Practical Notes on the Virtualization Stack

-- QEMU, KVM, IOMMU, and more

Yizhou Shan
syzwhat@gmail.com

Created: Jan 25, 2020
Last Updated: Jun 30, 2021

Table of Content

1. Introduction

2. List of Open Source Projects

3. QEMU Source Code Study
- 3.1. References
- 3.2. Code Layout
- 3.3. High-Level Summary

4. I/O Device Virtualization
- 4.1. Models
- 4.2. QEMU Device Emulation Design
- 4.3. QEMU Device Emulation Implementation (Code)
- 4.4. QEMU + KVM Implementation Code Flow
- 4.5. virtio/vhost (Paravirtualization for devices drivers):
- 4.6. IOMMU
  - 4.6.1. dma_map
  - 4.6.2. QEMU vIOMMU Emulation
  - 4.6.3. Nested Translation in IOMMU
  - 4.6.4. Device-TLB
  - 4.6.5. References:
- 4.7. VFIO
- 4.8. Samecore v.s. Sidecore Emulation

5. KVM

6. libvirt and virsh

7. Misc Knowledge
- 7.1. Timer Interrupt and IPI delivery to VMs

1. Introduction

This is a scratchy and raw note about QEMU and KVM. It tries to explain how QEMU interacts with KVM, and some code snippets from both QEMU and Linux kernel KVM modules.

Honestly, I didn't fully understand the whole QEMU/KVM thing the whole time, until I decided to take a deep tour recently. End of the day, I'm satisfied, mostly. I now know how QEMU invokes KVM, how KVM launches guests, how KVM handles vmexit, and how vmexit translates to userspace-visible `KVM_EXIT_REASON` and so on. More importantly, I took a deep dive into how the device is emulated in QEMU. Specifically, IO and MMIO emulation.

2. List of 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.

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

3. QEMU Source Code Study

3.1. References

- 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 the QEMU device will "fire" interrupts, via Linux signals to the main loop.

3.2. Code Layout

- `**ui/**` has VNC code and more
- `**softmmu/main.c**` and `**softmmu/vl.c**` are the main entry files
- `**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**
  - You must understand this if you plan on hacking QEMU.
  - Guest Physical Memory Layout Management: `memory.c`
  - It looks like each device claims 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

3.3. High-Level 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 context*, 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.”

4. I/O Device Virtualization

This section walks through various bits on I/O virtualization.

4.1. Models

- ***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.

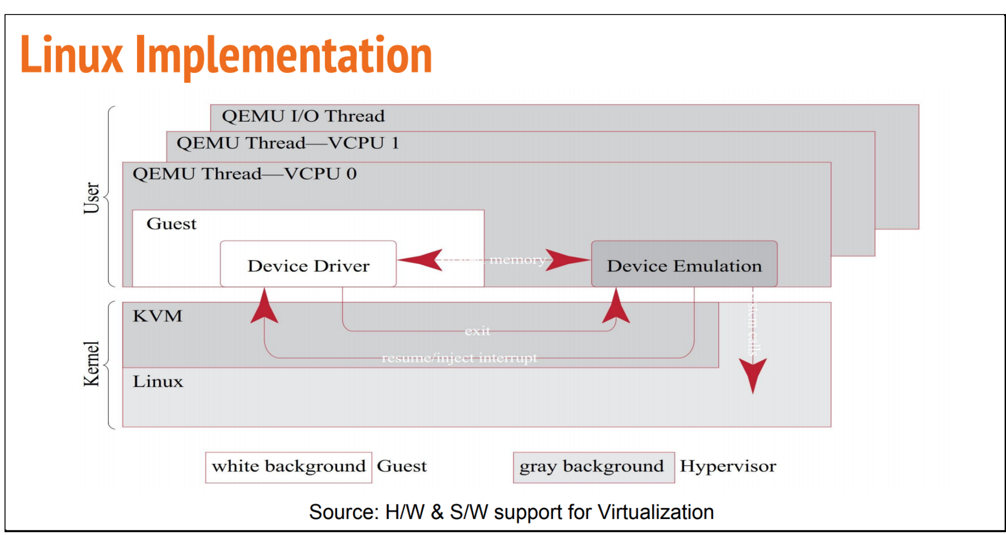
- ***Hardware-assisted Emulation***.
    1. **Direct Assignment**. Let a VM directly talk to device. Thus the guest device drivers can directly access the device configuration space to, e.g., launch a DMA operation. The device can DMA to physical memory in a safe manner, via IOMMU. This is enabled by 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”?

Further Reading

- 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

4.2. QEMU Device Emulation Design

- Understanding QEMU devices.
    - The ***best*** blog explaining how to emulate IO devices in general.
    - Think about how the 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 an API to let users 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
- IO Virtualization
    - This is the ***best*** slides explaining ****how QEMU/KVM handle MMIO/PIO**** and emulate devices! Basically, the guest driver writes to MMIO addresses, which causes pgfault and vm exit. KVM forwards this to QEMU.
    - End of the day, I now understand how QEMU manages memory models, how it associates physical memory with devices, how QEMU registers 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

Linux Implementation

- Each VM encapsulated in Qemu process.
- Each virtual core(VCPU) represented by a thread
  - Each VCPU thread has 2 execution contexts - guest VM and host QEMU
  - Host context - for handling exits of guest VCPU context.
- Qemu creates "IO thread" for each virtual device.
  - IO thread handles asynchronous activity like handling network packets
- Here , there are 2 VCPUs and one virtual device
- Guest VM device driver issues MMIO/PIO instructions to drive the device - directed at read/write protected memory locations - suspend VCPU context - invoke KVM
- KVM relays events to same thread but to the host execution context
- Events are handled by the device emulation layer of host context through regular system calls
- Device emulation layer emulates DMA by read/write from guest IO buffers - accessible through shared memory
- Resumes guest execution context via KVM injecting interrupts to signal the guest about IO operation

  -
- 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 provides both. They are both the backend for the guest virtio drivers.

4.3. QEMU Device Emulation Implementation (Code)

In this section, I want to take a tour on how QEMU works without accelerators like KVM. I think at this point we already know the basic designs. Hence this section will focus on code details. (Added on Jun 19, 2021.)

References

1. QEMU's instance_init() vs. realize() (redhat.com)

4.4. QEMU + KVM Implementation Code Flow

For QEMU, KVM is a kind of accelerator. By default, QEMU uses binary translation. With KVM, QEMU is able to run native instructions. To do so, QEMU must interact with the linux kvm via ioctl. Also, with KVM, the device emulation flow is slightly different as it would trap to the host kernel then bounce back to usespace QEMU.

Also note there are many examples demonstrating the KVM flow. QEMU-KVM is one, the rust-virt vmm is also one. Generally this is where you should start from if you want to write your own virtualization.

- First, QEMU's device models use `*memory_region_init_io/rw*()` to declare the IO ports and physical memory addresses each device is operating upon. The device will provide a set of callbacks when the read/write access happens.
- 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.)

4.5. 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!

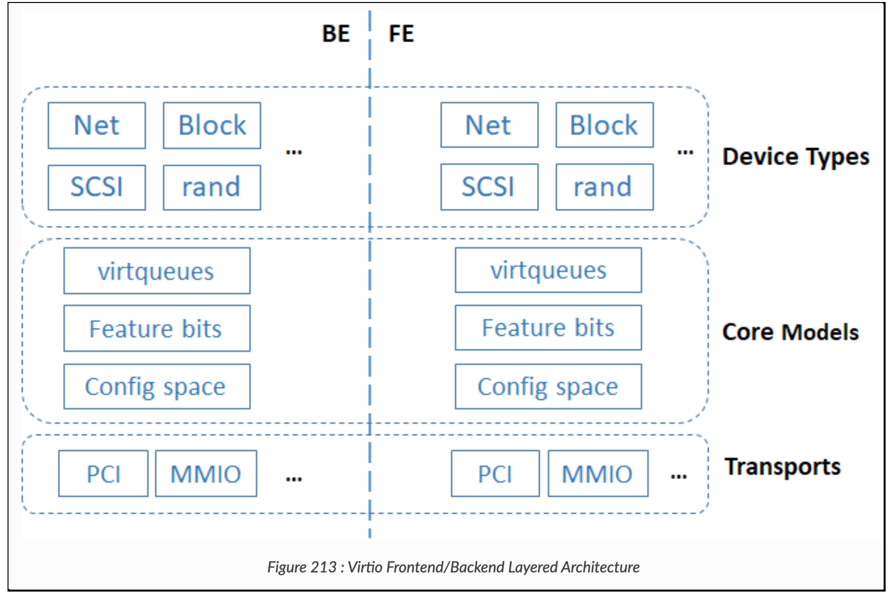
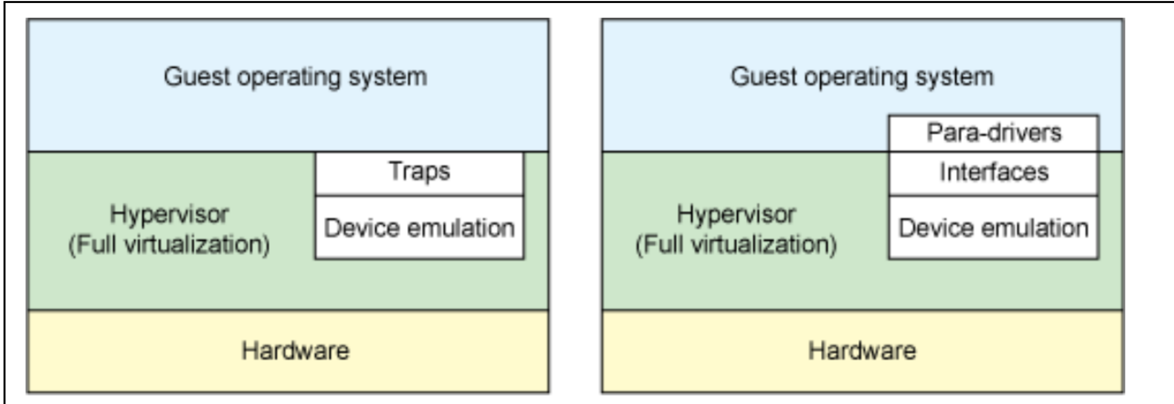
- And note that the backend drivers can be implemented in either kernel-space or user-space, it's just an implementation choice, as long as the backend driver follows the protocol. However, note the difference: kernel-based backend driver incurs less user-kernel switches:
- 

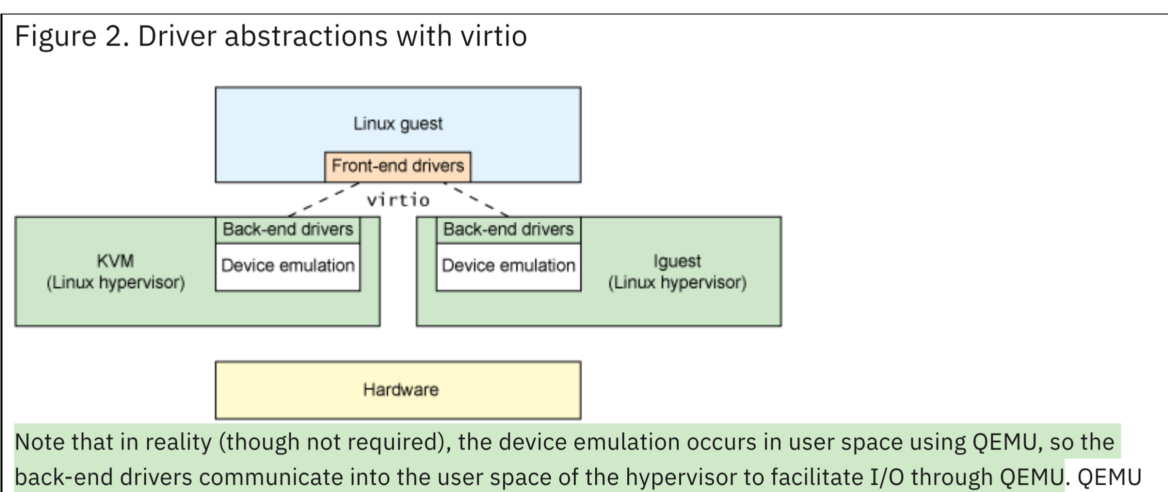
Figure 213 : Virtio Frontend/Backend Layered Architecture
- **Project ACRN: Virtio devices high-level design**
  - **One of the BEST BLOGs explaining virtio.**
- **QEMU Internals: vhost architecture**
  - **One of the BEST BLOGs explaining virtio.**
  - "Vhost does not emulate a complete virtio PCI adapter. 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*

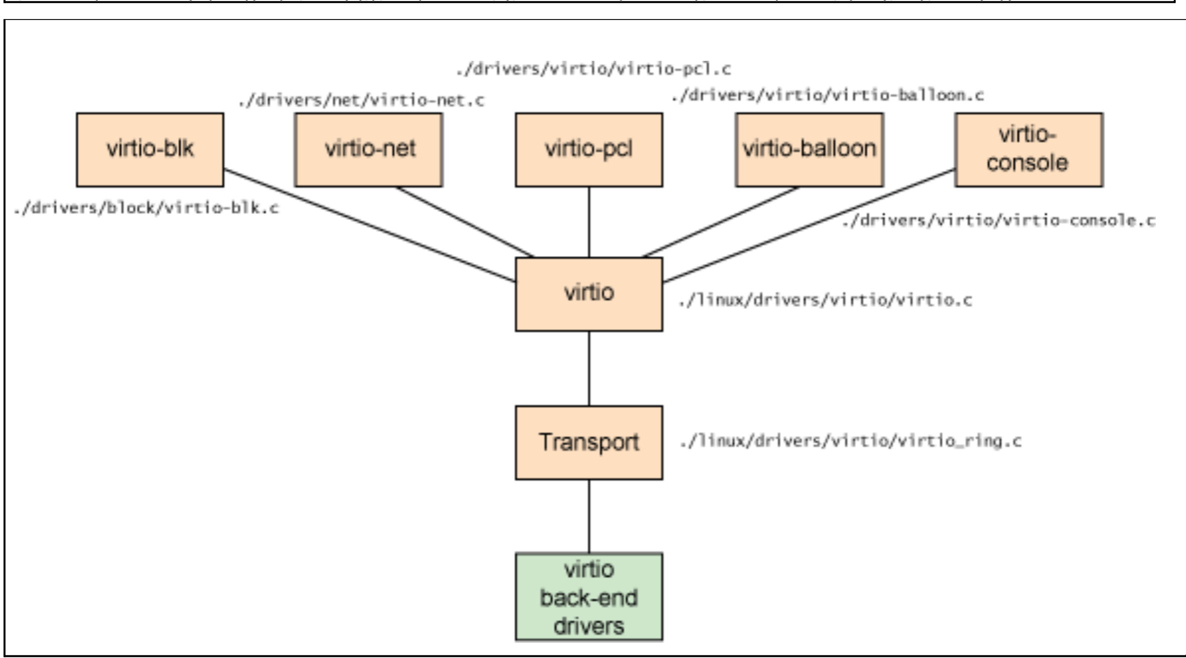
write to a specific memory address. I think this address will be an MMIO address, which causes vmexit, and then KVM will pass it to QEMU!)

    - Original virto flow:

- [Virtio: An I/O virtualization framework for Linux](#)

- 

- 

- 

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.)

4.6. 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 talk with IOMMU via MMIO register access. The IOMMU is involved in every DMA related operation, 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. (See the following QEMU vIOMMU link for why we need this)

Intel IOMMU is very flexible. Each device can have more than one associated address space, presumably each client VM can install their address space onto a device. In addition, the IOMMU can translate from not only gPA, but also gVA (think about RNIC in virtualized usage!), and some others. This is described int Intel IO-d spec:

- Each device can have number of PASID domains
- Each domain can have multiple modes, first-level, second-level, or nested (in case of RNIC!)

Figure 3-8 illustrates device to domain mapping with scalable-mode context-table.

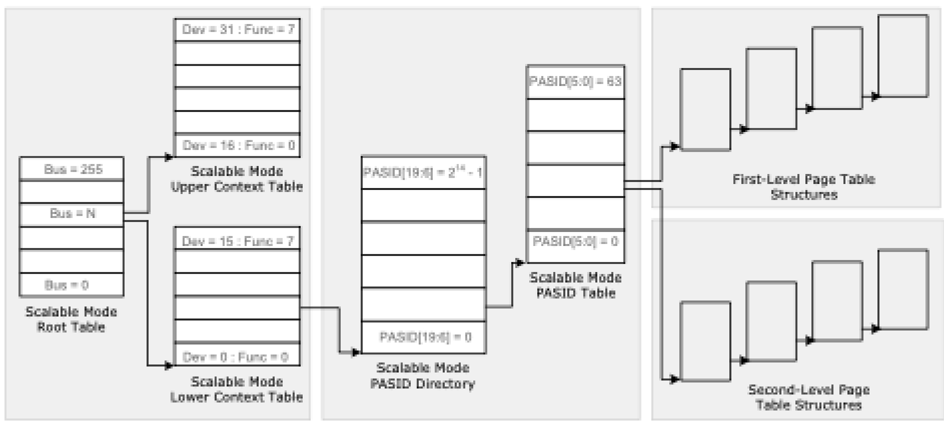


Figure 3-8. Device to Domain Mapping Structures in Scalable Mode

The scalable-mode root-entry format is described in Section 9.2, the scalable-mode context-entry format is described in Section 9.4, the scalable-mode PASID-directory-entry format is described in Section 9.5, and the scalable-mode PASID-table entry format is described in Section 9.6.

- Thanks to the ISCA'20 paper **HyperTRIO: Hyper-Tenant Translation of I/O Addresses** that reminds me of this:

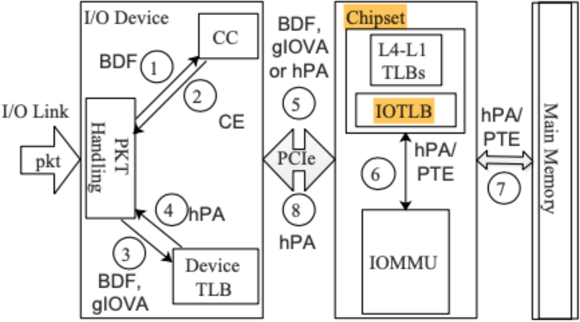


Fig. 3: Translation steps performed for gIOVA triggered by an incoming packet to a device. BDF - PCIe Bus/Device/Function triplet. CC - Context Cache. CE - Context Entry. gIOVA - guest I/O Virtual Address. hPA - host Physical Address. IOMMU - I/O Memory Management Unit. IOTLB - I/O Translation Lookaside Buffer. PTE - Page Table Entry.

4.6.1. 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 buffer, 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…

4.6.2. 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()

4.6.3. Nested Translation in IOMMU

IOMMU can also do GVA -> HPA translation using its two-level (nested) translation. The device needs to work with physical IOMMU. The device can send requests to IOMMU to request address translation.

To utilize nested translation in IOMMU, there must be a companion vIOMMU from QEMU exposed to guest VM. And QEMU needs to intercept and record whatever changes the

guest is trying to make to vIOMMU, the guest is trying to do gVA->gPA mapping. Then QEMU will install that onto the physical IOMMU.

[RFC Design Doc v3] Enable Shared Virtual Memory feature in pass-through scenarios
Shared Virtual Memory in KVM

- "Shared Virtual Memory feature in pass-through scenarios is actually SVM virtualization. It is to let application programs(running in guest)share their virtual address with assigned device(e.g. graphics processors or accelerators)"
- Okay, I think this SVM feature is useful for devices that DO NOT have any VA->PA capabilities. By enabling this, these devices can directly use gVA while the underlying IOMMU will take care of two level translation (gVA->gPA->hPA).
- However, for devices that already have VA->PA capability (e.g., RDMA NIC, GPUs), this is not useful, or rather, redundant. They can work with a normal IOMMU. So the device does gVA->gPA, then IOMMU does gPA->hPA.

QUESTION (see paragraph above): It seems RDMA/GPU cards can both use this feature, right? But I've never seen anyone mention this before. My impression is always that RDMA/GPU cards will do their own GVA to GPA translation and not care whether there is iommu present.

- I don't really think Mellanox NIC is using this. From the slide above, it says that the guest OS must interact with vIOMMU (by QEMU), and then the emulated vIOMMU will interact with physical IOMMUto set up things.
- We don't have any IOMMU code in LegoOS but we are still able to run LegoOS +passthrough RNIC. That means RNIC is doing its own GVA -> GPA and then physical IOMMU doing GPA -> HPA translation.

Now the question is to find out who is actually using this nested IOMMU translation.

4.6.4. Device-TLB

The Intel vt-d spec talks about Device-TLB. As its name suggests, devices can cache some entries in their chip! There is a protocol between the device and IOMMU.

- IOMMU exposes Address Translation Service (ATS)
- Device can ask IOMMU to translate sth and send the result back to device, which will then cache it locally.
- There are also shootdown mechanisms.

Do note, this is totally different from RDMA/GPU's own VA->PA translation facility.
This one is IOMMU specific and is generic to all devices who want to use it. And it seems it is used for the nested translation above (gVA->hPA).

- I don't think RDMA/GPU is using this.

4.6.5. References:

- Virtualizing IO through THE IO Memory Management Unit (IOMMU),
    - This is a very detailed slide.

- [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).
- Nested
    - [[RFC Design Doc v3] Enable Shared Virtual Memory feature in pass-through scenarios]()
    - [Shared Virtual Memory in KVM]()
- 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

4.7. 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!

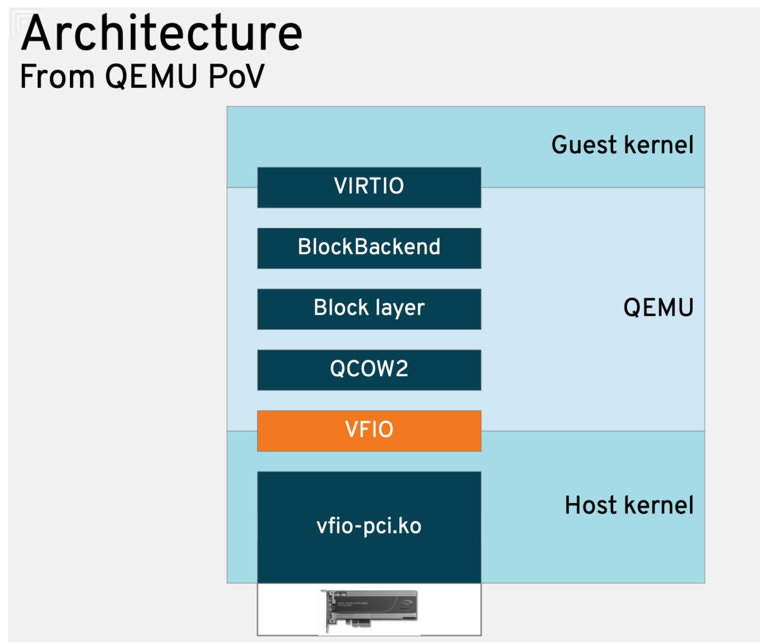
(**the assumption is QEMU allocates the guest memory during start, i.e., just eager-allocate all memories. Is this true?**)

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, specifically IOMMU.
- https://events.static.linuxfound.org/sites/events/files/slides/Userspace%20NVMe%20driver%20in%20QEMU%20-%20Fam%20Zheng_0.pdf
    - 

4.8. 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, it 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

4.9. AWS Nitro/Microsoft

The latest in this design space is to use dedicated hardware for IO virtualization, like AWS Nitro and Microsoft SmartNIC.

The device exposes a unique partition/MMIO to each guest. Guest sees a passthrough raw device. Within the device itself, there will be stack handling necessary virtualization tasks. These tasks are similar to what QEMU was doing.

2.4 I/O Virtualization

A VMM must support virtualization of I/O requests from guest software. I/O virtualization may be supported by a VMM through any of the following models:

- *Emulation*: A VMM may expose a virtual device to guest software by emulating an existing (legacy) I/O device. VMM emulates the functionality of the I/O device in software over whatever physical devices are available on the physical platform. I/O virtualization through emulation provides good compatibility (by allowing existing device drivers to run within a guest), but pose limitations with performance and functionality.
- *New Software Interfaces*: This model is similar to I/O emulation, but instead of emulating legacy devices, VMM software exposes a synthetic device interface to guest software. The synthetic device interface is defined to be virtualization-friendly to enable efficient virtualization compared to the overhead associated with I/O emulation. This model provides improved performance over emulation, but has reduced compatibility (due to the need for specialized guest software or drivers utilizing the new software interfaces).

- *Assignment*: A VMM may directly assign the physical I/O devices to VMs. In this model, the driver for an assigned I/O device runs in the VM to which it is assigned and is allowed to interact directly with the device hardware with minimal or no VMM involvement. Robust I/O assignment requires additional hardware support to ensure the assigned device accesses are isolated and restricted to resources owned by the assigned partition. The I/O assignment model may also be used to create one or more I/O container partitions that support emulation or software interfaces for virtualizing I/O requests from other guests. The I/O-container-based approach removes the need for running the physical device drivers as part of VMM privileged software.
- *I/O Device Sharing*: In this model, which is an extension to the I/O assignment model, an I/O device supports multiple functional interfaces, each of which may be independently assigned to a VM. The device hardware itself is capable of accepting multiple I/O requests through any of these functional interfaces and processing them utilizing the device's hardware resources.

Depending on the usage requirements, a VMM may support any of the above models for I/O virtualization. For example, I/O emulation may be best suited for virtualizing legacy devices. I/O assignment may provide the best performance when hosting I/O-intensive workloads in a guest. Using new software interfaces makes a trade-off between compatibility and performance, and device I/O sharing provides more virtual devices than the number of physical devices in the platform.

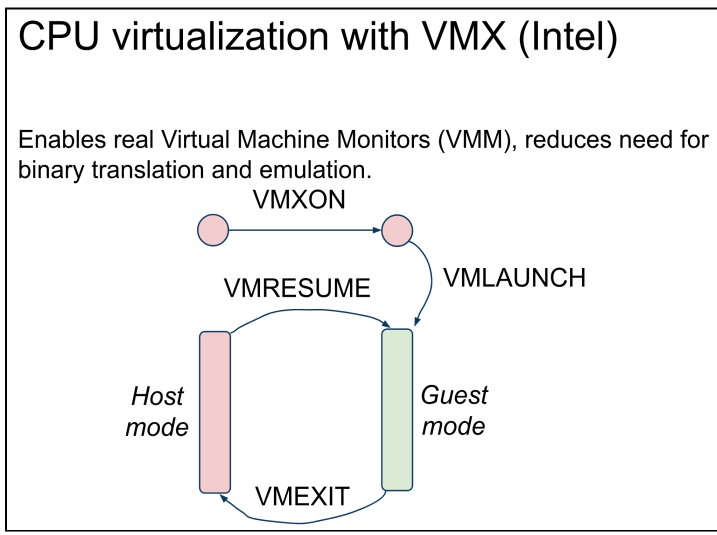
5. KVM

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!

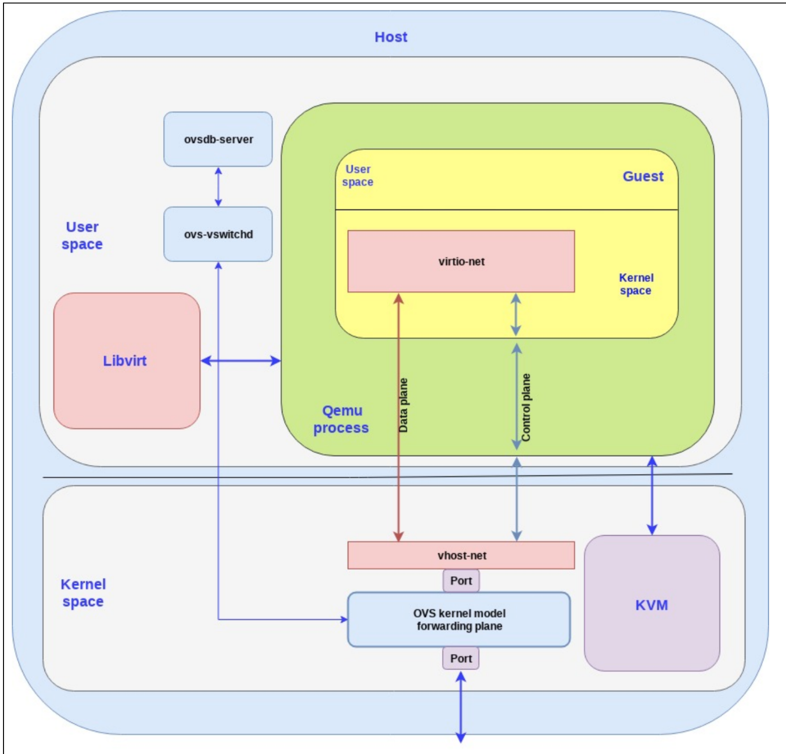
- 

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]

6. 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



7. Misc Knowledge

7.1. Timer Interrupt and IPI delivery to VMs

So today (Arp 2, 2020) I came across this VEE'20 Directvisor paper, which is trying to deploy a very thin hypervisor to bare-metal cloud to regain some manageability. The general approach is to reduce VM exit (by disabling the vt-d feature) and try to deliver interrupts to VM directly without VM exit. It talks about the flow of the current timer interrupt and IPI delivery mechanism, I found it useful thus post it here:

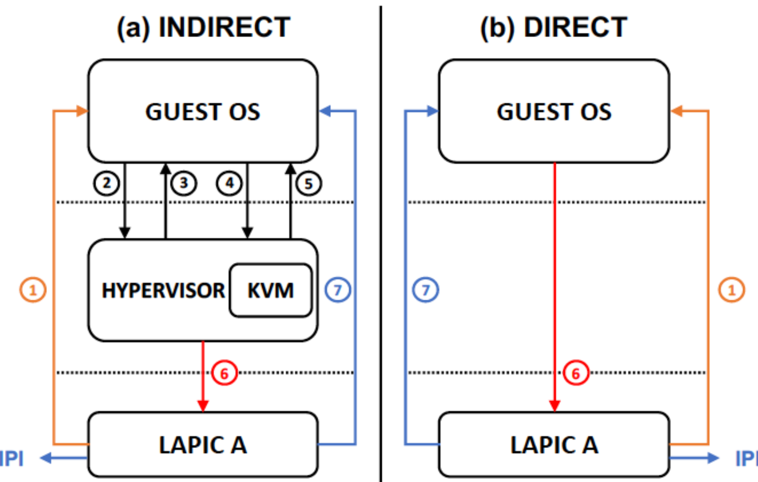


Figure 2. Indirect vs. direct timer interrupts and IPIs.

(see Section 9). Directvisor eliminates both overheads by disabling VM exits when a guest's virtual CPU executes the HLT instruction; thus, the physical CPU directly idles in guest mode without any emulation or polling overheads.

4 Direct Timer Interrupts and IPIs

We next describe how Directvisor enables a VM to directly control and receive local timer interrupts and IPIs. Consider Figure 2 which compares the relative costs for indirect and direct delivery of timer interrupts and IPIs to a VM. Traditional indirect processing of timer interrupts and IPIs is expensive due to several VM exits and entries. A timer interrupt (1) triggers a VM exit (2). The hypervisor injects a virtual timer interrupt into the guest upon VM entry (3). The guest processes the interrupt and attempts to program the next timer interrupt, which triggers another VM exit (4) and VM entry (5). The hypervisor emulates the guest's request by programming the local APIC (6). For IPIs, a guest's request to send an IPI from one virtual CPU to another is still trapped into the hypervisor (4, 5, 6). However, if the processor supports posted interrupts, then the IPI is delivered to the target virtual CPU without any VM exits (7). As shown in Figure 2b, Directvisor completely removes itself from the guest's interrupt processing path by configuring the DirectVM to use the posted interrupt mechanism for both local timers and IPIs. Next we describe these two mechanisms in more detail.