ref<DeviceData>;
    type PowerOps = Self;

    declare_amba_id_table! [
        { id: 0x00041061, mask: 0x000fffff, data: () },
    ];

    // [...]
}
```



We need to generate a const zero-terminated array of `amba_id` entries.

Const support: device id tables, compiles but incomplete

```
trait Driver<const ID_TABLE_SIZE: usize> {
    const ID_TABLE: [Id; ID_TABLE_SIZE];
}

const fn null_terminated<const N: usize>(
    table: &[Id; N],
) -> [MaybeUninit<RawId>; N + 1] {
    let mut ret = [MaybeUninit::uninit(); N + 1];
    let mut i = 0;
    while i < N {
        ret[i].write(table[i].into_raw(i));
        i += 1;
    }
    ret
}
```

Driver is parameterized on the size of the table.

Specific to `Id` and `RawId` types.

Requires `generic_const_exprs`, which is incomplete.

Not guaranteed to be zeroed. `MaybeUninit::zeroed` isn't `const`.

Specific to `Id` and `RawId` types.

Const support: device id tables, possible way forward

```
trait HasRaw {
    type RawType: Copy;
    const fn into_raw(&self, index: usize) -> Self::RawType;
}

trait Driver {
    type IdType: HasRaw;
    const ID_TABLE_SIZE: usize;
    const ID_TABLE: [Self::IdType; Self::ID_TABLE_SIZE];
}

const fn null_terminated<T: Driver>(
) -> [MaybeUninit<<T::IdType as HasRaw>::RawType>; T::ID_TABLE_SIZE + 1] {
    let mut ret = [MaybeUninit::zeroed(); T::ID_TABLE_SIZE + 1];
    let mut i = 0;
    while i < T::ID_TABLE_SIZE {
        ret[i].write(T::ID_TABLE[i].into_raw(i));
        i += 1;
    }
    ret
}
```

Functions in traits cannot be const.

Unconstrained generic constant error.

Calls in constant functions are limited to constant functions.

Language

Const support: `struct file_operations` example

```
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
   // [...]
};
```

Example from socket.c:

```
static const struct file_operations socket_file_ops = {
   .owner =    THIS_MODULE,
   .llseek =   no_llseek,
   .read_iter =    sock_read_iter,
   .write_iter =   sock_write_iter,
   // [...]
};
```

Constant instance of `file_operations`.

Contains pointer to non-constant instance of `struct module`.

Language

Const support: pointers to non-const from const

Minimal example:

```
static mut MODULE: u32 = 10;

const PTR: *mut u32 = unsafe { &mut MODULE };
```

Compiler error:

```
error[E0013]: constants cannot refer to statics
 --> src/lib.rs:3:37
  |
3 | const PTR: *mut u32 = unsafe { &mut MODULE };
  |                                    ^^^^^^
  |
  = help: consider extracting the value of the `static` to a `const`, and referring to that

For more information about this error, try `rustc --explain E0013`.
error: could not compile `playground` due to previous error
```

Const support: checking offsets, simplified example

```
const fn build_error() {
    panic!("Bad offset");
}

struct IoMem<const SIZE: usize> {
    ptr: *mut u8,
}

impl<const SIZE: usize> IoMem<SIZE> {
    const fn offset_ok(offset: usize) {
        if offset >= SIZE {
            build_error();
        }
    }

    pub fn write(&self, value: u8, offset: usize) {
        Self::offset_ok(offset);
        let ptr = self.ptr.wrapping_add(offset);
        unsafe { core::ptr::write_volatile(ptr, value) };
    }
}
```

Omitted from optimised builds, resulting in link errors when offset is misused.

Runtime panic on unoptimised builds.

Const support: checking offsets, build error example

```
ld.lld: error: undefined symbol: build_error
>>> referenced by io_mem.rs:177 (/linux-rust-arm/rust/kernel/io_mem.rs:177)
>>>
gpio/gpio_pl061_rust.o:(_RNvXCsbjFALuaDPVH_15gpio_pl061_rustNtB2_11PL061DeviceNtNtCshhMfd4mY5QU_6kernel4gpio4Chip16direction_output) in archive drivers/built-in.a
```

Can we provide a better developer experience?

- Show offending file and line number.
- Suggest `try_readX`/`try_writeX` if offset is not know at compile time.
- Build-time errors even on unoptimised builds.

Architecture & GCC support

Tooling

We are constrained by LLVM arch support + LLVM support in Linux itself.

- Currently we have `arm`, `arm64`, `powerpc`, `riscv`, `riscv64`, `x86`.
- GCC support would alleviate this point.

`bindgen`

- A GCC backend for fully-GCC builds would be nice:
  - https://github.com/rust-lang/rust-bindgen/issues/1949
- GCC plugins could break ABI.
  - Though GCC plugins might be on the way out (in the kernel).

Target specification

Tooling

The kernel tweaks targets.

- We should avoid creating a kernel ↔ compiler cyclic dependency.

Custom targets are not stable.

- Unlikely to be (too tied to LLVM).

Not all target options seem to be available/exposed.

- e.g. `-mregparm` for x86.

Target specification

Tooling

Files are harder to integrate.

Flags (GCC, Clang) would be easier.

- Ideally, the same names.

Moonshot: cross-language, cross-toolchain standard way.

- Point already raised to a few LLVM & kernel folks to test the waters.
- Idea: bring all stakeholders to the `linux-toolchains` ML.

Ability to implement our own `Arc`

Language

Library

Or, in general, any standard library types that are “magic”.

Currently we are forced to use “internal to the compiler” features.

`Arc` aborts when the count overflows:

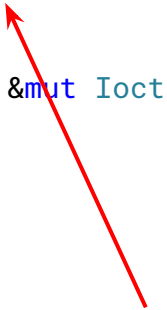
- We've been asked to avoid introducing new panics.
- Kernel's `refcount_t` saturates the count.

Other changes:

- No weak refs, always pinned, borrowing without double-dereference.

Ergonomics of operation tables: example usage

```
impl FileOperations for Process {
    type Wrapper = Ref<Self>;

    declare_file_operations!(ioctl, compat_ioctl, mmap, poll);

    // [...]

    fn ioctl(this: RefBorrow<'_, Process>, file: &File, cmd: &mut IoctlCommand) -> Result<i32> {
        todo!()
    }

    fn compat_ioctl(
        this: RefBorrow<'_, Process>,
        file: &File,
        cmd: &mut IoctlCommand,
    ) -> Result<i32> {
        todo!()
    }

    // [...]
}
```



Driver writer must specify which functions to populate.

Ergonomics of operation tables: ToUse

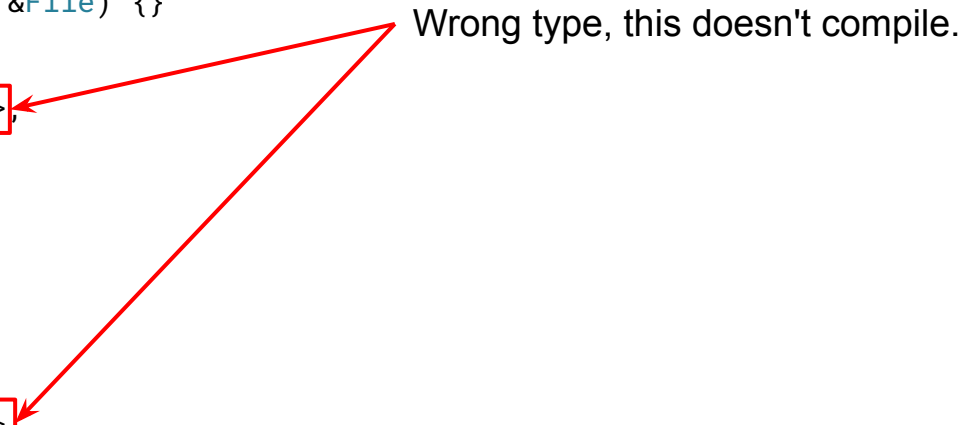
```rust
/// Represents which fields of [`struct file_operations`] should be populated with pointers.
pub struct ToUse {
    /// The `read` field of [`struct file_operations`].
    pub read: bool,
    /// The `read_iter` field of [`struct file_operations`].
    pub read_iter: bool,
    /// The `write` field of [`struct file_operations`].
    pub write: bool,
    // [...]
}

pub trait FileOperations: Send + Sync + Sized + 'static {
    /// The methods to use to populate [`struct file_operations`].
    const TO_USE: ToUse;

    // [...]
}
```

This is defined by the `declare_file_operations` macro. It is used to define which function pointers to initialise in operations table.

Ergonomics of implementing traits: "implement members"

Tooling

```
struct X;
impl FileOperations for X {
    type Wrapper = Box<Self>;

    fn release(_obj: Self::Wrapper, _file: &File) {}

    fn read(
        _this: PointerWrapper::Borrowed<'_>,
        _file: &File,
        _data: &mut impl IoBufferWriter,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }

    fn write(
        _this: PointerWrapper::Borrowed<'_>
        _file: &File,
        _data: &mut impl IoBufferReader,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }
```



Ergonomics of implementing traits: correct types

Tooling

```
struct X;
impl FileOperations for X {
    type Wrapper = Box<Self>;

    fn release(_obj: Self::Wrapper, _file: &File) {}

    fn read(
        _this: <Self::Wrapper as PointerWrapper>::Borrowed<'_>,
        _file: &File,
        _data: &mut impl IoBufferWriter,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }

    fn write(
        _this: <Self::Wrapper as PointerWrapper>::Borrowed<'_>,
        _file: &File,
        _data: &mut impl IoBufferReader,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }
```

Correct types, but too long.

Ergonomics of implementing traits: simplified types

Tooling

```
struct X;
impl FileOperations for X {
    type Wrapper = Box<Self>;

    fn release(_obj: Box<Self>, _file: &File) {}

    fn read(
        _this: &Self,
        _file: &File,
        _data: &mut impl IoBufferWriter,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }

    fn write(
        _this: &Self,
        _file: &File,
        _data: &mut impl IoBufferReader,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }
```

Simplified types. Thanks to GAT, borrowed type for Box<T> is &T.

Language

Ergonomics of type names: fully-qualified syntax

```
struct X;
impl FileOperations for X {
    type Wrapper = Box<Self>;

    fn release(_obj: Self::Wrapper, _file: &File) {}

    fn read(
        _this: <Self::Wrapper as PointerWrapper>::Borrowed<'_>,
        _file: &File,
        _data: &mut impl IoBufferWriter,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }

    fn write(
        _this: <Self::Wrapper as PointerWrapper>::Borrowed<'_>,
        _file: &File,
        _data: &mut impl IoBufferReader,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }
```

Can we omit "`as PointerWrapper`"?

Ergonomics of type names: avoiding fully-qualified syntax

```
struct X;
impl FileOperations for X {
    type Wrapper = Box<Self>;

    fn release(_obj: Self::Wrapper, _file: &File) {}

    fn read(
        _this: Self::Wrapper::Borrowed<'_>,
        _file: &File,
        _data: &mut impl IoBufferWriter,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }

    fn write(
        _this: Self::Wrapper::Borrowed<'_>,
        _file: &File,
        _data: &mut impl IoBufferReader,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }
```

Still long but more palatable.
Compiler thinks this is ambiguous.

Ergonomics of type names: lifetimes

```
struct X;
impl FileOperations for X {
    type Wrapper = Ref<Self>;

    fn release(_obj: Ref<Self>, _file: &File) {}

    fn read(
        _this: RefBorrow<'_, Self>
        _file: &File,
        _data: &mut impl IoBufferWriter,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }

    fn write(
        _this: RefBorrow<'_, Self>
        _file: &File,
        _data: &mut impl IoBufferReader,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }
```

Now using `Ref<Self>` as wrapper. Thanks to GAT, borrowed type for it is `RefBorrow<'_, Self>`.

Language

Ergonomics of type names: lifetime elision

```
struct X;
impl FileOperations for X {
    type Wrapper = Ref<Self>;

    fn release(_obj: Ref<Self>, _file: &File) {}

    fn read(
        _this: RefBorrow<Self>,
        _file: &File,
        _data: &mut impl IoBufferWriter,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }

    fn write(
        _this: RefBorrow<Self>,
        _file: &File,
        _data: &mut impl IoBufferReader,
        _offset: u64,
    ) -> Result<usize> {
        Err(Error::EINVAL)
    }
```

Type has less noise around.

Can we elide these lifetime annotations using the same rules we used for `&Self`?

Building `std` is hard

Library

Tooling

We currently build `std` for `test`.

Harder than `core` and `alloc`:

- Requires external crates and `build-std`.
- `build-std` assumes is compiling a package, ignores `RUSTFLAGS`…
- How to override the dependency graph for a custom `alloc`?

Making `test` not depend on `std` somehow?

Testing

Language
Library
Tooling

Different types (unifying KUnit and selftests):

```
#[test(host)]
fn test_that_runs_in_the_host() {
    // Something that can be tested in the host.
}

#[test(user)]
fn test_that_runs_in_the_target's_userspace() {
    // Something that must be tested in the target,
    // but the test runs in userspace.
}

#[test(kernel)]
fn test_that_runs_in_the_target's_kernelspace() {
    // Something that must be tested in the target,
    // but the test runs in kernelspace.
}
```

Testing

Language
Library
Tooling

Similarly, for doctests:

```
/// ```host
/// assert_eq!(run_some_pure_function(), 42);
/// ```
///
/// ```user
/// assert_eq!(run_some_syscall(), 42);
/// ```
///
/// ```kernel
/// assert_eq!(run_some_kernel_api(), 42);
/// ```
pub fn f() {
    // ...
}
```

Testing

Language
Library
Tooling

Wide design space:

- Compiler as a library? Plugins? …?
- Retrieving the source code: pipe it out, `TokenStream`, …?

How to make it useful for other projects?

Moonshot: `rust-analyzer` support (e.g. “▶ Run Test | Debug”).

Codegen quality: minimal source code example 1

Tooling

```
struct Example(Option<u32>);

impl Drop for Example {
    fn drop(&mut self) {
        self.0.take();
    }
}

pub fn example() -> u32 {
    Example(Some(10u32)).0.take().unwrap()
}
```

Codegen quality: output

Tooling

```
example::example:
        pushq   %rbx
        subq    $16, %rsp
        movabsq $42949672961, %rax
        movq    %rax, 8(%rsp)
        movl    $0, 8(%rsp)
        movb    $1, %cl
        testb   %cl, %cl
        je      .LBB2_1
        shrq    $32, %rax
        addq    $16, %rsp
        popq    %rbx
        retq
.LBB2_1:
```

```
example::example:
        movl    $10, %eax
        retq
```

When `unwrap_unchecked` is used instead.

Effectively a no-op as `%cl` will always be 1.

Codegen quality: example 2, minimal source code

Tooling

```
use std::ptr::read_volatile;
pub unsafe fn test1(ptr: *const u32) {
    let mut first = true;
    let mut seq = 0;
    loop {
        if !first && read_volatile(ptr) == seq {
            break;
        }
        first = false;
        seq = loop {
            let v = read_volatile(ptr);
            if v & 1 == 0 {
                break v;
            }
        };
    }
}
```

Codegen quality: example 2 output, expected

Tooling

```
example::test1:
        mov     w8, wzr
        mov     w9, wzr
        tbz     w9, #0, .LBB1_2
.LBB1_1:
        ldr     w9, [x0]
        cmp     w9, w8
        b.eq    .LBB1_4
.LBB1_2:
        ldr     w8, [x0]
        tbnz    w8, #0, .LBB1_2
        mov     w9, #1
        tbz     w9, #0, .LBB1_2
        b       .LBB1_1
.LBB1_4:
        ret
```

```
example::test1:
.LBB0_1:
        ldr     w8, [x0]
        tbnz    w8, #0, .LBB0_1
        ldr     w9, [x0]
        cmp     w9, w8
        b.ne    .LBB0_1
        ret
```

Unconditional branch to .LBB1_2

No-op

Padding: current solution, punting to developer

Language
Library
Tooling

```
/// Specifies that a type is safely writable to byte slices.
///
/// This means that we don't read undefined values (which leads to UB) in preparation for writing
/// to the byte slice. It also ensures that no potentially sensitive information is leaked into the
/// byte slices.
///
/// # Safety
///
/// A type must not include padding bytes and must be fully initialised to safely implement
/// [`WritableToBytes`] (i.e., it doesn't contain [`MaybeUninit`] fields). A composition of
/// writable types in a structure is not necessarily writable because it may result in padding
/// bytes.
pub unsafe trait WritableToBytes {}

pub trait IoBufferWriter {
    /// Writes the contents of the given data into the io buffer.
    fn write<T: WritableToBytes>(&mut self, data: &T) -> Result {
        todo!()
    }
    // [...]
}
```

Deserialising data: current solution

Library

```
/// Specifies that a type is safely readable from byte slices.
///
/// Not all types can be safely read from byte slices; examples from
/// <https://doc.rust-lang.org/reference/behavior-considered-undefined.html> include `bool`
/// that must be either `0` or `1`, and `char` that cannot be a surrogate or above `char::MAX`.
///
/// # Safety
///
/// Implementers must ensure that the type is made up only of types that can be safely read from
/// arbitrary byte sequences (e.g., `u32`, `u64`, etc.).
pub unsafe trait ReadableFromBytes {}

// SAFETY: All bit patterns are acceptable values of the types below.
unsafe impl ReadableFromBytes for u8 {}
unsafe impl ReadableFromBytes for u16 {}
unsafe impl ReadableFromBytes for u32 {}
unsafe impl ReadableFromBytes for u64 {}
unsafe impl ReadableFromBytes for usize {}
```

Language

Rust specification

The Rust reference is not complete/normative yet.

Part of the kernel community values having a language specification.

Specially useful to have for writing subtle unsafe code and tooling.

It may also help the GCC Rust effort and vice versa.

Branded types: check once, reuse many times

Language
Library
Tooling

```
if self.dbbuf_update_and_check_event(inner.sq_tail, 0) {
    if let Some(res) = self.data.resources() {
        let _ = res.bar.try_writel(inner.sq_tail.into(), self.db_offset);
    }
}
```

Simplified syntax, no need to 'consume' `Result`.

We could then use a variant of `writel`, which has the same cost as C.

This can be checked only once on the constructor.

Branded types: locking patterns, RCU

Language
Library
Tooling

```
struct Process: exists 'a {
    inner: SpinLock<'a, ProcessInner>,
    mm: RcuPointer<'a, Option<Ref<MemoryManager>>>,
}

fn read(process: &Process) {
    let rcu_guard = rcu::read_lock();
    let mm = process.mm.get(&rcu_guard);
}

fn write(process: &Process) {
    let old;
    {
        let guard = process.inner.lock();
        old = process.mm.swap(None, &guard);
    }
    // `drop` for `old` calls `synchronize_rcu`.
}
```

Example syntax: `'a` is not part of the `Process` type.

`mm` is readable and usable while RCU read is locked.

`process.mm` is writable while `process.inner` is locked, not any `process` though, exactly the same one

Language

Function context restrictions (“colored `unsafe`”)

Atomic vs. sleepable context.

C side has runtime checking with `might_sleep`.

Could Rust provide compile-time checking?

```
fn called_from_atomic_context() {
   this_may_sleep(); // ideally a compile-time error
   does_not_sleep(); // OK
}

fn called_from_sleepable_context() {
   this_may_sleep(); // OK
   does_not_sleep(); // OK
}
```

Language

Function context restrictions (“colored `unsafe`”)

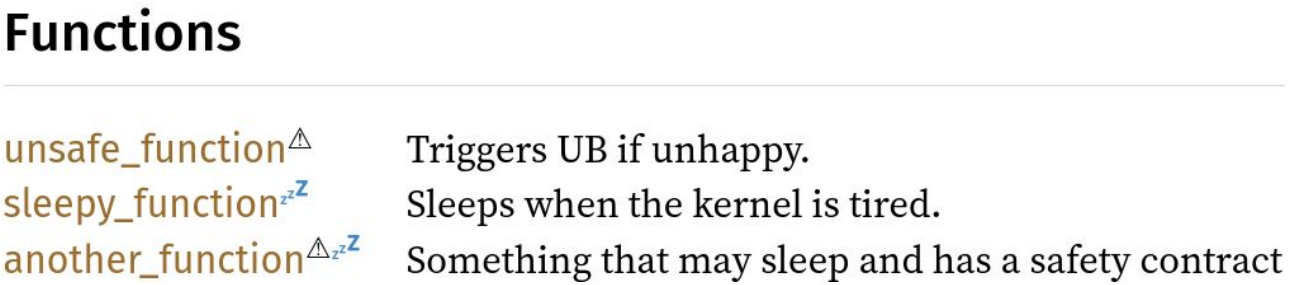
Could we automatically infer it under some rules?

```
// Manual tagging of functions that definitely sleep,
// or perhaps the other way around.
sleepy fn sleep() {
   // ...
}

// Ideally automatically inferred.
fn this_might_sleep(b: bool) {
   if b {
       sleep();
   }
}

// FFI? Function pointer types? ...?
```

Function context restrictions (“colored `unsafe`”)

Tooling

`rustdoc` could learn about them too:



Others

Language
Library
Tooling

Bindgen support for C macros and `inline` functions.

Implied bounds.

`static_assert`, `build_assert`.

“Custom prelude” (e.g. avoiding `#![no_std]` etc. in every module).

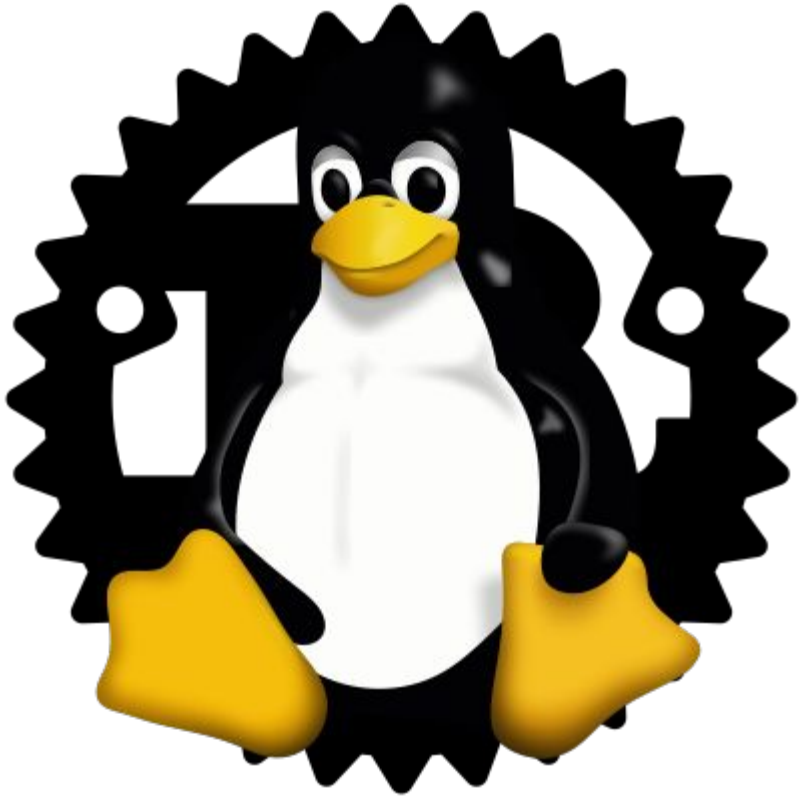
Support for cross-language documentation (external references file).

Improved language/ergonomics for intrusive data structures.

Const `offset_of`, supporting compile-time-known fields of unsized types.

Thank you!

Questions?



Rust for Linux

Miguel Ojeda
Wedson Almeida Filho
Alex Gaynor