Exercise in subject:	**DAT320 Operating Systems**
Title:	**Lab 6: Linux kernel drivers**
Deadline:	**Tuesday Nov 4 2014 23:59**
Expected effort:	**20-25 hours**
Grading:	**Graded**
Submission:	**Group or Individually**
Remarks:	Links in this document are clickable.

University of Stavanger
Faculty of Science and Technology

1 Introduction

This lab assignment is divided into two parts. In the first part, you will build a simple Linux device driver, and install it on a virtual machine image. Then you will write two small programs, one in C and one in Go, that uses this device driver. In the second part of the lab, you will study how an operating system handles virtual and physical memory.

Very Important: *Read the entire document before you begin programming.*

2 Virtual Image

In some parts of this lab you are going to use a stripped down version of Linux Slackware, which we will run from a VirtualBox virtual machine. This image does not have a GUI, and is very minimalistic, so everything must be done directly from the shell. We are using virtual machines in this lab, because you will need to have root access to the system when developing device drivers.

2.1 Starting the Virtual Machine Image

The virtual machine image can be found at */home/share/ide/dat320/StudentSlack.ova* on the machines on the Unix system, e.g. one of the pitter machines. Download/copy the image to the machine you are working on. You can run the image on Linux, Windows, and Mac hosts. The Linux lab machines has the latest version of VirtualBox installed, but if you prefer to use your own machine, you first need to install VirtualBox from **www.virtualbox.org/wiki/Downloads**.

Once VirtualBox has been installed, use `File - Import appliance` inside Oracle VM VirtualBox Manager to import the image. Start the image and log in with the following user/password: *user/user*. You can also log in as root using: *root/root*, but only use the root account if you absolutely need to. We recommend that you adopt the habit of using the *user* account, and the `sudo` command to become root when necessary.

3 Part 1: Loadable Kernel Module

In this first part of the lab you will develop a Loadable Kernel Module that can be dynamically loaded into a running Linux kernel. Kernel modules runs in kernel mode (i.e., not in user mode as normal applications does) within the kernel's address space, and it can interact with the running kernel in just about any way. Key OS concepts such as execution context, synchronization, and low-level memory management come into play in the construction of a Loadable Kernel Module.

The most common use of a kernel module is in the construction of device drivers. There are three main types of device drivers in Linux: character drivers, block drivers, and network drivers. In this assignment we will only work with character drivers.

Character device files look just like regular files, and can be "read from" and "written to" using the standard filesystem API: `open(), close() read(), write()` system calls at the granularity of an individual byte. These types of drivers are used for most devices, including keyboards, mice, webcams, etc.

Please be aware that inside the device driver you have unrestricted access. This means that you need to program carefully to avoid crashing the kernel (which is another reason to use a virtual machine when developing modules). Since a kernel module has this much power, all operations regarding the installation, removal and configuration of a kernel module must be done by the *root* user, e.g. using the `sudo` command.

3.1 Preparations

Before you start developing your kernel module, you must read chapter 2 and 3 of "Linux Device Drivers, Third edition". This book is free and can be found at `http://lwn.net/Kernel/LDD3/`. And remember that: google is your friend!

3.2 Getting Started

We will start with a very simple module. This module will not have any real functionality; our objective is rather to install and to remove the module. We will use the example source code in Listing 1 and the example makefile in Listing 2.

Tasks and Questions:

3.3 Tasks

1. To move files back and forth between your computer and the virtual machine image, it is highly recommended that you *share* a folder on your computer with the VM. This will allow you to edit your files on your host computer (in a GUI), and only do the actual build on the VM.

2. Log into the Virtual Image as *user*. Copy the `lab6` folder to the VM, and ensure that it contains a folder named `simp_lko` and that it contains the two files from Listing 1 and Listing 2. Build the driver in this folder by running `make`. The output from the build is the actual device driver `simp_lko.ko`.

3. Install the driver by executing the `insmod` command:
   `sudo insmod simp_lko.ko`
   Use `sudo lsmod | grep simp` to verify that the module has been loaded.

   PS: `insmod, lsmod, rmmod and mknod` is located in `/sbin`. `/sbin` might not be added to the path. Add to path to avoid specifying full path each time you use those commands.

4. Check out the code in `simp_lko.c`. The `printk()` messages in the driver will be written to a special log file. You can use the `dmesg` command to look at the messages.

5. Remove the driver by using the `rmmod` command.

6. Use `dmesg` to verify that the remove message is written.

3.4 Questions

(a) (5 points) A kernel module needs knowledge about where the kernel expects modules to be installed. This is done in the build process. Examine the two files in the `simp_lko` folder. Where is the include files for the Linux kernel located on the virtual image, and how does C and make know how to access these files?

(b) (5 points) A kernel module must use `printk()` for debugging. Why can't `printf()` be used?

(c) (5 points) The makefile for a driver is only an entry for calling another subset of makefiles. What is the location of the makefile that actually performs the build?

(d) (5 points) What happens if you remove the `MODULE_LICENSE` line in the code?

Listing 1: Simple kernel module.

```
/*****************************************************************************

DESCRIPTION:  A very simple LKM

*****************************************************************************/

/*--------------------  I n c l u d e   F i l e s  -------------------------*/

#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

/*--------------------  C o n s t a n t s  ---------------------------------*/

#define DEV_NAME "simplkm"

/*--------------------  T y p e s  -----------------------------------------*/

/*--------------------  V a r i a b l e s  ---------------------------------*/

/*--------------------  F u n c t i o n s  ---------------------------------*/

int init_module(void)
{
  printk(KERN_INFO "Hello␣world!␣I'm␣%s␣and␣I'm␣being␣installed!\n",DEV_NAME);
  /*
  * A non 0 return means init_module failed; module can't be loaded.
  */
  return 0;
}

void cleanup_module(void)
{
  printk(KERN_INFO "Goodbye␣world,␣the␣%s␣lko␣is␣being␣removed.\n", DEV_NAME);
}

MODULE_AUTHOR("Morten␣Mossige,␣University␣of␣Stavanger");
MODULE_DESCRIPTION("Simple␣Linux␣devicedriver");
MODULE_LICENSE("GPL");
```

Listing 2: Makefile for a simple kernel module.

```
obj-m := simp_lko.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
```

```
all:
  $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
  $(MAKE) -C $(KDIR) M=$(PWD) clean
```

3.5 Extending the Module with `read()`

Now we are going to extend our simple kernel module with some functionality. First we will add support for `read()`. There is a working version of a driver on github in the `simp_read_s` folder. However, its functionality is not as we would like to have it.

Tasks and Questions:

3.6 Tasks

1. Make sure that your `lab6` folder contains the `simp_read_s` folder pulled from github. Build the new driver and install it using `insmod`.

2. To attach the driver to the filesystem we must do an additional step using the `mknod` command. Details on how to use the `mknod` command can be obtained by reading log produced by the `dmesg` command after the driver has been installed with `insmod`.

3. Next, we must also take care of access rights for the new drivers we create. The *user* account on the VM image belong to the group `wheel` and this group must be given read and write access to the device file `simp_read` as follows:
   `sudo chgrp wheel /dev/simp_read`
   `sudo chmod 664 /dev/simp_read`

4. Now you can read from the driver by executing the command: `cat /dev/simp_read`. The output should be many repeating lines with the text *Number of driver read=????*. Hit Ctrl-C to stop the cat command's output from the driver.

5. The small Python script given in Listing 3 can be used to access the the driver from user space. The script is also in the corresponding folder pulled from github. Try it!

Listing 3: Userspace python script to access a devicedriver

```
# A small userspace application i python for accessing the
# simp_read devicedriver

a=file('/dev/simp_read')
b=a.readline()
print 'This␣is␣what␣I␣read␣from␣the␣driver:', b
```

3.7 Questions

(a) (10 points) Write a Go program that replicates the behavior of the Python script in Listing 3.

(b) (10 points) Rewrite the driver so that it produces the following output:

```
% cat /dev/simp_read
Number of driver reads=1
% cat /dev/simp_read
Number of driver reads=2
% cat /dev/simp_read
Number of driver reads=3
...
```

(c) (10 points) Write a Go program called `userread` that replicates the last task. The output should be as follows:

```
% ./userread
Number of driver reads=1
% ./userread
Number of driver reads=2
% ./userread
Number of driver reads=3
...
```

3.8 Extend the Module with `write()`

We will now continue and expand our driver module to support both `read()` and `write()`. We will also use this driver from Go. Our new driver should now behave as a small mailbox. It should be possible to write short messages to the mail box, and later read the message back. Below is shown the expected behavior of an interaction with our driver.

`% echo hello > /dev/simp_rw`	Writing a message
`% cat /dev/simp_rw`	Read back
`hello`	The displayed result
`% cat /dev/simp_rw`	Try to read once more
	No message left to read
`% echo msg1 > /dev/simp_rw`	Writing a message
`% echo msg2 > /dev/simp_rw`	Writing a message
`% cat /dev/simp_rw`	Read back
`msg2`	msg1 is lost
`% cat /dev/simp_rw`	Try to read once more
	No message left to read

3.9 Questions

(a) (10 points) Extend the driver to support `write()` to a message box as illustrated with the above execution example.

(b) (5 points) Create a user-space Go program called `msgbox` that accesses the driver and can be used in the same way as the `cat` and `echo` commands above. That is, it should provide the following command line interface:

`% msgbox hello`	Writing a message
`% msgbox`	Read back
`hello`	The displayed result
`% msgbox`	Try to read once more
	No message left to read
`% msgbox msg1`	Writing a message
`% msgbox msg2`	Writing a message
`% msgbox`	Read back
`msg2`	msg1 is lost
`% msgbox`	Try to read once more
	No message left to read

4 Part 2: Virtual Memory

In this section we will look at how memory is handled by an operating system. We will have special focus on what happens when we try to access memory we don't have legal access to by the operating system.

We can get the address of a memory location in C by using a pointer. If a pointer in C points to a memory location, we can get the actual location of this memory.

You can decide either to solve the following tasks by writing small programs and a short explanation, or by giving a more detailed theoretical answer.

Tasks and Questions:

4.1 Tasks

(a) (5 points) If memory is allocated in the kernel (or in a kernel module), and then the pointer to this memory is returned to a user-space process (application) where the memory is accessed, what will happen? Why?

(b) (5 points) If memory is allocated in a user-space application, and then the pointer is passed to the kernel where the memory accessed, what will happen? Why?

(c) (5 points) If memory is allocated in a user-space application, and then the pointer is passed to another user-space application where the memory accessed, what will happen? Why?

(d) (5 points) If memory is allocated in a thread inside a user-space application, and then a pointer is passed to another thread in the same user-space application where the memory accessed, what will happen? Why?

5 Some Useful Information

dmesg The dmesg command shows messages in kernel ring buffers. These messages contain valuable information about device drivers loaded into the kernel when booting as well as when connecting a hardware device to the system on the fly. In other words dmesg will give us details about hardware drivers connected to or disconnected from a machine and any errors when hardware driver is loaded into the kernel. These messages are helpful in diagnosing and debugging hardware and device driver issues. See also: http://www.linuxnix.com/2013/05/what-is-linuxunix-dmesg-command-and-how-to-use-it.html

insmod The *insmod* command tries to insert a module into the running kernel by resolving all symbols from the kernel's exported symbol table.

mknod The mknod command creates device special files. This is the command that allows us to attach a kernel module with the Linux file system.

screen The screen utility is a full-screen window manager that multiplexes a physical terminal between several processes (typically interactive shells). The same way tabbed browsing revolutionized the web experience, screen can do the same for your experience in the command line. Instead of opening up several terminal instances on your desktop, screen can do it better and simpler. See also http://www.cyberciti.biz/tips/linux-screen-command-howto.html

An overview of some useful screen commands:

- Ctrl+a c - Creates a new screen session so that you can use more than one screen session at once.
- Ctrl+a n - Switches to the next screen session (if you use more than one).
- Ctrl+a p - Switches to the previous screen session (if you use more than one).
- Ctrl+a d - Detaches a screen session (without killing the processes in it - they continue).