```
1  /*
2   *  linux/fs/namei.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   * Some corrections by tytso.
9   */
10
11 #include <linux/sched.h>  // 调度程序头文件，定义了任务结构 task_struct、任务 0 的数据等。
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。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。
18 #include <const.h>        // 常数符号头文件。目前仅定义 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
   // 由文件名查找对应 i 节点的内部函数。
21 static struct m_inode * _namei(const char * filename, struct m_inode * base,
22         int follow_links);
23
   // 下面宏中右侧表达式是访问数组的一种特殊使用方法。它基于这样的一个事实，即用数组名和
   // 数组下标所表示的数组项（例如 a[b]）的值等同于使用数组首指针（地址）加上该项偏移地址
   // 的形式的值 *(a + b)，同时可知项 a[b]也可以表示成 b[a]的形式。因此对于字符数组项形式
   // 为 "LoveYou"[2]（或者 2["LoveYou"]）就等同于*("LoveYou" + 2)。另外，字符串"LoveYou"
   // 在内存中被存储的位置就是其地址，因此数组项"LoveYou"[2]的值就是该字符串中索引值为 2
   // 的字符"v"所对应的 ASCII 码值 0x76，或用八进制表示就是 0166。在 C 语言中，字符也可以用
   // 其 ASCII 码值来表示，方法是在字符的 ASCII 码值前面加一个反斜杠。例如字符 "v"可以表示
   // 成"\x76"或者"\166"。因此对于不可显示的字符（例如 ASCII 码值为 0x00--0x1f 的控制字符）
   // 就可用其 ASCII 码值来表示。
   //
   // 下面是访问模式宏。x 是头文件 include/fcntl.h 中第 7 行开始定义的文件访问（打开）标志。
   // 这个宏根据文件访问标志 x 的值来索引双引号中对应的数值。双引号中有 4 个八进制数值（实
   // 际表示 4 个控制字符）："\004\002\006\377"，分别表示读、写和执行的权限为: r、w、rw
   // 和 wxrwxrwx，并且分别对应 x 的索引值 0--3。 例如，如果 x 为 2，则该宏返回八进制值 006，
   // 表示可读可写（rw）。另外，其中 O_ACCMODE = 00003，是索引值 x 的屏蔽码。
24 #define ACC_MODE(x) ("\004\002\006\377"[(x)&O_ACCMODE])
25
26 /*
27  * comment out this line if you want names > NAME_LEN chars to be
28  * truncated. Else they will be disallowed.
29  */
    /*
    * 如果想让文件名长度 > NAME_LEN 个的字符被截掉，就将下面定义注释掉。
    */
30 /* #define NO_TRUNCATE */
31
32 #define MAY_EXEC 1        // 可执行(可进入)。
33 #define MAY_WRITE 2       // 可写。
```

```
34 #define MAY_READ 4        // 可读。
35
36 /*
37  *      permission()
38  *
39  * is used to check for read/write/execute permissions on a file.
40  * I don't know if we should look at just the euid or both euid and
41  * uid, but that should be easily changed.
42  */
   /*
    *     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 - 文件的 i 节点指针；mask - 访问属性屏蔽码。
   // 返回：访问许可返回 1，否则返回 0。
43 static int permission(struct m_inode * inode,int mask)
44 {
45         int mode = inode->i_mode;                    // 文件访问属性。
46
47 /* 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 节点的链接计数值等于 0，表示该文件已被删除，则返回。
   // 否则，如果进程的有效用户 id（euid）与 i 节点的用户 id 相同，则取文件宿主的访问权限。
   // 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。
48         if (inode->i_dev && !inode->i_nlinks)
49                 return 0;
50         else if (current->euid==inode->i_uid)
51                 mode >>= 6;
52         else if (in_group_p(inode->i_gid))
53                 mode >>= 3;
   // 最后判断如果所取的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。
54         if (((mode & mask & 0007) == mask) || suser())
55                 return 1;
56         return 0;
57 }
58
59 /*
60  * ok, we cannot use strncmp, as the name is not in our data space.
61  * Thus we'll have to use match. No big problem. Match also makes
62  * some sanity tests.
63  *
64  * NOTE! unlike strncmp, match returns 1 for success, 0 for failure.
65  */
   /*
    * 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。
// 第 68 行上定义了一个局部寄存器变量 same。该变量将被保存在 eax 寄存器中，以便于高效
// 访问。
66 static int match(int len,const char * name,struct dir_entry * de)
67 {
68         register int same __asm__("ax");
69
// 首先判断函数参数的有效性。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（不匹配）。如果比较的长度 len 等于 0 并且目录项
// 中文件名的第 1 个字符是 '.'，并且只有这么一个字符，那么我们就认为是相同的，因此返
// 回 1（匹配）。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，
// 则也返回 0（不匹配）。
// 第 75 行上对目录项中文件名长度是否超过 len 的判断方法是检测 name[len] 是否为 NULL。
// 若长度超过 len，则 name[len]处就是一个不是 NULL 的普通字符。而对于长度为 len 的字符
// 串 name，字符 name[len]就应该是 NULL。
70         if (!de || !de->inode || len > NAME_LEN)
71                 return 0;
72         /* "" means "." ---> so paths like "/usr/lib//libc.a" work */
         /* "" 当作 "." 来看待 ---> 这样就能处理象 "/usr/lib//libc.a" 那样的路径名 */
73         if (!len && (de->name[0]=='.') && (de->name[1]=='\0'))
74                 return 1;
75         if (len < NAME_LEN && de->name[len])
76                 return 0;
// 然后使用嵌入汇编语句进行快速比较操作。它会在用户数据空间（fs 段）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)。
77         __asm__("cld\n\t"                       // 清方向标志位。
78                 "fs ; repe ; cmpsb\n\t"         // 用户空间执行循环比较[esi++]和[edi++]操作，
79                 "setz %%al"                     // 若比较结果一样（zf=0）则置 al=1（same=eax）。
80                 :"=a" (same)
81                 :"" (0),"S" ((long) name),"D" ((long) de->name),"c" (len)
82                 :"cx","di","si");
83         return same;                            // 返回比较结果。
84 }
85
86 /*
87  *      find_entry()
88  *
89  * finds an entry in the specified directory with the wanted name. It
90  * returns the cache buffer in which the entry was found, and the entry
91  * itself (as a parameter - res_dir). It does NOT read the inode of the
92  * entry - you'll have to do that yourself if you want to.
93  *
94  * This also takes care of the few special cases due to '..'-traversal
95  * over a pseudo-root and a mount point.
96  */
   /*
    *     find_entry()
    *
    * 在指定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
    * 缓冲块以及目录项本身（作为一个参数 - res_dir）。该函数并不读取目录项
```

```
 * 的 i 节点 - 如果需要的话则自己操作。
 *
 * 由于有'..'目录项，因此在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；
// 该函数在指定目录的数据（文件）中搜索指定文件名的目录项。并对指定文件名是'..'的
// 情况根据当前进行的相关设置进行特殊处理。关于函数参数传递指针的指针的作用，请参
// 见 linux/sched.c 第 151 行前的注释。
// 返回：成功则函数高速缓冲区指针，并在*res_dir 处返回的目录项结构指针。失败则返回
// 空指针 NULL。
97  static struct buffer_head * find_entry(struct m_inode ** dir,
98          const char * name, int namelen, struct dir_entry ** res_dir)
99  {
100         int entries;
101         int block,i;
102         struct buffer_head * bh;
103         struct dir_entry * de;
104         struct super_block * sb;
105
// 同样，本函数一上来也需要对函数参数的有效性进行判断和验证。如果我们在前面第 30 行
// 定义了符号常数 NO_TRUNCATE，那么如果文件名长度超过最大长度 NAME_LEN，则不予处理。
// 如果没有定义过 NO_TRUNCATE，那么在文件名长度超过最大长度 NAME_LEN 时截短之。
106 #ifdef NO_TRUNCATE
107         if (namelen > NAME_LEN)
108                 return NULL;
109 #else
110         if (namelen > NAME_LEN)
111                 namelen = NAME_LEN;
112 #endif

// 首先计算本目录中目录项项数 entries。 目录 i 节点 i_size 字段中含有本目录包含的数据
// 长度，因此其除以一个目录项的长度（16 字节）即可得到该目录中目录项数。然后置空返回
// 目录项结构指针。
113         entries = (*dir)->i_size / (sizeof (struct dir_entry));
114         *res_dir = NULL;
// 接下来我们对目录项文件名是'..'的情况进行特殊处理。如果当前进程指定的根 i 节点就是
// 函数参数指定的目录，则说明对于本进程来说，这个目录就是它的伪根目录，即进程只能访
// 问该目录中的项而不能后退到其父目录中去。也即对于该进程本目录就如同是文件系统的根
// 目录。因此我们需要将文件名修改为'.'。
// 否则，如果该目录的 i 节点号等于 ROOT_INO（1 号）的话，说明确实是文件系统的根 i 节点。
// 则取文件系统的超级块。如果被安装到的 i 节点存在，则先放回原 i 节点，然后对被安装到
// 的 i 节点进行处理。于是我们让*dir 指向该被安装到的 i 节点；并且该 i 节点的引用数加 1。
// 即针对这种情况，我们悄悄地进行了“偷梁换柱”工程:)
115 /* check for '..', as we might have to do some "magic" for it */
/* 检查目录项 '..'，因为我们可能需要对其进行特殊处理 */
116         if (namelen==2 && get_fs_byte(name)=='.' && get_fs_byte(name+1)=='.') {
117 /* '..' 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） */
118                 if ((*dir) == current->root)
119                         namelen=1;
120                 else if ((*dir)->i_num == ROOT_INO) {
```

```
121 /* '..' over a mount-point results in 'dir' being exchanged for the mounted
122    directory-inode. NOTE! We set mounted, so that we can iput the new dir */
    /* 在一个安装点上的 '..' 将导致目录交换到被安装文件系统的目录 i 节点上。注意！
       由于我们设置了 mounted 标志，因而我们能够放回该新目录 */
123                     sb=get_super((*dir)->i_dev);
124                     if (sb->s_imount) {
125                             iput(*dir);
126                             (*dir)=sb->s_imount;
127                             (*dir)->i_count++;
128                     }
129             }
130     }

    // 现在我们开始正常操作，查找指定文件名的目录项在什么地方。因此我们需要读取目录的数
    // 据，即取出目录 i 节点对应块设备数据区中的数据块（逻辑块）信息。这些逻辑块的块号保
    // 存在 i 节点结构的 i_zone[9]数组中。我们先取其中第 1 个块号。如果目录 i 节点指向的第
    // 一个直接磁盘块号为 0，则说明该目录竟然不含数据，这不正常。于是返回 NULL 退出。否则
    // 我们就从节点所在设备读取指定的目录项数据块。当然，如果不成功，则也返回 NULL 退出。
131     if (!(block = (*dir)->i_zone[0]))
132             return NULL;
133     if (!(bh = bread((*dir)->i_dev,block)))
134             return NULL;

    // 此时我们就在这个读取的目录 i 节点数据块中搜索匹配指定文件名的目录项。首先让 de 指
    // 向缓冲块中的数据块部分，并在不超过目录中目录项数的条件下，循环执行搜索。其中 i 是
    // 目录中的目录项索引号，在循环开始时初始化为 0。
135     i = 0;
136     de = (struct dir_entry *) bh->b_data;
137     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。
    // 再读入目录的下一个逻辑块。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，继续读目录的下一逻辑块。若该块不空，就让 de 指向该数据块，然后在其中
    // 继续搜索。其中 141 行上 i/DIR_ENTRIES_PER_BLOCK 可得到当前搜索的目录项所在目录文
    // 件中的块号，而 bmap()函数（inode.c，第 142 行）则可计算出在设备上对应的逻辑块号。
138             if ((char *)de >= BLOCK_SIZE+bh->b_data) {
139                     brelse(bh);
140                     bh = NULL;
141                     if (!(block = bmap(*dir,i/DIR_ENTRIES_PER_BLOCK)) ||
142                         !(bh = bread((*dir)->i_dev,block))) {
143                             i += DIR_ENTRIES_PER_BLOCK;
144                             continue;
145                     }
146                     de = (struct dir_entry *) bh->b_data;
147             }
    // 如果找到匹配的目录项的话，则返回该目录项结构指针 de 和该目录项 i 节点指针*dir 以
    // 及该目录项数据块指针 bh，并退出函数。否则继续在目录项数据块中比较下一个目录项。
148             if (match(namelen,name,de)) {
149                     *res_dir = de;
150                     return bh;
151             }
152             de++;
153             i++;
154     }
```

```
    // 如果指定目录中的所有目录项都搜索完后，还没有找到相应的目录项，则释放目录的数据
    // 块，最后返回 NULL（失败）。
155         brelse(bh);
156         return NULL;
157 }
158
159 /*
160  *      add_entry()
161  *
162  * adds a file entry to the specified directory, using the same
163  * semantics as find_entry(). It returns NULL if it failed.
164  *
165  * NOTE!! The inode part of 'de' is left at 0 - which means you
166  * may not sleep between calling this and putting something into
167  * the entry, as someone else might have used it while you slept.
168  */
    /*
     *     add_entry()
     * 使用与 find_entry()同样的方法，往指定目录中添加一指定文件名的目
     * 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去睡眠。 因为如果睡眠，
     * 那么其他人(进程)可能会使用了该目录项。
     */
    //// 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；
169 static struct buffer_head * add_entry(struct m_inode * dir,
170         const char * name, int namelen, struct dir_entry ** res_dir)
171 {
172         int block,i;
173         struct buffer_head * bh;
174         struct dir_entry * de;
175
    // 同样，本函数一上来也需要对函数参数的有效性进行判断和验证。如果我们在前面第 30 行
    // 定义了符号常数 NO_TRUNCATE，那么如果文件名长度超过最大长度 NAME_LEN，则不予处理。
    // 如果没有定义过 NO_TRUNCATE，那么在文件名长度超过最大长度 NAME_LEN 时截短之。
176         *res_dir = NULL;                    // 用于返回目录项结构指针。
177 #ifdef NO_TRUNCATE
178         if (namelen > NAME_LEN)
179                 return NULL;
180 #else
181         if (namelen > NAME_LEN)
182                 namelen = NAME_LEN;
183 #endif
    // 现在我们开始操作，向指定目录中添加一个指定文件名的目录项。因此我们需要先读取目录
    // 的数据，即取出目录 i 节点对应块设备数据区中的数据块（逻辑块）信息。这些逻辑块的块
    // 号保存在 i 节点结构的 i_zone[9]数组中。我们先取其中第 1 个块号。如果目录 i 节点指向
    // 的第一个直接磁盘块号为 0，则说明该目录竟然不含数据，这不正常。于是返回 NULL 退出。
    // 否则我们就从节点所在设备读取指定的目录项数据块。当然，如果不成功，则也返回 NULL
    // 退出。另外，如果参数提供的文件名长度等于 0，则也返回 NULL 退出。
184         if (!namelen)
```

```
185                 return NULL;
186         if (!(block = dir->i_zone[0]))
187                 return NULL;
188         if (!(bh = bread(dir->i_dev,block)))
189                 return NULL;
    // 此时我们就在这个目录 i 节点数据块中循环查找最后未使用的空目录项。首先让目录项结构
    // 指针 de 指向缓冲块中的数据块部分，即第一个目录项处。其中 i 是目录中的目录项索引号，
    // 在循环开始时初始化为 0。
190         i = 0;
191         de = (struct dir_entry *) bh->b_data;
192         while (1) {
    // 如果当前目录项数据块已经搜索完毕，但还没有找到需要的空目录项，则释放当前目录项数
    // 据块，再读入目录的下一个逻辑块。如果对应的逻辑块不存在就创建一块。若读取或创建操
    // 作失败则返回空。如果此次读取的磁盘逻辑块数据返回的缓冲块指针为空，说明这块逻辑块
    // 可能是因为不存在而新创建的空块，则把目录项索引值加上一块逻辑块所能容纳的目录项数
    // DIR_ENTRIES_PER_BLOCK，用以跳过该块并继续搜索。否则说明新读入的块上有目录项数据，
    // 于是让目录项结构指针 de 指向该块的缓冲块数据部分，然后在其中继续搜索。其中 196 行
    // 上的 i/DIR_ENTRIES_PER_BLOCK 可计算得到当前搜索的目录项 i 所在目录文件中的块号，
    // 而 create_block()函数（inode.c，第 147 行）则可读取或创建出在设备上对应的逻辑块。
193                 if ((char *)de >= BLOCK_SIZE+bh->b_data) {
194                         brelse(bh);
195                         bh = NULL;
196                         block = create_block(dir,i/DIR_ENTRIES_PER_BLOCK);
197                         if (!block)
198                                 return NULL;
199                         if (!(bh = bread(dir->i_dev,block))) { // 若空则跳过该块继续。
200                                 i += DIR_ENTRIES_PER_BLOCK;
201                                 continue;
202                         }
203                         de = (struct dir_entry *) bh->b_data;
204                 }
    // 如果当前所操作的目录项序号 i 乘上目录结构大小所的长度值已经超过了该目录 i 节点信息
    // 所指出的目录数据长度值 i_size ，则说明整个目录文件数据中没有由于删除文件留下的空
    // 目录项，因此我们只能把需要添加的新目录项附加到目录文件数据的末端处。于是对该处目
    // 录项进行设置（置该目录项的 i 节点指针为空），并更新该目录文件的长度值（加上一个目
    // 录项的长度），然后设置目录的 i 节点已修改标志，再更新该目录的改变时间为当前时间。
205                 if (i*sizeof(struct dir_entry) >= dir->i_size) {
206                         de->inode=0;
207                         dir->i_size = (i+1)*sizeof(struct dir_entry);
208                         dir->i_dirt = 1;
209                         dir->i_ctime = CURRENT_TIME;
210                 }
    // 若当前搜索的目录项 de 的 i 节点为空，则表示找到一个还未使用的空闲目录项或是添加的
    // 新目录项。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，置含有本目录项的相应高速缓冲块已修改标志。返回该目录项的指针以及该高
    // 速缓冲块的指针，退出。
211                 if (!de->inode) {
212                         dir->i_mtime = CURRENT_TIME;
213                         for (i=0; i < NAME_LEN ; i++)
214                                 de->name[i]=(i<namelen)?get_fs_byte(name+i):0;
215                         bh->b_dirt = 1;
216                         *res_dir = de;
217                         return bh;
```

```
218                 }
219                 de++;           // 如果该目录项已经被使用，则继续检测下一个目录项。
220                 i++;
221         }
    // 本函数执行不到这里。这也许是 Linus 在写这段代码时，先复制了上面 find_entry()函数
    // 的代码，而后修改成本函数的☺。
222         brelse(bh);
223         return NULL;
224 }
225
    //// 查找符号链接的 i 节点。
    // 参数：dir - 目录 i 节点；inode - 目录项 i 节点。
    // 返回：返回符号链接到文件的 i 节点指针。出错返回 NULL。
226 static struct m_inode * follow_link(struct m_inode * dir, struct m_inode * inode)
227 {
228         unsigned short fs;                      // 用于临时保存 fs 段寄存器值。
229         struct buffer_head * bh;
230
    // 首先判断函数参数的有效性。如果没有给出目录 i 节点，我们就使用进程任务结构中设置的
    // 根 i 节点，并把链接数增 1。如果没有给出目录项 i 节点，则放回目录 i 节点后返回 NULL。
    // 如果指定目录项不是一个符号链接，就直接返回目录项对应的 i 节点 inode。
231         if (!dir) {
232                 dir = current->root;
233                 dir->i_count++;
234         }
235         if (!inode) {
236                 iput(dir);
237                 return NULL;
238         }
239         if (!S_ISLNK(inode->i_mode)) {
240                 iput(dir);
241                 return inode;
242         }
    // 然后取 fs 段寄存器值。fs 通常保存着指向任务数据段的选择符 0x17。如果 fs 没有指向用户
    // 数据段，或者给出的目录项 i 节点第 1 个直接块块号等于 0，或者是读取第 1 个直接块出错，
    // 则放回 dir 和 inode 两个 i 节点并返回 NULL 退出。否则说明现在 fs 正指向用户数据段、并且
    // 我们已经成功地读取了这个符号链接目录项的文件内容，并且文件内容已经在 bh 指向的缓冲
    // 块数据区中。实际上，这个缓冲块数据区中仅包含一个链接指向的文件路径名字符串。
243         __asm__("mov %%fs,%0":"=r" (fs));
244         if (fs != 0x17 || !inode->i_zone[0] ||
245            !(bh = bread(inode->i_dev, inode->i_zone[0]))) {
246                 iput(dir);
247                 iput(inode);
248                 return NULL;
249         }
    // 此时我们已经不需要符号链接目录项的 i 节点了，于是把它放回。现在碰到一个问题，那就
    // 是内核函数处理的用户数据都是存放在用户数据空间中的，并使用了 fs 段寄存器来从用户
    // 空间传递数据到内核空间中。而这里需要处理的数据却在内核空间中。因此为了正确地处理
    // 位于内核中的用户数据，我们需要让 fs 段寄存器临时指向内核空间，即让 fs =0x10。并在
    // 调用函数处理完后再恢复原 fs 的值。最后释放相应缓冲块，并返回 _namei()解析得到的符
    // 号链接指向的文件 i 节点。
250         iput(inode);
251         __asm__("mov %0,%%fs"::"r" ((unsigned short) 0x10));
```

```
252         inode = _namei(bh->b_data,dir,0);
253         __asm__("mov %0,%%fs"::"r" (fs));
254         brelse(bh);
255         return inode;
256 }
257
258 /*
259  *      get_dir()
260  *
261  * Getdir traverses the pathname until it hits the topmost directory.
262  * It returns NULL on failure.
263  */
    /*
     *     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。
     */
    //// 从指定目录开始搜寻指定路径名的目录（或文件名）的 i 节点。
    // 参数：pathname - 路径名；inode - 指定起始目录的 i 节点。
    // 返回：目录或文件的 i 节点指针。失败时返回 NULL。
264 static struct m_inode * get_dir(const char * pathname, struct m_inode * inode)
265 {
266         char c;
267         const char * thisname;
268         struct buffer_head * bh;
269         int namelen,inr;
270         struct dir_entry * de;
271         struct m_inode * dir;
272
    // 首先判断参数有效性。如果给出的指定目录的 i 节点指针 inode 为空，则使用当前进程的当
    // 前工作目录 i 节点。如果用户指定路径名的第 1 个字符是'/'，则说明路径名是绝对路径名。
    // 则应该从当前进程任务结构中设置的根（或伪根）i 节点开始操作。于是我们需要先放回参
    // 数指定的或者设定的目录 i 节点，并取得进程使用的根 i 节点。然后把该 i 节点的引用计数
    // 加 1，并删除路径名的第 1 个字符 '/'。这样就可以保证当前进程只能以其设定的根 i 节点
    // 作为搜索的起点。
273         if (!inode) {
274                 inode = current->pwd;                   // 进程的当前工作目录 i 节点。
275                 inode->i_count++;
276         }
277         if ((c=get_fs_byte(pathname))=='/') {
278                 iput(inode);                            // 放回原 i 节点。
279                 inode = current->root;                  // 为进程指定的根 i 节点。
280                 pathname++;
281                 inode->i_count++;
282         }
    // 然后针对路径名中的各个目录名部分和文件名进行循环处理。在循环处理过程中，我们先要
    // 对当前正在处理的目录名部分的 i 节点进行有效性判断，并且把变量 thisname 指向当前正
    // 在处理的目录名部分。如果该 i 节点表明当前处理的目录名部分不是目录类型，或者没有可
    // 进入该目录的访问许可，则放回该 i 节点，并返回 NULL 退出。 当然在刚进入循环时，当前
    // 目录的 i 节点 inode 就是进程根 i 节点或者是当前工作目录的 i 节点，或者是参数指定的某
    // 个搜索起始目录的 i 节点。
283         while (1) {
```

```
284                 thisname = pathname;
285                 if (!S_ISDIR(inode->i_mode) || !permission(inode,MAY_EXEC)) {
286                         iput(inode);
287                         return NULL;
288                 }
    // 每次循环我们处理路径名中一个目录名（或文件名）部分。因此在每次循环中我们都要从路
    // 径名字符串中分离出一个目录名（或文件名）。方法是从当前路径名指针 pathname 开始处
    // 搜索检测字符，直到字符是一个结尾符（NULL）或者是一个'/'字符。此时变量 namelen 正
    // 好是当前处理目录名部分的长度，而变量 thisname 正指向该目录名部分的开始处。此时如
    // 果字符是结尾符 NULL，则表明已经搜索到路径名末尾，并已到达最后指定目录名或文件名，
    // 则返回该 i 节点指针退出。
    // 注意！如果路径名中最后一个名称也是一个目录名，但其后面没有加上 '/'字符，则函数不
    // 会返回该最后目录名的 i 节点！例如：对于路径名/usr/src/linux，该函数将只返回 src/目
    // 录名的 i 节点。
289                 for(namelen=0;(c=get_fs_byte(pathname++))&&(c!='/');namelen++)
290                         /* nothing */ ;
291                 if (!c)
292                         return inode;
    // 在得到当前目录名部分（或文件名）后，我们调用查找目录项函数 find_entry()在当前处
    // 理的目录中寻找指定名称的目录项。如果没有找到，则放回该 i 节点，并返回 NULL 退出。
    // 然后在找到的目录项中取出其 i 节点号 inr 和设备号 idev，释放包含该目录项的高速缓冲
    // 块并放回该 i 节点。 然后取节点号 inr 的 i 节点 inode，并以该目录项为当前目录继续循
    // 环处理路径名中的下一目录名部分（或文件名）。如果当前处理的目录项是一个符号链接
    // 名，则使用 follow_link()就可以得到其指向的目录项名的 i 节点。
293                 if (!(bh = find_entry(&inode,thisname,namelen,&de))) {
294                         iput(inode);
295                         return NULL;
296                 }
297                 inr = de->inode;                        // 当前目录名部分的 i 节点号。
298                 brelse(bh);
299                 dir = inode;
300                 if (!(inode = iget(dir->i_dev,inr))) {      // 取 i 节点内容。
301                         iput(dir);
302                         return NULL;
303                 }
304                 if (!(inode = follow_link(dir,inode)))
305                         return NULL;
306         }
307 }
308
309 /*
310  *      dir_namei()
311  *
312  * dir_namei() returns the inode of the directory of the
313  * specified name, and the name within that directory.
314  */
    /*
     *     dir_namei()
     *
     * 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。
     */
    // 参数：pathname - 目录路径名；namelen - 路径名长度；name - 返回的最顶层目录名。
```

```
    // base - 搜索起始目录的 i 节点。
    // 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称及长度。出错时返回 NULL。
    // 注意！！这里“最顶层目录”是指路径名中最靠近末端的目录。
315 static struct m_inode * dir_namei(const char * pathname,
316         int * namelen, const char ** name, struct m_inode * base)
317 {
318         char c;
319         const char * basename;
320         struct m_inode * dir;
321
    // 首先取得指定路径名最顶层目录的 i 节点。然后对路径名 pathname 进行搜索检测，查出最后
    // 一个'/'字符后面的名字字符串，计算其长度，并且返回最顶层目录的 i 节点指针。注意！如
    // 果路径名最后一个字符是斜杠字符'/'，那么返回的目录名为空，并且长度为 0。但返回的 i
    // 节点指针仍然指向最后一个'/'字符前目录名的 i 节点。参见第 289 行上的“注意”说明。
322         if (!(dir = get_dir(pathname,base)))      // base 是指定的起始目录 i 节点。
323                 return NULL;
324         basename = pathname;
325         while (c=get_fs_byte(pathname++))
326                 if (c=='/')
327                         basename=pathname;
328         *namelen = pathname-basename-1;
329         *name = basename;
330         return dir;
331 }
332
    //// 取指定路径名的 i 节点内部函数。
    // 参数：pathname - 路径名；base - 搜索起点目录 i 节点；follow_links - 是否跟随
    // 符号链接的标志，1 - 需要，0 不需要。
    // 返回：对应的 i 节点。
333 struct m_inode * _namei(const char * pathname, struct m_inode * base,
334         int follow_links)
335 {
336         const char * basename;
337         int inr,namelen;
338         struct m_inode * inode;
339         struct buffer_head * bh;
340         struct dir_entry * de;
341
    // 首先查找指定路径名中最顶层目录的目录名并得到其 i 节点。若不存在，则返回 NULL 退出。
    // 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。因此说明我
    // 们已经找到对应目录的 i 节点，可以直接返回该 i 节点退出。如果返回的名字长度不是 0，
    // 则我们以指定的起始目录 base，再次调用 dir_namei()函数来搜索顶层目录名，并根据返回
    // 的信息作类似判断。
311         if (!(dir = dir_namei(pathname,&namelen,&basename)))
312                 return NULL;
313         if (!namelen)                   /* special case: '/usr/' etc */
314                 return dir;             /* 对应于'/usr/'等情况 */
342         if (!(base = dir_namei(pathname,&namelen,&basename,base)))
343                 return NULL;
344         if (!namelen)                   /* special case: '/usr/' etc */
345                 return base;
    // 然后在返回的顶层目录中寻找指定文件名目录项的 i 节点。注意！因为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！ 例如：/usr/src/linux，将只
```

```
    // 返回 src/目录名的 i 节点。因为函数 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情况使用寻找目录项 i 节点函数 find_entry()进行
    // 处理。此时 de 中含有寻找到的目录项指针，而 dir 是包含该目录项的目录的 i 节点指针。
346         bh = find_entry(&base,basename,namelen,&de);
347         if (!bh) {
348                 iput(base);
349                 return NULL;
350         }
    // 接着取该目录项的 i 节点号，并释放包含该目录项的高速缓冲块并放回目录 i 节点。然后取
    // 对应节点号的 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 inode。如果当前处理的目录项是一个符号链接名，则使用 follow_link()得到其指向的
    // 目录项名的 i 节点。
351         inr = de->inode;
352         brelse(bh);
353         if (!(inode = iget(base->i_dev,inr))) {
354                 iput(base);
355                 return NULL;
356         }
357         if (follow_links)
358                 inode = follow_link(base,inode);
359         else
360                 iput(base);
361         inode->i_atime=CURRENT_TIME;
362         inode->i_dirt=1;
363         return inode;
364 }
365
    //// 取指定路径名的 i 节点，不跟随符号链接。
    // 参数：pathname - 路径名。
    // 返回：对应的 i 节点。
366 struct m_inode * lnamei(const char * pathname)
367 {
368         return _namei(pathname, NULL, 0);
369 }
370
371 /*
372  *      namei()
373  *
374  * is used by most simple commands to get the inode of a specified name.
375  * Open, link etc use their own routines, but this is enough for things
376  * like 'chmod' etc.
377  */
    /*
     *     namei()
     *
     * 该函数被许多简单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，跟随符号链接。
    // 参数：pathname - 路径名。
    // 返回：对应的 i 节点。
378 struct m_inode * namei(const char * pathname)
379 {
```

```
380         return _namei(pathname,NULL,1);
381 }
382
383 /*
384  *      open_namei()
385  *
386  * namei for open - this is in fact almost the whole open-routine.
387  */
    /*
     *     open_namei()
     *
     * open()函数使用的 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 filename 是文件路径名，flag 是打开文件标志，可取值 O_RDONLY（只读）、O_WRONLY
    // （只写）或 O_RDWR（读写），以及 O_CREAT（创建）、O_EXCL（被创建文件必须不存在）、
    // O_APPEND（在文件尾添加数据）等其他一些标志的组合。如果本调用创建了一个新文件，则
    // mode 就用于指定文件的许可属性。这些属性有 S_IRWXU（文件宿主具有读、写和执行权限）、
    // S_IRUSR（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。参见包含文件 sys/stat.h、fcntl.h。
    // 返回：成功返回 0，否则返回出错码；res_inode - 返回对应文件路径名的 i 节点指针。
388 int open_namei(const char * pathname, int flag, int mode,
389         struct m_inode ** res_inode)
390 {
391         const char * basename;
392         int inr,dev,namelen;
393         struct m_inode * dir, *inode;
394         struct buffer_head * bh;
395         struct dir_entry * de;
396
    // 首先对函数参数进行合理的处理。如果文件访问模式标志是只读（0），但是文件截零标志
    // O_TRUNC 却置位了，则在文件打开标志中添加只写标志 O_WRONLY。这样做的原因是由于截零
    // 标志 O_TRUNC 必须在文件可写情况下才有效。然后使用当前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 I_REGULAR。该标志将用于打开的文件不存
    // 在而需要创建文件时，作为新文件的默认属性。参见下面 411 行上的注释。
397         if ((flag & O_TRUNC) && !(flag & O_ACCMODE))
398                 flag |= O_WRONLY;
399         mode &= 0777 & ~current->umask;
400         mode |= I_REGULAR;              // 常规文件标志。见参见 include/const.h 文件）。
    // 然后根据指定的路径名寻找到对应的 i 节点，以及最顶端目录名及其长度。此时如果最顶端
    // 目录名长度为 0（ 例如'/usr/' 这种路径名的情况），那么若操作不是读写、创建和文件长
    // 度截 0，则表示是在打开一个目录名文件操作。于是直接返回该目录的 i 节点并返回 0 退出。
    // 否则说明进程操作非法，于是放回该 i 节点，返回出错码。
401         if (!(dir = dir_namei(pathname,&namelen,&basename,NULL)))
402                 return -ENOENT;
403         if (!namelen) {                 /* special case: '/usr/' etc */
404                 if (!(flag & (O_ACCMODE|O_CREAT|O_TRUNC))) {
405                         *res_inode=dir;
406                         return 0;
407                 }
408                 iput(dir);
409                 return -EISDIR;
```

```
410         }
    // 接着根据上面得到的最顶层目录名的 i 节点 dir，在其中查找取得路径名字符串中最后的文
    // 件名对应的目录项结构 de，并同时得到该目录项所在的高速缓冲区指针。 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。 此时如
    // 果不是创建文件，则放回该目录的 i 节点，返回出错号退出。如果用户在该目录没有写的权
    // 力，则放回该目录的 i 节点，返回出错号退出。
411         bh = find_entry(&dir,basename,namelen,&de);
412         if (!bh) {
413                 if (!(flag & O_CREAT)) {
414                         iput(dir);
415                         return -ENOENT;
416                 }
417                 if (!permission(dir,MAY_WRITE)) {
418                         iput(dir);
419                         return -EACCES;
420                 }
    // 现在我们确定了是创建操作并且有写操作许可。 因此我们就在目录 i 节点对应设备上申请
    // 一个新的 i 节点给路径名上指定的文件使用。 若失败则放回目录的 i 节点，并返回没有空
    // 间出错码。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。然后并在指定目录 dir 中添加一个新目录项。
421                 inode = new_inode(dir->i_dev);
422                 if (!inode) {
423                         iput(dir);
424                         return -ENOSPC;
425                 }
426                 inode->i_uid = current->euid;
427                 inode->i_mode = mode;
428                 inode->i_dirt = 1;
429                 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，放回该 i 节点与目录的 i 节点并返回出错码退出。 否则
    // 说明添加目录项操作成功。 于是我们来设置该新目录项的一些初始值：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，放回目录的 i 节
    // 点。返回新目录项的 i 节点指针，并成功退出。
430                 if (!bh) {
431                         inode->i_nlinks--;
432                         iput(inode);
433                         iput(dir);
434                         return -ENOSPC;
435                 }
436                 de->inode = inode->i_num;
437                 bh->b_dirt = 1;
438                 brelse(bh);
439                 iput(dir);
440                 *res_inode = inode;
441                 return 0;
442         }
    // 若上面（411 行）在目录中取文件名对应目录项结构的操作成功（即 bh 不为 NULL），则说
    // 明指定打开的文件已经存在。于是取出该目录项的 i 节点号和其所在设备号，并释放该高速
    // 缓冲区以及放回目录的 i 节点。如果此时独占操作标志 O_EXCL 置位，但现在文件已经存在，
    // 则返回文件已存在出错码退出。
443         inr = de->inode;
444         dev = dir->i_dev;
```

```
445         brelse(bh);
446         if (flag & O_EXCL) {
447                 iput(dir);
448                 return -EEXIST;
449         }
    // 然后我们读取该目录项的 i 节点内容。若该 i 节点是一个目录的 i 节点并且访问模式是只
    // 写或读写，或者没有访问的许可权限，则放回该 i 节点，返回访问权限出错码退出。
450         if (!(inode = follow_link(dir,iget(dev,inr))))
451                 return -EACCES;
452         if ((S_ISDIR(inode->i_mode) && (flag & O_ACCMODE)) ||
453             !permission(inode,ACC_MODE(flag))) {
454                 iput(inode);
455                 return -EPERM;
456         }
    // 接着我们更新该 i 节点的访问时间字段值为当前时间。如果设立了截 0 标志，则将该 i 节
    // 点的文件长度截为 0。最后返回该目录项 i 节点的指针，并返回 0（成功）。
457         inode->i_atime = CURRENT_TIME;
458         if (flag & O_TRUNC)
459                 truncate(inode);
460         *res_inode = inode;
461         return 0;
462 }
463
    //// 创建一个设备特殊文件或普通文件节点（node）。
    // 该函数创建名称为 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）。
    // 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。
    // 返回：成功则返回 0，否则返回出错码。
464 int sys_mknod(const char * filename, int mode, int dev)
465 {
466         const char * basename;
467         int namelen;
468         struct m_inode * dir, * inode;
469         struct buffer_head * bh;
470         struct dir_entry * de;
471
    // 首先检查操作许可和参数的有效性并取路径名中顶层目录的 i 节点。如果不是超级用户，则
    // 返回访问许可出错码。如果找不到对应路径名中顶层目录的 i 节点，则返回出错码。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，放回该目录 i 节点，返
    // 回出错码退出。如果在该目录中没有写的权限，则放回该目录的 i 节点，返回访问许可出错
    // 码退出。如果不是超级用户，则返回访问许可出错码。
472         if (!suser())
473                 return -EPERM;
474         if (!(dir = dir_namei(filename,&namelen,&basename, NULL)))
475                 return -ENOENT;
476         if (!namelen) {
477                 iput(dir);
478                 return -ENOENT;
479         }
480         if (!permission(dir,MAY_WRITE)) {
481                 iput(dir);
482                 return -EPERM;
483         }
```

```
    // 然后我们搜索一下路径名指定的文件是否已经存在。若已经存在则不能创建同名文件节点。
    // 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缓冲区块并放回
    // 目录的 i 节点，返回文件已经存在的出错码退出。
484         bh = find_entry(&dir,basename,namelen,&de);
485         if (bh) {
486                 brelse(bh);
487                 iput(dir);
488                 return -EEXIST;
489         }
    // 否则我们就申请一个新的 i 节点，并设置该 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 节点的直接逻辑块指针 0 等于设备号。即对于设备文件来说，
    // 其 i 节点的 i_zone[0]中存放的是该设备文件所定义设备的设备号。然后设置该 i 节点的修
    // 改时间、访问时间为当前时间，并设置 i 节点已修改标志。
490         inode = new_inode(dir->i_dev);
491         if (!inode) {           // 若不成功则放回目录 i 节点，返回无空间出错码退出。
492                 iput(dir);
493                 return -ENOSPC;
494         }
495         inode->i_mode = mode;
496         if (S_ISBLK(mode) || S_ISCHR(mode))
497                 inode->i_zone[0] = dev;
498         inode->i_mtime = inode->i_atime = CURRENT_TIME;
499         inode->i_dirt = 1;
    // 接着为这个新的 i 节点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
    // 块指针为 NULL），则放回目录的 i 节点；把所申请的 i 节点引用连接计数复位，并放回该
    // i 节点，返回出错码退出。
500         bh = add_entry(dir,basename,namelen,&de);
501         if (!bh) {
502                 iput(dir);
503                 inode->i_nlinks=0;
504                 iput(inode);
505                 return -ENOSPC;
506         }
    // 现在添加目录项操作也成功了，于是我们来设置这个目录项内容。令该目录项的 i 节点字
    // 段等于新 i 节点号，并置高速缓冲区已修改标志，放回目录和新的 i 节点，释放高速缓冲
    // 区，最后返回 0(成功)。
507         de->inode = inode->i_num;
508         bh->b_dirt = 1;
509         iput(dir);
510         iput(inode);
511         brelse(bh);
512         return 0;
513 }
514
    ////  创建一个目录。
    // 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。
515 int sys_mkdir(const char * pathname, int mode)
516 {
517         const char * basename;
518         int namelen;
519         struct m_inode * dir, * inode;
520         struct buffer_head * bh, *dir_block;
```

```
521         struct dir_entry * de;
522
    // 首先检查参数的有效性并取路径名中顶层目录的 i 节点。如果找不到对应路径名中顶层目录
    // 的 i 节点，则返回出错码。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，放回该目录 i 节点，返回出错码退出。如果在该目录中没有写的权限，则放回该
    // 目录的 i 节点，返回访问许可出错码退出。如果不是超级用户，则返回访问许可出错码。
523         if (!(dir = dir_namei(pathname,&namelen,&basename, NULL)))
524                 return -ENOENT;
525         if (!namelen) {
526                 iput(dir);
527                 return -ENOENT;
528         }
529         if (!permission(dir,MAY_WRITE)) {
530                 iput(dir);
531                 return -EPERM;
532         }
    // 然后我们搜索一下路径名指定的目录名是否已经存在。若已经存在则不能创建同名目录节点。
    // 如果对应路径名上最后的目录名的目录项已经存在，则释放包含该目录项的缓冲区块并放回
    // 目录的 i 节点，返回文件已经存在的出错码退出。否则我们就申请一个新的 i 节点，并设置
    // 该 i 节点的属性模式：置该新 i 节点对应的文件长度为 32 字节 （2 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。2 个目录项分别用于'.'和'..'目录。
533         bh = find_entry(&dir,basename,namelen,&de);
534         if (bh) {
535                 brelse(bh);
536                 iput(dir);
537                 return -EEXIST;
538         }
539         inode = new_inode(dir->i_dev);
540         if (!inode) {               // 若不成功则放回目录的 i 节点，返回无空间出错码。
541                 iput(dir);
542                 return -ENOSPC;
543         }
544         inode->i_size = 32;
545         inode->i_dirt = 1;
546         inode->i_mtime = inode->i_atime = CURRENT_TIME;
    // 接着为该新 i 节点申请一用于保存目录项数据的磁盘块，并令 i 节点的第一个直接块指针等
    // 于该块号。如果申请失败则放回对应目录的 i 节点；复位新申请的 i 节点连接计数；放回该
    // 新的 i 节点，返回没有空间出错码退出。否则置该新的 i 节点已修改标志。
547         if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
548                 iput(dir);
549                 inode->i_nlinks--;
550                 iput(inode);
551                 return -ENOSPC;
552         }
553         inode->i_dirt = 1;
    // 从设备上读取新申请的磁盘块（目的是把对应块放到高速缓冲区中）。若出错，则放回对应
    // 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；放回该新的 i 节点，返
    // 回没有空间出错码退出。
554         if (!(dir_block=bread(inode->i_dev,inode->i_zone[0]))) {
555                 iput(dir);
556                 inode->i_nlinks--;
557                 iput(inode);
558                 return -ERROR;
```

```
559         }
    // 然后我们在缓冲块中建立起所创建目录文件中的 2 个默认的新目录项（'.'和'..'）结构数
    // 据。首先令 de 指向存放目录项的数据块，然后置该目录项的 i 节点号字段等于新申请的 i
    // 节点号，名字字段等于"."。 然后 de 指向下一个目录项结构，并在该结构中存放上级目录
    // 的 i 节点号和名字".."。然后设置该高速缓冲块已修改标志，并释放该缓冲块。再初始化
    // 设置新 i 节点的模式字段，并置该 i 节点已修改标志。
560         de = (struct dir_entry *) dir_block->b_data;
561         de->inode=inode->i_num;                     // 设置'.'目录项。
562         strcpy(de->name,".");
563         de++;
564         de->inode = dir->i_num;                     // 设置'..'目录项。
565         strcpy(de->name,"..");
566         inode->i_nlinks = 2;
567         dir_block->b_dirt = 1;
568         brelse(dir_block);
569         inode->i_mode = I_DIRECTORY | (mode & 0777 & ~current->umask);
570         inode->i_dirt = 1;
    // 现在我们在指定目录中新添加一个目录项，用于存放新建目录的 i 节点和目录名。如果失
    // 败（包含该目录项的高速缓冲区指针为 NULL），则放回目录的 i 节点；所申请的 i 节点引
    // 用连接计数复位，并放回该 i 节点。返回出错码退出。
571         bh = add_entry(dir,basename,namelen,&de);
572         if (!bh) {
573                 iput(dir);
574                 inode->i_nlinks=0;
575                 iput(inode);
576                 return -ENOSPC;
577         }
    // 最后令该新目录项的 i 节点字段等于新 i 节点号，并置高速缓冲块已修改标志，放回目录
    // 和新的 i 节点，释放高速缓冲区，最后返回 0（成功）。
578         de->inode = inode->i_num;
579         bh->b_dirt = 1;
580         dir->i_nlinks++;
581         dir->i_dirt = 1;
582         iput(dir);
583         iput(inode);
584         brelse(bh);
585         return 0;
586 }
587
588 /*
589  * routine to check that the specified directory is empty (for rmdir)
590  */
    /*
     * 用于检查指定的目录是否为空的子程序（用于 rmdir 系统调用）。
     */
    //// 检查指定目录是否空。
    // 参数：inode - 指定目录的 i 节点指针。
    // 返回：1 - 目录中是空的；0 - 不空。
591 static int empty_dir(struct m_inode * inode)
592 {
593         int nr,block;
594         int len;
595         struct buffer_head * bh;
```

```
596         struct dir_entry * de;
597
    // 首先计算指定目录中现有目录项个数并检查开始两个特定目录项中信息是否正确。一个目录
    // 中应该起码有 2 个目录项：即"."和".."。 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，或者该直接块读不出，则显示警告信息“设备 dev 上目
    // 录错”，返回 0(失败)。
598         len = inode->i_size / sizeof (struct dir_entry);       // 目录中目录项个数。
599         if (len<2 || !inode->i_zone[0] ||
600             !(bh=bread(inode->i_dev,inode->i_zone[0]))) {
601                 printk("warning - bad directory on dev %04x\n",inode->i_dev);
602                 return 0;
603         }
    // 此时 bh 所指缓冲块中含有目录项数据。我们让目录项指针 de 指向缓冲块中第 1 个目录项。
    // 对于第 1 个目录项（"."），它的 i 节点号字段 inode 应该等于当前目录的 i 节点号。对于
    // 第 2 个目录项（".."），它的 i 节点号字段 inode 应该等于上一层目录的 i 节点号，不会
    // 为 0。因此如果第 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。
604         de = (struct dir_entry *) bh->b_data;
605         if (de[0].inode != inode->i_num || !de[1].inode ||
606             strcmp(".",de[0].name) || strcmp("..",de[1].name)) {
607                 printk("warning - bad directory on dev %04x\n",inode->i_dev);
608                 return 0;
609         }
    // 然后我们令 nr 等于目录项序号（从 0 开始计）；de 指向第三个目录项。并循环检测该目录
    // 中其余所有的（len - 2）个目录项，看有没有目录项的 i 节点号字段不为 0（被使用）。
610         nr = 2;
611         de += 2;
612         while (nr<len) {
    // 如果该块磁盘块中的目录项已经全部检测完毕，则释放该磁盘块的缓冲块，并读取目录数据
    // 文件中下一块含有目录项的磁盘块。读取的方法是根据当前检测的目录项序号 nr 计算出对
    // 应目录项在目录数据文件中的数据块号（nr/DIR_ENTRIES_PER_BLOCK），然后使用 bmap()
    // 函数取得对应的盘块号 block，再使用读设备盘块函数 bread() 把相应盘块读入缓冲块中，
    // 并返回该缓冲块的指针。若所读取的相应盘块没有使用（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。
613                 if ((void *) de >= (void *) (bh->b_data+BLOCK_SIZE)) {
614                         brelse(bh);
615                         block=bmap(inode,nr/DIR_ENTRIES_PER_BLOCK);
616                         if (!block) {
617                                 nr += DIR_ENTRIES_PER_BLOCK;
618                                 continue;
619                         }
620                         if (!(bh=bread(inode->i_dev,block)))
621                                 return 0;
622                         de = (struct dir_entry *) bh->b_data;
623                 }
    // 对于 de 指向的当前目录项，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。否则，若还没有查询完该目录中的所有目录项，
    // 则把目录项序号 nr 增 1、de 指向下一个目录项，继续检测。
624                 if (de->inode) {
625                         brelse(bh);
626                         return 0;
627                 }
```

```
628                 de++;
629                 nr++;
630         }
    // 执行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)，则释放缓冲块返回 1。
631         brelse(bh);
632         return 1;
633 }
634
    //// 删除目录。
    // 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。
635 int sys_rmdir(const char * name)
636 {
637         const char * basename;
638         int namelen;
639         struct m_inode * dir, * inode;
640         struct buffer_head * bh;
641         struct dir_entry * de;
642
    // 首先检查参数的有效性并取路径名中顶层目录的 i 节点。如果找不到对应路径名中顶层目录
    // 的 i 节点，则返回出错码。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，放回该目录 i 节点，返回出错码退出。如果在该目录中没有写的权限，则放回该
    // 目录的 i 节点，返回访问许可出错码退出。如果不是超级用户，则返回访问许可出错码。
643         if (!(dir = dir_namei(name,&namelen,&basename, NULL)))
644                 return -ENOENT;
645         if (!namelen) {
646                 iput(dir);
647                 return -ENOENT;
648         }
649         if (!permission(dir,MAY_WRITE)) {
650                 iput(dir);
651                 return -EPERM;
652         }
    // 然后根据指定目录的 i 节点和目录名利用函数 find_entry()寻找对应目录项，并返回包含该
    // 目录项的缓冲块指针 bh、包含该目录项的目录的 i 节点指针 dir 和该目录项指针 de。再根据
    // 该目录项 de 中的 i 节点号利用 iget()函数得到对应的 i 节点 inode。如果对应路径名上最
    // 后目录名的目录项不存在，则释放包含该目录项的高速缓冲区，放回目录的 i 节点，返回文
    // 件已经存在出错码，并退出。如果取目录项的 i 节点出错，则放回目录的 i 节点，并释放含
    // 有目录项的高速缓冲区，返回出错号。
653         bh = find_entry(&dir,basename,namelen,&de);
654         if (!bh) {
655                 iput(dir);
656                 return -ENOENT;
657         }
658         if (!(inode = iget(dir->i_dev, de->inode))) {
659                 iput(dir);
660                 brelse(bh);
661                 return -EPERM;
662         }
    // 此时我们已有包含要被删除目录项的目录 i 节点 dir、要被删除目录项的 i 节点 inode 和要
    // 被删除目录项指针 de。下面我们通过对这 3 个对象中信息的检查来验证删除操作的可行性。

    // 若该目录设置了受限删除标志并且进程的有效用户 id（euid）不是 root，并且进程的有效
```

```
    // 用户 id（euid）不等于该 i 节点的用户 id，则表示当前进程没有权限删除该目录，于是放
    // 回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，然后释放高速缓冲区，返回
    // 出错码。
663         if ((dir->i_mode & S_ISVTX) && current->euid &&
664             inode->i_uid != current->euid) {
665                 iput(dir);
666                 iput(inode);
667                 brelse(bh);
668                 return -EPERM;
669         }
    // 如果要被删除的目录项 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，释放高速缓冲块，返回出错码。
670         if (inode->i_dev != dir->i_dev || inode->i_count>1) {
671                 iput(dir);
672                 iput(inode);
673                 brelse(bh);
674                 return -EPERM;
675         }
    // 如果要被删除目录的目录项 i 节点就等于包含该需删除目录的目录 i 节点，则表示试图删除
    // "."目录，这是不允许的。于是放回包含要删除目录名的目录 i 节点和要删除目录的 i 节点，
    // 释放高速缓冲块，返回出错码。
676         if (inode == dir) {     /* we may not delete ".", but "../dir" is ok */
677                 iput(inode);
678                 iput(dir);
679                 brelse(bh);
680                 return -EPERM;
681         }
    // 若要被删除目录 i 节点的属性表明这不是一个目录，则本删除操作的前提完全不存在。于是
    // 放回包含删除目录名的目录 i 节点和该要删除目录的 i 节点，释放高速缓冲块，返回出错码。
682         if (!S_ISDIR(inode->i_mode)) {
683                 iput(inode);
684                 iput(dir);
685                 brelse(bh);
686                 return -ENOTDIR;
687         }
    // 若该需被删除的目录不空，则也不能删除。于是放回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，释放高速缓冲块，返回出错码。
688         if (!empty_dir(inode)) {
689                 iput(inode);
690                 iput(dir);
691                 brelse(bh);
692                 return -ENOTEMPTY;
693         }
    // 对于一个空目录，其目录项链接数应该为 2（链接到上层目录和本目录）。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。但删除操作仍然继续执行。于是置该需被
    // 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
    // 缓冲块已修改标志，并释放该缓冲块。然后再置被删除目录 i 节点的链接数为 0（表示空闲），
    // 并置 i 节点已修改标志。
694         if (inode->i_nlinks != 2)
695                 printk("empty directory has nlink!=2 (%d)",inode->i_nlinks);
696         de->inode = 0;
697         bh->b_dirt = 1;
```

```
698         brelse(bh);
699         inode->i_nlinks=0;
700         inode->i_dirt=1;
    // 再将包含被删除目录名的目录的 i 节点链接计数减 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。最后放回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（删除操作成功）。
701         dir->i_nlinks--;
702         dir->i_ctime = dir->i_mtime = CURRENT_TIME;
703         dir->i_dirt=1;
704         iput(dir);
705         iput(inode);
706         return 0;
707 }
708
    //// 删除（释放）文件名对应的目录项。
    // 从文件系统删除一个名字。如果是文件的最后一个链接，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（路径名）。
    // 返回：成功则返回 0，否则返回出错号。
709 int sys_unlink(const char * name)
710 {
711         const char * basename;
712         int namelen;
713         struct m_inode * dir, * inode;
714         struct buffer_head * bh;
715         struct dir_entry * de;
716
    // 首先检查参数的有效性并取路径名中顶层目录的 i 节点。如果找不到对应路径名中顶层目录
    // 的 i 节点，则返回出错码。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，放回该目录 i 节点，返回出错码退出。如果在该目录中没有写的权限，则放回该
    // 目录的 i 节点，返回访问许可出错码退出。如果找不到对应路径名顶层目录的 i 节点，则返
    // 回出错码。
717         if (!(dir = dir_namei(name,&namelen,&basename, NULL)))
718                 return -ENOENT;
719         if (!namelen) {
720                 iput(dir);
721                 return -ENOENT;
722         }
723         if (!permission(dir,MAY_WRITE)) {
724                 iput(dir);
725                 return -EPERM;
726         }
    // 然后根据指定目录的 i 节点和目录名利用函数 find_entry()寻找对应目录项，并返回包含该
    // 目录项的缓冲块指针 bh、包含该目录项的目录的 i 节点指针 dir 和该目录项指针 de。再根据
    // 该目录项 de 中的 i 节点号利用 iget()函数得到对应的 i 节点 inode。如果对应路径名上最
    // 后目录名的目录项不存在，则释放包含该目录项的高速缓冲区，放回目录的 i 节点，返回文
    // 件已经存在出错码，并退出。如果取目录项的 i 节点出错，则放回目录的 i 节点，并释放含
    // 有目录项的高速缓冲区，返回出错号。
727         bh = find_entry(&dir,basename,namelen,&de);
728         if (!bh) {
729                 iput(dir);
730                 return -ENOENT;
731         }
```

```
732         if (!(inode = iget(dir->i_dev, de->inode))) {
733                 iput(dir);
734                 brelse(bh);
735                 return -ENOENT;
736         }
    // 此时我们已有包含要被删除目录项的目录 i 节点 dir、要被删除目录项的 i 节点 inode 和要
    // 被删除目录项指针 de。下面我们通过对这 3 个对象中信息的检查来验证删除操作的可行性。

    // 若该目录设置了受限删除标志并且进程的有效用户 id（euid）不是 root，并且进程的 euid
    // 不等于该 i 节点的用户 id，并且进程的 euid 也不等于目录 i 节点的用户 id，则表示当前进
    // 程没有权限删除该目录，于是放回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，
    // 然后释放高速缓冲块，返回出错码。
737         if ((dir->i_mode & S_ISVTX) && !suser() &&
738             current->euid != inode->i_uid &&
739             current->euid != dir->i_uid) {
740                 iput(dir);
741                 iput(inode);
742                 brelse(bh);
743                 return -EPERM;
744         }
    // 如果该指定文件名是一个目录，则也不能删除。放回该目录 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块，返回出错号。
745         if (S_ISDIR(inode->i_mode)) {
746                 iput(inode);
747                 iput(dir);
748                 brelse(bh);
749                 return -EPERM;
750         }
    // 如果该 i 节点的链接计数值已经为 0，则显示警告信息，并修正其为 1。
751         if (!inode->i_nlinks) {
752                 printk("Deleting nonexistent file (%04x:%d), %d\n",
753                         inode->i_dev,inode->i_num,inode->i_nlinks);
754                 inode->i_nlinks=1;
755         }
    // 现在我们可以删除文件名对应的目录项了。于是将该文件名目录项中的 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块已修改标志，释放该高速缓冲块。
756         de->inode = 0;
757         bh->b_dirt = 1;
758         brelse(bh);
    // 然后把文件名对应 i 节点的链接数减 1，置已修改标志，更新改变时间为当前时间。最后放
    // 回该 i 节点和目录的 i 节点，返回 0（成功）。如果是文件的最后一个链接，即 i 节点链接
    // 数减 1 后等于 0，并且此时没有进程正打开该文件，那么在调用 iput()放回 i 节点时，该文
    // 件也将被删除，并释放所占用的设备空间。参见 fs/inode.c，第 183 行。
759         inode->i_nlinks--;
760         inode->i_dirt = 1;
761         inode->i_ctime = CURRENT_TIME;
762         iput(inode);
763         iput(dir);
764         return 0;
765 }
766
    //// 建立符号链接。
    // 为一个已存在文件创建一个符号链接（也称为软连接 - 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。
767 int sys_symlink(const char * oldname, const char * newname)
768 {
769         struct dir_entry * de;
770         struct m_inode * dir, * inode;
771         struct buffer_head * bh, * name_block;
772         const char * basename;
773         int namelen, i;
774         char c;
775
    // 首先查找新路径名的最顶层目录的 i 节点 dir，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，则返回出错号。如果新路径名中不包括文件名，则放回新路径名目录的 i
    // 节点，返回出错号。另外，如果用户没有在新目录中写的权限，则也不能建立连接，于是放
    // 回新路径名目录的 i 节点，返回出错号。
776         dir = dir_namei(newname,&namelen,&basename, NULL);
777         if (!dir)
778                 return -EACCES;
779         if (!namelen) {
780                 iput(dir);
781                 return -EPERM;
782         }
783         if (!permission(dir,MAY_WRITE)) {
784                 iput(dir);
785                 return -EACCES;
786         }
    // 现在我们在目录指定设备上申请一个新的 i 节点，并设置该 i 节点模式为符号链接类型以及
    // 进程规定的模式屏蔽码。并且设置该 i 节点已修改标志。
787         if (!(inode = new_inode(dir->i_dev))) {
788                 iput(dir);
789                 return -ENOSPC;
790         }
791         inode->i_mode = S_IFLNK | (0777 & ~current->umask);
792         inode->i_dirt = 1;
    // 为了保存符号链接路径名字符串信息，我们需要为该 i 节点申请一个磁盘块，并让 i 节点的
    // 第 1 个直接块号 i_zone[0]等于得到的逻辑块号。然后置 i 节点已修改标志。如果申请失败
    // 则放回对应目录的 i 节点；复位新申请的 i 节点链接计数；放回该新的 i 节点，返回没有空
    // 间出错码退出。
793         if (!(inode->i_zone[0]=new_block(inode->i_dev))) {
794                 iput(dir);
795                 inode->i_nlinks--;
796                 iput(inode);
797                 return -ENOSPC;
798         }
799         inode->i_dirt = 1;
    // 然后从设备上读取新申请的磁盘块（目的是把对应块放到高速缓冲区中）。若出错，则放回
    // 对应目录的 i 节点；复位新申请的 i 节点链接计数；放回该新的 i 节点，返回没有空间出错
    // 码退出。
800         if (!(name_block=bread(inode->i_dev,inode->i_zone[0]))) {
801                 iput(dir);
802                 inode->i_nlinks--;
803                 iput(inode);
804                 return -ERROR;
```

```
805         }
    // 现在我们可以把符号链接名字符串放入这个盘块中了。盘块长度为 1024 字节，因此默认符号
    // 链接名长度最大也只能是 1024 字节。我们把用户空间中的符号链接名字符串复制到盘块所在
    // 的缓冲块中，并置缓冲块已修改标志。为防止用户提供的字符串没有以 null 结尾，我们在缓
    // 冲块数据区最后一个字节处放上一个 NULL。然后释放该缓冲块，并设置 i 节点对应文件中数
    // 据长度等于符号链接名字符串长度，并置 i 节点已修改标志。
806         i = 0;
807         while (i < 1023 && (c=get_fs_byte(oldname++)))
808                 name_block->b_data[i++] = c;
809         name_block->b_data[i] = 0;
810         name_block->b_dirt = 1;
811         brelse(name_block);
812         inode->i_size = i;
813         inode->i_dirt = 1;
    // 然后我们搜索一下路径名指定的符号链接文件名是否已经存在。若已经存在则不能创建同名
    // 目录项 i 节点。如果对应符号链接文件名已经存在，则释放包含该目录项的缓冲区块，复位
    // 新申请的 i 节点连接计数，并放回目录的 i 节点，返回文件已经存在的出错码退出。
814         bh = find_entry(&dir,basename,namelen,&de);
815         if (bh) {
816                 inode->i_nlinks--;
817                 iput(inode);
818                 brelse(bh);
819                 iput(dir);
820                 return -EEXIST;
821         }
    // 现在我们在指定目录中新添加一个目录项，用于存放新建符号链接文件名的 i 节点号和目录
    // 名。如果失败（包含该目录项的高速缓冲区指针为 NULL），则放回目录的 i 节点；所申请的
    // i 节点引用连接计数复位，并放回该 i 节点。返回出错码退出。
822         bh = add_entry(dir,basename,namelen,&de);
823         if (!bh) {
824                 inode->i_nlinks--;
825                 iput(inode);
826                 iput(dir);
827                 return -ENOSPC;
828         }
    // 最后令该新目录项的 i 节点字段等于新 i 节点号，并置高速缓冲块已修改标志，释放高速缓
    // 冲块，放回目录和新的 i 节点，最后返回 0（成功）。
829         de->inode = inode->i_num;
830         bh->b_dirt = 1;
831         brelse(bh);
832         iput(dir);
833         iput(inode);
834         return 0;
835 }
836
    //// 为文件建立一个文件名目录项。
    // 为一个已存在的文件创建一个新链接（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。
837 int sys_link(const char * oldname, const char * newname)
838 {
839         struct dir_entry * de;
840         struct m_inode * oldinode, * dir;
```

```
841         struct buffer_head * bh;
842         const char * basename;
843         int namelen;
844
    // 首先对原文件名进行有效性验证，它应该存在并且不是一个目录名。所以我们先取原文件路
    // 径名对应的 i 节点 oldinode。如果为 0，则表示出错，返回出错号。如果原路径名对应的是
    // 一个目录名，则放回该 i 节点，也返回出错号。
845         oldinode=namei(oldname);
846         if (!oldinode)
847                 return -ENOENT;
848         if (S_ISDIR(oldinode->i_mode)) {
849                 iput(oldinode);
850                 return -EPERM;
851         }
    // 然后查找新路径名的最顶层目录的 i 节点 dir，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，则放回原路径名的 i 节点，返回出错号。如果新路径名中不包括文件名，
    // 则放回原路径名 i 节点和新路径名目录的 i 节点，返回出错号。
852         dir = dir_namei(newname,&namelen,&basename, NULL);
853         if (!dir) {
854                 iput(oldinode);
855                 return -EACCES;
856         }
857         if (!namelen) {
858                 iput(oldinode);
859                 iput(dir);
860                 return -EPERM;
861         }
    // 我们不能跨设备建立硬链接。因此如果新路径名顶层目录的设备号与原路径名的设备号
    // 不一样，则放回新路径名目录的 i 节点和原路径名的 i 节点，返回出错号。另外，如果用户
    // 没有在新目录中写的权限，则也不能建立连接，于是放回新路径名目录的 i 节点和原路径名
    // 的 i 节点，返回出错号。
862         if (dir->i_dev != oldinode->i_dev) {
863                 iput(dir);
864                 iput(oldinode);
865                 return -EXDEV;
866         }
867         if (!permission(dir,MAY_WRITE)) {
868                 iput(dir);
869                 iput(oldinode);
870                 return -EACCES;
871         }
    // 现在查询该新路径名是否已经存在，如果存在则也不能建立链接。于是释放包含该已存在目
    // 录项的高速缓冲块，放回新路径名目录的 i 节点和原路径名的 i 节点，返回出错号。
872         bh = find_entry(&dir,basename,namelen,&de);
873         if (bh) {
874                 brelse(bh);
875                 iput(dir);
876                 iput(oldinode);
877                 return -EEXIST;
878         }
    // 现在所有条件都满足了，于是我们在新目录中添加一个目录项。若失败则放回该目录的 i 节
    // 点和原路径名的 i 节点，返回出错号。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缓冲块已修改标志，释放该缓冲块，放回目录的 i 节点。
```

```
879         bh = add_entry(dir,basename,namelen,&de);
880         if (!bh) {
881                 iput(dir);
882                 iput(oldinode);
883                 return -ENOSPC;
884         }
885         de->inode = oldinode->i_num;
886         bh->b_dirt = 1;
887         brelse(bh);
888         iput(dir);
    // 再将原节点的链接计数加 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
    // 放回原路径名的 i 节点，并返回 0（成功）。
889         oldinode->i_nlinks++;
890         oldinode->i_ctime = CURRENT_TIME;
891         oldinode->i_dirt = 1;
892         iput(oldinode);
893         return 0;
894 }
895
```