NAME

qemu−block−drivers − QEMU block drivers reference

SYNOPSIS

QEMU block driver reference manual

DESCRIPTION

Disk image file formats

QEMU supports many image file formats that can be used with VMs as well as with any of the tools (like `qemu-img`). This includes the preferred formats raw and qcow2 as well as formats that are supported for compatibility with older QEMU versions or other hypervisors.

Depending on the image format, different options can be passed to `qemu-img create` and `qemu-img convert` using the `-o` option. This section describes each format and the options that are supported for it.

raw

Raw disk image format. This format has the advantage of being simple and easily exportable to all other emulators. If your file system supports *holes* (for example in ext2 or ext3 on Linux or NTFS on Windows), then only the written sectors will reserve space. Use `qemu-img info` to know the real size used by the image or `ls -ls` on Unix/Linux.

Supported options:

`preallocation`

Preallocation mode (allowed values: `off`, `falloc`, `full`). `falloc` mode preallocates space for image by calling **posix_fallocate()**. `full` mode preallocates space for image by writing data to underlying storage. This data may or may not be zero, depending on the storage location.

qcow2

QEMU image format, the most versatile format. Use it to have smaller images (useful if your filesystem does not supports holes, for example on Windows), zlib based compression and support of multiple VM snapshots.

Supported options:

`compat`

Determines the qcow2 version to use. `compat=0.10` uses the traditional image format that can be read by any QEMU since 0.10. `compat=1.1` enables image format extensions that only QEMU 1.1 and newer understand (this is the default). Amongst others, this includes zero clusters, which allow efficient copy-on-read for sparse images.

`backing_file`

File name of a base image (see **create** subcommand)

`backing_fmt`

Image format of the base image

`encryption`

This option is deprecated and equivalent to `encrypt.format=aes`

`encrypt.format`

If this is set to `luks`, it requests that the qcow2 payload (not qcow2 header) be encrypted using the LUKS format. The passphrase to use to unlock the LUKS key slot is given by the `encrypt.key-secret` parameter. LUKS encryption parameters can be tuned with the other `encrypt.*` parameters.

If this is set to `aes`, the image is encrypted with 128−bit AES-CBC. The encryption key is given by the `encrypt.key-secret` parameter. This encryption format is considered to be flawed by modern cryptography standards, suffering from a number of design problems:

−<The AES-CBC cipher is used with predictable initialization vectors based>

on the sector number. This makes it vulnerable to chosen plaintext attacks which can reveal the existence of encrypted data.

–<The user passphrase is directly used as the encryption key. A poorly>
chosen or short passphrase will compromise the security of the encryption.

–<In the event of the passphrase being compromised there is no way to>
change the passphrase to protect data in any qcow images. The files must be cloned, using a different encryption passphrase in the new file. The original file must then be securely erased using a program like shred, though even this is ineffective with many modern storage technologies.

The use of this is no longer supported in system emulators. Support only remains in the command line utilities, for the purposes of data liberation and interoperability with old versions of QEMU. The `luks` format should be used instead.

`encrypt.key-secret`
Provides the ID of a `secret` object that contains the passphrase (`encrypt.format=luks`) or encryption key (`encrypt.format=aes`).

`encrypt.cipher-alg`
Name of the cipher algorithm and key length. Currently defaults to `aes-256`. Only used when `encrypt.format=luks`.

`encrypt.cipher-mode`
Name of the encryption mode to use. Currently defaults to `xts`. Only used when `encrypt.format=luks`.

`encrypt.ivgen-alg`
Name of the initialization vector generator algorithm. Currently defaults to `plain64`. Only used when `encrypt.format=luks`.

`encrypt.ivgen-hash-alg`
Name of the hash algorithm to use with the initialization vector generator (if required). Defaults to `sha256`. Only used when `encrypt.format=luks`.

`encrypt.hash-alg`
Name of the hash algorithm to use for PBKDF algorithm Defaults to `sha256`. Only used when `encrypt.format=luks`.

`encrypt.iter-time`
Amount of time, in milliseconds, to use for PBKDF algorithm per key slot. Defaults to `2000`. Only used when `encrypt.format=luks`.

`cluster_size`
Changes the qcow2 cluster size (must be between 512 and 2M). Smaller cluster sizes can improve the image file size whereas larger cluster sizes generally provide better performance.

`preallocation`
Preallocation mode (allowed values: `off`, `metadata`, `falloc`, `full`). An image with preallocated metadata is initially larger but can improve performance when the image needs to grow. `falloc` and `full` preallocations are like the same options of `raw` format, but sets up metadata also.

`lazy_refcounts`
If this option is set to `on`, reference count updates are postponed with the goal of avoiding metadata I/O and improving performance. This is particularly interesting with **cache=writethrough** which doesn't batch metadata updates. The tradeoff is that after a host crash, the reference count tables must be rebuilt, i.e. on the next open an (automatic) `qemu-img check -r all` is required, which may take some time.

This option can only be enabled if `compat=1.1` is specified.

`nocow`
If this option is set to `on`, it will turn off COW of the file. It's only valid on btrfs, no effect on other file systems.

Btrfs has low performance when hosting a VM image file, even more when the guest on the VM also using btrfs as file system. Turning off COW is a way to mitigate this bad performance. Generally there are two ways to turn off COW on btrfs: a) Disable it by mounting with nodatacow, then all newly created files will be NOCOW. b) For an empty file, add the NOCOW file attribute. That's what this option does.

Note: this option is only valid to new or empty files. If there is an existing file which is COW and has data blocks already, it couldn't be changed to NOCOW by setting `nocow=on`. One can issue `lsattr filename` to check if the NOCOW flag is set or not (Capital 'C' is NOCOW flag).

qed

Old QEMU image format with support for backing files and compact image files (when your filesystem or transport medium does not support holes).

When converting QED images to qcow2, you might want to consider using the `lazy_refcounts=on` option to get a more QED-like behaviour.

Supported options:

`backing_file`
File name of a base image (see **create** subcommand).

`backing_fmt`
Image file format of backing file (optional). Useful if the format cannot be autodetected because it has no header, like some vhd/vpc files.

`cluster_size`
Changes the cluster size (must be power−of−2 between 4K and 64K). Smaller cluster sizes can improve the image file size whereas larger cluster sizes generally provide better performance.

`table_size`
Changes the number of clusters per L1/L2 table (must be power−of−2 between 1 and 16). There is normally no need to change this value but this option can be used for performance benchmarking.

qcow

Old QEMU image format with support for backing files, compact image files, encryption and compression.

Supported options:

`backing_file`
File name of a base image (see **create** subcommand)

`encryption`
This option is deprecated and equivalent to `encrypt.format=aes`

`encrypt.format`
If this is set to `aes`, the image is encrypted with 128−bit AES-CBC. The encryption key is given by the `encrypt.key−secret` parameter. This encryption format is considered to be flawed by modern cryptography standards, suffering from a number of design problems enumerated previously against the `qcow2` image format.

The use of this is no longer supported in system emulators. Support only remains in the command line utilities, for the purposes of data liberation and interoperability with old versions of QEMU.

Users requiring native encryption should use the `qcow2` format instead with `encrypt.format=luks`.

`encrypt.key−secret`
Provides the ID of a `secret` object that contains the encryption key (`encrypt.format=aes`).

luks

LUKS v1 encryption format, compatible with Linux dm−crypt/cryptsetup

Supported options:

`key−secret`
Provides the ID of a `secret` object that contains the passphrase.

`cipher−alg`
Name of the cipher algorithm and key length. Currently defaults to `aes−256`.

`cipher−mode`
Name of the encryption mode to use. Currently defaults to `xts`.

`ivgen−alg`
Name of the initialization vector generator algorithm. Currently defaults to `plain64`.

`ivgen−hash−alg`
Name of the hash algorithm to use with the initialization vector generator (if required). Defaults to `sha256`.

`hash−alg`
Name of the hash algorithm to use for PBKDF algorithm Defaults to `sha256`.

`iter−time`
Amount of time, in milliseconds, to use for PBKDF algorithm per key slot. Defaults to `2000`.

vdi VirtualBox 1.1 compatible image format. Supported options:

`static`
If this option is set to `on`, the image is created with metadata preallocation.

vmdk

VMware 3 and 4 compatible image format.

Supported options:

`backing_file`
File name of a base image (see **create** subcommand).

`compat6`
Create a VMDK version 6 image (instead of version 4)

`hwversion`
Specify vmdk virtual hardware version. Compat6 flag cannot be enabled if hwversion is specified.

`subformat`
Specifies which VMDK subformat to use. Valid options are `monolithicSparse` (default), `monolithicFlat`, `twoGbMaxExtentSparse`, `twoGbMaxExtentFlat` and `streamOptimized`.

vpc VirtualPC compatible image format (VHD). Supported options:

`subformat`
Specifies which VHD subformat to use. Valid options are `dynamic` (default) and `fixed`.

VHDX

Hyper-V compatible image format (VHDX). Supported options:

`subformat`
Specifies which VHDX subformat to use. Valid options are `dynamic` (default) and `fixed`.

`block_state_zero`
Force use of payload blocks of type 'ZERO'. Can be set to `on` (default) or `off`. When set to `off`, new blocks will be created as `PAYLOAD_BLOCK_NOT_PRESENT`, which means parsers are free to return arbitrary data for those blocks. Do not set to `off` when using `qemu−img`

`convert` with `subformat=dynamic`.

`block_size`
Block size; min 1 MB, max 256 MB. 0 means auto-calculate based on image size.

`log_size`
Log size; min 1 MB.

Read-only formats

More disk image file formats are supported in a read-only mode.

bochs
Bochs images of `growing` type.

cloop
Linux Compressed Loop image, useful only to reuse directly compressed CD-ROM images present for example in the Knoppix CD-ROMs.

dmg
Apple disk image.

parallels
Parallels disk image format.

Using host drives

In addition to disk image files, QEMU can directly access host devices. We describe here the usage for QEMU version >= 0.8.3.

Linux

On Linux, you can directly use the host device filename instead of a disk image filename provided you have enough privileges to access it. For example, use */dev/cdrom* to access to the CDROM.

`CD` You can specify a CDROM device even if no CDROM is loaded. QEMU has specific code to detect CDROM insertion or removal. CDROM ejection by the guest OS is supported. Currently only data CDs are supported.

`Floppy`
You can specify a floppy device even if no floppy is loaded. Floppy removal is currently not detected accurately (if you change floppy without doing floppy access while the floppy is not loaded, the guest OS will think that the same floppy is loaded). Use of the host's floppy device is deprecated, and support for it will be removed in a future release.

`Hard disks`
Hard disks can be used. Normally you must specify the whole disk (*/dev/hdb* instead of */dev/hdb1*) so that the guest OS can see it as a partitioned disk. WARNING: unless you know what you do, it is better to only make READ-ONLY accesses to the hard disk otherwise you may corrupt your host data (use the **−snapshot** command line option or modify the device permissions accordingly).

Windows

`CD` The preferred syntax is the drive letter (e.g. *d:*). The alternate syntax *\\.\d:* is supported. */dev/cdrom* is supported as an alias to the first CDROM drive.

Currently there is no specific code to handle removable media, so it is better to use the `change` or `eject` monitor commands to change or eject media.

`Hard disks`
Hard disks can be used with the syntax: *\\.\PhysicalDriveN* where *N* is the drive number (0 is the first hard disk).

WARNING: unless you know what you do, it is better to only make READ-ONLY accesses to the hard disk otherwise you may corrupt your host data (use the **−snapshot** command line so that the modifications are written in a temporary file).

Mac OS X

/dev/cdrom is an alias to the first CDROM.

Currently there is no specific code to handle removable media, so it is better to use the `change` or `eject` monitor commands to change or eject media.

Virtual FAT disk images

QEMU can automatically create a virtual FAT disk image from a directory tree. In order to use it, just type:

```
        qemu-system-x86_64 linux.img -hdb fat:/my_directory
```

Then you access access to all the files in the */my_directory* directory without having to copy them in a disk image or to export them via SAMBA or NFS. The default access is *read-only*.

Floppies can be emulated with the `:floppy:` option:

```
        qemu-system-x86_64 linux.img -fda fat:floppy:/my_directory
```

A read/write support is available for testing (beta stage) with the `:rw:` option:

```
        qemu-system-x86_64 linux.img -fda fat:floppy:rw:/my_directory
```

What you should *never* do:

*<use non-ASCII filenames ;>
*<use "−snapshot" together with ":rw:" ;>
*<expect it to work when loadvm'ing ;>
*<write to the FAT directory on the host system while accessing it with the guest system.>

NBD access

QEMU can access directly to block device exported using the Network Block Device protocol.

```
        qemu-system-x86_64 linux.img -hdb nbd://my_nbd_server.mydomain.org:1024/
```

If the NBD server is located on the same host, you can use an unix socket instead of an inet socket:

```
        qemu-system-x86_64 linux.img -hdb nbd+unix://?socket=/tmp/my_socket
```

In this case, the block device must be exported using qemu-nbd:

```
        qemu-nbd --socket=/tmp/my_socket my_disk.qcow2
```

The use of qemu-nbd allows sharing of a disk between several guests:

```
        qemu-nbd --socket=/tmp/my_socket --share=2 my_disk.qcow2
```

and then you can use it with two guests:

```
        qemu-system-x86_64 linux1.img -hdb nbd+unix://?socket=/tmp/my_socket
        qemu-system-x86_64 linux2.img -hdb nbd+unix://?socket=/tmp/my_socket
```

If the nbd-server uses named exports (supported since NBD 2.9.18, or with QEMU's own embedded NBD server), you must specify an export name in the URI:

```
        qemu-system-x86_64 -cdrom nbd://localhost/debian-500-ppc-netinst
        qemu-system-x86_64 -cdrom nbd://localhost/openSUSE-11.1-ppc-netinst
```

The URI syntax for NBD is supported since QEMU 1.3. An alternative syntax is also available. Here are some example of the older syntax:

```
        qemu-system-x86_64 linux.img -hdb nbd:my_nbd_server.mydomain.org:1024
        qemu-system-x86_64 linux2.img -hdb nbd:unix:/tmp/my_socket
        qemu-system-x86_64 -cdrom nbd:localhost:10809:exportname=debian-500-ppc-neti
```

Sheepdog disk images

Sheepdog is a distributed storage system for QEMU. It provides highly available block level storage volumes that can be attached to QEMU-based virtual machines.

You can create a Sheepdog disk image with the command:

```
qemu-img create sheepdog:///<image> <size>
```

where *image* is the Sheepdog image name and *size* is its size.

To import the existing *filename* to Sheepdog, you can use a convert command.

```
qemu-img convert <filename> sheepdog:///<image>
```

You can boot from the Sheepdog disk image with the command:

```
qemu-system-x86_64 sheepdog:///<image>
```

You can also create a snapshot of the Sheepdog image like qcow2.

```
qemu-img snapshot -c <tag> sheepdog:///<image>
```

where *tag* is a tag name of the newly created snapshot.

To boot from the Sheepdog snapshot, specify the tag name of the snapshot.

```
qemu-system-x86_64 sheepdog:///<image>#<tag>
```

You can create a cloned image from the existing snapshot.

```
qemu-img create -b sheepdog:///<base>#<tag> sheepdog:///<image>
```

where *base* is an image name of the source snapshot and *tag* is its tag name.

You can use an unix socket instead of an inet socket:

```
qemu-system-x86_64 sheepdog+unix:///<image>?socket=<path>
```

If the Sheepdog daemon doesn't run on the local host, you need to specify one of the Sheepdog servers to connect to.

```
qemu-img create sheepdog://<hostname>:<port>/<image> <size>
qemu-system-x86_64 sheepdog://<hostname>:<port>/<image>
```

iSCSI LUNs

iSCSI is a popular protocol used to access SCSI devices across a computer network.

There are two different ways iSCSI devices can be used by QEMU.

The first method is to mount the iSCSI LUN on the host, and make it appear as any other ordinary SCSI device on the host and then to access this device as a /dev/sd device from QEMU. How to do this differs between host OSes.

The second method involves using the iSCSI initiator that is built into QEMU. This provides a mechanism that works the same way regardless of which host OS you are running QEMU on. This section will describe this second method of using iSCSI together with QEMU.

In QEMU, iSCSI devices are described using special iSCSI URLs

```
URL syntax:
iscsi://[<username>[%<password>]@]<host>[:<port>]/<target-iqn-name>/<lun>
```

Username and password are optional and only used if your target is set up using CHAP authentication for access control. Alternatively the username and password can also be set via environment variables to have these not show up in the process list

```
export LIBISCSI_CHAP_USERNAME=<username>
export LIBISCSI_CHAP_PASSWORD=<password>
iscsi://<host>/<target-iqn-name>/<lun>
```

Various session related parameters can be set via special options, either in a configuration file provided via '-readconfig' or directly on the command line.

If the initiator-name is not specified qemu will use a default name of 'iqn.2008-11.org.linux-kvm[:<uuid>'] where <uuid> is the UUID of the virtual machine. If the UUID is not specified qemu will use ’iqn.2008−11.org.linux−kvm[:<name>’] where <name> is the name of the virtual machine.

```
Setting a specific initiator name to use when logging in to the target
-iscsi initiator-name=iqn.qemu.test:my-initiator
```

```
Controlling which type of header digest to negotiate with the target
-iscsi header-digest=CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
```

These can also be set via a configuration file

```
[iscsi]
  user = "CHAP username"
  password = "CHAP password"
  initiator-name = "iqn.qemu.test:my-initiator"
  # header digest is one of CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
  header-digest = "CRC32C"
```

Setting the target name allows different options for different targets

```
[iscsi "iqn.target.name"]
  user = "CHAP username"
  password = "CHAP password"
  initiator-name = "iqn.qemu.test:my-initiator"
  # header digest is one of CRC32C|CRC32C-NONE|NONE-CRC32C|NONE
  header-digest = "CRC32C"
```

Howto use a configuration file to set iSCSI configuration options:

```
cat >iscsi.conf <<EOF
[iscsi]
  user = "me"
  password = "my password"
  initiator-name = "iqn.qemu.test:my-initiator"
  header-digest = "CRC32C"
EOF

qemu-system-x86_64 -drive file=iscsi://127.0.0.1/iqn.qemu.test/1 \
    -readconfig iscsi.conf
```

How to set up a simple iSCSI target on loopback and access it via QEMU:

```
This example shows how to set up an iSCSI target with one CDROM and one DISK
using the Linux STGT software target. This target is available on Red Hat ba
systems as the package 'scsi-target-utils'.

tgtd --iscsi portal=127.0.0.1:3260
tgtadm --lld iscsi --op new --mode target --tid 1 -T iqn.qemu.test
tgtadm --lld iscsi --mode logicalunit --op new --tid 1 --lun 1 \
    -b /IMAGES/disk.img --device-type=disk
tgtadm --lld iscsi --mode logicalunit --op new --tid 1 --lun 2 \
    -b /IMAGES/cd.iso --device-type=cd
tgtadm --lld iscsi --op bind --mode target --tid 1 -I ALL

qemu-system-x86_64 -iscsi initiator-name=iqn.qemu.test:my-initiator \
    -boot d -drive file=iscsi://127.0.0.1/iqn.qemu.test/1 \
    -cdrom iscsi://127.0.0.1/iqn.qemu.test/2
```

GlusterFS disk images

GlusterFS is a user space distributed file system.

You can boot from the GlusterFS disk image with the command:

```
URI:
qemu-system-x86_64 -drive file=gluster[+<type>]://[<host>[:<port>]]/<volume>
                               [?socket=...][,file.debug=9][,file.logfile=..

JSON:
qemu-system-x86_64 'json:{"driver":"qcow2",
                           "file":{"driver":"gluster",
                                    "volume":"testvol","path":"a.img","debug
                                    "server":[{"type":"tcp","host":"...","po
                                              {"type":"unix","socket":"..."}
```

gluster is the protocol.

type specifies the transport type used to connect to gluster management daemon (glusterd). Valid transport types are tcp and unix. In the URI form, if a transport type isn't specified, then tcp type is assumed.

host specifies the server where the volume file specification for the given volume resides. This can be either a hostname or an ipv4 address. If transport type is unix, then *host* field should not be specified. Instead *socket* field needs to be populated with the path to unix domain socket.

port is the port number on which glusterd is listening. This is optional and if not specified, it defaults to port 24007. If the transport type is unix, then *port* should not be specified.

volume is the name of the gluster volume which contains the disk image.

path is the path to the actual disk image that resides on gluster volume.

debug is the logging level of the gluster protocol driver. Debug levels are 0–9, with 9 being the most verbose, and 0 representing no debugging output. The default level is 4. The current logging levels defined in the gluster source are 0 – None, 1 – Emergency, 2 – Alert, 3 – Critical, 4 – Error, 5 – Warning, 6 – Notice, 7 – Info, 8 – Debug, 9 – Trace

logfile is a commandline option to mention log file path which helps in logging to the specified file and also help in persisting the gfapi logs. The default is stderr.

You can create a GlusterFS disk image with the command:

```
qemu-img create gluster://<host>/<volume>/<path> <size>
```

Examples

```
qemu-system-x86_64 -drive file=gluster://1.2.3.4/testvol/a.img
qemu-system-x86_64 -drive file=gluster+tcp://1.2.3.4/testvol/a.img
qemu-system-x86_64 -drive file=gluster+tcp://1.2.3.4:24007/testvol/dir/a.img
qemu-system-x86_64 -drive file=gluster+tcp://[1:2:3:4:5:6:7:8]/testvol/dir/a
qemu-system-x86_64 -drive file=gluster+tcp://[1:2:3:4:5:6:7:8]:24007/testvol
qemu-system-x86_64 -drive file=gluster+tcp://server.domain.com:24007/testvol
qemu-system-x86_64 -drive file=gluster+unix:///testvol/dir/a.img?socket=/tmp
qemu-system-x86_64 -drive file=gluster+rdma://1.2.3.4:24007/testvol/a.img
qemu-system-x86_64 -drive file=gluster://1.2.3.4/testvol/a.img,file.debug=9,
qemu-system-x86_64 'json:{"driver":"qcow2",
                           "file":{"driver":"gluster",
                                    "volume":"testvol","path":"a.img",
                                    "debug":9,"logfile":"/var/log/qemu-glust
                                    "server":[{"type":"tcp","host":"1.2.3.4"
                                              {"type":"unix","socket":"/var/
qemu-system-x86_64 -drive driver=qcow2,file.driver=gluster,file.volume=testv
```

```
                                                    file.debug=9,file.logfile=/var/log/qe
                                                    file.server.0.type=tcp,file.server.0.
                                                    file.server.1.type=unix,file.server.1
```

Secure Shell (ssh) disk images

You can access disk images located on a remote ssh server by using the ssh protocol:

```
qemu-system-x86_64 -drive file=ssh://[<user>@]<server>[:<port>]/<path>[?host
```

Alternative syntax using properties:

```
qemu-system-x86_64 -drive file.driver=ssh[,file.user=<user>],file.host=<serv
```

ssh is the protocol.

user is the remote user. If not specified, then the local username is tried.

server specifies the remote ssh server. Any ssh server can be used, but it must implement the sftp-server protocol. Most Unix/Linux systems should work without requiring any extra configuration.

port is the port number on which sshd is listening. By default the standard ssh port (22) is used.

path is the path to the disk image.

The optional *host_key_check* parameter controls how the remote host's key is checked. The default is `yes` which means to use the local *.ssh/known_hosts* file. Setting this to `no` turns off known-hosts checking. Or you can check that the host key matches a specific fingerprint: `host_key_check=md5:78:45:8e:14:57:4f:d5:45:83:0a:0e:f3:49:82:c9:c8` (`sha1:` can also be used as a prefix, but note that OpenSSH tools only use MD5 to print fingerprints).

Currently authentication must be done using ssh-agent. Other authentication methods may be supported in future.

Note: Many ssh servers do not support an `fsync`–style operation. The ssh driver cannot guarantee that disk flush requests are obeyed, and this causes a risk of disk corruption if the remote server or network goes down during writes. The driver will print a warning when `fsync` is not supported:

warning: ssh server `ssh.example.com:22` does not support fsync

With sufficiently new versions of libssh and OpenSSH, `fsync` is supported.

NVMe disk images

NVM Express (NVMe) storage controllers can be accessed directly by a userspace driver in QEMU. This bypasses the host kernel file system and block layers while retaining QEMU block layer functionalities, such as block jobs, I/O throttling, image formats, etc. Disk I/O performance is typically higher than with `-drive file=/dev/sda` using either thread pool or linux-aio.

The controller will be exclusively used by the QEMU process once started. To be able to share storage between multiple VMs and other applications on the host, please use the file based protocols.

Before starting QEMU, bind the host NVMe controller to the host vfio-pci driver. For example:

```
# modprobe vfio-pci
# lspci -n -s 0000:06:0d.0
06:0d.0 0401: 1102:0002 (rev 08)
# echo 0000:06:0d.0 > /sys/bus/pci/devices/0000:06:0d.0/driver/unbind
# echo 1102 0002 > /sys/bus/pci/drivers/vfio-pci/new_id

# qemu-system-x86_64 -drive file=nvme://<host>:<bus>:<slot>.<func>/<namespac
```

Alternative syntax using properties:

```
qemu-system-x86_64 -drive file.driver=nvme,file.device=<host>:<bus>:<slot>.<
```

host:*bus*:*slot*.*func* is the NVMe controller's PCI device address on the host.

namespace is the NVMe namespace number, starting from 1.

Disk image file locking

By default, QEMU tries to protect image files from unexpected concurrent access, as long as it's supported by the block protocol driver and host operating system. If multiple QEMU processes (including QEMU emulators and utilities) try to open the same image with conflicting accessing modes, all but the first one will get an error.

This feature is currently supported by the file protocol on Linux with the Open File Descriptor (OFD) locking API, and can be configured to fall back to POSIX locking if the POSIX host doesn't support Linux OFD locking.

To explicitly enable image locking, specify ''locking=on'' in the file protocol driver options. If OFD locking is not possible, a warning will be printed and the POSIX locking API will be used. In this case there is a risk that the lock will get silently lost when doing hot plugging and block jobs, due to the shortcomings of the POSIX locking API.

QEMU transparently handles lock handover during shared storage migration. For shared virtual disk images between multiple VMs, the ''share-rw'' device option should be used.

By default, the guest has exclusive write access to its disk image. If the guest can safely share the disk image with other writers the `-device ...,share-rw=on` parameter can be used. This is only safe if the guest is running software, such as a cluster file system, that coordinates disk accesses to avoid corruption.

Note that share−rw=on only declares the guest's ability to share the disk. Some QEMU features, such as image file formats, require exclusive write access to the disk image and this is unaffected by the share−rw=on option.

Alternatively, locking can be fully disabled by ''locking=off'' block device option. In the command line, the option is usually in the form of ''file.locking=off'' as the protocol driver is normally placed as a ''file'' child under a format driver. For example:

```
-blockdev
driver=qcow2,file.filename=/path/to/image,file.locking=off,file.driver=file
```

To check if image locking is active, check the output of the ''lslocks'' command on host and see if there are locks held by the QEMU process on the image file. More than one byte could be locked by the QEMU instance, each byte of which reflects a particular permission that is acquired or protected by the running block driver.

SEE ALSO

The HTML documentation of QEMU for more precise information and Linux user mode emulator invocation.

AUTHOR

Fabrice Bellard and the QEMU Project developers