

六 Linux 内核模块编程

实验目的

- 1、了解和熟悉编译一个基本的内核模块需要包含的元素；
- 2、掌握内核模块编程原理；
- 3、掌握模块参数传递方法。

实验环境

安装有 Linux 操作系统的计算机

实验步骤

作为宏内核结构，Linux 内核具有效率高的特点，但也有可扩展性和可维护性相对较差的不足，Linux 提供模块机制正是弥补这一缺陷。Linux 内核模块全称为“动态可加载内核模块(Loadable Kernel Module, LKM)”，是系统内核向外部提供的功能插口。

模块是具有独立功能的程序，它可以被单独编译，但不能独立运行。模块在运行时被链接到内核作为内核的一部分在内核空间运行，这与运行在用户控件的进程是不同的。模块通常有一组函数和数据结构组成，用来实现某种文件系统、驱动程序或其它内核上层功能。

一、 编写一个简单的内核模块

1. 编写模块程序

编写如下简单代码，本示例中代码文件命名“hello_module.c”。

代码清单 6.1 hello_module.c

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

static int __init hello_init(void){
    printk("This is hello_module, welcome to Linux kernel \n");
    return 0;
}

static void __exit hello_exit(void){
    printk("see you next time!\n");
}
```

```

}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mr Yu");
MODULE_DESCRIPTION("hello kernel module");
MODULE_ALIAS("hello");

```

以上代码解释如下：

(1) `#include <linux/module.h>`: **必须**。module.h 头文件包含了对模块的结构定义以及模块的版本控制，任何模块程序的编写都要包含这个头文件；

(2) `#include <linux/kernel.h>`: kernel.h 包含了常用的内核函数，如以上程序中的 `printk()` 函数；

(3) `#include <linux/init.h>`: **必须**。init.h 包含了 `module_init()` 和 `module_exit()` 函数的声明；

(4) `module_init()`: **必须**。模块加载函数，加载模块时该函数自动执行，进行初始化操作；

(5) `module_exit()`: **必须**。模块卸载函数，卸载模块时函数自动执行，进行清理操作；

(6) `MODULE_LICENSE()`: 表示模块代码接受的软件许可协议。Linux 内核是使用 GPL V2 的开源项目，其要求所有使用和修改了 Linux 内核代码的个人或组织都有义务把修改后的源代码公开，这是一个强制的开源协议，所以一般编写驱动代码都需要显示的声明和遵循本协议，否则内核 UI 发出被污染的警告；

(7) `MODULE_AUTHOR()`: 描述模块的作者信息；

(8) `MODULE_DESCRIPTION()`: 简单描述模块的用途、功能介绍等；

(9) `MODULE_ALIAS()`: 为用户控件提供的别名；

(10) `printk()`: 内核输出函数，默认打印系统文件 `/var/log/messages` 的内容。

2. 编译内核模块

编写 Makefile 文件，文件名必须为 “Makefile”

代码清单 6.2 Makefile

```
obj-m := hello_module.o

KERNELBUILD := /lib/modules/$(shell uname -r)/build
CURRENT_PATH := $(shell pwd)

all:
    make -C $(KERNELBUILD) M=$(CURRENT_PATH) modules

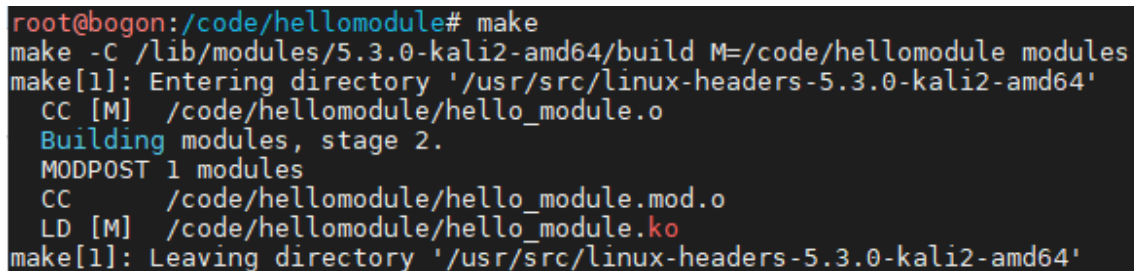
clean:
    make -C $(KERNELBUILD) M=$(CURRENT_PATH) clean
```

以上代码解释如下：

- (1) `obj-m := <模块名>.o`：定义要生成的模块名称
- (2) `KERNELBUILD := /lib/modules/$(shell uname -r)/build`：
KERNELBUILD 为自定义名称，用于指向正在运行 Linux 的内核编译目录，其中“`uname -r`”标识显示对应的内核版本；
- (3) `CURRENT_PATH := $(shell pwd)`：CURRENT_PATH 为自定义名称，用于指向当前当前目录；
- (4) `all`:编译执行的动作
- (5) `clean`:执行 `make clean` 需要的动作。“`make clean`”用于清除上次的 `make` 命令所产生的 object 文件（后缀为“.o”的文件）及可执行文件。

3. 编译

将以上两个文件(hello_module.c 和 Makefile)保存于同一目录下，如本演示中代码存放路径为“/code/hellomodule/”，**编译需在文件保存目录中进行**。



```
root@bogon:/code/hellomodule# make
make -C /lib/modules/5.3.0-kali2-amd64/build M=/code/hellomodule modules
make[1]: Entering directory '/usr/src/linux-headers-5.3.0-kali2-amd64'
CC [M] /code/hellomodule/hello_module.o
Building modules, stage 2.
MODPOST 1 modules
CC /code/hellomodule/hello_module.mod.o
LD [M] /code/hellomodule/hello_module.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.3.0-kali2-amd64'
```

图 6.1 编译

编译成功后，可看到生成的 hello_module.ko 目标文件

```
root@bogon:/code/hellomodule# ls
hello_module.c  hello_module.mod  hello_module.mod.o  Makefile  Module.symvers
hello_module.ko  hello_module.mod.c  hello_module.o  modules.order
```

图 6.2 模块文件

4. 检查编译模块

可通过 `file` 命令检查编译的模块是否正确，可以看到 x86-64 架构的 elf 文件，说明编译成功。

```
root@bogon:/code/hellomodule# file hello_module.ko
hello_module.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), BuildID[sha1]=768b1e67291678b9f080fb4e6da8019f618e5aab, with debug_info, not stripped
```

图 6.3 检查编译

也可通过 `modinfo` 命令进一步检查。

```
root@bogon:/code/hellomodule# modinfo hello_module.ko
filename:       /code/hellomodule/hello_module.ko
alias:          hello
description:     hello kernel module
author:         Mr Yu
license:        GPL
author:         everyone
license:        GPL
depends:
retpoline:      Y
name:           hello_module
vermagic:       5.3.0-kali2-amd64 SMP mod_unload modversions
```

图 6.4 检查编译

5. 插入模块

通过 `insmod` 命令插入模块，完成插入后可通过 `lsmod` 命令查看当前模块是否已经被加载到系统中。

```
root@bogon:/code/hellomodule# insmod hello_module.ko
root@bogon:/code/hellomodule# lsmod
Module          Size  Used by
hello_module    16384  0
vmw_vsock_vmci_transport 32768  2
vsock           40960  3 vmw_vsock_vmci_transport
intel_rapl_msr  20480  0
intel_rapl_common 28672  1 intel_rapl_msr
intel_rapl_perf  16384  0
vmw_balloon     24576  0
btusb           57344  0
btrtl           24576  1 btusb
btbcm           16384  1 btusb
btintel         28672  1 btusb
```

图 6.5 插入模块

系统加载模块后，也会在“/sys/modules”目录下新建以模块名命名的目录。

```
root@bogon:/code/hellomodule# ls /sys/module/
8250      crc32c_generic  hello_module  pciehp      soundcore
ac         crc32c_intel    hid           pci_hotplug  spidev
ac97_bus  crc32_pclmul    hid_generic  pcnet32      spurious
acpi      crc_t10dif      i2c_piix4    pcspkr       srcutree
acpihp    crc_t10dif_pclmul  i8042        printk       sr_mod
aesni_intel  cryptd          intel_idle   processor    sunrpc
aes_x86_64  cryptomgr       intel_rapl_common  psmouse     suspend
ahci       crypto_simd     intel_rapl_msr  pstore       syscall
ansi_cprng  drbg            intel_rapl_perf  random       sysrq
apparmor    drm             ip_tables      rcupdate     tcp_cubic
ata_generic  drm_kms_helper  ipv6           rcutree      thermal
ata_piix    dynamic_debug   jbd2           rfkill       ttm
autofs4     ecc              joydev         rtc_cmos     uhci_hcd
binfmt_misc  ecdh_generic    kernel         scsi_mod     usb_common
blk_cgrou   edac_core       keyboard      scsi_transport_spi  usbcore
block       ehci_hcd        libahci       sd_mod       usbhid
bluetooth   ehci_pci        libata        serio_raw    vmw_balloon
btbcm       evdev           mbcache       sg           vmwgfx
```

图 6.6 模块目录

6. 查看输出

因本演示中 `prink()` 采用默认输出等级，可通过“`dmesg`”或“`tail /var/log/messages`”命令查看输出结果。

```
root@bogon:/code/hellomodule# tail /var/log/messages
May 10 10:25:47 bogon colord[1157]: failed to get edid data: EDID length is too small
May 10 10:25:49 bogon org.freedesktop.thumbnails.Thumbnailer1[1000]: Registered thumbnailer /usr/bin/gdk-p
ixbuf-thumbnailer -s %s %u %o
May 10 10:25:49 bogon org.freedesktop.thumbnails.Thumbnailer1[1000]: Registered thumbnailer /usr/bin/gdk-p
ixbuf-thumbnailer -s %s %u %o
May 10 10:25:49 bogon udisksd[1198]: udisks daemon version 2.8.4 starting
May 10 10:25:49 bogon udisksd[1198]: failed to load module mdraid: libbd_mdraid.so.2: cannot open shared o
bject file: No such file or directory
May 10 10:25:49 bogon udisksd[1198]: Failed to load the 'mdraid' libblockdev plugin
May 10 10:25:50 bogon udisksd[1198]: Acquired the name org.freedesktop.UDisks2 on the system message bus
May 10 10:39:10 bogon lightdm[5285]: Error getting user list from org.freedesktop.Accounts: GDBus.Error:or
g.freedesktop.DBUS.Error.ServiceUnknown: The name org.freedesktop.Accounts was not provided by any .servic
e files
May 10 10:43:48 bogon kernel: [ 1113.027285] hello_module: loading out-of-tree module taints kernel.
May 10 10:43:48 bogon kernel: [ 1113.028903] This is hello_module, welcome to Linux kernel
```

图 6.7 查看模块程序结果

7. 卸载模块

卸载模块，可通过“`rmmod 模块名`”实现。

```
root@bogon:/code/hellomodule# rmmod hello_module
```

图 6.8 卸载模块

可通过 `dmesg` 命令查看结果。

```

[ 1113.027283] netto_module: loading out-of-tree module taints ke
[ 1113.028903] This is hello_module, welcome to Linux kernel
[ 4731.418055] pcnet32 0000:02:01.0 eth0: link down
[ 4741.402115] pcnet32 0000:02:01.0 eth0: link up
[ 4755.417392] pcnet32 0000:02:01.0 eth0: link down
[ 4767.431797] pcnet32 0000:02:01.0 eth0: link up
[10679.853457] see you next time!

```

图 6.9 卸载模块

二、 模块参数

Linux 内核提供一个宏来实现模块的参数传递。

```

1.  #define module_param(name, type, perm) \
2.      module_param_named(name, name, type, perm)
3.
4.  #define MODULE_PARM_DESC(_parm, desc) \
5.      _MODULE_INFO(parm, _parm, #_parm ":" desc);

```

`module_param()` 宏由 3 个参数组成, `name` 表示参数名, `type` 表示参数类型, `perm` 表示参数读写权限。

`MODULE_PARM_DESC()` 宏提供参数的简单说明, 参数类型可为 `byte`、`short`、`int`、`long`、`char`、`bool` 等类型; `perm` 指定在 `sysfs` 中相应文件的访问权限, 如设置为 0 则不会出现在 `sysfs` 文件系统中, 设置为 0644 标识 `root` 用户可修改本参数。

如以下实际代码所示 (`driver/misc/altera-stap1/altera.c`), 实际定义模块参数 `debug`, 类型是 `int`, 访问权限是 0644。参数用途是打开调试信息, 实际内核编程中常用此方法进行内核调试。

```

1.  static int debug = 1;
2.  module_param(debug, int, 0644);
3.  MODULE_PARM_DESC(debug, "enable debugging information");
4.
5.  #define dprintk(args...) if(debug){printk(KERN_DEBUG args);}

```

1. 编写以下代码

代码清单 6.3 parm_module.c

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

static int debug = 1;
module_param(debug, int, 0644);
MODULE_PARM_DESC(debug, "debugging information");

#define dprintk(args...) if(debug){printk(KERN_DEBUG args);}
static int myparm = 10;
module_param(myparm, int, 0644);
MODULE_PARM_DESC(myparm, "kernel module parameter experiment.");

static int __init parm_init(void){
    dprintk("my linux kernel module init.\n");
    dprintk("module parameter = %d\n", myparm);

    return 0;
}

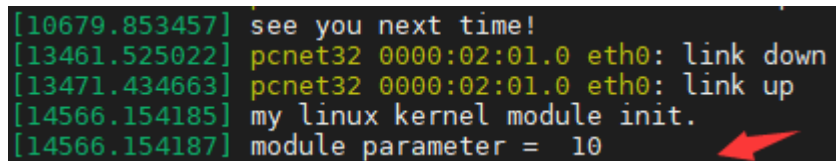
static void __exit parm_exit(void){
    printk("see you next time!\n");
}

module_init(parm_init);
module_exit(parm_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mr Yu");
MODULE_DESCRIPTION("kernel module paramter experiment");
MODULE_ALIAS("myparm");
```

2. 修改相应 Makefile 文件，编译后插入模块

通过 dmesg 查看日志信息，可发现输出以上程序中 myparm 的默认值。



```
[10679.853457] see you next time!  
[13461.525022] pcnet32 0000:02:01.0 eth0: link down  
[13471.434663] pcnet32 0000:02:01.0 eth0: link up  
[14566.154185] my linux kernel module init.  
[14566.154187] module parameter = 10
```

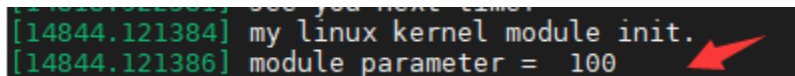
图 6.10 模块参数默认值

3. 卸载模块，赋值重新加载模块

修改参数 myparm 值为 100:

```
insmod parm_module.ko myparm=100
```

通过 dmesg 查看日志信息，可发现 myparm 值已经改变。



```
[14844.121384] my linux kernel module init.  
[14844.121386] module parameter = 100
```

图 6.11 内核模块传参

实验内容

- 1、完成简单内核模块编译，理解模块编程主要结构。
- 2、完成模块参数程序编译，理解内核模块传参方法。