阻塞与非阻塞指的是我们在执行设备操作时，如果不满足执行条件时两种不同的处理方式。阻塞会一直睡眠等待到条件满足，而非阻塞则不会等待，直接返回结果，此时就需要应用不停的去轮询。

# 阻塞IO

阻塞IO在不满足条件时进程会进入休眠将CPU让出，直到被唤醒。内核提供了等待队列来完成唤醒工作。

## 操作方法

等待队列的使用需要创建队列头，然后往队列头下添加队列项。队列头定义如下：

struct \_\_wait\_queue\_head {

spinlock\_t lock;

struct list\_head task\_list;

};

typedef struct \_\_wait\_queue\_head wait\_queue\_head\_t;

然后需要将队列头进行初始化：

void init\_waitqueue\_head(wait\_queue\_head\_t \*q)

当然也可以使用下面的宏直接完成队列头的定义与初始化：

DECLARE\_WAIT\_QUEUE\_HEAD(name)

DECLARE\_WAIT\_QUEUE\_HEAD\_ONSTACK(name)

上面这两个宏的区别是DECLARE\_WAIT\_QUEUE\_HEAD\_ONSTACK根据CONFIG\_LOCKDEP配置决定实现，可以检查spinlock使用的正确性。

之后可以创建队列项添加到队列：

void init\_waitqueue\_entry(wait\_queue\_t \*q, struct task\_struct \*p)

同样可以使用宏直接完成队列项的定义与初始化

DECLARE\_WAITQUEUE(name, tsk)

tsk或者p为队列项指代的进程，一般设置为current，current表示当前进程。

添加队列项到等待队列：

void add\_wait\_queue(wait\_queue\_head\_t \*q, wait\_queue\_t \*wait)

将进程休眠：

\_\_set\_current\_state(state\_value);

state\_value 可以为：

TASK\_UNINTERRUPTIBLE：休眠不可被信号打断；

TASK\_INTERRUPTIBLE：休眠可被信号打断。

调度其他进程执行：

schedule();

此时进程就会等待被唤醒，如果进程被设置为TASK\_INTERRUPTIBLE状态，还需判断是否是被信号唤醒：

signal\_pending(current); //返回真代表是信号唤醒

进程被唤醒后，将进程设置为运行状态：

set\_current\_state(TASK\_RUNNING) ;

然后将队列项移除：

void remove\_wait\_queue(wait\_queue\_head\_t \*q, wait\_queue\_t \*wait);

唤醒队列中的所有进程：

void wake\_up(wait\_queue\_head\_t \*q)

void wake\_up\_interruptible(wait\_queue\_head\_t \*q)

两者的区别是wake\_up可以唤醒处于TASK\_UNINTERRUPTIBLE和TASK\_INTERRUPTIBLE状态的进程，wake\_up\_interruptible只能唤醒处于TASK\_INTERRUPTIBLE状态的进程。

除了添加队列项等待被唤醒外，还可以添加唤醒条件，被唤醒后检查条件是否满足，满足则向下执行，不满足则继续睡眠：

wait\_event(wq, condition)

wait\_event\_timeout(wq, condition, timeout)

wait\_event\_interruptible(wq, condition)

wait\_event\_interruptible\_timeout(wq,condition, timeout)

有无interruptible的区别就是是否将进程设置为TASK\_INTERRUPTIBLE状态，返回-ERESTARTSYS时就是被信号打断，timeout则是添加了超时机制，返回值大于等于1则表示在超时时间耗尽前条件满足。分析wait\_event的实现其实就是上述队列项的使用，同样也是初始化队列项，添加到队列头，然后将进程睡眠后进行调度。

## 示例说明

示例为中断示例基础上更改，目的是应用层读取按键状态，如果按键未被按下则阻塞。相比中断中的示例，添加了按键的去抖，防止误触。在read中实现了上述的两种唤醒方式，一种是添加队列项，另一种就是等待事件，最终结果是一致的。同时两种实现方式都对是否是被信号打断做了判断，可打印相应信息。

应用程序中建了两个线程，线程2往线程1发送SIGCHLD信号，注意主线程中要关联SIGCHLD信号的处理函数，否则此信号默认被忽略。

运行应用程序后，如果按键被按下：



如果收到信号：



### 设备树

block\_io: block\_io@0 {

compatible = "block-io";

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

pinctrl-names = "default";

pinctrl-0 = <&interrupt\_gpio>;

};

# 非阻塞IO

阻塞IO是直接去读写，如果不满足条件就会阻塞。而非阻塞IO先查询是否可以读写，如果不可读写，则不进行操作，然后继续查询。所以非阻塞IO的处理方式就是轮询。

## 应用程序的轮询方法

### select

目前使用最广泛的当属select，其函数原型如下：

int select(int maxfd, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

maxfd：集合中所有文件描述符的最大值加1

readfds、writefds、errorfds分别是select监视的读、写、异常处理的文件描述符集合。readfds集合中的任一文件可读，select就会返回，writefds集合中的任一文件可写，select也会返回。如下操作可对fd\_set进行配置：

FD\_ZERO(fd\_set \*fdset) //清除集合

FD\_SET(int fd, fd\_set \*fdset)//将某个文件描述符加入集合

FD\_CLR(int fd, fd\_set \*fdset) //将某个文件描述符从集合中清除

FD\_ISSET(int fd, fdset \*fdset) //判断文件描述符是否置位

timeout：超时时间指针，如果为NULL，则为阻塞函数，直到描述符集合中有描述符发生变化才会返回；如果将结构体的秒和微妙值全部设置为0，则为非阻塞函数，会立刻返回，如果文件描述符无变化返回0，有变化返回正值；如果时间值大于0，则在超时时间内没有文件描述符的变化，则返回0，有变化则返回正值。struct timeval定义如下：

struct timeval{

long tv\_sec; /\*秒 \*/

long tv\_usec; /\*微秒 \*/

};

### poll

poll相对select只是接口有所不同，函数原型如下：

int poll (struct pollfd \*fds, unsigned int nfds, int timeout);

fds：poll不再为读/写/异常这些条件构建描述集，而是将文件句柄与事件进行组合，struct pollfd定义如下：

struct pollfd {

int fd; //文件句柄

short events; //监听的事件类型

short revents; //内核返回的事件类型

};

事件类型如下，可以通过或监听多个事件：

POLLIN //有数据可读

POLLOUT //数据可写

POLLERR //发生错误

nfds：fds的个数

timeout：超时时间，单位为ms

### epoll

epoll不会像select或者poll一样随着fd数目的增多而降低性能。使用方法如下：

int epoll\_create(int size)

创建epoll句柄，size是监听fd的个数，此创建的句柄，同样需要close关闭。

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)

epfd：epoll句柄

op：操作目的

EPOLL\_CTL\_ADD：注册新的fd到epfd。

EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件。

EPOLL\_CTL\_DEL：从epfd中删除fd。

fd：需要监听的文件句柄

\*event：监听的事件类型，struct epoll\_event定义如下：

struct epoll\_event {

\_\_u32 events;//事件类型

\_\_u64 data;

};

events可以是如下宏的或：

EPOLLIN:可读

EPOLLOUT:可写

EPOLLERR:发生错误

EPOLLHUP:挂起

int epoll\_wait(int epfd, struct epoll\_event \*event, int maxevents, int timeout)

epfd：epoll句柄

\*event：输出参数，是从内核获取的事件集合

maxevents：最多接收的事件数目

timeout:超时时间，单位为ms，如果等于0则立即返回，等于-1则表示阻塞等待。

## 驱动中的轮询实现

应用程序中无论调用select、poll还是epoll，最终都会执行到file\_operations中的poll函数。其原型如下：

unsigned int (\*poll) (struct file \*filp, struct poll\_table\_struct \*wait)

wait：轮询表指针，我们需要使用poll\_wait将等待队列添加到轮询表中。

void poll\_wait(struct file \* filp, wait\_queue\_head\_t \* wait\_address, poll\_table \*p)

poll函数需要返回设备资源的状态：

POLLIN //有数据可读

POLLOUT //数据可写

POLLERR //发生错误

## 示例说明

示例中相对阻塞的示例只是添加了poll，可以看到改动很小。应用程序中实现了select/poll/epoll三种轮询方式，可以通过配置宏定义自由选择。同样也是通过发送信号的方式模拟信号的影响。

运行应用程序后，如果按键按下：



如果收到信号：



如果超过目前设置的1.5s超时时间未按键：

