# Linux驱动\_global\_mem

在驱动层实现一个虚拟内存(FIFO)，应用层可以将数据写入到该内存中，也可以从该内存中读取。当没有数据时，读取会阻塞，当数据满了时，写数据会阻塞。

该驱动将会展示异步通知编程技巧。

## 1.驱动中的异步通知

当按键按下，内存中有数据等关键事件发生时，可以发送SIGIO信号，应用层需注册该信号的回调函数，当这些关键事件发生时，信号回调函数运行，做后续的处理，例如读取键值，读取内存中的数据等。

完成异步通知，**驱动层需要完成的动作如下**：

1.定义一个fasync\_struct 类型的变量。  
2.在fops中增加要使用的函数.fasync = xxx\_fasync。  
3.实现xxx\_fasync这个函数，调用fasync\_helper进行初始化。  
4.在需要发信号的地方使用kill\_fasync发送信号。  
5.在close中需要异步通知列表中删除该flip，fasync(-1,filp,0)。

**应用层的操作如下：**

1.open该驱动设备  
2.fcntl(fd, F\_SETOWN, getpid()，指定一个文件为属主，与驱动程序联系起来  
设置将要在文件描述符fd上接收SIGIO或SIGURG事件信号的进程或进程组标识。  
3.oflags = fcntl(fd, F\_GETFL)，取得open时文件标志符 。      
4.fcntl(fd, F\_SETFL, Oflags | FASYNC)，每当FASYNC标志状态改变时，linux设备驱动程序中的fasync()函数将得以执行。  
5.signal(SIGIO, my\_signal\_fun)，底层驱动发上来SIGIO的信号，然后调用my\_signal\_fun。

## 2. global\_mem驱动程序

代码如下：

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/device.h>

#include <linux/io.h>

#include <linux/cdev.h>

#include <linux/irq.h>

#include <linux/delay.h>

#include <linux/workqueue.h>

#include <linux/interrupt.h>

#include <linux/poll.h>

#include <linux/errno.h>

#include <asm/uaccess.h>

#include <asm/arch/map.h>

#include <asm/arch/regs-gpio.h>

#define DEVICE\_NAME "global\_mem"

#define GLOBAL\_MEM\_SIZE 4096

#define MEM\_CLEAR 1

struct \_global\_mem\_dev

**{**

struct cdev cdev**;**

unsigned int current\_len**;**

unsigned char mem**[**GLOBAL\_MEM\_SIZE**];**

struct mutex mutex**;**

wait\_queue\_head\_t r\_wait**;**

wait\_queue\_head\_t w\_wait**;**

struct fasync\_struct **\***async\_queue**;**

**}**global\_mem\_dev**;**

static struct class **\***global\_mem\_drv\_class**;**

static int global\_mem\_drv\_fasync**(**int fd**,** struct file**\*** file**,** int mode**)**

**{**

struct \_global\_mem\_dev **\***dev **=** file**->**private\_data**;**

**return** fasync\_helper**(**fd**,**file**,**mode**,&**dev**->**async\_queue**);**

**}**

static int global\_mem\_drv\_open**(**struct inode **\***inode**,** struct file **\***file**)**

**{**

struct \_global\_mem\_dev **\***devp**;**

devp **=** container\_of**(**inode**->**i\_cdev**,**struct \_global\_mem\_dev**,**cdev**);**

file**->**private\_data **=** devp**;**

**return** 0**;**

**}**

static int global\_mem\_drv\_close**(**struct inode **\***inode**,** struct file **\***file**)**

**{**

global\_mem\_drv\_fasync**(-**1**,** file**,** 0**);**

**return** 0**;**

**}**

static ssize\_t global\_mem\_drv\_write**(**struct file **\***file**,** const char \_\_user **\***buf**,** size\_t count**,** loff\_t **\*** ppos**)**

**{**

int ret**;**

struct \_global\_mem\_dev **\***dev **=** file**->**private\_data**;**

DECLARE\_WAITQUEUE**(**wait**,**current**);** //声明等待队列

mutex\_lock**(&**dev**->**mutex**);**

add\_wait\_queue**(&**dev**->**r\_wait**,&**wait**);** //加入读等待队列头

//如果队列已经满了

**while(**dev**->**current\_len **==** GLOBAL\_MEM\_SIZE**)**

**{**

**if(**file**->**f\_flags **&** O\_NONBLOCK**)**

**{**

ret **=** **-**EAGAIN**;**

**goto** out**;**

**}**

\_\_set\_current\_state**(**TASK\_INTERRUPTIBLE**);**

mutex\_unlock**(&**dev**->**mutex**);**

schedule**();**

**if(**signal\_pending**(**current**))**

**{**

ret **=** **-**ERESTARTSYS**;**

**goto** out2**;**

**}**

//如果mem满了，在睡眠状态被被读操作唤醒

mutex\_lock**(&**dev**->**mutex**);**

**}**

//如果写入的数据超过了mem的最大值，则只写入部分

**if(**count **>** GLOBAL\_MEM\_SIZE **-** dev**->**current\_len**)**

count **=** GLOBAL\_MEM\_SIZE **-** dev**->**current\_len**;**

**if(**copy\_from\_user**(**dev**->**mem**+**dev**->**current\_len**,** buf**,** count**))**

**{**

ret **=** **-**EFAULT**;**

**goto** out**;**

**}**

**else**

**{**

//如果写入成功，更新长度

dev**->**current\_len **+=** count**;**

printk**(**"write %d bytes,current\_len %d\n"**,**count**,**dev**->**current\_len**);**

//如果读操作因为mem为空阻塞，则会被该写操作唤醒

wake\_up\_interruptible**(&**dev**->**r\_wait**);**

**if(**dev**->**async\_queue**)**

**{**

kill\_fasync**(&**dev**->**async\_queue**,** SIGIO**,** POLLIN**);**

printk**(**KERN\_DEBUG "%s kill SIGIO"**,**\_\_func\_\_**);**

**}**

ret **=** count**;**

**}**

out**:**

mutex\_unlock**(&**dev**->**mutex**);**

out2**:**

remove\_wait\_queue**(&**dev**->**w\_wait**,&**wait**);**

set\_current\_state**(**TASK\_RUNNING**);**

**return** ret**;**

**}**

static ssize\_t global\_mem\_drv\_read**(**struct file **\***file**,** char **\***buf**,** size\_t len**,** loff\_t **\***off**)**

**{**

int ret**;**

struct \_global\_mem\_dev **\***dev **=** file**->**private\_data**;**

DECLARE\_WAITQUEUE**(**wait**,**current**);** //声明等待队列

mutex\_lock**(&**dev**->**mutex**);**

add\_wait\_queue**(&**dev**->**r\_wait**,&**wait**);** //加入读等待队列头

**while(**dev**->**current\_len **==** 0**)**

**{**

**if(**file**->**f\_flags **&** O\_NONBLOCK**)**

**{**

ret **=** **-**EAGAIN**;**

**goto** out**;**

**}**

//如果mem中没有数据，设置进程状态为休眠

\_\_set\_current\_state**(**TASK\_INTERRUPTIBLE**);**

mutex\_unlock**(&**dev**->**mutex**);**

schedule**();**//进程调度，休眠

//如果被信号唤醒，则退出

**if(**signal\_pending**(**current**))**

**{**

ret **=** **-**ERESTARTSYS**;**

**goto** out2**;**

**}**

//否则被写操作唤醒

//还要继续判断current是否为0，上锁

mutex\_lock**(&**dev**->**mutex**);**

**}**

**if(**len **>** dev**->**current\_len**)**

len **=** dev**->**current\_len**;**

**if(**copy\_to\_user**(**buf**,**dev**->**mem**,**len**))**

**{**

//如果拷贝给用户失败，退出

ret **=** **-**EFAULT**;**

**goto** out**;**

**}**

**else**

**{**

//如果拷贝给用户成功，重新移动mem中的数据

memcpy**(**dev**->**mem**,**dev**->**mem**+**len**,**dev**->**current\_len**-**len**);**

dev**->**current\_len **-=** len**;**

printk**(**"read %d bytes,current len:%d\n"**,**len**,**dev**->**current\_len**);**

//如果读出了一些数据，写操作处于阻塞时，可以唤醒写操作

wake\_up\_interruptible**(&**dev**->**w\_wait**);**

ret **=** len**;**

**}**

out**:**

mutex\_unlock**(&**dev**->**mutex**);**

out2**:**

remove\_wait\_queue**(&**dev**->**r\_wait**,&**wait**);**

set\_current\_state**(**TASK\_RUNNING**);**

**return** ret**;**

**}**

static int global\_mem\_drv\_ioctl**(**struct inode **\***inode**,** struct file **\***file**,** unsigned int cmd**,**unsigned long arg**)**

**{**

struct \_global\_mem\_dev **\***dev **=** file**->**private\_data**;**

**switch(**cmd**)**

**{**

**case** MEM\_CLEAR**:**

mutex\_lock**(&**dev**->**mutex**);**

memset**(**dev**->**mem**,**0**,**GLOBAL\_MEM\_SIZE**);**

mutex\_unlock**(&**dev**->**mutex**);**

printk**(**"global\_mem clear\n"**);**

**break;**

**}**

**return** 0**;**

**}**

static int global\_mem\_drv\_poll**(**struct file **\***file**,** struct poll\_table\_struct **\***wait**)**

**{**

unsigned int mask **=** 0**;**

struct \_global\_mem\_dev **\***dev **=** file**->**private\_data**;**

mutex\_lock**(&**dev**->**mutex**);**

poll\_wait**(**file**,** **&**dev**->**r\_wait**,** wait**);**

poll\_wait**(**file**,** **&**dev**->**w\_wait**,** wait**);**

**if(**dev**->**current\_len **!=** 0**)**

**{**

mask **|=** POLLIN **|** POLLRDNORM**;**

**}**

**if(**dev**->**current\_len **!=** GLOBAL\_MEM\_SIZE**)**

**{**

mask **|=** POLLOUT **|** POLLWRNORM**;**

**}**

mutex\_unlock**(&**dev**->**mutex**);**

**return** mask**;**

**}**

static struct file\_operations global\_mem\_drv\_fops **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**open **=** global\_mem\_drv\_open**,**

**.**write **=** global\_mem\_drv\_write**,**

**.**read **=** global\_mem\_drv\_read**,**

**.**release **=** global\_mem\_drv\_close**,**

**.**poll **=** global\_mem\_drv\_poll**,**

**.**ioctl **=** global\_mem\_drv\_ioctl**,**

**.**fasync **=** global\_mem\_drv\_fasync**,**

**};**

int global\_mem\_major**;**

static int global\_mem\_drv\_init**(**void**)**

**{**

dev\_t dev\_id**;**

//1.申请设备号

//注册字符设备编号，在proc/devices中创建global\_mem选项

**if(**global\_mem\_major**)**

**{**

//静态

dev\_id **=** MKDEV**(**global\_mem\_major**,**0**);**

register\_chrdev\_region**(**dev\_id**,**1**,**DEVICE\_NAME**);**

**}**

**else**

**{**

//动态

alloc\_chrdev\_region**(&**dev\_id**,**0**,**1**,**DEVICE\_NAME**);**

global\_mem\_major **=** MAJOR**(**dev\_id**);**

**}**

//2.注册字符设备

//初始化并且注册keys cdev

cdev\_init**(&**global\_mem\_dev**.**cdev**,** **&**global\_mem\_drv\_fops**);**

global\_mem\_dev**.**cdev**.**owner **=** THIS\_MODULE**;**

cdev\_add**(&**global\_mem\_dev**.**cdev**,**dev\_id**,**1**);**

//3.自动创建设备节点

//创建一个类，这个类存放于sysfs下面 /sys/class/keys

global\_mem\_drv\_class **=** class\_create**(**THIS\_MODULE**,** DEVICE\_NAME**);**

//mdev daemon就会自动创建节点/dev/keys

//加载模块的时候，用户空间中的mdev会自动响应device\_create(…)函数，去/sysfs下寻找对应的类从而创建设备节点。

class\_device\_create**(**global\_mem\_drv\_class**,** **NULL,** MKDEV**(**global\_mem\_major**,** 0**),** **NULL,** DEVICE\_NAME**);**

//初始化等待队列

init\_waitqueue\_head**(&**global\_mem\_dev**.**r\_wait**);**

init\_waitqueue\_head**(&**global\_mem\_dev**.**w\_wait**);**

//初始化互斥锁

mutex\_init**(&**global\_mem\_dev**.**mutex**);**

global\_mem\_dev**.**current\_len **=** 0**;**

**return** 0**;**

**}**

static void global\_mem\_drv\_exit**(**void**)**

**{**

//1.删除设备节点

//删除/dev/keys设备节点

class\_device\_destroy**(**global\_mem\_drv\_class**,**MKDEV**(**global\_mem\_major**,**0**));**

class\_destroy**(**global\_mem\_drv\_class**);**

//2.删除字符设备

//删除cdev

cdev\_del**(&**global\_mem\_dev**.**cdev**);**

//3.释放设备号

unregister\_chrdev\_region**(**MKDEV**(**global\_mem\_major**,**0**),**1**);**

**}**

module\_init**(**global\_mem\_drv\_init**);**

module\_exit**(**global\_mem\_drv\_exit**);**

MODULE\_LICENSE**(**"GPL"**);**

## 3. global\_mem应用程序

#include <stdio.h>

#include <signal.h>

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

static void signalio\_handler**(**int signum**)**

**{**

printf**(**"receive a signal from globalmem,signal num:%d\n"**,**signum**);**

**}**

int main**()**

**{**

int fd**,**oflags**;**

fd **=** open**(**"/dev/global\_mem"**,** O\_RDWR**,** S\_IRUSR **|** S\_IWUSR**);**

**if(**fd **!=** **-**1**)**

**{**

signal**(**SIGIO**,** signalio\_handler**);**

fcntl**(**fd**,** F\_SETOWN**,** getpid**());**

oflags **=** fcntl**(**fd**,** F\_GETFL**);**

fcntl**(**fd**,** F\_SETFL**,** oflags **|** FASYNC**);**

**while(**1**)**

**{**

sleep**(**100**);**

**}**

**}**

**else**

**{**

printf**(**"open fail\n"**);**

**}**

**}**

**简单测试：**

将应用程序运行在后台，当有数据写入时，对应的信号中断发生，cat可以读取内存中的数据。

# ./global\_mem\_drv\_test &

#

#

# echo 1 > /dev/global\_mem

write 2 bytes**,**current\_len 2

# receive a signal from globalmem,signal num:29

#

# cat /dev/global\_mem

1