Kernel-based Virtual Machine (KVM)

Kernel-based Virtual Machine (KVM) is a full virtualization solution built into the Linux kernel. User space tools such as *libvirt* provide a standardized way to interface with virtualization engines, not only KVM but also Xen, OpenVZ and VirtualBox. Graphical tools like *virt-manager* allow for user-friendly management of virtual machines running on the local machine or on remote hosts running *libvirt*.

Preparation

KVM supports a wide range of guests, including Linux, BSD and even Windows. It's architecture allows for near-metal performance. To achieve this *KVM* utilizes hardware assisted virtualization technologies on the host machine's CPU (Intel VT, AMD-V).

On most desktop systems, these virtualization technologies are not enabled out of the box. To check if virtualization technologies are available on your system, use `lscpu`:

```
LC_ALL=C.UTF-8 lscpu | grep Virtualization
```

This will query the CPU's specifications and filter the output for the relevant virtualization feature section.

If the command produces **no output**, it indicates the system's CPU either does not support hardware assisted virtualization (unlikely if the machine's CPU was purchased in the last 10 years) or the feature is disabled in the machine's firmware options.

Hardware assisted virtualization technologies are a pre-requisite to being able to use KVM. Refer to your mainboard's user manual to learn how to enable the option.

> **NOTE:** Settings related to CPU virtualization features can sometimes be found in the "Overclocking" section of your firmware settings.

Also consider enabling IOMMU (Intel VT-d, AMD-Vi) for direct device pass-through.

>

> **NOTE:** On Intel-based systems, unless your kernel has the config option `CONFIG_INTEL_IOMMU_DEFAULT_ON` set (default is unset) you will also have to explicitly add `intel_iommu=on` to your kernel boot parameters.

Installation

The most common way to start using *KVM* is by installing *QEMU*, a generic and open source machine emulator and virtualizer. It utilizes *KVM* to achieve very good performance and can even emulate a different architecture from the one in the host machine. Though *KVM* is the most commonly used hypervisor for *QEMU*, it can also utilize other hypervisors, such as *Xen*, *OpenVZ* or *VirtualBox*.

Arch Linux offers several levels of completeness of the QEMU suite of emulators:

- `qemu-full`: installs the entirety of QEMU tools and libraries, capable of emulating many different systems and architectures.
- `qemu-desktop`: installs the essentials for running a QEMU environment for emulating x86_64 systems to run virtual machines on a desktop computer.
- `qemu-base`: the most basic QEMU environment intended for use on servers and headless environments.

Install the package that applies best for your use-case, e.g. if you plan on running virtual machines on your desktop computer:

```
sudo pacman -S qemu-desktop
```

QEMU itself does not provide graphical tools to set up and manage virtual machines. This is where *libvirt* comes in: it provides APIs for user-facing applications to offer graphical front-ends to users. One such application is *virt-manager* available from Arch repositories:

```
sudo pacman -S virt-manager libvirt
```

By default, only `root` can interface with *KVM* and thus the full system emulator provided by *QEMU* and *libvirt* will require elevated privileges every time you intend to run virtualized environments.

To allow your user to use virtual machines powered by *KVM* and *libvirt* add it to the `libvirt` group:

```
sudo usermod -aG libvirt $USER
```

> **NOTE:** The change will apply after logging out and back in again, or after a reboot. Feel free to continue the steps below until you actually start using QEMU.

Then, enable and start the *libvirt* daemon:

```
sudo systemctl enable --now libvirtd
```

Networking

For virtual machines to have network access, a network bridge needs to be created.

This is relatively easily achieved with `nmcli`, the CLI tool for *NetworkManager*:

```
# Create the new bridge interface
nmcli connection add \
      type bridge \
      ifname br0 \
      con-name "Bridge" \
      stp no

# Determine the current default route network adapter and add it to the bridge
DEFAULT_IF=$(ip r s default | awk '{print $5}')
nmcli connection add \
      type bridge-slave \
      ifname "$DEFAULT_IF" \
      con-name "Ethernet" \
      master br0

# Disable the old wired connection and bring up the bridge
nmcli connection down "Wired Connection 1"
nmcli connection up "Bridge"
```

To easily select the bridge network in a guest's network settings, create a small *libvirt* network XML definition file, e.g. as `br0.xml`:

```
<network>
  <name>br0</name>
  <forward mode='bridge'/>
  <bridge name='br0'/>
</network>
```

Import the definition file and set the network to autostart:

```
sudo virsh -c qemu:///system net-define br0.xml
sudo virsh -c qemu:///system net-autostart br0
```

Storage

Storage under *libvirt* is defined as **pools** which can be any of the following:

- a local directory
- a dedicated disk
- a pre-formatted block device
- an iSCSI target
- an LVM group
- a multi-path device enumerator (RAID device)
- an exported network directory
- a ZFS pool

The default storage pool is defined as a local directory at `/var/lib/libvirt/images`. This is where *libvirt* will store disk images for guests.

> **ATTENTION:** If your storage pool is on a copy-on-write file system, such as btrfs, it is recommended to disable CoW for that directory:
>
> ```
> sudo chattr +C /var/lib/libvirt/images
> ```

Storage pools can be created from within *virt-manager* or by writing custom XML definitions and importing them with `virsh`, in the case that the storage pool type is not exposed through the GUI or you need more fine-grained control over the specifications of the pool.

If you have a remote storage location that holds disk images, i.e. an exported NFS share with ISO images, it's possible to add it as a pool and mount it into the guest without the need to copy the images to your computer first.

The XML definition for a storage pool, named `remote-iso.xml` for example, could look something like this:

```
<pool type="netfs">
  <name>iso</name> <!-- name of the pool -->
  <source>
    <host name="dragonhoard"/> <!-- hostname/IP of remote machine -->
    <dir path="/mnt/user/downloads/ISOs"/> <!-- full path to images on remote machine -->
```

```
    <format type="auto"/>
  </source>
  <target>
    <path>/var/lib/libvirt/images/iso</path> <!-- mount point on local machine -->
  </target>
</pool>
```

In order for *libvirt* to successfully mount the network share, the mount point must exist prior to activating the pool:

```
sudo mkdir -p /var/lib/libvirt/images/iso
```

Finally, import the pool definition and set it to autostart:

```
sudo virsh -c qemu:///system pool-define remote-iso.xml
sudo virsh -c qemu:///system pool-autostart iso
```

When creating new virtual machines, the contents of the remote location are now easily selectable from within *virt-manager*'s creation wizard.

Revision #14
Created 2020-05-06 20:38:15 UTC by Sebin
Updated 2026-03-14 19:02:48 UTC by Sebin