# 文件io读写的基本四种机制

注：主要参考 百问网 、[linux基础——linux下五种IO模型小结（阻塞IO、非阻塞IO、IO复用、信号驱动式IO、异步IO）_yexz的博客-CSDN博客_linux阻塞型io](#) （这篇文章会讲的很明白）等。

文件io的读写的基本四种机制：非阻塞（查询），阻塞（休眠死等-唤醒返回），poll/select（设置阻塞事件和时间），异步通知。

**注意**：所有 应用程序 与 驱动程序 相互收发的机制，包括 read/write/ioctl，还有 poll、异步通知、mmap 等，都要在 设备文件 被打开 的状态 下去执行，否则啥也得不到。

## 非阻塞（查询）

- open 时 传入 O_NONBLOCK 标志，例子：`int fd = open(argv[1], O_RDWR | O_NONBLOCK);`，然后正常的 使用 read/write 进行读写。
- 在 open 之后，还可以通过 fcntl() 继续修改 flag 标志，修改为 阻塞 或 非阻塞，本小节尾的例子程序。。
- APP 调用 read 函数读取数据时，如果驱动程序中有数据，那么 APP 的 read 函数会返回数据，否则也会立刻返回错误。
  - 对于普通文件、块设备文件，O_NONBLOCK 不起作用。
  - 对于字符设备文件，O_NONBLOCK 起作用的前提是 驱动程序内 针对 O_NONBLOCK 做了处理，即驱动程序支持 阻塞/非阻塞。

```
int set_nonblock(int fd)
{
        int old_option = fcntl(fd, F_GETFL);
        int new_option = old_option | O_NONBLOCK;
        fcntl(fd, F_SETFL, new_option);

        return old_option;
}

或者这么写

int flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);   /* 非阻塞方式 */
fcntl(fd, F_SETFL, flags & ~O_NONBLOCK);  /* 阻塞方式 */
```

## 阻塞（休眠等待-唤醒返回）

- APP 调用 open 函数时，不要传入 O_NONBLOCK 标志。或者像上一节那样 通过 `fcntl(fd, F_SETFL, flags & ~O_NONBLOCK);` 主动设置为阻塞。
- APP 调用 read 函数读取数据时，如果驱动程序中有数据，那么 APP 的 read 函数会返回数据。
- 否则 APP 就会在内核态休眠，当有数据时驱动程序会把 APP 唤醒，read 函数恢复执行并返回数据给 APP。

## poll / select（设置事件的等待时间）

- 先 open 时 传入 O_NONBLOCK 标志；再 调用 poll/select（二者 机制是完全一样的，只是 APP 接口函数不一样）
    - 关于使用 poll 为什么一定要传入 O_NONBLOCK 标志 为什么poll/select在open时要使用非阻塞NONBLOCK_stone_322的博客-CSDN博客_open 非阻塞。
    - 可以同时监测多个文件，每个文件可以设置不同的 阻塞/等待 条件 events（可以多个事件 用或 `|`），但 超时时间 都一并设置一样的。
- 当 poll/select 返回时候，根据 返回事件 revents 判断，再进行 读写，否则返回是 超时 或者 错误。
    - 如果驱动程序中有数据，则立刻返回；否则就休眠。
    - 在休眠期间，如果有人操作了硬件，驱动程序获得数据后就会把 APP 唤醒，导致 poll 或 select 立刻返回；
    - 如果在"超时时间"内无人操作硬件，则时间到后 poll 或 select 函数也会返回。
- APP可以根据函数的返回值判断返回原因：有数据？无数据超时返回？
- poll/select 的头文件分别是 `#include <poll.h>` 和 `#include <sys/select.h>`。
- 例子：（以 poll 为例）

```
int fd = open(argv[1], O_RDWR | O_NONBLOCK);
struct pollfd fds[1]; /* fds 结构体对每一个文件填入文件描述符、poll 事件 */
nfds_t nfds = 1; /* 要监测几个文件，这里是一个 所以 下面 只填入 fds 结构体数组中 第一个数组元素 fds[0] */

/* 填入文件描述符、poll 事件（要等待的事件） 和 实际发生的事件，这个结构体就这三个元素 */
fds[0].fd = fd; fds[0].events  = POLLIN; fds[0].revents = 0;

int ret = poll(fds, nfds, 5000); /* 5000 为超时时间，单位 毫秒，超时 写 0 表示不等待直接返回，写 -1 表示死等 直到条件发生 */

if (ret > 0) /* 返回值大于 0 表示 有数据可读 */
{
    if (fds[0].revents == POLLIN)
    {
        while (read(fd, &event, sizeof(event)) == sizeof(event))
        {
            printf("get event: type = 0x%x, code = 0x%x, value = 0x%x\n", event.type, event.code, event.value);
        }
    }
}else if (0 == ret) /* 返回值为 0 表示 超时 */
{
    printf("time out\n");
}else /* 返回值为 负值 表示 错误 */
{
    printf("poll err\n");
}
```

```
events 的选项：
    POLLIN      有数据可读--用于读
    POLLRDNORM  等同于POLLIN
    POLLRDBAND  Priority band data can be read，有优先级较较高的“band data”可读，
Linux系统中很少使用这个事件
    POLLPRI     高优先级数据可读
    POLLOUT     可以写数据--用于写
    POLLWRNORM  等同于POLLOUT
    POLLWRBAND  Priority data may be written
    POLLERR     发生了错误
    POLLHUP     挂起
    POLLNVAL    无效的请求，一般是fd未open
```

关于 poll 更详细的说明 / 更多好的参考：

- poll函数详解青季的博客-*CSDN*博客poll函数。
- struct pollfd wocjj的博客-CSDN博客_pollfd。
- linux基础——linux下多路IO复用接口之select/poll_yexz的博客-CSDN博客。

# 异步通知（捕获信号 SIGIO）

主要是使用信号机制。

- app 向 一个具体的驱动程序 注册 异步IO事件（SIGIO）的 信号（signal） 和 处理/回调/钩子/handler 函数，并告知自己的进程号，并使能 异步通知。
- 一个具体的驱动程序 在 特定 情况下 通过 内核提供的 API 向 APP 发 异步IO事件（SIGIO）信号。
- 则此时 APP 自动去 调用 并 执行 处理/回调/钩子/handler 函数。

信号类型有很多，常用的就是 `SIGIO` 异步IO事件。

信号注册 API：`void (* signal( int signo, void (*func)(int) ) )(int);`。

- 函数名　　：`signal`。
- 函数参数　：`int signo, void (*func)(int)`，两个，一个信号类型，一个函数指针。
- 返回值类型　：`void (*)(int);`，函数指针。
- signal 函数详解：信号之signal函数 - ITtecman - 博客园 (cnblogs.com)。

头文件：`#include <signal.h>`。

例子：

```c
/* 回调函数，函数声明形式固定，可以根据 sig 来区分接收到了哪一种信号 */
void my_sig_handler(int sig)
{
    struct input_event event;
    while (read(fd, &event, sizeof(event)) == sizeof(event))
    {
        printf("get event: type = 0x%x, code = 0x%x, value = 0x%x\n",
event.type, event.code, event.value);
    }
}

/* 注册信号处理函数，告知信号类型为 SIGIO，回调函数为 my_sig_handler */
signal(SIGIO, my_sig_handler);

/* 打开驱动程序，传入 O_NONBLOCK 标志 */
fd = open(argv[1], O_RDWR | O_NONBLOCK);
```

```
/* 把 APP 的进程号 getpid() 告诉驱动程序 fd */
fcntl(fd, F_SETOWN, getpid());

/* 使能 "异步通知" */
flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | FASYNC);
/* F_GETFL 取得文件描述词状态旗标，此旗标为open()的参数 flags。
   F_SETFL 设置文件描述词状态旗标，让打开时候的参数 flags 可重新设，但只允许O_APPEND、
O_NONBLOCK和O_ASYNC位的改变，其他位的改变将不受影响。 */

    while(1)
    {
        printf("main loop count = %d\n", count++);
        sleep(2);
    }

    这个 while(1) 死循环 这里 或者 可以换成：

    for(;;)
        pause(); /* pause函数，它使调用进程在接到一个信号前挂起 */
```

下面的情况可以产生Signal：引自 signal函数使用随意的风的博客-*CSDN*博客signal 函数。

1. 按下 CTRL+C 产生 SIGINT。
2. 硬件中断，如除0，非法内存访问（SIGSEV）等等。
3. Kill() 函数可以对进程发送 Signal。shell 中的 "Kill -s <信号> <进程号>" 命令。实际上是对 Kill() 函数的一个包装。
4. 软件中断。如当Alarm Clock超时（SIGURG），当Reader中止之后又向管道写数据（SIGPIPE），等等。