Passthrough in QEMU/KVM on Linux

G. Lettieri

2 Nov. 2017

Let see how hardware passthrough can be used in practice, using the QEMU hypervisor with the KVM API on Linux. We assume Intel hardware throughout (Intel VMX and Vt-d). We want to passthrough an Intel 82598 10 Gb/s Ethernet NIC (Network Interface Card) to a guest running inside QEMU (with KVM enabled).

1 Linux setup

First, let us see what we need to do on the host system.

1.1 Enabling the IOMMU

Once the IOMMU has been enabled in the BIOS, we also need to tell the Linux kernel to use it, since it is typically disabled by default.

We need to pass the `intel_iommu=on` command-line argument to the kernel at boot. How to do this depends on the boot loader and the Linux distribution we are using. On Ubuntu we can modify the the `/etc/default/grub` file, adding the command to the `GRUB_CMDLINE_LINUX_DEFAULT` environment variable, e.g.:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash intel_iommu=on"
```

Then we need to run `update-grub` (as root) and reboot. We can check that the IOMMU has been activated by looking at the output of

```
dmesg | grep DMAR
```

The `dmesg` command shows on its standard output the contents of the kernel log. We pipe the output into `grep` to look for log messages containing the "DMAR" string (DMAR stands for DMA Remapping, which is what the IOMMU is used for). If the IOMMU has been correctly enabled, we should see several lines of output.

1.2 Attaching the device to the `vfio-pci` driver

To passthrough a device to a Virtual Machine we first need to detach it from its normal driver on the host and attach it to a special driver called `vfio-pci`.

The standard way to interact with drivers in Linux is through the `sys` pseudo-filesystem. A pseudo-filesystem in Linux is something that looks almost exactly like any other file system, with directories and files, but these actually give access to kernel data-structures, rather than to data in secondary storage. A typical Linux installation will mount many of these pseudo-filesystems. One of the oldest examples is the one usually mounted in `/proc`. The `proc` pseudo-filesystem contains a subdirectory for each task in the system; each subdirectory in turn contains many pseudo-files, symbolic links and other directories, giving a lot of information about each task.

The advantage of pseudo-filesystems is that we can use all the ordinary Unix tools and shell commands (`cd`, `ls`, `cat`, ...) to browse, read and update the informations. For example, if we know the pid of a process (let us call it P) and we have sufficient privilege, we can see the path of all its open files with

```
ls -l /proc/P/fd
```

For our purposes, we need to interact with the pseudo-filesystem typically mounted on `/sys`. This contains subdirectories for every device and every driver installed in the system. If we limit ourselves to PCI devices, we can take a look at the `/sys/bus/pci` directory. Here we find the `devices` and `drivers` subdirectories.

- The `drivers` directory contains a subdirectory for each loaded PCI driver. In our example, we are interested in `ixgbe` (the host driver for the Intel 82598 card) and `vfio-pci`.

- The `devices` subdirectory contains a symbolic link for each PCI device. The links are named according to the following schema

  PCI-domain:*bus*:*device*.*function*

  where *PCI-domain* is usually `0000`. The other numbers make up the standard triple that identifies each device in a PCI system. Assume that the card we are interested in is `0000:01:00.0` (found using a tool like `lspci`).

The `driver` symbolic-link in `/sys/bus/pci/devices/0000:01:00.0` is now pointing to `/sys/bus/pci/drivers/ixgbe` (through a relative path). We want it to point to `/sys/bus/pci/drivers/vfio-pci`, but we cannot just overwrite the link. Instead, we need to use the `bind` and `unbind` pseudo-files in the drivers directories. By writing `0000:01:00.0` into the `unbind` file of the `ixgbe` driver we detach our device from its current driver. Then, by writing `0000:01:00.0` into the `bind` file of the `vfio-pci` driver, we attach it to the driver we need for passthrough.

Before doing that, however, we need to tell the `vfio-pci` driver that it can accept devices like our network card. This is done using the `new_id` pseudo-files found in all drivers directories. These files accept strings of the form "*xxxx yyyy*" where *xxxx* is the *Vendor ID* and *yyyy* is the *Device ID* of a PCI device (the first two registers of the device in the PCI configuration space). We can find the correct values for our device by looking at the `vendor` and `device` pseudo-files in the device directory:

```
$ cat /sys/bus/pci/devices/0000:01:00.0/vendor
0x8086
$ cat /sys/bus/pci/devices/0000:01:00.0/device
0x10b6
```

In our example we obtain the Vendor ID for Intel (8086) and the Device ID for 82598 cards (10b6). We can now proceed to the driver change:

```
$ echo 8086 10b6 |
    sudo tee /sys/bus/pci/drivers/vfio-pci/new_id
8086 10b6
$ echo 0000:01:00.0 |
    sudo tee /sys/bus/pci/drivers/ixgbe/unbind
0000:01:00.0
```

Now the `driver` symlink in the device directory should point to `vfio-pci`. (In this particular case we don't need to write into the `bind` file of `vfio-pci` since, once we have told it that it can handle our device, it will grab it as soon as the `ixgbe` driver releases it). We can also look at `dmesg`, where we should see messages from `ixgbe` releasing the device. An `ifconfig -a` will also no longer show the network interface, which has now been hidden from the host network stack.

2 QEMU setup

QEMU was originally written by Fabrice Bellard as a very general emulator based on binary-translation. In 2008, Avi Kivity introduced the `kvm` module in Linux and forked the project into `qemu-kvm` to use the new API. The KVM support was later backported into the original QEMU, where we can find it today.

By default, QEMU still uses binary-translation. KVM support must be enabled from the commandline with either `-enable-kvm` or the more general `-machine accel=kvm`.

QEMU uses the front-end/back-end model for most of its emulated devices. For networking, it emulates several network cards (Intel e1000, Realtek rtl8139, and many others; recent versions also emulate the Intel 82598 we are trying to passthrough). These are the front-ends. It also implements several networking back-ends (called netdevs): user, tap, socket, and so on. Any emulated network card can be attached to any netdev using commandline arguments like:

```
-device e1000,netdev=mynetdev -netdev tap,id=mynetdev
```

Where an arbitrary identifier (`mynetdev` in this case) is used to connect the two.

In our case we need a device, but not a netdev: we don't need to emulate the network, since we are passing to the VM a real networking device. The commandline argument that we need is:

```
-device vfio-pci,host=01:00.0
```

(Note the PCI triple of our device, without the PCI-domain part).

To actually run QEMU we also need a Linux installation (e.g., a file containing the ISO image of a live CD). The final command line may be:

```
qemu-system-x86_64 -enable-kvm -cdrom linux.iso -boot d \
    -device vfio-pci,host=01:00.0
```

If everything has worked correctly, we should be able to see the 82598 device inside the virtual machine (note that the PCI triple will be different).

3 IOMMU groups

Linux will prevent the passthrough of devices that can communicate with other host-devices without going through the IOMMU, since this would pose a security risk. To complete the passthrough of a device, we always have to detach *all* the devices that can talk to it in this way.

Note that sometimes Linux assumes, conservatively, that two devices can communicate in this way unless the vendor certifies that they don't. This affects us, since the Intel card typically contains *two* devices, and Linux will assume that they can talk to each other bypassing the IOMMU. We can find out how Linux has grouped together the devices by looking in the `/sys/kernel/iommu_groups` directory. This directory contains a subdirectory for each "group" of devices that Linux thinks must be detached together. In our case we may see something like:

```
$ ls /sys/kernel/iommu_groups/1
00000:01:00.0        0000:01:00.1
```

where we can see that our example device has been grouped with `0000:01:00.1` (which we may note is indeed another function of the same physical device). In order to successfully run QEMU with the passthrough device, we need to bind both of them to `vfio-pci`.

```
$ echo 0000:01:00.1 |
      sudo tee /sys/bus/pci/drivers/ixgbe/unbind
0000:01:00.1
```

Note that this device has the same Vendor ID and Device ID as the other one, so `vfio-pci` already knows that it can handle it and will automatically grab it when `ixgbe` releases it. Finally, note that we don't need to pass *both* `0000:01:00.0` and `0000:01:00.1` to the VM if we don't want to. The only important thing is that nobody else may use `0000:01:00.1` if we passthrough `0000:01:00.0`, and vice-versa.

4 SR-IOV

Our network device supports SR-IOV, so it can create several “virtual functions” that behave like independent network cards, even if they share the same hardware.

To create n virtual functions we just need to write n in a pseudo-file in the device directory. E.g., for $n = 4$,

```
$ echo 4 |
    sudo tee /sys/bus/pci/devices/0000:01:00.0/sriov_numvfs
4
```

If the command completes without errors, “`ifconfig -a`” should show 4 new network interfaces. Note that the virtual functions are similar, but not exactly the same as the origina device: their driver is `ixgbevf`, not `ixgbe`. The bus/device/function triple of these new devices is chosen by Linux and has no relationship with the actual physical location of the devices in the bus. You may need to pass `pci=assign-busses` to Linux at boot (see the above example for `intel_iommu=on`) in order for this to work.

Theoretically, we can now pass each virtual function to a different VM. If we are unlucky, however, Linux will group all of them in the same IOMMU group, essentially defeating the purpose of SR-IOV.