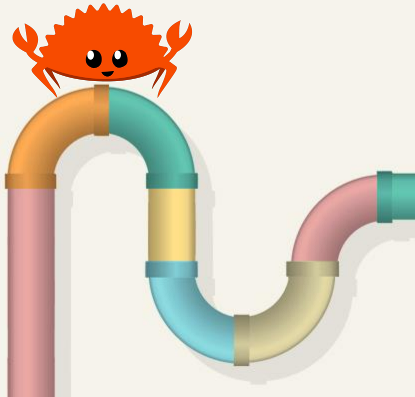
LINUX
PLUMBERS
CONFERENCE
August 24-28, 2020

Barriers to in-tree Rust

Linux Plumbers Conf 2020 - LLVM MC

John Baublitz, Nick Desaulniers,
Alex Gaynor, Miguel Ojeda,
Geoffrey Thomas, Josh Triplett



What is Rust? Why should we do this?

- Systems programming language well-suited to low-level/kernel dev
- "A better C" in the way the kernel wants C to be
  - No GC, errors via return, OO via ops structs, no implicit casts, no FFI overhead
- Similar syntax (and model) to C
- Safe and unsafe subsets ➔ easier to review tricky code
- More information about types and validity ➔ no safe undefined behavior
  - Memory-safe: no use-after-frees, buffer overflows, use of uninitialized memory, etc.
  - Data race-free

… also, 2/3 of security bugs are memory safety

This session will not be an intro to the Rust language - see Further Resources for more information.

What are we not suggesting?

Rewriting 25 million lines of C in Rust.

(Let's add the option of using Rust for new code.)

What does Rust Linux kernel code look like?

```rust
fn read(&self, file: &File, buf: &mut UserSlicePtrWriter, _offset: u64) -> KernelResult<()> {
    let mut chunkbuf = [0; 256];
    while !buf.is_empty() {
        let len = min(chunkbuf.len(), buf.len());
        let chunk = &mut chunkbuf[0..len];
        if file.flags().contains(FileFlags::NONBLOCK) {
            getrandom_nonblock(chunk)?;
        } else {
            getrandom(chunk)?;
        }
        buf.write(chunk)?;
    }
    Ok(())
}
```

from https://github.com/alex/just-use

Kbuild, rustc, cargo

Definitely need integration with Kbuild/Kconfig

Use Cargo as a builder? Some nice things

Use Cargo for dependencies? NO! In tree code only. Not going to make builds require the network

Cargo supports explicitly building offline-only, using in-tree code only

We may use external projects but they should be copied into the tree and be editable by maintainers. Plenty of precedent (genksyms, libgcc_s routines, device tree compiler, ACPI compiler, zlib implementation, GCC plugins, etc.)

Discussion

A rough plan towards adding Rust

1. .
2. .
3. .
4. .
5. .

If you want to chat more:

#rust https://chat.2020.linuxplumbersconf.org/channel/rust

We'll be around in general, if there is interest, let's do scheduled meetings:

- Tonight 5:00 PM PDT (00:00 UTC)
- Tomorrow morning 6:30 AM PDT (13:30 UTC)

https://github.com/Rust-for-Linux

Further resources

- Alex and Geoff's implementation + link to our LSSNA talk
  https://github.com/fishinabarrel/linux-kernel-module-rust
  several examples in tests/*/
- Josh's implementation
  https://github.com/jbaublitz/knock-out
- Just Use /dev/urandom
  https://github.com/alex/just-use
- Rust for Linux
  https://github.com/Rust-for-Linux
- Data on memory unsafety
  https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/
- Android kernel driver bugs
  https://events.static.linuxfound.org/sites/events/files/slides/Android-%20protecting%20the%20kernel.pdf

extra slides

Architecture support

rustc uses LLVM - major limiting factor

https://github.com/fishinabarrel/linux-kernel-module-rust/issues/112

tl;dr:
OK: arm, arm64 hexagon, mips, powerpc, riscv, s390, sparc64, x86
LLVM OK, rustc needs port: arc, sparc32
A few of the rest have work-in-progress towards LLVM or rustc

mrustc: experimental Rust-to-C compiler, less checking (but that's okay)

LLVM C backend: experimental, recent work by Julia team

GCC, Clang, LLVM, rustc

Backend: rustc uses LLVM just like Clang does

Frontend: rust-bindgen uses libclang to parse C headers and generate Rust function/structure declarations with the same ABI

If you want to be very safe about matching compilers, compile your kernel with Clang, use the same libclang and libllvm versions in bindgen and rustc, so you have only one C parser and only one codegen. (Rust + GCC works fine though)

Do in-tree Rust drivers need to require CONFIG_CC_IS_CLANG?
(Are there enough users who are using Clang kernels to make that worthwhile?)

bindgen

You can write bindings by hand:

```
#[repr(C)] struct list {
    data: u64,
    next: *const list,
}
extern "C" {
    fn length(l: *const list) -> u32;
}
```

Bitfields, #ifdefs, typedefs, … compiler plugins. Use libclang.

Treat it like a build tool? Treat it like a dependency and vendor it?

Rust versions, stable vs. nightly

Rust enables certain unstable features only on nightly builds to preserve the backwards-compatibility promise for stable releases (never Rust 2.0)

For now, we need nightly Rust to compile our own libcore + liballoc

Nice to have: https://github.com/fishinabarrel/linux-kernel-module-rust/issues/41 (mostly allocator APIs and constant functions, both making good progress)

How quickly do we need to get to stable?

Firefox stays close to latest stable Rust - do we expect problems? Can we compile using the existing distro version? https://lkml.org/lkml/2020/7/12/88

Rust team will add the kernel to their pre-release CI ("crater")

“Linus would never…”

https://lkml.org/lkml/2020/7/10/1261

"I'd want the first rust driver (or whatever) to be introduced in such a simple format that failures will be obvious and simple."

https://www.theregister.com/2020/06/30/hard_to_find_linux_maintainers_says_torvalds/ (OSS+ELC NA)

"having interfaces to do those, for example, in Rust... I'm convinced it's going to happen. It might not be Rust. But it is going to happen that we will have different models for writing these kinds of things, and C won't be the only one."

Target triple

We added the "x86_64-linux-kernel" target (= triple + -mcmodel=kernel -mno-x87 etc.):

https://github.com/rust-lang/rust/blob/master/src/librustc_target/spec/x86_64_linux_kernel.rs
https://github.com/rust-lang/rust/blob/master/src/librustc_target/spec/linux_kernel_base.rs

The existing Clang build uses the normal userspace triple + CFLAGS

Need to add support for other arches beyond x86_64. Can we point rustc at target.json files inside arch/? (That's an unstable feature)

Rust will accept patches, and may build libcore for us, which would enable stable Rust

Rust core libraries

libcore: basic types, no OS or allocation (arrays, iterators, Option, Result, etc.)

liballoc: things that need a heap but no OS (String, Vec, etc.), depends on a malloc (we have this wired up to kmalloc(GFP_KERNEL))

libstd: OS support (files, processes, sockets), depends on a libc

In userspace, libstd can be linked as a dynamic library, but usually isn't because there's no stable Rust ABI. We don't care about stable ABIs :)

Make a rust.ko, EXPORT_SYMBOL all the dynamic symbols of libcore and liballoc, every Rust user depends on it

I heard you couldn't write XYZ in Rust

Safe Rust is based around a "const/mutable" aka "borrowed/owned" (think rwlock) reference system. Data cannot be mutated (or deleted) while someone else has a reference. Designs where data has multiple owners cannot be expressed *entirely* in **safe Rust**.

That's okay! It's fine to use **unsafe Rust** - certainly no worse than C :)
Write a small `unsafe {…}` block, code-review it well, call it from safe functions.

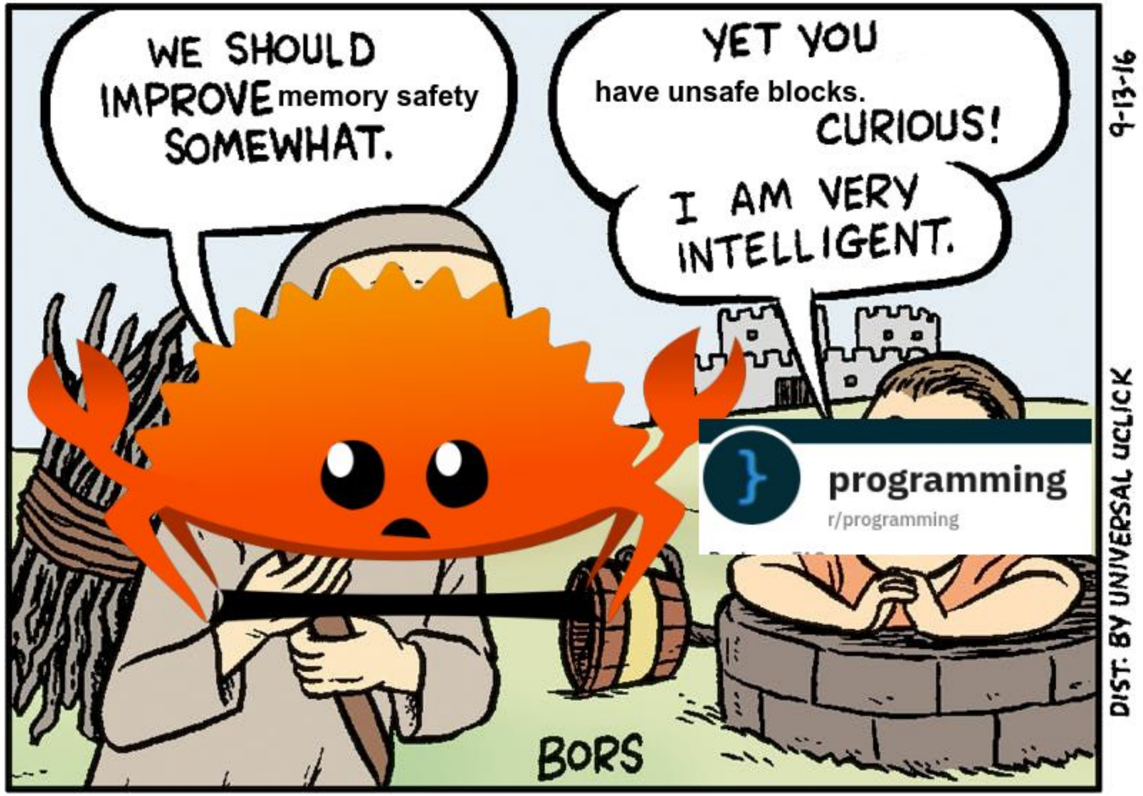
Doubly-linked lists, intrusive data structures, etc. have standard implementations

RCU: working on it (it actually maps nicely onto the reader/writer model, and the userspace library Crossbeam resembles RCU)

memes if needed

ReWrItE iT iN rUsT - NO!

We're not here to tell you to convert your entire working kernel to Rust


WE SHOULD IMPROVE memory safety SOMEWHAT.
YET YOU have unsafe blocks. CURIOUS!
I AM VERY INTELLIGENT.
programming
r/programming
BORS
9-13-16
DIST. BY UNIVERSAL UCLICK

(from @jam1garner)