第46回 | 读硬盘数据全流程

Original 闪客 低并发编程 2022-08-07 17:30 Posted on 北京

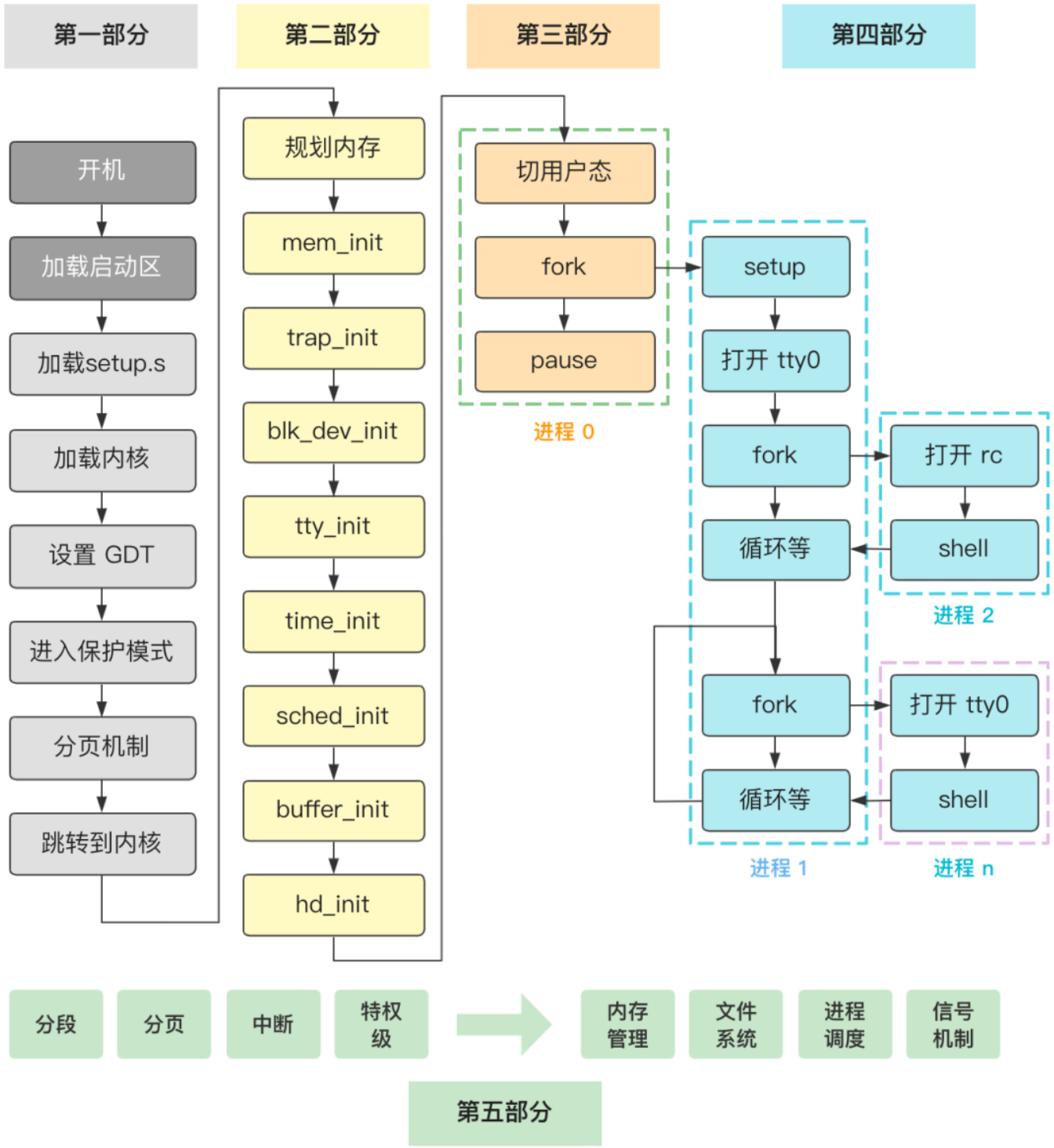
收录于合集
#操作系统源码 52 #一条shell命令的执行 8

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

第1回 | 最开始的两行代码
第2回 | 自己给自己挪个地儿
第3回 | 做好最最基础的准备工作
第4回 | 把自己在硬盘里的其他部分也放到内存来
第5回 | 进入保护模式前的最后一次折腾内存
第6回 | 先解决段寄存器的历史包袱问题
第7回 | 六行代码就进入了保护模式
第8回 | 烦死了又要重新设置一遍 idt 和 gdt
第9回 | Intel 内存管理两板斧：分段与分页
第10回 | 进入 main 函数前的最后一跃！
第一部分总结与回顾

第二部分 大战前期的初始化工作

第11回 | 整个操作系统就 20 几行代码
第12回 | 管理内存前先划分出三个边界值
第13回 | 主内存初始化 mem_init
第14回 | 中断初始化 trap_init
第15回 | 块设备请求项初始化 blk_dev_init
第16回 | 控制台初始化 tty_init
第17回 | 时间初始化 time_init
第18回 | 进程调度初始化 sched_init
第19回 | 缓冲区初始化 buffer_init
第20回 | 硬盘初始化 hd_init
第二部分总结与回顾

第三部分 一个新进程的诞生

第21回 | 新进程诞生全局概述
第22回 | 从内核态切换到用户态
第23回 | 如果让你来设计进程调度
第24回 | 从一次定时器滴答来看进程调度
第25回 | 通过 fork 看一次系统调用
第26回 | fork 中进程基本信息的复制
第27回 | 透过 fork 来看进程的内存规划
第28回 | 番外篇 - 我居然会认为权威书籍写错了...
第29回 | 番外篇 - 让我们一起来写本书？
第30回 | 番外篇 - 写时复制就这么几行代码
第三部分总结与回顾

第四部分 shell 程序的到来

第31回 | 拿到硬盘信息
第32回 | 加载根文件系统
第33回 | 打开终端设备文件
第34回 | 进程2的创建

第35回 | execve 加载并执行 shell 程序
第36回 | 缺页中断
第37回 | shell 程序跑起来了
第38回 | 操作系统启动完毕
第39回 | 番外篇 - Linux 0.11 内核调试
第40回 | 番外篇 - 为什么你怎么看也看不懂
第四部分总结与回顾

第五部分 一条 shell 命令的执行

第41回 | 番外篇 - 跳票是不可能的
第42回 | 用键盘输入一条命令
第43回 | shell 程序读取你的命令
第44回 | 进程的阻塞与唤醒
第45回 | 解析并执行 shell 命令
第46回 | 读硬盘数据全流程（本文）

------- 正文开始 -------

新建一个非常简单的 info.txt 文件。

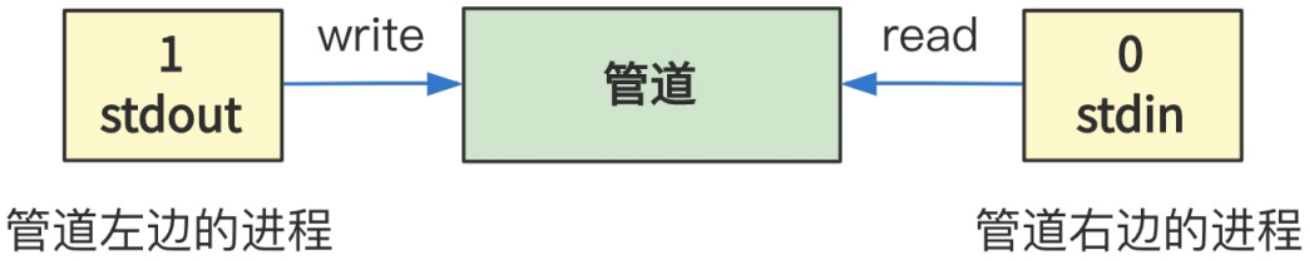
```
name:flash
age:28
language:java
```

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，我们解释了 shell 程序是如何解释并执行我们输入的命令的，并展开讲解了管道类型命令的原理。



同时也说了，在 第35回 | execve 加载并执行 shell 程序 和 第36回 | 缺页中断，我们已经讲过如何通过 execve 加载并执行 shell 程序，但略过了将数据从硬盘加载到内存的逻辑细节。

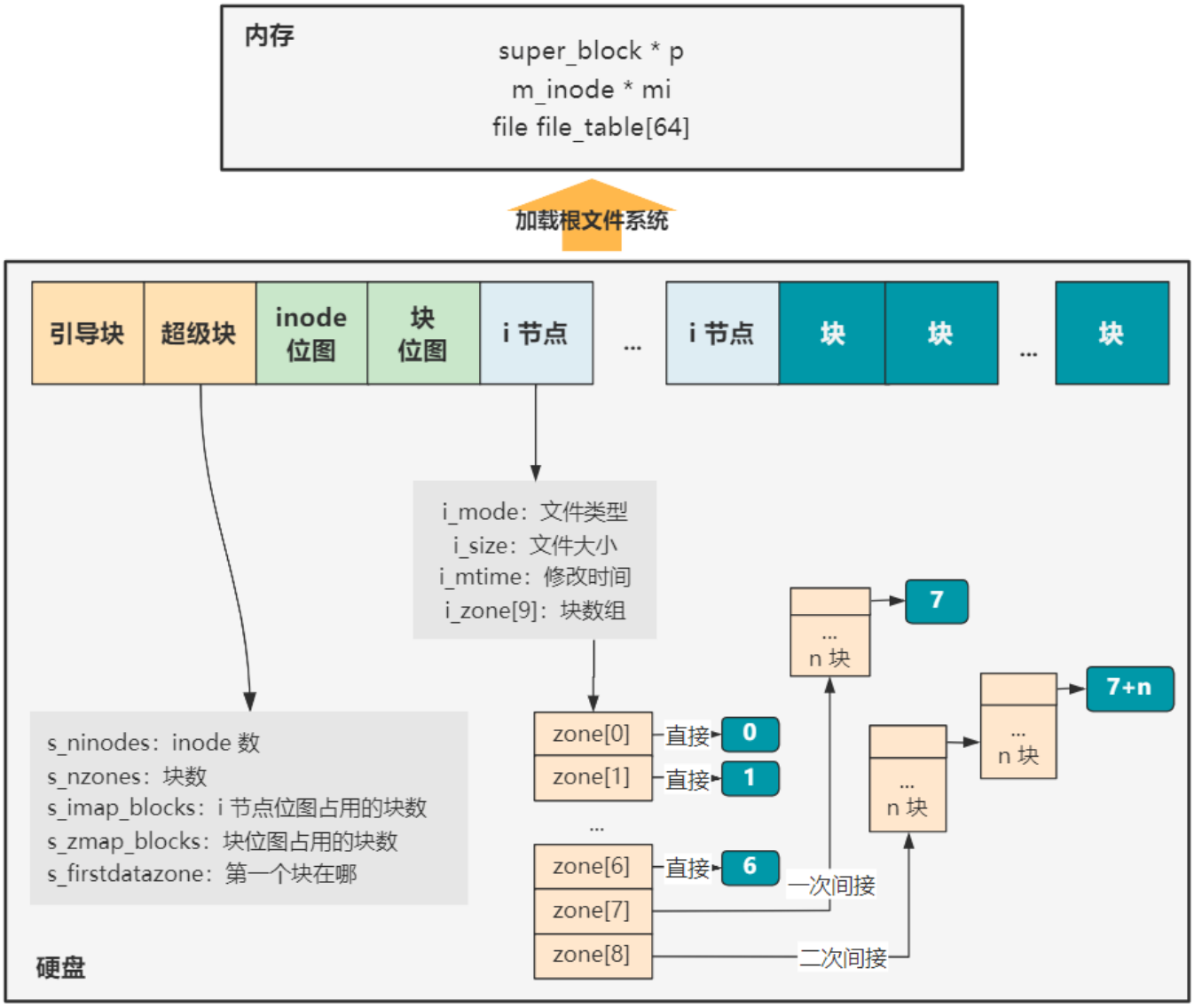
那我们这一讲就把它扒开来看看。

将硬盘中的数据读入内存，听起来是个很简单的事情，但操作系统要考虑的问题很多。

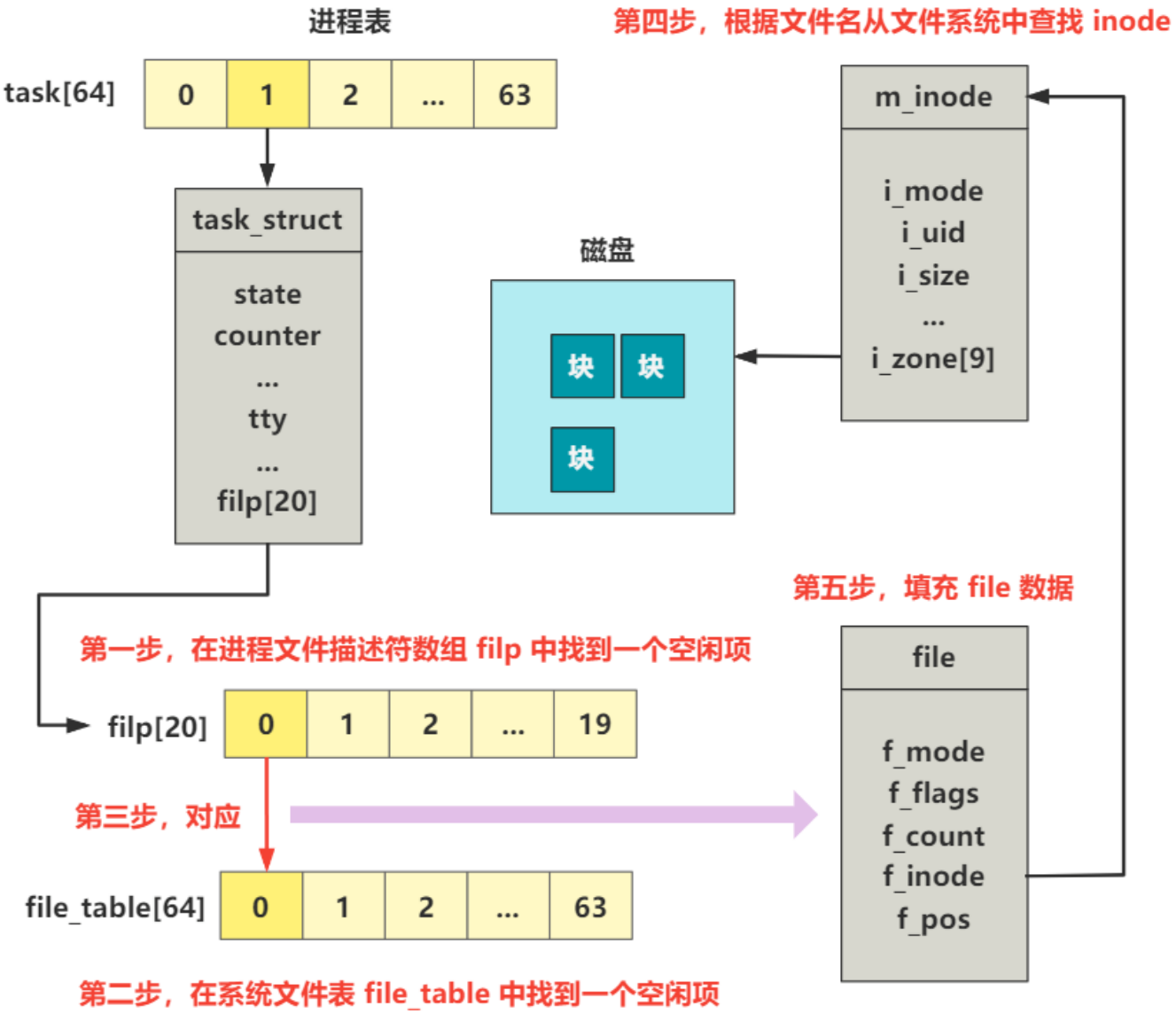
如果让你来设计这个函数

我们先别急，一点点来，想想看，如果让你设计这个函数，你会怎么设计呢？

首先我们知道，通过 第32回 | 加载根文件系统 中文件系统的建设。



以及 第33回 | 打开终端设备文件 讲解的打开一个文件的操作。



我们已经可以很方便地通过一个**文件描述符 fd**，寻找到存储在硬盘中的一个文件了，再具体点就是知道这个文件在硬盘中的哪几个扇区中。

所以，设计这个函数第一个要指定的参数就可以是 fd 了，它仅仅是个数字。当然，之所以能这样方便，就要感谢刚刚说的**文件系统建设**以及**打开文件的逻辑**这两项工作。

之后，我们得告诉这个函数，把这个 fd 指向的硬盘中的文件，复制到内存中的**哪个位置**，**复制多大**。

那更简单了，内存中的位置，我们用一个表示地址值的参数 **buf**，复制多大，我们用 **count** 来表示，单位是字节。

那这个函数就可以设计为。

```
int sys_read(unsigned int fd,char * buf,int count) {
    ...
}
```

是不是合情合理，无法反驳。

鸟瞰操作系统的读操作函数

实际上，你刚刚设计出来的读操作函数，这正是 Linux 0.11 读操作的系统调用入口函数，在 read_write.c 这个文件里。

```
// read_write.c
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;
    if (!count)
        return 0;
    verify_area(buf,count);
    inode = file->f_inode;
    if (inode->i_pipe)
        return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
    if (S_ISCHR(inode->i_mode))
        return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
    if (S_ISBLK(inode->i_mode))
        return block_read(inode->i_zone[0],&file->f_pos,buf,count);
    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;
}
```

那我们就分析这个函数就好了。

不过首先我先简化一下，去掉一些错误校验逻辑等旁路分支，并添加上注释。

```
// read_write.c
int sys_read(unsigned int fd,char * buf,int count) {
    struct file * file = current->filp[fd];
    // 校验 buf 区域的内存限制
    verify_area(buf,count);
    struct m_inode * inode = file->f_inode;
    // 管道文件
    if (inode->i_pipe)
        return (file->f_mode&1)?read_pipe(inode,buf,count):-EIO;
    // 字符设备文件
    if (S_ISCHR(inode->i_mode))
        return rw_char(READ,inode->i_zone[0],buf,count,&file->f_pos);
    // 块设备文件
    if (S_ISBLK(inode->i_mode))
        return block_read(inode->i_zone[0],&file->f_pos,buf,count);
    // 目录文件或普通文件
    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);
    }
    // 不是以上几种，就报错
    printk("(Read)inode->i_mode=%06o\n\r",inode->i_mode);
    return -EINVAL;
}
```

这样，整个的逻辑就非常清晰了。

由此也可以注意到，操作系统源码的设计比我刚刚说的更通用，我刚刚只让你设计了读取硬盘的函数，但其实在 Linux 下一切皆文件，所以这个函数将**管道文件、字符设备文件、块设备文件、目录文件、普通文件**分别指向了不同的具体实现。

那我们今天仅仅关注最常用的，读取目录文件或普通文件，并且不考虑读取的字节数大于文件本身大小这种不合理情况。

再简化下代码。

```
// read_write.c
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);
}
```

太棒了！没剩多少了，一个个击破！

第一步，根据文件描述符 fd，在进程表里拿到了 **file** 信息，进而拿到了 **inode** 信息。第二步，对 buf 区域的内存做校验。第三步，调用具体的 file_read 函数进行读操作。

就这三步，很简单吧～

在进程表 filp 中拿到 file 信息进而拿到 inode 信息这一步就不用多说了，这是在打开一个文件时，或者像管道文件一样创建出一个管道文件时，就封装好了 file 以及它的 inode 信息。

我们看接下来的两步。

对 buf 区域的内存做校验 verify_area

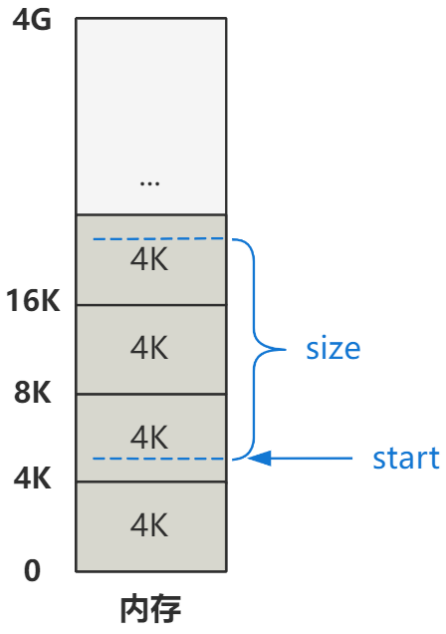
对 buf 区域的内存做校验的部分，说是校验，里面还挺有说道呢。

```c
// fork.c
void verify_area(void * addr,int size) {
    unsigned long start;
    start = (unsigned long) addr;
    size += start & 0xfff;
    start &= 0xfffff000;
    start += get_base(current->ldt[2]);
    while (size>0) {
        size -= 4096;
        write_verify(start);
        start += 4096;
    }
}
```

addr 就是刚刚的 buf，size 就是刚刚的 count。然后这里又将 addr 赋值给了 start 变量。所以代码开始，**start 就表示要复制到的内存的起始地址，size 就是要复制的字节数。**

这段代码很简单，但如果不了解内存的分段和分页机制，将会难以理解。

Linux 0.11 对内存是以 **4K** 为一页单位来划分内存的，所以内存看起来就是一个个 4K 的小格子。

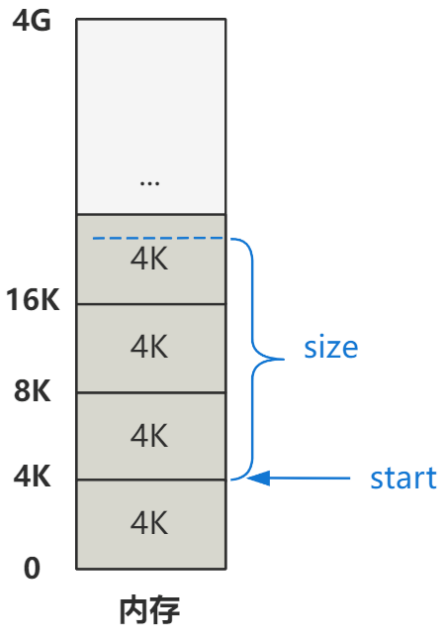


你看，我们假设要复制到的内存的起始地址 start 和要复制的字节数 size 在图中的那个位置。

那么开始的两行计算代码。

```
// fork.c
void verify_area(void * addr,int size) {
    ...
    size += start & 0xfff;
    start &= 0xfffff000;
    ...
}
```

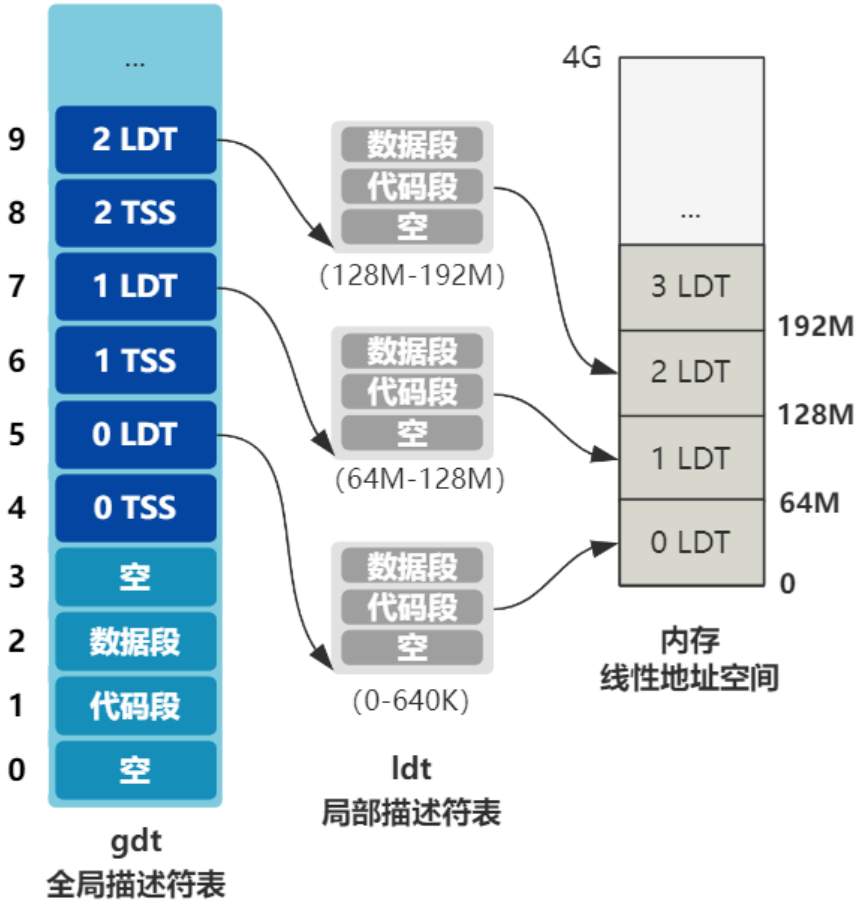
就是将 start 和 size 按页对齐一下。



然后，又由于每个进程有不同的数据段基址，所以还要加上它。

```
// fork.c
void verify_area(void * addr,int size) {
    ...
    start += get_base(current->ldt[2]);
    ...
}
```

具体说来就是加上当前进程的局部描述符表 LDT 中的数据段的段基址。



每个进程的 LDT 表，由 Linux 创建进程时的代码给规划好了。具体说来，就是如上图所示，每个进程的线性地址范围，是

(进程号)*64M ~ (进程号+1)*64M

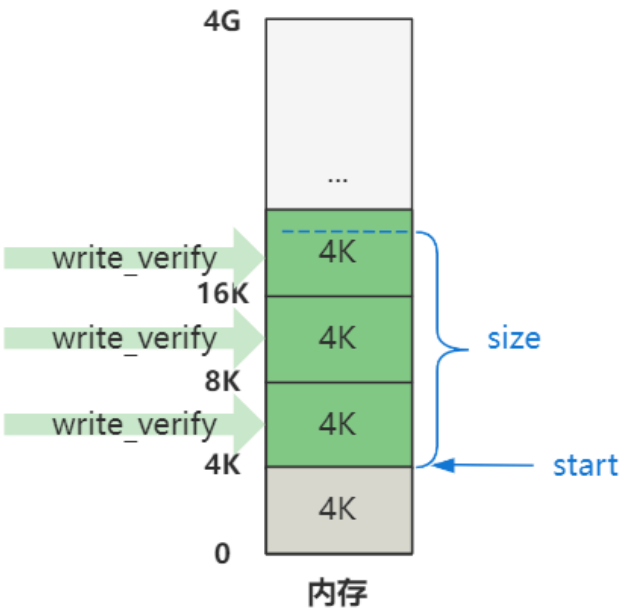
而对于进程本身来说，都以为自己是从零号地址开始往后的 64M，所以传入的 start 值也是以零号地址为起始地址算出来的。

但现在经过系统调用进入 sys_write 后会切换为内核态，内核态访问数据会通过**基地址为 0 的全局描述符表中的数据段**来访问数据。所以，start 要加上它自己进程的数据段基址，才对。

再之后，就是对这些页进行具体的验证操作。

```
// fork.c
void verify_area(void * addr,int size) {
    ...
    while (size>0) {
        size -= 4096;
        write_verify(start);
        start += 4096;
    }
}
```

也就是这些页。



这些 write_verify 将会对这些页进行**写页面验证**，如果页面存在但不可写，则执行 un_wp_page 复制页面。

```
// memory.c
void write_verify(unsigned long address) {
    unsigned long page;
    if (!( (page = *((unsigned long *) ((address>>20) & 0xffc)) )&1))
        return;
    page &= 0xfffff000;
    page += ((address>>10) & 0xffc);
    if ((3 & *(unsigned long *) page) == 1)  /* non-writeable, present */
        un_wp_page((unsigned long *) page);
    return;
}
```

看，那个 un_wp_page 意思就是取消页面的写保护，就是写时复制的原理，在 第30回 | 番外篇 - 写时复制就这么几行代码 已经讨论过了，这里就不做展开了。

执行读操作 file_read

下面终于开始进入读操作的正题了，页校验完之后，就可以真正调用 file_read 函数了。

```
// read_write.c
int sys_read(unsigned int fd,char * buf,int count) {
    ...
    return file_read(inode,file,buf,count);
}

// file_dev.c
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。

while 去掉，简化起来就是这样。

```
// file_dev.c
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count) {
    ...
    int nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE);
    struct buffer_head *bh=bread(inode->i_dev,nr);
    ...
    char * p = nr + bh->b_data;
    while (chars-->0)
         put_fs_byte(*(p++),buf++);
    ...
}
```

首先 bmap 获取全局数据块号，然后 bread 将数据块的数据复制到缓冲区，然后 put_fs_byte 再一个字节一个字节地将缓冲区数据复制到用户指定的内存中。

我们一个个看。

bmap：获取全局的数据块号

先看第一个函数调用，bmap。

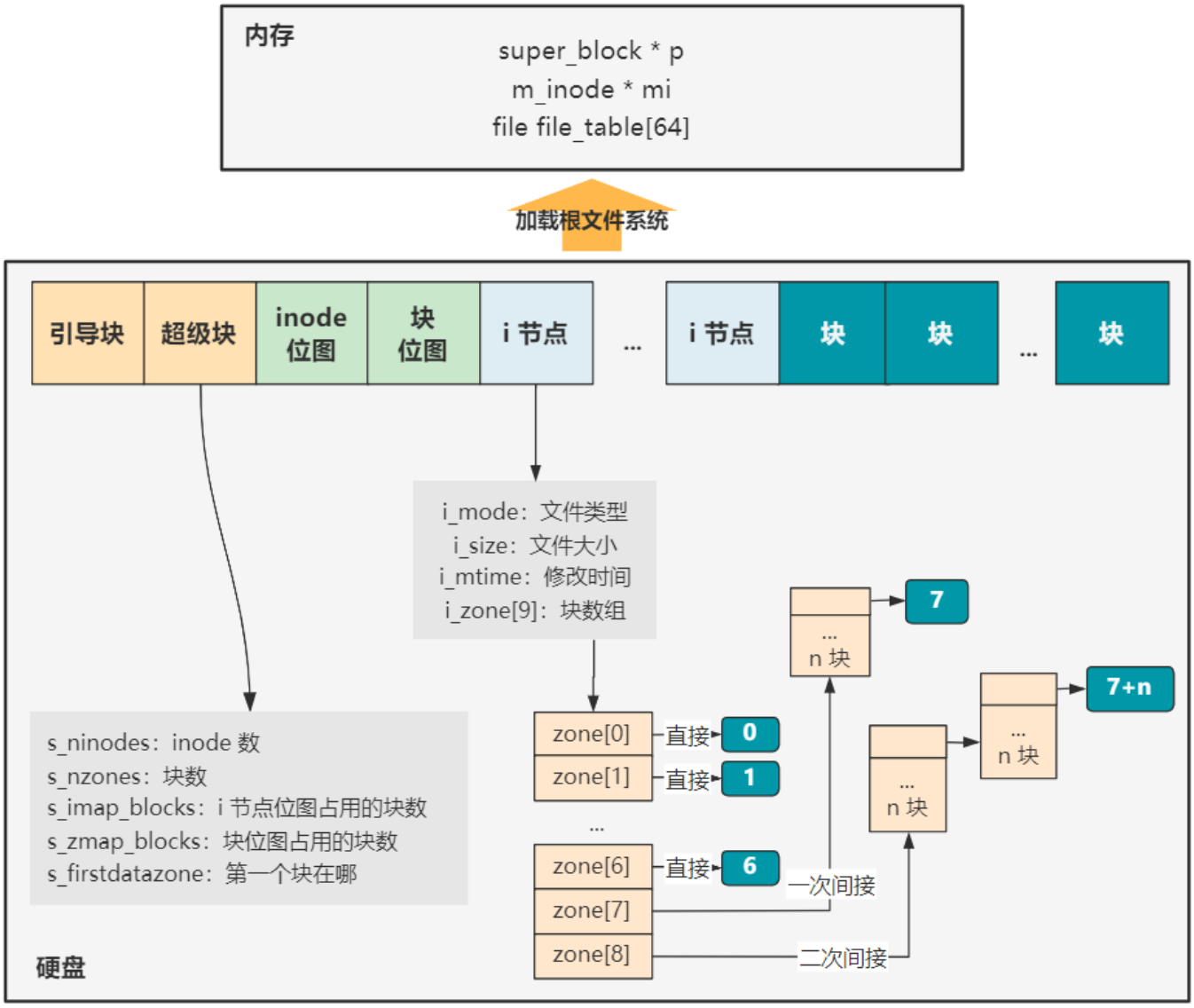
```
// file_dev.c
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count) {
    ...
    int nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE);
    ...}

// inode.c
int bmap(struct m_inode * inode,int block) {
    return _bmap(inode,block,0);
}

static int _bmap(struct m_inode * inode,int block,int create) {
    ...
    if (block<0)
        ...
    if (block >= 7+512+512*512)
        ...
    if (block<7)
        // zone[0] 到 zone[7] 采用直接索引，可以索引小于 7 的块号
        ...
    if (block<512)
        // zone[7] 是一次间接索引，可以索引小于 512 的块号
        ...
    // zone[8] 是二次间接索引，可以索引大于 512 的块号
}
```

我们看到整个条件判断的结构是根据 block 来划分的。

block 就是要读取的块号，之所以要划分，就是因为 inode 在记录文件所在块号时，采用了多级索引的方式。



zone[0] 到 zone[7] 采用直接索引，zone[7] 是一次间接索引，zone[8] 是二次间接索引。

那我们刚开始读，块号肯定从零开始，所以我们就先看 block<7，通过直接索引这种最简单的方式读的代码。

```
// inode.c
static int _bmap(struct m_inode * inode,int block,int create) {
    ...
    if (block<7) {
        if (create && !inode->i_zone[block])
            if (inode->i_zone[block]=new_block(inode->i_dev)) {
                inode->i_ctime=CURRENT_TIME;
                inode->i_dirt=1;
            }
        return inode->i_zone[block];
    }
    ...
}
```

由于 create = 0，也就是并不需要创建一个新的数据块，所以里面的 if 分支也没了。

```
// inode.c
static int _bmap(struct m_inode * inode,int block,int create) {
    ...
    if (block<7) {
        ...
        return inode->i_zone[block];
    }
    ...
}
```

可以看到，其实 bmap 返回的，就是要读入的块号，**从全局看在块设备的哪个逻辑块号下。**

也就是说，假如我想要读这个文件的第一个块号的数据，该函数返回的事你这个文件的第一个块在整个硬盘中的哪个块中。

bread：将 bmap 获取的数据块号读入到高速缓冲块

好了，拿到这个数据块号后，回到 file_read 函数接着看。

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

nr 就是具体的数据块号，作为其中其中一个参数，传入下一个函数 bread。

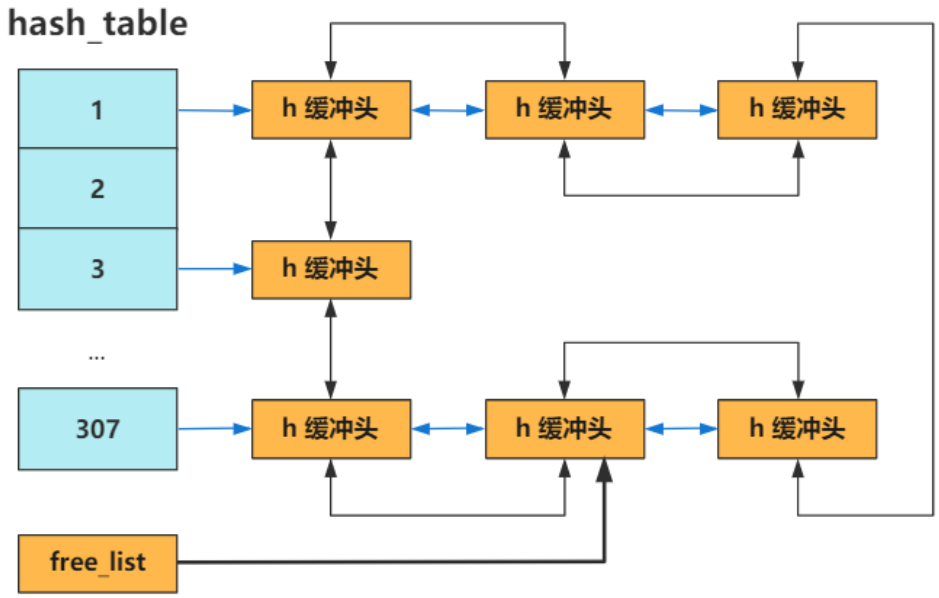
bread 这个方法的入参除了数据块号 block（就是刚刚传入的 nr）外，还有 inode 结构中的 i_dev，表示设备号。

```
// buffer.c
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;
}
```

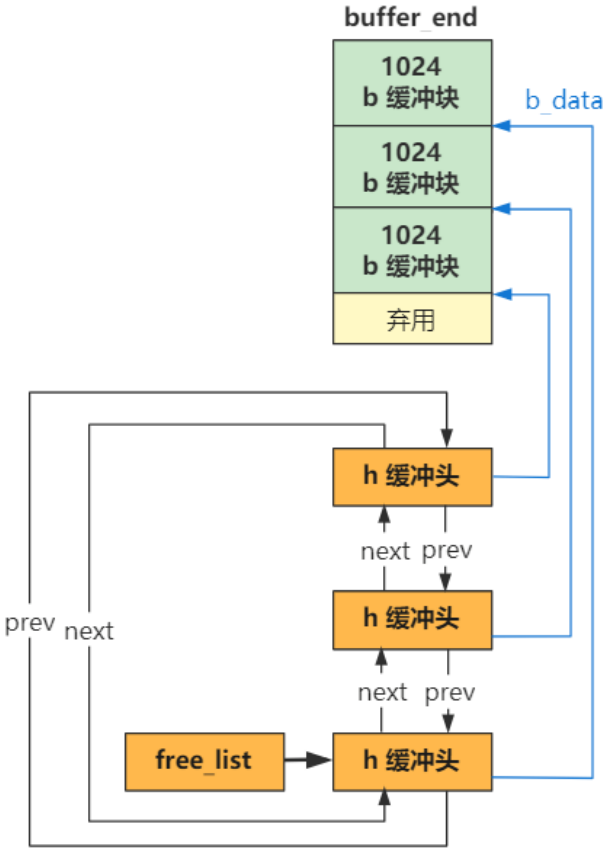
这个 bread 方法就是根据一个设备号 dev 和一个数据块号 block，将这个数据块的数据，从硬盘复制到缓冲区里。

关于缓冲区，已经在 第19回 | 缓冲区初始化 buffer_init 说明过了，有些久远。而 getblk 方法，**就是根据设备号 dev 和数据块号 block，申请到一个缓冲块。**

简单说就是，先根据 hash 结构快速查找这个 dev 和 block 是否有对应存在的缓冲块。



如果没有，那就从之前建立好的双向链表结构的头指针 free_list 开始寻找，直到找到一个可用的缓冲块。



具体代码逻辑，还包含当缓冲块正在被其他进程使用，或者缓冲块对应的数据已经被修改时的

处理逻辑，你可以看一看，关键流程我已加上了注释。

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

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

repeat:
    // 先从 hash 结构中找
    if (bh = get_hash_table(dev,block))
        return bh;

    // 如果没有就从 free_list 开始找遍双向链表
    tmp = free_list;
    do {
        if (tmp->b_count)
            continue;
        if (!bh || BADNESS(tmp)<BADNESS(bh)) {
            bh = tmp;
            if (!BADNESS(tmp))
                break;
        }
    } while ((tmp = tmp->b_next_free) != free_list);

    // 如果还没找到，那就说明没有缓冲块可用了，就先阻塞住等一会
    if (!bh) {
        sleep_on(&buffer_wait);
        goto repeat;
    }

    // 到这里已经说明申请到了缓冲块，但有可能被其他进程上锁了
    // 如果上锁了的话，就先等等
    wait_on_buffer(bh);
    if (bh->b_count)
        goto repeat;

    // 到这里说明缓冲块已经申请到，且没有上锁
    // 但还得看 dirt 位，也就是有没有被修改
```

```
        // 如果被修改了，就先重新从硬盘中读入新数据
        while (bh->b_dirt) {
            sync_dev(bh->b_dev);
            wait_on_buffer(bh);
            if (bh->b_count)
                goto repeat;
        }
        if (find_buffer(dev,block))
            goto repeat;

        // 给刚刚获取到的缓冲头 bh 重新赋值
        // 并调整在双向链表和 hash 表中的位置
        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;
    }
```

总之，经过 getblk 之后，我们就在内存中，**找到了一处缓冲块**，用来接下来存储硬盘中指定数据块的数据。

那接下来的一步，自然就是把硬盘中的数据复制到这里啦，没错，**ll_rw_block** 就是干这个事的。

这个方法的细节特别复杂，也是我看了好久才看明白的地方，我会在下一回把这个方法详细地展开讲解。

在这一回里，你就当它已经成功地把硬盘中的一个数据块的数据，一个字节都不差地复制到了我们刚刚申请好的缓冲区里。

接下来，就要通过 **put_fs_byte** 方法，一个字节一个字节地，将缓冲区里的数据，复制到用户指定的内存 buf 中去了，当然，只会复制 count 字节。

```
// file_dev.c
int file_read(struct m_inode * inode, struct file * filp, char * buf, int count) {
    ...
    int  nr = bmap(inode,(filp->f_pos)/BLOCK_SIZE);
    struct buffer_head *bh=bread(inode->i_dev,nr);
    ...
    char * p = nr + bh->b_data;
    while (chars-->0)
         put_fs_byte(*(p++),buf++);
    ...
}
```

put_fs_byte：将 bread 读入的缓冲块数据复制到用户指定的内存中

这个过程，仅仅是内存之间的复制，所以不必紧张。

```
// segment.h
extern _inline void
put_fs_byte (char val, char *addr) {
    __asm__ ("movb %0,%%fs:%1"::"r" (val),"m" (*addr));
}
```

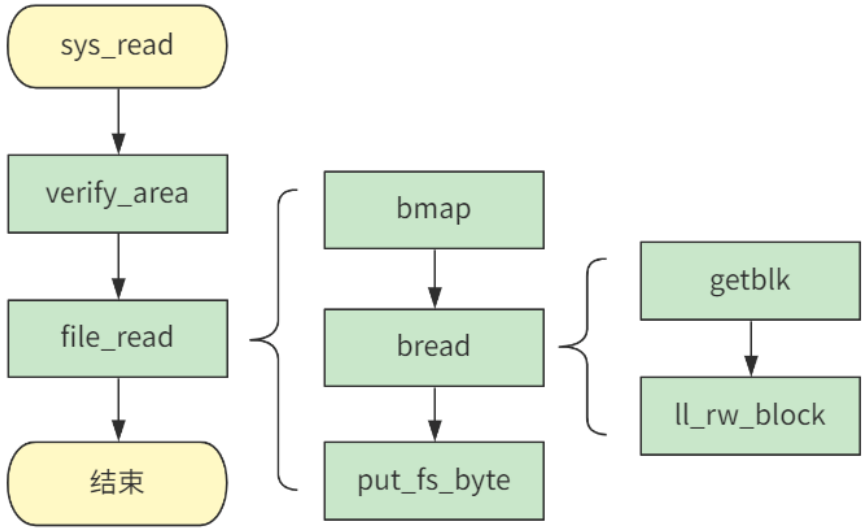
有点难以理解，我改成较为好看的样子。（参考赵炯《Linux 内核完全注释 V1.9.5》）

```
// segment.h
extern _inline void
put_fs_byte (char val, char *addr) {
    _asm mov ebx,addr
    _asm mov al,val;
    _asm mov byte ptr fs:[ebx],al;
}
```

其实就是三个汇编指令的 mov 操作。

至此，我们就将数据从硬盘读入缓冲区，再从缓冲区读入用户内存，一个 read 函数完美谢

幕！



首先通过 verify_area 对内存做了校验，需要写时复制的地方在这里提前进行好了。

接下来，file_read 方法做了读盘的全部操作，通过 bmap 获取到了硬盘全局维度的数据块号，然后 bread 将数据块数据复制到缓冲区，然后 put_fs_byte 再将缓冲区数据复制到用户内存。

今天内容较多，好好消化一下，欲知后事如何如何，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。