Operating Systems

WiSe 2015/2016

— Lab Assignment 2 —

Linux Loadable Kernel Modules:
ProcFS and FIFO Character Device Driver

— Submission Email —

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

— Course Website —

www.deeds.informatik.tu-darmstadt.de/teaching/courses/ws20152016/operating-systems/news/

Publication:	Mon, Nov 16, 2015
Submission:	Tue, Dec 8, 2015 before 23:59
Testing:	Fri, Dec 11, 2015 (room A213)

Preface

The purpose of this lab is to get familiar with the Linux kernel, its build system, and loadable kernel modules (LKMs). You will learn to find and browse the Linux kernel sources and to use the `/proc` file system from both the user's and the system programmer's point of view. Moreover, you will learn how to develop and build loadable kernel modules as well as how to write a simple device driver.

This lab contains multiple sections covering the different topics. Each numbered section contains both programming tasks and questions that have to be answered. Although the programming tasks may build upon each other, you have to clearly separate them is your solution by providing distinct source files for each programming task. You have to use `make` for compiling your code, i.e., you need to *supply appropriate Makefiles* for that purpose. After compiling your code, we want to have one separate executable or LKM that implements your solution for each programming task. For answering the stated questions, create a *plain text file* and write down your answers. *Do not use other file formats such as Office formats or PDF!* ***Make sure that every group member has detailed knowledge of the whole lab solution! We except that every group member can answer questions related to all tasks.***

You should do the lab on your own machine. Since you will work on the kernel level, we strongly recommend doing all the programming tasks *inside a virtual machine* in order not to damage your machine. Make sure that you use a recent Linux distribution that includes recent (but not necessarily the latest) versions of the Linux kernel and the required build tools. Most common distributions allow for easy installation and upgrade/downgrade of the required tools. The documentation and examples provided in this lab were tested and verified in our testing environment (Debian 8.2) using a *Linux 3.16* kernel, *GCC 4.9*, and *GNU Make 4.0*. We recommend that you use a similar setup to ensure that your solutions work properly in our testing environment. ***If you use different kernel or tool versions, it is your responsibility to check for compatibility with our testing environment!***

Make sure to send your lab solution as compressed tar archive via email to the address stated on the cover page before the deadline. You have to work in groups of 4 to 5 students. Do not submit individual solutions but send one email per group! ***In your submission email, include the names, matriculation numbers and email addresses of ALL group members.*** The testing date (see cover page) is mandatory for all group members. Therefore, ***indicate collisions with other TU courses on the testing date*** so that we can assign you a suitable time slot. Assigned time slots are fixed and cannot be changed. Students that do not appear for the assigned testing time, will get no bonus points for that lab. During the test, we discuss details of your solutions with you to verify that you are the original authors and have a good understanding of your code. The amount of bonus points you achieve depends on your performance in the testing session.

Please regularly check the course website as we publish up-to-date information and further updates there.

Questions regarding the lab can be sent to: os-lab@deeds.informatik.tu-darmstadt.de

Good Luck!

General Advice

For the programming tasks, you have to use the *C programming language*, the *GCC compiler*, and the *Make* build tool. We expect that you not only provide a *working and robust solution* to each of the tasks, but also adhere to *a consistent, structured, clean, and documented coding style*. ***Keep your code simple and as close to the tasks as possible. Do not implement more than what is asked. Do not use kernel mechanisms, subsystem and interfaces, for instance, for synchronization purposes, that we do not hint at.***

For all programming tasks, you have to *implement proper error handling* mechanisms in your solution. Function calls can fail and their failure must be handled properly. The failure of a function call is usually indicated by its return value being set to a magic value (e.g. `-1` or `NULL`). Proper error handling and cleanup in error cases is especially important for kernel code since improper error handling can have severe effects on your system. Please note that you cannot use the same approaches for in-kernel error handling as you would do for user-space programs, e.g., printing to `stderr` and calling `exit()` does not work inside the kernel.

Make sure that *your* code compiles without any warnings by using the `-Wall` compiler switch during compilation. If your code produces warnings during the testing session, we expect that you are able to explain each and every warning and justify why you did not fix it.

Linux systems provide an extensive online manual, the so-called man pages. Please make extensive use of this feature when you work on the tasks of this lab as we only provide a very brief description of some important functions and tools. You can access the man pages with the following command line:

```
man [section] <command-name>
```

Alternatively, there are also online versions of the man pages available, e.g., at

https://www.kernel.org/doc/man-pages

But be aware that online versions may not be complete compared to a local installation.

Section 3 refers to C library functions, section 2 to system calls, and section 9 to general kernel routines. These three should be the most interesting sections for you. The usage of section numbers is optional, but may be necessary when identical command names exist in different sections. Also, do not hesitate to search for additional information on the web or at the ULB Darmstadt.

For building your solution for each task, you have to use the `make` tool and write appropriate Makefiles instead of directly invoking the compiler. For building Linux kernel modules, your Makefiles need to follow a specific structure, which is explained below. You may choose to write a separate Makefile for each programming task, or you may write one Makefile that can be used for building all the solutions. If you do not know how to use the make tool, have a look at its man pages. You may also have a look at online tutorials such as the following ones.

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

http://mrbook.org/blog/tutorials/make/

Linux Kernel Basics

The Linux kernel, being a monolithic kernel, is a collection of data structures, instances, and functions. The collective data structures during runtime define the kernel’s view of the current state of the computer system, which changes when different events occur, e.g., system calls or interrupts. In the following, we will have a look at the basics of the Linux kernel and explain how to extend its features.

The Kernel Sources

The Linux kernel is mostly written in C and its source code is organized in a large directory tree, which is usually just called the Linux source tree. You can have a look at the complete source tree online. A very convenient way to browse, search, and read the sources is by using one of the available indexing sites, for instance (for Linux 3.16):

http://lxr.free-electrons.com/source/?v=3.16 or

https://lxr.missinglinkelectronics.com/#linux+v3.16

Similar to most other C projects, the Linux sources can be broadly divided into header files (`*.h`), where data structures and functions are declared, and C files (`*.c`), where functions are implemented. In the context of this lab, we are mostly interested in the header files as they define the interfaces that we can use to interact with the kernel if we want to extend its functionality. We will see later in this lab that we need these header files in order to build our own loadable kernel modules. Most Linux distributions provide special packages for installing the Linux header files for the installed kernel version. An easy method for finding out the currently running kernel version is by using the `uname` command. Have a look at its help output for details.

Once you installed the appropriate kernel headers, they should be located in a subdirectory of `/usr/src/` (e.g. `/usr/src/linux-headers-3.16.xx`). Note that the location can vary depending on your Linux distribution. Most of the C header files can be found in subdirectories of `<source-dir>/include/` which separate the headers for different device types (e.g. `scsi/`, `sound/`, `video/`) and the general headers (`linux/`) which are machine-independent. Note that you can also install the complete source tree for your kernel instead of only the headers. However, in the context of this lab, we only need the header files installed.

Loadable Kernel Modules

A Linux loadable kernel module (LKM) is a collection of functions and data types that are compiled as a separate object file to extend the running kernel. Typically, modules contain code to enhance the kernel’s abilities to handle 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, i.e., kernel functionality can be added/removed on demand during runtime, for instance, when hardware devices such as USB sticks are added or removed from the system.

A module C file has the following basic structure:

```
/* Linux kernel headers */
#include <linux/module.h>
#include <linux/init.h>

MODULE_LICENSE("GPL"); /* code license for the module */
MODULE_AUTHOR("Name"); /* name of module author */

/* module initialization, first function to be called */
static int __init name_of_initialization_routine(void) {
    ...
    return 0; /* or some error code */
}

/* module cleanup, last function to be called */
static void __exit name_of_cleanup_routine(void) {
    ...
}

/* declare init/exit functions for the kernel */
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 internal variables and informs the kernel about the module's functionality (registration). The cleanup routine, which is declared with the `module_exit()` macro, frees memory and informs the kernel that the module is no longer available (de-registration). Since kernel version 2.6, the `MODULE_LICENSE` macro defines the license of the module. Only a GPL-licensed module is allowed to use all kernel interfaces, which are often GPL-licensed and, therefore, only allow other GPL-licensed code to use them. The `MODULE_AUTHOR` macro, although optional, should be used to declare the name of the module author.

Let's name the basic module example from above `test.c`. To compile this file and to build a LKM from it, we use the `make` command and a Makefile that invokes the kernel build system (kbuild). The Makefile must be located in the same directory as the module C file (`test.c` in our example). The following example Makefile works for Linux 3.x; check the documentation for your kernel in case the below example does not work for you. Generally, `Documentation/kbuild/modules.txt` (within the source tree, see online resources above) provides a good overview of kernel modules and describes how to build them out-of-tree (in a separate folder rather than from within the kernel source tree).

Example of a Makefile:

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

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

Two lines may need to be edited in our example Makefile. In the first line, the output file is defined, which in this case is `test.o` (the kernel module file will be called `test.ko`) as our source file is called `test.c`. 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. In this line, the path to the Linux kernel build code may need to be adapted to the used computer system. Note that pointing to the kernel headers directly does not work here. Please make sure that you do not hardcode local paths from your machine here as this will not work on our test machines.

By using the Makefile shown above, the module can be compiled by simply executing `make`. After a successful compilation, `test.ko` can be found in the same directory that contains `test.c`. The module can now be loaded with `insmod test.ko`, which results in the module initialization routine being executed. The module can be removed with `rmmod test.ko`, which results in the module cleanup routine being executed. Note that the `insmod` and `rmmod` commands require root privileges. Another interesting related command is `lsmod`, which lists all modules that are currently loaded. The command `modinfo` is also quite useful to get general information about a LKM. Have a look at the man pages for more details.

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 general strategies that device drivers may employ:

- **Polling**: A device driver that uses polling is periodically requesting input values from a device until it completes its operation. For instance, a driver actively polls the device's status register inside a loop to determine when a requested operation has completed or to check whether new data is available.
- **Interrupt**: An interrupt-driven device driver starts a device operation and suspends itself or continues with other work until an IRQ (Interrupt ReQuest) is raised by the device signaling the completion of the requested operation. Instead of actively asking the device for state changes or data, the driver only needs to react on IRQs.

Furthermore, Linux differentiates between block and character devices. The kernel (driver) communicates with character devices by sending and receiving single characters (bytes) at a time and the device usually only supports sequential-access. Examples for character devices include serial ports, parallel ports, tape drives, terminals, and sound cards. The kernel (driver) communicates with block devices by sending and receiving blocks of data (data junks of several bytes) at a time and the device usually supports random-access. Examples for block devices include hard disks, DVD drives, and USB sticks. Irrespective of the device type, Linux device drivers are typically implemented as loadable kernel modules (see above), which are loaded on-demand during runtime.

Device Driver Organization

A device driver contains a collection of functions and data structures that provide the interface for managing a device. This interface is used by the kernel to request the driver to manipulate its device for I/O operations. From a user's point of view, the interface to devices is intended to look the same as the interface to a file system, i.e., all devices are represented as file objects (also called device nodes), for instance, within the `/dev` directory. To interact with a device, a user must open the corresponding files object and perform the desired operations on that opened file object.

The Linux kernel references device drivers by a major and a minor number. The major number defines the class of the device, e.g., hard disk, floppy disk. The file `include/uapi/linux/major.h` (in the 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. For instance, two floppy disks (major number 2) in a machine would have a common major number, but one floppy disk would have minor number 0 and the other minor number 1.

In order to make devices accessible from user-space, the file objects (device nodes) in `/dev` must be created and associated with the corresponding major/minor pairs. In modern Linux systems, this is usually done dynamically upon device connection by the udev system. However, device nodes can also be created manually (statically) from the command line using:

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

`<dev_name>` can be freely chosen by the user. The `<type>` parameter can be either **`c`** for a character device or **`b`** for a block device. The optional `-m` switch can be used to set the permissions on the new device file. Note that this command requires root privileges. See `man 1 mknod` for further details.

For every device driver, several operations should be defined. Since a device driver works like an interface to the file system, the operations that can be defined are declared using the `file_operations` struct, which is defined in `/include/linux/fs.h`. However, only those functions that are actually needed by the driver have 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. While a device driver module is being loaded, its initialization routine gets executed to initialize the driver and register the supported functions with the kernel. 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. Other system calls such as `read` or `write` are also handled in a similar manner.

Loadable Kernel Module Drivers

Basically all device drivers in Linux are implemented as loadable kernel modules as introduced above. Usually, the loading and unloading of drivers are handled fully automatically when devices are connected and disconnected. However, drivers can also be loaded/unloaded manually by means of `insmod`/`rmmod`.

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, available online at http://lwn.net/Kernel/LDD3. Beware that the book was written for Linux 2.6.x; hence, some of the information is outdated. Nonetheless, the book can still be considered one of the better resources.

For further details on individual subsystems, interfaces, and functions of the Linux kernel, it is always worth having a look into the `Documentation/` folder and the corresponding kernel sources (see online resources above).

1) Linux ProcFS

Linux uses a virtual file system that is mounted as `/proc` for providing a rich interface to system internals. The files in `/proc` are not actual files that are stored on a storage device, but they are virtual files provided programmatically by the kernel and its loaded modules. By reading and writing from/to files in `/proc`, a user can view and change system properties and configuration options (although most files are read-only). Please have a look at `man 5 proc` for a detailed explanation and an overview of the information that can be found in `/proc`.

Every kernel module can add its own virtual files to `/proc`. The kernel provides an interface for managing ProcFS entries in `include/linux/proc_fs.h`, which needs to be included by a module that wants to use it. The most interesting functions for us are the ones for creating and removing ProcFS files.

```
/* create a new ProcFS file */
struct proc_dir_entry *proc_create(
  const char *name, umode_t mode, struct proc_dir_entry *parent,
  const struct file_operations *proc_fops);

/* remove an existing ProcFS entry */
void proc_remove(struct proc_dir_entry *entry);
```

To create a new file, use `proc_create`, `name` is the file name, `mode` is the file access mode, `parent` is the parent directory (can be `NULL` if no parent needed), and `proc_fops` point to the `file_operations` struct that implements the file operations needed such as open, close, read, write, etc.; the returned `proc_dir_entry` pointer needs to be stored to allow for later removal of the created file. To remove a previously created file, use `proc_remove` where `entry` is the pointer to the `proc_dir_entry` returned by `proc_create`.

The kernel also provides functions for accessing and manipulating date and time that are very similar to the user-space functions you know. Have a look at `include/linux/time.h` to see which functions the kernel offers, for instance, `do_gettimeofday` retrieves the current time as `timeval` struct.

Task 1.1

Design a loadable kernel module that provides a virtual read-only file called `deeds_clock` in `/proc` (`/proc/deeds_clock`). The virtual file has to contain the current system time in seconds whenever read. Use the following formatting: "`current time: %llu seconds\n`" (`%llu` is the correct format code for printing `unsigned long long` values). Make sure that the file operations you implement return the correct error codes to ensure that tools like `cat`, `head`, or `less` work properly. For instance, have a look at `include/uapi/asm-generic/errno-base.h` and `include/uapi/asm-generic/errno.h`.

Make use of the kernel's printing/logging facilities, i.e., use the `printk` function with an appropriate logging level (e.g. `KERN_INFO` or `KERN_NOTICE`). Log appropriate messages upon module loading and unloading. Do not forget the newline at the end of each log message. The `dmesg` tool can be used to read the kernel logs, see `man 1 dmesg` for details.

You may use the provided generic module skeleton code as a starting point. However, make sure to properly rename the file and the included functions to your use case. Note that not all functions in the skeleton code may be needed; remove the unneeded code.

> You may want to have a look at `include/linux/seq_file.h` for a more convenient higher level interface for file operations. Also, the Linux kernel has a large library of string functions similar to those you know from user-space. You should have a look at the printf utilities in `include/linux/kernel.h`.

Task 1.2

> Extend your module from task 1.1. Provide another virtual read-write file called `deeds_clock_config`. The virtual file should allow to configure the output formatting of `deeds_clock`. When a "`0`" is written to `deeds_clock_config`, the output format for `deeds_clock` should be as specified in task 1.1; however, if a "`1`" is written, the output format should be changed to something prettier. Use the following format for that case: "`current time: yyyy-mm-dd hh:MM:sec`". An example output looks as follows: "`current time: 2015-12-08 23:59:59`".
>
> When `deeds_clock_config` is read, the output has to depict the current clock output configuration. Use the following format for that: "`current clock format: %d\n`" where "`%d`" is replaced by the currently active option (`0` or `1`). Make sure to implement appropriate error handling, for instance, a user may try to write other values or huge amounts of junk to the file, which should be rejected.

Question 1.1

> Load your module from task 1.2 and perform tests with your clock implementation including the configuration options. What behavior are you expecting and what are you observing? What happens if you read the clock file multiple times in a loop while concurrently changing the output formatting option?

2) FIFO Device Driver

A FIFO is a buffer of a certain size with a read and a write end, similar to a pipe. All bytes written to the FIFO are read from it in the same order they were written (first in, first out). A FIFO can be implemented as a circular byte buffer similar to the example you know from the lecture. In the context of this lab, we assume a simplified FIFO model in which the FIFO can be written to and read from by only one process at a time, i.e., the write end can only be opened and written to if the read end is closed and the read end can only be opened and read from if the write end is closed. Any attempt to access the FIFO while another access is already in progress results in an error.

Reading from the FIFO returns all the bytes currently stored in the FIFO. If the FIFO is empty it returns `EOF`. Writing to the FIFO is only possible if it has enough space left. An attempt to write to a full FIFO fails with an error code. Attempting to write more data to the FIFO than it has free space also fails with an error code without writing any data.

The described simple FIFO model is non-blocking and does not rely on synchronization mechanisms such as semaphores. This works since there is always only one process that can access the FIFO and the two interesting cases (empty/full buffer) are handled by returning an appropriate error code to the accessing process.

Task 2.1

> Design and implement a *character device driver* (as kernel module) for a virtual FIFO device that implements the above described simple FIFO behavior. Your driver has to provide one

FIFO queue that is accessed via two distinct device nodes `fifo0` and `fifo1` in `/dev`, i.e., your driver must support two minor numbers where minor number `0` is a write-only and minor number `1` is a read-only device. Use the major device number `240` for your driver. You may either use the `mknod` command, as explained earlier, to create the needed device nodes manually or you may create/remove them programmatically as part of your driver initialization. Make sure to use the proper file access rights for the device nodes. Stick to the FIFO model described and do not introduce synchronization or blocking behavior.

The FIFO buffer size that your driver provides has to be configurable during runtime. Use `kmalloc` with the `GFP_KERNEL` flag to allocate the FIFO buffer and `kfree` to deallocate it. Your driver has to provide `/proc/fifo_config` for viewing and changing FIFO settings, similar to task 1.2. By writing to `fifo_config`, the user sets the current FIFO buffer size in bytes, e.g., "`echo -n 1024 > fifo_config`". The buffer size can only be changed if the current buffer is empty and both FIFO devices are closed; valid values are between 4 and 4096 bytes; any other values have to result in a proper error code. The default buffer size must be 8 bytes. Reading `fifo_config` has to result in nicely formatted output that states the current FIFO buffer size, the current number of bytes stored within the FIFO, the total number of bytes written to the FIFO, and the total number of bytes read from the FIFO.

Use the kernel logging facilities (`printk`) to log interesting events in your driver, for instance, log device open/close, FIFO full/empty, and attempts to open/write/read while another access to the FIFO is in progress.

You may use and adapt the provided module skeleton code. Make sure to return appropriate error codes for the functions you write. Also make sure to test your FIFO device driver to validate its correct behaviour in all the corner cases, similar to the following:

```
# echo 4 > /proc/fifo_config
# echo -n "123" > /dev/fifo0
# echo -n "HH" > /dev/fifo0
  => error (too much data)
# cat /dev/fifo1
123
# cat /dev/fifo1
  => (FIFO is empty = empty output)
# echo -n "1234" > /dev/fifo0
# cat /dev/fifo1 & cat /dev/fifo1
1234 & error (FIFO already open)
```

Question 2.1

Load your FIFO device driver. What happens if you attempt to change the FIFO buffer size from two processes at the same time? What happens if you run two parallel processes where one is attempting to change the buffer size while the other is attempting to write data? Perform your tests multiple times; is the outcome always the same?

Question 2.2

Why is a FIFO that can be accessed by only one process at a time not very useful? What is the problem in extending your FIFO to allow for reading and writing in parallel?

Question 2.3

Imagine you want to extend your FIFO driver to behave like a traditional blocking pipe. Where would you need to add block and unblock operations for the involved processes?