IC411 Lab2 Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Spring AY2016

**Summary**: Using examples as a guide, create a kernel module.

**Learning Objectives**: Understand how kernel modules are implemented in a microkernel OS architecture.

**Estimated Completion Time**: 2-3 hours **Lab Weight**: 20 points

**Collaboration**: Per course policy.

**Provided Files**

* hello/hello.c hello/Makefile : part 0
* hellodev/gonavy.c hellodev/Makefile : part 1
* reverse/reverse.c reverse/Makefile: part 2

**What to submit**

* README : answering any questions in this worksheet and a description of your kernel module
* gonavy.c : kernel module code
* reverse.c : kernel module code

**Prep**

You will need a Linux OS that you have sudo on. This lab has been tested in Ubuntu. You will also need the kernel headers installed:

sudo apt-get install linux-headers-$(uname -r)

**Part 0: Hello Kernel Module**

To keep the kernel small be extensible, kernel modules are used to expand and add features to the kernel in a dynamic way. It is relatively simple to implement a kernel module and to load it into the kernel. Here is a “hello world” example:

#include <linux/module.h> // included for all kernel modules

#include <linux/kernel.h> // included for KERN\_INFO

#include <linux/init.h> // included for \_\_init and \_\_exit macros

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Aviv");

MODULE\_DESCRIPTION("A Simple Hello World module");

/\* Initialize the LKM \*/

static int \_\_init hello\_init(void)

{

printk("Hello, world - this is the kernel speaking\n");

return 0;

}

/\* Cleanup - undo whatever init\_module did \*/

static void \_\_exit hello\_cleanup(void)

{

printk(KERN\_INFO "Cleaning up module.\n");

}

module\_init(hello\_init);

module\_exit(hello\_cleanup);

You’ll notice that the C programming is a bit different because we are NOT writing code that runs in user space, but rather writing code that runs in the kernel space. The two functions implemented initialize the module and cleanup the module. Each also will print to the kernel log, also known as dmesg, when performing these actions.

We can now compile the kernel module with the provided Makefile, install the module, and remove the module.

$ make

make -C /lib/modules/3.13.0-32-generic/build M=/home/user/git/ic411/lab/03-kernel-modules/sol/hello modules

make[1]: Entering directory `/usr/src/linux-headers-3.13.0-32-generic'

Building with KERNELRELEASE = 3.13.0-32-generic

CC [M] /home/user/git/ic411/lab/03-kernel-modules/sol/hello/hello.o

Building modules, stage 2.

Building with KERNELRELEASE = 3.13.0-32-generic

MODPOST 1 modules

CC /home/user/git/ic411/lab/03-kernel-modules/sol/hello/hello.mod.o

LD [M] /home/user/git/ic411/lab/03-kernel-modules/sol/hello/hello.ko

make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'

$ sudo insmod hello.ko

[sudo] password for user:

$ sudo rmmod hello.ko

Looking at the dmesg output, we see the kernel speaking to us:

$ dmesg | tail -2

[1047471.384934] Hello, world - this is the kernel speaking

[1047479.764562] Cleaning up module.

Questions/Instructions:

1. You need to have super user access to install and uninstall a kernel module, why might his be a good thing?
2. The command lsmod will list installed modules. Install your hello module and copy the results of lsmod into your README file to show you properly installed.
3. The lsmod command is actually just a nice program to read /proc/modules. Do a cat of /proc/modules and copy that output to your README file.

**Part 1: Hello Device Drivers**

One nice feature of the kernel modules is the ability to quickly add device drivers. Device drivers are “files” that live under /dev that user programs can read and write, but at the other end is the kernel program doing processing. You should already be aware of a few examples of such device drivers, such as /dev/null, /dev/zero, and /dev/random which produces EOF, zeros, and random bits when read or written too.

We can also produce our own device driver via the misc device driver functionality. An example of the intitialization of such a device driver is on the next page.

//function declarations

static int gonavy\_init(void);

static void gonavy\_cleanup(void);

static ssize\_t gonavy\_read(struct file \* file, char \* buf, size\_t len, loff\_t \*ppos);

static int gonavy\_open(struct inode \*inodep, struct file \*filep);

//setting up device structures

static struct file\_operations gonavy\_fops = {

.open = gonavy\_open,

.read = gonavy\_read,

};

static struct miscdevice gonavy\_dev = { MISC\_DYNAMIC\_MINOR,"gonavy",&gonavy\_fops};

//globals

static char gonavy\_str[]="Go Navy!\n";

static int gonavy\_len=9;

//specify our init and cleanup functions

module\_init(gonavy\_init);

module\_exit(gonavy\_cleanup);

//Initialize the module

static int \_\_init gonavy\_init(void){

int ret;

printk(KERN\_INFO "gonavy: Initiallizing /dev/gonavy\n");

if( (ret = misc\_register(&gonavy\_dev) ))

printk(KERN\_ERR "gonavy: Unable to register \"/dev/gonavy\" device\n");

return ret;

}

//De-initialize the module

static void \_\_exit gonavy\_cleanup(void){

printk(KERN\_INFO "gonavy: Deinitillizing /dev/gonavy\n");

misc\_deregister(&gonavy\_dev);

}

//Open the /dev/gonavy

static int gonavy\_open(struct inode \*inodep, struct file \*filep){

printk(KERN\_INFO "gonavy: /dev/gonavy opened\n");

return 0;

}

The specifics of this code is provided in the comments, but the general idea is that you must provide the functionality for which functions are called when the device is read and opened. That particular functionality, at least for this device driver, is that when you read from /dev/gonavy, you always read “Go Navy!\n”. Missing from the above code is the gonavy\_read() function, and we describe that on the next page.

Here’s the gonavy\_read() function which is called when you read from /dev/gonavy:

//called whenever /dev/gonavy is read from

static ssize\_t gonavy\_read(struct file \* filep, char \* buf, size\_t len, loff\_t \*ppos){

//filep : structure of the file

//buf : userspace address to write what was read to

//len : request number of bytes to read

//ppos : offset into the file

if ( len < gonavy\_len)

return -EINVAL; //support reading the whole string

if ( \*ppos != 0 )

return 0; //read the string already, indicate EOF

//write string to user space, returns 0 on success

if(copy\_to\_user(buf,gonavy\_str,gonavy\_len))

return -EINVAL; //error condition

\*ppos = gonavy\_len; //indicate we've read this far into the file

return gonavy\_len;

}

The arguments are described in the function comments, but the important thing to note is that buf is a user space address and this is running in the kernel. Once the module has identified that, yes, there is enough space in the buffer buf to write “Go Navy\n” it will copy\_to\_user() the “Go Navy\n” string.

We can now compile, load the module, and read from the device:

$ make

make -C /lib/modules/3.13.0-32-generic/build M=/home/user/git/ic411/lab/03-kernel-modules/sol/hellodev modules

make[1]: Entering directory `/usr/src/linux-headers-3.13.0-32-generic'

Building with KERNELRELEASE = 3.13.0-32-generic

Building modules, stage 2.

Building with KERNELRELEASE = 3.13.0-32-generic

MODPOST 1 modules

make[1]: Leaving directory `/usr/src/linux-headers-3.13.0-32-generic'

$ sudo insmod gonavy.ko

$ sudo cat /dev/gonavy

Go Navy!

Questions/Instructions:

1. Change the module such that in addition to get “Go Navy!\n” when reading from /dev/gonavy you also get “Go Navy!\nBeat Army!\n”.
2. Try compiling your kernel module with strlen() instead of hard coding the length? Can you? Why will this work or why not? Answer this in you README

**Part 2: Reverse Module**

In this part of the lab you will close the loop on kernel modules by implementing a device driver that will reverse it’s input. This means adding a write() functionality for your device. For example, here is the module in action:

$ sudo insmod reverse.ko

$ echo -n "Hello, World!" | sudo tee /dev/reverse > /dev/null

$ echo $(sudo cat /dev/reverse)

!dlroW ,olleH

$ dmesg | tail -6

[1049219.376630] reverse: Initiallizing /dev/reverse

[1049221.622616] reverse: /dev/reverse opened

[1049221.622629] reverse: write: wrote 13 bytes to revbuffer

[1049231.654508] reverse: /dev/reverse opened

[1049231.654529] reverse: read: revbuffer="!dlroW ,olleH" revbuffer\_size=13

[1049231.654566] reverse: read: revbuffer="!dlroW ,olleH" revbuffer\_size=0

Your module should conform to the following requirements for the device

- Only one string can be written and stored in the device at one time, subsequent writes to the device should fail with EINVAL

$ echo -n "Hello, World!" | sudo tee /dev/reverse > /dev/null

$ echo -n "Hello, World!" | sudo tee /dev/reverse > /dev/null

tee: /dev/reverse: Invalid argument

- If nothing is the device when read, an error should occur with EINVAL

$ echo $(sudo cat /dev/reverse)

!dlroW ,olleH

$ echo $(sudo cat /dev/reverse)

cat: /dev/reverse: Invalid argument

$

In your README file, describe your algorithm and any debugging strategies you used.