1 The structure of a Linux kernel module

1.1 Install XV6 on RPI2

Before we get our hands on Linux kernel modules, we first install XV6, which will be useful for us to understand the concepts of OS through reading the XV6 code and its execution on Raspberry Pi (RPI).

Download, compile, and install **XV6**:

```
$ git clone https://github.com/zhiyihuang/xv6_rpi2_port
$ cd xv6_rpi2_port
$ make loader
$ sudo cp kernel7.img /boot/kernel-xv6.img
```

Then follow **README** in the current directory to update **/boot/config.txt** and reboot. You will see XV6 running with a simple command shell. You may type commands like **ls**, **cat**, etc if there is a serial cable connecting to a virtual terminal such as **minicom** or **CoolTerm**.

You will be provided a USB to TTL Serial Cable for the virtual terminal. You have to open the lid to connect the cable to the GPIO pins of the Pi. **Warning:** before opening the lid, you MUST unplug the power cable of the Pi; otherwise you might damage the Pi. When you handle the pins, don't touch any metal on the board except the metal of the USB ports and network port; otherwise you might damage the Pi with static electricity.

Use only the black, white and green wires of the cable. There are two rows of GPIO pins on the board of Raspberry Pi. Connect them to the outer row of the GPIO pins. The pin connections are as follows. The black wire connects to the third pin from the right, the white wire connects to the fourth pin, and the green wire connects to the fifth pin from the right.

1.2 Compile and load a module

Kernel modules are code components that can be loaded into and unloaded from the kernel dynamically. Modules enable a modular design of the kernel and can help slow down the kernel expansion. You are going to use modules to learn kernel programming, which is more flexible and less error prone than modifying the complicated kernel source tree.

You are provided with a working template module under the skelenton code 02/temp. Note that this template is very important for you to complete your first assignment, especially the **temp_init_module** and **temp_exit_module** functions which you should understand in order to initialize your device properly. We will discuss the details of this template module shortly.

Under the skelenton code 02/temp, compile the module:

```
$ make
```

Load the module:

```
$ sudo insmod ./temp.ko
```

Run **dmesg** to see if the module is loaded successfully: You should find the message "Hello world from Template Module" and etc.

To unload the module, use:

```
$ sudo rmmod temp
```

If a user-space program wants to access a device, the device needs to have a device node under the **/dev** directory. In our case, **/dev/temp** is automatically created after the module is loaded.

Just in case a device node is not automatically created, you may use the following command to create a node of a device:

```
$ sudo mknod [-m MODE] /dev/<node name> <device type> <major number> <minor number>
```

where the device type can be one of:

- **b** block device
- **c** character device
- **n** network device
- **t** tty device

and the optional argument **-m** specified the file permission of the node.

If you want to change the permission or owner of the node, use:

```
$ sudo chmod 666 /dev/temp
$ sudo chown pi:pi /dev/temp
```

Alternatively, if you want the permission to be changed automatically whenever you add the device, you can create a udev rule in a file called **10-local-temp.rules** under **/etc/udev/rules.d** with the following line:

```
KERNEL==“temp”, MODE=“0666"
```

After the module is loaded, you can find out the device major number allocated to the module using:

```
$ cat /proc/devices | grep temp
249 temp
```

If **/dev/temp** is NOT automatically created, you can use this number to create a device file with **mknod**:

```
$ sudo mknod -m 666 /dev/temp c 249 0
```

Then the user-space program can open the device with **open()** like it open any filesystems. Here is an example:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

.....
int fd = open("/dev/temp", O_RDWR);
.....
```

1.3 The Makefile for module compilation

The Makefile file is very simple, since it just jumps into the kernel tree to do the compilation. You should have installed the kernel source before any module compilation.

This is an example of a Makefile:

```
MODULE_NAME = temp

obj-m   := $(MODULE_NAME).o

KDIR    := /lib/modules/$(shell uname -r)/build
PWD     := $(shell pwd)

all: module

module:
        $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
        $(MAKE) -C $(KDIR) M=$(PWD) clean
```

The first line in the Makefile:

```
MODULE_NAME = temp
obj-m    := $(MODULE_NAME).o
```

specifies the module name temp and the target. It assumes the source file has the name temp.c. If you have a different file name, e.g. test.c, you will need the following line:

```
$(MODULE_NAME)-objs := test.o
```

If you have multiple source files for the module, you should have:

```
$(MODULE_NAME)-objs := src1.o src2.o src3.o
```

The next line specifies the kernel source that the module is going to be compiled with:

```
KDIR    := /lib/modules/$(shell uname -r)/build
```

/$(shell uname -r) will return the current running kernel source. The module will be compiled with the same options and flags as the modules in the kernel source tree. You can compile your module with other kernel sources by changing KDIR, but you have to make sure you load your module into the same release kernel that it is compiled with.

The next line specifies the location of the module source:

```
PWD  := $(shell pwd)
```

$(shell pwd) returns the current working directory. You can simply set the source directory to any absolute pathname such as /home/foo/test. The target default and clean are commands for compilation and cleanup respectively.

1.4 About the template module

This module is a character device driver for a simple and naïve memory device. We will talk about the details later. Here we are going to explain a few module related code.

You need include a number of header files depending on which kernel functions the module calls. For module compilation, you often should include the following header files:

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
```

The macros MODULE_XXX allow the programmer specify the module related information. For example, MODULE_LICENSE specifies the license of the module. If it is not an open source license, the module will be marked as "tainted", in which case it will function normally but the kernel developers will not be keen to help.

You may specify module parameters using:

```
module_param(major, int, S_IRUGO);
MODULE_PARM_DESC(major, "device major number");
```

The first argument in "module_param" is the variable name, the second is type, and the third is permission mask which is used for an accompanying entry in the sysfs file system. You can find the parameter under **/sys/module/temp**. If you are not interested in sysfs, you can put 0 there. The S_IRUGO (0444) in temp.c means read permission for all users. If you want the entry both readable and writable for all users, you can use S_IRUGO | S_IWUGO. You can find other related macros from include/linux/stat.h from the linux source tree.

When the major parameter is declared, the module can be loaded as:

```
$ sudo insmod ./temp.ko major=250
```

where the variable major will be initialized as 250 in the module.

At the end of temp.c, callback functions for module initialization and cleanup are declared:

```
module_init(temp_init_module);
module_exit(temp_exit_module);
```

1.5 Basics of a device driver

In Linux kernel, device drivers are divided into different classes:

- character
- block
- network
- tty

Different interfaces are provided by the Linux kernel API for each class of device drivers. In COSC440, you will work with character devices.

Each driver is uniquely identified by a **major number** and a **minor number**. The major number identifies the device class or group of the same type, such as a controller for several terminal. The minor number identifies a specific device, such as a single terminal.

1.5.1 Initialization, cleanup and character device

As explained above, a kernel module must be loaded before use, and when a kernel module is loaded, init() will be called to set up the class.

First, a driver needs to be registered with:

```
int register_chrdev_region(dev_t from, unsigned count, char *name);
```

where:

- **from** is the first in the desired range of device numbers; must include the major number
- **count** is the number of consecutive device numbers required
- **name** is the device name

If the major number of the device is not known, the system can dynamically allocate one. In this case, this function should be used instead:

```
int alloc_chrdev_region (dev_t      *dev,
                         unsigned    baseminor,
                         unsigned    count,
                         const char *name);
```

After a major number is dynamically allocated, the device name and the allocated number can be found in /proc/devices.

Then the function:

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

is called to initialize **cdev** (the internal structure of your character device) and register the file operation struct (see next section).

After this step, init() needs to set the **owner** field:

```
struct cdev cdev;

.....
cdev.owner = THIS_MODULE;
```

If a major number is registered with the device during init(), the driver module needs to free the major number with:

```
void unregister_chrdev_region (dev_t from, unsigned count);
```

while it is unloaded.

During initialization, each initialization step should check return value. A negative return value indicates an error, Then init() needs to **undo** all previously completed initialization steps in the **reverse** order, and return a suitable errno code. This will be more important as you will learn other resource allocation functions later in this course.

Similarly, the exit() function, which is called when the module is unloaded, needs to free up resources allocated in init() in the reverse order.

Since this is a kernel module, and we are now in the kernel space, if the cleanup steps are not properly done, resources could remain occupied after the module is unloaded, until the system is rebooted.

To automatically create a device node, you must create a class using:

```
#include <linux/device.h>

struct class *class_create(struct module *owner, const char *name);
```

where owner is usually set to THIS_MODULE and name will be the name of the class, which does not have to be same as the module name.

Then the device node is created using:

```
struct device *device_create(struct class *cls, struct device *parent, dev_t devt,
                             const char **fmt...);
```

In this way, your device name **temp** automatically appear under **/dev**. More details of these functions will be explained in Lab 6.

Note
Please have a look at the example kernel module code at the end of this section to see how it handles initialization and cleanup steps. They are very important for your first assignment.

1.5.2 File operations

The kernel API provides a number of file operations for device drivers. The main file operation-related functions are listed below:

open()

```
int (*open)(struct inode *inode, struct file *filp);
```

is provided for a driver to initialize a sub-device. If implemented, it may perform the following tasks:

- Check for device-specific errors (such as device-not-ready or similar hardware problems)
- Initialize the device if it is being opened for the first time
- Update the **f_op** pointer, if necessary
- Allocate and fill any data structure to be put in **filp→private_data**

If open() is not implemented, opening the device always succeeds, but your driver is not notified.

release()

```
int (*release)(struct inode *inode, struct file *filp);
```

releases the device. any resources allocated in open() must be released in this function.

read()

```
ssize_t (*read)(struct file *filp, char __user *buf,
                size_t count, loff_t *offp);
```

is used to retrieve data from the device to the user-space **buf**. A null pointer in this position causes the read system call to fail with -EINVAL ("Invalid argument"). A nonnegative return value represents the number of bytes successfully read(the return value is a "signed size" type, usually the native integer type for the target platform).

write()

```
ssize_t write(struct file *filp, const char _ _user *buff,
              size_t count, loff_t *offp);
```

sends data from the user-space **buf** to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if nonnegative, represents the number of bytes successfully written.

llseek()

sets the file offset of the device to the specified value. Will be covered in lab 3.

ioctl()

allows you to implement special functions for your device that is not available in existing system calls. Will be covered in lab 6.

Registering your file operation functions

Implementations of above file operation functions are registered by struct file_operation:

```
#include <linux/module.h>

struct file_operations fops = {
    .owner = THIS_MODULE,
    .llseek = temp_llseek,
    .read = temp_read,
    .write = temp_write,
    .unlocked_ioctl = temp_ioctl,
    .open = temp_open,
    .release = temp_release,
};
```

which is in turn registered by the function:

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

where **cdev** is the cdev structure you declares (statically), and **fops** is the struct holding pointers to your file operation-related functions.

init() need to call cdev_init(). Then it needs to add the character device to the system by:

```
int cdev_add (struct cdev *p, dev_t dev, unsigned count);
```

where:

- **p** points to the cdev structure
- **dev** is the first device number for which this device is responsible

- **count** is the number of consecutive minor numbers corresponding to this device

Here is an example of a real kernel module:

```
/*-----------------------------------------------------------------------------*/
/* File: tem.c                                                                 */
/* Date: 13/03/2006                                                            */
/* Author: Zhiyi Huang                                                         */
/* Version: 0.1                                                                */
/*-----------------------------------------------------------------------------*/

/* This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version
 * 2 of the License, or (at your option) any later version.
 */

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/types.h>
#include <linux/proc_fs.h>
#include <linux/fcntl.h>
#include <linux/aio.h>
#include <linux/uaccess.h>

#include <linux/ioctl.h>
#include <linux/cdev.h>
#include <linux/device.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Zhiyi Huang");
MODULE_DESCRIPTION("A template module");


/* The parameter for testing */
int major=0;
module_param(major, int, S_IRUGO);
MODULE_PARM_DESC(major, "device major number");

#define MAX_DSIZE       3071
struct my_dev {
        char data[MAX_DSIZE+1];
        size_t size;              /* 32-bit will suffice */
        struct semaphore sem;     /* Mutual exclusion */
        struct cdev cdev;
        struct class *class;
        struct device *device;
} *temp_dev;

int temp_open (struct inode *inode, struct file *filp)
{
        return 0;
}

int temp_release (struct inode *inode, struct file *filp)
{
        return 0;
```

```
}

ssize_t temp_read (struct file *filp, char __user *buf, size_t count,loff_t *f_pos)
{
        int rv=0;

        if (down_interruptible (&temp_dev->sem))
                return -ERESTARTSYS;
        if (*f_pos > MAX_DSIZE)
                goto wrap_up;
        if (*f_pos + count > MAX_DSIZE)
                count = MAX_DSIZE - *f_pos;
        if (copy_to_user (buf, temp_dev->data+*f_pos, count)) {
                rv = -EFAULT;
                goto wrap_up;
        }
        up (&temp_dev->sem);
        *f_pos += count;
        return count;

wrap_up:
        up (&temp_dev->sem);
        return rv;
}

ssize_t temp_write (struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
        int count1=count, rv=count;

        if (down_interruptible (&temp_dev->sem))
                return -ERESTARTSYS;

        if (*f_pos > MAX_DSIZE)
                goto wrap_up;
        if (*f_pos + count > MAX_DSIZE)
                count1 = MAX_DSIZE - *f_pos;

        if (copy_from_user (temp_dev->data+*f_pos, buf, count1)) {
                rv = -EFAULT;
                goto wrap_up;
        }
        up (&temp_dev->sem);
        *f_pos += count1;
        return count;

wrap_up:
        up (&temp_dev->sem);
        return rv;
}

long temp_ioctl (struct file *filp, unsigned int cmd, unsigned long arg)
{

        return 0;
}

loff_t temp_llseek (struct file *filp, loff_t off, int whence)
{
        long newpos;

        switch(whence) {
        case SEEK_SET:
```

```
                newpos = off;
                break;

        case SEEK_CUR:
                newpos = filp->f_pos + off;
                break;

        case SEEK_END:
                newpos = temp_dev->size + off;
                break;

        default: /* can't happen */
                return -EINVAL;
        }
        if (newpos<0 || newpos>MAX_DSIZE) return -EINVAL;
        filp->f_pos = newpos;
        return newpos;
}

struct file_operations temp_fops = {
        .owner =     THIS_MODULE,
        .llseek =    temp_llseek,
        .read =      temp_read,
        .write =     temp_write,
        .unlocked_ioctl = temp_ioctl,
        .open =      temp_open,
        .release =   temp_release,
};


/**
 * Initialise the module and create the master device
 */
int __init temp_init_module(void){
        int rv;
        dev_t devno = MKDEV(major, 0);

        if(major) {
                rv = register_chrdev_region(devno, 1, "temp");
                if(rv < 0){
                        printk(KERN_WARNING "Can't use the major number %d; try atomatic  ↩
                            allocation...\n", major);
                        rv = alloc_chrdev_region(&devno, 0, 1, "temp");
                        major = MAJOR(devno);
                }
        }
        else {
                rv = alloc_chrdev_region(&devno, 0, 1, "temp");
                major = MAJOR(devno);
        }

        if(rv < 0) return rv;

        temp_dev = kmalloc(sizeof(struct my_dev), GFP_KERNEL);
        if(temp_dev == NULL){
                rv = -ENOMEM;
                unregister_chrdev_region(devno, 1);
                return rv;
        }

        memset(temp_dev, 0, sizeof(struct my_dev));
        cdev_init(&temp_dev->cdev, &temp_fops);
```

```
        temp_dev->cdev.owner = THIS_MODULE;
        temp_dev->size = MAX_DSIZE;
        sema_init (&temp_dev->sem, 1);
        rv = cdev_add (&temp_dev->cdev, devno, 1);
        if (rv < 0) {
                unregister_chrdev_region(devno, 1);
                kfree(temp_dev);
                printk(KERN_WARNING "Error %d adding device temp", rv);
                return rv;
        }

        temp_dev->class = class_create(THIS_MODULE, "temp");
        if(IS_ERR(temp_dev->class)) {
                cdev_del(&temp_dev->cdev);
                unregister_chrdev_region(devno, 1);
                kfree(temp_dev);
                rv = PTR_ERR(temp_dev->class);
                printk(KERN_WARNING "%s: can't create udev class %d\n", "temp", rv);
                return rv;
        }

        temp_dev->device = device_create(temp_dev->class, NULL,
                                         devno, NULL, "temp");
        if(IS_ERR(temp_dev->device)){
                class_destroy(temp_dev->class);
                cdev_del(&temp_dev->cdev);
                unregister_chrdev_region(devno, 1);
                kfree(temp_dev);
                rv = PTR_ERR(temp_dev->device);
                printk(KERN_WARNING "%s: can't create udev device %d\n", "temp", rv);
                return rv;
        }

        printk(KERN_WARNING "Hello world from Template Module\n");
        printk(KERN_WARNING "temp device MAJOR is %d, dev addr: %lx\n", major, (unsigned  ↩
            long)temp_dev);

  return 0;
}


/**
 * Finalise the module
 */
void __exit temp_exit_module(void){
        device_destroy(temp_dev->class, MKDEV(major, 0));
        class_destroy(temp_dev->class);
        cdev_del(&temp_dev->cdev);
        kfree(temp_dev);
        unregister_chrdev_region(MKDEV(major, 0), 1);
        printk(KERN_WARNING "Good bye from Template Module\n");
}


module_init(temp_init_module);
module_exit(temp_exit_module);
```

1.6 Reference

1. Writing Linux Device Drivers, by Jerry Cooperstein

2. README under linux source tree
3. UBUNTU INSTALLATION GUIDE