专题5-内核模块开发

一、Linux内核模块基础

1.1、为什么需要内核模块

Linux内核的整体结构非常庞大，其包含的组件也非常多，如何使用这些组件呢，方法1：把所有的组件都编译进内核文件，即：zImage或bzImage，但这样会导致一个问题：占用内存过多.

有没有一种机制能让内核文件本身并不包含某组件，而是在该组件需要被使用的时候，动态地添加到正在运行的内核中呢？

内核模块具有如下特点

- 模块本身并不被编译进内核文件(zImage或 者bzImage)
- 可以根据需求，在内核运行期间动态的安 装或卸载

1.2、如何使用内核模块

1.2.1、安装模块

```
insmod /home/dnw_usb.ko
```

1.2.2、卸载模块

```
rmmod dnw_usb
```

1.2.3、查看模块

```
lsmod
```

二、Linux内核模块设计

2.1、加载函数（一般需要）

当通过insmod或者modprobe命令加载内核模块时，模块的加载函数会自动被内核执行，完成本模块的相关初始化工作。

```
static int hello_init(void)
{
    printk(KERN_WARNING"Hello world!\n");
    return 0;
}
module_init(hello_init);
```

模块加载函数必须以“module_init(函数名)”的形式被指定。它返回整型值，若初始化成功，返回0。而在初始化失败时，应该返回错误编码。

2.2、卸载函数（一般需要）

当通过rmmod命令卸载某模块时，模块的卸载函数会自动被内核执行，完成与模块加载相反的功能。

```
static void hello_exit(void)
{
    printk(KERN_WARNING"hello exit!\n");
}
module_exit(hello_exit);
```

模块卸载函数在模块卸载的时候执行，不返回任何值，必须以“module_exit(函数名)”的形式指定。

通常来说，模块卸载函数要完成与模块加载函数相反的功能，如下所示。

a）若模块加载函数注册了XXX,则模块卸载函数应该注销XXX。

b）若模块加载函数动态申请了内存，则模块卸载函数应释放该内存。

c）若模块加载函数申请了硬件资源（中断，DMA通道，I/O端口和I/O内存等）的占用，则模块卸载函数应释 放这些硬件资源。

d）若模块加载函数开启了硬件，则卸载函数中一般要关闭硬件。

2.3、头文件

所有的内核模块都需要包含<linux/init.h>，<linux/module.h>两个头文件。

2.4、模块许可证声明（必须）

许可证(LICENSE)声明描述内核模块的许可权限，如果不声明LICENSE，模块被加载时，将收到内核被污染（kernel tainted）的警告. 在Linux2.6内核中，可接受的 LICENSE包括“GPL”，“GPL v2”，“GPL and additonal rights”，“Dual BSD/GPL”，“Dual MPL/GPL”和“Proprietary”。

大多数情况下，内核模块应遵循GPL兼容许可权。

Linux2.6内核模块最常见的是以MODULE_LICENSE("Dual BSD/GPL")语句声明模块采用BSD/GPL双 LICENSE 。

2.5、模块参数（可选）

模块参数是模块被加载的时候可以被传递给它的值，它本身对应模块内部的全局变量。

2.6、模块导出符号（可选）

内核模块可以导出符号，这样其它模块可以使用本模块中的变量或函数。

2.7、模块作者等信息声明（可选）

2.8、模块的Makefile

```
obj-m := modulename.o
modulename-objs := file1.o file2.o

KDIR := /home/S5-driver/lesson7/linux-ok6410    //存放你开发板的内核代码的目录
all:
    make -C $(KDIR) M=$(PWD) modules CROSS_COMPILE=arm-linux- ARCH=arm    // -C 表示进入后面的目录中，modules 为执行的命令， M=$(PWD)表示内核模块代码的所在的目录。

clean:
    rm -f *.o *.ko *.order *.symvers
```

2.9、创建内核模块目录

在卸载模块时，必须在/lib/modules下有和内核版本对应的目录。

```
mkdir -p /lib/modules/$(uname -r)
```

三、Linux内核模块可选项

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

extern add(int a, int b);

int a = 3;
char * p;

static int hello_init()
{
    printk(KERN_WARNING"Hello world!\n");
 printk(KERN_WARNING"a = %d\n", a);
 printk(KERN_WARNING"p is %s\n", p);
    return 0;
}

int b;
static void hello_exit()
{
 b = add(1,4);
 printk(KERN_WARNING"b = %d\n", b);
    printk(KERN_WARNING"hello exit!\n");
}

module_init(hello_init);
module_exit(hello_exit);

module_param(a, int, S_IRUGO|S_IWUSR);
module_param(p, charp, S_IRUGO|S_IWUSR);

MODULE_AUTHOR("Johnson Zhou");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("A simple hello world module");
MODULE_ALIAS("A simpleST module");
MODULE_VERSION("V1.0");
```

3.1、模块申明

1、MODULE_LICENSE（”遵守的协议”）

申明该模块遵守的许可证协议，如：“GPL“、” GPL v2“等

2、MODULE_AUTHOR(“作者”)

申明模块的作者

3、MODULE_DESCRIPTION(“模块的功能描述”)

申明模块的功能
4、MODULE_VERSION("V1.0")
申明模块的版本

使用modinfo <模块名>可以查看模块的信息（如果安装了modinfo工具）

3.2、模块参数
可以用“module_param（参数名，参数类型，参数读/写权限）”为模块定义一个参数。
module_param(name,type,perm)
name:变量的名称
type:变量类型，bool:布尔型 int:整型 charp:字符串型
perm是访问权限。 S_IRUGO:读权限 S_IWUSR:写权限
例:

```
int a = 3;
char *st;
module_param(a,int, S_IRUGO) ;
module_param(st,charp, S_IRUGO) ;
```

在加载模块时使用如insmod helloworld.ko a=10 p='nihao'即可传递参数到模块中。

3.3、符号导出
内核符号的导出使用宏
EXPORT_SYMBOL(符号名)
EXPORT_SYMBOL_GPL(符号名)
说明：
其中EXPORT_SYMBOL_GPL只能用于包含GPL许可证的模块。
add.c代码：

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

int add(int a, int b)
{
 return a+b;
}

static int add_init()
{
    return 0;
}

static void add_exit()
{

}

module_init(add_init);
module_exit(add_exit);

EXPORT_SYMBOL(add);

MODULE_AUTHOR("Johnson Zhou");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_DESCRIPTION("A simple add module");
MODULE_ALIAS("A add module");
MODULE_VERSION("V1.0");
```

Makefile代码：

```
obj-m := helloworld.o add.o

KDIR := /home/S5-driver/lesson7/linux-ok6410

all:
 make -C $(KDIR) M=$(PWD) modules CROSS_COMPILE=arm-linux- ARCH=arm
```

```
clean:
 rm -f *.o *.ko *.order *.symvers
```