Rust For Linux pain points

Kangrejos ‘25

Joel Fernandes, Nvidia

Obscure mod post errors (due to build_assert)

- Simple way to repro: Blowing the bar region offset.

```
bar.write32(0x12345678,  <large addr outside of bar’s bounds>);
```

```
ERROR: modpost: "rust_build_error" [drivers/gpu/nova-core/nova_core.ko] undefined!
```

- Very common to incorrectly write to large addresses.
- Unclear where the error happened if change/patch causing the error is large.
  - Also developer may wrongly think a different build_assert caused it.
- Wastes developer time as it results in guess work, trial/error.

Obscure mod post errors (due to build_assert)

Can also happen if you pass variables to C bindings

```
let mut offset = 0;

pr_err("{}", offset);

io.write32(self.0, offset);   //-> build_assert!(Self::offset_valid::<U>(offset, SIZE))
```

```
MODPOST Module.symvers

ERROR: modpost: "rust_build_error" [drivers/gpu/nova-core/nova_core.ko] undefined!
```

Obscure mod post errors (due to build_assert)

Good news, I might have a fix — move the FFI call into a closure. Works!

Obscure mod post errors (due to build_assert)

- Klint handling this is the best but..
- My colleague Mikko Perttunen has a partial remedy:
  - Print file number and line number in the build_assert invocation
  - grep all object files looking for the messages, if compiler didn't optimize them away, we'll find them.
  - Does not give trace back, so still limited, but it is something.
  - https://github.com/cyndis/linux/commit/585f40e0c01b371ddfe0ffd8ac138e46b5e1bfce
  - Can we do something like this? Where-in, the linker/modpost stages parse object files to give a more meaningful build error.

Unpredictable stack overflows

- Large structures or several medium size structures can intermittently overflow stack.
  - It seems in Rust this is much easier to do than C
  - Adding print statements can cause overflows with perfectly working code

Unpredictable stack overflows

- Rust compiler can create intermediate copies more than needed.
  - Copying sometimes optimized away - but not always
  - Rust debug options / optimization levels change stack usage / copying behavior
  - This causes failures sometimes but not always.
  - Bad - we want to fail early

Unpredictable stack overflows

- Stack overflows in rust can just over write the stack (or best shows stack guard issues)
- Stack guard storage page may be hit:

```
[  126.809010] BUG: TASK stack guard page was hit at 000000004ac8a24c

               (stack is 00000000794a5987..00000000f9f6bfd9)

[  126.809013] Oops: stack guard page: 0000 [#1] SMP

[  126.809033] Call Trace:

[  126.809034]  <TASK>

[  126.809035]  _RNvNtCs1h9PaebWwla_4core3fmt5write+0x18c/0x200
```

(Thanks to my colleague Alistair Popple for running these experiments)

Unpredictable stack overflows

Without CONFIG_VMAP_STACK on x86_64 you just get random memory corruption. Eg:

```
[   16.943146] BUG: kernel NULL pointer dereference, address: 0000000000000102

[...]

[   16.960464] RIP: 0010:kmem_cache_free_bulk.part.0+0x1e5/0x4f0
```

(Full OOPs is trimmed)

(Thanks to my colleague Alistair Popple for running these experiments)

Unpredictable stack overflows

- Idea: Can we fail at compile time if stack frame usage is high
    - Stack overflows in rust can just over write the stack.
    - Stackoverflow seems something that should be related to memory-safety.

- Next: Disabling Copy attribute in nova’s bindgen (discussing with team)
    - What is Copy attributed behavior in bindgen for regular kernel C bindings?

Unpredictable stack overflows

```
fn pass_struct_by_value(x: MyStruct) {  println!("We pass the struct by value: {:?}", x);  }

fn pass_struct_by_reference(x: &MyStruct) {  println!("We pass the struct by value: {:?}", x); }

#[derive(Debug)]

struct MyStruct {

    num: [u32; 16],

}

pub fn main() {

    let x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];

    let y = MyStruct { num: x };


    // Moved because MyStruct doesn't implement Copy

    pass_struct_by_value(y);
```

Unpredictable stack overflows

```
pub fn main() {
    let x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
    let y = MyStruct { num: x };

    // Moved because MyStruct doesn't implement Copy
    pass_struct_by_value(y);

    // Won't work because struct y was moved/passed by value above
    // pass_struct_by_reference(&y);
}
```

Unpredictable stack overflows

- C warnings - large stack frames - at compile time
- Devs avoid borrow checker issues absent in C, by copying.
- Safe on top of unsafe code creates deeper stack frames.
- Stack overflow crashes may not be obvious from the logs: Overwrites memory
    - Probably should not since it is a memory safe language.

Rust calling methods on enum objects

Ran into this in VBIOS code.

```
enum BiosImage { PciAt(PciAtBiosImage), .. }

impl BiosImage {
    /// Check if this is the last image.
    fn is_last(&self) -> bool {
        let base = self.base();
}
```

And say we want to implement base() method common to all objects. Calling base() requires ugly match statements.

Currently it requires ugly macro: `see next slide.`

Rust calling methods on enum objects

```
macro_rules! bios_image {( $($variant:ident $class:ident),* $(,)? ) => {
        // BiosImage enum with variants for each image type
        enum BiosImage {
            $($variant($class)),*
        }
        impl BiosImage {
            fn base(&self) -> &BiosImageBase {
                match self {
                    $(Self::$variant(img) => &img.base),*
                }
            }
```

Rust calling methods on enum objects

1. Can we implement receiver Trait?
2. Field/Method projections?
3. Something else?

Other minor pain points

- Auto-demangling for kernel logs
- Build dependencies, sometimes updating core rust code, but building the driver does not result in the core rust for Linux code re-building.
    - example make a change to rust/
    - Then make M=driver/gpu/nova-core/ , this wont rebuild core rust for Linux.

Buffer ownership between driver and HW (time permitting)

- *Problem statement: - “GSP ring buffer for example, at any point in time, part of buffer is owned by HW and part owner by driver. This changes continuously as the shared buffer is filled and consumed by both parties.”*
- CoherentAllocation APIs return entire buffer, not “part which is unused by driver”
- This is hard to implement in those APIs, because this is device driver dependent.

Rust compiler version issues (time permitting or discuss in hallway)

- Policy is to use minimal kernel version, CLIPPY warns if your kernel code uses newer compiler features.
    - Fix is to not use the new compiler features.
- Some kernel options requires newer compiler (violating the kernel policy, that only a minimum kernel version is required)
    - Alistair: to confirm which config options.
    - This kind of violates the policy that kernel code does not use newer compiler features.
- Would this really work for backports of fixes to older kernels? Backports will have to done manually to satisfy “minimum version”