```
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 */
   /* bitmap.c 程序含有处理 i 节点和磁盘块位图的代码 */
8  #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。
9                             // 这里使用了其中的 memset()函数。
10 #include <linux/sched.h>  // 调度程序头文件，定义任务结构 task_struct、任务 0 数据。
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
   //// 将指定地址（addr）处的一块 1024 字节内存清零。
   // 输入：eax = 0；ecx = 以长字为单位的数据块长度（BLOCK_SIZE/4）；edi = 指定起始地
   // 址 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
   //// 把指定地址开始的第 nr 个位偏移处的比特位置位（nr 可大于 32！）。返回原比特位值。
   // 输入：%0 -eax（返回值）；%1 -eax(0)；%2 -nr，位偏移值；%3 -(addr)，addr 的内容。
   // 第 20 行定义了一个局部寄存器变量 res。该变量将被保存在指定的 eax 寄存器中，以便于
   // 高效访问和操作。这种定义变量的方法主要用于内嵌汇编程序中。详细说明参见 gcc 手册
   // “在指定寄存器中的变量”。整个宏定义是一个语句表达式，该表达式值是最后 res 的值。
   // 第 21 行上的 btsl 指令用于测试并设置比特位（Bit Test and Set）。把基地址（%3）和
   // 比特位偏移值（%2）所指定的比特位值先保存到进位标志 CF 中，然后设置该比特位为 1。
   // 指令 setb 用于根据进位标志 CF 设置操作数（%al）。如果 CF=1 则%al =1，否则%al =0。
19 #define set_bit(nr,addr) ({\
20 register int res __asm__("ax"); \
21 __asm__ __volatile__("btsl %2,%3\n\tsetb %%al": \
22 "=a" (res):"" (0),"r" (nr),"m" (*(addr))); \
23 res;})
24
   //// 复位指定地址开始的第 nr 位偏移处的比特位。返回原比特位值的反码。
   // 输入：%0 -eax（返回值）；%1 -eax(0)；%2 -nr，位偏移值；%3 -(addr)，addr 的内容。
   // 第 27 行上的 btrl 指令用于测试并复位比特位（Bit Test and Reset）。其作用与上面的
   // btsl 类似，但是复位指定比特位。指令 setnb 用于根据进位标志 CF 设置操作数（%al）。
   // 如果 CF = 1 则%al = 0，否则%al = 1。
25 #define clear_bit(nr,addr) ({\
26 register int res __asm__("ax"); \
27 __asm__ __volatile__("btrl %2,%3\n\tsetnb %%al": \
28 "=a" (res):"" (0),"r" (nr),"m" (*(addr))); \
29 res;})
30
   //// 从 addr 开始寻找第 1 个 0 值比特位。
   // 输入：%0 - ecx(返回值)；%1 - ecx(0)；%2 - esi(addr)。
   // 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。addr 是缓冲块数据区的地址，扫描寻找的范围是 1024 字节（8192 比特位）。
31 #define find_first_zero(addr) ({ \
```

```
32 int __res; \
33 __asm__("cld\n" \                    // 清方向位。
34         "1:\tlodsl\n\t" \            // 取[esi]➔eax。
35         "notl %%eax\n\t" \           // eax 中每位取反。
36         "bsfl %%eax,%%edx\n\t" \     // 从位 0 扫描 eax 中是 1 的第 1 个位，其偏移值➔edx。
37         "je 2f\n\t" \                // 如果 eax 中全是 0，则向前跳转到标号 2 处(40 行)。
38         "addl %%edx,%%ecx\n\t" \     // 偏移值加入 ecx（ecx 是位图首个 0 值位的偏移值）。
39         "jmp 3f\n" \                 // 向前跳转到标号 3 处（结束）。
40         "2:\taddl $32,%%ecx\n\t" \   // 未找到 0 值位，则将 ecx 加 1 个长字的位偏移量 32。
41         "cmpl $8192,%%ecx\n\t" \     // 已经扫描了 8192 比特位（1024 字节）了吗？
42         "jl 1b\n" \                  // 若还没有扫描完 1 块数据，则向前跳转到标号 1 处。
43         "3:" \                       // 结束。此时 ecx 中是位偏移量。
44         :"=c" (__res):"c" (0),"S" (addr):"ax","dx","si"); \
45 __res;})
46
   //// 释放设备 dev 上数据区中的逻辑块 block。
   // 复位指定逻辑块 block 对应的逻辑块位图比特位。成功则返回 1，否则返回 0。
   // 参数：dev 是设备号，block 是逻辑块号（盘块号）。
47 int free_block(int dev, int block)
48 {
49         struct super_block * sb;
50         struct buffer_head * bh;
51
   // 首先取设备 dev 上文件系统的超级块信息，根据其中数据区开始逻辑块号和文件系统中逻辑
   // 块总数信息判断参数 block 的有效性。如果指定设备超级块不存在，则出错停机。若逻辑块
   // 号小于盘上数据区第 1 个逻辑块的块号或者大于设备上总逻辑块数，也出错停机。
52         if (!(sb = get_super(dev)))                    // fs/super.c，第 56 行。
53                 panic("trying to free block on nonexistent device");
54         if (block < sb->s_firstdatazone || block >= sb->s_nzones)
55                 panic("trying to free block not in datazone");
56         bh = get_hash_table(dev,block);
   // 然后从 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。该段代码的主要用途是如果该逻辑块目前存在于高速缓冲区中，就释放对应的缓
   // 冲块。
57         if (bh) {
58                 if (bh->b_count > 1) {      // 如果引用次数大于 1，则调用 brelse()，
59                         brelse(bh);         // b_count--后即退出，该块还有人用。
60                         return 0;
61                 }
62                 bh->b_dirt=0;               // 否则复位已修改和已更新标志。
63                 bh->b_uptodate=0;
64                 if (bh->b_count)            // 若此时 b_count 为 1，则调用 brelse()释放之。
65                         brelse(bh);
66         }
   // 接着我们复位 block 在逻辑块位图中的比特位（置 0）。先计算 block 在数据区开始算起的
   // 数据逻辑块号（从 1 开始计数）。然后对逻辑块(区块)位图进行操作，复位对应的比特位。
   // 如果对应比特位原来就是 0，则出错停机。由于 1 个缓冲块有 1024 字节，即 8192 比特位，
   // 因此 block/8192 即可计算出指定块 block 在逻辑位图中的哪个块上。而 block&8191 可
   // 以得到 block 在逻辑块位图当前块中的比特偏移位置。
67         block -= sb->s_firstdatazone - 1 ; // 即 block = block - ( s_firstdatazone -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                 printk("free_block: bit already cleared\n");
```

```
71          }
    // 最后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          return 1;
74  }
75
    ////向设备申请一个逻辑块（盘块，区块）。
    // 函数首先取得设备的超级块，并在超级块中的逻辑块位图中寻找第一个 0 值比特位（代表
    // 一个空闲逻辑块）。然后置位对应逻辑块在逻辑块位图中的比特位。接着为该逻辑块在缓
    // 冲区中取得一块对应缓冲块。最后将该缓冲块清零，并设置其已更新标志和已修改标志。
    // 并返回逻辑块号。函数执行成功则返回逻辑块号（盘块号），否则返回 0。
76  int new_block(int dev)
77  {
78          struct buffer_head * bh;
79          struct super_block * sb;
80          int i,j;
81
    // 首先获取设备 dev 的超级块。如果指定设备的超级块不存在，则出错停机。然后扫描文件
    // 系统的 8 块逻辑块位图，寻找首个 0 值比特位，以寻找空闲逻辑块，获取放置该逻辑块的
    // 块号。 如果全部扫描完 8 块逻辑块位图的所有比特位（i >= 8 或 j >= 8192）还没找到
    // 0 值比特位或者位图所在的缓冲块指针无效(bh = NULL)则 返回 0 退出（没有空闲逻辑块）。
82          if (!(sb = get_super(dev)))
83                  panic("trying to get new block from nonexistant device");
84          j = 8192;
85          for (i=0 ; i<8 ; i++)
86                  if (bh=sb->s_zmap[i])
87                          if ((j=find_first_zero(bh->b_data))<8192)
88                                  break;
89          if (i>=8 || !bh || j>=8192)
90                  return 0;
    // 接着设置找到的新逻辑块 j 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
    // 停机。否则置存放位图的对应缓冲区块已修改标志。因为逻辑块位图仅表示盘上数据区中
    // 逻辑块的占用情况，即逻辑块位图中比特位偏移值表示从数据区开始处算起的块号，因此
    // 这里需要加上数据区第 1 个逻辑块的块号，把 j 转换成逻辑块号。此时如果新逻辑块大于
    // 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。
91          if (set_bit(j,bh->b_data))
92                  panic("new_block: bit already set");
93          bh->b_dirt = 1;
94          j += i*8192 + sb->s_firstdatazone-1;
95          if (j >= sb->s_nzones)
96                  return 0;
    // 然后在高速缓冲区中为该设备上指定的逻辑块号取得一个缓冲块，并返回缓冲块头指针。
    // 因为刚取得的逻辑块其引用次数一定为 1（getblk()中会设置），因此若不为 1 则停机。
    // 最后将新逻辑块清零，并设置其已更新标志和已修改标志。然后释放对应缓冲块，返回
    // 逻辑块号。
97          if (!(bh=getblk(dev,j)))
98                  panic("new_block: cannot get block");
99          if (bh->b_count != 1)
100                 panic("new block: count is != 1");
101         clear_block(bh->b_data);
102         bh->b_uptodate = 1;
103         bh->b_dirt = 1;
104         brelse(bh);
```

```
105         return j;
106 }
107
    //// 释放指定的 i 节点。
    // 该函数首先判断参数给出的 i 节点号的有效性和可释放性。若 i 节点仍然在使用中则不能
    // 被释放。然后利用超级块信息对 i 节点位图进行操作，复位 i 节点号对应的 i 节点位图中
    // 比特位，并清空 i 节点结构。
108 void free_inode(struct m_inode * inode)
109 {
110         struct super_block * sb;
111         struct buffer_head * bh;
112
    // 首先判断参数给出的需要释放的 i 节点有效性或合法性。如果 i 节点指针=NULL，则退出。
    // 如果 i 节点上的设备号字段为 0，说明该节点没有使用。于是用 0 清空对应 i 节点所占内存
    // 区并返回。memset()定义在 include/string.h 第 395 行开始处。这里表示用 0 填写 inode
    // 指针指定处、长度是 sizeof(*inode) 的内存块。
113         if (!inode)
114                 return;
115         if (!inode->i_dev) {
116                 memset(inode,0,sizeof(*inode));
117                 return;
118         }
    // 如果此 i 节点还有其他程序引用，则不能释放，说明内核有问题，停机。如果文件连接数
    // 不为 0，则表示还有其他文件目录项在使用该节点，因此也不应释放，而应该放回等。
119         if (inode->i_count>1) {
120                 printk("trying to free inode with count=%d\n",inode->i_count);
121                 panic("free_inode");
122         }
123         if (inode->i_nlinks)
124                 panic("trying to free inode with links");
    // 在判断完 i 节点的合理性之后，我们开始利用其超级块信息对其中的 i 节点位图进行操作。
    // 首先取 i 节点所在设备的超级块，测试设备是否存在。然后判断 i 节点号的范围是否正确，
    // 如果 i 节点号等于 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）。
    // 如果该 i 节点对应的节点位图不存在，则出错。因为一个缓冲块的 i 节点位图有 8192 比
    // 特位。因此 i_num>>13（即 i_num/8192）可以得到当前 i 节点号所在的 s_imap[]项，即所
    // 在盘块。
125         if (!(sb = get_super(inode->i_dev)))
126                 panic("trying to free inode on nonexistent device");
127         if (inode->i_num < 1 || inode->i_num > sb->s_ninodes)
128                 panic("trying to free inode 0 or nonexistant inode");
129         if (!(bh=sb->s_imap[inode->i_num>>13]))
130                 panic("nonexistent imap in superblock");
    // 现在我们复位 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，则显示出错
    // 警告信息。最后置 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。
131         if (clear_bit(inode->i_num&8191,bh->b_data))
132                 printk("free_inode: bit already cleared.\n\r");
133         bh->b_dirt = 1;
134         memset(inode,0,sizeof(*inode));
135 }
136
    //// 为设备 dev 建立一个新 i 节点。初始化并返回该新 i 节点的指针。
    // 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。
137 struct m_inode * new_inode(int dev)
```

```
138 {
139         struct m_inode * inode;
140         struct super_block * sb;
141         struct buffer_head * bh;
142         int i,j;
143 
    // 首先从内存 i 节点表（inode_table）中获取一个空闲 i 节点项，并读取指定设备的超级块
    // 结构。 然后扫描超级块中 8 块 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。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），
    // 则放回先前申请的 i 节点表中的 i 节点，并返回空指针退出（没有空闲 i 节点）。
144         if (!(inode=get_empty_inode()))             // fs/inode.c，第 197 行。
145                 return NULL;
146         if (!(sb = get_super(dev)))                 // fs/super.c，第 56 行。
147                 panic("new_inode with unknown device");
148         j = 8192;
149         for (i=0 ; i<8 ; i++)
150                 if (bh=sb->s_imap[i])
151                         if ((j=find_first_zero(bh->b_data))<8192)
152                                 break;
153         if (!bh || j >= 8192 || j+i*8192 > sb->s_ninodes) {
154                 iput(inode);
155                 return NULL;
156         }
    // 现在我们已经找到了还未使用的 i 节点号 j。于是置位 i 节点 j 对应的 i 节点位图相应比
    // 特位（如果已经置位，则出错）。然后置 i 节点位图所在缓冲块已修改标志。最后初始化
    // 该 i 节点结构（i_ctime 是 i 节点内容改变时间）。
157         if (set_bit(j,bh->b_data))
158                 panic("new_inode: bit already set");
159         bh->b_dirt = 1;
160         inode->i_count=1;                       // 引用计数。
161         inode->i_nlinks=1;                      // 文件目录项链接数。
162         inode->i_dev=dev;                       // i 节点所在的设备号。
163         inode->i_uid=current->euid;             // i 节点所属用户 id。
164         inode->i_gid=current->egid;             // 组 id。
165         inode->i_dirt=1;                        // 已修改标志置位。
166         inode->i_num = j + i*8192;              // 对应设备中的 i 节点号。
167         inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;  // 设置时间。
168         return inode;                           // 返回该 i 节点指针。
169 }
170 
```