QEMU

Florent Gluck - Florent.Gluck@hesge.ch

March 11, 2022

What is QEMU?

- QEMU (Quick EMUlator) is a generic and open source **machine emulator and virtualizer** (`http://qemu.org`)
- **Hosted VMM**: emulates the target machine's CPU through dynamic binary translation
- **Provides various hardware and device models** enabling a variety of systems and guest OSes
- Can be used with KVM to **run VMs at near-native speed** using hardware extensions (Intel VT-x, AMD-V)
- Emulate user-level processes → allow applications compiled for one architecture to run on another

A bit of history

- QEMU project started in 2003 by geek-jedi Fabrice Bellard
  - Bellard author of FFMPEG, JSLinux and many other projects: `https://bellard.org`
- Origin of QEMU: portable Just In Time translation engine for cross architecture emulation
- QEMU quickly grew to system emulation
- QEMU started with PC hardware but now support many more: ARM, Alpha, MIPS, RISC-V, Sparc, PowerPC, SH4, etc.

Where is QEMU being used?

- Cloud computing:
  - Everything OpenStack
  - KVM and Xen guests
- Cross-compilation development environments
- Android Emulator (part of SDK) (fork)
- VirtualBox (fork)
- Almost every embedded SDK out there

What can QEMU do?

- Run i386, AMD64, ARM, Alpha, Sparc, PowerPC, s390 or MIPS OS on a i386, AMD64, Alpha, etc. computers
- Can run any i386 (or other) OS as a user application
  - Complete with graphics, sound, and network support
  - Don't need to be root!
- Tolerable performance for real world OSes
  - Orders of magnitude faster than Wind River Simics (simulator)

QEMU operating modes

3 operating modes:

- **Full-system emulation**: emulate a full computer system, including CPU & peripherals; can be used to provide virtual hosting of several virtual computers on a single computer
  - QEMU as a “System VM”
- **User-mode emulation**: run a program compiled for a different architecture (instruction set)
  - QEMU as a “Process VM”
- **Virtualization**: run KVM and Xen virtual machines with near native performance

QEMU monitor

- QEMU monitor = console for interacting with QEMU
- Various commands to:
  - control various aspects of the VM
  - inspect the running guest OS
  - change removable media and USB devices
  - take snapshots, screenshots, audio grabs
  - etc.
- Accessed:
  - CLI: via command line argument `-monitor stdio`
  - GUI: `View` → `compatmonitor0` (or similar)
  - Key shortcut: `Ctrl-Alt-2` (`Ctrl-Alt-1` switches back to guest OS)

Disk images

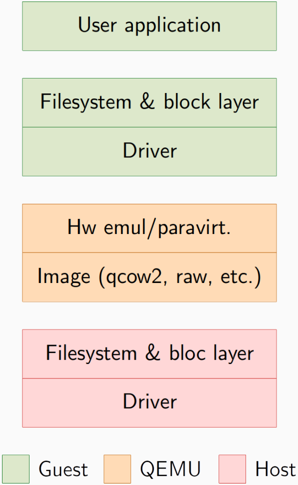
- QEMU supports many image formats:
  - qcow2, qed, vmdk, vhd, vdi, raw, rbd, nbd, tftp, ftp, vvfat, ftps, dmg, iscsi, parallels, bochs, quorum, etc.
- `qemu-img` utility to manipulate images:
  - create images
  - convert among image formats
  - resize images
  - manage disk snapshots
  - etc.

Disk images: recommendation

- Best to use either **qcow2** or **raw**:
  - **qcow2**: QEMU image format
    - most versatile and flexible
    - many features: thin provisioning, encryption, compression, snapshots, sparse files (when host filesystem permits), etc.
- **raw**: raw disk image format (default)
  - simple and very portable (exportable to other hypervisors)
  - best portability and performance, but almost no features

Storage stack

- Application and guest kernel work similar to bare metal
- Guest talks to QEMU via emulated hardware and/or paravirtualized devices
- QEMU performs I/O to an image file on behalf of the guest
- Host kernel treats guest I/O like any userspace application



User application

Filesystem & block layer

Driver

Hw emul/paravirt.

Image (qcow2, raw, etc.)

Filesystem & bloc layer

Driver

🟩 Guest 🟧 QEMU 🟥 Host

Snapshots

- QEMU supports two types of snapshots:
  - **Disk snapshots**: only saves content of the disk
  - **VM snapshots**: saves content of disk + RAM + device state
- Snapshots are stored in qcow2 image files
- Snapshots can use two backing strategies: **internal** and/or **external**
- Disk snapshots can use either internal or external backing
- VM snapshots use internal backing

Internal vs external snapshots

Internal snapshots

- All snapshots are stored inside the same qcow2 file

External snapshots

- Each snapshot is stored in a different qcow2 file
  - chain of qcow2 files
- Last qcow2 file in a chain represents the current state and is read-write
  - previous qcow2 files in a chain are read-only
- To display the chain of snapshots up to some state:

```
qemu-img info --backing-chain some_state.qcow
```

Internal disk snapshots

- Use qemu-img to manage both internal and external disk snapshots
- Internal disk snapshots are straightforward:

`qemu-img snapshot -c <name> <img>`	creates an internal disk snapshot
`qemu-img snapshot -d <name> <img>`	delete an internal disk snapshot
`qemu-img snapshot -a <name> <img>`	apply an internal disk snapshot (revert disk to saved state)
`qemu-img snapshot -l <img>`	lists all internal snapshots in the image (disk and VM)

- Deleting internal snapshots does not reduce the image file size!

External disk snapshots

- Require a base image
  - used as the backing (or base) file
  - read-only access
- Here, create an overlay image (state1.qcow) that will store the differences from the backing file base.qcow

  ```
  qemu-img create -F qcow2 -b base.qcow -f qcow2 state1.qcow
  ```

  Illustration of the above command where QEMU is ran to use state1.qcow:

  ```
      [base] <------------ [state1]
  (backing file)       (active overlay)
  ```

- At any point, a new overlay can be added to a chain of overlays

Disk image chain & merging

- Disk images in a chain can be **merged** together
  - offline using `qemu-img`
  - online using QEMU Machine Protocol (QMP) commands
- Two types of merges:
  - **commit**: merge of data from overlay files into backing files
    - committed file not removed by QEMU: must be manually removed
    - intermediate images are invalid: no more overlays can be created based on them
  - **stream**: copy of data from backing files into overlay files
    - streamed file not removed by QEMU
    - streamed file remains valid

Merging: commit operations

- Example of disk image chain ([A] = backing file, [D] = active overlay):

```
[A] <-- [B] <-- [C] <-- [D]
```

- Case 1, merge [B] into [A]:

```
[A] <-- [C] <-- [D]
```

- Case 2, merge [B] and [C] into [A]:

```
[A] <-- [D]
```

- Case 3, merge [B], [C] and [D] into [A]:

```
[A]
```

- Case 4, merge [C] into [B]:

```
[A] <-- [B] <-- [D]
```

- Case 5, merge [C] and [D] into [B]:

```
[A] <-- [B]
```

Merging: stream operations

- Example of disk image chain ([A] = backing file, [D] = active overlay):

```
[A] <-- [B] <-- [C] <-- [D]
```

- Case 1, merge everything into [D]:

```
[D]
```

- Case 2, merge [B] and [C] into [D]:

```
[A] <-- [D]
```

- Case 3, merge [B] into [C]:

```
[A] <-- [C] <-- [D]
```

Commit operations with qemu-img

- Command `qemu-img commit` can be used to perform a merge "commit"
- The combined state up to a given overlay image can be merged back into a previous image in the chain
- Example with the previous chain:

```
[A] <-- [B] <-- [C] <-- [D]
```

  - commit changes from [D] into image [A]:

```
qemu-img commit -f qcow2 -b A.qcow D.qcow
```

VM snapshots

- VM snapshots = content of disk + RAM + device state
- Managed from the QEMU monitor:

`savevm <tag>`	creates a VM snapshot
`delvm <tag>`	deletes a VM snapshot
`loadvm <tag>`	applies a VM snapshot
`info snapshots`	lists all snapshots (disk and VM)

- QEMU argument `-loadvm <tag>` starts the VM from the specified snapshot

I/O drivers

Full virtualization: IDE, SATA, SCSI, network

- Uses emulation
- Lots of trap-and-emulate → limited performance
- Good guest compatibility

Paravirtualization: Virtio

- Requires drivers in the guest OS
- Good performance

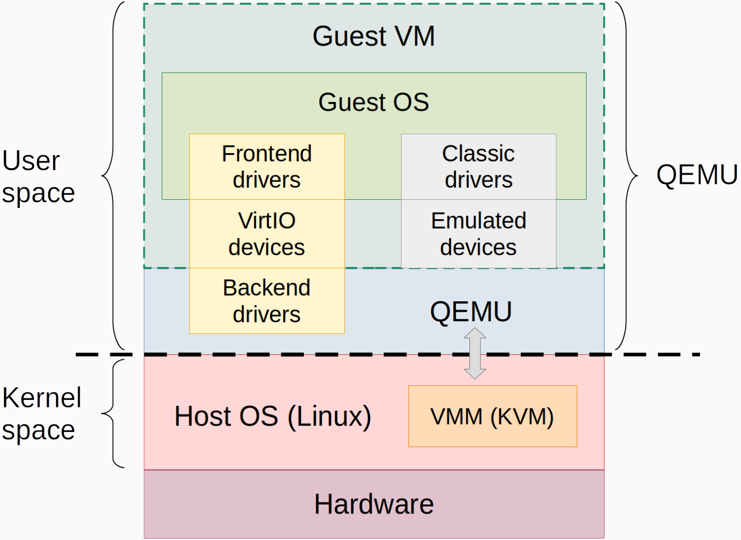
Passthrough: VFIO (requires VT-d)

- Pass hardware to guest → ∼ native performance
- Limited number of PCI Devices
- *Tricky* live migration

Virtio

- **Paravirtualized I/O** (driver) virtualization framework (standard) for Linux
- Abstraction layer over the hardware
- Common API for all paravirtualized devices
- Uses shared memory (ring buffer) between guest OS and QEMU

Virtio vs emulated drivers



Virtio architecture

Front-end driver

- Kernel module in guest OS
- Accepts I/O requests from user process
- Transfer I/O requests to back-end driver

Back-end driver

- A device in QEMU
- Accepts I/O requests from front-end driver
- Perform I/O operations via physical device

Networking

- QEMU can bridge guest network → provide direct access
- Can provide network address translation (NAT)
  - NAT address local to machine on which guest is running
  - QEMU provides address translation to guest to hide its address

QEMU guest agent

- Helper daemon installed in the guest (package `qemu-guest-agent`)
- Allow the hypervisor to perform various operations in the guest:
  - Get information
  - Read/write a file
  - Sync and freeze the filesystems
  - Shutdown/reset/suspend the guest
  - Reconfigure the vCPUs
  - etc.
- Uses QMP to exchange messages via a UNIX socket
- Supported commands: https://qemu.readthedocs.io/en/latest/interop/qemu-ga-ref.html

QEMU guest agent example (1/2)

- Guest VM must be started with these additional options:

```
-chardev socket,path=/tmp/qga.sock,server,nowait,id=
    qga0
-device virtio-serial
-device virtserialport,chardev=qga0,name=org.qemu.
    guest_agent.0
```

- In the guest, qemu-guest-agent must be started:

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

QEMU guest agent example (2/2)

- Shutdown the guest:

```
{ echo '{"execute": "guest-shutdown"}'; sleep 1; } |
    socat unix-connect:/tmp/qga.sock -
```

- Close a previously opened file on the guest (handle 1000):

```
{ echo '{"execute":"guest-file-close", "arguments":{"
    handle":1000}}'; sleep 1; } | socat unix-connect:/
    tmp/qga.sock -
```

Shared folders

- QEMU uses the 9p[1] protocol to share folders between host and guests
  - same folder can be shared by multiple guests
- **Host**: run QEMU with these additionnal arguments, where MOUNT_TAG is the share name:

```
-virtfs local,path=PATH_TO_SHARE,mount_tag=MOUNT_TAG,
    security_model=mapped
```

- **Guests**: mount the virtual filesystem, specifying the 9p type:

```
sudo mount -t 9p MOUNT_TAG MOUNT_DIR
```

  - requires 9p, 9pnet, and 9pnet_virtio kernel modules
    - mount loads them automatically
  - uses the virtio driver

[1] https://en.wikipedia.org/wiki/9P_(protocol)

Useful QEMU commands

`man qemu-system`	Exhaustive help
`-writeconfig <f>`	Write device configuration to
`-readconfig <f>`	Read device configuration from
`-machine <name>`	Set the machine to
`-smp cpus=<n>`	Set the number of CPUs to
`-m <mem>`	Set the ammount or RAM to
`-drive ...`	Define a new drive
`-device ...`	Add a device driver
`-netdev ...`	Configure user mode host network backend
`-nic ...`	Shortcut for configuring both the guest NIC and the host network backend
`-spice ...`	Enable a Spice server
`-monitor stdio`	Redirect the monitor to the console
`-vga ...`	Select the type of VGA card to emulate
`-redir tcp:x::y`	Redirect port y in the guest to x in the host
`-enable-kvm`	Use KVM to provide hardware full virtualization

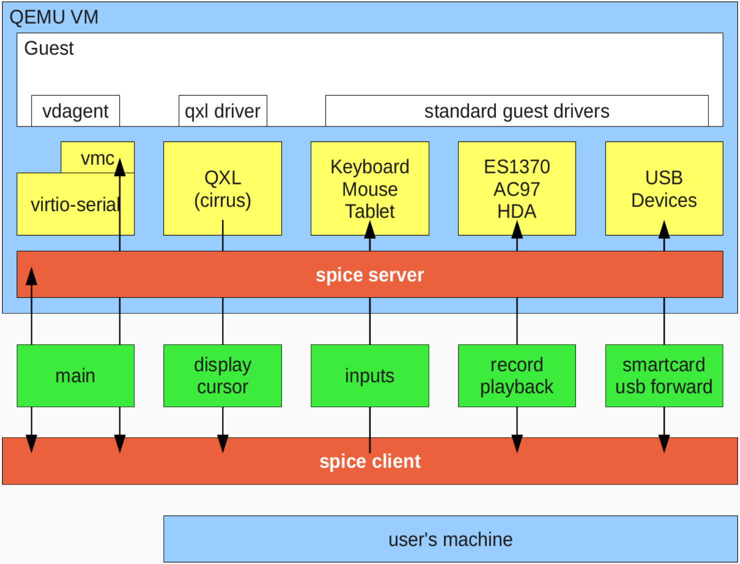
Desktop virtualization

- **Server** virtualization is commonplace and offered everywhere
  - manage virtual machines: CPU, RAM, storage, network, etc.
  - administrator access: text mode (ssh), low end graphics (VNC)
- **Desktop** virtualization **needs more**:
  - better graphics (3D, multihead, etc.)
  - sound forwarding
  - video stream support
  - USB forwarding
  - desktop integration (copy/past, shared folder, dynamic display resize, etc.)

SPICE

- **S**imple **P**rotocol for **I**ndependent **C**omputing **E**nvironments
- SPICE's goal is to provide desktop virtualizazion
- Provides virtual **desktop** infrastructure
  - SPICE network protocol
  - Virtual hardware (virtio gpu, qxl)
  - Server and client implementations

SPICE architecture



SPICE components

SPICE divided into 4 different components:

- **Client**: responsible to send data and translate the data from the VM so you can interact with it
  - Examples: `virt-viewer`, `spice-client-gtk`, `vinagre`, etc.
- **Server**: library used by the hypervisor to share the VM
  - Typically: `QEMU`
- **Guest**: software that must be running in the VM to make SPICE fully functional
  - Typically for Linux guest: virtio VGA driver, SPICE Vdagent, etc.
- **Protocol**: the network protocol

Resources

- QEMU documentation
  https://qemu.readthedocs.io/en/latest/index.html
- Live Block Device Operations
  https://qemu.readthedocs.io/en/latest/interop/live-block-operations.html
- QEMU shared folders with 9pfs
  https://wiki.qemu.org/Documentation/9psetup
- QMP documentation
  https://wiki.qemu.org/Documentation/QMP
- SPICE
  https://www.spice-space.org/
- QEMU Spice Reference
  https://people.freedesktop.org/ teuf/spice-doc/html/ch03.html