第四章，Setup驱动器

有王伟强《图像处理》和罗平《高级人工智能》这两个课程学得好的同学请联系我！

我这两个课程感觉快挂科了TAT，我操作系统和计算机体系结构学得还行，可以互助一下

email: lujunfeng@junfeng.lu

QQ:464270342

上面是我的联系方式，希望大佬能出手相助

之前讲到，在fork出进程1之后，进程1进行了init操作。

```
static inline _syscall1(int,setup,void *,BIOS)
void init(void)
{
    int pid,i;

    setup((void *) &drive_info);
    .......
```

其中init操作是调用了`setup`这个函数。

`setup`这个函数也是一个系统调用，只不过他用的宏是`_syscall1`

第一步，sys_setup()

```
int sys_setup(void * BIOS)
{
    static int callable = 1;
    int i,drive;
    unsigned char cmos_disks;
    struct partition *p;
    struct buffer_head * bh;
    。。。。。。。
    for (drive=0 ; drive<NR_HD ; drive++) {
        if (!(bh = bread(0x300 + drive*5,0))) {//b read:读取一个块，目前是在读引导块，参数是（dev,blk）
            printk("Unable to read partition table of drive %d\n\r",
                drive);
            panic("");
        }
```

这个函数，开头的大量部分是设置与硬盘相关的参数。这部分可以先略过。

我们的重点是这个`bread`函数

关键部分，块设备读取bread()

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

    if (!(bh=getblk(dev,block)))//申请一个空的缓冲块
        panic("bread: getblk returned NULL\n");
    if (bh->b_uptodate) //是否已经更新到最新了
        return bh;
    ll_rw_block(READ,bh);//读这个缓冲区
    wait_on_buffer(bh);//等到读完
    if (bh->b_uptodate)//如果已经uptodate那么就返回，否则，这块有大毛病了
        return bh;
    brelse(bh);//释放这个块
    return NULL;
}
```

这一个函数实际上是做了几件事情:

- 给对应的设备申请一个缓冲块
- 如果发现这个设备的缓冲块已经是最新的了
    - 那么直接返回这个缓冲块
- 否则，把设备的内容读到这个缓冲块中
    - 如果读取正常，那么返回这个缓冲块

getblk() （part 1 试图找到之前打开的缓冲块）

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

repeat:
    if (bh = get_hash_table(dev,block))
        return bh;//如果发现之前打开了buffer，那就直接用

    。。。。
```

以上 是上半截的内容。

首先，试图从Hash表中找一下，看看之前有没有打开过这个缓冲块。

如果找到了，那就直接返回，其具体的代码是这样的

get_hash_table() 试图在哈希表上查找，并进行校验

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

    for (;;) {
        if (!(bh=find_buffer(dev,block)))   //如果找不到之前打开过的buffer，那就缓冲一
个null
            return NULL;
        bh->b_count++;  //如果之前打开过这个buffer，增加引用计数
        wait_on_buffer(bh);     //等待buffer空闲为止
```

```
        if (bh->b_dev == dev && bh->b_blocknr == block)//如果这个块没有被修改过，那么就返回这个块
            return bh;
        bh->b_count--;//否则这个块不是我们想要的，把引用次数还原
    }
}
```

这个函数里面，通过调用`find_buffer`函数，来试图查找一个之前打开过的buffer.

找到之后，将其引用次数+1。并且等到没有人在占用这个buffer为止。

然后检查这个buffer是否打开了我们要找的那个设备，如果不是，那么再试图寻找一次。

find_buffer() 在哈希表上查找

```
#define _hashfn(dev,block) (((unsigned)(dev^block))%NR_HASH)
#define hash(dev,block) hash_table[_hashfn(dev,block)]
static struct buffer_head * find_buffer(int dev, int block)
{
    struct buffer_head * tmp;
    // 用Hash函数：(dev^block))%NR_HASH 找到一个块
    // 然后开始顺着链表往下找
    for (tmp = hash(dev,block) ; tmp != NULL ; tmp = tmp->b_next)
        if (tmp->b_dev==dev && tmp->b_blocknr==block)
            return tmp;//如果在这个链表上找到了完全Match的dev和block，那么就返回这一个块。
    return NULL;
}
```

这个才是真正在哈希表上寻找的函数。之前说过了，缓冲区有两个链

- 一个是环形链（free list），把所有的节点给串起来
- 第二个是哈希表，每个哈希表上挂着哈希值相同的节点

这个是在对应哈希值的链上寻找，看一下这个链上有没有一个节点刚好打开了我们所需要的块设备。

wait_on_buffer() 等待buffer解锁

```
static inline void wait_on_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)//等待到其他进程不锁定这个buffer为止
        sleep_on(&bh->b_wait);
    sti();
}
```

顾名思义，这个函数的意思是，等待到没有其他进程锁定这个buffer为止（因为可能在从设备读取数据中）

实际上就是调用`sleep_on`这个函数，等待`bh->b_wait`这个进程

sleep on 隐式的等待队列

```
void sleep_on(struct task_struct **p)
{
```

```
    struct task_struct *tmp;
    if (!p)//如果没有task_struct，那么啥都不干
        return;
    if (current == &(init_task.task))
        panic("task[0] trying to sleep");
    tmp = *p;// tmp = b_wait 此时为NULL
    *p = current;
    current->state = TASK_UNINTERRUPTIBLE;//把当前进程设置为不可中断（在等待中，所以不
响应中断）
    schedule();//切换到下一个运行的程序
    if (tmp)//这是此前的队头程序，也把这个程序唤醒
        tmp->state=0;
}
```

这个sleep_on机制比较有意思。

这个`struct task_struct **p`实际上是一个链尾指针，指向了最后一个在等待该事件的进程。

- 一个新进程在等待的时候，会自己用`tmp`变量去记录上一个在等待中的进程，然后把自己的指针放回`p`处。
- 然后把自己设置为不可被中断的等待
- 进行进程切换

当被唤醒之后：

- 把前一个进程也设置为可启动的状态

也就是，每一个进程只唤醒自己的前一个进程。

getblk 第二步：找到一个victim

```
#define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
    if (bh = get_hash_table(dev,block))
        return bh;//如果发现之前打开了buffer，那就直接用
    tmp = free_list;
    do {
        if (tmp->b_count)//如果这个是真的free的，即，没人在用，那就继续，否则继续等待
            continue;
        if (!bh || BADNESS(tmp)<BADNESS(bh)) {
//认为脏比被锁更不好
            bh = tmp;
            if (!BADNESS(tmp))  //如果这个freelist的头是能用的，那就break
                break;
        }
/* and repeat until we find something good */
    } while ((tmp = tmp->b_next_free) != free_list);
```

如果之前没在hash表中找到之前打开过的块，那么就必须得自己新创建一个块了。

- 直接从`freelist`中拿一个节点出来。
- 因为这个缓冲块实际上有点类似于缓存，一般情况下是不会被主动换出的，因此这个freelist中的节点不一定是真的free,第一步是找到一个没人在用的节点
- 然后，我们认为脏的节点比被锁的节点更不好，因为脏的节点要写回，要写一万年，被锁的节点，其他的进程用完之后就很快返回了

接着，我们需要确保的是，**找到的节点是好使的**：

如果没有空闲节点，那就等待，然后重新查找

```
if (!bh) {
        sleep_on(&buffer_wait);
        goto repeat;
    }
```

之前在freelist上查找的过程中，如果找完了所有的缓冲块，但是都有人在用，那么此时bh就是空的，此时需要等待到没人用为止，然后继续查找。

如果还有人在用这个块，那么再找一下

```
    wait_on_buffer(bh);
    if (bh->b_count)
        goto repeat;
```

因为这个函数执行的过程中，可能突然发生了进程切换，所以在这里还需要再进行一下检查，确保真的没人用了。

如果这个块是脏的，那么需要写回

```
while (bh->b_dirt) {
        sync_dev(bh->b_dev);
        wait_on_buffer(bh);
        if (bh->b_count)
            goto repeat;
    }
```

实际上就是调用了`sync_dev`v这个函数。然后进入等待。

如果等待回来之后发现，别人拿走了这个缓冲区，那么还需要重复申请一下

检查一下是不是有人偷偷拿走了这一块

```
/* NOTE!! While we slept waiting for this block, somebody else might */
/* already have added "this" block to the cache. check it */
    if (find_buffer(dev,block))
        goto repeat;
```

因为之前有各种Sleep的操作，因此可能等待过程中，别人偷偷拿走了这一块。

拿走这一块的最终标志是挂在了Hash表上，所以如果我们在hash表上找到了一个块，那么说明已经被用了

getblk第三步：设置块的各项属性，在链表上进行修改，然后返回

```
/* OK, FINALLY we know that this buffer is the only one of it's kind, */
/* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
    bh->b_count=1;
    bh->b_dirt=0;
    bh->b_uptodate=0;
    remove_from_queues(bh);
    bh->b_dev=dev;
    bh->b_blocknr=block;
    insert_into_queues(bh);
    return bh;
```

此时我们已经得到了一个没人使用（`b_count=0`）且非锁定(`b_lock=10`)并且是干净的块。

所以我们要进行一些初始化的设置。注意，此时我们**设置为没有uptodate**,因为我们只是打开了缓冲区，没进行磁盘的读写

接着，先把这个节点从两个链表上都摘出来。

```
static inline void remove_from_queues(struct buffer_head * bh)
{
/* remove from hash-queue */
    if (bh->b_next)
        bh->b_next->b_prev = bh->b_prev;
    if (bh->b_prev)
        bh->b_prev->b_next = bh->b_next;
    if (hash(bh->b_dev,bh->b_blocknr) == bh)
        hash(bh->b_dev,bh->b_blocknr) = bh->b_next;
/* remove from free list */
    if (!(bh->b_prev_free) || !(bh->b_next_free))
        panic("Free block list corrupted");
    bh->b_prev_free->b_next_free = bh->b_next_free;
    bh->b_next_free->b_prev_free = bh->b_prev_free;
    if (free_list == bh)
        free_list = bh->b_next_free;
}
```

摘出来之后，修改对应的驱动号。这样子是保证在Hash表上的所有节点都满足Hash值相等。

最后再插回链表上。

注意，这一个块是插入到了free_list的末端，这样子就有点类似于FIFO的结构了。

```
static inline void insert_into_queues(struct buffer_head * bh)
{
/* put at end of free list */
    bh->b_next_free = free_list;
    bh->b_prev_free = free_list->b_prev_free;
    free_list->b_prev_free->b_next_free = bh;
    free_list->b_prev_free = bh;
/* put the buffer in new hash-queue if it has a device */
    bh->b_prev = NULL;
    bh->b_next = NULL;
    if (!bh->b_dev)
        return;
    bh->b_next = hash(bh->b_dev,bh->b_blocknr);
    hash(bh->b_dev,bh->b_blocknr) = bh;
    bh->b_next->b_prev = bh;
```

```
}
```

弄好这些后，我们就可以返回找到的空闲块了。

回到Bread中

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

    if (!(bh=getblk(dev,block)))//申请一个空的缓冲块
        panic("bread: getblk returned NULL\n");
    if (bh->b_uptodate) //是否已经更新到最新了
        return bh;
    ll_rw_block(READ,bh);//读这个缓冲区
    wait_on_buffer(bh);//等到读完
    if (bh->b_uptodate)//如果已经uptodate那么就返回，否则，这块有大毛病了
        return bh;
    brelse(bh);//释放这个块
    return NULL;
}
```

目前我们已经从getblk返回了。如果这个块已经是`uptodate`的了，那就可以直接用。否则需要进行读取

用的是`ll_rw_block()`函数

ll_rw_block()读取块设备

```
void ll_rw_block(int rw, struct buffer_head * bh)
{
    unsigned int major;
    // #define MAJOR(a) (((unsigned)(a))>>8)
    // #define MINOR(a) ((a)&0xff)
    if ((major=MAJOR(bh->b_dev)) >= NR_BLK_DEV ||  //驱动器号大于驱动器数量
    !(blk_dev[major].request_fn)) { // 或者这个驱动器没有请求函数，那么报错
        printk("Trying to read nonexistent block-device\n\r");
        return;
    }
    make_request(major,rw,bh);
}
```

此处对major进行了一下判断，这个dev的值是在`sys_setup`中设置的

```
bh = bread(0x300 + drive*5,0)
```

即，`major`是`3`，而为什么是3呢？因为之前在main函数中，调用`hd_init`的时候，把相关的东西挂在了#3上

```
struct blk_dev_struct blk_dev[NR_BLK_DEV] = {
    { NULL, NULL },     /* no_dev */
    { NULL, NULL },     /* dev mem */
    { NULL, NULL },     /* dev fd */
    { NULL, NULL },     /* dev hd */
```

```
    { NULL, NULL },     /* dev ttyx */
    { NULL, NULL },     /* dev tty */
    { NULL, NULL }      /* dev lp */
};
#define MAJOR_NR 3
void hd_init(void)
{
    blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
    ....
}
```

实际上，这个函数就是检查了一下，想要读取的块设备是否有效（ID没超上限，并且有请求函数）

然后就调用`make_request(major,rw,bh);`

make_request() 第一步 检查相关状态

```
                            //3         read                bh
static void make_request(int major,int rw, struct buffer_head * bh)
{
    struct request * req;
    int rw_ahead;

/* WRITEA/READA is special case - it is not really needed, so if the */
/* buffer is locked, we just forget about it, else it's a normal read */
    if (rw_ahead = (rw == READA || rw == WRITEA)) {
        if (bh->b_lock)
            return;
        if (rw == READA)
            rw = READ;
        else
            rw = WRITE;
    }
    if (rw!=READ && rw!=WRITE)
        panic("Bad block dev command, must be R/W/RA/WA");
    lock_buffer(bh);
    if ((rw == WRITE && !bh->b_dirt) || (rw == READ && bh->b_uptodate)) {
        unlock_buffer(bh);
        return;
    }
```

这部分内容首先判断了：是否是预读/预写

这种是为了提升性能的提前操作，所以假设失败了也不影响正常运行。因此该状态下，遇到blk被锁，直接返回。

接着判断，是否为不支持的操作，如果遇到了不支持的操作（不是读或者写），那进入panic

接着，先**锁住这个buffer**，判断一下是否需要接着进行操作。

如果遇到这两种情况，那么没必要继续进行了：

- 需要写块设备，但是缓冲区是干净的
- 需要读块设备，但是缓冲区已经同步过了

这两种情况，直接解锁缓冲区，然后返回就行了。

锁定和解锁缓冲区的代码分别如下：

```
static inline void lock_buffer(struct buffer_head * bh)
{
    cli();
    while (bh->b_lock)
        sleep_on(&bh->b_wait);
    bh->b_lock=1;
    sti();
}
static inline void unlock_buffer(struct buffer_head * bh)
{
    if (!bh->b_lock)
        printk("ll_rw_block.c: buffer not locked\n\r");
    bh->b_lock = 0;
    wake_up(&bh->b_wait);
}
```

make_request()第二步 找到一个空的请求项

```
struct request request[NR_REQUEST];
////////////////////////////////////

repeat:
/* we don't allow the write-requests to fill up the queue completely:
 * we want some room for reads: they take precedence. The last third
 * of the requests are only for reads.
 */
    if (rw == READ)
        req = request+NR_REQUEST;
    else
        req = request+((NR_REQUEST*2)/3);
        //req = &request[NR_REQUEST]
/* find an empty request */
    while (--req >= request)
        if (req->dev<0)
            break;
/* if none found, sleep on new requests: check for rw_ahead */
    if (req < request) {
        if (rw_ahead) {
            unlock_buffer(bh);
            return;
        }
        sleep_on(&wait_for_request);
        goto repeat;
    }
```

这里，`request`是一个数组，因此`req = request+NR_REQUEST;`这个条目完全可以用如下代码表示：

`req = *request[NR_REQUEST]`,这也就是为什么`while (--req >= request)`中req需要先`--`再比较

之前我们在`blk_dev_init()`v中，将所有的请求项的dev设置成了`-1`代表这个请求项是空的。

如果没找到空的请求项，那么分两种情况：

- 如果是预读写，那么直接返回（预读写失败不影响正常运行）
- 否则，等待到有空闲的请求项为止

```
void blk_dev_init(void)
{
    int i;

    for (i=0 ; i<NR_REQUEST ; i++) {
        request[i].dev = -1;
        request[i].next = NULL;
    }
}
```

make_request() 第三步 发出请求

```
req->dev = bh->b_dev;
    req->cmd = rw;
    req->errors=0;
    req->sector = bh->b_blocknr<<1;
    req->nr_sectors = 2;
    req->buffer = bh->b_data;
    req->waiting = NULL;
    req->bh = bh;
    req->next = NULL;
    add_request(major+blk_dev,req);
```

现在我们已经有了一个空的请求项，那么我们就可以直接进行发送请求了

这里的`major+blk_dev`实际上等效于`&blk_dev[major]`

add_request() 第一种情况，之前没有请求，直接处理

```
static void add_request(struct blk_dev_struct * dev, struct request * req)
{
    struct request * tmp;

    req->next = NULL;
    cli();
    if (req->bh)
        req->bh->b_dirt = 0;
    if (!(tmp = dev->current_request)) {//如果这设备没有请求，那么直接发出请求即可
        dev->current_request = req;
        sti();
        (dev->request_fn)();
        return;
    }
    ......
```

在这里进行了一下判断，如果这个设备的当前请求为空，那么可以直接进行该请求并返回。

add_request() 第二种情况，之前有请求，那么插入到链表中去

```
    for ( ; tmp->next ; tmp=tmp->next)//电梯算法
        if ((IN_ORDER(tmp,req) ||
            !IN_ORDER(tmp,tmp->next)) &&
            IN_ORDER(req,tmp->next))
            break;
    req->next=tmp->next;
    tmp->next=req;
    sti();
```

tmp是一个扫描之前请求的指针。

而req是当前的新请求。

如果找到到了一个位置，要么`<tmp,req>`是顺序的，要么`<req,tmp->next>`是顺序的，就把这个请求插在该位置。

如果都找不到，此时就是在末尾，直接把当前请求给插到末尾即可。

PS：请求项的处理，是每一个块设备的`do_request`,然后处理`current_request`这个请求项。处理完成之后，再把链表上的下一个请求拿出来继续处理的。

do_hd_request()

```
void do_hd_request(void)
{
    int i,r;
    unsigned int block,dev;
    unsigned int sec,head,cyl;
    unsigned int nsect;

    INIT_REQUEST;
    dev = MINOR(CURRENT->dev);
    block = CURRENT->sector;
    if (dev >= 5*NR_HD || block+2 > hd[dev].nr_sects) {
        end_request(0);
        goto repeat;//repeat 在INIT--REQUEST里面
        /*#define INIT_REQUEST \
repeat: \
……
*/
    }
```

这里有一点值得注意，那就是发现需要读的地方不太对的时候，用了一句`goto repeat`。这个repeat并没有直接出现在前面。

因为这个是用`INIT_REQUEST`这个宏定义展开得到的。

然后中间是一大堆和设备控制相关的代码，重点关注的是后面的`hd_out`这个函数是真正的给硬盘下命令了。

```
......
    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);
        //最关键的，读的时候
    } else
        panic("unknown hd-command");
......
```

hd_out() 给硬盘发命令，并注册硬盘中断回调函数

```
static void hd_out(unsigned int drive,unsigned int nsect,unsigned int sect,
        unsigned int head,unsigned int cyl,unsigned int cmd,
        void (*intr_addr)(void))
{
    register int port asm("dx");

    if (drive>1 || head>15)
        panic("Trying to write bad sector");
    if (!controller_ready())
        panic("HD controller not ready");
    do_hd = intr_addr;//这个是中断地址（read_intr） 和硬盘中断服务程序挂钩
    outb_p(hd_info[drive].ctl,HD_CMD);
    port=HD_DATA;
    outb_p(hd_info[drive].wpcom>>2,++port);
    outb_p(nsect,++port);
    outb_p(sect,++port);
    outb_p(cyl,++port);
    outb_p(cyl>>8,++port);
    outb_p(0xA0|(drive<<4)|head,++port);
    outb(cmd,++port);
}
```

这一部分的代码，主要做的操作是，

- 检查相关参数是否正常
- **注册回调函数**
- 启动硬盘进行读写操作

  重点注意注册回调函数这个部分，在进行`do_hd_request`的时候，有两种可能的函数：
  `hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);`
  `hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);`

也就是说，读写完成时，硬盘返回的中断会根据挂载的不同回调函数去到不同的函数进行执行。

发送完成硬盘请求后，进程1进入休眠

```
    ll_rw_block(READ,bh);//读这个缓冲区
    wait_on_buffer(bh);//等到读完
    if (bh->b_uptodate)//如果已经uptodate那么就返回，否则，这块有大毛病了
        return bh;
```

我们用`ll_rw_block`向硬盘发送了请求，现在需要等待中断返回。

于是进程1用`wait_on_buffer`命令将自己进入休眠状态，直到被缓冲区相关机制唤醒。

此时在运行的，是**进程0**，他会不断的在死循环中试图调度其他的程序。

也就是说，只要进程1一进入了可运行状态，进程0就会马上把进程1给唤醒起来。

（虽然此时所有进程都是不可运行的，但是默认情况下，会把进程0给启起来）

进程0运行时硬盘完成请求，发来了中断

之前初始化硬盘时，设置了硬盘中断入口

```
void hd_init(void)
{
    blk_dev[MAJOR_NR].request_fn = DEVICE_REQUEST;
    set_intr_gate(0x2E,&hd_interrupt);  //设置中断的入口
    outb_p(inb_p(0x21)&0xfb,0x21);
    outb(inb_p(0xA1)&0xbf,0xA1);
}
```

这个硬盘中断的代码是：

```
_hd_interrupt:
    ......
    outb %al,$0xA0      # EOI to interrupt controller #1
    jmp 1f          # give port chance to breathe
1:  jmp 1f
1:  xorl %edx,%edx  //清零
    xchgl _do_hd,%edx//将_do_hd和edx交换，这是在hd_out上出现的（可能被赋成read_intr或者
write_intr)
    testl %edx,%edx//看一下_do_hd是否为NULL
    jne 1f
    movl $_unexpected_hd_interrupt,%edx
1:  outb %al,$0x20//停止硬盘中断
    call *%edx      # "interesting" way of handling intr.
                    //跳到_do_hd中断处理程序里面
```

简单地说，这个代码会判断一下`_do_hd`，即回调函数的接口是否为NULL，如果为NULL，那就报错。

如果正常，那么停止硬盘的中断，并且调到之前的回调函数里面。

进入读的回调函数

```
static void read_intr(void)
{
    if (win_result()) {
        bad_rw_intr();
        do_hd_request();
```

```
        return;
    }
    port_read(HD_DATA,CURRENT->buffer,256);
    CURRENT->errors = 0;
    CURRENT->buffer += 512;
    CURRENT->sector++;
    if (--CURRENT->nr_sectors) {
        do_hd = &read_intr;//还没读完，那么就继续挂上
        return;
    }
    end_request(1);
    do_hd_request();
}
```

如果我们是读硬盘，那么会进入到这个回调函数里面

如果发现得到了`bad_rw_intr`那么需要再做一遍。

如果发现没读完，那么继续把回调函数挂上，然后继续等待中断的到来。

如果读完了，那么结束当前的这个请求，并且继续做其他的请求

结束请求并挂载下一个请求

```
extern inline void end_request(int uptodate)//uptodate=1由read_intr调用
{
    DEVICE_OFF(CURRENT->dev);//关掉这个dev
    if (CURRENT->bh) {
        CURRENT->bh->b_uptodate = uptodate;
        unlock_buffer(CURRENT->bh);
    }
    if (!uptodate) {//不 uptodate说明是炸了
        printk(DEVICE_NAME " I/O error\n\r");
        printk("dev %04x, block %d\n\r",CURRENT->dev,
            CURRENT->bh->b_blocknr);
    }
    wake_up(&CURRENT->waiting);
    wake_up(&wait_for_request);
    CURRENT->dev = -1;
    CURRENT = CURRENT->next;
}
```

这里我们关心最后的两行。

```
CURRENT->dev = -1;
CURRENT = CURRENT->next;
```

这是把CURRENT，即当前请求项给释放，然后把链表上的下一个请求给挂过来。

在这之前，唤醒两个队列中的请求:

- 因当前请求项而等待的进程队列
- 因为申请不到请求项而等待的进程队列

```
wake_up(&CURRENT->waiting);
wake_up(&wait_for_request);
```

唤醒等待队列

```
#define TASK_RUNNING        0
#define TASK_INTERRUPTIBLE  1
#define TASK_UNINTERRUPTIBLE    2
#define TASK_ZOMBIE     3
#define TASK_STOPPED        4
void wake_up(struct task_struct **p)
{
    if (p && *p) {
        (**p).state=0;//task running
        *p=NULL;//不等了
    }
}
```

这个意思是，如果当前有等待中的进程，那么把当前进程设置为可运行

我们的等待队列实际上是一个尾插的链表，这会同时把链表头指针给设置NULL，也就说目前已经不存在等待队列了。

然后当下一次进程调度的时间到来之后，被设置为唤醒状态的进程会恢复执行：

```
void sleep_on(struct task_struct **p)
{
    。。。。。。。
    schedule();//切换到下一个运行的程序
    if (tmp)//这是链表中上一个程序，也把这个程序唤醒
        tmp->state=0;
}
```

回来之后，第一个执行的命令就是把链表中，他的上一个程序也给唤醒了。

这样，一个唤醒一个，最终实现每一个等待中的进程都唤醒

再次回到Bread中

```
struct buffer_head * bread(int dev,int block)
{
    。。。。。
    ll_rw_block(READ,bh);//读这个缓冲区
    wait_on_buffer(bh);//等到读完
    if (bh->b_uptodate)//如果已经uptodate那么就返回，否则，这块有大毛病了
        return bh;
    brelse(bh);//释放这个块
    return NULL;
}
```

由于在被设置为可运行，到真正被调度中间还间隔了一段举例，同时硬盘读取本身也可能失效。

所以如果发现没有`b_uptodate`那么就释放这个块，并且返回。

否则返回已经读取好的块。

释放块的代码如下：

```
void brelse(struct buffer_head * buf)
{
    if (!buf)
        return;
    wait_on_buffer(buf);
    if (!(buf->b_count--))//这个count不能为0，否则在试图释放一个没人用的buffer
        panic("Trying to free free buffer");
    wake_up(&buffer_wait);//唤醒等待中的进程
}
```

最终，我们回到了`sys_setup`函数