LCOV - code coverage report

Current view: directory - fs/ext4 - ialloc.c (source / functions)		Found	Hit	Coverage
Test: kernel_2_6_31_ext4_round_3.info	Lines:	555	354	63.8 %
Date: 2009-10-24	Functions:	13	11	84.6 %

```
                :  /*
                :   *  linux/fs/ext4/ialloc.c
                :   *
                :   * Copyright (C) 1992, 1993, 1994, 1995
                :   * Remy Card (card@masi.ibp.fr)
                :   * Laboratoire MASI - Institut Blaise Pascal
                :   * Universite Pierre et Marie Curie (Paris VI)
                :   *
                :   *  BSD ufs-inspired inode and directory allocation by
                :   *  Stephen Tweedie (sct@redhat.com), 1993
                :   *  Big-endian to little-endian byte-swapping/bitmaps by
                :   *        David S. Miller (davem@caip.rutgers.edu), 1995
                :   */
                :
                :  #include <linux/time.h>
                :  #include <linux/fs.h>
                :  #include <linux/jbd2.h>
                :  #include <linux/stat.h>
                :  #include <linux/string.h>
                :  #include <linux/quotaops.h>
                :  #include <linux/buffer_head.h>
                :  #include <linux/random.h>
                :  #include <linux/bitops.h>
                :  #include <linux/blkdev.h>
                :  #include <asm/byteorder.h>
                :
                :  #include "ext4.h"
                :  #include "ext4_jbd2.h"
                :  #include "xattr.h"
                :  #include "acl.h"
                :
                :  #include <trace/events/ext4.h>
                :
                :  /*
                :   * ialloc.c contains the inodes allocation and deallocation routines
                :   */
                :
                :  /*
                :   * The free inodes are managed by bitmaps.  A file system contains several
                :   * blocks groups.  Each group contains 1 bitmap block for blocks, 1 bitmap
                :   * block for inodes, N blocks for the inode table and data blocks.
                :   *
                :   * The file system contains group descriptors which are located after the
                :   * super block.  Each descriptor contains the number of the bitmap block and
                :   * the free blocks count in the block.
                :   */
                :
                :  /*
                :   * To avoid calling the atomic setbit hundreds or thousands of times, we only
                :   * need to use it within a single byte (to ensure we get endianness right).
                :   * We can use memset for the rest of the bitmap as there are no other users.
                :   */
                :  void mark_bitmap_end(int start_bit, int end_bit, char *bitmap)
         203259 :  {
                :          int i;
                :
         203259 :          if (start_bit >= end_bit)
         112720 :                  return;
```

```
     59                 :
     60                 :         ext4_debug("mark end bits +%d through +%d used\n", start_bit, end_bit);
     61          90539  :         for (i = start_bit; i < ((start_bit + 7) & ~7UL); i++)
     62              0  :                 ext4_set_bit(i, bitmap);
     63          90539  :         if (i < end_bit)
     64         181078  :                 memset(bitmap + (i >> 3), 0xff, (end_bit - i) >> 3);
     65                 : }
     66                 :
     67                 : /* Initializes an uninitialized inode bitmap */
     68                 : unsigned ext4_init_inode_bitmap(struct super_block *sb, struct buffer_head *bh,
     69                 :                                 ext4_group_t block_group,
     70                 :                                 struct ext4_group_desc *gdp)
     71          90539  : {
     72          90539  :         struct ext4_sb_info *sbi = EXT4_SB(sb);
     73                 :
     74          90539  :         J_ASSERT_BH(bh, buffer_locked(bh));
     75                 :
     76                 :         /* If checksum is bad mark all blocks and inodes use to prevent
     77                 :          * allocation, essentially implementing a per-group read-only flag. */
     78          90539  :         if (!ext4_group_desc_csum_verify(sbi, block_group, gdp)) {
     79              0  :                 ext4_error(sb, __func__, "Checksum bad for group %u",
     80                 :                            block_group);
     81              0  :                 ext4_free_blks_set(sb, gdp, 0);
     82              0  :                 ext4_free_inodes_set(sb, gdp, 0);
     83              0  :                 ext4_itable_unused_set(sb, gdp, 0);
     84              0  :                 memset(bh->b_data, 0xff, sb->s_blocksize);
     85              0  :                 return 0;
     86                 :         }
     87                 :
     88          90539  :         memset(bh->b_data, 0, (EXT4_INODES_PER_GROUP(sb) + 7) / 8);
     89         181078  :         mark_bitmap_end(EXT4_INODES_PER_GROUP(sb), sb->s_blocksize * 8,
     90                 :                         bh->b_data);
     91                 :
     92          90539  :         return EXT4_INODES_PER_GROUP(sb);
     93                 : }
     94                 :
     95                 : /*
     96                 :  * Read the inode allocation bitmap for a given block_group, reading
     97                 :  * into the specified slot in the superblock's bitmap cache.
     98                 :  *
     99                 :  * Return buffer_head of bitmap on success or NULL.
    100                 :  */
    101                 : static struct buffer_head *
    102                 : ext4_read_inode_bitmap(struct super_block *sb, ext4_group_t block_group)
    103       10976924  : {
    104                 :         struct ext4_group_desc *desc;
    105       10976924  :         struct buffer_head *bh = NULL;
    106                 :         ext4_fsblk_t bitmap_blk;
    107                 :
    108       10976924  :         desc = ext4_get_group_desc(sb, block_group, NULL);
    109       10976925  :         if (!desc)
    110              0  :                 return NULL;
    111       10976925  :         bitmap_blk = ext4_inode_bitmap(sb, desc);
    112       10976926  :         bh = sb_getblk(sb, bitmap_blk);
    113       10976926  :         if (unlikely(!bh)) {
    114              0  :                 ext4_error(sb, __func__,
    115                 :                            "Cannot read inode bitmap - "
    116                 :                            "block_group = %u, inode_bitmap = %llu",
    117                 :                            block_group, bitmap_blk);
    118              0  :                 return NULL;
    119                 :         }
    120       10976922  :         if (bitmap_uptodate(bh))
    121       10801699  :                 return bh;
    122                 :
    123                 :         lock_buffer(bh);
    124         175227  :         if (bitmap_uptodate(bh)) {
    125              0  :                 unlock_buffer(bh);
    126              0  :                 return bh;
    127                 :         }
    128                 :         ext4_lock_group(sb, block_group);
    129         175227  :         if (desc->bg_flags & cpu_to_le16(EXT4_BG_INODE_UNINIT)) {
```

```
   130          90539 :                  ext4_init_inode_bitmap(sb, bh, block_group, desc);
   131                :                  set_bitmap_uptodate(bh);
   132                :                  set_buffer_uptodate(bh);
   133                :                  ext4_unlock_group(sb, block_group);
   134          90539 :                  unlock_buffer(bh);
   135          90539 :                  return bh;
   136                :          }
   137                :          ext4_unlock_group(sb, block_group);
   138          84688 :          if (buffer_uptodate(bh)) {
   139                :                  /*
   140                :                   * if not uninit if bh is uptodate,
   141                :                   * bitmap is also uptodate
   142                :                   */
   143                :                  set_bitmap_uptodate(bh);
   144              0 :                  unlock_buffer(bh);
   145              0 :                  return bh;
   146                :          }
   147                :          /*
   148                :           * submit the buffer_head for read. We can
   149                :           * safely mark the bitmap as uptodate now.
   150                :           * We do it here so the bitmap uptodate bit
   151                :           * get set with buffer lock held.
   152                :           */
   153                :          set_bitmap_uptodate(bh);
   154          84688 :          if (bh_submit_read(bh) < 0) {
   155                :                  put_bh(bh);
   156              0 :                  ext4_error(sb, __func__,
   157                :                             "Cannot read inode bitmap - "
   158                :                             "block_group = %u, inode_bitmap = %llu",
   159                :                             block_group, bitmap_blk);
   160              0 :                  return NULL;
   161                :          }
   162          84688 :          return bh;
   163                : }
   164                :
   165                : /*
   166                :  * NOTE! When we get the inode, we're the only people
   167                :  * that have access to it, and as such there are no
   168                :  * race conditions we have to worry about. The inode
   169                :  * is not on the hash-lists, and it cannot be reached
   170                :  * through the filesystem because the directory entry
   171                :  * has been deleted earlier.
   172                :  *
   173                :  * HOWEVER: we must make sure that we get no aliases,
   174                :  * which means that we have to call "clear_inode()"
   175                :  * _before_ we mark the inode not in use in the inode
   176                :  * bitmaps. Otherwise a newly created file might use
   177                :  * the same inode number (not actually the same pointer
   178                :  * though), and then we'd have two inodes sharing the
   179                :  * same inode number and space on the harddisk.
   180                :  */
   181                : void ext4_free_inode(handle_t *handle, struct inode *inode)
   182       1664354 : {
   183       1664354 :          struct super_block *sb = inode->i_sb;
   184                :          int is_directory;
   185                :          unsigned long ino;
   186       1664354 :          struct buffer_head *bitmap_bh = NULL;
   187                :          struct buffer_head *bh2;
   188                :          ext4_group_t block_group;
   189                :          unsigned long bit;
   190                :          struct ext4_group_desc *gdp;
   191                :          struct ext4_super_block *es;
   192                :          struct ext4_sb_info *sbi;
   193       1664354 :          int fatal = 0, err, count, cleared;
   194                :
   195       3328708 :          if (atomic_read(&inode->i_count) > 1) {
   196              0 :                  printk(KERN_ERR "ext4_free_inode: inode has count=%d\n",
   197                :                                  atomic_read(&inode->i_count));
   198              0 :                  return;
   199                :          }
   200       1664354 :          if (inode->i_nlink) {
```

```
   201                0 :                printk(KERN_ERR "ext4_free_inode: inode has nlink=%d\n",
   202                  :                       inode->i_nlink);
   203                0 :                return;
   204                  :        }
   205          1664354 :        if (!sb) {
   206                0 :                printk(KERN_ERR "ext4_free_inode: inode on "
   207                  :                       "nonexistent device\n");
   208                0 :                return;
   209                  :        }
   210          1664354 :        sbi = EXT4_SB(sb);
   211                  :
   212          1664354 :        ino = inode->i_ino;
   213                  :        ext4_debug("freeing inode %lu\n", ino);
   214                  :        trace_ext4_free_inode(inode);
   215                  :
   216                  :        /*
   217                  :         * Note: we must free any quota before locking the superblock,
   218                  :         * as writing the quota to disk may need the lock as well.
   219                  :         */
   220          1664354 :        vfs_dq_init(inode);
   221          1664354 :        ext4_xattr_delete_inode(handle, inode);
   222          1664355 :        vfs_dq_free_inode(inode);
   223          1664355 :        vfs_dq_drop(inode);
   224                  :
   225          1664355 :        is_directory = S_ISDIR(inode->i_mode);
   226                  :
   227                  :        /* Do this BEFORE marking the inode not in use or returning an error */
   228          1664355 :        clear_inode(inode);
   229                  :
   230          1664355 :        es = EXT4_SB(sb)->s_es;
   231          1664355 :        if (ino < EXT4_FIRST_INO(sb) || ino > le32_to_cpu(es->s_inodes_count)) {
   232                2 :                ext4_error(sb, "ext4_free_inode",
   233                  :                           "reserved or nonexistent inode %lu", ino);
   234                0 :                goto error_return;
   235                  :        }
   236          3328706 :        block_group = (ino - 1) / EXT4_INODES_PER_GROUP(sb);
   237          3328706 :        bit = (ino - 1) % EXT4_INODES_PER_GROUP(sb);
   238          1664353 :        bitmap_bh = ext4_read_inode_bitmap(sb, block_group);
   239          1664355 :        if (!bitmap_bh)
   240                0 :                goto error_return;
   241                  :
   242                  :        BUFFER_TRACE(bitmap_bh, "get_write_access");
   243          1664355 :        fatal = ext4_journal_get_write_access(handle, bitmap_bh);
   244          1664354 :        if (fatal)
   245                0 :                goto error_return;
   246                  :
   247                  :        /* Ok, now we can actually update the inode bitmaps.. */
   248          3328709 :        cleared = ext4_clear_bit_atomic(ext4_group_lock_ptr(sb, block_group),
   249                  :                                        bit, bitmap_bh->b_data);
   250          1664355 :        if (!cleared)
   251                0 :                ext4_error(sb, "ext4_free_inode",
   252                  :                           "bit already cleared for inode %lu", ino);
   253                  :        else {
   254          1664355 :                gdp = ext4_get_group_desc(sb, block_group, &bh2);
   255                  :
   256                  :                BUFFER_TRACE(bh2, "get_write_access");
   257          1664351 :                fatal = ext4_journal_get_write_access(handle, bh2);
   258          1664354 :                if (fatal) goto error_return;
   259                  :
   260          1664354 :                if (gdp) {
   261                  :                        ext4_lock_group(sb, block_group);
   262          1664355 :                        count = ext4_free_inodes_count(sb, gdp) + 1;
   263          1664355 :                        ext4_free_inodes_set(sb, gdp, count);
   264          1664355 :                        if (is_directory) {
   265           655410 :                                count = ext4_used_dirs_count(sb, gdp) - 1;
   266           655410 :                                ext4_used_dirs_set(sb, gdp, count);
   267           655410 :                                if (sbi->s_log_groups_per_flex) {
   268                  :                                        ext4_group_t f;
   269                  :
   270           655410 :                                        f = ext4_flex_group(sbi, block_group);
   271           655410 :                                        atomic_dec(&sbi->s_flex_groups[f].free_inodes);
```

```
272                 :                                       }
273                 :
274                 :                               }
275         1664355 :                               gdp->bg_checksum = ext4_group_desc_csum(sbi,
276                 :                                                                       block_group, gdp);
277                 :                               ext4_unlock_group(sb, block_group);
278         1664354 :                               percpu_counter_inc(&sbi->s_freeinodes_counter);
279         1664353 :                               if (is_directory)
280          655410 :                                       percpu_counter_dec(&sbi->s_dirs_counter);
281                 :
282         1664354 :                               if (sbi->s_log_groups_per_flex) {
283                 :                                       ext4_group_t f;
284                 :
285         1664354 :                                       f = ext4_flex_group(sbi, block_group);
286         1664354 :                                       atomic_inc(&sbi->s_flex_groups[f].free_inodes);
287                 :                               }
288                 :                       }
289                 :                       BUFFER_TRACE(bh2, "call ext4_handle_dirty_metadata");
290         1664355 :                       err = ext4_handle_dirty_metadata(handle, NULL, bh2);
291         1664355 :                       if (!fatal) fatal = err;
292                 :               }
293                 :               BUFFER_TRACE(bitmap_bh, "call ext4_handle_dirty_metadata");
294         1664355 :               err = ext4_handle_dirty_metadata(handle, NULL, bitmap_bh);
295         1664354 :               if (!fatal)
296         1664355 :                       fatal = err;
297         1664354 :               sb->s_dirt = 1;
298         1664354 : error_return:
299                 :               brelse(bitmap_bh);
300         1664355 :               ext4_std_error(sb, fatal);
301                 : }
302                 :
303                 : /*
304                 :  * There are two policies for allocating an inode.  If the new inode is
305                 :  * a directory, then a forward search is made for a block group with both
306                 :  * free space and a low directory-to-inode ratio; if that fails, then of
307                 :  * the groups with above-average free space, that group with the fewest
308                 :  * directories already is chosen.
309                 :  *
310                 :  * For other inodes, search forward from the parent directory\'s block
311                 :  * group to find a free inode.
312                 :  */
313                 : static int find_group_dir(struct super_block *sb, struct inode *parent,
314                 :                           ext4_group_t *best_group)
315                 : {
316               0 :               ext4_group_t ngroups = ext4_get_groups_count(sb);
317                 :               unsigned int freei, avefreei;
318               0 :               struct ext4_group_desc *desc, *best_desc = NULL;
319                 :               ext4_group_t group;
320               0 :               int ret = -1;
321                 :
322               0 :               freei = percpu_counter_read_positive(&EXT4_SB(sb)->s_freeinodes_counter);
323               0 :               avefreei = freei / ngroups;
324                 :
325               0 :               for (group = 0; group < ngroups; group++) {
326               0 :                       desc = ext4_get_group_desc(sb, group, NULL);
327               0 :                       if (!desc || !ext4_free_inodes_count(sb, desc))
328                 :                               continue;
329               0 :                       if (ext4_free_inodes_count(sb, desc) < avefreei)
330                 :                               continue;
331               0 :                       if (!best_desc ||
332                 :                           (ext4_free_blks_count(sb, desc) >
333                 :                            ext4_free_blks_count(sb, best_desc))) {
334               0 :                               *best_group = group;
335               0 :                               best_desc = desc;
336               0 :                               ret = 0;
337                 :                       }
338                 :               }
339               0 :               return ret;
340                 : }
341                 :
342                 : #define free_block_ratio 10
```

```
343                 :
344                 : static int find_group_flex(struct super_block *sb, struct inode *parent,
345                 :                            ext4_group_t *best_group)
346               0 : {
347               0 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
348                 :         struct ext4_group_desc *desc;
349               0 :         struct flex_groups *flex_group = sbi->s_flex_groups;
350               0 :         ext4_group_t parent_group = EXT4_I(parent)->i_block_group;
351               0 :         ext4_group_t parent_fbg_group = ext4_flex_group(sbi, parent_group);
352               0 :         ext4_group_t ngroups = ext4_get_groups_count(sb);
353               0 :         int flex_size = ext4_flex_bg_size(sbi);
354               0 :         ext4_group_t best_flex = parent_fbg_group;
355               0 :         int blocks_per_flex = sbi->s_blocks_per_group * flex_size;
356                 :         int flexbg_free_blocks;
357                 :         int flex_freeb_ratio;
358                 :         ext4_group_t n_fbg_groups;
359                 :         ext4_group_t i;
360                 :
361               0 :         n_fbg_groups = (ngroups + flex_size - 1) >>
362                 :                 sbi->s_log_groups_per_flex;
363                 :
364               0 : find_close_to_parent:
365               0 :         flexbg_free_blocks = atomic_read(&flex_group[best_flex].free_blocks);
366               0 :         flex_freeb_ratio = flexbg_free_blocks * 100 / blocks_per_flex;
367               0 :         if (atomic_read(&flex_group[best_flex].free_inodes) &&
368                 :             flex_freeb_ratio > free_block_ratio)
369               0 :                 goto found_flexbg;
370                 :
371               0 :         if (best_flex && best_flex == parent_fbg_group) {
372               0 :                 best_flex--;
373               0 :                 goto find_close_to_parent;
374                 :         }
375                 :
376               0 :         for (i = 0; i < n_fbg_groups; i++) {
377               0 :                 if (i == parent_fbg_group || i == parent_fbg_group - 1)
378                 :                         continue;
379                 :
380               0 :                 flexbg_free_blocks = atomic_read(&flex_group[i].free_blocks);
381               0 :                 flex_freeb_ratio = flexbg_free_blocks * 100 / blocks_per_flex;
382                 :
383               0 :                 if (flex_freeb_ratio > free_block_ratio &&
384                 :                     (atomic_read(&flex_group[i].free_inodes))) {
385               0 :                         best_flex = i;
386               0 :                         goto found_flexbg;
387                 :                 }
388                 :
389               0 :                 if ((atomic_read(&flex_group[best_flex].free_inodes) == 0) ||
390                 :                     ((atomic_read(&flex_group[i].free_blocks) >
391                 :                       atomic_read(&flex_group[best_flex].free_blocks)) &&
392                 :                      atomic_read(&flex_group[i].free_inodes)))
393               0 :                         best_flex = i;
394                 :         }
395                 :
396               0 :         if (!atomic_read(&flex_group[best_flex].free_inodes) ||
397                 :             !atomic_read(&flex_group[best_flex].free_blocks))
398               0 :                 return -1;
399                 :
400               0 : found_flexbg:
401               0 :         for (i = best_flex * flex_size; i < ngroups &&
402               0 :                      i < (best_flex + 1) * flex_size; i++) {
403               0 :                 desc = ext4_get_group_desc(sb, i, NULL);
404               0 :                 if (ext4_free_inodes_count(sb, desc)) {
405               0 :                         *best_group = i;
406               0 :                         goto out;
407                 :                 }
408                 :         }
409                 :
410               0 :         return -1;
411               0 : out:
412               0 :         return 0;
413                 : }
```

```
    414                 :
    415                 : struct orlov_stats {
    416                 :         __u32 free_inodes;
    417                 :         __u32 free_blocks;
    418                 :         __u32 used_dirs;
    419                 : };
    420                 :
    421                 : /*
    422                 :  * Helper function for Orlov's allocator; returns critical information
    423                 :  * for a particular block group or flex_bg.  If flex_size is 1, then g
    424                 :  * is a block group number; otherwise it is flex_bg number.
    425                 :  */
    426                 : void get_orlov_stats(struct super_block *sb, ext4_group_t g,
    427                 :                      int flex_size, struct orlov_stats *stats)
    428      1392015087 : {
    429                 :         struct ext4_group_desc *desc;
    430      1392015087 :         struct flex_groups *flex_group = EXT4_SB(sb)->s_flex_groups;
    431                 :
    432      1392015087 :         if (flex_size > 1) {
    433     -1511285364 :                 stats->free_inodes = atomic_read(&flex_group[g].free_inodes);
    434     -1511285364 :                 stats->free_blocks = atomic_read(&flex_group[g].free_blocks);
    435     -1511285364 :                 stats->used_dirs = atomic_read(&flex_group[g].used_dirs);
    436      1391840966 :                 return;
    437                 :         }
    438                 :
    439          174121 :         desc = ext4_get_group_desc(sb, g, NULL);
    440          174121 :         if (desc) {
    441          174121 :                 stats->free_inodes = ext4_free_inodes_count(sb, desc);
    442          174121 :                 stats->free_blocks = ext4_free_blks_count(sb, desc);
    443          174121 :                 stats->used_dirs = ext4_used_dirs_count(sb, desc);
    444                 :         } else {
    445               0 :                 stats->free_inodes = 0;
    446               0 :                 stats->free_blocks = 0;
    447               0 :                 stats->used_dirs = 0;
    448                 :         }
    449                 : }
    450                 :
    451                 : /*
    452                 :  * Orlov's allocator for directories.
    453                 :  *
    454                 :  * We always try to spread first-level directories.
    455                 :  *
    456                 :  * If there are blockgroups with both free inodes and free blocks counts
    457                 :  * not worse than average we return one with smallest directory count.
    458                 :  * Otherwise we simply return a random group.
    459                 :  *
    460                 :  * For the rest rules look so:
    461                 :  *
    462                 :  * It's OK to put directory into a group unless
    463                 :  * it has too many directories already (max_dirs) or
    464                 :  * it has too few free inodes left (min_inodes) or
    465                 :  * it has too few free blocks left (min_blocks) or
    466                 :  * Parent's group is preferred, if it doesn't satisfy these
    467                 :  * conditions we search cyclically through the rest. If none
    468                 :  * of the groups look good we just look for a group with more
    469                 :  * free inodes than average (starting at parent's group).
    470                 :  */
    471                 :
    472                 : static int find_group_orlov(struct super_block *sb, struct inode *parent,
    473                 :                             ext4_group_t *group, int mode,
    474                 :                             const struct qstr *qstr)
    475          774694 : {
    476          774694 :         ext4_group_t parent_group = EXT4_I(parent)->i_block_group;
    477          774694 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
    478          774694 :         ext4_group_t real_ngroups = ext4_get_groups_count(sb);
    479          774694 :         int inodes_per_group = EXT4_INODES_PER_GROUP(sb);
    480                 :         unsigned int freei, avefreei;
    481                 :         ext4_fsblk_t freeb, avefreeb;
    482                 :         unsigned int ndirs;
    483                 :         int max_dirs, min_inodes;
    484                 :         ext4_grpblk_t min_blocks;
```

```
    485                 :          ext4_group_t i, grp, g, ngroups;
    486                 :          struct ext4_group_desc *desc;
    487                 :          struct orlov_stats stats;
    488          774694 :          int flex_size = ext4_flex_bg_size(sbi);
    489                 :          struct dx_hash_info hinfo;
    490                 :
    491          774694 :          ngroups = real_ngroups;
    492          774694 :          if (flex_size > 1) {
    493          774652 :                  ngroups = (real_ngroups + flex_size - 1) >>
    494                 :                          sbi->s_log_groups_per_flex;
    495          774652 :                  parent_group >>= sbi->s_log_groups_per_flex;
    496                 :          }
    497                 :
    498         1549388 :          freei = percpu_counter_read_positive(&sbi->s_freeinodes_counter);
    499          774694 :          avefreei = freei / ngroups;
    500         1549388 :          freeb = percpu_counter_read_positive(&sbi->s_freeblocks_counter);
    501          774694 :          avefreeb = freeb;
    502          774694 :          do_div(avefreeb, ngroups);
    503         1549388 :          ndirs = percpu_counter_read_positive(&sbi->s_dirs_counter);
    504                 :
    505         1124717 :          if (S_ISDIR(mode) &&
    506                 :              ((parent == sb->s_root->d_inode) ||
    507                 :               (EXT4_I(parent)->i_flags & EXT4_TOPDIR_FL))) {
    508          347643 :                  int best_ndir = inodes_per_group;
    509          347643 :                  int ret = -1;
    510                 :
    511          347643 :                  if (qstr) {
    512          347643 :                          hinfo.hash_version = DX_HASH_HALF_MD4;
    513          347643 :                          hinfo.seed = sbi->s_hash_seed;
    514          347643 :                          ext4fs_dirhash(qstr->name, qstr->len, &hinfo);
    515          347643 :                          grp = hinfo.hash;
    516                 :                  } else
    517               0 :                          get_random_bytes(&grp, sizeof(grp));
    518          347643 :                  parent_group = (unsigned)grp % ngroups;
    519       687787863 :                  for (i = 0; i < ngroups; i++) {
    520       687440220 :                          g = (parent_group + i) % ngroups;
    521       687440220 :                          get_orlov_stats(sb, g, flex_size, &stats);
    522       687440220 :                          if (!stats.free_inodes)
    523               0 :                                  continue;
    524       687440220 :                          if (stats.used_dirs >= best_ndir)
    525          177526 :                                  continue;
    526       687262694 :                          if (stats.free_inodes < avefreei)
    527       687262668 :                                  continue;
    528           26 :                          if (stats.free_blocks < avefreeb)
    529            0 :                                  continue;
    530           26 :                          grp = g;
    531           26 :                          ret = 0;
    532           26 :                          best_ndir = stats.used_dirs;
    533                 :                  }
    534          347643 :                  if (ret)
    535          347617 :                          goto fallback;
    536           62850 :          found_flex_bg:
    537           62850 :                  if (flex_size == 1) {
    538              42 :                          *group = grp;
    539              42 :                          return 0;
    540                 :                  }
    541                 :
    542                 :                  /*
    543                 :                   * We pack inodes at the beginning of the flexgroup's
    544                 :                   * inode tables.  Block allocation decisions will do
    545                 :                   * something similar, although regular files will
    546                 :                   * start at 2nd block group of the flexgroup.  See
    547                 :                   * ext4_ext_find_goal() and ext4_find_near().
    548                 :                   */
    549           62808 :                  grp *= flex_size;
    550         1067615 :                  for (i = 0; i < flex_size; i++) {
    551         1004816 :                          if (grp+i >= real_ngroups)
    552               0 :                                  break;
    553         1004816 :                          desc = ext4_get_group_desc(sb, grp+i, NULL);
    554         1004816 :                          if (desc && ext4_free_inodes_count(sb, desc)) {
    555               9 :                                  *group = grp+i;
```

```
556                9 :                                return 0;
557                  :                        }
558                  :                }
559                  :                goto fallback;
560                  :        }
561                  :
562           427051 :        max_dirs = ndirs / ngroups + inodes_per_group / 16;
563           427051 :        min_inodes = avefreei - inodes_per_group*flex_size / 4;
564           427051 :        if (min_inodes < 1)
565                0 :                min_inodes = 1;
566           854102 :        min_blocks = avefreeb - EXT4_BLOCKS_PER_GROUP(sb)*flex_size / 4;
567                  :
568                  :        /*
569                  :         * Start looking in the flex group where we last allocated an
570                  :         * inode for this parent directory
571                  :         */
572           427051 :        if (EXT4_I(parent)->i_last_alloc_group != ~0) {
573           426969 :                parent_group = EXT4_I(parent)->i_last_alloc_group;
574           426969 :                if (flex_size > 1)
575           426930 :                        parent_group >>= sbi->s_log_groups_per_flex;
576                  :        }
577                  :
578        704939094 :        for (i = 0; i < ngroups; i++) {
579        704574867 :                grp = (parent_group + i) % ngroups;
580        704574867 :                get_orlov_stats(sb, grp, flex_size, &stats);
581        704574867 :                if (stats.used_dirs >= max_dirs)
582                1 :                        continue;
583        704574866 :                if (stats.free_inodes < min_inodes)
584        704315061 :                        continue;
585           259805 :                if (stats.free_blocks < min_blocks)
586           196981 :                        continue;
587                  :                goto found_flex_bg;
588                  :        }
589                  :
590           774643 : fallback:
591           774643 :        ngroups = real_ngroups;
592           774643 :        avefreei = freei / ngroups;
593           774643 : fallback_retry:
594           774643 :        parent_group = EXT4_I(parent)->i_block_group;
595               -1 :        for (i = 0; i < ngroups; i++) {
596               -1 :                grp = (parent_group + i) % ngroups;
597               -1 :                desc = ext4_get_group_desc(sb, grp, NULL);
598               -1 :                if (desc && ext4_free_inodes_count(sb, desc) &&
599                  :                    ext4_free_inodes_count(sb, desc) >= avefreei) {
600           774643 :                        *group = grp;
601           774643 :                        return 0;
602                  :                }
603                  :        }
604                  :
605                0 :        if (avefreei) {
606                  :                /*
607                  :                 * The free-inodes counter is approximate, and for really small
608                  :                 * filesystems the above test can fail to find any blockgroups
609                  :                 */
610                0 :                avefreei = 0;
611                0 :                goto fallback_retry;
612                  :        }
613                  :
614                0 :        return -1;
615                  : }
616                  :
617                  : static int find_group_other(struct super_block *sb, struct inode *parent,
618                  :                             ext4_group_t *group, int mode)
619        8614906 : {
620        8614906 :        ext4_group_t parent_group = EXT4_I(parent)->i_block_group;
621        8614906 :        ext4_group_t i, last, ngroups = ext4_get_groups_count(sb);
622                  :        struct ext4_group_desc *desc;
623        8614906 :        int flex_size = ext4_flex_bg_size(EXT4_SB(sb));
624                  :
625                  :        /*
626                  :         * Try to place the inode is the same flex group as its
```

```
    627                 :          * parent.  If we can't find space, use the Orlov algorithm to
    628                 :          * find another flex group, and store that information in the
    629                 :          * parent directory's inode information so that use that flex
    630                 :          * group for future allocations.
    631                 :          */
    632         8614906 :         if (flex_size > 1) {
    633         8514898 :                 int retry = 0;
    634                 :
    635        10866607 :         try_again:
    636        10866607 :                 parent_group &= ~(flex_size-1);
    637        10866607 :                 last = parent_group + flex_size;
    638        10866607 :                 if (last > ngroups)
    639               0 :                         last = ngroups;
    640        87688056 :                 for  (i = parent_group; i < last; i++) {
    641        85259319 :                         desc = ext4_get_group_desc(sb, i, NULL);
    642        85259319 :                         if (desc && ext4_free_inodes_count(sb, desc)) {
    643         8437870 :                                 *group = i;
    644         8437870 :                                 return 0;
    645                 :                         }
    646                 :                 }
    647         4780451 :                 if (!retry && EXT4_I(parent)->i_last_alloc_group != ~0) {
    648         2351709 :                         retry = 1;
    649         2351709 :                         parent_group = EXT4_I(parent)->i_last_alloc_group;
    650         2351709 :                         goto try_again;
    651                 :                 }
    652                 :                 /*
    653                 :                  * If this didn't work, use the Orlov search algorithm
    654                 :                  * to find a new flex group; we pass in the mode to
    655                 :                  * avoid the topdir algorithms.
    656                 :                  */
    657           77028 :                 *group = parent_group + flex_size;
    658           77028 :                 if (*group > ngroups)
    659               0 :                         *group = 0;
    660           77028 :                 return find_group_orlov(sb, parent, group, mode, 0);
    661                 :         }
    662                 :
    663                 :         /*
    664                 :          * Try to place the inode in its parent directory
    665                 :          */
    666          100008 :         *group = parent_group;
    667          100008 :         desc = ext4_get_group_desc(sb, *group, NULL);
    668          100008 :         if (desc && ext4_free_inodes_count(sb, desc) &&
    669                 :                         ext4_free_blks_count(sb, desc))
    670             991 :                 return 0;
    671                 :
    672                 :         /*
    673                 :          * We're going to place this inode in a different blockgroup from its
    674                 :          * parent.  We want to cause files in a common directory to all land in
    675                 :          * the same blockgroup.  But we want files which are in a different
    676                 :          * directory which shares a blockgroup with our parent to land in a
    677                 :          * different blockgroup.
    678                 :          *
    679                 :          * So add our directory's i_ino into the starting point for the hash.
    680                 :          */
    681          99017 :         *group = (*group + parent->i_ino) % ngroups;
    682                 :
    683                 :         /*
    684                 :          * Use a quadratic hash to find a group with a free inode and some free
    685                 :          * blocks.
    686                 :          */
    687          509786 :         for (i = 1; i < ngroups; i <<= 1) {
    688          509786 :                 *group += i;
    689          509786 :                 if (*group >= ngroups)
    690               0 :                         *group -= ngroups;
    691          509786 :                 desc = ext4_get_group_desc(sb, *group, NULL);
    692          509786 :                 if (desc && ext4_free_inodes_count(sb, desc) &&
    693                 :                                 ext4_free_blks_count(sb, desc))
    694           99017 :                         return 0;
    695                 :         }
    696                 :
    697                 :         /*
```

```
   698                :          * That failed: try linear search for a free inode, even if that group
   699                :          * has no free blocks.
   700                :          */
   701              0 :         *group = parent_group;
   702              0 :         for (i = 0; i < ngroups; i++) {
   703              0 :                 if (++*group >= ngroups)
   704              0 :                         *group = 0;
   705              0 :                 desc = ext4_get_group_desc(sb, *group, NULL);
   706              0 :                 if (desc && ext4_free_inodes_count(sb, desc))
   707              0 :                         return 0;
   708                :         }
   709                :
   710              0 :         return -1;
   711                : }
   712                :
   713                : /*
   714                :  * claim the inode from the inode bitmap. If the group
   715                :  * is uninit we need to take the groups's ext4_group_lock
   716                :  * and clear the uninit flag. The inode bitmap update
   717                :  * and group desc uninit flag clear should be done
   718                :  * after holding ext4_group_lock so that ext4_read_inode_bitmap
   719                :  * doesn't race with the ext4_claim_inode
   720                :  */
   721                : static int ext4_claim_inode(struct super_block *sb,
   722                :                             struct buffer_head *inode_bitmap_bh,
   723                :                             unsigned long ino, ext4_group_t group, int mode)
   724        9312572 : {
   725        9312572 :         int free = 0, retval = 0, count;
   726        9312572 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
   727        9312572 :         struct ext4_group_desc *gdp = ext4_get_group_desc(sb, group, NULL);
   728                :
   729                :         ext4_lock_group(sb, group);
   730       18625144 :         if (ext4_set_bit(ino, inode_bitmap_bh->b_data)) {
   731                :                 /* not a free inode */
   732              0 :                 retval = 1;
   733              0 :                 goto err_ret;
   734                :         }
   735        9312572 :         ino++;
   736       19062305 :         if ((group == 0 && ino < EXT4_FIRST_INO(sb)) ||
   737                :                         ino > EXT4_INODES_PER_GROUP(sb)) {
   738                :                 ext4_unlock_group(sb, group);
   739              0 :                 ext4_error(sb, __func__,
   740                :                            "reserved inode or inode > inodes count - "
   741                :                            "block_group = %u, inode=%lu", group,
   742                :                            ino + group * EXT4_INODES_PER_GROUP(sb));
   743              0 :                 return 1;
   744                :         }
   745                :         /* If we didn't allocate from within the initialized part of the inode
   746                :          * table then we need to initialize up to this inode. */
   747        9312572 :         if (EXT4_HAS_RO_COMPAT_FEATURE(sb, EXT4_FEATURE_RO_COMPAT_GDT_CSUM)) {
   748                :
   749        9212522 :                 if (gdp->bg_flags & cpu_to_le16(EXT4_BG_INODE_UNINIT)) {
   750          90539 :                         gdp->bg_flags &= cpu_to_le16(~EXT4_BG_INODE_UNINIT);
   751                :                         /* When marking the block group with
   752                :                          * ~EXT4_BG_INODE_UNINIT we don't want to depend
   753                :                          * on the value of bg_itable_unused even though
   754                :                          * mke2fs could have initialized the same for us.
   755                :                          * Instead we calculated the value below
   756                :                          */
   757                :
   758          90539 :                         free = 0;
   759                :                 } else {
   760        9121983 :                         free = EXT4_INODES_PER_GROUP(sb) -
   761                :                                ext4_itable_unused_count(sb, gdp);
   762                :                 }
   763                :
   764                :                 /*
   765                :                  * Check the relative inode number against the last used
   766                :                  * relative inode number in this group. if it is greater
   767                :                  * we need to  update the bg_itable_unused count
   768                :                  *
```

```
769                :                  */
770          9212522 :                 if (ino > free)
771          8475822 :                         ext4_itable_unused_set(sb, gdp,
772                  :                                                (EXT4_INODES_PER_GROUP(sb) - ino));
773                  :         }
774          9312572 :         count = ext4_free_inodes_count(sb, gdp) - 1;
775          9312572 :         ext4_free_inodes_set(sb, gdp, count);
776          9312572 :         if (S_ISDIR(mode)) {
777           697666 :                 count = ext4_used_dirs_count(sb, gdp) + 1;
778           697666 :                 ext4_used_dirs_set(sb, gdp, count);
779           697666 :                 if (sbi->s_log_groups_per_flex) {
780           697624 :                         ext4_group_t f = ext4_flex_group(sbi, group);
781                  :
782           697624 :                         atomic_inc(&sbi->s_flex_groups[f].free_inodes);
783                  :                 }
784                  :         }
785          9312572 :         gdp->bg_checksum = ext4_group_desc_csum(sbi, group, gdp);
786          9312572 : err_ret:
787                  :         ext4_unlock_group(sb, group);
788          9312572 :         return retval;
789                  : }
790                  :
791                  : /*
792                  :  * There are two policies for allocating an inode.  If the new inode is
793                  :  * a directory, then a forward search is made for a block group with both
794                  :  * free space and a low directory-to-inode ratio; if that fails, then of
795                  :  * the groups with above-average free space, that group with the fewest
796                  :  * directories already is chosen.
797                  :  *
798                  :  * For other inodes, search forward from the parent directory's block
799                  :  * group to find a free inode.
800                  :  */
801                  : struct inode *ext4_new_inode(handle_t *handle, struct inode *dir, int mode,
802                  :                              const struct qstr *qstr, __u32 goal)
803          9312572 : {
804                  :         struct super_block *sb;
805          9312572 :         struct buffer_head *inode_bitmap_bh = NULL;
806                  :         struct buffer_head *group_desc_bh;
807          9312572 :         ext4_group_t ngroups, group = 0;
808          9312572 :         unsigned long ino = 0;
809                  :         struct inode *inode;
810          9312572 :         struct ext4_group_desc *gdp = NULL;
811                  :         struct ext4_inode_info *ei;
812                  :         struct ext4_sb_info *sbi;
813          9312572 :         int ret2, err = 0;
814                  :         struct inode *ret;
815                  :         ext4_group_t i;
816          9312572 :         int free = 0;
817                  :         static int once = 1;
818                  :         ext4_group_t flex_group;
819                  :
820                  :         /* Cannot create files in a deleted directory */
821          9312572 :         if (!dir || !dir->i_nlink)
822                0 :                 return ERR_PTR(-EPERM);
823                  :
824          9312572 :         sb = dir->i_sb;
825          9312572 :         ngroups = ext4_get_groups_count(sb);
826                  :         trace_ext4_request_inode(dir, mode);
827          9312572 :         inode = new_inode(sb);
828          9312572 :         if (!inode)
829                0 :                 return ERR_PTR(-ENOMEM);
830          9312572 :         ei = EXT4_I(inode);
831          9312572 :         sbi = EXT4_SB(sb);
832                  :
833          9312572 :         if (!goal)
834          9312572 :                 goal = sbi->s_inode_goal;
835                  :
836          9312572 :         if (goal && goal <= le32_to_cpu(sbi->s_es->s_inodes_count)) {
837                0 :                 group = (goal - 1) / EXT4_INODES_PER_GROUP(sb);
838                0 :                 ino = (goal - 1) % EXT4_INODES_PER_GROUP(sb);
839                0 :                 ret2 = 0;
```

```
       0 :                 goto got_group;
         :         }
         :
18525094 :         if (sbi->s_log_groups_per_flex && test_opt(sb, OLDALLOC)) {
       0 :                 ret2 = find_group_flex(sb, dir, &group);
       0 :                 if (ret2 == -1) {
       0 :                         ret2 = find_group_other(sb, dir, &group, mode);
       0 :                         if (ret2 == 0 && once) {
       0 :                                 once = 0;
       0 :                                 printk(KERN_NOTICE "ext4: find_group_flex "
         :                                        "failed, fallback succeeded dir %lu\n",
         :                                        dir->i_ino);
         :                         }
         :                 }
         :                 goto got_group;
         :         }
         :
 9312572 :         if (S_ISDIR(mode)) {
  697666 :                 if (test_opt(sb, OLDALLOC))
       0 :                         ret2 = find_group_dir(sb, dir, &group);
         :                 else
  697666 :                         ret2 = find_group_orlov(sb, dir, &group, mode, qstr);
         :         } else
 8614906 :                 ret2 = find_group_other(sb, dir, &group, mode);
         :
 9312572 : got_group:
 9312572 :         EXT4_I(dir)->i_last_alloc_group = group;
 9312572 :         err = -ENOSPC;
 9312572 :         if (ret2 == -1)
       0 :                 goto out;
         :
 9312572 :         for (i = 0; i < ngroups; i++, ino = 0) {
 9312572 :                 err = -EIO;
         :
 9312572 :                 gdp = ext4_get_group_desc(sb, group, &group_desc_bh);
 9312572 :                 if (!gdp)
       0 :                         goto fail;
         :
         :                 brelse(inode_bitmap_bh);
 9312572 :                 inode_bitmap_bh = ext4_read_inode_bitmap(sb, group);
 9312572 :                 if (!inode_bitmap_bh)
       0 :                         goto fail;
         :
 9312572 : repeat_in_this_group:
 9312572 :                 ino = ext4_find_next_zero_bit((unsigned long *)
         :                                               inode_bitmap_bh->b_data,
         :                                               EXT4_INODES_PER_GROUP(sb), ino);
         :
 9312572 :                 if (ino < EXT4_INODES_PER_GROUP(sb)) {
         :
         :                         BUFFER_TRACE(inode_bitmap_bh, "get_write_access");
 9312572 :                         err = ext4_journal_get_write_access(handle,
         :                                                             inode_bitmap_bh);
 9312572 :                         if (err)
       0 :                                 goto fail;
         :
         :                         BUFFER_TRACE(group_desc_bh, "get_write_access");
 9312572 :                         err = ext4_journal_get_write_access(handle,
         :                                                             group_desc_bh);
 9312572 :                         if (err)
       0 :                                 goto fail;
 9312572 :                         if (!ext4_claim_inode(sb, inode_bitmap_bh,
         :                                               ino, group, mode)) {
         :                                 /* we won it */
         :                                 BUFFER_TRACE(inode_bitmap_bh,
         :                                         "call ext4_handle_dirty_metadata");
 9312572 :                                 err = ext4_handle_dirty_metadata(handle,
         :                                                                  inode,
         :                                                                  inode_bitmap_bh);
 9312572 :                                 if (err)
       0 :                                         goto fail;
```

```
    911                :                                   /* zero bit is inode number 1*/
    912          9312572 :                                   ino++;
    913          9312572 :                                   goto got;
    914                :                           }
    915                :                           /* we lost it */
    916                :                           ext4_handle_release_buffer(handle, inode_bitmap_bh);
    917                0 :                           ext4_handle_release_buffer(handle, group_desc_bh);
    918                :
    919                0 :                           if (++ino < EXT4_INODES_PER_GROUP(sb))
    920                0 :                                   goto repeat_in_this_group;
    921                :                   }
    922                :
    923                :                   /*
    924                :                    * This case is possible in concurrent environment.  It is very
    925                :                    * rare.  We cannot repeat the find_group_xxx() call because
    926                :                    * that will simply return the same blockgroup, because the
    927                :                    * group descriptor metadata has not yet been updated.
    928                :                    * So we just go onto the next blockgroup.
    929                :                    */
    930                0 :                   if (++group == ngroups)
    931                0 :                           group = 0;
    932                :           }
    933                0 :           err = -ENOSPC;
    934                0 :           goto out;
    935                :
    936          9312572 : got:
    937                :           /* We may have to initialize the block bitmap if it isn't already */
    938          9312572 :           if (EXT4_HAS_RO_COMPAT_FEATURE(sb, EXT4_FEATURE_RO_COMPAT_GDT_CSUM) &&
    939                :               gdp->bg_flags & cpu_to_le16(EXT4_BG_BLOCK_UNINIT)) {
    940                :                   struct buffer_head *block_bitmap_bh;
    941                :
    942            82880 :                   block_bitmap_bh = ext4_read_block_bitmap(sb, group);
    943                :                   BUFFER_TRACE(block_bitmap_bh, "get block bitmap access");
    944            82880 :                   err = ext4_journal_get_write_access(handle, block_bitmap_bh);
    945            82880 :                   if (err) {
    946                :                           brelse(block_bitmap_bh);
    947                :                           goto fail;
    948                :                   }
    949                :
    950            82880 :                   free = 0;
    951            82880 :                   ext4_lock_group(sb, group);
    952                :                   /* recheck and clear flag under lock if we still need to */
    953            82880 :                   if (gdp->bg_flags & cpu_to_le16(EXT4_BG_BLOCK_UNINIT)) {
    954            82880 :                           free = ext4_free_blocks_after_init(sb, group, gdp);
    955            82880 :                           gdp->bg_flags &= cpu_to_le16(~EXT4_BG_BLOCK_UNINIT);
    956            82880 :                           ext4_free_blks_set(sb, gdp, free);
    957            82880 :                           gdp->bg_checksum = ext4_group_desc_csum(sbi, group,
    958                :                                                                   gdp);
    959                :                   }
    960            82880 :                   ext4_unlock_group(sb, group);
    961                :
    962                :                   /* Don't need to dirty bitmap block if we didn't change it */
    963            82880 :                   if (free) {
    964                :                           BUFFER_TRACE(block_bitmap_bh, "dirty block bitmap");
    965            82880 :                           err = ext4_handle_dirty_metadata(handle,
    966                :                                                            NULL, block_bitmap_bh);
    967                :                   }
    968                :
    969                :                   brelse(block_bitmap_bh);
    970            82880 :                   if (err)
    971                0 :                           goto fail;
    972                :           }
    973                :           BUFFER_TRACE(group_desc_bh, "call ext4_handle_dirty_metadata");
    974          9312572 :           err = ext4_handle_dirty_metadata(handle, NULL, group_desc_bh);
    975          9312572 :           if (err)
    976                0 :                   goto fail;
    977                :
    978          9312572 :           percpu_counter_dec(&sbi->s_freeinodes_counter);
    979          9312572 :           if (S_ISDIR(mode))
    980           697666 :                   percpu_counter_inc(&sbi->s_dirs_counter);
    981          9312572 :           sb->s_dirt = 1;
```

```
    982                 :
    983         9312572 :         if (sbi->s_log_groups_per_flex) {
    984        18425044 :                 flex_group = ext4_flex_group(sbi, group);
    985         9212522 :                 atomic_dec(&sbi->s_flex_groups[flex_group].free_inodes);
    986                 :         }
    987                 :
    988         9312572 :         inode->i_uid = current_fsuid();
    989         9312572 :         if (test_opt(sb, GRPID))
    990               0 :                 inode->i_gid = dir->i_gid;
    991         9312572 :         else if (dir->i_mode & S_ISGID) {
    992               0 :                 inode->i_gid = dir->i_gid;
    993               0 :                 if (S_ISDIR(mode))
    994               0 :                         mode |= S_ISGID;
    995                 :         } else
    996         9312572 :                 inode->i_gid = current_fsgid();
    997         9312572 :         inode->i_mode = mode;
    998                 :
    999        18625144 :         inode->i_ino = ino + group * EXT4_INODES_PER_GROUP(sb);
   1000                 :         /* This is the optimal IO size (for stat), not the fs block size */
   1001         9312572 :         inode->i_blocks = 0;
   1002         9312572 :         inode->i_mtime = inode->i_atime = inode->i_ctime = ei->i_crtime =
   1003                 :                                                        ext4_current_time(inode);
   1004                 :
   1005         9312572 :         memset(ei->i_data, 0, sizeof(ei->i_data));
   1006         9312572 :         ei->i_dir_start_lookup = 0;
   1007         9312572 :         ei->i_disksize = 0;
   1008                 :
   1009                 :         /*
   1010                 :          * Don't inherit extent flag from directory, amongst others. We set
   1011                 :          * extent flag on newly created directory and file only if -o extent
   1012                 :          * mount option is specified
   1013                 :          */
   1014        18625144 :         ei->i_flags =
   1015                 :                 ext4_mask_flags(mode, EXT4_I(dir)->i_flags & EXT4_FL_INHERITED);
   1016         9312572 :         ei->i_file_acl = 0;
   1017         9312572 :         ei->i_dtime = 0;
   1018         9312572 :         ei->i_block_group = group;
   1019         9312572 :         ei->i_last_alloc_group = ~0;
   1020                 :
   1021         9312572 :         ext4_set_inode_flags(inode);
   1022         9312572 :         if (IS_DIRSYNC(inode))
   1023                 :                 ext4_handle_sync(handle);
   1024         9312572 :         if (insert_inode_locked(inode) < 0) {
   1025               0 :                 err = -EINVAL;
   1026               0 :                 goto fail_drop;
   1027                 :         }
   1028         9312572 :         spin_lock(&sbi->s_next_gen_lock);
   1029         9312572 :         inode->i_generation = sbi->s_next_generation++;
   1030         9312572 :         spin_unlock(&sbi->s_next_gen_lock);
   1031                 :
   1032         9312572 :         ei->i_state = EXT4_STATE_NEW;
   1033                 :
   1034         9312572 :         ei->i_extra_isize = EXT4_SB(sb)->s_want_extra_isize;
   1035                 :
   1036         9312572 :         ret = inode;
   1037         9312572 :         if (vfs_dq_alloc_inode(inode)) {
   1038               0 :                 err = -EDQUOT;
   1039               0 :                 goto fail_drop;
   1040                 :         }
   1041                 :
   1042         9312572 :         err = ext4_init_acl(handle, inode, dir);
   1043         9312572 :         if (err)
   1044               0 :                 goto fail_free_drop;
   1045                 :
   1046         9312572 :         err = ext4_init_security(handle, inode, dir);
   1047         9312572 :         if (err)
   1048               0 :                 goto fail_free_drop;
   1049                 :
   1050         9312572 :         if (EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_EXTENTS)) {
   1051                 :                 /* set extent flag only for directory, file and normal symlink*/
   1052         9212522 :                 if (S_ISDIR(mode) || S_ISREG(mode) || S_ISLNK(mode)) {
```

```
1053  9212522 :                         EXT4_I(inode)->i_flags |= EXT4_EXTENTS_FL;
1054  9212522 :                         ext4_ext_tree_init(handle, inode);
1055          :                 }
1056          :         }
1057          :
1058  9312572 :         err = ext4_mark_inode_dirty(handle, inode);
1059  9312572 :         if (err) {
1060        0 :                 ext4_std_error(sb, err);
1061          :                 goto fail_free_drop;
1062          :         }
1063          :
1064          :         ext4_debug("allocating inode %lu\n", inode->i_ino);
1065          :         trace_ext4_allocate_inode(inode, dir, mode);
1066          :         goto really_out;
1067        0 : fail:
1068        0 :         ext4_std_error(sb, err);
1069        0 : out:
1070        0 :         iput(inode);
1071        0 :         ret = ERR_PTR(err);
1072  9312572 : really_out:
1073          :         brelse(inode_bitmap_bh);
1074  9312572 :         return ret;
1075          :
1076        0 : fail_free_drop:
1077        0 :         vfs_dq_free_inode(inode);
1078          :
1079        0 : fail_drop:
1080        0 :         vfs_dq_drop(inode);
1081        0 :         inode->i_flags |= S_NOQUOTA;
1082        0 :         inode->i_nlink = 0;
1083        0 :         unlock_new_inode(inode);
1084        0 :         iput(inode);
1085          :         brelse(inode_bitmap_bh);
1086        0 :         return ERR_PTR(err);
1087          : }
1088          :
1089          : /* Verify that we are loading a valid orphan from disk */
1090          : struct inode *ext4_orphan_get(struct super_block *sb, unsigned long ino)
1091        0 : {
1092        0 :         unsigned long max_ino = le32_to_cpu(EXT4_SB(sb)->s_es->s_inodes_count);
1093          :         ext4_group_t block_group;
1094          :         int bit;
1095          :         struct buffer_head *bitmap_bh;
1096        0 :         struct inode *inode = NULL;
1097        0 :         long err = -EIO;
1098          :
1099          :         /* Error cases - e2fsck has already cleaned up for us */
1100        0 :         if (ino > max_ino) {
1101        0 :                 ext4_warning(sb, __func__,
1102          :                              "bad orphan ino %lu!  e2fsck was run?", ino);
1103        0 :                 goto error;
1104          :         }
1105          :
1106        0 :         block_group = (ino - 1) / EXT4_INODES_PER_GROUP(sb);
1107        0 :         bit = (ino - 1) % EXT4_INODES_PER_GROUP(sb);
1108        0 :         bitmap_bh = ext4_read_inode_bitmap(sb, block_group);
1109        0 :         if (!bitmap_bh) {
1110        0 :                 ext4_warning(sb, __func__,
1111          :                              "inode bitmap error for orphan %lu", ino);
1112        0 :                 goto error;
1113          :         }
1114          :
1115          :         /* Having the inode bit set should be a 100% indicator that this
1116          :          * is a valid orphan (no e2fsck run on fs).  Orphans also include
1117          :          * inodes that were being truncated, so we can't check i_nlink==0.
1118          :          */
1119        0 :         if (!ext4_test_bit(bit, bitmap_bh->b_data))
1120        0 :                 goto bad_orphan;
1121          :
1122        0 :         inode = ext4_iget(sb, ino);
1123        0 :         if (IS_ERR(inode))
```

```
1124                0 :                 goto iget_failed;
1125                  :
1126                  :         /*
1127                  :          * If the orphans has i_nlinks > 0 then it should be able to be
1128                  :          * truncated, otherwise it won't be removed from the orphan list
1129                  :          * during processing and an infinite loop will result.
1130                  :          */
1131                0 :         if (inode->i_nlink && !ext4_can_truncate(inode))
1132                0 :                 goto bad_orphan;
1133                  :
1134                0 :         if (NEXT_ORPHAN(inode) > max_ino)
1135                0 :                 goto bad_orphan;
1136                  :         brelse(bitmap_bh);
1137                0 :         return inode;
1138                  :
1139                0 : iget_failed:
1140                0 :         err = PTR_ERR(inode);
1141                0 :         inode = NULL;
1142                0 : bad_orphan:
1143                0 :         ext4_warning(sb, __func__,
1144                  :                      "bad orphan inode %lu!  e2fsck was run?", ino);
1145                0 :         printk(KERN_NOTICE "ext4_test_bit(bit=%d, block=%llu) = %d\n",
1146                  :                bit, (unsigned long long)bitmap_bh->b_blocknr,
1147                  :                ext4_test_bit(bit, bitmap_bh->b_data));
1148                0 :         printk(KERN_NOTICE "inode=%p\n", inode);
1149                0 :         if (inode) {
1150                0 :                 printk(KERN_NOTICE "is_bad_inode(inode)=%d\n",
1151                  :                        is_bad_inode(inode));
1152                0 :                 printk(KERN_NOTICE "NEXT_ORPHAN(inode)=%u\n",
1153                  :                        NEXT_ORPHAN(inode));
1154                0 :                 printk(KERN_NOTICE "max_ino=%lu\n", max_ino);
1155                0 :                 printk(KERN_NOTICE "i_nlink=%u\n", inode->i_nlink);
1156                  :                 /* Avoid freeing blocks if we got a bad deleted inode */
1157                0 :                 if (inode->i_nlink == 0)
1158                0 :                         inode->i_blocks = 0;
1159                0 :                 iput(inode);
1160                  :         }
1161                  :         brelse(bitmap_bh);
1162                0 : error:
1163                0 :         return ERR_PTR(err);
1164                  : }
1165                  :
1166                  : unsigned long ext4_count_free_inodes(struct super_block *sb)
1167              188 : {
1168                  :         unsigned long desc_count;
1169                  :         struct ext4_group_desc *gdp;
1170              188 :         ext4_group_t i, ngroups = ext4_get_groups_count(sb);
1171                  : #ifdef EXT4FS_DEBUG
1172                  :         struct ext4_super_block *es;
1173                  :         unsigned long bitmap_count, x;
1174                  :         struct buffer_head *bitmap_bh = NULL;
1175                  :
1176                  :         es = EXT4_SB(sb)->s_es;
1177                  :         desc_count = 0;
1178                  :         bitmap_count = 0;
1179                  :         gdp = NULL;
1180                  :         for (i = 0; i < ngroups; i++) {
1181                  :                 gdp = ext4_get_group_desc(sb, i, NULL);
1182                  :                 if (!gdp)
1183                  :                         continue;
1184                  :                 desc_count += ext4_free_inodes_count(sb, gdp);
1185                  :                 brelse(bitmap_bh);
1186                  :                 bitmap_bh = ext4_read_inode_bitmap(sb, i);
1187                  :                 if (!bitmap_bh)
1188                  :                         continue;
1189                  :
1190                  :                 x = ext4_count_free(bitmap_bh, EXT4_INODES_PER_GROUP(sb) / 8);
1191                  :                 printk(KERN_DEBUG "group %lu: stored = %d, counted = %lu\n",
1192                  :                         i, ext4_free_inodes_count(sb, gdp), x);
1193                  :                 bitmap_count += x;
1194                  :         }
```

```
    1195                :         brelse(bitmap_bh);
    1196                :         printk(KERN_DEBUG "ext4_count_free_inodes: "
    1197                :                "stored = %u, computed = %lu, %lu\n",
    1198                :                le32_to_cpu(es->s_free_inodes_count), desc_count, bitmap_count);
    1199                :         return desc_count;
    1200                : #else
    1201             188 :         desc_count = 0;
    1202         3438268 :         for (i = 0; i < ngroups; i++) {
    1203         3438080 :                 gdp = ext4_get_group_desc(sb, i, NULL);
    1204         3438080 :                 if (!gdp)
    1205               0 :                         continue;
    1206         3438080 :                 desc_count += ext4_free_inodes_count(sb, gdp);
    1207                :                 cond_resched();
    1208                :         }
    1209             188 :         return desc_count;
    1210                : #endif
    1211                : }
    1212                :
    1213                : /* Called at mount-time, super-block is locked */
    1214                : unsigned long ext4_count_dirs(struct super_block * sb)
    1215              94 : {
    1216              94 :         unsigned long count = 0;
    1217              94 :         ext4_group_t i, ngroups = ext4_get_groups_count(sb);
    1218                :
    1219         1719134 :         for (i = 0; i < ngroups; i++) {
    1220         1719040 :                 struct ext4_group_desc *gdp = ext4_get_group_desc(sb, i, NULL);
    1221         1719040 :                 if (!gdp)
    1222               0 :                         continue;
    1223         1719040 :                 count += ext4_used_dirs_count(sb, gdp);
    1224                :         }
    1225              94 :         return count;
    1226                : }
```