CSIE 5310 Assignment 2 (Due on October 29th 14:10)

In the class, we introduced virtualized I/O for virtual machines. In this assignment, you are required to create a simple virtualized I/O system. To accomplish this, you will have to implement a virtual device.

0. Late submission policy

- 1 pt deduction for late submissions within a day (before Oct. 30th 14:10)
- 2 pts deduction for late submissions within 2 days (before Oct. 31st 14:10)
- zero points for submissions delayed by more than 2 days.

1. Before you start

- This assignment builds on the Ubuntu environment for KVM for Armv8 that you created in Assignment 1.
- Download the attachment `vm_hw2_files.zip` from NTU Cool into your Ubuntu host. There are 3 files used in this assignment:
  - `test-program.c`
  - `virt_walker.c`
  - `Makefile`

2. Implementing a virtual device and a kernel module

Virtual Device

You are asked to implement a simple virtual device containing two 1-byte registers named *HIDE* and *SEEK*. Your VM can access the device via MMIO.

The device is mapped to the physical address range `0x0b000000` to `0x0b000002`. The address of *HIDE* is `0x0b000000`, and the address of *SEEK* is `0x0b000001`.

Your device should perform operations on the guest's stage-2 page tables. Specifically:

- When the VM writes a byte value to *HIDE*, the device should store the value to bit `[58:51]` of the leaf stage-2 page table entry that maps GPA `0x40000000`.
- When the VM reads from *SEEK*, the device should return the byte value stored in the bit `[58:51]` of the leaf stage-2 page table entry that maps GPA `0x40000000`.

> Hint: In ARMv8, the MMU ignores bit `[58:51]` on a stage-2 page table entry so that you can modify those bits without affecting the page table walk.

> Hint: How do I find the leaf stage-2 page table entry that maps GPA 0x40000000? You might think of walking the VM's stage-2 page table using that GPA.

> Hint: Since your device will operate on stage-2 page tables, it may be easier to implement it in KVM rather than QEMU. To do this, examine how KVM handles MMIO and modify the relevant KVM code.

Kernel Module

After implementing the virtual device, the next step is to write a kernel module so the guest can use it. The kernel module performs the writes to *HIDE* and reads from *SEEK* on behalf of the application from user space.

The kernel module should create a `/dev/virt_walker` device node. User-space processes can then use system calls: `open()`, `read()`, and `write()` on this device node to perform MMIO operations.

- When a process `write()` a byte to `/dev/virt_walker`, the module should perform a MMIO write to *HIDE* with that byte. The return value of the `write()` system call should be `1`.
- When a process `read()` from `/dev/virt_walker`, the module should perform a MMIO read from *SEEK*. The return value of the `read()` system call should be `1`. Additionally, the result from the MMIO read should be propagated to the memory buffer specified by the `buf` argument of the `read()` system call.

You should submit the code for your kernel module and a `Makefile` to compile the module.

We provide a kernel module template (`virt_walker.c`) and a `Makefile` for you. You can extend the kernel module template. To compile the module, place the `Makefile` and `virt_walker.c` in the same directory on the machine where you compiled the kernel, and execute the following command.

```
make KDIR=/PATH/TO/kernel-source ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
```

Alternatively, you can craft your kernel module to implement the required features.

In either case (extend `virt_walker.c` or build your own), make sure to **submit the module's source code and the respective `Makefile`.** (see below).

3. Grading (10 pts)

Test (8 pts)

We will test your device and kernel module with the following setup:

1. Run the patched software for the KVM environment.
2. Launch a VM on the KVM host. (If you modified QEMU, we will use your modified QEMU to launch the guest.)
3. Compile your kernel module and install it on the VM.

You will receive full points if your device and kernel module implementation are correct.

You will receive zero points if your patch fails to apply or your implementation fails to compile. You must properly manage resources (e.g., free allocated memory) and handle errors.

The file `test_program.c` makes system calls to the kernel module and allows you to test your virtual device.

Write-up (2 pts)

Your write-up should include two sections:

- Instructions on compiling your kernel module.
- Explanations of how your code works, including the kernel module, kernel patch, and QEMU patch (if you modified QEMU).

4. Homework submission

You should submit the assignment via NTU Cool.

Submission format

Please submit the following files. Replace `[Student-ID]` with your student number. For example, if your student number is `r01234567`, then `[Student-ID]` should be `r01234567`.

- A write-up file named `write-up.pdf`.
- A folder named `kernel_module`, which contains the code for your kernel module and a `Makefile` to compile it. The `Makefile` does not have to copy or deploy the test program to your VM.
- A patch for the KVM (based on Linux v5.15) named `[Student-ID]_hw2_kernel.patch`.
- A patch for the QEMU (v7.0.0) named `[Student-ID]_hw2_qemu.patch`.

> You do not need to submit the QEMU patch if you did not modify it.

Please place those files in a folder named `[Student-ID]_hw2`. The folder structure should be the same as the following.

```
[Student-ID]_hw2
|---- write-up.pdf
|---- kernel_module
|     |---- Makefile
|     └---- CODE_FOR_YOUR_KERNEL_MODULE
|---- [Student-ID]_hw2_kernel.patch
└---- [Student-ID]_hw2_qemu.patch (if needed)
```

Then, compress the folder into `[Student-ID]_hw2.zip` and submit it to NTU Cool.