I/O Virtualization

Luca Abeni
luca.abeni@santannapisa.it

March 9, 2020

I/O Devices

- I/O devices are generally accessed through registers and memory-mapped buffers
  - Registers to read/set the device state and send commands to the device
  - Memory buffer to transfer large amount of data
  - The device can act as a *bus master* to move data from/to the memory buffer
- Device registers: either memory-mapped, or in their own I/O address space
  - Accessed through the `in` and `out` assembly instruction in the Intel architecture
- The device can also raise interrupts to notify the OS kernel about something

I/O Devices Virtualization

- How to handle I/O in a VM?
  - The hypervisor/VMM virtualizes I/O devices!
  - Handle accesses to the devices' registers, move data, generate (virtual) interrupts, ...
- Device registers in the I/O space: the machine instructions accessing them are sensitive
  - Example: on Intel x86, `in` and `out` can be trapped by the hypervisor
- Memory-mapped registers: must be in a read-only (or privileged) memory page
  - When the guest access them, a page fault is generated...
  - ...And the hypervisor can trap it!

Accesses to Virtual Device Registers

- Virtual device: the hypervisor intercepts accesses to registers
  - Again, trap and emulate!
- VM exit every time the guest acceses a register of a virtual device
  - Bare-metal hypervisor: the hypervisor can emulate the device by itself...
  - Hosted hypervisor: the device is often emulated by a user-space program (VMM/DM)
  - Bare-metal hypervisors can sometimes use a helper process running in a “special guest” (example: QEMU DM, running in Xen Dom0)

I/O Virtualization Overhead

- For bare-metal hypervisors: VM exit
  - Save the guest state, invoke hypervisor, restore guest state)
- For hosted hypervisors, larger overhead
  - Also including (host) kernel / userspace switches
  - Example: QEMU/KVM. When the guest accesses a register, KVM exit: VM exit, the KVM driver executes, switch to userspace and schedule QEMU, handle the KVM exit, switch back to kernel space, and finally restore the guest
- Bare-metal hypervisor using a userspace DM, running in a guest: the overhead is even higher

Virtualizing Real Devices

- Existing real devices might have complex interfaces
  - Lots of registers to be virtualized
  - Look at the device documentation for all the details
- The hypervisor (or VMM/DM) has to emulate all of them
  - Some features might be useless for VMs (example: line-speed negotiation for a NIC)
- The protocol for handling the device can be virtualization-unfriendly
  - Example: when handling an interrupt, the device driver might need to read/write multiple registers

Virtualizing Real Devices — Overhead

- Correctly emulating a real device might introduce a lot of overhead/complexity
  - Lot of complexity (even for backward compatibility / hystorical reasons) not really needed in virtual environments
- Interfaces/protocols designed to optimize the performance of physical hardware, not VMs/hypervisors!!!
  - Lot of register accesses $\rightarrow$ become VM exits
  - The hypervisor/VMM/DM might have to copy a lot of data between host and guest address spaces
- Hardware and software can be asynchronous, writes to virtual registers are often synchronous

Paravirtualized Devices

- Most of the mentioned issues can be addressed by using a different host/guest interface
  - Instead of emulating a real physical device, design a virtual device from scratch
  - The guest must be aware that it runs in a VM, to provide drivers for the new device $\Rightarrow$ paravirtualization!
- Paravirtualized devices designed to reduce virtualization overhead
  - Reduce the device complexity and the number of virtual registers
  - Reduce the amount of VM exits
  - Allow to share buffers between host and guest

Paravirtualization Mess

- There is not a standard anymore!
  - Every different hypervisor/VM defines its own devices...
  - Guest OSs/kernels must provide drivers for all of these devices!
- Example: network card
  - The VMM can emulate an Intel e1000
  - Well-defined “standard”: all OSs support it
  - ...But performance are not so good
  - The VMM provides its own virtual NIC $\rightarrow$ better performance...
  - ...But every OS/kernel must write drivers for it
- Need for a virtual devices standard!!!

Virtio

- Paravirtualization standard, usable for many different devices
  - Provides standard interfaces and mechanisms upon which different kind of devices can be built
  - Here, focus on network and block devices
  - Can be seen as a standard message-passing interface between guest and VMM/hypervisor
- Standard designed to address the explosion of paravirtualized devices
  - Differen hypervisors/VMMs can support it
  - Providing drivers for virtio devices, the guest does not need to care about the VMM details

Virtio Design Goals

- Generic enough to support different kinds of devices (network, block, video, ...)
- Not bound to any specific hypervisor or guest OS
  - Different hypervisors/VMMs implement virtio devices
  - Different guests provide virtio drivers
- Reduce the number of register accesses
- Reduce the number of interrupts
- Use shared memory buffers to exchange information without VM exits
- Allow NAPI-like techniques to process as much data as possible before blocking
- Asynchronous operations using different threads

Design — More Details

- The interface/abstractions should be “compatible” with the internal data structures used by guest kernel and host hypervisor
  - Example: skb for guest network packets
- The host/guest interface has to support fragmented buffers!
- Use “Scatter-Gather lists” (SG)
  - Lists of buffers described as (base,limit) couples
  - Base: physical address
- The SG implementation can be smarter than a simple list
  - Allow lock-free access, ...

The VirtQueue Abstraction

- VirtQueue (VQ): transport abstraction used by virtio
  - It is a queue of SGs
- The guest (virtio driver) *posts* (inserts) SGs (buffers) in the VQ
- The host (VMM/hypervisor) *consumes* the SGs in the VQ
  - Pushes back SG lists as responses
- There are output SGs (used by the driver to send data) and input SG lists (used by the driver to receive data)
- A virtio device contains one or more VQs

VirtQueue Interface — 1

- `add_buf`: used by the guest (virtio driver) to add SGs in the VQ
  - These are commands sent to the virtio device (to the VMM/hypervisor)
  - A *token* is associated to each SGs to support out-of-order replies
- `get_buf`: used by the guest (virtio driver) to cleanup SGs
  - Previously added to the VQ by the guest
  - Already processed by the host
  - Used to receive responses from the virtio device
- `kick`: used by the guest to notify the host that SGs have been added

VirtQueue Interface — 2

- The host answers to `kick` by consuming SGs posted in the VQ by the guest
- Then the host somehow notifies the guest, and the guest cleans up the SGs
  - Notice that the VQ *interface* does not specify the notification mechanism used by the host
  - This notification mechanism will be specified in the implementation
- The guest can poll on `get_buf` until the host sends notifications, or wait for notifications in some way
- Notifications from the host can be disabled with `disable_cb` and re-enabled with `enable_cb`

Virtio and Throughput

- The throughput of virtio devices can be improved by (large) batch processing
  - The guest (virtio driver) should enqueue as many buffers as possible before kicking the host
  - The virtio device (the host) should consume as many buffers as possible before sending back notifications to the guest
  - Risk to increase the latency!!!
- Host thread (thread in the hypervisor/VMM/DM) to serve the guest `kick`s
- Guest thread (thread in the virtio driver) to serve the host/device notifications
  - The two threads can have a NAPI-like behaviour

Implementing the VirtQueue Abstraction: virtio_ring

- virtio_ring: VQ implementation, based on an array of descriptors (actually, a ring buffer)
  - Descriptor: base, size, flags, index of the next descriptor in SG
  - Next is for creating a linked list
- Array (ring buffer) of descriptors ready for use, posted by the guest: ready ring
  - This array is only manipulated by the guest
- Array (ring buffer) of descriptors already processed (consumed) by the host: used ring
  - This array is only manipulated by the host
- This smells lock-free!!!

Implementation Details

- The virtio_ring implementation also specifies the details of guest/host notifications
- Guest notifications to host: `kick` $\rightarrow$ performed by writing in a (virtual) register
  - Only one register write after posting buffers
  - Reduce the number of VM exits
- Host notifications to guest $\rightarrow$ performed by sending an interrupt to the guest
  - Interrupt handled by the virtio driver
  - Can use a NAPI-like thread, can disable interrupts, ...

Example: virtio-block

- One single VQ for reading and writing
- Every buffer is compsed by at least 3 parts
  - A header (which is read-only for the host)
  - A data buffer (read-only or write-only for the host, depending on the type of request)
  - A status byte (success, error, or unsupported; write-only for the host)
- Example: read operation
  - The guest allocates the 3 buffers and uses 3 (linked) descriptors for them
  - The index of the first descriptor (header) is inserted in the ready ring
  - `kick`; the host is notified

virtio-block Read — Continued

- The host reacts to the `kick` by consuming the SG
  - Find the index of the header descriptor in the ready ring
  - Reads the header, copies data to the data buffer (linked by the header descriptor) and writes the status byte (linked by the data descriptor)
  - The index of the first descriptor (header) is inserted in the used ring
  - An interrupt is generated for the guest
- The guest serves the interrupt
  - Find the header index in the used ring
  - Copy and use the data
  - Cleanup

Example: virtio-net

- At least a VQ for rx and a VQ for tx
- Packet transmission: the driver transforms an skb into an SG posts it in the tx VQ
  - List to cope with fragmented packets!
  - Inserted in tx VQ by adding the index of the first descriptor in the ready ring
- `kick`
- The consumes the SG, sending the packet (example: QEMU writes to TAP, or similar)
- The index of the first descriptor is inserted in the used ring, and an interrupt is sent to the guest
- The driver can then cleanup the tx SG processed by the host (for example, packets sent by QEMU)

virtio-net: Receiving Packets

- The driver posts free buffers (the skb buffers) in the rx VQ
  - Insert the descriptors' indexes in the ready ring
  - `kick`, and do something else waiting for interrupts
- When a packet arrives, the host consumes an SG from the rx VQ
  - Find the index of the first usable buffer in the ready ring
  - Copy the packet in tbe buffer
- Then, pushes the SG back in the rx VQ
  - Insert the index in the used ring
  - Send an interrupt to the guest

Receiving Packets — Continued

- The driver cleans up the SG, receiving the packet
  - Get the descriptor index from the used ring
  - Allocate a new skb and post its buffer(s) in the rx VQ (replacing the ones of the received packet)
- Notice: the packet is already in the skb buffers
  - The host copied it there
- No locking (or, very simple locking!)

A Linux-Specific Optimization

- Possible setup: KVM driver (hypervisor) + QEMU userspace process (VMM or DM)
- Every time the guest wants to read/write some data:
  - At least one register access $\rightarrow$ VM Exit
  - Handled by KVM in kernel space, then KVM Exit
  - QEMU is scheduled to handle the KVM exit; moves some data and then restart `KVM_RUN`
  - KVM executes in kernel space again
  - The guest is restarted (interrupt handler)
- For complex devices, there are no alternatives...
- ...But for virtio QEMU is scheduled just to do a little bit more than `memcpy()` !!!
  - Can we do something better (more optimized)?

User-Space DM

- Advantage: ,ove complexity to userspace (more secure, ...)
- Disadvantage: more overhead
- For virtio, one register write (kick) when new buffers are in the VQ
- The DM just has to consume the buffers, copying some data
    - Example: to send packets through virtio-net, copy them from VQ to a TAP device
- Maybe this data movement can be performed in kernel space?
    - Without involving user-space prcesses!

Vhost

- Vhost: kernel-space implementation of virtio
  - Kernel thread moving data from/to the VQ
- Example: vhost-net $\rightarrow$ `vhost-net` kernel thread copying buffers between VQ and a TAP-like device
  - Standard TAP device, macvtap, ...
  - Does not avoid VM exits, but avoids KVM exits
  - Can avoid a lot of kernel-space/user-space switches
- Can improve virtio throguhput (and reduce latency) by moving functionalities to the kernel

Vhost-User

- Vhost idea: virtio implementation out of QEMU
  - More in general, out of the user-space DM
  - Original vhost: use a kernel thread
- Vhost-user: implement virtio in an external user-space process
  - Example: for the network, implement in a user-space vswitch
  - Instead of using a kernel thread to copy packets tp a TAP device and read them from a vswitch, process, implement virtio in the vswitch
- Isn’t vhost-user re-introducing overhead?
  - Kernel-space/user-space switches, signalling via sockets, ...

Vhost-User Performance

- User-space implementation of vhost
- Use unix-domain sockets for signalling
- Use shared memory buffers for virtio_ring
  - Memory buffers shared between guest and vhost-user process...
  - Instead of relying on signalling, the vhost-user process can busy-wait (poll) for buffers in the virtio_ring...
  - The vhost-user process does not block $\leftarrow$ no user-space/kernel-space switches
- Exitless virtio implementation!
  - Kicks and interrupts are not needed
- Example: implementation based on DPDK PMD