

八 Linux 设备驱动

实验目的

1. 理解 Linux 设备管理方式
2. 理解 Linux 设备驱动程序框架
3. 掌握 Linux 设备驱动相关 API

实验环境

安装有 Linux 操作系统的计算机

实验步骤

Linux 根据设备共有特性划分为**字符设备**、**块设备**、**网络设备**三类。字符设备以字节为单位进行 I/O 传输，如鼠标、键盘，触摸屏等；块设备以块为单位进行传输，如磁盘等；网络设备在 Linux 中比较特殊，一般作单独考虑。

Linux 内核根据各类设备抽象出一套完整的驱动框架和 API 接口，以便驱动开发者在编写驱动程序时可重复使用，如图 8.1 是 Linux 内核设备驱动组织。

Linux 设备驱动开发一般需要以下知识技能：

1. 了解 Linux 内核字符设备驱动程序的架构；
2. 了解 Linux 内核字符设备驱动相关 API；
3. 了解 Linux 内核内存管理 API；
4. 了解 Linux 内核中断管理 API；
5. 了解 Linux 内核同步和锁等相关 API；
6. 了解所编写驱动设备的相关芯片原理。

对于技能 1，需了解 Linux 字符设备驱动是如何组织、应用程序如何与驱动交互；**对于技能 2**，需掌握字符设备的描述、设备号的管理、file_operations 的实现、ioctl 交互的设计及 Linux 设备模型管理等；**对于技能 3**，设备驱动不可避免的需要和内存打交道，需要使用到 mmap 的 API 及 DMA 相关操作；**对于技能 4**，几乎所有的设备都支持中断模式，所以中断程序是设备驱动中不可或缺的部分，所以需要了解和熟悉如中断注册、编写中断程序等中断管理相关接口函数；**对于技能 5**，因为 Linux 是多进程、多用户的操作系统，且支持内核抢占，所以需考虑同步和竞争的问题；**技能 6** 需研究具体设备的芯片手册。

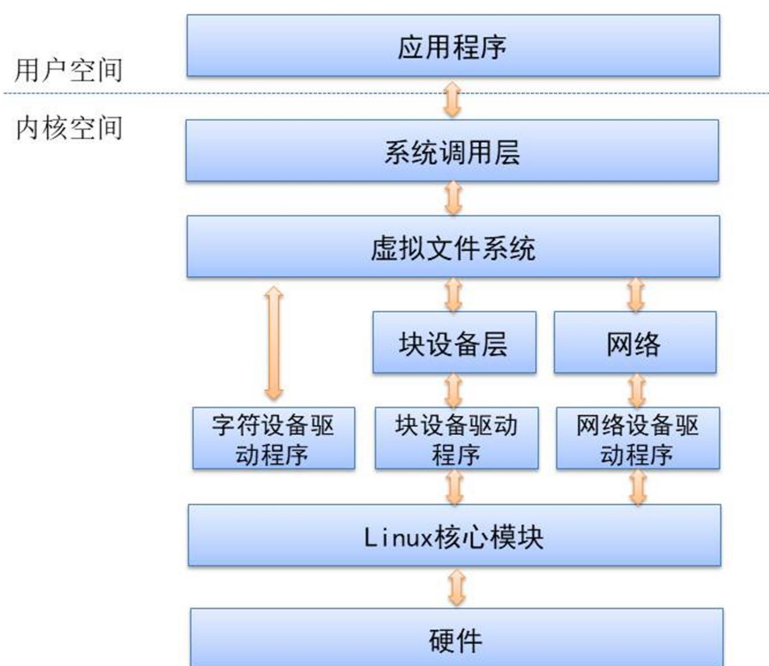


图 8.1 Linux 内核设备驱动

Linux 将设备纳入文件系统的管理框架之下，用户进程通过 file 结构与文件或设备进行交互。file 结构定义如下(include/linux/fs.h)：

```

1. struct file_operations {
2.     struct module *owner;
3.     loff_t (*llseek) (struct file *, loff_t, int);
4.     ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5.     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6.     ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
7.     ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
8.     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
9.     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
10.    int (*iterate) (struct file *, struct dir_context *);
11.    unsigned int (*poll) (struct file *, struct poll_table_struct *);
12.    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
13.    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
14.    int (*mmap) (struct file *, struct vm_area_struct *);
15.    int (*open) (struct inode *, struct file *);
16.    int (*flush) (struct file *, fl_owner_t id);
17.    int (*release) (struct inode *, struct file *);
18.    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
19.    int (*aio_fsync) (struct kiocb *, int datasync);
20.    int (*fasync) (int, struct file *, int);
21.    int (*lock) (struct file *, int, struct file_lock *);
22.    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
  
```

```

23.     unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
24.     int (*check_flags)(int);
25.     int (*flock) (struct file *, int, struct file_lock *);
26.     ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
27.     ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
28.     int (*setlease)(struct file *, long, struct file_lock **, void **);
29.     long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
30.     int (*show_fdinfo)(struct seq_file *m, struct file *f);
31. };

```

可以看出，file_operation 结构中对文件操作的函数只进行了定义，其实现有具体的驱动程序完成。如下所示为字符设备程序打开、释放资源函数模板。

```

1. static int xxx_open(struct inode *inode, struct file *fp){
2.     .....
3.     return 0;
4. }
5.
6. static int xxx_release(struct inode *inode, struct file *fp){
7.     .....
8.     return 0;
9. }

```

本示例编写一个简单的字符设备驱动，实现基本的 open、read、write 方法，并编写相应的用户控件测试程序。

1. 编写驱动程序代码

```

1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/init.h>
4. #include <linux/types.h>
5. #include <linux/fs.h>
6. #include <linux/mm.h>
7. #include <linux/sched.h>
8. #include <linux/cdev.h>
9. #include <asm/io.h>
10. #include <asm/switch_to.h>
11. #include <asm/uaccess.h>
12. #include <linux/errno.h>

```

```
13.
14. #define MYCDEV_MAJOR 300 /*主设备号, 查询/proc/devices , 选择未使用的设备号*/
15. #define MYCDEV_SIZE 1024
16.
17.
18. static int mycdev_open(struct inode *inode, struct file *fp){
19.     return 0;
20. }
21.
22. static int mycdev_release(struct inode *inode, struct file *fp){
23.     return 0;
24. }
25.
26. /*实现 read 程序*/
27. static ssize_t mycdev_read(struct file *fp, char __user *buf, size_t size, l
    off_t *pos){
28.     unsigned long p = *pos;
29.     unsigned int count = size;
30.     char kernel_buf[MYCDEV_SIZE] = "This is mycdev driver!";
31.     int i;
32.
33.     if(p >= MYCDEV_SIZE)
34.         return -1;
35.     if(count > MYCDEV_SIZE)
36.         count = MYCDEV_SIZE - p;
37.     if(copy_to_user(buf, kernel_buf, count) != 0){
38.         printk("read error!\n");
39.
40.         return -1;
41.     }
42.
43.     printk("reader: %d bytes was read.\n", count);
44.
45.     return size;
46. }
47.
48. /*实现 write 程序*/
49. static ssize_t mycdev_write(struct file *fp, const char __user *buf, size_t
    size, loff_t *pos){
50.     return size;
51. }
52.
53. /*填充 file operations 结构*/
54. static const struct file_operations mycdev_fops =
```

```
55. {
56.     .owner = THIS_MODULE,
57.     .open = mycdev_open,
58.     .release = mycdev_release,
59.     .read = mycdev_read,
60.     .write = mycdev_write,
61. };
62.
63.
64. /*模块初始化函数*/
65. static int __init mycdev_init(void){
66.     printk("mycdev driver is now starting!\n");
67.
68.     /*注册驱动程序*/
69.     int ret = register_chrdev(MYCDEV_MAJOR, "my_cdev_driver", &mycdev_fops);
70.
71.     if(ret < 0){
72.         printk("register failed!\n");
73.         return 0;
74.     }else{
75.         printk("register successfully!\n");
76.     }
77.
78.     return 0;
79. }
80.
81. /*卸载模块函数*/
82. static void __exit mycdev_exit(void){
83.     printk("mycdev driver is now leaving!\n");
84.
85.     unregister_chrdev(MYCDEV_MAJOR, "my_cdev_driver");
86. }
87.
88. module_init(mycdev_init);
89. module_exit(mycdev_exit);
90. MODULE_LICENSE("GPL");
```

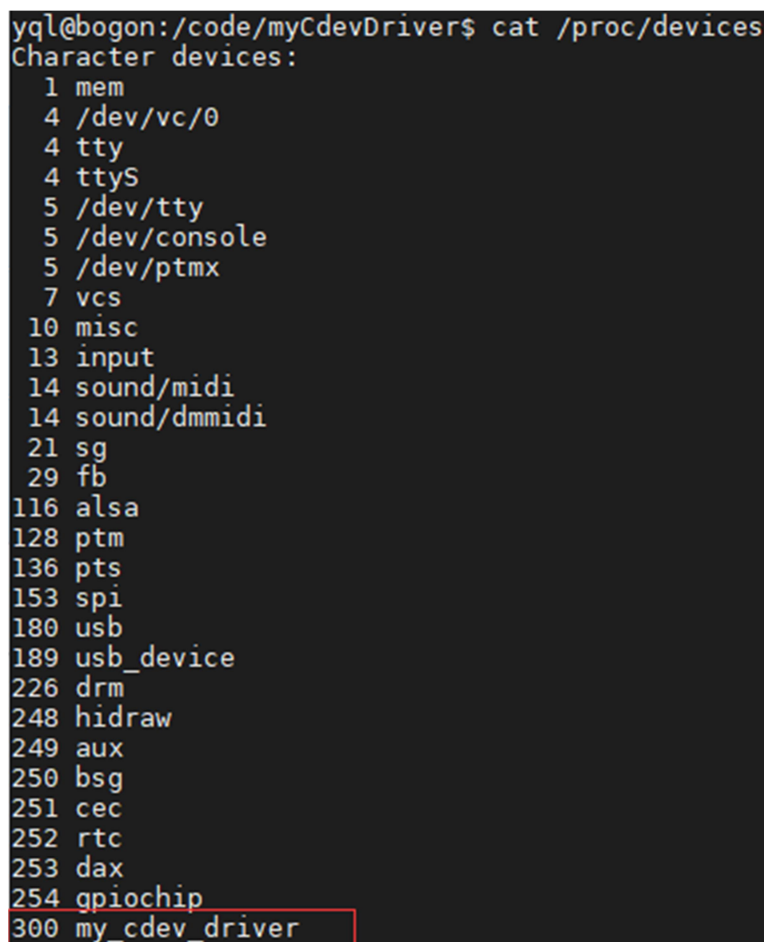
2. 编写 Makefile 文件

```
1. obj-m := mycdev_driver.o
2.
3. #kernel path
4. KERNELBUILD := /lib/modules/$(shell uname -r)/build
```

```
5.  
6. CURRENT_PATH := $(shell pwd)  
7.  
8. all:  
9.     make -C $(KERNELBUILD) M=$(PWD) modules  
10.  
11. clean:  
12.     make -C $(KERNELBUILD) M=$(PWD) clean
```

3. 查看设备号

编译以上程序，并插入模块（编译与插入模块详见实验 6）。此时查看 /proc/devices 文件，可看到自定义的设备和设备号。



```
yql@bogon:/code/myCdevDriver$ cat /proc/devices  
Character devices:  
1 mem  
4 /dev/vc/0  
4 tty  
4 ttyS  
5 /dev/tty  
5 /dev/console  
5 /dev/ptmx  
7 vcs  
10 misc  
13 input  
14 sound/midi  
14 sound/dmrmidi  
21 sg  
29 fb  
116 alsa  
128 ptm  
136 pts  
153 spi  
180 usb  
189 usb_device  
226 drm  
248 hidraw  
249 aux  
250 bsg  
251 cec  
252 rtc  
253 dax  
254 gpiochip  
300 my_cdev_driver
```

图 8.2 查看设备号

4. 创建文件设备节点

生成的设备需在 /dev/ 目录下生成对应的节点，此步骤由手动生成，注意文件需与设备号对应，如本演示中设备号为 300。

```
sudo mknod /dev/mycdev c 300 0
```

```
yql@bogon:/code/myCdevDrivers$ sudo mknod /dev/mycdev c 300 0
```

图 8.3 创建设备节点

5. 修改设备文件权限

本实验中需修改上一步中设备节点权限，否则测试程序无权限打开设备驱动文件。如下图所示为修改权限并查看。

```
sudo chmod 777 /dev/mycdev
```

```
yql@bogon:/code/myCdevDrivers$ sudo chmod 777 /dev/mycdev
yql@bogon:/code/myCdevDrivers$ ls -l /dev/ | grep 'mycdev'
crwxrwxrwx 1 root root 300, 0 Jun 7 15:32 mycdev
```

图 8.4 修改设备节点权限

6. 编写测试程序

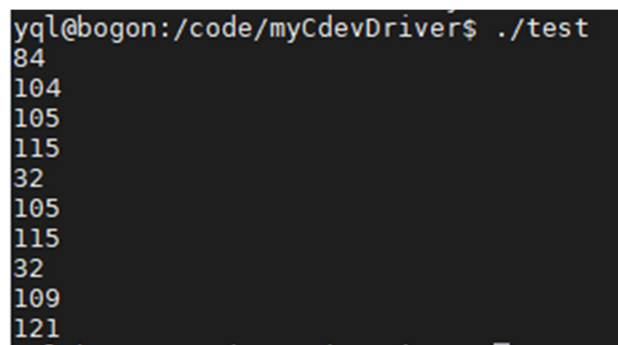
```
1. #include <stdio.h>
2. #include <sys/types.h>
3. #include <sys/stat.h>
4. #include <fcntl.h>
5. #include <stdlib.h>
6.
7. int main()
8. {
9.     int testdev;
10.    char buf[10];
11.
12.    testdev = open("/dev/mycdev", O_RDWR);
13.
14.    if(testdev == -1){
15.        printf("open file failed!\n");
16.        exit(1);
17.    }
18.
19.    //将 testdev 所指的文件读 10 个字节到 buf 中
20.    if(read(testdev, buf, 10) < 10){
21.        printf("Read error!\n");
22.        exit(1);
23.    }
24.
25.    for(int i = 0; i < 10; i++)
26.        printf("%d\n", buf[i]);
27.
28.    close(testdev);
```

```
29.  
30.     return 0;  
31. }
```

7. 编译以上程序，并执行

本示例中，测试程序保存为 `test.c`，并编译生成名为 `test` 的执行文件。

```
sudo gcc test.c -o test
```



```
yql@bogon:/code/myCdevDriver$ ./test  
84  
104  
105  
115  
32  
105  
115  
32  
109  
121
```

图 8.5 执行测试程序

同时通过 `dmesg` 查看日志信息（略）。

实验内容

- 1、理解并运行本字符驱动示例程序。