前面讲到的阻塞与非阻塞IO，需要应用程序去查询设备状态，它们都属于同步IO。而异步IO是在设备可被访问时，由驱动主动通知应用程序。linux使用信号来实现异步IO机制。

# 应用程序中信号的接收

为了捕捉信号，首先需要将指定信号指定一个信号处理函数，方式有两种：

1. signal

这种方式比较简单，函数原型如下：

sighandler\_t signal(int signo, sighandler\_t handler)

signo：信号值

handler：信号处理函数

返回值：成功则返回信号的前一个处理函数，失败返回SGI\_ERR

其中sighandler\_t的原型如下：

typedef void (\*sighandler\_t)(int)

1. sigaction

sigaction比signal复杂，也比signal强大。其函数原型如下：

int sigaction(int signo, const struct sigaction \*act, struct sigaction \*oldact)

signo：信号值

act：如果非空，即为修改信号的配置

oldact：信号原来的配置

struct sigaction定义如下：

struct sigaction {

void (\*sa\_handler)(int); //信号处理函数，SIG\_IGN 为忽略，SIG\_DFL 为默认动作

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*); //信号处理函数，只能一次使用两个处理函数中的一个

sigset\_t sa\_mask;//阻塞关键字的信号集。

int sa\_flags;//信号处理选项

};

其中sa\_mask为一个信号集，在调用信号处理函数之前，将此信号集加入到进程的信号屏蔽集中，待信号处理函数返回后再将进程的信号屏蔽字恢复。这样，在调用信号处理函数时就能屏蔽指定信号。信号处理函数时，操作系统建立的新的信号屏蔽字包含正在被传递的信号，这样就可以保证正在处理指定信号时，此信号再次发生，它会被阻塞到这个信号处理结束，这样可以保证信号不会丢失。

其中sa\_flags如果设置为SA\_SIGINFO 则使用sa\_sigaction处理函数，否则使用sa\_handler处理函数。

指定信号的处理函数之后，还需要配置接收指定设备信号的进程:

fcntl(fd, F\_SETOWN, getpid())//设置为当前进程

然后还需要设置设备进入异步方式：

flag = fcntl(fd, F\_GETFL);//获取当前进程状态

fcntl(fd, F\_SETFL, flags | FASYNC);//设置为异步方式

# 驱动程序中信号的实现

驱动程序中对于信号的实现对应着应用程序中信号接收的三步：

1. 支持F\_SETOWN命令，设置file->f\_owner为对应进程ID，内核中已自动处理。
2. 支持F\_SETFL命令，当设置FASYNC方式时，会执行file\_operations中的fasync函数，所以设备驱动中需要实现此函数。
3. 设备可操作性时，发送信号。

fasync函数的实现使用fasync\_helper函数，其原型如下：

int fasync\_helper(int fd, struct file \* filp, int on, struct fasync\_struct \*\*fapp)

前三个参数就是fasync函数的参数。

fapp：注意是双指针，需要传递struct fasync\_struct结构体指针的指针。

文件关闭时，应调用设备驱动的fasync函数将文件从异步通知的列表中删除：

xxx\_fasync(-1, filp, 0);

设备可操作性时，驱动需要发送信号：

void kill\_fasync(struct fasync\_struct \*\*fp, int sig, int band)

fp：struct fasync\_struct结构体指针的指针

sig：发送的信号

band：设备可读时设为POLL\_IN，可写时设为POLL\_OUT

# siginfo\_t信息的传递

我们在测试示例中使用sigaction而不是signal就是因为sigaction的sa\_sigaction信号处理函数能传递siginfo\_t，siginfo\_t中包含了信号产生原因的信息，其定义如下：

typedef struct siginfo {

int si\_signo;//信号值

int si\_errno;//错误编码

int si\_code;//信号产生原因，不同信号，定义不同，可参考siginfo.h

union {

struct {

\_\_kernel\_pid\_t \_pid; /\* sender's pid \*/

\_\_ARCH\_SI\_UID\_T \_uid; /\* sender's uid \*/

} \_kill;

.

.

.

} \_sifields;//附加信息，不同信号有不同字段

} \_\_ARCH\_SI\_ATTRIBUTES siginfo\_t;

使用过程中发现在sa\_sigaction中打印info->si\_code一直为0x80，并不是我们在kill\_fasync发送的POLL\_IN，那么如果内核既传递POLL\_IN又传递POLL\_OUT我们就没法区分了。kill\_fasync分析如下：

void kill\_fasync(struct fasync\_struct \*\*fp, int sig, int band)

send\_sigio(fown, fa->fa\_fd, band);

send\_sigio\_to\_task(p, fown, fd, band, group);

send\_signal(sig, info, p, group);

switch ((unsigned long) info) {

case (unsigned long) SEND\_SIG\_PRIV:

q->info.si\_signo = sig;

q->info.si\_errno = 0;

q->info.si\_code = SI\_KERNEL;

q->info.si\_pid = 0;

q->info.si\_uid = 0;

break;

default:

copy\_siginfo(&q->info, info);

if (from\_ancestor\_ns)

q->info.si\_pid = 0;

break;

}

我们可以看到对于siginfo\_t(q->info)的赋值是在send\_signal中完成的，其会根据形参info值进行不同的赋值操作，如果等于SEND\_SIG\_PRIV，其si\_code为SI\_KERNEL，正好是我们一直看到的0x80，而info又是在send\_sigio\_to\_task中初始的，其分析如下：

void send\_sigio\_to\_task(struct task\_struct \*p, struct fown\_struct \*fown, int fd, int reason, int group)

int signum = ACCESS\_ONCE(fown->signum);

switch (signum) {

default:

si.si\_code = reason;

do\_send\_sig\_info(signum, &si, p, group)

case 0:

do\_send\_sig\_info(SIGIO, SEND\_SIG\_PRIV, p, group);

其会根据fown->signum值做出不同操作，如果等于0，则info默认为SEND\_SIG\_PRIV，如果非0，info才会被赋值，si.si\_code会被赋值为我们在kill\_fasync传递的POLL\_IN，也就是信号发生的原因。

那么fown->signum又是在哪赋值的呢，其在系统调用fcntl执行的do\_fcntl的case F\_SETSIG中进行了赋值，filp->f\_owner.signum = arg，所以现在就很明确了，我们如果想要在应用层获取不同的si\_code，那么就需要调用fcntl的F\_SETSIG命令，这个命令的含义就是设置IO可用时发送的信号。

# 示例说明

示例相对非阻塞IO示例只是在file\_operations中添加了fasync。

应用程序中因为使用了F\_SETSIG，这个命令不属于POSIX标准，是GNU扩展的，所以编译时需要添加-D\_GNU\_SOURCE。

因为使用了F\_SETSIG命令，我们可以在信号处理函数中获取正确的si\_code。

应用程序运行后，如果有按键按下：



## 设备树

fasync\_io: fasync\_io@0 {

compatible = "fasync-io";

gpios = <&gpio2 23 GPIO\_ACTIVE\_LOW>;

pinctrl-names = "default";

pinctrl-0 = <&interrupt\_gpio>;

};