实验 3　Linux 动态模块与设备驱动

Linux 的动态模块

1. Linux（内核）动态模块基础知识

Linux 内核是一种 monolithic 类型的内核，即单一的大程序，内核中所有的功能部件都可以对其全部内部数据结构和例程进行访问。通过配置内核将新部件加入内核的方式非常耗时（要求重新编译内核）。如果想为一个 NCR 810 SCSI 卡配置 SCSI 驱动，但是内核中没有这个部分，那么就必须重新配置并重构内核。

Linux 提供了一种机制，可以随意动态的加载与卸载操作系统部件。

Linux 动态模块就是这样一种可在系统启动后的任何时候动态连入内核的代码块。当不再需要它时又可以将它从内核中卸载并删除。Linux 模块多指设备驱动，　如网络设备和文件系统。

Linux 为提供了两个命令：使用 insmod 来显式加载内核模块的目标代码(.o 或.ko),使用 rmmod 来卸载模块。同时内核自身也可以请求内核后台进程 kerneld 来加载与卸载模块。

2. 优点与注意事项

动态可加载模块代码的好处

- 可以让内核保持很小的尺寸同时非常灵活。
- 模块同时还可以无需重构（编译）内核并频繁重新启动来尝试运行新内核代码。
- 一旦 Linux 模块被加载则它和普通内核代码一样都是内核的一部分。

使用 Linux（内核）动态模块注意事项

- 尽管使用模块很自由，但是也有可能同时带来与内核模块相关的性能与内存损失。
- 可加载模块的代码以及额外的数据结构可能会占据一些内存，同时对内核资源的间接使用可能带来一些效率问题。
- 它们具有与其他内核代码相同的权限与职责，也就是说 Linux 内核模块可以像所有内核代码和设备驱动一样使内核崩溃。

3. 内核驱动模块的创建与加载

- 创建动态模块源码

- ❖ 修改 Makefile 文件生成编译规则
- ❖ 编译创建的模块源码，生成驱动模块
- ❖ 安装驱动模块
- ❖ 查看是否安装成功
- ❖ 使用驱动模块
- ❖ 卸载驱动模块

在编写驱动程序的时候，需要注意的一点是，老版本的内核驱动编译可以允许不写入 modinfo 数据。高版本内核是需要进行填写的，如果不写，那么将会编译不能通过。

MODULE_AUTHOR(""); 模块作者
MODULE_DESCRIPTION(""); 模块描述
MODULE_ALIAS(""); 模块别名
MODULE_LICENSE(""); 开源协议

编译内核模块的条件：已安装了 GCC 工具链，有一份内核源码，且至少被编译过一次，内核模块程序在编译过程中要使用内核源码的头文件（在 include 目录）和编译内核时生成的符号文件。

V2.4 版的模块程序组织

```
#include  <linux/kernel.h>          // 说明是个内核功能

#include  <linux/module.h>     // 说明是个模块

// 其他 header 信息

int init_module()      // 声明是一个模块

{

  ...      // 加载时,初始化模块的编码

}

  ...      // 期望实现的其它功能，如 read()、ioctl()等函数

void cleanup_module()

{

...     // 卸载时,注销模块的编码

}
```

V2.6 版的模块程序组织

```
#include <linux/init.h>                    /*必须要包含的头文件*/
#include <linux/kernel.h>
#include <linux/module.h>                  /*必须要包含的头文件*/
 // 其他 header 信息
static int mymodule_init(void)              //模块初始化函数
{
.....
}
.....                                                /*其它函数*/
static void mymodule_exit(void)             //模块清理函数
{
}
module_init(mymodule_init);                 //注册初始化函数
module_exit(mymodule_exit);                 //注册清除函数
MODULE_LICENSE("GPL");                      //模块许可声明
```

4.模块编译与加/卸载

- 模块的编译
  - V2.4

```
#gcc -O2 -g  -Wall -DMODULE -D  __KERNEL__  -c  filename.c
// filename.c 为自己编写的模块程序源代码文件
```

  - V2.6
    - 当前目录建立 Makefile 文件
    - 执行 make 即可按照 Makefile 的规定进行编译，形成.ko 模块文件
- 模块的加载
  - insmod 命令

- 如： insmod filename.ko 或 insmod filename.o

❖ 模块的查看
- lsmod
- more /proc/modules
- dmesg ——查看日志（printk）

❖ 模块的卸载
- rmmod 命令
- 如：rmmod filename
-

❖ 模块编程_示例 1（v2.4）：

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#if CONFIG_MODVERSION==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif
int init_module()
{       printk("Hello! This is a testing module!\n");
return 0;
}
void cleanup_module()
 {      printk("Sorry! The testing module is unloading
now!\n");
}
```

❖ 模块编程_示例 2（v2.6）：

```
#include <linux/init.h>
/*必须要包含的头文件*/
#include <linux/kernel.h>
#include <linux/module.h>            /*必须要包含的头文件*/
static int mymodule_init(void)                    //模块初始化函数
{
  printk("hello,my module wored! \n");  /*输出信息到内核日志*/
  return 0;
}
static void mymodule_exit(void) //模块清理函数
{
  printk("goodbye,unloading my module.\n")/*输出信息到内核日志*/
}
module_init(mymodule_init);      //注册初始化函数
module_exit(mymodule_exit);      //注册清理函数
MODULE_LICENSE("GPL");           //模块许可声明
```

Makefile 文件

```
ifneq ($(KERNELRELEASE),)
obj-m := mymodules.o        #obj-m 指编译成外部模块
else
KERNELDIR := /lib/modules/$(shell uname -r)/build  #定义一个变量，指向内核目录
PWD := $(shell pwd)
  modules:
 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules  #编译内核模块
  Endif
```

- 编译模块

  #make

  #ls

  文件列表包含 mymodules.ko

  注： mymodules.c 和 Makefile 文件应该位于同一个目录下。

5.模块源程序 modify_syscall.c

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
//original,syscall 78 function: gettimeofday
// new syscall 78 function: print "No 78 syscall has changed to hello" and return a+b
#define sys_No 78
unsigned long old_sys_call_func;
unsigned long p_sys_call_table=0xc0361860; // find in /boot/System.map-'uname -r'
asmlinkage int hello(int a,int b) //new function
{
   printk("No 78  syscall has changed to hello");
   return a+b;
}
void modify_syscall(void)
{
    unsigned long *sys_call_addr;
    sys_call_addr=(unsigned long *)(p_sys_call_table+sys_No*4);
    old_sys_call_func=*(sys_call_addr);
    *(sys_call_addr)=(unsigned long)&hello;                    // point to new function
}
void restore_syscall(void)
```

```
    {
        unsigned long *sys_call_addr;
        sys_call_addr=(unsigned long *)(p_sys_call_table+sys_No*4);
        *(sys_call_addr)=old_sys_call_func;                // point to
original function
    }
    static int mymodule_init(void)
    {
      modify_syscall();
      return 0;
    }
    static void mymodule_exit(void)
    {
      restore_syscall();
    }
    module_init(mymodule_init);
    module_exit(mymodule_exit);
    MODULE_LICENSE("GPL");
```

6.测试程序（用户程序）

- modify_old_syscall.c

```
    #include<stdio.h>
    #include<sys/time.h>
    #include<unistd.h>
    int main()
    {
        struct  timeval    tv;
        syscall(78,&tv,NULL); //before modify  syscall 78 :gettimeofday
        printf("tv_sec:%d\n",tv.tv_sec);
        printf("tv_usec:%d\n",tv.tv_usec);
       return 0;
    }
```

- modify_new_syscall.c

```
    #include<stdio.h>
    #include<sys/time.h>
    #include<unistd.h>
    int main()
    {
       int ret=syscall(78,10,20); //after modify syscall 78
       printf("%d\n",ret);
       return 0;
```

```
}
```

7.测试

编译模块

```
#make
```

加载模块

```
#insmod modify_syscall.ko
```

编译并执行用户程序

```
#./modify_old_syscall
#./modify_new_syscall
```

卸除模块

```
#rmmod modify_syscall
```

Linux 设备驱动

1. 实验目的

a) 理解 LINUX 字符设备驱动程序的基本原理；
b) 掌握字符设备的驱动运作机制；
c) 学会编写字符设备驱动程序。

2. 实验内容

编写一个简单的字符设备驱动程序，以内核空间模拟字符设备，完成对该设备的打开，读写和释放操作，并编写聊天程序实现对该设备的同步和互斥操作。

3. 实验前准备

字符设备

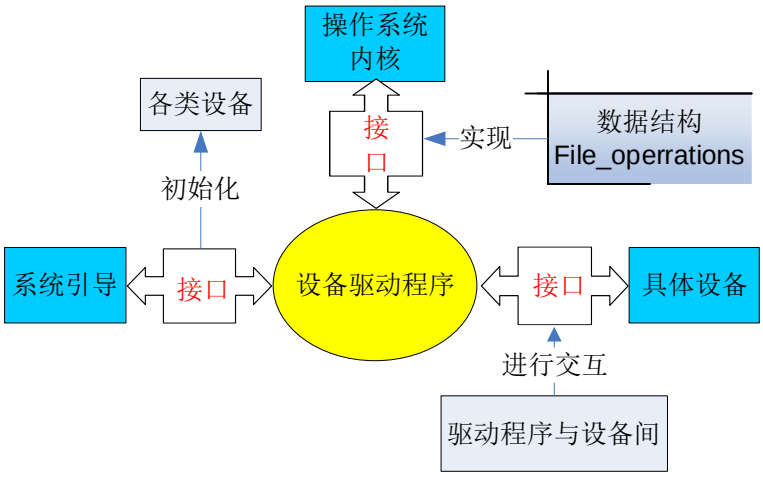
a) 键盘——键盘驱动程序
b) 串口——串口驱动程序
c) 并口——并口驱动程序
d) 显卡——显卡驱动程序

块设备

e) 磁盘——磁盘驱动程序
f) 软盘——软盘驱动程序
g) 光盘——光盘驱动程序
h) 优盘——优盘驱动程序

驱动程序与操作系统内核的接口

通过数据结构 file_operations 来完成
驱动程序与系统引导的接口
利用驱动程序对设备进行初始化
驱动程序与设备的接口
描述驱动程序如何与设备进行交互



设备文件的 VFS 处理:
普通文件——文件系统将用户的操作转换成对磁盘分区的数据块操作
设备文件——文件系统将用户的操作转换成对设备的驱动操作
VFS 中，每个文件都有一个 inode 与其对应，内核的 inode 结构中的 i_fop 成员，类型为 file_operations,file_operations 定义了文件的各种操作.
用户对文件操作是通过调用 file_operations 来实现的，或者说内核使用 file_operations 来访问设备驱动程序中的函数,为了使用户对设备文件的操作能够转换成对设备的操作，VFS 必须在设备文件打开时，改变其 inode 结构中 i_fop 成员的默认值,将其替换为与设备相关的具体函数操作。用户访问设备时，文件系统读取设备文件在磁盘上相应的 inode，并存入主存 inode 结构中,内核将文件的主设备号与次设备号写入 inode 结构中的 i_rdev 字段，并将 i_fop 字段设置成 def_chr_fops（或 def_blk_fops），以便用户对设备文件的操作转换成对设备的驱动操作。

4. 设备驱动程序的代码

模块初始化函数：该函数用来完成对所控制设备的初始化工作，并调用 register_chrdev() 函数注册字符设备。

```
static int __init globalvar_init(void)
{
   if (register_chrdev(MAJOR_NUM, " globalvar ", &globalvar_fops))

      {

        //…注册失败
```

```
    }
  else
    {
      //…注册成功
    }
}
```

模块卸载函数：需要调用函数 unregister_chrdev()。

```
static void __exit globalvar_exit(void)
{
  if (unregister_chrdev(MAJOR_NUM, " globalvar "))
      {
      //…卸载失败
      }
  Else
      {
        //…卸载成功
      }
}
```

open() 函数

对设备特殊文件进行 open()系统调用时，将调用驱动程序的 open() 函数：

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

其中参数 inode 为设备特殊文件的 inode (索引结点) 结构的指针,参数 file 是指向这一设备的文件结构的指针。open()的主要任务是确定硬件处在就绪状态、验证次设备号的合法性(次设备号可以用 MINOR(inode-> i - rdev) 取得)、控制使用设备的进程数、根据执行情况返回状态码(0 表示成功，负数表示存在

错误)等;

release()函数

当最后一个打开设备的用户进程执行 close ()系统调用时，内核将调用驱动程序的 release() 函数 ：

```
void (*release) (struct inode * ,struct file *) ;
```

release 函数的主要任务是清理未结束的输入/输出操作、释放资源、用户自定义排他标志的复位等。

read()函数

当对设备特殊文件进行 read() 系统调用时，将调用驱动程序 read()函数：

```
ssize_t (*read) (struct file *, char *, size_t, loff_t *)
```

内核空间与用户空间的内存交互需要借助函数： globalvar_read，因此设备"globalvar"的基本入口点结构变量 gobalvar_fops 赋值如下：

```
struct file_operations globalvar_fops = {

  read: globalvar_read,

write: globalvar_write,

};
```

globalvar_read 函数

```
static ssize_t globalvar_read(struct file *filp, char *buf, size_t len,
loff_t *off)

{

  …

  copy_to_user(buf, &global_var, sizeof(int));

   …

}
```

write()函数

当设备特殊文件进行 write()系统调用时，将调用驱动程序的 write()函数：

```
ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
```

内核空间与用户空间的内存交互需要借助函数： globalvar_read，因此设备"globalvar"的基本入口点结构变量 gobalvar_fops 赋值如下：

```
  struct file_operations globalvar_fops = {

    read: globalvar_read,

  write: globalvar_write,

};
```

globalvar_write 函数

```
static ssize_t globalvar_write(struct file *filp, const char *buf, size_t
len, loff_t  *off)

{

…

copy_from_user(&global_var, buf, sizeof(int));

…

}
```

ioctl()函数

ioctl() 函数是特殊的控制函数，可以通过它向设备传递控制信息或从设备取得状态信息。

```
int (*ioctl) (struct inode *, struct file *, unsigned int , unsigned long);
```

unsigned int 参数为设备驱动程序要执行的命令的代码，由用户自定义，unsigned long 参数为相应的命令提供参数，类型可以是整型、指针等。如果设备不提供 ioctl 入口点，则对于任何内核未预先定义的请求，ioctl 系统调用将返回错误（-ENOTTY，"No such ioctl fordevice，该设备无此 ioctl 命令"）。如果该设备方法返回一个非负值，那么该值会被返回给调用程序以表示调用成功。

llseek()函数

llseek()函数用来修改文件的当前读写位置，并将新位置作为（正的）返回值返回。

loff_t (*llseek) (struct file *, loff_t, int)

实验参考代码

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

#include <linux/module.h>

#include <linux/fs.h>

#include <asm/uaccess.h>

#include <linux/init.h>

MODULE_LICENSE("GPL");

#define MAJOR_NUM 290

static ssize_t globalvar_read(struct file *, char *, size_t, loff_t*);

static ssize_t globalvar_write(struct file *, const char *, size_t,
loff_t*);

struct file_operations globalvar_fops ={

 read: globalvar_read,

write: globalvar_write

         };

static int global_var = 0;

static int init_mymodule(void){

int ret;

ret = register_chrdev(MAJOR_NUM, "globalvar", &globalvar_fops);

if (ret){

    printk("globalvar register failure");
```

```
        }
else
        {
printk("globalvar register success");
         }
return ret;
         }
static void cleanup_mymodule(void){
unregister_chrdev(MAJOR_NUM, "globalvar");
         }
 static ssize_t globalvar_read(struct file *filp, char *buf, size_t len,
loff_t *off){
if(copy_to_user(buf, &global_var, sizeof(int))){
              return -EFAULT;
         }
            return sizeof(int);
         }
static ssize_t globalvar_write(struct file *filp, const char *buf, size_t
len, loff_t *off){
if (copy_from_user(&global_var, buf, sizeof(int))){
         return -EFAULT;
         }
         return sizeof(int);
         }
```

```
module_init(init_mymodule);
module_exit(cleanup_mymodule);
```

5. 设备驱动程序测试代码

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>
main()
{
int fd, num;
//打开"/dev/ch_device"
fd = open("/dev/ch_device", O_RDWR, S_IRUSR | S_IWUSR);
if (fd != -1 ){
//初次读 ch_device
read(fd, &num, sizeof(int));
printf("The ch_device is %d\n", num);
//写 ch_device
printf("Please input the num written to ch_device\n");
scanf("%d", &num);
write(fd, &num, sizeof(int));
//再次读 ch_device
read(fd, &num, sizeof(int));
```

```
printf("The ch_device is %d\n", num);
//关闭"/dev/ch_device"
close(fd);
   }else
{
printf("Device open failure\n");
}
}
```