

Chapter 6: Virtual Display Devices and Protocols

In this chapter, we will discuss the way in which we access our virtual machines by using virtual graphic cards and protocols. There are almost 10 available virtual display adapters that we can use in our virtual machines, and there are multiple available protocols and applications that we can use to access our virtual machines. If we forget about SSH for a second and any kind of console-based access in general, there are various protocols available on the market that we can use to access the console of our virtual machine, such as VNC, SPICE, and noVNC.

In Microsoft-based environments, we tend to use a **remote desktop protocol (RDP)**. If we are talking about **Virtual Desktop Infrastructure (VDI)**, then there are even more protocols available – **PC over IP (PCoIP)**, VMware Blast, and so on. Some of these technologies offer additional functionality, such as greater color depth, encryption, audio and filesystem redirection, printer redirection, bandwidth management, and USB and other port redirection. These are key technologies for your remote desktop experience in today's cloud-based world.

All of this means that we must put a bit more time and effort into getting to know various display devices and protocols, as well as how to configure and use them. We don't want to end up in

situations in which we can't see the display of a virtual machine because we selected the wrong virtual display device, or in a situation where we try to open a console to see the content of a virtual machine and the console doesn't open.

In this chapter, we will cover the following topics:

- Using virtual machine display devices
- Discussing remote display protocols
- Using the VNC display protocol
- Using the SPICE display protocol
- Getting display portability with NoVNC
- Let's get started!

Using virtual machine display devices

To make the graphics work on virtual machines, QEMU needs to provide two components to its virtual machines: a virtual graphic adapter and a method or protocol to access the graphics from the client. Let's discuss these two concepts, starting with a virtual graphic adapter. The latest version of QEMU has eight different types of virtual/emulated graphics adapters. All of these have some similarities and differences, all of which can be in terms of features and/or resolutions supported or other, more technical details. So, let's describe them and see which use cases we are going to favor a specific virtual graphic card for:

- **tcx:** A SUN TCX virtual graphics card that can be used with old SUN OSes.
- **cirrus:** A virtual graphic card that's based on an old Cirrus Logic GD5446 VGA chip. It can be used with any guest OS after Windows 95.
- **std:** A standard VGA card that can be used with high-resolution modes for guest OSes after Windows XP.
- **vmware:** VMware's SVGA graphics adapter, which requires additional drivers in Linux guest OSes and VMware Tools installation for Windows OSes.
- **QXL:** The de facto standard paravirtual graphics card that we need to use when we use SPICE remote display protocol, which we will cover in detail a bit later in this chapter. There's an older version of this virtual graphics card called QXL VGA, which lacks some more advanced features, while offering lower overhead (it uses less memory).
- **Virtio:** A paravirtual 3D virtual graphics card that is based on the virgl project, which provides 3D acceleration for QEMU guest OSes. It has two different types (VGA and gpu). virtio-vga is commonly used for situations where we need multi-monitor support and OpenGL hardware acceleration. The virtio-gpu version doesn't have a built-in standard VGA compatibility mode.
- **cg3:** A virtual graphics card that we can use with older SPARC-based guest OSes.
- **none:** Disables the graphics card in the guest OS.

When configuring your virtual machine, you can select these options at startup or virtual machine creation. In CentOS 8, the default virtual graphics card that gets assigned to a newly created virtual machine is **QXL**, as shown in the following screenshot of the configuration for a new virtual machine:

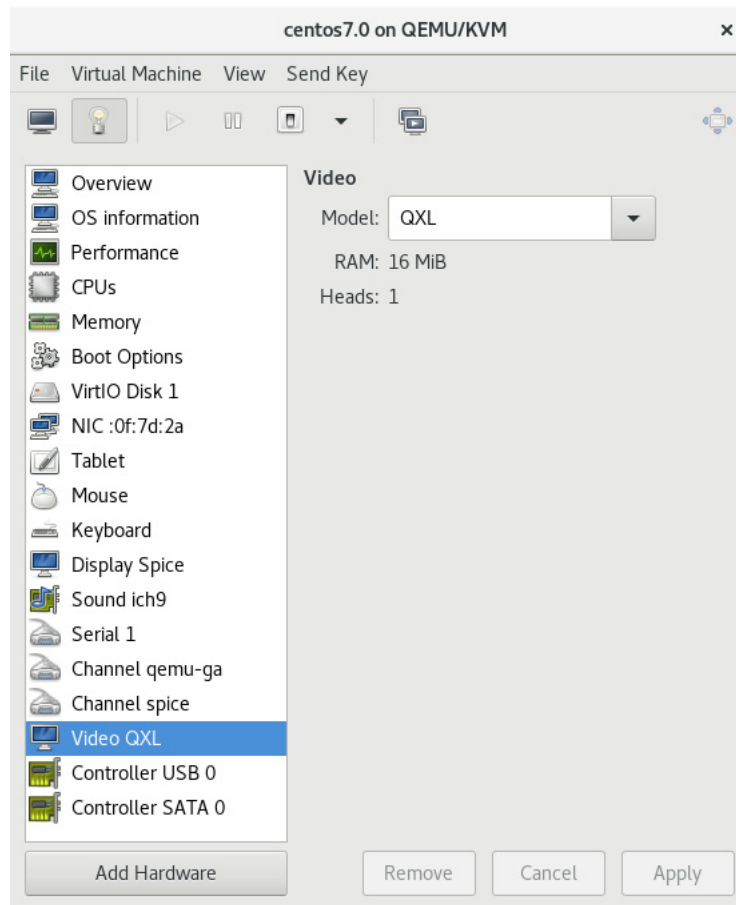


Figure 6.1 – Default virtual graphics card for a guest OS – QXL

Also, by default, we can select three of these types of virtual graphics cards for any given virtual machine, as these are usually pre-installed for us on any Linux server that's configured for virtualization:

- QXL
- VGA
- Virtio

Some of the new OSes running in KVM virtualization shouldn't use older graphics card adapters for a variety of reasons. For example, ever since Red Hat Enterprise Linux/CentOS 7, there's an advisory not to use the cirrus virtual graphics card for Windows 10 and Windows Server 2016. The reason for this is related to the instability of the virtual machine, as well as the fact that – for example – you can't use a full HD resolution display with the cirrus virtual graphics card. Just in case you start installing these guest OSes, make sure that you're using a QXL video graphics card as it offers the best performance and compatibility with the SPICE remote display protocol.

Theoretically, you could still use cirrus virtual graphics card for some of the *really* old guest OSes (older Windows NTs such as 4.0 and older client guest OSes such as Windows XP), but that's about it. For everything else, it's much better to either use a std or QXL driver as they offer the best performance and acceleration support. Furthermore, these virtual graphics cards also offer higher display resolutions.

There are some other virtual graphics cards available for QEMU, such as embedded drivers for various **System on a Chip (SoC)** devices, ati vga, bochs, and so on. Some of these are often used, such as SoCs – just remember all of the world's Raspberry Pis, and BBC Micro:bits. These new virtual graphics options are further expanded by **Internet of Things (IoT)**. So, there are loads of good reasons why we should pay

close attention to what's happening in this market space.

Let's show this via an example. Let's say that we want to create a new virtual machine that is going to have a set of custom parameters assigned to it in terms of how we access its virtual display. If you remember in [Chapter 3, Installing KVM Hypervisor, libvirt, and ovirt](#), we discussed various libvirt management commands (**virsh**, **virt-install**) and we also created some virtual machines by using **virt-install** and some custom parameters. Let's add to those and use a similar example:

```
virt-install --virt-type=kvm --name
MasteringKVM01 --vcpus 2 --ram 4096 -
-os-variant=rhel8.0 --/iso/CentOS-8-
x86_64-1905-dvd1.iso --
network=default --video=vga --
graphics vnc,password=Packt123 --disk
size=16
```

Here's what's going to happen:

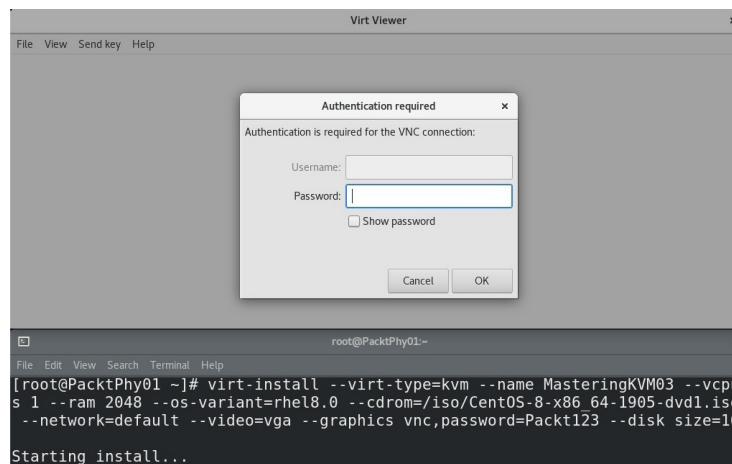


Figure 6.2 – KVM virtual machine with a VGA virtual graphics card is created. Here, VNC is asking for a password to be specified

After we type in the password (**Packt123**, as specified in the `virt-install` configuration option), we're faced with this screen:

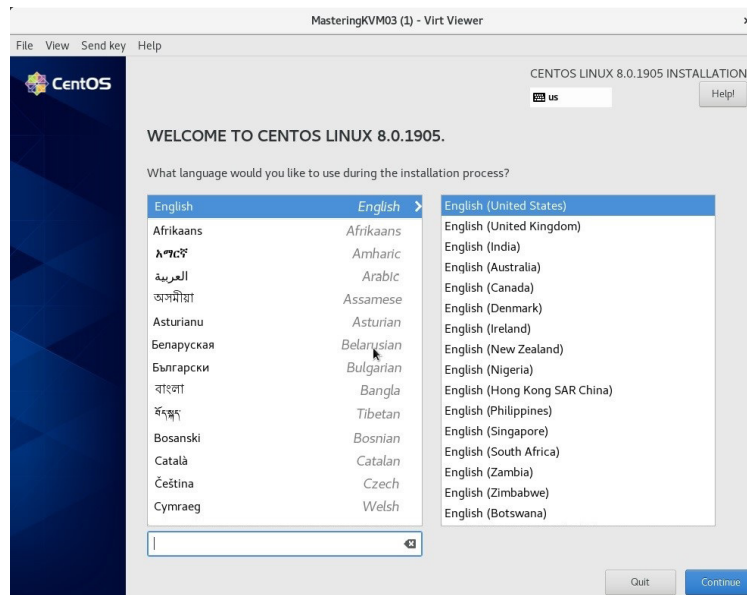


Figure 6.3 – VGA display adapter and its low default (640x480) initial resolution - a familiar resolution for all of us who grew up in the 80s

That being said, we just used this as an example of how to add an advanced option to the **virt-install** command – specifically, how to install a virtual machine with a specific virtual graphics card.

There are other, more advanced concepts of using real graphics cards that we installed in our computers or servers to forward their *capabilities* directly to virtual machines. This is *very* important for concepts such as VDI, as we mentioned earlier. Let's discuss these concepts for a second and use some real-world examples and comparisons to understand the complexity of VDI solutions on a larger scale.

Physical and virtual graphics cards in VDI scenarios

As we discussed in [*Chapter 1, Understanding Linux Virtualization*](#), VDI is a concept that uses virtualization paradigm for client OSes. This means that end users connect *directly* to their virtual machines by running a client OS (for example, Windows 8.1, Windows 10, or Linux Mint) that is either *reserved* for them or *pooled*, which means that multiple users can access the same virtual machines and get access to their *data* via additional VDI capabilities.

Now, if we're talking about most business users, they just need something that we jokingly call a *typewriter*. This usage model relates to a scenario in which the user uses a client OS for reading and writing documents, email, and browsing the internet. And for these use cases, if we were to use any vendor-based solution out there (VMware's Horizon, Citrix Xen Desktop, or Microsoft's Remote Desktop Services-based VDI solutions), we could do so with any one of them.

However, there's a big *but*. What happens if the scenario includes hundreds of users who need access to 2D and/or 3D video acceleration? What happens if we are designing a VDI solution for a company creating designs – architecture, plumbing, oil and gas, and video production? Running VDI solutions based on CPU and software-based virtual graphics cards will get us nowhere, especially at scale. This is where Xen Desktop and Horizon will be much more feature-packed if

we're talking about the technology level. And – to be quite honest – KVM-based methods aren't all that far behind in terms of display options, it's just that they lag in some other enterprise-class features, which we will discuss in later chapters, such as ***Chapter 12, Scaling Out KVM with OpenStack***.

Basically, there are three concepts we can use to obtain graphics card performance for a virtual machine:

- We can use a software renderer that's CPU-based.
- We can reserve a GPU for a specific virtual machine (PCI passthrough).
- We can *partition* a GPU so that we can use it in multiple virtual machines.

Just to use the VMware Horizon solution as a metaphor, these solutions would be called CPU rendering, **Virtual Direct Graphics Acceleration (vDGA)**, and **Virtual Shared Graphics Acceleration (vSGA)**. Or, in Citrix, we'd be talking about HDX 3D Pro. In CentOS 8, we are talking about *mediated devices* in the shared graphics card scenario.

If we're talking about PCI passthrough, it definitely delivers the best performance as you can use a PCI-Express graphics card, forward it directly to a virtual machine, install a native driver inside the guest OS, and have the complete graphics card all for yourself. But that creates four problems:

- You can only have that PCI-Express graphics card forwarded to *one* virtual machine.
- As servers can be limited in terms of upgradeability, for example, you can't run 50 virtual machines like that on one physical server as you can't fit 50 graphics cards on a single server – physically or in terms of PCI-Express slots, where you usually have up to six in a typical 2U rack server.
- If you're using Blade servers (for example, HP c7000), it's going to be even worse as you're going to use half of the server density per blade chassis if you're going to use additional graphics cards as these cards can only be fitted to double-height blades.
- You're going to spend an awful lot of money scaling any kind of solution like this to hundreds of virtual desktops, or – even worse – thousands of virtual desktops.

If we're talking about a shared approach in which you partition a physical graphics card so that you can use it in multiple virtual machines, that's going to create another set of problems:

- You're much more limited in terms of which graphics card to use as there are maybe 20 graphics cards that support this usage model (some include NVIDIA GRID, Quadro, Tesla cards, and a couple of AMD and Intel cards).
- If you share the same graphics card with four, eight, 16, or 32 virtual machines, you have to be aware of the fact that you'll get less performance, as you're sharing the same GPU with multiple virtual machines.

- Compatibility with DirectX, OpenGL, CUDA, and video encoding offload won't be as good as you might expect, and you might be forced to use older versions of these standards.
- There might be additional licensing involved, depending on the vendor and solution.

The next topic on our list is how to use a GPU in a more advanced way – by using the GPU partitioning concept to provide parts of a GPU to multiple virtual machines. Let's explain how that works and gets configured by using an NVIDIA GPU as an example.

GPU partitioning using an NVIDIA vGPU as an example

Let's use an example to see how we can use the scenario in which we partition our GPU (NVIDIA vGPU) with our KVM-based virtual machine. This procedure is very similar to the SR-IOV procedure we discussed in [*Chapter 4, Libvirt Networking*](#), where we used a supported Intel network card to present virtual functions to our CentOS host, which we then presented to our virtual machines by using them as uplinks for the KVM virtual bridge.

First, we need to check which kind of graphic cards we have, and it must be a supported one (in our case, we're using a Tesla P4). Let's use the **lshw** command to check our display devices, which should look similar to this:

```
# yum -y install lshw
# lshw -C display
*-display
```

```
description: 3D controller
product: GP104GL [Tesla P4]
vendor: NVIDIA Corporation
physical id: 0
bus info: pci@0000:01:00.0
version: a0
width: 64 bits
clock: 33MHz
capabilities: pm msi
pciexpress cap_list
configuration: driver=vfio-pci
latency=0
resources: irq:15
memory:f6000000-f6ffffff
memory:e0000000-efffffff
memory:f0000000-f1ffffff
```

The output of this command tells us that we have a 3D-capable GPU – specifically, a NVIDIA GP104GL-based product. It tells us that this device is already using the **vfio-pci** driver. This driver is the native SR-IOV driver for **Virtualized Functions (VF)**. These functions are the core of SR-IOV functionality. We will describe this by using this SR-IOV-capable GPU.

The first thing that we need to do – which all of us NVIDIA GPU users have been doing for years – is to blacklist the nouveau driver, which gets in the way. And if we are going to use GPU partitioning on a permanent basis, we need to do this permanently so that it doesn't get loaded when our server starts. But be warned – this can lead to unexpected behavior at times, such as the server booting and not showing any output without any real reason. So, we need to create a con-

figuration file for **modprobe** that will blacklist the nouveau driver. Let's create a file called **nouveauoff.conf** in the **/etc/modprobe.d** directory with the following content:

```
blacklist nouveau
options nouveau modeset 0
```

Then, we need to force our server to recreate the **initrd** image that gets loaded as our server starts and reboot the server to make that change is active. We are going to do that with the **dracut** command, followed by a regular **reboot** command:

```
# dracut --regenerate-all -force
# systemctl reboot
```

After the reboot, let's check if our **vfio** driver for the NVIDIA graphics card has loaded and, if it has, check the vGPU manager service:

```
# lsmod | grep nvidia | grep vfio
nvidia_vgpu_vfio 45011 0
nvidia 14248203 10 nvidia_vgpu_vfio
mdev 22078 2
vfio_mdev,nvidia_vgpu_vfio
vfio 34373 3
vfio_mdev,nvidia_vgpu_vfio,vfio_iommu
_type1
# systemctl status nvidia-vgpu-mgr
nvidia-vgpu-mgr.service - NVIDIA vGPU
Manager Daemon
    Loaded: loaded
    (/usr/lib/systemd/system/nvidia-vgpu-
mgr.service; enabled; vendor preset:
disabled)
```

```
Active: active (running) since Thu
2019-12-12 20:17:36 CET; 0h 3min ago
Main PID: 1327 (nvidia-vgpu-mgr)
```

We need to create a UUID that we will use to present our virtual function to a KVM virtual machine. We will use the **uuidgen** command for that:

```
uuidgen
c7802054-3b97-4e18-86a7-3d68dff2594d
```

Now, let's use this UUID for the virtual machines that will share our GPU. For that, we need to create an XML template file that we will add to the existing XML files for our virtual machines in a copy-paste fashion. Let's call this **vsga.xml**:

```
<hostdev mode='subsystem' type='mdev'
managed='no' model='vfio-pci'>
  <source>
    <address uuid='c7802054-3b97-
4e18-86a7-3d68dff2594d' />
  </source>
</hostdev>
```

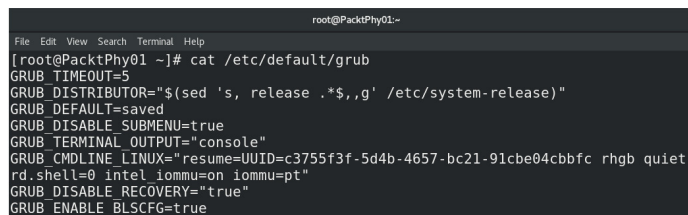
Use these settings as a template and just copy-paste the complete content to any virtual machine's XML file where you want to have access to our shared GPU.

The next concept that we need to discuss is the complete opposite of SR-IOV, where we're slicing a device into multiple pieces to present these pieces to virtual machines. In GPU passthrough, we're taking the *whole* device and presenting it directly to *one* object, meaning one virtual machine. Let's learn how to configure that.

GPU PCI passthrough

As with every advanced feature, enabling GPU PCI passthrough requires multiple steps to be done in sequence. By doing these steps in the correct order, we're directly presenting this hardware device to a virtual machine. Let's explain these configuration steps and do them:

1. To enable GPU PCI passthrough, we need to configure and enable IOMMU – first in our server's BIOS, then in our Linux distribution. We're using Intel-based servers, so we need to add `iommu` options to our `/etc/default/grub` file, as shown in the following screenshot:



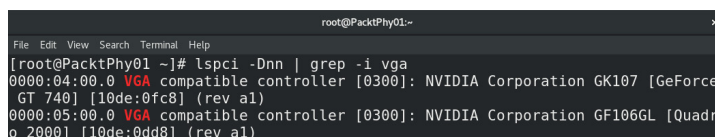
```
root@PacktPhy01:~  
File Edit View Search Terminal Help  
[root@PacktPhy01 ~]# cat /etc/default/grub  
GRUB_TIMEOUT=5  
GRUB_DISTRIBUTOR="$(sed 's, release .*$,g' /etc/system-release)"  
GRUB_DEFAULT=saved  
GRUB_DISABLE_SUBMENU=true  
GRUB_TERMINAL_OUTPUT="console"  
GRUB_CMDLINE_LINUX="resume=UUID=c3755f3f-5d4b-4657-bc21-91cbe04cbbfc rhgb quiet"  
rd.shell=0 intel_iommu=on iommu=pt  
GRUB_DISABLE_RECOVERY="true"  
GRUB_ENABLE_BLSCFG=true
```

Figure 6.4 – Adding `intel_iommu iommu=pt` options to a GRUB file

2. The next step is to reconfigure the GRUB configuration and reboot it, which can be achieved by typing in the following commands:

```
# grub2-mkconfig -o /etc/grub2.cfg  
# systemctl reboot
```

3. After rebooting the host, we need to acquire some information – specifically, ID information about the GPU device that we want to forward to our virtual machine. Let's do that:



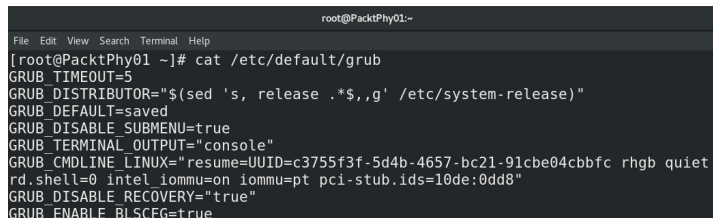
```
root@PacktPhy01:~  
File Edit View Search Terminal Help  
[root@PacktPhy01 ~]# lspci -Dnn | grep -i vga  
0000:04:00.0 VGA compatible controller [0300]: NVIDIA Corporation GK107 [GeForce GT 740] [10de:0fc8] (rev a1)  
0000:05:00.0 VGA compatible controller [0300]: NVIDIA Corporation GF106GL [Quadro 2000] [10de:0dd8] (rev a1)
```

Figure 6.5 – Using `lspci` to display relevant configuration information

In our use case, we want to forward the Quadro 2000 card to our virtual machine as we're using the GT740 to hook up our monitors and the Quadro card is currently free of any workloads or connections. So, we need to take note of two numbers; that is, **0000:05:00.0** and **10de:0dd8**.

We will need both IDs going forward, with each one for defining which device we want to use and where.

4. The next step is to explain to our host OS that it will not be using this PCI express device (Quadro card) for itself. In order to do that, we need to change the GRUB configuration again and add another parameter to the same file (**/etc/default/grub**):



```

root@PacktPhy01:~# cat /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="resume=UUID=c3755f3f-5d4b-4657-bc21-91cbe04cbbfc rhgb quiet
rd.shell=0 intel_iommu=on iommu=pt pci-stub.ids=10de:0dd8"
GRUB_DISABLE_RECOVERY="true"
GRUB_ENABLE_BLS_CFG=true

```

Figure 6.6 – Adding the `pci-stub.ids` option to GRUB so that it ignores this device when booting the OS

Again, we need to reconfigure GRUB and reboot the server after this, so type in the following commands:

```

# grub2-mkconfig -o /etc/grub2.cfg
# systemctl reboot

```

This step marks the end of the *physical* server configuration. Now, we can move on to the next stage of the process, which is how to use the now fully configured PCI passthrough device in our virtual machine.

5. Let's check if everything was done correctly by using the **virsh nodedev-dumpxml** command

on the PCI device ID:

```

root@PacktPhy01:~# virsh nodedev-dumpxml pci_0000_05_00_0
<device>
  <name>pci_0000_05_00_0</name>
  <path>/sys/devices/pci0000:00/0000:00:02.0/0000:05:00.0</path>
  <parent>pci_0000_00_02_0</parent>
  <driver>
    <name>pci-stub</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>5</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x0dd8'>GF106GL [Quadro 2000]</product>
    <vendor id='0x10de'>NVIDIA Corporation</vendor>
    <iommuGroup number='18'>
      <address domain='0x0000' bus='0x05' slot='0x00' function='0x1' />
      <address domain='0x0000' bus='0x05' slot='0x00' function='0x0' />
    </iommuGroup>
    <numa node='0' />
    <pci-express>
      <link validity='cap' port='0' speed='2.5' width='16' />
      <link validity='sta' speed='2.5' width='16' />
    </pci-express>
  </capability>
</device>

```

Figure 6.7 – Checking if the KVM stack can see our PCIe device

Here, we can see that QEMU sees two functions: **0x1** and **0x0**. The **0x1** function is actually the GPU device's *audio* chip, which we won't be using for our procedure. We just need the **0x0** function, which is the GPU itself. This means that we need to mask it. We can do that by using the following command:

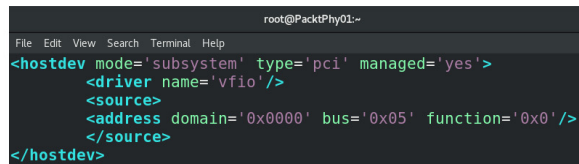
```

root@PacktPhy01:~# virsh nodedev-detach pci_0000_05_00_1
Device pci_0000_05_00_1 detached

```

Figure 6.8 – Detaching the 0x1 device so that it can't be used for passthrough

- Now, let's add the GPU via PCI passthrough to our virtual machine. For this purpose, we're using a freshly installed virtual machine called **MasteringKVM03**, but you can use any virtual machine you want. We need to create an XML file that QEMU will use to know which device to add to a virtual machine. After that, we need to shut down the machine and import that XML file into our virtual machine. In our case, the XML file will look like this:



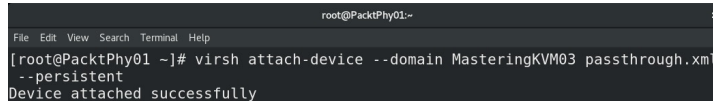
```

root@PacktPhy01:~#
File Edit View Search Terminal Help
<hostdev mode='subsystem' type='pci' managed='yes'>
  <driver name='vfio' />
  <source>
    <address domain='0x0000' bus='0x05' function='0x0' />
  </source>
</hostdev>

```

Figure 6.9 – The XML file with our GPU PCI passthrough definition for KVM

7. The next step is to attach this XML file to the **MasteringKVM03** virtual machine. We can do this by using the **virsh attach-device** command:



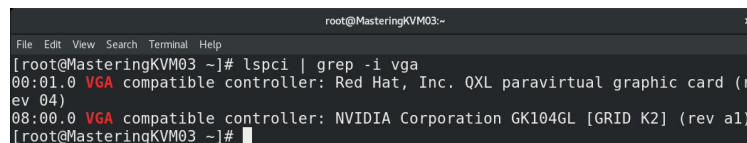
```

root@PacktPhy01:~#
File Edit View Search Terminal Help
[root@PacktPhy01 ~]# virsh attach-device --domain MasteringKVM03 passthrough.xml
--persistent
Device attached successfully

```

Figure 6.10 – Importing the XML file into a domain/virtual machine

8. After the previous step, we can start our virtual machine, log in, and check if the virtual machine sees our GPU:



```

root@MasteringKVM03:~#
File Edit View Search Terminal Help
[root@MasteringKVM03 ~]# lspci | grep -i vga
00:01.0 VGA compatible controller: Red Hat, Inc. QXL paravirtual graphic card (rev 04)
08:00.0 VGA compatible controller: NVIDIA Corporation GK104GL [GRID K2] (rev a1)
[root@MasteringKVM03 ~]#

```

Figure 6.11 – Checking GPU visibility in our virtual machine

The next logical step would be to install the NVIDIA driver for this card for Linux so that we can freely use it as our discrete GPU.

Now, let's move on to another important subject that is related to remote display protocols. We kind of danced around this subject in the previous part of this chapter as well, but now we are going to tackle it head-on.

Discussing remote display protocols

As we mentioned previously, there are different virtualization solutions, so it's only normal that there are different methods to *access* virtual machines. If you take a look at the history of virtual machines, we had a number of different display protocols taking care of this particular problem. So, let's discuss this history a bit.

Remote display protocols history

There will be people disputing this premise, but remote protocols started as text-only protocols. Whichever way you look at it, serial, text-mode terminals were here before we had X Windows or anything remotely resembling a GUI in the Microsoft, Apple, and UNIX-based worlds. Also, you can't dispute the fact that the telnet and rlogin protocols are also used to access remote display. It just so happens that the remote display that we're accessing by using telnet and rlogin is a text-based display. By extension, the same thing applies to SSH. And serial terminals, text consoles, and text-based protocols such as telnet and rlogin were some of the most commonly used starting points that go way back to the 1970s.

The end of the 1970s was an important time in computer history as there were numerous attempts to start mass-producing a personal computer for large amounts of people (for example, Apple II from 1977). In the 1980s, people started using personal computers more, as any Amiga, Commodore, Atari, Spectrum, or Amstrad fan will tell you. Keep in mind that the first real, publicly available GUI-based OSes didn't start appearing until Xerox Star (1981) and Apple Lisa

(1983). The first widely available Apple-based GUI OS was Mac OS System 1.0 in 1984. Most of the other previously mentioned computers were all using a text-based OS. Even games from that era (and for many years to come) looked like they were drawn by hand while you were playing them. Amiga's Workbench 1.0 was released in 1985 and with its GUI and color usage model, it was miles ahead of its time. However, 1985 is probably going to be remembered for something else – this is the year that the first Microsoft Windows OS (v1.0) was released. Later, that became Windows 2.0 (1987), Windows 3.0 (1990), Windows 3.1 (1992), by which time Microsoft was already taking the OS world by storm. Yes, there were other OSes by other manufacturers too:

- Apple: Mac OS System 7 (1991)
- IBM: OS/2 v1 (1988), v1.2 (1989), v2.0 (1992), Warp 4 (1996)

All of these were just a tiny dot on the horizon compared to the big storm that happened in 1995, when Microsoft introduced Windows 95. It was the first Microsoft client OS that was able to boot to GUI by default since the previous versions were started from a command line. Then came Windows 98 and XP, which meant even more market share for Microsoft. The rest of that story is probably very familiar, with Vista, Windows 7, Windows 8, and Windows 10.

The point of this story is not to teach you about OS history per se. It's about noticing the trend, which is simple enough. We started with text in-

interfaces in the command line (for example, IBM and MS DOS, early versions of Windows, Linux, UNIX, Amiga, Atari, and so on). Then, we slowly moved toward more visual interfaces (GUI). With advancements in networking, GPU, CPU, and monitoring technologies, we've reached a phase in which we want a shiny, 4K-resolution monitor with 4-megapixel resolutions, low latency, huge CPU power, fantastic colors, and a specific user experience. That user experience needs to be immediate, and it shouldn't really matter that we're using a local OS or a remote one (VDI, the cloud, or whatever the background technology is).

This means that along with all the hardware components that we just mentioned, other (software) components needed to be developed as well. Specifically, what needed to be developed were high-quality remote display protocols, which nowadays must be able to be extended to a browser-based usage model, as well. People don't want to be forced to install additional applications (clients) to access their remote resources.

Types of remote display protocols

Let's just mention some protocols that are very active on the market *now*:

- Microsoft Remote Desktop Protocol/Remote FX:
Used by Remote Desktop Connection, this multi-channel protocol allows us to connect to Microsoft-based virtual machines.
- VNC: Short for Virtual Network Computing, this is a remote desktop sharing system that

transmits mouse and keyboard events to access remote machines.

- **SPICE:** Short for Simple Protocol for Independent Computing Environments, this is another remote display protocol that can be used to access remote machines. It was developed by Qumranet, which was bought by Red Hat.

If we further expand our list to protocols that are being used for VDI, then the list increases further:

- **Teradici PCoIP (PC over IP):** A UDP-based VDI protocol that we can use to access virtual machines on VMware, Citrix and Microsoft-based VDI solutions
- **VMware Blast Extreme:** VMware's answer to PcoIP for VMware Horizon-based VDI solution
- **Citrix HDX:** Citrix's protocol for virtual desktops.

Of course, there are others that are available but not used as much and are way less important, such as the following:

- Colorado CodeCraft
- OpenText Exceed TurboX
- NoMachine
- FreeNX
- Apache Guacamole
- Chrome Remote Desktop
- Miranex

The major differences between regular remote protocols and fully featured VDI protocols are related to additional functionalities. For example,

on PCoIP, Blast Extreme, and HDX, you can fine-tune bandwidth settings, control USB and printer redirection (manually or centrally via policies), use multimedia redirection (to offload media decoding), Flash redirection (to offload Flash), client drive redirection, serial port redirection, and dozens of other features. You can't do some of these things on VNC or Remote Desktop, for example.

Having said that, let's discuss two of the most common ones in the open source world: VNC and SPICE.

Using the VNC display protocol

When the VNC graphics server is enabled through libvirt, QEMU will redirect the graphics output to its inbuilt VNC server implementation. The VNC server will listen to a network port where the VNC clients can connect.

The following screenshot shows how to add a VNC graphics server. Just go to **Virtual Machine Manager**, open the settings of your virtual machine, and go to the **Display Spice** tab on the left-hand side:

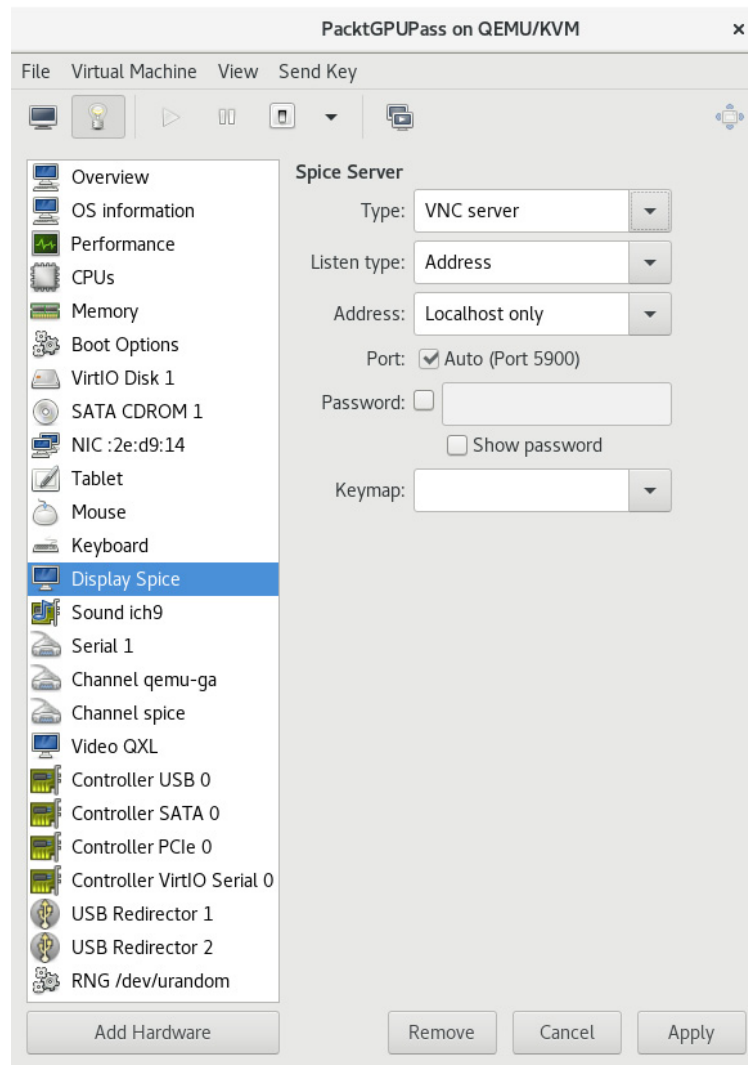


Figure 6.12 – VNC configuration for a KVM virtual machine

When adding VNC graphics, you will be presented with the options shown in the preceding screenshot:

- **Type:** The type of the graphics server. Here, it is **VNC server**.
- **Address:** VNC server listening address. It can be all, localhost, or an IP address. By default, it is **Localhost only**.
- **Port:** VNC server listening port. You can either choose auto, where libvirt defines the port based on the availability, or you can define one

yourself. Make sure it does not create a conflict.

- **Password:** The password protecting VNC access.
- **Keymap:** If you want to use a specific keyboard layout instead of an auto detected one, you can do the same using the **virt-xml** command-line tool.

For example, let's add VNC graphics to a virtual machine called **PacktGPUPass** and then modify its VNC listening IP to **192.168.122.1**:

```
# virt-xml MasteringKVM03 --add-  
device --graphics type=vnc  
# virt-xml MasteringKVM03 --edit --  
graphics listen=192.168.122.1
```

This is how it looks in the **PacktVM01** XML configuration file:

```
<graphics type='vnc' port='-1'  
autoport='yes'  
listen='192.168.122.1'>  
  <listen type='address'  
address='192.168.122.1' />  
</graphics>
```

You can also use **virsh** to edit **PacktGPUPass** and change the parameters individually.

Why VNC?

You can use VNC when you access virtual machines on LAN or to access the VMs directly from the console. It is not a good idea to expose virtual machines over a public network using VNC as the connection is not encrypted. VNC is a good

option if the virtual machines are servers with no GUI installed. Another point that is in favor of VNC is the availability of clients. You can access a virtual machine from any operating system platform as there will be a VNC viewer available for that platform.

Using the SPICE display protocol

Like KVM, a **Simple Protocol for Independent Computing Environments (SPICE)** is one of the best innovations that came into open source virtualization technologies. It propelled the open source virtualization to a large **Virtual Desktop Infrastructure (VDI)** implementation.

Important Note

Qumranet originally developed SPICE as a closed source code base in 2007. Red Hat, Inc. acquired Qumranet in 2008, and in December 2009, they decided to release the code under an open source license and treat the protocol as an open standard.

SPICE is the only open source solution available on Linux that gives two-way audio. It has high-quality 2D rendering capabilities that can make use of a client system's video card. SPICE also supports multiple HD monitors, encryption, smart card authentication, compression, and USB passthrough over the network. For a complete list of features, you can visit

<http://www.spice-space.org/features.html>. If you are a developer and want to know about the

internals of SPICE, visit <http://www.spice-space.org/documentation.html>. If you are planning for VDI or installing virtual machines that need GUIs, SPICE is the best option for you.

SPICE may not be compatible with some older virtual machines as they do not have support for QXL. In those cases, you can use SPICE along with other video generic virtual video cards.

Now, let's learn how to add a SPICE graphics server to our virtual machine. This can be considered the best-performing virtual display protocol in the open source world.

Adding a SPICE graphics server

Libvirt now selects SPICE as the default graphics server for most virtual machine installations. You must follow the same procedures that we mentioned earlier for VNC to add the SPICE graphics server. Just change the VNC to SPICE in the dropdown. Here, you will get an additional option to select a **TLS port** since SPICE supports encryption:

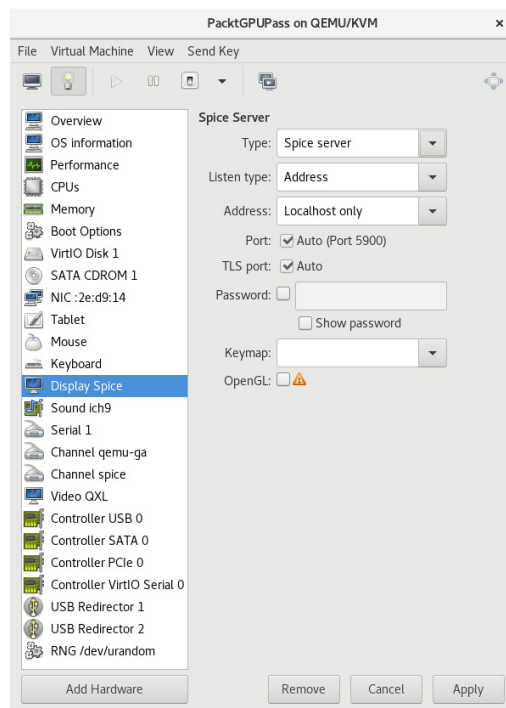


Figure 6.13 – SPICE configuration for a KVM virtual machine

To get to this configuration window, just edit the settings of your virtual machine. Go to the **Display Spice** options and select **Spice server** from the pull-down menu. All the other options are optional, so you don't necessarily have to do any additional configuration.

With this previous procedure completed, we've covered all the necessary topics regarding display protocols. Let's now discuss the various methods we can use to access the virtual machine console.

Methods to access a virtual machine console

There are multiple ways to connect to a virtual machine console. If your environment has full GUI access, then the easiest method is to use the

virt-manager console itself. **virt-viewer** is another tool that can give you access to your virtual machine console. This tool is very helpful if you are trying to access a virtual machine console from a remote location. In the following example, we are going to make a connection to a remote hypervisor that has an IP of **192.168.122.1**. The connection is tunneled through an SSH session and is secure.

The first step is to set up an authentication system without a password between your client system and the hypervisor:

1. On the client machine, use the following code:

```
# ssh-keygen
# ssh-copy-id root@192.168.122.1
# virt-viewer -c
qemu+ssh://root@192.168.122.1/system
m
```

You will be presented with a list of virtual machines available on the hypervisor. Select the one you have to access, as shown in the following screenshot:

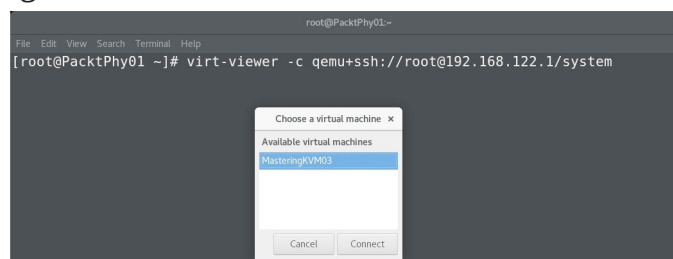


Figure 6.14 – virt-viewer selection menu for virtual machine access

2. To connect to a VM's console directly, use the following command:

```
# virt-viewer -c
qemu+ssh://root@192.168.122.1/system
m MasteringKVM03
```

If your environment is restricted to only a text console, then you must rely on your favorite **virsh** – to be more specific, **virsh console vm_name**. This needs some additional configuration inside the virtual machine OS, as described in the following steps.

3. If your Linux distro is using GRUB (not GRUB2), append the following line to your existing boot Kernel line in **/boot/grub/grub.conf** and shut down the virtual machine:

```
console=tty0 console=ttyS0,115200
```

If your Linux distro is using GRUB2, then the steps become a little complicated. Note that the following command has been tested on a Fedora 22 virtual machine. For other distros, the steps to configure GRUB2 might be different, though the changes that are required for GRUB configuration file should remain the same:

```
# cat /etc/default/grub (only  
relevant variables are shown)  
GRUB_TERMINAL_OUTPUT="console"  
GRUB_CMDLINE_LINUX="rd.lvm.lv=fedor  
a/swap rd.lvm.lv=fedora/root rhgb  
quiet"
```

The changed configuration is as follows:

```
# cat /etc/default/grub (only  
relevant variables are shown)  
GRUB_TERMINAL_OUTPUT="serial  
console"  
GRUB_CMDLINE_LINUX="rd.lvm.lv=fedor  
a/swap rd.lvm.lv=fedora/root  
console=tty0 console=ttyS0"
```

```
# grub2-mkconfig -o  
/boot/grub2/grub.cfg
```

4. Now, shut down the virtual machine. Then, start it again using **virsh**:

```
# virsh shutdown PacktGPUPass  
# virsh start PacktGPUPass --  
console
```

5. Run the following command to connect to a virtual machine console that has already started:

```
# virsh console PacktGPUPass
```

You can also do this from a remote client, as follows:

```
# virsh -c  
qemu+ssh://root@192.168.122.1/system console PacktGPUPass  
Connected to domain PacktGPUPass:  
Escape character is ^]
```

In some cases, we have seen a console command stuck at **^]**. To work around this, press the *Enter* key multiple times to see the login prompt. Sometimes, configuring a text console is very useful when you want to capture the boot messages for troubleshooting purposes. Use *ctrl +]* to exit from the console.

Our next topic takes us to the world of noVNC, another VNC-based protocol that has a couple of major advantages over the *regular* VNC. Let's discuss these advantages and the implementation of noVNC now.

Getting display portability with noVNC

All these display protocols rely on having access to some type of client application and/or additional software support that will enable us to access the virtual machine console. But what happens when we just don't have access to all of these additional capabilities? What happens if we only have text mode access to our environment, but we still want to have GUI-based management of connections to our virtual machines?

Enter noVNC, a HTML5-based VNC client that you can use via a HTML5-compatible web browser, which is just fancy talk for *practically every web browser on the market*. It supports all the most popular browsers, including mobile ones, and loads of other features, such as the following:

- Clipboard copy-paste
- Supports resolution scaling and resizing
- It's free under the MPL 2.0 license
- It's rather easy to install and supports authentication and can easily be implemented securely via HTTPS

If you want to make noVNC work, you need two things:

- Virtual machine(s) that are configured to accept VNC connections, preferably with a bit of configuration done – a password and a correctly set up network interface to connect to the virtual machine, for instance. You can freely use **tigervnc-server**, configure it to accept connections on – for example – port **5901**

for a specific user, and use that port and server's IP address for client connections.

- noVNC installation on a client computer, which you can either download from EPEL repositories or as a **zip/tar.gz** package and run directly from your web browser. To install it, we need to type in the following sequence of commands:

```
yum -y install novnc
cd /etc/pki/tls/certs
openssl req -x509 -nodes -newkey
rsa:2048 -keyout
/etc/pki/tls/certs/nv.pem -out
/etc/pki/tls/certs/nv.pem -days 365
websockify -D --
web=/usr/share/novnc --
cert=/etc/pki/tls/certs/nv.pem 6080
localhost:5901
```

The end result will look something like this:

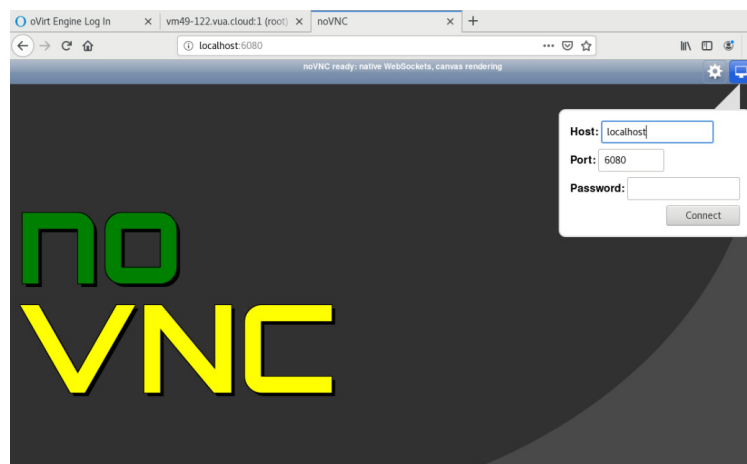


Figure 6.15 – noVNC console configuration screen

Here, we can use our VNC server password for that specific console. After typing in the password, we get this:

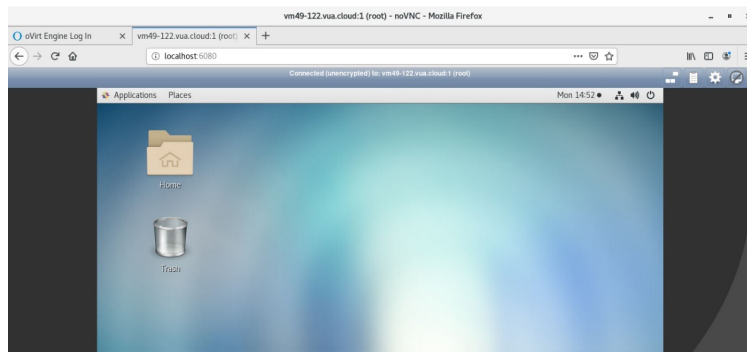


Figure 6.16 – noVNC console in action – we can see the virtual machine console and use it to work with our virtual machine

We can also use all these options in oVirt. During the installation of oVirt, we just need to select one additional option during the engine-setup phase:

```
--otopi-  
environment="OVESETUP_CONFIG/websocketProxyConfig=bool:True"
```

This option will enable oVirt to use noVNC as a remote display client, on top of the existing SPICE and VNC.

Let's take a look at an example of configuring a virtual machine in oVirt with pretty much all of the options that we've discussed in this chapter. Pay close attention to the **Monitors** configuration option:

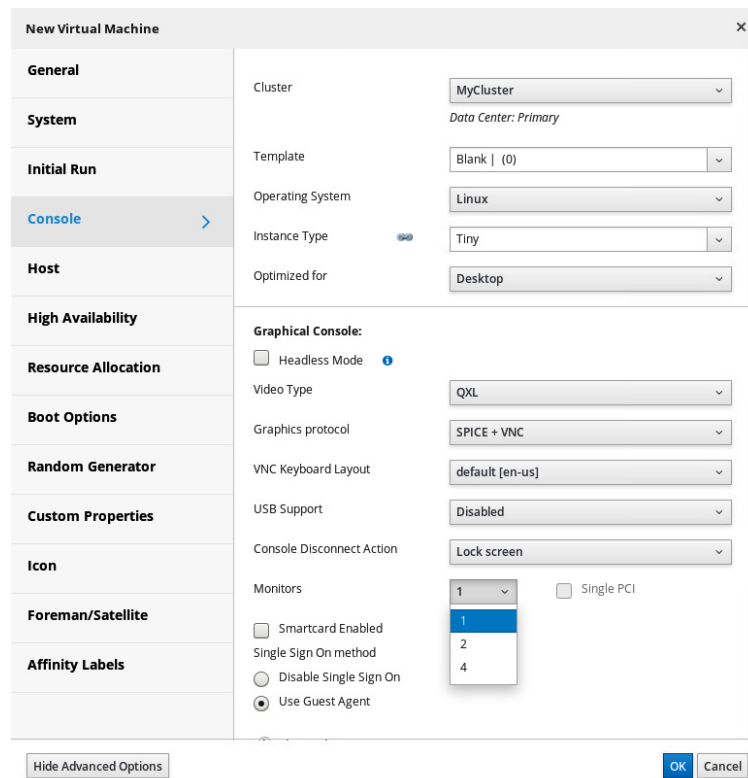


Figure 6.17 – oVirt also supports all the devices we discussed in this chapter

If we click on the **Graphics protocol** submenu, we will get the option to use SPICE, VNC, noVNC, and various combinations thereof. Also, at the bottom of the screen, we have available options for a number of monitors that we want to see in our remote display. This can be very useful if we want to have a high-performance multi-display remote console.

Seeing that noVNC has been integrated to noVNC as well, you can treat this as a sign of things to come. Think about it from this perspective – everything related to management applications in IT has steadily been moving to web-based applications for years now. It's only logical that the same things happen to virtual machine consoles. This has also been implemented in other ven-

dors' solutions, so seeing noVNC being used here shouldn't be a big surprise.

Summary

In this chapter, we covered virtual display devices and protocols used to display virtual machine data. We also did some digging into the world of GPU sharing and GPU passthrough, which are important concepts for large-scale virtualized environments running VDI. We discussed some benefits and drawbacks to these scenarios as they tend to be rather complex to implement and require a lot of resources – financial resources included. Imagine having to do PCI passthrough for 2D/3D acceleration for 100 virtual machines. That would actually require buying 100 graphic cards, which is a big, big ask financially. Among the other topics we discussed, we went through various display protocols and options that can be used for console access to our virtual machines.

In the next chapter, we will take you through some regular virtual machine operations – installation, configuration, and life cycle management, including discussing snapshots and virtual machine migration.

Questions

1. Which types of virtual machine display devices can we use?
2. What are the main benefits of using a QXL virtual display device versus VGA?

3. What are the benefits and drawbacks of GPU sharing?
4. What are the benefits of GPU PCI passthrough?
5. What are the main advantages of SPICE versus VNC?
6. Why would you use noVNC?

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- Configuring and managing virtualization:
[https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_virtualization](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/configuring_and_managing_virtualization/configuring_and_managing_virtualization)
- QEMU documentation:
<https://www.qemu.org/documentation/>
- NVIDIA virtual GPU software documentation:
<https://docs.nvidia.com/grid/latest/grid-vgpu-release-notes-red-hat-el-kvm/index.html>
- Working with IOMMU groups:
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_deployment_and_administration/virtualization_deployment_and_administration_iommu