

VirtIO without the Virt

Towards implementations in hardware

Michael S. Tsirkin

Distinguished Engineer
Chair of Virtio TC



1

Hello! I'm Michael Tsirkin. I work at Red Hat as a distinguished engineer and I'm a chair of the Virtio Technical committee.

Today I'm going to talk about hardware implementations of VirtIO.

I plan to describe some interesting challenges that surface when you try to unite both software and hardware under a single interface umbrella.

I will answer some frequently asked questions and talk about best practices.

Hopefully this is going to be an interesting study into how to build software interfaces.

Why hardware virtio?

- Originally a software interface
 - Guest/Hypervisor interface for VMs
 - vhost: userspace/kernel interface
 - vhost/virtio-user: userspace/userspace
- 2012 virtio multimedia hardware offload
"cool and random" – Rusty
- 2017 vdpa: hardware interface



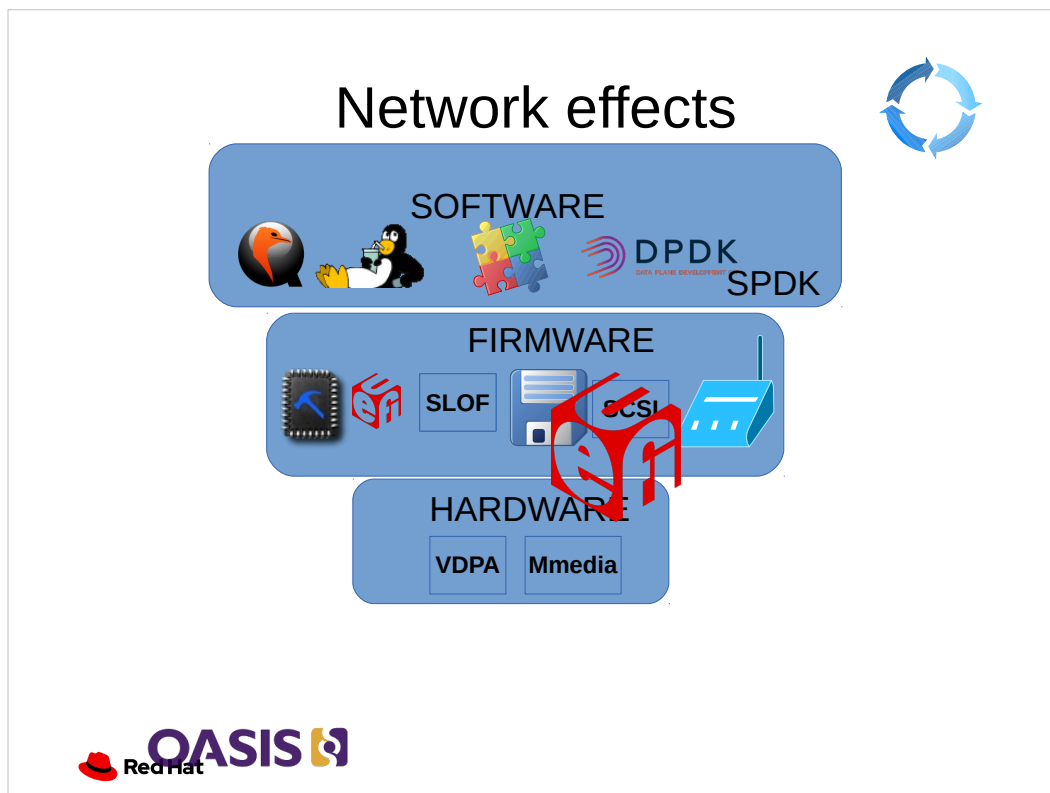
2

Virtio was started by Rusty Russell as a Guest to hypervisor interface. Pretty soon afterwards Linux gained a vhost module making it also a kernel/userspace interface. However it was still mostly used with KVM.

So when people first started using virtio outside of virtualization for a multimedia hardware offload, Rusty called this development "cool and random".

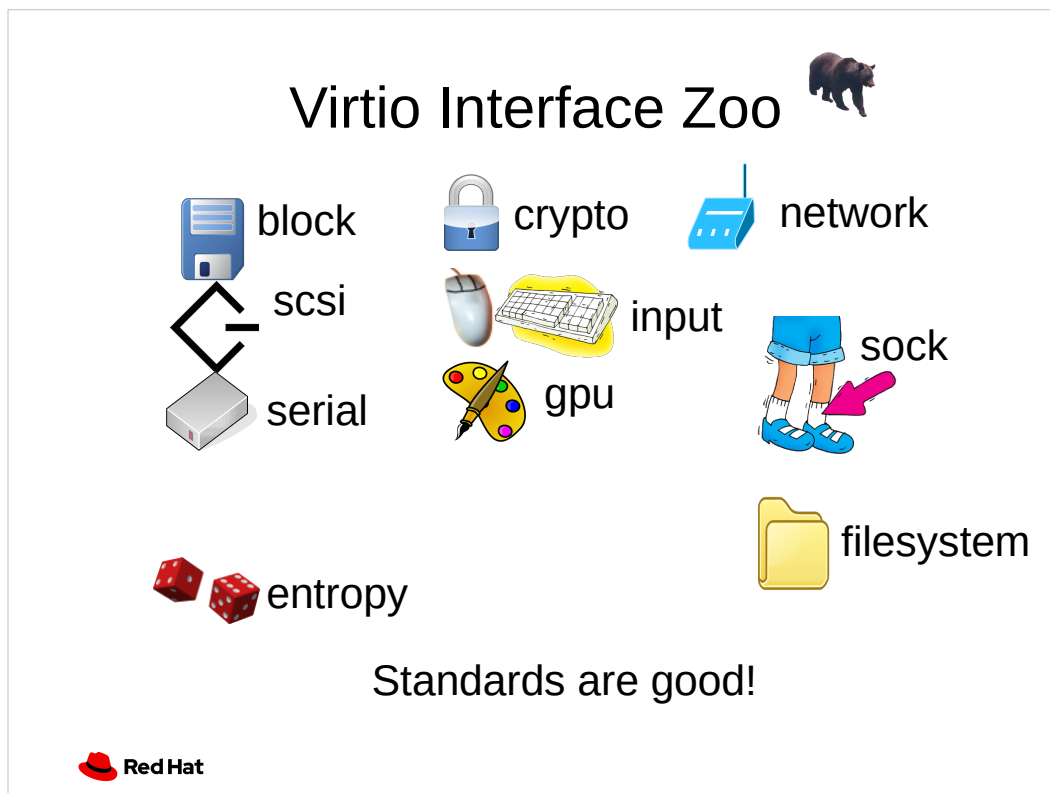
However this turned out not to be an isolated event: In 2017 Intel presented a virtio net hardware accelerator card.

So not a random event – multiple hardware vendors are choosing to build interfaces around virtio. But do they choose virtio specifically? It's a valid question.



Some of the explanation is undoubtedly network effects (like facebook): there is a large number of projects already using virtio. This slide shows only some of them that I'm aware of and that are under active development.

Note how virtio is present at all levels of the stack: QEMU supports a huge variety of virtio devices, and you can use windows or linux kernel level drivers, or userspace dpdk and spdk drivers. There are firmware drivers in ROMs used by the bios, uefi and slof for network, block and scsi virtio devices, and hardware devices using virtio are emerging. So by choosing to use virtio a new device gets to leverage some of this ecosystem as opposed to writing all of this software from scratch.



How much code you can reuse would of course depend on which virtio device you are building, but there's such a large amount of existing functionality that it's likely that one can leverage some of it with no or little change.

This slide shows just some of the available functionality: older device types have been originally developed at IBM by Rusty Russell, Anthony Ligouri and others in early 2000s and are now deployed very widely so by re-using them you immediately gain a huge potential user-base.

So standards are good – I guess I do not have to prove it in this forum. Still let me supply a bit more examples of what one gains by following an interface standard such as virtio.

Motivation: userspace drivers

- Drivers often packaged with application
Unlike kernel: New devices require app
- Kernel has no visibility into device state



- Link with a virtio library and forget
- Snapshot/restore can be made to work (WIP)



For example userspace drivers are pretty popular: direct hardware access allows bypassing system call overhead.

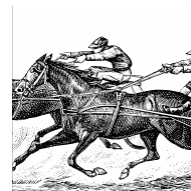
Unfortunately with userspace drivers an application is linked with the driver and so it isn't always easy to add support for new hardware. Building hardware to an existing interface sidesteps this problem.

Snapshot/restore is a challenge when using userspace drivers since kernel (doing the snapshotting) does not have visibility into device operation. With virtio it is possible in theory to load a kernel level driver, have that take over and snapshot/restore the device state according to the virtio spec for the application.

Motivation: VM guests

- Pass-through for performance
- Cross-host migration without guest changes
- Multi-vendor clusters supported
- Live migration also works

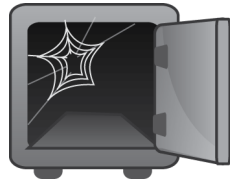
Hypervisor aware of guest visible state



For VMs, pass through of hardware devices to VM guests is very helpful for performance. However this results in a set of problems that's very similar to that of userspace drivers. In this case solutions with virtio based hardware devices have been presented. These allow moving VM guests across hosts with save/restore of HW state based on the virtio specification. Moreover, live migration without stopping VM guests has been reported to work, and even between device implementations by different vendors.

Motivation: overcommit

- Hardware
- Memory
- Switching to software
- Possibly live (WIP)



Scraping the barrel: you are typically limited to one driver per device, and have to pin memory used by a hardware device.

Using virtio one can utilize the existence of software device implementations to address both limitations: you can switch – potentially without downtime – between a hardware and software device, enjoying software goodies such as better scalability and memory swap but with a higher CPU utilization.

Motivation: bugs

- Who's to blame for a crash?
Buggy card or buggy driver?
- Swap in a different device and find out!
- Software implementations available
- Fix it in the right place



Debugging: is the device or a driver to blame for a problem? With two devices supporting virtio you can swap in a different device – either a hardware device from another vendor or a software device implementation. And if it turns out to be the device, you can point at the virtio spec and demand a fix from the vendor.

Virtio Properties



- Forward and Backward Compatibility
- PCI for Device Discovery
- Notifications through device memory/MSI
- Virtqueue Communication
- Reasonable Specification Process

Let's drill down ...

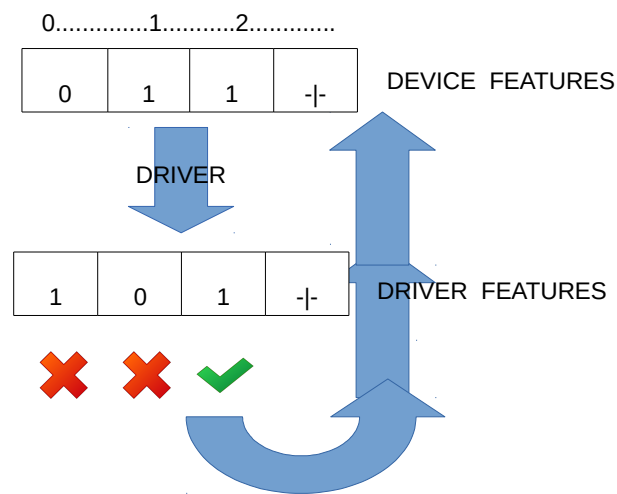


The above was pretty generic and might apply to other standard interfaces. But that's not all. For example consider a new device – what is the advantage of building the interface around virtio as opposed to rolling my own? There will be no installed userbase but I think that there are still properties of virtio that make it an attractive choice for a new project.

One is virtio's strong focus on compatibility: both forward and backward. Another is ability to use PCI for discovery. The Virtqueue mechanism within virtio has some attractive properties that I'm going to describe.

Finally, there is a reasonably lightweight process in place to extend the virtio specification and I'm going to talk a little bit about that too.

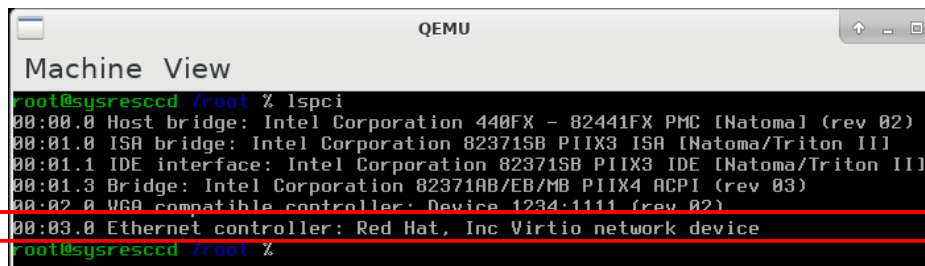
Virtio feature negotiation



Compatibility is handled by virtio feature negotiation, which is one of the most successful aspects of virtio. Each feature, is mapped to a bit in a feature bitmap. There are 2 of these: device and driver features. Driver and device exchange these bitmaps, and do a logical AND. Only bits set in both masks are considered negotiated. Negotiated bits are sent back to device.

Each time the virtio specification is changed, the change is always gated by a feature bit. This allows cherry-picking of new features by devices, as opposed to updating the device each time a new spec version is out.

PCI based discovery



```
Machine View
root@sysresccd /root % lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton III]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton III]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)
00:03.0 Ethernet controller: Red Hat, Inc Virtio network device
root@sysresccd /root %
```

- Standard VendorID/DeviceID registers donated by Red Hat for use by Virtio
- Use these values → drivers will bind to device



11

Another important property of virtio is its ability to leverage PCI as a discovery and configuration mechanism. It's important simply because PCI support is widely deployed in existing software and hardware.

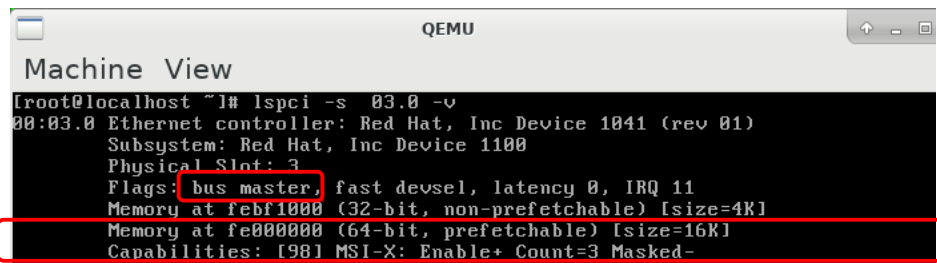
Here I run the `lspci` command within a VM, its output lists a bunch of PCI devices, among them a virtio device.

Red Hat has donated some vendor/device ID pairs for use by virtio, so if you implement a device with one of these IDs, existing drivers will bind and work with your device.

That's a lot of software that does not have to be rewritten.

What else does it take to move the virtio functionality from host software to a hardware PCI device?

PCI based communication



```
Machine View
[root@localhost ~]# lspci -s 03.0 -v
00:03.0 Ethernet controller: Red Hat, Inc Device 1041 (rev 01)
Subsystem: Red Hat, Inc Device 1100
Physical Slot: 3
Flags: bus master, fast devsel, latency 0, IRQ 11
Memory at febf1000 (32-bit, non-prefetchable) [size=4K]
Memory at fe000000 (64-bit, prefetchable) [size=16K]
Capabilities: [98] MSI-X: Enable+ Count=3 Masked-
```

- Give device access to guest memory
- Forward guest writes to device memory
- Forward device interrupts to guest



12

Specifically, once the driver binds to the device, what does it take for it to talk to the device?

To answer this question, let's take a look at a dump of the PCI configuration of a virtio device.

I highlighted the relevant parts of the dump.

First we see that bus mastering is enabled.

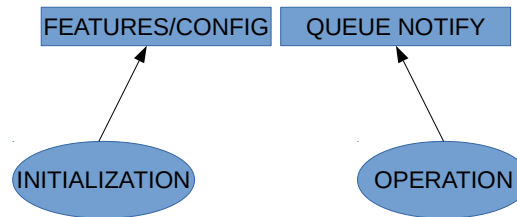
This means that device can access the system RAM using DMA. So for a hardware implementation to replace software, we need to give the device access to memory using DMA.

Second, we see that device has some memory, which can be accessed by the guest software. We'll need to map that into guest memory.

Finally, we see a message signalled interrupt capability. Device can signal interrupts, and we need to forward these to the guest.

Virtio PCI

- PCI config space
- Device memory (BAR)



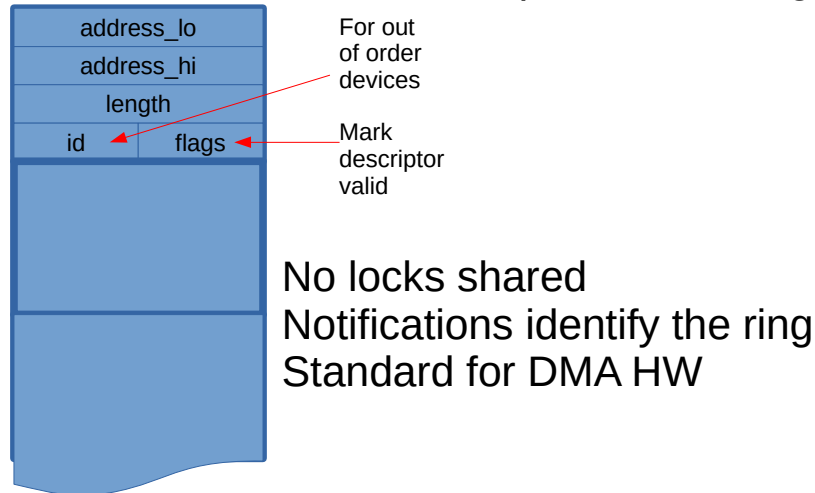
Since I mentioned device memory, I'd like to mention in passing that there are actually two parts to it. One part is device configuration. This includes the feature bits that I mentioned previously, together with the PCI configuration that I showed on the previous slide this is used for initialization.

During data path operation, the only region of device memory that's accessed by the driver is the notification region.

The idea is that during device operation driver manipulates data in queues that reside in RAM. These are accessed by device using DMA. Queue notification is what starts this access.

Virtqueue ring

Device and driver write descriptors into a ring



14

A central part of virtio is the virtqueue – a queue in RAM memory on which requests are placed. We support several formats, here's the most recent one: It consists of a ring of descriptors, where each descriptor includes a physical address and length. Descriptors also include an ID which allows supporting out of order completions, and a flags field that marks a descriptor as available or used.

The key observations here are:

First that if you look at a typical hardware device, ring structures with descriptors are very common so this queue structure is probably generic enough to support similar functionality to an arbitrary hardware device.

And second, that the only device memory access is the driver notification, which is a very limited interface that is easy to implement in hardware.

Platform issues

- Hardware Virtio device behind a PCI bus:

wmb()
dma_wmb()

- Software Virtio device:

interrupt
smp_wmb()

- VIRTIO_F_ORDER_PLATFORM



15

Hardware accelerators generally use less CPU than software virtio implementations. However it turns out that on some platforms we can actually implement optimizations that are specific to software devices. For example, when talking to hardware you generally need to use strong memory barriers to make sure that device accesses the data in the correct order. This includes both device notifications and DMA by device. However, a software device running on another CPU in an SMP configuration can typically get by with a weaker SMP memory barrier. Further, a notification typically involves sending an IPI to another CPU which already includes a memory barrier – its expensive but at least an extra barrier is not really required.

A hardware device needs to a feature bit flag to make sure this optimization is disabled

Platform issues (cont)

- Hardware Virtio device: Access can be limited:
 - Cache flushes
 - Bounce buffer
 - IOMMU
- Software Virtio device:
 - SMP cache coherency
- VIRTIO_F_ACCESS_PLATFORM



Hardware generally does not benefit from cache synchronization to the same level as another CPU.

It is common to have a need to flush caches when you talk to hardware, copy data to a bounce buffer, or set up IOMMU mappings.

Software bypasses all that but a hardware device must set flags to make sure it is handled correctly.

Simple Hardware Virtio Devices

- Implement VIRTIO_F_VERSION_1 ✓
- Implement VIRTIO_F_ORDER_PLATFORM ✓
- Implement VIRTIO_F_ACCESS_PLATFORM ✓
- Don't implement the legacy interface ✗
 - No way to support the above
 - IO BAR – VMEXIT

OK so you are building a hardware Virtio device, what are the current best practices?

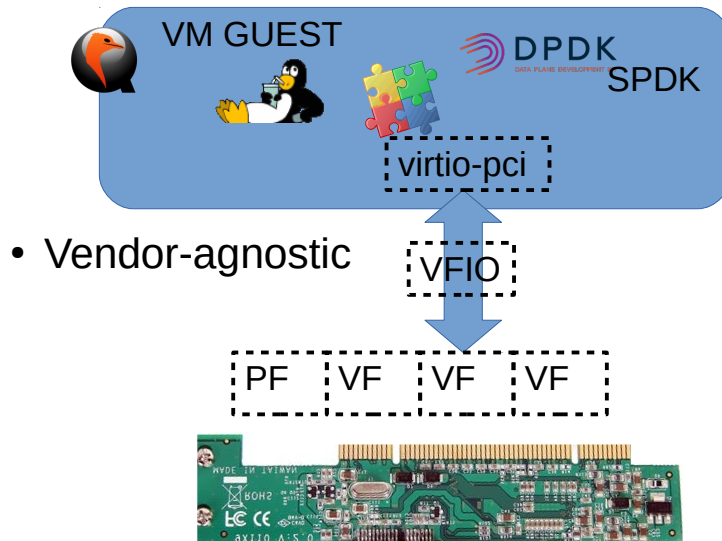
First you need to implement at least Virtio 1.0. That's a dependency of the rest of the features.

You should set the ORDER_PLATFORM flag. This makes sure memory accesses are ordered correctly for a hardware device. It doesn't really change anything for x86, but you don't know that your hardware will be used on x86.

Set the ACCESS_PLATFORM flag. Without this flag, driver assumes device runs on another CPU. E.g. it assumes an IOMMU bypass, so the device won't function correctly behind an IOMMU.

I advise against supporting the pre-1.0 legacy interface. Some systems still lack 1.0 support but effort is better spent adding this support in software. In particular, legacy devices use IO (not memory) which is slow on lots of systems, causing Vmexits etc

Pass-through (full offloading)



If you do all this, which software do you need to use to offload virtio functionality from a hypervisor to your device?

This slide focuses on a VM, but it's exactly the same for a userspace driver.

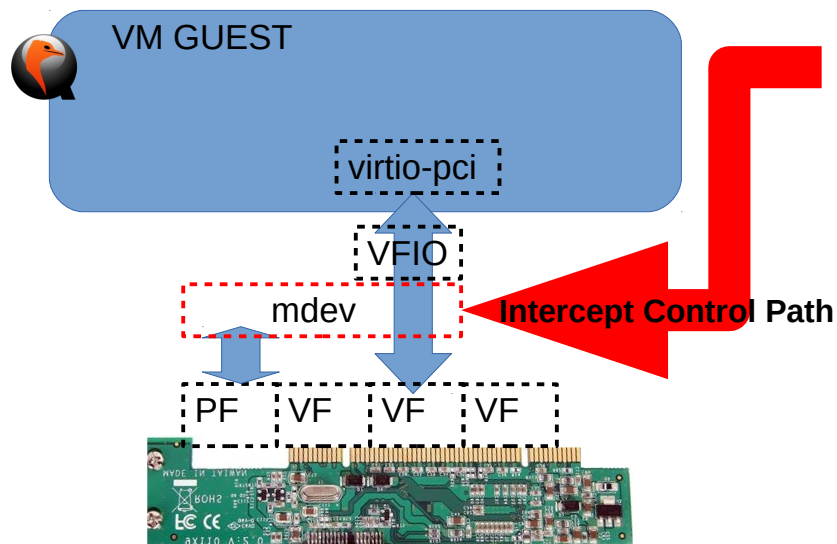
Well, you can use standard device passthrough.

So you can have multiple VFs each one implementing a virtio device interface. Bind a standard VFIO driver to this device and pass it through to a VM.

We are done: all software here is completely vendor agnostic and should just work.

Some people call this full offloading: both initialization and data path operation are offloaded - that is, implemented in hardware.

Data path offloading (VDPA)



Another option is to keep part of the functionality in software.

So we load a device specific driver, in this case the vhost mdev driver, conceptually inserting it between the VM guest and the device.

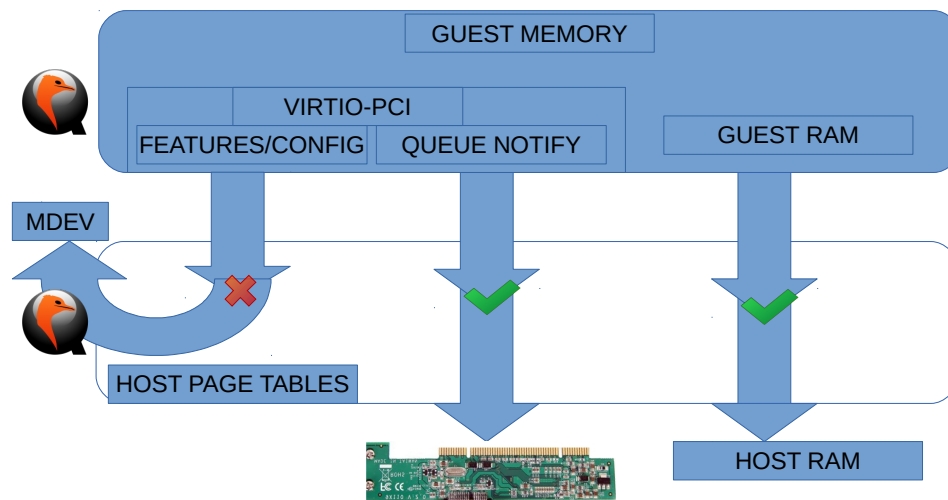
The job of the mdev driver is to implement the initialization parts of the virtio device.

Accesses to the configuration part of device memory are intercepted and forwarded to the mdev driver.

Mdev can be vendor specific. It can access both the PF and the VF to implement virtio configuration in software, and update the hardware as appropriate.

This is called data path offloading or virtio data path acceleration.

VDPA operation



20

We talked about configuration. Let's look at device operation now.

So the page table entry for the device configuration is marked non-present in host page tables. Guest accesses are trapped by the host and forwarded to the mdev driver on the host.

On the other hand, guest notification area is mapped to the device with present page table entries. Guest thus can send notifications directly to the device without involving the host. This is just like pass-through.

Finally guest memory needs to be mapped to host RAM and made available to device in the IOMMU, for DMA operation, again just like pass-through.

⇒ Configuration in config space

- Device BAR - easy to intercept
- DMA is harder!
 - E.g. control virtq
- discovery through device configuration is preferable
- Software/hardware switch?

I'd like to stress here that what makes it work well is the split between control path and data path:

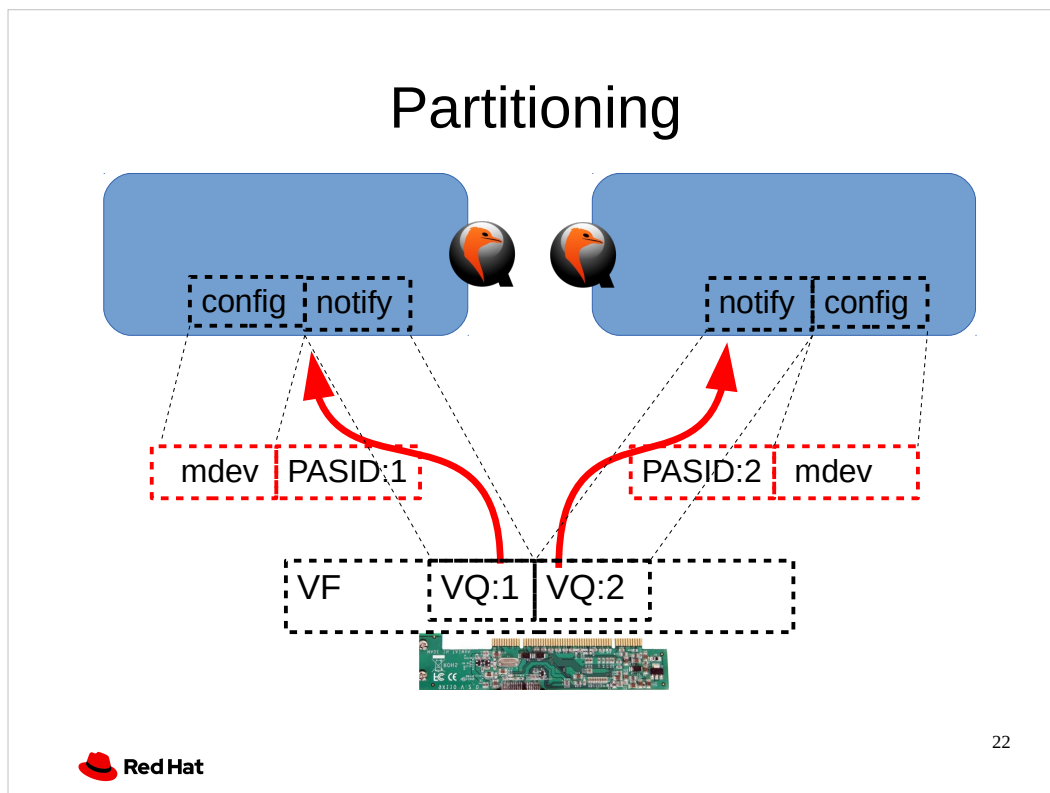
Control path and discovery is through configuration, data path is through virtqueues.

Unfortunately some devices support initiating special control operations through a virtqueue. This makes them harder to intercept.

We should strive to avoid this in future devices, and find some kind of work around for existing ones.

For example, the new virtio filesystem has its own version negotiation through a virtqueue. Only happens at boot – the solution can be to switch between a software implementation and a hardware one on the fly.

This is something we should keep in mind when designing new interfaces.



Let's talk about vDPA some more. It turns out that the mdev driver allows a bunch of tricks. First, it is possible to do partitioning, splitting up a single VF between multiple drivers.

Take a look at this slide. This shows two virtqueues in a single VF. Each has a different notification page, and each is mapped to a different VM.

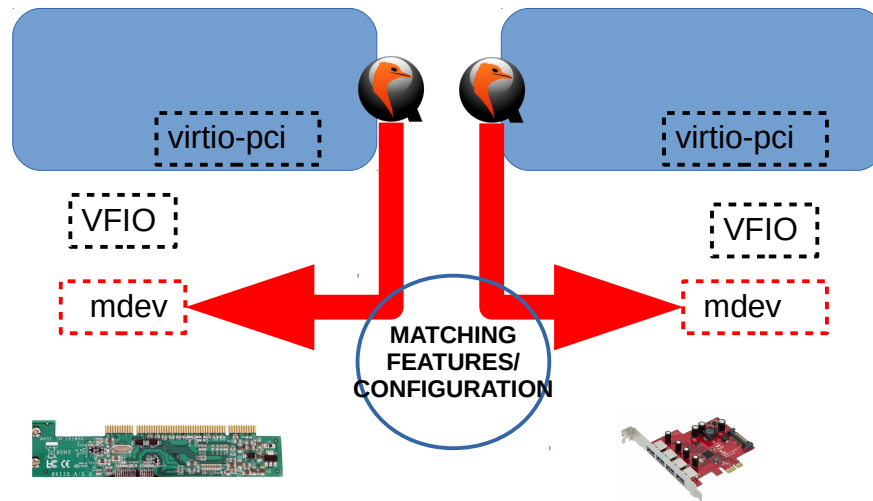
Configuration is handled by creating a separate mdev device for each VM.

To make this secure we need a way to protect Vqs from each other. For example, some systems support a Process Address Space Identifier (PASID) PCI Express extension.

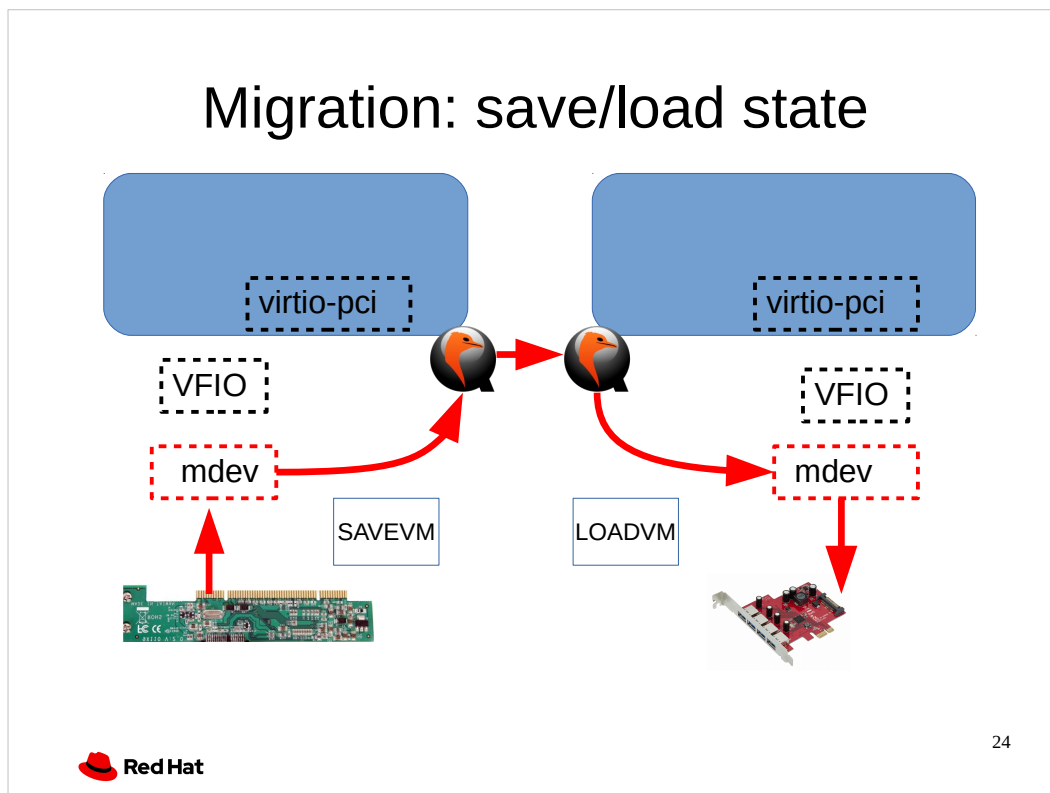
In this case mdev can allocate and attach a distinct PASID for each VQ, mapping different PASIDs to address space of different VMs.

PASID is relatively new so support is not yet widespread.

Migration: feature compatibility



Another use for mdev is to enable VM migration. In order to migrate a VM between two Virtio devices without shutting it down, device features and configuration need to look exactly the same. This is hard to ensure especially if the devices are from different vendors: for example, they can have slightly different feature sets depending on the vendor. Here, mdev can again play a role: we can program it with the expected features and configuration on both sides. As a result, identical virtio interfaces will be presented by both the source and the destination.



Another requirement for live migration is ability to save and restore the internal device state.

To this end, we can again use the mediated device.

The device will present a standard vhost based interface, allowing us to retrieve the internal state and to load it back into the device.

On source we can retrieve the state, save it and send it to the destination.

The destination will load the state to the device again through the mediated device.

This works for VM snapshots and userspace snapshots such as CRIU.

Migration is a complex topic, so I'm not completely done with it yet, but let me talk about something different first

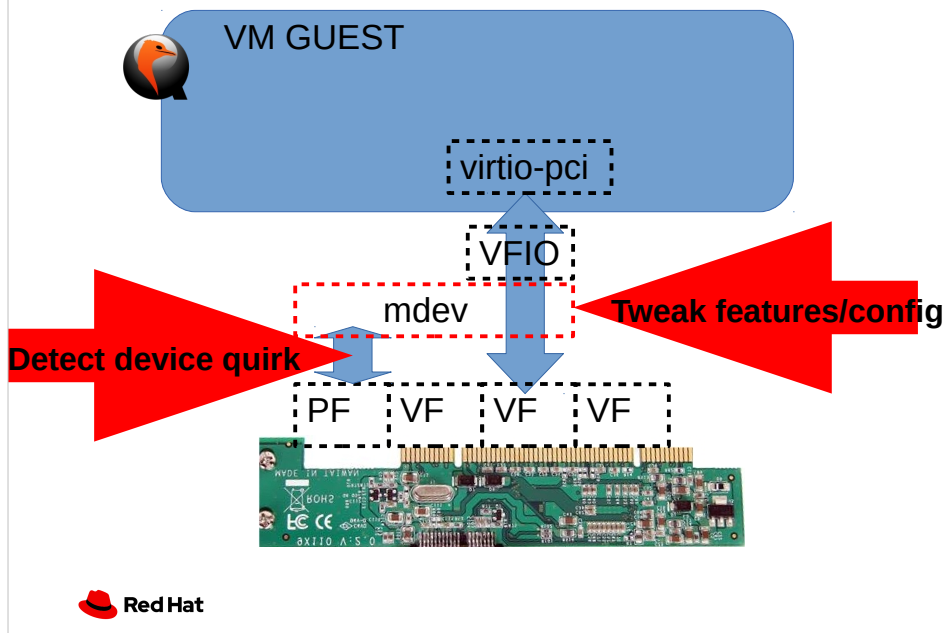
Device quirks

- Don't do it!
- Mask affected features
- Treat it as a feature
Document in spec
- Support at least 128 feature bits
- Blacklist device, use a vendor-specific driver



Device bugs. Politely known as quirks. How are we going to handle device quirks? Preferably devices will have a way to fix them, for example a device can have a small amount of programmable memory where you can mask specific problematic features. If the spec is ambiguous, then in the past we just documented the actual device behavior in the spec. Again, if features are programmable then we can allow drivers to detect the behavior. We are more than half way through the first 64 bit, so I recommend at least 128 feature bits. Worst case we can always black-list a device and then a vendor specific driver will load.

Host side quirks (mask/set features)



This shows how we can actually use the mdev infrastructure for quirks.

This is similar to what we had to do for migration: in this case we can have vendor specific code that can test both the PF and the VF as appropriate and detect the quirk.

It can then clear any feature bits that correspond to broken features.

It can actually also set feature bits if appropriate: for example, this would be a way to address a device which does not set the PLATFORM_ACCESS or PLATFORM_ORDER feature bits.

This actually happened with a real device.

Device quirks (cont)

- How to detect the device?
- Vendor/Device ID in use
- Subsystem Vendor ID?
- Reason not to support legacy
 - At least optionally
- Proposal: Vendor data capability
 - Includes Vendor ID



In previous slides, we assumed that we can detect the device vendor and make, making the quirk specific. The normal way to do this for PCI devices is with a vendor specific vendor/device ID. Unfortunately the virtio spec dictates a specific vendor and device ID. It is a good idea to have a vendor specific subsystem vendor ID so drivers can detect it, making the work around vendor specific.

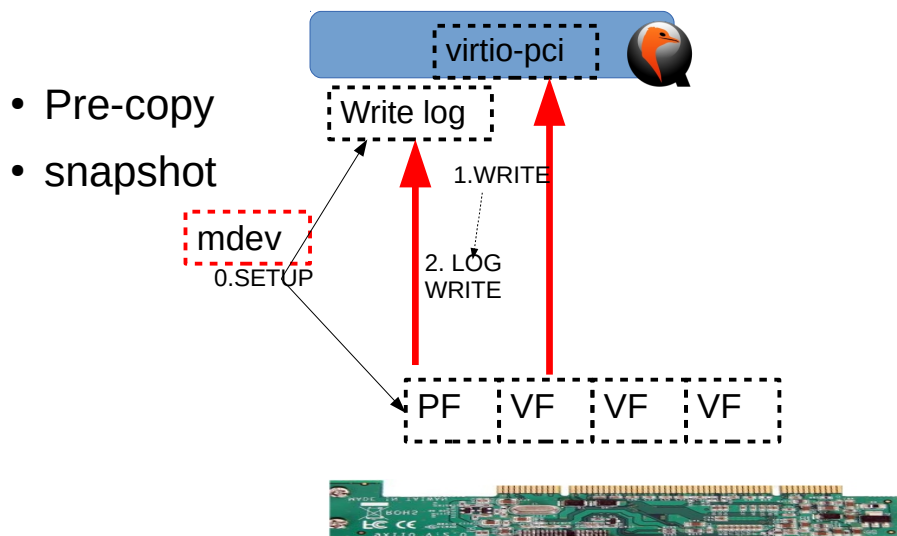
For modern devices, that is probably sufficient.

For legacy devices, the subsystem ID was set in the spec, so the quirk can't be specific.

That's another reason not to support legacy, or at least to have a way to reprogram the device and disable legacy.

There is a spec proposal for a vendor specific capability for devices which can't use a subsystem ID for legacy reasons.

Write logging



Once you have the mdev device, you keep finding more uses for it.

Here's another one: for migration and snapshotting, it is handy to know which pages the device modifies.

The mdev driver can setup a write log buffer and associate it with the PF. Now, whenever the VF writes into memory, the PF can log it in the buffer. The affected pages can then be considered dirty, or changed, for example they can be resent for migration or saved to file for snapshotting.

This approach has been implemented by some vendors but has the problem of not supporting post-copy migration. I will later describe some ideas for supporting post-copy if I have the time.

But first, let me talk a bit about the administrative aspects.

Virtio 1.2 plans



- Freeze spec by end of November 2019
- Public review draft by end of year
- Public review to run until early next year



Here's a tentative time-line.

Assuming all goes well we should be able to freeze the spec and publish a public review draft by end of November 2019.

A public review process will then start. It takes a month, so it should finish beginning of next year.

What happens after that – a vote on making the draft the next release or another draft and another review round would depend on the kind of comments we'll get.

Help implementing the spec

- Don't:
 - send a private email to Michael 
- Subscribe to virtio-dev@lists.oasis.org
- Send questions to virtio-dev@lists.oasis.org 
 - Copy relevant people

What to do if you need to figure out how to interpret the spec.

What not to do – private email to michael

What to do – email the list

And feel free to copy me

Device needs a spec change !

- Don't:
 - send a private email to Michael ✗
 - Wait until it's ready, then publish ✗
- Do:
 - Subscribe:
virtio-comment-subscribe@lists.oasis-open.org ✓
 - Send **General description** to:
virtio-comment@lists.oasis-open.org ✓
 - Copy relevant people ✓

What if the spec needs to change?

What not to do is send a private email.

Another bad idea is to lock yourself up in a garage for a year and work hard on a spec proposal and only post it when it's completely ready.

It is a good idea to publish as early as possible.

Posting just a general idea on the mailing list is a good start.

Specification process

do I have to write a spec?

- Absolutely the right thing to do
- Does not have to be step 0!
- Virtio priorities:
 - Code compatibility
 - IPR compatibility
 - Interface compatibility



Okay – you decided to base your new project on virtio. Congratulations! Is it true that any change to a virtio device requires one to write a full specification?

This isn't completely wrong. It is definitely encouraged that people document any changes they are making to the interfaces.

However this isn't a top priority - at least not for all virtio developers.

What is our priority?

- code compatibility - avoid conflicting with existing, past or future devices and drivers
- IPR compatibility - allow others to implement compatible virtio devices and drivers
- interface compatibility - allow different implementations to inter-operate

Code compatibility: avoid conflicting with others

- New device: reserve an ID. Spec patch:

```
diff --git a/content.tex b/content.tex
@@ -3022,3 +3022,5 @@ Device ID & Virtio Device  \\
\hline
+23      & misc device \\
+\hline
\end{tabular}
```

- Existing device: reserve a feature bit. E.g. :

```
@@ -4800,5 +4802,6 @@ guest memory statistics
\item[VIRTIO_BALLOON_F_DEFLATE_ON_OOM (2) ] Deflate balloon on
guest out of memory condition.
+\item[VIRTIO_BALLOON_F_XXXX (3) ] Reserved for
+ feature XXXX.
\end{description}
```



33

How does one achieve code compatibility?

For new devices – you just need to reserve a device id.


On top is a minimal patch to reserve a device id.

This will guarantee that no other future virtio driver will be bound to your device.

If you are changing an existing device, then the minimal patch reserves a feature bit. Here's an example for the balloon device.

I think everyone will agree this isn't a lot of text to write. What is the process to get this into the spec?

How to get it in the spec?

- git clone <https://github.com/oasis-tcs/virtio-spec>
Edit :)
- sh makeall.sh (needs xelatex, e.g. from texlive)
- virtio-comment-subscribe@lists.oasis-open.org
- Patch: virtio-comment@lists.oasis-open.org
- If no comments – email, ask for a vote ballot
- Total time: up to 2 weeks 



34

Clone the spec git repo, and edit the spec. It is written in xelatex which is a latex dialect, so you need to compile it to make sure it looks right.

Right now it builds on linux. Windows and MAC should be possible using the texlive distribution.

Subscribe to the comment list by sending a blank email message to:

virtio-comment-subscribe@lists.oasis-open.org. You need to confirm your subscription request.

Send your patch to:

virtio-comment@lists.oasis-open.org. If everyone's happy with it, send email to the mailing list, and copy the chair (Michael Tsirkin) asking for a ballot to approve the change. Ballot runs for 1 week.

Total time to merge: up to 2 weeks.

IPR compatibility: allow others to implement compatible devices

- Open-source an implementation
- Subscribe to virtio-dev@lists.oasis.org
- Agree to IPR rules (non-assertion mode)
- Send a copy of the patches (e.g. qemu, linux, dpdk) to virtio-dev@lists.oasis.org



35

We generally ask that an open-source implementation draft is available, and that you agree to the non-assertion mode of the virtio spec.

To this end:

Subscribe to virtio-dev@lists.oasis-open.org

You will be asked to agree to the IPR rules.

Send a copy of the patches (e.g. qemu, linux, dpdk) to virtio-dev@lists.oasis-open.org

Interface compatibility



- Document assumptions for inter-operability
- Submit as comments
- Virtio membership is not required
- Membership is open - members vote on ballots
- Hints:
 - Document device and driver separately
 - Use MUST/SHOULD/MAY keywords
 - Ask for help!
- Virtio crypto, input, gpu added recently



36

Interface compatibility is hard. To inter-operate with others one really needs documentation. Ideally one would add a full description of the new feature to the specification. Virtio TC membership is not required for this!

Things to consider when writing the documentation: separate device and driver requirements in separate Normative sections. These and only these are to use MUST/SHOULD/MAY keywords. Ask for help if you are not sure how to start!

Example: virtio crypto and virtio input devices - specification is still WIP, implementations are already in use!

If you maintain a part of specification, please consider joining the TC so you can vote on spec changes. Membership is open for all - your employer might already be an OASIS member - there's no limit to the number of employees that may participate.

Work-in-progress



- Manageability
- Performance

It would be a boring talk if I did not describe some challenges that we are trying to address in virtio. These include performance optimizations and manageability improvements.

Page faults



- No faults in classical pass-through
- Required for:
 - Overcommit (KSM/swap)
 - Pre-/Post Copy Live migration / Snapshots
- SVA in Linux for PCI PRI
- No userfaultfd support (post-copy)
- Not universally supported by IOMMUs

For manageability, one interesting idea is to support page faults for hardware viirtio.

Regular PCI passthrough tends to pin all of guest memory, and there are never any pagefaults.

But pagefaults are very handy to support.

First you can do overcommit, moving memory to a disk with swap, or using copy on write with KSM to de-duplicate memory.

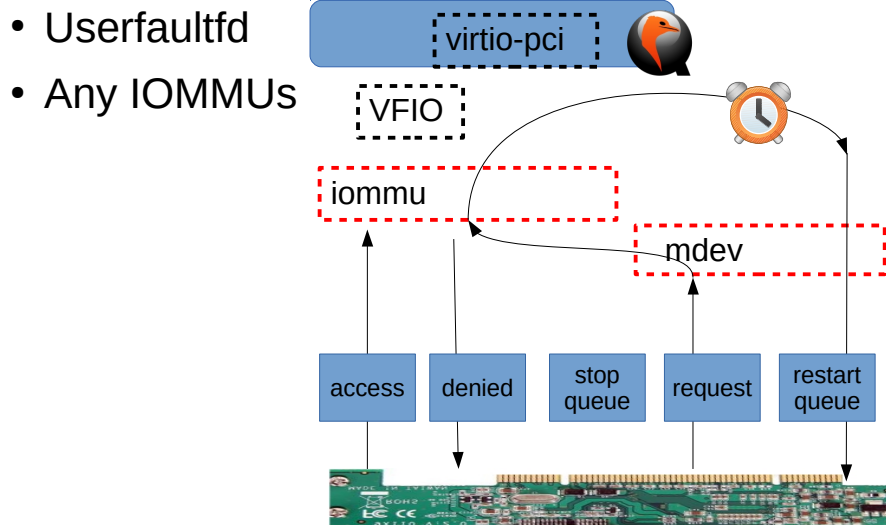
You can use pagefault for live-migration or snapshots: precopy needs write faults for write tracking, while postcopy needs both read and write faults to work.

There's a PCI PRI support in linux, but that attempts to handle all faults in the kernel, which means no userfaultfd so for example no post-copy.

It is also not universally supported by IOMMUs yet.

So what can be done?

Page faults by mdev



It might be possible to support faults by the mdev driver. The idea is to extend the linux IOMMU API to allow protecting individual pages from access by device.

A device access will then trigger an error.

Normally device would enter an error state and require a reset.

A smart device could instead stop the queue and send an event to the mdev driver. Mdev could then try to fault in the page. On success, it could restart the queue.

Hardware vs software



- Let's assume a pass-through device implementing virtio. Shouldn't this just work?
- Maybe – but not optimally!
- Hypervisor: processes descriptors one by one
- Hardware: can process many in parallel
- Needs to be told how many are available
- Include number of available entries in a kick

Software virtio devices pretend to be pci devices, so it seems natural to assume that things just work if you implement a virtio device in hardware.

This is mostly true, but sometimes we found out that hardware devices require different optimizations from software.

For example, while software has to process descriptors one by one, a hardware device can actually access and process multiple descriptors in parallel.

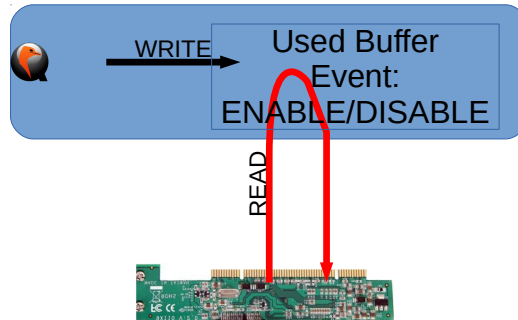
That would be nice except device generally does not know ahead of the time how many descriptors are available.

To enable this optimization, driver kick must give the device a hint how many descriptors are available for processing. Device then uses this hint to fetch and start processing multiple descriptors in one go.

Interrupt suppression



- Optimized for software



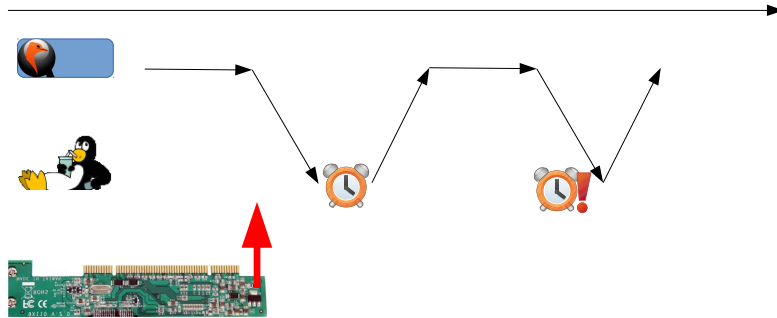
- Move to device memory?

Interrupt suppression is optimized for software in that software tweaks a suppression structure in RAM. This means that a hardware device needs to go to a lot of effort to figure out whether to send an interrupt. It might be beneficial to move this structure over to the device.

Interrupt coalescing



- Expensive in software



- Easy in hardware

Interrupt coalescing never made any sense for software virtio: you get an interrupt when a packet arrives from the NIC, you exit to the host. If you now start a timer in order to deliver the actual interrupt to the guest you are going to get two exits for a single interrupt.

With a hardware virtio device there's no first interrupt: hardware does the coalescing and the timer, so it suddenly makes sense to add it to the spec.

Summary

- Network effects and a set of unique properties make Virtio a compelling option for new devices
 - Large Software and Hardware ecosystem
- Join the fun
 - Easy to extend
 - A lot going on
 - Performance + new features



To summarize in virtio we have a lot of existing infrastructure, a decent spec to work from and a reasonable process to extend both. Multiple Vendors are looking to enhance the way we handle hardware Virtio devices. So development proceeds at a high speed. Contribution is open to everyone – so join us!

Questions?

