# 实验 Linux设备驱动程序

**实验目的：**

1. 熟悉Linux操作系统环境；
2. 了解设备驱动的实现过程；
3. 学会Linux中设备驱动程序的编写。

**实验内容：**

编写一个模块化的字符设备驱动程序。

**实验步骤：**

**(一)Linux中的设备的概念和分类**

Linux中每个设备用“设备文件”来代表，所有对硬件设备的操作与通常的文件一样，利用标准的文件操作系统调用可以在设备上进行打开、关闭、读取、写入或控制等操作，对映射为一个特殊设备文件的设备，用户程序可以象对其它文件一样对此设备文件进行操作。

Linux 将设备区分为三种基本设备类型：字符设备，块设备和网络设备。本次实验重点了解字符设备驱动的实现。

字符设备是一种可以当作一个字节流来有序存取的设备(如同一个文件)，无缓冲且只能顺序存取，由一个字符驱动负责实现这种行为。字符驱动常常至少实现open，close，read和write系统调用，例如文本控制台(/dev/console)和串口(/dev/ttyS0)。字符设备通过文件系统结点来存取，例如/dev/tty1和/dev/lp0。

**(二)Linux中的设备的查看**

设备由一个主设备号和一个次设备号标识。主设备号唯一标识了设备类型，即设备驱动类型，它是设备表中设备表项的索引。次设备号仅由设备驱动程序解释，一般用于识别在同一主设备下的若干可能的硬件设备中I／O请求所涉及到的那个设备。

在linux的/dev目录下，使用ls –l命令可以查看设备文件的信息。首字母c表示字符设备文件，b则表示块设备文件。第5列数字表示主设备号，第6列表示次设备号。

|  |
| --- |
| **[root@www hello]# cd /dev**  **[root@www dev]# ls -l**  总计 0  **……**  crw-r----- 1 root kmem 1, 1 09-03 17:35 mem  crw-rw---- 1 root audio 14, 2 09-03 17:36 midi  crw-rw---- 1 root audio 14, 0 09-03 17:36 mixer  crw-rw-rw- 1 root root 1, 3 09-03 17:35 null  crw------- 1 root root 4, 0 09-03 17:35 systty  crw-rw-rw- 1 root tty 5, 0 09-03 17:35 tty  crw-rw---- 1 root root 4, 0 09-03 17:35 tty0  crw--w---- 1 root tty 4, 1 09-04 20:59 tty1  brw-r----- 1 root disk 1, 0 09-03 17:35 ram0  **……** |

**(三)一个简单的Linux设备文件**

本例涉及的是内核模块而不是程序！下面的代码是一个"hello world"模块。

(1)源文件helloworld.c

|  |
| --- |
| #include <linux/init.h>  #include <linux/module.h>  MODULE\_LICENSE("Dual BSD/GPL");  static int hello\_init(void)  {  printk(KERN\_ALERT "Hello, world\n");//KERN\_ALERT 是消息的优先级。  return 0;  }  static void hello\_exit(void)  {  printk(KERN\_ALERT "Goodbye, cruel world\n");  }  module\_init(hello\_init);  module\_exit(hello\_exit); |

说明：

1. 这个模块定义了两个函数：hello\_init在模块加载到内核时被调用；hello\_exit在模块被去除时被调用；
2. 宏(MODULE\_LICENSE)是用来告知内核，该模块带有一个自由的许可证；
3. moudle\_init和module\_exit这几行使用了特别的内核宏来指出这两个函数的角色；
4. printk函数在Linux内核中定义并且对模块可用，它与标准C库函数printf的行为相似。内核需要它自己的打印函数，因为它靠自己运行，没有C库的帮助。模块能够调用printk是因为，在insmod加载了它之后，模块被连接到内核并且可存取内核的公用符号。

(2)Makefile 文件是用来告诉make命令如何编译和链接程序。

|  |
| --- |
| obj-m := helloworld.o  PWD := $(shell pwd)  all:  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  clean:  rm -rf \*.o \*~ core .\*.cmd \*.mod.c ./tmp\_version |

(3)运行程序，并观察效果。

|  |
| --- |
| **[root@www driver]# cd hello**  **[root@www hello]# ls**  helloworld.c Makefile  **[root@www hello]# make**  make -C /lib/modules/2.6.18-194.el5xen/build M=/root/driver/hello modules  make[1]: Entering directory `/usr/src/kernels/2.6.18-194.el5-xen-i686'  CC [M] /root/driver/hello/helloworld.o  Building modules, stage 2.  MODPOST  CC /root/driver/hello/helloworld.mod.o  LD [M] /root/driver/hello/helloworld.ko  make[1]: Leaving directory `/usr/src/kernels/2.6.18-194.el5-xen-i686'  **[root@www hello]# ls**  helloworld.c helloworld.mod.c helloworld.o Module.markers  helloworld.ko helloworld.mod.o Makefile Module.symvers  **[root@www hello]# insmod helloworld.ko**  **[root@www hello]# dmesg|grep world**  ……  Hello, world  **[root@www hello]# lsmod|grep hello**  helloworld 5632 0  **[root@www hello]# rmmod helloworld**  **[root@www hello]# dmesg|grep world**  ……  Hello, world  Goodbye, cruel world |

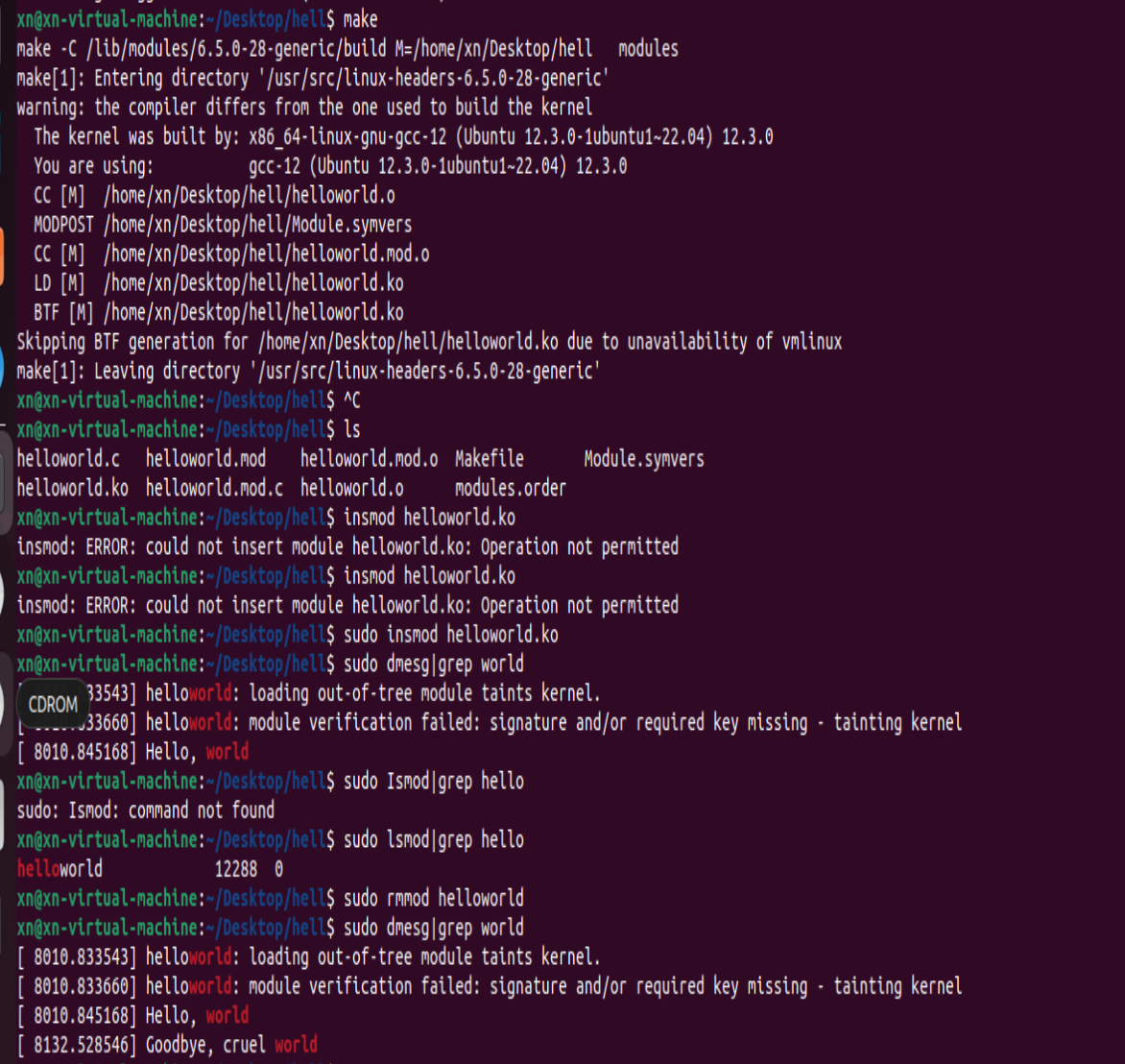
说明：

insmod：载入模块；

rmmod：删除不必要的模块；

lsmod：显示已载入系统的模块；

dmesg：显示内核缓冲区系统控制信息。



**(四)模块化的字符设备驱动程序**

在Linux系统中，字符设备驱动由如下五个部分组成：

(1)头文件。

|  |
| --- |
| #include <linux/init.h>#include <linux/module.h>  #include <linux/types.h>  #include <linux/fs.h>#include <asm/uaccess.h>  #include <linux/cdev.h> |

(2)宏和全局变量。

|  |
| --- |
| MODULE\_AUTHOR("huj");  MODULE\_LICENSE("GPL");  #define MYCDEV\_MAJOR 231 //给定的主设备号  #define MYCDEV\_SIZE 100 |

(3)字符设备驱动模块的初始化函数和卸载函数，并加载与卸载函数。

|  |
| --- |
| //模块初始化函数  static int \_\_init mycdev\_init(void)  {  int ret;  printk("mycdev module is staring..\n");  ret=register\_chrdev(MYCDEV\_MAJOR,"my\_cdev",&mycdev\_fops); //注册驱动程序  if(ret<0){  printk("register failed..\n");  return 0;  }else{  printk("register success..\n");  }  return 0;  }  //模块卸载函数  static void \_\_exit mycdev\_exit(void)  {  printk("mycdev module is leaving..\n");  unregister\_chrdev(MYCDEV\_MAJOR,"my\_cdev"); //注销驱动程序  }  module\_init(mycdev\_init);  module\_exit(mycdev\_exit); |

(4)在字符设备驱动中，需要定义一个file\_operations的实例。file\_operations是一个函数指针的集合，它的结构体中成员函数是字符设备驱动与内核的接口，是用户空间对Linux进行系统调用最终的落实者。结构体中包含对文件打开，关闭，读写，控制等的一系列成员函数。

|  |
| --- |
| // 填充 mycdev的 file operation 结构  static const struct file\_operations mycdev\_fops =  {  .owner = THIS\_MODULE,  .read = mycdev\_read,  .write = mycdev\_write,  .open = mycdev\_open,  .release = mycdev\_release,  }; |

说明：

\*owner成员不是一个操作，它是一个指向拥有这个结构的模块的指针，该成员用来在它的操作还在被使用时阻止模块被卸载。几乎所有程序中，它被简单初始化为THIS\_MODULE，在<linux/module.h>中定义的宏。

(5)将具体设备驱动的函数赋值给file\_operations的成员。

|  |
| --- |
| static int mycdev\_open(struct inode \*inode, struct file \*fp)  {  printk("mycdev module open a file..\n");  return 0;  }  static int mycdev\_release(struct inode \*inode, struct file \*fp)  {  printk("mycdev module release a file..\n");  return 0;  }  static ssize\_t mycdev\_read(struct file \*fp, char \_\_user \*buf, size\_t size, loff\_t \*pos)  {  unsigned long p = \*pos;  unsigned int count = size;  char kernel\_buf[MYCDEV\_SIZE];  int i;  memset(kernel\_buf,0,MYCDEV\_SIZE);  memcpy(kernel\_buf, "This is huj!", MYCDEV\_SIZE);  if(p >= MYCDEV\_SIZE)  return -1;  if(count > MYCDEV\_SIZE)  count = MYCDEV\_SIZE - p;  if (copy\_to\_user(buf, kernel\_buf, count) != 0) {  printk("read error!\n");  return -1;  }  for (i = 0; i < count; i++) {  \_\_put\_user(kernel\_buf[i], buf);//write 'i' from kernel space to user space's buf;  buf++;  }  printk("reader: %d bytes was read...\n", count);  return count;  }  static ssize\_t mycdev\_write(struct file \*fp, const char \_\_user \*buf, size\_t size, loff\_t \*pos)  {  return size;  } |

说明：

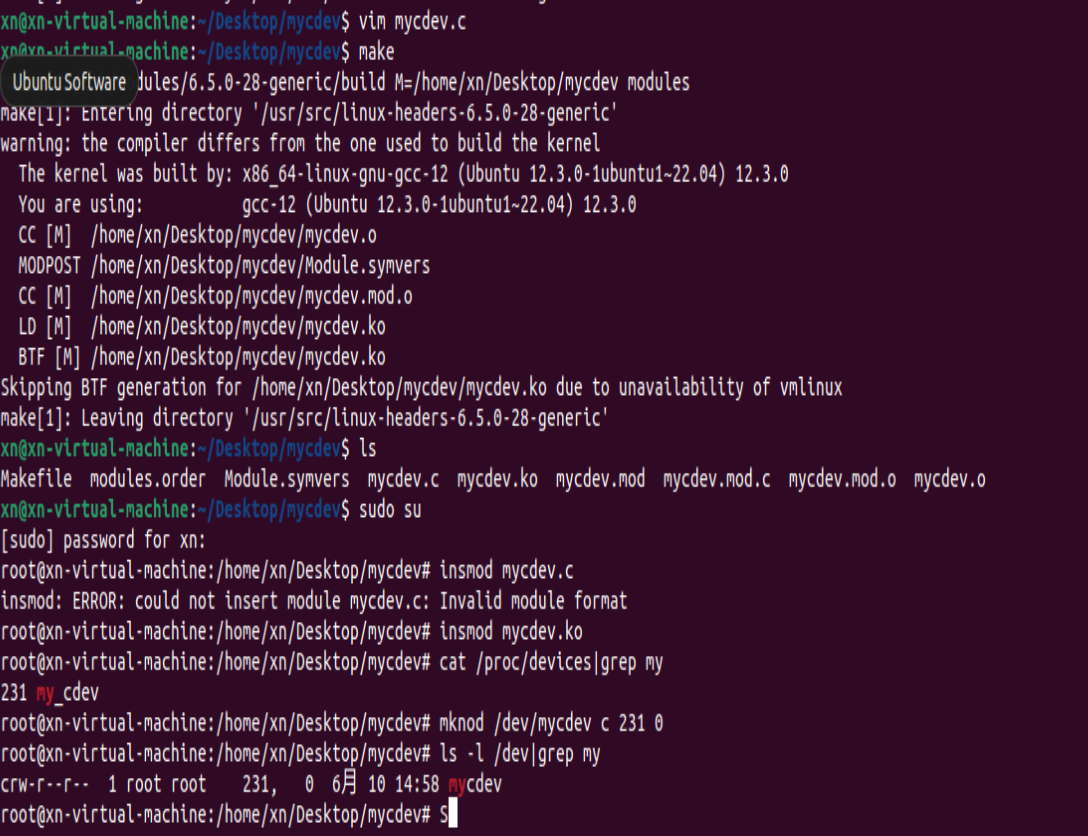
1. mycdev\_read：读函数，利用copy\_to\_user()函数让内核读取用户空间的数据，并返回访问的字节数；
2. mycdev\_write：返回写入的字节数。

编写Makefile 文件。KERNEL\_PATH为内核源代码的绝对路径。

|  |
| --- |
| obj-m:=mycdev.o  PWD:=$(shell pwd)  CUR\_PATH:=$(shell uname -r)  KERNEL\_PATH:=/usr/src/kernels/$(CUR\_PATH)  all:  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  clean:  rm -rf \*.o \*~ core .\*.cmd \*.mod.c ./tmp\_version |

编译驱动模块，加载模块，并观察设备的读和写。

|  |
| --- |
| **[root@www mycdev]# make**  make -C /lib/modules/2.6.18-194.el5xen/build M=/mnt/hgfs/share/mycdev modules  make[1]: Entering directory `/usr/src/kernels/2.6.18-194.el5-xen-i686'  CC [M] /mnt/hgfs/share/mycdev/mycdev.o  Building modules, stage 2.  MODPOST  CC /mnt/hgfs/share/mycdev/mycdev.mod.o  LD [M] /mnt/hgfs/share/mycdev/mycdev.ko  make[1]: Leaving directory `/usr/src/kernels/2.6.18-194.el5-xen-i686'  **[root@www mycdev]# ls**  Makefile Module.markers mycdev.c mycdev.mod.c mycdev.o  Makefile~ Module.symvers **mycdev.ko** mycdev.mod.o mycdev\_test.c  **[root@www mycdev]# insmod mycdev.ko**  **[root@www mycdev]# cat /proc/devices|grep my**  231 my\_cdev  **[root@www mycdev]# mknod /dev/mycdev c 231 0**  **[root@www mycdev]# ls -l /dev|grep my**  crw-r--r-- 1 root root 231, 0 12-12 11:49 mycdev |

也可编写应用层程序进行验证。

|  |
| --- |
| #include <stdio.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdlib.h>  int main( )  {  int testdev;  int i, ret;  char buf[20];  testdev = open("/dev/mycdev", O\_RDWR);  if (-1 == testdev) {  printf("cannot open file.\n");  exit(1);  }  if (ret = read(testdev, buf, 20) <15) {  printf("read error!\n");  exit(1);  }  printf("%s\n", buf);  printf("%d\n", write(testdev,buf,20));  close(testdev);  return 0;  } |

运行程序，并观察效果。

|  |
| --- |
| **[root@www mycdev]# gcc -o test mycdev\_test.c**  **[root@www mycdev]# mknod /dev/mycdev c 231 0**  **[root@www mycdev]# cat /proc/devices|grep 231**  **231 my\_cdev**  **[root@www mycdev]# ./test**  This is huj!  20  **[root@www mycdev]# dmesg**  ……  mycdev module is staring..  register success..  mycdev module open a file..  reader: 20 bytes was read...  mycdev module release a file.. |

