国嵌

www.enjoylinux.cn

LINUX
高级字符设备驱动程序

Contents

Contents

功能

Read 的功能?

Write的功能?

Ioctl 用来做什么?

功能

大部分驱动除了需要具备读写设备的能力外,还需要具备对硬件控制的能力。例如,要求设备报告错误信息,改变波特率,这些操作常常通过 ioctl 方法来实现。

用户使用方法

在用户空间，使用ioctl 系统调用来控制设备，原型如下：

```
int ioctl(int fd,unsigned long cmd,...)
```

原型中的点表示这是一个可选的参数，存在与否依赖于控制命令(第 2 个参数)是否涉及到与设备的数据交互。

驱动ioctl方法

ioctl 驱动方法有和用户空间版本不同的原型:

```
int (*ioctl)(struct inode *inode,struct file
*filp,unsigned int cmd,unsigned long arg)
```

cmd参数从用户空间传下来,可选的参数 arg 以一个 unsigned long 的形式传递,不管它是一个整数或一个指针。如果cmd命令不涉及数据传输，则第 3 个参数arg的值无任何意义。

loctl实现

如何实现loctl方法？

步骤：

1. 定义命令
2. 实现命令

定义命令

在编写ioctl代码之前，首先需要定义命令。为了防止对错误的设备使用正确的命令，命令号应该在系统范围内是唯一的。ioctl 命令编码被划分为几个位段，include/asm/ioctl.h中定义了这些位字段：类型（幻数），序号，传送方向，参数的大小。

Documentation/ioctl-number.txt文件中罗列了在内核中已经使用了的幻数。

定义命令

定义 ioctl 命令的正确方法是使用 4 个位段, 这个列表中介绍的符号定义在<linux/ioctl.h>中:

- **Type**

  幻数(类型): 表明哪个设备的命令,在参考了 ioctl-number.txt之后选出，8 位宽。

- **Number**

  序号，表明设备命令中的第几个,8 位宽。

定义命令

- **Direction**

  数据传送的方向，可能的值是 _IOC_NONE(没有数据传输), _IOC_READ, _IOC_WRITE。 数据传送是从应用程序的观点来看待的，_IOC_READ 意思是从设备读。

- **Size**

  用户数据的大小。（13/14位宽，视处理器而定）

定义命令

内核提供了下列宏来帮助定义命令:

- _IO(type，nr)

  没有参数的命令

- _IOR(type，nr，datatype)

  从驱动中读数据

- _IOW(type，nr，datatype)

  写数据到驱动

- _IOWR(type，nr，datatype)

  双向传送，type 和 number 成员作为参数被传递。

定义命令(范例)


```
#define MEM_IOC_MAGIC 'm'  //定义幻数

#define MEM_IOCSET
_IOW(MEM_IOC_MAGIC, 0, int)

#define MEM_IOCGQSET
_IOR(MEM_IOC_MAGIC, 1, int)
```

loctl函数实现

定义好了命令,下一步就是要实现loctl函数了，loctl函数的实现包括如下3个技术环节:

1. 返回值
2. 参数使用
3. 命令操作

loctl函数实现(返回值)

loctl函数的实现通常是根据命令执行的一个switch语句。但是，当命令号不能匹配任何一个设备所支持的命令时，通常返回-EINVAL（“非法参数”）。

loctl函数实现(参数)

如何使用loctl中的参数?

如果是一个整数，可以直接使用。如果是指针，我们必须确保这个用户地址是有效的,因此使用前需进行正确的检查。

loctl函数实现(参数检查)

不需要检测:

- copy_from_user
- copy_to_user
- get_user
- put_user

需要检测:

- __get_user
- __put_user

Ioctl函数实现(参数检查)

int access_ok(int type, const void *addr, unsigned long size)

第一个参数是 VERIFY_READ 或者 VERIFY_WRITE，用来表明是读用户内存还是写用户内存。addr 参数是要操作的用户内存地址，size 是操作的长度。如果 ioctl 需要从用户空间读一个整数，那么size参数等于 sizeof(int)。

access_ok 返回一个布尔值: 1 是成功(存取没问题)和 0 是失败(存取有问题),如果该函数返回失败, 则Ioctl应当返回–EFAULT 。

Ioctl函数实现(参数检查)


```
if (_IOC_DIR(cmd) & _IOC_READ)
    err = !access_ok(VERIFY_WRITE, (void __user *)arg,
    _IOC_SIZE(cmd));
//why _IOC_READ 对应 VERIFY_WRITE ???
else if (_IOC_DIR(cmd) & _IOC_WRITE)
    err = !access_ok(VERIFY_READ, (void __user *)arg,
    _IOC_SIZE(cmd));
if (err)
    return -EFAULT;
```


Ioctl函数实现(命令操作)


```
switch(cmd)
{
    case MEM_IOCSQUANTUM: /* Set: arg points to the value */
    retval = __get_user(scull_quantum, (int *)arg);
    break;

    case MEM_IOCGQUANTUM: /* Get: arg is pointer to result */
    retval = __put_user(scull_quantum, (int *)arg);
    break;

    default:
    return –EINVAL;
}
```


实例

Ioctl驱动实例分析

实验

www.enjoylinux.cn

设计带Ioctl命令的字符设备驱动

编写驱动

编写应用程序，与驱动交互

（在mini2440平台实现）

Contents

等待队列

在Linux驱动程序设计中，可以使用等待队列来实现进程的阻塞，等待队列可看作保存进程的容器，在阻塞进程时，将进程放入等待队列，当唤醒进程时，从等待等列中取出进程。

等待队列

Linux 2.6内核提供了如下关于等待队列的操作:

1、定义等待队列

```
wait_queue_head_t my_queue
```

2、初始化等待队列

```
init_waitqueue_head(&my_queue)
```

3、定义并初始化等待队列

```
DECLARE_WAIT_QUEUE_HEAD(my_queue)
```

等待队列

4、有条件睡眠

- wait_event(queue,condition)

  当condition(一个布尔表达式)为真时，立即返回；否则让进程进入TASK_UNINTERRUPTIBLE模式的睡眠，并挂在queue参数所指定的等待队列上。

- wait_event_interruptible(queue,condition)

  当condition(一个布尔表达式)为真时，立即返回；否则让进程进入TASK_INTERRUPTIBLE的睡眠，并挂在queue参数所指定的等待队列上。

等待队列

- int wait_event_killable(wait_queue_t queue, condition)

  当condition(一个布尔表达式)为真时，立即返回；否则让进程进入TASK_KILLABLE的睡眠，并挂在queue参数所指定的等待队列上。

等待队列

5、无条件睡眠（老版本，建议不再使用）

- **sleep_on(wait_queue_head_t *q)**

  让进程进入不可中断的睡眠,并把它放入等待队列q。

- **interruptible_sleep_on(wait_queue_head_t *q)**

  让进程进入可中断的睡眠,并把它放入等待队列q。

等待队列

6、从等待队列中唤醒进程

- **wake_up(wait_queue_t *q)**

  从等待队列q中唤醒状态为TASK_UNINTERRUPTIBLE，TASK_INTERRUPTIBLE，TASK_KILLABLE 的所有进程。

- **wake_up_interruptible(wait_queue_t *q)**

  从等待队列q中唤醒状态为TASK_INTERRUPTIBLE 的进程。

Contents

功能

前一节我们在设计简单字符驱动程序时，跳过了一个重要的问题：当一个设备无法立刻满足用户的读写请求时应当如何处理？ 例如：调用read时没有数据可读，但以后可能会有；或者一个进程试图向设备写入数据，但是设备暂时没有准备好接收数据。应用程序通常不关心这种问题，应用程序只是调用 read 或 write 并得到返回值。驱动程序应当(缺省地)阻塞进程，使它进入睡眠，直到请求可以得到满足。

阻塞方式

在阻塞型驱动程序中，Read实现方式如下：

如果进程调用read，但设备没有数据或数据不足，进程阻塞。当新数据到达后，唤醒被阻塞进程。

阻塞方式

在阻塞型驱动程序中，Write实现方式如下：如果进程调用了write，但设备没有足够的空间供其写入数据，进程阻塞。当设备中的数据被读走后，缓冲区中空出部分空间，则唤醒进程。

非阻塞方式

阻塞方式是文件读写操作的默认方式，但应用程序员可通过使用O_NONBLOCK标志来人为的设置读写操作为非阻塞方式（该标志定义在<linux/fcntl.h>中，在打开文件时指定）。

非阻塞方式

如果设置了O_NONBLOCK标志，read和write的行为是不同的。如果进程在没有数据就绪时调用了read，或者在缓冲区没有空间时调用了write，系统只是简单地返回-EAGAIN，而不会阻塞进程。

实例分析

阻塞型字符驱动

实验

www.enjoylinux.cn

设计阻塞型字符设备驱动

实现字符驱动Read、Write方法的阻塞操作

编写应用程序，与驱动交互

（在mini2440平台实现）

Contents

Poll方法

什么是Poll方法，功能是什么？

系统调用（用户空间）	驱动（内核空间）
Open	*Open*
Close	*Release*
Read	*Read*
Write	*Write*
Ioctl	*Ioctl*
lseek	*llseek*
Select	*Poll*

Select系统调用（功能）

Select系统调用用于多路监控，当没有一个文件满足要求时，select将阻塞调用进程。

```
int select(int maxfd, fd_set *readfds, fd_set *writefds, fe_set
*exceptfds, const struct  timeval *timeout)
```

Select系统调用（参数）

- **Maxfd:**

  文件描述符的范围，比待检测的最大文件描述符大1

- **Readfds:**

  被读监控的文件描述符集

- **Writefds:**

  被写监控的文件描述符集

- **Exceptfds:**

  被异常监控的文件描述符集;

- **Timeout:**

  定时器

Select系统调用（参数）

Timeout取不同的值，该调用有不同的表现:

- Timeout值为0，不管是否有文件满足要求，都立刻返回，无文件满足要求返回0，有文件满足要求返回一个正值。
- Timeout为NULL，select将阻塞进程，直到某个文件满足要求
- Timeout值为正整数，就是等待的最长时间，即select在timeout时间内阻塞进程。

Select系统调用（返回值）

Select调用返回时，返回值有如下情况:

1. 正常情况下返回满足要求的文件描述符个数;
2. 经过了timeout等待后仍无文件满足要求，返回值为0;
3. 如果select被某个信号中断，它将返回-1并设置errno为EINTR。
4. 如果出错，返回-1并设置相应的errno。

Select系统调用（使用方法）

www.enjoylinux.cn

1. 将要监控的文件添加到文件描述符集
2. 调用Select开始监控
3. 判断文件是否发生变化

Select系统调用（使用方法）

系统提供了4个宏对描述符集进行操作:

```
#include <sys/select.h>
void FD_SET(int fd, fd_set *fdset)
void FD_CLR(int fd, fd_set *fdset)
void FD_ZERO(fd_set *fdset)
void FD_ISSET(int fd, fd_set *fdset)
```

宏FD_SET将文件描述符fd添加到文件描述符集fdset中;

宏FD_CLR从文件描述符集fdset中清除文件描述符fd;

宏FD_ZERO清空文件描述符集fdset;

在调用select后使用FD_ISSET来检测文件描述符集fdset中的文件fd发生了变化。

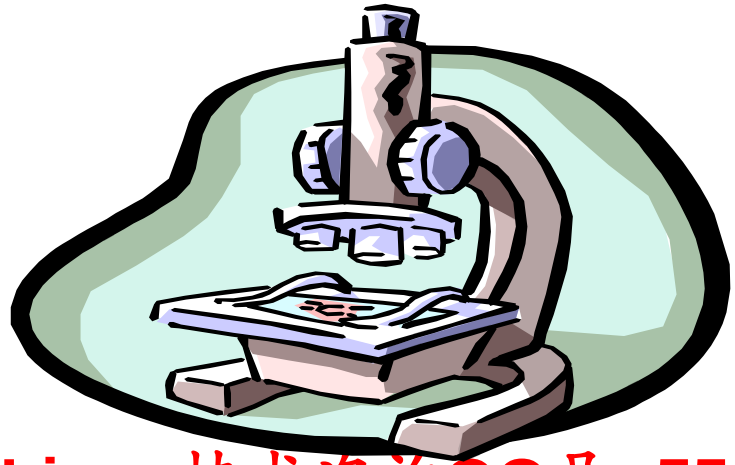
Select系统调用（使用方法）


```
FD_ZERO(&fds); //清空集合
FD_SET(fd1,&fds); //设置描述符
FD_SET(fd2,&fds); //设置描述符

maxfdp=fd1+1;    //描述符最大值加1,假设fd1>fd2

switch(select(maxfdp,&fds,NULL,NULL,&timeout))

case -1: exit(-1);break; //select错误，退出程序

case 0:break;

default:
        if(FD_ISSET(fd1,&fds)) //测试fd1是否可读
```

Poll方法

应用程序常常使用select系统调用,它可能会阻塞进程。这个调用由驱动的 poll 方法实现，原型为:

```
unsigned int (*poll)(struct file *filp,poll_table *wait)
```



Poll方法

Poll设备方法负责完成:

1. 使用poll_wait将等待队列添加到poll_table中。
2. 返回描述设备是否可读或可写的掩码。

位掩码

- **POLLIN**
  设备可读
- **POLLRDNORM**
  数据可读
- **POLLOUT**
  设备可写
- **POLLWRNORM**
  数据可写

设备可读通常返回(POLLIN|POLLRDNORM)

设备可写通常返回(POLLOUT|POLLWRNORM)

范例


```
static unsigned int mem_poll(struct file *filp,poll_table *wait)
{
    struct scull_pipe *dev =filp->private_data;
    unsigned int mask =0;

    /* 把进程添加到等待队列 */
    poll_wait(filp,&dev->inq,wait);

    /*返回掩码*/
    if (有数据可读)
        mask = POLLIN |POLLRDNORM;/*设备可读*/
    return mask;
}
```


工作原理

Poll方法只是做一个登记，真正的阻塞发生在select.c 中的 do_select函数。

内核代码分析

实例分析

Poll型设备驱动

实验

www.enjoylinux.cn

设计字符设备驱动的Poll函数

编写驱动

编写应用程序，与驱动交互

（在mini2440平台实现）

Contents

自动创建

创建设备文件的方法：

1. mknod手工创建
2. ？

自动创建（2.4内核）


```
devfs_register (devfs_handle_t dir,const char
    *name,unsigned int flags,unsigned int
    major,unsigned int minor,umode_t mode,void *ops,
    void *info)
```

在指定的目录中创建设备文件。dir:目录名，为空表示在/dev/目录下创建；name:文件名；flags:创建标志；major: 主设备号；minor:次设备号；mode:创建模式；ops:操作函数集;info:通常为空

自动创建（2.6内核）

从Linux 2.6.13开始，devfs不复存在，udev成为devfs的替代。相比devfs，udev（mdev）存在于应用层。利用udev(mdev)来实现设备文件的自动创建很简单，在驱动初始化的代码里调用class_create为该设备创建一个class，再为每个设备调用 device_create创建对应的设备。

自动创建（2.6内核）

例:

```
struct class *myclass = class_create(THIS_MODULE,
    “my_device_driver”);
device_create(myclass, NULL, MKDEV(major_num, 0), NULL,
    “my_device”);
```

当驱动被加载时，udev(mdev)就会自动在/dev下创建my_device设备文件。

实验

www.enjoylinux.cn

实现设备文件自动创建

编写字符设备驱动

在驱动初始化中自动创建设备文件

（在mini2440平台实现）