QEMU

According to the QEMU about page (http://wiki.qemu.org/Main_Page), "QEMU is a generic and open source machine emulator and virtualizer."

When used as a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your x86 PC). By using dynamic translation, it achieves very good performance.

Related articles

- Category:Hypervisors
- Libvirt
- PCI passthrough via OVMF

QEMU can use other hypervisors like Xen or KVM to use CPU extensions (HVM) for virtualization. When used as a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU.

Contents

Installation

Install the `qemu (https://www.archlinux.org/packages/?name=qemu)` package (or `qemu-headless (https://www.archlinux.org/packages/?name=qemu-headless)` for the version without GUI) and below optional packages for your needs:

- `qemu-arch-extra (https://www.archlinux.org/packages/?name=qemu-arch-extra)` - extra architectures support
- `qemu-block-gluster (https://www.archlinux.org/packages/?name=qemu-block-gluster)` - **Glusterfs** block support
- `qemu-block-iscsi (https://www.archlinux.org/packages/?name=qemu-block-iscsi)` - **iSCSI** block support
- `qemu-block-rbd (https://www.archlinux.org/packages/?name=qemu-block-rbd)` - RBD block support
- `samba (https://www.archlinux.org/packages/?name=samba)` - **SMB/CIFS** server support

Graphical front-ends for QEMU

Unlike other virtualization programs such as **VirtualBox** and **VMware**, QEMU does not provide a GUI to manage virtual machines (other than the window that appears when running a virtual machine), nor does it provide a way to create persistent virtual machines with saved

settings. All parameters to run a virtual machine must be specified on the command line at every launch, unless you have created a custom script to start your virtual machine(s). However, there are several GUI front-ends for QEMU:

- **virt-manager (https://www.archlinux.org/packages/?name=virt-manager)**
- **gnome-boxes (https://www.archlinux.org/packages/?name=gnome-boxes)**
- **qemu-launcher (https://aur.archlinux.org/packages/qemu-launcher/)** AUR
- **qtemu (https://aur.archlinux.org/packages/qtemu/)** AUR
- **aqemu (https://aur.archlinux.org/packages/aqemu/)** AUR

Additional front-ends with QEMU support are available for **libvirt**.

Creating new virtualized system

Creating a hard disk image

Tip: See the **QEMU Wikibook (https://en.wikibooks.org/wiki/QEMU/Images)** for more information on QEMU images.

To run QEMU you will need a hard disk image, unless you are booting a live system from CD-ROM or the network (and not doing so to install an operating system to a hard disk image). A hard disk image is a file which stores the contents of the emulated hard disk.

A hard disk image can be *raw*, so that it is literally byte-by-byte the same as what the guest sees, and will always use the full capacity of the guest hard drive on the host. This method provides the least I/O overhead, but can waste a lot of space, as not-used space on the guest cannot be used on the host.

Alternatively, the hard disk image can be in a format such as *qcow2* which only allocates space to the image file when the guest operating system actually writes to those sectors on its virtual hard disk. The image appears as the full size to the guest operating system, even though it may take up only a very small amount of space on the host system. This image format also supports QEMU snapshotting functionality (see **#Creating and managing snapshots via the monitor console** for details). However, using this format instead of *raw* will likely affect performance.

QEMU provides the `qemu-img` command to create hard disk images. For example to create a 4 GB image in the *raw* format:

```
$ qemu-img create -f raw image_file 4G
```

You may use `-f qcow2` to create a *qcow2* disk instead.

Note: You can also simply create a *raw* image by creating a file of the needed size using `dd` or `fallocate`.

Warning: If you store the hard disk images on a **Btrfs** file system, you should consider disabling **Copy-on-Write** for the directory before creating any images.

Overlay storage images

You can create a storage image once (the 'backing' image) and have QEMU keep mutations to this image in an overlay image. This allows you to revert to a previous state of this storage image. You could revert by creating a new overlay image at the time you wish to revert, based on the original backing image.

To create an overlay image, issue a command like:

```
$ qemu-img create -o backing_file=img1.raw,backing_fmt=raw -f qcow2 img1.cow
```

After that you can run your QEMU VM as usual (see **#Running virtualized system**):

```
$ qemu-system-x86_64 img1.cow
```

The backing image will then be left intact and mutations to this storage will be recorded in the overlay image file.

When the path to the backing image changes, repair is required.

Warning: The backing image's absolute filesystem path is stored in the (binary) overlay image file. Changing the backing image's path requires some effort.

Make sure that the original backing image's path still leads to this image. If necessary, make a symbolic link at the original path to the new path. Then issue a command like:

```
$ qemu-img rebase -b /new/img1.raw /new/img1.cow
```

At your discretion, you may alternatively perform an 'unsafe' rebase where the old path to the backing image is not checked:

```
$ qemu-img rebase -u -b /new/img1.raw /new/img1.cow
```

Resizing an image

Warning: Resizing an image containing an NTFS boot file system could make the operating system installed on it unbootable. For full explanation and workaround see **[1] (http://tjworld.net/wiki/Howto/ResizeQemuDiskImages)**.

The `qemu-img` executable has the `resize` option, which enables easy resizing of a hard drive image. It works for both *raw* and *qcow2*. For example, to increase image space by 10 GB, run:

```
$ qemu-img resize disk_image +10G
```

After enlarging the disk image, you must use file system and partitioning tools inside the virtual machine to actually begin using the new space. When shrinking a disk image, you must **first reduce the allocated file systems and partition sizes** using the file system and partitioning tools inside the virtual machine and then shrink the disk image accordingly, otherwise shrinking the disk image will result in data loss!

Preparing the installation media

To install an operating system into your disk image, you need the installation medium (e.g. optical disk, USB-drive, or ISO image) for the operating system. The installation medium should not be mounted because QEMU accesses the media directly.

Tip: If using an optical disk, it is a good idea to first dump the media to a file because this both improves performance and does not require you to have direct access to the devices (that is, you can run QEMU as a regular user without having to change access permissions on the media's device file). For example, if the CD-ROM device node is named `/dev/cdrom`, you can dump it to a file with the command:

```
$ dd if=/dev/cdrom of=cd_image.iso
```

Installing the operating system

This is the first time you will need to start the emulator. To install the operating system on the disk image, you must attach both the disk image and the installation media to the virtual machine, and have it boot from the installation media.

For example on i386 guests, to install from a bootable ISO file as CD-ROM and a raw disk image:

```
$ qemu-system-x86_64 -cdrom iso_image -boot order=d -drive file=disk_image,format=raw
```

See **qemu(1) (https://jlk.fjfi.cvut.cz/arch/manpages/man/qemu.1)** for more information about loading other media types (such as floppy, disk images or physical drives) and **#Running virtualized system** for other useful options.

After the operating system has finished installing, the QEMU image can be booted directly (see **#Running virtualized system**).

Warning: By default only 128 MB of memory is assigned to the machine. The amount of memory can be adjusted with the `-m` switch, for example `-m 512M` or `-m 2G`.

Tip:

- Instead of specifying `-boot order=x`, some users may feel more comfortable using a boot menu: `-boot menu=on`, at least during configuration and experimentation.
- If you need to replace floppies or CDs as part of the installation process, you can use the QEMU machine monitor (press `Ctrl+Alt+2` in the virtual machine's window) to remove

and attach storage devices to a virtual machine. Type `info block` to see the block devices, and use the `change` command to swap out a device. Press `Ctrl+Alt+1` to go back to the virtual machine.

Running virtualized system

`qemu-system-*` binaries (for example `qemu-system-i386` or `qemu-system-x86_64`, depending on guest's architecture) are used to run the virtualized guest. The usage is:

```
$ qemu-system-x86_64 options disk_image
```

Options are the same for all `qemu-system-*` binaries, see **qemu(1) (https://jlk.fjfi.cvut.cz/arch/manpages/man/qemu.1)** for documentation of all options.

By default, QEMU will show the virtual machine's video output in a window. One thing to keep in mind: when you click inside the QEMU window, the mouse pointer is grabbed. To release it, press `Ctrl+Alt+g`.

Warning: QEMU should never be run as root. If you must launch it in a script as root, you should use the `-runas` option to make QEMU drop root privileges.

Enabling KVM

KVM must be supported by your processor and kernel, and necessary **kernel modules** must be loaded. See **KVM** for more information.

To start QEMU in KVM mode, append `-enable-kvm` to the additional start options. To check if KVM is enabled for a running VM, enter the QEMU **Monitor (https://en.wikibooks.org/wiki/QEMU/Monitor)** using `Ctrl+Alt+Shift+2`, and type `info kvm`.

Note:

- If you start your VM with a GUI tool and experience very bad performance, you should check for proper KVM support, as QEMU may be falling back to software emulation.
- KVM needs to be enabled in order to start Windows 7 and Windows 8 properly without a *blue screen*.

Enabling IOMMU (Intel VT-d/AMD-Vi) support

Using IOMMU opens to features like PCI passthrough and memory protection from faulty or malicious devices, see **wikipedia:Input-output memory management unit#Advantages** and **Memory Management (computer programming): Could you explain IOMMU in plain English? (https://www.quora.com/Memory-Management-computer-programming/Could-you-explain-IOMMU-in-plain-English)**.

To enable IOMMU:

1. Ensure that AMD-Vi/Intel VT-d is supported by the CPU and is enabled in the BIOS settings.
2. Set the correct **kernel parameter** based on the CPU-vendor:
   - Intel - `intel_iommu=on`
   - AMD - `amd_iommu=on`
3. If you want IOMMU *only* for PCI passthrough, also add the `iommu=pt` parameter. This will make Linux only enable IOMMU for devices that can be passed through, and will avoid affecting the performance of other devices on the system. The CPU vendor-specific parameter (see above) is still required to be `on`.
4. Reboot and ensure IOMMU is enabled by checking `dmesg` for `DMAR`: `[0.000000] DMAR: IOMMU enabled`
5. Add `-device intel-iommu` to create the IOMMU device:

```
$ qemu-system-x86_64 -enable-kvm -machine q35,accel=kvm -device intel-iommu -cpu host,hv_relaxed,hv_spinlocks=0x1fff,hv_vapic,hv_time ..
```

Note: On Intel CPU based systems creating an IOMMU device in a QEMU guest with `-device intel-iommu` will disable PCI passthrough with an error like:

```
Device at bus pcie.0 addr 09.0 requires iommu notifier which is currently not supported by intel-iommu emulation
```

While adding the kernel parameter `intel_iommu=on` is still needed for remapping IO (e.g. **PCI passthrough with vfio-pci**), `-device intel-iommu` should not be set if PCI PCI passthrough is required.

Moving data between host and guest OS

Network

Data can be shared between the host and guest OS using any network protocol that can transfer files, such as **NFS**, **SMB**, **NBD**, HTTP, **FTP**, or **SSH**, provided that you have set up the network appropriately and enabled the appropriate services.

The default user-mode networking allows the guest to access the host OS at the IP address 10.0.2.2. Any servers that you are running on your host OS, such as a SSH server or SMB server, will be accessible at this IP address. So on the guests, you can mount directories exported on the host via **SMB** or **NFS**, or you can access the host's HTTP server, etc. It will not be possible for the host OS to access servers running on the guest OS, but this can be done with other network configurations (see **#Tap networking with QEMU**).

QEMU's built-in SMB server

QEMU's documentation says it has a "built-in" SMB server, but actually it just starts up **Samba** with an automatically generated `smb.conf` file located at `/tmp/qemu-smb.pid-0/smb.conf` and makes it accessible to the guest at a different IP address (10.0.2.4 by default). This only works for user networking, and this is not necessarily very useful since the guest can also access the normal **Samba** service on the host if you have set up shares on it.

To enable this feature, start QEMU with a command like:

```
$ qemu-system-x86_64 disk_image -net nic -net user,smb=shared_dir_path
```

where `shared_dir_path` is a directory that you want to share between the guest and host.

Then, in the guest, you will be able to access the shared directory on the host 10.0.2.4 with the share name "qemu". For example, in Windows Explorer you would go to `\\10.0.2.4\qemu`.

Note:

- If you are using sharing options multiple times like `-net user,smb=shared_dir_path1 -net user,smb=shared_dir_path2` or `-net user,smb=shared_dir_path1,smb=shared_dir_path2` then it will share only the last defined one.
- If you cannot access the shared folder and the guest system is Windows, check that the **NetBIOS protocol is enabled (http://ecross.mvps.org/howto/enable-netbios-over-tcp-ip-with-windows.htm)** and that a firewall does not block **ports (http://technet.microsoft.com/en-us/library/cc940063.aspx)** used by the NetBIOS protocol.

Mounting a partition inside a raw disk image

When the virtual machine is not running, it is possible to mount partitions that are inside a raw disk image file by setting them up as loopback devices. This does not work with disk images in special formats, such as qcow2, although those can be mounted using `qemu-nbd`.

Warning: You must make sure to unmount the partitions before running the virtual machine again. Otherwise, data corruption is very likely to occur.

With manually specifying byte offset

One way to mount a disk image partition is to mount the disk image at a certain offset using a command like the following:

```
# mount -o loop,offset=32256 disk_image mountpoint
```

The `offset=32256` option is actually passed to the `losetup` program to set up a loopback device that starts at byte offset 32256 of the file and continues to the end. This loopback device is then mounted. You may also use the `sizelimit` option to specify the exact size of the partition, but this is usually unnecessary.

Depending on your disk image, the needed partition may not start at offset 32256. Run `fdisk -l disk_image` to see the partitions in the image. fdisk gives the start and end offsets in 512-byte sectors, so multiply by 512 to get the correct offset to pass to `mount`.

With loop module autodetecting partitions

The Linux loop driver actually supports partitions in loopback devices, but it is disabled by default. To enable it, do the following:

- Get rid of all your loopback devices (unmount all mounted images, etc.).
- **Unload** the `loop` kernel module, and load it with the `max_part=15` parameter set. Additionally, the maximum number of loop devices can be controlled with the `max_loop` parameter.

Tip: You can put an entry in `/etc/modprobe.d` to load the loop module with `max_part=15` every time, or you can put `loop.max_part=15` on the kernel command-line, depending on whether you have the `loop.ko` module built into your kernel or not.

Set up your image as a loopback device:

```
# losetup -f -P disk_image
```

Then, if the device created was `/dev/loop0`, additional devices `/dev/loop0pX` will have been automatically created, where X is the number of the partition. These partition loopback devices can be mounted directly. For example:

```
# mount /dev/loop0p1 mountpoint
```

To mount the disk image with *udisksctl*, see **Udisks#Mount loop devices**.

With kpartx

kpartx from the multipath-tools (https://aur.archlinux.org/packages/multipath-tools/)[AUR] package can read a partition table on a device and create a new device for each partition. For example:

```
# kpartx -a disk_image
```

This will setup the loopback device and create the necessary partition(s) device(s) in `/dev/mapper/`.

Mounting a partition inside a qcow2 image

You may mount a partition inside a qcow2 image using `qemu-nbd`. See **Wikibooks (http://en.wikibooks.org/wiki/QEMU/Images#Mounting_an_image_on_the_host)**.

Using any real partition as the single primary partition of a hard disk image

Sometimes, you may wish to use one of your system partitions from within QEMU. Using a raw partition for a virtual machine will improve performance, as the read and write operations do not go through the file system layer on the physical host. Such a partition also provides a way to share data between the host and guest.

In Arch Linux, device files for raw partitions are, by default, owned by *root* and the *disk* group. If you would like to have a non-root user be able to read and write to a raw partition, you need to change the owner of the partition's device file to that user.

Warning:

- Although it is possible, it is not recommended to allow virtual machines to alter critical data on the host system, such as the root partition.
- You must not mount a file system on a partition read-write on both the host and the guest at the same time. Otherwise, data corruption will result.

After doing so, you can attach the partition to a QEMU virtual machine as a virtual disk.

However, things are a little more complicated if you want to have the *entire* virtual machine contained in a partition. In that case, there would be no disk image file to actually boot the virtual machine since you cannot install a bootloader to a partition that is itself formatted as a file system and not as a partitioned device with a MBR. Such a virtual machine can be booted either by specifying the **kernel** and **initrd** manually, or by simulating a disk with a MBR by using linear **RAID**.

By specifying kernel and initrd manually

QEMU supports loading **Linux kernels** and **init ramdisks** directly, thereby circumventing bootloaders such as **GRUB**. It then can be launched with the physical partition containing the root file system as the virtual disk, which will not appear to be partitioned. This is done by issuing a command similar to the following:

Note: In this example, it is the **host's** images that are being used, not the guest's. If you wish to use the guest's images, either mount `/dev/sda3` read-only (to protect the file system from the host) and specify the `/full/path/to/images` or use some kexec hackery in the guest to reload the guest's kernel (extends boot time).

```
$ qemu-system-x86_64 -kernel /boot/vmlinuz-linux -initrd /boot/initramfs-linux.img -append root=/dev/sda /dev/sda3
```

In the above example, the physical partition being used for the guest's root file system is `/dev/sda3` on the host, but it shows up as `/dev/sda` on the guest.

You may, of course, specify any kernel and initrd that you want, and not just the ones that come with Arch Linux.

When there are multiple **kernel parameters** to be passed to the `-append` option, they need to be quoted using single or double quotes. For example:

```
... -append 'root=/dev/sda1 console=ttyS0'
```

Simulate virtual disk with MBR using linear RAID

A more complicated way to have a virtual machine use a physical partition, while keeping that partition formatted as a file system and not just having the guest partition the partition as if it were a disk, is to simulate a MBR for it so that it can boot using a bootloader such as GRUB.

You can do this using software **RAID** in linear mode (you need the `linear.ko` kernel driver) and a loopback device: the trick is to dynamically prepend a master boot record (MBR) to the real partition you wish to embed in a QEMU raw disk image.

Suppose you have a plain, unmounted `/dev/hdaN` partition with some file system on it you wish to make part of a QEMU disk image. First, you create some small file to hold the MBR:

```
$ dd if=/dev/zero of=/path/to/mbr count=32
```

Here, a 16 KB (32 * 512 bytes) file is created. It is important not to make it too small (even if the MBR only needs a single 512 bytes block), since the smaller it will be, the smaller the chunk size of the software RAID device will have to be, which could have an impact on performance. Then, you setup a loopback device to the MBR file:

```
# losetup -f /path/to/mbr
```

Let us assume the resulting device is `/dev/loop0`, because we would not already have been using other loopbacks. Next step is to create the "merged" MBR + `/dev/hdaN` disk image using software RAID:

```
# modprobe linear
# mdadm --build --verbose /dev/md0 --chunk=16 --level=linear --raid-devices=2 /dev/loop0 /dev/hdaN
```

The resulting `/dev/md0` is what you will use as a QEMU raw disk image (do not forget to set the permissions so that the emulator can access it). The last (and somewhat tricky) step is to set the disk configuration (disk geometry and partitions table) so that the primary partition start point in the MBR matches the one of `/dev/hdaN` inside `/dev/md0` (an offset of exactly 16 * 512 = 16384 bytes in this example). Do this using `fdisk` on the host machine, not in the emulator: the default raw disc detection routine from QEMU often results in non-kilobyte-roundable offsets (such as 31.5 KB, as in the previous section) that cannot be managed by the software RAID code. Hence, from the the host:

```
# fdisk /dev/md0
```

Press `X` to enter the expert menu. Set number of 's'ectors per track so that the size of one cylinder matches the size of your MBR file. For two heads and a sector size of 512, the number of sectors per track should be 16, so we get cylinders of size 2x16x512=16k.

Now, press `R` to return to the main menu.

Press `P` and check that the cylinder size is now 16k.

Now, create a single primary partition corresponding to `/dev/hdaN`. It should start at cylinder 2 and end at the end of the disk (note that the number of cylinders now differs from what it was when you entered fdisk.

Finally, 'w'rite the result to the file: you are done. You now have a partition you can mount directly from your host, as well as part of a QEMU disk image:

```
$ qemu-system-x86_64 -hdc /dev/md0 [...]
```

You can, of course, safely set any bootloader on this disk image using QEMU, provided the original `/dev/hdaN` partition contains the necessary tools.

Alternative: use nbd-server

Instead of linear RAID, you may use `nbd-server` (from the **nbd (https://www.archlinux.org/packages/?name=nbd)** package) to create an MBR wrapper for QEMU.

Assuming you have already set up your MBR wrapper file like above, rename it to `wrapper.img.0`. Then create a symbolic link named `wrapper.img.1` in the same directory, pointing to your partition. Then put the following script in the same directory:

```
#!/bin/sh
dir="$(realpath "$(dirname "$0")")"
cat >wrapper.conf <<EOF
[generic]
allowlist = true
listenaddr = 127.713705
port = 10809

[wrap]
exportname = $dir/wrapper.img
multifile = true
EOF

nbd-server \
    -C wrapper.conf \
```

```
    -p wrapper.pid \
    "$@"
```

The `.0` and `.1` suffixes are essential; the rest can be changed. After running the above script (which you may need to do as root to make sure nbd-server is able to access the partition), you can launch QEMU with:

```
qemu-system-x86_64 -drive file=nbd:127.713705:10809:exportname=wrap [...]
```

Networking

The performance of virtual networking should be better with tap devices and bridges than with user-mode networking or vde because tap devices and bridges are implemented in-kernel.

In addition, networking performance can be improved by assigning virtual machines a **virtio (http://wiki.libvirt.org/page/Virtio)** network device rather than the default emulation of an e1000 NIC. See **#Installing virtio drivers** for more information.

Link-level address caveat

By giving the `-net nic` argument to QEMU, it will, by default, assign a virtual machine a network interface with the link-level address `52:54:00:12:34:56`. However, when using bridged networking with multiple virtual machines, it is essential that each virtual machine has a unique link-level (MAC) address on the virtual machine side of the tap device. Otherwise,

the bridge will not work correctly, because it will receive packets from multiple sources that have the same link-level address. This problem occurs even if the tap devices themselves have unique link-level addresses because the source link-level address is not rewritten as packets pass through the tap device.

Make sure that each virtual machine has a unique link-level address, but it should always start with `52:54:` . Use the following option, replace *X* with arbitrary hexadecimal digit:

```
$ qemu-system-x86_64 -net nic,macaddr=52:54:XX:XX:XX:XX -net vde disk_image
```

Generating unique link-level addresses can be done in several ways:

1. Manually specify unique link-level address for each NIC. The benefit is that the DHCP server will assign the same IP address each time the virtual machine is run, but it is unusable for large number of virtual machines.
2. Generate random link-level address each time the virtual machine is run. Practically zero probability of collisions, but the downside is that the DHCP server will assign a different IP address each time. You can use the following command in a script to generate random link-level address in a `macaddr` variable:

   ```
   printf -v macaddr "52:54:%02x:%02x:%02x:%02x" $(( $RANDOM & 0xff)) $(( $RANDOM & 0xff )) $(( $RANDOM & 0xff)) $(( $RANDOM & 0xff ))
   qemu-system-x86_64 -net nic,macaddr="$macaddr" -net vde disk_image
   ```

3. Use the following script `qemu-mac-hasher.py` to generate the link-level address from the virtual machine name using a hashing function. Given that the names of virtual machines are unique, this method combines the benefits of the aforementioned methods: it generates

the same link-level address each time the script is run, yet it preserves the practically zero probability of collisions.

qemu-mac-hasher.py

```
#!/usr/bin/env python

import sys
import zlib

if len(sys.argv) != 2:
    print("usage: %s <VM Name>" % sys.argv[0])
    sys.exit(1)

crc = zlib.crc32(sys.argv[1].encode("utf-8")) & 0xffffffff
crc = str(hex(crc))[2:]
print("52:54:%s%s:%s%s:%s%s:%s%s" % tuple(crc))
```

In a script, you can use for example:

```
vm_name="VM Name"
qemu-system-x86_64 -name "$vm_name" -net nic,macaddr=$(qemu-mac-hasher.py "$vm_name") -net vde disk_image
```

User-mode networking

By default, without any `-netdev` arguments, QEMU will use user-mode networking with a built-in DHCP server. Your virtual machines will be assigned an IP address when they run their DHCP client, and they will be able to access the physical host's network through IP masquerading done by QEMU.

Warning: This only works with the TCP and UDP protocols, so ICMP, including `ping`, will not work. Do not use `ping` to test network connectivity.

This default configuration allows your virtual machines to easily access the Internet, provided that the host is connected to it, but the virtual machines will not be directly visible on the external network, nor will virtual machines be able to talk to each other if you start up more than one concurrently.

QEMU's user-mode networking can offer more capabilities such as built-in TFTP or SMB servers, redirecting host ports to the guest (for example to allow SSH connections to the guest) or attaching guests to VLANs so that they can talk to each other. See the QEMU documentation on the `-net user` flag for more details.

However, user-mode networking has limitations in both utility and performance. More advanced network configurations require the use of tap devices or other methods.

Note: If the host system uses **systemd-networkd**, make sure to symlink the `/etc/resolv.conf` file as described in **systemd-networkd#Required services and setup**, otherwise the DNS lookup in the guest system will not work.

Tap networking with QEMU

Tap devices are a Linux kernel feature that allows you to create virtual network interfaces that appear as real network interfaces. Packets sent to a tap interface are delivered to a userspace program, such as QEMU, that has bound itself to the interface.

QEMU can use tap networking for a virtual machine so that packets sent to the tap interface will be sent to the virtual machine and appear as coming from a network interface (usually an Ethernet interface) in the virtual machine. Conversely, everything that the virtual machine sends through its network interface will appear on the tap interface.

Tap devices are supported by the Linux bridge drivers, so it is possible to bridge together tap devices with each other and possibly with other host interfaces such as `eth0`. This is desirable if you want your virtual machines to be able to talk to each other, or if you want other machines on your LAN to be able to talk to the virtual machines.

Warning: If you bridge together tap device and some host interface, such as `eth0`, your virtual machines will appear directly on the external network, which will expose them to possible attack. Depending on what resources your virtual machines have access to, you may need to take all the **precautions** you normally would take in securing a computer to secure your virtual machines. If the risk is too great, virtual machines have little resources or you set up multiple virtual machines, a better solution might be to use **host-only networking** and set up NAT. In this case you only need one firewall on the host instead of multiple firewalls for each guest.

As indicated in the user-mode networking section, tap devices offer higher networking performance than user-mode. If the guest OS supports virtio network driver, then the networking performance will be increased considerably as well. Supposing the use of the tap0 device, that the virtio driver is used on the guest, and that no scripts are used to help start/stop networking, next is part of the qemu command one should see:

```
-net nic,model=virtio -net tap,ifname=tap0,script=no,downscript=no
```

But if already using a tap device with virtio networking driver, one can even boost the networking performance by enabling vhost, like:

```
-net nic,model=virtio -net tap,ifname=tap0,script=no,downscript=no,vhost=on
```

See **http://www.linux-kvm.com/content/how-maximize-virtio-net-performance-vhost-net** for more information.

Host-only networking

If the bridge is given an IP address and traffic destined for it is allowed, but no real interface (e.g. `eth0`) is connected to the bridge, then the virtual machines will be able to talk to each other and the host system. However, they will not be able to talk to anything on the external

network, provided that you do not set up IP masquerading on the physical host. This configuration is called *host-only networking* by other virtualization software such as **VirtualBox**.

Tip:

- If you want to set up IP masquerading, e.g. NAT for virtual machines, see the **Internet sharing#Enable NAT** page.
- You may want to have a DHCP server running on the bridge interface to service the virtual network. For example, to use the `172.20.0.1/16` subnet with **dnsmasq** as the DHCP server:

```
# ip addr add 172.20.0.1/16 dev br0
# ip link set br0 up
# dnsmasq --interface=br0 --bind-interfaces --dhcp-range=172.20.0.2,172.20.255.254
```

Internal networking

If you do not give the bridge an IP address and add an **iptables** rule to drop all traffic to the bridge in the INPUT chain, then the virtual machines will be able to talk to each other, but not to the physical host or to the outside network. This configuration is called *internal networking* by other virtualization software such as **VirtualBox**. You will need to either assign static IP addresses to the virtual machines or run a DHCP server on one of them.

By default iptables would drop packets in the bridge network. You may need to use such iptables rule to allow packets in a bridged network:

```
# iptables -I FORWARD -m physdev --physdev-is-bridged -j ACCEPT
```

Bridged networking using qemu-bridge-helper

Note: This method is available since QEMU 1.1, see **http://wiki.qemu.org/Features/HelperNetworking**.

This method does not require a start-up script and readily accommodates multiple taps and multiple bridges. It uses `/usr/lib/qemu/qemu-bridge-helper` binary, which allows creating tap devices on an existing bridge.

Tip: See **Network bridge** for information on creating bridge.

First, create a configuration file containing the names of all bridges to be used by QEMU:

/etc/qemu/bridge.conf

```
allow bridge0
allow bridge1
...
```

Now start the VM. The most basic usage would be:

```
$ qemu-system-x86_64 -net nic -net bridge,br=bridge0 [...]
```

With multiple taps, the most basic usage requires specifying the VLAN for all additional NICs:

```
$ qemu-system-x86_64 -net nic -net bridge,br=bridge0 -net nic,vlan=1 -net bridge,vlan=1,br=bridge1 [...]
```

Creating bridge manually

Tip: Since QEMU 1.1, the **network bridge helper (http://wiki.qemu.org/Features/HelperNetworking)** can set tun/tap up for you without the need for additional scripting. See **#Bridged networking using qemu-bridge-helper**.

The following describes how to bridge a virtual machine to a host interface such as `eth0`, which is probably the most common configuration. This configuration makes it appear that the virtual machine is located directly on the external network, on the same Ethernet segment as the physical host machine.

We will replace the normal Ethernet adapter with a bridge adapter and bind the normal Ethernet adapter to it.

- Install **`bridge-utils (https://www.archlinux.org/packages/?name=bridge-utils)`**, which provides `brctl` to manipulate bridges.

- Enable IPv4 forwarding:

```
# sysctl net.ipv4.ip_forward=1
```

To make the change permanent, change `net.ipv4.ip_forward = 0` to `net.ipv4.ip_forward = 1` in `/etc/sysctl.d/99-sysctl.conf`.

- Load the `tun` module and configure it to be loaded on boot. See **Kernel modules** for details.
- Now create the bridge. See **Bridge with netctl** for details. Remember to name your bridge as `br0`, or change the scripts below to your bridge's name.
- Create the script that QEMU uses to bring up the tap adapter with `root:kvm` 750 permissions:

/etc/qemu-ifup

```
#!/bin/sh

echo "Executing /etc/qemu-ifup"
echo "Bringing up $1 for bridged mode..."
sudo /usr/bin/ip link set $1 up promisc on
echo "Adding $1 to br0..."
sudo /usr/bin/brctl addif br0 $1
sleep 2
```

- Create the script that QEMU uses to bring down the tap adapter in `/etc/qemu-ifdown` with `root:kvm` 750 permissions:

/etc/qemu-ifdown

```
#!/bin/sh

echo "Executing /etc/qemu-ifdown"
sudo /usr/bin/ip link set $1 down
sudo /usr/bin/brctl delif br0 $1
sudo /usr/bin/ip link delete dev $1
```

- Use `visudo` to add the following to your `sudoers` file:

```
Cmnd_Alias      QEMU=/usr/bin/ip,/usr/bin/modprobe,/usr/bin/brctl
%kvm     ALL=NOPASSWD: QEMU
```

- You launch QEMU using the following `run-qemu` script:

run-qemu

```
#!/bin/bash
USERID=$(whoami)

# Get name of newly created TAP device; see https://bbs.archlinux.org/viewtopic.php?pid=1285079#p1285079
precreationg=$(/usr/bin/ip tuntap list | /usr/bin/cut -d: -f1 | /usr/bin/sort)
sudo /usr/bin/ip tuntap add user $USERID mode tap
postcreation=$(/usr/bin/ip tuntap list | /usr/bin/cut -d: -f1 | /usr/bin/sort)
IFACE=$(comm -13 <(echo "$precreationg") <(echo "$postcreation"))

# This line creates a random MAC address. The downside is the DHCP server will assign a different IP address each time
printf -v macaddr "52:54:%02x:%02x:%02x:%02x" $(( $RANDOM & 0xff)) $(( $RANDOM & 0xff )) $(( $RANDOM & 0xff)) $(( $RANDOM & 0xff ))
# Instead, uncomment and edit this line to set a static MAC address. The benefit is that the DHCP server will assign the same IP address.
# macaddr='52:54:be:36:42:a9'

qemu-system-x86_64 -net nic,macaddr=$macaddr -net tap,ifname="$IFACE" $*

sudo ip link set dev $IFACE down &> /dev/null
sudo ip tuntap del $IFACE mode tap &> /dev/null
```

Then to launch a VM, do something like this

```
$ run-qemu -hda myvm.img -m 512 -vga std
```

- It is recommended for performance and security reasons to disable the **firewall on the bridge (http://ebtables.netfilter.org/documentation/bridge-nf.html)**:

```
/etc/sysctl.d/10-disable-firewall-on-bridge.conf
```

```
net.bridge.bridge-nf-call-ip6tables = 0
net.bridge.bridge-nf-call-iptables = 0
net.bridge.bridge-nf-call-arptables = 0
```

Run `sysctl -p /etc/sysctl.d/10-disable-firewall-on-bridge.conf` to apply the changes immediately.

See the **libvirt wiki (http://wiki.libvirt.org/page/Networking#Creating_network_initscripts)** and **Fedora bug 512206 (https://bugzilla.redhat.com/show_bug.cgi?id=512206)**. If you get errors by sysctl during boot about non-existing files, make the `bridge` module load at boot. See **Kernel modules#Automatic module handling**.

Alternatively, you can configure **iptables** to allow all traffic to be forwarded across the bridge by adding a rule like this:

```
-I FORWARD -m physdev --physdev-is-bridged -j ACCEPT
```

Network sharing between physical device and a Tap device through iptables

Bridged networking works fine between a wired interface (Eg. eth0), and it is easy to setup. However if the host gets connected to the network through a wireless device, then bridging is not possible.

See **Network bridge#Wireless interface on a bridge** as a reference.

One way to overcome that is to setup a tap device with a static IP, making linux automatically handle the routing for it, and then forward traffic between the tap interface and the device connected to the network through iptables rules.

See **Internet sharing** as a reference.

There you can find what is needed to share the network between devices, included tap and tun ones. The following just hints further on some of the host configurations required. As indicated in the reference above, the client needs to be configured for a static IP, using the IP assigned to the tap interface as the gateway. The caveat is that the DNS servers on the client might need to be manually edited if they change when changing from one host device connected to the network to another.

To allow IP forwarding on every boot, one need to add the following lines to sysctl configuration file inside /etc/sysctl.d:

```
net.ipv4.ip_forward = 1
net.ipv6.conf.default.forwarding = 1
net.ipv6.conf.all.forwarding = 1
```

The iptables rules can look like:

```
# Forwarding from/to outside
iptables -A FORWARD -i ${INT} -o ${EXT_0} -j ACCEPT
iptables -A FORWARD -i ${INT} -o ${EXT_1} -j ACCEPT
iptables -A FORWARD -i ${INT} -o ${EXT_2} -j ACCEPT
iptables -A FORWARD -i ${EXT_0} -o ${INT} -j ACCEPT
iptables -A FORWARD -i ${EXT_1} -o ${INT} -j ACCEPT
iptables -A FORWARD -i ${EXT_2} -o ${INT} -j ACCEPT
# NAT/Masquerade (network address translation)
iptables -t nat -A POSTROUTING -o ${EXT_0} -j MASQUERADE
iptables -t nat -A POSTROUTING -o ${EXT_1} -j MASQUERADE
iptables -t nat -A POSTROUTING -o ${EXT_2} -j MASQUERADE
```

The prior supposes there are 3 devices connected to the network sharing traffic with one internal device, where for example:

```
INT=tap0
EXT_0=eth0
EXT_1=wlan0
EXT_2=tun0
```

The prior shows a forwarding that would allow sharing wired and wireless connections with the tap device.

The forwarding rules shown are stateless, and for pure forwarding. One could think of restricting specific traffic, putting a firewall in place to protect the guest and others. However those would decrease the networking performance, while a simple bridge does not include any of that.

Bonus: Whether the connection is wired or wireless, if one gets connected through VPN to a remote site with a tun device, supposing the tun device opened for that connection is tun0, and the prior iptables rules are applied, then the remote connection gets also shared with the guest. This avoids the need for the guest to also open a VPN connection. Again, as the guest networking needs to be static, then if connecting the host remotely this way, one most probably will need to edit the DNS servers on the guest.

Networking with VDE2

What is VDE?

VDE stands for Virtual Distributed Ethernet. It started as an enhancement of **uml**_switch. It is a toolbox to manage virtual networks.

The idea is to create virtual switches, which are basically sockets, and to "plug" both physical and virtual machines in them. The configuration we show here is quite simple; However, VDE is much more powerful than this, it can plug virtual switches together, run them on different hosts and monitor the traffic in the switches. You are invited to read **the documentation of the project (http://wiki.virtualsquare.org/wiki/index.php/Main_Page)**.

The advantage of this method is you do not have to add sudo privileges to your users. Regular users should not be allowed to run modprobe.

Basics

VDE support can be **installed** via the **vde2 (https://www.archlinux.org/packages/?name=vde2)** package.

In our config, we use tun/tap to create a virtual interface on my host. Load the `tun` module (see **Kernel modules** for details):

```
# modprobe tun
```

Now create the virtual switch:

```
# vde_switch -tap tap0 -daemon -mod 660 -group users
```

This line creates the switch, creates `tap0`, "plugs" it, and allows the users of the group `users` to use it.

The interface is plugged in but not configured yet. To configure it, run this command:

```
# ip addr add 192.168.100.254/24 dev tap0
```

Now, you just have to run KVM with these `-net` options as a normal user:

```
$ qemu-system-x86_64 -net nic -net vde -hda [...]
```

Configure networking for your guest as you would do in a physical network.

Tip: You might want to set up NAT on tap device to access the internet from the virtual machine. See **Internet sharing#Enable NAT** for more information.

Startup scripts

Example of main script starting VDE:

/etc/systemd/scripts/qemu-network-env

```
#!/bin/sh
# QEMU/VDE network environment preparation script

# The IP configuration for the tap device that will be used for
# the virtual machine network:

TAP_DEV=tap0
TAP_IP=192.168.100.254
TAP_MASK=24
TAP_NETWORK=192.168.100.0

# Host interface
NIC=eth0

case "$1" in
  start)
        echo -n "Starting VDE network for QEMU: "

        # If you want tun kernel module to be loaded by script uncomment here
        #modprobe tun 2>/dev/null
        ## Wait for the module to be loaded
        #while ! lsmod | grep -q "^tun"; do echo "Waiting for tun device"; sleep 1; done

        # Start tap switch
        vde_switch -tap "$TAP_DEV" -daemon -mod 660 -group users

        # Bring tap interface up
        ip address add "$TAP_IP"/"$TAP_MASK" dev "$TAP_DEV"
        ip link set "$TAP_DEV" up

        # Start IP Forwarding
        echo "1" > /proc/sys/net/ipv4/ip_forward
        iptables -t nat -A POSTROUTING -s "$TAP_NETWORK"/"$TAP_MASK" -o "$NIC" -j MASQUERADE
```

```
        ;;
  stop)
        echo -n "Stopping VDE network for QEMU: "
        # Delete the NAT rules
        iptables -t nat -D POSTROUTING "$TAP_NETWORK"/"$TAP_MASK" -o "$NIC" -j MASQUERADE

        # Bring tap interface down
        ip link set "$TAP_DEV" down

        # Kill VDE switch
        pgrep -f vde_switch | xargs kill -TERM
        ;;
  restart|reload)
        $0 stop
        sleep 1
        $0 start
        ;;
  *)
        echo "Usage: $0 {start|stop|restart|reload}"
        exit 1
esac
exit 0
```

Example of systemd service using the above script:

/etc/systemd/system/qemu-network-env.service

```
[Unit]
Description=Manage VDE Switch

[Service]
Type=oneshot
ExecStart=/etc/systemd/scripts/qemu-network-env start
ExecStop=/etc/systemd/scripts/qemu-network-env stop
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
```

Change permissions for `qemu-network-env` to be executable

```
# chmod u+x /etc/systemd/scripts/qemu-network-env
```

You can **start** `qemu-network-env.service` as usual.

Alternative method

If the above method does not work or you do not want to mess with kernel configs, TUN, dnsmasq, and iptables you can do the following for the same result.

```
# vde_switch -daemon -mod 660 -group users
# slirpvde --dhcp --daemon
```

Then, to start the VM with a connection to the network of the host:

```
$ qemu-system-x86_64 -net nic,macaddr=52:54:00:00:EE:03 -net vde disk_image
```

VDE2 Bridge

Based on **quickhowto: qemu networking using vde, tun/tap, and bridge (http://selamatpagicikgu.wordpress.com/2011/06/08/quickhowto-qemu-networking-using-vde-tuntap-and-bridge/)** graphic. Any virtual machine connected to vde is externally exposed. For example, each virtual machine can receive DHCP configuration directly from your ADSL router.

Basics

Remember that you need `tun` module and **bridge-utils (https://www.archlinux.org/packages/?name=bridge-utils)** package.

Create the vde2/tap device:

```
# vde_switch -tap tap0 -daemon -mod 660 -group users
# ip link set tap0 up
```

Create bridge:

```
# brctl addbr br0
```

Add devices:

```
# brctl addif br0 eth0
# brctl addif br0 tap0
```

And configure bridge interface:

```
# dhcpcd br0
```

Startup scripts

All devices must be set up. And only the bridge needs an IP address. For physical devices on the bridge (e.g. `eth0`), this can be done with **netctl** using a custom Ethernet profile with:

```
/etc/netctl/ethernet-noip
```

```
Description='A more versatile static Ethernet connection'
Interface=eth0
Connection=ethernet
IP=no
```

The following custom systemd service can be used to create and activate a VDE2 tap interface for use in the `users` user group.

```
/etc/systemd/system/vde2@.service
```

```
[Unit]
Description=Network Connectivity for %i
Wants=network.target
Before=network.target

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/bin/vde_switch -tap %i -daemon -mod 660 -group users
ExecStart=/usr/bin/ip link set dev %i up
ExecStop=/usr/bin/ip addr flush dev %i
ExecStop=/usr/bin/ip link set dev %i down

[Install]
WantedBy=multi-user.target
```

And finally, you can create the **bridge interface with netctl**.

Graphics

QEMU can use the following different graphic outputs: `std`, `qxl`, `vmware`, `virtio`, `cirrus` and `none`.

std

With `-vga std` you can get a resolution of up to 2560 x 1600 pixels without requiring guest drivers. This is the default since QEMU 2.2.

qxl

QXL is a paravirtual graphics driver with 2D support. To use it, pass the `-vga qxl` option and install drivers in the guest. You may want to use SPICE for improved graphical performance when using QXL.

On Linux guests, the `qxl` and `bochs_drm` kernel modules must be loaded in order to gain a decent performance.

SPICE

The **SPICE project (http://spice-space.org/)** aims to provide a complete open source solution for remote access to virtual machines in a seamless way.

SPICE can only be used when using QXL as the graphical output.

The following is example of booting with SPICE as the remote desktop protocol, including the support for copy and paste from host:

```
$ qemu-system-x86_64 -vga qxl -spice port=5930,disable-ticketing -device virtio-serial-pci -device virtserialport,chardev=spicechannel0,name=com.redhat.spice.0 -chardev spicevmc,id=spicechannel0,name=vdagent
```

From the **SPICE page on the KVM wiki (https://www.linux-kvm.org/page/SPICE)**: "*The* `-device virtio-serial-pci` *option adds the virtio-serial device,* `-device virtserialport,chardev=spicechannel0,name=com.redhat.spice.0` *opens a port for spice vdagent in that device and* `-chardev spicevmc,id=spicechannel0,name=vdagent` *adds a spicevmc chardev for that port. It is important that the* `chardev=` *option of the* `virtserialport` *device matches the* `id=` *option given to the* `chardev` *option (* `spicechannel0` *in this example). It is also important that the port name is* `com.redhat.spice.0`*, because that is the namespace where vdagent is looking for in the guest. And finally, specify* `name=vdagent` *so that spice knows what this channel is for.*"

Tip: Since QEMU in SPICE mode acts similarly to a remote desktop server, it may be more convenient to run QEMU in daemon mode with the `-daemonize` parameter.

Connect to the guest by using a SPICE client. **virt-viewer (https://www.archlinux.org/packages/?name=virt-viewer)** is the recommended SPICE client by the protocol developers:

```
$ remote-viewer spice://127.0.0.1:5930
```

The reference and test implementation **spice-gtk (https://www.archlinux.org/packages/?name=spice-gtk)** can also be used:

```
$ spicy -h 127.0.0.1 -p 5930
```

Other **clients (http://www.spice-space.org/download.html)**, including for other platforms, are also available.

Using **Unix sockets** instead of TCP ports does not involve using network stack on the host system, so it is **reportedly (https://unix.stackexchange.com/questions/91774/performance-of-unix-sockets-vs-tcp-ports)** better for performance. Example:

```
$ qemu-system-x86_64 -vga qxl -device virtio-serial-pci -device virtserialport,chardev=spicechannel0,name=com.redhat.spice.0 -chardev spicevmc,id=spicechannel0,name=vdagent -spice unix,addr=/tmp/vm_spice.socket,disable-ticketing
```

Then connect via:

```
$ remote-viewer spice+unix:///tmp/vm_spice.socket
```

or via:

```
$ spicy --uri="spice+unix:///tmp/vm_spice.socket"
```

For improved support for multiple monitors, clipboard sharing, etc. the following packages should be installed on the guest:

- `spice-vdagent (https://www.archlinux.org/packages/?name=spice-vdagent)`: Spice agent xorg client that enables copy and paste between client and X-session and

more. **Enable** `spice-vdagentd.service` after installation.
- **xf86-video-qxl (https://www.archlinux.org/packages/?name=xf86-video-qxl) xf86-video-qxl-git (https://aur.archlinux.org/packages/xf86-video-qxl-git/)**[AUR]: Xorg X11 qxl video driver
- For other operating systems, see the Guest section on **SPICE-Space download (http://www.spice-space.org/download.html)** page.

Password authentication with SPICE

If you want to enable password authentication with SPICE you need to remove `disable-ticketing` from the `-spice` argument and instead add `password=yourpassword`. For example:

```
$ qemu-system-x86_64 -vga qxl -spice port=5900,password=yourpassword -device virtio-serial-pci -device virtserialport,chardev=spicechannel0,name=com.
redhat.spice.0 -chardev spicevmc,id=spicechannel0,name=vdagent
```

Your SPICE client should now ask for the password to be able to connect to the SPICE server.

TLS encryption

You can also configure TLS encryption for communicating with the SPICE server. First, you need to have a directory which contains the following files (the names must be exactly as indicated):

- `ca-cert.pem` : the CA master certificate.
- `server-cert.pem` : the server certificate signed with `ca-cert.pem` .
- `server-key.pem` : the server private key.

An example of generation of self-signed certificates with your own generated CA for your server is shown in the **Spice User Manual (https://www.spice-space.org/spice-user-manual.html#_generating_self_signed_certificates)**.

Afterwards, you can run QEMU with SPICE as explained above but using the following `-spice` argument:
`-spice tls-port=5901,password=yourpassword,x509-dir=/path/to/pki_certs` , where `/path/to/pki_certs` is the directory path that contains the three needed files shown earlier.

It is now possible to connect to the server using **`virt-viewer` (https://www.archlinux.org/packages/?name=virt-viewer)**:

```
$ remote-viewer spice://hostname?tls-port=5901 --spice-ca-file=/path/to/ca-cert.pem --spice-host-subject="C=XX,L=city,O=organization,CN=hostname" --spice-secure-channels=all
```

Keep in mind that the `--spice-host-subject` parameter needs to be set according to your `server-cert.pem` subject. You also need to copy `ca-cert.pem` to every client to verify the server certificate.

Tip: You can get the subject line of the server certificate in the correct format for `--spice-host-subject` (with entries separated by commas) using the following command:

```
$ openssl x509 -noout -subject -in server-cert.pem | cut -d' ' -f2- | sed 's/\///' | sed 's/\//,/g'
```

The equivalent **spice-gtk (https://www.archlinux.org/packages/?name=spice-gtk)** command is:

```
$ spicy -h hostname -s 5901 --spice-ca-file=ca-cert.pem --spice-host-subject="C=XX,L=city,O=organization,CN=hostname" --spice-secure-channels=all
```

vmware

Although it is a bit buggy, it performs better than std and cirrus. Install the VMware drivers **xf86-video-vmware (https://www.archlinux.org/packages/?name=xf86-video-vmware)** and **xf86-input-vmmouse (https://www.archlinux.org/packages/?name=xf86-input-vmmouse)** for Arch Linux guests.

virtio

`virtio-vga` / `virtio-gpu` is a paravirtual 3D graphics driver based on **virgl (https://virgil3d.github.io/)**. Currently a work in progress, supporting only very recent (>= 4.4) Linux guests with **mesa (https://www.archlinux.org/packages/?name=mesa)** (>=11.2) compiled with the option `--with-gallium-drivers=virgl`.

To enable 3D acceleration on the guest system select this vga with `-vga virtio` and enable the opengl context in the display device with `-display sdl,gl=on` or `-display gtk,gl=on` for the sdl and gtk display output respectively. Successful configuration can be confirmed looking at the kernel log in the guest:

```
$ dmesg | grep drm
```

```
[drm] pci: virtio-vga detected
[drm] virgl 3d acceleration enabled
```

As of September 2016, support for the spice protocol is under development and can be tested installing the development release of **spice (https://www.archlinux.org/packages/?name=spice)** (>= 0.13.2) and recompiling qemu.

For more information visit **kraxel's blog (https://www.kraxel.org/blog/tag/virgl/)**.

cirrus

The cirrus graphical adapter was the default **before 2.2 (http://wiki.qemu.org/ChangeLog/2.2#VGA)**. It **should not (https://www.kraxel.org/blog/2014/10/qemu-using-cirrus-considered-harmful/)** be used on modern systems.

none

This is like a PC that has no VGA card at all. You would not even be able to access it with the `-vnc` option. Also, this is different from the `-nographic` option which lets QEMU emulate a VGA card, but disables the SDL display.

vnc

Given that you used the `-nographic` option, you can add the `-vnc display` option to have QEMU listen on `display` and redirect the VGA display to the VNC session. There is an example of this in the **#Starting QEMU virtual machines on boot** section's example configs.

```
$ qemu-system-x86_64 -vga std -nographic -vnc :0
$ gvncviewer :0
```

When using VNC, you might experience keyboard problems described (in gory details) **here (https://www.berrange.com/posts/2010/07/04/more-than-you-or-i-ever-wanted-to-know-about-virtual-keyboard-handling/)**. The solution is *not* to use the `-k` option on QEMU, and to use `gvncviewer` from **`gtk-vnc (https://www.archlinux.org/packages/?name=gtk-vnc)`**. See also **this (http://www.mail-archive.com/libvir-list@redhat.com/msg13340.html)** message posted on libvirt's mailing list.

Audio

Host

The audio driver used by QEMU is set with the `QEMU_AUDIO_DRV` environment variable:

```
$ export QEMU_AUDIO_DRV=pa
```

Run the following command to get QEMU's configuration options related to PulseAudio:

```
$ qemu-system-x86_64 -audio-help | awk '/Name: pa/' RS=
```

The listed options can be exported as environment variables, for example:

```
$ export QEMU_PA_SINK=alsa_output.pci-0000_04_01.0.analog-stereo.monitor
$ export QEMU_PA_SOURCE=input
```

Guest

To get list of the supported emulation audio drivers:

```
$ qemu-system-x86_64 -soundhw help
```

To use e.g. `hda` driver for the guest use the `-soundhw hda` command with QEMU.

Note: Video graphic card emulated drivers for the guest machine may also cause a problem with the sound quality. Test one by one to make it work. You can list possible options with `qemu-system-x86_64 -h | grep vga`.

Installing virtio drivers

QEMU offers guests the ability to use paravirtualized block and network devices using the **virtio (http://wiki.libvirt.org/page/Virtio)** drivers, which provide better performance and lower overhead.

- A virtio block device requires the option `-drive` for passing a disk image, with parameter `if=virtio`:

```
$ qemu-system-x86_64 -boot order=c -drive file=disk_image,if=virtio
```

- Almost the same goes for the network:

```
$ qemu-system-x86_64 -net nic,model=virtio
```

Note: This will only work if the guest machine has drivers for virtio devices. Linux does, and the required drivers are included in Arch Linux, but there is no guarantee that virtio devices will work with other operating systems.

Preparing an (Arch) Linux guest

To use virtio devices after an Arch Linux guest has been installed, the following modules must be loaded in the guest: `virtio`, `virtio_pci`, `virtio_blk`, `virtio_net`, and `virtio_ring`. For 32-bit guests, the specific "virtio" module is not necessary.

If you want to boot from a virtio disk, the initial ramdisk must contain the necessary modules. By default, this is handled by **mkinitcpio**'s `autodetect` hook. Otherwise use the `MODULES` array in `/etc/mkinitcpio.conf` to include the necessary modules and rebuild the initial ramdisk.

/etc/mkinitcpio.conf

```
MODULES="virtio virtio_blk virtio_pci virtio_net"
```

Virtio disks are recognized with the prefix `v` (e.g. `vda`, `vdb`, etc.); therefore, changes must be made in at least `/etc/fstab` and `/boot/grub/grub.cfg` when booting from a virtio disk.

Tip: When referencing disks by **UUID** in both `/etc/fstab` and bootloader, nothing has to be done.

Further information on paravirtualization with KVM can be found **here (http://www.linux-kvm.org/page/Boot_from_virtio_block_device)**.

You might also want to install **`qemu-guest-agent (https://www.archlinux.org/packages/?name=qemu-guest-agent)`** to implement support for QMP commands that will enhance the hypervisor management capabilities. After installing the package you can enable and start the

`qemu-ga.service`.

Preparing a Windows guest

Note: The only (reliable) way to upgrade a Windows 8.1 guest to Windows 10 seems to be to temporarily choose cpu core2duo,nx for the install **[2] (http://ubuntuforums.org/showthread.php?t=2289210)**. After the install, you may revert to other cpu settings (8/8/2015).

Block device drivers

New Install of Windows

Windows does not come with the virtio drivers. Therefore, you will need to load them during installation. There are basically two ways to do this: via Floppy Disk or via ISO files. Both images can be downloaded from the **Fedora repository (https://fedoraproject.org/wiki/Windows_Virtio_Drivers)**.

The floppy disk option is difficult because you will need to press F6 (Shift-F6 on newer Windows) at the very beginning of powering on the QEMU. This is difficult since you need time to connect your VNC console window. You can attempt to add a delay to the boot sequence. See `qemu(1) (https://jlk.fjfi.cvut.cz/arch/manpages/man/qemu.1)` for more details about applying a delay at boot.

The ISO option to load drivers is the preferred way, but it is available only on Windows Vista and Windows Server 2008 and later. The procedure is to load the image with virtio drivers in an additional cdrom device along with the primary disk device and Windows installer:

```
$ qemu-system-x86_64 ... \
-drive file=/path/to/primary/disk.img,index=0,media=disk,if=virtio \
-drive file=/path/to/installer.iso,index=2,media=cdrom \
-drive file=/path/to/virtio.iso,index=3,media=cdrom \
...
```

During the installation, the Windows installer will ask you for your Product key and perform some additional checks. When it gets to the "Where do you want to install Windows?" screen, it will give a warning that no disks are found. Follow the example instructions below (based on Windows Server 2012 R2 with Update).

- Select the option `Load Drivers`.
- Uncheck the box for "Hide drivers that aren't compatible with this computer's hardware".
- Click the Browse button and open the CDROM for the virtio iso, usually named "virtio-win-XX".
- Now browse to `E:\viostor\[your-os]\amd64`, select it, and press OK.
- Click Next

You should now see your virtio disk(s) listed here, ready to be selected, formatted and installed to.

Change Existing Windows VM to use virtio

Modifying an existing Windows guest for booting from virtio disk is a bit tricky.

You can download the virtio disk driver from the **Fedora repository (https://fedoraproject.org/wiki/Windows_Virtio_Drivers)**.

Now you need to create a new disk image, which will force Windows to search for the driver. For example:

```
$ qemu-img create -f qcow2 fake.qcow2 1G
```

Run the original Windows guest (with the boot disk still in IDE mode) with the fake disk (in virtio mode) and a CD-ROM with the driver.

```
$ qemu-system-x86_64 -m 512 -vga std -drive file=windows_disk_image,if=ide -drive file=fake.qcow2,if=virtio -cdrom virtio-win-0.1-81.iso
```

Windows will detect the fake disk and try to find a driver for it. If it fails, go to the *Device Manager*, locate the SCSI drive with an exclamation mark icon (should be open), click *Update driver* and select the virtual CD-ROM. Do not forget to select the checkbox which says to search for directories recursively.

When the installation is successful, you can turn off the virtual machine and launch it again, now with the boot disk attached in virtio mode:

```
$ qemu-system-x86_64 -m 512 -vga std -drive file=windows_disk_image,if=virtio
```

Note: If you encounter the Blue Screen of Death, make sure you did not forget the `-m` parameter, and that you do not boot with virtio instead of ide for the system drive before drivers are installed.

Network drivers

Installing virtio network drivers is a bit easier, simply add the `-net` argument as explained above.

```
$ qemu-system-x86_64 -m 512 -vga std -drive file=windows_disk_image,if=virtio -net nic,model=virtio -cdrom virtio-win-0.1-74.iso
```

Windows will detect the network adapter and try to find a driver for it. If it fails, go to the *Device Manager*, locate the network adapter with an exclamation mark icon (should be open), click *Update driver* and select the virtual CD-ROM. Do not forget to select the checkbox which says to search for directories recursively.

Balloon driver

If you want to track you guest memory state (for example via `virsh` command `dommemstat`) or change guest's memory size in runtime (you still won't be able to change memory size, but can limit memory usage via inflating balloon driver) you will need to install guest balloon driver.

For this you will need to go to *Device Manager*, locate *PCI standard RAM Controller* in *System devices* (or unrecognized PCI controller from *Other devices*) and choose *Update driver*. In opened window you will need to choose *Browse my computer...* and select the CD-ROM (and don't forget the *Include subdirectories* checkbox). Reboot after installation. This will install the driver and you will be able to inflate the balloon (for example via hmp command `balloon memory_size`, which will cause balloon to take as much memory as possible in order to shrink the guest's available memory size to *memory_size*). However, you still won't be able to track guest memory state. In order to do this you will need to install *Balloon* service properly. For that open command line as administrator, go to the CD-ROM, *Balloon* directory and deeper, depending on your system and architecture. Once you are in *amd64* (*x86*) directory, run `blnsrv.exe -i` which will do the installation. After that `virsh` command `dommemstat` should be outputting all supported values.

Preparing a FreeBSD guest

Install the `emulators/virtio-kmod` port if you are using FreeBSD 8.3 or later up until 10.0-CURRENT where they are included into the kernel. After installation, add the following to your `/boot/loader.conf` file:

```
virtio_loader="YES"
virtio_pci_load="YES"
virtio_blk_load="YES"
if_vtnet_load="YES"
virtio_balloon_load="YES"
```

Then modify your `/etc/fstab` by doing the following:

```
sed -i bak "s/ada/vtbd/g" /etc/fstab
```

And verify that `/etc/fstab` is consistent. If anything goes wrong, just boot into a rescue CD and copy `/etc/fstab.bak` back to `/etc/fstab`.

QEMU Monitor

While QEMU is running, a monitor console is provided in order to provide several ways to interact with the virtual machine running. The QEMU Monitor offers interesting capabilities such as obtaining information about the current virtual machine, hotplugging devices, creating snapshots of the current state of the virtual machine, etc. To see the list of all commands, run `help` or `?` in the QEMU monitor console or review the relevant section of the **official QEMU documentation (https://qemu.weilnetz.de/doc/qemu-doc.html#pcsys_005fmonitor)**.

Accessing the monitor console

When using the `std` default graphics option, one can access the QEMU Monitor by pressing `Ctrl+Alt+2` or by clicking *View* > *compatmonitor0* in the QEMU window. To return to the virtual machine graphical view either press `Ctrl+Alt+1` or click *View* > *VGA*.

However, the standard method of accessing the monitor is not always convenient and does not work in all graphic outputs QEMU supports. Alternative options of accessing the monitor are described below:

- **telnet**: Run QEMU with the `-monitor telnet:127.0.0.1:port,server,nowait` parameter. When the virtual machine is started you will be able to access the monitor via telnet:

```
$ telnet 127.0.0.1 port
```

Note: If `127.0.0.1` is specified as the IP to listen it will be only possible to connect to the monitor from the same host QEMU is running on. If connecting from remote hosts is desired, QEMU must be told to listen `0.0.0.0` as follows: `-monitor telnet:0.0.0.0:port,server,nowait`. Keep in mind that it is recommended to have a **firewall** configured in this case or make sure your local network is completely trustworthy since this connection is completely unauthenticated and unencrypted.

- UNIX socket: Run QEMU with the `-monitor unix:socketfile,server,nowait` parameter. Then you can connect with either **socat (https://www.archlinux.org/packages/?name=socat)** or **openbsd-netcat (https://www.archlinux.org/packages/?name=openbsd-netcat)**.

For example, if QEMU is run via:

```
$ qemu-system-x86_64 [...] -monitor unix:/tmp/monitor.sock,server,nowait [...]
```

It is possible to connect to the monitor with:

```
$ socat - UNIX-CONNECT:/tmp/monitor.sock
```

Or with:

```
$ nc -U /tmp/monitor.sock
```

- TCP: You can expose the monitor over TCP with the argument `-monitor tcp:127.0.0.1:port,server,nowait`. Then connect with netcat, either **openbsd-netcat (https://www.archlinux.org/packages/?name=openbsd-netcat)** or **gnu-netcat (https://www.archlinux.org/packages/?name=gnu-netcat)** by running:

```
$ nc 127.0.0.1 port
```

Note: In order to be able to connect to the tcp socket from other devices other than the same host QEMU is being run on you need to listen to `0.0.0.0` like explained in the telnet case. The same security warnings apply in this case as well.

- Standard I/O: It is possible to access the monitor automatically from the same terminal QEMU is being run by running it with the argument `-monitor stdio`.

Sending keyboard presses to the virtual machine using the monitor console

Some combinations of keys may be difficult to perform on virtual machines due to the host intercepting them instead in some configurations (a notable example is the `Ctrl+Alt+F*` key combinations, which change the active tty). To avoid this problem, the problematic combination of keys may be sent via the monitor console instead. Switch to the monitor and use the `sendkey` command to forward the necessary keypresses to the virtual machine. For example:

```
(qemu) sendkey ctrl-alt-f2
```

Creating and managing snapshots via the monitor console

Note: This feature will **only** work when the virtual machine disk image is in *qcow2* format. It will not work with *raw* images.

It is sometimes desirable to save the current state of a virtual machine and having the possibility of reverting the state of the virtual machine to that of a previously saved snapshot at any time. The QEMU monitor console provides the user with the necessary utilities to create snapshots, manage them, and revert the machine state to a saved snapshot.

- Use `savevm name` in order to create a snapshot with the tag *name*.
- Use `loadvm name` to revert the virtual machine to the state of the snapshot *name*.
- Use `delvm name` to delete the snapshot tagged as *name*.
- Use `info snapshots` to see a list of saved snapshots. Snapshots are identified by both an auto-incremented ID number and a text tag (set by the user on snapshot creation).

Running the virtual machine in immutable mode

It is possible to run a virtual machine in a frozen state so that all changes will be discarded when the virtual machine is powered off just by running QEMU with the `-snapshot` parameter. When the disk image is written by the guest, changes will be saved in a temporary file in `/tmp` and will be discarded when QEMU halts.

However, if a machine is running in frozen mode it is still possible to save the changes to the disk image if it is afterwards desired by using the monitor console and running the following command:

```
(qemu) commit
```

If snapshots are created when running in frozen mode they will be discarded as soon as QEMU is exited unless changes are explicitly commited to disk, as well.

Pause and power options via the monitor console

Some operations of a physical machine can be emulated by QEMU using some monitor commands:

- `system_powerdown` will send an ACPI shutdown request to the virtual machine. This effect is similar to the power button in a physical machine.

- `system_reset` will reset the virtual machine similarly to a reset button in a physical machine. This operation can cause data loss and file system corruption since the virtual machine is not cleanly restarted.
- `stop` will pause the virtual machine.
- `cont` will resume a virtual machine previously paused.

Taking screenshots of the virtual machine

Screenshots of the virtual machine graphic display can be obtained in the PPM format by running the following command in the monitor console:

```
(qemu) screendump file.ppm
```

Tips and tricks

Starting QEMU virtual machines on boot

With libvirt

If a virtual machine is set up with **libvirt**, it can be configured with `virsh autostart` or through the *virt-manager* GUI to start at host boot by going to the Boot Options for the virtual machine and selecting "Start virtual machine on host boot up".

Custom script

To run QEMU VMs on boot, you can use following systemd unit and config.

/etc/systemd/system/qemu@.service

```
[Unit]
Description=QEMU virtual machine

[Service]
Environment="type=system-x86_64" "haltcmd=kill -INT $MAINPID"
EnvironmentFile=/etc/conf.d/qemu.d/%i
PIDFile=/tmp/%i.pid
ExecStart=/usr/bin/env qemu-${type} -name %i -nographic -pidfile /tmp/%i.pid $args
ExecStop=/bin/sh -c ${haltcmd}
TimeoutStopSec=30
KillMode=none

[Install]
WantedBy=multi-user.target
```

Note:

- According to **systemd.service(5) (https://jlk.fjfi.cvut.cz/arch/manpages/man/systemd.service.5)** and `5` man pages it is necessary to use the `KillMode=none` option. Otherwise the main qemu process will be killed immediately after the `ExecStop` command quits (it simply echoes one string) and your quest system will not be able to shutdown correctly.
- It is necessary to use the `PIDFile` option. Otherwise systemd cannot tell whether the main qemu process was terminated and your quest system will not be able to shutdown correctly. On host shutdown it will proceed without waiting for the VM to shutdown.

Then create per-VM configuration files, named `/etc/conf.d/qemu.d/vm_name`, with the following variables set:

type
: QEMU binary to call. If specified, will be prepended with `/usr/bin/qemu-` and that binary will be used to start the VM. I.e. you can boot e.g. `qemu-system-arm` images with `type="system-arm"`.

args
: QEMU command line to start with. Will always be prepended with `-name ${vm} -nographic`.

haltcmd
: Command to shut down a VM safely. In this example, the QEMU monitor is exposed via telnet using `-monitor telnet:..` and the VMs are powered off via ACPI by sending `system_powerdown` to monitor with the `nc` command. You can use SSH or some other ways as well.

Example configs:

/etc/conf.d/qemu.d/one

```
type="system-x86_64"

args="-enable-kvm -m 512 -hda /dev/mapper/vg0-vm1 -net nic,macaddr=DE:AD:BE:EF:E0:00 \
 -net tap,ifname=tap0 -serial telnet:localhost:7000,server,nowait,nodelay \
 -monitor telnet:localhost:7100,server,nowait,nodelay -vnc :0"

haltcmd="echo 'system_powerdown' | nc localhost 7100" # or netcat/ncat

# You can use other ways to shut down your VM correctly
#haltcmd="ssh powermanager@vm1 sudo poweroff"
```

```
/etc/conf.d/qemu.d/two
```

```
args="-enable-kvm -m 512 -hda /srv/kvm/vm2.img -net nic,macaddr=DE:AD:BE:EF:E0:01 \
 -net tap,ifname=tap1 -serial telnet:localhost:7001,server,nowait,nodelay \
 -monitor telnet:localhost:7101,server,nowait,nodelay -vnc :1"

haltcmd="echo 'system_powerdown' | nc localhost 7101"
```

To set which virtual machines will start on boot-up, **enable** the `qemu@vm_name.service` systemd unit.

Mouse integration

To prevent the mouse from being grabbed when clicking on the guest operating system's window, add the options `-usb -device usb-tablet`. This means QEMU is able to report the mouse position without having to grab the mouse. This also overrides PS/2 mouse emulation when activated. For example:

```
$ qemu-system-x86_64 -hda disk_image -m 512 -vga std -usb -device usb-tablet
```

If that does not work, try the tip at **#Mouse cursor is jittery or erratic**.

Pass-through host USB device

To access physical USB device connected to host from VM, you can use the option: `-usbdevice host:vendor_id:product_id`.

You can find `vendor_id` and `product_id` of your device with `lsusb` command.

Since the default I440FX chipset emulated by qemu feature a single UHCI controller (USB 1), the `-usbdevice` option will try to attach your physical device to it. In some cases this may cause issues with newer devices. A possible solution is to emulate the **ICH9 (http://wiki.qemu.org/Features/Q35)** chipset, which offer an EHCI controller supporting up to 12 devices, using the option `-machine type=q35`.

A less invasive solution is to emulate an EHCI (USB 2) or XHCI (USB 3) controller with the option `-device usb-ehci,id=ehci` or `-device nec-usb-xhci,id=xhci` respectively and then attach your physical device to it with the option `-device usb-host,..` as follows:

```
-device usb-host,bus=controller_id.0,vendorid=0xvendor_id,productid=0xproduct_id
```

You can also add the `...,port=<n>` setting to the previous option to specify in which physical port of the virtual controller you want to attach your device, useful in the case you want to add multiple usb devices to the VM.

Note: If you encounter permission errors when running QEMU, see **Udev#Writing udev rules** for information on how to set permissions of the device.

USB redirection with SPICE

When using **#SPICE** it is possible to redirect USB devices from the client to the virtual machine without needing to specify them in the QEMU command. It is possible to configure the number of USB slots available for redirected devices (the number of slots will determine the maximum number of devices which can be redirected simultaneously). The main advantages of using SPICE for redirection compared to the previously-mentioned `-usbdevice` method is the possibility of hot-swapping USB devices after the virtual machine has started, without needing to halt it in order to remove USB devices from the redirection or adding new ones. This method of USB redirection also allows us to redirect USB devices over the network, from the client to the server. In summary, it is the most flexible method of using USB devices in a QEMU virtual machine.

We need to add one EHCI/UHCI controller per available USB redirection slot desired as well as one SPICE redirection channel per slot. For example, adding the following arguments to the QEMU command you use for starting the virtual machine in SPICE mode will start the virtual machine with three available USB slots for redirection:

```
-device ich9-usb-ehci1,id=usb \
-device ich9-usb-uhci1,masterbus=usb.0,firstport=0,multifunction=on \
-device ich9-usb-uhci2,masterbus=usb.0,firstport=2 \
-device ich9-usb-uhci3,masterbus=usb.0,firstport=4 \
-chardev spicevmc,name=usbredir,id=usbredirchardev1 \
-device usb-redir,chardev=usbredirchardev1,id=usbredirdev1 \
-chardev spicevmc,name=usbredir,id=usbredirchardev2 \
-device usb-redir,chardev=usbredirchardev2,id=usbredirdev2 \
-chardev spicevmc,name=usbredir,id=usbredirchardev3 \
-device usb-redir,chardev=usbredirchardev3,id=usbredirdev3
```

Both `spicy` from **spice-gtk (https://www.archlinux.org/packages/?name=spice-gtk)** (*Input > Select USB Devices for redirection*) and `remote-viewer` from **virt-viewer (https://www.archlinux.org/packages/?name=virt-viewer)** (*File > USB device selection*) support this feature. Please make sure that you have installed the necessary SPICE Guest Tools on the virtual machine for this functionality to work as expected (see the **#SPICE** section for more information).

Warning: Keep in mind that when a USB device is redirected from the client, it will not be usable from the client operating system itself until the redirection is stopped. It is specially important to never redirect the input devices (namely mouse and keyboard), since it will be then difficult to access the SPICE client menus to revert the situation, because the client will not respond to the input devices after being redirected to the virtual machine.

Enabling KSM

Kernel Samepage Merging (KSM) is a feature of the Linux kernel that allows for an application to register with the kernel to have its pages merged with other processes that also register to have their pages merged. The KSM mechanism allows for guest virtual machines to share pages with each other. In an environment where many of the guest operating systems are similar, this can result in significant memory savings.

To enable KSM, simply run

```
# echo 1 > /sys/kernel/mm/ksm/run
```

To make it permanent, you can use **systemd's temporary files**:

```
/etc/tmpfiles.d/ksm.conf
w /sys/kernel/mm/ksm/run - - - - 1
```

If KSM is running, and there are pages to be merged (i.e. at least two similar VMs are running), then `/sys/kernel/mm/ksm/pages_shared` should be non-zero. See **https://www.kernel.org/doc/Documentation/vm/ksm.txt** for more information.

Tip: An easy way to see how well KSM is performing is to simply print the contents of all the files in that directory:

```
$ grep . /sys/kernel/mm/ksm/*
```

Multi-monitor support

The Linux QXL driver supports four heads (virtual screens) by default. This can be changed via the `qxl.heads=N` kernel parameter.

The default VGA memory size for QXL devices is 16M (VRAM size is 64M). This is not sufficient if you would like to enable two 1920x1200 monitors since that requires 2 × 1920 × 4 (color depth) × 1200 = 17.6 MiB VGA memory. This can be changed by replacing `-vga qxl`

by `-vga none -device qxl-vga,vgamem_mb=32`. If you ever increase vgamem_mb beyond 64M, then you also have to increase the `vram_size_mb` option.

Copy and paste

To have copy and paste between the host and the guest you need to enable the spice agent communication channel. It requires to add a virtio-serial device to the guest, and open a port for the spice vdagent. It is also required to install the spice vdagent in guest (**`spice-vdagent (https://www.archlinux.org/packages/?name=spice-vdagent)`** for Arch guests, **Windows guest tools (http://www.spice-space.org/download.html)** for Windows guests). Make sure the agent is running (and for future, started automatically). See **#SPICE** for the necessary procedure to use QEMU with the SPICE protocol.

Windows-specific notes

QEMU can run any version of Windows from Windows 95 through Windows 10.

It is possible to run **Windows PE** in QEMU.

Fast startup

Note: An administrator account is required to change power settings.

For Windows 8 (or later) guests it is better to disable "Turn on fast startup (recommended)" from the Power Options of the Control Panel as explained in the following **forum page (https://www.tenforums.com/tutorials/4189-turn-off-fast-startup-windows-10-a.html)**, as it causes the guest to hang during every other boot.

Fast Startup may also need to be disabled for changes to the `-smp` option to be properly applied.

Remote Desktop Protocol

If you use a MS Windows guest, you might want to use RDP to connect to your guest VM. If you are using a VLAN or are not in the same network as the guest, use:

```
$ qemu-system-x86_64 -nographic -net user,hostfwd=tcp::5555-:3389
```

Then connect with either **rdesktop (https://www.archlinux.org/packages/?name=rdesktop)** or **freerdp (https://www.archlinux.org/packages/?name=freerdp)** to the guest. For example:

```
$ xfreerdp -g 2048x1152 localhost:5555 -z -x lan
```

Troubleshooting

Virtual machine runs too slowly

There are a number of techniques that you can use to improve the performance of your virtual machine. For example:

- Use the `-cpu host` option to make QEMU emulate the host's exact CPU. If you do not do this, it may be trying to emulate a more generic CPU.
- Especially for Windows guests, enable **Hyper-V enlightenments (http://blog.wikichoon.com/2014/07/enabling-hyper-v-enlightenments-with-kvm.html)**: `-cpu host,hv_relaxed,hv_spinlocks=0x1fff,hv_vapic,hv_time`.
- If the host machine has multiple CPUs, assign the guest more CPUs using the `-smp` option.
- Make sure you have assigned the virtual machine enough memory. By default, QEMU only assigns 128 MiB of memory to each virtual machine. Use the `-m` option to assign more memory. For example, `-m 1024` runs a virtual machine with 1024 MiB of memory.
- Use KVM if possible: add `-machine type=pc,accel=kvm` to the QEMU start command you use.
- If supported by drivers in the guest operating system, use **virtio (http://wiki.libvirt.org/page/Virtio)** for network and/or block devices. For example:

```
$ qemu-system-x86_64 -net nic,model=virtio -net tap,if=tap0,script=no -drive file=disk_image,media=disk,if=virtio
```

- Use TAP devices instead of user-mode networking. See **#Tap networking with QEMU**.

- If the guest OS is doing heavy writing to its disk, you may benefit from certain mount options on the host's file system. For example, you can mount an **ext4 file system** with the option `barrier=0`. You should read the documentation for any options that you change because sometimes performance-enhancing options for file systems come at the cost of data integrity.
- If you have a raw disk image, you may want to disable the cache:

```
$ qemu-system-x86_64 -drive file=disk_image,if=virtio,cache=none
```

- Use the native Linux AIO:

```
$ qemu-system-x86_64 -drive file=disk_image,if=virtio,aio=native,cache.direct=on
```

- If you use a qcow2 disk image, I/O performance can be improved considerably by ensuring that the L2 cache is of sufficient size. The **formula (https://blogs.igalia.com/berto/2015/12/17/improving-disk-io-performance-in-qemu-2-5-with-the-qcow2-l2-cache/)** to calculate L2 cache is: l2_cache_size = disk_size * 8 / cluster_size. Assuming the qcow2 image was created with the default cluster size of 64K, this means that for every 8 GB in size of the qcow2 image, 1 MB of L2 cache is best for performance. Only 1 MB is used by QEMU by default; specifying a larger cache is done on the QEMU command line. For instance, to specify 4 MB of cache (suitable for a 32 GB disk with a cluster size of 64K):

```
$ qemu-system-x86_64 -drive file=disk_image,format=qcow2,l2-cache-size=4M
```

- If you are running multiple virtual machines concurrently that all have the same operating system installed, you can save memory by enabling **kernel same-page merging**. See **#Enabling KSM**.
- In some cases, memory can be reclaimed from running virtual machines by running a memory ballooning driver in the guest operating system and launching QEMU with the `-balloon virtio` option.
- It is possible to use a emulation layer for an ICH-9 AHCI controller (although it may be unstable). The AHCI emulation supports **NCQ**, so multiple read or write requests can be outstanding at the same time:

```
$ qemu-system-x86_64 -drive id=disk,file=disk_image,if=none -device ich9-ahci,id=ahci -device ide-drive,drive=disk,bus=ahci.0
```

See **http://www.linux-kvm.org/page/Tuning_KVM** for more information.

Mouse cursor is jittery or erratic

If the cursor jumps around the screen uncontrollably, entering this on the terminal before starting QEMU might help:

```
$ export SDL_VIDEO_X11_DGAMOUSE=0
```

If this helps, you can add this to your `~/.bashrc` file.

No visible Cursor

Add `-show-cursor` to QEMU's options to see a mouse cursor.

If that still does not work, make sure you have set your display device appropriately.

For example: `-vga qxl`

Unable to move/attach Cursor

Replace `-usbdevice tablet` with `-usb` as QEMU option.

Keyboard seems broken or the arrow keys do not work

Should you find that some of your keys do not work or "press" the wrong key (in particular, the arrow keys), you likely need to specify your keyboard layout as an option. The keyboard layouts can be found in `/usr/share/qemu/keymaps`.

```
$ qemu-system-x86_64 -k keymap disk_image
```

Guest display stretches on window resize

To restore default window size, press `Ctrl+Alt+u`.

ioctl(KVM_CREATE_VM) failed: 16 Device or resource busy

If an error message like this is printed when starting QEMU with `-enable-kvm` option:

```
ioctl(KVM_CREATE_VM) failed: 16 Device or resource busy
failed to initialize KVM: Device or resource busy
```

that means another **hypervisor** is currently running. It is not recommended or possible to run several hypervisors in parallel.

libgfapi error message

The error message displayed at startup:

```
Failed to open module: libgfapi.so.0: cannot open shared object file: No such file or directory
```

Install `glusterfs (https://www.archlinux.org/packages/?name=glusterfs)` or ignore the error message as GlusterFS is a optional dependency.

Kernel panic on LIVE-environments

If you start a live-environment (or better: booting a system) you may encounter this:

```
[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown block(0,0)
```

or some other boot hindering process (e.g. cannot unpack initramfs, cant start service foo). Try starting the VM with the `-m VALUE` switch and an appropriate amount of RAM, if the ram is to low you will probably encounter similar issues as above/without the memory-switch.

Windows 7 guest suffers low-quality sound

Using the `hda` audio driver for Windows 7 guest may result in low-quality sound. Changing the audio driver to `ac97` by passing the `-soundhw ac97` arguments to QEMU and installing the AC97 driver from **Realtek AC'97 Audio Codecs (http://www.realtek.com.tw/downloads/downloadsView.aspx?Langid=1&PNid=14&PFid=23&Level=4&Conn=3&DownTypeID=3&GetDown=false)** in the guest may solve the problem. See **Red Hat Bugzilla – Bug 1176761 (https://bugzilla.redhat.com/show_bug.cgi?id=1176761#c16)** for more information.

Could not access KVM kernel module: Permission denied

If you encounter the following error:

```
 libvirtError: internal error: process exited while connecting to monitor: Could not access KVM kernel module: Permission denied failed to initialize KVM: Permission denied
```

Systemd 234 assign it a dynamic id to group kvm (see **bug (https://bugs.archlinux.org/task/54943)**). A workground for avoid this error, you need edit the file /etc/libvirt/qemu.conf and change the line:

```
group = "78"
```

to

```
group = "kvm"
```

"System Thread Exception Not Handled" when booting a Windows VM

Windows 8 or Windows 10 guests may raise a generic compatibility exception at boot, namely "System Thread Exception Not Handled", which tends to be caused by legacy drivers acting strangely on real machines. On KVM machines this issue can generally be solved by setting the CPU model to `core2duo`.

Certain Windows games/applications crashing/causing a bluescreen

Occasionally, applications running in the VM may crash unexpectedly, whereas they'd run normally on a physical machine. If, while running `dmesg -wH`, you encounter an error mentioning `MSR`, the reason for those crashes is that KVM injects a **General protection fault** (GPF) when the guest tries to access unsupported **Model-specific registers** (MSRs) - this often results in guest applications/OS crashing. A number of those issues can be solved by passing the `ignore_msrs=1` option to the KVM module, which will ignore unimplemented MSRs.

/etc/modprobe.d/kvm.conf

```
...
options kvm ignore_msrs=1
...
```

Cases where adding this option might help:

- GeForce Experience complaining about an unsupported CPU being present.
- StarCraft 2 and L.A. Noire reliably blue-screening Windows 10 with `KMODE_EXCEPTION_NOT_HANDLED`. The blue screen information does not identify a driver file in these cases.

Warning: While this is normally safe and some applications might not work without this, silently ignoring unknown MSR accesses could potentially break other software within the VM or other VMs.

See also

- **Official QEMU website (http://qemu.org)**
- **Official KVM website (http://www.linux-kvm.org)**
- **QEMU Emulator User Documentation (http://qemu.weilnetz.de/qemu-doc.html)**
- **QEMU Wikibook (https://en.wikibooks.org/wiki/QEMU)**
- **Hardware virtualization with QEMU (http://alien.slackbook.org/dokuwiki/doku.php?id=slackware:qemu)** by AlienBOB (last updated in 2008)

- **Building a Virtual Army (http://blog.falconindy.com/articles/build-a-virtual-army.html)** by Falconindy
- **Lastest docs (http://git.qemu.org/?p=qemu.git;a=tree;f=docs)**
- **QEMU on Windows (http://qemu.weilnetz.de/)**
- **Wikipedia**
- **Debian Wiki - QEMU**
- **QEMU Networking on gnome.org (https://people.gnome.org/~markmc/qemu-networking.html)**
- **Networking QEMU Virtual BSD Systems (http://bsdwiki.reedmedia.net/wiki/networking_qemu_virtual_bsd_systems.html)**
- **QEMU on gnu.org (https://www.gnu.org/software/hurd/hurd/running/qemu.html)**
- **QEMU on FreeBSD as host (https://wiki.freebsd.org/qemu)**
- **KVM/QEMU Virtio Tuning and SSD VM Optimization Guide (https://wiki.mikejung.biz/KVM_/_Xen)**
- **Managing Virtual Machines with QEMU - OpenSUSE documentation (https://doc.opensuse.org/documentation/leap/virtualization/html/book.virt/part.virt.qemu.html)**
- **KVM on IBM Knowledge Center (https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaat/liaatkvm.htm)**
