Rust Safety Standard

Increasing the Correctness of `unsafe` Code

Benno Lossin <benno.lossin@proton.me>

September 7th 2024

1. What is Safety Documentation?
2. Why do we need Safety Documentation?
3. Status of Safety Documentation
4. Reasons and Goals for Standardization
5. Discussion

What is Safety Documentation?

- Requirements:

```rust
/// # Safety
///
/// `ptr` must have been returned by a previous
/// call to [`Arc::into_raw`]. Additionally, it
/// must not be called more than once for each
/// previous call to [`Arc::into_raw`].
pub unsafe fn from_raw(ptr: *const T) -> Arc<T>;
```

- Justifications:

```rust
let ptr = Arc::into_raw(arc);
// SAFETY: `ptr` comes from `Arc::into_raw` and
// we only call this function once with `ptr`.
let arc = unsafe { Arc::from_raw(ptr) };
```

---

all code self-authored or taken from the kernel with modifications to improve presentability

Why do we need Safety Documentation?

C does not have it, why do we need it for Rust?

- **Higher stakes:**
  - Safe Rust has the "no memory bugs guarantee*".
  - **But!** `unsafe` code is a big asterisk!
  - We want to uphold the no-memory-bugs privilege!
    (that's the strongest argument in favor of Rust)
- **More complexity:**
  Correct `unsafe` code is more difficult to get right:
  - Field drop order
  - Fat pointers
  - Lifetime annotations
  - Ownership
- Opportunity to catch errors when writing safety documentation.

Status of Safety Documentation

- Mandated by review since the very first patches,
- They are of course not perfect, things do slip through: `https://lore.kernel.org/rust-for-linux/20240905-rust-lockdep-v1-1-d2c9c21aa8b2@gmail.com/`
  - One reason: miscommunication of what C requires.
- Almost all `unsafe` blocks and functions have safety documentation,
- Will soon enable a clippy lint: `https://lore.kernel.org/rust-for-linux/20240904204347.168520-1-ojeda@kernel.org/`
- But: inconsistent style!
  - “`ptr is valid, ...`” vs “`ptr is valid, non-null, ...`”
  - “`struct invariants`” vs “`type invariants`”

Status of Safety Documentation

- The style, wording and quality varies a lot:

```rust
/// # Safety
///
/// The provided pointer must point at a valid
/// struct of type `Self`.
#[inline]
unsafe fn raw_get_work(
    ptr: *mut Self
) -> *mut Work<T, ID> {
    let ptr = ptr as *mut u8;
    // SAFETY: The caller promises that the
    // pointer is valid.
    unsafe { ptr.add(Self::OFFSET).cast() }
}
```

```rust
pub fn into_raw(self) -> *const T {
    let ptr = self.ptr.as_ptr();
    core::mem::forget(self);
    // SAFETY: The pointer is valid.
    unsafe { core::ptr::addr_of!((*ptr).data) }
}
```

Reasons and Goals for Standardization

- Goal: Always use the same wording for the same situation.
- Better documented requirements, justifications, invariants and guarantees:
  - Make authors write as little as possible,
  - Give readers extensive explanations.
- Easier to write: No need to come up with wording yourself.
- Easier to learn: Only one way needs to be learned.
- Leave no room for misinterpretations: everyone knows the semantics of the comments

How to get involved

- view the RFC: `https://lore.kernel.org/rust-for-linux/20240717221133.459589-1-benno.lossin@proton.me/`
- Discuss with me on Zulip: `https://rust-for-linux.zulipchat.com/`
- Join the LPC talk.
- And of course this discussion.

Thanks for your Attention

Discussion