MINIX File System 探悉

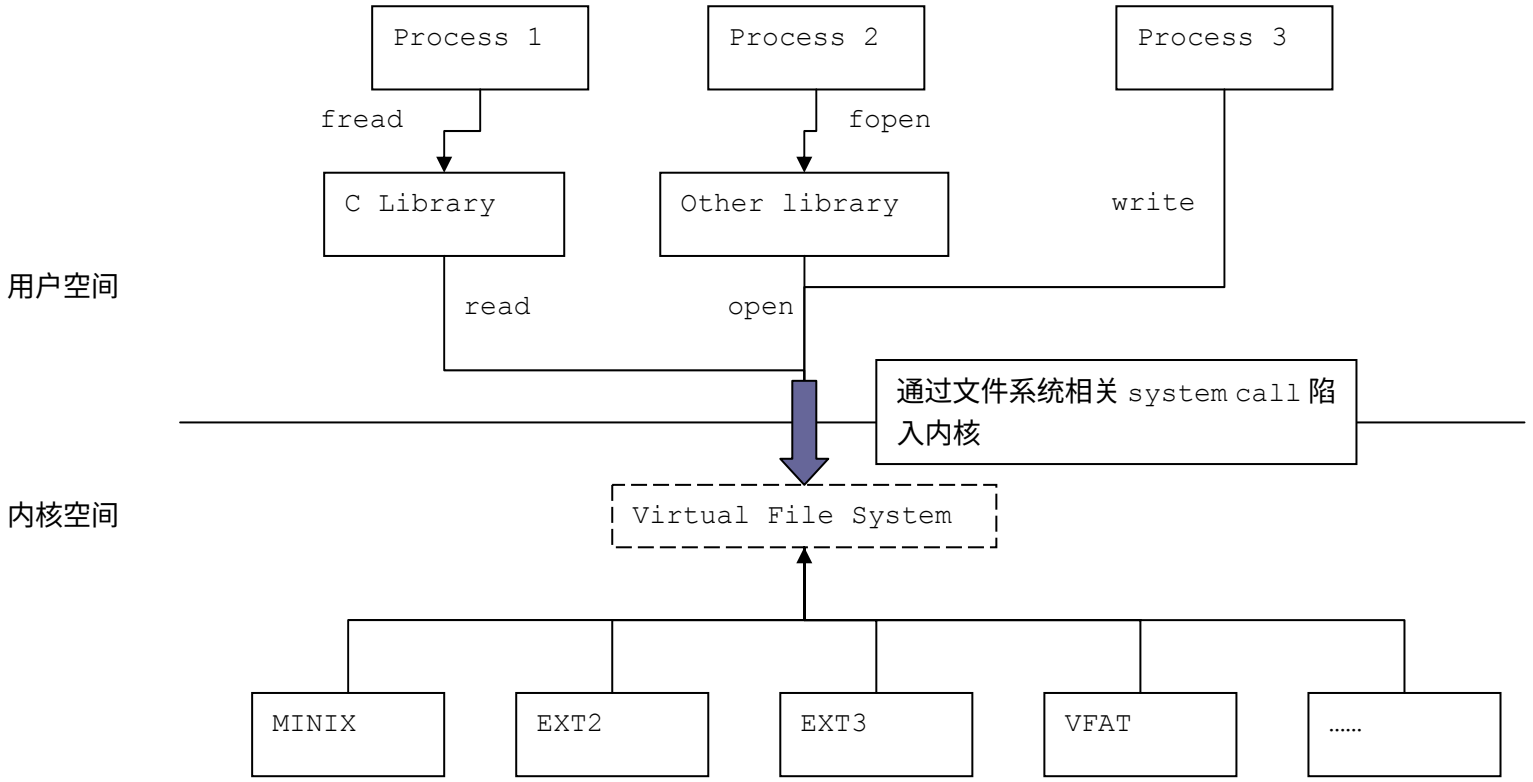
前言

本文详细分析了 MINIX 文件系统的磁盘布局及其在 Linux OS 中的实现。站在当今时代，MINIX 文件系统是如此简陋，以至于在现实中几乎没有实用价值。但它毕竟曾经是教学操作系统 MINIX 的主文件系统，最初的 Linux 也是在 MINIX 文件系统上开始成长的。在 Linux 内核源代码树中依然保留着 MINIX File System 的分支，而且它依然被维护，以与 Linux 文件系统的架构变化同步。从实用角度说，实在没有这个必要，但 Linus Torvalds 还保留着它，或许 Linus 有点舍不得吧？从学习文件系统角度说，MINIX 文件系统是一个极好的起点，它简单，但拥有 Unix 文件系统的主要要素。以它为基础，再学习 Linux OS 的主流文件系统，比如 ext2/ext3，会轻松得多（最起码这是我的学习体会）。

Linux 中的文件系统架构

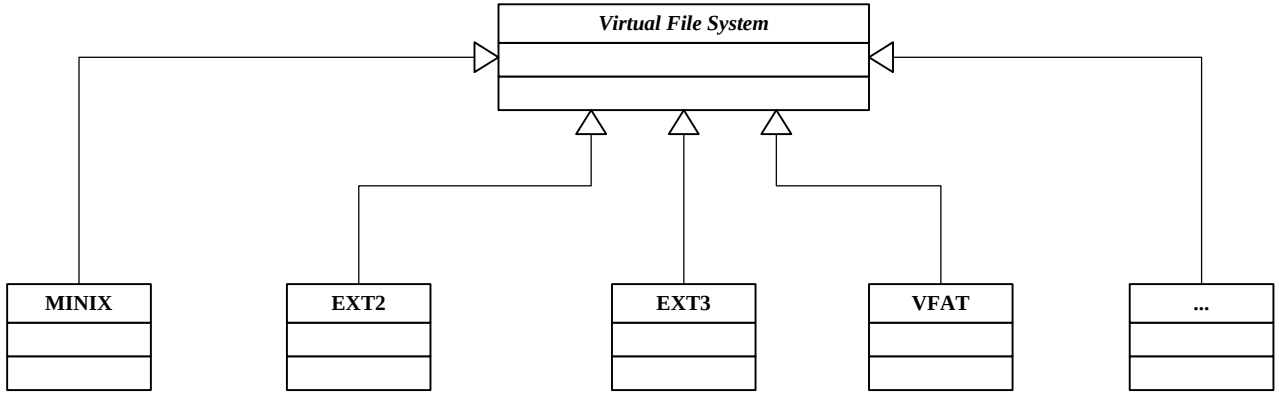
Linux 文件系统的架构介绍在 Linux 内核介绍的第一书籍《Understanding The Linux Kernel, 3rd Edition》中有极其精彩的分析，我不会再这里再照本宣科一遍。这里我想说说我的一些理解，不一定贴切，但最起码我觉得我这么理解蛮有新意的。☺

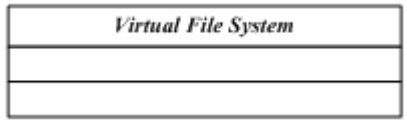
下面是我理解的 Linux 文件系统的架构图：



用户程序或者直接调用文件系统 system call，或者调用库中的与文件相关函数间接调用文件系统 system call，由内核中的 Virtual File System 来处理。

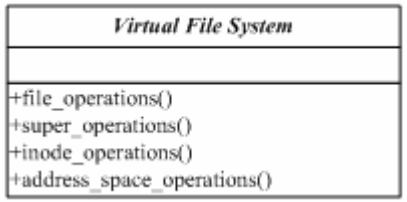
VFS(**V**irtual **F**ile **S**witcher)实现了一个虚拟的文件系统（Virtual File System)。该虚拟文件系统与实际的文件系统之间的关系如下面的 UML class diagram。





是一个抽象类，它不对应到任何实际的文件系统，但它却抽取并定义了实际文件系统的骨架，把相关文件系统共有的代码，数据结构等集成到该类中，并通过下面几个结构来提供客户定制化。

1. struct file_operations
2. struct super_operations
3. struct inode_operations
4. struct address_space_operations



Virtual File System 这个“ class ”定义了这 4 个多态接口（虚函数），任何的文件系统都必须全部或部分实现上面的接口。比如像 MINIX 文件系统中有如下代码：

Linux-2.6.20/fs/minix/inode.c

```
96 static struct super_operations minix_sops = {
97     .alloc_inode  = minix_alloc_inode,
98     .destroy_inode   = minix_destroy_inode,
99     .read_inode   = minix_read_inode,
100    .write_inode  = minix_write_inode,
101    .delete_inode= minix_delete_inode,
102    .put_super    = minix_put_super,
103    .statfs       = minix_statfs,
104    .remount_fs   = minix_remount,
105 };


338 static const struct address_space_operations minix_aops = {
339    .readpage = minix_readpage,
340    .writepage = minix_writepage,
341    .sync_page = block_sync_page,
```

```
342    .prepare_write = minix_prepare_write,
343    .commit_write = generic_commit_write,
344    .bmap = minix_bmap
345 };
346
347 static struct inode_operations minix_symlink_inode_operations = {
348    .readlink = generic_readlink,
349    .follow_link  = page_follow_link_light,
350    .put_link = page_put_link,
351    .getattr  = minix_getattr,
352 };
```

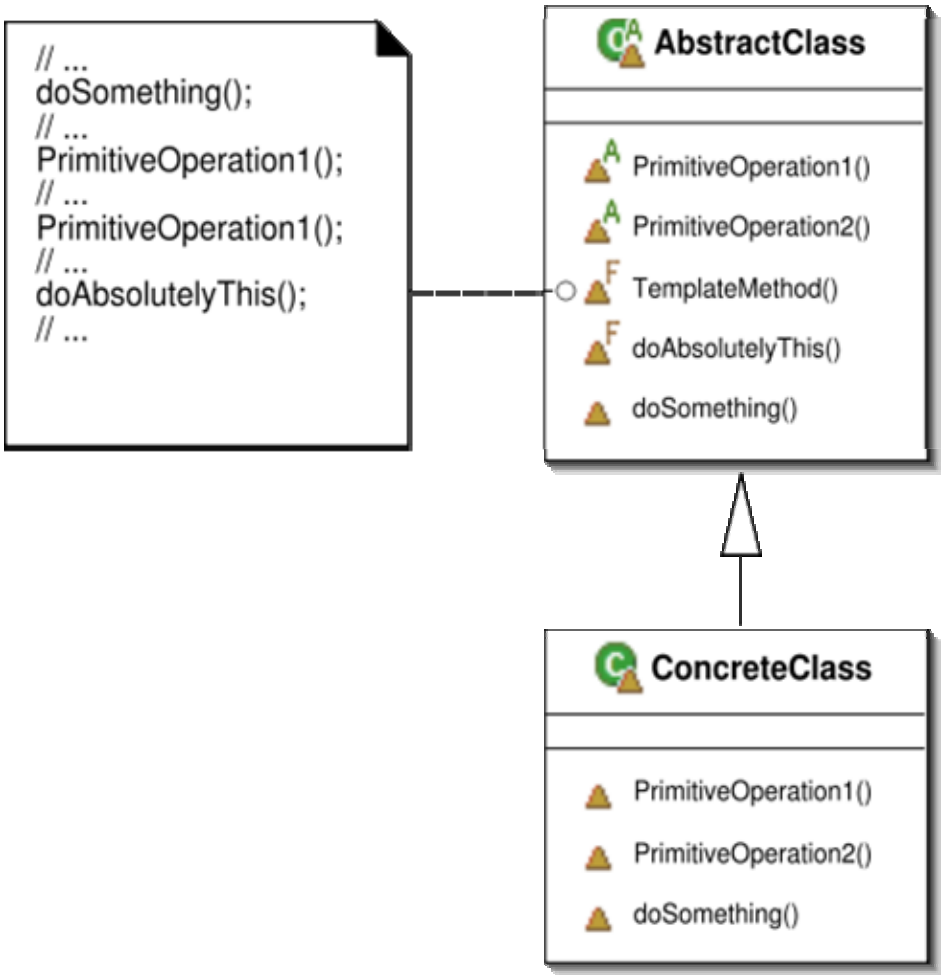
Linux-2.6.20/fs/minix/file.c

```
18 const struct file_operations minix_file_operations = {
19     .llseek       = generic_file_llseek,
20     .read     = do_sync_read,
21     .aio_read = generic_file_aio_read,
22     .write    = do_sync_write,
23     .aio_write    = generic_file_aio_write,
24     .mmap     = generic_file_mmap,
25     .fsync    = minix_sync_file,
26     .sendfile = generic_file_sendfile,
27 };
```

上面的代码就是在 MINIX 文件系统这个 Virtual File System 这个“子类”中重新实现了父类的“虚函数”。

在阅读 Linux 内核某个具体文件系统的源代码时，你会发觉该文件的驱动是零碎的，不完整的。为什么有这种感觉呢，很简单，因为具体文件系统中的代码只是实现该文件系统特定的功能，凡是所有文件系统共有的数据和操作都被上移到父类 Virtual File System 中去了，所以实现文件系统也就是去实现这几个固定的接口。这非常符合面向对象的设计思维吧。

另外，整个 Linux 内核文件系统，按我的理解是应用了 Design Pattern 中的“ Template Method Pattern ”。我们先看一下 Template Method Pattern 的类图：



下面是 ***Wekipedia*** 上对 `Template Method Pattern` 的解释：

A template method defines the program skeleton of an algorithm. The algorithm itself is made abstract, and the subclasses override the abstract methods to provide concrete behavior.

Virtual File System 就是“AbstractClass”，其定义了来自用户态程序访问文件系统操作的骨架（skeleton），但毕竟每个文件系统都是不同的，比如 super block 与 inode 的数据结构，inode bitmap 于数据区的分布等等，都是完全不同的。这些不同点就是上面的‘PrimitiveOperation”，由各个具体的文件系统驱动（即各个 ConcreteClass）来实现。上面的 4 大 xxx_operations 就是 Template Method 中的 PrimitiveOperation 1，PrimitiveOperation 2，PrimitiveOperation 3，PrimitiveOperation 4。

以 sys_read()系统调用为例：

```
asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;

    file = fget_light(fd, &fput_needed);
    if (file) {
        loff_t pos = file_pos_read(file);
        ret = vfs_read(file, buf, count, &pos);     这里就进入了 vfs(Virtual File System) 你可以把该函数 vfs_read()
        file_pos_write(file, pos);                  看成 VFS 中的一个 Template Method。
        fput_light(file, fput_needed);
    }

    return ret;
}
```

进入 Virtual File System class 中的 Template Method。

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
    ssize_t ret;

    if (!(file->f_mode & FMODE_READ))
        return -EBADF;
    if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
        return -EINVAL;
    if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
        return -EFAULT;

    ret = rw_verify_area(READ, file, pos, count);
    if (ret >= 0) {
        count = ret;
        ret = security_file_permission (file, MAY_READ);
        if (!ret) {
            if (file->f_op->read)
                ret = file->f_op->read(file, buf, count, pos);
            else
                ret = do_sync_read(file, buf, count, pos);
            if (ret > 0) {
                fsnotify_access(file->f_path.dentry);
                current->rchar += ret;
            }
            current->syscr++;
        }
```

这就是虚函数调用，该函数在各个子类，即实际的文件系统中实现。在父类代码中通过多态调用的是子类中的函数，典型的面向对象特征。

```
    }

    return ret;
}
```

上面的 ret = file->f_op->read(file, buf, count, pos)调用的是什么呢？这完全由打开（open）正在读的文件时选定的文件系统相关。如果正在读的文件在 MINIX 文件系统上，则该调用将调用的是下面表红的函数。

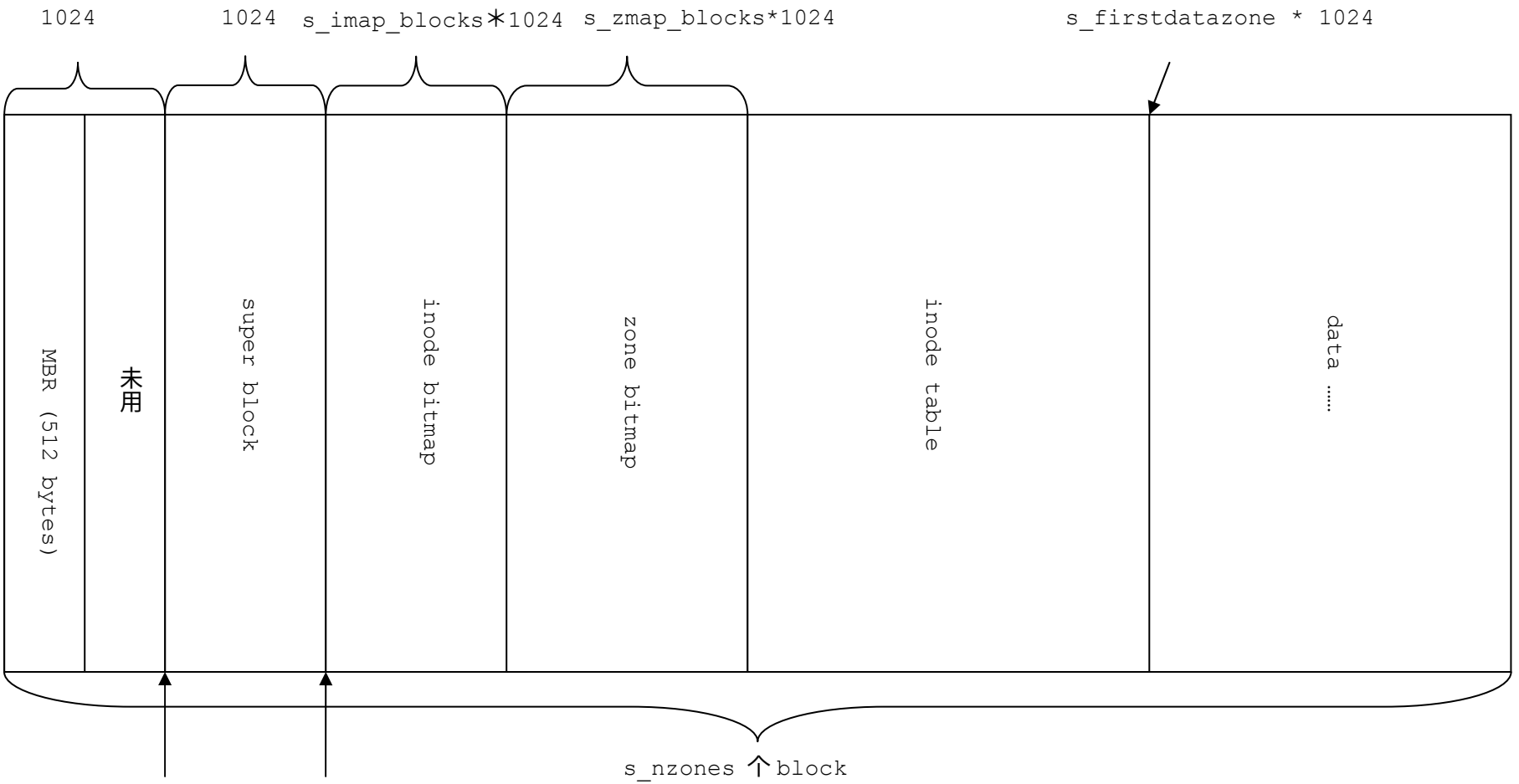
```
18 const struct file_operations minix_file_operations = {
19     .llseek       = generic_file_llseek,
20     .read      = do_sync_read,
21     .aio_read = generic_file_aio_read,
22     .write     = do_sync_write,
23     .aio_write    = generic_file_aio_write,
24     .mmap      = generic_file_mmap,
25     .fsync     = minix_sync_file,
26     .sendfile = generic_file_sendfile,
27 };
```

我在这里不想解释 file_operations,inode_operations,super_operations,address_space_operations 的具体含义,《Understanding The Linux Kernel, 3rd Edition》中的说明胜我百倍。

上面的理解你可以把它看成是一派胡言，因为在 Linux 内核源代码中根本没有 Virtual File System 这个 class，也没有子类的概念，整个内核是用 C 语言来编写的，没有用面向对象的语言，比如 C++。我对 Linux 文件系统的诠释完全基于自己的理解，有点“另类”。经典与权威的解释请看《Understanding The Linux Kernel, 3rd Edition》一书。我喜欢我的理解，它使我更容易抓住 Linux 文件系统架构的本质，但愿对你也有所帮助，但我可不保证一定正确哦！☺

MINIX 文件系统的 disk layout

下图是管理 MINIX 文件系统的元信息（Metadata）的磁盘布局。



MINIX 文件系统的总体磁盘布局

在 Linus 所写的 mkfs.minix 中的 write_tables()函数精确的反映了上图的磁盘布局。

```
241 static void
242 write_tables(void) {
243    /* Mark the super block valid. */
244    Super.s_state |= MINIX_VALID_FS;
245    Super.s_state &= ~MINIX_ERROR_FS;
246
247    if (lseek(DEV, 0, SEEK_SET))
248       die(_("seek to boot block failed in write_tables"));
249    if (512 != write(DEV, boot_block_buffer, 512))
250       die(_("unable to clear boot sector"));
251    if (BLOCK_SIZE != lseek(DEV, BLOCK_SIZE, SEEK_SET))
252       die(_("seek failed in write_tables"));
253    if (BLOCK_SIZE != write(DEV, super_block_buffer, BLOCK_SIZE))
254       die(_("unable to write super-block"));
255    if (IMAPS*BLOCK_SIZE != write(DEV,inode_map,IMAPS*BLOCK_SIZE))
256       die(_("unable to write inode map"));
257    if (ZMAPS*BLOCK_SIZE != write(DEV,zone_map,ZMAPS*BLOCK_SIZE))
258       die(_("unable to write zone map"));
259    if (INODE_BUFFER_SIZE != write(DEV,inode_buffer,INODE_BUFFER_SIZE))
260       die(_("unable to write inodes"));
261
262 }
```

1. 行 247 移动到分区的头部。
2. 行 249 在分区的头部开始的 512 字节空间写入 MBR。

3. 行 251 移动到分区的 block 1 处。
4. 行 253 在分区的 block 1 处写入一个 block 的 super block。指针移动到 block 2。
5. 行 255 在分区的 block 2 处写入 inode bitmap，其所占 IMAPS(即 s_imap_blocks)个 block。指针移动到 block 2 + s_imap_blocks。
6. 行 257 在分区的 block 2 + s_imap_blocks 处写入 zone bitmap，其所占 ZMAPS(即 s_zmap_blocks) 个 block。指针移动到 block 2 + s_imap_blocks + s_zmap_blocks。
7. 行 257 在分区的 block 2 + s_imap_blocks + s_zmap_blocks 处写入 inode table，其所占 INODE_BUFFER_SIZE 个。
8. 在 inode table 之后即是数据区，即用户真正可用的空间。

下面是本文实验 MINIX 文件系统的 hexdump。

```
[root@localhost disk-utils]# hexdump -C -s 0 /dev/hdd1 | less
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400  60 27 00 76 02 00 04 00  43 01 00 00 00 1c 08 10  |`'.v....C.......|
00000410  8f 13 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000420  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000800  ff ff 3f 00 00 00 00 00  00 00 00 00 00 00 00 00  |..?.............|
00000810  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000ce0  00 00 00 00 00 00 00 00  00 00 00 00 fe ff ff ff  |................|
00000cf0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
000010e0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff 03  |................|
000010f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001e90  00 00 00 00 00 00 00 c0  ff ff ff ff ff ff ff ff  |................|
```

0 到 0x3ff 是分区的引导块

0x400 到 0x7ff 是 super block

0x800 到 0xfff 是 inode bitmap 在这里有 2 个 block

从 0x1000 到 0x1fff 是 zone bitmap，在这里有 4 个 block

```
00001ea0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00002000  ed 41 00 00 a0 01 00 00  8b 5d c2 46 00 0d 43 01  |.A.......].F..C.|
00002010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002020  ed 41 00 00 a0 00 00 00  95 5f c2 46 00 02 44 01  |.A......._.F..D.|
00002030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002040  ed 41 00 00 40 00 00 00  58 5d c2 46 00 02 45 01  |.A..@...X].F..E.|
00002050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002060  ed 41 00 00 60 00 00 00  da 5f c2 46 00 03 46 01  |.A..`...._.F..F.|
00002070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002080  ed 41 00 00 40 00 00 00  6a 5d c2 46 00 02 47 01  |.A..@...j].F..G.|
00002090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020a0  ed 41 00 00 60 00 00 00  2b 60 c2 46 00 02 48 01  |.A..`...+`.F..H.|
000020b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020c0  ed 41 00 00 40 00 00 00  75 5d c2 46 00 02 49 01  |.A..@...u].F..I.|
000020d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020e0  ed 41 00 00 40 00 00 00  79 5d c2 46 00 02 4a 01  |.A..@...y].F..J.|
000020f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002100  ed 41 00 00 80 00 00 00  ac 5f c2 46 00 02 4b 01  |.A......._.F..K.|
00002110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002120  ed 41 00 00 60 00 00 00  0a 60 c2 46 00 02 4c 01  |.A..`....`.F..L.|
00002130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002140  ed 41 00 00 40 00 00 00  87 5d c2 46 00 02 4d 01  |.A..@....].F..M.|
00002150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002160  ed 41 00 00 40 00 00 00  8b 5d c2 46 00 02 4e 01  |.A..@....].F..N.|
00002170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

[0x2000,0x50c00)为 inode table，这里的 inode 是 32 字节

```
00002180  ed 81 00 00 f8 a9 00 00  72 5f c2 46 00 01 4f 01  |........r_.F..O.|
00002190  50 01 51 01 52 01 53 01  54 01 55 01 56 01 00 00  |P.Q.R.S.T.U.V...|
000021a0  ed 81 00 00 78 5d 01 00  81 5f c2 46 00 01 7b 01  |....x]..._.F..{.|
000021b0  7c 01 7d 01 7e 01 7f 01  80 01 81 01 82 01 00 00  ||.}.~...........|
000021c0  ed 81 00 00 7c 5a 03 00  95 5f c2 46 00 01 d4 01  |....|Z..._.F....|
000021d0  d5 01 d6 01 d7 01 d8 01  d9 01 da 01 db 01 00 00  |................|
000021e0  a4 81 00 00 07 07 00 00  a6 5f c2 46 00 01 ac 02  |........._.F....|
000021f0  ad 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002200  00 81 00 00 7b 04 00 00  ac 5f c2 46 00 01 ae 02  |....{...._.F....|
00002210  af 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002220  ed 41 00 00 60 00 00 00  eb 5f c2 46 00 02 b0 02  |.A..`...._.F....|
00002230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002240  a4 81 00 00 7c 00 00 00  eb 5f c2 46 00 01 b1 02  |....|...._.F....|
00002250  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002260  ed 81 00 00 04 03 18 00  0a 60 c2 46 00 01 b2 02  |.........`.F....|
00002270  b3 02 b4 02 b5 02 b6 02  b7 02 b8 02 b9 02 ba 04  |................|
00002280  ed 81 00 00 44 13 00 00  2b 60 c2 46 00 01 b7 08  |....D...+`.F....|
00002290  b8 08 b9 08 ba 08 bb 08  00 00 00 00 00 00 00 00  |................|
000022a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00050c00  01 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c20  01 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c40  02 00 62 69 6e 00 00 00  00 00 00 00 00 00 00 00  |..bin...........|
00050c50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

从这里开始是数据区

```
00050c60  03 00 75 73 72 00 00 00  00 00 00 00 00 00 00 00  |..usr...........|
00050c70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c80  04 00 68 6f 6d 65 00 00  00 00 00 00 00 00 00 00  |..home..........|
00050c90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050ca0  05 00 6d 69 73 63 00 00  00 00 00 00 00 00 00 00  |..misc..........|
00050cb0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050cc0  06 00 73 62 69 6e 00 00  00 00 00 00 00 00 00 00  |..sbin..........|
```

上面的 hexdump，现在你可能不知所以然，不要紧，在阅读完整篇文章后，你可以再回来看看，应该会有较深的体会。通过看磁盘的二进制映像，绝对有助于你对文件系统的理解。

super block

super block(超级块)包含有文件系统的整个布局信息。MINIX 文件系统的 super block 可以占用 1 个 block，即 1024 个 byte。但实际上只占用了很少几个字节，sizof(minix_super_block)个字节。它位于 block 1。

linux-2.6.20/include/linux/minix_fs.h

```
62 /*
63  * minix super-block data on disk
64  */
65 struct minix_super_block {
66     __u16 s_ninodes;
67     __u16 s_nzones;
68     __u16 s_imap_blocks;
```

```
69      __u16 s_zmap_blocks;
70      __u16 s_firstdatazone;
71      __u16 s_log_zone_size;
72      __u32 s_max_size;
73      __u16 s_magic;
74      __u16 s_state;
75      __u32 s_zones;
76  };
```

```
[root@localhost wzhou]# hexdump -C -s 0x400[1] -n 1024[2] /dev/hdd1 | less
00000400  60 27 00 76 02 00 04 00  43 01 00 00 00 1c 08 10  |`'.v....C.......|
00000410  8f 13 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000420  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000800
```

成员	十六进制	十进制	成员含义	
s_ninodes	0x2760	10080	该文件系统的总 Inode 数	
s_nzones	0x7600	30208	该文件系统的总 block 数，是指整个分区的 block 数，不是数据区的 block 数	
s_imap_blocks	0x002	2	inode bitmap 所占的 block 数	

[1] MINIX 文件系统的 super block 开始与第 1 个 block，而一个 block 的大小为 1024 字节。
[2] MINIX 文件系统的 super block 大小为占一个 block。

s_zmap_blocks	0x0004	4	zone bitmap 所占的 block 数	
s_firstdatazone	0x0143	323	Data block 从哪儿开始	
s_log_zone_size	0x00	0	$2^0=1$，即 1 个 zone 对应 1 个 block。(在此文中，我假设 zone 与 block 相同，并不加区分)。理论上该成员可以用于指定一个 zone 可以包含多少个 block，比如 1 的话，表示 1 个 zone 包含 $2^1=2$ 个 block，等等。但在 linus 的 mkfs.minix 的源代码里，对该成员的赋值固定是 0，即在 Linux 下的 minix 文件系统，zone 和 block 一一对应。所以我在此文中就不区分了。zone 即 block，block 即 zone。	
s_max_size	0x10081c00	268966912	允许的最大文件长度	为什么是这个数呢？该数等于 (7+512+512*512)*1024[3]
s_magic	0x138f		MINIX 文件系统的签名，这是升级版的 MINIX 文件系统	

[3] 在 mkfs.minix 的代码里有如下代码：461 Super.s_max_size = version2 ? 0x7fffffff : (7+512+512*512)*1024;

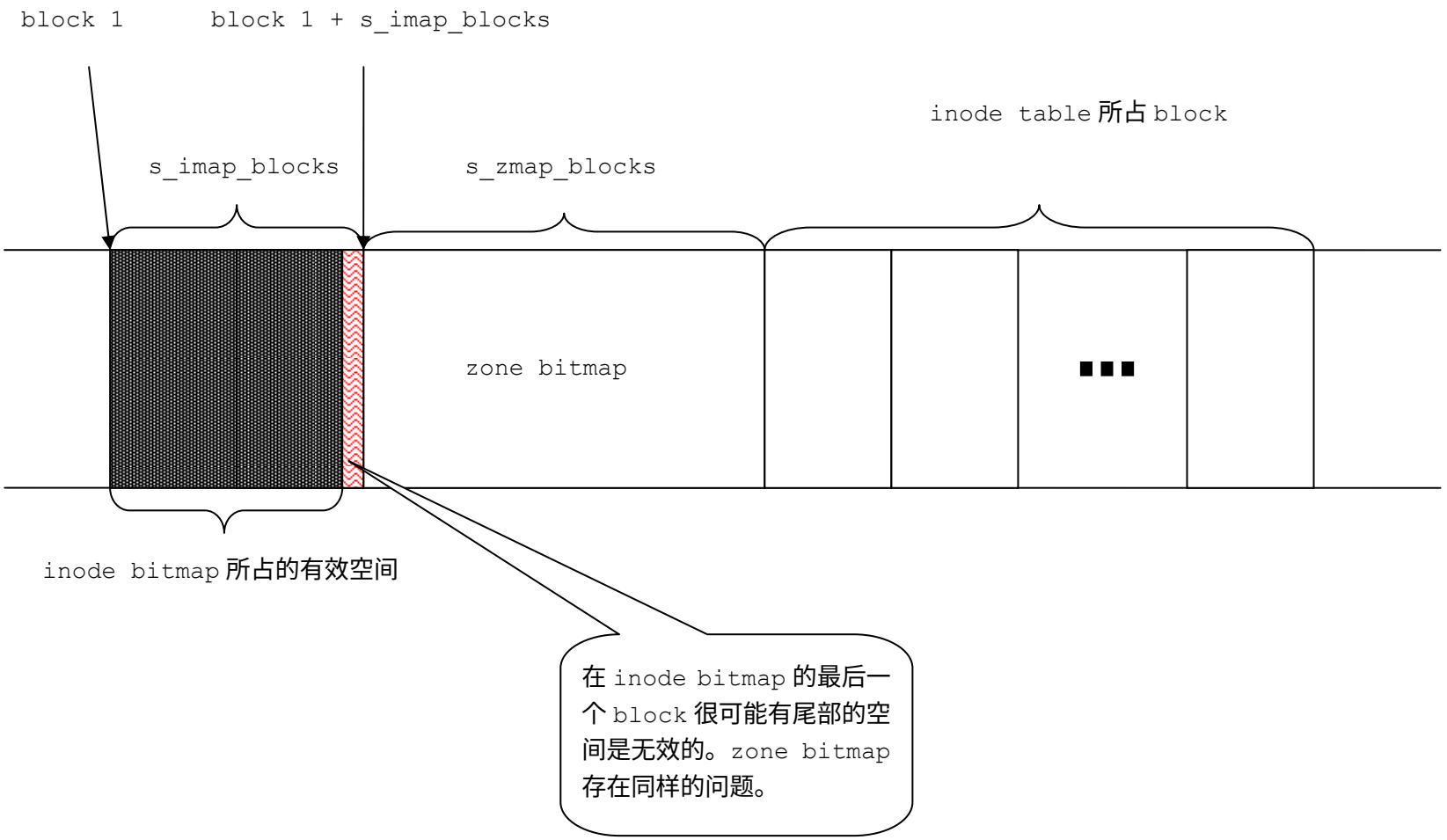
s_state	0x00	0x00		
s_zones	0x00	0x00	数据块数	

inode bitmap

inode bitmap 紧挨着 super block，从 block 2 开始。而其所占的空间则是由 inode table 的大小所决定的。inode bitmap 中的每一个 bit 对应 inode table 中的一项，置 1 表示对应的 inode 已经被占用，反之则表示空闲。mkfs.minix 在“格式化”（Windows 下的用语）分区时会对 inode table 所占的 block 数取整。即 inode table 是占满其所在的 block 的，不会在最后的一个 block 留有空隙。下面即是对 inode table 的取整逻辑。

```
472    /* Round up inode count to fill block size */
473    if (version2)
474        inodes = ((inodes + MINIX2_INODES_PER_BLOCK - 1) &
475             ~(MINIX2_INODES_PER_BLOCK - 1));
476    else
477        inodes = ((inodes + MINIX_INODES_PER_BLOCK - 1) &
478             ~(MINIX_INODES_PER_BLOCK - 1));
```

这样很显然，inode bitmap 所占空间则不一定是占满其所在的 block，一般最后一个 block 都留有空隙。


block 1
block 1 + s_imap_blocks
inode table 所占 block
s_imap_blocks
s_zmap_blocks
zone bitmap
■■■
inode bitmap 所占的有效空间
在 inode bitmap 的最后一个 block 很可能有尾部的空间是无效的。zone bitmap 存在同样的问题。

zone bitmap

zone bitmap 紧接着 inode bitmap 后面,其大小是由数据区的空间决定的。super block 中的 s_nzones 是指整个分区的 block 数,而 zone bitmap 中的位只用来标示数据区的 block，不包括 MBR，inode bitmap，zone bitmap，inode table 所占的 block。一个 block 为 1024 个 byte，有 8192 个 bit，可以指向连续的 8192 个数据区中的 block，即一个 block 的 zone bitmap 代表了用户可用的 8M 空间。而 zone bitmap 的第 0 位就代表数据区的第 0 个 block。

inode table

在 zone bitmap 之后紧接着就是 inode table。在 Unix 下，每个文件（注意目录也是一种特殊的文件）都有一个 inode 对应。整个 inode table 可以看成是 inode 构成的一个数组。MINIX 文件系统有原始版和升级版两种，对应的 inode 也有两种形式。

1. 原始版

```
30 /*
31  * This is the original minix inode layout on disk.
32  * Note the 8-bit gid and atime and ctime.
33  */
34 struct minix_inode {
35     __u16 i_mode;
36     __u16 i_uid;
37     __u32 i_size;
38     __u32 i_time;
39     __u8  i_gid;
40     __u8  i_nlinks;
41     __u16 i_zone[9];
42 };
```

2. 升级版

```
44 /*
45  * The new minix inode has all the time entries, as well as
46  * long block numbers and a third indirect block (7+1+1+1
47  * instead of 7+1+1). Also, some previously 8-bit values are
48  * now 16-bit. The inode is now 64 bytes instead of 32.
49  */
50 struct minix2_inode {
51     __u16 i_mode;
52     __u16 i_nlinks;
53     __u16 i_uid;
54     __u16 i_gid;
55     __u32 i_size;
56     __u32 i_atime;
57     __u32 i_mtime;
58     __u32 i_ctime;
59     __u32 i_zone[10];
60 };
```

从上可看出升级版的 inode 纪录了除 create time 外的 modification time(mtime)，access time(atime)。文件大小也变为 32 位(__u32)类型。在 mkfs.minix 的代码中又如下代码：

```
461    Super.s_max_size = version2 ? 0x7fffffff : (7+512+512*512)*1024;
```

这里 version2 非 0，即表示是升级版的 MINIX 文件系统。如果是原始文件系统，则最大文件大小为(7+512+512*512)*1024，而如果是升级版的，则为 2G。另外一个大的不同是 inode 中指向该 inode 所代表的文件存贮空间的 i_zone。差异有两点：

1. 原始版中是 16 位的 block number，而升级版中是 32 位，这样对数据区的寻址大大加强。

2. 原始版中最多到二次间接块，而升级版中有三次间接块。这当然也是为了与最大文件 size 进行匹配。因为在升级版中最大文件为 2G，这已经远远超出了二次间接块的范围了。

data area(数据区)

数据区也是用户和非文件系统驱动 programmer 能看到与操作的。没什么好说的。

目录与文件

在 MINIX 文件系统中目录是一种特殊的文件。既然也是文件，自然目录也用 inode 来标示。在目录标示的文件中的内容就是该目录下的文件与子目录。而每一项都用所谓目录项(directory entry)来表示。

```
78 struct minix_dir_entry {
79     __u16 inode;
80     char name[0];
81 };
```

这是一个变长的结构，因为原始版的 MINIX 文件系统的文件名长度最大允许 14 个字节，而升级版的则允许 30 个字节。

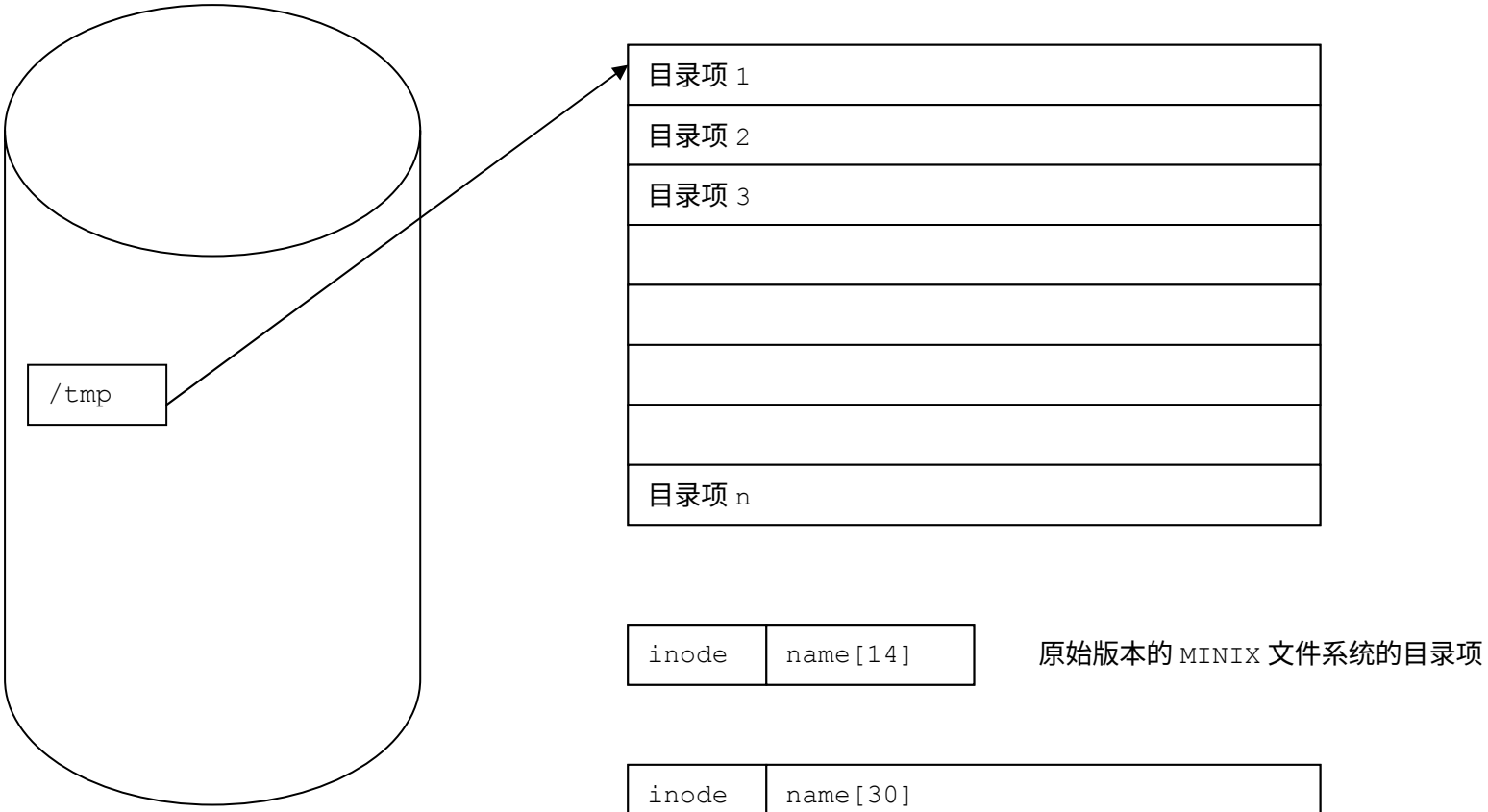
原始版的 MINIX 文件系统目录项：

```
   struct minix_dir_entry {
       __u16 inode;
       char name[14];
   };
```

升级版的 MINIX 文件系统目录项：

```
struct minix_dir_entry {
    __u16 inode;
    char name[30];
};
```

目录文件中的内容就是这些目录项。


/tmp
目录项 1
目录项 2
目录项 3
目录项 n
inode
name[14]
原始版本的 MINIX 文件系统的目录项
inode
name[30]
升级版本的 MINIX 文件系统的目录项

在上面《MINIX 文件系统的 disk layout》一节中，我们列出了实验用的 MINIX 文件系统的二进制映像。我们这里看一下该 MINIX 文件系统的根目录文件中的内容来理解目录文件中到底是什么东西。

```
[root@localhost disk-utils]# mount -t minix /dev/hdd1 /mnt/minix/          先把 MINIX 文件系统 mount 上
[root@localhost disk-utils]# ls -l /mnt/minix/
total 11
drwxr-xr-x 2 root root 160 Aug 15 10:06 bin              列出根目录包含的子目录
drwxr-xr-x 2 root root 128 Aug 15 10:06 etc
drwxr-xr-x 3 root root  96 Aug 15 10:07 home
drwxr-xr-x 2 root root  96 Aug 15 10:08 lib
drwxr-xr-x 2 root root  64 Aug 15 09:56 misc
drwxr-xr-x 2 root root  64 Aug 15 09:57 mnt
drwxr-xr-x 2 root root  64 Aug 15 09:57 opt
drwxr-xr-x 2 root root  96 Aug 15 10:08 sbin
drwxr-xr-x 2 root root  64 Aug 15 09:57 tmp
drwxr-xr-x 2 root root  64 Aug 15 09:56 usr
drwxr-xr-x 2 root root  64 Aug 15 09:57 var
```

让我们模拟 MINIX 文件系统驱动代码来“ run ”一下吧！

首先根目录文件的 inode 是 inode table 中的第一项，在实验 MINIX 文件系统的分区上 inode table 从 block 8（0x2000）开始。

```
[root@localhost disk-utils]# hexdump -C -s 0x2000 /dev/hdd1 | less
00002000  ed 41 00 00 a0 01 00 00  8b 5d c2 46 00 0d 43 01  |.A.......].F..C.|
00002010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002020  ed 41 00 00 a0 00 00 00  95 5f c2 46 00 02 44 01  |.A......._.F..D.|
00002030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

上面标兰的 32 字节即是代表根目录的 inode 的内容。

其意义解释如下：

```
34 struct minix_inode {
35     __u16 i_mode;
36     __u16 i_uid;
37     __u32 i_size;
38     __u32 i_time;
39     __u8  i_gid;
40     __u8  i_nlinks;
41     __u16 i_zone[9];
42 };
```

inode 成员	i_mode	i_uid	i_size	i_time	i_gid	i_nlinks	i_zone
值	0x41ed	0	0x1a0	0x46c25d8b	0	0x0d	0x143

由 i_zone[0]指出了根目录的文件在数据区的 block number 是 0x143，其文件大小为 0x1a0(416)字节。

0x143 * BLOCK_SIZE = 323 * 1024 = 330752 (0x50C00)

```
[root@localhost disk-utils]# hexdump -C -s 0x50c00 -n 416 /dev/hdd1
00050c00  01 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c20  01 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c40  02 00 62 69 6e 00 00 00  00 00 00 00 00 00 00 00  |..bin...........|
00050c50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c60  03 00 75 73 72 00 00 00  00 00 00 00 00 00 00 00  |..usr...........|
00050c70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

```
00050c80  04 00 68 6f 6d 65 00 00  00 00 00 00 00 00 00 00  |..home..........|
00050c90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050ca0  05 00 6d 69 73 63 00 00  00 00 00 00 00 00 00 00  |..misc..........|
00050cb0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050cc0  06 00 73 62 69 6e 00 00  00 00 00 00 00 00 00 00  |..sbin..........|
00050cd0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050ce0  07 00 74 6d 70 00 00 00  00 00 00 00 00 00 00 00  |..tmp...........|
00050cf0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d00  08 00 76 61 72 00 00 00  00 00 00 00 00 00 00 00  |..var...........|
00050d10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d20  09 00 65 74 63 00 00 00  00 00 00 00 00 00 00 00  |..etc...........|
00050d30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d40  0a 00 6c 69 62 00 00 00  00 00 00 00 00 00 00 00  |..lib...........|
00050d50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d60  0b 00 6d 6e 74 00 00 00  00 00 00 00 00 00 00 00  |..mnt...........|
00050d70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d80  0c 00 6f 70 74 00 00 00  00 00 00 00 00 00 00 00  |..opt...........|
00050d90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

这就是根目录文件的内容！

上面每两行（32 个 byte）构成一个目录项。

```
struct minix_dir_entry {
    __u16 inode;
    char name[14];
};
```

比如：

```
00050c00  01 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

这里 inode = 1，name[14] = “ . ”。根目录的本身当然指向自身。

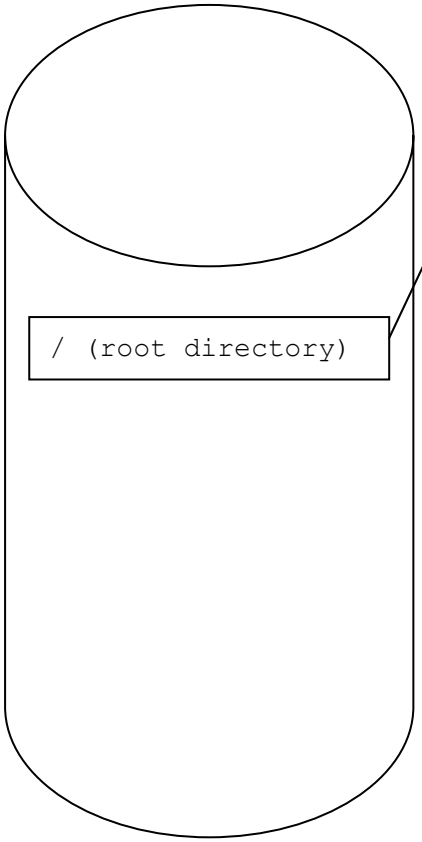
```
00050c20  01 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

这里 inode = 1，name[14] = “ .. ”。根目录父目录当然还是指向自身。

```
00050c40  02 00 62 69 6e 00 00 00  00 00 00 00 00 00 00 00  |..bin...........|
00050c50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

这里 inode = 2，name[14] = “ bin ”。至于目录 bin 则有 inode number 2 来指示。

所以根目录中的内容可用下图表示：



1	.
1	..
2	bin
3	usr
4	home
5	misc
6	sbin
7	tmp
8	var
9	etc
0xa	lib
0xb	mnt
0xc	opt

root directory 中的内容

从磁盘布局想到的

看了 MINIX 文件系统在磁盘上的布局，可能你会产生如下想法：inode 数与数据区的 block 数之间有什么关系呢？（最起码刚接触文件系统时我就很有疑问。）
一个分区的总 block 数是固定的，inode bitmap 与 inode table 是同方向变化的，而 zone bitmap 与数据区是也是同方向变化的。当然两者变化的幅度是不一样的。下面详细分析一下。
这里以原始的 inode 为例。inode bitmap 中的一位，代表一个 inode，即 sizeof(struct minix_inode)，为 32 个 byte。即两者间的变化幅度差异为 1:32。而 zone bitmap 中的一位代表数据区的一个 block,即变化幅度差异为 1:1024。假设 inode bitmap 占 x 个 block,而 zone bitmap 占 y 个 block，而 MINIX 文件系统所占分区的总 block 数为常熟 c，则显然有下面的公式。

$$1^{4} + 1^{5} + x^{6} + y^{7} + 32x^{8} + 1024y^{9} = c^{10}$$

把上面的公式化简以下：

$$33x + 1025y = C'$$ (C'依然时常数)

小学的公式，分配给 x(inode table)的多了，自然相应的数据区就少了。但问题不是那么简单，这里的关键是 x，y 前面的系数。一个是 33，一个是 1025，相差有 31 倍。即你多分配给 inode 相关的空间所造成的 block 数的变化要远远小于分配给数据区而引起的变化。inode 数代表了用户可以在文件系统上建立的文件数（包括目录）。从这一点看，为了避免出现数据区还有 20-30%的空闲，但系统报告已经不能再创建文件，因为 inode 被用光了的尴尬，你还是多分配一点空间给 inode 空间。一是没有了上面的尴尬，二是从上面的公式可知这种分配即使造成另一种“ 尴尬 ”，即数据区已经用光，但 inode 还有富裕，其浪费的空间也远远小于前者的浪费（1/31）。

---

[4] MBR 所占 block
[5] super block 所占 block
[6] inode bitmap 所占 block 数
[7] zone bitmap 所占 block 数
[8] inode table 所占 block 数
[9] 数据区所占 block 数
[10] 分区总 block 数

从上面的分析，可以得出分配 inode 与数据区的一个指导性原则：

如果在要“ format ”的分区上工作环境上，文件比较多，但各个文件又都比较小，则你需要加大 inode 的数目；反之，要建立的文件超大，而文件数目较少，则可以减少 inode 数目。

你可能问，如果文件又多，文件又大怎么办，那你只能去买个更大的硬盘了。有时候跟很多事情一样，钱多了，很多问题都不存在了。说句题外话，当

你有时候因千思万虑的想用某种技术解决某个问题而很痛苦，实际上你可以换一种思路，用钱解决吗，原始，简单，但确实有效。☺

上面的建议不单单对 MINIX 文件系统有效，几乎对所有 UNIX 下的文件系统都有效。但对 Windows 下的 FAT 类型的文件系统无效，因为 FAT 是完全不同的一种文件系统。当然，FAT 一般也没有这种问题。

当你“ format ”你的分区时，对应的工具都提供指定 inode 数的选项。比如 mkfs.minix 就有“ -i ”option。对 mkfs.minix 而言，如果你不指定 inode 数目，则 inode 数为整个分区 block 数的 1/3，代码如下：

```
467 /* some magic nrs: 1 inode / 3 blocks */
468    if ( req_nr_inodes == 0 )
469        inodes = BLOCKS/3;
470    else
471        inodes = req_nr_inodes;
```

上面如果用户通过-i option 指定了 inode 数，则 req_nr_inodes 为非 0，否则就是总 block 数除以 3。

格式化分区工具 mkfs.minix 解析

要理解某个文件系统，莫如看它是怎样被制作出来的，也就是怎样“格式化”的。MINIX文件系统的格式化工具是mkfs.minix[11]。虽然代码不长，但那可是Linux之父的手笔啊！整个源码都在一个C文件util-linux-2.12r\disk-utils\mkfs.minix.c中。代码不长，只要稍有C语言编程知识应该都能看懂。整个源码的注释在附录中的 **MINIX分区格式化工具mkfs.minix源码注释**一节。这里我仅列出主要的过程。

1. 首先获得分区的大小，即总 block 数。
2. 原始 MINIX 文件系统不能大于 64M，而升级版的则没有该限制。
3. 检查要格式化的分区是否被 mount 着。不能对正 mount 的分区格式化。
4. 准备两个目录项，根目录“/”和坏块文件“/.badblocks”。格式化后的分区并不是空空如也，最起码根目录是毕有的，如果该分区又有坏块的话，还会有/.badblocks 这个文件。
5. 在内存里先建立 MINIX 文件系统的元信息（super block，inode bitmap，zone bitmap，inode table）。
6. 如果用户要检查分区上是否有坏块，则检查。
7. 如果用户指定了包含已知坏块的文件，则处理之。
8. 在建立在内存中的 inode bitmap，zone bitmap，inode table 中建立根目录和坏块文件。
9. 标记其他的块为空闲可利用块。
10. 把内存中的 inode bitmap，zone bitmap，inode table 写入分区。

[11] 现在你安装的 Linux 一般都不会装 mkfs.minix 的 utility 了，因为 MINIX 文件系统实在只具有教学意义，不具有实用价值。你可以去网上下载 util-linux 的源码包来编译。

MINIX File System Driver 解析

Linux 最初始的文件系统借用了 MINIX 操作系统的文件系统，也就是这里分析的 MINIX 文件系统。它可是 Linux 之父的手笔！从理解 MINIX 文件系统实现学习的角度，选择较早版本的 MINIX 文件系统驱动可能更易理解，但我已经厌倦了世面上中国人写的形形色色的有关 Linux 内核相关的书都拿较老版本的书来分析，有的夸张到拿 0.01 版本的来学习，我不知作者是何居心。是出于好心，让学习者能易懂？可学习这远离现实太远的东东跟不学有什么区别？是怕用最新的版本，读者太笨，理解不了？一本技术书对该技术的初学者而言如果读第一遍的时候就大叹写得真好，真容易理解，那这本书的价值大概跟擦屁股纸差不了多少。台湾就盛产这种技术书，所谓能无痛而快乐的学习某种技术（我真怀疑那能叫技术吗？），但到目前为止，我还未发现有。说句题外话，台湾还算是仁慈的，因为你买了本技术书，虽然花了笔银子，但反正读书的过程到也没太痛苦。虽然除了些概念，没学到什么，但也没对自己的心灵造成什么伤害。可如果你买的是我们大陆这边的技术书的话，很可能在读书的过程中对自己的父母产生怨恨，为什么把自己生得那么笨！唉，真是陪了夫人又折兵，花钱花时间花精力，心灵又遭受煎熬，又没学到什么，最终还得感叹中国的这些大师们真非我等凡夫俗子所能理解。对很多国内的大师，只能用一句话来说，“ 你们真太有才了 ”!

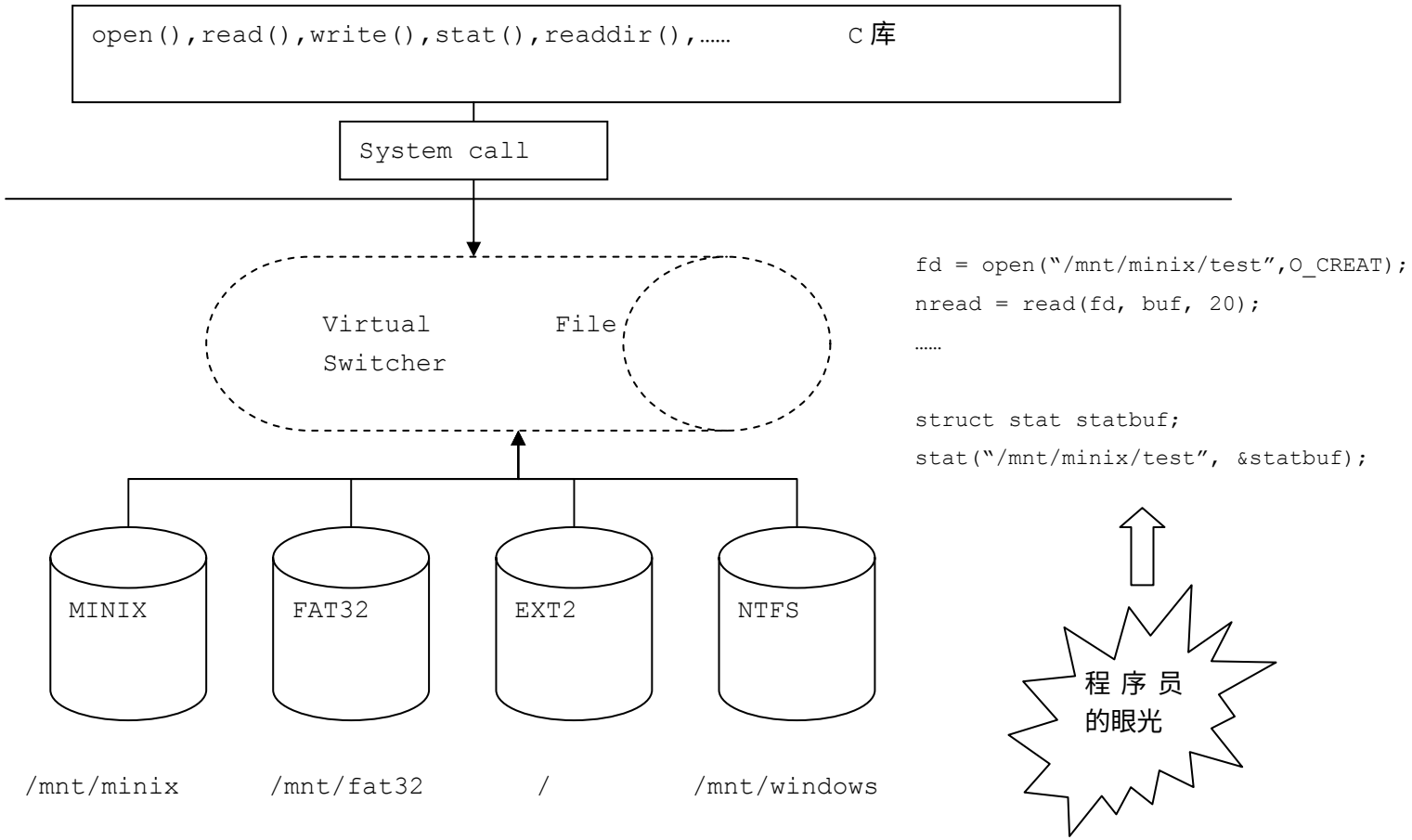
我这里解析的版本是 2.6.20(当前最新的 Linux 内核版本是 2.6.22)。由于具体的某个文件系统驱动只是 Linux 内核文件系统大架构下的一个 component，所以如果单看某个文件系统的具体实现而不了解 VFS(Virtual Filesystem Switcher)，则会见木不见林。而 VFS 的最权威说明还是那本 Linux 内核介绍的权威《Understanding The Linux Kernel, 3rd Edition》。小子实在不敢多言。

Programmer 眼中的文件系统实现

对 programmer 而言，对某个文件系统的访问就是一系列的与文件操作相关的 API(System Call)。像

```
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int readdir(unsigned int fd, struct dirent *dirp, unsigned int count);
int rename(const char *oldpath, const char *newpath);
int stat(const char *file_name, struct stat *buf);

……
```


open(),read(),write(),stat(),readdir(),……
C 库
System call
Virtual
Switcher
File
MINIX
FAT32
EXT2
NTFS
/mnt/minix
/mnt/fat32
/
/mnt/windows
fd = open(“/mnt/minix/test”,O_CREAT);
nread = read(fd, buf, 20);
……
struct stat statbuf;
stat(“/mnt/minix/test”, &statbuf);
程 序 员
的眼光

程序员通过传递给 system call 的路径（path）来访问不同的文件系统。对他而言，无需知道具体文件系统的知识，比如 super block 位于哪儿，有哪些信息，数据区从哪儿开始，inode table 有多大，这一切都被 OS 中的文件系统部分封装在一个个 system call 中。在这里，我将从程序员的眼光出发，来看一下 Linux 操作系统是怎样通过文件系统相关 system call 来把 Programmer 看到的“ 文件系统 ”与我在上面几节分析的 MINIX 文件系统联系起来的。由于文件系统相关 system call 比较多，我不可能全部介绍，这里只挑了几个最典型的来介绍。

1. open()
2. read()
3. rename()
4. mkdir()

下面的 system call 的分析，我将以手工方式来解读。

所谓“ 手工 ”，即是面对 MINIX 文件系统的磁盘二进制映像，programmer 怎样用大脑这个 CPU 来实现系统调用的，比如像打开文件这个操作，我们会追踪怎样从“ / ”目录开始（“ / ”目录是固定位置的，不需要寻找）一直找到要打开的文件的位置。寻找的依据就是上面几节介绍的 MINIX 文件系统的数据结构。通过这个“ 手工 ” open 的过程，读者绝对会的 MINIX 文件系统上面的 open 操作有一个崭新的认识。

open 解读

在 MINIX 文件系统的/tmp/test 目录下有一个文本文件 a.txt，包含下面一段文本“ This is a test text file. ”对 programmer 而言是下面一行代码：

```
int file = open(“/tmp/test/a.txt”, O_RDONLY);
```

该 open 调用的主要作用就是在磁盘上找到代表“ /tmp/test/a.txt ”文件的 inode。

“ /tmp/test/a.txt ”路径可分为下面 4 个部分：

1. /

2. tmp
3. test
4. a.txt

现在我们只知道“ / ”的位置，其他部分的位置信息都依赖于其上层部分来告诉。“ a.txt ”的位置信息由“ test ”告知，“ test ”的位置信息由“ tmp ”告知，“ tmp ” 的位置信息由“ / ”告知。OK，一切的源头从根目录开始。

MINIX文件系统的根目录的inode number是 1，它是inode table这个数组的index，如果把inode table看成是inode[]的话，则由inode number这个索引可获得inode节点的信息，inode[inode number – 1]。这里之所以要“ -1 ”，是因为inode number的编号是从 1（根节点号）开始，而inode table中的数组的index是从零开始的(在文件系统驱动代码中看到减去某个固定常数[12],就是这个原因)。在我们实验的MINIX文件系统中inode table开始与第 8 个block，也就是从分区头的 0x2000 偏移开始。

```
[root@localhost test]# hexdump -C  -s 0x2000 -n 32[13] /dev/hdd1
00002000  ed 41 00 00 a0 01 00 00  8b 5d c2 46 00 0d 43 01  |.A.......].F..C.|
00002010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

解读上面的数据如下

inode 成员	i_mode	i_uid	i_size	i_time	i_gid	i_nlinks	i_zone
值	0x41ed	0	0x1a0	0x46c25d8b	0	0x0d	0x143

从 i_zone 可知根目录文件在该分区的第 0x143 个 block 上，即偏移 0x50c00 开始处。

```
[root@localhost test]# hexdump -C  -s 0x50c00 -n 1024 /dev/hdd1
00050c00  01 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c20  01 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c40  02 00 62 69 6e 00 00 00  00 00 00 00 00 00 00 00  |..bin...........|
00050c50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

[12] 之所以是常数，是因为根节点的 inode number 并不一定为 1，比如 ext2 文件系统的根节点的 inode number 就为 2。
[13] 我们实验的 MINIX 文件系统的 inode 大小为 32 个字节。

```
00050c60  03 00 75 73 72 00 00 00  00 00 00 00 00 00 00 00  |..usr...........|
00050c70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050c80  04 00 68 6f 6d 65 00 00  00 00 00 00 00 00 00 00  |..home..........|
00050c90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050ca0  05 00 6d 69 73 63 00 00  00 00 00 00 00 00 00 00  |..misc..........|
00050cb0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050cc0  06 00 73 62 69 6e 00 00  00 00 00 00 00 00 00 00  |..sbin..........|
00050cd0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050ce0  07 00 74 6d 70 00 00 00  00 00 00 00 00 00 00 00  |..tmp...........|   找到“ tmp ”的名字，并获得 tmp 的 inode
00050cf0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|   number(07)
00050d00  08 00 76 61 72 00 00 00  00 00 00 00 00 00 00 00  |..var...........|
00050d10  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d20  09 00 65 74 63 00 00 00  00 00 00 00 00 00 00 00  |..etc...........|
00050d30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d40  0a 00 6c 69 62 00 00 00  00 00 00 00 00 00 00 00  |..lib...........|
00050d50  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d60  0b 00 6d 6e 74 00 00 00  00 00 00 00 00 00 00 00  |..mnt...........|
00050d70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00050d80  0c 00 6f 70 74 00 00 00  00 00 00 00 00 00 00 00  |..opt...........|
00050d90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00051000
```

“ / ”是目录文件，自然其文件内容包含的都是在根目录下的子目录或文件名。从上面标兰的可看到“ tmp ”正是我们要接下来要找的部分，从而得到“ tmp ”的 inode number，07。即获得“ tmp ”目录的 inode 信息，inode[7 – 1]。

```
[root@localhost ~]# hexdump -C -s 0x2000 -n 224 /dev/hdd1
```

```
00002000  ed 41 00 00 a0 01 00 00  8b 5d c2 46 00 0d 43 01  |.A.......].F..C.|
00002010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002020  ed 41 00 00 a0 00 00 00  95 5f c2 46 00 02 44 01  |.A......._.F..D.|
00002030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002040  ed 41 00 00 40 00 00 00  58 5d c2 46 00 02 45 01  |.A..@...X].F..E.|
00002050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002060  ed 41 00 00 60 00 00 00  da 5f c2 46 00 03 46 01  |.A..`...._.F..F.|
00002070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002080  ed 41 00 00 40 00 00 00  6a 5d c2 46 00 02 47 01  |.A..@...j].F..G.|
00002090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020a0  ed 41 00 00 60 00 00 00  2b 60 c2 46 00 02 48 01  |.A..`...+`.F..H.|
000020b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020c0  ed 41 00 00 60 00 00 00  5f 9a cb 46 00 03 49 01  |.A..`..._..F..I.|   这两行就是 inode[6]的信息
000020d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

解读上面的数据：

inode 成员	i_mode	i_uid	i_size	i_time	i_gid	i_nlinks	i_zone
值	0x41ed	0	0x60	0x46cb9a5f	0	0x03	0x149

从上可知“ tmp ”目录文件的内容在 block 0x149，即偏移 0x52400 处。

```
[root@localhost ~]# hexdump -C -s 0x52400 -n 1024 /dev/hdd1
00052400  07 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00052410  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00052420  01 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00052430  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00052440  16 00 74 65 73 74 00 00  00 00 00 00 00 00 00 00  |..test..........|   这就是我们要找的“ test ”目录，得到
00052450  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|   其 inode number，0x16
```

```
*
00052800
```

“ /tmp ”是目录文件，自然其文件内容包含的都是子目录或文件名。从上面标兰的可看到“ test ”正是我们要接下来要找的部分，从而得到“ test ”的 inode number，0x16。即获得“ test ”目录的 inode 信息，inode[0x16 - 1]。

```
[root@localhost ~]# hexdump -C -s 0x2000 -n 704 /dev/hdd1
00002000  ed 41 00 00 a0 01 00 00  8b 5d c2 46 00 0d 43 01  |.A.......].F..C.|
00002010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002020  ed 41 00 00 a0 00 00 00  95 5f c2 46 00 02 44 01  |.A......._.F..D.|
00002030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002040  ed 41 00 00 40 00 00 00  58 5d c2 46 00 02 45 01  |.A..@...X].F..E.|
00002050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002060  ed 41 00 00 60 00 00 00  da 5f c2 46 00 03 46 01  |.A..`...._.F..F.|
00002070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002080  ed 41 00 00 40 00 00 00  6a 5d c2 46 00 02 47 01  |.A..@...j].F..G.|
00002090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020a0  ed 41 00 00 60 00 00 00  2b 60 c2 46 00 02 48 01  |.A..`...+`.F..H.|
000020b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020c0  ed 41 00 00 60 00 00 00  5f 9a cb 46 00 03 49 01  |.A..`..._..F..I.|
000020d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020e0  ed 41 00 00 40 00 00 00  79 5d c2 46 00 02 4a 01  |.A..@...y].F..J.|
000020f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002100  ed 41 00 00 80 00 00 00  ac 5f c2 46 00 02 4b 01  |.A......._.F..K.|
00002110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002120  ed 41 00 00 60 00 00 00  0a 60 c2 46 00 02 4c 01  |.A..`....`.F..L.|
00002130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002140  ed 41 00 00 40 00 00 00  87 5d c2 46 00 02 4d 01  |.A..@....].F..M.|
```

```
00002150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002160  ed 41 00 00 40 00 00 00  8b 5d c2 46 00 02 4e 01  |.A..@....].F..N.|
00002170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002180  ed 81 00 00 f8 a9 00 00  72 5f c2 46 00 01 4f 01  |........r_.F..O.|
00002190  50 01 51 01 52 01 53 01  54 01 55 01 56 01 00 00  |P.Q.R.S.T.U.V...|
000021a0  ed 81 00 00 78 5d 01 00  81 5f c2 46 00 01 7b 01  |....x]..._.F..{.|
000021b0  7c 01 7d 01 7e 01 7f 01  80 01 81 01 82 01 00 00  ||.}.~...........|
000021c0  ed 81 00 00 7c 5a 03 00  95 5f c2 46 00 01 d4 01  |....|Z..._.F....|
000021d0  d5 01 d6 01 d7 01 d8 01  d9 01 da 01 db 01 00 00  |................|
000021e0  a4 81 00 00 07 07 00 00  a6 5f c2 46 00 01 ac 02  |.........._.F....|
000021f0  ad 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002200  00 81 00 00 7b 04 00 00  ac 5f c2 46 00 01 ae 02  |....{...._.F....|
00002210  af 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002220  ed 41 00 00 60 00 00 00  eb 5f c2 46 00 02 b0 02  |.A..`...._.F....|
00002230  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002240  a4 81 00 00 7c 00 00 00  eb 5f c2 46 00 01 b1 02  |....|...._.F....|
00002250  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00002260  ed 81 00 00 04 03 18 00  0a 60 c2 46 00 01 b2 02  |.........`.F....|
00002270  b3 02 b4 02 b5 02 b6 02  b7 02 b8 02 b9 02 ba 04  |................|
00002280  ed 81 00 00 44 13 00 00  2b 60 c2 46 00 01 b7 08  |....D...+`.F....|
00002290  b8 08 b9 08 ba 08 bb 08  00 00 00 00 00 00 00 00  |................|
000022a0  ed 41 00 00 80 00 00 00  84 9a cb 46 00 02 bc 08  |.A.........F....|
000022b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

解读上面的数据：

inode 成员	i_mode	i_uid	i_size	i_time	i_gid	i_nlinks	i_zone
值	0x41ed	0	0x80	0x46cb9a84	0	0x02	0x8bc

从上可知“ test ”目录文件的内容在 block 0x8bc，即偏移 0x22f000 处。

```
[root@localhost ~]# hexdump -C -s 0x22f000 -n 1024 /dev/hdd1
0022f000  16 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f020  07 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f040  00 00 2e 61 2e 74 78 74  2e 73 77 70 00 00 00 00  |...a.txt.swp....|
0022f050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f060  18 00 61 2e 74 78 74 00  00 00 00 00 00 00 00 00  |..a.txt.........|   找到“a.txt”文件，其inode number
0022f070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|   为0x18
*
0022f400
```

“ /tmp/test ”是目录文件，自然其文件内容包含的都是子目录或文件名。从上面标兰的可看到“ a.txt ”正是我们要接下来要找的部分，从而得到“ a.txt ”的 inode number，0x18。即获得“ a.txt ”文件的 inode 信息，inode[0x18 - 1]。

```
[root@localhost ~]# hexdump -C -s 0x22e0 -n 32 /dev/hdd1
000022e0  a4 81 00 00 1a 00 00 00  84 9a cb 46 00 01 ca 08  |...........F....|
000022f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

解读上面的数据：

inode 成员	i_mode	i_uid	i_size	i_time	i_gid	i_nlinks	i_zone
值	0x41e4	0	0x1a	0x46cb9a84	0	0x01	0x8ca

上面的 i_zone 指示的即是“ a.txt ”文件的内容所在的 block number，0x8ca。

```
[root@localhost ~]# hexdump -C -s 0x232800 /dev/hdd1 -n 26
00232800  54 68 69 73 20 69 73 20  61 20 74 65 73 74 20 74  |This is a test t|
00232810  65 78 74 20 66 69 6c 65  2e 0a                    |ext file..|
```

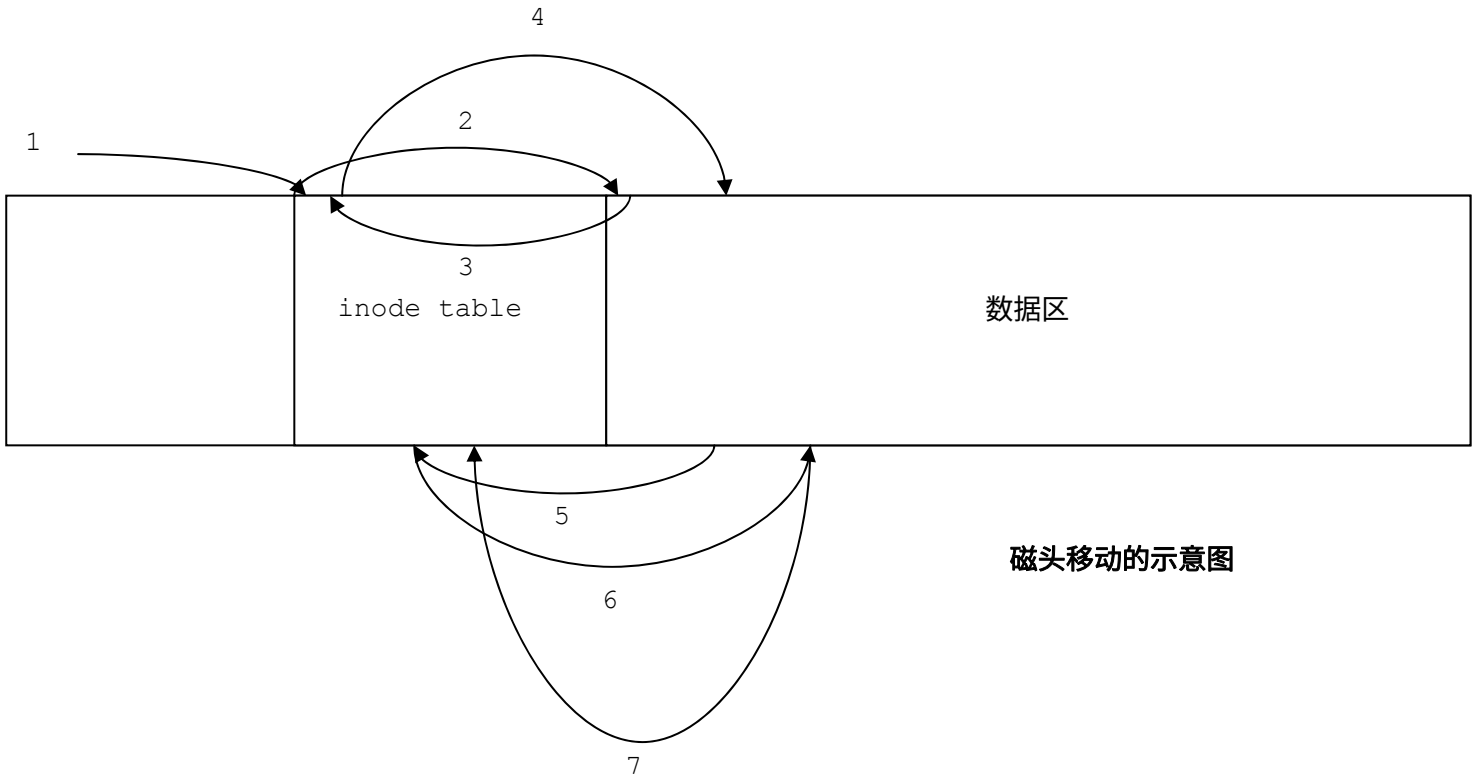
啊！千辛万苦总算看到“ /tmp/test/a.txt ”文件的内容了。

而 open()就是建立 file descriptor 与“ /tmp/test/a.txt ”文件的 inode 之间的关系。

对于这么一个并不算长的路径，从“ / ”开始到获得“ a.txt ”的 inode，这里面需要多少次读盘呢？我们可以算一下。

1．从 inode table 中读取根目录的 inode 信息。（第一次）
2．由 root inode 中获取 root 目录文件的 block number，然后读取该 block。（第二次）
3．在 root 目录文件的内容的 block 中得到“ tmp ”目录的 inode number，然后根据该 inode number 在 inode table 中读取“ tmp ”目录的 inode 信息。（第三次）
4．由“ tmp ”的 inode 中获得其内容所在的 block number，然后读取该 block。（第四次）
5．在“ tmp ”目录文件中的内容的 block 中得到“ test ”目录的 inode number，然后根据该 inode number 在 inode table 中读取“ test ”目录的 inode 信息。（第五次）
6．由“ test ”的 inode 中获得其内容所在的 block number，然后读取该 block。（第六次）
7．在“ test ”目录文件中的内容的 block 中得到“ a.txt ”目录的 inode number，然后根据该 inode number 在 inode table 中读取“ a.txt ”文件的 inode 信息。（第七次）

OK!这么短路径的一个文件的打开动作竟然要有 7 次 disk I/O，而且磁头移动方向是来来回回的，不是单一方向的（这样更花时间）。下面的图形象的描述了这个过程。



磁头移动的示意图

上面的七步是极其典型的文件操作的关键步骤，即由对用户友好的路径名方式到对 CPU，磁盘友好的位置方式，几乎所有的接受路径名为参数的系统调用都要有这个步骤。当然在实际的文件系统实现中不可能真的为打开“ /tmp/test/a.txt ”这么个文件而又如此多的读盘操作，因为文件系统的 disk I/O 操作都是建立在 Buffer(缓冲)的基础上的。文件系统驱动并不会直接发 I/O 命令给 IDE Controller,去直接驱动硬件读取磁盘上某一块，而是会通过缓冲管理器。这样很多读盘请求会被缓冲满足，这极大的减少了磁盘操作，性能得到极大提高。

read 解读

还是以“ /tmp/test/a.txt ”为例。在打开该文件，获得其 file descriptor 以后，可以通过 read()来读取其中的内容。

```
char buf[100];
ssize_t nread = read(file, buf, sizeof(buf));
```

在 open()的最后获得了“ /tmp/test/a.txt ”文件的 inode 信息，在期间包含了该文件在数据区的位置。

inode 成员	I_mode	i_uid	i_size	i_time	i_gid	i_nlinks	i_zone
值	0x41e4	0	0x1a	0x46cb9a84	0	0x01	0x8ca

这里文件大小只占一个 block，为 0x8ca。MINIX 文件驱动会读取 block 0x8ca 中的内容。就是“ This is a test text file. ”的字符串。比 open()要简单多了。

rename 解读

比如要修改“ /tmp/test/a.txt ”文件名为“ /tmp/test/b.txt ”，Linux 用户一般用“ mv ”命令来改名。

```
[wzhou@dcmp10 ~]$ mv /tmp/test/a.txt /tmp/test/b.txt
```

如果我们用 strace 来跟踪一下的话，可以看到在这么一行：

```
rename("/tmp/test/a.txt", "/tmp/test/b.txt") = 0
```

也就是真正完成该功能的是系统调用 rename()。

rename()类似于 open()一样完成从路径名到磁盘位置的翻译（上面 7 步中的前 6 步），在第 6 步完成后，可获得 a.txt 在目录“ /tmp/test/ ”中的位置，如下：

```
[root@localhost ~]# hexdump -C -s 0x22f000 -n 1024 /dev/hdd1
0022f000  16 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f020  07 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f040  00 00 2e 61 2e 74 78 74  2e 73 77 70 00 00 00 00  |...a.txt.swp....|
0022f050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f060  18 00 61 2e 74 78 74 00  00 00 00 00 00 00 00 00  |..a.txt.........|   找到“a.txt”文件，其 inode number
0022f070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|   为 0x18
```

OK，这时只要把上面标兰的目录项中的“ a.txt ”字符串改成“ b.txt ”即可。

```
[root@localhost ~]# hexdump -C -s 0x22f000 -n 1024 /dev/hdd1
0022f000  16 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f020  07 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f040  00 00 2e 61 2e 74 78 74  2e 73 77 70 00 00 00 00  |...a.txt.swp....|
0022f050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f060  18 00 62 2e 74 78 74 00  00 00 00 00 00 00 00 00  |..b.txt.........|   变为“b.txt”文件，其 inode number
```

0022f070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................| 不变

就这么简单。当然，由于该例子是在同一目录下改名，所以这么简单。如果把“ /tmp/test/a.txt ”改为“ /tmp/test/ttt/b.txt ”，这要稍微复杂一点。步骤如下：

1．通过路径名翻译成磁盘上的位置得到“ /tmp/test/ ”目录文件中的目录项，得到 a.txt 文件的 inode number，0x18，并把该 inode number 设为 0，这样就把 a.txt 文件给删除了。

```
0022f060  00 00 62 2e 74 78 74 00  00 00 00 00 00 00 00 00  |..a.txt.........|    “ a.txt ”文件，其 inode number
0022f070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|    被设为 0
```

2．通过路径名翻译成磁盘上的位置得到“ /tmp/test/ttt/ ”目录文件中的目录项，在其中添加一个新的目录项

```
0022f860  18 00 63 2e 74 78 74 00  00 00 00 00 00 00 00 00  |..b.txt.........|    “ b.txt ”文件，其 inode number
0022f870  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|    被设为 0x18
```

文件本身没有任何移动，在 rename 前与后该文件的 inode number 都为 0x18，改变的是该文件的目录项。

mkdir 解读

以在“ /tmp/test/ ”目录下建立 c 子目录为例。

步骤如下：

1．因为要建立一个新文件（目录也是文件），所以先要获得一个 inode。通过搜索 inode bitmap 中的第一个非置位的 bit，得到该 bit 相对于 inode bitmap 头部的偏移，此即新分配的 inode number，比如为 0x102，并置位 inode bitmap 中的该位，表示第 0x102 号 inode 已被分配。

2．以 inode number(0x102)为 index，得到 inode table 数组中的 inode 项，此即新文件的 inode 节点。

3．新建文件自然要在数据区占空间，对建立新目录而言，这会分配一个 block。因为新目录即使是空目录，但也有两个目录项“ . ”，“ .. ”。在 zone bitmap 中搜索第一个非置位的 bit，该 bit 相对于 zone bitmap 头部的偏移 + 数据区开始的偏移即是新分配的 block number（相对于整个分区的 block 偏移），比如为 0x10234，并置位 zone bitmap 中的该位，表示第 0x10234 号 block 已被分配。

4．通过路径名翻译成磁盘上的位置得到“ /tmp/test/ ”目录文件中的内容如下：

```
[root@localhost ~]# hexdump -C -s 0x22f000 -n 1024 /dev/hdd1
0022f000  16 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f020  07 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f040  00 00 2e 61 2e 74 78 74  2e 73 77 70 00 00 00 00  |...a.txt.swp....|
0022f050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f060  18 00 61 2e 74 78 74 00  00 00 00 00 00 00 00 00  |..a.txt.........|
0022f070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

5．在“ /tmp/test/ ”目录文件中添加要建立的“ c ”子目录的目录项，如下：

```
0022f000  16 00 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f020  07 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f040  00 00 2e 61 2e 74 78 74  2e 73 77 70 00 00 00 00  |...a.txt.swp....|
0022f050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f060  18 00 61 2e 74 78 74 00  00 00 00 00 00 00 00 00  |..a.txt.........|
0022f070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0022f080  02 01 63 00 00 00 00 00  00 00 00 00 00 00 00 00  |..x.............|   文件名为“ c ”，其 inode number
0022f090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|   为 0x102
```

6．把该目录相关信息写入 inode table 中的第 102 号 inode，如下：

inode 成员	i_mode	i_uid	i_size	i_time	i_gid	i_nlinks	i_zone
值	0x41ed	0	0x60	0x46cbca5f	0	0x02	0x10234

把分配的 block number 写入 inode 中。

7．在 0x10234 block 的数据区写入“ . ”,“ .. ”两个目录项，如下：

```
0408d000  02 01 2e 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0408d010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0408d020  16 00 2e 2e 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0408d030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

“ . ”指向目录“ /tmp/test/c/ ”自身

“ .. ”指向父目录“ /tmp/test/ ”

后记

第一次读Linux OS下MINIX文件系统代码时内核版本还是 2.2，原来做的笔记也是基于 2.2 的。这次为了与时俱进，解析的是 2.6.20 的 [14]。如果单纯为了理解文件系统，可能较低版本的内核比较容易理解，甚至可以看 0.01 版本的Linux内核代码。但学习文件系统，必然要涉及到该文件系统在某个OS下的实现，而从实用的角度看，同样要花时间精力，为什么要花代价在已经成为历史的代码上呢？要响应胡总书记与时俱进的号召呀！

[14] 当前最新的 Linux 内核版本是 2.6.22

试验环境

Linux 2.6.20 kernel + kdb

```
[root@localhost wzhou]# uname -a
Linux localhost.localdomain 2.6.20 #1 SMP Wed Feb 28 08:46:06 CST 2007 i686 i686 i386 GNU/Linux

[root@localhost wzhou]# ~wzhou/src/util-linux-2.12r/disk-utils/mkfs.minix -v
mkfs.minix (util-linux-2.12r)
Usage: mkfs.minix [-c | -l filename] [-nXX] [-iXX] /dev/name [blocks]
```

FUSE 2.6.3

参考资料

1. 《Operation Systems: Design and Implementation(Second Edition)》[15]
2. Linux 2.6.20 源代码
3. util-linux Package (util-linux-2.12r版本)[16]
4. FUSE官方网站 http://fuse.sourceforge.net/

[15] 这是我 9 年前自学操作系统的教材。现在有了第三版，据说最新的 MINIX 操作系统被彻底重写，内核态代码被缩减到几千行，“ 微内核 ” 得非常彻底。如有时间真想读读。
[16] 包含 mkfs.minix 的源代码

附录

2.6.20 内核中 MINIX file system driver 完全注释

linux-2.6.20/fs/minix/minix.h

```
1  #include <linux/fs.h>
2  #include <linux/pagemap.h>
3  #include <linux/minix_fs.h>
4
5  /*
6   * change the define below to 0 if you want names > info->s_namelen chars to be
7   * truncated. Else they will be disallowed (ENAMETOOLONG).
8   */
9  #define NO_TRUNCATE 1
10
11 #define INODE_VERSION(inode)  minix_sb(inode->i_sb)->s_version
12
13 #define MINIX_V1    0x0001    /* original minix fs */
14 #define MINIX_V2    0x0002    /* minix V2 fs */
15
```

```
16 /*
17  * minix fs inode data in memory
18  */
19 struct minix_inode_info {
20     union {
21         __u16 i1_data[16];
22         __u32 i2_data[16];
23     } u;
24     struct inode vfs_inode;
25 };
26
27 /*
28  * minix super-block data in memory
29  */
30 struct minix_sb_info {
31     unsigned long s_ninodes;          该文件系统节点数
32     unsigned long s_nzones;
33     unsigned long s_imap_blocks;      inode bitmap 的 block 数
34     unsigned long s_zmap_blocks;      zone bitmap 的 block 数
35     unsigned long s_firstdatazone;    数据区开始的 block number
36     unsigned long s_log_zone_size;
37     unsigned long s_max_size;         允许最大文件长度
38     int s_dirsize;                    目录项长度
39     int s_namelen;                    文件名长度
40     int s_link_max;
41     struct buffer_head ** s_imap;     指向已经读入内存的 inode bitmap
```

```
42      struct buffer_head ** s_zmap;     指向已经读入内存的 zone bitmap
43      struct buffer_head * s_sbh;
44      struct minix_super_block * s_ms;
45      unsigned short s_mount_state;
46      unsigned short s_version;
47  };
48
49  extern struct minix_inode * minix_V1_raw_inode(struct super_block *, ino_t, struct buffer_head **);
50  extern struct minix2_inode * minix_V2_raw_inode(struct super_block *, ino_t, struct buffer_head **);
51  extern struct inode * minix_new_inode(const struct inode * dir, int * error);
52  extern void minix_free_inode(struct inode * inode);
53  extern unsigned long minix_count_free_inodes(struct minix_sb_info *sbi);
54  extern int minix_new_block(struct inode * inode);
55  extern void minix_free_block(struct inode * inode, int block);
56  extern unsigned long minix_count_free_blocks(struct minix_sb_info *sbi);
57
58  extern int minix_getattr(struct vfsmount *, struct dentry *, struct kstat *);
59
60  extern void V2_minix_truncate(struct inode *);
61  extern void V1_minix_truncate(struct inode *);
62  extern void V2_minix_truncate(struct inode *);
63  extern void minix_truncate(struct inode *);
64  extern int minix_sync_inode(struct inode *);
65  extern void minix_set_inode(struct inode *, dev_t);
66  extern int V1_minix_get_block(struct inode *, long, struct buffer_head *, int);
67  extern int V2_minix_get_block(struct inode *, long, struct buffer_head *, int);
```

```
68 extern unsigned V1_minix_blocks(loff_t);
69 extern unsigned V2_minix_blocks(loff_t);
70
71 extern struct minix_dir_entry *minix_find_entry(struct dentry*, struct page**);
72 extern int minix_add_link(struct dentry*, struct inode*);
73 extern int minix_delete_entry(struct minix_dir_entry*, struct page*);
74 extern int minix_make_empty(struct inode*, struct inode*);
75 extern int minix_empty_dir(struct inode*);
76 extern void minix_set_link(struct minix_dir_entry*, struct page*, struct inode*);
77 extern struct minix_dir_entry *minix_dotdot(struct inode*, struct page**);
78 extern ino_t minix_inode_by_name(struct dentry*);
79
80 extern int minix_sync_file(struct file *, struct dentry *, int);
81
82 extern struct inode_operations minix_file_inode_operations;
83 extern struct inode_operations minix_dir_inode_operations;
84 extern const struct file_operations minix_file_operations;
85 extern const struct file_operations minix_dir_operations;
86 extern struct dentry_operations minix_dentry_operations;
87
88 static inline struct minix_sb_info *minix_sb(struct super_block *sb)
89 {
90     return sb->s_fs_info;
91 }
92
93 static inline struct minix_inode_info *minix_i(struct inode *inode)
```

```
94 {
95     return list_entry(inode, struct minix_inode_info, vfs_inode);
96 }
```

linux-2.6.20/include/linux/minix_fs.h

```
1  #ifndef _LINUX_MINIX_FS_H
2  #define _LINUX_MINIX_FS_H
3
4  #include <linux/magic.h>
5
6  /*
7   * The minix filesystem constants/structures
8   */
9
10 /*
11  * Thanks to Kees J Bot for sending me the definitions of the new
12  * minix filesystem (aka V2) with bigger inodes and 32-bit block
13  * pointers.
14  */
15
16 #define MINIX_ROOT_INO 1
17
```

```
18 /* Not the same as the bogus LINK_MAX in <linux/limits.h>. Oh well. */
19 #define MINIX_LINK_MAX  250
20 #define MINIX2_LINK_MAX 65530
21
22 #define MINIX_I_MAP_SLOTS  8
23 #define MINIX_Z_MAP_SLOTS  64
24 #define MINIX_VALID_FS     0x0001    /* Clean fs. */
25 #define MINIX_ERROR_FS     0x0002    /* fs has errors. */
26
27 #define MINIX_INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct minix_inode)))
28 #define MINIX2_INODES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct minix2_inode)))
29
30 /*
31  * This is the original minix inode layout on disk.
32  * Note the 8-bit gid and atime and ctime.
33  */
    原始版的 MINIX 文件系统的 inode
34 struct minix_inode {
35     __u16 i_mode;
36     __u16 i_uid;
37     __u32 i_size;
38     __u32 i_time;
39     __u8  i_gid;
40     __u8  i_nlinks;
41     __u16 i_zone[9];    0-6 为直接块，7 为一次间接块，8 为二次间接块
42 };
```

```
43
44 /*
45  * The new minix inode has all the time entries, as well as
46  * long block numbers and a third indirect block (7+1+1+1
47  * instead of 7+1+1). Also, some previously 8-bit values are
48  * now 16-bit. The inode is now 64 bytes instead of 32.
49  */
```

升级版的 MINIX 文件系统的 inode

```
50 struct minix2_inode {
51     __u16 i_mode;
52     __u16 i_nlinks;
53     __u16 i_uid;
54     __u16 i_gid;
55     __u32 i_size;
56     __u32 i_atime;      access time
57     __u32 i_mtime;      modification time
58     __u32 i_ctime;      create time
59     __u32 i_zone[10];   0-6 为直接块，7 为一次间接块，8 为二次间接块，9 是三次间接块
60 };
61
62 /*
63  * minix super-block data on disk
64  */
```

这是 minix file system super block 在磁盘上的内容。

```
65 struct minix_super_block {
66     __u16 s_ninodes;
```

```
67      __u16 s_nzones;
68      __u16 s_imap_blocks;
69      __u16 s_zmap_blocks;
70      __u16 s_firstdatazone;
71      __u16 s_log_zone_size;
72      __u32 s_max_size;
73      __u16 s_magic;
74      __u16 s_state;
75      __u32 s_zones;
76  };
77
78  struct minix_dir_entry {
79      __u16 inode;
80      char name[0];
81  };
```

MINIX 文件系统的目录项非常简单，只有该目录项的名字与其所在的 inode number。原始的 MINIX 文件系统只支持文件名为 14 个字符，升级版的支持 30 个字符。

```
82
83  #endif
```

linux-2.6.20/fs/minix/inode.c

```
1   /*
```

```
2    *  linux/fs/minix/inode.c
3    *
4    *  Copyright (C) 1991, 1992  Linus Torvalds
5    *
6    *  Copyright (C) 1996  Gertjan van Wingerde   (gertjan@cs.vu.nl)
7    *  Minix V2 fs support.
8    *
9    *  Modified for 680x0 by Andreas Schwab
10   */
11
12  #include <linux/module.h>
13  #include "minix.h"
14  #include <linux/buffer_head.h>
15  #include <linux/slab.h>
16  #include <linux/init.h>
17  #include <linux/highuid.h>
18  #include <linux/vfs.h>
19
20  static void minix_read_inode(struct inode * inode);
21  static int minix_write_inode(struct inode * inode, int wait);
22  static int minix_statfs(struct dentry *dentry, struct kstatfs *buf);
23  static int minix_remount (struct super_block * sb, int * flags, char * data);
24
25  static void minix_delete_inode(struct inode *inode)
26  {
27     truncate_inode_pages(&inode->i_data, 0);
```

```
28      inode->i_size = 0;
29      minix_truncate(inode);
30      minix_free_inode(inode);
31  }
32
33  static void minix_put_super(struct super_block *sb)
34  {
35      int i;
36      struct minix_sb_info *sbi = minix_sb(sb);
37
38      if (!(sb->s_flags & MS_RDONLY)) {
39          sbi->s_ms->s_state = sbi->s_mount_state;
40          mark_buffer_dirty(sbi->s_sbh);
41      }
42      for (i = 0; i < sbi->s_imap_blocks; i++)
43          brelse(sbi->s_imap[i]);
44      for (i = 0; i < sbi->s_zmap_blocks; i++)
45          brelse(sbi->s_zmap[i]);
46      brelse (sbi->s_sbh);
47      kfree(sbi->s_imap);
48      sb->s_fs_info = NULL;
49      kfree(sbi);
50
51      return;
52  }
53
```

```
54 static struct kmem_cache * minix_inode_cachep;
55
56 static struct inode *minix_alloc_inode(struct super_block *sb)
57 {
58     struct minix_inode_info *ei;
59     ei = (struct minix_inode_info *)kmem_cache_alloc(minix_inode_cachep, GFP_KERNEL);
60     if (!ei)
61         return NULL;
62     return &ei->vfs_inode;
63 }
64
65 static void minix_destroy_inode(struct inode *inode)
66 {
67     kmem_cache_free(minix_inode_cachep, minix_i(inode));
68 }
69
70 static void init_once(void * foo, struct kmem_cache * cachep, unsigned long flags)
71 {
72     struct minix_inode_info *ei = (struct minix_inode_info *) foo;
73
74     if ((flags & (SLAB_CTOR_VERIFY|SLAB_CTOR_CONSTRUCTOR)) ==
75         SLAB_CTOR_CONSTRUCTOR)
76         inode_init_once(&ei->vfs_inode);
77 }
78
79 static int init_inodecache(void)
```

```
80 {
81     minix_inode_cachep = kmem_cache_create("minix_inode_cache",
82                       sizeof(struct minix_inode_info),
83                       0, (SLAB_RECLAIM_ACCOUNT|
84                      SLAB_MEM_SPREAD),
85                       init_once, NULL);
86     if (minix_inode_cachep == NULL)
87         return -ENOMEM;
88     return 0;
89 }
90
91 static void destroy_inodecache(void)
92 {
93     kmem_cache_destroy(minix_inode_cachep);
94 }
95
96 static struct super_operations minix_sops = {
97     .alloc_inode  = minix_alloc_inode,
98     .destroy_inode   = minix_destroy_inode,
99     .read_inode   = minix_read_inode,
100    .write_inode  = minix_write_inode,
101    .delete_inode= minix_delete_inode,
102    .put_super    = minix_put_super,
103    .statfs       = minix_statfs,
104    .remount_fs   = minix_remount,
105};
```

```
106
107 static int minix_remount (struct super_block * sb, int * flags, char * data)
108 {
109     struct minix_sb_info * sbi = minix_sb(sb);
110     struct minix_super_block * ms;
111
112     ms = sbi->s_ms;
113     if ((*flags & MS_RDONLY) == (sb->s_flags & MS_RDONLY))
114         return 0;
115     if (*flags & MS_RDONLY) {
116         if (ms->s_state & MINIX_VALID_FS ||
117             !(sbi->s_mount_state & MINIX_VALID_FS))
118             return 0;
119         /* Mounting a rw partition read-only. */
120         ms->s_state = sbi->s_mount_state;
121         mark_buffer_dirty(sbi->s_sbh);
122     } else {
123         /* Mount a partition which is read-only, read-write. */
124         sbi->s_mount_state = ms->s_state;
125         ms->s_state &= ~MINIX_VALID_FS;
126         mark_buffer_dirty(sbi->s_sbh);
127
128         if (!(sbi->s_mount_state & MINIX_VALID_FS))
129             printk("MINIX-fs warning: remounting unchecked fs, "
130                 "running fsck is recommended\n");
131         else if ((sbi->s_mount_state & MINIX_ERROR_FS))
```

```
132          printk("MINIX-fs warning: remounting fs with errors, "
133              "running fsck is recommended\n");
134    }
135    return 0;
136 }
137
```

下面的函数通过读取分区的第 1 个 block(第 0 个 block 后面的那个)来初始化内存中的 super block 结构。

本函数一般在 mount 某个 MINIX 分区时才会被调用到。

该函数的主要工作大致如下：

1. 通过 bh = sb_bread(s, 1)，读取要 mount 的分区的 block 1。

2. Block 1 中为 super block，然后依次从 block 2 开始读起。假设从 super block 得知 inode bitmap 的 block 数为 2，zone bitmap 的 block 数为 4，则先读取 block 2，block 3 这两块 block，在 sbi->s_imap[0]，sbi->s_imap[1]中记录了这两个 inode bitmap 的 block 内容。

3. 而后读出 block 4,5,6,7，由 sbi->z_zmap[0], sbi->z_zmap[1], sbi->z_zmap[2], sbi->z_zmap[3]指向 zone bitmap 的 block 内容。

4. 读取根节点，一个文件系统的 root 总是 hard-coding 的，MINIX 文件系统的 root inode number 为 1。

```
138 static int minix_fill_super(struct super_block *s, void *data, int silent)
139 {
140    struct buffer_head *bh;
141    struct buffer_head **map;
142    struct minix_super_block *ms;
143    int i, block;
144    struct inode *root_inode;
145    struct minix_sb_info *sbi;
146
```

```
147    sbi = kzalloc(sizeof(struct minix_sb_info), GFP_KERNEL);
```
分配内存中的 minix super block 空间，并初始化为 0。
```
148    if (!sbi)
149        return -ENOMEM;
150    s->s_fs_info = sbi;
151
152    BUILD_BUG_ON(32 != sizeof (struct minix_inode));
153    BUILD_BUG_ON(64 != sizeof(struct minix2_inode));
154
155    if (!sb_set_blocksize(s, BLOCK_SIZE))
```
设置 minix file system driver 中的 block 大小为 1K。
```
156        goto out_bad_hblock;
157
158    if (!(bh = sb_bread(s, 1)))
```
读取该分区的第 1 个 block。读取出来的数据由 bh->b_data 指向。
```
159        goto out_bad_sb;
160
161    ms = (struct minix_super_block *) bh->b_data;
```

用 minix file system super block 中的信息来初始化通用的 super block 结构。
```
162    sbi->s_ms = ms;
163    sbi->s_sbh = bh;
164    sbi->s_mount_state = ms->s_state;
165    sbi->s_ninodes = ms->s_ninodes;
166    sbi->s_nzones = ms->s_nzones;
167    sbi->s_imap_blocks = ms->s_imap_blocks;
```

```
168    sbi->s_zmap_blocks = ms->s_zmap_blocks;
169    sbi->s_firstdatazone = ms->s_firstdatazone;
170    sbi->s_log_zone_size = ms->s_log_zone_size;
171    sbi->s_max_size = ms->s_max_size;
172    s->s_magic = ms->s_magic;
```

判断是否 MINIX file system。目前 MINIX file system 有四种签名。如下：

```
#define MINIX_SUPER_MAGIC  0x137F    /* original minix fs */
#define MINIX_SUPER_MAGIC2 0x138F    /* minix fs, 30 char names */
#define MINIX2_SUPER_MAGIC 0x2468    /* minix V2 fs */
#define MINIX2_SUPER_MAGIC2   0x2478    /* minix V2 fs, 30 char names */
```

每种文件系统的目录项与允许的文件名长度有不同。

```
173    if (s->s_magic == MINIX_SUPER_MAGIC) {
174        sbi->s_version = MINIX_V1;
175        sbi->s_dirsize = 16;
176        sbi->s_namelen = 14;        文件名长度为 14 个字符
177        sbi->s_link_max = MINIX_LINK_MAX;
178    } else if (s->s_magic == MINIX_SUPER_MAGIC2) {
179        sbi->s_version = MINIX_V1;
180        sbi->s_dirsize = 32;
181        sbi->s_namelen = 30;        文件名长度为 30 个字符
182        sbi->s_link_max = MINIX_LINK_MAX;
183    } else if (s->s_magic == MINIX2_SUPER_MAGIC) {
184        sbi->s_version = MINIX_V2;
```

```
185         sbi->s_nzones = ms->s_zones;
186         sbi->s_dirsize = 16;
187         sbi->s_namelen = 14;
188         sbi->s_link_max = MINIX2_LINK_MAX;
189     } else if (s->s_magic == MINIX2_SUPER_MAGIC2) {
190         sbi->s_version = MINIX_V2;
191         sbi->s_nzones = ms->s_zones;
192         sbi->s_dirsize = 32;
193         sbi->s_namelen = 30;
194         sbi->s_link_max = MINIX2_LINK_MAX;
195     } else
196         goto out_no_fs;
197
198     /*
199      * Allocate the buffer map to keep the superblock small.
200      */
```

下面是从磁盘上读取 inode bitmap 与 zone bitmap。

```
201     if (sbi->s_imap_blocks == 0 || sbi->s_zmap_blocks == 0)
202         goto out_illegal_sb;
203     i = (sbi->s_imap_blocks + sbi->s_zmap_blocks) * sizeof(bh);
204     map = kzalloc(i, GFP_KERNEL);
205     if (!map)
206         goto out_no_map;
207     sbi->s_imap = &map[0];
208     sbi->s_zmap = &map[sbi->s_imap_blocks];
209
```

从“MINIX 文件系统的总体磁盘布局”图可知 inode bitmap 所在的位置从 block 2 开始。

```
210    block=2;
```

读取 inode bitmap。

```
211    for (i=0 ; i < sbi->s_imap_blocks ; i++) {
212        if (!(sbi->s_imap[i]=sb_bread(s, block)))
213            goto out_no_bitmap;
214        block++;
215    }
```

读取 zone bitmap。

```
216    for (i=0 ; i < sbi->s_zmap_blocks ; i++) {
217        if (!(sbi->s_zmap[i]=sb_bread(s, block)))
218            goto out_no_bitmap;
219        block++;
220    }
221
```

inode 0 总是被占用的。同样 zone 1 也总是被占用的。

```
222    minix_set_bit(0,sbi->s_imap[0]->b_data);
223    minix_set_bit(0,sbi->s_zmap[0]->b_data);
224
225    /* set up enough so that it can read an inode */
```

设置 MINIX 文件系统的 super block 的操作函数集。

```
226    s->s_op = &minix_sops;
```

```
#define MINIX_ROOT_INO 1
```

MINIX 文件系统的 root inode 节点号是 1。

```
227     root_inode = iget(s, MINIX_ROOT_INO);
```

iget（）用于从 inode number 获得其所指向的 indoe 信息。

由 ino(inode number)为 index 从 inode 数组(inode table)中得到 inode。

```
static inline struct inode *iget(struct super_block *sb, unsigned long ino)
{
    struct inode *inode = iget_locked(sb, ino);

    if (inode && (inode->i_state & I_NEW)) {
        sb->s_op->read_inode(inode);
        unlock_new_inode(inode);
    }

    return inode;
}
```

上面标红的代码行

```
sb->s_op->read_inode(inode);
```

中 sb-> s_op 也即是上面 226 行的 minix_sops。这里通过调用 minix_read_inode（）函数来获得 MINIX file system 的 root(“/”)的 inode 信息。

```
228     if (!root_inode || is_bad_inode(root_inode))
229         goto out_no_root;
230
231     s->s_root = d_alloc_root(root_inode);
```

分配 root 目录的 directory entry(dentry)。

```
232     if (!s->s_root)
```

```
233         goto out_iput;
234 
235     if (!NO_TRUNCATE)
236         s->s_root->d_op = &minix_dentry_operations;
237 
238     if (!(s->s_flags & MS_RDONLY)) {
239         ms->s_state &= ~MINIX_VALID_FS;
240         mark_buffer_dirty(bh);
241     }
242     if (!(sbi->s_mount_state & MINIX_VALID_FS))
243         printk("MINIX-fs: mounting unchecked file system, "
244             "running fsck is recommended\n");
245     else if (sbi->s_mount_state & MINIX_ERROR_FS)
246         printk("MINIX-fs: mounting file system with errors, "
247             "running fsck is recommended\n");
248     return 0;
249 
250 out_iput:
251     iput(root_inode);
252     goto out_freemap;
253 
254 out_no_root:
255     if (!silent)
256         printk("MINIX-fs: get root inode failed\n");
257     goto out_freemap;
258 
```

```
259 out_no_bitmap:
260    printk("MINIX-fs: bad superblock or unable to read bitmaps\n");
261 out_freemap:
262    for (i = 0; i < sbi->s_imap_blocks; i++)
263       brelse(sbi->s_imap[i]);
264    for (i = 0; i < sbi->s_zmap_blocks; i++)
265       brelse(sbi->s_zmap[i]);
266    kfree(sbi->s_imap);
267    goto out_release;
268
269 out_no_map:
270    if (!silent)
271       printk("MINIX-fs: can't allocate map\n");
272    goto out_release;
273
274 out_illegal_sb:
275    if (!silent)
276       printk("MINIX-fs: bad superblock\n");
277    goto out_release;
278
279 out_no_fs:
280    if (!silent)
281       printk("VFS: Can't find a Minix or Minix V2 filesystem "
282          "on device %s\n", s->s_id);
283 out_release:
284    brelse(bh);
```

```
285     goto out;
286
287 out_bad_hblock:
288     printk("MINIX-fs: blocksize too small for device\n");
289     goto out;
290
291 out_bad_sb:
292     printk("MINIX-fs: unable to read superblock\n");
293 out:
294     s->s_fs_info = NULL;
295     kfree(sbi);
296     return -EINVAL;
297 }
298
```

下面的函数返回与 MINIX 文件系统相关的信息，比如，文件系统的类型（签名），块大小，该文件系统的总块数，该文件系统的当前空闲块数，等等。

```
299 static int minix_statfs(struct dentry *dentry, struct kstatfs *buf)
300 {
301     struct minix_sb_info *sbi = minix_sb(dentry->d_sb);
302     buf->f_type = dentry->d_sb->s_magic;
303     buf->f_bsize = dentry->d_sb->s_blocksize;
304     buf->f_blocks = (sbi->s_nzones - sbi->s_firstdatazone) << sbi->s_log_zone_size;
305     buf->f_bfree = minix_count_free_blocks(sbi);
306     buf->f_bavail = buf->f_bfree;
307     buf->f_files = sbi->s_ninodes;
308     buf->f_ffree = minix_count_free_inodes(sbi);
```

```
309    buf->f_namelen = sbi->s_namelen;
310    return 0;
311 }
312
313 static int minix_get_block(struct inode *inode, sector_t block,
314           struct buffer_head *bh_result, int create)
315 {
316    if (INODE_VERSION(inode) == MINIX_V1)
317       return V1_minix_get_block(inode, block, bh_result, create);
318    else
319       return V2_minix_get_block(inode, block, bh_result, create);
320 }
321
322 static int minix_writepage(struct page *page, struct writeback_control *wbc)
323 {
324    return block_write_full_page(page, minix_get_block, wbc);
325 }
326 static int minix_readpage(struct file *file, struct page *page)
327 {
328    return block_read_full_page(page,minix_get_block);
329 }
330 static int minix_prepare_write(struct file *file, struct page *page, unsigned from, unsigned to)
331 {
332    return block_prepare_write(page,from,to,minix_get_block);
333 }
334 static sector_t minix_bmap(struct address_space *mapping, sector_t block)
```

```
335 {
336    return generic_block_bmap(mapping,block,minix_get_block);
337 }
338 static const struct address_space_operations minix_aops = {
339    .readpage = minix_readpage,
340    .writepage = minix_writepage,
341    .sync_page = block_sync_page,
342    .prepare_write = minix_prepare_write,
343    .commit_write = generic_commit_write,
344    .bmap = minix_bmap
345 };
346
347 static struct inode_operations minix_symlink_inode_operations = {
348    .readlink = generic_readlink,
349    .follow_link = page_follow_link_light,
350    .put_link = page_put_link,
351    .getattr  = minix_getattr,
352 };
353
354 void minix_set_inode(struct inode *inode, dev_t rdev)
355 {
356    if (S_ISREG(inode->i_mode)) {
357       inode->i_op = &minix_file_inode_operations;
358       inode->i_fop = &minix_file_operations;
359       inode->i_mapping->a_ops = &minix_aops;
360    } else if (S_ISDIR(inode->i_mode)) {
```

```
361         inode->i_op = &minix_dir_inode_operations;
362         inode->i_fop = &minix_dir_operations;
363         inode->i_mapping->a_ops = &minix_aops;
364     } else if (S_ISLNK(inode->i_mode)) {
365         inode->i_op = &minix_symlink_inode_operations;
366         inode->i_mapping->a_ops = &minix_aops;
367     } else
368         init_special_inode(inode, inode->i_mode, rdev);
369 }
370
371 /*
372  * The minix V1 function to read an inode.
373  */
```

当读取文件时会调用到本函数。

```
374 static void V1_minix_read_inode(struct inode * inode)
375 {
376     struct buffer_head * bh;
377     struct minix_inode * raw_inode;
378     struct minix_inode_info *minix_inode = minix_i(inode);
379     int i;
380
```

先从磁盘上读取编号为 inode->i_ino 的 inode。

```
381     raw_inode = minix_V1_raw_inode(inode->i_sb, inode->i_ino, &bh);
382     if (!raw_inode) {
383         make_bad_inode(inode);
384         return;
```

```
385     }
```

用磁盘上读出的信息来填充 inode 节点。

```
386     inode->i_mode = raw_inode->i_mode;
387     inode->i_uid = (uid_t)raw_inode->i_uid;
388     inode->i_gid = (gid_t)raw_inode->i_gid;
389     inode->i_nlink = raw_inode->i_nlinks;
390     inode->i_size = raw_inode->i_size;
391     inode->i_mtime.tv_sec = inode->i_atime.tv_sec = inode->i_ctime.tv_sec = raw_inode->i_time;
392     inode->i_mtime.tv_nsec = 0;
393     inode->i_atime.tv_nsec = 0;
394     inode->i_ctime.tv_nsec = 0;
395     inode->i_blocks = 0;
396     for (i = 0; i < 9; i++)
397         minix_inode->u.i1_data[i] = raw_inode->i_zone[i];
398     minix_set_inode(inode, old_decode_dev(raw_inode->i_zone[0]));
399     brelse(bh);
400 }
401
402 /*
403  * The minix V2 function to read an inode.
404  */
405 static void V2_minix_read_inode(struct inode * inode)
406 {
407     struct buffer_head * bh;
408     struct minix2_inode * raw_inode;
409     struct minix_inode_info *minix_inode = minix_i(inode);
```

```
410    int i;
411
412    raw_inode = minix_V2_raw_inode(inode->i_sb, inode->i_ino, &bh);
413    if (!raw_inode) {
414       make_bad_inode(inode);
415       return;
416    }
417    inode->i_mode = raw_inode->i_mode;
418    inode->i_uid = (uid_t)raw_inode->i_uid;
419    inode->i_gid = (gid_t)raw_inode->i_gid;
420    inode->i_nlink = raw_inode->i_nlinks;
421    inode->i_size = raw_inode->i_size;
422    inode->i_mtime.tv_sec = raw_inode->i_mtime;
423    inode->i_atime.tv_sec = raw_inode->i_atime;
424    inode->i_ctime.tv_sec = raw_inode->i_ctime;
425    inode->i_mtime.tv_nsec = 0;
426    inode->i_atime.tv_nsec = 0;
427    inode->i_ctime.tv_nsec = 0;
428    inode->i_blocks = 0;
429    for (i = 0; i < 10; i++)
430       minix_inode->u.i2_data[i] = raw_inode->i_zone[i];
431    minix_set_inode(inode, old_decode_dev(raw_inode->i_zone[0]));
432    brelse(bh);
433 }
434
435 /*
```

```
436 * The global function to read an inode.
437 */
```

根据不同版本从 inode table 中读出 inode 信息来初始化传入的 inode。

```
438 static void minix_read_inode(struct inode * inode)
439 {
440     if (INODE_VERSION(inode) == MINIX_V1)
441         V1_minix_read_inode(inode);
442     else
443         V2_minix_read_inode(inode);
444 }
445
446 /*
447 * The minix V1 function to synchronize an inode.
448 */
449 static struct buffer_head * V1_minix_update_inode(struct inode * inode)
450 {
451     struct buffer_head * bh;
452     struct minix_inode * raw_inode;
453     struct minix_inode_info *minix_inode = minix_i(inode);
454     int i;
455
456     raw_inode = minix_V1_raw_inode(inode->i_sb, inode->i_ino, &bh);
457     if (!raw_inode)
458         return NULL;
459     raw_inode->i_mode = inode->i_mode;
460     raw_inode->i_uid = fs_high2lowuid(inode->i_uid);
```

```
461    raw_inode->i_gid = fs_high2lowgid(inode->i_gid);
462    raw_inode->i_nlinks = inode->i_nlink;
463    raw_inode->i_size = inode->i_size;
464    raw_inode->i_time = inode->i_mtime.tv_sec;
465    if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode))
466       raw_inode->i_zone[0] = old_encode_dev(inode->i_rdev);
467    else for (i = 0; i < 9; i++)
468       raw_inode->i_zone[i] = minix_inode->u.i1_data[i];
469    mark_buffer_dirty(bh);
470    return bh;
471 }
472
473 /*
474  * The minix V2 function to synchronize an inode.
475  */
476 static struct buffer_head * V2_minix_update_inode(struct inode * inode)
477 {
478    struct buffer_head * bh;
479    struct minix2_inode * raw_inode;
480    struct minix_inode_info *minix_inode = minix_i(inode);
481    int i;
482
483    raw_inode = minix_V2_raw_inode(inode->i_sb, inode->i_ino, &bh);
484    if (!raw_inode)
485       return NULL;
486    raw_inode->i_mode = inode->i_mode;
```

```
487     raw_inode->i_uid = fs_high2lowuid(inode->i_uid);
488     raw_inode->i_gid = fs_high2lowgid(inode->i_gid);
489     raw_inode->i_nlinks = inode->i_nlink;
490     raw_inode->i_size = inode->i_size;
491     raw_inode->i_mtime = inode->i_mtime.tv_sec;
492     raw_inode->i_atime = inode->i_atime.tv_sec;
493     raw_inode->i_ctime = inode->i_ctime.tv_sec;
494     if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode))
495        raw_inode->i_zone[0] = old_encode_dev(inode->i_rdev);
496     else for (i = 0; i < 10; i++)
497        raw_inode->i_zone[i] = minix_inode->u.i2_data[i];
498     mark_buffer_dirty(bh);
499     return bh;
500 }
501
502 static struct buffer_head *minix_update_inode(struct inode *inode)
503 {
504     if (INODE_VERSION(inode) == MINIX_V1)
505        return V1_minix_update_inode(inode);
506     else
507        return V2_minix_update_inode(inode);
508 }
509
510 static int minix_write_inode(struct inode * inode, int wait)
511 {
512     brelse(minix_update_inode(inode));
```

```
513    return 0;
514 }
515
516 int minix_sync_inode(struct inode * inode)
517 {
518    int err = 0;
519    struct buffer_head *bh;
520
521    bh = minix_update_inode(inode);
522    if (bh && buffer_dirty(bh))
523    {
524       sync_dirty_buffer(bh);
525       if (buffer_req(bh) && !buffer_uptodate(bh))
526       {
527          printk("IO error syncing minix inode [%s:%08lx]\n",
528             inode->i_sb->s_id, inode->i_ino);
529          err = -1;
530       }
531    }
532    else if (!bh)
533       err = -1;
534    brelse (bh);
535    return err;
536 }
537
538 int minix_getattr(struct vfsmount *mnt, struct dentry *dentry, struct kstat *stat)
```

```
539 {
540     generic_fillattr(dentry->d_inode, stat);
541     if (INODE_VERSION(dentry->d_inode) == MINIX_V1)
542         stat->blocks = (BLOCK_SIZE / 512) * V1_minix_blocks(stat->size);
543     else
544         stat->blocks = (BLOCK_SIZE / 512) * V2_minix_blocks(stat->size);
545     stat->blksize = BLOCK_SIZE;
546     return 0;
547 }
548
549 /*
550  * The function that is called for file truncation.
551  */
```

截断文件。

```
552 void minix_truncate(struct inode * inode)
553 {
554     if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) || S_ISLNK(inode->i_mode)))
555         return;
```

不是普通文件或是目录或是 link，都不能“截断”。

```
556     if (INODE_VERSION(inode) == MINIX_V1)
557         V1_minix_truncate(inode);
558     else
559         V2_minix_truncate(inode);
560 }
561
562 static int minix_get_sb(struct file_system_type *fs_type,
```

```
563    int flags, const char *dev_name, void *data, struct vfsmount *mnt)
564 {
565    return get_sb_bdev(fs_type, flags, dev_name, data, minix_fill_super,
566             mnt);
567 }
```

上面函数中的参数 minix_fill_super 是个 callback function，其是 minix file system driver 的鉴别器。在该函数中通过读取要鉴别的分区的第 1 个 block(注意，分区的 block 编号从 0 开始)，然后分析上面的数据来实现鉴别。具体见对该函数的注释。

```
568
569 static struct file_system_type minix_fs_type = {
570    .owner    = THIS_MODULE,
571    .name     = "minix",
572    .get_sb      = minix_get_sb,
573    .kill_sb  = kill_block_super,
574    .fs_flags = FS_REQUIRES_DEV,
575 };
```

上面结构中的 minix_get_sb 是 minix file system 的辨认函数。内核在 mount 某个分区是，会依次调用注册在 **file_systems** 链表上的 file_system_type 结构的 get_sb 接口，如果其返回 >= 0，则表示与该 file_system_type 结构相关的文件系统 driver 认识该分区。见 linux-2.6.20/fs/super.c 中的如下函数

```
vfs_kern_mount(struct file_system_type *type, int flags, const char *name, void *data)
{
   ......
   error = type->get_sb(type, flags, name, data, mnt);
   if (error < 0)
      goto out_free_secdata;
```

```
    ......

out_free_secdata:
    free_secdata(secdata);
out_mnt:
    free_vfsmnt(mnt);
out:
    return ERR_PTR(error);
}

576
577 static int __init init_minix_fs(void)
578 {
579     int err = init_inodecache();
```

见本文件 line 79 该函数的定义。用内核的 slab 内存分配器来为 inode 建立名字为“ minix_inode_cache ”的快速 cache。

```
580     if (err)
581         goto out1;
582     err = register_filesystem(&minix_fs_type);
```

向内核注册 minix 文件系统 driver。内核中凡是 mount 的分区的文件系统 driver 都必须通过该函数挂接到全局变量 **file_systems** 所在的链表中。

```
    linux-2.6.20/fs/filesystems.c,line 31

    static struct file_system_type *file_systems;

583     if (err)
```

```
584         goto out;
585     return 0;
586 out:
587     destroy_inodecache();
588 out1:
589     return err;
590 }
591
592 static void __exit exit_minix_fs(void)
593 {
594     unregister_filesystem(&minix_fs_type);
595     destroy_inodecache();
    做 init_minix_fs () 中相反的工作。
596 }
597
598 module_init(init_minix_fs)
599 module_exit(exit_minix_fs)
600 MODULE_LICENSE("GPL");
601
```

linux-2.6.20/fs/minix/file.c

```
1   /*
2    *  linux/fs/minix/file.c
```

```
3    *
4    *  Copyright (C) 1991, 1992 Linus Torvalds
5    *
6    *  minix regular file handling primitives
7    */
8
9   #include <linux/buffer_head.h>       /* for fsync_inode_buffers() */
10  #include "minix.h"
11
12  /*
13   * We have mostly NULLs here: the current defaults are OK for
14   * the minix filesystem.
15   */
16  int minix_sync_file(struct file *, struct dentry *, int);
17
18  const struct file_operations minix_file_operations = {
19      .llseek       = generic_file_llseek,
20      .read     = do_sync_read,
21      .aio_read = generic_file_aio_read,
22      .write    = do_sync_write,
23      .aio_write    = generic_file_aio_write,
24      .mmap     = generic_file_mmap,
25      .fsync    = minix_sync_file,
26      .sendfile = generic_file_sendfile,
27  };
28
```

```
29 struct inode_operations minix_file_inode_operations = {
30     .truncate = minix_truncate,
31     .getattr  = minix_getattr,
32 };
33
34 int minix_sync_file(struct file * file, struct dentry *dentry, int datasync)
35 {
36     struct inode *inode = dentry->d_inode;
37     int err;
38
39     err = sync_mapping_buffers(inode->i_mapping);
40     if (!(inode->i_state & I_DIRTY))
41        return err;
42     if (datasync && !(inode->i_state & I_DIRTY_DATASYNC))
43        return err;
44
45     err |= minix_sync_inode(inode);
46     return err ? -EIO : 0;
47 }
```

linux-2.6.20/fs/minix/namei.c

```
1  /*
2   *  linux/fs/minix/namei.c
```

```
 *
 *  Copyright (C) 1991, 1992  Linus Torvalds
 */

#include "minix.h"

static int add_nondir(struct dentry *dentry, struct inode *inode)
{
    int err = minix_add_link(dentry, inode);
    if (!err) {
        d_instantiate(dentry, inode);
        return 0;
    }
    inode_dec_link_count(inode);
    iput(inode);
    return err;
}

static int minix_hash(struct dentry *dentry, struct qstr *qstr)
{
    unsigned long hash;
    int i;
    const unsigned char *name;

    i = minix_sb(dentry->d_inode->i_sb)->s_namelen;
    if (i >= qstr->len)
```

```
29          return 0;
30      /* Truncate the name in place, avoids having to define a compare
31         function. */
32      qstr->len = i;
33      name = qstr->name;
34      hash = init_name_hash();
35      while (i--)
36          hash = partial_name_hash(*name++, hash);
37      qstr->hash = end_name_hash(hash);
38      return 0;
39  }
40
41  struct dentry_operations minix_dentry_operations = {
42      .d_hash       = minix_hash,
43  };
44
```

这是从 path 到 inode 的转换函数中的与文件系统相关部分的解析函数。

```
45  static struct dentry *minix_lookup(struct inode * dir, struct dentry *dentry, struct nameidata *nd)
46  {
47      struct inode * inode = NULL;
48      ino_t ino;
49
50      dentry->d_op = dir->i_sb->s_root->d_op;
51
52      if (dentry->d_name.len > minix_sb(dir->i_sb)->s_namelen)
53          return ERR_PTR(-ENAMETOOLONG);
```

```
54
   根据 dentry 中的子目录名在父目录中查找。
55      ino = minix_inode_by_name(dentry);
56      if (ino) {
57          inode = iget(dir->i_sb, ino);
```

调用 Linux-2.6.20/include/linux/fs.h 中的 iget 函数

```
static inline struct inode *iget(struct super_block *sb, unsigned long ino)
{
    struct inode *inode = iget_locked(sb, ino);

    if (inode && (inode->i_state & I_NEW)) {
        sb->s_op->read_inode(inode);
        unlock_new_inode(inode);
    }

    return inode;
}
```

上面标红的行会调用对应文件系统 super block 的 read_inode 函数。而 MINIX 文件系统的 super block 操作函数表如下

Linux-2.6.20/fs/minix/inode.c

```
96  static struct super_operations minix_sops = {
97      .alloc_inode  = minix_alloc_inode,
98      .destroy_inode    = minix_destroy_inode,
99      .read_inode   = minix_read_inode,
100     .write_inode  = minix_write_inode,
```

```
101     .delete_inode = minix_delete_inode,
102     .put_super    = minix_put_super,
103     .statfs       = minix_statfs,
104     .remount_fs   = minix_remount,
105 };
```

调用到 minix_read_inode()去读取磁盘上的 ino 对应的 inode number。super_operations 函数操作集就虚拟文件系统定义的接口函数（类似 C++中的存虚函数，而 MINIX 文件系统类似虚拟文件系统的子类，自然要实现该接口）。由此可见，如果操作系统用 C++来实现的话，抛开性能上可能有所损失外，将更容易维护。

```
58
59          if (!inode)
60              return ERR_PTR(-EACCES);
61      }
```

加入到 directory entry cache 中。

```
62      d_add(dentry, inode);
63      return NULL;
64  }
65
```

Mkdir 系统调用的文件系统相关部分。

```
66  static int minix_mknod(struct inode * dir, struct dentry *dentry, int mode, dev_t rdev)
67  {
68      int error;
69      struct inode *inode;
70
71      if (!old_valid_dev(rdev))
72          return -EINVAL;
73
```

```
74      inode = minix_new_inode(dir, &error);
75
76      if (inode) {
77          inode->i_mode = mode;
78          minix_set_inode(inode, rdev);
79          mark_inode_dirty(inode);
80          error = add_nondir(dentry, inode);
81      }
82      return error;
83  }
84
85  static int minix_create(struct inode * dir, struct dentry *dentry, int mode,
86          struct nameidata *nd)
87  {
88      return minix_mknod(dir, dentry, mode, 0);
89  }
90
    建立 symbol link 的文件系统相关部分
91  static int minix_symlink(struct inode * dir, struct dentry *dentry,
92       const char * symname)
93  {
94      int err = -ENAMETOOLONG;
95      int i = strlen(symname)+1;
96      struct inode * inode;
97
98      if (i > dir->i_sb->s_blocksize)
```

```
99          goto out;
100
101     inode = minix_new_inode(dir, &err);
102     if (!inode)
103         goto out;
104
105     inode->i_mode = S_IFLNK | 0777;
106     minix_set_inode(inode, 0);
107     err = page_symlink(inode, symname, i);
108     if (err)
109         goto out_fail;
110
111     err = add_nondir(dentry, inode);
112 out:
113     return err;
114
115 out_fail:
116     inode_dec_link_count(inode);
117     iput(inode);
118     goto out;
119 }
120
121 static int minix_link(struct dentry * old_dentry, struct inode * dir,
122     struct dentry *dentry)
123 {
124     struct inode *inode = old_dentry->d_inode;
```

```
125
126     if (inode->i_nlink >= minix_sb(inode->i_sb)->s_link_max)
127         return -EMLINK;
128
129     inode->i_ctime = CURRENT_TIME_SEC;
130     inode_inc_link_count(inode);
131     atomic_inc(&inode->i_count);
132     return add_nondir(dentry, inode);
133 }
134
```

“mkdir”系统调用的文件系统相关部分，在参数 dir 表示的父目录下建立子目录。

```
135 static int minix_mkdir(struct inode * dir, struct dentry *dentry, int mode)
136 {
137     struct inode * inode;
138     int err = -EMLINK;
139
140     if (dir->i_nlink >= minix_sb(dir->i_sb)->s_link_max)
141         goto out;
142
143     inode_inc_link_count(dir);
144
145     inode = minix_new_inode(dir, &err);  申请一个 inode
146     if (!inode)
147         goto out_dir;
148
149     inode->i_mode = S_IFDIR | mode;
```

```
150     if (dir->i_mode & S_ISGID)
151         inode->i_mode |= S_ISGID;
152     minix_set_inode(inode, 0);          该函数根据 inode 所代表的不同 object(文件，目录，link，或设备)来设置 inode 操作函数集，file 操作函数集，映射操作函数集。
153
154     inode_inc_link_count(inode);
155
156     err = minix_make_empty(inode, dir);
    为新建立的子目录建立文件内容,即“ . ”,“ .. ”
157     if (err)
158         goto out_fail;
159
160     err = minix_add_link(dentry, inode);
    该函数把子目录的目录项放到父目录的目录文件中。该函数枚举父目录文件中的一个一个目录项，如果找到一个 inode number 为 0 的目录项，表示其是被删除的目录，所以可以利用；或者枚举到已有目录项的最后，自能把该新的子目录添加到父目录文件的最后。具体解释间对该函数的注释。
161     if (err)
162         goto out_fail;
163
164     d_instantiate(dentry, inode);
165 out:
166     return err;
167
168 out_fail:
169     inode_dec_link_count(inode);
170     inode_dec_link_count(inode);
171     iput(inode);
```

```
172 out_dir:
173     inode_dec_link_count(dir);
174     goto out;
175 }
176
177 static int minix_unlink(struct inode * dir, struct dentry *dentry)
178 {
179     int err = -ENOENT;
180     struct inode * inode = dentry->d_inode;
181     struct page * page;
182     struct minix_dir_entry * de;
183
```

查找 dentry 对应的目录项，page 返回查找到的目录所在的页框，而 de 则指向该目录项

```
184     de = minix_find_entry(dentry, &page);
185     if (!de)
186         goto end_unlink;
187
```

删除该目录项。MINIX 文件系统的所谓删除某个目录，只是把该目录所对应的目录项的 inode numebr 置为 0。从这一点可以看出，在 MINIX 文件系统中目录文件的大小是只会增加而不会减少的，因为在具体文件系统的代码中根本就没有缩减目录文件的处理。

比如有“ /tmp/wzhou ”这个目录，对其做 ls 命令，有

A B C D E F G

这 7 个子目录或文件（子目录或文件无关紧要）。我删除 A，B，C，D 这四个文件后，MINIX 文件系统只是把这四个文件的目录项的 inode number 标为零，并没有缩小目录文件“ /tmp/wzhou ”的大小。当我在“ /tmp/wzhou ”目录下 create H 这个文件时，其将占用原来 A 文件所占的目录项，“ /tmp/wzhou ”的文件大小也没有增大。而后我建立了 3 个文件 I，J，K，“ /tmp/wzhou ”目录文件依然没有增大，如果我再 create 一个文件 L，则“ /tmp/wzhou ”要增大了。

我用下图来描述这个过程：

A B C D E F G	假设原始目录文件“ /tmp/wzhou ”的大小为 7（7 个单位，一个文件一个单位，示意性质而已）
■[17] ■ ■ ■ E F G	删除文件A，B，C，D但其原来所占的目录项依然存在。文件系统并不会缩小“ /tmp/wzhou ”文件大小
H ■ ■ ■ E F G	create 新文件 H，它将被安排在原来 A 的目录项的位置
H I ■ ■ E F G	
H I J ■ E F G	
H I J K E F G	
H I J K E F G L	L 文件只能添加在尾部，所以这是“ /tmp/wzhou ”的大小变为 8，增大了。

从上图可以看出目录文件是只增不减的。不单是教学文件系统 MINIX 这样，ext2/3 文件系统也这样。这样做纯粹是为了性能，牺牲的是一点磁盘空间，毕竟现在磁盘不值钱。

```
188    err = minix_delete_entry(de, page);
189    if (err)
190       goto end_unlink;
191
192    inode->i_ctime = dir->i_ctime;
193    inode_dec_link_count(inode);
194 end_unlink:
195    return err;
196 }
197
```

删除目录

```
198 static int minix_rmdir(struct inode * dir, struct dentry *dentry)
```

[17] 表示被删除文件或目录的目录项

```
199 {
200     struct inode * inode = dentry->d_inode;
201     int err = -ENOTEMPTY;
202
203     if (minix_empty_dir(inode)) {
    判断是否空目录
204         err = minix_unlink(dir, dentry);
205         if (!err) {
206             inode_dec_link_count(dir);
207             inode_dec_link_count(inode);
208         }
209     }
210     return err;
211 }
212
    Rename 的文件系统相关部分实现。
213 static int minix_rename(struct inode * old_dir, struct dentry *old_dentry,
214                         struct inode * new_dir, struct dentry *new_dentry)
215 {
216     struct minix_sb_info * info = minix_sb(old_dir->i_sb);
217     struct inode * old_inode = old_dentry->d_inode;
218     struct inode * new_inode = new_dentry->d_inode;
219     struct page * dir_page = NULL;
220     struct minix_dir_entry * dir_de = NULL;
221     struct page * old_page;
222     struct minix_dir_entry * old_de;
```

```
223    int err = -ENOENT;
224
225    old_de = minix_find_entry(old_dentry, &old_page);
226    if (!old_de)
227       goto out;
228
229    if (S_ISDIR(old_inode->i_mode)) {
230       err = -EIO;
231       dir_de = minix_dotdot(old_inode, &dir_page);
232       if (!dir_de)
233          goto out_old;
234    }
235
236    if (new_inode) {
237       struct page * new_page;
238       struct minix_dir_entry * new_de;
239
240       err = -ENOTEMPTY;
241       if (dir_de && !minix_empty_dir(new_inode))
242          goto out_dir;
243
244       err = -ENOENT;
245       new_de = minix_find_entry(new_dentry, &new_page);
246       if (!new_de)
247          goto out_dir;
248       inode_inc_link_count(old_inode);
```

```
249         minix_set_link(new_de, new_page, old_inode);
250         new_inode->i_ctime = CURRENT_TIME_SEC;
251         if (dir_de)
252            drop_nlink(new_inode);
253         inode_dec_link_count(new_inode);
254      } else {
255         if (dir_de) {
256            err = -EMLINK;
257            if (new_dir->i_nlink >= info->s_link_max)
258               goto out_dir;
259         }
260         inode_inc_link_count(old_inode);
261         err = minix_add_link(new_dentry, old_inode);
262         if (err) {
263            inode_dec_link_count(old_inode);
264            goto out_dir;
265         }
266         if (dir_de)
267            inode_inc_link_count(new_dir);
268      }
269
270      minix_delete_entry(old_de, old_page);
271      inode_dec_link_count(old_inode);
272
273      if (dir_de) {
274         minix_set_link(dir_de, dir_page, new_dir);
```

```
275         inode_dec_link_count(old_dir);
276     }
277     return 0;
278 
279 out_dir:
280     if (dir_de) {
281         kunmap(dir_page);
282         page_cache_release(dir_page);
283     }
284 out_old:
285     kunmap(old_page);
286     page_cache_release(old_page);
287 out:
288     return err;
289 }
290 
291 /*
292  * directories can handle most operations...
293  */
```

目录型 inode 的操作函数。

```
294 struct inode_operations minix_dir_inode_operations = {
295     .create       = minix_create,
296     .lookup       = minix_lookup,
297     .link     = minix_link,
298     .unlink       = minix_unlink,
299     .symlink  = minix_symlink,
```

```
300    .mkdir    = minix_mkdir,
301    .rmdir    = minix_rmdir,
302    .mknod    = minix_mknod,
303    .rename       = minix_rename,
304    .getattr  = minix_getattr,
305 };
```

linux-2.6.20/fs/minix/bitmap.c

```
1  /*
2   *  linux/fs/minix/bitmap.c
3   *
4   *  Copyright (C) 1991, 1992  Linus Torvalds
5   */
6
7  /*
8   * Modified for 680x0 by Hamish Macdonald
9   * Fixed for 680x0 by Andreas Schwab
10  */
11
12 /* bitmap.c contains the code that handles the inode and block bitmaps */
13
14 #include "minix.h"
15 #include <linux/smp_lock.h>
```

```
16 #include <linux/buffer_head.h>
17 #include <linux/bitops.h>
18
19 static int nibblemap[] = { 4,3,3,2,3,2,2,1,3,2,2,1,2,1,1,0 };
```

上面的数组中的值是半字（nibble）中的 1 的个数。

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Bin	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
1 的个数	0	1	1	2	1	2	2	3	1	2	2	3	2	3	3	4

用该数组可以通过索引来获得该索引对应的 1 的个数。

```
20
```

该函数用于统计位图(bitmap)中空闲位。

参数map[numblocks]构成一个block buffer的数组。每个成员指向一个block buffer。最后一个block buffer中可能并不是所有的bitmap都有效，所以由参数 numbits 指示该 map[numblocks]数组中的有效位数。一个 block buffer 包含 BLOCK_SIZE(1024)个 bytes，即 8092 个位。

```
21 static unsigned long count_free(struct buffer_head *map[], unsigned numblocks, __u32 numbits)
22 {
23     unsigned i, j, sum = 0;
24     struct buffer_head *bh;
25
```

对 numblocks-1 个 block buffer 进行枚举，不包括最后一个。

```
26     for (i=0; i<numblocks-1; i++) {
27         if (!(bh=map[i]))
28             return(0);
```

```
29          for (j=0; j<BLOCK_SIZE; j++)
30              sum += nibblemap[bh->b_data[j] & 0xf]            累加字节低 4 位的 1 的个数
31                  + nibblemap[(bh->b_data[j]>>4) & 0xf];       累加字节高 4 位的 1 的个数
32      }
33
34      if (numblocks==0 || !(bh=map[numblocks-1]))
35          return(0);
```

对最后一个 block buffer 的处理。

```
36      i = ((numbits-(numblocks-1)*BLOCK_SIZE*8)/16)*2;
```

numbits-(numblocks-1)*BLOCK_SIZE*8 为最后一个 block 中的有效位数。
i 为最后一个 block buffer 的有效位占的字节数(最后一个 byte 可能有无效位)。

```
37      for (j=0; j<i; j++) {
38          sum += nibblemap[bh->b_data[j] & 0xf]
39              + nibblemap[(bh->b_data[j]>>4) & 0xf];
40      }
41
42      i = numbits%16;
43      if (i!=0) {
44          i = *(__u16 *)(&bh->b_data[j]) | ~((1<<i) - 1);
45          sum += nibblemap[i & 0xf] + nibblemap[(i>>4) & 0xf];
46          sum += nibblemap[(i>>8) & 0xf] + nibblemap[(i>>12) & 0xf];
47      }
48      return(sum);
49  }
```

```
50
```
释放由 block 标示的数据块。
参数 block 是相对于整个分区的 block number。即 block 0 是分区引导块，而 block 1 是超级块。
```
51 void minix_free_block(struct inode * inode, int block)
52 {
53     struct super_block * sb = inode->i_sb;
54     struct minix_sb_info * sbi = minix_sb(sb);
55     struct buffer_head * bh;
56     unsigned int bit,zone;
57
```
首先检查要释放的 block 的合法性，它必须落在数据区吧。数据区由[sbi->s_firstdatazone，sbi->s_nzones]表示。
```
58     if (block < sbi->s_firstdatazone || block >= sbi->s_nzones) {
59         printk("Trying to free block not in datazone\n");
60         return;
61     }
62     zone = block - sbi->s_firstdatazone + 1;
```
zone 为相对于数据区头部的 block number。因为 zone bitmap 中的位图是相对于数据区的开始的 block 的。即 zone bitmap 中的第 0 号 bit 是反映数据区的第 0 号 block 的分配状况的，也即相对于整个分区的 sbi->s_firstdatazone 号 block。
```
63     bit = zone & 8191;      一个 block 有 8192 个 bit
64     zone >>= 13;
```
(zone, bit)中的 zone 用于定位中的块号，而 bit 用于定位块中的位号。
```
65     if (zone >= sbi->s_zmap_blocks) {
```
显然 zone 不能超出 zone bitmap 的总块数。
```
66         printk("minix_free_block: nonexistent bitmap buffer\n");
67         return;
```

```
68      }
```
在 minix_fill_super()函数中已经把整个 zone bitmap 都读入内存，存放在 sbi->s_zmap[]数组中。
```
69      bh = sbi->s_zmap[zone];
70      lock_kernel();
71      if (!minix_test_and_clear_bit(bit,bh->b_data))
```
如果要释放的 block 的在 zone bitmap 所代表的位表示该块是空闲的，那表示释放一块没有分配的块。
```
72          printk("free_block (%s:%d): bit already cleared\n",
73                 sb->s_id, block);
74      unlock_kernel();
```
设置一下包含修改了位的 zone bitmap 的 buffer 为 dirty，以便写入磁盘。
```
75      mark_buffer_dirty(bh);
76      return;
77  }
78
```
本函数为 inode 代表的文件分配一块空闲的 block。文件系统是通过 zone bitmap 来获知数据区是否还有空闲块的。Zone bitmap 中的每一位代表数据区的一块。如果某位为 0，表示对应的数据区的该快是空闲的，即还没有分配给任何文件；而某位为 1，则表示对应的数据区的该快已经被分配给某个文件了。所以当文件系统要为某个文件分配空间时，就要稍描 zone bitmap，在其中查找没有置位的位，然后返回其在 zone bitmap 中的位偏移。

参数 inode 代表要分配空间的文件。
```
79  int minix_new_block(struct inode * inode)
80  {
81      struct minix_sb_info *sbi = minix_sb(inode->i_sb);
```
由 inode 获得 super block，获得整个文件系统信息。因为像 zone bitmap 的信息都记录在 super block 中。
```
82      int i;
83
```
对 zone bitmap 进行枚举。下面整个逻辑就是搜寻第一个没有置位的 bit。

```
84     for (i = 0; i < sbi->s_zmap_blocks; i++) {
85        struct buffer_head *bh = sbi->s_zmap[i];
86        int j;
87
88        lock_kernel();
89        if ((j = minix_find_first_zero_bit(bh->b_data, 8192)) < 8192) {
90           minix_set_bit(j,bh->b_data);
91           unlock_kernel();
92           mark_buffer_dirty(bh);
93           j += i*8192 + sbi->s_firstdatazone-1;
94           if (j < sbi->s_firstdatazone || j >= sbi->s_nzones)
95              break;
96           return j;
97        }
98        unlock_kernel();
99     }
100    return 0;
101 }
102
103 unsigned long minix_count_free_blocks(struct minix_sb_info *sbi)
104 {
105    return (count_free(sbi->s_zmap, sbi->s_zmap_blocks,
106       sbi->s_nzones - sbi->s_firstdatazone + 1)
107       << sbi->s_log_zone_size);
108 }
109
```

该函数从磁盘上获取由 ino 所代表的 inode 信息。所谓 raw inode 是指磁盘上记录的 inode 信息。

```
110 struct minix_inode *
111 minix_V1_raw_inode(struct super_block *sb, ino_t ino, struct buffer_head **bh)
112 {
113     int block;
114     struct minix_sb_info *sbi = minix_sb(sb);
115     struct minix_inode *p;
116
```

ino(inode number)的合法性检查。ino 不可能为 0，inode number 是从 root inode 编号开始的，在 MINIX 文件系统中 root inode 编号为 1。sbi->s_ninodes 为该文件系统的总 inode 数。即在数值上 inode number 的范围是[0, sbi->s_ninodes)，但排出 0 以后，实际上是[1, sbi->s_ninodes)。

```
117     if (!ino || ino > sbi->s_ninodes) {
118         printk("Bad inode number on dev %s: %ld is out of range\n",
119                sb->s_id, (long)ino);
120         return NULL;
121     }
122     ino--;          因为 MINIX 文件系统的 inode number 是从 1 开始的。
```

2 = 分区引导块 + super block 块
Inode table 开始于(2 + sbi->s_imap_blocks + sbi->s_zmap_blocks)块
ino / MINIX_INODES_PER_BLOCK 为 ino 所代表的 inode 在 inode table 所占块中的相对块号。

```
123     block = 2 + sbi->s_imap_blocks + sbi->s_zmap_blocks +
124         ino / MINIX_INODES_PER_BLOCK;
```

读出该块。

```
125     *bh = sb_bread(sb, block);
126     if (!*bh) {
```

```
127         printk("Unable to read inode block\n");
128         return NULL;
129     }
130     p = (void *)(*bh)->b_data;
131     return p + ino % MINIX_INODES_PER_BLOCK;
132 }
133
134 struct minix2_inode *
135 minix_V2_raw_inode(struct super_block *sb, ino_t ino, struct buffer_head **bh)
136 {
137     int block;
138     struct minix_sb_info *sbi = minix_sb(sb);
139     struct minix2_inode *p;
140
141     *bh = NULL;
142     if (!ino || ino > sbi->s_ninodes) {
143         printk("Bad inode number on dev %s: %ld is out of range\n",
144                sb->s_id, (long)ino);
145         return NULL;
146     }
147     ino--;
148     block = 2 + sbi->s_imap_blocks + sbi->s_zmap_blocks +
149         ino / MINIX2_INODES_PER_BLOCK;
150     *bh = sb_bread(sb, block);
151     if (!*bh) {
152         printk("Unable to read inode block\n");
```

```
153         return NULL;
154     }
155     p = (void *)(*bh)->b_data;
156     return p + ino % MINIX2_INODES_PER_BLOCK;
157 }
158
159 /* Clear the link count and mode of a deleted inode on disk. */
160
```

清除 inode 在磁盘上的 link count 与 文件 mode。

```
161 static void minix_clear_inode(struct inode *inode)
162 {
163     struct buffer_head *bh = NULL;
164
165     if (INODE_VERSION(inode) == MINIX_V1) {
166         struct minix_inode *raw_inode;
167         raw_inode = minix_V1_raw_inode(inode->i_sb, inode->i_ino, &bh);
168         if (raw_inode) {
169             raw_inode->i_nlinks = 0;
170             raw_inode->i_mode = 0;
171         }
172     } else {
173         struct minix2_inode *raw_inode;
174         raw_inode = minix_V2_raw_inode(inode->i_sb, inode->i_ino, &bh);
175         if (raw_inode) {
176             raw_inode->i_nlinks = 0;
177             raw_inode->i_mode = 0;
```

```
178         }
179     }
180     if (bh) {
181         mark_buffer_dirty(bh);
182         brelse (bh);
183     }
184 }
185
```

该函数释放 inode。主要有两项资源要释放。

1. Inode 所代表的 inode table 中的 inode 节点要清除，minix_clear_inode(inode)完成

2. 在 inode bitmap 中代表该 inode 的位要置位，表示可以再次被分配。

```
186 void minix_free_inode(struct inode * inode)
187 {
188     struct minix_sb_info *sbi = minix_sb(inode->i_sb);
189     struct buffer_head * bh;
190     unsigned long ino;
191
192     ino = inode->i_ino;         取得 inode number
```

Inode number 的合法性检查。

```
193     if (ino < 1 || ino > sbi->s_ninodes) {
194         printk("minix_free_inode: inode 0 or nonexistent inode\n");
195         goto out;
196     }
```

该 inode number 所在的 bit 必须落在 inode bitmap 的 block 内。

```
197     if ((ino >> 13) >= sbi->s_imap_blocks) {
198         printk("minix_free_inode: nonexistent imap in superblock\n");
```

```
199         goto out;
200     }
201
202     minix_clear_inode(inode);  /* clear on-disk copy */
203
204     bh = sbi->s_imap[ino >> 13];      取得该 inode number 所代表位的 block buffer.
205     lock_kernel();
206     if (!minix_test_and_clear_bit(ino & 8191, bh->b_data))
    清除该位。
207         printk("minix_free_inode: bit %lu already cleared\n", ino);
208     unlock_kernel();
209     mark_buffer_dirty(bh);
210 out:
211     clear_inode(inode);     /* clear in-memory copy */
212 }
213
```

在磁盘上分配一个 inode，也就是在 inode bitmap 中寻找第一个没有置位的位。

该函数用于相应创建一个新文件（包括目录）

```
214 struct inode * minix_new_inode(const struct inode * dir, int * error)
215 {
216     struct super_block *sb = dir->i_sb;
217     struct minix_sb_info *sbi = minix_sb(sb);
218     struct inode *inode = new_inode(sb);     分配 inode 的空间
219     struct buffer_head * bh;
220     int i,j;
221
```

```
222     if (!inode) {
223         *error = -ENOMEM;
224         return NULL;
225     }
226     j = 8192;       一个 block 空间包括 8192 bits。
227     bh = NULL;
228     *error = -ENOSPC;
229     lock_kernel();
    对 inode 的 bitmap 进行枚举。
230     for (i = 0; i < sbi->s_imap_blocks; i++) {
231         bh = sbi->s_imap[i];
232         if ((j = minix_find_first_zero_bit(bh->b_data, 8192)) < 8192)
    在整个 block 中寻找第一个为置位的位。
233             break;
234     }
235     if (!bh || j >= 8192) {
236         unlock_kernel();
237         iput(inode);
238         return NULL;
239     }
240     if (minix_test_and_set_bit(j,bh->b_data)) { /* shouldn't happen */
241         printk("new_inode: bit already set\n");
242         unlock_kernel();
243         iput(inode);
244         return NULL;
245     }
```

```
246     unlock_kernel();
247     mark_buffer_dirty(bh);

    i 为 inode bitmap 的 block 偏移，j 位 block 内偏移。
248     j += i*8192;
    i 即为这个新 inode 的编号，也是新文件的 inode number。
249     if (!j || j > sbi->s_ninodes) {
250         iput(inode);
251         return NULL;
252     }
253     inode->i_uid = current->fsuid;     创建的文件的 i_uid 是当前 process 的 fsuid。
254     inode->i_gid = (dir->i_mode & S_ISGID) ? dir->i_gid : current->fsgid;
255     inode->i_ino = j;
256     inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME_SEC;
257     inode->i_blocks = 0;     还没有为该文件（目录）分配空间呢
258     memset(&minix_i(inode)->u, 0, sizeof(minix_i(inode)->u));     初始化 raw inode 部分
259     insert_inode_hash(inode);
260     mark_inode_dirty(inode);
261
262     *error = 0;
263     return inode;
264 }
265
266 unsigned long minix_count_free_inodes(struct minix_sb_info *sbi)
267 {
268     return count_free(sbi->s_imap, sbi->s_imap_blocks, sbi->s_ninodes + 1);
```

```
269 }
```

linux-2.6.20/fs/minix/itree_common.c

```
1  /* Generic part */
2
3  typedef struct {
4     block_t   *p;
5     block_t   key;
6     struct buffer_head *bh;
7  } Indirect;
8
9  static DEFINE_RWLOCK(pointers_lock);
10
11 static inline void add_chain(Indirect *p, struct buffer_head *bh, block_t *v)
12 {
13    p->key = *(p->p = v);
14    p->bh = bh;
15 }
16
17 static inline int verify_chain(Indirect *from, Indirect *to)
18 {
19    while (from <= to && from->key == *from->p)
20       from++;
21    return (from > to);
22 }
```

```
23
24 static inline block_t *block_end(struct buffer_head *bh)
25 {
26     return (block_t *)((char*)bh->b_data + BLOCK_SIZE);
27 }
28
29 static inline Indirect *get_branch(struct inode *inode,
30                   int depth,
31                   int *offsets,
32                   Indirect chain[DEPTH],
33                   int *err)
34 {
35     struct super_block *sb = inode->i_sb;
36     Indirect *p = chain;
37     struct buffer_head *bh;
38
39     *err = 0;
40     /* i_data is not going away, no lock needed */
41     add_chain (chain, NULL, i_data(inode) + *offsets);
42     if (!p->key)
43         goto no_block;
44     while (--depth) {
45         bh = sb_bread(sb, block_to_cpu(p->key));
46         if (!bh)
47             goto failure;
48         read_lock(&pointers_lock);
```

```
49        if (!verify_chain(chain, p))
50           goto changed;
51        add_chain(++p, bh, (block_t *)bh->b_data + *++offsets);
52        read_unlock(&pointers_lock);
53        if (!p->key)
54           goto no_block;
55     }
56     return NULL;
57
58  changed:
59     read_unlock(&pointers_lock);
60     brelse(bh);
61     *err = -EAGAIN;
62     goto no_block;
63  failure:
64     *err = -EIO;
65  no_block:
66     return p;
67  }
68
69  static int alloc_branch(struct inode *inode,
70                int num,
71                int *offsets,
72                Indirect *branch)
73  {
74     int n = 0;
```

```
75     int i;
76     int parent = minix_new_block(inode);
77
78     branch[0].key = cpu_to_block(parent);
79     if (parent) for (n = 1; n < num; n++) {
80        struct buffer_head *bh;
81        /* Allocate the next block */
82        int nr = minix_new_block(inode);
83        if (!nr)
84           break;
85        branch[n].key = cpu_to_block(nr);
86        bh = sb_getblk(inode->i_sb, parent);
87        lock_buffer(bh);
88        memset(bh->b_data, 0, BLOCK_SIZE);
89        branch[n].bh = bh;
90        branch[n].p = (block_t*) bh->b_data + offsets[n];
91        *branch[n].p = branch[n].key;
92        set_buffer_uptodate(bh);
93        unlock_buffer(bh);
94        mark_buffer_dirty_inode(bh, inode);
95        parent = nr;
96     }
97     if (n == num)
98        return 0;
99
100    /* Allocation failed, free what we already allocated */
```

```
101    for (i = 1; i < n; i++)
102       bforget(branch[i].bh);
103    for (i = 0; i < n; i++)
104       minix_free_block(inode, block_to_cpu(branch[i].key));
105    return -ENOSPC;
106 }
107
108 static inline int splice_branch(struct inode *inode,
109                  Indirect chain[DEPTH],
110                  Indirect *where,
111                  int num)
112 {
113    int i;
114
115    write_lock(&pointers_lock);
116
117    /* Verify that place we are splicing to is still there and vacant */
118    if (!verify_chain(chain, where-1) || *where->p)
119       goto changed;
120
121    *where->p = where->key;
122
123    write_unlock(&pointers_lock);
124
125    /* We are done with atomic stuff, now do the rest of housekeeping */
126
```

```
127    inode->i_ctime = CURRENT_TIME_SEC;
128
129    /* had we spliced it onto indirect block? */
130    if (where->bh)
131       mark_buffer_dirty_inode(where->bh, inode);
132
133    mark_inode_dirty(inode);
134    return 0;
135
136 changed:
137    write_unlock(&pointers_lock);
138    for (i = 1; i < num; i++)
139       bforget(where[i].bh);
140    for (i = 0; i < num; i++)
141       minix_free_block(inode, block_to_cpu(where[i].key));
142    return -EAGAIN;
143 }
144
145 static inline int get_block(struct inode * inode, sector_t block,
146          struct buffer_head *bh, int create)
147 {
148    int err = -EIO;
149    int offsets[DEPTH];
150    Indirect chain[DEPTH];
151    Indirect *partial;
152    int left;
```

```
153    int depth = block_to_path(inode, block, offsets);
154
155    if (depth == 0)
156       goto out;
157
158 reread:
159    partial = get_branch(inode, depth, offsets, chain, &err);
160
161    /* Simplest case - block found, no allocation needed */
162    if (!partial) {
163 got_it:
164       map_bh(bh, inode->i_sb, block_to_cpu(chain[depth-1].key));
165       /* Clean up and exit */
166       partial = chain+depth-1; /* the whole chain */
167       goto cleanup;
168    }
169
170    /* Next simple case - plain lookup or failed read of indirect block */
171    if (!create || err == -EIO) {
172 cleanup:
173       while (partial > chain) {
174          brelse(partial->bh);
175          partial--;
176       }
177 out:
178       return err;
```

```
179    }
180
181    /*
182     * Indirect block might be removed by truncate while we were
183     * reading it. Handling of that case (forget what we've got and
184     * reread) is taken out of the main path.
185     */
186    if (err == -EAGAIN)
187       goto changed;
188
189    left = (chain + depth) - partial;
190    err = alloc_branch(inode, left, offsets+(partial-chain), partial);
191    if (err)
192       goto cleanup;
193
194    if (splice_branch(inode, chain, partial, left) < 0)
195       goto changed;
196
197    set_buffer_new(bh);
198    goto got_it;
199
200 changed:
201    while (partial > chain) {
202       brelse(partial->bh);
203       partial--;
204    }
```

```
205    goto reread;
206 }
207
208 static inline int all_zeroes(block_t *p, block_t *q)
209 {
210    while (p < q)
211       if (*p++)
212          return 0;
213    return 1;
214 }
215
216 static Indirect *find_shared(struct inode *inode,
217              int depth,
218              int offsets[DEPTH],
219              Indirect chain[DEPTH],
220              block_t *top)
221 {
222    Indirect *partial, *p;
223    int k, err;
224
225    *top = 0;
226    for (k = depth; k > 1 && !offsets[k-1]; k--)
227       ;
228    partial = get_branch(inode, k, offsets, chain, &err);
229
230    write_lock(&pointers_lock);
```

```
231    if (!partial)
232       partial = chain + k-1;
233    if (!partial->key && *partial->p) {
234       write_unlock(&pointers_lock);
235       goto no_top;
236    }
237    for (p=partial;p>chain && all_zeroes((block_t*)p->bh->b_data,p->p);p--)
238       ;
239    if (p == chain + k - 1 && p > chain) {
240       p->p--;
241    } else {
242       *top = *p->p;
243       *p->p = 0;
244    }
245    write_unlock(&pointers_lock);
246
247    while(partial > p)
248    {
249       brelse(partial->bh);
250       partial--;
251    }
252 no_top:
253    return partial;
254 }
255
256 static inline void free_data(struct inode *inode, block_t *p, block_t *q)
```

```
257 {
258    unsigned long nr;
259
260    for ( ; p < q ; p++) {
261       nr = block_to_cpu(*p);
262       if (nr) {
263          *p = 0;
264          minix_free_block(inode, nr);
265       }
266    }
267 }
268
269 static void free_branches(struct inode *inode, block_t *p, block_t *q, int depth)
270 {
271    struct buffer_head * bh;
272    unsigned long nr;
273
274    if (depth--) {
275       for ( ; p < q ; p++) {
276          nr = block_to_cpu(*p);
277          if (!nr)
278             continue;
279          *p = 0;
280          bh = sb_bread(inode->i_sb, nr);
281          if (!bh)
282             continue;
```

```
283           free_branches(inode, (block_t*)bh->b_data,
284                   block_end(bh), depth);
285           bforget(bh);
286           minix_free_block(inode, nr);
287           mark_inode_dirty(inode);
288       }
289    } else
290       free_data(inode, p, q);
291 }
292
293 static inline void truncate (struct inode * inode)
294 {
295    block_t *idata = i_data(inode);
296    int offsets[DEPTH];
297    Indirect chain[DEPTH];
298    Indirect *partial;
299    block_t nr = 0;
300    int n;
301    int first_whole;
302    long iblock;
303
304    iblock = (inode->i_size + BLOCK_SIZE-1) >> 10;
305    block_truncate_page(inode->i_mapping, inode->i_size, get_block);
306
307    n = block_to_path(inode, iblock, offsets);
308    if (!n)
```

```
309         return;
310
311     if (n == 1) {
312         free_data(inode, idata+offsets[0], idata + DIRECT);
313         first_whole = 0;
314         goto do_indirects;
315     }
316
317     first_whole = offsets[0] + 1 - DIRECT;
318     partial = find_shared(inode, n, offsets, chain, &nr);
319     if (nr) {
320         if (partial == chain)
321             mark_inode_dirty(inode);
322         else
323             mark_buffer_dirty_inode(partial->bh, inode);
324         free_branches(inode, &nr, &nr+1, (chain+n-1) - partial);
325     }
326     /* Clear the ends of indirect blocks on the shared branch */
327     while (partial > chain) {
328         free_branches(inode, partial->p + 1, block_end(partial->bh),
329                 (chain+n-1) - partial);
330         mark_buffer_dirty_inode(partial->bh, inode);
331         brelse (partial->bh);
332         partial--;
333     }
334 do_indirects:
```

```
335     /* Kill the remaining (whole) subtrees */
336     while (first_whole < DEPTH-1) {
337        nr = idata[DIRECT+first_whole];
338        if (nr) {
339           idata[DIRECT+first_whole] = 0;
340           mark_inode_dirty(inode);
341           free_branches(inode, &nr, &nr+1, first_whole+1);
342        }
343        first_whole++;
344     }
345     inode->i_mtime = inode->i_ctime = CURRENT_TIME_SEC;
346     mark_inode_dirty(inode);
347 }
348
349 static inline unsigned nblocks(loff_t size)
350 {
351     unsigned blocks, res, direct = DIRECT, i = DEPTH;
352     blocks = (size + BLOCK_SIZE - 1) >> BLOCK_SIZE_BITS;
353     res = blocks;
354     while (--i && blocks > direct) {
355        blocks -= direct;
356        blocks += BLOCK_SIZE/sizeof(block_t) - 1;
357        blocks /= BLOCK_SIZE/sizeof(block_t);
358        res += blocks;
359        direct = 1;
360     }
```

```
361     return res;
362 }
```

linux-2.6.20/fs/minix/dir.c

```
1   /*
2    *  linux/fs/minix/dir.c
3    *
4    *  Copyright (C) 1991, 1992 Linus Torvalds
5    *
6    *  minix directory handling functions
7    */
8
```

该文件包含 MINIX 文件系统的目录处理相关函数。

```
9   #include "minix.h"
10  #include <linux/highmem.h>
11  #include <linux/smp_lock.h>
12
13  typedef struct minix_dir_entry minix_dirent;
14
15  static int minix_readdir(struct file *, void *, filldir_t);
16
```

目录操作函数

```
17 const struct file_operations minix_dir_operations = {
18     .read     = generic_read_dir,
19     .readdir  = minix_readdir,
20     .fsync    = minix_sync_file,
21 };
22
23 static inline void dir_put_page(struct page *page)
24 {
25     kunmap(page);
26     page_cache_release(page);
27 }
28
29 /*
30  * Return the offset into page `page_nr' of the last valid
31  * byte in that page, plus one.
32  */
```

本函数的原意是返回目录文件第 page_nr 页的最后一个有效 byte，因为目录文件中的每一页的尾部都有可能有空隙。在 i386 CPU 下，一页大小为 4096，MINIX 文件系统的原始版本的目录项大小为 16，升级版大小为 32，则一页最多容纳的目录项分别为 256 与 128。这样在 MINIX 文件系统中只会计算最后一页的最后一项目录项的偏移。

```
33 static unsigned
34 minix_last_byte(struct inode *inode, unsigned long page_nr)
35 {
36     unsigned last_byte = PAGE_CACHE_SIZE;
37
38     if (page_nr == (inode->i_size >> PAGE_CACHE_SHIFT))
```

这个判断就是只有在目录文件的最后一页时才会计算如下的偏移，否则总是返回整页的尾部边界。

```
39          last_byte = inode->i_size & (PAGE_CACHE_SIZE - 1);
40      return last_byte;
41  }
42
```

本函数把 inode 所代表的文件大小转化成系统中的 page 数。

```
43  static inline unsigned long dir_pages(struct inode *inode)
44  {
45      return (inode->i_size+PAGE_CACHE_SIZE-1)>>PAGE_CACHE_SHIFT;
46  }
47
48  static int dir_commit_chunk(struct page *page, unsigned from, unsigned to)
49  {
50      struct inode *dir = (struct inode *)page->mapping->host;
51      int err = 0;
52      page->mapping->a_ops->commit_write(NULL, page, from, to);
53      if (IS_DIRSYNC(dir))
54          err = write_one_page(page, 1);
55      else
56          unlock_page(page);
57      return err;
58  }
59
```

本函数返回 dir 所代表的目录文件中的第 n 页内容。

```
60  static struct page * dir_get_page(struct inode *dir, unsigned long n)
61  {
```

```
62     struct address_space *mapping = dir->i_mapping;
```
通过文件映射的方式来实现。
```
63     struct page *page = read_mapping_page(mapping, n, NULL);
64     if (!IS_ERR(page)) {
65         wait_on_page_locked(page);
66         kmap(page);
67         if (!PageUptodate(page))
68             goto fail;
69     }
70     return page;
71 
72 fail:
73     dir_put_page(page);
74     return ERR_PTR(-EIO);
75 }
```
76 获取 de 指向的 directory entry(de)的下一项。由于在 MINIX 文件系统中目录项是固定长度的（长度由 sbi->s_dirsize 指示），所以非常简单，只要在指向 de 后的 sbi->s_dirsize 个 byte 即可。
```
77 static inline void *minix_next_entry(void *de, struct minix_sb_info *sbi)
78 {
79     return (void*)((char*)de + sbi->s_dirsize);
80 }
81 
```
本函数是系统调用 readdir()的底层实现，对某个目录下子目录或文件进行枚举。
```
82 static int minix_readdir(struct file * filp, void * dirent, filldir_t filldir)
83 {
84     unsigned long pos = filp->f_pos;
```

```
85     struct inode *inode = filp->f_path.dentry->d_inode;
86     struct super_block *sb = inode->i_sb;
87     unsigned offset = pos & ~PAGE_CACHE_MASK;
88     unsigned long n = pos >> PAGE_CACHE_SHIFT;
89     unsigned long npages = dir_pages(inode);
90     struct minix_sb_info *sbi = minix_sb(sb);
91     unsigned chunk_size = sbi->s_dirsize;
92
93     lock_kernel();
94
95     pos = (pos + chunk_size-1) & ~(chunk_size-1);
96     if (pos >= inode->i_size)
97        goto done;
98
99     for ( ; n < npages; n++, offset = 0) {
100       char *p, *kaddr, *limit;
101       struct page *page = dir_get_page(inode, n);
102
103       if (IS_ERR(page))
104          continue;
105       kaddr = (char *)page_address(page);
106       p = kaddr+offset;
107       limit = kaddr + minix_last_byte(inode, n) - chunk_size;
108       for ( ; p <= limit ; p = minix_next_entry(p, sbi)) {
109          minix_dirent *de = (minix_dirent *)p;
110          if (de->inode) {
```

```
111             int over;
112             unsigned l = strnlen(de->name,sbi->s_namelen);
113
114             offset = p - kaddr;
115             over = filldir(dirent, de->name, l,
116                    (n<<PAGE_CACHE_SHIFT) | offset,
117                    de->inode, DT_UNKNOWN);
118             if (over) {
119                dir_put_page(page);
120                goto done;
121             }
122          }
123       }
124       dir_put_page(page);
125    }
126
127 done:
128    filp->f_pos = (n << PAGE_CACHE_SHIFT) | offset;
129    unlock_kernel();
130    return 0;
131 }
132
```

MINIX 文件系统的文件名（包括目录名）比较函数。

```
133 static inline int namecompare(int len, int maxlen,
134    const char * name, const char * buffer)
135 {
```

```
136    if (len < maxlen && buffer[len])
137        return 0;
```
如果你想把 MINIX 文件系统改称向 FAT 系列文件系统一样大小写无关的，则下面应该用大小无关比较才行。
```
138    return !memcmp(name, buffer, len);
139 }
140
141 /*
142  *  minix_find_entry()
143  *
144  * finds an entry in the specified directory with the wanted name. It
145  * returns the cache buffer in which the entry was found, and the entry
146  * itself (as a parameter - res_dir). It does NOT read the inode of the
147  * entry - you'll have to do that yourself if you want to.
148  */
```
本函数查找 dentry 所表示的目录或文件。

dentry 肯定得在其父目录中查找。就像在/etc 目录下查找 passwd 文件用命令“ ls -a /etc ”，该命令会列出/etc 目录下的所有目录或文件，然后你会在输出中查找 passwd 这个文件一样。
```
149 minix_dirent *minix_find_entry(struct dentry *dentry, struct page **res_page)
150 {
151    const char * name = dentry->d_name.name;    这是要查找的文件或目录名
152    int namelen = dentry->d_name.len;           这是要查找的文件或目录名的长度
153    struct inode * dir = dentry->d_parent->d_inode;    取得要查找的文件或目录名的父目录的 inode
154    struct super_block * sb = dir->i_sb;        要查找的文件或目录名所在的文件系统的 super block
155    struct minix_sb_info * sbi = minix_sb(sb);
156    unsigned long n;
157    unsigned long npages = dir_pages(dir);      要查找的文件或目录名的父目录有多少页？查询是基于页来的。由于目录项是不能
```

跨越页的（这纯粹是基于性能的考虑，如果跨页的话，编程上麻烦一点，当然性能自然也会有损失）由于 MINIX 文件系统的目录项是定长的，不像 ext2/3 文件系统那样是不定长的，所以不会跨页。但并不是除了最后一页外，其他页会填满有效的目录项。实际上不是，因为当你删除某个文件或目录时。该文件或目录的目录项可能位于其父目录文件的中间，所以目录项的数组中可能有空洞，即无效的目录项。无效的目录项的 inode number 被设置成 0（在 minix_delete_entry 函数中会赋值。）

```
158     struct page *page = NULL;
159     struct minix_dir_entry *de;
160
161     *res_page = NULL;
162
```

对父目录中的内容一页一页枚举处理。

```
163     for (n = 0; n < npages; n++) {
164         char *kaddr;
```

通过映射方式取得第 n 页

```
165         page = dir_get_page(dir, n);
166         if (IS_ERR(page))
167             continue;
168
```

由于 page 只是页框，不是可寻址的虚拟地址，所以要获得它的虚拟地址 kaddr。

```
169         kaddr = (char*)page_address(page);
```

正是由于目录项不能跨页，所以目录文件的一页的尾部可能有空闲，但该页的第一个目录项肯定在该页的起始处。

```
170         de = (struct minix_dir_entry *) kaddr;
```

kaddr 为当前页的最后一个目录项的地址。

```
171         kaddr += minix_last_byte(dir, n) - sbi->s_dirsize;
```

```
172         for ( ; (char *) de <= kaddr ; de = minix_next_entry(de,sbi)) {
173             if (!de->inode)      如果该目录项指向的 inode number 为 0，表示这是一个被删除的文件或目录。
174                 continue;
175             if (namecompare(namelen,sbi->s_namelen,name,de->name))    比较文件或目录名
176                 goto found;
177         }
178         dir_put_page(page);          释放该页
179     }
180     return NULL;
181
182 found:
183     *res_page = page;            返回指向找到的目录项及该目录项所在的页框
184     return de;
185 }
186
```

下面的函数实现把 dentry 所表示的子目录的目录项添加到其父目录的目录文件中。

添加方式如下：

枚举父目录的目录文件中的目录项。

1．如果在其中找到 inode number 为 0 的目录项，表示该目录项是一个被删除的目录，可以利用。

2．枚举完毕，没有找到可用目录项，则在父目录的目录文件的尾部添加目录项。

```
187 int minix_add_link(struct dentry *dentry, struct inode *inode)
188 {
189     struct inode *dir = dentry->d_parent->d_inode;          父目录的 inode
190     const char * name = dentry->d_name.name;                子目录的目录名
191     int namelen = dentry->d_name.len;                       子目录的目录名长度
192     struct super_block * sb = dir->i_sb;
```

```
193    struct minix_sb_info * sbi = minix_sb(sb);
194    struct page *page = NULL;
195    struct minix_dir_entry * de;
196    unsigned long npages = dir_pages(dir);            父目录的目录文件长度，几页
197    unsigned long n;
198    char *kaddr;
199    unsigned from, to;
200    int err;
201
202    /*
203     * We take care of directory expansion in the same loop
204     * This code plays outside i_size, so it locks the page
205     * to protect that region.
206     */
   对父目录文件每一页进行枚举
207    for (n = 0; n <= npages; n++) {
208       char *dir_end;
209
210       page = dir_get_page(dir, n);
   取得当前页的页框
211       err = PTR_ERR(page);
212       if (IS_ERR(page))
213          goto out;
214       lock_page(page);
215       kaddr = (char*)page_address(page);
216       dir_end = kaddr + minix_last_byte(dir, n);
```

```
217         de = (minix_dirent *)kaddr;
218         kaddr += PAGE_CACHE_SIZE - sbi->s_dirsize;
```
下面的循环是对一页内的目录项进行枚举
```
219         while ((char *)de <= kaddr) {
220             if ((char *)de == dir_end) {
```
已经到该页的最后一个目录项，即没有找到可用的目录项，只能添加了
```
221                 /* We hit i_size */
222                 de->inode = 0;
223                 goto got_it;
224             }
225             if (!de->inode)
```
找到可利用目录项
```
226                 goto got_it;
227             err = -EEXIST;
228             if (namecompare(namelen,sbi->s_namelen,name,de->name))
```
表示要创建的子目录已经在父目录中存在了，即重名了
```
229                 goto out_unlock;
230             de = minix_next_entry(de, sbi);
231         }
232         unlock_page(page);
233         dir_put_page(page);
234     }
235     BUG();
236     return -EINVAL;
237
```
把子目录的目录项加入到父目录文件中

```
got_it:
    from = (char*)de - (char*)page_address(page);
    to = from + sbi->s_dirsize;
    err = page->mapping->a_ops->prepare_write(NULL, page, from, to);
    if (err)
        goto out_unlock;
```
拷贝子目录名
```
    memcpy (de->name, name, namelen);
```
不到最长目录名的空间填充以 0
```
    memset (de->name + namelen, 0, sbi->s_dirsize - namelen - 2);
    de->inode = inode->i_ino;
    err = dir_commit_chunk(page, from, to);
    dir->i_mtime = dir->i_ctime = CURRENT_TIME_SEC;
    mark_inode_dirty(dir);
out_put:
    dir_put_page(page);
out:
    return err;
out_unlock:
    unlock_page(page);
    goto out_put;
}

int minix_delete_entry(struct minix_dir_entry *de, struct page *page)
{
    struct address_space *mapping = page->mapping;
```

```
262     struct inode *inode = (struct inode*)mapping->host;
263     char *kaddr = page_address(page);
264     unsigned from = (char*)de - kaddr;
265     unsigned to = from + minix_sb(inode->i_sb)->s_dirsize;
266     int err;
267
268     lock_page(page);
269     err = mapping->a_ops->prepare_write(NULL, page, from, to);
270     if (err == 0) {
271        de->inode = 0;
272        err = dir_commit_chunk(page, from, to);
273     } else {
274        unlock_page(page);
275     }
276     dir_put_page(page);
277     inode->i_ctime = inode->i_mtime = CURRENT_TIME_SEC;
278     mark_inode_dirty(inode);
279     return err;
280 }
281
```

在父目录 dir 下创建 inode 所代表的子目录

```
282 int minix_make_empty(struct inode *inode, struct inode *dir)
283 {
284     struct address_space *mapping = inode->i_mapping;
```

首先映射要建立子目录的 first page，因为一个子目录即使是空的，也有“ . ”,“ .. ”两个固定的目录。

```
285     struct page *page = grab_cache_page(mapping, 0);
```

```
286    struct minix_sb_info * sbi = minix_sb(inode->i_sb);
287    struct minix_dir_entry * de;
288    char *kaddr;
289    int err;
290
291    if (!page)    L285 行映射失败，只可能是没内存了
292        return -ENOMEM;
293    err = mapping->a_ops->prepare_write(NULL, page, 0, 2 * sbi->s_dirsize);
   这里的 2 * sbi->s_dirsize 即是“ . ”,“ .. ”两个目录
294    if (err) {
295        unlock_page(page);
296        goto fail;
297    }
298
299    kaddr = kmap_atomic(page, KM_USER0);
   获得页框的虚拟地址。
300    memset(kaddr, 0, PAGE_CACHE_SIZE);
301
302    de = (struct minix_dir_entry *)kaddr;
   目录文件的第一个目录项必定是“ . ”目录。
303    de->inode = inode->i_ino;      “ . ”指向自己
304    strcpy(de->name,".");
   目录文件的第二个目录项必定是“ .. ”目录。
305    de = minix_next_entry(de, sbi);
306    de->inode = dir->i_ino;
307    strcpy(de->name,"..");         “ .. ”指向父目录
```

```
308     kunmap_atomic(kaddr, KM_USER0);
309
310     err = dir_commit_chunk(page, 0, 2 * sbi->s_dirsize);     提交
311 fail:
312     page_cache_release(page);
313     return err;
314 }
315
316 /*
317  * routine to check that the specified directory is empty (for rmdir)
318  */
```

检查 inode 所代表的目录是否为空，即该目录下只有“ . ”,“ .. ”两个目录项。

```
319 int minix_empty_dir(struct inode * inode)
320 {
321     struct page *page = NULL;
322     unsigned long i, npages = dir_pages(inode);
323     struct minix_sb_info *sbi = minix_sb(inode->i_sb);
324
```

逻辑同 minix_find_entry（）有点相像，只不过查找的是“ . ”,“ .. ”两个目录项。

```
325     for (i = 0; i < npages; i++) {
326         char *kaddr;
327         minix_dirent * de;
328         page = dir_get_page(inode, i);
329
330         if (IS_ERR(page))
331             continue;
```

```
332
333         kaddr = (char *)page_address(page);
334         de = (minix_dirent *)kaddr;
335         kaddr += minix_last_byte(inode, i) - sbi->s_dirsize;
336
337         while ((char *)de <= kaddr) {
338             if (de->inode != 0) {
339                 /* check for . and .. */
340                 if (de->name[0] != '.')          是“.”目录吗？
341                     goto not_empty;
342                 if (!de->name[1]) {
343                     if (de->inode != inode->i_ino)
344                         goto not_empty;
345                 } else if (de->name[1] != '.')   是“..”目录吗？
346                     goto not_empty;
347                 else if (de->name[2])            包含非上面两个目录，则非空。
348                     goto not_empty;
349             }
350             de = minix_next_entry(de, sbi);
351         }
352         dir_put_page(page);
353     }
354     return 1;
355
356 not_empty:
357     dir_put_page(page);
```

```
358    return 0;
359 }
360
361 /* Releases the page */
362 void minix_set_link(struct minix_dir_entry *de, struct page *page,
363    struct inode *inode)
364 {
365    struct inode *dir = (struct inode*)page->mapping->host;
366    struct minix_sb_info *sbi = minix_sb(dir->i_sb);
367    unsigned from = (char *)de-(char*)page_address(page);
368    unsigned to = from + sbi->s_dirsize;
369    int err;
370
371    lock_page(page);
372    err = page->mapping->a_ops->prepare_write(NULL, page, from, to);
373    if (err == 0) {
374       de->inode = inode->i_ino;
375       err = dir_commit_chunk(page, from, to);
376    } else {
377       unlock_page(page);
378    }
379    dir_put_page(page);
380    dir->i_mtime = dir->i_ctime = CURRENT_TIME_SEC;
381    mark_inode_dirty(dir);
382 }
383
```

```
384 struct minix_dir_entry * minix_dotdot (struct inode *dir, struct page **p)
385 {
386     struct page *page = dir_get_page(dir, 0);
387     struct minix_sb_info *sbi = minix_sb(dir->i_sb);
388     struct minix_dir_entry *de = NULL;
389
390     if (!IS_ERR(page)) {
391         de = minix_next_entry(page_address(page), sbi);
392         *p = page;
393     }
394     return de;
395 }
396
```

根据名字来查找某个文件或目录的 inode number

```
397 ino_t minix_inode_by_name(struct dentry *dentry)
398 {
399     struct page *page;
400     struct minix_dir_entry *de = minix_find_entry(dentry, &page);
401     ino_t res = 0;
402
403     if (de) {
404         res = de->inode;
405         dir_put_page(page);
406     }
407     return res;
408 }
```

linux-2.6.20/fs/minix/itree_common.c

```
1  /* Generic part */
2
3  typedef struct {
4     block_t   *p;
5     block_t   key;
6     struct buffer_head *bh;
7  } Indirect;
8
9  static DEFINE_RWLOCK(pointers_lock);
10
11 static inline void add_chain(Indirect *p, struct buffer_head *bh, block_t *v)
12 {
13    p->key = *(p->p = v);
14    p->bh = bh;
15 }
16
17 static inline int verify_chain(Indirect *from, Indirect *to)
18 {
19    while (from <= to && from->key == *from->p)
20       from++;
21    return (from > to);
22 }
23
```

```
24 static inline block_t *block_end(struct buffer_head *bh)
25 {
26    return (block_t *)((char*)bh->b_data + BLOCK_SIZE);
27 }
28
29 static inline Indirect *get_branch(struct inode *inode,
30                  int depth,
31                  int *offsets,
32                  Indirect chain[DEPTH],
33                  int *err)
34 {
35    struct super_block *sb = inode->i_sb;
36    Indirect *p = chain;
37    struct buffer_head *bh;
38
39    *err = 0;
40    /* i_data is not going away, no lock needed */
41    add_chain (chain, NULL, i_data(inode) + *offsets);
42    if (!p->key)
43       goto no_block;
44    while (--depth) {
45       bh = sb_bread(sb, block_to_cpu(p->key));
46       if (!bh)
47          goto failure;
48       read_lock(&pointers_lock);
49       if (!verify_chain(chain, p))
```

```
50            goto changed;
51        add_chain(++p, bh, (block_t *)bh->b_data + *++offsets);
52        read_unlock(&pointers_lock);
53        if (!p->key)
54            goto no_block;
55    }
56    return NULL;
57
58 changed:
59    read_unlock(&pointers_lock);
60    brelse(bh);
61    *err = -EAGAIN;
62    goto no_block;
63 failure:
64    *err = -EIO;
65 no_block:
66    return p;
67 }
68
69 static int alloc_branch(struct inode *inode,
70                int num,
71                int *offsets,
72                Indirect *branch)
73 {
74    int n = 0;
75    int i;
```

```
76      int parent = minix_new_block(inode);
77
78      branch[0].key = cpu_to_block(parent);
79      if (parent) for (n = 1; n < num; n++) {
80          struct buffer_head *bh;
81          /* Allocate the next block */
82          int nr = minix_new_block(inode);
83          if (!nr)
84              break;
85          branch[n].key = cpu_to_block(nr);
86          bh = sb_getblk(inode->i_sb, parent);
87          lock_buffer(bh);
88          memset(bh->b_data, 0, BLOCK_SIZE);
89          branch[n].bh = bh;
90          branch[n].p = (block_t*) bh->b_data + offsets[n];
91          *branch[n].p = branch[n].key;
92          set_buffer_uptodate(bh);
93          unlock_buffer(bh);
94          mark_buffer_dirty_inode(bh, inode);
95          parent = nr;
96      }
97      if (n == num)
98          return 0;
99
100     /* Allocation failed, free what we already allocated */
101     for (i = 1; i < n; i++)
```

```
102         bforget(branch[i].bh);
103     for (i = 0; i < n; i++)
104         minix_free_block(inode, block_to_cpu(branch[i].key));
105     return -ENOSPC;
106 }
107
108 static inline int splice_branch(struct inode *inode,
109                     Indirect chain[DEPTH],
110                     Indirect *where,
111                     int num)
112 {
113     int i;
114
115     write_lock(&pointers_lock);
116
117     /* Verify that place we are splicing to is still there and vacant */
118     if (!verify_chain(chain, where-1) || *where->p)
119         goto changed;
120
121     *where->p = where->key;
122
123     write_unlock(&pointers_lock);
124
125     /* We are done with atomic stuff, now do the rest of housekeeping */
126
127     inode->i_ctime = CURRENT_TIME_SEC;
```

```
128
129     /* had we spliced it onto indirect block? */
130     if (where->bh)
131         mark_buffer_dirty_inode(where->bh, inode);
132
133     mark_inode_dirty(inode);
134     return 0;
135
136 changed:
137     write_unlock(&pointers_lock);
138     for (i = 1; i < num; i++)
139         bforget(where[i].bh);
140     for (i = 0; i < num; i++)
141         minix_free_block(inode, block_to_cpu(where[i].key));
142     return -EAGAIN;
143 }
144
145 static inline int get_block(struct inode * inode, sector_t block,
146             struct buffer_head *bh, int create)
147 {
148     int err = -EIO;
149     int offsets[DEPTH];
150     Indirect chain[DEPTH];
151     Indirect *partial;
152     int left;
153     int depth = block_to_path(inode, block, offsets);
```

```
154
155    if (depth == 0)
156        goto out;
157
158 reread:
159    partial = get_branch(inode, depth, offsets, chain, &err);
160
161    /* Simplest case - block found, no allocation needed */
162    if (!partial) {
163 got_it:
164        map_bh(bh, inode->i_sb, block_to_cpu(chain[depth-1].key));
165        /* Clean up and exit */
166        partial = chain+depth-1; /* the whole chain */
167        goto cleanup;
168    }
169
170    /* Next simple case - plain lookup or failed read of indirect block */
171    if (!create || err == -EIO) {
172 cleanup:
173        while (partial > chain) {
174            brelse(partial->bh);
175            partial--;
176        }
177 out:
178        return err;
179    }
```

```
180
181    /*
182     * Indirect block might be removed by truncate while we were
183     * reading it. Handling of that case (forget what we've got and
184     * reread) is taken out of the main path.
185     */
186    if (err == -EAGAIN)
187       goto changed;
188
189    left = (chain + depth) - partial;
190    err = alloc_branch(inode, left, offsets+(partial-chain), partial);
191    if (err)
192       goto cleanup;
193
194    if (splice_branch(inode, chain, partial, left) < 0)
195       goto changed;
196
197    set_buffer_new(bh);
198    goto got_it;
199
200 changed:
201    while (partial > chain) {
202       brelse(partial->bh);
203       partial--;
204    }
205    goto reread;
```

```
206 }
207
208 static inline int all_zeroes(block_t *p, block_t *q)
209 {
210    while (p < q)
211       if (*p++)
212          return 0;
213    return 1;
214 }
215
216 static Indirect *find_shared(struct inode *inode,
217              int depth,
218              int offsets[DEPTH],
219              Indirect chain[DEPTH],
220              block_t *top)
221 {
222    Indirect *partial, *p;
223    int k, err;
224
225    *top = 0;
226    for (k = depth; k > 1 && !offsets[k-1]; k--)
227       ;
228    partial = get_branch(inode, k, offsets, chain, &err);
229
230    write_lock(&pointers_lock);
231    if (!partial)
```

```
232        partial = chain + k-1;
233     if (!partial->key && *partial->p) {
234        write_unlock(&pointers_lock);
235        goto no_top;
236     }
237     for (p=partial;p>chain && all_zeroes((block_t*)p->bh->b_data,p->p);p--)
238        ;
239     if (p == chain + k - 1 && p > chain) {
240        p->p--;
241     } else {
242        *top = *p->p;
243        *p->p = 0;
244     }
245     write_unlock(&pointers_lock);
246
247     while(partial > p)
248     {
249        brelse(partial->bh);
250        partial--;
251     }
252 no_top:
253     return partial;
254 }
255
256 static inline void free_data(struct inode *inode, block_t *p, block_t *q)
257 {
```

```
258    unsigned long nr;
259
260    for ( ; p < q ; p++) {
261        nr = block_to_cpu(*p);
262        if (nr) {
263            *p = 0;
264            minix_free_block(inode, nr);
265        }
266    }
267 }
268
269 static void free_branches(struct inode *inode, block_t *p, block_t *q, int depth)
270 {
271    struct buffer_head * bh;
272    unsigned long nr;
273
274    if (depth--) {
275        for ( ; p < q ; p++) {
276            nr = block_to_cpu(*p);
277            if (!nr)
278                continue;
279            *p = 0;
280            bh = sb_bread(inode->i_sb, nr);
281            if (!bh)
282                continue;
283            free_branches(inode, (block_t*)bh->b_data,
```

```
284                     block_end(bh), depth);
285             bforget(bh);
286             minix_free_block(inode, nr);
287             mark_inode_dirty(inode);
288         }
289     } else
290         free_data(inode, p, q);
291 }
292
293 static inline void truncate (struct inode * inode)
294 {
295     block_t *idata = i_data(inode);
296     int offsets[DEPTH];
297     Indirect chain[DEPTH];
298     Indirect *partial;
299     block_t nr = 0;
300     int n;
301     int first_whole;
302     long iblock;
303
304     iblock = (inode->i_size + BLOCK_SIZE-1) >> 10;
305     block_truncate_page(inode->i_mapping, inode->i_size, get_block);
306
307     n = block_to_path(inode, iblock, offsets);
308     if (!n)
309         return;
```

```
310
311    if (n == 1) {
312       free_data(inode, idata+offsets[0], idata + DIRECT);
313       first_whole = 0;
314       goto do_indirects;
315    }
316
317    first_whole = offsets[0] + 1 - DIRECT;
318    partial = find_shared(inode, n, offsets, chain, &nr);
319    if (nr) {
320       if (partial == chain)
321          mark_inode_dirty(inode);
322       else
323          mark_buffer_dirty_inode(partial->bh, inode);
324       free_branches(inode, &nr, &nr+1, (chain+n-1) - partial);
325    }
326    /* Clear the ends of indirect blocks on the shared branch */
327    while (partial > chain) {
328       free_branches(inode, partial->p + 1, block_end(partial->bh),
329              (chain+n-1) - partial);
330       mark_buffer_dirty_inode(partial->bh, inode);
331       brelse (partial->bh);
332       partial--;
333    }
334 do_indirects:
335    /* Kill the remaining (whole) subtrees */
```

```
336    while (first_whole < DEPTH-1) {
337       nr = idata[DIRECT+first_whole];
338       if (nr) {
339          idata[DIRECT+first_whole] = 0;
340          mark_inode_dirty(inode);
341          free_branches(inode, &nr, &nr+1, first_whole+1);
342       }
343       first_whole++;
344    }
345    inode->i_mtime = inode->i_ctime = CURRENT_TIME_SEC;
346    mark_inode_dirty(inode);
347 }
348
349 static inline unsigned nblocks(loff_t size)
350 {
351    unsigned blocks, res, direct = DIRECT, i = DEPTH;
352    blocks = (size + BLOCK_SIZE - 1) >> BLOCK_SIZE_BITS;
353    res = blocks;
354    while (--i && blocks > direct) {
355       blocks -= direct;
356       blocks += BLOCK_SIZE/sizeof(block_t) - 1;
357       blocks /= BLOCK_SIZE/sizeof(block_t);
358       res += blocks;
359       direct = 1;
360    }
361    return res;
```

```
362 }
```

linux-2.6.20/fs/minix/itree_v1.c

```
1  #include <linux/buffer_head.h>
2  #include "minix.h"
3
4  enum {DEPTH = 3, DIRECT = 7}; /* Only double indirect */
5
6  typedef u16 block_t;    /* 16 bit, host order */
7
8  static inline unsigned long block_to_cpu(block_t n)
9  {
10     return n;
11 }
12
13 static inline block_t cpu_to_block(unsigned long n)
14 {
15     return n;
16 }
17
18 static inline block_t *i_data(struct inode *inode)
19 {
20     return (block_t *)minix_i(inode)->u.i1_data;
```

```
21 }
22
23 static int block_to_path(struct inode * inode, long block, int offsets[DEPTH])
24 {
25     int n = 0;
26
27     if (block < 0) {
28         printk("minix_bmap: block<0\n");
29     } else if (block >= (minix_sb(inode->i_sb)->s_max_size/BLOCK_SIZE)) {
30         printk("minix_bmap: block>big\n");
31     } else if (block < 7) {
32         offsets[n++] = block;
33     } else if ((block -= 7) < 512) {
34         offsets[n++] = 7;
35         offsets[n++] = block;
36     } else {
37         block -= 512;
38         offsets[n++] = 8;
39         offsets[n++] = block>>9;
40         offsets[n++] = block & 511;
41     }
42     return n;
43 }
44
45 #include "itree_common.c"
46
```

```
47 int V1_minix_get_block(struct inode * inode, long block,
48           struct buffer_head *bh_result, int create)
49 {
50    return get_block(inode, block, bh_result, create);
51 }
52
53 void V1_minix_truncate(struct inode * inode)
54 {
55    truncate(inode);
56 }
57
58 unsigned V1_minix_blocks(loff_t size)
59 {
60    return nblocks(size);
61 }
```

linux-2.6.20/fs/minix/itree_v2.c

```
1  #include <linux/buffer_head.h>
2  #include "minix.h"
3
4  enum {DIRECT = 7, DEPTH = 4}; /* Have triple indirect */
5
6  typedef u32 block_t;    /* 32 bit, host order */
```

```
7
8  static inline unsigned long block_to_cpu(block_t n)
9  {
10    return n;
11 }
12
13 static inline block_t cpu_to_block(unsigned long n)
14 {
15    return n;
16 }
17
18 static inline block_t *i_data(struct inode *inode)
19 {
20    return (block_t *)minix_i(inode)->u.i2_data;
21 }
22
23 static int block_to_path(struct inode * inode, long block, int offsets[DEPTH])
24 {
25    int n = 0;
26
27    if (block < 0) {
28       printk("minix_bmap: block<0\n");
29    } else if (block >= (minix_sb(inode->i_sb)->s_max_size/BLOCK_SIZE)) {
30       printk("minix_bmap: block>big\n");
31    } else if (block < 7) {
32       offsets[n++] = block;
```

```
33      } else if ((block -= 7) < 256) {
34          offsets[n++] = 7;
35          offsets[n++] = block;
36      } else if ((block -= 256) < 256*256) {
37          offsets[n++] = 8;
38          offsets[n++] = block>>8;
39          offsets[n++] = block & 255;
40      } else {
41          block -= 256*256;
42          offsets[n++] = 9;
43          offsets[n++] = block>>16;
44          offsets[n++] = (block>>8) & 255;
45          offsets[n++] = block & 255;
46      }
47      return n;
48  }
49
50  #include "itree_common.c"
51
52  int V2_minix_get_block(struct inode * inode, long block,
53            struct buffer_head *bh_result, int create)
54  {
55      return get_block(inode, block, bh_result, create);
56  }
57
58  void V2_minix_truncate(struct inode * inode)
```

```
59 {
60     truncate(inode);
61 }
62
63 unsigned V2_minix_blocks(loff_t size)
64 {
65     return nblocks(size);
66 }
```

部分 minix-fuse file system driver 注释

MINIX 分区格式化工具 mkfs.minix 源码注释

util-linux-2.12r/disk-utils/mkfs.minix.c

```
1  /*
2   * mkfs.c - make a linux (minix) file-system.
3   *
4   * (C) 1991 Linus Torvalds. This file may be redistributed as per
5   * the Linux copyright.
6   */
7
8  /*
```

```
9   * DD.MM.YY
10  *
11  * 24.11.91  -   Time began. Used the fsck sources to get started.
12  *
13  * 25.11.91  -   Corrected some bugs. Added support for ".badblocks"
14  *     The algorithm for ".badblocks" is a bit weird, but
15  *     it should work. Oh, well.
16  *
17  * 25.01.92  -   Added the -l option for getting the list of bad blocks
18  *     out of a named file. (Dave Rivers, rivers@ponds.uucp)
19  *
20  * 28.02.92  -   Added %-information when using -c.
21  *
22  * 28.02.93  -   Added support for other namelengths than the original
23  *     14 characters so that I can test the new kernel routines..
24  *
25  * 09.10.93  -   Make exit status conform to that required by fsutil
26  *     (Rik Faith, faith@cs.unc.edu)
27  *
28  * 31.10.93  -   Added inode request feature, for backup floppies: use
29  *     32 inodes, for a news partition use more.
30  *     (Scott Heavner, sdh@po.cwru.edu)
31  *
32  * 03.01.94  -   Added support for file system valid flag.
33  *     (Dr. Wettstein, greg%wind.uucp@plains.nodak.edu)
34  *
```

```
35  * 30.10.94  -  Added support for v2 filesystem
36  *     (Andreas Schwab, schwab@issan.informatik.uni-dortmund.de)
37  *
38  * 09.11.94  -   Added test to prevent overwrite of mounted fs adapted
39  *     from Theodore Ts'o's (tytso@athena.mit.edu) mke2fs
40  *     program.  (Daniel Quinlan, quinlan@yggdrasil.com)
41  *
42  * 03.20.95  -   Clear first 512 bytes of filesystem to make certain that
43  *     the filesystem is not misidentified as a MS-DOS FAT filesystem.
44  *     (Daniel Quinlan, quinlan@yggdrasil.com)
45  *
46  * 02.07.96  -  Added small patch from Russell King to make the program a
47  *     good deal more portable (janl@math.uio.no)
48  *
49  * Usage:  mkfs [-c | -l filename ] [-v] [-nXX] [-iXX] device [size-in-blocks]
50  *
51  *  -c for readablility checking (SLOW!)
52  *     -l for getting a list of bad blocks from a file.
53  *  -n for namelength (currently the kernel only uses 14 or 30)
54  *  -i for number of inodes
55  *  -v for v2 filesystem
56  *
57  * The device may be a block device or a image of one, but this isn't
58  * enforced (but it's not much fun on a character device :-).
59  */
60
```

```
61 #include <stdio.h>
62 #include <time.h>
63 #include <unistd.h>
64 #include <string.h>
65 #include <signal.h>
66 #include <fcntl.h>
67 #include <ctype.h>
68 #include <stdlib.h>
69 #include <termios.h>
70 #include <sys/stat.h>
71 #include <sys/ioctl.h>
72 #include <mntent.h>
73 #include <getopt.h>
74
75 #include "minix.h"
76 #include "nls.h"
77
78 #ifndef BLKGETSIZE
79 #define BLKGETSIZE _IO(0x12,96)    /* return device size */
80 #endif
81
82 #ifndef __GNUC__
83 #error "needs gcc for the bitop-__asm__'s"
84 #endif
85
86 #define MINIX_ROOT_INO 1
```

```
87 #define MINIX_BAD_INO 2
88
89 #define TEST_BUFFER_BLOCKS 16
90 #define MAX_GOOD_BLOCKS 512
91
92 #define UPPER(size,n) ((size+((n)-1))/(n))
93 #define INODE_SIZE (sizeof(struct minix_inode))
94
95 #define INODE_SIZE2 (sizeof(struct minix2_inode))
96 #define INODE_BLOCKS UPPER(INODES, (version2 ? MINIX2_INODES_PER_BLOCK \
97                 : MINIX_INODES_PER_BLOCK))
98 #define INODE_BUFFER_SIZE (INODE_BLOCKS * BLOCK_SIZE)
99
100 #define BITS_PER_BLOCK (BLOCK_SIZE<<3)
101
102 static char * program_name = "mkfs";
103 static char * device_name = NULL;
104 static int DEV = -1;
105 static long BLOCKS = 0;
106 static int check = 0;
107 static int badblocks = 0;
108 static int namelen = 30;   /* default (changed to 30, per Linus's
109                suggestion, Sun Nov 21 08:05:07 1993) */
110 static int dirsize = 32;
111 static int magic = MINIX_SUPER_MAGIC2;
112 static int version2 = 0;
```

```
113
114 static char root_block[BLOCK_SIZE] = "\0";
115
116 static char * inode_buffer = NULL;
117 #define Inode (((struct minix_inode *) inode_buffer)-1)
118 #define Inode2 (((struct minix2_inode *) inode_buffer)-1)
119
120 static char super_block_buffer[BLOCK_SIZE];
121 static char boot_block_buffer[512];
122 #define Super (*(struct minix_super_block *)super_block_buffer)
123 #define INODES ((unsigned long)Super.s_ninodes)
124 #define ZONES ((unsigned long)(version2 ? Super.s_zones : Super.s_nzones))
125 #define IMAPS ((unsigned long)Super.s_imap_blocks)
126 #define ZMAPS ((unsigned long)Super.s_zmap_blocks)
127 #define FIRSTZONE ((unsigned long)Super.s_firstdatazone)
128 #define ZONESIZE ((unsigned long)Super.s_log_zone_size)
129 #define MAXSIZE ((unsigned long)Super.s_max_size)
130 #define MAGIC (Super.s_magic)
131 #define NORM_FIRSTZONE (2+IMAPS+ZMAPS+INODE_BLOCKS)
132
133 static char *inode_map;
134 static char *zone_map;
135
136 static unsigned short good_blocks_table[MAX_GOOD_BLOCKS];
137 static int used_good_blocks = 0;
138 static unsigned long req_nr_inodes = 0;
```

```
139
140 #include "bitops.h"
141
142 #define inode_in_use(x) (bit(inode_map,(x)))
143 #define zone_in_use(x) (bit(zone_map,(x)-FIRSTZONE+1))
144
145 #define mark_inode(x) (setbit(inode_map,(x)))
146 #define unmark_inode(x) (clrbit(inode_map,(x)))
147
148 #define mark_zone(x) (setbit(zone_map,(x)-FIRSTZONE+1))
149 #define unmark_zone(x) (clrbit(zone_map,(x)-FIRSTZONE+1))
150
151 static void
152 die(char *str) {
153    fprintf(stderr, "%s: ", program_name);
154    fprintf(stderr, str, device_name);
155    fprintf(stderr, "\n");
156    exit(8);
157 }
158
159 static void
160 usage(void) {
161    fprintf(stderr, "%s (%s)\n", program_name, util_linux_version);
162    fprintf(stderr,
163       _("Usage: %s [-c | -l filename] [-nXX] [-iXX] /dev/name [blocks]\n"),
164         program_name);
```

```
165    exit(16);
166 }
167
168 /*
169  * Check to make certain that our new filesystem won't be created on
170  * an already mounted partition.  Code adapted from mke2fs, Copyright
171  * (C) 1994 Theodore Ts'o.  Also licensed under GPL.
172  */
```

检查要被“ format ”的分区是否正被 mount 着。对正 mount 的分区当然禁止格式化。

通过读取/etc/mtab 中有无要格式化的分区的设备名来检验是否被 mount。

/etc/mtab 当前安装的文件系统列表。由 scripts 初始化，并由 mount 命令自动更新。

```
173 static void
174 check_mount(void) {
175    FILE * f;
176    struct mntent * mnt;
177
178    if ((f = setmntent (MOUNTED, "r")) == NULL)
179       return;
180    while ((mnt = getmntent (f)) != NULL)
181       if (strcmp (device_name, mnt->mnt_fsname) == 0)
182          break;
183    endmntent (f);
184    if (!mnt)
185       return;
186
187    die(_("%s is mounted; will not make a filesystem here!"));
```

```
188 }
189
```
验证 fd 所代表的设备文件又 offset 那么大吗。用的方法非常土，先移动文件指针到 offset，然后在该处读取一个字节，如果都成功，表示该设备最起码有 offset 大。
```
190 static long
191 valid_offset (int fd, int offset) {
192     char ch;
193
194     if (lseek (fd, offset, 0) < 0)
195         return 0;
196     if (read (fd, &ch, 1) < 1)
197         return 0;
198     return 1;
199 }
200
```
在设备不能告诉该分区大小的情况下，自能用土办法来探测该分区的大小。
```
201 static int
202 count_blocks (int fd) {
203     int high, low;
204
205     low = 0;
```
先成倍的扩展，看到哪儿失败
```
206     for (high = 1; valid_offset (fd, high); high *= 2)
207         low = high;
```
然后在最后一次成功与最后一次失败之间用对半查找到边界。
```
208     while (low < high - 1)
```

```
209    {
210       const int mid = (low + high) / 2;
211
212       if (valid_offset (fd, mid))
213          low = mid;
214       else
215          high = mid;
216    }
217    valid_offset (fd, 0);
218    return (low + 1);
219 }
220
```

这里 file 是要“ format 的“ 分区的设备名，如/dev/hdd1。

```
221 static int
222 get_size(const char  *file) {
223    int fd;
224    long   size;
225
226    fd = open(file, O_RDWR);
227    if (fd < 0) {
228       perror(file);
229       exit(1);
230    }
```

通过 IO CONTROL 来获得该设备的 sector 数。

```
231    if (ioctl(fd, BLKGETSIZE, &size) >= 0) {
232       close(fd);
```

```
   每个 section 512 bytes。
233        return (size * 512);
234    }
235
   如果通过接口无法获得，则只能试探法来探测大小。即通过不断向外移动文件指针，看是否“移动”失败。如果失败，表示大小在这个范围内，再定位。方法虽土，但确实有效。
236    size = count_blocks(fd);
237    close(fd);
238    return size;
239 }
240
   该函数文件指针移动的位置就是 MINIX 文件系统相关数据结构的起始位置。
241 static void
242 write_tables(void) {
243    /* Mark the super block valid. */
244    Super.s_state |= MINIX_VALID_FS;
245    Super.s_state &= ~MINIX_ERROR_FS;
246
247    if (lseek(DEV, 0, SEEK_SET))
248        die(_("seek to boot block failed in write_tables"));
249    if (512 != write(DEV, boot_block_buffer, 512))
250        die(_("unable to clear boot sector"));
251    if (BLOCK_SIZE != lseek(DEV, BLOCK_SIZE, SEEK_SET))
252        die(_("seek failed in write_tables"));
253    if (BLOCK_SIZE != write(DEV, super_block_buffer, BLOCK_SIZE))
254        die(_("unable to write super-block"));
```

```
255    if (IMAPS*BLOCK_SIZE != write(DEV,inode_map,IMAPS*BLOCK_SIZE))
256       die(_("unable to write inode map"));
257    if (ZMAPS*BLOCK_SIZE != write(DEV,zone_map,ZMAPS*BLOCK_SIZE))
258       die(_("unable to write zone map"));
259    if (INODE_BUFFER_SIZE != write(DEV,inode_buffer,INODE_BUFFER_SIZE))
260       die(_("unable to write inodes"));
261
262 }
263
264 static void
265 write_block(int blk, char * buffer) {
266    if (blk*BLOCK_SIZE != lseek(DEV, blk*BLOCK_SIZE, SEEK_SET))
267       die(_("seek failed in write_block"));
268    if (BLOCK_SIZE != write(DEV, buffer, BLOCK_SIZE))
269       die(_("write failed in write_block"));
270 }
271
272 static int
273 get_free_block(void) {
274    int blk;
275
276    if (used_good_blocks+1 >= MAX_GOOD_BLOCKS)
277       die(_("too many bad blocks"));
278    if (used_good_blocks)
279       blk = good_blocks_table[used_good_blocks-1]+1;
280    else
```

```
281         blk = FIRSTZONE;
282     while (blk < ZONES && zone_in_use(blk))
283         blk++;
284     if (blk >= ZONES)
285         die(_("not enough good blocks"));
286     good_blocks_table[used_good_blocks] = blk;
287     used_good_blocks++;
288     return blk;
289 }
290
291 static void
292 mark_good_blocks(void) {
293     int blk;
294
295     for (blk=0 ; blk < used_good_blocks ; blk++)
296         mark_zone(good_blocks_table[blk]);
297 }
298
299 static inline int
300 next(int zone) {
301     if (!zone)
302         zone = FIRSTZONE-1;
303     while (++zone < ZONES)
304         if (zone_in_use(zone))
305             return zone;
306     return 0;
```

```
307 }
308
```

在根目录下生成一个文件.badblocks，它所占的 block 就是在检查分区时发现的坏块。下面就是手工合成该文件的 inode。这确实是一个巧妙的方法。

```
309 static void
310 make_bad_inode(void) {
311     struct minix_inode * inode = &Inode[MINIX_BAD_INO];
312     int i,j,zone;
313     int ind=0,dind=0;
314     unsigned short ind_block[BLOCK_SIZE>>1];
315     unsigned short dind_block[BLOCK_SIZE>>1];
316
317 #define NEXT_BAD (zone = next(zone))
318
319     if (!badblocks)
320         return;
321     mark_inode(MINIX_BAD_INO);
322     inode->i_nlinks = 1;
323     inode->i_time = time(NULL);
324     inode->i_mode = S_IFREG + 0000;
325     inode->i_size = badblocks*BLOCK_SIZE;
326     zone = next(0);
327     for (i=0 ; i<7 ; i++) {
328         inode->i_zone[i] = zone;
329         if (!NEXT_BAD)
330             goto end_bad;
```

```
331    }
332    inode->i_zone[7] = ind = get_free_block();
333    memset(ind_block,0,BLOCK_SIZE);
334    for (i=0 ; i<512 ; i++) {
335       ind_block[i] = zone;
336       if (!NEXT_BAD)
337          goto end_bad;
338    }
339    inode->i_zone[8] = dind = get_free_block();
340    memset(dind_block,0,BLOCK_SIZE);
341    for (i=0 ; i<512 ; i++) {
342       write_block(ind,(char *) ind_block);
343       dind_block[i] = ind = get_free_block();
344       memset(ind_block,0,BLOCK_SIZE);
345       for (j=0 ; j<512 ; j++) {
346          ind_block[j] = zone;
347          if (!NEXT_BAD)
348             goto end_bad;
349       }
350    }
351    die(_("too many bad blocks"));
352 end_bad:
353    if (ind)
354       write_block(ind, (char *) ind_block);
355    if (dind)
356       write_block(dind, (char *) dind_block);
```

```
357 }
358
359 static void
360 make_bad_inode2 (void) {
361    struct minix2_inode *inode = &Inode2[MINIX_BAD_INO];
362    int i, j, zone;
363    int ind = 0, dind = 0;
364    unsigned long ind_block[BLOCK_SIZE >> 2];
365    unsigned long dind_block[BLOCK_SIZE >> 2];
366
367    if (!badblocks)
368       return;
369    mark_inode (MINIX_BAD_INO);
370    inode->i_nlinks = 1;
371    inode->i_atime = inode->i_mtime = inode->i_ctime = time (NULL);
372    inode->i_mode = S_IFREG + 0000;
373    inode->i_size = badblocks * BLOCK_SIZE;
374    zone = next (0);
375    for (i = 0; i < 7; i++) {
376       inode->i_zone[i] = zone;
377       if (!NEXT_BAD)
378          goto end_bad;
379    }
380    inode->i_zone[7] = ind = get_free_block ();
381    memset (ind_block, 0, BLOCK_SIZE);
382    for (i = 0; i < 256; i++) {
```

```
383       ind_block[i] = zone;
384       if (!NEXT_BAD)
385          goto end_bad;
386    }
387    inode->i_zone[8] = dind = get_free_block ();
388    memset (dind_block, 0, BLOCK_SIZE);
389    for (i = 0; i < 256; i++) {
390       write_block (ind, (char *) ind_block);
391       dind_block[i] = ind = get_free_block ();
392       memset (ind_block, 0, BLOCK_SIZE);
393       for (j = 0; j < 256; j++) {
394          ind_block[j] = zone;
395          if (!NEXT_BAD)
396             goto end_bad;
397       }
398    }
399    /* Could make triple indirect block here */
400    die (_("too many bad blocks"));
401 end_bad:
402    if (ind)
403       write_block (ind, (char *) ind_block);
404    if (dind)
405       write_block (dind, (char *) dind_block);
406 }
407
```

下面的函数手工建立 root(/)目录的 inode 节点。

磁盘上 inode 节点的结构如下。

```
30 /*
31  * This is the original minix inode layout on disk.
32  * Note the 8-bit gid and atime and ctime.
33  */
34 struct minix_inode {
35     __u16 i_mode;
36     __u16 i_uid;
37     __u32 i_size;
38     __u32 i_time;
39     __u8  i_gid;
40     __u8  i_nlinks;
41     __u16 i_zone[9];
42 };
```

```
408 static void
409 make_root_inode(void) {
410     struct minix_inode * inode = &Inode[MINIX_ROOT_INO];
411
412     mark_inode(MINIX_ROOT_INO);    对该 inode 对应的 inode bitmap 置位，标示已占用。
413     inode->i_zone[0] = get_free_block();
414     inode->i_nlinks = 2;        “ / ”本身 + ".."
415     inode->i_time = time(NULL);
```

如果有坏块，则会生成“ .badblocks ”文件，所以为 3。

```
416     if (badblocks)
417         inode->i_size = 3*dirsize;
```

```
418    else {
419       root_block[2*dirsize] = '\0';
420       root_block[2*dirsize+1] = '\0';
421       inode->i_size = 2*dirsize;
422    }
423    inode->i_mode = S_IFDIR + 0755;
424    inode->i_uid = getuid();
425    if (inode->i_uid)
426       inode->i_gid = getgid();
427    write_block(inode->i_zone[0],root_block);
428 }
429
430 static void
431 make_root_inode2 (void) {
432    struct minix2_inode *inode = &Inode2[MINIX_ROOT_INO];
433
434    mark_inode (MINIX_ROOT_INO);
435    inode->i_zone[0] = get_free_block ();
436    inode->i_nlinks = 2;
437    inode->i_atime = inode->i_mtime = inode->i_ctime = time (NULL);
438    if (badblocks)
439       inode->i_size = 3 * dirsize;
440    else {
441       root_block[2 * dirsize] = '\0';
442       root_block[2 * dirsize + 1] = '\0';
443       inode->i_size = 2 * dirsize;
```

```
444     }
445     inode->i_mode = S_IFDIR + 0755;
446     inode->i_uid = getuid();
447     if (inode->i_uid)
448        inode->i_gid = getgid();
449     write_block (inode->i_zone[0], root_block);
450 }
451
```

该函数名曰“setup tables”，即建立 inode bitmap，zone bitmap，inode table。

```
452 static void
453 setup_tables(void) {
454     int i;
455     unsigned long inodes;
456
457     memset(super_block_buffer,0,BLOCK_SIZE);    super block 的 1K buffer
458     memset(boot_block_buffer,0,512);            分区头部 MBR 的 buffer
459     Super.s_magic = magic;      设文件系统签名
460     Super.s_log_zone_size = 0;
```

不同 MINIX 文件系统的允许的文件大小是不同的。

这里 7+512+512*512 的解释如下：

7 是 inode 中的直接块数

512 是一次间接块数

512*512 是二次间接块数

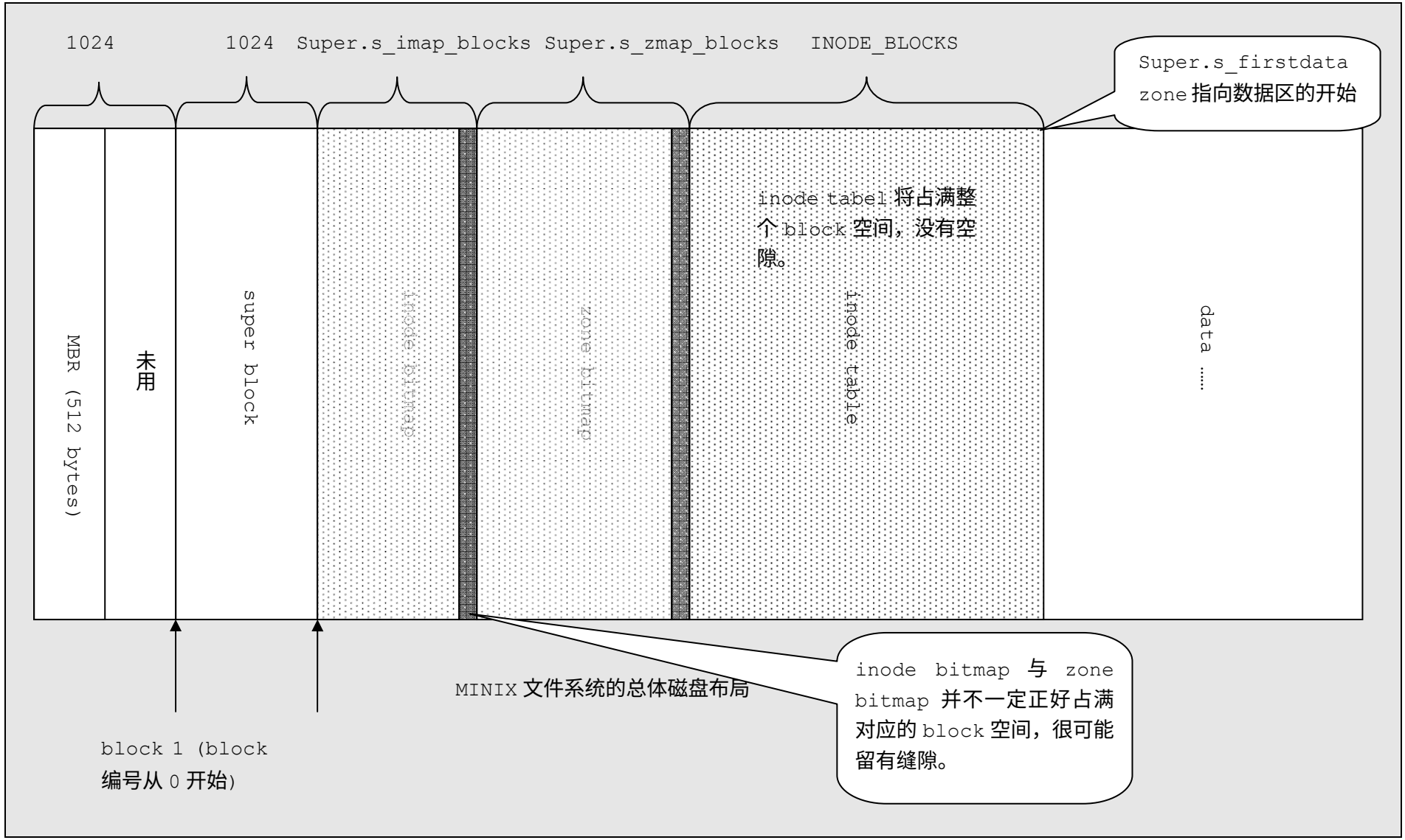
```
461     Super.s_max_size = version2 ? 0x7fffffff : (7+512+512*512)*1024;
462     if (version2)
```

```
463       Super.s_zones = BLOCKS;
464    else
465       Super.s_nzones = BLOCKS;
466
```

用户在“ format ”分区时可以指定 inode 的分配数。如果没有指定，则默认为 inode 数是总体块数的 1/3。

```
467 /* some magic nrs: 1 inode / 3 blocks */
468    if ( req_nr_inodes == 0 )
469       inodes = BLOCKS/3;
470    else
471       inodes = req_nr_inodes;
472    /* Round up inode count to fill block size */
473    if (version2)
474       inodes = ((inodes + MINIX2_INODES_PER_BLOCK - 1) &
475            ~(MINIX2_INODES_PER_BLOCK - 1));
476    else
477       inodes = ((inodes + MINIX_INODES_PER_BLOCK - 1) &
478            ~(MINIX_INODES_PER_BLOCK - 1));
```

从上面的取整可知，inode table 所占空间是充满整个 block 的，不会再最后一个 block 留有余地。这样的话，inode bitmap，zone bitmap 所占空间则不一定会正好对齐在边界上。对“ MINIX 文件系统的总体磁盘布局 ”图细化。


1024
1024
Super.s_imap_blocks
Super.s_zmap_blocks
INODE_BLOCKS
Super.s_firstdata zone 指向数据区的开始
MBR (512 bytes)
未用
super block
inode bitmap
zone bitmap
inode table
data ……
inode tabel 将占满整个 block 空间，没有空隙。
MINIX 文件系统的总体磁盘布局
inode bitmap 与 zone bitmap 并不一定正好占满对应的 block 空间，很可能留有缝隙。
block 1 (block 编号从 0 开始)

```
479    if (inodes > 65535)
480       inodes = 65535;
```
MINIX 文件系统最多文件数（目录也使一种文件）为 65535。
```
481    Super.s_ninodes = inodes;
482
483    /* The old code here
484     * ZMAPS = 0;
485     * while (ZMAPS != UPPER(BLOCKS - NORM_FIRSTZONE + 1,BITS_PER_BLOCK))
486     *   ZMAPS = UPPER(BLOCKS - NORM_FIRSTZONE + 1,BITS_PER_BLOCK);
487     * was no good, since it may loop. - aeb
488     */
```
计算 inode bitmap 所占用的 block 数目。
```
489     = UPPER(INODES + 1, BITS_PER_BLOCK);
```
计算 zone bitmap 所占用的 block 数目。
```
490    Super.s_zmap_blocks = UPPER(BLOCKS - (1+IMAPS+INODE_BLOCKS),
491                  BITS_PER_BLOCK+1);
492    Super.s_firstdatazone = NORM_FIRSTZONE;

   #define NORM_FIRSTZONE (2+IMAPS+ZMAPS+INODE_BLOCKS)
```
从上面的宏可以看出，数据区紧接着 inode table。
```
493
494    inode_map = malloc(IMAPS * BLOCK_SIZE);
495    zone_map = malloc(ZMAPS * BLOCK_SIZE);
496    if (!inode_map || !zone_map)
497       die(_("unable to allocate buffers for maps"));
498    memset(inode_map,0xff,IMAPS * BLOCK_SIZE);
```

```
499    memset(zone_map,0xff,ZMAPS * BLOCK_SIZE);
```
首先都初始化成已被占用。bitmap 中置 1 表示被占用。

```
500    for (i = FIRSTZONE ; i<ZONES ; i++)
501        unmark_zone(i);
502    for (i = MINIX_ROOT_INO ; i<=INODES ; i++)
503        unmark_inode(i);
```
然后在依次清零。

```
504    inode_buffer = malloc(INODE_BUFFER_SIZE);
505    if (!inode_buffer)
506        die(_("unable to allocate buffer for inodes"));
507    memset(inode_buffer,0,INODE_BUFFER_SIZE);
```
初始化 indoe table 的 buffer 为全零。

```
508    printf(_("%ld inodes\n"),INODES);
509    printf(_("%ld blocks\n"),ZONES);
510    printf(_("Firstdatazone=%ld (%ld)\n"),FIRSTZONE,NORM_FIRSTZONE);
511    printf(_("Zonesize=%d\n"),BLOCK_SIZE<<ZONESIZE);
512    printf(_("Maxsize=%ld\n\n"),MAXSIZE);
513 }
514
515 /*
516  * Perform a test of a block; return the number of
517  * blocks readable/writeable.
518  */
```

```
519 static long
520 do_check(char * buffer, int try, unsigned int current_block) {
521    long got;
522
523    /* Seek to the correct loc. */
524    if (lseek(DEV, current_block * BLOCK_SIZE, SEEK_SET) !=
525             current_block * BLOCK_SIZE ) {
526        die(_("seek failed during testing of blocks"));
527    }
528
529
530    /* Try the read */
531    got = read(DEV, buffer, try * BLOCK_SIZE);
532    if (got < 0) got = 0;
533    if (got & (BLOCK_SIZE - 1 )) {
534       printf(_("Weird values in do_check: probably bugs\n"));
535    }
536    got /= BLOCK_SIZE;
537    return got;
538 }
539
540 static unsigned int currently_testing = 0;
541
542 static void
543 alarm_intr(int alnum) {
544    if (currently_testing >= ZONES)
```

```
545         return;
546     signal(SIGALRM,alarm_intr);
547     alarm(5);
548     if (!currently_testing)
549         return;
550     printf("%d ...", currently_testing);
551     fflush(stdout);
552 }
553
```

所谓 check，也就是对分区中的 block 来一遍读操作，如果都成功，表示应该该分区中没有坏块。

```
554 static void
555 check_blocks(void) {
556     int try,got;
557     static char buffer[BLOCK_SIZE * TEST_BUFFER_BLOCKS];
558
559     currently_testing=0;
560     signal(SIGALRM,alarm_intr);
561     alarm(5);
562     while (currently_testing < ZONES) {
563         if (lseek(DEV,currently_testing*BLOCK_SIZE,SEEK_SET) !=
564         currently_testing*BLOCK_SIZE)
565             die(_("seek failed in check_blocks"));
566         try = TEST_BUFFER_BLOCKS;
567         if (currently_testing + try > ZONES)
568             try = ZONES-currently_testing;
569         got = do_check(buffer, try, currently_testing);
```

```
570         currently_testing += got;
571         if (got == try)
572             continue;
573         if (currently_testing < FIRSTZONE)
574             die(_("bad blocks before data-area: cannot make fs"));
575         mark_zone(currently_testing);
576         badblocks++;
577         currently_testing++;
578     }
579     if (badblocks > 1)
580         printf(_("%d bad blocks\n"), badblocks);
581     else if (badblocks == 1)
582         printf(_("one bad block\n"));
583 }
584
585 static void
586 get_list_blocks(char *filename) {
587     FILE *listfile;
588     unsigned long blockno;
589
590     listfile = fopen(filename,"r");
591     if (listfile == NULL)
592         die(_("can't open file of bad blocks"));
593
```

该文件的格式很简单，每一行就是有问题的 `block number`，向下面那样：

```
1456
1568
2390
10034
ooo
```

该函数会把该文件标示的 block 在 zone bitmap 中标记为已被占用。

```
594    while (!feof(listfile)) {
595        fscanf(listfile,"%ld\n", &blockno);
596        mark_zone(blockno);
597        badblocks++;
598    }
599    fclose(listfile);
600
601    if(badblocks > 1)
602        printf(_("%d bad blocks\n"), badblocks);
603    else if (badblocks == 1)
604        printf(_("one bad block\n"));
605 }
606
607 int
608 main(int argc, char ** argv) {
609   int i;
610   char * tmp;
611   struct stat statbuf;
```

```
612  char * listfile = NULL;
613  char * p;
614
615  if (argc && *argv)
616    program_name = *argv;
617  if ((p = strrchr(program_name, '/')) != NULL)
618    program_name = p+1;
619
620  setlocale(LC_ALL, "");
621  bindtextdomain(PACKAGE, LOCALEDIR);
622  textdomain(PACKAGE);
623
624  if (argc == 2 &&
625      (!strcmp(argv[1], "-V") || !strcmp(argv[1], "--version"))) {
626      printf(_("%s from %s\n"), program_name, util_linux_version);
627      exit(0);
628  }
629
630  if (INODE_SIZE * MINIX_INODES_PER_BLOCK != BLOCK_SIZE)
631    die(_("bad inode size"));
632  if (INODE_SIZE2 * MINIX2_INODES_PER_BLOCK != BLOCK_SIZE)
633    die(_("bad inode size"));
```

无论是哪一种 MINIX 文件系统，节点数必须被块大小整除。

```
634
635  opterr = 0;
```

```
636  while ((i = getopt(argc, argv, "ci:l:n:v")) != -1)
637    switch (i) {
638      case 'c':       用-c option 可以在“ format ”时 check 磁盘。
639    check=1; break;
640      case 'i':       用-i option 可以让用户指定创建 MINIX 文件系统的 inode 数
641    req_nr_inodes = (unsigned long) atol(optarg);
642    break;
643      case 'l':       用-l option 可以一个有问题块（坏块）的列表的文件，在“ format ”时就可以标示出来
644    listfile = optarg; break;
645      case 'n':
646    i = strtoul(optarg,&tmp,0);
647    if (*tmp)
648      usage();
649    if (i == 14)          原始版的 MINIX 文件系统
650      magic = MINIX_SUPER_MAGIC;
651    else if (i == 30)     升级版的 MINIX 文件系统
652      magic = MINIX_SUPER_MAGIC2;
653    else
654      usage();
655    namelen = i;
656    dirsize = i+2;
657    break;
658      case 'v':
659    version2 = 1;
660    break;
661      default:
```

```
662    usage();
663    }
664   argc -= optind;
665   argv += optind;
666   if (argc > 0 && !device_name) {
667    device_name = argv[0];
668    argc--;
669    argv++;
670   }
671   if (argc > 0) {
672     BLOCKS = strtol(argv[0],&tmp,0);
673     if (*tmp) {
674       printf(_("strtol error: number of blocks not specified"));
675       usage();
676     }
677   }
678
679   if (device_name && !BLOCKS)
680    BLOCKS = get_size (device_name) / 1024;
```

获得该分区的块数。

```
681   if (!device_name || BLOCKS<10) {
682    usage();
683   }
684   if (version2) {
685    if (namelen == 14)
686      magic = MINIX2_SUPER_MAGIC;
```

原始版 MINIX 文件系统的签名

```
687    else
688      magic = MINIX2_SUPER_MAGIC2;     升级版 MINIX 文件系统的签名
689   } else
690    if (BLOCKS > 65535)              原始 MINIX 文件系统是一种比较小型的文件系统。它最大只能管理 64M 的分区。
691      BLOCKS = 65535;
692   check_mount();     /* is it already mounted? */      只能对没有被 mount 的分区进行"format"
```

一个 MINIX 文件系统建立后，默认在 root 目录下有如下的目录文件。

1。 “ . ”

2。 “ .. “

3。 “ .badblocks “

.badblocks 文件是用于在 “ format ” 时检测到的坏块构成的文件，如果没有坏块，当然就没有该文件。

```
693   tmp = root_block;
```

在磁盘上目录项的结构如下：

```
78  struct minix_dir_entry {
79      __u16 inode;
80      char name[0];
81  };
```

头两个 byte 是该目录项的 inode number，而后面则是该目录项的名字，其长度因不同 MINIX 文件系统而不同。

在处理根目录本身，其 inode number 是 1，名字是 “ . ”。

```
694   *(short *)tmp = 1;
```

```
695  strcpy(tmp+2,".");

   指向下一个目录项。
696  tmp += dirsize;
   根目录的父目录是根目录本身，所以 inode number 也是 1，名字是“ .. ”。
697  *(short *)tmp = 1;
698  strcpy(tmp+2,"..");
   指向下一个目录项。
699  tmp += dirsize;
   .badblocks 文件的 inode number 是 2。
700  *(short *)tmp = 2;
701  strcpy(tmp+2,".badblocks");
702  DEV = open(device_name,O_RDWR );
703  if (DEV<0)
704    die(_("unable to open %s"));
705  if (fstat(DEV,&statbuf)<0)
706    die(_("unable to stat %s"));
707  if (!S_ISBLK(statbuf.st_mode))      对非 block 设备，不用 check
708    check=0;
709  else if (statbuf.st_rdev == 0x0300 || statbuf.st_rdev == 0x0340)      这是什么设备？
710    die(_("will not try to make filesystem on '%s'"));
   计算该分区所占用的 inode bitmap，zone bitmap，inode table 的 block 数。
711  setup_tables();
712  if (check)
713    check_blocks();
714  else if (listfile)
```

```
715    get_list_blocks(listfile);
```

一个刚格式化完的分区在数据区并不是光秃秃什么都没有，最起码根目录是有的，下面就是建立根目录，同时把在 check 分区时检查出的坏块看成是一个文件的内容，其文件名为根目录下的/.badblocks 文件。

```
716  if (version2) {
717    make_root_inode2 ();
718    make_bad_inode2 ();
719  } else {
720      make_root_inode();
721      make_bad_inode();
722  }
723  mark_good_blocks();
```

把缓冲的 MBR，super block，inode bitmap，zone bitmap，inode table 写入磁盘。

```
724  write_tables();
725  return 0;
726 }
```

联系

Walter Zhou

mailto:z-l-dragon@hotmail.com