Exercise in subject:	**DAT320 Operating Systems**
Title:	**Lab 6: Linux kernel drivers**
Deadline:	**Sunday Nov 11 2013 23:59**
Expected effort:	**20 hours**
Remarks:	Links in this document are clickable.

University of Stavanger

Faculty of Science and Technology

1 Introduction

Also this lab assignment is divided into two parts. In the first part we will build our own small device-driver. We will install the driver on the Virtual Image, and we will write some small applications in both C and GO that uses this driver.

In part two of the lab, we will look into how an operating system like Linux handle virtual and physical memory. We will also write some small programs i C to check how the operating system behaves when we try to access memory. We will use the knowledge we got from the first part of the lab to write some small Loadable Kernel Module for doing experiments with memory.

2 Important information

There is alot of information in this lab. Don't start on **any** programming before you have read the entire text in this lab.

3 Part 1: Loadable Kernel Module

In this first part of the lab we will make a Loadable Kernel Module that can be dynamically loaded into a running Linux kernel. Loadable Kernel Module will run the kernels address space (i.e., not in user mode like an 'normal' application), 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 Loadable Kernel Module.

The most common usage of Loadable 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 `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 full access to do whatever you would like to. That means you can easily crash the kernel if you're not aware. But that gives you also the power to do very sophisticated things that can not be done in user space like accessing the physical hardware and the CPU etc. Since a Loadable Kernel Module has this much power, all operation of installing, removing and setting up a Loadable Kernel Module must be done by *root*-user.

3.1 Preparations

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

3.2 Getting started

We will start with the a very simple Loadable Kernel Module. This Loadable Kernel Module will not have any other functionality other then being able to install, and to remove. 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 Image, it is highly recommended that you *share* a folder on your computer with the Virtual Image. By doing this, you can edit your files on your host computer, and only do the actual build on the Virtual Image. For more info look at *Share folder between host and guest on VirtualBox* on It's learning.
2. Log into the Virtual Image as *user*. Create a folder, and create the two files in Listing 1 and Listing 2(or download *simp_lko.zip* from It's learning). Build the driver (execute *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 is listed.
   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. Look in the *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) The Loadable Kernel Module must have some knowledge about the kernel where it is intended to be installed on. This is done in the buildprocess of the Loadable Kernel Module. Where is the include files for the Linux kernel located on the virtual image, and how does *C/make* know how to access these files?

(b) (5 points) The Loadable Kernel Module uses `printk()` for debugging/info. Why can't `printf()` be used?

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

(d) (5 points) What is the result of removing 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 the 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 Loadable Kernel Module with `read()`

We are now going to extend the Loadable Kernel Module with more functionality. First we will add support for `read()`. There is a fully functional, working version of a driver on It's learning (*simp_read_s.zip*), but the functionality is not as we would like to have it.

Tasks and Questions:

3.6 Tasks

1. Make a new working directory on the Virtual Image. Download the files from It's learning. Build the new driver and install it with *insmod*.

2. To 'connect' the driver with the file-system, we need to do an additional step by using the *mknod* command. The info on how to use *mknod* command to install our driver into the file-system can be read in the dmesg-log after the driver is installed with *insmod*.

3. We also need to take care of access rights for the new drivers we create. The *user* account on the Virtual Image belong to the group *wheel* and this group must be given the correct access rights:

   ```
   sudo chgrp wheel /dev/DEVNAME
   sudo chmod 664 /dev/DEVNAME
   ```

4. Do a *read* from the driver now by executing *cat /dev/simp_read*. The output should be many repeating lines of *Number of driver read=????*. Hit ctrl+C to abort the output.

5. The small Python-script given in Listing 3 can be used to access the the driver from userspace. Try it!

3.7 Questions

(a) (10 points) Write a small Go application that replicate the behaviour of the Python-script given in Listing 3.

(b) (10 points) Rewrite the driver to give the following output (when executing *cat /dev/simp_read*):
cat /dev/simp_read
Number of driver read=1
cat /dev/simp_read
Number of driver read=2
cat /dev/simp_read
Number of driver read=3
...

(c) (10 points) Write a small application in GO replicate last task. The output should be as follows (assuming that the name of your application is *userread*:
./userread
Number of driver read=1
./userread
Number of driver read=2
./userread
Number of driver read=3
...

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
```

3.8 Extend the driver with `write()`

We will now continue and expand our driver to support both `read()` and `write()` We will also use this driver from GO. Our new driver should now behave like a small mailbox. It should be possible to write short messages to the mail box, and later read the message back. You can decide how many messaged the driver should be able to handle.

If you support multiple number of messages, you will able to earn 20 extra point for this lab. If you decide to support only one message, you will still be able complete the whole lab with a total of 100 points. The option with extra points is to be considered a bonus for students that want to challenge themselves. Behaviour for the single message should be as follows:

`%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

The behaviour for the multiple message *could* be like this:

`%echo message1 > /dev/simp_rw`	Write first message
`%echo message2 > /dev/simp_rw`	Write second message
`%echo message3 > /dev/simp_rw`	Write third message
`%cat /dev/simp_rw`	Read back all messages
`message1`	The displayed result
`message2`	The displayed result
`message3`	The displayed result
`%cat /dev/simp_rw`	Try a new read
	No message left to read

3.9 Questions

(a) (10 points) Extend the driver to support `write()` with single message-box.
Or

(b) (20 points) Extend the driver to support `write()` with multiple message-box.

(c) (5 points) Make a userspace GO-application that access the driver and can be used in the same way as the `cat, echo` previously shown.

(d) (5 points) Explain how the driver is protected if more messages is written then the number of messages the driver is able to handle. You should decide and explain your strategy.

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.

4.1 Passing pointers as strings

In a real application we will never pass a pointer to an object as a readable string. But in this lab it will be easier to debug if we send and receive pointers as strings, especially when we pass pointers back and forth between a driver. In Listing 4 a small program written in C shows how to manipulate a pointer into a text string and then back again to a pointer.

Before you start you should take a snap-shot of the Virtual Image. There is risks that the Virtual Image will crash catastrophically in this lab, and you might have to roll back.

Listing 4: Demo of memory access

```c
/****************************************************************************

DESCRIPTION A small main() that opens shows how to convert a pointer
to a string and convert it back to a pointer

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

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

#include<stdio.h>
#include<stdlib.h>

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

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

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

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

int main()
{
  char *aBuff;
  char *anotherBuff;

  char pString[16];
  char *end;

  /*Allocate some memory*/
  aBuff = malloc( 64 );

  /*... and write something into the memory */
  sprintf(aBuff, "Some␣text␣in␣the␣buffer");
  /*Then convert the address of the buffer to a string*/
  sprintf(pString,"%p",aBuff);

  /*We now have a string (pString) where we have stored a pointer */
  /*This string can be printed and read as text */
  printf("aBuff=%s\n",pString);


  /* Now onvert the string back to a 'real' pointer*/
  anotherBuff = (char*)strtol( pString, &end, 16 );

  /* The easy way would have been to do like this: */
  /* anotherBuff = aBuff */
  /* but instead we have gone through convertion to text */

  /* finaly print what our new pointer is pointing to*/
  /* which should be the same as aBuff is pointing to*/
  printf("anotherBuff=%p,␣and␣the␣contenst␣of␣this␣memory␣is:␣\"%s\"\n", anotherBuff, anotherBuff );

  /* release out allocated memory to avoid memory lekage */
  free( aBuff );

  return 0;
}
```

Tasks and Questions:

4.2 Tasks

1. Allocate memory in kernel (in a Loadable Kernel Module), pass pointer to userspace, and `read` what the pointer is pointing to.

   You can extend the driver you already have written and the C application in Listing 4 Please be aware that it is not possible to do `malloc(), free()` in the kernel.

   What is the name of the kernel versions of these functions? When you have finished the driver, try to do a `cat` from it and verify that you get something like `0x0badbeef` or similar

2. Allocate memory in userspace, pass to kernel to, and try to access inside the driver. Again use `cat, echo` to verify the driver.

3. Allocate memory in one process, pass a pointer to this memory to another process, and from that process do `read` to what the pointer is pointing to.

   Please be aware that both processes need to run at the same time. Use the `screen` utility or *ctrl+alt+F(1-9)* to open a new console/tty.

4. Allocate memory in one thread, pass pointer to another thread. Also here you must be sure that both threads are running at the same time.

5. Allocate memory in one kernel-driver, pass to another kernel-driver, try to access.
   Use `cat` and `echo` to read and write between the two different drivers.

4.3 Questions

(a) (25 points) Fill out Table 1. For row 1-4 it is possible to earn 5 points pr row, and for row 5 is is possible to earn 10 points.

(b) (5 points) Explain why we need to use `copy_to_user()`, `copy_from_user()` in a driver

Table 1: Memory allocation and dereference

Points	**Where memory is allocated**	**Tried to dereference**	**Textual value of pointer**	**Result**
5p	Kernel	Userspace		
5p	Userspace	Kernel		
5p	Userspace process 1	Userspace process 2		
5p	Userspace thread 1	Userspace thread 2		
5p	Kernel driver 1	Kernel driver 2		

5 Deliverable

Please refer to Table 2 for an overview of what to deliver in this lab. As mentioned in Section 3.9 you can decide if you want to do question a or b. But you clearly need to indicate your selection.

6 Some use full information

6.1 The *dmesg* command

dmesg is a command which will show Kernel ring buffers. These messages contain valuable information about device drivers loaded into the kernel at the time of booting as well as when we connect a hardware to the system on the fly. In other words *dmesg* will give us details about hardware drivers connected to, disconnected from a machine and any errors when hardware driver is loaded into the kernel. These messages are helpful in diagnosing or 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`

6.2 The *insmod* command

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

6.3 The *mknod* command

The *mknod* command creates device special files. This the command that *connects* a Loadable Kernel Module to the Linux file system.

Table 2: Grading

Max points	**Section**	**What to deliver**
20p	3.4 question a-d	Written answer to the questions
10p	3.7, question a	Source-code of the GO-application
10p	3.7, question b	Source-code of the driver
10p	3.7, question c	Source-code of the GO-application
10p	3.9, question a	Source-code of the driver
20p	3.9, question b	Source-code of the driver
5p	3.9, question c	Source-code of the GO-application
5p	3.9, question d	Written answer to the question
5p	4.3, question a, row 1	Source-code of the driver and C-application, written explanation of the result
5p	4.3, question a, row 2	Source-code of the driver and C-application, written explanation of the result
5p	4.3, question a, row 3	Source-code of the C-application(s), written explanation of the result
5p	4.3, question a, row 4	Source-code of the C-application, written explanation of the result
5p	4.3, question a, row 5	Source-code of the driver(s), written explanation of the result
5p	4.3, question b	Written answer to the question

6.4 The *screen* command

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 use full *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).