第47回 | 读取硬盘数据的细节

Original 闪客 低并发编程 2022-08-14 17:30 Posted on 北京

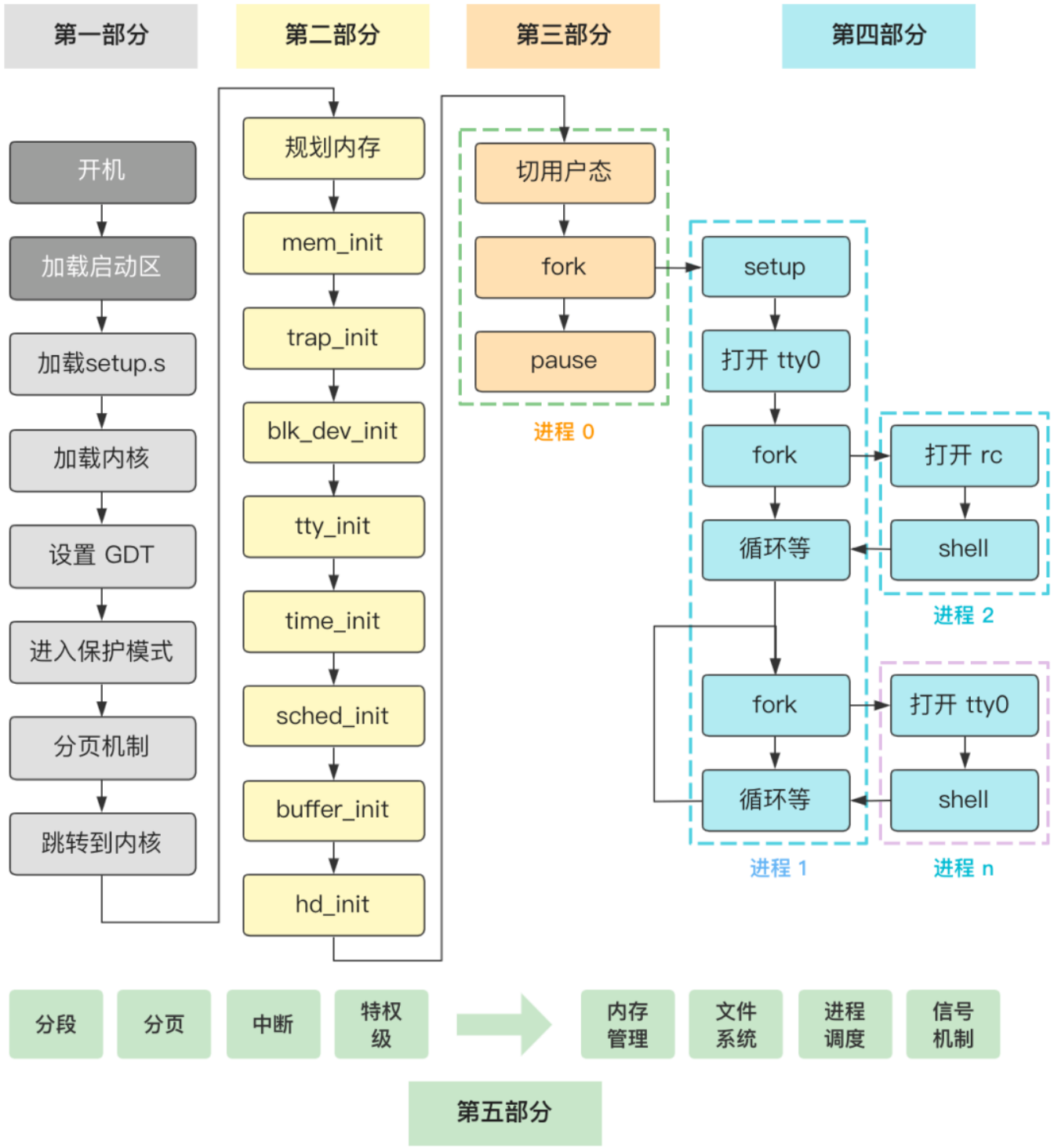
收录于合集
#操作系统源码 52 #一条shell命令的执行 8

新读者看这里，老读者直接跳过。

本系列会以一个读小说的心态，从开机启动后的代码执行顺序，带着大家阅读和赏析 Linux 0.11 全部核心代码，了解操作系统的技术细节和设计思想。

本系列的 GitHub 地址如下，希望给个 star 以示鼓励（文末**阅读原文**可直接跳转，也可以将下面的链接复制到浏览器里打开）
https://github.com/sunym1993/flash-linux0.11-talk

本回的内容属于第五部分。



你会跟着我一起，看着一个操作系统从啥都没有开始，一步一步最终实现它复杂又精巧的设计，读完这个系列后希望你能发出感叹，原来操作系统源码就是这破玩意。

以下是**已发布文章**的列表，详细了解本系列可以先从开篇词看起。

开篇词

第一部分 进入内核前的苦力活

------- 正文开始 -------

新建一个非常简单的 info.txt 文件。

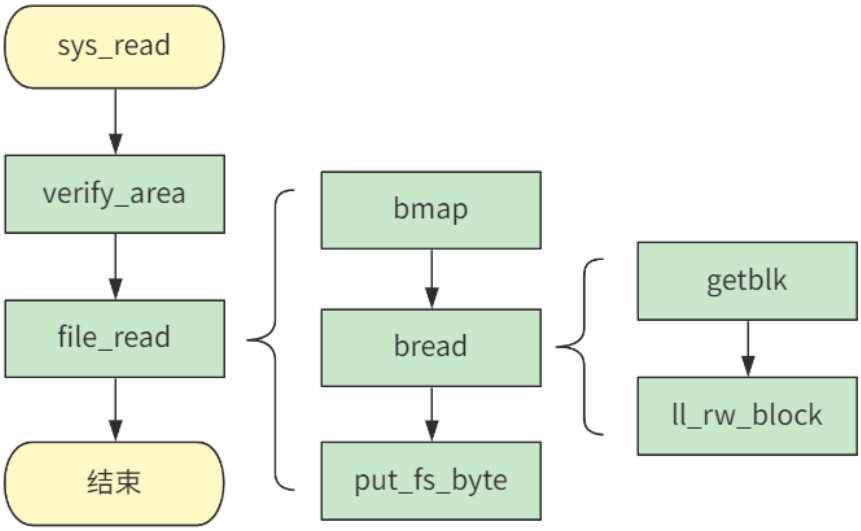
```
name:flash
age:28
language:java
```

在命令行输入一条十分简单的命令。

```
[root@linux0.11] cat info.txt | wc -l
3
```

这条命令的意思是读取刚刚的 info.txt 文件，输出它的行数。

上一回中，我们讲述了读硬盘数据的全流程。



其中 **ll_rw_block** 方法负责把硬盘中指定数据块中的数据，复制到 **getblk** 方法申请到的缓冲块里，上一回没有展开详细讲解。

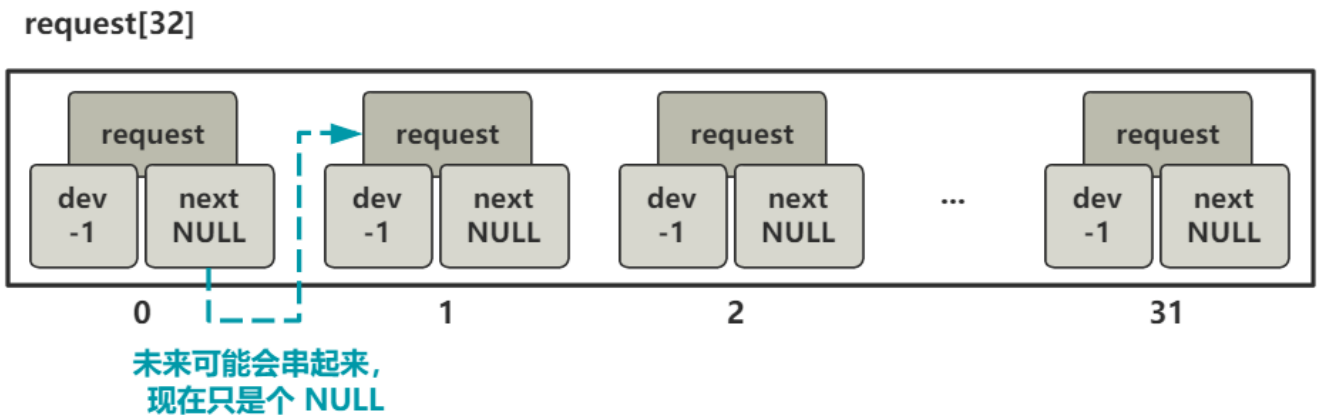
所以我们这一回，就详细讲讲，ll_rw_block 是如何完成这一任务的。

```
// buffer.c
struct buffer_head * bread(int dev,int block) {
    ...
    ll_rw_block(READ,bh);
    ...
}

void ll_rw_block (int rw, struct buffer_head *bh) {
    ...
    make_request(major, rw, bh);
}

struct request request[NR_REQUEST] = {0};
static void make_request(int major,int rw, struct buffer_head * bh) {
    struct request *req;
    ...
    // 从 request 队列找到一个空位
    if (rw == READ)
        req = request+NR_REQUEST;
    else
        req = request+((NR_REQUEST*2)/3);
    while (--req >= request)
        if (req->dev<0)
            break;
    ...
    // 构造 request 结构
    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);
}

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

```
    req->next = NULL;
    cli();
    // 清空 dirt 位
    if (req->bh)
        req->bh->b_dirt = 0;
    // 当前请求项为空，那么立即执行当前请求项
    if (!(tmp = dev->current_request)) {
        dev->current_request = req;
        sti();
        (dev->request_fn)();
        return;
    }
    // 插入到链表中
    for ( ; tmp->next ; tmp=tmp->next)
        if ((IN_ORDER(tmp,req) ||
            !IN_ORDER(tmp,tmp->next)) &&
            IN_ORDER(req,tmp->next))
            break;
    req->next=tmp->next;
    tmp->next=req;
    sti();
}
```

调用链很长，主线是从 request 数组中找到一个空位，然后作为链表项插入到 request 链表中。没错 request 是一个 32 大小的数组，里面的每一个 request 结构间通过 next 指针相连又形成链表。

如果你熟悉 第15回 | 块设备请求项初始化 blk_dev_init 所讲的内容，就会明白这个说法咯。



request 的具体结构是。

```
// blk.h
struct request {
    int dev;        /* -1 if no request */
    int cmd;        /* READ or WRITE */
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    char * buffer;
    struct task_struct * waiting;
    struct buffer_head * bh;
    struct request * next;
};
```

表示一个读盘的请求参数。

有了这些参数，底层方法拿到这个结构之后，就知道怎么样访问硬盘了。

那是谁不断从这个 request 队列中取出 request 结构并对硬盘发起读请求操作的呢？这里 Linux 0.11 有个很巧妙的设计，我们看看。

有没有注意到 add_request 方法有如下分支。

```
// blk.h
struct blk_dev_struct {
    void (*request_fn)(void);
    struct request * current_request;
};

// ll_rw_blk.c
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 */
};

static void make_request(int major,int rw, struct buffer_head * bh) {
    ...
    add_request(major+blk_dev,req);
}

static void add_request (struct blk_dev_struct *dev, struct request *req) {
    ...
    // 当前请求项为空，那么立即执行当前请求项
    if (!(tmp = dev->current_request)) {
        ...
        (dev->request_fn)();
        ...
    }
    ...
}
```

就是当设备的当前请求项为空，也就是第一次收到硬盘操作请求时，会立即执行该设备的 **request_fn** 方法，**这便是整个读盘循环的最初推手。**

当前设备的设备号是 3，也就是硬盘，会从 blk_dev 数组中取索引下标为 3 的设备结构。

在 第20回 | 硬盘初始化 hd_init 的时候，设备号为 3 的设备结构的 request_fn 被赋值为**硬盘**

请求函数 do_hd_request 了。

```
// hd.c
void hd_init(void) {
    blk_dev[3].request_fn = do_hd_request;
    ...
}
```

所以，刚刚的 request_fn 背后的具体执行函数，就是这个 do_hd_request。

```
#define CURRENT (blk_dev[MAJOR_NR].current_request)
// hd.c
void do_hd_request(void) {
    ...
    unsigned int dev = MINOR(CURRENT->dev);
    unsigned int block = CURRENT->sector;
    ...
    nsect = CURRENT->nr_sectors;
    ...
    if (CURRENT->cmd == WRITE) {
        hd_out(dev,nsect,sec,head,cyl,WIN_WRITE,&write_intr);
        ...
    } else if (CURRENT->cmd == READ) {
        hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
    } else
        panic("unknown hd-command");
}
```

我去掉了一大坨根据起始扇区号计算对应硬盘的磁头 head、柱面 cyl、扇区号 sec 等信息的代码。

可以看到最终会根据当前请求是写（WRITE）还是读（READ），在调用 **hd_out** 时传入不同的参数。

hd_out 就是读硬盘的最最最最底层的函数了。

```
// hd.c
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))
{
    ...
    do_hd = intr_addr;
    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);
}
```

可以看到，最底层的读盘请求，其实就是向一堆外设端口做读写操作。

这个函数实际上在 第17回 | 时间初始化 time_init 为了讲解与 CMOS 外设交互方式的时候讲过了，简单说硬盘的端口表是这样的。

端口	读	写
0x1F0	数据寄存器	数据寄存器
0x1F1	错误寄存器	特征寄存器
0x1F2	扇区计数寄存器	扇区计数寄存器
0x1F3	扇区号寄存器或 LBA 块地址 0~7	扇区号或 LBA 块地址 0~7
0x1F4	磁道数低 8 位或 LBA 块地址 8~15	磁道数低 8 位或 LBA 块地址 8~15
0x1F5	磁道数高 8 位或 LBA 块地址 16~23	磁道数高 8 位或 LBA 块地址 16~23
0x1F6	驱动器/磁头或 LBA 块地址 24~27	驱动器/磁头或 LBA 块地址 24~27
0x1F7	命令寄存器或状态寄存器	命令寄存器

读硬盘就是，往除了第一个以外的后面几个端口写数据，告诉要读硬盘的哪个扇区，读多少。然后再从 0x1F0 端口一个字节一个字节的读数据。这就完成了一次硬盘读操作。

当然，从 0x1F0 端口读出硬盘数据，是在硬盘读好数据并放在 0x1F0 后发起的硬盘中断，进而执行硬盘中断处理函数里进行的。

在 第20回 | 硬盘初始化 hd_init 的时候，将 hd_interrupt 设置为了硬盘中断处理函数，中断号是 0x2E，代码如下。

```
// hd.c
void hd_init(void) {
    ...
    set_intr_gate(0x2E,&hd_interrupt);
    ...
}
```

所以，在硬盘读完数据后，发起 0x2E 中断，便会进入到 hd_interrupt 方法里。

```
// system_call.s
_hd_interrupt:
    ...
    xchgl _do_hd,%edx
    ...
    call *%edx
    ...
    iret
```

这个方法主要是调用 do_hd 方法，这个方法是一个指针，就是高级语言里所谓的接口，读操作的时候，将会指向 read_intr 这个具体实现。

```
// hd.c
void do_hd_request(void) {
    ...
    } else if (CURRENT->cmd == READ) {
        hd_out(dev,nsect,sec,head,cyl,WIN_READ,&read_intr);
    }
    ...
}

static void hd_out(..., void (*intr_addr)(void)) {
    ...
    do_hd = intr_addr;
    ...
}
```

看，一切都有千丝万缕的联系，是不是很精妙。

我们展开 read_intr 方法继续看。

```
// hd.c
#define port_read(port,buf,nr) \
__asm__("cld;rep;insw"::"d" (port),"D" (buf),"c" (nr):"cx","di")

static void read_intr(void) {
    ...
    // 从数据端口读出数据到内存
    port_read(HD_DATA,CURRENT->buffer,256);
    CURRENT->errors = 0;
    CURRENT->buffer += 512;
    CURRENT->sector++;
    // 还没有读完，则直接返回等待下次
    if (--CURRENT->nr_sectors) {
        do_hd = &read_intr;
        return;
    }
    // 所有扇区都读完了
    // 删除本次都请求项
    end_request(1);
    // 再次触发硬盘操作
    do_hd_request();
}
```

这里使用了 port_read 宏定义的方法，从端口 HD_DATA 中读 256 次数据，每次读一个字，总共就是 512 字节的数据。

如果没有读完发起读盘请求时所要求的字节数，那么直接返回，等待下次硬盘触发中断并执行到 read_intr 即可。

如果已经读完了，就调用 end_request 方法将请求项清除掉，然后再次调用 do_hd_request 方法循环往复。

那重点就在于，如何结束掉本次请求的 end_request 方法。

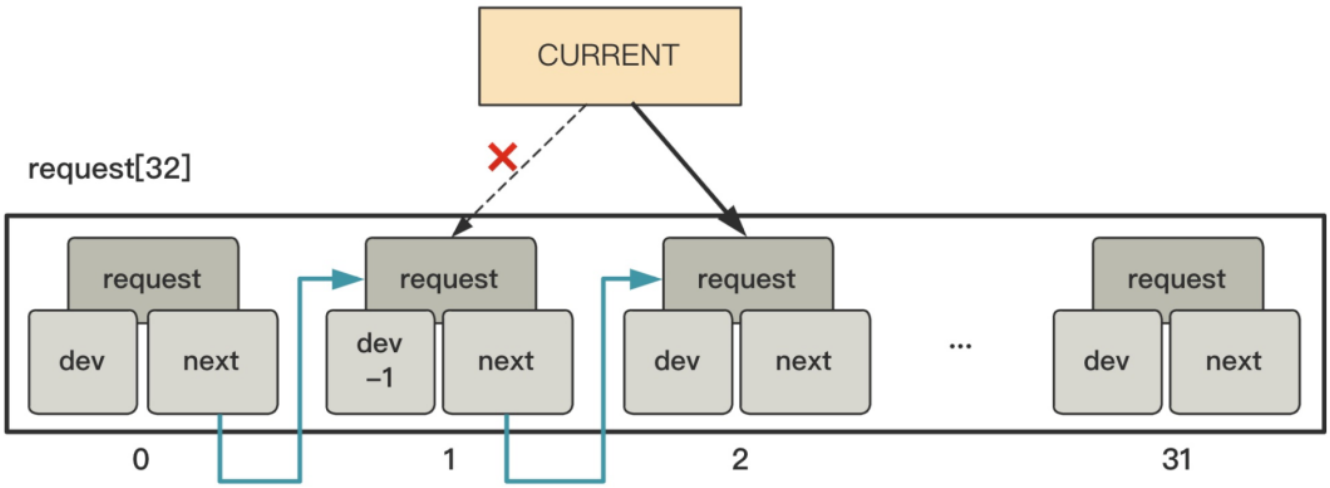
```
// blk.h
#define CURRENT (blk_dev[MAJOR_NR].current_request)

extern inline void end_request(int uptodate) {
    DEVICE_OFF(CURRENT->dev);
    if (CURRENT->bh) {
        CURRENT->bh->b_uptodate = uptodate;
        unlock_buffer(CURRENT->bh);
    }
    ...
    wake_up(&CURRENT->waiting);
    wake_up(&wait_for_request);
    CURRENT->dev = -1;
    CURRENT = CURRENT->next;
}
```

两个 **wake_up** 方法。

第一个唤醒了该请求项所对应的进程 **&CURRENT->waiting**，告诉这个进程我这个请求项的读盘操作处理完了，你继续执行吧。

另一个是唤醒了因为 request 队列满了没有将请求项插进来的进程 **&wait_for_request**。

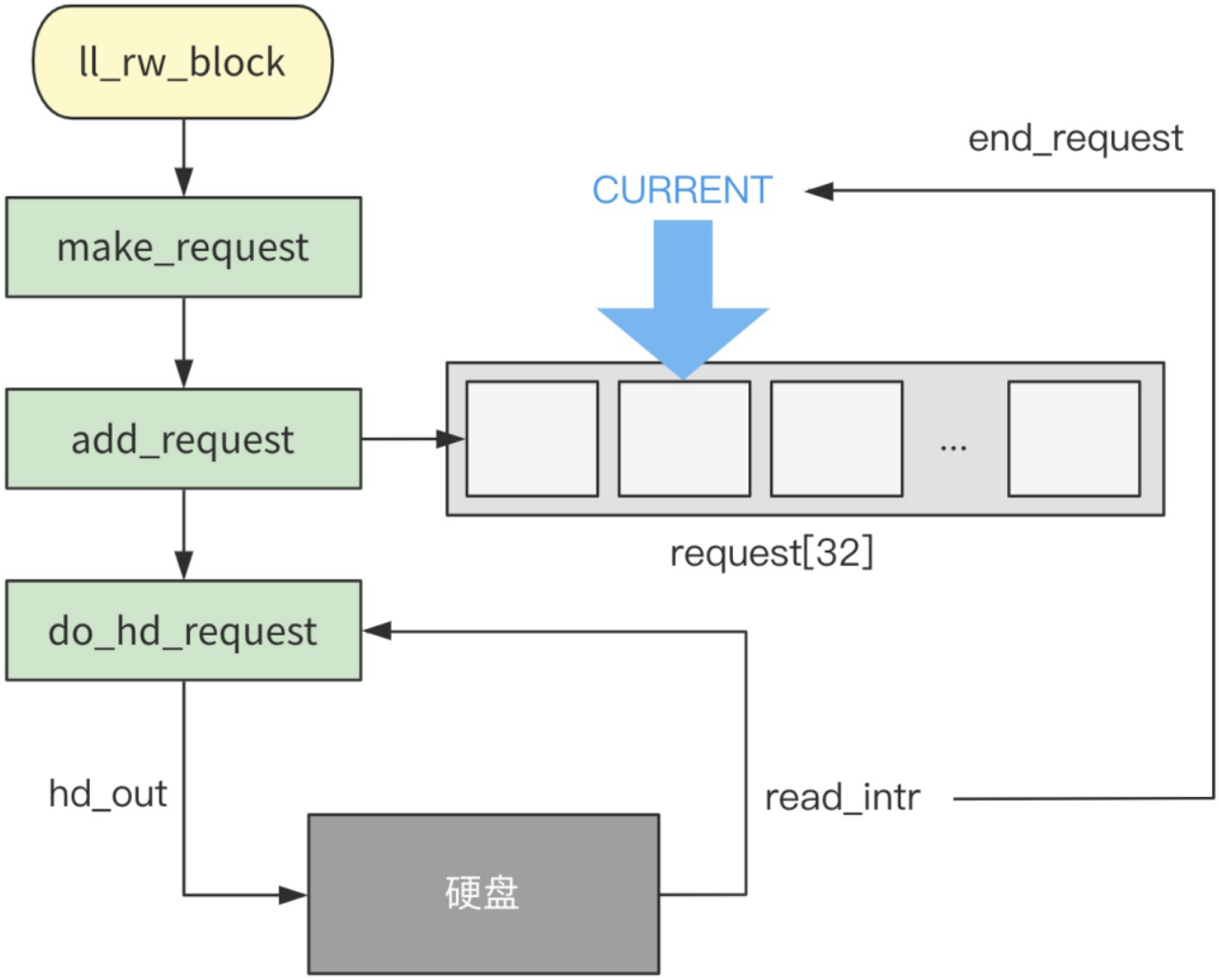
随后，将当前设备的当前请求项 CURRENT，即 request 数组里的一个请求项 request 的 dev 置空，并将当前请求项指向链表中的下一个请求项。



这样，do_hd_request 方法处理的就是下一个请求项的内容了，直到将所有请求项都处理完

毕。

整个流程就这样形成了闭环，**通过这样的机制，可以做到好似存在一个额外的进程，在不断处理 request 链表里的读写盘请求一样。**



当设备的当前请求项为空时，也就是没有在执行的块设备请求项时，ll_rw_block 就会在执行到 add_request 方法时，直接执行 do_hd_request 方法发起读盘请求。

如果已经有在执行的请求项了，就插入 request 链表中。

do_hd_request 方法执行完毕后，硬盘发起读或写请求，执行完毕后会发起硬盘中断，进而调用 read_intr 中断处理函数。

read_intr 会改变当前请求项指针指向 request 链表的下一个请求项，并再次调用

do_hd_request 方法。

所以 do_hd_request 方法一旦调用，就会不断处理 request 链表中的一项一项的硬盘请求项，这个循环就形成了，是不是很精妙！

OK，通过上一回和这一回的讲解，读盘请求的全部细节终于讲解完毕了！你还好么？

欲知后事如何，且听下回分解。

------- 关于本系列 -------

本系列的开篇词看这，开篇词

本系列的番外故事看这，让我们一起来写本书？也可以直接无脑加入星球，共同参与这场旅行。

最后，本系列**完全免费**，希望大家能多多传播给同样喜欢的人，同时给我的 GitHub 项目点个