EEE335 – Lab 3

Kernel Modules and Process Structure

Objective

The purpose of this lab is to familiarize yourself with Linux modules, the `/proc` directory and some internal structures of the Linux kernel.

Introduction

This laboratory will require research from you. You will have to read in the The Linux Kernel Module Programming Guide available on-line to be able to accomplish the 3 tasks assigned to you below, and enable you to answer the questions at the end of the lab. As we discussed in class, Linux is a monolithic operating system. The different services provided by the kernel and device drivers all reside in kernel mode and have access to the same address space. Linux, however, offers the advantage of modules that can be inserted into the kernel without having to recompile it completely. We will take advantage of this to explore the internal kernel structure for process management.

Note: For this lab, you will use the desktop VM. Start Visual Studio Code (VSCode) and select **Menu > Terminal > New Terminal**. In the terminal window, type the following command to download the lab 3 start code in your home directory.

```
$ cd ~
$ wget --user 335 --password YellowMorningSun https://
roberge.segfaults.net/wordpress/files/EEE335/labs/lab3.zip
$ unzip lab3.zip
```

In VSCode open the `~/lab2/task1` directory. You are now ready to complete task 1.

Task 1

To familiarize yourself with kernel modules, you will write one and execute it. This first module will do nothing very useful since it will only write *Hello World!* in the console when inserted and Goodbye *World!* when removed.

In the **`task1`** directory, you should find a Makefile and a C file. Use these two files to program and compile your kernel module. Note that you will not be able to compile and run your module using the VSCode buttons, you will need to use the terminal window within VSCode.

In order to guide you in your search for modules, you can look at The Linux Kernel Module Programming Guide available online. Sections 1.1, 1.2 and 2.1 to 2.5 will be helpful; it is recommended you read them carefully. In this lab, you will also be introduced to the `make` utility for building programs and *Makefiles*, generally used to manage groups of programs. It is recommended you read the man page for `make`, specifically the description section.

Before compiling your module, be sure to write the appropriate *Makefile*. Be sure to compile your module for the current Linux kernel for which you are creating your modules.

To insert the module, you will need to use administrator privileges. Since you do not have access to the root account, we added your account to the `sudo` group that enables you to use the `sudo` command.

It is necessary that you insert the module from the console rather than a terminal emulator into the interface graphic. To go from the X server to a console, you must do `Ctrl+Alt+F3`. TTY 3 of the console will open and you will need to input your credentials again. If you want to go back to the GUI, you only have to do `Ctrl+Alt+F1 or Ctrl+Alt+F7`.

When you are ready to demonstrate your module, please call your instructor.

Task 2

In Linux, the `/proc` directory is different from other directories in that the files it contains are not really files, they are rather windows on the inner workings of the core. These contain all kinds of information about the operating system. For example, if you reinsert your module, you can confirm that it is in the kernel by printing on the screen the content of `/proc/modules`. The name of your module should appear at the top of the list which is displayed.

The `/proc` directory also contains information about running processes. Read the man page for `proc`, specifically up to the end of the *overview* section, skim the rest to get a sense of the information contained in it. To better understand how information is structured, start two terminal emulator windows. In the first, open a session of Vim, you will use the second to search for information on the Vim process from `/proc`.

In your lab report, include the following information (ensure you include details to how and where you found information, not just the response itself):

1. The process number (PID) assigned to Vim.
2. The number of the parent process.
3. The name of the parent process.

Explain how you found this information. You will initially have to use the command `ps` to find the PID associated with the `vim` process. The rest of the information must be found inside from the `/proc` directory.

The following provides useful overview information on the proc filesystem:

- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/deployment_guide/ch-proc
- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/5/html/deployment_guide/s1-proc-directories

Task 3

For this task, you will need to create a module that adds a special file into `/proc`. This file will be called `taskspy`. When you do `cat taskspy`, the module will look for information about process running on the system and `cat` will display them. The information that we ask you to display for each process is the process ID (PID), the PID of the parent (PPID), the user identifier (UID), the command associated with the process (COMMAND), the moment where the process was started (ELAPSED) and its status (STATE). The displayed output should look like:

```
  PID   PPID    UID        COMMAND        ELAPSED      STATE
  751    749      0           bash        6:14:46          S
  756    751      0            vim     1-02:28:56          S
```

The *ELAPSED* field represents the time elapsed between the start of the operating system and the creation of the process. A newly created process will therefore have a longer *ELAPSED* time than a process that was created some time ago. The format is `[[dd-]hh:]mm:ss` where square parentheses `[]` are optional fields. In the kernel, process information is contained in a `task_struct` data structure. The latter contains information such as PID, status, stack pointer, process priority, and so on. It is possible to find the definition of `task_struct` in `the /usr/src/linux-headers-<version>/include/linux/sched.h` file. You can find a copy of this file on the internet. Make sure you choose the correct version of the kernel. To find out which version is installed on your system, use the `uname -r` command. Note that the `task_struct` definition contains many pre-compiler instructions.

Also note that the header files in `/usr/include/linux/` are not the same as those in `/usr/src /linux-headers - <version>/include/linux/`. The header files in `/usr/include/linux/` are the ones used when compiling the standard C library installed on your system. They provide a user interface to the kernel. The header files in `/usr/src/linux-headers-<version>/include/linux/` themselves belong to the kernel and are installed by the `linux-headers-$(uname -r)` package. These files represent kernel internals and are used for compiling modules.

As we saw in class, when Linux is started, an initial process is created. This process was `init` in older versions of the kernel and is now `systemd`. It is possible to access the `systemd task_struct` by using the global variable `struct task_struct init_task` exported by the kernel. Since the processes are in a circular, doubly linked list, it is possible to go from one process to another using the function `struct task_struct* next_task` (`struct task_struct* task`). The list is circular which means that as soon as you reach the end of the list, you start again at the beginning of the list. Be careful not to fall into an infinite loop. The following link provides useful details you may require to get the information you require: https://elixir.bootlin.com/linux/latest/source/kernel

After displaying information about running processes, display the total number of processes on a new line.

Useful Tips:

Some of the information you may need are contained in structs that themselves are members of `task_struct`. For example, `pid` is found in `struct real_parent` and `uid` is found in `struct cred`. Search through `sched.h` to find what you need. Some of the nomenclature may look

unfamiliar, that is Ok. For example, when looking for `*cred` in this file, you will see information related to:

```
const struct cred __rcu  *cred
```

`__rcu` refers to a *read-copy-update* function that simply allows concurrent access to a struct without having to lock it. The double underline is common in kernel programming. From here you can examine the details of struct cred in the associated c file, `cred.c`:
https://elixir.bootlin.com/linux/latest/source/kernel/cred.c

Rotating through the circular, doubly linked list and stopping so you do not loop through it forever. See `linux/sched/signal.h`:
https://elixir.bootlin.com/linux/latest/source/include/linux/sched/signal.h

Creating a special file in /proc

To help you, here is the code of a kernel module to create a special file in `/proc`. After compiling and inserting the module into the kernel, you will be able to read the contents of the file using the command `cat /proc/hello` which should return `Hello World!`.

```c
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

static int hello_show(struct seq_file *m, void *v) {
    seq_printf(m, "Hello World!\n");
    return 0;
}

static int hello_open(struct inode *inode, struct  file *file) {
    return single_open(file, hello_show, NULL);
}

static const struct  proc_ops hello_pops = {
    .proc_open = hello_open,
    .proc_read = seq_read,
    .proc_lseek = seq_lseek,
    .proc_release = single_release,
};

static int __init hello_init(void) {
    proc_create("hello", 0, NULL, &hello_pops);
    return 0;
}

static void __exit hello_exit(void) {
    remove_proc_entry("hello", NULL);
}
MODULE_LICENSE("GPL");
module_init(hello_init);
module_exit(hello_exit);
```

Some explanations to better understand the code:

- Nothing too special with the header files. The linux `/modules.h` file is needed for the module's functions and macros. The linux `/proc_fs.h` file is needed for the `proc_create()` and `remove_proc_entry()` function is used to create and delete the special file in `/proc`. The linux `/seq_file.h` file is needed for the `seq_printf()` function that is used to generate the contents of the special file.
- The `hello_show()` function generates the contents of the special file.
- The `hello_open()` function is a callback called when the special file is opened. Here, we call `single_open()` since the contents of the special file are generated at once with a sequence file. Some explanations on the sequence files are given later.
- The `hello_fops` structure defines the callback functions used for handling the special file.
- The special file is created by the function `proc_create(file_name,permission_bits, parent_dir,file_operations)` where `"hello"` is the name of the special file, 0444 are the permissions of the file, NULL in `parent_dir` means that the file is directly in `/proc` and not in a subdirectory.
- The `remove_proc_entry(name,parent_dir)` function is used to delete our pseudo-file.

A note on file sequence

An important limitation of the original implementation of the `/proc` special file system is that it allows one page of data to be displayed at a time. This page is actually a preset buffer in memory of a usual size of 4096 bytes. The exact value is available using the `PAGE_SIZE` macro. This limitation is bypassed by using sequence files that provide an interface for easily printing multi-page output by printing a sequence of pages. The interface is so convenient that it is also used for a single paged output as we did in the example above.

For the curious, an example code for a sequence file that uses several pages is given in chapter 4 of the Linux Device Drivers 3rd edition book available for free on the internet.

Necessary Header Files

To compile your module, you will need at least the following .h files:

```
#include <linux/module.h>        // Needed by all modules
#include <linux/init.h>          // Needed for initializing the module
#include <linux/proc_fs.h>       // Needed for proc operations
#include <linux/seq_file.h>      // Needed for writing to the sequence file
#include <linux/sched.h>         // Needed for task_struct
#include <linux/sched/signal.h>  // Needed for next_task
```

Test

You can confirm that your module is returning the correct information using the `ps` command. See the `ps` manual page to learn about `ps` options and what they mean.

Demonstration

When you are ready to demonstrate your module, please call your instructor.

Questions

1. Why do we say that it is advantageous not to have to recompile the kernel when a module is inserted?
2. What is the purpose of the `sudo` command?
3. Why does a user need to be part of a the `sudo` group in order to use `sudo`?
4. Why do files inside `/proc` often have a size of 0, but can hold a great quantity of information?
5. What is the significance of the numbered folders in `/proc` folder?
6. Why is it not possible to directly access the function that you defined in your module from a user program?
7. Why does a module have access to kernel variables?
8. Why do some modules in the Linux kernel create files in `/proc` in their initialization routine, but do not erase them in their output routine? Is it simply forgotten?

Lab Report

Your lab report must include a cover page, a brief introduction, discussion and conclusion.
Your discussion should explain how your Task 3 works (i.e how you designed your kernel module), what you learned, any discoveries you made, what was challenging and how you overcame any difficulties.
It should not be a list of followed steps or empty phrases about the importance of this lab. Be concise, a good discussion paragraph is better than 10 pages of filler. The report should also include answers to the above eight (8) questions. Be sure to include the question numbers, the questions themselves and the answers in you report.

Submission

For this lab, you must submit a zip archive that includes the following:

1. Your lab report in PDF which includes all the sections as per the lab report template provided on the course webpage. Your source code and Makefiles for Task 1 and 3 must be included in an annex in your lab report PDF.
2. Include your answers to the questions asked at the end of part 2.
3. Include your answers to the 8 questions asked at the end of the lab instructions.
4. When answering questions, be sure to include the question numbers, the questions themselves and your answers.

The above files need to be submitted via a single compressed file (e.g. .zip) with the following name (e.g. `eee335_lab3_lastname1_lastname2.zip`) and submitted by email to your instructor.