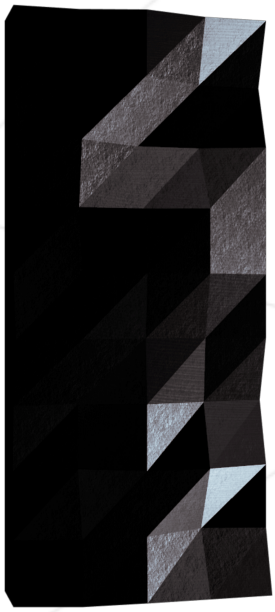

Little Penguin

Linux Kernel Development

Louis Solofrizzo louis@ne02ptzero.me
42 Staff pedago@42.fr

Summary: This entire project is based on the Eudyptula Challenge.
All credits go to little@eudyptula-challenge.org
http://eudyptula-challenge.org/

Version: 1.1

Contents

Chapter I

Preamble

I.1 Extract from Documentation/CodingStyle

Linux kernel coding style

This is a brief document describing the preferred coding style for the Linux kernel. Coding style is very personal, and I won't _force_ my views on anybody, but this is the standard for anything I maintain, and I would prefer it applied to most other projects as well. Please at least consider the points made here.

First of all, I recommend printing out a copy of the GNU coding standards, and then NOT reading it. Burn it—it's a great symbolic gesture.
;) ok, it's definitely not an eco-responsible move.

Anyway, let's begin:

Chapter 1: Indentation

Tabs are 8 characters wide, and therefore, indentations should also be 8 characters. There are some misguided attempts to use 4 (or even 2!) character indentations, which is akin to trying to define PI as 3.

Chapter II

Introduction

This "challenge" is a series of Linux kernel programming assignments, starting with small tasks and gradually increasing in complexity. If all goes well, by the end, you will be qualified enough to maintain a subsystem—if you so desire. Well, maybe not a full maintainer, but certainly skilled enough to identify issues caused by your favorite maintainer. And that can be even more fun than being in charge.

Chapter III

Objectives

- Compile a custom Kernel.
- Build and use a kernel module.
- Learn how drivers work in Linux.

Chapter IV

General instructions

- This subject is similar to a 'Piscine' day. However, some tasks are quite difficult, so take your time.
- All work must be done on your custom Linux distribution. If you don't have one—why are you even here?
- Caution! Some tasks do not require you to submit code, but a form of proof.

Chapter V

Assignment 00

Time to take off the training wheels and build your own custom kernel. No more relying on pre-built versions.

For this task, you must run your own kernel—and use Git! Exciting, right? No? Well... okay.

To Do

- Download Linus's latest Git tree from git.kernel.org (You'll need to figure out which one is his. It's not that hard—just recall his last name.)
- Build, install, and boot it. You may choose whatever kernel configuration you prefer, but you must enable `CONFIG_LOCALVERSION_AUTO=y`.

Turn In

- Kernel boot log file.
- Your `.config` file

Chapter VI

Assignment 01

To Do

- Create a "Hello World" kernel module with the following behavior:

```
% sudo insmod main.ko
% dmesg | tail -1
[Wed May 13 12:59:18 2015] Hello world!
% sudo rmmod main.ko
% dmesg | tail -1
[Wed May 13 12:59:24 2015] Cleaning up module.
%
```

Be careful—the module must compile on any system. (<– version compatibility hint)

Turn In

- Makefile and source code.

Chapter VII

Assignment 02

To Do

- Take the kernel Git tree from Assignment 00 and modify the Makefile to change the `EXTRAVERSION` field. Do it so the running kernel, after modifying, rebuilding, and rebooting, includes "-thor_kernel" in its version string.

Turn In

- Kernel boot log.
- A patch to the original Makefile, following Linux submission standards (Documentation/SubmittingPatches).

Chapter VIII

Assignment 03

Great job making it this far! I hope you're still having fun. Feeling a little bored just booting and installing kernels?
Well then, it's time for something more pedantic—just to remind you how fun those kernel builds actually are!

To Do

- Modify the following C file to comply with the Linux coding style (see Documentation/CodingStyle).

Turn In

- Your updated version of the C file.

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/slab.h>

int do_work(int *my_int, int retval) {
        int x;
        int y = *my_int;
        int z;

        for (x = 0; x < my_int; ++x)
        {
                udelay(10);
        }

        if (y < 10)
                /* That was a long sleep, tell userspace about it */
                pr_info("We slept a long time!");

        z = x * y;
        return z;
            return 1;
}

int my_init(void)
{
        int x = 10;

        x = do_work(&x, x);
        return x;
}

void my_exit(void)
{
}

module_init(my_init);
module_exit(my_exit);
```

Chapter IX

Assignment 04

You survived the coding style chaos—well done!
Now, let's tackle something more "real" because I can tell you're getting a bit bored.

To Do

- Modify the kernel module you wrote in Assignment 01 so that it is automatically loaded when any USB keyboard is plugged in. This should be triggered by the appropriate userspace hotplug tools, which may include depmod, kmod, udev, mdev, or systemd, depending on your distribution.

Turn In

- A rules file, appropriate for your system.
- Your updated code.
- Proof that your module works as intended.

Yes, it seems simple, but it's actually a bit tricky.
For a hint, consult Chapter 14 of the book *Linux Device Drivers, 3rd Edition*. It's freely available online—no purchase required.

Chapter X

Assignment 05

Nice job getting module auto-loading to work. Those hotplug macros can be tricky, but mastering them is a useful skill, especially in real kernel development.
Speaking of real development—let's write some proper code.

To Do

- Take the kernel module you wrote for Assignment 01, and modify it to become a misc character device driver. The misc interface is a super simple way to create a character device without getting caught up in the mess of sysfs and device registration. Trust me, it's way easier this way!
- The misc device should be created with a dynamic minor number—no need to go crazy trying to reserve a static one for your test module. That would be a hassle for no reason.
- Implement both the read and write operations for the misc device.
- The misc device node should appear in `/dev/fortytwo`.
- When the device is read from, it should return your student login to the caller.
- When written to, the input data should be compared to your student login. If it matches, return a successful write response. If not, return an "invalid value" error.
- Register the misc device when your module is loaded, and unregister it when it's unloaded.

Turn In

- Your updated code.
- Some form of proof.

Chapter XI

Assignment 06

Great job with the misc device driver—pretty clean and simple, right?
Just when you thought this challenge was only about coding in the kernel, we're throwing it back to a previous topic. That's right: building kernels again!

This reflects real life. Kernel developers often spend more time rebuilding than writing new code. It's not glamorous, but it's a vital skill.

To Do

- Download the latest `linux-next` kernel. It changes daily, so just use the most recent version. Build it and boot it.

Turn In

- Kernel boot log.

What is the `linux-next` kernel?
That's part of the challenge.

For a hint, check the excellent documentation on the kernel development process in `Documentation/development-process/` within the kernel source tree. It's a good read—you'll learn a lot about how kernel developers work.

For a hint, you should read the excellent documentation about how the Linux kernel is developed in Documentation/development-process/ in the kernel source itself.
It's a great read and should tell you everything you never wanted to know about what Linux kernel developers do and how they do it.

Chapter XII

Assignment 07

We'll return to the `linux-next` kernel in a later task, so don't delete that directory—you'll need it again. But for now, enough kernel building. Let's write more code!

This task is similar to Assignment 05, where you created a misc device. This time, we'll explore a different interface between user and kernel: `debugfs`.

Rumor has it the creator of `debugfs` said the only rule is: "There are no rules." Let's test that theory.

Your distribution should mount `debugfs` at `/sys/kernel/debug/`. If it doesn't, mount it with:

```
mount -t debugfs none /sys/kernel/debug/
```

Make sure it is enabled in your kernel with the `CONFIG_DEBUG_FS` option, as you will need it for this task.

To Do

- Modify the module from Assignment 01 to create a `debugfs` subdirectory named `fortytwo`.
- In this directory, create three virtual files: `id`, `jiffies`, and `foo`.
- `id`: Behaves exactly as in Assignment 05. It must be readable and writable by all users.
- `jiffies`: Read-only by any user. When read, it should return the current value of the kernel jiffies timer.
- `foo`: Writable only by root; readable by everyone. Data written should be stored (up to one page). Reading should return the stored data. Implement proper locking to handle concurrent read/write operations.
- When the module is unloaded, all `debugfs` files must be cleaned up and any allocated memory freed.
- Note: The debug directory must be globally readable. Since there's no option for that, use good old `chown`.

Turn In

- Your code.
- Proof that the module works as expected.

Chapter XIII

Assignment 08

Great work with that `debugfs` task.
Let's slow it down a bit and revisit coding style.

To Do

- Take the following file, fix its coding style, and correct its behavior.

Turn In

- Your updated and style-compliant C file.

```
But wait-what is this code supposed to do?
```

That's part of the challenge. Have fun figuring it out!

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/fs.h>
#include <linux/slab.h>

// Dont have a license, LOL
MODULE_LICENSE("LICENSE");
MODULE_AUTHOR("Louis Solofrizzo <louis@ne02ptzero.me>");
MODULE_DESCRIPTION("Useless module");

static ssize_t      myfd_read
(struct file *fp, char __user *user,
size_t size, loff_t *offs);
static ssize_t      myfd_write(struct file *fp, const char __user *user,
size_t size, loff_t *offs);

static struct file_operations   myfd_fops = {
    .owner      = THIS_MODULE, .read    = &myfd_read, .write        = &myfd_write
};

static struct miscdevice        myfd_device = {
    .minor      = MISC_DYNAMIC_MINOR,.name      = "reverse",
    .fops       = &myfd_fops };

char    str[PAGE_SIZE];
char    *tmp;

static int __init myfd_init
        (void) {
    int     retval;

    retval = misc_register(&(*(&(myfd_device))));
    return 1;
}

static void __exit myfd_cleanup
    (void) {
}

ssize_t     myfd_read
        (struct file *fp,
        char __user *user,
        size_t size,
        loff_t *offs)
{
        size_t      t, i;
        char        *tmp2;

/***************
 * Malloc like a boss
 ***************/
        tmp2 = kmalloc(sizeof(char) * PAGE_SIZE * 2, GFP_KERNEL);
        tmp = tmp2;
        for (t = strlen(str) - 1, i = 0; t >= 0; t--, i++) {
            tmp[i] = str[t];
        }
        tmp[i] = 0x0;
        return simple_read_from_buffer(user, size, offs, tmp, i);
}

ssize_t     myfd_write
        (struct file *fp,
        const char __user *user,
        size_t size,
        loff_t *offs) {
    ssize_t res;

    res = 0;
    res = simple_write_to_buffer(str, size, offs, user, size) + 1;
    // 0x0 = '\0'
    str[size + 1] = 0x0;
    return res;
}

module_init(myfd_init);
module_exit(myfd_cleanup);
```

Chapter XIV

Assignment 09

Phew... I don't know who wrote that last file, but wow.
Anyway, let's get back to writing something more meaningful.

To Do

- Create a module that can list mount points on your system, with the associated name.
- Your file must be named `/proc/mymounts`.

```
$> cat /proc/mymounts
root        /
sys         /sys
proc        /proc
run         /run
dev         /dev
```

Turn In

- The module source code and a Makefile.

This isn't a hard task, but it can be tricky.
Check out documentation on mount points, directory listing, and linked list traversal in the kernel.
And above all—have fun :)

Chapter XV

Submission and Peer Evaluation

Submit your project in your `Git` repository as usual.
Only the contents of your repository will be considered during your evaluation.
Don't hesitate to double check the names of your folders and files to ensure they are correct.