Creating device drivers in archlinux running on a VirtualBox VM

Adam Hoover

The following instructions allow you to complete this lab on your own computer. Additional information is given in class and on the course website.

Prelude and outline

You will complete 3 exercises. In the first, you create a simple module. In the second, you create a device driver for a virtual 1-byte storage. In the third, you write your own code for a device driver for a virtual keygen device.

You can try to work from one of the same virtual machines (VMs) that you used in the lab on the boot process and kernel compiling. However, I cannot guarantee that all the following instructions will work as-is, depending on choices you made and problems you had to solve while setting up your VM and compiling the kernel. It may be easier for you to repeat these steps from the previous lab so you can start with a clean kernel. The following is a summary of those steps.

Download and install VirtualBox Download archlinux iso for attaching to VM. Create new VirtualBox VM (perhaps named arch2). Note it needs 16 GB of HD space, which is twice the default. Install archlinux in VirtualBox VM. Note the root partition should be 12 GB in size to give you room to update packages. Set up pacman, install packages. Set up filesystem, timezone and boot loader. Reboot.

Customize your bash shell (optional). Initialize networking, and configure the machine to it always starts networking on reboot. Optionally, install any other packages/commands you like to use.

Download the kernel source and compile in archlinux. I recommend turning off the option “Compile the kernel with warnings as errors” so you have less problems to fix. Set up kernel, module and ramdisk files. Make new grub bootloader. Reboot, and verify new kernel is running.

Create a DoNothing module

In this first exercise, you learn how to add compiled object code to the kernel while it is running. This type of code is called a module.

Change directory into your kernel source code (e.g. linux-6.14), and make a new folder to hold your work for this lab (e.g. lab7):

Command	Description
cd ~/linux-6.14	change directory to folder holding kernel source
mkdir lab7	make folder to hold work for this lab
cd lab7	change directory to this new folder

Download the file nothing.c from the course website:

```
wget https://cecas.clemson.edu/ahoover/ece468/labs/nothing.c
```

It contains the following code. There are two functions, the "init" function and the "exit" function. These are called when the module is loaded and unloaded, respectively.

```
#include <linux/module.h>       /* all modules need this*/
#include <linux/kernel.h>       /* KERN_ALERT */
#include <linux/init.h>         /* the macro definition */

static int donothing_init(void)
{
printk(KERN_ALERT "This is a DoNothing module\n");
return 0;
}

static void donothing_exit(void)
{
printk(KERN_ALERT "DoNothing module is unloaded\n");
}

module_init(donothing_init);    /* called on insmod command */
module_exit(donothing_exit);    /* called on rmmod cmomand */

MODULE_LICENSE("GPL");          /* required in latest kernel */
```

Create a Makefile.

```
vim Makefile
```

Add the line given below. This is the only content in the file. Save it.

```
obj-m := nothing.o
```

You will be recompiling the kernel multiple times in this lab. But this is not a full compile; you only need to (re)compile updated modules. The below line gives the command. Make note of it and consider simplifying how you execute it (for example, using "!make" in the shell to re-execute the last make command).

Command	Description
make -C ~/linux-6.14 M=$PWD modules	recompile kernel modules

If everything goes correctly, you will see something like the below:

```
~/linux-6.14/lab7> !ma
make -C ~/linux-6.14 M=$PWD modules
make: Entering directory '/root/linux-6.14'
make[1]: Entering directory '/root/linux-6.14/lab7'
  CC [M]       nothing.o
  MODPOST Module.symvers
  LD [M]       nothing.ko
make[1]: Leaving directory '/root/linux-6.14/lab7'
make: Leaving directory '/root/linux-6.14'
~/linux-6.14/lab7>
```

Next, insert the module into the running kernel:

```
~/linux-6.14/lab7> insmod nothing.ko
[######.####] This is a DoNothing module
~/linux-6.14/lab7>
```

List the modules currently in the kernel:

```
~/linux-6.14/lab7> lsmod
Module              Size    Used by
nothing             12288   0
```

```
~/linux-6.14/lab7>
```

Remove the module from the running kernel:

```
~/linux-6.14/lab7> rmmod nothing
[######.####] DoNothing module is unloaded
~/linux-6.14/lab7>
```

Use the dmesg command to view the most recent kernel messages. This example pipes the output to tail and limits the output to the 2 most recent messages.

```
~/linux-6.14/lab7> dmesg | tail -2
[######.####] This is a DoNothing module
[######.####] DoNothing module is unloaded
~/linux-6.14/lab7>
```

Create a onebyte device

In the second exercise, you create a 1-byte storage device, including device driver functions that can read and write that single byte.

Make a device file for the 1-byte storage device:

```
mknod –m666 /dev/onebyte c 61 0
```

The device filename is onebyte. All device files are typically stored in /dev on a linux system. The permissions are 666 (read/write for all/group/root). The device is a character (c) device, meaning the operating system will not buffer reads and writes. It uses a major number of 61 and a minor number of 0.

Do all the next work in the same folder you used for the DoNothing module. Download the file onebyte.c from the course website.

```
cd ~/linux-6.14/lab7
wget https://cecas.clemson.edu/ahoover/ece468/labs/nothing.c
```

It contains a lot of code. We will look at it piece by piece. First, it contains the same two functions as the last module, init and exit, which are called when the module is loaded and unloaded, respectively. In this case however these functions do a bit more work. The init function registers the device with the operating system using a major number (61), and a pointer to a structure of functions (described below). This lets the operating system know which functions to use when opening a device file with this major number. The init function also allocates 1 byte of memory to use as storage for our device. The exit function frees the memory, and unregisters the device functions.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
```

```
#define MAJOR_NUMBER 61

	/* the one-byte memory provided by this device */
char *onebyte_data = NULL;

static int onebyte_init(void)
{
int result;
	// register the device
result=register_chrdev(MAJOR_NUMBER, "onebyte", &onebyte_fops);
if (result < 0)
  return(result);	/* failed to find /dev/onebyte */
	// allocate one byte of memory for storage
	// kmalloc is just like malloc, the second parameter is
	// the type of memory to be allocated.
	// To release the memory allocated by kmalloc, use kfree.
onebyte_data=kmalloc(sizeof(char),GFP_KERNEL);
if (!onebyte_data)
  {
  onebyte_exit();	// cannot allocate memory
  return(-ENOMEM);	// return no memory error
  }
	// initialize the value to be X
*onebyte_data='X';
printk(KERN_ALERT "Onebyte device has been initialized\n");
return 0;
}

static void onebyte_exit(void)
{
if (onebyte_data)
  {	// free the memory and assign the pointer to NULL
  kfree(onebyte_data);
  onebyte_data=NULL;
  }
	// unregister the device
unregister_chrdev(MAJOR_NUMBER, "onebyte");
printk(KERN_ALERT "Onebyte device has been removed\n");
}

module_init(onebyte_init);
```

```
module_exit(onebyte_exit);

MODULE_LICENSE("GPL");
```

The below piece of code defines the device driver functions. The structure contains a pointer to each of the functions, and is used in the above code registering the device.

```
        /* function prototypes */
int onebyte_open(struct inode *inode, struct file *filep);
int onebyte_release(struct inode *inode, struct file *filep);
ssize_t onebyte_read(struct file *filep, char *buf,
                size_t count, loff_t *f_pos);
ssize_t onebyte_write(struct file *filep, const char *buf,
                size_t count, loff_t *f_pos);
static void onebyte_exit(void);

        /* file_operation structure */
struct file_operations onebyte_fops = {
        read: onebyte_read,
        write: onebyte_write,
        open: onebyte_open,
        release: onebyte_release
        };
```

The remaining code is the functions themselves. For the onebyte device we have 4 functions: open, release (close), read, and write. Open and release do basically nothing, because we assume the onebyte storage is always available. Read and write do the work of copying the 1 byte from the user space to the kernel space, or vice versa. Note that the functions maintain a file pointer position, but because we have only 1 byte of storage, this value is either 0 (beginning of file) or 1 (end of file).

```
int onebyte_open(struct inode *inode, struct file *filep)
{
return 0; // always successful
}

int onebyte_release(struct inode *inode, struct file *filep)
{
return 0; // always successful
}
```

```
        /* the read function for this device */
ssize_t onebyte_read(struct file *filep, char *buf,
                     size_t count, loff_t *f_pos)
{
if (*f_pos == 0)
  {
        // copy_to_user moves from kernel memory to user memory
  copy_to_user(buf, onebyte_data, 1);
        // advance the f_pos
  *f_pos += 1;
        // return 1, since the device only has 1 byte
  return(1);
  }
else    // no data on the device
  return(0);
}

        /* the write function for this device */
ssize_t onebyte_write(struct file *filep, const char *buf,
              size_t count, loff_t *f_pos)
{
if (*f_pos == 0)
  {
        // Use copy_from_user due to move into kernel memory
  copy_from_user(onebyte_data, buf, 1);
        // advance f_pos
  *f_pos+=1;
  return 1;
  }
else    // no more space to write
  return(-ENOSPC);
}
```

To compile this device driver, edit the Makefile and add onebyte.o:

```
vim Makefile
obj-m := nothing.o onebyte.o
```

Run the same make command given previously.

```
make -C ~/linux-6.14 M=$PWD modules
```

If everything goes correctly, you will see output like the first module, but this time for onebyte.o and onebyte.ko. Insert the module into the running kernel:

```
~/linux-6.14/lab7> insmod onebyte.ko
[######.####] Onebyte device has been initialized
~/linux-6.14/lab7>
```

Display the contents of the onebyte storage device:

```
~/linux-6.14/lab7> cat /dev/onebyte
X~/linux-6.14/lab7>
```

I have highlighted in yellow the expected output. Since there is only 1 byte, there is not even room for a newline. Use printf to write some values to the device, and cat to read it, confirming the following tests work:

```
~/linux-6.14/lab7> printf A > /dev/onebyte
~/linux-6.14/lab7> cat /dev/onebyte
A~/linux-6.14/lab7>
~/linux-6.14/lab7> printf B > /dev/onebyte
~/linux-6.14/lab7> cat /dev/onebyte
B~/linux-6.14/lab7>
~/linux-6.14/lab7> printf CDE > /dev/onebyte
-bash: printf: write error: No space left on device
~/linux-6.14/lab7> cat /dev/onebyte
C~/linux-6.14/lab7>
```

Insert the DoNothing module, and list all modules currently loaded. Note that you can have any number of modules loaded.

```
~/linux-6.14/lab7> insmod nothing.ko
[######.####] This is a DoNothing module
~/linux-6.14/lab7> lsmod
Module                  Size  Used by
nothing                12288  0
onebyte                12288  0
~/linux-6.14/lab7>
```

Create a keygen device

In this third exercise, write your own driver to make a keygen device. It should use /dev/keygen for access, same permissions and parameters as the onebyte device. Note it needs a new major number. The only thing the device does is return the 8-byte string "PASSKEY" on a read (the 8th byte is 0, signifying end of string). You can download the file testkey.c from the course website, compile it and run it to test your driver. You should see the below output.

```
~/linux-6.14/lab7> wget https://cecas.clemson.edu/ahoover/ece468/labs/testkey.c
~/linux-6.14/lab7> gcc –o testkey testkey.c
~/linux-6.14/lab7> ./testkey
i=8 key=PASSKEY
~/linux-6.14/lab7>
```

I recommend copying onebyte.c to keygen.c and editing to create the driver code. You do not need a write function at all. You do not need random storage, since you are only dealing with a constant output.

You will need to think about how to get 8 bytes from the device. Note that using cat to read the device can yield an endless stream of data, depending on how you write the driver, so you may need to debug with testkey.c.

You will need to edit the Makefile and add keygen.o, and use the same make command as previously.