Notes about Virtualization

QEMU, KVM, IOMMU, and more

Yizhou Shan
ys@ucsd.edu

Created: Jan 25, 2020
Last Updated: Feb 4, 2020

Table of Content

Introduction

Scratchy and raw notes about QEMU and KVM. It has some links to various posts. It has some sentences trying to explain how QEMU interact with KVM, and some code snippets from both QEMU and Linux kernel KVM module.

Honestly, I didn't fully understand the whole QEMU/KVM thing the whole time, until I decide to take a deep tour recently. End of the day, I'm satisfied, mostly. I now know how QEMU invokes KVM, how KVM launch guest, how KVM handles vmexit, and vmexit translate to userspace-visible `KVM_EXIT_REASON` and so on, all in concrete code pieces. More importantly, I dive deep into how device is emulated. More specific, IO and MMIO emulation, it's super interesting but really fundamental, I strongly recommend the QEMU blog explaining the device emulation.

Device emulation is costly, that's why Amazon use Nitro to offload now (not too much tech details available online)! VM can use IOMMU to have exclusive access to a device, the IOMMU ensures memory safety, and optimizes the interrupt-to-cpu delivery. SR-IOV improves scalability, it makes a physical device appear as multiple virtual devices. Combing SR-IOV and IOMMU, each running VM can have exclusive access to a virtual function, no VMM involved, even if there is just one physical PCIe device. Note that, a) IOMMU can be used without SR-IOV, that means a physical device can be used by one VM only, b) in theory, SR-IOV-capable device can be used without IOMMU, as long as the guest VM can see host physical address. However, the usual practice is always use SR-IOV with IOMMU, thus it can translate guest physical address to host physical address.

Open Source Projects

(If you want to develop a hypervisor, or a VMM, most likely you can find codes and references in the following projects.)

- Wenzel/awesome-virtualization: Collection of resources about Virtualization
- QEMU
- https://github.com/cloud-hypervisor/cloud-hypervisor
- Xen-PV, Xen-HVM
- Rust-VMM, Firecracker, Cloud-Hypervisor
- https://projectacrn.org/ Good documentation!
- Documentation
    - Red Hat Virtualization. Good ones!
    - Intel Virtualization Technology (Intel VT) Hub
    - Intel Vt-d Specification, 2019
    - PCI-SIG SR-IOV Primer from Intel, 2011 (Auto download pdf)
    - Intel White Paper: Enabling Intel® Virtualization Technology Features and Benefits

QEMU Source Code Study:

References about the source code:

- Qemu Detailed Study. Some pointers to the main loop flow, TCG flow.
- QEMU Code Overview Architecture & internals tour
- QEMU Internals: Overall architecture and threading model.
    - There are two mechanisms for executing guest code: **Tiny Code Generator (TCG)** and **KVM**.
    - The “main” queue thread runs a main loop
- Devices
    - QEMU’s new device model qdev
    - Virtio: An I/O virtualization framework for Linux
    - Also QEMU device will “fire” interrupts, via Linux signals to main loop.

Details:

- `ui/` has VNC code and more
- `vl.c` is the main entry file
- `hw/` has the devices
    - hw/net/virtio-net.c: the virtio network device**.** Both data and control path.
    - hw/net/vhost_net.c: the vhost network device. The one deal with the host vhost device driver, this file should only have the control plane stuff to handle PCIe control access.
- KVM invocations at `**accel/kvm/*.c**`
- BIOS binary at `**pc-bios/**`. Almost all files are in `.bin` and `.rom` format.
- Devices
    - `hw/char/serial-isa.c`: serial ttyS0/ttyS1, 0x3f8 IO port.
    - `hw/net/e1000.c`: the classical NIC card
    - And some legacy i8294, i8295 devices
    - And many more
    - Vhost: QEMU Internals: vhost architecture
- Memory Model
    - Guest Physical Memory Layout Management: `memory.c`
    - It looks like each device claim its address via `memory_region_init_io/rw()` API.
        - For example, `hw/char/serial-isa.c` used the “memory_region_init_io” to claim an IO port. And it registers a set of callbacks as well.
    - Memory model explained: https://qemu.weilnetz.de/doc/devel/memory.html

My Summary:

1. QEMU has many pieces, including dynamic binary translation (or tiny code generation, or TCG), KVM acceleration, device emulation, and more helpers. **From my understanding, TCG and KVM are two exclusive modes to run guest code, it’s either TCG or KVM**.

2. QEMU runtime is mostly event-driven. QEMU uses one “main” thread to run a repeated main loop. Inside the loop, QEMU will run the guest code, either via TCG or via KVM. A single “main” thread can run one vCPU or multiple ones, it depends on if CONFIG_IOTHREAD is enabled. Normal practice is one “main” thread for one vCPU.
3. Conceptually, the “main” thread loop has two user-mode contexts. One is the *QEMU contex*t, another is the *guest context*, and they are exclusive. The goal is to run in guest mode as much as possible, thus dedicating the whole pCPU to vCPU.
    a. **QEMU->Guest context transition**: When the “main” thread starts (`vl.c`), we run in QEMU context, where we first allocate/prepare misc stuff. Next call KVM to allocate a VM. Next, the “main” thread ask KVM to start running guest via an ioctl (i.e., `ioctl(vcpufd, KVM_RUN, NULL)`). *During this particular ioctl call, we transition from user-mode QEMU context to kernel mode, then the kernel mode will transition to user-mode guest context*.
    b. **Guest->QEMU context transition**: Whenever guest context incurs a vmexit (e.g., MMIO read/write), the CPU will exit to kernel mode KVM handlers first. Then, the KVM module will determine if this particular vmexit should be handled by userspace (note that kernel KVM module will handle some particular vmexit itself rather than exposing to userspace). If so, the `ioctl()` syscall that caused the QEMU->Guest transition will return to userspace, and then we will be back at QEMU context!. And repeat.
    c. The whole thing is demonstrated use real code.
4. Other than those “main” threads, QEMU has other *worker threads*. The motivation is simple, many device operations are asynchronous to guest, i.e., interrupt-based. The “main” threads will offload some tasks to those worker threads, mostly some asynchronous tasks. For instance, the “main” thread may offload VNC computation, disk operation to worker threads. Upon completion, the worker threads can either send signals or use file descriptors to notify the main thread.
5. The main loop is waiting for events, some from guest, some from worker threads, some from timer expires. Overall, QEMU is like the Linux kernel, it needs opportunities to gain control thus run code. And “main” thread cannot always run in guest mode, what if guest is running spinning code, right? So QEMU has either timer or signals to let QEMU context has a chance to run.
    a. Actually, I’m not sure how this happens. The user mode QEMU cannot just regain control. It has to be the kernel side KVM to help, right?
6. To me, with the help of KVM, the main thing left for QEMU is to emulate all the devices. Like this blog said, QEMU will catch all the IO and MMIO accesses and emulate the effect as they will do in the bare-metal machine.
    a. “With QEMU, one thing to remember is that we are trying to emulate what an Operating System (OS) would see on bare-metal hardware”
    b. “ And at the end of the day, all virtualization really means is running a particular set of assembly instructions (the guest OS) to manipulate locations within a giant memory map for causing a particular set of side effects, where QEMU is just a user-space application providing a memory map and mimicking the same side effects you would get when executing those guest instructions on the appropriate bare metal hardware.”

Device Emulation:

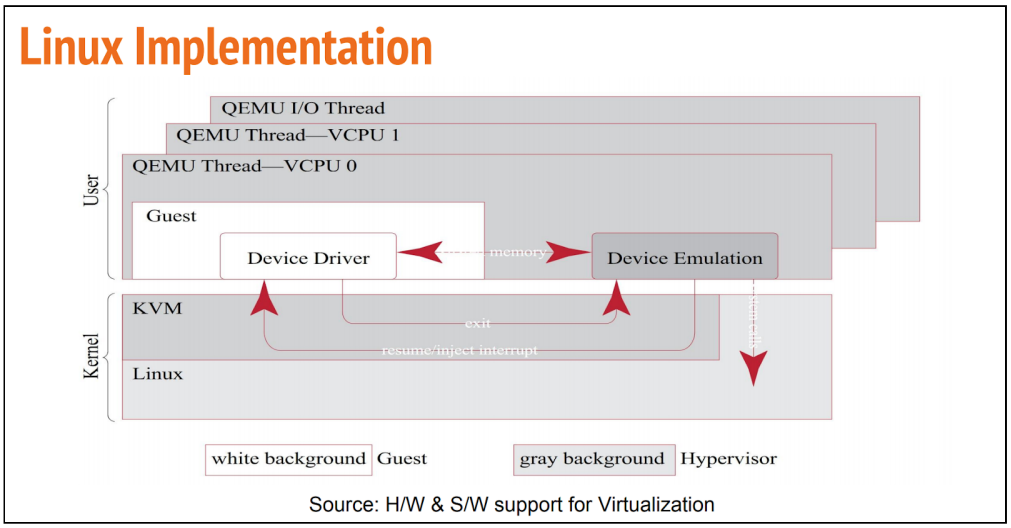
Models:

- Great Reference:
  - PCI-SIG SR-IOV Primer from Intel, 2011. It helps to understand the difference between SR-IOV and Intel VT-d (IOMMU).
  - Intel Vt-d Specification, 2019
- ***Software-Based Emulation***.
  1. **Traditional Device Emulation**. In this case, the guest device drivers are ***not*** aware of the virtualization environment. During runtime, the VMM (QEMU/KVM) will trap all the IO and MMIO access and emulate the device behavior. The VMM emulates the I/O device to ensure compatibility and then processes I/O operations before passing them on to the physical device (which may be different) The downside is obvious, there will be A LOT vmexit!.
  2. **Paravirtualized Device Emulation**, or **virtio**. In this case, the guest device drivers are aware of the virtualization environment. This approach uses a front-end driver in the guest that works in concert with a back-end driver in the VMM. These drivers are optimized for sharing and have the benefit of not needing to emulate an entire device. The back-end driver communicates with the physical device A lot of vmexits can be coalesced thus perf can be improved.
- ***Hardware-assisted Emulation***.
  1. **Direct Assignment**. Let a VM directly talk to device. Thus the guest device drivers can directly access the device configuration space to, e.g., launch a DMA operation. The device can DMA to physical memory in a safe manner, via IOMMU. This is enabled by Intel vt-d. **Drawback**: One concern with direct assignment is that it has limited scalability; a physical device can only be assigned to one VM.
  2. **SR-IOV and Direct Assignment**. Incremental to above one. With SR-IOV, each physical device can appear as multiple virtual ones. Each virtual one can be directly assigned to one VM, and this direct assignment is using the vt-d/IOMMU feature.
  3. AWS Nitro. Details?
- **Questions**
  - If I were to implement a “qemu” myself, except those ioctls, do I need to any net/blk etc device emulation parts? Will those virtio in guest/host linux take care of everything?
    1. Yes, you have to! Read the virto section. Reasons: 1) there is no complete virtio backend drivers in the kernel space. Even the vhost only has data path, control path is still handled by userspace
    2. That being said, if you want to implement a new VMM to run Linux kernel, you will have to implement either raw emulation or virtio backends.

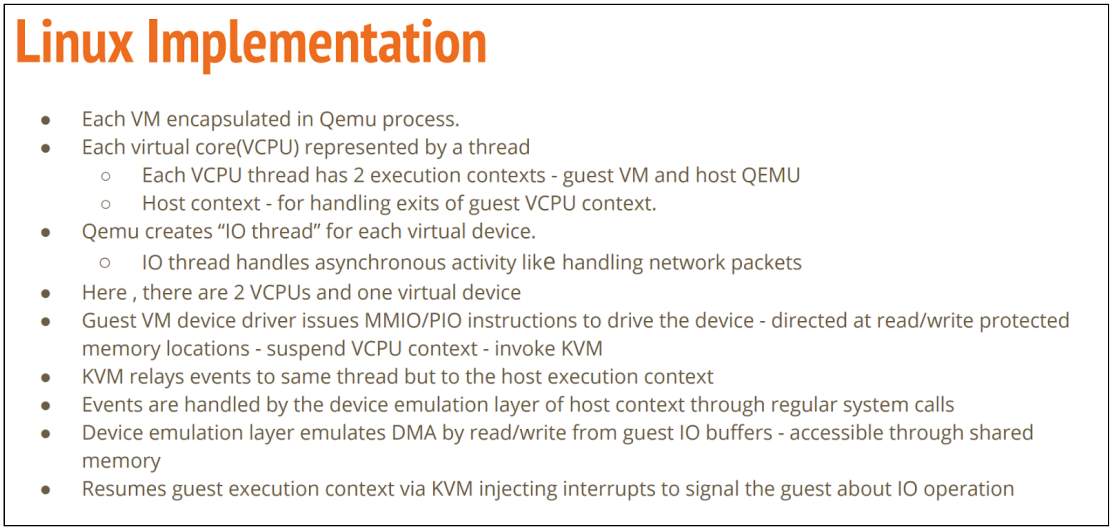
    3. Also I think that's why every new VMM needs to deal with virtio and vhost. If you check QEMU, cloud-hypervisor, ACRN etc, they all handle these.
- SR-IOV and IOMMU: a) Find out more about the intel-iommu part, what has been setup etc. b) Also How qemu/kvm setup the SR-IOV device? At a high-level, it just bypass host linux. The guest can talk with device directly, which means it can access that portion of physical memory directly. So the trick maybe as simple as "having a valid EPT pgtables rather than the ones would trigger MMIO faults"?

QEMU Device Emulation Details

- Understanding QEMU devices.
    - The ***best*** blog explaining how to emulate IO devices in general.
    - Think about how OS sees and uses devices: they either use x86 IO ports, or use memory-mapped memory, or MMIO. So, at a high level, if we can catch all the accesses (read/write) to MMIO and IO ports, we can emulate the devices. And this is exactly what QEMU and KVM did!
    - First off, QEMU will present the address space as the bare-metal machine did. It will associate QEMU device emulation code to a portion of address space, e.g., a PCIe device address range. This is done via the `memory_region_init_io/rw()` API. All devices will claim their portion before VM runs.
    - More importantly, KVM provides API to let user to define the guest machine's physical memory layout, i.e., `ioctl(KVM_SET_USER_MEMORY_REGION)`. You will see later that KVM is able to catch the MMIO fault in a special way, and then pass it up to QEMU.
    - This naive solution is too slow, because every MMIO access results in a world switch: guest -> host kernel -> host QEMU user context. That's we have virtio, paravirtualized IO devices! See the below slides
- https://compas.cs.stonybrook.edu/~nhonarmand/courses/sp17/cse506/slides/io_virtualization.pdf
    - This is the ***best*** slides explaining ****how QEMU/KVM handle MMIO/PIO**** and emulate devices! Basically, guest driver writes to MMIO addresses, which causes pgfault and vm exit. KVM forward this to QEMU.
    - End of the day, I now understand how QEMU manages memory model, how it associates physical memory with devices, how QEMU register guest physical memory layout in KVM, how KVM exports the MMIO/IO exits to userspace QEMU etc. I now know how it works top down, just like the second slide.

- 

  Source: H/W & S/W support for Virtualization

- 

- https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2018/09/03/kvm-mmio
  - Explains how MMIO emulation is implemented in KVM
  - For a summary, the following shows the process of MMIO implementation:
    1. QEMU declares a memory region
    2. Guest's first access to MMIO addr will cause an EPT violation vmexit
    3. KVM constructs EPT pgtable and marks the PTE with special mark
    4. Later the guest access these MMIO, it will be processed by EPT misconfig VM-exit handler
- Virtio and vhost:
  - Refer to the following sections.
  - QEMU provide both. They are both the backend for the guest virtio drivers.

QEMU/KVM Code Flow:

- First, QEMU's device models uses `*memory_region_init_io/rw*()` to declare the IO ports and physical memory addresses each device operating upon. The device will provide a set of callbacks when the read/write access happen.
- Then, inside `accel/kvm/kvm-all.c`, during preparation, QEMU will use the valid registered `memory_region_init_io()` devices as the input, and call `KVM ioctl *KVM_SET_USER_MEMORY_REGION*` to define the guest physical memory layout! This part is very interesting!
- Finally, I think the KVM will sets the EPT pgtable accordingly, as the normal memory probably could just go through, but the MMIO-type of memory has invalid EPT pgtable entries which will cause VM exit (EXIT_EPT_VIOLATION and so on,)
- For instance, the part where QEMU handles IO and MMIO:

```
qemu: accel/kvm/kvm-all.c

        switch (run->exit_reason) {
        case KVM_EXIT_IO:
            DPRINTF("handle_io\n");
            /* Called outside BQL */
            kvm_handle_io(run->io.port, attrs,
                          (uint8_t *)run + run->io.data_offset,
                          run->io.direction,
                          run->io.size,
                          run->io.count);
            ret = 0;
            break;
        case KVM_EXIT_MMIO:
            DPRINTF("handle_mmio\n");
            /* Called outside BQL */
            address_space_rw(&address_space_memory,
                             run->mmio.phys_addr, attrs,
                             run->mmio.data,
                             run->mmio.len,
                             run->mmio.is_write);
            ret = 0;
            break;
```

- The part where QEMU registers MMIO via KVM:

```
kvm_set_user_memory_region
```

- KVM APU, about the **KVM_EXIT_MMIO**:

```
            /* KVM_EXIT_MMIO */
            struct {
                    __u64 phys_addr;
                    __u8  data[8];
                    __u32 len;
```

```
                __u8  is_write;
        } mmio;

If exit_reason is KVM_EXIT_MMIO, then the vcpu has
executed a memory-mapped I/O instruction which could not be satisfied
by kvm.  The 'data' member contains the written data if 'is_write' is
true, and should be filled by application code otherwise.
```

- KVM Internal flow. This is the transition from QEMU to guest and guest back to QEMU. The functions are from Linux kernel, spread over `virt/kvm`, `arch/x86/kvm/vmx/vmx.c`, `arch/x86/kvm/*`.

```
(Linux)
x86_emulate_instruction()
  -> x86_emulate_insn
       -> exec() -> IO/MMIO -> fill in the  KVM_EXIT_IO info etc

(Linux)
kvm_mmu_page_fault (this is called from the VM EXIT handler array)
  -> x86_emulate_instruction

(QEMU -> Linux -> QEMU)
QEMU kvm_vcpu_ioctl(cpu, KVM_RUN, 0)
 -> Linux kvm_arch_vcpu_ioctl_run
     -> vcpu_run
          -> vcpu_enter_guest
                -> kvm_x86_ops->run(vcpu); (run the guest!)
                -> handle_exit_irqoff()
                -> handle_exit() which is vmx_handle_exit
                    -> handle all the vmexit, fill in the KVM_EXIT reasons.
                        (kvm_vmx_exit_handlers[exit_reason](vcpu))
                        -> handle_ept_misconfig (just one of many handlers!)
                            -> kvm_mmu_page_fault
                                -> x86_emulate_instruction
                                    -> x86_emulate_insn
                                         -> exec() -> IO/MMIO -> fill KVM_EXIT_IO


(if vcpu_enter_guest returns 1, the whole thing will break the loop and return back to userspace, where the above qemu code can inspect the KVM_EXIT reasons.)
```

virtio/vhost (Paravirtualization for devices drivers):

Notes and references:

- The original virtio protocol is not hard to understand: guest traps to KVM once it writes to some MMIO location, then KVM returns the ioctl call, returns to QEMU context, the QEMU emulates the device behavior.
- **vhost** is a specific kind of virtio where the data plane is put into host kernel space to reduce the context switch while processing the IO request. (Will this always be on? How to use it?)

    - According to QEMU Internals: vhost architecture: the linux vhost code is not self-contained. The kernel vhost only handles data paths. The control pach such as PCI negotiation is still done at user QEMU context!
- And note that the backend drivers can be implemented in either kernel-space or user-space, it's just an implementation choice, as long as the backend driver follows the protocol. However, note the difference: kernel-based backend driver incurs less user-kernel switches:
    -

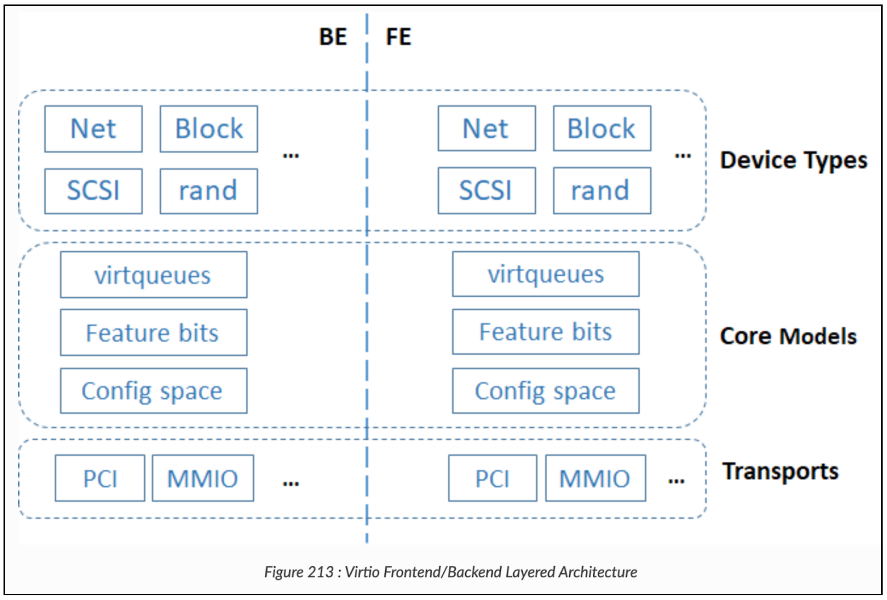


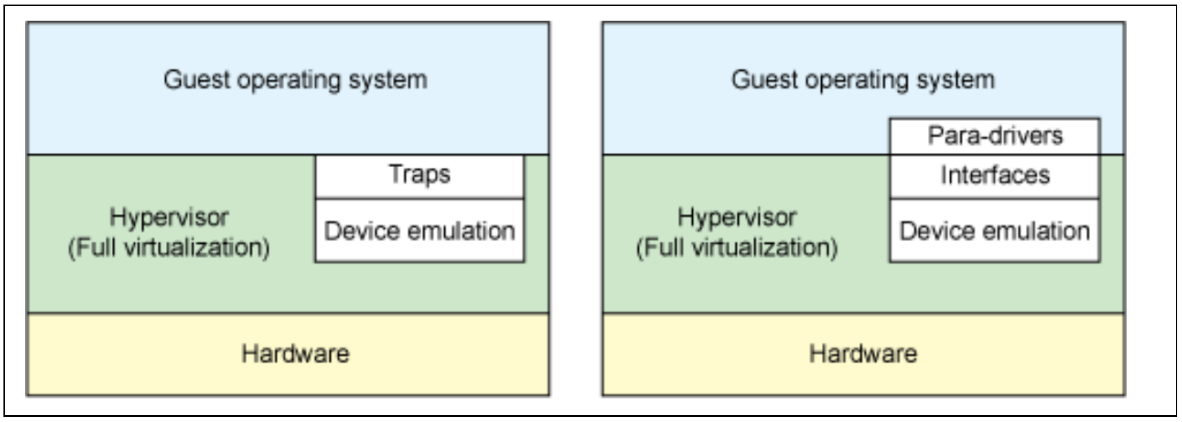
Figure 213 : Virtio Frontend/Backend Layered Architecture

- **Project ACRN: Virtio devices high-level design**
    - **One of the BEST BLOGs explaining virtio.**
- **QEMU Internals: vhost architecture**
    - **One of the BEST BLOGs explaining virtio.**
    - "Vhost does not emulate a complete virtio PCI adapter. Instead it restricts itself to virtqueue operations only. QEMU is still used to perform virtio feature negotiation and live migration, for example. This means a vhost driver is not a self-contained virtio device implementation, it depends on userspace to handle the control plane while the data plane is done in-kernel"
- virtio: Towards a De-Facto Standard For Virtual I/O Devices
    - The paper by Rusty Russell.
- Virtual I/O Device (VIRTIO) Version 1.1, 2018
    - Specification..
- Red Hat Blog: Deep dive into Virtio-networking and vhost-net
    - "The virtio specification is based on two elements: **devices** and **drivers**. In a typical implementation, *the hypervisor exposes the virtio devices to the guest* through a number of transport methods. By design they look like physical devices to the guest within the virtual machine."
    - "When the guest boots and uses the PCI/PCIe auto discovering mechanism, the virtio devices identify themselves with the PCI vendor ID and their PCI Device ID. The guest's kernel uses these identifiers to know which driver must

handle the device. In particular, the linux kernel already includes virtio drivers."

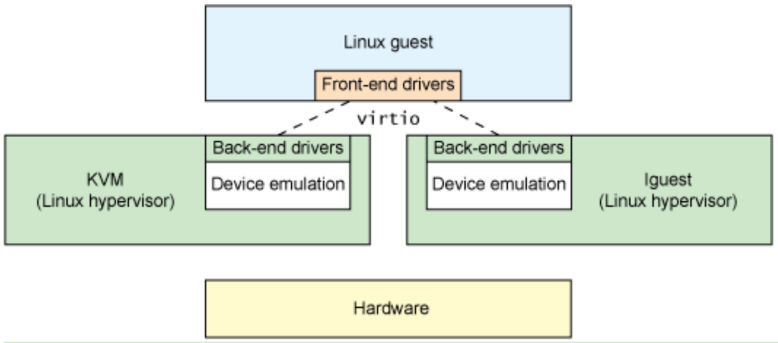
- "In the PCI case, the guest sends the available buffer *notification by writing to a specific memory address*, and the device (in this case, QEMU) uses a vCPU interrupt to send the used buffer notification." (Yizhou: *After writing the data into the queue, the guest driver will notify the host. This is done via a write to a specific memory address. I think this address will be an MMIO address, which causes vmexit, and then KVM will pass it to QEMU!*)
- Original virto flow:

- Virtio: An I/O virtualization framework for Linux



-

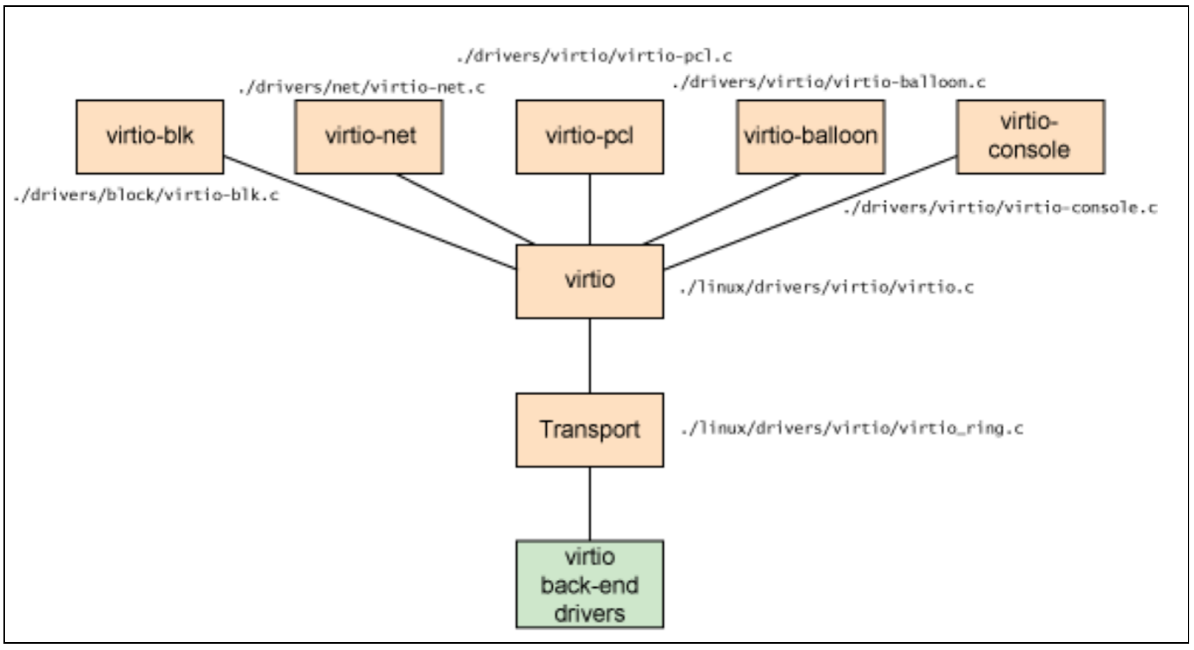
Figure 2. Driver abstractions with virtio



Note that in reality (though not required), the device emulation occurs in user space using QEMU, so the back-end drivers communicate into the user space of the hypervisor to facilitate I/O through QEMU. QEMU

-

-



I think when you start QEMU, you can choose if you want to use `virio` or `vhost` as the network/block/etc devices. QEMU will use `vhost` accordingly, i.e., use the host's `/dev/vhost-xxx` interface.

Source code:

- Linux**:**
  - drivers/virtio/*, drivers/net/virtio-net.c ..
  - vhost char devices: drivers/vhost/*.c
- QEMU:
  - hw/net/vhost_net.c
  - hw/net/virtio-net.c (you can see the difference between virtio and vhost, the latter's data path is handed by kernel vhost driver.)

IOMMU:

IOMMU can be used for virtualization or non-virtualization cases. If a device is directly assigned to a guest, IOMMU must be used, because we need to prevent the guest drivers from corrupting arbitrary hypervisor memory.

The BIOS presents IOMMU related information via the ACPI DMAR tables. The Linux IOMMU driver will parse the table and build necessary stuff. The whole linux intel-iommu.c follows the Intel VT-d specification, i.e., setup Root Table, Context Entry, and talks with IOMMU via MMIO register access. The IOMMU is involved in every DMA related operations, because it needs to prepare the page table entries.

IOMMU can also be emulated. QEMU can emulate an IOMMU for the guest, so the guest linux can also run its intel-iommu code. The emulation is done similar to other MMIO emulation techniques.

dma_map

Drivers will call **dma_map** and **dma_unmap** before and after each DMA operation
Remember dma_map? I have implemented in LegoOS, mainly for IB's needs. At the time of implementation, I've only implemented a **pci-nommu** version, that means the dma_map is barely just doing a "kernel virtual address to physical address" translation, that's all, no additional setup. The code is here:
https://github.com/WukLab/LegoOS/blob/master/arch/x86/kernel/pci-nommu.c#L50

Linux has the real IOMMU-based dma_map and dma_unmap. Ultimately, it uses the **dma_map_ops**, from linux/intel-iommu.c at master · torvalds/linux. I took a brief read of the code, it's my understanding that the code is following the Intel VT-d specification. More specific, this intel-iommu.c source file will:

1. Allocate the Root Table and write into the IOMMU registers
2. Allocate context pages
3. Setup the IOMMU page tables during dma_map

For Linux, there are multiple dma_ops: nommu, iommu, and swiotlb. If Intel IOMMU is not present in the ACPI table, usually the swiotlb is used by default. SWIOTLB is like a bounce buffers, I don't really get why it is the default. LegoOS has the nommu version, which really does nothing.

So when you enable pass-through and expose a device to guest this is what happens:

- Host linux must have IOMMU enabled, thus intel-iommu's dma_ops is used.
- Guest linux may not have IOMMU present, thus swiotlb dma_ops is used
    - If guest linux has an emulated IOMMU from QEMU, then guest is also using intel-iommu's dma_ops.
    - Of course you can also control more via the intel_iommu command line
- The real physical IOMMU will be in charge of all the devices' DMA requests.

One thing I still don't understand: (Answer below, in VFIO section)

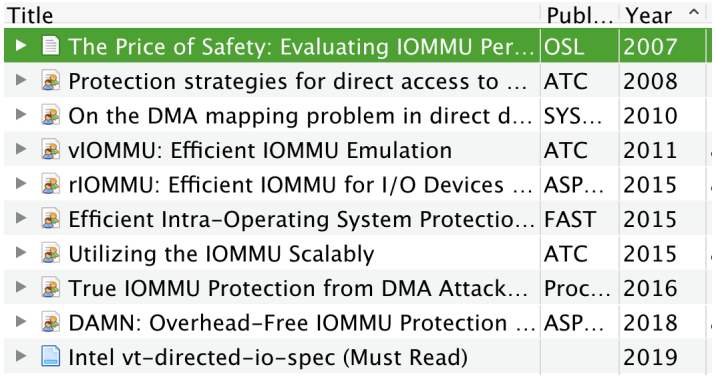
- before launching a DMA, when (and how) the guest asks the host to insert the gPA->hPA mapping into the physical IOMMU pgtable?
    - Maybe all pages allocated to guest is inserted into the IOMMU table?
    - Which ever dma_ops the guest is using, how would this be possible…

QEMU vIOMMU Emulation:

- The official link on vIOMMU emulation: Features/VT-d
- The vIOMMU is trying to emulate the effect of a real IOMMU! It emulates all the registers, translations, interrupt remapping etc. The mechanism is similar to emulating other devices: QEMU construct the ACPI DMAR table, so Linux thought there is a IOMMU present. QEMU will tell KVM to mark the register space of this vIOMMU as "not present", thus QEMU can catch all the accesses to vIOMMU via MMIO faults. Very interesting,.

- The Paper: vIOMMU: Efficient IOMMU Emulation, ATC'11 is doing the same thing, and proposed alternative solutions and optimizations. Not sure which one is first.
- Code is in: hw/i386/intel_iommu.c.
    - vtd_mem_write()
    - vtd_iommu_translate()

References:

- Intel Vt-d Specification, 2019
- Linux
    - Mastering the DMA and IOMMU APIs
    - Intel IOMMU driver analysis
    - IOMMU Groups, inside and out
    - Utilizing IOMMUs for Virtualization in Linux and Xen
        - SWIOTLB, NOMMU, IOMMU dma_ops.. Good stuff!
    - PCI Device Passthrough for KVM!
        - For a pass-through PCIe device, what's KVM's role in it? It seems KVM is still in charge of the configuration space (Marked as not present, thus catch EPT faults).
- Papers
    - On the DMA Mapping Problem in Direct Device Assignment, SYSTOR'10
    - **rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers, ASPLOS'15**
        - **Explains how the various Linux API interact with the IOMMU. This paper educated me a lot. MUST READ!**
    - vIOMMU: Efficient IOMMU Emulation, ATC'11
    - Efficient Intra-Operating System Protection Against Harmful DMAs, FAST'15
    - Thesis: Rethinking the I/O Memory Management Unit (IOMMU)
    - True IOMMU Protection from DMA Attacks: When Copy Is Faster Than Zero Copy, ASPLOS'16
    - DAMN: Overhead-Free IOMMU Protection for Networking, ASPLOS'18
    - https://www.usenix.org/legacy/event/usenix08/tech/full_papers/willmann/willmann_html/
    - 

Title	Publ…	Year
The Price of Safety: Evaluating IOMMU Per…	OSL	2007
Protection strategies for direct access to …	ATC	2008
On the DMA mapping problem in direct d…	SYS…	2010
vIOMMU: Efficient IOMMU Emulation	ATC	2011
rIOMMU: Efficient IOMMU for I/O Devices …	ASP…	2015
Efficient Intra-Operating System Protectio…	FAST	2015
Utilizing the IOMMU Scalably	ATC	2015
True IOMMU Protection from DMA Attack…	Proc…	2016
DAMN: Overhead-Free IOMMU Protection …	ASP…	2018
Intel vt-directed-io-spec (Must Read)		2019

VFIO

Funny that I missed vfio in the first place, such a critical piece. vfio exposes device's configuration spaces to the user space (via mmap of course), so that people could run user-level device drivers!

To understand that, you first need to understand how the driver talks with the device: PIO, MMIO, interrupt, and DMA. The most important thing of course is MMIO (assume PCIe devices): Interrupt/DMA will happen because the driver touched the configuration space in the first place. Vfio exposes the PCIe configuration space to userspace, so that people can write a device driver just like they have done in kernel space.

But, a DMA-capable device is able to write to anywhere. That's why we need IOMMU, that's why within the kernel, virtio subsystem is closely bound to the IOMMU part.

The use cases of vfio are straightforward. Both of them need to directly access device:

- Virtualization guest OS
- High performance user-level IO stacks such as DPDK

QEMU uses VFIO to directly assign physical devices to guest OS, and the command line option is `-device vfio-pci,host:00:05.0`. In fact, virsh PCIe device passthrough configuration eventually translates to the above QEMU option. (Check /var/log/libvirt/qemu/$vmname)

Note that QEMU itself has valid VA-PA mapping to access the physical device configuration spaces, which was established via mmap and ioctl. When QEMU launches the guest, it will expose this part of VA to the guest OS, thus when guest OS tries to access the physical device's configuration space, it will just through without any EPT VMEXIT. (The mechanism should be: qemu will register the PCI device's memory region via its memory_region_init_rw() API, which will finally ask KVM to setup the ETP pgtables)

As for the IOMMU mapping, QEMU will use ioctl to ask vfio driver to install the mappings. I think QEMU will simply map all guest's memory (gPA -> hPA), so whatever gPA is used by guest device driver, IOMMU has a valid PTE. Wow, this actually solved my concern!

(**the assumption is QEMU allocates the guest memory during start, i.e., just eager-allocate all memories. Is this true?**)

The usage of IOMMU is really smart here. Even though the user app can use ioctl to ask vfio kernel driver to setup some mappings, the user app is only allowed setup pages belong to itself! Thus IOMMU is safe, thus DMA is safe, thus the whole "userspace device driver" is space.

As for other interrupts, it says it was implemented as a file descriptor+eventfd. Sure, shouldn't be too hard. :p

References:

- qemu VM device passthrough using VFIO, the code analysis
    - I like this dude's blog, always answers my concern and doubt! This particular blog is explains how QEMU use vfio, and how QEMU exposes the mapping to guest OS so that guest can directly access physical device without vmexit.
- Kernel documentation: VFIO
    - This is from the designer, explains details on its implementation, specifially IOMMU.

Samecore v.s. Sidecore Emulation

Well, most QEMU/KVM device emulation is samecore, right? For instance, the serial device is definitely samecore, as you can see from the above QEMU/KVM code flow.

We also know that QEMU has some worker threads. The "main" thread will offload jobs to them, and the jobs include I/O related stuff, right? So maybe the state-of-the-art QEMU is already doing both samecore and sidecore emulation?

Well, depends on how you define "sidecore emulation", I guess. If it means "avoid vmexit" at all, then it's slightly different.

- vIOMMU: Efficient IOMMU Emulation, ATC'11
- Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization

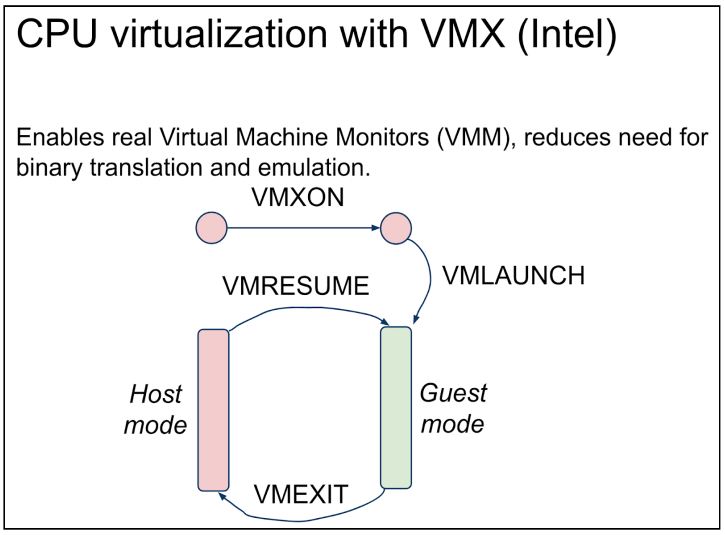
KVM:

Source code:

- KVM handles all vmexit directly, right? Yes.
- KVM source code has an array of handlers invoked upon vmexit. But not all of them will directly translate to a userspace KVM_EXIT_XXX.
    - `kvm_vmx_exit_handlers[exit_reason](vcpu)`
- So what's handled by KVM itself?
    - Looks like we can create some devices localling inside kernel via KVM APIs.
    - KVM_CREATE_PIT for the interrupt chip.

References:

- Architecture of the Kernel-based Virtual Machine (KVM), 2010
- kvm: the Linux Virtual Machine Monitor, 2007
- How VT-x, KVM, QEMU fit together, a good walkthrough!

- 

How to use KVM APIs:

- LWN: Using the KVM API
- KVMTool, should be a better start than QEMU. And this kvm-hello-world.
- QEMU source code: `accel/kvm`
- Rust KVM ioctls

libvirt and virsh:

- virsh/libvirt's main task is to convert XML to QEMU command lines. This maintains a consistent view to deploy VMs. No matter its QEMU or any other new VMMs.
- https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net