

Systems virtualization

Lab 3 : QEMU introduction



EL HAKOUNI Yassin
17/03/2025

1. Introduction.....	2
1.1 Context.....	2
1.2 Objectives.....	2
2. Experimental Environment.....	2
3. Hands-on with QEMU.....	2
3.1 Verifying KVM Support.....	2
3.2 Downloading and Verifying the Ubuntu ISO.....	3
3.3 Creating Disk Images (qcow2 and raw).....	3
3.4 Benchmark: Booting Ubuntu Live CD (No KVM vs. KVM).....	3
3.5 Creating and Using the qcow2 Disk for VM Installation.....	4
3.6 Checking Disk Usage in the Guest.....	5
3.7 Upgrading to VirtIO Paravirtualization.....	5
3.8 QEMU Guest Agent.....	6
4. ARM Emulation with QEMU: Raspberry Pi Simulation.....	8
5. Automating VM Deployment by using a script.....	9
5.1 Script Objectives and Usage.....	9
5.2 Boot Priority Logic.....	9
5.3 QEMU Options Summary.....	10
5.4 Example Execution.....	10
6. Conclusion.....	10

1. Introduction

1.1 Context

Virtualization is a core technology in modern IT infrastructures, enabling the isolation, deployment, and management of multiple operating systems on a single physical machine. For this lab, I focused on QEMU, an open-source machine emulator and virtualizer, paired with KVM for hardware-assisted virtualization on x86_64 platforms.

1.2 Objectives

The main objectives of this lab were:

- To understand and compare hardware virtualization (KVM) versus full emulation.
- To manipulate and benchmark various disk image formats.
- To automate VM deployment via a custom Bash script.
- To use paravirtualized devices to optimize VM performance.
- To experiment with QEMU Guest Agent for host-guest interaction.

2. Experimental Environment

All tasks were conducted on my physical machine running Pop!_OS (GNU/Linux), equipped with an Intel Core i7 CPU that supports virtualization (VT-x/KVM). I performed every operation through the command line to gain a deeper understanding of each step.

3. Hands-on with QEMU

3.1 Verifying KVM Support

To begin, I needed to ensure that my hardware and OS were configured for virtualization.

- I checked if my CPU supports hardware virtualization with the following command:

```
popito@pop-os:~$ egrep -c '(vmx|svm)' /proc/cpuinfo
16
```

Here, the result "16" confirms that my CPU supports VT-x (KVM).

- Next, I verified that the KVM kernel modules were properly loaded:

```
popito@pop-os:~$ lsmod | grep kvm
kvm_intel          483328    4
kvm                1421312    1 kvm_intel
popito@pop-os:~$
```

The presence of both `kvm_intel` and `kvm` modules confirms that KVM is enabled on my system.

3.2 Downloading and Verifying the Lubuntu ISO

For all my experiments, I used the official Lubuntu Desktop 24.04 x86_64 ISO image.

- After downloading the ISO, I verified its integrity using SHA256 checksum to ensure that my installation media was not corrupted:

```
popito@pop-os:~$ sha256sum lubuntu-24.04.2-desktop-amd64.iso
d29e07f791eec68f76521bd8fecb2fb15507c9aab9f8197979cbc70e117dacb1  lubuntu-24.04.2-desktop-amd64.iso
```

3.3 Creating Disk Images (qcow2 and raw)

Before starting the VM, I prepared two types of disk images to explore their differences: `qcow2` (QEMU Copy-On-Write) and `raw`.

- I created the images as follows:

```
popito@pop-os:~$ qemu-img create -f qcow2 disk.qcow2 100G
Formatting 'disk.qcow2', fmt=qcow2 cluster_size=65536 extended_l2=off compressio
n_type=zlib size=107374182400 lazy_refcounts=off refcount_bits=16
popito@pop-os:~$ qemu-img create -f raw disk.raw 100G
Formatting 'disk.raw', fmt=raw size=107374182400
popito@pop-os:~$
```

- I then listed their sizes to compare the *apparent* and *real* disk usage:

```
popito@pop-os:~$ ls -lh | grep disk
-rw-r--r-- 1 popito popito 194K mai 30 03:56 disk.qcow2
-rw-r--r-- 1 popito popito 100G mai 30 03:56 disk.raw
popito@pop-os:~$ ls -ls | grep disk
196 -rw-r--r-- 1 popito popito      198208 mai 30 03:56 disk.qcow2
  4 -rw-r--r-- 1 popito popito 107374182400 mai 30 03:56 disk.raw
popito@pop-os:~$
```

As seen above, the `disk.raw` image immediately takes up 100GB of space, while the `disk.qcow2` is very lightweight at creation.

3.4 Benchmark: Booting Lubuntu Live CD (No KVM vs. KVM)

To compare the performance of full emulation versus hardware-assisted virtualization, I measured the time taken to boot into the Lubuntu desktop.

Without KVM :

```
popito@pop-os:~$ time qemu-system-x86_64 \
-cdrom lubuntu-24.04.2-desktop-amd64.iso \
-m 4096 \
-smp cpus=2
^Cqemu-system-x86_64: terminating on signal 2

real    12m28.520s
user    10m33.701s
sys      1m41.189s
```

With KVM :

```
popito@pop-os:~$ time qemu-system-x86_64 \
-cdrom lubuntu-24.04.2-desktop-amd64.iso \
-m 4096 \
-smp cpus=2 \
-machine accel=kvm
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000
001H:ECX.svm [bit 2]
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000
001H:ECX.svm [bit 2]
^Cqemu-system-x86_64: terminating on signal 2

real    0m33.829s
user    0m49.254s
sys      0m3.838s
popito@pop-os:~$
```

The screenshots above clearly show that enabling KVM drastically reduces the boot time compared to pure emulation (more than 10X faster).

3.5 Creating and Using the qcow2 Disk for VM Installation

After benchmarking the live CD boot, I proceeded to install Ubuntu onto a dedicated qcow2 disk image.

- I launched the installation with the following command, specifying the **qcow2** image as the main disk :

```
popito@pop-os:~$ time qemu-system-x86_64 \
-drive file=lubuntu.qcow,format=qcow2 \
-m 4096 \
-smp cpus=2 \
-machine accel=kvm
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000001H:ECX.svm [bi
t 2]
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000001H:ECX.svm [bi
t 2]
^Cqemu-system-x86_64: terminating on signal 2

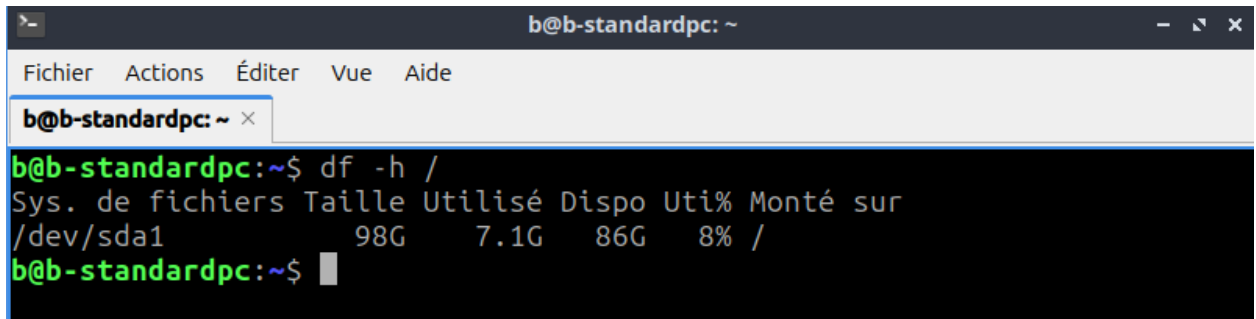
real    0m22.640s
user    0m24.796s
sys      0m5.980s
popito@pop-os:~$
```

I measured the time taken to boot the VM using the **qcow2** disk with KVM enabled. Here the boot process was about 10 seconds faster than when booting from the CD-ROM.

3.6 Checking Disk Usage in the Guest

Once the VM was up and running, I wanted to verify how the guest operating system saw the disk.

- Inside the Lubuntu VM, I used the **df -h /** command to display the size and usage of the root filesystem:



```

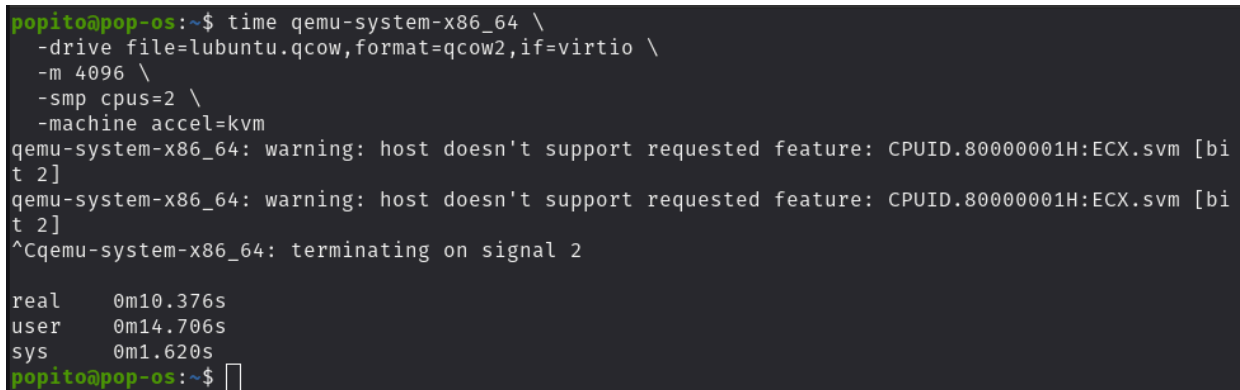
b@b-standardpc: ~
Fichier Actions Éditer Vue Aide
b@b-standardpc: ~ x
b@b-standardpc:~$ df -h /
Sys. de fichiers Taille Utilisé Dispo Uti% Monté sur
/dev/sda1          98G   7.1G   86G   8% /
b@b-standardpc:~$
  
```

This confirmed that the virtual disk (**/dev/sda1**) had a total size of 98GB, matching the size of the disk image I created.

3.7 Upgrading to VirtIO Paravirtualization

After validating that my Lubuntu guest was using a standard emulated disk controller (**/dev/sda1**), I decided to switch to VirtIO for better disk performance.

First, I relaunched the VM using VirtIO as the disk interface, by adding **if=virtio** to the QEMU command :



```

popito@pop-os:~$ time qemu-system-x86_64 \
-drive file=lubuntu.qcow,format=qcow2,if=virtio \
-m 4096 \
-smp cpus=2 \
-machine accel=kvm
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000001H:ECX.svm [bit 2]
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000001H:ECX.svm [bit 2]
^Cqemu-system-x86_64: terminating on signal 2

real    0m10.376s
user    0m14.706s
sys     0m1.620s
popito@pop-os:~$
  
```

The measured boot time dropped significantly, showing the performance gain from paravirtualization. We went from 22 seconds as seen in section 3.5, to 10 seconds here (2X faster).

Inside the guest, I immediately ran:


```
b@b-standardpc:~$ df -h /
Sys. de fichiers Taille Utilisé Dispo Uti% Monté sur
/dev/vda1          98G   7.1G   86G    8% /
b@b-standardpc:~$
```

This time, the root disk was `/dev/vda1` instead of `/dev/sda1`, indicating that the guest now detects a VirtIO disk.

I then verified the kernel log for VirtIO devices :

```
b@b-standardpc:~$ journalctl -ka | grep virtio
May 29 16:36:28 b-standardpc kernel: virtio_blk virtio0: 2/0/0 default
/read/poll queues
May 29 16:36:28 b-standardpc kernel: virtio_blk virtio0: [vda] 2097152
00 512-byte logical blocks (107 GB/100 GiB)
```

I found that the kernel had detected `virtio_blk` (the VirtIO block device driver). To further confirm, I listed PCI devices and checked loaded kernel modules :

```
b@b-standardpc:~$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (r
ev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton
II]
00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Tri
ton II]
00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)
00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Etherne
t Controller (rev 03)
00:04.0 SCSI storage controller: Red Hat, Inc. Virtio block device
b@b-standardpc:~$ lsmod | grep virtio
b@b-standardpc:~$
```

With `lspci`, I could clearly see an entry for the "Red Hat, Inc. Virtio block device," which proved that the paravirtualized disk controller was detected by the operating system.

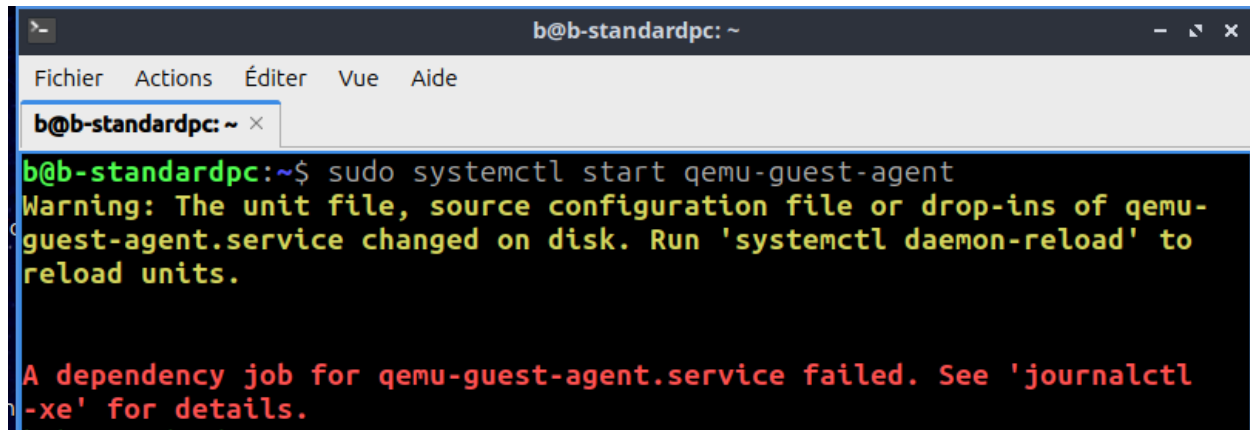
However, when I tried to list the loaded VirtIO kernel modules using `lsmod | grep virtio`, nothing was displayed. This is expected in some modern Linux distributions because the VirtIO drivers (such as `virtio_blk`) are often built directly into the kernel rather than loaded as modules. So, even if the command returns nothing, the system is still using VirtIO, as confirmed by both the kernel log and the PCI device listing.

3.8 QEMU Guest Agent

After optimizing disk I/O with VirtIO, I focused on improving host-guest integration by setting up the QEMU Guest Agent (QGA). The QEMU Guest Agent allows the host to communicate with the

guest OS, enabling advanced features like live file operations, precise shutdown, and user enumeration.

To enable the QEMU Guest Agent, I started by launching the service inside my Ubuntu guest. I tried to start the agent with systemd using the command. However, as shown in the screenshot below, I encountered a dependency error and the service failed to start :



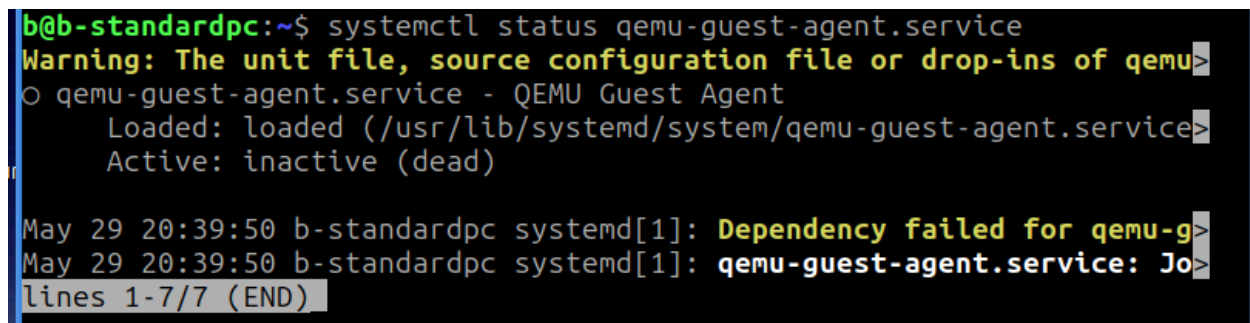
```

b@b-standardpc: ~
Fichier Actions Éditer Vue Aide
b@b-standardpc: ~ x
b@b-standardpc:~$ sudo systemctl start qemu-guest-agent
Warning: The unit file, source configuration file or drop-ins of qemu-guest-agent.service changed on disk. Run 'systemctl daemon-reload' to reload units.

A dependency job for qemu-guest-agent.service failed. See 'journalctl -xe' for details.

```

To understand what was happening, I checked the status of the service :



```

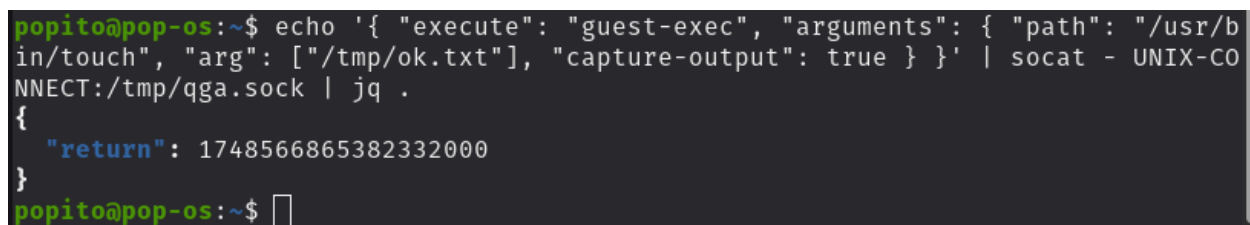
b@b-standardpc:~$ systemctl status qemu-guest-agent.service
Warning: The unit file, source configuration file or drop-ins of qemu-guest-agent.service changed on disk. Run 'systemctl daemon-reload' to reload units.
○ qemu-guest-agent.service - QEMU Guest Agent
   Loaded: loaded (/usr/lib/systemd/system/qemu-guest-agent.service; vendor preset: enabled)
   Active: inactive (dead)

May 29 20:39:50 b-standardpc systemd[1]: Dependency failed for qemu-guest-agent.service: QEMU Guest Agent.
May 29 20:39:50 b-standardpc systemd[1]: qemu-guest-agent.service: Job for qemu-guest-agent.service failed because the control process exited with error code.
lines 1-7/7 (END)

```

This confirmed that a dependency job for **qemu-guest-agent.service** had failed. To solve this, I shut down the VM and then relaunched it from the host, making sure to add the correct QEMU options to enable the Guest Agent socket. For example, I specified the **-chardev** and **-device** arguments.

After relaunching the VM with the right parameters, the Guest Agent worked as expected. On the host, I was able to communicate with the guest via the QEMU Guest Agent socket (**/tmp/qga.sock**). For example, I sent a command to create a file in the guest system :



```

popito@pop-os:~$ echo '{ "execute": "guest-exec", "arguments": { "path": "/usr/bin/touch", "arg": ["/tmp/ok.txt"], "capture-output": true } }' | socat - UNIX-CONNECT:/tmp/qga.sock | jq .
{
  "return": 1748566865382332000
}
popito@pop-os:~$

```


I then switched to the guest VM (through SSH) and checked for the presence of the `/tmp/ok.txt` file, confirming that the Guest Agent was fully operational :

```
b@b-standardpc:~$ ls /tmp/ok.txt
/tmp/ok.txt
b@b-standardpc:~$
```

This demonstrated that once the VM is launched with the correct configuration, the QEMU Guest Agent works perfectly and enables direct file operations and other advanced interactions from the host to the guest.

4. ARM Emulation with QEMU: Raspberry Pi Simulation

After working extensively with x86_64 virtual machines using KVM and QEMU, I wanted to explore QEMU's ability to emulate a completely different CPU architecture. I chose to emulate an ARM environment, more specifically a setup compatible with the Raspberry Pi. This part of the lab allowed me to better understand how QEMU handles full system emulation without relying on hardware acceleration like KVM.

To perform this test, I used an image of Raspbian Stretch Lite, the official lightweight Debian-based OS for Raspberry Pi devices. Since KVM cannot be used with non-x86 architectures on an x86 host, this setup relied entirely on software emulation — a much slower, but more flexible process.

After launching the ARM virtual machine with the proper QEMU arguments, I logged into the guest and ran the `lscpu` command to inspect the virtual CPU. The output below confirmed that the system was successfully running an ARMv6-compatible processor, in Little Endian mode, with only one core as I specified it :

```
pi@raspberrypi:~$ lscpu
Architecture:        armv6l
Byte Order:          Little Endian
CPU(s):              1
On-line CPU(s) list: 0
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):           1
Model:               7
Model name:          ARMv6-compatible processor rev 7 (v6l)
BogoMIPS:            915.86
Flags:               half thumb fastmult vfp edsp java tls
pi@raspberrypi:~$
```

This output matched the expected characteristics of an emulated Raspberry Pi 1, which also uses an ARMv6 processor. Additionally, the `BogoMIPS` value of ~915 indicates the performance level achieved under full emulation. While not comparable to KVM-based x86 guests in terms of speed,

the fact that a full OS could run under emulation demonstrates QEMU's versatility.

This exercise also helped highlight the difference between system emulation (used here) and hardware-assisted virtualization. Unlike the previous x86_64 guests that benefited from the host CPU's virtualization extensions, this ARM guest was entirely emulated, meaning every instruction executed was translated by QEMU, greatly increasing CPU overhead.

Nonetheless, the guest remained stable and usable for lightweight operations. This validates the use of QEMU for embedded development and cross-platform testing, especially in scenarios where real ARM hardware is unavailable.

5. Automating VM Deployment by using a script

After manually launching and configuring multiple QEMU virtual machines throughout this lab, I decided to streamline the process by writing a Bash script called `vm_run.sh`. The goal was to simplify and automate the most common VM launch configurations, while retaining flexibility via command-line arguments.

5.1 Script Objectives and Usage

The script is designed to launch a virtual machine with the following default configuration:

- 2 virtual CPUs
- 4 GB of RAM
- KVM acceleration (if supported)
- VirtIO paravirtualized disk and network
- Optional ISO boot support
- QEMU Guest Agent socket
- Port forwarding from host 4000 to guest 22 (for SSH)

The script takes the following arguments:

```
./vm_run.sh -d <disk_image> -i <iso_image>
```

- `-d` specifies the disk image (qcow2 format)
- `-i` specifies the ISO image (optional; when provided, the system boots from CD-ROM)

At least one of the two must be specified; otherwise, the script exits with an error.

5.2 Boot Priority Logic

I implemented logic in the script to prioritize booting from the ISO when both `-d` and `-i` are

provided. This allows me to launch a live CD for installation when needed, and to boot from disk otherwise. This behavior was controlled via the `-boot d` flag in QEMU.

5.3 QEMU Options Summary

Here's a breakdown of the key QEMU options used in the script:

- `-m 4096`: Allocates 4GB of memory
- `-smp cpus=2`: Sets the number of virtual CPUs to 2
- `-enable-kvm`: Enables KVM acceleration (if supported by the host)
- `-drive file=... ,if=virtio`: Mounts the main disk image using VirtIO
- `-cdrom ...`: Mounts the ISO image (if specified)
- `-netdev user,hostfwd=tcp::4000-:22`: Forwards port 4000 on the host to port 22 in the guest
- `-device virtio-net-pci`: Adds a VirtIO network interface
- `-device virtio-blk-pci`: Adds a VirtIO disk controller
- `-chardev socket,path=/tmp/qga.sock,server,nowait,id=qga`: Enables the QEMU Guest Agent socket
- `-device virtio-serial`: Adds the serial device for QGA

5.4 Example Execution

To test the script, I executed:

```
./vm_run.sh -d lubuntu.qcow2 -i lubuntu-24.04.2-desktop-amd64.iso
```

This launched the VM, successfully booted into the ISO, and allowed me to proceed with installation. Once the ISO was removed, the same script allowed me to reboot directly into the installed Ubuntu system from the qcow2 disk.

This scripting approach significantly improved efficiency and reduced manual errors. It also allowed me to maintain consistent configurations across multiple tests.

6. Conclusion

This lab provided a comprehensive and hands-on experience with QEMU, demonstrating its versatility and performance optimization capabilities. Through systematic experimentation, I gained practical insights into :

-
- the difference between full software emulation and hardware-assisted virtualization using KVM,
 - the structure and trade-offs between disk image formats like **raw** and **qcow2**,
 - performance improvements enabled by paravirtualized devices such as VirtIO,
 - and the use of the QEMU Guest Agent to establish seamless host-to-guest communication.

Working exclusively from the command line forced me to understand each QEMU option in depth. I relied on system tools like **df**, **lspci** and log analysis to validate each configuration change and its effect.

The ARM emulation experience confirmed QEMU's capability to support multiple architectures. This is particularly relevant in contexts like cross-platform development and embedded systems testing.

Lastly, creating a reusable Bash script gave me a clearer understanding of how to automate VM deployment and configure advanced options (e.g. port forwarding, ISO mounting, guest agent socket) in a reliable, repeatable way.