LCOV - code coverage report

Current view: directory - fs/ext4 - namei.c (source / functions)		Found	Hit	Coverage
Test: kernel_2_6_31_ext4_round_3.info	Lines:	1122	796	70.9 %
Date: 2009-10-24	Functions:	32	28	87.5 %

```
      1                :  /*
      2                :   *  linux/fs/ext4/namei.c
      3                :   *
      4                :   * Copyright (C) 1992, 1993, 1994, 1995
      5                :   * Remy Card (card@masi.ibp.fr)
      6                :   * Laboratoire MASI - Institut Blaise Pascal
      7                :   * Universite Pierre et Marie Curie (Paris VI)
      8                :   *
      9                :   *  from
     10                :   *
     11                :   *  linux/fs/minix/namei.c
     12                :   *
     13                :   *  Copyright (C) 1991, 1992  Linus Torvalds
     14                :   *
     15                :   *  Big-endian to little-endian byte-swapping/bitmaps by
     16                :   *        David S. Miller (davem@caip.rutgers.edu), 1995
     17                :   *  Directory entry file type support and forward compatibility hooks
     18                :   *      for B-tree directories by Theodore Ts'o (tytso@mit.edu), 1998
     19                :   *  Hash Tree Directory indexing (c)
     20                :   *      Daniel Phillips, 2001
     21                :   *  Hash Tree Directory indexing porting
     22                :   *      Christopher Li, 2002
     23                :   *  Hash Tree Directory indexing cleanup
     24                :   *      Theodore Ts'o, 2002
     25                :   */
     26                :
     27                :  #include <linux/fs.h>
     28                :  #include <linux/pagemap.h>
     29                :  #include <linux/jbd2.h>
     30                :  #include <linux/time.h>
     31                :  #include <linux/fcntl.h>
     32                :  #include <linux/stat.h>
     33                :  #include <linux/string.h>
     34                :  #include <linux/quotaops.h>
     35                :  #include <linux/buffer_head.h>
     36                :  #include <linux/bio.h>
     37                :  #include "ext4.h"
     38                :  #include "ext4_jbd2.h"
     39                :
     40                :  #include "xattr.h"
     41                :  #include "acl.h"
     42                :
     43                :  /*
     44                :   * define how far ahead to read directories while searching them.
     45                :   */
     46                :  #define NAMEI_RA_CHUNKS  2
     47                :  #define NAMEI_RA_BLOCKS  4
     48                :  #define NAMEI_RA_SIZE        (NAMEI_RA_CHUNKS * NAMEI_RA_BLOCKS)
     49                :  #define NAMEI_RA_INDEX(c,b)  (((c) * NAMEI_RA_BLOCKS) + (b))
     50                :
     51                :  static struct buffer_head *ext4_append(handle_t *handle,
     52                :                                         struct inode *inode,
     53                :                                         ext4_lblk_t *block, int *err)
     54           149916 :  {
     55                :          struct buffer_head *bh;
     56                :
     57           149916 :          *block = inode->i_size >> inode->i_sb->s_blocksize_bits;
     58                :
     59           149916 :          bh = ext4_bread(handle, inode, *block, 1, err);
     60           149916 :          if (bh) {
     61           149916 :                  inode->i_size += inode->i_sb->s_blocksize;
     62           149916 :                  EXT4_I(inode)->i_disksize = inode->i_size;
     63           149916 :                  *err = ext4_journal_get_write_access(handle, bh);
     64           149916 :                  if (*err) {
     65                0 :                          brelse(bh);
     66                0 :                          bh = NULL;
     67                :                  }
     68                :          }
     69           149916 :          return bh;
     70                :  }
     71                :
     72                :  #ifndef assert
     73                :  #define assert(test) J_ASSERT(test)
     74                :  #endif
```

```
:
: #ifdef DX_DEBUG
: #define dxtrace(command) command
: #else
: #define dxtrace(command)
: #endif
:
: struct fake_dirent
: {
:         __le32 inode;
:         __le16 rec_len;
:         u8 name_len;
:         u8 file_type;
: };
:
: struct dx_countlimit
: {
:         __le16 limit;
:         __le16 count;
: };
:
: struct dx_entry
: {
:         __le32 hash;
:         __le32 block;
: };
:
: /*
:  * dx_root_info is laid out so that if it should somehow get overlaid by a
:  * dirent the two low bits of the hash version will be zero.  Therefore, the
:  * hash version mod 4 should never be 0.  Sincerely, the paranoia department.
:  */
:
: struct dx_root
: {
:         struct fake_dirent dot;
:         char dot_name[4];
:         struct fake_dirent dotdot;
:         char dotdot_name[4];
:         struct dx_root_info
:         {
:                 __le32 reserved_zero;
:                 u8 hash_version;
:                 u8 info_length; /* 8 */
:                 u8 indirect_levels;
:                 u8 unused_flags;
:         }
:         info;
:         struct dx_entry entries[0];
: };
:
: struct dx_node
: {
:         struct fake_dirent fake;
:         struct dx_entry entries[0];
: };
:
:
: struct dx_frame
: {
:         struct buffer_head *bh;
:         struct dx_entry *entries;
:         struct dx_entry *at;
: };
:
: struct dx_map_entry
: {
:         u32 hash;
:         u16 offs;
:         u16 size;
: };
:
: static inline ext4_lblk_t dx_get_block(struct dx_entry *entry);
: static void dx_set_block(struct dx_entry *entry, ext4_lblk_t value);
: static inline unsigned dx_get_hash(struct dx_entry *entry);
: static void dx_set_hash(struct dx_entry *entry, unsigned value);
: static unsigned dx_get_count(struct dx_entry *entries);
: static unsigned dx_get_limit(struct dx_entry *entries);
: static void dx_set_count(struct dx_entry *entries, unsigned value);
: static void dx_set_limit(struct dx_entry *entries, unsigned value);
: static unsigned dx_root_limit(struct inode *dir, unsigned infosize);
: static unsigned dx_node_limit(struct inode *dir);
: static struct dx_frame *dx_probe(const struct qstr *d_name,
:                                  struct inode *dir,
:                                  struct dx_hash_info *hinfo,
:                                  struct dx_frame *frame,
:                                  int *err);
: static void dx_release(struct dx_frame *frames);
: static int dx_make_map(struct ext4_dir_entry_2 *de, unsigned blocksize,
:                        struct dx_hash_info *hinfo, struct dx_map_entry map[]);
```

```
   165                :  static void dx_sort_map(struct dx_map_entry *map, unsigned count);
   166                :  static struct ext4_dir_entry_2 *dx_move_dirents(char *from, char *to,
   167                :                 struct dx_map_entry *offsets, int count, unsigned blocksize);
   168                :  static struct ext4_dir_entry_2* dx_pack_dirents(char *base, unsigned blocksize);
   169                :  static void dx_insert_block(struct dx_frame *frame,
   170                :                                      u32 hash, ext4_lblk_t block);
   171                :  static int ext4_htree_next_block(struct inode *dir, __u32 hash,
   172                :                                   struct dx_frame *frame,
   173                :                                   struct dx_frame *frames,
   174                :                                   __u32 *start_hash);
   175                :  static struct buffer_head * ext4_dx_find_entry(struct inode *dir,
   176                :                 const struct qstr *d_name,
   177                :                 struct ext4_dir_entry_2 **res_dir,
   178                :                 int *err);
   179                :  static int ext4_dx_add_entry(handle_t *handle, struct dentry *dentry,
   180                :                               struct inode *inode);
   181                :
   182                :  unsigned int ext4_rec_len_from_disk(__le16 dlen, unsigned blocksize)
   183    -1921647156 :  {
   184           -1 :          unsigned len = le16_to_cpu(dlen);
   185                :
   186           -1 :          if (len == EXT4_MAX_REC_LEN || len == 0)
   187         1041 :                  return blocksize;
   188           -1 :          return (len & 65532) | ((len & 3) << 16);
   189                :  }
   190                :
   191                :  __le16 ext4_rec_len_to_disk(unsigned len, unsigned blocksize)
   192     34697152 :  {
   193     34697152 :          if ((len > blocksize) || (blocksize > (1 << 18)) || (len & 3))
   194            0 :                  BUG();
   195     34697152 :          if (len < 65536)
   196     34697152 :                  return cpu_to_le16(len);
   197            0 :          if (len == blocksize) {
   198            0 :                  if (blocksize == 65536)
   199            0 :                          return cpu_to_le16(EXT4_MAX_REC_LEN);
   200                :                  else
   201            0 :                          return cpu_to_le16(0);
   202                :          }
   203            0 :          return cpu_to_le16((len & 65532) | ((len >> 16) & 3));
   204                :  }
   205                :
   206                :  /*
   207                :   * p is at least 6 bytes before the end of page
   208                :   */
   209                :  static inline struct ext4_dir_entry_2 *
   210                :  ext4_next_entry(struct ext4_dir_entry_2 *p, unsigned long blocksize)
   211                :  {
   212    -1598630696 :          return (struct ext4_dir_entry_2 *)((char *)p +
   213                :                  ext4_rec_len_from_disk(p->rec_len, blocksize));
   214                :  }
   215                :
   216                :  /*
   217                :   * Future: use high four bits of block for coalesce-on-delete flags
   218                :   * Mask them off for now.
   219                :   */
   220                :
   221                :  static inline ext4_lblk_t dx_get_block(struct dx_entry *entry)
   222                :  {
   223     34235312 :          return le32_to_cpu(entry->block) & 0x00ffffff;
   224                :  }
   225                :
   226                :  static inline void dx_set_block(struct dx_entry *entry, ext4_lblk_t value)
   227                :  {
   228       149916 :          entry->block = cpu_to_le32(value);
   229                :  }
   230                :
   231                :  static inline unsigned dx_get_hash(struct dx_entry *entry)
   232                :  {
   233    177325809 :          return le32_to_cpu(entry->hash);
   234                :  }
   235                :
   236                :  static inline void dx_set_hash(struct dx_entry *entry, unsigned value)
   237                :  {
   238       147700 :          entry->hash = cpu_to_le32(value);
   239                :  }
   240                :
   241                :  static inline unsigned dx_get_count(struct dx_entry *entries)
   242                :  {
   243     43215110 :          return le16_to_cpu(((struct dx_countlimit *) entries)->count);
   244                :  }
   245                :
   246                :  static inline unsigned dx_get_limit(struct dx_entry *entries)
   247                :  {
   248     68747973 :          return le16_to_cpu(((struct dx_countlimit *) entries)->limit);
   249                :  }
   250                :
   251                :  static inline void dx_set_count(struct dx_entry *entries, unsigned value)
   252                :  {
   253       150358 :           ((struct dx_countlimit *) entries)->count = cpu_to_le16(value);
   254                :  }
```

```
    255                 :
    256                 : static inline void dx_set_limit(struct dx_entry *entries, unsigned value)
    257                 : {
    258              2658 :         ((struct dx_countlimit *) entries)->limit = cpu_to_le16(value);
    259                 : }
    260                 :
    261                 : static inline unsigned dx_root_limit(struct inode *dir, unsigned infosize)
    262                 : {
    263                 :         unsigned entry_space = dir->i_sb->s_blocksize - EXT4_DIR_REC_LEN(1) -
    264          24180547 :                 EXT4_DIR_REC_LEN(2) - infosize;
    265          24180547 :         return entry_space / sizeof(struct dx_entry);
    266                 : }
    267                 :
    268                 : static inline unsigned dx_node_limit(struct inode *dir)
    269                 : {
    270          10049480 :         unsigned entry_space = dir->i_sb->s_blocksize - EXT4_DIR_REC_LEN(0);
    271          10049480 :         return entry_space / sizeof(struct dx_entry);
    272                 : }
    273                 :
    274                 : /*
    275                 :  * Debug
    276                 :  */
    277                 : #ifdef DX_DEBUG
    278                 : static void dx_show_index(char * label, struct dx_entry *entries)
    279                 : {
    280                 :         int i, n = dx_get_count (entries);
    281                 :         printk(KERN_DEBUG "%s index ", label);
    282                 :         for (i = 0; i < n; i++) {
    283                 :                 printk("%x->%lu ", i ? dx_get_hash(entries + i) :
    284                 :                                 0, (unsigned long)dx_get_block(entries + i));
    285                 :         }
    286                 :         printk("\n");
    287                 : }
    288                 :
    289                 : struct stats
    290                 : {
    291                 :         unsigned names;
    292                 :         unsigned space;
    293                 :         unsigned bcount;
    294                 : };
    295                 :
    296                 : static struct stats dx_show_leaf(struct dx_hash_info *hinfo, struct ext4_dir_entry_2 *de,
    297                 :                                  int size, int show_names)
    298                 : {
    299                 :         unsigned names = 0, space = 0;
    300                 :         char *base = (char *) de;
    301                 :         struct dx_hash_info h = *hinfo;
    302                 :
    303                 :         printk("names: ");
    304                 :         while ((char *) de < base + size)
    305                 :         {
    306                 :                 if (de->inode)
    307                 :                 {
    308                 :                         if (show_names)
    309                 :                         {
    310                 :                                 int len = de->name_len;
    311                 :                                 char *name = de->name;
    312                 :                                 while (len--) printk("%c", *name++);
    313                 :                                 ext4fs_dirhash(de->name, de->name_len, &h);
    314                 :                                 printk(":%x.%u ", h.hash,
    315                 :                                        ((char *) de - base));
    316                 :                         }
    317                 :                         space += EXT4_DIR_REC_LEN(de->name_len);
    318                 :                         names++;
    319                 :                 }
    320                 :                 de = ext4_next_entry(de, size);
    321                 :         }
    322                 :         printk("(%i)\n", names);
    323                 :         return (struct stats) { names, space, 1 };
    324                 : }
    325                 :
    326                 : struct stats dx_show_entries(struct dx_hash_info *hinfo, struct inode *dir,
    327                 :                              struct dx_entry *entries, int levels)
    328                 : {
    329                 :         unsigned blocksize = dir->i_sb->s_blocksize;
    330                 :         unsigned count = dx_get_count(entries), names = 0, space = 0, i;
    331                 :         unsigned bcount = 0;
    332                 :         struct buffer_head *bh;
    333                 :         int err;
    334                 :         printk("%i indexed blocks...\n", count);
    335                 :         for (i = 0; i < count; i++, entries++)
    336                 :         {
    337                 :                 ext4_lblk_t block = dx_get_block(entries);
    338                 :                 ext4_lblk_t hash  = i ? dx_get_hash(entries): 0;
    339                 :                 u32 range = i < count - 1? (dx_get_hash(entries + 1) - hash): ~hash;
    340                 :                 struct stats stats;
    341                 :                 printk("%s%3u:%03u hash %8x/%8x ",levels?"":"   ", i, block, hash, range);
    342                 :                 if (!(bh = ext4_bread (NULL,dir, block, 0,&err))) continue;
    343                 :                 stats = levels?
    344                 :                    dx_show_entries(hinfo, dir, ((struct dx_node *) bh->b_data)->entries, levels - 1):
```

```
   345                :                    dx_show_leaf(hinfo, (struct ext4_dir_entry_2 *) bh->b_data, blocksize, 0);
   346                :                names += stats.names;
   347                :                space += stats.space;
   348                :                bcount += stats.bcount;
   349                :                brelse(bh);
   350                :        }
   351                :        if (bcount)
   352                :                printk(KERN_DEBUG "%snames %u, fullness %u (%u%%)\n",
   353                :                       levels ? "" : "   ", names, space/bcount,
   354                :                       (space/bcount)*100/blocksize);
   355                :        return (struct stats) { names, space, bcount};
   356                : }
   357                : #endif /* DX_DEBUG */
   358                :
   359                : /*
   360                :  * Probe for a directory leaf block to search.
   361                :  *
   362                :  * dx_probe can return ERR_BAD_DX_DIR, which means there was a format
   363                :  * error in the directory index, and the caller should fall back to
   364                :  * searching the directory normally.  The callers of dx_probe **MUST**
   365                :  * check for this error code, and make sure it never gets reflected
   366                :  * back to userspace.
   367                :  */
   368                : static struct dx_frame *
   369                : dx_probe(const struct qstr *d_name, struct inode *dir,
   370                :         struct dx_hash_info *hinfo, struct dx_frame *frame_in, int *err)
   371         24178779 : {
   372                :        unsigned count, indirect;
   373                :        struct dx_entry *at, *entries, *p, *q, *m;
   374                :        struct dx_root *root;
   375                :        struct buffer_head *bh;
   376         24178779 :        struct dx_frame *frame = frame_in;
   377                :        u32 hash;
   378                :
   379         24178779 :        frame->bh = NULL;
   380         24178779 :        if (!(bh = ext4_bread (NULL,dir, 0, 0, err)))
   381                0 :                goto fail;
   382         24178913 :        root = (struct dx_root *) bh->b_data;
   383         24178913 :        if (root->info.hash_version != DX_HASH_TEA &&
   384                :            root->info.hash_version != DX_HASH_HALF_MD4 &&
   385                :            root->info.hash_version != DX_HASH_LEGACY) {
   386                0 :                ext4_warning(dir->i_sb, __func__,
   387                :                             "Unrecognised inode hash code %d",
   388                :                             root->info.hash_version);
   389                0 :                brelse(bh);
   390                0 :                *err = ERR_BAD_DX_DIR;
   391                0 :                goto fail;
   392                :        }
   393         24178913 :        hinfo->hash_version = root->info.hash_version;
   394         24178913 :        if (hinfo->hash_version <= DX_HASH_TEA)
   395         48357818 :                hinfo->hash_version += EXT4_SB(dir->i_sb)->s_hash_unsigned;
   396         48357826 :        hinfo->seed = EXT4_SB(dir->i_sb)->s_hash_seed;
   397         24178913 :        if (d_name)
   398         24103957 :                ext4fs_dirhash(d_name->name, d_name->len, hinfo);
   399         24178351 :        hash = hinfo->hash;
   400                :
   401         24178351 :        if (root->info.unused_flags & 1) {
   402                0 :                ext4_warning(dir->i_sb, __func__,
   403                :                             "Unimplemented inode hash flags: %#06x",
   404                :                             root->info.unused_flags);
   405                0 :                brelse(bh);
   406                0 :                *err = ERR_BAD_DX_DIR;
   407                0 :                goto fail;
   408                :        }
   409                :
   410         24178351 :        if ((indirect = root->info.indirect_levels) > 1) {
   411                0 :                ext4_warning(dir->i_sb, __func__,
   412                :                             "Unimplemented inode hash depth: %#06x",
   413                :                             root->info.indirect_levels);
   414                0 :                brelse(bh);
   415                0 :                *err = ERR_BAD_DX_DIR;
   416                0 :                goto fail;
   417                :        }
   418                :
   419         24178351 :        entries = (struct dx_entry *) (((char *)&root->info) +
   420                :                                       root->info.info_length);
   421                :
   422         48356702 :        if (dx_get_limit(entries) != dx_root_limit(dir,
   423                :                                                   root->info.info_length)) {
   424                0 :                ext4_warning(dir->i_sb, __func__,
   425                :                             "dx entry: limit != root limit");
   426                0 :                brelse(bh);
   427                0 :                *err = ERR_BAD_DX_DIR;
   428                0 :                goto fail;
   429                :        }
   430                :
   431                :        dxtrace(printk("Look up %x", hash));
   432                :        while (1)
   433                :        {
   434         34227369 :                count = dx_get_count(entries);
```

```
    435  68454749 :                 if (!count || count > dx_get_limit(entries)) {
    436         0 :                         ext4_warning(dir->i_sb, __func__,
    437           :                                      "dx entry: no count or count > limit");
    438         0 :                         brelse(bh);
    439         0 :                         *err = ERR_BAD_DX_DIR;
    440         0 :                         goto fail2;
    441           :                 }
    442           :
    443  34228015 :                 p = entries + 1;
    444  34228015 :                 q = entries + count - 1;
    445 237745843 :                 while (p <= q)
    446           :                 {
    447 169289813 :                         m = p + (q - p)/2;
    448           :                         dxtrace(printk("."));
    449 169289813 :                         if (dx_get_hash(m) > hash)
    450  81767209 :                                 q = m - 1;
    451           :                         else
    452  87522604 :                                 p = m + 1;
    453           :                 }
    454           :
    455           :                 if (0) // linear search cross check
    456           :                 {
    457           :                         unsigned n = count - 1;
    458           :                         at = entries;
    459           :                         while (n--)
    460           :                         {
    461           :                                 dxtrace(printk(","));
    462           :                                 if (dx_get_hash(++at) > hash)
    463           :                                 {
    464           :                                         at--;
    465           :                                         break;
    466           :                                 }
    467           :                         }
    468           :                         assert (at == p - 1);
    469           :                 }
    470           :
    471  34228015 :                 at = p - 1;
    472           :                 dxtrace(printk(" %x->%u\n", at == entries? 0: dx_get_hash(at), dx_get_block(at)));
    473  34228015 :                 frame->bh = bh;
    474  34228015 :                 frame->entries = entries;
    475  34228015 :                 frame->at = at;
    476  34228015 :                 if (!indirect--) return frame;
    477  10049018 :                 if (!(bh = ext4_bread (NULL,dir, dx_get_block(at), 0, err)))
    478         0 :                         goto fail2;
    479  10049018 :                 at = entries = ((struct dx_node *) bh->b_data)->entries;
    480  10049018 :                 if (dx_get_limit(entries) != dx_node_limit (dir)) {
    481         0 :                         ext4_warning(dir->i_sb, __func__,
    482           :                                      "dx entry: limit != node limit");
    483         0 :                         brelse(bh);
    484         0 :                         *err = ERR_BAD_DX_DIR;
    485         0 :                         goto fail2;
    486           :                 }
    487  10049018 :                 frame++;
    488  10049018 :                 frame->bh = NULL;
    489  10049018 :         }
    490           : fail2:
    491         0 :         while (frame >= frame_in) {
    492         0 :                 brelse(frame->bh);
    493         0 :                 frame--;
    494           :         }
    495         0 : fail:
    496         0 :         if (*err == ERR_BAD_DX_DIR)
    497         0 :                 ext4_warning(dir->i_sb, __func__,
    498           :                              "Corrupt dir inode %ld, running e2fsck is "
    499           :                              "recommended.", dir->i_ino);
    500         0 :         return NULL;
    501           : }
    502           :
    503           : static void dx_release (struct dx_frame *frames)
    504  24181195 : {
    505  24181195 :         if (frames[0].bh == NULL)
    506         0 :                 return;
    507           :
    508  24181195 :         if (((struct dx_root *) frames[0].bh->b_data)->info.indirect_levels)
    509  10049038 :                 brelse(frames[1].bh);
    510  24181195 :         brelse(frames[0].bh);
    511           : }
    512           :
    513           : /*
    514           :  * This function increments the frame pointer to search the next leaf
    515           :  * block, and reads in the necessary intervening nodes if the search
    516           :  * should be necessary.  Whether or not the search is necessary is
    517           :  * controlled by the hash parameter.  If the hash value is even, then
    518           :  * the search is only continued if the next block starts with that
    519           :  * hash value.  This is used if we are searching for a specific file.
    520           :  *
    521           :  * If the hash value is HASH_NB_ALWAYS, then always go to the next block.
    522           :  *
    523           :  * This function returns 1 if the caller should continue to search,
    524           :  * or 0 if it should not.  If there is an error reading one of the
```

```
    525                 :  * index blocks, it will a negative error code.
    526                 :  *
    527                 :  * If start_hash is non-null, it will be filled in with the starting
    528                 :  * hash of the next page.
    529                 :  */
    530                 : static int ext4_htree_next_block(struct inode *dir, __u32 hash,
    531                 :                                  struct dx_frame *frame,
    532                 :                                  struct dx_frame *frames,
    533                 :                                  __u32 *start_hash)
    534         8684132 : {
    535                 :         struct dx_frame *p;
    536                 :         struct buffer_head *bh;
    537         8684132 :         int err, num_frames = 0;
    538                 :         __u32 bhash;
    539                 :
    540         8684132 :         p = frame;
    541                 :         /*
    542                 :          * Find the next leaf page by incrementing the frame pointer.
    543                 :          * If we run out of entries in the interior node, loop around and
    544                 :          * increment pointer in the parent node.  When we break out of
    545                 :          * this loop, num_frames indicates the number of interior
    546                 :          * nodes need to be read.
    547                 :          */
    548                 :         while (1) {
    549        17388090 :                 if (++(p->at) < p->entries + dx_get_count(p->entries))
    550         8035554 :                         break;
    551          658491 :                 if (p == frames)
    552          648578 :                         return 0;
    553            9913 :                 num_frames++;
    554            9913 :                 p--;
    555            9913 :         }
    556                 :
    557                 :         /*
    558                 :          * If the hash is 1, then continue only if the next page has a
    559                 :          * continuation hash of any value.  This is used for readdir
    560                 :          * handling.  Otherwise, check to see if the hash matches the
    561                 :          * desired contiuation hash.  If it doesn't, return since
    562                 :          * there's no point to read in the successive index pages.
    563                 :          */
    564        16071108 :         bhash = dx_get_hash(p->at);
    565         8035554 :         if (start_hash)
    566           82335 :                 *start_hash = bhash;
    567         8035554 :         if ((hash & 1) == 0) {
    568         7953219 :                 if ((bhash & ~1) != hash)
    569         7953217 :                         return 0;
    570                 :         }
    571                 :         /*
    572                 :          * If the hash is HASH_NB_ALWAYS, we always go to the next
    573                 :          * block so no check is necessary
    574                 :          */
    575           82802 :         while (num_frames--) {
    576             930 :                 if (!(bh = ext4_bread(NULL, dir, dx_get_block(p->at),
    577                 :                                       0, &err)))
    578               0 :                         return err; /* Failure */
    579             465 :                 p++;
    580             465 :                 brelse(p->bh);
    581             465 :                 p->bh = bh;
    582             465 :                 p->at = p->entries = ((struct dx_node *) bh->b_data)->entries;
    583                 :         }
    584           82337 :         return 1;
    585                 : }
    586                 :
    587                 :
    588                 : /*
    589                 :  * This function fills a red-black tree with information from a
    590                 :  * directory block.  It returns the number directory entries loaded
    591                 :  * into the tree.  If there is an error it is returned in err.
    592                 :  */
    593                 : static int htree_dirblock_to_tree(struct file *dir_file,
    594                 :                                   struct inode *dir, ext4_lblk_t block,
    595                 :                                   struct dx_hash_info *hinfo,
    596                 :                                   __u32 start_hash, __u32 start_minor_hash)
    597           82381 : {
    598                 :         struct buffer_head *bh;
    599                 :         struct ext4_dir_entry_2 *de, *top;
    600           82381 :         int err, count = 0;
    601                 :
    602                 :         dxtrace(printk(KERN_INFO "In htree dirblock_to_tree: block %lu\n",
    603                 :                                                         (unsigned long)block));
    604           82381 :         if (!(bh = ext4_bread (NULL, dir, block, 0, &err)))
    605               0 :                 return err;
    606                 :
    607           82381 :         de = (struct ext4_dir_entry_2 *) bh->b_data;
    608           82381 :         top = (struct ext4_dir_entry_2 *) ((char *) de +
    609                 :                                            dir->i_sb->s_blocksize -
    610                 :                                            EXT4_DIR_REC_LEN(0));
    611         3699555 :         for (; de < top; de = ext4_next_entry(de, dir->i_sb->s_blocksize)) {
    612         1808587 :                 if (!ext4_check_dir_entry("htree_dirblock_to_tree", dir, de, bh,
    613                 :                                           (block<<EXT4_BLOCK_SIZE_BITS(dir->i_sb))
    614                 :                                                 +((char *)de - bh->b_data))) {
```

```
    615                 :                         /* On error, skip the f_pos to the next block. */
    616               0 :                         dir_file->f_pos = (dir_file->f_pos |
    617                 :                                            (dir->i_sb->s_blocksize - 1)) + 1;
    618               0 :                         brelse(bh);
    619               0 :                         return count;
    620                 :                 }
    621         1808587 :                 ext4fs_dirhash(de->name, de->name_len, hinfo);
    622         1808587 :                 if ((hinfo->hash < start_hash) ||
    623                 :                     ((hinfo->hash == start_hash) &&
    624                 :                      (hinfo->minor_hash < start_minor_hash)))
    625                 :                         continue;
    626         1795373 :                 if (de->inode == 0)
    627            7353 :                         continue;
    628         1788020 :                 if ((err = ext4_htree_store_dirent(dir_file,
    629                 :                                    hinfo->hash, hinfo->minor_hash, de)) != 0) {
    630               0 :                         brelse(bh);
    631               0 :                         return err;
    632                 :                 }
    633         1788020 :                 count++;
    634                 :         }
    635           82381 :         brelse(bh);
    636           82381 :         return count;
    637                 : }
    638                 :
    639                 :
    640                 : /*
    641                 :  * This function fills a red-black tree with information from a
    642                 :  * directory.  We start scanning the directory in hash order, starting
    643                 :  * at start_hash and start_minor_hash.
    644                 :  *
    645                 :  * This function returns the number of entries inserted into the tree,
    646                 :  * or a negative error code.
    647                 :  */
    648                 : int ext4_htree_fill_tree(struct file *dir_file, __u32 start_hash,
    649                 :                          __u32 start_minor_hash, __u32 *next_hash)
    650           75038 : {
    651                 :         struct dx_hash_info hinfo;
    652                 :         struct ext4_dir_entry_2 *de;
    653                 :         struct dx_frame frames[2], *frame;
    654                 :         struct inode *dir;
    655                 :         ext4_lblk_t block;
    656           75038 :         int count = 0;
    657                 :         int ret, err;
    658                 :         __u32 hashval;
    659                 :
    660                 :         dxtrace(printk(KERN_DEBUG "In htree_fill_tree, start hash: %x:%x\n",
    661                 :                        start_hash, start_minor_hash));
    662           75038 :         dir = dir_file->f_path.dentry->d_inode;
    663           75038 :         if (!(EXT4_I(dir)->i_flags & EXT4_INDEX_FL)) {
    664              48 :                 hinfo.hash_version = EXT4_SB(dir->i_sb)->s_def_hash_version;
    665              24 :                 if (hinfo.hash_version <= DX_HASH_TEA)
    666              48 :                         hinfo.hash_version +=
    667                 :                                 EXT4_SB(dir->i_sb)->s_hash_unsigned;
    668              48 :                 hinfo.seed = EXT4_SB(dir->i_sb)->s_hash_seed;
    669              24 :                 count = htree_dirblock_to_tree(dir_file, dir, 0, &hinfo,
    670                 :                                                start_hash, start_minor_hash);
    671              24 :                 *next_hash = ~0;
    672              24 :                 return count;
    673                 :         }
    674           75014 :         hinfo.hash = start_hash;
    675           75014 :         hinfo.minor_hash = 0;
    676           75014 :         frame = dx_probe(NULL, dir, &hinfo, frames, &err);
    677           75014 :         if (!frame)
    678               0 :                 return err;
    679                 :
    680                 :         /* Add '.' and '..' from the htree header */
    681           75014 :         if (!start_hash && !start_minor_hash) {
    682              21 :                 de = (struct ext4_dir_entry_2 *) frames[0].bh->b_data;
    683              21 :                 if ((err = ext4_htree_store_dirent(dir_file, 0, 0, de)) != 0)
    684               0 :                         goto errout;
    685              21 :                 count++;
    686                 :         }
    687           75014 :         if (start_hash < 2 || (start_hash ==2 && start_minor_hash==0)) {
    688              21 :                 de = (struct ext4_dir_entry_2 *) frames[0].bh->b_data;
    689              42 :                 de = ext4_next_entry(de, dir->i_sb->s_blocksize);
    690              21 :                 if ((err = ext4_htree_store_dirent(dir_file, 2, 0, de)) != 0)
    691               0 :                         goto errout;
    692              21 :                 count++;
    693                 :         }
    694                 :
    695                 :         while (1) {
    696          164714 :                 block = dx_get_block(frame->at);
    697           82357 :                 ret = htree_dirblock_to_tree(dir_file, dir, block, &hinfo,
    698                 :                                              start_hash, start_minor_hash);
    699           82357 :                 if (ret < 0) {
    700               0 :                         err = ret;
    701               0 :                         goto errout;
    702                 :                 }
    703           82357 :                 count += ret;
    704           82357 :                 hashval = ~0;
```

```
    705      82357 :                 ret = ext4_htree_next_block(dir, HASH_NB_ALWAYS,
    706            :                                             frame, frames, &hashval);
    707      82357 :                 *next_hash = hashval;
    708      82357 :                 if (ret < 0) {
    709          0 :                         err = ret;
    710          0 :                         goto errout;
    711            :                 }
    712            :                 /*
    713            :                  * Stop if:  (a) there are no more entries, or
    714            :                  * (b) we have inserted at least one entry and the
    715            :                  * next hash value is not a continuation
    716            :                  */
    717      82357 :                 if ((ret == 0) ||
    718            :                     (count && ((hashval & 1) == 0)))
    719            :                         break;
    720            :         }
    721      75014 :         dx_release(frames);
    722            :         dxtrace(printk(KERN_DEBUG "Fill tree: returned %d entries, "
    723            :                        "next hash: %x\n", count, *next_hash));
    724      75014 :         return count;
    725          0 : errout:
    726          0 :         dx_release(frames);
    727          0 :         return (err);
    728            : }
    729            :
    730            :
    731            : /*
    732            :  * Directory block splitting, compacting
    733            :  */
    734            :
    735            : /*
    736            :  * Create map of hash values, offsets, and sizes, stored at end of block.
    737            :  * Returns number of entries mapped.
    738            :  */
    739            : static int dx_make_map(struct ext4_dir_entry_2 *de, unsigned blocksize,
    740            :                        struct dx_hash_info *hinfo,
    741            :                        struct dx_map_entry *map_tail)
    742            : {
    743     147258 :         int count = 0;
    744     147258 :         char *base = (char *) de;
    745     147258 :         struct dx_hash_info h = *hinfo;
    746            :
    747   12924735 :         while ((char *) de < base + blocksize) {
    748   12777477 :                 if (de->name_len && de->inode) {
    749   12777477 :                         ext4fs_dirhash(de->name, de->name_len, &h);
    750   12777477 :                         map_tail--;
    751   12777477 :                         map_tail->hash = h.hash;
    752   12777477 :                         map_tail->offs = ((char *) de - base)>>2;
    753   12777477 :                         map_tail->size = le16_to_cpu(de->rec_len);
    754   12777477 :                         count++;
    755            :                         cond_resched();
    756            :                 }
    757            :                 /* XXX: do we need to check rec_len == 0 case? -Chris */
    758   12777477 :                 de = ext4_next_entry(de, blocksize);
    759            :         }
    760     147258 :         return count;
    761            : }
    762            :
    763            : /* Sort map by hash value */
    764            : static void dx_sort_map (struct dx_map_entry *map, unsigned count)
    765            : {
    766     147258 :         struct dx_map_entry *p, *q, *top = map + count - 1;
    767            :         int more;
    768            :         /* Combsort until bubble sort doesn't suck */
    769    1558871 :         while (count > 2) {
    770    1411613 :                 count = count*10/13;
    771    1411613 :                 if (count - 9 < 2) /* 9, 10 -> 11 */
    772      36919 :                         count = 11;
    773  144370620 :                 for (p = top, q = p - count; q >= map; p--, q--)
    774  142959007 :                         if (p->hash < q->hash)
    775   31029840 :                                 swap(*p, *q);
    776            :         }
    777            :         /* Garden variety bubble sort */
    778            :         do {
    779     313596 :                 more = 0;
    780     313596 :                 q = top;
    781   29964293 :                 while (q-- > map) {
    782   29650697 :                         if (q[1].hash >= q[0].hash)
    783            :                                 continue;
    784    3298717 :                         swap(*(q+1), *q);
    785    3298717 :                         more = 1;
    786            :                 }
    787     313596 :         } while(more);
    788            : }
    789            :
    790            : static void dx_insert_block(struct dx_frame *frame, u32 hash, ext4_lblk_t block)
    791     147700 : {
    792     147700 :         struct dx_entry *entries = frame->entries;
    793     147700 :         struct dx_entry *old = frame->at, *new = old + 1;
    794     147700 :         int count = dx_get_count(entries);
```

```
    795                 :
    796          295400 :         assert(count < dx_get_limit(entries));
    797          147700 :         assert(old < entries + count);
    798          147700 :         memmove(new + 1, new, (char *)(entries + count) - (char *)(new));
    799                 :         dx_set_hash(new, hash);
    800                 :         dx_set_block(new, block);
    801          147700 :         dx_set_count(entries, count + 1);
    802          147700 : }
    803                 :
    804                 : static void ext4_update_dx_flag(struct inode *inode)
    805                 : {
    806        23349216 :         if (!EXT4_HAS_COMPAT_FEATURE(inode->i_sb,
    807                 :                                      EXT4_FEATURE_COMPAT_DIR_INDEX))
    808               0 :                 EXT4_I(inode)->i_flags &= ~EXT4_INDEX_FL;
    809                 : }
    810                 :
    811                 : /*
    812                 :  * NOTE! unlike strncmp, ext4_match returns 1 for success, 0 for failure.
    813                 :  *
    814                 :  * `len <= EXT4_NAME_LEN' is guaranteed by caller.
    815                 :  * `de != NULL' is guaranteed by caller.
    816                 :  */
    817                 : static inline int ext4_match (int len, const char * const name,
    818                 :                               struct ext4_dir_entry_2 * de)
    819                 : {
    820     -1927426568 :         if (len != de->name_len)
    821      804238993 :                 return 0;
    822     1563301735 :         if (!de->inode)
    823         885246 :                 return 0;
    824     1562416489 :         return !memcmp(name, de->name, len);
    825                 : }
    826                 :
    827                 : /*
    828                 :  * Returns 0 if not found, -1 on failure, and 1 on success
    829                 :  */
    830                 : static inline int search_dirblock(struct buffer_head *bh,
    831                 :                                   struct inode *dir,
    832                 :                                   const struct qstr *d_name,
    833                 :                                   unsigned int offset,
    834                 :                                   struct ext4_dir_entry_2 ** res_dir)
    835                 : {
    836                 :         struct ext4_dir_entry_2 * de;
    837                 :         char * dlimit;
    838                 :         int de_len;
    839          391613 :         const char *name = d_name->name;
    840          391613 :         int namelen = d_name->len;
    841                 :
    842          391613 :         de = (struct ext4_dir_entry_2 *) bh->b_data;
    843          391613 :         dlimit = bh->b_data + dir->i_sb->s_blocksize;
    844        39411126 :         while ((char *) de < dlimit) {
    845                 :                 /* this code is executed quadratically often */
    846                 :                 /* do minimal checking `by hand' */
    847                 :
    848        78039858 :                 if ((char *) de + namelen <= dlimit &&
    849                 :                     ext4_match (namelen, name, de)) {
    850                 :                         /* found a match - just to be sure, do a full check */
    851             416 :                         if (!ext4_check_dir_entry("ext4_find_entry",
    852                 :                                                   dir, de, bh, offset))
    853               0 :                                 return -1;
    854             416 :                         *res_dir = de;
    855             416 :                         return 1;
    856                 :                 }
    857                 :                 /* prevent looping on a bad block */
    858        78039026 :                 de_len = ext4_rec_len_from_disk(de->rec_len,
    859                 :                                                 dir->i_sb->s_blocksize);
    860        39019513 :                 if (de_len <= 0)
    861               0 :                         return -1;
    862        39019513 :                 offset += de_len;
    863        39019513 :                 de = (struct ext4_dir_entry_2 *) ((char *) de + de_len);
    864                 :         }
    865          391197 :         return 0;
    866                 : }
    867                 :
    868                 :
    869                 : /*
    870                 :  *      ext4_find_entry()
    871                 :  *
    872                 :  * finds an entry in the specified directory with the wanted name. It
    873                 :  * returns the cache buffer in which the entry was found, and the entry
    874                 :  * itself (as a parameter - res_dir). It does NOT read the inode of the
    875                 :  * entry - you'll have to do that yourself if you want to.
    876                 :  *
    877                 :  * The returned buffer_head has ->b_count elevated.  The caller is expected
    878                 :  * to brelse() it when appropriate.
    879                 :  */
    880                 : static struct buffer_head * ext4_find_entry (struct inode *dir,
    881                 :                                              const struct qstr *d_name,
    882                 :                                              struct ext4_dir_entry_2 ** res_dir)
    883        15574501 : {
    884                 :         struct super_block *sb;
```

```
    885                 :          struct buffer_head *bh_use[NAMEI_RA_SIZE];
    886          15574501 :          struct buffer_head *bh, *ret = NULL;
    887                 :          ext4_lblk_t start, block, b;
    888          15574501 :          int ra_max = 0;         /* Number of bh's in the readahead
    889                 :                                     buffer, bh_use[] */
    890          15574501 :          int ra_ptr = 0;         /* Current index into readahead
    891                 :                                     buffer */
    892          15574501 :          int num = 0;
    893                 :          ext4_lblk_t  nblocks;
    894                 :          int i, err;
    895                 :          int namelen;
    896                 :
    897          15574501 :          *res_dir = NULL;
    898          15574501 :          sb = dir->i_sb;
    899          15574501 :          namelen = d_name->len;
    900          15574501 :          if (namelen > EXT4_NAME_LEN)
    901                 0 :                  return NULL;
    902          46723486 :          if (is_dx(dir)) {
    903          15182928 :                  bh = ext4_dx_find_entry(dir, d_name, res_dir, &err);
    904                 :                  /*
    905                 :                   * On success, or if the error was file not found,
    906                 :                   * return.  Otherwise, fall back to doing a search the
    907                 :                   * old fashioned way.
    908                 :                   */
    909          15182938 :                  if (bh || (err != ERR_BAD_DX_DIR))
    910          15182898 :                          return bh;
    911                 :                  dxtrace(printk(KERN_DEBUG "ext4_find_entry: dx failed, "
    912                 :                                 "falling back\n"));
    913                 :          }
    914            391613 :          nblocks = dir->i_size >> EXT4_BLOCK_SIZE_BITS(sb);
    915            391613 :          start = EXT4_I(dir)->i_dir_start_lookup;
    916            391613 :          if (start >= nblocks)
    917                 0 :                  start = 0;
    918            391613 :          block = start;
    919            391613 : restart:
    920                 :          do {
    921                 :                  /*
    922                 :                   * We deal with the read-ahead logic here.
    923                 :                   */
    924            391613 :                  if (ra_ptr >= ra_max) {
    925                 :                          /* Refill the readahead buffer */
    926            391613 :                          ra_ptr = 0;
    927            391613 :                          b = block;
    928            783226 :                          for (ra_max = 0; ra_max < NAMEI_RA_SIZE; ra_max++) {
    929                 :                                  /*
    930                 :                                   * Terminate if we reach the end of the
    931                 :                                   * directory and must wrap, or if our
    932                 :                                   * search has finished at this block.
    933                 :                                   */
    934            783226 :                                  if (b >= nblocks || (num && block == start)) {
    935            391613 :                                          bh_use[ra_max] = NULL;
    936            391613 :                                          break;
    937                 :                                  }
    938            391613 :                                  num++;
    939            391613 :                                  bh = ext4_getblk(NULL, dir, b++, 0, &err);
    940            391613 :                                  bh_use[ra_max] = bh;
    941            391613 :                                  if (bh)
    942            391613 :                                          ll_rw_block(READ_META, 1, &bh);
    943                 :                          }
    944                 :                  }
    945            391613 :                  if ((bh = bh_use[ra_ptr++]) == NULL)
    946                 0 :                          goto next;
    947            391613 :                  wait_on_buffer(bh);
    948            783226 :                  if (!buffer_uptodate(bh)) {
    949                 :                          /* read error, skip block & hope for the best */
    950                 0 :                          ext4_error(sb, __func__, "reading directory #%lu "
    951                 :                                     "offset %lu", dir->i_ino,
    952                 :                                     (unsigned long)block);
    953                 0 :                          brelse(bh);
    954                 0 :                          goto next;
    955                 :                  }
    956            783226 :                  i = search_dirblock(bh, dir, d_name,
    957                 :                              block << EXT4_BLOCK_SIZE_BITS(sb), res_dir);
    958            391613 :                  if (i == 1) {
    959               416 :                          EXT4_I(dir)->i_dir_start_lookup = block;
    960               416 :                          ret = bh;
    961               416 :                          goto cleanup_and_exit;
    962                 :                  } else {
    963            391197 :                          brelse(bh);
    964            391197 :                          if (i < 0)
    965                 0 :                                  goto cleanup_and_exit;
    966                 :                  }
    967            391197 :          next:
    968            391197 :                  if (++block >= nblocks)
    969            391197 :                          block = 0;
    970            391197 :          } while (block != start);
    971                 :
    972                 :          /*
    973                 :           * If the directory has grown while we were searching, then
    974                 :           * search the last part of the directory before giving up.
```

```
    975                :          */
    976          391197 :         block = nblocks;
    977          391197 :         nblocks = dir->i_size >> EXT4_BLOCK_SIZE_BITS(sb);
    978          391197 :         if (block < nblocks) {
    979                0 :                 start = 0;
    980                0 :                 goto restart;
    981                :         }
    982                :
    983                : cleanup_and_exit:
    984                :         /* Clean up the read-ahead blocks */
    985                0 :         for (; ra_ptr < ra_max; ra_ptr++)
    986                0 :                 brelse(bh_use[ra_ptr]);
    987          391613 :         return ret;
    988                : }
    989                :
    990                : static struct buffer_head * ext4_dx_find_entry(struct inode *dir, const struct qstr *d_name,
    991                :                                struct ext4_dir_entry_2 **res_dir, int *err)
    992        15182920 : {
    993                :         struct super_block * sb;
    994                :         struct dx_hash_info     hinfo;
    995                :         u32 hash;
    996                :         struct dx_frame frames[2], *frame;
    997                :         struct ext4_dir_entry_2 *de, *top;
    998                :         struct buffer_head *bh;
    999                :         ext4_lblk_t block;
   1000                :         int retval;
   1001        15182920 :         int namelen = d_name->len;
   1002        15182920 :         const u8 *name = d_name->name;
   1003                :
   1004        15182920 :         sb = dir->i_sb;
   1005                :         /* NFS may look up ".." - look at dx_root directory block */
   1006        15182920 :         if (namelen > 2 || name[0] != '.'||(name[1] != '.' && name[1] != '\0')){
   1007        15182920 :                 if (!(frame = dx_probe(d_name, dir, &hinfo, frames, err)))
   1008                0 :                         return NULL;
   1009                :         } else {
   1010                0 :                 frame = frames;
   1011                0 :                 frame->bh = NULL;                       /* for dx_release() */
   1012                0 :                 frame->at = (struct dx_entry *)frames;       /* hack for zero entry*/
   1013                0 :                 dx_set_block(frame->at, 0);          /* dx_root block is 0 */
   1014                :         }
   1015        15182408 :         hash = hinfo.hash;
   1016                :         do {
   1017        30364820 :                 block = dx_get_block(frame->at);
   1018        15182410 :                 if (!(bh = ext4_bread (NULL,dir, block, 0, err)))
   1019                0 :                         goto errout;
   1020        15183092 :                 de = (struct ext4_dir_entry_2 *) bh->b_data;
   1021        15183092 :                 top = (struct ext4_dir_entry_2 *) ((char *) de + sb->s_blocksize -
   1022                :                                                    EXT4_DIR_REC_LEN(0));
   1023   -1725615774 :                 for (; de < top; de = ext4_next_entry(de, sb->s_blocksize)) {
   1024                :                         int off = (block << EXT4_BLOCK_SIZE_BITS(sb))
   1025    1283665532 :                                   + ((char *) de - bh->b_data);
   1026                :
   1027    1283665532 :                         if (!ext4_check_dir_entry(__func__, dir, de, bh, off)) {
   1028                0 :                                 brelse(bh);
   1029                0 :                                 *err = ERR_BAD_DX_DIR;
   1030                0 :                                 goto errout;
   1031                :                         }
   1032                :
   1033   -1727638431 :                         if (ext4_match(namelen, name, de)) {
   1034        6581172 :                                 *res_dir = de;
   1035        6581172 :                                 dx_release(frames);
   1036        6581163 :                                 return bh;
   1037                :                         }
   1038                :                 }
   1039        8601775 :                 brelse(bh);
   1040                :                 /* Check to see if we should continue to search */
   1041        8601775 :                 retval = ext4_htree_next_block(dir, hash, frame,
   1042                :                                                frames, NULL);
   1043        8601775 :                 if (retval < 0) {
   1044                0 :                         ext4_warning(sb, __func__,
   1045                :                              "error reading index page in directory #%lu",
   1046                :                              dir->i_ino);
   1047                0 :                         *err = retval;
   1048                0 :                         goto errout;
   1049                :                 }
   1050        8601775 :         } while (retval == 1);
   1051                :
   1052        8601773 :         *err = -ENOENT;
   1053        8601773 : errout:
   1054                :         dxtrace(printk(KERN_DEBUG "%s not found\n", name));
   1055        8601773 :         dx_release (frames);
   1056        8601773 :         return NULL;
   1057                : }
   1058                :
   1059                : static struct dentry *ext4_lookup(struct inode *dir, struct dentry *dentry, struct nameidata *nd)
   1060        13910171 : {
   1061                :         struct inode *inode;
   1062                :         struct ext4_dir_entry_2 *de;
   1063                :         struct buffer_head *bh;
   1064                :
```

```
1065  13910171 :          if (dentry->d_name.len > EXT4_NAME_LEN)
1066         0 :                  return ERR_PTR(-ENAMETOOLONG);
1067           :
1068  13910171 :          bh = ext4_find_entry(dir, &dentry->d_name, &de);
1069  13910142 :          inode = NULL;
1070  13910142 :          if (bh) {
1071   4917193 :                  __u32 ino = le32_to_cpu(de->inode);
1072   4917193 :                  brelse(bh);
1073   9834418 :                  if (!ext4_valid_inum(dir->i_sb, ino)) {
1074         0 :                          ext4_error(dir->i_sb, "ext4_lookup",
1075           :                                     "bad inode number: %u", ino);
1076         0 :                          return ERR_PTR(-EIO);
1077           :                  }
1078   4917209 :                  inode = ext4_iget(dir->i_sb, ino);
1079   4917216 :                  if (unlikely(IS_ERR(inode))) {
1080         0 :                          if (PTR_ERR(inode) == -ESTALE) {
1081         0 :                                  ext4_error(dir->i_sb, __func__,
1082           :                                             "deleted inode referenced: %u",
1083           :                                             ino);
1084         0 :                                  return ERR_PTR(-EIO);
1085           :                          } else {
1086         0 :                                  return ERR_CAST(inode);
1087           :                          }
1088           :                  }
1089           :          }
1090  13910164 :          return d_splice_alias(inode, dentry);
1091           : }
1092           :
1093           :
1094           : struct dentry *ext4_get_parent(struct dentry *child)
1095         0 : {
1096           :          __u32 ino;
1097           :          struct inode *inode;
1098           :          static const struct qstr dotdot = {
1099           :                  .name = "..",
1100           :                  .len = 2,
1101           :          };
1102           :          struct ext4_dir_entry_2 * de;
1103           :          struct buffer_head *bh;
1104           :
1105         0 :          bh = ext4_find_entry(child->d_inode, &dotdot, &de);
1106         0 :          inode = NULL;
1107         0 :          if (!bh)
1108         0 :                  return ERR_PTR(-ENOENT);
1109         0 :          ino = le32_to_cpu(de->inode);
1110         0 :          brelse(bh);
1111           :
1112         0 :          if (!ext4_valid_inum(child->d_inode->i_sb, ino)) {
1113         0 :                  ext4_error(child->d_inode->i_sb, "ext4_get_parent",
1114           :                             "bad inode number: %u", ino);
1115         0 :                  return ERR_PTR(-EIO);
1116           :          }
1117           :
1118         0 :          return d_obtain_alias(ext4_iget(child->d_inode->i_sb, ino));
1119           : }
1120           :
1121           : #define S_SHIFT 12
1122           : static unsigned char ext4_type_by_mode[S_IFMT >> S_SHIFT] = {
1123           :          [S_IFREG >> S_SHIFT]      = EXT4_FT_REG_FILE,
1124           :          [S_IFDIR >> S_SHIFT]      = EXT4_FT_DIR,
1125           :          [S_IFCHR >> S_SHIFT]      = EXT4_FT_CHRDEV,
1126           :          [S_IFBLK >> S_SHIFT]      = EXT4_FT_BLKDEV,
1127           :          [S_IFIFO >> S_SHIFT]      = EXT4_FT_FIFO,
1128           :          [S_IFSOCK >> S_SHIFT]     = EXT4_FT_SOCK,
1129           :          [S_IFLNK >> S_SHIFT]      = EXT4_FT_SYMLINK,
1130           : };
1131           :
1132           : static inline void ext4_set_de_type(struct super_block *sb,
1133           :                                     struct ext4_dir_entry_2 *de,
1134           :                                     umode_t mode) {
1135  10707909 :          if (EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_FILETYPE))
1136  10707909 :                  de->file_type = ext4_type_by_mode[(mode & S_IFMT)>>S_SHIFT];
1137           : }
1138           :
1139           : /*
1140           :  * Move count entries from end of map between two memory locations.
1141           :  * Returns pointer to last entry moved.
1142           :  */
1143           : static struct ext4_dir_entry_2 *
1144           : dx_move_dirents(char *from, char *to, struct dx_map_entry *map, int count,
1145           :                 unsigned blocksize)
1146    147258 : {
1147    147258 :          unsigned rec_len = 0;
1148           :
1149   6688020 :          while (count--) {
1150           :                  struct ext4_dir_entry_2 *de = (struct ext4_dir_entry_2 *)
1151   6393504 :                                                (from + (map->offs<<2));
1152   6393504 :                  rec_len = EXT4_DIR_REC_LEN(de->name_len);
1153  12787008 :                  memcpy (to, de, rec_len);
1154   6393504 :                  ((struct ext4_dir_entry_2 *) to)->rec_len =
```

```
1155                :                                   ext4_rec_len_to_disk(rec_len, blocksize);
1156        6393504 :                 de->inode = 0;
1157        6393504 :                 map++;
1158        6393504 :                 to += rec_len;
1159                :         }
1160         147258 :         return (struct ext4_dir_entry_2 *) (to - rec_len);
1161                : }
1162                :
1163                : /*
1164                :  * Compact each dir entry in the range to the minimal rec_len.
1165                :  * Returns pointer to last entry in range.
1166                :  */
1167                : static struct ext4_dir_entry_2* dx_pack_dirents(char *base, unsigned blocksize)
1168                : {
1169         147258 :         struct ext4_dir_entry_2 *next, *to, *prev, *de = (struct ext4_dir_entry_2 *) base;
1170         147258 :         unsigned rec_len = 0;
1171                :
1172         147258 :         prev = to = de;
1173       12924735 :         while ((char*)de < base + blocksize) {
1174       12777477 :                 next = ext4_next_entry(de, blocksize);
1175       12777477 :                 if (de->inode && de->name_len) {
1176        6383973 :                         rec_len = EXT4_DIR_REC_LEN(de->name_len);
1177        6383973 :                         if (de > to)
1178        4218065 :                                 memmove(to, de, rec_len);
1179        6383973 :                         to->rec_len = ext4_rec_len_to_disk(rec_len, blocksize);
1180        6383973 :                         prev = to;
1181        6383973 :                         to = (struct ext4_dir_entry_2 *) (((char *) to) + rec_len);
1182                :                 }
1183       12777477 :                 de = next;
1184                :         }
1185         147258 :         return prev;
1186                : }
1187                :
1188                : /*
1189                :  * Split a full leaf block to make room for a new dir entry.
1190                :  * Allocate a new block, and move entries so that they are approx. equally full.
1191                :  * Returns pointer to de in block into which the new entry will be inserted.
1192                :  */
1193                : static struct ext4_dir_entry_2 *do_split(handle_t *handle, struct inode *dir,
1194                :                         struct buffer_head **bh,struct dx_frame *frame,
1195                :                         struct dx_hash_info *hinfo, int *error)
1196         147258 : {
1197         147258 :         unsigned blocksize = dir->i_sb->s_blocksize;
1198                :         unsigned count, continued;
1199                :         struct buffer_head *bh2;
1200                :         ext4_lblk_t newblock;
1201                :         u32 hash2;
1202                :         struct dx_map_entry *map;
1203         147258 :         char *data1 = (*bh)->b_data, *data2;
1204                :         unsigned split, move, size;
1205         147258 :         struct ext4_dir_entry_2 *de = NULL, *de2;
1206         147258 :         int     err = 0, i;
1207                :
1208         147258 :         bh2 = ext4_append (handle, dir, &newblock, &err);
1209         147258 :         if (!(bh2)) {
1210              0 :                 brelse(*bh);
1211              0 :                 *bh = NULL;
1212              0 :                 goto errout;
1213                :         }
1214                :
1215                :         BUFFER_TRACE(*bh, "get_write_access");
1216         147258 :         err = ext4_journal_get_write_access(handle, *bh);
1217         147258 :         if (err)
1218              0 :                 goto journal_error;
1219                :
1220                :         BUFFER_TRACE(frame->bh, "get_write_access");
1221         147258 :         err = ext4_journal_get_write_access(handle, frame->bh);
1222         147258 :         if (err)
1223              0 :                 goto journal_error;
1224                :
1225         147258 :         data2 = bh2->b_data;
1226                :
1227                :         /* create map in the end of data2 block */
1228         147258 :         map = (struct dx_map_entry *) (data2 + blocksize);
1229         294516 :         count = dx_make_map((struct ext4_dir_entry_2 *) data1,
1230                :                             blocksize, hinfo, map);
1231         147258 :         map -= count;
1232                :         dx_sort_map(map, count);
1233                :         /* Split the existing block in the middle, size-wise */
1234         147258 :         size = 0;
1235         147258 :         move = 0;
1236        6540762 :         for (i = count-1; i >= 0; i--) {
1237                :                 /* is more than half of this entry in 2nd half of the block? */
1238        6540762 :                 if (size + map[i].size/2 > blocksize/2)
1239         147258 :                         break;
1240        6393504 :                 size += map[i].size;
1241        6393504 :                 move++;
1242                :         }
1243                :         /* map index at which we will split */
1244         147258 :         split = count - move;
```

```
1245          147258 :          hash2 = map[split].hash;
1246          147258 :          continued = hash2 == map[split - 1].hash;
1247                 :          dxtrace(printk(KERN_INFO "Split block %lu at %x, %i/%i\n",
1248                 :                         (unsigned long)dx_get_block(frame->at),
1249                 :                                        hash2, split, count-split));
1250                 :
1251                 :          /* Fancy dance to stay within two buffers */
1252          147258 :          de2 = dx_move_dirents(data1, data2, map + split, count - split, blocksize);
1253          147258 :          de = dx_pack_dirents(data1, blocksize);
1254          147258 :          de->rec_len = ext4_rec_len_to_disk(data1 + blocksize - (char *) de,
1255                 :                                             blocksize);
1256          147258 :          de2->rec_len = ext4_rec_len_to_disk(data2 + blocksize - (char *) de2,
1257                 :                                              blocksize);
1258                 :          dxtrace(dx_show_leaf (hinfo, (struct ext4_dir_entry_2 *) data1, blocksize, 1));
1259                 :          dxtrace(dx_show_leaf (hinfo, (struct ext4_dir_entry_2 *) data2, blocksize, 1));
1260                 :
1261                 :          /* Which block gets the new entry? */
1262          147258 :          if (hinfo->hash >= hash2)
1263                 :          {
1264           72771 :                  swap(*bh, bh2);
1265           72771 :                  de = de2;
1266                 :          }
1267          147258 :          dx_insert_block(frame, hash2 + continued, newblock);
1268          147258 :          err = ext4_handle_dirty_metadata(handle, dir, bh2);
1269          147258 :          if (err)
1270               0 :                  goto journal_error;
1271          147258 :          err = ext4_handle_dirty_metadata(handle, dir, frame->bh);
1272          147258 :          if (err)
1273               0 :                  goto journal_error;
1274          147258 :          brelse(bh2);
1275                 :          dxtrace(dx_show_index("frame", frame->entries));
1276          147258 :          return de;
1277                 :
1278               0 : journal_error:
1279               0 :          brelse(*bh);
1280               0 :          brelse(bh2);
1281               0 :          *bh = NULL;
1282               0 :          ext4_std_error(dir->i_sb, err);
1283               0 : errout:
1284               0 :          *error = err;
1285               0 :          return NULL;
1286                 : }
1287                 :
1288                 : /*
1289                 :  * Add a new entry into a directory (leaf) block.  If de is non-NULL,
1290                 :  * it points to a directory entry which is guaranteed to be large
1291                 :  * enough for new directory entry.  If de is NULL, then
1292                 :  * add_dirent_to_buf will attempt search the directory block for
1293                 :  * space.  It will return -ENOSPC if no space is available, and -EIO
1294                 :  * and -EEXIST if directory entry already exists.
1295                 :  *
1296                 :  * NOTE!  bh is NOT released in the case where ENOSPC is returned.  In
1297                 :  * all other cases bh is released.
1298                 :  */
1299                 : static int add_dirent_to_buf(handle_t *handle, struct dentry *dentry,
1300                 :                              struct inode *inode, struct ext4_dir_entry_2 *de,
1301                 :                              struct buffer_head *bh)
1302       9459845 : {
1303       9459845 :          struct inode    *dir = dentry->d_parent->d_inode;
1304       9459845 :          const char      *name = dentry->d_name.name;
1305       9459845 :          int             namelen = dentry->d_name.len;
1306       9459845 :          unsigned int    offset = 0;
1307       9459845 :          unsigned int    blocksize = dir->i_sb->s_blocksize;
1308                 :          unsigned short  reclen;
1309                 :          int             nlen, rlen, err;
1310                 :          char            *top;
1311                 :
1312       9459845 :          reclen = EXT4_DIR_REC_LEN(namelen);
1313       9459845 :          if (!de) {
1314       9312587 :                  de = (struct ext4_dir_entry_2 *)bh->b_data;
1315       9312587 :                  top = bh->b_data + blocksize - reclen;
1316    1054317176 :                  while ((char *) de <= top) {
1317    1044857321 :                          if (!ext4_check_dir_entry("ext4_add_entry", dir, de,
1318                 :                                                    bh, offset)) {
1319               0 :                                  brelse(bh);
1320               0 :                                  return -EIO;
1321                 :                          }
1322    1044857321 :                          if (ext4_match(namelen, name, de)) {
1323               0 :                                  brelse(bh);
1324               0 :                                  return -EEXIST;
1325                 :                          }
1326    1044857321 :                          nlen = EXT4_DIR_REC_LEN(de->name_len);
1327    2089714642 :                          rlen = ext4_rec_len_from_disk(de->rec_len, blocksize);
1328    1044857321 :                          if ((de->inode? rlen - nlen: rlen) >= reclen)
1329       9165319 :                                  break;
1330    1035692002 :                          de = (struct ext4_dir_entry_2 *)((char *)de + rlen);
1331    1035692002 :                          offset += rlen;
1332                 :                  }
1333       9312587 :                  if ((char *) de > top)
1334        147268 :                          return -ENOSPC;
```

```
1335                :         }
1336                :         BUFFER_TRACE(bh, "get_write_access");
1337          9312577 :         err = ext4_journal_get_write_access(handle, bh);
1338          9312577 :         if (err) {
1339                0 :                 ext4_std_error(dir->i_sb, err);
1340                0 :                 brelse(bh);
1341                0 :                 return err;
1342                :         }
1343                :
1344                :         /* By now the buffer is marked for journaling */
1345          9312577 :         nlen = EXT4_DIR_REC_LEN(de->name_len);
1346         18625154 :         rlen = ext4_rec_len_from_disk(de->rec_len, blocksize);
1347          9312577 :         if (de->inode) {
1348          9306009 :                 struct ext4_dir_entry_2 *de1 = (struct ext4_dir_entry_2 *)((char *)de + nlen);
1349          9306009 :                 de1->rec_len = ext4_rec_len_to_disk(rlen - nlen, blocksize);
1350          9306009 :                 de->rec_len = ext4_rec_len_to_disk(nlen, blocksize);
1351          9306009 :                 de = de1;
1352                :         }
1353          9312577 :         de->file_type = EXT4_FT_UNKNOWN;
1354          9312577 :         if (inode) {
1355          9312577 :                 de->inode = cpu_to_le32(inode->i_ino);
1356          9312577 :                 ext4_set_de_type(dir->i_sb, de, inode->i_mode);
1357                :         } else
1358                0 :                 de->inode = 0;
1359          9312577 :         de->name_len = namelen;
1360         18625154 :         memcpy(de->name, name, namelen);
1361                :         /*
1362                :          * XXX shouldn't update any times until successful
1363                :          * completion of syscall, but too many callers depend
1364                :          * on this.
1365                :          *
1366                :          * XXX similarly, too many callers depend on
1367                :          * ext4_new_inode() setting the times, but error
1368                :          * recovery deletes the inode, so the worst that can
1369                :          * happen is that the times are slightly out of date
1370                :          * and/or different from the directory change time.
1371                :          */
1372          9312577 :         dir->i_mtime = dir->i_ctime = ext4_current_time(dir);
1373                :         ext4_update_dx_flag(dir);
1374          9312577 :         dir->i_version++;
1375          9312577 :         ext4_mark_inode_dirty(handle, dir);
1376                :         BUFFER_TRACE(bh, "call ext4_handle_dirty_metadata");
1377          9312577 :         err = ext4_handle_dirty_metadata(handle, dir, bh);
1378          9312577 :         if (err)
1379                0 :                 ext4_std_error(dir->i_sb, err);
1380          9312577 :         brelse(bh);
1381          9312577 :         return 0;
1382                : }
1383                :
1384                : /*
1385                :  * This converts a one block unindexed directory to a 3 block indexed
1386                :  * directory, and adds the dentry to the indexed directory.
1387                :  */
1388                : static int make_indexed_dir(handle_t *handle, struct dentry *dentry,
1389                :                             struct inode *inode, struct buffer_head *bh)
1390             2196 : {
1391             2196 :         struct inode    *dir = dentry->d_parent->d_inode;
1392             2196 :         const char      *name = dentry->d_name.name;
1393             2196 :         int             namelen = dentry->d_name.len;
1394                :         struct buffer_head *bh2;
1395                :         struct dx_root  *root;
1396                :         struct dx_frame frames[2], *frame;
1397                :         struct dx_entry *entries;
1398                :         struct ext4_dir_entry_2 *de, *de2;
1399                :         char            *data1, *top;
1400                :         unsigned        len;
1401                :         int             retval;
1402                :         unsigned        blocksize;
1403                :         struct dx_hash_info hinfo;
1404                :         ext4_lblk_t  block;
1405                :         struct fake_dirent *fde;
1406                :
1407             2196 :         blocksize =  dir->i_sb->s_blocksize;
1408                :         dxtrace(printk(KERN_DEBUG "Creating index: inode %lu\n", dir->i_ino));
1409             2196 :         retval = ext4_journal_get_write_access(handle, bh);
1410             2196 :         if (retval) {
1411                0 :                 ext4_std_error(dir->i_sb, retval);
1412                0 :                 brelse(bh);
1413                0 :                 return retval;
1414                :         }
1415             2196 :         root = (struct dx_root *) bh->b_data;
1416                :
1417                :         /* The 0th block becomes the root, move the dirents out */
1418             2196 :         fde = &root->dotdot;
1419             4392 :         de = (struct ext4_dir_entry_2 *)((char *)fde +
1420                :                 ext4_rec_len_from_disk(fde->rec_len, blocksize));
1421             2196 :         if ((char *) de >= (((char *) root) + blocksize)) {
1422                0 :                 ext4_error(dir->i_sb, __func__,
1423                :                            "invalid rec_len for '..' in inode %lu",
1424                :                            dir->i_ino);
```

```
    1425              0 :                   brelse(bh);
    1426              0 :                   return -EIO;
    1427                :           }
    1428           2196 :           len = ((char *) root) + blocksize - (char *) de;
    1429                :
    1430                :           /* Allocate new block for the 0th block's dirents */
    1431           2196 :           bh2 = ext4_append(handle, dir, &block, &retval);
    1432           2196 :           if (!(bh2)) {
    1433              0 :                   brelse(bh);
    1434              0 :                   return retval;
    1435                :           }
    1436           2196 :           EXT4_I(dir)->i_flags |= EXT4_INDEX_FL;
    1437           2196 :           data1 = bh2->b_data;
    1438                :
    1439           4392 :           memcpy (data1, de, len);
    1440           2196 :           de = (struct ext4_dir_entry_2 *) data1;
    1441           2196 :           top = data1 + len;
    1442         390965 :           while ((char *)(de2 = ext4_next_entry(de, blocksize)) < top)
    1443         386573 :                   de = de2;
    1444           2196 :           de->rec_len = ext4_rec_len_to_disk(data1 + blocksize - (char *) de,
    1445                :                                              blocksize);
    1446                :           /* Initialize the root; the dot dirents already exist */
    1447           2196 :           de = (struct ext4_dir_entry_2 *) (&root->dotdot);
    1448           2196 :           de->rec_len = ext4_rec_len_to_disk(blocksize - EXT4_DIR_REC_LEN(2),
    1449                :                                              blocksize);
    1450           2196 :           memset (&root->info, 0, sizeof(root->info));
    1451           2196 :           root->info.info_length = sizeof(root->info);
    1452           4392 :           root->info.hash_version = EXT4_SB(dir->i_sb)->s_def_hash_version;
    1453           2196 :           entries = root->entries;
    1454                :           dx_set_block(entries, 1);
    1455                :           dx_set_count(entries, 1);
    1456                :           dx_set_limit(entries, dx_root_limit(dir, sizeof(root->info)));
    1457                :
    1458                :           /* Initialize as for dx_probe */
    1459           2196 :           hinfo.hash_version = root->info.hash_version;
    1460           2196 :           if (hinfo.hash_version <= DX_HASH_TEA)
    1461           4392 :                   hinfo.hash_version += EXT4_SB(dir->i_sb)->s_hash_unsigned;
    1462           4392 :           hinfo.seed = EXT4_SB(dir->i_sb)->s_hash_seed;
    1463           2196 :           ext4fs_dirhash(name, namelen, &hinfo);
    1464           2196 :           frame = frames;
    1465           2196 :           frame->entries = entries;
    1466           2196 :           frame->at = entries;
    1467           2196 :           frame->bh = bh;
    1468           2196 :           bh = bh2;
    1469           2196 :           de = do_split(handle,dir, &bh, frame, &hinfo, &retval);
    1470           2196 :           dx_release (frames);
    1471           2196 :           if (!(de))
    1472              0 :                   return retval;
    1473                :
    1474           2196 :           return add_dirent_to_buf(handle, dentry, inode, de, bh);
    1475                : }
    1476                :
    1477                : /*
    1478                :  *      ext4_add_entry()
    1479                :  *
    1480                :  * adds a file entry to the specified directory, using the same
    1481                :  * semantics as ext4_find_entry(). It returns NULL if it failed.
    1482                :  *
    1483                :  * NOTE!! The inode part of 'de' is left at 0 - which means you
    1484                :  * may not sleep between calling this and putting something into
    1485                :  * the entry, as someone else might have used it while you slept.
    1486                :  */
    1487                : static int ext4_add_entry(handle_t *handle, struct dentry *dentry,
    1488                :                           struct inode *inode)
    1489        9312587 : {
    1490        9312587 :           struct inode *dir = dentry->d_parent->d_inode;
    1491                :           struct buffer_head *bh;
    1492                :           struct ext4_dir_entry_2 *de;
    1493                :           struct super_block *sb;
    1494                :           int     retval;
    1495        9312587 :           int     dx_fallback=0;
    1496                :           unsigned blocksize;
    1497                :           ext4_lblk_t block, blocks;
    1498                :
    1499        9312587 :           sb = dir->i_sb;
    1500        9312587 :           blocksize = sb->s_blocksize;
    1501        9312587 :           if (!dentry->d_name.len)
    1502              0 :                   return -EINVAL;
    1503       27937761 :           if (is_dx(dir)) {
    1504        8921062 :                   retval = ext4_dx_add_entry(handle, dentry, inode);
    1505        8921062 :                   if (!retval || (retval != ERR_BAD_DX_DIR))
    1506        8921062 :                           return retval;
    1507              0 :                   EXT4_I(dir)->i_flags &= ~EXT4_INDEX_FL;
    1508              0 :                   dx_fallback++;
    1509              0 :                   ext4_mark_inode_dirty(handle, dir);
    1510                :           }
    1511         391525 :           blocks = dir->i_size >> sb->s_blocksize_bits;
    1512         391525 :           for (block = 0; block < blocks; block++) {
    1513         391525 :                   bh = ext4_bread(handle, dir, block, 0, &retval);
    1514         391525 :                   if(!bh)
```

```
    1515              0 :                         return retval;
    1516         391525 :                 retval = add_dirent_to_buf(handle, dentry, inode, NULL, bh);
    1517         391525 :                 if (retval != -ENOSPC)
    1518         389329 :                         return retval;
    1519                :
    1520           4392 :                 if (blocks == 1 && !dx_fallback &&
    1521                :                     EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_DIR_INDEX))
    1522           2196 :                         return make_indexed_dir(handle, dentry, inode, bh);
    1523              0 :                 brelse(bh);
    1524                :         }
    1525              0 :         bh = ext4_append(handle, dir, &block, &retval);
    1526              0 :         if (!bh)
    1527              0 :                 return retval;
    1528              0 :         de = (struct ext4_dir_entry_2 *) bh->b_data;
    1529              0 :         de->inode = 0;
    1530              0 :         de->rec_len = ext4_rec_len_to_disk(blocksize, blocksize);
    1531              0 :         return add_dirent_to_buf(handle, dentry, inode, de, bh);
    1532                : }
    1533                :
    1534                : /*
    1535                :  * Returns 0 for success, or a negative error value
    1536                :  */
    1537                : static int ext4_dx_add_entry(handle_t *handle, struct dentry *dentry,
    1538                :                              struct inode *inode)
    1539        8921062 : {
    1540                :         struct dx_frame frames[2], *frame;
    1541                :         struct dx_entry *entries, *at;
    1542                :         struct dx_hash_info hinfo;
    1543                :         struct buffer_head *bh;
    1544        8921062 :         struct inode *dir = dentry->d_parent->d_inode;
    1545        8921062 :         struct super_block *sb = dir->i_sb;
    1546                :         struct ext4_dir_entry_2 *de;
    1547                :         int err;
    1548                :
    1549        8921062 :         frame = dx_probe(&dentry->d_name, dir, &hinfo, frames, &err);
    1550        8921062 :         if (!frame)
    1551              0 :                 return err;
    1552        8921062 :         entries = frame->entries;
    1553        8921062 :         at = frame->at;
    1554                :
    1555       17842124 :         if (!(bh = ext4_bread(handle,dir, dx_get_block(frame->at), 0, &err)))
    1556              0 :                 goto cleanup;
    1557                :
    1558                :         BUFFER_TRACE(bh, "get_write_access");
    1559        8921062 :         err = ext4_journal_get_write_access(handle, bh);
    1560        8921062 :         if (err)
    1561              0 :                 goto journal_error;
    1562                :
    1563        8921062 :         err = add_dirent_to_buf(handle, dentry, inode, NULL, bh);
    1564        8921062 :         if (err != -ENOSPC) {
    1565        8775990 :                 bh = NULL;
    1566        8775990 :                 goto cleanup;
    1567                :         }
    1568                :
    1569                :         /* Block full, should compress but for now just split */
    1570                :         dxtrace(printk(KERN_DEBUG "using %u of %u node entries\n",
    1571                :                        dx_get_count(entries), dx_get_limit(entries)));
    1572                :         /* Need to split index? */
    1573         145072 :         if (dx_get_count(entries) == dx_get_limit(entries)) {
    1574                :                 ext4_lblk_t newblock;
    1575            472 :                 unsigned icount = dx_get_count(entries);
    1576            472 :                 int levels = frame - frames;
    1577                :                 struct dx_entry *entries2;
    1578                :                 struct dx_node *node2;
    1579                :                 struct buffer_head *bh2;
    1580                :
    1581           1376 :                 if (levels && (dx_get_count(frames->entries) ==
    1582                :                                dx_get_limit(frames->entries))) {
    1583             10 :                         ext4_warning(sb, __func__,
    1584                :                                      "Directory index full!");
    1585             10 :                         err = -ENOSPC;
    1586             10 :                         goto cleanup;
    1587                :                 }
    1588            462 :                 bh2 = ext4_append (handle, dir, &newblock, &err);
    1589            462 :                 if (!(bh2))
    1590              0 :                         goto cleanup;
    1591            462 :                 node2 = (struct dx_node *)(bh2->b_data);
    1592            462 :                 entries2 = node2->entries;
    1593            462 :                 node2->fake.rec_len = ext4_rec_len_to_disk(sb->s_blocksize,
    1594                :                                                           sb->s_blocksize);
    1595            462 :                 node2->fake.inode = 0;
    1596                :                 BUFFER_TRACE(frame->bh, "get_write_access");
    1597            462 :                 err = ext4_journal_get_write_access(handle, frame->bh);
    1598            462 :                 if (err)
    1599              0 :                         goto journal_error;
    1600            462 :                 if (levels) {
    1601            442 :                         unsigned icount1 = icount/2, icount2 = icount - icount1;
    1602            884 :                         unsigned hash2 = dx_get_hash(entries + icount1);
    1603                :                         dxtrace(printk(KERN_DEBUG "Split index %i/%i\n",
    1604                :                                        icount1, icount2));
```

```
1605                :
1606                :                               BUFFER_TRACE(frame->bh, "get_write_access"); /* index root */
1607              442 :                               err = ext4_journal_get_write_access(handle,
1608                :                                                                   frames[0].bh);
1609              442 :                               if (err)
1610                0 :                                       goto journal_error;
1611                :
1612              884 :                               memcpy((char *) entries2, (char *) (entries + icount1),
1613                :                                      icount2 * sizeof(struct dx_entry));
1614                :                               dx_set_count(entries, icount1);
1615                :                               dx_set_count(entries2, icount2);
1616                :                               dx_set_limit(entries2, dx_node_limit(dir));
1617                :
1618                :                               /* Which index block gets the new entry? */
1619              442 :                               if (at - entries >= icount1) {
1620              199 :                                       frame->at = at = at - entries - icount1 + entries2;
1621              199 :                                       frame->entries = entries = entries2;
1622              199 :                                       swap(frame->bh, bh2);
1623                :                               }
1624              442 :                               dx_insert_block(frames + 0, hash2, newblock);
1625                :                               dxtrace(dx_show_index("node", frames[1].entries));
1626                :                               dxtrace(dx_show_index("node",
1627                :                                       ((struct dx_node *) bh2->b_data)->entries));
1628              442 :                               err = ext4_handle_dirty_metadata(handle, inode, bh2);
1629              442 :                               if (err)
1630                0 :                                       goto journal_error;
1631              442 :                               brelse (bh2);
1632                :                       } else {
1633                :                               dxtrace(printk(KERN_DEBUG
1634                :                                              "Creating second level index...\n"));
1635               40 :                               memcpy((char *) entries2, (char *) entries,
1636                :                                      icount * sizeof(struct dx_entry));
1637                :                               dx_set_limit(entries2, dx_node_limit(dir));
1638                :
1639                :                               /* Set up root */
1640                :                               dx_set_count(entries, 1);
1641               20 :                               dx_set_block(entries + 0, newblock);
1642               20 :                               ((struct dx_root *) frames[0].bh->b_data)->info.indirect_levels = 1;
1643                :
1644                :                               /* Add new access path frame */
1645               20 :                               frame = frames + 1;
1646               20 :                               frame->at = at = at - entries + entries2;
1647               20 :                               frame->entries = entries = entries2;
1648               20 :                               frame->bh = bh2;
1649               20 :                               err = ext4_journal_get_write_access(handle,
1650                :                                                                   frame->bh);
1651               20 :                               if (err)
1652                0 :                                       goto journal_error;
1653                :                       }
1654              462 :                       ext4_handle_dirty_metadata(handle, inode, frames[0].bh);
1655                :               }
1656           145062 :               de = do_split(handle, dir, &bh, frame, &hinfo, &err);
1657           145062 :               if (!de)
1658                0 :                       goto cleanup;
1659           145062 :               err = add_dirent_to_buf(handle, dentry, inode, de, bh);
1660           145062 :               bh = NULL;
1661           145062 :               goto cleanup;
1662                :
1663                0 : journal_error:
1664                0 :               ext4_std_error(dir->i_sb, err);
1665          8921062 : cleanup:
1666          8921062 :               if (bh)
1667               10 :                       brelse(bh);
1668          8921062 :               dx_release(frames);
1669          8921062 :               return err;
1670                : }
1671                :
1672                : /*
1673                :  * ext4_delete_entry deletes a directory entry by merging it with the
1674                :  * previous entry
1675                :  */
1676                : static int ext4_delete_entry(handle_t *handle,
1677                :                              struct inode *dir,
1678                :                              struct ext4_dir_entry_2 *de_del,
1679                :                              struct buffer_head *bh)
1680          1664360 : {
1681                :               struct ext4_dir_entry_2 *de, *pde;
1682          1664360 :               unsigned int blocksize = dir->i_sb->s_blocksize;
1683                :               int i;
1684                :
1685          1664360 :               i = 0;
1686          1664360 :               pde = NULL;
1687          1664360 :               de = (struct ext4_dir_entry_2 *) bh->b_data;
1688         44646036 :               while (i < bh->b_size) {
1689         42981676 :                       if (!ext4_check_dir_entry("ext4_delete_entry", dir, de, bh, i))
1690                0 :                               return -EIO;
1691         42981676 :                       if (de == de_del)  {
1692                :                               BUFFER_TRACE(bh, "get_write_access");
1693          1664360 :                               ext4_journal_get_write_access(handle, bh);
1694          1664360 :                               if (pde)
```

```
1695  4838865 :                                   pde->rec_len = ext4_rec_len_to_disk(
1696          :                                           ext4_rec_len_from_disk(pde->rec_len,
1697          :                                                                  blocksize) +
1698          :                                           ext4_rec_len_from_disk(de->rec_len,
1699          :                                                                  blocksize),
1700          :                                           blocksize);
1701          :                           else
1702    51405 :                                   de->inode = 0;
1703  1664360 :                           dir->i_version++;
1704          :                           BUFFER_TRACE(bh, "call ext4_handle_dirty_metadata");
1705  1664360 :                           ext4_handle_dirty_metadata(handle, dir, bh);
1706  1664360 :                           return 0;
1707          :                   }
1708 82634632 :                   i += ext4_rec_len_from_disk(de->rec_len, blocksize);
1709 41317316 :                   pde = de;
1710 41317316 :                   de = ext4_next_entry(de, blocksize);
1711          :           }
1712        0 :           return -ENOENT;
1713          : }
1714          :
1715          : /*
1716          :  * DIR_NLINK feature is set if 1) nlinks > EXT4_LINK_MAX or 2) nlinks == 2,
1717          :  * since this indicates that nlinks count was previously 1.
1718          :  */
1719          : static void ext4_inc_count(handle_t *handle, struct inode *inode)
1720          : {
1721          :           inc_nlink(inode);
1722  2093013 :           if (is_dx(inode) && inode->i_nlink > 1) {
1723          :                   /* limit is 16-bit i_links_count */
1724   694200 :                   if (inode->i_nlink >= EXT4_LINK_MAX || inode->i_nlink == 2) {
1725   305396 :                           inode->i_nlink = 1;
1726   610792 :                           EXT4_SET_RO_COMPAT_FEATURE(inode->i_sb,
1727          :                                                      EXT4_FEATURE_RO_COMPAT_DIR_NLINK);
1728          :                   }
1729          :           }
1730          : }
1731          :
1732          : /*
1733          :  * If a directory had nlink == 1, then we should let it be 1. This indicates
1734          :  * directory has >EXT4_LINK_MAX subdirs.
1735          :  */
1736          : static void ext4_dec_count(handle_t *handle, struct inode *inode)
1737          : {
1738          :           drop_nlink(inode);
1739   655415 :           if (S_ISDIR(inode->i_mode) && inode->i_nlink == 0)
1740          :                   inc_nlink(inode);
1741          : }
1742          :
1743          :
1744          : static int ext4_add_nondir(handle_t *handle,
1745          :                   struct dentry *dentry, struct inode *inode)
1746  8614906 : {
1747  8614906 :           int err = ext4_add_entry(handle, dentry, inode);
1748  8614906 :           if (!err) {
1749  8614906 :                   ext4_mark_inode_dirty(handle, inode);
1750  8614906 :                   d_instantiate(dentry, inode);
1751  8614906 :                   unlock_new_inode(inode);
1752  8614906 :                   return 0;
1753          :           }
1754          :           drop_nlink(inode);
1755        0 :           unlock_new_inode(inode);
1756        0 :           iput(inode);
1757        0 :           return err;
1758          : }
1759          :
1760          : /*
1761          :  * By the time this is called, we already have created
1762          :  * the directory cache entry for the new file, but it
1763          :  * is so far negative - it has no inode.
1764          :  *
1765          :  * If the create succeeds, we fill in the inode information
1766          :  * with d_instantiate().
1767          :  */
1768          : static int ext4_create(struct inode *dir, struct dentry *dentry, int mode,
1769          :                        struct nameidata *nd)
1770  8614906 : {
1771          :           handle_t *handle;
1772          :           struct inode *inode;
1773  8614906 :           int err, retries = 0;
1774          :
1775  8614906 : retry:
1776 43074530 :           handle = ext4_journal_start(dir, EXT4_DATA_TRANS_BLOCKS(dir->i_sb) +
1777          :                                       EXT4_INDEX_EXTRA_TRANS_BLOCKS + 3 +
1778        0 :                                       2*EXT4_QUOTA_INIT_BLOCKS(dir->i_sb));
1779  8614906 :           if (IS_ERR(handle))
1780        0 :                   return PTR_ERR(handle);
1781          :
1782  8614906 :           if (IS_DIRSYNC(dir))
1783          :                   ext4_handle_sync(handle);
1784          :
```

```
1785  8614906 :          inode = ext4_new_inode(handle, dir, mode, &dentry->d_name, 0);
1786  8614906 :          err = PTR_ERR(inode);
1787  8614906 :          if (!IS_ERR(inode)) {
1788  8614906 :                  inode->i_op = &ext4_file_inode_operations;
1789  8614906 :                  inode->i_fop = &ext4_file_operations;
1790  8614906 :                  ext4_set_aops(inode);
1791  8614906 :                  err = ext4_add_nondir(handle, dentry, inode);
1792           :          }
1793  8614906 :          ext4_journal_stop(handle);
1794  8614906 :          if (err == -ENOSPC && ext4_should_retry_alloc(dir->i_sb, &retries))
1795        0 :                  goto retry;
1796  8614906 :          return err;
1797           : }
1798           :
1799           : static int ext4_mknod(struct inode *dir, struct dentry *dentry,
1800           :                       int mode, dev_t rdev)
1801        0 : {
1802           :          handle_t *handle;
1803           :          struct inode *inode;
1804        0 :          int err, retries = 0;
1805           :
1806        0 :          if (!new_valid_dev(rdev))
1807        0 :                  return -EINVAL;
1808           :
1809        0 : retry:
1810        0 :          handle = ext4_journal_start(dir, EXT4_DATA_TRANS_BLOCKS(dir->i_sb) +
1811           :                                      EXT4_INDEX_EXTRA_TRANS_BLOCKS + 3 +
1812        0 :                                      2*EXT4_QUOTA_INIT_BLOCKS(dir->i_sb));
1813        0 :          if (IS_ERR(handle))
1814        0 :                  return PTR_ERR(handle);
1815           :
1816        0 :          if (IS_DIRSYNC(dir))
1817           :                  ext4_handle_sync(handle);
1818           :
1819        0 :          inode = ext4_new_inode(handle, dir, mode, &dentry->d_name, 0);
1820        0 :          err = PTR_ERR(inode);
1821        0 :          if (!IS_ERR(inode)) {
1822        0 :                  init_special_inode(inode, inode->i_mode, rdev);
1823           : #ifdef CONFIG_EXT4_FS_XATTR
1824        0 :                  inode->i_op = &ext4_special_inode_operations;
1825           : #endif
1826        0 :                  err = ext4_add_nondir(handle, dentry, inode);
1827           :          }
1828        0 :          ext4_journal_stop(handle);
1829        0 :          if (err == -ENOSPC && ext4_should_retry_alloc(dir->i_sb, &retries))
1830        0 :                  goto retry;
1831        0 :          return err;
1832           : }
1833           :
1834           : static int ext4_mkdir(struct inode *dir, struct dentry *dentry, int mode)
1835   697658 : {
1836           :          handle_t *handle;
1837           :          struct inode *inode;
1838           :          struct buffer_head *dir_block;
1839           :          struct ext4_dir_entry_2 *de;
1840   697658 :          unsigned int blocksize = dir->i_sb->s_blocksize;
1841   697658 :          int err, retries = 0;
1842           :
1843  2092974 :          if (EXT4_DIR_LINK_MAX(dir))
1844        0 :                  return -EMLINK;
1845           :
1846   697666 : retry:
1847  3488330 :          handle = ext4_journal_start(dir, EXT4_DATA_TRANS_BLOCKS(dir->i_sb) +
1848           :                                      EXT4_INDEX_EXTRA_TRANS_BLOCKS + 3 +
1849        0 :                                      2*EXT4_QUOTA_INIT_BLOCKS(dir->i_sb));
1850   697666 :          if (IS_ERR(handle))
1851        0 :                  return PTR_ERR(handle);
1852           :
1853   697666 :          if (IS_DIRSYNC(dir))
1854           :                  ext4_handle_sync(handle);
1855           :
1856   697666 :          inode = ext4_new_inode(handle, dir, S_IFDIR | mode,
1857           :                                 &dentry->d_name, 0);
1858   697666 :          err = PTR_ERR(inode);
1859   697666 :          if (IS_ERR(inode))
1860        0 :                  goto out_stop;
1861           :
1862   697666 :          inode->i_op = &ext4_dir_inode_operations;
1863   697666 :          inode->i_fop = &ext4_dir_operations;
1864   697666 :          inode->i_size = EXT4_I(inode)->i_disksize = inode->i_sb->s_blocksize;
1865   697666 :          dir_block = ext4_bread(handle, inode, 0, 1, &err);
1866   697666 :          if (!dir_block)
1867        0 :                  goto out_clear_inode;
1868           :          BUFFER_TRACE(dir_block, "get_write_access");
1869   697666 :          ext4_journal_get_write_access(handle, dir_block);
1870   697666 :          de = (struct ext4_dir_entry_2 *) dir_block->b_data;
1871   697666 :          de->inode = cpu_to_le32(inode->i_ino);
1872   697666 :          de->name_len = 1;
1873   697666 :          de->rec_len = ext4_rec_len_to_disk(EXT4_DIR_REC_LEN(de->name_len),
1874           :                                             blocksize);
```

```
1875     697666 :          strcpy(de->name, ".");
1876     697666 :          ext4_set_de_type(dir->i_sb, de, S_IFDIR);
1877     697666 :          de = ext4_next_entry(de, blocksize);
1878     697666 :          de->inode = cpu_to_le32(dir->i_ino);
1879     697666 :          de->rec_len = ext4_rec_len_to_disk(blocksize - EXT4_DIR_REC_LEN(1),
1880             :                                            blocksize);
1881     697666 :          de->name_len = 2;
1882     697666 :          strcpy(de->name, "..");
1883     697666 :          ext4_set_de_type(dir->i_sb, de, S_IFDIR);
1884     697666 :          inode->i_nlink = 2;
1885             :          BUFFER_TRACE(dir_block, "call ext4_handle_dirty_metadata");
1886     697666 :          ext4_handle_dirty_metadata(handle, dir, dir_block);
1887     697666 :          brelse(dir_block);
1888     697666 :          ext4_mark_inode_dirty(handle, inode);
1889     697666 :          err = ext4_add_entry(handle, dentry, inode);
1890     697666 :          if (err) {
1891         10 : out_clear_inode:
1892             :                  clear_nlink(inode);
1893         10 :                  unlock_new_inode(inode);
1894         10 :                  ext4_mark_inode_dirty(handle, inode);
1895         10 :                  iput(inode);
1896         10 :                  goto out_stop;
1897             :          }
1898             :          ext4_inc_count(handle, dir);
1899             :          ext4_update_dx_flag(dir);
1900     697656 :          ext4_mark_inode_dirty(handle, dir);
1901     697656 :          d_instantiate(dentry, inode);
1902     697656 :          unlock_new_inode(inode);
1903     697666 : out_stop:
1904     697666 :          ext4_journal_stop(handle);
1905     697666 :          if (err == -ENOSPC && ext4_should_retry_alloc(dir->i_sb, &retries))
1906          8 :                  goto retry;
1907     697658 :          return err;
1908             : }
1909             :
1910             : /*
1911             :  * routine to check that the specified directory is empty (for rmdir)
1912             :  */
1913             : static int empty_dir(struct inode *inode)
1914     655400 : {
1915             :          unsigned int offset;
1916             :          struct buffer_head *bh;
1917             :          struct ext4_dir_entry_2 *de, *de1;
1918             :          struct super_block *sb;
1919     655400 :          int err = 0;
1920             :
1921     655400 :          sb = inode->i_sb;
1922     655400 :          if (inode->i_size < EXT4_DIR_REC_LEN(1) + EXT4_DIR_REC_LEN(2) ||
1923             :              !(bh = ext4_bread(NULL, inode, 0, 0, &err))) {
1924          0 :                  if (err)
1925          0 :                          ext4_error(inode->i_sb, __func__,
1926             :                                     "error %d reading directory #%lu offset 0",
1927             :                                     err, inode->i_ino);
1928             :                  else
1929          0 :                          ext4_warning(inode->i_sb, __func__,
1930             :                                       "bad directory (dir #%lu) - no data block",
1931             :                                       inode->i_ino);
1932          0 :                  return 1;
1933             :          }
1934     655400 :          de = (struct ext4_dir_entry_2 *) bh->b_data;
1935    1310800 :          de1 = ext4_next_entry(de, sb->s_blocksize);
1936     655400 :          if (le32_to_cpu(de->inode) != inode->i_ino ||
1937             :                          !le32_to_cpu(de1->inode) ||
1938             :                          strcmp(".", de->name) ||
1939             :                          strcmp("..", de1->name)) {
1940          0 :                  ext4_warning(inode->i_sb, "empty_dir",
1941             :                               "bad directory (dir #%lu) - no `.' or `..'",
1942             :                               inode->i_ino);
1943          0 :                  brelse(bh);
1944          0 :                  return 1;
1945             :          }
1946    1966200 :          offset = ext4_rec_len_from_disk(de->rec_len, sb->s_blocksize) +
1947             :                   ext4_rec_len_from_disk(de1->rec_len, sb->s_blocksize);
1948    1310800 :          de = ext4_next_entry(de1, sb->s_blocksize);
1949    1316742 :          while (offset < inode->i_size) {
1950       5942 :                  if (!bh ||
1951             :                          (void *) de >= (void *) (bh->b_data+sb->s_blocksize)) {
1952       5942 :                          err = 0;
1953       5942 :                          brelse(bh);
1954       5942 :                          bh = ext4_bread(NULL, inode,
1955             :                                  offset >> EXT4_BLOCK_SIZE_BITS(sb), 0, &err);
1956       5942 :                          if (!bh) {
1957          0 :                                  if (err)
1958          0 :                                          ext4_error(sb, __func__,
1959             :                                                     "error %d reading directory"
1960             :                                                     " #%lu offset %u",
1961             :                                                     err, inode->i_ino, offset);
1962          0 :                          offset += sb->s_blocksize;
1963          0 :                          continue;
1964             :                  }
```

```
1965          5942 :                          de = (struct ext4_dir_entry_2 *) bh->b_data;
1966               :                  }
1967          5942 :                  if (!ext4_check_dir_entry("empty_dir", inode, de, bh, offset)) {
1968             0 :                          de = (struct ext4_dir_entry_2 *)(bh->b_data +
1969               :                                                           sb->s_blocksize);
1970             0 :                          offset = (offset | (sb->s_blocksize - 1)) + 1;
1971             0 :                          continue;
1972               :                  }
1973          5942 :                  if (le32_to_cpu(de->inode)) {
1974             0 :                          brelse(bh);
1975             0 :                          return 0;
1976               :                  }
1977         11884 :                  offset += ext4_rec_len_from_disk(de->rec_len, sb->s_blocksize);
1978         11884 :                  de = ext4_next_entry(de, sb->s_blocksize);
1979               :          }
1980        655400 :          brelse(bh);
1981        655400 :          return 1;
1982               : }
1983               :
1984               : /* ext4_orphan_add() links an unlinked or truncated inode into a list of
1985               :  * such inodes, starting at the superblock, in case we crash before the
1986               :  * file is closed/deleted, or in case the inode truncate spans multiple
1987               :  * transactions and the last transaction is not recovered after a crash.
1988               :  *
1989               :  * At filesystem recovery time, we walk this list deleting unlinked
1990               :  * inodes and truncating linked inodes in ext4_orphan_cleanup().
1991               :  */
1992               : int ext4_orphan_add(handle_t *handle, struct inode *inode)
1993       3603999 : {
1994       3603999 :          struct super_block *sb = inode->i_sb;
1995               :          struct ext4_iloc iloc;
1996       3603999 :          int err = 0, rc;
1997               :
1998       3603999 :          if (!ext4_handle_valid(handle))
1999             0 :                  return 0;
2000               :
2001       3603999 :          mutex_lock(&EXT4_SB(sb)->s_orphan_lock);
2002       7207998 :          if (!list_empty(&EXT4_I(inode)->i_orphan))
2003       1662896 :                  goto out_unlock;
2004               :
2005               :          /* Orphan handling is only valid for files with data blocks
2006               :           * being truncated, or files being unlinked. */
2007               :
2008               :          /* @@@ FIXME: Observation from aviro:
2009               :           * I think I can trigger J_ASSERT in ext4_orphan_add().  We block
2010               :           * here (on s_orphan_lock), so race with ext4_link() which might bump
2011               :           * ->i_nlink. For, say it, character device. Not a regular file,
2012               :           * not a directory, not a symlink and ->i_nlink > 0.
2013               :           *
2014               :           * tytso, 4/25/2009: I'm not sure how that could happen;
2015               :           * shouldn't the fs core protect us from these sort of
2016               :           * unlink()/link() races?
2017               :           */
2018       1941103 :          J_ASSERT((S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) ||
2019               :                    S_ISLNK(inode->i_mode)) || inode->i_nlink == 0);
2020               :
2021               :          BUFFER_TRACE(EXT4_SB(sb)->s_sbh, "get_write_access");
2022       1941103 :          err = ext4_journal_get_write_access(handle, EXT4_SB(sb)->s_sbh);
2023       1941103 :          if (err)
2024             0 :                  goto out_unlock;
2025               :
2026       1941103 :          err = ext4_reserve_inode_write(handle, inode, &iloc);
2027       1941103 :          if (err)
2028             0 :                  goto out_unlock;
2029               :
2030               :          /* Insert this inode at the head of the on-disk orphan list... */
2031       1941103 :          NEXT_ORPHAN(inode) = le32_to_cpu(EXT4_SB(sb)->s_es->s_last_orphan);
2032       1941103 :          EXT4_SB(sb)->s_es->s_last_orphan = cpu_to_le32(inode->i_ino);
2033       1941103 :          err = ext4_handle_dirty_metadata(handle, inode, EXT4_SB(sb)->s_sbh);
2034       1941103 :          rc = ext4_mark_iloc_dirty(handle, inode, &iloc);
2035       1941103 :          if (!err)
2036       1941103 :                  err = rc;
2037               :
2038               :          /* Only add to the head of the in-memory list if all the
2039               :           * previous operations succeeded.  If the orphan_add is going to
2040               :           * fail (possibly taking the journal offline), we can't risk
2041               :           * leaving the inode on the orphan list: stray orphan-list
2042               :           * entries can cause panics at unmount time.
2043               :           *
2044               :           * This is safe: on error we're going to ignore the orphan list
2045               :           * anyway on the next recovery. */
2046       1941103 :          if (!err)
2047       3882206 :                  list_add(&EXT4_I(inode)->i_orphan, &EXT4_SB(sb)->s_orphan);
2048               :
2049       1941103 :          jbd_debug(4, "superblock will point to %lu\n", inode->i_ino);
2050       1941103 :          jbd_debug(4, "orphan inode %lu will point to %d\n",
2051               :                          inode->i_ino, NEXT_ORPHAN(inode));
2052       3603999 : out_unlock:
2053       3603999 :          mutex_unlock(&EXT4_SB(sb)->s_orphan_lock);
2054       3603999 :          ext4_std_error(inode->i_sb, err);
```

```
2055  3603999 :          return err;
2056          : }
2057          :
2058          : /*
2059          :  * ext4_orphan_del() removes an unlinked or truncated inode from the list
2060          :  * of such inodes stored on disk, because it is finally being cleaned up.
2061          :  */
2062          : int ext4_orphan_del(handle_t *handle, struct inode *inode)
2063  1941117 : {
2064          :          struct list_head *prev;
2065  1941117 :          struct ext4_inode_info *ei = EXT4_I(inode);
2066          :          struct ext4_sb_info *sbi;
2067          :          __u32 ino_next;
2068          :          struct ext4_iloc iloc;
2069  1941117 :          int err = 0;
2070          :
2071  1941117 :          if (!ext4_handle_valid(handle))
2072        0 :                  return 0;
2073          :
2074  3882234 :          mutex_lock(&EXT4_SB(inode->i_sb)->s_orphan_lock);
2075  3882238 :          if (list_empty(&ei->i_orphan))
2076       16 :                  goto out;
2077          :
2078  1941103 :          ino_next = NEXT_ORPHAN(inode);
2079  1941103 :          prev = ei->i_orphan.prev;
2080  3882206 :          sbi = EXT4_SB(inode->i_sb);
2081          :
2082  1941103 :          jbd_debug(4, "remove inode %lu from orphan list\n", inode->i_ino);
2083          :
2084  1941103 :          list_del_init(&ei->i_orphan);
2085          :
2086          :          /* If we're on an error path, we may not have a valid
2087          :           * transaction handle with which to update the orphan list on
2088          :           * disk, but we still need to remove the inode from the linked
2089          :           * list in memory. */
2090  1941103 :          if (sbi->s_journal && !handle)
2091        0 :                  goto out;
2092          :
2093  1941103 :          err = ext4_reserve_inode_write(handle, inode, &iloc);
2094  1941103 :          if (err)
2095        0 :                  goto out_err;
2096          :
2097  1941103 :          if (prev == &sbi->s_orphan) {
2098  1743716 :                  jbd_debug(4, "superblock will point to %u\n", ino_next);
2099          :                  BUFFER_TRACE(sbi->s_sbh, "get_write_access");
2100  1743716 :                  err = ext4_journal_get_write_access(handle, sbi->s_sbh);
2101  1743716 :                  if (err)
2102        0 :                          goto out_brelse;
2103  1743716 :                  sbi->s_es->s_last_orphan = cpu_to_le32(ino_next);
2104  1743716 :                  err = ext4_handle_dirty_metadata(handle, inode, sbi->s_sbh);
2105          :          } else {
2106          :                  struct ext4_iloc iloc2;
2107          :                  struct inode *i_prev =
2108   197387 :                          &list_entry(prev, struct ext4_inode_info, i_orphan)->vfs_inode;
2109          :
2110   197387 :                  jbd_debug(4, "orphan inode %lu will point to %u\n",
2111          :                            i_prev->i_ino, ino_next);
2112   197387 :                  err = ext4_reserve_inode_write(handle, i_prev, &iloc2);
2113   197387 :                  if (err)
2114        0 :                          goto out_brelse;
2115   197387 :                  NEXT_ORPHAN(i_prev) = ino_next;
2116   197387 :                  err = ext4_mark_iloc_dirty(handle, i_prev, &iloc2);
2117          :          }
2118  1941103 :          if (err)
2119        0 :                  goto out_brelse;
2120  1941103 :          NEXT_ORPHAN(inode) = 0;
2121  1941103 :          err = ext4_mark_iloc_dirty(handle, inode, &iloc);
2122          :
2123  1941103 : out_err:
2124  1941103 :          ext4_std_error(inode->i_sb, err);
2125  1941119 : out:
2126  3882238 :          mutex_unlock(&EXT4_SB(inode->i_sb)->s_orphan_lock);
2127  1941119 :          return err;
2128          :
2129        0 : out_brelse:
2130        0 :          brelse(iloc.bh);
2131        0 :          goto out_err;
2132          : }
2133          :
2134          : static int ext4_rmdir(struct inode *dir, struct dentry *dentry)
2135   655400 : {
2136          :          int retval;
2137          :          struct inode *inode;
2138          :          struct buffer_head *bh;
2139          :          struct ext4_dir_entry_2 *de;
2140          :          handle_t *handle;
2141          :
2142          :          /* Initialize quotas before so that eventual writes go in
2143          :           * separate transaction */
2144   655400 :          vfs_dq_init(dentry->d_inode);
```

```
2145  2621600 :          handle = ext4_journal_start(dir, EXT4_DELETE_TRANS_BLOCKS(dir->i_sb));
2146   655400 :          if (IS_ERR(handle))
2147        0 :                  return PTR_ERR(handle);
2148          :
2149   655400 :          retval = -ENOENT;
2150   655400 :          bh = ext4_find_entry(dir, &dentry->d_name, &de);
2151   655400 :          if (!bh)
2152        0 :                  goto end_rmdir;
2153          :
2154   655400 :          if (IS_DIRSYNC(dir))
2155          :                  ext4_handle_sync(handle);
2156          :
2157   655400 :          inode = dentry->d_inode;
2158          :
2159   655400 :          retval = -EIO;
2160   655400 :          if (le32_to_cpu(de->inode) != inode->i_ino)
2161        0 :                  goto end_rmdir;
2162          :
2163   655400 :          retval = -ENOTEMPTY;
2164   655400 :          if (!empty_dir(inode))
2165        0 :                  goto end_rmdir;
2166          :
2167   655400 :          retval = ext4_delete_entry(handle, dir, de, bh);
2168   655400 :          if (retval)
2169        0 :                  goto end_rmdir;
2170   655400 :          if (!EXT4_DIR_LINK_EMPTY(inode))
2171        0 :                  ext4_warning(inode->i_sb, "ext4_rmdir",
2172          :                               "empty directory has too many links (%d)",
2173          :                               inode->i_nlink);
2174   655400 :          inode->i_version++;
2175          :          clear_nlink(inode);
2176          :          /* There's no need to set i_disksize: the fact that i_nlink is
2177          :           * zero will ensure that the right thing happens during any
2178          :           * recovery. */
2179   655400 :          inode->i_size = 0;
2180   655400 :          ext4_orphan_add(handle, inode);
2181   655400 :          inode->i_ctime = dir->i_ctime = dir->i_mtime = ext4_current_time(inode);
2182   655400 :          ext4_mark_inode_dirty(handle, inode);
2183          :          ext4_dec_count(handle, dir);
2184          :          ext4_update_dx_flag(dir);
2185   655400 :          ext4_mark_inode_dirty(handle, dir);
2186          :
2187   655400 : end_rmdir:
2188   655400 :          ext4_journal_stop(handle);
2189   655400 :          brelse(bh);
2190   655400 :          return retval;
2191          : }
2192          :
2193          : static int ext4_unlink(struct inode *dir, struct dentry *dentry)
2194  1008945 : {
2195          :          int retval;
2196          :          struct inode *inode;
2197          :          struct buffer_head *bh;
2198          :          struct ext4_dir_entry_2 *de;
2199          :          handle_t *handle;
2200          :
2201          :          /* Initialize quotas before so that eventual writes go
2202          :           * in separate transaction */
2203  1008945 :          vfs_dq_init(dentry->d_inode);
2204  4035780 :          handle = ext4_journal_start(dir, EXT4_DELETE_TRANS_BLOCKS(dir->i_sb));
2205  1008945 :          if (IS_ERR(handle))
2206        0 :                  return PTR_ERR(handle);
2207          :
2208  1008945 :          if (IS_DIRSYNC(dir))
2209          :                  ext4_handle_sync(handle);
2210          :
2211  1008945 :          retval = -ENOENT;
2212  1008945 :          bh = ext4_find_entry(dir, &dentry->d_name, &de);
2213  1008945 :          if (!bh)
2214        0 :                  goto end_unlink;
2215          :
2216  1008945 :          inode = dentry->d_inode;
2217          :
2218  1008945 :          retval = -EIO;
2219  1008945 :          if (le32_to_cpu(de->inode) != inode->i_ino)
2220        0 :                  goto end_unlink;
2221          :
2222  1008945 :          if (!inode->i_nlink) {
2223        0 :                  ext4_warning(inode->i_sb, "ext4_unlink",
2224          :                               "Deleting nonexistent file (%lu), %d",
2225          :                               inode->i_ino, inode->i_nlink);
2226        0 :                  inode->i_nlink = 1;
2227          :          }
2228  1008945 :          retval = ext4_delete_entry(handle, dir, de, bh);
2229  1008945 :          if (retval)
2230        0 :                  goto end_unlink;
2231  1008945 :          dir->i_ctime = dir->i_mtime = ext4_current_time(dir);
2232          :          ext4_update_dx_flag(dir);
2233  1008945 :          ext4_mark_inode_dirty(handle, dir);
2234          :          drop_nlink(inode);
```

```
2235  1008945 :          if (!inode->i_nlink)
2236  1008945 :                  ext4_orphan_add(handle, inode);
2237  1008945 :          inode->i_ctime = ext4_current_time(inode);
2238  1008945 :          ext4_mark_inode_dirty(handle, inode);
2239  1008945 :          retval = 0;
2240        :
2241  1008945 : end_unlink:
2242  1008945 :          ext4_journal_stop(handle);
2243  1008945 :          brelse(bh);
2244  1008945 :          return retval;
2245        : }
2246        :
2247        : static int ext4_symlink(struct inode *dir,
2248        :                         struct dentry *dentry, const char *symname)
2249        0 : {
2250        :          handle_t *handle;
2251        :          struct inode *inode;
2252        0 :          int l, err, retries = 0;
2253        :
2254        0 :          l = strlen(symname)+1;
2255        0 :          if (l > dir->i_sb->s_blocksize)
2256        0 :                  return -ENAMETOOLONG;
2257        :
2258        0 : retry:
2259        0 :          handle = ext4_journal_start(dir, EXT4_DATA_TRANS_BLOCKS(dir->i_sb) +
2260        :                                      EXT4_INDEX_EXTRA_TRANS_BLOCKS + 5 +
2261        0 :                                      2*EXT4_QUOTA_INIT_BLOCKS(dir->i_sb));
2262        0 :          if (IS_ERR(handle))
2263        0 :                  return PTR_ERR(handle);
2264        :
2265        0 :          if (IS_DIRSYNC(dir))
2266        :                  ext4_handle_sync(handle);
2267        :
2268        0 :          inode = ext4_new_inode(handle, dir, S_IFLNK|S_IRWXUGO,
2269        :                                 &dentry->d_name, 0);
2270        0 :          err = PTR_ERR(inode);
2271        0 :          if (IS_ERR(inode))
2272        0 :                  goto out_stop;
2273        :
2274        0 :          if (l > sizeof(EXT4_I(inode)->i_data)) {
2275        0 :                  inode->i_op = &ext4_symlink_inode_operations;
2276        0 :                  ext4_set_aops(inode);
2277        :                  /*
2278        :                   * page_symlink() calls into ext4_prepare/commit_write.
2279        :                   * We have a transaction open.  All is sweetness.  It also sets
2280        :                   * i_size in generic_commit_write().
2281        :                   */
2282        0 :                  err = __page_symlink(inode, symname, l, 1);
2283        0 :                  if (err) {
2284        :                          clear_nlink(inode);
2285        0 :                          unlock_new_inode(inode);
2286        0 :                          ext4_mark_inode_dirty(handle, inode);
2287        0 :                          iput(inode);
2288        0 :                          goto out_stop;
2289        :                  }
2290        :          } else {
2291        :                  /* clear the extent format for fast symlink */
2292        0 :                  EXT4_I(inode)->i_flags &= ~EXT4_EXTENTS_FL;
2293        0 :                  inode->i_op = &ext4_fast_symlink_inode_operations;
2294        0 :                  memcpy((char *)&EXT4_I(inode)->i_data, symname, l);
2295        0 :                  inode->i_size = l-1;
2296        :          }
2297        0 :          EXT4_I(inode)->i_disksize = inode->i_size;
2298        0 :          err = ext4_add_nondir(handle, dentry, inode);
2299        0 : out_stop:
2300        0 :          ext4_journal_stop(handle);
2301        0 :          if (err == -ENOSPC && ext4_should_retry_alloc(dir->i_sb, &retries))
2302        0 :                  goto retry;
2303        0 :          return err;
2304        : }
2305        :
2306        : static int ext4_link(struct dentry *old_dentry,
2307        :                      struct inode *dir, struct dentry *dentry)
2308        0 : {
2309        :          handle_t *handle;
2310        0 :          struct inode *inode = old_dentry->d_inode;
2311        0 :          int err, retries = 0;
2312        :
2313        0 :          if (EXT4_DIR_LINK_MAX(inode))
2314        0 :                  return -EMLINK;
2315        :
2316        :          /*
2317        :           * Return -ENOENT if we've raced with unlink and i_nlink is 0.  Doing
2318        :           * otherwise has the potential to corrupt the orphan inode list.
2319        :           */
2320        0 :          if (inode->i_nlink == 0)
2321        0 :                  return -ENOENT;
2322        :
2323        0 : retry:
2324        0 :          handle = ext4_journal_start(dir, EXT4_DATA_TRANS_BLOCKS(dir->i_sb) +
```

```
2325                :                                               EXT4_INDEX_EXTRA_TRANS_BLOCKS);
2326              0 :         if (IS_ERR(handle))
2327              0 :                 return PTR_ERR(handle);
2328                :
2329              0 :         if (IS_DIRSYNC(dir))
2330                :                 ext4_handle_sync(handle);
2331                :
2332              0 :         inode->i_ctime = ext4_current_time(inode);
2333                :         ext4_inc_count(handle, inode);
2334              0 :         atomic_inc(&inode->i_count);
2335                :
2336              0 :         err = ext4_add_entry(handle, dentry, inode);
2337              0 :         if (!err) {
2338              0 :                 ext4_mark_inode_dirty(handle, inode);
2339              0 :                 d_instantiate(dentry, inode);
2340                :         } else {
2341                :                 drop_nlink(inode);
2342              0 :                 iput(inode);
2343                :         }
2344              0 :         ext4_journal_stop(handle);
2345              0 :         if (err == -ENOSPC && ext4_should_retry_alloc(dir->i_sb, &retries))
2346              0 :                 goto retry;
2347              0 :         return err;
2348                : }
2349                :
2350                : #define PARENT_INO(buffer, size) \
2351                :         (ext4_next_entry((struct ext4_dir_entry_2 *)(buffer), size)->inode)
2352                :
2353                : /*
2354                :  * Anybody can rename anything with this: the permission checks are left to the
2355                :  * higher-level routines.
2356                :  */
2357                : static int ext4_rename(struct inode *old_dir, struct dentry *old_dentry,
2358                :                        struct inode *new_dir, struct dentry *new_dentry)
2359             15 : {
2360                :         handle_t *handle;
2361                :         struct inode *old_inode, *new_inode;
2362                :         struct buffer_head *old_bh, *new_bh, *dir_bh;
2363                :         struct ext4_dir_entry_2 *old_de, *new_de;
2364             15 :         int retval, force_da_alloc = 0;
2365                :
2366             15 :         old_bh = new_bh = dir_bh = NULL;
2367                :
2368                :         /* Initialize quotas before so that eventual writes go
2369                :          * in separate transaction */
2370             15 :         if (new_dentry->d_inode)
2371              0 :                 vfs_dq_init(new_dentry->d_inode);
2372             60 :         handle = ext4_journal_start(old_dir, 2 *
2373                :                                     EXT4_DATA_TRANS_BLOCKS(old_dir->i_sb) +
2374                :                                     EXT4_INDEX_EXTRA_TRANS_BLOCKS + 2);
2375             15 :         if (IS_ERR(handle))
2376              0 :                 return PTR_ERR(handle);
2377                :
2378             15 :         if (IS_DIRSYNC(old_dir) || IS_DIRSYNC(new_dir))
2379                :                 ext4_handle_sync(handle);
2380                :
2381             15 :         old_bh = ext4_find_entry(old_dir, &old_dentry->d_name, &old_de);
2382                :         /*
2383                :          *  Check for inode number is _not_ due to possible IO errors.
2384                :          *  We might rmdir the source, keep it as pwd of some process
2385                :          *  and merrily kill the link to whatever was created under the
2386                :          *  same name. Goodbye sticky bit ;-<
2387                :          */
2388             15 :         old_inode = old_dentry->d_inode;
2389             15 :         retval = -ENOENT;
2390             15 :         if (!old_bh || le32_to_cpu(old_de->inode) != old_inode->i_ino)
2391                :                 goto end_rename;
2392                :
2393             15 :         new_inode = new_dentry->d_inode;
2394             15 :         new_bh = ext4_find_entry(new_dir, &new_dentry->d_name, &new_de);
2395             15 :         if (new_bh) {
2396              0 :                 if (!new_inode) {
2397              0 :                         brelse(new_bh);
2398              0 :                         new_bh = NULL;
2399                :                 }
2400                :         }
2401             15 :         if (S_ISDIR(old_inode->i_mode)) {
2402             15 :                 if (new_inode) {
2403              0 :                         retval = -ENOTEMPTY;
2404              0 :                         if (!empty_dir(new_inode))
2405              0 :                                 goto end_rename;
2406                :                 }
2407             15 :                 retval = -EIO;
2408             15 :                 dir_bh = ext4_bread(handle, old_inode, 0, 0, &retval);
2409             15 :                 if (!dir_bh)
2410              0 :                         goto end_rename;
2411             30 :                 if (le32_to_cpu(PARENT_INO(dir_bh->b_data,
2412                :                                 old_dir->i_sb->s_blocksize)) != old_dir->i_ino)
2413              0 :                         goto end_rename;
2414             15 :                 retval = -EMLINK;
```

```
2415      15 :                 if (!new_inode && new_dir != old_dir &&
2416         :                                 new_dir->i_nlink >= EXT4_LINK_MAX)
2417       0 :                         goto end_rename;
2418         :         }
2419      15 :         if (!new_bh) {
2420      15 :                 retval = ext4_add_entry(handle, new_dentry, old_inode);
2421      15 :                 if (retval)
2422       0 :                         goto end_rename;
2423         :         } else {
2424         :                 BUFFER_TRACE(new_bh, "get write access");
2425       0 :                 ext4_journal_get_write_access(handle, new_bh);
2426       0 :                 new_de->inode = cpu_to_le32(old_inode->i_ino);
2427       0 :                 if (EXT4_HAS_INCOMPAT_FEATURE(new_dir->i_sb,
2428         :                                               EXT4_FEATURE_INCOMPAT_FILETYPE))
2429       0 :                         new_de->file_type = old_de->file_type;
2430       0 :                 new_dir->i_version++;
2431       0 :                 new_dir->i_ctime = new_dir->i_mtime =
2432         :                                         ext4_current_time(new_dir);
2433       0 :                 ext4_mark_inode_dirty(handle, new_dir);
2434         :                 BUFFER_TRACE(new_bh, "call ext4_handle_dirty_metadata");
2435       0 :                 ext4_handle_dirty_metadata(handle, new_dir, new_bh);
2436       0 :                 brelse(new_bh);
2437       0 :                 new_bh = NULL;
2438         :         }
2439         :
2440         :         /*
2441         :          * Like most other Unix systems, set the ctime for inodes on a
2442         :          * rename.
2443         :          */
2444      15 :         old_inode->i_ctime = ext4_current_time(old_inode);
2445      15 :         ext4_mark_inode_dirty(handle, old_inode);
2446         :
2447         :         /*
2448         :          * ok, that's it
2449         :          */
2450      15 :         if (le32_to_cpu(old_de->inode) != old_inode->i_ino ||
2451         :             old_de->name_len != old_dentry->d_name.len ||
2452         :             strncmp(old_de->name, old_dentry->d_name.name, old_de->name_len) ||
2453         :             (retval = ext4_delete_entry(handle, old_dir,
2454         :                                         old_de, old_bh)) == -ENOENT) {
2455         :                 /* old_de could have moved from under us during htree split, so
2456         :                  * make sure that we are deleting the right entry.  We might
2457         :                  * also be pointing to a stale entry in the unused part of
2458         :                  * old_bh so just checking inum and the name isn't enough. */
2459         :                 struct buffer_head *old_bh2;
2460         :                 struct ext4_dir_entry_2 *old_de2;
2461         :
2462       0 :                 old_bh2 = ext4_find_entry(old_dir, &old_dentry->d_name, &old_de2);
2463       0 :                 if (old_bh2) {
2464       0 :                         retval = ext4_delete_entry(handle, old_dir,
2465         :                                                    old_de2, old_bh2);
2466       0 :                         brelse(old_bh2);
2467         :                 }
2468         :         }
2469      15 :         if (retval) {
2470       0 :                 ext4_warning(old_dir->i_sb, "ext4_rename",
2471         :                              "Deleting old file (%lu), %d, error=%d",
2472         :                              old_dir->i_ino, old_dir->i_nlink, retval);
2473         :         }
2474         :
2475      15 :         if (new_inode) {
2476         :                 ext4_dec_count(handle, new_inode);
2477       0 :                 new_inode->i_ctime = ext4_current_time(new_inode);
2478         :         }
2479      15 :         old_dir->i_ctime = old_dir->i_mtime = ext4_current_time(old_dir);
2480         :         ext4_update_dx_flag(old_dir);
2481      15 :         if (dir_bh) {
2482         :                 BUFFER_TRACE(dir_bh, "get_write_access");
2483      15 :                 ext4_journal_get_write_access(handle, dir_bh);
2484      30 :                 PARENT_INO(dir_bh->b_data, new_dir->i_sb->s_blocksize) =
2485         :                                                 cpu_to_le32(new_dir->i_ino);
2486         :                 BUFFER_TRACE(dir_bh, "call ext4_handle_dirty_metadata");
2487      15 :                 ext4_handle_dirty_metadata(handle, old_dir, dir_bh);
2488         :                 ext4_dec_count(handle, old_dir);
2489      15 :                 if (new_inode) {
2490         :                         /* checked empty_dir above, can't have another parent,
2491         :                          * ext4_dec_count() won't work for many-linked dirs */
2492       0 :                         new_inode->i_nlink = 0;
2493         :                 } else {
2494         :                         ext4_inc_count(handle, new_dir);
2495         :                         ext4_update_dx_flag(new_dir);
2496      15 :                         ext4_mark_inode_dirty(handle, new_dir);
2497         :                 }
2498         :         }
2499      15 :         ext4_mark_inode_dirty(handle, old_dir);
2500      15 :         if (new_inode) {
2501       0 :                 ext4_mark_inode_dirty(handle, new_inode);
2502       0 :                 if (!new_inode->i_nlink)
2503       0 :                         ext4_orphan_add(handle, new_inode);
2504       0 :                 if (!test_opt(new_dir->i_sb, NO_AUTO_DA_ALLOC))
```

```
2505        0 :                         force_da_alloc = 1;
2506          :         }
2507       15 :         retval = 0;
2508          :
2509       15 : end_rename:
2510       15 :         brelse(dir_bh);
2511       15 :         brelse(old_bh);
2512       15 :         brelse(new_bh);
2513       15 :         ext4_journal_stop(handle);
2514       15 :         if (retval == 0 && force_da_alloc)
2515        0 :                 ext4_alloc_da_blocks(old_inode);
2516       15 :         return retval;
2517          : }
2518          :
2519          : /*
2520          :  * directories can handle most operations...
2521          :  */
2522          : const struct inode_operations ext4_dir_inode_operations = {
2523          :         .create         = ext4_create,
2524          :         .lookup         = ext4_lookup,
2525          :         .link           = ext4_link,
2526          :         .unlink         = ext4_unlink,
2527          :         .symlink        = ext4_symlink,
2528          :         .mkdir          = ext4_mkdir,
2529          :         .rmdir          = ext4_rmdir,
2530          :         .mknod          = ext4_mknod,
2531          :         .rename         = ext4_rename,
2532          :         .setattr        = ext4_setattr,
2533          : #ifdef CONFIG_EXT4_FS_XATTR
2534          :         .setxattr       = generic_setxattr,
2535          :         .getxattr       = generic_getxattr,
2536          :         .listxattr      = ext4_listxattr,
2537          :         .removexattr    = generic_removexattr,
2538          : #endif
2539          :         .permission     = ext4_permission,
2540          :         .fiemap         = ext4_fiemap,
2541          : };
2542          :
2543          : const struct inode_operations ext4_special_inode_operations = {
2544          :         .setattr        = ext4_setattr,
2545          : #ifdef CONFIG_EXT4_FS_XATTR
2546          :         .setxattr       = generic_setxattr,
2547          :         .getxattr       = generic_getxattr,
2548          :         .listxattr      = ext4_listxattr,
2549          :         .removexattr    = generic_removexattr,
2550          : #endif
2551          :         .permission     = ext4_permission,
2552          : };
```