# ← The Kernel Module

## Project 5

Writing a device driver from scratch is kind of tedious and wouldn't really teach you much, so we've given you a starting point and example code which you will expand upon.

Just follow these example module directions carefully. It'll only take a few minutes.

---

## An example kernel module

On thoth, login and cd to your `~/private` directory. Then run:

```
(1) thoth $ tar xvfz /u/SysLab/shared/hello_dev.tar.gz
```

Then,

```
(10) thoth $ cd hello_dev
(11) thoth $ ls
hello_dev.c  hello.rules  Makefile
(12) thoth $ _
```

First we have to modify the Makefile to compile against the proper version of the kernel. Open the Makefile in your editor and change the line:

```
KDIR  := /lib/modules/$(shell uname -r)/build
```

to

```
KDIR  := /u/SysLab/shared/linux-2.6.23.1
```

Save the Makefile, and run `make ARCH=i386`. The `ARCH=i386` is important because thoth is a 64-bit machine, but the VM is a 32-bit machine.

```
(22) thoth $ make ARCH=i386
make -C /u/SysLab/shared/linux-2.6.23.1 M=/afs/pitt.edu/home/x
```

```
make[1]: Entering directory '/u/SysLab/shared/linux-2.6.23.1'
  CC [M]  /afs/pitt.edu/home/x/y/xyz00/private/hello_dev/hello
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /afs/pitt.edu/home/x/y/xyz00/private/hello_dev/hello
  LD [M]  /afs/pitt.edu/home/x/y/xyz00/private/hello_dev/hello
make[1]: Leaving directory '/u/SysLab/shared/linux-2.6.23.1'
(23) thoth $ _
```

Now there should be a `hello_dev.ko` file. This is the device driver kernel module!

Now we have to get it into your VM.

"I CAN'T BUILD THE DEVICE DRIVER, I GOT THIS ERROR"

```
  MODPOST 1 modules
FATAL: /afs/pitt.edu/home/x/y/xyz00/private/hello_dev/hello_de
```

You didn't put `ARCH=i386` on the command line when running `make`.

---

## Installing the driver into the VM

**In your QEMU VM, run the following (using your username instead of USERNAME... 😔):**

```
root@tiny ~ # scp USERNAME@thoth.cs.pitt.edu:~/private/hello_d
```

Enter your password, and it will copy `hello_dev.ko` from your private directory on thoth into the VM!

Now you can load the driver using insmod:

```
root@tiny ~ # insmod hello_dev.ko
root@tiny ~ # _
```

We now need to make the device file in `/dev`. First, we need to find the MAJOR and MINOR numbers that identify the new device:

```
root@tiny ~ # cat /sys/class/misc/hello/dev
10:63
root@tiny ~ # _
```

You can see the output is `10:63`. The 10 is the MAJOR device number, and the 63 is the MINOR device number. **Your numbers may be different!**

To make `/dev/hello`, we can use the `mknod` command. The name will be `/dev/hello` and it is a character device. The 10 and the 63 correspond to the MAJOR and MINOR numbers we discovered above (use whatever numbers the `cat` above gave you).

```
root@tiny ~ # mknod /dev/hello c 10 63
root@tiny ~ # _
```

And finally, we can call the device's `read()` function by using `cat`:

```
root@tiny ~ # cat /dev/hello
Hello, world!
root@tiny ~ # _
```

---

## Uninstalling the driver

We can clean up by removing the device and unloading the module:

```
root@tiny ~ # rm /dev/hello
rm: remove '/dev/hello'? y
root@tiny ~ # rmmod hello_dev.ko
root@tiny ~ # _
```

---

## Your dice driver

Go back to thoth now.

`cd` out of the `hello_dev` folder and rename it:

```
mv hello_dev dice_dev
```

Then `cd` back into it and:

- Rename `hello_dev.c` to `dice_dev.c`
- **Edit the** `Makefile` so the `obj-m` line says `dice_dev.o` instead.

Open and read through `dice_dev.c`. The comments explain many of the things that are going on. **Remove them after you have read them.**

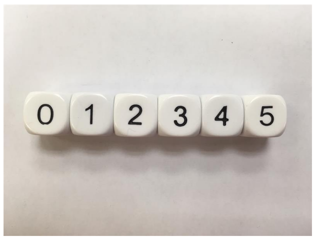
Then search-and-replace `hello` to `dice`, and change the `MODULE_AUTHOR` and `MODULE_DESCRIPTION` at the bottom. (You're the author, silly!)

Then try building it as before. It should still work, since you just changed the names of some things. You should end up with `dice_dev.ko`, which you could `scp` into your VM just like with the example.

---

## Your task

You will change the `dice_read` (which used to be `hello_read`) function so that it will **fill the output buffer** with a random sequence of dice rolls.


*You know, like normal dice.*

A single dice roll is a **1 byte value** in the range **0 to 5, inclusive.**

Your driver must support reading an **arbitrary number of die rolls.** If the user requests 10 rolls (that is, they do a `read()` asking for 10 bytes), you should fill the buffer with **10 different random numbers from 0 to 5.**

### The parameters to `dice_read` and its return value

- `file`: ignore this!
- `buf`: the pointer to the **user-mode process's buffer** to fill.
  - You **cannot write directly to this!** See below.
- `count`: how many bytes (dice rolls) the user wants.
- `ppos`: a pointer to a "position" variable. See below.

It returns **the number of bytes that were read.** In your driver's case, it's just gonna be `count`, unless some error occurred.

### What to do

Look at the original `hello_dev` code. It:

- returns an error if the user is trying to read fewer bytes than the entire string
- returns 0 to say “no more data” if the position ( `*ppos` ) is non-zero
- uses `copy_to_user` to copy data from its `hello_str` buffer to the user `buf`

You are going to be doing something very similar.

The `ppos` argument is a pointer to the “current position” within the file - this is managed by the kernel at a slightly higher level than your read function.

First, **remove this code that was from the example driver:**

```
if(*ppos != 0)
        return 0;
```

Your “file” will be infinitely long,

At the end of your `dice_read` function, do `*ppos = *ppos + count`. This will tell the OS “hey, we’ve moved `count` bytes through the file.”

The return value from your `dice_read` function should just be the `count` that the user asked for (if it’s not an error value).

## Generating the random numbers

The kernel does not have the full C Standard library available to it, so we need to get use a different function to get random numbers.

**Include `<linux/random.h>`.** Now you can use the function `get_random_bytes()` . This function fills a buffer with a sequence of random bytes in the range 0 to 255 inclusive. For example:

```
unsigned char mybuf[8];
get_random_bytes(mybuf, sizeof(mybuf));
// Now mybuf contains 8 random bytes!
```

From there, you can turn each byte into a random number in **the range 0-5** by using the modulo operator. Loop over the array and use `%=` on each item.

## Hmm…

Let’s say the user wants 100 bytes. Remember, **stack space in the kernel is limited.** So you can’t do this:

```
// NO!!!!! BAD!!!!!!!!!
unsigned char mybuf[count];
```

So there are two possible ways to solve this:

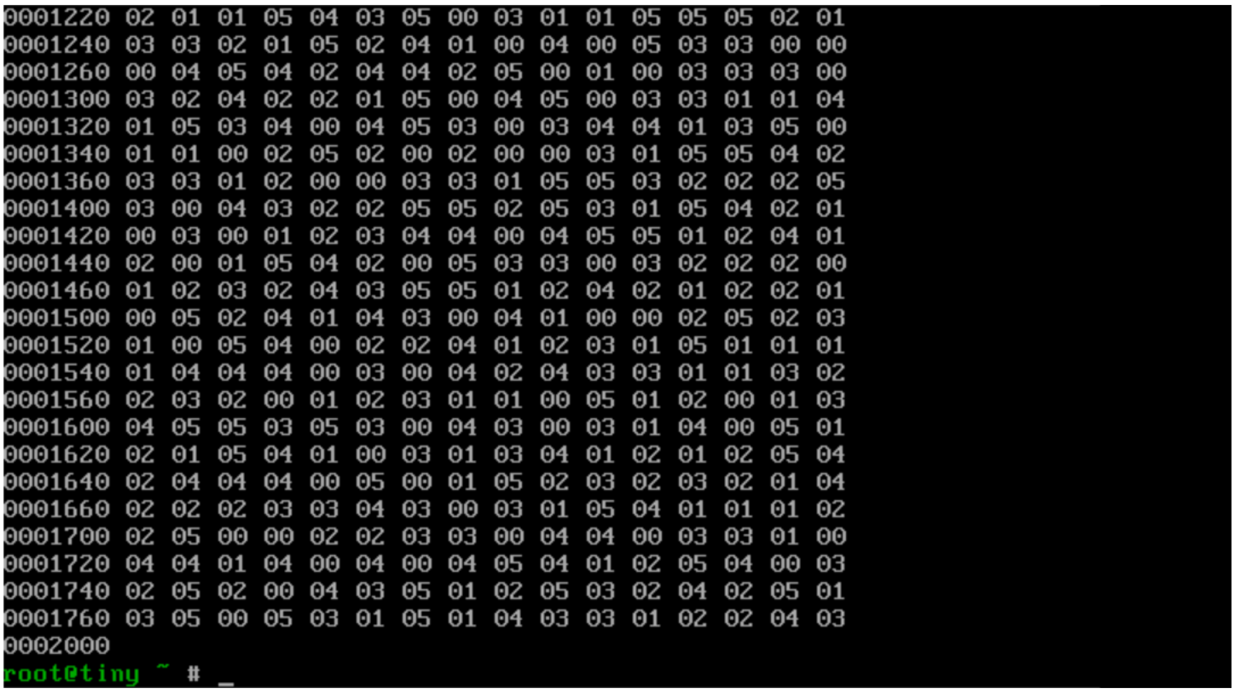
1. Use a small, fixed-size buffer (as shown above), and repeatedly fill/copy it to fill the user buffer up piece by piece.
    - This is more efficient, but trickier to write.
2. Use `kmalloc/kfree` to allocate a dynamically-sized buffer.
    - Do `kmalloc(size, GFP_KERNEL)` - it returns a `void*` like `malloc()` does.
    - Then be sure to `kfree()` everything you `kmalloc()`...
        - In ALL CASES!
        - Consider error cases too!

## Notes

- If you want to print some debug messages, use `printk()`:
    - `printk(KERN_ERR "Hmm, something went wrong...\n");`
- If the module crashes, it may become impossible to delete the `/dev/dice` file. If that happens, just grab a fresh `tty.qcow2` disk image and `scp` the driver again.
    - This is a big advantage to developing in a VM!
- To test your driver without the Craps game, you can run this in your QEMU VM after installing your driver:

```
# od -t x1 /dev/dice
```

This should print a bunch of hex numbers and they should all be 00, 01, 02, 03, 04, 05 (except for the addresses in the first column):



```
0001220 02 01 01 05 04 03 05 00 03 01 01 05 05 05 02 01
0001240 03 03 02 01 05 02 04 01 00 04 00 05 03 03 00 00
0001260 00 04 05 04 02 04 04 02 05 00 01 00 03 03 03 00
0001300 03 02 04 02 02 01 05 00 04 05 00 03 03 01 01 04
0001320 01 05 03 04 00 04 05 03 00 03 04 04 01 03 05 00
0001340 01 01 00 02 05 02 00 02 00 00 03 01 05 05 04 02
0001360 03 03 01 02 00 00 03 03 01 05 05 03 02 02 02 05
0001400 03 00 04 03 02 02 05 05 02 05 03 01 05 04 02 01
0001420 00 03 00 01 02 03 04 04 00 04 05 05 01 02 04 01
0001440 02 00 01 05 04 02 00 05 03 03 00 03 02 02 02 00
0001460 01 02 03 02 04 03 05 05 01 02 04 02 01 02 02 01
0001500 00 05 02 04 01 04 03 00 04 01 00 00 02 05 02 03
0001520 01 00 05 04 00 02 02 04 01 02 03 01 05 01 01 01
0001540 01 04 04 04 00 03 00 04 02 04 03 03 01 01 03 02
0001560 02 03 02 00 01 02 03 01 01 00 05 01 02 00 01 03
0001600 04 05 05 03 05 03 00 04 03 00 03 01 04 00 05 01
0001620 02 01 05 04 01 00 03 01 03 04 01 02 01 02 05 04
0001640 02 04 04 04 00 05 00 01 05 02 03 02 03 02 01 04
0001660 02 02 02 03 03 04 03 00 03 01 05 04 01 01 01 02
0001700 02 05 00 00 02 02 03 03 00 04 04 00 03 03 01 00
0001720 04 04 01 04 00 04 00 04 05 04 01 02 05 04 00 03
0001740 02 05 02 00 04 03 05 01 02 05 03 02 04 02 05 01
0001760 03 05 00 05 03 01 05 01 04 03 03 01 02 02 04 03
0002000
root@tiny ~ # _
```

And hit ctrl-C to stop the madness.

If you are getting numbers like 255, 254, 253, etc. make sure your buffer is an `unsigned char` array, and not a regular `char` array.