Programming Project 1

SWU-OS-计科中外 34 班

Oct 17, 2024

Experimental topic

Programming Projects in Chapter 2

Page P-1 to P-7 of Operating System Concepts (10th)

Experimental report naming format:

StudentID-name-Project-X (e.g., 2220183211-张三-Project-1)

Experimental objectives

1. (Course Objective 1) understand the basic concepts and organizational structure of the operating system, understand the characteristics of linux operating system, the user interface of linux operating system, and the general shell commands.
2. (Course objective 4) Master the loading / unloading method of linux kernel module and understand the development of operating system.

Experimental report submission

Please upload your report with naming format to the ftp server within two weeks.

1 Introduction to Linux Kernel Modules

In this project, you will learn how to create a kernel module and load it into the Linux kernel. The project can be completed using the Linux virtual machine that is available with this text. Although you may use an editor to write these C programs, you will have to use the terminal application to compile the programs, and you will have to enter commands on the command line to manage the modules in the kernel.

As you ' ll discover, the advantage of developing kernel modules is that it is a relatively easy method of interacting with the kernel, thus allowing you to write programs that directly invoke kernel functions. It is important for you to keep in mind that you are indeed writing kernel code that directly interacts with the kernel. That normally means that any errors in the code could crash the system! However, since you will be using a virtual machine, any failures will at worst only require rebooting the system.

Note

如果你的代码有问题，重启将会有效解决。因为你在编写 kernel 的代码，很多时候可能都需要 root 权限，使用 `sudo`，密码是 `111111`

2 Part I - Kernel Modules Overview

2.1 Creating Kernel Modules

The first part of this project involves following a series of steps for creating and inserting a module into the Linux kernel.

You can list all kernel modules that are currently loaded by entering the command

```
lsmod
```

This command will list the current kernel modules in three columns: name, size, and where the module is being used.

The following program (named simple.c and available with the source code for this text) illustrates a very basic kernel module that prints appropriate messages when the kernel module is loaded and unloaded.

```
/**
 * simple.c
 *
 * A simple kernel module.
 *
 * To compile, run makefile by entering "make"
 *
 * Operating System Concepts - 10th Edition
 * Copyright John Wiley & Sons - 2018
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
       printk(KERN_INFO "Loading Module\n");

       return 0;
```

(continues on next page)

(continued from previous page)

```
}

/* This function is called when the module is removed. */
void simple_exit(void) {
        printk(KERN_INFO "Removing Module\n");
}

/* Macros for registering module entry and exit points. */
module_init( simple_init );
module_exit( simple_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```

The function `simple_init()` is the module entry point, which represents the function that is invoked when the module is loaded into the kernel. Similarly, the `simple_exit()` function is the module exit point—the function that is called when the module is removed from the kernel.

The module entry point function must return an integer value, with 0 representing success and any other value representing failure. The module exit point function returns void. Neither the module entry point nor the module exit point is passed any parameters. The two following macros are used for registering the module entry and exit points with the kernel:

```
module_init(simple_init)

module_exit(simple_exit)
```

Notice how both the module entry and exit point functions make calls to the `printk()` function. `printk()` is the kernel equivalent of `printf()`, yet its output is sent to a kernel log buffer whose contents can be read by the dmesg command. One difference between `printf()` and `printk()` is that `printk()` allows us to specify a priority flag whose values are given in the `<linux/printk.h>` include file.

In this instance, the priority is `KERN INFO`, which is defined as an **informational message**.

The final lines —`MODULE LICENSE()`, `MODULE DESCRIPTION()`, and `MODULEAUTHOR()` - represent details regarding the software license, description of the module, and author. For our purposes, we do not depend on this information, but we include it because it is standard practice in developing kernel modules.

This kernel module `simple.c` is compiled using the `Makefile` accompanying the source code with this project.

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

> **Note**
>
> 关于 Makefile 怎么写, 可以查看相关的资料，例 [makefiletutorial]

To compile the module, enter the following on the command line:

```
make
```

> **Note**
>
> 注意这里需要将 `simple.c` 和 `Makefile` 在一个目录下，并且命令行要在这个目录下。不知道目前在那个目录，使用 `pwd` 可以查看。

The compilation produces several files. The file `simple.ko` represents the compiled kernel module. The following step illustrates inserting this module into the Linux kernel.

2.2 Loading and Removing Kernel Modules

Kernel modules are loaded using the insmod command, which is run as follows:

```
sudo insmod simple.ko
```

To check whether the module has loaded, enter the lsmod command and search for the module simple. Recall that the module entry point is invoked when the module is inserted into the kernel. To check the contents of this message in the kernel log buffer, enter the command

```
dmesg
```

You should see the message *Loading Module*.

Removing the kernel module involves invoking the `rmmod` command (notice that the .ko suffix is unnecessary):

```
sudo rmmod simple
```

Be sure to check with the dmesg command to ensure the module has been removed.

Because the kernel log buffer can fill up quickly, it often makes sense to clear the buffer

periodically. This can be accomplished as follows:

```
sudo dmesg -c
```

As kernel modules are running within the kernel, it is possible to obtain values and call functions that are available only in the kernel and not to regular user applications.

For example, the Linux include file <linux/hash.h> defines several hashing functions for use within the kernel. This file also defines the constant value GOLDEN_RATIO_PRIME (which is defined as an unsigned long).

This value can be printed out as follows

```
printk(KERN_INFO "%lu\n", GOLDEN_RATIO_PRIME);
```

As another example, the include file <linux/gcd.h> defines the following function

```
unsigned long gcd(unsigned long a, unsigned b);
```

which returns the greatest common divisor of the parameters a and b. Once you are able to correctly load and unload your module, complete the following additional steps:

1. Print out the value of GOLDEN_RATIO_PRIME in the simple_init() function.
2. Print out the greatest common divisor of 3,300 and 24 in the simple_exit() function.

As compiler errors are not often helpful when performing kernel development, it is important to compile your program often by running make regularly. Be sure to load and remove the kernel module and check the kernel log buffer using dmesg to ensure that your changes to simple.c are working properly.

In Section 1.4.3, we described the role of the timer as well as the timer interrupt handler. In Linux, the rate at which the timer ticks (the tick rate) is the value HZ defined in <asm/param.h>. The value of HZ determines the frequency of the timer interrupt, and its value varies by machine type and architecture. For example, if the value of HZ is 100, a timer interrupt occurs 100 times per second, or every 10 milliseconds. Additionally, the kernel keeps track of the global variable jiffies, which maintains the number of timer interrupts that have occurred since the system was booted. The jiffies variable is declared in the file <linux/jiffies.h>.

1. Print out the values of jiffies and HZ in the simple_init() function.
2. Print out the value of jiffies in the simple_exit() function.

Before proceeding to the next set of exercises, consider how you can use the different values of jiffies in simple_init() and simple_exit() to determine the number of seconds that have elapsed since the time the kernel module was loaded and then removed.

2.3 Part I Assignment

Proceed through the steps described above to create the kernel module and to load and unload the module. Be sure to check the contents of the kernel log buffer using `dmesg` to ensure you have properly followed the steps.

3 Part II The `/proc` File System

The `/proc` file system is a "pseudo" file system that exists only in kernel mem ory and is used primarily for querying various kernel and per-process statistics.

```
/**
 * hello.c
 *
 * Kernel module that communicates with /proc file system.
 *
 * The makefile must be modified to compile this program.
 * Change the line "simple.o" to "hello.o"
 *
 * Operating System Concepts - 10th Edition
 * Copyright John Wiley & Sons - 2018
 */

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>

#define BUFFER_SIZE 128

#define PROC_NAME "hello"
#define MESSAGE "Hello World\n"

/**
 * Function prototypes
 */
ssize_t proc_read(struct file *file, char *buf, size_t count, loff_t *pos);

static struct file_operations proc_ops = {
        .owner = THIS_MODULE,
        .read = proc_read,
};
```

(continues on next page)

(continued from previous page)

```
/* This function is called when the module is loaded. */
int proc_init(void)
{

        // creates the /proc/hello entry
        // the following function call is a wrapper for
        // proc_create_data() passing NULL as the last argument
        proc_create(PROC_NAME, 0, NULL, &proc_ops);

        printk(KERN_INFO "/proc/%s created\n", PROC_NAME);

        return 0;
}

/* This function is called when the module is removed. */
void proc_exit(void) {

        // removes the /proc/hello entry
        remove_proc_entry(PROC_NAME, NULL);

        printk( KERN_INFO "/proc/%s removed\n", PROC_NAME);
}

/**
 * This function is called each time the /proc/hello is read.
 *
 * This function is called repeatedly until it returns 0, so
 * there must be logic that ensures it ultimately returns 0
 * once it has collected the data that is to go into the
 * corresponding /proc file.
 *
 * params:
 *
 * file:
 * buf: buffer in user space
 * count:
 * pos:
 */
ssize_t proc_read(struct file *file, char __user *usr_buf, size_t count, loff_t␣
↪*pos)
{
        int rv = 0;
        char buffer[BUFFER_SIZE];
        static int completed = 0;
```

(continues on next page)

(continued from previous page)

```
        if (completed) {
                completed = 0;
                return 0;
        }

        completed = 1;

        rv = sprintf(buffer, "Hello World\n");

        // copies the contents of buffer to userspace usr_buf
        copy_to_user(usr_buf, buffer, rv);

        return rv;
}


/* Macros for registering module entry and exit points. */
module_init( proc_init );
module_exit( proc_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Hello Module");
MODULE_AUTHOR("SGG");
```

This exercise involves designing kernel modules that create additional entries in the `/proc` file system involving both kernel statistics and information related to specific processes.

We begin by describing how to create a new entry in the `/proc` file system. The following program example (named `hello.c` and available with the source code for this text) creates a `/proc` entry named `/proc/hello`.

If a user enters the command

```
cat /proc/hello
```

the infamous *Hello World* message is returned.

In the module entry point `proc_init()`, we create the new `/proc/hello` entry using the `proc_create()` function. This function is passed `proc_ops`, which contains a reference to a struct file operations. This struct initializes the *.owner* and *.read* members. The value of *.read* is the name of the function `proc_read()` that is to be called whenever `/proc/hello` is read. Examining this `proc_read()` function, we see that the string *Hello World* is written to the variable *buffer* where *buffer* exists in kernel memory. Since `/proc/hello` can be accessed from user space, we must copy the contents of *buffer* to user space using the kernel function `copy_to_user()`. This function copies the contents of kernel memory

buffer to the variable usr buf, which exists in user space. Each time the `/proc/hello` file is read, the `proc_read()` function is called repeatedly until it returns 0, so there must be logic to ensure that this function returns 0 once it has collected the data (in this case, the string *Hello World*) that is to go into the corresponding `/proc/hello` file. Finally, notice that the `/proc/hello` file is removed in the module exit point `proc_exit()` using the function remove `proc_entry()`.

3.1 Part II - Assignment

This assignment will involve designing two kernel modules:

1. Design a kernel module that creates a `/proc` file named `/proc/jiffies` that reports the current value of jiffies when the `/proc/jiffies` file is read, such as with the command

```
cat /proc/jiffies
```

Be sure to remove `/proc/jiffies` when the module is removed.

2. Design a kernel module that creates a proc file named `/proc/seconds` that reports the number of elapsed seconds since the kernel module was loaded. This will involve using the value of jiffies as well as the HZ rate. When a user enters the command

```
cat /proc/seconds
```

your kernel module will report the number of seconds that have elapsed since the kernel module was first loaded. Be sure to remove `/proc/seconds` when the module is removed.

References

[makefiletutorial] Learn Makefiles With the tastiest examples[1]

[1] https://makefiletutorial.com/