缓冲区管理功能说明文档

版本：V0.1

最后修改时间：2023-7-10

一、缓冲区管理基本数据结构

```
typedef struct buf_head
{
    u32 count;                  // 为以后优化预留
    u32 dev, block;             // 设备号，块号(只有busy=true时才有效)
    void *buffer;               // 该缓冲块的起始地址，为cache的地址
    struct buf_head *pre_lru;   // LRU链表指针，指向LRU链表的前一个元素
    struct buf_head *nxt_lru;   // 指向下一个缓冲块头部
    struct buf_head *pre_hash; // hash表链表中的前一项
    struct buf_head *nxt_hash; // hash表链表中的后一项
    SPIN_LOCK lock;
    u8 used;                    // 该缓冲块是否被使用 0 未被使用；1 已经被使用；
    u8 dirty;                   // 该缓冲块是否是脏的 0 clean; 1 dirty
} buf_head;
```

`buf_head`结构体用于描述一个缓冲区，其中`dev`和`block`两个成员变量在物理设备上确定一块数据块，`buffer`指针指向该数据块对应的缓冲区（大小为4KB）。

系统中所有的buf_head组成一个LRU队列（双向链表），已经使用的缓冲区还会存在于一个哈希队列中。`pre_lru、nxt_lru、pre_hash、nxt_hash`是指向LRU队列、Hash队列的前一个或后一个元素的指针。

```
struct buf_lru_list
{
    struct buf_head *lru_head; // buffer head组成的双向链表的头
    struct buf_head *lru_tail; // buffer head组成的双向链表的尾
};
```

`buf_lru_list`用于记录LRU链表的头和尾元素，`lru_head`指向的是头元素(最近最久未使用的元素)，通常新分配一个`buf_head`时会从此位置处分配。`lru_tail`指向的是整个LRU链表的尾元素，通常刚刚被用过的元素会从LRU中剥离出来插入到尾部。

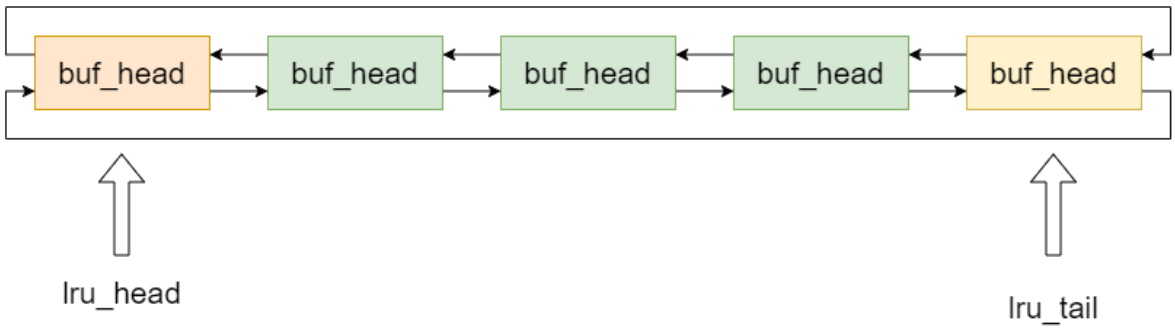
`static struct buf_head *buf_hash_table[NR_HASH]`，hash散列表;`NR_HASH = 19`,表的每一个的元素指向的是一个双向链表。

二、缓冲区的组织

有两个数据结构用于组织缓冲区：LRU链表和Hash散列表。

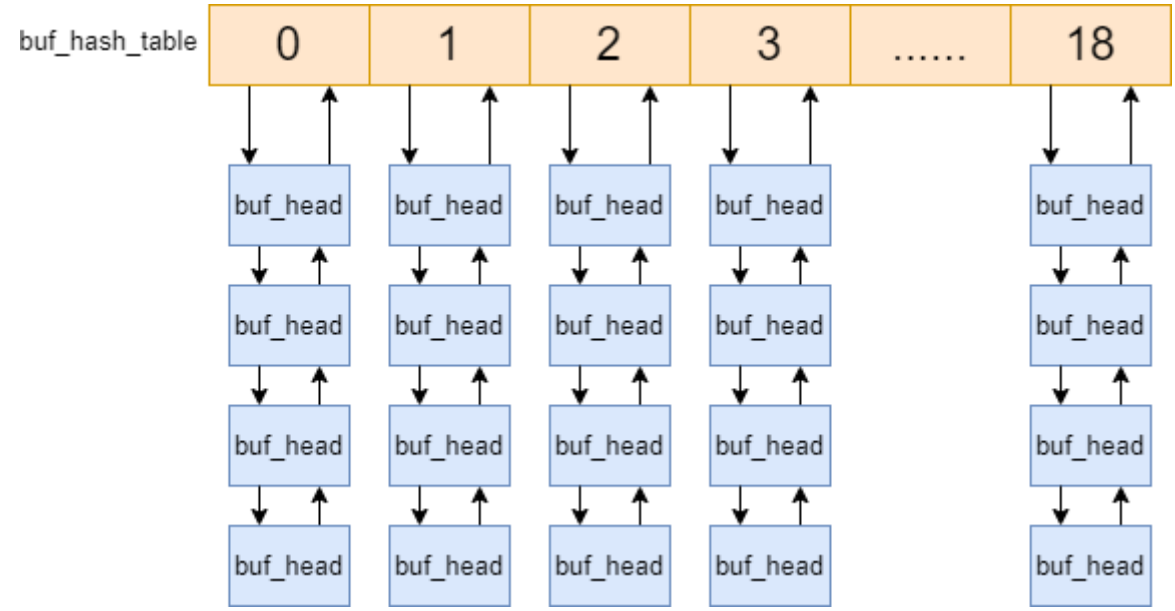
2.1 LRU链表

LRU链表是一个双向环形链表，链表由所有的buf_head组成。



所有的buf_head，不管是否被使用，都会在LRU链表中。lru_head指向的是LRU链表的头部，该buf_head是最近最久未使用的缓冲区，当进行分配新缓冲区时会将该buf_head分配出去。lru_tail指向的是LRU的尾部，表示最近被使用的缓冲区，当某一个缓冲区被使用时会将其从LRU原来的位置删除掉，加到lru_tail中。

2.2 Hash 表



`buf_hash_table`一共有19个表项，每一个表项指向的是一个buf_head组成的双向链表，这个双向链表中存储的都是已经被使用且hash code相同的`buf_head`。

hash code的计算方法为：

$$(dev \oplus block) \mod 19$$

⊕表示异或操作。

3 缓冲区管理实现的关键点

3.1初始化缓冲区

```
/*****************************************************************************
 *                                  init_buffer
 *****************************************************************************/
/**
 * 初始化缓冲区，分配好缓冲区buf head和缓冲区的内存数据块，
 * @param[in] num_block   缓冲区数据块的块数
 *
 *****************************************************************************/
void init_buffer(int num_block)
{
    // 申请buffe head和buffer block的内存，初始化lru 双向链表
    buf_head *pre = kern_kzalloc(sizeof(buf_head));
    pre->buffer = kern_kzalloc(BLOCK_SIZE);
    initlock(&(pre->lock),NULL);
    lru_list.lru_head = pre;
    initlock(&buf_lock,"buf_lock");
    //创建buf_head，为每个缓冲区分配4KB空间，构造LRU链表
    for (int i = 0; i < num_block - 1; i++)
    {
        pre->nxt_lru = kern_kzalloc(sizeof(buf_head));
        pre->nxt_lru->buffer = kern_kzalloc(BLOCK_SIZE);
        pre->nxt_lru->pre_lru = pre;
        initlock(&(pre->nxt_lru->lock),NULL);
        pre = pre->nxt_lru;
    }
    pre->nxt_lru = lru_list.lru_head;
    lru_list.lru_head->pre_lru = pre;
    lru_list.lru_tail = pre;
}
```

创建`num_block`个未使用的缓冲区和buf_head，构造出LRU链表。

3.2 获取缓冲区

`buf_head *getblk(int dev, int block)` 函数获取一个缓冲区，参数是缓冲区的设备号和块号。

```
/**********************************************************************
 *                              getblk
 **********************************************************************/
/**
 * 获取一个缓冲区块,如果dev block确定的缓冲区块已经在hash tbl中，则从hash tbl中返回buf head。
 * 若没在hash tbl中则从lru表中取出一个最近未使用的buf head（新分配一个buf head），此时会暂时将buf head从lru表中删除，
 * 若该buf head之前被用过则从hash tbl中删除掉，如果buf head的数据修改过并且还没同步到硬盘 则写入到硬盘
 * @param[in] dev  the block device id
 * @param[in] block  the block id in block device
 * @return         Ptr to buf_head.
 *
 **********************************************************************/
buf_head *getblk(int dev, int block)
{
    acquire(&buf_lock);
    buf_head *bh = find_buffer_hash(dev, block);
    // 如果在hash table中能找到，则直接返回
    if (!bh)
    {
        // 从lru 表中获取一个最近未使用的buf head
        bh = get_bh_lru();
        // 如果是脏块就立即写回硬盘
        if (bh->dirty)
        {
            sync_buff(bh);
        }
        // 如果它被用过则，从hash表中删掉它
        if (bh->used)
        {
            rm_bh_hashtbl(bh);
            bh->used = 0;
            //缓冲区重新初始化为0
            memset(bh->buffer,0,num_4K);
        }
        bh->dev = dev;
        bh->block = block;
        //放到hash 表中
        put_bh_hashtbl(bh);
    }

    bh->count+=1;
    //从lru原来位置移除
    rm_bh_lru(bh);
    //放到lru的头部
    put_bh_lru(bh);
    release(&buf_lock);
    return bh;
}
```

该函数的执行步骤如下：

1. 获取一个全局的锁
2. 去hash散列表中搜索该dev和block对应的数据块是否已经有对应的缓冲区。
3. 若hash散列表中没有对应buf_head则从lru表中获取一个最近最久未使用的`buf_head`.
4. 若在lru中新分配的这个buf_head之前已经被使用过则需要将其从hash散列表中删除，如果该`buf_head`标记为脏块，则需要将其同步回硬盘。
5. 将新分配的buf_head放到hash散列表对应的双向链表中。
6. 将新分配的或者在hash表中找到的 buf_head的count+1表示增加一次引用，将其从lru原来的位置删除，重新放入lru的头部。
7. 释放锁，返回buf_head的指针。

3.3释放缓冲区

```
void brelse(buf_head *bh)
{

    acquire(&bh->lock);
    bh->count--;
    if(bh->dirty&&bh->count == 0){
        sync_buff(bh);
    }
    release(&bh->lock);

}
```

获取该缓冲区的锁，对count变量-1，若count = 0则说明该缓冲区暂时没有被使用，若dirty位为1，说明该缓冲区还未同步回硬盘，此时应该执行将缓冲区写回硬盘操作。

4 缓冲区向文件系统提供的接口

- `buf_head *getblk(int dev, int block)`,通过getblk获取一块缓冲区，但是这块缓冲区有可能是新分配的缓冲区，若是新分配的缓冲区，则缓冲区的内全部为0。目前OrangeFS并未直接调用此接口。
- `buf_head *bread(int dev, int block)` 具体实现如下：

```
buf_head *bread(int dev, int block)
{
    buf_head *bh = getblk(dev, block);
    // 若used == 1，说明已经在hash tbl中了，buffer中也有数据了
    acquire(&bh->lock);
    if (!bh->used)
    {
        // 该buf head是一个新分配的，此时应该从硬盘读数据进来
        RD_BLOCK_SCHED(dev, block, bh->buffer);
        // 标记为已被使用
        bh->used = 1;

    }
    release(&bh->lock);
    return bh;
}
```

该函数会先获取一个缓冲区，若缓冲区是一个新分配的缓冲区（即缓冲区里并没有存放数据）则从硬盘中读取对应的数据块到该缓冲区中。该函数在OrangeFS中被广泛使用。

- `void mark_buff_dirty(buf_head *bh)`用与将缓冲区标记为*脏*。若文件系统对该缓冲区有写入则应该标记此位。

```
void mark_buff_dirty(buf_head *bh)
{
    acquire(&bh->lock);
    bh->dirty = 1;
    release(&bh->lock);
}
```

- `void brelse(buf_head *bh)`用于释放缓冲区。

```
void brelse(buf_head *bh)
{
    acquire(&bh->lock);
    bh->count--;
    if(bh->dirty&&bh->count == 0){
        sync_buff(bh);
    }
    release(&bh->lock);
}
```

所谓的释放只是将缓冲区的`count`-1。若count值为0，说明暂时没有进程在用该缓冲区此时需要同步回硬盘。

缓冲区写回硬盘的时机

- 在调用`brelse`时若该缓冲区没有被其他进程使用且被标记为`dirty`则应该写回硬盘。

- 在调用 `getblk` 时，若从lru链表中分配的缓冲区之前被用过并且被标记为 `dirty` 时需要将该缓冲区写回硬盘。

5 缓冲区与IO层的接口

IO层负责响应文件系统层或者缓冲区管理层的对硬盘读写请求，将请求转化为 `RWInfo` 结构体：

```
typedef struct rdwt_info
{
    MESSAGE *msg;//记录读写请求
    void *kbuf; //临时buffer的指针
    PROCESS *proc; //发起IO的进程
    struct rdwt_info *next;//设备IO队列的下一个
} RWInfo;
```

该结构体用于描述一个具体的IO请求。将对应的 `RWInfo` 结构体放入到 `HDQueue` ,唤醒 `hd_service` 进程依次处理队列中的IO请求,处理过程中会调用驱动层的硬盘读写接口，读取数据。

IO层与缓冲区管理层的接口

IO层提供了两个数据块读写API给缓冲区管理层分别是： `RD_BLOCK_SCHED、WR_BLOCK_SCHED`

```
//硬盘中数据块的读写
//added by sundong 2023.5.26
#define RD_BLOCK_SCHED(dev,block_nr,fsbuf) rw_blocks_sched(DEV_READ, \
                        dev,            \
                        (block_nr) * BLOCK_SIZE,    \
                        BLOCK_SIZE, /* read one block */ \
                        proc2pid(p_proc_current),/*current task id*/        \
                        fsbuf);
//added by sundong 2023.5.26
#define WR_BLOCK_SCHED(dev,block_nr,fsbuf) rw_blocks_sched(DEV_WRITE, \
                        dev,            \
                        (block_nr) * BLOCK_SIZE,    \
                        BLOCK_SIZE, /* write one block */ \
                        proc2pid(p_proc_current),               \
                        fsbuf);
```

这两个API是对 `rw_blocks_sched` 函数的进一步封装。 `rw_blocks_sched` 函数根据读写请求构造 `MESSAGE` 结构体，然后调用 `hd_rdwt_sched` ,

`hd_rdwt_sched` 构造 `RWInfo` 结构体并将其加入IO队列中，唤醒hd_service去处理IO队列中的请求。

6 IO层和驱动层的接口

驱动层负责真正的数据读写，IO层负责响应文件系统层的读写请求并将读写请求放入到一个IO队列中，唤醒hd_service进程去响应队列中IO请求。hd_service进程处理IO请求时会调用驱动层的API。

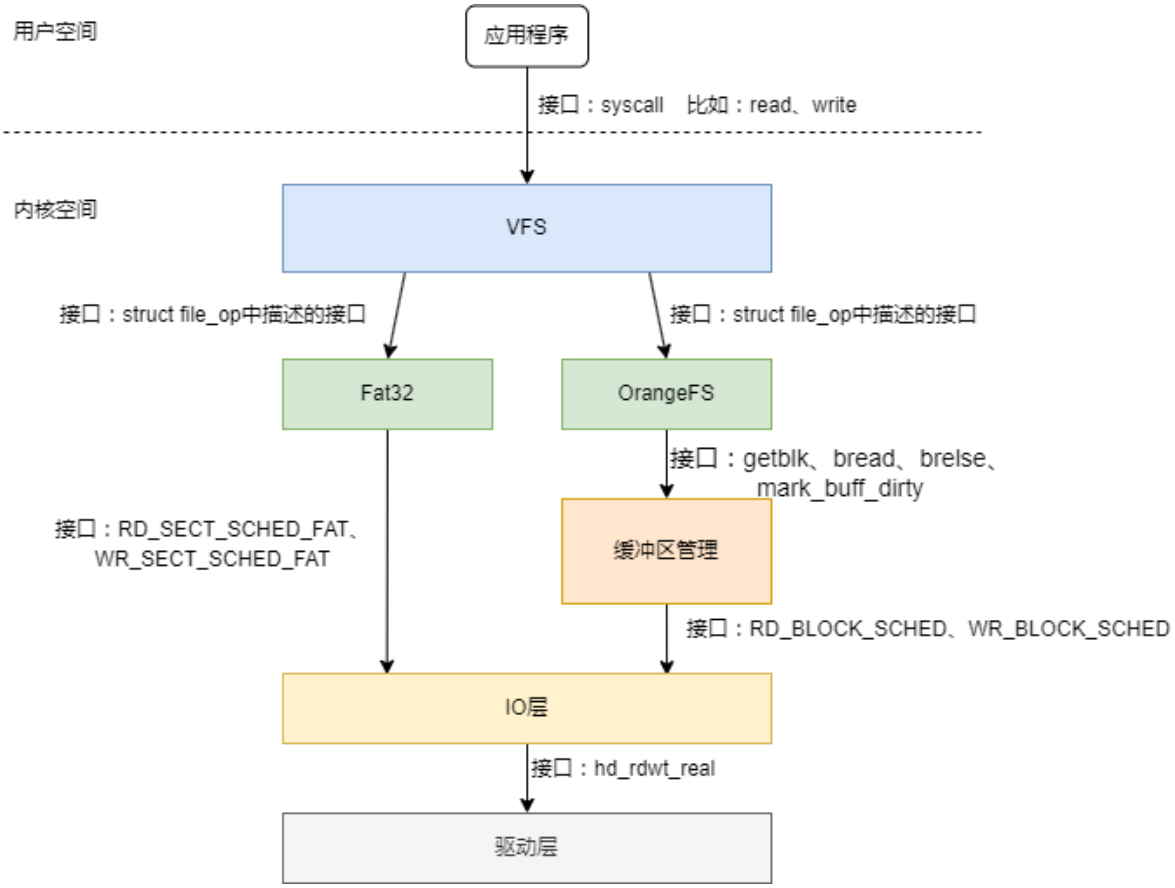
驱动层向IO层读写提供的API为 `void hd_rdwt_real(RWInfo *p)` ,具体的实现如下：

```
PRIVATE void hd_rdwt_real(RWInfo *p)
{
    int drive = DRV_OF_DEV(p->msg->DEVICE);


    u64 pos = p->msg->POSITION;

    //We only allow to R/W from a SECTOR boundary:

    // u32 sect_nr = (u32)(pos >> SECTOR_SIZE_SHIFT);   // pos / SECTOR_SIZE
    u64 sect_nr = (pos >> SECTOR_SIZE_SHIFT);
    u32 n=(p->msg->CNT + SECTOR_SIZE - 1) / SECTOR_SIZE;

    if (drive >=SATA_BASE && drive < SATA_LIMIT)//SATA read or write
    {
        SATA_rdwt(p->msg,p->kbuf);
        return 1;
    }
```

```
    // int logidx = (p->msg->DEVICE - MINOR_hd1a) % NR_SUB_PER_DRIVE;
    // sect_nr += p->msg->DEVICE < MAX_PRIM ?
    //  hd_info[drive].primary[p->msg->DEVICE].base :
    //  hd_info[drive].logical[logidx].base;

    sect_nr += hd_info[drive].part[p->msg->DEVICE & 0x0F].base;

    struct hd_cmd cmd;
    cmd.features    = 0;
    // cmd.count    = (p->msg->CNT + SECTOR_SIZE - 1) / SECTOR_SIZE;
    cmd.count = n & 0xFF;
    cmd.lba_low = sect_nr & 0xFF;
    cmd.lba_mid = (sect_nr >>  8) & 0xFF;
    cmd.lba_high    = (sect_nr >> 16) & 0xFF;

    if(hd_LBA48_sup[drive]==1){//LBA48
        cmd.count_LBA48     = (n>>8)&0xFF;
        cmd.lba_low_LBA48   = (sect_nr >> 24) & 0xFF;
        cmd.lba_mid_LBA48   = (sect_nr >> 32) & 0xFF;
        cmd.lba_high_LBA48  = (sect_nr >> 40) & 0xFF;
        cmd.device  = 0x40|((drive<<4)&0xFF);//0~3位,0：第4位0表示主盘,1表示从盘；7~5位,010,表示为LBA
        cmd.command = (p->msg->type == DEV_READ) ? ATA_READ_EXT : ATA_WRITE_EXT;
    }//by qianglong 2022.4.26
    else{//LBA28
        cmd.device  = MAKE_DEVICE_REG(1, drive, (sect_nr >> 24) & 0xF);
        cmd.command = (p->msg->type == DEV_READ) ? ATA_READ : ATA_WRITE;
    }
    hd_cmd_out(&cmd,drive);

    int bytes_left = p->msg->CNT;
    void *la = p->kbuf; //attention here!

    while (bytes_left) {
        int bytes = min(SECTOR_SIZE, bytes_left);
        if (p->msg->type == DEV_READ) {
            interrupt_wait();
            port_read(REG_DATA, hdbuf, SECTOR_SIZE);
            phys_copy(la, hdbuf, bytes);
        }
        else {
            if (!waitfor(STATUS_DRQ, STATUS_DRQ, HD_TIMEOUT))
                disp_str("hd writing error.");

            phys_copy(hdbuf, la, bytes);
            port_write(REG_DATA, hdbuf, SECTOR_SIZE);
            interrupt_wait();
        }
        bytes_left -= SECTOR_SIZE;
        la += SECTOR_SIZE;
    }
}
```

简单的说，`hd_rdwt_real`先根据`RWInfo`中描述的IO请求所对应的设备来区分是SATA设备还是IDE设备，然后执行对应设备的驱动中的读写函数。

7 v1.3.22版本中的IO层次结构及接口



注：

FAT32文件系统未适配缓冲区管理层，因此直接调用的IO层的**读写扇区函数。**