Exercises for

Linux Device Drivers

v 2012.02

WARNING:

The order of the exercises does not always follow the same order of the explanations in the slides. When carrying out the exercises, carefully follow the exercise requirements. Read every exercise in its entirety before carrying out the instructions.

These exercises created by: Karim Yaghmour

Originals at: www.opersys.com/training/linux-device-drivers

Delivered and/or customized by:

IMPORTANT NOTES

- Many of the exercises below are open-ended. The criteria for "success" is therefore your own appreciation of how much your level of understanding of the studied topic has increased in carrying out the exercise.
- Some exercises build on previous exercises. Hence, **DO NOT** discard the work you do once you're done with an exercise.

Exercises Day 1

Chapter 1: On your host

1) Log in and install a guest Ubuntu VM.
2) Start the VM. You now have a fully functional Linux system running within a sandbox. Anything you in the guest will be limited to the guest. Hence, there is no danger of damaging your host system. Whenever we test modules and drivers, we will do so in the guest.

Chapter 2:

1) **On your host:** Go to a virtual console (CTRL+ALT+F2) and log in as root. Now do the following:

```
# cat /dev/input/mice
```

If you move your mouse, you should now see your mouse's input dumped to your standard output. This will display weird text. Typically, X window would have this device open. This is a good illustration of how the "everything is a file" philosophy in Unix. Once you're done testing this, log out and go back to your GUI (ALT+F7).

2) **On your host:** Given that everything is a file, including terminals, you can transfer data from one terminal to the next quite easily. To try it out, open two terminals, in the first terminal, do the following:

```
$ tty | tee /tmp/example-tty
```

Now, go to the second terminal and do the following:

```
$ echo "Hello from `tty`" > `cat /tmp/example-tty`
```

Now go back to the first terminal, you should be able to see the string dumped by the second terminal.

3) User-space applications that interact with or provide status reporting about hardware must often interact with specific filesystem entries to carry out their tasks. To get a better idea of how this works, match the following commands with the set of directories or files they use. There can be more than one directory or file being used, and there may be none.

Applications	Directories
`lspci`	`/proc/net/unix`
`lsusb`	`/etc/protocols`
`netstat -a`	`/lib/modules/`uname -r`/modules.dep`
`ifconfig`	`/sys/bus`
`modprobe`	`/proc/net/if_inet6`
`lsmod`	`/proc/ioports`
`insmod`	`/lib/modules/`uname -r`/modules.aliases`
`rmmod`	`/lib/modules/`uname -r`/kernel/...`
`depmod -e`	`/proc/interrupts`
`hdparm /dev/hda`	`/dev/hda`
`ps -e`	`/sys/bus/pci/devices/...`
`uname -a`	`/proc/cpuinfo`
	`/proc/bus/usb`
	`/proc/1/status`
	`/proc/net/dev`
	`/proc/net/tcp`
	`/sys/modules/...`
	`/proc/bus`
	`/proc/iomem`

```
/proc/modules
/proc/1/maps
/sys/bus/usb/devices/...
```

To check what filesystem accesses are carried out by an application try:

```
# strace application 2>&1 | grep ^open
```

Chapter 3:

1) Create a basic module with just an initialization function and an exit function. Have both functions print something out to the console using printk().
2) Create a makefile for building your module using the one on p.24 of LDD3 as an example.
3) Build your module.
4) Load your module, verify that it's been properly loaded, and unload it. Make sure that the printk() messages you inserted have indeed been printed, either directly on the console or through the dmesg command.
5) Use the "nm" utility on your module to check the symbols therein defined.
6) Add the MODULE_AUTHOR, MODULE_DESCRIPTION and MODULE_LICENSE statements to your basic module, and recompile it.
7) Use the modinfo utility to look at the information exported by your module to the kernel.
8) Try loading your module with MODULE_LICENSE set to "proprietary". Revert that to "GPL" for the rest of the exercises. (Have a look at /proc/sys/kernel/tainted).
9) Have your module recognize a string parameter. Modify your module so that the output it prints on load indicates when the parameter is absent, and prints the parameter as part of its output when it is provided.
10) Use the EXPORT_SYMBOL macro to export the function

```
void fct(){printk("fct\n");}
```

and EXPORT_SYMBOL_GPL statements to export

```
void gpl_fct(){printk("gpl_fct\n");}
```

from your module.
11) Create a simple add-on module that relies on the callbacks just defined being present by calling them in the init function. For now, keep this add-on's MODULE_LICENSE to "GPL".
12) Try loading the add-on both with the basic module absent and then present.
13) Modify your add-on module's MODULE_LICENSE to "proprietary" and try loading it.

Chapter 5:

1) Write a module that adds an entry in the /proc filesystem. Have the /proc read callback print out the value of "jiffies" and the PID of the current process.
2) Browse around /sys and compare the content to the notes shown in the slides. Notice how entries in /sys/bus/*/devices are all symbolic links back to /sys/devices/...
3) Write a module that registers a new type of system bus with the kernel's object model. Populate this new bus with two phony devices. Register one driver for each of these devices. These additions should result in entries being created in /sys/bus/drivers and /sys/devices. Compare the results of your additions to what is presented by existing buses and devices.

4) Write a hotplug script that reacts to the loading of one your phony devices. Check /sbin/hotplug, your script should be under /etc/hotplug/. Your script should write a messages with a timestamps to the system log (check logger).

Exercises Day 2

Chapter 6:
1) Write a module that creates two kernel threads running in parallel, see the kthread_run function (linux/kthread.h). The threads should be created at module load time, and killed at module unloading. The threads should typically have a while(1) that calls on msleep_interruptible() for periods of 1ms for one thread and 3 ms for the other thread. To terminate the kernel threads, use kill_proc function (linux/sched.h) and completions.
2) Have your driver declare two pointers to an int, and an actual int variable. At startup, your module should set one of the pointers to the variable's address and the other to NULL. Also, at startup, your module should set the integer to zero. Modify your threads so that they grab two locks (initially implement using semaphores) and then proceed to (this isn't an optimal algorithm, but it proves the point):

1. Test which one of the pointers is NULL
2. Sets that pointer to the address in the other pointer
3. Sets the other pointer to NULL
4. Uses the pointer to which the address was assigned in #2 to increment the value of the integer.

One of the threads should sleep for 5ms in between grabbing both locks and the other thread should sleep for 5ms in between releasing both locks. The delays inserted in part 1 should remain, but must be outside any lock-grabbing code. In order to be able to safely remove this module, you will need to use kthread_stop() and kthread_should_stop(). Have a look at the comments in linux/kthread.h for explanations on how to use these functions.
3) Modify the module created above to use spinlocks instead of semaphores. You will need to determine which type of spinlock is appropriate. You may need to modify the threads' functionality so that they can use spinlocks.

Chapter 7:
Write a module that registers a shared callback for the interrupt used by the network driver. Having been loaded, your callback will be invoked every time there is traffic on the wire. The callback should tell the OS that it doesn't handle the interrupt so that Linux feeds it to the network driver. Write the handler so that every 200 interrupts, it prints out a message using printk(). For this exercise, you'll need to use request_irq and free_irq.

Chapter 8:
1) Use TIMER_INITIALIZER, add_timer and del_timer to add a timer-activated, periodic callback to the module you wrote in chapter 6 and have it do the same accesses to the shared pointers. Modify the locking mechanisms appropriately.
2) Modify the module developed in chapter 7 so that it triggers a tasklet for every interrupt it receives. Have the tasklet use printk() every 2 seconds to print out the number of interrupts that occurred.

Chapter 9: Difficult / Assumes knowledge of how to implement circular buffers

1) Create a module that exports primitives to other modules for:

1. Allocating memory areas
2. Deleting allocated memory areas
3. Atomically writing to an allocated memory area
4. Atomically reading to an allocated memory area

For 3 and 4 you will need to maintain read and write pointers. Start by implementing your module so that it can hand out a maximum of 128kb (use kmalloc). Here are some hints:

- Each allocate area should be implemented as a circular buffer with regards to reading and writing.
- Use spin_lock_irqsave() and spin_lock_irqrestore()
- Use only one “read” pointer and only one “write” pointer for each allocated memory area.

2) Modify the module developed in chapter 6 and extended in chapter 8 so that it uses the services of the module just implemented to log any access (a timestamp) to the shared integer value into an allocated buffer.

3) Modify the module implemented in exercise 1 of this chapter so that it can now serve buffers up to 2MB in size and provide an interface from /proc for reading the content of the buffer. To simplify implementation, assume that the /proc callback only reads 4KB at a time.

Exercises Day 3

Chapter 11:
1) Write a char driver that implements the read/write/open/release methods on top of the mechanism implemented in chapter 9. You will need to create appropriate entries in /dev for applications to be able to access your "device". Having created those entries, you should be able to use the "cat" and "echo" commands to read and write, respectively, from and to the device. An "echo" should result in your driver recording what's being fed by the "echo" command, and a "cat" should result in the reading of what had been previously fed into it. For this part, assume that reading from an empty buffer results in telling user-space that the file is empty.
2) Modify the mechanism implemented in chapter 9 so that it allows blocking I/O. Specifically, make it so that when the user-space does a read on the character device, it will wait if the device is empty until something gets written to it.

Chapter 12:
Modify the driver implemented in chapter 11 so that it provides a block device interface for accessing the buffer. Add the appropriate /dev entry for the block device (mknod), and use a user-space application (fdisk) to read/write to the device.

Chapter 13:
1) Modify the driver implemented in chapter 11 and extended in chapter 12 to also provide an Ethernet network interface. In effect, any application reading from a socket attached to this device should receive input as a result of any writing to the shared buffer. You will need to use ifconfig to "configure" your newly created device.
2) Write a user application that opens a socket, binds to the network interface and reads input from it.
3) Modify the module that was developed and extended in chapters 6, 8, and 9 so it feeds its log writing into the shared buffer, therefore automatically generating "network traffic".
4) Modify the module that was developed and extended in chapters 7 and 8 so that it too feeds content into the shared buffer as a result of the interrupts it receives.