QEMU

Florent Gluck - Florent.Gluck@hesge.ch

March 3, 2023

Introduction to QEMU

What is QEMU?

- QEMU (Quick EMUlator - http://qemu.org) is an open source **machine emulator and hosted VMM**
- Feature full virtualization of the CPU through Dynamic Binary Translation
- Can use Linux KVM module (requires Intel VT-x or AMD-V) for hardware assisted virtualization of the CPU
- Emulate many hardware platforms and devices, "real" and virtual (paravirtualization)
- Emulate user-level processes $\rightarrow$ allow applications compiled for one architecture to run on another

A bit of history

- QEMU started in 2003 by geek jedi master Fabrice 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, RISC-V, MIPS, PowerPC, Alpha, Sparc, SH4, etc.

Where is QEMU being used?

- Cloud computing:
  - everything OpenStack
  - along 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[1], AMD64[2], 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!
- Descent emulation performance for real world OSes
  - orders of magnitude faster than Wind River Simics (simulator)

[1] i386 = x86 = IA-32 = Intel or AMD 32-bits architecture
[2] AMD64 = Intel or AMD 64-bits architecture

QEMU usage

- VM hardware is specified on the command line (by opposition to VirtualBox or VMWare!)
- Binary for AMD64 is `qemu-system-x86_64` (Debian package `qemu-system-x86`)
- Use man `qemu-system` for the manual
- Typical use:
  1. create an image disk with `qemu-img` (once)
  2. run `qemu-system-x86_64` to configure the VM with the previous disk and run it

Typical QEMU options

- Use hardware assisted virtualization via KVM

```
-enable-kvm
```

- Same CPU architecture as host, but 2 cores (vCPU):

```
-cpu host -smp cpus=2
```

- 4GB of RAM

```
-m 4096
```

- Paravirtualized graphics card

```
-vga virtio
```

- Emulated Intel 82574L GbE network card

```
-nic user,model=e1000e
```

- Paravirtualized disk controller using `disk.qcow` as hard disk

```
-drive file=disk.qcow,index=0,media=disk,format=qcow2,if=virtio
```

Can a host run KVM?

- Check for hardware virtualization support (Intel or AMD):

```
$ lscpu|grep Flags|grep "vmx\|svm"
Flags:  fpu vme de pse tsc msr pae mce cx8 apic sep
    mtrr pge mca cmov pat pse36 clflush dts acpi mmx
    fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
     lm constant_tsc arch_perfmon pebs bts rep_good nopl
     xtopology nonstop_tsc cpuid aperfmperf pni
    pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2
    ssse3 sdbg fma cx16 xtpr pdcm ...
```

- Check the `kvm` module is loaded in the kernel:

```
$ lsmod|grep kvm
kvm_intel              282624  0
kvm                    663552  1 kvm_intel
```

Using KVM

- To access the KVM device, `/dev/kvm`, one **must** either:
  - be in the `kvm` group
    - to add user `pipo` to the `kvm` group:

```
sudo usermod -a -G kvm pipo
```

  - have the proper ACL permissions (`getfacl /dev/kvm`)
- Typical examples of KVM API use:
  - QEMU when launched with the `-enable-kvm` option
  - Any other Linux-based hypervisor using KVM
  - Any application using `/dev/kvm`, typically a custom VMM

QEMU nested virtualization

- Pass the following argument to tell QEMU to expose a CPU with all supported host features, notably hardware virtualization instructions

```
-cpu host
```

- Provide an arbitrary level of nested virtualization!

Devices in QEMU

QEMU devices

- QEMU supports a very large number of devices, including CPU architectures:
  - emulated devices (“real” devices)
  - paravirtualized devices (mostly virtio devices)
- To list all supported devices:

```
qemu-system-x86_64 -device help
```

- To list supported options for a specific device:

```
qemu-system-x86_64 -device rtl8139,help
qemu-system-x86_64 -device virtio-net-pci,help
```

Device types

Emulated: IDE, SATA, SCSI disk controlers, network cards, etc.

- Good compatibility (drivers usually present in the guest OS)
- Low performance

Paravirtualized: virtio devices

- Good performance
- Require dedicated paravirtualized drivers in the guest OS

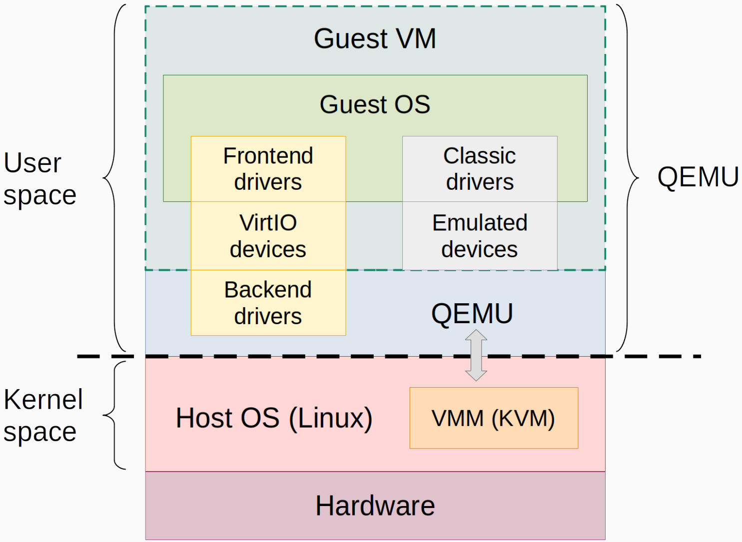
Passthrough: via VFIO

- Near native performance
- Limited number of PCI devices supported
- *Tricky* live migration
- Requires VT-d hardware extension

Virtio framework

- Specification for **paravirtualized** device (I/O) virtualization
- Abstraction layer over the hardware (devices)
- virtio provides:
  - classes of virtual devices (network, block, memory, etc.)
  - common I/O registers
  - virtual queues (shared memory)
  - device probing and configuration
- Common API for all paravirtualized devices
- Use 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

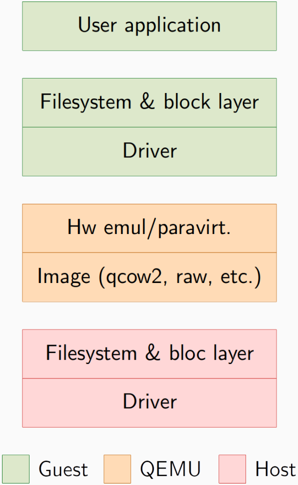
Networking

- QEMU can bridge guest network $\rightarrow$ 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

Storage in QEMU

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



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.
- Use `qemu-img` tool 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** formats
- **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
  - simple and very portable (exportable to other hypervisors)
  - best portability and performance, but few features

Disk image growth

- Over time, a VM disk image can grow larger than the actual data stored within it
- Guest OS typically only marks a deleted file as zero
  - blocks are not actually deleted for performance reasons
  - qcow2 file cannot differentiate between allocated and used, and allocated but not used
- Solution to avoid disk image from growing more than necessary:
  - on the host, add these options to QEMU's `-drive` argument:

```
discard=unmap,detect-zeroes=unmap
```

  - in the guest OS, reclaim blocks from the filesystem:

```
sudo fstrim -av
```

  - even better: automate the process with a service:

```
systemctl enable fstrim.timer && systemctl start
    fstrim.timer
```

Inspecting/modifying VM disk image files (1/2)

guestfish

- Shell and command-line tool for examining and modifying VM filesystems

guestmount

- Mount a guest filesystem on the host using FUSE and `libguestfs`

virt-filesystems

- List filesystems and partitions found in a disk image file
- Typical use:

```
virt-filesystems -a image_file -lh
```

Inspecting/modifying VM disk image files (2/2)

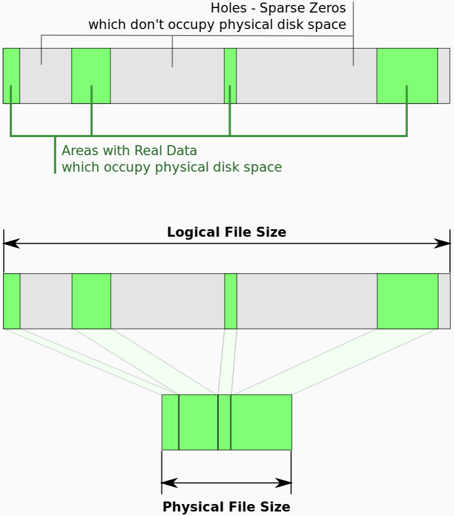
- The tools from the previous slide use `libguestfs` from the `guestfs-tools` Debian package
- **These tools must not be run as root!**
- However, on Ubuntu `/boot/vmlinuz*` has the wrong permissions[3], thus they must be changed with:

```
sudo chmod 0644 /boot/vmlinuz*
```

[3] https://libguestfs.org/guestfs-faq.1.html

Sparse files

- A sparse file is a file that does not store "empty" (unused) space
- Data blocks containing no data are not actually written to disk
- Most modern file systems support sparse files
  - ext4, xfs, ntfs, btrfs do
  - FAT filesystems do not



Handling sparse files

- To create a 10MB sparse file:

```
truncate -s 10M myfile
dd if=/dev/zero of=myfile bs=10M count=1 conv=sparse
```

- To display a file's real allocated space:

```
ls -ls file
du file
```

- To convert a file into a sparse file:

```
fallocate -v -d file
```

- To convert a sparse file into a non-sparse file:

```
cp file nonsparse_file --sparse=never
```

Copying sparse files

- Linux transparently handles copy of sparse files
  - if unsure, use:

```
cp --sparse=always
```

- **Transfering sparse files over the network is usually not supported**
  - files lose their sparse property
  - server and client must both implement support for sparse files
  - `scp` does not support sparse files!
  - use `rsync` over `scp` instead (using ssh key pairs)

```
rsync -P --sparse source_file  destination_machine:
```

Snapshots in QEMU

QEMU snapshots

- QEMU supports two types of snapshots:
  - **disk snapshots**: only save content of the disk
  - **VM snapshots**: save 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/written by QEMU
  - previous qcow2 files in a chain are **only read** by QEMU
- 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>	create 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>	list 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 by QEMU
- 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[4]
- 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

[4] https://qemu.readthedocs.io/en/latest/interop/qemu-qmp-ref.html

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

Tools for QEMU

QEMU monitor

- QEMU monitor = console for interacting with QEMU to:
  - control various aspects of the VM
  - inspect the running guest OS
  - change removable media and USB devices
  - take snapshots, screenshots, audio grabs
  - etc.
- Monitor access:
  - telnet server running in QEMU:
    `-monitor telnet::1234,server,nowait`
    - client side: `telnet ip_server 1234`
  - GUI: `View` → `compatmonitor0` (or similar)
  - in the shell QEMU is running in, with: `-monitor stdio`
  - [Ctrl-Alt-2] ([Ctrl-Alt-1] switches back to guest OS)

QEMU Guest Agent (QGA)

- Service installed in the guest OS
- Allow QEMU to perform many operations:
  - Get guest OS information
  - Read/write a file in the guest
  - Sync and freeze the filesystems
  - Shutdown/reset/suspend the guest
  - etc.
- Uses QEMU Guest Agent Protocol to exchange messages via a UNIX socket
- Supported commands here[5]

[5] https://qemu.readthedocs.io/en/latest/interop/qemu-ga-ref.html

QEMU guest agent example (1/2)

- VM must be started with these additional arguments:

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

- In the guest OS, `qemu-guest-agent` must be installed and started (usually already the case):

```
sudo apt-get install qemu-guest-agent
sudo systemctl enable qemu-guest-agent
sudo 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 directories

- QEMU uses the 9p[6] protocol to share dirs between host and guests
  - same dir 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 virtio

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

Useful QEMU commands and arguments

`man qemu-system`	Exhaustive help on QEMU
`-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

Useful tools

Debian package that provides many VM tools: `guestfs-tools`

`virt-rescue`	run a rescue shell on a VM
`virt-builder`	build VM images quickly
`virt-copy-out`	copy files and dirs out of a VM disk image
`virt-copy-in`	copy files and dirs into a VM disk image
`virt-resize`	resize a VM disk
`virt-sparsify`	make a VM disk sparse
`virt-edit`	edit a file in a VM
`virt-ls`	list files in a VM
`virt-filesystems`	list filesystems, partitions, block devices, LVM in a VM
...	

Virtual Desktop Infrastructure (VDI)

Desktop virtualization

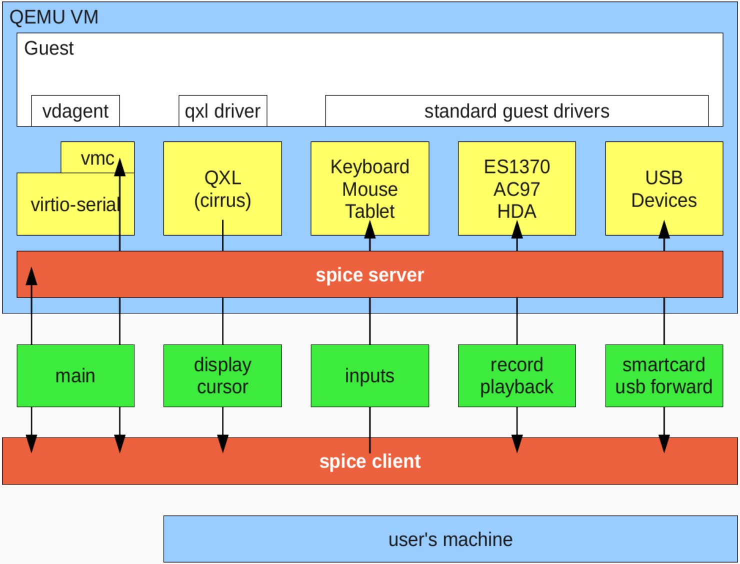
- **Server** virtualization is commonplace and offered everywhere
    - manage virtual machines: CPU, RAM, storage, network, etc.
    - administrator access: text mode (ssh), low end graphics (VNC)
- **Desktop** virtualization **needs more**:
    - better graphics (3D, multihead, etc.)
    - sound forwarding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

SPICE

- **S**imple **P**rotocol for **I**ndependent **C**omputing **E**nvironments[7]
- SPICE's goal is to provide desktop virtualizazion
- Provides virtual **desktop** infrastructure
  - SPICE network protocol
  - virtual hardware (virtio gpu, qxl)
  - server and client implementations
- VM remotely accessed via a dedicated port on the host (one port per VM)

[7] https://www.spice-space.org/

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: `remote-viewer`, `spicy`, 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

SPICE features

- More bandwidth efficient than VNC
- Multiple channels: main, display, cursor, inputs, record, playback
  - any combination of channels can be encrypted via TLS
- Access can be password protected (or not)
- Copy/paste host $\leftrightarrow$ guest OS
- USB redirection over the network
- Shared directory over the network
- Image compression
- OpenGL acceleration

SPICE basic usage

Server side

- Require either `-vga virtio` (prefered) or `-vga qxl` paravirtualized graphics driver
- Arguments to start a SPICE server on port 8000 in the VM (without authentication):

```
-device virtio-serial-pci
-spice port=8000,disable-ticketing=on
-device virtserialport,chardev=spicechannel0,name=com.redhat.spice.0
-chardev spicevmc,id=spicechannel0,name=vdagent
```

Client side

- Require a spice client, for instance `remote-viewer` (part of `virt-viewer` Debian package):

```
remote-viewer "spice://server_ip?port=8000"
```

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
- Using QEMU Machine Protocol (QMP)
  https://wiki.qemu.org/Documentation/QMP
- Introduction to VirtIO
  https://blogs.oracle.com/linux/post/introduction-to-virtio
- Virtio Driver Implementation
  http://www.dumais.io/index.php?article=aca38a9a2b065b24dfa1dee728062a12