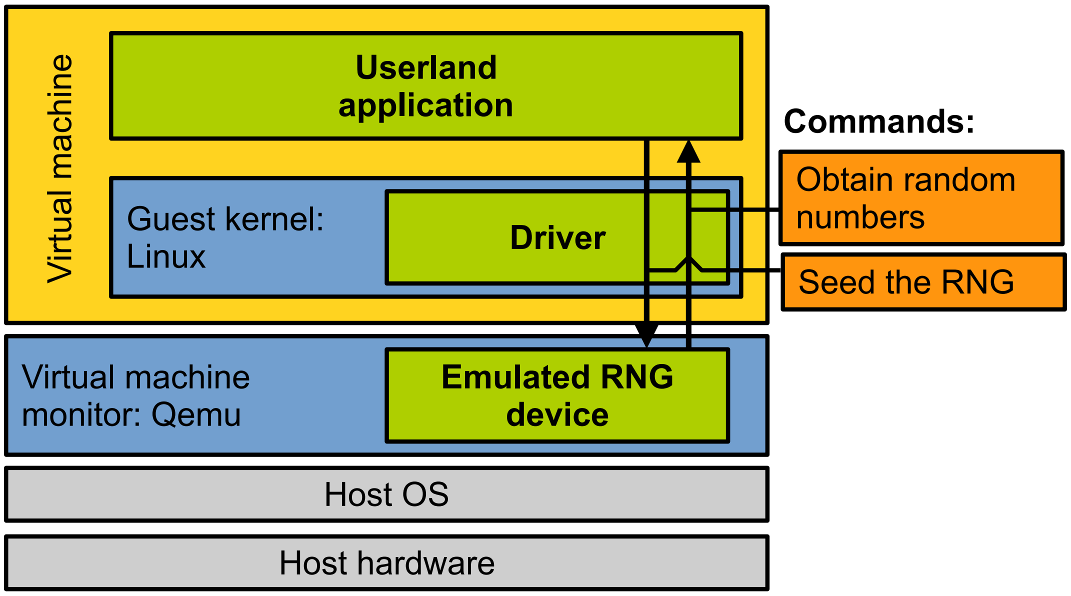
Assignment 8: Implement a Virtual Device and a Device Driver (Part 1)

Overview

[35 points] The objective of this assignment is to develop a simple emulated device in the QEMU virtual machine monitor (**Part 1**), to develop the guest driver controlling that device in the Linux kernel, and to write a small guest user space application making use of the device through the driver (**Part 2**). The emulated device is a simple random number generator (RNG).

The different software components you will have to develop are illustrated in green on the figure below:



The components to develop are:

1. The emulated device running on the host within QEMU.
2. A driver for the device running within the guest operating system, Linux.
3. A user space application leveraging the driver to make use of the device.

Random Number Generator Virtual Device

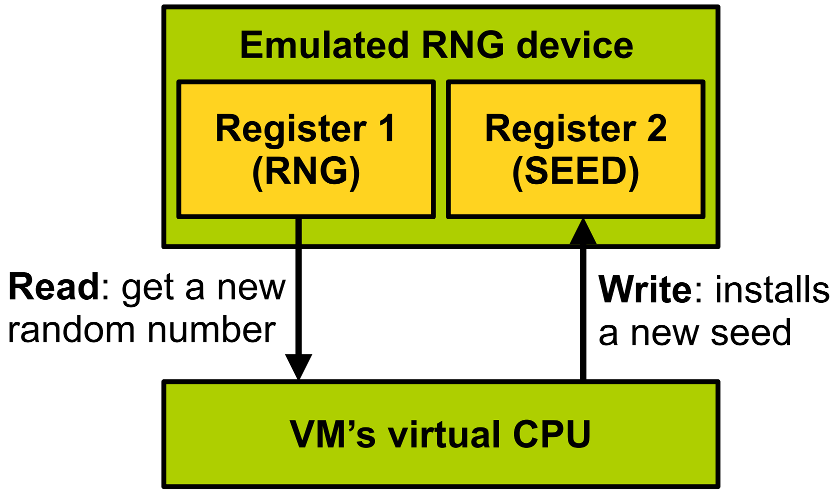
Our virtual RNG device offers two functionalities:

1. **Generating random numbers**: applications can query the virtual device through a driver in the

guest kernel to obtain random numbers.

2. **Seeding the RNG**: applications can initialize the RNG with a particular seed.

The random number generator will be connected to the VM's virtual CPU on the PCI bus, and communication between the device and the CPU will be achieved with memory mapped I/O registers. To function the devices exposes an interface made of two registers, illustrated below:



You can find more information about the registers in the table below.

Register name	Offset from base address	Size	Mode (R/W)	Description
`RNG`	`0x0`	4 bytes	R	Reading this register returns a random number in the form of a 32 bits unsigned integer. Each new read returns a new random number.
`SEED`	`0x4`	4 bytes	W	Writing an unsigned 32 bits number to this register seeds the random number generator with that value.

Part 1: Develop a Simple Virtual Device in QEMU

Part 1.1: Build QEMU from Source Code

[5 points] To develop the virtual device we will need to modify the QEMU virtual machine monitor, so a first step is to download its sources and make sure we can compile it.

Downloading and Extracting QEMU Sources

First, install the following packages:

```
sudo apt update -y
sudo apt install -y build-essential git bc libssl-dev flex bison wget python3 pyt
    ninja-build pkg-config libglib2.0-dev libelf-dev libslirp-dev
```

We will download QEMU from its official download page. Place yourself in the exercise base directory, and download the sources of QEMU version `8.2.9`:

```
mkdir ~/hw8
cd ~/hw8
wget https://download.qemu.org/qemu-8.2.9.tar.xz
```

Extract the archive as follows:

```
tar xf qemu-8.2.9.tar.xz
```

QEMU's sources are now in the folder `qemu-8.2.9.tar.xz`.

Compiling QEMU

Place yourself into QEMU's source folder and prepare the build by calling the `configure` script:

```
cd qemu-8.2.9
./configure --prefix=$PWD/prefix --target-list=x86_64-softmmu
```

Launch the build and trigger the installation once done: `make -j4 install`

This can take a bit of time depending on the processing power of your host.

Trying Out QEMU

Once the installation is done, you can check that all went well by launching the Linux kernel using this newly built QEMU. You can simply replace the `QEMU_BIN` inside `run-ubuntu.sh` to the path of the QEMU binary you just built:

```
#!/bin/bash
QEMU_BIN=~/hw8/qemu-8.2.9/prefix/bin/qemu-system-x86_64
... ...
```

You should be able to launch the Ubuntu VM using the QEMU you built.

Take a screenshot, and name it `part_1.1.png|jpg`.

Part 1.2: Implement a Virtual Device in Qemu

[20 points] We now have the VM set up and the sources of QEMU ready to be modified. We'll start by modifying QEMU to implement the virtual random number generator. The goal is to **emulate** that device, e.g. adhere to the same interface the guest OS would use to communicate with a real hardware component: the random number generator will be connected to the VM's virtual CPU on the PCI bus, and communication between the device and the CPU will be achieved with memory mapped I/O registers.

The implementation of the RNG itself (e.g. how random numbers are generated) will be done completely in software, for example by using the `rand()` and `srand()` functions provided by the C standard library in QEMU on the host.

Adding a New Source File in Qemu

We'll start by creating a new C file in which we will implement the device:

```
cd ~/hw8/qemu-8.2.9
touch hw/misc/my-rng.c
```

Next we need to add that file to the build system so that it gets compiled and linked against the rest of QEMU sources. Add the following at the top of the file `hw/misc/Kconfig`:

```
config MY_RNG
    bool
    default y
```

And add that line at the top of the file `hw/misc/meson.build`:

```
system_ss.add(when: 'CONFIG_MY_RNG', if_true: files('my-rng.c'))
```

A modification of the build system requires reconfiguring and recompiling all of QEMU sources. To do so simply type the following command in QEMU's sources root directory:

```
make -j4 install
```

To make sure your file is included in the build you can force its recompilation as follows:

```
touch hw/misc/my-rng.c
make
```

You should see in the output:

```
[3/4] Compiling C object libcommon.fa.p/hw_misc_my-rng.c.o
```

Implementing the Device

Now we will implement the virtual random number generator in `hw/misc/my-rng.c`. We'll first need to include the following headers as they define data structures and functions we need:

```
#include "qemu/osdep.h"
#include "hw/pci/msi.h"
#include "hw/pci/pci.h"
```

Next we define the device's name with a macro, and create a data structure representing the device:

```
#define TYPE_MY_RNG "my_rng"
#define MY_RNG(obj) OBJECT_CHECK(my_rng, (obj), TYPE_MY_RNG)

typedef struct {
    PCIDevice parent_obj;
    uint32_t seed_register;
    MemoryRegion mmio;
} my_rng;
```

The important bits here are the `seed_register` member, that we will use to hold the seed, and `mmio`, a data structure that will hold functions to read and write from the device's memory mapped registers.

Next, we define the functions that will run when the device's memory mapped registers are read/written:

```
static uint64_t mmio_read(void *opaque, hwaddr addr, unsigned size) {
    /* TODO implement that function later */
    return 0x0;
}

static void mmio_write(void *opaque, hwaddr addr, uint64_t val, unsigned size) {
    /* TODO implement that function later */
    return;
}

static const MemoryRegionOps my_rng_ops = {
    .read = mmio_read,
    .write = mmio_write,
};
```

It will be your task to implement these functions later. For now it is fine to leave them empty. Notice the `my_rng_ops` data structure that contain members pointing to both functions.

The rest of the source file contains a series of initialisation functions:

```
static void my_rng_realize(PCIDevice *pdev, Error **errp) {
    my_rng *s = MY_RNG(pdev);
    memory_region_init_io(&s->mmio, OBJECT(s), &my_rng_ops, s,
                          "my_rng", 4096);
    pci_register_bar(&s->parent_obj, 0, PCI_BASE_ADDRESS_SPACE_MEMORY, &s->mmio);
}

static void my_rng_class_init(ObjectClass *class, void *data) {
    DeviceClass *dc = DEVICE_CLASS(class);
    PCIDeviceClass *k = PCI_DEVICE_CLASS(class);

    k->realize = my_rng_realize;
    k->vendor_id = PCI_VENDOR_ID_QEMU;
    k->device_id = 0xcafe;
    k->revision = 0x10;
    k->class_id = PCI_CLASS_OTHERS;

    set_bit(DEVICE_CATEGORY_MISC, dc->categories);
}

static void my_rng_register_types(void) {
    static InterfaceInfo interfaces[] = {
        { INTERFACE_CONVENTIONAL_PCI_DEVICE },
        { },
    };

    static const TypeInfo my_rng_info = {
        .name = TYPE_MY_RNG,
        .parent = TYPE_PCI_DEVICE,
        .instance_size = sizeof(my_rng),
        .class_init    = my_rng_class_init,
        .interfaces = interfaces,
    };

    type_register_static(&my_rng_info);
}

type_init(my_rng_register_types)
```

You don't need to fully understand this code. Notable things here are:

- The `my_rng_realize` function that initializes an instance of the virtual random number generator by:

    - Creating a region of I/O memory for the memory mapped registers with `memory_region_init`. That region has a size of 4 KB which is much larger than what we need (we have 2 registers of 4 bytes each) but corresponds to the size of a memory page.
    - Registering the device on the PCI bus with `pci_register_bar`.

- The `my_rng_class_init` that will run once when QEMU starts and define a few characteristics common to all instances of our virtual device, such as an easily identifiable device ID (`0xcafe`). A member `realize` of the corresponding `PCIDeviceClass` data structure also points to the per-instance initialisation function `my_rng_realize`.

At that point you can try to recompile QEMU by typing, at the root of its source folder:

```
make install
```

You should fix any error or warning at that stage. Once everything compiles fine we can check if the device appears in the VM.

Checking the Presence of the Device in the VM

To enable the device in the VM, edit the launch script `run-ubuntu.sh` and add the following command line option to QEMU's invocation:

```
-device my_rng
```

You can check the presence of the virtual device by enumerating PCI devices in the VM. Boot the VM and enumerate PCI devices:

```
sudo lspci -v
```

You should see the following device:

```
00:04.0 Unclassified device [00ff]: Device 1234:cafe (rev 10)
    Subsystem: Red Hat, Inc. Device 1100
    Flags: fast devsel
    Memory at febf1000 (32-bit, non-prefetchable) [size=4K]
```

You can recognize the device ID `0xcafe` we defined earlier. Notice also the address where the device's registers are mapped in (physical) memory. Here it is `0xfebf1000` but it may be different on your computer.

Take a screenshot, and name it `part_1.2.png|jpg`.

Implementing the Read/Write MMIO Functions

To finalize the implementation of our virtual random number generator, one must implement the two functions we defined earlier:

```
static uint64_t mmio_read(void *opaque, hwaddr addr, unsigned size) {
    /* TODO */
    return 0x0;
}

static void mmio_write(void *opaque, hwaddr addr, uint64_t val, unsigned size) {
    /* TODO */
    return;
}
```

You are responsible to implement these functions. A bit of information to help you achieve that:

- `mmio_read` is called when the guest OS tries to read in one of the device's memory mapped registers, and `mmio_write` is called when a register is written. `mmio_read` returns the value that the guest OS will read.
- The `addr` parameter will contain the offset from the base address in the area of memory mapped I/O at which the read/write takes place, which should allow you to identify the target register.
- The `size` parameter denotes the size of the read/write operation.
- The `opaque` pointer points to the device's data structure of type `my_rng`, so you can get a pointer to the device's data structure with a cast: `my_rng *dev = (my_rng *)opaque;`.
- The actual RNG should be implemented in software and the easiest way to achieve that is probably to use the standard C library's functions `rand()` (to get a random number) and `srand()` (to seed the random number generator).

Rubric:

- **[10 points]** for seeing the virtual device.
- **[10 points]** for implementing Read/Write MMIO Functions

Part 1.3: Test the Virtual Device from the Guest Kernel

[10 points] Before writing the actual driver it is probably a good idea to a quick test of the device from the guest kernel and check it behaves correctly. To that aim we can do a small modification of the Linux guest kernel sources, and insert some calls to the device. For the sake of simplicity we'll insert these calls at the end of the boot process, when the system is well initialized but without involving the user space.

Locating the Kernel Main Function

The kernel is a computer program like any other and as such it has an entry point. This entry point is written in assembly but after a short early initialisation, the CPU will jump to C code. More precisely, the C entry point of the kernel is the function `start_kernel`, which is implemented in the Linux sources in the file `init/main.c`.

If you check out its implementation, you'll see that `start_kernel` initializes many subsystems and then call `arch_call_rest_init`, which itself calls `rest_init`. `rest_init` spawns a kernel thread that runs the `kernel_init` function. The `kernel_init` function finalizes the initialisation of the system and then starts the first user space application. This is a suitable point in the boot process to insert our test calls to the device, because the system is fully initialized, and we are also still in kernel space.

Inserting Test Calls to the Device

Our test will perform the following things:

1. Seed the RNG with a fixed seed e.g. `0x42`
2. Generate 5 random numbers and print them on the kernel log

Steps 1 and 2 will be repeated twice, so we can check that the 5 random numbers generated from the same seed are the same for both iterations.

In the `kernel_init` function, add the following code after the call to `do_sysctl_args();` (it's around line 1464):

```
printk("------------------------------------------------------------------\n");
printk("BEGIN MY-RNG TEST\n");
printk("------------------------------------------------------------------\n");

// Map the area of physical memory corresponding to the device's registers
// (starting 0xfebf1000, size 4KB) somewhere in virtual memory at address
// devmem. Notice that the physical memory where the device's registers are
// present may be different on your computer, use lspci -v in the VM to
// find it
void *devmem = ioremap(0xfebf1000, 4096);
unsigned int data = 0x0;
if(devmem) {
    for(int i = 0; i < 2; i++) {
        // seed with 0x42 by writing that value in the seed register which
        // is located at base address + 4 bytes
        iowrite32(0x42, devmem + 4);

        // obtain and print 5 random numbers by reading the relevant
        // register located at base address + 0
        for(int j = 0; j < 5; j++) {
            data = ioread32(devmem);
            printk("Round %d number %d: %u", i, j, data);
        }
    }
} else {
    printk("ERROR: cannot map device registers\n");
}

printk("------------------------------------------------------------------\n");
printk("END MY-RNG TEST\n");
printk("------------------------------------------------------------------\n");
```

A few notable things in this code:

- We use `printk` to print to the kernel log. It's very similar to the `printf` function you are familiar with in user space. With printk we display when the test starts and ends so that things are clearly visible in the kernel log.
- The test code starts by mapping the physical memory where the device's registers are present into virtual memory (shortly after the very early boot process the CPU can only access virtual memory) at an address pointed by `devmem`. This is achieved with the `ioremap` function, that takes as parameters the physical address to map into virtual memory, as well as the size of the area to map (here one page, i.e. 4 KB, as defined when we implemented the device). Note the address in physical memory where the device's registers are mapped, here `0xfebf1000`. It may be

different on your computer. To find it out, you can use `lspci` within the VM, as previously explained.

- Once the device's registers are mapped into virtual memory, we can read and write to them using `ioread32` and `iowrite32`. It's important to use these functions rather than directly read/write to memory because these are not standard memory access operations: these functions will ensure important things like bypassing the CPU caches, disabling compiler optimisations, and will have memory barriers preventing the compiler/CPU to reorder the corresponding instructions. Through these functions we have two types of operations when talking to the device:
    - **Generating a random number**: to achieve that we read with `ioread32` the first register which is located directly at the device's base address
    - **Seeding the RNG**: to do so we write with `iowrite32` the second register which is located 32 bits (4 bytes) from the base address

Launching the Test

Once the test code is ready you can recompile the guest Linux kernel:

```
make
```

When you launch the VM with this newly compiled kernel, you should see in the log at the end of the kernel boot process something like that:

```
[    3.519214] ------------------------------------------------------------------
[    3.519510] BEGIN MY-RNG TEST
[    3.519620] ------------------------------------------------------------------
[    3.520024] Round 0 number 0: 286129175
[    3.520046] Round 0 number 1: 1594929109
[    3.520199] Round 0 number 2: 971802288
[    3.520394] Round 0 number 3: 222134722
[    3.520559] Round 0 number 4: 1335014133
[    3.520754] Round 1 number 0: 286129175
[    3.520918] Round 1 number 1: 1594929109
[    3.521073] Round 1 number 2: 971802288
[    3.521227] Round 1 number 3: 222134722
[    3.521406] Round 1 number 4: 1335014133
[    3.521545] ------------------------------------------------------------------
[    3.521965] END MY-RNG TEST
[    3.522101] ------------------------------------------------------------------
```

As you can see for each round the series of random number generated are the same, which confirms that the RNG virtual device works well.

Take a screenshot, name it to `part_1.3.png|jpg` .

Deliverables:

- Upload 3 screenshots.
- Create a patch for QEMU and a patch for the Linux kernel, then upload both patch files.