LCOV - code coverage report

			Found	Hit	Covera
Current view:	directory - fs/ext4 - dir.c (source / functions)				
Test:	kernel_2_6_31_ext4_round_3.info	**Lines:**	219	181	82.6
Date:	2009-10-24	**Functions:**	8	8	100.0

```
                :  1 /*
                :  2  *  linux/fs/ext4/dir.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/dir.c
                :  12 *
                :  13 *  Copyright (C) 1991, 1992  Linus Torvalds
                :  14 *
                :  15 *  ext4 directory handling functions
                :  16 *
                :  17 *  Big-endian to little-endian byte-swapping/bitmaps by
                :  18 *        David S. Miller (davem@caip.rutgers.edu), 1995
                :  19 *
                :  20 * Hash Tree Directory indexing (c) 2001  Daniel Phillips
                :  21 *
                :  22 */
                :  23
                :  24 #include <linux/fs.h>
                :  25 #include <linux/jbd2.h>
                :  26 #include <linux/buffer_head.h>
                :  27 #include <linux/slab.h>
                :  28 #include <linux/rbtree.h>
                :  29 #include "ext4.h"
                :  30
                :  31 static unsigned char ext4_filetype_table[] = {
                :  32         DT_UNKNOWN, DT_REG, DT_DIR, DT_CHR, DT_BLK, DT_FIFO, DT_SOCK, DT_LNK
                :  33 };
                :  34
                :  35 static int ext4_readdir(struct file *, void *, filldir_t);
                :  36 static int ext4_dx_readdir(struct file *filp,
                :  37                            void *dirent, filldir_t filldir);
                :  38 static int ext4_release_dir(struct inode *inode,
                :  39                              struct file *filp);
                :  40
                :  41 const struct file_operations ext4_dir_operations = {
                :  42         .llseek         = generic_file_llseek,
                :  43         .read           = generic_read_dir,
                :  44         .readdir        = ext4_readdir,         /* we take BKL. needed?*/
                :  45         .unlocked_ioctl = ext4_ioctl,
                :  46 #ifdef CONFIG_COMPAT
                :  47         .compat_ioctl   = ext4_compat_ioctl,
                :  48 #endif
                :  49         .fsync          = ext4_sync_file,
                :  50         .release        = ext4_release_dir,
                :  51 };
                :  52
                :  53
                :  54 static unsigned char get_dtype(struct super_block *sb, int filetype)
                :  55 {
          1438522 :         if (!EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_FILETYPE) ||
                :  57             (filetype >= EXT4_FT_MAX))
```

```
    58                0 :                         return DT_UNKNOWN;
    59                  :
    60          1438522 :                 return (ext4_filetype_table[filetype]);
    61                  : }
    62                  :
    63                  :
    64                  : int ext4_check_dir_entry(const char *function, struct inode *dir,
    65                  :                          struct ext4_dir_entry_2 *de,
    66                  :                          struct buffer_head *bh,
    67                  :                          unsigned int offset)
    68 -1921647951      : {
    69 -1921647951      :         const char *error_msg = NULL;
    70                  :         const int rlen = ext4_rec_len_from_disk(de->rec_len,
    71 -1921647951      :                                                 dir->i_sb->s_blocksize);
    72                  :
    73 -1921647851      :         if (rlen < EXT4_DIR_REC_LEN(1))
    74                0 :                 error_msg = "rec_len is smaller than minimal";
    75 -1921647851      :         else if (rlen % 4 != 0)
    76                0 :                 error_msg = "rec_len % 4 != 0";
    77 -1921647851      :         else if (rlen < EXT4_DIR_REC_LEN(de->name_len))
    78                0 :                 error_msg = "rec_len is too small for name_len";
    79 -1921647851      :         else if (((char *) de - bh->b_data) + rlen > dir->i_sb->s_blocksize)
    80                0 :                 error_msg = "directory entry across blocks";
    81               -1 :         else if (le32_to_cpu(de->inode) >
    82                  :                  le32_to_cpu(EXT4_SB(dir->i_sb)->s_es->s_inodes_count))
    83                0 :                 error_msg = "inode out of bounds";
    84                  :
    85 -1921647851      :         if (error_msg != NULL)
    86                0 :                 ext4_error(dir->i_sb, function,
    87                  :                            "bad entry in directory #%lu: %s - "
    88                  :                            "offset=%u, inode=%u, rec_len=%d, name_len=%d",
    89                  :                            dir->i_ino, error_msg, offset,
    90                  :                            le32_to_cpu(de->inode),
    91                  :                            rlen, de->name_len);
    92 -1921647851      :         return error_msg == NULL ? 1 : 0;
    93                  : }
    94                  :
    95                  : static int ext4_readdir(struct file *filp,
    96                  :                         void *dirent, filldir_t filldir)
    97           53646 : {
    98           53646 :         int error = 0;
    99                  :         unsigned int offset;
   100                  :         int i, stored;
   101                  :         struct ext4_dir_entry_2 *de;
   102                  :         struct super_block *sb;
   103                  :         int err;
   104           53646 :         struct inode *inode = filp->f_path.dentry->d_inode;
   105           53646 :         int ret = 0;
   106           53646 :         int dir_has_error = 0;
   107                  :
   108           53646 :         sb = inode->i_sb;
   109                  :
   110          160938 :         if (EXT4_HAS_COMPAT_FEATURE(inode->i_sb,
   111                  :                                     EXT4_FEATURE_COMPAT_DIR_INDEX) &&
   112                  :             ((EXT4_I(inode)->i_flags & EXT4_INDEX_FL) ||
   113                  :              ((inode->i_size >> sb->s_blocksize_bits) == 1))) {
   114           53634 :                 err = ext4_dx_readdir(filp, dirent, filldir);
   115           53634 :                 if (err != ERR_BAD_DX_DIR) {
   116           53634 :                         ret = err;
   117           53634 :                         goto out;
   118                  :                 }
   119                  :                 /*
   120                  :                  * We don't set the inode dirty flag since it's not
   121                  :                  * critical that it get flushed back to the disk.
   122                  :                  */
   123                0 :                 EXT4_I(filp->f_path.dentry->d_inode)->i_flags &= ~EXT4_INDEX_FL;
   124                  :         }
   125              12 :         stored = 0;
   126              12 :         offset = filp->f_pos & (sb->s_blocksize - 1);
   127                  :
   128              48 :         while (!error && !stored && filp->f_pos < inode->i_size) {
```

```
   129              24 :                  ext4_lblk_t blk = filp->f_pos >> EXT4_BLOCK_SIZE_BITS(sb);
   130                 :                  struct buffer_head map_bh;
   131              24 :                  struct buffer_head *bh = NULL;
   132                 :
   133              24 :                  map_bh.b_state = 0;
   134              24 :                  err = ext4_get_blocks(NULL, inode, blk, 1, &map_bh, 0);
   135              24 :                  if (err > 0) {
   136                 :                          pgoff_t index = map_bh.b_blocknr >>
   137              24 :                                          (PAGE_CACHE_SHIFT - inode->i_blkbits);
   138              48 :                          if (!ra_has_index(&filp->f_ra, index))
   139               6 :                                  page_cache_sync_readahead(
   140                 :                                          sb->s_bdev->bd_inode->i_mapping,
   141                 :                                          &filp->f_ra, filp,
   142                 :                                          index, 1);
   143              24 :                          filp->f_ra.prev_pos = (loff_t)index << PAGE_CACHE_SHIFT;
   144              24 :                          bh = ext4_bread(NULL, inode, blk, 0, &err);
   145                 :                  }
   146                 :
   147                 :                  /*
   148                 :                   * We ignore I/O errors on directories so users have a chance
   149                 :                   * of recovering data when there's a bad sector
   150                 :                   */
   151              24 :                  if (!bh) {
   152               0 :                          if (!dir_has_error) {
   153               0 :                                  ext4_error(sb, __func__, "directory #%lu "
   154                 :                                             "contains a hole at offset %Lu",
   155                 :                                             inode->i_ino,
   156                 :                                             (unsigned long long) filp->f_pos);
   157               0 :                                  dir_has_error = 1;
   158                 :                          }
   159                 :                          /* corrupt size?  Maybe no more blocks to read */
   160               0 :                          if (filp->f_pos > inode->i_blocks << 9)
   161               0 :                                  break;
   162               0 :                          filp->f_pos += sb->s_blocksize - offset;
   163               0 :                          continue;
   164                 :                  }
   165                 :
   166              24 : revalidate:
   167                 :                  /* If the dir block has changed since the last call to
   168                 :                   * readdir(2), then we might be pointing to an invalid
   169                 :                   * dirent right now.  Scan from the start of the block
   170                 :                   * to make sure. */
   171              24 :                  if (filp->f_version != inode->i_version) {
   172               0 :                          for (i = 0; i < sb->s_blocksize && i < offset; ) {
   173               0 :                                  de = (struct ext4_dir_entry_2 *)
   174                 :                                          (bh->b_data + i);
   175                 :                                  /* It's too expensive to do a full
   176                 :                                   * dirent test each time round this
   177                 :                                   * loop, but we do have to test at
   178                 :                                   * least that it is non-zero.  A
   179                 :                                   * failure will be detected in the
   180                 :                                   * dirent test below. */
   181               0 :                                  if (ext4_rec_len_from_disk(de->rec_len,
   182                 :                                          sb->s_blocksize) < EXT4_DIR_REC_LEN(1))
   183               0 :                                          break;
   184               0 :                                  i += ext4_rec_len_from_disk(de->rec_len,
   185                 :                                                             sb->s_blocksize);
   186                 :                          }
   187               0 :                          offset = i;
   188               0 :                          filp->f_pos = (filp->f_pos & ~(sb->s_blocksize - 1))
   189                 :                                  | offset;
   190               0 :                          filp->f_version = inode->i_version;
   191                 :                  }
   192                 :
   193              54 :                  while (!error && filp->f_pos < inode->i_size
   194                 :                         && offset < sb->s_blocksize) {
   195              30 :                          de = (struct ext4_dir_entry_2 *) (bh->b_data + offset);
   196              30 :                          if (!ext4_check_dir_entry("ext4_readdir", inode, de,
   197                 :                                                    bh, offset)) {
   198                 :                                  /*
   199                 :                                   * On error, skip the f_pos to the next block
```

```
   200                :                                         */
   201                0 :                                        filp->f_pos = (filp->f_pos |
   202                  :                                                      (sb->s_blocksize - 1)) + 1;
   203                  :                                        brelse(bh);
   204                0 :                                        ret = stored;
   205                0 :                                        goto out;
   206                  :                                }
   207               30 :                                offset += ext4_rec_len_from_disk(de->rec_len,
   208                  :                                                  sb->s_blocksize);
   209               30 :                                if (le32_to_cpu(de->inode)) {
   210                  :                                        /* We might block in the next section
   211                  :                                         * if the data destination is
   212                  :                                         * currently swapped out.  So, use a
   213                  :                                         * version stamp to detect whether or
   214                  :                                         * not the directory has been modified
   215                  :                                         * during the copy operation.
   216                  :                                         */
   217               12 :                                        u64 version = filp->f_version;
   218                  :
   219               24 :                                        error = filldir(dirent, de->name,
   220                  :                                                        de->name_len,
   221                  :                                                        filp->f_pos,
   222                  :                                                        le32_to_cpu(de->inode),
   223                  :                                                        get_dtype(sb, de->file_type));
   224               12 :                                        if (error)
   225                0 :                                                break;
   226               12 :                                        if (version != filp->f_version)
   227                0 :                                                goto revalidate;
   228               12 :                                        stored++;
   229                  :                                }
   230               30 :                                filp->f_pos += ext4_rec_len_from_disk(de->rec_len,
   231                  :                                                        sb->s_blocksize);
   232                  :                        }
   233               24 :                        offset = 0;
   234                  :                        brelse(bh);
   235                  :                }
   236            53646 : out:
   237            53646 :                 return ret;
   238                  : }
   239                  :
   240                  : /*
   241                  :  * These functions convert from the major/minor hash to an f_pos
   242                  :  * value.
   243                  :  *
   244                  :  * Currently we only use major hash numer.  This is unfortunate, but
   245                  :  * on 32-bit machines, the same VFS interface is used for lseek and
   246                  :  * llseek, so if we use the 64 bit offset, then the 32-bit versions of
   247                  :  * lseek/telldir/seekdir will blow out spectacularly, and from within
   248                  :  * the ext2 low-level routine, we don't know if we're being called by
   249                  :  * a 64-bit version of the system call or the 32-bit version of the
   250                  :  * system call.  Worse yet, NFSv2 only allows for a 32-bit readdir
   251                  :  * cookie.  Sigh.
   252                  :  */
   253                  : #define hash2pos(major, minor)  (major >> 1)
   254                  : #define pos2maj_hash(pos)       ((pos << 1) & 0xffffffff)
   255                  : #define pos2min_hash(pos)       (0)
   256                  :
   257                  : /*
   258                  :  * This structure holds the nodes of the red-black tree used to store
   259                  :  * the directory entry in hash order.
   260                  :  */
   261                  : struct fname {
   262                  :         __u32           hash;
   263                  :         __u32           minor_hash;
   264                  :         struct rb_node  rb_hash;
   265                  :         struct fname    *next;
   266                  :         __u32           inode;
   267                  :         __u8            name_len;
   268                  :         __u8            file_type;
   269                  :         char            name[0];
   270                  : };
```

```
   271                :
   272                : /*
   273                :  * This functoin implements a non-recursive way of freeing all of the
   274                :  * nodes in the red-black tree.
   275                :  */
   276                : static void free_rb_tree_fname(struct rb_root *root)
   277          75083 : {
   278          75083 :         struct rb_node  *n = root->rb_node;
   279                :         struct rb_node  *parent;
   280                :         struct fname    *fname;
   281                :
   282        3651255 :         while (n) {
   283                :                 /* Do the node's children first */
   284        3501089 :                 if (n->rb_left) {
   285         844829 :                         n = n->rb_left;
   286         844829 :                         continue;
   287                :                 }
   288        2656260 :                 if (n->rb_right) {
   289         868198 :                         n = n->rb_right;
   290         868198 :                         continue;
   291                :                 }
   292                :                 /*
   293                :                  * The node has no children; free it, and then zero
   294                :                  * out parent's link to it.  Finally go to the
   295                :                  * beginning of the loop and try to free the parent
   296                :                  * node.
   297                :                  */
   298        1788062 :                 parent = rb_parent(n);
   299        1788062 :                 fname = rb_entry(n, struct fname, rb_hash);
   300        5364186 :                 while (fname) {
   301        1788062 :                         struct fname *old = fname;
   302        1788062 :                         fname = fname->next;
   303        1788062 :                         kfree(old);
   304                :                 }
   305        1788062 :                 if (!parent)
   306          75035 :                         root->rb_node = NULL;
   307        1713027 :                 else if (parent->rb_left == n)
   308         844829 :                         parent->rb_left = NULL;
   309         868198 :                 else if (parent->rb_right == n)
   310         868198 :                         parent->rb_right = NULL;
   311        1788062 :                 n = parent;
   312                :         }
   313          75083 : }
   314                :
   315                :
   316                : static struct dir_private_info *ext4_htree_create_dir_info(loff_t pos)
   317                : {
   318                :         struct dir_private_info *p;
   319                :
   320             42 :         p = kzalloc(sizeof(struct dir_private_info), GFP_KERNEL);
   321             42 :         if (!p)
   322              0 :                 return NULL;
   323             42 :         p->curr_hash = pos2maj_hash(pos);
   324             42 :         p->curr_minor_hash = pos2min_hash(pos);
   325             42 :         return p;
   326                : }
   327                :
   328                : void ext4_htree_free_dir_info(struct dir_private_info *p)
   329             42 : {
   330             42 :         free_rb_tree_fname(&p->root);
   331             42 :         kfree(p);
   332             42 : }
   333                :
   334                : /*
   335                :  * Given a directory entry, enter it into the fname rb tree.
   336                :  */
   337                : int ext4_htree_store_dirent(struct file *dir_file, __u32 hash,
   338                :                             __u32 minor_hash,
   339                :                             struct ext4_dir_entry_2 *dirent)
   340        1788062 : {
   341        1788062 :         struct rb_node **p, *parent = NULL;
```

```
      :         struct fname *fname, *new_fn;
      :         struct dir_private_info *info;
      :         int len;
      :
1788062 :         info = (struct dir_private_info *) dir_file->private_data;
1788062 :         p = &info->root.rb_node;
      :
      :         /* Create and allocate the fname structure */
1788062 :         len = sizeof(struct fname) + dirent->name_len + 1;
1788062 :         new_fn = kzalloc(len, GFP_KERNEL);
1788062 :         if (!new_fn)
      0 :                 return -ENOMEM;
1788062 :         new_fn->hash = hash;
1788062 :         new_fn->minor_hash = minor_hash;
1788062 :         new_fn->inode = le32_to_cpu(dirent->inode);
1788062 :         new_fn->name_len = dirent->name_len;
1788062 :         new_fn->file_type = dirent->file_type;
3576124 :         memcpy(new_fn->name, dirent->name, dirent->name_len);
1788062 :         new_fn->name[dirent->name_len] = 0;
      :
13909268 :         while (*p) {
10333144 :                 parent = *p;
10333144 :                 fname = rb_entry(parent, struct fname, rb_hash);
      :
      :                 /*
      :                  * If the hash and minor hash match up, then we put
      :                  * them on a linked list.  This rarely happens...
      :                  */
10333144 :                 if ((new_fn->hash == fname->hash) &&
      :                     (new_fn->minor_hash == fname->minor_hash)) {
      0 :                         new_fn->next = fname->next;
      0 :                         fname->next = new_fn;
      0 :                         return 0;
      :                 }
      :
10333144 :                 if (new_fn->hash < fname->hash)
 2332787 :                         p = &(*p)->rb_left;
 8000357 :                 else if (new_fn->hash > fname->hash)
 8000294 :                         p = &(*p)->rb_right;
      63 :                 else if (new_fn->minor_hash < fname->minor_hash)
      33 :                         p = &(*p)->rb_left;
      :                 else /* if (new_fn->minor_hash > fname->minor_hash) */
      30 :                         p = &(*p)->rb_right;
      :         }
      :
1788062 :         rb_link_node(&new_fn->rb_hash, parent, p);
1788062 :         rb_insert_color(&new_fn->rb_hash, &info->root);
1788062 :         return 0;
      : }
      :
      :
      :
      : /*
      :  * This is a helper function for ext4_dx_readdir.  It calls filldir
      :  * for all entres on the fname linked list.  (Normally there is only
      :  * one entry on the linked list, unless there are 62 bit hash collisions.)
      :  */
      : static int call_filldir(struct file *filp, void *dirent,
      :                         filldir_t filldir, struct fname *fname)
1438510 : {
1438510 :         struct dir_private_info *info = filp->private_data;
      :         loff_t  curr_pos;
1438510 :         struct inode *inode = filp->f_path.dentry->d_inode;
      :         struct super_block *sb;
      :         int error;
      :
1438510 :         sb = inode->i_sb;
      :
1438510 :         if (!fname) {
      0 :                 printk(KERN_ERR "EXT4-fs: call_filldir: called with "
      :                        "null fname?!?\n");
```

```
    413                0 :                 return 0;
    414                  :         }
    415          1438510 :         curr_pos = hash2pos(fname->hash, fname->minor_hash);
    416          4261980 :         while (fname) {
    417          2877020 :                 error = filldir(dirent, fname->name,
    418                  :                                 fname->name_len, curr_pos,
    419                  :                                 fname->inode,
    420                  :                                 get_dtype(sb, fname->file_type));
    421          1438510 :                 if (error) {
    422            53550 :                         filp->f_pos = curr_pos;
    423            53550 :                         info->extra_fname = fname;
    424            53550 :                         return error;
    425                  :                 }
    426          1384960 :                 fname = fname->next;
    427                  :         }
    428          1384960 :         return 0;
    429                  : }
    430                  :
    431                  : static int ext4_dx_readdir(struct file *filp,
    432                  :                            void *dirent, filldir_t filldir)
    433            53634 : {
    434            53634 :         struct dir_private_info *info = filp->private_data;
    435            53634 :         struct inode *inode = filp->f_path.dentry->d_inode;
    436                  :         struct fname *fname;
    437                  :         int     ret;
    438                  :
    439            53634 :         if (!info) {
    440               84 :                 info = ext4_htree_create_dir_info(filp->f_pos);
    441               42 :                 if (!info)
    442                0 :                         return -ENOMEM;
    443               42 :                 filp->private_data = info;
    444                  :         }
    445                  :
    446            53634 :         if (filp->f_pos == EXT4_HTREE_EOF)
    447               42 :                 return 0;       /* EOF */
    448                  :
    449                  :         /* Some one has messed with f_pos; reset the world */
    450            53592 :         if (info->last_pos != filp->f_pos) {
    451                3 :                 free_rb_tree_fname(&info->root);
    452                3 :                 info->curr_node = NULL;
    453                3 :                 info->extra_fname = NULL;
    454                3 :                 info->curr_hash = pos2maj_hash(filp->f_pos);
    455                3 :                 info->curr_minor_hash = pos2min_hash(filp->f_pos);
    456                  :         }
    457                  :
    458                  :         /*
    459                  :          * If there are any leftover names on the hash collision
    460                  :          * chain, return them first.
    461                  :          */
    462            53592 :         if (info->extra_fname) {
    463            53547 :                 if (call_filldir(filp, dirent, filldir, info->extra_fname))
    464                0 :                         goto finished;
    465            53547 :                 info->extra_fname = NULL;
    466            53547 :                 goto next_node;
    467               45 :         } else if (!info->curr_node)
    468               45 :                 info->curr_node = rb_first(&info->root);
    469                  :
    470                  :         while (1) {
    471                  :                 /*
    472                  :                  * Fill the rbtree if we have no more entries,
    473                  :                  * or the inode has changed since we last read in the
    474                  :                  * cached entries.
    475                  :                  */
    476          1384966 :                 if ((!info->curr_node) ||
    477                  :                     (filp->f_version != inode->i_version)) {
    478            75038 :                         info->curr_node = NULL;
    479            75038 :                         free_rb_tree_fname(&info->root);
    480            75038 :                         filp->f_version = inode->i_version;
    481            75038 :                         ret = ext4_htree_fill_tree(filp, info->curr_hash,
    482                  :                                                    info->curr_minor_hash,
    483                  :                                                    &info->next_hash);
```

```
   484          75038 :                         if (ret < 0)
   485              0 :                                 return ret;
   486          75038 :                         if (ret == 0) {
   487              3 :                                 filp->f_pos = EXT4_HTREE_EOF;
   488              3 :                                 break;
   489                :                         }
   490          75035 :                         info->curr_node = rb_first(&info->root);
   491                :                 }
   492                :
   493        1384963 :                 fname = rb_entry(info->curr_node, struct fname, rb_hash);
   494        1384963 :                 info->curr_hash = fname->hash;
   495        1384963 :                 info->curr_minor_hash = fname->minor_hash;
   496        1384963 :                 if (call_filldir(filp, dirent, filldir, fname))
   497          53550 :                         break;
   498        1384960 :         next_node:
   499        1384960 :                 info->curr_node = rb_next(info->curr_node);
   500        1384960 :                 if (info->curr_node) {
   501        1316563 :                         fname = rb_entry(info->curr_node, struct fname,
   502                :                                          rb_hash);
   503        1316563 :                         info->curr_hash = fname->hash;
   504        1316563 :                         info->curr_minor_hash = fname->minor_hash;
   505                :                 } else {
   506          68397 :                         if (info->next_hash == ~0) {
   507             39 :                                 filp->f_pos = EXT4_HTREE_EOF;
   508             39 :                                 break;
   509                :                         }
   510          68358 :                         info->curr_hash = info->next_hash;
   511          68358 :                         info->curr_minor_hash = 0;
   512                :                 }
   513                :         }
   514          53592 : finished:
   515          53592 :         info->last_pos = filp->f_pos;
   516          53592 :         return 0;
   517                : }
   518                :
   519                : static int ext4_release_dir(struct inode *inode, struct file *filp)
   520             60 : {
   521             60 :         if (filp->private_data)
   522             42 :                 ext4_htree_free_dir_info(filp->private_data);
   523                :
   524             60 :         return 0;
   525                : }
```