JOURNEY OF A C KERNEL ENGINEER STARTING A RUST DRIVER PROJECT

LPC '24

Danilo Krummrich, Red Hat

NOVA

- DRM GPU driver for NVIDIA GPUs with GPU System Processor (GSP) using the Rust programming language
  - GSP provides a firmware interface that abstracts the actual hardware
- long term, Nova is intended to serve as the successor of Nouveau for all GSP-based GPUs

MOTIVATION

1. Why start a new driver project for NVIDIA GPUs?
    - create a welcoming place for new contributors
    - make the driver more accessible for new contributors
    - clean up page table management for Vulkan
    - reduce complexity by supporting GSP only
    - be maintainable in the long term; reduce maintainance cost

MOTIVATION

2. What are the reasons for choosing Rust?
    - Rust's language features help avoiding memory safety bugs and improve general maintainability
        - Rust in the Linux ecosystem - Miguel Ojeda
        - pin-init: Solving Address Stability in Rust - Benno Lossin
        - Using Rust in the binder driver - Alice Ryhl
    - GSP firmware interface is entirely unstable
        - Rust (Procedural Macros) significantly help with that
    - serve as example for next generation DRM drivers
    - help out with bringing Rust into DRM and the kernel in general

LEARN RUST

- personal background: kernel engineer for more than 10 years
- read through the The Rust Programming Language
  - learn about the basics, such as:
    - ownership, references, borrowing,
    - generic types, traits, lifetimes,
    - etc.

LEARN RUST

```rust
#[pin_data(PinnedDrop)]
pub struct Registration<T: RegistrationOps> {
    #[pin]
    reg: Opaque<T::RegType>,
}

impl<T: RegistrationOps> Registration<T> {
    /// Creates a new instance of the registration object.
    pub fn new(name: &'static CStr, module: &'static ThisModule) -> impl PinInit<Self, Error> {
        try_pin_init!(Self {
            reg <- Opaque::try_ffi_init(|ptr: *mut T::RegType| {
                // SAFETY: `try_ffi_init` guarantees that `ptr` is valid for write.
                unsafe { ptr.write(T::RegType::default()) };

                // SAFETY: `try_ffi_init` guarantees that `ptr` is valid for write, and it has
                // just been initialised above, so it's also valid for read.
                let drv = unsafe { &mut *ptr };

                T::register(drv, name, module)
            }),
        })
    }
}
```

LEARN RUST

```
error[E0277]: `core::cell::UnsafeCell<kernel::pci::Bar<16777216>>` cannot be shared between threads
   --> drivers/gpu/drm/nova/driver.rs:77:17
    |
77  |     type Data = Arc<NovaData>;
    |                 ^^^^^^^^^^^^^ `core::cell::UnsafeCell<kernel::pci::Bar<16777216>>` cannot be sl

error[E0596]: cannot borrow data in dereference of `kernel::sync::Arc<gpu::Gpu>` as mutable
  --> drivers/gpu/drm/nova/driver.rs:54:9
   |
54 |         gpu.init()?;
   |         ^^^ cannot borrow as mutable

error[E0277]: the trait bound `impl init::PinInit<revocable::Revocable<T>>: init::Init<revocable::Re
    --> rust/kernel/devres.rs:97:13
     |
97   | /             pin_init!( DevresInner {
98   | |                 data <- Revocable::new(data),
     | |                 ---- required by a bound introduced by this call
99   | |             }),
     | |_____________^ the trait `init::Init<revocable::Revocable<_>, _>` is not implemented for `i
     |
```

LEARN RUST

STRATEGY

- analysis of existing Rust infrastructure
- chicken-egg problem in upstreaming (some) Rust abstractions
  - drivers require Rust abstractions to exist
  - abstractions require a user before they can be upstreamed
- difficult for new projects (e.g. Asahi) → start Nova with just a stub driver
  - reference implementation and justification for Rust abstractions
  - demonstrates how they fit together
  - basis for further Nova development

STRATEGY

- General Rust abstractions (Device / Driver, PCI, Devres, I/O)
  - based on preceding work from Wedson Almeida Filho et al.
- DRM Rust abstractions (Device / Driver, File, IOCTL, GEM)
  - based on preceding work from Lina Asahi
- Rust Allocator
  - Generic Kernel `Allocator` interface (`Kmalloc`, `Vmalloc`, `KVmalloc`)
  - Kernel implementation of `Box` and `Vec` types using the `Allocator` trait
- Nova (stub) driver

DEVICE / DRIVER ABSTRACTIONS

- mailing list discussion with Greg KH on the device / driver, PCI, I/O patch series
  - Greg proposed to keep driver registration in C
- met with Greg at Kangrejos '24
  - held a talk about the device / driver abstractions
  - → agreed to move forward with those abstractions

DEVICE / DRIVER ABSTRACTIONS

```
struct sample_info {
        u32 data;
};

static const struct sample_info qemu_info = { .data = 0  };
static const struct sample_info foo_info  = { .data = 42 };

static const struct pci_device_id sample_pci_ids[] = {
        { PCI_DEVICE_DATA(PCI_VENDOR_ID_REDHAT, PCI_DEVICE_ID_QEMU_PCI_TESTDEV, &qemu_info) },
        { PCI_DEVICE_DATA(PCI_VENDOR_ID_REDHAT, PCI_DEVICE_ID_FOO,              &foo_info)  },
        {}  /* Don't forget the sentinel. */
};

static int sample_pci_probe(struct pci_dev *pdev, const struct pci_device_id *id)
{
        /* Do we have the correct type? */
        struct sample_info *info = (struct sample_info *)id->driver_data;

        /* Can info ever be NULL? */
        dev_info(&pdev->dev, "%u\n", info->data);

        [...]
}
```

DEVICE / DRIVER ABSTRACTIONS

```
pub(crate) struct Driver;

#[derive(Debug)]
pub(crate) struct Info(u32);

impl pci::Driver for Driver {
    define_pci_id_table! {
        Info, [
            (pci::DeviceId::new(PCI_VENDOR_ID_REDHAT, PCI_DEVICE_ID_QEMU_PCI_TESTDEV), None),
            (pci::DeviceId::new(PCI_VENDOR_ID_REDHAT, PCI_DEVICE_ID_FOO), Some(Info(42)))
        ]
    }

    fn probe(pdev: pci::Device, info: Option<&Self::IdInfo>) -> Result {
        dev_info!(pdev.as_ref(), "{:?}\n", info);

        [...]
    }
}
```

DEVICE / DRIVER ABSTRACTIONS

```rust
/// Abstraction for `bindings::pci_device_id`.
#[derive(Clone, Copy)]
pub struct DeviceId {
    /// Vendor ID
    pub vendor: u32,
    /// Device ID
    pub device: u32,
    /// Subsystem vendor ID
    pub subvendor: u32,
    /// Subsystem device ID
    pub subdevice: u32,
    /// Device class and subclass
    pub class: u32,
    /// Limit which sub-fields of the class
    pub class_mask: u32,
}

impl DeviceId {
    /// Zeroed `bindings::pci_device_id`.
    // SAFETY: The all-zero byte-pattern is valid for `bindings::pci_devi
    pub const ZERO: bindings::pci_device_id = unsafe { core::mem::zeroed(
    const PCI_ANY_ID: u32 = !0;

    /// Equivalent to the PCI_DEVICE macro.
    pub const fn new(vendor: u32, device: u32) -> Self {
        Self {
            vendor,
            device,
            subvendor: DeviceId::PCI_ANY_ID,
            subdevice: DeviceId::PCI_ANY_ID,
            class: 0,
            class_mask: 0,
        }
    }

    /// Convert `DeviceId` to raw `bindings::pci_device_id`.
    ///
    /// `offset` is the offset of `ids[i]` to `id_infos[i]` within `IdArra
    pub const fn to_rawid(&self, offset: usize) -> bindings::pci_device_i
        let mut raw = Self::ZERO;

        raw.vendor = self.vendor;
        raw.device = self.device;
        raw.subvendor = self.subvendor;
        raw.subdevice = self.subdevice;
        raw.class = self.class;
        raw.class_mask = self.class_mask;
        raw.driver_data = offset as _;

        raw
    }
}

/// A zero-terminated PCI device ID array.
#[repr(C)]
pub struct IdArray<U, const N: usize> {
    ids: [bindings::pci_device_id; N],
    sentinel: bindings::pci_device_id,
    id_infos: [Option<U>; N],
}
```

```rust
impl<U, const N: usize> IdArray<U, N> {
    /// Creates a new instance of the ID array.
    ///
    /// The contents are derived from the given identifiers.
    #[doc(hidden)]
    pub const fn new(ids: [bindings::pci_device_id; N], infos: [Option<U>; N]) -> Self {
        Self {
            ids,
            sentinel: DeviceId::ZERO,
            id_infos: infos,
        }
    }

    /// Returns an `IdTable` backed by `self`.
    ///
    /// This is used to essentially erase the array size.
    pub const fn as_table(&self) -> IdTable<'_, U> {
        IdTable {
            first: &self.ids[0],
            _p: PhantomData,
        }
    }

    /// Returns the offset of `ids[i]` to `id_infos[i]` within `IdArray`.
    #[doc(hidden)]
    pub const fn get_offset(index: usize) -> usize {
        let id_size = core::mem::size_of::<bindings::pci_device_id>();
        let info_size = core::mem::size_of::<Option<U>>();

        id_size * (N - index + 1) + info_size * index
    }
}

/// A device ID table.
///
/// The table is guaranteed to be zero-terminated.
#[repr(C)]
pub struct IdTable<'a, U> {
    first: &'a bindings::pci_device_id,
    _p: PhantomData<&'a U>,
}

impl<U> AsRef<bindings::pci_device_id> for IdTable<'_, U> {
    fn as_ref(&self) -> &bindings::pci_device_id {
        self.first
    }
}

/// Converts a comma-separated list of pairs into an array with the first element. That
/// discards the second element of the pair.
///
/// Additionally, it automatically introduces a type if the first element is warpped in
/// braces, for example, if it's `{v: 10}`, it becomes `X { v: 10 }`; this is to avoid r
/// the type.
#[macro_export]
macro_rules! first_item {
    ($id_type:ty, $(({$($first:tt)*}, $second:expr)),* $(,)?) => {
        {
            type IdType = $id_type;
            [$(IdType($($first)*),)*]
        }
    };
    ($id_type:ty, $(($first:expr, $second:expr)),* $(,)?) => { [$($first,)*] };
}
```

```rust
/// Converts a comma-separated list of pairs into an array with the second element. That is,
/// discards the first element of the pair.
#[macro_export]
macro_rules! second_item {
    ($(({$($first:tt)*}, $second:expr)),* $(,)?) => { [$($second,)*] };
    ($(($first:expr, $second:expr)),* $(,)?) => { [$($second,)*] };
}

/// Counts the number of parenthesis-delimited, comma-separated items.
#[macro_export]
macro_rules! count_paren_items {
    (($($item:tt)*), $($remaining:tt)*) => { 1 + $crate::count_paren_items!($($remaining)*)
    (($($item:tt)*)) => { 1 };
    () => { 0 };
}

#[macro_export]
#[doc(hidden)]
macro_rules! define_pci_id_array {
    ($id_type:ty, $($args:tt)*) => {{
        const fn new<U, const N: usize>(
            ids: [$crate::pci::DeviceId; N],
            infos: [Option<U>; N]) -> $crate::pci::IdArray<U, N>
        {
            let mut raw_ids = [$crate::pci::DeviceId::ZERO; N];

            // The offset of `ids[i]` to `id_infos[i]` within `IdArray`.

            let mut i = 0usize;
            while i < N {
                let offset = $crate::pci::IdArray::<U, N>::get_offset(i);

                raw_ids[i] = ids[i].to_rawid(offset);
                i += 1;
            }

            $crate::pci::IdArray::<U, N>::new(raw_ids, infos)
        }

        new($crate::first_item!($id_type, $($args)*), $crate::second_item!($($args)*))
    }}
}

/// Define a const PCI device ID table.
#[macro_export]
macro_rules! define_pci_id_table {
    ($id_type:ty, [ $($args:tt)* ]) => {
        type IdInfo = $id_type;
        const ID_TABLE: $crate::pci::IdTable<'static, $id_type> = {
            const ARRAY: $crate::pci::IdArray<$id_type, { $crate::count_paren_items!($($args
                $crate::define_pci_id_array!($id_type, $($args)*);
            ARRAY.as_table()
        };
    };
}
pub use define_pci_id_table;
```

RUST ALLOCATOR

```
// Current API based on Rust's stdlib with extentions, defined as:
//
// struct Box<T: ?Sized>(NonNull<T>)
let val = Box::new(4, GFP_KERNEL)?;

// New API with generic `Allocator` trait, defined as:
//
// struct Box<T: ?Sized, A: Allocator>(NonNull<T>, PhantomData<A>)
let val = Box::<_, Kmalloc>::new(4, GFP_KERNEL)?;
let val = Box::<_, Vmalloc>::new(4, GFP_KERNEL)?;
let val = Box::<_, KVmalloc>::new(4, GFP_KERNEL)?;

// New API with type alias.
let val = KBox::new(4, GFP_KERNEL)?;
let val = VBox::new(4, GFP_KERNEL)?;
let val = KVBox::new(4, GFP_KERNEL)?;
```

RUST ALLOCATOR

```rust
pub unsafe trait Allocator {
    fn alloc(layout: Layout, flags: Flags) -> Result<NonNull<[u8]>, AllocError> {
        unsafe { Self::realloc(None, layout, flags) }
    }

    unsafe fn realloc(
        ptr: Option<NonNull<u8>>,
        layout: Layout,
        flags: Flags,
    ) -> Result<NonNull<[u8]>, AllocError>;

    unsafe fn free(ptr: NonNull<u8>) {
        let _ = unsafe { Self::realloc(Some(ptr), Layout::new::<()>(), Flags(0)) };
    }
}
```

```c
/* Old API (declarations simplified) */

void * krealloc(const void *p, size_t size, gfp_t flags);

/* Different signature; different semantics */
void * kvrealloc(const void *p, size_t old_size, size_t new_size, gfp_t flags);
```

```c
/* New API (declarations simplified) */

void * krealloc(const void *p, size_t size, gfp_t flags);
void * vrealloc(const void *p, size_t size, gfp_t flags);
void * kvrealloc(const void *p, size_t size, gfp_t flags);
```

RUST ALLOCATOR

- revealed an issue where `__GFP_ZERO` is not properly honored by `krealloc`

```
/* Allocate memory in 64-byte kmalloc bucket. */
buf = kzalloc(64, GFP_KERNEL);
memset(buf, 0xff, 64);

buf = krealloc(buf, 48, GFP_KERNEL | __GFP_ZERO);

/* After this call the last 16 bytes are still 0xff. */
buf = krealloc(buf, 64, GFP_KERNEL | __GFP_ZERO)
```

- Solution: handle `__GFP_ZERO` the same way we handle the KASAN redzone

STATUS

- Lyude Paul
  - rVKMS (Rust version of VKMS)
    - driver to develop DRM KMS Rust abstractions
  - Rust abstractions for IRQ management
- Dave Airlie
  - Rust VFIO userspace driver
    - playground and PoC for initial and upcoming firmware abstractions
  - Rust tool to parse GSP header files and generate the corresponding Rust structures

STATUS

- Abdiel Janulgue
  - Rust abstractions (ELF header, scatterlist)
  - Nova (load firmware into GPU)
- Philipp Stanner
  - DRM GPU scheduler
    - documentation
    - lifetime issues [1] [2]
    - preparation for Rust abstractions

STATUS

- Danilo Krummrich
  - driving and coordinating the project
  - General Rust abstractions (Device / Driver, PCI, Devres, I/O)
    - based on preceding work from Wedson Almeida Filho et al.
  - DRM Rust abstractions (Device / Driver, File, IOCTL, GEM)
    - based on preceding work from Lina Asahi
  - Rust Allocator
    - Generic Kernel `Allocator` interface (`Kmalloc`, `Vmalloc`, `KVmalloc`)
    - Kernel implementation of `Box` and `Vec` types using the `Allocator` trait
  - Nova (stub) driver

QUESTIONS?