Project 1 Module program

- 5130309717
- 朱文豪

I. Compile a kernel and run it in the system

1. Download The Kernel

The first step is to download the kernel from kernel.org. You may choose the latest stable version or some other versions you like. However, you'd better choose close versions, for example, **4.3.1** and **4.3.2**.

```
$ cd /usr/src/
$ sudo wget https://cdn.kernel.org/pub/linux/kernel/.../linux-4.X.X.tar.xz
```

2. Extract The kernel

```
$ sudo tar -xzvf linux-4.X.X.tar.gz or
$ sudo tar -xjvf linux-4.X.X.tar.bz2 or
$ sudo tar -xJvf linux-4.X.X.tar.xz
```

3. Compile The Linux Kernel

```
$ cd /usr/src/linux-4.X.X
$ sudo make clean
$ sudo make mrproper
```

> [Note] make clean and make mrproper can be passed with the first compilation.

```
$ sudo make xconfig
```

> [Note] ‘make xconfig’ is not the only way to configure the kernel, however, it may be the best choice for people who are not familiar with Linux kernel.
>
> [Note] You may add “NTFS file system support” in the configure menu if you use the virtual machine under Windows.

Now, you can compile the kernel. **Be patient**. ☺

```
$ sudo make
$ sudo make modules_install
$ make install
```

4. Update Boot Setting

Update your GRUB (or LILO), so that you can choose the kernel version when booting.

```
$ cd /boot
$ sudo mkinitrd -o initrd.img-<kernel version>
#or under ubuntu
sudo mkinitramfs <kernel version> -o initrd.img-<kernel version>
```

Everthing has been done!
Reboot the system and enjoy the new kernel.
(or maybe a bunch of problems. LOL!)

5. Remove Compiled Kernel

What if we want to remove the kernel previously compiled ?
It’s not complicated.

```
$ sudo rm -rf /boot/vmlinuz*KERNEL-VERSION*
$ sudo rm -rf /boot/initrd*KERNEL-VERSION*
$ sudo rm -rf /boot/System-map*KERNEL-VERSION*
$ sudo rm -rf /boot/config-*KERNEL-VERSION*
$ sudo rm -rf /lib/modules/*KERNEL-VERSION*/
$ sudo update-grub
```

II. Module 1

– Load/unload the module can output some info

1. A Brief Introduction

What's a kernel module?

> (wiki) An object file that contains code to extend the running kernel
>
> (RedHat) Modules are pieces of code that can be loaded and unloaded into the kernel upon demand

Why do we need module?

- Advantages
  - Save memory cost
  - Allowing the dynamic insertion and removal of code from the kernel at run-time
- Disadvantages
  - Fragmentation Penalty -> decrease memory performance

How to find current loadable modules?

```
$ lsmod
```

All of the current loadable modules' information will be displayed, including their name, size and used by what.

2. Write A Simple Module

An example of 'Hello World!'

```c
/*
 *  hello.c
 */
#include <linux/module.h>   /* Needed by all modules */
#include <linux/kernel.h>   /* Needed for KERN_INFO */

int init_module(void)
{
    printk(KERN_INFO "Hello DELVIN\n");
    return 0;
}
/*A non 0 return means init_module failed; module can't be loaded.*/
```

```
void cleanup_module(void)
{
    printk(KERN_INFO "Goodbye!\n");
}
```

Be aware that `printk` function contains an `KERN_INFO` , what dose this mean?
`KERN_INFO` defines the level of log message. More details are shown below:

```
#define KERN_EMERG    "<0>"    /* system is unusable */
#define KERN_ALERT    "<1>"    /* action must be taken immediately */
#define KERN_CRIT     "<2>"    /* critical conditions */
#define KERN_ERR      "<3>"    /* error conditions */
#define KERN_WARNING  "<4>"    /* warning conditions */
#define KERN_NOTICE   "<5>"    /* normal but significant */
#define KERN_INFO     "<6>"    /* informational */
#define KERN_DEBUG    "<7>"    /* debug-level messages */
```

Write a Makefile

After writing the code of your first module, how can we compile it ? A simple version of the `Makefile` is shown bellow:

```
obj-m:= hello.o
KDIR:= /lib/modules/$(shell uname -r)/build
PWD:= $(shell pwd)
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```

The model will be compiled under your current folder.

Test the module

```
$ make
$ sudo insmod hello.ko   #insert module
$ lsmod                  #list current loadable modules
$ modinfo hello.ko       #list the information of a module
$ sudo rmmod hello.ko    #remove module
```

But where can we see the `printk` output ?
Under Ubuntu, there are mainly two ways to display the message.

```
$ dmesg
#or
$ cat /var/log/syslog
```

Find what you see ? Congratulations, you are now the author of Linux kernel code.

III. Module 2

– Module accepts a parameter (an integer)

– Load the module, output the parameter's value

Last time we have written a simple module, what if we want to pass a parameter to the module in command line? We're used to `argc/argv`, however, kernel programming is different.

In order to allow parameter passing, we need to declare the variables as global and then use the `module_param()`, (which is defined in `linux/moduleparam.h`) to set up. Then, when we excute `insmod`, parameters can be included like `$ sudo insmod hello.ko myint=5`.

A simple example is shown bellow:

```
 #include <linux/module.h>  /* Needed by all modules */
  #include <linux/kernel.h> /* Needed for KERN_INFO */

  MODULE_LICENSE("GPL");
  MODULE_AUTHOR("Wenhao Zhu");

  static long myint = 5130309717;
  static char *mystring = "questionmark";

  module_param(myint, long, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
  MODULE_PARM_DESC(myint, "An integer");/*description will be shown in the
modinfo*/

  module_param(mystring, charp, 0000);/*Use numbers to represent permmisons
directly*/
  MODULE_PARM_DESC(mystring, "A character string");


  int init_module(void)
  {
    printk(KERN_INFO "Hello ZWH\n");
    printk(KERN_INFO "My student id is an integer: %ld\n", myint);
          printk(KERN_INFO "mystring is a string: %s\n", mystring);
    return 0;
  }

  void cleanup_module(void)
  {
    printk(KERN_INFO "Goodbye world.\n");
  }
```

`module_param()` takes 3 arguments: the name of the variable, its type and permissions for the corresponding file in sysfs. If the permission is not 0, you can find the parameter file under `/sys/module/YOUR_MODULE_NAME/parameters/`

You can now pass the parameters!

```
$ make
$ sudo insmod hello.ko myint=12321 mystring="Hello"
$ dmesg
```

IV. Module 3

– Module creates a proc file, reading the proc file returns some info

Using sequence read is very easy for read only proc files. In newer version, some proc api are changed.

```
  #include <linux/module.h>   /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

#define proc_name  "myproc"

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Wenhao Zhu");

struct proc_dir_entry *entry;

static int hello_proc_show(struct seq_file *m, void *v) {
  seq_printf(m, "Hello this is a new proc!\n");
  return 0;
}

static int hello_proc_open(struct inode *inode, struct  file *file) {
  return single_open(file, hello_proc_show, NULL);
}

static const struct file_operations hello_proc_fops = {
  .owner = THIS_MODULE,
  .open = hello_proc_open,
  .read = seq_read,
  .llseek = seq_lseek,
  .release = single_release,
};

int init_module()
```

```
  {
    entry = proc_create(proc_name, 0, NULL, &hello_proc_fops);

    if (!entry) {
        remove_proc_entry(proc_name, NULL);
        printk(KERN_ALERT "Error: Could not initialize
/proc/%s\n",proc_name);
        return -ENOMEM;
    }

    printk(KERN_INFO "/proc/%s created\n", proc_name);
    return 0;   /* everything is ok */
  }

  void cleanup_module()
  {
    remove_proc_entry(proc_name,NULL);
    printk(KERN_INFO "/proc/%s removed\n", proc_name);
  }
```