RVKMS and KMS drivers in Rust

What I’ll be talking about

- A general overview of the RVKMS and the bindings so far
- An honest opinion on what I think of Rust, and why I think it’s a game changer for the kernel. And it does /not/ talk about memory safety. Not even once. (Except to tell people to stop focusing on that :)

How this started

- We’ve been working on a new Nvidia kernel driver: nova
- Nova is written in Rust, particularly because:
  - GSP firmware doesn’t have a stable ABI
  - Trying to handle this in C (we’ve tried) is /hard/
- We also think that through upstreaming rust drivers, we can prove Rust’s readiness and projects like Asahi can have an easier time upstreaming their work

How this started

- Device-bring up for a new driver is hard and takes time
- All of this also implies needing to write Rust bindings for kernel drivers
- Thanks to the Asahi project – we already have a number of bindings we can reuse
- They just need to be upstreamed

How this started

- Suprisingly, this comes with one notable exception: Asahi's KMS driver is one of the few parts written in C
- This means there aren't any KMS bindings for us to use
- And Nova is far too early in development to have a KMS driver written for it yet
- But KMS is a large enough surface we wanted to start work on this sooner than later, and ideally in parallel to nova

Then came RVKMS

- We decided while Nova was getting ready, I would try to port a KMS driver into Rust to come up with a set of bindings
- Due to how simple VKMS is, we made that the driver to port to Rust
- VKMS is a “virtual” KMS driver, meaning it pretends to be a display device
  - It is also capable of CRC generation
  - It also supports writeback connectors

RVKMS

- RVKMS is the rust port of the VKMS driver. So far it’s very early in development driver-wise.
- Currently the things I have working on the driver are:
  - Basic KMS driver registration (registering a KMS device, creating connectors/planes/etc. And registering with fbcon)
  - Vblank emulation using hrtimers
- Planned (eventually, hopefully :)
  - CRC generation
  - Connector writeback

RVKMS

- Despite RVKMS being very early in development, I’ve made wonderful progress with the KMS bindings
- Have done my best to structure the API not just around VKMS’s needs, but also what I’ve seen from drivers like i915 and nouveau.
- A lot more time went into bindings then the actual RVKMS driver!
- But it’s still worked well for testing these things :)

My goal with the KMS bindings

- Prevent UB within safe code of course
- Make incorrect KMS API implementations nearly impossible
- Be ergonomic – protection from mistakes shouldn’t make code messier, inflexible, or more difficult to write
- Limit conditional handling to only where it’s needed and makes sense (thanks Sima!)

My goal with the KMS bindings

- Subclassing objects should be trivial
  - (we'll talk about what this means shortly)
- Outside of supporting legacy atomic helpers, these bindings should be atomic only
  - This makes synchronization a lot easier than with legacy drivers
  - We don't want legacy modesetting drivers anyway

What I have so far – KMS bindings

- These bindings work on top of the DRM bindings from Asahi and Danilo
- Unlike in C, the kernel crate is mostly in control of the order of events during device registration
- The main entrypoint for enabling KMS support on a device is implementing the kernel::drm::kms::Kms trait

kernel::drm::kms::Kms

- mode_config_info
  - Fills various static information in drm_device.mode_config, e.g. min/max resolution, cursor capabilities, etc.
- create_objects
  - Provides temporary access to a special UnregisteredKmsDevice type
  - This type allows creation of static mode objects like CRTCs and planes
  - Non-static objects like connectors can be created here as well

kernel::drm::kms::Kms

- In the future, we'll add hooks for customizing what happens during the initial modeset
- This would be tasks like reading back display hardware state (for fastset support)
- For the time being, we just do drm_mode_config_reset()

kernel::drm::kms::KmsDriver

- Drivers implementing Kms get a KmsDriver implementation automatically.
- Makes KMS-dependent bindings only visible to KMS drivers at compile time (Device<KmsDriver>)
- Without this we'd need to both:
  - Stub all KMS methods where possible (easy in some places, doesn't work well in many others)
  - Use a Lot of Option<>s in places they don't make sense

Modesetting objects

- DRM has a concept of a “mode object”
- Typically, this is any type of object that:
  - Is exposed to userspace through an object ID
  - May have a refcount and can be created at any time
  - Or may have no refcount, and can only be made before registration
- They also have properties sometimes but we haven’t gotten here yet in rvkms yet!

Modesetting objects

- We present a ModeObject trait for such objects
- Ref-counted objects like connectors are easy: refcounting fits quite nicely with Rust’s lifetime requirements.
  - We provide RcModeObject to reduce boilerplate for refcounting.
- Static objects are more challenging – they share the lifetime of a Device
  - This isn’t easy to map to a rust lifetime
  - Examples: CRTCs, Planes, Encoders

StaticModeObject

- To workaround this, we provide StaticModeObject and KmsRef
- KmsRef acts as a ref-count on the parent Device, while providing access to the static mode object
- Fulfills rust’s lifetime requirements and allows “owned” references of static mode objects
- Might be replaced with Devres in the future? Still unsure

Planes, CRTCs, Connectors, Encoders

- More complex than one might expect
- Very few drivers actually use these structures unmodified
- More often, they’re embedded within driver-private structures
  - Example: vkms_crtc embeds drm_crtc
- Usually these structures contain private driver state
  - Display state tracked outside of a single atomic commit
  - Static driver info (capabilities, port mappings, etc.)
- Drivers very often have more then one subclass of an object (i915, nouveau)

Asahi does it again!

- Luckily, this isn’t the only place such requirements exist
- For the GEM infrastructure, this type of subclassing is also very common
- Luckily Asahi is a GEM driver, and has to subclass GEM objects for its own state in the same manner
- Less variable than KMS (drivers don’t have that many subclasses for gem, I think?), but still a wonderful starting point

DriverPlane, DriverCrtc, etc.

- We introduce DriverPlane, DriverCrtc, DriverConnector and DriverEncoder traits
- A driver is allowed to have multiple implementations of these traits
- Driver data can be stored in them in two ways:
  - Passing immutable data to each object's constructor (mutexes etc. not needed) in create_objects
  - At any other point, via Send+Sync containers (Mutex<T>, SpinLock<T>, etc.)

Fully typed and opaque objects

- In KMS drivers written in C, drivers typically switch between a common representation and subclassed representation
  - E.g. drm_crtc vs. vkms_crtc
- This is also possible with our bindings, each core mode object has two variants
  - A “fully typed” interface (e.g. Crtc<DriverCrtc>)
    Provides access to driver-private data and common DRM methods
  - An “opaque” interface (e.g. OpaqueCrtc<KmsDriver>)
    Only provides access to common DRM methods

Fully typed and opaque objects

- We do the same thing for atomic states
  - Each have fully-typed and opaque representations
  - They can store private data in the same way (though Send+Sync is not required)
- Type conversion is only needed for driver-private data
- Opaque objects can be fallibly “upcasted” at runtime to fully-typed objects
  - Handled the same way as C (e.g. vtable pointers used for identity comparison)
  - Don’t worry – we added the ability to get consistent vtable memory locations through a new #[unique] macro!

Still TBD, further subclassing

- But what if a driver needs further classing?
  - e.g. i915 has things like intel_connector, intel_hdmi, intel_dp, etc.
- I -think- this will be very easy just using rust's trait system as usual, but this is still TBD
- This is something needed by drivers like i915

Atomic commits

- Things diverge a bit when we get to handling atomic states
- Rust’s data aliasing rules says you can have only one of these at any given time:
  - Infinite immutable refs
  - Only one mutable ref
- If each atomic_check/update/etc. callback only affected a given plane/crtc/etc. in question – this would be very easy
- But many drivers iterate through the state of other objects in atomic callbacks – not just the object the callback belongs to

Atomic commits

- I originally implemented this just using references, but this worked out very poorly:
  - Assume you’re in an atomic_check for a plane
  - You have a &mut to the PlaneState, which borrows from a &mut AtomicState
  - How do you even iterate through other objects then? You already have a mutable reference, you can’t take another.
  - And you definitely can’t take an immutable reference either

Atomic commits

- The second time, I decided to take some inspiration from RefCell
  - This is a simple rust API for handling situations where the data aliasing rules aren’t ideal
  - Immutable and mutable borrows still exist, but they’re checked at runtime
  - This is all available just through immutable references

AtomicStateMutator

- Act as a wrapper around AtomicState and allows “borrowing” the state of a mode object at runtime:
  - Only one borrow for a state may be taken at a time
  - But multiple different states may be borrowed at the same time
- Borrowing is fallible – but we still make this pretty ergonomic
  - Callbacks like atomic_check pass a borrowed state as an argument
  - Iterators for unborrowed states will be implemented in the future
  - Most drivers iterate over the **other** object’s states in their callbacks
- Example: BorrowedCrtcState<CrtcState> or BorrowedCrtcState<OpaqueCrtcState>

AtomicStateComposer

- Wrapper around an AtomicStateMutator
- Represents an atomic state that hasn’t finished the atomic_check phase
- Same thing, but also allows adding new objects into an atomic state
- Can easily be used for allowing drivers to perform arbitrary commits in-kernel that aren’t triggered by userspace (e.g. CRC enablement)

High level commit callbacks

- E.g. atomic_commit_tail
- Have additional constraints:
  - Certain orders of operation never make sense: e.g. CRTCs can’t be enabled in a commit before they’re disabled in the same commit
  - We have to ensure the state can’t be mutated once it’s made visible outside of the commit context (e.g. commit_hw_done)
  - DRM doesn’t know the state of borrows, that’s all on the rust side
- Breaking these requirements would result in UB, we can’t have that!

Type state patterns!!!

- Not entirely unique to rust, but not common in other languages
- Allows you to encode the runtime state of something into compile-time state
- We've already used two:
  - AtomicStateComposer → AtomicStateMutator
  - UnregisteredKmsDevice → Device
- This gives us a very powerful tool for enforcing API correctness

AtomicCommitTail

- Another AtomicState wrapper – passed to the optional Kms::atomic_commit_tail method
- Lets you control the order in which commits happen safely by providing tokens for each step of the process. Each token is consumed to retrieve the next one
- All CommittedToken types need to be passed to .commit_hw_done(), and a CommittedAtomicState must be returned from atomic_commit_tail to prove this happened
- .commit_modeset_disables() → .commit_modeset_enables() → .commit_hw_done()
  ModesetsReadyToken → DisablesCommittedToken → EnablesCommittedToken
- .update_planes() → .commit_hw_done()
  PlaneUpdatesReadyToken → PlaneUpdatesCommittedToken
- atomic_commit_tail() → .commit_hw_done() → return
  AtomicCommitTail → CommittedAtomicState → return

AtomicCommitTail

- (TODO) Can be trivially expanded for more fine grained control
  - e.g. specifying the order CRTCs are updated, planes, etc.
- Literally makes it impossible to write an incomplete atomic_commit_tail that compiles
- Every step takes a &mut to AtomicCommitTail
  - (TODO) Allows AtomicStateMutators to be taken out from an AtomicCommitTail
  - Ensures a Mutator can’t exist while a commit occurs (rust’s data aliasing rules)

Optional driver features?

- KMS drivers have a lot of optional features
- Vblanks are a great example:
  - Everything uses them
  - But not all hardware actually has a hw vblank interrupt, so sometimes it needs to be emulated and the driver doesn’t implement this
- Could we gate these features like we do with KmsDriver? (e.g. only hw vblank drivers can call vblank methods).
- YES WE CAN! Just add more traits

VblankSupport

- Hardware vblank supports can be added by implementing this trait on a DriverCrtc
- Exposes vblank-exclusive methods through an automatic DriverCrtcVblank trait (Crtc<DriverCrtcVblank>)
- We can expand this same pattern to any optional piece of functionality
- Since we control the driver init and registration process – this is trivial for us to handle without the user needing to do it
- Not much else to go over here – you implement the traits, etc etc.

Special thanks to

- The Asahi Project – a huge amount of our work depends on their bindings!
- Maira Canal

My experience with Rust

What I won’t talk about

- Memory safety. Seriously. It is the least convincing point.
- We know C is unsafe, that it’s a problem, and we know that’s inherent to C, we’ve been writing it for decades. It’s not a point worth talking about.
- It comes across as talking down to kernel devs.
- Considering we know all this: more often then not, focusing on this point comes across as talking down to kernel developers.
- I honestly avoided touching rust for years because the only point people ever presented to me was “it’s memory safe!”. So is Python, we don’t write kernels in it though!
- If you want to hear better arguments for rust in the kernel, this part is for you :3

Rust can be a kernel maintainer.

- Maintainers don’t scale, and a huge part of being a maintainer is trying to stop bad patterns in code.
- It is time consuming, and requires you re-explain yourself repeatedly and hope you didn’t miss something. It can make you snippy, it can burn through your motivation depending on what kind of contributors you’re working with.
- But, in this sense, Rust is also a kernel maintainer! We can enforce code patterns far, FAR more then C can.
- Type state patterns like the ones in atomic_commit_tail are a wonderful example of this – they have little, and more often times **no** runtime cost
- The upfront cost is large, the learning curve is present and it takes time – but the potential for saving time long term is astounding.

Unsafe doesn’t change this much

- Of course, the first thing people think about this: but what about unsafe? You can violate any language invariant there.
- In my experience, it’s been a bit shocking how little this matters.
- If anything – I feel that having unsafe code is also a tremendously powerful tool for stopping bad code patterns just like type state patterns in safe code.
- The compiler will complain if you don’t write a SAFETY comment explaining yourself, so you already have someone’s explanation on any invariant they’re breaking.
- It also forces you to write a safety contract, meaning any time someone violates it in a contribution – all you have to do is point to the contract. No more explaining yourself.

Unsafe doesn’t change this much

- Even in the few times I’ve had to debug RVKMS, having sections of “it’s probably broken here” from unsafe saved me a huge amount of time.
- Despite interacting with C APIs, it is unbelievable how little time I actually spent debugging this driver.
- Just about every time I’ve worked on a driver in C, it’s gone like this:
  - Write a bunch of changes that are sound
  - Spend the next day, maybe multiple days, debugging subtle errors all over the place (you forgot to check something for NULL, you forgot to initialize something, you forgot to make sure something was thread-safe)
  - Repeat.
- With rust, over the few months I worked on this driver and bindings: I think I spent a total of ~24-32 hours debugging it across that entire timespan. _Everything_ else was designing and dealing with the tougher problems.

It doesn’t even feel clunky

- Before I worked with rust, I was pretty off-put by things like no NULL. What, I have to handle both conditions EVERY time?
- Types? This many types? That sounds like a nightmare
- But in reality – rust’s ergonomic enough you barely think about this stuff a lot of the time once you’ve finished designing a set of bindings.
- Not even just that – it almost always feels /obvious/ what the right design should be.
- Just about every construct rust forces you to deal with has a million different shortcuts for making it legible and simple.
- A lot of the time even with how many types there are, you rarely need to keep close track of them.

The scope is smaller then it seems

- Another common misunderstanding I see: comparisons like C++ vs. Rust.
- I don’t like these, because it ignores the fact Rust is a shockingly small language. Most languages like C++ are, ok, but often times feel like frameworks rather than a language.
- A lot of the stdlib takes from the same KISS philosophy that C stdlibs tried to do, but because it’s iterated on – it actually works!
- There’s rarely that many different ways to actually do something. And once you get used to rust, the correct way almost always feels obvious.

So, one last question?

- Would you rather:
  - Repeat yourself a million times on a mailing list to stop people making the same exact mistakes in their code using their mail client?
  - Have the compiler do it.
- Give rust a try :).