Lab 0. QEMU Simulator and Raspberry PI Setup

We setup a simulation environment with QEMU to run Raspberry PI code on your PC. The following instructions are for Windows 11.

Step 1. Set up QEMU and KVM

Follow the instructions in this video to set up QEMU and KVM (skip the last step of installing Ubuntu Linux)

QEMU Installation Guide for Windows PC [with KVM]
https://www.youtube.com/watch?v=dPg8P5DYZNg

First, from Search box, start "Turn Windows features on or off", and enable these three options: HyperV, Windows Subsystem for Linux, Virtual Machine Platform.

From Windows Powershell, run:

```
wsl --install
wsl --set-default-version 2
```

From WSL, run:

```
sudo apt update && sudo apt upgrade -y
sudo apt install qemu-kvm libvirt-daemon-system libvirt-clients bridge-utils -y
sudo usermod -aG kvm $USER
```

From Windows Powershell, run:

```
wsl --shutdown
```

Skip the rest of the steps for installing Ubuntu, since we will run Raspberry PI OS instead.

When starting WSL, always choose "run as administrator".

For Mac users: please refer to the following links:

- Install QEMU on OSX https://gist.github.com/Jatapiaro/6a7c769a07911adc629e1604729d4c7a
- How To Install And Use QEMU Virtual Machine On macOS With HVF Acceleration - Command Line Tutorial https://www.youtube.com/watch?v=EZ56AdVO2Mc
- How to install Linux on Mac step by step https://macpaw.com/how-to/install-linux-on-mac

If you run into difficulties on Windows or Mac, I can give you a Raspberry PI board. Note that SSH to Linux machines in the lab does not work, since you need root privileges to install QEMU on the machine.

Step 2. Install Raspberry PI on QEMU

This step is based on this page for installing Raspberry PI on QEMU:

Emulating a Raspberry Pi in QEMU

https://interrupt.memfault.com/blog/emulating-raspberry-pi-in-qemu

However, we need more storage in the emulated Raspberry PI, so please follow the updated instructions below.

1. Get the Raspberry Pi Image (same as original)

```
sudo apt-get install -y qemu-system-aarch64
cd ~
wget https://downloads.raspberrypi.org/raspios_arm64/images/raspios_arm64-2023-05-03/2023-05-03-raspios-bullseye-arm64.img.xz
xz -d 2023-05-03-raspios-bullseye-arm64.img.xz
```

2. Inspect and Mount the Boot Partition (same as original)

```
fdisk -l ./2023-05-03-raspios-bullseye-arm64.img
# Note the boot partition offset (should be 8192 * 512 = 4194304)
sudo mkdir /mnt/image
sudo mount -o loop,offset=4194304 ./2023-05-03-raspios-bullseye-arm64.img /mnt/image/
```

Extract kernel and device tree as before:

```
cp /mnt/image/bcm2710-rpi-3-b-plus.dtb ~
cp /mnt/image/kernel8.img ~
```

3. Resize the Image to 32 GB (or desired size)

Instead of resizing to the next power of two, pick a larger number directly. For example, **32 GB**:

```
qemu-img resize ./2023-05-03-raspios-bullseye-arm64.img 32G
```

You can change 32G to 16G, 64G, etc.

4. Enable SSH (same as original)

Create hashed password and add userconf + ssh. In the echo command, replace <your_hash_here> with what you got from running “openssl passwd -6”, and save to userconf.txt instead of userconf. (Copy this command into a text file and edit it.)

```
openssl passwd -6
# type your password twice; copy the hash
echo 'pi:<your_hash_here>' | sudo tee /mnt/image/userconf
sudo touch /mnt/image/ssh
```

Unmount:

```
sudo umount /mnt/image
```

5. Launch QEMU with Expanded Disk

```
qemu-system-aarch64 \
  -machine raspi3b -cpu cortex-a72 -nographic \
  -dtb bcm2710-rpi-3-b-plus.dtb \
  -m 1G -smp 4 \
  -kernel kernel8.img \
  -sd 2023-05-03-raspios-bullseye-arm64.img \
  -append "rw earlyprintk loglevel=8 console=ttyAMA0,115200 dwc_otg.lpm_enable=0
root=/dev/mmcblk0p2 rootdelay=1" \
  -device usb-net,netdev=net0 -netdev user,id=net0,hostfwd=tcp::5010-:22
```

You can log into the PI within the current window, or you can SSH in from another WSL window:

```
ssh -p 5010 pi@localhost
```

You can save the above commands in a text file for starting up the simulator in the future.

6. Expand the Filesystem Inside the Emulated Pi

Once inside the Pi:

```
sudo raspi-config
```

- Go to **Advanced Options → Expand Filesystem**
- Reboot

Alternatively (manual method, usually not needed):

```
sudo fdisk /dev/mmcblk0
# delete partition 2 and recreate it to extend to the full disk
sudo reboot
sudo resize2fs /dev/mmcblk0p2
```

7. Verify the New Size

```
df -h
```

You should now see nearly 32 GB available in the root folder /.

This modified workflow only changes **Step 3 (resize)** + **Step 6 (expand FS)**, everything else is identical to the original blog's process.

8. Verify the New Size

```
df -h
```

Step 3. Enable 32-bit ARMv7 assembly on Raspberry Pi 3/4

Follow the next steps for running **32-bit ARMv7 assembly/programs on a Raspberry Pi 3/4 running Raspberry Pi OS 64-bit**:

1. Verify Hardware and OS Architecture

Check the CPU and OS:

```
uname -m    # should show aarch64
lscpu | grep -i model
```

Raspberry Pi 3 and 4 (ARMv8-A) *can* run ARMv7 binaries if the kernel supports AArch32.

2. Enable 32-bit (armhf) Support in Raspberry Pi OS 64-bit

By default, the 64-bit OS is "multi-arch capable." You need the 32-bit runtime:

```
sudo dpkg --add-architecture armhf
sudo apt update
sudo apt install libc6:armhf libstdc++6:armhf
```

If you need to run more complex ARMv7 programs (usually not needed):

```
sudo apt install gcc-arm-linux-gnueabihf g++-arm-linux-gnueabihf binutils-arm-linux-gnueabihf
```

3. Generating ARMv7 Binary

Save your **ARMv7 assembly** program, e.g., `hello.s`, by using a text editor (c.f., Linux text editors, the vi Editor Tutorial), and copying the following commands into it. The program is from this page https://embeddedjourneys.com/blog/hello-world-arm-assembly-raspberry-pi/. (Note: this program does not work in CPUlator, since CPUlator does not implement Linux system calls via SWI 0.)

```
global _start
_start:
        MOV R0, #1
        LDR R1, =msg
        MOV R2, #12
        MOV R7, #4
        SWI 0

        MOV R7, #1
        SWI 0
```

```
msg:
        .asciz "Hello, ARM!\n"
```

then assemble and link with the 32-bit toolchain:

```
arm-linux-gnueabihf-as -o hello.o hello.s
arm-linux-gnueabihf-ld -o hello hello.o
```

You'll notice the output binary is ELF32 for ARM, which will run in AArch32 compatibility mode. You can create command aliases to reduce typing, c.f. How to Create Aliases (Shortcuts) for Common Commands in Linux. For example, if you are running bash shell, edit .bashrc

vi ~/.bashrc

and add these lines:

```
alias as32='arm-linux-gnueabihf-as'
alias ld32='arm-linux-gnueabihf-ld'
alias gcc32='arm-linux-gnueabihf-gcc'
```

Check with:

```
file hello
# ./hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, not stripped
```

4. Run the ARMv7 Program Directly

Now run it:

```
./hello
```

The Pi's 64-bit kernel automatically switches into AArch32 user-mode for execution.

5. Debug/Inspect (Optional)

You can disassemble or inspect with:

```
arm-linux-gnueabihf-objdump -d hello
```

Step 4. Lab Report

Write a short report that records your experience in going through these steps. What problems you encountered and how did you resolve them.