linux misc device 混杂设备

转载

没钱的笨小孩 于 2014-02-26 19:18:00 发布 550 收藏

misc device要点：

1. misc 混杂设备 是char 字符设备。【driver/char/misc.c 文件位置可见一斑】

2. misc device 是 主设备号为10 的字符设备。

3. misc device 使用 struct miscdevice 结构提进行描述。

4. misc device 使用 misc_register 函数进行注册， misc_deregister 进行卸载。

5. 注册为misc device，因为 misc device init , 以及register 的时候调用了 class_create(), device_create() 因此 /sys/class/misc 类会被创建， 设备节点也会自动创建。

转载：

http://www.linuxidc.com/Linux/2012-05/60856.htm

Linux驱动中把无法归类的五花八门的设备定义为混杂设备(用miscdevice结构体表述)。 miscdevice共享一个主设备号MISC_MAJOR(即10)，但次设备号不同。 所有的miscdevice设备形成了一个链表，对设备访问时内核根据次设备号查找对应的miscdevice设备，然后调用其file_operations结构中注册的文件操作接口进行操作。 在内核中用struct miscdevice表示miscdevice设备，然后调用其file_operations结构中注册的文件操作接口进行操作。miscdevice的API实现在drivers/char/misc.c中。

下边是描述这个设备的结构体：

```
1.  struct miscdevice  {
2.      int minor;                      //次设备号
3.      const char *name;                   //设备的名称
4.      const struct file_operations *fops;     //文件操作
5.      struct list_head list;              //misc_list的链表头
6.      struct device *parent;              //父设备(Linux设备模型中的东东了，哈哈)
7.      struct device *this_device;         //当前设备，是device_create的返回值，下边会看到
8.  };
```

然后来看看misc子系统的初始化函数：

```
1.  static int __init misc_init(void)
2.  {
3.      int err;
4.
5.  #ifdef CONFIG_PROC_FS
6.      /*创建一个proc入口项*/
7.      proc_create("misc", 0, NULL, &misc_proc_fops);
8.  #endif
9.      /*在/sys/class/目录下创建一个名为misc的类*/
10.     misc_class = class_create(THIS_MODULE, "misc");
11.     err = PTR_ERR(misc_class);
12.     if (IS_ERR(misc_class))
13.         goto fail_remove;
14.
15.     err = -EIO;
16.     /*注册设备，其中设备的主设备号为MISC_MAJOR，为10。设备名为misc，misc_fops是操作函数的集合*/
17.     if (register_chrdev(MISC_MAJOR,"misc",&misc_fops))
```

```
        goto fail_printk;
    return 0;

fail_printk:
    printk("unable to get major %d for misc devices/n", MISC_MAJOR);
    class_destroy(misc_class);
fail_remove:
    remove_proc_entry("misc", NULL);
    return err;
}
/*misc作为一个子系统被注册到linux内核中*/
subsys_initcall(misc_init);
```

下边是register_chrdev函数的实现：

```
int register_chrdev(unsigned int major, const char *name,
          const struct file_operations *fops)
{
    struct char_device_struct *cd;
    struct cdev *cdev;
    char *s;
    int err = -ENOMEM;
    /*主设备号是10，次设备号为从0开始，分配256个设备*/
    cd = __register_chrdev_region(major, 0, 256, name);
    if (IS_ERR(cd))
        return PTR_ERR(cd);
    /*分配字符设备*/
    cdev = cdev_alloc();
    if (!cdev)
        goto out2;

    cdev->owner = fops->owner;
    cdev->ops = fops;
    /*Linux设备模型中的,设置kobject的名字*/
    kobject_set_name(&cdev->kobj, "%s", name);
    for (s = strchr(kobject_name(&cdev->kobj),'/'); s; s = strchr(s, '/'))
        *s = '!';
    /*把这个字符设备注册到系统中*/
    err = cdev_add(cdev, MKDEV(cd->major, 0), 256);
    if (err)
        goto out;

    cd->cdev = cdev;

    return major ? 0 : cd->major;
out:
    kobject_put(&cdev->kobj);
out2:
    kfree(__unregister_chrdev_region(cd->major, 0, 256));
    return err;
}
```

来看看这个设备的操作函数的集合：

```
static const struct file_operations misc_fops = {
    .owner      = THIS_MODULE,
    .open       = misc_open,
};
```

可以看到这里只有一个打开函数，用户打开miscdevice设备是通过主设备号对应的打开函数，在这个函数中找到次设备号对应的相应的具体设备的open函数。它的实现如下：

```
static int misc_open(struct inode * inode, struct file * file)
{
    int minor = iminor(inode);
```

```
    struct miscdevice *c;
    int err = -ENODEV;
    const struct file_operations *old_fops, *new_fops = NULL;

    lock_kernel();
    mutex_lock(&misc_mtx);
    /*找到次设备号对应的操作函数集合，让new_fops指向这个具体设备的操作函数集合*/
    list_for_each_entry(c, &misc_list, list) {
        if (c->minor == minor) {
            new_fops = fops_get(c->fops);
            break;
        }
    }

    if (!new_fops) {
        mutex_unlock(&misc_mtx);
        /*如果没有找到，则请求加载这个次设备号对应的模块*/
        request_module("char-major-%d-%d", MISC_MAJOR, minor);
        mutex_lock(&misc_mtx);
        /*重新遍历misc_list链表，如果没有找到就退出，否则让new_fops指向这个具体设备的操作函数集合*/
        list_for_each_entry(c, &misc_list, list) {
            if (c->minor == minor) {
                new_fops = fops_get(c->fops);
                break;
            }
        }
        if (!new_fops)
            goto fail;
    }

    err = 0;
    /*保存旧打开函数的地址*/
    old_fops = file->f_op;
    /*让主设备号的操作函数集合指针指向具体设备的操作函数集合*/
    file->f_op = new_fops;
    if (file->f_op->open) {
        /*使用具体设备的打开函数打开设备*/
        err=file->f_op->open(inode,file);
        if (err) {
            fops_put(file->f_op);
            file->f_op = fops_get(old_fops);
        }
    }
    fops_put(old_fops);
fail:
    mutex_unlock(&misc_mtx);
    unlock_kernel();
    return err;
}
```

再来看看misc子系统对外提供的两个重要的API,misc_register,misc_deregister：

```
int misc_register(struct miscdevice * misc)
{
    struct miscdevice *c;
    dev_t dev;
    int err = 0;
    /*初始化misc_list链表*/
    INIT_LIST_HEAD(&misc->list);
    mutex_lock(&misc_mtx);
    /*遍历misc_list链表，看这个次设备号以前有没有被用过，如果次设备号已被占有则退出*/
    list_for_each_entry(c, &misc_list, list) {
        if (c->minor == misc->minor) {
            mutex_unlock(&misc_mtx);
            return -EBUSY;
        }
    }
    /*看是否是需要动态分配次设备号*/
    if (misc->minor == MISC_DYNAMIC_MINOR) {
        /*
```

```
        *#define DYNAMIC_MINORS 64 /* like dynamic majors */
        *static unsigned char misc_minors[DYNAMIC_MINORS / 8];
        *这里存在一个次设备号的位图，一共64位。下边是遍历每一位，
        *如果这位为0，表示没有被占有，可以使用，为1表示被占用。
        */
        int i = DYNAMIC_MINORS;
        while (--i >= 0)
            if ( (misc_minors[i>>3] & (1 << (i&7))) == 0)
                break;
        if (i<0) {
            mutex_unlock(&misc_mtx);
            return -EBUSY;
        }
        /*得到这个次设备号*/
        misc->minor = i;
    }
    /*设置位图中相应位为1*/
    if (misc->minor < DYNAMIC_MINORS)
        misc_minors[misc->minor >> 3] |= 1 << (misc->minor & 7);
    /*计算出设备号*/
    dev = MKDEV(MISC_MAJOR, misc->minor);
    /*在/dev下创建设备节点，这就是有些驱动程序没有显式调用device_create，却出现了设备节点的原因*/
    misc->this_device = device_create(misc_class, misc->parent, dev, NULL,
                      "%s", misc->name);
    if (IS_ERR(misc->this_device)) {
        err = PTR_ERR(misc->this_device);
        goto out;
    }

    /*
     * Add it to the front, so that later devices can "override"
     * earlier defaults
     */
    /*将这个miscdevice添加到misc_list链表中*/
    list_add(&misc->list, &misc_list);
 out:
    mutex_unlock(&misc_mtx);
    return err;
}
```

这个是miscdevice的卸载函数：

```
int misc_deregister(struct miscdevice *misc)
{
    int i = misc->minor;

    if (list_empty(&misc->list))
        return -EINVAL;

    mutex_lock(&misc_mtx);
    /*在misc_list链表中删除miscdevice设备*/
    list_del(&misc->list);
    /*删除设备节点*/
    device_destroy(misc_class, MKDEV(MISC_MAJOR, misc->minor));
    if (i < DYNAMIC_MINORS && i>0) {
        /*释放位图相应位*/
        misc_minors[i>>3] &= ~(1 << (misc->minor & 7));
    }
    mutex_unlock(&misc_mtx);
    return 0;
}
```

总结一下miscdevice驱动的注册和卸载流程：

misc_register:

匹配次设备号->找到一个没有占用的次设备号(如果需要动态分配的话)->计算设备号->创建设备文件

miscdevice结构体添加到misc_list链表中。

misc_deregister:

从mist_list中删除miscdevice->删除设备文件->位图位清零。