Red Hat Enterprise Linux 7

Virtualization Deployment and Administration Guide

Installing, configuring, and managing virtual machines on a Red Hat Enterprise Linux physical machine

Last Updated: 2018-10-25

Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide

Installing, configuring, and managing virtual machines on a Red Hat Enterprise Linux physical machine

Jiri Herrmann
Red Hat Customer Content Services
jherrman@redhat.com

Yehuda Zimmerman
Red Hat Customer Content Services
yzimmerm@redhat.com

Laura Novich
Red Hat Customer Content Services

Dayle Parker
Red Hat Customer Content Services

Scott Radvan
Red Hat Customer Content Services

Tahlia Richardson
Red Hat Customer Content Services

Legal Notice

Copyright © 2018 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide covers how to configure a Red Hat Enterprise Linux 7 machine to act as a virtualization host system, and how to install and configure guest virtual machines using the KVM hypervisor. Other topics include PCI device configuration, SR-IOV, networking, storage, device and guest virtual machine management, as well as troubleshooting, compatibility and restrictions. Procedures that need to be run on the guest virtual machine are explicitly marked as such. All procedures described in this guide are intended to be performed on an AMD64 or Intel 64 host machine, unless otherwise stated. For using Red Hat Enterprise Linux 7 virtualization on architectures other than AMD64 and Intel 64, see . For a more general introduction into virtualization solutions provided by Red Hat, see the Red Hat Enterprise Linux 7 Virtualization Getting Started Guide.

Table of Contents

PART I. DEPLOYMENT **8**

CHAPTER 1. SYSTEM REQUIREMENTS **9**
1.1. HOST SYSTEM REQUIREMENTS 9
1.2. KVM HYPERVISOR REQUIREMENTS 10
1.3. KVM GUEST VIRTUAL MACHINE COMPATIBILITY 11
1.4. SUPPORTED GUEST CPU MODELS 11

CHAPTER 2. INSTALLING THE VIRTUALIZATION PACKAGES **13**
2.1. INSTALLING VIRTUALIZATION PACKAGES DURING A RED HAT ENTERPRISE LINUX INSTALLATION 13
2.2. INSTALLING VIRTUALIZATION PACKAGES ON AN EXISTING RED HAT ENTERPRISE LINUX SYSTEM 16

CHAPTER 3. CREATING A VIRTUAL MACHINE **19**
3.1. GUEST VIRTUAL MACHINE DEPLOYMENT CONSIDERATIONS 19
3.2. CREATING GUESTS WITH VIRT-INSTALL 19
3.3. CREATING GUESTS WITH VIRT-MANAGER 23
3.4. COMPARISON OF VIRT-INSTALL AND VIRT-MANAGER INSTALLATION OPTIONS 35

CHAPTER 4. CLONING VIRTUAL MACHINES **37**
4.1. PREPARING VIRTUAL MACHINES FOR CLONING 37
4.2. CLONING A VIRTUAL MACHINE 40

CHAPTER 5. KVM PARAVIRTUALIZED (VIRTIO) DRIVERS **44**
5.1. USING KVM VIRTIO DRIVERS FOR EXISTING STORAGE DEVICES 44
5.2. USING KVM VIRTIO DRIVERS FOR NEW STORAGE DEVICES 45
5.3. USING KVM VIRTIO DRIVERS FOR NETWORK INTERFACE DEVICES 49

CHAPTER 6. NETWORK CONFIGURATION **52**
6.1. NETWORK ADDRESS TRANSLATION (NAT) WITH LIBVIRT 52
6.2. DISABLING VHOST-NET 53
6.3. ENABLING VHOST-NET ZERO-COPY 54
6.4. BRIDGED NETWORKING 54

CHAPTER 7. OVERCOMMITTING WITH KVM **59**
7.1. INTRODUCTION 59
7.2. OVERCOMMITTING MEMORY 59
7.3. OVERCOMMITTING VIRTUALIZED CPUS 59

CHAPTER 8. KVM GUEST TIMING MANAGEMENT **61**
8.1. HOST-WIDE TIME SYNCHRONIZATION 62
8.2. REQUIRED TIME MANAGEMENT PARAMETERS FOR RED HAT ENTERPRISE LINUX GUESTS 63
8.3. STEAL TIME ACCOUNTING 64

CHAPTER 9. NETWORK BOOTING WITH LIBVIRT **65**
9.1. PREPARING THE BOOT SERVER 65
9.2. BOOTING A GUEST USING PXE 66

CHAPTER 10. REGISTERING THE HYPERVISOR AND VIRTUAL MACHINE **68**
10.1. INSTALLING VIRT-WHO ON THE HOST PHYSICAL MACHINE 68
10.2. REGISTERING A NEW GUEST VIRTUAL MACHINE 71
10.3. REMOVING A GUEST VIRTUAL MACHINE ENTRY 71
10.4. INSTALLING VIRT-WHO MANUALLY 72

10.5. TROUBLESHOOTING VIRT-WHO 73

CHAPTER 11. ENHANCING VIRTUALIZATION WITH THE QEMU GUEST AGENT AND SPICE AGENT ... 74
11.1. QEMU GUEST AGENT 74
11.2. USING THE QEMU GUEST AGENT WITH LIBVIRT 78
11.3. SPICE AGENT 80

CHAPTER 12. NESTED VIRTUALIZATION ... 83
12.1. OVERVIEW 83
12.2. SETUP 83
12.3. RESTRICTIONS AND LIMITATIONS 85

PART II. ADMINISTRATION ... 86

CHAPTER 13. MANAGING STORAGE FOR VIRTUAL MACHINES ... 87
13.1. STORAGE CONCEPTS 87
13.2. USING STORAGE POOLS 88
13.3. USING STORAGE VOLUMES 122

CHAPTER 14. USING QEMU-IMG ... 144
14.1. CHECKING THE DISK IMAGE 144
14.2. COMMITTING CHANGES TO AN IMAGE 144
14.3. COMPARING IMAGES 144
14.4. MAPPING AN IMAGE 145
14.5. AMENDING AN IMAGE 146
14.6. CONVERTING AN EXISTING IMAGE TO ANOTHER FORMAT 146
14.7. CREATING AND FORMATTING NEW IMAGES OR DEVICES 146
14.8. DISPLAYING IMAGE INFORMATION 147
14.9. REBASING A BACKING FILE OF AN IMAGE 147
14.10. RE-SIZING THE DISK IMAGE 148
14.11. LISTING, CREATING, APPLYING, AND DELETING A SNAPSHOT 148
14.12. SUPPORTED QEMU-IMG FORMATS 148

CHAPTER 15. KVM MIGRATION ... 150
15.1. MIGRATION DEFINITION AND BENEFITS 150
15.2. MIGRATION REQUIREMENTS AND LIMITATIONS 150
15.3. LIVE MIGRATION AND RED HAT ENTERPRISE LINUX VERSION COMPATIBILITY 152
15.4. SHARED STORAGE EXAMPLE: NFS FOR A SIMPLE MIGRATION 153
15.5. LIVE KVM MIGRATION WITH VIRSH 154
15.6. MIGRATING WITH VIRT-MANAGER 159

CHAPTER 16. GUEST VIRTUAL MACHINE DEVICE CONFIGURATION ... 164
16.1. PCI DEVICES 164
16.2. PCI DEVICE ASSIGNMENT WITH SR-IOV DEVICES 177
16.3. USB DEVICES 189
16.4. CONFIGURING DEVICE CONTROLLERS 190
16.5. SETTING ADDRESSES FOR DEVICES 194
16.6. RANDOM NUMBER GENERATOR DEVICE 196
16.7. ASSIGNING GPU DEVICES 198

CHAPTER 17. VIRTUAL NETWORKING ... 207
17.1. VIRTUAL NETWORK SWITCHES 207
17.2. BRIDGED MODE 207
17.3. NETWORK ADDRESS TRANSLATION 208
17.4. DNS AND DHCP 209

17.5. ROUTED MODE 210
17.6. ISOLATED MODE 210
17.7. THE DEFAULT CONFIGURATION 211
17.8. EXAMPLES OF COMMON SCENARIOS 212
17.9. MANAGING A VIRTUAL NETWORK 213
17.10. CREATING A VIRTUAL NETWORK 214
17.11. ATTACHING A VIRTUAL NETWORK TO A GUEST 224
17.12. ATTACHING A VIRTUAL NIC DIRECTLY TO A PHYSICAL INTERFACE 226
17.13. DYNAMICALLY CHANGING A HOST PHYSICAL MACHINE OR A NETWORK BRIDGE THAT IS ATTACHED TO A VIRTUAL NIC 230
17.14. APPLYING NETWORK FILTERING 231
17.15. CREATING TUNNELS 263
17.16. SETTING VLAN TAGS 264
17.17. APPLYING QOS TO YOUR VIRTUAL NETWORK 265

CHAPTER 18. REMOTE MANAGEMENT OF GUESTS 266
18.1. TRANSPORT MODES 266
18.2. REMOTE MANAGEMENT WITH SSH 269
18.3. REMOTE MANAGEMENT OVER TLS AND SSL 271
18.4. CONFIGURING A VNC SERVER 274
18.5. ENHANCING REMOTE MANAGEMENT OF VIRTUAL MACHINES WITH NSS 274

CHAPTER 19. MANAGING GUESTS WITH THE VIRTUAL MACHINE MANAGER (VIRT-MANAGER) 276
19.1. STARTING VIRT-MANAGER 276
19.2. THE VIRTUAL MACHINE MANAGER MAIN WINDOW 277
19.3. THE VIRTUAL HARDWARE DETAILS WINDOW 278
19.4. VIRTUAL MACHINE GRAPHICAL CONSOLE 283
19.5. ADDING A REMOTE CONNECTION 285
19.6. DISPLAYING GUEST DETAILS 286
19.7. MANAGING SNAPSHOTS 293

CHAPTER 20. MANAGING GUEST VIRTUAL MACHINES WITH VIRSH 297
20.1. GUEST VIRTUAL MACHINE STATES AND TYPES 297
20.2. DISPLAYING THE VIRSH VERSION 298
20.3. SENDING COMMANDS WITH ECHO 298
20.4. CONNECTING TO THE HYPERVISOR WITH VIRSH CONNECT 298
20.5. DISPLAYING INFORMATION ABOUT A GUEST VIRTUAL MACHINE AND THE HYPERVISOR 299
20.6. STARTING, RESUMING, AND RESTORING A VIRTUAL MACHINE 300
20.7. MANAGING A VIRTUAL MACHINE CONFIGURATION 302
20.8. SHUTTING OFF, SHUTTING DOWN, REBOOTING, AND FORCING A SHUTDOWN OF A GUEST VIRTUAL MACHINE 305
20.9. REMOVING AND DELETING A VIRTUAL MACHINE 307
20.10. CONNECTING THE SERIAL CONSOLE FOR THE GUEST VIRTUAL MACHINE 308
20.11. INJECTING NON-MASKABLE INTERRUPTS 308
20.12. RETRIEVING INFORMATION ABOUT YOUR VIRTUAL MACHINE 309
20.13. WORKING WITH SNAPSHOTS 314
20.14. DISPLAYING A URI FOR CONNECTION TO A GRAPHICAL DISPLAY 317
20.15. DISPLAYING THE IP ADDRESS AND PORT NUMBER FOR THE VNC DISPLAY 317
20.16. DISCARDING BLOCKS NOT IN USE 318
20.17. GUEST VIRTUAL MACHINE RETRIEVAL COMMANDS 318
20.18. CONVERTING QEMU ARGUMENTS TO DOMAIN XML 321
20.19. CREATING A DUMP FILE OF A GUEST VIRTUAL MACHINE'S CORE USING VIRSH DUMP 322
20.20. CREATING A VIRTUAL MACHINE XML DUMP (CONFIGURATION FILE) 323
20.21. CREATING A GUEST VIRTUAL MACHINE FROM A CONFIGURATION FILE 324

20.22. EDITING A GUEST VIRTUAL MACHINE'S XML CONFIGURATION SETTINGS 324
20.23. ADDING MULTIFUNCTION PCI DEVICES TO KVM GUEST VIRTUAL MACHINES 324
20.24. DISPLAYING CPU STATISTICS FOR A SPECIFIED GUEST VIRTUAL MACHINE 326
20.25. TAKING A SCREENSHOT OF THE GUEST CONSOLE 326
20.26. SENDING A KEYSTROKE COMBINATION TO A SPECIFIED GUEST VIRTUAL MACHINE 327
20.27. HOST MACHINE MANAGEMENT 328
20.28. RETRIEVING GUEST VIRTUAL MACHINE INFORMATION 336
20.29. STORAGE POOL COMMANDS 337
20.30. STORAGE VOLUME COMMANDS 345
20.31. DELETING STORAGE VOLUMES 347
20.32. DELETING A STORAGE VOLUME'S CONTENTS 347
20.33. DUMPING STORAGE VOLUME INFORMATION TO AN XML FILE 348
20.34. LISTING VOLUME INFORMATION 349
20.35. RETRIEVING STORAGE VOLUME INFORMATION 349
20.36. DISPLAYING PER-GUEST VIRTUAL MACHINE INFORMATION 350
20.37. MANAGING VIRTUAL NETWORKS 356
20.38. INTERFACE COMMANDS 361
20.39. MANAGING SNAPSHOTS 364
20.40. GUEST VIRTUAL MACHINE CPU MODEL CONFIGURATION 370
20.41. CONFIGURING THE GUEST VIRTUAL MACHINE CPU MODEL 374
20.42. MANAGING RESOURCES FOR GUEST VIRTUAL MACHINES 375
20.43. SETTING SCHEDULE PARAMETERS 376
20.44. DISK I/O THROTTLING 377
20.45. DISPLAY OR SET BLOCK I/O PARAMETERS 377
20.46. CONFIGURING MEMORY TUNING 377

CHAPTER 21. GUEST VIRTUAL MACHINE DISK ACCESS WITH OFFLINE TOOLS 379
21.1. INTRODUCTION 379
21.2. TERMINOLOGY 381
21.3. INSTALLATION 381
21.4. THE GUESTFISH SHELL 381
21.5. OTHER COMMANDS 386
21.6. VIRT-RESCUE: THE RESCUE SHELL 387
21.7. VIRT-DF: MONITORING DISK USAGE 388
21.8. VIRT-RESIZE: RESIZING GUEST VIRTUAL MACHINES OFFLINE 389
21.9. VIRT-INSPECTOR: INSPECTING GUEST VIRTUAL MACHINES 391
21.10. USING THE API FROM PROGRAMMING LANGUAGES 393
21.11. VIRT-SYSPREP: RESETTING VIRTUAL MACHINE SETTINGS 397
21.12. VIRT-CUSTOMIZE: CUSTOMIZING VIRTUAL MACHINE SETTINGS 400
21.13. VIRT-DIFF: LISTING THE DIFFERENCES BETWEEN VIRTUAL MACHINE FILES 404
21.14. VIRT-SPARSIFY: RECLAIMING EMPTY DISK SPACE 407

CHAPTER 22. GRAPHICAL USER INTERFACE TOOLS FOR GUEST VIRTUAL MACHINE MANAGEMENT 412
22.1. VIRT-VIEWER 412
22.2. REMOTE-VIEWER 414
22.3. GNOME BOXES 416

CHAPTER 23. MANIPULATING THE DOMAIN XML 421
23.1. GENERAL INFORMATION AND METADATA 421
23.2. OPERATING SYSTEM BOOTING 422
23.3. SMBIOS SYSTEM INFORMATION 425
23.4. CPU ALLOCATION 425
23.5. CPU TUNING 426
23.6. MEMORY BACKING 428

23.7. MEMORY TUNING 429
23.8. MEMORY ALLOCATION 430
23.9. NUMA NODE TUNING 431
23.10. BLOCK I/O TUNING 431
23.11. RESOURCE PARTITIONING 432
23.12. CPU MODELS AND TOPOLOGY 433
23.13. EVENTS CONFIGURATION 438
23.14. POWER MANAGEMENT 440
23.15. HYPERVISOR FEATURES 441
23.16. TIMEKEEPING 442
23.17. TIMER ELEMENT ATTRIBUTES 445
23.18. DEVICES 446
23.19. STORAGE POOLS 501
23.20. STORAGE VOLUMES 507
23.21. SECURITY LABEL 512
23.22. A SAMPLE CONFIGURATION FILE 514

PART III. APPENDICES 515

APPENDIX A. TROUBLESHOOTING 516
A.1. DEBUGGING AND TROUBLESHOOTING TOOLS 516
A.2. CREATING DUMP FILES 517
A.3. CAPTURING TRACE DATA ON A CONSTANT BASIS USING THE SYSTEMTAP FLIGHT RECORDER 519
A.4. KVM_STAT 520
A.5. TROUBLESHOOTING WITH SERIAL CONSOLES 524
A.6. VIRTUALIZATION LOGS 525
A.7. LOOP DEVICE ERRORS 525
A.8. LIVE MIGRATION ERRORS 526
A.9. ENABLING INTEL VT-X AND AMD-V VIRTUALIZATION HARDWARE EXTENSIONS IN BIOS 526
A.10. SHUTTING DOWN RED HAT ENTERPRISE LINUX 6 GUESTS ON A RED HAT ENTERPRISE LINUX 7 HOST 527
A.11. OPTIONAL WORKAROUND TO ALLOW FOR GRACEFUL SHUTDOWN 529
A.12. KVM NETWORKING PERFORMANCE 532
A.13. WORKAROUND FOR CREATING EXTERNAL SNAPSHOTS WITH LIBVIRT 533
A.14. MISSING CHARACTERS ON GUEST CONSOLE WITH JAPANESE KEYBOARD 534
A.15. GUEST VIRTUAL MACHINE FAILS TO SHUTDOWN 534
A.16. DISABLE SMART DISK MONITORING FOR GUEST VIRTUAL MACHINES 535
A.17. LIBGUESTFS TROUBLESHOOTING 535
A.18. TROUBLESHOOTING SR-IOV 536
A.19. COMMON LIBVIRT ERRORS AND TROUBLESHOOTING 536

APPENDIX B. USING KVM VIRTUALIZATION ON MULTIPLE ARCHITECTURES 564
B.1. USING KVM VIRTUALIZATION ON IBM POWER SYSTEMS 564
B.2. USING KVM VIRTUALIZATION ON IBM Z 566
B.3. USING KVM VIRTUALIZATION ON ARM SYSTEMS 569

APPENDIX C. VIRTUALIZATION RESTRICTIONS 570
C.1. SYSTEM RESTRICTIONS 570
C.2. FEATURE RESTRICTIONS 570
C.3. APPLICATION RESTRICTIONS 573
C.4. OTHER RESTRICTIONS 573
C.5. STORAGE SUPPORT 573
C.6. USB 3 / XHCI SUPPORT 574

APPENDIX D. ADDITIONAL RESOURCES **575**
D.1. ONLINE RESOURCES 575
D.2. INSTALLED DOCUMENTATION 575

APPENDIX E. WORKING WITH IOMMU GROUPS[1] **576**
E.1. IOMMU OVERVIEW 576
E.2. A DEEP-DIVE INTO IOMMU GROUPS 577
E.3. HOW TO IDENTIFY AND ASSIGN IOMMU GROUPS 578
E.4. IOMMU STRATEGIES AND USE CASES 580

APPENDIX F. REVISION HISTORY **582**

PART I. DEPLOYMENT

CHAPTER 1. SYSTEM REQUIREMENTS

Virtualization is available with the KVM hypervisor for Red Hat Enterprise Linux 7 on the Intel 64 and AMD64 architectures. This chapter lists system requirements for running virtual machines, also referred to as VMs.

For information on installing the virtualization packages, see Chapter 2, *Installing the Virtualization Packages*.

1.1. HOST SYSTEM REQUIREMENTS

Minimum host system requirements

- 6 GB free disk space.
- 2 GB RAM.

Recommended system requirements

- One core or thread for each virtualized CPU and one for the host.
- 2 GB of RAM, plus additional RAM for virtual machines.
- 6 GB disk space for the host, plus the required disk space for the virtual machine(s).

  Most guest operating systems require at least 6 GB of disk space. Additional storage space for each guest depends on their workload.

  Swap space

  Swap space in Linux is used when the amount of physical memory (RAM) is full. If the system needs more memory resources and the RAM is full, inactive pages in memory are moved to the swap space. While swap space can help machines with a small amount of RAM, it should not be considered a replacement for more RAM. Swap space is located on hard drives, which have a slower access time than physical memory. The size of your swap partition can be calculated from the physical RAM of the host. The Red Hat Customer Portal contains an article on safely and efficiently determining the size of the swap partition: https://access.redhat.com/site/solutions/15244.

  - When using raw image files, the total disk space required is equal to or greater than the sum of the space required by the image files, the 6 GB of space required by the host operating system, and the swap space for the guest.

    Equation 1.1. Calculating required space for guest virtual machines using raw images

    total for raw format = images + hostspace + swap

    For qcow images, you must also calculate the expected maximum storage requirements of the guest **`(total for qcow format)`**, as qcow and qcow2 images are able to grow as required. To allow for this expansion, first multiply the expected maximum storage requirements of the guest **`(expected maximum guest storage)`** by 1.01, and add to this the space required by the host **`(host)`**, and the necessary swap space **`(swap)`**.

Equation 1.2. Calculating required space for guest virtual machines using qcow images

total for qcow format = (expected maximum guest storage * 1.01) + host + swap

Guest virtual machine requirements are further outlined in Chapter 7, *Overcommitting with KVM*.

1.2. KVM HYPERVISOR REQUIREMENTS

The KVM hypervisor requires:

- an Intel processor with the Intel VT-x and Intel 64 virtualization extensions for x86-based systems; or
- an AMD processor with the AMD-V and the AMD64 virtualization extensions.

Virtualization extensions (Intel VT-x or AMD-V) are required for full virtualization. Enter the following commands to determine whether your system has the hardware virtualization extensions, and that they are enabled.

Procedure 1.1. Verifying virtualization extensions

1. **Verify the CPU virtualization extensions are available**
   enter the following command to verify the CPU virtualization extensions are available:

   ```
   $ grep -E 'svm|vmx' /proc/cpuinfo
   ```

2. **Analyze the output**

   - The following example output contains a **vmx** entry, indicating an Intel processor with the Intel VT-x extension:

     ```
     flags   : fpu tsc msr pae mce cx8 vmx apic mtrr mca cmov pat
     pse36 clflush
     dts acpi mmx fxsr sse sse2 ss ht tm syscall lm constant_tsc pni
     monitor ds_cpl
     vmx est tm2 cx16 xtpr lahf_lm
     ```

   - The following example output contains an **svm** entry, indicating an AMD processor with the AMD-V extensions:

     ```
     flags   :  fpu tsc msr pae mce cx8 apic mtrr mca cmov pat pse36
     clflush
     mmx fxsr sse sse2 ht syscall nx mmxext svm fxsr_opt lm 3dnowext
     3dnow pni cx16
     lahf_lm cmp_legacy svm cr8legacy ts fid vid ttp tm stc
     ```

   If the **grep -E 'svm|vmx' /proc/cpuinfo** command returns any output, the processor contains the hardware virtualization extensions. In some circumstances, manufacturers disable the virtualization extensions in the BIOS. If the extensions do not appear, or full virtualization does not work, see Procedure A.3, “Enabling virtualization extensions in BIOS” for instructions on enabling the extensions in your BIOS configuration utility.

3. **Ensure the KVM kernel modules are loaded**

As an additional check, verify that the **kvm** modules are loaded in the kernel with the following command:

```
# lsmod | grep kvm
```

If the output includes **kvm_intel** or **kvm_amd**, the **kvm** hardware virtualization modules are loaded.

NOTE

The **virsh** utility (provided by the libvirt-client package) can output a full list of your system's virtualization capabilities with the following command:

```
# virsh capabilities
```

1.3. KVM GUEST VIRTUAL MACHINE COMPATIBILITY

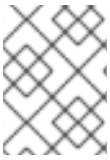
Red Hat Enterprise Linux 7 servers have certain support limits.

The following URLs explain the processor and memory amount limitations for Red Hat Enterprise Linux:

- For host systems: https://access.redhat.com/articles/rhel-limits
- For the KVM hypervisor: https://access.redhat.com/articles/rhel-kvm-limits

The following URL lists guest operating systems certified to run on a Red Hat Enterprise Linux KVM host:

- https://access.redhat.com/articles/973133



NOTE

For additional information on the KVM hypervisor's restrictions and support limits, see Appendix C, *Virtualization Restrictions*.

1.4. SUPPORTED GUEST CPU MODELS

Every hypervisor has its own policy for which CPU features the guest will see by default. The set of CPU features presented to the guest by the hypervisor depends on the CPU model chosen in the guest virtual machine configuration.

1.4.1. Listing the Guest CPU Models

To view a full list of the CPU models supported for an architecture type, run the **virsh cpu-models *architecture*** command. For example:

```
$ virsh cpu-models x86_64
486
pentium
pentium2
pentium3
pentiumpro
coreduo
```

```
n270
core2duo
qemu32
kvm32
cpu64-rhel5
cpu64-rhel6
kvm64
qemu64
Conroe
Penryn
Nehalem
Westmere
SandyBridge
Haswell
athlon
phenom
Opteron_G1
Opteron_G2
Opteron_G3
Opteron_G4
Opteron_G5
```

```
$ virsh cpu-models ppc64
POWER7
POWER7_v2.1
POWER7_v2.3
POWER7+_v2.1
POWER8_v1.0
```

The full list of supported CPU models and features is contained in the **cpu_map.xml** file, located in **/usr/share/libvirt/**:

```
# cat /usr/share/libvirt/cpu_map.xml
```

A guest's CPU model and features can be changed in the **<cpu>** section of the domain XML file. See Section 23.12, “CPU Models and Topology” for more information.

The host model can be configured to use a specified feature set as needed. For more information, see Section 23.12.1, “Changing the Feature Set for a Specified CPU”.

CHAPTER 2. INSTALLING THE VIRTUALIZATION PACKAGES

To use virtualization, Red Hat virtualization packages must be installed on your computer. Virtualization packages can be installed when installing Red Hat Enterprise Linux or after installation using the **yum** command and the Subscription Manager application.

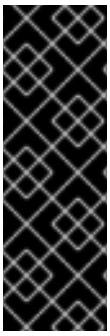
The KVM hypervisor uses the default Red Hat Enterprise Linux kernel with the **kvm** kernel module.

2.1. INSTALLING VIRTUALIZATION PACKAGES DURING A RED HAT ENTERPRISE LINUX INSTALLATION

This section provides information about installing virtualization packages while installing Red Hat Enterprise Linux.

NOTE

For detailed information about installing Red Hat Enterprise Linux, see the Red Hat Enterprise Linux 7 Installation Guide.



IMPORTANT

The Anaconda interface only offers the option to install Red Hat virtualization packages during the installation of Red Hat Enterprise Linux Server.

When installing a Red Hat Enterprise Linux Workstation, the Red Hat virtualization packages can only be installed after the workstation installation is complete. See Section 2.2, “Installing Virtualization Packages on an Existing Red Hat Enterprise Linux System”

Procedure 2.1. Installing virtualization packages

1. **Select software**
   Follow the installation procedure until the **Installation Summary** screen.

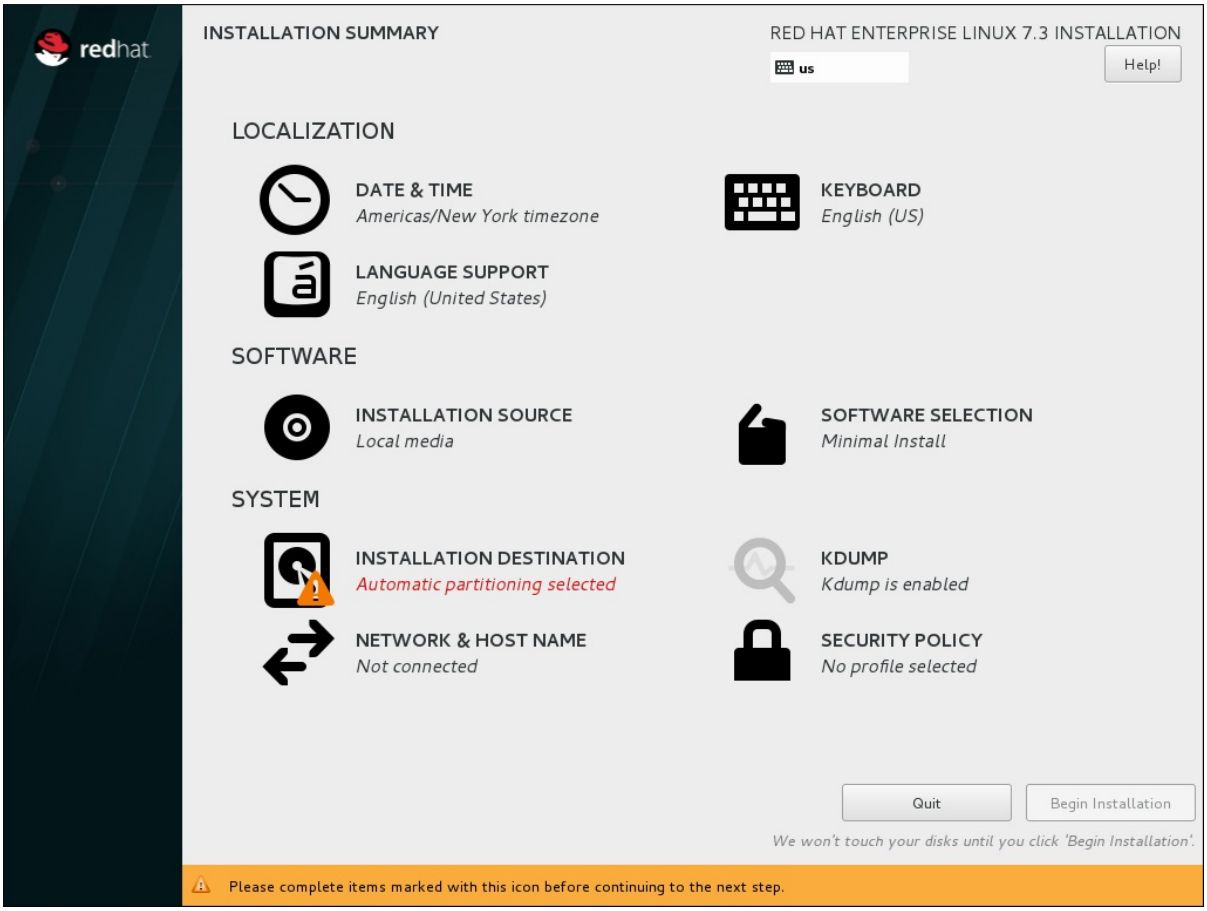


Figure 2.1. The Installation Summary screen

In the **Installation Summary** screen, click **Software Selection**. The **Software Selection** screen opens.

2. **Select the server type and package groups**
   You can install Red Hat Enterprise Linux 7 with only the basic virtualization packages or with packages that allow management of guests through a graphical user interface. Do one of the following:

   - **Install a minimal virtualization host**

     Select the **Virtualization Host** radio button in the **Base Environment** pane and the **Virtualization Platform** check box in the **Add-Ons for Selected Environment** pane. This installs a basic virtualization environment which can be run with **virsh** or remotely over the network.

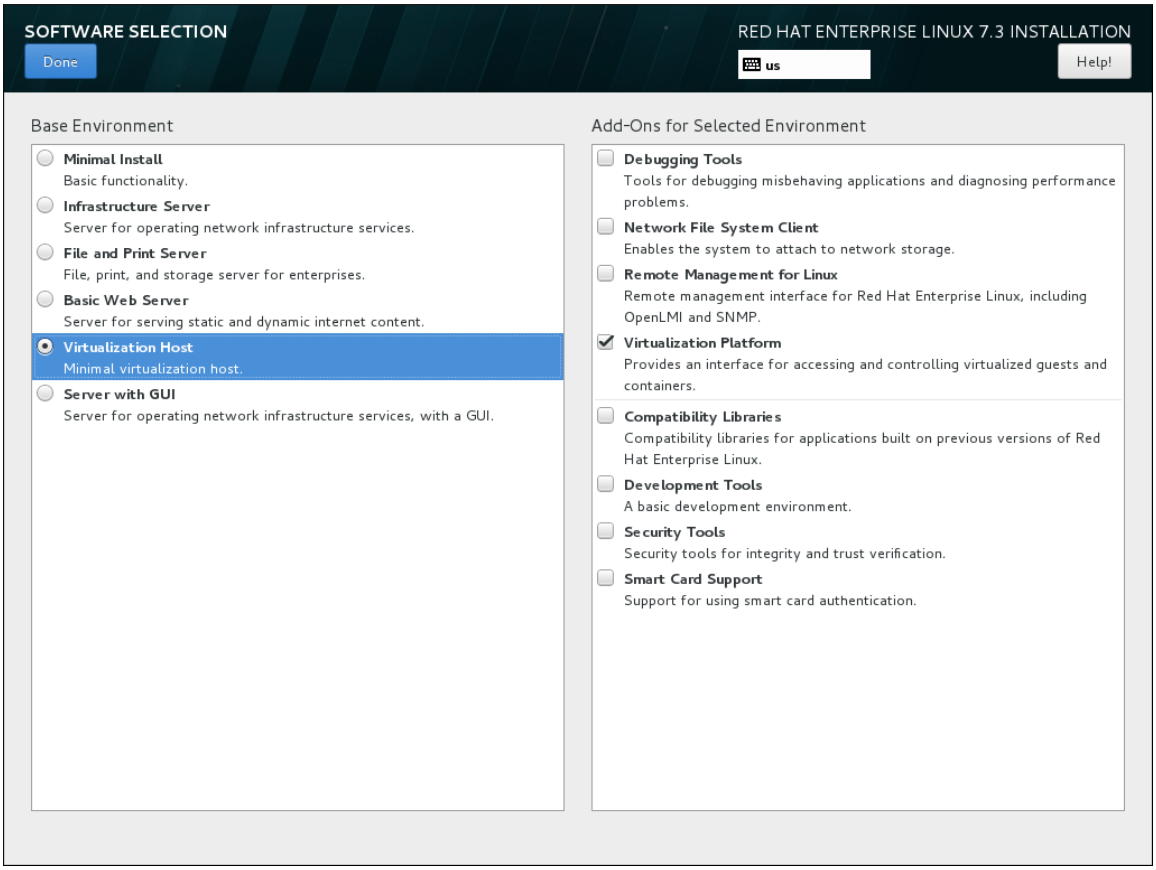


Figure 2.2. Virtualization Host selected in the Software Selection screen

- **`Install a virtualization host with a graphical user interface`**

  Select the **`Server with GUI`** radio button in the **`Base Environment`** pane and the **`Virtualization Client`**, **`Virtualization Hypervisor`**, and **`Virtualization Tools`** check boxes in the **`Add-Ons for Selected Environment`** pane. This installs a virtualization environment along with graphical tools for installing and managing guest virtual machines.

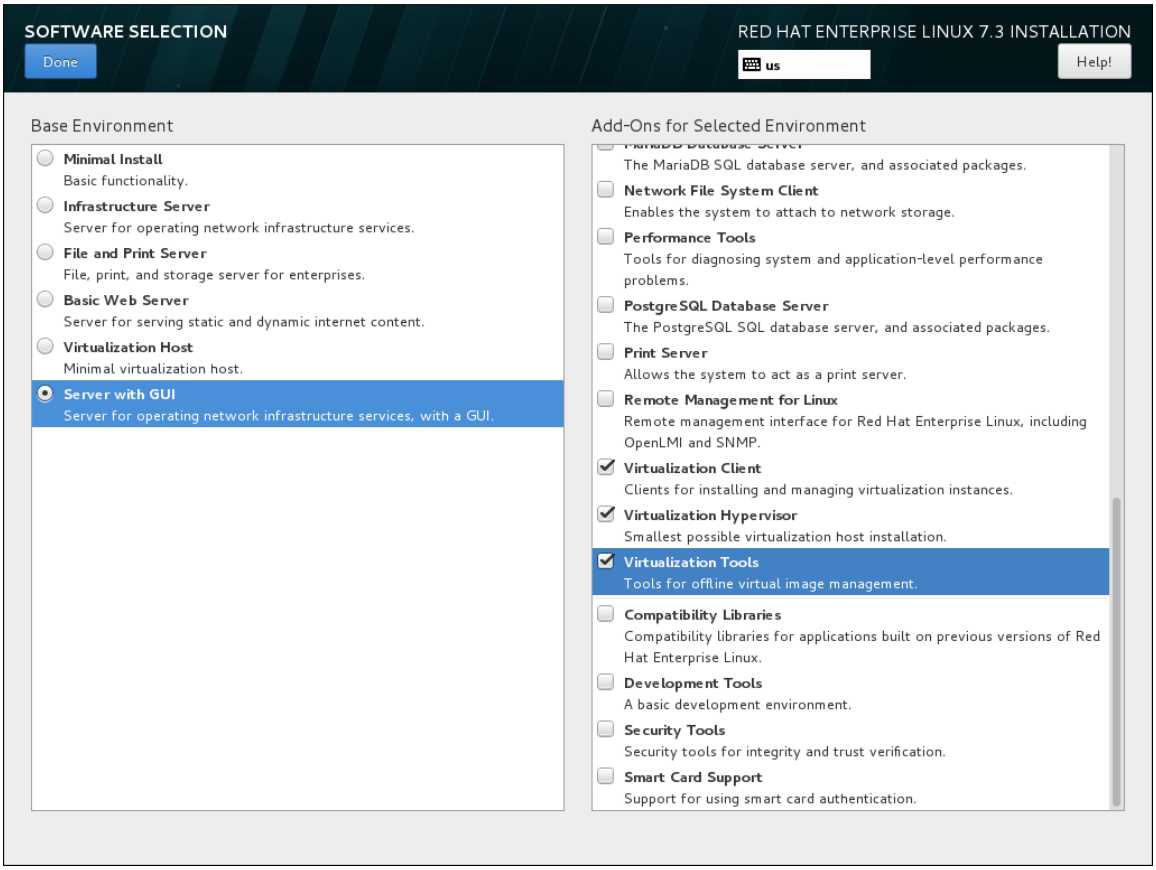
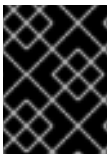


Figure 2.3. Server with GUI selected in the software selection screen

3. **Finalize installation**
   Click **Done** and continue with the installation.



IMPORTANT

You need a valid Red Hat Enterprise Linux subscription to receive updates for the virtualization packages.

2.1.1. Installing KVM Packages with Kickstart Files

To use a Kickstart file to install Red Hat Enterprise Linux with the virtualization packages, append the following package groups in the **%packages** section of your Kickstart file:

```
@virtualization-hypervisor
@virtualization-client
@virtualization-platform
@virtualization-tools
```

For more information about installing with Kickstart files, see the Red Hat Enterprise Linux 7 Installation Guide.

2.2. INSTALLING VIRTUALIZATION PACKAGES ON AN EXISTING RED HAT ENTERPRISE LINUX SYSTEM

This section describes the steps for installing the KVM hypervisor on an existing Red Hat Enterprise Linux 7 system.

To install the packages, your machine must be registered and subscribed to the Red Hat Customer Portal. To register using Red Hat Subscription Manager, run the **`subscription-manager register`** command and follow the prompts. Alternatively, run the Red Hat Subscription Manager application from **Applications** → **System Tools** on the desktop to register.

If you do not have a valid Red Hat subscription, visit the Red Hat online store to obtain one. For more information on registering and subscribing a system to the Red Hat Customer Portal, see https://access.redhat.com/solutions/253273.

2.2.1. Installing Virtualization Packages Manually

To use virtualization on Red Hat Enterprise Linux, at minimum, you need to install the following packages:

- qemu-kvm: This package provides the user-level KVM emulator and facilitates communication between hosts and guest virtual machines.
- qemu-img: This package provides disk management for guest virtual machines.

NOTE

The qemu-img package is installed as a dependency of the qemu-kvm package.

- libvirt: This package provides the server and host-side libraries for interacting with hypervisors and host systems, and the **`libvirtd`** daemon that handles the library calls, manages virtual machines, and controls the hypervisor.

To install these packages, enter the following command:

```
# yum install qemu-kvm libvirt
```

Several additional virtualization management packages are also available and are recommended when using virtualization:

- virt-install: This package provides the **`virt-install`** command for creating virtual machines from the command line.
- libvirt-python: This package contains a module that permits applications written in the Python programming language to use the interface supplied by the libvirt API.
- virt-manager: This package provides the **virt-manager** tool, also known as **Virtual Machine Manager**. This is a graphical tool for administering virtual machines. It uses the libvirt-client library as the management API.
- libvirt-client: This package provides the client-side APIs and libraries for accessing libvirt servers. The libvirt-client package includes the **`virsh`** command-line tool to manage and control virtual machines and hypervisors from the command line or a special virtualization shell.

You can install all of these recommended virtualization packages with the following command:

```
# yum install virt-install libvirt-python virt-manager virt-install
libvirt-client
```

2.2.2. Installing Virtualization Package Groups

The virtualization packages can also be installed from package groups. The following table describes the virtualization package groups and what they provide.

Table 2.1. Virtualization Package Groups

Package Group	Description	Mandatory Packages	Optional Packages
`Virtualization Hypervisor`	Smallest possible virtualization host installation	libvirt, qemu-kvm, qemu-img	qemu-kvm-tools
`Virtualization Client`	Clients for installing and managing virtualization instances	gnome-boxes, virt-install, virt-manager, virt-viewer, qemu-img	virt-top, libguestfs-tools, libguestfs-tools-c
`Virtualization Platform`	Provides an interface for accessing and controlling virtual machines and containers	libvirt, libvirt-client, virt-who, qemu-img	fence-virtd-libvirt, fence-virtd-multicast, fence-virtd-serial, libvirt-cim, libvirt-java, libvirt-snmp, perl-Sys-Virt
`Virtualization Tools`	Tools for offline virtual image management	libguestfs, qemu-img	libguestfs-java, libguestfs-tools, libguestfs-tools-c

To install a package group, run the **`yum groupinstall`** ***`package_group`*** command. Use the **`--optional`** option to install the optional packages in the package group. For example, to install the **`Virtualization Tools`** package group with all of its optional packages, run:

```
# yum groupinstall "Virtualization Tools" --optional
```

CHAPTER 3. CREATING A VIRTUAL MACHINE

After you have installed the virtualization packages on your Red Hat Enterprise Linux 7 host system, you can create virtual machines and install guest operating systems using the **virt-manager** interface. Alternatively, you can use the **`virt-install`** command-line utility by a list of parameters or with a script. Both methods are covered by this chapter.

3.1. GUEST VIRTUAL MACHINE DEPLOYMENT CONSIDERATIONS

Various factors should be considered before creating any guest virtual machines. The role of a virtual machine should be evaluated before deployment, but regular monitoring and assessment based on variable factors (load, amount of clients) should also be performed. The factors include:

Performance

Guest virtual machines should be deployed and configured based on their intended tasks. Some guest systems (for instance, guests running a database server) may require special performance considerations. Guests may require more assigned CPUs or memory based on their role and projected system load.

Input/Output requirements and types of Input/Output

Some guest virtual machines may have a particularly high I/O requirement or may require further considerations or projections based on the type of I/O (for instance, typical disk block size access, or the amount of clients).

Storage

Some guest virtual machines may require higher priority access to storage or faster disk types, or may require exclusive access to areas of storage. The amount of storage used by guests should also be regularly monitored and taken into account when deploying and maintaining storage. Make sure to read all the considerations outlined in Red Hat Enterprise Linux 7 Virtualization Security Guide. It is also important to understand that your physical storage may limit your options in your virtual storage.

Networking and network infrastructure

Depending upon your environment, some guest virtual machines could require faster network links than other guests. Bandwidth or latency are often factors when deploying and maintaining guests, especially as requirements or load changes.

Request requirements

SCSI requests can only be issued to guest virtual machines on virtio drives if the virtio drives are backed by whole disks, and the disk device parameter is set to **`lun`** in the domain XML file, as shown in the following example:

```
<devices>
  <emulator>/usr/libexec/qemu-kvm</emulator>
  <disk type='block' device='lun'>
```

3.2. CREATING GUESTS WITH VIRT-INSTALL

You can use the **`virt-install`** command to create virtual machines and install operating system on those virtual machines from the command line. **`virt-install`** can be used either interactively or as

part of a script to automate the creation of virtual machines. If you are using an interactive graphical installation, you must have **virt-viewer** installed before you run **virt-install**. In addition, you can start an unattended installation of virtual machine operating systems using **virt-install** with kickstart files.

NOTE

You might need root privileges in order for some **virt-install** commands to complete successfully.

The **virt-install** utility uses a number of command-line options. However, most **virt-install** options are not required.

The main required options for virtual guest machine installations are:

--name

The name of the virtual machine.

--memory

The amount of memory (RAM) to allocate to the guest, in MiB.

Guest storage

Use one of the following guest storage options:

- **--disk**

  The storage configuration details for the virtual machine. If you use the **--disk none** option, the virtual machine is created with no disk space.

- **--filesystem**

  The path to the file system for the virtual machine guest.

Installation method

Use one of the following installation methods:

- **--location**

  The location of the installation media.

- **--cdrom**

  The file or device used as a virtual CD-ROM device. It can be path to an ISO image, or to a CDROM device. It can also be a URL from which to fetch or access a minimal boot ISO image.

- **--pxe**

  Uses the PXE boot protocol to load the initial ramdisk and kernel for starting the guest installation process.

- **--import**

Skips the OS installation process and builds a guest around an existing disk image. The device used for booting is the first device specified by the **disk** or **filesystem** option.

- **--boot**

  The post-install VM boot configuration. This option allows specifying a boot device order, permanently booting off kernel and initrd with optional kernel arguments and enabling a BIOS boot menu.

To see a complete list of options, enter the following command:

```
# virt-install --help
```

To see a complete list of attributes for an option, enter the following command:

```
# virt install --option=?
```

The **virt-install** man page also documents each command option, important variables, and examples.

Prior to running **virt-install**, you may also need to use **qemu-img** to configure storage options. For instructions on using **qemu-img**, see Chapter 14, *Using qemu-img*.

3.2.1. Installing a virtual machine from an ISO image

The following example installs a virtual machine from an ISO image:

```
# virt-install \
  --name guest1-rhel7 \
  --memory 2048 \
  --vcpus 2 \
  --disk size=8 \
  --cdrom /path/to/rhel7.iso \
  --os-variant rhel7
```

The **--cdrom /path/to/rhel7.iso** option specifies that the virtual machine will be installed from the CD or DVD image at the specified location.

3.2.2. Importing a virtual machine image

The following example imports a virtual machine from a virtual disk image:

```
# virt-install \
  --name guest1-rhel7 \
  --memory 2048 \
  --vcpus 2 \
  --disk /path/to/imported/disk.qcow \
  --import \
  --os-variant rhel7
```

The **--import** option specifies that the virtual machine will be imported from the virtual disk image specified by the **--disk /path/to/imported/disk.qcow** option.

3.2.3. Installing a virtual machine from the network

The following example installs a virtual machine from a network location:

```
# virt-install \
  --name guest1-rhel7 \
  --memory 2048 \
  --vcpus 2 \
  --disk size=8 \
  --location http://example.com/path/to/os \
  --os-variant rhel7
```

The **`--location http://example.com/path/to/os`** option specifies that the installation tree is at the specified network location.

3.2.4. Installing a virtual machine using PXE

When installing a virtual machine using the PXE boot protocol, both the **`--network`** option specifying a bridged network and the **`--pxe`** option must be specified.

The following example installs a virtual machine using PXE:

```
# virt-install \
  --name guest1-rhel7 \
  --memory 2048 \
  --vcpus 2 \
  --disk size=8 \
  --network=bridge:br0 \
  --pxe \
  --os-variant rhel7
```

3.2.5. Installing a virtual machine with Kickstart

The following example installs a virtual machine using a kickstart file:

```
# virt-install \
  --name guest1-rhel7 \
  --memory 2048 \
  --vcpus 2 \
  --disk size=8 \
  --location http://example.com/path/to/os \
  --os-variant rhel7 \
  --initrd-inject /path/to/ks.cfg \
  --extra-args="ks=file:/ks.cfg console=tty0 console=ttyS0,115200n8"
```

The **`initrd-inject`** and the **`extra-args`** options specify that the virtual machine will be installed using a Kickstarter file.

3.2.6. Configuring the guest virtual machine network during guest creation

When creating a guest virtual machine, you can specify and configure the network for the virtual machine. This section provides the options for each of the guest virtual machine main network types.

Default network with NAT

The default network uses **libvirtd**'s network address translation (NAT) virtual network switch. For more information about NAT, see Section 6.1, “Network Address Translation (NAT) with libvirt”.

Before creating a guest virtual machine with the default network with NAT, ensure that the libvirt-daemon-config-network package is installed.

To configure a NAT network for the guest virtual machine, use the following option for **virt-install**:

```
--network default
```

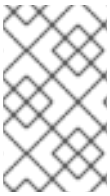
NOTE

If no **network** option is specified, the guest virtual machine is configured with a default network with NAT.

Bridged network with DHCP

When configured for bridged networking, the guest uses an external DHCP server. This option should be used if the host has a static networking configuration and the guest requires full inbound and outbound connectivity with the local area network (LAN). It should be used if live migration will be performed with the guest virtual machine. To configure a bridged network with DHCP for the guest virtual machine, use the following option:

```
--network br0
```



NOTE

The bridge must be created separately, prior to running **virt-install**. For details on creating a network bridge, see Section 6.4.1, “Configuring Bridged Networking on a Red Hat Enterprise Linux 7 Host”.

Bridged network with a static IP address

Bridged networking can also be used to configure the guest to use a static IP address. To configure a bridged network with a static IP address for the guest virtual machine, use the following options:

```
--network br0 \
--extra-args
"ip=192.168.1.2::192.168.1.1:255.255.255.0:test.example.com:eth0:none"
```

For more information on network booting options, see the Red Hat Enterprise Linux 7 Virtualization Guide.

No network

To configure a guest virtual machine with no network interface, use the following option:

```
--network=none
```

3.3. CREATING GUESTS WITH VIRT-MANAGER

The Virtual Machine Manager, also known as **virt-manager**, is a graphical tool for creating and managing guest virtual machines.

This section covers how to install a Red Hat Enterprise Linux 7 guest virtual machine on a Red Hat Enterprise Linux 7 host using **virt-manager**.

These procedures assume that the KVM hypervisor and all other required packages are installed and the host is configured for virtualization. For more information on installing the virtualization packages, see Chapter 2, *Installing the Virtualization Packages*.

3.3.1. virt-manager installation overview

The **New VM** wizard breaks down the virtual machine creation process into five steps:

1. Choosing the hypervisor and installation type
2. Locating and configuring the installation media
3. Configuring memory and CPU options
4. Configuring the virtual machine's storage
5. Configuring virtual machine name, networking, architecture, and other hardware settings

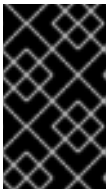
Ensure that **`virt-manager`** can access the installation media (whether locally or over the network) before you continue.

3.3.2. Creating a Red Hat Enterprise Linux 7 Guest with virt-manager

This procedure covers creating a Red Hat Enterprise Linux 7 guest virtual machine with a locally stored installation DVD or DVD image. Red Hat Enterprise Linux 7 DVD images are available from the Red Hat Customer Portal.

Procedure 3.1. Creating a Red Hat Enterprise Linux 7 guest virtual machine with virt-manager using local installation media

1. **Optional: Preparation**
   Prepare the storage environment for the virtual machine. For more information on preparing storage, see Chapter 13, *Managing Storage for Virtual Machines*.

   IMPORTANT

   Various storage types may be used for storing guest virtual machines. However, for a virtual machine to be able to use migration features, the virtual machine must be created on networked storage.

   Red Hat Enterprise Linux 7 requires at least 1 GB of storage space. However, Red Hat recommends at least 5 GB of storage space for a Red Hat Enterprise Linux 7 installation and for the procedures in this guide.

2. **Open virt-manager and start the wizard**
   Open virt-manager by executing the **`virt-manager`** command as root or opening **`Applications`** → **`System Tools`** → **`Virtual Machine Manager`**.

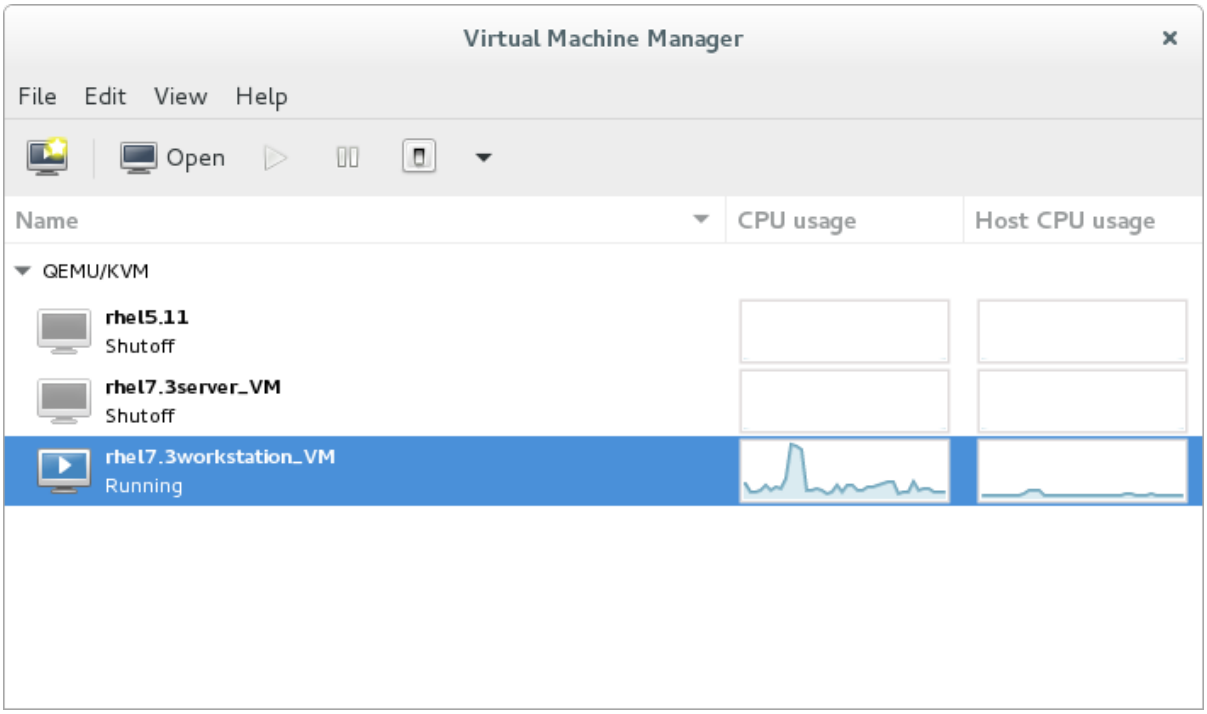


Figure 3.1. The Virtual Machine Manager window

Optionally, open a remote hypervisor by selecting the hypervisor and clicking the **Connect** button.

Click to start the new virtualized guest wizard.

The **New VM** window opens.

3. **Specify installation type**
Select the installation type:

Local install media (ISO image or CDROM)

This method uses a CD-ROM, DVD, or image of an installation disk (for example, **.iso**).

Network Install (HTTP, FTP, or NFS)

This method involves the use of a mirrored Red Hat Enterprise Linux or Fedora installation tree to install a guest. The installation tree must be accessible through either HTTP, FTP, or NFS.

If you select **Network Install**, provide the installation URL and also Kernel options, if required.

Network Boot (PXE)

This method uses a Preboot eXecution Environment (PXE) server to install the guest virtual machine. Setting up a PXE server is covered in the *Red Hat Enterprise Linux 7 Installation Guide*. To install using network boot, the guest must have a routable IP address or shared network device.

If you select **Network Boot**, continue to STEP 5. After all steps are completed, a DHCP request is sent and if a valid PXE server is found the guest virtual machine's installation processes will start.

Import existing disk image

This method allows you to create a new guest virtual machine and import a disk image (containing a pre-installed, bootable operating system) to it.

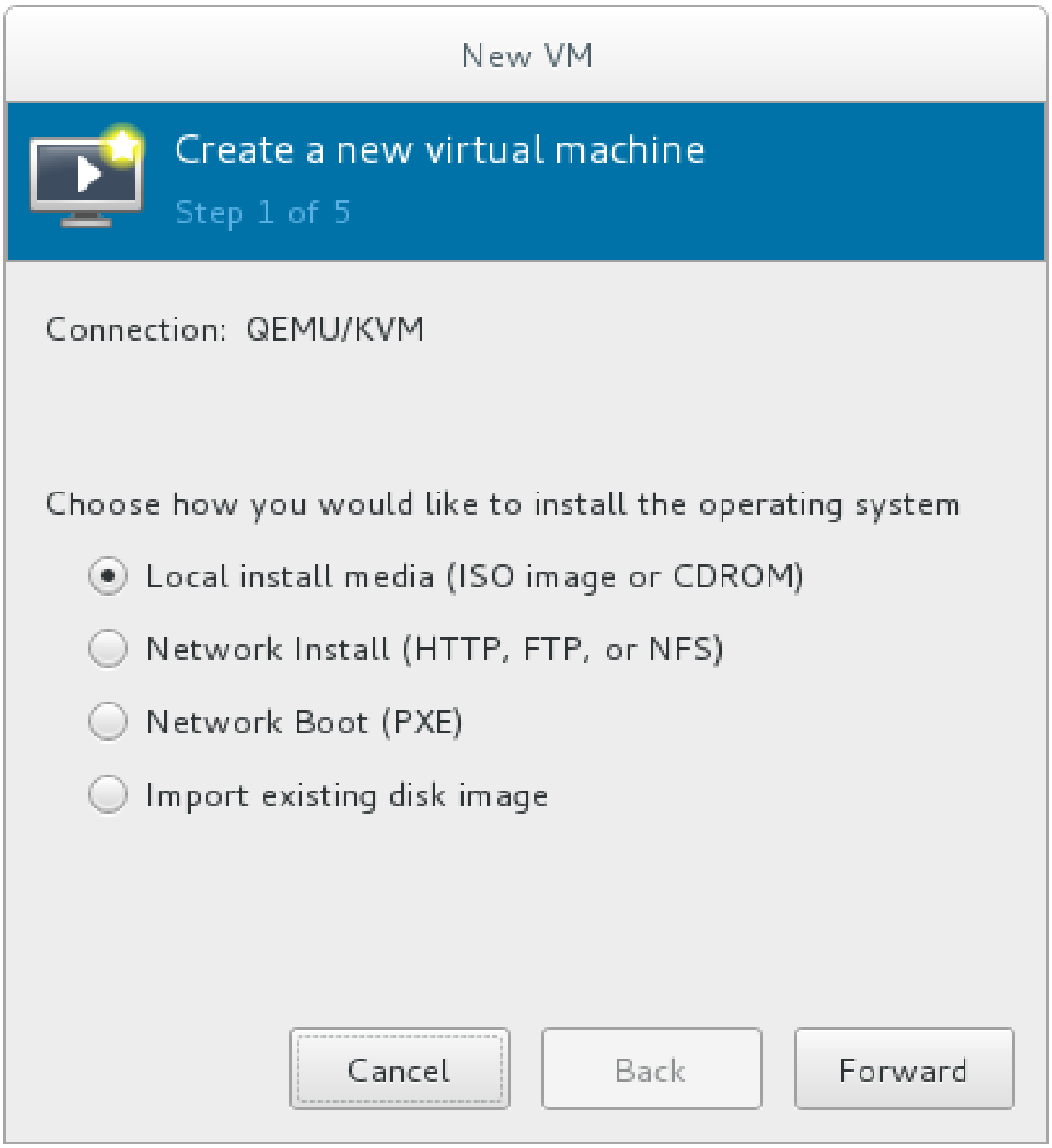


Figure 3.2. Virtual machine installation method

Click **Forward** to continue.

4. **Select the installation source**

   a. If you selected **Local install media (ISO image or CDROM)**, specify your intended local installation media.

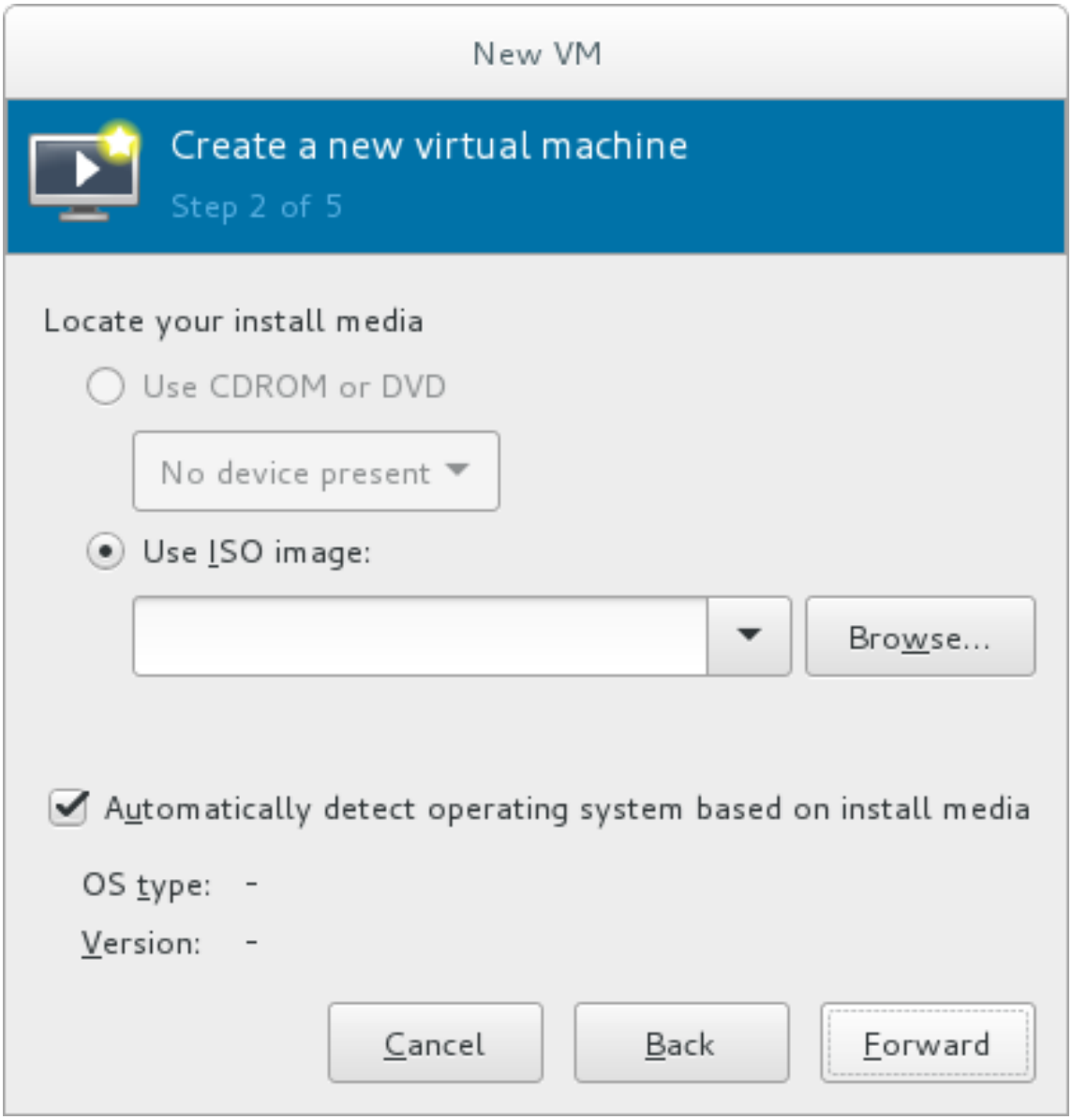


Figure 3.3. Local ISO image installation

- If you wish to install from a CD-ROM or DVD, select the **Use CDROM or DVD** radio button, and select the appropriate disk drive from the drop-down list of drives available.
- If you wish to install from an ISO image, select **Use ISO image**, and then click the **Browse...** button to open the **Locate media volume** window.

  Select the installation image you wish to use, and click **Choose Volume**.

  If no images are displayed in the **Locate media volume** window, click the **Browse Local** button to browse the host machine for the installation image or DVD drive containing the installation disk. Select the installation image or DVD drive containing the installation disk and click **Open**; the volume is selected for use and you are returned to the **Create a new virtual machine** wizard.

IMPORTANT

For ISO image files and guest storage images, the recommended location to use is **`/var/lib/libvirt/images/`**. Any other location may require additional configuration by SELinux. See the Red Hat Enterprise Linux Virtualization Security Guide or the Red Hat Enterprise Linux SELinux User's and Administrator's Guide for more details on configuring SELinux.

b. If you selected **`Network Install`**, input the URL of the installation source and also the required Kernel options, if any. The URL must point to the root directory of an installation tree, which must be accessible through either HTTP, FTP, or NFS.

To perform a kickstart installation, specify the URL of a kickstart file in Kernel options, starting with **`ks=`**

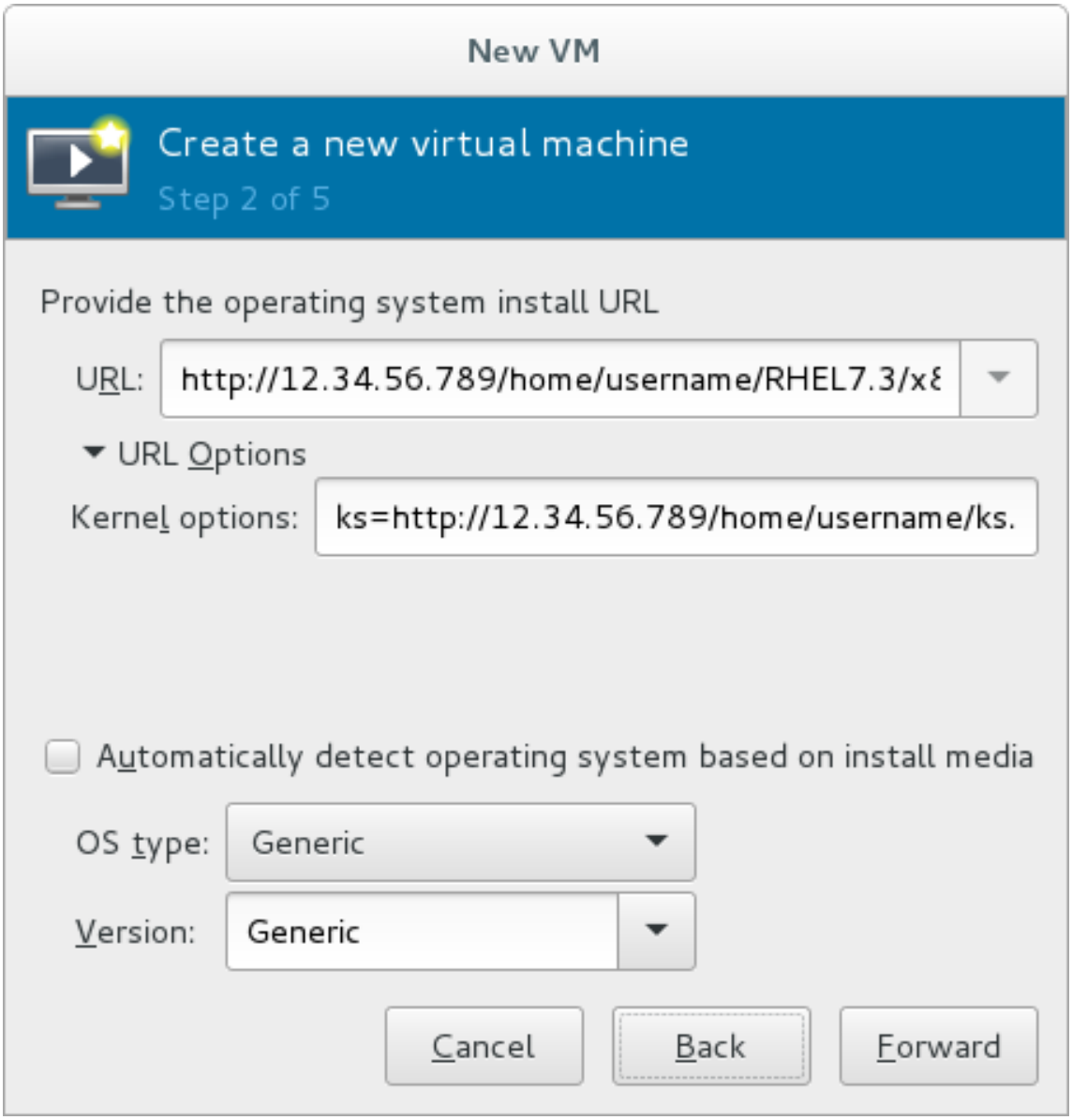
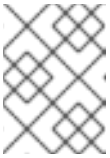


Figure 3.4. Network kickstart installation



NOTE

For a complete list of kernel options, see the Red Hat Enterprise Linux 7 Installation Guide.

Next, configure the **OS type** and **Version** of the installation. Ensure that you select the appropriate operating system type for your virtual machine. This can be specified manually or by selecting the **`Automatically detect operating system based on install media`** check box.

Click **`Forward`** to continue.

5. **Configure memory (RAM) and virtual CPUs**
   Specify the number of CPUs and amount of memory (RAM) to allocate to the virtual machine. The wizard shows the number of CPUs and amount of memory you can allocate; these values affect the host's and guest's performance.

   Virtual machines require sufficient physical memory (RAM) to run efficiently and effectively. Red Hat supports a minimum of 512MB of RAM for a virtual machine. Red Hat recommends at least 1024MB of RAM for each logical core.

   Assign sufficient virtual CPUs for the virtual machine. If the virtual machine runs a multi-threaded application, assign the number of virtual CPUs the guest virtual machine will require to run efficiently.

   You cannot assign more virtual CPUs than there are physical processors (or hyper-threads) available on the host system. The number of virtual CPUs available is noted in the **`Up to X available`** field.

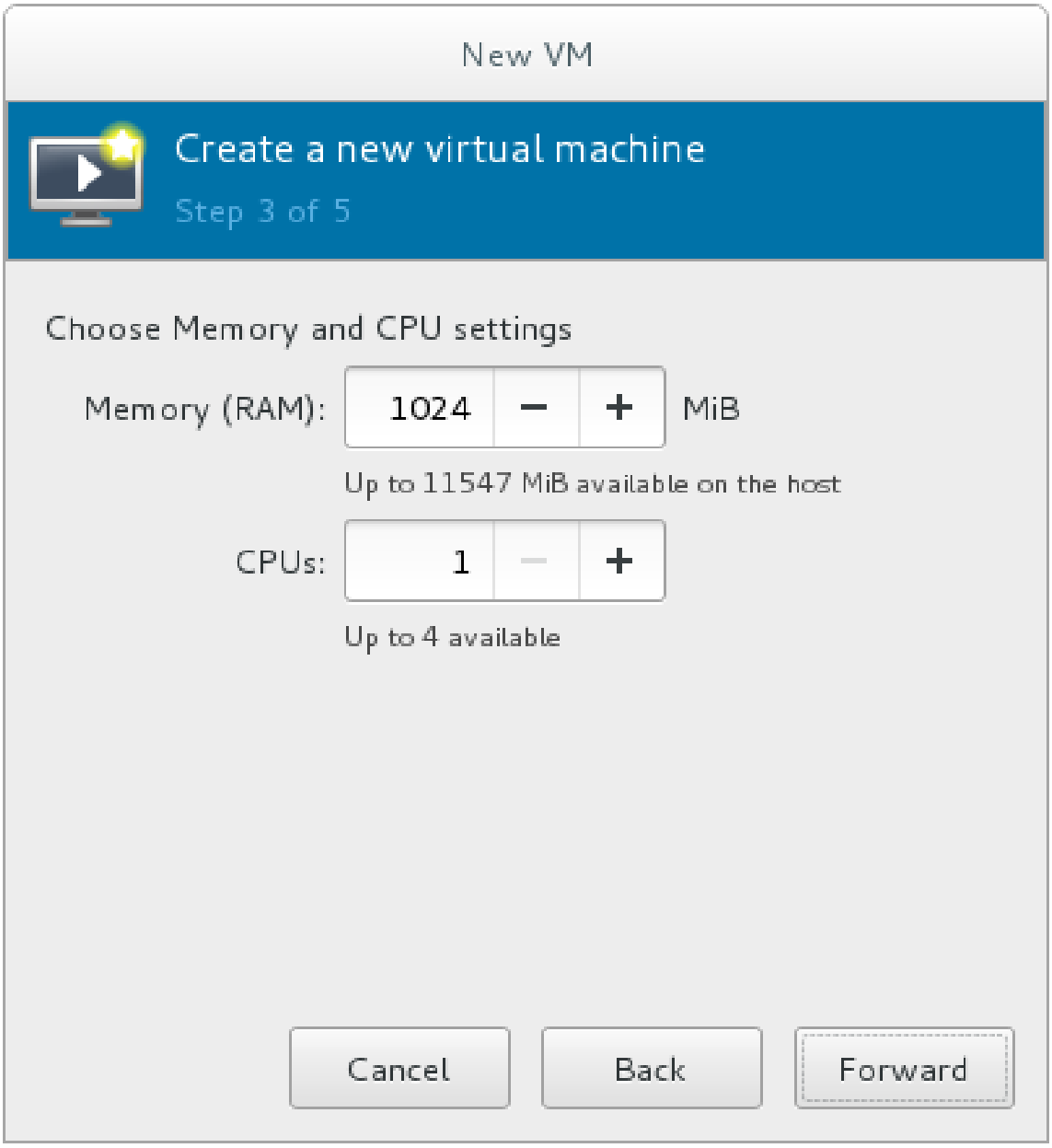


Figure 3.5. Configuring Memory and CPU

After you have configured the memory and CPU settings, click **Forward** to continue.

NOTE

Memory and virtual CPUs can be overcommitted. For more information on overcommitting, see Chapter 7, *Overcommitting with KVM*.

6. **Configure storage**
   Enable and assign sufficient space for your virtual machine and any applications it requires. Assign at least 5 GB for a desktop installation or at least 1 GB for a minimal installation.

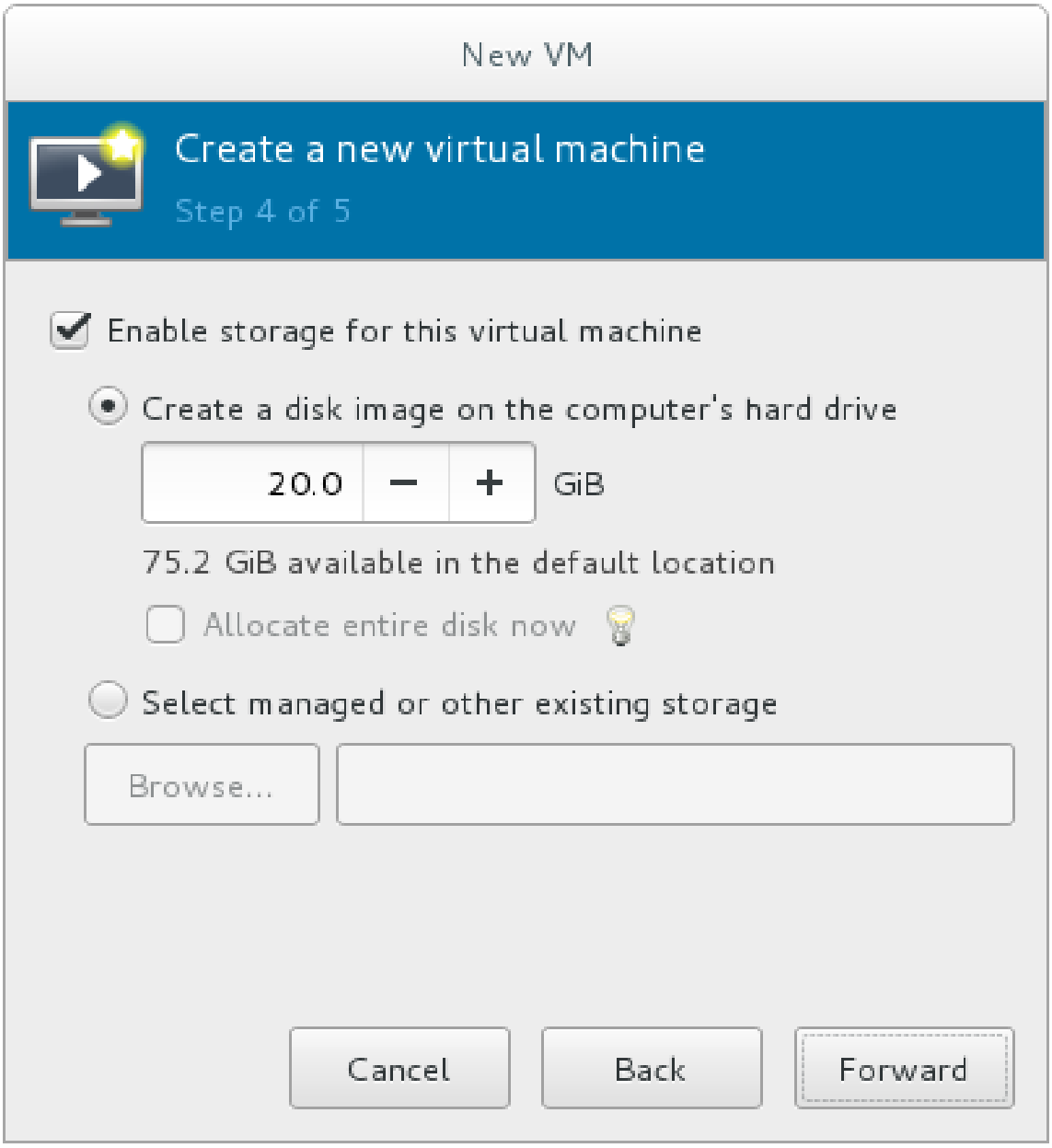
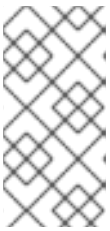


Figure 3.6. Configuring virtual storage



NOTE

Live and offline migrations require virtual machines to be installed on shared network storage. For information on setting up shared storage for virtual machines, see Section 15.4, “Shared Storage Example: NFS for a Simple Migration”.

a. **With the default local storage**
Select the **`Create a disk image on the computer's hard drive`** radio button to create a file-based image in the default storage pool, the **`/var/lib/libvirt/images/`** directory. Enter the size of the disk image to be created. If the **`Allocate entire disk now`** check box is selected, a disk image of the size specified will be created immediately. If not, the disk image will grow as it becomes filled.

NOTE

Although the storage pool is a virtual container it is limited by two factors: maximum size allowed to it by qemu-kvm and the size of the disk on the host physical machine. Storage pools may not exceed the size of the disk on the host physical machine. The maximum sizes are as follows:

- virtio-blk = 2^63 bytes or 8 Exabytes(using raw files or disk)
- Ext4 = ~ 16 TB (using 4 KB block size)
- XFS = ~8 Exabytes
- qcow2 and host file systems keep their own metadata and scalability should be evaluated/tuned when trying very large image sizes. Using raw disks means fewer layers that could affect scalability or max size.

Click **Forward** to create a disk image on the local hard drive. Alternatively, select **Select managed or other existing storage**, then select **Browse** to configure managed storage.

b. **With a storage pool**
If you select **Select managed or other existing storage** to use a storage pool, click **Browse** to open the **Locate or create storage volume** window.

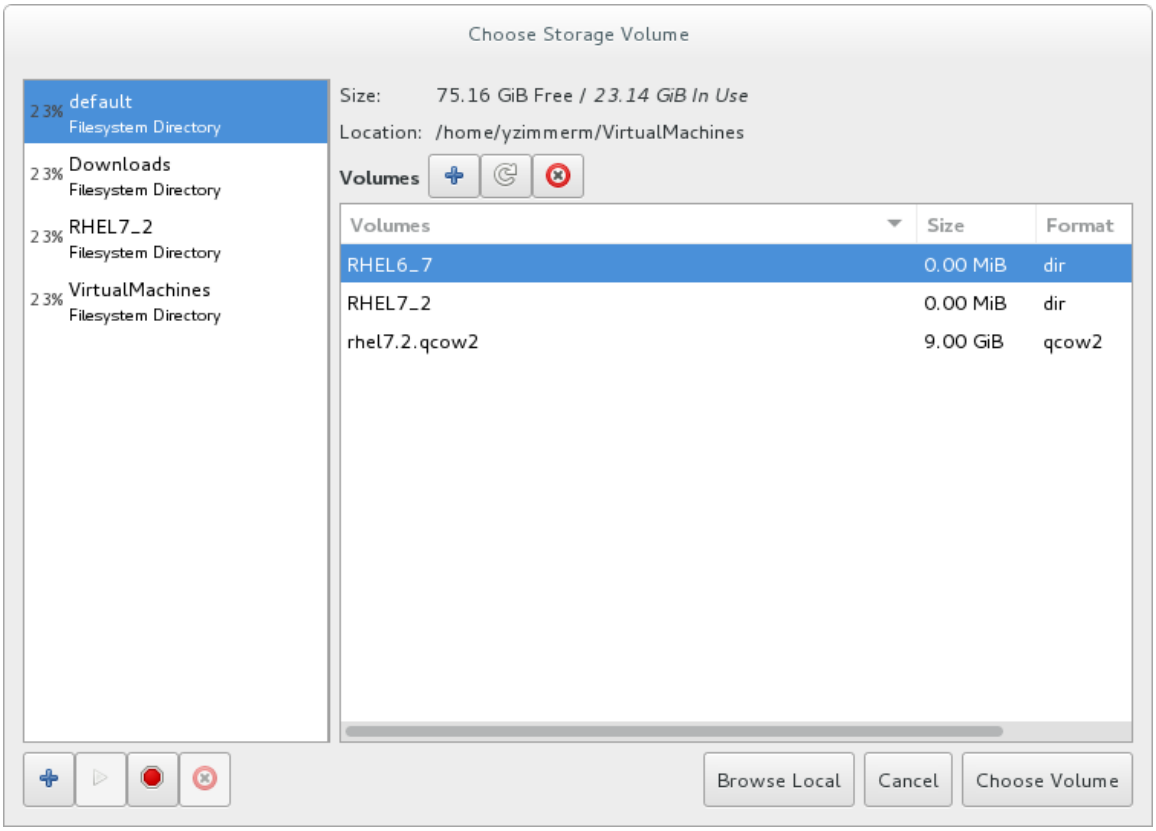


Figure 3.7. The Choose Storage Volume window

i. Select a storage pool from the **Storage Pools** list.

ii. Optional: Click to create a new storage volume. The **Add a Storage Volume** screen will appear. Enter the name of the new storage volume.

Choose a format option from the **Format** drop-down menu. Format options include raw, qcow2, and qed. Adjust other fields as needed. Note that the qcow2 version used here is version 3. To change the qcow version see Section 23.20.2, “Setting Target Elements”

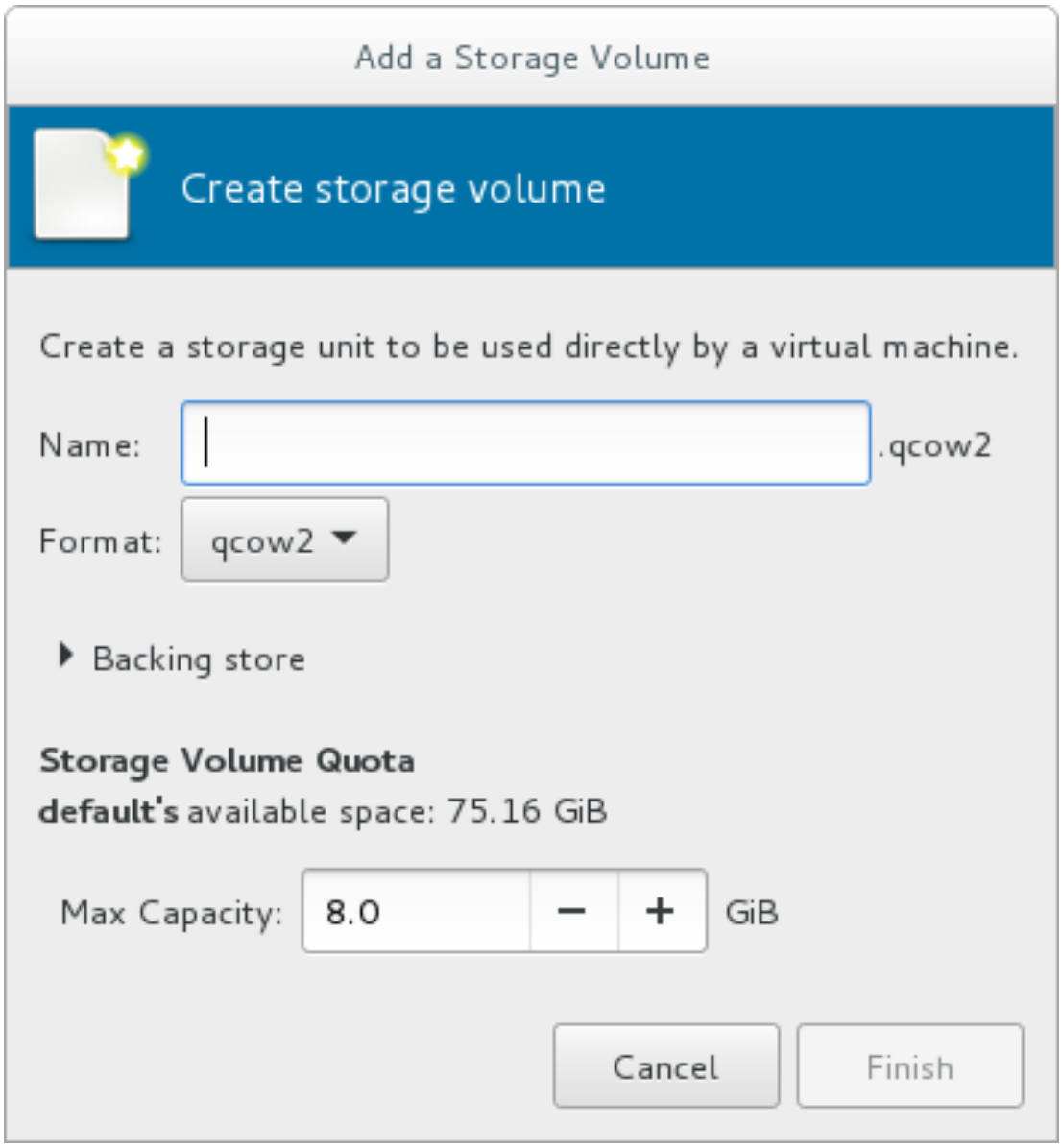


Figure 3.8. The Add a Storage Volume window

Select the new volume and click **Choose volume**. Next, click **Finish** to return to the **New VM** wizard. Click **Forward** to continue.

7. **Name and final configuration**
   Name the virtual machine. Virtual machine names can contain letters, numbers and the following characters: underscores (_), periods (.), and hyphens (-). Virtual machine names must be unique for migration and cannot consist only of numbers.

   By default, the virtual machine will be created with network address translation (NAT) for a network called 'default' . To change the network selection, click **Network selection** and select a host device and source mode.

   Verify the settings of the virtual machine and click **Finish** when you are satisfied; this will create the virtual machine with specified networking settings, virtualization type, and architecture.

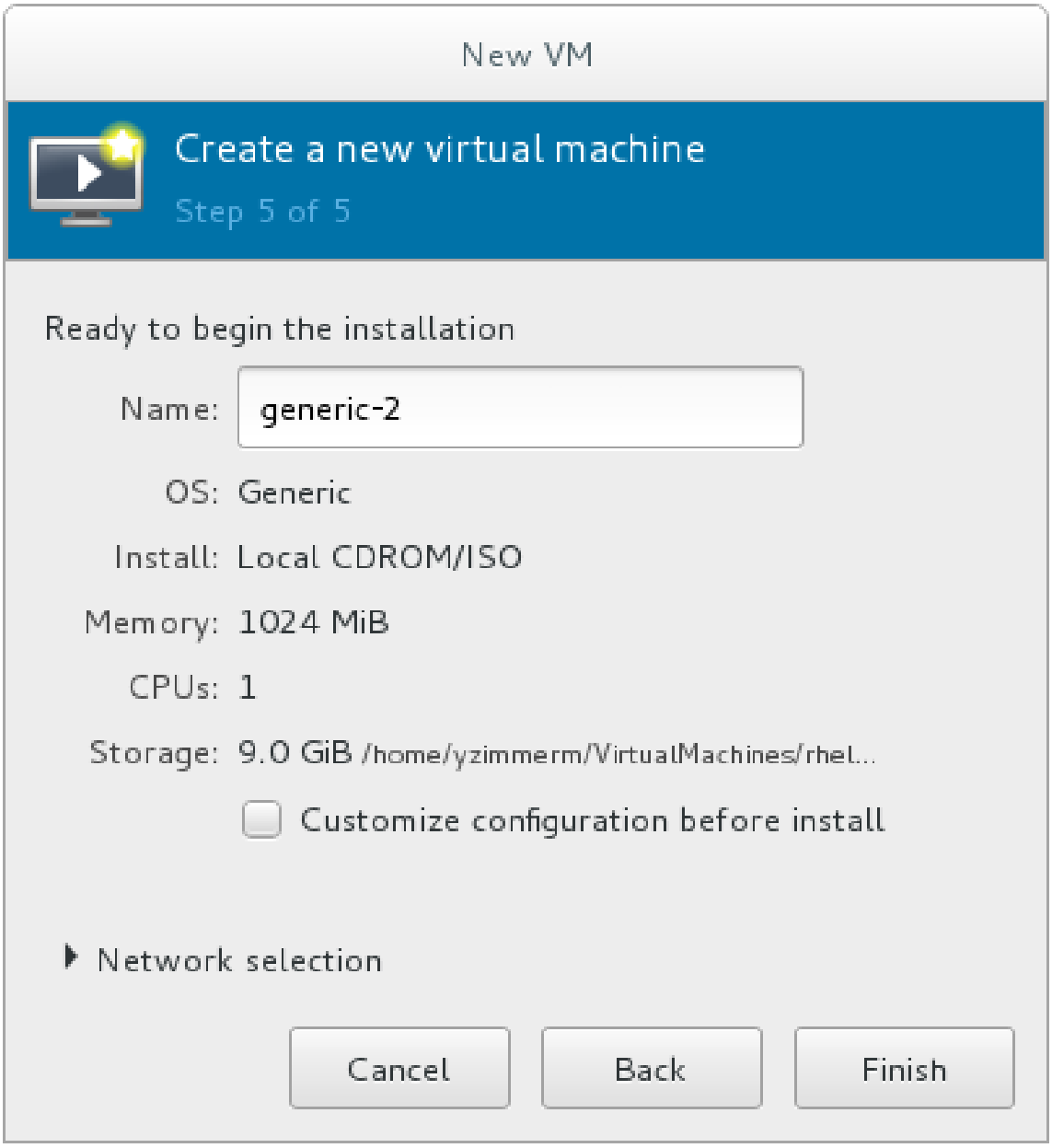


Figure 3.9. Verifying the configuration

Or, to further configure the virtual machine's hardware, check the **Customize configuration before install** check box to change the guest's storage or network devices, to use the paravirtualized (virtio) drivers or to add additional devices. This opens another wizard that will allow you to add, remove, and configure the virtual machine's hardware settings.

NOTE

Red Hat Enterprise Linux 4 or Red Hat Enterprise Linux 5 guest virtual machines cannot be installed using graphical mode. As such, you must select "Cirrus" instead of "QXL" as a video card.

After configuring the virtual machine's hardware, click **Apply**. **virt-manager** will then create the virtual machine with your specified hardware settings.

WARNING

When installing a Red Hat Enterprise Linux 7 guest virtual machine from a remote medium but without a configured TCP/IP connection, the installation fails. However, when installing a guest virtual machine of Red Hat Enterprise Linux 5 or 6 in such circumstances, the installer opens a "Configure TCP/IP" interface.

For further information about this difference, see the related knowledgebase article.

Click **Finish** to continue into the Red Hat Enterprise Linux installation sequence. For more information on installing Red Hat Enterprise Linux 7, see the Red Hat Enterprise Linux 7 Installation Guide.

A Red Hat Enterprise Linux 7 guest virtual machine is now created from an ISO installation disk image.

3.4. COMPARISON OF VIRT-INSTALL AND VIRT-MANAGER INSTALLATION OPTIONS

This table provides a quick reference to compare equivalent **virt-install** and **virt-manager** installation options for when installing a virtual machine.

Most **virt-install** options are not required. The minimum requirements are **--name**, **--memory**, guest storage (**--disk**, **--filesystem** or **--disk none**), and an install method (**--location**, **--cdrom**, **--pxe**, **--import**, or **boot**). These options are further specified with arguments; to see a complete list of command options and related arguments, enter the following command:

```
# virt-install --help
```

In **virt-manager**, at minimum, a name, installation method, memory (RAM), vCPUs, storage are required.

Table 3.1. virt-install and virt-manager configuration comparison for guest installations

Configuration on virtual machine	virt-install option	virt-manager installation wizard label and step number
Virtual machine name	--name, -n	Name (step 5)
RAM to allocate (MiB)	--ram, -r	Memory (RAM) (step 3)
Storage - specify storage media	--disk	Enable storage for this virtual machine → Create a disk image on the computer's hard drive, or Select managed or other existing storage (step 4)

Configuration on virtual machine	`virt-install` option	virt-manager installation wizard label and step number
Storage - export a host directory to the guest	--filesystem	Enable storage for this virtual machine → Select managed or other existing storage (step 4)
Storage - configure no local disk storage on the guest	--nodisks	Deselect the Enable storage for this virtual machine check box (step 4)
Installation media location (local install)	--file	Local install media → Locate your install media (steps 1-2)
Installation using a distribution tree (network install)	--location	Network install → URL (steps 1-2)
Install guest with PXE	--pxe	Network boot (step 1)
Number of vCPUs	--vcpus	CPUs (step 3)
Host network	--network	Advanced options drop-down menu (step 5)
Operating system variant/version	--os-variant	Version (step 2)
Graphical display method	--graphics, --nographics	** virt-manager provides GUI installation only*

CHAPTER 4. CLONING VIRTUAL MACHINES

There are two types of guest virtual machine instances used in creating guest copies:

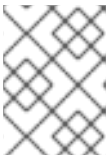
- *Clones* are instances of a single virtual machine. Clones can be used to set up a network of identical virtual machines, and they can also be distributed to other destinations.
- *Templates* are instances of a virtual machine that are designed to be used as a source for cloning. You can create multiple clones from a template and make minor modifications to each clone. This is useful in seeing the effects of these changes on the system.

Both clones and templates are virtual machine instances. The difference between them is in how they are used.

For the created clone to work properly, information and configurations unique to the virtual machine that is being cloned usually has to be removed before cloning. The information that needs to be removed differs, based on how the clones will be used.

The information and configurations to be removed may be on any of the following levels:

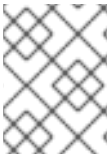
- *Platform level* information and configurations include anything assigned to the virtual machine by the virtualization solution. Examples include the number of Network Interface Cards (NICs) and their MAC addresses.
- *Guest operating system level* information and configurations include anything configured within the virtual machine. Examples include SSH keys.
- *Application level* information and configurations include anything configured by an application installed on the virtual machine. Examples include activation codes and registration information.



NOTE

This chapter does not include information about removing the application level, because the information and approach is specific to each application.

As a result, some of the information and configurations must be removed from within the virtual machine, while other information and configurations must be removed from the virtual machine using the virtualization environment (for example, Virtual Machine Manager or VMware).



NOTE

For information on cloning storage volumes, see Section 13.3.2.1, “Creating Storage Volumes with virsh”.

4.1. PREPARING VIRTUAL MACHINES FOR CLONING

Before cloning a virtual machine, it must be prepared by running the virt-sysprep utility on its disk image, or by using the following steps:

Procedure 4.1. Preparing a virtual machine for cloning

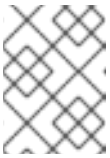
1. **Setup the virtual machine**
   a. Build the virtual machine that is to be used for the clone or template.

- Install any software needed on the clone.
- Configure any non-unique settings for the operating system.
- Configure any non-unique application settings.

2. **Remove the network configuration**

a. Remove any persistent udev rules using the following command:

```
# rm -f /etc/udev/rules.d/70-persistent-net.rules
```



NOTE

If udev rules are not removed, the name of the first NIC may be eth1 instead of eth0.

b. Remove unique network details from ifcfg scripts by making the following edits to **`/etc/sysconfig/network-scripts/ifcfg-eth[x]`**:

i. Remove the HWADDR and Static lines

NOTE

If the HWADDR does not match the new guest's MAC address, the ifcfg will be ignored. Therefore, it is important to remove the HWADDR from the file.

```
DEVICE=eth[x]
BOOTPROTO=none
ONBOOT=yes
#NETWORK=10.0.1.0       <- REMOVE
#NETMASK=255.255.255.0  <- REMOVE
#IPADDR=10.0.1.20       <- REMOVE
#HWADDR=xx:xx:xx:xx:xx  <- REMOVE
#USERCTL=no             <- REMOVE
# Remove any other *unique* or non-desired settings, such as
UUID.
```

ii. Ensure that a DHCP configuration remains that does not include a HWADDR or any unique information.

```
DEVICE=eth[x]
BOOTPROTO=dhcp
ONBOOT=yes
```

iii. Ensure that the file includes the following lines:

```
DEVICE=eth[x]
ONBOOT=yes
```

c. If the following files exist, ensure that they contain the same content:

- **/etc/sysconfig/networking/devices/ifcfg-eth[x]**
- **/etc/sysconfig/networking/profiles/default/ifcfg-eth[x]**

NOTE

If NetworkManager or any special settings were used with the virtual machine, ensure that any additional unique information is removed from the ifcfg scripts.

3. **Remove registration details**

   a. Remove registration details using one of the following:

   - For Red Hat Network (RHN) registered guest virtual machines, use the following command:

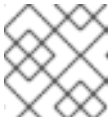
```
# rm /etc/sysconfig/rhn/systemid
```

   - For Red Hat Subscription Manager (RHSM) registered guest virtual machines:
     - If the original virtual machine will not be used, use the following commands:

```
# subscription-manager unsubscribe --all
# subscription-manager unregister
# subscription-manager clean
```

     - If the original virtual machine will be used, run only the following command:

```
# subscription-manager clean
```



NOTE

The original RHSM profile remains in the portal.

4. **Removing other unique details**

   a. Remove any sshd public/private key pairs using the following command:

```
# rm -rf /etc/ssh/ssh_host_*
```

NOTE

Removing ssh keys prevents problems with ssh clients not trusting these hosts.

   b. Remove any other application-specific identifiers or configurations that may cause conflicts if running on multiple machines.

5. **Configure the virtual machine to run configuration wizards on the next boot**

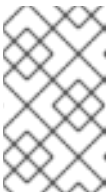
a. Configure the virtual machine to run the relevant configuration wizards the next time it is booted by doing one of the following:

   - For Red Hat Enterprise Linux 6 and below, create an empty file on the root file system called .unconfigured using the following command:

     ```
     # touch /.unconfigured
     ```

   - For Red Hat Enterprise Linux 7, enable the first boot and initial-setup wizards by running the following commands:

     ```
     # sed -ie 's/RUN_FIRSTBOOT=NO/RUN_FIRSTBOOT=YES/'
     /etc/sysconfig/firstboot
     # systemctl enable firstboot-graphical
     # systemctl enable initial-setup-graphical
     ```



NOTE

The wizards that run on the next boot depend on the configurations that have been removed from the virtual machine. In addition, on the first boot of the clone, it is recommended that you change the hostname.

4.2. CLONING A VIRTUAL MACHINE

Before proceeding with cloning, shut down the virtual machine. You can clone the virtual machine using **virt-clone** or **virt-manager**.

4.2.1. Cloning Guests with virt-clone

You can use **virt-clone** to clone virtual machines from the command line.

Note that you need root privileges for **virt-clone** to complete successfully.

The **virt-clone** command provides a number of options that can be passed on the command line. These include general options, storage configuration options, networking configuration options, and miscellaneous options. Only the **--original** is required. To see a complete list of options, enter the following command:

```
# virt-clone --help
```

The **virt-clone** man page also documents each command option, important variables, and examples.

The following example shows how to clone a guest virtual machine called "demo" on the default connection, automatically generating a new name and disk clone path.

Example 4.1. Using virt-clone to clone a guest

```
# virt-clone --original demo --auto-clone
```

The following example shows how to clone a QEMU guest virtual machine called "demo" with multiple disks.

Example 4.2. Using `virt-clone` to clone a guest

```
# virt-clone --connect qemu:///system --original demo --name newdemo --
file /var/lib/libvirt/images/newdemo.img --file
/var/lib/libvirt/images/newdata.img
```

4.2.2. Cloning Guests with virt-manager

This procedure describes cloning a guest virtual machine using the **virt-manager** utility.

Procedure 4.2. Cloning a Virtual Machine with virt-manager

1. **Open virt-manager**
   Start **virt-manager**. Launch the **Virtual Machine Manager** application from the **Applications** menu and **System Tools** submenu. Alternatively, run the **`virt-manager`** command as root.

   Select the guest virtual machine you want to clone from the list of guest virtual machines in **Virtual Machine Manager**.

   Right-click the guest virtual machine you want to clone and select **Clone**. The Clone Virtual Machine window opens.

Figure 4.1. Clone Virtual Machine window

2. **Configure the clone**
   - To change the name of the clone, enter a new name for the clone.

- To change the networking configuration, click **Details**.

  Enter a new MAC address for the clone.

  Click **OK**.

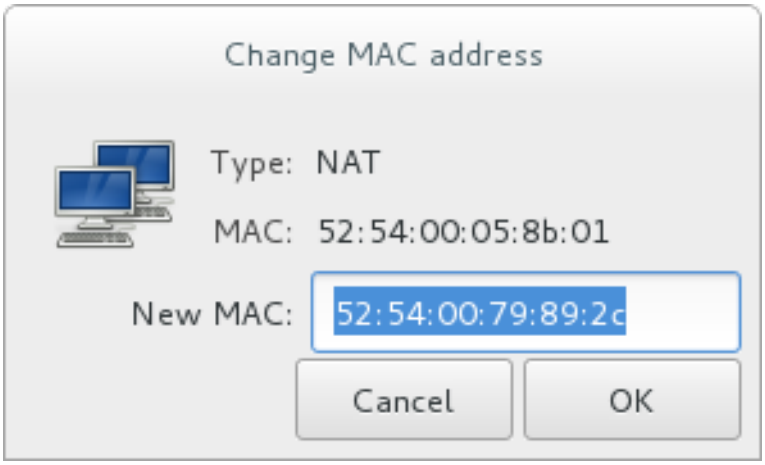


Figure 4.2. Change MAC Address window

- For each disk in the cloned guest virtual machine, select one of the following options:

  - **Clone this disk** - The disk will be cloned for the cloned guest virtual machine
  - **Share disk with *guest virtual machine name*** - The disk will be shared by the guest virtual machine that will be cloned and its clone
  - **Details** - Opens the Change storage path window, which enables selecting a new path for the disk

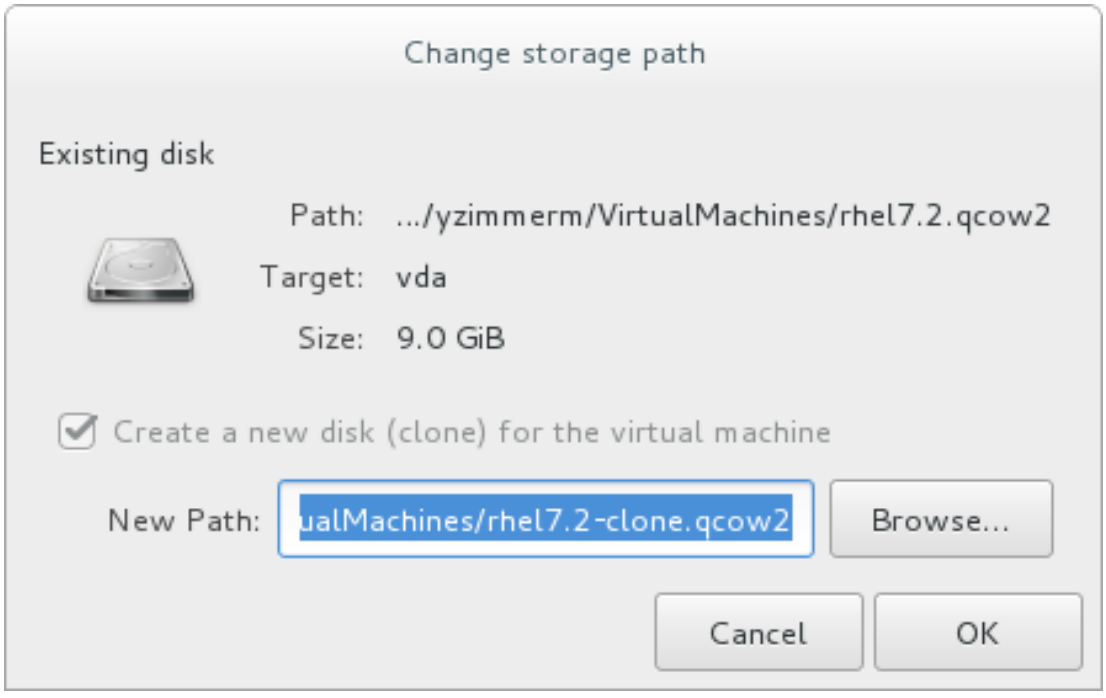


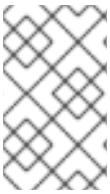
Figure 4.3. Change storage path window

3. **Clone the guest virtual machine**
   Click **Clone**.

CHAPTER 5. KVM PARAVIRTUALIZED (VIRTIO) DRIVERS

Paravirtualized drivers enhance the performance of guests, decreasing guest I/O latency and increasing throughput almost to bare-metal levels. It is recommended to use the paravirtualized drivers for fully virtualized guests running I/O-heavy tasks and applications.

Virtio drivers are KVM's paravirtualized device drivers, available for guest virtual machines running on KVM hosts. These drivers are included in the **virtio** package. The virtio package supports block (storage) devices and network interface controllers.



NOTE

PCI devices are limited by the virtualized system architecture. See Chapter 16, *Guest Virtual Machine Device Configuration* for additional limitations when using assigned devices.

5.1. USING KVM VIRTIO DRIVERS FOR EXISTING STORAGE DEVICES

You can modify an existing hard disk device attached to the guest to use the **virtio** driver instead of the virtualized IDE driver. The example shown in this section edits libvirt configuration files. Note that the guest virtual machine does not need to be shut down to perform these steps, however the change will not be applied until the guest is completely shut down and rebooted.

Procedure 5.1. Using KVM virtio drivers for existing devices

1. Ensure that you have installed the appropriate driver (**viostor**), before continuing with this procedure.

2. Run the **virsh edit *guestname*** command as root to edit the XML configuration file for your device. For example, **virsh edit guest1**. The configuration files are located in the **/etc/libvirt/qemu/** directory.

3. Below is a file-based block device using the virtualized IDE driver. This is a typical entry for a virtual machine not using the virtio drivers.

```
<disk type='file' device='disk'>
  ...
   <source file='/var/lib/libvirt/images/disk1.img'/>
   <target dev='hda' bus='ide'/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x07'
function='0x0'/>
</disk>
```

4. Change the entry to use the virtio device by modifying the **bus=** entry to **virtio**. Note that if the disk was previously IDE, it has a target similar to **hda**, **hdb**, or **hdc**. When changing to **bus=virtio** the target needs to be changed to **vda**, **vdb**, or **vdc** accordingly.

```
<disk type='file' device='disk'>
   ...
   <source file='/var/lib/libvirt/images/disk1.img'/>
   <target dev='vda' bus='virtio'/>
```

```
  <address type='pci' domain='0x0000' bus='0x00' slot='0x07'
function='0x0'/>
</disk>
```

5. Remove the **address** tag inside the **disk** tags. This must be done for this procedure to work. Libvirt will regenerate the **address** tag appropriately the next time the virtual machine is started.

Alternatively, **virt-manager**, **virsh attach-disk** or **virsh attach-interface** can add a new device using the virtio drivers.

See the libvirt website for more details on using Virtio: http://www.linux-kvm.org/page/Virtio

5.2. USING KVM VIRTIO DRIVERS FOR NEW STORAGE DEVICES

This procedure covers creating new storage devices using the KVM virtio drivers with **virt-manager**.

Alternatively, the **virsh attach-disk** or **virsh attach-interface** commands can be used to attach devices using the virtio drivers.

IMPORTANT

Ensure the drivers have been installed on the guest before proceeding to install new devices. If the drivers are unavailable the device will not be recognized and will not work.

Procedure 5.2. Adding a storage device using the virtio storage driver

1. Open the guest virtual machine by double clicking the name of the guest in **virt-manager**.
2. Open the **Show virtual hardware details** tab by clicking .
3. In the **Show virtual hardware details** tab, click the **Add Hardware** button.
4. **Select hardware type**
   Select **Storage** as the **Hardware type**.

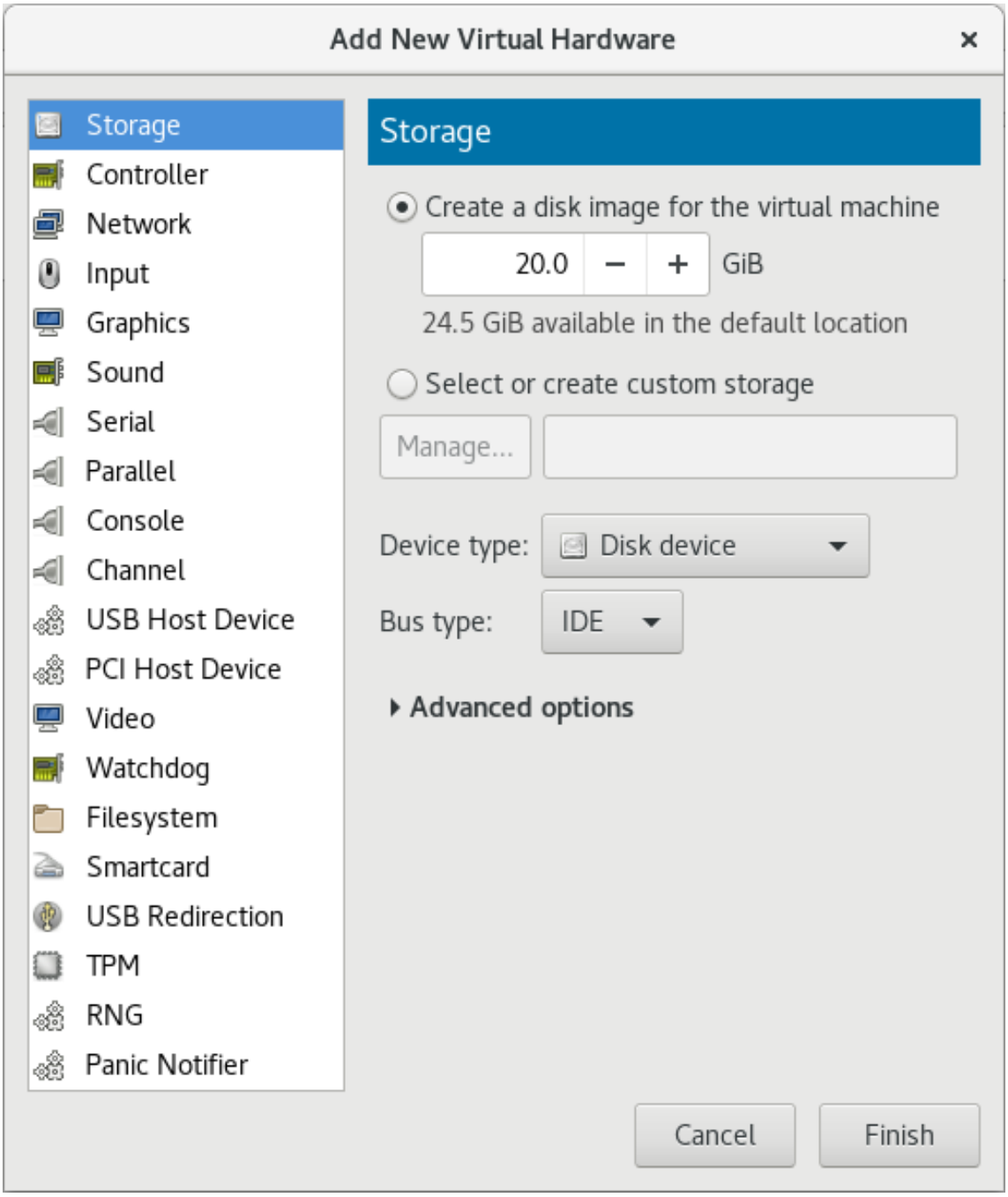


Figure 5.1. The Add new virtual hardware wizard

5. **Select the storage device and driver**
   Create a new disk image or select a storage pool volume.

   Set the **`Device type`** to **`Disk device`** and the **`Bus type`** to **`VirtIO`** to use the virtio drivers.

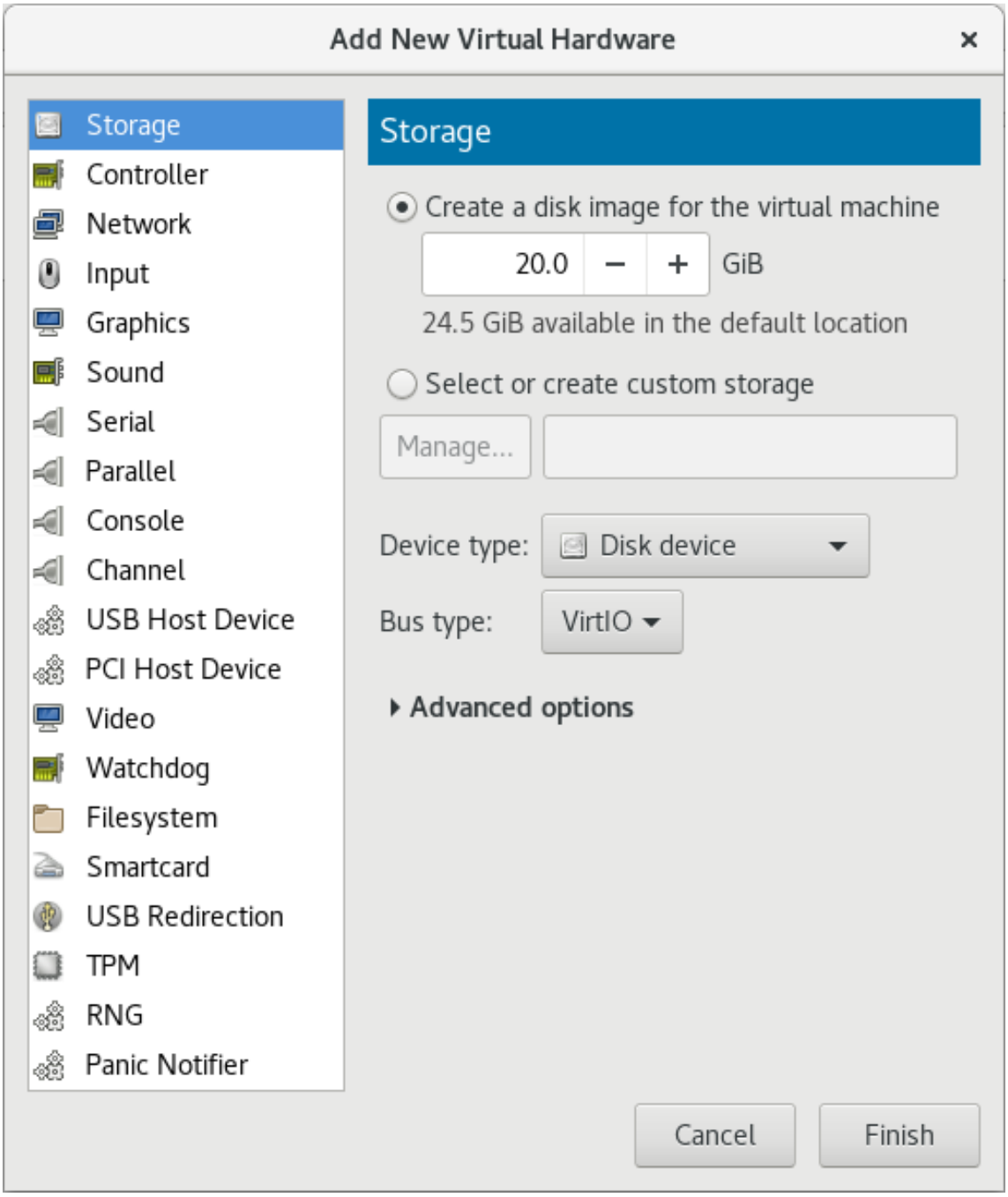


Figure 5.2. The Add New Virtual Hardware wizard

Click **Finish** to complete the procedure.

Procedure 5.3. Adding a network device using the virtio network driver

1. Open the guest virtual machine by double clicking the name of the guest in **virt-manager**.
2. Open the **Show virtual hardware details** tab by clicking .
3. In the **Show virtual hardware details** tab, click the **Add Hardware** button.
4. **Select hardware type**
   Select **Network** as the **Hardware type**.

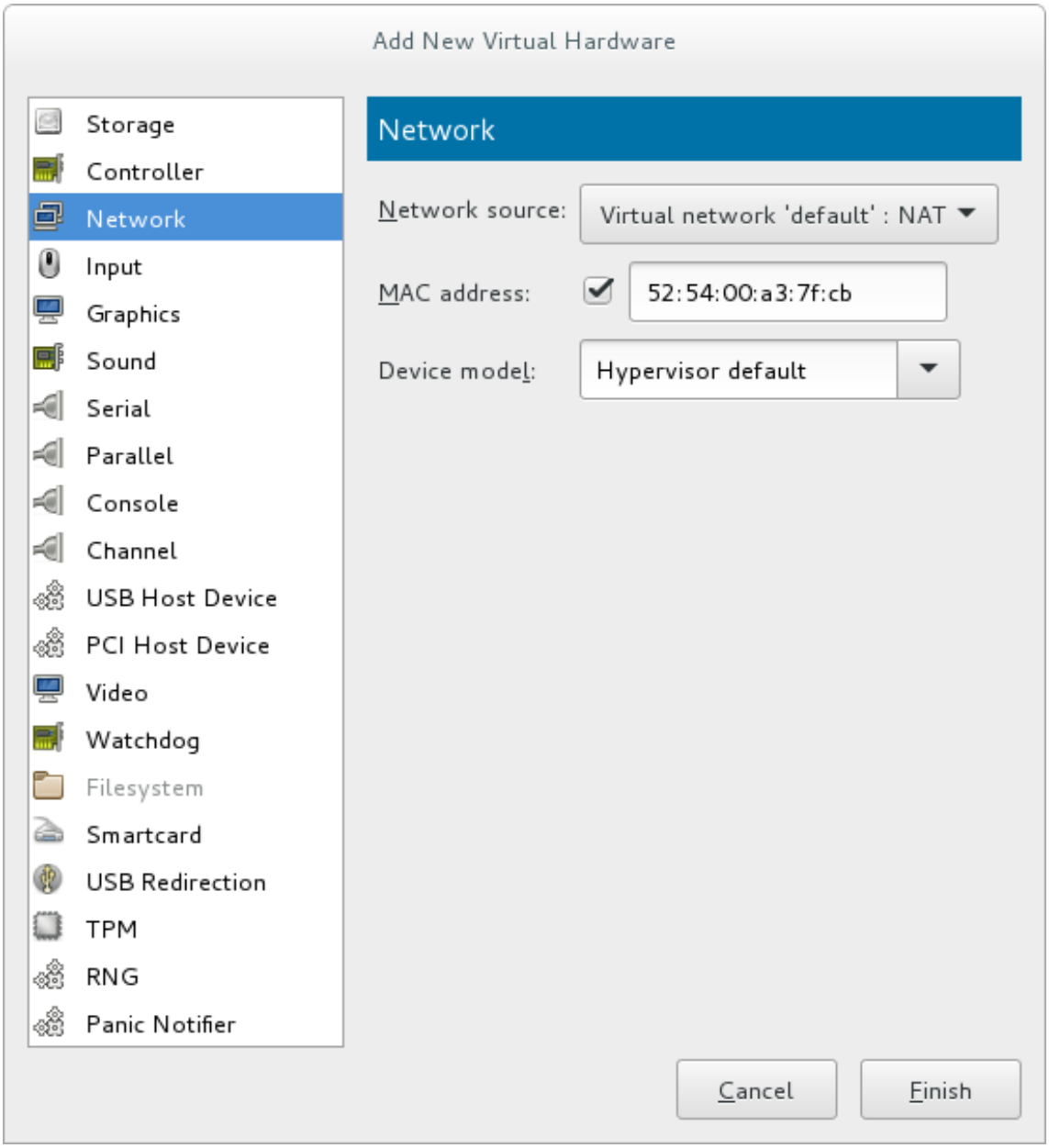


Figure 5.3. The Add new virtual hardware wizard

5. **Select the network device and driver**
   Set the **`Device model`** to **`virtio`** to use the virtio drivers. Choose the required **`Host device`**.

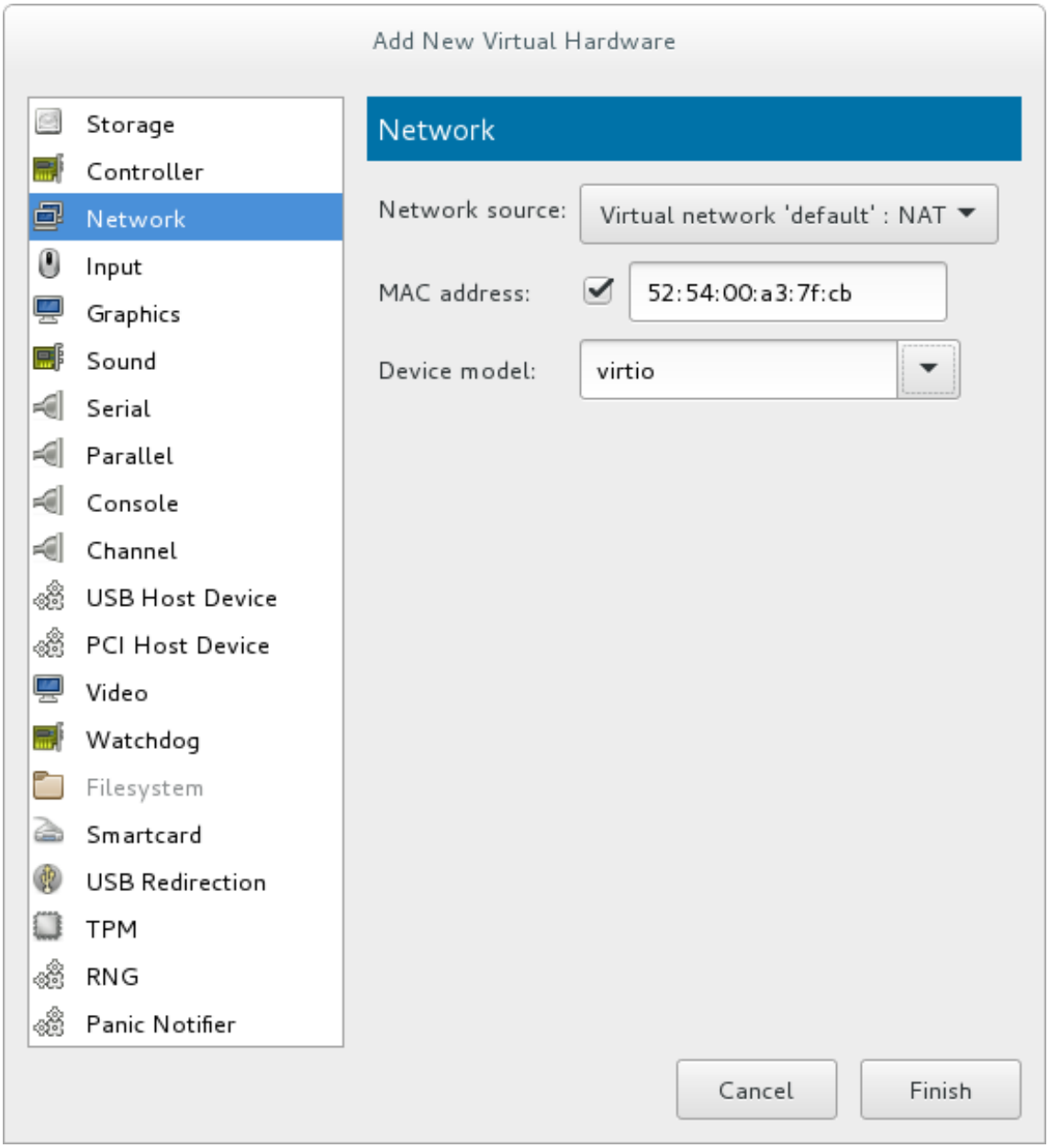


Figure 5.4. The Add new virtual hardware wizard

Click **Finish** to complete the procedure.

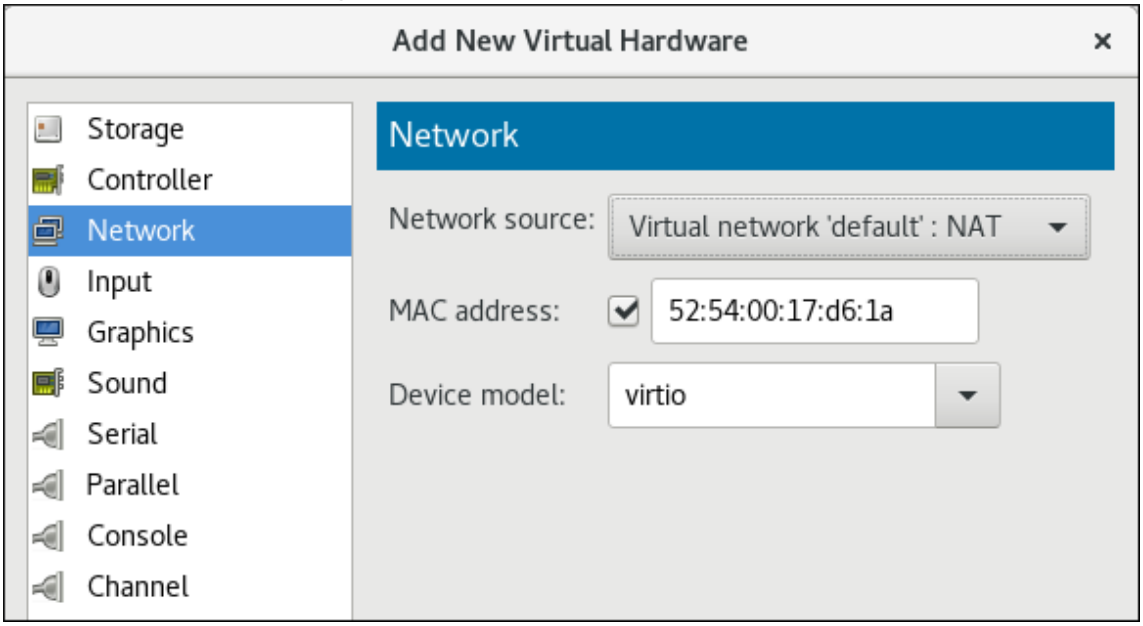
Once all new devices are added, reboot the virtual machine. Virtual machines may not recognize the devices until the guest is rebooted.

5.3. USING KVM VIRTIO DRIVERS FOR NETWORK INTERFACE DEVICES

When network interfaces use KVM virtio drivers, KVM does not emulate networking hardware which removes processing overhead and can increase the guest performance. In Red Hat Enterprise Linux 7, virtio is used as the default network interface type. However, if this is configured differently on your system, you can use the following procedures:

- To **attach a virtio network device** to a guest, use the `virsh attach-interface` command with the `model --virtio` option.

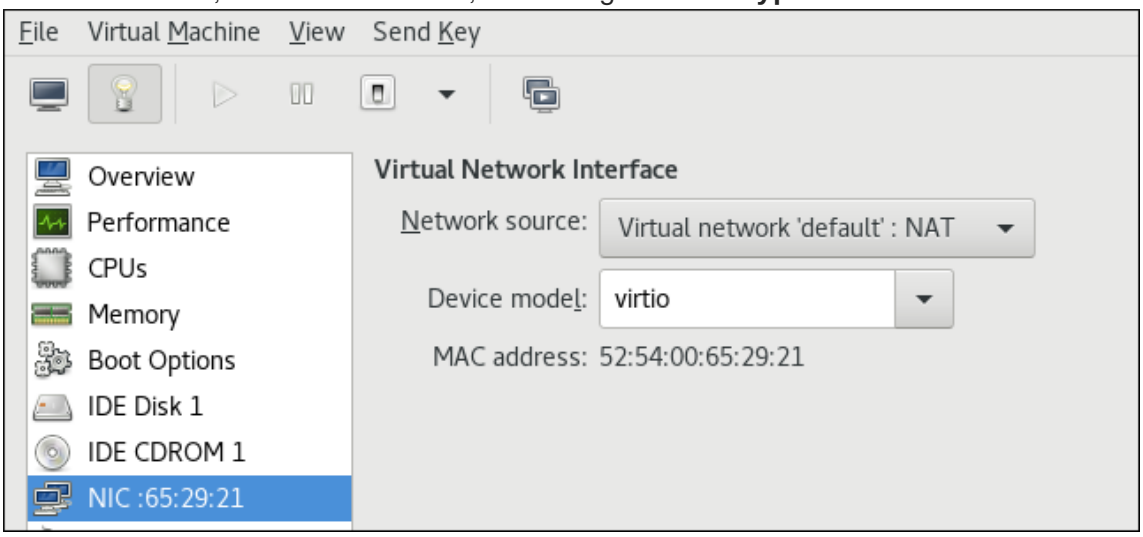
Alternatively, in the **virt-manager** interface, navigate to the guest's **Virtual hardware details** screen and click **Add Hardware**. In the **Add New Virtual Hardware** screen, select **Network**, and change **Device model** to **virtio**:

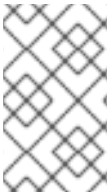


- To **change the type of an existing interface to virtio**, use the **virsh edit** command to edit the XML configuration of the intended guest, and change the **model type** attribute to **virtio**, for example as follows:

```
  <devices>
    <interface type='network'>
      <source network='default'/>
      <target dev='vnet1'/>
      <model type='virtio'/>
      <driver name='vhost' txmode='iothread' ioeventfd='on'
event_idx='off'/>
    </interface>
  </devices>
  ...
```

Alternatively, in the **virt-manager** interface, navigate to the guest's **Virtual hardware details** screen, select the **NIC** item, and change **Model type** to **virtio**:





NOTE

If the naming of network interfaces inside the guest is not consistent across reboots, ensure all interfaces presented to the guest are of the same device model, preferably **`virtio-net`**. For details, see the Red Hat KnowledgeBase.

CHAPTER 6. NETWORK CONFIGURATION

This chapter provides an introduction to the common networking configurations used by libvirt-based guest virtual machines.

Red Hat Enterprise Linux 7 supports the following networking setups for virtualization:

- virtual networks using Network Address Translation (*NAT*)
- directly allocated physical devices using PCI device assignment
- directly allocated virtual functions using PCIe SR-IOV
- bridged networks

You must enable NAT, network bridging or directly assign a PCI device to allow external hosts access to network services on guest virtual machines.

6.1. NETWORK ADDRESS TRANSLATION (NAT) WITH LIBVIRT

One of the most common methods for sharing network connections is to use Network Address Translation (NAT) forwarding (also known as virtual networks).

Host Configuration

Every standard **libvirt** installation provides NAT-based connectivity to virtual machines as the default virtual network. Verify that it is available with the **virsh net-list --all** command.

```
# virsh net-list --all
Name                 State      Autostart
-----------------------------------------
default              active     yes
```

If it is missing, the following can be used in the XML configuration file (such as /etc/libvirtd/qemu/myguest.xml) for the guest:

```
# ll /etc/libvirt/qemu/
total 12
drwx------. 3 root root 4096 Nov  7 23:02 networks
-rw-------. 1 root root 2205 Nov 20 01:20 r6.4.xml
-rw-------. 1 root root 2208 Nov  8 03:19 r6.xml
```

The default network is defined from **/etc/libvirt/qemu/networks/default.xml**

Mark the default network to automatically start:

```
# virsh net-autostart default
Network default marked as autostarted
```

Start the default network:

```
# virsh net-start default
Network default started
```

Once the **libvirt** default network is running, you will see an isolated bridge device. This device does *not* have any physical interfaces added. The new device uses NAT and IP forwarding to connect to the physical network. Do not add new interfaces.

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
virbr0          8000.000000000000       yes
```

libvirt adds **iptables** rules which allow traffic to and from guest virtual machines attached to the **virbr0** device in the **INPUT**, **FORWARD**, **OUTPUT** and **POSTROUTING** chains. **libvirt** then attempts to enable the **ip_forward** parameter. Some other applications may disable **ip_forward**, so the best option is to add the following to **/etc/sysctl.conf**.

```
 net.ipv4.ip_forward = 1
```

Guest Virtual Machine Configuration

Once the host configuration is complete, a guest virtual machine can be connected to the virtual network based on its name. To connect a guest to the 'default' virtual network, the following can be used in the XML configuration file (such as **/etc/libvirtd/qemu/myguest.xml**) for the guest:

```
<interface type='network'>
   <source network='default'/>
</interface>
```

NOTE

Defining a MAC address is optional. If you do not define one, a MAC address is automatically generated and used as the MAC address of the bridge device used by the network. Manually setting the MAC address may be useful to maintain consistency or easy reference throughout your environment, or to avoid the very small chance of a conflict.

```
<interface type='network'>
  <source network='default'/>
  <mac address='00:16:3e:1a:b3:4a'/>
</interface>
```

6.2. DISABLING VHOST-NET

The **vhost-net** module is a kernel-level back end for virtio networking that reduces virtualization overhead by moving virtio packet processing tasks out of user space (the QEMU process) and into the kernel (the **vhost-net** driver). vhost-net is only available for virtio network interfaces. If the vhost-net kernel module is loaded, it is enabled by default for all virtio interfaces, but can be disabled in the interface configuration if a particular workload experiences a degradation in performance when vhost-net is in use.

Specifically, when UDP traffic is sent from a host machine to a guest virtual machine on that host, performance degradation can occur if the guest virtual machine processes incoming data at a rate slower than the host machine sends it. In this situation, enabling **vhost-net** causes the UDP socket's receive buffer to overflow more quickly, which results in greater packet loss. It is therefore better to disable **vhost-net** in this situation to slow the traffic, and improve overall performance.

To disable **vhost-net**, edit the **<interface>** sub-element in the guest virtual machine's XML configuration file and define the network as follows:

```
<interface type="network">
   ...
   <model type="virtio"/>
   <driver name="qemu"/>
   ...
</interface>
```

Setting the driver name to **qemu** forces packet processing into QEMU user space, effectively disabling vhost-net for that interface.

6.3. ENABLING VHOST-NET ZERO-COPY

In Red Hat Enterprise Linux 7, vhost-net zero-copy is disabled by default. To enable this action on a permanent basis, add a new file **vhost-net.conf** to **/etc/modprobe.d** with the following content:

```
options vhost_net  experimental_zcopytx=1
```

If you want to disable this again, you can run the following:

```
modprobe -r vhost_net
```

```
modprobe vhost_net experimental_zcopytx=0
```

The first command removes the old file, the second one makes a new file (like above) and disables zero-copy. You can use this to enable as well but the change will not be permanent.

To confirm that this has taken effect, check the output of **cat /sys/module/vhost_net/parameters/experimental_zcopytx**. It should show:

```
$ cat /sys/module/vhost_net/parameters/experimental_zcopytx
0
```

6.4. BRIDGED NETWORKING

Bridged networking (also known as network bridging or virtual network switching) is used to place virtual machine network interfaces on the same network as the physical interface. Bridges require minimal configuration and make a virtual machine appear on an existing network, which reduces management overhead and network complexity. As bridges contain few components and configuration variables, they provide a transparent setup which is straightforward to understand and troubleshoot, if required.

Bridging can be configured in a virtualized environment using standard Red Hat Enterprise Linux tools, virt-manager, or libvirt, and is described in the following sections.

However, even in a virtualized environment, bridges may be more easily created using the host operating system's networking tools. More information about this bridge creation method can be found in the Red Hat Enterprise Linux 7 Networking Guide.

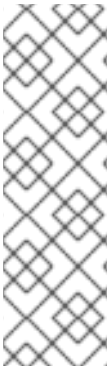
6.4.1. Configuring Bridged Networking on a Red Hat Enterprise Linux 7 Host

Bridged networking can be configured for virtual machines on a Red Hat Enterprise Linux host, independent of the virtualization management tools. This configuration is mainly recommended when the virtualization bridge is the host's only network interface, or is the host's management network interface.

For instructions on configuring network bridging without using virtualization tools, see the Red Hat Enterprise Linux 7 Networking Guide.

6.4.2. Bridged Networking with Virtual Machine Manager

This section provides instructions on creating a bridge from a host machine's interface to a guest virtual machine using virt-manager.



NOTE

Depending on your environment, setting up a bridge with libvirt tools in Red Hat Enterprise Linux 7 may require disabling Network Manager, which is not recommended by Red Hat. A bridge created with libvirt also requires libvirtd to be running for the bridge to maintain network connectivity.

It is recommended to configure bridged networking on the physical Red Hat Enterprise Linux host as described in the Red Hat Enterprise Linux 7 Networking Guide, while using libvirt after bridge creation to add virtual machine interfaces to the bridges.

Procedure 6.1. Creating a bridge with virt-manager

1. From the virt-manager main menu, click **Edit ⇒ Connection Details** to open the **`Connection Details`** window.
2. Click the **Network Interfaces** tab.
3. Click the **+** at the bottom of the window to configure a new network interface.
4. In the **Interface type** drop-down menu, select **Bridge**, and then click **Forward** to continue.

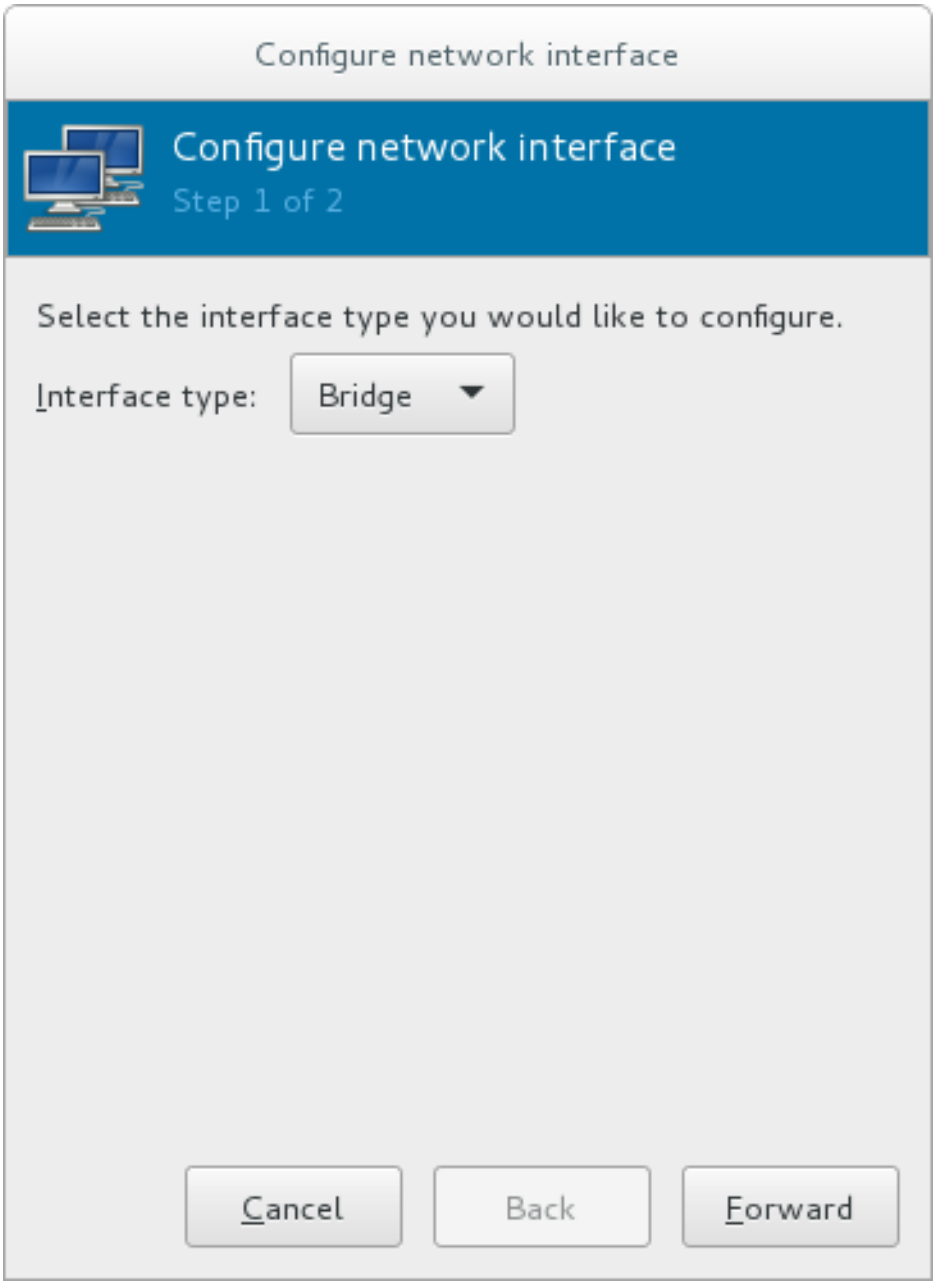


Figure 6.1. Adding a bridge

5. a. In the **Name** field, enter a name for the bridge, such as *br0*.

   b. Select a **Start mode** from the drop-down menu. Choose from one of the following:

      - none - deactivates the bridge
      - onboot - activates the bridge on the next guest virtual machine reboot
      - hotplug - activates the bridge even if the guest virtual machine is running

   c. Check the **Activate now** check box to activate the bridge immediately.

   d. To configure either the **IP settings** or **Bridge settings**, click the appropriate **Configure** button. A separate window will open to specify the required settings. Make any necessary changes and click **OK** when done.

   e. Select the physical interface to connect to your virtual machines. If the interface is currently in use by another guest virtual machine, you will receive a warning message.

6. Click **Finish** and the wizard closes, taking you back to the **Connections** menu.

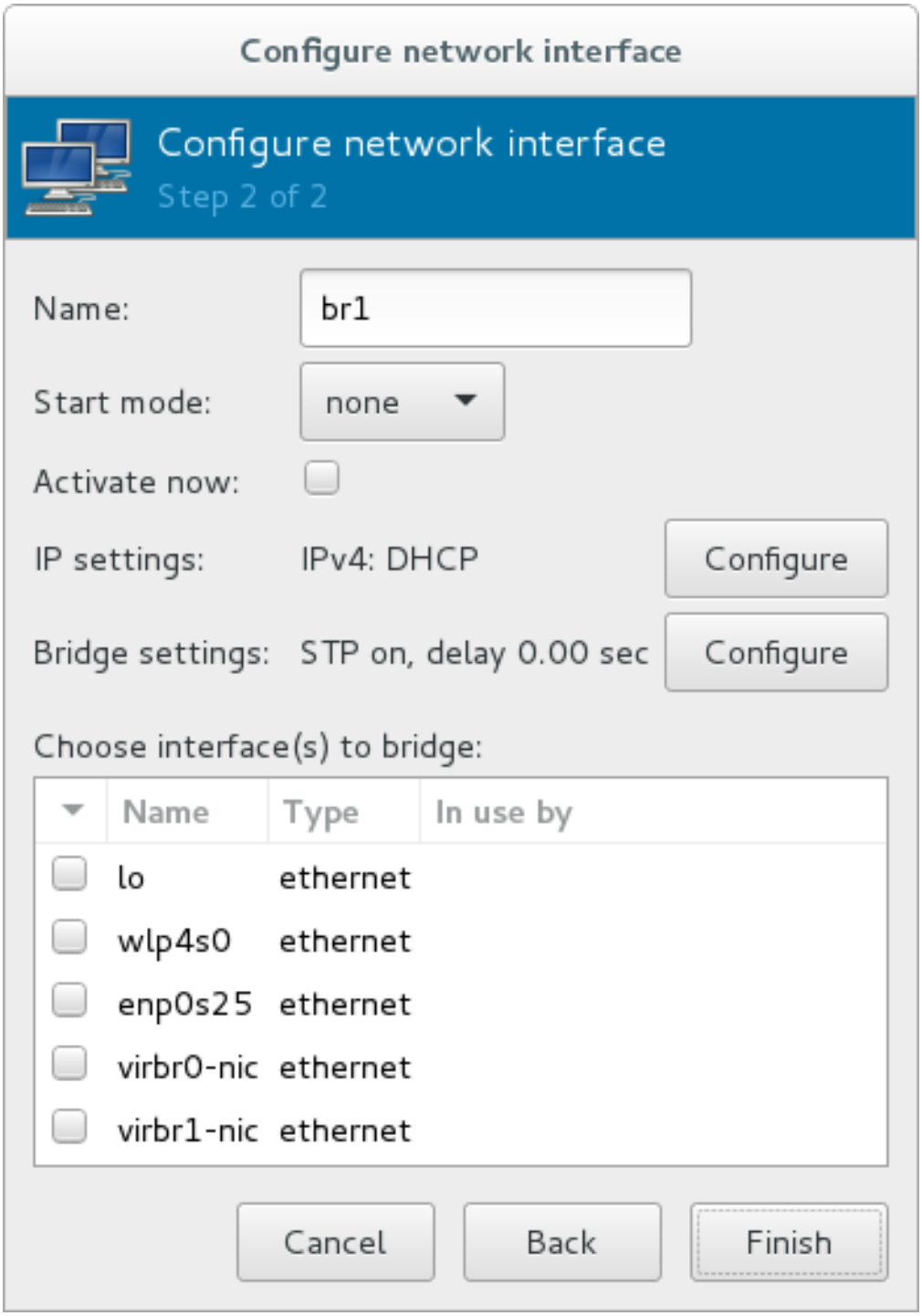


Figure 6.2. Adding a bridge

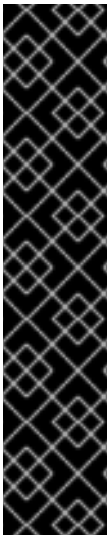
Select the bridge to use, and click **`Apply`** to exit the wizard.

To stop the interface, click the **`Stop Interface`** key. Once the bridge is stopped, to delete the interface, click the **`Delete Interface`** key.

6.4.3. Bridged Networking with libvirt

Depending on your environment, setting up a bridge with libvirt in Red Hat Enterprise Linux 7 may require disabling Network Manager, which is not recommended by Red Hat. This also requires libvirtd to be running for the bridge to operate.

It is recommended to configure bridged networking on the physical Red Hat Enterprise Linux host as described in the Red Hat Enterprise Linux 7 Networking Guide.



IMPORTANT

libvirt is now able to take advantage of new kernel tunable parameters to manage host bridge forwarding database (FDB) entries, thus potentially improving system network performance when bridging multiple virtual machines. Set the ***macTableManager*** attribute of a network's ***<bridge>*** element to **'libvirt'** in the host's XML configuration file:

```
<bridge name='br0' macTableManager='libvirt'/>
```

This will turn off learning (flood) mode on all bridge ports, and libvirt will add or remove entries to the FDB as necessary. Along with removing the overhead of learning the proper forwarding ports for MAC addresses, this also allows the kernel to disable promiscuous mode on the physical device that connects the bridge to the network, which further reduces overhead.

CHAPTER 7. OVERCOMMITTING WITH KVM

7.1. INTRODUCTION

The KVM hypervisor automatically overcommits CPUs and memory. This means that more virtualized CPUs and memory can be allocated to virtual machines than there are physical resources on the system. This is possible because most processes do not access 100% of their allocated resources all the time.

As a result, under-utilized virtualized servers or desktops can run on fewer hosts, which saves a number of system resources, with the net effect of less power, cooling, and investment in server hardware.

7.2. OVERCOMMITTING MEMORY

Guest virtual machines running on a KVM hypervisor do not have dedicated blocks of physical RAM assigned to them. Instead, each guest virtual machine functions as a Linux process where the host physical machine's Linux kernel allocates memory only when requested. In addition the host's memory manager can move the guest virtual machine's memory between its own physical memory and swap space.

Overcommitting requires allotting sufficient swap space on the host physical machine to accommodate all guest virtual machines as well as enough memory for the host physical machine's processes. As a basic rule, the host physical machine's operating system requires a maximum of 4GB of memory along with a minimum of 4GB of swap space. For advanced instructions on determining an appropriate size for the swap partition, see the Red Hat KnowledgeBase.

IMPORTANT

Overcommitting is not an ideal solution for general memory issues. The recommended methods to deal with memory shortage are to allocate less memory per guest, add more physical memory to the host, or utilize swap space.

A virtual machine will run slower if it is swapped frequently. In addition, overcommitting can cause the system to run out of memory (OOM), which may lead to the Linux kernel shutting down important system processes. If you decide to overcommit memory, ensure sufficient testing is performed. Contact Red Hat support for assistance with overcommitting.

Overcommitting does not work with all virtual machines, but has been found to work in a desktop virtualization setup with minimal intensive usage or running several identical guests with KSM. For more information on KSM and overcommitting, see the Red Hat Enterprise Linux 7 Virtualization Tuning and Optimization Guide.

IMPORTANT

When device assignment is in use, all virtual machine memory must be statically pre-allocated to enable direct memory access (DMA) with the assigned device. Memory overcommit is therefore not supported with device assignment.

7.3. OVERCOMMITTING VIRTUALIZED CPUS

The KVM hypervisor supports overcommitting virtualized CPUs (vCPUs). Virtualized CPUs can be overcommitted as far as load limits of guest virtual machines allow. Use caution when overcommitting vCPUs, as loads near 100% may cause dropped requests or unusable response times.

In Red Hat Enterprise Linux 7, it is possible to overcommit guests with more than one vCPU, known as symmetric multiprocessing (SMP) virtual machines. However, you may experience performance deterioration when running more cores on the virtual machine than are present on your physical CPU.

For example, a virtual machine with four vCPUs should not be run on a host machine with a dual core processor, but on a quad core host. Overcommitting SMP virtual machines beyond the physical number of processing cores causes significant performance degradation, due to programs getting less CPU time than required. In addition, it is not recommended to have more than 10 total allocated vCPUs per physical processor core.

With SMP guests, some processing overhead is inherent. CPU overcommitting can increase the SMP overhead, because using time-slicing to allocate resources to guests can make inter-CPU communication inside a guest slower. This overhead increases with guests that have a larger number of vCPUs, or a larger overcommit ratio.

Virtualized CPUs are overcommitted best when when a single host has multiple guests, and each guest has a small number of vCPUs, compared to the number of host CPUs. KVM should safely support guests with loads under 100% at a ratio of five vCPUs (on 5 virtual machines) to one physical CPU on one single host. The KVM hypervisor will switch between all of the virtual machines, making sure that the load is balanced.

For best performance, Red Hat recommends assigning guests only as many vCPUs as are required to run the programs that are inside each guest.

IMPORTANT

Applications that use 100% of memory or processing resources may become unstable in overcommitted environments. Do not overcommit memory or CPUs in a production environment without extensive testing, as the CPU overcommit ratio and the amount of SMP are workload-dependent.

CHAPTER 8. KVM GUEST TIMING MANAGEMENT

Virtualization involves several challenges for time keeping in guest virtual machines.

- Interrupts cannot always be delivered simultaneously and instantaneously to all guest virtual machines. This is because interrupts in virtual machines are not true interrupts. Instead, they are injected into the guest virtual machine by the host machine.
- The host may be running another guest virtual machine, or a different process. Therefore, the precise timing typically required by interrupts may not always be possible.

Guest virtual machines without accurate time keeping may experience issues with network applications and processes, as session validity, migration, and other network activities rely on timestamps to remain correct.

KVM avoids these issues by providing guest virtual machines with a paravirtualized clock (**kvm-clock**). However, it is still important to test timing before attempting activities that may be affected by time keeping inaccuracies, such as guest migration.

IMPORTANT

To avoid the problems described above, the Network Time Protocol (NTP) should be configured on the host and the guest virtual machines. On guests using Red Hat Enterprise Linux 6 and earlier, NTP is implemented by the **ntpd** service. For more information, see the Red Hat Enterprise 6 Deployment Guide.

On systems using Red Hat Enterprise Linux 7, NTP time synchronization service can be provided by **ntpd** or by the **chronyd** service. Note that Chrony has some advantages on virtual machines. For more information, see the Configuring NTP Using the chrony Suite and Configuring NTP Using ntpd sections in the Red Hat Enterprise Linux 7 System Administrator's Guide.

The mechanics of guest virtual machine time synchronization

By default, the guest synchronizes its time with the hypervisor as follows:

- When the guest system boots, the guest reads the time from the emulated Real Time Clock (RTC).
- When the NTP protocol is initiated, it automatically synchronizes the guest clock. Afterwards, during normal guest operation, NTP performs clock adjustments in the guest.
- When a guest is resumed after a pause or a restoration process, a command to synchronize the guest clock to a specified value should be issued by the management software (such as virt-manager). This synchronization works only if the QEMU guest agent is installed in the guest and supports the feature. The value to which the guest clock synchronizes is usually the host clock value.

Constant Time Stamp Counter (TSC)

Modern Intel and AMD CPUs provide a constant Time Stamp Counter (TSC). The count frequency of the constant TSC does not vary when the CPU core itself changes frequency, for example to comply with a power-saving policy. A CPU with a constant TSC frequency is necessary in order to use the TSC as a clock source for KVM guests.

Your CPU has a constant Time Stamp Counter if the **constant_tsc** flag is present. To determine if your CPU has the **constant_tsc** flag enter the following command:

```
$ cat /proc/cpuinfo | grep constant_tsc
```

If any output is given, your CPU has the **constant_tsc** bit. If no output is given, follow the instructions below.

Configuring Hosts without a Constant Time Stamp Counter

Systems without a constant TSC frequency cannot use the TSC as a clock source for virtual machines, and require additional configuration. Power management features interfere with accurate time keeping and must be disabled for guest virtual machines to accurately keep time with KVM.

IMPORTANT

These instructions are for AMD revision F CPUs only.

If the CPU lacks the **constant_tsc** bit, disable all power management features . Each system has several timers it uses to keep time. The TSC is not stable on the host, which is sometimes caused by **cpufreq** changes, deep C state, or migration to a host with a faster TSC. Deep C sleep states can stop the TSC. To prevent the kernel using deep C states append **processor.max_cstate=1** to the kernel boot. To make this change persistent, edit values of the **GRUB_CMDLINE_LINUX** key in the **/etc/default/grub**file. For example. if you want to enable emergency mode for each boot, edit the entry as follows:

```
GRUB_CMDLINE_LINUX="emergency"
```

Note that you can specify multiple parameters for the GRUB_CMDLINE_LINUX key, similarly to adding the parameters in the GRUB 2 boot menu.

To disable **cpufreq** (only necessary on hosts without the **constant_tsc**), install kernel-tools and enable the **cpupower.service** (**systemctl enable cpupower.service**). If you want to disable this service every time the guest virtual machine boots, change the configuration file in **/etc/sysconfig/cpupower** and change the CPUPOWER_START_OPTS and CPUPOWER_STOP_OPTS. Valid limits can be found in the **/sys/devices/system/cpu/*cpuid*/cpufreq/scaling_available_governors** files. For more information on this package or on power management and governors, see the *Red Hat Enterprise Linux 7 Power Management Guide*.

8.1. HOST-WIDE TIME SYNCHRONIZATION

Virtual network devices in KVM guests do not support hardware timestamping, which means it is difficult to synchronize the clocks of guests that use a network protocol like NTP or PTP with better accuracy than tens of microseconds.

When a more accurate synchronization of the guests is required, it is recommended to synchronize the clock of the host using NTP or PTP with hardware timestamping, and to synchronize the guests to the host directly. Red Hat Enterprise Linux 7.5 and later provide a virtual PTP hardware clock (PHC), which enables the guests to synchronize to the host with a sub-microsecond accuracy.

To enable the PHC device, do the following:

1. Set the **ptp_kvm** module to load after reboot.

```
# echo ptp_kvm > /etc/modules-load.d/ptp_kvm.conf
```

2. Add the **/dev/ptp0** clock as a reference to the chrony configuration:

```
# echo "refclock PHC /dev/ptp0 poll 2" >> /etc/chrony.conf
```

3. Restart the chrony daemon:

```
# systemctl restart chronyd
```

4. To verify the host-guest time synchronization has been configured correctly, use the **chronyc sources** command on a guest. The output should look similar to the following:

```
# chronyc sources
210 Number of sources = 1
MS Name/IP address         Stratum Poll Reach LastRx Last sample
===============================================================================
===========
#* PHC0                           0   2   377     4     -6ns[   -
6ns] +/-  726ns
```

8.2. REQUIRED TIME MANAGEMENT PARAMETERS FOR RED HAT ENTERPRISE LINUX GUESTS

For certain Red Hat Enterprise Linux guest virtual machines, additional kernel parameters are required for their system time to be synchronised correctly. These parameters can be set by appending them to the end of the **/kernel** line in the **/etc/grub2.cfg** file of the guest virtual machine.

NOTE

Red Hat Enterprise Linux 5.5 and later, Red Hat Enterprise Linux 6.0 and later, and Red Hat Enterprise Linux 7 use **kvm-clock** as their default clock source. Running **kvm-clock** avoids the need for additional kernel parameters, and is recommended by Red Hat.

The table below lists versions of Red Hat Enterprise Linux and the parameters required on the specified systems.

Table 8.1. Kernel parameter requirements

Red Hat Enterprise Linux version	Additional guest kernel parameters
7.0 and later on AMD64 and Intel 64 systems with kvm-clock	Additional parameters are not required
6.1 and later on AMD64 and Intel 64 systems with kvm-clock	Additional parameters are not required
6.0 on AMD64 and Intel 64 systems with kvm-clock	Additional parameters are not required

Red Hat Enterprise Linux version	Additional guest kernel parameters
6.0 on AMD64 and Intel 64 systems without kvm-clock	notsc lpj=*n*

NOTE

The ***lpj*** parameter requires a numeric value equal to the *loops per jiffy* value of the specific CPU on which the guest virtual machine runs. If you do not know this value, do not set the ***lpj*** parameter.

8.3. STEAL TIME ACCOUNTING

Steal time is the amount of CPU time needed by a guest virtual machine that is not provided by the host. Steal time occurs when the host allocates these resources elsewhere: for example, to another guest.

Steal time is reported in the CPU time fields in **/proc/stat**. It is automatically reported by utilities such as **top** and **vmstat**. It is displayed as "%st", or in the "st" column. Note that it cannot be switched off.

Large amounts of steal time indicate CPU contention, which can reduce guest performance. To relieve CPU contention, increase the guest's CPU priority or CPU quota, or run fewer guests on the host.

CHAPTER 9. NETWORK BOOTING WITH LIBVIRT

Guest virtual machines can be booted with PXE enabled. PXE allows guest virtual machines to boot and load their configuration off the network itself. This section demonstrates some basic configuration steps to configure PXE guests with libvirt.

This section does not cover the creation of boot images or PXE servers. It is used to explain how to configure libvirt, in a private or bridged network, to boot a guest virtual machine with PXE booting enabled.

WARNING

These procedures are provided only as an example. Ensure that you have sufficient backups before proceeding.

9.1. PREPARING THE BOOT SERVER

To perform the steps in this chapter you will need:

- A PXE Server (DHCP and TFTP) - This can be a libvirt internal server, manually-configured dhcpd and tftpd, dnsmasq, a server configured by Cobbler, or some other server.
- Boot images - for example, PXELINUX configured manually or by Cobbler.

9.1.1. Setting up a PXE Boot Server on a Private libvirt Network

This example uses the *default* network. Perform the following steps:

Procedure 9.1. Configuring the PXE boot server

1. Place the PXE boot images and configuration in **`/var/lib/tftpboot`**.
2. enter the following commands:

```
# virsh net-destroy default
# virsh net-edit default
```

3. Edit the **`<ip>`** element in the configuration file for the *default* network to include the appropriate address, network mask, DHCP address range, and boot file, where *BOOT_FILENAME* represents the file name you are using to boot the guest virtual machine.

```
<ip address='192.168.122.1' netmask='255.255.255.0'>
   <tftp root='/var/lib/tftpboot' />
   <dhcp>
      <range start='192.168.122.2' end='192.168.122.254' />
      <bootp file='BOOT_FILENAME' />
   </dhcp>
</ip>
```

4. Run:

```
# virsh net-start default
```

5. Boot the guest using PXE (refer to Section 9.2, “Booting a Guest Using PXE”).

9.2. BOOTING A GUEST USING PXE

This section demonstrates how to boot a guest virtual machine with PXE.

9.2.1. Using bridged networking

Procedure 9.2. Booting a guest using PXE and bridged networking

1. Ensure bridging is enabled such that the PXE boot server is available on the network.

2. Boot a guest virtual machine with PXE booting enabled. You can use the **virt-install** command to create a new virtual machine with PXE booting enabled, as shown in the following example command:

```
virt-install --pxe --network bridge=breth0 --prompt
```

Alternatively, ensure that the guest network is configured to use your bridged network, and that the XML guest configuration file has a **<boot dev='network'/>** element inside the **<os>** element, as shown in the following example:

```
<os>
   <type arch='x86_64' machine='pc-i440fx-rhel7.0.0'>hvm</type>
   <boot dev='network'/>
   <boot dev='hd'/>
</os>
<interface type='bridge'>
   <mac address='52:54:00:5a:ad:cb'/>
   <source bridge='breth0'/>
   <target dev='vnet0'/>
   <alias name='net0'/>
   <address type='pci' domain='0x0000' bus='0x00' slot='0x03'
function='0x0'/>
</interface>
```

9.2.2. Using a Private libvirt Network

Procedure 9.3. Using a private libvirt network

1. Configure PXE booting on libvirt as shown in Section 9.1.1, “Setting up a PXE Boot Server on a Private libvirt Network”.

2. Boot a guest virtual machine using libvirt with PXE booting enabled. You can use the **virt-install** command to create/install a new virtual machine using PXE:

```
virt-install --pxe --network network=default --prompt
```

Alternatively, ensure that the guest network is configured to use your private libvirt network, and that the XML guest configuration file has a **<boot dev='network'/>** element inside the **<os>** element, as shown in the following example:

```
<os>
   <type arch='x86_64' machine='pc-i440fx-rhel7.0.0'>hvm</type>
   <boot dev='network'/>
   <boot dev='hd'/>
</os>
```

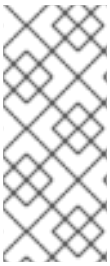
Also ensure that the guest virtual machine is connected to the private network:

```
<interface type='network'>
   <mac address='52:54:00:66:79:14'/>
   <source network='default'/>
   <target dev='vnet0'/>
   <alias name='net0'/>
   <address type='pci' domain='0x0000' bus='0x00' slot='0x03'
function='0x0'/>
</interface>
```

CHAPTER 10. REGISTERING THE HYPERVISOR AND VIRTUAL MACHINE

Red Hat Enterprise Linux 6 and 7 require that every guest virtual machine is mapped to a specific hypervisor in order to ensure that every guest is allocated the same level of subscription service. To do this you need to install a subscription agent that automatically detects all guest Virtual Machines (VMs) on each KVM hypervisor that is installed and registered, which in turn will create a mapping file that sits on the host. This mapping file ensures that all guest VMs receive the following benefits:

- Subscriptions specific to virtual systems are readily available and can be applied to all of the associated guest VMs.
- All subscription benefits that can be inherited from the hypervisor are readily available and can be applied to all of the associated guest VMs.



NOTE

The information provided in this chapter is specific to Red Hat Enterprise Linux subscriptions only. If you also have a Red Hat Virtualization subscription, or a Red Hat Satellite subscription, you should also consult the virt-who information provided with those subscriptions. More information on Red Hat Subscription Management can also be found in the Red Hat Subscription Management Guide found on the customer portal.

10.1. INSTALLING VIRT-WHO ON THE HOST PHYSICAL MACHINE

1. **Register the KVM hypervisor**
   Register the KVM Hypervisor by running the **`subscription-manager register [options]`** command in a terminal as the root user on the host physical machine. More options are available using the # **`subscription-manager register --help`** menu. In cases where you are using a user name and password, use the credentials that are known to the **Subscription Manager** application. If this is your very first time subscribing and you do not have a user account, contact customer support. For example to register the VM as 'admin' with 'secret' as a password, you would send the following command:

   ```
   [root@rhel-server ~]# subscription-manager register --username=admin
   --password=secret --auto-attach --type=hypervisor
   ```

2. **Install the virt-who packages**
   Install the virt-who packages, by running the following command on the host physical machine:

   ```
   # yum install virt-who
   ```

3. **Create a virt-who configuration file**
   For each hypervisor, add a configuration file in the **`/etc/virt-who.d/`** directory. At a minimum, the file must contain the following snippet:

   ```
   [libvirt]
   type=libvirt
   ```

   For more detailed information on configuring **`virt-who`**, see Section 10.1.1, “Configuring **`virt-who`**”.

4. **Start the virt-who service**
   Start the virt-who service by running the following command on the host physical machine:

   ```
   # systemctl start virt-who.service
   # systemctl enable virt-who.service
   ```

5. **Confirm virt-who service is receiving guest information**
   At this point, the virt-who service will start collecting a list of domains from the host. Check the **/var/log/rhsm/rhsm.log** file on the host physical machine to confirm that the file contains a list of the guest VMs. For example:

   ```
   2015-05-28 12:33:31,424 DEBUG: Libvirt domains found: [{'guestId':
   '58d59128-cfbb-4f2c-93de-230307db2ce0', 'attributes': {'active': 0,
   'virtWhoType': 'libvirt', 'hypervisorType': 'QEMU'}, 'state': 5}]
   ```

Procedure 10.1. Managing the subscription on the customer portal

1. **Subscribing the hypervisor**
   As the virtual machines will be receiving the same subscription benefits as the hypervisor, it is important that the hypervisor has a valid subscription and that the subscription is available for the VMs to use.

   a. **Log in to the Customer Portal**
      Provide your Red Hat account credentials at the Red Hat Customer Portal to log in.

   b. **Click the Systems link**
      Go to the Systems section of the **My Subscriptions** interface.

   c. **Select the hypervisor**
      On the Systems page, there is a table of all subscribed systems. Click the name of the hypervisor (for example **localhost.localdomain**). In the details page that opens, click **Attach a subscription** and select all the subscriptions listed. Click **Attach Selected**. This will attach the host's physical subscription to the hypervisor so that the guests can benefit from the subscription.

2. **Subscribing the guest virtual machines - first time use**
   This step is for those who have a new subscription and have never subscribed a guest virtual machine before. If you are adding virtual machines, skip this step. To consume the subscription assigned to the hypervisor profile on the machine running the virt-who service, auto subscribe by running the following command in a terminal on the guest virtual machine.

   ```
   [root@virt-who ~]# subscription-manager attach --auto
   ```

3. **Subscribing additional guest virtual machines**
   If you just subscribed a virtual machine for the first time, skip this step. If you are adding additional virtual machines, note that running this command will not necessarily re-attach the same subscriptions to the guest virtual machine. This is because removing all subscriptions then allowing auto-attach to resolve what is necessary for a given guest virtual machine may result in different subscriptions consumed than before. This may not have any effect on your system, but it is something you should be aware about. If you used a manual attachment procedure to attach the virtual machine, which is not described below, you will need to re-attach those virtual

machines manually as the auto-attach will not work. Use the following command to first remove the subscriptions for the old guests, and then use the auto-attach to attach subscriptions to all the guests. Run these commands on the guest virtual machine.

```
[root@virt-who ~]# subscription-manager remove --all
[root@virt-who ~]# subscription-manager attach --auto
```

4. **Confirm subscriptions are attached**
   Confirm that the subscription is attached to the hypervisor by running the following command on the guest virtual machine:

```
[root@virt-who ~]# subscription-manager list --consumed
```

Output similar to the following will be displayed. Pay attention to the Subscription Details. It should say 'Subscription is current'.

```
[root@virt-who ~]#  subscription-manager list --consumed
+-------------------------------------------+
   Consumed Subscriptions
+-------------------------------------------+
Subscription Name: Awesome OS with unlimited virtual guests
Provides:   Awesome OS Server Bits
SKU:    awesomeos-virt-unlimited
Contract:   0
Account:   ######### Your account number #####
Serial:   ######### Your serial number ######
Pool ID:   XYZ123
```

```
Provides Management:  No
Active:   True
Quantity Used:   1
Service Level:
Service Type:
Status Details:  Subscription is current
```

```
Subscription Type:
Starts:   01/01/2015
Ends:    12/31/2015
System Type:   Virtual
```

1	The ID for the subscription to attach to the system is displayed here. You will need this ID if you need to attach the subscription manually.
2	Indicates if your subscription is current. If your subscription is not current, an error message appears. One example is Guest has not been reported on any host and is using a temporary unmapped guest subscription. In this case the guest needs to be subscribed. In other cases, use the information as indicated in Section 10.5.2, “I have subscription status errors, what do I do?” .

5. **Register additional guests**

When you install new guest VMs on the hypervisor, you must register the new VM and use the subscription attached to the hypervisor, by running the following commands on the guest virtual machine:

```
# subscription-manager register
# subscription-manager attach --auto
# subscription-manager list --consumed
```

10.1.1. Configuring `virt-who`

The **`virt-who`** service is configured using the following files:

- **`/etc/virt-who.conf`** - Contains general configuration information including the interval for checking connected hypervisors for changes.
- **`/etc/virt-who.d/`*`hypervisor_name`*`.conf`** - Contains configuration information for a specific hypervisor.

A web-based wizard is provided to generate hypervisor configuration files and the snippets required for **`virt-who.conf`**. To run the wizard, browse to Red Hat Virtualization Agent (virt-who) Configuration Helper on the Customer Portal.

On the second page of the wizard, select the following options:

- **Where does your virt-who report to?** : **`Subscription Asset Manager`**
- **Hypervisor Type**: **`libvirt`**

Follow the wizard to complete the configuration. If the configuration is performed correctly, **`virt-who`** will automatically provide the selected subscriptions to existing and future guests on the specified hypervisor.

For more information on hypervisor configuration files, see the **`virt-who-config`** man page.

10.2. REGISTERING A NEW GUEST VIRTUAL MACHINE

In cases where a new guest virtual machine is to be created on a host that is already registered and running, the virt-who service must also be running. This ensures that the virt-who service maps the guest to a hypervisor, so the system is properly registered as a virtual system. To register the virtual machine, enter the following command:

```
[root@virt-server ~]# subscription-manager register --username=admin --
password=secret --auto-attach
```

10.3. REMOVING A GUEST VIRTUAL MACHINE ENTRY

If the guest virtual machine is running, unregister the system, by running the following command in a terminal window as root on the guest:

```
[root@virt-guest ~]# subscription-manager unregister
```

If the system has been deleted, however, the virtual service cannot tell whether the service is deleted or paused. In that case, you must manually remove the system from the server side, using the following steps:

1. **Login to the Subscription Manager**
   The Subscription Manager is located on the Red Hat Customer Portal. Login to the Customer Portal using your user name and password, by clicking the login icon at the top of the screen.

2. **Click the Subscriptions tab**
   Click the **Subscriptions** tab.

3. **Click the Systems link**
   Scroll down the page and click the **Systems** link.

4. **Delete the system**
   To delete the system profile, locate the specified system's profile in the table, select the check box beside its name and click **Delete**.

10.4. INSTALLING VIRT-WHO MANUALLY

This section will describe how to manually attach the subscription provided by the hypervisor.

Procedure 10.2. How to attach a subscription manually

1. **List subscription information and find the Pool ID**
   First you need to list the available subscriptions which are of the virtual type. Enter the following command:

```
[root@server1 ~]# subscription-manager list --avail --match-
installed | grep 'Virtual' -B12
Subscription Name: Red Hat Enterprise Linux ES (Basic for
Virtualization)
Provides:          Red Hat Beta
                   Oracle Java (for RHEL Server)
                   Red Hat Enterprise Linux Server
SKU:               -------
Pool ID:           XYZ123
Available:         40
Suggested:         1
Service Level:     Basic
Service Type:      L1-L3
Multi-Entitlement: No
Ends:              01/02/2017
System Type:       Virtual
```

   Note the Pool ID displayed. Copy this ID as you will need it in the next step.

2. **Attach the subscription with the Pool ID**
   Using the Pool ID you copied in the previous step run the attach command. Replace the Pool ID *XYZ123* with the Pool ID you retrieved. Enter the following command:

```
[root@server1 ~]# subscription-manager attach --pool=XYZ123

Successfully attached a subscription for: Red Hat Enterprise Linux
ES (Basic for Virtualization)
```

10.5. TROUBLESHOOTING VIRT-WHO

10.5.1. Why is the hypervisor status red?

Scenario: On the server side, you deploy a guest on a hypervisor that does not have a subscription. 24 hours later, the hypervisor displays its status as red. To remedy this situation you must get a subscription for that hypervisor. Or, permanently migrate the guest to a hypervisor with a subscription.

10.5.2. I have subscription status errors, what do I do?

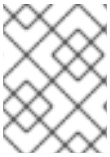
Scenario: Any of the following error messages display:

- System not properly subscribed
- Status unknown
- Late binding of a guest to a hypervisor through virt-who (host/guest mapping)

To find the reason for the error open the virt-who log file, named **`rhsm.log`**, located in the **`/var/log/rhsm/`** directory.

CHAPTER 11. ENHANCING VIRTUALIZATION WITH THE QEMU GUEST AGENT AND SPICE AGENT

Agents in Red Hat Enterprise Linux such as the QEMU guest agent and the SPICE agent can be deployed to help the virtualization tools run more optimally on your system. These agents are described in this chapter.



NOTE

To further optimize and tune host and guest performance, see the Red Hat Enterprise Linux 7 Virtualization Tuning and Optimization Guide.

11.1. QEMU GUEST AGENT

The QEMU guest agent runs inside the guest and allows the host machine to issue commands to the guest operating system using libvirt, helping with functions such as freezing and thawing filesystems. The guest operating system then responds to those commands asynchronously. The QEMU guest agent package, qemu-guest-agent, is installed by default in Red Hat Enterprise Linux 7.

This section covers the libvirt commands and options available to the guest agent.

IMPORTANT

Note that it is only safe to rely on the QEMU guest agent when run by trusted guests. An untrusted guest may maliciously ignore or abuse the guest agent protocol, and although built-in safeguards exist to prevent a denial of service attack on the host, the host requires guest co-operation for operations to run as expected.

Note that QEMU guest agent can be used to enable and disable virtual CPUs (vCPUs) while the guest is running, thus adjusting the number of vCPUs without using the hot plug and hot unplug features. For more information, see Section 20.36.6, “Configuring Virtual CPU Count”.

11.1.1. Setting up Communication between the QEMU Guest Agent and Host

The host machine communicates with the QEMU guest agent through a VirtIO serial connection between the host and guest machines. A VirtIO serial channel is connected to the host via a character device driver (typically a Unix socket), and the guest listens on this serial channel.

NOTE

The qemu-guest-agent does not detect if the host is listening to the VirtIO serial channel. However, as the current use for this channel is to listen for host-to-guest events, the probability of a guest virtual machine running into problems by writing to the channel with no listener is very low. Additionally, the qemu-guest-agent protocol includes synchronization markers that allow the host physical machine to force a guest virtual machine back into sync when issuing a command, and libvirt already uses these markers, so that guest virtual machines are able to safely discard any earlier pending undelivered responses.

11.1.1.1. Configuring the QEMU Guest Agent on a Linux Guest

The QEMU guest agent can be configured on a running or shut down virtual machine. If configured on a running guest, the guest will start using the guest agent immediately. If the guest is shut down, the QEMU guest agent will be enabled at next boot.

Either **virsh** or **virt-manager** can be used to configure communication between the guest and the QEMU guest agent. The following instructions describe how to configure the QEMU guest agent on a Linux guest.

Procedure 11.1. Setting up communication between guest agent and host with virsh on a shut down Linux guest

1. **Shut down the virtual machine**
   Ensure the virtual machine (named *rhel7* in this example) is shut down before configuring the QEMU guest agent:

   ```
   # virsh shutdown rhel7
   ```

2. **Add the QEMU guest agent channel to the guest XML configuration**
   Edit the guest's XML file to add the QEMU guest agent details:

   ```
   # virsh edit rhel7
   ```

   Add the following to the guest's XML file and save the changes:

   ```
   <channel type='unix'>
      <target type='virtio' name='org.qemu.guest_agent.0'/>
   </channel>
   ```

3. **Start the virtual machine**

   ```
   # virsh start rhel7
   ```

4. **Install the QEMU guest agent on the guest**
   Install the QEMU guest agent if not yet installed in the guest virtual machine:

   ```
   # yum install qemu-guest-agent
   ```

5. **Start the QEMU guest agent in the guest**
   Start the QEMU guest agent service in the guest:

   ```
   # systemctl start qemu-guest-agent
   ```

Alternatively, the QEMU guest agent can be configured on a running guest with the following steps:

Procedure 11.2. Setting up communication between guest agent and host on a running Linux guest

1. **Create an XML file for the QEMU guest agent**

   ```
   # cat agent.xml
   <channel type='unix'>
      <target type='virtio' name='org.qemu.guest_agent.0'/>
   ```

```
</channel>
```

2. **Attach the QEMU guest agent to the virtual machine**
   Attach the QEMU guest agent to the running virtual machine (named *rhel7* in this example) with this command:

   ```
   # virsh attach-device rhel7 agent.xml
   ```

3. **Install the QEMU guest agent on the guest**
   Install the QEMU guest agent if not yet installed in the guest virtual machine:

   ```
   # yum install qemu-guest-agent
   ```

4. **Start the QEMU guest agent in the guest**
   Start the QEMU guest agent service in the guest:

   ```
   # systemctl start qemu-guest-agent
   ```

Procedure 11.3. Setting up communication between the QEMU guest agent and host with `virt-manager`

1. **Shut down the virtual machine**
   Ensure the virtual machine is shut down before configuring the QEMU guest agent.

   To shut down the virtual machine, select it from the list of virtual machines in **Virtual Machine Manager**, then click the light switch icon from the menu bar.

2. **Add the QEMU guest agent channel to the guest**
   Open the virtual machine's hardware details by clicking the lightbulb icon at the top of the guest window.

   Click the **`Add Hardware`** button to open the **`Add New Virtual Hardware`** window, and select **`Channel`**.

   Select the QEMU guest agent from the **`Name`** drop-down list and click **`Finish`**:

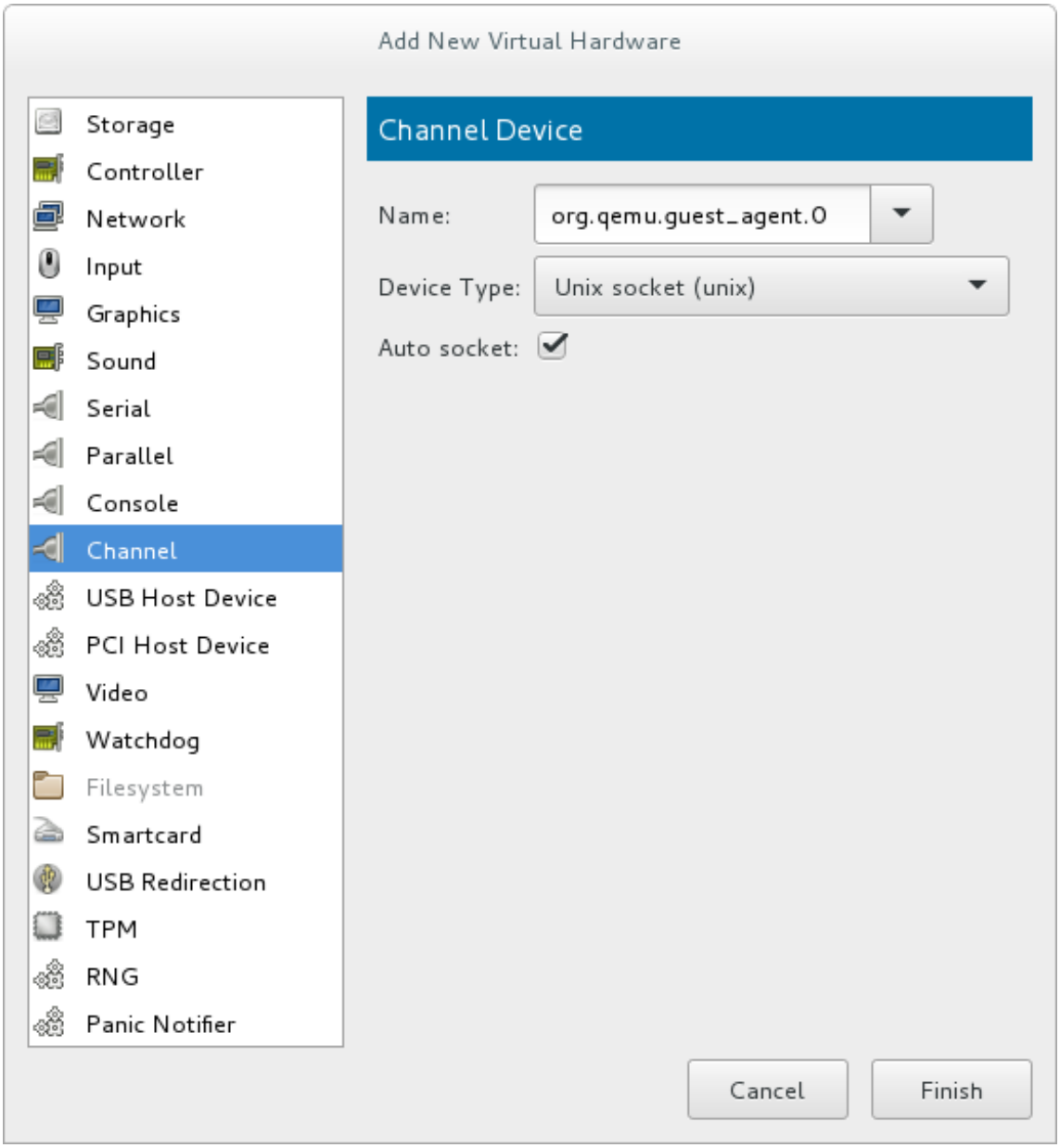


Figure 11.1. Selecting the QEMU guest agent channel device

3. **Start the virtual machine**
   To start the virtual machine, select it from the list of virtual machines in **Virtual Machine Manager**, then click on the menu bar.

4. **Install the QEMU guest agent on the guest**
   Open the guest with **virt-manager** and install the QEMU guest agent if not yet installed in the guest virtual machine:

   ```
   # yum install qemu-guest-agent
   ```

5. **Start the QEMU guest agent in the guest**
   Start the QEMU guest agent service in the guest:

   ```
   # systemctl start qemu-guest-agent
   ```

The QEMU guest agent is now configured on the *rhel7* virtual machine.

11.2. USING THE QEMU GUEST AGENT WITH LIBVIRT

Installing the QEMU guest agent allows various libvirt commands to become more powerful. The guest agent enhances the following **virsh** commands:

- **virsh shutdown --mode=agent** - This shutdown method is more reliable than **virsh shutdown --mode=acpi**, as **virsh shutdown** used with the QEMU guest agent is guaranteed to shut down a cooperative guest in a clean state. If the agent is not present, libvirt must instead rely on injecting an ACPI shutdown event, but some guests ignore that event and thus will not shut down.

  Can be used with the same syntax for **virsh reboot**.

- **virsh snapshot-create --quiesce** - Allows the guest to flush its I/O into a stable state before the snapshot is created, which allows use of the snapshot without having to perform a fsck or losing partial database transactions. The guest agent allows a high level of disk contents stability by providing guest co-operation.

- **virsh domfsfreeze** and **virsh domfsthaw** - Quiesces the guest filesystem in isolation.

- **virsh domfstrim** - Instructs the guest to trim its filesystem.

- **virsh domtime** - Queries or sets the guest's clock.

- **virsh setvcpus --guest** - Instructs the guest to take CPUs offline.

- **virsh domifaddr --source agent** - Queries the guest operating system's IP address via the guest agent.

- **virsh domfsinfo** - Shows a list of mounted filesystems within the running guest.

- **virsh set-user-password** - Sets the password for a user account in the guest.

11.2.1. Creating a Guest Disk Backup

libvirt can communicate with qemu-guest-agent to ensure that snapshots of guest virtual machine file systems are consistent internally and ready to use as needed. Guest system administrators can write and install application-specific freeze/thaw hook scripts. Before freezing the filesystems, the qemu-guest-agent invokes the main hook script (included in the qemu-guest-agent package). The freezing process temporarily deactivates all guest virtual machine applications.

The snapshot process is comprised of the following steps:

- File system applications / databases flush working buffers to the virtual disk and stop accepting client connections

- Applications bring their data files into a consistent state

- Main hook script returns

- qemu-guest-agent freezes the filesystems and the management stack takes a snapshot

- Snapshot is confirmed

- Filesystem function resumes

Thawing happens in reverse order.

To create a snapshot of the guest's file system, run the **virsh snapshot-create --quiesce --disk-only** command (alternatively, run **virsh snapshot-create-as *guest_name* --quiesce --disk-only**, explained in further detail in Section 20.39.2, "Creating a Snapshot for the Current Guest Virtual Machine").

NOTE

An application-specific hook script might need various SELinux permissions in order to run correctly, as is done when the script needs to connect to a socket in order to talk to a database. In general, local SELinux policies should be developed and installed for such purposes. Accessing file system nodes should work out of the box, after issuing the **restorecon -FvvR** command listed in Table 11.1, "QEMU guest agent package contents" in the table row labeled **/etc/qemu-ga/fsfreeze-hook.d/**.

The qemu-guest-agent binary RPM includes the following files:

Table 11.1. QEMU guest agent package contents

File name	Description
/usr/lib/systemd/system/qemu-guest-agent.service	Service control script (start/stop) for the QEMU guest agent.
/etc/sysconfig/qemu-ga	Configuration file for the QEMU guest agent, as it is read by the **/usr/lib/systemd/system/qemu-guest-agent.service** control script. The settings are documented in the file with shell script comments.
/usr/bin/qemu-ga	QEMU guest agent binary file.
/etc/qemu-ga	Root directory for hook scripts.
/etc/qemu-ga/fsfreeze-hook	Main hook script. No modifications are needed here.
/etc/qemu-ga/fsfreeze-hook.d	Directory for individual, application-specific hook scripts. The guest system administrator should copy hook scripts manually into this directory, ensure proper file mode bits for them, and then run **restorecon -FvvR** on this directory.
/usr/share/qemu-kvm/qemu-ga/	Directory with sample scripts (for example purposes only). The scripts contained here are not executed.

The main hook script, **/etc/qemu-ga/fsfreeze-hook** logs its own messages, as well as the application-specific script's standard output and error messages, in the following log file: **/var/log/qemu-ga/fsfreeze-hook.log**. For more information, see the libvirt upstream website.

11.3. SPICE AGENT

The SPICE agent helps run graphical applications such as **virt-manager** more smoothly, by helping integrate the guest operating system with the SPICE client.

For example, when resizing a window in **virt-manager**, the SPICE agent allows for automatic X session resolution adjustment to the client resolution. The SPICE agent also provides support for copy and paste between the host and guest, and prevents mouse cursor lag.

For system-specific information on the SPICE agent's capabilities, see the spice-vdagent package's README file.

11.3.1. Setting up Communication between the SPICE Agent and Host

The SPICE agent can be configured on a running or shut down virtual machine. If configured on a running guest, the guest will start using the guest agent immediately. If the guest is shut down, the SPICE agent will be enabled at next boot.

Either **virsh** or **virt-manager** can be used to configure communication between the guest and the SPICE agent. The following instructions describe how to configure the SPICE agent on a Linux guest.

Procedure 11.4. Setting up communication between guest agent and host with virsh on a Linux guest

1. **Shut down the virtual machine**
   Ensure the virtual machine (named *rhel7* in this example) is shut down before configuring the SPICE agent:

   ```
   # virsh shutdown rhel7
   ```

2. **Add the SPICE agent channel to the guest XML configuration**
   Edit the guest's XML file to add the SPICE agent details:

   ```
   # virsh edit rhel7
   ```

   Add the following to the guest's XML file and save the changes:

   ```
   <channel type='spicevmc'>
      <target type='virtio' name='com.redhat.spice.0'/>
   </channel>
   ```

3. **Start the virtual machine**

   ```
   # virsh start rhel7
   ```

4. **Install the SPICE agent on the guest**
   Install the SPICE agent if not yet installed in the guest virtual machine:

   ```
   # yum install spice-vdagent
   ```

5. **Start the SPICE agent in the guest**
   Start the SPICE agent service in the guest:

```
# systemctl start spice-vdagent
```

Alternatively, the SPICE agent can be configured on a running guest with the following steps:

Procedure 11.5. Setting up communication between SPICE agent and host on a running Linux guest

1. **Create an XML file for the SPICE agent**

```
# cat agent.xml
<channel type='spicevmc'>
   <target type='virtio' name='com.redhat.spice.0'/>
</channel>
```

2. **Attach the SPICE agent to the virtual machine**
   Attach the SPICE agent to the running virtual machine (named *rhel7* in this example) with this command:

```
# virsh attach-device rhel7 agent.xml
```

3. **Install the SPICE agent on the guest**
   Install the SPICE agent if not yet installed in the guest virtual machine:

```
# yum install spice-vdagent
```

4. **Start the SPICE agent in the guest**
   Start the SPICE agent service in the guest:

```
# systemctl start spice-vdagent
```

Procedure 11.6. Setting up communication between the SPICE agent and host with `virt-manager`

1. **Shut down the virtual machine**
   Ensure the virtual machine is shut down before configuring the SPICE agent.

   To shut down the virtual machine, select it from the list of virtual machines in **Virtual Machine Manager**, then click the light switch icon from the menu bar.

2. **Add the SPICE agent channel to the guest**
   Open the virtual machine's hardware details by clicking the lightbulb icon at the top of the guest window.

   Click the **`Add Hardware`** button to open the **`Add New Virtual Hardware`** window, and select **`Channel`**.

   Select the SPICE agent from the **`Name`** drop-down list, edit the channel address, and click **`Finish`**:

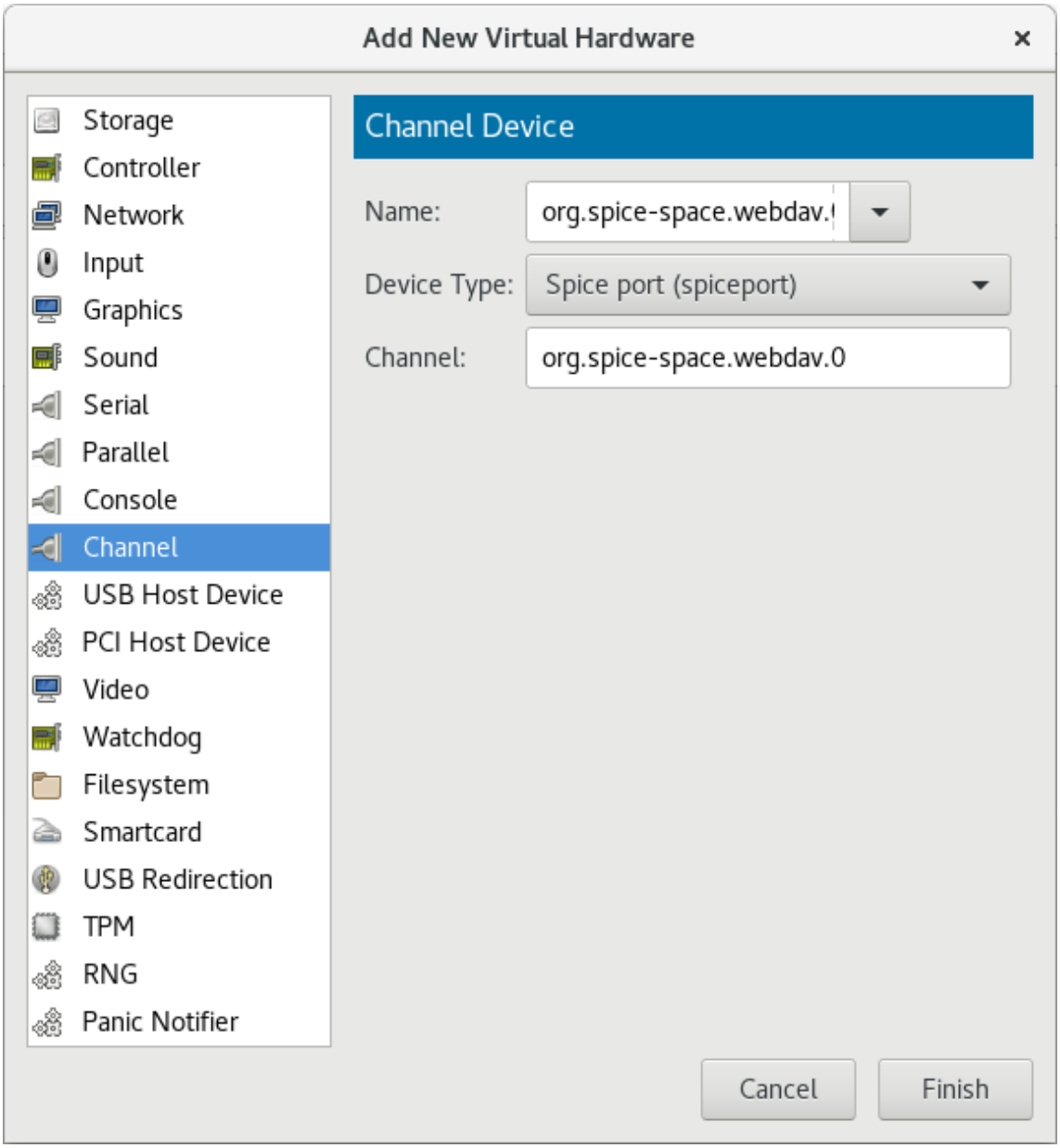


Figure 11.2. Selecting the SPICE agent channel device

3. **Start the virtual machine**
   To start the virtual machine, select it from the list of virtual machines in **Virtual Machine Manager**, then click on the menu bar.

4. **Install the SPICE agent on the guest**
   Open the guest with **virt-manager** and install the SPICE agent if not yet installed in the guest virtual machine:

   ```
   # yum install spice-vdagent
   ```

5. **Start the SPICE agent in the guest**
   Start the SPICE agent service in the guest:

   ```
   # systemctl start spice-vdagent
   ```

The SPICE agent is now configured on the *rhel7* virtual machine.

CHAPTER 12. NESTED VIRTUALIZATION

12.1. OVERVIEW

As of Red Hat Enterprise Linux 7.5, *nested virtualization* is available as a Technology Preview for KVM guest virtual machines. With this feature, a guest virtual machine (also referred to as *level 1* or *L1*) that runs on a physical host (*level 0* or *L0*) can act as a hypervisor, and create its own guest virtual machines (*L2*).

Nested virtualization is useful in a variety of scenarios, such as debugging hypervisors in a constrained environment and testing larger virtual deployments on a limited amount of physical resources. However, note that nested virtualization is not supported or recommended in production user environments, and is primarily intended for development and testing.

Nested virtualization relies on host virtualization extensions to function, and it should not to be confused with running guests in a virtual environment using the QEMU Tiny Code Generator (TCG) emulation, which is not supported in Red Hat Enterprise Linux.

12.2. SETUP

Follow these steps to enable, configure, and start using nested virtualization:

1. **Enable**: The feature is disabled by default. To enable it, use the following procedure on the L0 host physical machine.

   For Intel:

   1. Check whether nested virtualization is available on your host system.

      ```
      $ cat /sys/module/kvm_intel/parameters/nested
      ```

      If this command returns **`Y`** or **`1`**, the feature is enabled.

      If the command returns **`0`** or **`N`**, use steps *b* and *c*.

   2. Unload the **`kvm_intel`** module:

      ```
      # modprobe -r kvm_intel
      ```

   3. Activate the nesting feature:

      ```
      # modprobe kvm_intel nested=1
      ```

   4. The nesting feature is now enabled only until the next reboot of the L0 host. To enable it permanently, add the following line to the **`/etc/modprobe.d/kvm.conf`** file:

      ```
      options kvm_intel nested=1
      ```

   For AMD:

   1. Check whether nested virtualization is available on your system:

```
$ cat /sys/module/kvm_amd/parameters/nested
```

If this command returns **Y** or **1**, the feature is enabled.

If the command returns **0** or **N**, use steps *b* and *c*.

2. Unload the **kvm_amd** module

```
# modprobe -r kvm_amd
```

3. Activate the nesting feature

```
# modprobe kvm_amd nested=1
```

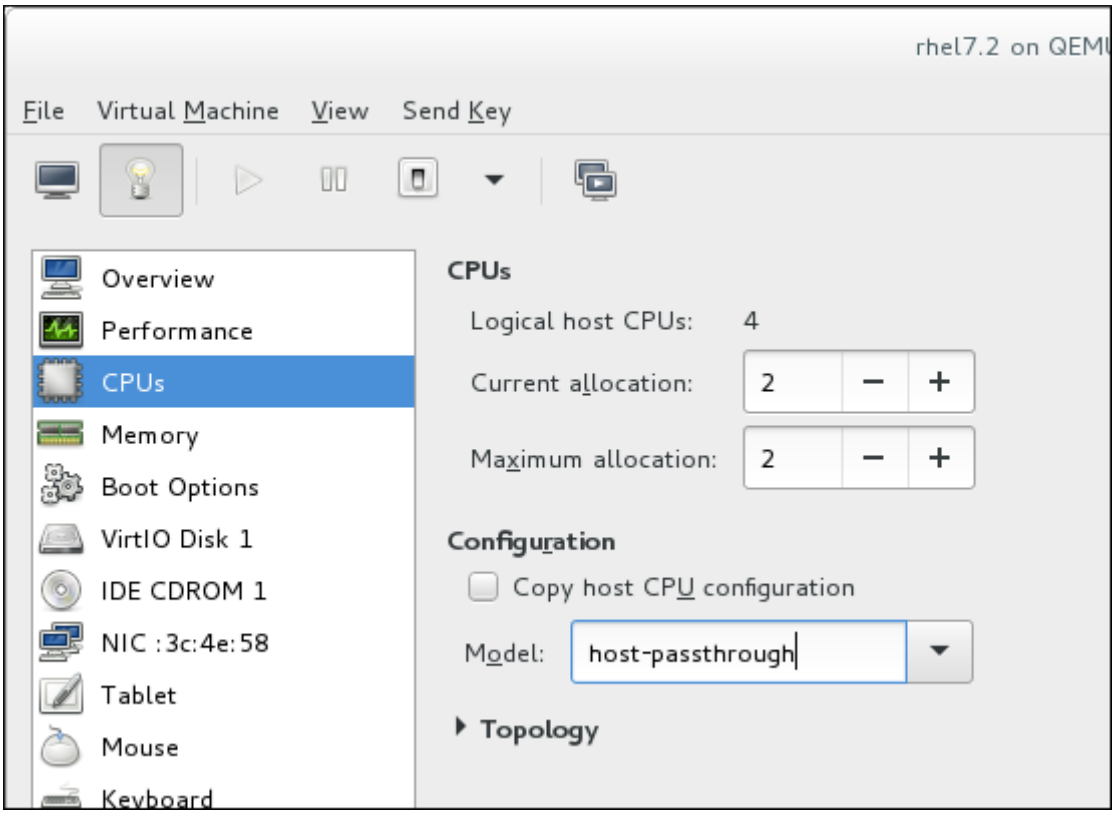
4. The nesting feature is now enabled only until the next reboot of the L0 host. To enable it permanently, add the following line to the **/etc/modprobe.d/kvm.conf** file:

```
options kvm_amd nested=1
```

2. **Configure** your L1 virtual machine for nested virtualization using one of the following methods:

virt-manager

1. Open the GUI of the intended guest and click the *Show Virtual Hardware Details* icon.
2. Select the *Processor* menu, and in the *Configuration* section, type **host-passthrough** in the *Model* field (do not use the drop-down selection), and click *Apply*.



Domain XML

Add the following line to the domain XML file of the guest:

```
<cpu mode='host-passthrough'/>
```

If the guest's XML configuration file already contains a **<cpu>** element, rewrite it.

3. To **start using** nested virtualization, install an L2 guest within the L1 guest. To do this, follow the same procedure as when installing the L1 guest - see Chapter 3, *Creating a Virtual Machine* for more information.

12.3. RESTRICTIONS AND LIMITATIONS

- It is strongly recommended to run Red Hat Enterprise Linux 7.2 or later in the L0 host and the L1 guests. L2 guests can contain any guest system supported by Red Hat.
- It is not supported to migrate L1 or L2 guests.
- Use of L2 guests as hypervisors and creating L3 guests is not supported.
- Not all features available on the host are available to be utilized by the L1 hypervisor. For instance, IOMMU/VT-d or APICv cannot be used by the L1 hypervisor.
- To use nested virtualization, the host CPU must have the necessary feature flags. To determine if the L0 and L1 hypervisors are set up correctly, use the **`cat /proc/cpuinfo`** command on both L0 and L1, and make sure that the following flags are listed for the respective CPUs on both hypervisors:
  - For Intel - **vmx** (Hardware Virtualization) and **ept** (Extended Page Tables)
  - For AMD - **svm** (equivalent to vmx) and **npt** (equivalent to ept)

PART II. ADMINISTRATION

CHAPTER 13. MANAGING STORAGE FOR VIRTUAL MACHINES

This chapter provides information about storage for virtual machines. Virtual storage is abstracted from the physical storage allocated to a virtual machine connection. The storage is attached to the virtual machine using paravirtualized or emulated block device drivers.

13.1. STORAGE CONCEPTS

A *storage pool* is a quantity of storage set aside for use by guest virtual machines. Storage pools are divided into *storage volumes*. Each storage volume is assigned to a guest virtual machine as a block device on a guest bus.

Storage pools and volumes are managed using libvirt. With libvirt's remote protocol, it is possible to manage all aspects of a guest virtual machine's life cycle, as well as the configuration of the resources required by the guest virtual machine. These operations can be performed on a remote host. As a result, a management application, such as the **Virtual Machine Manager**, using libvirt can enable a user to perform all the required tasks for configuring the host physical machine for a guest virtual machine. These include allocating resources, running the guest virtual machine, shutting it down, and de-allocating the resources, without requiring shell access or any other control channel.

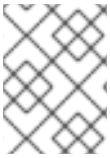
The libvirt API can be used to query the list of volumes in the storage pool or to get information regarding the capacity, allocation, and available storage in the storage pool. A storage volume in the storage pool may be queried to get information such as allocation and capacity, which may differ for sparse volumes.

NOTE

For more information about sparse volumes, see the Virtualization Getting Started Guide.

For storage pools that support it, the libvirt API can be used to create, clone, resize, and delete storage volumes. The APIs can also be used to upload data to storage volumes, download data from storage volumes, or wipe data from storage volumes.

Once a storage pool is started, a storage volume can be assigned to a guest using the storage pool name and storage volume name instead of the host path to the volume in the domain XML.



NOTE

For more information about the domain XML, see Chapter 23, *Manipulating the Domain XML*.

Storage pools can be stopped (destroyed). This removes the abstraction of the data, but keeps the data intact.

For example, an NFS server that uses **`mount -t nfs nfs.example.com:/path/to/share /path/to/data`**. A storage administrator responsible could define an NFS Storage Pool on the virtualization host to describe the exported server path and the client target path. This will allow libvirt to perform the mount either automatically when libvirt is started or as needed while libvirt is running. Files with the NFS Server exported directory are listed as storage volumes within the NFS storage pool.

When the storage volume is added to the guest, the administrator does not need to add the target path to the volume. He just needs to add the storage pool and storage volume by name. Therefore, if the target client path changes, it does not affect the virtual machine.

When the storage pool is started, libvirt mounts the share on the specified directory, just as if the system administrator logged in and executed **`mount nfs.example.com:/path/to/share /vmdata`**. If the storage pool is configured to autostart, libvirt ensures that the NFS shared disk is mounted on the directory specified when libvirt is started.

Once the storage pool is started, the files in the NFS shared disk are reported as storage volumes, and the storage volumes' paths may be queried using the libvirt API. The storage volumes' paths can then be copied into the section of a guest virtual machine's XML definition that describes the source storage for the guest virtual machine's block devices. In the case of NFS, an application that uses the libvirt API can create and delete storage volumes in the storage pool (files in the NFS share) up to the limit of the size of the pool (the storage capacity of the share).

Not all storage pool types support creating and deleting volumes. Stopping the storage pool (pool-destroy) undoes the start operation, in this case, unmounting the NFS share. The data on the share is not modified by the destroy operation, despite what the name of the command suggests. For more details, see **`man virsh`**.

Procedure 13.1. Creating and Assigning Storage

This procedure provides a high-level understanding of the steps needed to create and assign storage for virtual machine guests.

1. **Create storage pools**
   Create one or more storage pools from available storage media. For more information, see Section 13.2, “Using Storage Pools”.

2. **Create storage volumes**
   Create one or more storage volumes from the available storage pools. For more information, see Section 13.3, “Using Storage Volumes”.

3. **Assign storage devices to a virtual machine.**
   Assign one or more storage devices abstracted from storage volumes to a guest virtual machine. For more information, see Section 13.3.6, “Adding Storage Devices to Guests”.

13.2. USING STORAGE POOLS

This section provides information about using storage pools with virtual machines. It provides conceptual information, as well as detailed instructions on creating, configuring, and deleting storage pools using **`virsh`** commands and the **Virtual Machine Manager**.

13.2.1. Storage Pool Concepts

A storage pool is a file, directory, or storage device, managed by libvirt to provide storage to virtual machines. Storage pools are divided into storage volumes that store virtual machine images or are attached to virtual machines as additional storage. Multiple guests can share the same storage pool, allowing for better allocation of storage resources.

Storage pools can be either local or network-based (shared):

Local storage pools

Local storage pools are attached directly to the host server. They include local directories, directly attached disks, physical partitions, and Logical Volume Management (LVM) volume groups on local devices. Local storage pools are useful for development, testing, and small deployments that do not require migration or large numbers of virtual machines. Local storage pools may not be suitable for many production environments, because they cannot be used for live migration.

Networked (shared) storage pools

Networked storage pools include storage devices shared over a network using standard protocols. Networked storage is required when migrating virtual machines between hosts with **virt-manager**, but is optional when migrating with **virsh**.

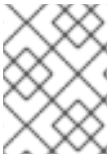
For more information on migrating virtual machines, see Chapter 15, *KVM Migration*.

The following is a list of storage pool types supported by Red Hat Enterprise Linux:

- Directory-based storage pools
- Disk-based storage pools
- Partition-based storage pools
- GlusterFS storage pools
- iSCSI-based storage pools
- LVM-based storage pools
- NFS-based storage pools
- vHBA-based storage pools with SCSI devices

The following is a list of **libvirt** storage pool types that are not supported by Red Hat Enterprise Linux:

- Multipath-based storage pool
- RBD-based storage pool
- Sheepdog-based storage pool
- Vstorage-based storage pool
- ZFS-based storage pool



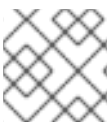
NOTE

Some of the unsupported storage pool types appear in the **Virtual Machine Manager** interface. However, they should not be used.

13.2.2. Creating Storage Pools

This section provides general instructions for creating storage pools using **virsh** and the **Virtual Machine Manager**. Using **virsh** enables you to specify all parameters, whereas using **Virtual Machine Manager** provides a graphic method for creating simpler storage pools.

13.2.2.1. Creating Storage Pools with virsh



NOTE

This section shows the creation of a partition-based storage pool as an example.

Procedure 13.2. Creating Storage Pools with virsh

1. **Read recommendations and ensure that all prerequisites are met**
   For some storage pools, this guide recommends that you follow certain practices. In addition, there are prerequisites for some types of storage pools. To see the recommendations and prerequisites, if any, see Section 13.2.3, “Storage Pool Specifics”.

2. **Define the storage pool**
   Storage pools can be persistent or transient. A persistent storage pool survives a system restart of the host machine. A transient storage pool only exists until the host reboots.

   Do one of the following:

   - Define the storage pool using an XML file.

     a. Create a temporary XML file containing the storage pool information required for the new device.

     The XML file must contain specific fields, based on the storage pool type. For more information, see Section 13.2.3, “Storage Pool Specifics”.

     The following shows an example a storage pool definition XML file. In this example, the file is saved to **~/guest_images.xml**

     ```
     <pool type='fs'>
       <name>guest_images_fs</name>
       <source>
         <device path='/dev/sdc1'/>
       </source>
       <target>
         <path>/guest_images</path>
       </target>
     </pool>
     ```

     b. Use the **virsh pool-define** command to create a persistent storage pool or the **virsh pool-create** command to create and start a transient storage pool.

     ```
     # virsh pool-define ~/guest_images.xml
     Pool defined from guest_images_fs
     ```

     or

     ```
     # virsh pool-create ~/guest_images.xml
     Pool created from guest_images_fs
     ```

     c. Delete the XML file created in step a.

   - Use the **virsh pool-define-as** command to create a persistent storage pool or the **virsh pool-create-as** command to create a transient storage pool.

     The following examples create a persistent and then a transient filesystem-based storage pool mapped to **/dev/sdc1** from the **/guest_images** directory.

```
# virsh pool-define-as guest_images_fs fs - - /dev/sdc1 -
"/guest_images"
Pool guest_images_fs defined
```

or

```
# virsh pool-create-as guest_images_fs fs - - /dev/sdc1 -
"/guest_images"
Pool guest_images_fs created
```

NOTE

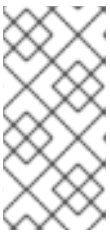
When using the **virsh** interface, option names in the commands are optional. If option names are not used, use dashes for fields that do not need to be specified.

3. **Verify that the pool was created**
   List all existing storage pools using the **virsh pool-list --all**.

```
# virsh pool-list --all
Name                 State      Autostart
-----------------------------------------
default              active     yes
guest_images_fs      inactive   no
```

4. **Define the storage pool target path**
   Use the **virsh pool-build** command to create a storage pool target path for a pre-formatted file system storage pool, initialize the storage source device, and define the format of the data. Then use the **virsh pool-list** command to ensure that the storage pool is listed.

```
# virsh pool-build guest_images_fs
Pool guest_images_fs built
# ls -la /guest_images
total 8
drwx------.  2 root root 4096 May 31 19:38 .
dr-xr-xr-x. 25 root root 4096 May 31 19:38 ..
# virsh pool-list --all
Name                 State      Autostart
-----------------------------------------
default              active     yes
guest_images_fs      inactive   no
```



NOTE

Building the target path is only necessary for disk-based, file system-based, and logical storage pools. If libvirt detects that the source storage device's data format differs from the selected storage pool type, the build fails, unless the **overwrite** option is specified.

5. **Start the storage pool**
   Use the **virsh pool-start** command to prepare the source device for usage.

The action taken depends on the storage pool type. For example, for a file system-based storage pool, the **virsh pool-start** command mounts the file system. For an LVM-based storage pool, the **virsh pool-start** command activates the volume group usng the **vgchange** command.

Then use the **virsh pool-list command** to ensure that the storage pool is active.

```
# virsh pool-start guest_images_fs
Pool guest_images_fs started
# virsh pool-list --all
Name                 State      Autostart
-----------------------------------------
default              active     yes
guest_images_fs      active     no
```

NOTE

The **virsh pool-start** command is only necessary for persistent storage pools. Transient storage pools are automatically started when they are created.

6. **Turn on autostart (optional)**
By default, a storage pool defined with **virsh** is not set to automatically start each time **libvirtd** starts. You can configure the storage pool to start automatically using the **virsh pool-autostart** command.

```
# virsh pool-autostart guest_images_fs
Pool guest_images_fs marked as autostarted

# virsh pool-list --all
Name                 State      Autostart
-----------------------------------------
default              active     yes
guest_images_fs      active     yes
```

The storage pool is now automatically started each time **libvirtd** starts.

7. **Verify the storage pool**
Verify that the storage pool was created correctly, the sizes reported are as expected, and the state is reported as **running**. Verify there is a "lost+found" directory in the target path on the file system, indicating that the device is mounted.

```
# virsh pool-info guest_images_fs
Name:           guest_images_fs
UUID:           c7466869-e82a-a66c-2187-dc9d6f0877d0
State:          running
Persistent:     yes
Autostart:      yes
Capacity:       458.39 GB
Allocation:     197.91 MB
Available:      458.20 GB
# mount | grep /guest_images
/dev/sdc1 on /guest_images type ext4 (rw)
# ls -la /guest_images
```

```
total 24
drwxr-xr-x.  3 root root  4096 May 31 19:47 .
dr-xr-xr-x. 25 root root  4096 May 31 19:38 ..
drwx------.  2 root root 16384 May 31 14:18 lost+found
```

13.2.2.2. Creating storage pools with Virtual Machine Manager

NOTE

This section shows the creation of a disk-based storage pool as an example.

Procedure 13.3. Creating Storage Pools with Virtual Machine Manager

1. **Prepare the medium on which the storage pool will be created**
   This will differ for different types of storage pools. For details, see Section 13.2.3, “Storage Pool Specifics”.

   In this example, you may need to relabel the disk with a *GUID Partition Table*.

2. **Open the storage settings**

   a. In **Virtual Machine Manager**, select the host connection you want to configure.

   Open the **Edit** menu and select **Connection Details**.

   b. Click the **Storage** tab in the **Connection Details** window.

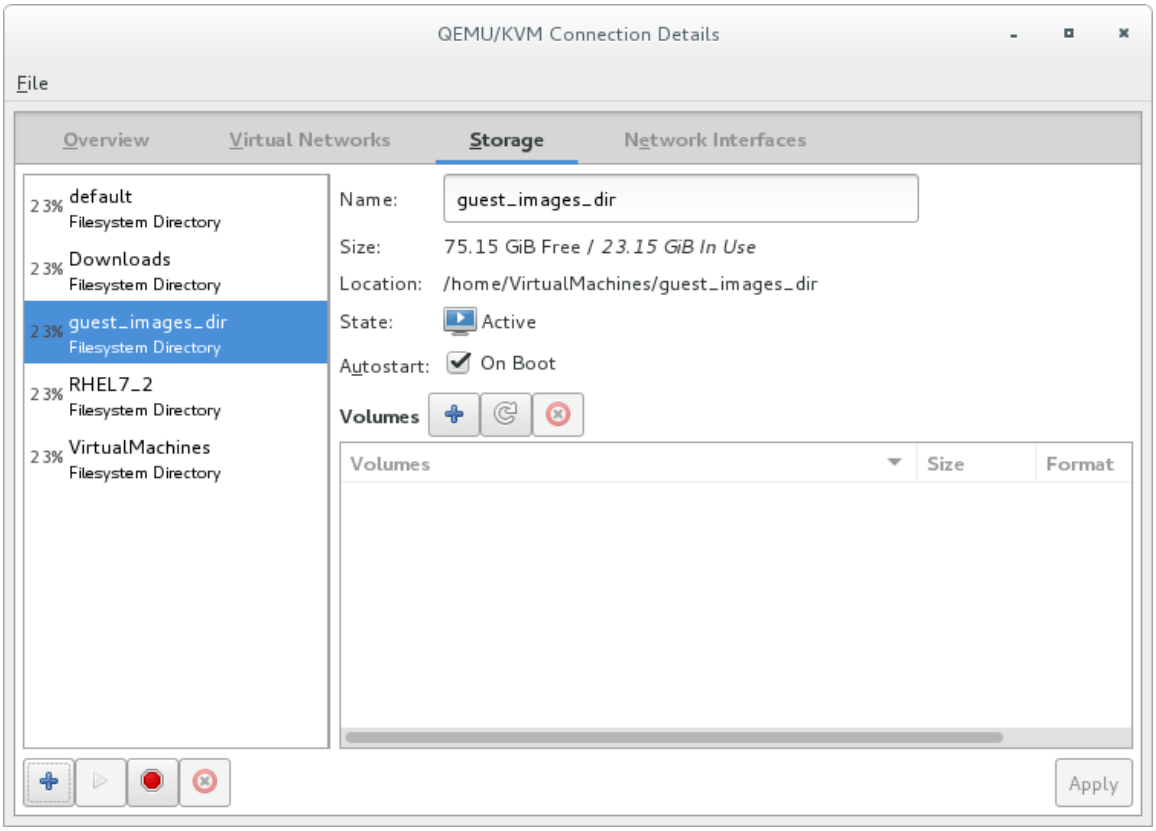


Figure 13.1. Storage tab

3. **Create a new storage pool**

NOTE

Using **Virtual Machine Manager**, you can only create persistent storage pools. Transient storage pools can only be created using **virsh**.

a. **Add a new storage pool (part 1)**

Click the button at the bottom of the window. The **Add a New Storage Pool** wizard appears.

Enter a **Name** for the storage pool. This example uses the name *guest_images_fs*.

Select a storage pool type to create from the **Type** drop-down list. This example uses **fs: Pre-Formatted Block Device**.

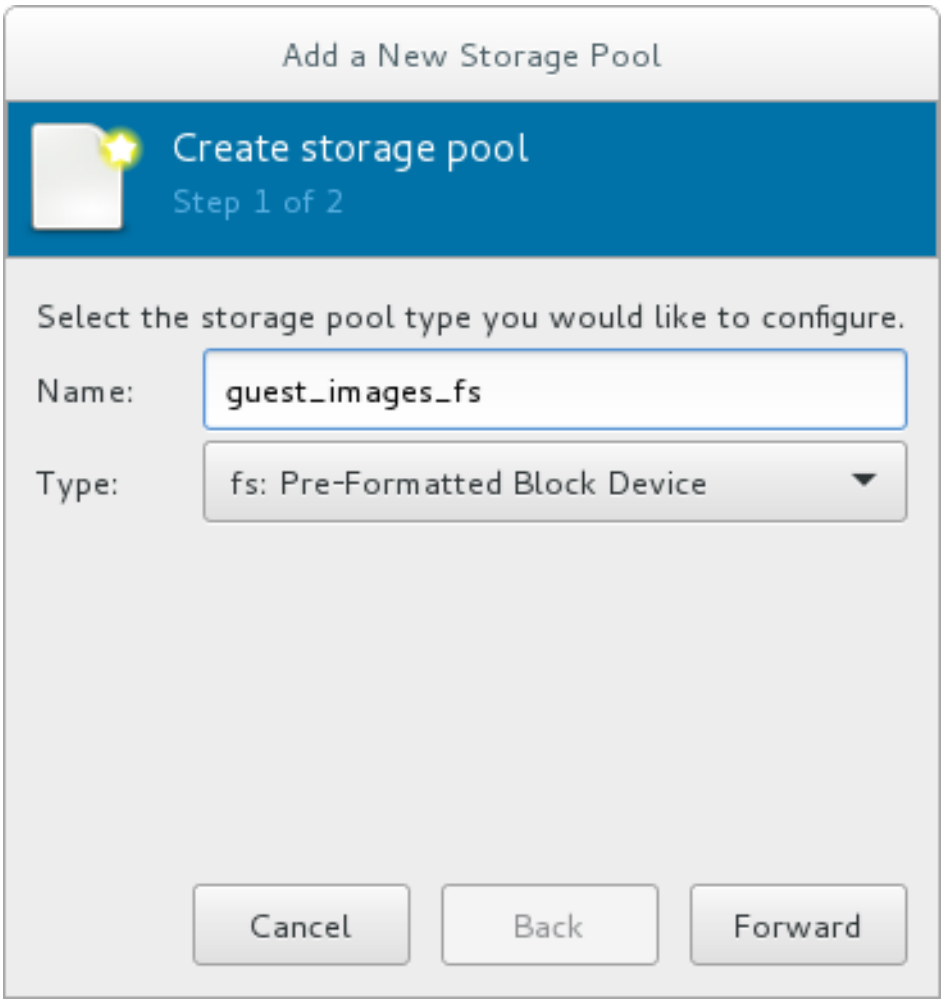


Figure 13.2. Storage pool name and type

Click the **Forward** button to continue.

b. **Add a new pool (part 2)**

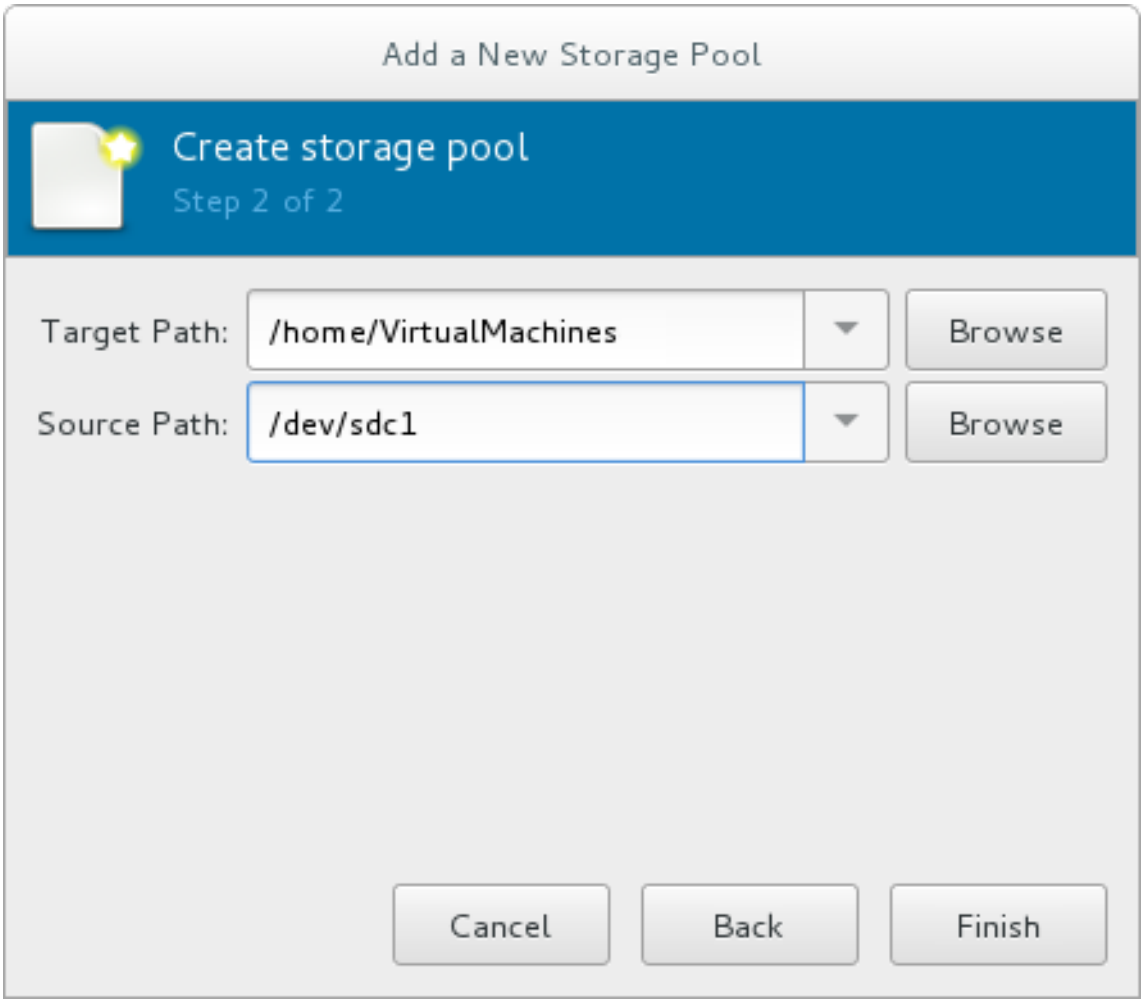


Figure 13.3. Storage pool path

Configure the storage pool with the relevant parameters. For information on the parameters for each type of storage pool, see Section 13.2.3, “Storage Pool Specifics”.

For some types of storage pools, a **`Build Pool`** check box appears in the dialog. If you want to build the storage pool from the storage, check the **`Build Pool`** check box.

Verify the details and click the **`Finish`** button to create the storage pool.

13.2.3. Storage Pool Specifics

This section provides information specific to each type of storage pool, including prerequisites, parameters, and additional information. It includes the following topics:

- Section 13.2.3.1, “Directory-based storage pools”
- Section 13.2.3.2, “Disk-based storage pools”
- Section 13.2.3.3, “Filesystem-based storage pools”
- Section 13.2.3.4, “GlusterFS-based storage pools”
- Section 13.2.3.5, “iSCSI-based storage pools”
- Section 13.2.3.6, “LVM-based storage pools”

- Section 13.2.3.7, “NFS-based storage pools”
- Section 13.2.3.8, “vHBA-based storage pools using SCSI devices”

13.2.3.1. Directory-based storage pools

Parameters
The following table provides a list of required parameters for the XML file, the **`virsh pool-define-as`** command, and the **Virtual Machine Manager** application, for creating a directory-based storage pool.

Table 13.1. Directory-based storage pool parameters

Description	XML	pool-define-as	Virtual Machine Manager
The type of storage pool	<pool type='dir'>	[type] directory	dir: Filesystem Directory
The name of the storage pool	<name>*name*</name>	[name] *name*	Name
The path specifying the target. This will be the path used for the storage pool.	<target> <path>*target_path*</path> </target>	target *path_to_pool*	Target Path

If you are using **`virsh`** to create the storage pool, continue by verifying that the pool was created.

Examples
The following is an example of an XML file for a storage pool based on the **`/guest_images`** directory:

```
<pool type='dir'>
  <name>dirpool</name>
  <target>
    <path>/guest_images</path>
  </target>
</pool>
```

The following is an example of a command for creating a storage pool based on the **`/guest_images`** directory:

```
# virsh pool-define-as dirpool dir --target "/guest_images"
Pool FS_directory defined
```

The following images show an example of the **Virtual Machine Manager** Add a New Storage Pool dialog boxes for creating a storage pool based on the **`/guest_images`** directory:

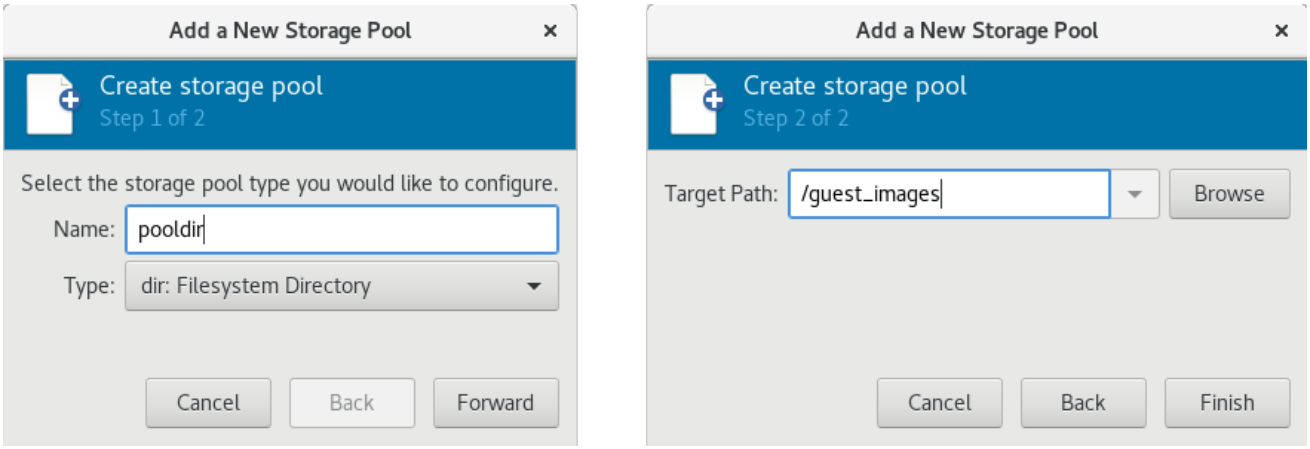


Figure 13.4. Add a new directory-based storage pool example

13.2.3.2. Disk-based storage pools

Recommendations
Be aware of the following before creating a disk-based storage pool:

- Depending on the version of libvirt being used, dedicating a disk to a storage pool may reformat and erase all data currently stored on the disk device. It is strongly recommended that you back up the data on the storage device before creating a storage pool.
- Guests should not be given write access to whole disks or block devices (for example, **`/dev/sdb`**). Use partitions (for example, **`/dev/sdb1`**) or LVM volumes.

  If you pass an entire block device to the guest, the guest will likely partition it or create its own LVM groups on it. This can cause the host physical machine to detect these partitions or LVM groups and cause errors.

Prerequisites

NOTE

The steps in this section are only required if you do not run the **`virsh pool-build`** command.

Before a disk-based storage pool can be created on a host disk, the disk must be relabeled with a *GUID Partition Table* (GPT) disk label. GPT disk labels allow for creating up to 128 partitions on each device.

```
# parted /dev/sdb
GNU Parted 2.1
Using /dev/sdb
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) mklabel
New disk label type? gpt
(parted) quit
Information: You may need to update /etc/fstab.
#
```

After relabeling the disk, continue creating the storage pool with defining the storage pool.

Parameters

The following table provides a list of required parameters for the XML file, the **`virsh pool-define-as`** command, and the **Virtual Machine Manager** application, for creating a disk-based storage pool.

Table 13.2. Disk-based storage pool parameters

Description	XML	pool-define-as	Virtual Machine Manager
The type of storage pool	<pool type='disk'>	[type] disk	disk: Physical Disk Device
The name of the storage pool	<name>*name*</name>	[name] *name*	Name
The path specifying the storage device. For example, /dev/sdb	<source> <device path=/dev/*sdb*/> <source>	source-dev *path_to_disk*	Source Path
The path specifying the target. This will be the path used for the storage pool.	<target> <path>/*path_to_pool*</path> </target>	target *path_to_pool*	Target Path

If you are using **`virsh`** to create the storage pool, continue with defining the storage pool.

Examples
The following is an example of an XML file for a disk-based storage pool:

```
<pool type='disk'>
  <name>phy_disk</name>
  <source>
    <device path='/dev/sdb'/>
    <format type='gpt'/>
  </source>
  <target>
    <path>/dev</path>
  </target>
</pool>
```

The following is an example of a command for creating a disk-based storage pool:

```
# virsh pool-define-as phy_disk disk gpt --source-dev=/dev/sdb --target
/dev
Pool phy_disk defined
```

The following images show an example of the virtual machine XML configuration**Virtual Machine Manager** Add a New Storage Pool dialog boxes for creating a disk-based storage pool:

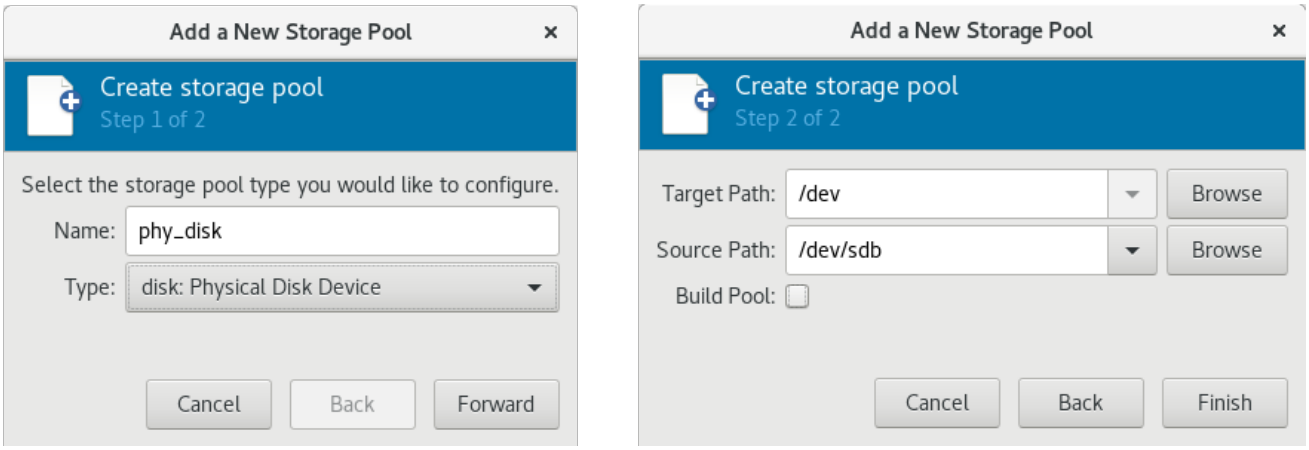


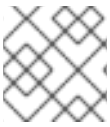
Figure 13.5. Add a new disk-based storage pool example

13.2.3.3. Filesystem-based storage pools

Recommendations

Do not use the procedures in this section to assign an entire disk as a storage pool (for example, **`/dev/sdb`**). Guests should not be given write access to whole disks or block devices. This method should only be used to assign partitions (for example, **`/dev/sdb1`**) to storage pools.

Prerequisites



NOTE

This is only required if you do not run the **`virsh pool-build`** command.

To create a storage pool from a partition, format the file system to ext4.

```
# mkfs.ext4 /dev/sdc1
```

After formatting the file system, continue creating the storage pool with defining the storage pool.

Parameters

The following table provides a list of required parameters for the XML file, the **`virsh pool-define-as`** command, and the **Virtual Machine Manager** application, for creating a filesystem-based storage pool from a partition.

Table 13.3. Filesystem-based storage pool parameters

Description	XML	pool-define-as	Virtual Machine Manager
The type of storage pool	<pool type='fs'>	[type] fs	fs: Pre-Formatted Block Device
The name of the storage pool	<name>*name*</name>	[name] *name*	Name

Description	XML	pool-define-as	Virtual Machine Manager
The path specifying the partition. For example, /dev/sdc1	<source> <device path='*source_path*' />	[source] *path_to_partition*	Source Path
The filesystem type, for example ext4	<format type='*fs_type*' /> </source>	[source format] *FS-format*	N/A
The path specifying the target. This will be the path used for the storage pool.	<target> <path>/*path_to_pool*</path> </target>	[target] *path_to_pool*	Target Path

If you are using **virsh** to create the storage pool, continue with verifying that the storage pool was created.

Examples
The following is an example of an XML file for a filesystem-based storage pool:

```
<pool type='fs'>
  <name>guest_images_fs</name>
  <source>
    <device path='/dev/sdc1'/>
    <format type='auto'/>
  </source>
  <target>
    <path>/guest_images</path>
  </target>
</pool>
```

The following is an example of a command for creating a partition-based storage pool:

```
# virsh pool-define-as guest_images_fs fs --source-dev /dev/sdc1 --target
/guest_images
Pool guest_images_fs defined
```

The following images show an example of the virtual machine XML configuration**Virtual Machine Manager** Add a New Storage Pool dialog boxes for creating a filesystem-based storage pool:

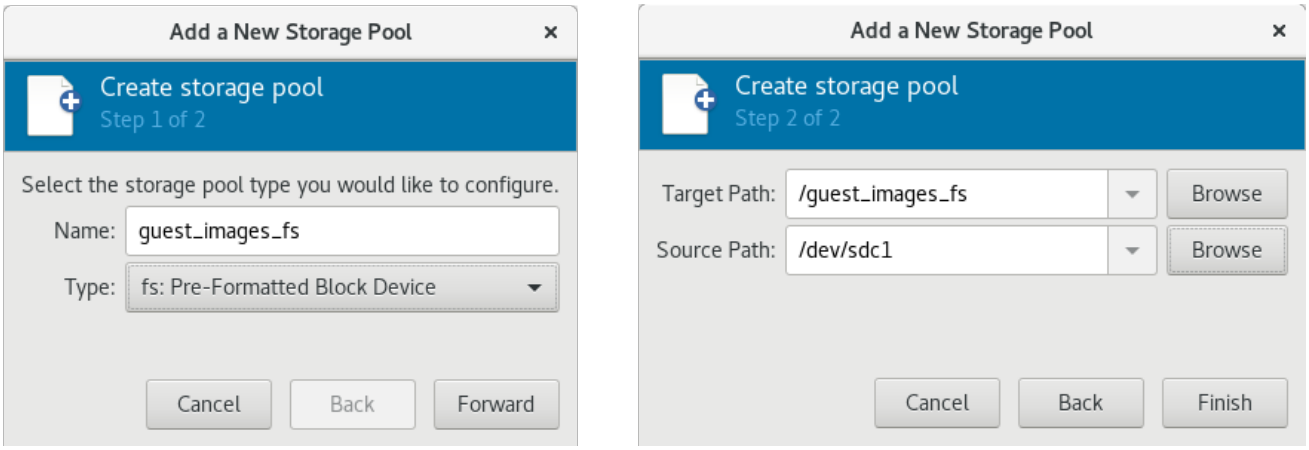


Figure 13.6. Add a new filesystem-based storage pool example

13.2.3.4. GlusterFS-based storage pools

Recommendations
GlusterFS is a user space file system that uses File System in User Space (FUSE).

Prerequisites
Before a GlusterFS-based storage pool can be created on a host, a Gluster server must be prepared.

Procedure 13.4. Preparing a Gluster server

1. Obtain the IP address of the Gluster server by listing its status with the following command:

```
# gluster volume status
Status of volume: gluster-vol1
Gluster process      Port Online Pid
------------------------------------------------------------------
--------------
Brick 222.111.222.111:/gluster-vol1    49155 Y 18634

Task Status of Volume gluster-vol1
------------------------------------------------------------------
--------------
There are no active volume tasks
```

2. If not installed, install the glusterfs-fuse package.

3. If not enabled, enable the virt_use_fusefs boolean. Check that it is enabled.

```
# setsebool virt_use_fusefs on
# getsebool virt_use_fusefs
virt_use_fusefs --> on
```

After ensuring that the required packages are installed and enabled, continue creating the storage pool continue creating the storage pool with defining the storage pool.

Parameters
The following table provides a list of required parameters for the XML file, the **`virsh pool-define-as`** command, and the **Virtual Machine Manager** application, for creating a GlusterFS-based storage pool.

Table 13.4. GlusterFS-based storage pool parameters

Description	XML	pool-define-as	Virtual Machine Manager
The type of storage pool	<pool type='gluster'>	[type] gluster	Gluster: Gluster Filesystem
The name of the storage pool	<name>*name*</name>	[name] *name*	Name
The hostname or IP address of the Gluster server	<source> <hostname='*hostname*' />	source-host *hostname*	Host Name
The name of the Gluster server	<name='*Gluster-name*' />	source-name *Gluster-name*	Source Name
The path on the Gluster server used for the storage pool	<dir path='*Gluster-path*' /> </source>	source-path *Gluster-path*	Source Path

If you are using **`virsh`** to create the storage pool, continue with verifying that the storage pool was created.

Examples
The following is an example of an XML file for a GlusterFS-based storage pool:

```
<pool type='gluster'>
  <name>Gluster_pool</name>
  <source>
    <host name='111.222.111.222'/>
    <dir path='/'/>
    <name>gluster-vol1</name>
  </source>
</pool>
```

The following is an example of a command for creating a GlusterFS-based storage pool:

```
# pool-define-as --name Gluster_pool --type gluster --source-host
111.222.111.222 --source-name gluster-vol1 --source-path /
Pool Gluster_pool defined
```

The following images show an example of the virtual machine XML configuration**Virtual Machine Manager** Add a New Storage Pool dialog boxes for creating a GlusterFS-based storage pool:

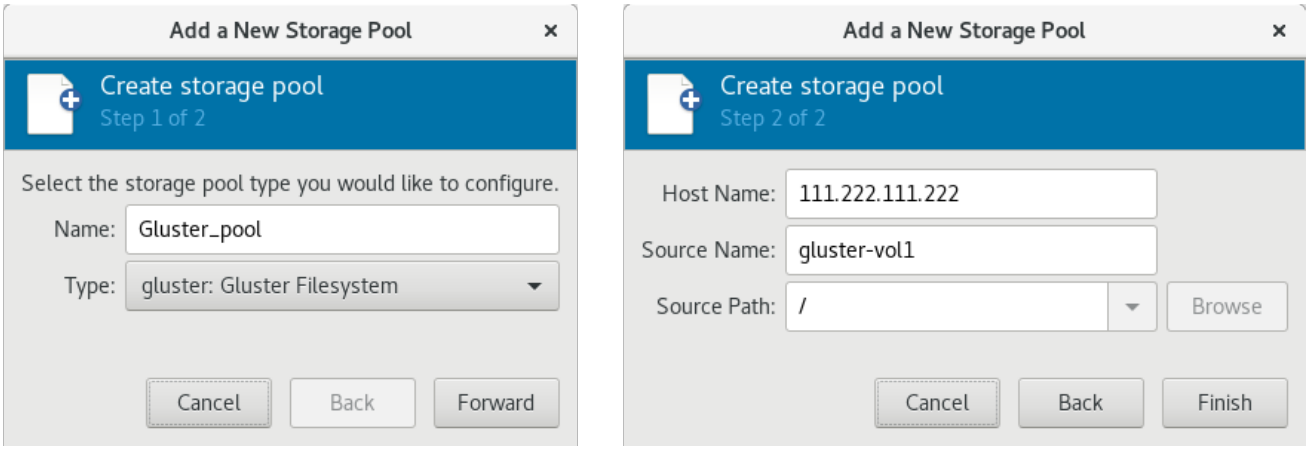


Figure 13.7. Add a new GlusterFS-based storage pool example

13.2.3.5. iSCSI-based storage pools

Recommendations
Internet Small Computer System Interface (iSCSI) is a network protocol for sharing storage devices. iSCSI connects initiators (storage clients) to targets (storage servers) using SCSI instructions over the IP layer.

Using iSCSI-based devices to store guest virtual machines allows for more flexible storage options, such as using iSCSI as a block storage device. The iSCSI devices use a Linux-IO (LIO) target. This is a multi-protocol SCSI target for Linux. In addition to iSCSI, LIO also supports Fibre Channel and Fibre Channel over Ethernet (FCoE).

Prerequisites
Before an iSCSI-based storage pool can be created, iSCSI targets must be created. iSCSI targets are created with the targetcli package, which provides a command set for creating software-backed iSCSI targets.

Procedure 13.5. Creating an iSCSI target

1. **Install the targetcli package**

   ```
   # yum install targetcli
   ```

2. **Launch the `targetcli` command set**

   ```
   # targetcli
   ```

3. **Create storage objects**
   Create three storage objects, using a storage pool.

   a. Create a block storage object

      i. Navigate to the **`/backstores/block`** directory.

      ii. Run the **`create`** command.

      ```
      # create [block-name][filepath]
      ```

      For example:

```
# create block1 dev=/dev/sdb1
```

b. Create a fileio object

i. Navigate to the **/fileio** directory.

ii. Run the **create** command.

```
# create [fileio-name][image-name] [image-size]
```

For example:

```
# create fileio1 /foo.img 50M
```

c. Create a ramdisk object

i. Navigate to the **/ramdisk** directory.

ii. Run the **create** command.

```
# create [ramdisk-name] [ramdisk-size]
```

For example:

```
# create ramdisk1 1M
```

d. Make note of the names of the disks created in this step. They will be used later.

4. **Create an iSCSI target**

a. Navigate to the **/iscsi** directory.

b. Create the target in one of two ways:

- Run the **create** command with no parameters.

  The *iSCSI qualified name* (IQN) is generated automatically.

- Run the **create** command specifying the IQN and the server. For example:

```
# create iqn.2010-05.com.example.server1:iscsirhel7guest
```

5. **Define the portal IP address**
To export the block storage over iSCSI, the portal, LUNs, and *access control lists* ACLs must first be configured.

The portal includes the IP address and TCP that the target monitors, and the initiators to which it connects. iSCSI uses port 3260. This port is configured by default.

To connect to port 3260:

a. Navigate to the **/tpg** directory.

b. Run the following:

```
# portals/ create
```

This command makes all available IP addresses listening to port 2360.

If you want only a single IP address to listen to port 3260, add the IP address to the end of the command. For example:

```
# portals/ create 143.22.16.33
```

6. **Configure the LUNs and assign storage objects to the fabric**
This step uses the storage objects created in creating storage objects.

a. Navigate to the **luns** directory for the TPG created in defining the portal IP address. For example:

```
# iscsi>iqn.iqn.2010-05.com.example.server1:iscsirhel7guest
```

b. Assign the first LUN to the ramdisk. For example:

```
# create /backstores/ramdisk/ramdisk1
```

c. Assign the second LUN to the block disk. For example:

```
# create /backstores/block/block1
```

d. Assign the third LUN to the fileio disk. For example:

```
# create /backstores/fileio/fileio1
```

e. List the resulting LUNs.

```
/iscsi/iqn.20...csirhel7guest ls

o- tgp1
.....
[enabled, auth]
  o-
acls.....
......[0 ACL]
  o-
luns.....
.....[3 LUNs]
  | o- lun0.....
[ramdisk/ramdisk1]
  | o- lun1.....
[block/block1 (dev/vdb1)]
  | o- lun2.....
[fileio/file1 (foo.img)]
  o-
portals.....
...[1 Portal]
```

```
    o- IP-
ADDRESS:3260.....................................................
....[OK]
```

7. **Create ACLs for each initiator**
   Enable authentication when the initiator connects. You can also resrict specified LUNs to specified intiators. Targets and initiators have unique names. iSCSI initiators use IQNs.

   a. Find the IQN of the iSCSI initiator, using the initiator name. For example:

   ```
   # cat /etc/iscsi/initiator2.iscsi
   InitiatorName=create iqn.2010-
   05.com.example.server1:iscsirhel7guest
   ```

   This IQN is used to create the ACLs.

   b. Navigate to the **acls** directory.

   c. Create ACLs by doing one of the following:

   - Create ACLS for all LUNs and initiators by running the **create** command with no parameters.

     ```
     # create
     ```

   - Create an ACL for a specific LUN and initiator, run the **create** command specifying the IQN of the iSCSI intiator. For example:

     ```
     # create iqn.2010-05.com.example.server1:888
     ```

   - Configure the kernel target to use a single user ID and password for all initiators.

     ```
     # set auth userid=user_ID
     # set auth password=password
     # set attribute authentication=1
     # set attribute generate_node_acls=1
     ```

   After completing this procedure, continue by securing the storage pool.

8. **Save the configuration**
   Make the configuration persistent by overwriting the previous boot settings.

   ```
   # saveconfig
   ```

9. **Enable the service**
   To apply the saved settings on the next boot, enable the service.

   ```
   # systemctl enable target.service
   ```

Optional procedures
There are a number of optional procedures that you can perform with the iSCSI targets before creating the iSCSI-based storage pool.

Procedure 13.6. Configuring a logical volume on a RAID array

1. **Create a RAID5 array**
   For information on creating a RAID5 array, see the *Red Hat Enterprise Linux 7 Storage Administration Guide*.

2. **Create an LVM logical volume on the RAID5 array**
   For information on creating an LVM logical volume on a RAID5 array, see the *Red Hat Enterprise Linux 7 Logical Volume Manager Administration Guide*.

Procedure 13.7. Testing discoverability

- **Ensure that the new iSCSI device is discoverable.**

```
# iscsiadm --mode discovery --type sendtargets --portal
server1.example.com
143.22.16.33:3260,1 iqn.2010-05.com.example.server1:iscsirhel7guest
```

Procedure 13.8. Testing device attachment

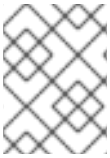
1. **Attach the new iSCSI device**
   Attach the new device (*iqn.2010-05.com.example.server1:iscsirhel7guest*) to determine whether the device can be attached.

```
# iscsiadm -d2 -m node --login
scsiadm: Max file limits 1024 1024

Logging in to [iface: default, target: iqn.2010-
05.com.example.server1:iscsirhel7guest, portal: 143.22.16.33,3260]
Login to [iface: default, target: iqn.2010-
05.com.example.server1:iscsirhel7guest, portal: 143.22.16.33,3260]
successful.
```

2. **Detach the device**

```
# iscsiadm -d2 -m node --logout
scsiadm: Max file limits 1024 1024

Logging out of session [sid: 2, target: iqn.2010-
05.com.example.server1:iscsirhel7guest, portal: 143.22.16.33,3260
Logout of [sid: 2, target: iqn.2010-
05.com.example.server1:iscsirhel7guest, portal: 143.22.16.33,3260]
successful.
```

Procedure 13.9. Using libvirt secrets for an iSCSI storage pool



NOTE

This procedure is required if a *user_ID* and *password* were defined when creating an iSCSI target.

User name and password parameters can be configured with **virsh** to secure an iSCSI storage pool. This can be configured before or after the pool is defined, but the pool must be started for the authentication settings to take effect.

1. **Create a libvirt secret file**
   Create a libvirt secret file with a challenge-handshake authentication protocol (CHAP) user name. For example:

```
<secret ephemeral='no' private='yes'>
    <description>Passphrase for the iSCSI example.com
server</description>
    <usage type='iscsi'>
        <target>iscsirhel7secret</target>
    </usage>
</secret>
```

2. **Define the secret**

```
# virsh secret-define secret.xml
```

3. **Verify the UUID**

```
# virsh secret-list
UUID                                   Usage
--------------------------------------------------------------------
---------------
2d7891af-20be-4e5e-af83-190e8a922360  iscsi iscsirhel7secret
```

4. **Assign a secret to the UID**
   Use the following commands to assign a secret to the UUID in the output of the previous step. This ensures that the CHAP username and password are in a libvirt-controlled secret list.

```
# MYSECRET=`printf %s "password123" | base64`
# virsh secret-set-value 2d7891af-20be-4e5e-af83-190e8a922360
$MYSECRET
```

5. **Add an authentication entry to the storage pool**
   Modify the **<source>** entry in the storage pool's XML file using **virsh edit**, and add an **<auth>** element, specifying ***authentication type***, ***username***, and ***secret usage***.

   For example:

```
<pool type='iscsi'>
  <name>iscsirhel7pool</name>
    <source>
       <host name='192.168.122.1'/>
       <device path='iqn.2010-
05.com.example.server1:iscsirhel7guest'/>
       <auth type='chap' username='redhat'>
          <secret usage='iscsirhel7secret'/>
       </auth>
    </source>
  <target>
```

```
    <path>/dev/disk/by-path</path>
  </target>
</pool>
```

NOTE

The ***<auth>*** sub-element exists in different locations within the guest XML's ***<pool>*** and ***<disk>*** elements. For a ***<pool>***, ***<auth>*** is specified within the ***<source>*** element, as this describes where to find the pool sources, since authentication is a property of some pool sources (iSCSI and RBD). For a ***<disk>***, which is a sub-element of a domain, the authentication to the iSCSI or RBD disk is a property of the disk.

In addition, the ***<auth>*** sub-element for a disk differs from that of a storage pool.

```
<auth username='redhat'>
  <secret type='iscsi' usage='iscsirhel7secret'/>
</auth>
```

6. **Activate the changes**
   The storage pool must be started to activate these changes.

   - If the storage pool has not yet been started, follow the steps in Creating Storage Pools with virsh to define and start the storage pool.
   - If the pool has already been started, enter the following commands to stop and restart the storage pool:

     ```
     # virsh pool-destroy iscsirhel7pool
     # virsh pool-start iscsirhel7pool
     ```

Parameters
The following table provides a list of required parameters for the XML file, the **`virsh pool-define-as`** command, and the **Virtual Machine Manager** application, for creating an iSCSI-based storage pool.

Table 13.5. iSCSI-based storage pool parameters

Description	XML	pool-define-as	Virtual Machine Manager
The type of storage pool	<pool type='iscsi'>	[type] iscsi	iscsi: iSCSI Target
The name of the storage pool	<name>*name*</name>	[name] *name*	Name
The name of the host.	<source> <host name='*hostname*' />	source-host *hostname*	Host Name

Description	XML	pool-define-as	Virtual Machine Manager
The iSCSI IQN.	device path="*iSCSI_IQN*" /> </source>	source-dev *iSCSI_IQN*	Source IQN
The path specifying the target. This will be the path used for the storage pool.	<target> <path>/dev/*disk/by-path*</path> </target>	target *path_to_pool*	Target Path
(Optional) The IQN of the iSCSI initiator. This is only needed when the ACL restricts the LUN to a particular initiator.	<initiator> <iqn name='*initiator0*' /> </initiator>	*See the note below.*	Initiator IQN

NOTE

The IQN of the iSCSI initiator can be determined using the **`virsh find-storage-pool-sources-as iscsi`** command.

If you are using **`virsh`** to create the storage pool, continue with verifying that the storage pool was created.

Examples
The following is an example of an XML file for an iSCSI-based storage pool:

```
<pool type='iscsi'>
  <name>iSCSI_pool</name>
  <source>
    <host name='server1.example.com'/>
    <device path='iqn.2010-05.com.example.server1:iscsirhel7guest'/>
  </source>
  <target>
    <path>/dev/disk/by-path</path>
  </target>
</pool>
```

The following is an example of a command for creating an iSCSI-based storage pool:

```
# virsh pool-define-as --name iSCSI_pool --type iscsi --source-host
server1.example.com --source-dev iqn.2010-
05.com.example.server1:iscsirhel7guest --target /dev/disk/by-path
Pool iSCSI_pool defined
```

The following images show an example of the virtual machine XML configuration**Virtual Machine Manager** Add a New Storage Pool dialog boxes for creating an iSCSI-based storage pool:

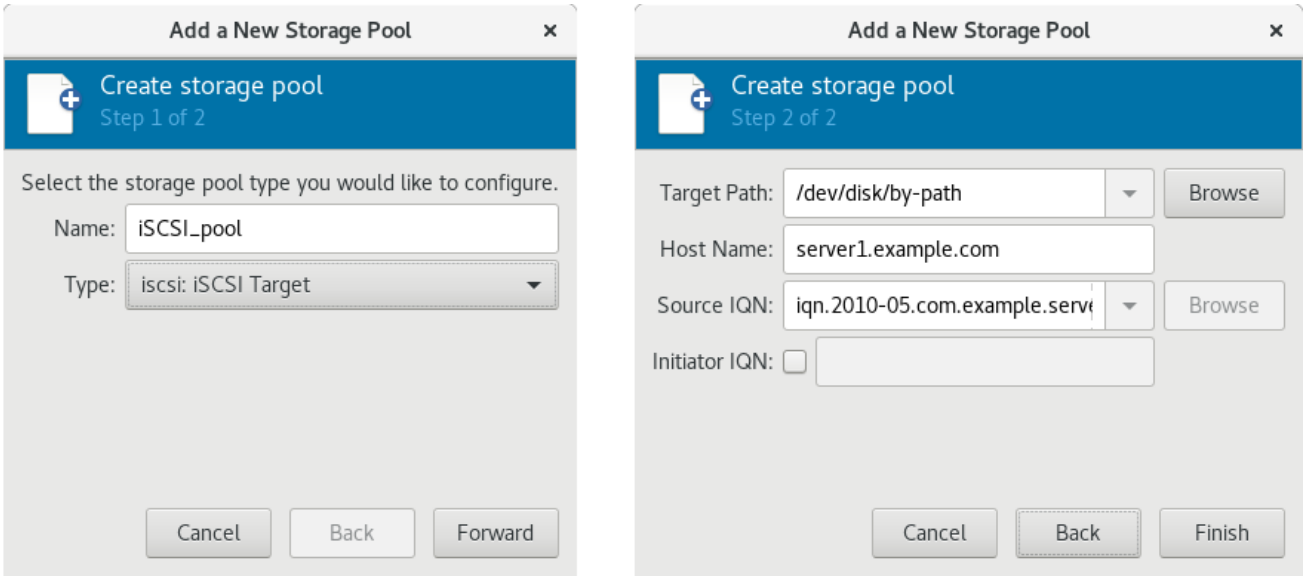


Figure 13.8. Add a new iSCSI-based storage pool example

13.2.3.6. LVM-based storage pools

Recommendations
Be aware of the following before creating an LVM-based storage pool:

- LVM-based storage pools do not provide the full flexibility of LVM.
- libvirt supports thin logical volumes, but does not provide the features of thin storage pools.
- LVM-based storage pools are volume groups. You can create volume groups using Logical Volume Manager commands or **`virsh`** commands. To manage volume groups using the **`virsh`** interface, use the **`virsh`** commands to create volume groups.

  For more information about volume groups, see the *Red Hat Enterprise Linux Logical Volume Manager Administration Guide*.
- LVM-based storage pools require a full disk partition. If activating a new partition or device with these procedures, the partition will be formatted and all data will be erased. If using the host's existing Volume Group (VG) nothing will be erased. It is recommended to back up the storage device before commencing the following procedure.

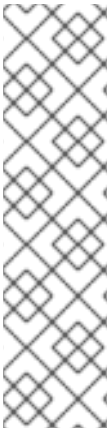
  For information on creating LVM volume groups, see the *Red Hat Enterprise Linux Logical Volume Manager Administration Guide*.
- If you create an LVM-based storage pool on an existing VG, you should not run the **`pool-build`** command.

After ensuring that the VG is prepared, continue creating the storage pool with defining the storage pool.

Parameters
The following table provides a list of required parameters for the XML file, the **`virsh pool-define-as`** command, and the **Virtual Machine Manager** application, for creating an LVM-based storage pool.

Table 13.6. LVM-based storage pool parameters

Description	XML	pool-define-as	Virtual Machine Manager
The type of storage pool	<pool type='logical'>	[type] logical	logical: LVM Volume Group
The name of the storage pool	<name>*name*</name>	[name] *name*	Name
The path to the device for the storage pool	<source> <device path='*device_path*' />	source-dev *device_path*	Source Path
The name of the volume group	<name='*VG-name*' />	source-name *VG-name*	Source Path
The virtual group format	<format type='lvm2' /> </source>	source-format lvm2	N/A
The target path	<target> <path='*target-path*' /> </target>	target *target-path*	Target Path



NOTE

If the logical volume group is made of multiple disk partitions, there may be multiple source devices listed. For example:

```
<source>
  <device path='/dev/sda1'/>
  <device path='/dev/sdb3'/>
  <device path='/dev/sdc2'/>
  ...
  </source>
```

If you are using **virsh** to create the storage pool, continue with verifying that the storage pool was created.

Examples
The following is an example of an XML file for an LVM-based storage pool:

```
<pool type='logical'>
  <name>guest_images_lvm</name>
  <source>
    <device path='/dev/sdc'/>
    <name>libvirt_lvm</name>
```

```
    <format type='lvm2'/>
  </source>
  <target>
    <path>/dev/libvirt_lvm</path>
  </target>
</pool>
```

The following is an example of a command for creating an LVM-based storage pool:

```
# virsh pool-define-as guest_images_lvm logical --source-dev=/dev/sdc --
source-name libvirt_lvm --target /dev/libvirt_lvm
Pool guest_images_lvm defined
```

The following images show an example of the virtual machine XML configuration**Virtual Machine Manager** Add a New Storage Pool dialog boxes for creating an LVM-based storage pool:

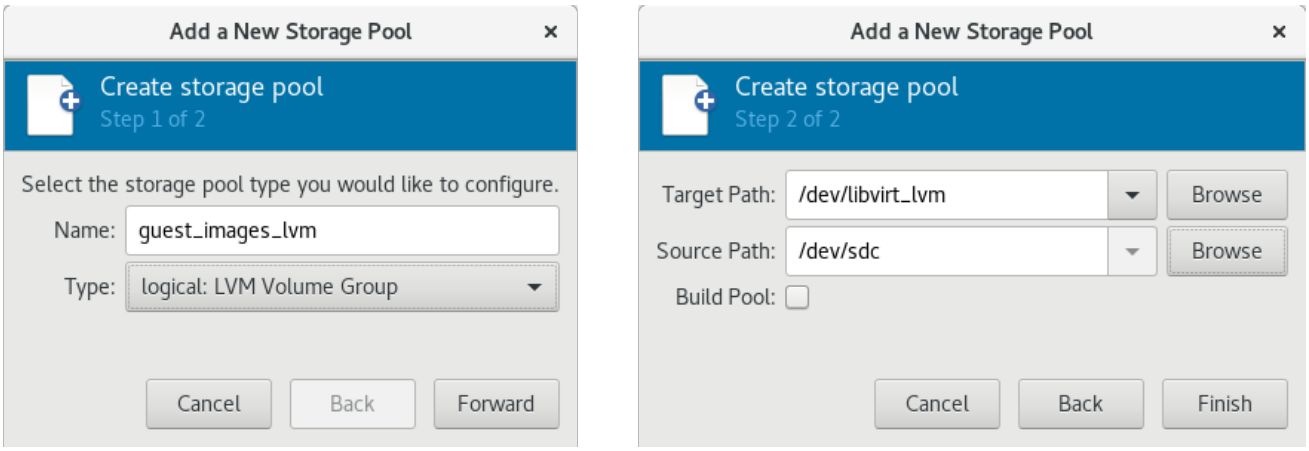


Figure 13.9. Add a new LVM-based storage pool example

13.2.3.7. NFS-based storage pools

Prerequisites
To create an Network File System (NFS)-based storage pool, an NFS Server should already be configured to be used by the host machine. For more information about NFS, see the *Red Hat Enterprise Linux Storage Administration Guide*.

After ensuring that the NFS Server is properly configured, continue creating the storage pool with defining the storage pool.

Parameters
The following table provides a list of required parameters for the XML file, the **`virsh pool-define-as`** command, and the **Virtual Machine Manager** application, for creating an NFS-based storage pool.

Table 13.7. NFS-based storage pool parameters

Description	XML	pool-define-as	Virtual Machine Manager

Description	XML	pool-define-as	Virtual Machine Manager
The type of storage pool	<pool type='netfs'>	[type] netfs	netfs: Network Exported Directory
The name of the storage pool	<name>*name*</name>	[name] *name*	Name
The hostname of the NFS server where the mount point is located. This can be a hostname or an IP address.	<source> <host name='*host_name*' />	source-host *host_name*	Host Name
The directory used on the NFS server	<dir path='*source_path*' /> </source>	source-path *source_path*	Source Path
The path specifying the target. This will be the path used for the storage pool.	<target> <path>/*target_path*</path> </target>	target *target_path*	Target Path

If you are using **`virsh`** to create the storage pool, continue with verifying that the storage pool was created.

Examples
The following is an example of an XML file for an NFS-based storage pool:

```
<pool type='netfs'>
  <name>nfspool</name>
  <source>
    <host name='localhost'/>
    <dir path='/home/net_mount'/>
  </source>
  <target>
    <path>/var/lib/libvirt/images/nfspool</path>
  </target>
</pool>
```

The following is an example of a command for creating an NFS-based storage pool:

```
# virsh pool-define-as nfspool netfs --source-host localhost --source-path
/home/net_mount --target /var/lib/libvirt/images/nfspool
Pool nfspool defined
```

The following images show an example of the virtual machine XML configuration**Virtual Machine Manager** Add a New Storage Pool dialog boxes for creating an NFS-based storage pool:

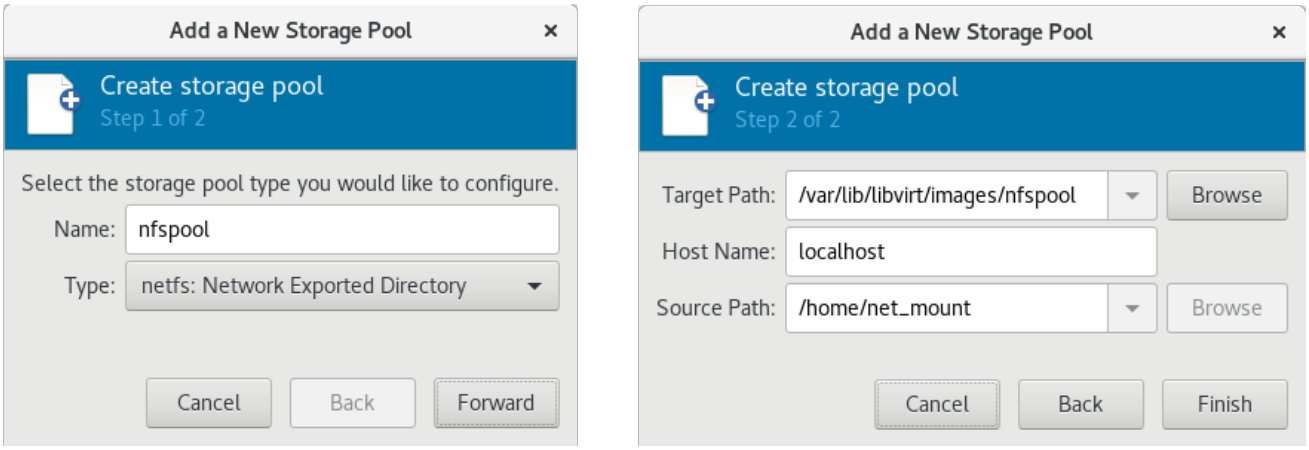


Figure 13.10. Add a new NFS-based storage pool example

13.2.3.8. vHBA-based storage pools using SCSI devices

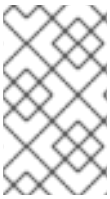
NOTE

You cannot use **Virtual Machine Manager** to create vHBA-based storage pools using SCSI devices.

Recommendations
N_Port ID Virtualization (NPIV) is a software technology that allows sharing of a single physical Fibre Channel host bus adapter (HBA). This allows multiple guests to see the same storage from multiple physical hosts, and thus allows for easier migration paths for the storage. As a result, there is no need for the migration to create or copy storage, as long as the correct storage path is specified.

In virtualization, the *virtual host bus adapter*, or *vHBA*, controls the Logical Unit Numbers (LUNs) for virtual machines. For a host to share one Fibre Channel device path between multiple KVM guests, a vHBA must be created for each virtual machine. A single vHBA must not be used by multiple KVM guests.

Each vHBA for NPIV is identified by its parent HBA and its own World Wide Node Name (WWNN) and World Wide Port Name (WWPN). The path to the storage is determined by the WWNN and WWPN values. The parent HBA can be defined as **`scsi_host#`** or as a WWNN/WWPN pair.



NOTE

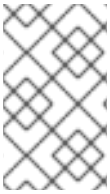
If a parent HBA is defined as **`scsi_host#`** and hardware is added to the host machine, the **`scsi_host#`** assignment may change. Therefore, it is recommended that you define a parent HBA using a WWNN/WWPN pair.

It is recommended that you define a libvirt storage pool based on the vHBA, because this preserves the vHBA configuration.

Using a libvirt storage pool has two primary advantages:

- The libvirt code can easily find the LUN's path using the virsh command output.
- Virtual machine migration requires only defining and starting a storage pool with the same vHBA name on the target machine. To do this, the vHBA LUN, libvirt storage pool and volume name must be specified in the virtual machine's XML configuration. Refer to Section 13.2.3.8, “vHBA-

based storage pools using SCSI devices" for an example.



NOTE

Before creating a vHBA, it is recommended that you configure storage array (SAN)-side zoning in the host LUN to provide isolation between guests and prevent the possibility of data corruption.

To create a persistent vHBA configuration, first create a libvirt **'scsi'** storage pool XML file using the format below. When creating a single vHBA that uses a storage pool on the same physical HBA, it is recommended to use a stable location for the **<path>** value, such as one of the **/dev/disk/by-{path|id|uuid|label}** locations on your system.

When creating multiple vHBAs that use storage pools on the same physical HBA, the value of the **<path>** field must be only **/dev/**, otherwise storage pool volumes are visible only to one of the vHBAs, and devices from the host cannot be exposed to multiple guests with the NPIV configuration.

For more information on **<path>** and the elements in **<target>**, see upstream libvirt documentation.

Prerequisites
Before creating a vHBA-based storage pools with SCSI devices, create a vHBA:

Procedure 13.10. Creating a vHBA

1. **Locate HBAs on the host system**
   To locate the HBAs on your host system, use the **virsh nodedev-list --cap vports** command.

   The following example shows a host that has two HBAs that support vHBA:

   ```
   # virsh nodedev-list --cap vports
   scsi_host3
   scsi_host4
   ```

2. **Check the HBA's details**
   Use the **virsh nodedev-dumpxml *HBA_device*** command to see the HBA's details.

   ```
   # virsh nodedev-dumpxml scsi_host3
   ```

   The output from the command lists the **<name>**, **<wwnn>**, and **<wwpn>** fields, which are used to create a vHBA. **<max_vports>** shows the maximum number of supported vHBAs. For example:

   ```
   <device>
     <name>scsi_host3</name>

   <path>/sys/devices/pci0000:00/0000:00:04.0/0000:10:00.0/host3</path>
     <parent>pci_0000_10_00_0</parent>
     <capability type='scsi_host'>
       <host>3</host>
       <unique_id>0</unique_id>
       <capability type='fc_host'>
         <wwnn>20000000c9848140</wwnn>
   ```

```
      <wwpn>10000000c9848140</wwpn>
      <fabric_wwn>2002000573de9a81</fabric_wwn>
    </capability>
    <capability type='vport_ops'>
      <max_vports>127</max_vports>
      <vports>0</vports>
    </capability>
  </capability>
</device>
```

In this example, the **<max_vports>** value shows there are a total 127 virtual ports available for use in the HBA configuration. The **<vports>** value shows the number of virtual ports currently being used. These values update after creating a vHBA.

3. **Create a vHBA host device**
   Create an XML file similar to one of the following for the vHBA host. In this examples, the file is named *vhba_host3.xml*.

   This example uses **scsi_host*3*** to describe the parent vHBA.

```
# cat vhba_host3.xml
<device>
  <parent>scsi_host3</parent>
  <capability type='scsi_host'>
    <capability type='fc_host'>
    </capability>
  </capability>
</device>
```

   This example uses a WWNN/WWPN pair to describe the parent vHBA.

```
# cat vhba_host3.xml
<device>
  <name>vhba</name>
  <parent wwnn='20000000c9848140' wwpn='10000000c9848140'/>
  <capability type='scsi_host'>
    <capability type='fc_host'>
    </capability>
  </capability>
</device>
```

NOTE

The WWNN and WWPN values must match those in the HBA details seen in Procedure 13.10, “Creating a vHBA”.

   The **<parent>** field specifies the HBA device to associate with this vHBA device. The details in the **<device>** tag are used in the next step to create a new vHBA device for the host. For more information on the **nodedev** XML format, see the libvirt upstream pages.

4. **Create a new vHBA on the vHBA host device**
   To create a vHBA on the basis of *vhba_host3*, use the **virsh nodedev-create** command:

```
# virsh nodedev-create vhba_host3.xml
Node device scsi_host5 created from vhba_host3.xml
```

5. **Verify the vHBA**
   Verify the new vHBA's details (**scsi_host5**) with the **virsh nodedev-dumpxml** command:

```
# virsh nodedev-dumpxml scsi_host5
<device>
  <name>scsi_host5</name>

<path>/sys/devices/pci0000:00/0000:00:04.0/0000:10:00.0/host3/vport-
3:0-0/host5</path>
  <parent>scsi_host3</parent>
  <capability type='scsi_host'>
    <host>5</host>
    <unique_id>2</unique_id>
    <capability type='fc_host'>
      <wwnn>5001a4a93526d0a1</wwnn>
      <wwpn>5001a4ace3ee047d</wwpn>
      <fabric_wwn>2002000573de9a81</fabric_wwn>
    </capability>
  </capability>
</device>
```

After verifying the vHBA, continue creating the storage pool with defining the storage pool.

Parameters
The following table provides a list of required parameters for the XML file, the **virsh pool-define-as** command, and the **Virtual Machine Manager** application, for creating a vHBA-based storage pool.

Table 13.8. vHBA-based storage pool parameters

Description	XML	pool-define-as
The type of storage pool	<pool type='scsi'>	scsi
The name of the storage pool	<name>*name*</name>	--adapter-name *name*
The identifier of the vHBA. The **parent** attribute is optional.	<source> <adapter type='fc_host' [parent=*parent_scsi_device*] wwnn='*WWNN*' wwpn='*WWPN*' /> </source>	[--adapter-parent *parent*] --adapter-wwnn *wwnn* --adapter-wpnn *wwpn*
The path specifying the target. This will be the path used for the storage pool.	<target> <path>*target_path*</path> </target>	target *path_to_pool*

IMPORTANT

When the **<path>** field is **/dev/**, libvirt generates a unique short device path for the volume device path. For example, **/dev/sdc**. Otherwise, the physical host path is used. For example, **/dev/disk/by-path/pci-0000:10:00.0-fc-0x5006016044602198-lun-0**. The unique short device path allows the same volume to be listed in multiple guests by multiple storage pools. If the physical host path is used by multiple guests, duplicate device type warnings may occur.

NOTE

The **parent** attribute can be used in the **<adapter>** field to identify the physical HBA parent from which the NPIV LUNs by varying paths can be used. This field, **scsi_host*N***, is combined with the **vports** and **max_vports** attributes to complete the parent identification. The **parent**, **parent_wwnn**, **parent_wwpn**, or **parent_fabric_wwn** attributes provide varying degrees of assurance that after the host reboots the same HBA is used.

- If no **parent** is specified, libvirt uses the first **scsi_host*N*** adapter that supports NPIV.
- If only the **parent** is specified, problems can arise if additional SCSI host adapters are added to the configuration.
- If **parent_wwnn** or **parent_wwpn** is specified, after the host reboots the same HBA is used.
- If **parent_fabric_wwn** is used, after the host reboots an HBA on the same fabric is selected, regardless of the **scsi_host*N*** used.

If you are using **virsh** to create the storage pool, continue with verifying that the storage pool was created.

Examples

The following are examples of XML files for vHBA-based storage pools. The first example is for an example of a storage pool that is the only storage pool on the HBA. The second example is for a storage pool that is one of several storage pools that use a single vHBA and uses the **parent** attribute to identify the SCSI host device.

```
<pool type='scsi'>
  <name>vhbapool_host3</name>
  <source>
    <adapter type='fc_host' wwnn='5001a4a93526d0a1'
wwpn='5001a4ace3ee047d'/>
  </source>
  <target>
    <path>/dev/disk/by-path</path>
  </target>
</pool>
```

```
<pool type='scsi'>
  <name>vhbapool_host3</name>
  <source>
    <adapter type='fc_host' parent='scsi_host3' wwnn='5001a4a93526d0a1'
```

```
wwpn='5001a4ace3ee047d'/>
  </source>
  <target>
    <path>/dev/disk/by-path</path>
  </target>
</pool>
```

The following is an example of a command for creating a vHBA-based storage pool:

```
# virsh pool-define-as vhbapool_host3 scsi --adapter-parent scsi_host3 --
adapter-wwnn 5001a4a93526d0a1 --adapter-wwpn 5001a4ace3ee047d --target
/dev/disk/by-path
Pool vhbapool_host3 defined
```

NOTE

The **virsh** command does not provide a way to define the **parent_wwnn**, **parent_wwpn**, or **parent_fabric_wwn** attributes.

Configuring a virtual machine to use a vHBA LUN
After a storage pool is created for a vHBA, the vHBA LUN must be added to the virtual machine configuration.

1. Create a disk volume on the virtual machine in the virtual machine's XML.
2. Specify the **storage pool** and the **storage volume** in the **<source>** parameter.

The following shows an example:

```
<disk type='volume' device='disk'>
  <driver name='qemu' type='raw'/>
  <source pool='vhbapool_host3' volume='unit:0:4:0'/>
  <target dev='hda' bus='ide'/>
</disk>
```

To specify a **lun** device instead of a **disk**, see the following example:

```
<disk type='volume' device='lun' sgio='unfiltered'>
  <driver name='qemu' type='raw'/>
  <source pool='vhbapool_host3' volume='unit:0:4:0' mode='host'/>
  <target dev='sda' bus='scsi'/>
  <shareable />
</disk>
```

For XML configuration examples of adding SCSI LUN-based storage to a guest, see Section 13.3.6.3, “Adding SCSI LUN-based Storage to a Guest”.

Note that to ensure successful reconnection to a LUN in case of a hardware failure, it is recommended that you edit the **fast_io_fail_tmo** and **dev_loss_tmo** options. For more information, see Reconnecting to an exposed LUN after a hardware failure.

13.2.4. Deleting Storage Pools

You can delete storage pools using **virsh** or the **Virtual Machine Manager**.

13.2.4.1. Prerequisites for deleting a storage pool

To avoid negatively affecting other guest virtual machines that use the storage pool you want to delete, it is recommended that you stop the storage pool and release any resources being used by it.

13.2.4.2. Deleting storage pools using virsh

1. List the defined storage pools:

```
# virsh pool-list --all
Name                 State      Autostart
-----------------------------------------
default              active     yes
guest_images_pool    active     yes
```

2. Stop the storage pool you want to delete.

```
# virsh pool-destroy guest_images_disk
```

3. *(Optional)* For some types of storage pools, you can optionally remove the directory where the storage pool resides:

```
# virsh pool-delete guest_images_disk
```

4. Remove the storage pool's definition.

```
# virsh pool-undefine guest_images_disk
```

5. Confirm the pool is undefined:

```
# virsh pool-list --all
Name                 State      Autostart
-----------------------------------------
default              active     yes
```

13.2.4.3. Deleting storage pools using Virtual Machine Manager

1. Select the storage pool you want to delete in the storage pool list in the Storage tab of the Connection Details window.
2. Click at the bottom of the Storage window. This stops the storage pool and releases any resources in use by it.
3. Click .

NOTE

The icon is only enabled if the storage pool is stopped.

The storage pool is deleted.

13.3. USING STORAGE VOLUMES

This section provides information about using storage volumes. It provides conceptual information, as well as detailed instructions on creating, configuring, and deleting storage volumes using **virsh** commands and the **Virtual Machine Manager**.

13.3.1. Storage Volume Concepts

Storage pools are divided into *storage volumes*. Storage volumes are abstractions of physical partitions, LVM logical volumes, file-based disk images, and other storage types handled by libvirt. Storage volumes are presented to guest virtual machines as local storage devices regardless of the underlying hardware.

NOTE

The sections below do not contain all of the possible commands and arguments that virsh provides for managing storage volumes> For more information, see Section 20.30, “Storage Volume Commands”.

On the host machine, a storage volume is referred to by its name and an identifier for the storage pool from which it derives. On the **virsh** command line, this takes the form ***--pool*** *storage_pool volume_name*.

For example, a volume named *firstimage* in the *guest_images* pool.

```
# virsh vol-info --pool guest_images firstimage
  Name:           firstimage
  Type:           block
  Capacity:       20.00 GB
  Allocation:     20.00 GB
```

For additional parameters and arguments, see Section 20.34, “Listing Volume Information”.

13.3.2. Creating Storage Volumes

This section provides general instructions for creating storage volumes from storage pools using **virsh** and the **Virtual Machine Manager**. After creating storage volumes, you can add storage devices to guests.

13.3.2.1. Creating Storage Volumes with virsh

Do one of the following:

- Define the storage volume using an XML file.

a. Create a temporary XML file containing the storage volume information required for the new device.

The XML file must contain specific fields including the following:

- **name** - The name of the storage volume.
- **allocation** - The total storage allocation for the storage volume.
- **capacity** - The logical capacity of the storage volume. If the volume is sparse, this value can differ from the **allocation** value.
- **target** - The path to the storage volume on the host system and optionally its permissions and label.

The following shows an example a storage volume definition XML file. In this example, the file is saved to **~/guest_volume.xml**

```
<volume>
  <name>volume1</name>
  <allocation>0</allocation>
  <capacity>20G</capacity>
  <target>
    <path>/var/lib/virt/images/sparse.img</path>
  </target>
</volume>
```

b. Use the **virsh vol-create** command to create the storage volume based on the XML file.

```
# virsh vol-create guest_images_dir ~/guest_volume.xml
  Vol volume1 created
```

c. Delete the XML file created in step a.

- Use the **virsh vol-create-as** command to create the storage volume.

```
# virsh vol-create-as guest_images_dir volume1 20GB --allocation 0
```

- Clone an existing storage volume using the **virsh vol-clone** command. The **virsh vol-clone** command must specify the storage pool that contains the storage volume to clone and the name of the newly created storage volume.

```
# virsh vol-clone --pool guest_images_dir volume1 clone1
```

13.3.2.2. Creating storage volumes with Virtual Machine Manager

Procedure 13.11. Creating Storage Volumes with Virtual Machine Manager

1. **Open the storage settings**

   a. In the **Virtual Machine Manager**, open the **Edit** menu and select **Connection Details**.

   b. Click the **Storage** tab in the **Connection Details** window.

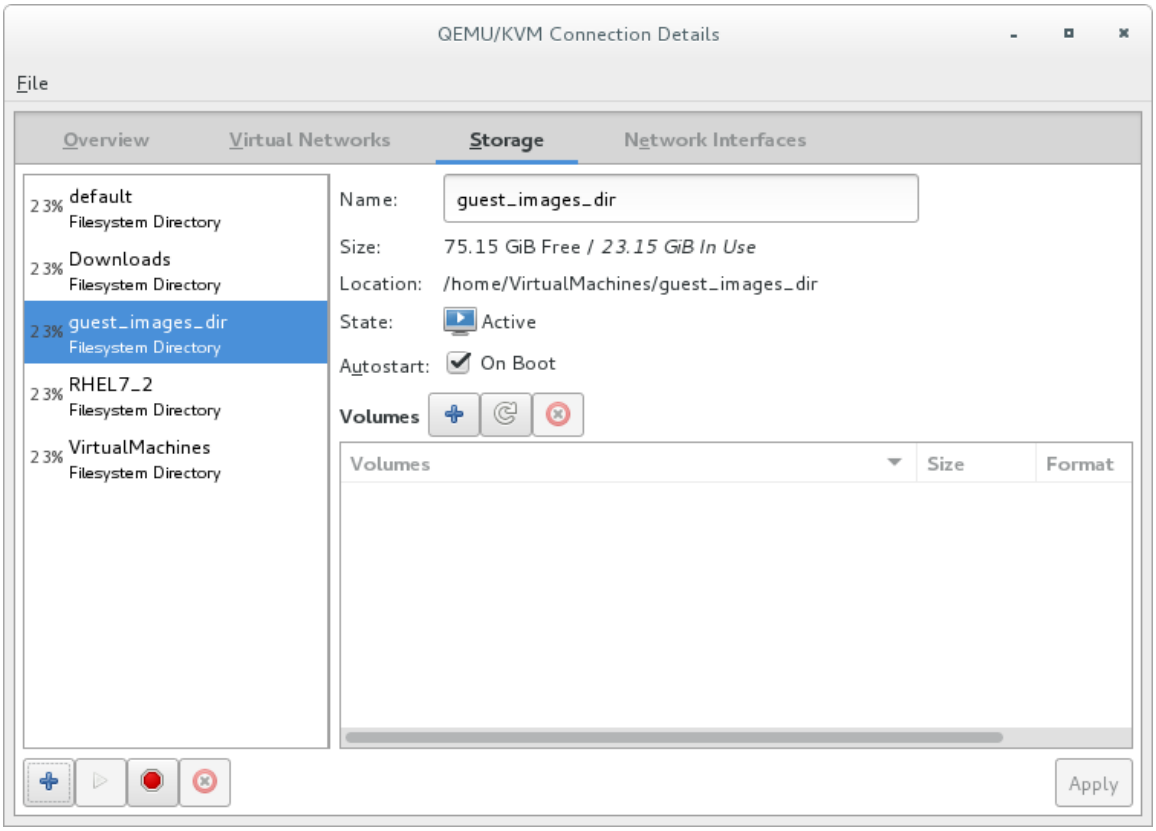


Figure 13.11. Storage tab

The pane on the left of the **Connection Details** window shows a list of storage pools.

2. **Select the storage pool in which you want to create a storage volume**
   In the list of storage pools, click the storage pool in which you want to create the storage volume.

   Any storage volumes configured on the selected storage pool appear in the **Volumes** pane at the bottom of the window.

3. **Add a new storage volume**

   Click the button above the **Volumes** list. The **Add a Storage Volume** dialog appears.

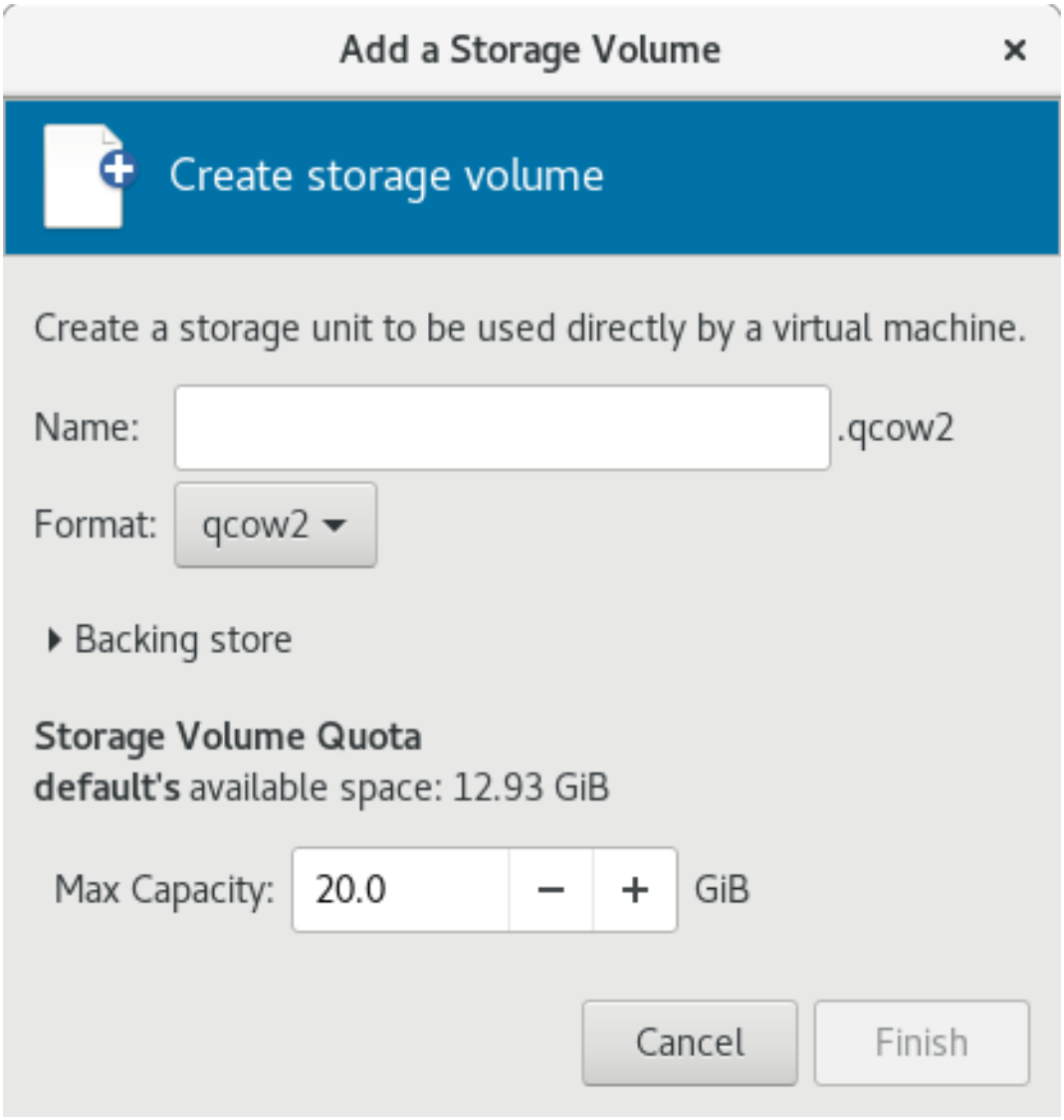


Figure 13.12. Create storage volume

4. **Configure the storage volume**
   Configure the storage volume with the following parameters:
   - Enter a name for the storage pool in the **Name** field.
   - Select a format for the storage volume from the **Format** list.
   - Enter the maximum size for the storage volume in the **Max Capacity** field.
5. **Finish the creation**
   Click **Finish**. The **Add a Storage Volume** dialog closes, and the storage volume appears in the **Volumes** list.

13.3.3. Viewing Storage Volumes

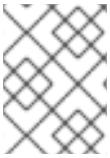
You can create multiple storage volumes from a storage pool. You can also use the **virsh vol-list** command to list the storage volumes in a storage pool. In the following example, the *guest_images_disk* contains three volumes.

```
# virsh vol-create-as guest_images_disk volume1 8G
Vol volume1 created
```

```
# virsh vol-create-as guest_images_disk volume2 8G
Vol volume2 created

# virsh vol-create-as guest_images_disk volume3 8G
Vol volume3 created

# virsh vol-list guest_images_disk
Name                 Path
-----------------------------------------
volume1              /home/VirtualMachines/guest_images_dir/volume1
volume2              /home/VirtualMachines/guest_images_dir/volume2
volume3              /home/VirtualMachines/guest_images_dir/volume3
```

13.3.4. Managing Data

This section provides information about managing the data on storage volumes.



NOTE

Some types of storage volumes do not support all of the data management commands. For specific information, see the sections below.

13.3.4.1. Wiping Storage Volumes

To ensure that data on a storage volume cannot be accessed, a storage volume can be wiped using the **virsh vol-wipe** command.

Use the **virsh vol-wipe** command to wipe a storage volume:

```
# virsh vol-wipe new-vol vdisk
```

By default, the data is overwritten by zeroes. However, there are a number of different methods that can be specified for wiping the storage volume. For detailed information about all of the options for the **virsh vol-wipe** command, refer to Section 20.32, “Deleting a Storage Volume's Contents”.

13.3.4.2. Uploading Data to a Storage Volume

You can upload data from a specified local file to a storage volume using the **virsh vol-upload** command.

```
# virsh vol-upload --pool pool-or-uuid --offset bytes --length bytes vol-
name-or-key-or-path local-file
```

The following are the main **virsh vol-upload** options:

- **--pool *pool-or-uuid*** - The name or UUID of the storage pool the volume is in.
- ***vol-name-or-key-or-path*** - The name or key or path of the volume to upload.
- **--offset *bytes*** The position in the storage volume at which to start writing the data.
- **--length *length*** - An upper limit for the amount of data to be uploaded.

NOTE

An error will occur if *local-file* is greater than the specified **--length**.

Example 13.1. Uploading data to a storage volume

```
# virsh vol-upload sde1 /tmp/data500m.empty disk-pool
```

In this example **sde1** is a volume in the **disk-pool** storage pool. The data in **/tmp/data500m.empty** is copied to **sde1**.

13.3.4.3. Downloading Data to a Storage Volume

You can download data from a storage volume to a specified local file using the **virsh vol-download** command.

```
# vol-download --pool pool-or-uuid --offset bytes --length bytes vol-name-or-key-or-path local-file
```

The following are the main **virsh vol-download** options:

- **--pool *pool-or-uuid*** - The name or UUID of the storage pool that the volume is in.
- ***vol-name-or-key-or-path*** - The name, key, or path of the volume to download.
- **--offset** - The position in the storage volume at which to start reading the data.
- **--length *length*** - An upper limit for the amount of data to be downloaded.

Example 13.2. Downloading data from a storage volume

```
# virsh vol-download sde1 /tmp/data-sde1.tmp disk-pool
```

In this example **sde1** is a volume in the **disk-pool** storage pool. The data in **sde1** is downloaded to **/tmp/data-sde1.tmp**.

13.3.4.4. Resizing Storage Volumes

You can resize the capacity of a specified storage volume using the **vol-resize** command.

```
# virsh vol-resize --pool pool-or-uuid vol-name-or-path pool-or-uuid capacity --allocate --delta --shrink
```

The *capacity* is expressed in bytes. The command requires **--pool *pool-or-uuid*** which is the name or UUID of the storage pool the volume is in. This command also requires *vol-name-or-key-or-path*, the name, key, or path of the volume to resize.

The new capacity might be sparse unless **--allocate** is specified. Normally, capacity is the new size, but if **--delta** is present, then it is added to the existing size. Attempts to shrink the volume will fail unless **--shrink** is present.

Note that *capacity* cannot be negative unless **--shrink** is provided and a negative sign is not necessary. *capacity* is a scaled integer which defaults to bytes if there is no suffix. In addition, note that this command is only safe for storage volumes not in use by an active guest. Refer to Section 20.13.3, "Changing the Size of a Guest Virtual Machine's Block Device" for live resizing.

Example 13.3. Resizing a storage volume

For example, if you created a 50M storage volume, you can resize it to 100M with the following command:

```
# virsh vol-resize --pool disk-pool sde1 100M
```

13.3.5. Deleting Storage Volumes

You can delete storage volumes using **virsh** or the **Virtual Machine Manager**.

NOTE

To avoid negatively affecting guest virtual machines that use the storage volume you want to delete, it is recommended that you release any resources using it.

13.3.5.1. Deleting storage volumes using virsh

Delete a storage volume using the **virsh vol-delete** command. The command must specify the name or path of the storage volume and the storage pool from which the storage volume is abstracted.

The following example deletes the *volume_name* storage volume from the *guest_images_dir* storage pool:

```
# virsh vol-delete volume_name --pool guest_images_dir
  vol volume_name deleted
```

13.3.5.2. Deleting storage volumes using Virtual Machine Manager

Procedure 13.12. Deleting Storage Volumes with Virtual Machine Manager

1. **Open the storage settings**
   a. In the **Virtual Machine Manager**, open the **Edit** menu and select **Connection Details**.
   b. Click the **Storage** tab in the **Connection Details** window.

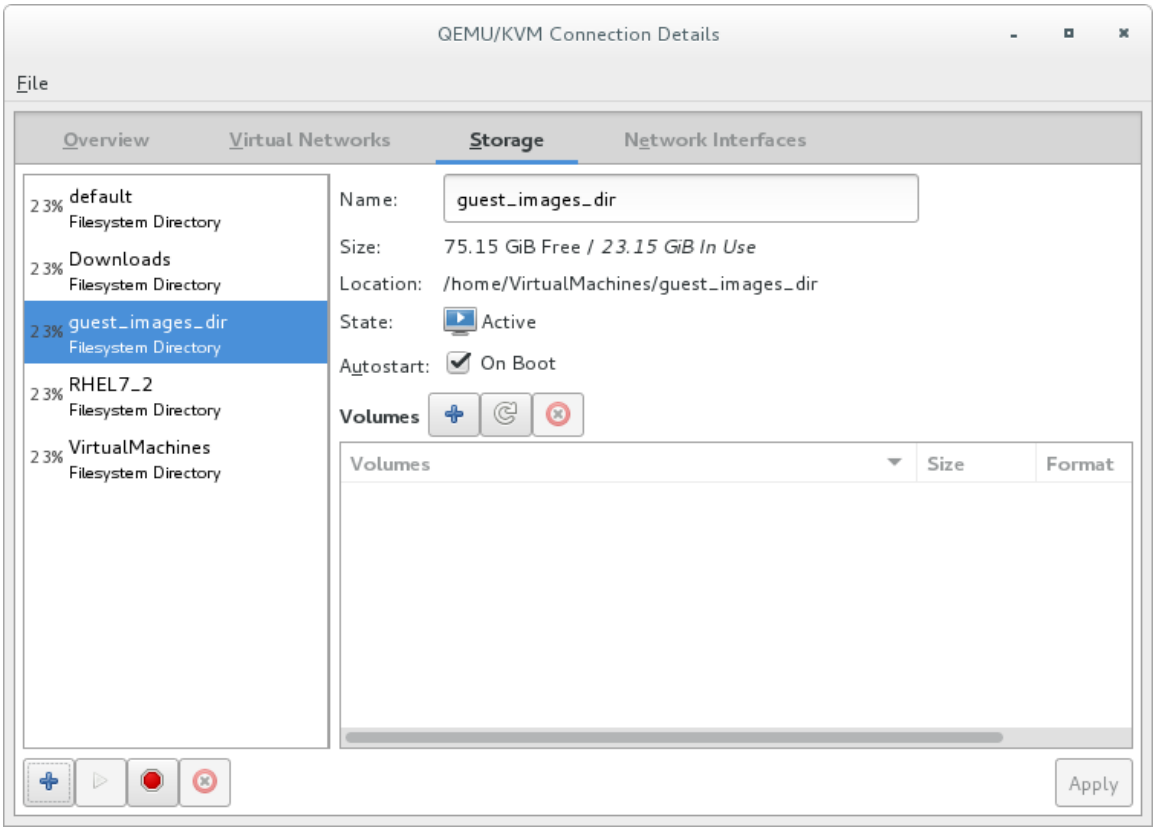


Figure 13.13. Storage tab

The pane on the left of the **Connection Details** window shows a list of storage pools.

2. **Select the storage volume you want to delete**

   a. In the list of storage pools, click the storage pool from which the storage volume is abstracted.

      A list of storage volumes configured on the selected storage pool appear in the **Volumes** pane at the bottom of the window.

   b. Select the storage volume you want to delete.

3. **Delete the storage volume**

   a. Click the button (above the **Volumes** list). A confirmation dialog appears.

   b. Click **Yes**. The selected storage volume is deleted.

13.3.6. Adding Storage Devices to Guests

You can add storage devices to guest virtual machines using **virsh** or **Virtual Machine Manager**.

13.3.6.1. Adding Storage Devices to Guests Using virsh

To add storage devices to a guest virtual machine, use the **attach-disk** command. The arguments that contain information about the disk to add can be specified in an XML file or on the command line.

The following is a sample XML file with the definition of the storage.

```
<disk type='file' device='disk'>'>
  <driver name='qemu' type='raw' cache='none'/>
  <source file='/var/lib/libvirt/images/FileName.img'/>
  <target dev='vdb' bus='virtio'/>
</disk>
```

The following command attaches a disk to *Guest1* using an XML file called **NewStorage.xml**.

```
# virsh attach-disk --config Guest1 ~/NewStorage.xml
```

The following command attaches a disk to *Guest1* without using an xml file.

```
# virsh attach-disk --config Guest1 --source
/var/lib/libvirt/images/FileName.img --target vdb
```

13.3.6.2. Adding Storage Devices to Guests Using Virtual Machine Manager

You can add a storage volume to a guest virtual machine or create and add a default storage device to a guest virtual machine.

13.3.6.2.1. Adding a storage volume to a guest

To add a storage volume to a guest virtual machine:

1. **Open Virtual Machine Manager to the virtual machine hardware details window**
   Open virt-manager by executing the **virt-manager** command as root or opening **Applications** → **System Tools** → **Virtual Machine Manager**.

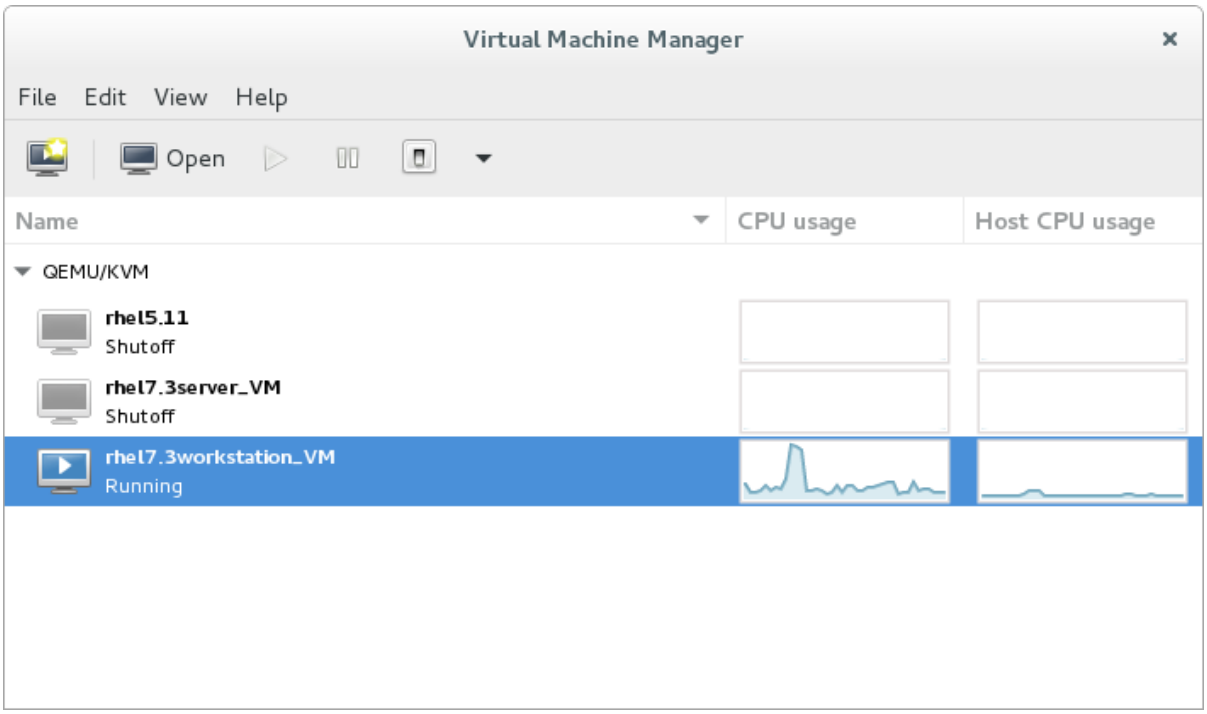


Figure 13.14. The Virtual Machine Manager window

Select the guest virtual machine to which you want to add a storage volume.

Click **Open**. The Virtual Machine window opens.

Click . The hardware details window appears.

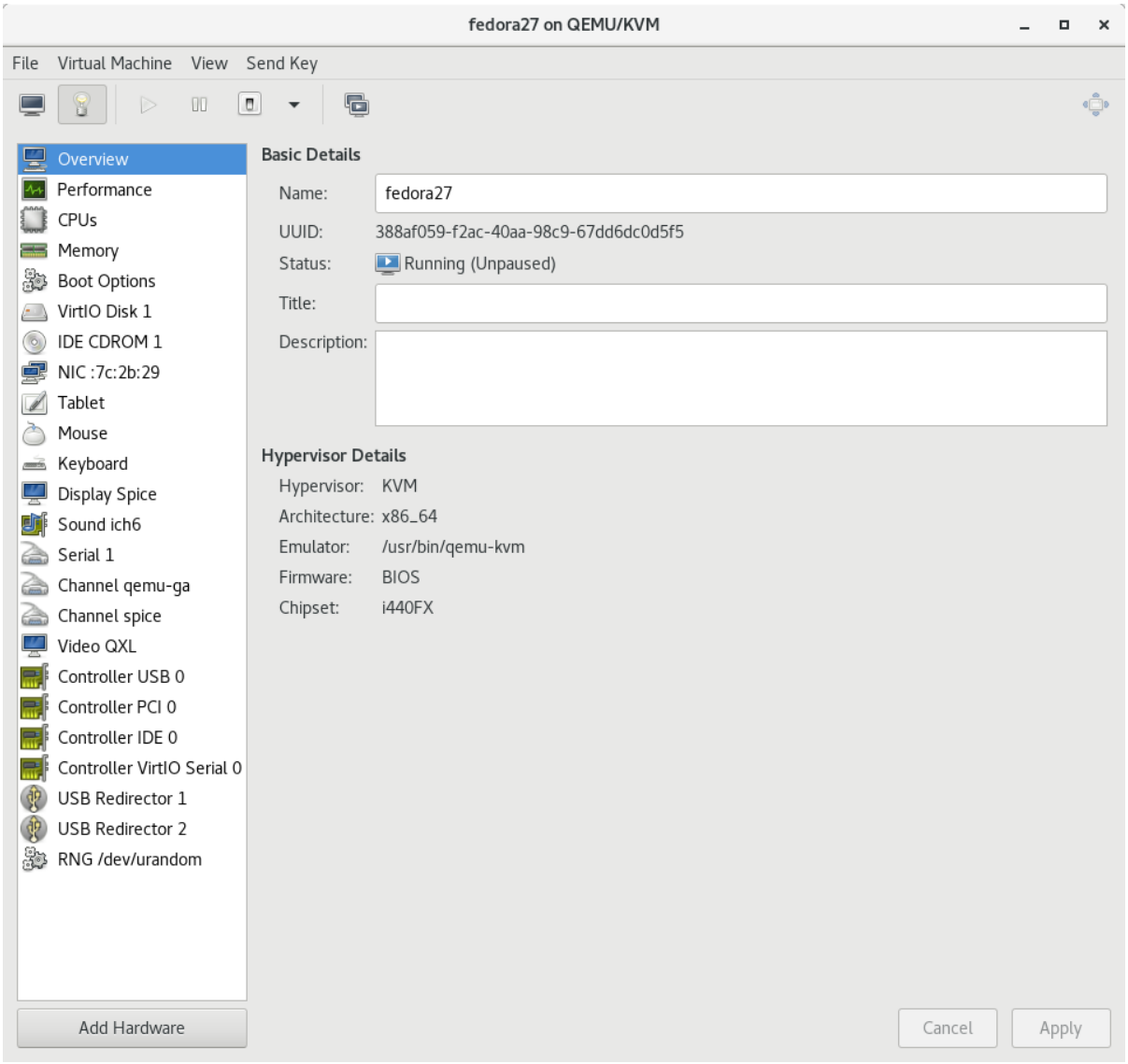


Figure 13.15. The Hardware Details window

2. **Open the Add New Virtual Hardware window**
   Click **Add Hardware**. The Add New Virtual Hardware window appears.

   Ensure that **Storage** is selected in the hardware type pane.

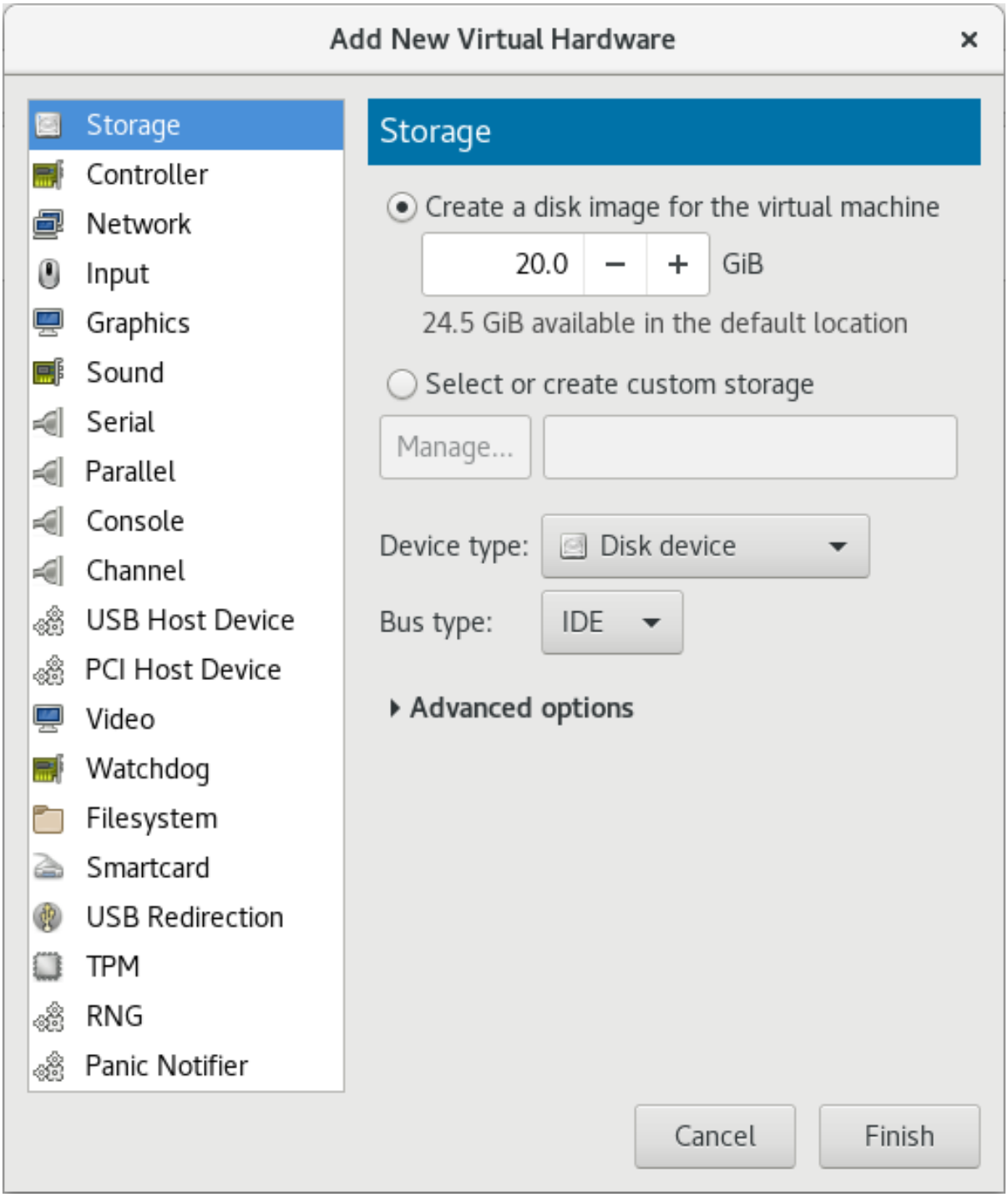


Figure 13.16. The Add New Virtual Hardware window

3. **View a list of storage volumes**
   Select the **`Select or create custom storage`** option button.

   Click **`Manage`**. The Choose Storage Volume dialog appears.

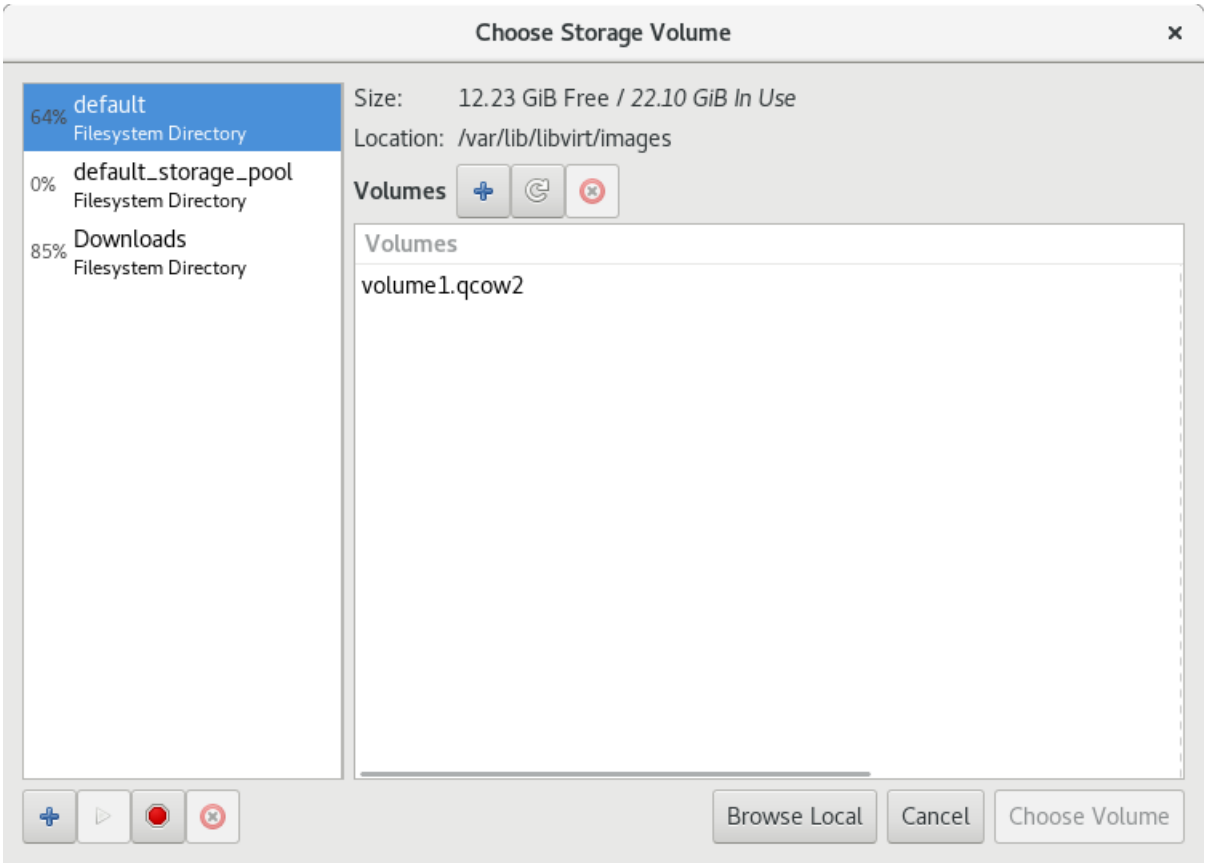


Figure 13.17. The Select Storage Volume window

4. **Select a storage volume**
   Select a storage pool from the list on the left side of the Select Storage Volume window. A list of storage volumes in the selected storage pool appears in the **Volumes** list.

   NOTE

   You can create a storage pool from the Select Storage Volume window. For more information, see Section 13.2.2.2, "Creating storage pools with Virtual Machine Manager".

   Select a storage volume from the **Volumes** list.

   NOTE

   You can create a storage volume from the Select Storage Volume window. For more information, see Section 13.3.2.2, "Creating storage volumes with Virtual Machine Manager".

   Click **Choose Volume**. The Select Storage Volume window closes.

5. **Configure the storage volume**
   Select a device type from the **Device type** list. Available types are: Disk device, CDROM device, Floppy device, LUN Passthrough

   Select a bus type from the **Bus type** list. The available bus types are dependent on the selected device type.

Select a cache mode from the **Cache mode** list. Available cache modes are: Hypervisor default, none, writethrough, writeback, directsync, unsafe

Click **Finish**. The Add New Virtual Hardware window closes.

13.3.6.2.2. Adding default storage to a guest

The default storage pool is a file-based image in **/var/lib/libvirt/images/** directory.

To add default storage to a guest virtual machine:

1. **Open Virtual Machine Manager to the virtual machine hardware details window**
   Open virt-manager by executing the **virt-manager** command as root or opening **Applications** → **System Tools** → **Virtual Machine Manager**.

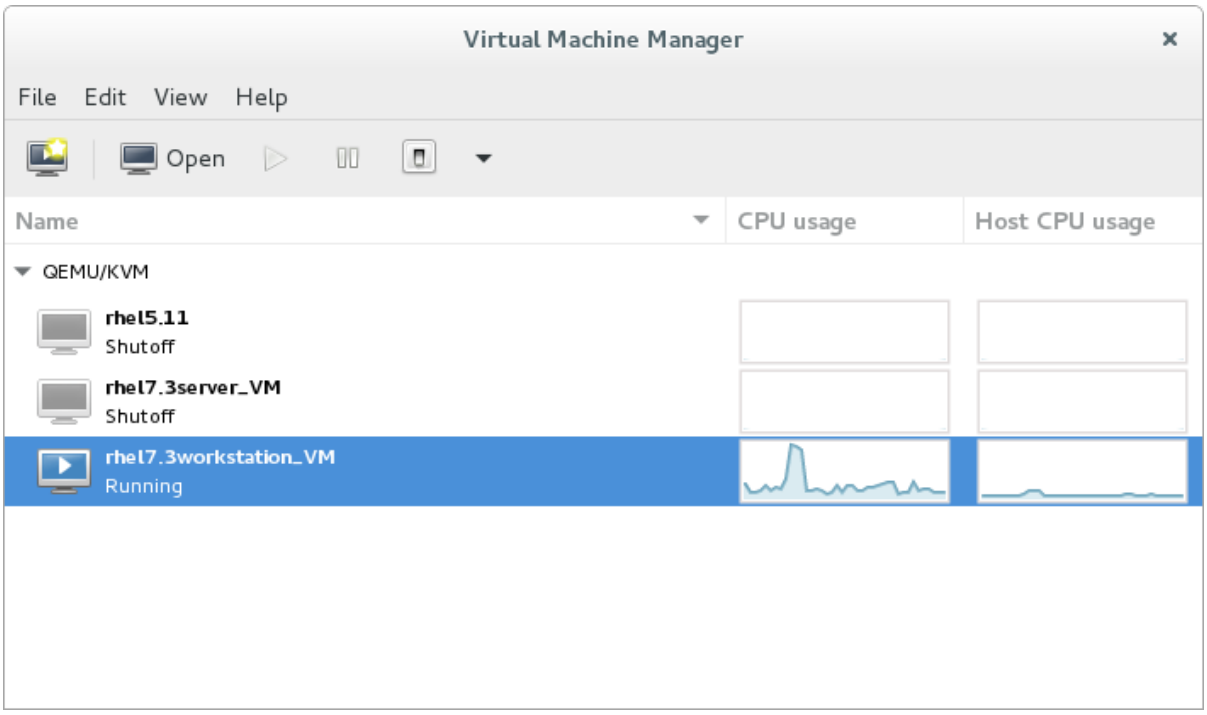


Figure 13.18. The Virtual Machine Manager window

Select the guest virtual machine to which you want to add a storage volume.

Click **Open**. The Virtual Machine window opens.

Click . The hardware details window appears.

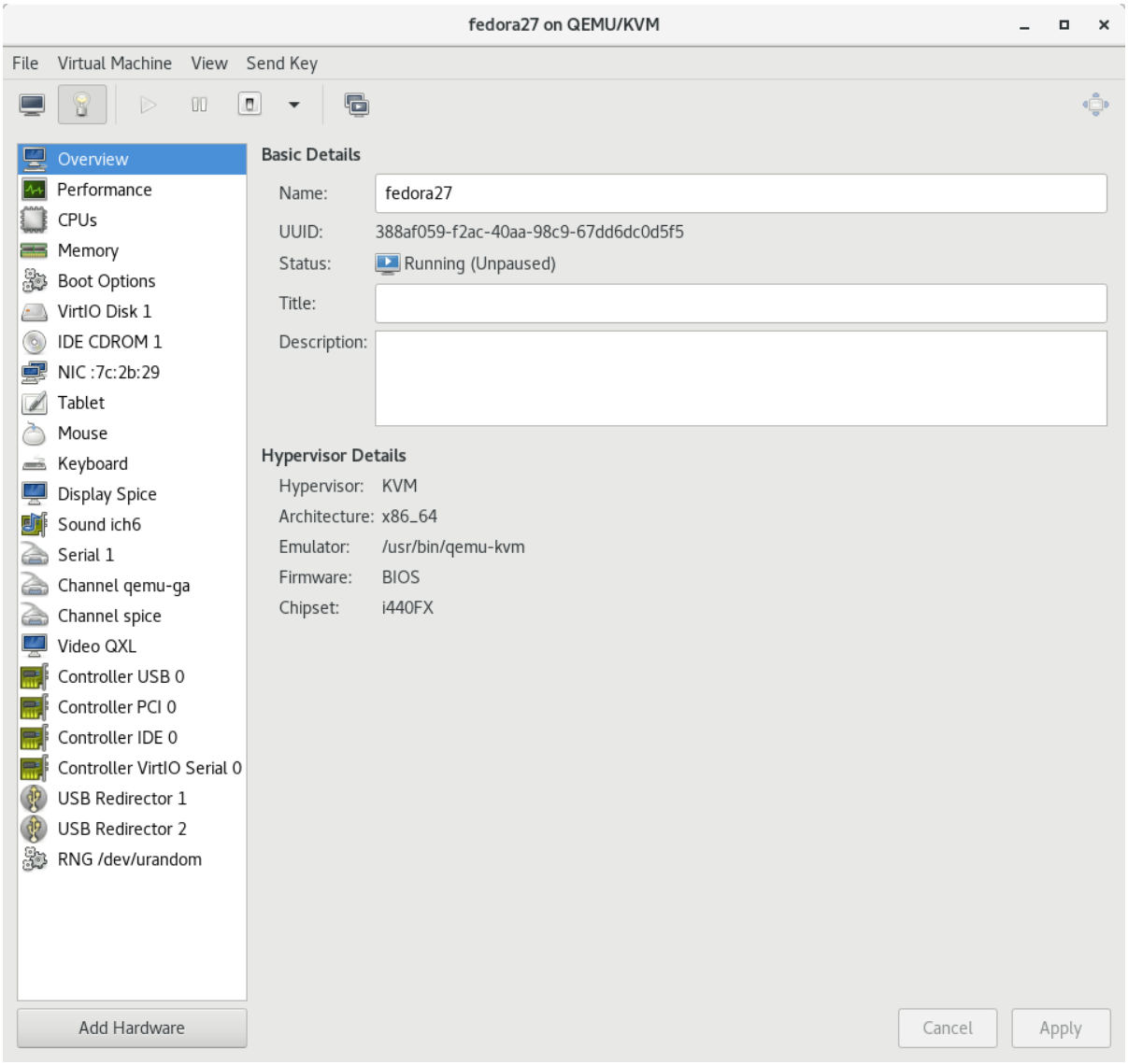


Figure 13.19. The Hardware Details window

2. **Open the Add New Virtual Hardware window**
   Click **Add Hardware**. The Add New Virtual Hardware window appears.

   Ensure that **Storage** is selected in the hardware type pane.

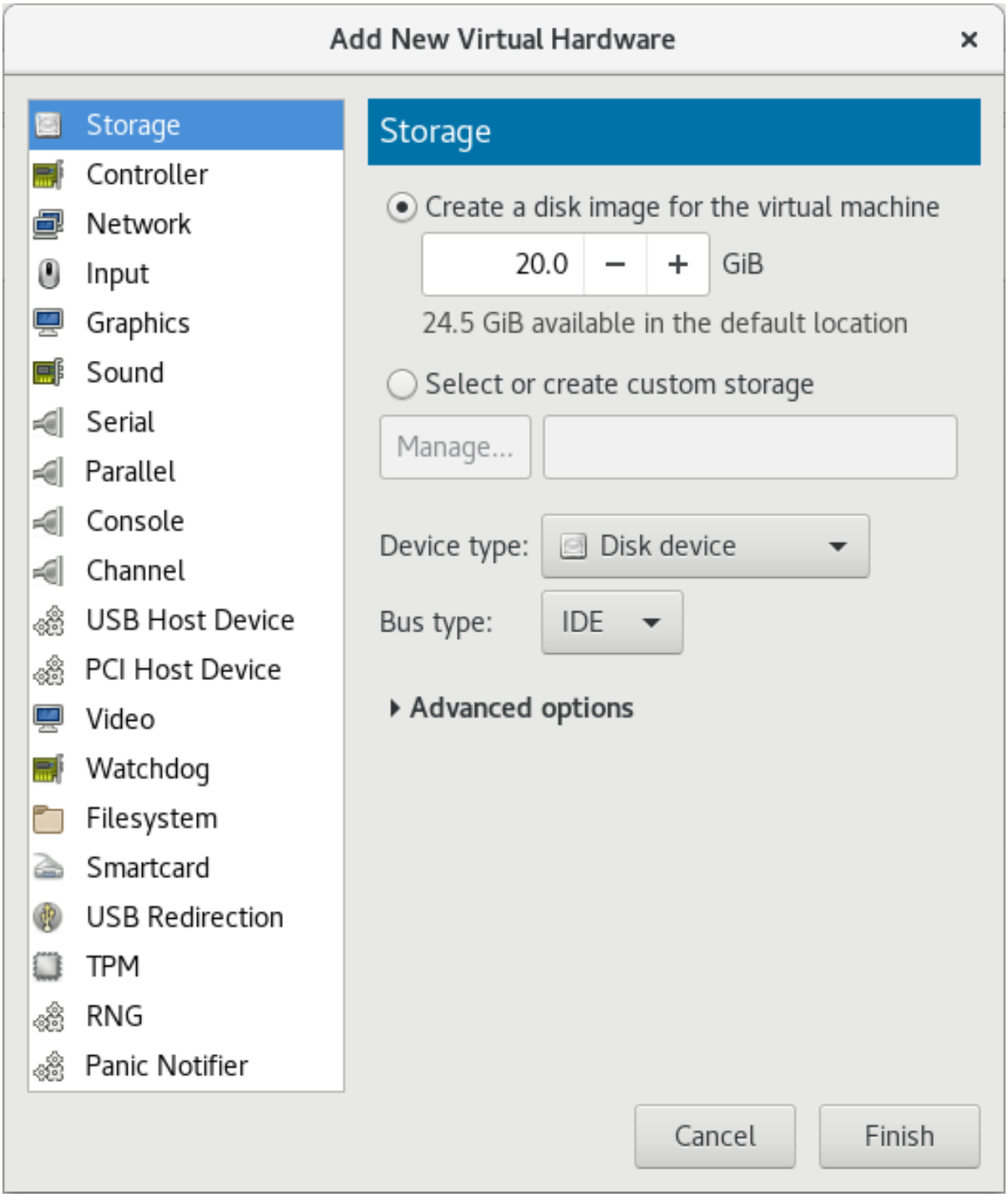


Figure 13.20. The Add New Virtual Hardware window

3. **Create a disk for the guest**
   Ensure that the **`Create a disk image for the virtual machine`** option.

   Enter the size of the disk to create in the textbox below the **`Create a disk image for the virtual machine`** option button.

   Click **`Finish`**. The Add New Virtual Hardware window closes.

13.3.6.3. Adding SCSI LUN-based Storage to a Guest

There are multiple ways to expose a host SCSI LUN entirely to the guest. Exposing the SCSI LUN to the guest provides the capability to execute SCSI commands directly to the LUN on the guest. This is useful as a means to share a LUN between guests, as well as to share Fibre Channel storage between hosts.

For more information on SCSI LUN-based storage, see vHBA-based storage pools using SCSI devices.

IMPORTANT

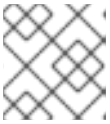
The optional ***sgio*** attribute controls whether unprivileged SCSI Generical I/O (SG_IO) commands are filtered for a **device='lun'** disk. The ***sgio*** attribute can be specified as **'filtered'** or **'unfiltered'**, but must be set to **'unfiltered'** to allow SG_IO **ioctl** commands to be passed through on the guest in a persistent reservation.

In addition to setting **sgio='unfiltered'**, the ***<shareable>*** element must be set to share a LUN between guests. The ***sgio*** attribute defaults to **'filtered'** if not specified.

The ***<disk>*** XML attribute **device='lun'** is valid for the following guest disk configurations:

- **type='block'** for **<source dev='/dev/disk/by-{path|id|uuid|label}'/>**

```
<disk type='block' device='lun' sgio='unfiltered'>
  <driver name='qemu' type='raw'/>
  <source dev='/dev/disk/by-path/pci-0000\:04\:00.1-fc-
0x203400a0b85ad1d7-lun-0'/>
  <target dev='sda' bus='scsi'/>
  <shareable/>
</disk>
```



NOTE

The backslashes prior to the colons in the ***<source>*** device name are required.

- **type='network'** for **<source protocol='iscsi'... />**

```
<disk type='network' device='lun' sgio='unfiltered'>
  <driver name='qemu' type='raw'/>
  <source protocol='iscsi' name='iqn.2013-07.com.example:iscsi-net-
pool/1'>
    <host name='example.com' port='3260'/>
    <auth username='myuser'>
      <secret type='iscsi' usage='libvirtiscsi'/>
    </auth>
  </source>
  <target dev='sda' bus='scsi'/>
  <shareable/>
</disk>
```

- **type='volume'** when using an iSCSI or a NPIV/vHBA source pool as the SCSI source pool.

  The following example XML shows a guest using an iSCSI source pool (named *iscsi-net-pool*) as the SCSI source pool:

```
<disk type='volume' device='lun' sgio='unfiltered'>
  <driver name='qemu' type='raw'/>
  <source pool='iscsi-net-pool' volume='unit:0:0:1' mode='host'/>
```

```
  <target dev='sda' bus='scsi'/>
  <shareable/>
</disk>
```

NOTE

The **mode=** option within the **<source>** tag is optional, but if used, it must be set to **'host'** and not **'direct'**. When set to **'host'**, libvirt will find the path to the device on the local host. When set to **'direct'**, libvirt will generate the path to the device using the source pool's source host data.

The iSCSI pool (*iscsi-net-pool*) in the example above will have a similar configuration to the following:

```
# virsh pool-dumpxml iscsi-net-pool
<pool type='iscsi'>
  <name>iscsi-net-pool</name>
  <capacity unit='bytes'>11274289152</capacity>
  <allocation unit='bytes'>11274289152</allocation>
  <available unit='bytes'>0</available>
  <source>
    <host name='192.168.122.1' port='3260'/>
    <device path='iqn.2013-12.com.example:iscsi-chap-netpool'/>
    <auth type='chap' username='redhat'>
      <secret usage='libvirtiscsi'/>
    </auth>
  </source>
  <target>
    <path>/dev/disk/by-path</path>
    <permissions>
      <mode>0755</mode>
    </permissions>
  </target>
</pool>
```

To verify the details of the available LUNs in the iSCSI source pool, enter the following command:

```
# virsh vol-list iscsi-net-pool
Name                 Path
-----------------------------------------------------------------
--------------
unit:0:0:1           /dev/disk/by-path/ip-192.168.122.1:3260-iscsi-
iqn.2013-12.com.example:iscsi-chap-netpool-lun-1
unit:0:0:2           /dev/disk/by-path/ip-192.168.122.1:3260-iscsi-
iqn.2013-12.com.example:iscsi-chap-netpool-lun-2
```

- **type='volume'** when using a NPIV/vHBA source pool as the SCSI source pool.

  The following example XML shows a guest using a NPIV/vHBA source pool (named *vhbapool_host3*) as the SCSI source pool:

```
<disk type='volume' device='lun' sgio='unfiltered'>
  <driver name='qemu' type='raw'/>
```

```
  <source pool='vhbapool_host3' volume='unit:0:1:0'/>
  <target dev='sda' bus='scsi'/>
  <shareable/>
</disk>
```

The NPIV/vHBA pool (*vhbapool_host3*) in the example above will have a similar configuration to:

```
# virsh pool-dumpxml vhbapool_host3
<pool type='scsi'>
  <name>vhbapool_host3</name>
  <capacity unit='bytes'>0</capacity>
  <allocation unit='bytes'>0</allocation>
  <available unit='bytes'>0</available>
  <source>
    <adapter type='fc_host' parent='scsi_host3' managed='yes'
wwnn='5001a4a93526d0a1' wwpn='5001a4ace3ee045d'/>
  </source>
  <target>
    <path>/dev/disk/by-path</path>
    <permissions>
      <mode>0700</mode>
      <owner>0</owner>
      <group>0</group>
    </permissions>
  </target>
</pool>
```

To verify the details of the available LUNs on the vHBA, enter the following command:

```
# virsh vol-list vhbapool_host3
Name                 Path
-----------------------------------------------------------------
--------------
unit:0:0:0           /dev/disk/by-path/pci-0000:10:00.0-fc-
0x5006016044602198-lun-0
unit:0:1:0           /dev/disk/by-path/pci-0000:10:00.0-fc-
0x5006016844602198-lun-0
```

For more information on using a NPIV vHBA with SCSI devices, see Section 13.2.3.8, “vHBA-based storage pools using SCSI devices”.

The following procedure shows an example of adding a SCSI LUN-based storage device to a guest. Any of the above ***<disk device='lun'>*** guest disk configurations can be attached with this method. Substitute configurations according to your environment.

Procedure 13.13. Attaching SCSI LUN-based storage to a guest

1. Create the device file by writing a <disk> element in a new file, and save this file with an XML extension (in this example, *sda.xml*):

```
# cat sda.xml
<disk type='volume' device='lun' sgio='unfiltered'>
  <driver name='qemu' type='raw'/>
  <source pool='vhbapool_host3' volume='unit:0:1:0'/>
```

```
  <target dev='sda' bus='scsi'/>
  <shareable/>
</disk>
```

2. Associate the device created in *sda.xml* with your guest virtual machine (*Guest1*, for example):

```
# virsh attach-device --config Guest1 ~/sda.xml
```

NOTE

Running the **virsh attach-device** command with the **--config** option requires a guest reboot to add the device permanently to the guest. Alternatively, the **--persistent** option can be used instead of **--config**, which can also be used to hotplug the device to a guest.

Alternatively, the SCSI LUN-based storage can be attached or configured on the guest using **virt-manager**. To configure this using **virt-manager**, click the **Add Hardware** button and add a virtual disk with the required parameters, or change the settings of an existing SCSI LUN device from this window. In Red Hat Enterprise Linux 7.2 and above, the SGIO value can also be configured in **virt-manager**:

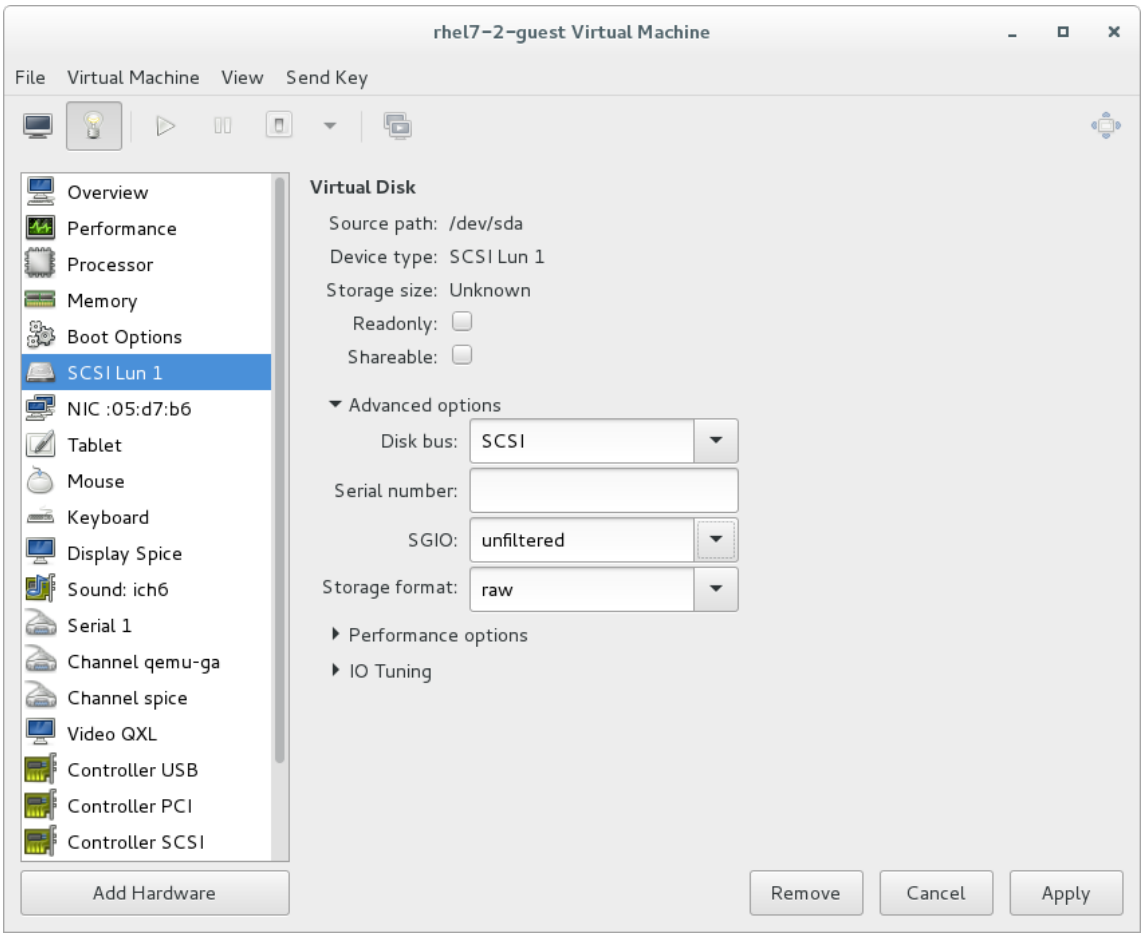


Figure 13.21. Configuring SCSI LUN storage with virt-manager

Reconnecting to an exposed LUN after a hardware failure

If the connection to an exposed Fiber Channel (FC) LUN is lost due to a failure of hardware (such as the host bus adapter), the exposed LUNs on the guest may continue to appear as failed even after the hardware failure is fixed. To prevent this, edit the **dev_loss_tmo** and **fast_io_fail_tmo** kernel options:

- **dev_loss_tmo** controls how long the SCSI layer waits after a SCSI device fails before marking it as failed. To prevent a timeout, it is recommended to set the option to the maximum value, which is **2147483647**.

- **fast_io_fail_tmo** controls how long the SCSI layer waits after a SCSI device fails before failing back to the I/O. To ensure that **dev_loss_tmo** is not ignored by the kernel, set this option's value to any number lower than the value of **dev_loss_tmo**.

To modify the value of **dev_loss_tmo** and **fast_io_fail**, do one of the following:

- Edit the **/etc/multipath.conf** file, and set the values in the **defaults** section:

```
defaults {
...
fast_io_fail_tmo     20
dev_loss_tmo    infinity
}
```

- Set **dev_loss_tmo** and **fast_io_fail** on the level of the FC host or remote port, for example as follows:

```
# echo 20 >
/sys/devices/pci0000:00/0000:00:06.0/0000:13:00.0/host1/rport-1:0-
0/fc_remote_ports/rport-1:0-0/fast_io_fail_tmo
# echo 2147483647 >
/sys/devices/pci0000:00/0000:00:06.0/0000:13:00.0/host1/rport-1:0-
0/fc_remote_ports/rport-1:0-0/dev_loss_tmo
```

To verify that the new values of **dev_loss_tmo** and **fast_io_fail** are active, use the following command:

```
# find /sys -name dev_loss_tmo -print -exec cat {} \;
```

If the parameters have been set correctly, the output will look similar to this, with the appropriate device or devices instead of **pci0000:00/0000:00:06.0/0000:13:00.0/host1/rport-1:0-0/fc_remote_ports/rport-1:0-0**:

```
# find /sys -name dev_loss_tmo -print -exec cat {} \;
...
/sys/devices/pci0000:00/0000:00:06.0/0000:13:00.0/host1/rport-1:0-
0/fc_remote_ports/rport-1:0-0/dev_loss_tmo
2147483647
...
```

13.3.6.4. Managing Storage Controllers in a Guest Virtual Machine

Unlike virtio disks, SCSI devices require the presence of a controller in the guest virtual machine. This section details the necessary steps to create a virtual SCSI controller (also known as "Host Bus Adapter", or HBA), and to add SCSI storage to the guest virtual machine.

Procedure 13.14. Creating a virtual SCSI controller

1. Display the configuration of the guest virtual machine (**Guest1**) and look for a pre-existing SCSI controller:

```
# virsh dumpxml Guest1 | grep controller.*scsi
```

If a device controller is present, the command will output one or more lines similar to the following:

```
<controller type='scsi' model='virtio-scsi' index='0'/>
```

2. If the previous step did not show a device controller, create the description for one in a new file and add it to the virtual machine, using the following steps:

a. Create the device controller by writing a **<controller>** element in a new file and save this file with an XML extension. **virtio-scsi-controller.xml**, for example.

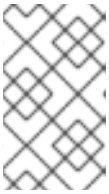
```
<controller type='scsi' model='virtio-scsi'/>
```

b. Associate the device controller you just created in **virtio-scsi-controller.xml** with your guest virtual machine (Guest1, for example):

```
# virsh attach-device --config Guest1 ~/virtio-scsi-
controller.xml
```

In this example the **--config** option behaves the same as it does for disks. See Section 13.3.6, “Adding Storage Devices to Guests” for more information.

3. Add a new SCSI disk or CD-ROM. The new disk can be added using the methods in Section 13.3.6, “Adding Storage Devices to Guests”. In order to create a SCSI disk, specify a target device name that starts with *sd*.



NOTE

The supported limit for each controller is 1024 virtio-scsi disks, but it is possible that other available resources in the host (such as file descriptors) are exhausted with fewer disks.

For more information, see the following Red Hat Enterprise Linux 6 whitepaper: The next-generation storage interface for the Red Hat Enterprise Linux Kernel Virtual Machine: virtio-scsi.

```
# virsh attach-disk Guest1 /var/lib/libvirt/images/FileName.img sdb
--cache none
```

Depending on the version of the driver in the guest virtual machine, the new disk may not be detected immediately by a running guest virtual machine. Follow the steps in the *Red Hat Enterprise Linux Storage Administration Guide*.

13.3.7. Removing Storage Devices from Guests

You can remove storage device from virtual guest machines using **virsh** or **Virtual Machine Manager**.

13.3.7.1. Removing Storage from a Virtual Machine with virsh

The following example removes the vdb storage volume from the Guest1 virtual machine:

```
# virsh detach-disk Guest1 vdb
```

13.3.7.2. Removing Storage from a Virtual Machine with Virtual Machine Manager

Procedure 13.15. Removing storage from a virtual machine with Virtual Machine Manager

To remove storage from a guest virtual machine using **Virtual Machine Manager**:

1. **Open Virtual Machine Manager to the virtual machine hardware details window**
   Open virt-manager by executing the **`virt-manager`** command as root or opening **`Applications`** → **`System Tools`** → **`Virtual Machine Manager`**.

   Select the guest virtual machine from which you want to remove a storage device.

   Click **`Open`**. The Virtual Machine window opens.

   Click . The hardware details window appears.

2. **Remove the storage from the guest virtual machine**
   Select the storage device from the list of hardware on the left side of the hardware details pane.

   Click **`Remove`**. A confirmation dialog appears.

   Click **`Yes`**. The storage is removed from the guest virtual machine.

CHAPTER 14. USING QEMU-IMG

The qemu-img command-line tool is used for formatting, modifying, and verifying various file systems used by KVM. qemu-img options and usages are highlighted in the sections that follow.

WARNING

Never use qemu-img to modify images in use by a running virtual machine or any other process. This may destroy the image. Also, be aware that querying an image that is being modified by another process may encounter inconsistent state.

14.1. CHECKING THE DISK IMAGE

To perform a consistency check on a disk image with the file name *imgname*.

```
# qemu-img check [-f format] imgname
```

NOTE

Only a selected group of formats support consistency checks. These include *qcow2*, *vdi*, *vhdx*, *vmdk*, and *qed*.

14.2. COMMITTING CHANGES TO AN IMAGE

Commit any changes recorded in the specified image file (*imgname*) to the file's base image with the **qemu-img commit** command. Optionally, specify the file's format type (*fmt*).

```
# qemu-img commit [-f fmt] [-t cache] imgname
```

14.3. COMPARING IMAGES

Compare the contents of two specified image files (*imgname1* and *imgname2*) with the **qemu-img compare** command. Optionally, specify the files' format types (*fmt*). The images can have different formats and settings.

By default, images with different sizes are considered identical if the larger image contains only unallocated or zeroed sectors in the area after the end of the other image. In addition, if any sector is not allocated in one image and contains only zero bytes in the other one, it is evaluated as equal. If you specify the **-s** option, the images are not considered identical if the image sizes differ or a sector is allocated in one image and is not allocated in the second one.

```
# qemu-img compare [-f fmt] [-F fmt] [-p] [-s] [-q] imgname1 imgname2
```

The **qemu-img compare** command exits with one of the following exit codes:

- **0** - The images are identical

- **1** - The images are different
- **2** - There was an error opening one of the images
- **3** - There was an error checking a sector allocation
- **4** - There was an error reading the data

14.4. MAPPING AN IMAGE

Using the **qemu-img map** command, your can dump the metadata of the specified image file (*imgname*) and its backing file chain. The dump shows the allocation state of every sector in the (*imgname*) with the topmost file that allocates it in the backing file chain. Optionally, specify the file's format type (*fmt*).

```
# qemu-img map [-f fmt] [--output=fmt] imgname
```

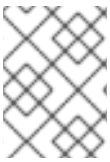
There are two output formats, the **human** format and the **json** format:

14.4.1. The human Format

The default format (**human**) only dumps non-zero, allocated parts of the file. The output identifies a file from where data can be read and the offset in the file. Each line includes four fields. The following shows an example of an output:

```
Offset          Length          Mapped to       File
0               0x20000         0x50000         /tmp/overlay.qcow2
0x100000        0x10000         0x95380000      /tmp/backing.qcow2
```

The first line means that **0x20000** (131072) bytes starting at offset 0 in the image are available in **tmp/overlay.qcow2** (opened in raw format) starting at offset **0x50000** (327680). Data that is compressed, encrypted, or otherwise not available in raw format causes an error if **human** format is specified.



NOTE

File names can include newline characters. Therefore, it is not safe to parse output in **human** format in scripts.

14.4.2. The json Format

If the **json** option is specified, the output returns an array of dictionaries in JSON format. In addition to the information provided in the **human** option, the output includes the following information:

- **data** - A Boolean field that shows whether or not the sectors contain data
- **zero** - A Boolean field that shows whether or not the data is known to read as zero
- **depth** - The depth of the backing file of **filename**

NOTE

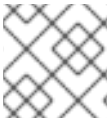
When the **json** option is specified, the **offset** field is optional.

For more information about the **qemu-img map** command and additional options, see the relevant man page.

14.5. AMENDING AN IMAGE

Amend the image format-specific options for the image file. Optionally, specify the file's format type (*fmt*).

```
# qemu-img amend [-p] [-f fmt] [-t cache] -o options filename
```



NOTE

This operation is only supported for the qcow2 file format.

14.6. CONVERTING AN EXISTING IMAGE TO ANOTHER FORMAT

The ***convert*** option is used to convert one recognized image format to another image format. For a list of accepted formats, see Section 14.12, “Supported qemu-img Formats”.

```
# qemu-img convert [-c] [-p] [-f fmt] [-t cache] [-O output_fmt] [-o
options] [-S sparse_size] filename output_filename
```

The ***-p*** parameter shows the progress of the command (optional and not for every command) and ***-S*** flag allows for the creation of a *sparse file*, which is included within the disk image. Sparse files in all purposes function like a standard file, except that the physical blocks that only contain zeros (that is, nothing). When the Operating System sees this file, it treats it as it exists and takes up actual disk space, even though in reality it does not take any. This is particularly helpful when creating a disk for a guest virtual machine as this gives the appearance that the disk has taken much more disk space than it has. For example, if you set -S to 50Gb on a disk image that is 10Gb, then your 10Gb of disk space will appear to be 60Gb in size even though only 10Gb is actually being used.

Convert the disk image ***filename*** to disk image ***output_filename*** using format ***output_format***. The disk image can be optionally compressed with the ***-c*** option, or encrypted with the ***-o*** option by setting **-o encryption**. Note that the options available with the ***-o*** parameter differ with the selected format.

Only the **qcow2** and *qcow2* format supports encryption or compression. **qcow2** encryption uses the AES format with secure 128-bit keys. **qcow2** compression is read-only, so if a compressed sector is converted from **qcow2** format, it is written to the new format as uncompressed data.

Image conversion is also useful to get a smaller image when using a format which can grow, such as **qcow** or **cow**. The empty sectors are detected and suppressed from the destination image.

14.7. CREATING AND FORMATTING NEW IMAGES OR DEVICES

Create the new disk image *filename* of size ***size*** and format ***format***.

```
# qemu-img create [-f format] [-o options] filename [size]
```

If a base image is specified with **-o backing_file=*filename***, the image will only record differences between itself and the base image. The backing file will not be modified unless you use the **commit** command. No size needs to be specified in this case.

14.8. DISPLAYING IMAGE INFORMATION

The **info** parameter displays information about a disk image *filename*. The format for the **info** option is as follows:

```
# qemu-img info [-f format] filename
```

This command is often used to discover the size reserved on disk which can be different from the displayed size. If snapshots are stored in the disk image, they are displayed also. This command will show for example, how much space is being taken by a qcow2 image on a block device. This is done by running the **qemu-img**. You can check that the image in use is the one that matches the output of the **qemu-img info** command with the **qemu-img check** command.

```
# qemu-img info /dev/vg-90.100-sluo/lv-90-100-sluo
image: /dev/vg-90.100-sluo/lv-90-100-sluo
file format: qcow2
virtual size: 20G (21474836480 bytes)
disk size: 0
cluster_size: 65536
```

14.9. REBASING A BACKING FILE OF AN IMAGE

The **qemu-img rebase** changes the backing file of an image.

```
# qemu-img rebase [-f fmt] [-t cache] [-p] [-u] -b backing_file [-F
backing_fmt] filename
```

The backing file is changed to *backing_file* and (if the format of *filename* supports the feature), the backing file format is changed to *backing_format*.

NOTE

Only the *qcow2* format supports changing the backing file (rebase).

There are two different modes in which *rebase* can operate: **safe** and **unsafe**.

safe mode is used by default and performs a real rebase operation. The new backing file may differ from the old one and the **qemu-img rebase** command will take care of keeping the guest virtual machine-visible content of *filename* unchanged. In order to achieve this, any clusters that differ between *backing_file* and old backing file of *filename* are merged into *filename* before making any changes to the backing file.

Note that **safe** mode is an expensive operation, comparable to converting an image. The old backing file is required for it to complete successfully.

unsafe mode is used if the *-u* option is passed to **qemu-img rebase**. In this mode, only the backing file name and format of *filename* is changed, without any checks taking place on the file contents. Make sure the new backing file is specified correctly or the guest-visible content of the image will be corrupted.

This mode is useful for renaming or moving the backing file. It can be used without an accessible old backing file. For instance, it can be used to fix an image whose backing file has already been moved or renamed.

14.10. RE-SIZING THE DISK IMAGE

Change the disk image *filename* as if it had been created with size *size*. Only images in raw format can be resized in both directions, whereas qcow2 version 2 or qcow2 version 3 images can be grown but cannot be shrunk.

Use the following to set the size of the disk image *filename* to *size* bytes:

```
# qemu-img resize filename size
```

You can also resize relative to the current size of the disk image. To give a size relative to the current size, prefix the number of bytes with **+** to grow, or **-** to reduce the size of the disk image by that number of bytes. Adding a unit suffix allows you to set the image size in kilobytes (K), megabytes (M), gigabytes (G) or terabytes (T).

```
# qemu-img resize filename [+|-]size[K|M|G|T]
```

WARNING

Before using this command to shrink a disk image, you *must* use file system and partitioning tools inside the VM itself to reduce allocated file systems and partition sizes accordingly. Failure to do so will result in data loss.

After using this command to grow a disk image, you must use file system and partitioning tools inside the VM to actually begin using the new space on the device.

14.11. LISTING, CREATING, APPLYING, AND DELETING A SNAPSHOT

Using different parameters from the **qemu-img snapshot** command you can list, apply, create, or delete an existing snapshot (*snapshot*) of specified image (*filename*).

```
# qemu-img snapshot [ -l | -a snapshot | -c snapshot | -d snapshot ]
filename
```

The accepted arguments are as follows:

- **-l** lists all snapshots associated with the specified disk image.
- The apply option, **-a**, reverts the disk image (*filename*) to the state of a previously saved *snapshot*.
- **-c** creates a snapshot (*snapshot*) of an image (*filename*).
- **-d** deletes the specified snapshot.

14.12. SUPPORTED QEMU-IMG FORMATS

When a format is specified in any of the **qemu-img** commands, the following format types may be used:

- **raw** - Raw disk image format (default). This can be the fastest file-based format. If your file system supports holes (for example in ext2 or ext3), then only the written sectors will reserve space. Use **qemu-img info** to obtain the real size used by the image or **ls -ls** on Unix/Linux. Although Raw images give optimal performance, only very basic features are available with a Raw image. For example, no snapshots are available.

- **qcow2** - QEMU image format, the most versatile format with the best feature set. Use it to have optional AES encryption, zlib-based compression, support of multiple VM snapshots, and smaller images, which are useful on file systems that do not support holes . Note that this expansive feature set comes at the cost of performance.

  Although only the formats above can be used to run on a guest virtual machine or host physical machine, **qemu-img** also recognizes and supports the following formats in order to convert from them into either **raw** , or **qcow2** format. The format of an image is usually detected automatically. In addition to converting these formats into **raw** or **qcow2** , they can be converted back from **raw** or **qcow2** to the original format. Note that the qcow2 version supplied with Red Hat Enterprise Linux 7 is 1.1. The format that is supplied with previous versions of Red Hat Enterprise Linux will be 0.10. You can revert image files to previous versions of qcow2. To know which version you are using, run **qemu-img info qcow2** ***[imagefilename.img]*** command. To change the qcow version see Section 23.20.2, “Setting Target Elements”.

- **bochs** - Bochs disk image format.

- **cloop** - Linux Compressed Loop image, useful only to reuse directly compressed CD-ROM images present for example in the Knoppix CD-ROMs.

- **cow** - User Mode Linux Copy On Write image format. The **cow** format is included only for compatibility with previous versions.

- **dmg** - Mac disk image format.

- **nbd** - Network block device.

- **parallels** - Parallels virtualization disk image format.

- **qcow** - Old QEMU image format. Only included for compatibility with older versions.

- **qed** - Old QEMU image format. Only included for compatibility with older versions.

- **vdi** - Oracle VM VirtualBox hard disk image format.

- **vhdx** - Microsoft Hyper-V virtual hard disk-X disk image format.

- **vmdk** - VMware 3 and 4 compatible image format.

- **vvfat** - Virtual VFAT disk image format.

CHAPTER 15. KVM MIGRATION

This chapter covers the migration guest virtual machines from one host physical machine that runs the KVM hypervisor to another. Migrating guests is possible because virtual machines run in a virtualized environment instead of directly on the hardware.

15.1. MIGRATION DEFINITION AND BENEFITS

Migration works by sending the state of the guest virtual machine's memory and any virtualized devices to a destination host physical machine. It is recommended to use shared, networked storage to store the guest's images to be migrated. It is also recommended to use libvirt-managed storage pools for shared storage when migrating virtual machines.

Migrations can be performed both with *live* (running) and *non-live* (shut-down) guests.

In a *live migration*, the guest virtual machine continues to run on the source host machine, while the guest's memory pages are transferred to the destination host machine. During migration, KVM monitors the source for any changes in pages it has already transferred, and begins to transfer these changes when all of the initial pages have been transferred. KVM also estimates transfer speed during migration, so when the remaining amount of data to transfer will reaches a certain configurable period of time (10ms by default), KVM suspends the original guest virtual machine, transfers the remaining data, and resumes the same guest virtual machine on the destination host physical machine.

In contrast, a *non-live migration* (offline migration) suspends the guest virtual machine and then copies the guest's memory to the destination host machine. The guest is then resumed on the destination host machine and the memory the guest used on the source host machine is freed. The time it takes to complete such a migration only depends on network bandwidth and latency. If the network is experiencing heavy use or low bandwidth, the migration will take much longer. Note that if the original guest virtual machine modifies pages faster than KVM can transfer them to the destination host physical machine, offline migration must be used, as live migration would never complete.

Migration is useful for:

Load balancing

Guest virtual machines can be moved to host physical machines with lower usage if their host machine becomes overloaded, or if another host machine is under-utilized.

Hardware independence

When you need to upgrade, add, or remove hardware devices on the host physical machine, you can safely relocate guest virtual machines to other host physical machines. This means that guest virtual machines do not experience any downtime for hardware improvements.

Energy saving

Virtual machines can be redistributed to other host physical machines, and the unloaded host systems can thus be powered off to save energy and cut costs in low usage periods.

Geographic migration

Virtual machines can be moved to another location for lower latency or when required by other reasons.

15.2. MIGRATION REQUIREMENTS AND LIMITATIONS

Before using KVM migration, make sure that your system fulfills the migration's requirements, and that you are aware of its limitations.

Migration requirements

- A guest virtual machine installed on shared storage using one of the following protocols:
  - Fibre Channel-based LUNs
  - iSCSI
  - NFS
  - GFS2
  - SCSI RDMA protocols (SCSI RCP): the block export protocol used in Infiniband and 10GbE iWARP adapters
- Make sure that the **libvirtd** service is enabled and running.

```
# systemctl enable libvirtd.service
# systemctl restart libvirtd.service
```

- The ability to migrate effectively is dependant on the parameter setting in the **/etc/libvirt/libvirtd.conf** file. To edit this file, use the following procedure:

  Procedure 15.1. Configuring libvirtd.conf

  1. Opening the **libvirtd.conf** requires running the command as root:

```
# vim /etc/libvirt/libvirtd.conf
```

  2. Change the parameters as needed and save the file.
  3. Restart the **libvirtd** service:

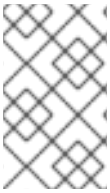
```
# systemctl restart libvirtd
```

- The migration platforms and versions should be checked against Table 15.1, “Live Migration Compatibility”
- Use a separate system exporting the shared storage medium. Storage should not reside on either of the two host physical machines used for the migration.
- Shared storage must mount at the same location on source and destination systems. The mounted directory names must be identical. Although it is possible to keep the images using different paths, it is not recommended. Note that, if you intend to use virt-manager to perform the migration, the path names must be identical. If you intend to use virsh to perform the migration, different network configurations and mount directories can be used with the help of **--xml** option or pre-hooks . For more information on pre-hooks, see the libvirt upstream documentation, and for more information on the XML option, see Chapter 23, *Manipulating the Domain XML*.

- When migration is attempted on an existing guest virtual machine in a public bridge+tap network, the source and destination host machines must be located on the same network. Otherwise, the guest virtual machine network will not operate after migration.

Migration Limitations

- Guest virtual machine migration has the following limitations when used on Red Hat Enterprise Linux with virtualization technology based on KVM:
  - Point to point migration – must be done manually to designate destination hypervisor from originating hypervisor
  - No validation or roll-back is available
  - Determination of target may only be done manually
  - Storage migration cannot be performed live on Red Hat Enterprise Linux 7, but you can migrate storage while the guest virtual machine is powered down. Live storage migration is available on Red Hat Virtualization. Call your service representative for details.



NOTE

If you are migrating a guest machine that has virtio devices on it, make sure to set the number of vectors on any virtio device on either platform to 32 or fewer. For detailed information, see Section 23.18, “Devices”.

15.3. LIVE MIGRATION AND RED HAT ENTERPRISE LINUX VERSION COMPATIBILITY

Live Migration is supported as shown in Table 15.1, “Live Migration Compatibility”:

Table 15.1. Live Migration Compatibility

Migration Method	Release Type	Example	Live Migration Support	Notes
Forward	Major release	6.5+ → 7.x	Fully supported	Any issues should be reported
Backward	Major release	7.x → 6.y	Not supported	
Forward	Minor release	7.x → 7.y (7.0 → 7.1)	Fully supported	Any issues should be reported
Backward	Minor release	7.y → 7.x (7.1 → 7.0)	Fully supported	Any issues should be reported

Troubleshooting problems with migration

- **Issues with the migration protocol** — If backward migration ends with "unknown section error", repeating the migration process can repair the issue as it may be a transient error. If not, report the problem.

- **Issues with audio devices** — When migrating from Red Hat Enterprise Linux 6.x to Red Hat Enterprise Linux 7.y, note that the es1370 audio card is no longer supported. Use the ac97 audio card instead.

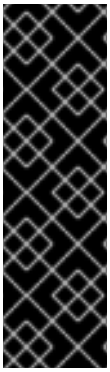
- **Issues with network cards** — When migrating from Red Hat Enterprise Linux 6.x to Red Hat Enterprise Linux 7.y, note that the pcnet and ne2k_pci network cards are no longer supported. Use the virtio-net network device instead.

Configuring Network Storage

Configure shared storage and install a guest virtual machine on the shared storage.

Alternatively, use the NFS example in Section 15.4, “Shared Storage Example: NFS for a Simple Migration”

15.4. SHARED STORAGE EXAMPLE: NFS FOR A SIMPLE MIGRATION



IMPORTANT

This example uses NFS to share guest virtual machine images with other KVM host physical machines. Although not practical for large installations, it is presented to demonstrate migration techniques only. Do not use this example for migrating or running more than a few guest virtual machines. In addition, it is required that the ***synch*** parameter is enabled. This is required for proper export of the NFS storage.

iSCSI storage is a better choice for large deployments. For configuration details, see Section 13.2.3.5, “iSCSI-based storage pools”.

For detailed information on configuring NFS, opening IP tables, and configuring the firewall, see *Red Hat Linux Storage Administration Guide*.

Make sure that NFS file locking is not used as it is not supported in KVM.

1. **Export your libvirt image directory**
   Migration requires storage to reside on a system that is separate to the migration target systems. On this separate system, export the storage by adding the default image directory to the **`/etc/exports`** file:

   ```
   /var/lib/libvirt/images *.example.com(rw,no_root_squash,sync)
   ```

   Change the ***`hostname`*** parameter as required for your environment.

2. **Start NFS**

   a. Install the NFS packages if they are not yet installed:

   ```
   # yum install nfs-utils
   ```

   b. Make sure that the ports for NFS in **`iptables`** (2049, for example) are opened and add NFS to the **`/etc/hosts.allow`** file.

   c. Start the NFS service:

   ```
   # systemctl start nfs-server
   ```

3. **Mount the shared storage on the source and the destination**
   On the migration source and the destination systems, mount the **/var/lib/libvirt/images** directory:

```
# mount storage_host:/var/lib/libvirt/images /var/lib/libvirt/images
```

WARNING

Whichever directory is chosen for the source host physical machine must be exactly the same as that on the destination host physical machine. This applies to all types of shared storage. The directory must be the same or the migration with virt-manager will fail.

15.5. LIVE KVM MIGRATION WITH VIRSH

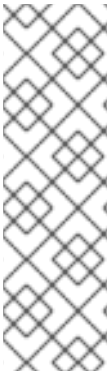
A guest virtual machine can be migrated to another host physical machine with the **virsh** command. The **migrate** command accepts parameters in the following format:

```
# virsh migrate --live GuestName DestinationURL
```

Note that the *--live* option may be eliminated when live migration is not required. Additional options are listed in Section 15.5.2, “Additional Options for the virsh migrate Command”.

The ***GuestName*** parameter represents the name of the guest virtual machine which you want to migrate.

The ***DestinationURL*** parameter is the connection URL of the destination host physical machine. The destination system must run the same version of Red Hat Enterprise Linux, be using the same hypervisor and have **libvirt** running.



NOTE

The ***DestinationURL*** parameter for normal migration and peer2peer migration has different semantics:

- normal migration: the ***DestinationURL*** is the URL of the target host physical machine as seen from the source guest virtual machine.
- peer2peer migration: ***DestinationURL*** is the URL of the target host physical machine as seen from the source host physical machine.

Once the command is entered, you will be prompted for the root password of the destination system.

IMPORTANT

Name resolution must be working on both sides (source and destination) in order for migration to succeed. Each side must be able to find the other. Make sure that you can ping one side to the other to check that the name resolution is working.

Example: live migration with virsh

This example migrates from **host1.example.com** to **host2.example.com**. Change the host physical machine names for your environment. This example migrates a virtual machine named **guest1-rhel6-64**.

This example assumes you have fully configured shared storage and meet all the prerequisites (listed here: Migration requirements).

1. **Verify the guest virtual machine is running**
   From the source system, **host1.example.com**, verify **guest1-rhel6-64** is running:

   ```
   [root@host1 ~]# virsh list
   Id Name                 State
   ----------------------------------
    10 guest1-rhel6-64     running
   ```

2. **Migrate the guest virtual machine**
   Execute the following command to live migrate the guest virtual machine to the destination, **host2.example.com**. Append **/system** to the end of the destination URL to tell libvirt that you need full access.

   ```
   # virsh migrate --live guest1-rhel7-64
   qemu+ssh://host2.example.com/system
   ```

   Once the command is entered you will be prompted for the root password of the destination system.

3. **Wait**
   The migration may take some time depending on load and the size of the guest virtual machine. **virsh** only reports errors. The guest virtual machine continues to run on the source host physical machine until fully migrated.

4. **Verify the guest virtual machine has arrived at the destination host**
   From the destination system, **host2.example.com**, verify **guest1-rhel7-64** is running:

   ```
   [root@host2 ~]# virsh list
   Id Name                 State
   ----------------------------------
    10 guest1-rhel7-64     running
   ```

The live migration is now complete.

NOTE

libvirt supports a variety of networking methods including TLS/SSL, UNIX sockets, SSH, and unencrypted TCP. For more information on using other methods, see Chapter 18, *Remote Management of Guests*.

NOTE

Non-running guest virtual machines can be migrated using the following command:

```
# virsh migrate --offline --persistent
```

15.5.1. Additional Tips for Migration with virsh

It is possible to perform multiple, concurrent live migrations where each migration runs in a separate command shell. However, this should be done with caution and should involve careful calculations as each migration instance uses one MAX_CLIENT from each side (source and target). As the default setting is 20, there is enough to run 10 instances without changing the settings. Should you need to change the settings, see the procedure Procedure 15.1, “Configuring libvirtd.conf”.

1. Open the libvirtd.conf file as described in Procedure 15.1, “Configuring libvirtd.conf”.

2. Look for the Processing controls section.

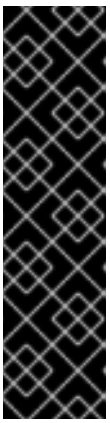
```
#################################################################
#
# Processing controls
#

# The maximum number of concurrent client connections to allow
# over all sockets combined.
#max_clients = 5000

# The maximum length of queue of connections waiting to be
# accepted by the daemon. Note, that some protocols supporting
# retransmission may obey this so that a later reattempt at
# connection succeeds.
#max_queued_clients = 1000

# The minimum limit sets the number of workers to start up
# initially. If the number of active clients exceeds this,
# then more threads are spawned, upto max_workers limit.
# Typically you'd want max_workers to equal maximum number
# of clients allowed
#min_workers = 5
#max_workers = 20


# The number of priority workers. If all workers from above
# pool will stuck, some calls marked as high priority
# (notably domainDestroy) can be executed in this pool.
#prio_workers = 5

# Total global limit on concurrent RPC calls. Should be
# at least as large as max_workers. Beyond this, RPC requests
# will be read into memory and queued. This directly impact
# memory usage, currently each request requires 256 KB of
# memory. So by default upto 5 MB of memory is used
#
# XXX this isn't actually enforced yet, only the per-client
# limit is used so far
```

```
#max_requests = 20

# Limit on concurrent requests from a single client
# connection. To avoid one client monopolizing the server
# this should be a small fraction of the global max_requests
# and max_workers parameter
#max_client_requests = 5

#################################################################
```

3. Change the ***max_clients*** and ***max_workers*** parameters settings. It is recommended that the number be the same in both parameters. The ***max_clients*** will use 2 clients per migration (one per side) and ***max_workers*** will use 1 worker on the source and 0 workers on the destination during the perform phase and 1 worker on the destination during the finish phase.

IMPORTANT

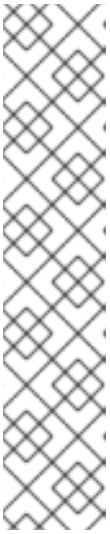
The ***max_clients*** and ***max_workers*** parameters settings are affected by all guest virtual machine connections to the libvirtd service. This means that any user that is using the same guest virtual machine and is performing a migration at the same time will also beholden to the limits set in the ***max_clients*** and ***max_workers*** parameters settings. This is why the maximum value needs to be considered carefully before performing a concurrent live migration.



IMPORTANT

The ***max_clients*** parameter controls how many clients are allowed to connect to libvirt. When a large number of containers are started at once, this limit can be easily reached and exceeded. The value of the ***max_clients*** parameter could be increased to avoid this, but doing so can leave the system more vulnerable to denial of service (DoS) attacks against instances. To alleviate this problem, a new ***max_anonymous_clients*** setting has been introduced in Red Hat Enterprise Linux 7.0 that specifies a limit of connections which are accepted but not yet authenticated. You can implement a combination of ***max_clients*** and ***max_anonymous_clients*** to suit your workload.

4. Save the file and restart the service.



NOTE

There may be cases where a migration connection drops because there are too many ssh sessions that have been started, but not yet authenticated. By default, **sshd** allows only 10 sessions to be in a "pre-authenticated state" at any time. This setting is controlled by the **MaxStartups** parameter in the sshd configuration file (located here: **/etc/ssh/sshd_config**), which may require some adjustment. Adjusting this parameter should be done with caution as the limitation is put in place to prevent DoS attacks (and over-use of resources in general). Setting this value too high will negate its purpose. To change this parameter, edit the file **/etc/ssh/sshd_config**, remove the # from the beginning of the **MaxStartups** line, and change the **10** (default value) to a higher number. Remember to save the file and restart the **sshd** service. For more information, see the **sshd_config** man page.

15.5.2. Additional Options for the virsh migrate Command

In addition to **--live**, virsh migrate accepts the following options:

- **--direct** - used for direct migration
- **--p2p** - used for peer-to-peer migration
- **--tunneled** - used for tunneled migration
- **--offline** - migrates domain definition without starting the domain on destination and without stopping it on source host. Offline migration may be used with inactive domains and it must be used with the **--persistent** option.
- **--persistent** - leaves the domain persistent on destination host physical machine
- **--undefinesource** - undefines the domain on the source host physical machine
- **--suspend** - leaves the domain paused on the destination host physical machine
- **--change-protection** - enforces that no incompatible configuration changes will be made to the domain while the migration is underway; this flag is implicitly enabled when supported by the hypervisor, but can be explicitly used to reject the migration if the hypervisor lacks change protection support.
- **--unsafe** - forces the migration to occur, ignoring all safety procedures.
- **--verbose** - displays the progress of migration as it is occurring
- **--compressed** - activates compression of memory pages that have to be transferred repeatedly during live migration.
- **--abort-on-error** - cancels the migration if a soft error (for example I/O error) happens during the migration.
- **--domain *[name]*** - sets the domain name, id or uuid.
- **--desturi *[URI]*** - connection URI of the destination host as seen from the client (normal migration) or source (p2p migration).
- **--migrateuri *[URI]*** - the migration URI, which can usually be omitted.
- **--graphicsuri *[URI]*** - graphics URI to be used for seamless graphics migration.
- **--listen-address *[address]*** - sets the listen address that the hypervisor on the destination side should bind to for incoming migration.
- **--timeout *[seconds]*** - forces a guest virtual machine to suspend when the live migration counter exceeds N seconds. It can only be used with a live migration. Once the timeout is initiated, the migration continues on the suspended guest virtual machine.
- **--dname *[newname]*** - is used for renaming the domain during migration, which also usually can be omitted
- **--xml *[filename]*** - the filename indicated can be used to supply an alternative XML file for use on the destination to supply a larger set of changes to any host-specific portions of the

domain XML, such as accounting for naming differences between source and destination in accessing underlying storage. This option is usually omitted.

- **`--migrate-disks`** ***`[disk_identifiers]`*** - this option can be used to select which disks are copied during the migration. This allows for more efficient live migration when copying certain disks is undesirable, such as when they already exist on the destination, or when they are no longer useful. *[disk_identifiers]* should be replaced by a comma-separated list of disks to be migrated, identified by their arguments found in the **`<target dev= />`** line of the Domain XML file.

In addition, the following commands may help as well:

- **`virsh migrate-setmaxdowntime`** ***`[domain] [downtime]`*** - will set a maximum tolerable downtime for a domain which is being live-migrated to another host. The specified downtime is in milliseconds. The domain specified must be the same domain that is being migrated.

- **`virsh migrate-compcache`** ***`[domain]`*** **`--size`** - will set and or get the size of the cache in bytes which is used for compressing repeatedly transferred memory pages during a live migration. When the **`--size`** is not used the command displays the current size of the compression cache. When **`--size`** is used, and specified in bytes, the hypervisor is asked to change compression to match the indicated size, following which the current size is displayed. The **`--size`** argument is supposed to be used while the domain is being live migrated as a reaction to the migration progress and increasing number of compression cache misses obtained from the **`domjobinfo`**.

- **`virsh migrate-setspeed`** ***`[domain] [bandwidth]`*** - sets the migration bandwidth in Mib/sec for the specified domain which is being migrated to another host.

- **`virsh migrate-getspeed`** ***`[domain]`*** - gets the maximum migration bandwidth that is available in Mib/sec for the specified domain.

For more information, see Migration Limitations or the virsh man page.

15.6. MIGRATING WITH VIRT-MANAGER

This section covers migrating a KVM guest virtual machine with **`virt-manager`** from one host physical machine to another.

1. **Connect to the target host physical machine**
   In the **virt-manager** interface, connect to the target host physical machine by selecting the **File** menu, then click **Add Connection**.

2. **Add connection**
   The **`Add Connection`** window appears.

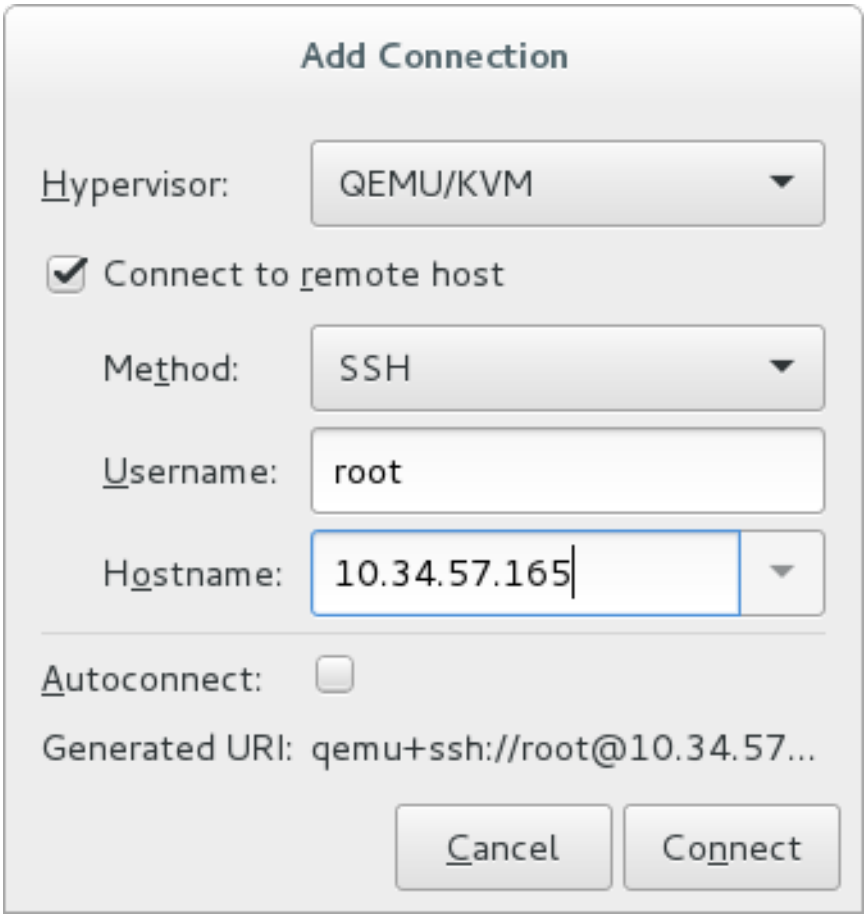


Figure 15.1. Adding a connection to the target host physical machine

Enter the following details:

- **`Hypervisor`**: Select **QEMU/KVM**.
- **`Method`**: Select the connection method.
- **`Username`**: Enter the user name for the remote host physical machine.
- **`Hostname`**: Enter the host name for the remote host physical machine.

NOTE

For more information on the connection options, see Section 19.5, “Adding a Remote Connection”.

Click **`Connect`**. An SSH connection is used in this example, so the specified user's password must be entered in the next step.

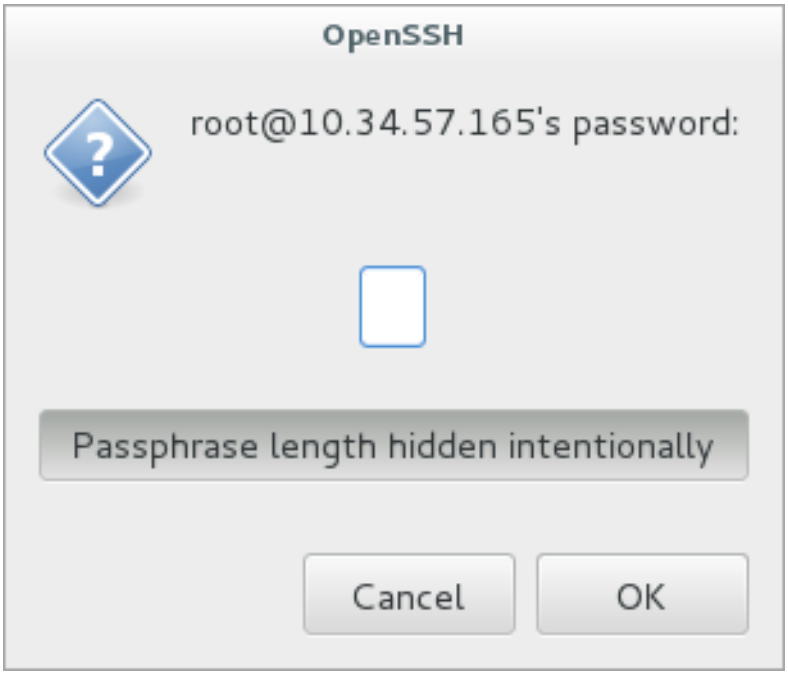


Figure 15.2. Enter password

3. **Configure shared storage**
   Ensure that both the source and the target host are sharing storage, for example using NFS.

4. **Migrate guest virtual machines**
   Right-click the guest that is to be migrated, and click **Migrate**.

   In the **New Host** field, use the drop-down list to select the host physical machine you wish to migrate the guest virtual machine to and click **Migrate**.

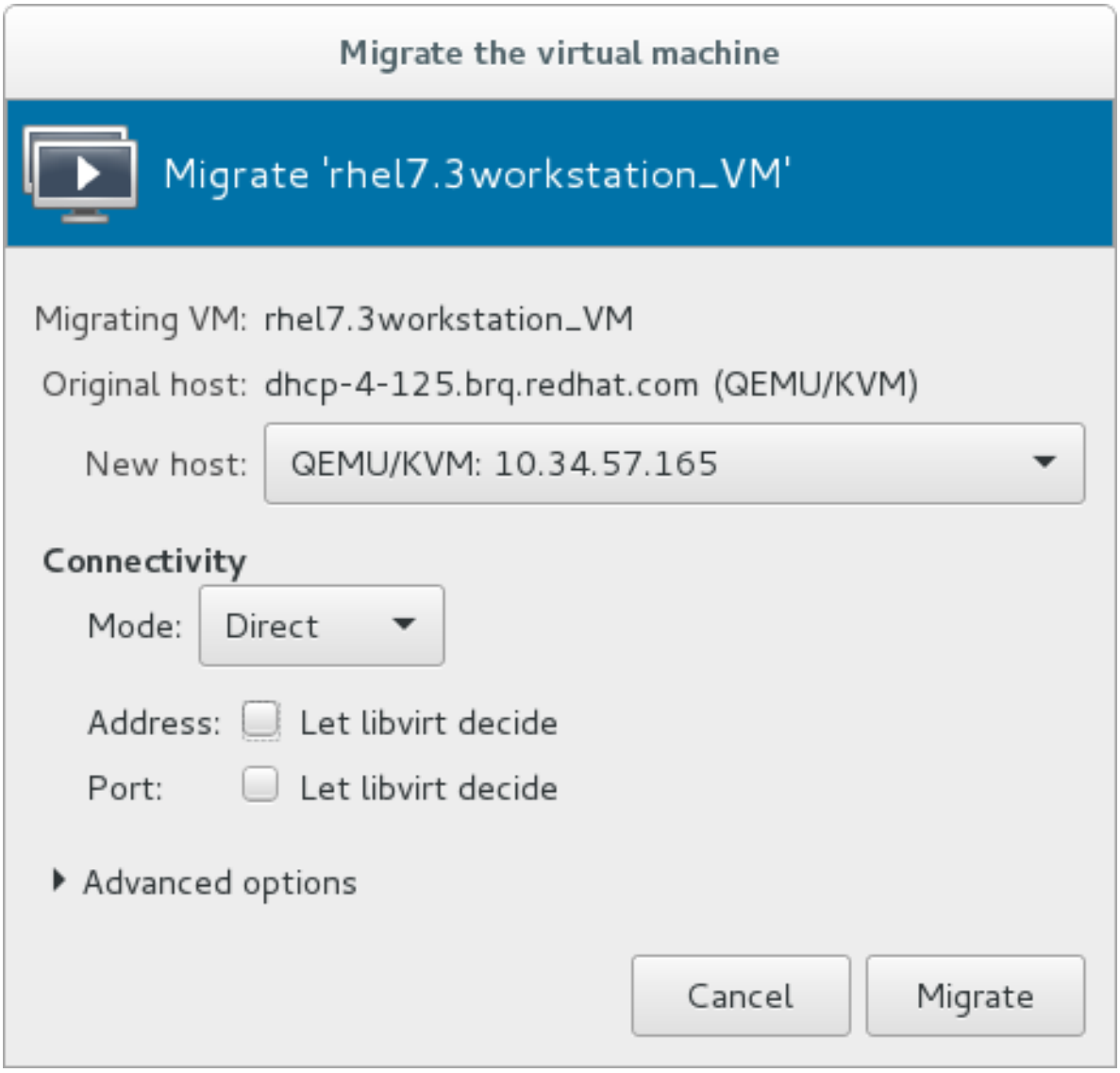


Figure 15.3. Choosing the destination host physical machine and starting the migration process

A progress window appears.

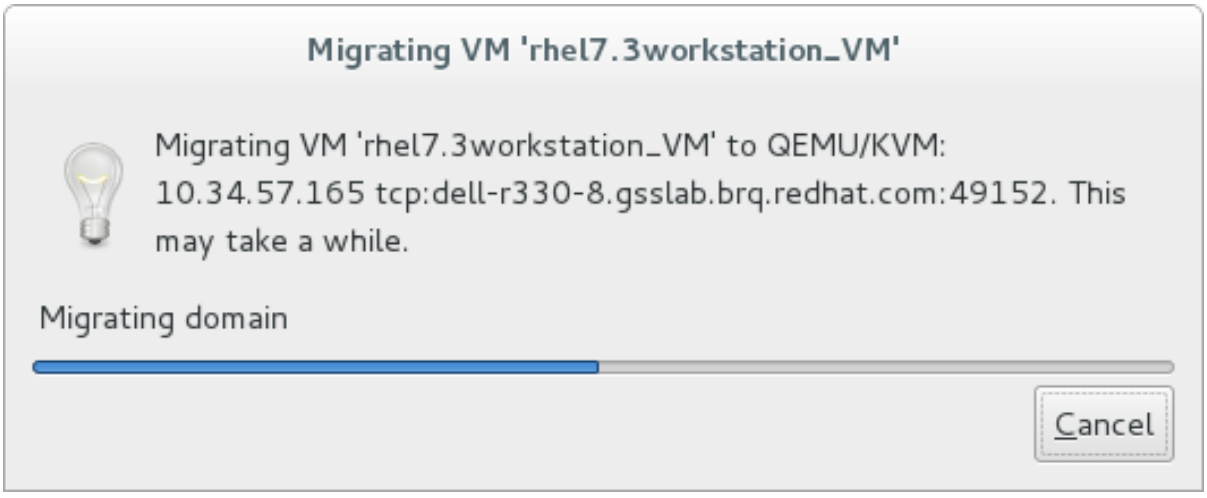


Figure 15.4. Progress window

If the migration finishes without any problems, **virt-manager** displays the newly migrated guest virtual machine running in the destination host.

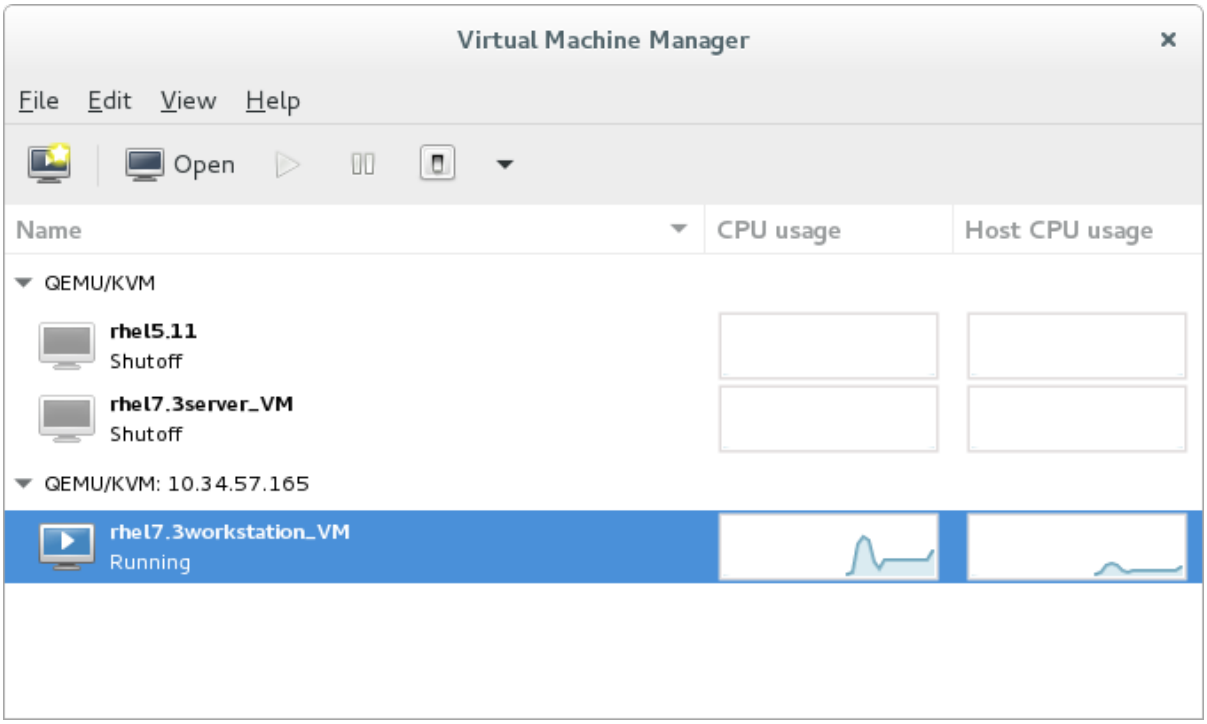


Figure 15.5. Migrated guest virtual machine running in the destination host physical machine

CHAPTER 16. GUEST VIRTUAL MACHINE DEVICE CONFIGURATION

Red Hat Enterprise Linux 7 supports three classes of devices for guest virtual machines:

- **Emulated devices** are purely virtual devices that mimic real hardware, allowing unmodified guest operating systems to work with them using their standard in-box drivers.
- **Virtio devices** (also known as *paravirtualized*) are purely virtual devices designed to work optimally in a virtual machine. Virtio devices are similar to emulated devices, but non-Linux virtual machines do not include the drivers they require by default. Virtualization management software like the Virtual Machine Manager (**virt-manager**) and the Red Hat Virtualization Hypervisor install these drivers automatically for supported non-Linux guest operating systems. Red Hat Enterprise Linux 7 supports up to 216 virtio devices. For more information, see Chapter 5, *KVM Paravirtualized (virtio) Drivers*.
- **Assigned devices** are physical devices that are exposed to the virtual machine. This method is also known as *passthrough*. Device assignment allows virtual machines exclusive access to PCI devices for a range of tasks, and allows PCI devices to appear and behave as if they were physically attached to the guest operating system. Red Hat Enterprise Linux 7 supports up to 32 assigned devices per virtual machine.

  Device assignment is supported on PCIe devices, including select graphics devices. Parallel PCI devices may be supported as assigned devices, but they have severe limitations due to security and system configuration conflicts.

Red Hat Enterprise Linux 7 supports PCI hot plug of devices exposed as single-function slots to the virtual machine. Single-function host devices and individual functions of multi-function host devices may be configured to enable this. Configurations exposing devices as multi-function PCI slots to the virtual machine are recommended only for non-hotplug applications.

For more information on specific devices and related limitations, see Section 23.18, “Devices”.

NOTE

Platform support for interrupt remapping is required to fully isolate a guest with assigned devices from the host. Without such support, the host may be vulnerable to interrupt injection attacks from a malicious guest. In an environment where guests are trusted, the admin may opt-in to still allow PCI device assignment using the **allow_unsafe_interrupts** option to the ***vfio_iommu_type1*** module. This may either be done persistently by adding a .conf file (for example **local.conf**) to **/etc/modprobe.d** containing the following:

```
options vfio_iommu_type1 allow_unsafe_interrupts=1
```

or dynamically using the sysfs entry to do the same:

```
# echo 1 >
/sys/module/vfio_iommu_type1/parameters/allow_unsafe_interrupts
```

16.1. PCI DEVICES

PCI device assignment is only available on hardware platforms supporting either Intel VT-d or AMD IOMMU. These Intel VT-d or AMD IOMMU specifications must be enabled in the host BIOS for PCI device assignment to function.

Procedure 16.1. Preparing an Intel system for PCI device assignment

1. **Enable the Intel VT-d specifications**
   The Intel VT-d specifications provide hardware support for directly assigning a physical device to a virtual machine. These specifications are required to use PCI device assignment with Red Hat Enterprise Linux.

   The Intel VT-d specifications must be enabled in the BIOS. Some system manufacturers disable these specifications by default. The terms used to see these specifications can differ between manufacturers; consult your system manufacturer's documentation for the appropriate terms.

2. **Activate Intel VT-d in the kernel**
   Activate Intel VT-d in the kernel by adding the ***intel_iommu=on*** and ***iommu=pt*** parameters to the end of the GRUB_CMDLINX_LINUX line, within the quotes, in the **/etc/sysconfig/grub** file.

   The example below is a modified **grub** file with Intel VT-d activated.

   ```
   GRUB_CMDLINE_LINUX="rd.lvm.lv=vg_VolGroup00/LogVol01
   vconsole.font=latarcyrheb-sun16 rd.lvm.lv=vg_VolGroup_1/root
   vconsole.keymap=us $([ -x /usr/sbin/rhcrashkernel-param ] &&
   /usr/sbin/
   rhcrashkernel-param || :) rhgb quiet intel_iommu=on iommu=pt"
   ```

3. **Regenerate config file**
   Regenerate /etc/grub2.cfg by running:

   ```
   grub2-mkconfig -o /etc/grub2.cfg
   ```

   Note that if you are using a UEFI-based host, the target file should be **/etc/grub2-efi.cfg**.

4. **Ready to use**
   Reboot the system to enable the changes. Your system is now capable of PCI device assignment.

Procedure 16.2. Preparing an AMD system for PCI device assignment

1. **Enable the AMD IOMMU specifications**
   The AMD IOMMU specifications are required to use PCI device assignment in Red Hat Enterprise Linux. These specifications must be enabled in the BIOS. Some system manufacturers disable these specifications by default.

2. **Enable IOMMU kernel support**
   Append ***amd_iommu=pt*** to the end of the GRUB_CMDLINX_LINUX line, within the quotes, in **/etc/sysconfig/grub** so that AMD IOMMU specifications are enabled at boot.

3. **Regenerate config file**
   Regenerate /etc/grub2.cfg by running:

   ```
   grub2-mkconfig -o /etc/grub2.cfg
   ```

Note that if you are using a UEFI-based host, the target file should be **/etc/grub2-efi.cfg**.

4. **Ready to use**
   Reboot the system to enable the changes. Your system is now capable of PCI device assignment.

NOTE

For further information on IOMMU, see Appendix E, *Working with IOMMU Groups*.

16.1.1. Assigning a PCI Device with virsh

These steps cover assigning a PCI device to a virtual machine on a KVM hypervisor.

This example uses a PCIe network controller with the PCI identifier code, **pci_0000_01_00_0**, and a fully virtualized guest machine named *guest1-rhel7-64*.

Procedure 16.3. Assigning a PCI device to a guest virtual machine with virsh

1. **Identify the device**
   First, identify the PCI device designated for device assignment to the virtual machine. Use the **lspci** command to list the available PCI devices. You can refine the output of **lspci** with **grep**.

   This example uses the Ethernet controller highlighted in the following output:

```
# lspci | grep Ethernet
00:19.0 Ethernet controller: Intel Corporation 82567LM-2 Gigabit
Network Connection
01:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
01:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
```

   This Ethernet controller is shown with the short identifier **00:19.0**. We need to find out the full identifier used by **virsh** in order to assign this PCI device to a virtual machine.

   To do so, use the **virsh nodedev-list** command to list all devices of a particular type (**pci**) that are attached to the host machine. Then look at the output for the string that maps to the short identifier of the device you wish to use.

   This example shows the string that maps to the Ethernet controller with the short identifier **00:19.0**. Note that the **:** and **.** characters are replaced with underscores in the full identifier.

```
# virsh nodedev-list --cap pci
pci_0000_00_00_0
pci_0000_00_01_0
pci_0000_00_03_0
pci_0000_00_07_0
pci_0000_00_10_0
pci_0000_00_10_1
pci_0000_00_14_0
pci_0000_00_14_1
pci_0000_00_14_2
```

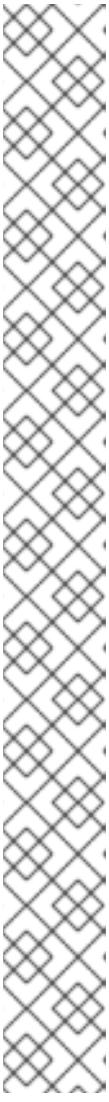
```
pci_0000_00_14_3
pci_0000_00_19_0
pci_0000_00_1a_0
pci_0000_00_1a_1
pci_0000_00_1a_2
pci_0000_00_1a_7
pci_0000_00_1b_0
pci_0000_00_1c_0
pci_0000_00_1c_1
pci_0000_00_1c_4
pci_0000_00_1d_0
pci_0000_00_1d_1
pci_0000_00_1d_2
pci_0000_00_1d_7
pci_0000_00_1e_0
pci_0000_00_1f_0
pci_0000_00_1f_2
pci_0000_00_1f_3
pci_0000_01_00_0
pci_0000_01_00_1
pci_0000_02_00_0
pci_0000_02_00_1
pci_0000_06_00_0
pci_0000_07_02_0
pci_0000_07_03_0
```

Record the PCI device number that maps to the device you want to use; this is required in other steps.

2. **Review device information**
Information on the domain, bus, and function are available from output of the **`virsh nodedev-dumpxml`** command:

```
# virsh nodedev-dumpxml pci_0000_00_19_0
<device>
  <name>pci_0000_00_19_0</name>
  <parent>computer</parent>
  <driver>
    <name>e1000e</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>0</bus>
    <slot>25</slot>
    <function>0</function>
    <product id='0x1502'>82579LM Gigabit Network
Connection</product>
    <vendor id='0x8086'>Intel Corporation</vendor>
    <iommuGroup number='7'>
      <address domain='0x0000' bus='0x00' slot='0x19'
function='0x0'/>
    </iommuGroup>
  </capability>
</device>
```

Figure 16.1. Dump contents



NOTE

An IOMMU group is determined based on the visibility and isolation of devices from the perspective of the IOMMU. Each IOMMU group may contain one or more devices. When multiple devices are present, all endpoints within the IOMMU group must be claimed for any device within the group to be assigned to a guest. This can be accomplished either by also assigning the extra endpoints to the guest or by detaching them from the host driver using **`virsh nodedev-detach`**. Devices contained within a single group may not be split between multiple guests or split between host and guest. Non-endpoint devices such as PCIe root ports, switch ports, and bridges should not be detached from the host drivers and will not interfere with assignment of endpoints.

Devices within an IOMMU group can be determined using the iommuGroup section of the **`virsh nodedev-dumpxml`** output. Each member of the group is provided in a separate "address" field. This information may also be found in sysfs using the following:

```
$ ls
/sys/bus/pci/devices/0000:01:00.0/iommu_group/devices/
```

An example of the output from this would be:

```
0000:01:00.0  0000:01:00.1
```

To assign only 0000.01.00.0 to the guest, the unused endpoint should be detached from the host before starting the guest:

```
$ virsh nodedev-detach pci_0000_01_00_1
```

3. **Determine required configuration details**
   See the output from the **virsh nodedev-dumpxml pci_0000_00_19_0** command for the values required for the configuration file.

   The example device has the following values: bus = 0, slot = 25 and function = 0. The decimal configuration uses those three values:

   ```
   bus='0'
   slot='25'
   function='0'
   ```

4. **Add configuration details**
   Run **virsh edit**, specifying the virtual machine name, and add a device entry in the **<source>** section to assign the PCI device to the guest virtual machine.

   ```
   # virsh edit guest1-rhel7-64
   ```

   ```
   <hostdev mode='subsystem' type='pci' managed='yes'>
     <source>
        <address domain='0' bus='0' slot='25' function='0'/>
     </source>
   </hostdev>
   ```

   Figure 16.2. Add PCI device

   Alternately, run **virsh attach-device**, specifying the virtual machine name and the guest's XML file:

   ```
   virsh attach-device guest1-rhel7-64 file.xml
   ```

5. **Start the virtual machine**

   ```
   # virsh start guest1-rhel7-64
   ```

The PCI device should now be successfully assigned to the virtual machine, and accessible to the guest operating system.

16.1.2. Assigning a PCI Device with virt-manager

PCI devices can be added to guest virtual machines using the graphical **virt-manager** tool. The following procedure adds a Gigabit Ethernet controller to a guest virtual machine.

Procedure 16.4. Assigning a PCI device to a guest virtual machine using virt-manager

1. **Open the hardware settings**
   Open the guest virtual machine and click the **Add Hardware** button to add a new device to the virtual machine.

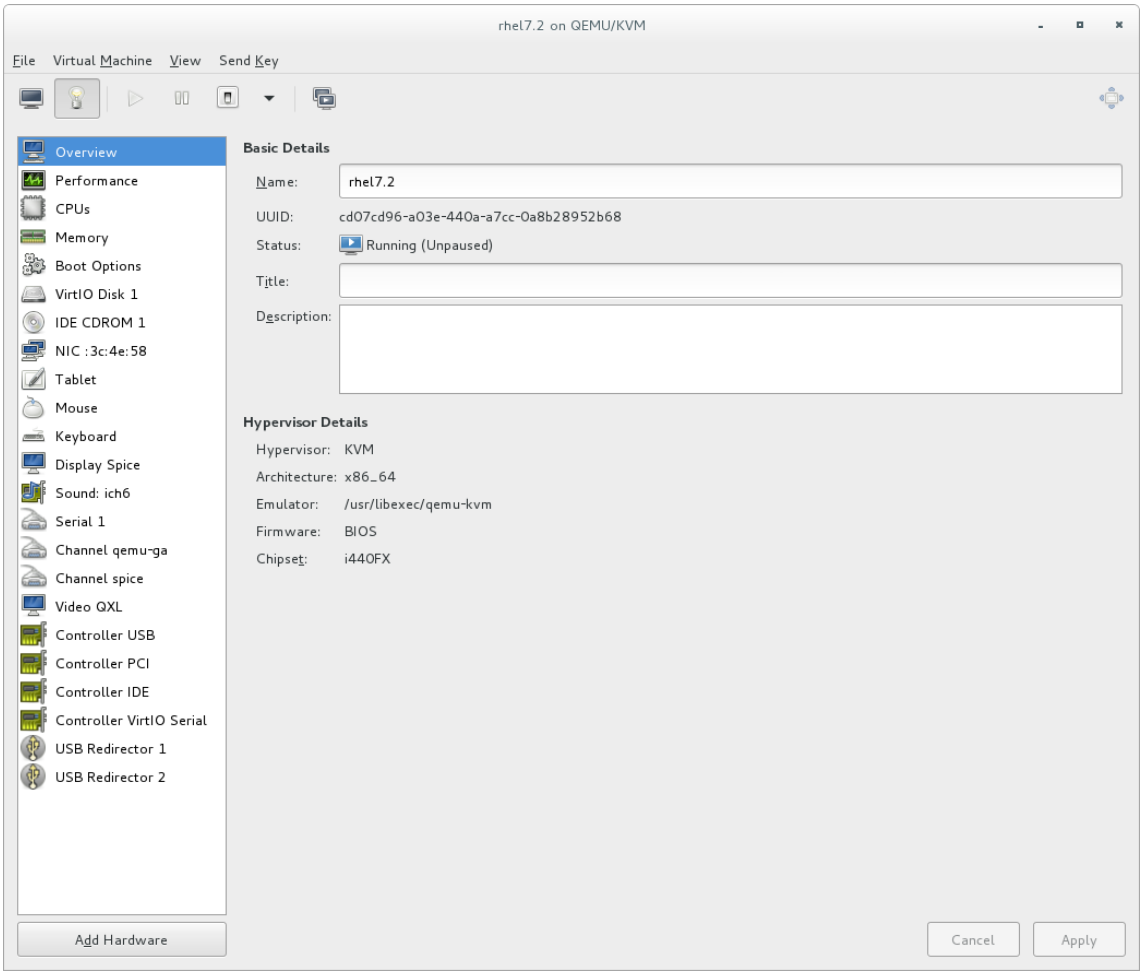


Figure 16.3. The virtual machine hardware information window

2. **Select a PCI device**
   Select **PCI Host Device** from the **`Hardware`** list on the left.

   Select an unused PCI device. Note that selecting PCI devices presently in use by another guest causes errors. In this example, a spare audio controller is used. Click **`Finish`** to complete setup.

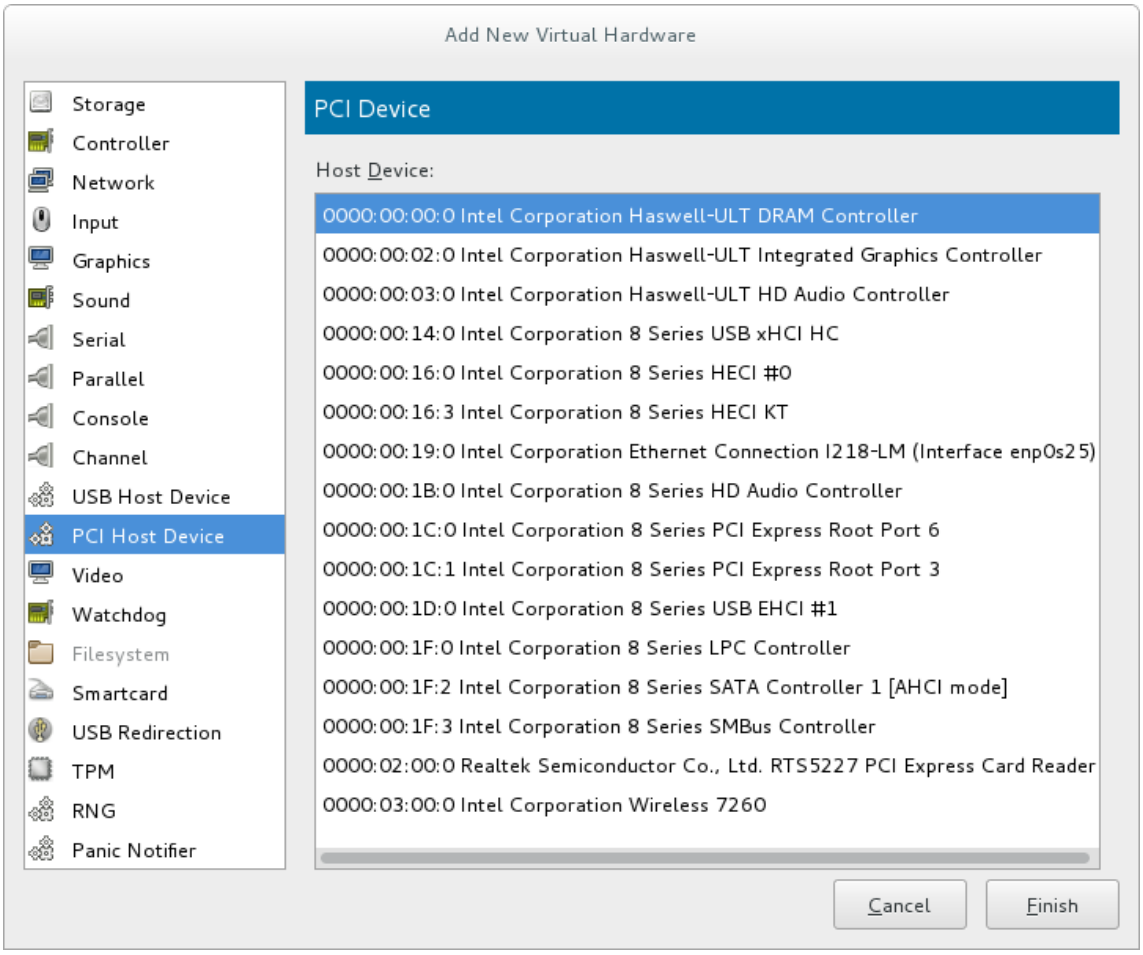


Figure 16.4. The Add new virtual hardware wizard

3. **Add the new device**
   The setup is complete and the guest virtual machine now has direct access to the PCI device.

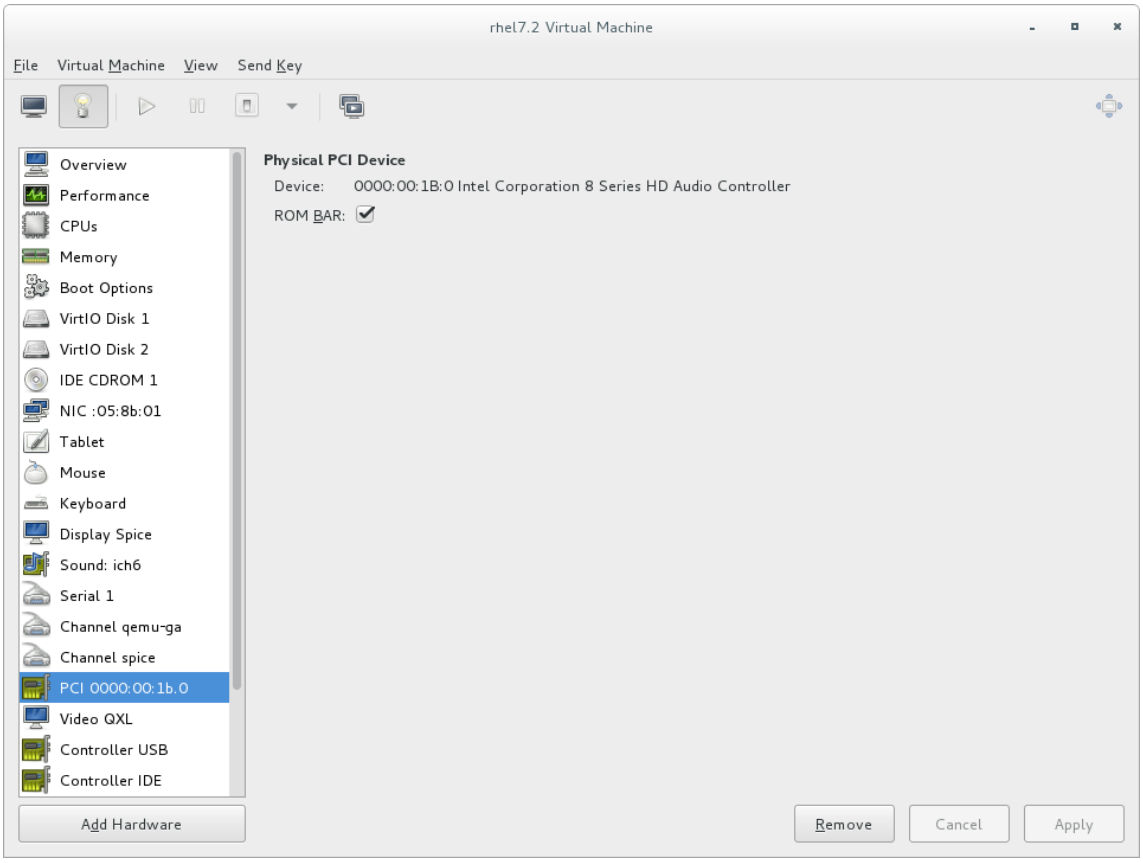
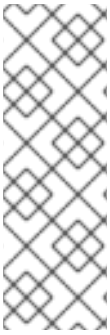


Figure 16.5. The virtual machine hardware information window



NOTE

If device assignment fails, there may be other endpoints in the same IOMMU group that are still attached to the host. There is no way to retrieve group information using virt-manager, but virsh commands can be used to analyze the bounds of the IOMMU group and if necessary sequester devices.

See the Note in Section 16.1.1, “Assigning a PCI Device with virsh” for more information on IOMMU groups and how to detach endpoint devices using virsh.

16.1.3. PCI Device Assignment with virt-install

It is possible to assign a PCI device when installing a guest using the **virt-install** command. To do this, use the ***--host-device*** parameter.

Procedure 16.5. Assigning a PCI device to a virtual machine with virt-install

1. **Identify the device**
   Identify the PCI device designated for device assignment to the guest virtual machine.

```
# lspci | grep Ethernet
00:19.0 Ethernet controller: Intel Corporation 82567LM-2 Gigabit
Network Connection
01:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
01:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
```

The **virsh nodedev-list** command lists all devices attached to the system, and identifies each PCI device with a string. To limit output to only PCI devices, enter the following command:

```
# virsh nodedev-list --cap pci
pci_0000_00_00_0
pci_0000_00_01_0
pci_0000_00_03_0
pci_0000_00_07_0
pci_0000_00_10_0
pci_0000_00_10_1
pci_0000_00_14_0
pci_0000_00_14_1
pci_0000_00_14_2
pci_0000_00_14_3
pci_0000_00_19_0
pci_0000_00_1a_0
pci_0000_00_1a_1
pci_0000_00_1a_2
pci_0000_00_1a_7
pci_0000_00_1b_0
pci_0000_00_1c_0
pci_0000_00_1c_1
pci_0000_00_1c_4
pci_0000_00_1d_0
pci_0000_00_1d_1
pci_0000_00_1d_2
pci_0000_00_1d_7
pci_0000_00_1e_0
pci_0000_00_1f_0
pci_0000_00_1f_2
pci_0000_00_1f_3
pci_0000_01_00_0
pci_0000_01_00_1
pci_0000_02_00_0
pci_0000_02_00_1
pci_0000_06_00_0
pci_0000_07_02_0
pci_0000_07_03_0
```

Record the PCI device number; the number is needed in other steps.

Information on the domain, bus and function are available from output of the **virsh nodedev-dumpxml** command:

```
# virsh nodedev-dumpxml pci_0000_01_00_0
```

```
<device>
  <name>pci_0000_01_00_0</name>
  <parent>pci_0000_00_01_0</parent>
  <driver>
    <name>igb</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>1</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x10c9'>82576 Gigabit Network Connection</product>
    <vendor id='0x8086'>Intel Corporation</vendor>
    <iommuGroup number='7'>
      <address domain='0x0000' bus='0x00' slot='0x19'
function='0x0'/>
    </iommuGroup>
  </capability>
</device>
```

Figure 16.6. PCI device file contents

NOTE

If there are multiple endpoints in the IOMMU group and not all of them are assigned to the guest, you will need to manually detach the other endpoint(s) from the host by running the following command before you start the guest:

```
$ virsh nodedev-detach pci_0000_00_19_1
```

See the Note in Section 16.1.1, “Assigning a PCI Device with virsh” for more information on IOMMU groups.

2. **Add the device**
   Use the PCI identifier output from the **`virsh nodedev`** command as the value for the ***`--host-device`*** parameter.

```
virt-install \
--name=guest1-rhel7-64 \
--disk path=/var/lib/libvirt/images/guest1-rhel7-64.img,size=8 \
--vcpus=2 --ram=2048 \
--location=http://example1.com/installation_tree/RHEL7.0-Server-
x86_64/os \
--nonetworks \
--os-type=linux \
--os-variant=rhel7
--host-device=pci_0000_01_00_0
```

3. **Complete the installation**
   Complete the guest installation. The PCI device should be attached to the guest.

16.1.4. Detaching an Assigned PCI Device

When a host PCI device has been assigned to a guest machine, the host can no longer use the device. If the PCI device is in **managed** mode (configured using the **managed='yes'** parameter in the domain XML file), it attaches to the guest machine and detaches from the guest machine and re-attaches to the host machine as necessary. If the PCI device is not in **managed** mode, you can detach the PCI device from the guest machine and re-attach it using **virsh** or **virt-manager**.

Procedure 16.6. Detaching a PCI device from a guest with virsh

1. **Detach the device**
   Use the following command to detach the PCI device from the guest by removing it in the guest's XML file:

   ```
   # virsh detach-device name_of_guest file.xml
   ```

2. **Re-attach the device to the host (optional)**
   If the device is in ***managed*** mode, skip this step. The device will be returned to the host automatically.

   If the device is not using ***managed*** mode, use the following command to re-attach the PCI device to the host machine:

   ```
   # virsh nodedev-reattach device
   ```

   For example, to re-attach the **pci_0000_01_00_0** device to the host:

   ```
   # virsh nodedev-reattach pci_0000_01_00_0
   ```

   The device is now available for host use.

Procedure 16.7. Detaching a PCI Device from a guest with virt-manager

1. **Open the virtual hardware details screen**
   In **virt-manager**, double-click the virtual machine that contains the device. Select the **Show virtual hardware details** button to display a list of virtual hardware.

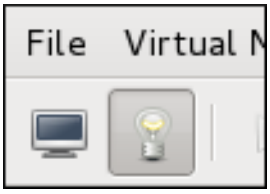
   

   Figure 16.7. The virtual hardware details button

2. **Select and remove the device**
   Select the PCI device to be detached from the list of virtual devices in the left panel.

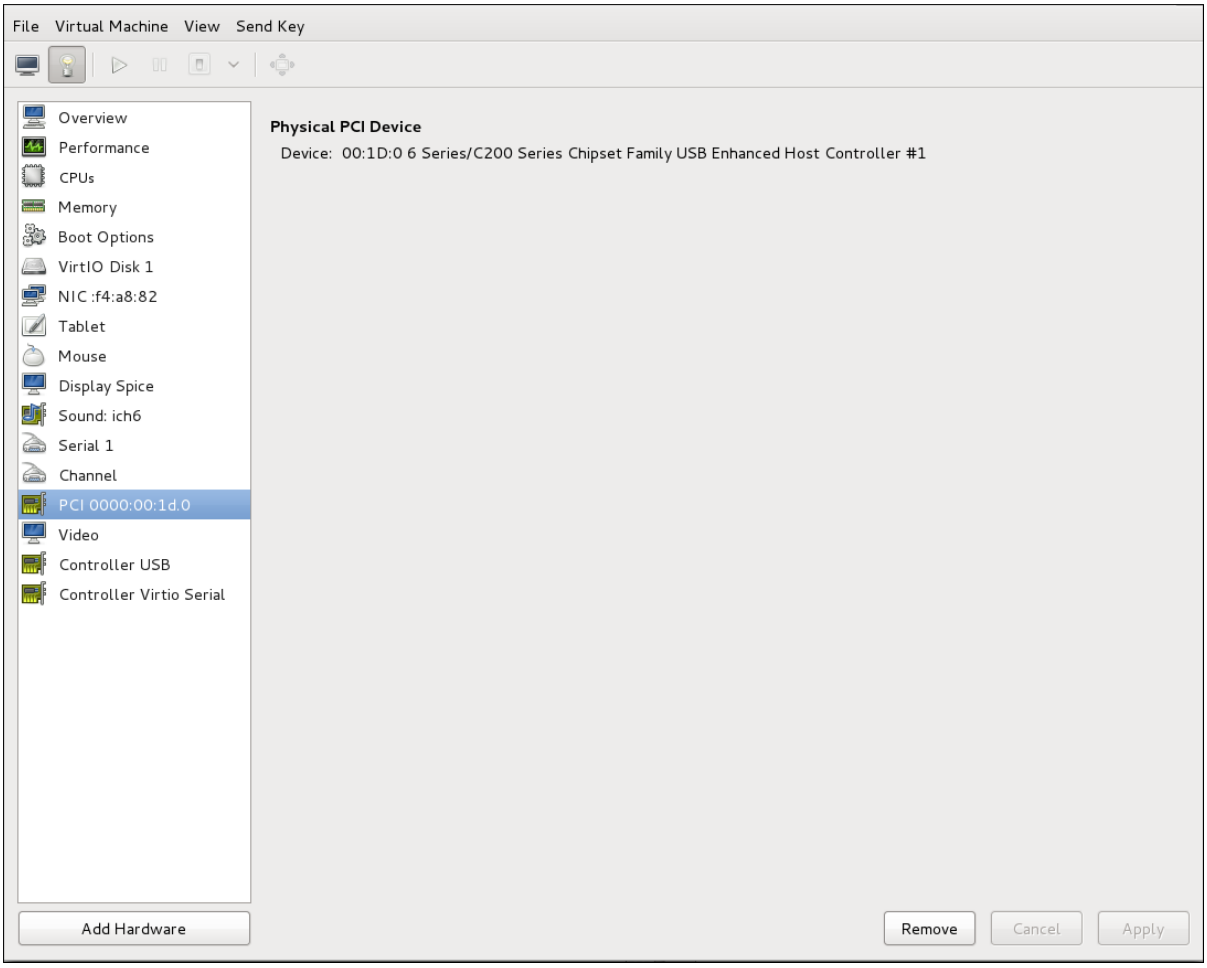


Figure 16.8. Selecting the PCI device to be detached

Click the **Remove** button to confirm. The device is now available for host use.

16.1.5. Creating PCI Bridges

Peripheral Component Interconnects (PCI) bridges are used to attach to devices such as network cards, modems and sound cards. Just like their physical counterparts, virtual devices can also be attached to a PCI Bridge. In the past, only 31 PCI devices could be added to any guest virtual machine. Now, when a 31st PCI device is added, a PCI bridge is automatically placed in the 31st slot moving the additional PCI device to the PCI bridge. Each PCI bridge has 31 slots for 31 additional devices, all of which can be bridges. In this manner, over 900 devices can be available for guest virtual machines. Note that this action cannot be performed when the guest virtual machine is running. You must add the PCI device on a guest virtual machine that is shutdown.

16.1.5.1. PCI Bridge hot plug/hot unplug Support

PCI Bridge hot plug/hot unplug is supported on the following device types:

- virtio-net-pci
- virtio-scsi-pci
- e1000
- rtl8139
- virtio-serial-pci

- virtio-balloon-pci

16.1.6. PCI Device Assignment Restrictions

PCI device assignment (attaching PCI devices to virtual machines) requires host systems to have AMD IOMMU or Intel VT-d support to enable device assignment of PCIe devices.

Red Hat Enterprise Linux 7 has limited PCI configuration space access by guest device drivers. This limitation could cause drivers that are dependent on device capabilities or features present in the extended PCI configuration space, to fail configuration.

There is a limit of 32 total assigned devices per Red Hat Enterprise Linux 7 virtual machine. This translates to 32 total PCI functions, regardless of the number of PCI bridges present in the virtual machine or how those functions are combined to create multi-function slots.

Platform support for interrupt remapping is required to fully isolate a guest with assigned devices from the host. Without such support, the host may be vulnerable to interrupt injection attacks from a malicious guest. In an environment where guests are trusted, the administrator may opt-in to still allow PCI device assignment using the **allow_unsafe_interrupts** option to the ***vfio_iommu_type1*** module. This may either be done persistently by adding a .conf file (for example **local.conf**) to **/etc/modprobe.d** containing the following:

```
options vfio_iommu_type1 allow_unsafe_interrupts=1
```

or dynamically using the sysfs entry to do the same:

```
# echo 1 > /sys/module/vfio_iommu_type1/parameters/allow_unsafe_interrupts
```

16.2. PCI DEVICE ASSIGNMENT WITH SR-IOV DEVICES

A PCI network device (specified in the domain XML by the **<source>** element) can be directly connected to the guest using direct device assignment (sometimes referred to as *passthrough*). Due to limitations in standard single-port PCI ethernet card driver design, only *Single Root I/O Virtualization* (SR-IOV) *virtual function* (VF) devices can be assigned in this manner; to assign a standard single-port PCI or PCIe Ethernet card to a guest, use the traditional **<hostdev>** device definition.

```
     <devices>
    <interface type='hostdev'>
      <driver name='vfio'/>
      <source>
        <address type='pci' domain='0x0000' bus='0x00' slot='0x07'
function='0x0'/>
      </source>
      <mac address='52:54:00:6d:90:02'>
      <virtualport type='802.1Qbh'>
        <parameters profileid='finance'/>
      </virtualport>
    </interface>
  </devices>
```

Figure 16.9. XML example for PCI device assignment

Developed by the PCI-SIG (PCI Special Interest Group), the Single Root I/O Virtualization (SR-IOV) specification is a standard for a type of PCI device assignment that can share a single device to multiple virtual machines. SR-IOV improves device performance for virtual machines.

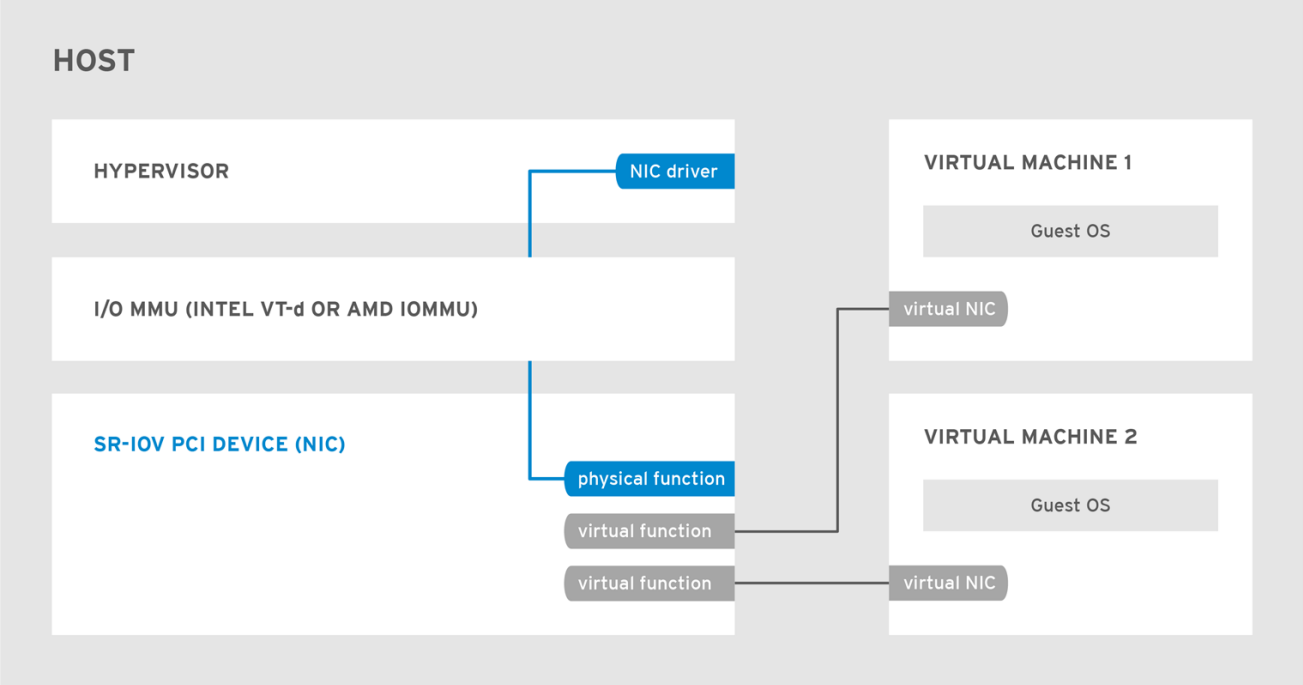


Figure 16.10. How SR-IOV works

SR-IOV enables a Single Root Function (for example, a single Ethernet port), to appear as multiple, separate, physical devices. A physical device with SR-IOV capabilities can be configured to appear in the PCI configuration space as multiple functions. Each device has its own configuration space complete with Base Address Registers (BARs).

SR-IOV uses two PCI functions:

- Physical Functions (PFs) are full PCIe devices that include the SR-IOV capabilities. Physical Functions are discovered, managed, and configured as normal PCI devices. Physical Functions configure and manage the SR-IOV functionality by assigning Virtual Functions.
- Virtual Functions (VFs) are simple PCIe functions that only process I/O. Each Virtual Function is derived from a Physical Function. The number of Virtual Functions a device may have is limited by the device hardware. A single Ethernet port, the Physical Device, may map to many Virtual Functions that can be shared to virtual machines.

The hypervisor can assign one or more Virtual Functions to a virtual machine. The Virtual Function's configuration space is then assigned to the configuration space presented to the guest.

Each Virtual Function can only be assigned to a single guest at a time, as Virtual Functions require real hardware resources. A virtual machine can have multiple Virtual Functions. A Virtual Function appears as a network card in the same way as a normal network card would appear to an operating system.

The SR-IOV drivers are implemented in the kernel. The core implementation is contained in the PCI subsystem, but there must also be driver support for both the Physical Function (PF) and Virtual Function (VF) devices. An SR-IOV capable device can allocate VFs from a PF. The VFs appear as PCI devices which are backed on the physical PCI device by resources such as queues and register sets.

16.2.1. Advantages of SR-IOV

SR-IOV devices can share a single physical port with multiple virtual machines.

When an SR-IOV VF is assigned to a virtual machine, it can be configured to (transparently to the virtual machine) place all network traffic leaving the VF onto a particular VLAN. The virtual machine cannot detect that its traffic is being tagged for a VLAN, and will be unable to change or eliminate this tagging.

Virtual Functions have near-native performance and provide better performance than paravirtualized drivers and emulated access. Virtual Functions provide data protection between virtual machines on the same physical server as the data is managed and controlled by the hardware.

These features allow for increased virtual machine density on hosts within a data center.

SR-IOV is better able to utilize the bandwidth of devices with multiple guests.

16.2.2. Using SR-IOV

This section covers the use of PCI passthrough to assign a Virtual Function of an SR-IOV capable multiport network card to a virtual machine as a network device.

SR-IOV Virtual Functions (VFs) can be assigned to virtual machines by adding a device entry in **<*hostdev*>** with the **`virsh edit`** or **`virsh attach-device`** command. However, this can be problematic because unlike a regular network device, an SR-IOV VF network device does not have a permanent unique MAC address, and is assigned a new MAC address each time the host is rebooted. Because of this, even if the guest is assigned the same VF after a reboot, when the host is rebooted the guest determines its network adapter to have a new MAC address. As a result, the guest believes there is new hardware connected each time, and will usually require re-configuration of the guest's network settings.

libvirt contains the **<*interface type='hostdev'*>** interface device. Using this interface device, **libvirt** will first perform any network-specific hardware/switch initialization indicated (such as setting the MAC address, VLAN tag, or 802.1Qbh virtualport parameters), then perform the PCI device assignment to the guest.

Using the **<*interface type='hostdev'*>** interface device requires:

- an SR-IOV-capable network card,
- host hardware that supports either the Intel VT-d or the AMD IOMMU extensions
- the PCI address of the VF to be assigned.

IMPORTANT

Assignment of an SR-IOV device to a virtual machine requires that the host hardware supports the Intel VT-d or the AMD IOMMU specification.

To attach an SR-IOV network device on an Intel or an AMD system, follow this procedure:

Procedure 16.8. Attach an SR-IOV network device on an Intel or AMD system

1. **Enable Intel VT-d or the AMD IOMMU specifications in the BIOS and kernel**

On an Intel system, enable Intel VT-d in the BIOS if it is not enabled already. See Procedure 16.1, “Preparing an Intel system for PCI device assignment” for procedural help on enabling Intel VT-d in the BIOS and kernel.

Skip this step if Intel VT-d is already enabled and working.

On an AMD system, enable the AMD IOMMU specifications in the BIOS if they are not enabled already. See Procedure 16.2, “Preparing an AMD system for PCI device assignment” for procedural help on enabling IOMMU in the BIOS.

2. **Verify support**
   Verify if the PCI device with SR-IOV capabilities is detected. This example lists an Intel 82576 network interface card which supports SR-IOV. Use the **lspci** command to verify whether the device was detected.

```
# lspci
03:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
03:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
```

   Note that the output has been modified to remove all other devices.

3. **Activate Virtual Functions**
   Run the following command:

```
# echo ${num_vfs} > /sys/class/net/enp14s0f0/device/sriov_numvfs
```

4. **Make the Virtual Functions persistent**
   To make the Virtual Functions persistent across reboots, use the editor of your choice to create an udev rule similar to the following, where you specify the intended number of VFs (in this example, ***2***), up to the limit supported by the network interface card. In the following example, replace *enp14s0f0* with the PF network device name(s) and adjust the value of **ENV{ID_NET_DRIVER}** to match the driver in use:

```
# vim /etc/udev/rules.d/enp14s0f0.rules
```

```
ACTION=="add", SUBSYSTEM=="net", ENV{ID_NET_DRIVER}=="ixgbe",
ATTR{device/sriov_numvfs}="2"
```

   This will ensure the feature is enabled at boot-time.

5. **Inspect the new Virtual Functions**
   Using the **lspci** command, list the newly added Virtual Functions attached to the Intel 82576 network device. (Alternatively, use **grep** to search for **Virtual Function**, to search for devices that support Virtual Functions.)

```
# lspci | grep 82576
0b:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
0b:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network
Connection (rev 01)
0b:10.0 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
```

```
0b:10.1 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:10.2 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:10.3 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:10.4 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:10.5 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:10.6 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:10.7 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:11.0 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:11.1 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:11.2 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:11.3 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:11.4 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
0b:11.5 Ethernet controller: Intel Corporation 82576 Virtual
Function (rev 01)
```

The identifier for the PCI device is found with the ***-n*** parameter of the **lspci** command. The Physical Functions correspond to **0b:00.0** and **0b:00.1**. All Virtual Functions have **Virtual Function** in the description.

6. **Verify devices exist with virsh**
   The **libvirt** service must recognize the device before adding a device to a virtual machine. **libvirt** uses a similar notation to the **lspci** output. All punctuation characters, **:** and **.**, in **lspci** output are changed to underscores (_).

   Use the **virsh nodedev-list** command and the **grep** command to filter the Intel 82576 network device from the list of available host devices. ***0b*** is the filter for the Intel 82576 network devices in this example. This may vary for your system and may result in additional devices.

```
# virsh nodedev-list | grep 0b
pci_0000_0b_00_0
pci_0000_0b_00_1
pci_0000_0b_10_0
pci_0000_0b_10_1
pci_0000_0b_10_2
pci_0000_0b_10_3
pci_0000_0b_10_4
pci_0000_0b_10_5
pci_0000_0b_10_6
pci_0000_0b_11_7
pci_0000_0b_11_1
pci_0000_0b_11_2
```

```
pci_0000_0b_11_3
pci_0000_0b_11_4
pci_0000_0b_11_5
```

The PCI addresses for the Virtual Functions and Physical Functions should be in the list.

7. **Get device details with virsh**
   The **pci_0000_0b_00_0** is one of the Physical Functions and **pci_0000_0b_10_0** is the first corresponding Virtual Function for that Physical Function. Use the **virsh nodedev-dumpxml** command to get device details for both devices.

```
# virsh nodedev-dumpxml pci_0000_03_00_0
<device>
  <name>pci_0000_03_00_0</name>
  <path>/sys/devices/pci0000:00/0000:00:01.0/0000:03:00.0</path>
  <parent>pci_0000_00_01_0</parent>
  <driver>
    <name>igb</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>3</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x10c9'>82576 Gigabit Network Connection</product>
    <vendor id='0x8086'>Intel Corporation</vendor>
    <capability type='virt_functions'>
      <address domain='0x0000' bus='0x03' slot='0x10'
function='0x0'/>
      <address domain='0x0000' bus='0x03' slot='0x10'
function='0x2'/>
      <address domain='0x0000' bus='0x03' slot='0x10'
function='0x4'/>
      <address domain='0x0000' bus='0x03' slot='0x10'
function='0x6'/>
      <address domain='0x0000' bus='0x03' slot='0x11'
function='0x0'/>
      <address domain='0x0000' bus='0x03' slot='0x11'
function='0x2'/>
      <address domain='0x0000' bus='0x03' slot='0x11'
function='0x4'/>
    </capability>
    <iommuGroup number='14'>
      <address domain='0x0000' bus='0x03' slot='0x00'
function='0x0'/>
      <address domain='0x0000' bus='0x03' slot='0x00'
function='0x1'/>
    </iommuGroup>
  </capability>
</device>
```

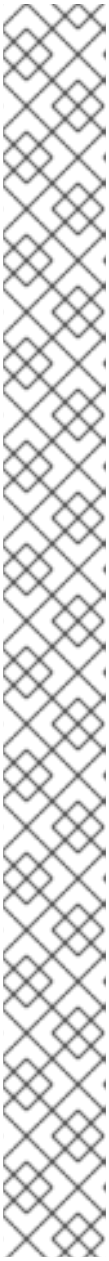
```
# virsh nodedev-dumpxml pci_0000_03_11_5
<device>
  <name>pci_0000_03_11_5</name>
  <path>/sys/devices/pci0000:00/0000:00:01.0/0000:03:11.5</path>
```

```
  <parent>pci_0000_00_01_0</parent>
  <driver>
    <name>igbvf</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>3</bus>
    <slot>17</slot>
    <function>5</function>
    <product id='0x10ca'>82576 Virtual Function</product>
    <vendor id='0x8086'>Intel Corporation</vendor>
    <capability type='phys_function'>
      <address domain='0x0000' bus='0x03' slot='0x00'
function='0x1'/>
    </capability>
    <iommuGroup number='35'>
      <address domain='0x0000' bus='0x03' slot='0x11'
function='0x5'/>
    </iommuGroup>
  </capability>
</device>
```

This example adds the Virtual Function **pci_0000_03_10_2** to the virtual machine in Step 8. Note the **bus**, **slot** and **function** parameters of the Virtual Function: these are required for adding the device.

Copy these parameters into a temporary XML file, such as **/tmp/new-interface.xml** for example.

```
   <interface type='hostdev' managed='yes'>
     <source>
       <address type='pci' domain='0x0000' bus='0x03' slot='0x10'
function='0x2'/>
     </source>
   </interface>
```



NOTE

When the virtual machine starts, it should see a network device of the type provided by the physical adapter, with the configured MAC address. This MAC address will remain unchanged across host and guest reboots.

The following ***<interface>*** example shows the syntax for the optional ***<mac address>***, ***<virtualport>***, and ***<vlan>*** elements. In practice, use either the ***<vlan>*** or ***<virtualport>*** element, not both simultaneously as shown in the example:

```
...
 <devices>
   ...
   <interface type='hostdev' managed='yes'>
     <source>
       <address type='pci' domain='0' bus='11' slot='16'
function='0'/>
     </source>
     <mac address='52:54:00:6d:90:02'>
     <vlan>
        <tag id='42'/>
     </vlan>
     <virtualport type='802.1Qbh'>
       <parameters profileid='finance'/>
     </virtualport>
   </interface>
   ...
 </devices>
```

If you do not specify a MAC address, one will be automatically generated. The ***<virtualport>*** element is only used when connecting to an 802.11Qbh hardware switch. The ***<vlan>*** element will transparently put the guest's device on the VLAN tagged ***42***.

8. **Add the Virtual Function to the virtual machine**
   Add the Virtual Function to the virtual machine using the following command with the temporary file created in the previous step. This attaches the new device immediately and saves it for subsequent guest restarts.

   ```
   virsh attach-device MyGuest /tmp/new-interface.xml --live --config
   ```

   Specifying the **--live** option with **virsh attach-device** attaches the new device to the running guest. Using the **--config** option ensures the new device is available after future guest restarts.

NOTE

The **--live** option is only accepted when the guest is running. **virsh** will return an error if the **--live** option is used on a non-running guest.

The virtual machine detects a new network interface card. This new card is the Virtual Function of the SR-IOV device.

16.2.3. Configuring PCI Assignment with SR-IOV Devices

SR-IOV network cards provide multiple VFs that can each be individually assigned to a guest virtual machines using PCI device assignment. Once assigned, each behaves as a full physical network device. This permits many guest virtual machines to gain the performance advantage of direct PCI device assignment, while only using a single slot on the host physical machine.

These VFs can be assigned to guest virtual machines in the traditional manner using the **<hostdev>** element. However, SR-IOV VF network devices do not have permanent unique MAC addresses, which causes problems where the guest virtual machine's network settings need to be re-configured each time the host physical machine is rebooted. To fix this, you need to set the MAC address prior to assigning the VF to the host physical machine after every boot of the guest virtual machine. In order to assign this MAC address, as well as other options, see the following procedure:

Procedure 16.9. Configuring MAC addresses, vLAN, and virtual ports for assigning PCI devices on SR-IOV

The **<hostdev>** element cannot be used for function-specific items like MAC address assignment, vLAN tag ID assignment, or virtual port assignment, because the **<mac>**, **<vlan>**, and **<virtualport>** elements are not valid children for **<hostdev>**. Instead, these elements can be used with the hostdev interface type: **<interface type='hostdev'>**. This device type behaves as a hybrid of an **<interface>** and **<hostdev>**. Thus, before assigning the PCI device to the guest virtual machine, libvirt initializes the network-specific hardware/switch that is indicated (such as setting the MAC address, setting a vLAN tag, or associating with an 802.1Qbh switch) in the guest virtual machine's XML configuration file. For information on setting the vLAN tag, see Section 17.16, “Setting vLAN Tags”.

1. **Gather information**
   In order to use **<interface type='hostdev'>**, you must have an SR-IOV-capable network card, host physical machine hardware that supports either the Intel VT-d or AMD IOMMU extensions, and you must know the PCI address of the VF that you wish to assign.

2. **Shut down the guest virtual machine**
   Using **virsh shutdown** command, shut down the guest virtual machine (here named *guestVM*).

   ```
   # virsh shutdown guestVM
   ```

3. **Open the XML file for editing**
   Run the **virsh save-image-edit** command to open the XML file for editing (refer to Section 20.7.5, “Editing the Guest Virtual Machine Configuration” for more information) with the **--running** option. The name of the configuration file in this example is *guestVM.xml*.

   ```
   # virsh save-image-edit guestVM.xml --running
   ```

   The *guestVM.xml* opens in your default editor.

4. **Edit the XML file**
   Update the configuration file (*guestVM.xml*) to have a **<devices>** entry similar to the following:

```
 <devices>
   ...
   <interface type='hostdev' managed='yes'>
     <source>
       <address type='pci' domain='0x0' bus='0x00' slot='0x07'
function='0x0'/> <!--these values can be decimal as well-->
     </source>
     <mac address='52:54:00:6d:90:02'/>
<!--sets the mac address-->
     <virtualport type='802.1Qbh'>
<!--sets the virtual port for the 802.1Qbh switch-->
       <parameters profileid='finance'/>
     </virtualport>
     <vlan>
<!--sets the vlan tag-->
      <tag id='42'/>
     </vlan>
   </interface>
   ...
 </devices>
```

Figure 16.11. Sample domain XML for hostdev interface type

Note that if you do not provide a MAC address, one will be automatically generated, just as with any other type of interface device. In addition, the **<virtualport>** element is only used if you are connecting to an 802.11Qgh hardware switch. 802.11Qbg (also known as "VEPA") switches are currently not supported.

5. **Restart the guest virtual machine**
   Run the **virsh start** command to restart the guest virtual machine you shut down in step 2. See Section 20.6, “Starting, Resuming, and Restoring a Virtual Machine” for more information.

   ```
   # virsh start guestVM
   ```

   When the guest virtual machine starts, it sees the network device provided to it by the physical host machine's adapter, with the configured MAC address. This MAC address remains unchanged across guest virtual machine and host physical machine reboots.

16.2.4. Setting PCI device assignment from a pool of SR-IOV virtual functions

Hard coding the PCI addresses of particular Virtual Functions (VFs) into a guest's configuration has two serious limitations:

- The specified VF must be available any time the guest virtual machine is started. Therefore, the administrator must permanently assign each VF to a single guest virtual machine (or modify the configuration file for every guest virtual machine to specify a currently unused VF's PCI address each time every guest virtual machine is started).
- If the guest virtual machine is moved to another host physical machine, that host physical machine must have exactly the same hardware in the same location on the PCI bus (or the guest virtual machine configuration must be modified prior to start).

It is possible to avoid both of these problems by creating a libvirt network with a device pool containing

all the VFs of an SR-IOV device. Once that is done, configure the guest virtual machine to reference this network. Each time the guest is started, a single VF will be allocated from the pool and assigned to the guest virtual machine. When the guest virtual machine is stopped, the VF will be returned to the pool for use by another guest virtual machine.

Procedure 16.10. Creating a device pool

1. **Shut down the guest virtual machine**
   Using **virsh shutdown** command, shut down the guest virtual machine, here named *guestVM*.

   ```
   # virsh shutdown guestVM
   ```

2. **Create a configuration file**
   Using your editor of choice, create an XML file (named *passthrough.xml*, for example) in the **/tmp** directory. Make sure to replace **pf dev='eth3'** with the netdev name of your own SR-IOV device's *Physical Function* (PF).

   The following is an example network definition that will make available a pool of all VFs for the SR-IOV adapter with its PF at "eth3' on the host physical machine:

   ```
   <network>
      <name>passthrough</name> <!-- This is the name of the file you
   created -->
      <forward mode='hostdev' managed='yes'>
        <pf dev='myNetDevName'/>  <!-- Use the netdev name of your SR-
   IOV devices PF here -->
      </forward>
   </network>
   ```

   Figure 16.12. Sample network definition domain XML

3. **Load the new XML file**
   Enter the following command, replacing */tmp/passthrough.xml* with the name and location of your XML file you created in the previous step:

   ```
   # virsh net-define /tmp/passthrough.xml
   ```

4. **Restarting the guest**
   Run the following, replacing *passthrough.xml* with the name of your XML file you created in the previous step:

   ```
   # virsh net-autostart passthrough # virsh net-start passthrough
   ```

5. **Re-start the guest virtual machine**
   Run the **virsh start** command to restart the guest virtual machine you shutdown in the first step (example uses guestVM as the guest virtual machine's domain name). See Section 20.6, “Starting, Resuming, and Restoring a Virtual Machine” for more information.

   ```
   # virsh start guestVM
   ```

6. **Initiating passthrough for devices**
   Although only a single device is shown, libvirt will automatically derive the list of all VFs associated with that PF the first time a guest virtual machine is started with an interface definition in its domain XML like the following:

```
<interface type='network'>
   <source network='passthrough'>
</interface>
```

Figure 16.13. Sample domain XML for interface network definition

7. **Verification**
   You can verify this by running **`virsh net-dumpxml passthrough`** command after starting the first guest that uses the network; you will get output similar to the following:

```
<network connections='1'>
   <name>passthrough</name>
   <uuid>a6b49429-d353-d7ad-3185-4451cc786437</uuid>
   <forward mode='hostdev' managed='yes'>
     <pf dev='eth3'/>
     <address type='pci' domain='0x0000' bus='0x02' slot='0x10'
function='0x1'/>
     <address type='pci' domain='0x0000' bus='0x02' slot='0x10'
function='0x3'/>
     <address type='pci' domain='0x0000' bus='0x02' slot='0x10'
function='0x5'/>
     <address type='pci' domain='0x0000' bus='0x02' slot='0x10'
function='0x7'/>
     <address type='pci' domain='0x0000' bus='0x02' slot='0x11'
function='0x1'/>
     <address type='pci' domain='0x0000' bus='0x02' slot='0x11'
function='0x3'/>
     <address type='pci' domain='0x0000' bus='0x02' slot='0x11'
function='0x5'/>
   </forward>
</network>
```

Figure 16.14. XML dump file *passthrough* contents

16.2.5. SR-IOV Restrictions

SR-IOV is only thoroughly tested with the following devices:

- Intel® 82576NS Gigabit Ethernet Controller (**`igb`** driver)
- Intel® 82576EB Gigabit Ethernet Controller (**`igb`** driver)
- Intel® 82599ES 10 Gigabit Ethernet Controller (**`ixgbe`** driver)

- Intel® 82599EB 10 Gigabit Ethernet Controller (**ixgbe** driver)

Other SR-IOV devices may work but have not been tested at the time of release

16.3. USB DEVICES

This section gives the commands required for handling USB devices.

16.3.1. Assigning USB Devices to Guest Virtual Machines

Most devices such as web cameras, card readers, disk drives, keyboards, mice are connected to a computer using a USB port and cable. There are two ways to pass such devices to a guest virtual machine:

- Using USB passthrough - this requires the device to be physically connected to the host physical machine that is hosting the guest virtual machine. SPICE is not needed in this case. USB devices on the host can be passed to the guest in the command line or **virt-manager**. See Section 19.3.2, “Attaching USB Devices to a Guest Virtual Machine” for **virt manager** directions. Note that the **virt-manager** directions are not suitable for hot plugging or hot unplugging devices. If you want to hot plug/or hot unplug a USB device, see Procedure 20.4, “Hot plugging USB devices for use by the guest virtual machine”.

- Using USB re-direction - USB re-direction is best used in cases where there is a host physical machine that is running in a data center. The user connects to his/her guest virtual machine from a local machine or thin client. On this local machine there is a SPICE client. The user can attach any USB device to the thin client and the SPICE client will redirect the device to the host physical machine on the data center so it can be used by the guest virtual machine that is running on the thin client. For instructions via the virt-manager see Section 19.3.3, “USB Redirection”.

16.3.2. Setting a Limit on USB Device Redirection

To filter out certain devices from redirection, pass the filter property to ***-device usb-redir***. The filter property takes a string consisting of filter rules, the format for a rule is:

```
<class>:<vendor>:<product>:<version>:<allow>
```

Use the value ***-1*** to designate it to accept any value for a particular field. You may use multiple rules on the same command line using | as a separator. Note that if a device matches none of the passed in rules, redirecting it will not be allowed!

Example 16.1. An example of limiting redirection with a guest virtual machine

1. Prepare a guest virtual machine.

2. Add the following code excerpt to the guest virtual machine's' domain XML file:

```
    <redirdev bus='usb' type='spicevmc'>
      <alias name='redir0'/>
      <address type='usb' bus='0' port='3'/>
    </redirdev>
    <redirfilter>
      <usbdev class='0x08' vendor='0x1234' product='0xBEEF'
version='2.0' allow='yes'/>
```

```
      <usbdev class='-1' vendor='-1' product='-1' version='-1'
allow='no'/>
    </redirfilter>
```

3. Start the guest virtual machine and confirm the setting changes by running the following:

```
#ps -ef | grep $guest_name
```

```
-device usb-redir,chardev=charredir0,id=redir0,/
filter=0x08:0x1234:0xBEEF:0x0200:1|-1:-1:-1:-1:0,bus=usb.0,port=3
```

4. Plug a USB device into a host physical machine, and use **virt-manager** to connect to the guest virtual machine.

5. Click **USB device selection** in the menu, which will produce the following message: "Some USB devices are blocked by host policy". Click **OK** to confirm and continue.

   The filter takes effect.

6. To make sure that the filter captures properly check the USB device vendor and product, then make the following changes in the host physical machine's domain XML to allow for USB redirection.

```
   <redirfilter>
      <usbdev class='0x08' vendor='0x0951' product='0x1625'
version='2.0' allow='yes'/>
      <usbdev allow='no'/>
    </redirfilter>
```

7. Restart the guest virtual machine, then use **virt-viewer** to connect to the guest virtual machine. The USB device will now redirect traffic to the guest virtual machine.

16.4. CONFIGURING DEVICE CONTROLLERS

Depending on the guest virtual machine architecture, some device buses can appear more than once, with a group of virtual devices tied to a virtual controller. Normally, libvirt can automatically infer such controllers without requiring explicit XML markup, but in some cases it is better to explicitly set a virtual controller element.

```
  ...
  <devices>
    <controller type='ide' index='0'/>
    <controller type='virtio-serial' index='0' ports='16' vectors='4'/>
    <controller type='virtio-serial' index='1'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x0a'
function='0x0'/>
    </controller>
    ...
  </devices>
  ...
```

Figure 16.15. Domain XML example for virtual controllers

Each controller has a mandatory attribute **<controller type>**, which must be one of:

- ide
- fdc
- scsi
- sata
- usb
- ccid
- virtio-serial
- pci

The **<controller>** element has a mandatory attribute **<controller index>** which is the decimal integer describing in which order the bus controller is encountered (for use in controller attributes of **<address>** elements). When **<controller type ='virtio-serial'>** there are two additional optional attributes (named **ports** and **vectors**), which control how many devices can be connected through the controller.

When **<controller type ='scsi'>**, there is an optional attribute **model** model, which can have the following values:

- auto
- buslogic
- ibmvscsi
- lsilogic
- lsisas1068
- lsisas1078
- virtio-scsi
- vmpvscsi

When **<controller type ='usb'>**, there is an optional attribute **model** model, which can have the following values:

- piix3-uhci
- piix4-uhci
- ehci
- ich9-ehci1
- ich9-uhci1
- ich9-uhci2
- ich9-uhci3
- vt82c686b-uhci
- pci-ohci
- nec-xhci

Note that if the USB bus needs to be explicitly disabled for the guest virtual machine, **<model='none'>** may be used. .

For controllers that are themselves devices on a PCI or USB bus, an optional sub-element **<address>** can specify the exact relationship of the controller to its master bus, with semantics as shown in Section 16.5, “Setting Addresses for Devices”.

An optional sub-element **<driver>** can specify the driver-specific options. Currently, it only supports attribute queues, which specifies the number of queues for the controller. For best performance, it is recommended to specify a value matching the number of vCPUs.

USB companion controllers have an optional sub-element **<master>** to specify the exact relationship of the companion to its master controller. A companion controller is on the same bus as its master, so the companion **index** value should be equal.

An example XML which can be used is as follows:

```
    ...
  <devices>
    <controller type='usb' index='0' model='ich9-ehci1'>
      <address type='pci' domain='0' bus='0' slot='4' function='7'/>
    </controller>
    <controller type='usb' index='0' model='ich9-uhci1'>
      <master startport='0'/>
      <address type='pci' domain='0' bus='0' slot='4' function='0'
multifunction='on'/>
    </controller>
    ...
  </devices>
  ...
```

Figure 16.16. Domain XML example for USB controllers

PCI controllers have an optional **model** attribute with the following possible values:

- pci-root
- pcie-root
- pci-bridge
- dmi-to-pci-bridge

For machine types which provide an implicit PCI bus, the pci-root controller with **index='0'** is auto-added and required to use PCI devices. pci-root has no address. PCI bridges are auto-added if there are too many devices to fit on the one bus provided by **model='pci-root'**, or a PCI bus number greater than zero was specified. PCI bridges can also be specified manually, but their addresses should only see PCI buses provided by already specified PCI controllers. Leaving gaps in the PCI controller indexes might lead to an invalid configuration. The following XML example can be added to the **<devices>** section:

```
  ...
  <devices>
    <controller type='pci' index='0' model='pci-root'/>
    <controller type='pci' index='1' model='pci-bridge'>
      <address type='pci' domain='0' bus='0' slot='5' function='0'
multifunction='off'/>
    </controller>
  </devices>
  ...
```

Figure 16.17. Domain XML example for PCI bridge

For machine types which provide an implicit PCI Express (PCIe) bus (for example, the machine types based on the Q35 chipset), the pcie-root controller with **index='0'** is auto-added to the domain's configuration. pcie-root has also no address, but provides 31 slots (numbered 1-31) and can only be used to attach PCIe devices. In order to connect standard PCI devices on a system which has a pcie-root controller, a pci controller with **model='dmi-to-pci-bridge'** is automatically added. A dmi-to-

pci-bridge controller plugs into a PCIe slot (as provided by pcie-root), and itself provides 31 standard PCI slots (which are not hot-pluggable). In order to have hot-pluggable PCI slots in the guest system, a pci-bridge controller will also be automatically created and connected to one of the slots of the auto-created dmi-to-pci-bridge controller; all guest devices with PCI addresses that are auto-determined by libvirt will be placed on this pci-bridge device.

```
    ...
  <devices>
    <controller type='pci' index='0' model='pcie-root'/>
    <controller type='pci' index='1' model='dmi-to-pci-bridge'>
      <address type='pci' domain='0' bus='0' slot='0xe' function='0'/>
    </controller>
    <controller type='pci' index='2' model='pci-bridge'>
      <address type='pci' domain='0' bus='1' slot='1' function='0'/>
    </controller>
  </devices>
  ...
```

Figure 16.18. Domain XML example for PCIe (PCI express)

The following XML configuration is used for USB 3.0 / xHCI emulation:

```
    ...
  <devices>
    <controller type='usb' index='3' model='nec-xhci'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x0f'
function='0x0'/>
    </controller>
  </devices>
    ...
```

Figure 16.19. Domain XML example for USB3/xHCI devices

16.5. SETTING ADDRESSES FOR DEVICES

Many devices have an optional **<address>** sub-element which is used to describe where the device is placed on the virtual bus presented to the guest virtual machine. If an address (or any optional attribute within an address) is omitted on input, libvirt will generate an appropriate address; but an explicit address is required if more control over layout is required. For domain XML device examples that include an **<address>** element, see Figure 16.9, “XML example for PCI device assignment”.

Every address has a mandatory attribute **type** that describes which bus the device is on. The choice of which address to use for a given device is constrained in part by the device and the architecture of the guest virtual machine. For example, a **<disk>** device uses **type='drive'**, while a **<console>** device would use **type='pci'** on i686 or x86_64 guest virtual machine architectures. Each address type has further optional attributes that control where on the bus the device will be placed as described in the table:

Table 16.1. Supported device address types

Address type	Description
type='pci'	PCI addresses have the following additional attributes: • domain (a 2-byte hex integer, not currently used by qemu) • bus (a hex value between 0 and 0xff, inclusive) • slot (a hex value between 0x0 and 0x1f, inclusive) • function (a value between 0 and 7, inclusive) • multifunction controls turning on the multifunction bit for a particular slot/function in the PCI control register By default it is set to 'off', but should be set to 'on' for function 0 of a slot that will have multiple functions used.
type='drive'	Drive addresses have the following additional attributes: • controller (a 2-digit controller number) • bus (a 2-digit bus number • target (a 2-digit bus number) • unit (a 2-digit unit number on the bus)
type='virtio-serial'	Each virtio-serial address has the following additional attributes: • controller (a 2-digit controller number) • bus (a 2-digit bus number) • slot (a 2-digit slot within the bus)
type='ccid'	A CCID address, for smart-cards, has the following additional attributes: • bus (a 2-digit bus number) • slot attribute (a 2-digit slot within the bus)

Address type	Description
type='usb'	USB addresses have the following additional attributes: • bus (a hex value between 0 and 0xfff, inclusive) • port (a dotted notation of up to four octets, such as 1.2 or 2.1.3.1)
type='isa'	ISA addresses have the following additional attributes: • iobase • irq

16.6. RANDOM NUMBER GENERATOR DEVICE

Random number generators are very important for operating system security. For securing virtual operating systems, Red Hat Enterprise Linux 7 includes **`virtio-rng`**, a virtual hardware random number generator device that can provide the guest with fresh entropy on request.

On the host physical machine, the hardware RNG interface creates a chardev at **`/dev/hwrng`**, which can be opened and then read to fetch entropy from the host physical machine. In co-operation with the **`rngd`** daemon, the entropy from the host physical machine can be routed to the guest virtual machine's **`/dev/random`**, which is the primary source of randomness.

Using a random number generator is particularly useful when a device such as a keyboard, mouse, and other inputs are not enough to generate sufficient entropy on the guest virtual machine. The virtual random number generator device allows the host physical machine to pass through entropy to guest virtual machine operating systems. This procedure can be performed using either the command line or the virt-manager interface. For more information about **`virtio-rng`**, see Red Hat Enterprise Linux Virtual Machines: Access to Random Numbers Made Easy.

Procedure 16.11. Implementing virtio-rng using the Virtual Machine Manager

1. Shut down the guest virtual machine.
2. Select the guest virtual machine and from the **Edit** menu, select **Virtual Machine Details**, to open the Details window for the specified guest virtual machine.
3. Click the **`Add Hardware`** button.
4. In the **Add New Virtual Hardware** window, select **RNG** to open the **Random Number Generator** window.

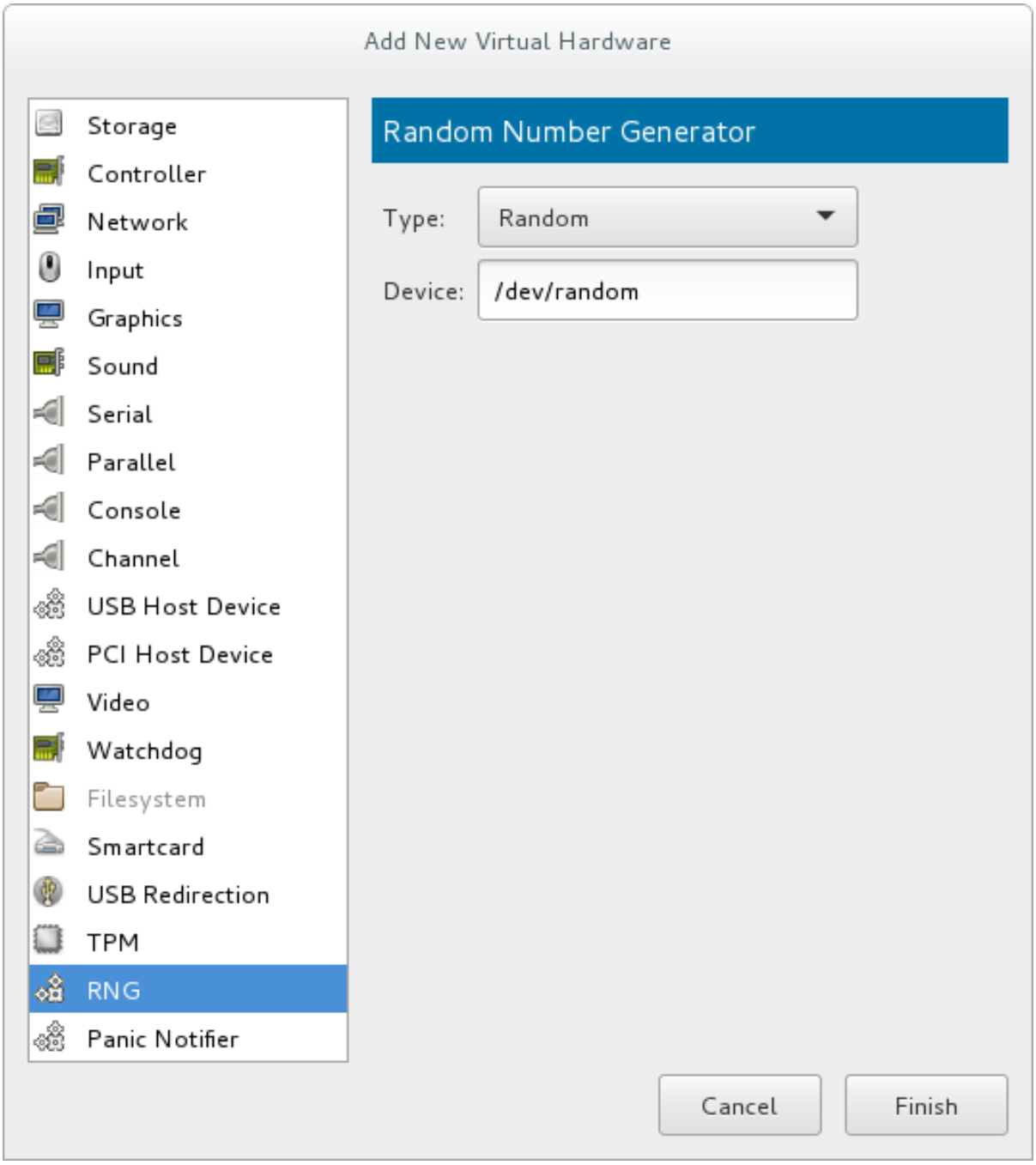


Figure 16.20. Random Number Generator window

Enter the intended parameters and click **Finish** when done. The parameters are explained in virtio-rng elements.

Procedure 16.12. Implementing virtio-rng using command-line tools

1. Shut down the guest virtual machine.
2. Using the **`virsh edit`** ***`domain-name`*** command, open the XML file for the intended guest virtual machine.
3. Edit the **`<devices>`** element to include the following:

```
...
<devices>
  <rng model='virtio'>
    <rate period='2000' bytes='1234'/>
    <backend model='random'>/dev/random</backend>
    <!-- OR -->
    <backend model='egd' type='udp'>
      <source mode='bind' service='1234'/>
      <source mode='connect' host='1.2.3.4' service='1234'/>
    </backend>
  </rng>
</devices>
...
```

Figure 16.21. Random number generator device

The random number generator device allows the following XML attributes and elements:

virtio-rng elements

- **<model>** - The required **model** attribute specifies what type of RNG device is provided.
- **<backend model>** - The **<backend>** element specifies the source of entropy to be used for the guest. The source model is configured using the **model** attribute. Supported source models include **'random'** and **'egd'** .
  - **<backend model='random'>** - This **<backend>** type expects a non-blocking character device as input. Examples of such devices are **/dev/random** and **/dev/urandom**. The file name is specified as contents of the **<backend>** element. When no file name is specified the hypervisor default is used.
  - **<backend model='egd'>** - This back end connects to a source using the EGD protocol. The source is specified as a character device. See character device host physical machine interface for more information.

16.7. ASSIGNING GPU DEVICES

To assign a GPU to a guest, use one of the following method:

- GPU PCI Device Assignment - Using this method, it is possible to remove a GPU device from the host and assign it to a single guest.
- NVIDIA vGPU Assignment - This method makes it possible to create multiple *mediated devices* from a physical GPU, and assign these devices as virtual GPUs to multiple guests. This is only supported on selected NVIDIA GPUs, and only one mediated device can be assigned to a single guest.

16.7.1. GPU PCI Device Assignment

Red Hat Enterprise Linux 7 supports PCI device assignment of the following PCIe-based GPU devices as non-VGA graphics devices:

- NVIDIA Quadro K-Series, M-Series, and P-Series (models 2000 series or higher)

- NVIDIA GRID K-Series
- NVIDIA Tesla K-Series and M-Series

Currently, up to two GPUs may be attached to the virtual machine, in addition to one of the standard emulated VGA interfaces. The emulated VGA is used for pre-boot and installation and the NVIDIA GPU takes over when the NVIDIA graphics drivers are loaded.

To assign a GPU to a guest virtual machine, you must enable the I/O Memory Management Unit (IOMMU) on the host machine, identify the GPU device by using the **lspci** command, detach the device from host, attach it to the guest, and configure Xorg on the guest - as described in the following procedures:

Procedure 16.13. Enable IOMMU support in the host machine kernel

1. **Edit the kernel command line**
   For an Intel VT-d system, IOMMU is activated by adding the ***intel_iommu=on*** and ***iommu=pt*** parameters to the kernel command line. For an AMD-Vi system, the option needed is ***amd_iommu=pt***. To enable this option, edit or add the **GRUB_CMDLINX_LINUX** line to the **/etc/sysconfig/grub** configuration file as follows:

   ```
   GRUB_CMDLINE_LINUX="rd.lvm.lv=vg_VolGroup00/LogVol01
   vconsole.font=latarcyrheb-sun16 rd.lvm.lv=vg_VolGroup_1/root
   vconsole.keymap=us $([ -x /usr/sbin/rhcrashkernel-param ]  &&
   /usr/sbin/rhcrashkernel-param || :) rhgb quiet intel_iommu=on
   iommu=pt"
   ```

   NOTE

   For further information on IOMMU, see Appendix E, *Working with IOMMU Groups*.

2. **Regenerate the boot loader configuration**
   For the changes to the kernel command line to apply, regenerate the boot loader configuration using the **grub2-mkconfig** command:

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

   Note that if you are using a UEFI-based host, the target file should be **/etc/grub2-efi.cfg**.

3. **Reboot the host**
   For the changes to take effect, reboot the host machine:

   ```
   # reboot
   ```

Procedure 16.14. Excluding the GPU device from binding to the host physical machine driver

For GPU assignment, it is recommended to exclude the device from binding to host drivers, as these drivers often do not support dynamic unbinding of the device.

1. **Identify the PCI bus address**
   To identify the PCI bus address and IDs of the device, run the following **lspci** command. In this example, a VGA controller such as an NVIDIA Quadro or GRID card is used:

```
# lspci -Dnn | grep VGA
0000:02:00.0 VGA compatible controller [0300]: NVIDIA Corporation
GK106GL [Quadro K4000] [10de:11fa] (rev a1)
```

The resulting search reveals that the PCI bus address of this device is 0000:02:00.0 and the PCI IDs for the device are 10de:11fa.

2. **Prevent the native host machine driver from using the GPU device**
   To prevent the native host machine driver from using the GPU device, you can use a PCI ID with the pci-stub driver. To do this, append the **pci-stub.ids** option, with the PCI IDs as its value, to the **GRUB_CMDLINX_LINUX** line located in the **/etc/sysconfig/grub** configuration file, for example as follows:

```
GRUB_CMDLINE_LINUX="rd.lvm.lv=vg_VolGroup00/LogVol01
vconsole.font=latarcyrheb-sun16 rd.lvm.lv=vg_VolGroup_1/root
vconsole.keymap=us $([ -x /usr/sbin/rhcrashkernel-param ]  &&
/usr/sbin/rhcrashkernel-param || :) rhgb quiet intel_iommu=on
iommu=pt pci-stub.ids=10de:11fa"
```

   To add additional PCI IDs for pci-stub, separate them with a comma.

3. **Regenerate the boot loader configuration**
   Regenerate the boot loader configuration using the **grub2-mkconfig** to include this option:

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

   Note that if you are using a UEFI-based host, the target file should be **/etc/grub2-efi.cfg**.

4. **Reboot the host machine**
   In order for the changes to take effect, reboot the host machine:

```
# reboot
```

Procedure 16.15. Optional: Editing the GPU IOMMU configuration

Prior to attaching the GPU device, editing its IOMMU configuration may be needed for the GPU to work properly on the guest.

1. **Display the XML information of the GPU**
   To display the settings of the GPU in XML form, you first need to convert its PCI bus address to libvirt-compatible format by appending **pci_** and converting delimiters to underscores. In this example, the GPU PCI device identified with the 0000:02:00.0 bus address (as obtained in the previous procedure) becomes **pci_0000_02_00_0**. Use the libvirt address of the device with the **virsh nodedev-dumpxml** to display its XML configuration:

```
# virsh nodedev-dumpxml pci_0000_02_00_0
```

```
<device>
 <name>pci_0000_02_00_0</name>
 <path>/sys/devices/pci0000:00/0000:00:03.0/0000:02:00.0</path>
 <parent>pci_0000_00_03_0</parent>
 <driver>
```

```
    <name>pci-stub</name>
  </driver>
  <capability type='pci'>
   <domain>0</domain>
   <bus>2</bus>
   <slot>0</slot>
   <function>0</function>
   <product id='0x11fa'>GK106GL [Quadro K4000]</product>
   <vendor id='0x10de'>NVIDIA Corporation</vendor>
      <!-- pay attention to the following lines -->
   <iommuGroup number='13'>
    <address domain='0x0000' bus='0x02' slot='0x00' function='0x0'/>
    <address domain='0x0000' bus='0x02' slot='0x00' function='0x1'/>
   </iommuGroup>
   <pci-express>
    <link validity='cap' port='0' speed='8' width='16'/>
    <link validity='sta' speed='2.5' width='16'/>
   </pci-express>
  </capability>
</device>
```

Note the **<iommuGroup>** element of the XML. The iommuGroup indicates a set of devices that are considered isolated from other devices due to IOMMU capabilities and PCI bus topologies. All of the endpoint devices within the iommuGroup (meaning devices that are not PCIe root ports, bridges, or switch ports) need to be unbound from the native host drivers in order to be assigned to a guest. In the example above, the group is composed of the GPU device (0000:02:00.0) as well as the companion audio device (0000:02:00.1). For more information, see Appendix E, *Working with IOMMU Groups*.

2. **Adjust IOMMU settings**
   In this example, assignment of NVIDIA audio functions is not supported due to hardware issues with legacy interrupt support. In addition, the GPU audio function is generally not useful without the GPU itself. Therefore, in order to assign the GPU to a guest, the audio function must first be detached from native host drivers. This can be done using one of the following:

   - Detect the PCI ID for the device and append it to the **pci-stub.ids** option in the **/etc/sysconfig/grub** file, as detailed in Procedure 16.14, “Excluding the GPU device from binding to the host physical machine driver”
   - Use the **virsh nodedev-detach** command, for example as follows:

     ```
     # virsh nodedev-detach pci_0000_02_00_1
     Device pci_0000_02_00_1 detached
     ```

Procedure 16.16. Attaching the GPU

The GPU can be attached to the guest using any of the following methods:

1. Using the **Virtual Machine Manager** interface. For details, see Section 16.1.2, “Assigning a PCI Device with virt-manager”.
2. Creating an XML configuration fragment for the GPU and attaching it with the **virsh attach-device**:
   1. Create an XML for the device, similar to the following:

```
<hostdev mode='subsystem' type='pci' managed='yes'>
 <driver name='vfio'/>
 <source>
  <address domain='0x0000' bus='0x02' slot='0x00'
function='0x0'/>
 </source>
</hostdev>
```

2. Save this to a file and run **`virsh attach-device [domain] [file] --persistent`** to include the XML in the guest configuration. Note that the assigned GPU is added in addition to the existing emulated graphics device in the guest machine. The assigned GPU is handled as a secondary graphics device in the virtual machine. Assignment as a primary graphics device is not supported and emulated graphics devices in the guest's XML should not be removed.

3. Editing the guest XML configuration using the **`virsh edit`** command and adding the appropriate XML segment manually.

Procedure 16.17. Ḿodifying the Xorg configuration on the guest

The GPU's PCI bus address on the guest will be different than on the host. To enable the host to use the GPU properly, configure the guest's Xorg display server to use the assigned GPU address:

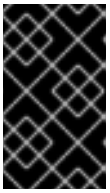
1. In the guest, use the **`lspci`** command to determine the PCI bus adress of the GPU:

```
# lspci | grep VGA
00:02.0 VGA compatible controller: Device 1234:111
00:09.0 VGA compatible controller: NVIDIA Corporation GK106GL
[Quadro K4000] (rev a1)
```

   In this example, the bus address is 00:09.0.

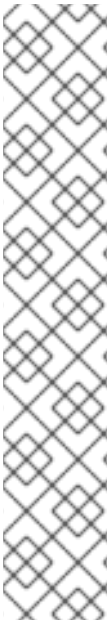
2. In the **`/etc/X11/xorg.conf`** file on the guest, add a **`BusID`** option with the detected address adjusted as follows:

```
Section "Device"
    Identifier     "Device0"
    Driver         "nvidia"
    VendorName     "NVIDIA Corporation"
    BusID          "PCI:0:9:0"
EndSection
```



IMPORTANT

If the bus address detected in Step 1 is hexadecimal, you need to convert the values between delimiters to the decimal system. For example, 00:0a.0 should be converted into PCI:0:10:0.



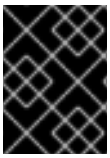
NOTE

When using an assigned NVIDIA GPU in the guest, only the NVIDIA drivers are supported. Other drivers may not work and may generate errors. For a Red Hat Enterprise Linux 7 guest, the nouveau driver can be blacklisted using the option **`modprobe.blacklist=nouveau`** on the kernel command line during install. For information on other guest virtual machines, see the operating system's specific documentation.

Depending on the guest operating system, with the NVIDIA drivers loaded, the guest may support using both the emulated graphics and assigned graphics simultaneously or may disable the emulated graphics. Note that access to the assigned graphics framebuffer is not provided by applications such as **virt-manager**. If the assigned GPU is not connected to a physical display, guest-based remoting solutions may be necessary to access the GPU desktop. As with all PCI device assignment, migration of a guest with an assigned GPU is not supported and each GPU is owned exclusively by a single guest. Depending on the guest operating system, hot plug support of GPUs may be available.

16.7.2. NVIDIA vGPU Assignment

The NVIDIA vGPU feature makes it possible to divide a physical GPU device into multiple virtual devices referred to as *mediated devices*. These mediated devices can then be assigned to multiple guests as virtual GPUs. As a result, these guests share the performance of a single physical GPU.



IMPORTANT

This feature is only available on a limited set of NVIDIA GPUs. For an up-to-date list of these devices, see the NVIDIA GPU Software Documentation.

NVIDIA vGPU Setup

To set up the vGPU feature, you first need to obtain NVIDIA vGPU drivers for your GPU device, then create mediated devices, and assign them to the intended guest machines:

1. Obtain the NVIDIA vGPU drivers and install them on your system. For instructions, see the NVIDIA documentation.

2. If the NVIDIA software installer did not create the **`/etc/modprobe.d/nvidia-installer-disable-nouveau.conf`** file, create a **`.conf`** file (of any name) in the **`/etc/modprobe.d/`** directory. Add the following lines in the file:

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

3. Regenerate the initial ramdisk for the current kernel, then reboot:

```
# dracut --force
# reboot
```

If you need to use a prior supported kernel version with mediated devices, regenerate the initial ramdisk for all installed kernel versions:

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

4. Check that the **nvidia_vgpu_vfio** module has been loaded by the kernel and that the **nvidia-vgpu-mgr.service** service is running.

```
# lsmod | grep nvidia_vgpu_vfio
nvidia_vgpu_vfio 45011 0
nvidia 14333621 10 nvidia_vgpu_vfio
mdev 20414 2 vfio_mdev,nvidia_vgpu_vfio
vfio 32695 3 vfio_mdev,nvidia_vgpu_vfio,vfio_iommu_type1
# systemctl status nvidia-vgpu-mgr.service
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 Fri 2018-03-16 10:17:36 CET; 5h
8min ago
 Main PID: 1553 (nvidia-vgpu-mgr)
 [...]
```

5. Write a device UUID to **/sys/class/mdev_bus/*pci_dev*/mdev_supported_types/*type-id*/create**, where *pci_dev* is the PCI address of the host GPU, and *type-id* is an ID of the host GPU type.

   The following example shows how to create a mediated device of **nvidia-63** vGPU type on an NVIDIA Tesla P4 card:

```
# uuidgen
30820a6f-b1a5-4503-91ca-0c10ba58692a
# echo "30820a6f-b1a5-4503-91ca-0c10ba58692a" >
/sys/class/mdev_bus/0000:01:00.0/mdev_supported_types/nvidia-
63/create
```

   For *type-id* values for specific devices, see *section 1.3.1. Virtual GPU Types* in Virtual GPU software documentation. Note that only Q-series NVIDIA vGPUs, such as **GRID P4-2Q**, are supported as mediated device GPU types on Linux guests.

6. Add the following lines to the **<devices/>** sections in XML configurations of guests that you want to share the vGPU resources. Use the UUID value generated by the **uuidgen** command in the previous step. Each UUID can only be assigned to one guest at a time.

```
<hostdev mode='subsystem' type='mdev' managed='no' model='vfio-pci'>
  <source>
    <address uuid='30820a6f-b1a5-4503-91ca-0c10ba58692a'/>
  </source>
</hostdev>
```

IMPORTANT

For the vGPU mediated devices to work properly on the assigned guests, NVIDIA vGPU guest software licensing needs to be set up for the guests. For further information and instructions, see the NVIDIA virtual GPU software documentation.

Removing NVIDIA vGPU Devices

To remove a mediated vGPU device, use the following command when the device is inactive, and replace *uuid* with the UUID of the device, for example **30820a6f-b1a5-4503-91ca-0c10ba58692a**.

```
# echo 1 > /sys/bus/mdev/devices/uuid/remove
```

Note that attempting to remove a vGPU device that is currently in use by a guest triggers the following error:

```
echo: write error: Device or resource busy
```

Querying NVIDIA vGPU Capabilities

To obtain additional information about the mediated devices on your system, such as how many mediated devices of a given type can be created, use the **virsh nodedev-list --cap mdev_types** and **virsh nodedev-dumpxml** commands. For example, the following displays available vGPU types on a Tesla P4 card:

```
$ virsh nodedev-list --cap mdev_types
pci_0000_01_00_0
$ virsh nodedev-dumpxml pci_0000_01_00_0
<...>
  <capability type='mdev_types'>
    <type id='nvidia-70'>
      <name>GRID P4-8A</name>
      <deviceAPI>vfio-pci</deviceAPI>
      <availableInstances>1</availableInstances>
    </type>
    <type id='nvidia-69'>
      <name>GRID P4-4A</name>
      <deviceAPI>vfio-pci</deviceAPI>
      <availableInstances>2</availableInstances>
    </type>
    <type id='nvidia-67'>
      <name>GRID P4-1A</name>
      <deviceAPI>vfio-pci</deviceAPI>
      <availableInstances>8</availableInstances>
    </type>
    <type id='nvidia-65'>
      <name>GRID P4-4Q</name>
      <deviceAPI>vfio-pci</deviceAPI>
      <availableInstances>2</availableInstances>
    </type>
    <type id='nvidia-63'>
      <name>GRID P4-1Q</name>
      <deviceAPI>vfio-pci</deviceAPI>
      <availableInstances>8</availableInstances>
    </type>
    <type id='nvidia-71'>
      <name>GRID P4-1B</name>
      <deviceAPI>vfio-pci</deviceAPI>
      <availableInstances>8</availableInstances>
    </type>
    <type id='nvidia-68'>
      <name>GRID P4-2A</name>
      <deviceAPI>vfio-pci</deviceAPI>
```

```
      <availableInstances>4</availableInstances>
    </type>
    <type id='nvidia-66'>
      <name>GRID P4-8Q</name>
      <deviceAPI>vfio-pci</deviceAPI>
      <availableInstances>1</availableInstances>
    </type>
    <type id='nvidia-64'>
      <name>GRID P4-2Q</name>
      <deviceAPI>vfio-pci</deviceAPI>
      <availableInstances>4</availableInstances>
    </type>
  </capability>
</...>
```

Remote Desktop Streaming Services for NVIDIA vGPU

The following remote desktop streaming services have been successfully tested for use with the NVIDIA vGPU feature on Red Hat Enterprise Linux 7:

- HP-RGS
- Mechdyne TGX - It is currently not possible to use Mechdyne TGX with Windows Server 2016 guests.
- NICE DCV - When using this streaming service, Red Hat recommends using fixed resolution settings, as using dynamic resolution in some cases results in a black screen.

CHAPTER 17. VIRTUAL NETWORKING

This chapter introduces the concepts needed to create, start, stop, remove, and modify virtual networks with libvirt.

Additional information can be found in the libvirt reference chapter

17.1. VIRTUAL NETWORK SWITCHES

Libvirt virtual networking uses the concept of a *virtual network switch*. A virtual network switch is a software construct that operates on a host physical machine server, to which virtual machines (guests) connect. The network traffic for a guest is directed through this switch:

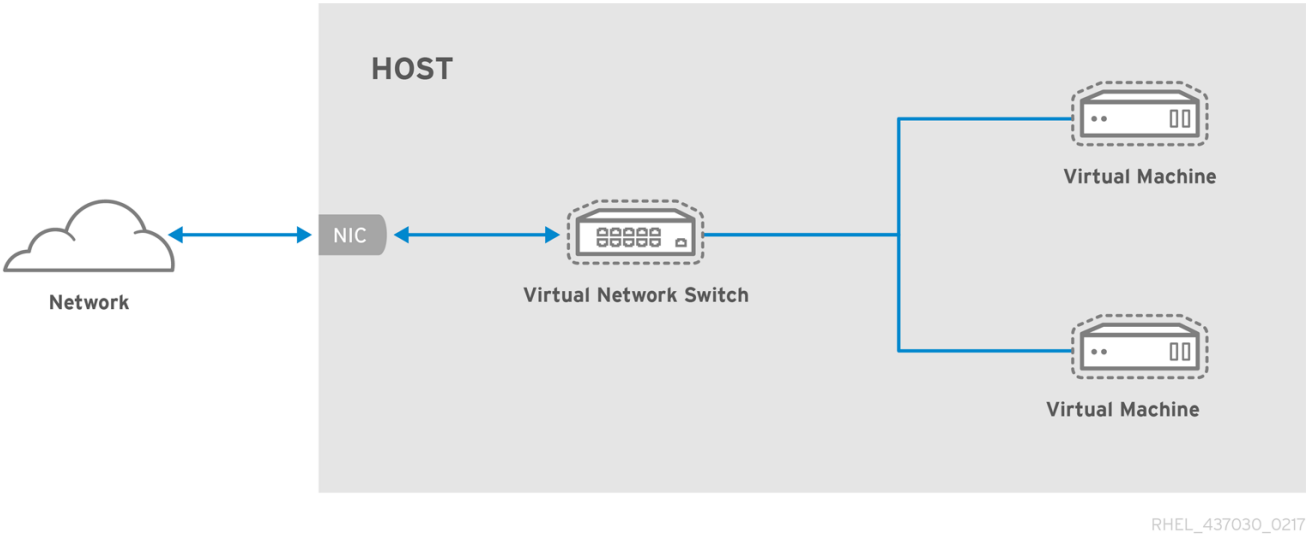


Figure 17.1. Virtual network switch with two guests

Linux host physical machine servers represent a virtual network switch as a network interface. When the libvirtd daemon (**libvirtd**) is first installed and started, the default network interface representing the virtual network switch is **virbr0**.

This **virbr0** interface can be viewed with the **ip** command like any other interface:

```
 $ ip addr show virbr0
 3: virbr0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UNKNOWN
     link/ether 1b:c4:94:cf:fd:17 brd ff:ff:ff:ff:ff:ff
     inet 192.168.122.1/24 brd 192.168.122.255 scope global virbr0
```

17.2. BRIDGED MODE

When using *Bridged mode*, all of the guest virtual machines appear within the same subnet as the host physical machine. All other physical machines on the same physical network are aware of the virtual machines, and can access the virtual machines. Bridging operates on Layer 2 of the OSI networking model.

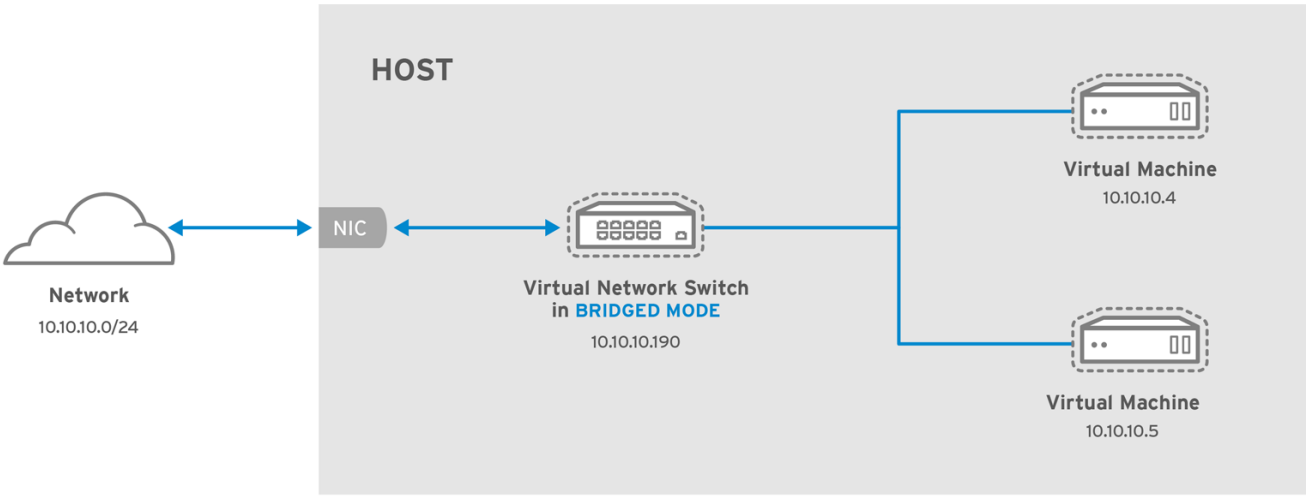


Figure 17.2. Virtual network switch in bridged mode

It is possible to use multiple physical interfaces on the hypervisor by joining them together with a *bond*. The bond is then added to a bridge and then guest virtual machines are added onto the bridge as well. However, the bonding driver has several modes of operation, and only a few of these modes work with a bridge where virtual guest machines are in use.

WARNING

When using bridged mode, the only bonding modes that should be used with a guest virtual machine are Mode 1, Mode 2, and Mode 4. Using modes 0, 3, 5, or 6 is likely to cause the connection to fail. Also note that Media-Independent Interface (MII) monitoring should be used to monitor bonding modes, as Address Resolution Protocol (ARP) monitoring does not work.

For more information on bonding modes, see related Knowledgebase article, or the Red Hat Enterprise Linux 7 Networking Guide.

For a detailed explanation of bridge_opts parameters, used to configure bridged networking mode, see the Red Hat Virtualization Administration Guide.

17.3. NETWORK ADDRESS TRANSLATION

By default, virtual network switches operate in NAT mode. They use IP masquerading rather than Source-NAT (SNAT) or Destination-NAT (DNAT). IP masquerading enables connected guests to use the host physical machine IP address for communication to any external network. By default, computers that are placed externally to the host physical machine cannot communicate to the guests inside when the virtual network switch is operating in NAT mode, as shown in the following diagram:

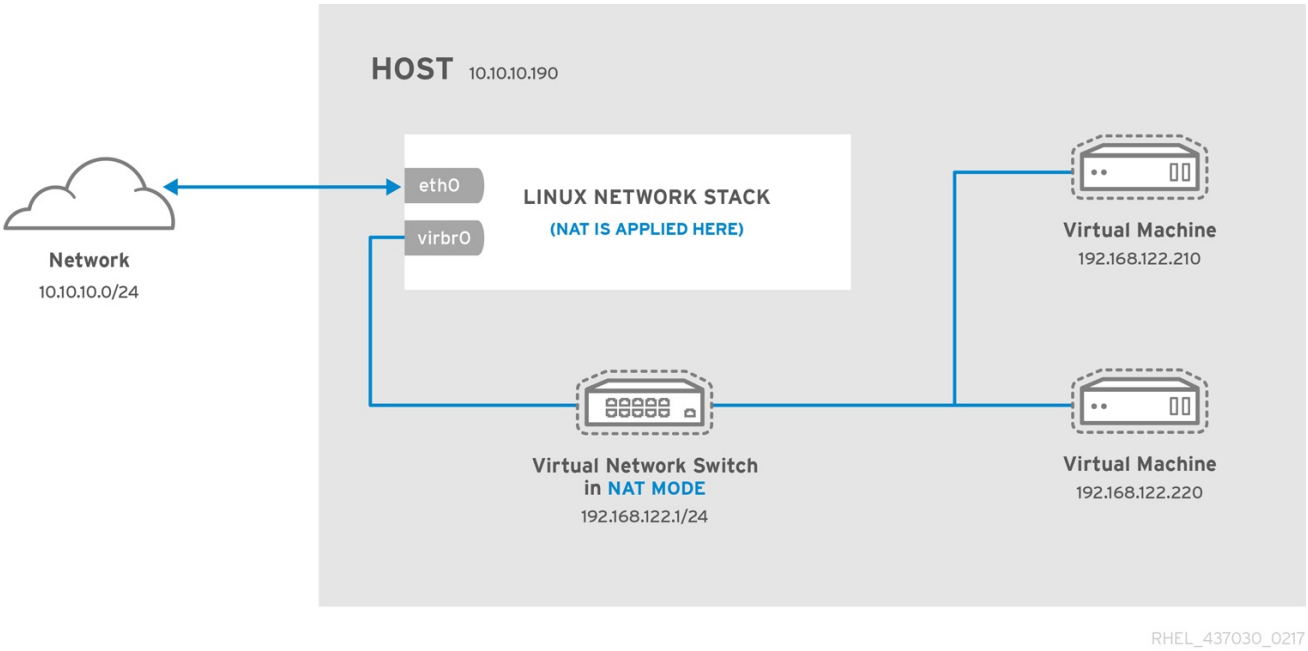


Figure 17.3. Virtual network switch using NAT with two guests

WARNING

Virtual network switches use NAT configured by iptables rules. Editing these rules while the switch is running is not recommended, as incorrect rules may result in the switch being unable to communicate.

If the switch is not running, you can set the public IP range for forward mode NAT in order to create a port masquerading range by running:

```
# iptables -j SNAT --to-source [start]-[end]
```

17.4. DNS AND DHCP

IP information can be assigned to guests via DHCP. A pool of addresses can be assigned to a virtual network switch for this purpose. Libvirt uses the **dnsmasq** program for this. An instance of dnsmasq is automatically configured and started by libvirt for each virtual network switch that needs it.

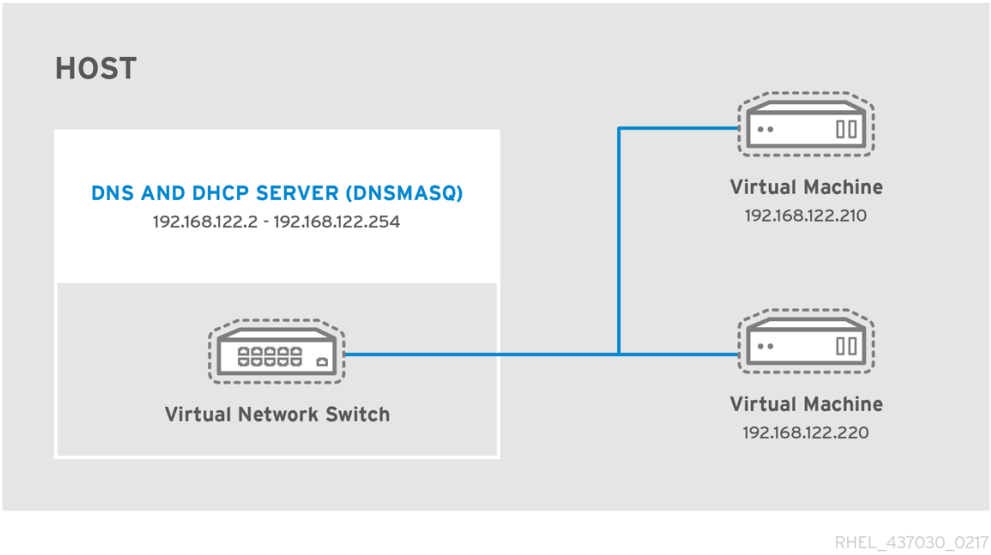


Figure 17.4. Virtual network switch running dnsmasq

17.5. ROUTED MODE

When using *Routed mode*, the virtual switch connects to the physical LAN connected to the host physical machine, passing traffic back and forth without the use of NAT. The virtual switch can examine all traffic and use the information contained within the network packets to make routing decisions. When using this mode, all of the virtual machines are in their own subnet, routed through a virtual switch. This situation is not always ideal as no other host physical machines on the physical network are aware of the virtual machines without manual physical router configuration, and cannot access the virtual machines. Routed mode operates at Layer 3 of the OSI networking model.

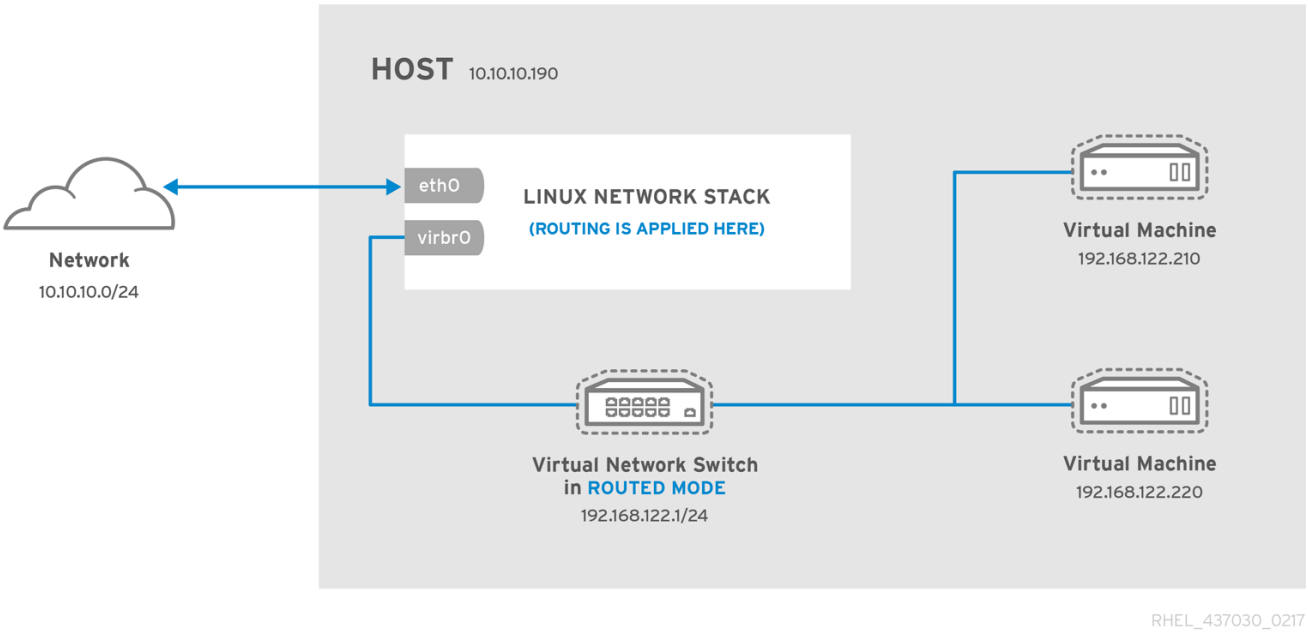


Figure 17.5. Virtual network switch in routed mode

17.6. ISOLATED MODE

When using *Isolated mode*, guests connected to the virtual switch can communicate with each other, and with the host physical machine, but their traffic will not pass outside of the host physical machine, and they cannot receive traffic from outside the host physical machine. Using dnsmasq in this mode is

required for basic functionality such as DHCP. However, even if this network is isolated from any physical network, DNS names are still resolved. Therefore, a situation can arise when DNS names resolve but ICMP echo request (ping) commands fail.

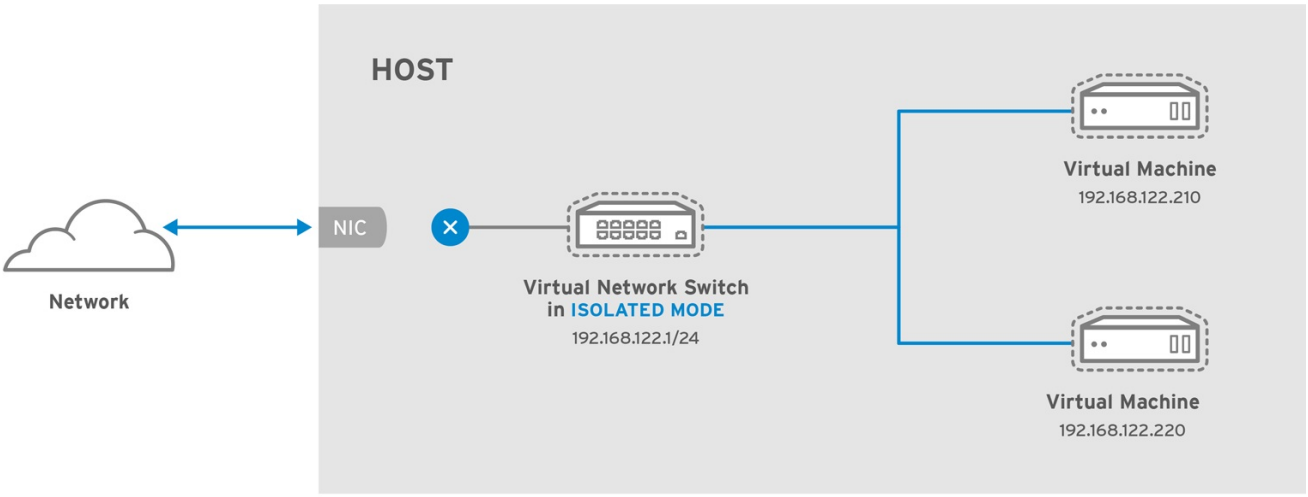


Figure 17.6. Virtual network switch in isolated mode

17.7. THE DEFAULT CONFIGURATION

When the libvirtd daemon (**`libvirtd`**) is first installed, it contains an initial virtual network switch configuration in NAT mode. This configuration is used so that installed guests can communicate to the external network, through the host physical machine. The following image demonstrates this default configuration for **`libvirtd`**:

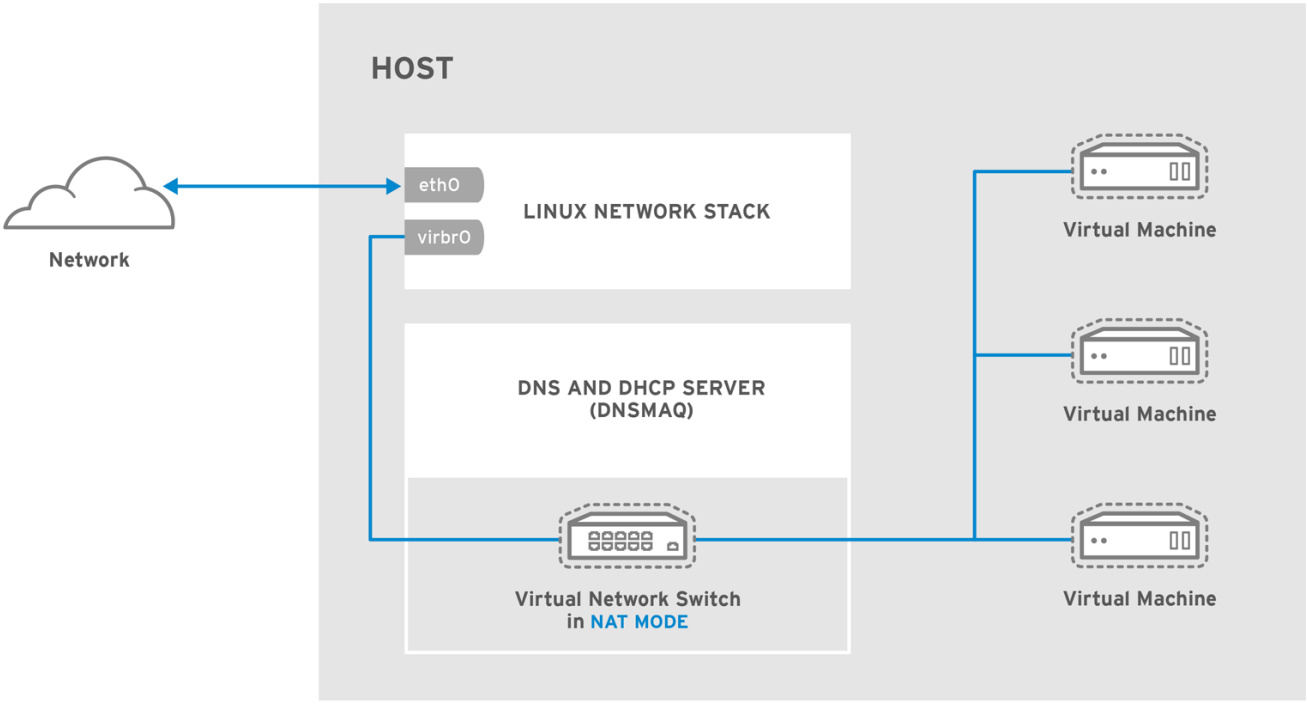
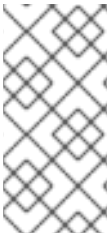


Figure 17.7. Default libvirt network configuration



NOTE

A virtual network can be restricted to a specific physical interface. This may be useful on a physical system that has several interfaces (for example, **eth0**, **eth1** and **eth2**). This is only useful in routed and NAT modes, and can be defined in the **dev=<interface>** option, or in **virt-manager** when creating a new virtual network.

17.8. EXAMPLES OF COMMON SCENARIOS

This section demonstrates different virtual networking modes and provides some example scenarios.

17.8.1. Bridged Mode

Bridged mode operates on Layer 2 of the OSI model. When used, all of the guest virtual machines will appear on the same subnet as the host physical machine. The most common use cases for bridged mode include:

- Deploying guest virtual machines in an existing network alongside host physical machines making the difference between virtual and physical machines transparent to the end user.
- Deploying guest virtual machines without making any changes to existing physical network configuration settings.
- Deploying guest virtual machines which must be easily accessible to an existing physical network. Placing guest virtual machines on a physical network where they must access services within an existing broadcast domain, such as DHCP.
- Connecting guest virtual machines to an exsting network where VLANs are used.

17.8.2. Routed Mode

DMZ

Consider a network where one or more nodes are placed in a controlled sub-network for security reasons. The deployment of a special sub-network such as this is a common practice, and the sub-network is known as a DMZ. See the following diagram for more details on this layout:

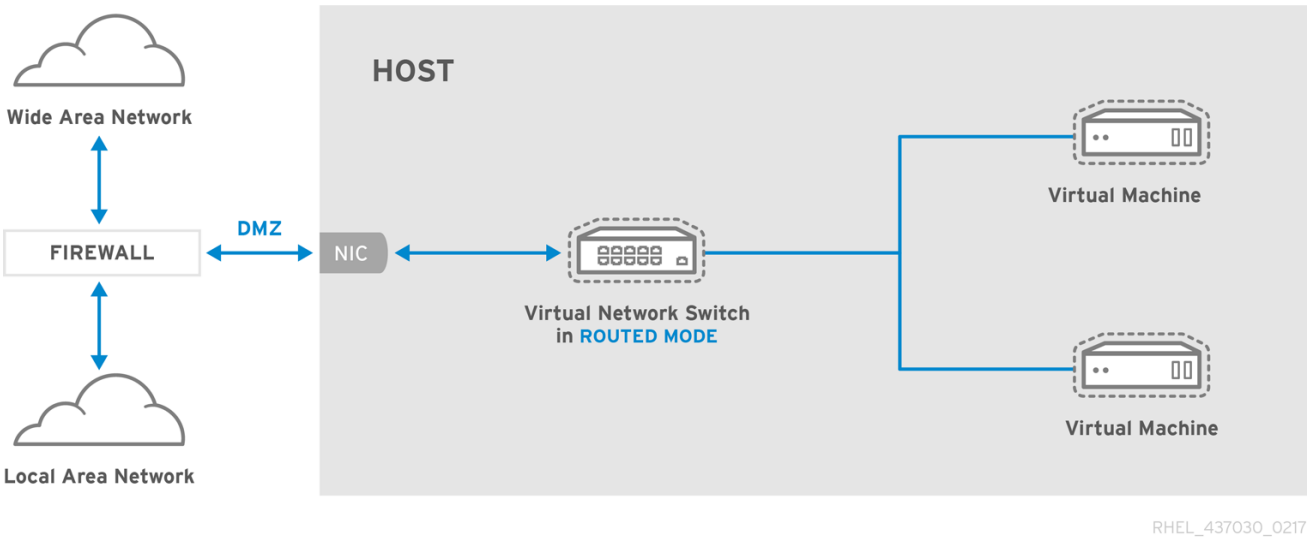


Figure 17.8. Sample DMZ configuration

Host physical machines in a DMZ typically provide services to WAN (external) host physical machines as well as LAN (internal) host physical machines. As this requires them to be accessible from multiple locations, and considering that these locations are controlled and operated in different ways based on their security and trust level, routed mode is the best configuration for this environment.

Virtual Server Hosting

Consider a virtual server hosting company that has several host physical machines, each with two physical network connections. One interface is used for management and accounting, the other is for the virtual machines to connect through. Each guest has its own public IP address, but the host physical machines use private IP address as management of the guests can only be performed by internal administrators. See the following diagram to understand this scenario:

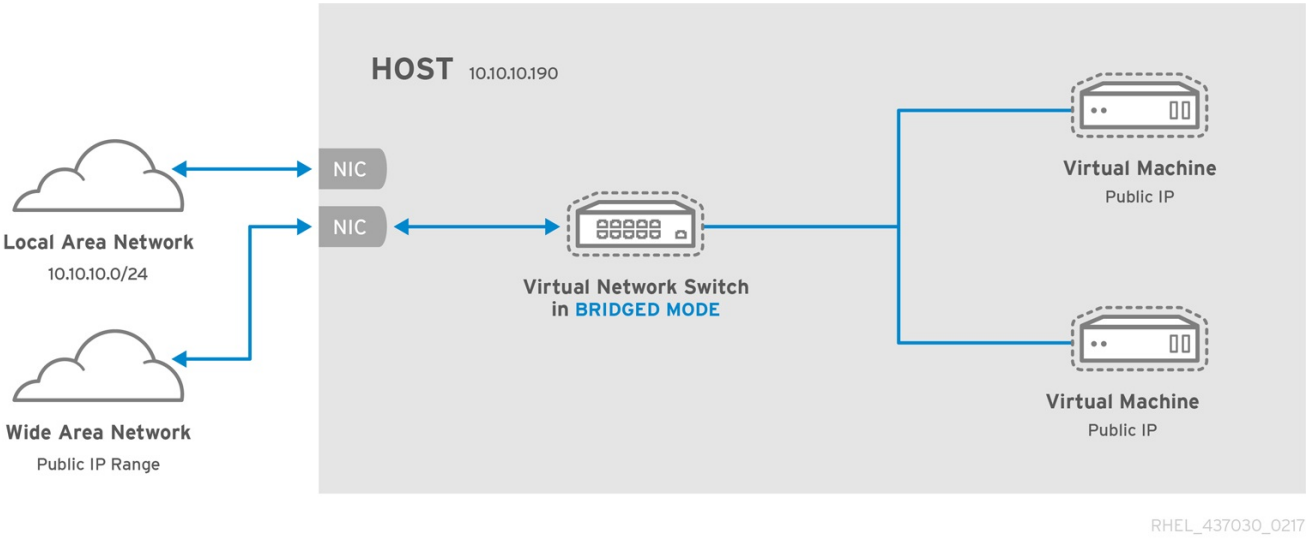


Figure 17.9. Virtual server hosting sample configuration

17.8.3. NAT Mode

NAT (Network Address Translation) mode is the default mode. It can be used for testing when there is no need for direct network visibility.

17.8.4. Isolated Mode

Isolated mode allows virtual machines to communicate with each other only. They are unable to interact with the physical network.

17.9. MANAGING A VIRTUAL NETWORK

To configure a virtual network on your system:

1. From the **Edit** menu, select **Connection Details**.

2. This will open the **Connection Details** menu. Click the **Virtual Networks** tab.

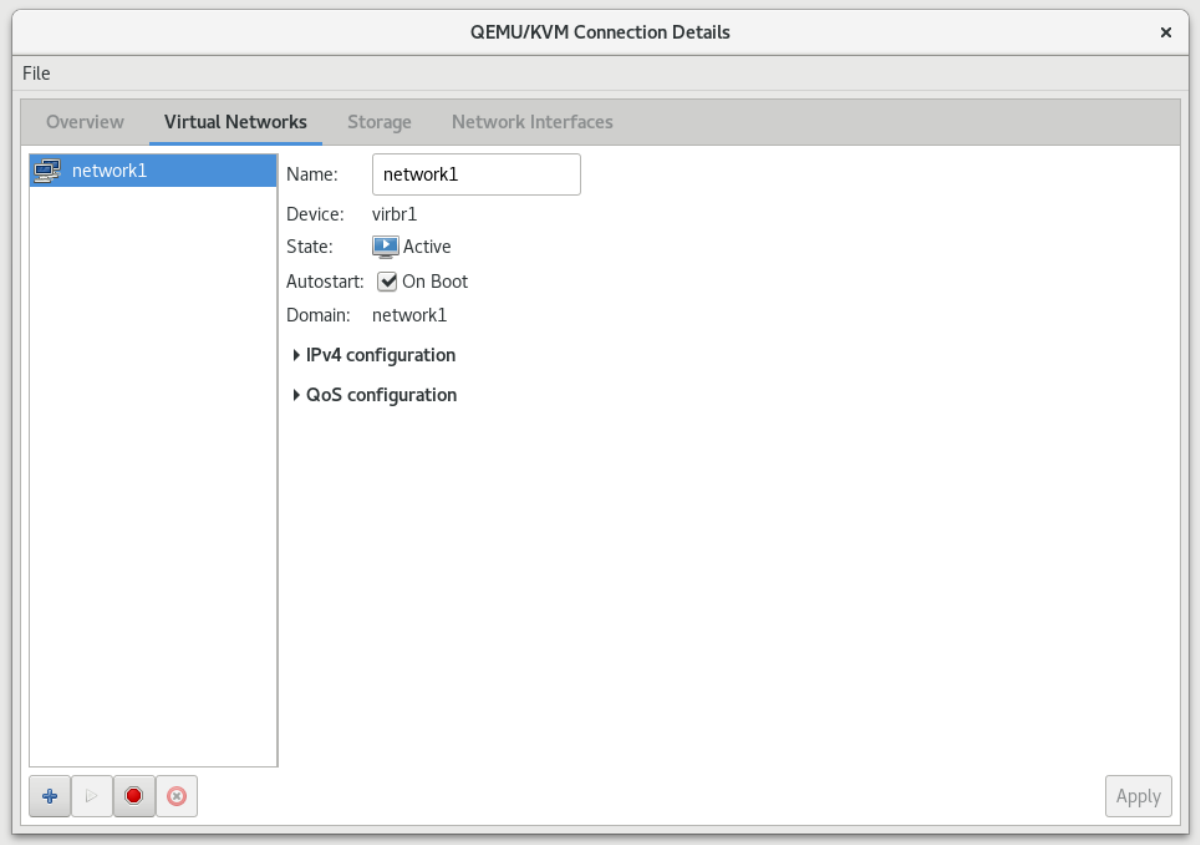


Figure 17.10. Virtual network configuration

3. All available virtual networks are listed on the left of the menu. You can edit the configuration of a virtual network by selecting it from this box and editing as you see fit.

17.10. CREATING A VIRTUAL NETWORK

To create a virtual network on your system using the Virtual Machine Manager (virt-manager):

1. Open the **Virtual Networks** tab from within the **Connection Details** menu. Click the **Add Network** button, identified by a plus sign (+) icon. For more information, see Section 17.9, “Managing a Virtual Network”.

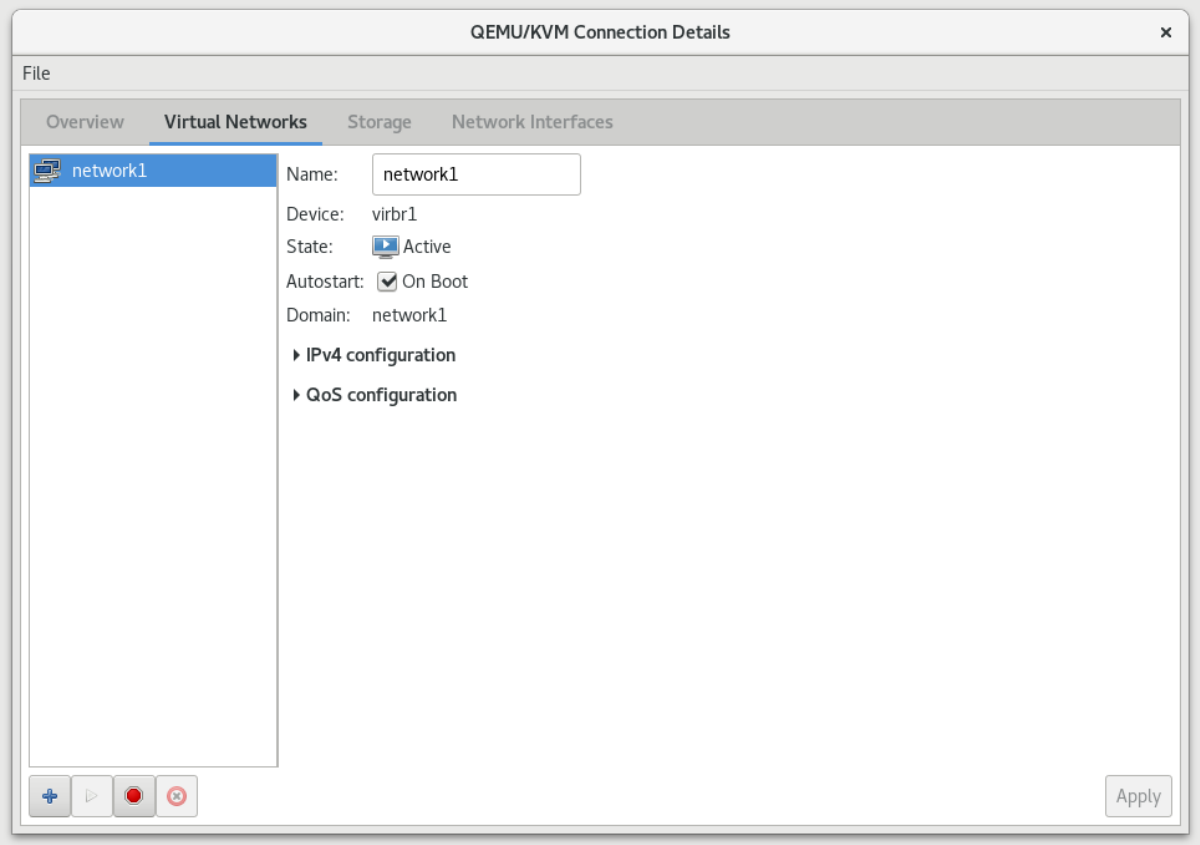


Figure 17.11. Virtual network configuration

This will open the **Create a new virtual network** window. Click **`Forward`** to continue.

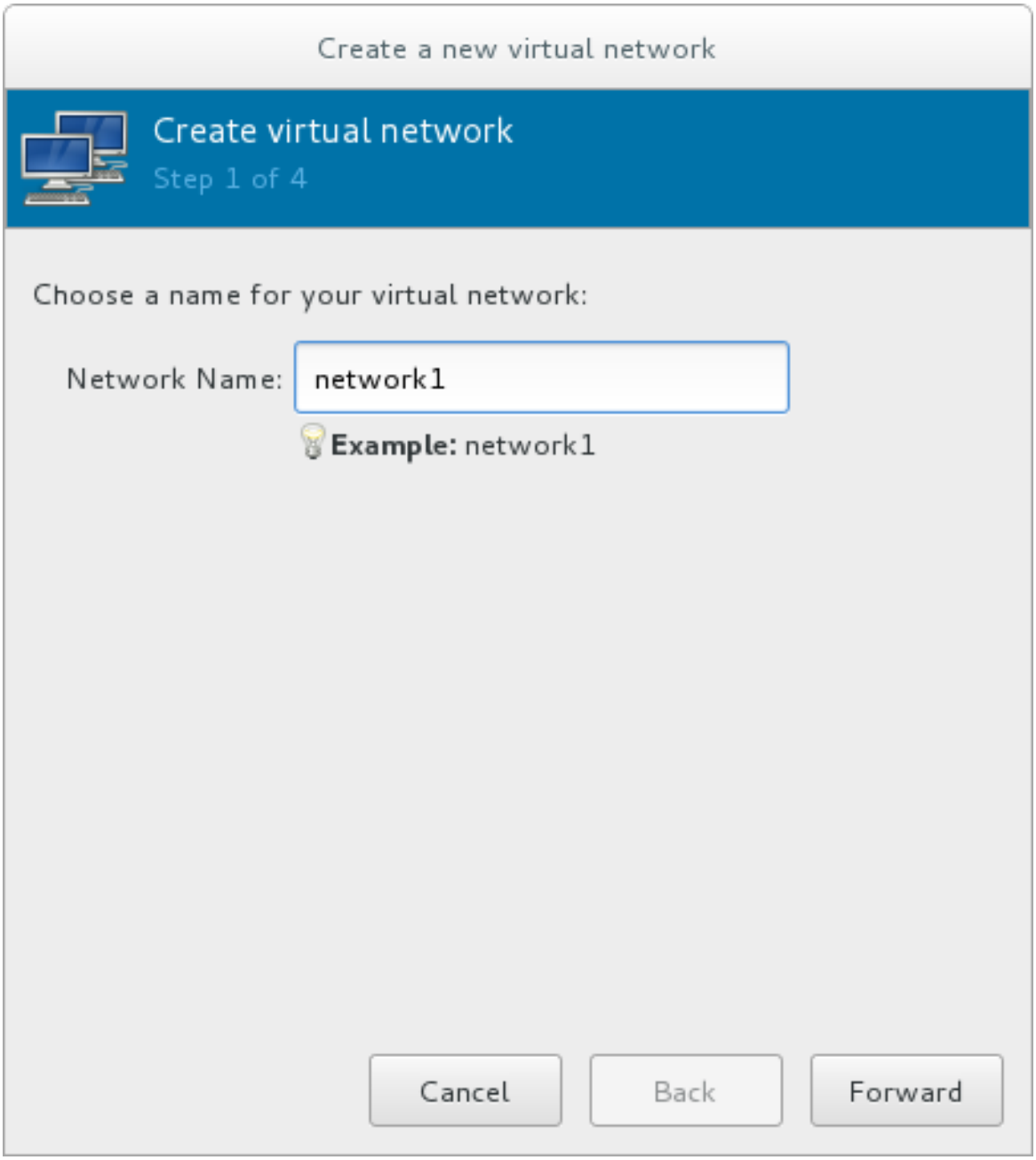


Figure 17.12. Naming your new virtual network

2. Enter an appropriate name for your virtual network and click **Forward**.

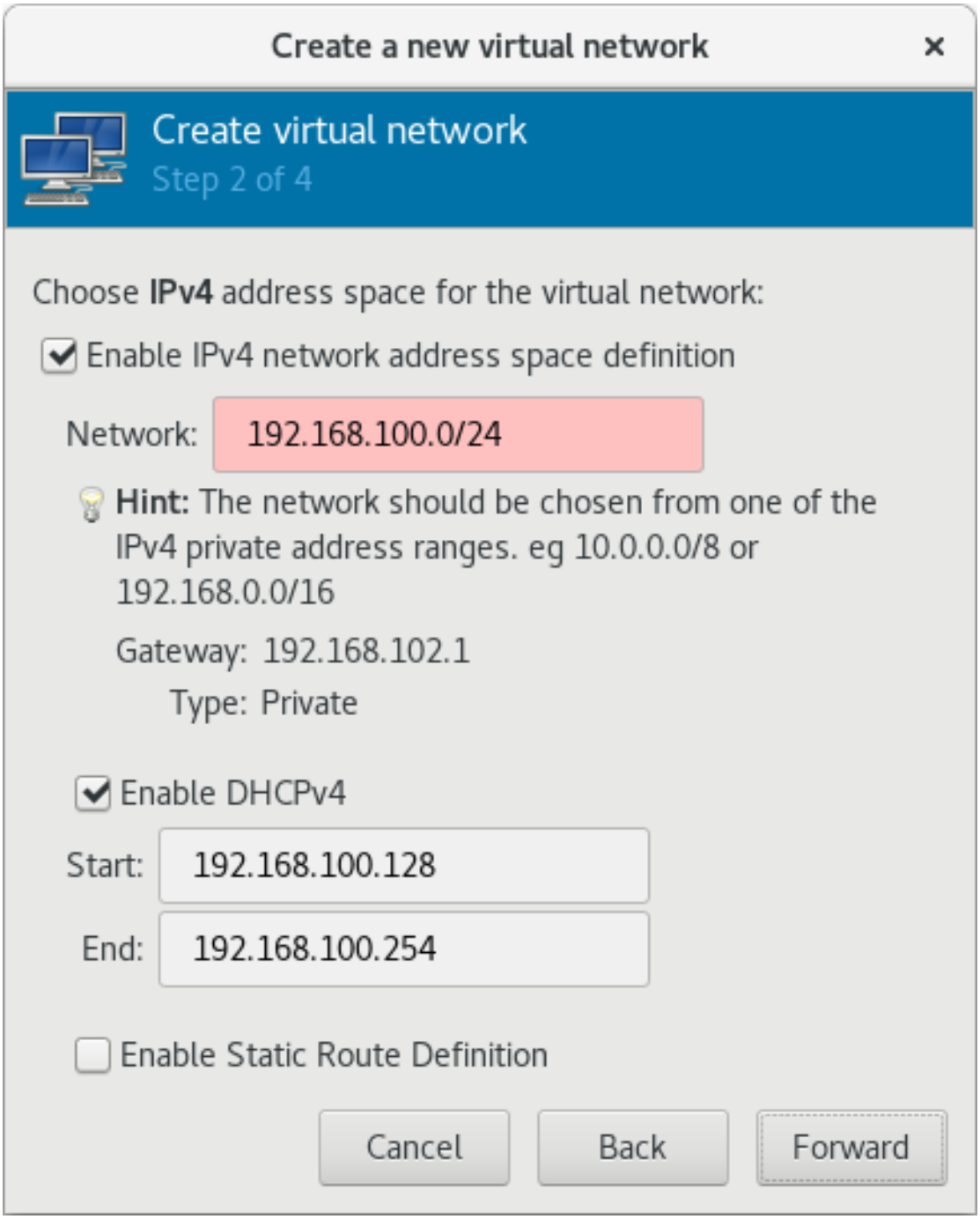


Figure 17.13. Choosing an IPv4 address space

3. Check the **Enable IPv4 network address space definition** check box.

   Enter an IPv4 address space for your virtual network in the **Network** field.

   Check the **Enable DHCPv4** check box.

   Define the DHCP range for your virtual network by specifying a **Start** and **End** range of IP addresses.

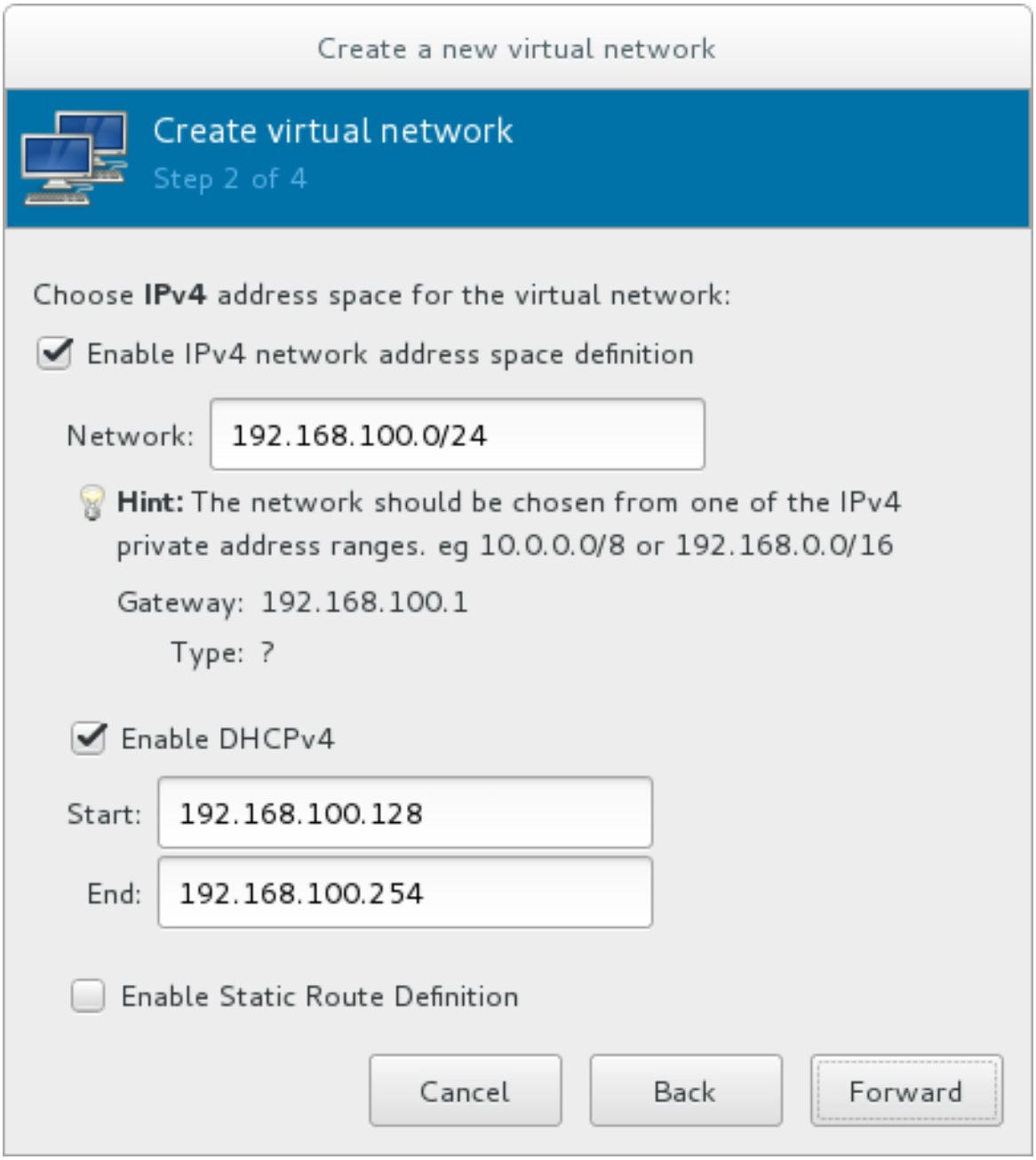


Figure 17.14. Choosing an IPv4 address space

Click **Forward** to continue.

4. If you want to enable IPv6, check the **Enable IPv6 network address space definition**.

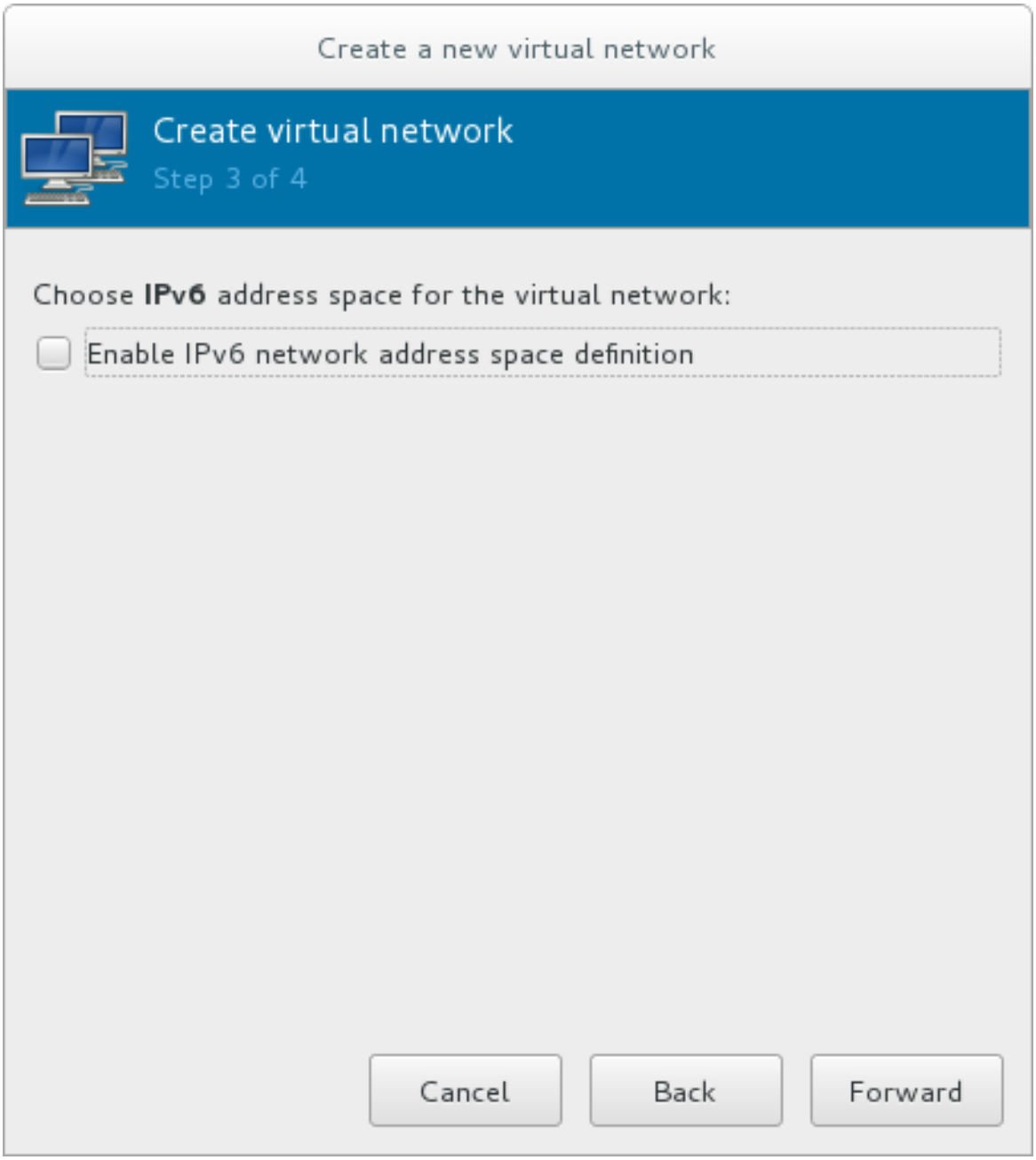


Figure 17.15. Enabling IPv6

Additional fields appear in the Create a new virtual network window.

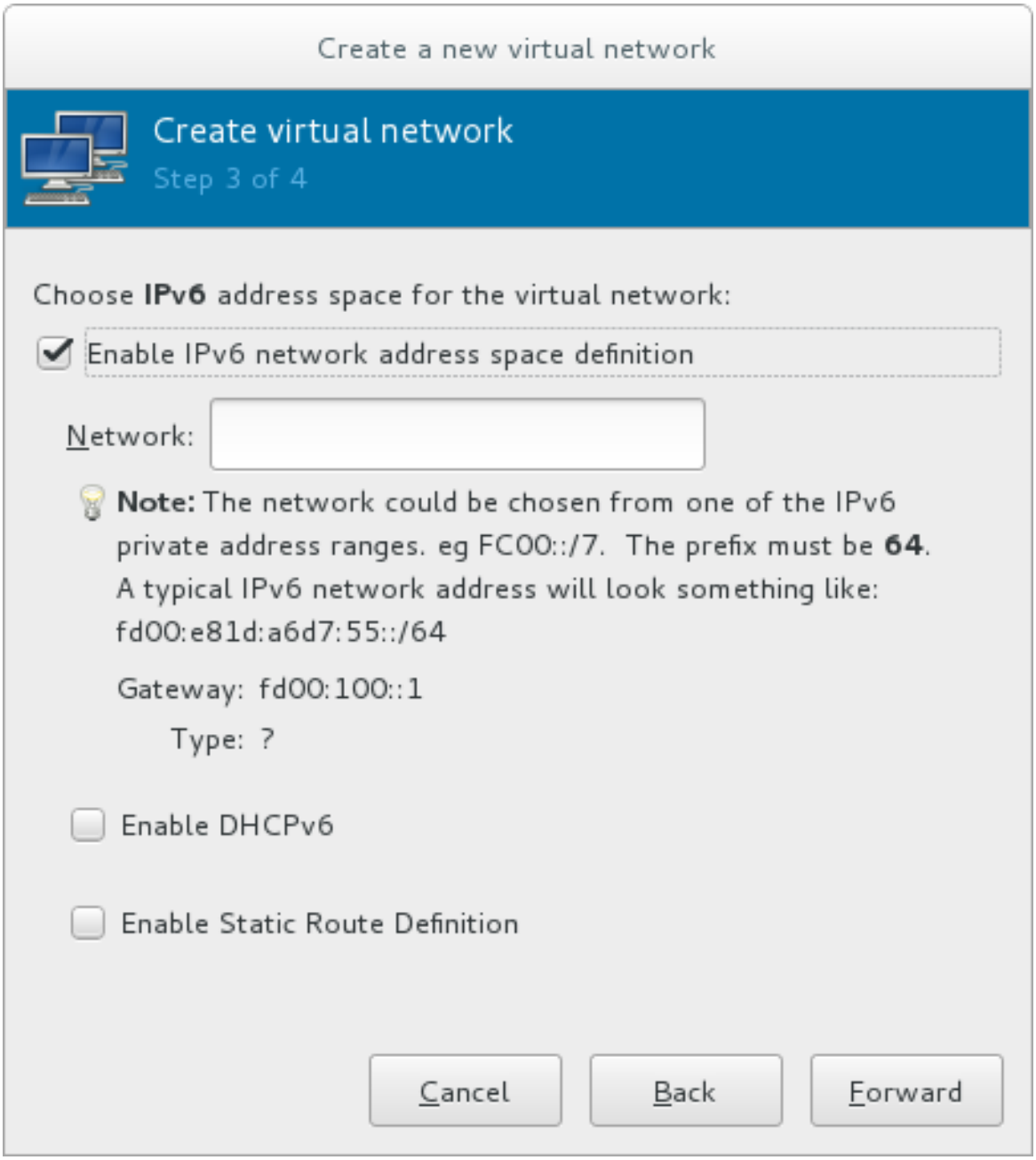


Figure 17.16. Configuring IPv6

Enter an IPv6 address in the **Network** field.

5. If you want to enable DHCPv6, check the **Enable DHCPv6** check box.

   Additional fields appear in the Create a new virtual network window.

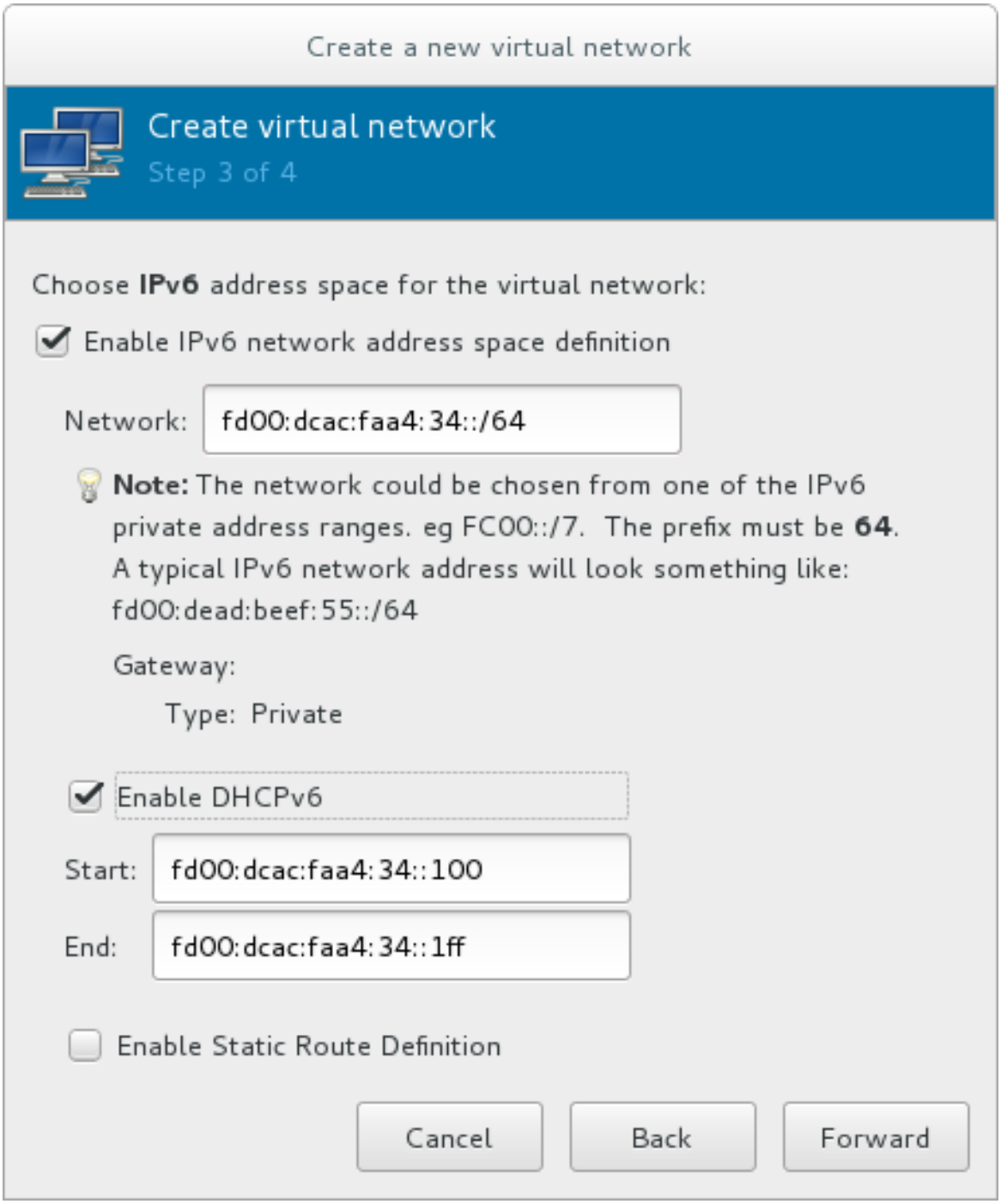


Figure 17.17. Configuring DHCPv6

(Optional) Edit the start and end of the DHCPv6 range.

6. If you want to enable static route definitions, check the **`Enable Static Route Definition`** check box.

   Additional fields appear in the Create a new virtual network window.

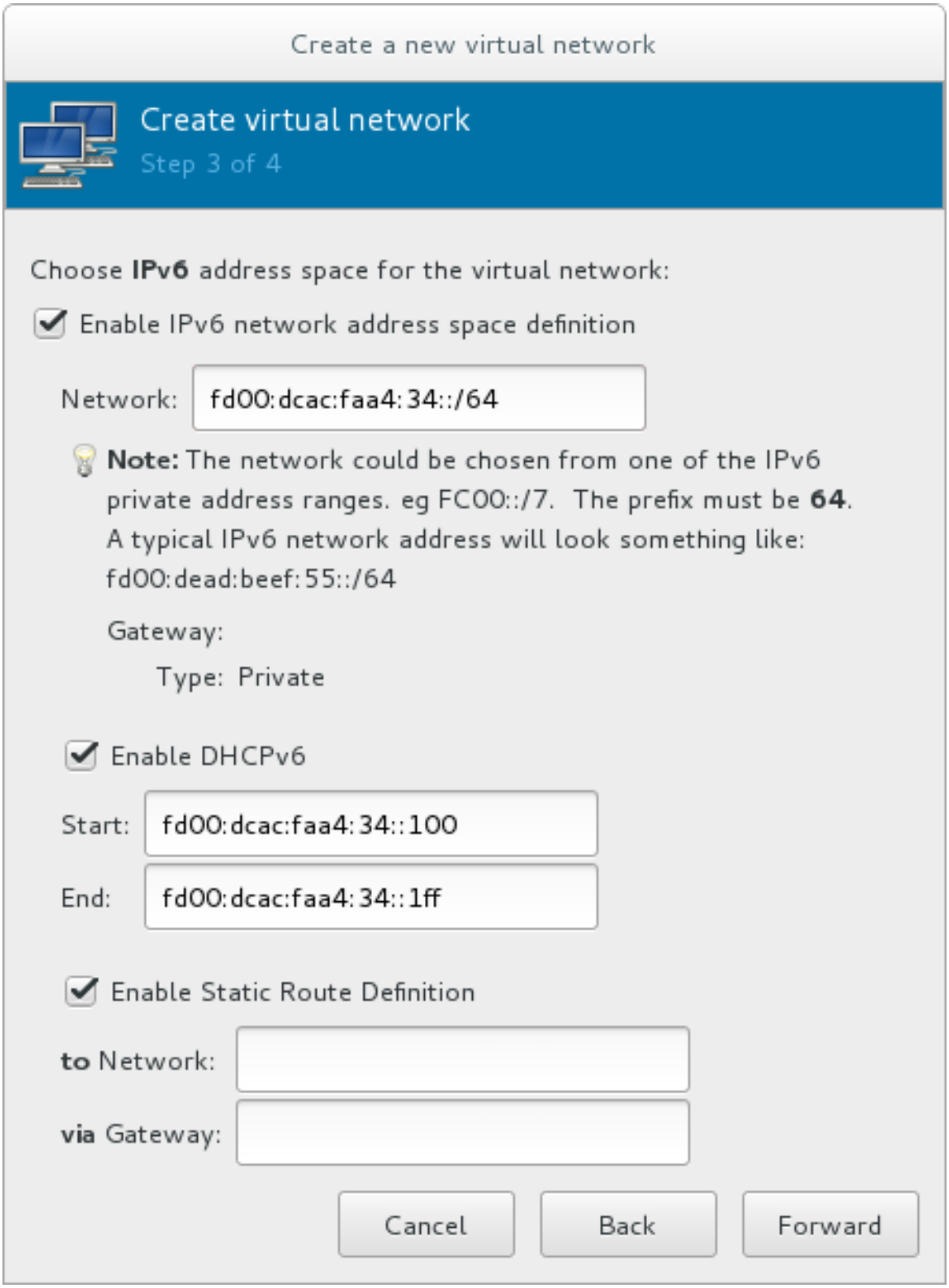


Figure 17.18. Defining static routes

Enter a network address and the gateway that will be used for the route to the network in the appropriate fields.

Click **Forward**.

7. Select how the virtual network should connect to the physical network.

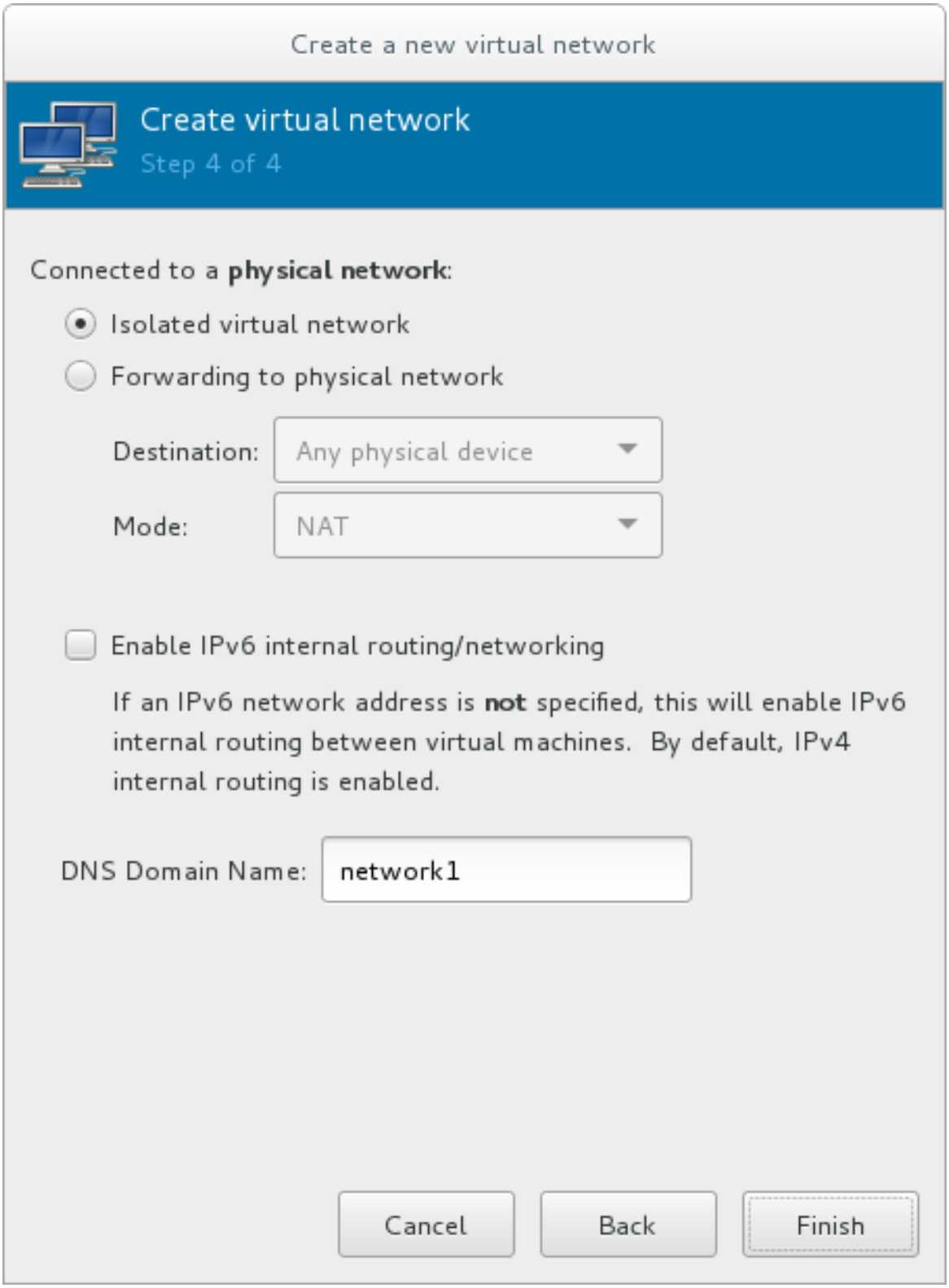


Figure 17.19. Connecting to the physical network

If you want the virtual network to be isolated, ensure that the **`Isolated virtual network`** radio button is selected.

If you want the virtual network to connect to a physical network, select **`Forwarding to physical network`**, and choose whether the **`Destination`** should be **`Any physical device`** or a specific physical device. Also select whether the **`Mode`** should be **`NAT`** or **`Routed`**.

If you want to enable IPv6 routing within the virtual network, check the **`Enable IPv6 internal routing/networking`** check box.

Enter a DNS domain name for the virtual network.

Click **Finish** to create the virtual network.

8. The new virtual network is now available in the **Virtual Networks** tab of the **Connection Details** window.

17.11. ATTACHING A VIRTUAL NETWORK TO A GUEST

To attach a virtual network to a guest:

1. In the **Virtual Machine Manager** window, highlight the guest that will have the network assigned.

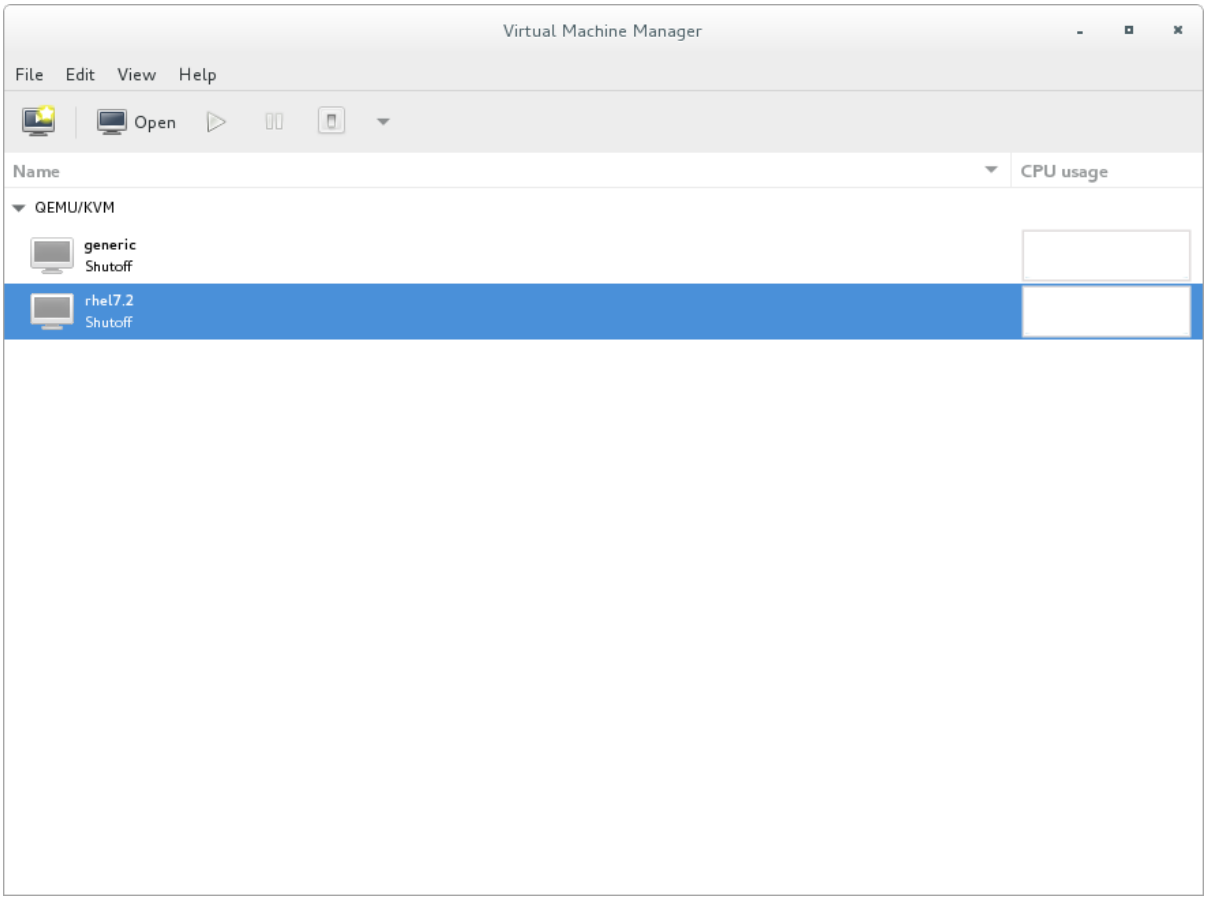


Figure 17.20. Selecting a virtual machine to display

2. From the Virtual Machine Manager **Edit** menu, select **Virtual Machine Details**.

3. Click the **Add Hardware** button on the Virtual Machine Details window.

4. In the **Add new virtual hardware** window, select **Network** from the left pane, and select your network name (*network1* in this example) from the **Network source** menu. Modify the MAC address, if necessary, and select a **Device model**. Click **Finish**.

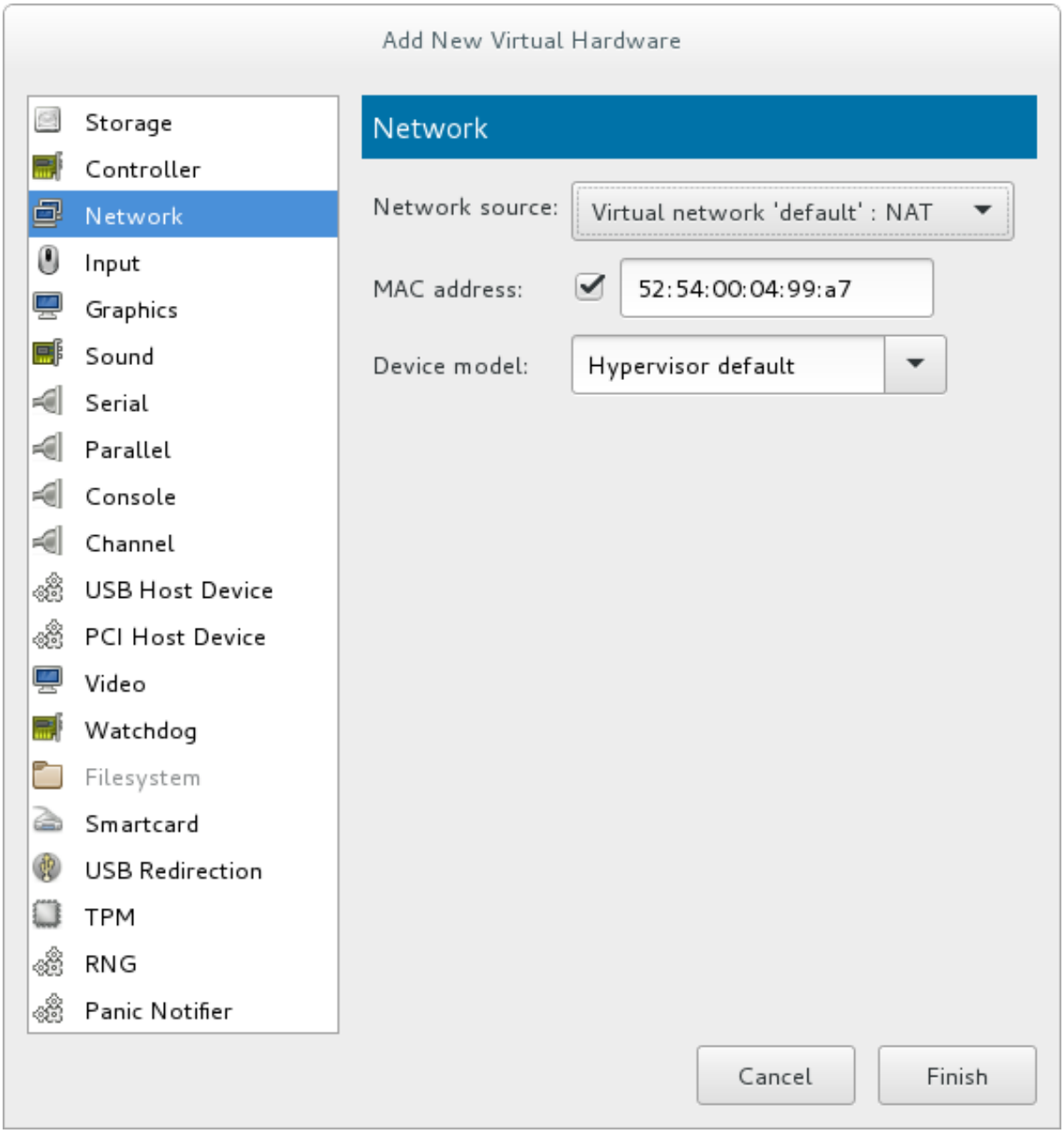


Figure 17.21. Select your network from the Add new virtual hardware window

5. The new network is now displayed as a virtual network interface that will be presented to the guest upon launch.

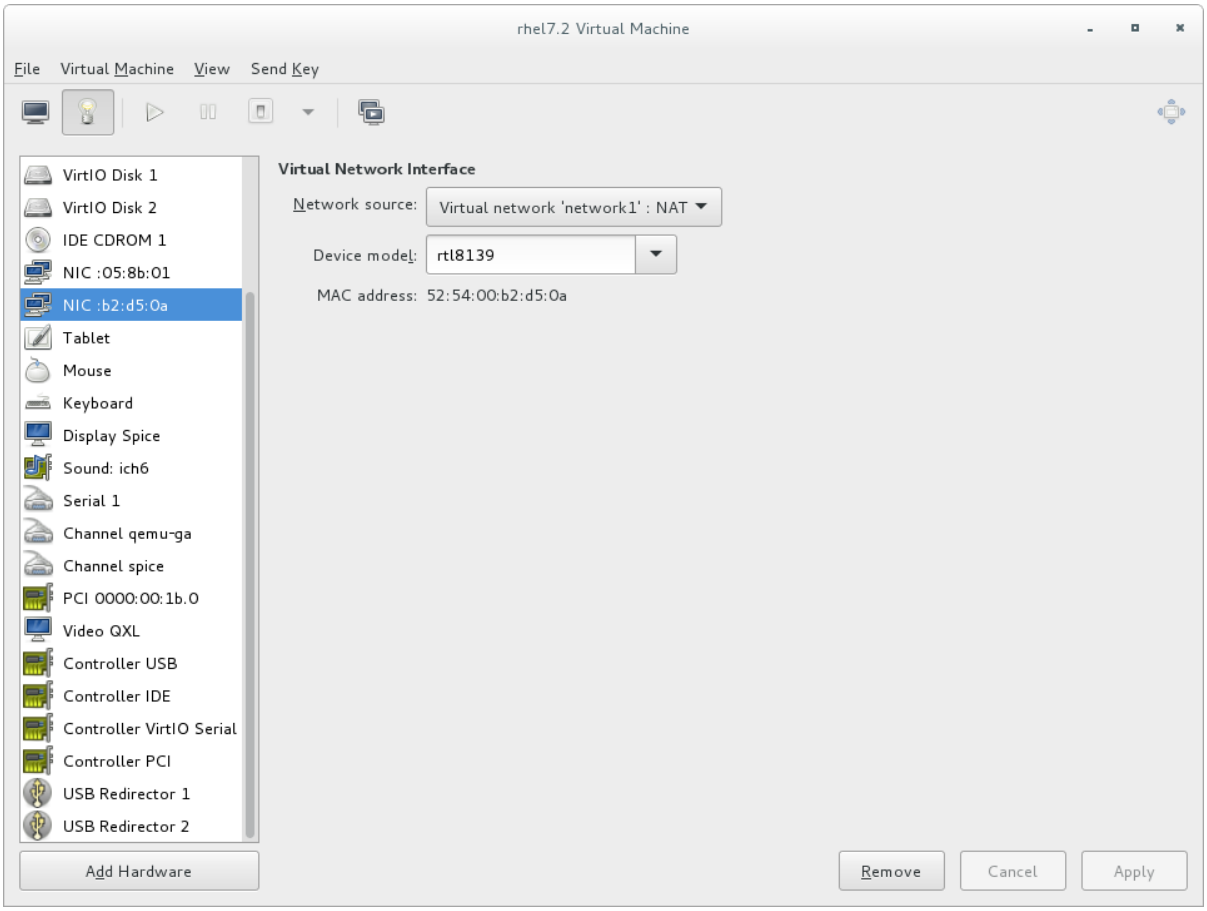


Figure 17.22. New network shown in guest hardware list

17.12. ATTACHING A VIRTUAL NIC DIRECTLY TO A PHYSICAL INTERFACE

As an alternative to the default NAT connection, you can use the *macvtap* driver to attach the guest's NIC directly to a specified physical interface of the host machine. This is not to be confused with device assignment (also known as passthrough). Macvtap connection has the following modes, each with different benefits and usecases:

Physical interface delivery modes

VEPA

In virtual ethernet port aggregator (VEPA) mode, all packets from the guests are sent to the external switch. This enables the user to force guest traffic through the switch. For VEPA mode to work correctly, the external switch must also support *hairpin mode*, which ensures that packets whose destination is a guest on the same host machine as their source guest are sent back to the host by the external switch.

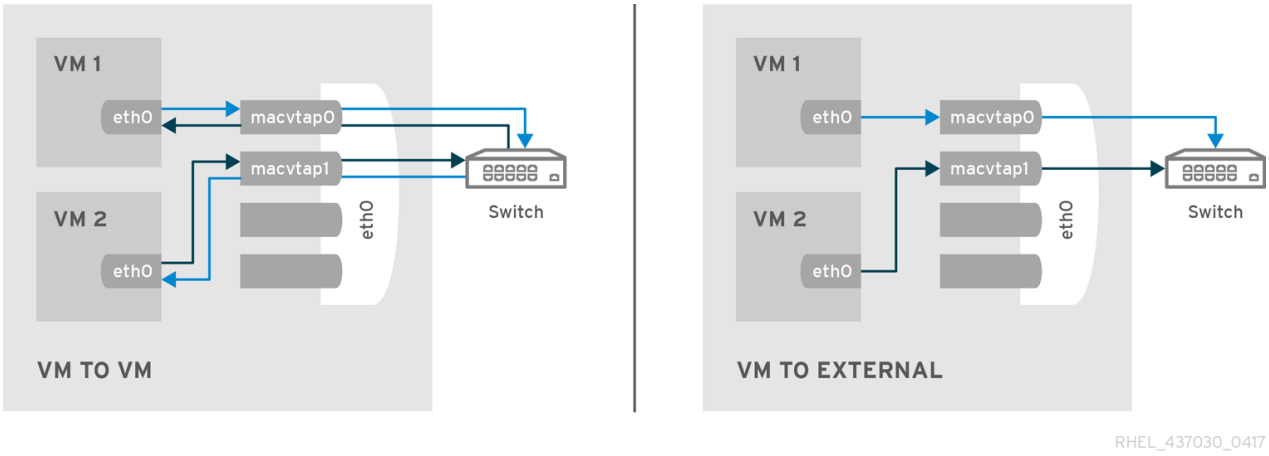


Figure 17.23. VEPA mode

bridge

Packets whose destination is on the same host machine as their source guest are directly delivered to the target macvtap device. Both the source device and the destination device need to be in bridge mode for direct delivery to succeed. If either one of the devices is in VEPA mode, a hairpin-capable external switch is required.

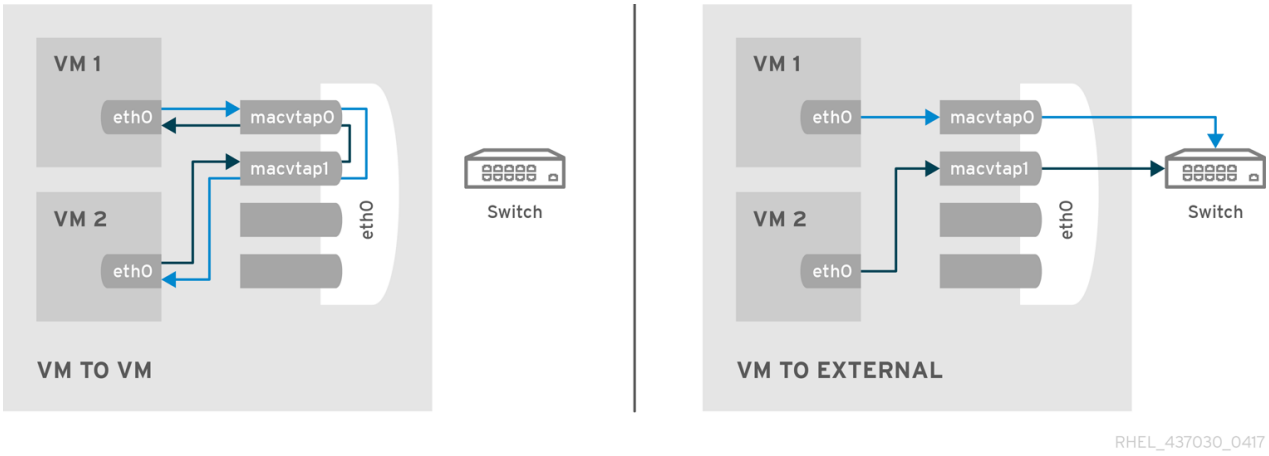


Figure 17.24. Bridge mode

private

All packets are sent to the external switch and will only be delivered to a target guest on the same host machine if they are sent through an external router or gateway and these send them back to the host. Private mode can be used to prevent the individual guests on the single host from communicating with each other. This procedure is followed if either the source or destination device is in private mode.

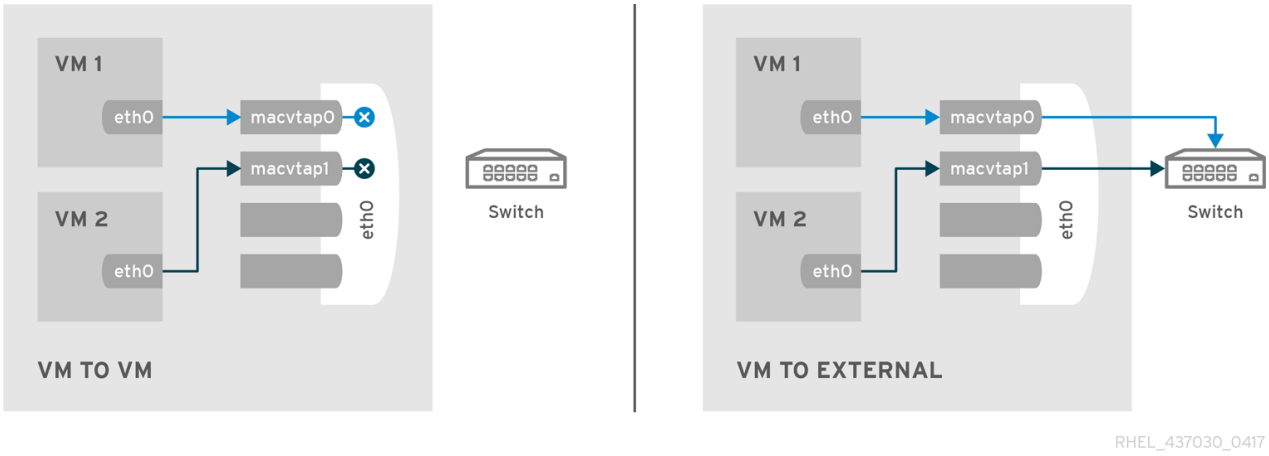


Figure 17.25. Private mode

passthrough

This feature attaches a physical interface device or a SR-IOV Virtual Function (VF) directly to a guest without losing the migration capability. All packets are sent directly to the designated network device. Note that a single network device can only be passed through to a single guest, as a network device cannot be shared between guests in passthrough mode.

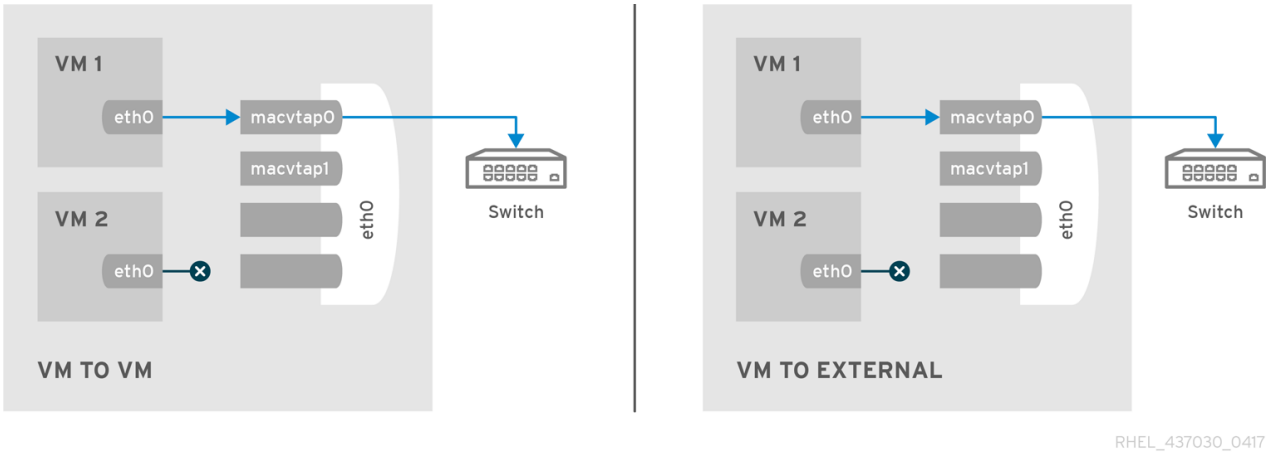


Figure 17.26. Passthrough mode

Macvtap can be configured by changing the domain XML file or by using the **virt-manager** interface.

17.12.1. Configuring macvtap using domain XML

Open the domain XML file of the guest and modify the **<devices>** element as follows:

```
<devices>
 ...
 <interface type='direct'>
  <source dev='eth0' mode='vepa'/>
 </interface>
</devices>
```

The network access of direct attached guest virtual machines can be managed by the hardware switch to which the physical interface of the host physical machine is connected.

The interface can have additional parameters as shown below, if the switch is conforming to the IEEE 802.1Qbg standard. The parameters of the virtualport element are documented in more detail in the IEEE 802.1Qbg standard. The values are network specific and should be provided by the network administrator. In 802.1Qbg terms, the Virtual Station Interface (VSI) represents the virtual interface of a virtual machine. Also note that IEEE 802.1Qbg requires a non-zero value for the VLAN ID.

Virtual Station Interface types

managerid

The VSI Manager ID identifies the database containing the VSI type and instance definitions. This is an integer value and the value 0 is reserved.

typeid

The VSI Type ID identifies a VSI type characterizing the network access. VSI types are typically managed by network administrator. This is an integer value.

typeidversion

The VSI Type Version allows multiple versions of a VSI Type. This is an integer value.

instanceid

The VSI Instance ID is generated when a VSI instance (a virtual interface of a virtual machine) is created. This is a globally unique identifier.

profileid

The profile ID contains the name of the port profile that is to be applied onto this interface. This name is resolved by the port profile database into the network parameters from the port profile, and those network parameters will be applied to this interface.

Each of the four types is configured by changing the domain XML file. Once this file is opened, change the mode setting as shown:

```
<devices>
 ...
 <interface type='direct'>
  <source dev='eth0.2' mode='vepa'/>
   <virtualport type="802.1Qbg">
    <parameters managerid="11" typeid="1193047" typeidversion="2"
instanceid="09b11c53-8b5c-4eeb-8f00-d84eaa0aaa4f"/>
   </virtualport>
  </interface>
</devices>
```

The profile ID is shown here:

```
<devices>
 ...
 <interface type='direct'>
  <source dev='eth0' mode='private'/>
   <virtualport type='802.1Qbh'>
    <parameters profileid='finance'/>
   </virtualport>
 </interface>
```

```
</devices>
...
```

17.12.2. Configuring macvtap using virt-manager

Open the virtual hardware details window ⇒ select **NIC** in the menu ⇒ for **Network source**, select **host device *name*: macvtap** ⇒ select the intended **Source mode**.

The virtual station interface types can then be set up in the **Virtual port** submenu.

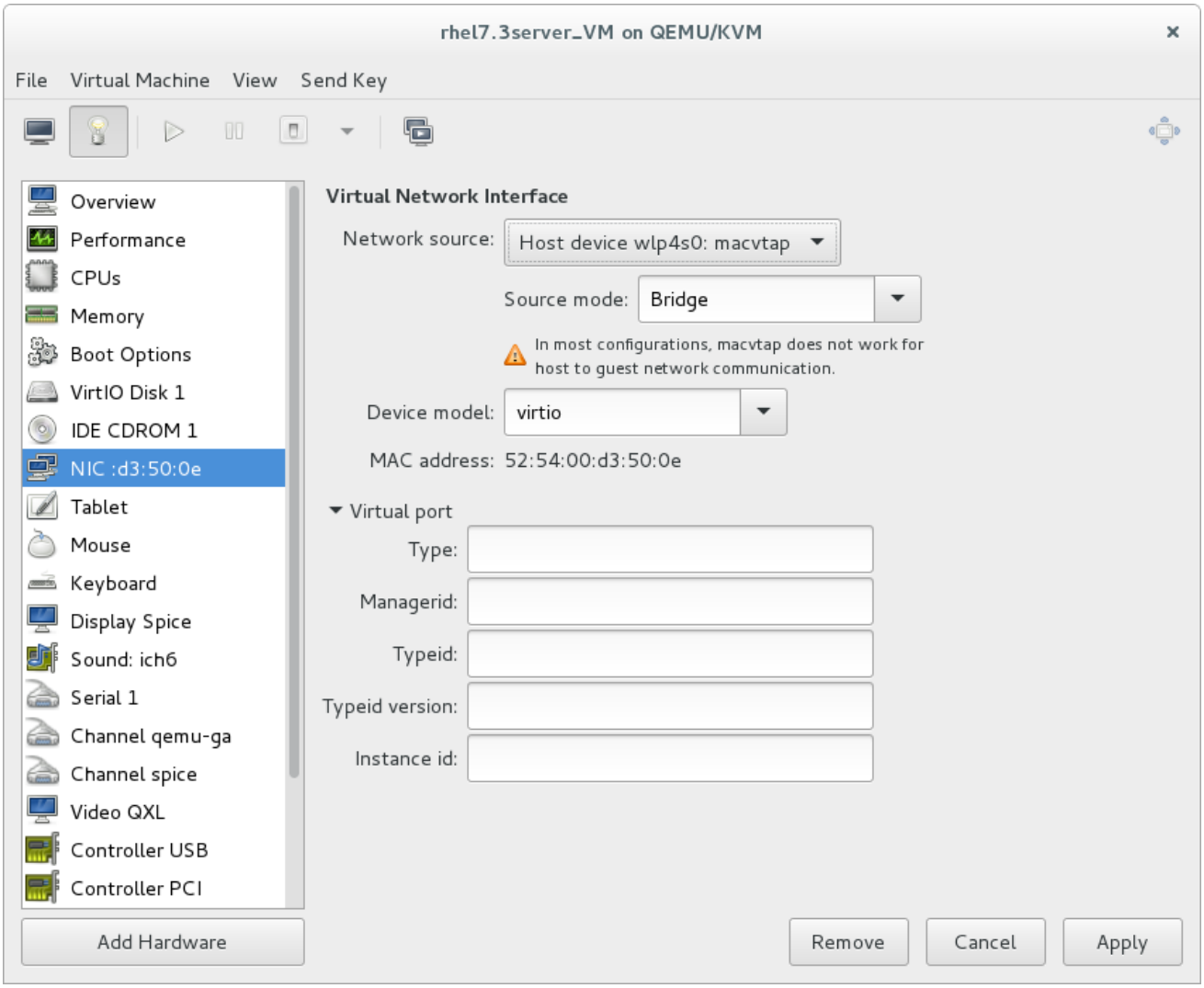


Figure 17.27. Configuring macvtap in virt-manager

17.13. DYNAMICALLY CHANGING A HOST PHYSICAL MACHINE OR A NETWORK BRIDGE THAT IS ATTACHED TO A VIRTUAL NIC

This section demonstrates how to move the vNIC of a guest virtual machine from one bridge to another while the guest virtual machine is running without compromising the guest virtual machine

1. Prepare guest virtual machine with a configuration similar to the following:

```
<interface type='bridge'>
      <mac address='52:54:00:4a:c9:5e'/>
      <source bridge='virbr0'/>
```

```
      <model type='virtio'/>
</interface>
```

2. Prepare an XML file for interface update:

```
# cat br1.xml
```

```
<interface type='bridge'>
      <mac address='52:54:00:4a:c9:5e'/>
      <source bridge='virbr1'/>
      <model type='virtio'/>
</interface>
```

3. Start the guest virtual machine, confirm the guest virtual machine's network functionality, and check that the guest virtual machine's vnetX is connected to the bridge you indicated.

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
virbr0          8000.5254007da9f2       yes
virbr0-nic

vnet0
virbr1          8000.525400682996       yes
virbr1-nic
```

4. Update the guest virtual machine's network with the new interface parameters with the following command:

```
# virsh update-device test1 br1.xml

Device updated successfully
```

5. On the guest virtual machine, run **`service network restart`**. The guest virtual machine gets a new IP address for virbr1. Check the guest virtual machine's vnet0 is connected to the new bridge(virbr1)

```
# brctl show
bridge name     bridge id               STP enabled     interfaces
virbr0          8000.5254007da9f2       yes             virbr0-nic
virbr1          8000.525400682996       yes             virbr1-nic
vnet0
```

17.14. APPLYING NETWORK FILTERING

This section provides an introduction to libvirt's network filters, their goals, concepts and XML format.

17.14.1. Introduction

The goal of the network filtering, is to enable administrators of a virtualized system to configure and enforce network traffic filtering rules on virtual machines and manage the parameters of network traffic that virtual machines are allowed to send or receive. The network traffic filtering rules are applied on the

host physical machine when a virtual machine is started. Since the filtering rules cannot be circumvented from within the virtual machine, it makes them mandatory from the point of view of a virtual machine user.

From the point of view of the guest virtual machine, the network filtering system allows each virtual machine's network traffic filtering rules to be configured individually on a per interface basis. These rules are applied on the host physical machine when the virtual machine is started and can be modified while the virtual machine is running. The latter can be achieved by modifying the XML description of a network filter.

Multiple virtual machines can make use of the same generic network filter. When such a filter is modified, the network traffic filtering rules of all running virtual machines that reference this filter are updated. The machines that are not running will update on start.

As previously mentioned, applying network traffic filtering rules can be done on individual network interfaces that are configured for certain types of network configurations. Supported network types include:

- network
- ethernet -- must be used in bridging mode
- bridge

Example 17.1. An example of network filtering

The interface XML is used to reference a top-level filter. In the following example, the interface description references the filter clean-traffic.

```
 <devices>
  <interface type='bridge'>
    <mac address='00:16:3e:5d:c7:9e'/>
    <filterref filter='clean-traffic'/>
  </interface>
</devices>
```

Network filters are written in XML and may either contain: references to other filters, rules for traffic filtering, or hold a combination of both. The above referenced filter clean-traffic is a filter that only contains references to other filters and no actual filtering rules. Since references to other filters can be used, a tree of filters can be built. The clean-traffic filter can be viewed using the command: **`# virsh nwfilter-dumpxml clean-traffic`**.

As previously mentioned, a single network filter can be referenced by multiple virtual machines. Since interfaces will typically have individual parameters associated with their respective traffic filtering rules, the rules described in a filter's XML can be generalized using variables. In this case, the variable name is used in the filter XML and the name and value are provided at the place where the filter is referenced.

Example 17.2. Description extended

In the following example, the interface description has been extended with the parameter IP and a dotted IP address as a value.

```
<devices>
  <interface type='bridge'>
    <mac address='00:16:3e:5d:c7:9e'/>
```

```
      <filterref filter='clean-traffic'>
        <parameter name='IP' value='10.0.0.1'/>
      </filterref>
    </interface>
  </devices>
```

In this particular example, the clean-traffic network traffic filter will be represented with the IP address parameter 10.0.0.1 and as per the rule dictates that all traffic from this interface will always be using 10.0.0.1 as the source IP address, which is one of the purpose of this particular filter.

17.14.2. Filtering Chains

Filtering rules are organized in filter chains. These chains can be thought of as having a tree structure with packet filtering rules as entries in individual chains (branches).

Packets start their filter evaluation in the root chain and can then continue their evaluation in other chains, return from those chains back into the root chain or be dropped or accepted by a filtering rule in one of the traversed chains.

Libvirt's network filtering system automatically creates individual root chains for every virtual machine's network interface on which the user chooses to activate traffic filtering. The user may write filtering rules that are either directly instantiated in the root chain or may create protocol-specific filtering chains for efficient evaluation of protocol-specific rules.

The following chains exist:

- root
- mac
- stp (spanning tree protocol)
- vlan
- arp and rarp
- ipv4
- ipv6

Multiple chains evaluating the mac, stp, vlan, arp, rarp, ipv4, or ipv6 protocol can be created using the protocol name only as a prefix in the chain's name.

Example 17.3. ARP traffic filtering

This example allows chains with names arp-xyz or arp-test to be specified and have their ARP protocol packets evaluated in those chains.

The following filter XML shows an example of filtering ARP traffic in the arp chain.

```
<filter name='no-arp-spoofing' chain='arp' priority='-500'>
  <uuid>f88f1932-debf-4aa1-9fbe-f10d3aa4bc95</uuid>
  <rule action='drop' direction='out' priority='300'>
    <mac match='no' srcmacaddr='$MAC'/>
  </rule>
```

```
    <rule action='drop' direction='out' priority='350'>
      <arp match='no' arpsrcmacaddr='$MAC'/>
    </rule>
    <rule action='drop' direction='out' priority='400'>
      <arp match='no' arpsrcipaddr='$IP'/>
    </rule>
    <rule action='drop' direction='in' priority='450'>
      <arp opcode='Reply'/>
      <arp match='no' arpdstmacaddr='$MAC'/>
    </rule>
    <rule action='drop' direction='in' priority='500'>
      <arp match='no' arpdstipaddr='$IP'/>
    </rule>
    <rule action='accept' direction='inout' priority='600'>
      <arp opcode='Request'/>
    </rule>
    <rule action='accept' direction='inout' priority='650'>
      <arp opcode='Reply'/>
    </rule>
    <rule action='drop' direction='inout' priority='1000'/>
  </filter>
```

The consequence of putting ARP-specific rules in the arp chain, rather than for example in the root chain, is that packets protocols other than ARP do not need to be evaluated by ARP protocol-specific rules. This improves the efficiency of the traffic filtering. However, one must then pay attention to only putting filtering rules for the given protocol into the chain since other rules will not be evaluated. For example, an IPv4 rule will not be evaluated in the ARP chain since IPv4 protocol packets will not traverse the ARP chain.

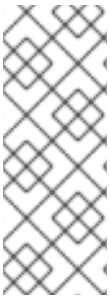
17.14.3. Filtering Chain Priorities

As previously mentioned, when creating a filtering rule, all chains are connected to the root chain. The order in which those chains are accessed is influenced by the priority of the chain. The following table shows the chains that can be assigned a priority and their default priorities.

Table 17.1. Filtering chain default priorities values

Chain (prefix)	Default priority
stp	-810
mac	-800
vlan	-750
ipv4	-700
ipv6	-600
arp	-500

Chain (prefix)	Default priority
rarp	-400



NOTE

A chain with a lower priority value is accessed before one with a higher value.

The chains listed in Table 17.1, “Filtering chain default priorities values” can be also be assigned custom priorities by writing a value in the range [-1000 to 1000] into the priority (XML) attribute in the filter node. Section 17.14.2, “Filtering Chains”filter shows the default priority of -500 for arp chains, for example.

17.14.4. Usage of Variables in Filters

There are two variables that have been reserved for usage by the network traffic filtering subsystem: MAC and IP.

MAC is designated for the MAC address of the network interface. A filtering rule that references this variable will automatically be replaced with the MAC address of the interface. This works without the user having to explicitly provide the MAC parameter. Even though it is possible to specify the MAC parameter similar to the IP parameter above, it is discouraged since libvirt knows what MAC address an interface will be using.

The parameter ***IP*** represents the IP address that the operating system inside the virtual machine is expected to use on the given interface. The IP parameter is special in so far as the libvirt daemon will try to determine the IP address (and thus the IP parameter's value) that is being used on an interface if the parameter is not explicitly provided but referenced. For current limitations on IP address detection, consult the section on limitations Section 17.14.12, “Limitations” on how to use this feature and what to expect when using it. The XML file shown in Section 17.14.2, “Filtering Chains” contains the filter ***no-arp-spoofing***, which is an example of using a network filter XML to reference the MAC and IP variables.

Note that referenced variables are always prefixed with the character ***$***. The format of the value of a variable must be of the type expected by the filter attribute identified in the XML. In the above example, the ***IP*** parameter must hold a legal IP address in standard format. Failure to provide the correct structure will result in the filter variable not being replaced with a value and will prevent a virtual machine from starting or will prevent an interface from attaching when hot plugging is being used. Some of the types that are expected for each XML attribute are shown in the example Example 17.4, “Sample variable types”.

Example 17.4. Sample variable types

As variables can contain lists of elements, (the variable IP can contain multiple IP addresses that are valid on a particular interface, for example), the notation for providing multiple elements for the IP variable is:

```
<devices>
  <interface type='bridge'>
    <mac address='00:16:3e:5d:c7:9e'/>
    <filterref filter='clean-traffic'>
      <parameter name='IP' value='10.0.0.1'/>
      <parameter name='IP' value='10.0.0.2'/>
```

```
        <parameter name='IP' value='10.0.0.3'/>
      </filterref>
    </interface>
  </devices>
```

This XML file creates filters to enable multiple IP addresses per interface. Each of the IP addresses will result in a separate filtering rule. Therefore, using the XML above and the following rule, three individual filtering rules (one for each IP address) will be created:

```
<rule action='accept' direction='in' priority='500'>
  <tcp srpipaddr='$IP'/>
</rule>
```

As it is possible to access individual elements of a variable holding a list of elements, a filtering rule like the following accesses the 2nd element of the variable *DSTPORTS*.

```
<rule action='accept' direction='in' priority='500'>
  <udp dstportstart='$DSTPORTS[1]'/>
</rule>
```

Example 17.5. Using a variety of variables

As it is possible to create filtering rules that represent all of the permissible rules from different lists using the notation **$VARIABLE[@<iterator id="x">]**. The following rule allows a virtual machine to receive traffic on a set of ports, which are specified in *DSTPORTS*, from the set of source IP address specified in *SRCIPADDRESSES*. The rule generates all combinations of elements of the variable *DSTPORTS* with those of *SRCIPADDRESSES* by using two independent iterators to access their elements.

```
<rule action='accept' direction='in' priority='500'>
  <ip srcipaddr='$SRCIPADDRESSES[@1]' dstportstart='$DSTPORTS[@2]'/>
</rule>
```

Assign concrete values to *SRCIPADDRESSES* and *DSTPORTS* as shown:

```
SRCIPADDRESSES = [ 10.0.0.1, 11.1.2.3 ]
DSTPORTS = [ 80, 8080 ]
```

Assigning values to the variables using **$SRCIPADDRESSES[@1]** and **$DSTPORTS[@2]** would then result in all variants of addresses and ports being created as shown:

- 10.0.0.1, 80
- 10.0.0.1, 8080
- 11.1.2.3, 80
- 11.1.2.3, 8080

Accessing the same variables using a single iterator, for example by using the notation **$SRCIPADDRESSES[@1]** and **$DSTPORTS[@1]**, would result in parallel access to both lists and result in the following combination:

- 10.0.0.1, 80
- 11.1.2.3, 8080

NOTE

$VARIABLE is short-hand for **$VARIABLE[@0]**. The former notation always assumes the role of iterator with **iterator id="0"** added as shown in the opening paragraph at the top of this section.

17.14.5. Automatic IP Address Detection and DHCP Snooping

This section provides information about automatic IP address detection and DHCP snooping.

17.14.5.1. Introduction

The detection of IP addresses used on a virtual machine's interface is automatically activated if the variable IP is referenced but no value has been assigned to it. The variable ***CTRL_IP_LEARNING*** can be used to specify the IP address learning method to use. Valid values include: *any*, *dhcp*, or *none*.

The value *any* instructs libvirt to use any packet to determine the address in use by a virtual machine, which is the default setting if the variable ***CTRL_IP_LEARNING*** is not set. This method will only detect a single IP address per interface. Once a guest virtual machine's IP address has been detected, its IP network traffic will be locked to that address, if for example, IP address spoofing is prevented by one of its filters. In that case, the user of the VM will not be able to change the IP address on the interface inside the guest virtual machine, which would be considered IP address spoofing. When a guest virtual machine is migrated to another host physical machine or resumed after a suspend operation, the first packet sent by the guest virtual machine will again determine the IP address that the guest virtual machine can use on a particular interface.

The value of *dhcp* instructs libvirt to only honor DHCP server-assigned addresses with valid leases. This method supports the detection and usage of multiple IP address per interface. When a guest virtual machine resumes after a suspend operation, any valid IP address leases are applied to its filters. Otherwise the guest virtual machine is expected to use DHCP to obtain a new IP addresses. When a guest virtual machine migrates to another physical host physical machine, the guest virtual machine is required to re-run the DHCP protocol.

If CTRL_IP_LEARNING is set to *none*, libvirt does not do IP address learning and referencing IP without assigning it an explicit value is an error.

17.14.5.2. DHCP Snooping

CTRL_IP_LEARNING=*dhcp* (DHCP snooping) provides additional anti-spoofing security, especially when combined with a filter allowing only trusted DHCP servers to assign IP addresses. To enable this, set the variable ***DHCPSERVER*** to the IP address of a valid DHCP server and provide filters that use this variable to filter incoming DHCP responses.

When DHCP snooping is enabled and the DHCP lease expires, the guest virtual machine will no longer be able to use the IP address until it acquires a new, valid lease from a DHCP server. If the guest virtual machine is migrated, it must get a new valid DHCP lease to use an IP address (for example by bringing the VM interface down and up again).

NOTE

Automatic DHCP detection listens to the DHCP traffic the guest virtual machine exchanges with the DHCP server of the infrastructure. To avoid denial-of-service attacks on libvirt, the evaluation of those packets is rate-limited, meaning that a guest virtual machine sending an excessive number of DHCP packets per second on an interface will not have all of those packets evaluated and thus filters may not get adapted. Normal DHCP client behavior is assumed to send a low number of DHCP packets per second. Further, it is important to setup appropriate filters on all guest virtual machines in the infrastructure to avoid them being able to send DHCP packets. Therefore, guest virtual machines must either be prevented from sending UDP and TCP traffic from port 67 to port 68 or the DHCPSERVER variable should be used on all guest virtual machines to restrict DHCP server messages to only be allowed to originate from trusted DHCP servers. At the same time anti-spoofing prevention must be enabled on all guest virtual machines in the subnet.

Example 17.6. Activating IPs for DHCP snooping

The following XML provides an example for the activation of IP address learning using the DHCP snooping method:

```
<interface type='bridge'>
  <source bridge='virbr0'/>
  <filterref filter='clean-traffic'>
    <parameter name='CTRL_IP_LEARNING' value='dhcp'/>
  </filterref>
</interface>
```

17.14.6. Reserved Variables

Table 17.2, “Reserved variables” shows the variables that are considered reserved and are used by libvirt:

Table 17.2. Reserved variables

Variable Name	Definition
MAC	The MAC address of the interface
IP	The list of IP addresses in use by an interface
IPV6	Not currently implemented: the list of IPV6 addresses in use by an interface
DHCPSERVER	The list of IP addresses of trusted DHCP servers
DHCPSERVERV6	Not currently implemented: The list of IPv6 addresses of trusted DHCP servers
CTRL_IP_LEARNING	The choice of the IP address detection mode

17.14.7. Element and Attribute Overview

The root element required for all network filters is named **<filter>** with two possible attributes. The **name** attribute provides a unique name of the given filter. The **chain** attribute is optional but allows certain filters to be better organized for more efficient processing by the firewall subsystem of the underlying host physical machine. Currently, the system only supports the following chains: **root**, **ipv4**, **ipv6**, **arp** and **rarp**.

17.14.8. References to Other Filters

Any filter may hold references to other filters. Individual filters may be referenced multiple times in a filter tree but references between filters must not introduce loops.

Example 17.7. An Example of a clean traffic filter

The following shows the XML of the clean-traffic network filter referencing several other filters.

```
<filter name='clean-traffic'>
  <uuid>6ef53069-ba34-94a0-d33d-17751b9b8cb1</uuid>
  <filterref filter='no-mac-spoofing'/>
  <filterref filter='no-ip-spoofing'/>
  <filterref filter='allow-incoming-ipv4'/>
  <filterref filter='no-arp-spoofing'/>
  <filterref filter='no-other-l2-traffic'/>
  <filterref filter='qemu-announce-self'/>
</filter>
```

To reference another filter, the XML node **<filterref>** needs to be provided inside a filter node. This node must have the attribute filter whose value contains the name of the filter to be referenced.

New network filters can be defined at any time and may contain references to network filters that are not known to libvirt, yet. However, once a virtual machine is started or a network interface referencing a filter is to be hot-plugged, all network filters in the filter tree must be available. Otherwise the virtual machine will not start or the network interface cannot be attached.

17.14.9. Filter Rules

The following XML shows a simple example of a network traffic filter implementing a rule to drop traffic if the IP address (provided through the value of the variable IP) in an outgoing IP packet is not the expected one, thus preventing IP address spoofing by the VM.

Example 17.8. Example of network traffic filtering

```
<filter name='no-ip-spoofing' chain='ipv4'>
  <uuid>fce8ae33-e69e-83bf-262e-30786c1f8072</uuid>
  <rule action='drop' direction='out' priority='500'>
    <ip match='no' srcipaddr='$IP'/>
  </rule>
</filter>
```

The traffic filtering rule starts with the rule node. This node may contain up to three of the following attributes:

- action is mandatory can have the following values:
  - drop (matching the rule silently discards the packet with no further analysis)
  - reject (matching the rule generates an ICMP reject message with no further analysis)
  - accept (matching the rule accepts the packet with no further analysis)
  - return (matching the rule passes this filter, but returns control to the calling filter for further analysis)
  - continue (matching the rule goes on to the next rule for further analysis)
- direction is mandatory can have the following values:
  - in for incoming traffic
  - out for outgoing traffic
  - inout for incoming and outgoing traffic
- priority is optional. The priority of the rule controls the order in which the rule will be instantiated relative to other rules. Rules with lower values will be instantiated before rules with higher values. Valid values are in the range of -1000 to 1000. If this attribute is not provided, priority 500 will be assigned by default. Note that filtering rules in the root chain are sorted with filters connected to the root chain following their priorities. This allows to interleave filtering rules with access to filter chains. See Section 17.14.3, “Filtering Chain Priorities” for more information.
- statematch is optional. Possible values are '0' or 'false' to turn the underlying connection state matching off. The default setting is 'true' or 1

For more information, see Section 17.14.11, “Advanced Filter Configuration Topics”.

The above example Example 17.7, “An Example of a clean traffic filter” indicates that the traffic of *type ip* will be associated with the chain *ipv4* and the rule will have **priority=*500***. If for example another filter is referenced whose traffic of *type ip* is also associated with the chain *ipv4* then that filter's rules will be ordered relative to the **priority=*500*** of the shown rule.

A rule may contain a single rule for filtering of traffic. The above example shows that traffic of type ip is to be filtered.

17.14.10. Supported Protocols

The following sections list and give some details about the protocols that are supported by the network filtering subsystem. This type of traffic rule is provided in the rule node as a nested node. Depending on the traffic type a rule is filtering, the attributes are different. The above example showed the single attribute ***srcipaddr*** that is valid inside the ip traffic filtering node. The following sections show what attributes are valid and what type of data they are expecting. The following datatypes are available:

- UINT8 : 8 bit integer; range 0-255
- UINT16: 16 bit integer; range 0-65535
- MAC_ADDR: MAC address in dotted decimal format, for example 00:11:22:33:44:55

- MAC_MASK: MAC address mask in MAC address format, for instance, FF:FF:FF:FC:00:00
- IP_ADDR: IP address in dotted decimal format, for example 10.1.2.3
- IP_MASK: IP address mask in either dotted decimal format (255.255.248.0) or CIDR mask (0-32)
- IPV6_ADDR: IPv6 address in numbers format, for example FFFF::1
- IPV6_MASK: IPv6 mask in numbers format (FFFF:FFFF:FC00::) or CIDR mask (0-128)
- STRING: A string
- BOOLEAN: 'true', 'yes', '1' or 'false', 'no', '0'
- IPSETFLAGS: The source and destination flags of the ipset described by up to 6 'src' or 'dst' elements selecting features from either the source or destination part of the packet header; example: src,src,dst. The number of 'selectors' to provide here depends on the type of ipset that is referenced

Every attribute except for those of type ***IP_MASK*** or ***IPV6_MASK*** can be negated using the match attribute with value *no*. Multiple negated attributes may be grouped together. The following XML fragment shows such an example using abstract attributes.

```
[...]
  <rule action='drop' direction='in'>
    <protocol match='no' attribute1='value1' attribute2='value2'/>
    <protocol attribute3='value3'/>
  </rule>
[...]
```

Rules behave evaluate the rule as well as look at it logically within the boundaries of the given protocol attributes. Thus, if a single attribute's value does not match the one given in the rule, the whole rule will be skipped during the evaluation process. Therefore, in the above example incoming traffic will only be dropped if: the protocol property ***attribute1*** does not match both ***value1*** and the protocol property ***attribute2*** does not match ***value2*** and the protocol property ***attribute3*** matches ***value3***.

17.14.10.1. MAC (Ethernet)

Protocol ID: mac

Rules of this type should go into the root chain.

Table 17.3. MAC protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination

Attribute Name	Datatype	Definition
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
protocolid	UINT16 (0x600-0xffff), STRING	Layer 3 protocol ID. Valid strings include [arp, rarp, ipv4, ipv6]
comment	STRING	text string up to 256 characters

The filter can be written as such:

```
[...]
<mac match='no' srcmacaddr='$MAC'/>
[...]
```

17.14.10.2. VLAN (802.1Q)

Protocol ID: vlan

Rules of this type should go either into the root or vlan chain.

Table 17.4. VLAN protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
vlan-id	UINT16 (0x0-0xfff, 0 - 4095)	VLAN ID
encap-protocol	UINT16 (0x03c-0xfff), String	Encapsulated layer 3 protocol ID, valid strings are arp, ipv4, ipv6
comment	STRING	text string up to 256 characters

17.14.10.3. STP (Spanning Tree Protocol)

Protocol ID: stp

Rules of this type should go either into the root or stp chain.

Table 17.5. STP protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
type	UINT8	Bridge Protocol Data Unit (BPDU) type
flags	UINT8	BPDU flagdstmacmask
root-priority	UINT16	Root priority range start
root-priority-hi	UINT16 (0x0-0xfff, 0 - 4095)	Root priority range end
root-address	MAC _ADDRESS	root MAC Address
root-address-mask	MAC _MASK	root MAC Address mask
roor-cost	UINT32	Root path cost (range start)
root-cost-hi	UINT32	Root path cost range end
sender-priority-hi	UINT16	Sender priority range end
sender-address	MAC_ADDRESS	BPDU sender MAC address
sender-address-mask	MAC_MASK	BPDU sender MAC address mask
port	UINT16	Port identifier (range start)
port_hi	UINT16	Port identifier range end
msg-age	UINT16	Message age timer (range start)
msg-age-hi	UINT16	Message age timer range end
max-age-hi	UINT16	Maximum age time range end
hello-time	UINT16	Hello time timer (range start)
hello-time-hi	UINT16	Hello time timer range end
forward-delay	UINT16	Forward delay (range start)

Attribute Name	Datatype	Definition
forward-delay-hi	UINT16	Forward delay range end
comment	STRING	text string up to 256 characters

17.14.10.4. ARP/RARP

Protocol ID: arp or rarp

Rules of this type should either go into the root or arp/rarp chain.

Table 17.6. ARP and RARP protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
hwtype	UINT16	Hardware type
protocoltype	UINT16	Protocol type
opcode	UINT16, STRING	Opcode valid strings are: Request, Reply, Request_Reverse, Reply_Reverse, DRARP_Request, DRARP_Reply, DRARP_Error, InARP_Request, ARP_NAK
arpsrcmacaddr	MAC_ADDR	Source MAC address in ARP/RARP packet
arpdstmacaddr	MAC _ADDR	Destination MAC address in ARP/RARP packet
arpsrcipaddr	IP_ADDR	Source IP address in ARP/RARP packet
arpdstipaddr	IP_ADDR	Destination IP address in ARP/RARP packet

Attribute Name	Datatype	Definition
gratuitous	BOOLEAN	Boolean indicating whether to check for a gratuitous ARP packet
comment	STRING	text string up to 256 characters

17.14.10.5. IPv4

Protocol ID: ip

Rules of this type should either go into the root or ipv4 chain.

Table 17.7. IPv4 protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
protocol	UINT8, STRING	Layer 4 protocol identifier. Valid strings for protocol are: tcp, udp, udplite, esp, ah, icmp, igmp, sctp
srcportstart	UINT16	Start of range of valid source ports; requires protocol
srcportend	UINT16	End of range of valid source ports; requires protocol
dstportstart	UNIT16	Start of range of valid destination ports; requires protocol

Attribute Name	Datatype	Definition
dstportend	UNIT16	End of range of valid destination ports; requires protocol
comment	STRING	text string up to 256 characters

17.14.10.6. IPv6

Protocol ID: ipv6

Rules of this type should either go into the root or ipv6 chain.

Table 17.8. IPv6 protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
protocol	UINT8, STRING	Layer 4 protocol identifier. Valid strings for protocol are: tcp, udp, udplite, esp, ah, icmpv6, sctp
scrportstart	UNIT16	Start of range of valid source ports; requires protocol
srcportend	UINT16	End of range of valid source ports; requires protocol
dstportstart	UNIT16	Start of range of valid destination ports; requires protocol

Attribute Name	Datatype	Definition
dstportend	UNIT16	End of range of valid destination ports; requires protocol
comment	STRING	text string up to 256 characters

17.14.10.7. TCP/UDP/SCTP

Protocol ID: tcp, udp, sctp

The chain parameter is ignored for this type of traffic and should either be omitted or set to root.

Table 17.9. TCP/UDP/SCTP protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
scripto	IP_ADDR	Start of range of source IP address
srcipfrom	IP_ADDR	End of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
scrportstart	UNIT16	Start of range of valid source ports; requires protocol
srcportend	UINT16	End of range of valid source ports; requires protocol
dstportstart	UNIT16	Start of range of valid destination ports; requires protocol

Attribute Name	Datatype	Definition
dstportend	UNIT16	End of range of valid destination ports; requires protocol
comment	STRING	text string up to 256 characters
state	STRING	comma separated list of NEW,ESTABLISHED,RELATED,INVALID or NONE
flags	STRING	TCP-only: format of mask/flags with mask and flags each being a comma separated list of SYN,ACK,URG,PSH,FIN,RST or NONE or ALL
ipset	STRING	The name of an IPSet managed outside of libvirt
ipsetflags	IPSETFLAGS	flags for the IPSet; requires ipset attribute

17.14.10.8. ICMP

Protocol ID: icmp

Note: The chain parameter is ignored for this type of traffic and should either be omitted or set to root.

Table 17.10. ICMP protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to the MAC address of the sender
dstmacaddr	MAD_ADDR	MAC address of the destination
dstmacmask	MAC_MASK	Mask applied to the MAC address of the destination
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address

Attribute Name	Datatype	Definition
dstipmask	IP_MASK	Mask applied to destination IP address
srcipfrom	IP_ADDR	start of range of source IP address
scripto	IP_ADDR	end of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
type	UNIT16	ICMP type
code	UNIT16	ICMP code
comment	STRING	text string up to 256 characters
state	STRING	comma separated list of NEW,ESTABLISHED,RELATED,INVALID or NONE
ipset	STRING	The name of an IPSet managed outside of libvirt
ipsetflags	IPSETFLAGS	flags for the IPSet; requires ipset attribute

17.14.10.9. IGMP, ESP, AH, UDPLITE, 'ALL'

Protocol ID: igmp, esp, ah, udplite, all

The chain parameter is ignored for this type of traffic and should either be omitted or set to root.

Table 17.11. IGMP, ESP, AH, UDPLITE, 'ALL'

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to the MAC address of the sender
dstmacaddr	MAD_ADDR	MAC address of the destination

Attribute Name	Datatype	Definition
dstmacmask	MAC_MASK	Mask applied to the MAC address of the destination
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
srcipfrom	IP_ADDR	start of range of source IP address
scripto	IP_ADDR	end of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
comment	STRING	text string up to 256 characters
state	STRING	comma separated list of NEW,ESTABLISHED,RELATED,INVALID or NONE
ipset	STRING	The name of an IPSet managed outside of libvirt
ipsetflags	IPSETFLAGS	flags for the IPSet; requires ipset attribute

17.14.10.10. TCP/UDP/SCTP over IPV6

Protocol ID: tcp-ipv6, udp-ipv6, sctp-ipv6

The chain parameter is ignored for this type of traffic and should either be omitted or set to root.

Table 17.12. TCP, UDP, SCTP over IPv6 protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender

Attribute Name	Datatype	Definition
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
srcipfrom	IP_ADDR	start of range of source IP address
scripto	IP_ADDR	end of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
srcportstart	UINT16	Start of range of valid source ports
srcportend	UINT16	End of range of valid source ports
dstportstart	UINT16	Start of range of valid destination ports
dstportend	UINT16	End of range of valid destination ports
comment	STRING	text string up to 256 characters
state	STRING	comma separated list of NEW,ESTABLISHED,RELATED,INVALID or NONE
ipset	STRING	The name of an IPSet managed outside of libvirt
ipsetflags	IPSETFLAGS	flags for the IPSet; requires ipset attribute

17.14.10.11. ICMPv6

Protocol ID: icmpv6

The chain parameter is ignored for this type of traffic and should either be omitted or set to root.

Table 17.13. ICMPv6 protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
srcipfrom	IP_ADDR	start of range of source IP address
scripto	IP_ADDR	end of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
type	UINT16	ICMPv6 type
code	UINT16	ICMPv6 code
comment	STRING	text string up to 256 characters
state	STRING	comma separated list of NEW,ESTABLISHED,RELATED,INVALID or NONE
ipset	STRING	The name of an IPSet managed outside of libvirt
ipsetflags	IPSETFLAGS	flags for the IPSet; requires ipset attribute

17.14.10.12. IGMP, ESP, AH, UDPLITE, 'ALL' over IPv6

Protocol ID: igmp-ipv6, esp-ipv6, ah-ipv6, udplite-ipv6, all-ipv6

The chain parameter is ignored for this type of traffic and should either be omitted or set to root.

Table 17.14. IGMP, ESP, AH, UDPLITE, 'ALL' over IPv6 protocol types

Attribute Name	Datatype	Definition
srcmacaddr	MAC_ADDR	MAC address of sender
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP address
srcipfrom	IP_ADDR	start of range of source IP address
scripto	IP_ADDR	end of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
comment	STRING	text string up to 256 characters
state	STRING	comma separated list of NEW,ESTABLISHED,RELATED,INVALID or NONE
ipset	STRING	The name of an IPSet managed outside of libvirt
ipsetflags	IPSETFLAGS	flags for the IPSet; requires ipset attribute

17.14.11. Advanced Filter Configuration Topics

The following sections discuss advanced filter configuration topics.

17.14.11.1. Connection tracking

The network filtering subsystem (on Linux) makes use of the connection tracking support of IP tables. This helps in enforcing the direction of the network traffic (state match) as well as counting and limiting the number of simultaneous connections towards a guest virtual machine. As an example, if a guest virtual machine has TCP port 8080 open as a server, clients may connect to the guest virtual machine on port 8080. Connection tracking and enforcement of the direction and then prevents the guest virtual machine from initiating a connection from (TCP client) port 8080 to the host physical machine back to a remote host physical machine. More importantly, tracking helps to prevent remote attackers from establishing a connection back to a guest virtual machine. For example, if the user inside the guest

virtual machine established a connection to port 80 on an attacker site, the attacker will not be able to initiate a connection from TCP port 80 back towards the guest virtual machine. By default the connection state match that enables connection tracking and then enforcement of the direction of traffic is turned on.

Example 17.9. XML example for turning off connections to the TCP port

The following shows an example XML fragment where this feature has been turned off for incoming connections to TCP port 12345.

```
[...]
 <rule direction='in' action='accept' statematch='false'>
   <cp dstportstart='12345'/>
 </rule>
[...]
```

This now allows incoming traffic to TCP port 12345, but would also enable the initiation from (client) TCP port 12345 within the VM, which may or may not be desirable.

17.14.11.2. Limiting number of connections

To limit the number of connections a guest virtual machine may establish, a rule must be provided that sets a limit of connections for a given type of traffic. If for example a VM is supposed to be allowed to only ping one other IP address at a time and is supposed to have only one active incoming ssh connection at a time.

Example 17.10. XML sample file that sets limits to connections

The following XML fragment can be used to limit connections

```
[...]
<rule action='drop' direction='in' priority='400'>
  <tcp connlimit-above='1'/>
</rule>
<rule action='accept' direction='in' priority='500'>
  <tcp dstportstart='22'/>
</rule>
<rule action='drop' direction='out' priority='400'>
  <icmp connlimit-above='1'/>
</rule>
<rule action='accept' direction='out' priority='500'>
  <icmp/>
</rule>
<rule action='accept' direction='out' priority='500'>
  <udp dstportstart='53'/>
</rule>
<rule action='drop' direction='inout' priority='1000'>
  <all/>
</rule>
[...]
```

NOTE

Limitation rules must be listed in the XML prior to the rules for accepting traffic. According to the XML file in Example 17.10, “XML sample file that sets limits to connections”, an additional rule for allowing DNS traffic sent to port 22 go out the guest virtual machine, has been added to avoid ssh sessions not getting established for reasons related to DNS lookup failures by the ssh daemon. Leaving this rule out may result in the ssh client hanging unexpectedly as it tries to connect. Additional caution should be used in regards to handling timeouts related to tracking of traffic. An ICMP ping that the user may have terminated inside the guest virtual machine may have a long timeout in the host physical machine's connection tracking system and will therefore not allow another ICMP ping to go through.

The best solution is to tune the timeout in the host physical machine's **sysfs** with the following command:# **echo 3 > /proc/sys/net/netfilter/nf_conntrack_icmp_timeout**. This command sets the ICMP connection tracking timeout to 3 seconds. The effect of this is that once one ping is terminated, another one can start after 3 seconds.

If for any reason the guest virtual machine has not properly closed its TCP connection, the connection to be held open for a longer period of time, especially if the TCP timeout value was set for a large amount of time on the host physical machine. In addition, any idle connection may result in a timeout in the connection tracking system which can be re-activated once packets are exchanged.

However, if the limit is set too low, newly initiated connections may force an idle connection into TCP backoff. Therefore, the limit of connections should be set rather high so that fluctuations in new TCP connections do not cause odd traffic behavior in relation to idle connections.

17.14.11.3. Command-line tools

virsh has been extended with life-cycle support for network filters. All commands related to the network filtering subsystem start with the prefix ***nwfilter***. The following commands are available:

- **nwfilter-list** : lists UUIDs and names of all network filters
- **nwfilter-define** : defines a new network filter or updates an existing one (must supply a name)
- **nwfilter-undefine** : deletes a specified network filter (must supply a name). Do not delete a network filter currently in use.
- **nwfilter-dumpxml** : displays a specified network filter (must supply a name)
- **nwfilter-edit** : edits a specified network filter (must supply a name)

17.14.11.4. Pre-existing network filters

The following is a list of example network filters that are automatically installed with libvirt:

Table 17.15. ICMPv6 protocol types

Protocol Name	Description
allow-arp	Accepts all incoming and outgoing Address Resolution Protocol (ARP) traffic to a guest virtual machine.
no-arp-spoofing, **no-arp-mac-spoofing**, **and no-arp-ip-spoofing**	These filters prevent a guest virtual machine from spoofing ARP traffic. In addition, they only allows ARP request and reply messages, and enforce that those packets contain: • no-arp-spoofing - the MAC and IP addresses of the guest • no-arp-mac-spoofing - the MAC address of the guest • no-arp-ip-spoofing - the IP address of the guest
low-dhcp	Allows a guest virtual machine to request an IP address via DHCP (from any DHCP server).
low-dhcp-server	Allows a guest virtual machine to request an IP address from a specified DHCP server. The dotted decimal IP address of the DHCP server must be provided in a reference to this filter. The name of the variable must be *DHCPSERVER*.
low-ipv4	Accepts all incoming and outgoing IPv4 traffic to a virtual machine.
low-incoming-ipv4	Accepts only incoming IPv4 traffic to a virtual machine. This filter is a part of the **clean-traffic** filter.
no-ip-spoofing	Prevents a guest virtual machine from sending IP packets with a source IP address different from the one inside the packet. This filter is a part of the **clean-traffic** filter.
no-ip-multicast	Prevents a guest virtual machine from sending IP multicast packets.
no-mac-broadcast	Prevents outgoing IPv4 traffic to a specified MAC address. This filter is a part of the **clean-traffic** filter.
no-other-l2-traffic	Prevents all layer 2 networking traffic except traffic specified by other filters used by the network. This filter is a part of the **clean-traffic** filter.

Protocol Name	Description
`no-other-rarp-traffic`, **`qemu-announce-self`**, **`qemu-announce-self-rarp`**	These filters allow QEMU's self-announce Reverse Address Resolution Protocol (RARP) packets, but prevent all other RARP traffic. All of them are also included in the **`clean-traffic`** filter.
`clean-traffic`	Prevents MAC, IP and ARP spoofing. This filter references several other filters as building blocks.

These filters are only building blocks and require a combination with other filters to provide useful network traffic filtering. The most used one in the above list is the *clean-traffic* filter. This filter itself can for example be combined with the *no-ip-multicast* filter to prevent virtual machines from sending IP multicast traffic on top of the prevention of packet spoofing.

17.14.11.5. Writing your own filters

Since libvirt only provides a couple of example networking filters, you may consider writing your own. When planning on doing so there are a couple of things you may need to know regarding the network filtering subsystem and how it works internally. Certainly you also have to know and understand the protocols very well that you want to be filtering on so that no further traffic than what you want can pass and that in fact the traffic you want to allow does pass.

The network filtering subsystem is currently only available on Linux host physical machines and only works for QEMU and KVM type of virtual machines. On Linux, it builds upon the support for ebtables, iptables and ip6tables and makes use of their features. Considering the list found in Section 17.14.10, “Supported Protocols” the following protocols can be implemented using ebtables:

- mac
- stp (spanning tree protocol)
- vlan (802.1Q)
- arp, rarp
- ipv4
- ipv6

Any protocol that runs over IPv4 is supported using iptables, those over IPv6 are implemented using ip6tables.

Using a Linux host physical machine, all traffic filtering rules created by libvirt's network filtering subsystem first passes through the filtering support implemented by ebtables and only afterwards through iptables or ip6tables filters. If a filter tree has rules with the protocols including: mac, stp, vlan arp, rarp, ipv4, or ipv6; the ebtable rules and values listed will automatically be used first.

Multiple chains for the same protocol can be created. The name of the chain must have a prefix of one of the previously enumerated protocols. To create an additional chain for handling of ARP traffic, a chain with name arp-test, can for example be specified.

As an example, it is possible to filter on UDP traffic by source and destination ports using the ip protocol filter and specifying attributes for the protocol, source and destination IP addresses and ports of UDP

packets that are to be accepted. This allows early filtering of UDP traffic with ebtables. However, once an IP or IPv6 packet, such as a UDP packet, has passed the ebtables layer and there is at least one rule in a filter tree that instantiates iptables or ip6tables rules, a rule to let the UDP packet pass will also be necessary to be provided for those filtering layers. This can be achieved with a rule containing an appropriate udp or udp-ipv6 traffic filtering node.

Example 17.11. Creating a custom filter

Suppose a filter is needed to fulfill the following list of requirements:

- prevents a VM's interface from MAC, IP and ARP spoofing
- opens only TCP ports 22 and 80 of a VM's interface
- allows the VM to send ping traffic from an interface but not let the VM be pinged on the interface
- allows the VM to do DNS lookups (UDP towards port 53)

The requirement to prevent spoofing is fulfilled by the existing ***clean-traffic*** network filter, thus the way to do this is to reference it from a custom filter.

To enable traffic for TCP ports 22 and 80, two rules are added to enable this type of traffic. To allow the guest virtual machine to send ping traffic a rule is added for ICMP traffic. For simplicity reasons, general ICMP traffic will be allowed to be initiated from the guest virtual machine, and will not be specified to ICMP echo request and response messages. All other traffic will be prevented to reach or be initiated by the guest virtual machine. To do this a rule will be added that drops all other traffic. Assuming the guest virtual machine is called **test** and the interface to associate our filter with is called **eth0**, a filter is created named **test-eth0**.

The result of these considerations is the following network filter XML:

```
<filter name='test-eth0'>
  <!- - This rule references the clean traffic filter to prevent MAC, IP
and ARP spoofing. By not providing an IP address parameter, libvirt will
detect the IP address the guest virtual machine is using. - ->
  <filterref filter='clean-traffic'/>

  <!- - This rule enables TCP ports 22 (ssh) and 80 (http) to be
reachable - ->
  <rule action='accept' direction='in'>
    <tcp dstportstart='22'/>
  </rule>

  <rule action='accept' direction='in'>
    <tcp dstportstart='80'/>
  </rule>

  <!- - This rule enables general ICMP traffic to be initiated by the
guest virtual machine including ping traffic - ->
  <rule action='accept' direction='out'>
    <icmp/>
  </rule>>

  <!- - This rule enables outgoing DNS lookups using UDP - ->
  <rule action='accept' direction='out'>
```

```
    <udp dstportstart='53'/>
  </rule>

  <!- - This rule drops all other traffic - ->
  <rule action='drop' direction='inout'>
    <all/>
  </rule>

</filter>
```

17.14.11.6. Sample custom filter

Although one of the rules in the above XML contains the IP address of the guest virtual machine as either a source or a destination address, the filtering of the traffic works correctly. The reason is that whereas the rule's evaluation occurs internally on a per-interface basis, the rules are additionally evaluated based on which (tap) interface has sent or will receive the packet, rather than what their source or destination IP address may be.

Example 17.12. Sample XML for network interface descriptions

An XML fragment for a possible network interface description inside the domain XML of the test guest virtual machine could then look like this:

```
[...]
 <interface type='bridge'>
   <source bridge='mybridge'/>
   <filterref filter='test-eth0'/>
 </interface>
[...]
```

To more strictly control the ICMP traffic and enforce that only ICMP echo requests can be sent from the guest virtual machine and only ICMP echo responses be received by the guest virtual machine, the above ICMP rule can be replaced with the following two rules:

```
<!- - enable outgoing ICMP echo requests- ->
<rule action='accept' direction='out'>
  <icmp type='8'/>
</rule>
```

```
<!- - enable incoming ICMP echo replies- ->
<rule action='accept' direction='in'>
  <icmp type='0'/>
</rule>
```

Example 17.13. Second example custom filter

This example demonstrates how to build a similar filter as in the example above, but extends the list of requirements with an ftp server located inside the guest virtual machine. The requirements for this filter are:

- prevents a guest virtual machine's interface from MAC, IP, and ARP spoofing

- opens only TCP ports 22 and 80 in a guest virtual machine's interface
- allows the guest virtual machine to send ping traffic from an interface but does not allow the guest virtual machine to be pinged on the interface
- allows the guest virtual machine to do DNS lookups (UDP towards port 53)
- enables the ftp server (in active mode) so it can run inside the guest virtual machine

The additional requirement of allowing an FTP server to be run inside the guest virtual machine maps into the requirement of allowing port 21 to be reachable for FTP control traffic as well as enabling the guest virtual machine to establish an outgoing TCP connection originating from the guest virtual machine's TCP port 20 back to the FTP client (FTP active mode). There are several ways of how this filter can be written and two possible solutions are included in this example.

The first solution makes use of the state attribute of the TCP protocol that provides a hook into the connection tracking framework of the Linux host physical machine. For the guest virtual machine-initiated FTP data connection (FTP active mode) the RELATED state is used to enable detection that the guest virtual machine-initiated FTP data connection is a consequence of (or 'has a relationship with') an existing FTP control connection, thereby allowing it to pass packets through the firewall. The RELATED state, however, is only valid for the very first packet of the outgoing TCP connection for the FTP data path. Afterwards, the state is ESTABLISHED, which then applies equally to the incoming and outgoing direction. All this is related to the FTP data traffic originating from TCP port 20 of the guest virtual machine. This then leads to the following solution:

```
<filter name='test-eth0'>
  <!- - This filter (eth0) references the clean traffic filter to
prevent MAC, IP, and ARP spoofing. By not providing an IP address
parameter, libvirt will detect the IP address the guest virtual machine
is using. - ->
  <filterref filter='clean-traffic'/>

  <!- - This rule enables TCP port 21 (FTP-control) to be reachable - ->
  <rule action='accept' direction='in'>
    <tcp dstportstart='21'/>
  </rule>

  <!- - This rule enables TCP port 20 for guest virtual machine-
initiated FTP data connection related to an existing FTP control
connection - ->
  <rule action='accept' direction='out'>
    <tcp srcportstart='20' state='RELATED,ESTABLISHED'/>
  </rule>

  <!- - This rule accepts all packets from a client on the FTP data
connection - ->
  <rule action='accept' direction='in'>
    <tcp dstportstart='20' state='ESTABLISHED'/>
  </rule>

  <!- - This rule enables TCP port 22 (SSH) to be reachable - ->
  <rule action='accept' direction='in'>
    <tcp dstportstart='22'/>
  </rule>

  <!- -This rule enables TCP port 80 (HTTP) to be reachable - ->
```

```
  <rule action='accept' direction='in'>
    <tcp dstportstart='80'/>
  </rule>

  <!- - This rule enables general ICMP traffic to be initiated by the
guest virtual machine, including ping traffic - ->
  <rule action='accept' direction='out'>
    <icmp/>
  </rule>

  <!- - This rule enables outgoing DNS lookups using UDP - ->
  <rule action='accept' direction='out'>
    <udp dstportstart='53'/>
  </rule>

  <!- - This rule drops all other traffic - ->
  <rule action='drop' direction='inout'>
    <all/>
  </rule>

</filter>
```

Before trying out a filter using the RELATED state, you have to make sure that the appropriate connection tracking module has been loaded into the host physical machine's kernel. Depending on the version of the kernel, you must run either one of the following two commands before the FTP connection with the guest virtual machine is established:

- # **modprobe nf_conntrack_ftp** - where available OR
- # **modprobe ip_conntrack_ftp** if above is not available

If protocols other than FTP are used in conjunction with the RELATED state, their corresponding module must be loaded. Modules are available for the protocols: ftp, tftp, irc, sip, sctp, and amanda.

The second solution makes use of the state flags of connections more than the previous solution did. This solution takes advantage of the fact that the NEW state of a connection is valid when the very first packet of a traffic flow is detected. Subsequently, if the very first packet of a flow is accepted, the flow becomes a connection and thus enters into the ESTABLISHED state. Therefore, a general rule can be written for allowing packets of ESTABLISHED connections to reach the guest virtual machine or be sent by the guest virtual machine. This is done writing specific rules for the very first packets identified by the NEW state and dictates the ports that the data is acceptable. All packets meant for ports that are not explicitly accepted are dropped, thus not reaching an ESTABLISHED state. Any subsequent packets sent from that port are dropped as well.

```
<filter name='test-eth0'>
  <!- - This filter references the clean traffic filter to prevent MAC,
IP and ARP spoofing. By not providing and IP address parameter, libvirt
will detect the IP address the VM is using. - ->
  <filterref filter='clean-traffic'/>

  <!- - This rule allows the packets of all previously accepted
connections to reach the guest virtual machine - ->
  <rule action='accept' direction='in'>
    <all state='ESTABLISHED'/>
  </rule>
```

```
  <!- - This rule allows the packets of all previously accepted and
related connections be sent from the guest virtual machine - ->
  <rule action='accept' direction='out'>
    <all state='ESTABLISHED,RELATED'/>
  </rule>

  <!- - This rule enables traffic towards port 21 (FTP) and port 22
(SSH)- ->
  <rule action='accept' direction='in'>
    <tcp dstportstart='21' dstportend='22' state='NEW'/>
  </rule>

  <!- - This rule enables traffic towards port 80 (HTTP) - ->
  <rule action='accept' direction='in'>
    <tcp dstportstart='80' state='NEW'/>
  </rule>

  <!- - This rule enables general ICMP traffic to be initiated by the
guest virtual machine, including ping traffic - ->
  <rule action='accept' direction='out'>
    <icmp state='NEW'/>
  </rule>

  <!- - This rule enables outgoing DNS lookups using UDP - ->
  <rule action='accept' direction='out'>
    <udp dstportstart='53' state='NEW'/>
  </rule>

  <!- - This rule drops all other traffic - ->
  <rule action='drop' direction='inout'>
    <all/>
  </rule>

</filter>
```

17.14.12. Limitations

The following is a list of the currently known limitations of the network filtering subsystem.

- VM migration is only supported if the whole filter tree that is referenced by a guest virtual machine's top level filter is also available on the target host physical machine. The network filter **`clean-traffic`** for example should be available on all libvirt installations and thus enable migration of guest virtual machines that reference this filter. To assure version compatibility is not a problem make sure you are using the most current version of libvirt by updating the package regularly.

- Migration must occur between libvirt insallations of version 0.8.1 or later in order not to lose the network traffic filters associated with an interface.

- VLAN (802.1Q) packets, if sent by a guest virtual machine, cannot be filtered with rules for protocol IDs arp, rarp, ipv4 and ipv6. They can only be filtered with protocol IDs, MAC and VLAN. Therefore, the example filter clean-traffic Example 17.1, “An example of network filtering” will not work as expected.

17.15. CREATING TUNNELS

This section will demonstrate how to implement different tunneling scenarios.

17.15.1. Creating Multicast Tunnels

A multicast group is setup to represent a virtual network. Any guest virtual machines whose network devices are in the same multicast group can talk to each other even across host physical machines. This mode is also available to unprivileged users. There is no default DNS or DHCP support and no outgoing network access. To provide outgoing network access, one of the guest virtual machines should have a second NIC which is connected to one of the first four network types thus providing appropriate routing. The multicast protocol is compatible the guest virtual machine user mode. Note that the source address that you provide must be from the address used for the multicast address block.

To create a multicast tunnel place the following XML details into the **<devices>** element:

```
...
<devices>
  <interface type='mcast'>
    <mac address='52:54:00:6d:90:01'>
    <source address='230.0.0.1' port='5558'/>
  </interface>
</devices>
...
```

Figure 17.28. Multicast tunnel domain XMl example

17.15.2. Creating TCP Tunnels

A TCP client-server architecture provides a virtual network. In this configuration, one guest virtual machine provides the server end of the network while all other guest virtual machines are configured as clients. All network traffic is routed between the guest virtual machine clients via the guest virtual machine server. This mode is also available for unprivileged users. Note that this mode does not provide default DNS or DHCP support and it does not provide outgoing network access. To provide outgoing network access, one of the guest virtual machines should have a second NIC which is connected to one of the first four network types thus providing appropriate routing.

To create a TCP tunnel place the following XML details into the **<devices>** element:

```
    ...
  <devices>
    <interface type='server'>
      <mac address='52:54:00:22:c9:42'>
      <source address='192.168.0.1' port='5558'/>
    </interface>
    ...
    <interface type='client'>
      <mac address='52:54:00:8b:c9:51'>
      <source address='192.168.0.1' port='5558'/>
    </interface>
  </devices>
  ...
```

Figure 17.29. TCP tunnel domain XMI example

17.16. SETTING VLAN TAGS

virtual local area network (vLAN) tags are added using the **`virsh net-edit`** command. This tag can also be used with PCI device assignment with SR-IOV devices. For more information, see Section 16.2.3, “Configuring PCI Assignment with SR-IOV Devices”.

```
<network>
  <name>ovs-net</name>
  <forward mode='bridge'/>
  <bridge name='ovsbr0'/>
  <virtualport type='openvswitch'>
    <parameters interfaceid='09b11c53-8b5c-4eeb-8f00-d84eaa0aaa4f'/>
  </virtualport>
  <vlan trunk='yes'>
    <tag id='42' nativeMode='untagged'/>
    <tag id='47'/>
  </vlan>
  <portgroup name='dontpanic'>
    <vlan>
      <tag id='42'/>
    </vlan>
  </portgroup>
</network>
```

Figure 17.30. vSetting VLAN tag (on supported network types only)

If (and only if) the network type supports vlan tagging transparent to the guest, an optional **`<vlan>`** element can specify one or more vlan tags to apply to the traffic of all guests using this network. (openvswitch and type='hostdev' SR-IOV networks do support transparent vlan tagging of guest traffic; everything else, including standard linux bridges and libvirt's own virtual networks, do not support it. 802.1Qbh (vn-link) and 802.1Qbg (VEPA) switches provide their own way (outside of libvirt) to tag guest traffic onto specific vlans.) As expected, the tag attribute specifies which vlan tag to use. If a network has

more than one **<vlan>** element defined, it is assumed that the user wants to do VLAN trunking using all the specified tags. If vlan trunking with a single tag is required, the optional attribute trunk='yes' can be added to the vlan element.

For network connections using openvswitch it is possible to configure the 'native-tagged' and 'native-untagged' vlan modes. This uses the optional nativeMode attribute on the **<tag>** element: nativeMode may be set to 'tagged' or 'untagged'. The id attribute of the element sets the native vlan.

<vlan> elements can also be specified in a **<portgroup>** element, as well as directly in a domain's **<interface>** element. If a vlan tag is specified in multiple locations, the setting in **<interface>** takes precedence, followed by the setting in the **<portgroup>** selected by the interface config. The **<vlan>** in **<network>** will be selected only if none is given in **<portgroup>** or **<interface>**.

17.17. APPLYING QOS TO YOUR VIRTUAL NETWORK

Quality of Service (QoS) refers to the resource control systems that guarantees an optimal experience for all users on a network, making sure that there is no delay, jitter, or packet loss. QoS can be application specific or user / group specific. See Section 23.18.9.14, “Quality of service (QoS)” for more information.

CHAPTER 18. REMOTE MANAGEMENT OF GUESTS

This section explains how to remotely manage your guests.

18.1. TRANSPORT MODES

For remote management, **libvirt** supports the following transport modes:

Transport Layer Security (TLS)

Transport Layer Security TLS 1.0 (SSL 3.1) authenticated and encrypted TCP/IP socket, usually listening on a public port number. To use this, you will need to generate client and server certificates. The standard port is 16514. For detailed instructions, see Section 18.3, “Remote Management over TLS and SSL”.

SSH

Transported over a Secure Shell protocol (SSH) connection. The libvirt daemon (**libvirtd**) must be running on the remote machine. Port 22 must be open for SSH access. You should use some sort of SSH key management (for example, the **ssh-agent** utility) or you will be prompted for a password. For detailed instructions, see Section 18.2, “Remote Management with SSH”.

UNIX Sockets

UNIX domain sockets are only accessible on the local machine. Sockets are not encrypted, and use UNIX permissions or SELinux for authentication. The standard socket names are **/var/run/libvirt/libvirt-sock** and **/var/run/libvirt/libvirt-sock-ro** (for read-only connections).

ext

The ***ext*** parameter is used for any external program which can make a connection to the remote machine by means outside the scope of libvirt. This parameter is unsupported.

TCP

Unencrypted TCP/IP socket. Not recommended for production use, this is normally disabled, but an administrator can enable it for testing or use over a trusted network. The default port is 16509.

The default transport, if no other is specified, is TLS.

Remote URIs

A Uniform Resource Identifier (URI) is used by **virsh** and libvirt to connect to a remote host. URIs can also be used with the **--connect** parameter for the **virsh** command to execute single commands or migrations on remote hosts. Remote URIs are formed by taking ordinary local URIs and adding a host name or a transport name, or both. As a special case, using a URI scheme of 'remote' will tell the remote libvirtd server to probe for the optimal hypervisor driver. This is equivalent to passing a NULL URI for a local connection

libvirt URIs take the general form (content in square brackets, "[]", represents optional functions):

```
driver[+transport]://[username@][hostname][:port]/path[?extraparameters]
```

Note that if the hypervisor (driver) is QEMU, the path is mandatory.

The following are examples of valid remote URIs:

- qemu://hostname/

The transport method or the host name must be provided to target an external location. For more information, see the libvirt upstream documentation.

Examples of remote management parameters

- Connect to a remote KVM host named **host2**, using SSH transport and the SSH user name **virtuser**. The connect command for each is **connect [*URI*] [--readonly]**. For more information about the **virsh connect** command, see Section 20.4, “Connecting to the Hypervisor with virsh Connect”

```
qemu+ssh://virtuser@host2/
```

- Connect to a remote KVM hypervisor on the host named **host2** using TLS.

```
qemu://host2/
```

Testing examples

- Connect to the local KVM hypervisor with a non-standard UNIX socket. The full path to the UNIX socket is supplied explicitly in this case.

```
qemu+unix:///system?socket=/opt/libvirt/run/libvirt/libvirt-sock
```

- Connect to the libvirt daemon with an non-encrypted TCP/IP connection to the server with the IP address 10.1.1.10 on port 5000. This uses the test driver with default settings.

```
test+tcp://10.1.1.10:5000/default
```

Extra URI Parameters

Extra parameters can be appended to remote URIs. The table below covers the recognized parameters. All other parameters are ignored. Note that parameter values must be URI-escaped (that is, a question mark (?) is appended before the parameter and special characters are converted into the URI format).

Table 18.1. Extra URI parameters

Name	Transport mode	Description	Example usage

Name	Transport mode	Description	Example usage
name	all modes	The name passed to the remote **virConnectOpen** function. The name is normally formed by removing ***transport***, ***hostname***, ***port number***, ***username***, and extra parameters from the remote URI, but in certain very complex cases it may be better to supply the name explicitly.	name=qemu:///system
command	ssh and ext	The external command. For ext transport this is required. For ssh the default is ssh. The PATH is searched for the command.	command=/opt/openssh/bin/ssh
socket	unix and ssh	The path to the UNIX domain socket, which overrides the default. For ssh transport, this is passed to the remote **netcat** command (see netcat).	socket=/opt/libvirt/run/libvirt/libvirt-sock
no_verify	tls	If set to a non-zero value, this disables client checks of the server's certificate. Note that to disable server checks of the client's certificate or IP address you must change the libvirtd configuration.	no_verify=1
no_tty	ssh	If set to a non-zero value, this stops ssh from asking for a password if it cannot log in to the remote machine automatically . Use this when you do not have access to a terminal.	no_tty=1

18.2. REMOTE MANAGEMENT WITH SSH

The **ssh** package provides an encrypted network protocol that can securely send management functions to remote virtualization servers. The method described below uses the **libvirt** management connection, securely tunneled over an **SSH** connection, to manage the remote machines. All the authentication is done using **SSH** public key cryptography and passwords or passphrases gathered by your local **SSH** agent. In addition, the **VNC** console for each guest is tunneled over **SSH**.

When using using **SSH** for remotely managing your virtual machines, be aware of the following problems:

- You require root log in access to the remote machine for managing virtual machines.
- The initial connection setup process may be slow.
- There is no standard or trivial way to revoke a user's key on all hosts or guests.
- **SSH** does not scale well with larger numbers of remote machines.

NOTE

Red Hat Virtualization enables remote management of large numbers of virtual machines. For further details, see the Red Hat Virtualization documentation.

The following packages are required for SSH access:

- openssh
- openssh-askpass
- openssh-clients
- openssh-server

Configuring Password-less or Password-managed SSH Access for `virt-manager`

The following instructions assume you are starting from scratch and do not already have **SSH** keys set up. If you have SSH keys set up and copied to the other systems, you can skip this procedure.

IMPORTANT

SSH keys are user-dependent and may only be used by their owners. A key's owner is the user who generated it. Keys may not be shared across different users.

`virt-manager` must be run by the user who owns the keys to connect to the remote host. That means, if the remote systems are managed by a non-root user, **`virt-manager`** must be run in unprivileged mode. If the remote systems are managed by the local root user, then the SSH keys must be owned and created by root.

You cannot manage the local host as an unprivileged user with **`virt-manager`**.

1. **Optional: Changing user**
   Change user, if required. This example uses the local root user for remotely managing the other hosts and the local host.

   ```
   $ su -
   ```

2. **Generating the SSH key pair**
   Generate a public key pair on the machine where **virt-manager** is used. This example uses the default key location, in the **~/.ssh/** directory.

   ```
   # ssh-keygen -t rsa
   ```

3. **Copying the keys to the remote hosts**
   Remote login without a password, or with a pass-phrase, requires an SSH key to be distributed to the systems being managed. Use the **ssh-copy-id** command to copy the key to root user at the system address provided (in the example, ***root@host2.example.com***).

   ```
   # ssh-copy-id -i ~/.ssh/id_rsa.pub root@host2.example.com
   root@host2.example.com's password:
   ```

   Afterwards, try logging into the machine and check the **.ssh/authorized_keys** file to make sure unexpected keys have not been added:

   ```
   ssh root@host2.example.com
   ```

   Repeat for other systems, as required.

4. **Optional: Add the passphrase to the ssh-agent**
   Add the pass-phrase for the SSH key to the **ssh-agent**, if required. On the local host, use the following command to add the pass-phrase (if there was one) to enable password-less login.

   ```
   # ssh-add ~/.ssh/id_rsa
   ```

   This command will fail to run if the ssh-agent is not running. To avoid errors or conflicts, make sure that your SSH parameters are set correctly. See the *Red Hat Enterprise System Administration Guide* for more information.

The libvirt daemon (libvirtd)

The **libvirt** daemon provides an interface for managing virtual machines. You must have the **libvirtd** daemon installed and running on every remote host that you intend to manage this way.

```
$ ssh root@somehost
# systemctl enable libvirtd.service
# systemctl start libvirtd.service
```

After **libvirtd** and **SSH** are configured, you should be able to remotely access and manage your virtual machines. You should also be able to access your guests with **VNC** at this point.

Accessing Remote Hosts with virt-manager

Remote hosts can be managed with the **virt-manager** GUI tool. SSH keys must belong to the user executing **virt-manager** for password-less login to work.

1. Start **virt-manager**.

2. Open the **File** ⇒ **Add Connection** menu.

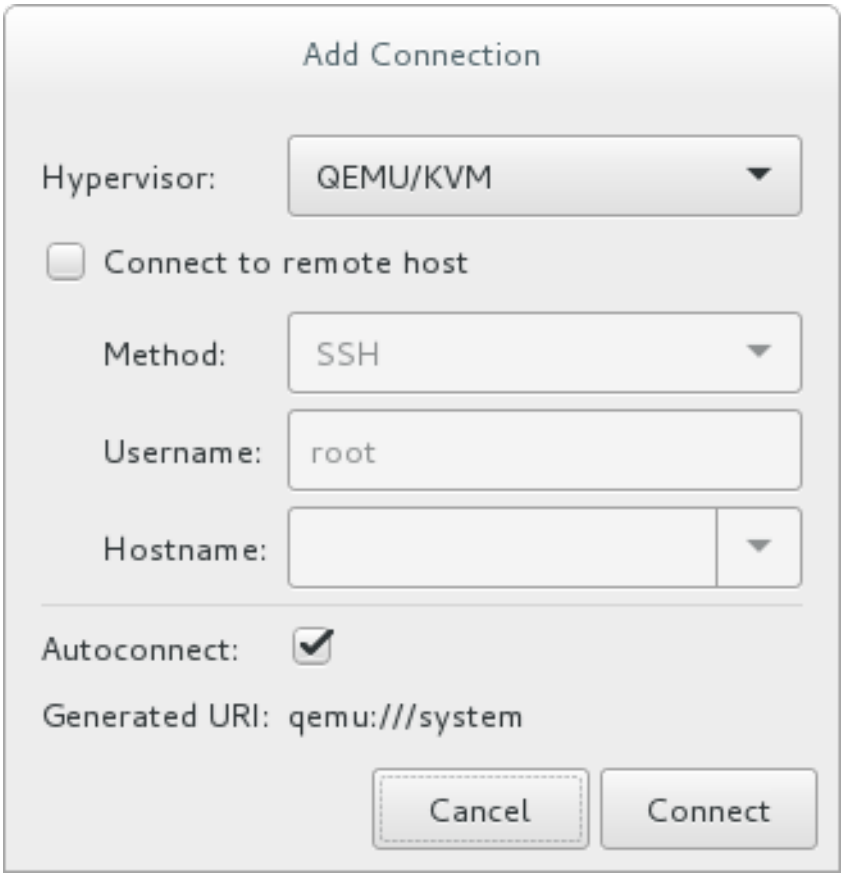


Figure 18.1. Add connection menu

3. Use the drop down menu to select hypervisor type, and click the **Connect to remote host** check box to open the Connection **Method** (in this case Remote tunnel over SSH), enter the **User name** and **Hostname**, then click **Connect**.

18.3. REMOTE MANAGEMENT OVER TLS AND SSL

You can manage virtual machines using the TLS and SSL protocols. TLS and SSL provides greater scalability but is more complicated than SSH (refer to Section 18.2, “Remote Management with SSH”). TLS and SSL is the same technology used by web browsers for secure connections. The **`libvirt`** management connection opens a TCP port for incoming connections, which is securely encrypted and authenticated based on x509 certificates. The following procedures provide instructions on creating and deploying authentication certificates for TLS and SSL management.

Procedure 18.1. Creating a certificate authority (CA) key for TLS management

1. Before you begin, confirm that **`gnutls-utils`** is installed. If not, install it:

```
# yum install gnutls-utils
```

2. Generate a private key, using the following command:

```
# certtool --generate-privkey > cakey.pem
```

3. After the key is generated, create a signature file so the key can be self-signed. To do this, create a file with signature details and name it **`ca.info`**. This file should contain the following:

```
cn = Name of your organization
```

```
ca
cert_signing_key
```

4. Generate the self-signed key with the following command:

```
# certtool --generate-self-signed --load-privkey cakey.pem --
template ca.info --outfile cacert.pem
```

After the file is generated, the **ca.info** file can be deleted using the **rm** command. The file that results from the generation process is named **cacert.pem**. This file is the public key (certificate). The loaded file **cakey.pem** is the private key. For security purposes, this file should be kept private, and not reside in a shared space.

5. Install the **cacert.pem** CA certificate file on all clients and servers in the **/etc/pki/CA/cacert.pem** directory to let them know that the certificate issued by your CA can be trusted. To view the contents of this file, run:

```
# certtool -i --infile cacert.pem
```

This is all that is required to set up your CA. Keep the CA's private key safe, as you will need it in order to issue certificates for your clients and servers.

Procedure 18.2. Issuing a server certificate

This procedure demonstrates how to issue a certificate with the X.509 Common Name (CN) field set to the host name of the server. The CN must match the host name which clients will be using to connect to the server. In this example, clients will be connecting to the server using the URI: **qemu://*mycommonname*/system**, so the CN field should be identical, for this example "mycommoname".

1. Create a private key for the server.

```
# certtool --generate-privkey > serverkey.pem
```

2. Generate a signature for the CA's private key by first creating a template file called **server.info**. Make sure that the CN is set to be the same as the server's host name:

```
organization = Name of your organization
cn = mycommonname
tls_www_server
encryption_key
signing_key
```

3. Create the certificate:

```
# certtool --generate-certificate --load-privkey serverkey.pem --
load-ca-certificate cacert.pem --load-ca-privkey cakey.pem \ --
template server.info --outfile servercert.pem
```

This results in two files being generated:

- serverkey.pem - The server's private key

- servercert.pem - The server's public key

4. Make sure to keep the location of the private key secret. To view the contents of the file, use the following command:

```
# certtool -i --infile servercert.pem
```

When opening this file, the **CN=** parameter should be the same as the CN that you set earlier. For example, **mycommonname**.

5. Install the two files in the following locations:

- **serverkey.pem** - the server's private key. Place this file in the following location: **/etc/pki/libvirt/private/serverkey.pem**
- **servercert.pem** - the server's certificate. Install it in the following location on the server: **/etc/pki/libvirt/servercert.pem**

Procedure 18.3. Issuing a client certificate

1. For every client (that is to say any program linked with libvirt, such as **virt-manager**), you need to issue a certificate with the X.509 Distinguished Name (DN) field set to a suitable name. This needs to be decided on a corporate level.

For example purposes, the following information will be used:

```
C=USA,ST=North Carolina,L=Raleigh,O=Red Hat,CN=name_of_client
```

2. Create a private key:

```
# certtool --generate-privkey > clientkey.pem
```

3. Generate a signature for the CA's private key by first creating a template file called **client.info**. The file should contain the following (fields should be customized to reflect your region/location):

```
country = USA
state = North Carolina
locality = Raleigh
organization = Red Hat
cn = client1
tls_www_client
encryption_key
signing_key
```

4. Sign the certificate with the following command:

```
# certtool --generate-certificate --load-privkey clientkey.pem --
load-ca-certificate cacert.pem \ --load-ca-privkey cakey.pem --
template client.info --outfile clientcert.pem
```

5. Install the certificates on the client machine:

```
# cp clientkey.pem /etc/pki/libvirt/private/clientkey.pem
# cp clientcert.pem /etc/pki/libvirt/clientcert.pem
```

18.4. CONFIGURING A VNC SERVER

To set up graphical desktop sharing between the host and the guest machine using Virtual Network Computing (VNC), a VNC server has to be configured on the guest you wish to connect to. To do this, VNC has to be specified as a graphics type in the **devices** element of the guest's XML file. For further information, see Section 23.18.12, “Graphical Framebuffers”.

To connect to a VNC server, use the **virt-viewer** utility or the **virt-manager** interface.

18.5. ENHANCING REMOTE MANAGEMENT OF VIRTUAL MACHINES WITH NSS

In Red Hat Enterprise Linux 7.3 and later, you can use the libvirt Network Security Services (NSS) module to make it easier to connect to guests with SSH, TLS, SSL, as well as other remote login services. In addition, the module also benefits utilities that use host name translation, such as **ping**.

To be able to use this functionality, install the libvirt-nss package:

```
# yum install libvirt-nss
```

NOTE

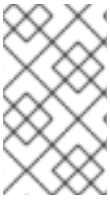
If installing libvirt-nss fails, make sure that the **Optional** repository for Red Hat Enterprise Linux is enabled. For instructions, see the System Administrator's Guide.

Finally, enable the module by adding **libvirt** to the **hosts** line of the **/etc/nsswitch.conf** file, for example as follows:

```
passwd:     compat
shadow:     compat
group:      compat
hosts:      files libvirt dns
...
```

The order in which modules are listed on the **hosts** line determines the order in which these modules are consulted to find the specified remote guest. As a result, libvirt's NSS module is added to modules that translate host domain names to IP addresses. This for example enables connecting to a remote guest in NAT mode without setting a static IP address and only using the guest's **hostname** value:

```
# ssh root@guest-hostname
root@guest-hostname's password:
Last login: Thu Aug 10 09:12:31 2017 from 192.168.122.1
[root@guest1-rhel7 ~]#
```



NOTE

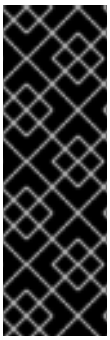
The guest's hostname may differ from the guest name displayed for example by **`virsh list`**. To display or configure the hostname on the guest, use the hostnamectl commands.

CHAPTER 19. MANAGING GUESTS WITH THE VIRTUAL MACHINE MANAGER (VIRT-MANAGER)

This chapter describes the Virtual Machine Manager (**virt-manager**) windows, dialog boxes, and various GUI controls.

virt-manager provides a graphical view of hypervisors and guests on your host system and on remote host systems. **virt-manager** can perform virtualization management tasks, including:

- defining and creating guests,
- assigning memory,
- assigning virtual CPUs,
- monitoring operational performance,
- saving and restoring, pausing and resuming, and shutting down and starting guests,
- links to the textual and graphical consoles, and
- live and offline migrations.



IMPORTANT

It is important to note which user you are using. If you create a guest virtual machine with one user, you will not be able to retrieve information about it using another user. This is especially important when you create a virtual machine in virt-manager. The default user is root in that case unless otherwise specified. Should you have a case where you cannot list the virtual machine using the **virsh list --all** command, it is most likely due to you running the command using a different user than you used to create the virtual machine.

19.1. STARTING VIRT-MANAGER

To start **virt-manager** session open the **Applications** menu, then the **System Tools** menu and select **Virtual Machine Manager** (**virt-manager**).

The **virt-manager** main window appears.

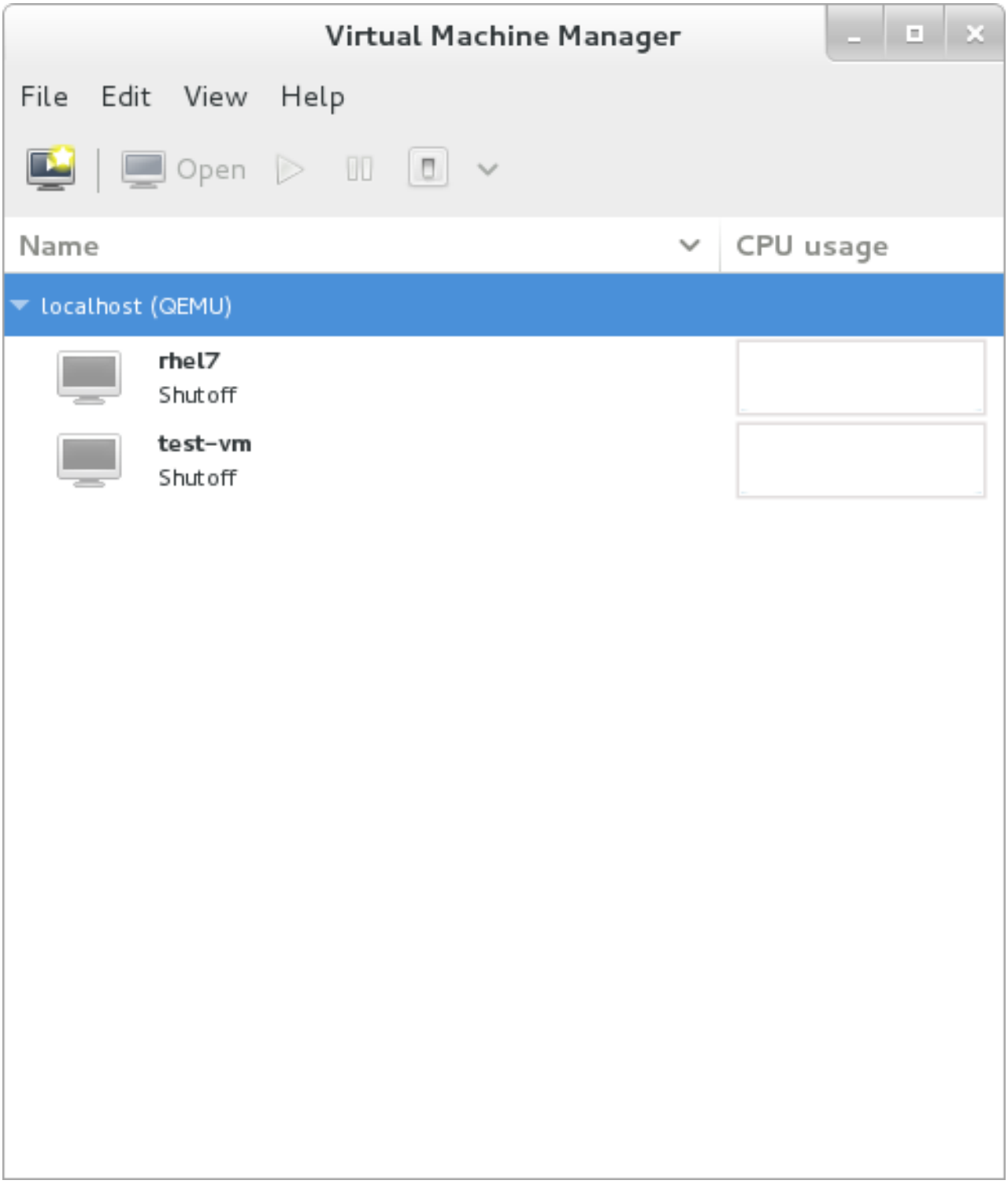


Figure 19.1. Starting `virt-manager`

Alternatively, **`virt-manager`** can be started remotely using **`ssh`** as demonstrated in the following command:

```
# ssh -X host's address
[remotehost]# virt-manager
```

Using **`ssh`** to manage virtual machines and hosts is discussed further in Section 18.2, “Remote Management with SSH”.

19.2. THE VIRTUAL MACHINE MANAGER MAIN WINDOW

This main window displays all the running guests and resources used by guests. Select a guest by double clicking the guest's name.

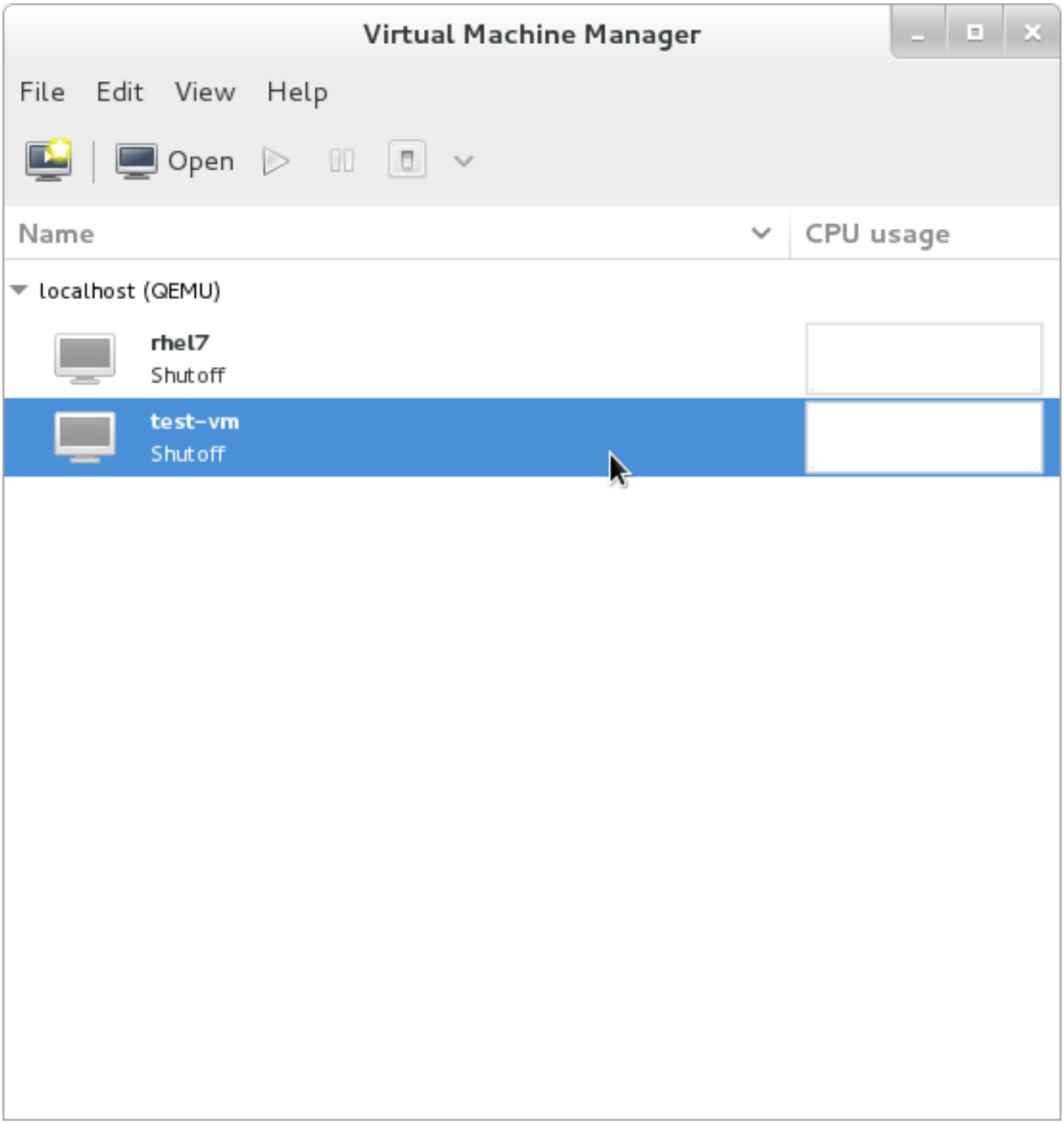


Figure 19.2. Virtual Machine Manager main window

19.3. THE VIRTUAL HARDWARE DETAILS WINDOW

The virtual hardware details window displays information about the virtual hardware configured for the guest. Virtual hardware resources can be added, removed and modified in this window. To access the virtual hardware details window, click the icon in the toolbar.

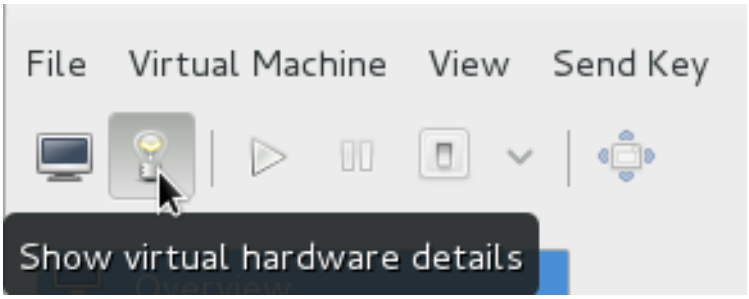


Figure 19.3. The virtual hardware details icon

Clicking the icon displays the virtual hardware details window.

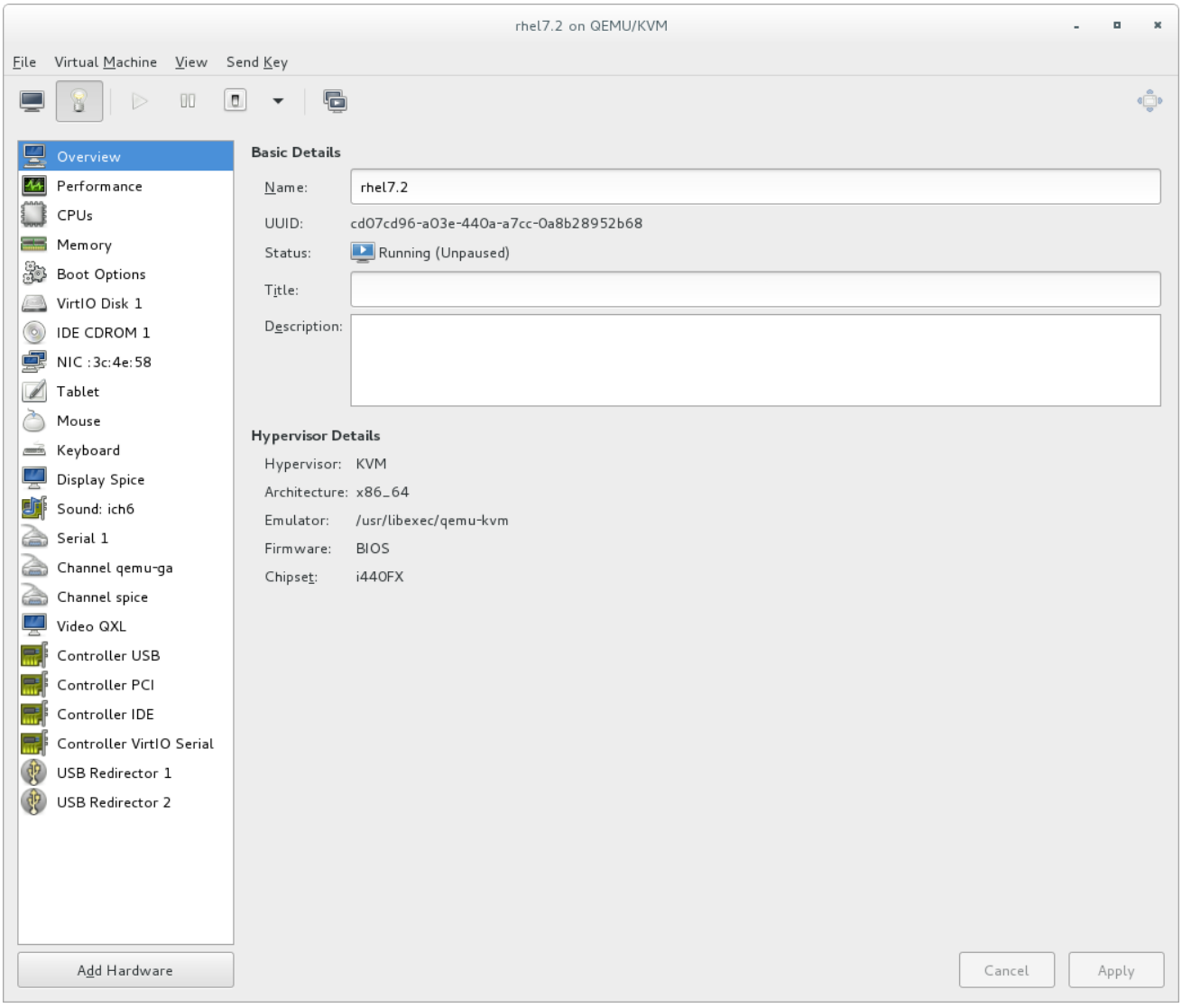


Figure 19.4. The virtual hardware details window

19.3.1. Applying Boot Options to Guest Virtual Machines

Using virt-manager you can select how the guest virtual machine will act on boot. The boot options will not take effect until the guest virtual machine reboots. You can either power down the virtual machine before making any changes, or you can reboot the machine afterwards. If you do not do either of these options, the changes will happen the next time the guest reboots.

Procedure 19.1. Configuring boot options

1. From the Virtual Machine Manager **Edit** menu, select **Virtual Machine Details**.

2. From the side panel, select **Boot Options** and then complete any or all of the following optional steps:

   a. To indicate that this guest virtual machine should start each time the host physical machine boots, select the **Autostart** check box.

   b. To indicate the order in which guest virtual machine should boot, click the **Enable boot menu** check box. After this is checked, you can then check the devices you want to boot from and using the arrow keys change the order that the guest virtual machine will use when booting.

   c. If you want to boot directly from the Linux kernel, expand the **Direct kernel boot** menu. Fill in the **Kernel path**, **Initrd path**, and the **Kernel arguments** that you want to use.

3. Click **Apply**.

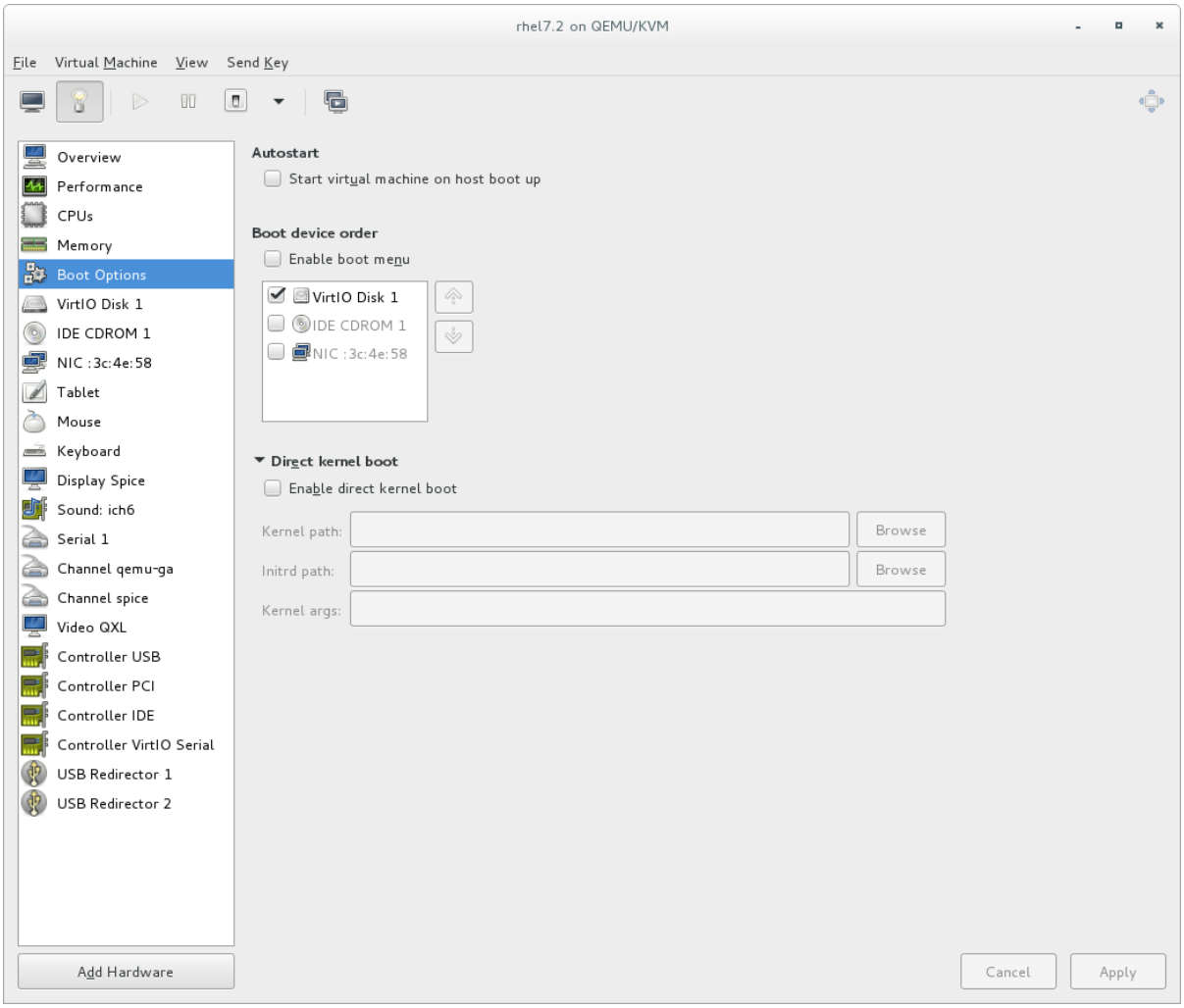
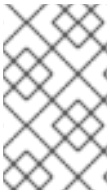


Figure 19.5. Configuring boot options

19.3.2. Attaching USB Devices to a Guest Virtual Machine



NOTE

In order to attach the USB device to the guest virtual machine, you first must attach it to the host physical machine and confirm that the device is working. If the guest is running, you need to shut it down before proceeding.

Procedure 19.2. Attaching USB devices using Virt-Manager

1. Open the guest virtual machine's Virtual Machine Details screen.
2. Click **`Add Hardware`**
3. In the **Add New Virtual Hardware** popup, select **USB Host Device**, select the device you want to attach from the list and Click **`Finish`**.

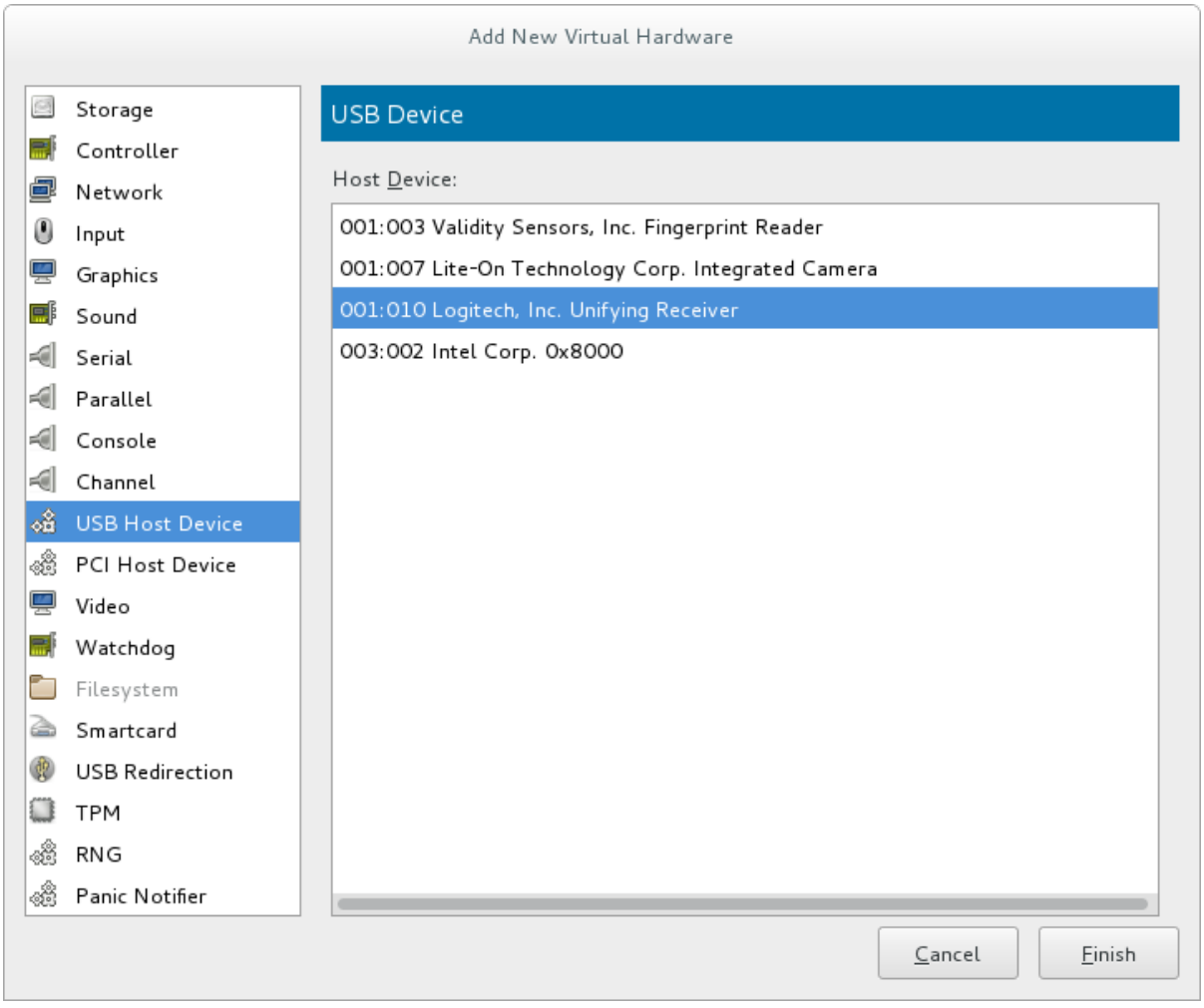


Figure 19.6. Add USB Device

4. To use the USB device in the guest virtual machine, start the guest virtual machine.

19.3.3. USB Redirection

USB re-direction is best used in cases where there is a host physical machine that is running in a data center. The user connects to his/her guest virtual machine from a local machine or thin client. On this local machine there is a SPICE client. The user can attach any USB device to the thin client and the SPICE client will redirect the device to the host physical machine on the data center so it can be used by the VM that is running on the thin client.

Procedure 19.3. Redirecting USB devices

1. Open the guest virtual machine's Virtual Machine Details screen.
2. Click **`Add Hardware`**
3. In the **Add New Virtual Hardware** popup, select **USB Redirection**. Make sure to select **Spice channel** from the **Type** drop-down menu and click **`Finish`**.

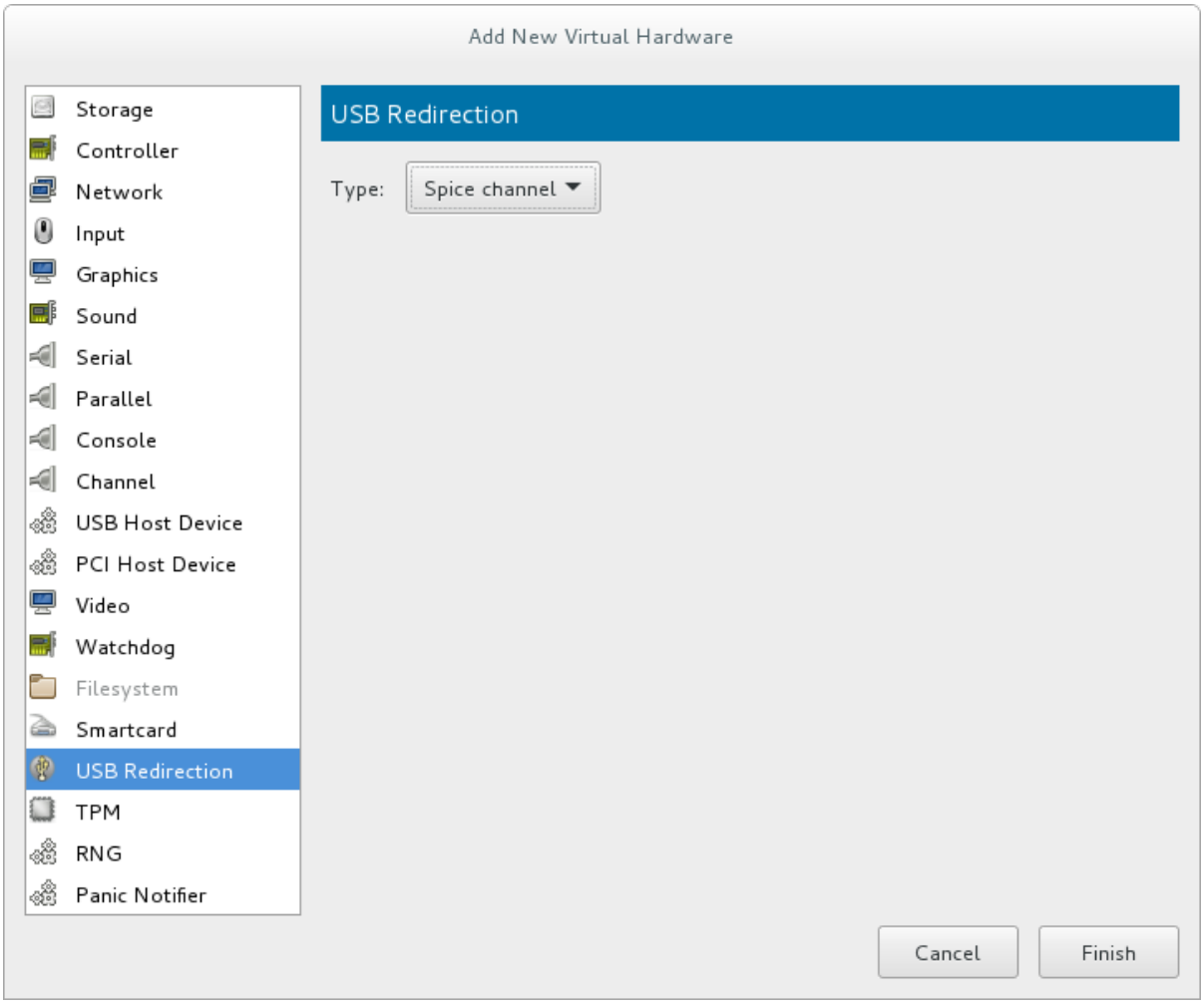


Figure 19.7. Add New Virtual Hardware window

4. Open the **Virtual Machine** menu and select **Redirect USB device**. A pop-up window opens with a list of USB devices.

Figure 19.8. Select a USB device

5. Select a USB device for redirection by checking its check box and click **OK**.

19.4. VIRTUAL MACHINE GRAPHICAL CONSOLE

This window displays a guest's graphical console. Guests can use several different protocols to export their graphical frame buffers: **`virt-manager`** supports **VNC** and **SPICE**. If your virtual machine is set to require authentication, the Virtual Machine graphical console prompts you for a password before the display appears.

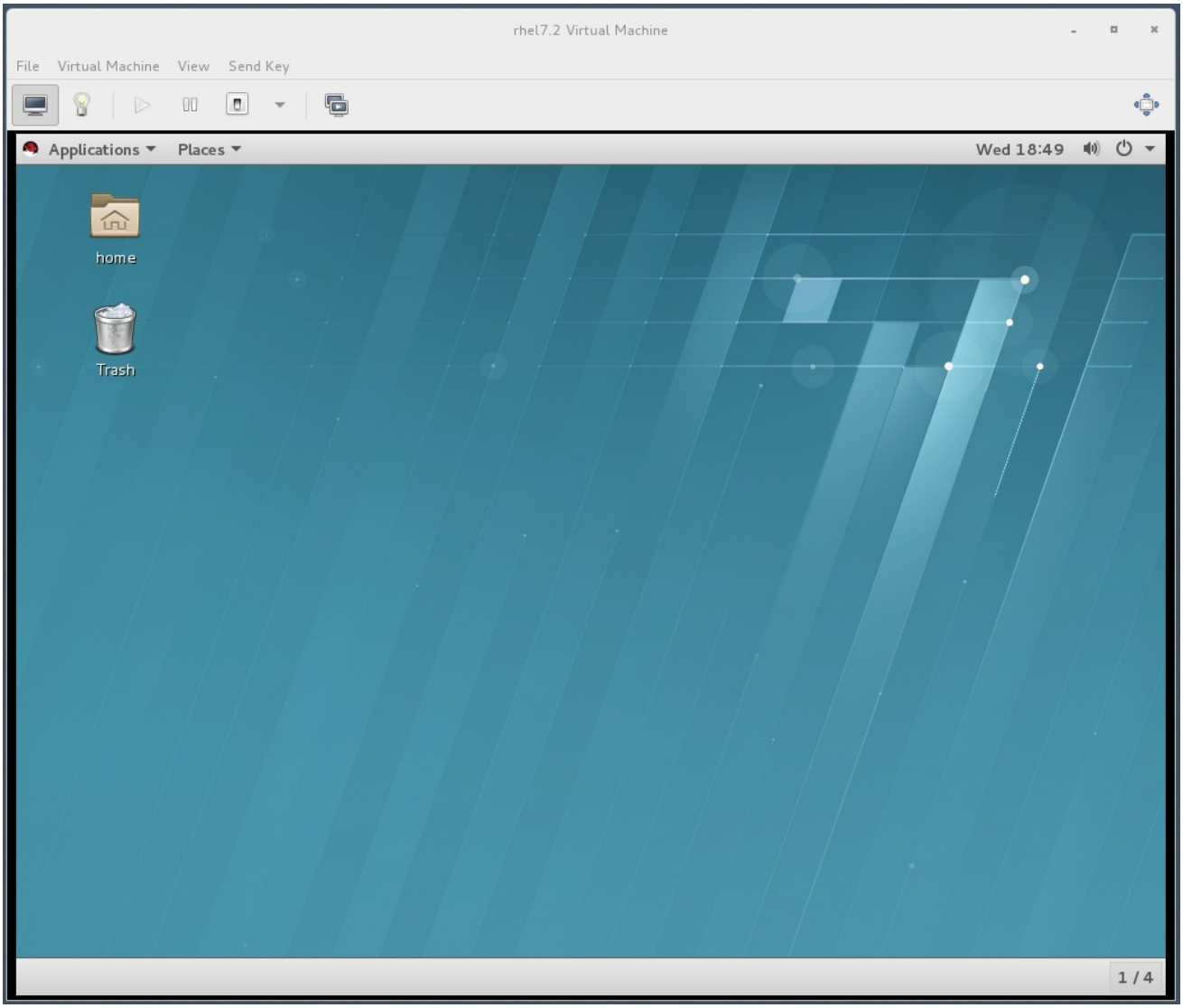


Figure 19.9. Graphical console window

NOTE

VNC is considered insecure by many security experts, however, several changes have been made to enable the secure usage of VNC for virtualization on Red Hat enterprise Linux. The guest machines only listen to the local host's loopback address (**`127.0.0.1`**). This ensures only those with shell privileges on the host can access virt-manager and the virtual machine through VNC. Although virt-manager is configured to listen to other public network interfaces and alternative methods can be configured, it is not recommended.

Remote administration can be performed by tunneling over SSH which encrypts the traffic. Although VNC can be configured to access remotely without tunneling over SSH, for security reasons, it is not recommended. To remotely administer the guest follow the instructions in: Chapter 18, *Remote Management of Guests*. TLS can provide enterprise level security for managing guest and host systems.

Your local desktop can intercept key combinations (for example, Ctrl+Alt+F1) to prevent them from being sent to the guest machine. You can use the **Send key** menu option to send these sequences. From the guest machine window, click the **Send key** menu and select the key sequence to send. In addition, from this menu you can also capture the screen output.

SPICE is an alternative to VNC available for Red Hat Enterprise Linux.

19.5. ADDING A REMOTE CONNECTION

This procedure covers how to set up a connection to a remote system using **virt-manager**.

1. To create a new connection open the **File** menu and select the **Add Connection** menu item.

2. The **Add Connection** wizard appears. Select the hypervisor. For Red Hat Enterprise Linux 7, systems select **QEMU/KVM**. Select **Local** for the local system or one of the remote connection options and click **Connect**. This example uses Remote tunnel over SSH, which works on default installations. For more information on configuring remote connections, see Chapter 18, *Remote Management of Guests*

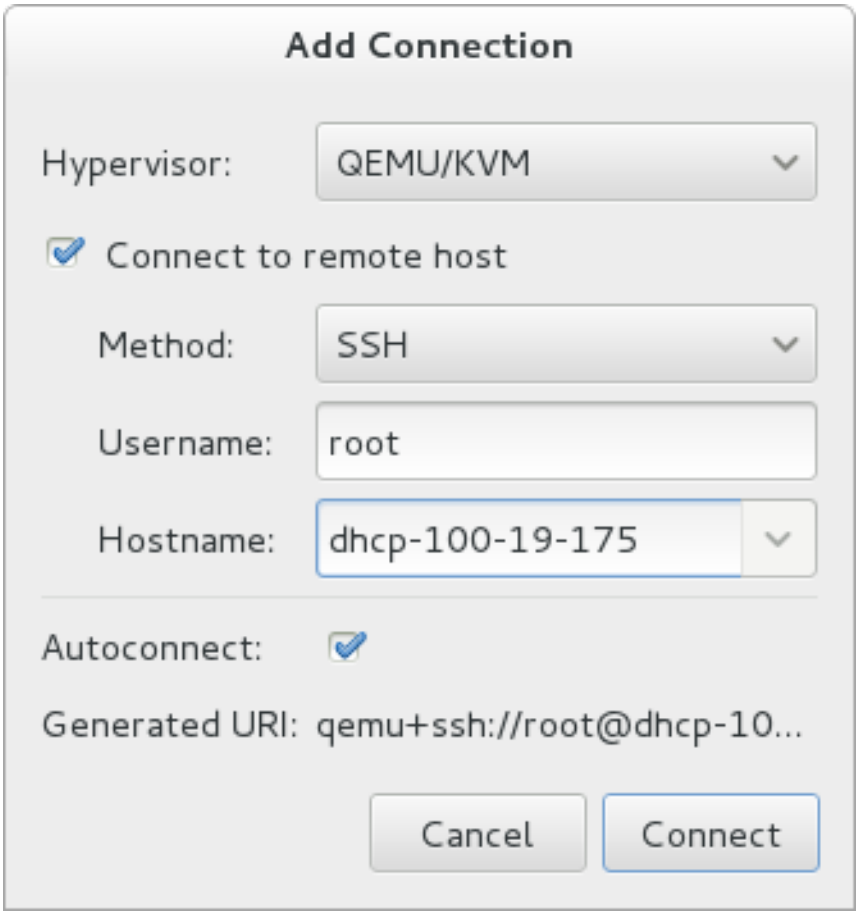


Figure 19.10. Add Connection

3. Enter the root password for the selected host when prompted.

A remote host is now connected and appears in the main **virt-manager** window.

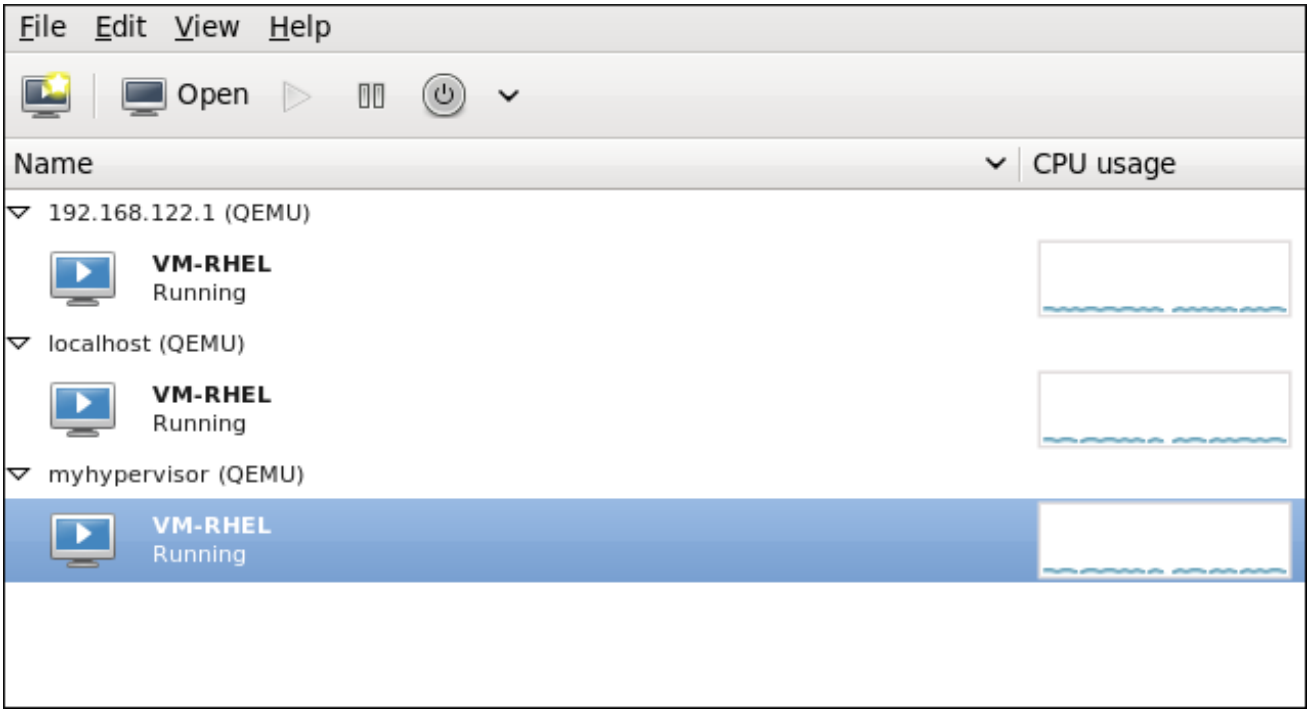


Figure 19.11. Remote host in the main virt-manager window

19.6. DISPLAYING GUEST DETAILS

You can use the Virtual Machine Monitor to view activity information for any virtual machines on your system.

To view a virtual system's details:

1. In the Virtual Machine Manager main window, highlight the virtual machine that you want to view.

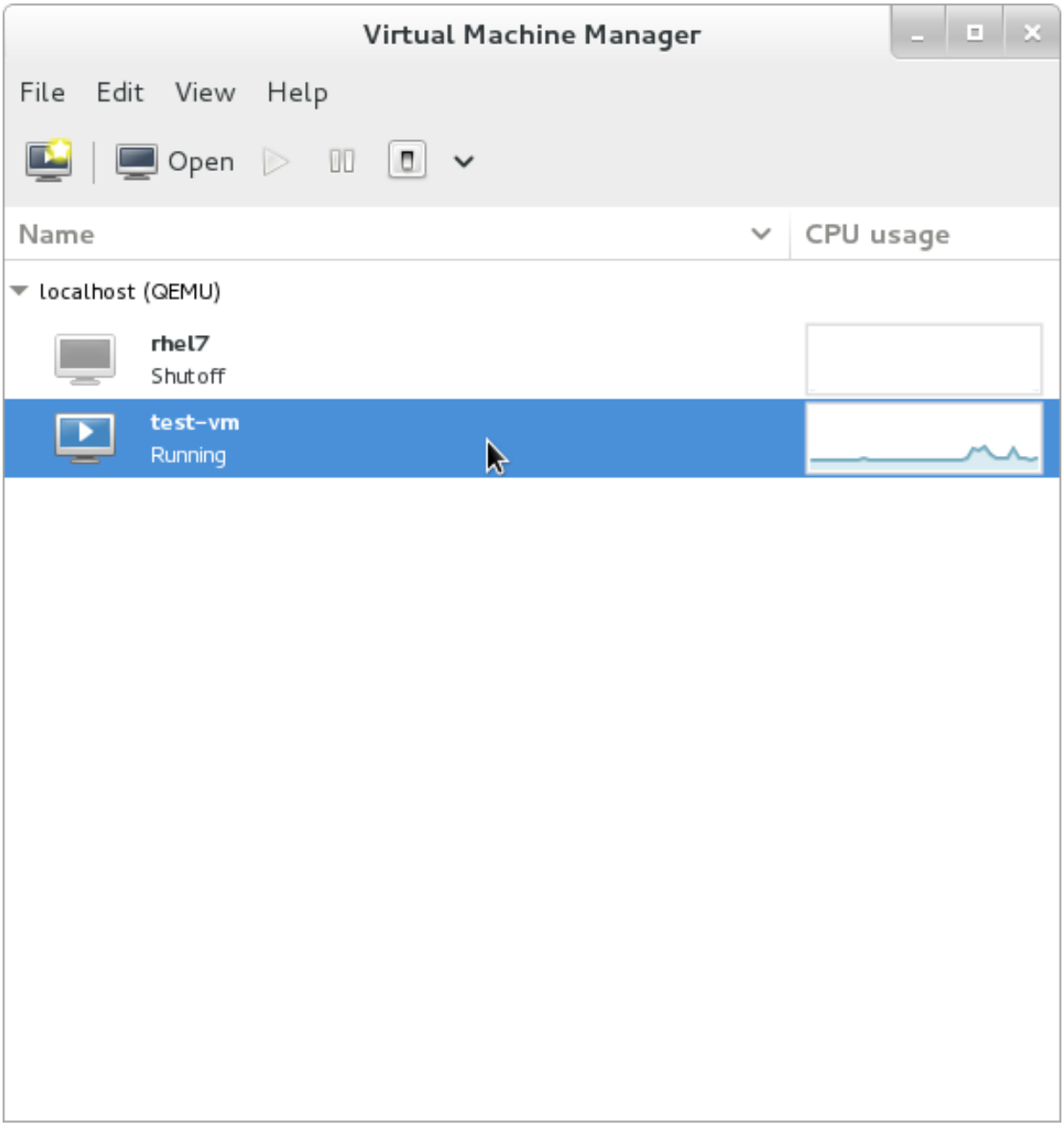


Figure 19.12. Selecting a virtual machine to display

2. From the Virtual Machine Manager **Edit** menu, select **Virtual Machine Details**.

   When the Virtual Machine details window opens, there may be a console displayed. Should this happen, click **View** and then select **Details**. The Overview window opens first by default. To go back to this window, select **Overview** from the navigation pane on the left-hand side.

   The **Overview** view shows a summary of configuration details for the guest.

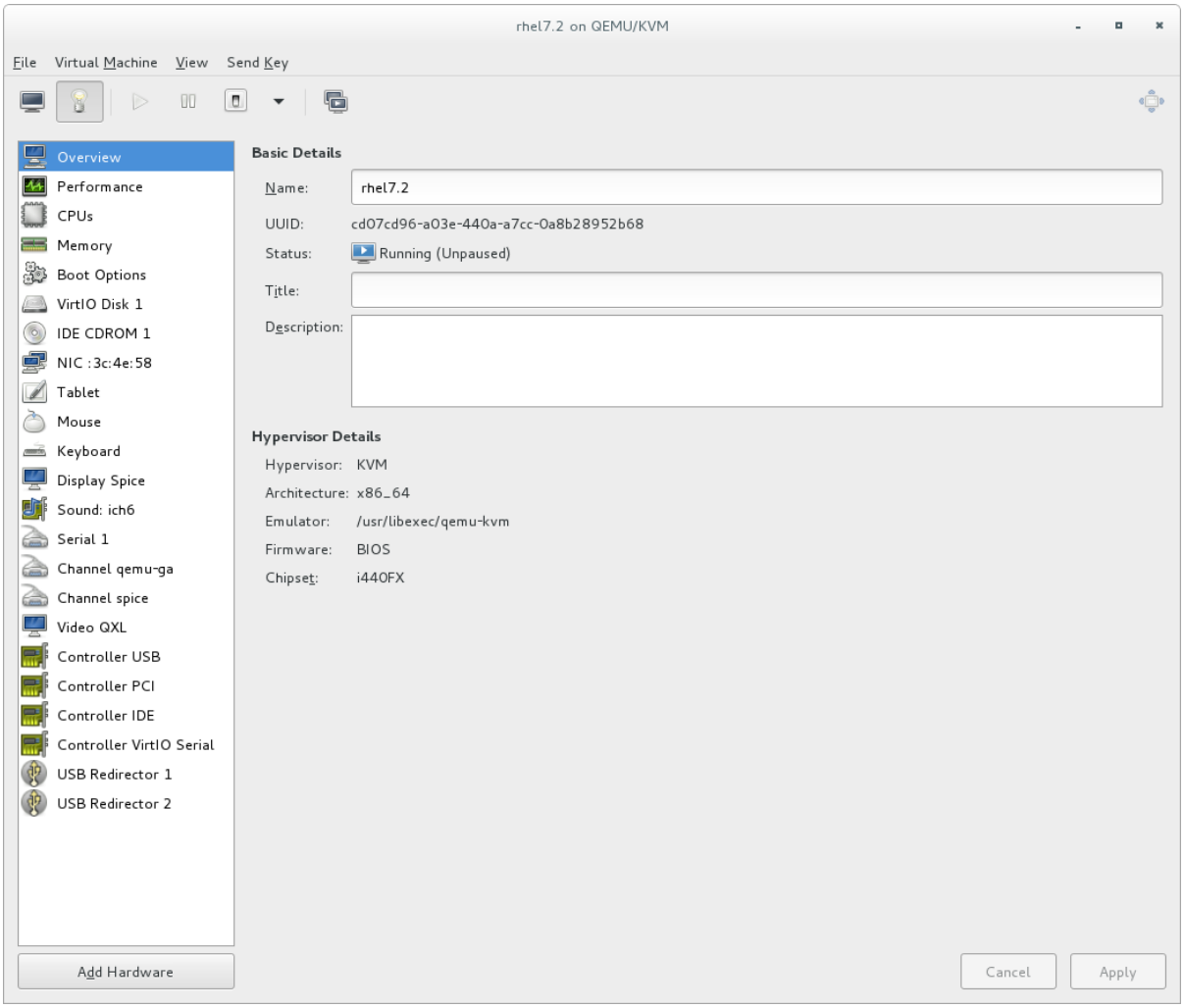
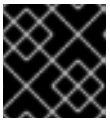


Figure 19.13. Displaying guest details overview

3. Select **CPUs** from the navigation pane on the left-hand side. The**CPUs** view allows you to view or change the current processor allocation.

   It is also possible to increase the number of virtual CPUs (vCPUs) while the virtual machine is running, which is referred to as *hot plugging*.



IMPORTANT

Hot unplugging vCPUs is not supported in Red Hat Enterprise Linux 7.

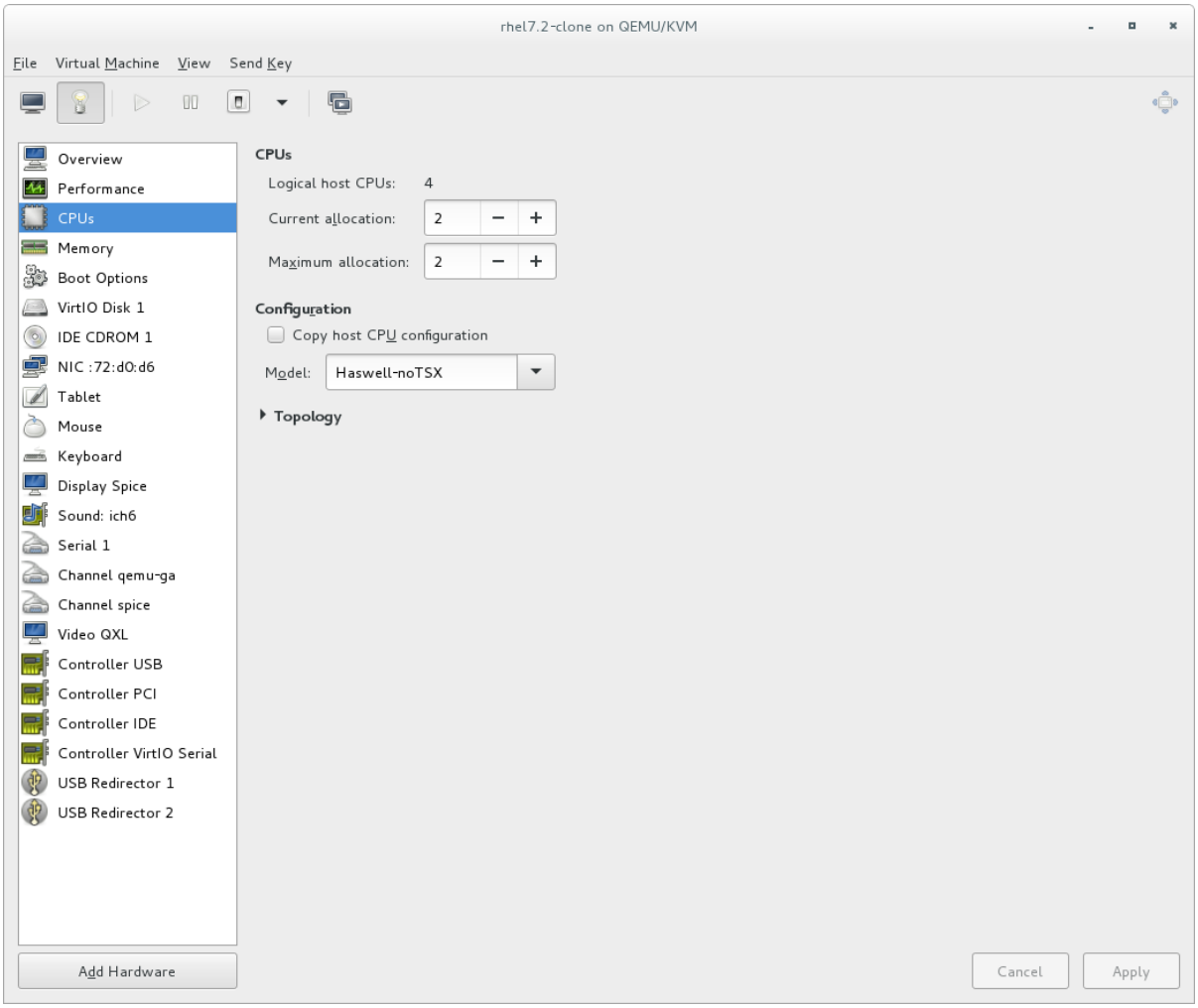


Figure 19.14. Processor allocation panel

4. Select **Memory** from the navigation pane on the left-hand side. The**Memory** view allows you to view or change the current memory allocation.

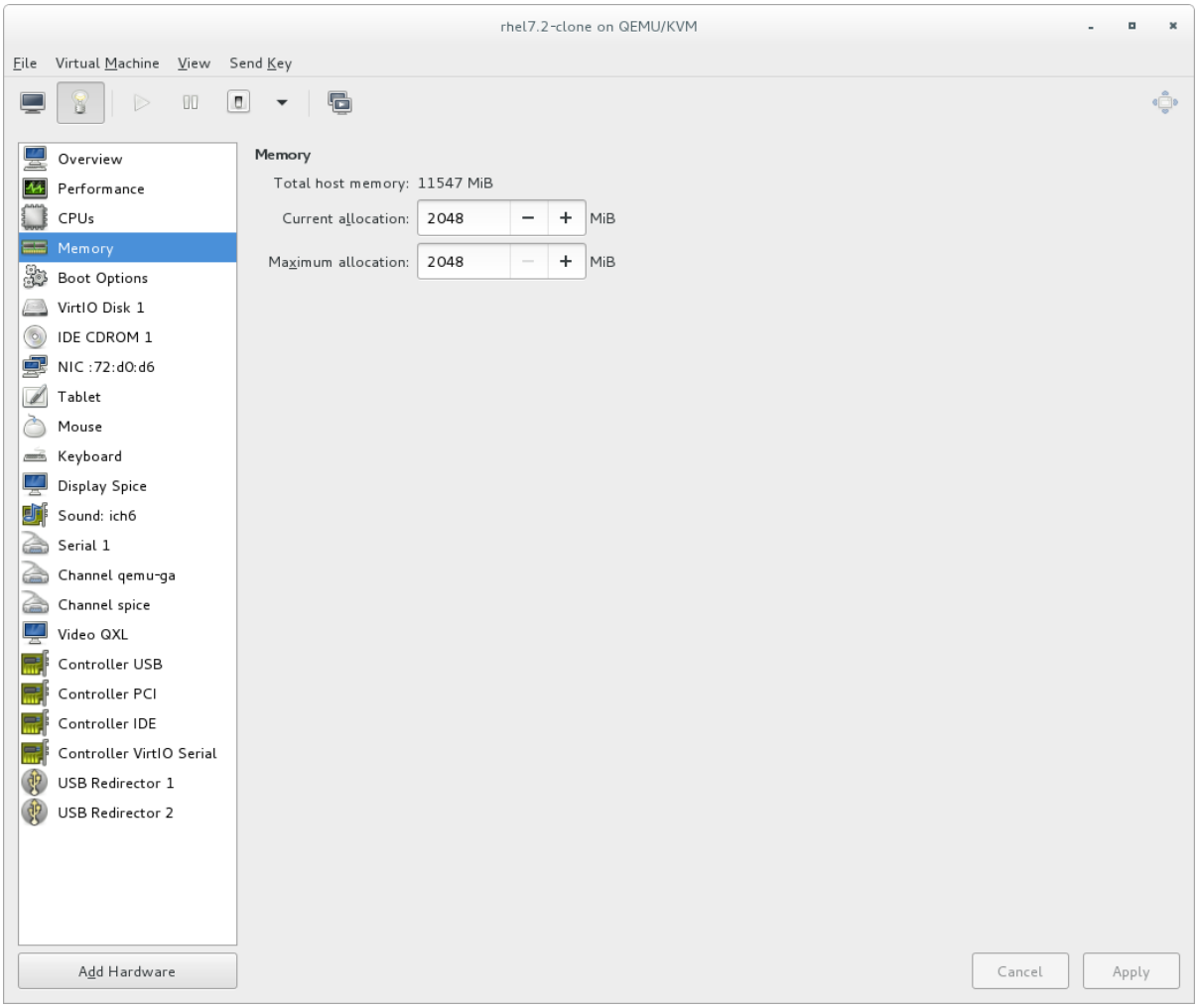


Figure 19.15. Displaying memory allocation

5. Select **Boot Options** from the navigation pane on the left-hand side. The**Boot Options** view allows you to view or change the boot options including whether or not the virtual machine starts when the host boots and the boot device order for the virtual machine.

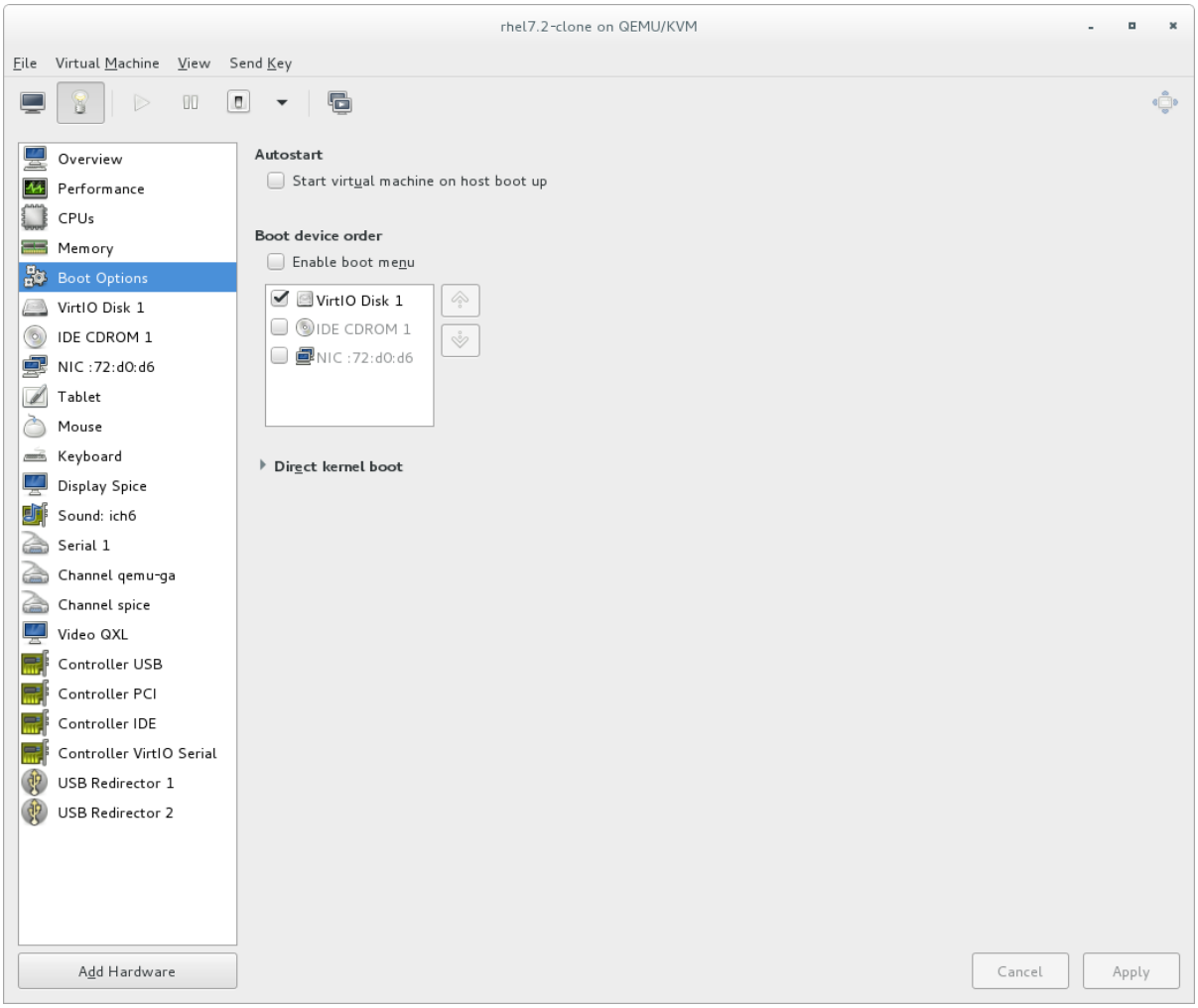


Figure 19.16. Displaying boot options

6. Each virtual disk attached to the virtual machine is displayed in the navigation pane. click a virtual disk to modify or remove it.

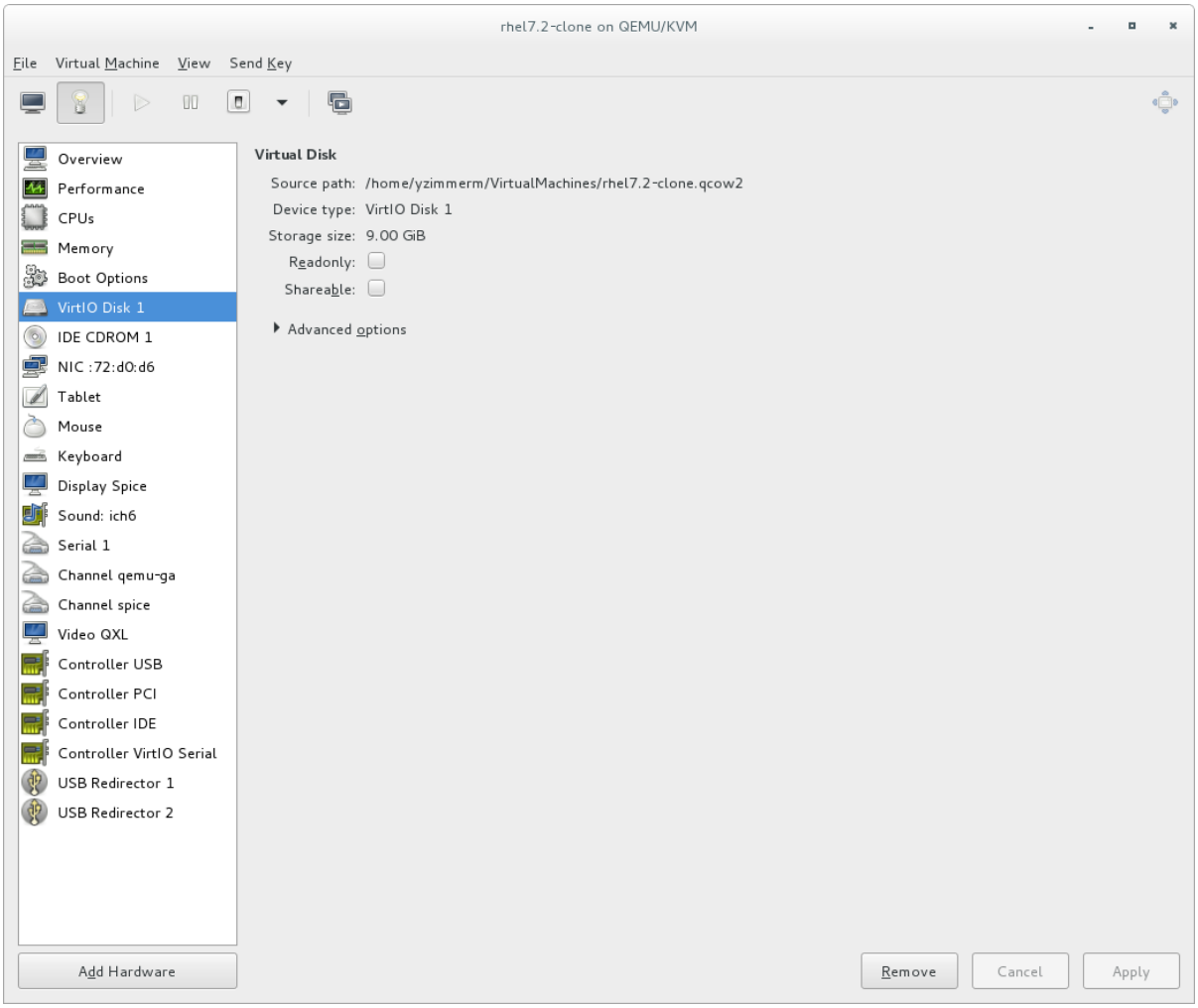


Figure 19.17. Displaying disk configuration

7. Each virtual network interface attached to the virtual machine is displayed in the navigation pane. click a virtual network interface to modify or remove it.

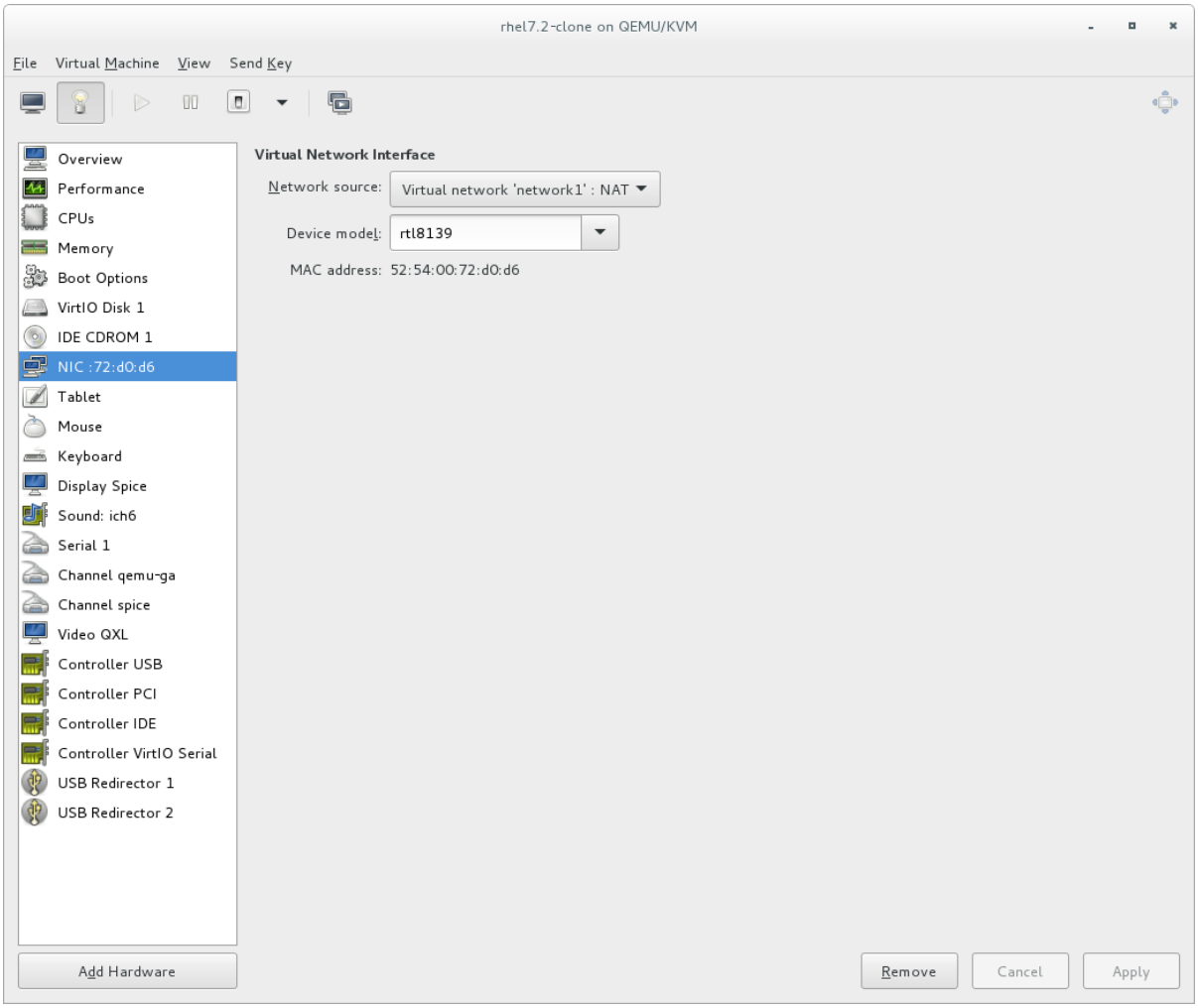


Figure 19.18. Displaying network configuration

19.7. MANAGING SNAPSHOTS

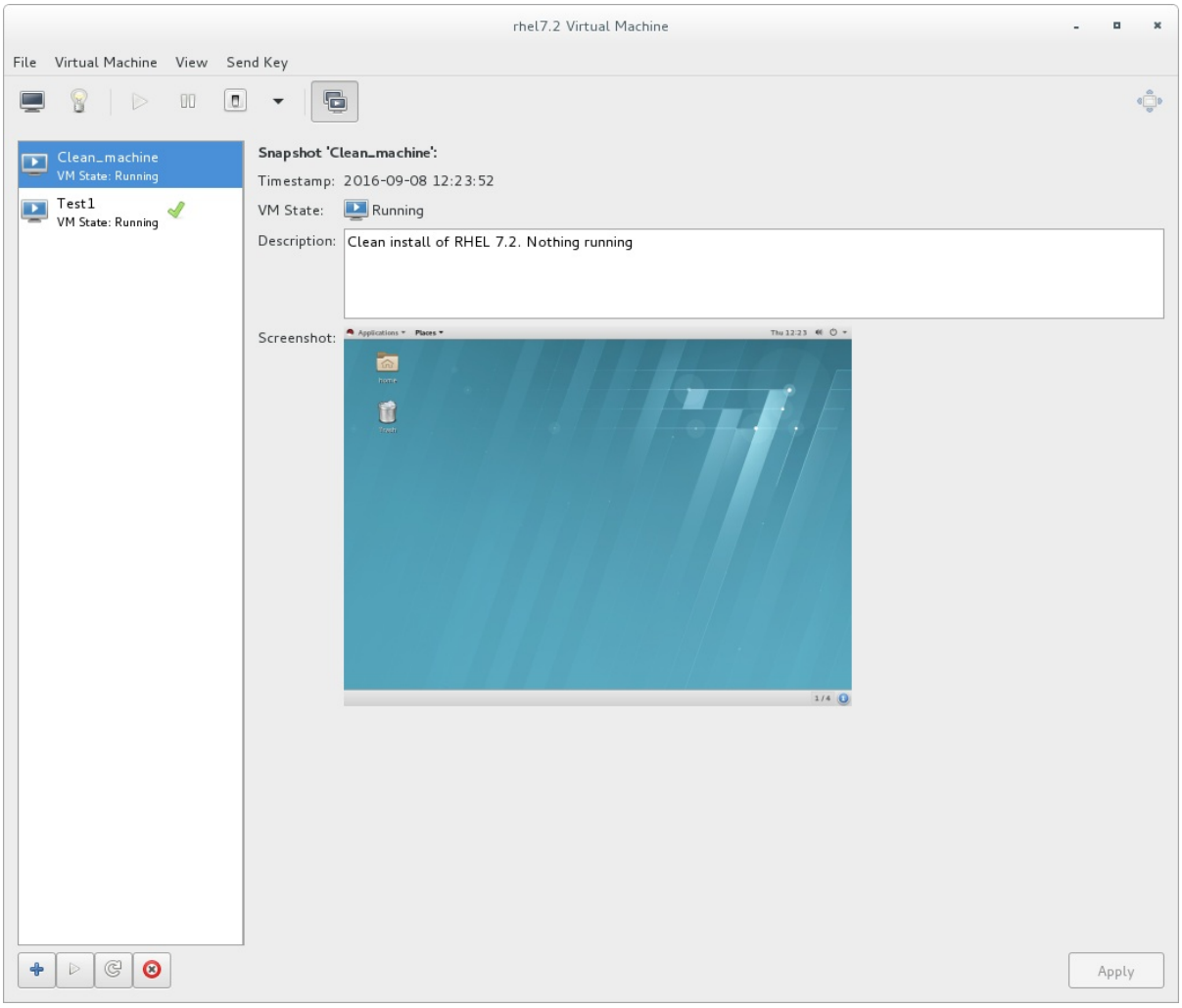
Using **virt-manager**, it is possible to create, run, and delete guest *snapshots*. A snapshot is a saved image of the guest's hard disk, memory, and device state at a single point in time. After a snapshot is created, the guest can be returned to the snapshot's configuration at any time.

IMPORTANT

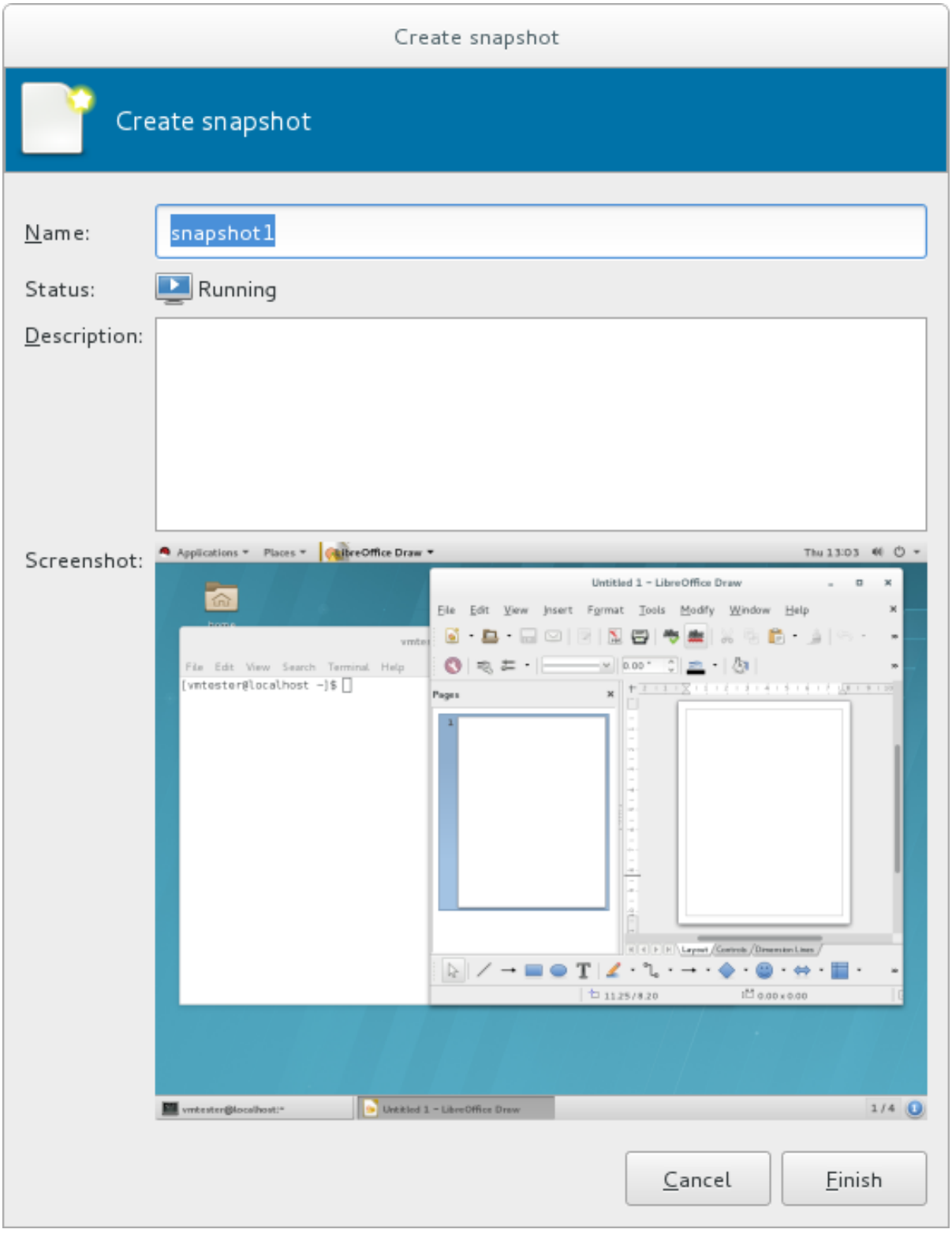
Red Hat recommends the use of external snapshots, as they are more flexible and reliable when handled by other virtualization tools. However, it is currently not possible to create external snapshots in **virt-manager**.

To create external snapshots, use the **`virsh snapshot-create-as`** command with the **`--diskspec vda,snapshot=external`** option. For more information, see Section A.13, “Workaround for Creating External Snapshots with libvirt”.

- To manage snapshots in virt-manager, open the snapshot management interface by clicking on the guest console.



- To create a new snapshot, click under the snapshot list. In the snapshot creation interface, input the name of the snapshot and, optionally, a description, and click **Finish**.



- To revert the guest to a snapshot's configuration, select the snapshot and click
- To remove the selected snapshot, click

WARNING

Creating and loading snapshots while the virtual machine is running (also referred to as *live snapshots*) is only supported with qcow2 disk images.

For more in-depth snapshot management, use the **`virsh snapshot-create`** command. See Section 20.39, “Managing Snapshots” for details about managing snapshots with **`virsh`**.

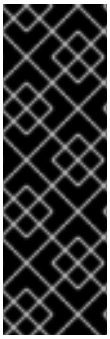
CHAPTER 20. MANAGING GUEST VIRTUAL MACHINES WITH VIRSH

virsh is a command-line interface tool for managing guest virtual machines, and works as the primary means of controlling virtualization on Red Hat Enterprise Linux 7. The **virsh** command-line tool is built on the libvirt management API, and can be used to create, deploy, and manage guest virtual machines. The **virsh** utility is ideal for creating virtualization administration scripts, and users without root privileges can use it in read-only mode. The **virsh** package is installed with **yum** as part of the libvirt-client package.

For installation instructions, see Section 2.2.1, “Installing Virtualization Packages Manually”. For a general introduction of virsh, including a practical demonstration, see the Virtualization Getting Started Guide The remaining sections of this chapter cover the **virsh** command set in a logical order based on usage.

NOTE

Note that when using the help or when reading the man pages, the term 'domain' will be used instead of the term guest virtual machine. This is the term used by libvirt. In cases where the screen output is displayed and the word 'domain' is used, it will not be switched to guest or guest virtual machine. In all examples, the guest virtual machine 'guest1' will be used. You should replace this with the name of your guest virtual machine in all cases. When creating a name for a guest virtual machine you should use a short easy to remember integer (0,1,2...), a text string name, or in all cases you can also use the virtual machine's full UUID.



IMPORTANT

It is important to note which user you are using. If you create a guest virtual machine using one user, you will not be able to retrieve information about it using another user. This is especially important when you create a virtual machine in virt-manager. The default user is root in that case unless otherwise specified. Should you have a case where you cannot list the virtual machine using the **virsh list --all** command, it is most likely due to you running the command using a different user than you used to create the virtual machine. See Important for more information.

20.1. GUEST VIRTUAL MACHINE STATES AND TYPES

Several **virsh** commands are affected by the state of the guest virtual machine:

- *Transient* - A transient guest does not survive reboot.
- *Persistent* - A persistent guest virtual machine survives reboot and lasts until it is deleted.

During the life cycle of a virtual machine, libvirt will classify the guest as any of the following states:

- Undefined - This is a guest virtual machine that has not been defined or created. As such, libvirt is unaware of any guest in this state and will not report about guest virtual machines in this state.
- Shut off - This is a guest virtual machine which is defined, but is not running. Only persistent guests can be considered shut off. As such, when a transient guest virtual machine is put into this state, it ceases to exist.

- Running - The guest virtual machine in this state has been defined and is currently working. This state can be used with both persistent and transient guest virtual machines.
- Paused - The guest virtual machine's execution on the hypervisor has been suspended, or its state has been temporarily stored until it is resumed. Guest virtual machines in this state are not aware they have been suspended and do not notice that time has passed when they are resumed.
- Saved - This state is similar to the paused state, however the guest virtual machine's configuration is saved to persistent storage. Any guest virtual machine in this state is not aware it is paused and does not notice that time has passed once it has been restored.

20.2. DISPLAYING THE VIRSH VERSION

The **virsh version** command displays the current libvirt version and displays information about the local virsh client. For example:

```
$ virsh version
Compiled against library: libvirt 1.2.8
Using library: libvirt 1.2.8
Using API: QEMU 1.2.8
Running hypervisor: QEMU 1.5.3
```

The **virsh version --daemon** is useful for getting information about the **libvirtd** version and package information, including information about the libvirt daemon that is running on the host.

```
$ virsh version --daemon
Compiled against library: libvirt 1.2.8
Using library: libvirt 1.2.8
Using API: QEMU 1.2.8
Running hypervisor: QEMU 1.5.3
Running against daemon: 1.2.8
```

20.3. SENDING COMMANDS WITH ECHO

The **virsh echo [--shell][--xml]** ***arguments*** command displays the specified argument in the specified format. The formats you can use are **--shell** and **--xml**. Each argument queried is displayed separated by a space. Tthe **--shell** option generates output that is formatted in single quotes where needed, so it is suitable for copying and pasting into the bash mode as a command. If the **--xml** argument is used, the output is formatted for use in an XML file, which can then be saved or used for guest's configuration.

20.4. CONNECTING TO THE HYPERVISOR WITH VIRSH CONNECT

The **virsh connect [*hostname-or-URI*] [--readonly]** command begins a local hypervisor session using virsh. After the first time you run this command it will run automatically each time the virsh shell runs. The hypervisor connection URI specifies how to connect to the hypervisor. The most commonly used URIs are:

- **qemu:///system** - connects locally as the root user to the daemon supervising guest virtual machines on the KVM hypervisor.

- **qemu:///session** - connects locally as a user to the user's set of guest local machines using the KVM hypervisor.
- **lxc:///** - connects to a local Linux container.

The command can be run as follows, with the target guest being specified either either by its machine name (hostname) or the URL of the hypervisor (the output of the **virsh uri** command), as shown:

```
$ virsh uri
qemu:///session
```

For example, to establish a session to connect to your set of guest virtual machines, with you as the local user:

```
$ virsh connect qemu:///session
```

To initiate a read-only connection, append the above command with **--readonly**. For more information on URIs, see Remote URIs. If you are unsure of the URI, the **virsh uri** command will display it:

20.5. DISPLAYING INFORMATION ABOUT A GUEST VIRTUAL MACHINE AND THE HYPERVISOR

The **virsh list** command will list guest virtual machines connected to your hypervisor that fit the search parameter requested. The output of the command has 3 columns in a table. Each guest virtual machine is listed with its ID, name, and state.

A wide variety of search parameters is available for **virsh list**. These options are available on the man page, by running **man virsh** or by running the **virsh list --help** command.

NOTE

Note that this if this command only displays guest virtual machines created by the root user. If it does not display a virtual machine you know you have created, it is probable you did not create the virtual machine as root.

Guests created using the virt-manager interface are by default created by root.

Example 20.1. How to list all locally connected virtual machines

The following example lists all the virtual machines your hypervisor is connected to. Note that this command lists both persistent and transient virtual machines.

```
#  virsh list --all

Id              Name                        State
-----------------------------------------------
8  guest1   running
22  guest2   paused
35  guest3   shut off
38              guest4   shut off
```

Example 20.2. How to list the inactive guest virtual machines

The following example lists guests that are currently inactive, or not running. Note that the list only contains persistent virtual machines.

```
#  virsh list --inactive

Id              Name                          State
----------------------------------------------------
35  guest3   shut off
38  guest4   shut off
```

In addition, the following commands can also be used to display basic information about the hypervisor:

- # **virsh hostname** - displays the hypervisor's host name, for example:

  ```
  # virsh hostname
  dhcp-2-157.eus.myhost.com
  ```

- # **virsh sysinfo** - displays the XML representation of the hypervisor's system information, if available, for example:

  ```
  # virsh sysinfo
  <sysinfo type='smbios'>
    <bios>
      <entry name='vendor'>LENOVO</entry>
      <entry name='version'>GJET71WW (2.21 )</entry>
  [...]
  ```

20.6. STARTING, RESUMING, AND RESTORING A VIRTUAL MACHINE

20.6.1. Starting a Guest Virtual Machine

The **virsh start *domain*; [--console] [--paused] [--autodestroy] [--bypass-cache] [--force-boot]** command starts an inactive virtual machine that was already defined but whose state is inactive since its last managed save state or a fresh boot. By default, if the domain was saved by the **virsh managedsave** command, the domain will be restored to its previous state. Otherwise, it will be freshly booted. The command can take the following arguments and the name of the virtual machine is required.

- **--console** - will attach the terminal running **virsh** to the domain's console device. This is runlevel 3.
- **--paused** - if this is supported by the driver, it will start the guest virtual machine in a paused state
- **--autodestroy** - the guest virtual machine is automatically destroyed when virsh disconnects
- **--bypass-cache** - used if the guest virtual machine is in the **managedsave**
- **--force-boot** - discards any **managedsave** options and causes a fresh boot to occur

Example 20.3. How to start a virtual machine

The following example starts the *guest1* virtual machine that you already created and is currently in the inactive state. In addition, the command attaches the guest's console to the terminal running virsh:

```
# virsh start guest1 --console
Domain guest1 started
Connected to domain guest1
Escape character is ^]
```

20.6.2. Configuring a Virtual Machine to be Started Automatically at Boot

The **virsh autostart [--disable]** ***domain*** command will automatically start the guest virtual machine when the host machine boots. Adding the **--disable** argument to this command disables autostart. The guest in this case will not start automatically when the host physical machine boots.

Example 20.4. How to make a virtual machine start automatically when the host physical machine starts

The following example sets the *guest1* virtual machine which you already created to autostart when the host boots:

```
# virsh autostart guest1
```

20.6.3. Rebooting a Guest Virtual Machine

Reboot a guest virtual machine using the **virsh reboot** ***domain*** **[--mode** ***modename*****]** command. Remember that this action will only return once it has executed the reboot, so there may be a time lapse from that point until the guest virtual machine actually reboots. You can control the behavior of the rebooting guest virtual machine by modifying the **on_reboot** element in the guest virtual machine's XML configuration file. By default, the hypervisor attempts to select a suitable shutdown method automatically. To specify an alternative method, the **--mode** argument can specify a comma separated list which includes **initctl**, **acpi**, **agent**, **signal**. The order in which drivers will try each mode is undefined, and not related to the order specified in virsh. For strict control over ordering, use a single mode at a time and repeat the command.

Example 20.5. How to reboot a guest virtual machine

The following example reboots a guest virtual machine named *guest1*. In this example, the reboot uses the **initctl** method, but you can choose any mode that suits your needs.

```
# virsh reboot guest1 --mode initctl
```

20.6.4. Restoring a Guest Virtual Machine

The **virsh restore <file> [--bypass-cache] [--xml** ***/path/to/file*****] [--running] [--paused]** command restores a guest virtual machine previously saved with the **virsh save** command. See Section 20.7.1, “Saving a Guest Virtual Machine's Configuration” for information on the

virsh save command. The restore action restarts the saved guest virtual machine, which may take some time. The guest virtual machine's name and UUID are preserved, but the ID will not necessarily match the ID that the virtual machine had when it was saved.

The **virsh restore** command can take the following arguments:

- **--bypass-cache** - causes the restore to avoid the file system cache but note that using this flag may slow down the restore operation.
- **--xml** - this argument must be used with an XML file name. Although this argument is usually omitted, it can be used to supply an alternative XML file for use on a restored guest virtual machine with changes only in the host-specific portions of the domain XML. For example, it can be used to account for the file naming differences in underlying storage due to disk snapshots taken after the guest was saved.
- **--running** - overrides the state recorded in the save image to start the guest virtual machine as running.
- **--paused** - overrides the state recorded in the save image to start the guest virtual machine as paused.

Example 20.6. How to restore a guest virtual machine

The following example restores the guest virtual machine and its running configuration file *guest1-config.xml*:

```
# virsh restore guest1-config.xml --running
```

20.6.5. Resuming a Guest Virtual Machine

The **virsh resume *domain*** command restarts the CPUs of a domain that was suspended. This operation is immediate. The guest virtual machine resumes execution from the point it was suspended. Note that this action will not resume a guest virtual machine that has been undefined. This action will not resume transient virtual machines and will only work on persistent virtual machines.

Example 20.7. How to restore a suspended guest virtual machine

The following example restores the *guest1* virtual machine:

```
# virsh resume guest1
```

20.7. MANAGING A VIRTUAL MACHINE CONFIGURATION

This section provides information about managing a virtual machine configuration.

20.7.1. Saving a Guest Virtual Machine's Configuration

The **virsh save [--bypass-cache] *domain* file [--xml *string*] [--running] [--paused] [--verbose]** command stops the specified domain, saving the current state of the guest virtual machine's system memory to a specified file. This may take a considerable amount of time,

depending on the amount of memory in use by the guest virtual machine. You can restore the state of the guest virtual machine with the **virsh restore** (Section 20.6.4, “Restoring a Guest Virtual Machine”) command.

The difference between the **virsh save** command and the **virsh suspend** command, is that the **virsh suspend** stops the domain CPUs, but leaves the domain's **qemu** process running and its memory image resident in the host system. This memory image will be lost if the host system is rebooted.

The **virsh save** command stores the state of the domain on the hard disk of the host system and terminates the **qemu** process. This enables restarting the domain from the saved state.

You can monitor the process of **virsh save** with the **virsh domjobinfo** command and cancel it with the **virsh domjobabort** command.

The **virsh save** command can take the following arguments:

- **--bypass-cache** - causes the restore to avoid the file system cache but note that using this flag may slow down the restore operation.
- **--xml** - this argument must be used with an XML file name. Although this argument is usually omitted, it can be used to supply an alternative XML file for use on a restored guest virtual machine with changes only in the host-specific portions of the domain XML. For example, it can be used to account for the file naming differences in underlying storage due to disk snapshots taken after the guest was saved.
- **--running** - overrides the state recorded in the save image to start the guest virtual machine as running.
- **--paused** - overrides the state recorded in the save image to start the guest virtual machine as paused.
- **--verbose** - displays the progress of the save.

Example 20.8. How to save a guest virtual machine running configuration

The following example saves the *guest1* virtual machine's running configuration to the **guest1-config.xml** file:

```
# virsh save guest1 guest1-config.xml --running
```

20.7.2. Defining a Guest Virtual Machine with an XML File

The **virsh define *filename*** command defines a guest virtual machine from an XML file. The guest virtual machine definition in this case is registered but not started. If the guest virtual machine is already running, the changes the changes will take effect once the domain is shut down and started again.

Example 20.9. How to create a guest virtual machine from an XML file

The following example creates a virtual machine from the pre-existing *guest1-config.xml* XML file, which contains the configuration for the virtual machine:

```
# virsh define guest1-config.xml
```

20.7.3. Updating the XML File That will be Used for Restoring a Guest Virtual Machine

NOTE

This command should only be used to recover from a situation where the guest virtual machine does not run properly. It is not meant for general use.

The **virsh save-image-define *filename* [--xml */path/to/file*] [--running] [--paused]** command updates the guest virtual machine's XML file that will be used when the virtual machine is restored used during the **virsh restore** command. The *--xml* argument must be an XML file name containing the alternative XML elements for the guest virtual machine's XML. For example, it can be used to account for the file naming differences resulting from creating disk snapshots of underlying storage after the guest was saved. The save image records if the guest virtual machine should be restored to a running or paused state. Using the arguments **--running** or **--paused** dictates the state that is to be used.

Example 20.10. How to save the guest virtual machine's running configuration

The following example updates the *guest1-config.xml* configuration file with the state of the corresponding running guest:

```
# virsh save-image-define guest1-config.xml --running
```

20.7.4. Extracting the Guest Virtual Machine XML File

NOTE

This command should only be used to recover from a situation where the guest virtual machine does not run properly. It is not meant for general use.

The **virsh save-image-dumpxml *file* --security-info** command will extract the guest virtual machine XML file that was in effect at the time the saved state file (used in the **virsh save** command) was referenced. Using the **--security-info** argument includes security sensitive information in the file.

Example 20.11. How to pull the XML configuration from the last save

The following example triggers a dump of the configuration file that was created the last time the guest virtual machine was saved. In this example, the resulting dump file is named *guest1-config-xml*:

```
# virsh save-image-dumpxml guest1-config.xml
```

20.7.5. Editing the Guest Virtual Machine Configuration

NOTE

This command should only be used to recover from a situation where the guest virtual machine does not run properly. It is not meant for general use.

The **virsh save-image-edit <file> [--running] [--paused]** command edits the XML configuration file that was created by the **virsh save** command. See Section 20.7.1, “Saving a Guest Virtual Machine's Configuration” for information on the **virsh save** command.

When the guest virtual machine is saved, the resulting image file will indicate if the virtual machine should be restored to a **--running** or **--paused** state. Without using these arguments in the **save-image-edit** command, the state is determined by the image file itself. By selecting **--running** (to select the running state) or **--paused** (to select the paused state) you can overwrite the state that **virsh restore** should use.

Example 20.12. How to edit a guest virtual machine's configuration and restore the machine to running state

The following example opens a guest virtual machine's configuration file, named *guest1-config.xml*, for editing in your default editor. When the edits are saved, the virtual machine boots with the new settings:

```
# virsh save-image-edit guest1-config.xml --running
```

20.8. SHUTTING OFF, SHUTTING DOWN, REBOOTING, AND FORCING A SHUTDOWN OF A GUEST VIRTUAL MACHINE

20.8.1. Shutting down a Guest Virtual Machine

The **virsh shutdown *domain* [--mode *modename*]** command shuts down a guest virtual machine. You can control the behavior of how the guest virtual machine reboots by modifying the **on_shutdown** parameter in the guest virtual machine's configuration file. Any change to the **on_shutdown** parameter will only take effect after the domain has been shutdown and restarted.

The **virsh shutdown** command command can take the following optional argument:

- **--mode** chooses the shutdown mode. This can be either **acpi**, **agent**, **initctl**, **signal**, or **paravirt**.

Example 20.13. How to shutdown a guest virtual machine

The following example shuts down the *guest1* virtual machine using the **acpi** mode:

```
# virsh shutdown guest1 --mode acpi
Domain guest1 is being shutdown
```

20.8.2. Suspending a Guest Virtual Machine

The **virsh suspend *domain*** command suspends a guest virtual machine.

When a guest virtual machine is in a suspended state, it consumes system RAM but not processor resources. Disk and network I/O does not occur while the guest virtual machine is suspended. This operation is immediate and the guest virtual machine can only be restarted with the **virsh resume** command. Running this command on a transient virtual machine will delete it.

Example 20.14. How to suspend a guest virtual machine

The following example suspends the *guest1* virtual machine:

```
# virsh suspend guest1
```

20.8.3. Resetting a Virtual Machine

The **virsh reset *domain*** resets the guest virtual machine immediately without any guest shutdown. A reset emulates the reset button on a machine, where all guest hardware sees the RST line and re-initializes the internal state. Note that without any guest virtual machine OS shutdown, there are risks for data loss.

NOTE

Resetting a virtual machine does not apply any pending domain configuration changes. Changes to the domain's configuration only take effect after a complete shutdown and restart of the domain.

Example 20.15. How to reset a guest virtual machine

The following example resets the *guest1* virtual machines:

```
# virsh reset guest1
```

20.8.4. Stopping a Running Guest Virtual Machine in Order to Restart It Later

The **virsh managedsave *domain* --bypass-cache --running | --paused | --verbose** command saves and destroys (stops) a running guest virtual machine so that it can be restarted from the same state at a later time. When used with a **virsh start** command it is automatically started from this save point. If it is used with the **--bypass-cache** argument the save will avoid the filesystem cache. Note that this option may slow down the save process speed and using the **--verbose** option displays the progress of the dump process. Under normal conditions, the managed save will decide between using the running or paused state as determined by the state the guest virtual machine is in when the save is done. However, this can be overridden by using the **--running** option to indicate that it must be left in a running state or by using **--paused** option which indicates it is to be left in a paused state. To remove the managed save state, use the **virsh managedsave-remove** command which will force the guest virtual machine to do a full boot the next time it is started. Note that the entire managed save process can be monitored using the **domjobinfo** command and can also be canceled using the **domjobabort** command.

Example 20.16. How to stop a running guest and save its configuration

The following example stops the *guest1* virtual machine and saves its running configuration setting so that you can restart it:

```
# virsh managedsave guest1 --running
```

20.9. REMOVING AND DELETING A VIRTUAL MACHINE

20.9.1. Undefining a Virtual Machine

The **virsh undefine *domain* [--managed-save] [*storage*] [--remove-all-storage] [--wipe-storage] [--snapshots-metadata] [--nvram]** command undefines a domain. If domain is inactive, the configuration is removed completely. If the domain is active (running), it is converted to a transient domain. When the guest virtual machine becomes inactive, the configuration is removed completely.

This command can take the following arguments:

- **--managed-save** - this argument guarantees that any managed save image is also cleaned up. Without using this argument, attempts to undefine a guest virtual machine with a managed save will fail.
- **--snapshots-metadata** - this argument guarantees that any snapshots (as shown with **snapshot-list**) are also cleaned up when undefining an inactive guest virtual machine. Note that any attempts to undefine an inactive guest virtual machine with snapshot metadata will fail. If this argument is used and the guest virtual machine is active, it is ignored.
- **--storage** - using this argument requires a comma separated list of volume target names or source paths of storage volumes to be removed along with the undefined domain. This action will undefine the storage volume before it is removed. Note that this can only be done with inactive guest virtual machines and that this will only work with storage volumes that are managed by libvirt.
- **--remove-all-storage** - in addition to undefining the guest virtual machine, all associated storage volumes are deleted. If you want to delete the virtual machine, choose this option only if there are no other virtual machines using the same associated storage. An alternative way is with the **virsh vol-delete**. See Section 20.31, “Deleting Storage Volumes” for more information.
- **--wipe-storage** - in addition to deleting the storage volume, the contents are wiped.

Example 20.17. How to delete a guest virtual machine and delete its storage volumes

The following example undefines the *guest1* virtual machine and remove all associated storage volumes. An undefined guest becomes transient and thus is deleted after it shuts down:

```
# virsh undefine guest1 --remove-all-storage
```

20.9.2. Forcing a Guest Virtual Machine to Stop

NOTE

This command should only be used when you cannot shut down the virtual guest machine by any other method.

The **virsh destroy** command initiates an immediate ungraceful shutdown and stops the specified guest virtual machine. Using **virsh destroy** can corrupt guest virtual machine file systems. Use the **virsh destroy** command only when the guest virtual machine is unresponsive. The **virsh destroy** command with the **--graceful** option attempts to flush the cache for the disk image file before powering off the virtual machine.

Example 20.18. How to immediately shutdown a guest virtual machine with a hard shutdown

The following example immediately shuts down the *guest1* virtual machine, probably because it is unresponsive:

```
# virsh destroy guest1
```

You may want to follow this with the **virsh undefine** command. See Example 20.17, “How to delete a guest virtual machine and delete its storage volumes”

20.10. CONNECTING THE SERIAL CONSOLE FOR THE GUEST VIRTUAL MACHINE

The **virsh console *domain* [--devname *devicename*] [--force] [--safe]** command connects the virtual serial console for the guest virtual machine. This is very useful for example for guests that do not provide VNC or SPICE protocols (and thus does not offer video display for GUI tools) and that do not have network connection (and thus cannot be interacted with using SSH).

The optional **--devname** parameter refers to the device alias of an alternate console, serial, or parallel device configured for the guest virtual machine. If this parameter is omitted, the primary console will be opened. If the **--safe** option is specified, the connection is only attempted if the driver supports safe console handling. This option specifies that the server has to ensure exclusive access to console devices. Optionally, the **force** option may be specified, which requests to disconnect any existing sessions, such as in the case of a broken connection.

Example 20.19. How to start a guest virtual machine in console mode

The following example starts a previously created *guest1* virtual machine so that it connects to the serial console using safe console handling:

```
# virsh console guest1 --safe
```

20.11. INJECTING NON-MASKABLE INTERRUPTS

The **virsh inject-nmi *domain*** injects a non-maskable interrupt (NMI) message to the guest virtual machine. This is used when response time is critical, such as during non-recoverable hardware errors. In addition, **virsh inject-nmi** is useful for triggering a crashdump in Windows guests.

Example 20.20. How to inject an NMI to the guest virtual machine

The following example sends an NMI to the *guest1* virtual machine:

```
# virsh inject-nmi guest1
```

20.12. RETRIEVING INFORMATION ABOUT YOUR VIRTUAL MACHINE

20.12.1. Displaying Device Block Statistics

By default, the **virsh domblkstat** command displays the block statistics for the first block device defined for the domain. To view statistics of other block devices, use the **virsh domblklist *domain*** command to list all block devices, and then select a specific block device and display it by specifying either the *Target* or *Source* name from the **virsh domblklist** command output after the domain name. Note that not every hypervisor can display every field. To make sure that the output is presented in its most legible form use the **--human** argument.

Example 20.21. How to display block statistics for a guest virtual machine

The following example displays the devices that are defined for the *guest1* virtual machine, and then lists the block statistics for that device.

```
# virsh domblklist guest1

Target     Source
------------------------------------------------
vda        /VirtualMachines/guest1.img
hdc        -

# virsh domblkstat guest1 vda --human
Device: vda
 number of read operations:      174670
 number of bytes read:           3219440128
 number of write operations:     23897
 number of bytes written:        164849664
 number of flush operations:     11577
 total duration of reads (ns):   1005410244506
 total duration of writes (ns):  1085306686457
 total duration of flushes (ns): 340645193294
```

20.12.2. Retrieving Network Interface Statistics

The **virsh domifstat *domain interface-device*** command displays the network interface statistics for the specified device running on a given guest virtual machine.

To determine which interface devices are defined for the domain, use the **virsh domiflist** command and use the output in the Interface column.

Example 20.22. How to display networking statistics for a guest virtual machine

The following example obtains the networking interface defined for the *guest1* virtual machine, and then displays the networking statistics on the obtained interface (*macvtap0*):

```
# virsh domiflist guest1
Interface  Type       Source     Model       MAC
-------------------------------------------------------
macvtap0   direct     em1        rtl8139     12:34:00:0f:8a:4a
```

```
# virsh domifstat guest1 macvtap0
macvtap0 rx_bytes 51120
macvtap0 rx_packets 440
macvtap0 rx_errs 0
macvtap0 rx_drop 0
macvtap0 tx_bytes 231666
macvtap0 tx_packets 520
macvtap0 tx_errs 0
macvtap0 tx_drop 0
```

20.12.3. Modifying the Link State of a Guest Virtual Machine's Virtual Interface

The **virsh domif-setlink *domain interface-device state*** command configures the status of the specified interface device link state as either **up** or **down**. To determine which interface devices are defined for the domain, use the **virsh domiflist** command and use either the **Interface** or **MAC** column as the interface device option. By default, **virsh domif-setlink** changes the link state for the running domain. To modify the domain's persistent configuration use the **--config** argument.

Example 20.23. How to enable a guest virtual machine interface

The following example shows determining the interface device of the *rhel7* domain, then setting the link as *down*, and finally as *up*:

```
# virsh domiflist rhel7
Interface  Type       Source     Model       MAC
-------------------------------------------------------
vnet0      network    default    virtio      52:54:00:01:1d:d0

# virsh domif-setlink rhel7 vnet0 down
Device updated successfully

# virsh domif-setlink rhel7 52:54:00:01:1d:d0 up
Device updated successfully
```

20.12.4. Listing the Link State of a Guest Virtual Machine's Virtual Interface

The **virsh domif-getlink *domain interface-device*** command retrieves the specified interface device link state. To determine which interface devices are defined for the domain, use the **virsh domiflist** command and use either the **Interface** or **MAC** column as the interface device option. By default, **virsh domif-getlink** retrieves the link state for the running domain. To retrieve the domain's persistent configuration use the **--config option**.

Example 20.24. How to display the link state of a guest virtual machine's interface

The following example shows determining the interface device of the *rhel7* domain, then determining its state as *up*, then changing the state to *down*, and then verifying the change was successful:

```
# virsh domiflist rhel7
Interface  Type       Source     Model       MAC
-------------------------------------------------------
vnet0      network    default    virtio      52:54:00:01:1d:d0
```

```
# virsh domif-getlink rhel7 52:54:00:01:1d:d0
52:54:00:01:1d:d0 up

# virsh domif-setlink rhel7 vnet0 down
Device updated successfully

# virsh domif-getlink rhel7 vnet0
vnet0 down
```

20.12.5. Setting Network Interface Bandwidth Parameters

The **virsh domiftune *domain interface-device*** command either retrieves or sets the specified domain's interface bandwidth parameters. To determine which interface devices are defined for the domain, use the **virsh domiflist** command and use either the **Interface** or **MAC** column as the interface device option. The following format should be used:

```
# virsh domiftune domain interface [--inbound] [--outbound] [--config] [--
live] [--current]
```

The **--config**, **--live**, and **--current** options are described in Section 20.43, “Setting Schedule Parameters”. If the **--inbound** or the **--outbound** option is not specified, **virsh domiftune** queries the specified network interface and displays the bandwidth settings. By specifying **--inbound** or **--outbound**, or both, and the average, peak, and burst values, **virsh domiftune** sets the bandwidth settings. At minimum the average value is required. In order to clear the bandwidth settings, provide 0 (zero). For a description of the average, peak, and burst values, see Section 20.27.6.2, “Attaching interface devices”.

Example 20.25. How to set the guest virtual machine network interface parameters

The following example sets *eth0* parameters for the guest virtual machine named *guest1*:

```
# virsh domiftune guest1 eth0 outbound --live
```

20.12.6. Retrieving Memory Statistics

The **virsh dommemstat *domain* [<period in seconds>] [--config] [--live] [--current]** command displays the memory statistics for a running guest virtual machine. Using the optional **period** switch requires a time period in seconds. Setting this option to a value larger than 0 will allow the balloon driver to return additional statistics which will be displayed by running subsequent **dommemstat** commands. Setting the **period** option to 0, stops the balloon driver collection but does not clear the statistics already in the balloon driver. You cannot use the **--live**, **--config**, or **--current** options without also setting the **period** option. If the **--live** option is specified, only the guest's running statistics will be collected. If the **--config** option is used, it will collect the statistics for a persistent guest, but only after the next boot. If the **--current** option is used, it will collect the current statistics.

Both the **--live** and **--config** options may be used but **--current** is exclusive. If no flag is specified, the guest's state will dictate the behavior of the statistics collection (running or not).

Example 20.26. How to collect memory statistics for a running guest virtual machine

The following example shows displaying the memory statistics in the *rhel7* domain:

```
# virsh dommemstat rhel7
actual 1048576
swap_in 0
swap_out 0
major_fault 2974
minor_fault 1272454
unused 246020
available 1011248
rss 865172
```

20.12.7. Displaying Errors on Block Devices

The **virsh domblkerror *domain*** command lists all the block devices in the *error* state and the error detected on each of them. This command is best used after a **virsh domstate** command reports that a guest virtual machine is paused due to an I/O error.

Example 20.27. How to display the block device errors for a virtual machine

The following example displays the block device errors for the *guest1* virtual machine:

```
# virsh domblkerror guest1
```

20.12.8. Displaying the Block Device Size

The **virsh domblkinfo *domain*** command lists the capacity, allocation, and physical block sizes for a specific block device in the virtual machine. Use the virsh domblklist command to list all block devices and then choose to display a specific block device by specifying either the *Target* or *Source* name from the **virsh domblklist** output after the domain name.

Example 20.28. How to display the block device size

In this example, you list block devices on the *rhel7* virtual machine, and then display the block size for each of the devices.

```
# virsh domblklist rhel7
Target     Source
------------------------------------------------
vda        /home/vm-images/rhel7-os
vdb        /home/vm-images/rhel7-data

# virsh domblkinfo rhel7 vda
Capacity:       10737418240
Allocation:     8211980288
Physical:       10737418240

# virsh domblkinfo rhel7 /home/vm-images/rhel7-data
Capacity:       104857600
Allocation:     104857600
Physical:       104857600
```

20.12.9. Displaying the Block Devices Associated with a Guest Virtual Machine

The **virsh domblklist *domain* [--inactive] [--details]** command displays a table of all block devices that are associated with the specified guest virtual machine.

If **--inactive** is specified, the result will show the devices that are to be used at the next boot and will not show those that are currently running in use by the running guest virtual machine. If **--details** is specified, the disk type and device value will be included in the table. The information displayed in this table can be used with other commands that require a block-device to be provided, such as **virsh domblkinfo** and **virsh snapshot-create**. The disk *Target* or *Source* contexts can also be used when generating the xmlfile context information for the **virsh snapshot-create** command.

Example 20.29. How to display the block devices that are associated with a virtual machine

The following example displays details about block devices associated with the *rhel7* virtual machine.

```
# virsh domblklist rhel7 --details
Type       Device     Target     Source
-----------------------------------------------
file       disk       vda        /home/vm-images/rhel7-os
file       disk       vdb        /home/vm-images/rhel7-data
```

20.12.10. Displaying Virtual Interfaces Associated with a Guest Virtual Machine

The **virsh domblklist *domain*** command displays a table of all the virtual interfaces that are associated with the specified domain. The **virsh domiflist** command requires the name of the virtual machine (or *domain*), and optionally can take the **--inactive** argument. The latter retrieves the inactive rather than the running configuration, which is retrieved with the default setting. If **--inactive** is specified, the result shows devices that are to be used at the next boot, and does not show devices that are currently in use by the running guest. Virsh commands that require a MAC address of a virtual interface (such as **detach-interface**, **domif-setlink**, **domif-getlink**, **domifstat**, and **domiftune**) accept the output displayed by this command.

Example 20.30. How to display the virtual interfaces associated with a guest virtual machine

The following example displays the virtual interfaces that are associated with the *rhel7* virtual machine, and then displays the network interface statistics for the *vnet0* device.

```
# virsh domiflist rhel7
Interface  Type       Source     Model       MAC
-------------------------------------------------------
vnet0      network    default    virtio      52:54:00:01:1d:d0

# virsh domifstat rhel7 vnet0
vnet0 rx_bytes 55308
vnet0 rx_packets 969
vnet0 rx_errs 0
vnet0 rx_drop 0
vnet0 tx_bytes 14341
```

```
vnet0 tx_packets 148
vnet0 tx_errs 0
vnet0 tx_drop 0
```

20.13. WORKING WITH SNAPSHOTS

20.13.1. Shortening a Backing Chain by Copying the Data

This section demonstrates how to use the **virsh blockcommit *domain* <path> [<bandwidth>] [<base>] [--shallow] [<top>] [--active] [--delete] [--wait] [--verbose] [--timeout <number>] [--pivot] [--keep-overlay] [--async] [--keep-relative]** command to shorten a backing chain. The command has many options, which are listed in the help menu or man page.

The **virsh blockcommit** command copies data from one part of the chain down into a backing file, allowing you to pivot the rest of the chain in order to bypass the committed portions. For example, suppose this is the current state:

```
base ← snap1 ← snap2 ← active.
```

Using **virsh blockcommit** moves the contents of snap2 into snap1, allowing you to delete snap2 from the chain, making backups much quicker.

Procedure 20.1. How to shorten a backing chain

- Enter the following command, replacing *guest1* with the name of your guest virtual machine and *disk1* with the name of your disk.

  ```
  # virsh blockcommit guest1 disk1 --base snap1 --top snap2 --wait --verbose
  ```

  The contents of snap2 are moved into snap1, resulting in:

  base ← snap1 ← active. Snap2 is no longer valid and can be deleted

WARNING

virsh blockcommit will corrupt any file that depends on the ***--base*** argument (other than files that depended on the ***--top*** argument, as those files now point to the base). To prevent this, do not commit changes into files shared by more than one guest. The ***--verbose*** option will allow the progress to be printed on the screen.

20.13.2. Shortening a Backing Chain by Flattening the Image

virsh blockpull can be used in in the following applications:

1. Flattens an image by populating it with data from its backing image chain. This makes the image file self-contained so that it no longer depends on backing images and looks like this:
   - Before: base.img ← active
   - After: base.img is no longer used by the guest and Active contains all of the data.
2. Flattens part of the backing image chain. This can be used to flatten snapshots into the top-level image and looks like this:
   - Before: base ← sn1 ←sn2 ← active
   - After: base.img ← active. Note that **active** now contains all data from **sn1** and **sn2**, and neither sn1 nor sn2 are used by the guest.
3. Moves the disk image to a new file system on the host. This is allows image files to be moved while the guest is running and looks like this:
   - Before (The original image file): **`/fs1/base.vm.img`**
   - After: **`/fs2/active.vm.qcow2`** is now the new file system and **`/fs1/base.vm.img`** is no longer used.
4. Useful in live migration with post-copy storage migration. The disk image is copied from the source host to the destination host after live migration completes.

   In short this is what happens: Before:**`/source-host/base.vm.img`** After:**`/destination-host/active.vm.qcow2`**.**`/source-host/base.vm.img`** is no longer used.

Procedure 20.2. How to shorten a backing chain by flattening the data

1. It may be helpful to create a snapshot prior to running **`virsh blockpull`**. To do so, use the **`virsh snapshot-create-as`** command. In the following example, replace *guest1* with the name of your guest virtual machine, and *snap1* with the name of your snapshot.

   ```
   # virsh snapshot-create-as guest1 snap1 --disk-only
   ```

2. If the chain looks like this: **`base`** ← **`snap1`** ← **`snap2`** ← **`active`**, enter the following command, replacing *guest1* with the name of your guest virtual machine and *path1* with the source path to your disk (**`/home/username/VirtualMachines/*`**, for example).

   ```
   # virsh blockpull guest1 path1
   ```

   This command makes *snap1* the backing file of active, by pulling data from *snap2* into active resulting in: **`base`** ← **`snap1`** ← **`active`**.

3. Once the **`virsh blockpull`** is complete, the **libvirt** tracking of the snapshot that created the extra image in the chain is no longer useful. Delete the tracking on the outdated snapshot with this command, replacing guest1 with the name of your guest virtual machine and snap1 with the name of your snapshot.

   ```
   # virsh snapshot-delete guest1 snap1 --metadata
   ```

Additional applications of **`virsh blockpull`** can be performed as follows:

Example 20.31. How to flatten a single image and populate it with data from its backing image chain

The following example flattens the *vda* virtual disk on guest *guest1* and populates the image with data from its backing image chain, waiting for the populate action to be complete.

```
# virsh blockpull guest1 vda --wait
```

Example 20.32. How to flatten part of the backing image chain

The following example flattens the *vda* virtual disk on guest *guest1* based on the */path/to/base.img* disk image.

```
# virsh blockpull guest1 vda /path/to/base.img --base --wait
```

Example 20.33. How to move the disk image to a new file system on the host

To move the disk image to a new file system on the host, run the following two commands. In each command replace *guest1* with the name of your guest virtual machine and *disk1* with the name of your virtual disk. Change as well the XML file name and path to the location and name of the snapshot:

```
# virsh snapshot-create guest1 --xmlfile /path/to/snap1.xml --disk-only
```

```
# virsh blockpull guest1 disk1 --wait
```

Example 20.34. How to use live migration with post-copy storage migration

To use live migration with post-copy storage migration enter the following commands:

On the destination enter the following command replacing the backing file with the name and location of the backing file on the host.

```
# qemu-img create -f qcow2 -o backing_file=/source-host/vm.img
/destination-host/vm.qcow2
```

On the source enter the following command, replacing *guest1* with the name of your guest virtual machine:

```
# virsh migrate guest1
```

On the destination, enter the following command, replacing *guest1* with the name of your guest virtual machine and *disk1* with the name of your virtual disk:

```
# virsh blockpull guest1 disk1 --wait
```

20.13.3. Changing the Size of a Guest Virtual Machine's Block Device

The **virsh blockresize** command can be used to resize a block device of a guest virtual machine while the guest virtual machine is running, using the absolute path of the block device, which also corresponds to a unique target name (**<target dev="name"/>**) or source file (**<source file="name"/>**). This can be applied to one of the disk devices attached to guest virtual machine (you can use the command **virsh domblklist** to print a table showing the brief information of all block devices associated with a given guest virtual machine).

NOTE

Live image resizing will always resize the image, but may not immediately be picked up by guests. With recent guest kernels, the size of virtio-blk devices is automatically updated (older kernels require a guest reboot). With SCSI devices, it is required to manually trigger a re-scan in the guest with the command, **echo > /sys/class/scsi_device/0:0:0:0/device/rescan**. In addition, with IDE it is required to reboot the guest before it picks up the new size.

Example 20.35. How to resize the guest virtual machine block device

The following example resizes the *guest1* virtual machine's block device to 90 bytes:

```
# virsh blockresize guest1 90 B
```

20.14. DISPLAYING A URI FOR CONNECTION TO A GRAPHICAL DISPLAY

Running the **virsh domdisplay** command will output a URI that can then be used to connect to the graphical display of the guest virtual machine via VNC, SPICE, or RDP. The optional **--type** can be used to specify the graphical display type. If the argument **--include-password** is used, the SPICE channel password will be included in the URI.

Example 20.36. How to display the URI for SPICE

The following example displays the URI for SPICE, which is the graphical display that the virtual machine *guest1* is using:

```
# virsh domdisplay --type spice guest1
spice://192.0.2.1:5900
```

For more information about connection URIs, see the libvirt upstream pages.

20.15. DISPLAYING THE IP ADDRESS AND PORT NUMBER FOR THE VNC DISPLAY

The **virsh vncdisplay** command returns the IP address and port number of the VNC display for the specified guest virtual machine. If the information is unavailable for the guest, the exit code **1** is displayed.

Note that for this command to work, VNC has to be specified as a graphics type in the **devices** element of the guest's XML file. For further information, see Section 23.18.12, “Graphical Framebuffers”.

Example 20.37. How to display the IP address and port number for VNC

The following example displays the port number for the VNC display of the *guest1* virtual machine:

```
# virsh vncdisplay guest1
127.0.0.1:0
```

20.16. DISCARDING BLOCKS NOT IN USE

The **virsh domfstrim *domain* [--minimum *bytes*] [--mountpoint *mountPoint*]** command invokes the **fstrim** utility on all mounted file systems within a specified running guest virtual machine. This discards blocks not in use by the file system. If the argument **--minimum** is used, an amount in bytes must be specified. This amount will be sent to the guest kernel as its length of contiguous free range. Values smaller than this amount may be ignored. Increasing this value will create competition with file systems with badly fragmented free space. Note that not all blocks in this case are discarded. The default minimum is zero which means that every free block is discarded. If you increase this value to greater than zero, the fstrim operation will complete more quickly for file systems with badly fragmented free space, although not all blocks will be discarded. If a user only wants to trim one specific mount point, the **--mountpoint** argument should be used and a mount point should be specified.

Example 20.38. How to discard blocks not in use

The following example trims the file system running on the guest virtual machine named *guest1*:

```
# virsh domfstrim guest1 --minimum 0
```

20.17. GUEST VIRTUAL MACHINE RETRIEVAL COMMANDS

20.17.1. Displaying the Host Physical Machine Name

The **virsh domhostname *domain*** command displays the specified guest virtual machine's physical host name provided the hypervisor can publish it.

Example 20.39. How to display the host physical machine name

The following example displays the host physical machine name for the *guest1* virtual machine, if the hypervisor makes it available:

```
# virsh domhostname guest1
```

20.17.2. Displaying General Information about a Virtual Machine

The **virsh dominfo *domain*** command displays basic information about a specified guest virtual machine. This command may also be used with the option **[--domain] *guestname***.

Example 20.40. How to display general information about the guest virtual machine

The following example displays general information about the guest virtual machine named *guest1*:

```
# virsh dominfo guest1
Id:             8
Name:           guest1
UUID:           90e0d63e-d5c1-4735-91f6-20a32ca22c40
OS Type:        hvm
State:          running
CPU(s):         1
CPU time:       271.9s
Max memory:     1048576 KiB
Used memory:    1048576 KiB
Persistent:     yes
Autostart:      disable
Managed save:   no
Security model: selinux
Security DOI:   0
Security label: system_u:system_r:svirt_t:s0:c422,c469 (enforcing)
```

20.17.3. Displaying a Virtual Machine's ID Number

Although **virsh list** includes the ID in its output, the **virsh domid *domain*>|<*ID*** displays the ID for the guest virtual machine, provided it is running. An ID will change each time you run the virtual machine. If guest virtual machine is shut off, the machine name will be displayed as a series of dashes ('----'). This command may also be used with the **[--domain *guestname*]** option.

Example 20.41. How to display a virtual machine's ID number

In order to run this command and receive any usable output, the virtual machine should be running. The following example produces the ID number of the *guest1* virtual machine:

```
# virsh domid guest1
8
```

20.17.4. Aborting Running Jobs on a Guest Virtual Machine

The **virsh domjobabort *domain*** command aborts the currently running job on the specified guest virtual machine. This command may also be used with the **[--domain *guestname*]** option.

Example 20.42. How to abort a running job on a guest virtual machine

In this example, there is a job running on the *guest1* virtual machine that you want to abort. When running the command, change *guest1* to the name of your virtual machine:

```
# virsh domjobabort guest1
```

20.17.5. Displaying Information about Jobs Running on the Guest Virtual Machine

The **virsh domjobinfo *domain*** command displays information about jobs running on the specified guest virtual machine, including migration statistics. This command may also be used with the **[--domain *guestname*]** option, or with the **--completed** option to return information on the statistics of a recently completed job.

Example 20.43. How to display statistical feedback

The following example lists statistical information about the *guest1* virtual machine:

```
# virsh domjobinfo guest1
Job type:         Unbounded
Time elapsed:     1603         ms
Data processed:   47.004 MiB
Data remaining:   658.633 MiB
Data total:       1.125 GiB
Memory processed: 47.004 MiB
Memory remaining: 658.633 MiB
Memory total:     1.125 GiB
Constant pages:   114382
Normal pages:     12005
Normal data:      46.895 MiB
Expected downtime: 0            ms
Compression cache: 64.000 MiB
Compressed data:  0.000 B
Compressed pages: 0
Compression cache misses: 12005
Compression overflows: 0
```

20.17.6. Displaying the Guest Virtual Machine's Name

The **virsh domname *domainID*** command displays the name guest virtual machine name, given its ID or UUID. Although the **virsh list --all** command will also display the guest virtual machine's name, this command only lists the guest's name.

Example 20.44. How to display the name of the guest virtual machine

The following example displays the name of the guest virtual machine with domain ID *8*:

```
# virsh domname 8
guest1
```

20.17.7. Displaying the Virtual Machine's State

The **virsh domstate *domain*** command displays the state of the given guest virtual machine. Using the **--reason** argument will also display the reason for the displayed state. This command may also be used with the **[--domain *guestname*]** option, as well as the **--reason** option, which displays the reason for the state. If the command reveals an error, you should run the command **virsh domblkerror**. See Section 20.12.7, “Displaying Errors on Block Devices” for more details.

Example 20.45. How to display the guest virtual machine's current state

The following example displays the current state of the *guest1* virtual machine:

```
# virsh domstate guest1
running
```

20.17.8. Displaying the Connection State to the Virtual Machine

`virsh domcontrol` ***`domain`*** displays the state of an interface to the hypervisor that is used to control a specified guest virtual machine. For states that are not OK or Error, it will also print the number of seconds that have elapsed since the control interface entered the displayed state.

Example 20.46. How to display the guest virtual machine's interface state

The following example displays the current state of the *guest1* virtual machine's interface.

```
# virsh domcontrol guest1
ok
```

20.18. CONVERTING QEMU ARGUMENTS TO DOMAIN XML

The **`virsh domxml-from-native`** command provides a way to convert an existing set of QEMU arguments into a Domain XML configuration file that can then be used by libvirt. Note that this command is intended to be used only to convert existing QEMU guests previously started from the command line, in order to enable them to be managed through libvirt. Therefore, the method described here should not be used to create new guests from scratch. New guests should be created using either virsh, virt-install, or virt-manager. Additional information can be found on the libvirt upstream website.

Procedure 20.3. How to convert a QEMU guest to libvirt

1. Start with a QEMU guest with a arguments file (file type **`*.args`**), named *demo.args* in this example:

```
$ cat demo.args
LC_ALL=C
PATH=/bin
HOME=/home/test
USER=test
LOGNAME=test /usr/bin/qemu -S -M pc -m 214 -smp 1 -nographic -
monitor pty -no-acpi -boot c -hda /dev/HostVG/QEMUGuest1 -net none -
serial none -parallel none -usb
```

2. To convert this file into a domain XML file so that the guest can be managed by libvirt, enter the following command. Remember to replace *qemu-guest1* with the name of your guest virtual machine and *demo.args* with the filename of your QEMU args file.

```
# virsh domxml-from-native qemu-guest1 demo.args
```

This command turns the *demo.args* file into the following domain XML file:

```
<domain type='qemu'>
  <uuid>00000000-0000-0000-0000-000000000000</uuid>
  <memory>219136</memory>
  <currentMemory>219136</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type arch='i686' machine='pc'>hvm</type>
    <boot dev='hd'/>
  </os>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu</emulator>
    <disk type='block' device='disk'>
      <source dev='/dev/HostVG/QEMUGuest1'/>
      <target dev='hda' bus='ide'/>
    </disk>
  </devices>
</domain>
```

Figure 20.1. Guest virtual machine new configuration file

20.19. CREATING A DUMP FILE OF A GUEST VIRTUAL MACHINE'S CORE USING `VIRSH DUMP`

One of the methods of troubleshooting guest virtual machines (in addition to **kdump** and **pvpanic**) is using the **`virsh dump` *`domain corefilepath`* `[--bypass-cache] {--live | --crash | --reset} [--verbose] [--memory-only] [--format=`*`format`*`]`** command. This creates a dump file containing the core of the guest virtual machine so that it can be analyzed, for example by the **`crash`** utility.

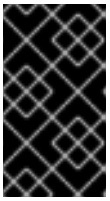
Specifically, running **`virsh dump`** command dumps the guest virtual machine core to a file specified by the core file path that you supply. Note that some hypervisors may give restrictions on this action and may require the user to manually ensure proper permissions on the file and path specified in the *corefilepath* parameter. This command is supported with SR-IOV devices as well as other passthrough devices. The following arguments are supported and have the following effect:

- **`--bypass-cache`** - The file saved will not bypass the host's file system cache. It has no effect on the content of the file. Note that selecting this option may slow down the dump operation.
- **`--live`** will save the file as the guest virtual machine continues to run and will not pause or stop the guest virtual machine.
- **`--crash`** puts the guest virtual machine in a crashed status rather than leaving it in a paused state while the dump file is saved. The guest virtual machine will be listed as "Shut off", with the reason as "Crashed".
- **`--reset`** - When the dump file is successfully saved, the guest virtual machine will reset.
- **`--verbose`** displays the progress of the dump process

- **--memory-only** - Running a dump using this option will create a dump file where the contents of the dump file will only contain the guest virtual machine's memory and CPU common register file. This option should be used in cases where running a full dump will fail. This may happen when a guest virtual machine cannot be live migrated (due to a passthrough PCI device).

  You can save the memory-only dump using the **--format=*format*** option. The following formats are available:

  - **elf** - the default, uncompressed format
  - **kdump-zlib** - kdump-compressed format with zlib compression
  - **kdump-lzo** - kdump-compressed format with LZO compression
  - **kdump-snappy** - kdump-compressed format with Snappy compression



IMPORTANT

The **crash** utility no longer supports the default core dump file format of the **virsh dump** command. If you use **crash** to analyze to core dump file created by **virsh dump**, you must use the **--memory-only** option.

Note that the entire process can be monitored using the **virsh domjobinfo** command and can be canceled using the **virsh domjobabort** command.

Example 20.47. How to create a dump file with virsh

The following example creates a dump file of the *guest1* virtual machine's core, saves it into the **core/file/path.file** file, and then resets the guest. The most common scenario for using this command is if your guest virtual machine is not behaving properly:

```
# virsh dump guest1 core/file/path.file --reset
```

20.20. CREATING A VIRTUAL MACHINE XML DUMP (CONFIGURATION FILE)

The **virsh dumpxml** command will return the guest virtual machine's XML configuration file which you can then use, save, or change as needed.

The XML file (**guest.xml**) can then be used to recreate the guest virtual machine (refer to Section 20.22, “Editing a Guest Virtual Machine's XML Configuration Settings”. You can edit this XML configuration file to configure additional devices or to deploy additional guest virtual machines.

Example 20.48. How to retrieve the XML file for a guest virtual machine

The following example retrieves the XML configuration of the *guest1* virtual machine, and pipes it int othe *guest1.xml* file.

```
# virsh dumpxml guest1 | guest1.xml
<domain type='kvm'>
  <name>guest1-rhel6-64</name>
```

```
  <uuid>b8d7388a-bbf2-db3a-e962-b97ca6e514bd</uuid>
  <memory>2097152</memory>
  <currentMemory>2097152</currentMemory>
  <vcpu>2</vcpu>
  <os>
    <type arch='x86_64' machine='rhel6.2.0'>hvm</type>
    <boot dev='hd'/>
  </os>
[...]
```

20.21. CREATING A GUEST VIRTUAL MACHINE FROM A CONFIGURATION FILE

Guest virtual machines can be created from XML configuration files. You can copy existing XML from previously created guest virtual machines or use the **virsh dumpxml** command.

Example 20.49. How to create a guest virtual machine from an XML file

The following example creates a new virtual machine from the existing *guest1.xml* configuration file. You need to have this file before beginning. You can retrieve the file using the **virsh dumpxml** command. See Example 20.48, “How to retrieve the XML file for a guest virtual machine” for instructions.

```
# virsh create guest1.xml
```

20.22. EDITING A GUEST VIRTUAL MACHINE'S XML CONFIGURATION SETTINGS

The **virsh edit** command enables the user to edit the domain XML configuration file of a specified guest. Running this command opens the XML file in a text editor, specified by the **$EDITOR** shell parameter (set to **vi** by default).

Example 20.50. How to edit a guest virtual machine's XML configuration settings

The following example opens the XML configuration file associated with the *guest1* virtual machine in your default text editor:

```
#  virsh edit guest1
```

20.23. ADDING MULTIFUNCTION PCI DEVICES TO KVM GUEST VIRTUAL MACHINES

To add a multi-function PCI device to a KVM guest virtual machine:

1. Run the **virsh edit *guestname*** command to edit the XML configuration file for the guest virtual machine.

2. In the **<address>** element, add a **multifunction='on'** attribute. This enables the use of other functions for the particular multifunction PCI device.

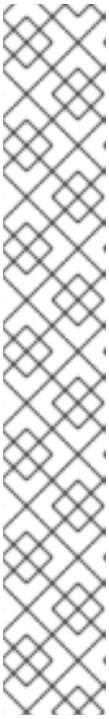
```
<disk type='file' device='disk'>
<driver name='qemu' type='raw' cache='none'/>
<source file='/var/lib/libvirt/images/rhel62-1.img'/>
<target dev='vda' bus='virtio'/>
<address type='pci' domain='0x0000' bus='0x00' slot='0x05'
function='0x0' multifunction='on'/>
</disk>
```

For a PCI device with two functions, amend the XML configuration file to include a second device with the same slot number as the first device and a different function number, such as **function='0x1'**. For Example:

```
<disk type='file' device='disk'>
<driver name='qemu' type='raw' cache='none'/>
<source file='/var/lib/libvirt/images/rhel62-1.img'/>
<target dev='vda' bus='virtio'/>
<address type='pci' domain='0x0000' bus='0x00' slot='0x05'
function='0x0' multifunction='on'/>
</disk>
<disk type='file' device='disk'>
<driver name='qemu' type='raw' cache='none'/>
<source file='/var/lib/libvirt/images/rhel62-2.img'/>
<target dev='vdb' bus='virtio'/>
<address type='pci' domain='0x0000' bus='0x00' slot='0x05'
function='0x1'/>
</disk>
```

3. Run the **lspci** command. The output from the KVM guest virtual machine shows the virtio block device:

```
$ lspci

00:05.0 SCSI storage controller: Red Hat, Inc Virtio block device
00:05.1 SCSI storage controller: Red Hat, Inc Virtio block device
```



NOTE

The SeaBIOS application runs in real mode for compatibility with BIOS interfaces. This limits the amount of memory available. As a consequence, SeaBIOS is only able to handle a limited number of disks. Currently, the supported number of disks is:

- virtio-scsi — 64
- virtio-blk — 4
- ahci/sata — 24 (4 controllers with all 6 ports connected)
- usb-storage — 4

As a workaround for this problem, when attaching a large number of disks to your virtual machine, make sure that your system disk has a small pci slot number, so SeaBIOS sees it first when scanning the pci bus. It is also recommended to use the virtio-scsi device instead of virtio-blk as the per-disk memory overhead is smaller.

20.24. DISPLAYING CPU STATISTICS FOR A SPECIFIED GUEST VIRTUAL MACHINE

The **virsh cpu-stats *domain* --total start count** command provides the CPU statistical information on the specified guest virtual machine. By default, it shows the statistics for all CPUs, as well as a total. The **--total** option will only display the total statistics. The **--count** option will only display statistics for *count* CPUs.

Example 20.51. How to generate CPU statistics for the guest virtual machine

The following example generates CPU statistics for the guest virtual machine named *guest1*.

```
# virsh cpu-stats guest1

CPU0:
 cpu_time           242.054322158 seconds
 vcpu_time          110.969228362 seconds
CPU1:
 cpu_time           170.450478364 seconds
 vcpu_time          106.889510980 seconds
CPU2:
 cpu_time           332.899774780 seconds
 vcpu_time          192.059921774 seconds
CPU3:
 cpu_time           163.451025019 seconds
 vcpu_time           88.008556137 seconds
Total:
 cpu_time           908.855600321 seconds
 user_time           22.110000000 seconds
 system_time         35.830000000 seconds
```

20.25. TAKING A SCREENSHOT OF THE GUEST CONSOLE

The **virsh screenshot *guestname* [imagefilepath]** command takes a screenshot of a current guest virtual machine console and stores it into a file. If no file path is provided, the screenshot is saved to the current directory. If the hypervisor supports multiple displays for a guest virtual machine, use the **--*screen* screenID** option to specify the screen to be captured.

Example 20.52. How to take a screenshot of a guest machine's console

The following example takes a screenshot of the *guest1* machine's console and saves it as **/home/username/pics/guest1-screen.png**:

```
#  virsh screenshot guest1 /home/username/pics/guest1-screen.ppm
Screenshot saved to /home/username/pics/guest1-screen.ppm, with type of
image/x-portable-pixmap
```

20.26. SENDING A KEYSTROKE COMBINATION TO A SPECIFIED GUEST VIRTUAL MACHINE

The **virsh send-key *domain* --codeset --holdtime *keycode*** command allows you to send a sequence as a *keycode* to a specific guest virtual machine. Each *keycode* can either be a numeric value or a symbolic name from the corresponding *codeset* below.

If a **--holdtime** is given, each keystroke will be held for the specified amount in milliseconds. The **--codeset** allows you to specify a code set, the default being **Linux**, but the following options are permitted:

- **linux** - choosing this option causes the symbolic names to match the corresponding Linux key constant macro names and the numeric values are those offered by the Linux generic input event subsystems.
- **xt** - this will send a value that is defined by the XT keyboard controller. No symbolic names are provided
- **atset1** - the numeric values are those that are defined by the AT keyboard controller, set1 (XT compatible set). Extended keycodes from the atset1 may differ from extended keycodes in the XT codeset. No symbolic names are provided.
- **atset2** - The numeric values are those defined by the AT keyboard controller, set 2. No symbolic names are provided.
- **atset3** - The numeric values are those defined by the AT keyboard controller, set 3 (PS/2 compatible). No symbolic names are provided.
- **os_x** - The numeric values are those defined by the OS-X keyboard input subsystem. The symbolic names match the corresponding OS-X key constant macro names.
- **xt_kbd** - The numeric values are those defined by the Linux KBD device. These are a variant on the original XT codeset, but often with different encoding for extended keycodes. No symbolic names are provided.
- **win32** - The numeric values are those defined by the Win32 keyboard input subsystem. The symbolic names match the corresponding Win32 key constant macro names.

- **usb** - The numeric values are those defined by the USB HID specification for keyboard input. No symbolic names are provided.
- **rfb** - The numeric values are those defined by the RFB extension for sending raw keycodes. These are a variant on the XT codeset, but extended keycodes have the low bit of the second bite set, instead of the high bit of the first byte. No symbolic names are provided.

Example 20.53. How to send a keystroke combination to a guest virtual machine

The following example sends the **Left Ctrl**, **Left Alt**, and **Delete** in the Linux encoding to the *guest1* virtual machine, and holds them for 1 second. These keys are all sent simultaneously, and may be received by the guest in a random order:

```
# virsh send-key guest1 --codeset Linux --holdtime 1000 KEY_LEFTCTRL
KEY_LEFTALT KEY_DELETE
```

NOTE

If multiple *keycodes* are specified, they are all sent simultaneously to the guest virtual machine and as such may be received in random order. If you need distinct keycodes, you must run the **virsh send-key** command multiple times in the order you want the sequences to be sent.

20.27. HOST MACHINE MANAGEMENT

This section contains the commands needed for managing the host system (referred to as a *node* by the commands).

20.27.1. Displaying Host Information

The **virsh nodeinfo** command displays basic information about the host, including the model number, number of CPUs, type of CPU, and size of the physical memory. The output corresponds to the **virNodeInfo** structure. Specifically, the "CPU socket(s)" field indicates the number of CPU sockets per NUMA cell.

Example 20.54. How to display information about your host machine

The following example retrieves information about your host:

```
$ virsh nodeinfo
CPU model:           x86_64
CPU(s):              4
CPU frequency:       1199 MHz
CPU socket(s):       1
Core(s) per socket:  2
Thread(s) per core:  2
NUMA cell(s):        1
Memory size:         3715908 KiB
```

20.27.2. Setting NUMA Parameters

The **virsh numatune** command can either set or retrieve the NUMA parameters for a specified guest virtual machine. Within the guest virtual machine's configuration XML file these parameters are nested within the **<numatune>** element. Without using flags, only the current settings are displayed. The **numatune *domain*** command requires a specified guest virtual machine name and can take the following arguments:

- **--mode** - The mode can be set to either **strict**, **interleave**, or **preferred**. Running domains cannot have their mode changed while live unless the guest virtual machine was started within **strict** mode.
- **--nodeset** contains a list of NUMA nodes that are used by the host physical machine for running the guest virtual machine. The list contains nodes, each separated by a comma, with a dash **-** used for node ranges and a caret **^** used for excluding a node.
- Only one of the three following flags can be used per instance
  - **--config** will effect the next boot of a persistent guest virtual machine
  - **--live** will set the scheduler information of a running guest virtual machine.
  - **--current** will effect the current state of the guest virtual machine.

Example 20.55. How to set the NUMA parameters for the guest virtual machine

The following example sets the NUMA mode to **strict** for nodes 0, 2, and 3 for the running *guest1* virtual machine:

```
#  virsh numatune guest1 --mode strict --nodeset 0,2-3 --live
```

Running this command will change the running configuration for guest1 to the following configuration in its XML file.

```
<numatune>
        <memory mode='strict' nodeset='0,2-3'/>
</numatune>
```

20.27.3. Displaying the Amount of Free Memory in a NUMA Cell

The **virsh freecell** command displays the available amount of memory on the machine within a specified NUMA cell. This command can provide one of three different displays of available memory on the machine depending on the options specified. specified cell.

Example 20.56. How to display memory properties for virtual machines and NUMA cells

The following command displays the total amount of available memory in all cells:

```
# virsh freecell
Total: 684096 KiB
```

To display also the amount of available memory in individual cells, use the **--all** option:

```
# virsh freecell --all
```

```
    0:     804676 KiB
--------------------
Total:     804676 KiB
```

To display the amount of individual memory in a specific cell, use the **--cellno** option:

```
# virsh freecell --cellno 0
0: 772496 KiB
```

20.27.4. Displaying a CPU List

The **virsh nodecpumap** command displays the number of CPUs that are available to the host machine, and it also lists how many are currently online.

Example 20.57. How to display number of CPUs that available to the host

The following example displays the number of CPUs available to the host:

```
# virsh nodecpumap
   CPUs present: 4
   CPUs online: 1
   CPU map: y
```

20.27.5. Displaying CPU Statistics

The **virsh nodecpustats [*cpu_number*] [--percent]** command displays statistical information about the CPUs load status of the host. If a CPU is specified, the statistics are only for the specified CPU. If the **percent** option is specified, the command displays the percentage of each type of CPU statistics that were recorded over an one (1) second interval.

Example 20.58. How to display statistical information about CPU usage

The following example returns general statistics about the host CPUs load:

```
# virsh nodecpustats
user:               1056442260000000
system:              401675280000000
idle:               7549613380000000
iowait:               94593570000000
```

This example displays the statistics for CPU number 2 as percentages:

```
# virsh nodecpustats 2 --percent
usage:            2.0%
user:             1.0%
system:           1.0%
idle:            98.0%
iowait:           0.0%
```

20.27.6. Managing Devices

20.27.6.1. Attaching and updating a device with virsh

For information on attaching storage devices, see Section 13.3.6, “Adding Storage Devices to Guests”.

Procedure 20.4. Hot plugging USB devices for use by the guest virtual machine

USB devices can be either attached to the virtual machine that is running by hot plugging, or while the guest is shut off. The device you want to use in the guest must be attached to the host machine.

1. Locate the USB device you want to attach by running the following command:

```
# lsusb -v

idVendor           0x17ef Lenovo
idProduct          0x480f Integrated Webcam [R5U877]
```

2. Create an XML file and give it a logical name (**usb_device.xml**, for example). Copy the vendor and product ID number (a hexidecimal number) exactly as was displayed in your search. Add this information to the XML file as shown in Figure 20.2, “USB devices XML snippet”. Remember the name of this file as you will need it in the next step.

```
<hostdev mode='subsystem' type='usb' managed='yes'>
  <source>
    <vendor id='0x17ef'/>
    <product id='0x480f'/>
  </source>
</hostdev>
```

Figure 20.2. USB devices XML snippet

3. Attach the device by running the following command. When you run the command, replace *guest1* with the name of your virtual machine and *usb_device.xml* with the name of your XML file that contains the vendor and product ID of your device, which you created in the previous step. For the change take effect at the next reboot, use the **--config** argument. For the change to take effect on the current guest virtual machine, use the **--current** argument. See the virsh man page for additional arguments.

```
# virsh attach-device guest1 --file usb_device.xml --config
```

Example 20.59. How to hot unplug devices from a guest virtual machine

The following example detaches the USB device configured with the *usb_device1.xml* file from the *guest1* virtual machine:

```
# virsh detach-device guest1 --file usb_device.xml
```

20.27.6.2. Attaching interface devices

The **virsh attach-interface *domain type source* [<target>] [<mac>] [<script>] [<model>] [<inbound>] [<outbound>] [--config] [--live] [--current]** command can take the following arguments:

- **--type** - allows you to set the interface type
- **--source** - allows you to set the source of the network interface
- **--live** - gets its value from running guest virtual machine configuration settings
- **--config** - takes effect at next boot
- **--current** - gets its value according to the current configuration settings
- **--target** - indicates the target device in the guest virtual machine.
- **--mac** - use this option to specify the MAC address of the network interface
- **--script** - use this option to specify a path to a script file handling a bridge instead of the default one.
- **--model** - use this option to specify the model type.
- **--inbound** - controls the inbound bandwidth of the interface. Acceptable values are **average**, **peak**, and **burst**.
- **--outbound** - controls the outbound bandwidth of the interface. Acceptable values are **average**, **peak**, and **burst**.

NOTE

Values for average and peak are expressed in kilobytes per second, while burst is expressed in kilobytes in a single burst at peak speed as described in the Network XML upstream documentation.

The *type* can be either **network** to indicate a physical network device, or **bridge** to indicate a bridge to a device. *source* is the source of the device. To remove the attached device, use the **virsh detach-device** command.

Example 20.60. How to attach a device to the guest virtual machine

The following example attaches the *networkw* network device to the *guest1* virtual machine. The interface model is going to be presented to the guest as **virtio**:

```
# virsh attach-interface guest1 networkw --model virtio
```

20.27.6.3. Changing the media of a CDROM

The **virsh change-media** command changes the media of a CDROM to another source or format. The command takes the following arguments. More examples and explanation for these arguments can also be found in the man page.

- **--path** - A string containing a fully-qualified path or target of disk device
- **--source** - A string containing the source of the media
- **--eject** - Ejects the media
- **--insert** - Inserts the media
- **--update** - Updates the media
- **--current** - Can be either or both of **--live** and **--config**, which depends on implementation of hypervisor driver
- **--live** - Alters the live configuration of running guest virtual machine
- **--config** - Alters the persistent configuration, effect observed on next boot
- **--force** - Forces media to change

20.27.7. Suspending the Host

The **virsh nodesuspend *targetduration*** command puts the host machine into a system-wide sleep state similar to that of Suspend-to-RAM (s3), Suspend-to-Disk (s4), or Hybrid-Suspend, and sets up a Real-Time-Clock to wake up the host after the duration that is set has past. The **target** variable can be set to either ***mem***, ***disk***, or ***hybrid***. These options indicate to set the memory, disk, or combination of the two to suspend. Setting the **--duration** instructs the node to wake up after the set duration time has run out. It is set in seconds. It is recommended that the duration time be longer than 60 seconds.

Example 20.61. How to suspend the host machine to disk s4

The following example suspends the host physical machine to disk for 90 seconds:

```
# virsh nodesuspend disk 90
```

20.27.8. Setting and Displaying the Node Memory Parameters

The **virsh node-memory-tune [shm-pages-to-scan] [shm-sleep-milisecs] [shm-merge-across-nodes]** command displays and allows you to set the node memory parameters. The following parameters may be set with this command:

- **--shm-pages-to-scan** - sets the number of pages to scan before the kernel samepage merging (KSM) service goes to sleep.
- **--shm-sleep-milisecs** - sets the number of miliseconds that KSM will sleep before the next scan
- **--shm-merge-across-nodes** - specifies if pages from different NUMA nodes can be merged

Example 20.62. How to merge memory pages across NUMA nodes

The following example merges all of the memory pages from all of the NUMA nodes:

```
# virsh node-memory-tune --shm-merge-across-nodes 1
```

20.27.9. Listing Devices on a Host

The **virsh nodedev-list *--cap* --tree** command lists all the devices available on the host that are known to the **libvirt** service. *--cap* is used to filter the list by capability types, each separated by a comma, and cannot be used with **--tree**. Using the argument **--tree**, puts the output into a tree structure.

Example 20.63. How to display the devices available on a host

The following example lists devices that are available on a host in a tree format. Note that the list has been truncated:

```
# virsh nodedev-list --tree
computer
  |
  +- net_lo_00_00_00_00_00_00
  +- net_macvtap0_52_54_00_12_fe_50
  +- net_tun0
  +- net_virbr0_nic_52_54_00_03_7d_cb
  +- pci_0000_00_00_0
  +- pci_0000_00_02_0
  +- pci_0000_00_16_0
  +- pci_0000_00_19_0
  |   |
  |   +- net_eth0_f0_de_f1_3a_35_4f
 [...]
```

This example lists SCSI devices available on a host:

```
# virsh nodedev-list --cap scsi
scsi_0_0_0_0
```

20.27.10. Creating Devices on Host Machines

The **virsh nodedev-create *file*** command allows you to create a device on a host physical machine and then assign it to a guest virtual machine. Although **libvirt** automatically detects which host nodes are available for use, this command allows you to register hardware that **libvirt** did not detect. The specified file should contain the XML description for the top level **<device>** description of the host device. For an example of such file, see Example 20.66, “How to retrieve the XML file for a device”.

Example 20.64. How to create a device from an XML file

In this example, you have already created an XML file for your PCI device and have saved it as **scsi_host2.xml**. The following command enables you to attach this device to your guests:

```
# virsh nodedev-create scsi_host2.xml
```

20.27.11. Removing a Device

The **virsh nodedev-destroy** command removes the device from the host. Note that the virsh node device driver does not support persistent configurations, so rebooting the host machine makes the device usable again.

Also note that different assignments expect the device to be bound to different back-end driver (vfio, kvm). Using the **--driver** argument allows you to specify the intended back-end driver.

Example 20.65. How to remove a device from a host physical machine

The following example removes a SCSI device named *scsi_host2* from the host machine:

```
# virsh nodedev-destroy scsi_host2
```

20.27.12. Collect Device Configuration Settings

The **virsh nodedev-dumpxml *device*** command outputs the XML representation for the specified host device, including information such as the device name, the bus to which the device is connected, the vendor, product ID, capabilities, as well as any information usable by **libvirt**. The argument *device* can either be a device name or WWN pair in WWNN, WWPN format (HBA only).

Example 20.66. How to retrieve the XML file for a device

The following example retrieves the XML file for a SCSI device identified as *scsi_host2*. The name was obtained by using the virsh nodedev-list command:

```
# virsh nodedev-dumpxml scsi_host2
   <device>
     <name>scsi_host2</name>
     <parent>scsi_host1</parent>
     <capability type='scsi_host'>
       <capability type='fc_host'>
         <wwnn>2001001b32a9da5b</wwnn>
         <wwpn>2101001b32a9da5b</wwpn>
       </capability>
     </capability>
   </device>
```

20.27.13. Triggering a Reset for a Device

The **virsh nodedev-reset *device*** command triggers a device reset for the specified device. Running this command is useful prior to transferring a node device between guest virtual machine pass through or the host physical machine. **libvirt** will do this action automatically, when required, but this command allows an explicit reset when needed.

Example 20.67. How to reset a device on a guest virtual machine

The following example resets the device on the guest virtual machine named *scsi_host2*:

```
# virsh nodedev-reset scsi_host2
```

20.28. RETRIEVING GUEST VIRTUAL MACHINE INFORMATION

20.28.1. Getting the Domain ID of a Guest Virtual Machine

The **virsh domid** command returns the guest virtual machine's ID. Note that this changes each time the guest starts or restarts. This command requires either the name of the virtual machine or the virtual machine's UUID.

Example 20.68. How to retrieve the domain ID for a guest virtual machine

The following example retrieves the domain ID of a guest virtual machine named *guest1*:

```
# virsh domid guest1
8
```

Note, **domid** returns **-** for guest virtual machines that are in shut off state. To confirm that the virtual machine is shutoff, you can run the **virsh list --all** command.

20.28.2. Getting the Domain Name of a Guest Virtual Machine

The **virsh domname** command returns the name of the guest virtual machine given its ID or UUID. Note that the ID changes each time the guest starts.

Example 20.69. How to retrieve a virtual machine's ID

The following example retrieves the name for the guest virtual machine whose ID is *8*:

```
# virsh domname 8
guest1
```

20.28.3. Getting the UUID of a Guest Virtual Machine

The **virsh domuuid** command returns the UUID or the *Universally Unique Identifier* for a given guest virtual machine or ID.

Example 20.70. How to display the UUID for a guest virtual machine

The following example retrieves the UUID for the guest virtual machine named *guest1*:

```
# virsh domuuid guest1
r5b2-mySQL01 4a4c59a7-ee3f-c781-96e4-288f2862f011
```

20.28.4. Displaying Guest Virtual Machine Information

The **virsh dominfo** command displays information on that guest virtual machine given a virtual machine's name, ID, or UUID. Note that the ID changes each time the virtual machine starts.

Example 20.71. How to display guest virtual machine general details

The following example displays the general details about the guest virtual machine named *guest1*:

```
# virsh dominfo guest1
Id:             8
Name:           guest1
UUID:           90e0d63e-d5c1-4735-91f6-20a32ca22c48
OS Type:        hvm
State:          running
CPU(s):         1
CPU time:       32.6s
Max memory:     1048576 KiB
Used memory:    1048576 KiB
Persistent:     yes
Autostart:      disable
Managed save:   no
Security model: selinux
Security DOI:   0
Security label: system_u:system_r:svirt_t:s0:c552,c818 (enforcing)
```

20.29. STORAGE POOL COMMANDS

Using libvirt, you can manage various storage solutions, including files, raw partitions, and domain-specific formats, used to provide the storage volumes visible as devices within virtual machines. For more detailed information, see the libvirt upstream pages. Many of the commands for administering storage pools are similar to the ones used for guest virtual machines.

20.29.1. Searching for a Storage Pool XML

The **virsh find-storage-pool-sources *type*** command displays the XML describing all storage pools of a given source that could be found. Types include: netfs, disk, dir, fs, iscsi, logical, and gluster. Note that all of the types correspond to the storage back-end drivers and there are more types available (see the man page for more details). You can also further restrict the query for pools by providing an template source XML file using the *--srcSpec* option.

Example 20.72. How to list the XML setting of available storage pools

The following example outputs the XML setting of all logical storage pools available on the system:

```
# virsh find-storage-pool-sources logical
<sources>
  <source>
    <device path='/dev/mapper/luks-7a6bfc59-e7ed-4666-a2ed-
6dcbff287149'/>
    <name>RHEL_dhcp-2-157</name>
    <format type='lvm2'/>
  </source>
</sources>
```

20.29.2. Finding a storage Pool

The **virsh find-storage-pool-sources-as *type*** command finds potential storage pool sources, given a specific type. Types include: netfs, disk, dir, fs, iscsi, logical, and gluster. Note that all of the types correspond to the storage back-end drivers and there are more types available (see the man page for more details). The command also takes the optional arguments *host*, *port*, and *initiator*. Each of these options will dictate what gets queried.

Example 20.73. How to find potential storage pool sources

The following example searches for a disk-based storage pool on the specified host machine. If you are unsure of your host name run the command **virsh hostname** first:

```
# virsh find-storage-pool-sources-as disk --host myhost.example.com
```

20.29.3. Listing Storage Pool Information

The **virsh pool-info *pool*** command lists the basic information about the specified storage pool object. This command requires the name or UUID of the storage pool. To retrieve this information, use the **pool-list** command.

Example 20.74. How to retrieve information on a storage pool

The following example retrieves information on the storage pool named *vdisk*:

```
# virsh pool-info vdisk

Name:           vdisk
UUID:
State:          running
Persistent:     yes
Autostart:      no
Capacity:       125 GB
Allocation:     0.00
Available:      125 GB
```

20.29.4. Listing the Available Storage Pools

The **virsh pool-list** command lists all storage pool objects known to libvirt. By default, only active pools are listed; but using the **--inactive** argument lists just the inactive pools, and using the **--all** argument lists all of the storage pools. This command takes the following optional arguments, which filter the search results:

- **--inactive** - lists the inactive storage pools
- **--all** - lists both active and inactive storage pools
- **--persistent** - lists the persistent storage pools

- **--transient** - lists the transient storage pools
- **--autostart** - lists the storage pools with autostart enabled
- **--no-autostart** - lists the storage pools with autostart disabled
- **--type *type*** - lists the pools that are only of the specified type
- **--details** - lists the extended details for the storage pools

In addition to the above arguments, there are several sets of filtering flags that can be used to filter the content of the list. **--persistent** restricts the list to persistent pools, **--transient** restricts the list to transient pools, **--autostart** restricts the list to autostarting pools and finally **--no-autostart** restricts the list to the storage pools that have autostarting disabled.

For all storage pool commands which require a *--type*, the pool types must be separated by comma. The valid pool types include: **dir**, **fs**, **netfs**, **logical**, **disk**, **iscsi**, **scsi**, **mpath**, **rbd**, **sheepdog**, and **gluster**.

The **--details** option instructs **virsh** to additionally display pool persistence and capacity related information where available.

NOTE

When this command is used with older servers, it is forced to use a series of API calls with an inherent race, where a pool might not be listed or might appear more than once if it changed its state between calls while the list was being collected. Newer servers however, do not have this problem.

Example 20.75. How to list all storage pools

This example lists storage pools that are both active and inactive:

```
# virsh pool-list --all
Name                 State      Autostart
-----------------------------------------
default              active     yes
vdisk                active     no
```

20.29.5. Refreshing a Storage Pool List

The **virsh pool-refresh *pool*** command refreshes the list of storage volumes contained in storage pool.

Example 20.76. How to refresh the list of the storage volumes in a storage pool

The following example refreshes the list for the storage volume named *vdisk*:

```
# virsh pool-refresh vdisk

Pool vdisk refreshed
```

20.29.6. Creating, Defining, and Starting Storage Pools

20.29.6.1. Building a storage pool

The **virsh pool-build *pool*** command builds a storage pool using the name given in the command. The optional arguments **--overwrite** and **--no-overwrite** can only be used for an FS storage pool or with a disk or logical type based storage pool. Note that if [--overwrite] or [--no-overwrite] are not provided and the pool used is FS, it is assumed that the type is actually directory-based. In addition to the pool name, the storage pool UUID may be used as well.

If **--no-overwrite** is specified, it probes to determine if a file system already exists on the target device, returning an error if it exists, or using **mkfs** to format the target device if it does not. If **--overwrite** is specified, then the **mkfs** command is executed and any existing data on the target device is overwritten.

Example 20.77. How to build a storage pool

The following example creates a disk-based storage pool named *vdisk*:

```
# virsh pool-build vdisk

Pool vdisk built
```

20.29.6.2. Defining a storage pool from an XML file

The **virsh pool-define *file*** command creates, but does not start, a storage pool object from the XML *file*.

Example 20.78. How to define a storage pool from an XML file

This example assumes that you have already created an XML file with the settings for your storage pool. For example:

```
<pool type="dir">
 <name>vdisk</name>
 <target>
          <path>/var/lib/libvirt/images</path>
 </target>
</pool>
```

The following command then builds a directory type storage pool from the XML file (named *vdisk.xml* in this example):

```
# virsh pool-define vdisk.xml

Pool vdisk defined
```

To confirm that the storage pool was defined, run the **virsh pool-list --all** command as shown in Example 20.75, “How to list all storage pools”. When you run the command, however, the status will show as inactive as the pool has not been started. For directions on starting the storage

pool see Example 20.82, “How to start a storage pool”.

20.29.6.3. Creating storage pools

The **virsh pool-create *file*** command creates and starts a storage pool from its associated XML file.

Example 20.79. How to create a storage pool from an XML file

In this example assumes that you have already created an XML file with the settings for your storage pool. For example:

```
<pool type="dir">
 <name>vdisk</name>
 <target>
         <path>/var/lib/libvirt/images</path>
 </target>
</pool>
```

The following example builds a directory-type storage pool based on the XML file (named *vdisk.xml* in this example):

```
# virsh pool-create vdisk.xml

Pool vdisk created
```

To confirm that the storage pool was created, run the **virsh pool-list --all** command as shown in Example 20.75, “How to list all storage pools”. When you run the command, however, the status will show as inactive as the pool has not been started. For directions on starting the storage pool see Example 20.82, “How to start a storage pool”.

20.29.6.4. Creating storage pools

The **virsh pool-create-as *name*** command creates and starts a pool object name from the raw parameters given. This command takes the following options:

- **--print-xml** - displays the contents of the XML file, but does not define or create a storage pool from it
- **--type** *type* defines the storage pool type. See Section 20.29.4, “Listing the Available Storage Pools” for the types you can use.
- **--source-host** *hostname* - the source host physical machine for underlying storage
- **--source-path** *path* - the location of the underlying storage
- **--source-dev** *path* - the device for the underlying storage
- **--source-name** *name* - the name of the source underlying storage
- **--source-format** *format* - the format of the source underlying storage

- **--target** *path* - the target for the underlying storage

Example 20.80. How to create and start a storage pool

The following example creates and starts a storage pool named *vdisk* at the **/mnt** directory:

```
# virsh pool-create-as --name vdisk --type dir --target /mnt

Pool vdisk created
```

20.29.6.5. Defining a storage pool

The **virsh pool-define-as <*name*>** command creates, but does not start, a pool object name from the raw parameters given. This command accepts the following options:

- **--print-xml** - displays the contents of the XML file, but does not define or create a storage pool from it
- **--type** *type* defines the storage pool type. See Section 20.29.4, "Listing the Available Storage Pools" for the types you can use.
- **--source-host** *hostname* - source host physical machine for underlying storage
- **--source-path** *path* - location of the underlying storage
- **--source-dev** *devicename* - device for the underlying storage
- **--source-name** *sourcename* - name of the source underlying storage
- **--source-format** *format* - format of the source underlying storage
- **--target** *targetname* - target for the underlying storage

If **--print-xml** is specified, then it prints the XML of the pool object without creating or defining the pool. Otherwise, the pool requires a specified type to be built. For all storage pool commands which require a *type*, the pool types must be separated by comma. The valid pool types include: **dir**, **fs**, **netfs**, **logical**, **disk**, **iscsi**, **scsi**, **mpath**, **rbd**, **sheepdog**, and **gluster**.

Example 20.81. How to define a storage pool

The following example defines a storage pool named *vdisk*, but does not start it. After this command runs, use the **virsh pool-start** command to activate the storage pool:

```
# virsh pool-define-as --name vdisk --type dir --target /mnt

Pool vdisk defined
```

20.29.6.6. Starting a storage pool

The **virsh pool-start *pool*** command starts the specified storage pool, which was previously defined but inactive. This command may also use the UUID for the storage pool as well as the pool's name.

Example 20.82. How to start a storage pool

The following example starts the *vdisk* storage pool that you built in Example 20.79, “How to create a storage pool from an XML file”:

```
# virsh pool-start vdisk

Pool vdisk started
```

To verify the pool has started run the **virsh pool-list --all** command and confirm that the status is active, as shown in Example 20.75, “How to list all storage pools”.

20.29.6.7. Auto-starting a storage pool

The **virsh pool-autostart *pool*** command enables a storage pool to automatically start at boot. This command requires the pool name or UUID. To disable the **pool-autostart** command use the **--disable** argument in the command.

Example 20.83. How to autostart a storage pool

The following example autostarts the *vdisk* storage pool that you built in Example 20.79, “How to create a storage pool from an XML file”:

```
# virsh pool-autostart vdisk

Pool vdisk autostarted
```

20.29.7. Stopping and Deleting Storage Pools

The **virsh pool-destroy *pool*** command stops a storage pool. Once stopped, libvirt will no longer manage the pool but the raw data contained in the pool is not changed, and can be later recovered with the **pool-create** command.

Example 20.84. How to stop a storage pool

The following example stops the *vdisk* storage pool that you built in Example 20.79, “How to create a storage pool from an XML file”:

```
# virsh pool-destroy vdisk

Pool vdisk destroyed
```

The **virsh pool-delete *pool*** command destroys the resources used by the specified storage pool. It is important to note that this operation is non-recoverable and non-reversible. However, the pool structure will still exist after this command, ready to accept the creation of new storage volumes.

Example 20.85. How to delete a storage pool

The following sample deletes the *vdisk* storage pool that you built in Example 20.79, “How to create a storage pool from an XML file”.

```
# virsh pool-delete vdisk

Pool vdisk deleted
```

The **`virsh pool-undefine pool`** command undefines the configuration for an inactive pool.

Example 20.86. How to undefine a storage pool

The following examples undefines the *vdisk* storage pool that you built in Example 20.79, “How to create a storage pool from an XML file”. This makes your storage pool transient.

```
# virsh pool-undefine vdisk

Pool vdisk undefined
```

20.29.8. Creating an XML Dump File for a Pool

The **`virsh pool-dumpxml pool`** command returns the XML information about the specified storage pool object. Using the option **`--inactive`** dumps the configuration that will be used on next start of the pool instead of the current pool configuration.

Example 20.87. How to retrieve a storage pool's configuration settings

The following example retrieves the configuration settings for the *vdisk* storage pool that you built in Example 20.79, “How to create a storage pool from an XML file”. Once the command runs, the configuration file opens in the terminal:

```
# virsh pool-dumpxml vdisk
<pool type="dir">
 <name>vdisk</name>
 <target>
         <path>/var/lib/libvirt/images</path>
 </target>
</pool>
```

20.29.9. Editing the Storage Pool's Configuration File

The **`pool-edit pool`** command opens the specified storage pool's XML configuration file for editing.

This method is the only method that should be used to edit an XML configuration file as it does error checking before applying.

Example 20.88. How to edit a storage pool's configuration settings

The following example edits the configuration settings for the *vdisk* storage pool that you built in Example 20.79, “How to create a storage pool from an XML file”. Once the command runs, the configuration file opens in your default editor:

```
# virsh pool-edit vdisk
<pool type="dir">
 <name>vdisk</name>
 <target>
         <path>/var/lib/libvirt/images</path>
 </target>
</pool>
```

20.30. STORAGE VOLUME COMMANDS

This section covers commands for creating, deleting, and managing storage volumes. Creating a storage volume requires at least one storage pool. For an example on how to create a storage pool see Example 20.79, “How to create a storage pool from an XML file” For information on storage pools see Section 13.2, “Using Storage Pools”. For information on storage volumes see, Section 13.3, “Using Storage Volumes”.

20.30.1. Creating Storage Volumes

The **virsh vol-create-from *pool file vol*** command creates a volume, using another volume as input. This command requires either a storage pool name or storage pool UUID, and accepts requires the following parameters and options:

- **--pool *string*** - required - Contains the name of the storage pool or the storage pool's UUID which will be attached to the storage volume. This storage pool does not have to be the same storage pool that is associated with the storage volume you are using to base this new storage volume on.
- **--file *string*** - required - Contains the name of the XML file that contains the parameters for the storage volume.
- **--vol *string*** - required - Contains the name of the storage volume you are using to base this new storage volume on.
- **--inputpool *string*** - optional - Allows you to name the storage pool that is associated with the storage volume that you are using as input for the new storage volume.
- **--prealloc-metadata** - optional - preallocates metadata (for qcow2 instead of full allocation) for the new storage volume.

For examples, see Section 13.3.2, “Creating Storage Volumes”.

20.30.2. Creating a Storage Volume from Parameters

The **virsh vol-create-as *pool name capacity*** command creates a volume from a set of arguments. The *pool* argument contains the name or UUID of the storage pool to create the volume in. This command takes the following required parameters and options:

- **[--pool] *string*** - required - Contains the name of the associated storage pool.

- **[--name]** ***string*** - required - Contains the name of the new storage volume.
- **[--capacity]** ***string*** - required - Contains the size of the storage volume, expressed as an integer. The default is bytes, unless specified. Use the suffixes b, k, M, G, T for byte, kilobyte, megabyte, gigabyte, and terabyte, respectively.
- **--allocation** ***string*** - optional - Contains the initial allocation size, expressed as an integer. The default is bytes, unless specified.
- **--format** ***string*** - optional - Contains the file format type. Acceptable types include: raw, bochs, qcow, qcow2, qed, host_device, and vmdk. These are, however, only meant for file-based storage pools. By default the qcow version that is used is version 3. If you want to change the version, see Section 23.20.2, “Setting Target Elements”.
- **--backing-vol** ***string*** - optional - Contains the backing volume. This will be used if you are taking a snapshot.
- **--backing-vol-format** ***string*** - optional - Contains the format of the backing volume. This will be used if you are taking a snapshot.
- **--prealloc-metadata** - optional - Allows you to preallocate metadata (for qcow2 instead of full allocation).

Example 20.89. How to create a storage volume from a set of parameters

The following example creates a 100MB storage volume named *vol-new*. It contains the *vdisk*storage pool that you created in Example 20.79, “How to create a storage pool from an XML file”:

```
# virsh vol-create-as vdisk vol-new 100M


vol vol-new created
```

20.30.3. Creating a Storage Volume from an XML File

The **virsh vol-create** ***pool file*** command creates a new storage volume from an XML file which contains the storage volume parameters.

Example 20.90. How to create a storage volume from an existing XML file

The following example creates a storage volume-based on the file *vol-new.xml*, as shown:

```
<volume>
 <name>vol-new</name>
 <allocation>0</allocation>
 <capacity unit="M">100</capacity>
 <target>
  <path>/var/lib/libvirt/images/vol-new</path>
  <permissions>
   <owner>107</owner>
   <group>107</group>
              <mode>0744</mode>
              <label>virt_image_t</label>
```

```
            </permissions>
        </target>
</volume>
```

The storage volume is associated with the storage pool *vdisk*. The path to the image is **/var/lib/libvirt/images/vol-new**:

```
# virsh vol-create vdisk vol-new.xml

vol vol-new created
```

20.30.4. Cloning a Storage Volume

The **virsh vol-clone *vol-name new-vol-name*** command clones an existing storage volume. Although the **virsh vol-create-from** command may also be used, it is not the recommended way to clone a storage volume. The command accepts the **--pool *string*** option, which allows you to specify the storage pool that is associated to the new storage volume. The *vol* argument is the name or key or path of the source storage volume and the *name* argument refers to the name of the new storage volume. For additional information, see Section 13.3.2.1, “Creating Storage Volumes with virsh”.

Example 20.91. How to clone a storage volume

The following example clones a storage volume named *vol-new* to a new volume named *vol-clone*:

```
# virsh vol-clone vol-new vol-clone


vol vol-clone cloned from vol-new
```

20.31. DELETING STORAGE VOLUMES

The **virsh vol-delete *vol* pool** command deletes a given volume. The command requires a the name or UUID of the storage pool the volume is in as well as the name of the storage volume. In lieu of the volume name the key or path of the volume to delete may also be used.

Example 20.92. How to delete a storage volume

The following example deletes a storage volume named *new-vol*, which contains the storage pool *vdisk*:

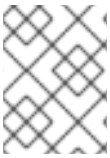
```
# virsh vol-delete new-vol vdisk

vol new-vol deleted
```

20.32. DELETING A STORAGE VOLUME'S CONTENTS

The **virsh vol-wipe *vol pool*** command wipes a volume, to ensure data previously on the volume

is not accessible to future reads. The command requires a **--pool *pool*** which is the name or UUID of the storage pool the volume is in as well as *pool* which is the name the name or key or path of the volume to wipe. Note that it is possible to choose different wiping algorithms instead of re-writing volume with zeroes, using the argument **--algorithm** and using one of the following supported algorithm types:

- **zero** - 1-pass all zeroes
- **nnsa** - 4-pass NNSA Policy Letter NAP-14.1-C (XVI-8) for sanitizing removable and non-removable hard disks: random x2, 0x00, verify.
- **dod** - 4-pass DoD 5220.22-M section 8-306 procedure for sanitizing removable and non-removable rigid disks: random, 0x00, 0xff, verify.
- **bsi** - 9-pass method recommended by the German Center of Security in Information Technologies (http://www.bsi.bund.de): 0xff, 0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0xbf, 0x7f.
- **gutmann** - The canonical 35-pass sequence described in Gutmann's paper.
- **schneier** - 7-pass method described by Bruce Schneier in "Applied Cryptography" (1996): 0x00, 0xff, random x5.
- **pfitzner7** - Roy Pfitzner's 7-random-pass method: random x7
- **pfitzner33** - Roy Pfitzner's 33-random-pass method: random x33.
- **random** - 1-pass pattern: random.s



NOTE

The availability of algorithms may be limited by the version of the "scrub" binary installed on the host.

Example 20.93. How to delete a storage volume's contents (How to wipe the storage volume)

The following example wipes the contents of the storage volume *new-vol*, which has the storage pool *vdisk* associated with it:

```
# virsh vol-wipe new-vol vdisk

vol new-vol wiped
```

20.33. DUMPING STORAGE VOLUME INFORMATION TO AN XML FILE

The **virsh vol-dumpxml *vol*** command takes the volume information, creates an XML file with the contents and outputs it to the settings that are set on the stdout stream. Optionally, you can supply the name of the associated storage pool using the **--pool** option.

Example 20.94. How to dump the contents of a storage volume

The following example dumps the contents of the storage volume named *vol-new* into an XML file:

```
# virsh vol-dumpxml vol-new
```

20.34. LISTING VOLUME INFORMATION

The **virsh vol-info *vol*** command lists basic information about the given storage volume. You must supply either the storage volume name, key, or path. The command also accepts the option **--pool**, where you can specify the storage pool that is associated with the storage volume. You can either supply the pool name, or the UUID.

Example 20.95. How to view information about a storage volume

The following example retrieves information about the storage volume named *vol-new*. When you run this command you should change the name of the storage volume to the name of your storage volume:

```
# virsh vol-info vol-new
```

The **virsh vol-list *pool*** command lists all of volumes that are associated to a given storage pool. This command requires a name or UUID of the storage pool. The **--details** option instructs **virsh** to additionally display volume type and capacity related information where available.

Example 20.96. How to display the storage pools that are associated with a storage volume

The following example lists all storage volumes that are associated with the storage pool *vdisk*:

```
# virsh vol-list vdisk
```

20.35. RETRIEVING STORAGE VOLUME INFORMATION

The **virsh vol-pool *vol*** command returns the pool name or UUID for a given storage volume. By default, the storage pool name is returned. If the **--uuid** option is used, the pool UUID is returned instead. The command requires the key or path of the storage volume for which to return the requested information.

Example 20.97. How to display the storage volume's name or UUID

The following examples retrieves the name for the storage volume that is found in the path **/var/lib/libvirt/images/vol-new**:

```
# virsh vol-pool /var/lib/libvirt/images/vol-new

vol-new
```

The **vol-path --pool *pool-or-uuid vol-name-or-key*** command returns the path for a given volume. The command requires **--pool *pool-or-uuid***, which is the name or UUID of the storage pool the volume is in. It also requires *vol-name-or-key* which is the name or key of the volume for which

the path has been requested.

The **vol-name *vol-key-or-path*** command returns the name for a given volume, where *vol-key-or-path* is the key or path of the volume to return the name for.

The **vol-key --pool *pool-or-uuid vol-name-or-path*** command returns the volume key for a given volume where **--pool *pool-or-uuid*** is the name or UUID of the storage pool the volume is in and *vol-name-or-path* is the name or path of the volume to return the volume key for.

20.36. DISPLAYING PER-GUEST VIRTUAL MACHINE INFORMATION

20.36.1. Displaying the Guest Virtual Machines

To display a list of active guest virtual machines and their current states with **virsh**:

```
# virsh list
```

Other options available include:

- **--all** - Lists all guest virtual machines. For example:

```
# virsh list --all
 Id Name                 State
----------------------------------
  0 Domain-0             running
  1 Domain202            paused
  2 Domain010            shut off
  3 Domain9600           crashed
```

NOTE

If no results are displayed when running **virsh list --all**, it is possible that you did not create the virtual machine as the root user.

The **virsh list --all** command recognizes the following states:

- *running* - The **running** state refers to guest virtual machines that are currently active on a CPU.
- *idle* - The **idle** state indicates that the guest virtual machine is idle, and may not be running or able to run. This can occur when the guest virtual machine is waiting on I/O (a traditional wait state) or has gone to sleep because there was nothing else for it to do.
- *paused* - When a guest virtual machine is paused, it consumes memory and other resources, but it is not eligible for scheduling CPU resources from the hypervisor. The **paused** state occurs after using the **paused** button in **virt-manager** or the **virsh suspend** command.
- *in shutdown* - The **in shutdown** state is for guest virtual machines in the process of shutting down. The guest virtual machine is sent a shutdown signal and should be in the process of stopping its operations gracefully. This may not work with all guest virtual machine operating systems; some operating systems do not respond to these signals.

- *shut off* - The **shut off** state indicates that the guest virtual machine is not running. This can be caused when a guest virtual machine completely shuts down or has not been started.
- *crashed* - The **crashed** state indicates that the guest virtual machine has crashed and can only occur if the guest virtual machine has been configured not to restart on crash.
- *pmsuspended* - The guest has been suspended by guest power management.

- **--inactive** - Lists guest virtual machines that have been defined but are not currently active. This includes machines that are **shut off** and **crashed**.
- **--managed-save** - Guests that have managed save state enabled will be listed as **saved**. Note that to filter guests with this option, you also need to use the **--all** or **--inactive** options.
- **--name** - The command lists the names of the guests instead of the default table format. This option is mutually exclusive with the **--uuid** option, which only prints a list of guest UUIDs, and with the **--table** option, which determines that the table style output should be used.
- **--title** - Lists also the guest **title** field, which typically contains a short description of the guest. This option must be used with the default (**--table**) output format. For example:

```
$ virsh list --title

Id       Name                                           State
Title
-------------------------------------------------------------------
------------
0        Domain-0                                       running
Mailserver1
2        rhelvm                                         paused
```

- **--persistent** - Only persistent guests are included in a list. Use the **--transient** argument to list transient guests.
- **--with-managed-save** - Lists guests that have been configured with a managed save. To list the guests without one, use the **--without-managed-save** option.
- **--state-running** - Lists only guests that are running. Similarly, use **--state-paused** for paused guests, **--state-shutoff** for guests that are turned off, and **--state-other** lists all states as a fallback.
- **--autostart** - Only auto-starting guests are listed. To list guests with this feature disabled, use the argument **--no-autostart**.
- **--with-snapshot** - Lists the guests whose snapshot images can be listed. To filter for guests without a snapshot, use the **--without-snapshot** option.

20.36.2. Displaying Virtual CPU Information

To display virtual CPU information from a guest virtual machine with **virsh**:

```
# virsh vcpuinfo {domain-id, domain-name or domain-uuid}
```

An example of **virsh vcpuinfo** output:

```
# virsh vcpuinfo guest1
VCPU:           0
CPU:            2
State:          running
CPU time:       7152.4s
CPU Affinity:   yyyy

VCPU:           1
CPU:            2
State:          running
CPU time:       10889.1s
CPU Affinity:   yyyy
```

20.36.3. Pinning vCPU to a Host Physical Machine's CPU

The **virsh vcpupin** command assigns a virtual CPU to a physical one.

```
# virsh vcpupin guest1
VCPU: CPU Affinity
----------------------------------
   0: 0-3
   1: 0-3
```

The **vcpupin** command can take the following arguments:

- **--vcpu** requires the vcpu number
- **[--cpulist]** ***string*** lists the host physical machine's CPU number(s) to set, or omit option to query
- **--config** affects next boot
- **--live** affects the running guest virtual machine
- **--current** affects the current guest virtual machine state

20.36.4. Displaying Information about the Virtual CPU Counts of a Given Domain

The **virsh vcpucount** command requires a *domain* name or a domain ID

```
# virsh vcpucount guest1
maximum      config         2
maximum      live           2
current      config         2
current      live           2
```

The **vcpucount** can take the following arguments:

- **--maximum** get maximum cap on vcpus
- **--active** get number of currently active vcpus

- **--live** get value from running guest virtual machine
- **--config** get value to be used on next boot
- **--current** get value according to current guest virtual machine state
- **--guest** count that is returned is from the perspective of the guest

20.36.5. Configuring Virtual CPU Affinity

To configure the affinity of virtual CPUs with physical CPUs:

```
# virsh vcpupin domain-id vcpu cpulist
```

The **domain-id** parameter is the guest virtual machine's ID number or name.

The **vcpu** parameter denotes the number of virtualized CPUs allocated to the guest virtual machine.The **vcpu** parameter must be provided.

The **cpulist** parameter is a list of physical CPU identifier numbers separated by commas. The **cpulist** parameter determines which physical CPUs the VCPUs can run on.

Additional parameters such as **--config** effect the next boot, whereas **--live** effects the running guest virtual machine and **--current** affects the current guest virtual machine state.

20.36.6. Configuring Virtual CPU Count

Use this command to change the number of virtual CPUs active in a guest virtual machine. By default, this command works on active guest virtual machines. To change the inactive settings that will be used the next time a guest virtual machine is started, use the **--config** flag.To modify the number of CPUs assigned to a guest virtual machine with **virsh**:

```
# virsh setvcpus {domain-name, domain-id or domain-uuid} count [[--config]
[--live] | [--current]] [--maximum] [--guest]
```

For example:

```
# virsh setvcpus guestVM1 2 --live
```

will set the number of vCPUs to guestVM1 to two and this action will be performed while the guestVM1 is running.

IMPORTANT

Hot unplugging vCPUs is not supported on Red Hat Enterprise Linux 7.

The count value may be limited by host, hypervisor, or a limit coming from the original description of the guest virtual machine.

If the **--config** flag is specified, the change is made to the stored XML configuration for the guest virtual machine, and will only take effect when the guest is started.

If **--live** is specified, the guest virtual machine must be active, and the change takes place immediately. This option will allow hot plugging of a vCPU. Both the **--config** and **--live** flags may be specified together if supported by the hypervisor.

If **--current** is specified, the flag affects the current guest virtual machine state.

When no flags are specified, the **--live** flag is assumed. The command will fail if the guest virtual machine is not active. In addition, if no flags are specified, it is up to the hypervisor whether the **--config** flag is also assumed. This determines whether the XML configuration is adjusted to make the change persistent.

The **--maximum** flag controls the maximum number of virtual CPUs that can be hot-plugged the next time the guest virtual machine is booted. Therefore, it can only be used with the **--config** flag, not with the **--live** flag.

Note that ***count*** cannot exceed the number of CPUs assigned to the guest virtual machine.

If **--guest** is specified, the flag modifies the CPU state in the current guest virtual machine.

20.36.7. Configuring Memory Allocation

To modify a guest virtual machine's memory allocation with **virsh**:

```
# virsh setmem {domain-id or domain-name} count
```

For example:

```
# virsh setmem vr-rhel6u1-x86_64-kvm --kilobytes 1025000
```

You must specify the **count** in kilobytes. The new count value cannot exceed the amount you specified for the guest virtual machine. Values lower than 64 MB are unlikely to work with most guest virtual machine operating systems. A higher maximum memory value does not affect active guest virtual machines. If the new value is lower than the available memory, it will shrink possibly causing the guest virtual machine to crash.

This command has the following options

- *domain* - specified by a domain name, id, or uuid
- *size* - Determines the new memory size, as a scaled integer. The default unit is KiB, but a different one can be specified:

  Valid memory units include:

  - ***b*** or ***bytes*** for bytes
  - ***KB*** for kilobytes (10^3 or blocks of 1,000 bytes)
  - ***k*** or ***KiB*** for kibibytes (2^{10} or blocks of 1024 bytes)
  - ***MB*** for megabytes (10^6 or blocks of 1,000,000 bytes)
  - ***M*** or ***MiB*** for mebibytes (2^{20} or blocks of 1,048,576 bytes)

- ***GB*** for gigabytes (10^9 or blocks of 1,000,000,000 bytes)
- ***G*** or ***GiB*** for gibibytes (2^{30} or blocks of 1,073,741,824 bytes)
- ***TB*** for terabytes (10^{12} or blocks of 1,000,000,000,000 bytes)
- ***T*** or ***TiB*** for tebibytes (2^{40} or blocks of 1,099,511,627,776 bytes)

Note that all values will be rounded up to the nearest kibibyte by libvirt, and may be further rounded to the granularity supported by the hypervisor. Some hypervisors also enforce a minimum, such as 4000KiB (or 4000 x 2^{10} or 4,096,000 bytes). The units for this value are determined by the optional attribute ***memory unit***, which defaults to the kibibytes (KiB) as a unit of measure where the value given is multiplied by 2^{10} or blocks of 1024 bytes.

- **--config** - the command takes effect on the next boot
- **--live** - the command controls the memory of a running guest virtual machine
- **--current** - the command controls the memory on the current guest virtual machine

20.36.8. Changing the Memory Allocation for the Domain

The **virsh setmaxmem *domain size* --config --live --current** command allows the setting of the maximum memory allocation for a guest virtual machine as shown:

```
# virsh setmaxmem guest1 1024 --current
```

The size that can be given for the maximum memory is a scaled integer that by default is expressed in kibibytes, unless a supported suffix is provided. The following arguments can be used with this command:

- **--config** - takes affect next boot
- **--live** - controls the memory of the running guest virtual machine, providing the hypervisor supports this action as not all hypervisors allow live changes of the maximum memory limit.
- **--current** - controls the memory on the current guest virtual machine

20.36.9. Displaying Guest Virtual Machine Block Device Information

Use the **virsh domblkstat** command to display block device statistics for a running guest virtual machine. Use the **--human** to display the statistics in a more user friendly way.

```
# virsh domblkstat GuestName block-device
```

20.36.10. Displaying Guest Virtual Machine Network Device Information

Use the **virsh domifstat** command to display network interface statistics for a running guest virtual machine.

```
# virsh domifstat GuestName interface-device
```

20.37. MANAGING VIRTUAL NETWORKS

This section covers managing virtual networks with the **virsh** command. To list virtual networks:

```
# virsh net-list
```

This command generates output similar to:

```
# virsh net-list
Name                 State      Autostart
-----------------------------------------
default              active     yes
vnet1              active     yes
vnet2              active     yes
```

To view network information for a specific virtual network:

```
# virsh net-dumpxml NetworkName
```

This displays information about a specified virtual network in XML format:

```
# virsh net-dumpxml vnet1
<network>
  <name>vnet1</name>
  <uuid>98361b46-1581-acb7-1643-85a412626e70</uuid>
  <forward dev='eth0'/>
  <bridge name='vnet0' stp='on' forwardDelay='0' />
  <ip address='192.168.100.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.100.128' end='192.168.100.254' />
    </dhcp>
  </ip>
</network>
```

Other **virsh** commands used in managing virtual networks are:

- **virsh net-autostart *network-name*** : Marks a *network-name* to be started automatically when the **libvirt** daemon starts. The **--disable** option un-marks the *network-name*.
- **virsh net-create *XMLfile*** : Starts a new (transient) network using an XML definition from an existing file.
- **virsh net-define *XMLfile*** : Defines a new network using an XML definition from an existing file without starting it.
- **virsh net-destroy *network-name*** : Destroys a network specified as *network-name*.
- **virsh net-name *networkUUID*** : Converts a specified *networkUUID* to a network name.
- **virsh net-uuid *network-name*** : Converts a specified *network-name* to a network UUID.
- **virsh net-start *nameOfInactiveNetwork*** : Starts an inactive network.
- **virsh net-undefine *nameOfInactiveNetwork*** : Removes the inactive XML definition of

a network. This has no effect on the network state. If the domain is running when this command is executed, the network continues running. However, the network becomes transient instead of persistent.

libvirt has the capability to define virtual networks which can then be used by domains and linked to actual network devices. For more detailed information about this feature see the documentation at libvirt upstream website . Many of the commands for virtual networks are similar to the ones used for domains, but the way to name a virtual network is either by its name or UUID.

20.37.1. Autostarting a Virtual Network

The **virsh net-autostart** command configures a virtual network to be started automatically when the guest virtual machine boots.

```
# virsh net-autostart network [--disable]
```

This command accepts the **--disable** option, which disables the autostart command.

20.37.2. Creating a Virtual Network from an XML File

The **virsh net-create** command creates a virtual network from an XML file. To get a description of the XML network format used by libvirt, see the libvirt upstream website. In this command *file* is the path to the XML file. To create the virtual network from an XML file, run:

```
# virsh net-create file
```

20.37.3. Defining a Virtual Network from an XML File

The **virsh net-define** command defines a virtual network from an XML file, the network is just defined but not instantiated.

```
# virsh net-define file
```

20.37.4. Stopping a Virtual Network

The **virsh net-destroy** command destroys (stops) a given virtual network specified by its name or UUID. This takes effect immediately. To stop the specified network *network* is required.

```
# virsh net-destroy network
```

20.37.5. Creating a Dump File

The **virsh net-dumpxml** command outputs the virtual network information as an XML dump to stdout for the specified virtual network. If **--inactive** is specified, physical functions are not expanded into their associated virtual functions.

```
# virsh net-dumpxml network [--inactive]
```

20.37.6. Editing a Virtual Network's XML Configuration File

The following command edits the XML configuration file for a network:

```
# virsh net-edit network
```

The editor used for editing the XML file can be supplied by the $VISUAL or $EDITOR environment variables, and defaults to **vi**.

20.37.7. Getting Information about a Virtual Network

The **virsh net-info** returns basic information about the *network* object.

```
# virsh net-info network
```

20.37.8. Listing Information about a Virtual Network

The **virsh net-list** command returns the list of active networks. If **--all** is specified this will also include defined but inactive networks. If **--inactive** is specified only the inactive ones will be listed. You may also want to filter the returned networks by **--persistent** to list the persistent ones, **--transient** to list the transient ones, **--autostart** to list the ones with autostart enabled, and **--no-autostart** to list the ones with autostart disabled.

Note: When talking to older servers, this command is forced to use a series of API calls with an inherent race, where a pool might not be listed or might appear more than once if it changed state between calls while the list was being collected. Newer servers do not have this problem.

To list the virtual networks, run:

```
# virsh net-list [--inactive | --all] [--persistent] [<--transient>] [--
autostart] [<--no-autostart>]
```

20.37.9. Converting a Network UUID to Network Name

The **virsh net-name** command converts a network UUID to network name.

```
# virsh net-name network-UUID
```

20.37.10. Converting a Network Name to Network UUID

The **virsh net-uuid** command converts a network name to network UUID.

```
# virsh net-uuid network-name
```

20.37.11. Starting a Previously Defined Inactive Network

The **virsh net-start** command starts a (previously defined) inactive network.

```
# virsh net-start network
```

20.37.12. Undefining the Configuration for an Inactive Network

The **virsh net-undefine** command undefines the configuration for an inactive network.

```
# virsh net-undefine network
```

20.37.13. Updating an Existing Network Definition File

```
# virsh net-update network directive section XML [--parent-index index]
[[--live] [--config] | [--current]]
```

The **virsh net-update** command updates a specified section of an existing network definition by issuing one of the following *directives* to the section:

- **add-first**
- **add-last** or **add** (these are synonymous)
- **delete**
- **modify**

The *section* can be one of the following:

- **bridge**
- **domain**
- **ip**
- **ip-dhcp-host**
- **ip-dhcp-range**
- **forward**
- **forward interface**
- **forward-pf**
- **portgroup**
- **dns-host**
- **dns-txt**
- **dns-srv**

Each section is named by a concatenation of the XML element hierarchy leading to the element that is changed. For example, **ip-dhcp-host** changes a **<host>** element that is contained inside a **<dhcp>** element inside an **<ip>** element of the network.

XML is either the text of a complete XML element of the type being changed (for instance, **<host mac="00:11:22:33:44:55' ip='1.2.3.4'/>**), or the name of a file that contains a complete XML element. Disambiguation is done by looking at the first character of the provided text - if the first character is **<**, it is XML text, if the first character is not **>**, it is the name of a file that contains the xml text to be used. The ***--parent-index*** option is used to specify which of several parent elements the requested element is in (0-based).

For example, a dhcp **<host>** element could be in any one of multiple **<ip>** elements in the network; if a parent-index is not provided, the most appropriate **<ip>** element will be selected (usually the only one that already has a **<dhcp>** element), but if ***--parent-index*** is given, that particular instance of **<ip>** will get the modification. If ***--live*** is specified, affect a running network. If ***--config*** is specified, affect the next startup of a persistent network. If ***--current*** is specified, affect the current network state. Both ***--live*** and ***--config*** flags may be given, but **--current** is exclusive. Not specifying any flag is the same as specifying ***--current***.

20.37.14. Migrating Guest Virtual Machines with virsh

Information on migration using virsh is located in the section entitled Live KVM Migration with virsh See Section 15.5, “Live KVM Migration with virsh”

20.37.15. Setting a Static IP Address for the Guest Virtual Machine

In cases where a guest virtual machine is configured to acquire its IP address from DHCP, but you still need it to have a predictable static IP address, you can use the following procedure to modify the DHCP server configuration used by libvirt. This procedure requires that you know the MAC address of the guest interface in order to make this change. Therefore, you will need to perform the operation after the guest has been created, or decide on a MAC address for the guest prior to creating it, and then set this same address manually when creating the guest virtual machine.

In addition, you should note that this procedure only works for guest interfaces that are connected to a libvirt virtual network with a forwarding mode of **"nat"**, **"route"**, or no forwarding mode at all. This procedure will not work if the network has been configured with **forward mode="bridge"** or **"hostdev"** . In those cases, the DCHP server is located elsewhere on the network, and is therefore not under control of libvirt. In this case the static IP entry would need to be made on the remote DHCP server. To do that see the documentation that is supplied with the server.

Procedure 20.5. Setting a static IP address

This procedure is performed on the host physical machine.

1. **Check the guest XML configuration file**
   Display the guest's network configuration settings by running the **virsh domiflist *guest1*** command. Substitute the name of your virtual machine in place of *guest1*. A table is displayed. Look in the Source column. That is the name of your network. In this example the network is called default. This name will be used for the rest of the procedure as well as the MAC address.

   ```
   # virsh domiflist guest1
   Interface  Type       Source     Model       MAC
   -------------------------------------------------------
   vnet4      network    default    virtio      52:54:00:48:27:1D
   ```

2. **Verify the DHCP range**
   The IP address that you set must be within the dhcp range that is specified for the network. In addition, it must also not conflict with any other existing static IP addresses on the network. To check the range of addresses available as well as addresses used, use the following command on the host machine:

   ```
   # virsh net-dumpxml default | egrep 'range|host\ mac'

   <range start='198.51.100.2' end='198.51.100.254'/>
   ```

```
<host mac='52:54:00:48:27:1C:1D' ip='198.51.100.2'/>
```

The output you see will differ from the example and you may see more lines and multiple host mac lines. Each guest static IP address will have one line.

3. **Set a static IP address**
   Use the following command on the host machine, and replace *default* with the name of the network.

```
# virsh net-update default add ip-dhcp-host '<host
mac='52:54:00:48:27:1D' ip='198.51.100.3"/>'--live --config
```

   The **--live** option allows this change to immediately take place and the **--config** option makes the change persistent. This command will also work for guest virtual machines that you have not yet created as long as you use a valid IP and MAC address. The MAC address should be a valid unicast MAC address (6 hexadecimal digit pairs separated by **:**, with the first digit pair being an even number); when libvirt creates a new random MAC address, it uses **52:54:00** for the first three digit pairs, and it is recommended to follow this convention.

4. **Restart the interface (optional)**
   If the guest virtual machine is currently running, you will need to force the guest virtual machine to re-request a DHCP address. If the guest is not running, the new IP address will be implemented the next time you start it. To restart the interface, enter the following commands on the host machine:

```
# virsh domif-setlink guest1 52:54:00:48:27:1D down
# sleep 10
# virsh domif-setlink guest1 52:54:00:48:27:1D up
```

   This command makes the guest virtual machine's operating system think that the Ethernet cable has been unplugged, and then re-plugged after ten seconds. The **sleep** command is important because many DHCP clients allow for a short disconnect of the cable without re-requesting the IP address. Ten seconds is long enough so that the DHCP client forgets the old IP address and will request a new one once the **up** command is executed. If for some reason this command fails, you will have to reset the guest's interface from the guest operating system's management interface.

20.38. INTERFACE COMMANDS

The following commands manipulate host interfaces and as such should not be run from the guest virtual machine. These commands should be run from a terminal on the host physical machine.

WARNING

The commands in this section are only supported if the machine has the NetworkManager service disabled, and is using the **network** service instead.

Often, these host interfaces can then be used by name within guest virtual machine **<interface>** elements (such as a system-created bridge interface), but there is no requirement that host interfaces be

tied to any particular guest configuration XML at all. Many of the commands for host interfaces are similar to the ones used for guest virtual machines, and the way to name an interface is either by its name or its MAC address. However, using a MAC address for an **iface** argument only works when that address is unique (if an interface and a bridge share the same MAC address, which is often the case, then using that MAC address results in an error due to ambiguity, and you must resort to a name instead).

20.38.1. Defining and Starting a Host Physical Machine Interface via an XML File

The **virsh iface-define *file*** command define a host interface from an XML file. This command will only define the interface and will not start it.

```
# virsh iface-define iface.xml
```

To start an interface which has already been defined, run **iface-start *interface***, where *interface* is the interface name.

20.38.2. Editing the XML Configuration File for the Host Interface

The command **virsh iface-edit *interface*** edits the XML configuration file for a host interface. This is the **only** recommended way to edit the XML configuration file. (For more information about these files, see Chapter 23, *Manipulating the Domain XML*.)

20.38.3. Listing Host Interfaces

The **virsh iface-list** displays a list of active host interfaces. If **--all** is specified, this list will also include interfaces that are defined but are inactive. If **--inactive** is specified only the inactive interfaces will be listed.

20.38.4. Converting a MAC Address into an Interface Name

The **virsh iface-name *interface*** command converts a host interface MAC address to an interface name, the provided MAC address is unique among the host's interfaces. This command requires *interface* which is the interface's MAC address.

The **virsh iface-mac *interface*** command will convert a host's interface name to MAC address where in this case *interface*, is the interface name.

20.38.5. Stopping and Undefining a Specific Host Physical Machine Interface

The **virsh iface-destroy *interface*** command destroys (stops) a given host interface, which is the same as running **virsh if-down** on the host. This command will disable that interface from active use and takes effect immediately.

To undefine the interface, use the **virsh iface-undefine *interface*** command along with the interface name.

20.38.6. Displaying the Host Configuration File

THe **virsh iface-dumpxml *interface* --inactive** command displays the host interface information as an XML dump to stdout. If the **--inactive** argument is specified, then the output reflects the persistent state of the interface that will be used the next time it is started.

20.38.7. Creating Bridge Devices

The **virsh iface-bridge** command creates a bridge device named *bridge*, and attaches the existing network device *interface* to the new bridge, which starts working immediately, with STP enabled and a delay of 0.

```
# virsh iface-bridge interface bridge
```

Note that these settings can be altered with the **--no-stp** option, **--no-start** option, and an number of seconds for delay. The IP address configuration of the interface will be moved to the new bridge device. For information on tearing down the bridge, see Section 20.38.8, “Tearing Down a Bridge Device”

20.38.8. Tearing Down a Bridge Device

The **virsh iface-unbridge *bridge* --no-start** command tears down a specified bridge device named *bridge*, releases its underlying interface back to normal usage, and moves all IP address configuration from the bridge device to the underlying device. The underlying interface is restarted unless **--no-start** argument is used, but keep in mind not restarting is generally not recommended. For the command to create a bridge, see Section 20.38.7, “Creating Bridge Devices”.

20.38.9. Manipulating Interface Snapshots

The **virsh iface-begin** command creates a snapshot of current host interface settings, which can later be committed (with **virsh iface-commit**) or restored (**virsh iface-rollback**). This is useful for situations where something fails when defining and starting a new host interface, and a system misconfiguration occurs. If a snapshot already exists, then this command will fail until the previous snapshot has been committed or restored. Undefined behavior will result if any external changes are made to host interfaces outside of the libvirt API between the time of the creation of a snapshot and its eventual commit or rollback.

Use the **virsh iface-commit** command to declare all changes made since the last **virsh iface-begin** as working, and then delete the rollback point. If no interface snapshot has already been started using **virsh iface-begin**, then this command will fail.

Use the **virsh iface-rollback** to revert all host interface settings back to the state that recorded the last time the **virsh iface-begin** command was executed. If **virsh iface-begin** command had not been previously executed, then **virsh iface-rollback** will fail. Note that if the host physical machine is rebooted before **virsh iface-commit** is run, an automatic rollback will be performed which will restore the host's configuration to the state it was at the time that the **virsh iface-begin** was executed. This is useful in cases where an improper change to the network configuration renders the host unreachable for purposes of undoing the change, but the host is either power-cycled or otherwise forced to reboot.

Example 20.98. An example of working with snapshots

Define and start a new host interface.

```
# virsh iface-begin
# virsh iface-define eth4-if.xml
# virsh if-start eth4
```

If something fails and the network stops running, roll back the changes.

```
# virsh iface-rollback
```

If everything works properly, commit the changes.

```
# virsh iface-commit
```

20.39. MANAGING SNAPSHOTS

The sections that follow describe actions that can be done in order to manipulate guest virtual machine snapshots. *Snapshots* take the disk, memory, and device state of a guest virtual machine at a specified point in time, and save it for future use. Snapshots have many uses, from saving a "clean" copy of an OS image to saving a guest virtual machine's state before what may be a potentially destructive operation. Snapshots are identified with a unique name. See the libvirt upstream website for documentation of the XML format used to represent properties of snapshots.

IMPORTANT

Red Hat Enterprise Linux 7 only supports creating snapshots while the guest virtual machine is paused or powered down. Creating snapshots of running guests (also known as *live snapshots*) is available on Red Hat Virtualization. For details, call your service representative.

20.39.1. Creating Snapshots

The **virsh snapshot-create** command creates a snapshot for guest virtual machine with the properties specified in the guest virtual machine's XML file (such as **<name>** and **<description>** elements, as well as **<disks>**). To create a snapshot run:

```
# virsh snapshot-create domain XML file [--redefine [--current] [--no-
metadata] [--halt] [--disk-only] [--reuse-external] [--quiesce] [--atomic]
```

The guest virtual machine name, id, or uid may be used as the guest virtual machine requirement. The XML requirement is a string that must in the very least contain the *name*, *description*, and *disks* elements.

The remaining optional arguments are as follows:

- **--disk-only** - the memory state of the guest virtual machine is not included in the snapshot.
- If the XML file string is completely omitted, libvirt will choose a value for all fields. The new snapshot will become current, as listed by snapshot-current. In addition, the snapshot will only include the disk state rather than the usual system checkpoint with guest virtual machine state. Disk snapshots are faster than full system checkpoints, but reverting to a disk snapshot may require **fsck** or journal replays, since it is like the disk state at the point when the power cord is abruptly pulled. Note that mixing **--halt** and **--disk-only** loses any data that was not flushed to disk at the time.
- **--halt** - causes the guest virtual machine to be left in an inactive state after the snapshot is created. Mixing **--halt** and **--disk-only** loses any data that was not flushed to disk at the time as well as the memory state.
- **--redefine** specifies that if all XML elements produced by **virsh snapshot-dumpxml** are

valid; it can be used to migrate snapshot hierarchy from one machine to another, to recreate hierarchy for the case of a transient guest virtual machine that goes away and is later recreated with the same name and UUID, or to make slight alterations in the snapshot metadata (such as host-specific aspects of the guest virtual machine XML embedded in the snapshot). When this flag is supplied, the **xmlfile** argument is mandatory, and the guest virtual machine's current snapshot will not be altered unless the **--current** flag is also given.

- **--no-metadata** creates the snapshot, but any metadata is immediately discarded (that is, libvirt does not treat the snapshot as current, and cannot revert to the snapshot unless **--redefine** is later used to teach libvirt about the metadata again).
- **--reuse-external**, if used and snapshot XML requests an external snapshot with a destination of an existing file, the destination must exist, and is reused; otherwise, a snapshot is refused to avoid losing contents of the existing files.
- **--quiesce** libvirt will try to freeze and unfreeze the guest virtual machine's mounted file system(s), using the guest agent. However, if the guest virtual machine does not have a guest agent, snapshot creation will fail. The snapshot can contain the memory state of the virtual guest machine. The snapshot must be external.
- **--atomic** causes libvirt to guarantee that the snapshot either succeeds, or fails with no changes. Note that not all hypervisors support this. If this flag is not specified, then some hypervisors may fail after partially performing the action, and **virsh dumpxml** must be used to see whether any partial changes occurred.

Existence of snapshot metadata will prevent attempts to undefine a persistent guest virtual machine. However, for transient guest virtual machines, snapshot metadata is silently lost when the guest virtual machine quits running (whether by a command such as **destroy** or by an internal guest action).

20.39.2. Creating a Snapshot for the Current Guest Virtual Machine

The **virsh snapshot-create-as** command creates a snapshot for guest virtual machine with the properties specified in the domain XML file (such as **name** and **description** elements). If these values are not included in the XML string, libvirt will choose a value. To create a snapshot run:

```
# snapshot-create-as domain {[--print-xml] | [--no-metadata] [--halt] [--
reuse-external]} [name] [description] [--disk-only [--quiesce]] [--atomic]
[[--memspec memspec]] [--diskspec] diskspec]
```

The remaining optional arguments are as follows:

- **--print-xml**creates appropriate XML for **snapshot-create** as output, rather than actually creating a snapshot.
- **--halt** keeps the guest virtual machine in an inactive state after the snapshot is created.
- **--disk-only** creates a snapshot that does not include the guest virtual machine state.
- **--memspec** can be used to control whether a checkpoint is internal or external. The flag is mandatory, followed by a **memspec** of the form **[file=]name[,snapshot=type]**, where type can be none, internal, or external. To include a literal comma in file=name, escape it with a second comma.
- **--diskspec** option can be used to control how **--disk-only** and external checkpoints create external files. This option can occur multiple times, according to the number of **<disk>**

elements in the domain XML. Each **<diskspec>** is in the form disk**[,snapshot=type][,driver=type][,file=name]**. If **--diskspec** is omitted for a specific disk, the default behavior in the virtual machine configuraition is used. To include a literal comma in disk or in **file=name**, escape it with a second comma. A literal **--diskspec** must precede each diskspec unless all three of *domain*, *name*, and *description* are also present. For example, a diskspec of **vda,snapshot=external,file=/path/to,,new** results in the following XML:

```
<disk name='vda' snapshot='external'>
   <source file='/path/to,new'/>
</disk>
```

IMPORTANT

Red Hat recommends the use of external snapshots, as they are more flexible and reliable when handled by other virtualization tools. To create an external snapshot, use the **virsh-create-as** command with the **--diskspec vda,snapshot=external** option

If this option is not used, **virsh** creates internal snapshots, which are not recommended for use due to their lack of stability and optimization. For more information, see Section A.13, "Workaround for Creating External Snapshots with libvirt".

- **--reuse-external** is specified, and the domain XML or diskspec option requests an external snapshot with a destination of an existing file, then the destination must exist, and is reused; otherwise, a snapshot is refused to avoid losing contents of the existing files.
- **--quiesce** is specified, libvirt will try to use guest agent to freeze and unfreeze guest virtual machine's mounted file systems. However, if domain has no guest agent, snapshot creation will fail. Currently, this requires **--disk-only** to be passed as well.
- **--no-metadata** creates snapshot data but any metadata is immediately discarded (that is,libvirt does not treat the snapshot as current, and cannot revert to the snapshot unless snapshot-create is later used to teach libvirt about the metadata again). This flag is incompatible with **--print-xml**
- **--atomic** will cause libvirt to guarantee that the snapshot either succeeds, or fails with no changes. Note that not all hypervisors support this. If this flag is not specified, then some hypervisors may fail after partially performing the action, and **virsh dumpxml** must be used to see whether any partial changes occurred.

WARNING

Creating snapshots of KVM guests running on a 64-bit ARM platform host currently does not work. Note that KVM on 64-bit ARM is provided as a Development Preview.

20.39.3. Displaying the Snapshot Currently in Use

The **virsh snapshot-current** command is used to query which snapshot is currently in use.

```
# virsh snapshot-current domain {[--name] | [--security-info] |
[snapshotname]}
```

If **snapshotname** is not used, snapshot XML for the guest virtual machine's current snapshot (if there is one) will be displayed as output. If **--name** is specified, just the current snapshot name instead of the full XML will be sent as output. If **--security-info** is supplied security sensitive information will be included in the XML. Using **snapshotname**, generates a request to make the existing named snapshot become the current snapshot, without reverting it to the guest virtual machine.

20.39.4. snapshot-edit

This command is used to edit the snapshot that is currently in use:

```
# virsh snapshot-edit domain [snapshotname] [--current] {[--rename] [--
clone]}
```

If both **snapshotname** and **--current** are specified, it forces the edited snapshot to become the current snapshot. If **snapshotname** is omitted, then **--current** must be supplied, in order to edit the current snapshot.

This is equivalent to the following command sequence below, but it also includes some error checking:

```
# virsh snapshot-dumpxml dom name > snapshot.xml
# vi snapshot.xml [note - this can be any editor]
# virsh snapshot-create dom snapshot.xml --redefine [--current]
```

If the **--rename** is specified, then the snapshot is renamed. If **--clone** is specified, then changing the snapshot name will create a clone of the snapshot metadata. If neither is specified, then the edits will not change the snapshot name. Note that changing a snapshot name must be done with care, since the contents of some snapshots, such as internal snapshots within a single qcow2 file, are accessible only from the original snapshot name.

20.39.5. snapshot-info

The **snapshot-info *domain*** command displays information about the snapshots. To use, run:

```
# snapshot-info domain {snapshot | --current}
```

Outputs basic information about a specified **snapshot** , or the current snapshot with **--current**.

20.39.6. snapshot-list

List all of the available snapshots for the given guest virtual machine, defaulting to show columns for the snapshot name, creation time, and guest virtual machine state. To use, run:

```
# virsh snapshot-list domain [{--parent | --roots | --tree}] [{[--from]
snapshot | --current} [--descendants]] [--metadata] [--no-metadata] [--
leaves] [--no-leaves] [--inactive] [--active] [--disk-only] [--internal]
[--external]
```

The optional arguments are as follows:

- **--parent** adds a column to the output table giving the name of the parent of each snapshot. This option may not be used with **--roots** or **--tree**.
- **--roots** filters the list to show only the snapshots that have no parents. This option may not be used with **--parent** or **--tree**.
- **--tree** displays output in a tree format, listing just snapshot names. This option may not be used with **--roots** or **--parent**.
- **--from** filters the list to snapshots which are children of the given snapshot or, if **--current** is provided, will cause the list to start at the current snapshot. When used in isolation or with **--parent**, the list is limited to direct children unless **--descendants** is also present. When used with **--tree**, the use of **--descendants** is implied. This option is not compatible with **--roots**. Note that the starting point of **--from** or **--current** is not included in the list unless the **--tree** option is also present.
- **--leaves** is specified, the list will be filtered to just snapshots that have no children. Likewise, if **--no-leaves** is specified, the list will be filtered to just snapshots with children. (Note that omitting both options does no filtering, while providing both options will either produce the same list or error out depending on whether the server recognizes the flags) Filtering options are not compatible with **--tree**.
- **--metadata** is specified, the list will be filtered to just snapshots that involve libvirt metadata, and thus would prevent the undefining of a persistent guest virtual machine, or be lost on destroy of a transient guest virtual machine. Likewise, if **--no-metadata** is specified, the list will be filtered to just snapshots that exist without the need for libvirt metadata.
- **--inactive** is specified, the list will be filtered to snapshots that were taken when the guest virtual machine was shut off. If **--active** is specified, the list will be filtered to snapshots that were taken when the guest virtual machine was running, and where the snapshot includes the memory state to revert to that running state. If **--disk-only** is specified, the list will be filtered to snapshots that were taken when the guest virtual machine was running, but where the snapshot includes only disk state.
- **--internal** is specified, the list will be filtered to snapshots that use internal storage of existing disk images. If --external is specified, the list will be filtered to snapshots that use external files for disk images or memory state.

20.39.7. snapshot-dumpxml

The **virsh snapshot-dumpxml *domain snapshot*** command outputs the snapshot XML for the guest virtual machine's snapshot named snapshot. To use, run:

```
# virsh snapshot-dumpxml domain snapshot [--security-info]
```

The **--security-info** option will also include security sensitive information. Use **virsh snapshot-current** to easily access the XML of the current snapshot.

20.39.8. snapshot-parent

Outputs the name of the parent snapshot, if any, for the given snapshot, or for the current snapshot with **--current**. To use, run:

```
# virsh snapshot-parent domain {snapshot | --current}
```

20.39.9. snapshot-revert

Reverts the given domain to the snapshot specified by **snapshot**, or to the current snapshot with **--current**.

WARNING

Be aware that this is a destructive action; any changes in the domain since the last snapshot was taken will be lost. Also note that the state of the domain after **snapshot-revert** is complete will be the state of the domain at the time the original snapshot was taken.

To revert the snapshot, run:

```
# virsh snapshot-revert domain {snapshot | --current} [{--running | --
paused}] [--force]
```

Normally, reverting to a snapshot leaves the domain in the state it was at the time the snapshot was created, except that a disk snapshot with no guest virtual machine state leaves the domain in an inactive state. Passing either the **--running** or **--paused** option will perform additional state changes (such as booting an inactive domain, or pausing a running domain). Since transient domains cannot be inactive, it is required to use one of these flags when reverting to a disk snapshot of a transient domain.

There are two cases where a **snapshot revert** involves extra risk, which requires the use of **--force** to proceed. One is the case of a snapshot that lacks full domain information for reverting configuration; since libvirt cannot prove that the current configuration matches what was in use at the time of the snapshot, supplying **--force** assures libvirt that the snapshot is compatible with the current configuration (and if it is not, the domain will likely fail to run). The other is the case of reverting from a running domain to an active state where a new hypervisor has to be created rather than reusing the existing hypervisor, because it implies drawbacks such as breaking any existing VNC or Spice connections; this condition happens with an active snapshot that uses a provably incompatible configuration, as well as with an inactive snapshot that is combined with the **--start** or **--pause** flag.

20.39.10. snapshot-delete

The **virsh snapshot-delete *domain*** command deletes the snapshot for the specified domain. To do this, run:

```
# virsh snapshot-delete domain {snapshot | --current} [--metadata] [{--
children | --children-only}]
```

This command deletes the snapshot for the domain named **snapshot**, or the current snapshot with **--current**. If this snapshot has child snapshots, changes from this snapshot will be merged into the

children. If the option **--children** is used, then it will delete this snapshot and any children of this snapshot. If **--children-only** is used, then it will delete any children of this snapshot, but leave this snapshot intact. These two flags are mutually exclusive.

The **--metadata** is used it will delete the snapshot's metadata maintained by libvirt, while leaving the snapshot contents intact for access by external tools; otherwise deleting a snapshot also removes its data contents from that point in time.

20.40. GUEST VIRTUAL MACHINE CPU MODEL CONFIGURATION

20.40.1. Introduction

Every hypervisor has its own policy for what a guest virtual machine will see for its CPUs by default. Whereas some hypervisors decide which CPU host physical machine features will be available for the guest virtual machine, QEMU/KVM presents the guest virtual machine with a generic model named **qemu32** or **qemu64**. These hypervisors perform more advanced filtering, classifying all physical CPUs into a handful of groups and have one baseline CPU model for each group that is presented to the guest virtual machine. Such behavior enables the safe migration of guest virtual machines between host physical machines, provided they all have physical CPUs that classify into the same group. libvirt does not typically enforce policy itself, rather it provides the mechanism on which the higher layers define their own required policy. Understanding how to obtain CPU model information and define a suitable guest virtual machine CPU model is critical to ensure guest virtual machine migration is successful between host physical machines. Note that a hypervisor can only emulate features that it is aware of and features that were created after the hypervisor was released may not be emulated.

20.40.2. Learning about the Host Physical Machine CPU Model

The **virsh capabilities** command displays an XML document describing the capabilities of the hypervisor connection and host physical machine. The XML schema displayed has been extended to provide information about the host physical machine CPU model. One of the big challenges in describing a CPU model is that every architecture has a different approach to exposing their capabilities. QEMU/KVM and **libvirt** use a scheme which combines a CPU model name string, with a set of named flags.

It is not practical to have a database listing all known CPU models, so libvirt has a small list of baseline CPU model names. It chooses the one that shares the greatest number of CPUID bits with the actual host physical machine CPU and then lists the remaining bits as named features. Notice that libvirt does not display which features the baseline CPU contains. This might seem like a flaw at first, but as will be explained in this section, it is not actually necessary to know this information.

20.40.3. Determining Support for VFIO IOMMU Devices

Use the **virsh domcapabilities** command to determine support for VFIO. See the following example output:

```
# virsh domcapabilities

[...output truncated...]

<enum name='pciBackend'>
        <value>default</value>
        <value>vfio</value>

[...output truncated...]
```

Figure 20.3. Determining support for VFIO

20.40.4. Determining a Compatible CPU Model to Suit a Pool of Host Physical Machines

Now that it is possible to find out what CPU capabilities a single host physical machine has, the next step is to determine what CPU capabilities are best to expose to the guest virtual machine. If it is known that the guest virtual machine will never need to be migrated to another host physical machine, the host physical machine CPU model can be passed straight through unmodified. A virtualized data center may have a set of configurations that can guarantee all servers will have 100% identical CPUs. Again the host physical machine CPU model can be passed straight through unmodified. The more common case, though, is where there is variation in CPUs between host physical machines. In this mixed CPU environment, the lowest common denominator CPU must be determined. This is not entirely straightforward, so libvirt provides an API for exactly this task. If libvirt is provided a list of XML documents, each describing a CPU model for a host physical machine, libvirt will internally convert these to CPUID masks, calculate their intersection, and convert the CPUID mask result back into an XML CPU description.

Here is an example of what libvirt reports as the capabilities on a basic workstation, when the **`virsh capabilities`** is executed:

```
<capabilities>
  <host>
    <cpu>
      <arch>i686</arch>
      <model>pentium3</model>
      <topology sockets='1' cores='2' threads='1'/>
      <feature name='lahf_lm'/>
      <feature name='lm'/>
      <feature name='xtpr'/>
      <feature name='cx16'/>
      <feature name='ssse3'/>
      <feature name='tm2'/>
      <feature name='est'/>
      <feature name='vmx'/>
      <feature name='ds_cpl'/>
      <feature name='monitor'/>
      <feature name='pni'/>
      <feature name='pbe'/>
      <feature name='tm'/>
      <feature name='ht'/>
      <feature name='ss'/>
      <feature name='sse2'/>
      <feature name='acpi'/>
      <feature name='ds'/>
      <feature name='clflush'/>
      <feature name='apic'/>
    </cpu>
 </host>
</capabilities>
```

Figure 20.4. Pulling host physical machine's CPU model information

Now compare that to a different server, with the same **`virsh capabilities`** command:

```
<capabilities>
  <host>
    <cpu>
      <arch>x86_64</arch>
      <model>phenom</model>
      <topology sockets='2' cores='4' threads='1'/>
      <feature name='osvw'/>
      <feature name='3dnowprefetch'/>
      <feature name='misalignsse'/>
      <feature name='sse4a'/>
      <feature name='abm'/>
      <feature name='cr8legacy'/>
      <feature name='extapic'/>
      <feature name='cmp_legacy'/>
      <feature name='lahf_lm'/>
      <feature name='rdtscp'/>
      <feature name='pdpe1gb'/>
      <feature name='popcnt'/>
      <feature name='cx16'/>
      <feature name='ht'/>
      <feature name='vme'/>
    </cpu>
    ...snip...
```

Figure 20.5. Generate CPU description from a random server

To see if this CPU description is compatible with the previous workstation CPU description, use the **virsh cpu-compare** command.

The reduced content was stored in a file named **virsh-caps-workstation-cpu-only.xml** and the **virsh cpu-compare** command can be executed on this file:

```
# virsh cpu-compare virsh-caps-workstation-cpu-only.xml
Host physical machine CPU is a superset of CPU described in virsh-caps-
workstation-cpu-only.xml
```

As seen in this output, libvirt is correctly reporting that the CPUs are not strictly compatible. This is because there are several features in the server CPU that are missing in the client CPU. To be able to migrate between the client and the server, it will be necessary to open the XML file and comment out some features. To determine which features need to be removed, run the **virsh cpu-baseline** command, on the **both-cpus.xml** which contains the CPU information for both machines. Running **# virsh cpu-baseline both-cpus.xml** results in:

```
<cpu match='exact'>
  <model>pentium3</model>
  <feature policy='require' name='lahf_lm'/>
  <feature policy='require' name='lm'/>
  <feature policy='require' name='cx16'/>
  <feature policy='require' name='monitor'/>
  <feature policy='require' name='pni'/>
  <feature policy='require' name='ht'/>
  <feature policy='require' name='sse2'/>
  <feature policy='require' name='clflush'/>
  <feature policy='require' name='apic'/>
</cpu>
```

Figure 20.6. Composite CPU baseline

This composite file shows which elements are in common. Everything that is not in common should be commented out.

20.41. CONFIGURING THE GUEST VIRTUAL MACHINE CPU MODEL

For simple defaults, the guest virtual machine CPU configuration accepts the same basic XML representation as the host physical machine capabilities XML exposes. In other words, the XML from the **`virsh cpu-baseline`** command can now be copied directly into the guest virtual machine XML at the top level under the *domain* element. In the previous XML snippet, there are a few extra attributes available when describing a CPU in the guest virtual machine XML. These can mostly be ignored, but for the curious here is a quick description of what they do. The top level **<cpu>** element has an attribute called **`match`** with possible values of:

- match='minimum' - the host physical machine CPU must have at least the CPU features described in the guest virtual machine XML. If the host physical machine has additional features beyond the guest virtual machine configuration, these will also be exposed to the guest virtual machine.
- match='exact' - the host physical machine CPU must have at least the CPU features described in the guest virtual machine XML. If the host physical machine has additional features beyond the guest virtual machine configuration, these will be masked out from the guest virtual machine.
- match='strict' - the host physical machine CPU must have exactly the same CPU features described in the guest virtual machine XML.

The next enhancement is that the <feature> elements can each have an extra 'policy' attribute with possible values of:

- policy='force' - expose the feature to the guest virtual machine even if the host physical machine does not have it. This is usually only useful in the case of software emulation.

  NOTE

  It is possible that even using the **`force`** policy, the hypervisor may not be able to emulate the particular feature.

- policy='require' - expose the feature to the guest virtual machine and fail if the host physical machine does not have it. This is the sensible default.

- policy='optional' - expose the feature to the guest virtual machine if it happens to support it.
- policy='disable' - if the host physical machine has this feature, then hide it from the guest virtual machine.
- policy='forbid' - if the host physical machine has this feature, then fail and refuse to start the guest virtual machine.

The 'forbid' policy is for a niche scenario where an incorrectly functioning application will try to use a feature even if it is not in the CPUID mask, and you wish to prevent accidentally running the guest virtual machine on a host physical machine with that feature. The 'optional' policy has special behavior with respect to migration. When the guest virtual machine is initially started the flag is optional, but when the guest virtual machine is live migrated, this policy turns into 'require', since you cannot have features disappearing across migration.

20.42. MANAGING RESOURCES FOR GUEST VIRTUAL MACHINES

virsh allows the grouping and allocation of resources on a per guest virtual machine basis. This is managed by the libvirt daemon which creates *cgroups* and manages them on behalf of the guest virtual machine. The only thing that is left for the system administrator to do is to either query or set tunables against specified guest virtual machines. The **`libvirt`** service uses the following cgroups for tuning and monitoring virtual machines:

- **`memory`** - The memory controller allows for setting limits on RAM and swap usage and querying cumulative usage of all processes in the group
- **`cpuset`** - The CPU set controller binds processes within a group to a set of CPUs and controls migration between CPUs.
- **`cpuacct`** - The CPU accounting controller provides information about CPU usage for a group of processes.
- **`cpu`** -The CPU scheduler controller controls the prioritization of processes in the group. This is similar to granting **`nice`** level privileges.
- **`devices`** - The devices controller grants access control lists on character and block devices.
- **`freezer`** - The freezer controller pauses and resumes execution of processes in the group. This is similar to **`SIGSTOP`** for the whole group.
- **`net_cls`** - The network class controller manages network utilization by associating processes with a **`tc`** network class.

cgroups are set up by **`systemd`** in **libvirt**. The following **`virsh`** tuning commands affect the way cgroups are configured:

- **`schedinfo`** - described in Section 20.43, “Setting Schedule Parameters”
- **`blkdeviotune`** - described in Section 20.44, “Disk I/O Throttling”
- **`blkiotune`** - described in Section 20.45, “Display or Set Block I/O Parameters”
- **`domiftune`** - described in Section 20.12.5, “Setting Network Interface Bandwidth Parameters”
- **`memtune`** - described in Section 20.46, “Configuring Memory Tuning”

For more information about cgroups, see the Red Hat Enterprise Linux 7 Resource Management Guide.

20.43. SETTING SCHEDULE PARAMETERS

The **virsh schedinfo** command modifies host scheduling parameters of the virtual machine process on the host machine. The following command format should be used:

```
# virsh schedinfo domain --set --current --config --live
```

Each parameter is explained below:

- *domain* - the guest virtual machine domain
- **--set** - the string placed here is the controller or action that is to be called. The string uses the *parameter=value* format. Additional parameters or values if required should be added as well.
- **--current** - when used with **--set**, will use the specified **set** string as the current scheduler information. When used without will display the current scheduler information.
- **--config** - - when used with **--set**, will use the specified **set** string on the next reboot. When used without will display the scheduler information that is saved in the configuration file.
- **--live** - when used with **--set**, will use the specified **set** string on a guest virtual machine that is currently running. When used without will display the configuration setting currently used by the running virtual machine

The scheduler can be set with any of the following parameters: ***cpu_shares***, ***vcpu_period*** and ***vcpu_quota***. These parameters are applied to the vCPU threads.

The following shows how the parameters map to cgroup field names:

- ***cpu_shares***:cpu.shares
- ***vcpu_period***:cpu.cfs_period_us
- ***vcpu_quota***:cpu.cfs_quota_us

Example 20.99. schedinfo show

This example shows the shell guest virtual machine's schedule information

```
# virsh schedinfo shell
Scheduler      : posix
cpu_shares     : 1024
vcpu_period    : 100000
vcpu_quota     : -1
```

Example 20.100. schedinfo set

In this example, the cpu_shares is changed to 2046. This effects the current state and not the configuration file.

```
# virsh schedinfo --set cpu_shares=2046 shell
```

```
Scheduler      : posix
cpu_shares     : 2046
vcpu_period    : 100000
vcpu_quota     : -1
```

libvirt also supports the ***emulator_period*** and ***emulator_quota*** parameters that modify the setting of the emulator process.

20.44. DISK I/O THROTTLING

The **virsh blkdeviotune** command sets disk I/O throttling for a specified guest virtual machine. This can prevent a guest virtual machine from over utilizing shared resources and thus impacting the performance of other guest virtual machines. The following format should be used:

```
#virsh blkdeviotune domain <device> [[--config] [--live] | [--current]]
[[total-bytes-sec] | [read-bytes-sec] [write-bytes-sec]] [[total-iops-sec]
[read-iops-sec] [write-iops-sec]]
```

The only required parameter is the domain name of the guest virtual machine. To list the domain name, run the **virsh domblklist** command. The **--config**, **--live**, and **--current** arguments function the same as in Section 20.43, “Setting Schedule Parameters”. If no limit is specified, it will query current I/O limits setting. Otherwise, alter the limits with the following flags:

- **--total-bytes-sec** - specifies total throughput limit in bytes per second.
- **--read-bytes-sec** - specifies read throughput limit in bytes per second.
- **--write-bytes-sec** - specifies write throughput limit in bytes per second.
- **--total-iops-sec** - specifies total I/O operations limit per second.
- **--read-iops-sec** - specifies read I/O operations limit per second.
- **--write-iops-sec** - specifies write I/O operations limit per second.

For more information, see the blkdeviotune section of the virsh man page. For an example domain XML see Figure 23.28, “Devices - Hard drives, floppy disks, CD-ROMs Example”.

20.45. DISPLAY OR SET BLOCK I/O PARAMETERS

The **blkiotune** command sets or displays the I/O parameters for a specified guest virtual machine. The following format should be used:

```
# virsh blkiotune domain [--weight weight] [--device-weights device-
weights] [---device-read-iops-sec -device-read-iops-sec] [--device-write-
iops-sec device-write-iops-sec] [--device-read-bytes-sec device-read-
bytes-sec] [--device-write-bytes-sec device-write-bytes-sec] [[--config]
[--live] | [--current]]
```

More information on this command can be found in the *Virtualization Tuning and Optimization Guide*

20.46. CONFIGURING MEMORY TUNING

The **virsh memtune virtual_machine --parameter size** command is covered in the *Virtualization Tuning and Optimization Guide*.

CHAPTER 21. GUEST VIRTUAL MACHINE DISK ACCESS WITH OFFLINE TOOLS

21.1. INTRODUCTION

Red Hat Enterprise Linux 7 provides a number of **`libguestfs`** utilities that enable accessing, editing, and creating guest virtual machine disks or other disk images. There are multiple uses for these tools, including:

- Viewing or downloading files located on a guest virtual machine disk.
- Editing or uploading files on a guest virtual machine disk.
- Reading or writing guest virtual machine configuration.
- Preparing new disk images containing files, directories, file systems, partitions, logical volumes and other options.
- Rescuing and repairing guest virtual machines that fail to boot or those that need boot configuration changes.
- Monitoring disk usage of guest virtual machines.
- Auditing compliance of guest virtual machines, for example to organizational security standards.
- Deploying guest virtual machines by cloning and modifying templates.
- Reading CD and DVD ISO images and floppy disk images.

WARNING

You must **never** use the utilities listed in this chapter to write to a guest virtual machine or disk image that is attached to a running virtual machine, not even to open such a disk image in write mode.

Doing so will result in disk corruption of the guest virtual machine. The tools try to prevent you from doing this, but do not secure all cases. If there is any suspicion that a guest virtual machine might be running, Red Hat strongly recommends not using the utilities.

For increased safety, certain utilities can be used in read-only mode (using the **`--ro`** option), which does not save the changes.

NOTE

The primary source for documentation for libguestfs and the related utilities are the Linux man pages. The API is documented in *guestfs(3)*, guestfish is documented in *guestfish(1)*, and the virtualization utilities are documented in their own man pages (such as *virt-df(1)*). For troubleshooting information, see Section A.17, “libguestfs Troubleshooting”

21.1.1. Caution about Using Remote Connections

Some virtualization commands in Red Hat Enterprise Linux 7 allow you to specify a remote libvirt connection. For example:

```
# virt-df -c  qemu://remote/system -d Guest
```

However, libguestfs utilities in Red Hat Enterprise Linux 7 cannot access the disks of remote libvirt guests, and commands using remote URLs as shown above do not work as expected.

Nevertheless, beginning with Red Hat Enterprise Linux 7, libguestfs can access remote disk sources over network block device (NBD). You can export a disk image from a remote machine using the **qemu-nbd** command, and access it using a **nbd://** URL. You may need to open a port on your firewall (port 10809) as shown here:

On the remote system: **qemu-nbd -t disk.img**

On the local system: **virt-df -a nbd://remote**

The following libguestfs commands are affected:

- guestfish
- guestmount
- virt-alignment-scan
- virt-cat
- virt-copy-in
- virt-copy-out
- virt-df
- virt-edit
- virt-filesystems
- virt-inspector
- virt-ls
- virt-rescue
- virt-sysprep
- virt-tar-in

- virt-tar-out
- virt-win-reg

21.2. TERMINOLOGY

This section explains the terms used throughout this chapter.

- **libguestfs (GUEST FileSystem LIBrary)** - the underlying C library that provides the basic functionality for opening disk images, reading and writing files, and so on. You can write C programs directly to this API.
- **guestfish (GUEST Filesystem Interactive SHell)** is an interactive shell that you can use from the command line or from shell scripts. It exposes all of the functionality of the libguestfs API.
- Various virt tools are built on top of libguestfs, and these provide a way to perform specific single tasks from the command line. These tools include **virt-df**, **virt-rescue**, **virt-resize**, and **virt-edit**.
- **augeas** is a library for editing the Linux configuration files. Although this is separate from libguestfs, much of the value of libguestfs comes from the combination with this tool.
- **guestmount** is an interface between libguestfs and FUSE. It is primarily used to mount file systems from disk images on your host physical machine. This functionality is not necessary, but can be useful.

21.3. INSTALLATION

To install libguestfs, guestfish, the libguestfs tools, and guestmount, enter the following command:

```
# yum install libguestfs libguestfs-tools
```

To install every libguestfs-related package including the language bindings, enter the following command:

```
# yum install '*guestf*'
```

21.4. THE GUESTFISH SHELL

guestfish is an interactive shell that you can use from the command line or from shell scripts to access guest virtual machine file systems. All of the functionality of the libguestfs API is available from the shell.

To begin viewing or editing a virtual machine disk image, enter the following command, substituting the path to your intended disk image:

```
$ guestfish --ro -a /path/to/disk/image
```

--ro means that the disk image is opened read-only. This mode is always safe but does not allow write access. Only omit this option when you are **certain** that the guest virtual machine is not running, or the disk image is not attached to a live guest virtual machine. It is not possible to use **libguestfs** to edit a live guest virtual machine, and attempting to will result in irreversible disk corruption.

/**path**/**to**/**disk**/**image** is the path to the disk. This can be a file, a host physical machine logical volume (such as /dev/VG/LV), a host physical machine device (/dev/cdrom) or a SAN LUN (/dev/sdf3).

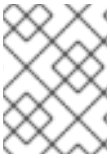
NOTE

libguestfs and guestfish do not require root privileges. You only need to run them as root if the disk image being accessed needs root to read or write or both.

When you start guestfish interactively, it will display this prompt:

```
$ guestfish --ro -a /path/to/disk/image

Welcome to guestfish, the guest filesystem shell for
editing virtual machine filesystems and disk images.

 Type: 'help' for help on commands
       'man' to read the manual
       'quit' to quit the shell

><fs>
```

At the prompt, type **run** to initiate the library and attach the disk image. This can take up to 30 seconds the first time it is done. Subsequent starts will complete much faster.



NOTE

libguestfs will use hardware virtualization acceleration such as KVM (if available) to speed up this process.

Once the **run** command has been entered, other commands can be used, as the following section demonstrates.

21.4.1. Viewing File Systems with guestfish

This section provides information on viewing file systems with guestfish.

21.4.1.1. Manual Listing and Viewing

The **list-filesystems** command will list file systems found by libguestfs. This output shows a Red Hat Enterprise Linux 4 disk image:

```
><fs> run
><fs> list-filesystems
/dev/vda1: ext3
/dev/VolGroup00/LogVol00: ext3
/dev/VolGroup00/LogVol01: swap
```

Other useful commands are **list-devices**, **list-partitions**, **lvs**, **pvs**, **vfs-type** and **file**. You can get more information and help on any command by typing **help** *command*, as shown in the following output:

```
><fs> help vfs-type
 NAME
    vfs-type - get the Linux VFS type corresponding to a mounted device

 SYNOPSIS
```

```
     vfs-type mountable

 DESCRIPTION
    This command gets the filesystem type corresponding to the filesystem
on
    "device".

    For most filesystems, the result is the name of the Linux VFS module
    which would be used to mount this filesystem if you mounted it without
    specifying the filesystem type. For example a string such as "ext3" or
    "ntfs".
```

To view the actual contents of a file system, it must first be mounted.

You can use guestfish commands such as **ls**, **ll**, **cat**, **more**, **download** and **tar-out** to view and download files and directories.

NOTE

There is no concept of a current working directory in this shell. Unlike ordinary shells, you cannot for example use the **cd** command to change directories. All paths must be fully qualified starting at the top with a forward slash (/) character. Use the *Tab* key to complete paths.

To exit from the guestfish shell, type **exit** or enter **Ctrl+d**.

21.4.1.2. Via guestfish inspection

Instead of listing and mounting file systems by hand, it is possible to let guestfish itself inspect the image and mount the file systems as they would be in the guest virtual machine. To do this, add the **-i** option on the command line:

```
$ guestfish --ro -a /path/to/disk/image -i

Welcome to guestfish, the guest filesystem shell for
editing virtual machine filesystems and disk images.

 Type: 'help' for help on commands
       'man' to read the manual
       'quit' to quit the shell

 Operating system: Red Hat Enterprise Linux AS release 4 (Nahant Update 8)
 /dev/VolGroup00/LogVol00 mounted on /
 /dev/vda1 mounted on /boot

 ><fs> ll /
 total 210
 drwxr-xr-x. 24 root root  4096 Oct 28 09:09 .
 drwxr-xr-x  21 root root  4096 Nov 17 15:10 ..
 drwxr-xr-x.  2 root root  4096 Oct 27 22:37 bin
 drwxr-xr-x.  4 root root  1024 Oct 27 21:52 boot
 drwxr-xr-x.  4 root root  4096 Oct 27 21:21 dev
 drwxr-xr-x. 86 root root 12288 Oct 28 09:09 etc
 ...
```

Because guestfish needs to start up the libguestfs back end in order to perform the inspection and mounting, the **run** command is not necessary when using the **-i** option. The **-i** option works for many common Linux guest virtual machines.

21.4.1.3. Accessing a guest virtual machine by name

A guest virtual machine can be accessed from the command line when you specify its name as known to libvirt (in other words, as it appears in **virsh list --all**). Use the **-d** option to access a guest virtual machine by its name, with or without the **-i** option:

```
$ guestfish --ro -d GuestName -i
```

21.4.2. Adding Files with guestfish

To add a file with guestfish you need to have the complete URI. The file can be a local file or a file located on a network block device (NBD) or a remote block device (RBD).

The format used for the URI should be like any of these examples. For local files, use ///:

- **guestfish -a *disk*.img**
- **guestfish -a file:///*directory*/*disk*.img**
- **guestfish -a nbd://*example.com*[:*port*]**
- **guestfish -a nbd://*example.com*[:*port*]/*exportname***
- **guestfish -a nbd://?socket=/*socket***
- **guestfish -a nbd:///*exportname*?socket=/*socket***
- **guestfish -a rbd:///*pool*/*disk***
- **guestfish -a rbd://*example.com*[:*port*]/*pool*/*disk***

21.4.3. Modifying Files with guestfish

To modify files, create directories or make other changes to a guest virtual machine, first heed the warning at the beginning of this section: your guest virtual machine must be shut down. Editing or changing a running disk with guestfish **will** result in disk corruption. This section gives an example of editing the **/boot/grub/grub.conf** file. When you are sure the guest virtual machine is shut down you can omit the **--ro** flag in order to get write access using a command such as:

```
$ guestfish -d RHEL3 -i

Welcome to guestfish, the guest filesystem shell for
editing virtual machine filesystems and disk images.

 Type: 'help' for help on commands
       'man' to read the manual
       'quit' to quit the shell

 Operating system: Red Hat Enterprise Linux AS release 3 (Taroon Update 9)
 /dev/vda2 mounted on /
```

```
 /dev/vda1 mounted on /boot

><fs> edit /boot/grub/grub.conf
```

Commands to edit files include **edit**, **vi** and **emacs**. Many commands also exist for creating files and directories, such as **write**, **mkdir**, **upload** and **tar-in**.

21.4.4. Other Actions with guestfish

You can also format file systems, create partitions, create and resize LVM logical volumes and much more, with commands such as **mkfs**, **part-add**, **lvresize**, **lvcreate**, **vgcreate** and **pvcreate**.

21.4.5. Shell Scripting with guestfish

Once you are familiar with using guestfish interactively, according to your needs, writing shell scripts with it may be useful. The following is a simple shell script to add a new MOTD (message of the day) to a guest:

```
#!/bin/bash -
 set -e
 guestname="$1"

 guestfish -d "$guestname" -i <<'EOF'
   write /etc/motd "Welcome to Acme Incorporated."
   chmod 0644 /etc/motd
 EOF
```

21.4.6. Augeas and libguestfs Scripting

Combining libguestfs with Augeas can help when writing scripts to manipulate Linux guest virtual machine configuration. For example, the following script uses Augeas to parse the keyboard configuration of a guest virtual machine, and to print out the layout. Note that this example only works with guest virtual machines running Red Hat Enterprise Linux:

```
#!/bin/bash -
 set -e
 guestname="$1"

 guestfish -d "$1" -i --ro <<'EOF'
   aug-init / 0
   aug-get /files/etc/sysconfig/keyboard/LAYOUT
 EOF
```

Augeas can also be used to modify configuration files. You can modify the above script to change the keyboard layout:

```
#!/bin/bash -
 set -e
 guestname="$1"

 guestfish -d "$1" -i <<'EOF'
   aug-init / 0
   aug-set /files/etc/sysconfig/keyboard/LAYOUT '"gb"'
   aug-save
```

```
EOF
```

Note the three changes between the two scripts:

1. The **--ro** option has been removed in the second example, giving the ability to write to the guest virtual machine.
2. The **aug-get** command has been changed to **aug-set** to modify the value instead of fetching it. The new value will be **"gb"** (including the quotes).
3. The **aug-save** command is used here so Augeas will write the changes out to disk.

NOTE

More information about Augeas can be found on the website http://augeas.net.

guestfish can do much more than we can cover in this introductory document. For example, creating disk images from scratch:

```
guestfish -N fs
```

Or copying out whole directories from a disk image:

```
><fs> copy-out /home /tmp/home
```

For more information see the man page guestfish(1).

21.5. OTHER COMMANDS

This section describes tools that are simpler equivalents to using guestfish to view and edit guest virtual machine disk images.

- **virt-cat** is similar to the guestfish **download** command. It downloads and displays a single file to the guest virtual machine. For example:

  ```
  # virt-cat RHEL3 /etc/ntp.conf | grep ^server
   server     127.127.1.0       # local clock
  ```

- **virt-edit** is similar to the guestfish **edit** command. It can be used to interactively edit a single file within a guest virtual machine. For example, you may need to edit the **grub.conf** file in a Linux-based guest virtual machine that will not boot:

  ```
  # virt-edit LinuxGuest /boot/grub/grub.conf
  ```

  virt-edit has another mode where it can be used to make simple non-interactive changes to a single file. For this, the **-e** option is used. For example, the following command changes the root password in a Linux guest virtual machine to having no password:

  ```
  # virt-edit LinuxGuest /etc/passwd -e 's/^root:.*?:/root::/'
  ```

- **virt-ls** is similar to the guestfish **ls**, **ll** and **find** commands. It is used to list a directory or directories (recursively). For example, the following command would recursively list files and directories under /home in a Linux guest virtual machine:

```
# virt-ls -R LinuxGuest /home/ | less
```

21.6. VIRT-RESCUE: THE RESCUE SHELL

This section provides information about the rescue shell.

21.6.1. Introduction

This section describes **virt-rescue**, which can be considered analogous to a rescue CD for virtual machines. It boots a guest virtual machine into a rescue shell so that maintenance can be performed to correct errors and the guest virtual machine can be repaired.

There is some overlap between virt-rescue and guestfish. It is important to distinguish their differing uses. virt-rescue is for making interactive, ad-hoc changes using ordinary Linux file system tools. It is particularly suited to rescuing a guest virtual machine that has failed . virt-rescue cannot be scripted.

In contrast, guestfish is particularly useful for making scripted, structured changes through a formal set of commands (the libguestfs API), although it can also be used interactively.

21.6.2. Running virt-rescue

Before you use **virt-rescue** on a guest virtual machine, make sure the guest virtual machine is not running, otherwise disk corruption will occur. When you are sure the guest virtual machine is not live, enter:

```
$ virt-rescue -d GuestName
```

(where GuestName is the guest name as known to libvirt), or:

```
$ virt-rescue -a /path/to/disk/image
```

(where the path can be any file, any logical volume, LUN, or so on) containing a guest virtual machine disk.

You will first see output scroll past, as virt-rescue boots the rescue VM. In the end you will see:

```
Welcome to virt-rescue, the libguestfs rescue shell.

 Note: The contents of / are the rescue appliance.
 You have to mount the guest virtual machine's partitions under /sysroot
 before you can examine them.

 bash: cannot set terminal process group (-1): Inappropriate ioctl for
device
 bash: no job control in this shell
 ><rescue>
```

The shell prompt here is an ordinary bash shell, and a reduced set of ordinary Red Hat Enterprise Linux commands is available. For example, you can enter:

```
><rescue> fdisk -l /dev/vda
```

The previous command will list disk partitions. To mount a file system, it is suggested that you mount it under **/sysroot**, which is an empty directory in the rescue machine for the user to mount anything you like. Note that the files under **/** are files from the rescue VM itself:

```
><rescue> mount /dev/vda1 /sysroot/
EXT4-fs (vda1): mounted filesystem with ordered data mode. Opts: (null)
><rescue> ls -l /sysroot/grub/
 total 324
 -rw-r--r--. 1 root root     63 Sep 16 18:14 device.map
 -rw-r--r--. 1 root root  13200 Sep 16 18:14 e2fs_stage1_5
 -rw-r--r--. 1 root root  12512 Sep 16 18:14 fat_stage1_5
 -rw-r--r--. 1 root root  11744 Sep 16 18:14 ffs_stage1_5
 -rw-------. 1 root root   1503 Oct 15 11:19 grub.conf
 [...]
```

When you are finished rescuing the guest virtual machine, exit the shell by entering **exit** or **Ctrl+d**.

virt-rescue has many command-line options. The options most often used are:

- **--ro**: Operate in read-only mode on the guest virtual machine. No changes will be saved. You can use this to experiment with the guest virtual machine. As soon as you exit from the shell, all of your changes are discarded.
- **--network**: Enable network access from the rescue shell. Use this for example if you need to download RPM or other files into the guest virtual machine.

21.7. VIRT-DF: MONITORING DISK USAGE

This section provides information about monitoring disk usage.

21.7.1. Introduction

This section describes **virt-df**, which displays file system usage from a disk image or a guest virtual machine. It is similar to the Linux **df** command, but for virtual machines.

21.7.2. Running virt-df

To display file system usage for all file systems found in a disk image, enter the following:

```
# virt-df -a /dev/vg_guests/RHEL7
 Filesystem                    1K-blocks       Used  Available  Use%
 RHEL6:/dev/sda1                  101086      10233      85634   11%
 RHEL6:/dev/VolGroup00/LogVol00 7127864    2272744    4493036   32%
```

(Where **/dev/vg_guests/RHEL7** is a Red Hat Enterprise Linux 7 guest virtual machine disk image. The path in this case is the host physical machine logical volume where this disk image is located.)

You can also use **virt-df** on its own to list information about all of your guest virtual machines known to libvirt. The **virt-df** command recognizes some of the same options as the standard **df** such as **-h** (human-readable) and **-i** (show inodes instead of blocks).

```
# virt-df -h -d domname
 Filesystem                          Size       Used  Available  Use%
 F14x64:/dev/sda1                  484.2M      66.3M     392.9M   14%
 F14x64:/dev/vg_f14x64/lv_root       7.4G       3.0G       4.4G   41%
 RHEL6brewx64:/dev/sda1            484.2M      52.6M     406.6M   11%
 RHEL6brewx64:/dev/vg_rhel6brewx64/lv_root
                                    13.3G       3.4G       9.2G   26%
```

NOTE

You can use **virt-df** safely on live guest virtual machines, since it only needs read-only access. However, you should not expect the numbers to be precisely the same as those from a **df** command running inside the guest virtual machine. This is because what is on disk will be slightly out of sync with the state of the live guest virtual machine. Nevertheless it should be a good enough approximation for analysis and monitoring purposes.

virt-df is designed to allow you to integrate the statistics into monitoring tools, databases and so on. This allows system administrators to generate reports on trends in disk usage, and alerts if a guest virtual machine is about to run out of disk space. To do this you should use the **--csv** option to generate machine-readable Comma-Separated-Values (CSV) output. CSV output is readable by most databases, spreadsheet software and a variety of other tools and programming languages. The raw CSV looks like the following:

```
# virt-df --csv -d RHEL6Guest
 Virtual Machine,Filesystem,1K-blocks,Used,Available,Use%
 RHEL6brewx64,/dev/sda1,102396,24712,77684,24.1%
 RHEL6brewx64,/dev/sda2,20866940,7786652,13080288,37.3%
```

21.8. VIRT-RESIZE: RESIZING GUEST VIRTUAL MACHINES OFFLINE

This section provides information about resizing offline guest virtual machines.

21.8.1. Introduction

This section describes **virt-resize**, a tool for expanding or shrinking guest virtual machines. It only works for guest virtual machines that are offline (shut down). It works by copying the guest virtual machine image and leaving the original disk image untouched. This is ideal because you can use the original image as a backup, however there is a trade-off as you need twice the amount of disk space.

21.8.2. Expanding a Disk Image

This section demonstrates a simple case of expanding a disk image:

1. Locate the disk image to be resized. You can use the command **virsh dumpxml GuestName** for a libvirt guest virtual machine.

2. Decide on how you wish to expand the guest virtual machine. Run **virt-df -h** and **virt-filesystems** on the guest virtual machine disk, as shown in the following output:

```
# virt-df -h -a /dev/vg_guests/RHEL6
Filesystem                      Size       Used  Available  Use%
```

```
RHEL6:/dev/sda1                98.7M      10.0M      83.6M   11%
RHEL6:/dev/VolGroup00/LogVol00  6.8G       2.2G       4.3G   32%

# virt-filesystems -a disk.img --all --long -h
/dev/sda1 ext3 101.9M
/dev/sda2 pv 7.9G
```

The following example demonstrates how to:

- Increase the size of the first (boot) partition, from approximately 100MB to 500MB.
- Increase the total disk size from 8GB to 16GB.
- Expand the second partition to fill the remaining space.
- Expand **/dev/VolGroup00/LogVol00** to fill the new space in the second partition.

1. Make sure the guest virtual machine is shut down.
2. Rename the original disk as the backup. How you do this depends on the host physical machine storage environment for the original disk. If it is stored as a file, use the **mv** command. For logical volumes (as demonstrated in this example), use **lvrename**:

   ```
   # lvrename /dev/vg_guests/RHEL6 /dev/vg_guests/RHEL6.backup
   ```

3. Create the new disk. The requirements in this example are to expand the total disk size up to 16GB. Since logical volumes are used here, the following command is used:

   ```
   # lvcreate -L 16G -n RHEL6 /dev/vg_guests
   Logical volume "RHEL6" created
   ```

4. The requirements from step 2 are expressed by this command:

   ```
   # virt-resize \
          /dev/vg_guests/RHEL6.backup /dev/vg_guests/RHEL6 \
          --resize /dev/sda1=500M \
          --expand /dev/sda2 \
          --LV-expand /dev/VolGroup00/LogVol00
   ```

   The first two arguments are the input disk and output disk. **--resize /dev/sda1=500M** resizes the first partition up to 500MB. **--expand /dev/sda2** expands the second partition to fill all remaining space. **--LV-expand /dev/VolGroup00/LogVol00** expands the guest virtual machine logical volume to fill the extra space in the second partition.

   virt-resize describes what it is doing in the output:

   ```
   Summary of changes:
      /dev/sda1: partition will be resized from 101.9M to 500.0M
      /dev/sda1: content will be expanded using the 'resize2fs' method
      /dev/sda2: partition will be resized from 7.9G to 15.5G
      /dev/sda2: content will be expanded using the 'pvresize' method
      /dev/VolGroup00/LogVol00: LV will be expanded to maximum size
      /dev/VolGroup00/LogVol00: content will be expanded using the
   'resize2fs' method
   ```

```
   Copying /dev/sda1 ...
   [############################################################]
   Copying /dev/sda2 ...
   [############################################################]
   Expanding /dev/sda1 using the 'resize2fs' method
   Expanding /dev/sda2 using the 'pvresize' method
   Expanding /dev/VolGroup00/LogVol00 using the 'resize2fs' method
```

5. Try to boot the virtual machine. If it works (and after testing it thoroughly) you can delete the backup disk. If it fails, shut down the virtual machine, delete the new disk, and rename the backup disk back to its original name.

6. Use **virt-df** or **virt-filesystems** to show the new size:

```
# virt-df -h -a /dev/vg_pin/RHEL6
   Filesystem                        Size       Used  Available
Use%
   RHEL6:/dev/sda1                 484.4M      10.8M     448.6M
3%
   RHEL6:/dev/VolGroup00/LogVol00 14.3G       2.2G      11.4G   16%
```

Resizing guest virtual machines is not an exact science. If **virt-resize** fails, there are a number of tips that you can review and attempt in the virt-resize(1) man page. For some older Red Hat Enterprise Linux guest virtual machines, you may need to pay particular attention to the tip regarding GRUB.

21.9. VIRT-INSPECTOR: INSPECTING GUEST VIRTUAL MACHINES

This section provides information about inspecting guest virtual machines.

21.9.1. Introduction

virt-inspector is a tool for inspecting a disk image to find out what operating system it contains.

21.9.2. Installation

To install virt-inspector and the documentation, enter the following command:

```
# yum install libguestfs-tools
```

The documentation, including example XML output and a Relax-NG schema for the output, will be installed in **/usr/share/doc/libguestfs-devel-*/** where * is replaced by the version number of libguestfs.

21.9.3. Running virt-inspector

You can run **virt-inspector** against any disk image or libvirt guest virtual machine as shown in the following example:

```
$ virt-inspector -a disk.img > report.xml
```

Or as shown here:

```
$ virt-inspector -d GuestName > report.xml
```

The result will be an XML report (**report.xml**). The main components of the XML file are a top-level **<operatingsytems>** element containing usually a single **<operatingsystem>** element, similar to the following:

```
 <operatingsystems>
   <operatingsystem>

     <!-- the type of operating system and Linux distribution -->
     <name>linux</name>
     <distro>rhel</distro>
     <!-- the name, version and architecture -->
     <product_name>Red Hat Enterprise Linux Server release 6.4
</product_name>
     <major_version>6</major_version>
     <minor_version>4</minor_version>
     <package_format>rpm</package_format>
     <package_management>yum</package_management>
     <root>/dev/VolGroup/lv_root</root>
     <!-- how the filesystems would be mounted when live -->
     <mountpoints>
       <mountpoint dev="/dev/VolGroup/lv_root">/</mountpoint>
       <mountpoint dev="/dev/sda1">/boot</mountpoint>
       <mountpoint dev="/dev/VolGroup/lv_swap">swap</mountpoint>
     </mountpoints>

    < !-- filesystems-->
      <filesystem dev="/dev/VolGroup/lv_root">
        <label></label>
        <uuid>b24d9161-5613-4ab8-8649-f27a8a8068d3</uuid>
        <type>ext4</type>
        <content>linux-root</content>
        <spec>/dev/mapper/VolGroup-lv_root</spec>
      </filesystem>
      <filesystem dev="/dev/VolGroup/lv_swap">
        <type>swap</type>
        <spec>/dev/mapper/VolGroup-lv_swap</spec>
      </filesystem>
     <!-- packages installed -->
     <applications>
       <application>
         <name>firefox</name>
         <version>3.5.5</version>
         <release>1.fc12</release>
       </application>
     </applications>

   </operatingsystem>
 </operatingsystems>
```

Processing these reports is best done using W3C standard XPath queries. Red Hat Enterprise Linux 7 comes with the **xpath** command-line program, which can be used for simple instances. However, for long-term and advanced usage, you should consider using an XPath library along with your favorite programming language.

As an example, you can list out all file system devices using the following XPath query:

```
$ virt-inspector GuestName | xpath //filesystem/@dev
 Found 3 nodes:
 -- NODE --
 dev="/dev/sda1"
 -- NODE --
 dev="/dev/vg_f12x64/lv_root"
 -- NODE --
 dev="/dev/vg_f12x64/lv_swap"
```

Or list the names of all applications installed by entering:

```
$ virt-inspector GuestName | xpath //application/name
 [...long list...]
```

21.10. USING THE API FROM PROGRAMMING LANGUAGES

The libguestfs API can be used directly from the following languages in Red Hat Enterprise Linux 7: C, C++, Perl, Python, Java, Ruby and OCaml.

- To install C and C++ bindings, enter the following command:

  ```
  # yum install libguestfs-devel
  ```

- To install Perl bindings:

  ```
  # yum install 'perl(Sys::Guestfs)'
  ```

- To install Python bindings:

  ```
  # yum install python-libguestfs
  ```

- To install Java bindings:

  ```
  # yum install libguestfs-java libguestfs-java-devel libguestfs-
  javadoc
  ```

- To install Ruby bindings:

  ```
  # yum install ruby-libguestfs
  ```

- To install OCaml bindings:

  ```
  # yum install ocaml-libguestfs ocaml-libguestfs-devel
  ```

The binding for each language is essentially the same, but with minor syntactic changes. A C statement:

```
guestfs_launch (g);
```

Would appear like the following in Perl:

```
$g->launch ()
```

Or like the following in OCaml:

```
g#launch ()
```

Only the API from C is detailed in this section.

In the C and C++ bindings, you must manually check for errors. In the other bindings, errors are converted into exceptions; the additional error checks shown in the examples below are not necessary for other languages, but conversely you may wish to add code to catch exceptions. See the following list for some points of interest regarding the architecture of the libguestfs API:

- The libguestfs API is synchronous. Each call blocks until it has completed. If you want to make calls asynchronously, you have to create a thread.
- The libguestfs API is not thread safe: each handle should be used only from a single thread, or if you want to share a handle between threads you should implement your own mutex to ensure that two threads cannot execute commands on one handle at the same time.
- You should not open multiple handles on the same disk image. It is permissible if all the handles are read-only, but still not recommended.
- You should not add a disk image for writing if anything else could be using that disk image (a live VM, for example). Doing this will cause disk corruption.
- Opening a read-only handle on a disk image that is currently in use (for example by a live VM) is possible. However, the results may be unpredictable or inconsistent, particularly if the disk image is being heavily written to at the time you are reading it.

21.10.1. Interaction with the API using a C program

Your C program should start by including the **<guestfs.h>** header file, and creating a handle:

```
#include <stdio.h>
#include <stdlib.h>
#include <guestfs.h>

int
main (int argc, char *argv[])
{
  guestfs_h *g;

  g = guestfs_create ();
  if (g == NULL) {
    perror ("failed to create libguestfs handle");
    exit (EXIT_FAILURE);
   }

   /* ... */

   guestfs_close (g);

   exit (EXIT_SUCCESS);
 }
```

Save this program to a file (**test.c**). Compile this program and run it with the following two commands:

```
gcc -Wall test.c -o test -lguestfs
./test
```

At this stage it should print no output. The rest of this section demonstrates an example showing how to extend this program to create a new disk image, partition it, format it with an ext4 file system, and create some files in the file system. The disk image will be called **disk.img** and be created in the current directory.

The outline of the program is:

- Create the handle.
- Add disk(s) to the handle.
- Launch the libguestfs back end.
- Create the partition, file system and files.
- Close the handle and exit.

Here is the modified program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <guestfs.h>

 int
 main (int argc, char *argv[])
 {
   guestfs_h *g;
   size_t i;

   g = guestfs_create ();
   if (g == NULL) {
     perror ("failed to create libguestfs handle");
     exit (EXIT_FAILURE);
  }

   /* Create a raw-format sparse disk image, 512 MB in size. */
   int fd = open ("disk.img", O_CREAT|O_WRONLY|O_TRUNC|O_NOCTTY, 0666);
   if (fd == -1) {
     perror ("disk.img");
     exit (EXIT_FAILURE);
   }
   if (ftruncate (fd, 512 * 1024 * 1024) == -1) {
     perror ("disk.img: truncate");
     exit (EXIT_FAILURE);
   }
   if (close (fd) == -1) {
     perror ("disk.img: close");
     exit (EXIT_FAILURE);
```

```
    }

    /* Set the trace flag so that we can see each libguestfs call. */
    guestfs_set_trace (g, 1);

    /* Set the autosync flag so that the disk will be synchronized
     * automatically when the libguestfs handle is closed.
     */
    guestfs_set_autosync (g, 1);

    /* Add the disk image to libguestfs. */
    if (guestfs_add_drive_opts (g, "disk.img",
          GUESTFS_ADD_DRIVE_OPTS_FORMAT, "raw", /* raw format */
          GUESTFS_ADD_DRIVE_OPTS_READONLY, 0,   /* for write */
          -1 /* this marks end of optional arguments */ )
         == -1)
      exit (EXIT_FAILURE);

    /* Run the libguestfs back-end. */
    if (guestfs_launch (g) == -1)
      exit (EXIT_FAILURE);

    /* Get the list of devices. Because we only added one drive
     * above, we expect that this list should contain a single
     * element.
     */
    char **devices = guestfs_list_devices (g);
    if (devices == NULL)
      exit (EXIT_FAILURE);
    if (devices[0] == NULL || devices[1] != NULL) {
      fprintf (stderr,
               "error: expected a single device from list-devices\n");
      exit (EXIT_FAILURE);
    }

    /* Partition the disk as one single MBR partition. */
    if (guestfs_part_disk (g, devices[0], "mbr") == -1)
      exit (EXIT_FAILURE);

    /* Get the list of partitions. We expect a single element, which
     * is the partition we have just created.
     */
    char **partitions = guestfs_list_partitions (g);
    if (partitions == NULL)
      exit (EXIT_FAILURE);
    if (partitions[0] == NULL || partitions[1] != NULL) {
      fprintf (stderr,
               "error: expected a single partition from list-
partitions\n");
      exit (EXIT_FAILURE);
    }

    /* Create an ext4 filesystem on the partition. */
    if (guestfs_mkfs (g, "ext4", partitions[0]) == -1)
      exit (EXIT_FAILURE);
```

```
  /* Now mount the filesystem so that we can add files. */
  if (guestfs_mount_options (g, "", partitions[0], "/") == -1)
    exit (EXIT_FAILURE);

  /* Create some files and directories. */
  if (guestfs_touch (g, "/empty") == -1)
    exit (EXIT_FAILURE);

  const char *message = "Hello, world\n";
  if (guestfs_write (g, "/hello", message, strlen (message)) == -1)
    exit (EXIT_FAILURE);

  if (guestfs_mkdir (g, "/foo") == -1)
    exit (EXIT_FAILURE);

  /* This uploads the local file /etc/resolv.conf into the disk image. */
  if (guestfs_upload (g, "/etc/resolv.conf", "/foo/resolv.conf") == -1)
    exit (EXIT_FAILURE);

  /* Because 'autosync' was set (above) we can just close the handle
   * and the disk contents will be synchronized. You can also do
   * this manually by calling guestfs_umount_all and guestfs_sync.
   */
  guestfs_close (g);

  /* Free up the lists. */
  for (i = 0; devices[i] != NULL; ++i)
    free (devices[i]);
  free (devices);
  for (i = 0; partitions[i] != NULL; ++i)
    free (partitions[i]);
  free (partitions);

  exit (EXIT_SUCCESS);
}
```

Compile and run this program with the following two commands:

```
gcc -Wall test.c -o test -lguestfs
./test
```

If the program runs to completion successfully, you should be left with a disk image called **disk.img**, which you can examine with guestfish:

```
guestfish --ro -a disk.img -m /dev/sda1
><fs> ll /
><fs> cat /foo/resolv.conf
```

By default (for C and C++ bindings only), libguestfs prints errors to stderr. You can change this behavior by setting an error handler. The guestfs(3) man page discusses this in detail.

21.11. VIRT-SYSPREP: RESETTING VIRTUAL MACHINE SETTINGS

The **virt-sysprep** command-line tool can be used to reset or unconfigure a guest virtual machine so that clones can be made from it. This process involves removing SSH host keys, removing persistent

network MAC configuration, and removing user accounts. Virt-sysprep can also customize a virtual machine, for instance by adding SSH keys, users or logos. Each step can be enabled or disabled as required.

To use **virt-sysprep**, the guest virtual machine must be offline, so you must shut it down before running the commands. Note that **virt-sysprep** modifies the guest or disk image in place without making a copy of it. If you want to preserve the existing contents of the guest virtual machine, you must snapshot, copy or clone the disk first. For more information on copying and cloning disks, see libguestfs.org.

It is recommended not to use **virt-sysprep** as root, unless you need root in order to access the disk image. In such a case, however, it is better to change the permissions on the disk image to be writable by the non-root user running **virt-sysprep**.

To install **virt-sysprep**, enter the following command:

```
$ sudo yum install /usr/bin/virt-sysprep
```

The following command options are available to use with **virt-sysprep**:

Table 21.1. virt-sysprep commands

Command	Description	Example
--help	Displays a brief help entry about a particular command or about the **virt-sysprep** command. For additional help, see the virt-sysprep man page.	$ **virt-sysprep --help**
-a [*file*] or --add [*file*]	Adds the specified *file*, which should be a disk image from a guest virtual machine. The format of the disk image is auto-detected. To override this and force a particular format, use the **--format** option.	$ **virt-sysprep --add /dev/vms/disk.img**
-a [*URI*] or --add [*URI*]	Adds a remote disk. The URI format is compatible with guestfish. For more information, see Section 21.4.2, “Adding Files with guestfish”.	$ **virt-sysprep -a rbd://example.com[:port]/pool/disk**
-c [*URI*] or --connect [*URI*]	Connects to the given URI, if using libvirt. If omitted, then it connects via the KVM hypervisor. If you specify guest block devices directly (**virt-sysprep -a**), then libvirt is not used at all.	$ **virt-sysprep -c qemu:///system**

Command	Description	Example
-d [*guest*] or --domain [*guest*]	Adds all the disks from the specified guest virtual machine. Domain UUIDs can be used instead of domain names.	$ **virt-sysprep --domain *90df2f3f-8857-5ba9-2714-7d95907b1c9e***
-n or --dry-run	Performs a read-only "dry run" sysprep operation on the guest virtual machine. This runs the sysprep operation, but throws away any changes to the disk at the end.	$ **virt-sysprep -n**
--enable [*operations*]	Enables the specified *operations*. To list the possible operations, use the --list command.	$ **virt-sysprep --enable ssh-hostkeys,udev-persistent-net**
--operation or --operations	Chooses which sysprep operations to perform. To disable an operation, use the - before the operation name.	$ **virt-sysprep --operations ssh-hotkeys,udev-persistent-net** would enable both operations, while $ **virt-sysprep --operations firewall-rules,-tmp-files** would enable the firewall-rules operation and disable the tmp-files operation. For a list of valid operations, see libguestfs.org.
--format [**raw**|**qcow2**|**auto**]	The default for the -a option is to auto-detect the format of the disk image. Using this forces the disk format for -a options that follow on the command line. Using --format auto switches back to auto-detection for subsequent -a options (see the -a command above).	$ **virt-sysprep --format raw -a *disk.img*** forces raw format (no auto-detection) for disk.img, but **virt-sysprep --format raw -a *disk.img* --format auto -a another.img** forces raw format (no auto-detection) for **disk.img** and reverts to auto-detection for **another.img**. If you have untrusted raw-format guest disk images, you should use this option to specify the disk format. This avoids a possible security problem with malicious guests.

Command	Description	Example
--list-operations	List the operations supported by the virt-sysprep program. These are listed one per line, with one or more single-space-separated fields. The first field in the output is the operation name, which can be supplied to the **--enable** flag. The second field is a * character if the operation is enabled by default, or is blank if not. Additional fields on the same line include a description of the operation.	$ **virt-sysprep --list-operations**
--mount-options	Sets the mount options for each mount point in the guest virtual machine. Use a semicolon-separated list of mountpoint:options pairs. You may need to place quotes around this list to protect it from the shell.	$ **virt-sysprep --mount-options "/:notime"** will mount the root directory with the **notime** operation.
-q or --quiet	Prevents the printing of log messages.	$ **virt-sysprep -q**
-v or --verbose	Enables verbose messages for debugging purposes.	$ **virt-sysprep -v**
-V or --version	Displays the virt-sysprep version number and exits.	$ **virt-sysprep -V**
--root-password	Sets the root password. Can either be used to specify the new password explicitly, or to use the string from the first line of a selected file, which is more secure.	$ **virt-sysprep --root-password password:***123456* **-a** *guest.img* or $ **virt-sysprep --root-password file:***SOURCE_FILE_PATH* **-a** *guest.img*

For more information, see the libguestfs documentation.

21.12. VIRT-CUSTOMIZE: CUSTOMIZING VIRTUAL MACHINE SETTINGS

The **virt-customize** command-line tool can be used to customize a virtual machine. For example, by installing packages and editing configuration files.

To use **virt-customize**, the guest virtual machine must be offline, so you must shut it down before running the commands. Note that **virt-customize** modifies the guest or disk image in place without

making a copy of it. If you want to preserve the existing contents of the guest virtual machine, you must snapshot, copy or clone the disk first. For more information on copying and cloning disks, see libguestfs.org.

WARNING

Using **virt-customize** on live virtual machines, or concurrently with other disk editing tools can cause disk corruption. The virtual machine *must* be shut down before using this command. In addition, disk images should not be edited concurrently.

It is recommended that you do not run **virt-customize** as root.

To install **virt-customize**, run one of the following commands:

```
$ sudo yum install /usr/bin/virt-customize
```

or

```
$ sudo yum install libguestfs-tools-c
```

The following command options are available to use with **virt-customize**:

Table 21.2. virt-customize options

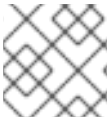
Command	Description	Example
--help	Displays a brief help entry about a particular command or about the **virt-customize** utility. For additional help, see the virt-customize man page.	$ **virt-customize --help**
-a [*file*] or --add [*file*]	Adds the specified *file*, which should be a disk image from a guest virtual machine. The format of the disk image is auto-detected. To override this and force a particular format, use the **--format** option.	$ **virt-customize --add /dev/vms/disk.img**
-a [*URI*] or --add [*URI*]	Adds a remote disk. The URI format is compatible with guestfish. For more information, see Section 21.4.2, “Adding Files with guestfish”.	$ **virt-customize -a rbd://example.com[:port]/pool/disk**

Command	Description	Example
-c [*URI*] or --connect [*URI*]	Connects to the given URI, if using libvirt. If omitted, then it connects via the KVM hypervisor. If you specify guest block devices directly (**virt-customize -a**), then libvirt is not used at all.	$ **virt-customize -c qemu:///system**
-d [*guest*] or --domain [*guest*]	Adds all the disks from the specified guest virtual machine. Domain UUIDs can be used instead of domain names.	$ **virt-customize --domain *90df2f3f-8857-5ba9-2714-7d95907b1c9e***
-n or --dry-run	Performs a read-only "dry run" customize operation on the guest virtual machine. This runs the customize operation, but throws away any changes to the disk at the end.	$ **virt-customize -n**
--format [**raw**|**qcow2**|**auto**]	The default for the -a option is to auto-detect the format of the disk image. Using this forces the disk format for -a options that follow on the command line. Using **--format auto** switches back to auto-detection for subsequent -a options (see the **-a** command above).	$ **virt-customize --format raw -a *disk.img*** forces raw format (no auto-detection) for disk.img, but **virt-customize --format raw -a *disk.img* --format auto -a another.img** forces raw format (no auto-detection) for **disk.img** and reverts to auto-detection for **another.img**. If you have untrusted raw-format guest disk images, you should use this option to specify the disk format. This avoids a possible security problem with malicious guests.
-m [*MB*] or --memsize [*MB*]	Changes the amount of memory allocated to **--run** scripts. If **--run** scripts or the **--install** option cause out of memory issues, increase the memory allocation.	$ **virt-customize --memsize 1024**

Command	Description	Example
--network or --no-network	Enables or disables network access from the guest during installation. The default is enabled. Use **--no-network** to disable access. This command does not affect guest access to the network after booting. For more information, see libguestfs documentation.	$ **virt-customize -a http://[user@]example.com[:port]/disk.img**
-q or --quiet	Prevents the printing of log messages.	$ **virt-customize -q**
-smp [*N*]	Enables *N* virtual CPUs that can be used by **--install** scripts. *N* must be 2 or more.	$ **virt-customize -smp 4**
-v or --verbose	Enables verbose messages for debugging purposes.	$ **virt-customize --verbose**
-V or --version	Displays the **virt-customize** version number and exits.	$ **virt-customize --V**
-x	Enables tracing of libguestfs API calls.	$ **virt-customize -x**

The **virt-customize** command uses customization options to configure how the guest is customized. The following provides information about the **--selinux-relabel** customization option.

The **--selinux-relabel** customization option relabels files in the guest so that they have the correct SELinux label. This option tries to relabel files immediately. If unsuccessful, **/.autorelabel** is activated on the image. This schedules the relabel operation for the next time the image boots.



NOTE

This option should only be used for guests that support SELinux.

The following example installs the GIMP and Inkscape packages on the guest and and ensures that the SELinux labels will be correct the next time the guest boots.

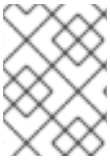
Example 21.1. Using `virt-customize` to install packages on a guest

```
virt-customize -a disk.img --install gimp,inkscape --selinux-relabel
```

For more information, including customization options, see libguestfs.org.

21.13. VIRT-DIFF: LISTING THE DIFFERENCES BETWEEN VIRTUAL MACHINE FILES

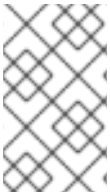
The **virt-diff** command-line tool can be used to lists the differences between files in two virtual machines disk images. The output shows the changes to a virtual machine's disk images after it has been running. The command can also be used to show the difference between overlays.



NOTE

You can use **virt-diff** safely on live guest virtual machines, because it only needs read-only access.

This tool finds the differences in file names, file sizes, checksums, extended attributes, file content and more between the running virtual machine and the selected image.



NOTE

The **virt-diff** command does not check the boot loader, unused space between partitions or within file systems, or "hidden" sectors. Therefore, it is recommended that you do not use this as a security or forensics tool.

To install **virt-diff**, run one of the following commands:

```
# yum install /usr/bin/virt-diff
```

or

```
# yum install libguestfs-tools-c
```

To specify two guests, you have to use the **-a** or **-d** option for the first guest, and the **-A** or **-D** option for the second guest. For example:

```
$ virt-diff -a old.img -A new.img
```

You can also use names known to libvirt. For example:

```
$ virt-diff -d oldguest -D newguest
```

The following command options are available to use with **virt-diff**:

Table 21.3. virt-diff options

Command	Description	Example
--help	Displays a brief help entry about a particular command or about the **virt-diff** utility. For additional help, see the virt-diff man page.	$ **virt-diff --help**

Command	Description	Example
-a [*file*] or --add [*file*]	Adds the specified *file*, which should be a disk image from the first virtual machine. If the virtual machine has multiple block devices, you must supply all of them with separate **-a** options. The format of the disk image is auto-detected. To override this and force a particular format, use the **--format** option.	$ **virt-customize --add /dev/vms/original.img -A /dev/vms/new.img**
-a [*URI*] or --add [*URI*]	Adds a remote disk. The URI format is compatible with guestfish. For more information, see Section 21.4.2, “Adding Files with guestfish”.	$ **virt-diff -a rbd://example.com[:port]/pool/newdisk -A rbd://example.com[:port]/pool/olddisk**
--all	Same as **--extra-stats --times --uids --xattrs**.	$ **virt-diff --all**
--atime	By default, **virt-diff** ignores changes in file access times, since those are unlikely to be interesting. Use the **--atime** option to show access time differences.	$ **virt-diff --atime**
-A [*file*]	Adds the specified *file* or *URI*, which should be a disk image from the second virtual machine.	$ **virt-diff --add /dev/vms/original.img -A /dev/vms/new.img**
-c [*URI*] or --connect [*URI*]	Connects to the given URI, if using libvirt. If omitted, then it connects to the default libvirt hypervisor. If you specify guest block devices directly (**virt-diff -a**), then libvirt is not used at all.	$ **virt-diff -c qemu:///system**
--csv	Provides the results in a comma-separated values (CSV) format. This format can be imported easily into databases and spreadsheets. For further information, see Note.	virt-diff --csv

Command	Description	Example
-d [*guest*] or --domain [*guest*]	Adds all the disks from the specified guest virtual machine as the first guest virtual machine. Domain UUIDs can be used instead of domain names.	$ **virt-diff --domain *90df2f3f-8857-5ba9-2714-7d95907b1c9e***
-D [*guest*]	Adds all the disks from the specified guest virtual machine as the second guest virtual machine. Domain UUIDs can be used instead of domain names.	$ **virt-diff --D *90df2f3f-8857-5ba9-2714-7d95907b1cd4***
--extra-stats	Displays extra statistics.	$ **virt-diff --extra-stats**
--format or --format=[**raw**|**qcow2**]	The default for the **-a**/**-A** option is to auto-detect the format of the disk image. Using this forces the disk format for **-a**/**-A** options that follow on the command line. Using **--format auto** switches back to auto-detection for subsequent -a options (see the **-a** command above).	$ **virt-diff --format raw -a *new.img* -A *old.img*** forces raw format (no auto-detection) for new.img and old.img, but **virt-diff --format raw -a *new.img* --format auto -a *old.img*** forces raw format (no auto-detection) for **new.img** and reverts to auto-detection for **old.img**. If you have untrusted raw-format guest disk images, you should use this option to specify the disk format. This avoids a possible security problem with malicious guests.
-h or --human-readable	Displays file sizes in human-readable format.	$ **virt-diff -h**
--time-days	Displays time fields for changed files as days before now (negative if in the future). Note that **0** in the output means between 86,399 seconds (23 hours, 59 minutes, and 59 seconds) before now and 86,399 seconds in the future.	$ **virt-diff --time-days**
-v or --verbose	Enables verbose messages for debugging purposes.	$ **virt-diff --verbose**
-V or --version	Displays the **virt-diff** version number and exits.	$ **virt-diff -V**

Command	Description	Example
-x	Enables tracing of libguestfs API calls.	$ **virt-diff -x**

NOTE

The comma-separated values (CSV) format can be difficult to parse. Therefore, it is recommended that for shell scripts, you should use csvtool and for other languages, use a CSV processing library (such as Text::CSV for Perl or Python's built-in csv library). In addition, most spreadsheets and databases can import CSV directly.

For more information, including additional options, see libguestfs.org.

21.14. VIRT-SPARSIFY: RECLAIMING EMPTY DISK SPACE

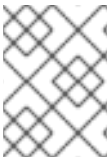
The **virt-sparsify** command-line tool can be used to make a virtual machine disk (or any disk image) sparse. This is also known as thin-provisioning. Free disk space on the disk image is converted to free space on the host.

The **virt-sparsify** command can work with most filesystems, such as ext2, ext3, ext4, btrfs, NTFS. It also works with LVM physical volumes. **virt-sparsify** can operate on any disk image, not just virtual machine disk images.

WARNING

Using **virt-sparsify** on live virtual machines, or concurrently with other disk editing tools can cause disk corruption. The virtual machine *must* be shut down before using this command. In addition, disk images should not be edited concurrently.

The command can also be used to convert between some disk formats. For example, **virt-sparsify** can convert a raw disk image to a thin-provisioned qcow2 image.



NOTE

If a virtual machine has multiple disks and uses volume management, **virt-sparsify** will work, but it will not be very effective.

If the input is raw, then the default output is raw sparse. The size of the output image must be checked using a tool that understands sparseness.

```
$ ls -lh test1.img
-rw-rw-r--. 1 rjones rjones 100M Aug  8 08:08 test1.img
$ du -sh test1.img
3.6M   test1.img
```

Note that the **ls** command shows the image size to be 100M. However, the **du** command correctly shows the image size to be 3.6M.

Important limitations

The following is a list of important limitations:

- The virtual machine *must be shutdown* before using **virt-sparsify**.
- In a worst case scenario, **virt-sparsify** may require up to twice the virtual size of the source disk image. One for the temporary copy and one for the destination image.

  If you use the **--in-place** option, large amounts of temporary space are not needed.
- **virt-sparsify** cannot be used to resize disk images. To resize disk images, use **virt-resize**. For information about **virt-resize**, see Section 21.8, “virt-resize: Resizing Guest Virtual Machines Offline”.
- **virt-sparsify** does not work with encrypted disks, because encrypted disks cannot be sparsified.
- **virt-sparsify** cannot sparsify the space between partitions. This space is often used for critical items like bootloaders, so it is not really unused space.
- In **copy** mode, qcow2 internal snapshots are not copied to the destination image.

Examples

To install **virt-sparsify**, run one of the following commands:

```
# yum install /usr/bin/virt-sparsify
```

or

```
# yum install libguestfs-tools-c
```

To sparsify a disk:

```
# virt-sparsify /dev/sda1 /dev/device
```

Copies the contents of **/dev/sda1** to **/dev/device**, making the output sparse. If **/dev/device** already exists, it is overwritten. The format of **/dev/sda1** is detected and used as the format for **/dev/device**.

To convert between formats:

```
# virt-sparsify disk.raw --convert qcow2 disk.qcow2
```

Tries to zero and sparsify free space on every filesystem it can find within the source disk image.

To prevent free space from being overwritten with zeros on certain filesystems:

```
# virt-sparsify --ignore /dev/device /dev/sda1 /dev/device
```

Creates sparsified disk images from all filesystems in the disk image, without overwriting free space on the filesystems with zeros.

To make a disk image sparse without creating a temporary copy:

```
# virt-sparsify --in-place disk.img
```

Makes the specified disk image sparse, overwriting the image file.

`virt-sparsify` options

The following command options are available to use with **`virt-sparsify`**:

Table 21.4. `virt-sparsify` options

Command	Description	Example
--help	Displays a brief help entry about a particular command or about the **`virt-sparsify`** utility. For additional help, see the virt-sparsify man page.	$ **`virt-sparsify --help`**
--check-tmpdir **`ignore`**|**`continue`**|**`warn`**|**`fail`**	Estimates if *tmpdir* has enough space to complete the operation. The specified option determines the behavior if there is not enough space to complete the operation. • **`ignore`**: The issue is ignored and the operation continues. • **`continue`**: Reports an error and the operation continues. • **`warn`**: Reports an error and waits for the user to press Enter. • **`fail`**: Reports an error and aborts the operation. This option cannot be used with the **`--in-place`** option.	$ **`virt-sparsify --check-tmpdir ignore /dev/sda1 /dev/device`** $ **`virt-sparsify --check-tmpdir continue /dev/sda1 /dev/device`** $ **`virt-sparsify --check-tmpdir warn /dev/sda1 /dev/device`** $ **`virt-sparsify --check-tmpdir fail /dev/sda1 /dev/device`**
--compress	Compresses the output file. This *only* works if the output format is qcow2. This option cannot be used with the **`--in-place`** option.	$ **`virt-sparsify --compress /dev/sda1 /dev/device`**

Command	Description	Example
--convert	Creates the sparse image using a specified format. If no format is specified, the input format is used. The following output formats are supported and known to work: raw, qcow, vdi You can use any format supported by the **QEMU** emulator. It is recommended that you use the **`--convert`** option. This way, **`virt-sparsify`** does not need to guess the input format. This option cannot be used with the **`--in-place`** option.	$ **`virt-sparsify --convert raw /dev/sda1 /dev/device`** $ **`virt-sparsify --convert qcow2 /dev/sda1 /dev/device`** $ **`virt-sparsify --convert *other_format* *indisk outdisk*`**
--format	Specifies the format of the input disk image. If not specified, the format is detected from the image. When working with untrusted raw-format guest disk images, ensure to specify the format.	$ **`virt-sparsify --format raw /dev/sda1 /dev/device`** $ **`virt-sparsify --format qcow2 /dev/sda1 /dev/device`**
--ignore	Ignores the specified file system or volume group. When a filesystem is specified and the **`--in-place`** option is not specified, free space on the filesystem is not zeroed. However, existing blocks of zeroes are sparsified. When the **`--in-place`** option is specified, the filesystem is completely ignored. When a volume group is specified, the volume group is ignored. The volume group name should be used without the **`/dev/`** prefix. For example, **`--ignore vg_foo`** The **`--ignore`** option can be included in the command multiple times.	$ **`virt-sparsify --ignore filesystem1 /dev/sda1 /dev/device`** $ **`virt-sparsify --ignore volume_group/dev/sda1 /dev/device`**

Command	Description	Example
--in-place	Makes an image sparse in-place, instead of making a temporary copy. Although in-place sparsification is more efficient than copying sparsification, it cannot recover quite as much disk space as copying sparsification. In-place sparsification works using discard (also known as trim or unmap) support. To use in-place sparsification, specify a disk image that will be sparsified in-place. When specifying in-place sparsification, the following options cannot be used: • **`--convert`** and **`--compress`**, because they require wholesale disk format changes. **`--check-tmpdir`**, because large amounts of temporary space are not required.	$ **`virt-sparsify --in-place disk.img`**
-x	Enables tracing of libguestfs API calls.	$ **`virt-sparsify -x filesystem1 /dev/sda1 /dev/device`**

For more information, including additional options, see libguestfs.org.

CHAPTER 22. GRAPHICAL USER INTERFACE TOOLS FOR GUEST VIRTUAL MACHINE MANAGEMENT

In addition to **virt-manager**, Red Hat Enterprise Linux 7 provides the following tools that enable you to access your guest virtual machine's console.

22.1. VIRT-VIEWER

virt-viewer is a minimalistic command-line utility for displaying the graphical console of a guest virtual machine. The console is accessed using the VNC or SPICE protocol. The guest can be referred to by its name, ID, or UUID. If the guest is not already running, the viewer can be set to wait until it starts before attempting to connect to the console. The viewer can connect to remote hosts to get the console information and then also connect to the remote console using the same network transport.

In comparison with *virt-manager*, *virt-viewer* offers a smaller set of features, but is less resource-demanding. In addition, unlike *virt-manager*, *virt-viewer* in most cases does not require read-write permissions to libvirt. Therefore, it can be used by non-privileged users who should be able to connect to and display guests, but not to configure them.

To install virt-viewer, run:

```
# sudo yum install virt-viewer
```

Syntax

The basic *virt-viewer* command-line syntax is as follows:

```
# virt-viewer [OPTIONS] {guest-name|id|uuid}
```

To see the full list of options available for use with virt-viewer, see the *virt-viewer* man page.

Connecting to a guest virtual machine

If used without any options, *virt-viewer* lists guests that it can connect to on the default hypervisor of the local system.

To connect to a specified guest virtual machine that uses the default hypervisor:

```
# virt-viewer guest-name
```

To connect to a guest virtual machine that uses the KVM-QEMU hypervisor:

```
# virt-viewer --connect qemu:///system guest-name
```

To connect to a remote console using TLS:

```
# virt-viewer --connect qemu://example.org/ guest-name
```

To connect to a console on a remote host by using SSH, look up the guest configuration and then make a direct non-tunneled connection to the console:

```
# virt-viewer --direct --connect qemu+ssh://root@example.org/ guest-name
```

Interface

By default, the *virt-viewer* interface provides only the basic tools for interacting with the guest:

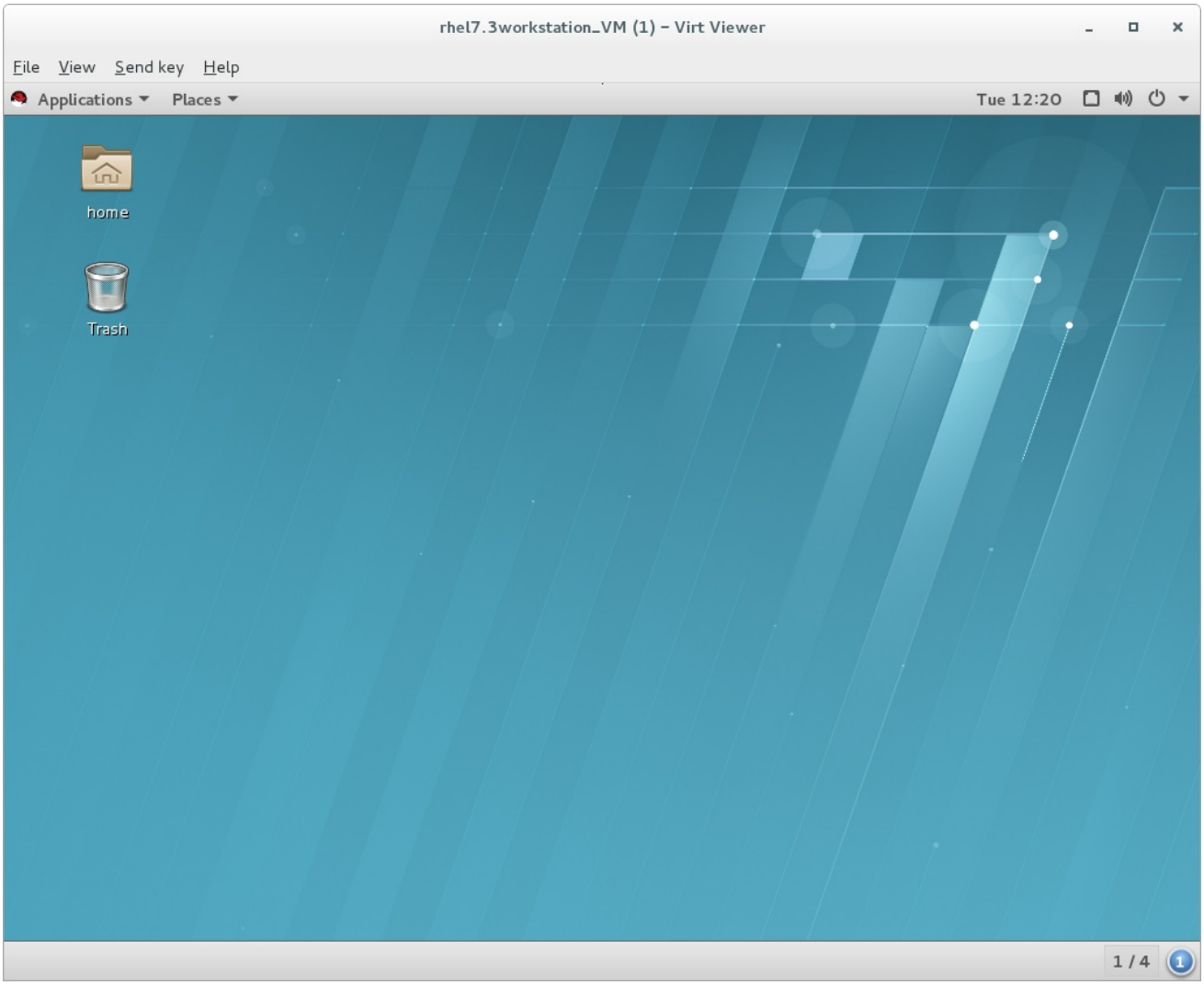


Figure 22.1. Sample *virt-viewer* interface

Setting hotkeys

To create a customized keyboard shortcut (also referred to as a hotkey) for the *virt-viewer* session, use the **--hotkeys** option:

```
# virt-viewer --hotkeys=action1=key-combination1[,action2=key-
combination2] guest-name
```

The following actions can be assigned to a hotkey:

- toggle-fullscreen
- release-cursor
- smartcard-insert
- smartcard-remove

Key-name combination hotkeys are not case-sensitive. Note that the hotkey setting does not carry over to future *virt-viewer* sessions.

Example 22.1. Setting a *virt-viewer* hotkey

To add a hotkey to change to full screen mode when connecting to a KVM-QEMU guest called testguest:

```
# virt-viewer --hotkeys=toggle-fullscreen=shift+f11 qemu:///system testguest
```

Kiosk mode

In kiosk mode, *virt-viewer* only allows the user to interact with the connected desktop, and does not provide any options to interact with the guest settings or the host system unless the guest is shut down. This can be useful for example when an administrator wants to restrict a user's range of actions to a specified guest.

To use kiosk mode, connect to a guest with the **-k** or **--kiosk** option.

Example 22.2. Using *virt-viewer* in kiosk mode

To connect to a KVM-QEMU virtual machine in kiosk mode that terminates after the machine is shut down, use the following command:

```
# virt-viewer --connect qemu:///system guest-name --kiosk --kiosk-quit on-disconnect
```

Note, however, that kiosk mode alone cannot ensure that the user does not interact with the host system or the guest settings after the guest is shut down. This would require further security measures, such as disabling the window manager on the host.

22.2. REMOTE-VIEWER

The **remote-viewer** is a simple remote desktop display client that supports SPICE and VNC. It shares most of the features and limitations with **virt-viewer**.

However, unlike *virt-viewer*, *remote-viewer* does not require libvirt to connect to the remote guest display. As such, *remote-viewer* can be used for example to connect to a virtual machine on a remote host that does not provide permissions to interact with libvirt or to use SSH connections.

To install the **remote-viewer** utility, run:

```
# sudo yum install virt-viewer
```

Syntax

The basic *remote-viewer* command-line syntax is as follows:

```
# remote-viewer [OPTIONS] {guest-name|id|uuid}
```

To see the full list of options available for use with remote-viewer, see the *remote-viewer* man page.

Connecting to a guest virtual machine

If used without any options, *remote-viewer* lists guests that it can connect to on the default URI of the local system.

To connect to a specific guest using remote-viewer, use the VNC/SPICE URI. For information about obtaining the URI, see Section 20.14, “Displaying a URI for Connection to a Graphical Display”.

Example 22.3. Connecting to a guest display using SPICE

Use the following to connect to a SPICE server on a machine called "testguest" that uses port 5900 for SPICE communication:

```
# remote-viewer spice://testguest:5900
```

Example 22.4. Connecting to a guest display using VNC

Use the following to connect to a VNC server on a machine called **testguest2** that uses port 5900 for VNC communication:

```
# remote-viewer vnc://testguest2:5900
```

Interface

By default, the *remote-viewer* interface provides only the basic tools for interacting with the guest:

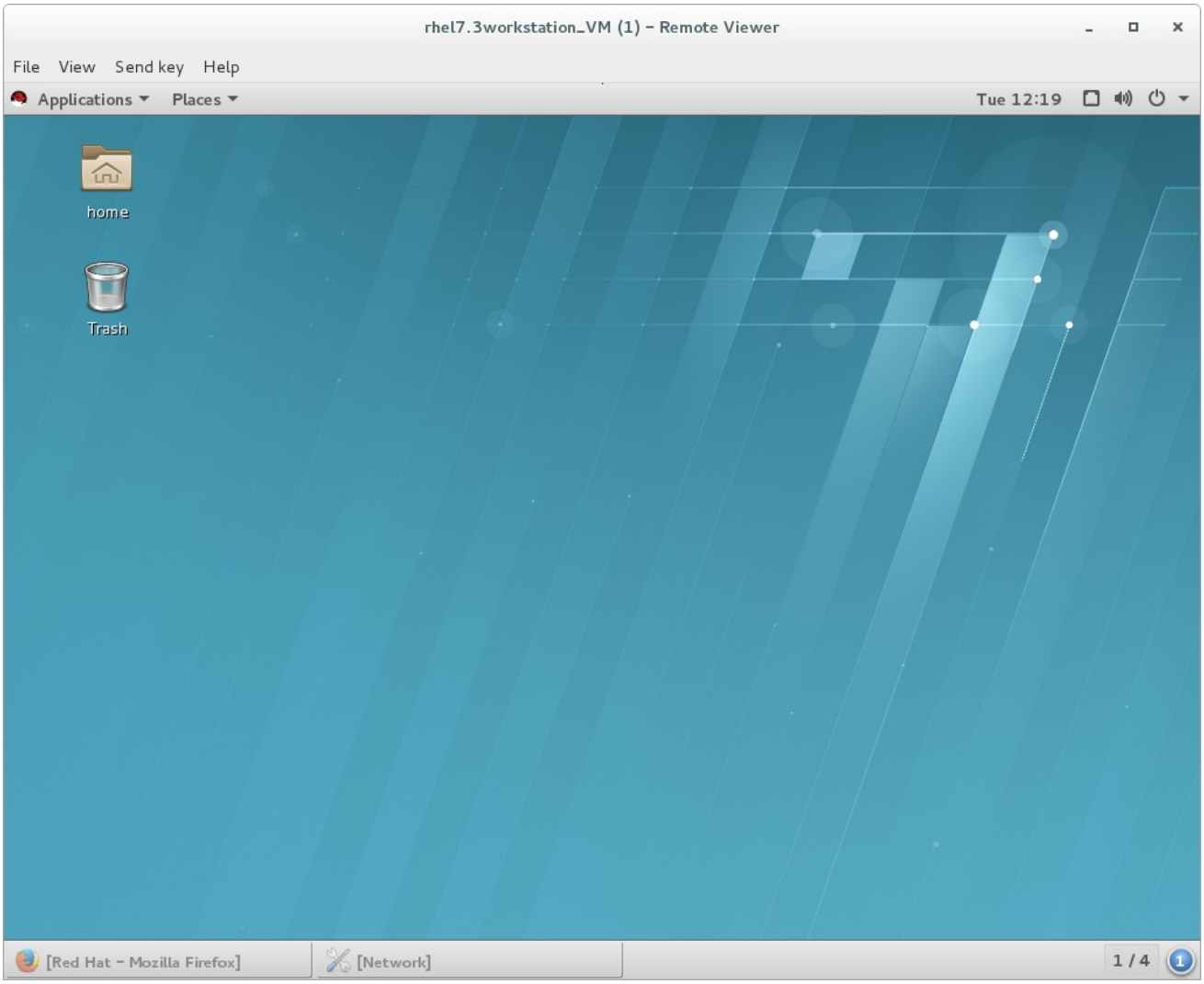


Figure 22.2. Sample *remote-viewer* interface

22.3. GNOME BOXES

Boxes is a lightweight graphical desktop virtualization tool used to view and access virtual machines and remote systems.

Unlike *virt-viewer* and *remote-viewer*, *Boxes* allows viewing guest virtual machines, but also creating and configuring them, similar to **virt-manager**. However, in comparison with *virt-manager*, *Boxes* offers fewer management options and features, but is easier to use.

To install *Boxes*, run:

```
# sudo yum install gnome-boxes
```

Open Boxes through **Applications** ⇒ **System Tools**.

The main screen shows the available guest virtual machines. The right side of the screen has two buttons:

- - the search button, to search for guest virtual machines by name, and
- - the selection button.

Clicking the selection button allows you to select one or more guest virtual machines in order to perform operations individually or as a group. The available operations are shown at the bottom of the screen on the operations bar:

Figure 22.3. The Operations Bar

There are four operations that can be performed:

- **Favorite**: Adds a heart to selected guest virtual machines and moves them to the top of the list of guests. This becomes increasingly helpful as the number of guests grows.
- **Pause**: The selected guest virtual machines will stop running.
- **Delete**: Removes selected guest virtual machines.
- **Properties**: Shows the properties of the selected guest virtual machine.

Create new guest virtual machines using the **New** button on the left side of the main screen.

Procedure 22.1. Creating a new guest virtual machine with Boxes

1. **Click New**
   This opens the **Introduction** screen. Click **Continue**.

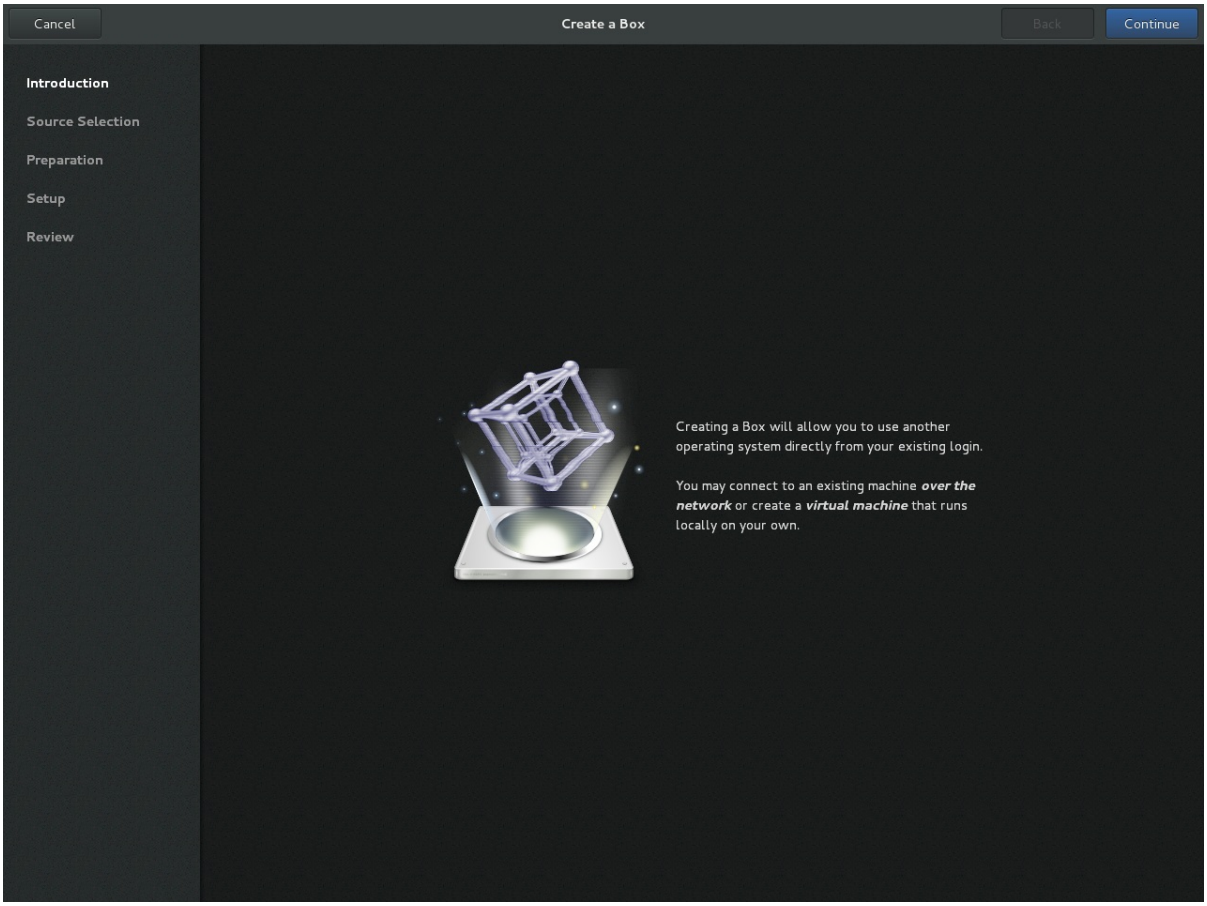


Figure 22.4. Introduction screen

2. **Select source**
   The **Source Selection** screen has three options:

   - Available media: Any immediately available installation media will be shown here. Clicking any of these will take you directly to the **Review** screen.

   - **`Enter a URL`**: Type in a URL to specify a local URI or path to an ISO file. This can also be used to access a remote machine. The address should follow the pattern of ***`protocol://IPaddress?port;`***, for example:

     ```
     spice://192.168.122.1?port=5906;
     ```

     The protocols can be **`spice://`**, **`qemu://`**, or **`vnc://`**

   - **`Select a file`**: Open a file directory to search for installation media manually.

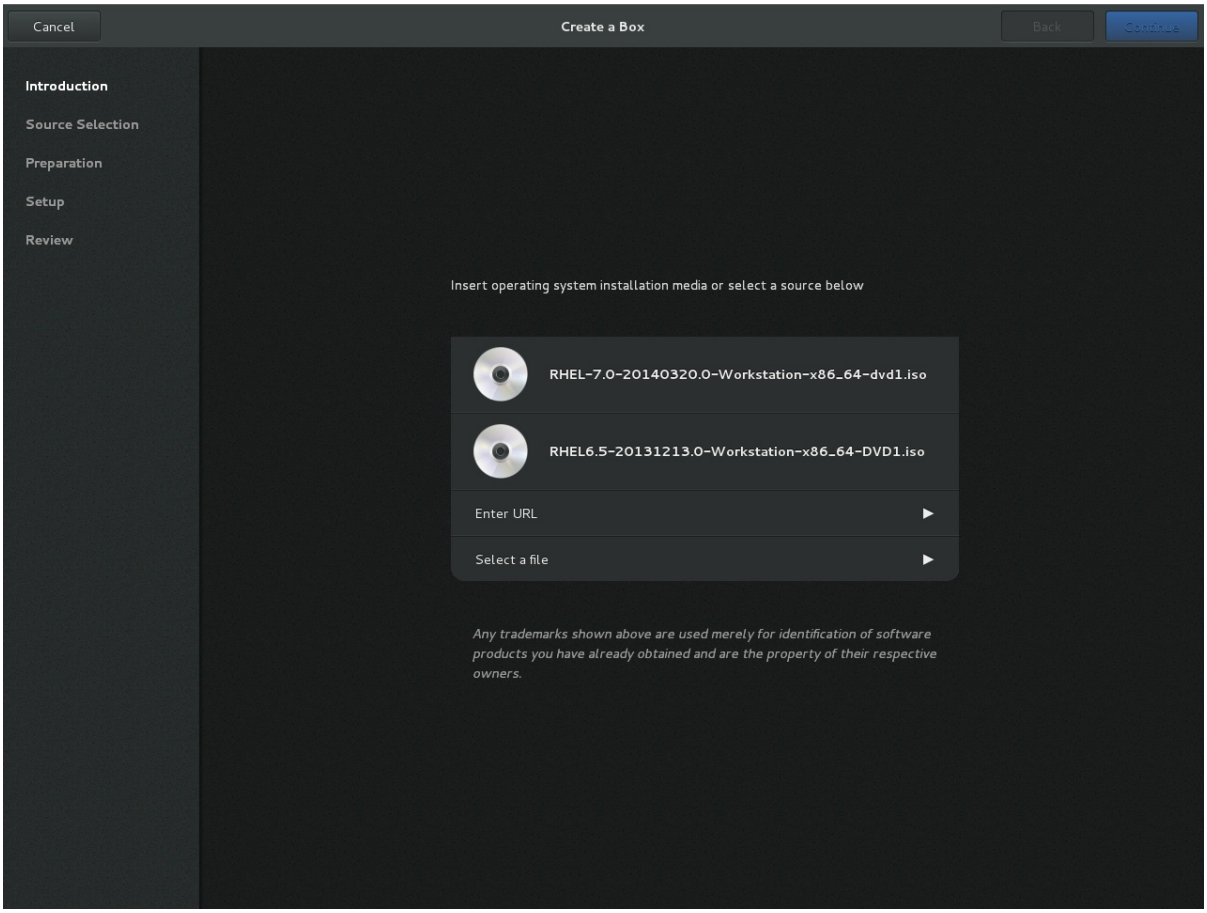


Figure 22.5. Source Selection screen

3. **Review the details**
   The **Review** screen shows the details of the guest virtual machine.

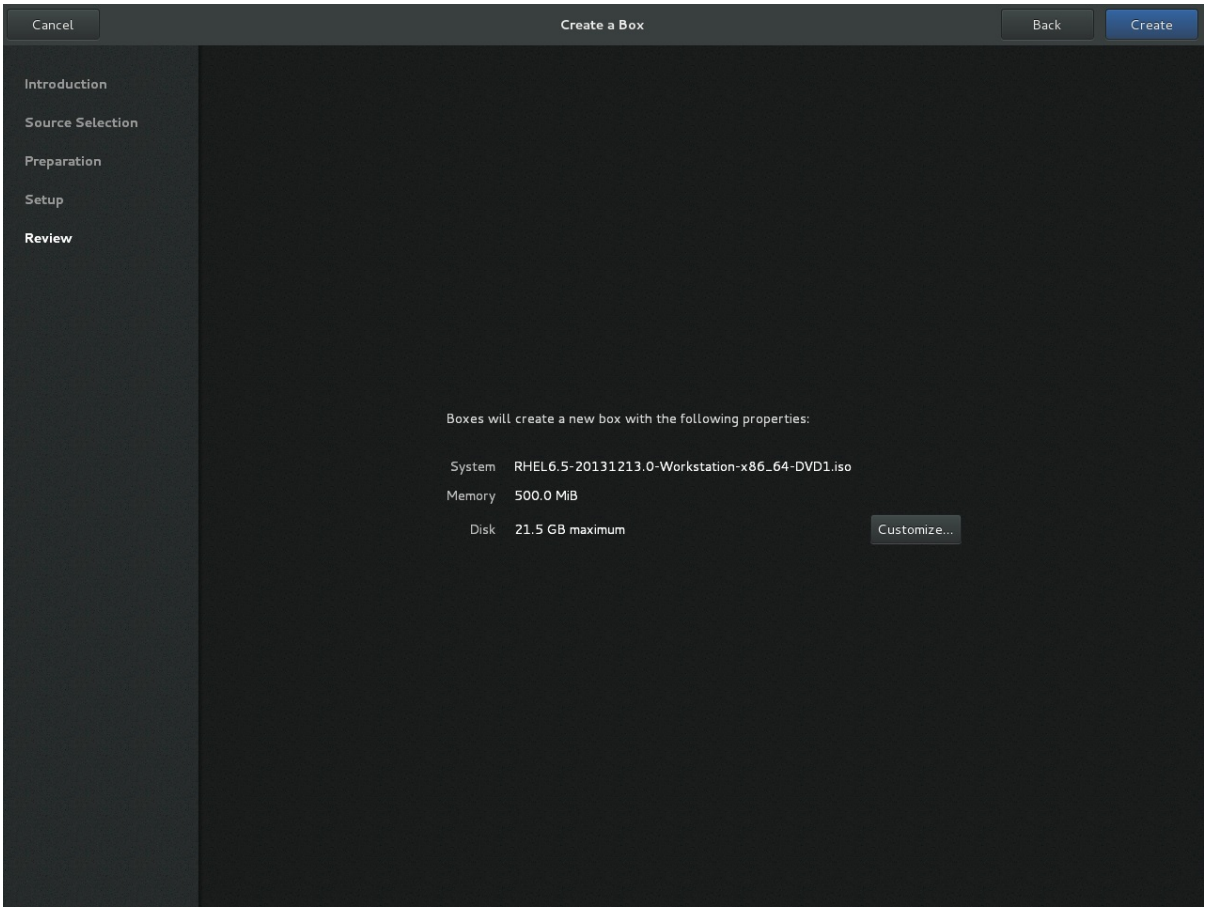


Figure 22.6. Review screen

These details can be left as is, in which case proceed to the final step, or:

4. **Optional: customize the details**
   clicking **`Customize`** allows you to adjust the configuration of the guest virtual machine, such as the memory and disk size.

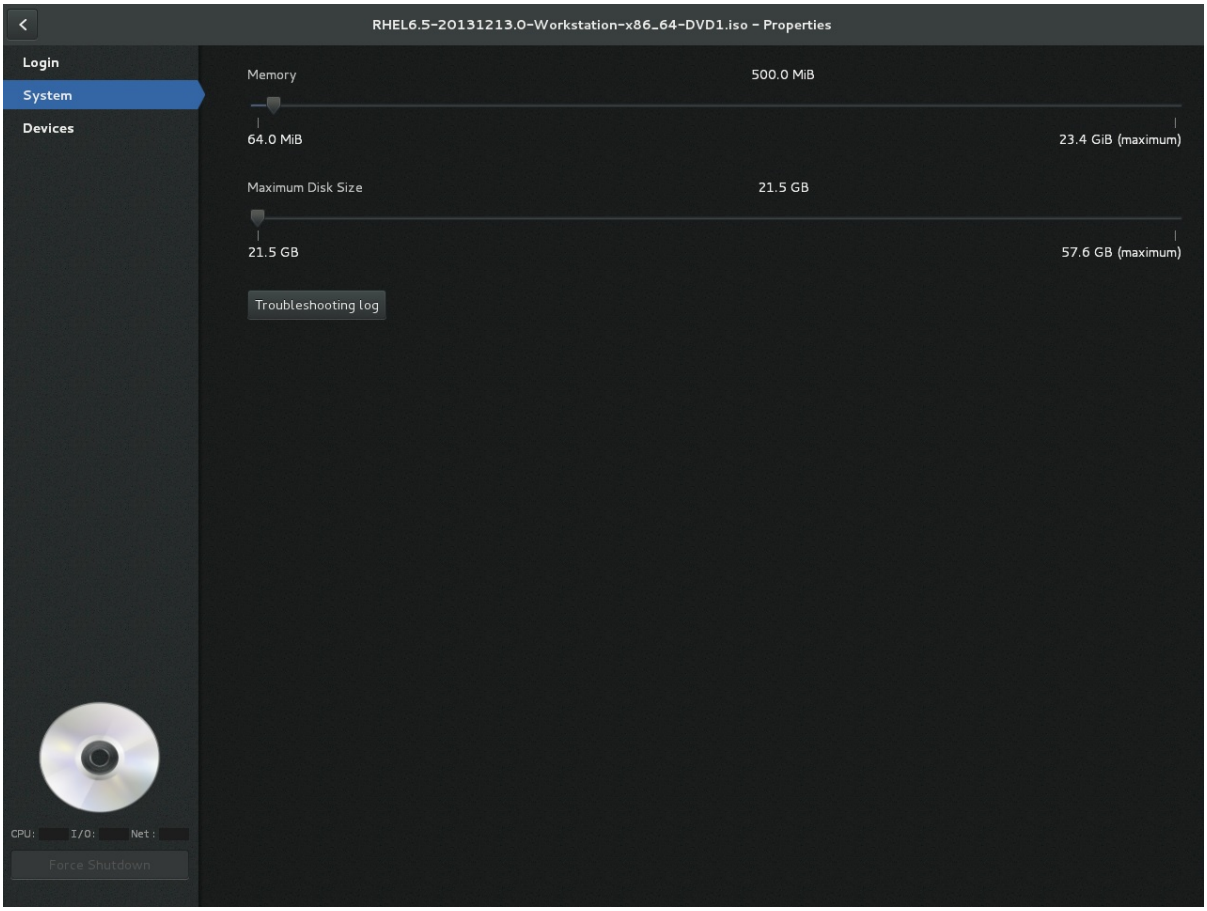
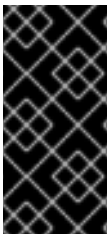


Figure 22.7. Customization screen

5. **Create**
   Click **Create**. The new guest virtual machine will open.

CHAPTER 23. MANIPULATING THE DOMAIN XML

This chapter explains in detail the components of guest virtual machine XML configuration files, also known as *domain XML*. In this chapter, the term *domain* refers to the root **<domain>** element required for all guest virtual machines. The domain XML has two attributes: **type** and **id**. **type** specifies the hypervisor used for running the domain. The allowed values are driver-specific, but include **KVM** and others. **id** is a unique integer identifier for the running guest virtual machine. Inactive machines have no **id** value. The sections in this chapter will describe the components of the domain XML. Additional chapters in this manual may see this chapter when manipulation of the domain XML is required.



IMPORTANT

Use only supported management interfaces (such as **virsh** and the **Virtual Machine Manager**) and commands (such as **virt-xml**) to edit the components of the domain XML file. Do not open and edit the domain XML file directly with a text editor. If you absolutely must edit the domain XML file directly, use the **virsh edit** command.

NOTE

This chapter is based on the libvirt upstream documentation.

23.1. GENERAL INFORMATION AND METADATA

This information is in this part of the domain XML:

```
<domain type='kvm' id='3'>
  <name>fv0</name>
  <uuid>4dea22b31d52d8f32516782e98ab3fa0</uuid>
  <title>A short description - title - of the domain</title>
  <description>A human readable description</description>
  <metadata>
    <app1:foo xmlns:app1="http://app1.org/app1/">..</app1:foo>
    <app2:bar xmlns:app2="http://app1.org/app2/">..</app2:bar>
  </metadata>
  ...
</domain>
```

Figure 23.1. Domain XML metadata

The components of this section of the domain XML are as follows:

Table 23.1. General metadata elements

Element	Description
<name>	Assigns a name for the virtual machine. This name should consist only of alpha-numeric characters and is required to be unique within the scope of a single host physical machine. It is often used to form the file name for storing the persistent configuration files.

Element	Description
<uuid>	Assigns a globally unique identifier for the virtual machine. The format must be RFC 4122-compliant, for example **3e3fce45-4f53-4fa7-bb32-11f34168b82b**. If omitted when defining or creating a new machine, a random UUID is generated. It is also possible to provide the UUID using a **sysinfo** specification.
<title>	Creates space for a short description of the domain. The title should not contain any new lines.
<description>	Different from the title, this data is not used by libvirt. It can contain any information the user chooses to display.
<metadata>	Can be used by applications to store custom metadata in the form of XML nodes/trees. Applications must use custom name spaces on XML nodes/trees, with only one top-level element per name space (if the application needs structure, they should have sub-elements to their name space element).

23.2. OPERATING SYSTEM BOOTING

There are a number of different ways to boot virtual machines, including BIOS boot loader, host physical machine boot loader, direct kernel boot, and container boot.

23.2.1. BIOS Boot Loader

Booting the BIOS is available for hypervisors supporting full virtualization. In this case, the BIOS has a boot order priority (floppy, hard disk, CD-ROM, network) determining where to locate the boot image. The **<os>** section of the domain XML contains the following information:

```
...
<os>
  <type>hvm</type>
  <boot dev='fd'/>
  <boot dev='hd'/>
  <boot dev='cdrom'/>
  <boot dev='network'/>
  <bootmenu enable='yes'/>
  <smbios mode='sysinfo'/>
  <bios useserial='yes' rebootTimeout='0'/>
</os>
...
```

Figure 23.2. BIOS boot loader domain XML

The components of this section of the domain XML are as follows:

Table 23.2. BIOS boot loader elements

Element	Description
<type>	Specifies the type of operating system to be booted on the guest virtual machine. **hvm** indicates that the operating system is designed to run on bare metal and requires full virtualization. **linux** refers to an operating system that supports the KVM hypervisor guest ABI. There are also two optional attributes: **arch** specifies the CPU architecture to virtualization, and **machine** refers to the machine type. For more information, see the *libvirt upstream documentation*.
<boot>	Specifies the next boot device to consider with one of the following values:**fd**, **hd**, **cdrom** or **network**. The boot element can be repeated multiple times to set up a priority list of boot devices to try in turn. Multiple devices of the same type are sorted according to their targets while preserving the order of buses. After defining the domain, its XML configuration returned by libvirt lists devices in the sorted order. Once sorted, the first device is marked as bootable. For more information, see the libvirt upstream documentation.
<bootmenu>	Determines whether or not to enable an interactive boot menu prompt on guest virtual machine start up. The **enable** attribute can be either **yes** or **no**. If not specified, the hypervisor default is used.
<smbios>	determines how SMBIOS information is made visible in the guest virtual machine. The **mode** attribute must be specified, as either **emulate** (allows the hypervisor generate all values), **host** (copies all of Block 0 and Block 1, except for the UUID, from the host physical machine's SMBIOS values; the **virConnectGetSysinfo** call can be used to see what values are copied), or **sysinfo** (uses the values in the sysinfo element). If not specified, the hypervisor's default setting is used.
<bios>	This element has attribute **useserial** with possible values **yes** or **no**. The attribute enables or disables the Serial Graphics Adapter, which enables users to see BIOS messages on a serial port. Therefore, one needs to have serial port defined. The **rebootTimeout** attribute controls whether and after how long the guest virtual machine should start booting again in case the boot fails (according to the BIOS). The value is set in milliseconds with a maximum of **65535**; setting **-1** disables the reboot.

23.2.2. Direct Kernel Boot

When installing a new guest virtual machine operating system, it is often useful to boot directly from a kernel and **initrd** stored in the host physical machine operating system, allowing command-line arguments to be passed directly to the installer. This capability is usually available for both fully virtualized and paravirtualized guest virtual machines.

```
...
<os>
  <type>hvm</type>
  <kernel>/root/f8-i386-vmlinuz</kernel>
  <initrd>/root/f8-i386-initrd</initrd>
  <cmdline>console=ttyS0 ks=http://example.com/f8-i386/os/</cmdline>
  <dtb>/root/ppc.dtb</dtb>
</os>
...
```

Figure 23.3. Direct kernel boot

The components of this section of the domain XML are as follows:

Table 23.3. Direct kernel boot elements

Element	Description
<type>	Same as described in the BIOS boot section.
<kernel>	Specifies the fully-qualified path to the kernel image in the host physical machine operating system.
<initrd>	Specifies the fully-qualified path to the (optional) ramdisk image in the host physical machine operating system.
<cmdline>	Specifies arguments to be passed to the kernel (or installer) at boot time. This is often used to specify an alternate primary console (such as a serial port), or the installation media source or kickstart file.

23.2.3. Container Boot

When booting a domain using container-based virtualization, instead of a kernel or boot image, a path to the **init** binary is required, using the **init** element. By default, this will be launched with no arguments. To specify the initial **argv**, use the **initarg** element, repeated as many times as required. The **cmdline** element provides an equivalent to **/proc/cmdline** but will not affect **<initarg>**.

```
...
<os>
  <type arch='x86_64'>exe</type>
  <init>/bin/systemd</init>
  <initarg>--unit</initarg>
  <initarg>emergency.service</initarg>
</os>
...
```

Figure 23.4. Container boot

23.3. SMBIOS SYSTEM INFORMATION

Some hypervisors allow control over what system information is presented to the guest virtual machine (for example, SMBIOS fields can be populated by a hypervisor and inspected using the **dmidecode** command in the guest virtual machine). The optional **sysinfo** element covers all such categories of information.

```
...
<os>
  <smbios mode='sysinfo'/>
  ...
</os>
<sysinfo type='smbios'>
  <bios>
    <entry name='vendor'>LENOVO</entry>
  </bios>
  <system>
    <entry name='manufacturer'>Fedora</entry>
    <entry name='vendor'>Virt-Manager</entry>
  </system>
</sysinfo>
...
```

Figure 23.5. SMBIOS system information

The **<sysinfo>** element has a mandatory attribute **type** that determines the layout of sub-elements, and may be defined as follows:

- **<smbios>** - Sub-elements call out specific SMBIOS values, which will affect the guest virtual machine if used in conjunction with the **smbios** sub-element of the **<os>** element. Each sub-element of **<sysinfo>** names a SMBIOS block, and within those elements can be a list of entry elements that describe a field within the block. The following blocks and entries are recognized:
  - **<bios>** - This is block 0 of SMBIOS, with entry names drawn from **vendor**, **version**, **date**, and **release**.
  - **<system>** - This is block 1 of SMBIOS, with entry names drawn from **manufacturer**, **product**, **version**, **serial**, **uuid**, **sku**, and **family**. If a **uuid** entry is provided alongside a top-level **uuid** element, the two values must match.

23.4. CPU ALLOCATION

```
<domain>
  ...
  <vcpu placement='static' cpuset="1-4,^3,6" current="1">2</vcpu>
  ...
</domain>
```

Figure 23.6. CPU Allocation

The **<vcpu>** element defines the maximum number of virtual CPUs allocated for the guest virtual machine operating system, which must be between 1 and the maximum number supported by the hypervisor. This element can contain an optional **cpuset** attribute, which is a comma-separated list of physical CPU numbers that the domain process and virtual CPUs can be pinned to by default.

Note that the pinning policy of the domain process and virtual CPUs can be specified separately by using the **cputune** attribute. If the **emulatorpin** attribute is specified in **<cputune>**, **cpuset** specified by **<vcpu>** will be ignored.

Similarly, virtual CPUs that have set a value for **vcpupin** cause **cpuset** settings to be ignored. For virtual CPUs where **vcpupin** is not specified, it will be pinned to the physical CPUs specified by **cpuset**. Each element in the **cpuset** list is either a single CPU number, a range of CPU numbers, or a caret (^) followed by a CPU number to be excluded from a previous range. The attribute **current** can be used to specify whether fewer than the maximum number of virtual CPUs should be enabled.

The **placement** optional attribute can be used to indicate the CPU placement mode for domain processes. The value of **placement** can be set as one of the following:

- **static** - pins the vCPU to the physical CPUs defined by the **cpuset** attribute. If **cpuset** is not defined, the domain processes will be pinned to all the available physical CPUs.
- **auto** - indicates the domain process will be pinned to the advisory nodeset from the querying numad, and the value of attribute **cpuset** will be ignored if it is specified.

NOTE

If the **cpuset** attribute is used along with **placement**, the value of **placement** defaults to the value of the **<numatune>** element (if it is used), or to **static**.

23.5. CPU TUNING

```
<domain>
  ...
  <cputune>
    <vcpupin vcpu="0" cpuset="1-4,^2"/>
    <vcpupin vcpu="1" cpuset="0,1"/>
    <vcpupin vcpu="2" cpuset="2,3"/>
    <vcpupin vcpu="3" cpuset="0,4"/>
    <emulatorpin cpuset="1-3"/>
    <shares>2048</shares>
    <period>1000000</period>
    <quota>-1</quota>
    <emulator_period>1000000</emulator_period>
    <emulator_quota>-1</emulator_quota>
  </cputune>
  ...
</domain>
```

Figure 23.7. CPU Tuning

Although all are optional, the components of this section of the domain XML are as follows:

Table 23.4. CPU tuning elements

Element	Description
<cputune>	Provides details regarding the CPU tunable parameters for the domain. This is optional.
<vcpupin>	Specifies which of host physical machine's physical CPUs the domain vCPU will be pinned to. If this is omitted, and the **cpuset** attribute of the **<vcpu>** element is not specified, the vCPU is pinned to all the physical CPUs by default. It contains two required attributes: the **<vcpu>** attribute specifies **id**, and the **cpuset** attribute is same as the **cpuset** attribute in the **<vcpu>** element.
<emulatorpin>	Specifies which of the host physical machine CPUs the "emulator" (a subset of a domains not including **<vcpu>**) will be pinned to. If this is omitted, and the **cpuset** attribute in the **<vcpu>** element is not specified, the "emulator" is pinned to all the physical CPUs by default. It contains one required **cpuset** attribute specifying which physical CPUs to pin to. **emulatorpin** is not allowed if the **placement** attribute in the **<vcpu>** element is set as **auto**.

Element	Description
<shares>	Specifies the proportional weighted share for the domain. If this is omitted, it defaults to the operating system provided defaults. If there is no unit for the value, it is calculated relative to the setting of the other guest virtual machine. For example, a guest virtual machine configured with a **<shares>** value of 2048 will get twice as much CPU time as a guest virtual machine configured with a **<shares>** value of 1024.
<period>	Specifies the enforcement interval in microseconds. By using **<period>**, each of the domain's vCPUs will not be allowed to consume more than its allotted quota worth of run time. This value should be within the following range: **1000-1000000**. A **<period>** with a value of **0** means no value.
<quota>	Specifies the maximum allowed bandwidth in microseconds. A domain with **<quota>** as any negative value indicates that the domain has infinite bandwidth, which means that it is not bandwidth controlled. The value should be within the following range: **1000 - 18446744073709551** or less than **0**. A **quota** with value of **0** means no value. You can use this feature to ensure that all vCPUs run at the same speed.
<emulator_period>	Specifies the enforcement interval in microseconds. Within an **<emulator_period>**, emulator threads (those excluding vCPUs) of the domain will not be allowed to consume more than the **<emulator_quota>** worth of run time. The **<emulator_period>** value should be in the following range: **1000 - 1000000**. An **<emulator_period>** with value of **0** means no value.
<emulator_quota>	Specifies the maximum allowed bandwidth in microseconds for the domain's emulator threads (those excluding vCPUs). A domain with an **<emulator_quota>** as a negative value indicates that the domain has infinite bandwidth for emulator threads (those excluding vCPUs), which means that it is not bandwidth controlled. The value should be in the following range: **1000 - 18446744073709551**, or less than **0**. An **<emulator_quota>** with value **0** means no value.

23.6. MEMORY BACKING

Memory backing allows the hypervisor to properly manage large pages within the guest virtual machine.

```
<domain>
  ...
  <memoryBacking>
    <hugepages>
   <page size="1" unit="G" nodeset="0-3,5"/>
   <page size="2" unit="M" nodeset="4"/>
  </hugepages>
  <nosharepages/>
  <locked/>
  </memoryBacking>
  ...
</domain>
```

Figure 23.8. Memory backing

For detailed information on memoryBacking elements, see the libvirt upstream documentation.

23.7. MEMORY TUNING

```
<domain>
  ...
  <memtune>
    <hard_limit unit='G'>1</hard_limit>
    <soft_limit unit='M'>128</soft_limit>
    <swap_hard_limit unit='G'>2</swap_hard_limit>
    <min_guarantee unit='bytes'>67108864</min_guarantee>
  </memtune>
  ...
</domain>
```

Figure 23.9. Memory tuning

Although **<memtune>** is optional, the components of this section of the domain XML are as follows:

Table 23.5. Memory tuning elements

Element	Description
<memtune>	Provides details regarding the memory tunable parameters for the domain. If this is omitted, it defaults to the operating system provided defaults. As parameters are applied to the process as a whole, when setting limits, determine values by adding the guest virtual machine RAM to the guest virtual machine video RAM, allowing for some memory overhead. For each tunable, it is possible to designate which unit the number is in on input, using the same values as for **<memory>**. For backwards compatibility, output is always in kibibytes (KiB).

Element	Description
<hard_limit>	The maximum memory the guest virtual machine can use. This value is expressed in **kibibytes** (blocks of 1024 bytes).
<soft_limit>	The memory limit to enforce during memory contention. This value is expressed in kibibytes (blocks of 1024 bytes).
<swap_hard_limit>	The maximum memory plus swap the guest virtual machine can use. This value is expressed in kibibytes (blocks of 1024 bytes). This must be more than **<hard_limit>** value.
<min_guarantee>	The guaranteed minimum memory allocation for the guest virtual machine. This value is expressed in kibibytes (blocks of 1024 bytes).

23.8. MEMORY ALLOCATION

In cases where the guest virtual machine crashes, the optional attribute ***dumpCore*** can be used to control whether the guest virtual machine's memory should be included in the generated core dump(***dumpCore='on'***) or not included (***dumpCore='off'***). Note that the default setting is ***on***, so unless the parameter is set to ***off***, the guest virtual machine memory will be included in the core dumpfile.

The **<maxMemory>** element determines maximum run-time memory allocation of the guest. The ***slots*** attribute specifies the number of slots available for adding memory to the guest.

The **<memory>** element specifies the maximum allocation of memory for the guest at boot time. This can also be set using the NUMA cell size configuration, and can be increased by hot-plugging of memory to the limit specified by ***maxMemory***.

The **<currentMemory>** element determines the actual memory allocation for a guest virtual machine. This value can be less than the maximum allocation (set by **<memory>**) to allow for the guest virtual machine memory to balloon as needed. If omitted, this defaults to the same value as the **<memory>** element. The unit attribute behaves the same as for memory.

```
<domain>
  <maxMemory slots='16' unit='KiB'>1524288</maxMemory>
  <memory unit='KiB' dumpCore='off'>524288</memory>
  <!-- changes the memory unit to KiB and does not allow the guest virtual
machine's memory to be included in the generated core dumpfile -->
  <currentMemory unit='KiB'>524288</currentMemory>
  <!-- makes the current memory unit 524288 KiB -->
  ...
</domain>
```

Figure 23.10. Memory unit

23.9. NUMA NODE TUNING

After NUMA node tuning is done using **virsh edit**, the following domain XML parameters are affected:

```
<domain>
  ...
  <numatune>
    <memory mode="strict" nodeset="1-4,^3"/>
  </numatune>
  ...
</domain>
```

Figure 23.11. NUMA node tuning

Although all are optional, the components of this section of the domain XML are as follows:

Table 23.6. NUMA node tuning elements

Element	Description
<numatune>	Provides details of how to tune the performance of a NUMA host physical machine by controlling NUMA policy for domain processes.
<memory>	Specifies how to allocate memory for the domain processes on a NUMA host physical machine. It contains several optional attributes. The **mode** attribute can be set to **interleave**, **strict**, or **preferred**. If no value is given it defaults to **strict**. The **nodeset** attribute specifies the NUMA nodes, using the same syntax as the **cpuset** attribute of the **<vcpu>** element. Attribute **placement** can be used to indicate the memory placement mode for the domain process. Its value can be either **static** or **auto**. If the **<nodeset>** attribute is specified it defaults to the **<placement>** of **<vcpu>**, or **static**. **auto** indicates the domain process will only allocate memory from the advisory nodeset returned from querying numad and the value of the **nodeset** attribute will be ignored if it is specified. If the **<placement>** attribute in **vcpu** is set to **auto**, and the **<numatune>** attribute is not specified, a default **<numatune>** with **<placement> auto** and **strict** mode will be added implicitly.

23.10. BLOCK I/O TUNING

```
<domain>
  ...
  <blkiotune>
    <weight>800</weight>
    <device>
      <path>/dev/sda</path>
      <weight>1000</weight>
    </device>
    <device>
      <path>/dev/sdb</path>
      <weight>500</weight>
    </device>
  </blkiotune>
  ...
</domain>
```

Figure 23.12. Block I/O tuning

Although all are optional, the components of this section of the domain XML are as follows:

Table 23.7. Block I/O tuning elements

Element	Description
<blkiotune>	This optional element provides the ability to tune **blkio** cgroup tunable parameters for the domain. If this is omitted, it defaults to the operating system provided defaults.
<weight>	This optional weight element is the overall I/O weight of the guest virtual machine. The value should be within the range 100 - 1000.
<device>	The domain may have multiple **<device>** elements that further tune the weights for each host physical machine block device in use by the domain. Note that multiple guest virtual machine disks can share a single host physical machine block device. In addition, as they are backed by files within the same host physical machine file system, this tuning parameter is at the global domain level, rather than being associated with each guest virtual machine disk device (contrast this to the **<iotune>** element which can be applied to a single **<disk>**). Each device element has two mandatory sub-elements, **<path>** describing the absolute path of the device, and **<weight>** giving the relative weight of that device, which has an acceptable range of 100 - 1000.

23.11. RESOURCE PARTITIONING

Hypervisors may allow for virtual machines to be placed into resource partitions, potentially with nesting

of said partitions. The **<resource>** element groups together configurations related to resource partitioning. It currently supports a child element partition whose content defines the path of the resource partition in which to place the domain. If no partition is listed, then the domain will be placed in a default partition. The partition must be created prior to starting the guest virtual machine. Only the (hypervisor-specific) default partition can be assumed to exist by default.

```
<resource>
  <partition>/virtualmachines/production</partition>
</resource>
```

Figure 23.13. Resource partitioning

Resource partitions are currently supported by the KVM and LXC drivers, which map partition paths to cgroups directories in all mounted controllers.

23.12. CPU MODELS AND TOPOLOGY

This section covers the requirements for CPU models. Note that every hypervisor has its own policy for which CPU features guest will see by default. The set of CPU features presented to the guest by KVM depends on the CPU model chosen in the guest virtual machine configuration. **qemu32** and **qemu64** are basic CPU models, but there are other models (with additional features) available. Each model and its topology is specified using the following elements from the domain XML:

```
<cpu match='exact'>
    <model fallback='allow'>core2duo</model>
    <vendor>Intel</vendor>
    <topology sockets='1' cores='2' threads='1'/>
    <feature policy='disable' name='lahf_lm'/>
</cpu>
```

Figure 23.14. CPU model and topology example 1

```
<cpu mode='host-model'>
   <model fallback='forbid'/>
   <topology sockets='1' cores='2' threads='1'/>
</cpu>
```

Figure 23.15. CPU model and topology example 2

```
<cpu mode='host-passthrough'/>
```

Figure 23.16. CPU model and topology example 3

In cases where no restrictions are to be put on the CPU model or its features, a simpler **<cpu>** element such as the following may be used:

```
<cpu>
   <topology sockets='1' cores='2' threads='1'/>
</cpu>
```

Figure 23.17. CPU model and topology example 4

```
<cpu mode='custom'>
   <model>POWER8</model>
</cpu>
```

Figure 23.18. PPC64/PSeries CPU model example

```
<cpu mode='host-passthrough'/>
```

Figure 23.19. aarch64/virt CPU model example

The components of this section of the domain XML are as follows:

Table 23.8. CPU model and topology elements

Element	Description
<cpu>	This is the main container for describing guest virtual machine CPU requirements.
<match>	Specifies how the virtual CPU is provided to the guest virtual machine must match these requirements. The **match** attribute can be omitted if topology is the only element within **<cpu>**. Possible values for the **match** attribute are: • **minimum** - the specified CPU model and features describes the minimum requested CPU. • **exact** - the virtual CPU provided to the guest virtual machine will exactly match the specification. • **strict** - the guest virtual machine will not be created unless the host physical machine CPU exactly matches the specification. Note that the **match** attribute can be omitted and will default to **exact**.

Element	Description
<mode>	This optional attribute may be used to make it easier to configure a guest virtual machine CPU to be as close to the host physical machine CPU as possible. Possible values for the **mode** attribute are: • **custom** - Describes how the CPU is presented to the guest virtual machine. This is the default setting when the **mode** attribute is not specified. This mode makes it so that a persistent guest virtual machine will see the same hardware no matter what host physical machine the guest virtual machine is booted on. • **host-model** - A shortcut to copying host physical machine CPU definition from the capabilities XML into the domain XML. As the CPU definition is copied just before starting a domain, the same XML can be used on different host physical machines while still providing the best guest virtual machine CPU each host physical machine supports. The **match** attribute and any feature elements cannot be used in this mode. For more information, see the libvirt upstream website. • **host-passthrough** With this mode, the CPU visible to the guest virtual machine is exactly the same as the host physical machine CPU, including elements that cause errors within libvirt. The obvious the downside of this mode is that the guest virtual machine environment cannot be reproduced on different hardware and therefore, this mode is recommended with great caution. The **model** and **feature** elements are not allowed in this mode.
<model>	Specifies the CPU model requested by the guest virtual machine. The list of available CPU models and their definition can be found in the **cpu_map.xml** file installed in libvirt's data directory. If a hypervisor is unable to use the exact CPU model, libvirt automatically falls back to a closest model supported by the hypervisor while maintaining the list of CPU features. An optional **fallback** attribute can be used to forbid this behavior, in which case an attempt to start a domain requesting an unsupported CPU model will fail. Supported values for fallback attribute are: **allow** (the default), and **forbid**. The optional **vendor_id** attribute can be used to set the vendor ID seen by the guest virtual machine. It must be exactly 12 characters long. If not set, the vendor iID of the host physical machine is used. Typical possible values are **AuthenticAMD** and **GenuineIntel**.

Element	Description
<vendor>	Specifies the CPU vendor requested by the guest virtual machine. If this element is missing, the guest virtual machine runs on a CPU matching given features regardless of its vendor. The list of supported vendors can be found in **cpu_map.xml**.
<topology>	Specifies the requested topology of the virtual CPU provided to the guest virtual machine. Three non-zero values must be given for sockets, cores, and threads: the total number of CPU sockets, number of cores per socket, and number of threads per core, respectively.
<feature>	Can contain zero or more elements used to fine-tune features provided by the selected CPU model. The list of known feature names can be found in the **cpu_map.xml** file. The meaning of each feature element depends on its policy attribute, which has to be set to one of the following values: • **force** - forces the virtual to be supported, regardless of whether it is actually supported by host physical machine CPU. • **require** - dictates that guest virtual machine creation will fail unless the feature is supported by host physical machine CPU. This is the default setting, • **optional** - this feature is supported by virtual CPU but only if it is supported by host physical machine CPU. • **disable** - this is not supported by virtual CPU. • **forbid** - guest virtual machine creation will fail if the feature is supported by host physical machine CPU.

23.12.1. Changing the Feature Set for a Specified CPU

Although CPU models have an inherent feature set, the individual feature components can either be allowed or forbidden on a feature by feature basis, allowing for a more individualized configuration for the CPU.

Procedure 23.1. Enabling and disabling CPU features

1. To begin, shut down the guest virtual machine.
2. Open the guest virtual machine's configuration file by running the **virsh edit [domain]** command.

3. Change the parameters within the **<feature>** or **<model>** to include the attribute value **'allow'** to force the feature to be allowed, or **'forbid'** to deny support for the feature.

```
<!-- original feature set -->
<cpu mode='host-model'>
   <model fallback='allow'/>
   <topology sockets='1' cores='2' threads='1'/>
</cpu>

<!--changed feature set-->
<cpu mode='host-model'>
   <model fallback='forbid'/>
   <topology sockets='1' cores='2' threads='1'/>
</cpu>
```

Figure 23.20. Example for enabling or disabling CPU features

```
<!--original feature set-->
<cpu match='exact'>
    <model fallback='allow'>core2duo</model>
    <vendor>Intel</vendor>
    <topology sockets='1' cores='2' threads='1'/>
    <feature policy='disable' name='lahf_lm'/>
</cpu>

<!--changed feature set-->
<cpu match='exact'>
    <model fallback='allow'>core2duo</model>
    <vendor>Intel</vendor>
    <topology sockets='1' cores='2' threads='1'/>
    <feature policy='enable' name='lahf_lm'/>
  </cpu>
```

Figure 23.21. Example 2 for enabling or disabling CPU features

4. When you have completed the changes, save the configuration file and start the guest virtual machine.

23.12.2. Guest Virtual Machine NUMA Topology

Guest virtual machine NUMA topology can be specified using the **<numa>** element in the domain XML:

```
<cpu>
  <numa>
    <cell cpus='0-3' memory='512000'/>
    <cell cpus='4-7' memory='512000'/>
  </numa>
</cpu>
...
```

Figure 23.22. Guest virtual machine NUMA topology

Each cell element specifies a NUMA cell or a NUMA node. **cpus** specifies the CPU or range of CPUs that are part of the node. **memory** specifies the node memory in kibibytes (blocks of 1024 bytes). Each cell or node is assigned a **cellid** or **nodeid** in increasing order starting from 0.

23.13. EVENTS CONFIGURATION

Using the following sections of domain XML it is possible to override the default actions for various events:

```
<on_poweroff>destroy</on_poweroff>
<on_reboot>restart</on_reboot>
<on_crash>restart</on_crash>
<on_lockfailure>poweroff</on_lockfailure>
```

Figure 23.23. Events Configuration

The following collections of elements allow the actions to be specified when a guest virtual machine operating system triggers a life cycle operation. A common use case is to force a reboot to be treated as a power off when doing the initial operating system installation. This allows the VM to be re-configured for the first post-install boot up.

The components of this section of the domain XML are as follows:

Table 23.9. Event configuration elements

State	Description

State	Description
<on_poweroff>	Specifies the action that is to be executed when the guest virtual machine requests a power off. Four arguments are possible: • **destroy** - This action terminates the domain completely and releases all resources. • **restart** - This action terminates the domain completely and restarts it with the same configuration. • **preserve** - This action terminates the domain completely but and its resources are preserved to allow for future analysis. • **rename-restart** - This action terminates the domain completely and then restarts it with a new name.
<on_reboot>	Specifies the action to be executed when the guest virtual machine requests a reboot. Four arguments are possible: • **destroy** - This action terminates the domain completely and releases all resources. • **restart** - This action terminates the domain completely and restarts it with the same configuration. • **preserve** - This action terminates the domain completely but and its resources are preserved to allow for future analysis. • **rename-restart** - This action terminates the domain completely and then restarts it with a new name.

State	Description
<on_crash>	Specifies the action that is to be executed when the guest virtual machine crashes. In addition, it supports these additional actions: • **coredump-destroy** - The crashed domain's core is dumped, the domain is terminated completely, and all resources are released. • **coredump-restart** - The crashed domain's core is dumped, and the domain is restarted with the same configuration settings. Four arguments are possible: • **destroy** - This action terminates the domain completely and releases all resources. • **restart** - This action terminates the domain completely and restarts it with the same configuration. • **preserve** - This action terminates the domain completely but and its resources are preserved to allow for future analysis. • **rename-restart** - This action terminates the domain completely and then restarts it with a new name.
<on_lockfailure>	Specifies the action to take when a lock manager loses resource locks. The following actions are recognized by libvirt, although not all of them need to be supported by individual lock managers. When no action is specified, each lock manager will take its default action. The following arguments are possible: • **poweroff** - Forcefully powers off the domain. • **restart** - Restarts the domain to reacquire its locks. • **pause** - Pauses the domain so that it can be manually resumed when lock issues are solved. • **ignore** - Keeps the domain running as if nothing happened.

23.14. POWER MANAGEMENT

It is possible to forcibly enable or disable BIOS advertisements to the guest virtual machine operating system using conventional management tools which affects the following section of the domain XML:

```
...
<pm>
  <suspend-to-disk enabled='no'/>
  <suspend-to-mem enabled='yes'/>
</pm>
...
```

Figure 23.24. Power Management

The **<pm>** element can be enabled using the argument **yes** or disabled using the argument **no**. BIOS support can be implemented for S3 using the **suspend-to-disk** argument and S4 using the **suspend-to-mem** argument for ACPI sleep states. If nothing is specified, the hypervisor will be left with its default value.

23.15. HYPERVISOR FEATURES

Hypervisors may allow certain CPU or machine features to be enabled (**state='on'**) or disabled (**state='off'**).

```
...
<features>
  <pae/>
  <acpi/>
  <apic/>
  <hap/>
  <privnet/>
  <hyperv>
    <relaxed state='on'/>
  </hyperv>
</features>
...
```

Figure 23.25. Hypervisor features

All features are listed within the **<features>** element, if a **<state>** is not specified it is disabled. The available features can be found by calling the **capabilities** XML, but a common set for fully virtualized domains are:

Table 23.10. Hypervisor features elements

State	Description

State	Description
<pae>	Physical address extension mode allows 32-bit guest virtual machines to address more than 4 GB of memory.
<acpi>	Useful for power management. For example, with KVM guest virtual machines it is required for graceful shutdown to work.
<apic>	Allows the use of programmable IRQ management. This element has an optional attribute **eoi** with values **on** and **off**, which sets the availability of EOI (End of Interrupt) for the guest virtual machine.
<hap>	Enables the use of hardware assisted paging if it is available in the hardware.

23.16. TIMEKEEPING

The guest virtual machine clock is typically initialized from the host physical machine clock. Most operating systems expect the hardware clock to be kept in UTC, which is the default setting.

Accurate timekeeping on guest virtual machines is a key challenge for virtualization platforms. Different hypervisors attempt to handle the problem of timekeeping in a variety of ways. **libvirt** provides hypervisor-independent configuration settings for time management, using the **<clock>** and **<timer>** elements in the domain XML. The domain XML can be edited using the **virsh edit** command. For details, see Section 20.22, “Editing a Guest Virtual Machine's XML Configuration Settings”.

```
...
<clock offset='localtime'>
  <timer name='rtc' tickpolicy='catchup' track='guest'>
    <catchup threshold='123' slew='120' limit='10000'/>
  </timer>
  <timer name='pit' tickpolicy='delay'/>
</clock>
...
```

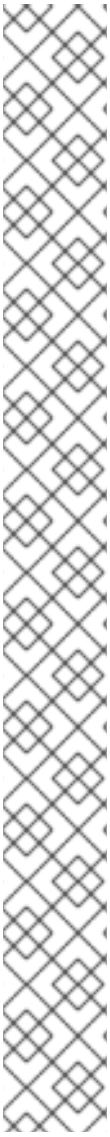
Figure 23.26. Timekeeping

The components of this section of the domain XML are as follows:

Table 23.11. Timekeeping elements

State	Description

State	Description
<clock>	The **<clock>** element is used to determine how the guest virtual machine clock is synchronized with the host physical machine clock. The **offset** attribute takes four possible values, allowing for fine grained control over how the guest virtual machine clock is synchronized to the host physical machine. Note that hypervisors are not required to support all policies across all time sources • **utc** - Synchronizes the clock to UTC when booted. **utc** mode can be converted to **variable** mode, which can be controlled by using the adjustment attribute. If the value is **reset**, the conversion is not done. A numeric value forces the conversion to **variable** mode using the value as the initial adjustment. The default adjustment is hypervisor-specific. • **localtime** - Synchronizes the guest virtual machine clock with the host physical machine's configured timezone when booted. The adjustment attribute behaves the same as in **utc** mode. • **timezone** - Synchronizes the guest virtual machine clock to the requested time zone. • **variable** - Gives the guest virtual machine clock an arbitrary offset applied relative to UTC or **localtime**, depending on the basis attribute. The delta relative to UTC (or **localtime**) is specified in seconds, using the **adjustment** attribute. The guest virtual machine is free to adjust the RTC over time and expect that it will be honored at next reboot. This is in contrast to **utc** and **localtime** mode (with the optional attribute **adjustment='reset'**), where the RTC adjustments are lost at each reboot. In addition, the **basis** attribute can be either **utc** (default) or **localtime**. The **clock** element may have zero or more **<timer>** elements.
<timer>	See Note
<present>	Specifies whether a particular timer is available to the guest virtual machine. Can be set to **yes** or **no**.



NOTE

A **<clock>** element can have zero or more **<timer>** elements as children. The **<timer>** element specifies a time source used for guest virtual machine clock synchronization.

In each **<timer>** element only the **name** is required, and all other attributes are optional:

- **name** - Selects which **timer** is being modified. The following values are acceptable: **kvmclock**, **pit**, or **rtc**.
- **track** - Specifies the timer track. The following values are acceptable: **boot**, **guest**, or **wall**. **track** is only valid for **name="rtc"**.
- **tickpolicy** - Determines what happens when the deadline for injecting a tick to the guest virtual machine is missed. The following values can be assigned:
  - **delay** - Continues to deliver ticks at the normal rate. The guest virtual machine time will be delayed due to the late tick.
  - **catchup** - Delivers ticks at a higher rate in order to catch up with the missed tick. The guest virtual machine time is not displayed once catch up is complete. In addition, there can be three optional attributes, each a positive integer: threshold, slew, and limit.
  - **merge** - Merges the missed tick(s) into one tick and injects them. The guest virtual machine time may be delayed, depending on how the merge is done.
  - **discard** - Throws away the missed tick(s) and continues with future injection at its default interval setting. The guest virtual machine time may be delayed, unless there is an explicit statement for handling lost ticks.

NOTE

The value **utc** is set as the clock offset in a virtual machine by default. However, if the guest virtual machine clock is run with the **localtime** value, the clock offset needs to be changed to a different value in order to have the guest virtual machine clock synchronized with the host physical machine clock.

Example 23.1. Always synchronize to UTC

```
<clock offset="utc" />
```

Example 23.2. Always synchronize to the host physical machine timezone

```
<clock offset="localtime" />
```

Example 23.3. Synchronize to an arbitrary time zone

```
<clock offset="timezone" timezone="Europe/Paris" />
```

Example 23.4. Synchronize to UTC + arbitrary offset

```
<clock offset="variable" adjustment="123456" />
```

23.17. TIMER ELEMENT ATTRIBUTES

The **name** element contains the name of the time source to be used. It can have any of the following values:

Table 23.12. Name attribute values

Value	Description
pit	Programmable Interval Timer - a timer with periodic interrupts. When using this attribute, the **tickpolicy** delay becomes the default setting.
rtc	Real Time Clock - a continuously running timer with periodic interrupts. This attribute supports the **tickpolicy catchup** sub-element.
kvmclock	KVM clock - the recommended clock source for KVM guest virtual machines. KVM pvclock, or kvm-clock allows guest virtual machines to read the host physical machine's wall clock time.

The **track** attribute specifies what is tracked by the timer, and is only valid for a **name** value of ***rtc***.

Table 23.13. track attribute values

Value	Description
boot	Corresponds to old *host physical machine* option, this is an unsupported tracking option.
guest	RTC always tracks the guest virtual machine time.
wall	RTC always tracks the host time.

The **tickpolicy** attribute and the values dictate the policy that is used to pass ticks on to the guest virtual machine.

Table 23.14. tickpolicy attribute values

Value	Description
delay	Continue to deliver at normal rate (ticks are delayed).
catchup	Deliver at a higher rate to catch up.
merge	Ticks merged into one single tick.
discard	All missed ticks are discarded.

The **present** attribute is used to override the default set of timers visible to the guest virtual machine. The **present** attribute can take the following values:

Table 23.15. present attribute values

Value	Description
yes	Force this timer to be visible to the guest virtual machine.
no	Force this timer to not be visible to the guest virtual machine.

23.18. DEVICES

This set of XML elements are all used to describe devices provided to the guest virtual machine domain. All of the devices below are indicated as children of the main **<devices>** element.

The following virtual devices are supported:

- virtio-scsi-pci - PCI bus storage device
- virtio-blk-pci - PCI bus storage device
- virtio-net-pci - PCI bus network device also known as virtio-net
- virtio-serial-pci - PCI bus input device
- virtio-balloon-pci - PCI bus memory balloon device
- virtio-rng-pci - PCI bus virtual random number generator device

IMPORTANT

If a virtio device is created where the number of vectors is set to a value higher than 32, the device behaves as if it was set to a zero value on Red Hat Enterprise Linux 6, but not on Enterprise Linux 7. The resulting vector setting mismatch causes a migration error if the number of vectors on any virtio device on either platform is set to 33 or higher. It is, therefore, not recommended to set the vector value to be greater than 32. All virtio devices with the exception of **virtio-balloon-pci** and **virtio-rng-pci** will accept a **vector** argument.

```
...
<devices>
  <emulator>/usr/libexec/qemu-kvm</emulator>
</devices>
...
```

Figure 23.27. Devices - child elements

The contents of the **<emulator>** element specify the fully qualified path to the device model emulator binary. The capabilities XML specifies the recommended default emulator to use for each particular domain type or architecture combination.

23.18.1. Hard Drives, Floppy Disks, and CD-ROMs

This section of the domain XML specifies any device that looks like a disk, including any floppy disk, hard disk, CD-ROM, or paravirtualized driver that is specified in the **<disk>** element.

```
    <disk type='network'>
      <driver name="qemu" type="raw" io="threads" ioeventfd="on"
event_idx="off"/>
      <source protocol="sheepdog" name="image_name">
        <host name="hostname" port="7000"/>
      </source>
      <target dev="hdb" bus="ide"/>
      <boot order='1'/>
      <transient/>
      <address type='drive' controller='0' bus='1' unit='0'/>
    </disk>
```

Figure 23.28. Devices - Hard drives, floppy disks, CD-ROMs Example

```
    <disk type='network'>
      <driver name="qemu" type="raw"/>
      <source protocol="rbd" name="image_name2">
        <host name="hostname" port="7000"/>
      </source>
      <target dev="hdd" bus="ide"/>
      <auth username='myuser'>
        <secret type='ceph' usage='mypassid'/>
      </auth>
    </disk>
```

Figure 23.29. Devices - Hard drives, floppy disks, CD-ROMs Example 2

```
<disk type='block' device='cdrom'>
  <driver name='qemu' type='raw'/>
  <target dev='hdc' bus='ide' tray='open'/>
  <readonly/>
</disk>
<disk type='network' device='cdrom'>
  <driver name='qemu' type='raw'/>
  <source protocol="http" name="url_path">
    <host name="hostname" port="80"/>
  </source>
  <target dev='hdc' bus='ide' tray='open'/>
  <readonly/>
</disk>
```

Figure 23.30. Devices - Hard drives, floppy disks, CD-ROMs Example 3

```
<disk type='network' device='cdrom'>
  <driver name='qemu' type='raw'/>
  <source protocol="https" name="url_path">
    <host name="hostname" port="443"/>
  </source>
  <target dev='hdc' bus='ide' tray='open'/>
  <readonly/>
</disk>
<disk type='network' device='cdrom'>
  <driver name='qemu' type='raw'/>
  <source protocol="ftp" name="url_path">
    <host name="hostname" port="21"/>
  </source>
  <target dev='hdc' bus='ide' tray='open'/>
  <readonly/>
</disk>
```

Figure 23.31. Devices - Hard drives, floppy disks, CD-ROMs Example 4

```
    <disk type='network' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <source protocol="ftps" name="url_path">
        <host name="hostname" port="990"/>
      </source>
      <target dev='hdc' bus='ide' tray='open'/>
      <readonly/>
    </disk>
    <disk type='network' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <source protocol="tftp" name="url_path">
        <host name="hostname" port="69"/>
      </source>
      <target dev='hdc' bus='ide' tray='open'/>
      <readonly/>
    </disk>
    <disk type='block' device='lun'>
      <driver name='qemu' type='raw'/>
      <source dev='/dev/sda'/>
      <target dev='sda' bus='scsi'/>
      <address type='drive' controller='0' bus='0' target='3' unit='0'/>
    </disk>
```

Figure 23.32. Devices - Hard drives, floppy disks, CD-ROMs Example 5

```
    <disk type='block' device='disk'>
      <driver name='qemu' type='raw'/>
      <source dev='/dev/sda'/>
      <geometry cyls='16383' heads='16' secs='63' trans='lba'/>
      <blockio logical_block_size='512' physical_block_size='4096'/>
      <target dev='hda' bus='ide'/>
    </disk>
    <disk type='volume' device='disk'>
      <driver name='qemu' type='raw'/>
      <source pool='blk-pool0' volume='blk-pool0-vol0'/>
      <target dev='hda' bus='ide'/>
    </disk>
    <disk type='network' device='disk'>
      <driver name='qemu' type='raw'/>
      <source protocol='iscsi' name='iqn.2013-07.com.example:iscsi-
nopool/2'>
        <host name='example.com' port='3260'/>
      </source>
      <auth username='myuser'>
        <secret type='chap' usage='libvirtiscsi'/>
      </auth>
      <target dev='vda' bus='virtio'/>
    </disk>
```

Figure 23.33. Devices - Hard drives, floppy disks, CD-ROMs Example 6

```
    <disk type='network' device='lun'>
      <driver name='qemu' type='raw'/>
      <source protocol='iscsi' name='iqn.2013-07.com.example:iscsi-
nopool/1'>
       iqn.2013-07.com.example:iscsi-pool
        <host name='example.com' port='3260'/>
      </source>
      <auth username='myuser'>
        <secret type='chap' usage='libvirtiscsi'/>
      </auth>
      <target dev='sda' bus='scsi'/>
    </disk>
    <disk type='volume' device='disk'>
      <driver name='qemu' type='raw'/>
      <source pool='iscsi-pool' volume='unit:0:0:1' mode='host'/>
      <auth username='myuser'>
        <secret type='chap' usage='libvirtiscsi'/>
      </auth>
      <target dev='vda' bus='virtio'/>
    </disk>
```

Figure 23.34. Devices - Hard drives, floppy disks, CD-ROMs Example 7

```
    <disk type='volume' device='disk'>
      <driver name='qemu' type='raw'/>
      <source pool='iscsi-pool' volume='unit:0:0:2' mode='direct'/>
      <auth username='myuser'>
        <secret type='chap' usage='libvirtiscsi'/>
      </auth>
      <target dev='vda' bus='virtio'/>
    </disk>
     <disk type='file' device='disk'>
      <driver name='qemu' type='raw' cache='none'/>
      <source file='/tmp/test.img' startupPolicy='optional'/>
      <target dev='sdb' bus='scsi'/>
      <readonly/>
    </disk>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw' discard='unmap'/>
      <source file='/var/lib/libvirt/images/discard1.img'/>
      <target dev='vdb' bus='virtio'/>
      <alias name='virtio-disk1'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x09'
function='0x0'/>
    </disk>
  </devices>
  ...
```

Figure 23.35. Devices - Hard drives, floppy disks, CD-ROMs Example 8

23.18.1.1. Disk element

The **<disk>** element is the main container for describing disks. The attribute **type** can be used with the **<disk>** element. The following types are allowed:

- **file**
- **block**
- **dir**
- **network**

For more information, see the libvirt upstream pages.

23.18.1.2. Source element

Represents the disk source. The disk source depends on the disk type attribute, as follows:

- **<file>** - The **file** attribute specifies the fully-qualified path to the file in which the disk is located.
- **<block>** - The **dev** attribute specifies the fully-qualified path to the host device that serves as the disk.
- **<dir>** - The **dir** attribute specifies the fully-qualified path to the directory used as the disk.
- **<network>** - The **protocol** attribute specifies the protocol used to access the requested image. Possible values are: **nbd**, **isci**, **rbd**, **sheepdog**, and **gluster**.
  - If the **protocol** attribute is **rbd**, **sheepdog**, or **gluster**, an additional attribute, **name** is mandatory. This attribute specifies which volume and image will be used.
  - If the **protocol** attribute is **nbd**, the **name** attribute is optional.
  - If the **protocol** attribute is **isci**, the **name** attribute may include a logical unit number, separated from the target's name with a slash. For example: iqn.2013-07.com.example:iscsi-pool/1. If not specified, the default LUN is zero.
- **<volume>** - The underlying disk source is represented by the **pool** and **volume** attributes.
  - **<pool>** - The name of the storage pool (managed by **libvirt**) where the disk source resides.
  - **<volume>** - The name of the storage volume (managed by **libvirt**) used as the disk source.

    The value for the **volume** attribute is the output from the Name column of a **virsh vol-list [pool-name]**

When the disk type is **network**, the **source** may have zero or more **host** sub-elements used to specify the host physical machines to connect, including: **type='dir'** and **type='network'**. For a **file** disk type which represents a CD-ROM or floppy (the device attribute), it is possible to define the policy for what to do with the disk if the source file is not accessible. This is done by setting the **startupPolicy** attribute with one of the following values:

- **mandatory** causes a failure if missing for any reason. This is the default setting.

- **requisite** causes a failure if missing on boot up, drops if missing on migrate, restore, or revert.
- **optional** drops if missing at any start attempt.

23.18.1.3. Mirror element

This element is present if the hypervisor has started a **BlockCopy** operation, where the **<mirror>** location in the attribute file will eventually have the same contents as the source, and with the file format in attribute format (which might differ from the format of the source). If an attribute ready is present, then it is known the disk is ready to pivot; otherwise, the disk is probably still copying. For now, this element only valid in output; it is ignored on input.

23.18.1.4. Target element

The **<target>** element controls the bus or device under which the disk is exposed to the guest virtual machine operating system. The **dev** attribute indicates the logical device name. The actual device name specified is not guaranteed to map to the device name in the guest virtual machine operating system. The optional bus attribute specifies the type of disk device to emulate; possible values are driver-specific, with typical values being **ide**, **scsi**, **virtio**, **kvm**, **usb** or **sata**. If omitted, the bus type is inferred from the style of the device name. For example, a device named **'sda'** will typically be exported using a SCSI bus. The optional attribute **tray** indicates the tray status of the removable disks (for example, CD-ROM or Floppy disk), where the value can be either **open** or **closed**. The default setting is **closed**.

23.18.1.5. iotune element

The optional **<iotune>** element provides the ability to provide additional per-device I/O tuning, with values that can vary for each device (contrast this to the **blkiotune** element, which applies globally to the domain). This element has the following optional sub-elements (note that any sub-element not specified or at all or specified with a value of **0** implies no limit):

- **<total_bytes_sec>** - The total throughput limit in bytes per second. This element cannot be used with **<read_bytes_sec>** or **<write_bytes_sec>**.
- **<read_bytes_sec>** - The read throughput limit in bytes per second.
- **<write_bytes_sec>** - The write throughput limit in bytes per second.
- **<total_iops_sec>** - The total I/O operations per second. This element cannot be used with **<read_iops_sec>** or **<write_iops_sec>**.
- **<read_iops_sec>** - The read I/O operations per second.
- **<write_iops_sec>** - The write I/O operations per second.

23.18.1.6. Driver element

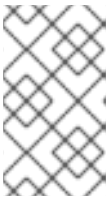
The optional **<driver>** element allows specifying further details related to the hypervisor driver that is used to provide the disk. The following options may be used:

- If the hypervisor supports multiple back-end drivers, the **name** attribute selects the primary back-end driver name, while the optional type attribute provides the sub-type.

- The optional **cache** attribute controls the cache mechanism. Possible values are: **default**, **none**, **writethrough**, **writeback**, **directsync** (similar to **writethrough**, but it bypasses the host physical machine page cache) and **unsafe** (host physical machine may cache all disk I/O, and sync requests from guest virtual machines are ignored).

- The optional **error_policy** attribute controls how the hypervisor behaves on a disk read or write error. Possible values are **stop**, **report**, **ignore**, and **enospace**. The default setting of **error_policy** is **report**. There is also an optional **rerror_policy** that controls behavior for read errors only. If no **rerror_policy** is given, **error_policy** is used for both read and write errors. If **rerror_policy** is given, it overrides the **error_policy** for read errors. Also note that **enospace** is not a valid policy for read errors, so if **error_policy** is set to **enospace** and **no rerror_policy** is given, the read error default setting, **report** will be used.

- The optional **io** attribute controls specific policies on I/O; **kvm** guest virtual machines support **threads** and **native**. The optional **ioeventfd** attribute allows users to set domain I/O asynchronous handling for virtio disk devices. The default is determined by the hypervisor. Accepted values are **on** and **off**. Enabling this allows the guest virtual machine to be executed while a separate thread handles I/O. Typically, guest virtual machines experiencing high system CPU utilization during I/O will benefit from this. On the other hand, an overloaded host physical machine can increase guest virtual machine I/O latency. However, it is recommended that you do not change the default setting, and allow the hypervisor to determine the setting.



NOTE

The **ioeventfd** attribute is included in the **<driver>** element of the **disk** XML section and also the **<driver>** element of the **device** XML section. In the former case, it influences the virtIO disk, and in the latter case the SCSI disk.

- The optional **event_idx** attribute controls some aspects of device event processing and can be set to either **on** or **off**. If set to **on**, it will reduce the number of interrupts and exits for the guest virtual machine. The default is determined by the hypervisor and the default setting is **on**. When this behavior is not required, setting **off** forces the feature off. However, it is highly recommended that you not change the default setting, and allow the hypervisor to dictate the setting.

- The optional **copy_on_read** attribute controls whether to copy the read backing file into the image file. The accepted values can be either **on** or **<off>**. **copy-on-read** avoids accessing the same backing file sectors repeatedly, and is useful when the backing file is over a slow network. By default **copy-on-read** is **off**.

- The **discard='unmap'** can be set to enable discard support. The same line can be replaced with **discard='ignore'** to disable. **discard='ignore'** is the default setting.

23.18.1.7. Additional Device Elements

The following attributes may be used within the **device** element:

- **<boot>** - Specifies that the disk is bootable.

  Additional boot values

  - **<order>** - Determines the order in which devices will be tried during boot sequence.

- **<per-device>** Boot elements cannot be used together with general boot elements in the BIOS boot loader section.

- **<encryption>** - Specifies how the volume is encrypted.
- **<readonly>** - Indicates the device cannot be modified by the guest virtual machine virtual machine. This setting is the default for disks with **attribute <device='cdrom'>**.
- **<shareable>** Indicates the device is expected to be shared between domains (as long as hypervisor and operating system support this). If **shareable** is used, **cache='no'** should be used for that device.
- **<transient>** - Indicates that changes to the device contents should be reverted automatically when the guest virtual machine exits. With some hypervisors, marking a disk **transient** prevents the domain from participating in migration or snapshots.
- **<serial>** - Specifies the serial number of guest virtual machine's hard drive. For example, **<serial>**WD-WMAP9A966149**</serial>**.
- **<wwn>** - Specifies the World Wide Name (WWN) of a virtual hard disk or CD-ROM drive. It must be composed of 16 hexadecimal digits.
- **<vendor>** - Specifies the vendor of a virtual hard disk or CD-ROM device. It must not be longer than 8 printable characters.
- **<product>** - Specifies the product of a virtual hard disk or CD-ROM device. It must not be longer than 16 printable characters
- **<host>** - Supports the following attributes:
  - **name** - specifies the host name
  - **port** - specifies the port number
  - **transport** - specifies the transport type
  - **socket** - specifies the path to the socket

  The meaning of this element and the number of the elements depend on the **protocol** attribute as shown in Additional host attributes based on the protocol

  Additional host attributes based on the protocol

  - **nbd** - Specifies a server running ***nbd-server*** and may only be used for only one host physical machine. The default port for this protcol is **10809**.
  - **rbd** - Monitors servers of RBD type and may be used for one or more host physical machines.
  - **sheepdog** - Specifies one of the ***sheepdog*** servers (default is localhost:7000) and can be used with one or none of the host physical machines.
  - **gluster** - Specifies a server running a **glusterd** daemon and may be used for only only one host physical machine. The valid values for transport attribute are **tcp**, **rdma** or **unix**. If nothing is specified, **tcp** is assumed. If transport is **unix**, the **socket** attribute specifies path to ***unix*** socket.

- **<address>** - Ties the disk to a given slot of a controller. The actual **<controller>** device can often be inferred but it can also be explicitly specified. The **type** attribute is mandatory, and is typically **pci** or **drive**. For a **pci** controller, additional attributes for **bus**, **slot**, and **function** must be present, as well as optional **domain** and **multifunction**. **multifunction** defaults to **off**. For a **drive** controller, additional attributes **controller**, **bus**, **target**, and **unit** are available, each with a default setting of **0**.

- **auth** - Provides the authentication credentials needed to access the source. It includes a mandatory attribute **username**, which identifies the user name to use during authentication, as well as a sub-element **secret** with mandatory attribute **type**.

- **geometry** - Provides the ability to override geometry settings. This mostly useful for S390 DASD-disks or older DOS-disks. It can have the following parameters:

  - **cyls** - Specifies the number of cylinders.

  - **heads** - Specifies the number of heads.

  - **secs** - Specifies the number of sectors per track.

  - **trans** - Specifies the BIOS-Translation-Modes and can have the following values: **none**, **lba** or **auto**.

- **blockio** - Allows the block device to be overridden with any of the block device properties listed below:

  blockio options

  - **logical_block_size** - Reports to the guest virtual machine operating system and describes the smallest units for disk I/O.

  - **physical_block_size** - Reports to the guest virtual machine operating system and describes the disk's hardware sector size, which can be relevant for the alignment of disk data.

23.18.2. File Systems

The file systems directory on the host physical machine can be accessed directly from the guest virtual machine.

```
[...]
 <devices>
   <filesystem type='template'>
     <source name='my-vm-template'/>
     <target dir='/'/>
   </filesystem>
   <filesystem type='mount' accessmode='passthrough'>
     <driver type='path' wrpolicy='immediate'/>
     <source dir='/export/to/guest'/>
     <target dir='/import/from/host'/>
     <readonly/>
   </filesystem>
   [...]
 </devices>
 [...]
```

Figure 23.36. Devices - file systems

The **filesystem** attribute has the following possible values:

- **type='mount'** - Specifies the host physical machine directory to mount in the guest virtual machine. This is the default type if one is not specified. This mode also has an optional sub-element **driver**, with an attribute **type='path'** or **type='handle'**. The driver block has an optional attribute **wrpolicy** that further controls interaction with the host physical machine page cache; omitting the attribute reverts to the default setting, while specifying a value **immediate** means that a host physical machine write back is immediately triggered for all pages touched during a guest virtual machine file write operation.
- **type='template'** - Specifies the OpenVZ file system template and is only used by OpenVZ driver.
- **type='file'** - Specifies that a host physical machine file will be treated as an image and mounted in the guest virtual machine. This file system format will be auto-detected and is only used by LXC driver.
- **type='block'** - Specifies the host physical machine block device to mount in the guest virtual machine. The file system format will be auto-detected and is only used by the LXC driver.
- **type='ram'** - Specifies that an in-memory file system, using memory from the host physical machine operating system will be used. The source element has a single attribute **usage**, which gives the memory usage limit in kibibytes and is only used by LXC driver.
- **type='bind'** - Specifies a directory inside the guest virtual machine which will be bound to another directory inside the guest virtual machine. This element is only used by LXC driver.
- **accessmode** - Specifies the security mode for accessing the source. Currently, this only works with **type='mount'** for the KVM driver. The possible values are:
  - **passthrough** - Specifies that the source is accessed with the user's permission settings that are set from inside the guest virtual machine. This is the default **accessmode** if one is not specified.

- **mapped** - Specifies that the source is accessed with the permission settings of the hypervisor.
- **squash** - Similar to **'passthrough'**, the exception is that failure of privileged operations like **chown** are ignored. This makes a passthrough-like mode usable for people who run the hypervisor as non-root.

- **source** - Specifies that the resource on the host physical machine that is being accessed in the guest virtual machine. The **name** attribute must be used with **<type='template'>**, and the **dir** attribute must be used with **<type='mount'>**. The **usage** attribute is used with **<type='ram'>** to set the memory limit in KB.
- **target** - Dictates where the source drivers can be accessed in the guest virtual machine. For most drivers, this is an automatic mount point, but for KVM this is merely an arbitrary string tag that is exported to the guest virtual machine as a hint for where to mount.
- **readonly** - Enables exporting the file system as a read-only mount for a guest virtual machine. By default **read-write** access is given.
- **space_hard_limit** - Specifies the maximum space available to this guest virtual machine's file system.
- **space_soft_limit** - Specifies the maximum space available to this guest virtual machine's file system. The container is permitted to exceed its soft limits for a grace period of time. Afterwards the hard limit is enforced.

23.18.3. Device Addresses

Many devices have an optional **<address>** sub-element to describe where the device placed on the virtual bus is presented to the guest virtual machine. If an address (or any optional attribute within an address) is omitted on input, libvirt will generate an appropriate address; but an explicit address is required if more control over layout is required. See below for device examples including an address element.

Every address has a mandatory attribute ***type*** that describes which bus the device is on. The choice of which address to use for a given device is constrained in part by the device and the architecture of the guest virtual machine. For example, a disk device uses **type='disk'**, while a console device would use **type='pci'** on the 32-bit AMD and Intel, or AMD64 and Intel 64, guest virtual machines, or **type='spapr-vio'** on PowerPC64 pseries guest virtual machines. Each address **<type>** has additional optional attributes that control where on the bus the device will be placed. The additional attributes are as follows:

- **type='pci'** - PCI addresses have the following additional attributes:
  - **domain** (a 2-byte hex integer, not currently used by KVM)
  - **bus** (a hex value between 0 and 0xff, inclusive)
  - **slot** (a hex value between 0x0 and 0x1f, inclusive)
  - **function** (a value between 0 and 7, inclusive)
  - Also available is the **multi-function** attribute, which controls turning on the multi-function bit for a particular slot or function in the PCI control register. This multi-function attribute defaults to **'off'**, but should be set to **'on'** for function 0 of a slot that will have multiple

functions used.

- **type='drive'** - **drive** addresses have the following additional attributes:
  - **controller** - (a 2-digit controller number)
  - **bus** - (a 2-digit bus number)
  - **target** - (a 2-digit bus number)
  - **unit** - (a 2-digit unit number on the bus)
- **type='virtio-serial'** - Each **virtio-serial** address has the following additional attributes:
  - **controller** - (a 2-digit controller number)
  - **bus** - (a 2-digit bus number)
  - **slot** - (a 2-digit slot within the bus)
- **type='ccid'** - A CCID address, used for smart-cards, has the following additional attributes:
  - **bus** - (a 2-digit bus number)
  - **slot** - (a 2-digit slot within the bus)
- **type='usb'** - USB addresses have the following additional attributes:
  - **bus** - (a hex value between 0 and 0xfff, inclusive)
  - **port** - (a dotted notation of up to four octets, such as 1.2 or 2.1.3.1)
- **type='spapr-vio'** - On PowerPC pseries guest virtual machines, devices can be assigned to the SPAPR-VIO bus. It has a flat 64-bit address space; by convention, devices are generally assigned at a non-zero multiple of 0x1000, but other addresses are valid and permitted by **libvirt**. The additional **reg** attribute, which determines the hex value address of the starting register, can be assigned to this attribute.

23.18.4. Controllers

Depending on the guest virtual machine architecture, it is possible to assign many virtual devices to a single bus. Under normal circumstances libvirt can automatically infer which controller to use for the bus. However, it may be necessary to provide an explicit **<controller>** element in the guest virtual machine XML:

```
  ...
  <devices>
    <controller type='ide' index='0'/>
    <controller type='virtio-serial' index='0' ports='16' vectors='4'/>
    <controller type='virtio-serial' index='1'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x0a'
function='0x0'/>
    <controller type='scsi' index='0' model='virtio-scsi'
num_queues='8'/>
    </controller>
    ...
  </devices>
  ...
```

Figure 23.37. Controller Elements

Each controller has a mandatory attribute **type**, which must be one of **"ide", "fdc", "scsi", "sata", "usb", "ccid", or "virtio-serial"**, and a mandatory attribute **index** which is the decimal integer describing in which order the bus controller is encountered (for use in controller attributes of **address** elements). The **"virtio-serial"** controller has two additional optional attributes, **ports** and **vectors**, which control how many devices can be connected through the controller.

A **<controller type='scsi'>** has an optional attribute **model**, which is one of **"auto", "buslogic", "ibmvscsi", "lsilogic", "lsias1068", "virtio-scsi or "vmpvscsi"**. The **<controller type='scsi'>** also has an attribute **num_queues** which enables multi-queue support for the number of queues specified. In addition, a **ioeventfd** attribute can be used, which specifies whether the controller should use asynchronous handling on the SCSI disk. Accepted values are "on" and "off".

A **"usb"** controller has an optional attribute **model**, which is one of **"piix3-uhci", "piix4-uhci", "ehci", "ich9-ehci1", "ich9-uhci1", "ich9-uhci2", "ich9-uhci3", "vt82c686b-uhci", "pci-ohci" or "nec-xhci"**. Additionally, if the USB bus needs to be explicitly disabled for the guest virtual machine, **model='none'** may be used. The PowerPC64 "spapr-vio" addresses do not have an associated controller.

For controllers that are themselves devices on a PCI or USB bus, an optional sub-element **address** can specify the exact relationship of the controller to its master bus, with semantics given above.

USB companion controllers have an optional sub-element **master** to specify the exact relationship of the companion to its master controller. A companion controller is on the same bus as its master, so the companion index value should be equal.

```
...
<devices>
  <controller type='usb' index='0' model='ich9-ehci1'>
    <address type='pci' domain='0' bus='0' slot='4' function='7'/>
  </controller>
  <controller type='usb' index='0' model='ich9-uhci1'>
    <master startport='0'/>
    <address type='pci' domain='0' bus='0' slot='4' function='0'
multifunction='on'/>
  </controller>
  ...
</devices>
...
```

Figure 23.38. Devices - controllers - USB

23.18.5. Device Leases

When using a lock manager, you have the option to record device leases against a guest virtual machine. The lock manager will ensure that the guest virtual machine does not start unless the leases can be acquired. When configured using conventional management tools, the following section of the domain XML is affected:

```
...
<devices>
  ...
  <lease>
    <lockspace>somearea</lockspace>
    <key>somekey</key>
    <target path='/some/lease/path' offset='1024'/>
  </lease>
  ...
</devices>
...
```

Figure 23.39. Devices - device leases

The **lease** section can have the following arguments:

- **lockspace** - An arbitrary string that identifies lockspace within which the key is held. Lock managers may impose extra restrictions on the format, or length of the lockspace name.
- **key** - An arbitrary string that uniquely identifies the lease to be acquired. Lock managers may impose extra restrictions on the format, or length of the key.
- **target** - The fully qualified path of the file associated with the lockspace. The offset specifies where the lease is stored within the file. If the lock manager does not require a offset, set this value to **0**.

23.18.6. Host Physical Machine Device Assignment

23.18.6.1. USB / PCI devices

The host physical machine's USB and PCI devices can be passed through to the guest virtual machine using the **hostdev** element, by modifying the host physical machine using a management tool, configure the following section of the domain XML file:

```
...
<devices>
  <hostdev mode='subsystem' type='usb'>
    <source startupPolicy='optional'>
      <vendor id='0x1234'/>
      <product id='0xbeef'/>
    </source>
    <boot order='2'/>
  </hostdev>
</devices>
...
```

Figure 23.40. Devices - Host physical machine device assignment

Alternatively, the following can also be done:

```
...
<devices>
  <hostdev mode='subsystem' type='pci' managed='yes'>
    <source>
      <address bus='0x06' slot='0x02' function='0x0'/>
    </source>
    <boot order='1'/>
    <rom bar='on' file='/etc/fake/boot.bin'/>
  </hostdev>
</devices>
...
```

Figure 23.41. Devices - Host physical machine device assignment alternative

Alternatively, the following can also be done:

```
...
<devices>
  <hostdev mode='subsystem' type='scsi'>
    <source>
      <adapter name='scsi_host0'/>
      <address type='scsi' bus='0' target='0' unit='0'/>
    </source>
    <readonly/>
    <address type='drive' controller='0' bus='0' target='0' unit='0'/>
  </hostdev>
</devices>
..
```

Figure 23.42. Devices - host physical machine scsi device assignment

The components of this section of the domain XML are as follows:

Table 23.16. Host physical machine device assignment elements

Parameter	Description

Parameter	Description
hostdev	This is the main element for describing host physical machine devices. It accepts the following options: • **mode** - the value is always **subsystem** for USB and PCI devices. • **type** - **usb** for USB devices and **pci** for PCI devices. • **managed** - Toggles the *Managed mode* of the device: ◦ When set to **yes** for a PCI device, it attaches to the guest machine and detaches from the guest machine and re-attaches to the host machine as necessary. **managed='yes'** is recommended for general use of device assignment. ◦ When set to **no** or omitted for PCI and for USB devices, the device stays attached to the guest. To make the device available to the host, the user must use the argument **virNodeDeviceDettach** or the **virsh nodedev-dettach** command before starting the guest or hot plugging the device. In addition, they must use **virNodeDeviceReAttach** or **virsh nodedev-reattach** after hot-unplugging the device or stopping the guest. **managed='no'** is mainly recommended for devices that are intended to be dedicated to a specific guest.

Parameter	Description
source	Describes the device as seen from the host physical machine. The USB device can be addressed by vendor or product ID using the **vendor** and **product** elements or by the device's address on the host physical machines using the **address** element. PCI devices on the other hand can only be described by their address. Note that the source element of USB devices may contain a **startupPolicy** attribute which can be used to define a rule for what to do if the specified host physical machine USB device is not found. The attribute accepts the following values: • **mandatory** - Fails if missing for any reason (the default). • **requisite** - Fails if missing on boot up, drops if missing on migrate/restore/revert. • **optional** - Drops if missing at any start attempt.
vendor, product	These elements each have an **id** attribute that specifies the USB vendor and product ID. The IDs can be given in decimal, hexadecimal (starting with 0x) or octal (starting with 0) form.
boot	Specifies that the device is bootable. The attribute's order determines the order in which devices will be tried during boot sequence. The per-device boot elements cannot be used together with general boot elements in BIOS boot loader section.
rom	Used to change how a PCI device's ROM is presented to the guest virtual machine. The optional **bar** attribute can be set to **on** or **off**, and determines whether or not the device's ROM will be visible in the guest virtual machine's memory map. (In PCI documentation, the **rom bar** setting controls the presence of the Base Address Register for the ROM). If no **rom bar** is specified, the default setting will be used. The optional **file** attribute is used to point to a binary file to be presented to the guest virtual machine as the device's ROM BIOS. This can be useful for example to provide a PXE boot ROM for a virtual function of an SR-IOV capable ethernet device (which has no boot ROMs for the VFs).

Parameter	Description
address	Also has a **bus** and **device** attribute to specify the USB bus and device number the device appears at on the host physical machine. The values of these attributes can be given in decimal, hexadecimal (starting with 0x) or octal (starting with 0) form. For PCI devices, the element carries 3 attributes allowing to designate the device as can be found with **lspci** or with **virsh nodedev-list**.

23.18.6.2. Block / character devices

The host physical machine's block / character devices can be passed through to the guest virtual machine by using management tools to modify the domain XML **hostdev** element. Note that this is only possible with container-based virtualization.

```
...
<hostdev mode='capabilities' type='storage'>
  <source>
    <block>/dev/sdf1</block>
  </source>
</hostdev>
...
```

Figure 23.43. Devices - Host physical machine device assignment block character devices

An alternative approach is this:

```
...
<hostdev mode='capabilities' type='misc'>
  <source>
    <char>/dev/input/event3</char>
  </source>
</hostdev>
...
```

Figure 23.44. Devices - Host physical machine device assignment block character devices alternative 1

Another alternative approach is this:

```
...
<hostdev mode='capabilities' type='net'>
  <source>
    <interface>eth0</interface>
  </source>
</hostdev>
...
```

Figure 23.45. Devices - Host physical machine device assignment block character devices alternative 2

The components of this section of the domain XML are as follows:

Table 23.17. Block / character device elements

Parameter	Description
hostdev	This is the main container for describing host physical machine devices. For block/character devices, passthrough **mode** is always **capabilities**, and **type** is **block** for a block device and **char** for a character device.
source	This describes the device as seen from the host physical machine. For block devices, the path to the block device in the host physical machine operating system is provided in the nested **block** element, while for character devices, the **char** element is used.

23.18.7. Redirected devices

USB device redirection through a character device is configured by modifying the following section of the domain XML:

```
  ...
  <devices>
    <redirdev bus='usb' type='tcp'>
      <source mode='connect' host='localhost' service='4000'/>
      <boot order='1'/>
    </redirdev>
    <redirfilter>
      <usbdev class='0x08' vendor='0x1234' product='0xbeef'
version='2.00' allow='yes'/>
      <usbdev allow='no'/>
    </redirfilter>
  </devices>
  ...
```

Figure 23.46. Devices - redirected devices

The components of this section of the domain XML are as follows:

Table 23.18. Redirected device elements

Parameter	Description
redirdev	This is the main container for describing redirected devices. **bus** must be **usb** for a USB device. An additional attribute type is required, matching one of the supported serial device types, to describe the host physical machine side of the tunnel: **type='tcp'** or **type='spicevmc'** (which uses the usbredir channel of a SPICE graphics device) are typical. The **redirdev** element has an optional sub-element, **address**, which can tie the device to a particular controller. Further sub-elements, such as **source**, may be required according to the given **type**, although a **target** sub-element is not required (since the consumer of the character device is the hypervisor itself, rather than a device visible in the guest virtual machine).
boot	Specifies that the device is bootable. The **order** attribute determines the order in which devices will be tried during boot sequence. The per-device boot elements cannot be used together with general boot elements in BIOS boot loader section.
redirfilter	This is used for creating the filter rule to filter out certain devices from redirection. It uses sub-element **usbdev** to define each filter rule. The **class** attribute is the USB Class code.

23.18.8. Smartcard Devices

A virtual smartcard device can be supplied to the guest virtual machine via the **smartcard** element. A USB smartcard reader device on the host physical machine cannot be used on a guest virtual machine with device passthrough. This is because it cannot be made available to both the host physical machine and guest virtual machine, and can lock the host physical machine computer when it is removed from the guest virtual machine. Therefore, some hypervisors provide a specialized virtual device that can present a smartcard interface to the guest virtual machine, with several modes for describing how the credentials are obtained from the host physical machine or even a from a channel created to a third-party smartcard provider.

Configure USB device redirection through a character device with management tools to modify the following section of the domain XML:

```
...
<devices>
  <smartcard mode='host'/>
  <smartcard mode='host-certificates'>
    <certificate>cert1</certificate>
    <certificate>cert2</certificate>
    <certificate>cert3</certificate>
    <database>/etc/pki/nssdb/</database>
  </smartcard>
  <smartcard mode='passthrough' type='tcp'>
    <source mode='bind' host='127.0.0.1' service='2001'/>
    <protocol type='raw'/>
    <address type='ccid' controller='0' slot='0'/>
  </smartcard>
  <smartcard mode='passthrough' type='spicevmc'/>
</devices>
...
```

Figure 23.47. Devices - smartcard devices

The **smartcard** element has a mandatory attribute **mode**. In each mode, the guest virtual machine sees a device on its USB bus that behaves like a physical USB CCID (Chip/Smart Card Interface Device) card.

The mode attributes are as follows:

Table 23.19. Smartcard mode elements

Parameter	Description
mode='host'	In this mode, the hypervisor relays all requests from the guest virtual machine into direct access to the host physical machine's smartcard via NSS. No other attributes or sub-elements are required. See below about the use of an optional **address** sub-element.

Parameter	Description
mode='host-certificates'	This mode allows you to provide three NSS certificate names residing in a database on the host physical machine, rather than requiring a smartcard to be plugged into the host physical machine. These certificates can be generated using the command **certutil -d /etc/pki/nssdb -x -t CT,CT,CT -S -s CN=cert1 -n cert1,** and the resulting three certificate names must be supplied as the content of each of three **certificate** sub-elements. An additional sub-element **database** can specify the absolute path to an alternate directory (matching the **-d** flag of the **certutil** command when creating the certificates); if not present, it defaults to **/etc/pki/nssdb**.
mode='passthrough'	Using this mode allows you to tunnel all requests through a secondary character device to a third-party provider (which may in turn be communicating to a smartcard or using three certificate files, rather than having the hypervisor directly communicate with the host physical machine. In this mode of operation, an additional attribute **type** is required, matching one of the supported serial device types, to describe the host physical machine side of the tunnel; **type='tcp'** or **type='spicevmc'** (which uses the smartcard channel of a SPICE graphics device) are typical. Further sub-elements, such as **source**, may be required according to the given type, although a **target** sub-element is not required (since the consumer of the character device is the hypervisor itself, rather than a device visible in the guest virtual machine).

Each mode supports an optional sub-element **address**, which fine-tunes the correlation between the smartcard and a ccid bus controller. For more information, see Section 23.18.3, “Device Addresses”).

23.18.9. Network Interfaces

Modify the network interface devices using management tools to configure the following part of the domain XML:

```
...
<devices>
  <interface type='direct' trustGuestRxFilters='yes'>
    <source dev='eth0'/>
    <mac address='52:54:00:5d:c7:9e'/>
    <boot order='1'/>
    <rom bar='off'/>
  </interface>
</devices>
...
```

Figure 23.48. Devices - network interfaces

There are several possibilities for configuring the network interface for the guest virtual machine. This is done by setting a value to the interface element's type attribute. The following values may be used:

- **"direct"** - Attaches the guest virtual machine's NIC to the physical NIC on the host physical machine. For details and an example, see Section 23.18.9.6, “Direct attachment to physical interfaces”.
- **"network"** - This is the recommended configuration for general guest virtual machine connectivity on host physical machines with dynamic or wireless networking configurations. For details and an example, see Section 23.18.9.1, “Virtual networks”.
- **"bridge"** - This is the recommended configuration setting for guest virtual machine connectivity on host physical machines with static wired networking configurations. For details and an example, see Section 23.18.9.2, “Bridge to LAN”.
- **"ethernet"** - Provides a means for the administrator to execute an arbitrary script to connect the guest virtual machine's network to the LAN. For details and an example, see Section 23.18.9.5, “Generic Ethernet connection”.
- **"hostdev"** - Allows a PCI network device to be directly assigned to the guest virtual machine using generic device passthrough. For details and an example, see Section 23.18.9.7, “PCI passthrough”.
- **"mcast"** - A multicast group can be used to represent a virtual network. For details and an example, see Section 23.18.9.8, “Multicast tunnel” .
- **"user"** - Using the user option sets the user space SLIRP stack parameters provides a virtual LAN with NAT to the outside world. For details and an example, see Section 23.18.9.4, “User space SLIRP stack”.
- **"server"** - Using the server option creates a TCP client-server architecture in order to provide a virtual network where one guest virtual machine provides the server end of the network and all other guest virtual machines are configured as clients. For details and an example, see Section 23.18.9.9, “TCP tunnel”.

Each of these options has a link to give more details. Additionally, each **<interface>** element can be defined with an optional **<trustGuestRxFilters>** attribute which allows host physical machine to detect and trust reports received from the guest virtual machine. These reports are sent each time the interface receives changes to the filter. This includes changes to the primary MAC address, the device address filter, or the vlan configuration. The **<trustGuestRxFilters>** attribute is disabled by default

for security reasons. It should also be noted that support for this attribute depends on the guest network device model as well as on the host physical machine's connection type. Currently, it is only supported for the virtio device models and for macvtap connections on the host physical machine. A simple use case where it is recommended to set the optional parameter **<trustGuestRxFilters>** is if you want to give your guest virtual machines the permission to control host physical machine side filters, as any filters that are set by the guest will also be mirrored on the host.

In addition to the attributes listed above, each **<interface>** element can take an optional **<address>** sub-element that can tie the interface to a particular PCI slot, with attribute **type='pci'**. For more information, see Section 23.18.3, “Device Addresses”.

23.18.9.1. Virtual networks

This is the recommended configuration for general guest virtual machine connectivity on host physical machines with dynamic or wireless networking configurations (or multi-host physical machine environments where the host physical machine hardware details, which are described separately in a **<network>** definition). In addition, it provides a connection with details that are described by the named network definition. Depending on the virtual network's **forward mode** configuration, the network may be totally isolated (no **<forward>** element given), using NAT to connect to an explicit network device or to the default route (**forward mode='nat'**), routed with no NAT (**forward mode='route'**), or connected directly to one of the host physical machine's network interfaces (using macvtap) or bridge devices (**forward mode='bridge|private|vepa|passthrough'**)

For networks with a forward mode of **bridge**, **private**, **vepa**, and **passthrough**, it is assumed that the host physical machine has any necessary DNS and DHCP services already set up outside the scope of libvirt. In the case of isolated, nat, and routed networks, DHCP and DNS are provided on the virtual network by libvirt, and the IP range can be determined by examining the virtual network configuration with **virsh net-dumpxml [networkname]**. The 'default' virtual network, which is set up out of the box, uses NAT to connect to the default route and has an IP range of 192.168.122.0/255.255.255.0. Each guest virtual machine will have an associated tun device created with a name of vnetN, which can also be overridden with the **<target>** element (refer to Section 23.18.9.11, “Overriding the target element”).

When the source of an interface is a network, a port group can be specified along with the name of the network; one network may have multiple portgroups defined, with each portgroup containing slightly different configuration information for different classes of network connections. Also, similar to **<direct>** network connections (described below), a connection of type **network** may specify a **<virtualport>** element, with configuration data to be forwarded to a 802.1Qbg or 802.1Qbh-compliant *Virtual Ethernet Port Aggregator* (VEPA)switch, or to an Open vSwitch virtual switch.

Since the type of switch is dependent on the configuration setting in the **<network>** element on the host physical machine, it is acceptable to omit the **<virtualport type>** attribute. You will need to specify the **<virtualport type>** either once or many times. When the domain starts up a complete **<virtualport>** element is constructed by merging together the type and attributes defined. This results in a newly-constructed virtual port. Note that the attributes from lower virtual ports cannot make changes on the attributes defined in higher virtual ports. Interfaces take the highest priority, while port group is lowest priority.

For example, to create a properly working network with both an 802.1Qbh switch and an Open vSwitch switch, you may choose to specify no type, but both **profileid** and an **interfaceid** must be supplied. The other attributes to be filled in from the virtual port, such as such as **managerid**, **typeid**, or **profileid**, are optional.

If you want to limit a guest virtual machine to connecting only to certain types of switches, you can specify the virtualport type, and only switches with the specified port type will connect. You can also

further limit switch connectivity by specifying additional parameters. As a result, if the port was specified and the host physical machine's network has a different type of virtualport, the connection of the interface will fail. The virtual network parameters are defined using management tools that modify the following part of the domain XML:

```
...
<devices>
  <interface type='network'>
    <source network='default'/>
  </interface>
  ...
  <interface type='network'>
    <source network='default' portgroup='engineering'/>
    <target dev='vnet7'/>
    <mac address="00:11:22:33:44:55"/>
    <virtualport>
      <parameters instanceid='09b11c53-8b5c-4eeb-8f00-d84eaa0aaa4f'/>
    </virtualport>

  </interface>
</devices>
...
```

Figure 23.49. Devices - network interfaces- virtual networks

23.18.9.2. Bridge to LAN

As mentioned in, Section 23.18.9, “Network Interfaces”, this is the recommended configuration setting for guest virtual machine connectivity on host physical machines with static wired networking configurations.

Bridge to LAN provides a bridge from the guest virtual machine directly onto the LAN. This assumes there is a bridge device on the host physical machine which has one or more of the host physical machines physical NICs enslaved. The guest virtual machine will have an associated **tun** device created with a name of **<vnetN>**, which can also be overridden with the **<target>** element (refer to Section 23.18.9.11, “Overriding the target element”). The **<tun>** device will be enslaved to the bridge. The IP range or network configuration is the same as what is used on the LAN. This provides the guest virtual machine full incoming and outgoing network access, just like a physical machine.

On Linux systems, the bridge device is normally a standard Linux host physical machine bridge. On host physical machines that support Open vSwitch, it is also possible to connect to an Open vSwitch bridge device by adding **virtualport type='openvswitch'/** to the interface definition. The Open vSwitch type **virtualport** accepts two parameters in its **parameters** element: an **interfaceid** which is a standard UUID used to uniquely identify this particular interface to Open vSwitch (if you do no specify one, a random **interfaceid** will be generated when first defining the interface), and an optional **profileid** which is sent to Open vSwitch as the interfaces **<port-profile>**. To set the bridge to LAN settings, use a management tool that will configure the following part of the domain XML:

```
  ...
  <devices>
    ...
    <interface type='bridge'>
      <source bridge='br0'/>
    </interface>
    <interface type='bridge'>
      <source bridge='br1'/>
      <target dev='vnet7'/>
      <mac address="00:11:22:33:44:55"/>
    </interface>
    <interface type='bridge'>
      <source bridge='ovsbr'/>
      <virtualport type='openvswitch'>
        <parameters profileid='menial' interfaceid='09b11c53-8b5c-4eeb-
8f00-d84eaa0aaa4f'/>
      </virtualport>
    </interface>
    ...
  </devices>
```

Figure 23.50. Devices - network interfaces- bridge to LAN

23.18.9.3. Setting a port masquerading range

In cases where you want to set the port masquerading range, set the port as follows:

```
<forward mode='nat'>
   <address start='1.2.3.4' end='1.2.3.10'/>
</forward>  ...
```

Figure 23.51. Port Masquerading Range

These values should be set using the **iptables** commands as shown in Section 17.3, “Network Address Translation”

23.18.9.4. User space SLIRP stack

Setting the user space SLIRP stack parameters provides a virtual LAN with NAT to the outside world. The virtual network has DHCP and DNS services and will give the guest virtual machine an IP addresses starting from 10.0.2.15. The default router is 10.0.2.2 and the DNS server is 10.0.2.3. This networking is the only option for unprivileged users who need their guest virtual machines to have outgoing access.

The user space SLIRP stack parameters are defined in the following part of the domain XML:

```
...
<devices>
  <interface type='user'/>
  ...
  <interface type='user'>
    <mac address="00:11:22:33:44:55"/>
  </interface>
</devices>
...
```

Figure 23.52. Devices - network interfaces- User space SLIRP stack

23.18.9.5. Generic Ethernet connection

This provides a means for the administrator to execute an arbitrary script to connect the guest virtual machine's network to the LAN. The guest virtual machine will have a **<tun>** device created with a name of **vnetN**, which can also be overridden with the **<target>** element. After creating the **tun** device a shell script will be run and complete the required host physical machine network integration. By default, this script is called **/etc/qemu-ifup** but can be overridden (refer to Section 23.18.9.11, “Overriding the target element”).

The generic ethernet connection parameters are defined in the following part of the domain XML:

```
...
<devices>
  <interface type='ethernet'/>
  ...
  <interface type='ethernet'>
    <target dev='vnet7'/>
    <script path='/etc/qemu-ifup-mynet'/>
  </interface>
</devices>
...
```

Figure 23.53. Devices - network interfaces- generic ethernet connection

23.18.9.6. Direct attachment to physical interfaces

This directly attaches the guest virtual machine's NIC to the physical interface of the host physical machine, if the physical interface is specified.

This requires the Linux macvtap driver to be available. One of the following **mode** attribute values **vepa** ('Virtual Ethernet Port Aggregator'), **bridge** or **private** can be chosen for the operation mode of the macvtap device. **vepa** is the default mode.

Manipulating direct attachment to physical interfaces involves setting the following parameters in this section of the domain XML:

```
...
<devices>
  ...
  <interface type='direct'>
    <source dev='eth0' mode='vepa'/>
  </interface>
</devices>
...
```

Figure 23.54. Devices - network interfaces- direct attachment to physical interfaces

The individual modes cause the delivery of packets to behave as shown in Table 23.20, “Direct attachment to physical interface elements”:

Table 23.20. Direct attachment to physical interface elements

Element	Description
vepa	All of the guest virtual machines' packets are sent to the external bridge. Packets whose destination is a guest virtual machine on the same host physical machine as where the packet originates from are sent back to the host physical machine by the VEPA capable bridge (today's bridges are typically not VEPA capable).
bridge	Packets whose destination is on the same host physical machine as where they originate from are directly delivered to the target macvtap device. Both origin and destination devices need to be in bridge mode for direct delivery. If either one of them is in **vepa** mode, a VEPA capable bridge is required.
private	All packets are sent to the external bridge and will only be delivered to a target virtual machine on the same host physical machine if they are sent through an external router or gateway and that device sends them back to the host physical machine. This procedure is followed if either the source or destination device is in private mode.
passthrough	This feature attaches a virtual function of a SR-IOV capable NIC directly to a guest virtual machine without losing the migration capability. All packets are sent to the VF/IF of the configured network device. Depending on the capabilities of the device, additional prerequisites or limitations may apply; for example, this requires kernel 2.6.38 or later.

The network access of directly attached virtual machines can be managed by the hardware switch to which the physical interface of the host physical machine is connected to.

The interface can have additional parameters as shown below, if the switch conforms to the IEEE 802.1Qbg standard. The parameters of the virtualport element are documented in more detail in the IEEE 802.1Qbg standard. The values are network specific and should be provided by the network administrator. In 802.1Qbg terms, the Virtual Station Interface (VSI) represents the virtual interface of a virtual machine.

Note that IEEE 802.1Qbg requires a non-zero value for the VLAN ID.

Additional elements that can be manipulated are described in Table 23.21, “Direct attachment to physical interface additional elements”:

Table 23.21. Direct attachment to physical interface additional elements

Element	Description
`managerid`	The VSI Manager ID identifies the database containing the VSI type and instance definitions. This is an integer value and the value `0` is reserved.
`typeid`	The VSI Type ID identifies a VSI type characterizing the network access. VSI types are typically managed by network administrator. This is an integer value.
`typeidversion`	The VSI Type Version allows multiple versions of a VSI Type. This is an integer value.
`instanceid`	The VSI Instance ID Identifier is generated when a VSI instance (a virtual interface of a virtual machine) is created. This is a globally unique identifier.
`profileid`	The profile ID contains the name of the port profile that is to be applied onto this interface. This name is resolved by the port profile database into the network parameters from the port profile, and those network parameters will be applied to this interface.

Additional parameters in the domain XML include:

```
...
<devices>
  ...
  <interface type='direct'>
    <source dev='eth0.2' mode='vepa'/>
    <virtualport type="802.1Qbg">
      <parameters managerid="11" typeid="1193047" typeidversion="2"
instanceid="09b11c53-8b5c-4eeb-8f00-d84eaa0aaa4f"/>
    </virtualport>
  </interface>
</devices>
...
```

Figure 23.55. Devices - network interfaces- direct attachment to physical interfaces additional parameters

The interface can have additional parameters as shown below if the switch conforms to the IEEE 802.1Qbh standard. The values are network specific and should be provided by the network administrator.

Additional parameters in the domain XML include:

```
...
<devices>
  ...
  <interface type='direct'>
    <source dev='eth0' mode='private'/>
    <virtualport type='802.1Qbh'>
      <parameters profileid='finance'/>
    </virtualport>
  </interface>
</devices>
...
```

Figure 23.56. Devices - network interfaces - direct attachment to physical interfaces more additional parameters

The **profileid** attribute contains the name of the port profile to be applied to this interface. This name is resolved by the port profile database into the network parameters from the port profile, and those network parameters will be applied to this interface.

23.18.9.7. PCI passthrough

A PCI network device (specified by the **source** element) is directly assigned to the guest virtual machine using generic device passthrough, after first optionally setting the device's MAC address to the configured value, and associating the device with an 802.1Qbh capable switch using an optionally specified **virtualport** element (see the examples of virtualport given above for **type='direct'** network devices). Note that due to limitations in standard single-port PCI ethernet card driver design, only SR-IOV (Single Root I/O Virtualization) virtual function (VF) devices can be assigned in this manner. To assign a standard single-port PCI or PCIe ethernet card to a guest virtual machine, use the traditional **hostdev** device definition.

Note that this "intelligent passthrough" of network devices is very similar to the functionality of a standard **hostdev** device, the difference being that this method allows specifying a MAC address and **virtualport** for the passed-through device. If these capabilities are not required, if you have a standard single-port PCI, PCIe, or USB network card that does not support SR-IOV (and hence would anyway lose the configured MAC address during reset after being assigned to the guest virtual machine domain), or if you are using libvirt version older than 0.9.11, use standard **hostdev** definition to assign the device to the guest virtual machine instead of **interface type='hostdev'**.

```
...
<devices>
  <interface type='hostdev'>
    <driver name='vfio'/>
    <source>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x07'
function='0x0'/>
    </source>
    <mac address='52:54:00:6d:90:02'>
    <virtualport type='802.1Qbh'>
      <parameters profileid='finance'/>
    </virtualport>
  </interface>
</devices>
...
```

Figure 23.57. Devices - network interfaces- PCI passthrough

23.18.9.8. Multicast tunnel

A multicast group can be used to represent a virtual network. Any guest virtual machine with network devices within the same multicast group will communicate with each other, even if they reside across multiple physical host physical machines. This mode may be used as an unprivileged user. There is no default DNS or DHCP support and no outgoing network access. To provide outgoing network access, one of the guest virtual machines should have a second NIC which is connected to one of the first 4 network types in order to provide appropriate routing. The multicast protocol is compatible with protocols used by **user mode** Linux guest virtual machines as well. Note that the source address used must be from the multicast address block. A multicast tunnel is created by manipulating the **interface type** using a management tool and setting it to **mcast**, and providing a **mac address** and **source address**, for example:

```
...
<devices>
  <interface type='mcast'>
    <mac address='52:54:00:6d:90:01'>
    <source address='230.0.0.1' port='5558'/>
  </interface>
</devices>
...
```

Figure 23.58. Devices - network interfaces- multicast tunnel

23.18.9.9. TCP tunnel

Creating a TCP client-server architecture is another way to provide a virtual network where one guest virtual machine provides the server end of the network and all other guest virtual machines are configured as clients. All network traffic between the guest virtual machines is routed through the guest virtual machine that is configured as the server. This model is also available for use to unprivileged users. There is no default DNS or DHCP support and no outgoing network access. To provide outgoing network access, one of the guest virtual machines should have a second NIC which is connected to one of the first 4 network types thereby providing the appropriate routing. A TCP tunnel is created by manipulating the **`interface type`** using a management tool and setting it to**`mcast`**, and providing a **`mac address`** and **`source address`**, for example:

```
...
<devices>
  <interface type='server'>
    <mac address='52:54:00:22:c9:42'>
    <source address='192.168.0.1' port='5558'/>
  </interface>
  ...
  <interface type='client'>
    <mac address='52:54:00:8b:c9:51'>
    <source address='192.168.0.1' port='5558'/>
  </interface>
</devices>
...
```

Figure 23.59. Devices - network interfaces- TCP tunnel

23.18.9.10. Setting NIC driver-specific options

Some NICs may have tunable driver-specific options. These options are set as attributes of the **`driver`** sub-element of the interface definition. These options are set by using management tools to configure the following sections of the domain XML:

```
  <devices>
    <interface type='network'>
      <source network='default'/>
      <target dev='vnet1'/>
      <model type='virtio'/>
      <driver name='vhost' txmode='iothread' ioeventfd='on'
event_idx='off'/>
    </interface>
  </devices>
  ...
```

Figure 23.60. Devices - network interfaces- setting NIC driver-specific options

The following attributes are available for the "virtio" NIC driver:

Table 23.22. virtio NIC driver elements

Parameter	Description
name	The optional **name** attribute forces which type of back-end driver to use. The value can be either **kvm** (a user-space back-end) or **vhost** (a kernel back-end, which requires the vhost module to be provided by the kernel); an attempt to require the vhost driver without kernel support will be rejected. The default setting is **vhost** if the vhost driver is present, but will silently fall back to **kvm** if not.
txmode	Specifies how to handle transmission of packets when the transmit buffer is full. The value can be either **iothread** or **timer**. If set to **iothread**, packet tx is all done in an iothread in the bottom half of the driver (this option translates into adding **"tx=bh"** to the **kvm** command-line "-device" virtio-net-pci option). If set to **timer**, tx work is done in KVM, and if there is more tx data than can be sent at the present time, a timer is set before KVM moves on to do other things; when the timer fires, another attempt is made to send more data. It is not recommended to change this value.
ioeventfd	Sets domain I/O asynchronous handling for the interface device. The default is left to the discretion of the hypervisor. Accepted values are **on** and **off**. Enabling this option allows KVM to execute a guest virtual machine while a separate thread handles I/O. Typically, guest virtual machines experiencing high system CPU utilization during I/O will benefit from this. On the other hand, overloading the physical host machine may also increase guest virtual machine I/O latency. It is not recommended to change this value.
event_idx	The **event_idx** attribute controls some aspects of device event processing. The value can be either **on** or **off**. **on** is the default, which reduces the number of interrupts and exits for the guest virtual machine. In situations where this behavior is sub-optimal, this attribute provides a way to force the feature off. It is not recommended to change this value.

23.18.9.11. Overriding the target element

To override the target element, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
  </interface>
</devices>
...
```

Figure 23.61. Devices - network interfaces- overriding the target element

If no target is specified, certain hypervisors will automatically generate a name for the created tun device. This name can be manually specified, however the name must not start with either **vnet** or **vif**, which are prefixes reserved by libvirt and certain hypervisors. Manually-specified targets using these prefixes will be ignored.

23.18.9.12. Specifying boot order

To specify the boot order, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
    <boot order='1'/>
  </interface>
</devices>
...
```

Figure 23.62. Specifying boot order

In hypervisors which support it, you can set a specific NIC to be used for the network boot. The order of attributes determine the order in which devices will be tried during boot sequence. Note that the per-device boot elements cannot be used together with general boot elements in BIOS boot loader section.

23.18.9.13. Interface ROM BIOS configuration

To specify the ROM BIOS configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
    <rom bar='on' file='/etc/fake/boot.bin'/>
  </interface>
</devices>
...
```

Figure 23.63. Interface ROM BIOS configuration

For hypervisors that support it, you can change how a PCI Network device's ROM is presented to the guest virtual machine. The **bar** attribute can be set to **on** or **off**, and determines whether or not the device's ROM will be visible in the guest virtual machine's memory map. (In PCI documentation, the **rom bar** setting controls the presence of the Base Address Register for the ROM). If no**rom bar**is specified, the KVM default will be used (older versions of KVM used **off** for the default, while newer KVM hypervisors default to **on**). The optional **file** attribute is used to point to a binary file to be presented to the guest virtual machine as the device's ROM BIOS. This can be useful to provide an alternative boot ROM for a network device.

23.18.9.14. Quality of service (QoS)

Incoming and outgoing traffic can be shaped independently to set Quality of Service (QoS). The **bandwidth** element can have at most one inbound and one outbound child elements. Leaving any of these child elements out results in no QoS being applied on that traffic direction. Therefore, to shape only a domain's incoming traffic, use inbound only, and vice versa.

Each of these elements has one mandatory attribute **average** (or **floor** as described below). **Average** specifies the average bit rate on the interface being shaped. In addition, there are two optional attributes:

- **peak** - This attribute specifies the maximum rate at which the bridge can send data, in kilobytes a second. A limitation of this implementation is this attribute in the outbound element is ignored, as Linux ingress filters do not know it yet.
- **burst** - Specifies the amount of bytes that can be burst at peak speed. Accepted values for attributes are integer numbers.

The units for **average** and **peak** attributes are kilobytes per second, whereas **burst** is only set in kilobytes. In addition, inbound traffic can optionally have a **floor** attribute. This guarantees minimal throughput for shaped interfaces. Using the **floor** requires that all traffic goes through one point where QoS decisions can take place. As such, it may only be used in cases where the **interface type='network'/** with a **forward** type of **route**, **nat**, or no forward at all). Noted that within a virtual network, all connected interfaces are required to have at least the inbound QoS set (**average** at least) but the floor attribute does not require specifying **average**. However, **peak** and **burst** attributes still require **average**. At the present time, ingress qdiscs may not have any classes, and therefore, **floor** may only be applied only on inbound and not outbound traffic.

To specify the QoS configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet0'/>
    <bandwidth>
      <inbound average='1000' peak='5000' floor='200' burst='1024'/>
      <outbound average='128' peak='256' burst='256'/>
    </bandwidth>
  </interface>
<devices>
...
```

Figure 23.64. Quality of service

23.18.9.15. Setting VLAN tag (on supported network types only)

To specify the VLAN tag configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <interface type='bridge'>
    <vlan>
      <tag id='42'/>
    </vlan>
    <source bridge='ovsbr0'/>
    <virtualport type='openvswitch'>
      <parameters interfaceid='09b11c53-8b5c-4eeb-8f00-d84eaa0aaa4f'/>
    </virtualport>
  </interface>
<devices>
...
```

Figure 23.65. Setting VLAN tag (on supported network types only)

If the network connection used by the guest virtual machine supports VLAN tagging transparent to the guest virtual machine, an optional **vlan** element can specify one or more VLAN tags to apply to the guest virtual machine's network traffic. Only OpenvSwitch and **type='hostdev'** SR-IOV interfaces support transparent VLAN tagging of guest virtual machine traffic; other interfaces, including standard Linux bridges and libvirt's own virtual networks, do not support it. 802.1Qbh (vn-link) and 802.1Qbg (VEPA) switches provide their own methods (outside of libvirt) to tag guest virtual machine traffic onto specific VLANs. To allow for specification of multiple tags (in the case of VLAN trunking), the **tag** subelement specifies which VLAN tag to use (for example, **tag id='42'/**). If an interface has more than one **vlan** element defined, it is assumed that the user wants to do VLAN trunking using all the specified tags. If VLAN trunking with a single tag is needed, the optional attribute **trunk='yes'** can be added to the top-level **vlan** element.

23.18.9.16. Modifying virtual link state

This element sets the virtual network link state. Possible values for attribute **state** are **up** and **down**. If **down** is specified as the value, the interface behaves as the network cable is disconnected. Default behavior if this element is unspecified is **up**.

To specify the virtual link state configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet0'/>
    <link state='down'/>
  </interface>
<devices>
...
```

Figure 23.66. Modifying virtual link state

23.18.10. Input Devices

Input devices allow interaction with the graphical framebuffer in the guest virtual machine. When enabling the framebuffer, an input device is automatically provided. It may be possible to add additional devices explicitly, for example to provide a graphics tablet for absolute cursor movement.

To specify the input device configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <input type='mouse' bus='usb'/>
</devices>
...
```

Figure 23.67. Input devices

The **<input>** element has one mandatory attribute: **type**, which can be set to **mouse** or **tablet**. **tablet** provides absolute cursor movement, while **mouse** uses relative movement. The optional **bus** attribute can be used to refine the exact device type and can be set to **kvm** (paravirtualized), **ps2**, and **usb**.

The input element has an optional sub-element **<address>**, which can tie the device to a particular PCI slot, as documented above.

23.18.11. Hub Devices

A hub is a device that expands a single port into several so that there are more ports available to connect devices to a host physical machine system.

To specify the hub device configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <hub type='usb'/>
</devices>
...
```

Figure 23.68. Hub devices

The **hub** element has one mandatory attribute, **type**, which can only be set to **usb**. The **hub** element has an optional sub-element, **address**, with **type='usb'**, which can tie the device to a particular controller.

23.18.12. Graphical Framebuffers

A graphics device allows for graphical interaction with the guest virtual machine operating system. A guest virtual machine will typically have either a framebuffer or a text console configured to allow interaction with the user.

To specify the graphical framebuffer device configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <graphics type='sdl' display=':0.0'/>
  <graphics type='vnc' port='5904'>
    <listen type='address' address='1.2.3.4'/>
  </graphics>
  <graphics type='rdp' autoport='yes' multiUser='yes' />
  <graphics type='desktop' fullscreen='yes'/>
  <graphics type='spice'>
    <listen type='network' network='rednet'/>
  </graphics>
</devices>
...
```

Figure 23.69. Graphical framebuffers

The **graphics** element has a mandatory **type** attribute, which takes the value **sdl**, **vnc**, **rdp**, **desktop** or **spice**, as explained in the tables below:

Table 23.23. Graphical framebuffer main elements

Parameter	Description

Parameter	Description
sdl	This displays a window on the host physical machine desktop. It accepts the following optional arguments: • A **display** attribute for the display to use • An **xauth** attribute for the authentication identifier • An optional **fullscreen** attribute accepting values **yes** or **no**
vnc	Starts a VNC server. • The **port** attribute specifies the TCP port number (with **-1** as legacy syntax indicating that it should be auto-allocated). • The **autoport** attribute is the preferred syntax for indicating auto-allocation of the TCP port to use. • The **listen** attribute is an IP address for the server to listen on. • The **passwd** attribute provides a VNC password in clear text. • The **keymap** attribute specifies the keymap to use. It is possible to set a limit on the validity of the password be giving an **timestamp passwdValidTo='2010-04-09T15:51:00'** assumed to be in UTC. • The **connected** attribute allows control of connected client during password changes. VNC accepts the **keep** value only; note that it may not be supported by all hypervisors. • Rather than using listen/port, KVM supports a socket attribute for listening on a UNIX domain socket path.

Parameter	Description
`spice`	Starts a SPICE server. • The `port` attribute specifies the TCP port number (with -1 as legacy syntax indicating that it should be auto-allocated), while `tlsPort` gives an alternative secure port number. • The `autoport` attribute is the new preferred syntax for indicating auto-allocation of both port numbers. • The `listen` attribute is an IP address for the server to listen on. • The `passwd` attribute provides a SPICE password in clear text. • The `keymap` attribute specifies the keymap to use. It is possible to set a limit on the validity of the password be giving an `timestamp passwdValidTo='2010-04-09T15:51:00'` assumed to be in UTC. • The `connected` attribute allows control of a connected client during password changes. SPICE accepts `keep` to keep a client connected, `disconnect` to disconnect the client and `fail` to fail changing password. Note that this is not supported by all hypervisors. • The `defaultMode` attribute sets the default channel security policy; valid values are `secure`, `insecure` and the default `any` (which is `secure` if possible, but falls back to `insecure` rather than erroring out if no secure path is available).

When SPICE has both a normal and a TLS-secured TCP port configured, it may be desirable to restrict what channels can be run on each port. To do this, add one or more `channel` elements inside the main `graphics` element. Valid channel names include `main`, `display`, `inputs`, `cursor`, `playback`, `record`, `smartcard`, and `usbredir`.

To specify the SPICE configuration settings, use a mangement tool to make the following changes to the domain XML:

```
<graphics type='spice' port='-1' tlsPort='-1' autoport='yes'>
  <channel name='main' mode='secure'/>
  <channel name='record' mode='insecure'/>
  <image compression='auto_glz'/>
  <streaming mode='filter'/>
  <clipboard copypaste='no'/>
  <mouse mode='client'/>
</graphics>
```

Figure 23.70. Sample SPICE configuration

SPICE supports variable compression settings for audio, images and streaming. These settings are configured using the **compression** attribute in the following elements:

- **image** to set image compression (accepts **auto_glz**, **auto_lz**, **quic**, **glz**, **lz**, **off**)
- **jpeg** for JPEG compression for images over WAN (accepts **auto**, **never**, **always**)
- **zlib** for configuring WAN image compression (accepts **auto**, **never**, **always**) and **playback** for enabling audio stream compression (accepts **on** or **off**)

The **streaming** element sets streaming mode. The **mode** attribute can be set to **filter**, **all** or **off**.

In addition, copy and paste functionality (through the SPICE agent) is set by the **clipboard** element. It is enabled by default, and can be disabled by setting the **copypaste** property to **no**.

The **mouse** element sets mouse mode. The **mode** attribute can be set to **server** or **client**. If no mode is specified, the KVM default will be used (**client** mode).

Additional elements include:

Table 23.24. Additional graphical framebuffer elements

Parameter	Description
rdp	Starts an RDP server. • The **port** attribute specifies the TCP port number (with -1 as legacy syntax indicating that it should be auto-allocated). • The **autoport** attribute is the preferred syntax for indicating auto-allocation of the TCP port to use. • The **replaceUser** attribute is a boolean deciding whether multiple simultaneous connections to the virtual machine are permitted. • The **multiUser** attribute decides whether the existing connection must be dropped and a new connection must be established by the VRDP server, when a new client connects in single connection mode.

Parameter	Description
desktop	This value is currently reserved for VirtualBox domains. It displays a window on the host physical machine desktop, similarly to **sdl**, but uses the VirtualBox viewer. Just like **sdl**, it accepts the optional attributes **display** and **fullscreen**.
listen	Rather than inputting the address information used to set up the listening socket for graphics types **vnc** and **spice**, the **listen** attribute, a separate sub-element of **graphics**, can be specified (see the examples above). **listen** accepts the following attributes: • **type** - Set to either **address** or **network**. This tells whether this listen element is specifying the address to be used directly, or by naming a network (which will then be used to determine an appropriate address for listening). • **address** - This attribute will contain either an IP address or host name (which will be resolved to an IP address via a DNS query) to listen on. In the "live" XML of a running domain, this attribute will be set to the IP address used for listening, even if **type='network'**. • **network** - If **type='network'**, the network attribute will contain the name of a network in libvirt's list of configured networks. The named network configuration will be examined to determine an appropriate listen address. For example, if the network has an IPv4 address in its configuration (for example, if it has a forward type of route, NAT, or an isolated type), the first IPv4 address listed in the network's configuration will be used. If the network is describing a host physical machine bridge, the first IPv4 address associated with that bridge device will be used. If the network is describing one of the 'direct' (macvtap) modes, the first IPv4 address of the first forward dev will be used.

23.18.13. Video Devices

To specify the video device configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <video>
    <model type='vga' vram='8192' heads='1'>
      <acceleration accel3d='yes' accel2d='yes'/>
    </model>
  </video>
</devices>
...
```

Figure 23.71. Video devices

The **graphics** element has a mandatory **type** attribute which takes the value "sdl", "vnc", "rdp" or "desktop" as explained below:

Table 23.25. Graphical framebuffer elements

Parameter	Description
video	The **video** element is the container for describing video devices. For backwards compatibility, if no video is set but there is a **graphics** element in the domain XML, then libvirt will add a default **video** according to the guest virtual machine type. If "ram" or "vram" are not supplied, a default value is used.
model	This has a mandatory **type** attribute which takes the value **vga**, **cirrus**, **vmvga**, **kvm**, **vbox**, or **qxl** depending on the hypervisor features available. You can also provide the amount of video memory in kibibytes (blocks of 1024 bytes) using vram and the number of figure with heads.
acceleration	If acceleration is supported it should be enabled using the **accel3d** and **accel2d** attributes in the **acceleration** element.
address	The optional address sub-element can be used to tie the video device to a particular PCI slot.

23.18.14. Consoles, Serial, and Channel Devices

A character device provides a way to interact with the virtual machine. Paravirtualized consoles, serial ports, and channels are all classed as character devices and are represented using the same syntax.

To specify the consoles, channel and other device configuration settings, use a management tool to make the following changes to the domain XML:

```
...
<devices>
  <serial type='pty'>
    <source path='/dev/pts/3'/>
    <target port='0'/>
  </serial>
  <console type='pty'>
    <source path='/dev/pts/4'/>
    <target port='0'/>
  </console>
  <channel type='unix'>
    <source mode='bind' path='/tmp/guestfwd'/>
    <target type='guestfwd' address='10.0.2.1' port='4600'/>
  </channel>
</devices>
...
```

Figure 23.72. Consoles, serial, and channel devices

In each of these directives, the top-level element name (**serial**, **console**, **channel**) describes how the device is presented to the guest virtual machine. The guest virtual machine interface is configured by the **target** element. The interface presented to the host physical machine is given in the **type** attribute of the top-level element. The host physical machine interface is configured by the source element. The **source** element may contain an optional **seclabel** to override the way that labeling is done on the socket path. If this element is not present, the security label is inherited from the per-domain setting. Each character device element has an optional sub-element **address** which can tie the device to a particular controller or PCI slot.

NOTE

Parallel ports, as well as the **isa-parallel** device, are no longer supported.

23.18.15. Guest Virtual Machine Interfaces

A character device presents itself to the guest virtual machine as one of the following types.

To set the serial port, use a management tool to make the following change to the domain XML:

```
...
<devices>
  <serial type='pty'>
    <source path='/dev/pts/3'/>
    <target port='0'/>
  </serial>
</devices>
...
```

Figure 23.73. Guest virtual machine interface serial port

<target> can have a **port** attribute, which specifies the port number. Ports are numbered starting from 0. There are usually 0, 1 or 2 serial ports. There is also an optional **type** attribute, which has two choices for its value, **isa-serial** or **usb-serial**. If **type** is missing, **isa-serial** will be used by

default. For **usb-serial**, an optional sub-element **<address>** with **type='usb'** can tie the device to a particular controller, documented above.

The **<console>** element is used to represent interactive consoles. Depending on the type of guest virtual machine in use, the consoles might be paravirtualized devices, or they might be a clone of a serial device, according to the following rules:

- If no **targetType** attribute is set, then the default device **type** is according to the hypervisor's rules. The default **type** will be added when re-querying the XML fed into libvirt. For fully virtualized guest virtual machines, the default device type will usually be a serial port.
- If the **targetType** attribute is **serial**, and if no **<serial>** element exists, the console element will be copied to the **<serial>** element. If a **<serial>** element does already exist, the console element will be ignored.
- If the **targetType** attribute is not **serial**, it will be treated normally.
- Only the first **<console>** element may use a **targetType** of **serial**. Secondary consoles must all be paravirtualized.
- On s390, the console element may use a **targetType** of **sclp** or **sclplm** (line mode). SCLP is the native console type for s390. There is no controller associated to SCLP consoles.

In the example below, a virtio console device is exposed in the guest virtual machine as **/dev/hvc[0-7]** (for more information, see the Fedora project's virtio-serial page):

```
...
<devices>
  <console type='pty'>
    <source path='/dev/pts/4'/>
    <target port='0'/>
  </console>

  <!-- KVM virtio console -->
  <console type='pty'>
    <source path='/dev/pts/5'/>
    <target type='virtio' port='0'/>
  </console>
</devices>
...

...
<devices>
  <!-- KVM s390 sclp console -->
  <console type='pty'>
    <source path='/dev/pts/1'/>
    <target type='sclp' port='0'/>
  </console>
</devices>
...
```

Figure 23.74. Guest virtual machine interface - virtio console device

If the console is presented as a serial port, the **<target>** element has the same attributes as for a serial port. There is usually only one console.

23.18.16. Channel

This represents a private communication channel between the host physical machine and the guest virtual machine. It is manipulated by making changes to a guest virtual machine using a management tool to edit following section of the domain XML:

```
...
<devices>
  <channel type='unix'>
    <source mode='bind' path='/tmp/guestfwd'/>
    <target type='guestfwd' address='10.0.2.1' port='4600'/>
  </channel>

  <!-- KVM virtio channel -->
  <channel type='pty'>
    <target type='virtio' name='arbitrary.virtio.serial.port.name'/>
  </channel>
  <channel type='unix'>
    <source mode='bind' path='/var/lib/libvirt/kvm/f16x86_64.agent'/>
    <target type='virtio' name='org.kvm.guest_agent.0'/>
  </channel>
  <channel type='spicevmc'>
    <target type='virtio' name='com.redhat.spice.0'/>
  </channel>
</devices>
...
```

Figure 23.75. Channel

This can be implemented in a variety of ways. The specific type of **<channel>** is given in the **type** attribute of the **<target>** element. Different channel types have different target attributes as follows:

- **guestfwd** - Dictates that TCP traffic sent by the guest virtual machine to a given IP address and port is forwarded to the channel device on the host physical machine. The **target** element must have address and port attributes.
- **virtio** - paravirtualized virtio channel. **<channel>** is exposed in the guest virtual machine under **/dev/vport***, and if the optional element **name** is specified, **/dev/virtio-ports/$name** (for more information, see the Fedora project's virtio-serial page). The optional element **address** can tie the channel to a particular **type='virtio-serial'** controller, documented above. With KVM, if name is "org.kvm.guest_agent.0", then libvirt can interact with a guest agent installed in the guest virtual machine, for actions such as guest virtual machine shutdown or file system quiescing.
- **spicevmc** - Paravirtualized SPICE channel. The domain must also have a SPICE server as a graphics device, at which point the host physical machine piggy-backs messages across the main channel. The **target** element must be present, with attribute **type='virtio';** an optional attribute **name** controls how the guest virtual machine will have access to the channel, and defaults to **name='com.redhat.spice.0'**. The optional **<address>** element can tie the channel to a particular **type='virtio-serial'** controller.

23.18.17. Host Physical Machine Interface

A character device presents itself to the host physical machine as one of the following types:

Table 23.26. Character device elements

Parameter	Description	XML snippet
Domain logfile	Disables all input on the character device, and sends output into the virtual machine's logfile.	`<devices> <console type='stdio'> <target port='1'/> </console> </devices>`
Device logfile	A file is opened and all data sent to the character device is written to the file. Note that the destination directory must have the **virt_log_t** SELinux label for a guest with this setting to start successfully.	`<devices> <serial type="file"> <source path="/var/log/vm/vm-serial.log"/> <target port="1"/> </serial> </devices>`
Virtual console	Connects the character device to the graphical framebuffer in a virtual console. This is typically accessed using a special hotkey sequence such as "ctrl+alt+3".	`<devices> <serial type='vc'> <target port="1"/> </serial> </devices>`
Null device	Connects the character device to the void. No data is ever provided to the input. All data written is discarded.	`<devices> <serial type='null'> <target port="1"/> </serial> </devices>`

Parameter	Description	XML snippet
Pseudo TTY	A Pseudo TTY is allocated using **/dev/ptmx**. A suitable client such as **virsh console** can connect to interact with the serial port locally.	`<devices> <serial type="pty"> <source path="/dev/pts/3"/> <target port="1"/> </serial> </devices>`
NB Special case	NB special case if **<console type='pty'>**, then the TTY path is also duplicated as an attribute **tty='/dev/pts/3'** on the top level **<console>** tag. This provides compat with existing syntax for **<console>** tags.	
Host physical machine device proxy	The character device is passed through to the underlying physical character device. The device types must match, for example the emulated serial port should only be connected to a host physical machine serial port - do not connect a serial port to a parallel port.	`<devices> <serial type="dev"> <source path="/dev/ttyS0"/> <target port="1"/> </serial> </devices>`
Named pipe	The character device writes output to a named pipe. See the pipe(7) man page for more info.	`<devices> <serial type="pipe"> <source path="/tmp/mypipe"/> <target port="1"/> </serial> </devices>`
TCP client-server	The character device acts as a TCP client connecting to a remote server.	`<devices> <serial type="tcp"> <source mode="connect" host="0.0.0.0" service="2445"/>`

Parameter	Description	XML snippet
		<protocol type="raw"/> <target port="1"/> </serial> </devices>

Or as a TCP server waiting for a client connection.

```
<devices>
   <serial type="tcp">
      <source mode="bind" host="127.0.0.1" service="2445"/>
      <protocol type="raw"/>
      <target port="1"/>
   </serial>
</devices>
```

Alternatively you can use telnet instead of raw TCP. In addition, you can also use telnets (secure telnet) and tls.

```
<devices>
   <serial type="tcp">
      <source mode="connect" host="0.0.0.0" service="2445"/>
      <protocol type="telnet"/>
      <target port="1"/>
   </serial>
      <serial type="tcp">
      <source mode="bind" host="127.0.0.1" service="2445"/>
      <protocol type="telnet"/>
      <target port="1"/>
   </serial>
</devices>
```

Parameter	Description	XML snippet
UDP network console	The character device acts as a UDP netconsole service, sending and receiving packets. This is a lossy service.	`<devices> <serial type="udp"> <source mode="bind" host="0.0.0.0" service="2445"/> <source mode="connect" host="0.0.0.0" service="2445"/> <target port="1"/> </serial> </devices>`
UNIX domain socket client-server	The character device acts as a UNIX domain socket server, accepting connections from local clients.	`<devices> <serial type="unix"> <source mode="bind" path="/tmp/foo"/> <target port="1"/> </serial> </devices>`

23.18.18. Sound Devices

A virtual sound card can be attached to the host physical machine using the **sound** element.

```
...
<devices>
  <sound model='ac97'/>
</devices>
...
```

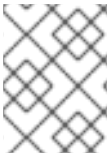
Figure 23.76. Virtual sound card

The **sound** element has one mandatory attribute, **model**, which specifies what real sound device is emulated. Valid values are specific to the underlying hypervisor, though typical choices are **'sb16'**, **'ac97'**, and **'ich6'**. In addition, a **sound** element with **'ich6'** model set can have optional **codec** sub-elements to attach various audio codecs to the audio device. If not specified, a default codec will be attached to allow playback and recording. Valid values are **'duplex'** (advertises a line-in and a line-out) and **'micro'** (advertises a speaker and a microphone).

```
...
<devices>
  <sound model='ich6'>
    <codec type='micro'/>
  <sound/>
</devices>
...
```

Figure 23.77. Sound Devices

Each sound element has an optional sub-element **<address>** which can tie the device to a particular PCI slot, documented above.



NOTE

The es1370 sound device is no longer supported in Red Hat Enterprise Linux 7. Use ac97 instead.

23.18.19. Watchdog Device

A virtual hardware watchdog device can be added to the guest virtual machine using the **<watchdog>** element. The watchdog device requires an additional driver and management daemon in the guest virtual machine. Currently there is no support notification when the watchdog fires.

```
...
<devices>
  <watchdog model='i6300esb'/>
</devices>
...

...
<devices>
  <watchdog model='i6300esb' action='poweroff'/>
</devices>
...
```

Figure 23.78. Watchdog Device

The following attributes are declared in this XML:

- **model** - The required **model** attribute specifies what real watchdog device is emulated. Valid values are specific to the underlying hypervisor.
- The **model** attribute may take the following values:
  - **i6300esb** — the recommended device, emulating a PCI Intel 6300ESB
  - **ib700** — emulates an ISA iBase IB700
- **action** - The optional **action** attribute describes what action to take when the watchdog expires. Valid values are specific to the underlying hypervisor. The **action** attribute can have the following values:

- **reset** — default setting, forcefully resets the guest virtual machine
- **shutdown** — gracefully shuts down the guest virtual machine (not recommended)
- **poweroff** — forcefully powers off the guest virtual machine
- **pause** — pauses the guest virtual machine
- **none** — does nothing
- **dump** — automatically dumps the guest virtual machine.

Note that the 'shutdown' action requires that the guest virtual machine is responsive to ACPI signals. In the sort of situations where the watchdog has expired, guest virtual machines are usually unable to respond to ACPI signals. Therefore, using 'shutdown' is not recommended. In addition, the directory to save dump files can be configured by auto_dump_path in file /etc/libvirt/kvm.conf.

23.18.20. Setting a Panic Device

Red Hat Enterprise Linux 7 hypervisor is capable of detecting Linux guest virtual machine kernel panics, using the **pvpanic** mechanism. When invoked, **pvpanic** sends a message to the **libvirtd** daemon, which initiates a preconfigured reaction.

To enable the **pvpanic** device, do the following:

- Add or uncomment the following line in the **/etc/libvirt/qemu.conf** file on the host machine.

```
auto_dump_path = "/var/lib/libvirt/qemu/dump"
```

- Run the **virsh edit** command to edit domain XML file of the specified guest, and add the **panic** into the **devices** parent element.

```
<devices>
  <panic>
    <address type='isa' iobase='0x505'/>
  </panic>
</devices>
```

The **<address>** element specifies the address of panic. The default ioport is 0x505. In most cases, specifying an address is not needed.

The way in which **libvirtd** reacts to the crash is determined by the **<on_crash>** element of the domain XML. The possible actions are as follows:

- **coredump-destroy** - Captures the guest virtual machine's core dump and shuts the guest down.
- **coredump-restart** - Captures the guest virtual machine's core dump and restarts the guest.
- **preserve** - Halts the guest virtual machine to await further action.

NOTE

If the **kdump** service is enabled, it takes precedence over the **<on_crash>** setting, and the selected **<on_crash>** action is not performed.

For more information on **pvpanic**, see the related Knowledgebase article.

23.18.21. Memory Balloon Device

The balloon device can designate a part of a virtual machine's RAM as not being used (a process known as *inflating* the balloon), so that the memory can be freed for the host, or for other virtual machines on that host, to use. When the virtual machine needs the memory again, the balloon can be *deflated* and the host can distribute the RAM back to the virtual machine.

The size of the memory balloon is determined by the difference between the **<currentMemory>** and **<memory>** settings. For example, if **<memory>** is set to 2 GiB and **<currentMemory>** to 1 GiB, the balloon contains 1 GiB. If manual configuration is necessary, the **<currentMemory>** value can be set by using the **virsh setmem** command and the **<memory>** value can be set by using the **virsh setmaxmem** command.

WARNING

If modifying the **<currentMemory>** value, make sure to leave sufficient memory for the guest OS to work properly. If the set value is too low, the guest may become unstable.

A virtual memory balloon device is automatically added to all KVM guest virtual machines. In the XML configuration, this is represented by the **<memballoon>** element. Memory ballooning is managed by the **libvirt** service, and will be automatically added when appropriate. Therefore, it is not necessary to explicitly add this element in the guest virtual machine XML unless a specific PCI slot needs to be assigned. Note that if the **<memballoon>** device needs to be explicitly disabled, **model='none'** can be be used for this purpose.

The following example a shows a memballoon device automatically added by **libvirt**:

```
...
<devices>
  <memballoon model='virtio'/>
</devices>
...
```

Figure 23.79. Memory balloon device

The following example shows a device that has been added manually with static PCI slot 2 requested:

```
  ...
  <devices>
    <memballoon model='virtio'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x02'
function='0x0'/>
    </memballoon>
  </devices>
...
```

Figure 23.80. Memory balloon device added manually

The required **model** attribute specifies what type of balloon device is provided. Valid values are specific to the virtualization platform; in the KVM hypervisor, **'virtio'** is the default setting.

23.19. STORAGE POOLS

Although all storage pool back-ends share the same public APIs and XML format, they have varying levels of capabilities. Some may allow creation of volumes, others may only allow use of pre-existing volumes. Some may have constraints on volume size, or placement.

The top level element for a storage pool document is **<pool>**. It has a single attribute **type**, which can take the following values: **dir, fs, netfs, disk, iscsi, logical, scsi, mpath, rbd, sheepdog**, or **gluster**.

23.19.1. Providing Metadata for the Storage Pool

The following XML example, shows the metadata tags that can be added to a storage pool. In this example, the pool is an iSCSI storage pool.

```
    <pool type="iscsi">
       <name>virtimages</name>
       <uuid>3e3fce45-4f53-4fa7-bb32-11f34168b82b</uuid>
       <allocation>10000000</allocation>
       <capacity>50000000</capacity>
       <available>40000000</available>
       ...
    </pool>
```

Figure 23.81. General metadata tags

The elements that are used in this example are explained in the Table 23.27, "**virt-sysprep** commands".

Table 23.27. virt-sysprep commands

Element	Description

Element	Description
<name>	Provides a name for the storage pool which must be unique to the host physical machine. This is mandatory when defining a storage pool.
<uuid>	Provides an identifier for the storage pool which must be globally unique. Although supplying the UUID is optional, if the UUID is not provided at the time the storage pool is created, a UUID will be automatically generated.
<allocation>	Provides the total storage allocation for the storage pool. This may be larger than the sum of the total allocation across all storage volumes due to the metadata overhead. This value is expressed in bytes. This element is read-only and the value should not be changed.
<capacity>	Provides the total storage capacity for the pool. Due to underlying device constraints, it may not be possible to use the full capacity for storage volumes. This value is in bytes. This element is read-only and the value should not be changed.
<available>	Provides the free space available for allocating new storage volumes in the storage pool. Due to underlying device constraints, it may not be possible to allocate the all of the free space to a single storage volume. This value is in bytes. This element is read-only and the value should not be changed.

23.19.2. Source Elements

Within the **<pool>** element there can be a single **<source>** element defined (only one). The child elements of **<source>** depend on the storage pool type. Some examples of the XML that can be used are as follows:

```
      ...
    <source>
      <host name="iscsi.example.com"/>
      <device path="demo-target"/>
      <auth type='chap' username='myname'>
        <secret type='iscsi' usage='mycluster_myname'/>
      </auth>
      <vendor name="Acme"/>
      <product name="model"/>
    </source>
    ...
```

Figure 23.82. Source element option 1

```
     ...
    <source>
      <adapter type='fc_host' parent='scsi_host5'
wwnn='20000000c9831b4b' wwpn='10000000c9831b4b'/>
    </source>
    ...
```

Figure 23.83. Source element option 2

The child elements that are accepted by **<source>** are explained in Table 23.28, “Source child elements commands”.

Table 23.28. Source child elements commands

Element	Description
<device>	Provides the source for storage pools backed by host physical machine devices (based on **<pool type=>** (as shown in Section 23.19, “Storage Pools”)). May be repeated multiple times depending on back-end driver. Contains a single attribute **path** which is the fully qualified path to the block device node.
<dir>	Provides the source for storage pools backed by directories (**<pool type='dir'>**), or optionally to select a subdirectory within a storage pool that is based on a filesystem (**<pool type='gluster'>**). This element may only occur once per (**<pool>**). This element accepts a single attribute (**<path>**) which is the full path to the backing directory.

Element	Description
<adapter>	Provides the source for storage pools backed by SCSI adapters (**<pool type='scsi'>**). This element may only occur once per (**<pool>**). Attribute name is the SCSI adapter name (ex. "scsi_host1". Although "host1" is still supported for backwards compatibility, it is not recommended. Attribute **type** specifies the adapter type. Valid values are **'fc_host'| 'scsi_host'**. If omitted and the **name** attribute is specified, then it defaults to **type='scsi_host'**. To keep backwards compatibility, the attribute **type** is optional for the **type='scsi_host'** adapter, but mandatory for the **type='fc_host'** adapter. Attributes **wwnn** (Word Wide Node Name) and **wwpn** (Word Wide Port Name) are used by the **type='fc_host'** adapter to uniquely identify the device in the Fibre Channel storage fabric (the device can be either a HBA or vHBA). Both **wwnn** and **wwpn** should be specified. For instructions on how to get wwnn/wwpn of a (v)HBA, see Section 20.27.12, “Collect Device Configuration Settings”. The optional attribute **parent** specifies the parent device for the **type='fc_host'** adapter.
<host>	Provides the source for storage pools backed by storage from a remote server (**type='netfs'|'iscsi'|'rbd'|'sheepdog'|'gluster'**). This element should be used in combination with a **<directory>** or **<device>** element. Contains an attribute **name** which is the host name or IP address of the server. May optionally contain a **port** attribute for the protocol specific port number.

Element	Description
<auth>	If present, the **<auth>** element provides the authentication credentials needed to access the source by the setting of the **type** attribute (pool **type='iscsi'|'rbd'**). The **type** must be either **type='chap'** or **type='ceph'**. Use "ceph" for Ceph RBD (Rados Block Device) network sources and use "iscsi" for CHAP (Challenge-Handshake Authentication Protocol) iSCSI targets. Additionally a mandatory attribute **username** identifies the user name to use during authentication as well as a sub-element secret with a mandatory attribute type, to tie back to a libvirt secret object that holds the actual password or other credentials. The domain XML intentionally does not expose the password, only the reference to the object that manages the password. The secret element requires either a uuid attribute with the UUID of the secret object or a usage attribute matching the key that was specified in the secret object.
<name>	Provides the source for storage pools backed by a storage device from a named element **<type>** which can take the values: (**type='logical'|'rbd'|'sheepdog','gluster'**).
<format>	Provides information about the format of the storage pool **<type>** which can take the following values: **type='logical'|'disk'|'fs'|'netfs'**). Note that this value is back-end specific. This is typically used to indicate a filesystem type, or a network filesystem type, or a partition table type, or an LVM metadata type. As all drivers are required to have a default value for this, the element is optional.
<vendor>	Provides optional information about the vendor of the storage device. This contains a single attribute **<name>** whose value is back-end specific.
<product>	Provides optional information about the product name of the storage device. This contains a single attribute **<name>** whose value is back-end specific.

23.19.3. Creating Target Elements

A single **<target>** element is contained within the top level **<pool>** element for the following types: (**type='dir'|'fs'|'netfs'|'logical'|'disk'|'iscsi'|'scsi'|'mpath'**). This tag is used to describe the mapping of the storage pool into the host filesystem. It can contain the following child elements:

```
<pool>
   ...
   <target>
     <path>/dev/disk/by-path</path>
     <permissions>
       <owner>107</owner>
       <group>107</group>
       <mode>0744</mode>
       <label>virt_image_t</label>
     </permissions>
     <timestamps>
       <atime>1341933637.273190990</atime>
       <mtime>1341930622.047245868</mtime>
       <ctime>1341930622.047245868</ctime>
     </timestamps>
     <encryption type='...'>
       ...
     </encryption>
   </target>
 </pool>
```

Figure 23.84. Target elements XML example

The table (Table 23.29, “Target child elements”) explains the child elements that are valid for the parent **<target>** element:

Table 23.29. Target child elements

Element	Description
<path>	Provides the location at which the storage pool will be mapped into the local filesystem namespace. For a filesystem or directory-based storage pool it will be the name of the directory in which storage volumes will be created. For device-based storage pools it will be the name of the directory in which the device's nodes exist. For the latter, **/dev/**may seem like the logical choice, however, the device's nodes there are not guaranteed to be stable across reboots, since they are allocated on demand. It is preferable to use a stable location such as one of the **/dev/disk/by-{path,id,uuid,label}** locations.

Element	Description
<permissions>	This is currently only useful for directory- or filesystem-based storage pools, which are mapped as a directory into the local filesystem namespace. It provides information about the permissions to use for the final directory when the storage pool is built. The **<mode>** element contains the octal permission set. The **<owner>** element contains the numeric user ID. The **<group>** element contains the numeric group ID. The **<label>** element contains the MAC (for example, SELinux) label string.
<timestamps>	Provides timing information about the storage volume. Up to four sub-elements are present, where **timestamps='atime'|'btime|'ctime'|'mtime'** holds the access, birth, change, and modification time of the storage volume, where known. The used time format is **<seconds>.<nanoseconds>** since the beginning of the epoch (1 Jan 1970). If nanosecond resolution is 0 or otherwise unsupported by the host operating system or filesystem, then the nanoseconds part is omitted. This is a read-only attribute and is ignored when creating a storage volume.
<encryption>	If present, specifies how the storage volume is encrypted. For more information, see libvirt upstream pages.

23.19.4. Setting Device Extents

If a storage pool exposes information about its underlying placement or allocation scheme, the **<device>** element within the **<source>** element may contain information about its available extents. Some storage pools have a constraint that a storage volume must be allocated entirely within a single constraint (such as disk partition pools). Thus, the extent information allows an application to determine the maximum possible size for a new storage volume.

For storage pools supporting extent information, within each **<device>** element there will be zero or more **<freeExtent>** elements. Each of these elements contains two attributes, **<start>** and **<end>** which provide the boundaries of the extent on the device, measured in bytes.

23.20. STORAGE VOLUMES

A storage volume will generally be either a file or a device node; since 1.2.0, an optional output-only attribute type lists the actual type (file, block, dir, network, or netdir),

23.20.1. General Metadata

The top section of the **<volume>** element contains information known as metadata as shown in this XML example:

```
...
<volume type='file'>
  <name>sparse.img</name>
  <key>/var/lib/libvirt/images/sparse.img</key>
  <allocation>0</allocation>
  <capacity unit="T">1</capacity>
  ...
</volume>
```

Figure 23.85. General metadata for storage volumes

The table (Table 23.30, “Volume child elements”) explains the child elements that are valid for the parent **<volume>** element:

Table 23.30. Volume child elements

Element	Description
<name>	Provides a name for the storage volume which is unique to the storage pool. This is mandatory when defining a storage volume.
<key>	Provides an identifier for the storage volume which identifies a single storage volume. In some cases it is possible to have two distinct keys identifying a single storage volume. This field cannot be set when creating a storage volume as it is always generated.
<allocation>	Provides the total storage allocation for the storage volume. This may be smaller than the logical capacity if the storage volume is sparsely allocated. It may also be larger than the logical capacity if the storage volume has substantial metadata overhead. This value is in bytes. If omitted when creating a storage volume, the storage volume will be fully allocated at time of creation. If set to a value smaller than the capacity, the storage pool has the option of deciding to sparsely allocate a storage volume or not. Different types of storage pools may treat sparse storage volumes differently. For example, a logical pool will not automatically expand a storage volume's allocation when it gets full; the user is responsible for configuring it or configuring **dmeventd** to do so automatically. By default, this is specified in bytes. See Note

Element	Description
<capacity>	Provides the logical capacity for the storage volume. This value is in bytes by default, but a **<unit>** attribute can be specified with the same semantics as for **<allocation>** described in Note. This is compulsory when creating a storage volume.
<source>	Provides information about the underlying storage allocation of the storage volume. This may not be available for some storage pool types.
<target>	Provides information about the representation of the storage volume on the local host physical machine.

NOTE

When necessary, an optional attribute **unit** can be specified to adjust the passed value. This attribute can be used with the elements **<allocation>** and **<capacity>**. Accepted values for the attribute **unit** include:

- **B** or **bytes** for bytes
- **KB** for kilobytes
- **K** or **KiB** for kibibytes
- **MB** for megabytes
- **M** or **MiB** for mebibytes
- **GB** for gigabytes
- **G** or **GiB** for gibibytes
- **TB** for terabytes
- **T** or **TiB** for tebibytes
- **PB** for petabytes
- **P** or **PiB** for pebibytes
- **EB** for exabytes
- **E** or **EiB** for exbibytes

23.20.2. Setting Target Elements

The **<target>** element can be placed in the **<volume>** top level element. It is used to describe the mapping that is done on the storage volume into the host physical machine filesystem. This element can take the following child elements:

```
<target>
   <path>/var/lib/libvirt/images/sparse.img</path>
   <format type='qcow2'/>
   <permissions>
     <owner>107</owner>
     <group>107</group>
     <mode>0744</mode>
     <label>virt_image_t</label>
   </permissions>
   <compat>1.1</compat>
   <features>
     <lazy_refcounts/>
   </features>
 </target>
```

Figure 23.86. Target child elements

The specific child elements for **<target>** are explained in Table 23.31, “Target child elements”:

Table 23.31. Target child elements

Element	Description
<path>	Provides the location at which the storage volume can be accessed on the local filesystem, as an absolute path. This is a read-only attribute, and should not be specified when creating a volume.
<format>	Provides information about the pool specific volume format. For disk-based storage pools, it will provide the partition type. For filesystem or directory-based storage pools, it will provide the file format type, (such as cow, qcow, vmdk, raw). If omitted when creating a storage volume, the storage pool's default format will be used. The actual format is specified by the **type** attribute. See the sections on the specific storage pools in Section 13.2, “Using Storage Pools” for the list of valid values.
<permissions>	Provides information about the default permissions to use when creating storage volumes. This is currently only useful for directory or filesystem-based storage pools, where the storage volumes allocated are simple files. For storage pools where the storage volumes are device nodes, the hot-plug scripts determine permissions. It contains four child elements. The **<mode>** element contains the octal permission set. The **<owner>** element contains the numeric user ID. The **<group>** element contains the numeric group ID. The **<label>** element contains the MAC (for example, SELinux) label string.

Element	Description
<compat>	Specify compatibility level. So far, this is only used for **<type='qcow2'>** volumes. Valid values are **<compat>**0.10**</compat>** for qcow2 (version 2) and **<compat>**1.1**</compat>** for qcow2 (version 3) so far for specifying the QEMU version the images should be compatible with. If the **<feature>** element is present, **<compat>**1.1**</compat>** is used. If omitted, qemu-img default is used.
<features>	Format-specific features. Presently is only used with **<format type='qcow2'/>** (version 3). Valid sub-elements include **<lazy_refcounts/>**. This reduces the amount of metadata writes and flushes, and therefore improves initial write performance. This improvement is seen especially for writethrough cache modes, at the cost of having to repair the image after a crash, and allows delayed reference counter updates. It is recommended to use this feature with qcow2 (version 3), as it is faster when this is implemented.

23.20.3. Setting Backing Store Elements

A single **<backingStore>** element is contained within the top level **<volume>** element. This tag is used to describe the optional copy-on-write backing store for the storage volume. It can contain the following child elements:

```
<backingStore>
  <path>/var/lib/libvirt/images/master.img</path>
  <format type='raw'/>
  <permissions>
    <owner>107</owner>
    <group>107</group>
    <mode>0744</mode>
    <label>virt_image_t</label>
  </permissions>
</backingStore>
```

Figure 23.87. Backing store child elements

Table 23.32. Backing store child elements

Element	Description

Element	Description
<path>	Provides the location at which the backing store can be accessed on the local filesystem, as an absolute path. If omitted, there is no backing store for this storage volume.
<format>	Provides information about the pool specific backing store format. For disk-based storage pools it will provide the partition type. For filesystem or directory-based storage pools it will provide the file format type (such as cow, qcow, vmdk, raw). The actual format is specified via the **<type>** attribute. Consult the pool-specific docs for the list of valid values. Most file formats require a backing store of the same format, however, the qcow2 format allows a different backing store format.
<permissions>	Provides information about the permissions of the backing file. It contains four child elements. The **<owner>** element contains the numeric user ID. The **<group>** element contains the numeric group ID. The **<label>** element contains the MAC (for example, SELinux) label string.

23.21. SECURITY LABEL

The **<seclabel>** element allows control over the operation of the security drivers. There are three basic modes of operation, **'dynamic'** where libvirt automatically generates a unique security label, **'static'** where the application/administrator chooses the labels, or **'none'** where confinement is disabled. With dynamic label generation, libvirt will always automatically relabel any resources associated with the virtual machine. With static label assignment, by default, the administrator or application must ensure labels are set correctly on any resources, however, automatic relabeling can be enabled if needed.

If more than one security driver is used by libvirt, multiple seclabel tags can be used, one for each driver and the security driver referenced by each tag can be defined using the attribute **model**. Valid input XML configurations for the top-level security label are:

```
<seclabel type='dynamic' model='selinux'/>

<seclabel type='dynamic' model='selinux'>
  <baselabel>system_u:system_r:my_svirt_t:s0</baselabel>
</seclabel>

<seclabel type='static' model='selinux' relabel='no'>
  <label>system_u:system_r:svirt_t:s0:c392,c662</label>
</seclabel>

<seclabel type='static' model='selinux' relabel='yes'>
  <label>system_u:system_r:svirt_t:s0:c392,c662</label>
</seclabel>

<seclabel type='none'/>
```

Figure 23.88. Security label

If no **'type'** attribute is provided in the input XML, then the security driver default setting will be used, which may be either **'none'** or **'dynamic'**. If a **<baselabel>** is set but no **'type'** is set, then the type is presumed to be **'dynamic'**. When viewing the XML for a running guest virtual machine with automatic resource relabeling active, an additional XML element, **imagelabel**, will be included. This is an output-only element, so will be ignored in user supplied XML documents.

The following elements can be manipulated with the following values:

- **type** - Either **static**, **dynamic** or **none** to determine whether libvirt automatically generates a unique security label or not.
- **model** - A valid security model name, matching the currently activated security model.
- **relabel** - Either **yes** or **no**. This must always be **yes** if dynamic label assignment is used. With static label assignment it will default to **no**.
- **<label>** - If static labeling is used, this must specify the full security label to assign to the virtual domain. The format of the content depends on the security driver in use:
  - **SELinux**: a SELinux context.
  - **AppArmor**: an AppArmor profile.
  - **DAC**: owner and group separated by colon. They can be defined both as user/group names or UID/GID. The driver will first try to parse these values as names, but a leading plus sign can used to force the driver to parse them as UID or GID.
- **<baselabel>** - If dynamic labeling is used, this can optionally be used to specify the base security label. The format of the content depends on the security driver in use.
- **<imagelabel>** - This is an output only element, which shows the security label used on resources associated with the virtual domain. The format of the content depends on the security driver in use. When relabeling is in effect, it is also possible to fine-tune the labeling done for specific source file names, by either disabling the labeling (useful if the file exists on NFS or other file system that lacks security labeling) or requesting an alternate label (useful when a

management application creates a special label to allow sharing of some, but not all, resources between domains). When a seclabel element is attached to a specific path rather than the top-level domain assignment, only the attribute relabel or the sub-element label are supported.

23.22. A SAMPLE CONFIGURATION FILE

KVM hardware accelerated guest virtual machine on AMD64 and Intel 64:

```
<domain type='kvm'>
  <name>demo2</name>
  <uuid>4dea24b3-1d52-d8f3-2516-782e98a23fa0</uuid>
  <memory>131072</memory>
  <vcpu>1</vcpu>
  <os>
    <type arch="x86_64">hvm</type>
  </os>
  <clock sync="localtime"/>
  <devices>
    <emulator>/usr/libexec/qemu-kvm</emulator>
    <disk type='file' device='disk'>
      <source file='/var/lib/libvirt/images/demo2.img'/>
      <target dev='hda'/>
    </disk>
    <interface type='network'>
      <source network='default'/>
      <mac address='24:42:53:21:52:45'/>
    </interface>
    <graphics type='vnc' port='-1' keymap='de'/>
  </devices>
</domain>
```

Figure 23.89. Example domain XML configuration

PART III. APPENDICES

APPENDIX A. TROUBLESHOOTING

This chapter covers common problems and solutions for Red Hat Enterprise Linux 7 virtualization issues.

Read this chapter to develop an understanding of some of the common problems associated with virtualization technologies. It is recommended that you experiment and test virtualization on Red Hat Enterprise Linux 7 to develop your troubleshooting skills.

If you cannot find the answer in this document, there may be an answer online from the virtualization community. See Section D.1, “Online Resources” for a list of Linux virtualization websites.

A.1. DEBUGGING AND TROUBLESHOOTING TOOLS

This section summarizes the system administrator applications, the networking utilities, and debugging tools. You can use these standard system administration tools and logs to assist with troubleshooting:

- **kvm_stat** - Retrieves KVM runtime statistics. For more information, see Section A.4, “kvm_stat”.
- **ftrace** - Traces kernel events. For more information, see the *Red Hat Enterprise Linux Developer Guide*.
- **vmstat** - Displays virtual memory statistics. For more information, use the **man vmstat** command.
- **iostat** - Provides I/O load statistics. For more information, see the Red Hat Enterprise Linux Performance Tuning Guide
- **lsof** - Lists open files. For more information, use the **man lsof** command.
- **systemtap** - A scripting utility for monitoring the operating system. For more information, see the Red Hat Enterprise Linux Developer Guide.
- **crash** - Analyzes kernel crash dump data or a live system. For more information, see the Red Hat Enterprise Linux Kernel Crash Dump Guide.
- **sysrq** - A key combination that the kernel responds to even if the console is unresponsive. For more information, see the Red Hat Knowledge Base.

These networking utilities can assist with troubleshooting virtualization networking problems:

- **ip addr**, **ip route**, and **ip monitor**
- **tcpdump** - diagnoses packet traffic on a network. This command is useful for finding network abnormalities and problems with network authentication. There is a graphical version of **tcpdump**, named **wireshark**.
- **brctl** - A networking utility that inspects and configures the Ethernet bridge configuration in the Linux kernel. For example:

```
$ brctl show
bridge-name    bridge-id          STP  enabled  interfaces
-----------------------------------------------------------------------
-------------
virtbr0             8000.feffffff       yes        eth0
```

```
$ brctl showmacs virtbr0
port-no           mac-addr                  local?       aging
timer
1                 fe:ff:ff:ff:ff:           yes            0.00
2                 fe:ff:ff:fe:ff:           yes            0.00
$ brctl showstp virtbr0
virtbr0
bridge-id              8000.fefffffffff
designated-root        8000.fefffffffff
root-port              0                   path-cost              0
max-age                20.00               bridge-max-age
20.00
hello-time             2.00                bridge-hello-time
2.00
forward-delay          0.00                bridge-forward-delay
0.00
aging-time            300.01
hello-timer            1.43                tcn-timer
0.00
topology-change-timer  0.00                gc-timer
0.02
```

Listed below are some other useful commands for troubleshooting virtualization:

- **strace** is a command which traces system calls and events received and used by another process.
- **vncviewer** connects to a VNC server running on your server or a virtual machine. Install **vncviewer** using the **`yum install tigervnc`** command.
- **vncserver** starts a remote desktop on your server. Gives you the ability to run graphical user interfaces, such as virt-manager, using a remote session. Install **vncserver** using the **`yum install tigervnc-server`** command.

In addition to all the commands listed above, examining virtualization logs can be helpful. For more information, see Section A.6, “Virtualization Logs”.

A.2. CREATING DUMP FILES

You can request a dump of the core of a guest virtual machine to a file so that errors in the virtual machine can be diagnosed, for example by the **`crash`** utility.

WARNING

In Red Hat Enterprise Linux 7.5 and later, the Kernel Address Space Randomization (KASLR) feature prevents guest dump files from being readable by **crash**. To fix this, add the **<vmcoreinfo/>** element to the **<features>** section of the XML configuration files of your guests.

Note, however, that migrating guests with **<vmcoreinfo/>** fails if the destination host is using an OS that does not support **<vmcoreinfo/>**. These include Red Hat Enterprise Linux 7.4 and earlier, as well as Red Hat Enterprise Linux 6.9 and earlier.

A.2.1. Creating virsh Dump Files

Executing the **virsh dump** command sends a request to dump the core of a guest virtual machine to a file so errors in the virtual machine can be diagnosed. Running this command may require you to manually ensure proper permissions on file and path specified by the argument **corefilepath**. The **virsh dump** command is similar to a core dump(or the **crash** utility).

For further information, see Creating a Dump File of a Guest Virtual Machine's Core.

A.2.2. Saving a Core Dump Using a Python Script

The **dump-guest-memory.py** python script implements a GNU Debugger (GDB) extension that extracts and saves a guest virtual machine's memory from the core dump after the **qemu-kvm** process crashes on a host. If the host-side QEMU process crash is related to guest actions, investigating the guest state at the time of the QEMU process crash could be useful.

The python script implements a GDB extension. This is a new command for the GDB. After opening the core dump file of the original (crashed) QEMU process with GDB, the python script can be loaded into GDB. The new command can then be executed from the GDB prompt. This extracts a guest memory dump from the QEMU core dumpto a new local file.

To use the **dump-guest-memory.py** python script:

1. Install the qemu-kvm-debuginfo package on the system.
2. Launch GDB, opening the core dump file saved for the crashed **/usr/libexec/qemu-kvm** binary. The debug symbols load automatically.
3. Load the new command in GDB:

```
# source /usr/share/qemu-kvm/dump-guest-memory.py
```

NOTE

After loading the python script, the built-in GDB **help** command can provide detailed information about the **dump-guest-memory** extension.

4. Run the command in GDB. For example:

```
# dump-guest-memory /home/user/extracted-vmcore X86_64
```

5. Open **/home/user/extracted-vmcore** with the **crash** utility for guest kernel analysis.

For more information about extracting guest virtual machine cores from QEMU core files for use with the **crash** utility, see *How to extract ELF cores from 'gcore' generated qemu core files for use with the 'crash' utility*.

A.3. CAPTURING TRACE DATA ON A CONSTANT BASIS USING THE SYSTEMTAP FLIGHT RECORDER

You can capture QEMU trace data all the time using a systemtap initscript provided in the qemu-kvm package. This package uses SystemTap's flight recorder mode to trace all running guest virtual machines and to save the results to a fixed-size buffer on the host. Old trace entries are overwritten by new entries when the buffer is filled.

Procedure A.1. Configuring and running systemtap

1. **Install the package**
   Install the systemtap-initscript package by running the following command:

   ```
   # yum install systemtap-initscript
   ```

2. **Copy the configuration file**
   Copy the systemtap scripts and the configuration files to the systemtap directory by running the following commands:

   ```
   # cp /usr/share/qemu-kvm/systemtap/script.d/qemu_kvm.stp
   /etc/systemtap/script.d/
   # cp /usr/share/qemu-kvm/systemtap/conf.d/qemu_kvm.conf
   /etc/systemtap/conf.d/
   ```

   The set of trace events to enable is given in qemu_kvm.stp. This SystemTap script can be customized to add or remove trace events provided in **/usr/share/systemtap/tapset/qemu-kvm-simpletrace.stp**.

   SystemTap customizations can be made to **qemu_kvm.conf** to control the flight recorder buffer size and whether to store traces in memory only or in the disk as well.

3. **Start the service**
   Start the systemtap service by running the following command:

   ```
   # systemctl start systemtap qemu_kvm
   ```

4. **Make systemtap enabled to run at boot time**
   Enable the systemtap service to run at boot time by running the following command:

   ```
   # systemctl enable systemtap qemu_kvm
   ```

5. **Confirmation the service is running**
   Confirm that the service is working by running the following command:

```
# systemctl status systemtap qemu_kvm
qemu_kvm is running...
```

Procedure A.2. Inspecting the trace buffer

1. **Create a trace buffer dump file**
   Create a trace buffer dump file called trace.log and place it in the tmp directory by running the following command:

   ```
   # staprun -A qemu_kvm >/tmp/trace.log
   ```

   You can change the file name and location to something else.

2. **Start the service**
   As the previous step stops the service, start it again by running the following command:

   ```
   # systemctl start systemtap qemu_kvm
   ```

3. **Convert the trace contents into a readable format**
   To convert the trace file contents into a more readable format, enter the following command:

   ```
   # /usr/share/qemu-kvm/simpletrace.py --no-header /usr/share/qemu-
   kvm/trace-events /tmp/trace.log
   ```

NOTE

The following notes and limitations should be noted:

- The systemtap service is disabled by default.
- There is a small performance penalty when this service is enabled, but it depends on which events are enabled in total.
- There is a README file located in **/usr/share/doc/qemu-kvm-*/README.systemtap**.

A.4. KVM_STAT

The **kvm_stat** command is a python script which retrieves runtime statistics from the **kvm** kernel module. The **kvm_stat** command can be used to diagnose guest behavior visible to **kvm**. In particular, performance related issues with guests. Currently, the reported statistics are for the entire system; the behavior of all running guests is reported. To run this script you need to install the qemu-kvm-tools package. For more information, see Section 2.2, “Installing Virtualization Packages on an Existing Red Hat Enterprise Linux System”.

The **kvm_stat** command requires that the **kvm** kernel module is loaded and **debugfs** is mounted. If either of these features are not enabled, the command will output the required steps to enable **debugfs** or the **kvm** module. For example:

```
# kvm_stat
Please mount debugfs ('mount -t debugfs debugfs /sys/kernel/debug')
and ensure the kvm modules are loaded
```

Mount **debugfs** if required:

```
# mount -t debugfs debugfs /sys/kernel/debug
```

kvm_stat Output

The **kvm_stat** command outputs statistics for all guests and the host. The output is updated until the command is terminated (using **Ctrl**+**c** or the **q** key). Note that the output you see on your screen may differ. For an explanation of the output elements, click any of the terms to link to the defintion.

```
  # kvm_stat

  kvm statistics
   kvm_exit                       17724        66
    Individual exit reasons follow, see kvm_exit (NAME) for more
information.
   kvm_exit(CLGI)                     0         0
   kvm_exit(CPUID)                    0         0
   kvm_exit(CR0_SEL_WRITE)            0         0
   kvm_exit(EXCP_BASE)                0         0
   kvm_exit(FERR_FREEZE)              0         0
   kvm_exit(GDTR_READ)                0         0
   kvm_exit(GDTR_WRITE)               0         0
   kvm_exit(HLT)                     11        11
   kvm_exit(ICEBP)                    0         0
   kvm_exit(IDTR_READ)                0         0
   kvm_exit(IDTR_WRITE)               0         0
   kvm_exit(INIT)                     0         0
   kvm_exit(INTR)                     0         0
   kvm_exit(INVD)                     0         0
   kvm_exit(INVLPG)                   0         0
   kvm_exit(INVLPGA)                  0         0
   kvm_exit(IOIO)                     0         0
   kvm_exit(IRET)                     0         0
   kvm_exit(LDTR_READ)                0         0
   kvm_exit(LDTR_WRITE)               0         0
   kvm_exit(MONITOR)                  0         0
   kvm_exit(MSR)                     40        40
   kvm_exit(MWAIT)                    0         0
   kvm_exit(MWAIT_COND)               0         0
   kvm_exit(NMI)                      0         0
   kvm_exit(NPF)                      0         0
   kvm_exit(PAUSE)                    0         0
   kvm_exit(POPF)                     0         0
   kvm_exit(PUSHF)                    0         0
   kvm_exit(RDPMC)                    0         0
   kvm_exit(RDTSC)                    0         0
   kvm_exit(RDTSCP)                   0         0
   kvm_exit(READ_CR0)                 0         0
   kvm_exit(READ_CR3)                 0         0
   kvm_exit(READ_CR4)                 0         0
   kvm_exit(READ_CR8)                 0         0
   kvm_exit(READ_DR0)                 0         0
   kvm_exit(READ_DR1)                 0         0
   kvm_exit(READ_DR2)                 0         0
   kvm_exit(READ_DR3)                 0         0
```

```
    kvm_exit(READ_DR4)                0         0
    kvm_exit(READ_DR5)                0         0
    kvm_exit(READ_DR6)                0         0
    kvm_exit(READ_DR7)                0         0
    kvm_exit(RSM)                     0         0
    kvm_exit(SHUTDOWN)                0         0
    kvm_exit(SKINIT)                  0         0
    kvm_exit(SMI)                     0         0
    kvm_exit(STGI)                    0         0
    kvm_exit(SWINT)                   0         0
    kvm_exit(TASK_SWITCH)             0         0
    kvm_exit(TR_READ)                 0         0
    kvm_exit(TR_WRITE)                0         0
    kvm_exit(VINTR)                   1         1
    kvm_exit(VMLOAD)                  0         0
    kvm_exit(VMMCALL)                 0         0
    kvm_exit(VMRUN)                   0         0
    kvm_exit(VMSAVE)                  0         0
    kvm_exit(WBINVD)                  0         0
    kvm_exit(WRITE_CR0)               2         2
    kvm_exit(WRITE_CR3)               0         0
    kvm_exit(WRITE_CR4)               0         0
    kvm_exit(WRITE_CR8)               0         0
    kvm_exit(WRITE_DR0)               0         0
    kvm_exit(WRITE_DR1)               0         0
    kvm_exit(WRITE_DR2)               0         0
    kvm_exit(WRITE_DR3)               0         0
    kvm_exit(WRITE_DR4)               0         0
    kvm_exit(WRITE_DR5)               0         0
    kvm_exit(WRITE_DR6)               0         0
    kvm_exit(WRITE_DR7)               0         0
    kvm_entry                     17724        66
    kvm_apic                      13935        51
    kvm_emulate_insn              13924        51
    kvm_mmio                      13897        50
    varl-kvm_eoi                   3222        12
    kvm_inj_virq                   3222        12
    kvm_apic_accept_irq            3222        12
    kvm_pv_eoi                     3184        12
    kvm_fpu                         376         2
    kvm_cr                          177         1
    kvm_apic_ipi                    278         1
    kvm_msi_set_irq                 295         0
    kvm_pio                          79         0
    kvm_userspace_exit               52         0
    kvm_set_irq                      50         0
    kvm_pic_set_irq                  50         0
    kvm_ioapic_set_irq               50         0
    kvm_ack_irq                      25         0
    kvm_cpuid                        90         0
    kvm_msr                          12         0
```

Explanation of variables:

- **`kvm_ack_irq`** - Number of interrupt controller (PIC/IOAPIC) interrupt acknowledgements.

- **kvm_age_page** - Number of page age iterations by memory management unit (MMU) notifiers.
- **kvm_apic** - Number of APIC register accesses.
- **kvm_apic_accept_irq** - Number of interrupts accepted into local APIC.
- **kvm_apic_ipi** - Number of inter processor interrupts.
- **kvm_async_pf_completed** - Number of completions of asynchronous page faults.
- **kvm_async_pf_doublefault** - Number of asynchronous page fault halts.
- **kvm_async_pf_not_present** - Number of initializations of asynchronous page faults.
- **kvm_async_pf_ready** - Number of completions of asynchronous page faults.
- **kvm_cpuid** - Number of CPUID instructions executed.
- **kvm_cr** - Number of trapped and emulated control register (CR) accesses (CR0, CR3, CR4, CR8).
- **kvm_emulate_insn** - Number of emulated instructions.
- **kvm_entry** - Number of emulated instructions.
- **kvm_eoi** - Number of Advanced Programmable Interrupt Controller (APIC) end of interrupt (EOI) notifications.
- **kvm_exit** - Number of **VM-exits**.
- **kvm_exit (NAME)** - Individual exits that are processor-specific. See your processor's documentation for more information.
- **kvm_fpu** - Number of KVM floating-point units (FPU) reloads.
- **kvm_hv_hypercall** - Number of Hyper-V hypercalls.
- **kvm_hypercall** - Number of non-Hyper-V hypercalls.
- **kvm_inj_exception** - Number of exceptions injected into guest.
- **kvm_inj_virq** - Number of interrupts injected into guest.
- **kvm_invlpga** - Number of INVLPGA instructions intercepted.
- **kvm_ioapic_set_irq** - Number of interrupts level changes to the virtual IOAPIC controller.
- **kvm_mmio** - Number of emulated memory-mapped I/O (MMIO) operations.
- **kvm_msi_set_irq** - Number of message-signaled interrupts (MSI).
- **kvm_msr** - Number of model-specific register (MSR) accesses.
- **kvm_nested_intercepts** - Number of L1 ⇒ L2 nested SVM switches.
- **kvm_nested_vmrun** - Number of L1 ⇒ L2 nested SVM switches.

- **kvm_nested_intr_vmexit** - Number of nested VM-exit injections due to interrupt window.
- **kvm_nested_vmexit** - Exits to hypervisor while executing nested (L2) guest.
- **kvm_nested_vmexit_inject** - Number of L2 ⇒ L1 nested switches.
- **kvm_page_fault** - Number of page faults handled by hypervisor.
- **kvm_pic_set_irq** - Number of interrupts level changes to the virtual programmable interrupt controller (PIC).
- **kvm_pio** - Number of emulated programmed I/O (PIO) operations.
- **kvm_pv_eoi** - Number of paravirtual end of input (EOI) events.
- **kvm_set_irq** - Number of interrupt level changes at the generic IRQ controller level (counts PIC, IOAPIC and MSI).
- **kvm_skinit** - Number of SVM SKINIT exits.
- **kvm_track_tsc** - Number of time stamp counter (TSC) writes.
- **kvm_try_async_get_page** - Number of asynchronous page fault attempts.
- **kvm_update_master_clock** - Number of pvclock masterclock updates.
- **kvm_userspace_exit** - Number of exits to user space.
- **kvm_write_tsc_offset** - Number of TSC offset writes.
- **vcpu_match_mmio** - Number of SPTE cached memory-mapped I/O (MMIO) hits.

The output information from the **kvm_stat** command is exported by the KVM hypervisor as pseudo files which are located in the **/sys/kernel/debug/tracing/events/kvm/** directory.

A.5. TROUBLESHOOTING WITH SERIAL CONSOLES

Linux kernels can output information to serial ports. This is useful for debugging kernel panics and hardware issues with video devices or headless servers.

To enable serial console output for KVM guests:

1. Ensure that the domain XML file of the guest includes configuration for the serial console. For example:

```
<console type='pty'>
  <source path='/dev/pts/16'/>
  <target type='virtio' port='1'/>
  <alias name='console1'/>
</console>
```

2. On the guest, follow the How can I enable serial console for Red Hat Enterprise Linux 7? article on Red Hat Knowledgebase.

On the host, you can then access the serial console with the following command, where *guestname* is the name of the guest virtual machine:

```
# virsh console guestname
```

You can also use **virt-manager** to display the virtual text console. In the guest console window, select **Serial 1** in **Text Consoles** from the **View** menu.

A.6. VIRTUALIZATION LOGS

The following methods can be used to access logged data about events on your hypervisor and your guests. This can be helpful when troubleshooting virtualization on your system.

- Each guest has a log, saved in the **/var/log/libvirt/qemu/** directory. The logs are named *GuestName*.log and are periodically compressed when a size limit is reached.
- To view **libvirt** events in the **systemd** Journal, use the following command:

  ```
  # journalctl _SYSTEMD_UNIT=libvirtd.service
  ```

- The **auvirt** command displays audit results related to guests on your hypervisor. The displayed data can be narrowed down by selecting specific guests, time frame, and information format. For example, the following command provides a summary of events on the **testguest** virtual machine on the current day.

  ```
  # auvirt --start today --vm testguest --summary
  Range of time for report:       Mon Sep  4 16:44 - Mon Sep  4 17:04
  Number of guest starts:         2
  Number of guest stops:          1
  Number of resource assignments: 14
  Number of related AVCs:         0
  Number of related anomalies:    0
  Number of host shutdowns:       0
  Number of failed operations:    0
  ```

  You can also configure auvirt information to be automatically included in the **systemd** Journal. To do so, edit the **/etc/libvirt/libvirtd.conf** file and set the value of the ***audit_logging*** parameter to **1**.

  For more information, see the **auvirt** man page.

- If you encounter any errors with the Virtual Machine Manager, you can review the generated data in the **virt-manager.log** file in the **$HOME/.virt-manager/** directory.
- For audit logs of the hypervisor system, see the **/var/log/audit/audit.log** file.
- Depending on the guest operating system, various system log files may also be saved on the guest.

For more information about logging in Red Hat Enterprise Linux, see the System Administrator's Guide.

A.7. LOOP DEVICE ERRORS

If file-based guest images are used you may have to increase the number of configured loop devices. The default configuration allows up to eight active loop devices. If more than eight file-based guests or loop devices are needed the number of loop devices configured can be adjusted in the **/etc/modprobe.d/**directory. Add the following line:

```
options loop max_loop=64
```

This example uses 64 but you can specify another number to set the maximum loop value. You may also have to implement loop device backed guests on your system. To use a loop device backed guests for a full virtualized system, use the **phy: device** or **file: file** commands.

A.8. LIVE MIGRATION ERRORS

There may be cases where a guest changes memory too fast, and the live migration process has to transfer it over and over again, and fails to finish (converge).

The current live-migration implementation has a default migration time configured to 30ms. This value determines the guest pause time at the end of the migration in order to transfer the leftovers. Higher values increase the odds that live migration will converge

A.9. ENABLING INTEL VT-X AND AMD-V VIRTUALIZATION HARDWARE EXTENSIONS IN BIOS

NOTE

To expand your expertise, you might also be interested in the Red Hat Virtualization (RH318) training course.

This section describes how to identify hardware virtualization extensions and enable them in your BIOS if they are disabled.

The Intel VT-x extensions can be disabled in the BIOS. Certain laptop vendors have disabled the Intel VT-x extensions by default in their CPUs.

The virtualization extensions cannot be disabled in the BIOS for AMD-V.

See the following section for instructions on enabling disabled virtualization extensions.

Verify the virtualization extensions are enabled in BIOS. The BIOS settings for Intel VT or AMD-V are usually in the **Chipset** or **Processor** menus. The menu names may vary from this guide, the virtualization extension settings may be found in **Security Settings** or other non standard menu names.

Procedure A.3. Enabling virtualization extensions in BIOS

1. Reboot the computer and open the system's BIOS menu. This can usually be done by pressing the **delete** key, the **F1** key or **Alt** and **F4** keys depending on the system.

2. **Enabling the virtualization extensions in BIOS**

NOTE

Many of the steps below may vary depending on your motherboard, processor type, chipset and OEM. See your system's accompanying documentation for the correct information on configuring your system.

a. Open the **Processor** submenu The processor settings menu may be hidden in the **Chipset**, **Advanced CPU Configuration** or **Northbridge**.

b. Enable **Intel Virtualization Technology** (also known as Intel VT-x). **AMD-V** extensions cannot be disabled in the BIOS and should already be enabled. The virtualization extensions may be labeled **Virtualization Extensions**, **Vanderpool** or various other names depending on the OEM and system BIOS.

c. Enable Intel VT-d or AMD IOMMU, if the options are available. Intel VT-d and AMD IOMMU are used for PCI device assignment.

d. Select **Save & Exit**.

3. Reboot the machine.

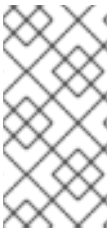
4. When the machine has booted, run **grep -E "vmx|svm" /proc/cpuinfo**. Specifying ***--color*** is optional, but useful if you want the search term highlighted. If the command outputs, the virtualization extensions are now enabled. If there is no output your system may not have the virtualization extensions or the correct BIOS setting enabled.

A.10. SHUTTING DOWN RED HAT ENTERPRISE LINUX 6 GUESTS ON A RED HAT ENTERPRISE LINUX 7 HOST

Installing Red Hat Enterprise Linux 6 guest virtual machines with the **Minimal installation** option does not install the acpid (acpi daemon). Red Hat Enterprise Linux 7 no longer requires this package, as it has been taken over by **systemd**. However, Red Hat Enterprise Linux 6 guest virtual machines running on a Red Hat Enterprise Linux 7 host still require it.

Without the acpid package, the Red Hat Enterprise Linux 6 guest virtual machine does not shut down when the **virsh shutdown** command is executed. The **virsh shutdown** command is designed to gracefully shut down guest virtual machines.

Using the **virsh shutdown** command is easier and safer for system administration. Without graceful shut down with the **virsh shutdown** command a system administrator must log into a guest virtual machine manually or send the **Ctrl**-**Alt**-**Del** key combination to each guest virtual machine.



NOTE

Other virtualized operating systems may be affected by this issue. The **virsh shutdown** command requires that the guest virtual machine operating system is configured to handle ACPI shut down requests. Many operating systems require additional configurations on the guest virtual machine operating system to accept ACPI shut down requests.

Procedure A.4. Workaround for Red Hat Enterprise Linux 6 guests

1. **Install the acpid package**
   The **acpid** service listens and processes ACPI requests.

Log into the guest virtual machine and install the acpid package on the guest virtual machine:

```
# yum install acpid
```

2. **Enable the acpid service on the guest**
   Set the **acpid** service to start during the guest virtual machine boot sequence and start the service:

```
# chkconfig acpid on
# service acpid start
```

3. **Prepare guest domain XML**
   Edit the domain XML file to include the following element. Replace the virtio serial port with **org.qemu.guest_agent.0** and use your guest's name instead of the one shown. In this example, the guest is guest1. Remember to save the file.

```
<channel type='unix'>
   <source mode='bind' path='/var/lib/libvirt/qemu/guest1.agent'/>
   <target type='virtio' name='org.qemu.guest_agent.0'/>
</channel>
```

Figure A.1. Guest XML replacement

4. **Install the QEMU guest agent**
   Install the QEMU guest agent (QEMU-GA) and start the service as directed in the *Red Hat Enterprise Linux 6 Virtualization Administration Guide*.

5. **Shut down the guest**

   a. List the known guest virtual machines so you can retrieve the name of the one you want to shutdown.

```
# virsh list --all
   Id Name                State
----------------------------------
  14 guest1               running
```

   b. Shut down the guest virtual machine.

```
# virsh shutdown guest1

guest virtual machine guest1 is being shutdown
```

   c. Wait a few seconds for the guest virtual machine to shut down. Verify it is shutdown.

```
# virsh list --all
 Id Name                 State
----------------------------------
 14 guest1               shut off
```

d. Start the guest virtual machine named *guest1*, with the XML file you edited.

```
# virsh start guest1
```

e. Shut down the acpi in the *guest1* guest virtual machine.

```
# virsh shutdown --mode acpi guest1
```

f. List all the guest virtual machines again, *guest1* should still be on the list, and it should indicate it is shut off.

```
#  virsh list --all
   Id Name                 State
----------------------------------
   14 guest1               shut off
```

g. Start the guest virtual machine named *guest1*, with the XML file you edited.

```
# virsh start guest1
```

h. Shut down the *guest1* guest virtual machine guest agent.

```
# virsh shutdown --mode agent guest1
```

i. List the guest virtual machines. *guest1* should still be on the list, and it should indicate it is shut off.

```
# virsh list --all
   Id Name                 State
----------------------------------
   guest1                shut off
```

The guest virtual machine will shut down using the **`virsh shutdown`** command for the consecutive shutdowns, without using the workaround described above.

In addition to the method described above, a guest can be automatically shutdown, by stopping the **`libvirt-guests`** service. See Section A.11, “Optional Workaround to Allow for Graceful Shutdown” for more information on this method.

A.11. OPTIONAL WORKAROUND TO ALLOW FOR GRACEFUL SHUTDOWN

The **`libvirt-guests`** service has parameter settings that can be configured to assure that the guest can shutdown properly. It is a package that is a part of the libvirt installation and is installed by default. This service automatically saves guests to the disk when the host shuts down, and restores them to their pre-shutdown state when the host reboots. By default, this setting is set to suspend the guest. If you want the guest to be gracefully shutdown, you will need to change one of the parameters of the **`libvirt-guests`** configuration file.

Procedure A.5. Changing the libvirt-guests service parameters to allow for the graceful shutdown of guests

The procedure described here allows for the graceful shutdown of guest virtual machines when the host physical machine is stuck, powered off, or needs to be restarted.

1. **Open the configuration file**
   The configuration file is located in **/etc/sysconfig/libvirt-guests**. Edit the file, remove the comment mark (#) and change the ***ON_SHUTDOWN=suspend*** to ***ON_SHUTDOWN=shutdown***. Remember to save the change.

```
$ vi /etc/sysconfig/libvirt-guests

# URIs to check for running guests
# example: URIS='default xen:/// vbox+tcp://host/system lxc:///'
```

```
#URIS=default

# action taken on host boot
# - start   all guests which were running on shutdown are started on
boot
#           regardless on their autostart settings
# - ignore  libvirt-guests init script won't start any guest on
boot, however,
#           guests marked as autostart will still be automatically
started by
#           libvirtd
```

```
#ON_BOOT=start

# Number of seconds to wait between each guest start. Set to 0 to
```

```
allow
# parallel startup.
#START_DELAY=0

# action taken on host shutdown
# - suspend   all running guests are suspended using virsh
managedsave
# - shutdown  all running guests are asked to shutdown. Please be
careful with
#             this settings since there is no way to distinguish
between a
#             guest which is stuck or ignores shutdown requests and
a guest
#             which just needs a long time to shutdown. When
setting
#             ON_SHUTDOWN=shutdown, you must also set
SHUTDOWN_TIMEOUT to a
#             value suitable for your guests.
```

```
ON_SHUTDOWN=shutdown

# If set to non-zero, shutdown will suspend guests concurrently.
```

```
Number of
# guests on shutdown at any time will not exceed number set in this
                    5
variable.
#PARALLEL_SHUTDOWN=0

# Number of seconds we're willing to wait for a guest to shut down.
If parallel
# shutdown is enabled, this timeout applies as a timeout for
shutting down all
# guests on a single URI defined in the variable URIS. If this is 0,
then there
# is no time out (use with caution, as guests might not respond to a
shutdown
# request). The default value is 300 seconds (5 minutes).
6
#SHUTDOWN_TIMEOUT=300

# If non-zero, try to bypass the file system cache when saving and
# restoring guests, even though this may give slower operation for
# some file systems.
7
#BYPASS_CACHE=0
```

1	***URIS*** - checks the specified connections for a running guest. The ***Default*** setting functions in the same manner as **virsh** does when no explicit URI is set In addition, one can explicitly set the URI from **/etc/libvirt/libvirt.conf**. Note that when using the libvirt configuration file default setting, no probing will be used.
2	***ON_BOOT*** - specifies the action to be done to / on the guests when the host boots. The ***start*** option starts all guests that were running prior to shutdown regardless on their autostart settings. The ***ignore*** option will not start the formally running guest on boot, however, any guest marked as autostart will still be automatically started by **libvirtd**.
3	The ***START_DELAY*** - sets a delay interval in between starting up the guests. This time period is set in seconds. Use the 0 time setting to make sure there is no delay and that all guests are started simultaneously.
4	***ON_SHUTDOWN*** - specifies the action taken when a host shuts down. Options that can be set include: ***suspend*** which suspends all running guests using **virsh managedsave** and ***shutdown*** which shuts down all running guests. It is best to be careful with using the **shutdown** option as there is no way to distinguish between a guest which is stuck or ignores shutdown requests and a guest that just needs a longer time to shutdown. When setting the ***ON_SHUTDOWN=shutdown***, you must also set ***SHUTDOWN_TIMEOUT*** to a value suitable for the guests.
5	***PARALLEL_SHUTDOWN*** Dictates that the number of guests on shutdown at any time will not exceed number set in this variable and the guests will be suspended concurrently. If set to **0**, then guests are not shutdown concurrently.

6	Number of seconds to wait for a guest to shut down. If ***SHUTDOWN_TIMEOUT*** is enabled, this timeout applies as a timeout for shutting down all guests on a single URI defined in the variable URIS. If ***SHUTDOWN_TIMEOUT*** is set to ***0***, then there is no timeout (use with caution, as guests might not respond to a shutdown request). The default value is 300 seconds (5 minutes).
7	***BYPASS_CACHE*** can have 2 values, 0 to disable and 1 to enable. If enabled it will by-pass the file system cache when guests are restored. Note that setting this may effect performance and may cause slower operation for some file systems.

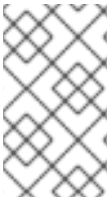
2. **Start libvirt-guests service**
   If you have not started the service, start the **libvirt-guests** service. Do not restart the service as this will cause all running guest virtual machines to shutdown.

A.12. KVM NETWORKING PERFORMANCE

By default, KVM virtual machines are assigned a virtual Realtek 8139 (rtl8139) NIC (network interface controller).

The rtl8139 virtualized NIC works fine in most environments,but this device can suffer from performance degradation problems on some networks, such as a 10 Gigabit Ethernet.

To improve performance, you can switch to the paravirtualized network driver.



NOTE

Note that the virtualized Intel PRO/1000 (**e1000**) driver is also supported as an emulated driver choice. To use the **e1000** driver, replace **virtio** in the procedure below with **e1000**. For the best performance it is recommended to use the **virtio** driver.

Procedure A.6. Switching to the virtio driver

1. Shut down the guest operating system.
2. Edit the guest's configuration file with the **virsh** command (where ***GUEST*** is the guest's name):

   ```
   # virsh edit GUEST
   ```

   The **virsh edit** command uses the **$EDITOR** shell variable to determine which editor to use.
3. Find the network interface section of the configuration. This section resembles the snippet below:

   ```
   <interface type='network'>
     [output truncated]
     <model type='rtl8139' />
   </interface>
   ```
4. Change the type attribute of the model element from ***'rtl8139'*** to ***'virtio'***. This will change the driver from the rtl8139 driver to the e1000 driver.

```
<interface type='network'>
  [output truncated]
  <model type='virtio' />
</interface>
```

5. Save the changes and exit the text editor

6. Restart the guest operating system.

Creating New Guests Using Other Network Drivers

Alternatively, new guests can be created with a different network driver. This may be required if you are having difficulty installing guests over a network connection. This method requires you to have at least one guest already created (possibly installed from CD or DVD) to use as a template.

1. Create an XML template from an existing guest (in this example, named *Guest1*):

```
# virsh dumpxml Guest1 > /tmp/guest-template.xml
```

2. Copy and edit the XML file and update the unique fields: virtual machine name, UUID, disk image, MAC address, and any other unique parameters. Note that you can delete the UUID and MAC address lines and virsh will generate a UUID and MAC address.

```
# cp /tmp/guest-template.xml /tmp/new-guest.xml
# vi /tmp/new-guest.xml
```

Add the model line in the network interface section:

```
 <interface type='network'>
  [output truncated]
  <model type='virtio' />
</interface>
```

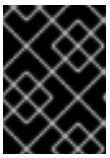
3. Create the new virtual machine:

```
# virsh define /tmp/new-guest.xml
# virsh start new-guest
```

A.13. WORKAROUND FOR CREATING EXTERNAL SNAPSHOTS WITH LIBVIRT

There are two classes of snapshots for KVM guests:

- **Internal snapshots** are contained completely within a qcow2 file, and fully supported by libvirt, allowing for creating, deleting, and reverting of snapshots. This is the default setting used by libvirt when creating a snapshot, especially when no option is specified. This file type take slightly longer than others for creating the snapshot, and has the drawback of requiring qcow2 disks.



IMPORTANT

Internal snapshots are not being actively developed, and Red Hat discourages their use.

- **External snapshots** work with any type of original disk image, can be taken with no guest downtime, and are more stable and reliable. As such, external snapshots are recommended for use on KVM guest virtual machines. However, external snapshots are currently not fully implemented on Red Hat Enterprise Linux 7, and are not available when using **virt-manager**.

  To create an external snapshot, use the **`snapshot-create-as`** with the **`--diskspec vda,snapshot=external`** option, or use the following *disk* line in the snapshot XML file:

  ```
  <disk name='vda' snapshot='external'>
    <source file='/path/to,new'/>
  </disk>
  ```

  At the moment, external snapshots are a one-way operation as libvirt can create them but cannot do anything further with them. A workaround is described on libvirt upstream pages.

A.14. MISSING CHARACTERS ON GUEST CONSOLE WITH JAPANESE KEYBOARD

On a Red Hat Enterprise Linux 7 host, connecting a Japanese keyboard locally to a machine may result in typed characters such as the underscore (the _ character) not being displayed correctly in guest consoles. This occurs because the required keymap is not set correctly by default.

With Red Hat Enterprise Linux 6 and Red Hat Enterprise Linux 7 guests, there is usually no error message produced when pressing the associated key. However, Red Hat Enterprise Linux 4 and Red Hat Enterprise Linux 5 guests may display an error similar to the following:

```
atkdb.c: Unknown key pressed (translated set 2, code 0x0 on
isa0060/serio0).
atkbd.c: Use 'setkeycodes 00 <keycode>' to make it known.
```

To fix this issue in **`virt-manager`**, perform the following steps:

- Open the affected guest in **`virt-manager`**.
- Click **View** → **Details**.
- Select **`Display VNC`** in the list.
- Change **Auto** to **ja** in the **`Keymap`** pull-down menu.
- Click the **`Apply`** button.

Alternatively, to fix this issue using the **`virsh edit`** command on the target guest:

- Run **`virsh edit`** ***`guestname`***
- Add the following attribute to the <graphics> tag: **keymap='ja'**. For example:

  ```
  <graphics type='vnc' port='-1' autoport='yes' keymap='ja'/>
  ```

A.15. GUEST VIRTUAL MACHINE FAILS TO SHUTDOWN

Traditionally, executing a **`virsh shutdown`** command causes a power button ACPI event to be sent,

thus copying the same action as when someone presses a power button on a physical machine. Within every physical machine, it is up to the OS to handle this event. In the past operating systems would just silently shutdown. Today, the most usual action is to show a dialog asking what should be done. Some operating systems even ignore this event completely, especially when no users are logged in. When such operating systems are installed on a guest virtual machine, running **`virsh shutdown`** just does not work (it is either ignored or a dialog is shown on a virtual display). However, if a qemu-guest-agent channel is added to a guest virtual machine and this agent is running inside the guest virtual machine's OS, the **`virsh shutdown`** command will ask the agent to shut down the guest OS instead of sending the ACPI event. The agent will call for a shutdown from inside the guest virtual machine OS and everything works as expected.

Procedure A.7. Configuring the guest agent channel in a guest virtual machine

1. Stop the guest virtual machine.
2. Open the Domain XML for the guest virtual machine and add the following snippet:

```
<channel type='unix'>
    <source mode='bind'/>
    <target type='virtio' name='org.qemu.guest_agent.0'/>
</channel>
```

Figure A.2. Configuring the guest agent channel

3. Start the guest virtual machine, by running **`virsh start [domain]`**.
4. Install qemu-guest-agent on the guest virtual machine (**`yum install qemu-guest-agent`**) and make it run automatically at every boot as a service (qemu-guest-agent.service).

A.16. DISABLE SMART DISK MONITORING FOR GUEST VIRTUAL MACHINES

SMART disk monitoring can be safely disabled as virtual disks and the physical storage devices are managed by the host physical machine.

```
# service smartd stop
# systemctl --del smartd
```

A.17. LIBGUESTFS TROUBLESHOOTING

A test tool is available to check that libguestfs is working. Enter the following command after installing libguestfs (root access not required) to test for normal operation:

```
$ libguestfs-test-tool
```

This tool prints a large amount of text to test the operation of libguestfs. If the test is successful, the following text will appear near the end of the output:

```
===== TEST FINISHED OK =====
```

A.18. TROUBLESHOOTING SR-IOV

This section contains solutions for problems which may affect SR-IOV. If you need additional help, see Section 16.2.4, “Setting PCI device assignment from a pool of SR-IOV virtual functions”.

Error starting the guest

When starting a configured virtual machine, an error occurs as follows:

```
# virsh start test
error: Failed to start domain test
error: Requested operation is not valid: PCI device 0000:03:10.1 is in
use by domain rhel7
```

This error is often caused by a device that is already assigned to another guest or to the host itself.

Error migrating, saving, or dumping the guest

Attempts to migrate and dump the virtual machine cause an error similar to the following:

```
# virsh dump rhel7/tmp/rhel7.dump

error: Failed to core dump domain rhel7 to /tmp/rhel7.dump
error: internal error: unable to execute QEMU command 'migrate': State
blocked by non-migratable device '0000:00:03.0/vfio-pci'
```

Because device assignment uses hardware on the specific host where the virtual machine was started, guest migration and save are not supported when device assignment is in use. Currently, the same limitation also applies to core-dumping a guest; this may change in the future. It is important to note that QEMU does not currently support migrate, save, and dump operations on guest virtual machines with PCI devices attached, unless the **`--memory-only`** option is specified. Currently, it only can support these actions with USB devices. Work is currently being done to improve this in the future.

A.19. COMMON LIBVIRT ERRORS AND TROUBLESHOOTING

This appendix documents common **libvirt**-related problems and errors along with instructions for dealing with them.

Locate the error on the table below and follow the corresponding link under **`Solution`** for detailed troubleshooting information.

Table A.1. Common libvirt errors

Error	Description of problem	Solution
`libvirtd failed to start`	The **libvirt** daemon failed to start. However, there is no information about this error in **`/var/log/messages`**.	Section A.19.1, “**libvirtd** failed to start”

Error	Description of problem	Solution
`Cannot read CA certificate`	This is one of several errors that occur when the URI fails to connect to the hypervisor.	Section A.19.2, “The URI Failed to Connect to the Hypervisor”
Other connectivity errors	These are other errors that occur when the URI fails to connect to the hypervisor.	Section A.19.2, “The URI Failed to Connect to the Hypervisor”
`Failed to create domain from vm.xml error: monitor socket did not show up.: Connection refused`	The guest virtual machine (or domain) starting fails and returns this error or similar.	Section A.19.3, “Guest Starting Fails with Error: **`monitor socket did not show up`**”
`Internal error cannot find character device (null)`	This error can occur when attempting to connect a guest's console. It reports that there is no serial console configured for the guest virtual machine.	Section A.19.4, “**`internal error cannot find character device (null)`**”
`No boot device`	After building a guest virtual machine from an existing disk image, the guest booting stalls. However, the guest can start successfully using the **`QEMU`** command directly.	Section A.19.5, “Guest Virtual Machine Booting Stalls with Error: **`No boot device`**”
`The virtual network "default" has not been started`	If the *default* network (or other locally-created network) is unable to start, any virtual machine configured to use that network for its connectivity will also fail to start.	Section A.19.6, “Virtual network *default* has not been started”
PXE boot (or DHCP) on guest failed	A guest virtual machine starts successfully, but is unable to acquire an IP address from DHCP, boot using the PXE protocol, or both. This is often a result of a long forward delay time set for the bridge, or when the iptables package and kernel do not support checksum mangling rules.	Section A.19.7, “PXE Boot (or DHCP) on Guest Failed”

Error	Description of problem	Solution
Guest can reach outside network, but cannot reach host when using macvtap interface	A guest can communicate with other guests, but cannot connect to the host machine after being configured to use a macvtap (or ***type='direct'***) network interface. This is actually not an error — it is the defined behavior of macvtap.	Section A.19.8, “Guest Can Reach Outside Network, but Cannot Reach Host When Using macvtap interface”
Could not add rule to fixup DHCP response checksums on network *'default'*	This warning message is almost always harmless, but is often mistakenly seen as evidence of a problem.	Section A.19.9, “Could not add rule to fixup DHCP response checksums on network *'default'*”
Unable to add bridge br0 port vnet0: No such device	This error message or the similar **Failed to add tap interface to bridge *'br0'*: No such device** reveal that the bridge device specified in the guest's (or domain's) **<interface>** definition does not exist.	Section A.19.10, “Unable to add bridge br0 port vnet0: No such device”
Warning: could not open /dev/net/tun: no virtual network emulation qemu-kvm: -netdev tap,script=/etc/my-qemu-ifup,id=hostnet0: Device 'tap' could not be initialized	The guest virtual machine does not start after configuring a **type='ethernet'** (or 'generic ethernet') interface in the host system. This error or similar appears either in **libvirtd.log**, **/var/log/libvirt/qemu/*name_of_guest*.log**, or in both.	Section A.19.11, “Guest is Unable to Start with Error: **warning: could not open /dev/net/tun**”
Unable to resolve address *name_of_host* service '49155': Name or service not known	**QEMU** guest migration fails and this error message appears with an unfamiliar host name.	Section A.19.12, “Migration Fails with **error: unable to resolve address**”
Unable to allow access for disk path /var/lib/libvirt/images/qemu.img: No such file or directory	A guest virtual machine cannot be migrated because **libvirt** cannot access the disk image(s).	Section A.19.13, “Migration Fails with **Unable to allow access for disk path: No such file or directory**”
No guest virtual machines are present when **libvirtd** is started	The **libvirt** daemon is successfully started, but no guest virtual machines appear to be present when running **virsh list --all**.	Section A.19.14, “No Guest Virtual Machines are Present when **libvirtd** is Started”

Error	Description of problem	Solution
Common XML errors	**libvirt** uses XML documents to store structured data. Several common errors occur with XML documents when they are passed to **libvirt** through the API. This entry provides instructions for editing guest XML definitions, and details common errors in XML syntax and configuration.	Section A.19.16, “Common XML Errors”

A.19.1. libvirtd failed to start

Symptom

The **libvirt** daemon does not start automatically. Starting the **libvirt** daemon manually fails as well:

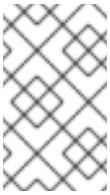
```
# systemctl start libvirtd.service
* Caching service dependencies ...
[ ok ]
* Starting libvirtd ...
/usr/sbin/libvirtd: error: Unable to initialize network sockets. Check
/var/log/messages or run without --daemon for more info.
* start-stop-daemon: failed to start `/usr/sbin/libvirtd'
[ !! ]
* ERROR: libvirtd failed to start
```

Moreover, there is not **'more info'** about this error in **/var/log/messages**.

Investigation

Change **libvirt's** logging in **/etc/libvirt/libvirtd.conf** by enabling the line below. To enable the setting the line, open the **/etc/libvirt/libvirtd.conf** file in a text editor, remove the hash (or #) symbol from the beginning of the following line, and save the change:

```
log_outputs="3:syslog:libvirtd"
```



NOTE

This line is commented out by default to prevent **libvirt** from producing excessive log messages. After diagnosing the problem, it is recommended to comment this line again in the **/etc/libvirt/libvirtd.conf** file.

Restart **libvirt** to determine if this has solved the problem.

If **libvirtd** still does not start successfully, an error similar to the following will be printed:

```
# systemctl restart libvirtd
Job for libvirtd.service failed because the control process exited with
error code. See "systemctl status libvirtd.service" and "journalctl -xe"
for details.
```

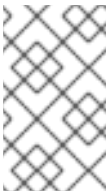
```
Sep 19 16:06:02 jsrh libvirtd[30708]: 2017-09-19 14:06:02.097+0000:
30708: info : libvirt version: 3.7.0, package: 1.el7 (Unknown, 2017-09-
06-09:01:55, js
Sep 19 16:06:02 jsrh libvirtd[30708]: 2017-09-19 14:06:02.097+0000:
30708: info : hostname: jsrh
Sep 19 16:06:02 jsrh libvirtd[30708]: 2017-09-19 14:06:02.097+0000:
30708: error : daemonSetupNetworking:502 : unsupported configuration: No
server certif
Sep 19 16:06:02 jsrh systemd[1]: libvirtd.service: main process exited,
code=exited, status=6/NOTCONFIGURED
Sep 19 16:06:02 jsrh systemd[1]: Failed to start Virtualization daemon.

-- Subject: Unit libvirtd.service has failed
-- Defined-By: systemd
-- Support: http://lists.freedesktop.org/mailman/listinfo/systemd-devel
--
-- Unit libvirtd.service has failed.
--
-- The result is failed.
```

The **libvirtd** man page shows that the missing **cacert.pem** file is used as TLS authority when **libvirt** is run in **Listen for TCP/IP connections** mode. This means the ***--listen*** parameter is being passed.

Solution

Configure the **libvirt** daemon's settings with one of the following methods:

- Install a CA certificate.

  NOTE

  For more information on CA certificates and configuring system authentication, see the Managing Certificates and Certificate Authorities chapter in the *Red Hat Enterprise Linux 7 Domain Identity, Authentication, and Policy Guide*.

- Do not use TLS; use bare TCP instead. In **/etc/libvirt/libvirtd.conf** set **listen_tls = 0** and **listen_tcp = 1**. The default values are **listen_tls = 1** and **listen_tcp = 0**.

- Do not pass the ***--listen*** parameter. In **/etc/sysconfig/libvirtd.conf** change the ***LIBVIRTD_ARGS*** variable.

A.19.2. The URI Failed to Connect to the Hypervisor

Several different errors can occur when connecting to the server (for example, when running **virsh**).

A.19.2.1. Cannot read CA certificate

Symptom

When running a command, the following error (or similar) appears:

```
$ virsh -c qemu://$hostname/system_list
error: failed to connect to the hypervisor
error: Cannot read CA certificate '/etc/pki/CA/cacert.pem': No such file
or directory
```

Investigation

The error message is misleading about the actual cause. This error can be caused by a variety of factors, such as an incorrectly specified URI, or a connection that is not configured.

Solution

Incorrectly specified URI

When specifying ***qemu://system*** or ***qemu://session*** as a connection URI, **virsh** attempts to connect to host names' ***system*** or ***session*** respectively. This is because **virsh** recognizes the text after the second forward slash as the host.

Use three forward slashes to connect to the local host. For example, specifying ***qemu:///system*** instructs **virsh** connect to the ***system*** instance of **libvirtd** on the local host.

When a host name is specified, the **QEMU** transport defaults to **TLS**. This results in certificates.

Connection is not configured

The URI is correct (for example, ***qemu[+tls]://server/system***) but the certificates are not set up properly on your machine. For information on configuring TLS, see the upstream **libvirt** website.

A.19.2.2. Other Connectivity Errors

Unable to connect to server at `server:port`: Connection refused

The daemon is not running on the server or is configured not to listen, using configuration option **listen_tcp** or **listen_tls**.

End of file while reading data: nc: using stream socket: Input/output error

If you specified **ssh** transport, the daemon is likely not running on the server. Solve this error by verifying that the daemon is running on the server.

A.19.3. Guest Starting Fails with Error: `monitor socket did not show up`

Symptom

The guest virtual machine (or domain) starting fails with this error (or similar):

```
# virsh -c qemu:///system create name_of_guest.xml error: Failed to
create domain from name_of_guest.xml error: monitor socket did not show
up.: Connection refused
```

Investigation

This error message shows:

1. **libvirt** is working;
2. The **QEMU** process failed to start up; and
3. **libvirt** quits when trying to connect **QEMU** or the QEMU agent monitor socket.

To understand the error details, examine the guest log:

```
# cat /var/log/libvirt/qemu/name_of_guest.log
LC_ALL=C PATH=/sbin:/usr/sbin:/bin:/usr/bin QEMU_AUDIO_DRV=none
/usr/bin/qemu-kvm -S -M pc -enable-kvm -m 768 -smp
1,sockets=1,cores=1,threads=1 -name name_of_guest -uuid ebfaadbe-e908-
ba92-fdb8-3fa2db557a42 -nodefaults -chardev
socket,id=monitor,path=/var/lib/libvirt/qemu/name_of_guest.monitor,serve
r,nowait -mon chardev=monitor,mode=readline -no-reboot -boot c -kernel
/var/lib/libvirt/boot/vmlinuz -initrd /var/lib/libvirt/boot/initrd.img -
append
method=http://www.example.com/pub/product/release/version/x86_64/os/ -
drive file=/var/lib/libvirt/images/name_of_guest.img,if=none,id=drive-
ide0-0-0,boot=on -device ide-drive,bus=ide.0,unit=0,drive=drive-ide0-0-
0,id=ide0-0-0 -device virtio-net-
pci,vlan=0,id=net0,mac=52:40:00:f4:f1:0a,bus=pci.0,addr=0x4 -net
tap,fd=42,vlan=0,name=hostnet0 -chardev pty,id=serial0 -device isa-
serial,chardev=serial0 -usb -vnc 127.0.0.1:0 -k en-gb -vga cirrus -
device virtio-balloon-pci,id=balloon0,bus=pci.0,
addr=0x3
char device redirected to /dev/pts/1
qemu: could not load kernel '/var/lib/libvirt/boot/vmlinuz':
Permission denied
```

Solution

The guest log contains the details needed to fix the error.

If a host physical machine is shut down while the guest is still running a **libvirt** version prior to 0.9.5, the libvirt-guest's init script attempts to perform a managed save of the guest. If the managed save was incomplete (for example, due to loss of power before the managed save image was flushed to disk), the save image is corrupted and will not be loaded by **QEMU**. The older version of **libvirt** does not recognize the corruption, making the problem perpetual. In this case, the guest log shows an attempt to use **-incoming** as one of its arguments, meaning that **libvirt** is trying to start **QEMU** by migrating in the saved state file.

This problem can be fixed by running **virsh managedsave-remove *name_of_guest*** to remove the corrupted managed save image. Newer versions of **libvirt** take steps to avoid the corruption in the first place, as well as adding **virsh start --force-boot *name_of_guest*** to bypass any managed save image.

A.19.4. internal error cannot find character device (null)

Symptom

This error message appears when attempting to connect to a guest virtual machine's console:

```
# virsh console test2
Connected to domain test2
```

```
Escape character is ^]
error: internal error cannot find character device (null)
```

Investigation

This error message shows that there is no serial console configured for the guest virtual machine.

Solution

Set up a serial console in the guest's XML file.

Procedure A.8. Setting up a serial console in the guest's XML

1. Add the following XML to the guest virtual machine's XML using **virsh edit**:

```
<serial type='pty'>
  <target port='0'/>
</serial>
<console type='pty'>
  <target type='serial' port='0'/>
</console>
```

2. Set up the console in the guest kernel command line.

   To do this, either log in to the guest virtual machine to edit the **/boot/grub2/grub.cfg** file directly, or use the **virt-edit** command-line tool. Add the following to the guest kernel command line:

```
console=ttyS0,115200
```

3. Run the followings command:

```
# virsh start vm && virsh console vm
```

A.19.5. Guest Virtual Machine Booting Stalls with Error: `No boot device`

Symptom

After building a guest virtual machine from an existing disk image, the guest booting stalls with the error message **No boot device**. However, the guest virtual machine can start successfully using the **QEMU** command directly.

Investigation

The disk's bus type is not specified in the command for importing the existing disk image:

```
# virt-install \
--connect qemu:///system \
--ram 2048 -n rhel_64 \
--os-type=linux --os-variant=rhel5 \
--disk  path=/root/RHEL-Server-5.8-64-
virtio.qcow2,device=disk,format=qcow2 \
--vcpus=2 --graphics spice --noautoconsole --import
```

However, the command line used to boot up the guest virtual machine using **QEMU** directly shows that it uses **`virtio`** for its bus type:

```
# ps -ef | grep qemu
/usr/libexec/qemu-kvm -monitor stdio -drive file=/root/RHEL-Server-5.8-
32-virtio.qcow2,index=0,if=virtio,media=disk,cache=none,format=qcow2 -
net nic,vlan=0,model=rtl8139,macaddr=00:30:91:aa:04:74 -net
tap,vlan=0,script=/etc/qemu-ifup,downscript=no -m 2048 -smp
2,cores=1,threads=1,sockets=2 -cpu qemu64,+sse2 -soundhw ac97 -rtc-td-
hack -M rhel5.6.0 -usbdevice tablet -vnc :10 -boot c -no-kvm-pit-
reinjection
```

Note the ***bus=*** in the guest's XML generated by **libvirt** for the imported guest:

```
<domain type='qemu'>
 <name>rhel_64</name>
 <uuid>6cd34d52-59e3-5a42-29e4-1d173759f3e7</uuid>
 <memory>2097152</memory>
 <currentMemory>2097152</currentMemory>
 <vcpu>2</vcpu>
 <os>
   <type arch='x86_64' machine='rhel5.4.0'>hvm</type>
   <boot dev='hd'/>
 </os>
 <features>
   <acpi/>
   <apic/>
   <pae/>
 </features>
 <clock offset='utc'>
   <timer name='pit' tickpolicy='delay'/>
 </clock>
 <on_poweroff>destroy</on_poweroff>
 <on_reboot>restart</on_reboot>
 <on_crash>restart</on_crash>
 <devices>
   <emulator>/usr/libexec/qemu-kvm</emulator>
   <disk type='file' device='disk'>
     <driver name='qemu' type='qcow2' cache='none'/>
     <source file='/root/RHEL-Server-5.8-64-virtio.qcow2'/>
     <emphasis role="bold"><target dev='hda' bus='ide'/></emphasis>
     <address type='drive' controller='0' bus='0' unit='0'/>
   </disk>
   <controller type='ide' index='0'/>
   <interface type='bridge'>
     <mac address='54:52:00:08:3e:8c'/>
     <source bridge='br0'/>
   </interface>
   <serial type='pty'>
     <target port='0'/>
   </serial>
   <console type='pty'>
     <target port='0'/>
   </console>
   <input type='mouse' bus='ps2'/>
```

```
    <graphics type='vnc' port='-1' autoport='yes' keymap='en-us'/>
    <video>
      <model type='cirrus' vram='9216' heads='1'/>
    </video>
  </devices>
  </domain>
```

The bus type for the disk is set as ***ide***, which is the default value set by **libvirt**. This is the incorrect bus type, and has caused the unsuccessful boot for the imported guest.

Solution

Procedure A.9. Correcting the disk bus type

1. Undefine the imported guest virtual machine, then re-import it with ***bus=virtio*** and the following:

```
# virsh destroy rhel_64
# virsh undefine rhel_64
# virt-install \
--connect qemu:///system \
--ram 1024 -n rhel_64 -r 2048 \
--os-type=linux --os-variant=rhel5  \
--disk path=/root/RHEL-Server-5.8-64-
virtio.qcow2,device=disk,bus=virtio,format=qcow2 \
--vcpus=2 --graphics spice --noautoconsole --import
```

2. Edit the imported guest's XML using **`virsh edit`** and correct the disk bus type.

A.19.6. Virtual network *default* has not been started

Symptom

Normally, the configuration for a virtual network named *default* is installed as part of the libvirt package, and is configured to autostart when **libvirtd** is started.

If the *default* network (or any other locally-created network) is unable to start, any virtual machine configured to use that network for its connectivity will also fail to start, resulting in this error message:

```
Virtual network default has not been started
```

Investigation

One of the most common reasons for a **libvirt** virtual network's failure to start is that the dnsmasq instance required to serve DHCP and DNS requests from clients on that network has failed to start.

To determine if this is the cause, run **`virsh net-start default`** from a root shell to start the *default* virtual network.

If this action does not successfully start the virtual network, open **`/var/log/libvirt/libvirtd.log`** to view the complete error log message.

If a message similar to the following appears, the problem is likely a system-wide dnsmasq instance that is already listening on **libvirt**'s bridge, and is preventing **libvirt**'s own dnsmasq instance from doing so. The most important parts to note in the error message are **dnsmasq** and **exit status 2**:

```
Could not start virtual network default: internal error
Child process (/usr/sbin/dnsmasq --strict-order --bind-interfaces
--pid-file=/var/run/libvirt/network/default.pid --conf-file=
--except-interface lo --listen-address 192.168.122.1
--dhcp-range 192.168.122.2,192.168.122.254
--dhcp-leasefile=/var/lib/libvirt/dnsmasq/default.leases
--dhcp-lease-max=253 --dhcp-no-override) status unexpected: exit status
2
```

Solution

If the machine is not using dnsmasq to serve DHCP for the physical network, disable dnsmasq completely.

If it is necessary to run dnsmasq to serve DHCP for the physical network, edit the **/etc/dnsmasq.conf** file. Add or remove the comment mark the first line, as well as one of the two lines following that line. Do not add or remove the comment from all three lines:

```
bind-interfaces
interface=name_of_physical_interface
listen-address=chosen_IP_address
```

After making this change and saving the file, restart the system wide dnsmasq service.

Next, start the *default* network with the **virsh net-start default** command.

Start the virtual machines.

A.19.7. PXE Boot (or DHCP) on Guest Failed

Symptom

A guest virtual machine starts successfully, but is then either unable to acquire an IP address from DHCP or boot using the PXE protocol, or both. There are two common causes of this error: having a long forward delay time set for the bridge, and when the iptables package and kernel do not support checksum mangling rules.

Long forward delay time on bridge

Investigation

This is the most common cause of this error. If the guest network interface is connecting to a bridge device that has STP (Spanning Tree Protocol) enabled, as well as a long forward delay set, the bridge will not forward network packets from the guest virtual machine onto the bridge until at least that number of forward delay seconds have elapsed since the guest connected to the bridge. This delay allows the bridge time to watch traffic from the interface and determine the MAC addresses behind it, and prevent forwarding loops in the network topology.

If the forward delay is longer than the timeout of the guest's PXE or DHCP client, the client's operation will fail, and the guest will either fail to boot (in the case of PXE) or fail to acquire an IP address (in the case of DHCP).

Solution

If this is the case, change the forward delay on the bridge to 0, disable STP on the bridge, or both.

NOTE

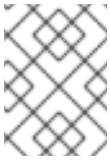
This solution applies only if the bridge is not used to connect multiple networks, but just to connect multiple endpoints to a single network (the most common use case for bridges used by **libvirt**).

If the guest has interfaces connecting to a **libvirt**-managed virtual network, edit the definition for the network, and restart it. For example, edit the default network with the following command:

```
# virsh net-edit default
```

Add the following attributes to the **<bridge>** element:

```
<name_of_bridge='virbr0' delay='0' stp='on'/>
```



NOTE

delay='0' and **stp='on'** are the default settings for virtual networks, so this step is only necessary if the configuration has been modified from the default.

If the guest interface is connected to a host bridge that was configured outside of **libvirt**, change the delay setting.

Add or edit the following lines in the **/etc/sysconfig/network-scripts/ifcfg-*name_of_bridge*** file to turn STP on with a 0 second delay:

```
STP=on DELAY=0
```

After changing the configuration file, restart the bridge device:

```
/usr/sbin/ifdown name_of_bridge
/usr/sbin/ifup name_of_bridge
```

NOTE

If *name_of_bridge* is not the root bridge in the network, that bridge's delay will be eventually reset to the delay time configured for the root bridge. To prevent this from occurring, disable STP on *name_of_bridge*.

The iptables package and kernel do not support checksum mangling rules

Investigation

This message is only a problem if all four of the following conditions are true:

- The guest is using **virtio** network devices.

  If so, the configuration file will contain **model type='virtio'**

- The host has the **vhost-net** module loaded.

  This is true if **ls /dev/vhost-net** does not return an empty result.

- The guest is attempting to get an IP address from a DHCP server that is running directly on the host.

- The iptables version on the host is older than 1.4.10.

  iptables 1.4.10 was the first version to add the **libxt_CHECKSUM** extension. This is the case if the following message appears in the **libvirtd** logs:

```
warning: Could not add rule to fixup DHCP response checksums
on network default
warning: May need to update iptables package and kernel to
support CHECKSUM rule.
```

IMPORTANT

Unless all of the other three conditions in this list are also true, the above warning message can be disregarded, and is not an indicator of any other problems.

When these conditions occur, UDP packets sent from the host to the guest have uncomputed checksums. This makes the host's UDP packets seem invalid to the guest's network stack.

Solution

To solve this problem, invalidate any of the four points above. The best solution is to update the host iptables and kernel to iptables-1.4.10 or newer where possible. Otherwise, the most specific fix is to disable the **vhost-net** driver for this particular guest. To do this, edit the guest configuration with this command:

```
virsh edit name_of_guest
```

Change or add a **<driver>** line to the **<interface>** section:

```
<interface type='network'>
  <model type='virtio'/>
  <driver name='qemu'/>
  ...
</interface>
```

Save the changes, shut down the guest, and then restart it.

If this problem is still not resolved, the issue may be due to a conflict between **firewalld** and the default **libvirt** network.

To fix this, stop **firewalld** with the **service firewalld stop** command, then restart **libvirt** with the **service libvirtd restart** command.

A.19.8. Guest Can Reach Outside Network, but Cannot Reach Host When Using macvtap interface

Symptom

A guest virtual machine can communicate with other guests, but cannot connect to the host machine after being configured to use a macvtap (also known as ***type='direct'***) network interface.

Investigation

Even when not connecting to a Virtual Ethernet Port Aggregator (VEPA) or VN-Link capable switch, macvtap interfaces can be useful. Setting the mode of such an interface to **bridge** allows the guest to be directly connected to the physical network in a very simple manner without the setup issues (or NetworkManager incompatibility) that can accompany the use of a traditional host bridge device.

However, when a guest virtual machine is configured to use a **type='direct'** network interface such as macvtap, despite having the ability to communicate with other guests and other external hosts on the network, the guest cannot communicate with its own host.

This situation is actually not an error — it is the defined behavior of macvtap. Due to the way in which the host's physical Ethernet is attached to the macvtap bridge, traffic into that bridge from the guests that is forwarded to the physical interface cannot be bounced back up to the host's IP stack. Additionally, traffic from the host's IP stack that is sent to the physical interface cannot be bounced back up to the macvtap bridge for forwarding to the guests.

Solution

Use **libvirt** to create an isolated network, and create a second interface for each guest virtual machine that is connected to this network. The host and guests can then directly communicate over this isolated network, while also maintaining compatibility with NetworkManager.

Procedure A.10. Creating an isolated network with libvirt

1. Add and save the following XML in the **/tmp/isolated.xml** file. If the 192.168.254.0/24 network is already in use elsewhere on your network, you can choose a different network.

```
...
<network>
  <name>isolated</name>
  <ip address='192.168.254.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.254.2' end='192.168.254.254'/>
    </dhcp>
  </ip>
</network>
...
```

Figure A.3. Isolated Network XML

2. Create the network with this command: **virsh net-define /tmp/isolated.xml**
3. Set the network to autostart with the **virsh net-autostart isolated** command.
4. Start the network with the **virsh net-start isolated** command.

5. Using **virsh edit *name_of_guest***, edit the configuration of each guest that uses macvtap for its network connection and add a new **<interface>** in the **<devices>** section similar to the following (note the **<model type='virtio'/>** line is optional to include):

```
...
<interface type='network' trustGuestRxFilters='yes'>
  <source network='isolated'/>
  <model type='virtio'/>
</interface>
```

Figure A.4. Interface Device XML

6. Shut down, then restart each of these guests.

The guests are now able to reach the host at the address 192.168.254.1, and the host will be able to reach the guests at the IP address they acquired from DHCP (alternatively, you can manually configure the IP addresses for the guests). Since this new network is isolated to only the host and guests, all other communication from the guests will use the macvtap interface. For more information, see Section 23.18.9, “Network Interfaces”.

A.19.9. Could not add rule to fixup DHCP response checksums on network *'default'*

Symptom

This message appears:

```
Could not add rule to fixup DHCP response checksums on network 'default'
```

Investigation

Although this message appears to be evidence of an error, it is almost always harmless.

Solution

Unless the problem you are experiencing is that the guest virtual machines are unable to acquire IP addresses through DHCP, this message can be ignored.

If this is the case, see Section A.19.7, “PXE Boot (or DHCP) on Guest Failed” for further details on this situation.

A.19.10. Unable to add bridge br0 port vnet0: No such device

Symptom

The following error message appears:

```
Unable to add bridge name_of_bridge port vnet0: No such device
```

For example, if the bridge name is *br0*, the error message appears as:

```
Unable to add bridge br0 port vnet0: No such device
```

In **libvirt** versions 0.9.6 and earlier, the same error appears as:

```
Failed to add tap interface to bridge name_of_bridge: No such device
```

Or for example, if the bridge is named *br0*:

```
Failed to add tap interface to bridge 'br0': No such device
```

Investigation

Both error messages reveal that the bridge device specified in the guest's (or domain's) **<interface>** definition does not exist.

To verify the bridge device listed in the error message does not exist, use the shell command **ip addr show *br0***.

A message similar to this confirms the host has no bridge by that name:

```
br0: error fetching interface information: Device not found
```

If this is the case, continue to the solution.

However, if the resulting message is similar to the following, the issue exists elsewhere:

```
br0       Link encap:Ethernet  HWaddr 00:00:5A:11:70:48
          inet addr:10.22.1.5  Bcast:10.255.255.255  Mask:255.0.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:249841 errors:0 dropped:0 overruns:0 frame:0
          TX packets:281948 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
    RX bytes:106327234 (101.4 MiB)  TX bytes:21182634 (20.2 MiB)
```

Solution

Edit the existing bridge or create a new bridge with `virsh`

Use **virsh** to either edit the settings of an existing bridge or network, or to add the bridge device to the host system configuration.

Edit the existing bridge settings using `virsh`

Use **virsh edit *name_of_guest*** to change the **<interface>** definition to use a bridge or network that already exists.

For example, change **type='bridge'** to **type='network'**, and **<source bridge='br0'/>** to **<source network='default'/>**.

Create a host bridge using `virsh`

For **libvirt** version 0.9.8 and later, a bridge device can be created with the **virsh iface-bridge** command. This creates a bridge device *br0* with **eth0**, the physical network interface that is set as part of a bridge, attached:

```
virsh iface-bridge eth0 br0
```

Optional: If needed, remove this bridge and restore the original **eth0** configuration with this command:

```
virsh iface-unbridge br0
```

Create a host bridge manually

For older versions of **libvirt**, it is possible to manually create a bridge device on the host. For instructions, see Section 6.4.3, “Bridged Networking with libvirt”.

A.19.11. Guest is Unable to Start with Error: warning: could not open /dev/net/tun

Symptom

The guest virtual machine does not start after configuring a **type='ethernet'** (also known as 'generic ethernet') interface in the host system. An error appears either in **libvirtd.log**, **/var/log/libvirt/qemu/*name_of_guest*.log**, or in both, similar to the below message:

```
warning: could not open /dev/net/tun: no virtual network emulation qemu-
kvm: -netdev tap,script=/etc/my-qemu-ifup,id=hostnet0: Device 'tap'
could not be initialized
```

Investigation

Use of the generic ethernet interface type (**<interface type='ethernet'>**) is discouraged, because using it requires lowering the level of host protection against potential security flaws in **QEMU** and its guests. However, it is sometimes necessary to use this type of interface to take advantage of some other facility that is not yet supported directly in **libvirt**. For example, **openvswitch** was not supported in **libvirt** until version 0.9.11, so in prior versions of **libvirt**, **<interface type='ethernet'>** was the only way to connect a guest to an **openvswitch** bridge.

However, if you configure a **<interface type='ethernet'>** interface without making any other changes to the host system, the guest virtual machine does not start successfully.

The reason for this failure is that for this type of interface, a script called by **QEMU** needs to manipulate the tap device. However, with **type='ethernet'** configured, in an attempt to lock down **QEMU**, **libvirt** and SELinux have put in place several checks to prevent this. (Normally, **libvirt** performs all of the tap device creation and manipulation, and passes an open file descriptor for the tap device to **QEMU**.)

Solution

Reconfigure the host system to be compatible with the generic ethernet interface.

Procedure A.11. Reconfiguring the host system to use the generic ethernet interface

1. Set SELinux to permissive by configuring **SELINUX=permissive** in **/etc/selinux/config**:

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
```

```
#       enforcing - SELinux security policy is enforced.
#       permissive - SELinux prints warnings instead of
enforcing.
#       disabled - No SELinux policy is loaded.
SELINUX=permissive
# SELINUXTYPE= can take one of these two values:
#       targeted - Targeted processes are protected,
#       mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

2. From a root shell, run the command **setenforce permissive**.

3. In **/etc/libvirt/qemu.conf** add or edit the following lines:

```
clear_emulator_capabilities = 0
```

```
user = "root"
```

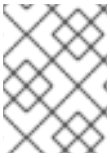
```
group = "root"
```

```
cgroup_device_acl = [
        "/dev/null", "/dev/full", "/dev/zero",
        "/dev/random", "/dev/urandom",
        "/dev/ptmx", "/dev/kvm", "/dev/kqemu",
 "/dev/rtc", "/dev/hpet", "/dev/net/tun",
```

4. Restart **libvirtd**.

IMPORTANT

Since each of these steps significantly decreases the host's security protections against **QEMU** guest domains, this configuration should only be used if there is no alternative to using **<interface type='ethernet'>**.



NOTE

For more information on SELinux, see the *Red Hat Enterprise Linux 7 SELinux User's and Administrator's Guide*.

A.19.12. Migration Fails with `error: unable to resolve address`

Symptom

QEMU guest migration fails and this error message appears:

```
# virsh migrate qemu qemu+tcp://192.168.122.12/system
  error: Unable to resolve address name_of_host service '49155': Name or
service not known
```

For example, if the destination host name is **newyork**, the error message appears as:

```
# virsh migrate qemu qemu+tcp://192.168.122.12/system
error: Unable to resolve address 'newyork' service '49155': Name or
service not known
```

However, this error looks strange as we did not use **newyork** host name anywhere.

Investigation

During migration, **libvirtd** running on the destination host creates a URI from an address and port where it expects to receive migration data and sends it back to **libvirtd** running on the source host.

In this case, the destination host (**192.168.122.12**) has its name set to *'newyork'*. For some reason, **libvirtd** running on that host is unable to resolve the name to an IP address that could be sent back and still be useful. For this reason, it returned the *'newyork'* host name hoping the source **libvirtd** would be more successful with resolving the name. This can happen if DNS is not properly configured or **/etc/hosts** has the host name associated with local loopback address (**127.0.0.1**).

Note that the address used for migration data cannot be automatically determined from the address used for connecting to destination **libvirtd** (for example, from **qemu+tcp://192.168.122.12/system**). This is because to communicate with the destination **libvirtd**, the source **libvirtd** may need to use network infrastructure different from the type that **virsh** (possibly running on a separate machine) requires.

Solution

The best solution is to configure DNS correctly so that all hosts involved in migration are able to resolve all host names.

If DNS cannot be configured to do this, a list of every host used for migration can be added manually to the **/etc/hosts** file on each of the hosts. However, it is difficult to keep such lists consistent in a dynamic environment.

If the host names cannot be made resolvable by any means, **virsh migrate** supports specifying the migration host:

```
# virsh migrate qemu qemu+tcp://192.168.122.12/system
tcp://192.168.122.12
```

Destination **libvirtd** will take the **tcp://192.168.122.12** URI and append an automatically generated port number. If this is not desirable (because of firewall configuration, for example), the port number can be specified in this command:

```
# virsh migrate qemu qemu+tcp://192.168.122.12/system
tcp://192.168.122.12:12345
```

Another option is to use tunneled migration. Tunneled migration does not create a separate connection for migration data, but instead tunnels the data through the connection used for communication with destination **libvirtd** (for example, **qemu+tcp://192.168.122.12/system**):

```
# virsh migrate qemu qemu+tcp://192.168.122.12/system --p2p --tunnelled
```

A.19.13. Migration Fails with `Unable to allow access for disk path: No such file or directory`

Symptom

A guest virtual machine (or domain) cannot be migrated because **libvirt** cannot access the disk image(s):

```
# virsh migrate qemu qemu+tcp://name_of_host/system
error: Unable to allow access for disk path
/var/lib/libvirt/images/qemu.img: No such file or directory
```

For example, if the destination host name is **newyork**, the error message appears as:

```
# virsh migrate qemu qemu+tcp://newyork/system
error: Unable to allow access for disk path
/var/lib/libvirt/images/qemu.img: No such file or directory
```

Investigation

By default, migration only transfers the in-memory state of a running guest (such as memory or CPU state). Although disk images are not transferred during migration, they need to remain accessible at the same path by both hosts.

Solution

Set up and mount shared storage at the same location on both hosts. The simplest way to do this is to use NFS:

Procedure A.12. Setting up shared storage

1. Set up an NFS server on a host serving as shared storage. The NFS server can be one of the hosts involved in the migration, as long as all hosts involved are accessing the shared storage through NFS.

   ```
   # mkdir -p /exports/images
   # cat >>/etc/exports <<EOF
   /exports/images    192.168.122.0/24(rw,no_root_squash)
   EOF
   ```

2. Mount the exported directory at a common location on all hosts running **libvirt**. For example, if the IP address of the NFS server is 192.168.122.1, mount the directory with the following commands:

   ```
   # cat >>/etc/fstab <<EOF
   192.168.122.1:/exports/images  /var/lib/libvirt/images  nfs  auto
   0 0
   EOF
   # mount /var/lib/libvirt/images
   ```

NOTE

It is not possible to export a local directory from one host using NFS and mount it at the same path on another host — the directory used for storing disk images must be mounted from shared storage on both hosts. If this is not configured correctly, the guest virtual machine may lose access to its disk images during migration, because the source host's **libvirt** daemon may change the owner, permissions, and SELinux labels on the disk images after it successfully migrates the guest to its destination.

If **libvirt** detects that the disk images are mounted from a shared storage location, it will not make these changes.

A.19.14. No Guest Virtual Machines are Present when libvirtd is Started

Symptom

The **libvirt** daemon is successfully started, but no guest virtual machines appear to be present.

```
# virsh list --all
 Id    Name                           State
----------------------------------------------------
```

Investigation

There are various possible causes of this problem. Performing these tests will help to determine the cause of this situation:

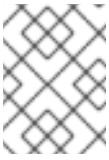
Verify KVM kernel modules

Verify that KVM kernel modules are inserted in the kernel:

```
# lsmod | grep kvm
kvm_intel             121346  0
kvm                   328927  1 kvm_intel
```

If you are using an AMD machine, verify the **kvm_amd** kernel modules are inserted in the kernel instead, using the similar command **lsmod | grep kvm_amd** in the root shell.

If the modules are not present, insert them using the **modprobe *<modulename>*** command.



NOTE

Although it is uncommon, KVM virtualization support may be compiled into the kernel. In this case, modules are not needed.

Verify virtualization extensions

Verify that virtualization extensions are supported and enabled on the host:

```
# egrep "(vmx|svm)" /proc/cpuinfo
flags  : fpu vme de pse tsc ... svm ... skinit wdt npt lbrv svm_lock
nrip_save
```

```
flags  : fpu vme de pse tsc ... svm ... skinit wdt npt lbrv svm_lock
nrip_save
```

Enable virtualization extensions in your hardware's firmware configuration within the BIOS setup. See your hardware documentation for further details on this.

Verify client URI configuration

Verify that the URI of the client is configured as intended:

```
# virsh uri
vbox:///system
```

For example, this message shows the URI is connected to the **VirtualBox** hypervisor, not **QEMU**, and reveals a configuration error for a URI that is otherwise set to connect to a **QEMU** hypervisor. If the URI was correctly connecting to **QEMU**, the same message would appear instead as:

```
# virsh uri
qemu:///system
```

This situation occurs when there are other hypervisors present, which **libvirt** may speak to by default.

Solution

After performing these tests, use the following command to view a list of guest virtual machines:

```
# virsh list --all
```

A.19.15. unable to connect to server at 'host:16509': Connection refused ... error: failed to connect to the hypervisor

Symptom

While **libvirtd** should listen on TCP ports for connections, the connections fail:

```
# virsh -c qemu+tcp://host/system
error: failed to connect to the hypervisor
error: unable to connect to server at 'host:16509': Connection refused
```

The **libvirt** daemon is not listening on TCP ports even after changing configuration in **`/etc/libvirt/libvirtd.conf`**:

```
# grep listen_ /etc/libvirt/libvirtd.conf
listen_tls = 1
listen_tcp = 1
listen_addr = "0.0.0.0"
```

However, the TCP ports for **libvirt** are still not open after changing configuration:

```
# netstat -lntp | grep libvirtd
#
```

Investigation

The **libvirt** daemon was started without the **--listen** option. Verify this by running this command:

```
# ps aux | grep libvirtd
root     10749  0.1  0.2 558276 18280 ?        Ssl  23:21   0:00
/usr/sbin/libvirtd
```

The output does not contain the **--listen** option.

Solution

Start the daemon with the **--listen** option.

To do this, modify the **/etc/sysconfig/libvirtd** file and uncomment the following line:

```
# LIBVIRTD_ARGS="--listen"
```

Then, restart the **libvirtd** service with this command:

```
# /bin/systemctl restart libvirtd.service
```

A.19.16. Common XML Errors

The **libvirt** tool uses XML documents to store structured data. A variety of common errors occur with XML documents when they are passed to **libvirt** through the API. Several common XML errors — including erroneous XML tags, inappropriate values, and missing elements — are detailed below.

A.19.16.1. Editing domain definition

Although it is not recommended, it is sometimes necessary to edit a guest virtual machine's (or a domain's) XML file manually. To access the guest's XML for editing, use the following command:

```
# virsh edit name_of_guest.xml
```

This command opens the file in a text editor with the current definition of the guest virtual machine. After finishing the edits and saving the changes, the XML is reloaded and parsed by **libvirt**. If the XML is correct, the following message is displayed:

```
# virsh edit name_of_guest.xml

Domain name_of_guest.xml XML configuration edited.
```

IMPORTANT

When using the **edit** command in **virsh** to edit an XML document, save all changes before exiting the editor.

After saving the XML file, use the **xmllint** command to validate that the XML is well-formed, or the **virt-xml-validate** command to check for usage problems:

```
# xmllint --noout config.xml
```

```
# virt-xml-validate config.xml
```

If no errors are returned, the XML description is well-formed and matches the **libvirt** schema. While the schema does not catch all constraints, fixing any reported errors will further troubleshooting.

XML documents stored by libvirt

These documents contain definitions of states and configurations for the guests. These documents are automatically generated and should not be edited manually. Errors in these documents contain the file name of the broken document. The file name is valid only on the host machine defined by the URI, which may see the machine the command was run on.

Errors in files created by **libvirt** are rare. However, one possible source of these errors is a downgrade of **libvirt** — while newer versions of **libvirt** can always read XML generated by older versions, older versions of **libvirt** may be confused by XML elements added in a newer version.

A.19.16.2. XML syntax errors

Syntax errors are caught by the XML parser. The error message contains information for identifying the problem.

This example error message from the XML parser consists of three lines — the first line denotes the error message, and the two following lines contain the context and location of the XML code containing the error. The third line contains an indicator showing approximately where the error lies on the line above it:

```
error: (name_of_guest.xml):6: StartTag: invalid element name
<vcpu>2</vcpu><
-----------------^
```

Information contained in this message:

(*name_of_guest.xml*)

This is the file name of the document that contains the error. File names in parentheses are symbolic names to describe XML documents parsed from memory, and do not directly correspond to files on disk. File names that are not contained in parentheses are local files that reside on the target of the connection.

6

This is the line number in the XML file that contains the error.

StartTag: invalid element name

This is the error message from the **libxml2** parser, which describes the specific XML error.

A.19.16.2.1. Stray < in the document

Symptom

The following error occurs:

```
error: (name_of_guest.xml):6: StartTag: invalid element name
<vcpu>2</vcpu><
-----------------^
```

Investigation

This error message shows that the parser expects a new element name after the **<** symbol on line 6 of a guest's XML file.

Ensure line number display is enabled in your text editor. Open the XML file, and locate the text on line 6:

```
<domain type='kvm'>
   <name>name_of_guest</name>
<memory>524288</memory>
<vcpu>2</vcpu><
```

This snippet of a guest's XML file contains an extra **<** in the document:

Solution

Remove the extra **<** or finish the new element.

A.19.16.2.2. Unterminated attribute

Symptom

The following error occurs:

```
error: (name_of_guest.xml):2: Unescaped '<' not allowed in attributes
values
<name>name_of_guest</name>
--^
```

Investigation

This snippet of a guest's XML file contains an unterminated element attribute value:

```
<domain type='kvm>
<name>name_of_guest</name>
```

In this case, **'kvm'** is missing a second quotation mark. Attribute values must be opened and closed with quotation marks or apostrophes, similar to XML start and end tags.

Solution

Correctly open and close all attribute value strings.

A.19.16.2.3. Opening and ending tag mismatch

Symptom

The following error occurs:

```
error: (name_of_guest.xml):61: Opening and ending tag mismatch: clock
line 16 and domain
</domain>
---------^
```

Investigation

The error message above contains three clues to identify the offending tag:

The message following the last colon, **clock line 16 and domain**, reveals that **<clock>** contains a mismatched tag on line 16 of the document. The last hint is the pointer in the context part of the message, which identifies the second offending tag.

Unpaired tags must be closed with **/>**. The following snippet does not follow this rule and has produced the error message shown above:

```
<domain type='kvm'>
  ...
    <clock offset='utc'>
```

This error is caused by mismatched XML tags in the file. Every XML tag must have a matching start and end tag.

Other examples of mismatched XML tags

The following examples produce similar error messages and show variations of mismatched XML tags.

This snippet contains an mismatch error for **<features>** because there is no end tag (**</name>**):

```
<domain type='kvm'>
 ...
 <features>
   <acpi/>
   <pae/>
 ...
 </domain>
```

This snippet contains an end tag (**</name>**) without a corresponding start tag:

```
<domain type='kvm'>
  </name>
  ...
</domain>
```

Solution

Ensure all XML tags start and end correctly.

A.19.16.2.4. Typographical errors in tags

Symptom

The following error message appears:

```
error: (name_of_guest.xml):1: Specification mandate value for attribute
ty
<domain ty pe='kvm'>
-----------^
```

Investigation

XML errors are easily caused by a simple typographical error. This error message highlights the XML error — in this case, an extra white space within the word **type** — with a pointer.

```
<domain ty pe='kvm'>
```

These XML examples will not parse correctly because of typographical errors such as a missing special character, or an additional character:

```
<domain type 'kvm'>
```

```
<dom#ain type='kvm'>
```

Solution

To identify the problematic tag, read the error message for the context of the file, and locate the error with the pointer. Correct the XML and save the changes.

A.19.16.3. Logic and configuration errors

A well-formatted XML document can contain errors that are correct in syntax but **libvirt** cannot parse. Many of these errors exist, with two of the most common cases outlined below.

A.19.16.3.1. Vanishing parts

Symptom

Parts of the change you have made do not show up and have no effect after editing or defining the domain. The **define** or **edit** command works, but when dumping the XML once again, the change disappears.

Investigation

This error likely results from a broken construct or syntax that libvirt does not parse. The **libvirt** tool will generally only look for constructs it knows but ignore everything else, resulting in some of the XML changes vanishing after **libvirt** parses the input.

Solution

Validate the XML input before passing it to the **edit** or **define** commands. The **libvirt** developers maintain a set of XML schemas bundled with **libvirt** that define the majority of the constructs allowed in XML documents used by **libvirt**.

Validate **libvirt** XML files using the following command:

```
# virt-xml-validate libvirt.xml
```

If this command passes, **libvirt** will likely understand all constructs from your XML, except if the schemas cannot detect options that are valid only for a given hypervisor. For example, any XML generated by **libvirt** as a result of a **`virsh dump`** command should validate without error.

A.19.16.3.2. Incorrect drive device type

Symptom

The definition of the source image for the CD-ROM virtual drive is not present, despite being added:

```
# virsh dumpxml domain
<domain type='kvm'>
  ...
  <disk type='block' device='cdrom'>
    <driver name='qemu' type='raw'/>
    <target dev='hdc' bus='ide'/>
    <readonly/>
  </disk>
  ...
</domain>
```

Solution

Correct the XML by adding the missing **`<source>`** parameter as follows:

```
<disk type='block' device='cdrom'>
  <driver name='qemu' type='raw'/>
  <source file='/path/to/image.iso'/>
  <target dev='hdc' bus='ide'/>
  <readonly/>
</disk>
```

A **`type='block'`** disk device expects that the source is a physical device. To use the disk with an image file, use **`type='file'`** instead.

APPENDIX B. USING KVM VIRTUALIZATION ON MULTIPLE ARCHITECTURES

By default, KVM virtualization on Red Hat Enterprise Linux 7 is compatible with the AMD64 and Intel 64 architectures. However, starting with Red Hat Enterprise Linux 7.5, KVM virtualization is also supported on the following architectures, thanks to the introduction of the kernel-alt packages:

- IBM POWER
- IBM Z
- ARM systems (Development Preview only)

Note that when using virtualization on these architectures, the installation, usage, and feature support differ from AMD64 and Intel 64 in certain respects. For more information, see the sections below:

B.1. USING KVM VIRTUALIZATION ON IBM POWER SYSTEMS

Starting with Red Hat Enterprise Linux 7.5, KVM virtualization is supported on IBM POWER8 Systems and IBM POWER9 systems. However, IBM POWER8 does not use kernel-alt, which means that these two architectures differ in certain aspects.

Installation

To install KVM virtualization on Red Hat Enterprise Linux 7 for IBM POWER 8 and POWER9 Systems:

1. Install the host system from the bootable image on the Customer Portal:
   - IBM POWER8
   - IBM POWER9

   For detailed instructions, see the Red Hat Enterprise Linux 7 Installation Guide.
2. Ensure that your host system meets the hypervisor requirements:
   - Verify that you have the correct machine type:

     ```
     # grep ^platform /proc/cpuinfo
     ```

     The output of this command must include the **PowerNV** entry, which indicates that you are running on a supported PowerNV machine type:

     ```
     platform        : PowerNV
     ```

   - Load the KVM-HV kernel module:

     ```
     # modprobe kvm_hv
     ```

   - Verify that the KVM-HV kernel module is loaded:

     ```
     # lsmod | grep kvm
     ```

     If KVM-HV was loaded successfully, the output of this command includes **kvm_hv**.

3. Install the qemu-kvm-ma package in addition to other virtualization packages described in Chapter 2, *Installing the Virtualization Packages*.

Architecture Specifics

KVM virtualization on Red Hat Enterprise Linux 7.5 for IBM POWER differs from KVM on AMD64 and Intel 64 systems in the following:

- The recommended **minimum memory** allocation for a guest on an IBM POWER host is **2GB RAM**.
- The SPICE protocol is not supported on IBM POWER Systems. To display the graphical output of a guest, use the VNC protocol. In addition, only the following virtual graphics card devices are supported:
  - **`vga`** - only supported in **`-vga std`** mode and not in **`-vga cirrus`** mode
  - **`virtio-vga`**
  - **`virtio-gpu`**
- The following virtualization features are disabled on AMD64 and Intel 64 hosts, but work on IBM POWER. However, they are not supported by Red Hat, and therefore not recommended for use:
  - I/O threads
- SMBIOS configuration is not available.
- POWER8 guests, including compatibility mode guests, may fail to start with an error similar to:

```
qemu-kvm: Failed to allocate KVM HPT of order 33 (try smaller
maxmem?): Cannot allocate memory
```

  This is significantly more likely to occur on guests that use Red Hat Enterprise Linux 7.3 or prior.

  To fix this problem, do one of the following:

  - Increase the CMA memory pool available for the guest's hashed page table (HPT) by adding **`kvm_cma_resv_ratio=`*`memory`*** to the host's kernel command line, where *memory* is the percentage of host memory that should be reserved for the CMA pool (defaults to 5).
  - Reduce the maximum hot-plugged memory for the guest by using the **`virsh edit `*`guestname`*** command and lowering the value of the **`<maxMemory>`** element.
  - If the guest allows for memory hot plug, update the guest to Red Hat Enterprise Linux 7.4 or later.

    To ensure that memory hot plug is enabled for the guest, search for the <maxMemory> element and NUMA topology setting in the guest's XML configuration:

```
# virsh dumpxml guestname | grep maxMemory
<maxMemory slots='16' unit='KiB'>16777216</maxMemory>
```

```
# virsh dumpxml guestname | grep cell
<cell id='0' cpus='0-2' memory='1048576' unit='KiB'/>
```

If these elements are not present in the XML configuration, updating to Red Hat Enterprise Linux 7.4 will not fix the problem.

- Transparent huge pages (THPs) currently do not provide any notable performance benefits on IBM POWER8 guests

  Also note that the sizes of static huge pages on IBM POWER8 systems are 16MiB and 16GiB, as opposed to 2MiB and 1GiB on AMD 64 and Intel64 and on IBM POWER9. As a consequence, migrating a guest from an IBM POWER8 host to an IBM POWER9 host fails if the guest is configured with static huge pages.

  In addition, to be able to use static huge pages or THPs on IBM POWER8 guests, you must first set up huge pages on the host.

- A number of virtual peripheral devices that are supported on AMD64 and Intel 64 systems are not supported on IBM POWER systems, or a different device is supported as a replacement:
  - Devices used for PCI-E hierarchy, including the **`ioh3420`** and **`xio3130-downstream`** devices, are not supported. This functionality is replaced by multiple independent PCI root bridges, provided by the **`spapr-pci-host-bridge`** device.
  - UHCI and EHCI PCI controllers are not supported. Use OHCI and XHCI controllers instead.
  - IDE devices, including the virtual IDE CD-ROM (**`ide-cd`**) and the virtual IDE disk (**`ide-hd`**), are not supported. Use the **`virtio-scsi`** and **`virtio-blk`** devices instead.
  - Emulated PCI NICs (**`rtl8139`**) are not supported. Use the **`virtio-net`** device instead.
  - Sound devices, including **`intel-hda`**, **`hda-output`**, and **`AC97`**, are not supported.
  - USB redirection devices, including **`usb-redir`** and **`usb-tablet`**, are not supported.
- The **`kvm-clock`** service does not have to be configured for time management on IBM POWER systems.
- The **`pvpanic`** device is not supported on IBM POWER systems. However, an equivalent functionality is available and activated on this architecture by default. To enable it on a guest, use the **`<on_crash>`** configuration element with the **`preserve`** value. In addition, make sure to remove the **`<panic>`** element from the **`<devices>`** section, as its presence can lead to the guest failing to boot on IBM POWER systems.
- On POWER8 systems, the host machine must run in single-threaded mode to support guests. This is automatically configured if the qemu-kvm-ma packages are installed. However, guests running on a single-threaded hosts can still use multiple threads.

B.2. USING KVM VIRTUALIZATION ON IBM Z

Installation

On IBM Z hosts, the KVM hypervisor has to be installed in a dedicated logical partition (LPAR). Running KVM on the z/VM OS is not supported. The LPAR also has to support the so-called *start-interpretive execution* (SIE) virtualization extensions.

To install KVM Virtualization on Red Hat Enterprise Linux 7 for IBM Z:

1. Install the system from the bootable image on the Customer Portal - for detailed instructions, see the Installation guide.

2. Ensure that your system meets the hypervisor requirements:

   - Verify that the CPU virtualization extensions are available:

     ```
     # grep sie /proc/cpuinfo
     ```

     The output of this command must include the **sie** entry, which indicates that your processor has the required virtualization extension.

     ```
     features        : esan3 zarch stfle msa ldisp eimm dfp edat
     etf3eh highgprs te sie
     ```

   - Load the KVM kernel module:

     ```
     # modprobe kvm
     ```

   - Verify that the KVM kernel module is loaded:

     ```
     # lsmod | grep kvm
     ```

     If KVM was loaded successfully, the output of this command includes **kvm**. If it does not, make sure that you are using the kernel-alt version of the kernel for Red Hat Enterprise Linux 7.

3. Install the qemu-kvm-ma package in addition to other virtualization packages described in Chapter 2, *Installing the Virtualization Packages*.

4. When setting up guests, it is recommended to configure their CPU in one of the following ways to protect the guests from the "Spectre" vulnerability:

   - Use the host CPU model, for example as follows:

     ```
     <cpu mode='host-model' check='partial'>
       <model fallback='allow'/>
     </cpu>
     ```

     This makes the **ppa15** and **bpb** features available to the guest if the host supports them.

   - If using a specific host model, add the **ppa15** and **bpb** features. The following example uses the **zEC12** CPU model:

     ```
     <cpu mode='custom' match='exact' check='partial'>
         <model fallback='allow'>zEC12</model>
         <feature policy='force' name='ppa15'/>
         <feature policy='force' name='bpb'/>
     </cpu>
     ```

NOTE

When using the **ppa15** feature with the **z114** and **z196** CPU models on a **z12** host machine, make sure to use the latest microcode level (bundle 95 or later).

Architecture Specifics

KVM Virtualization on Red Hat Enterprise Linux 7.5 for IBM Z differs from KVM on AMD64 and Intel 64 systems in the following:

- The SPICE and VNC protocols are not available and virtual graphical card devices are not supported on IBM Z. Therefore, displaying the guest graphical output is not possible.
- Virtual PCI and USB devices are not supported on IBM Z. This also means that *virtio-*-pci* devices are unsupported, and ***virtio-*-ccw*** devices should be used instead. For example, use **virtio-net-ccw** instead of **virtio-net-pci**.
- The <boot dev='*device*'/> XML configuration element is not supported on z Systems. To define device boot order, use the <boot order='*number*'/> in the <devices> section. For an example, see the upstream libvirt documentation.

  NOTE

  Using <boot order='*number*'/> for boot order management is preferred also on AMD64 and Intel 64 hosts.

- SMBIOS configuration is not available.
- The watchdog device model used on IBM Z should be **diag288**.
- To enable nested virtualization, do the following. Note that like on AMD64 and Intel 64 systems, the nested virtualization feature is available as a Technology Preview on IBM Z, and therefore is not recommended for use in production environments.

  1. 1. Check whether nested virtualization is already enabled on your system:

     ```
     $ cat /sys/module/kvm/parameters/nested
     ```

     If this command returns **1**, the feature is already enabled.

     If the command returns **0**, use the following steps to enable it.

  2. 2. Unload the **kvm** module:

     ```
     # modprobe -r kvm
     ```

  3. 3. Activate the nesting feature:

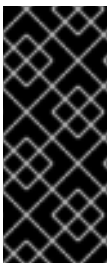
     ```
     # modprobe kvm nested=1
     ```

  4. 4. The nesting feature is now enabled only until the next reboot of the host. To enable it permanently, add the following line to the **/etc/modprobe.d/kvm.conf** file:

     ```
     options kvm nested=1
     ```

- The **kvm-clock** service is specific to AMD64 and Intel 64 systems, and does not have to be configured for time management on IBM Z.

B.3. USING KVM VIRTUALIZATION ON ARM SYSTEMS



IMPORTANT

KVM virtualization is provided as a Development Preview in Red Hat Enterprise Linux 7.5 for the 64-bit ARM architecture. As such, KVM virtualization on ARM systems is not supported, not intended for use in a production environment, and may not address known security vulnerabilities. In addition, because KVM virtualization on ARM is still in rapid development, the information below is not guaranteed to be accurate or complete.

Installation

To use install virtualization on Red Hat Enterprise Linux 7.5 for ARM:

1. Install the system from the bootable image on the Customer Portal.
2. After the system is installed, install the virtualization stack on the system by using the following command:

   ```
   # yum install qemu-kvm-ma libvirt libvirt-client virt-install AAVMF
   ```

   Make sure you have the **`Optional`** channel enabled for the installation to succeed.

Architecture Specifics

KVM virtualization on Red Hat Enterprise Linux 7.5 for the 64-bit ARM architecture differs from KVM on AMD64 and Intel 64 systems in the following:

- PXE booting is only supported with the **`virtio-net-device`** and **`virtio-net-pci`** network interface controllers (NICs). In addition, the built-in **`VirtioNetDxe`** driver of the ARM Architecture Virtual Machine Firmware (AAVMF) needs to be used for PXE booting. Note that iPXE option ROMs are not supported.
- Only up to 123 virtual CPUs (vCPUs) can be allocated to a single guest.

APPENDIX C. VIRTUALIZATION RESTRICTIONS

This appendix covers additional support and product restrictions of the virtualization packages in Red Hat Enterprise Linux 7.

C.1. SYSTEM RESTRICTIONS

Host Systems

Red Hat Enterprise Linux with KVM is supported only on the following host architectures:

- AMD64 and Intel 64
- IBM Z
- IBM POWER8
- IBM POWER9

This document primarily describes AMD64 and Intel64 features and functionalities, but the other supported architectures work very similarly. For details, see Appendix B, *Using KVM Virtualization on Multiple Architectures*.

Guest Systems

On Red Hat Enterprise Linux 7, Microsoft Windows guest virtual machines are only supported under specific subscription programs such as Advanced Mission Critical (AMC). If you are unsure whether your subscription model includes support for Windows guests, contact customer support.

For more information about Windows guest virtual machines on Red Hat Enterprise Linux 7, see Windows Guest Virtual Machines on Red Hat Enterprise Linux 7 Knowledgebase article.

C.2. FEATURE RESTRICTIONS

The hypervisor package included with Red Hat Enterprise Linux is qemu-kvm. This differs from the qemu-kvm-rhev package included with Red Hat Virtualization (RHV) and Red Hat OpenStack (RHOS) products. Many of the restrictions that apply to qemu-kvm do not apply to qemu-kvm-rhev.

For more information about the differences between the qemu-kvm and qemu-kvm-rhev packages, see What are the differences between qemu-kvm and qemu-kvm-rhev and all sub-packages?

The following restrictions apply to the KVM hypervisor included with Red Hat Enterprise Linux:

Maximum vCPUs per guest

Red Hat Enterprise Linux 7.2 and above supports 240 vCPUs per guest, up from 160 in Red Hat Enterprise Linux 7.0.

Nested virtualization

Nested virtualization is available as a Technology Preview in Red Hat Enterprise Linux 7.2 and later. This feature enables KVM to launch guests that can act as hypervisors and create their own guests.

TCG support

QEMU and **libvirt** include a dynamic translation mode using the QEMU Tiny Code Generator (TCG). This mode does not require hardware virtualization support. However, TCG is not supported by Red Hat.

When the qemu-kvm package is used to create nested guests in a virtual machine, it uses TCG unless nested virtualization is enabled on the parent virtual machine. Note that nested virtualization is currently a Technology Preview. For more information, see Chapter 12, *Nested Virtualization*.

A TCG-based guest can be recognized using the following:

- The domain XML file of the guests contains the **`<domain type='qemu'>`** line, whereas a KVM guest contains **`<domain type='kvm'>`**.
- In the Overview pane of the Virtual hardware details view, **virt-manager** displays the type of virtual machine as **`QEMU TCG`**, instead of **`KVM`**.

Constant TSC bit

Systems without a Constant Time Stamp Counter (TSC) require additional configuration. See Chapter 8, *KVM Guest Timing Management* for details on determining whether you have a Constant Time Stamp Counter and configuration steps for fixing any related issues.

Emulated SCSI adapters

SCSI device emulation is only supported with the virtio-scsi paravirtualized host bus adapter (HBA). Emulated SCSI HBAs are not supported with KVM in Red Hat Enterprise Linux.

Emulated IDE devices

KVM is limited to a maximum of four virtualized (emulated) IDE devices per virtual machine.

Paravirtualized devices

Paravirtualized devices are also known as VirtIO devices. They are purely virtual devices designed to work optimally in a virtual machine.

Red Hat Enterprise Linux 7 supports 32 PCI device slots per virtual machine bus, and 8 PCI functions per device slot. This gives a theoretical maximum of 256 PCI functions per bus when multi-function capabilities are enabled in the virtual machine, and PCI bridges are used. Each PCI bridge adds a new bus, potentially enabling another 256 device addresses. However, some buses do not make all 256 device addresses available for the user; for example, the root bus has several built-in devices occupying slots.

See Chapter 16, *Guest Virtual Machine Device Configuration* for more information on devices and Section 16.1.5, “Creating PCI Bridges” for more information on PCI bridges.

Migration restrictions

Device assignment refers to physical devices that have been exposed to a virtual machine, for the exclusive use of that virtual machine. Because device assignment uses hardware on the specific host where the virtual machine runs, migration and save/restore are not supported when device assignment is in use. If the guest operating system supports hot plugging, assigned devices can be removed prior to the migration or save/restore operation to enable this feature.

Live migration is only possible between hosts with the same CPU type (that is, Intel to Intel or AMD to AMD only).

For live migration, both hosts must have the same value set for the No eXecution (NX) bit, either **on** or **off**.

For migration to work, **cache=none** must be specified for all block devices opened in write mode.

WARNING

Failing to include the **cache=none** option can result in disk corruption.

Storage restrictions

There are risks associated with giving guest virtual machines write access to entire disks or block devices (such as **/dev/sdb**). If a guest virtual machine has access to an entire block device, it can share any volume label or partition table with the host machine. If bugs exist in the host system's partition recognition code, this can create a security risk. Avoid this risk by configuring the host machine to ignore devices assigned to a guest virtual machine.

WARNING

Failing to adhere to storage restrictions can result in risks to security.

Live snapshots

The backup and restore API in KVM in Red Hat Enterprise Linux does not support live snapshots.

Streaming, mirroring, and live-merge

Streaming, mirroring, and live-merge are not supported. This prevents block-jobs.

I/O throttling

Red Hat Enterprise Linux does not support configuration of maximum input and output levels for operations on virtual disks.

I/O threads

Red Hat Enterprise Linux does not support creation of separate threads for input and output operations on disks with VirtIO interfaces.

Memory hot plug and hot unplug

Red Hat Enterprise Linux does not support hot plugging or hot unplugging memory from a virtual machine.

vhost-user

Red Hat Enterprise Linux does not support implementation of a user space vhost interface.

CPU hot unplug

Red Hat Enterprise Linux does not support hot-unplugging CPUs from a virtual machine.

NUMA guest locality for PCIe

Red Hat Enterprise Linux does not support binding a virtual PCIe device to a specific NUMA node.

Core dumping restrictions

Because core dumping is currently implemented on top of migration, it is not supported when device assignment is in use.

Realtime kernel

KVM currently does not support the realtime kernel, and thus cannot be used on Red Hat Enterprise Linux for Real Time.

C.3. APPLICATION RESTRICTIONS

There are aspects of virtualization that make it unsuitable for certain types of applications.

Applications with high I/O throughput requirements should use KVM's paravirtualized drivers (virtio drivers) for fully-virtualized guests. Without the virtio drivers, certain applications may be unpredictable under heavy I/O loads.

The following applications should be avoided due to high I/O requirements:

- **kdump** server
- **netdump** server

You should carefully evaluate applications and tools that heavily utilize I/O or those that require real-time performance. Consider the virtio drivers or PCI device assignment for increased I/O performance. For more information on the virtio drivers for fully virtualized guests, see Chapter 5, *KVM Paravirtualized (virtio) Drivers*. For more information on PCI device assignment, see Chapter 16, *Guest Virtual Machine Device Configuration*.

Applications suffer a small performance loss from running in virtualized environments. The performance benefits of virtualization through consolidating to newer and faster hardware should be evaluated against the potential application performance issues associated with using virtualization.

C.4. OTHER RESTRICTIONS

For the list of all other restrictions and issues affecting virtualization read the *Red Hat Enterprise Linux 7 Release Notes*. The *Red Hat Enterprise Linux 7 Release Notes* cover the present new features, known issues, and restrictions as they are updated or discovered.

C.5. STORAGE SUPPORT

The following storage methods are supported for virtual machines:

- Files on local storage
- Physical disk partitions

- Locally connected physical LUNs
- LVM partitions
- NFS shared file systems
- iSCSI
- GFS2 clustered file systems
- Fibre Channel-based LUNs
- Fibre Channel over Ethernet (FCoE)

C.6. USB 3 / XHCI SUPPORT

USB 3 (xHCI) USB host adapter emulation is supported in Red Hat Enterprise Linux 7.2 and above. All USB speeds are available, meaning any generation of USB device can be plugged into a xHCI bus. Additionally, no companion controllers (for USB 1 devices) are required. Note, however, that USB 3 bulk streams are not supported.

Advantages of xHCI:

- Virtualization-compatible hardware design, meaning xHCI emulation requires less CPU resources than previous versions due to reduced polling overhead.
- USB passthrough of USB 3 devices is available.

Limitations of xHCI:

- Not supported for Red Hat Enterprise Linux 5 guests.

See Figure 16.19, “Domain XML example for USB3/xHCI devices” for a domain XML device example for xHCI devices.

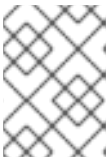
APPENDIX D. ADDITIONAL RESOURCES

To learn more about virtualization and Red Hat Enterprise Linux, see the following resources.

D.1. ONLINE RESOURCES

- http://www.libvirt.org/ is the official upstream website for the **libvirt** virtualization API.
- https://virt-manager.org/ is the upstream project website for the **Virtual Machine Manager** (virt-manager), the graphical application for managing virtual machines.
- Red Hat Virtualization - http://www.redhat.com/products/cloud-computing/virtualization/
- Red Hat product documentation - https://access.redhat.com/documentation/en/
- Virtualization technologies overview - http://virt.kernelnewbies.org

D.2. INSTALLED DOCUMENTATION

- **man virsh** and **/usr/share/doc/libvirt-*version-number*** — Contains sub-commands and options for the **virsh** virtual machine management utility as well as comprehensive information about the **libvirt** virtualization library API.
- **/usr/share/doc/gnome-applet-vm-*version-number*** — Documentation for the GNOME graphical panel applet that monitors and manages locally-running virtual machines.
- **/usr/share/doc/libvirt-python-*version-number*** — Provides details on the Python bindings for the **libvirt** library. The **libvirt-python** package allows python developers to create programs that interface with the **libvirt** virtualization management library.
- **/usr/share/doc/virt-install-*version-number*** — Provides documentation on the **virt-install** command that helps in starting installations of Fedora and Red Hat Enterprise Linux related distributions inside of virtual machines.
- **/usr/share/doc/virt-manager-*version-number*** — Provides documentation on the Virtual Machine Manager, which provides a graphical tool for administering virtual machines.
- 

  NOTE

  For more information about other Red Hat Enterprise Linux components, see the appropriate **man** page or file in **usr/share/doc/**.

APPENDIX E. WORKING WITH IOMMU GROUPS[1]

Introduced in Red Hat Enterprise Linux 7, *VFIO*, or Virtual Function I/O, is a set of Linux kernel modules that provide a user-space driver framework. This framework uses input–output memory management unit (IOMMU) protection to enable secure device access for user-space drivers. VFIO enables user-space drivers such as the *Data Plane Development Kit (DPDK)*, as well as the more common PCI device assignment.

VFIO uses IOMMU groups to isolate devices and prevent unintentional *Direct Memory Access (DMA)* between two devices running on the same host physical machine, which would impact host and guest functionality. IOMMU groups are available in Red Hat Enterprise Linux 7, which is a big improvement over the legacy KVM device assignment that is available in Red Hat Enterprise Linux 6. This appendix highlights the following:

- An overview of IOMMU groups
- The importance of device isolation
- VFIO benefits

E.1. IOMMU OVERVIEW

An IOMMU creates a virtual address space for the device, where each I/O Virtual Address (IOVA) may translate to different addresses in the physical system memory. When the translation is completed, the devices are connected to a different address within the physical system's memory. Without an IOMMU, all devices have a shared, flat view of the physical memory because they lack memory address translation. With an IOMMU, devices receive the IOVA space as a new address space, which is useful for device assignment.

Different IOMMUs have different levels of functionality. In the past, IOMMUs were limited, providing only translation, and often only for a small window of the address space. For example, the IOMMU would only reserve a small window (1GB or less) of IOVA space in low memory, which was shared by multiple devices. The AMD graphics address remapping table (GART), when used as a general-purpose IOMMU, is an example of this model. These classic IOMMUs mostly provided two capabilities: *bounce buffers* and *address coalescing*.

- *Bounce buffers* are necessary when the addressing capabilities of the device are less than that of the platform. For example, if a device's address space is limited to 4GB (32 bits) of memory and the driver was to allocate to a buffer above 4GB, the device would not be able to directly access the buffer. Such a situation necessitates using a bounce buffer; a buffer space located in lower memory, where the device can perform a DMA operation. The data in the buffer is only copied to the driver's allocated buffer on completion of the operation. In other words, the buffer is bounced from a lower memory address to a higher memory address. IOMMUs avoid bounce buffering by providing an IOVA translation within the device's address space. This allows the device to perform a DMA operation directly into the buffer, even when the buffer extends beyond the physical address space of the device. Historically, this IOMMU feature was often the exclusive use case for the IOMMU, but with the adoption of *PCI-Express* (PCIe), the ability to support addressing above 4GB is required for all non-legacy endpoints.

- In traditional memory allocation, blocks of memory are assigned and freed based on the needs of the application. Using this method creates memory gaps scattered throughout the physical address space. It would be better if the memory gaps were *coalesced* so they can be used more efficiently, or in basic terms it would be better if the memory gaps were gathered together. The IOMMU coalesces these scattered memory lists through the IOVA space, sometimes referred to as scatter-gather lists. In doing so the IOMMU creates contiguous DMA operations and ultimately increases the efficiency of the I/O performance. In the simplest example, a driver may

allocate two 4KB buffers that are not contiguous in the physical memory space. The IOMMU can allocate a contiguous range for these buffers allowing the I/O device to do a single 8KB DMA rather than two separate 4KB DMAs.

Although memory coalescing and bounce buffering are important for high performance I/O on the host, the IOMMU feature that is essential for a virtualization environment is the *isolation capability* of modern IOMMUs. Isolation was not possible on a wide scale prior to PCI-Express, because conventional PCI does not tag transactions with an ID of the requesting device (requester ID). Even though PCI-X included some degree of a requester ID, the rules for interconnecting devices that take ownership of the transaction did not provide complete support for device isolation.

With PCIe, each device's transaction is tagged with a requester ID unique to the device (the PCI bus/device/function number, often abbreviated as BDF), which is used to reference a unique IOVA table for that device. Now that isolation is possible, the IOVA space cannot only be used for translation operations such as offloading unreachable memory and coalescing memory, but it can also be used to restrict DMA access from the device. This allows devices to be isolated from each other, preventing duplicate assignment of memory spaces, which is essential for proper guest virtual machine device management. Using these features on a guest virtual machine involves populating the IOVA space for the assigned device with the guest-physical-to-host-physical memory mappings for the virtual machine. Once this is done, the device transparently performs DMA operations in the guest virtual machine's address space.

E.2. A DEEP-DIVE INTO IOMMU GROUPS

An IOMMU group is defined as the smallest set of devices that can be considered isolated from the IOMMU's perspective. The first step to achieve isolation is granularity. If the IOMMU cannot differentiate devices into separate IOVA spaces, they are not isolated. For example, if multiple devices attempt to alias to the same IOVA space, the IOMMU is not able to distinguish between them. This is the reason why a typical x86 PC will group all conventional-PCI devices together, with all of them aliased to the same requester ID, the PCIe-to-PCI bridge. Legacy KVM device assignment allows a user to assign these conventional-PCI devices separately, but the configuration fails because the IOMMU cannot distinguish between the devices. As VFIO is governed by IOMMU groups, it prevents any configuration that violates this most basic requirement of IOMMU granularity.

The next step is to determine whether the transactions from the device actually reach the IOMMU. The PCIe specification allows for transactions to be re-routed within the interconnect fabric. A PCIe downstream port can re-route a transaction from one downstream device to another. The downstream ports of a PCIe switch may be interconnected to allow re-routing from one port to another. Even within a multifunction endpoint device, a transaction from one function may be delivered directly to another function. These transactions from one device to another are called peer-to-peer transactions and can destroy the isolation of devices operating in separate IOVA spaces. Imagine for instance, if the network interface card assigned to a guest virtual machine, attempts a DMA write operation to a virtual address within its own IOVA space. However in the physical space, that same address belongs to a peer disk controller owned by the host. As the IOVA to physical translation for the device is only performed at the IOMMU, any interconnect attempting to optimize the data path of that transaction could mistakenly redirect the DMA write operation to the disk controller before it gets to the IOMMU for translation.

To solve this problem, the PCI Express specification includes support for PCIe Access Control Services (ACS), which provides visibility and control of these redirects. This is an essential component for isolating devices from one another, which is often missing in interconnects and multifunction endpoints. Without ACS support at every level from the device to the IOMMU, it must be assumed that redirection is possible. This will, therefore, break the isolation of all devices below the point lacking ACS support in the PCI topology. IOMMU groups in a PCI environment take this isolation into account, grouping together devices which are capable of untranslated peer-to-peer DMA.

In summary, the IOMMU group represents the smallest set of devices for which the IOMMU has visibility

and which is isolated from other groups. VFIO uses this information to enforce safe ownership of devices for user space. With the exception of bridges, root ports, and switches (all examples of interconnect fabric), all devices within an IOMMU group must be bound to a VFIO device driver or known safe stub driver. For PCI, these drivers are vfio-pci and pci-stub. pci-stub is allowed simply because it is known that the host does not interact with devices via this driver[2]. If an error occurs indicating the group is not viable when using VFIO, it means that all of the devices in the group need to be bound to an appropriate host driver. Using **`virsh nodedev-dumpxml`** to explore the composition of an IOMMU group and **`virsh nodedev-detach`** to bind devices to VFIO compatible drivers, will help resolve such problems.

E.3. HOW TO IDENTIFY AND ASSIGN IOMMU GROUPS

This example demonstrates how to identify and assign the PCI devices that are present on the target system. For additional examples and information, see Section 16.7, “Assigning GPU Devices”.

Procedure E.1. IOMMU groups

1. **List the devices**
   Identify the devices in your system by running the **`virsh nodev-list`** ***`device-type`*** command. This example demonstrates how to locate the PCI devices. The output has been truncated for brevity.

   ```
   # virsh nodedev-list pci

   pci_0000_00_00_0
   pci_0000_00_01_0
   pci_0000_00_03_0
   pci_0000_00_07_0
   [...]
   pci_0000_00_1c_0
   pci_0000_00_1c_4
   [...]
   pci_0000_01_00_0
   pci_0000_01_00_1
   [...]
   pci_0000_03_00_0
   pci_0000_03_00_1
   pci_0000_04_00_0
   pci_0000_05_00_0
   pci_0000_06_0d_0
   ```

2. **Locate the IOMMU grouping of a device**
   For each device listed, further information about the device, including the IOMMU grouping, can be found using the **`virsh nodedev-dumpxml`** ***`name-of-device`*** command. For example, to find the IOMMU grouping for the PCI device named pci_0000_04_00_0 (PCI address 0000:04:00.0), use the following command:

   ```
   # virsh nodedev-dumpxml pci_0000_04_00_0
   ```

   This command generates a XML dump similar to the one shown.

```
<device>
  <name>pci_0000_04_00_0</name>
  <path>/sys/devices/pci0000:00/0000:00:1c.0/0000:04:00.0</path>
  <parent>pci_0000_00_1c_0</parent>
  <capability type='pci'>
    <domain>0</domain>
    <bus>4</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x10d3'>82574L Gigabit Network Connection</product>
    <vendor id='0x8086'>Intel Corporation</vendor>
    <iommuGroup number='8'>   <!--This is the element block you
will need to use-->
      <address domain='0x0000' bus='0x00' slot='0x1c'
function='0x0'/>
      <address domain='0x0000' bus='0x00' slot='0x1c'
function='0x4'/>
      <address domain='0x0000' bus='0x04' slot='0x00'
function='0x0'/>
      <address domain='0x0000' bus='0x05' slot='0x00'
function='0x0'/>
    </iommuGroup>
    <pci-express>
      <link validity='cap' port='0' speed='2.5' width='1'/>
      <link validity='sta' speed='2.5' width='1'/>
    </pci-express>
  </capability>
</device>
```

Figure E.1. IOMMU Group XML

3. **View the PCI data**
   In the output collected above, there is one IOMMU group with 4 devices. This is an example of a multi-function PCIe root port without ACS support. The two functions in slot 0x1c are PCIe root ports, which can be identified by running the **`lspci`** command (from the pciutils package):

```
# lspci -s 1c

00:1c.0 PCI bridge: Intel Corporation 82801JI (ICH10 Family) PCI
Express Root Port 1
00:1c.4 PCI bridge: Intel Corporation 82801JI (ICH10 Family) PCI
Express Root Port 5
```

   Repeat this step for the two PCIe devices on buses 0x04 and 0x05, which are endpoint devices.

```
# lspci -s 4
04:00.0 Ethernet controller: Intel Corporation 82574L Gigabit
Network Connection This is used in the next step and is called
04:00.0
# lspci -s 5 This is used in the next step and is called 05:00.0
05:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5755
Gigabit Ethernet PCI Express (rev 02)
```

4. **Assign the endpoints to the guest virtual machine**
   In order to assign either one of the endpoints to a virtual machine, the endpoint which you are not assigning at the moment, must be bound to a VFIO compatible driver so that the IOMMU group is not split between user and host drivers. If for example, using the output received above, you were to configuring a virtual machine with only 04:00.0, the virtual machine will fail to start unless 05:00.0 is detached from host drivers. To detach 05:00.0, run the **`virsh nodedev-detach`** command as root:

   ```
   # virsh nodedev-detach pci_0000_05_00_0
   Device pci_0000_05_00_0 detached
   ```

   Assigning both endpoints to the virtual machine is another option for resolving this issue. Note that libvirt will automatically perform this operation for the attached devices when using the yes value for the **`managed`** attribute within the **`<hostdev>`** element. For example: **`<hostdev mode='subsystem' type='pci' managed='yes'>`**. See Note for more information.

NOTE

libvirt has two ways to handle PCI devices. They can be either managed or unmanaged. This is determined by the value given to the **`managed`** attribute within the **`<hostdev>`** element. When the device is managed, libvirt automatically detaches the device from the existing driver and then assigns it to the virtual machine by binding it to vfio-pci on boot (for the virtual machine). When the virtual machine is shutdown or deleted or the PCI device is detached from the virtual machine, libvirt unbinds the device from vfio-pci and rebinds it to the original driver. If the device is unmanaged, libvirt will not automate the process and you will have to ensure all of these management aspects as described are done before assigning the device to a virtual machine, and after the device is no longer used by the virtual machine you will have to reassign the devices as well. Failure to do these actions in an unmanaged device will cause the virtual machine to fail. Therefore, it may be easier to make sure that libvirt manages the device.

E.4. IOMMU STRATEGIES AND USE CASES

There are many ways to handle IOMMU groups that contain more devices than intended. For a plug-in card, the first option would be to determine whether installing the card into a different slot produces the intended grouping. On a typical Intel chipset, PCIe root ports are provided via both the processor and the Platform Controller Hub (PCH). The capabilities of these root ports can be very different. Red Hat Enterprise Linux 7 has support for exposing the isolation of numerous PCH root ports, even though many of them do not have native PCIe ACS support. Therefore, these root ports are good targets for creating smaller IOMMU groups. With Intel® Xeon® class processors (E5 series and above) and "High End Desktop Processors", the processor-based PCIe root ports typically provide native support for PCIe ACS, however the lower-end client processors, such as the Core™ i3, i5, and i7 and Xeon E3 processors do not. For these systems, the PCH root ports generally provide the most flexible isolation configurations.

Another option is to work with the hardware vendors to determine whether isolation is present and quirk the kernel to recognize this isolation. This is generally a matter of determining whether internal peer-to-peer between functions is possible, or in the case of downstream ports, also determining whether redirection is possible. The Red Hat Enterprise Linux 7 kernel includes numerous quirks for such devices and Red Hat Customer Support can help you work with hardware vendors to determine if ACS-equivalent isolation is available and how best to incorporate similar quirks into the kernel to expose this isolation. For hardware vendors, note that multifunction endpoints that do not support peer-to-peer can

expose this using a single static ACS table in configuration space, exposing no capabilities. Adding such a capability to the hardware will allow the kernel to automatically detect the functions as isolated and eliminate this issue for all users of your hardware.

In cases where the above suggestions are not available, a common reaction is that the kernel should provide an option to disable these isolation checks for certain devices or certain types of devices, specified by the user. Often the argument is made that previous technologies did not enforce isolation to this extent and everything "worked fine". Unfortunately, bypassing these isolation features leads to an unsupportable environment. Not knowing that isolation exists, means not knowing whether the devices are actually isolated and it is best to find out before disaster strikes. Gaps in the isolation capabilities of devices may be extremely hard to trigger and even more difficult to trace back to device isolation as the cause. VFIO's job is first and foremost to protect the host kernel from user owned devices and IOMMU groups are the mechanism used by VFIO to ensure that isolation.

In summary, by being built on top of IOMMU groups, VFIO is able to provide an increased degree of security and isolation between devices than was possible using legacy KVM device assignment. This isolation is now enforced at the Linux kernel level, allowing the kernel to protect itself and prevent dangerous configurations for the user. Additionally, hardware vendors should be encouraged to support PCIe ACS support, not only in multifunction endpoint devices, but also in chip sets and interconnect devices. For existing devices lacking this support, Red Hat may be able to work with hardware vendors to determine whether isolation is available and add Linux kernel support to expose this isolation.

---

[1] The original content for this appendix was provided by Alex Williamson, Principal Software Engineer.

[2] The exception is legacy KVM device assignment, which often interacts with the device while bound to the pci-stub driver. Red Hat Enterprise Linux 7 does not include legacy KVM device assignment, avoiding this interaction and potential conflict. Therefore, mixing the use of VFIO and legacy KVM device assignment within the same IOMMU group is not recommended.

APPENDIX F. REVISION HISTORY

Revision 2-39 **Thu Oct 25 2018** **Jiri Herrmann**
Version for 7.6 GA publication

Revision 2-37 **Thu Aug 14 2018** **Jiri Herrmann**
Version for 7.6 Beta publication

Revision 2-36 **Thu Aug 14 2018** **Jiri Herrmann**
Added a rework of the virtual storage chapters

Revision 2-35 **Thu Apr 5 2018** **Jiri Herrmann**
Version for 7.5 GA publication

Revision 2-32 **Thu Jul 27 2017** **Jiri Herrmann**
Version for 7.4 GA publication

Revision 2-32 **Thu Jul 27 2017** **Jiri Herrmann**
Version for 7.4 GA publication

Revision 2-29 **Mon Oct 17 2016** **Jiri Herrmann**
Version for 7.3 GA publication

Revision 2-24 **Thu Dec 17 2015** **Laura Novich**
Republished guide and fixed multiple issues

Revision 2-23 **Sun Nov 22 2015** **Laura Novich**
Republished guide

Revision 2-21 **Thu Nov 12 2015** **Laura Novich**
Multiple content updates for 7.2

Revision 2-19 **Thu Oct 08 2015** **Jiri Herrmann**
Cleaned up the Revision History

Revision 2-17 **Thu Aug 27 2015** **Dayle Parker**
Updates for the 7.2 beta release