openEuler
20.03 LTS

Virtualization User Guide

Date 2020-04-07

Contents

Terms of Use

About This Document

Overview

This document describes virtualization, installation method and usage of openEuler-based virtualization, and guidance for users and administrators to install and use virtualization.

Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

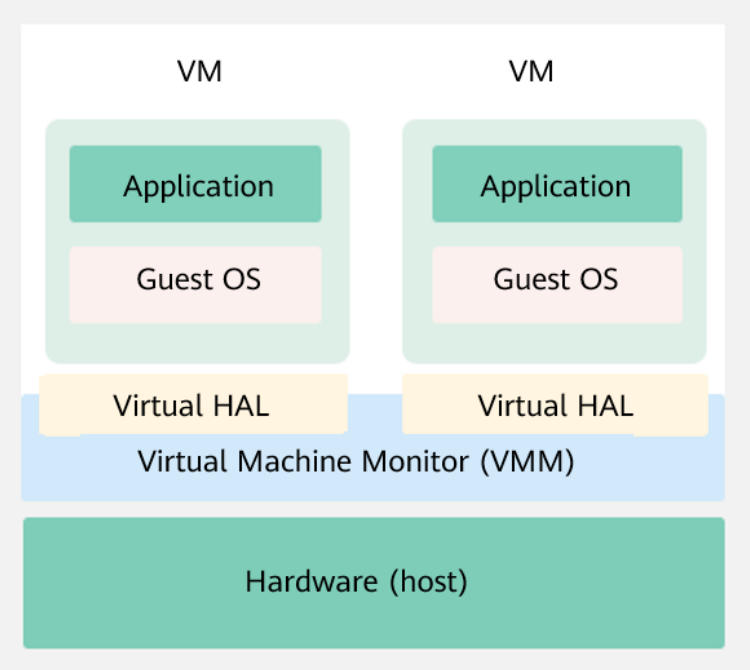
1 Introduction to Virtualization

Overview

In computer technologies, virtualization is a resource management technology. It abstracts various physical resources (such as processors, memory, disks, and network adapters) of a computer, converts the resources, and presents the resources for segmentation and combination into one or more computer configuration environments. This resource management technology breaks the inseparable barrier of the physical structure, and makes these resources not restricted by the architecture, geographical or physical configuration of the existing resources after virtualization. In this way, users can better leverage the computer hardware resources and maximize the resource utilization.

Virtualization enables multiple virtual machines (VMs) to run on a physical server. The VMs share the processor, memory, and I/O resources of the physical server, but are logically isolated from each other. In the virtualization technology, the physical server is called a host machine, the VM running on the host machine is called a guest, and the operating system (OS) running on the VM is called a guest OS. A layer of software, called the virtualization layer, exists between a host machine and a VM to simulate virtual hardware. This virtualization layer is called a VM monitor, as shown in the following figure.

Figure 1-1 Virtualized architecture



Virtualized Architecture

Currently, mainstream virtualization technologies are classified into two types based on the implementation structure of the Virtual Machine Monitor (VMM):

- Hypervisor model

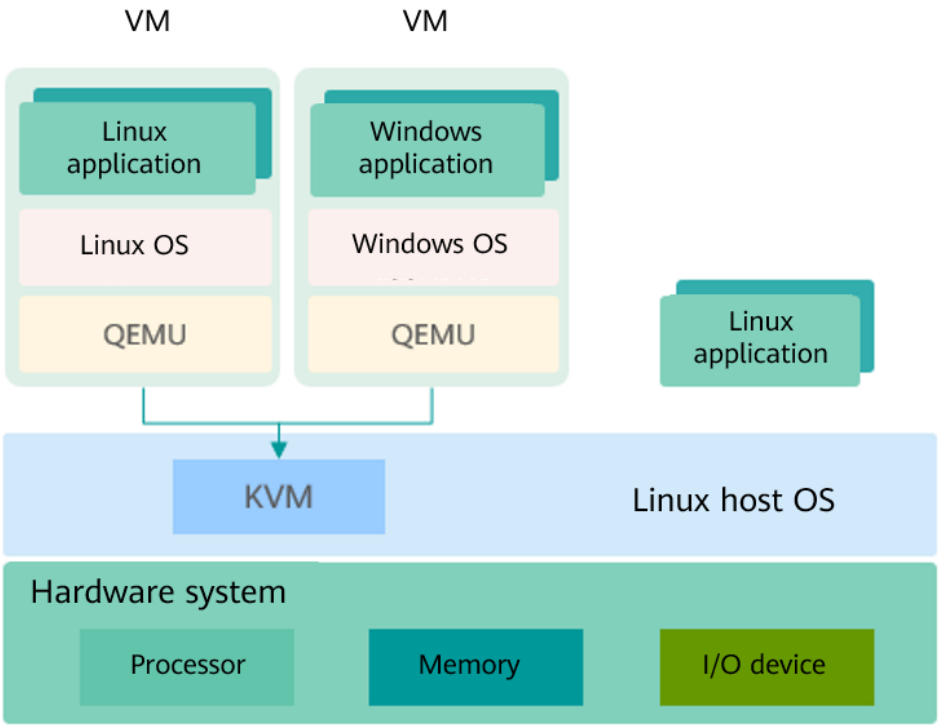
  In this model, VMM is considered as a complete operating system (OS) and has the virtualization function. VMM directly manages all physical resources, including processors, memory, and I/O devices.

- Host model

  In this model, physical resources are managed by a host OS, which is a traditional OS, such as Linux and Windows. The host OS does not provide the virtualization capability. The VMM that provides the virtualization capability runs on the host OS as a driver or software of the system. The VMM invokes the host OS service to obtain resources and simulate the processor, memory, and I/O devices. The virtualization implementation of this model includes KVM and Virtual Box.

Kernel-based Virtual Machine (KVM) is a kernel module of Linux. It makes Linux a hypervisor. Figure 1-2 shows the KVM architecture. KVM does not simulate any hardware device. It is used to enable virtualization capabilities provided by the hardware, such as Intel VT-x, AMD-V, Arm virtualization extensions. The user-mode QEMU simulates the mainboard, memory, and I/O devices. The user-mode QEMU works with the kernel KVM module to simulate VM hardware. The guest OS runs on the hardware simulated by the QEMU and KVM.

Figure 1-2 KVM architecture



Virtualization Components

Virtualization components provided in the openEuler software package:

- KVM: provides the core virtualization infrastructure to make the Linux system a hypervisor. Multiple VMs can run on the same host at the same time.
- QEMU: simulates a processor and provides a set of device models to work with KVM to implement hardware-based virtualization simulation acceleration.
- Libvirt: provides a tool set for managing VMs, including unified, stable, and open application programming interfaces (APIs), daemon process (libvirtd), and default command line management tool (virsh).
- Open vSwitch: provides a virtual network tool set for VMs, supports programming extension and standard management interfaces and protocols (such as NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, and 802.1ag).

Virtualization Characteristics

Virtualization has the following characteristics:

- Partition

  Virtualization can logically divide software on a physical server to run multiple VMs (virtual servers) with different specifications.
- Isolation

  Virtualization can simulate virtual hardware and provide hardware conditions for VMs to run complete OSs. The OSs of each VM are independent and isolated from each other.

For example, if the OS of a VM breaks down due to a fault or malicious damage, the OSs and applications of other VMs are not affected.

- Encapsulation

  Encapsulation is performed on a per VM basis. The excellent encapsulation capability makes VMs more flexible than physical machines. Functions such as live migration, snapshot, and cloning of VMs can be realized, implementing quick deployment and automatic O&M of data centers.

- Hardware-irrelevant

  After being abstracted by the virtualization layer, VMs are not directly bound to underlying hardware and can run on other servers without being modified.

Virtualization Advantages

Virtualization brings the following benefits to infrastructure of the data center:

- Flexibility and scalability

  Users can dynamically allocate and reclaim resources based to meet dynamic service requirements. In addition, users can plan different VM specifications based on product requirements and adjust the scale without changing the physical resource configuration.

- Higher availability and better O&M methods

  Virtualization provides O&M methods such as live migration, snapshot, live upgrade, and automatic DR. Physical resources can be deleted, upgraded, or changed without affecting users, improving service continuity and implementing automatic O&M.

- Security hardening

  Virtualization provides OS-level isolation and hardware-based processor operation privilege-level control. Compared with simple sharing mechanisms, virtualization provides higher security and implements controllable and secure access to data and services.

- High resource utilization

  Virtualization supports dynamic sharing of physical resources and resource pools, improving resource utilization.

openEuler Virtualization

openEuler provides KVM virtualization components that support the AArch64 and x86_64 processor architectures.

2 Installation Guide

This chapter describes how to install virtualization components in openEuler.

2.1 Minimum Hardware Requirements

The minimum hardware requirements for installing virtualization components on openEuler are as follows:

- AArch64 processor architecture: ARMv8 or later, supporting virtualization expansion
- x86_64 processor architecture, supporting VT-x
- 2-core CPU
- 4 GB memory
- 16 GB available disk space

2.2 Installing Core Virtualization Components

2.2.1 Installation Methods

Prerequisites

- The yum source has been configured. For details, see *openEuler 20.03 LTS Administrator Guide*.
- Only the administrator has permission to perform the installation.

Procedure

Step 1 Install the QEMU component.

```
# yum install -y qemu
```

Step 2 Install the libvirt component.

```
# yum install -y libvirt
```

Step 3 Start the libvirtd service.

```
# systemctl start libvirtd
```

----End

NOTE

The KVM module is integrated in the openEuler kernel and does not need to be installed separately.

2.2.2 Verifying the Installation

Step 1 Check whether the kernel supports KVM virtualization, that is, check whether the **/dev/kvm** and **/sys/module/kvm** files exist. The command and output are as follows:

```
# ls /dev/kvm
/dev/kvm
# ls /sys/module/kvm
parameters  uevent
```

If the preceding files exist, the kernel supports KVM virtualization. If the preceding files do not exist, KVM virtualization is not enabled during kernel compilation. In this case, you need to use the Linux kernel that supports KVM virtualization.

Step 2 Check whether QEMU is successfully installed. If the installation is successful, the QEMU software package information is displayed. The command and output are as follows:

```
# rpm -qi qemu
Name       : qemu
Epoch      : 2
Version    : 4.0.1
Release    : 10
Architecture: aarch64
Install Date: Wed 24 Jul 2019 04:04:47 PM CST
Group      : Unspecified
Size       : 16869484
License    : GPLv2 and BSD and MIT and CC-BY
Signature  : (none)
Source RPM  : qemu-4.0.0-1.src.rpm
Build Date  : Wed 24 Jul 2019 04:03:52 PM CST
Build Host  : localhost
Relocations : (not relocatable)
URL        : http://www.qemu.org
Summary    : QEMU is a generic and open source machine emulator and virtualizer
Description :
QEMU is a generic and open source processor emulator which achieves a good
emulation speed by using dynamic translation. QEMU has two operating modes:

 * Full system emulation. In this mode, QEMU emulates a full system (for
   example a PC), including a processor and various peripherials. It can be
   used to launch different Operating Systems without rebooting the PC or
   to debug system code.
 * User mode emulation. In this mode, QEMU can launch Linux processes compiled
   for one CPU on another CPU.
```

```
As QEMU requires no host kernel patches to run, it is safe and easy to use.
```

Step 3 Check whether libvirt is successfully installed. If the installation is successful, the libvirt software package information is displayed. The command and output are as follows:

```
# rpm -qi libvirt
Name       : libvirt
Version    : 5.5.0
Release    : 1
Architecture: aarch64
Install Date: Tue 30 Jul 2019 04:56:21 PM CST
Group      : Unspecified
Size       : 0
License    : LGPLv2+
Signature   : (none)
Source RPM  : libvirt-5.5.0-1.src.rpm
Build Date  : Mon 29 Jul 2019 08:14:57 PM CST
Build Host  : 71e8c1ce149f
Relocations : (not relocatable)
URL        : https://libvirt.org/
Summary    : Library providing a simple virtualization API
Description :
Libvirt is a C toolkit to interact with the virtualization capabilities
of recent versions of Linux (and other OSes). The main package includes
the libvirtd server exporting the virtualization support.
```

Step 4 Check whether the libvirt service is started successfully. If the service is in the **Active** state, the service is started successfully. You can use the virsh command line tool provided by the libvirt. The command and output are as follows:

```
# systemctl status libvirtd
● libvirtd.service - Virtualization daemon
   Loaded: loaded (/usr/lib/systemd/system/libvirtd.service; enabled; vendor preset:
enabled)
   Active: active (running) since Tue 2019-08-06 09:36:01 CST; 5h 12min ago
     Docs: man:libvirtd(8)
           https://libvirt.org
 Main PID: 40754 (libvirtd)
    Tasks: 20 (limit: 32768)
   Memory: 198.6M
   CGroup: /system.slice/libvirtd.service
           —40754 /usr/sbin/libvirtd
```

----End

3 User and Administrator Guide

This chapter describes how to create VMs on the virtualization platform, manage VM life cycles, and query information.

3.1 Environment Preparation

3.1.1 Preparing a VM Image

Overview

A VM image is a file that contains a virtual disk that has been installed and can be used to start the OS. VM images are in different formats, such as raw and qcow2. Compared with the raw format, the qcow2 format occupies less space and supports features such as snapshot, copy-on-write, AES encryption, and zlib compression. However, the performance of the qcow2 format is slightly lower than that of the raw format. The qemu-img tool is used to create image files. This section uses the qcow2 image file as an example to describe how to create a VM image.

Creating an Image

To create a qcow2 image file, perform the following steps:

Step 1 Install the **qemu-img** software package.

```
# yum install -y qemu-img
```

Step 2 Run the **create** command of the qemu-img tool to create an image file. The command format is as follows:

```
$ qemu-img create -f <imgFormat> -o <fileOption> <fileName> <diskSize>
```

The parameters are described as follows:

- *imgFormat*: Image format. The value can be **raw** or **qcow2**.
- *fileOption*: File option, which is used to set features of an image file, such as specifying a backend image file, compression, and encryption.
- *fileName*: File name.
- *diskSize*: Disk size, which specifies the size of a block disk. The unit can be K, M, G, or T, indicating KiB, MiB, GiB, or TiB.

For example, to create an image file openEuler-imge.qcow2 whose disk size is 4 GB and format is qcow2, the command and output are as follows:

```
$ qemu-img create -f qcow2 openEuler-image.qcow2 4G
Formatting 'openEuler-image.qcow2', fmt=qcow2 size=4294967296 cluster size=65536
lazy_refcounts=off refcount_bits=16
```

----End

Changing the Image Disk Space

If a VM requires larger disk space, you can use the qemu-img tool to change the disk space of the VM image. The method is as follows:

Step 1 Run the following command to query the disk space of the VM image:

```
# qemu-img info <imgFiLeName>
```

For example, if the command and output for querying the disk space of the openEuler-image.qcow2 image are as follows, the disk space of the image is 4 GiB.

```
# qemu-img info openEuler-image.qcow2
image: openEuler-image.qcow2
file format: qcow2
virtual size: 4.0G (4294967296 bytes)
disk size: 196K
cluster size: 65536
Format specific information:
    compat: 1.1
    lazy refcounts: false
    refcount bits: 16
    corrupt: false
```

Step 2 Run the following command to change the image disk space. In the command, *imgFiLeName* indicates the image name, and + and - indicate the image disk space to be increased and decreased, respectively. The unit is KB, MB, GB, and T, indicating KiB, MiB, GiB, and TiB, respectively.

```
# qemu-img resize <imgFiLeName> [+|-]<size>
```

For example, to expand the disk space of the openEuler-image.qcow2 image to 24 GiB, that is, to add 20 GiB to the original 4 GiB, the command and output are as follows:

```
# qemu-img resize openEuler-image.qcow2 +20G
Image resized.
```

Step 3 Run the following command to check whether the image disk space is changed successfully:

```
# qemu-img info <imgFiLeName>
```

For example, if the openEuler-image.qcow2 image disk space has been expanded to 24 GiB, the command and output are as follows:

```
# qemu-img info openEuler-image.qcow2
image: openEuler-image.qcow2
file format: qcow2
virtual size: 24G (25769803776 bytes)
disk size: 200K
cluster size: 65536
Format specific information:
    compat: 1.1
    lazy refcounts: false
    refcount bits: 16
    corrupt: false
```

----End

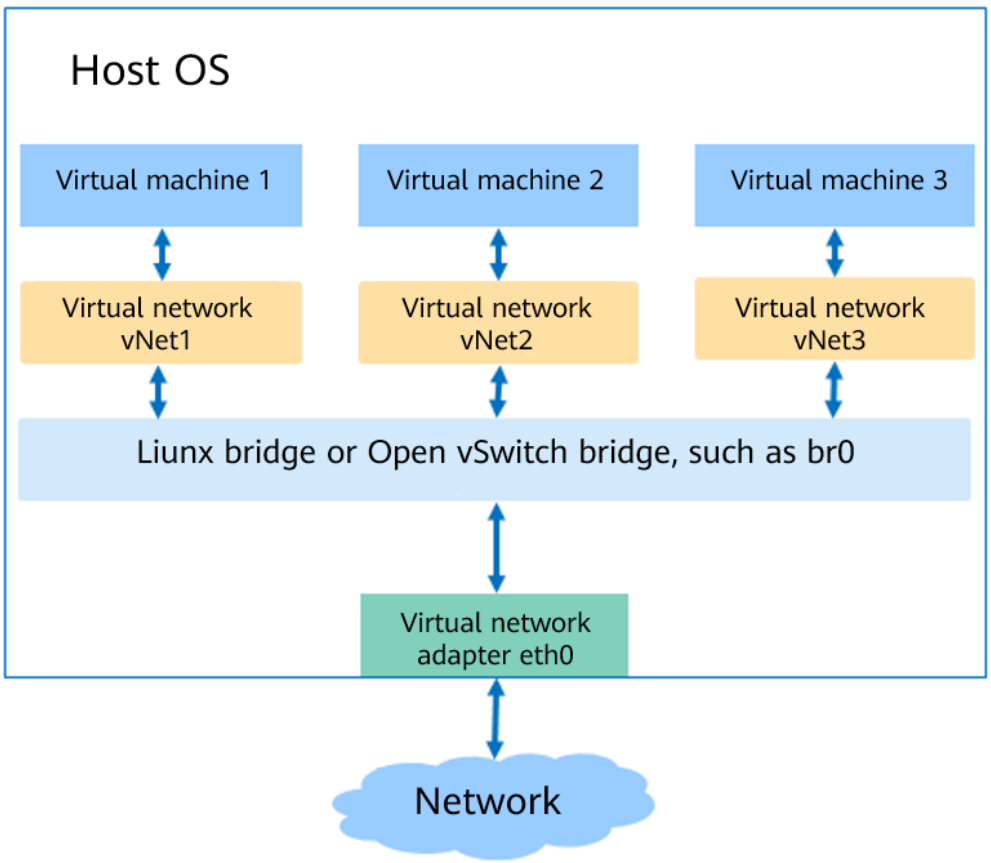
3.1.2 Preparing the VM Network

Overview

To enable the VM to communicate with external networks, you need to configure the network environment for the VM. KVM virtualization supports multiple types of bridges, such as Linux bridge and Open vSwitch bridge. As shown in Figure 3-1, the data transmission path is **VM > virtual NIC device > Linux bridge or Open vSwitch bridge > physical NIC**. In addition to configuring virtual NICs (vNICs) for VMs, creating a bridge for a host is the key to connecting to a virtualized network.

This section describes how to set up a Linux bridge and an Open vSwitch bridge to connect a VM to the network. You can select a bridge type based on the site requirements.

Figure 3-1 Virtual network structure



Setting Up a Linux Bridge

The following describes how to bind the physical NIC eth0 to the Linux bridge br0.

Step 1 Install the **bridge-utils** software package.

The Linux bridge is managed by the brctl tool. The corresponding installation package is bridge-utils. The installation command is as follows:

```
# yum install -y bridge-utils
```

Step 2 Create bridge br0.

```
# brctl addbr br0
```

Step 3 Bind the physical NIC eth0 to the Linux bridge.

```
# brctl addif br0 eth0
```

Step 4 After eth0 is connected to the bridge, the IP address of eth0 is set to 0.0.0.0.

```
# ifconfig eth0 0.0.0.0
```

Step 5 Set the IP address of br0.

- If a DHCP server is available, set a dynamic IP address through the dhclient.

  ```
  # dhclient br0
  ```

- If no DHCP server is available, configure a static IP address for br0. For example, set the static IP address to 192.168.1.2 and subnet mask to 255.255.255.0.

```
# ifconfig br0 192.168.1.2 netmask 255.255.255.0
```

----End

Setting Up an Open vSwitch Bridge

The Open vSwitch bridge provides more convenient automatic orchestration capabilities. This section describes how to install network virtualization components to set up an Open vSwitch bridge.

1. Install the Open vSwitch component.

If the Open vSwitch is used to provide virtual network, you need to install the Open vSwitch network virtualization component.

Step 1 Install the Open vSwitch component.

```
# yum install -y openvswitch-kmod
# yum install -y openvswitch
```

Step 2 Start the Open vSwitch service.

```
# systemctl start openvswitch
```

----End

2. Check whether the installation is successful.

Check whether the Open vSwitch components, openvswitch-kmod and openvswitch, are successfully installed.

Step 1 Check whether the openvswitch-kmod component is successfully installed. If the installation is successful, the software package information is displayed. The command and output are as follows:

```
# rpm -qi openvswitch-kmod
Name       : openvswitch-kmod
Version    : 2.11.1
Release    : 1.oe3
Architecture: aarch64
Install Date: Thu 15 Aug 2019 05:07:49 PM CST
Group      : System Environment/Daemons
Size       : 15766774
License    : GPLv2
Signature  : (none)
Source RPM  : openvswitch-kmod-2.11.1-1.oe3.src.rpm
Build Date  : Thu 08 Aug 2019 04:33:08 PM CST
Build Host  : armbuild10b175b113b44
Relocations : (not relocatable)
Vendor     : OpenSource Security Ralf Spenneberg <ralf@os-s.net>
URL        : http://www.openvswitch.org/
Summary    : Open vSwitch Kernel Modules
Description :
Open vSwitch provides standard network bridging functions augmented with
```

```
support for the OpenFlow protocol for remote per-flow control of
traffic. This package contains the kernel modules.
```

Step 2 Check whether the openvswitch component is successfully installed. If the installation is successful, the software package information is displayed. The command and output are as follows:

```
# rpm -qi openvswitch
Name       : openvswitch
Version    : 2.11.1
Release    : 1
Architecture: aarch64
Install Date: Thu 15 Aug 2019 05:08:35 PM CST
Group      : System Environment/Daemons
Size       : 6051185
License    : ASL 2.0
Signature  : (none)
Source RPM  : openvswitch-2.11.1-1.src.rpm
Build Date  : Thu 08 Aug 2019 05:24:46 PM CST
Build Host  : armbuild10b247b121b105
Relocations : (not relocatable)
Vendor     : Nicira, Inc.
URL        : http://www.openvswitch.org/
Summary    : Open vSwitch daemon/database/utilities
Description :
Open vSwitch provides standard network bridging functions and
support for the OpenFlow protocol for remote per-flow control of
traffic.
```

Step 3 Check whether the Open vSwitch service is started successfully. If the service is in the **Active** state, the service is started successfully. You can use the command line tool provided by the Open vSwitch. The command and output are as follows:

```
# systemctl status openvswitch
● openvswitch.service - LSB: Open vSwitch switch
   Loaded: loaded (/etc/rc.d/init.d/openvswitch; generated)
   Active: active (running) since Sat 2019-08-17 09:47:14 CST; 4min 39s ago
     Docs: man:systemd-sysv-generator(8)
  Process: 54554 ExecStart=/etc/rc.d/init.d/openvswitch start (code=exited,
status=0/SUCCESS)
    Tasks: 4 (limit: 9830)
   Memory: 22.0M
   CGroup: /system.slice/openvswitch.service
           ├─54580 ovsdb-server: monitoring pid 54581 (healthy)
           ├─54581 ovsdb-server /etc/openvswitch/conf.db -vconsole:emer -vsyslog:err
-vfile:info --remote=punix:/var/run/openvswitch/db.sock
--private-key=db:Open vSwitch,SSL,private key --certificate>
           ├─54602 ovs-vswitchd: monitoring pid 54603 (healthy)
           └─54603 ovs-vswitchd unix:/var/run/openvswitch/db.sock -vconsole:emer
-vsyslog:err -vfile:info --mlockall --no-chdir
--log-file=/var/log/openvswitch/ovs-vswitchd.log --pidfile=/var/run/open>
```

----End

3. Set up an Open vSwitch bridge

The following describes how to set up an Open vSwitch layer-1 bridge br0.

Step 1 Create the Open vSwitch bridge br0.

```
# ovs-vsctl add-br br0
```

Step 2 Add the physical NIC eth0 to br0.

```
# ovs-vsctl add-port br0 eth0
```

Step 3 After eth0 is connected to the bridge, the IP address of eth0 is set to 0.0.0.0.

```
# ifconfig eth0 0.0.0.0
```

Step 4 Assign an IP address to OVS bridge br0.

- If a DHCP server is available, set a dynamic IP address through the dhclient.

  ```
  # dhclient br0
  ```

- If no DHCP server is available, configure a static IP address for br0, for example, 192.168.1.2.

  ```
  # ifconfig br0 192.168.1.2
  ```

----End

3.1.3 Preparing Boot Firmware

Overview

The boot mode varies depending on the architecture. x86 servers support the Unified Extensible Firmware Interface (UFEI) and BIOS boot modes, and AArch64 servers support only the UFEI boot mode. By default, boot files corresponding to the BIOS mode have been installed on openEuler. No additional operations are required. This section describes how to install boot files corresponding to the UEFI mode.

The Unified Extensible Firmware Interface (UEFI) is a new interface standard used for power-on auto check and OS boot. It is an alternative to the traditional BIOS. EDK II is a set of open source code that implements the UEFI standard. In virtualization scenarios, the EDK II tool set is used to start a VM in UEFI mode. Before using the EDK II tool, you need to install the corresponding software package before starting a VM. This section describes how to install the EDK II tool.

Installation Methods

If the UEFI mode is used, the tool set EDK II needs to be installed. The installation package for the AArch64 architecture is **edk2-aarch64**, and that for the x86 architecture is **edk2-ovmf**. This section uses the AArch64 architecture as an example to describe the installation method. For the x86 architecture, you only need to replace **edk2-aarch64** with **edk2-ovmf**.

Step 1 Run the following command to install the **edk** software package:

In the AArch64 architecture, the **edk2** package name is **edk2-aarch64**.

```
# yum install -y edk2-aarch64
```

In the x86_64 architecture, the **edk2** package name is **edk2-ovmf**.

```
# yum install -y edk2-ovmf
```

Step 2 Run the following command to check whether the **edk** software package is successfully installed:

In the AArch64 architecture, the command is as follows:

```
# rpm -qi edk2-aarch64
```

If information similar to the following is displayed, the **edk** software package is successfully installed:

```
Name       : edk2-aarch64
Version    : 20180815gitcb5f4f45ce
Release    : 1.oe3
Architecture: noarch
Install Date: Mon 22 Jul 2019 04:52:33 PM CST
Group      : Applications/Emulators
```

In the x86_64 architecture, the command is as follows:

```
# rpm -qi edk2-ovmf
```

If information similar to the following is displayed, the **edk** software package is successfully installed:

```
Name       : edk2-ovmf
Version    : 201908
Release    : 6.oe1
Architecture: noarch
Install Date: Thu 19 Mar 2020 09:09:06 AM CST
```

----End

3.2 VM Configuration

3.2.1 Introduction

Overview

Libvirt tool uses XML files to describe a VM feature, including the VM name, CPU, memory, disk, NIC, mouse, and keyboard. You can manage a VM by modifying configuration files. This section describes the elements in the XML configuration file to help users configure VMs.

Format

The VM XML configuration file uses domain as the root element, which contains multiple other elements. Some elements in the XML configuration file can contain corresponding attributes and attribute values to describe VM information in detail. Different attributes of the same element are separated by spaces.

The basic format of the XML configuration file is as follows. In the format, **label** indicates the label name, **attribute** indicates the attribute, and **value** indicates the attribute value. Change them based on the site requirements.

```
<domain type='kvm'>
   <name>VMName</name>
   <memory attribute='value'>8</memory>
   <vcpu>4</vcpu>
   <os>
```

```
      <label attribute='value' attribute='value'>
        ...
      </label>
    </os>
    <label attribute='value' attribute='value'>
      ...
    </label>
</domain>
```

Process

1. Create an XML configuration file with domain root element.
2. Use the name tag to specify a unique VM name based on the naming rule.
3. Configure system resources such as the virtual CPU (vCPU) and virtual memory.
4. Configure virtual devices.
   a. Configure storage devices.
   b. Configure network devices.
   c. Configure the external bus structure.
   d. Configure external devices such as the mouse.
5. Save the XML configuration file.

3.2.2 VM Description

Overview

This section describes how to configure the VM **domain** root element and VM name.

Elements

- **domain**: Root element of a VM XML configuration file, which is used to configure the type of the hypervisor that runs the VM.

  type: Type of a domain in virtualization. In the openEuler virtualization, the attribute value is **kvm**.

- **name**: VM name.

  The VM name is a unique character string on the same host. The VM name can contain only digits, letters, underscores (_), hyphens (-), and colons (:), but cannot contain only digits. The VM name can contain a maximum of 64 characters.

Configuration Example

For example, if the VM name is **openEuler**, the configuration is as follows:

```
<domain type='kvm'>
    <name>openEuler</name>
    ...
</domain>
```

3.2.3 vCPU and Virtual Memory

Overview

This section describes how to configure the vCPU and virtual memory.

Elements

- **vcpu**: The number of virtual processors.
- **memory**: The size of the virtual memory.

  unit: The memory unit. The value can be KiB (2^{10} bytes), MiB (2^{20} bytes), GiB (2^{30} bytes), or TiB (2^{40} bytes).
- **cpu**: The mode of the virtual processor.

  mode: The mode of the vCPU. The **host-passthrough** indicates that the architecture and features of the virtual CPU are the same as those of the host.

  Sub-element **topology**: A sub-element of the element cpu, used to describe the topology structure of a vCPU mode.
  - The attributes **socket**, **cores**, and **threads** of the sub-element topology describe the number of CPU sockets of a VM, the number of processor cores included in each CPU socket, and the number of hyperthreads included in each processor core, respectively. The attribute value is a positive integer, and a product of the three values is equal to the number of of vCPUs.

Configuration Example

For example, if the number of vCPUs is 4, the processing mode is host-passthrough, the virtual memory is 8 GiB, the four CPUs are distributed in two CPU sockets, and hyperthreading is not supported, the configuration is as follows:

```
<domain type='kvm'>
   ...
   <vcpu>4</vcpu>
   <memory unit='GiB'>8</memory>
   <cpu mode='host-passthrough'>
      <topology sockets='2' cores='2' threads='1'/>
   </cpu>
...
</domain>
```

3.2.4 Virtual Device Configuration

The VM XML configuration file uses the **devices** elements to configure virtual devices, including storage devices, network devices, buses, and mouse devices. This section describes how to configure common virtual devices.

3.2.4.1 Storage Devices

Overview

This section describes how to configure virtual storage devices, including floppy disks, disks, and CD-ROMs and their storage types.

Elements

The XML configuration file uses the **disk** element to configure storage devices. Table 3-1 describes common **disk** attributes. Table 3-2 describes common subelements and their attributes.

Table 3-1 Common attributes of the **disk** element

Element	Attribute	Description	Attribute Value and Description
disk	**type**	Specifies the type of the backend storage medium.	**block**: block device **file**: file device **dir**: directory path
	device	Specifies the storage medium to be presented to the VM.	**disk**: disk (default) **floppy**: floppy disk **cdrom**: CD-ROM

Table 3-2 Common subelements and attributes of the **disk** element

Subelement	Subelement Description	Attribute Description
source	Specifies the backend storage medium, which corresponds to the type specified by the **type** attribute of the **disk** element.	**file**: file type. The value is the fully qualified path of the corresponding file. **dev**: block type. The value is the fully qualified path of the corresponding host device. **dir**: directory type. The value is the fully qualified path of the disk directory.
driver	Details about the specified backend driver	**type**: disk format type. The value can be **raw** or **qcow2**, which must be the same as that of source. **io**: disk I/O mode. The options are **native** and **threads**. **cache**: disk cache mode. The value can be **none**, **writethrough**, **writeback**, or **directsync**. **iothread**: I/O thread allocated to the disk.
target	The bus and device that a disk presents to a VM.	**dev**: specifies the logical device name of a disk, for example, **sd[a-p]** for SCSI, SATA, and USB buses and **hd[a-d]** for IDE disks. **bus**: specifies the type of a disk. Common types include scsi, usb, sata, and virtio.
boot	The disk can be used as the boot disk.	**order**: specifies the disk startup sequence.

Subelement	Subelement Description	Attribute Description
readonly	The disk is read-only and cannot be modified by the VM. Generally, it is used together with the CD-ROM drive.	-

Configuration Example

After the VM image is prepared according to 3.1.1 Preparing a VM Image, you can use the following XML configuration file to configure the virtual disk for the VM.

In this example, two I/O threads, one block disk device and one CD, are configured for the VM, and the first I/O thread is allocated to the block disk device for use. The backend medium of the disk device is in qcow2 format and is used as the preferred boot disk.

```
<domain type='kvm'>
   ...
   <iothreads>2</iothreads>
   <devices>
      <disk type='file' device='disk'>
       <driver name='qemu' type='qcow2' cache='none' io='native' iothread="1"/>
       <source file='/mnt/openEuler-image.qcow2'/>
       <target dev='vda' bus='virtio'/>
       <boot order='1'/>
    </disk>
    <disk type='file' device='cdrom'>
       <driver name='qemu' type='raw' cache='none' io='native'/>
       <source file='/mnt/openEuler-20.03-LTS-aarch64-dvd.iso'/>
       <target dev='sdb' bus='scsi'/>
       <readonly/>
       <boot order='2'/>
    </disk>
       ...
   </devices>
</domain>
```

3.2.4.2 Network Device

Overview

The XML configuration file can be used to configure virtual network devices, including the ethernet mode, bridge mode, and vhostuser mode. This section describes how to configure vNICs.

Elements

In the XML configuration file, the element **interface** is used, and its attribute **type** indicates the mode of the vNIC. The options are **ethernet**, **bridge**, and **vhostuser**. The following uses the vNIC in bridge mode as an example to describe its subelements and attributes.

Table 3-3 Common subelements of a vNIC in bridge mode

Subelement	Subelement Description	Attribute and Description
mac	The mac address of the vNIC.	**address**: specifies the mac address. If this parameter is not set, the system automatically generates a mac address.
target	Name of the backend vNIC.	**dev**: name of the created backend tap device.
source	Specify the backend of the vNIC.	**bridge**: used together with the bridge mode. The value is the **bridge name**.
boot	The NIC can be used for remote startup.	**order**: specifies the startup sequence of NICs.
model	Indicates the type of a vNIC.	**type**: **virtio** is usually used for the NIC in bridge mode.
virtualport	Port type	**type**: If an OVS bridge is used, set this parameter to **openvswitch**.
driver	Backend driver type	**name**: driver name. The value is **vhost**. **queues**: the number of NIC queues.

Configuration Example

- After creating the Linux bridge br0 by referring to 3.1.2 Preparing the VM Network, configure a vNIC of the VirtIO type bridged on the br0 bridge. The corresponding XML configuration is as follows:

```
<domain type='kvm'>
   ...
   <devices>
      <interface type='bridge'>
       <source bridge='br0'/>
          <model type='virtio'/>
    </interface>
       ...
   </devices>
</domain>
```

- If an OVS network bridge is created according to 3.1.2 Preparing the VM Network, configure a VirtIO vNIC device that uses the vhost driver and has four queues.

```
<domain type='kvm'>
   ...
   <devices>
```

```
      <interface type='bridge'>
       <source bridge='br0'/>
          <virtualport type='openvswitch'/>
          <model type='virtio'/>
          <driver name='vhost' queues='4'/>
    </interface>
       ...
   </devices>
</domain>
```

3.2.4.3 Bus Configuration

Overview

The bus is a channel for information communication between components of a computer. An external device needs to be mounted to a corresponding bus, and each device is assigned a unique address (specified by the subelement **address**). Information exchange with another device or a central processing unit (CPU) is completed through the bus network. Common device buses include the ISA bus, PCI bus, USB bus, SCSI bus, and PCIe bus.

The PCIe bus is a typical tree structure and has good scalability. The buses are associated with each other by using a controller. The following uses the PCIe bus as an example to describe how to configure a bus topology for a VM.

NOTE

The bus configuration is complex. If the device topology does not need to be precisely controlled, the default bus configuration automatically generated by libvirt can be used.

Elements

In the XML configuration of libvirt, each controller element (**controller**) represents a bus, and one or more controllers or devices can be mounted to one controller depending on the VM architecture. This topic describes common attributes and subelements.

controller: controller element, which indicates a bus.

- Attribute **type**: bus type, which is mandatory for the controller. The common values are **pci**, **usb**, **scsi**, **virtio-serial**, **fdc**, and **ccid**.
- Attribute **index**: bus number of the controller (the number starts from 0), which is mandatory for the controller. This attribute can be used in the **address** element.
- Attribute **model**: specific model of the controller, which is mandatory for the controller. The available values are related to the value of **type**. For details about the mapping and description, see Table 3-4.
- Subelement **address**: mount location of a device or controller on the bus network.
  - Attribute **type**: device address type. The common values are **pci**, **usb**, or **drive**. The attribute varies according to the **type** of the **address**. For details about the common **type** attribute value and the corresponding **address** attribute, see Table 3-5.
- Subelement **model**: name of a controller model.
  - Attribute **name**: name of a controller model, which corresponds to the **model** attribute in the parent element controller.

Table 3-4 Mapping between the common values of **type** and **model** for the controller.

Value of Type	Value of Model	Introduction
pci	pcie-root	PCIe root node, which can be used to mount PCIe devices or controllers.
	pcie-root-port	Only one slot can be used to mount a PCIe device or controller.
	pcie-to-pci-bridge	PCIe-to-PCI bridge controller, which can be used to mount PCI devices.
usb	ehci	USB 2.0 controller, which can be used to mount USB 2.0 devices.
	nec-xhci	USB 3.0 controller, which can be used to mount USB 3.0 devices.
scsi	virtio-scsi	VirtIO SCSI controller, which can be used to mount block devices, such as disks and CD-ROMs.
virtio-serial	virtio-serial	VirtIO serial port controller, which can be used to mount serial port devices, such as a pty serial port.

Table 3-5 Attributes of the **address** element in different devices.

Value of Type	Description	Address
pci	The **address** type is PCI address, indicating the mount location of the device on the PCI bus network.	**domain**: domain ID of the PCI device. **bus**: bus number of the PCI device. **slot**: device number of the PCI device. **function**: function number of the PCI device. **multifunction**: (optional) specifies whether to enable the multifunction function.
usb	The **address** type is USB address, indicating the location of the device on the USB bus.	**bus**: bus number of the USB device. **port**: port number of the USB device.
drive	The **address** type is storage device address, indicating the owning disk controller and its position on the bus.	**controller**: the number of the owning controller. **bus**: channel number of the device output. **target**: target number of the storage device. **unit**: lun number of the storage device.

Configuration Example

This example shows the topology of a PCIe bus. Three PCIe-Root-Port controllers are mounted to the PCIe root node (BUS 0). The multifunction function is enabled for the first PCIe-Root-Port controller (BUS 1). A PCIe-to-PCI-bridge controller is mounted to the first PCIe-Root-Port controller to form a PCI bus (BUS 3). A virtio-serial device and a USB 2.0 controller are mounted to the PCI bus. A SCSI controller is mounted to the second PCIe-Root-Port controller (BUS 2). No device is mounted to the third PCIe-Root-Port controller (BUS 0). The configuration details are as follows:

```
<domain type='kvm'>
   ...
   <devices>
      <controller type='pci' index='0' model='pcie-root'/>
    <controller type='pci' index='1' model='pcie-root-port'>
       <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x0'
multifunction='on'/>
    </controller>
    <controller type='pci' index='2' model='pcie-root-port'>
       <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
    </controller>
    <controller type='pci' index='3' model='pcie-to-pci-bridge'>
       <model name='pcie-pci-bridge'/>
       <address type='pci' domain='0x0000' bus='0x01' slot='0x00' function='0x0'/>
    </controller>
    <controller type='pci' index='4' model='pcie-root-port'>
       <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x2'/>
    </controller>
    <controller type='scsi' index='0' model='virtio-scsi'>
       <address type='pci' domain='0x0000' bus='0x02' slot='0x00' function='0x0'/>
    </controller>
    <controller type='virtio-serial' index='0'>
       <address type='pci' domain='0x0000' bus='0x03' slot='0x02' function='0x0'/>
    </controller>
    <controller type='usb' index='0' model='ehci'>
       <address type='pci' domain='0x0000' bus='0x03' slot='0x01' function='0x0'/>
    </controller>
    ...
    </devices>
</domain>
```

3.2.4.4 Other Common Devices

Overview

In addition to storage devices and network devices, some external devices need to be specified in the XML configuration file. This section describes how to configure these elements.

Elements

- **serial**: serial port device

  Attribute **type**: specifies the serial port type. The common attribute values are **pty**, **tcp**, **pipe**, and **file**.

- **video**: media device

  type attribute: media device type The common attribute value of the AArch architecture is **virtio**, and that of the x86_64 architecture is **vga** or **cirrus**.

  Subelement **model**: subelement of **video**, which is used to specify the media device type.

  In the subelement **model**, if **type** is set to **vga**, a Video Graphics Array (VGA) video card is configured. **vram** indicates the size of the video RAM, in KB by default.

  For example, if a 16 MB VGA video card is configured for an x86_64 VM, configuration in the XML file is as follows. In the example, the value of **vram** is the size of video RAM, in KB by default.

  ```
  <video>
     <model type='vga' vram='16384' heads='1' primary='yes'/>
  </video>
  ```

- **input**: output device

  type attribute: specifies the type of the output device. The common attribute values are **tabe** and **keyboard**, indicating that the output device is the tablet and keyboard respectively.

  bus: specifies the bus to be mounted. The common attribute value is **USB**.

- **emulator**: emulator application path
- **graphics**: graphics device

  type attribute: specifies the type of a graphics device. The common attribute value is **vnc**.

  listen attribute: specifies the IP address to be listened to.

Configuration Example

For example, in the following example, the VM emulator path, pty serial port, VirtIO media device, USB tablet, USB keyboard, and VNC graphics device are configured.

> NOTE
>
> When **type** of **graphics** is set to **VNC**, you are advised to set the **passwd** attribute, that is, the password for logging in to the VM using VNC.

```
<domain type='kvm'>
   ...
   <devices>
      <emulator>/usr/libexec/qemu-kvm</emulator>
      <console type='pty'/>
      <video>
         <model type='virtio'/>
      </video>
      <input type='tablet' bus='usb'/>
      <input type='keyboard' bus='usb'/>
      <graphics type='vnc' listen='0.0.0.0' passwd='n8VfjbFK'/>
    ...
    </devices>
</domain>
```

3.2.5 Configurations Related to the System Architecture

Overview

The XML configuration file contain configurations related to the system architecture, which cover the mainboard, CPU, and some features related to the architecture. This section describes meanings of these configurations.

Elements

- **os**: defines VM startup parameters.

  Subelement **type**: specifies the VM type. The attribute **arch** indicates the architecture type, for example, AArch64. The attribute **machine** indicates the type of VM chipset. Supported chipset type can be queried by running the **qemu-kvm -machine ?** command. For example, the AArch64 architecture supports the **virt** type.

  Subelement **loader**: specifies the firmware to be loaded, for example, the UEFI file provided by the EDK. The **readonly** attribute indicates whether the file is read-only. The value can be **yes** or **no**. The **type** attribute indicates the **loader** type. The common values are **rom** and **pflash**.

  Subelement **nvram**: specifies the path of the **nvram** file, which is used to store the UEFI startup configuration.

- **features**: Hypervisor controls some VM CPU/machine features, such as the advanced configuration and power interface (ACPI) and the GICv3 interrupt controller specified by the ARM processor.

Example for AArch64 Architecture

The VM is of the **aarch64** type and uses **virt** chipset. The VM configuration started using UEFI is as follows:

```
<domain type='kvm'>
   ...
   <os>
      <type arch='aarch64' machine='virt'>hvm</type>
      <loader readonly='yes'
type='pflash'>/usr/share/edk2/aarch64/QEMU EFI-pflash.raw</loader>
      <nvram>/var/lib/libvirt/qemu/nvram/openEulerVM.fd</nvram>
   </os>
   ...
</domain>
```

Configure ACPI and GIC V3 interrupt controller features for the VM.

```
<features>
   <acpi/>
   <gic version='3'/>
</features>
```

Example for x86_64 Architecture

The x86_64 architecture supports both BIOS and UEFI boot modes. If **loader** is not configured, the default BIOS boot mode is used. The following is a configuration example in which the UEFI boot mode and Q35 chipsets are used.

```
<domain type='kvm'>
   ...
   <os>
      <type arch='x86 64' machine='q35'>hvm</type>
      <loader type='rom'>/usr/share/edk2/ovmf/OVMF.fd</loader>
   </os>
   ...
</domain>
```

3.2.6 Other Common Configuration Items

Overview

In addition to system resources and virtual devices, other elements need to be configured in the XML configuration file. This section describes how to configure these elements.

Elements

- **iothreads**: specifies the number of **iothread**, which can be used to accelerate storage device performance.
- **on_poweroff**: action taken when a VM is powered off.
- **on_reboot**: action taken when a VM is rebooted.
- **on_crash**: action taken when a VM is on crash.
- **clock**: indicates the clock type.

  offset attribute: specifies the VM clock synchronization type. The value can be **localtime**, **utc**, **timezone**, or **variable**.

Configuration Example

Configure two **iothread** for the VM to accelerate storage device performance.

```
<iothreads>2</iothreads>
```

Destroy the VM when it is powered off.

```
<on_poweroff>destroy</on_poweroff>
```

Restart the VM.

```
<on_reboot>restart</on_reboot>
```

Restart the VM when it is crashed.

```
<on_crash>restart</on_crash>
```

The clock uses the **utc** synchronization mode.

```
<clock offset='utc'/>
```

3.2.7 XML Configuration File Example

Overview

This section provides XML configuration files of a basic AArch64 VM and a x86_64 VM as two examples for reference.

Example 1

An XML configuration file of AArch64 VM, which contains basic elements. The following is a configuration example:

```
<domain type='kvm'>
   <name>openEulerVM</name>
   <memory unit='GiB'>8</memory>
   <vcpu>4</vcpu>
   <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <loader readonly='yes'
type='pflash'>/usr/share/edk2/aarch64/QEMU EFI-pflash.raw</loader>
    <nvram>/var/lib/libvirt/qemu/nvram/openEulerVM.fd</nvram>
   </os>
   <features>
    <acpi/>
    <gic version='3'/>
   </features>
   <cpu mode='host-passthrough'>
      <topology sockets='2' cores='2' threads='1'/>
   </cpu>
   <iothreads>1</iothreads>
   <clock offset='utc'/>
   <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' iothread="1"/>
       <source file='/mnt/openEuler-image.qcow2'/>
       <target dev='vda' bus='virtio'/>
       <boot order='1'/>
    </disk>
    <disk type='file' device='cdrom'>
       <driver name='qemu' type='raw'/>
       <source file='/mnt/openEuler-20.03-LTS-aarch64-dvd.iso'/>
       <readonly/>
       <target dev='sdb' bus='scsi'/>
       <boot order='2'/>
    </disk>
    <interface type='bridge'>
       <source bridge='br0'/>
       <model type='virtio'/>
    </interface>
    <console type='pty'/>
      <video>
        <model type='virtio'/>
      </video>
      <controller type='scsi' index='0' model='virtio-scsi'/>
    <controller type='usb' model='ehci'/>
    <input type='tablet' bus='usb'/>
    <input type='keyboard' bus='usb'/>
    <graphics type='vnc' listen='0.0.0.0' passwd='n8VfjbFK'/>
```

```
    </devices>
</domain>
```

Example 2

An XML configuration file of x86_64 VM, which contains basic elements and bus elements. The following is a configuration example:

```
<domain type='kvm'>
  <name>openEulerVM</name>
  <memory unit='KiB'>8388608</memory>
  <currentMemory unit='KiB'>8388608</currentMemory>
  <vcpu placement='static'>4</vcpu>
  <iothreads>1</iothreads>
  <os>
    <type arch='x86 64' machine='pc-i440fx-4.0'>hvm</type>
  </os>
  <features>
    <acpi/>
  </features>
  <cpu mode='host-passthrough' check='none'>
    <topology sockets='2' cores='2' threads='1'/>
  </cpu>
  <clock offset='utc'/>
  <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' iothread='1'/>
      <source file='/mnt/openEuler-image.qcow2'/>
      <target dev='vda' bus='virtio'/>
      <boot order='1'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x08' function='0x0'/>
    </disk>
    <controller type='scsi' index='0' model='virtio-scsi'>
    </controller>
    <controller type='virtio-serial' index='0'>
    </controller>
    <controller type='usb' index='0' model='ehci'>
    </controller>
    <controller type='sata' index='0'>
    </controller>
    <controller type='pci' index='0' model='pci-root'/>
    <interface type='bridge'>
      <mac address='52:54:00:c1:c4:23'/>
      <source bridge='virbr0'/>
      <model type='virtio'/>
    </interface>
    <serial type='pty'>
      <target type='isa-serial' port='0'>
        <model name='isa-serial'/>
      </target>
    </serial>
    <console type='pty'>
```

```
    <target type='serial' port='0'/>
   </console>
   <input type='tablet' bus='usb'>
    <address type='usb' bus='0' port='1'/>
   </input>
   <input type='keyboard' bus='usb'>
    <address type='usb' bus='0' port='2'/>
   </input>
   <input type='mouse' bus='ps2'/>
   <input type='keyboard' bus='ps2'/>
   <graphics type='vnc' port='-1' autoport='yes' listen='0.0.0.0'>
    <listen type='address' address='0.0.0.0'/>
   </graphics>
   <video>
    <model type='vga' vram='16384' heads='1' primary='yes'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0'/>
   </video>
   <memballoon model='virtio'>
   </memballoon>
  </devices>
</domain>
```

3.3 Managing VMs

3.3.1 VM Life Cycle

3.3.1.1 Introduction

Overview

To leverage hardware resources and reduce costs, users need to properly manage VMs. This section describes basic operations during the VM lifecycle, such as creating, using, and deleting VMs.

VM Status

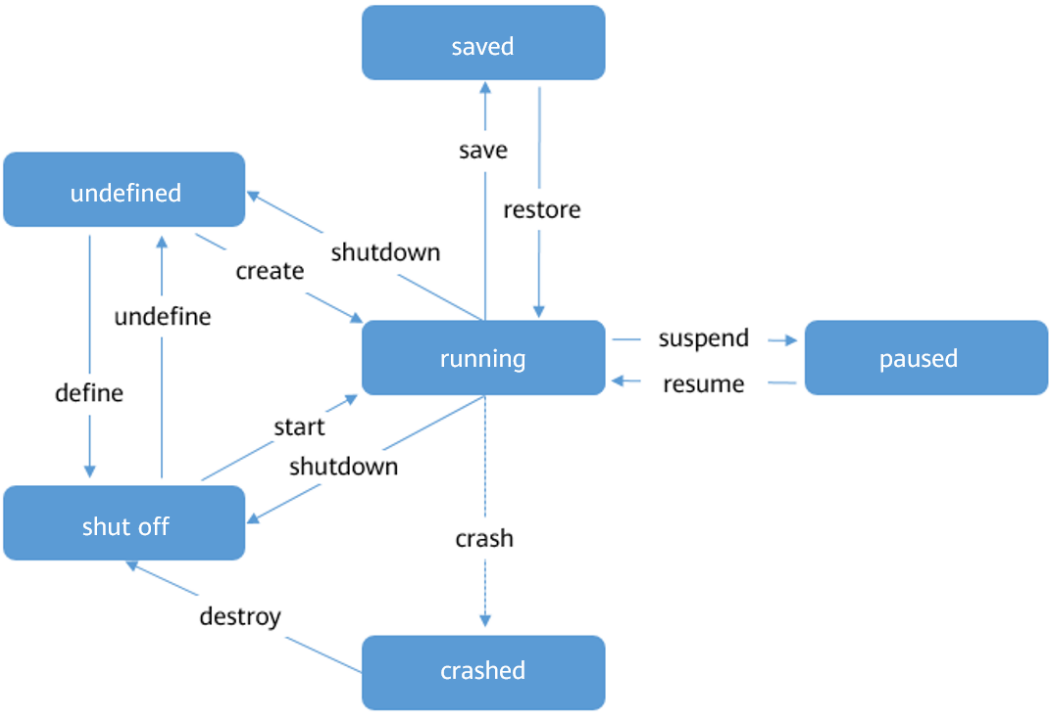
A VM can be in one of the following status:

- **undefined**: The VM is not defined or created. That is, libvirt considers that the VM does not exist.
- **shut off**: The VM has been defined but is not running, or the VM is terminated.
- **running**: The VM is running.
- **paused**: The VM is suspended and its running status is temporarily stored in the memory. The VM can be restored to the running status.
- **saved**: Similar to the **paused** status, the running state is stored in a persistent storage medium and can be restored to the running status.
- **crashed**: The VM crashes due to an internal error and cannot be restored to the running status.

Status Transition

VMs in different status can be converted, but certain rules must be met. Figure 3-2 describes the common rules for transiting the VM status.

Figure 3-2 Status transition diagram



VM ID

In libvirt, a created VM instance is called a **domain**, which describes the configuration information of resources such as the CPU, memory, network device, and storage device of the VM. On a host, each domain has a unique ID, which is represented by the VM **Name**, **UUID**, and **Id**. For details, see Table 3-6. During the VM lifecycle, an operation can be performed on a specific VM by using a VM ID.

Table 3-6 Domain ID description

ID	Description
Name	VM name
UUID	Universally unique identifier
Id	VM running ID **NOTE** The ID is not displayed for a powered off VM.

NOTE

Run the **virsh** command to query the VM ID and UUID. For details, see 3.3.3 Querying VM Information.

3.3.1.2 Management Commands

Overview

You can use the **virsh** command tool to manage the VM lifecycle. This section describes the commands related to the lifecycle.

Prerequisites

- Before performing operations on a VM, you need to query the VM status to ensure that the operations can be performed. For details about the conversion between status, see Status Transition in 3.3.1.1 Introduction.
- You have administrator rights.
- The VM XML configuration files are prepared.

Command Usage

You can run the **virsh** command to manage the VM lifecycle. The command format is as follows:

```
virsh <operate> <obj> <options>
```

The parameters are described as follows:

- *operate*: manages VM lifecycle operations, such as creating, deleting, and starting VMs.
- *obj*: specifies the operation object, for example, the VM to be operated.
- *options*: command option. This parameter is optional.

Table 3-7 describes the commands used for VM lifecycle management. *VMInstanse* indicates the VM name, VM ID, or VM UUID, *XMLFile* indicates the XML configuration file of the VM, and *DumpFile* indicates the dump file. Change them based on the site requirements.

Table 3-7 VM Lifecycle Management Commands

Command	Description
virsh define <*XMLFile*>	Define a persistent VM. After the definition is complete, the VM is shut down and is considered as a **domain** instance.
virsh create <*XMLFile*>	Create a temporary VM. After the VM is created, it is in the running status.
virsh start <*VMInstanse*>	Start the VM.
virsh shutdown <*VMInstanse*>	Shut down the VM. Start the VM shutdown process. If the VM fails to be shut down, forcibly stop it.
virsh destroy <*VMInstanse*>	Forcibly stop the VM.
virsh reboot	Reboot the VM.

Command	Description
<*VMInstanse*>	
virsh save <*VMInstanse*> <*DumpFile*>	Dump the VM running status to a file.
virsh restore <*DumpFile*>	Restore the VM from the VM status dump file.
virsh suspend <*VMInstanse*>	Suspend the VM to make the VM in the paused status.
virsh resume <*VMInstanse*>	Resume the VM and restore the VM in the paused status to the running status.
virsh undefine <*VMInstanse*>	After a persistent VM is destroyed, the VM lifecycle ends and no more operations can be performed on the VM.

3.3.1.3 Example

This section provides examples of commands related to VM life cycle management.

- Create a VM.

  The VM XML configuration file is **openEulerVM.xml**. The command and output are as follows:

  ```
  # virsh define openEulerVM.xml
  Domain openEulerVM defined from openEulerVM.xml
  ```

- Start a VM.

  Run the following command to start the *openEulerVM*:

  ```
  # virsh start openEulerVM
  Domain openEulerVM started
  ```

- Reboot a VM.

  Run the following command to reboot the *openEulerVM*:

  ```
  # virsh reboot openEulerVM
  Domain openEulerVM is being rebooted
  ```

- Shut down a VM.

  Run the following command to shut down the *openEulerVM*:

  ```
  # virsh shutdown openEulerVM
  Domain openEulerVM is being shutdown
  ```

- Destroy a VM.
  - If the **nvram** file is not used during the VM startup, run the following command to destroy the VM:

    ```
    # virsh undefine <VMInstanse>
    ```

  - If the **nvram** file is used during the VM startup, run the following command to specify the **nvram** processing policy when destroying the VM:

    ```
    # virsh undefine <VMInstanse> <strategy>
    ```

    strategy indicates the policy for destroying a VM. The values can be:

    --nvram: delete the corresponding **nvram** file when destroying a VM.

--**keep-nvram**: destroy a VM but retain the corresponding **nvram** file.

For example, to delete the *openEulerVM* and its **nvram** file, run the following command:

```
# virsh undefine openEulerVM --nvram
Domain openEulerVM has been undefined
```

3.3.2 Modify VM Configurations Online

Overview

After a VM is created, users can modify VM configurations. This process is called online modification of VM configuration. After the configuration is modified online, the new VM configuration file is persistent and takes effect after the VM is shut down and restarted.

The format of the command for modifying VM configuration is as follows:

```
virsh edit <VMInstance>
```

The **virsh edit** command is used to edit the XML configuration file corresponding to **domain** to update VM configuration. **virsh edit** uses the **vi** program as the default editor. You can specify the editor type by modifying the environment variable *EDITOR* or *VISUAL*. By default, **virsh edit** preferentially uses the text editor specified by the environment variable *VISUAL*.

Procedure

Step 1 (Optional) Set the editor of the **virsh edit** command to **vim**.

```
# export VISUAL=vim
```

Step 2 Run the **virsh edit** command to open the XML configuration file of the *openEulerVM*.

```
# virsh edit openEulerVM
```

Step 3 Modify the VM configuration file.

Step 4 Save the VM configuration file and exit.

Step 5 Reboot the VM for the configuration to take effect.

```
# virsh reboot openEulerVM
```

----End

3.3.3 Querying VM Information

Overview

The libvirt provides a set of command line tools to query VM information. This section describes how to use commands to obtain VM information.

Prerequisites

To query VM information, the following requirements must be met:

- The libvirtd service is running.

- Only the administrator has the permission to execute command line.

Querying VM Information on a Host.

- Query the list of running and paused VMs on a host.

```
# virsh list
```

For example, the following command output indicates that three VMs exist on the host. **openEulerVM01** and **openEulerVM02** are running, and **openEulerVM03** is paused.

```
 Id    Name                           State
----------------------------------------------------
 39    openEulerVM01                   running
 40    openEulerVM02                   running
 69    openEulerVM03                   paused
```

- Query the list of VM information defined on a host.

```
# virsh list --all
```

For example, the following command output indicates that four VMs are defined on the current host. **openEulerVM01** is running, **openEulerVM02** is paused, and **openEulerVM03** and **openEulerVM04** are shut down.

```
 Id    Name                           State
----------------------------------------------------
 39    openEulerVM01                   running
 69    openEulerVM02                   paused
 -     openEulerVM03                   shut off
 -     openEulerVM04                   shut off
```

Querying Basic VM Information

Libvirt component provides a group of commands for querying the VM status, including the VM running status, device information, and scheduling attributes. For details, see Table 3-8.

Table 3-8 Querying basic VM information

Information to be queried	Command line	Description
Basic information	**virsh dominfo** <*VMInstance*>	The information includes the VM ID, UUID, and VM specifications.
Current status	**virsh domstate** <*VMInstance*>	You can use the **--reason** option to query the reason why the VM changes to the current status.
Scheduling information	**virsh schedinfo** <*VMInstance*>	The information includes the vCPU share.
Number of vCPUs	**virsh vcpucount** <*VMInstance*>	Number of vCPUs of the VM.
Virtual block device status	**virsh domblkstat** <*VMInstance*>	To query the name of a block device, run the **virsh domblklist** command.
vNIC status	**virsh domifstat** <*VMInstance*>	To query the NIC name, run the **virsh**

Information to be queried	Command line	Description
		domiflist command.
I/O thread	**virsh iothreadinfo** *<VMInstance>*	VM I/O thread and CPU affinity.

Example

- Run the **virsh dominfo** command to query the basic information about a created VM. The query result shows that the VM ID is **5**, UUID is **ab472210-db8c-4018-9b3e-fc5319a769f7**, memory size is 8 GiB, and the number of vCPUs is 4.

```
# virsh dominfo openEulerVM
Id:           5
Name:         openEulerVM
UUID:         ab472210-db8c-4018-9b3e-fc5319a769f7
OS Type:      hvm
State:        running
CPU(s):       4
CPU time:     6.8s
Max memory:   8388608 KiB
Used memory:  8388608 KiB
Persistent:   no
Autostart:    disable
Managed save:  no
Security model: none
Security DOI:  0
```

- Run the **virsh domstate** command to query the VM status. The query result shows that VM **openEulerVM** is running.

```
# virsh domstate openEulerVM
running
```

- Run **virsh schedinfo** to query the VM scheduling information. The query result shows that the CPU reservation share of the VM is 1024.

```
# virsh schedinfo openEulerVM
Scheduler     : posix
cpu shares    : 1024
vcpu period   : 100000
vcpu quota    : -1
emulator period: 100000
emulator quota : -1
global period  : 100000
global quota   : -1
iothread period: 100000
iothread_quota : -1
```

- Run the **virsh vcpucount** command to query the number of vCPUs. The query result shows that the VM has four CPUs.

```
# virsh vcpucount openEulerVM
maximum     live         4
current     live         4
```

- Run the **virsh domblklist** command to query the VM disk information. The query result shows that the VM has two disks. sda is a virtual disk in qcow2 format, and sdb is a cdrom device.

```
 # virsh domblklist openEulerVM
 Target   Source
------------------------------------------------------------------------
 sda     /home/openeuler/vm/openEuler aarch64.qcow2
 sdb     /home/openeuler/vm/openEuler-20.03-LTS-aarch64-dvd.iso
```

- Run the **virsh domiflist** command to query the VM NIC information. The query result shows that the VM has one NIC, the backend is vnet0, which is on the br0 bridge of the host. The MAC address is 00:05:fe:d4:f1:cc.

```
# virsh domiflist openEulerVM
Interface  Type      Source    Model      MAC
-------------------------------------------------------
vnet0     bridge    br0       virtio     00:05:fe:d4:f1:cc
```

- Run the **virsh iothreadinfo** command to query the VM I/O thread information. The query result shows that the VM has five I/O threads, which are scheduled on physical CPUs 7-10.

```
# virsh iothreadinfo openEulerVM
 IOThread ID    CPU Affinity
---------------------------------------------------
 3              7-10
 4              7-10
 5              7-10
 1              7-10
 2              7-10
```

3.3.4 Logging In to a VM

This section describes how to log in to a VM using VNC.

3.3.4.1 Logging In Using VNC Passwords

Overview

After the OS is installed on a VM, you can remotely log in to the VM using VNC to manage the VM.

Prerequisites

Before logging in to a VM using a client, such as RealVNC or TightVNC, ensure that:

- You have obtained the IP address of the host where the VM resides.
- The environment where the client resides can access the network of the host.
- You have obtained the VNC listening port of the VM. This port is automatically allocated when the client is started. Generally, the port number is **5900 + x** (*x* is a positive integer and increases in ascending order based on the VM startup sequence. **5900** is invisible to users.)

- If a password has been set for the VNC, you also need to obtain the VNC password of the VM.

NOTE

To set a password for the VM VNC, edit the XML configuration file of the VM. That is, add the **passwd** attribute to the **graphics** element and set the attribute value to the password to be configured. For example, to set the VNC password of the VM to **n8VfjbFK**, configure the XML file as follows:

```
<graphics type='vnc' port='5900' autoport='yes' listen='0.0.0.0' keymap='en-us'
passwd='n8VfjbFK'>
  <listen type='address' address='0.0.0.0'/>
</graphics>
```

Procedure

Step 1 Query the VNC port number used by the VM. For example, if the VM name is *openEulerVM*, run the following command:

```
# virsh vncdisplay openEulerVM
:3
```

NOTE

To log in to the VNC, you need to configure firewall rules to allow the connection of the VNC port. The reference command is as follows, where *X* is **5900 + Port number**, for example, **5903**.

```
firewall-cmd --zone=public --add-port=X/tcp
```

Step 2 Start the VncViewer software and enter the IP address and port number of the host. The format is **host IP address:port number**, for example, **10.133.205.53:3**.

Step 3 Click **OK** and enter the VNC password (optional) to log in to the VM VNC.

----End

3.3.4.2 Configuring VNC TLS Login

Overview

By default, the VNC server and client transmit data in plaintext. Therefore, the communication content may be intercepted by a third party. To improve security, openEuler allows the VNC server to configure the Transport Layer Security (TLS) mode for encryption and authentication. TLS implements encrypted communication between the VNC server and client to prevent communication content from being intercepted by third parties.

NOTE

- To use the TLS encryption authentication mode, the VNC client must support the TLS mode (for example, TigerVNC). Otherwise, the VNC client cannot be connected.
- The TLS encryption authentication mode is configured at the host level. After this feature is enabled, the TLS encryption authentication mode is enabled for the VNC clients of all VMs running on the host.

Procedure

To enable the TLS encryption authentication mode for the VNC, perform the following steps:

Step 1 Log in to the host where the VNC server resides, and edit the corresponding configuration items in the **/etc/libvirt/qemu.conf** configuration file of the server. The configuration is as follows:

```
vnc_listen = "x.x.x.x"                # "x.x.x.x" indicates the listening IP address
of the VNC. Set this parameter based on the site requirements. The VNC server allows
only the connection requests from clients whose IP addresses are in this range.
vnc_tls = 1                    # If this parameter is set to 1, VNC TLS is enabled.
vnc tls x509 cert dir = "/etc/pki/libvirt-vnc"                      # Specify
/etc/pki/libvirt-vnc as the path for storing the certificate.
vnc tls x509 verify = 1                          #If this parameter is set to 1, the
X509 certificate is used for TLS authentication.
```

Step 2 Create a certificate and a private key file for the VNC. The following uses GNU TLS as an example.

> **NOTE**
>
> To use GNU TLS, install the gnu-utils software package in advance.

1. Create a certificate file issued by the Certificate Authority (CA).

   ```
   # certtool --generate-privkey > ca-key.pem
   ```

2. Create a self-signed public and private key for the CA certificate. *Your organization name* indicates the organization name, which is specified by the user.

   ```
   # cat > ca.info<<EOF
   cn = Your organization name
   ca
   cert_signing_key
   EOF
   # certtool --generate-self-signed \
          --load-privkey ca-key.pem \
          --template ca.info \
          --outfile ca-cert.pem
   ```

   In the preceding generated file, **ca-cert.pem** is the generated CA public key, and **ca-key.pem** is the generated CA private key. The CA must keep them properly to prevent disclosure.

3. Issue a certificate to the VNC server. **Client Organization Name** indicates the actual service name, for example, **cleint.foo.com**. Set this parameter based on the site requirements.

   ```
   # cat > server.info<<EOF
   cn = Server Organization Name
   tls_www_server
   encryption_key
   signing_key
   EOF
   # certtool --generate-privkey > server-key.pem
   # certtool --generate-certificate \
            --load-ca-certificate ca-cert.pem \
            --load-ca-privkey ca-key.pem \
            --load-privkey server-key.pem \
            --template server.info \
            --outfile server-cert.pem
   ```

   In the preceding generated file, **server-key.pem** is the private key of the VNC server, and **server-cert.pem** is the public key of the VNC server.

4. Issue a certificate to the VNC client.

```
# cat > client.info<<EOF
cn = Client Organization Name
tls_www_client
encryption_key
signing_key
EOF
# certtool --generate-privkey > client-key.pem
# certtool --generate-certificate \
         --load-ca-certificate ca-cert.pem \
         --load-ca-privkey ca-key.pem \
         --load-privkey client-key.pem \
         --template client.info \
         --outfile client-cert.pem
```

In the preceding generated file, **client-key.pem** is the private key of the VNC client, and **client-cert.pem** is the public key of the VNC client. The generated public and private key pairs need to be copied to the VNC client.

Step 3 Shut down the VM to be logged in to and restart the libvirtd service on the host where the VNC server resides.

```
# systemctl restart libvirtd
```

Step 4 Save the generated server certificate to the specified directory on the VNC server and grant the read and write permissions on the certificate only to the current user.

```
# sudo mkdir -m 750 /etc/pki/libvirt-vnc
# cp ca-cert.pem /etc/pki/libvirt-vnc/ca-cert.pem
# cp server-cert.pem /etc/pki/libvirt-vnc/server-cert.pem
# cp server-key.pem /etc/pki/libvirt-vnc/server-key.pem
# chmod 0600 /etc/pki/libvirt-vnc/*
```

Step 5 Copy the generated client certificates **ca-cert.pem**, **client-cert.pem**, and **client-key.pem** to the VNC client. After the TLS certificate of the VNC client is configured, you can use VNC TLS to log in to the VM.

NOTE

- For details about how to configure the VNC client certificate, see the usage description of each client.
- For details about how to log in to the VM, see Logging In Using VNC Passwords.

----End

3.4 VM Live Migration

3.4.1 Introduction

Overview

When a VM is running on a physical machine, the physical machine may be overloaded or underloaded due to uneven resource allocation. In addition, operations such as hardware replacement, software upgrade, networking adjustment, and troubleshooting are performed on the physical machine. Therefore, it is important to complete these operations without interrupting services. The VM live migration technology implements load balancing or the preceding operations on the premise of service continuity, improving user experience and

work efficiency. VM live migration is to save the running status of the entire VM and quickly restore the VM to the original or even different hardware platforms. After the VM is restored, it can still run smoothly without any difference to users. Because the VM data can be stored on the current host or a shared remote storage device, openEuler supports shared and non-shared storage live migration.

3.4.2 Application Scenarios

Shared and non-shared storage live migration applies to the following scenarios:

- When a physical machine is faulty or overloaded, you can migrate the running VM to another physical machine to prevent service interruption and ensure normal service running.
- When most physical machines are underloaded, migrate and integrate VMs to reduce the number of physical machines and improve resource utilization.
- When the hardware of a physical server becomes a bottleneck, such as the CPU, memory, and hard disk, replace the hardware with better performance or add devices. However, you cannot stop the VM or stop services.
- Server software upgrade, such as virtualization platform upgrade, allows the VM to be live migrated from the old virtualization platform to the new one.

Non-shared storage live migration can also be used in the following scenarios:

- If a physical machine is faulty and the storage space is insufficient, migrate the running VM to another physical machine to prevent service interruption and ensure normal service running.
- When the storage device of the physical machine is aged, the performance cannot support the current service data processing and becomes the bottleneck of the system performance. In this case, a storage device with higher performance needs to be used, but the VM cannot be shut down or stopped. The running VM needs to be migrated to a physical machine with better performance.

3.4.3 Precautions and Restrictions

- During the live migration, ensure that the network is in good condition. If the network is interrupted, live migration is suspended until the network is recovered. If the network connection times out, live migration fails.
- During the migration, do not perform operations such as VM life cycle management and VM hardware device management.
- During VM migration, ensure that the source and destination servers are not powered off or restarted unexpectedly. Otherwise, the live migration fails or the VM may be powered off.
- Do not shut down, restart, or restore the VM during the migration. Otherwise, the live migration may fail. If you perform live migration when the VM is rebooted in ACPI mode, the VM will be shut down.
- Only homogeneous live migration is supported. That is, the CPU models of the source and destination must be the same.
- A VM can be successfully migrated across service network segments. However, network exceptions may occur after the VM is migrated to the destination. To prevent this problem, ensure that the service network segments to be migrated are the same.
- If the number of vCPUs on the source VM is greater than that on the destination physical machine, the VM performance will be affected after the migration. Ensure that the

number of vCPUs on the destination physical machine is greater than or equal to that on the source VM.

Precautions for live migration of non-shared storage:

- The source and destination cannot be the same disk image file. You need to perform special processing on such migration to prevent image damage caused by data overwriting.
- Shared disks cannot be migrated. You need to perform foolproof operations on such migration.
- The destination image supports only files and does not support raw devices. You need to perform foolproof processing on the migration of raw devices.
- The size and number of disk images created on the destination must be the same as those on the source. Otherwise, the migration fails.
- In hybrid migration scenarios, the disks to be migrated must not include shared and read-only disks.

3.4.4 Live Migration Operations

Prerequisites

- Before live migration, ensure that the source and destination hosts can communicate with each other and have the same resource permissions. That is, the source and destination hosts can access the same storage and network resources.
- Before VM live migration, perform a health check on the VM and ensure that the destination host has sufficient CPU, memory, and storage resources.

(Optional) Setting Live Migration Parameters

Before live migration, run the **virsh migrate-setmaxdowntime** command to specify the maximum tolerable downtime during VM live migration. This is an optional configuration item.

For example, to set the maximum downtime of the VM named **openEulerVM** to **500 ms**, run the following command:

```
# virsh migrate-setmaxdowntime openEulerVM 500
```

In addition, you can run the **virsh migrate-setspeed** command to limit the bandwidth occupied by VM live migration. This prevents VM live migration from occupying too much bandwidth and affecting other VMs or services on the host. This operation is also optional for live migration.

For example, to set the live migration bandwidth of the VM named **openEulerVM** to **500 Mbit/s**, run the following command:

```
# virsh migrate-setspeed openEulerVM --bandwidth 500
```

You can run the **migrate-getspeed** command to query the maximum bandwidth during VM live migration.

```
# virsh migrate-getspeed openEulerVM
500
```

Live Migration Operations (Shared Storage Scenario)

Step 1 Check whether the storage device is shared.

```
# virsh domblklist <VMInstanse>
 Target   Source
------------------------------------------
sda     /dev/mapper/open euleros disk
sdb     /mnt/nfs/images/openeuler-test.qcow2
```

Run the **virsh domblklist** command to query the storage device information of the VM. For example, the preceding query result shows that the VM is configured with two storage devices: sda and sdb. Then, check whether the backend storage of the two devices is local storage or remote storage, if all storage devices are on the remote shared storage, the VM is a shared storage VM. Otherwise, the VM is a non-shared storage VM.

Step 2 Run the following command for VM live migration:

For example, run the **virsh migrate** command to migrate VM **openEulerVM** to the destination host.

```
# virsh migrate --live --unsafe openEulerVM qemu+ssh://<destination-host-ip>/system
```

<destination-host-ip> indicates the IP address of the destination host. Before live migration, SSH authentication must be performed to obtain the source host management permission.

In addition, the **virsh migrate** command has the **--auto-converge** and **--timeout** sub-options to ensure successful migration.

Related sub-options:

The **--unsafe** command forcibly performs live migration and skips the security check step.

The **--auto-converge** command reduces the CPU frequency to ensure that the live migration process can be converged.

The **--timeout** command specifies the live migration timeout period. If the live migration exceeds the specified period, the VM is forcibly suspended to reduce the live migration.

Step 3 After the live migration is complete, the VM is running properly on the destination host.

----End

Live Migration Operations (Non-Shared Storage Scenario)

Step 1 Query the VM storage device list to ensure that the VM uses non-shared storage.

For example, the **virsh domblklist** command output shows that the VM to be migrated has a disk sda in qcow2 format. The XML configuration of sda is as follows:

```
   <disk type='file' device='disk'>
     <driver name='qemu' type='qcow2' cache='none' io='native'/>
     <source file='/mnt/sdb/openeuler/openEulerVM.qcow2'/>
     <target dev='sda' bus='scsi'/>
     <address type='drive' controller='0' bus='0' target='0' unit='0'/>
   </disk>
```

Before live migration, create a virtual disk file in the same disk directory on the destination host. Ensure that the disk format and size are the same.

```
# qemu-img create -f qcow2 /mnt/sdb/openeuler/openEulerVM.qcow2 20G
```

Step 2 Run the **virsh migrate** command on the source to perform live migration. During the migration, the storage is also migrated to the destination.

```
# virsh migrate --live  --unsafe --copy-storage-all --migrate-disks sda \
openEulerVM qemu+ssh://<dest-host-ip>/system
```

Step 3 After the live migration is complete, the command output indicates that the VM is running properly on the destination host and the storage device is migrated to the destination host.

----End

3.5 System Resource Management

The **libvirt** command manages VM system resources, such as vCPU and virtual memory resources.

Before you start:

- Ensure that the libvirtd daemon is running on the host.
- Run the **virsh list --all** command to check that the VM has been defined.

3.5.1 Managing vCPU

3.5.1.1 CPU Shares

Overview

In a virtualization environment, multiple VMs on the same host compete for physical CPUs. To prevent some VMs from occupying too many physical CPU resources and affecting the performance of other VMs on the same host, you need to balance the vCPU scheduling of VMs to prevent excessive competition for physical CPUs.

The CPU share indicates the total capability of a VM to compete for physical CPU computing resources. You can set **cpu_shares** to specify the VM capacity to preempt physical CPU resources. The value of **cpu_shares** is a relative value without a unit. The CPU computing resources obtained by a VM are the available computing resources of physical CPUs (excluding reserved CPUs) allocated to VMs based on the CPU shares. Adjust the CPU shares to ensure the service quality of VM CPU computing resources.

Procedure

Change the value of **cpu_shares** allocated to the VM to balance the scheduling between vCPUs.

- Check the current CPU share of the VM.

```
# virsh schedinfo <VMInstance>
Scheduler     : posix
cpu shares    : 1024
vcpu period   : 100000
vcpu quota    : -1
emulator period: 100000
emulator_quota : -1
```

```
global period  : 100000
global quota   : -1
iothread period: 100000
iothread_quota : -1
```

- Online modification: Run the **virsh schedinfo** command with the **--live** parameter to modify the CPU share of a running VM.

  ```
  # virsh schedinfo <VMInstance> --live cpu_shares=<number>
  ```

  For example, to change the CPU share of the running *openEulerVM* from **1024** to **2048**, run the following commands:

  ```
  # virsh schedinfo openEulerVM --live cpu shares=2048
  Scheduler     : posix
  cpu shares    : 2048
  vcpu period   : 100000
  vcpu quota    : -1
  emulator period: 100000
  emulator quota : -1
  global period  : 100000
  global quota   : -1
  iothread period: 100000
  iothread_quota : -1
  ```

  The modification of the **cpu_shares** value takes effect immediately. The running time of the *openEulerVM* is twice the original running time. However, the modification will become invalid after the VM is shut down and restarted.

- Permanent modification: Run the **virsh schedinfo** command with the **--config** parameter to change the CPU share of the VM in the libvirt internal configuration.

  ```
  # virsh schedinfo <VMInstance> --config cpu_shares=<number>
  ```

  For example, run the following command to change the CPU share of *openEulerVM* from **1024** to **2048**:

  ```
  # virsh schedinfo openEulerVM --config cpu shares=2048
  Scheduler     : posix
  cpu shares    : 2048
  vcpu period   : 0
  vcpu quota    : 0
  emulator period: 0
  emulator quota : 0
  global period  : 0
  global quota   : 0
  iothread period: 0
  iothread_quota : 0
  ```

  The modification on **cpu_shares** does not take effect immediately. Instead, the modification takes effect after the *openEulerVM* is started next time and takes effect permanently. The running time of the *openEulerVM* is twice that of the original VM.

3.5.1.2 Binding the QEMU Process to a Physical CPU

Overview

You can bind the QEMU main process to a specific physical CPU range, ensuring that VMs running different services do not interfere with adjacent VMs. For example, in a typical cloud computing scenario, multiple VMs run on one physical machine, and they carry diversified services, causing different degrees of resource occupation. To avoid interference of a VM with dense-storage I/O to an adjacent VM, storage processes that process I/O of different VMs

need to be completely isolated. The QEMU main process handles frontend and backend services. Therefore, isolation needs to be implemented.

Procedure

Run the **virsh emulatorpin** command to bind the QEMU main process to a physical CPU.

- Check the range of the physical CPU bound to the QEMU process:

```
# virsh emulatorpin openEulerVM
emulator: CPU Affinity
----------------------------------
      *: 0-63
```

This indicates that the QEMU main process corresponding to VM **openEulerVM** can be scheduled on all physical CPUs of the host.

- Online binding: Run the **vcpu emulatorpin** command with the **--live** parameter to modify the binding relationship between the QEMU process and the running VM.

```
# virsh emulatorpin openEulerVM --live 2-3

# virsh emulatorpin openEulerVM
emulator: CPU Affinity
----------------------------------
      *: 2-3
```

The preceding commands bind the QEMU process corresponding to VM **openEulerVM** to physical CPUs **2** and **3**. That is, the QEMU process is scheduled only on the two physical CPUs. The binding relationship takes effect immediately but becomes invalid after the VM is shut down and restarted.

- Permanent binding: Run the **virsh emulatorpin** command with the **--config** parameter to modify the binding relationship between the VM and the QEMU process in the libvirt internal configuration.

```
# virsh emulatorpin openEulerVM --config 0-3,^1

# virsh emulatorpin euler
emulator: CPU Affinity
----------------------------------
      *: 0,2-3
```

The preceding commands bind the QEMU process corresponding to VM **openEulerVM** to physical CPUs **0**, **2** and **3**. That is, the QEMU process is scheduled only on the three physical CPUs. The modification of the binding relationship does not take effect immediately. Instead, the modification takes effect after the next startup of the VM and takes effect permanently.

3.5.1.3 Adjusting the vCPU Binding Relationship

Overview

The vCPU of a VM is bound to a physical CPU. That is, the vCPU is scheduled only on the bound physical CPU to improve VM performance in specific scenarios. For example, in a NUMA system, vCPUs are bound to the same NUMA node to prevent cross-node memory access and VM performance deterioration. If the vCPU is not bound, by default, the vCPU can be scheduled on any physical CPU. The specific binding policy is determined by the user.

Procedure

Run the **virsh vcpupin** command to adjust the binding relationship between vCPUs and physical CPUs.

- View the vCPU binding information of the VM.

```
# virsh vcpupin openEulerVM
VCPU   CPU Affinity
----------------------
0      0-63
1      0-63
2      0-63
3      0-63
```

This indicates that all vCPUs of VM **openEulerVM** can be scheduled on all physical CPUs of the host.

- Online adjustment: Run the **vcpu vcpupin** command with the **--live** parameter to modify the vCPU binding relationship of a running VM.

```
# virsh vcpupin openEulerVM  --live 0 2-3

# virsh vcpupin euler
VCPU   CPU Affinity
----------------------
0      2-3
1      0-63
2      0-63
3      0-63
```

The preceding commands bind vCPU **0** of VM **openEulerVM** to pCPU **2** and pCPU **3**. That is, vCPU **0** is scheduled only on the two physical CPUs. The binding relationship takes effect immediately but becomes invalid after the VM is shut down and restarted.

- Permanent adjustment: Run the **virsh vcpupin** command with the **--config** parameter to modify the vCPU binding relationship of the VM in the libvirt internal configuration.

```
# virsh vcpupin openEulerVM --config 0 0-3,^1

# virsh vcpupin openEulerVM
VCPU   CPU Affinity
----------------------
0      0,2-3
1      0-63
2      0-63
3      0-63
```

The preceding commands bind vCPU **0** of VM **openEulerVM** to physical CPUs **0**, **2**, and **3**. That is, vCPU **0** is scheduled only on the three physical CPUs. The modification of the binding relationship does not take effect immediately. Instead, the modification takes effect after the next startup of the VM and takes effect permanently.

3.5.2 Managing Virtual Memory

3.5.2.1 Introduction to NUMA

Traditional multi-core computing uses the symmetric multi-processor (SMP) mode. Multiple processors are connected to a centralized memory and I/O bus. All processors can access only the same physical memory. Therefore, the SMP system is also referred to as a uniform memory access (UMA) system. Uniformity means that a processor can only maintain or share

a unique value for each data record in memory at any time. Obviously, the disadvantage of SMP is its limited scalability, because when the memory and the I/O interface are saturated, adding a processor cannot obtain higher performance.

The non-uniform memory access architecture (NUMA) is a distributed memory access mode. In this mode, a processor can access different memory addresses at the same time, which greatly improves concurrency. With this feature, a processor is divided into multiple nodes, each of which is allocated a piece of local memory space. The processors of all nodes can access all physical memories, but the time required for accessing the memory on the local node is much shorter than that on a remote node.

3.5.2.2 Configuring Host NUMA

To improve VM performance, you can specify NUMA nodes for a VM using the VM XML configuration file before the VM is started so that the VM memory is allocated to the specified NUMA nodes. This feature is usually used together with the vCPU to prevent the vCPU from remotely accessing the memory.

Procedure

- Check the NUMA topology of the host.

```
 # numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
node 0 size: 31571 MB
node 0 free: 17095 MB
node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
node 1 size: 32190 MB
node 1 free: 28057 MB
node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 2 size: 32190 MB
node 2 free: 10562 MB
node 3 cpus: 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
node 3 size: 32188 MB
node 3 free: 272 MB
node distances:
node   0   1   2   3
  0:  10  15  20  20
  1:  15  10  20  20
  2:  20  20  10  15
  3:  20  20  15  10
```

- Add the **numatune** field to the VM XML configuration file to create and start the VM. For example, to allocate NUMA node 0 on the host to the VM, configure parameters as follows:

```
 <numatune>
   <memory mode="strict" nodeset="0"/>
 </numatune>
```

If the vCPU of the VM is bound to the physical CPU of **node 0**, the performance deterioration caused by the vCPU accessing the remote memory can be avoided.

NOTE

- The sum of memory allocated to the VM cannot exceed the remaining available memory of the NUMA node. Otherwise, the VM may fail to start.

- You are advised to bind the VM memory and vCPU to the same NUMA node to avoid the performance deterioration caused by vCPU access to the remote memory. For example, bind the vCPU to NUMA node 0 as well.

3.5.2.3 Configuring Guest NUMA

Many service software running on VMs is optimized for the NUMA architecture, especially for large-scale VMs. openEuler provides the Guest NUMA feature to display the NUMA topology in VMs. You can identify the structure to optimize the performance of service software and ensure better service running.

When configuring guest NUMA, you can specify the location of vNode memory on the host to implement memory block binding and vCPU binding so that the vCPU and memory on the vNode are on the same physical NUMA node.

Procedure

After Guest NUMA is configured in the VM XML configuration file, you can view the NUMA topology on the VM. **<numa>** is mandatory for Guest NUMA.

```
<cputune>
  <vcpupin vcpu='0' cpuset='0-3'/>
  <vcpupin vcpu='1' cpuset='0-3'/>
  <vcpupin vcpu='2' cpuset='16-19'/>
  <vcpupin vcpu='3' cpuset='16-19'/>
</cputune>
<numatune>
  <memnode cellid="0" mode="strict" nodeset="0"/>
  <memnode cellid="1" mode="strict" nodeset="1"/>
</numatune>
[...]
<cpu>
  <numa>
    <cell id='0' cpus='0-1' memory='2097152'/>
    <cell id='1' cpus='2-3' memory='2097152'/>
  </numa>
</cpu>
```

NOTE

- **<numa>** provides the NUMA topology function for VMs. **cell id** indicates the vNode ID, **cpus** indicates the vCPU ID, and **memory** indicates the memory size on the vNode.
- If you want to use Guest NUMA to provide better performance, configure <**numatune**> and **<cputune>** so that the vCPU and memory are distributed on the same physical NUMA node.
- **cellid** in **<numatune>** corresponds to **cell id** in **<numa>**. **mode** can be set to **strict** (apply for memory from a specified node strictly. If the memory is insufficient, the application fails.), **preferred** (apply for memory from a node first. If the memory is insufficient, apply for memory from another node), or **interleave** (apply for memory from a specified node in cross mode).; **nodeset** indicates the specified physical NUMA node.
- In **<cputune>**, you need to bind the vCPU in the same **cell id** to the physical NUMA node that is the same as the **memnode**.

3.6 Managing Devices

3.6.1 Configuring a PCIe Controller for a VM

Overview

Thr NIC, disk controller, and PCIe pass-through devices in a VM must be mounted to a PCIe root port. Each root port corresponds to a PCIe slot. The devices mounted to the root port support hot swap, but the root port does not support hot swap. Therefore, users need to consider the hot swap requirements and plan the maximum number of PCIe root ports reserved for the VM. Before the VM is started, the root port is statically configured.

Configuring the PCIe Root, PCIe Root Port, and PCIe-PCI-Bridge

The VM PCIe controller is configured using the XML file. The **model** corresponding to PCIe root, PCIe root port, and PCIe-PCI-bridge in the XML file are **pcie-root**, **pcie-root-port**, and **pcie-to-pci-bridge**, respectively.

- Simplified configuration method

  Add the following contents to the XML file of the VM. Other attributes of the controller are automatically filled by libvirt.

  ```
  <controller type='pci' index='0' model='pcie-root'/>
  <controller type='pci' index='1' model='pcie-root-port'/>
  <controller type='pci' index='2' model='pcie-to-pci-bridge'/>
  <controller type='pci' index='3' model='pcie-root-port'/>
  <controller type='pci' index='4' model='pcie-root-port'/>
  <controller type='pci' index='5' model='pcie-root-port'/>
  ```

  The **pcie-root** and **pcie-to-pci-bridge** occupy one **index** respectively. Therefore, the final **index** is the number of required **root ports + 1**.

- Complete configuration method

  Add the following contents to the XML file of the VM:

  ```
  <controller type='pci' index='0' model='pcie-root'/>
  <controller type='pci' index='1' model='pcie-root-port'>
    <model name='pcie-root-port'/>
    <target chassis='1' port='0x8'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x0'
multifunction='on'/>
  </controller>
  <controller type='pci' index='2' model='pcie-to-pci-bridge'>
    <model name='pcie-pci-bridge'/>
    <address type='pci' domain='0x0000' bus='0x01' slot='0x00' function='0x0'/>
  </controller>
  <controller type='pci' index='3' model='pcie-root-port'>
    <model name='pcie-root-port'/>
    <target chassis='3' port='0x9'/>
    <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
  </controller>
  <controller type='pci' index='3' model='pcie-root-port'>
  ```

  In the preceding contents:

- The **chassis** and **port** attributes of the root port must be in ascending order. Because a PCIe-PCI-bridge is inserted in the middle, the **chassis** number skips **2**, but the **port** numbers are still consecutive.
- The **address function** of the root port ranges from **0*0** to **0*7**.
- A maximum of eight functions can be mounted to each slot. When the slot is full, the slot number increases.

The complete configuration method is complex. Therefore, the simplified one is recommended.

3.6.2 Managing Virtual Disks

Overview

Virtual disk types include virtio-blk, virtio-scsi, and vhost-scsi. virtio-blk simulates a block device, and virtio-scsi and vhost-scsi simulate SCSI devices.

- virtio-blk: It can be used for common system disk and data disk. In this configuration, the virtual disk is presented as **vd[a-z]** or **vd[a-z][a-z]** in the VM.
- virtio-scsi: It is recommended for common system disk and data disk. In this configuration, the virtual disk is presented as **sd[a-z]** or **sd[a-z][a-z]** in the VM.
- vhost-scsi: It is recommended for the virtual disk that has high performance requirements. In this configuration, the virtual disk is presented as **sd[a-z]** or **sd[a-z][a-z]** on the VM.

Procedure

For details about how to configure a virtual disk, see 3.2.4.1 Storage Devices. This section uses the virtio-scsi disk as an example to describe how to attach and detach a virtual disk.

- Attach a virtio-scsi disk.

  Run the **virsh attach-device** command to attach the virtio-scsi virtual disk.

  ```
  # virsh attach-device <VMInstance> <attach-device.xml>
  ```

  The preceding command can be used to attach a disk to a VM online. The disk information is specified in the **attach-device.xml** file. The following is an example of the **attach-device.xml** file:

  ```
  ### attach-device.xml ###
     <disk type='file' device='disk'>
       <driver name='qemu' type='qcow2' cache='none' io='native'/>
       <source file='/path/to/another/qcow2-file'/>
       <backingStore/>
       <target dev='sdb' bus='scsi'/>
       <address type='drive' controller='0' bus='0' target='1' unit='0'/>
     </disk>
  ```

  The disk attached by running the preceding commands becomes invalid after the VM is shut down and restarted. If you need to permanently attach a virtual disk to a VM, run the **virsh attach-device** command with the **--config** parameter.

- Detach a virtio-scsi disk.

  If a disk attached online is no longer used, run the **virsh detach** command to dynamically detach it.

  ```
  # virsh detach-device <VMInstance> <detach-device.xml>
  ```

detach-device.xml specifies the XML information of the disk to be detached, which must be the same as the XML information during dynamic attachment.

3.6.3 Managing vNICs

Overview

The vNIC types include virtio-net, vhost-net, and vhost-user. After creating a VM, you may need to attach or detach a vNIC. openEuler supports NIC hot swap, which can change the network throughput and improve system flexibility and scalability.

Procedure

For details about how to configure a virtual NIC, see 3.2.4.2 Network Device. This section uses the vhost-net NIC as an example to describe how to attach and detach a vNIC.

- Attach the vhost-net NIC.

  Run the **virsh attach-device** command to attach the vhost-net vNIC.

  ```
  # virsh attach-device <VMInstance> <attach-device.xml>
  ```

  The preceding command can be used to attach a vhost-net NIC to a running VM. The NIC information is specified in the **attach-device.xml** file. The following is an example of the **attach-device.xml** file:

  ```
  ### attach-device.xml ###
     <interface type='bridge'>
       <mac address='52:54:00:76:f2:bb'/>
       <source bridge='br0'/>
       <virtualport type='openvswitch'/>
       <model type='virtio'/>
       <driver name='vhost' queues='2'/>
     </interface>
  ```

  The vhost-net NIC attached using the preceding commands becomes invalid after the VM is shut down and restarted. If you need to permanently attach a vNIC to a VM, run the **virsh attach-device** command with the **--config** parameter.

- Detach the vhost-net NIC.

  If a NIC attached online is no longer used, run the **virsh detach** command to dynamically detach it.

  ```
  # virsh detach-device <VMInstance> <detach-device.xml>
  ```

  detach-device.xml specifies the XML information of the vNIC to be detached, which must be the same as the XML information during dynamic attachment.

3.6.4 Configuring a Virtual Serial Port

Overview

In a virtualization environment, VMs and host machines need to communicate with each other to meet management and service requirements. However, in the complex network architecture of the cloud management system, services running on the management plane and VMs running on the service plane cannot communicate with each other at layer 3. As a result, service deployment and information collection are not fast enough. Therefore, a virtual serial port is required for communication between VMs and host machines. You can add serial port

configuration items to the XML configuration file of a VM to implement communication between VMs and host machines.

Procedure

The Linux VM serial port console is a pseudo terminal device connected to the host machine through the serial port of the VM. It implements interactive operations on the VM through the host machine. In this scenario, the serial port needs to be configured in the pty type. This section describes how to configure a pty serial port.

- Add the following virtual serial port configuration items under the **devices** node in the XML configuration file of the VM:

```
<serial type='pty'>
</serial>
<console type='pty'>
  <target type='serial'/>
</console>
```

- Run the **virsh console** command to connect to the pty serial port of the running VM.

```
# virsh console <VMInstance>
```

- To ensure that no serial port message is missed, use the **--console** option to connect to the serial port when starting the VM.

```
# virsh start --console <VMInstance>
```

3.6.5 Managing Device Passthrough

The device passthrough technology enables VMs to directly access physical devices. The I/O performance of VMs can be improved in this way.

Currently, the VFIO passthrough is used. It can be classified into PCI passthrough and SR-IOV passthrough based on device type.

3.6.5.1 PCI Passthrough

PCI passthrough directly assigns a physical PCI device on the host to a VM. The VM can directly access the device. PCI passthrough uses the VFIO device passthrough mode. The PCI passthrough configuration file in XML format for a VM is as follows:

```
<hostdev mode='subsystem' type='pci' managed='yes'>
   <driver name='vfio'/>
   <source>
      <address domain='0x0000' bus='0x04' slot='0x10' function='0x01'/>
   </source>
   <rom bar='off'/>
   <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
</hostdev>
```

Table 3-9 Device configuration items for PCI passthrough

Parameter	Description	Value
hostdev.source.address.domain	Domain ID of the PCI device on the host OS.	≥ 0
hostdev.source.address.bus	Bus ID of the PCI device on	≥ 1

Parameter	Description	Value
	the host OS.	
hostdev.source.address.slot	Device ID of the PCI device on the host OS.	≥ 0
hostdev.source.address.function	Function ID of the PCI device on the host OS.	≥ 0
hostdev.driver.name	Backend driver of PCI passthrough. This parameter is optional.	**vfio** (default value)
hostdev.rom	Specifies whether the VM can access the ROM of the passthrough device.	This parameter can be set to **on** or **off**. The default value is **on**. • **on**: indicates that the VM can access the ROM of the passthrough device. For example, if a VM with a passthrough NIC needs to boot from the preboot execution environment (PXE), or a VM with a passthrough Host Bus Adapter (HBA) card needs to boot from the ROM, you can set this parameter to **on**. • **off**: indicates that the VM cannot access the ROM of the passthrough device.
hostdev.address type	Bus, Device, and Function (BDF) IDs on the guest OS displayed on the PCI device.	[0x03–0x1e] (range of slot ID) Note: • **domain** indicates the domain information, **bus** indicates the bus ID, **slot** indicates the slot ID, and **function** indicates the function. • Except for **slot**, default values of these parameters are **0**. • The first slot **0x00** is occupied by the system, the second slot **0x01** is occupied by the IDE controller and USB controller, and the third slot **0x02** is occupied by

Parameter	Description	Value
		the video. • The last slot **0x1f** is occupied by the PV channel.

NOTE

VFIO passthrough is implemented by IOMMU group. Devices are divided to IOMMU groups based on access control services (ACS) on hardware. Devices in the same IOMMU group can be assigned to only one VM. If multiple functions on a PCI device belong to the same IOMMU group, they can be directly assigned to only one VM as well.

3.6.5.2 SR-IOV Passthrough

Overview

Single Root I/O Virtualization (SR-IOV) is a hardware-based virtualization solution. With the SR-IOV technology, a physical function (PF) can provide multiple virtual functions (VFs), and each VF can be directly assigned to a VM. This greatly improves hardware resource utilization and I/O performance of VMs. A typical application scenario is SR-IOV passthrough for NICs. With the SR-IOV technology, a physical NIC (PF) can function as multiple VF NICs, and then the VFs can be directly assigned to VMs.

NOTE

- SR-IOV requires the support of physical hardware. Before using SR-IOV, ensure that the hardware device to be directly assigned supports SR-IOV and the device driver on the host OS works in SR-IOV mode.
- The following describes how to query the NIC model:

In the following command output, values in the first column indicate the PCI numbers of NICs, and **19e5:1822** indicates the vendor ID and device ID of the NIC.

```
# lspci | grep Ether
05:00.0 Ethernet controller: Device 19e5:1822 (rev 45)
07:00.0 Ethernet controller: Device 19e5:1822 (rev 45)
09:00.0 Ethernet controller: Device 19e5:1822 (rev 45)
0b:00.0 Ethernet controller: Device 19e5:1822 (rev 45)
81:00.0 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network
Connection (rev 01)
81:00.1 Ethernet controller: Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network
Connection (rev 01)
```

Procedure

To configure SR-IOV passthrough for a NIC, perform the following steps:

1. Enable the SR-IOV mode for the NIC.
   a. Ensure that VF driver support provided by the NIC supplier exists on the guest OS. Otherwise, VFs in the guest OS cannot work properly.
   b. Enable the SMMU/IOMMU support in the BIOS of the host OS. The enabling method varies depending on the servers of different vendors. For details, see the help documents of the servers.

c. Configure the host driver to enable the SR-IOV VF mode. The following uses the Hi1822 NIC as an example to describe how to enable 16 VFs.

```
echo 16 > /sys/class/net/ethX/device/sriov_numvfs
```

2. Obtain the PCI BDF information of PFs and VFs.

a. Run the following command to obtain the NIC resource list on the current board:

```
# lspci | grep Eth
03:00.0 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
(4*25GE) (rev 45)
04:00.0 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
(4*25GE) (rev 45)
05:00.0 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
(4*25GE) (rev 45)
06:00.0 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
(4*25GE) (rev 45)
7d:00.0 Ethernet controller: Huawei Technologies Co., Ltd. Device a222 (rev
20)
7d:00.1 Ethernet controller: Huawei Technologies Co., Ltd. Device a222 (rev
20)
7d:00.2 Ethernet controller: Huawei Technologies Co., Ltd. Device a221 (rev
20)
7d:00.3 Ethernet controller: Huawei Technologies Co., Ltd. Device a221 (rev
20)
```

b. Run the following command to view the PCI BDF information of VFs:

```
# lspci | grep "Virtual Function"
03:00.1 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:00.2 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:00.3 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:00.4 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:00.5 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:00.6 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:00.7 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:01.0 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:01.1 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
03:01.2 Ethernet controller: Huawei Technologies Co., Ltd. Hi1822 Family
Virtual Function (rev 45)
```

c. Select an available VF and write its configuration to the VM configuration file based on its BDF information. For example, the bus ID of the device **03:00.1** is **03**, its slot ID is **00**, and its function ID is **1**.

3. Identify and manage the mapping between PFs and VFs.

a. Identify VFs corresponding to a PF. The following uses PF 03.00.0 as an example:

```
# ls -l /sys/bus/pci/devices/0000\:03\:00.0/
```

The following symbolic link information is displayed. You can obtain the VF IDs (virtfnX) and PCI BDF IDs based on the information.

b. Identify the PF corresponding to a VF. The following uses VF 03:00.1 as an example:

```
# ls -l /sys/bus/pci/devices/0000\:03\:00.1/
```

The following symbolic link information is displayed. You can obtain PCI BDF IDs of the PF based on the information.

```
lrwxrwxrwx 1 root root      0 Mar 28 22:44 physfn -> ../0000:03:00.0
```

c. Obtain names of NICs corresponding to the PFs or VFs. For example:

```
# ls /sys/bus/pci/devices/0000:03:00.0/net
eth0
```

d. Set the MAC address, VLAN, and QoS information of VFs to ensure that the VFs are in the **Up** state before passthrough. The following uses VF 03:00.1 as an example. The PF is eth0 and the VF ID is **0**.

```
# ip link set eth0 vf 0 mac 90:E2:BA:21:XX:XX   #Sets the MAC address.
# ifconfig eth0 up
# ip link set eth0 vf 0 rate 100              #Sets the VF outbound rate, in
Mbit/s.
# ip link show eth0                          #Views the MAC address, VLAN ID,
and QoS information to check whether the configuration is successful.
```

4. Mount the SR-IOV NIC to the VM.

When creating a VM, add the SR-IOV passthrough configuration item to the VM configuration file.

```
<interface type='hostdev' managed='yes'>
   <mac address='fa:16:3e:0a:xx:xx'/>
   <source>
      <address type='pci' domain='0x0000' bus='0x06' slot='0x11' function='0x6'/>
   </source>
   <vlan>
      <tag id='1'/>
   </vlan>
</interface>
```

Table 3-10 SR-IOV configuration options

Parameter	Description	Value
hostdev.managed	Two modes for libvirt to process PCI devices.	**no**: default value. The passthrough device is managed by the user. **yes**: The passthrough device is managed by libvirt. Set this parameter to **yes** in the SR-IOV passthrough scenario.
hostdev.source.address.bus	Bus ID of the PCI device on the host OS.	≥ 1
hostdev.source.address.slot	Device ID of the PCI device on the host OS.	≥ 0
hostdev.source.address.function	Function ID of the PCI device on the host OS.	≥ 0

NOTE

Disabling the SR-IOV function:

To disable the SR-IOV function after the VM is stopped and no VF is in use, run the following command:

The following uses the Hi1822 NIC (corresponding network interface name: eth0) as an example:

```
echo 0 > /sys/class/net/eth0/device/sriov_numvfs
```

3.6.6 Managing VM USB

To facilitate the use of USB devices such as USB key devices and USB mass storage devices on VMs, openEuler provides the USB device passthrough function. Through USB passthrough and hot-swappable interfaces, you can configure USB passthrough devices for VMs, or hot swap USB devices when VMs are running.

3.6.6.1 Configuring USB Controllers

Overview

A USB controller is a virtual controller that provides specific USB functions for USB devices on VMs. To use USB devices on a VM, you must configure USB controllers for the VM. Currently, openEuler supports the following types of USB controllers:

- Universal host controller interface (UHCI): also called the USB 1.1 host controller specification.
- Enhanced host controller interface (EHCI): also called the USB 2.0 host controller specification.
- Extensible host controller interface (xHCI): also called the USB 3.0 host controller specification.

Precautions

- The host server must have USB controller hardware and modules that support USB 1.1, USB 2.0, and USB 3.0 specifications.
- You need to configure USB controllers for the VM by following the order of USB 1.1, USB 2.0, and USB 3.0.
- An xHCI controller has eight ports and can be mounted with a maximum of four USB 3.0 devices and four USB 2.0 devices. An EHCI controller has six ports and can be mounted with a maximum of six USB 2.0 devices. A UHCI controller has two ports and can be mounted with a maximum of two USB 1.1 devices.
- On each VM, only one USB controller of the same type can be configured.
- USB controllers cannot be hot swapped.
- If the USB 3.0 driver is not installed on a VM, the xHCI controller may not be identified. For details about how to download and install the USB 3.0 driver, refer to the official description provided by the corresponding OS distributor.
- To ensure the compatibility of the OS, set the bus ID of the USB controller to **0** when configuring a USB tablet for the VM. The tablet is mounted to the USB 1.1 controller by default.

Configuration Methods

The following describes the configuration items of USB controllers for a VM. You are advised to configure USB 1.1, USB 2.0, and USB 3.0 to ensure the VM is compatible with three types of devices.

The configuration item of the USB 1.1 controller (UHCI) in the XML configuration file is as follows:

```
<controller type='usb' index='0' model='piix3-uhci'>
</controller>
```

The configuration item of the USB 2.0 controller (EHCI) in the XML configuration file is as follows:

```
<controller type='usb' index='1' model='ehci'>
</controller>
```

The configuration item of the USB 3.0 controller (xHCI) in the XML configuration file is as follows:

```
<controller type='usb' index='2' model='nec-xhci'>
</controller>
```

3.6.6.2 Configuring a USB Passthrough Device

Overview

After USB controllers are configured for a VM, a physical USB device on the host can be mounted to the VM through device passthrough for the VM to use. In the virtualization scenario, in addition to static configuration, hot swapping the USB device is supported. That is, the USB device can be mounted or unmounted when the VM is running.

Precautions

- A USB device can be assigned to only one VM.
- A VM with a USB passthrough device does not support live migration.
- VM creation fails if no USB passthrough devices exist in the VM configuration file.
- Forcibly hot removing a USB storage device that is performing read or write operation may damage files in the USB storage device.

Configuration Description

The following describes the configuration items of a USB device for a VM.

Description of the USB device in the XML configuration file:

```
<hostdev mode='subsystem' type='usb' managed='yes'>
   <source>
      <address bus='m' device='n'/>
   </source>
   <address type='usb' bus='x' port='y'/>
</hostdev>
```

- **<address bus='***m***'device='***n***'/>**: *m* indicates the USB bus address on the host, and *n* indicates the device ID.

- **<address type='usb'bus='***x***'port='***y***'>**: indicates that the USB device is to be mounted to the USB controller specified on the VM. *x* indicates the controller ID, which corresponds to the index number of the USB controller configured on the VM. *y* indicates the port address. When configuring a USB passthrough device, you need to set this parameter to ensure that the controller to which the device is mounted is as expected.

Configuration Methods

To configure USB passthrough, perform the following steps:

1. Configure USB controllers for the VM. For details, see 3.6.6.1 Configuring USB Controllers.
2. Query information about the USB device on the host.

   Run the **lsusb** command (the **usbutils** software package needs to be installed) to query the USB device information on the host, including the bus address, device address, device vendor ID, device ID, and product description. For example:

   ```
   # lsusb
   Bus 008 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
   Bus 007 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
   Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
   Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
   Bus 006 Device 002: ID 0bda:0411 Realtek Semiconductor Corp.
   Bus 006 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
   Bus 005 Device 003: ID 136b:0003 STEC
   Bus 005 Device 002: ID 0bda:5411 Realtek Semiconductor Corp.
   Bus 005 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
   Bus 001 Device 003: ID 12d1:0003 Huawei Technologies Co., Ltd.
   Bus 001 Device 002: ID 0bda:5411 Realtek Semiconductor Corp.
   Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
   Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
   ```

3. Prepare the XML description file of the USB device. Before hot removing the device, ensure that the USB device is not in use. Otherwise, data may be lost.
4. Run the hot swapping commands.

   Take a VM whose name is **openEulerVM** as an example. The corresponding configuration file is **usb.xml**.

   - Hot adding of the USB device takes effect only for the current running VM. After the VM is restarted, hot add the USB device again.

     ```
     # virsh attach-device openEulerVM usb.xml --live
     ```

   - Complete persistency configurations for hot adding of the USB device. After the VM is restarted, the USB device is automatically assigned to the VM.

     ```
     # virsh attach-device openEulerVM usb.xml --config
     ```

   - Hot removing of the USB device takes effect only for the current running VM. After the VM is restarted, the USB device with persistency configurations is automatically assigned to the VM.

     ```
     # virsh detach-device openEulerVM usb.xml --live
     ```

   - Complete persistency configurations for hot removing of the USB device.

     ```
     # virsh detach-device openEulerVM usb.xml --config
     ```

3.6.7 Storing Snapshots

Overview

The VM system may be damaged due to virus damage, system file deletion by mistake, or incorrect formatting. As a result, the system cannot be started. To quickly restore a damaged system, openEuler provides the storage snapshot function. openEuler can create a snapshot that records the VM status at specific time points without informing users (usually within a few seconds). The snapshot can be used to restore the VM to the status when the snapshots were taken. For example, a damaged system can be quickly restored with the help of snapshots, which improves system reliability.

NOTE

Currently, storage snapshots can be QCOW2 and RAW images only. Block devices are not supported.

Procedure

To create VM storage snapshots, perform the following steps:

1. Log in to the host and run the **virsh domblklist** command to query the disk used by the VM.

```
# virsh domblklist openEulerVM
 Target   Source
 ------------------------------------------------
 vda      /mnt/openEuler-image.qcow2
```

2. Run the following command to create the VM disk snapshot **openEuler-snapshot1.qcow2**:

```
# virsh snapshot-create-as --domain openEulerVM --disk-only --diskspec
vda,snapshot=external,file=/mnt/openEuler-snapshot1.qcow2 --atomic
Domain snapshot 1582605802 created
```

3. Run the following command to query disk snapshots:

```
# virsh snapshot-list openEulerVM
 Name        Creation Time             State
------------------------------------------------------------
 1582605802  2020-02-25 12:43:22 +0800  disk-snapshot
```

3.7 Best Practices

3.7.1 Performance Best Practices

3.7.1.1 Halt-Polling

Overview

If compute resources are sufficient, the halt-polling feature can be used to enable VMs to obtain performance similar to that of physical machines. If the halt-polling feature is not enabled, the host allocates CPU resources to other processes when the vCPU exits due to idle timeout. When the halt-polling feature is enabled on the host, the vCPU of the VM performs polling when it is idle. The polling duration depends on the actual configuration. If the vCPU

is woken up during the polling, the vCPU can continue to run without being scheduled from the host. This reduces the scheduling overhead and improves the VM system performance.

NOTE

The halt-polling mechanism ensures that the vCPU thread of the VM responds in a timely manner. However, when the VM has no load, the host also performs polling. As a result, the host detects that the CPU usage of the vCPU is high, but the actual CPU usage of the VM is not high.

Instructions

The halt-polling feature is enabled by default. You can dynamically change the halt-polling time of vCPU by modifying the **halt_poll_ns** file. The default value is **500000**, in ns.

For example, to set the polling duration to 400,000 ns, run the following command:

```
# echo 400000 > /sys/module/kvm/parameters/halt_poll_ns
```

3.7.1.2 I/O Thread Configuration

Overview

By default, QEMU main threads handle backend VM read and write operations on the KVM. This causes the following issues:

- VM I/O requests are processed by a QEMU main thread. Therefore, the single-thread CPU usage becomes the bottleneck of VM I/O performance.
- The QEMU global lock (qemu_global_mutex) is used when VM I/O requests are processed by the QEMU main thread. If the I/O processing takes a long time, the QEMU main thread will occupy the global lock for a long time. As a result, the VM vCPU cannot be scheduled properly, affecting the overall VM performance and user experience.

You can configure the I/O thread attribute for the virtio-blk disk or virtio-scsi controller. At the QEMU backend, an I/O thread is used to process read and write requests of a virtual disk. The mapping relationship between the I/O thread and the virtio-blk disk or virtio-scsi controller can be a one-to-one relationship to minimize the impact on the QEMU main thread, enhance the overall I/O performance of the VM, and improve user experience.

Configuration Description

To use I/O threads to process VM disk read and write requests, you need to modify VM configurations as follows:

- Configure the total number of high-performance virtual disks on the VM. For example, set **<iothreads>** to **4** to control the total number of I/O threads.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
    <name>VMName</name>
    <memory>4194304</memory>
    <currentMemory>4194304</currentMemory>
    <vcpu>4</vcpu>
    <iothreads>4</iothreads>
```

- Configure the I/O thread attribute for the virtio-blk disk. **<iothread>** indicates I/O thread IDs. The IDs start from 1 and each ID must be unique. The maximum ID is the value of **<iothreads>**. For example, to allocate I/O thread 2 to the virtio-blk disk, set parameters as follows:

```
<disk type='file' device='disk'>
     <driver name='qemu' type='raw' cache='none' io='native' iothread='2'/>
     <source file='/path/test.raw'/>
     <target dev='vdb' bus='virtio'/>
     <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
</disk>
```

- Configure the I/O thread attribute for the virtio-scsi controller. For example, to allocate I/O thread 2 to the virtio-scsi controller, set parameters as follows:

```
<controller type='scsi' index='0' model='virtio-scsi'>
     <driver iothread='2'/>
     <alias name='scsi0'/>
     <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
</controller>
```

- Bind I/O threads to a physical CPU.

  Binding I/O threads to specified physical CPUs does not affect the resource usage of vCPU threads. **<iothread>** indicates I/O thread IDs, and **<cpuset>** indicates IDs of the bound physical CPUs.

```
<cputune>
<iothreadpin iothread='1' cpuset='1-3,5,7-12' />
<iothreadpin iothread='2' cpuset='1-3,5,7-12' />
</cputune>
```

3.7.1.3 Raw Device Mapping

Overview

When configuring VM storage devices, you can use configuration files to configure virtual disks for VMs, or connect block devices (such as physical LUNs and LVs) to VMs for use to improve storage performance. The latter configuration method is called raw device mapping (RDM). Through RDM, a virtual disk is presented as a small computer system interface (SCSI) device to the VM and supports most SCSI commands.

RDM can be classified into virtual RDM and physical RDM based on backend implementation features. Compared with virtual RDM, physical RDM provides better performance and more SCSI commands. However, for physical RDM, the entire SCSI disk needs to be mounted to a VM for use. If partitions or logical volumes are used for configuration, the VM cannot identify the disk.

Configuration Example

VM configuration files need to be modified for RDM. The following is a configuration example.

- Virtual RDM

  The following is an example of mounting the SCSI disk **/dev/sdc** on the host to the VM as a virtual raw device:

```
<domain type='kvm'>
 <devices>
   ...
   <controller type='scsi' model='virtio-scsi' index='0'/>
   <disk type='block' device='disk'>
      <driver name='qemu' type='raw' cache='none' io='native'/>
      <source dev='/dev/sdc'/>
```

```
      <target dev='sdc' bus='scsi'/>
      <address type='drive' controller='0' bus='0' target='0' unit='0'/>
   </disk>
   ...
 </devices>
</domain>
```

- Physical RDM

  The following is an example of mounting the SCSI disk **/dev/sdc** on the host to the VM as a physical raw device:

```
<domain type='kvm'>
 <devices>
   ...
   <controller type='scsi' model='virtio-scsi' index='0'/>
   <disk type='block' device='lun' rawio='yes'>
      <driver name='qemu' type='raw' cache='none' io='native'/>
      <source dev='/dev/sdc'/>
      <target dev='sdc' bus='scsi'/>
      <address type='drive' controller='0' bus='0' target='0' unit='0'/>
   </disk>
   ...
 </devices>
</domain>
```

3.7.1.4 kworker Isolation and Binding

Overview

kworker is a per-CPU thread implemented by the Linux kernel. It is used to execute workqueue requests in the system. kworker threads will compete for physical core resources with vCPU threads, resulting in virtualization service performance jitter. To ensure that the VM can run stably and reduce the interference of kworker threads on the VM, you can bind kworker threads on the host to a specific CPU.

Instructions

You can modify the **/sys/devices/virtual/workqueue/cpumask** file to bind tasks in the workqueue to the CPU specified by **cpumasks**. Masks in **cpumask** are in hexadecimal format. For example, if you need to bind kworker to CPU0 to CPU7, run the following command to change the mask to **ff**:

```
# echo ff > /sys/devices/virtual/workqueue/cpumask
```

3.7.1.5 HugePage Memory

Overview

Compared with traditional 4 KB memory paging, openEuler also supports 2 MB/1 GB memory paging. HugePage memory can effectively reduce TLB misses and significantly improve the performance of memory-intensive services. openEuler uses two technologies to implement HugePage memory.

- Static HugePages

  The static HugePage requires that a static HugePage pool be reserved before the host OS is loaded. When creating a VM, you can modify the XML configuration file to specify

that the VM memory is allocated from the static HugePage pool. The static HugePage ensures that all memory of a VM exists on the host as the HugePage to ensure physical continuity. However, the deployment difficulty is increased. After the page size of the static HugePage pool is changed, the host needs to be restarted for the change to take effect. The size of a static HugePage can be 2 MB or 1 GB.

- THP

  If the transparent HugePage (THP) mode is enabled, the VM automatically selects available 2 MB consecutive pages and automatically splits and combines HugePages when allocating memory. When no 2 MB consecutive pages are available, the VM selects available 64 KB (AArch64 architecture) or 4 KB (x86_64 architecture) pages for allocation. By using THP, users do not need to be aware of it and 2 MB HugePages can be used to improve memory access performance.

If VMs use static HugePages, you can disable THP to reduce the overhead of the host OS and ensure stable VM performance.

Instructions

- Configure static HugePages.

  Before creating a VM, modify the XML file to configure a static HugePage for the VM.

  ```
  <memoryBacking>
    <hugepages>
      <page size='1' unit='GiB'/>
    </hugepages>
  </memoryBacking>
  ```

  The preceding XML segment indicates that a 1 GB static HugePage is configured for the VM.

  ```
  <memoryBacking>
    <hugepages>
      <page size='2' unit='MiB'/>
    </hugepages>
  </memoryBacking>
  ```

  The preceding XML segment indicates that a 2 MB static HugePage is configured for the VM.

- Configure transparent HugePage.

  Dynamically enable the THP through sysfs.

  ```
  # echo always > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  Dynamically disable the THP.

  ```
  # echo never > /sys/kernel/mm/transparent_hugepage/enabled
  ```

3.7.2 Security Best Practices

3.7.2.1 Libvirt Authentication

Overview

When a user uses libvirt remote invocation but no authentication is performed, any third-party program that connects to the host's network can operate VMs through the libvirt remote invocation mechanism. This poses security risks. To improve system security, openEuler provides the libvirt authentication function. That is, users can remotely invoke a VM through

libvirt only after identity authentication. Only specified users can access the VM, thereby protecting VMs on the network.

Enabling Libvirt Authentication

By default, the libvirt remote invocation function is disabled on openEuler. This following describes how to enable the libvirt remote invocation and libvirt authentication functions.

1. Log in to the host.
2. Modify the libvirt service configuration file **/etc/libvirt/libvirtd.conf** to enable the libvirt remote invocation and libvirt authentication functions. For example, to enable the TCP remote invocation that is based on the Simple Authentication and Security Layer (SASL) framework, configure parameters by referring to the following:

```
#Transport layer security protocol. The value 0 indicates that the protocol is disabled, and the value 1 indicates that the protocol is enabled. You can set the value as needed.
listen tls = 0
#Enable the TCP remote invocation. To enable the libvirt remote invocation and libvirt authentication functions, set the value to 1.
listen tcp = 1
#User-defined protocol configuration for TCP remote invocation. The following uses sasl as an example.
auth_tcp = "sasl"
```

3. Modify the **/etc/sasl2/libvirt.conf** configuration file to set the SASL mechanism and SASLDB.

```
#Authentication mechanism of the SASL framework.
mech list: digest-md5
#Database for storing usernames and passwords
sasldb_path: /etc/libvirt/passwd.db
```

4. Add the user for SASL authentication and set the password. Take the user **userName** as an example. The command is as follows:

```
# saslpasswd2 -a libvirt userName
Password:
Again (for verification):
```

5. Modify the **/etc/sysconfig/libvirtd** configuration file to enable the libvirt listening option.

```
LIBVIRTD_ARGS="--listen"
```

6. Restart the libvirtd service to make the modification to take effect.

```
# systemctl restart libvirtd
```

7. Check whether the authentication function for libvirt remote invocation takes effect. Enter the username and password as prompted. If the libvirt service is successfully connected, the function is successfully enabled.

```
# virsh -c qemu+tcp://192.168.0.1/system
Please enter your authentication name: openeuler
Please enter your password:
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit

virsh #
```

Managing SASL

The following describes how to manage SASL users.

- Query an existing user in the database.

```
# sasldblistusers2 -f /etc/libvirt/passwd.db
user@localhost.localdomain: userPassword
```

- Delete a user from the database.

```
# saslpasswd2 -a libvirt -d user
```

3.7.2.2 qemu-ga

Overview

QEMU guest agent (qemu-ga) is a daemon running within VMs. It allows users on a host OS to perform various management operations on the guest OS through outband channels provided by QEMU. The operations include file operations (open, read, write, close, seek, and flush), internal shutdown, VM suspend (suspend-disk, suspend-ram, and suspend-hybrid), and obtaining of VM internal information (including the memory, CPU, NIC, and OS information).

In some scenarios with high security requirements, qemu-ga provides the blacklist function to prevent internal information leakage of VMs. You can use a blacklist to selectively shield some functions provided by qemu-ga.

NOTE

The qemu-ga installation package is **qemu-guest-agent-*xx*.rpm**. It is not installed on openEuler by default. *xx* indicates the actual version number.

Procedure

To add a qemu-ga blacklist, perform the following steps:

1. Log in to the VM and ensure that the qemu-guest-agent service exists and is running.

```
# systemctl status qemu-guest-agent |grep Active
   Active: active (running) since Wed 2018-03-28 08:17:33 CST; 9h ago
```

2. Query which **qemu-ga** commands can be added to the blacklist:

```
# qemu-ga --blacklist ?
guest-sync-delimited
guest-sync
guest-ping
guest-get-time
guest-set-time
guest-info
...
```

3. Set the blacklist. Add the commands to be shielded to **--blacklist** in the **/usr/lib/systemd/system/qemu-guest-agent.service** file. Use spaces to separate different commands. For example, to add the **guest-file-open** and **guest-file-close** commands to the blacklist, configure the file by referring to the following:

```
[Service]
ExecStart=-/usr/bin/qemu-ga \
     --blacklist=guest-file-open guest-file-close
```

4. Restart the qemu-guest-agent service.

```
# systemctl daemon-reload
# systemctl restart qemu-guest-agent
```

5. Check whether the qemu-ga blacklist function takes effect on the VM, that is, whether the **--blacklist** parameter configured for the qemu-ga process is correct.

```
# ps -ef|grep qemu-ga|grep -E "blacklist=|b="
root      727    1  0 08:17 ?       00:00:00 /usr/bin/qemu-ga
--method=virtio-serial --path=/dev/virtio-ports/org.qemu.guest agent.0
--blacklist=guest-file-open guest-file-close guest-file-read guest-file-write
guest-file-seek guest-file-flush -F/etc/qemu-ga/fsfreeze-hook
```

NOTE

For more information about qemu-ga, visit https://wiki.qemu.org/Features/GuestAgent.

3.7.2.3 sVirt Protection

Overview

In a virtualization environment that uses the discretionary access control (DAC) policy only, malicious VMs running on hosts may attack the hypervisor or other VMs. To improve security in virtualization scenarios, openEuler uses sVirt for protection. sVirt is a security protection technology based on SELinux. It is applicable to KVM virtualization scenarios. A VM is a common process on the host OS. In the hypervisor, the sVirt mechanism labels QEMU processes corresponding to VMs with SELinux labels. In addition to types which are used to label virtualization processes and files, different categories are used to label different VMs. Each VM can access only file devices of the same category. This prevents VMs from accessing files and devices on unauthorized hosts or other VMs, thereby preventing VM escape and improving host and VM security.

Enabling sVirt Protection

Step 1 Enable SELinux on the host.

1. Log in to the host.
2. Enable the SELinux function on the host.
   a. Modify the system startup parameter file **grub.cfg** to set **selinux** to **1**.

   ```
   selinux=1
   ```

   b. Modify **/etc/selinux/config** to set the **SELINUX** to **enforcing**.

   ```
   SELINUX=enforcing
   ```

3. Restart the host.

```
# reboot
```

Step 2 Create a VM where the sVirt function is enabled.

1. Add the following information to the VM configuration file:

```
<seclabel type='dynamic' model='selinux' relabel='yes'/>
```

Or check whether the following configuration exists in the file:

```
<seclabel type='none' model='selinux'/>
```

2. Create a VM.

```
# virsh define openEulerVM.xml
```

Step 3 Check whether sVirt is enabled.

Run the following command to check whether sVirt protection has been enabled for the QEMU process of the running VM. If **svirt_t:s0:c** exists, sVirt protection has been enabled.

```
# ps -eZ|grep qemu |grep "svirt_t:s0:c"
system_u:system_r:svirt_t:s0:c200,c947 11359 ? 00:03:59 qemu-kvm
system_u:system_r:svirt_t:s0:c427,c670 13790 ? 19:02:07 qemu-kvm
```

----End

A Appendix

A.1 Terminology & Acronyms and Abbreviations

For the terminology & acronyms and abbreviation used in this document, see Table A-1 and Table A-2.

Table A-1 Terminology

Term	Description
AArch64	AArch64 is an execution state of the ARMv8 architecture. AArch64 is not only an extension of the 32-bit ARM architecture, but also a brand new architecture in ARMv8 that uses the brand new A64 instruction set.
Domain	A collection of configurable resources, including memory, vCPUs, network devices, and disk devices. Run the VM in the domain. A domain is allocated with virtual resources and can be independently started, stopped, and restarted.
Libvirt	A set of tools used to manage virtualization platforms, including KVM, QEMU, Xen, and other virtualization platforms.
Guest OS	The OS running on the VM.
Host OS	The OS of the virtual physical machine.
Hypervisor	Virtual machine monitor (VMM), is an intermediate software layer that runs between a basic physical server and an OS. It allows multiple OSs and applications to share hardware.
VM	A complete virtual computer system that is constructed by using the virtualization technology and simulating the functions of a complete computer hardware system through software.

Table A-2 Acronyms and abbreviations

Acronyms and abbreviations	Full spelling	Full name	Description
NUMA	Non-Uniform Memory Access Architecture	Non Uniform Memory Access Architecture	NUMA is a memory architecture designed for multi-processor computers. Under NUMA, a processor accesses its own local memory faster than accessing non-local memory (the memory is located on another processor, or the memory shared between processors).
KVM	Kernel-based Virtual Machine	Kernel-based VM	KVM is a kernel-based VM. It is a kernel module of Linux and makes Linux a hypervisor.
OVS	Open vSwitch	Open vSwitch	OVS is a high-quality multi-layer vSwitch that uses the open-source Apache 2.0 license protocol.
QEMU	Quick Emulator	Quick Emulator	QEMU is a general-purpose, open-source emulator that implements hardware virtualization.
SMP	Symmetric Multi-Processor	Symmetric Multi-Processor	SMP is a multi-processor computer hardware architecture. Currently, most processor systems use a symmetric multi-processor architecture. The architecture system has multiple processors, each processor shares the memory subsystem and bus structure.
UEFI	Unified Extensible Firmware Interface	Unified Extensible Firmware Interface	A standard that describes new interfaces in detail. This interface is used by the OS to automatically load the prestart operation environment to an OS.
VM	Virtual Machine	VM	A complete virtual computer system that is constructed by using the virtualization technology and simulating the functions of a complete computer hardware system through software.
VMM	Virtual Machine Monitor	VM Monitor	An intermediate software layer that runs between a basic physical server and an OS. It allows multiple OSs and applications to share hardware.