



















# 第 9 章 文件系统(fs)

## 9.1 概述

表 9.1 文件系统程序列表

	Name	Size	Last modified (GMT)	Description
	<a href="#">Makefile</a>	5053 bytes	1991-12-02 03:21:31	m
	<a href="#">bitmap.c</a>	4042 bytes	1991-11-26 21:31:53	m
	<a href="#">block_dev.c</a>	1422 bytes	1991-10-31 17:19:55	m
	<a href="#">buffer.c</a>	9072 bytes	1991-12-06 20:21:00	m
	<a href="#">char_dev.c</a>	2103 bytes	1991-11-19 09:10:22	
	<a href="#">exec.c</a>	9134 bytes	1991-12-01 20:01:01	m
	<a href="#">fcntl.c</a>	1455 bytes	1991-10-02 14:16:29	m
	<a href="#">file_dev.c</a>	1852 bytes	1991-12-01 19:02:43	m
	<a href="#">file_table.c</a>	122 bytes	1991-10-02 14:16:29	m
	<a href="#">inode.c</a>	6933 bytes	1991-12-06 20:16:35	m
	<a href="#">ioctl.c</a>	977 bytes	1991-11-19 09:13:05	
	<a href="#">namei.c</a>	16562 bytes	1991-11-25 19:19:59	m
	<a href="#">open.c</a>	4340 bytes	1991-11-25 19:21:01	m
	<a href="#">pipe.c</a>	2385 bytes	1991-10-18 19:02:33	m
	<a href="#">read_write.c</a>	2802 bytes	1991-11-25 15:47:20	m
	<a href="#">stat.c</a>	1175 bytes	1991-10-02 14:16:29	m
	<a href="#">super.c</a>	5628 bytes	1991-12-06 20:10:12	m
	<a href="#">truncate.c</a>	1148 bytes	1991-10-02 14:16:29	m

## 9.2 Makefile 文件

### 9.2.1 功能描述

### 9.2.2 代码注释

列表 linux/fs/Makefile 文件

[1](#) AR =gar # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。

```

2 AS      =gas      # GNU 的汇编程序。
3 CC      =gcc      # GNU C 语言编译器。
4 LD      =gld      # GNU 的连接程序。

# C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
# -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
# 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-mstring-insns Linux 自己
# 添加的优化选项，以后不再使用；-nostdinc -I../include 不使用默认路径中的包含文件，而使
# 用这里指定目录中的(../include)。
5 CFLAGS  =-Wall -O -fstrength-reduce -fcombine-regs -fomit-frame-pointer \
6         -mstring-insns -nostdinc -I../include

# C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
# 出设备或指定的输出文件中；-nostdinc -I../include 同前。
7 CPP     =gcc -E -nostdinc -I../include
8
# 下面的规则指示 make 利用下面的命令将所有的.c 文件编译生成.s 汇编程序。该规则的命令
# 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止（-S），从而产生与
# 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
# 去掉.c 而加上.s 后缀。-o 表示其后是输出文件的名称。其中$*.s（或$@）是自动目标变量，
# $<代表第一个先决条件，这里即是符合条件*.c 的文件。
9 .c.s:
10      $(CC) $(CFLAGS) \
11      -S -o $*.s $<
# 将所有*.c 文件编译成*.o 目标文件。不进行连接。
12 .c.o:
13      $(CC) $(CFLAGS) \
14      -c -o $*.o $<
# 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。16 行是实现该操作的具体命令。
15 .s.o:
16      $(AS) -o $*.o $<
17
# 定义目标文件变量 OBJS。
18 OBJS=  open.o read_write.o inode.o file_table.o buffer.o super.o \
19         block_dev.o char_dev.o file_dev.o stat.o exec.o pipe.o namei.o \
20         bitmap.o fcntl.o ioctl.o truncate.o
21
# 在有了先决条件 OBJS 后使用下面的命令连接成目标 fs.o
22 fs.o: $(OBJS)
23      $(LD) -r -o fs.o $(OBJS)
24
# 下面的规则用于清理工作。当执行'make clean'时，就会执行 26--27 行上的命令，去除所有编译
# 连接生成的文件。'rm' 是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
25 clean:
26      rm -f core *.o *.a tmp_make
27      for i in *.c;do rm -f `basename $$i .c`.s;done
28
# 下面得目标或规则用于检查各文件之间的依赖关系。方法如下：
# 使用字符串编辑程序 sed 对 Makefile 文件（这里即是自己）进行处理，输出为删除 Makefile
# 文件中'### Dependencies' 行后面的所有行（下面从 35 开始的行），并生成 tmp_make
# 临时文件（30 行的作用）。然后对 fs/目录下的每一个 C 文件执行 gcc 预处理操作。
# -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
# 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标

```

# 文件名加上其依赖关系—该源文件中包含的所有头文件列表。把预处理结果都添加到临时  
# 文件 tmp\_make 中, 然后将该临时文件复制成新的 Makefile 文件。

```

29 dep:
30     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
31     (for i in *.c;do $(CPP) -M $$i;done) >> tmp_make
32     cp tmp_make Makefile
33
34 ### Dependencies:
35 bitmap.o : bitmap.c ../include/string.h ../include/linux/sched.h \
36     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
37     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h
38 block_dev.o : block_dev.c ../include/errno.h ../include/linux/sched.h \
39     ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
40     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
41     ../include/asm/segment.h ../include/asm/system.h
42 buffer.o : buffer.c ../include/stdarg.h ../include/linux/config.h \
43     ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
44     ../include/sys/types.h ../include/linux/mm.h ../include/signal.h \
45     ../include/linux/kernel.h ../include/asm/system.h ../include/asm/io.h
46 char_dev.o : char_dev.c ../include/errno.h ../include/sys/types.h \
47     ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
48     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
49     ../include/asm/segment.h ../include/asm/io.h
50 exec.o : exec.c ../include/errno.h ../include/string.h \
51     ../include/sys/stat.h ../include/sys/types.h ../include/a.out.h \
52     ../include/linux/fs.h ../include/linux/sched.h ../include/linux/head.h \
53     ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
54     ../include/asm/segment.h
55 fcntl.o : fcntl.c ../include/string.h ../include/errno.h \
56     ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
57     ../include/sys/types.h ../include/linux/mm.h ../include/signal.h \
58     ../include/linux/kernel.h ../include/asm/segment.h ../include/fcntl.h \
59     ../include/sys/stat.h
60 file_dev.o : file_dev.c ../include/errno.h ../include/fcntl.h \
61     ../include/sys/types.h ../include/linux/sched.h ../include/linux/head.h \
62     ../include/linux/fs.h ../include/linux/mm.h ../include/signal.h \
63     ../include/linux/kernel.h ../include/asm/segment.h
64 file_table.o : file_table.c ../include/linux/fs.h ../include/sys/types.h
65 inode.o : inode.c ../include/string.h ../include/sys/stat.h \
66     ../include/sys/types.h ../include/linux/sched.h ../include/linux/head.h \
67     ../include/linux/fs.h ../include/linux/mm.h ../include/signal.h \
68     ../include/linux/kernel.h ../include/asm/system.h
69 ioctl.o : ioctl.c ../include/string.h ../include/errno.h \
70     ../include/sys/stat.h ../include/sys/types.h ../include/linux/sched.h \
71     ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
72     ../include/signal.h
73 namei.o : namei.c ../include/linux/sched.h ../include/linux/head.h \
74     ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
75     ../include/signal.h ../include/linux/kernel.h ../include/asm/segment.h \
76     ../include/string.h ../include/fcntl.h ../include/errno.h \
77     ../include/const.h ../include/sys/stat.h
78 open.o : open.c ../include/string.h ../include/errno.h ../include/fcntl.h \
79     ../include/sys/types.h ../include/utime.h ../include/sys/stat.h \

```

```

80 ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
81 ../include/linux/mm.h ../include/signal.h ../include/linux/tty.h \
82 ../include/termios.h ../include/linux/kernel.h ../include/asm/segment.h
83 pipe.o : pipe.c ../include/signal.h ../include/sys/types.h \
84 ../include/linux/sched.h ../include/linux/head.h ../include/linux/fs.h \
85 ../include/linux/mm.h ../include/asm/segment.h
86 read_write.o : read_write.c ../include/sys/stat.h ../include/sys/types.h \
87 ../include/errno.h ../include/linux/kernel.h ../include/linux/sched.h \
88 ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \
89 ../include/signal.h ../include/asm/segment.h
90 stat.o : stat.c ../include/errno.h ../include/sys/stat.h \
91 ../include/sys/types.h ../include/linux/fs.h ../include/linux/sched.h \
92 ../include/linux/head.h ../include/linux/mm.h ../include/signal.h \
93 ../include/linux/kernel.h ../include/asm/segment.h
94 super.o : super.c ../include/linux/config.h ../include/linux/sched.h \
95 ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
96 ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
97 ../include/asm/system.h ../include/errno.h ../include/sys/stat.h
98 truncate.o : truncate.c ../include/linux/sched.h ../include/linux/head.h \
99 ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
100 ../include/signal.h ../include/sys/stat.h

```

### 9.2.3 其它信息

## 9.3 bitmap.c 程序

### 9.3.1 功能描述

该程序主要用于处理 i 节点和逻辑块（磁盘块或区段）的位图。

### 9.3.2 代码注释

列表 linux/fs/bitmap.c 程序

```

1 /*
2  * linux/fs/bitmap.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /* bitmap.c contains the code that handles the inode and block bitmaps */
8 /* bitmap.c 程序含有处理 i 节点和磁盘块位图的代码 */
9 #include <string.h> // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
10 // 主要使用了其中的 memset() 函数。
11 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
12 // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
13 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
14
15 // 将指定地址(addr)处的一块内存清零。嵌入汇编程序宏。
16 // 输入: eax = 0, ecx = 数据块大小 BLOCK_SIZE/4, edi = addr。

```

```

13 #define clear_block(addr) \
14 __asm__ ( "cld\n\t" \           // 清方向位。
15          "rep\n\t" \           // 重复执行存储数据 (0)。
16          "stosl" \
17          :: "a" (0), "c" (BLOCK_SIZE/4), "D" ((long) (addr)): "cx", "di" )
18
19 // 置位指定地址开始的第 nr 个位偏移处的比特位(nr 可以大于 32!)。返回原比特位 (0 或 1)。
20 // 输入: %0 - eax (返回值), %1 - eax(0); %2 - nr, 位偏移值; %3 - (addr), addr 的内容。
21 #define set_bit(nr, addr) ({ \
22 register int res __asm__ ("ax"); \
23 __asm__ __volatile__ ("btsl %2,%3\n\tsetb %%al": \
24 "a" (res): "" (0), "r" (nr), "m" (*(addr))); \
25 res;})
26
27 // 复位指定地址开始的第 nr 位偏移处的比特位。返回原比特位的反码 (1 或 0)。
28 // 输入: %0 - eax (返回值), %1 - eax(0); %2 - nr, 位偏移值; %3 - (addr), addr 的内容。
29 #define clear_bit(nr, addr) ({ \
30 register int res __asm__ ("ax"); \
31 __asm__ __volatile__ ("btrl %2,%3\n\tsetnb %%al": \
32 "a" (res): "" (0), "r" (nr), "m" (*(addr))); \
33 res;})
34
35 // 从 addr 开始寻找第 1 个 0 值比特位。
36 // 输入: %0 - ecx(返回值); %1 - ecx(0); %2 - esi(addr)。
37 // 在 addr 指定地址开始的位图中寻找第 1 个是 0 的比特位, 并将其距离 addr 的比特位偏移值返回。
38 #define find_first_zero(addr) ({ \
39 int __res; \
40 __asm__ ( "cld\n\t" \           // 清方向位。
41          "l:|t lodsl\n\t" \     // 取[esi]→eax。
42          "notl %%eax\n\t" \     // eax 中每位取反。
43          "bsfl %%eax,%%edx\n\t" \ // 从位 0 扫描 eax 中是 1 的第 1 个位, 其偏移值→edx。
44          "je 2f\n\t" \         // 如果 eax 中全是 0, 则向前跳转到标号 2 处(40 行)。
45          "addl %%edx,%%ecx\n\t" \ // 偏移值加入 ecx(ecx 中是位图中首个是 0 的比特位的偏移值)
46          "jmp 3f\n\t" \         // 向前跳转到标号 3 处(结束)。
47          "2:|t addl $32,%%ecx\n\t" \ // 没有找到 0 比特位, 则将 ecx 加上 1 个长字的位偏移量 32。
48          "cmpl $8192,%%ecx\n\t" \ // 已经扫描了 8192 位 (1024 字节) 了吗?
49          "jl 1b\n\t" \         // 若还没有扫描完 1 块数据, 则向前跳转到标号 1 处, 继续。
50          "3:" \               // 结束。此时 ecx 中是位偏移量。
51          : "c" (__res): "c" (0), "S" (addr): "ax", "dx", "si"); \
52 __res;})
53
54 // 释放设备 dev 上数据区中的逻辑块 block。
55 // 复位指定逻辑块 block 的逻辑块位图比特位。
56 void free_block(int dev, int block)
57 {
58     struct super_block * sb;
59     struct buffer_head * bh;
60
61     // 从设备 dev 取超级块, 如果指定设备不存在, 则出错死机。
62     if (!(sb = get_super(dev)))
63         panic("trying to free block on nonexistent device");
64     // 若逻辑块号小于首个逻辑块号或者大于设备上总逻辑块数, 则出错, 死机。
65     if (block < sb->s_firstdatazone || block >= sb->s_nzones)

```

```

55         panic("trying to free block not in datazone");
// 从 hash 表中寻找该块数据。若找到了则判断其有效性，并清已修改和更新标志，释放该数据块。
56         bh = get_hash_table(dev, block);
57         if (bh) {
58             if (bh->b_count != 1) {
59                 printk("trying to free block (%04x:%d), count=%d\n",
60                     dev, block, bh->b_count);
61                 return;
62             }
63             bh->b_dirt=0;           // 复位脏（已修改）标志位。
64             bh->b_uptodate=0;       // 复位更新标志。
65             brelse(bh);
66         }
// 计算 block 在数据区开始算起的逻辑块号（从 1 开始计数）。然后对逻辑块(区段)位图进行操作，
// 复位对应的比特位。若对应比特位原来即是 0，则出错，死机。
67         block -= sb->s_firstdatazone - 1; // block = block - (-1);
68         if (clear_bit(block&8191, sb->s_zmap[block/8192]->b_data)) {
69             printk("block (%04x:%d) ", dev, block+sb->s_firstdatazone-1);
70             panic("free_block: bit already cleared");
71         }
// 置相应逻辑块位图所在缓冲区已修改标志。
72         sb->s_zmap[block/8192]->b_dirt = 1;
73     }
74
// 向设备 dev 申请一个逻辑块（磁盘块，区段）。返回逻辑块号。
// 置位指定逻辑块 block 的逻辑块位图比特位。
75 int new_block(int dev)
76 {
77     struct buffer_head * bh;
78     struct super_block * sb;
79     int i, j;
80
// 从设备 dev 取超级块，如果指定设备不存在，则出错死机。
81     if (!(sb = get_super(dev)))
82         panic("trying to get new block from nonexistant device");
// 扫描逻辑块位图，寻找首个 0 比特位，寻找空闲逻辑块，获取放置该逻辑块的块号。。
83     j = 8192;
84     for (i=0 ; i<8 ; i++)
85         if (bh=sb->s_zmap[i])
86             if ((j=find_first_zero(bh->b_data))<8192)
87                 break;
// 如果全部扫描完还没找到(i>=8 或 j>=8192)或者位图所在的缓冲块无效(bh=NULL)则 返回 0，
// 退出（没有空闲逻辑块）。
88     if (i>=8 || !bh || j>=8192)
89         return 0;
// 设置新逻辑块对应逻辑块位图中的比特位，若对应比特位已经置位，则出错，死机。
90     if (set_bit(j, bh->b_data))
91         panic("new_block: bit already set");
// 置对应缓冲区块的已修改标志。如果新逻辑块大于该设备上的总逻辑块数，则说明指定逻辑块在
// 对应设备上不存在。申请失败，返回 0，退出。
92     bh->b_dirt = 1;
93     j += i*8192 + sb->s_firstdatazone-1;
94     if (j >= sb->s_nzones)

```

```

95         return 0;
// 读取设备上的该新逻辑块数据（验证）。如果失败则死机。
96         if (!(bh=getblk(dev,j)))
97             panic("new_block: cannot get block");
// 新块的引用计数应为1。否则死机。
98         if (bh->b_count != 1)
99             panic("new block: count is != 1");
// 将该新逻辑块清零，并置位更新标志和已修改标志。然后释放对应缓冲区，返回逻辑块号。
100         clear_block(bh->b_data);
101         bh->b_uptodate = 1;
102         bh->b_dirt = 1;
103         brelse(bh);
104         return j;
105     }
106
//// 释放指定的 i 节点。
// 复位对应 i 节点位图比特位。
107 void free_inode(struct m_inode * inode)
108 {
109     struct super_block * sb;
110     struct buffer_head * bh;
111
// 如果 i 节点指针=NULL，则退出。
112     if (!inode)
113         return;
// 如果 i 节点上的设备号字段为 0，说明该节点无用，则用 0 清空对应 i 节点所占内存区，并返回。
114     if (!inode->i_dev) {
115         memset(inode, 0, sizeof(*inode));
116         return;
117     }
// 如果此 i 节点还有其它程序引用，则不能释放，说明内核有问题，死机。
118     if (inode->i_count>1) {
119         printk("trying to free inode with count=%d\n", inode->i_count);
120         panic("free_inode");
121     }
// 如果文件目录项连接数不为 0，则表示还有其它文件目录项在使用该节点，不应释放，而应该放回等。
122     if (inode->i_nlinks)
123         panic("trying to free inode with links");
// 取 i 节点所在设备的超级块，测试设备是否存在。
124     if (!(sb = get_super(inode->i_dev)))
125         panic("trying to free inode on nonexistent device");
// 如果 i 节点号=0 或大于该设备上 i 节点总数，则出错（0 号 i 节点保留没有使用）。
126     if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
127         panic("trying to free inode 0 or nonexistant inode");
// 如果该 i 节点对应的节点位图不存在，则出错。
128     if (!(bh=sb->s_imap[inode->i_num>>13]))
129         panic("nonexistent imap in superblock");
// 复位 i 节点对应的节点位图中的比特位，如果该比特位已经等于 0，则出错。
130     if (clear_bit(inode->i_num&8191, bh->b_data))
131         printk("free_inode: bit already cleared. \n|r");
// 置 i 节点位图所在缓冲区已修改标志，并清空该 i 节点结构所占内存区。
132     bh->b_dirt = 1;
133     memset(inode, 0, sizeof(*inode));

```



```

134 }
135
136 // 为设备 dev 建立一个新 i 节点。返回该新 i 节点的指针。
137 // 在内存 i 节点表中获取一个空闲 i 节点表项，并从 i 节点位图中找一个空闲 i 节点。
138 struct m_inode * new_inode(int dev)
139 {
140     struct m_inode * inode;
141     struct super_block * sb;
142     struct buffer_head * bh;
143     int i, j;
144
145     // 从内存 i 节点表(inode_table)中获取一个空闲 i 节点项(inode)。
146     if (!(inode=get_empty_inode()))
147         return NULL;
148     // 读取指定设备的超级块结构。
149     if (!(sb = get_super(dev)))
150         panic("new_inode with unknown device");
151     // 扫描 i 节点位图，寻找首个 0 比特位，寻找空闲节点，获取放置该 i 节点的节点号。
152     j = 8192;
153     for (i=0 ; i<8 ; i++)
154         if (bh=sb->s_imap[i])
155             if ((j=find_first_zero(bh->b_data))<8192)
156                 break;
157     // 如果全部扫描完还没找到，或者位图所在的缓冲块无效(bh=NULL)则 返回 0，退出(没有空闲 i 节点)。
158     if (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
159         iput(inode);
160         return NULL;
161     }
162     // 置位对应新 i 节点的 i 节点位图相应比特位，如果已经置位，则出错。
163     if (set_bit(j, bh->b_data))
164         panic("new_inode: bit already set");
165     // 置 i 节点位图所在缓冲区已修改标志。
166     bh->b_dirt = 1;
167     // 初始化该 i 节点结构。
168     inode->i_count=1; // 引用计数。
169     inode->i_nlinks=1; // 文件目录项链接数。
170     inode->i_dev=dev; // i 节点所在的设备号。
171     inode->i_uid=current->euid; // i 节点所属用户 id。
172     inode->i_gid=current->egid; // 组 id。
173     inode->i_dirt=1; // 已修改标志置位。
174     inode->i_num = j + i*8192; // 对应设备中的 i 节点号。
175     inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME; // 设置时间。
176     return inode; // 返回该 i 节点指针。
177 }
178
179

```

### 9.3.3 其它信息



## 9.4 inode.c 程序

### 9.4.1 功能描述

该程序含有处理 i 节点的函数。在注释中，“逻辑块”、“磁盘块”或“区段”的含义相同。

### 9.4.2 代码注释

列表 linux/fs/inode.c 程序

```

1  /*
2  *  linux/fs/inode.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <string.h>      // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8  #include <sys/stat.h>    // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
9
10 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/mm.h>     // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
13 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14
15 struct m_inode inode_table[NR_INODE]={0,},}; // 内存中 i 节点表 (NR_INODE=32 项)。
16
17 static void read_inode(struct m_inode * inode);
18 static void write_inode(struct m_inode * inode);
19
    // 等待指定的 i 节点可用。
    // 如果 i 节点已被锁定，则将当前任务置为不可中断的等待状态。直到该 i 节点解锁。
20 static inline void wait_on_inode(struct m_inode * inode)
21 {
22     cli();
23     while (inode->i_lock)
24         sleep_on(&inode->i_wait);
25     sti();
26 }
27
    // 对指定的 i 节点上锁（锁定指定的 i 节点）。
    // 如果 i 节点已被锁定，则将当前任务置为不可中断的等待状态。直到该 i 节点解锁，然后对其上锁。
28 static inline void lock_inode(struct m_inode * inode)
29 {
30     cli();
31     while (inode->i_lock)
32         sleep_on(&inode->i_wait);
33     inode->i_lock=1; // 置锁定标志。
34     sti();
35 }
36

```

```

    /// 对指定的 i 节点解锁。
    // 复位 i 节点的锁定标志，并明确地唤醒等待此 i 节点的进程。
37 static inline void unlock_inode(struct m_inode * inode)
38 {
39     inode->i_lock=0;
40     wake_up(&inode->i_wait);
41 }
42
    /// 释放内存中设备 dev 的所有 i 节点。
    // 扫描内存中的 i 节点表数组，如果是指定设备使用的 i 节点就释放之。
43 void invalidate_inodes(int dev)
44 {
45     int i;
46     struct m_inode * inode;
47
48     inode = 0+inode_table;           // 让指针首先指向 i 节点表指针数组首项。
49     for(i=0 ; i<NR_INODE ; i++,inode++) { // 扫描 i 节点表指针数组中的所有 i 节点。
50         wait_on_inode(inode);           // 等待该 i 节点可用（解锁）。
51         if (inode->i_dev == dev) {        // 如果是指定设备的 i 节点，则
52             if (inode->i_count)           // 如果其引用数不为 0，则显示出错警告；
53                 printk("inode in use on removed disk\n\r");
54             inode->i_dev = inode->i_dirt = 0; // 释放该 i 节点(置设备号为 0 等)。
55         }
56     }
57 }
58
    /// 同步所有 i 节点。
    // 同步内存与设备上的所有 i 节点信息。
59 void sync_inodes(void)
60 {
61     int i;
62     struct m_inode * inode;
63
64     inode = 0+inode_table;           // 让指针首先指向 i 节点表指针数组首项。
65     for(i=0 ; i<NR_INODE ; i++,inode++) { // 扫描 i 节点表指针数组。
66         wait_on_inode(inode);           // 等待该 i 节点可用（解锁）。
67         if (inode->i_dirt && !inode->i_pipe) // 如果该 i 节点已修改且不是管道节点，
68             write_inode(inode);         // 则写盘。
69     }
70 }
71
    /// 块映射处理操作。(block 位图处理函数, bmap - block map)
    // 参数: inode - i 节点指针; block - 数据块号; create - 创建标志。
    // 如果创建标志置位，则在对应逻辑块不存在时就申请新磁盘块。
    // 返回 block 数据块对应设备上的逻辑块号。
72 static int bmap(struct m_inode * inode,int block,int create)
73 {
74     struct buffer_head * bh;
75     int i;
76
    // 如果块号小于 0，则死机。
77     if (block<0)
78         panic("_bmap: block<0");

```

```

// 如果块号大于直接块数 + 间接块数 + 二次间接块数, 超出文件系统表示范围, 则死机。
99         if (block >= 7+512+512*512)
100             panic("_bmap: block>big");

// 如果该块号小于 7, 则使用直接块表示。
101         if (block<7) {
// 如果创建标志置位, 并且 i 节点中对应该块的逻辑块 (区段) 字段为 0, 则向相应设备申请一磁盘
// 块 (逻辑块, 区段), 并将磁盘上的实际逻辑块号填入逻辑块字段中。然后设置 i 节点修改时间,
// 置 i 节点已修改标志。最后返回磁盘上实际逻辑块号。
102             if (create && !inode->i_zone[block])
103                 if (inode->i_zone[block]=new\_block(inode->i_dev)) {
104                     inode->i_ctime=CURRENT\_TIME;
105                     inode->i_dirt=1;
106                 }
107             return inode->i_zone[block];
108         }

// 如果该块号>=7, 则如果该块小于 7+512, 则说明是一次间接块。下面对一次间接块进行处理。
109         block -= 7;
110         if (block<512) {
// 如果是创建, 并且该 i 节点中对应该间接块字段为 0, 表明文件是首次使用间接块, 则需申请
// 一磁盘块用于存放间接块信息, 并将此实际磁盘块号填入间接块字段中。然后设置 i 节点
// 已修改标志和修改时间。
111             if (create && !inode->i_zone[7])
112                 if (inode->i_zone[7]=new\_block(inode->i_dev)) {
113                     inode->i_dirt=1;
114                     inode->i_ctime=CURRENT\_TIME;
115                 }
// 若此时 i 节点间接块字段中为 0, 表明申请磁盘块失败, 返回 0 退出。
116             if (!inode->i_zone[7])
117                 return 0;
// 读取设备上的一次间接块。
118             if (!(bh = bread(inode->i_dev, inode->i_zone[7])))
119                 return 0;
// 取该间接块上第 block 项中的逻辑块号。
120             i = ((unsigned short *) (bh->b_data))[block];
// 如果是创建并且间接块的第 block 项中的逻辑块号为 0 的话, 则申请一磁盘块 (逻辑块), 并让
// 间接块中的第 block 项等于该新逻辑块号。然后置位间接块的已修改标志。
121             if (create && !i)
122                 if (i=new\_block(inode->i_dev)) {
123                     ((unsigned short *) (bh->b_data))[block]=i;
124                     bh->b_dirt=1;
125                 }
// 最后释放该间接块, 返回磁盘上新申请的对应 block 的逻辑块的块号。
126             brelse(bh);
127             return i;
128         }

// 程序运行到此, 表明数据块是二次间接块, 处理过程与一次间接块类似。下面是对二次间接块的处理。
// 将 block 再减去间接块所容纳的块数 (512)。
129         block -= 512;
// 如果是新创建并且 i 节点的二次间接块字段为 0, 则需申请一磁盘块用于存放二次间接块的一级块
// 信息, 并将此实际磁盘块号填入二次间接块字段中。之后置 i 节点已修改变化和修改时间。

```

```

110         if (create && !inode->i_zone[8])
111             if (inode->i_zone[8]=new\_block(inode->i_dev)) {
112                 inode->i_dirt=1;
113                 inode->i_ctime=CURRENT\_TIME;
114             }
115 // 若此时 i 节点二次间接块字段为 0，表明申请磁盘块失败，返回 0 退出。
116         if (!inode->i_zone[8])
117             return 0;
118 // 读取该二次间接块的一级块。
119         if (!(bh=bread(inode->i_dev, inode->i_zone[8])))
120             return 0;
121 // 取该二次间接块的一级块上第 (block/512) 项中的逻辑块号。
122         i = ((unsigned short *)bh->b_data)[block>>9];
123 // 如果是创建并且二次间接块的一级块上第 (block/512) 项中的逻辑块号为 0 的话，则需申请一磁盘
124 // 块（逻辑块）作为二次间接块的二级块，并让二次间接块的一级块中第 (block/512) 项等于该二级
125 // 块的块号。然后置位二次间接块的一级块已修改标志。并释放二次间接块的一级块。
126         if (create && !i)
127             if (i=new\_block(inode->i_dev)) {
128                 ((unsigned short *) (bh->b_data))[block>>9]=i;
129                 bh->b_dirt=1;
130             }
131         brelse(bh);
132 // 如果二次间接块的二级块块号为 0，表示申请磁盘块失败，返回 0 退出。
133         if (!i)
134             return 0;
135 // 读取二次间接块的二级块。
136         if (!(bh=bread(inode->i_dev, i)))
137             return 0;
138 // 取该二级块上第 block 项中的逻辑块号。（与上 511 是为了限定 block 值不超过 511）
139         i = ((unsigned short *)bh->b_data)[block&511];
140 // 如果是创建并且二级块的第 block 项中的逻辑块号为 0 的话，则申请一磁盘块（逻辑块），作为
141 // 最终存放数据信息的块。并让二级块中的第 block 项等于该新逻辑块块号 (i)。然后置位二级块的
142 // 已修改标志。
143         if (create && !i)
144             if (i=new\_block(inode->i_dev)) {
145                 ((unsigned short *) (bh->b_data))[block&511]=i;
146                 bh->b_dirt=1;
147             }
148 // 最后释放该二次间接块的二级块，返回磁盘上新申请的对应 block 的逻辑块的块号。
149         brelse(bh);
150         return i;
151 }
152
153 // 根据 i 节点信息取数据块 block 在设备上对应的逻辑块号。
154 int bmap(struct m\_inode * inode, int block)
155 {
156     return bmap(inode, block, 0);
157 }
158
159 // 创建数据块 block 在设备上对应的逻辑块，并返回在设备上的逻辑块号。
160 int create\_block(struct m\_inode * inode, int block)
161 {
162     return bmap(inode, block, 1);
163 }

```

```

148 }
149
150 void iput(struct m_inode * inode)
151 {
152     if (!inode)
153         return;
154     wait_on_inode(inode);    // 等待 inode 节点解锁(如果已上锁的话)。
155     if (!inode->i_count)
156         panic("iput: trying to free free inode");
157     // 如果是管道 i 节点, 则唤醒等待该管道的进程, 引用次数减 1, 如果还有引用则返回。否则释放
158     // 管道占用的内存页面, 并复位该节点的引用计数值、已修改标志和管道标志, 并返回。
159     if (inode->i_pipe) {
160         wake_up(&inode->i_wait);
161         if (--inode->i_count)
162             return;
163         free_page(inode->i_size);
164         inode->i_count=0;
165         inode->i_dirt=0;
166         inode->i_pipe=0;
167         return;
168     }
169     // 如果 i 节点对应的设备号=0, 则将此节点的引用计数递减 1, 返回。
170     if (!inode->i_dev) {
171         inode->i_count--;
172         return;
173     }
174     // 如果是块设备文件的 i 节点, 此时逻辑块字段 0 中是设备号, 则刷新该设备。并等待 i 节点解锁。
175     if (S_ISBLK(inode->i_mode)) {
176         sync_dev(inode->i_zone[0]);
177         wait_on_inode(inode);
178     }
179 repeat:
180     // 如果 i 节点的引用计数大于 1, 则递减 1。
181     if (inode->i_count>1) {
182         inode->i_count--;
183         return;
184     }
185     // 如果 i 节点的链接数为 0, 则释放该 i 节点的所有逻辑块, 并释放该 i 节点。
186     if (!inode->i_nlinks) {
187         truncate(inode);
188         free_inode(inode);
189         return;
190     }
191     // 如果该 i 节点已作过修改, 则更新该 i 节点, 并等待该 i 节点解锁。
192     if (inode->i_dirt) {
193         write_inode(inode);    /* we can sleep - so do again */
194         wait_on_inode(inode);
195         goto repeat;
196     }
197     // i 节点引用计数递减 1。
198     inode->i_count--;
199     return;

```

```

192 }
193
194 //从 i 节点表(inode_table)中获取一个空闲 i 节点项。
195 // 寻找引用计数 count 为 0 的 i 节点, 并将其写盘后清零, 返回其指针。
196 struct m_inode * get_empty_inode(void)
197 {
198     struct m_inode * inode;
199     static struct m_inode * last_inode = inode_table; // last_inode 指向 i 节点表第一项。
200     int i;
201
202     do {
203         // 扫描 i 节点表。
204         inode = NULL;
205         for (i = NR_INODE; i ; i--) {
206             // 如果 last_inode 已经指向 i 节点表的最后 1 项之后, 则让其重新指向 i 节点表开始处。
207             if (++last_inode >= inode_table + NR_INODE)
208                 last_inode = inode_table;
209             // 如果 last_inode 所指向的 i 节点的计数值为 0, 则说明可能找到空闲 i 节点项。让 inode 指向
210             // 该 i 节点。如果该 i 节点的已修改标志和锁定标志均为 0, 则我们可以使用该 i 节点, 于是退出循环。
211             if (!last_inode->i_count) {
212                 inode = last_inode;
213                 if (!inode->i_dirt && !inode->i_lock)
214                     break;
215             }
216         }
217         // 如果没有找到空闲 i 节点(inode=NULL), 则将整个 i 节点表打印出来供调试使用, 并死机。
218         if (!inode) {
219             for (i=0 ; i<NR_INODE ; i++)
220                 printk("%04x: %6d\t", inode_table[i].i_dev,
221                     inode_table[i].i_num);
222             panic("No free inodes in mem");
223         }
224         // 等待该 i 节点解锁 (如果又被上锁的话)。
225         wait_on_inode(inode);
226         // 如果该 i 节点已修改标志被置位的话, 则将该 i 节点刷新, 并等待该 i 节点解锁。
227         while (inode->i_dirt) {
228             write_inode(inode);
229             wait_on_inode(inode);
230         }
231         } while (inode->i_count); // 如果 i 节点又被其它占用的话, 则重新寻找空闲 i 节点。
232         // 已找到空闲 i 节点项。则将该 i 节点项内容清零, 并置引用标志为 1, 返回该 i 节点指针。
233         memset(inode, 0, sizeof(*inode));
234         inode->i_count = 1;
235         return inode;
236     }
237
238 // 获取管道节点。返回为 i 节点指针 (如果是 NULL 则失败)。
239 // 首先扫描 i 节点表, 寻找一个空闲 i 节点项, 然后取得一页空闲内存供管道使用。
240 // 然后将得到的 i 节点的引用计数置为 2 (读者和写者), 初始化管道头和尾, 置 i 节点的管道类型表示。
241 struct m_inode * get_pipe_inode(void)
242 {
243     struct m_inode * inode;
244

```

```

232     if (!(inode = get\_empty\_inode\(\)))           // 如果找不到空闲 i 节点则返回 NULL。
233         return NULL;
234     if (!(inode->i_size=get\_free\_page\(\))) { // 节点的 i_size 字段指向缓冲区。
235         inode->i_count = 0;                // 如果已没有空闲内存，则
236         return NULL;                      // 释放该 i 节点，并返回 NULL。
237     }
238     inode->i_count = 2;    /* sum of readers/writers */ /* 读/写两者总计 */
239     PIPE\_HEAD(*inode) = PIPE\_TAIL(*inode) = 0;
240     inode->i_pipe = 1;    // 置节点为管道使用的标志。
241     return inode;        // 返回 i 节点指针。
242 }
243
244 // 从设备上读取指定节点号的 i 节点。
245 // nr - i 节点号。
246 struct m\_inode * iget(int dev, int nr)
247 {
248     struct m\_inode * inode, * empty;
249     if (!dev)
250         panic("iget with dev==0");
251 // 从 i 节点表中取一个空闲 i 节点。
252     empty = get\_empty\_inode();
253 // 扫描 i 节点表。寻找指定节点号的 i 节点。并递增该节点的引用次数。
254     inode = inode\_table;
255     while (inode < NR\_INODE+inode\_table) {
256 // 如果当前扫描的 i 节点的设备号不等于指定的设备号或者节点号不等于指定的节点号，则继续扫描。
257         if (inode->i_dev != dev || inode->i_num != nr) {
258             inode++;
259             continue;
260         }
261 // 找到指定设备号和节点号的 i 节点，等待该节点解锁（如果已上锁的话）。
262     wait\_on\_inode(inode);
263 // 在等待该节点解锁的阶段，节点表可能会发生变化，所以再次判断，如果发生了变化，则再次重新
264 // 扫描整个 i 节点表。
265     if (inode->i_dev != dev || inode->i_num != nr) {
266         inode = inode\_table;
267         continue;
268     }
269 // 将该 i 节点引用计数增 1。
270     inode->i_count++;
271     if (inode->i_mount) {
272         int i;
273 // 如果该 i 节点是其它文件系统的安装点，则在超级块表中搜寻安装在此 i 节点的超级块。如果没有
274 // 找到，则显示出错信息，并释放函数开始获取的空闲节点，返回该 i 节点指针。
275         for (i = 0 ; i<NR\_SUPER ; i++)
276             if (super\_block[i].s_imount==inode)
277                 break;
278         if (i >= NR\_SUPER) {
279             printk("Mounted inode hasn't got sb\n");
280             if (empty)
281                 iput(empty);
282             return inode;

```



```

274         }
    // 将该 i 节点写盘。从安装在此 i 节点文件系统的超级块上取设备号，并令 i 节点号为 1。然后重新
    // 扫描整个 i 节点表，取该被安装文件系统的根节点。
275         iput(inode);
276         dev = super_block[i].s_dev;
277         nr = ROOT_INO;
278         inode = inode_table;
279         continue;
280     }
    // 已经找到相应的 i 节点，因此放弃临时申请的空闲节点，返回该找到的 i 节点。
281     if (empty)
282         iput(empty);
283     return inode;
284 }
    // 如果在 i 节点表中没有找到指定的 i 节点，则利用前面申请的空闲 i 节点在 i 节点表中建立该节点。
    // 并从相应设备上读取该 i 节点信息。返回该 i 节点。
285     if (!empty)
286         return (NULL);
287     inode=empty;
288     inode->i_dev = dev;
289     inode->i_num = nr;
290     read_inode(inode);
291     return inode;
292 }
293
    //// 从设备上读取指定 i 节点的信息到内存中。
294 static void read_inode(struct m_inode * inode)
295 {
296     struct super_block * sb;
297     struct buffer_head * bh;
298     int block;
299
    // 首先锁定该 i 节点，取该节点所在设备的超级块。
300     lock_inode(inode);
301     if (!(sb=get_super(inode->i_dev)))
302         panic("trying to read inode without dev");
    // 该 i 节点所在的逻辑块号 = (启动块+超级块) + i 节点位图占用的块数 + 逻辑块位图占用的块数 +
    // (i 节点号-1)/每块含有的 i 节点数。
303     block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
304         (inode->i_num-1)/INODES_PER_BLOCK;
    // 从设备上读取该 i 节点所在的逻辑块，并将该 inode 指针指向对应 i 节点信息。
305     if (!(bh=bread(inode->i_dev, block)))
306         panic("unable to read i-node block");
307     *(struct d_inode *)inode =
308         ((struct d_inode *)bh->b_data)
309         [(inode->i_num-1)%INODES_PER_BLOCK];
    // 最后释放读入的缓冲区，并解锁该 i 节点。
310     brelse(bh);
311     unlock_inode(inode);
312 }
313
    //// 将指定 i 节点信息写入设备（写入对应的缓冲区中，待缓冲区刷新时会写入盘中）。
314 static void write_inode(struct m_inode * inode)

```

```

315 {
316     struct super\_block * sb;
317     struct buffer\_head * bh;
318     int block;
319
320     // 首先锁定该 i 节点，如果该 i 节点没有被修改过或者该 i 节点的设备号等于零，则解锁该 i 节点，
321     // 并退出。
322     lock\_inode(inode);
323     if (!inode->i_dirt || !inode->i_dev) {
324         unlock\_inode(inode);
325         return;
326     }
327     // 获取该 i 节点的超级块。
328     if (!(sb=get\_super(inode->i_dev)))
329         panic("trying to write inode without device");
330     // 该 i 节点所在的逻辑块号 = (启动块+超级块) + i 节点位图占用的块数 + 逻辑块位图占用的块数 +
331     // (i 节点号-1)/每块含有的 i 节点数。
332     block = 2 + sb->s_imap_blocks + sb->s_zmap_blocks +
333             (inode->i_num-1)/INODES\_PER\_BLOCK;
334     // 从设备上读取该 i 节点所在的逻辑块。
335     if (!(bh=bread(inode->i_dev, block)))
336         panic("unable to read i-node block");
337     // 将该逻辑块对应该 i 节点的项指向该 i 节点信息。
338     ((struct d\_inode *)bh->b_data)
339     [(inode->i_num-1)%INODES\_PER\_BLOCK] =
340     *(struct d\_inode *)inode;
341     // 置缓冲区已修改标志，而 i 节点修改标志置零。然后释放该含有 i 节点的缓冲区，并解锁该 i 节点。
342     bh->b_dirt=1;
343     inode->i_dirt=0;
344     brelse(bh);
345     unlock\_inode(inode);
346 }
347
348
349

```

### 9.4.3 其它信息

## 9.5 buffer.c 文件

### 9.5.1 功能描述

该文件中的函数主要用于对设备高速缓冲的操作和处理。

空闲缓冲区链表中的缓冲区，并不是都是空闲的。!! 只有当被写盘刷新、解锁且没有其它进程引用时（引用计数=0），才能挪作它用。

### 9.5.2 代码注释

列表 linux/fs/buffer.c 程序

[1](#) /\*

```

2  * linux/fs/buffer.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * 'buffer.c' implements the buffer-cache functions. Race-conditions have
9  * been avoided by NEVER letting a interrupt change a buffer (except for the
10 * data, of course), but instead letting the caller do it. NOTE! As interrupts
11 * can wake up a caller, some cli-sti sequences are needed to check for
12 * sleep-on-calls. These should be extremely quick, though (I hope).
13 */
14 /*
15  * 'buffer.c' 用于实现缓冲区高速缓存功能。通过不让中断过程改变缓冲区，而是让调用者
16  * 来执行，避免了竞争条件（当然除改变数据以外）。注意！由于中断可以唤醒一个调用者，
17  * 因此就需要开关中断指令（cli-sti）序列来检测等待调用返回。但需要非常地快（希望是这样）。
18  */
19 /*
20  * NOTE! There is one discordant note here: checking floppies for
21  * disk change. This is where it fits best, I think, as it should
22  * invalidate changed floppy-disk-caches.
23  */
24 /*
25  * 注意！这里有一个程序应不属于这里：检测软盘是否更换。但我想这里是
26  * 放置该程序最好的地方了，因为它需要使已更换软盘缓冲失效。
27  */
28
29 #include <stdarg.h>          // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个
30                             // 类型(va_list)和三个宏(va_start, va_arg 和 va_end)，用于
31                             // vsprintf、vprintf、vfprintf 函数。
32
33 #include <linux/config.h>    // 内核配置头文件。定义键盘语言和硬盘类型(HD_TYPE)可选项。
34 #include <linux/sched.h>     // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
35                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
36 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
37 #include <asm/system.h>      // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
38 #include <asm/io.h>          // io 头文件。定义硬件端口输入/输出宏汇编语句。
39
40 extern int end;              // 由连接程序 ld 生成的表明程序末端的变量。[??]
41 struct buffer_head * start_buffer = (struct buffer_head *) &end;
42 struct buffer_head * hash_table[NR_HASH];          // NR_HASH = 307 项。
43 static struct buffer_head * free_list;
44 static struct task_struct * buffer_wait = NULL;
45 int NR_BUFFERS = 0;
46
47 //// 等待指定缓冲区解锁。
48 static inline void wait_on_buffer(struct buffer_head * bh)
49 {
50     cli();                      // 关中断。
51     while (bh->b_lock)          // 如果已被上锁，则进程进入睡眠，等待其解锁。
52         sleep_on(&bh->b_wait);
53     sti();                      // 开中断。

```

```

42 }
43
44 // 系统调用。同步设备和内存高速缓冲中数据。
45 int sys_sync(void)
46 {
47     int i;
48     struct buffer_head * bh;
49     sync_inodes();          /* write out inodes into buffers */ /*将 i 节点写入高速缓冲
*/
    // 扫描所有高速缓冲区，对于已被修改的缓冲块产生写盘请求，将缓冲中数据与设备中同步。
50     bh = start_buffer;
51     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
52         wait_on_buffer(bh);          // 等待缓冲区解锁（如果已上锁的话）。
53         if (bh->b_dirt)
54             ll_rw_block(WRITE, bh);    // 产生写设备块请求。
55     }
56     return 0;
57 }
58
59 // 对指定设备进行高速缓冲数据与设备上数据的同步操作。
60 int sync_dev(int dev)
61 {
62     int i;
63     struct buffer_head * bh;
64
65     bh = start_buffer;
66     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
67         if (bh->b_dev != dev)
68             continue;
69         wait_on_buffer(bh);
70         if (bh->b_dev == dev && bh->b_dirt)
71             ll_rw_block(WRITE, bh);
72     }
73     sync_inodes();          // 将 i 节点数据写入高速缓冲。
74     bh = start_buffer;
75     for (i=0 ; i<NR_BUFFERS ; i++,bh++) {
76         if (bh->b_dev != dev)
77             continue;
78         wait_on_buffer(bh);
79         if (bh->b_dev == dev && bh->b_dirt)
80             ll_rw_block(WRITE, bh);
81     }
82     return 0;
83 }
84
85 // 使指定设备在高速缓冲区中的数据无效。
86 // 扫描高速缓冲中的所有缓冲块，对于指定设备的缓冲区，复位其有效(更新)标志和已修改标志。
87 void inline invalidate_buffers(int dev)
88 {
89     int i;
90     struct buffer_head * bh;

```

```

89     bh = start\_buffer;
90     for (i=0 ; i<NR\_BUFFERS ; i++,bh++) {
91         if (bh->b_dev != dev)                // 如果不是指定设备的缓冲块，则
92             continue;                        // 继续扫描下一块。
93         wait\_on\_buffer(bh);                    // 等待该缓冲区解锁（如果已被上锁）。
// 由于进程执行过睡眠等待，所以需要再判断一下缓冲区是否是指定设备的。
94         if (bh->b_dev == dev)
95             bh->b_uptodate = bh->b_dirt = 0;
96     }
97 }
98
99 /*
100  * This routine checks whether a floppy has been changed, and
101  * invalidates all buffer-cache-entries in that case. This
102  * is a relatively slow routine, so we have to try to minimize using
103  * it. Thus it is called only upon a 'mount' or 'open'. This
104  * is the best way of combining speed and utility, I think.
105  * People changing diskettes in the middle of an operation deserve
106  * to loose :-))
107  *
108  * NOTE! Although currently this is only for floppies, the idea is
109  * that any additional removable block-device will use this routine,
110  * and that mount/open needn't know that floppies/whatever are
111  * special.
112  */
// 该子程序检查一个软盘是否已经被更换，如果已经更换就使高速缓冲中与该软驱
// 对应的所有缓冲区无效。该子程序相对来说较慢，所以我们要尽量少使用它。
// 所以仅在执行'mount'或'open'时才调用它。我想这是将速度和实用性相结合的
// 最好方法。若在操作过程当中更换软盘，会导致数据的丢失，这是咎由自取☺。
// 注意！尽管目前该子程序仅用于软盘，以后任何可移动介质的块设备都将使用该
// 程序，mount/open 操作是不需要知道是否是软盘或其它什么特殊介质的。
//// 检查磁盘是否更换，如果已更换就使对应高速缓冲区无效。
113 void check\_disk\_change(int dev)
114 {
115     int i;
116
117     // 是软盘设备吗？如果不是则退出。
118     if (MAJOR(dev) != 2)
119         return;
120     // 测试对应软盘是否已更换，如果没有则退出。
121     if (!floppy\_change(dev & 0x03))
122         return;
123     // 软盘已经更换，所以释放对应设备的 i 节点位图和逻辑块位图所占的高速缓冲区；并使该设备的
124     // i 节点和数据块信息所占的高速缓冲区无效。
125     for (i=0 ; i<NR\_SUPER ; i++)
126         if (super\_block[i].s_dev == dev)
127             put\_super(super\_block[i].s_dev);
128     invalidate\_inodes(dev);
129     invalidate\_buffers(dev);
130 }

```

```

127 // hash 函数和 hash 表项的计算宏定义。
128 #define hashfn(dev,block) (((unsigned)(dev^block))%NR\_HASH)
129 #define hash(dev,block) hash\_table[hashfn(dev,block)]
130
131 // 从 hash 队列和空闲缓冲队列中移走指定的缓冲块。
132 static inline void remove\_from\_queues(struct buffer\_head * bh)
133 {
134     /* remove from hash-queue */
135     /* 从 hash 队列中移除缓冲块 */
136     if (bh->b_next)
137         bh->b_next->b_prev = bh->b_prev;
138     if (bh->b_prev)
139         bh->b_prev->b_next = bh->b_next;
140     // 如果该缓冲区是该队列的头一个块，则让 hash 表的对应项指向本队列中的下一个缓冲区。
141     if (hash(bh->b_dev, bh->b_blocknr) == bh)
142         hash(bh->b_dev, bh->b_blocknr) = bh->b_next;
143     /* remove from free list */
144     /* 从空闲缓冲区表中移除缓冲块 */
145     if (!(bh->b_prev_free) || !(bh->b_next_free))
146         panic("Free block list corrupted");
147     bh->b_prev_free->b_next_free = bh->b_next_free;
148     bh->b_next_free->b_prev_free = bh->b_prev_free;
149     // 如果空闲链表头指向本缓冲区，则让其指向下一缓冲区。
150     if (free\_list == bh)
151         free\_list = bh->b_next_free;
152 }
153
154 // 将指定缓冲区插入空闲链表尾并放入 hash 队列中。
155 static inline void insert\_into\_queues(struct buffer\_head * bh)
156 {
157     /* put at end of free list */
158     /* 放在空闲链表末尾处 */
159     bh->b_next_free = free\_list;
160     bh->b_prev_free = free\_list->b_prev_free;
161     free\_list->b_prev_free->b_next_free = bh;
162     free\_list->b_prev_free = bh;
163     /* put the buffer in new hash-queue if it has a device */
164     /* 如果该缓冲块对应一个设备，则将其插入新 hash 队列中 */
165     bh->b_prev = NULL;
166     bh->b_next = NULL;
167     if (!bh->b_dev)
168         return;
169     bh->b_next = hash(bh->b_dev, bh->b_blocknr);
170     hash(bh->b_dev, bh->b_blocknr) = bh;
171     bh->b_next->b_prev = bh;
172 }
173
174 // 在高速缓冲中寻找给定设备和指定块的缓冲区块。
175 // 如果找到则返回缓冲区块的指针，否则返回 NULL。
176 static struct buffer\_head * find\_buffer(int dev, int block)
177 {
178     struct buffer\_head * tmp;

```

```

169
170     for (tmp = hash(dev, block) ; tmp != NULL ; tmp = tmp->b_next)
171         if (tmp->b_dev==dev && tmp->b_blocknr==block)
172             return tmp;
173     return NULL;
174 }
175
176 /*
177  * Why like this, I hear you say... The reason is race-conditions.
178  * As we don't lock buffers (unless we are reading them, that is),
179  * something might happen to it while we sleep (ie a read-error
180  * will force it bad). This shouldn't really happen currently, but
181  * the code is ready.
182  */
183
184 /*
185  * 代码为什么会是这样子的？我听见你问... 原因是竞争条件。由于我们没有对
186  * 缓冲区上锁（除非我们正在读取它们中的数据），那么当我们（进程）睡眠时
187  * 缓冲区可能会发生一些问题（例如一个读错误将导致该缓冲区出错）。目前
188  * 这种情况实际上是不会发生的，但处理的代码已经准备好了。
189  */
190
191 struct buffer_head * get_hash_table(int dev, int block)
192 {
193     struct buffer_head * bh;
194
195     for (;;) {
196         // 在高速缓冲中寻找给定设备和指定块的缓冲区，如果没有找到则返回 NULL，退出。
197         if (!(bh=find_buffer(dev, block)))
198             return NULL;
199         // 对该缓冲区增加引用计数，并等待该缓冲区解锁（如果已被上锁）。
200         bh->b_count++;
201         wait_on_buffer(bh);
202         // 由于经过了睡眠状态，因此有必要再验证该缓冲区块的正确性，并返回缓冲区头指针。
203         if (bh->b_dev == dev && bh->b_blocknr == block)
204             return bh;
205         // 如果该缓冲区所属的设备号或块号在睡眠时发生了改变，则撤消对它的引用计数，重新寻找。
206         bh->b_count--;
207     }
208 }
209
210 /*
211  * Ok, this is getblk, and it isn't very clear, again to hinder
212  * race-conditions. Most of the code is seldom used, (ie repeating),
213  * so it should be much more efficient than it looks.
214  *
215  * The algorithm is changed: hopefully better, and an elusive bug removed.
216  */
217
218 /*
219  * OK, 下面是 getblk 函数，该函数的逻辑并不是很清晰，同样也是因为要考虑
220  * 竞争条件问题。其中大部分代码很少用到，（例如重复操作语句），因此它应该
221  * 比看上去的样子有效得多。
222  *
223  * 算法以及作了改变：希望能更好，而且一个难以琢磨的错误已经去除。

```



```

    */
    // 下面宏定义用于同时判断缓冲区的修改标志和锁定标志，并且定义修改标志的权重要比锁定标志大。
205 #define BADNESS(bh) (((bh)->b_dirt<<1)+(bh)->b_lock)
    // 取高速缓冲中指定的缓冲区。
    // 检查所指定的缓冲区是否已经在高速缓冲中，如果不在，就需要在高速缓冲中建立一个对应的新项。
    // 返回相应缓冲区头指针。
206 struct buffer_head * getblk(int dev, int block)
207 {
208     struct buffer_head * tmp, * bh;
209
210 repeat:
    // 搜索 hash 表，如果指定块已经在高速缓冲中，则返回对应缓冲区头指针，退出。
211     if (bh = get_hash_table(dev, block))
212         return bh;
    // 扫描空闲数据块链表，寻找空闲缓冲区。
    // 首先让 tmp 指向空闲链表的第一个空闲缓冲区头。
213     tmp = free_list;
214     do {
    // 如果该缓冲区正被使用（引用计数不等于 0），则继续扫描下一项。
215         if (tmp->b_count)
216             continue;
    // 如果缓冲区头指针 bh 为空，或者 tmp 所指缓冲区头的标志(修改、锁定)少于(小于)bh 头的标志，
    // 则让 bh 指向该 tmp 缓冲区头。如果该 tmp 缓冲区头表明缓冲区既没有修改也没有锁定标志置位，
    // 则说明已为指定设备上的块取得对应的高速缓冲区，则退出循环。
217         if (!bh || BADNESS(tmp)<BADNESS(bh)) {
218             bh = tmp;
219             if (!BADNESS(tmp))
220                 break;
221         }
222 /* and repeat until we find something good */ /* 重复操作直到找到适合的缓冲区 */
223     } while ((tmp = tmp->b_next_free) != free_list);
    // 如果所有缓冲区都正被使用（所有缓冲区的头部引用计数都>0），则睡眠，等待有空闲的缓冲区可用。
224     if (!bh) {
225         sleep_on(&buffer_wait);
226         goto repeat;
227     }
    // 等待该缓冲区解锁（如果已被上锁的话）。
228     wait_on_buffer(bh);
    // 如果该缓冲区又被其它任务使用的话，只好重复上述过程。
229     if (bh->b_count)
230         goto repeat;
    // 如果该缓冲区已被修改，则将数据写盘，并再次等待缓冲区解锁。如果该缓冲区又被其它任务使用
    // 的话，只好再重复上述过程。
231     while (bh->b_dirt) {
232         sync_dev(bh->b_dev);
233         wait_on_buffer(bh);
234         if (bh->b_count)
235             goto repeat;
236     }
237 /* NOTE!! While we slept waiting for this block, somebody else might */
238 /* already have added "this" block to the cache. check it */
    // 注意！！当进程为了等待该缓冲块而睡眠时，其它进程可能已经将该缓冲块
    // 加入进高速缓冲中，所以要对此进行检查。*/

```

```

// 在高速缓冲 hash 表中检查指定设备和块的缓冲区是否已经被加入进去。如果是的话，就再次重复
// 上述过程。
239     if (find\_buffer(dev, block))
240         goto repeat;
241 /* OK, FINALLY we know that this buffer is the only one of it's kind, */
242 /* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
/* OK, 最终我们知道该缓冲区是指定参数的唯一一块，*/
/* 而且还没有被使用 (b_count=0)，未被上锁 (b_lock=0)，并且是干净的（未被修改的）*/
// 于是让我们占用此缓冲区。置引用计数为 1，复位修改标志和有效(更新)标志。
243     bh->b_count=1;
244     bh->b_dirt=0;
245     bh->b_uptodate=0;
// 从 hash 队列和空闲块链表中移出该缓冲区头，让该缓冲区用于指定设备和其上的指定块。
246     remove\_from\_queues(bh);
247     bh->b_dev=dev;
248     bh->b_blocknr=block;
// 然后根据此新的设备号和块号重新插入空闲链表和 hash 队列新位置处。并最终返回缓冲头指针。
249     insert\_into\_queues(bh);
250     return bh;
251 }
252
//// 释放指定的缓冲区。
// 等待该缓冲区解锁。引用计数递减 1。唤醒等待空闲缓冲区的进程。
253 void brelse(struct buffer head * buf)
254 {
255     if (!buf)                // 如果缓冲头指针无效则返回。
256         return;
257     wait\_on\_buffer(buf);
258     if (!(buf->b_count--))
259         panic("Trying to free free buffer");
260     wake\_up(&buffer\_wait);
261 }
262
263 /*
264 * bread() reads a specified block and returns the buffer that contains
265 * it. It returns NULL if the block was unreadable.
266 */
/*
* 从设备上读取指定的数据块并返回含有数据的缓冲区。如果指定的块不存在
* 则返回 NULL。
*/
//// 从指定设备上读取指定的数据块。
267 struct buffer head * bread(int dev, int block)
268 {
269     struct buffer head * bh;
270
// 在高速缓冲中申请一块缓冲区。如果返回值是 NULL 指针，表示内核出错，死机。
271     if (!(bh=getblk(dev, block)))
272         panic("bread: getblk returned NULL\n");
// 如果该缓冲区中的数据是有效的（已更新的）可以直接使用，则返回。
273     if (bh->b_uptodate)
274         return bh;
// 否则调用 ll_rw_block() 函数，产生读设备块请求。并等待缓冲区解锁。

```

```

275         ll_rw_block(READ, bh);
276         wait_on_buffer(bh);
    // 如果该缓冲区已更新, 则返回缓冲区头指针, 退出。
277         if (bh->b_uptodate)
278             return bh;
    // 否则表明读设备操作失败, 释放该缓冲区, 返回 NULL 指针, 退出。
279         brelse(bh);
280         return NULL;
281     }
282
    ///// 复制内存块。
    // 从 from 地址复制一块数据到 to 位置。
283 #define COPYBLK(from, to) \
284     __asm__( "cld\n\t" \
285             "rep\n\t" \
286             "movsl\n\t" \
287             :: "c" (BLOCK_SIZE/4), "S" (from), "D" (to) \
288             : "cx", "di", "si")
289
290 /*
291  * bread_page reads four buffers into memory at the desired address. It's
292  * a function of its own, as there is some speed to be got by reading them
293  * all at the same time, not waiting for one to be read, and then another
294  * etc.
295  */
296 /*
    * bread_page 一次读四个缓冲块内容读到内存指定的地址。它是一个完整的函数,
    * 因为同时读取四块可以获得速度上的好处, 不用等着读一块, 再读一块了。
    */
    ///// 读设备上一个页面(4个缓冲区)的内容到内存指定的地址。
296 void bread_page(unsigned long address, int dev, int b[4])
297 {
298     struct buffer_head * bh[4];
299     int i;
300
    // 循环执行4次, 读一页内容。
301     for (i=0 ; i<4 ; i++)
302         if (b[i]) {
    // 取高速缓冲中指定设备和块号的缓冲区, 如果该缓冲区数据无效则产生读设备请求。
303             if (bh[i] = getblk(dev, b[i]))
304                 if (!bh[i]->b_uptodate)
305                     ll_rw_block(READ, bh[i]);
306             } else
307                 bh[i] = NULL;
    // 将4块缓冲区上的内容顺序复制到指定地址处。
308     for (i=0 ; i<4 ; i++, address += BLOCK_SIZE)
309         if (bh[i]) {
310             wait_on_buffer(bh[i]);    // 等待缓冲区解锁(如果已被上锁的话)。
311             if (bh[i]->b_uptodate)    // 如果该缓冲区中数据有效的话, 则复制。
312                 COPYBLK((unsigned long) bh[i]->b_data, address);
313             brelse(bh[i]);            // 释放该缓冲区。
314         }
315 }

```

```

316
317 /*
318  * Ok, breada can be used as bread, but additionally to mark other
319  * blocks for reading as well. End the argument list with a negative
320  * number.
321  */
322 /*
323  * OK, breada 可以象 bread 一样使用，但会另外预读一些块。该函数参数列表
324  * 需要使用一个负数来表明参数列表的结束。
325  */
326 // 从指定设备读取指定的一些块。
327 // 成功时返回第 1 块的缓冲区头指针，否则返回 NULL。
328 struct buffer head * breada(int dev, int first, ...)
329 {
330     va\_list args;
331     struct buffer head * bh, *tmp;
332
333     // 取可变参数表中第 1 个参数（块号）。
334     va\_start(args, first);
335     // 取高速缓冲中指定设备和块号的缓冲区。如果该缓冲区数据无效，则发出读设备数据块请求。
336     if (!(bh=getblk(dev, first)))
337         panic("bread: getblk returned NULL\n");
338     if (!bh->b_uptodate)
339         ll\_rw\_block(READ, bh);
340     // 然后顺序取可变参数表中其它预读块号，并作与上面同样处理，但不引用。
341     while ((first=va\_arg(args, int))>=0) {
342         tmp=getblk(dev, first);
343         if (tmp) {
344             if (!tmp->b_uptodate)
345                 ll\_rw\_block(READA, bh);
346             tmp->b_count--;
347         }
348     }
349     // 可变参数表中所有参数处理完毕。等待第 1 个缓冲区解锁（如果已被上锁）。
350     va\_end(args);
351     wait\_on\_buffer(bh);
352     // 如果缓冲区中数据有效，则返回缓冲区头指针，退出。否则释放该缓冲区，返回 NULL，退出。
353     if (bh->b_uptodate)
354         return bh;
355     brelse(bh);
356     return (NULL);
357 }
358
359 // 缓冲区初始化函数。
360 // 参数 buffer_end 是指定的缓冲区内存的末端。对于系统有 16MB 内存，则缓冲区末端设置为 4MB。
361 // 对于系统有 8MB 内存，缓冲区末端设置为 2MB。
362 void buffer\_init(long buffer_end)
363 {
364     struct buffer head * h = start\_buffer;
365     void * b;
366     int i;
367
368     // 如果缓冲区高端等于 1Mb，则由于从 640KB-1MB 被显示内存和 BIOS 占用，因此实际可用缓冲区内存

```

```

// 高端应该是 640KB。否则内存高端一定大于 1MB。
354     if (buffer_end == 1<<20)
355         b = (void *) (640*1024);
356     else
357         b = (void *) buffer_end;
// 这段代码用于初始化缓冲区，建立空闲缓冲区环链表，并获取系统中缓冲块的数目。
// h 是指向缓冲头结构的指针，而 h+1 是指向内存地址连续的下一个缓冲头地址，也可以说是指向 h
// 缓冲头的末端外。为了保证有足够长度的内存来存储一个缓冲头结构，需要 b 所指向的内存块
// 地址 >= h 缓冲头的末端，也即要 >= h+1。
358     while ( (b -= BLOCK_SIZE) >= ((void *) (h+1)) ) {
359         h->b_dev = 0;           // 使用该缓冲区的设备号。
360         h->b_dirt = 0;          // 脏标志，也即缓冲区修改标志。
361         h->b_count = 0;         // 该缓冲区引用计数。
362         h->b_lock = 0;          // 缓冲区锁定标志。
363         h->b_uptodate = 0;       // 缓冲区更新标志（或称数据有效标志）。
364         h->b_wait = NULL;        // 指向等待该缓冲区解锁的进程。
365         h->b_next = NULL;        // 指向具有相同 hash 值的下一个缓冲头。
366         h->b_prev = NULL;        // 指向具有相同 hash 值的前一个缓冲头。
367         h->b_data = (char *) b;  // 指向对应缓冲区数据块（1024 字节）。
368         h->b_prev_free = h-1;    // 指向链表中前一项。
369         h->b_next_free = h+1;    // 指向链表中下一项。
370         h++;                    // h 指向下一新缓冲头位置。
371         NR_BUFFERS++;           // 缓冲区块数累加。
372         if (b == (void *) 0x100000) // 如果地址 b 递减到等于 1MB，则跳过 384KB，
373             b = (void *) 0xA0000; // 让 b 指向地址 0xA0000 (640KB) 处。
374     }
375     h--;                        // 让 h 指向最后一个有效缓冲头。
376     free_list = start_buffer;   // 让空闲链表头指向头一个缓冲区头。
377     free_list->b_prev_free = h;  // 链表头的 b_prev_free 指向前一项（即最后一项）。
378     h->b_next_free = free_list; // h 的下一项指针指向第一项，形成一个环链。
// 初始化 hash 表（哈希表、散列表），置表中所有的指针为 NULL。
379     for (i=0; i<NR_HASH; i++)
380         hash_table[i]=NULL;
381 }
382

```

### 9.5.3 其它信息

## 9.6 block\_dev.c 文件

### 9.6.1 功能描述

该文件包括 `block_read()` 和 `block_write()` 两个函数。这两个函数是供系统调用函数 `read()` 和 `write()` 调用的，其它地方没有引用。

块设备读写操作所使用的函数之间的层次关系为：

- `read()`，`write()`

- block\_read(), block\_write(), file\_read(), file\_write(), read\_pipe(), write()
- bread() 或 breada()
- getblk()
- ll\_rw\_block()

## 9.6.2 代码注释

列表 linux/fs/block\_dev.c 程序

```

1  /*
2   *  linux/fs/block_dev.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  #include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
8
9  #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
10                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <asm/segment.h>     // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
13 #include <asm/system.h>      // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14
15  /// 数据块写函数 - 向指定设备从给定偏移处写入指定字节数据。
16  // 参数: dev - 设备号; pos - 设备文件中偏移量指针; buf - 用户地址空间中缓冲区地址;
17  //      count - 要传送的字节数。
18  // 对于内核来说，写操作是向高速缓冲区中写入数据，什么时候数据最终写入设备是由高速缓冲管理
19  // 程序决定并处理的。另外，因为设备是以块为单位进行读写的，因此对于写开始位置不处于块起始
20  // 处时，需要先将开始字节所在的整个块读出，然后将需要写的数据从写开始处填写满该块，再将完
21  // 整的一块数据写盘（即交由高速缓冲程序去处理）。
22
23  int block_write(int dev, long * pos, char * buf, int count)
24  {
25      // 由 pos 地址换算成开始读写块的块序号 block。并求出需读第 1 字节在该块中的偏移位置 offset。
26      int block = *pos >> BLOCK_SIZE_BITS;
27      int offset = *pos & (BLOCK_SIZE-1);
28      int chars;
29      int written = 0;
30      struct buffer_head * bh;
31      register char * p;
32
33      // 针对要写入的字节数 count，循环执行以下操作，直到全部写入。
34      while (count>0) {
35          // 计算在该块中可写入的字节数。如果需要写入的字节数填不满一块，则只需写 count 字节。
36          chars = BLOCK_SIZE - offset;
37          if (chars > count)
38              chars=count;
39          // 如果正好要写 1 块数据，则直接申请 1 块高速缓冲块，否则需要读入将被修改的数据块，并预读
40          // 下两块数据，然后将块号递增 1。
41          if (chars == BLOCK_SIZE)
42              bh = getblk(dev, block);
43          else
44              bh = breada(dev, block, block+1, block+2, -1);
45          block++;

```

```

// 如果写出错，则返回已写字节数，如果没有写入任何字节，则返回出错号（负数）。
32         if (!bh)
33             return written?written:-EIO;
// p 指向读出数据块中开始写的位置。若最后写入的数据不足一块，则需从块开始填写（修改）所需
// 的字节，因此这里需置 offset 为零。
34         p = offset + bh->b_data;
35         offset = 0;
// 将文件中偏移指针前移已写字节数。累加已写字节数 chars。传送计数值减去此次已传送字节数。
36         *pos += chars;
37         written += chars;
38         count -= chars;
// 从用户缓冲区复制 chars 字节到 p 指向的高速缓冲区中开始写入的位置。
39         while (chars-->0)
40             *(p++) = get_fs_byte(buf++);
// 置该缓冲区块已修改标志，并释放该缓冲区（也即该缓冲区引用计数递减 1）。
41         bh->b_dirt = 1;
42         brelse(bh);
43     }
44     return written; // 返回已写入的字节数，正常退出。
45 }
46
//// 数据块读函数 - 从指定设备和位置读入指定字节数的数据到高速缓冲中。
47 int block_read(int dev, unsigned long * pos, char * buf, int count)
48 {
// 由 pos 地址换算成开始读写块的块序号 block。并求出需读第 1 字节在该块中的偏移位置 offset。
49     int block = *pos >> BLOCK_SIZE_BITS;
50     int offset = *pos & (BLOCK_SIZE-1);
51     int chars;
52     int read = 0;
53     struct buffer_head * bh;
54     register char * p;
55
// 针对要读入的字节数 count，循环执行以下操作，直到全部读入。
56     while (count>0) {
// 计算在该块中需读入的字节数。如果需要读入的字节数不满一块，则只需读 count 字节。
57         chars = BLOCK_SIZE-offset;
58         if (chars > count)
59             chars = count;
// 读入需要的数据块，并预读下两块数据，如果读操作出错，则返回已读字节数，如果没有读入任何
// 字节，则返回出错号。然后将块号递增 1。
60         if (!(bh = breada(dev, block, block+1, block+2, -1)))
61             return read?read:-EIO;
62         block++;
// p 指向从设备读出数据块中需要读取的开始位置。若最后需要读取的数据不足一块，则需从块开始
// 读取所需的字节，因此这里需将 offset 置零。
63         p = offset + bh->b_data;
64         offset = 0;
// 将文件中偏移指针前移已读出字节数 chars。累加已读字节数。传送计数值减去此次已传送字节数。
65         *pos += chars;
66         read += chars;
67         count -= chars;
// 从高速缓冲区中 p 指向的开始位置复制 chars 字节数据到用户缓冲区，并释放该高速缓冲区。
68         while (chars-->0)

```



```

69         put_fs_byte(*(p++), buf++);
70         brelse(bh);
71     }
72     return read;          // 返回已读取的字节数，正常退出。
73 }
74

```

### 9.6.3 其它信息

## 9.7 file\_dev.c 文件

### 9.7.1 功能描述

该文件包括file\_read()和file\_write()两个函数。这两个函数是供系统调用函数read()和write()调用的，其它地方没有引用。

### 9.7.2 代码注释

列表 linux/fs/file\_dev.c 程序

```

1  /*
2   * linux/fs/file_dev.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  #include <errno.h>        // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)
8  #include <fcntl.h>        // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
9
10 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
13
14 #define MIN(a,b) (((a)<(b))?(a):(b))    // 取 a,b 中的最小值。
15 #define MAX(a,b) (((a)>(b))?(a):(b))    // 取 a,b 中的最大值。
16
    // 文件读函数 - 根据 i 节点和文件结构，读设备数据。
    // 由 i 节点可以知道设备号，由 filp 结构可以知道文件中当前读写指针位置。buf 指定用户态中
    // 缓冲区的位置，count 为需要读取的字节数。返回值是实际读取的字节数，或出错号(小于 0)。
17 int file_read(struct m_inode * inode, struct file * filp, char * buf, int count)
18 {
19     int left, chars, nr;
20     struct buffer_head * bh;
21
    // 若需要读取的字节计数值小于等于零，则返回。
22     if ((left=count)<=0)
23         return 0;

```

```

// 若还需要读取的字节数不等于 0，就循环执行以下操作，直到全部读出。
24 while (left) {
// 根据 i 节点和文件表结构信息，取数据块文件当前读写位置在设备上对应的逻辑块号 nr。若 nr 不
// 为 0，则从 i 节点指定的设备上读取该逻辑块，如果读操作失败则退出循环。若 nr 为 0，表示指定
// 的数据块不存在，置缓冲块指针为 NULL。
25     if (nr = bmap(inode, (filp->f_pos)/BLOCK\_SIZE)) {
26         if (! (bh=bread(inode->i_dev, nr)))
27             break;
28     } else
29         bh = NULL;
// 计算文件读写指针在数据块中的偏移值 nr，则该块中可读字节数为 (BLOCK_SIZE-nr)，然后与还需
// 读取的字节数 left 作比较，其中小值即为本次需读的字节数 chars。若 (BLOCK_SIZE-nr) 大则说明
// 该块是需要读取的最后一块数据，反之则还需要读取一块数据。
30     nr = filp->f_pos % BLOCK\_SIZE;
31     chars = MIN( BLOCK\_SIZE-nr , left );
// 调整读写文件指针。指针前移此次将读取的字节数 chars。剩余字节计数相应减去 chars。
32     filp->f_pos += chars;
33     left -= chars;
// 若从设备上读到了数据，则将 p 指向读出数据块缓冲区中开始读取的位置，并且复制 chars 字节
// 到用户缓冲区 buf 中。否则往用户缓冲区中填入 chars 个 0 值字节。
34     if (bh) {
35         char * p = nr + bh->b_data;
36         while (chars-->0)
37             put\_fs\_byte(* (p++), buf++);
38         brelse(bh);
39     } else {
40         while (chars-->0)
41             put\_fs\_byte(0, buf++);
42     }
43 }
// 修改该 i 节点的访问时间为当前时间。返回读取的字节数，若读取字节数为 0，则返回出错号。
44     inode->i_atime = CURRENT\_TIME;
45     return (count-left)?(count-left):-ERROR;
46 }
47
///// 文件写函数 - 根据 i 节点和文件结构信息，将用户数据写入指定设备。
// 由 i 节点可以知道设备号，由 filp 结构可以知道文件中当前读写指针位置。buf 指定用户态中
// 缓冲区的位置，count 为需要写入的字节数。返回值是实际写入的字节数，或出错号(小于 0)。
48 int file\_write(struct m\_inode * inode, struct file * filp, char * buf, int count)
49 {
50     off\_t pos;
51     int block, c;
52     struct buffer\_head * bh;
53     char * p;
54     int i=0;
55
56     /*
57      * ok, append may not work when many processes are writing at the same time
58      * but so what. That way leads to madness anyway.
59      */
60     /*
61      * ok, 当许多进程同时写时，append 操作可能不行，但那又怎样。不管怎样那样做会
62      * 导致混乱一团。

```

```

*/
// 如果是要向文件后添加数据，则将文件读写指针移到文件尾部。否则就将在文件读写指针处写入。
60     if (filp->f_flags & O_APPEND)
61         pos = inode->i_size;
62     else
63         pos = filp->f_pos;
// 若已写入字节数 i 小于需要写入的字节数 count，则循环执行以下操作。
64     while (i < count) {
// 创建数据块号(pos/BLOCK_SIZE)在设备上对应的逻辑块，并返回在设备上的逻辑块号。如果逻辑
// 块号=0，则表示创建失败，退出循环。
65         if (!(block = create_block(inode, pos/BLOCK_SIZE)))
66             break;
// 根据该逻辑块号读取设备上的相应数据块，若出错则退出循环。
67         if (!(bh=bread(inode->i_dev, block)))
68             break;
// 求出文件读写指针在数据块中的偏移值 c，将 p 指向读出数据块缓冲区中开始读取的位置。置该
// 缓冲区已修改标志。
69         c = pos % BLOCK_SIZE;
70         p = c + bh->b_data;
71         bh->b_dirt = 1;
// 从开始读写位置到块末共可写入 c=(BLOCK_SIZE-c)个字节。若 c 大于剩余还需写入的字节数
// (count-i)，则此次只需再写入 c=(count-i)即可。
72         c = BLOCK_SIZE-c;
73         if (c > count-i) c = count-i;
// 文件读写指针前移此次需写入的字节数。如果当前文件读写指针位置值超过了文件的大小，则
// 修改 i 节点中文件大小字段，并置 i 节点已修改标志。
74         pos += c;
75         if (pos > inode->i_size) {
76             inode->i_size = pos;
77             inode->i_dirt = 1;
78         }
// 已写入字节计数累加此次写入的字节数 c。从用户缓冲区 buf 中复制 c 个字节到高速缓冲区中 p
// 指向开始的位置处。然后释放该缓冲区。
79         i += c;
80         while (c-->0)
81             *(p++) = get_fs_byte(buf++);
82         brelse(bh);
83     }
// 更改文件修改时间为当前时间。
84     inode->i_mtime = CURRENT_TIME;
// 如果此次文件操作是在文件尾添加数据，则仍将文件读写指针指向文件尾，并更改 i 节点修改
// 时间为当前时间。
85     if (!(filp->f_flags & O_APPEND)) {
86         filp->f_pos = pos;
87         inode->i_ctime = CURRENT_TIME;
88     }
// 返回写入的字节数，若写入字节数为 0，则返回出错号-1。
89     return (i?i:-1);
90 }
91

```

### 9.7.3 其它信息

## 9.8 file\_table.c 文件

### 9.8.1 功能描述

该程序目前是空的，仅定义了文件表数组。

### 9.8.2 代码注释

列表 linux/fs/file\_table.c 程序

---

```

1 /*
2  * linux/fs/file_table.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #include <linux/fs.h>    // 文件系统头文件。定义文件表结构（file, buffer_head, m_inode 等）。
8
9 struct file file_table[NR_FILE]; // 文件表数组 (64 项)。
10

```

---

### 9.8.3 其它信息

## 9.9 fcntl.c 文件

### 9.9.1 功能描述

### 9.9.2 代码注释

列表 linux/fs/fcntl.c 程序

---

```

1 /*
2  * linux/fs/fcntl.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #include <string.h>    // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8 #include <errno.h>    // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
9 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
// 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。

```

---

```

10 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
11 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
12
13 #include <fcntl.h> // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
14 #include <sys/stat.h> // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
15
16 extern int sys_close(int fd); // 关闭文件系统调用。(fs/open.c, 192)
17
18 // 复制文件句柄(描述符)。
19 // 参数 fd 是欲复制的文件句柄, arg 指定新文件句柄的最小数值。
20 // 返回新文件句柄或出错码。
21 static int dupfd(unsigned int fd, unsigned int arg)
22 {
23     // 如果文件句柄值大于一个程序最多打开文件数 NR_OPEN, 或者该句柄的文件结构不存在, 则出错,
24     // 返回出错码并退出。
25     if (fd >= NR_OPEN || !current->filp[fd])
26         return -EBADF;
27     // 如果指定的新句柄值 arg 大于最多打开文件数, 则出错, 返回出错码并退出。
28     if (arg >= NR_OPEN)
29         return -EINVAL;
30     // 在当前进程的文件结构指针数组中寻找索引号大于等于 arg 但还没有使用的项。
31     while (arg < NR_OPEN)
32         if (current->filp[arg])
33             arg++;
34     else
35         break;
36     // 如果找到的新句柄值 arg 大于最多打开文件数, 则出错, 返回出错码并退出。
37     if (arg >= NR_OPEN)
38         return -EMFILE;
39     // 在执行时关闭标志位图中复位该句柄位。也即在运行 exec() 类函数时不关闭该句柄。
40     current->close_on_exec &= ~(1<<arg);
41     // 令该文件结构指针等于原句柄 fd 的指针, 并将文件引用计数增 1。
42     (current->filp[arg] = current->filp[fd])->f_count++;
43     return arg; // 返回新的文件句柄。
44 }
45
46 // 复制文件句柄系统调用函数。
47 // 复制指定文件句柄 oldfd 为句柄值等于 newfd 的句柄。如果 newfd 已经打开, 则首先关闭之。
48 int sys_dup2(unsigned int oldfd, unsigned int newfd)
49 {
50     sys_close(newfd); // 若句柄 newfd 已经打开, 则首先关闭之。
51     return dupfd(oldfd, newfd); // 复制并返回新句柄。
52 }
53
54 // 复制文件句柄系统调用函数。
55 // 复制指定文件句柄 oldfd 为句柄值最小的未用句柄。
56 int sys_dup(unsigned int fildes)
57 {
58     return dupfd(fildes, 0);
59 }
60
61 // 文件句柄(描述符)操作系统调用函数。
62 // 参数 fd 是文件句柄, cmd 是操作命令(参见 include/fcntl.h, 23-30 行)。

```

```

47 int sys_fcntl(unsigned int fd, unsigned int cmd, unsigned long arg)
48 {
49     struct file * filp;
50     // 如果文件句柄值大于一个程序最多打开文件数 NR_OPEN, 或者该句柄的文件结构指针为空, 则出错,
    // 返回出错码并退出。
51     if (fd >= NR_OPEN || !(filp = current->filp[fd]))
52         return -EBADF;
    // 根据不同命令 cmd 进行分别处理。
53     switch (cmd) {
54         case F_DUPFD: // 复制文件句柄。
55             return dupfd(fd, arg);
56         case F_GETFD: // 取文件句柄的执行时关闭标志。
57             return (current->close_on_exec >> fd) & 1;
58         case F_SETFD: // 设置句柄执行时关闭标志。arg 位 0 置位是设置, 否则关闭。
59             if (arg & 1)
60                 current->close_on_exec |= (1 << fd);
61             else
62                 current->close_on_exec &= ~(1 << fd);
63             return 0;
64         case F_GETFL: // 取文件状态标志和访问模式。
65             return filp->f_flags;
66         case F_SETFL: // 设置文件状态和访问模式(根据 arg 设置添加、非阻塞标志)。
67             filp->f_flags &= ~(O_APPEND | O_NONBLOCK);
68             filp->f_flags |= arg & (O_APPEND | O_NONBLOCK);
69             return 0;
70         case F_GETLK: case F_SETLK: case F_SETLKW: // 未实现。
71             return -1;
72         default:
73             return -1;
74     }
75 }
76

```

### 9.9.3 其它信息

## 9.10 stat.c 文件

### 9.10.1 功能描述

该程序实现取文件状态信息系统调用函数 stat() 和 fstat(), 并将信息存放在用户的文件状态结构缓冲区中。stat() 是利用文件名取信息, 而 fstat() 是使用文件句柄(描述符)来取信息。

### 9.10.2 代码注释

列表 linux/fs/stat.c 程序

```

1 /*
2  * linux/fs/stat.c

```

```

3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
8 #include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
9
10 #include <linux/fs.h>      // 文件系统头文件。定义文件表结构 (file, buffer_head, m_inode 等)。
11 #include <linux/sched.h>   // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                            // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14
    ///// 复制文件状态信息。
    // 参数 inode 是文件对应的 i 节点，statbuf 是 stat 文件状态结构指针，用于存放取得的状态信息。
15 static void cp_stat(struct m_inode * inode, struct stat * statbuf)
16 {
17     struct stat tmp;
18     int i;
19
    // 首先验证(或分配)存放数据的内存空间。
20     verify_area(statbuf, sizeof (* statbuf));
    // 然后临时复制相应节点上的信息。
21     tmp.st_dev = inode->i_dev;          // 文件所在的设备号。
22     tmp.st_ino = inode->i_num;          // 文件 i 节点号。
23     tmp.st_mode = inode->i_mode;        // 文件属性。
24     tmp.st_nlink = inode->i_nlinks;     // 文件的连接数。
25     tmp.st_uid = inode->i_uid;          // 文件的用户 id。
26     tmp.st_gid = inode->i_gid;          // 文件的组 id。
27     tmp.st_rdev = inode->i_zone[0];     // 设备号(如果文件是特殊的字符文件或块文件)。
28     tmp.st_size = inode->i_size;        // 文件大小(字节数)(如果文件是常规文件)。
29     tmp.st_atime = inode->i_atime;      // 最后访问时间。
30     tmp.st_mtime = inode->i_mtime;      // 最后修改时间。
31     tmp.st_ctime = inode->i_ctime;      // 最后节点修改时间。
    // 最后将这些状态信息复制到用户缓冲区中。
32     for (i=0 ; i<sizeof (tmp) ; i++)
33         put_fs_byte(((char *) &tmp)[i], &((char *) statbuf)[i]);
34 }
35
    ///// 文件状态系统调用函数 - 根据文件名获取文件状态信息。
    // 参数 filename 是指定的文件名，statbuf 是存放状态信息的缓冲区指针。
    // 返回 0，若出错则返回出错码。
36 int sys_stat(char * filename, struct stat * statbuf)
37 {
38     struct m_inode * inode;
39
    // 首先根据文件名找出对应的 i 节点，若出错则返回错误码。
40     if (!(inode=namei(filename)))
41         return -ENOENT;
    // 将 i 节点上的文件状态信息复制到用户缓冲区中，并释放该 i 节点。
42     cp_stat(inode, statbuf);
43     iput(inode);
44     return 0;

```



```

45 }
46
47 // 文件状态系统调用 - 根据文件句柄获取文件状态信息。
48 // 参数 fd 是指定文件的句柄(描述符)，statbuf 是存放状态信息的缓冲区指针。
49 // 返回 0，若出错则返回出错码。
50 int sys_fstat(unsigned int fd, struct stat * statbuf)
51 {
52     struct file * f;
53     struct m_inode * inode;
54
55     // 如果文件句柄值大于一个程序最多打开文件数 NR_OPEN，或者该句柄的文件结构指针为空，或者
56     // 对应文件结构的 i 节点字段为空，则出错，返回出错码并退出。
57     if (fd >= NR_OPEN || !(f=current->filp[fd]) || !(inode=f->f_inode))
58         return -EBADF;
59     // 将 i 节点上的文件状态信息复制到用户缓冲区中。
60     cp_stat(inode, statbuf);
61     return 0;
62 }
63

```

### 9.10.3 其它信息

## 9.11 pipe.c 文件

### 9.11.1 功能描述

本程序执行管道文件的读写操作，并实现了管道系统调用 pipe()。

在初始化管道时，管道 i 节点的 i\_size 字段中被设置为指向管道缓冲区的指针，管道数据头部指针存放在 i\_zone[0] 字段中，而管道数据尾部指针存放在 i\_zone[1] 字段中。对于读管道操作，数据是从管道尾读出，并使管道尾指针前移读取字节数个位置；对于往管道中的写入操作，数据是向管道头部写入，并使管道头指针前移写入字节数个位置。参见下面的管道示意图。

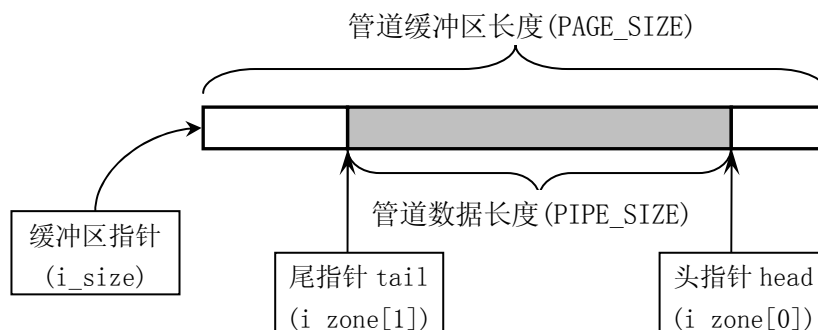


图 5.1 管道缓冲区操作示意图

## 9.11.2 代码注释

列表 linux/fs/pipe.c 程序

```

1  /*
2  *  linux/fs/pipe.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <signal.h>          // 信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
8
9  #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
10 #include <linux/mm.h>      /* for get_free_page */ /* 使用其中的 get_free_page */
    // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
11 #include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
12
    // 管道读操作函数。
    // 参数 inode 是管道对应的 i 节点，buf 是数据缓冲区指针，count 是读取的字节数。
13 int read_pipe(struct m_inode * inode, char * buf, int count)
14 {
15     int chars, size, read = 0;
16
    // 若欲读取的字节计数值 count 大于 0，则循环执行以下操作。
17     while (count > 0) {
    // 若当前管道中没有数据(size=0)，则唤醒等待该节点的进程，如果已没有写管道者，则返回已读
    // 字节数，退出。否则在该 i 节点上睡眠，等待信息。
18         while (!(size=PIPE_SIZE(*inode))) {
19             wake_up(&inode->i_wait);
20             if (inode->i_count != 2) /* are there any writers? */
21                 return read;
22             sleep_on(&inode->i_wait);
23         }
    // 取管道尾到缓冲区末端的字节数 chars。如果其大于还需要读取的字节数 count，则令其等于 count。
    // 如果 chars 大于当前管道中含有数据的长度 size，则令其等于 size。
24         chars = PAGE_SIZE-PIPE_TAIL(*inode);
25         if (chars > count)
26             chars = count;
27         if (chars > size)
28             chars = size;
    // 读字节计数减去此次可读的字节数 chars，并累加已读字节数。
29         count -= chars;
30         read += chars;
    // 令 size 指向管道尾部，调整当前管道尾指针（前移 chars 字节）。
31         size = PIPE_TAIL(*inode);
32         PIPE_TAIL(*inode) += chars;
33         PIPE_TAIL(*inode) &= (PAGE_SIZE-1);
    // 将管道中的数据复制到用户缓冲区中。对于管道 i 节点，其 i_size 字段中是管道缓冲块指针。
34         while (chars-->0)
35             put_fs_byte(((char *)inode->i_size)[size++], buf++);
36     }
    // 唤醒等待该 i 节点的进程，并返回读取的字节数。
37     wake_up(&inode->i_wait);

```

```

38         return read;
39     }
40     ///// 管道写操作函数。
    // 参数 inode 是管道对应的 i 节点, buf 是数据缓冲区指针, count 是将写入管道的字节数。
41 int write_pipe(struct m_inode * inode, char * buf, int count)
42 {
43     int chars, size, written = 0;
44     // 若将写入的字节计数值 count 还大于 0, 则循环执行以下操作。
45     while (count > 0) {
    // 若当前管道中没有已经满了(size=0), 则唤醒等待该节点的进程, 如果已没有读管道者, 则返回
    // 已写入的字节数并退出。若写入 0 字节, 则返回-1。否则在该 i 节点上睡眠, 等待管道腾出空间。
46         while (!(size=(PAGE_SIZE-1)-PIPE_SIZE(*inode))) {
47             wake_up(&inode->i_wait);
48             if (inode->i_count != 2) { /* no readers */
49                 current->signal |= (1<<(SIGPIPE-1));
50                 return written?written:-1;
51             }
52             sleep_on(&inode->i_wait);
53         }
    // 取管道头部到缓冲区末端空闲字节数 chars。如果其大于还需要写入的字节数 count, 则令其等于
    // count。如果 chars 大于当前管道中空闲空间长度 size, 则令其等于 size。
54         chars = PAGE_SIZE-PIPE_HEAD(*inode);
55         if (chars > count)
56             chars = count;
57         if (chars > size)
58             chars = size;
    // 写入字节计数减去此次可写入的字节数 chars, 并累加已写字节数到 written。
59         count -= chars;
60         written += chars;
    // 令 size 指向管道数据头部, 调整当前管道数据头部指针(前移 chars 字节)。
61         size = PIPE_HEAD(*inode);
62         PIPE_HEAD(*inode) += chars;
63         PIPE_HEAD(*inode) &= (PAGE_SIZE-1);
    // 从用户缓冲区复制 chars 个字节到管道中。对于管道 i 节点, 其 i_size 字段中是管道缓冲块指针。
64         while (chars-->0)
65             ((char *)inode->i_size)[size++]=get_fs_byte(buf++);
66     }
    // 唤醒等待该 i 节点的进程, 返回已写入的字节数, 退出。
67     wake_up(&inode->i_wait);
68     return written;
69 }
70
    ///// 创建管道系统调用函数。
    // 在 fildes 所指的数组中创建一对文件句柄(描述符)。这对文件句柄指向一管道 i 节点。fildes[0]
    // 用于读管道中数据, fildes[1]用于向管道中写入数据。
    // 成功时返回 0, 出错时返回-1。
71 int sys_pipe(unsigned long * fildes)
72 {
73     struct m_inode * inode;
74     struct file * f[2];
75     int fd[2];

```

```

76         int i, j;
77
78         // 从系统文件表中取两个空闲项（引用计数字段为 0 的项），并分别设置引用计数为 1。
79         j=0;
80         for(i=0; j<2 && i<NR_FILE; i++)
81             if (!file_table[i].f_count)
82                 (f[j++] = i + file_table) -> f_count++;
83         // 如果只有一个空闲项，则释放该项（引用计数复位）。
84         if (j==1)
85             f[0] -> f_count = 0;
86         // 如果没有找到两个空闲项，则返回-1。
87         if (j<2)
88             return -1;
89         // 针对上面取得的两个文件结构，分别分配一文件句柄，并使进程的文件结构指针分别指向这两个
90         // 文件结构。
91         j=0;
92         for(i=0; j<2 && i<NR_OPEN; i++)
93             if (!current->filp[i]) {
94                 current->filp[fd[j]=i] = f[j];
95                 j++;
96             }
97         // 如果只有一个空闲文件句柄，则释放该句柄。
98         if (j==1)
99             current->filp[fd[0]] = NULL;
100         // 如果没有找到两个空闲句柄，则释放上面获取的两个文件结构项（复位引用计数值），并返回-1。
101         if (j<2) {
102             f[0] -> f_count = f[1] -> f_count = 0;
103             return -1;
104         }
105         // 申请管道节点。如果不成功，则相应释放两个文件句柄和文件结构项，并返回-1。
106         if (!(inode = get_pipe_inode())) {
107             current->filp[fd[0]] =
108             current->filp[fd[1]] = NULL;
109             f[0] -> f_count = f[1] -> f_count = 0;
110             return -1;
111         }
112         // 初始化两个文件结构，都指向同一个 i 节点，读写指针都置零。第 1 个文件结构的文件模式置为读，
113         // 第 2 个文件结构的文件模式置为写。
114         f[0] -> f_inode = f[1] -> f_inode = inode;
115         f[0] -> f_pos = f[1] -> f_pos = 0;
116         f[0] -> f_mode = 1;                /* read */
117         f[1] -> f_mode = 2;                /* write */
118         // 将文件句柄数组复制到对应的用户数组中，并返回 0，退出。
119         put_fs_long(fd[0], 0 + fildes);
120         put_fs_long(fd[1], 1 + fildes);
121         return 0;
122     }
123 }

```

### 9.11.3 其它信息

## 9.12 read\_write.c 文件

### 9.12.1 功能描述

### 9.12.2 代码注释

列表 linux/fs/read\_write.c 程序

```

1  /*
2   * linux/fs/read_write.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  #include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
8  #include <errno.h>        // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
9  #include <sys/types.h>    // 类型头文件。定义了基本的系统数据类型。
10
11 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
12 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
13                          // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
14 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
15
16 // 字符设备读写函数。
17 extern int rw_char(int rw, int dev, char * buf, int count, off_t * pos);
18 // 读管道操作函数。
19 extern int read_pipe(struct m_inode * inode, char * buf, int count);
20 // 写管道操作函数。
21 extern int write_pipe(struct m_inode * inode, char * buf, int count);
22 // 块设备读操作函数。
23 extern int block_read(int dev, off_t * pos, char * buf, int count);
24 // 块设备写操作函数。
25 extern int block_write(int dev, off_t * pos, char * buf, int count);
26 // 读文件操作函数。
27 extern int file_read(struct m_inode * inode, struct file * filp,
28                      char * buf, int count);
29 // 写文件操作函数。
30 extern int file_write(struct m_inode * inode, struct file * filp,
31                       char * buf, int count);
32
33 // 重定位文件读写指针系统调用函数。
34 // 参数 fd 是文件句柄，offset 是新的文件读写指针偏移值，origin 是偏移的起始位置，是 SEEK_SET
35 // (0, 从文件开始处)、SEEK_CUR(1, 从当前读写位置)、SEEK_END(2, 从文件尾处)三者之一。
36 int sys_lseek(unsigned int fd, off_t offset, int origin)
37 {
38     struct file * file;
39     int tmp;
40
41     // 如果文件句柄值大于程序最多打开文件数 NR_OPEN，或者该句柄的文件结构指针为空，或者
42     // 对应文件结构的 i 节点字段为空，或者指定设备文件指针不可定位，则返回出错码并退出。
43     if (fd >= NR_OPEN || !(file=current->filp[fd]) || !(file->f_inode)

```

```

31         || !IS_SEEKABLE(MAJOR(file->f_inode->i_dev)))
32         return -EBADF;
// 如果文件对应的 i 节点是管道节点, 则返回出错码, 退出。管道头尾指针不可随意移动!
33         if (file->f_inode->i_pipe)
34             return -ESPIPE;
// 根据设置的起始位置, 分别重新定位文件读写指针。
35         switch (origin) {
// origin = SEEK_SET, 要求以文件起始处作为原点设置文件读写指针。若偏移值小于零, 则出错返
// 回错误码。否则设置文件读写指针等于 offset。
36             case 0:
37                 if (offset<0) return -EINVAL;
38                 file->f_pos=offset;
39                 break;
// origin = SEEK_CUR, 要求以文件当前读写指针处作为原点重定位读写指针。如果文件当前指针加
// 上偏移值小于 0, 则返回出错码退出。否则在当前读写指针上加上偏移值。
40             case 1:
41                 if (file->f_pos+offset<0) return -EINVAL;
42                 file->f_pos += offset;
43                 break;
// origin = SEEK_END, 要求以文件末尾作为原点重定位读写指针。此时若文件大小加上偏移值小于零
// 则返回出错码退出。否则重定位读写指针为文件长度加上偏移值。
44             case 2:
45                 if ((tmp=file->f_inode->i_size+offset) < 0)
46                     return -EINVAL;
47                 file->f_pos = tmp;
48                 break;
// origin 设置出错, 返回出错码退出。
49             default:
50                 return -EINVAL;
51         }
52         return file->f_pos;    // 返回重定位后的文件读写指针值。
53     }
54
//// 读文件系统调用函数。
// 参数 fd 是文件句柄, buf 是缓冲区, count 是欲读字节数。
55 int sys_read(unsigned int fd, char * buf, int count)
56 {
57     struct file * file;
58     struct m_inode * inode;
59
// 如果文件句柄值大于程序最多打开文件数 NR_OPEN, 或者需要读取的字节计数值小于 0, 或者该句柄
// 的文件结构指针为空, 则返回出错码并退出。
60     if (fd>=NR_OPEN || count<0 || !(file=current->filp[fd]))
61         return -EINVAL;
// 若需读取的字节数 count 等于 0, 则返回 0, 退出
62     if (!count)
63         return 0;
// 验证存放数据的缓冲区内存限制。
64     verify_area(buf, count);
// 取文件对应的 i 节点。若是管道文件, 并且是读管道文件模式, 则进行读管道操作, 若成功则返回
// 读取的字节数, 否则返回出错码, 退出。
65     inode = file->f_inode;
66     if (inode->i_pipe)

```

```

67         return (file->f_mode&1)?read_pipe(inode, buf, count):-EIO;
// 如果是字符型文件, 则进行读字符设备操作, 返回读取的字符数。
68         if (S_ISCHR(inode->i_mode))
69             return rw_char(READ, inode->i_zone[0], buf, count, &file->f_pos);
// 如果是块设备文件, 则执行块设备读操作, 并返回读取的字节数。
70         if (S_ISBLK(inode->i_mode))
71             return block_read(inode->i_zone[0], &file->f_pos, buf, count);
// 如果是目录文件或者是常规文件, 则首先验证读取数 count 的有效性并进行调整(若读取字节数加上
// 文件当前读写指针值大于文件大小, 则重新设置读取字节数为文件长度-当前读写指针值, 若读取数
// 等于0, 则返回0退出), 然后执行文件读操作, 返回读取的字节数并退出。
72         if (S_ISDIR(inode->i_mode) || S_ISREG(inode->i_mode)) {
73             if (count+file->f_pos > inode->i_size)
74                 count = inode->i_size - file->f_pos;
75             if (count<=0)
76                 return 0;
77             return file_read(inode, file, buf, count);
78         }
// 否则打印节点文件属性, 并返回出错码退出。
79         printk("(Read) inode->i_mode=%06o\n|r", inode->i_mode);
80         return -EINVAL;
81     }
82
83 int sys_write(unsigned int fd, char * buf, int count)
84 {
85     struct file * file;
86     struct m_inode * inode;
87
88     // 如果文件句柄值大于程序最多打开文件数 NR_OPEN, 或者需要写入的字节计数小于0, 或者该句柄
89     // 的文件结构指针为空, 则返回出错码并退出。
90     if (fd>NR_OPEN || count < 0 || !(file=current->filp[fd]))
91         return -EINVAL;
92     // 若需读取的字节数 count 等于0, 则返回0, 退出
93     if (!count)
94         return 0;
95     // 取文件对应的 i 节点。若是管道文件, 并且是写管道文件模式, 则进行写管道操作, 若成功则返回
96     // 写入的字节数, 否则返回出错码, 退出。
97     inode=file->f_inode;
98     if (inode->i_pipe)
99         return (file->f_mode&2)?write_pipe(inode, buf, count):-EIO;
100    // 如果是字符型文件, 则进行写字符设备操作, 返回写入的字符数, 退出。
101    if (S_ISCHR(inode->i_mode))
102        return rw_char(WRITE, inode->i_zone[0], buf, count, &file->f_pos);
103    // 如果是块设备文件, 则进行块设备写操作, 并返回写入的字节数, 退出。
104    if (S_ISBLK(inode->i_mode))
105        return block_write(inode->i_zone[0], &file->f_pos, buf, count);
106    // 若是常规文件, 则执行文件写操作, 并返回写入的字节数, 退出。
107    if (S_ISREG(inode->i_mode))
108        return file_write(inode, file, buf, count);
109    // 否则, 显示对应节点的文件模式, 返回出错码, 退出。
110    printk("(Write) inode->i_mode=%06o\n|r", inode->i_mode);
111    return -EINVAL;
112 }
113
114

```



### 9.12.3 其它信息

## 9.13 truncate.c 文件

### 9.13.1 功能描述

本程序用于释放指定 i 节点在设备上占用的所有逻辑块，包括直接块、一次间接块和二次间接块。

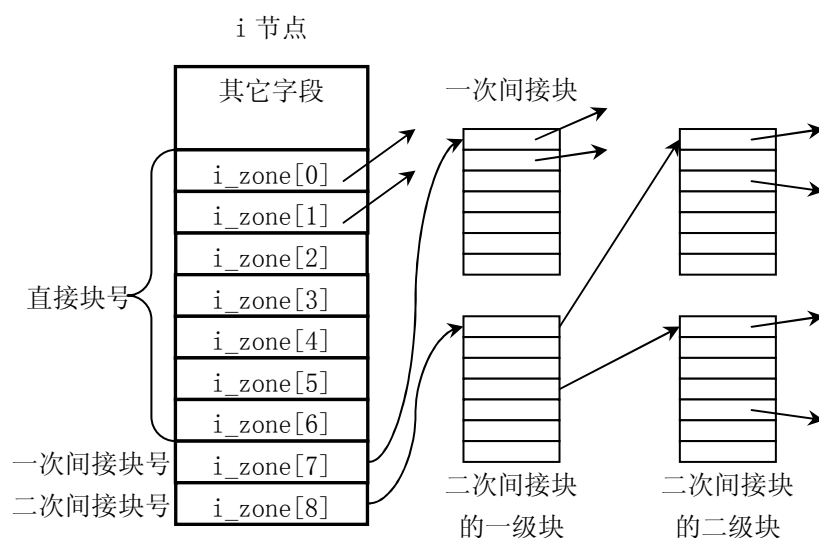


图 索引节点(i 节点)的逻辑块连接方式

### 9.13.2 代码注释

列表 linux/fs/truncate.c 程序

```

1 /*
2  * linux/fs/truncate.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                        // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
8
9 #include <sys/stat.h>    // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
10
11 // 释放一次间接块。
12 static void free_ind(int dev, int block)
13 {
14     struct buffer_head * bh;
15     unsigned short * p;
16     int i;
17
18     // 如果逻辑块号为 0，则返回。

```

```

17         if (!block)
18             return;
// 读取一次间接块，并释放其上表明使用的所有逻辑块，然后释放该一次间接块的缓冲区。
19         if (bh=bread(dev,block)) {
20             p = (unsigned short *) bh->b_data; // 指向数据缓冲区。
21             for (i=0;i<512;i++,p++)           // 每个逻辑块上可有 512 个块号。
22                 if (*p)
23                     free_block(dev,*p); // 释放指定的逻辑块。
24             brelse(bh); // 释放缓冲区。
25         }
// 释放设备上的一次间接块。
26         free_block(dev,block);
27     }
28
//// 释放二次间接块。
29 static void free_dind(int dev,int block)
30 {
31     struct buffer_head * bh;
32     unsigned short * p;
33     int i;
34
// 如果逻辑块号为 0，则返回。
35     if (!block)
36         return;
// 读取二次间接块的一级块，并释放其上表明使用的所有逻辑块，然后释放该一级块的缓冲区。
37     if (bh=bread(dev,block)) {
38         p = (unsigned short *) bh->b_data; // 指向数据缓冲区。
39         for (i=0;i<512;i++,p++)           // 每个逻辑块上可连接 512 个二级块。
40             if (*p)
41                 free_ind(dev,*p); // 释放所有一次间接块。
42             brelse(bh); // 释放缓冲区。
43     }
// 最后释放设备上的二次间接块。
44     free_block(dev,block);
45 }
46
//// 将节点对应的文件长度截为 0，并释放占用的设备空间。
47 void truncate(struct m_inode * inode)
48 {
49     int i;
50
// 如果不是常规文件或者是目录文件，则返回。
51     if (!(S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode)))
52         return;
// 释放 i 节点的 7 个直接逻辑块，并将这 7 个逻辑块项全置零。
53     for (i=0;i<7;i++)
54         if (inode->i_zone[i]) { // 如果块号不为 0，则释放之。
55             free_block(inode->i_dev,inode->i_zone[i]);
56             inode->i_zone[i]=0;
57         }
58     free_ind(inode->i_dev,inode->i_zone[7]); // 释放一次间接块。
59     free_dind(inode->i_dev,inode->i_zone[8]); // 释放二次间接块。
60     inode->i_zone[7] = inode->i_zone[8] = 0; // 逻辑块项 7、8 置零。

```

```

61         inode->i_size = 0;                                // 文件大小置零。
62         inode->i_dirt = 1;                                // 置节点已修改标志。
63         inode->i_mtime = inode->i_ctime = CURRENT_TIME; // 重置文件和节点修改时间为当前时间。
64     }
65
66

```

### 9.13.3 其它信息

## 9.14 super.c 文件

### 9.14.1 功能描述

### 9.14.2 代码注释

列表 linux/fs/super.c 程序

```

1  /*
2  *  linux/fs/super.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  super.c contains code to handle the super-block tables.
9  */
10 #include <linux/config.h> // 内核配置头文件。定义键盘语言和硬盘类型 (HD_TYPE) 可选项。
11 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/system.h> // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
14
15 #include <errno.h> // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
16 #include <sys/stat.h> // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
17
18 int sync_dev(int dev); // 对指定设备执行高速缓冲与设备上数据的同步操作。(fs/buffer.c, 59)
19 void wait for keypress(void); // 等待击键。(kernel/chr_drv/tty_io.c, 140)
20
21 /* set_bit uses setb, as gas doesn't recognize setc */
    /* set_bit() 使用了 setb 指令，因为汇编编译器 gas 不能识别指令 setc */
    // 测试指定位偏移处比特位的值 (0 或 1)，并返回该比特位值。(应该取名为 test_bit() 更妥帖)
    // 嵌入式汇编宏。参数 bitnr 是比特位偏移值，addr 是测试比特位操作的起始地址。
    // %0 - ax(__res), %1 - 0, %2 - bitnr, %3 - addr
22 #define set_bit(bitnr, addr) ({ \
23     register int __res __asm__("ax"); \
24     __asm__("bt %2,%3;setb %%al": "=a" (__res): "a" (0), "r" (bitnr), "m" (*(addr))); \
25     __res; })
26
27 struct super_block super_block[NR_SUPER]; // 超级块结构数组 (共 8 项)。

```

```

28 /* this is initialized in init/main.c */
   /* ROOT_DEV 已在 init/main.c 中被初始化 */
29 int ROOT_DEV = 0;
30
   //// 锁定指定的超级块。
31 static void lock_super(struct super_block * sb)
32 {
33     cli();                // 关中断。
34     while (sb->s_lock)      // 如果该超级块已经上锁，则睡眠等待。
35         sleep_on(&(sb->s_wait));
36     sb->s_lock = 1;         // 给该超级块加锁（置锁定标志）。
37     sti();                // 开中断。
38 }
39
   //// 对指定超级块解锁。（如果使用 ulock_super 这个名称则更妥帖）。
40 static void free_super(struct super_block * sb)
41 {
42     cli();                // 关中断。
43     sb->s_lock = 0;         // 复位锁定标志。
44     wake_up(&(sb->s_wait)); // 唤醒等待该超级块的进程。
45     sti();                // 开中断。
46 }
47
   //// 睡眠等待超级块解锁。
48 static void wait_on_super(struct super_block * sb)
49 {
50     cli();                // 关中断。
51     while (sb->s_lock)      // 如果超级块已经上锁，则睡眠等待。
52         sleep_on(&(sb->s_wait));
53     sti();                // 开中断。
54 }
55
   //// 取指定设备的超级块。返回该超级块结构指针。
56 struct super_block * get_super(int dev)
57 {
58     struct super_block * s;
59
   // 如果没有指定设备，则返回空指针。
60     if (!dev)
61         return NULL;
   // s 指向超级块数组开始处。搜索整个超级块数组，寻找指定设备的超级块。
62     s = 0+super_block;
63     while (s < NR_SUPER+super_block)
   // 如果当前搜索项是指定设备的超级块，则首先等待该超级块解锁（若已经被其它进程上锁的话）。
   // 在等待期间，该超级块有可能被其它设备使用，因此此时需再判断一次是否是指定设备的超级块，
   // 如果是则返回该超级块的指针。否则就重新对超级块数组再搜索一遍，因此 s 重又指向超级块数组
   // 开始处。
64         if (s->s_dev == dev) {
65             wait_on_super(s);
66             if (s->s_dev == dev)
67                 return s;
68             s = 0+super_block;
   // 如果当前搜索项不是，则检查下一项。如果没有找到指定的超级块，则返回空指针。

```

```

69         } else
70             s++;
71     return NULL;
72 }
73
74     ///// 释放指定设备的超级块。
75     // 释放设备所使用的超级块数组项（置 s_dev=0），并释放该设备 i 节点位图和逻辑块位图所占用
76     // 的高速缓冲块。如果超级块对应的文件系统是根文件系统，或者其 i 节点上已经安装有其它的文件
77     // 系统，则不能释放该超级块。
78 void put_super(int dev)
79 {
80     struct super_block * sb;
81     struct m_inode * inode;
82     int i;
83
84     // 如果指定设备是根文件系统设备，则显示警告信息“根系统盘改变了，准备生死决战吧”，并返回。
85     if (dev == ROOT_DEV) {
86         printk("root diskette changed: prepare for armageddon\n\r");
87         return;
88     }
89     // 如果找不到指定设备的超级块，则返回。
90     if (!(sb = get_super(dev)))
91         return;
92     // 如果该超级块指明本文件系统 i 节点上安装有其它的文件系统，则显示警告信息，返回。
93     if (sb->s_imount) {
94         printk("Mounted disk changed - tssk, tssk\n\r");
95         return;
96     }
97     // 找到指定设备的超级块后，首先锁定该超级块，然后置该超级块对应的设备号字段为 0，也即将
98     // 放弃该超级块。
99     lock_super(sb);
100    sb->s_dev = 0;
101    // 然后释放该设备 i 节点位图和逻辑块位图所占用的缓冲块。
102    for(i=0;i<I_MAP_SLOTS;i++)
103        brelse(sb->s_imap[i]);
104    for(i=0;i<Z_MAP_SLOTS;i++)
105        brelse(sb->s_zmap[i]);
106    // 最后对该超级块解锁，并返回。
107    free_super(sb);
108    return;
109 }
110
111     ///// 从设备上读取超级块到内存中。
112     // 如果该设备的超级块已经在高速缓冲中并且有效，则直接返回该超级块的指针。
113 static struct super_block * read_super(int dev)
114 {
115     struct super_block * s;
116     struct buffer_head * bh;
117     int i,block;
118
119     // 如果没有指明设备，则返回空指针。
120     if (!dev)
121         return NULL;

```

```

// 首先检查该设备是否可更换过盘片（也即是否是软盘设备），如果更换过盘，则高速缓冲区有关该
// 设备的所有缓冲块均失效，需要进行失效处理。
108     check\_disk\_change(dev);
// 如果该设备的超级块已经在高速缓冲中，则直接返回该超级块的指针。
109     if (s = get\_super(dev))
110         return s;
// 否则，首先在超级块数组中找出一个空项(也即其 s_dev=0 的项)。如果数组已经占满则返回空指针。
111     for (s = 0+super\_block ;; s++) {
112         if (s >= NR\_SUPER+super\_block)
113             return NULL;
114         if (!s->s_dev)
115             break;
116     }
// 找到超级块空项后，就将该超级块用于指定设备，对该超级块进行部分初始化。
117     s->s_dev = dev;
118     s->s_isup = NULL;
119     s->s_imount = NULL;
120     s->s_time = 0;
121     s->s_rd_only = 0;
122     s->s_dirt = 0;
// 然后所定该超级块，并从设备上读取超级块信息到 bh 指向的缓冲区中。如果读超级块操作失败，
// 则释放上面选定的超级块数组中的项，并解锁该项，返回空指针退出。
123     lock\_super(s);
124     if (!(bh = bread(dev, 1))) {
125         s->s_dev=0;
126         free\_super(s);
127         return NULL;
128     }
// 将设备上读取的超级块信息复制到内存超级块结构中。并释放存放读取信息的高速缓冲块。
129     *((struct d\_super\_block *) s) =
130         *((struct d\_super\_block *) bh->b_data);
131     brelse(bh);
// 如果读取的超级块的文件系统魔数字段内容不对，说明设备上不是正确的文件系统，因此同上面
// 一样，释放上面选定的超级块数组中的项，并解锁该项，返回空指针退出。
// 对于该版 linux 内核，只支持 minix 文件系统版本 1，其魔数是 0x137f。
132     if (s->s_magic != SUPER\_MAGIC) {
133         s->s_dev = 0;
134         free\_super(s);
135         return NULL;
136     }
// 下面开始读取设备上 i 节点位图和逻辑块位图数据。首先初始化内存超级块结构中位图空间。
137     for (i=0;i<I\_MAP\_SLOTS;i++)
138         s->s_imap[i] = NULL;
139     for (i=0;i<Z\_MAP\_SLOTS;i++)
140         s->s_zmap[i] = NULL;
// 然后从设备上读取 i 节点位图和逻辑块位图信息，并存放在超级块对应字段中。
141     block=2;
142     for (i=0 ; i < s->s_imap_blocks ; i++)
143         if (s->s_imap[i]=bread(dev, block))
144             block++;
145     else
146         break;
147     for (i=0 ; i < s->s_zmap_blocks ; i++)

```

```

148         if (s->s_zmap[i]=bread(dev, block))
149             block++;
150         else
151             break;
// 如果读出的位图逻辑块数不等于位图应该占有的逻辑块数, 说明文件系统位图信息有问题, 超级块
// 初始化失败。因此只能释放前面申请的所有资源, 返回空指针并退出。
152         if (block != 2+s->s_imap_blocks+s->s_zmap_blocks) {
// 释放 i 节点位图和逻辑块位图占用的高速缓冲区。
153             for(i=0;i<I_MAP_SLOTS;i++)
154                 brelse(s->s_imap[i]);
155             for(i=0;i<Z_MAP_SLOTS;i++)
156                 brelse(s->s_zmap[i]);
// 释放上面选定的超级块数组中的项, 并解锁该超级块项, 返回空指针退出。
157             s->s_dev=0;
158             free_super(s);
159             return NULL;
160         }
// 否则一切成功。对于申请空闲 i 节点的函数来讲, 如果设备上所有的 i 节点已经全被使用, 则查找
// 函数会返回 0 值。因此 0 号 i 节点是不能用的, 所以这里将位图中的最低位设置为 1, 以防止文件
// 系统分配 0 号 i 节点。同样的道理, 也将逻辑块位图的最低位设置为 1。
161         s->s_imap[0]->b_data[0] |= 1;
162         s->s_zmap[0]->b_data[0] |= 1;
// 解锁该超级块, 并返回超级块指针。
163         free_super(s);
164         return s;
165     }
166
///// 卸载文件系统调用函数。
// 参数 dev_name 是设备文件名。
167 int sys_umount(char * dev_name)
168 {
169     struct m_inode * inode;
170     struct super_block * sb;
171     int dev;
172
// 首先根据设备文件名找到对应的 i 节点, 并取其中的设备号。
173     if (!(inode=namei(dev_name)))
174         return -ENOENT;
175     dev = inode->i_zone[0];
// 如果不是块设备文件, 则释放刚申请的 i 节点 dev_i, 返回出错码。
176     if (!S_ISBLK(inode->i_mode)) {
177         iput(inode);
178         return -ENOTBLK;
179     }
// 释放刚申请的 i 节点 dev_i。
180     iput(inode);
// 如果设备是根文件系统, 则不能被卸载, 返回出错号。
181     if (dev==ROOT_DEV)
182         return -EBUSY;
// 如果取设备的超级块失败, 或者该设备文件系统没有安装过, 则返回出错码。
183     if (!(sb=get_super(dev)) || !(sb->s_imount))
184         return -ENOENT;
// 如果超级块所指明的被安装到的 i 节点没有置位其安装标志, 则显示警告信息。

```



```

185         if (!sb->s_imount->i_mount)
186             printk("Mounted inode has i_mount=0\n");
// 查找 i 节点表, 看是否有进程在使用该设备上的文件, 如果有则返回忙出错码。
187         for (inode=inode_table+0 ; inode<inode_table+NR_INODE ; inode++)
188             if (inode->i_dev==dev && inode->i_count)
189                 return -EBUSY;
// 复位被安装到的 i 节点的安装标志, 释放该 i 节点。
190         sb->s_imount->i_mount=0;
191         iput(sb->s_imount);
// 置超级块中被安装 i 节点字段为空, 并释放设备文件系统的根 i 节点, 置超级块中被安装系统
// 根 i 节点指针为空。
192         sb->s_imount = NULL;
193         iput(sb->s_isup);
194         sb->s_isup = NULL;
// 释放该设备的超级块, 并对该设备执行高速缓冲与设备上数据的同步操作。
195         put_super(dev);
196         sync_dev(dev);
197         return 0;
198     }
199 }
//// 安装文件系统调用函数。
// 参数 dev_name 是设备文件名, dir_name 是安装到的目录名, rw_flag 被安装文件的读写标志。
200 int sys_mount(char * dev_name, char * dir_name, int rw_flag)
201 {
202     struct m_inode * dev_i, * dir_i;
203     struct super_block * sb;
204     int dev;
205
// 首先根据设备文件名找到对应的 i 节点, 并取其中的设备号。
206     if (!(dev_i=namei(dev_name)))
207         return -ENOENT;
208     dev = dev_i->i_zone[0];
// 如果不是块设备文件, 则释放刚申请的 i 节点 dev_i, 返回出错码。
209     if (!S_ISBLK(dev_i->i_mode)) {
210         iput(dev_i);
211         return -EPERM;
212     }
// 释放刚申请的 i 节点 dev_i。
213     iput(dev_i);
// 根据给定的目录文件名找到对应的 i 节点 dir_i。
214     if (!(dir_i=namei(dir_name)))
215         return -ENOENT;
// 如果该 i 节点的引用计数不为 1 (仅在这里引用), 或者该 i 节点的节点号是根文件系统的节点
// 号 1, 则释放该 i 节点, 返回出错码。
216     if (dir_i->i_count != 1 || dir_i->i_num == ROOT_INO) {
217         iput(dir_i);
218         return -EBUSY;
219     }
// 如果该节点不是一个目录文件节点, 则也释放该 i 节点, 返回出错码。
220     if (!S_ISDIR(dir_i->i_mode)) {
221         iput(dir_i);
222         return -EPERM;
223     }

```

```

// 读取将安装文件系统的超级块，如果失败则也释放该 i 节点，返回出错码。
224     if (!(sb=read\_super(dev))) {
225         iput(dir_i);
226         return -EBUSY;
227     }
// 如果将要被安装的文件系统已经安装在其它地方，则释放该 i 节点，返回出错码。
228     if (sb->s_imount) {
229         iput(dir_i);
230         return -EBUSY;
231     }
// 如果将要安装到的 i 节点已经安装了文件系统(安装标志已经置位)，则释放该 i 节点，返回出错码。
232     if (dir_i->i_mount) {
233         iput(dir_i);
234         return -EPERM;
235     }
// 被安装文件系统超级块的被安装到 i 节点字段指向安装到的目录的 i 节点。
236     sb->s_imount=dir_i;
// 置安装 i 节点的安装标志和节点已修改标志。/* 注意！这里没有 iput(dir_i) */
237     dir_i->i_mount=1;                /* 这将在 umount 内操作 */
238     dir_i->i_dirt=1;                /* NOTE! we don't iput(dir_i) */
239     return 0;                       /* we do that in umount */
240 }
241
///// 安装根文件系统。
// 该函数是在系统开机初始化设置时(sys_setup())调用的。( kernel/blk_drv/hd.c, 157 )
242 void mount\_root(void)
243 {
244     int i, free;
245     struct super\_block * p;
246     struct m\_inode * mi;
247
// 如果磁盘 i 节点结构不是 32 个字节，则出错，死机。该判断是用于防止修改源代码时的一致性。
248     if (32 != sizeof (struct d\_inode))
249         panic("bad i-node size");
// 初始化文件表数组（共 64 项，也即系统同时只能打开 64 个文件），将所有文件结构中的引用计数
// 设置为 0。[??为什么放在这里初始化?]
250     for(i=0; i<NR\_FILE; i++)
251         file\_table[i].f_count=0;
// 如果根文件系统所在设备是软盘的话，就提示“插入根文件系统盘，并按回车键”，并等待按键。
252     if (MAJOR(ROOT\_DEV) == 2) {
253         printk("Insert root floppy and press ENTER");
254         wait\_for\_keypress();
255     }
// 初始化超级块数组（共 8 项）。
256     for(p = &super\_block[0] ; p < &super\_block[NR\_SUPER] ; p++) {
257         p->s_dev = 0;
258         p->s_lock = 0;
259         p->s_wait = NULL;
260     }
// 如果读设备上超级块失败，则显示信息，并死机。
261     if (!(p=read\_super(ROOT\_DEV)))
262         panic("Unable to mount root");
//从设备上读取文件系统的根 i 节点(1)，如果失败则显示出错信息，死机。

```

```

263         if (!(mi=iget(ROOT_DEV, ROOT_INO)))
264             panic("Unable to read root i-node");
// 该 i 节点引用次数递增 3 次。[?? Why?]
265         mi->i_count += 3;      /* NOTE! it is logically used 4 times, not 1 */
// 置该超级块的被安装文件系统 i 节点和被安装到的 i 节点为该 i 节点。
266         p->s_isup = p->s_imount = mi;
// 设置当前进程的当前工作目录和根目录 i 节点。
267         current->pwd = mi;
268         current->root = mi;
// 统计该设备上空闲块数。首先令 i 等于超级块中表明的设备逻辑块总数。
269         free=0;
270         i=p->s_nzones;
// 然后根据逻辑块位图中相应比特位的占用情况统计出空闲块数。宏函数 set_bit() 其实只是测试
// 比特位，而非设置比特位。"i&8191"用于取得 i 节点号在当前块中的偏移值。"i>>13"是将 i 除以
// 8192，也即除一个磁盘块包含的比特位数。
271         while (-- i >= 0)
272             if (!set_bit(i&8191, p->s_zmap[i>>13]->b_data))
273                 free++;
// 显示设备上空闲逻辑块数/逻辑块总数。
274         printk("%d/%d free blocks\n", free, p->s_nzones);
// 统计设备上空闲 i 节点数。首先令 i 等于超级块中表明的设备上 i 节点总数+1。加 1 是将 0 节点
// 也统计进去。[??]
275         free=0;
276         i=p->s_ninodes+1;
// 然后根据 i 节点位图中相应比特位的占用情况计算出空闲 i 节点数。
277         while (-- i >= 0)
278             if (!set_bit(i&8191, p->s_imap[i>>13]->b_data))
279                 free++;
// 显示设备上可用的空闲 i 节点数/i 节点总数。
280         printk("%d/%d free inodes\n", free, p->s_ninodes);
281     }
282

```

### 9.14.3 其它信息

## 9.15 open.c 文件

### 9.15.1 功能描述

### 9.15.2 代码注释

列表 linux/fs/open.c 程序

```

1  /*
2  *  linux/fs/open.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6

```

```

7 #include <string.h>          // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8 #include <errno.h>           // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
9 #include <fcntl.h>           // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
10 #include <sys/types.h>       // 类型头文件。定义了基本的系统数据类型。
11 #include <utime.h>           // 用户时间头文件。定义了访问和修改时间结构以及 utime() 原型。
12 #include <sys/stat.h>        // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
13
14 #include <linux/sched.h>      // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
                                // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
15 #include <linux/tty.h>       // tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。
16 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
17 #include <asm/segment.h>     // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
18
19 // 取文件系统信息函数调用函数。
20 int sys_ustat(int dev, struct ustat * ubuf)
21 {
22     return -ENOSYS;
23 }
24
25 // 设置文件访问和修改时间。
26 // 参数 filename 是文件名，times 是访问和修改时间结构指针。
27 // 如果 times 指针不为 NULL，则取 utimbuf 结构中的时间信息来设置文件的访问和修改时间。如果
28 // times 指针是 NULL，则取系统当前时间来设置指定文件的访问和修改时间域。
29 int sys_utime(char * filename, struct utimbuf * times)
30 {
31     struct m_inode * inode;
32     long actime, modtime;
33
34     // 根据文件名寻找对应的 i 节点，如果没有找到，则返回出错码。
35     if (!(inode=namei(filename)))
36         return -ENOENT;
37     // 如果访问和修改时间数据结构指针不为 NULL，则从结构中读取用户设置的时间值。
38     if (times) {
39         actime = get_fs_long((unsigned long *) &times->actime);
40         modtime = get_fs_long((unsigned long *) &times->modtime);
41     } else
42         // 否则将访问和修改时间置为当前时间。
43         actime = modtime = CURRENT_TIME;
44     // 修改 i 节点中的访问时间字段和修改时间字段。
45     inode->i_atime = actime;
46     inode->i_mtime = modtime;
47     // 置 i 节点已修改标志，释放该节点，并返回 0。
48     inode->i_dirt = 1;
49     iput(inode);
50     return 0;
51 }
52
53 /*
54  * XXX should we use the real or effective uid?  BSD uses the real uid,
55  * so as to make this call useful to setuid programs.
56  */
57 /*
58  * 文件属性 XXX，我们该用真实用户 id 还是有效用户 id? BSD 系统使用了真实用户 id，

```

```

    * 以使该调用可以供 setuid 程序使用。（注：POSIX 标准建议使用真实用户 ID）
    */
    ///// 检查对文件的访问权限。
    // 参数 filename 是文件名，mode 是屏蔽码，由 R_OK(4)、W_OK(2)、X_OK(1)和 F_OK(0)组成。
    // 如果请求访问允许的话，则返回 0，否则返回出错码。
47 int sys_access(const char * filename,int mode)
48 {
49     struct m_inode * inode;
50     int res, i_mode;
51
    // 屏蔽码由低 3 位组成，因此清除所有高比特位。
52     mode &= 0007;
    // 如果文件名对应的 i 节点不存在，则返回出错码。
53     if (!(inode=namei(filename)))
54         return -EACCES;
    // 取文件的属性码，并释放该 i 节点。
55     i_mode = res = inode->i_mode & 0777;
56     iput(inode);
    // 如果当前进程是该文件的宿主，则取文件宿主属性。
57     if (current->uid == inode->i_uid)
58         res >>= 6;
    // 否则如果当前进程是与该文件同属一组，则取文件组属性。
59     else if (current->gid == inode->i_gid)
60         res >>= 6;
    // 如果文件属性具有查询的属性位，则访问许可，返回 0。
61     if ((res & 0007 & mode) == mode)
62         return 0;
63     /*
64      * XXX we are doing this test last because we really should be
65      * swapping the effective with the real user id (temporarily),
66      * and then calling suser() routine. If we do call the
67      * suser() routine, it needs to be called last.
68      */
    /*
    * XXX 我们最后才做下面的测试，因为我们实际上需要交换有效用户 id 和
    * 真实用户 id（临时地），然后才调用 suser() 函数。如果我们确实要调用
    * suser() 函数，则需要最后才被调用。
    */
    // 如果当前用户 id 为 0（超级用户）并且屏蔽码执行位是 0 或文件可以被任何人访问，则返回 0。
69     if ((!current->uid) &&
70         (! (mode & 1) || (i_mode & 0111)))
71         return 0;
    // 否则返回出错码。
72     return -EACCES;
73 }
74
    ///// 改变当前工作目录系统调用函数。
    // 参数 filename 是目录名。
    // 操作成功则返回 0，否则返回出错码。
75 int sys_chdir(const char * filename)
76 {
77     struct m_inode * inode;
78

```

```

// 如果文件名对应的 i 节点不存在，则返回出错码。
79     if (!(inode = namei(filename)))
80         return -ENOENT;
// 如果该 i 节点不是目录的 i 节点，则释放该节点，返回出错码。
81     if (!S\_ISDIR(inode->i_mode)) {
82         iput(inode);
83         return -ENOTDIR;
84     }
// 释放当前进程原工作目录 i 节点，并指向该新置的工作目录 i 节点。返回 0。
85     iput(current->pwd);
86     current->pwd = inode;
87     return (0);
88 }
89
///// 改变根目录系统调用函数。
// 将指定的路径名改为根目录 '/'。
// 如果操作成功则返回 0，否则返回出错码。
90 int sys\_chroot(const char * filename)
91 {
92     struct m\_inode * inode;
93
94     // 如果文件名对应的 i 节点不存在，则返回出错码。
95     if (!(inode=namei(filename)))
96         return -ENOENT;
97     // 如果该 i 节点不是目录的 i 节点，则释放该节点，返回出错码。
98     if (!S\_ISDIR(inode->i_mode)) {
99         iput(inode);
100         return -ENOTDIR;
101     }
102     // 释放当前进程的根目录 i 节点，并重置为这里指定目录名的 i 节点，返回 0。
103     iput(current->root);
104     current->root = inode;
105     return (0);
106 }
107
///// 修改文件属性系统调用函数。
// 参数 filename 是文件名，mode 是新的文件属性。
// 若操作成功则返回 0，否则返回出错码。
108 int sys\_chmod(const char * filename,int mode)
109 {
110     struct m\_inode * inode;
111
112     // 如果文件名对应的 i 节点不存在，则返回出错码。
113     if (!(inode=namei(filename)))
114         return -ENOENT;
115     // 如果当前进程的有效用户 id 不等于文件 i 节点的用户 id，并且当前进程不是超级用户，则释放该
116     // 文件 i 节点，返回出错码。
117     if ((current->euid != inode->i_uid) && !suser()) {
118         iput(inode);
119         return -EACCES;
120     }
121     // 重新设置 i 节点的文件属性，并置该 i 节点已修改标志。释放该 i 节点，返回 0。
122     inode->i_mode = (mode & 07777) | (inode->i_mode & ~07777);

```

```

116     inode->i_dirt = 1;
117     iput(inode);
118     return 0;
119 }
120
121 // 修改文件宿主系统调用函数。
122 // 参数 filename 是文件名, uid 是用户标识符(用户 id), gid 是组 id。
123 // 若操作成功则返回 0, 否则返回出错码。
124 int sys\_chown(const char * filename, int uid, int gid)
125 {
126     struct m\_inode * inode;
127
128     // 如果文件名对应的 i 节点不存在, 则返回出错码。
129     if (!(inode=namei(filename)))
130         return -ENOENT;
131     // 若当前进程不是超级用户, 则释放该 i 节点, 返回出错码。
132     if (!suser()) {
133         iput(inode);
134         return -EACCES;
135     }
136     // 设置文件对应 i 节点的用户 id 和组 id, 并置 i 节点已经修改标志, 释放该 i 节点, 返回 0。
137     inode->i_uid=uid;
138     inode->i_gid=gid;
139     inode->i_dirt=1;
140     iput(inode);
141     return 0;
142 }
143
144 // 打开(或创建)文件系统调用函数。
145 // 参数 filename 是文件名, flag 是打开文件标志: 只读 O_RDONLY、只写 O_WRONLY 或读写 O_RDWR,
146 // 以及 O_CREAT、O_EXCL、O_APPEND 等其它一些标志的组合, 若本函数创建了一个新文件, 则 mode
147 // 用于指定使用文件的许可属性, 这些属性有 S_IRWXU(文件宿主具有读、写和执行权限)、S_IRUSR
148 // (用户具有读文件权限)、S_IRWXG(组成员具有读、写和执行权限)等等。对于新创建的文件, 这些
149 // 属性只应用于将来对文件的访问, 创建了只读文件的打开调用也将返回一个可读写的文件句柄。
150 // 若操作成功则返回文件句柄(文件描述符), 否则返回出错码。(参见 sys/stat.h, fcntl.h)
151 int sys\_open(const char * filename, int flag, int mode)
152 {
153     struct m\_inode * inode;
154     struct file * f;
155     int i, fd;
156
157     // 将用户设置的模式与进程的模式屏蔽码相与, 产生许可的文件模式。
158     mode &= 0777 & ~current->umask;
159     // 搜索进程结构中文件结构指针数组, 查找一个空闲项, 若已经没有空闲项, 则返回出错码。
160     for(fd=0 ; fd<NR\_OPEN ; fd++)
161         if (!current->filp[fd])
162             break;
163     if (fd>=NR\_OPEN)
164         return -EINVAL;
165     // 设置执行时关闭文件句柄位图, 复位对应比特位。
166     current->close_on_exec &= ~(1<<fd);
167     // 令 f 指向文件表数组开始处。搜索空闲文件结构项(句柄引用计数为 0 的项), 若已经没有空闲
168     // 文件表结构项, 则返回出错码。

```



```

151     f=0+file_table;
152     for (i=0 ; i<NR_FILE ; i++,f++)
153         if (!f->f_count) break;
154     if (i>=NR_FILE)
155         return -EINVAL;
156     // 让进程的对应文件句柄的文件结构指针指向搜索到的文件结构，并令句柄引用计数递增1。
157     (current->filp[fd]=f)->f_count++;
158     // 调用函数执行打开操作，若返回值小于0，则说明出错，释放刚申请到的文件结构，返回出错码。
159     if ((i=open_namei(filename, flag, mode, &inode))<0) {
160         current->filp[fd]=NULL;
161         f->f_count=0;
162         return i;
163     }
164     /* ttys are somewhat special (ttyxx major==4, tty major==5) */
165     /* ttys 有些特殊 (ttyxx 主号==4, tty 主号==5) */
166     // 如果是字符设备文件，那么如果设备号是 4 的话，则设置当前进程的 tty 号为该 i 节点的子设备号。
167     // 并设置当前进程 tty 对应的 tty 表项的父进程组号等于进程的父进程组号。
168     if (S_ISCHR(inode->i_mode))
169         if (MAJOR(inode->i_zone[0])==4) {
170             if (current->leader && current->tty<0) {
171                 current->tty = MINOR(inode->i_zone[0]);
172                 tty_table[current->tty].pgrp = current->pgrp;
173             }
174             // 否则如果该字符文件设备号是 5 的话，若当前进程没有 tty，则说明出错，释放 i 节点和申请到的
175             // 文件结构，返回出错码。
176             } else if (MAJOR(inode->i_zone[0])==5)
177                 if (current->tty<0) {
178                     iput(inode);
179                     current->filp[fd]=NULL;
180                     f->f_count=0;
181                     return -EPERM;
182                 }
183     /* Likewise with block-devices: check for floppy_change */
184     /* 同样对于块设备文件：需要检查盘片是否被更换 */
185     // 如果打开的是块设备文件，则检查盘片是否更换，若更换则需要是高速缓冲中对应该设备的所有
186     // 缓冲块失效。
187     if (S_ISBLK(inode->i_mode))
188         check_disk_change(inode->i_zone[0]);
189     // 初始化文件结构。置文件结构属性和标志，置句柄引用计数为 1，设置 i 节点字段，文件读写指针
190     // 初始化为 0。返回文件句柄。
191     f->f_mode = inode->i_mode;
192     f->f_flags = flag;
193     f->f_count = 1;
194     f->f_inode = inode;
195     f->f_pos = 0;
196     return (fd);
197 }
198
199 // 创建文件系统调用函数。
200 // 参数 pathname 是路径名，mode 与上面的 sys_open() 函数相同。
201 // 成功则返回文件句柄，否则返回出错码。
202 int sys_creat(const char * pathname, int mode)
203 {

```

```

189         return sys\_open(pathname, O\_CREAT | O\_TRUNC, mode);
190     }
191     // 关闭文件系统调用函数。
192     // 参数 fd 是文件句柄。
193     // 成功则返回 0，否则返回出错码。
194     int sys\_close(unsigned int fd)
195     {
196         struct file * filp;
197         // 若文件句柄值大于程序同时能打开的文件数，则返回出错码。
198         if (fd >= NR\_OPEN)
199             return -EINVAL;
200         // 复位进程的运行时关闭文件句柄位图对应位。
201         current->close_on_exec &= ~(1<<fd);
202         // 若该文件句柄对应的文件结构指针是 NULL，则返回出错码。
203         if (!(filp = current->filp[fd]))
204             return -EINVAL;
205         // 置该文件句柄的文件结构指针为 NULL。
206         current->filp[fd] = NULL;
207         // 若在关闭文件之前，对应文件结构中的句柄引用计数已经为 0，则说明内核出错，死机。
208         if (filp->f_count == 0)
209             panic("Close: file count is 0");
210         // 否则将对应文件结构的句柄引用计数减 1，如果还不为 0，则返回 0（成功）。若已等于 0，说明该
211         // 文件已经没有句柄引用，则释放该文件 i 节点，返回 0。
212         if (--filp->f_count)
213             return (0);
214         iput(filp->f_inode);
215         return (0);
216     }
217 }
218
219

```

### 9.15.3 其它信息

## 9.16 exec.c 程序

### 9.16.1 功能描述

本源程序实现对二进制可执行文件和 shell 脚本文件的加载与执行。其中主要的函数是函数 `do_execve()`，它是系统中断调用(int 0x80)功能号 `__NR_execve()` 调用的 C 处理函数，是 `exec()` 函数簇的主要实现函数。其主要功能为：

- 执行对参数和环境参数空间页面的初始化操作 —— 设置初始空间起始指针；初始化空间页面指针数组为 (NULL)；根据执行文件名取执行对象的 I 节点；计算参数个数和环境变量个数；检查文件类型，执行权限；
- 根据执行文件开始部分的头数据结构，对其中信息进行处理 —— 根据被执行文件 I 节点读取文件头部信息；若是 Shell 脚本程序（第一行以 #! 开始），则分析 Shell 程序名及其参数，并以被执行文件作为参数执行该执行的 Shell 程序；执行根据文件的幻数以及段长度等信息判断是否可执行；

- 对当前调用进程进行运行新文件前初始化操作 —— 指向新执行文件的 I 节点；复位信号处理句柄；根据头结构信息设置局部描述符基址和段长；设置参数和环境参数页面指针；修改进行各执行字段内容；
- 替换堆栈上原调用 `execve()` 程序的返回地址为新执行程序运行地址，运行新加载的程序。

`execve()` 函数有大量对参数和环境空间的处理操作，参数和环境空间共可有 `MAX_ARG_PAGES` 个页面，总长度可达 128kB 字节。在该空间中存放数据的方式类似于堆栈操作，即是从假设的 128kB 空间末端处逆向开始存放参数或环境变量字符串的。在初始时，程序定义了一个指向该空间末端(128kB-4 字节)处空间内偏移值 `p`，该偏移值随着存放数据的增多而后退，由图中可以看出，`p` 明确地指出了当前参数环境空间中还剩余多少可用空间。在分析程序中 `copy_string()` 函数时，可参照此图。

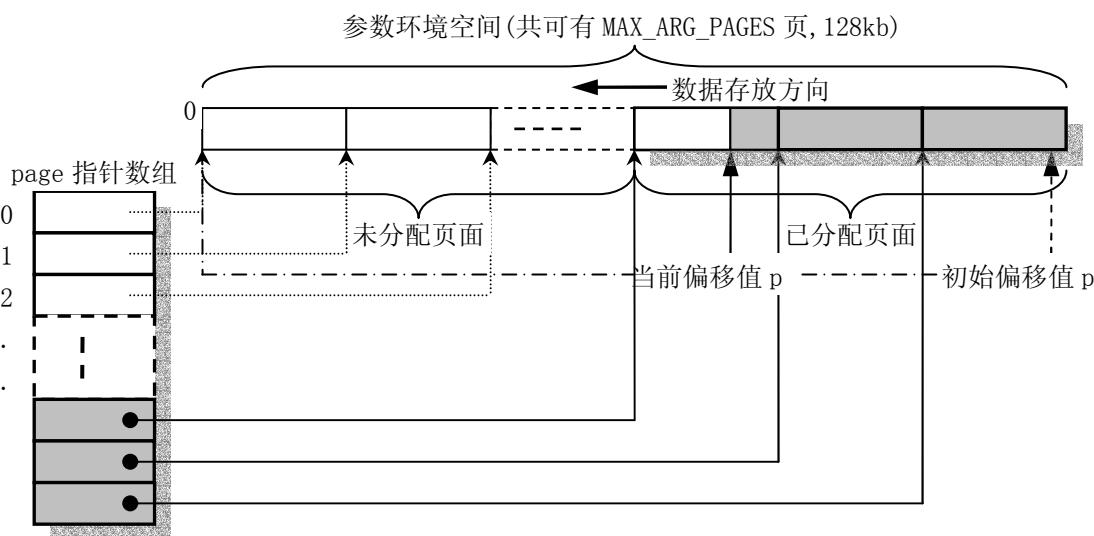


图 参数和环境变量字符串空间

`create_tables()` 函数用于根据给定的当前堆栈指针值 `p` 以及参数变量个数 `argc` 和环境变量个数 `envc`，在新的程序堆栈中创建环境和参数变量指针表，并返回此时的堆栈指针值 `sp`。创建完毕后堆栈指针表的形式见下图所示。

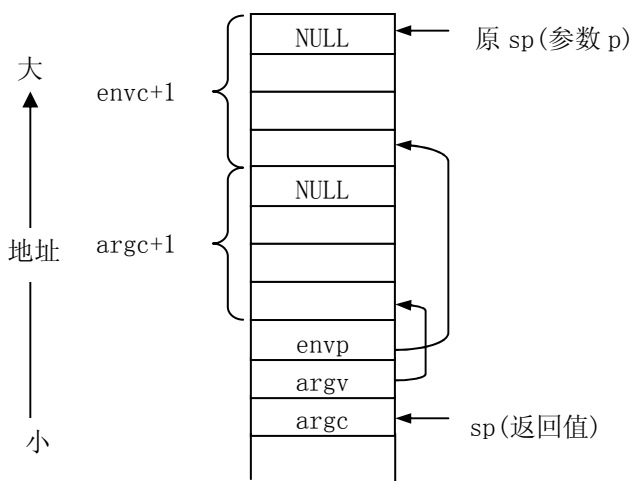


图 新程序堆栈中指针表示意图

## 9.16.2 代码注释

列表 linux/fs/exec.c 程序

```

1  /*
2   * linux/fs/exec.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * #-checking implemented by tytso.
9   */
10 /*
11  * #-开始的程序检测部分是由 tytso 实现的。
12  */
13
14 /*
15  * Demand-loading implemented 01.12.91 - no need to read anything but
16  * the header into memory. The inode of the executable is put into
17  * "current->executable", and page faults do the actual loading. Clean.
18  *
19  * Once more I can proudly say that linux stood up to being changed: it
20  * was less than 2 hours work to get demand-loading completely implemented.
21  */
22 /*
23  * 需求时加载是于 1991. 12. 1 实现的 - 只需将执行文件头部分读进内存而无须
24  * 将整个执行文件都加载进内存。执行文件的 i 节点被放在当前进程的可执行字段中
25  * ("current->executable"), 而页异常会进行执行文件的实际加载操作以及清理工作。
26  *
27  * 我可以再一次自豪地说, linux 经得起修改: 只用了不到 2 小时的工作时间就完全
28  * 实现了需求加载处理。
29  */
30
31 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
32 #include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
33 #include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
34 #include <a.out.h>         // a.out 头文件。定义了 a.out 执行文件格式和一些宏。
35
36 #include <linux/fs.h>      // 文件系统头文件。定义文件表结构 (file, buffer_head, m_inode 等)。
37 #include <linux/sched.h>   // 调度程序头文件, 定义了任务结构 task_struct、初始任务 0 的数据,
38 // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
39
40 #include <linux/kernel.h>  // 内核头文件。含有一些内核常用函数的原形定义。
41 #include <linux/mm.h>      // 内存管理头文件。含有页面大小定义和一些页面释放函数原型。
42 #include <asm/segment.h>   // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
43
44 extern int sys_exit(int exit_code); // 程序退出系统调用。
45 extern int sys_close(int fd);      // 文件关闭系统调用。
46
47 /*
48  * MAX_ARG_PAGES defines the number of pages allocated for arguments
49  * and envelope for the new program. 32 should suffice, this gives
50  * a maximum env+arg of 128kB !
51  */

```

```

/*
 * MAX_ARG_PAGES 定义了新程序分配给参数和环境变量使用的内存最大页数。
 * 32 页内存应该足够了，这使得环境和参数(env+arg)空间的总合达到 128kB!
 */
39 #define MAX_ARG_PAGES 32
40
41 /*
42  * create_tables() parses the env- and arg-strings in new user
43  * memory and creates the pointer tables from them, and puts their
44  * addresses on the "stack", returning the new stack pointer value.
45  */
/*
 * create_tables() 函数在新用户内存中解析环境变量和参数字符串，由此
 * 创建指针表，并将它们的地址放到“堆栈”上，然后返回新栈的指针值。
 */
//// 在新用户堆栈中创建环境和参数变量指针表。
// 参数: p - 以数据段为起点的参数和环境信息偏移指针; argc - 参数个数; envc - 环境变量数。
// 返回: 堆栈指针。
46 static unsigned long * create_tables(char * p, int argc, int envc)
47 {
48     unsigned long *argv, *envp;
49     unsigned long * sp;
50
51     // 堆栈指针是以 4 字节 (1 节) 为边界寻址的，因此这里让 sp 为 4 的整数倍。
52     sp = (unsigned long *) (0xffffffff & (unsigned long) p);
53     // sp 向下移动，空出环境参数占用的空间个数，并让环境参数指针 envp 指向该处。
54     sp -= envc+1;
55     envp = sp;
56     // sp 向下移动，空出命令行参数指针占用的空间个数，并让 argv 指针指向该处。
57     // 下面指针加 1，sp 将递增指针宽度字节值。
58     sp -= argc+1;
59     argv = sp;
60     // 将环境参数指针 envp 和命令行参数指针以及命令行参数个数压入堆栈。
61     put_fs_long((unsigned long)envp, --sp);
62     put_fs_long((unsigned long)argv, --sp);
63     put_fs_long((unsigned long)argc, --sp);
64     // 将命令行各参数指针放入前面空出来的相应地方，最后放置一个 NULL 指针。
65     while (argc-->0) {
66         put_fs_long((unsigned long) p, argv++);
67         while (get_fs_byte(p++)) /* nothing */; // p 指针前移 4 字节。
68     }
69     put_fs_long(0, argv);
70     // 将环境变量各指针放入前面空出来的相应地方，最后放置一个 NULL 指针。
71     while (envc-->0) {
72         put_fs_long((unsigned long) p, envp++);
73         while (get_fs_byte(p++)) /* nothing */;
74     }
75     put_fs_long(0, envp);
76     return sp; // 返回构造的当前新堆栈指针。
77 }
78
79 /*
80  * count() counts the number of arguments/envelopes

```

```

74  */
    /*
    * count() 函数计算命令行参数/环境变量的个数。
    */
    //// 计算参数个数。
    // 参数: argv - 参数指针数组, 最后一个指针项是 NULL。
    // 返回: 参数个数。
75 static int count(char ** argv)
76 {
77     int i=0;
78     char ** tmp;
79
80     if (tmp = argv)
81         while (get_fs_long((unsigned long *) (tmp++)))
82             i++;
83
84     return i;
85 }
86
87 /*
88  * 'copy_string()' copies argument/envelope strings from user
89  * memory to free pages in kernel mem. These are in a format ready
90  * to be put directly into the top of new user memory.
91  *
92  * Modified by TYT, 11/24/91 to add the from_kmem argument, which specifies
93  * whether the string and the string array are from user or kernel segments:
94  *
95  * from_kmem    argv *      argv **
96  * 0            user space  user space
97  * 1            kernel space user space
98  * 2            kernel space kernel space
99  *
100 * We do this by playing games with the fs segment register. Since it
101 * it is expensive to load a segment register, we try to avoid calling
102 * set_fs() unless we absolutely have to.
103 */
    /*
    * 'copy_string()' 函数从用户内存空间拷贝参数和环境字符串到内核空闲页面内存中。
    * 这些已具有直接放到新用户内存中的格式。
    *
    * 由 TYT(Tytso) 于 1991.12.24 日修改, 增加了 from_kmem 参数, 该参数指明了字符串或
    * 字符串数组是来自用户段还是内核段。
    *
    * from_kmem    argv *      argv **
    * 0            用户空间    用户空间
    * 1            内核空间    用户空间
    * 2            内核空间    内核空间
    *
    * 我们是通过巧妙处理 fs 段寄存器来操作的。由于加载一个段寄存器代价太大, 所以
    * 我们尽量避免调用 set_fs(), 除非实在必要。
    */
    //// 复制指定个数的参数字符串到参数和环境空间。
    // 参数: argc - 欲添加的参数个数; argv - 参数指针数组; page - 参数和环境空间页面指针数组。

```

```

//      p -在参数表空间中的偏移指针，始终指向已复制串的头；from_kmem - 字符串来源标志。
// 在 do_execve() 函数中，p 初始化为指向参数表(128kB)空间的最后一个长字处，参数字符串
// 是以堆栈操作方式逆向往其中复制存放的，因此 p 指针会始终指向参数字符串的头。
// 返回：参数和环境空间当前头部指针。
104 static unsigned long copy_strings(int argc, char ** argv, unsigned long *page,
105     unsigned long p, int from_kmem)
106 {
107     char *tmp, *pag;
108     int len, offset = 0;
109     unsigned long old_fs, new_fs;
110
111     if (!p)
112         return 0;      /* bullet-proofing */ /* 偏移指针验证 */
// 取 ds 寄存器值到 new_fs，并保存原 fs 寄存器值到 old_fs。
113     new_fs = get_ds();
114     old_fs = get_fs();
// 如果字符串和字符串数组来自内核空间，则设置 fs 段寄存器指向内核数据段 (ds)。
115     if (from_kmem==2)
116         set_fs(new_fs);
// 循环处理各个参数，从最后一个参数逆向开始复制，复制到指定偏移地址处。
117     while (argc-- > 0) {
// 如果字符串在用户空间而字符串数组在内核空间，则设置 fs 段寄存器指向内核数据段 (ds)。
118         if (from_kmem == 1)
119             set_fs(new_fs);
// 从最后一个参数开始逆向操作，取 fs 段中最后一参数指针到 tmp，如果为空，则出错死机。
120         if (!(tmp = (char *)get_fs_long(((unsigned long *)argv)+argc)))
121             panic("argc is wrong");
// 如果字符串在用户空间而字符串数组在内核空间，则恢复 fs 段寄存器原值。
122         if (from_kmem == 1)
123             set_fs(old_fs);
// 计算该参数字符串长度 len，并使 tmp 指向该参数字符串末端。
124         len=0;      /* remember zero-padding */
125         do {      /* 我们知道串是以 NULL 字节结尾的 */
126             len++;
127         } while (get_fs_byte(tmp++));
// 如果该字符串长度超过此时参数和环境空间中还剩余的空闲长度，则恢复 fs 段寄存器并返回 0。
128         if (p-len < 0) {      /* this shouldn't happen - 128kB */
129             set_fs(old_fs); /* 不会发生-因为有 128kB 的空间 */
130             return 0;
131         }
// 复制 fs 段中当前指定的参数字符串，是从该字符串尾逆向开始复制。
132         while (len) {
133             --p; --tmp; --len;
// 函数刚开始执行时，偏移变量 offset 被初始化为 0，因此若 offset-1<0，说明是首次复制字符串，
// 则令其等于 p 指针在页面内的偏移值，并申请空闲页面。
134             if (--offset < 0) {
135                 offset = p % PAGE_SIZE;
// 如果字符串和字符串数组在内核空间，则恢复 fs 段寄存器原值。
136                 if (from_kmem==2)
137                     set_fs(old_fs);
// 如果当前偏移值 p 所在的串空间页面指针数组项 page[p/PAGE_SIZE]==0，表示相应页面还不存在，
// 则需申请新的内存空闲页面，将该页面指针填入指针数组，并且也使 pag 指向该新页面，若申请不
// 到空闲页面则返回 0。

```



```

138             if (!(pag = (char *) page[p/PAGE_SIZE]) &&
139                 !(pag = (char *) page[p/PAGE_SIZE] =
140                     (unsigned long *) get_free_page()))
141                 return 0;
142 // 如果字符串和字符串数组来自内核空间, 则设置 fs 段寄存器指向内核数据段 (ds)。
143             if (from_kmem==2)
144                 set_fs(new_fs);
145         }
146 // 从 fs 段中复制参数字符串中一字节到 pag+offset 处。
147         *(pag + offset) = get_fs_byte(tmp);
148     }
149 // 如果字符串和字符串数组在内核空间, 则恢复 fs 段寄存器原值。
150     if (from_kmem==2)
151         set_fs(old_fs);
152 // 最后, 返回参数和环境空间中已复制参数信息的头部偏移值。
153     return p;
154 }
155
156 // 修改局部描述符表中的描述符基址和段限长, 并将参数和环境空间页面放置在数据段末端。
157 // 参数: text_size - 执行文件头部中 a_text 字段给出的代码段长度值;
158 //      page - 参数和环境空间页面指针数组。
159 // 返回: 数据段限长值 (64MB)。
160 static unsigned long change_ldt(unsigned long text_size, unsigned long * page)
161 {
162     unsigned long code_limit, data_limit, code_base, data_base;
163     int i;
164
165 // 根据执行文件头部 a_text 值, 计算以页面长度为边界的代码段限长。并设置数据段长度为 64MB。
166     code_limit = text_size+PAGE_SIZE -1;
167     code_limit &= 0xFFFFF000;
168     data_limit = 0x4000000;
169 // 取当前进程中局部描述符表代码段描述符中代码段基址, 代码段基址与数据段基址相同。
170     code_base = get_base(current->ldt[1]);
171     data_base = code_base;
172 // 重新设置局部表中代码段和数据段描述符的基址和段限长。
173     set_base(current->ldt[1], code_base);
174     set_limit(current->ldt[1], code_limit);
175     set_base(current->ldt[2], data_base);
176     set_limit(current->ldt[2], data_limit);
177 /* make sure fs points to the NEW data segment */
178 /* 要确信 fs 段寄存器已指向新的数据段 */
179 // fs 段寄存器中放入局部表数据段描述符的选择符 (0x17)。
180     __asm__ ("pushl $0x17\n\ttop %%fs");
181 // 将参数和环境空间已存放数据的页面 (共可有 MAX_ARG_PAGES 页, 128kB) 放到数据段线性地址的
182 // 末端。是调用函数 put_page() 进行操作的 (mm/memory.c, 197)。
183     data_base += data_limit;
184     for (i=MAX_ARG_PAGES-1; i>=0; i--) {
185         data_base -= PAGE_SIZE;
186         if (page[i]) // 如果该页面存在,
187             put_page(page[i], data_base); // 就放置该页面。
188     }

```

```

176         return data_limit;                                // 最后返回数据段限长(64MB)。
177     }
178
179     /*
180     * 'do_execve()' executes a new program.
181     */
182     /*
183     * 'do_execve()' 函数执行一个新程序。
184     */
185     /*// execve() 系统中断调用函数。加载并执行子进程（其它程序）。
186     // 该函数系统中断调用 (int 0x80) 功能号 __NR_execve 调用的函数。
187     // 参数: eip - 指向堆栈中调用系统中断的程序代码指针 eip 处, 参见 kernel/system_call.s 程序
188     // 开始部分的说明; tmp - 系统中断调用本函数时的返回地址, 无用;
189     //      filename - 被执行程序文件名; argv - 命令行参数指针数组; envp - 环境变量指针数组。
190     // 返回: 如果调用成功, 则不返回; 否则设置出错号, 并返回-1。
191 int do_execve(unsigned long * eip, long tmp, char * filename,
192               char ** argv, char ** envp)
193 {
194     struct m_inode * inode;                                // 内存中 I 节点指针结构变量。
195     struct buffer_head * bh;                               // 高速缓存块头指针。
196     struct exec ex;                                         // 执行文件头部数据结构变量。
197     unsigned long page[MAX_ARG_PAGES];                    // 参数和环境字符串空间的页面指针数组。
198     int i, argc, envc;
199     int e_uid, e_gid;                                       // 有效用户 id 和有效组 id。
200     int retval;                                             // 返回值。
201     int sh_bang = 0;                                        // 控制是否需要执行脚本处理代码。
202     // 参数和环境字符串空间中的偏移指针, 初始化为指向该空间的最后一个长字处。
203     unsigned long p = PAGE_SIZE * MAX_ARG_PAGES - 4;
204
205     // eip[1] 中是原代码段寄存器 cs, 其中的选择符不可以是内核段选择符, 也即内核不能调用本函数。
206     if ((0xffff & eip[1]) != 0x000f)
207         panic("execve called from supervisor mode");
208     // 初始化参数和环境串空间的页面指针数组 (表)。
209     for (i = 0; i < MAX_ARG_PAGES; i++)                    /* clear page-table */
210         page[i] = 0;
211     // 取可执行文件的对应 i 节点号。
212     if (!(inode = namei(filename)))                         /* get executables inode */
213         return -ENOENT;
214     // 计算参数个数和环境变量个数。
215     argc = count(argv);
216     envc = count(envp);
217
218     // 执行文件必须是常规文件。若不是常规文件则置出错返回码, 跳转到 exec_error2 (第 347 行)。
219 restart_interp:
220     if (!S_ISREG(inode->i_mode)) { /* must be regular file */
221         retval = -EACCES;
222         goto exec_error2;
223     }
224     // 检查被执行文件的执行权限。根据其属性 (对应 i 节点的 uid 和 gid), 看本进程是否有权执行它。
225     i = inode->i_mode;
226     e_uid = (i & S_ISUID) ? inode->i_uid : current->euid;
227     e_gid = (i & S_ISGID) ? inode->i_gid : current->egid;
228     if (current->euid == inode->i_uid)

```

```

213         i >>= 6;
214     else if (current->egid == inode->i_gid)
215         i >>= 3;
216     if (!(i & 1) &&
217         !((inode->i_mode & 0111) && suser())) {
218         retval = -ENOEXEC;
219         goto exec_error2;
220     }
221 // 读取执行文件的第一块数据到高速缓冲区, 若出错则置出错码, 跳转到 exec_error2 处去处理。
222     if (!(bh = bread(inode->i_dev, inode->i_zone[0]))) {
223         retval = -EACCES;
224         goto exec_error2;
225     }
226 // 下面对执行文件的头结构数据进行处理, 首先让 ex 指向执行头部分的数据结构。
227     ex = *((struct exec *) bh->b_data);    /* read exec-header */ /* 读取执行头部分 */
228 // 如果执行文件开始的两个字节为 '#!', 并且 sh_bang 标志没有置位, 则处理脚本文件的执行。
229     if ((bh->b_data[0] == '#') && (bh->b_data[1] == '!') && (!sh_bang)) {
230         /*
231          * This section does the #! interpretation.
232          * Sorta complicated, but hopefully it will work.  -TYT
233          */
234         /*
235          * 这部分处理对 '#!' 的解释, 有些复杂, 但希望能工作。-TYT
236          */
237         char buf[1023], *cp, *interp, *i_name, *i_arg;
238         unsigned long old_fs;
239
240         // 复制执行程序头一行字符 '#!' 后面的字符串到 buf 中, 其中含有脚本处理程序名。
241         strncpy(buf, bh->b_data+2, 1022);
242         // 释放高速缓冲块和该执行文件 i 节点。
243         brelse(bh);
244         iput(inode);
245         // 取第一行内容, 并删除开始的空格、制表符。
246         buf[1022] = '\0';
247         if (cp = strchr(buf, '\n')) {
248             *cp = '\0';
249             for (cp = buf; (*cp == ' ') || (*cp == '\t'); cp++);
250         }
251         // 若该行没有其它内容, 则出错。置出错码, 跳转到 exec_error1 处。
252         if (!cp || *cp == '\0') {
253             retval = -ENOEXEC; /* No interpreter name found */
254             goto exec_error1;
255         }
256         // 否则就得到了开头是脚本解释执行程序名称的一行内容。
257         interp = i_name = cp;
258         // 下面分析该行。首先取第一个字符串, 其应该是脚本解释程序名, i_name 指向该名称。
259         i_arg = 0;
260         for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++) {
261             if (*cp == '/')
262                 i_name = cp+1;
263         }
264         // 若文件名后还有字符, 则应该是参数串, 令 i_arg 指向该串。

```

```

253         if (*cp) {
254             *cp++ = '\0';
255             i_arg = cp;
256         }
257         /*
258          * OK, we've parsed out the interpreter name and
259          * (optional) argument.
260          */
261         /*
262          * OK, 我们已经解析出解释程序的文件名以及(可选的)参数。
263          */
264         // 若 sh_bang 标志没有设置, 则设置它, 并复制指定个数的环境变量串和参数串到参数和环境空间中。
265         if (sh_bang++ == 0) {
266             p = copy_strings(envc, envp, page, p, 0);
267             p = copy_strings(--argc, argv+1, page, p, 0);
268         }
269         /*
270          * Splice in (1) the interpreter's name for argv[0]
271          *          (2) (optional) argument to interpreter
272          *          (3) filename of shell script
273          *
274          * This is done in reverse order, because of how the
275          * user environment and arguments are stored.
276          */
277         /*
278          * 拼接 (1) argv[0] 中放解释程序的名称
279          *          (2) (可选的)解释程序的参数
280          *          (3) 脚本程序的名称
281          *
282          * 这是以逆序进行处理的, 是由于用户环境和参数的存放方式造成的。
283          */
284         // 复制脚本程序文件名到参数和环境空间中。
285         p = copy_strings(1, &filename, page, p, 1);
286         // 复制解释程序的参数到参数和环境空间中。
287         argc++;
288         if (i_arg) {
289             p = copy_strings(1, &i_arg, page, p, 2);
290             argc++;
291         }
292         // 复制解释程序文件名到参数和环境空间中。若出错, 则置出错码, 跳转到 exec_error1。
293         p = copy_strings(1, &i_name, page, p, 2);
294         argc++;
295         if (!p) {
296             retval = -ENOMEM;
297             goto exec_error1;
298         }
299         /*
300          * OK, now restart the process with the interpreter's inode.
301          */
302         /*
303          * OK, 现在使用解释程序的 i 节点重启进程。
304          */
305         // 保留原 fs 段寄存器 (原指向用户数据段), 现置其指向内核数据段。

```

```

288         old_fs = get\_fs();
289         set\_fs(get\_ds());
// 取解释程序的 i 节点，并跳转到 restart_interp 处重新处理。
290         if (!(inode=namei(interp))) { /* get executables inode */
291             set\_fs(old_fs);
292             retval = -ENOENT;
293             goto exec_error1;
294         }
295         set\_fs(old_fs);
296         goto restart_interp;
297     }
// 释放该缓冲区。
298     brelse(bh);
// 下面对执行头信息进行处理。
// 对于下列情况，将不执行程序：如果执行文件不是需求页可执行文件(ZMAGIC)、或者代码重定位部分
// 长度 a_trsize 不等于 0、或者数据重定位信息长度不等于 0、或者代码段+数据段+堆段长度超过 50MB、
// 或者 i 节点表明的该执行文件长度小于代码段+数据段+符号表长度+执行头部分长度的总和。
299     if (N\_MAGIC(ex) != ZMAGIC || ex.a_trsize || ex.a_drsize ||
300         ex.a_text+ex.a_data+ex.a_bss>0x3000000 ||
301         inode->i_size < ex.a_text+ex.a_data+ex.a_syms+N\_TXTOFF(ex)) {
302         retval = -ENOEXEC;
303         goto exec_error2;
304     }
// 如果执行文件执行头部分长度不等于一个内存块大小（1024 字节），也不能执行。转 exec_error2。
305     if (N\_TXTOFF(ex) != BLOCK\_SIZE) {
306         printk("%s: N\_TXTOFF != BLOCK\_SIZE. See a.out.h.", filename);
307         retval = -ENOEXEC;
308         goto exec_error2;
309     }
// 如果 sh_bang 标志没有设置，则复制指定个数的环境变量字符串和参数到参数和环境空间中。
// 若 sh_bang 标志已经设置，则表明是将运行脚本程序，此时环境变量页面已经复制，无须再复制。
310     if (!sh_bang) {
311         p = copy\_strings(envc, envp, page, p, 0);
312         p = copy\_strings(argc, argv, page, p, 0);
// 如果 p=0，则表示环境变量与参数空间页面已经被占满，容纳不下了。转至出错处理处。
313         if (!p) {
314             retval = -ENOMEM;
315             goto exec_error2;
316         }
317     }
318     /* OK, This is the point of no return */
// OK，下面开始就没有返回的地方了 */
// 如果原程序也是一个执行程序，则释放其 i 节点，并让进程 executable 字段指向新程序 i 节点。
319     if (current->executable)
320         iput(current->executable);
321     current->executable = inode;
// 清复位所有信号处理句柄。但对于 SIG_IGN 句柄不能复位，因此在 322 与 323 行之间需添加一条
// if 语句：if (current->sa[I].sa_handler != SIG_IGN)。这是源代码中的一个 bug。
322     for (i=0 ; i<32 ; i++)
323         current->sigaction[i].sa_handler = NULL;
// 根据执行时关闭(close_on_exec)文件句柄位图标志，关闭指定的打开文件，并复位该标志。
324     for (i=0 ; i<NR\_OPEN ; i++)
325         if ((current->close_on_exec>>i)&1)

```

```

326         sys_close(i);
327         current->close_on_exec = 0;
    // 根据指定的基址和限长，释放原来程序代码段和数据段所对应的内存页表指定的内存块及页表本
    身。
328         free_page_tables(get_base(current->ldt[1]), get_limit(0x0f));
329         free_page_tables(get_base(current->ldt[2]), get_limit(0x17));
    // 如果“上次任务使用了协处理器”指向的是当前进程，则将其置空，并复位使用了协处理器的标志。
330         if (last_task_used_math == current)
331             last_task_used_math = NULL;
332         current->used_math = 0;
    // 根据 a_text 修改局部表中描述符基址和段限长，并将参数和环境空间页面放置在数据段末端。
    // 执行下面语句之后，p 此时是以数据段起始处为原点的偏移值，仍指向参数和环境空间数据开始处，
    // 也即转换成为堆栈的指针。
333         p += change_ldt(ex.a_text, page) - MAX_ARG_PAGES * PAGE_SIZE;
    // create_tables() 在新用户堆栈中创建环境和参数变量指针表，并返回该堆栈指针。
334         p = (unsigned long) create_tables((char *)p, argc, envc);
    // 修改当前进程各字段为新执行程序的信息。令进程代码段尾值字段 end_code = a_text；令进程数据
    // 段尾字段 end_data = a_data + a_text；令进程堆结尾字段 brk = a_text + a_data + a_bss。
335         current->brk = ex.a_bss +
336             (current->end_data = ex.a_data +
337             (current->end_code = ex.a_text));
    // 设置进程堆栈开始字段为堆栈指针所在的页面，并重新设置进程的用户 id 和组 id。
338         current->start_stack = p & 0xfffff000;
339         current->euid = e_uid;
340         current->egid = e_gid;
    // 初始化一页 bss 段数据，全为零。
341         i = ex.a_text + ex.a_data;
342         while (i & 0xfff)
343             put_fs_byte(0, (char *) (i++));
    // 将原调用系统中断的程序在堆栈上的代码指针替换为指向新执行程序的入口点，并将堆栈指针替换
    // 为新执行程序的堆栈指针。返回指令将弹出这些堆栈数据并使得 CPU 去执行新的执行程序，因此不会
    // 返回到原调用系统中断的程序中去了。
344         eip[0] = ex.a_entry;          /* eip, magic happens :-) */ /* eip, 魔法起作用了 */
345         eip[3] = p;                  /* stack pointer */          /* esp, 堆栈指针 */
346         return 0;
347 exec_error2:
348         iput(inode);
349 exec_error1:
350         for (i = 0; i < MAX_ARG_PAGES; i++)
351             free_page(page[i]);
352         return(retval);
353 }
354

```

## 9.16.3 其它信息

### 9.16.3.1 a.out 执行文件格式

Linux 内核 0.11 版仅支持 a.out (Assembly & link editor output) 执行文件格式，虽然这种格式目前已经渐渐不用，而使用功能更为齐全的 ELF (Executable and Link Format) 格式，但是由于其简单性，作为学习入门的材料正好比较适用。下面全面介绍一下 a.out 格式。

在头文件<a.out.h>中声明了三个数据结构以及一些宏函数。这些数据结构描述了系统上可执行的机器码文件（二进制文件）。

一个执行文件共可有七个部分（七节）组成。按照顺序，这些部分是：

执行头部分(exec header)

执行文件头部分。该部分中含有一些参数，内核使用这些参数将执行文件加载到内存中并执行，而链接程序(ld)使用这些参数将一些二进制目标文件组合成一个可执行文件。这是唯一必要的组成部分。

代码段部分(text segment)

含有程序执行使被加载到内存中的指令代码和相关数据。可以以只读形式进行加载。

数据段部分(data segment)

这部分含有已经初始化过的数据，总是被加载到可读写的内存中。

代码重定位部分(text relocations)

这部分含有供链接程序使用的记录数据。在组合二进制目标文件时用于定位代码段中的指针或地址。

数据重定位部分(data relocations)

与代码重定位部分的作用类似，但是是用于数据段中指针的重定位。

符号表部分(symbol table)

这部分同样含有供链接程序使用的记录数据，用于在二进制目标文件之间对命名的变量和函数（符号）进行交叉引用。

字符串表部分(string table)

该部分含有与符号名相对应的字符串。

每个二进制执行文件均以执行数据结构（exec structure）开始。该数据结构的形式如下：

```
struct exec {
    unsigned long a_midmag;
    unsigned long a_text;
    unsigned long a_data;
    unsigned long a_bss;
    unsigned long a_syms;
    unsigned long a_entry;
    unsigned long a_trsize;
    unsigned long a_drsize;
};
```

各个字段的功能如下：

a\_midmag 该字段含有被 N\_GETFLAG()、N\_GETMID 和 N\_GETMAGIC() 访问的子部分，是由链接程序

在运行时加载到进程地址空间。宏 `N_GETMID()` 用于返回机器标识符 (machine-id)，指示二进制文件将在什么机器上运行。`N_GETMAGIC()` 宏指明魔数，它唯一地确定了二进制执行文件与其它加载的文件之间的区别。字段中必须包含以下值之一：

段 `OMAGIC` 表示代码和数据段紧随在执行头后面并且是连续存放的。内核将代码和数据都加载到可读写内存中。

写内存 `NMAGIC` 同 `OMAGIC` 一样，代码和数据段紧随在执行头后面并且是连续存放的。然而内核将代码加载到了只读内存中，并把数据段加载到代码段后下一页可读边界开始。

据 `ZMAGIC` 内核在必要时从二进制执行文件中加载独立的页面。执行头部、代码段和数据段都被链接程序处理成多个页面大小的块。内核加载的代码页面时只读的，而数据段的页面是可写的。

`a_text` 该字段含有代码段的长度值，字节数。

`a_data` 该字段含有数据段的长度值，字节数。

`a_bss` 含有‘bss 段’的长度，内核用其设置在数据段后初始的 break (brk)。内核在加载程序时，这段可写内存显现出处于数据段后面，并且初始时为全零。

`a_syms` 含有符号表部分的字节长度值。

`a_entry` 含有内核将执行文件加载到内存中以后，程序执行起始点的内存地址。

`a_trsize` 该字段含有代码重定位表的大小，是字节数。

`a_drsize` 该字段含有数据重定位表的大小，是字节数。

在 `a.out.h` 头文件中定义了几个宏，这些宏使用 `exec` 结构来测试一致性或者定位执行文件中各个部分（节）的位置偏移值。这些宏有：

`N_BADMAG(exec)` 如果 `a_magic` 字段不能被识别，则返回非零值。

`N_TXTOFF(exec)` 代码段的起始位置字节偏移值。

`N_DATOFF(exec)` 数据段的起始位置字节偏移值。

`N_DRELOFF(exec)` 数据重定位表的起始位置字节偏移值。

`N_TRELOFF(exec)` 代码重定位表的起始位置字节偏移值。

`N_SYMOFF(exec)` 符号表的起始位置字节偏移值。

`N_STROFF(exec)` 字符串表的起始位置字节偏移值。



重定位记录具有标准格式，它使用重定位信息(relocation\_info)结构来描述：

```
struct relocation_info {
    int            r_address;
    unsigned int   r_symbolnum : 24,
                  r_pcrel : 1,
                  r_length : 2,
                  r_extern : 1,
                  r_baserel : 1,
                  r_jmptable : 1,
                  r_relative : 1,
                  r_copy : 1;
};
```

该结构中各字段的含义如下：

**r\_address**            该字段含有需要链接程序处理（编辑）的指针的字节偏移值。代码重定位的偏移值是从代码段开始处计数的，数据重定位的偏移值是从数据段开始处计算的。链接程序会将已经存储在该偏移处的值与使用重定位记录计算出的新值相加。

**r\_symbolnum**        该字段含有符号表中一个符号结构的序号值（不是字节偏移值）。链接程序在算出符号的绝对地址以后，就将该地址加到正在进行重定位的指针上。（如果 r\_extern 比特位是 0，那么情况就不同，见下面。）

**r\_pcrel**            如果设置了该位，链接程序就认为正在更新一个指针，该指针使用 pc 相关寻址方式，是属于机器码指令部分。当运行程序使用这个被重定位的指针时，该指针的地址被隐式地加到该指针上。

**r\_length**            该字段含有指针长度的 2 的次方值：0 表示 1 字节长，1 表示 2 字节长，2 表示 4 字节长。

**r\_extern**            如果被置位，表示该重定位需要一个外部引用；此时链接程序必须使用一个符号地址来更新相应指针。当该位是 0 时，则重定位是“局部”的；链接程序更新指针以反映各个段加载地址中的变化，而不是反映一个符号值的变化（除非同时设置了 r\_baserel，见下面）。在这种情况下，r\_symbolnum 字段的内容是一个 n\_type 值（见下面）；这类字段告诉链接程序被重定位的指针指向那个段。

**r\_baserel**            如果设置了该位，则 r\_symbolnum 字段指定的符号将被重定位成全局偏移表(Global Offset Table)中的一个偏移值。在运行时刻，全局偏移表该偏移处被设置为符号的地址。

**r\_jmptable**        如果被置位，则 r\_symbolnum 字段指定的符号将被重定位成过程链接表(Procedure Linkage Table)中的一个偏移值。

**r\_relative**        如果被置位，则说明此重定位与该目标文件将成为其组成部分的映象文件在运行时被加载的地址相关。这类重定位仅在共享目标文件中出现。

`r_copy` 如果被置位, 该重定位记录指定了一个符号, 该符号的内容将被复制到 `r_address` 指定的地方。该复制操作是通过共享目标模块中一个合适的数据项中的运行时刻链接程序完成的。

符号将名称映射为地址 (或者更通俗地讲是字符串映射到值)。由于链接程序对地址的调整, 一个符号的名称必须用来表示其地址, 直到已被赋予一个绝对地址值。符号是由符号表中固定长度的记录以及字符串表中的可变长度名称组成。符号表是 `nlist` 结构的一个数组, 如下所示:

```
struct nlist {
    union {
        char    *n_name;
        long    n_strx;
    } n_un;
    unsigned char n_type;
    char          n_other;
    short         n_desc;
    unsigned long  n_value;
};
```

其中各字段的含义为:

`n_un.n_strx` 含有本符号的名称在字符串表中的字节偏移值。当程序使用 `nlist()` 函数访问一个符号表时, 该字段被替换为 `n_un.n_name` 字段, 这是内存中字符串的指针。

`n_type` 用于链接程序确定如何更新符号的值。使用位屏蔽(bitmasks)可以将 `n_type` 字段分割成三个子字段, 对于 `N_EXT` 类型位置位的符号, 链接程序将它们看作是“外部的”符号, 并且允许其它二进制目标文件对它们的引用。`N_TYPE` 屏蔽码用于链接程序感兴趣的比特位:

同  为一个 没有定 标文件	<code>N_UNDF</code> 一个未定义的符号。链接程序必须在其它二进制目标文件中定位一个具有相名称的外部符号, 以确定该符号的绝对数据值。特殊情况下, 如果 <code>n_type</code> 字段是非零值, 并且没有二进制文件定义了这个符号, 则链接程序在 BSS 段中将该符号解析地址, 保留长度等于 <code>n_value</code> 的字节。如果符号在多于一个二进制目标文件中都定义并且这些二进制目标文件对其长度值不一致, 则链接程序将选择所有二进制目标文件中最大的长度。
----------------------------	---

`N_ABS` 一个绝对符号。链接程序不会更新一个绝对符号。

会	<code>N_TEXT</code> 一个代码符号。该符号的值是代码地址, 链接程序在合并二进制目标文件时更新其值。
---	---

始加载	<code>N_DATA</code> 一个数据符号; 与 <code>N_TEXT</code> 类似, 但是用于数据地址。对应代码和数据符号的值不是文件的偏移值而是地址; 为了找出文件的偏移, 就有必要确定相关部分开地址并减去它, 然后加上该部分的偏移。
-----	---

偏 N\_BSS 一个 BSS 符号；与代码或数据符号类似，但在二进制目标文件中没有对应的移。

进 N\_FN 一个文件名符号。在合并二进制目标文件时，链接程序会将该符号插入在二进制文件中的符号之前。符号的名称就是给予链接程序的文件名，而其值是二进制文件首个代码段地址。链接和加载时不需要文件名符号，但对于调试程序非常有用。

N\_STAB 屏蔽码用于选择符号调试程序(例如 gdb)感兴趣的位；其值在 stab() 中说明。

n\_other 该字段按照 n\_type 确定的段，提供有关符号重定位操作的符号独立性信息。目前，n\_other 字段的最低 4 位含有两个值之一：AUX\_FUNC 和 AUX\_OBJECT (有关定义参见<link.h>)。AUX\_FUNC 将符号与可调用的函数相关，AUX\_OBJECT 将符号与数据相关，而不管它们是位于代码段还是数据段。该字段主要用于链接程序 ld，用于动态可执行程序创建。

n\_desc 保留给调试程序使用；链接程序不对其进行处理。不同的调试程序将该字段用作不同的用途。

n\_value 含有符号的值。对于代码、数据和 BSS 符号，这是一个地址；对于其它符号（例如调试程序符号），值可以是任意的。

字符串表是由长度为 u\_int32\_t 后跟一 null 结尾的符号字符串组成。长度代表整个表的字节大小，所以在 32 位的机器上其最小值（或者是第 1 个字符串的偏移）总是 4。

## 9.17 namei.c 文件

### 9.17.1 功能描述

### 9.17.2 代码注释

列表 linux/fs/namei.c 程序

---

```

1  /*
2   * linux/fs/namei.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7  /*
8   * Some corrections by tytso.
9   */
10  /*
    * tytso 作了一些纠正。
    */

```

```

11 #include <linux/sched.h> // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
12 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
13 #include <asm/segment.h> // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14
15 #include <string.h> // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
16 #include <fcntl.h> // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
17 #include <errno.h> // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
18 #include <const.h> // 常数符号头文件。目前仅定义了 i 节点中 i_mode 字段的各标志位。
19 #include <sys/stat.h> // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
20
21 #define ACC_MODE(x) ("004|002|006|377"[(x)&0_ACCMODE]) // 访问模式宏(前面是 8 进制数)。
22
23 /*
24  * comment out this line if you want names > NAME_LEN chars to be
25  * truncated. Else they will be disallowed.
26  */
    /*
    * 如果想让文件名长度>NAME_NAME 的字符被截掉，就将下面定义注释掉。
    */
27 /* #define NO_TRUNCATE */
28
29 #define MAY_EXEC 1 // 可执行(可进入)。
30 #define MAY_WRITE 2 // 可写。
31 #define MAY_READ 4 // 可读。
32
33 /*
34  * permission()
35  *
36  * is used to check for read/write/execute permissions on a file.
37  * I don't know if we should look at just the euid or both euid and
38  * uid, but that should be easily changed.
39  */
    /*
    * permission()
    * 该函数用于检测一个文件的读/写/执行权限。我不知道是否只需检查 euid 还是
    * 需要检查 euid 和 uid 两者，不过这很容易修改。
    */
    /* 检测文件访问许可权限。
    // 参数: inode - 文件对应的 i 节点; mask - 访问属性屏蔽码。
    // 返回: 访问许可返回 1，否则返回 0。
40 static int permission(struct m_inode * inode, int mask)
41 {
42     int mode = inode->i_mode;
43
44     /* special case: not even root can read/write a deleted file */
    /* 特殊情况: 即使是超级用户(root)也不能读/写一个已被删除的文件 */
    // 如果 i 节点有对应的设备，但该 i 节点的连接数等于 0，则返回。
45     if (inode->i_dev && !inode->i_nlinks)
46         return 0;
    // 否则，如果进程的有效用户 id(euid)与 i 节点的用户 id 相同，则取文件宿主的用户访问权限。
47     else if (current->euid==inode->i_uid)
48         mode >>= 6;

```

```

// 否则, 如果进程的有效组 id(egid)与 i 节点的组 id 相同, 则取组用户的访问权限。
49     else if (current->egid==inode->i_gid)
50         mode >>= 3;
// 如果上面所取的访问权限与屏蔽码相同, 或者是超级用户, 则返回 1, 否则返回 0。
51     if (((mode & mask & 0007) == mask) || suser())
52         return 1;
53     return 0;
54 }
55
56 /*
57  * ok, we cannot use strncmp, as the name is not in our data space.
58  * Thus we'll have to use match. No big problem. Match also makes
59  * some sanity tests.
60  *
61  * NOTE! unlike strncmp, match returns 1 for success, 0 for failure.
62  */
/*
 * ok, 我们不能使用 strncmp 字符串比较函数, 因为名称不在我们的数据空间(不在内核空间)。
 * 因而我们只能使用 match()。问题不大。match() 同样也处理一些完整的测试。
 *
 * 注意! 与 strncmp 不同的是 match() 成功时返回 1, 失败时返回 0。
 */
///// 指定长度字符串比较函数。
// 参数: len - 比较的字符串长度; name - 文件名指针; de - 目录项结构。
// 返回: 相同返回 1, 不同返回 0。
63 static int match(int len, const char * name, struct dir\_entry * de)
64 {
65     register int same __asm__ ("ax");
66
// 如果目录项指针空, 或者目录项 i 节点等于 0, 或者要比较的字符串长度超过文件名长度, 则返回 0。
67     if (!de || !de->inode || len > NAME\_LEN)
68         return 0;
// 如果要比较的长度 len 小于 NAME_LEN, 但是目录项中文件名长度超过 len, 则返回 0。
69     if (len < NAME\_LEN && de->name[len])
70         return 0;
// 下面嵌入汇编语句, 在用户数据空间(fs)执行字符串的比较操作。
// %0 - eax(比较结果 same); %1 - eax(eax 初值 0); %2 - esi(名字指针); %3 - edi(目录项名指针);
// %4 - ecx(比较的字节长度值 len)。
71     __asm__ ("cld\n\t"                // 清方向位。
72             "fs ; repe ; cmpsb\n\t"   // 用户空间执行循环比较[esi++]和[edi++]操作,
73             "setz %%al"               // 若比较结果一样(z=0)则设置 al=1(same=eax)。
74             : "=a" (same)
75             : "" (0), "S" ((long) name), "D" ((long) de->name), "c" (len)
76             : "cx", "di", "si");
77     return same;                      // 返回比较结果。
78 }
79
80 /*
81  *      find_entry()
82  *
83  * finds an entry in the specified directory with the wanted name. It
84  * returns the cache buffer in which the entry was found, and the entry
85  * itself (as a parameter - res_dir). It does NOT read the inode of the

```

```

86  * entry - you'll have to do that yourself if you want to.
87  *
88  * This also takes care of the few special cases due to '..'-traversal
89  * over a pseudo-root and a mount point.
90  */
/*
 *      find_entry()
 * 在指定的目录中寻找一个与名字匹配的目录项。返回一个含有找到目录项的高速
 * 缓冲区以及目录项本身(作为一个参数 - res_dir)。并不读目录项的 i 节点 - 如
 * 果需要的话需自己操作。
 *
 * '..' 目录项, 操作期间也会对几种特殊情况分别处理 - 比如横越一个伪根目录以
 * 及安装点。
 */
///// 查找指定目录和文件名的目录项。
// 参数: dir - 指定目录 i 节点的指针; name - 文件名; namelen - 文件名长度;
// 返回: 高速缓冲区指针; res_dir - 返回的目录项结构指针;
91 static struct buffer head * find entry(struct m\_inode ** dir,
92      const char * name, int namelen, struct dir entry ** res_dir)
93 {
94     int entries;
95     int block, i;
96     struct buffer head * bh;
97     struct dir entry * de;
98     struct super block * sb;
99
// 如果定义了 NO_TRUNCATE, 则若文件名长度超过最大长度 NAME_LEN, 则返回。
100 #ifdef NO_TRUNCATE
101     if (namelen > NAME\_LEN)
102         return NULL;
//如果没有定义 NO_TRUNCATE, 则若文件名长度超过最大长度 NAME_LEN, 则截短之。
103 #else
104     if (namelen > NAME\_LEN)
105         namelen = NAME\_LEN;
106 #endif
// 计算本目录中目录项项数 entries。置空返回目录项结构指针。
107     entries = (*dir)->i_size / (sizeof (struct dir entry));
108     *res_dir = NULL;
// 如果文件名长度等于 0, 则返回 NULL, 退出。
109     if (!namelen)
110         return NULL;
111 /* check for '..', as we might have to do some "magic" for it */
/* 检查目录项 '..', 因为可能需要对其特别处理 */
112     if (namelen==2 && get fs byte(name)=='.' && get fs byte(name+1)=='.') {
113 /* '..' in a pseudo-root results in a faked '.' (just change namelen) */
/* 伪根中的 '..' 如同一个假 '.' (只需改变名字长度) */
// 如果当前进程的根节点指针即是指定的目录, 则将文件名修改为 '.',
114         if ((*dir) == current->root)
115             namelen=1;
// 否则如果该目录的 i 节点号等于 ROOT_INO(1) 的话, 说明是文件系统根节点。则取文件系统的超级块。

116         else if ((*dir)->i_num == ROOT\_INO) {
117 /* '..' over a mount-point results in 'dir' being exchanged for the mounted

```

```

118 directory-inode. NOTE! We set mounted, so that we can iput the new dir */
/* 在一个安装点上的'..'将导致目录交换到安装到文件系统的目录 i 节点。
   注意！由于设置了 mounted 标志，因而我们能够取出该新目录 */
119         sb=get_super((*dir)->i_dev);
// 如果被安装到的 i 节点存在，则先释放原 i 节点，然后对被安装到的 i 节点进行处理。
// 让*dir 指向该被安装到的 i 节点；该 i 节点的引用数加 1。
120         if (sb->s_imount) {
121             iput(*dir);
122             (*dir)=sb->s_imount;
123             (*dir)->i_count++;
124         }
125     }
126 }
// 如果该 i 节点所指向的第一个直接磁盘块号为 0，则返回 NULL，退出。
127     if (!(block = (*dir)->i_zone[0]))
128         return NULL;
// 从节点所在设备读取指定的目录项数据块，如果不成功，则返回 NULL，退出。
129     if (!(bh = bread((*dir)->i_dev, block)))
130         return NULL;
// 在目录项数据块中搜索匹配指定文件名的目录项，首先让 de 指向数据块，并在不超过目录中目录项
数
// 的条件下，循环执行搜索。
131     i = 0;
132     de = (struct dir_entry *) bh->b_data;
133     while (i < entries) {
// 如果当前目录项数据块已经搜索完，还没有找到匹配的目录项，则释放当前目录项数据块。
134         if ((char *)de >= BLOCK_SIZE+bh->b_data) {
135             brelse(bh);
136             bh = NULL;
// 在读入下一目录项数据块。若这块为空，则只要还没有搜索完目录中的所有目录项，就跳过该块，
// 继续读下一目录项数据块。若该块不空，就让 de 指向该目录项数据块，继续搜索。
137             if (!(block = bmap(*dir, i/DIR_ENTRIES_PER_BLOCK)) ||
138                 !(bh = bread((*dir)->i_dev, block))) {
139                 i += DIR_ENTRIES_PER_BLOCK;
140                 continue;
141             }
142             de = (struct dir_entry *) bh->b_data;
143         }
// 如果找到匹配的目录项的话，则返回该目录项结构指针和该目录项数据块指针，退出。
144         if (match(namelen, name, de)) {
145             *res_dir = de;
146             return bh;
147         }
// 否则继续在目录项数据块中比较下一个目录项。
148         de++;
149         i++;
150     }
// 若指定目录中的所有目录项都搜索完还没有找到相应的目录项，则释放目录项数据块，返回 NULL。
151     brelse(bh);
152     return NULL;
153 }
154
155 /*

```

```

156 *      add_entry()
157 *
158 * adds a file entry to the specified directory, using the same
159 * semantics as find_entry(). It returns NULL if it failed.
160 *
161 * NOTE!! The inode part of 'de' is left at 0 - which means you
162 * may not sleep between calling this and putting something into
163 * the entry, as someone else might have used it while you slept.
164 */
/*
 *      add_entry()
 * 使用与 find_entry() 同样的方法，往指定目录中添加一文件目录项。
 * 如果失败则返回 NULL。
 *
 * 注意！！'de' (指定目录项结构指针) 的 i 节点部分被设置为 0 - 这表示
 * 在调用该函数和往目录项中添加信息之间不能睡眠，因为若睡眠那么其它
 * 人(进程)可能会已经使用了该目录项。
 */
///// 根据指定的目录和文件名添加目录项。
// 参数: dir - 指定目录的 i 节点; name - 文件名; namelen - 文件名长度;
// 返回: 高速缓冲区指针; res_dir - 返回的目录项结构指针;
165 static struct buffer_head * add_entry(struct m_inode * dir,
166      const char * name, int namelen, struct dir_entry ** res_dir)
167 {
168     int block, i;
169     struct buffer_head * bh;
170     struct dir_entry * de;
171
172     *res_dir = NULL;
173     // 如果定义了 NO_TRUNCATE，则若文件名长度超过最大长度 NAME_LEN，则返回。
174     #ifdef NO_TRUNCATE
175         if (namelen > NAME_LEN)
176             return NULL;
177     // 如果没有定义 NO_TRUNCATE，则若文件名长度超过最大长度 NAME_LEN，则截短之。
178     #else
179         if (namelen > NAME_LEN)
180             namelen = NAME_LEN;
181     #endif
182     // 如果文件名长度等于 0，则返回 NULL，退出。
183     if (!namelen)
184         return NULL;
185     // 如果该目录 i 节点所指向的第一个直接磁盘块号为 0，则返回 NULL 退出。
186     if (!(block = dir->i_zone[0]))
187         return NULL;
188     // 如果读取该磁盘块失败，则返回 NULL 并退出。
189     if (!(bh = bread(dir->i_dev, block)))
190         return NULL;
191     // 在目录项数据块中循环查找最后未使用的目录项。首先让目录项结构指针 de 指向高速缓冲的数据块
192     // 开始处，也即第一个目录项。
193     i = 0;
194     de = (struct dir_entry *) bh->b_data;
195     while (1) {
196         // 如果当前判别的目录项已经超出当前数据块，则释放该数据块，重新申请一块磁盘块 block。如果

```



```

// 申请失败, 则返回 NULL, 退出。
189         if ((char *)de >= BLOCK_SIZE+bh->b_data) {
190             brelse(bh);
191             bh = NULL;
192             block = create_block(dir, i/DIR_ENTRIES_PER_BLOCK);
193             if (!block)
194                 return NULL;
// 如果读取磁盘块返回的指针为空, 则跳过该块继续。
195             if (!(bh = bread(dir->i_dev, block))) {
196                 i += DIR_ENTRIES_PER_BLOCK;
197                 continue;
198             }
// 否则, 让目录项结构指针 de 指向该块的高速缓冲数据块开始处。
199             de = (struct dir_entry *) bh->b_data;
200         }
// 如果当前所操作的目录项序号 i*目录结构大小已经超过了该目录所指出的大小 i_size, 则说明该第 i
// 个目录项还未使用, 我们可以使用它。于是对该目录项进行设置(置该目录项的 i 节点指针为空)。并
// 更新该目录的长度值(加上一个目录项的长度, 设置目录的 i 节点已修改标志, 再更新该目录的改变
时
// 间为当前时间。
201         if (i*sizeof(struct dir_entry) >= dir->i_size) {
202             de->inode=0;
203             dir->i_size = (i+1)*sizeof(struct dir_entry);
204             dir->i_dirt = 1;
205             dir->i_ctime = CURRENT_TIME;
206         }
// 若该目录项的 i 节点为空, 则表示找到一个还未使用的目录项。于是更新目录的修改时间为当前时间。
// 并从用户数据区复制文件名到该目录项的文件名字段, 置相应的高速缓冲块已修改标志。返回该目录
// 项的指针以及该高速缓冲区的指针, 退出。
207         if (!de->inode) {
208             dir->i_mtime = CURRENT_TIME;
209             for (i=0; i < NAME_LEN ; i++)
210                 de->name[i]=(i<namelen)?get_fs_byte(name+i):0;
211             bh->b_dirt = 1;
212             *res_dir = de;
213             return bh;
214         }
// 如果该目录项已经被使用, 则继续检测下一个目录项。
215         de++;
216         i++;
217     }
// 执行不到这里。也许 Linus 在写这段代码时是先复制了上面 find_entry() 的代码, 而后修改的☺。
218     brelse(bh);
219     return NULL;
220 }
221
222 /*
223  *      get_dir()
224  *
225  * Getdir traverses the pathname until it hits the topmost directory.
226  * It returns NULL on failure.
227  */
228

```

```

*      get_dir()
* 该函数根据给出的路径名进行搜索，直到达到最顶端的目录。
* 如果失败则返回 NULL。
*/
///// 搜寻指定路径名的目录。
// 参数: pathname - 路径名。
// 返回: 目录的 i 节点指针。失败时返回 NULL。
228 static struct m\_inode * get\_dir(const char * pathname)
229 {
230     char c;
231     const char * thisname;
232     struct m\_inode * inode;
233     struct buffer\_head * bh;
234     int namelen, inr, idev;
235     struct dir\_entry * de;
236
// 如果进程没有设定根 i 节点，或者该进程根 i 节点的引用为 0，则系统出错，死机。
237     if (!current->root || !current->root->i_count)
238         panic("No root inode");
// 如果进程的当前工作目录指针为空，或者该当前目录 i 节点的引用计数为 0，也是系统有问题，死机。
239     if (!current->pwd || !current->pwd->i_count)
240         panic("No cwd inode");
// 如果用户指定的路径名的第 1 个字符是 '/'，则说明路径名是绝对路径名。则从根 i 节点开始操作。
241     if ((c=get\_fs\_byte(pathname))== '/') {
242         inode = current->root;
243         pathname++;
// 否则若第一个字符是其它字符，则表示给定的是相对路径名。应从进程的当前工作目录开始操作。
// 则取进程当前工作目录的 i 节点。
244     } else if (c)
245         inode = current->pwd;
// 则表示路径名为空，出错。返回 NULL，退出。
246     else
247         return NULL;    /* empty name is bad */ /* 空的路径名是错误的 */
// 将取得的 i 节点引用计数增 1。
248     inode->i_count++;
249     while (1) {
// 若该 i 节点不是目录节点，或者没有可进入的访问许可，则释放该 i 节点，返回 NULL，退出。
250         thisname = pathname;
251         if (!S\_ISDIR(inode->i_mode) || !permission(inode, MAY\_EXEC)) {
252             iput(inode);
253             return NULL;
254         }
// 从路径名开始起搜索检测字符，直到字符已是结尾符(NULL)或者是 '/'，此时 namelen 正好是当前处
理
// 目录名的长度。如果最后也是一个目录名，但其后没有加 '/'，则不会返回该最后目录的 i 节点！
// 比如: /var/log/httpd，将只返回 log/目录的 i 节点。
255         for(namelen=0; (c=get\_fs\_byte(pathname++))&&(c!='/'); namelen++)
256             /* nothing */;
// 若字符是结尾符 NULL，则表明已经到达指定目录，则返回该 i 节点指针，退出。
257         if (!c)
258             return inode;
// 调用查找指定目录和文件名的目录项函数，在当前处理目录中寻找子目录项。如果没有找到，则释放
// 该 i 节点，并返回 NULL，退出。

```

```

259         if (! (bh = find\_entry(&inode, thisname, namelen, &de))) {
260             iput(inode);
261             return NULL;
262         }
263     // 取该子目录项的 i 节点号 inr 和设备号 idev, 释放包含该目录项的高速缓冲块和该 i 节点。
264     inr = de->inode;
265     idev = inode->i_dev;
266     brelse(bh);
267     iput(inode);
268     // 取节点号 inr 的 i 节点信息, 若失败, 则返回 NULL, 退出。否则继续以该子目录的 i 节点进行操作。
269     if (! (inode = iget(idev, inr)))
270         return NULL;
271     }
272 }
273 /*
274 *      dir_namei()
275 *
276 * dir_namei() returns the inode of the directory of the
277 * specified name, and the name within that directory.
278 */
279 /*
280 *      dir_namei()
281 * dir_namei() 函数返回指定目录名的 i 节点指针, 以及在最顶层目录的名称。
282 */
283 // 参数: pathname - 目录路径名; namelen - 路径名长度。
284 // 返回: 指定目录名最顶层目录的 i 节点指针和最顶层目录名及其长度。
285 static struct m\_inode * dir\_namei(const char * pathname,
286     int * namelen, const char ** name)
287 {
288     char c;
289     const char * basename;
290     struct m\_inode * dir;
291     // 取指定路径名最顶层目录的 i 节点, 若出错则返回 NULL, 退出。
292     if (! (dir = get\_dir(pathname)))
293         return NULL;
294     // 对路径名 pathname 进行搜索检测, 查处最后一个 '/' 后面的名字字符串, 计算其长度, 并返回最顶
295     // 层目录的 i 节点指针。
296     basename = pathname;
297     while (c=get\_fs\_byte(pathname++))
298         if (c=='/')
299             basename=pathname;
300     *namelen = pathname-basename-1;
301     *name = basename;
302     return dir;
303 }
304 /*
305 *      namei()
306 *
307 * is used by most simple commands to get the inode of a specified name.
308 * Open, link etc use their own routines, but this is enough for things

```

```

301  * like 'chmod' etc.
302  */
/*
 *      namei()
 * 该函数被许多简单的命令用于取得指定路径名称的 i 节点。open、link 等则使用它们
 * 自己的相应函数，但对于象修改模式'chmod'等这样的命令，该函数已足够用了。
 */
//// 取指定路径名的 i 节点。
// 参数: pathname - 路径名。
// 返回: 对应的 i 节点。
303 struct m_inode * namei(const char * pathname)
304 {
305     const char * basename;
306     int inr, dev, namelen;
307     struct m_inode * dir;
308     struct buffer_head * bh;
309     struct dir_entry * de;
310
311     // 首先查找指定路径的最顶层目录的目录名及其 i 节点，若不存在，则返回 NULL，退出。
312     if (!(dir = dir_namei(pathname, &namelen, &basename)))
313         return NULL;
314     // 如果返回的最顶层名字的长度是 0，则表示该路径名以一个目录名为最后一项。
315     if (!namelen) /* special case: '/usr/' etc */
316         return dir; /* 对应于'/usr/'等情况 */
317     // 在返回的顶层目录中寻找指定文件名的目录项的 i 节点。因为如果最后也是一个目录名，但其后没
318     // 有加'/'，则不会返回该最后目录的 i 节点！比如: /var/log/httpd, 将只返回 log/目录的 i 节点。
319     // 因此 dir_namei() 将不以'/'结束的最后一个名字当作一个文件名来看待。因此这里需要单独对这种
320     // 情况使用寻找目录项 i 节点函数 find_entry() 进行处理。
321     bh = find_entry(&dir, basename, namelen, &de);
322     if (!bh) {
323         iput(dir);
324         return NULL;
325     }
326     // 取该目录项的 i 节点号和目录的设备号，并释放包含该目录项的高速缓冲区以及目录 i 节点。
327     inr = de->inode;
328     dev = dir->i_dev;
329     brelse(bh);
330     iput(dir);
331     // 取对应节号的 i 节点，修改其被访问时间为当前时间，并置已修改标志。最后返回该 i 节点指针。
332     dir = iget(dev, inr);
333     if (dir) {
334         dir->i_atime = CURRENT_TIME;
335         dir->i_dirt = 1;
336     }
337     return dir;
338 }
339
340 /*
341  *      open_namei()
342  *
343  * namei for open - this is in fact almost the whole open-routine.
344  */
345

```

```

*      open_namei()
* open()所使用的 namei 函数 - 这其实几乎是完整的打开文件程序。
*/
//// 文件打开 namei 函数。
// 参数: pathname - 文件路径名; flag - 文件打开标志; mode - 文件访问许可属性;
// 返回: 成功返回 0, 否则返回出错码; res_inode - 返回的对应文件路径名的 i 节点指针。
337 int open_namei(const char * pathname, int flag, int mode,
338                struct m\_inode ** res_inode)
339 {
340     const char * basename;
341     int inr, dev, namelen;
342     struct m\_inode * dir, *inode;
343     struct buffer\_head * bh;
344     struct dir\_entry * de;
345
    // 如果文件访问许可模式标志是只读(0), 但文件截 0 标志 O_TRUNC 却置位了, 则改为只写标志。
346     if ((flag & O\_TRUNC) && !(flag & O\_ACCMODE))
347         flag |= O\_WRONLY;
    // 使用进程的文件访问许可屏蔽码, 屏蔽掉给定模式中的相应位, 并添上普通文件标志。
348     mode &= 0777 & ~current->umask;
349     mode |= I\_REGULAR;
    // 根据路径名寻找到对应的 i 节点, 以及最顶端文件名及其长度。
350     if (!(dir = dir\_namei(pathname, &namelen, &basename)))
351         return -ENOENT;
    // 如果最顶端文件名长度为 0(例如 '/usr/' 这种路径名的情况), 那么若打开操作不是创建、截 0,
    // 则表示打开一个目录名, 直接返回该目录的 i 节点, 并退出。
352     if (!namelen) {
353         /* special case: '/usr/' etc */
354         if (!(flag & (O\_ACCMODE | O\_CREAT | O\_TRUNC))) {
355             *res_inode = dir;
356             return 0;
357         }
    // 否则释放该 i 节点, 返回出错码。
358     }
359     iput(dir);
360     return -EISDIR;
    // 在 dir 节点对应的目录中取文件名对应的目录项结构 de 和该目录项所在的高速缓冲区。
361     bh = find\_entry(&dir, basename, namelen, &de);
    // 如果该高速缓冲指针为 NULL, 则表示没有找到对应文件名的目录项, 因此只可能是创建文件操作。
362     if (!bh) {
363         // 如果不是创建文件, 则释放该目录的 i 节点, 返回出错号退出。
364         if (!(flag & O\_CREAT)) {
365             iput(dir);
366             return -ENOENT;
367         }
    // 如果用户在该目录没有写的权力, 则释放该目录的 i 节点, 返回出错号退出。
368     }
369     if (!permission(dir, MAY\_WRITE)) {
370         iput(dir);
371         return -EACCES;
372     }
    // 在目录节点对应的设备上申请一个新 i 节点, 若失败, 则释放目录的 i 节点, 并返回没有空间出错码。
373     inode = new\_inode(dir->i_dev);
374     if (!inode) {
375         iput(dir);

```

```

373         return -ENOSPC;
374     }
    // 否则使用该新 i 节点, 对其进行初始设置: 置节点的用户 id; 对应节点访问模式; 置已修改标志。
375     inode->i_uid = current->euid;
376     inode->i_mode = mode;
377     inode->i_dirt = 1;
    // 然后在指定目录 dir 中添加一新目录项。
378     bh = add_entry(dir, basename, namelen, &de);
    // 如果返回的应该含有新目录项的高速缓冲区指针为 NULL, 则表示添加目录项操作失败。于是将该
    // 新 i 节点的引用连接计数减 1; 并释放该 i 节点与目录的 i 节点, 返回出错码, 退出。
379     if (!bh) {
380         inode->i_nlinks--;
381         iput(inode);
382         iput(dir);
383         return -ENOSPC;
384     }
    // 初始设置该新目录项: 置 i 节点号为新申请到的 i 节点的号码; 并置高速缓冲区已修改标志。然后
    // 释放该高速缓冲区, 释放目录的 i 节点。返回新目录项的 i 节点指针, 退出。
385     de->inode = inode->i_num;
386     bh->b_dirt = 1;
387     brelse(bh);
388     iput(dir);
389     *res_inode = inode;
390     return 0;
391 }
    // 若上面在目录中取文件名对应的目录项结构操作成功(也即 bh 不为 NULL), 取出该目录项的 i 节点号
    // 和其所在的设备号, 并释放该高速缓冲区以及目录的 i 节点。
392     inr = de->inode;
393     dev = dir->i_dev;
394     brelse(bh);
395     iput(dir);
    // 如果独占使用标志 O_EXCL 置位, 则返回文件已存在出错码, 退出。
396     if (flag & O_EXCL)
397         return -EEXIST;
    // 如果取该目录项对应 i 节点的操作失败, 则返回访问出错码, 退出。
398     if (!(inode = iget(dev, inr)))
399         return -EACCES;
    // 若该 i 节点是一个目录的节点并且访问模式是只读或只写, 或者没有访问的许可权限, 则释放该 i
    // 节点, 返回访问权限出错码, 退出。
400     if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
401         !permission(inode, ACC_MODE(flag))) {
402         iput(inode);
403         return -EPERM;
404     }
    // 更新该 i 节点的访问时间字段为当前时间。
405     inode->i_atime = CURRENT_TIME;
    // 如果设立了截 0 标志, 则将该 i 节点的文件长度截为 0。
406     if (flag & O_TRUNC)
407         truncate(inode);
    // 最后返回该目录项 i 节点的指针, 并返回 0 (成功)。
408     *res_inode = inode;
409     return 0;
410 }

```

```

411  /// 系统调用函数 - 创建一个特殊文件或普通文件节点(node)。
412  // 创建名称为 filename, 由 mode 和 dev 指定的文件系统节点(普通文件、设备特殊文件或命名管道)。
413  // 参数: filename - 路径名; mode - 指定使用许可以及所创建节点的类型; dev - 设备号。
414  // 返回: 成功则返回 0, 否则返回出错码。
415  int sys_mknod(const char * filename, int mode, int dev)
416  {
417      const char * basename;
418      int namelen;
419      struct m_inode * dir, * inode;
420      struct buffer_head * bh;
421      struct dir_entry * de;
422
423      // 如果不是超级用户, 则返回访问许可出错码。
424      if (!suser())
425          return -EPERM;
426      // 如果找不到对应路径名目录的 i 节点, 则返回出错码。
427      if (!(dir = dir_namei(filename, &namelen, &basename)))
428          return -ENOENT;
429      // 如果最顶端的文件名长度为 0, 则说明给出的路径名最后没有指定文件名, 释放该目录 i 节点, 返回
430      // 出错码, 退出。
431      if (!namelen) {
432          iput(dir);
433          return -ENOENT;
434      }
435      // 如果在该目录中没有写的权限, 则释放该目录的 i 节点, 返回访问许可出错码, 退出。
436      if (!permission(dir, MAY_WRITE)) {
437          iput(dir);
438          return -EPERM;
439      }
440      // 如果对应路径名上最后的文件名的目录项已经存在, 则释放包含该目录项的高速缓冲区, 释放目录
441      // 的 i 节点, 返回文件已经存在出错码, 退出。
442      bh = find_entry(&dir, basename, namelen, &de);
443      if (bh) {
444          brelse(bh);
445          iput(dir);
446          return -EEXIST;
447      }
448      // 申请一个新的 i 节点, 如果不成功, 则释放目录的 i 节点, 返回无空间出错码, 退出。
449      inode = new_inode(dir->i_dev);
450      if (!inode) {
451          iput(dir);
452          return -ENOSPC;
453      }
454      // 设置该 i 节点的属性模式。如果要创建的是块设备文件或者是字符设备文件, 则令 i 节点的直接块
455      // 指针 0 等于设备号。
456      inode->i_mode = mode;
457      if (S_ISBLK(mode) || S_ISCHR(mode))
458          inode->i_zone[0] = dev;
459      // 设置该 i 节点的修改时间、访问时间为当前时间。
460      inode->i_mtime = inode->i_atime = CURRENT_TIME;
461      inode->i_dirt = 1;
462      // 在目录中新添加一个目录项, 如果失败(包含该目录项的高速缓冲区指针为 NULL), 则释放目录的

```

```

// i 节点; 所申请的 i 节点引用连接计数复位, 并释放该 i 节点。返回出错码, 退出。
448     bh = add\_entry(dir, basename, namelen, &de);
449     if (!bh) {
450         iput(dir);
451         inode->i_nlinks=0;
452         iput(inode);
453         return -ENOSPC;
454     }
// 令该目录项的 i 节点字段等于新 i 节点号, 置高速缓冲区已修改标志, 释放目录和新的 i 节点, 释放
// 高速缓冲区, 最后返回 0(成功)。
455     de->inode = inode->i_num;
456     bh->b_dirt = 1;
457     iput(dir);
458     iput(inode);
459     brelse(bh);
460     return 0;
461 }
462
//// 系统调用函数 - 创建目录。
// 参数: pathname - 路径名; mode - 目录使用的权限属性。
// 返回: 成功则返回 0, 否则返回出错码。
463 int sys\_mkdir(const char * pathname, int mode)
464 {
465     const char * basename;
466     int namelen;
467     struct m\_inode * dir, * inode;
468     struct buffer\_head * bh, *dir_block;
469     struct dir\_entry * de;
470
// 如果不是超级用户, 则返回访问许可出错码。
471     if (!suser())
472         return -EPERM;
// 如果找不到对应路径名目录的 i 节点, 则返回出错码。
473     if (!(dir = dir\_namei(pathname, &namelen, &basename)))
474         return -ENOENT;
// 如果最顶端的文件名长度为 0, 则说明给出的路径名最后没有指定文件名, 释放该目录 i 节点, 返回
// 出错码, 退出。
475     if (!namelen) {
476         iput(dir);
477         return -ENOENT;
478     }
// 如果在该目录中没有写的权限, 则释放该目录的 i 节点, 返回访问许可出错码, 退出。
479     if (!permission(dir, MAY\_WRITE)) {
480         iput(dir);
481         return -EPERM;
482     }
// 如果对应路径名上最后的文件名的目录项已经存在, 则释放包含该目录项的高速缓冲区, 释放目录
// 的 i 节点, 返回文件已经存在出错码, 退出。
483     bh = find\_entry(&dir, basename, namelen, &de);
484     if (bh) {
485         brelse(bh);
486         iput(dir);
487         return -EEXIST;

```



```

488     }
// 申请一个新的 i 节点，如果不成功，则释放目录的 i 节点，返回无空间出错码，退出。
489     inode = new_inode(dir->i_dev);
490     if (!inode) {
491         iput(dir);
492         return -ENOSPC;
493     }
// 置该新 i 节点对应的文件长度为 32(一个目录项的大小)，置节点已修改标志，以及节点的修改时间
// 和访问时间。
494     inode->i_size = 32;
495     inode->i_dirt = 1;
496     inode->i_mtime = inode->i_atime = CURRENT_TIME;
// 为该 i 节点申请一磁盘块，并令节点第一个直接块指针等于该块号。如果申请失败，则释放对应目录
// 的 i 节点；复位新申请的 i 节点连接计数；释放该新的 i 节点，返回没有空间出错码，退出。
497     if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
498         iput(dir);
499         inode->i_nlinks--;
500         iput(inode);
501         return -ENOSPC;
502     }
// 置该新的 i 节点已修改标志。
503     inode->i_dirt = 1;
// 读新申请的磁盘块。若出错，则释放对应目录的 i 节点；释放申请的磁盘块；复位新申请的 i 节点
// 连接计数；释放该新的 i 节点，返回没有空间出错码，退出。
504     if (!(dir_block=bread(inode->i_dev, inode->i_zone[0]))) {
505         iput(dir);
506         free_block(inode->i_dev, inode->i_zone[0]);
507         inode->i_nlinks--;
508         iput(inode);
509         return -ERROR;
510     }
// 令 de 指向目录项数据块，置该目录项的 i 节点号字段等于新申请的 i 节点号，名字字段等于“.”。
511     de = (struct dir_entry *) dir_block->b_data;
512     de->inode=inode->i_num;
513     strcpy(de->name, ".");
// 然后 de 指向下一个目录项结构，该结构用于存放上级目录的节点号和名字“..”。
514     de++;
515     de->inode = dir->i_num;
516     strcpy(de->name, "..");
517     inode->i_nlinks = 2;
// 然后设置该高速缓冲区已修改标志，并释放该缓冲区。
518     dir_block->b_dirt = 1;
519     brelse(dir_block);
// 初始化设置新 i 节点的模式字段，并置该 i 节点已修改标志。
520     inode->i_mode = I_DIRECTORY | (mode & 0777 & ~current->umask);
521     inode->i_dirt = 1;
// 在目录中新添加一个目录项，如果失败(包含该目录项的高速缓冲区指针为 NULL)，则释放目录的
// i 节点；所申请的 i 节点引用连接计数复位，并释放该 i 节点。返回出错码，退出。
522     bh = add_entry(dir, basename, namelen, &de);
523     if (!bh) {
524         iput(dir);
525         free_block(inode->i_dev, inode->i_zone[0]);
526         inode->i_nlinks=0;

```

```

527         iput(inode);
528         return -ENOSPC;
529     }
    // 令该目录项的 i 节点字段等于新 i 节点号, 置高速缓冲区已修改标志, 释放目录和新的 i 节点, 释放
    // 高速缓冲区, 最后返回 0(成功)。
530     de->inode = inode->i_num;
531     bh->b_dirt = 1;
532     dir->i_nlinks++;
533     dir->i_dirt = 1;
534     iput(dir);
535     iput(inode);
536     brelse(bh);
537     return 0;
538 }
539
540 /*
541  * routine to check that the specified directory is empty (for rmdir)
542  */
    /*
    * 用于检查指定的目录是否为空的子程序(用于 rmdir 系统调用函数)。
    */
    // 检查指定目录是否是空的。
    // 参数: inode - 指定目录的 i 节点指针。
    // 返回: 0 - 是空的; 1 - 不空。
543 static int empty_dir(struct m_inode * inode)
544 {
545     int nr, block;
546     int len;
547     struct buffer_head * bh;
548     struct dir_entry * de;
549
    // 计算指定目录中现有目录项的个数(应该起码有 2 个, 即"."和".."两个文件目录项)。
550     len = inode->i_size / sizeof (struct dir_entry);
    // 如果目录项个数少于 2 个或者该目录 i 节点的第 1 个直接块没有指向任何磁盘块号, 或者相应磁盘
    // 块读不出, 则显示警告信息“设备 dev 上目录错”, 返回 0(失败)。
551     if (len < 2 || !inode->i_zone[0] ||
552         !(bh = bread(inode->i_dev, inode->i_zone[0]))) {
553         printk("warning - bad directory on dev %04x\n", inode->i_dev);
554         return 0;
555     }
    // 让 de 指向含有读出磁盘块数据的高速缓冲区中第 1 项目录项。
556     de = (struct dir_entry *) bh->b_data;
    // 如果第 1 个目录项的 i 节点号字段值不等于该目录的 i 节点号, 或者第 2 个目录项的 i 节点号字段
    // 为零, 或者两个目录项的名字字段不分别等于"."和"..", 则显示出错警告信息“设备 dev 上目录错”
    // 并返回 0。
557     if (de[0].inode != inode->i_num || !de[1].inode ||
558         strcmp(".", de[0].name) || strcmp("..", de[1].name)) {
559         printk("warning - bad directory on dev %04x\n", inode->i_dev);
560         return 0;
561     }
    // 令 nr 等于目录项序号; de 指向第三个目录项。
562     nr = 2;
563     de += 2;

```

```

// 循环检测该目录中所有的目录项(len-2个), 看有没有目录项的 i 节点号字段不为 0(被使用)。
564 while (nr<len) {
// 如果该块磁盘块中的目录项已经检测完, 则释放该磁盘块的高速缓冲区, 读取下一块含有目录项的
// 磁盘块。若相应块没有使用(或已经不用, 如文件已经删除等), 则继续读下一块, 若读不出, 则出
// 错, 返回 0。否则让 de 指向读出块的首个目录项。
565     if ((void *) de >= (void *) (bh->b_data+BLOCK_SIZE)) {
566         brelse(bh);
567         block=bmap(inode,nr/DIR_ENTRIES_PER_BLOCK);
568         if (!block) {
569             nr += DIR_ENTRIES_PER_BLOCK;
570             continue;
571         }
572         if (!(bh=bread(inode->i_dev,block)))
573             return 0;
574         de = (struct dir_entry *) bh->b_data;
575     }
// 如果该目录项的 i 节点号字段不等于 0, 则表示该目录项目前正被使用, 则释放该高速缓冲区,
// 返回 0, 退出。
576     if (de->inode) {
577         brelse(bh);
578         return 0;
579     }
// 否则, 若还没有查询完该目录中的所有目录项, 则继续检测。
580     de++;
581     nr++;
582 }
// 到这里说明该目录中没有找到已用的目录项(当然除了头两个以外), 则返回缓冲区, 返回 1。
583     brelse(bh);
584     return 1;
585 }
586
//// 系统调用函数 - 删除指定名称的目录。
// 参数: name - 目录名(路径名)。
// 返回: 返回 0 表示成功, 否则返回出错号。
587 int sys_rmdir(const char * name)
588 {
589     const char * basename;
590     int namelen;
591     struct m_inode * dir, * inode;
592     struct buffer_head * bh;
593     struct dir_entry * de;
594
// 如果不是超级用户, 则返回访问许可出错码。
595     if (!suser())
596         return -EPERM;
// 如果找不到对应路径名目录的 i 节点, 则返回出错码。
597     if (!(dir = dir_namei(name,&namelen,&basename)))
598         return -ENOENT;
// 如果最顶端的文件名长度为 0, 则说明给出的路径名最后没有指定文件名, 释放该目录 i 节点, 返回
// 出错码, 退出。
599     if (!namelen) {
600         iput(dir);
601         return -ENOENT;

```

```

602     }
    // 如果在该目录中没有写的权限，则释放该目录的 i 节点，返回访问许可出错码，退出。
603     if (!permission(dir, MAY\_WRITE)) {
604         iput(dir);
605         return -EPERM;
606     }
    // 如果对应路径名上最后的文件名的目录项不存在，则释放包含该目录项的高速缓冲区，释放目录
    // 的 i 节点，返回文件已经存在出错码，退出。否则 dir 是包含要被删除目录名的目录 i 节点，de
    // 是要被删除目录的目录项结构。
607     bh = find\_entry(&dir, basename, namelen, &de);
608     if (!bh) {
609         iput(dir);
610         return -ENOENT;
611     }
    // 取该目录项指明的 i 节点。若出错则释放目录的 i 节点，并释放含有目录项的高速缓冲区，返回
    // 出错号。
612     if (!(inode = iget(dir->i_dev, de->inode))) {
613         iput(dir);
614         brelse(bh);
615         return -EPERM;
616     }
    // 若该目录设置了受限删除标志并且进程的有效用户 id 不等于该 i 节点的用户 id，则表示没有权限删
    // 除该目录，于是释放包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲区，返
    // 回出错码。
617     if ((dir->i_mode & S\_ISVTX) && current->euid &&
618         inode->i_uid != current->euid) {
619         iput(dir);
620         iput(inode);
621         brelse(bh);
622         return -EPERM;
623     }
    // 如果要被删除的目录项的 i 节点的设备号不等于包含该目录项的目录的设备号，或者该被删除目录的
    // 引用连接计数大于 1(表示有符号连接等)，则不能删除该目录，于是释放包含要删除目录名的目录
    // i 节点和该要删除目录的 i 节点，释放高速缓冲区，返回出错码。
624     if (inode->i_dev != dir->i_dev || inode->i_count > 1) {
625         iput(dir);
626         iput(inode);
627         brelse(bh);
628         return -EPERM;
629     }
    // 如果要被删除目录的目录项 i 节点的节点号等于包含该需删除目录的 i 节点号，则表示试图删除“.”
    // 目录。于是释放包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲区，返回
    // 出错码。
630     if (inode == dir) {      /* we may not delete ".", but "../dir" is ok */
631         iput(inode);        /* 我们不可以删除“.”，但可以删除“../dir” */
632         iput(dir);
633         brelse(bh);
634         return -EPERM;
635     }
    // 若要被删除的目录的 i 节点的属性表明这不是一个目录，则释放包含要删除目录名的目录 i 节点和
    // 该要删除目录的 i 节点，释放高速缓冲区，返回出错码。
636     if (!S\_ISDIR(inode->i_mode)) {
637         iput(inode);

```

```

638         iput(dir);
639         brelse(bh);
640         return -ENOTDIR;
641     }
    // 若该需被删除的目录不空，则释放包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，释放
    // 高速缓冲区，返回出错码。
642     if (!empty_dir(inode)) {
643         iput(inode);
644         iput(dir);
645         brelse(bh);
646         return -ENOTEMPTY;
647     }
    // 若该需被删除目录的 i 节点的连接数不等于 2，则显示警告信息。
648     if (inode->i_nlinks != 2)
649         printk("empty directory has nlink!=2 (%d)", inode->i_nlinks);
    // 置该需被删除目录的目录项的 i 节点号字段为 0，表示该目录项不再使用，并置含有该目录项的高速
    // 缓冲区已修改标志，并释放该缓冲区。
650     de->inode = 0;
651     bh->b_dirt = 1;
652     brelse(bh);
    // 置被删除目录的 i 节点的连接数为 0，并置 i 节点已修改标志。
653     inode->i_nlinks=0;
654     inode->i_dirt=1;
    // 将包含被删除目录名的目录的 i 节点引用计数减 1，修改其改变时间和修改时间为当前时间，并置
    // 该节点已修改标志。
655     dir->i_nlinks--;
656     dir->i_ctime = dir->i_mtime = CURRENT_TIME;
657     dir->i_dirt=1;
    // 最后释放包含要删除目录名的目录 i 节点和该要删除目录的 i 节点，返回 0(成功)。
658     iput(dir);
659     iput(inode);
660     return 0;
661 }
662
    ///// 系统调用函数 - 删除文件名以及可能也删除其相关的文件。
    // 从文件系统删除一个名字。如果是一个文件的最后一个连接，并且没有进程正打开该文件，则该文件
    // 也将被删除，并释放所占用的设备空间。
    // 参数: name - 文件名。
    // 返回: 成功则返回 0，否则返回出错号。
663 int sys_unlink(const char * name)
664 {
665     const char * basename;
666     int namelen;
667     struct m_inode * dir, * inode;
668     struct buffer_head * bh;
669     struct dir_entry * de;
670
    // 如果找不到对应路径名目录的 i 节点，则返回出错码。
671     if (!(dir = dir_namei(name, &namelen, &basename)))
672         return -ENOENT;
    // 如果最顶端的文件名长度为 0，则说明给出的路径名最后没有指定文件名，释放该目录 i 节点，返回
    // 出错码，退出。
673     if (!namelen) {

```

```

674         iput(dir);
675         return -ENOENT;
676     }
    // 如果在该目录中没有写的权限, 则释放该目录的 i 节点, 返回访问许可出错码, 退出。
677     if (!permission(dir, MAY_WRITE)) {
678         iput(dir);
679         return -EPERM;
680     }
    // 如果对应路径名上最后的文件名的目录项不存在, 则释放包含该目录项的高速缓冲区, 释放目录
    // 的 i 节点, 返回文件已经存在出错码, 退出。否则 dir 是包含要被删除目录名的目录 i 节点, de
    // 是要被删除目录的目录项结构。
681     bh = find_entry(&dir, basename, namelen, &de);
682     if (!bh) {
683         iput(dir);
684         return -ENOENT;
685     }
    // 取该目录项指明的 i 节点。若出错则释放目录的 i 节点, 并释放含有目录项的高速缓冲区, 返回
    // 出错号。
686     if (!(inode = iget(dir->i_dev, de->inode))) {
687         iput(dir);
688         brelse(bh);
689         return -ENOENT;
690     }
    // 如果该目录设置了受限删除标志并且用户不是超级用户, 并且进程的有效用户 id 不等于被删除文件
    // 名 i 节点的用户 id, 并且进程的有效用户 id 也不等于目录 i 节点的用户 id, 则没有权限删除该文件
    // 名。则释放该目录 i 节点和该文件名目录项的 i 节点, 释放包含该目录项的缓冲区, 返回出错号。
691     if ((dir->i_mode & S_ISVTX) && !suser() &&
692         current->euid != inode->i_uid &&
693         current->euid != dir->i_uid) {
694         iput(dir);
695         iput(inode);
696         brelse(bh);
697         return -EPERM;
698     }
    // 如果该指定文件名是一个目录, 则也不能删除, 释放该目录 i 节点和该文件名目录项的 i 节点, 释放
    // 包含该目录项的缓冲区, 返回出错号。
699     if (S_ISDIR(inode->i_mode)) {
700         iput(inode);
701         iput(dir);
702         brelse(bh);
703         return -EPERM;
704     }
    // 如果该 i 节点的连接数已经为 0, 则显示警告信息, 修正其为 1。
705     if (!inode->i_nlinks) {
706         printk("Deleting nonexistent file (%04x:%d), %d\n",
707             inode->i_dev, inode->i_num, inode->i_nlinks);
708         inode->i_nlinks=1;
709     }
    // 将该文件名的目录项中的 i 节点号字段置为 0, 表示释放该目录项, 并设置包含该目录项的缓冲区
    // 已修改标志, 释放该高速缓冲区。
710     de->inode = 0;
711     bh->b_dirt = 1;
712     brelse(bh);

```

```

// 该 i 节点的连接数减 1, 置已修改标志, 更新改变时间为当前时间。最后释放该 i 节点和目录的 i 节
// 点, 返回 0(成功)。
713     inode->i_nlinks--;
714     inode->i_dirt = 1;
715     inode->i_ctime = CURRENT_TIME;
716     iput(inode);
717     iput(dir);
718     return 0;
719 }
720
///// 系统调用函数 - 为文件建立一个文件名。
// 为一个已经存在的文件创建一个新连接(也称为硬连接 - hard link)。
// 参数: oldname - 原路径名; newname - 新的路径名。
// 返回: 若成功则返回 0, 否则返回出错号。
721 int sys_link(const char * oldname, const char * newname)
722 {
723     struct dir_entry * de;
724     struct m_inode * oldinode, * dir;
725     struct buffer_head * bh;
726     const char * basename;
727     int namelen;
728
// 取原文件路径名对应的 i 节点 oldinode。如果为 0, 则表示出错, 返回出错号。
729     oldinode=namei(oldname);
730     if (!oldinode)
731         return -ENOENT;
// 如果原路径名对应的是一个目录名, 则释放该 i 节点, 返回出错号。
732     if (S_ISDIR(oldinode->i_mode)) {
733         iput(oldinode);
734         return -EPERM;
735     }
// 查找新路径名的最顶层目录的 i 节点, 并返回最后的文件名及其长度。如果目录的 i 节点没有找到,
// 则释放原路径名的 i 节点, 返回出错号。
736     dir = dir_namei(newname, &namelen, &basename);
737     if (!dir) {
738         iput(oldinode);
739         return -EACCES;
740     }
// 如果新路径名中不包括文件名, 则释放原路径名 i 节点和新路径名目录的 i 节点, 返回出错号。
741     if (!namelen) {
742         iput(oldinode);
743         iput(dir);
744         return -EPERM;
745     }
// 如果新路径名目录的设备号与原路径名的设备号不一样, 则也不能建立连接, 于是释放新路径名
// 目录的 i 节点和原路径名的 i 节点, 返回出错号。
746     if (dir->i_dev != oldinode->i_dev) {
747         iput(dir);
748         iput(oldinode);
749         return -EXDEV;
750     }
// 如果用户没有在新目录中写的权限, 则也不能建立连接, 于是释放新路径名目录的 i 节点和原路径名
// 的 i 节点, 返回出错号。

```

```

751         if (!permission(dir, MAY_WRITE)) {
752             iput(dir);
753             iput(oldinode);
754             return -EACCES;
755         }
756         // 查询该新路径名是否已经存在, 如果存在, 则也不能建立连接, 于是释放包含该已存在目录项的高速
757         // 缓冲区, 释放新路径名目录的 i 节点和原路径名的 i 节点, 返回出错号。
758         bh = find_entry(&dir, basename, namelen, &de);
759         if (bh) {
760             brelse(bh);
761             iput(dir);
762             iput(oldinode);
763             return -EEXIST;
764         }
765         // 在新目录中添加一个目录项。若失败则释放该目录的 i 节点和原路径名的 i 节点, 返回出错号。
766         bh = add_entry(dir, basename, namelen, &de);
767         if (!bh) {
768             iput(dir);
769             iput(oldinode);
770             return -ENOSPC;
771         }
772         // 否则初始设置该目录项的 i 节点号等于原路径名的 i 节点号, 并置包含该新添目录项的高速缓冲区
773         // 已修改标志, 释放该缓冲区, 释放目录的 i 节点。
774         de->inode = oldinode->i_num;
775         bh->b_dirt = 1;
776         brelse(bh);
777         iput(dir);
778         // 将原节点的应用计数加 1, 修改其改变时间为当前时间, 并设置 i 节点已修改标志, 最后释放原
779         // 路径名的 i 节点, 并返回 0(成功)。
780         oldinode->i_nlinks++;
781         oldinode->i_ctime = CURRENT_TIME;
782         oldinode->i_dirt = 1;
783         iput(oldinode);
784         return 0;
785     }
786 }
787

```

## 9.18 char\_dev.c 文件

### 9.18.1 功能描述

### 9.18.2 代码注释

列表 linux/fs/char\_dev.c 程序

```

1  /*
2  *  linux/fs/char_dev.c
3  *
4  *  (C) 1991  Linus Torvalds

```



```

5  */
6
7 #include <errno.h>          // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
8 #include <sys/types.h>      // 类型头文件。定义了基本的系统数据类型。
9
10 #include <linux/sched.h>    // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
    // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
11 #include <linux/kernel.h>   // 内核头文件。含有一些内核常用函数的原形定义。
12
13 #include <asm/segment.h>    // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
14 #include <asm/io.h>        // io 头文件。定义硬件端口输入/输出宏汇编语句。
15
16 extern int tty_read(unsigned minor, char * buf, int count);    // 终端读。
17 extern int tty_write(unsigned minor, char * buf, int count);    // 终端写。
18
    // 定义字符设备读写函数指针类型。
19 typedef (*crw_ptr)(int rw, unsigned minor, char * buf, int count, off_t * pos);
20
    // 终端读写操作函数。
    // 参数: rw - 读写命令; minor - 终端子设备号; buf - 缓冲区; cout - 读写字节数;
    // pos - 读写操作当前指针，对于终端操作，该指针无用。
    // 返回: 实际读写的字节数。
21 static int rw_ttyx(int rw, unsigned minor, char * buf, int count, off_t * pos)
22 {
23     return ((rw==READ)?tty_read(minor, buf, count):
24             tty_write(minor, buf, count));
25 }
26
    // 终端读写操作函数。
    // 同上 rw_ttyx(), 只是增加了对进程是否有控制终端的检测。
27 static int rw_tty(int rw, unsigned minor, char * buf, int count, off_t * pos)
28 {
    // 若进程没有对应的控制终端，则返回出错号。
29     if (current->tty<0)
30         return -EPERM;
    // 否则调用终端读写函数 rw_ttyx(), 并返回实际读写字节数。
31     return rw_ttyx(rw, current->tty, buf, count, pos);
32 }
33
    // 内存数据读写。未实现。
34 static int rw_ram(int rw, char * buf, int count, off_t * pos)
35 {
36     return -EIO;
37 }
38
    // 内存数据读写操作函数。未实现。
39 static int rw_mem(int rw, char * buf, int count, off_t * pos)
40 {
41     return -EIO;
42 }
43
    // 内核数据区读写函数。未实现。
44 static int rw_kmem(int rw, char * buf, int count, off_t * pos)

```

```

45 {
46     return -EIO;
47 }
48
// 端口读写操作函数。
// 参数: rw - 读写命令; buf - 缓冲区; count - 读写字节数; pos - 端口地址。
// 返回: 实际读写的字节数。
49 static int rw_port(int rw, char * buf, int count, off_t * pos)
50 {
51     int i=*pos;
52
// 对于所要求读写的字节数, 并且端口地址小于 64k 时, 循环执行单个字节的读写操作。
53     while (count-->0 && i<65536) {
// 若是读命令, 则从端口 i 中读取一字节内容并放到用户缓冲区中。
54         if (rw==READ)
55             put_fs_byte(inb(i), buf++);
// 若是写命令, 则从用户数据缓冲区中取一字节输出到端口 i。
56         else
57             outb(get_fs_byte(buf++), i);
// 前移一个端口。[??]
58         i++;
59     }
// 计算读/写的字节数, 并相应调整读写指针。
60     i -= *pos;
61     *pos += i;
// 返回读/写的字节数。
62     return i;
63 }
64
///// 内存读写操作函数。
65 static int rw_memory(int rw, unsigned minor, char * buf, int count, off_t * pos)
66 {
// 根据内存设备子设备号, 分别调用不同的内存读写函数。
67     switch(minor) {
68         case 0:
69             return rw_ram(rw, buf, count, pos);
70         case 1:
71             return rw_mem(rw, buf, count, pos);
72         case 2:
73             return rw_kmem(rw, buf, count, pos);
74         case 3:
75             return (rw==READ)?0:count;      /* rw_null */
76         case 4:
77             return rw_port(rw, buf, count, pos);
78         default:
79             return -EIO;
80     }
81 }
82
// 定义系统中设备种数。
83 #define NRDEVS ((sizeof (crw_table))/(sizeof (crw_ptr)))
84
// 字符设备读写函数指针表。

```

```

85 static crw_ptr crw_table[]={
86     NULL,                /* nodev */                /* 无设备(空设备) */
87     rw_memory,          /* /dev/mem etc */        /* /dev/mem 等 */
88     NULL,                /* /dev/fd */            /* /dev/fd 软驱 */
89     NULL,                /* /dev/hd */            /* /dev/hd 硬盘 */
90     rw_ttyx,             /* /dev/ttyx */          /* /dev/ttyx 串口终端 */
91     rw_tty,             /* /dev/tty */           /* /dev/tty 终端 */
92     NULL,                /* /dev/lp */            /* /dev/lp 打印机 */
93     NULL};              /* unnamed pipes */     /* 未命名管道 */
94
95     // 字符设备读写操作函数。
96     // 参数: rw - 读写命令; dev - 设备号; buf - 缓冲区; count - 读写字节数; pos - 读写指针。
97     // 返回: 实际读/写字节数。
98
99     int rw_char(int rw,int dev, char * buf, int count, off_t * pos)
100 {
101     crw_ptr call_addr;
102
103     // 如果设备号超出系统设备数, 则返回出错码。
104     if (MAJOR(dev)>=NRDEVS)
105         return -ENODEV;
106     // 若该设备没有对应的读/写函数, 则返回出错码。
107     if (!(call_addr=crw_table[MAJOR(dev)]))
108         return -ENODEV;
109     // 调用对应设备的读写操作函数, 并返回实际读/写的字节数。
110     return call_addr(rw,MINOR(dev),buf,count,pos);
111 }
112
113
114
115

```

## 9.19 ioctl.c 文件

### 9.19.1 功能描述

### 9.19.2 代码注释

列表 linux/fs/ioctl.c 程序

```

1  /*
2  *  linux/fs/ioctl.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  #include <string.h>        // 字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
8  #include <errno.h>         // 错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
9  #include <sys/stat.h>      // 文件状态头文件。含有文件或文件系统状态结构 stat {} 和常量。
10
11 #include <linux/sched.h>    // 调度程序头文件, 定义了任务结构 task_struct、初始任务 0 的数据,
12                             // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

```

13 extern int tty\_ioctl(int dev, int cmd, int arg); // 终端 ioctl(chr_drv/tty_ioctl.c, 115)。
14 // 定义输入输出控制(ioctl)函数指针。
15 typedef int (\*ioctl\_ptr)(int dev, int cmd, int arg);
16 // 定义系统中设备种数。
17 #define NRDEVS ((sizeof (ioctl\_table))/(sizeof (ioctl\_ptr)))
18 // ioctl 操作函数指针表。
19 static ioctl\_ptr ioctl\_table[]={
20     NULL,          /* nodev */
21     NULL,          /* /dev/mem */
22     NULL,          /* /dev/fd */
23     NULL,          /* /dev/hd */
24     tty\_ioctl,     /* /dev/ttyx */
25     tty\_ioctl,     /* /dev/tty */
26     NULL,          /* /dev/lp */
27     NULL};         /* named pipes */
28
29 // 系统调用函数 - 输入输出控制函数。
30 // 参数: fd - 文件描述符; cmd - 命令码; arg - 参数。
31 // 返回: 成功则返回 0, 否则返回出错码。
32 int sys\_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
33 {
34     struct file * filp;
35     int dev, mode;
36
37     // 如果文件描述符超出可打开的文件数, 或者对应描述符的文件结构指针为空, 则返回出错码, 退出。
38     if (fd >= NR\_OPEN || !(filp = current->filp[fd]))
39         return -EBADF;
40     // 取对应文件的属性。如果该文件不是字符文件, 也不是块设备文件, 则返回出错码, 退出。
41     mode = filp->f_inode->i_mode;
42     if (!S\_ISCHR(mode) && !S\_ISBLK(mode))
43         return -EINVAL;
44     // 从字符或块设备文件的 i 节点中取设备号。如果设备号大于系统现有的设备数, 则返回出错号。
45     dev = filp->f_inode->i_zone[0];
46     if (MAJOR(dev) >= NRDEVS)
47         return -ENODEV;
48     // 如果该设备在 ioctl 函数指针表中没有对应函数, 则返回出错码。
49     if (!ioctl\_table[MAJOR(dev)])
50         return -ENOTTY;
51     // 否则返回实际 ioctl 函数返回码, 成功则返回 0, 否则返回出错码。
52     return ioctl\_table[MAJOR(dev)](dev, cmd, arg);
53 }
54

```