Operating Systems

WS 2014/2015

Lab 2 – Linux kernel modules and device drivers

Submission deadline: December 9, 2014

Testing: December 12, 2014

Form groups of 2 to 3 students!

Send solution to: os-lab@deeds.informatik.tu-darmstadt.de

Preface

The purpose of this lab is to get familiar with important kernel data structures and their use, to find and browse the source code of the Linux kernel, and to use the **`/proc`** file system to read and interpret certain kernel variables/structures. Furthermore, the lab introduces the concept of *kernel modules*. You will learn how to compile, install and test kernel modules as well as how to write a simple device driver. However, the overall goal is to introduce the tools needed to perform the subsequent lab on Linux kernel module implementation.

If you do the lab on your own machine, make sure that you use a recent Linux distribution that includes recent versions of the Linux kernel and the required build tools. Most common distributions will allow you to make automatic updates to the latest version. The examples provided in the lab were all tested and verified on the latest stable version of the 3.16 kernel. Do not use 2.6.x or earlier kernel versions, as some of the mechanisms and interfaces presented in the labs may not be present or work differently with old Linux versions. This will become even more important for the third lab.

For most Linux distributions, you have to install additional packages to enable the building of kernel modules. The required packages may vary between distributions. Usually, you need to **install at least the Linux headers package** for your running kernel. On some systems, you may even have to install the Linux source package for your running kernel. Check your package manager to find out how to install the required packages (like *Apt* in Debian/Ubuntu or *Pacman* in Arch Linux). Check the documentation of your Linux distribution if installing the headers (and maybe the sources) is not enough for building kernel modules.

Make sure to send your solution of the lab via email to the address mentioned on the cover page before the deadline. You have to **work in groups of 2 to 3 people**, do **refrain from submitting solutions individually!** In your submission, **include the names, matriculation numbers and e-mail addresses of ALL group members.** Currently, we target to test your solutions to the lab problems on December 12 in seminar room A213. In your submission mail, **indicate possible collisions on that day with an exact timeslot**, so we can schedule you accordingly – elsewise you have to accept our assignment. During the test we will discuss details of your solutions with you to verify that you are the original author and understand your own code well. In case of updates, information will be made available during the lecture or exercise, as well as on the course web page.

Questions regarding the lab can be sent to:

os-lab@deeds.informatik.tu-darmstadt.de

Good Luck!

Goal

The focus of this lab is on understanding Linux loadable kernel modules (LKMs), device drivers and their organization. After finishing the lab, you are able to write simple device drivers for Linux. This lab will also help you to implement some of the concepts and approaches discussed in the lecture, thereby improving your understanding of those.

Kernel Introduction

The Linux kernel is a collection of data structures, instances, and functions. The collective data structures define the kernel's view of the current state of the computer system, which changes when different events occur, e.g., a system call or an IRQ.

In general, the data structures and functions are defined in C header files in the Linux source code tree. The source code of the Linux kernel can be found in the directory **`/usr/src/linux`** (or maybe e.g. **`/usr/src/linux-3.16.xx`**) if you installed the Linux sources. Most of the C header files can be found in subdirectories of **`<source-dir>/include/`** which separate the definitions of different device types (e.g. **`scsi/`**, **`sound/`**, **`video/)`** and the general definitions (**`linux/)`** which are machine-independent. Note that the header files on your system may reside in another location. However, **`/usr/src`** is always a good start point for finding the headers. Also note that if you installed the Linux headers but not the sources, the source tree usually only contains the header files in the **`include`** folder but no other C files.

Kernel Modules

A Linux kernel module is a collection of functions and data types that are compiled as an object file to extend the running kernel. Typically the modules contain code to enhance the kernel's abilities of handling hardware and file systems. The advantage of kernel modules is the possibility to add and remove modules to/from the kernel during runtime as needed.

A module C file has the following structure:

```
#include <linux/module.h>
#include <linux/init.h>
MODULE_LICENSE("GPL");

static int __init name_of_initialization_routine(void) {
    /* code goes here */
    return 0;
}
static void __exit name_of_cleanup_routine(void) {
    /* code goes here */
}
module_init(name_of_initialization_routine);
module_exit(name_of_cleanup_routine);
```

The module initialization routine, which is declared with the **`module_init()`** macro, initializes the module's variables and data types. The cleanup routine, which is declared with the **`module_exit()`** macro, frees the memory that is used by the module's variables. Since kernel version 2.6 the **`MODULE_LICENSE`** macro defines the license of the module. A module must declare to be GPL-licensed in order to be able to use all kernel interfaces, which are often GPL-licensed and, therefore, only allow other GPL-licensed code to use them.

Let's name the file above **`test.c`**. To compile this file, we use the **`make`** command. The required actions for building the module are specified in a **`Makefile`**, which is placed in the same directory as the source code files. The following example will work for Linux 3.x; check the documentation for your kernel to find out how it works on your machine. The documentation is usually in **`/usr/src/linux/Documentation/kbuild/modules.txt`**.

Example of a **`Makefile`**:

```
obj-m := test.o
PWD := $(shell pwd)

default:
	make -C /lib/modules/$(shell uname -r)/build SUBDIRS=$(PWD) modules
```

Two lines may need to be edited. In the first line, the output-file is defined, which in this case is **`test.o`**. Your modules should have more meaningful names; so, you have to change this line accordingly. The second line to be edited is the actual command that is executed (**`make -C`** …). In this line the path to the Linux kernel build code possibly needs to be adapted to the used computer system.
By using the Makefile shown above, the module can be compiled by simply executing "make". After a successful compilation one can find a file called **`test.ko`** in the same directory that contains the **`test.c`** file. The module can now be installed by using "**`insmod test.ko`**", which results in the module initialization routine being executed. One can uninstall the module using "**`rmmod`**", which results in the module cleanup routine being executed. Note that the **`insmod`** and **`rmmod`** commands usually require root privileges. Another interesting related command is **`lsmod`**, which lists all modules that are currently loaded.

Device Driver Introduction

In order to enable an OS to utilize the available hardware devices, the OS kernel relies on device drivers, i.e., software components that mediate between physical devices and the kernel. Each physical device is associated with a device driver. The driver provides a specified interface that allows the kernel to interact with the device and control it. Furthermore, the kernel uses the provided interface to expose the devices and their functions to user-space programs, e.g., through the file system interface.

There are two main strategies that device drivers can use:

- **Polling:** A device driver that uses polling is constantly requesting input values from a device until it completes its operation. Hence, a user-space application enters the kernel and begins executing the device driver. During the usage of the device the corresponding task is periodically polling the device's status register to determine when the operation has completed.
- **Interrupt:** An interrupt-driven device driver starts the device operation and suspends itself until an IRQ (Interrupt ReQuest) is raised by the device which then has completed its operations. When an IRQ occurs the sleeping task is awaked and resumes user-space processing.

Furthermore, Linux differentiates between block and character devices. Block devices transfer multiple bytes from/to the primary memory cache on each hardware operation, whereas character devices transfer information byte-by-byte without using caching. Typically, Linux device drivers are implemented as kernel modules (see previous section), which are loaded on-demand during runtime.

Device Driver Organization

A device driver contains a collection of functions and data structures that provide a simple interface for managing the device. This interface is used by the kernel to request the driver to manipulate its device for I/O operations. The interface to a device is intended to look the same as an interface to the file system.

The Linux kernel references the device driver by a major and a minor number. The major number normally defines the class of the device, e.g., hard disk, floppy disk. The file **`include/uapi/linux/major.h`** (relative to the path of Linux kernel sources) provides a full list of already defined major numbers. The minor number is an 8-bit number that references a specific device of a particular class (major number). Thus, two floppy disks on a machine would have a common major number, one floppy disk having minor number 0 and the second having minor number 1.
The kernel must be informed about the device's existence. When Linux is booted, the kernel adds files for each device in the system to the directory **`/dev`**. One can add those files manually from the command line using:

```
mknod /dev/<dev_name> <type> <major_number> <minor_number>
```

`<dev_name>` can be freely chosen by the user. It is only restricted by the already existing files which will not be overwritten. The **`<type>`** parameter can be either **c** for a character device or **b** for a block device. The optional **`–m`** switch lets you set the permissions on the new device file. Note that this command usually requires root privileges.

For every device driver there is the possibility to define several operations. Since a device driver works like an interface to the file system, the operations that can be defined are declared by the **`file_operations struct`**, which can be found in **`/include/linux/fs.h`**.

Only those interface functions that are actually needed by the driver need to be defined. For example, an input-only device does not need a **`write()`**-function, whereas an output-only device does not need a **`read()`**-function. The needed functions for this lab are already included in the source skeleton (*gen_module.c*) that you can download from TUCaN.
When a device driver module is being installed, the initialization routine is executed to initialize the driver. Once the initialization is completed, the kernel can route a system call such as

```
open(/dev/<device>, O_RDONLY);
```

to the **`<device>_open()`** function in the driver (*gen_module_open()* in the source skeleton).

Loadable Kernel Module Drivers

Device drivers can be written in form of loadable kernel modules, which have been introduced previously. A driver that is implemented as a module can be loaded or unloaded at any time while the machine is executing. In contrast, device drivers that are not written as loadable kernel modules are being loaded only at boot time. A loadable kernel module driver can be loaded and unloaded like any other LKM.

Have a look at the code skeleton accompanying this lab description. The skeleton code contains all basic functions that are commonly needed for a device driver module.

More Information

You can get a lot of information from the web, e.g., try Google. **Be aware that the information you get is for specific kernel versions. There are substantial and important differences between the kernel versions! Not all information from the web may apply to your system!**

For more information on how to write a driver, check "Linux Device Drivers" by J. Corbet et al: http://lwn.net/Kernel/LDD3/ Beware that some of the information in this book is outdated, as it was written with a Linux 2.6.x kernel in mind, (cf. problem statements below), but can still be considered one of the better resources.

Also use the manual pages on your system. Try for instance "**`man rmmod`**" to get information on the **`rmmod`** command. Most system calls and POSIX library functions also have man pages.

Last but not least, you can also browse the respective kernel include files for code documentation and function declarations.

ProcFS

Problem 1

Design and construct a module that implements a clock in **`/proc`**. The file should be located at **`/proc/clock`** and return a formatted output (e.g. "Nov 24, 2014 13:33:37") of the **`do_gettimeofday(…)`** function. The definition of this function can be found in the file **`time.h`** (located in **`/usr/src/linux/include/linux`**), which needs to be included.

Attention: Linux recently deprecated support for the **`create_proc_entry`** function! Use the function **`proc_create`** instead! The definition of this function can be found in the file **`proc_fs.h`** (located in **`/usr/src/linux/include/linux`**), which needs to be included.

Ensure that your module builds on a recent kernel (i.e., Linux 3.x)! Test your module by using "**`cat /proc/clock`**".

Device Drivers

Problem 2

Design and implement a *character device driver* for a virtual FIFO device. Recall that a FIFO is equivalent to a pipe: one process can write into the FIFO, and another can read from it. For this task we assume that we have only one reader/writer at a time. The first character written to the FIFO is the first character returned by a read operation. Your driver should implement 2 FIFO queues by implementing 4 minor device numbers. Even-numbered devices are write-only devices, and odd-numbered are read-only devices. Characters written to device i are readable from device i+1.

Your driver should be a loadable device driver, that is, it should be implemented as a module. Use the major device number 240. Use **`mknod`** to create the special files in the **`/dev`** directory (e.g. **`/dev/fifo0`**). You remove devices with the normal **`rm`** command. **`mknod`** and **`rm`** normally requires root permissions, so you may have to use the **`sudo`** command (note that you also have to give the right permissions using the –m switch):

```
sudo mknod –m a+rwx /dev/fifo0 ...
```

You should remove your fifos afterwards, using **`rm`**:

```
sudo rm /dev/fifo*
```

The FIFO should have a fixed size (for instance 10) defined at compile time. Optionally you can improve your solution to dynamically allocate buffer memory as needed (up to some fixed limit); however, this is not a requirement.

Reading from the FIFO should return all the bytes currently in the FIFO. If the FIFO is empty and there is no writer having the FIFO opened, it should return EOF. If there is a writer having the FIFO opened, the reader should block waiting for the writer to write something.

Writing to the FIFO must wake up any blocked readers. Also closing the FIFO must wake up any blocked readers, to allow them to read the EOF.

Writing to a FIFO device is only possible if the queue of the device is not full. Otherwise, the FIFO device blocks and tries to wake up a reader. It returns as soon as it is able to write all bytes. It is ok to start writing the first bytes even though all of them will not fit in the FIFO.

Your driver should be implemented using the "interrupt-driven" approach. In this case, since we are not using any hardware device, you will not implement it using interrupts. In this case we use a sleeping/waking up approach instead. Your FIFO should not use polling to resolve race conditions, but sleep and wake up instead (cf. Linux Device Drivers 3rd edition[1], Chapter 6, *Blocking I/O*).

Example:

Consider a case with 2 FIFO devices:

- **`/dev/fifo0`** is a write-only device
- **`/dev/fifo1`** is a read-only device

We can now write some characters into /dev/fifo0:

```
pc:/# echo test > /dev/fifo0
pc:/#
```

Reading of these characters is then possible by doing:

```
pc:/# cat /dev/fifo1
test
pc:/#
```

Problem 3

Develop a test application in C, so that you can test all the special cases (opened/closed FIFOs, empty/full etc).

[1] http://lwn.net/Kernel/LDD3/