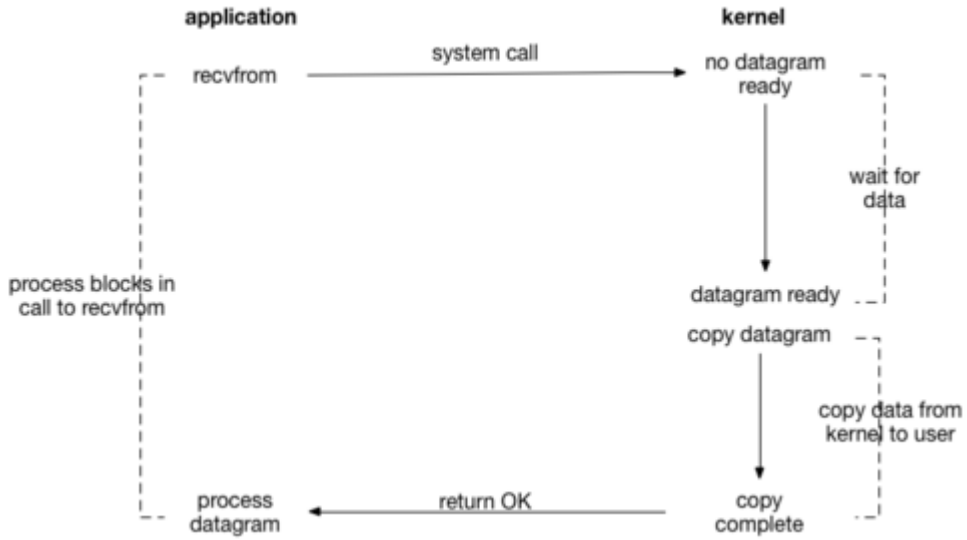
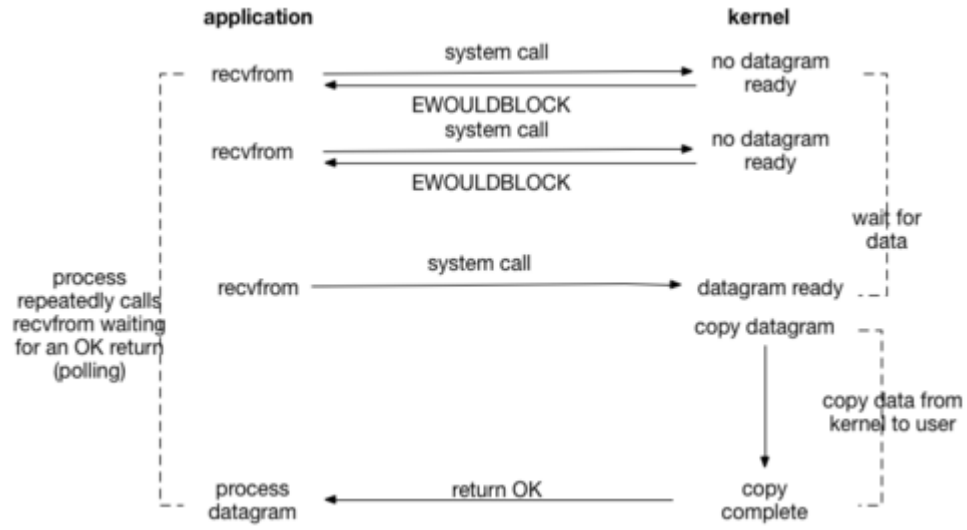
一、五种IO模型------读写外设数据的方式

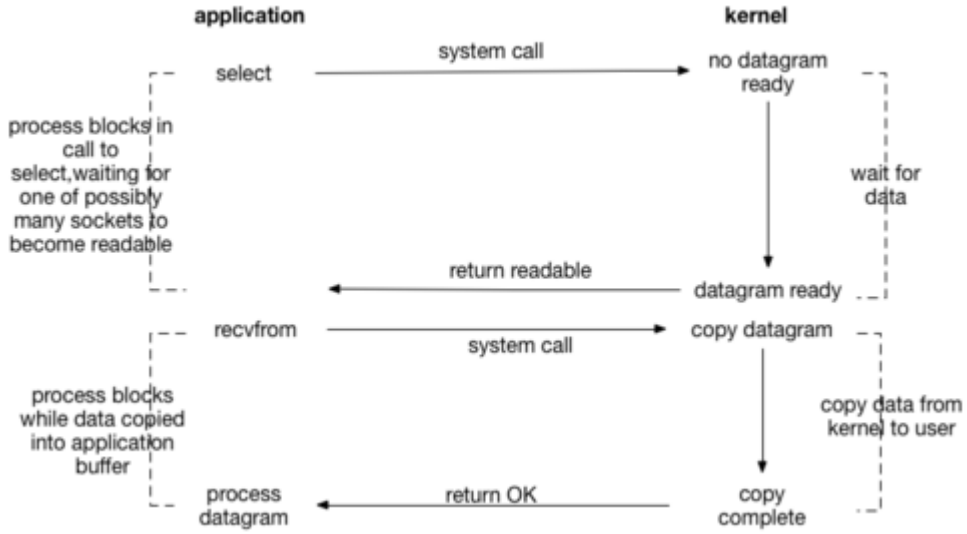
1. 阻塞: 不能操作就睡觉



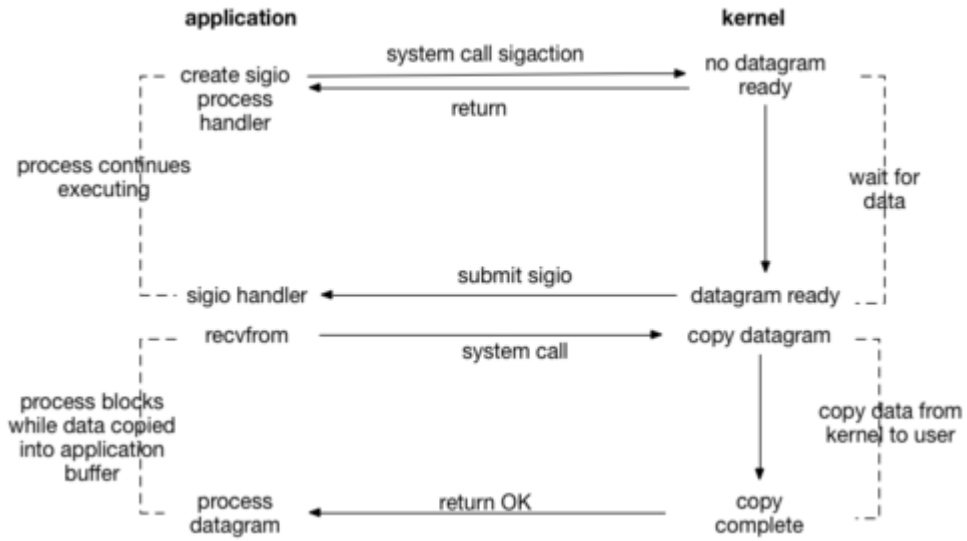
2. 非阻塞：不能操作就返回错误



3. 多路复用：委托中介监控

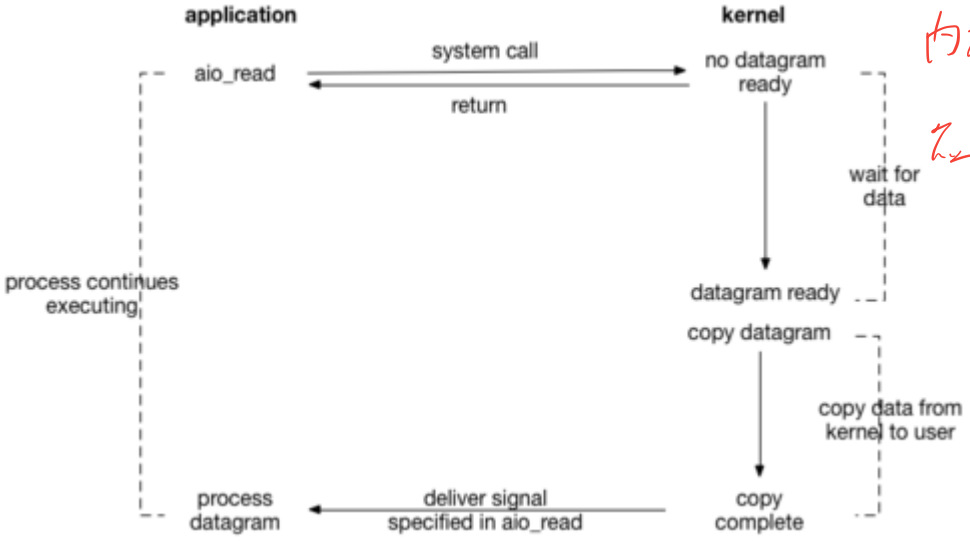


4. 信号驱动：让内核如果能操作时发信号，在信号处理函数中操作



5. 异步IO：向内核注册操作请求，内核完成操作后发通知信号

Q what? 哦，就让内核操作完了通知应用



二、阻塞与非阻塞

应用层：

open时由O_NONBLOCK指示read、write时是否阻塞

open以后可以由fcntl函数来改变是否阻塞：

```
flags = fcntl(fd,F_GETFL,0);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

驱动层：通过等待队列

```
wait_queue_head_t //等待队列头数据类型

init_waitqueue_head(wait_queue_head_t *pwq) //初始化等待队列头

wait_event_interruptible(wq,condition)
/*
功能：条件不成立则让任务进入浅度睡眠，直到条件成立醒来
    wq:等待队列头
    condition：C语言表达式
返回：正常唤醒返回0，信号唤醒返回非0（此时读写操作函数应返回-ERESTARTSYS）
*/

wait_event(wq,condition) //深度睡眠

wake_up_interruptible(wait_queue_head_t *pwq)

wake_up(wait_queue_head_t *pwq)


/*
1. 读、写用不同的等待队列头rq、wq
2. 无数据可读、可写时调用wait_event_interruptible(rq、wq,条件)
3. 写入数据成功时唤醒rq，读出数据成功唤醒wq
*/
```

三、多路复用

描述符：

1. 文件描述符：设备文件、管道文件
2. socket描述符

3.1 应用层：三套接口select、poll、epoll

select：位运算实现 监控的描述符数量有限（32位机1024,64位机2048） 效率差

poll：链表实现，监控的描述符数量不限 效率差

epoll：效率最高，监控的描述符数量不限

select

```
int select(int nfds, fd_set *readfds, fd_set *writefds,fd_set *exceptfds, struct
timeval *timeout);
/*  功能：监听多个描述符，阻塞等待有一个或者多个文件描述符，准备就绪。
        内核将没有准备就绪的文件描述符，从集合中清掉了。
    参数：   nfds          最大文件描述符数 ，加1
             readfds       读文件描述符集合
             writefds          写文件描述符集合
             exceptfds         其他异常的文件描述符集合
             timeout       超时时间（NULL）
    返回值：当timeout为NULL时返回0，成功:准备好的文件描述的个数   出错:-1
           当timeout不为NULL时，如超时设置为0，则select为非阻塞，超时设置 > 0，则无描述
符可被操作的情况下阻塞指定长度的时间
*/
void FD_CLR(int fd, fd_set *set);
//功能：将fd 从集合中清除掉

int  FD_ISSET(int fd, fd_set *set);
//功能：判断fd 是否存在于集合中

void FD_SET(int fd, fd_set *set);
//功能：将fd 添加到集合中

void FD_ZERO(fd_set *set);
//功能：将集合清零

//使用模型：

while(1)
{
    /*得到最大的描述符maxfd*/
    /*FD_ZERO清空描述符集合*/
    /*将被监控描述符加到相应集合rfds里  FD_SET*/
    /*设置超时*/
    ret = select(maxfd+1,&rfds,&wfds,NULL,NULL);
    if(ret < 0)
    {
        if(errno == EINTR)//错误时信号引起的
        {
            continue;
        }
        else
        {
            break;
        }
    }
```

```
    else if(ret == 0)
    {//超时
        //.....
    }
    else
    { //> 0 ret为可被操作的描述符个数
        if(FD_ISSET(fd1,&rfds))
        {//读数据
            //....
        }
        if(FD_ISSET(fd2,&rfds))
        {//读数据
            //....
        }
        ///.....
        if(FD_ISSET(fd1,&wfds))
        {//写数据
            //....
        }
    }
}
```

3.2 驱动层：实现poll函数

```
void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table
*p);
/*功能：将等待队列头添加至poll_table表中
  参数：struct file :设备文件
  Wait_queue_head_t :等待队列头
  Poll_table :poll_table表
*/

/*该函数与select、poll、epoll_wait函数相对应，协助这些多路监控函数判断本设备是否有数据可
读写*/
unsigned int xxx_poll(struct file *filp, poll_table *wait) //函数名初始化给struct
file_operations的成员.poll
{
    unsigned int mask = 0;
    /*
        1. 将所有等待队列头加入poll_table表中
        2. 判断是否可读，如可读则mask |= POLLIN | POLLRDNORM;
        3. 判断是否可写，如可写则mask |= POLLOUT | POLLWRNORM;
    */

    return mask;
}
```

四、信号驱动

4.1 应用层：信号注册+fcntl

```
signal(SIGIO, input_handler); //注册信号处理函数

fcntl(fd, F_SETOWN, getpid());//将描述符设置给对应进程，好由描述符获知PID

oflags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, oflags | FASYNC);//将该设备的IO模式设置成信号驱动模式

void input_handler(int signum)//应用自己实现的信号处理函数，在此函数中完成读写
{
    //读数据
}

//应用模板
int main()
{
    int fd = open("/dev/xxxx",O_RDONLY);

    fcntl(fd, F_SETOWN, getpid());

    oflags = fcntl(fd, F_GETFL);
    fcntl(fd, F_SETFL, oflags | FASYNC);

    signal(SIGIO,xxxx_handler);

    //......
}

void xxxx_handle(int signo)
{//读写数据

}
```

4.2 驱动层：实现fasync函数

```
/*设备结构中添加如下成员*/
struct fasync_struct *pasync_obj;

/*应用调用fcntl设置FASYNC时调用该函数产生异步通知结构对象，并将其地址设置到设备结构成员中*/
static int hello_fasync(int fd, struct file *filp, int mode) //函数名初始化给struct file_operations的成员.fasync
{
    struct hello_device *dev = filp->private_data;
    return fasync_helper(fd, filp, mode, &dev->pasync_obj);
}
```

```
/*写函数中有数据可读时向应用层发信号*/
if (dev->pasync_obj)
        kill_fasync(&dev->pasync_obj, SIGIO, POLL_IN);

/*release函数中释放异步通知结构对象*/
if (dev->pasync_obj)
    fasync_helper(-1, filp, 0, &dev->pasync_obj);

int fasync_helper(int fd, struct file *filp, int mode, struct fasync_struct **pp);
/*
    功能：产生或释放异步通知结构对象
    参数：
    返回值：成功为>=0,失败负数
*/

void kill_fasync(struct fasync_struct **, int, int);
/*
    功能：发信号
    参数：
        struct fasync_struct ** 指向保存异步通知结构地址的指针
        int      信号 SIGIO/SIGKILL/SIGCHLD/SIGCONT/SIGSTOP
        int      读写信息POLLIN、POLLOUT
*/
```