读取硬盘前的准备工作有哪些?

Original 闪客 低并发编程 2022-01-05 16:29

收录于合集
#操作系统源码
43个

读取硬盘数据到内存中，是操作系统的一个基础功能。

读取硬盘需要有块设备驱动程序，而以文件的方式来读取则还有要再上面包一层文件系统。

把读出来的数据放到内存，就涉及到内存中缓冲区的管理。

上面说的每一件事，都是一个十分庞大的体系，我们今天的文章一个都不展开讲，哈哈。

我们就讲讲，读取块设备与内存缓冲区之间的桥梁，**块设备请求项**的初始化工作。

我们以 Linux 0.11 源码为例，发现进入内核的 main 函数后不久，有这样一行代码。

```
void main(void) {
    ...
    blk_dev_init();
    ...
}
```

看到这个方法的全部代码后，你可能会会心一笑，也可能一脸懵逼。

```
void blk_dev_init(void) {
    int i;
    for (i=0; i<32; i++) {
        request[i].dev = -1;
        request[i].next = NULL;
    }
}
```

这也太简单了吧?

就是给 request 这个数组的前 32 个元素的两个变量 **dev** 和 **next** 附上值，看这俩值 **-1** 和 **NULL** 也可以大概猜出，这是没有任何作用时的初始化值。

我们看下 request 结构体。

```
/*
 * Ok, this is an expanded form so that we can use the same
 * request for paging requests when that is implemented. In
 * paging, 'bh' is NULL, and 'waiting' is used to wait for
 * read/write completion.
 */
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;
};
```

注释也附上了。

哎哟，这就有点头大了，刚刚的函数虽然很短，但看到这个结构体我们知道了，重点在这呢。

这也侧面说明了，学习操作系统，其实把遇到的重要数据结构牢记心中，就已经成功一半了。比如主内存管理结构 mem_map，知道它的数据结构是什么样子，其功能也基本就懂了。

收，继续说这个 request 结构，这个结构就代表了一次读盘请求，其中：

dev 表示设备号，-1 就表示空闲。
cmd 表示命令，其实就是 READ 还是 WRITE，也就表示本次操作是读还是写。
errors 表示操作时产生的错误次数。
sector 表示起始扇区。

nr_sectors 表示扇区数。

buffer 表示数据缓冲区，也就是读盘之后的数据放在内存中的什么位置。

waiting 是个 task_struct 结构，这可以表示一个进程，也就表示是哪个进程发起了这个请求。

bh 是缓冲区头指针，这个后面讲完缓冲区就懂了，因为这个 request 是需要与缓冲区挂钩的。

next 指向了下一个请求项。

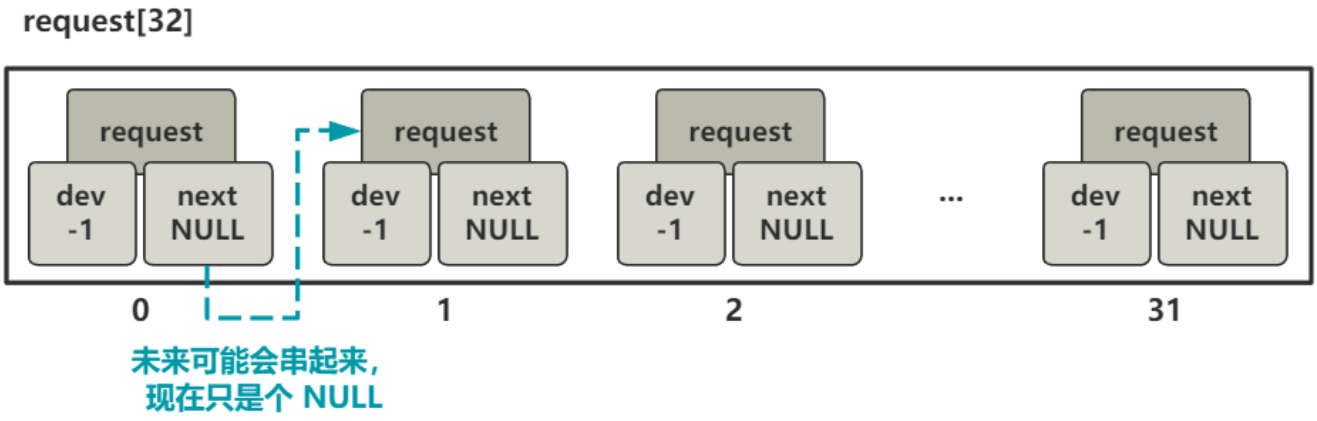
这里有的变量看不懂没关系。

不过我们倒是可以基于现有的重点参数猜测一下，比如读请求时，**cmd** 就是 READ，**sector** 和 **nr_sectors** 这俩就定位了所要读取的块设备（可以简单先理解为硬盘）的哪几个扇区，**buffer** 就定位了这些数据读完之后放在内存的什么位置。

这就够啦，想想看，这四个参数是不是就能完整描述了一个读取硬盘的需求了？而且完全没有歧义，就像下面这样。

而其他的参数，肯定是为了更好地配合操作系统进行读写块设备操作嘛，为了把多个读写块设备请求很好地组织起来。这个组织不但要有这个数据结构中 hb 和 next 等变量的配合，还要有后面的电梯调度算法的配合，仅此而已，先点到为止。

总之，我们这里就先明白，这个 request 结构可以完整描述一个读盘操作。然后那个 request 数组就是把它们都放在一起，并且它们又通过 next 指针串成链表。



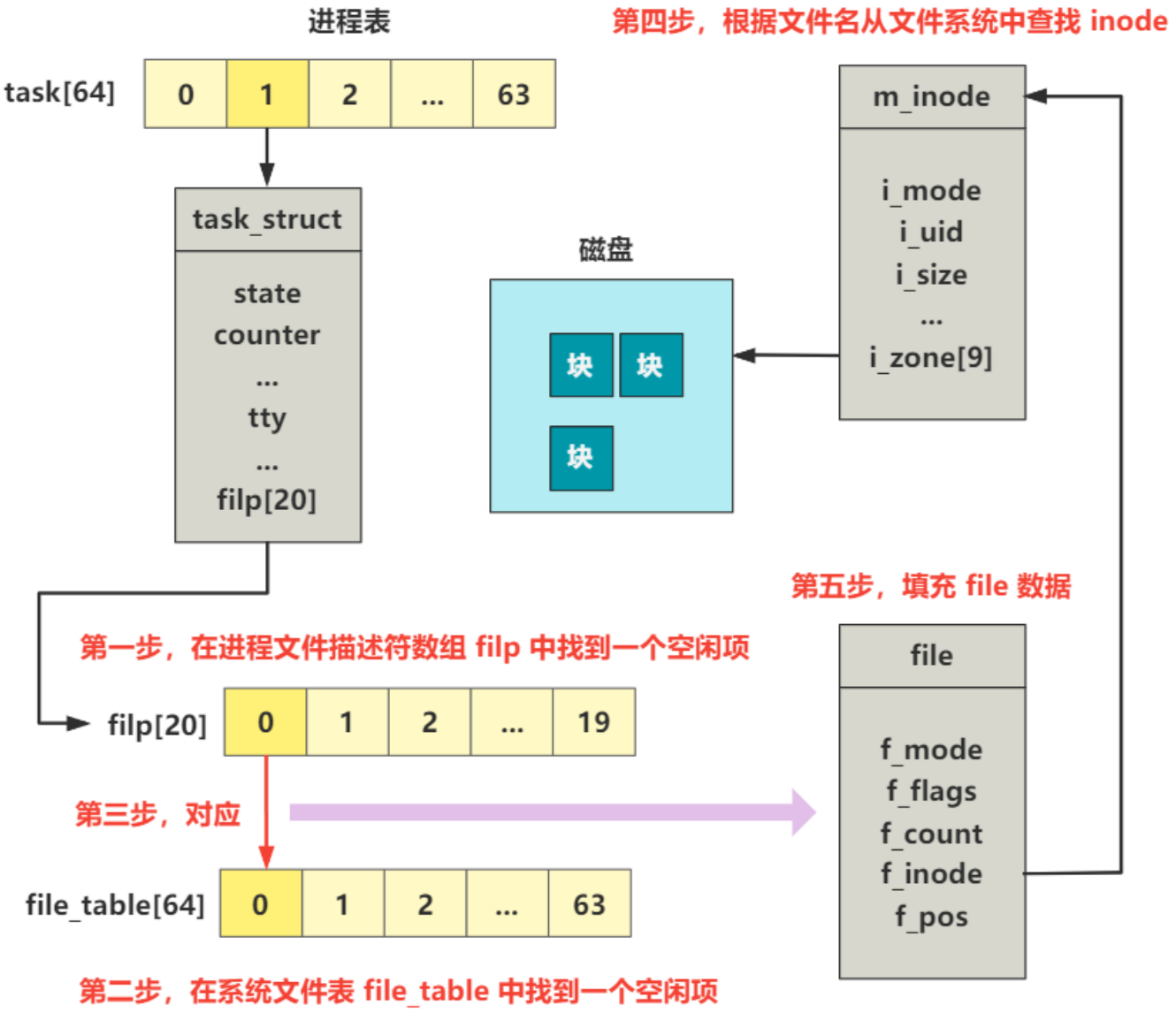
好，本文讲述的两行代码，其实就完成了上图所示的工作而已。

但讲到这就结束的话，很多同学可能会不太甘心，那我就简单展望一下，后面读盘的全流程中，是怎么用到刚刚初始化的这个 request[32] 结构的。

读操作的系统调用函数是 **sys_read**，源代码很长，我给简化一下，仅仅保留读取普通文件的分支，就是如下的样子。

```
int sys_read(unsigned int fd,char * buf,int count) {
    struct file * file = current->filp[fd];
    struct m_inode * inode = file->f_inode;
    // 校验 buf 区域的内存限制
    verify_area(buf,count);
    // 仅关注目录文件或普通文件
    return file_read(inode,file,buf,count);
}
```

看，入参 **fd** 是文件描述符，通过它可以找到一个文件的 inode，进而找到这个文件在硬盘中的位置。



另两个入参 **buf** 就是要复制到的内存中的位置，**count** 就是要复制多少个字节，很好理解。

钻到 file_read 函数里继续看。

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

整体看，就是一个 while 循环，每次读入一个块的数据，直到入参所要求的大小全部读完为止。

直接看 bread 那一行。

```
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count) {
    ...
    while (left) {
        ...
        if (!(bh=bread(inode->i_dev,nr)))
    }
}
```

这个函数就是去读某一个设备的某一个数据块号的内容，展开进去看。

```
struct buffer_head * bread(int dev,int block) {
    struct buffer_head * bh = getblk(dev,block);
    if (bh->b_uptodate)
        return bh;
    ll_rw_block(READ,bh);
    wait_on_buffer(bh);
    if (bh->b_uptodate)
        return bh;
    brelse(bh);
    return NULL;
}
```

其中 getblk 先申请了一个内存中的缓冲块，然后 ll_rw_block 负责把数据读入这个缓冲块，进去继续看。

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

static void make_request(int major,int rw, struct buffer_head * bh) {
    ...
if (rw == READ)
        req = request+NR_REQUEST;
    else
        req = request+((NR_REQUEST*2)/3);
/* find an empty request */
    while (--req >= request)
        if (req->dev<0)
            break;
    ...
/* fill up the request-info, and add it to the queue */
    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);
}
```

看，这里就用到了刚刚说的结构咯。

具体说来，就是该函数会往刚刚的设备的请求项链表 request[32] 中添加一个请求项，只要 request[32] 中有未处理的请求项存在，都会陆续地被处理，直到设备的请求项链表是空为止。

具体怎么读盘，就是与硬盘 IO 端口进行交互的过程了，可以继续往里跟，直到看到一个 hd_out 函数为止，本讲不展开了。

具体读盘操作，后面会有详细的章节展开讲解，本讲你只需要知道，我们在 main 函数的 init

系列函数中，通过 blk_dev_init 为后面的块设备访问，提前建立了一个数据结构，作为访问块设备和内存缓冲区之间的桥梁，就可以了。

本文可以当做 你管这破玩意叫操作系统源码 系列文章的第 15 回。

为了让不追更系列的读者也能很方便阅读并学到东西，我把它改造成了单独的不依赖系列上下文的文章，具体原因可以看 坚持不下去了...

点击下方的阅读原文可以跳转到本系列的 GitHub 页，那里也有完整目录和规划，以及一些辅助的资料，欢迎提出各种问题。

收录于合集 #操作系统源码 43

上一篇

你的键盘是什么时候生效的?

下一篇

第16回 | 按下键盘后为什么屏幕上就会有输出