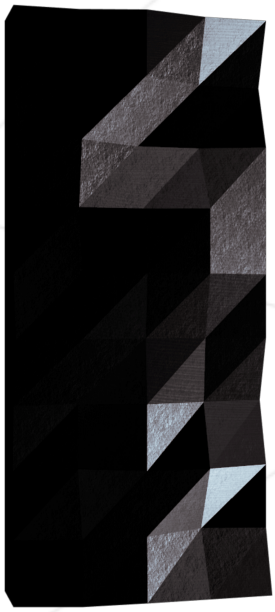

Little Penguin

Linux Kernel Development

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

Summary: *This all subject has been made from the eudyptula challenge.*
All credits goes to little@eudyptula-challenge.org
http://eudyptula-challenge.org/

Contents

Chapter I

Forewords

I.1 Extract from Documentation/CodingStyle

Linux kernel coding style

This is a short 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 what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

Chapter 1: Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Chapter II

Introduction

This "challenge" is a series of Linux kernel programming assignments starting out small, and in the end, if all goes well, you'll be the maintainer of a subsystem, if you so desire. Well, maybe not a maintainer, but you will be qualified enough to point out any problems that your favorite maintainer is causing, and that's actually way more fun than being in charge.

Chapter III

Goals

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

Chapter IV

General instructions

- This subjects is like a 'piscine' day. However, some assignments are hard, so take your time.
- All of your work will be done on your custom linux distribution. If you don't have one, what are you doing here ?
- Careful ! In some assignment you don't need to turn-in code, but a proof.

Chapter V

Assignment 00

It's time to take off the training wheels and move on to building a custom kernel. No more ready-made kernels for you.
For this task you must run your own kernel. And use git! Exciting, isn't it? No? Oh, ok...

Todo

- Download Linus's latest git tree from git.kernel.org (You have to figure out which one is his. It's not that hard, just remember what his last name is and you should be fine.)
- Build it, install it, and boot it. You can use whatever kernel configuration options you wish to use, but you must enable CONFIG_LOCALVERSION_AUTO=y.

Turn in

- Kernel boot log file.
- Your .config file

Chapter VI

Assignment 01

Todo

- Build a "Hello World module" with the following behaviour:

```
% 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.
%
```

Careful, the module must compile on any system (<——- Version hint).

Turn in

- Makefile and code

Chapter VII

Assignment 02

Todo

- Take the kernel git tree from assignment 00 and change the Makefile to modify the EXTRAVERSION field. Do this in a way that the running kernel (after modifying the Makefile, rebuilding, and rebooting) has the characters "-thor_kernel" in the version string.

Turn in

- Kernel boot log
- A patch to the original Makefile, compliant with Linux standards (Documentation/-SubmittingPatches)

Chapter VIII

Assignment 03

Wonderful job in making it this far. I hope you have been having fun. Oh, you're getting bored, just booting and installing kernels?
Well, time for some pedantic things to make you feel that those kernel builds are actually fun!

Todo

- Take the following file, and modify it to match the Linux coding style (Documentation/CodingStyle)

Turn in

- Your 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

Yeah, you survived the coding style mess!
Now, on to some "real" things, as I know you are getting bored already.

Todo

- Take the kernel module you wrote for task 01, and modify it so that when any USB keyboard is plugged in, the module will be automatically loaded by the correct userspace hotplug tools (which are implemented by depmod / kmod / udev / mdev / systemd, depending on what distro you are using.)

Turn in

- A rules file, depending on what system you are using.
- Your code
- Some proof that your code actually works!

Yes, so simple, and yet, it's a bit tricky. As a hint, go read chapter 14 of the book, "Linux Device Drivers, 3rd edition." Don't worry, it's free, and online, no need to go buy anything.

Chapter X

Assignment 05

Nice job with the module loading macros. Those are tricky, but a very valuable skill to know about, especially when running across them in real kernel code.
Speaking of real kernel code, let's write some!

Todo

- Take the kernel module you wrote for task 01, and modify it to be a misc char device driver. The misc interface is a very simple way to be able to create a character device, without having to worry about all of the sysfs and character device registration mess. And what a mess it is, so stick to the simple interfaces wherever possible.
- The misc device should be created with a dynamic minor number, no need to run off and trying to reserve a real minor number for your test module, that would be crazy.
- The misc device should implement the read and write functions.
- The misc device node should show up in /dev/fortytwo.
- When the character device node is read from, your student login is returned to the caller.
- When the character device node is written to, the data sent to the kernel needs to be checked. If it matches your assigned student login, then return a correct write return value. If the value does not match your assigned student login, return the "invalid value" error value.
- The misc device should be registered when your module is loaded, and unregistered when it is unloaded.

Turn in

- Your code
- Some proof

Chapter XI

Assignment 06

Todo

Great work with that misc device driver. Isn't that a nice and simple way to write a character driver?

Just when you think this challenge is all about writing kernel code, this task is a throwback to your second one. Yes, that's right, building kernels. Turns out that's what most developers end up doing, tons and tons of rebuilds, not writing new code. Sad, but it is a good skill to know.

- Download the linux-next kernel for today. Or tomorrow, just use the latest one. It changes every day so there is no specific one you need to pick. Build it. Boot it.

Turn in

- Kernel boot log

What is the linux-next kernel? Ah, that's part of the challenge.

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 all you never wanted to know about what Linux kernel developers do and how they do it.

Chapter XII

Assignment 07

We will come back to the linux-next kernel in a later assignment, so don't go and delete that directory, you'll want it around. But enough kernel building, let's write more code!

This task is much like task 06 with the misc device, but this time we are going to focus on another user/kernel interface, debugfs. It is rumored that the creator of debugfs said that there is only one rule for debugfs use, "There are no rules when using debugfs." So let's take them up on that offer and see how to use it.

debugfs should be mounted by your distro in /sys/kernel/debug/, if it isn't, then you can mount it with the line: mount -t debugfs none /sys/kernel/debug/

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

Todo

- Take the kernel module you wrote for task 01, and modify it to be create a debugfs subdirectory called "fortytwo". In that directory, create 3 virtual files called "id", "jiffies", and "foo".
- The file "id" operates just like it did for assignment 05, use the same logic there, the file must be readable and writable by any user.
- The file "jiffies" is to be read only by any user, and when read, should return the current value of the jiffies kernel timer.
- The file "foo" needs to be writable only by root, but readable by anyone. When writing to it, the value must be stored, up to one page of data. When read, which can be done by any user, the value stored in it must be returned. Properly handle the fact that someone could be reading from the file while someone else is writing to it (oh, a locking hint!)
- When the module is unloaded, all of the debugfs files are cleaned up, and any memory allocated is freed.
- Note: Besides the file's rights, the debug directory itself need to be readable by everyone. There's no option for that, so let's use that old good chown !

Turn in

- Your code
- Some proof

Chapter XIII

Assignment 08

Great job with that debugfs assignment. Let's calm down with some coding style !

Todo

- Take the following file, fix the coding style, and fix the behaviour of the code.

Turn in

- Your C file, style checked and fixed.

`Butt weight, what is the code supposed to do ?`

It's part of the assignment. Have fun !

```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, but man...
Anyway, let's do some real code.

Todo

- 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 code, with a Makefile

Not a hard assignment, but really tricky.
Read some docs about mountpoints, directory listing and linked-list loop in the Kernel.
And have fun :)