Assignment 9: Implement a Virtual Device and a Device Driver (Part 2)

Part 2.1: Implementing a Driver within the Guest Kernel

[25 points] Here we will modify the guest Linux kernel again, this time we will implement a proper driver, manipulating the device and exposing its functionalities to the application from user land.

User-Kernel Space Communication with `ioctl`

The goal of an operating system (OS) is to provide user space application with safe and controlled access to the hardware. To that aim the OS implements a driver that manipulates the hardware directly, and that driver offers an interface to user space applications. It is possible to use different types of interfaces, such as implementing a new system call or using virtual files. The one that we will use for this exercise is called input/output control (`ioctl`).

With `ioctl`, the driver will create a virtual file on the VM's root filesystem representing the device, `/dev/my_rng_driver`. A user space application wishing to access our random number generator device will open that file and perform operations on it through a particular system call, `ioctl`, as illustrated below:

As the hardware device provides 2 functionalities, there will be 2 `ioctl` operations available: one to generate a new random number, and another to seed the RNG.

Step 1: Adding a Source File to the Linux Kernel Sources

[5 points] Our driver should be implemented in its own source file. So first we need to create a new C file in the kernel sources and add it to the build system so that it gets compiled with the rest of the kernel sources. To do so, let's first navigate to the kernel sources directory and create a file in the `drivers/misc/` directory:

```
cd ~/linux        # replace it with your Linux code location

touch drivers/misc/my-rng.c
```

Next let's add it to the build system. Edit `drivers/misc/Makefile` and add this line at the top of the file:

```
obj-y += my-rng.o
```

This indicates the kernel build process to add my-rng.c into every build of the kernel.

Step 2: Implementing the Driver

[15 points] In `drivers/misc/my-rng.c`, let's start by including the necessary headers:

```
#include <linux/ioctl.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#include <linux/io.h>
```

These will let us define icotl operations, create the virtual file corresponding to the device, map the physical memory corresponding to the device's register into virtual memory, and access these registers.

Next we define the two `ioctl` operations our driver will support:

```
#define MY_RNG_IOCTL_RAND _IOR('q', 1, unsigned int)

#define MY_RNG_IOCTL_SEED _IOW('q', 1, unsigned int)
```

With `_IOR` we define an `ioctl` operation `MY_RNG_IOCTL_RAND` that will allow the application to *read* data from the device's `RNG` register, i.e. to get a random number. With `_IOW` we define an operation for the application to *write* data to the device's `SEED`register, i.e. to seed the random number generator. The parameters are not particularly important, but note that the last one specifies the size of what is read/written: an `unsigned int`, i.e. a 32-bit unsigned integer.

Next we define a macro with the base physical address where the device's registers are mapped into memory:

```
#define DEVICE_BASE_PHYS_ADDR 0xfebf1000
```

Please note that this value may be different on your computer. To find the proper value you can use `lspci -v` within the VM as previously explained.

We also need a pointer that will hold the location where the device's registers are mapped in virtual memory (recall that the CPU can only access virtual memory):

```
void *devmem = 0x0;
```

Next we implement the functions that will access the device. These will be called when the user land application invokes `ioctl` on the virtual file `/dev/my_rng_driver`.

```
static long my_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {

   switch (cmd) {

      case MY_RNG_IOCTL_RAND:
         /* Application requests a new random number */
         /* TODO implement that feature */

         break;

      case MY_RNG_IOCTL_SEED:
         /* Application requests to seed the RNG */
         /* TODO implement that feature */
         break;

      default:
         return -ENOTTY; // unknown command
   }

   return 0;
}

static struct file_operations my_rng_fops = {
   .unlocked_ioctl = my_ioctl,

};
```

Here the `cmd` parameter contains the exact `ioctl` command that was called by the application. With a switch we separate the processing according to what the application requests, either `MY_RNG_IOCTL_RAND` or `MY_RNG_IOCTL_SEED`.

It will be your responsibility to implement these commands. A few important things to note:

- When either reading or writing data from/to the device through `ioctl`, the parameter `arg` will contain:
  - The address in user space of the data to write to the device in case of a write operation.
  - The address in user space of where to store the data to read in case of a read operation.
- It is **unsafe** to read/write from/to user space addresses directly (the user space application could have for example passed the kernel `NULL` pointers). To properly access these addresses, you need to use:
  - `copy_to_user` when copying data read from the device into user space memory; and
  - `copy_from_user` when reading data from user space in order to write it to the device.
- At that stage you can assume that memory pointed by `devmem` has already been properly mapped somewhere in virtual memory by the driver initialization function (presented below) and you don't need to call `ioremap` in `my_ioctl`.
- You should access the device by taking inspiration from the `test code` we wrote in the Assignment 8 (Part 1.3).

The handler `my_ioctl` is wrapped into a `file_operations` data structure that we will use to indicate the operations possible on the virtual file the driver will create in `/dev`.

Finally, we can implement the initialization and destruction functions for our driver:

```
static int __init my_rng_driver_init(void) {
   devmem = ioremap(DEVICE_BASE_PHYS_ADDR, 4096);

   if(!devmem) {
      printk(KERN_ERR "Failed to map device registers in memory");
      return -1;
   }

   if (register_chrdev(250, "my_rng_driver", &my_rng_fops) < 0) {
      printk(KERN_ERR "Failed to register my_rng_driver\n");
      return -1;
   }

   printk("my_rng_driver loaded, registered ioctls 0x%lx (get a random "
      "number) and 0x%lx (seed the generator) \n", MY_RNG_IOCTL_RAND,
      MY_RNG_IOCTL_SEED);
   return 0;
}

static void __exit my_rng_driver_exit(void) {
   unregister_chrdev(250, "my_rng_driver");

   if(devmem)
      iounmap(devmem);

   printk(KERN_INFO "my_rng_driver unloaded\n");
}

module_init(my_rng_driver_init);

module_exit(my_rng_driver_exit);
```

The initialization function `my_rng_driver_init` is executed when the kernel boots. It starts by mapping the device's registers into virtual memory with `ioremap`, as we have seen in the `test code` we wrote previously. Next it registers a character device named `my_rng_driver` into the kernel, with an identification number (called **major number**) of `250`. We'll use that number later when we create in the VM the virtual file that will play the role of interface between a user space application and the driver living in the kernel.

The driver exit function `my_rng_driver_exit` is executed when the kernel shuts down. It simply unregisters the character device, and unmaps the device register's from virtual memory.

The initialization and exit functions are indicated with `module_init` and `module_exit`.

Once all the code is written, you can recompile Linux by typing at the root of its sources:

```
make
```

Step 3: Checking the Presence of the Driver

[5 points] Once the kernel is recompiled, reboot the VM and check in the kernel log the line written by the driver when it loads. It may be a bit hard to find because a lot of stuff is printed on that log when the kernel boots. Once you get a shell you can print and filter the kernel log with `dmesg` and `grep`:

```
dmesg | grep my_rng_driver
```

```
[    0.869353] my_rng_driver loaded, registered ioctls 0x80047101 (get a random number) and 0x40047101 (seed the generator)
```

Note that the `ioctl` numbers may be different on your machine.

Take a screenshot, and name it `part_2.1.png|jpg`.

Part 2.2: Accessing the Device From User Space

[15 points] This is the final step of the guided part of this assignment. We will now develop a simple user space application inside the QEMU VM that accesses the virtual device through the driver we just implemented.

Step 1: Creating the Virtual File for the Device

[5 points] Before we can write the user space app that connects to the device through the driver, we need to create the virtual file `/dev/my_rng_driver` mentioned in the Part 2.1. To do so, type the following command within the VM:

```
sudo mknod /dev/my_rng_driver c 250 0
```

The major number, here 250, must match the one you defined within the driver in the initialization function. After invoking `mknod`, a virtual file should be present in `/dev`:

```
ls -l /dev/my_rng_driver
```

```
crw-r--r--    1 root     root      250,   0 Dec 20 22:32 /dev/my_rng_driver
```

You will need to repeat this operation each time the VM reboots.

Writing the User Space Application

[10 points] The source code of the user space application follows. We start by including a few headers for printing to the standard output, accessing files, and performing `ioctl` commands.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#include <sys/ioctl.h>
```

Next we have two constants that are the `ioctl` numbers that were allocated for the 2 functions offered by the driver. To find them look in the VM's kernel log (Part 2.1 **Step 3**).

```
#define RAND_IOCTL  0x80047101
```

```
#define SEED_IOCTL  0x40047101
```

Finally, we have the main function that contains our test code:

```
int main() {
   int fd = open("/dev/my_rng_driver", O_RDWR);
   if (fd < 0) {
      perror("Failed to open the device file");
      return -1;
   }

   unsigned int seed = 0x0;
   unsigned int random_number = 0;

   for(int i=0; i<2; i++) {

      // seed the generator
      if(ioctl(fd, SEED_IOCTL, &seed)) {
         perror("ioctl seed");
         return -1;
      }

      // get 5 random numbers
      for (int j=0; j<5; j++) {
         if(ioctl(fd, RAND_IOCTL, &random_number)) {
            perror("ioctl rand");
            return -1;
         }

         printf("Round %d number %d: %u\n", i, j, random_number);
      }
   }

   close(fd);
   return 0;

}
```

This code starts by opening the virtual file representing the driver, `/dev/my_rng_driver`. It then follows similar steps to our in-kernel test we ran earlier: we seed the RNG, and generate 5 random numbers. We do that twice in a row to confirm that with the same seed, the device will return the same sequence of random numbers. Notice how `ioctl` is called with as parameter:

- The virtual file descriptor `fd`
- The `ioctl` code we want to invoke (`RAND_IOCTL` or `SEED_IOCTL`)
- The address of a variable that will be filled with the random number generated (for `RAND_IOCTL`), or the address of a variable holding the seed we want to use (for `SEED_IOCTL`).

You can compile that code within the VM, assuming you write it in a file named `my-app.c` as follows:

```
gcc my-app.c -o my-app
```

When launching the program, you should see a series of 2 similar random number sequences:

```
./my-app
Round 0 number 0: 1804289383
Round 0 number 1: 846930886
Round 0 number 2: 1681692777
```

```
Round 0 number 3: 1714636915
Round 0 number 4: 1957747793
Round 1 number 0: 1804289383
Round 1 number 1: 846930886
Round 1 number 2: 1681692777
Round 1 number 3: 1714636915

Round 1 number 4: 1957747793
```

That's it! By now, you should have a decent understanding of how I/O device virtualization, device drivers, and application/device interaction work!

Take a screenshot, and name it `part_2.2.png|jpg`.

Deliverables:

- Upload 2 screenshots.
- Create a patch for the Linux kernel, then upload the patch file and the user space code file.

[Optional] Part 2.3: Beyond this Assignment

[10 points] This optional task encourages you to explore the virtualization techniques covered in class and this assignment.

You may expand on what you've learned by adding novel features. For example, you could develop a tiny VMM by extending the KVM interface to support running the xv6 kernel, implement a virtual string encryption device (Assignment 3), or create anything that enhances this assignment or the course lectures.

You can email me your code by May 1, 2025. Please note that your solution cannot be part of your final project.