第4节 文件读写（PV操作）

参考链接：

IO控制方式

生磁盘的使用

linux0.11内核完全剖析 - 块设备驱动程序-CSDN博客

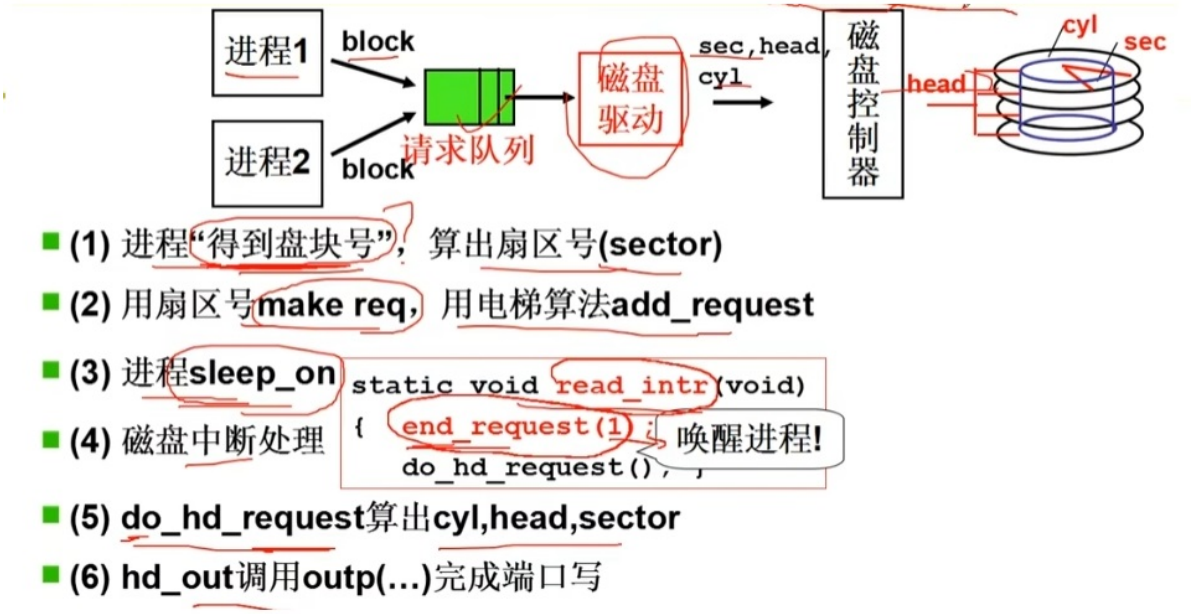
第九章 块设备驱动程序 - 掘金)

第47回 | 读取硬盘数据的细节 - 文章详情 (itpub.net)

操作系统学习笔记11 | 生磁盘的使用与管理 - climerecho - 博客园 (cnblogs.com)

linux0.11缓冲区管理程序阅读注释笔记_linux0.11内核阅读笔记-CSDN博客

Linux内核设计的艺术-关于缓冲块的进程等待队列_wait on buffer-CSDN博客



前言

看这篇文章之前，首先需要了解IO的几种方式，然后是生磁盘的使用，也就是磁盘驱动，接下来我们假定我们的

linux-0.11是DMA方式（实际不是，这样假设是为了后面解释代码方便）,也就是一个进程只要把块号block，内核缓冲区指针bh，这两个参数发送给磁盘控制器，然后这个进程挂起休眠让出CPU，CPU接下来该干什么干什么，直到磁盘驱动将block数据搬到bh缓冲块完成发出中断请求，才会唤醒进程，然后进程再把bh地方的数据搬到用户区处理。为了解释这种流程，我们需要举个例子说明，例子就是我们有2个进程，都是读取同一个文件，进程A读取100个字节到自己的用户区buffer，进程B读取200个字节到自己的用户区buffer，每个进程的核心代码如下：

```
void FunA()
{
    char buffer[100];
    int i,j;
    int fd = open("/mnt/user/user1/user2/hello.txt",O_RDWR,0644);
```

```
    read(fd,buffer,sizeof(buffer));
    close(fd);

    for(i=0;i<1000000;i++)
    {
        for(j=0;i<100000;j++)
        {
            ;
        }
    }

void FunB()
{
    char buffer[200];
    int i,j;
    int fd = open("/mnt/user/user1/user2/hello.txt",O_RDWR,0644);
    read(fd,buffer,sizeof(buffer));
    close(fd);

    for(i=0;i<1000000;i++)
    {
        for(j=0;i<100000;j++)
        {
            ;
        }
    }
}
```

进程 A：open()获取文件数据结构

用open函数通过路径解析找到文件的inode，并分配一个fd给进程的filp，后面通过filp[fd]即可找到这个文件的inode信息

进程 A：read()->sys_read()->file_read()

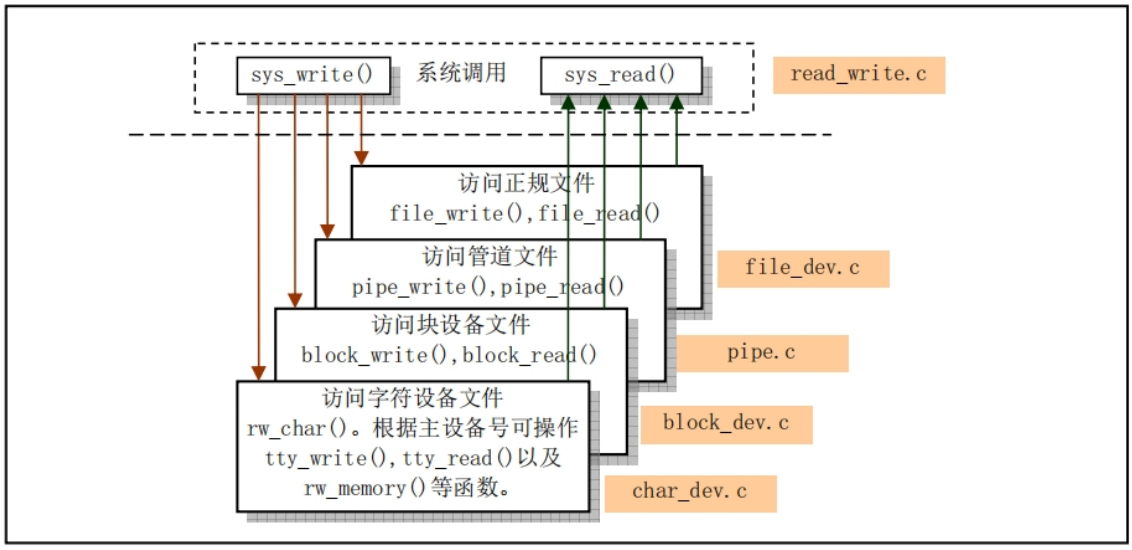


图 12-26 各种类型文件与文件系统和系统调用的接口函数

read(fd,buffer,sizeof(buffer))就是完成读文件fd中sizeof(buffer)个字符到buffer当中，但是不是一下子就读完的，具体是通过调用sys_read()里面的file_read(inode,file,buf,count)来完成读取数据操作。因此，真正核心的操作步骤主要在file_read()里面，我们从file_read()开始解读核心的流程。

```
int sys_read(unsigned int fd,char * buf,int count)
{
    struct file * file;
    struct m_inode * inode;

    if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
        return -EINVAL;
    ...
    inode = file->f_inode;
    ...
    if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
        if (count+file->f_pos > inode->i_size)
            count = inode->i_size - file->f_pos;
        if (count<=0)
            return 0;
        return file_read(inode,file,buf,count);
    }
    printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
    return -EINVAL;
}
```

```
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
{
    int left,chars,nr;
    struct buffer_head * bh;

    if ((left=count)<=0)
        return 0;
    while (left) {
        if ((nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE))) {
            if (!(bh=bread(inode->i_dev,nr)))
                break;
        } else
            bh = NULL;
        nr = filp->f_pos % BLOCK_SIZE;
        chars = MIN( BLOCK_SIZE-nr , left );
        filp->f_pos += chars;
        left -= chars;
        if (bh) {
            char * p = nr + bh->b_data;
            while (chars-->0)
                put_fs_byte(*(p++),buf++);
            brelse(bh);
        } else {
            while (chars-->0)
                put_fs_byte(0,buf++);
        }
    }
    inode->i_atime = CURRENT_TIME;
    return (count-left)?(count-left):-ERROR;
```

```
}
```

首先使用`struct buffer_head * bh;`定义一个缓冲块指针，在`bh=bread(inode->i_dev,nr)`中会申请一个缓冲块，并让磁盘把block的数据搬到bh缓冲块中，最后用一个`*p`指针作为源指针，buffer作为目标指针，在下面的循环中将缓冲块的数据搬到用户buffer中。

```
if (bh) {
            char * p = nr + bh->b_data;
            while (chars-->0)
                put_fs_byte(*(p++),buf++);
            brelse(bh);
        }
```

当然，进行到`bh=bread(inode->i_dev,nr)`时，也就是把磁盘block的数据搬到bh时，也不是一次性完成的，这步的工作具体流程是接下来讲解的重点

####

进程A：bread()->ll_rw_block()->make_request()->add_request

下面这段话引用linux内核完全注释修正版v3.0中第349页，先读完这段话后，再去看后面的解读

> 需要提醒的一件事是:在向硬盘或软盘控制器发送了读/写或其他命令后，发送命令函数并不会等待所发命令的执行过程，而是立刻返回调用它的程序中，并最终返回到调用块设备读写函数ll_rw_block的其他程序中去等待块设备IO的完成。例如高速缓冲区管理程序 fs/bufer.c 中的读块函数 bread)，在调用了ll_rw_block之后，就调用等待函数 wait_on_buffer让执行当前内核代码的进程立刻进入睡眠状态，直到在相关块设备IO结束，在end_request函数中被唤醒。

bread()

```
struct buffer_head * bread(int dev,int block)
{
    struct buffer_head * bh;

    if (!(bh=getblk(dev,block)))           // 申请缓冲区
        panic("bread: getblk returned NULL\n");
    if (bh->b_uptodate)
        return bh;
    ll_rw_block(READ,bh);                  // 发送搬运指令
    wait_on_buffer(bh);                    // 等待缓冲区数据
    if (bh->b_uptodate)
        return bh;                         // 搬运成功，返回bh指针
    brelse(bh);                            // 如果失败释放缓冲区返回NULL
    return NULL;
}
```

首先`bh=getblk(dev,block)`会申请相应设备相应block对应的缓冲块，对于进程B申请的也是同样的，因此涉及到共享资源的使用，必定涉及到同步互斥问题。因此接下来它是这样做的

```
    ll_rw_block(READ,bh);                  // 发送搬运指令
    wait_on_buffer(bh);                    // 等待缓冲区数据
```

也就是发送读的指令以及bh（由于和block对应，相当于也发送了block信息）给磁盘控制器，磁盘控制器收到信号，就知道需要将block搬运到bh中（如果是写指令，就是bh搬到block中）。

但是并不是这样的顺序执行的，而是发送完搬运指令后，磁盘开始在背后工作，cpu继续往后执行`wait_on_buffer(bh);`这句代码意思就是等待bh是不是搬完了，如果没搬完，进程A挂起等着，如果搬完了，那么就往后执行，也就是`return bh;`把缓冲块指针bh返回，这时候回到前面的函数执行后面的代码就能把bh的数据搬到buffer当中了。

所以接下来的重点是搞清楚，是如何做到"等缓冲区数据准备好，再去往后执行，如果没准备好，那就挂起休眠"?

背后潜在还包括"在没搬好数据之前，这个bh不能被其他读取进程访问"这件事。

那么接下来围绕上面两行代码的细节来解释这两点：

首先是发送读请求的代码

```
void ll_rw_block(int rw, struct buffer_head * bh)   // 接口函数 检查参数 调用
make_request
{
    // ......
    make_request(major,rw,bh);
}
```

```
static void make_request(int major,int rw, struct buffer_head * bh)
{
    // ......
    lock_buffer(bh);
    if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
        unlock_buffer(bh);
        return;
    }
// ......
    req->dev = bh->b_dev;   // 设备号
    req->cmd = rw;  // 命令
    req->errors=0;  // 操作时产生的错误次数
    req->sector = bh->b_blocknr<<1; // 起始扇区。块号转换成扇区号。
    req->nr_sectors = 2;    // 本请求项需要读写的扇区数 扇区数为2
    req->buffer = bh->b_data;   // 请求项缓冲区指针指向需读写的数据缓冲区
    req->waiting = NULL;    // 任务等待操作执行完成的地方
    req->bh = bh;   // 缓冲块头指针
    req->next = NULL;   // 指向下一请求项
    add_request(major+blk_dev,req); // 将请求项加入队列中(blk_dev[major],req)
}
```

```
static inline void lock_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)//如果已经上了锁，就要等待了
        sleep_on(&bh->b_wait);
    bh->b_lock=1;
    sti();
}
```

首先有一句`lock_buffer(bh);`它是给bh加锁，这句代码就是给bh上锁，防止后面磁盘往bh搬数据的时候其他进程使用bh，比如调度程序执行到进程B，执行顺序分别为`read()->sys_read()->file_read()->bread()->ll_rw_block()->make_request()->lock_buffer(bh)->强制进入休眠`，看完生产者消费者问题后不难知道，这句代码实际就是`P(bh->b_lock)`。

ok，首先挡住所有进程访问bh的锁到这里加上了，接下来进程A会执行`add_request(major+blk_dev,req)`，也就是向设备发出请求，这时候设备就开始干活了（linux内核完全注释修正版v3.0中第349页）。

注意：我们这里就是简单向设备发送一个请求，实际这个请求有个全局的请求队列（32个请求的数组），为了简单起见，这里简单向设备发送一个请求，然后设备开始干活。对于请求队列的加锁解锁，不是这里讲解的重点，这里仅仅关注共享缓冲区bh的加锁解锁。

然后make_request执行结束回到ll_rw_block，然后ll_rw_block也执行结束回到了bread的ll_rw_block下一行`wait_on_buffer`，代码片段如下：

```
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    sti();
}
```

`wait_on_buffer`等待资源bh，如果没有准备好，也就是现在上锁的状态，进程A直接进入休眠。ok，分析到这里，我们知道进程A和进程B都处于休眠状态，并且它们各自卡在哪里，从哪里休眠的我们都已经讲清楚了。此时cpu在干别的事情，然后磁盘正在源源不断向bh输送数据。当磁盘输送数据结束的那一刻开始，会给cpu发送一个中断信号，直接跳到中断函数`read_intr`执行。

磁盘中断：read_intr

磁盘会在完成某个请求或者需要的时刻向cpu发送信号引发磁盘中断，跳转到磁盘驱动程序执行。

硬盘中断处理过程中调用的C函数，如read_intr、write_intr()、bad_rw_intr()、recal_intr()

控制器读操作完成后调用的函数 read_intr()

- 调用 win_result() 读取状态寄存器，判断错误是否发生。
- 若有错误，读盘错误则调用 bad_rw_intr() 判断是否需要放弃此请求项，是否需要设置复位标志。
- 若无错误，使用 port_read() 函数从控制器缓冲区把一个扇区的数据复制到请求项指定的缓冲区中。
- 若还有数据要读，退出，等待下一个中断的到来。
- 调用 end_request() 函数来处理当前请求项的结束事宜：唤醒等待本请求项完成的进程、唤醒等待空闲请求项的进程、设置当前请求项所指缓冲区数据已更新标志、释放当前请求项（从块设备链表中删除该项）。
- 重新调用 do_hd_request() 函数。

```
static void read_intr(void)
{
    //...
    end_request(1);
    do_hd_request();
}
```

```
void do_hd_request(void)
{
    .....
    //得到dev,nsect,sec,head,cyl，WIN_WRITE,&write_intr 参数
    // dev 控制器，nsect 读写几个扇区，cyl 柱面，head 磁头，sec 扇区，WIN_WRITE 写，
&write_intr 缓存位置
    //传递给磁盘控制器
    if (CURRENT->cmd == WRITE) {
        hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
        for(i=0 ; i<3000 && !(r=inb_p(HD_STATUS)&DRQ_STAT) ; i++)
            /* nothing */ ;
        if (!r) {
            bad_rw_intr();
            goto repeat;
        }
        port_write(HD_DATA,CURRENT->buffer,256);
    } else if (CURRENT->cmd == READ) {
        hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
    } else
        panic("unknown hd-command");
}
```

```
extern inline void end_request(int uptodate)
{
    DEVICE_OFF(CURRENT->dev);   // 关闭设备
    if (CURRENT->bh) {  // CURRENT 为当前请求结构项指针
        CURRENT->bh->b_uptodate = uptodate; // 置更新标志
        unlock_buffer(CURRENT->bh); // 解锁缓冲区
    }
    if (!uptodate) {    // 此次请求项操作失败
        printk(DEVICE_NAME " I/O error\n\r");   // 显示相关块设备IO出错信息
        printk("dev %04x, block %d\n\r",CURRENT->dev,
            CURRENT->bh->b_blocknr);
    }
    wake_up(&CURRENT->waiting); // 唤醒等待该请求项的进程
    wake_up(&wait_for_request); // 唤醒等待空闲请求项出现的进程
    CURRENT->dev = -1;  // 释放该请求项
    CURRENT = CURRENT->next;    // 指向下一个请求项
}
```

发生磁盘中断后会调用read_intr，使用end_request(1)唤醒前面因为bh阻塞的两个进程，也就是进程A和进程B，然后再次调用do_hd_request搬运数据，搬完后会发送一个中断，再次调用read_intr，就这样反反复复。

然后我看完哈工大生磁盘使用的视频后，觉得这个工作其实不需要CPU参与，CPU仅仅发送几个简单指令，真正搬运的工作CPU是不参与的。目前我的理解是这样的，进程A发送请求后，CPU该干嘛干嘛，然后磁盘控制器处理请求队列里面的第一个，假如这时候磁盘干完活了，会发送中断信号调用read_intr来唤醒阻塞的进程，并继续使用do_hd_request处理下一个请求。

do_hd_request仅仅只是向磁盘控制器发送几个指令，发送完成后，cpu很快就会结束中断处理函数，该该嘛干嘛，然后磁盘控制器通过刚刚接收到的设置，来把一整块数据慢慢搬到相应的缓冲块。

这里就几个点：

1. end_request(1)相当于V(bh->b_lock)，将bh解锁
2. sleep_on和wake_up成对使用，这里包含一个隐藏的休眠队列，队列怎么形成的，怎么挨个唤醒的，这里不讲了

唤醒后的调度

假如两个进程唤醒后，进程A首先被调度，这时候会执行wait_on_buffer下一句，然后就是返回bh，让file_read把数据从bh搬到用户去buffer就完事了。然后进程B被调度，进程B是在下面的代码中卡住休眠的：

```
lock_buffer(bh);
if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
    unlock_buffer(bh);
    return;
}
```

```
static inline void lock_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)//如果已经上了锁，就要等待了
        sleep_on(&bh->b_wait);
    bh->b_lock=1;
    sti();
}
```

实际上就是下面这里卡住的：

```
while (bh->b_lock)
    sleep_on(&bh->b_wait);
```

由于此时bh已经解锁了，所以唤醒的时候再次执行这个循环的时候，会跳出循环，然后执行下一句，也就是`bh->b_lock=1`,然后执行下面的代码：

```
if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
    unlock_buffer(bh);
    return;
}
```

简单来讲，就是检查bh到底有没有准备好，由于已经准备好了，直接解锁bh，直接返回，后面不需要重复发送请求了。也就是从make_request()返回到ll_rw_block()，然后返回到bread()，继续执行ll_rw_block()下一句代码，也就是wait_on_buffer，由于上面已经解锁了bh，这句代码直接执行过去，然后就顺利返回bh，然后返回到file_write中，把bh数据搬到buffer中。

最后留个疑问：假如一开始进程B首先被调度呢，情况如何？

结语

由于本人对文件读写这一块不是很熟，只看了几天，涉及到太多的东西，涉及到缓冲区的管理（使用的bh咋就是同一个bh？bh里面除了有锁，其他信息又是什么意思，干什么用的？），请求队列，设备，整个磁盘驱动程序如何工作的，里面包含了方方面面的东西（就是一开始main程序里面好几个初始化工作，初始化哪些全局数据结构，每一个数据结构有哪些算法），因此想彻底搞懂需要花大量时间全部一个一个攻破，不然像我上面这种模棱两可的分析，必然有很多不严谨和错误的地方。

因此，对于有时间的同学，还是建议把操作系统，组成原理涉及到IO和进程同步互斥的地方反复观看，然后把read和write涉及到的方方面面的源码都拿出来学学，这样才能彻底搞懂linux-0.11是如何工作的。但是，个人感觉操作系统，组成原理里面对IO设备驱动讲的好像也不是特别详细，感觉可能单独学习《Linux设备驱动》相关的课程或者书籍才能搞清楚。