LCOV - code coverage report

			Found	Hit	Coverage
Current view:	directory - fs/ext4 - xattr.c (source / functions)				
Test:	kernel_2_6_31_ext4_round_3.info	**Lines:**	739	9	1.2 %
Date:	2009-10-24	**Functions:**	24	2	8.3 %

```
                :/*
                : * linux/fs/ext4/xattr.c
                : *
                : * Copyright (C) 2001-2003 Andreas Gruenbacher, <agruen@suse.de>
                : *
                : * Fix by Harrison Xing <harrison@mountainviewdata.com>.
                : * Ext4 code with a lot of help from Eric Jarman <ejarman@acm.org>.
                : * Extended attributes for symlinks and special files added per
                : *  suggestion of Luka Renko <luka.renko@hermes.si>.
                : * xattr consolidation Copyright (c) 2004 James Morris <jmorris@redhat.com>,
                : *  Red Hat Inc.
                : * ea-in-inode support by Alex Tomas <alex@clusterfs.com> aka bzzz
                : *  and Andreas Gruenbacher <agruen@suse.de>.
                : */
                :
                :/*
                : * Extended attributes are stored directly in inodes (on file systems with
                : * inodes bigger than 128 bytes) and on additional disk blocks. The i_file_acl
                : * field contains the block number if an inode uses an additional block. All
                : * attributes must fit in the inode and one additional block. Blocks that
                : * contain the identical set of attributes may be shared among several inodes.
                : * Identical blocks are detected by keeping a cache of blocks that have
                : * recently been accessed.
                : *
                : * The attributes in inodes and on blocks have a different header; the entries
                : * are stored in the same format:
                : *
                : *   +------------------+
                : *   | header           |
                : *   | entry 1          | |
                : *   | entry 2          | | growing downwards
                : *   | entry 3          | v
                : *   | four null bytes  |
                : *   | . . .            |
                : *   | value 1          | ^
                : *   | value 3          | | growing upwards
                : *   | value 2          | |
                : *   +------------------+
                : *
                : * The header is followed by multiple entry descriptors. In disk blocks, the
                : * entry descriptors are kept sorted. In inodes, they are unsorted. The
                : * attribute values are aligned to the end of the block in no specific order.
                : *
                : * Locking strategy
                : * ----------------
                : * EXT4_I(inode)->i_file_acl is protected by EXT4_I(inode)->xattr_sem.
                : * EA blocks are only changed if they are exclusive to an inode, so
                : * holding xattr_sem also means that nothing but the EA block's reference
                : * count can change. Multiple writers to the same block are synchronized
                : * by the buffer lock.
                : */
                :
                :#include <linux/init.h>
                :#include <linux/fs.h>
                :#include <linux/slab.h>
                :#include <linux/mbcache.h>
                :#include <linux/quotaops.h>
                :#include <linux/rwsem.h>
```

```
     59                 : #include "ext4_jbd2.h"
     60                 : #include "ext4.h"
     61                 : #include "xattr.h"
     62                 : #include "acl.h"
     63                 :
     64                 : #define BHDR(bh) ((struct ext4_xattr_header *)((bh)->b_data))
     65                 : #define ENTRY(ptr) ((struct ext4_xattr_entry *)(ptr))
     66                 : #define BFIRST(bh) ENTRY(BHDR(bh)+1)
     67                 : #define IS_LAST_ENTRY(entry) (*(__u32 *)(entry) == 0)
     68                 :
     69                 : #ifdef EXT4_XATTR_DEBUG
     70                 : # define ea_idebug(inode, f...) do { \
     71                 :                 printk(KERN_DEBUG "inode %s:%lu: ", \
     72                 :                         inode->i_sb->s_id, inode->i_ino); \
     73                 :                 printk(f); \
     74                 :                 printk("\n"); \
     75                 :         } while (0)
     76                 : # define ea_bdebug(bh, f...) do { \
     77                 :                 char b[BDEVNAME_SIZE]; \
     78                 :                 printk(KERN_DEBUG "block %s:%lu: ", \
     79                 :                         bdevname(bh->b_bdev, b), \
     80                 :                         (unsigned long) bh->b_blocknr); \
     81                 :                 printk(f); \
     82                 :                 printk("\n"); \
     83                 :         } while (0)
     84                 : #else
     85                 : # define ea_idebug(f...)
     86                 : # define ea_bdebug(f...)
     87                 : #endif
     88                 :
     89                 : static void ext4_xattr_cache_insert(struct buffer_head *);
     90                 : static struct buffer_head *ext4_xattr_cache_find(struct inode *,
     91                 :                                                 struct ext4_xattr_header *,
     92                 :                                                 struct mb_cache_entry **);
     93                 : static void ext4_xattr_rehash(struct ext4_xattr_header *,
     94                 :                               struct ext4_xattr_entry *);
     95                 : static int ext4_xattr_list(struct inode *inode, char *buffer,
     96                 :                            size_t buffer_size);
     97                 :
     98                 : static struct mb_cache *ext4_xattr_cache;
     99                 :
    100                 : static struct xattr_handler *ext4_xattr_handler_map[] = {
    101                 :         [EXT4_XATTR_INDEX_USER]              = &ext4_xattr_user_handler,
    102                 : #ifdef CONFIG_EXT4_FS_POSIX_ACL
    103                 :         [EXT4_XATTR_INDEX_POSIX_ACL_ACCESS]  = &ext4_xattr_acl_access_handler,
    104                 :         [EXT4_XATTR_INDEX_POSIX_ACL_DEFAULT] = &ext4_xattr_acl_default_handler,
    105                 : #endif
    106                 :         [EXT4_XATTR_INDEX_TRUSTED]           = &ext4_xattr_trusted_handler,
    107                 : #ifdef CONFIG_EXT4_FS_SECURITY
    108                 :         [EXT4_XATTR_INDEX_SECURITY]          = &ext4_xattr_security_handler,
    109                 : #endif
    110                 : };
    111                 :
    112                 : struct xattr_handler *ext4_xattr_handlers[] = {
    113                 :         &ext4_xattr_user_handler,
    114                 :         &ext4_xattr_trusted_handler,
    115                 : #ifdef CONFIG_EXT4_FS_POSIX_ACL
    116                 :         &ext4_xattr_acl_access_handler,
    117                 :         &ext4_xattr_acl_default_handler,
    118                 : #endif
    119                 : #ifdef CONFIG_EXT4_FS_SECURITY
    120                 :         &ext4_xattr_security_handler,
    121                 : #endif
    122                 :         NULL
    123                 : };
    124                 :
    125                 : static inline struct xattr_handler *
    126                 : ext4_xattr_handler(int name_index)
    127                 : {
    128               0 :         struct xattr_handler *handler = NULL;
    129                 :
    130               0 :         if (name_index > 0 && name_index < ARRAY_SIZE(ext4_xattr_handler_map))
```

```
131              0 :                 handler = ext4_xattr_handler_map[name_index];
132              0 :         return handler;
133                : }
134                :
135                : /*
136                :  * Inode operation listxattr()
137                :  *
138                :  * dentry->d_inode->i_mutex: don't care
139                :  */
140                : ssize_t
141                : ext4_listxattr(struct dentry *dentry, char *buffer, size_t size)
142              0 : {
143              0 :         return ext4_xattr_list(dentry->d_inode, buffer, size);
144                : }
145                :
146                : static int
147                : ext4_xattr_check_names(struct ext4_xattr_entry *entry, void *end)
148                : {
149              0 :         while (!IS_LAST_ENTRY(entry)) {
150              0 :                 struct ext4_xattr_entry *next = EXT4_XATTR_NEXT(entry);
151              0 :                 if ((void *)next >= end)
152              0 :                         return -EIO;
153              0 :                 entry = next;
154                :         }
155              0 :         return 0;
156                : }
157                :
158                : static inline int
159                : ext4_xattr_check_block(struct buffer_head *bh)
160                : {
161                :         int error;
162                :
163              0 :         if (BHDR(bh)->h_magic != cpu_to_le32(EXT4_XATTR_MAGIC) ||
164                :             BHDR(bh)->h_blocks != cpu_to_le32(1))
165              0 :                 return -EIO;
166              0 :         error = ext4_xattr_check_names(BFIRST(bh), bh->b_data + bh->b_size);
167              0 :         return error;
168                : }
169                :
170                : static inline int
171                : ext4_xattr_check_entry(struct ext4_xattr_entry *entry, size_t size)
172                : {
173              0 :         size_t value_size = le32_to_cpu(entry->e_value_size);
174                :
175              0 :         if (entry->e_value_block != 0 || value_size > size ||
176                :             le16_to_cpu(entry->e_value_offs) + value_size > size)
177              0 :                 return -EIO;
178              0 :         return 0;
179                : }
180                :
181                : static int
182                : ext4_xattr_find_entry(struct ext4_xattr_entry **pentry, int name_index,
183                :                       const char *name, size_t size, int sorted)
184              0 : {
185                :         struct ext4_xattr_entry *entry;
186                :         size_t name_len;
187              0 :         int cmp = 1;
188                :
189              0 :         if (name == NULL)
190              0 :                 return -EINVAL;
191              0 :         name_len = strlen(name);
192              0 :         entry = *pentry;
193              0 :         for (; !IS_LAST_ENTRY(entry); entry = EXT4_XATTR_NEXT(entry)) {
194              0 :                 cmp = name_index - entry->e_name_index;
195              0 :                 if (!cmp)
196              0 :                         cmp = name_len - entry->e_name_len;
197              0 :                 if (!cmp)
198              0 :                         cmp = memcmp(name, entry->e_name, name_len);
199              0 :                 if (cmp <= 0 && (sorted || cmp == 0))
200              0 :                         break;
201                :         }
202              0 :         *pentry = entry;
```

```
    203                0 :         if (!cmp && ext4_xattr_check_entry(entry, size))
    204                0 :                         return -EIO;
    205                0 :         return cmp ? -ENODATA : 0;
    206                  : }
    207                  :
    208                  : static int
    209                  : ext4_xattr_block_get(struct inode *inode, int name_index, const char *name,
    210                  :                      void *buffer, size_t buffer_size)
    211                0 : {
    212                0 :         struct buffer_head *bh = NULL;
    213                  :         struct ext4_xattr_entry *entry;
    214                  :         size_t size;
    215                  :         int error;
    216                  :
    217                  :         ea_idebug(inode, "name=%d.%s, buffer=%p, buffer_size=%ld",
    218                  :                   name_index, name, buffer, (long)buffer_size);
    219                  :
    220                0 :         error = -ENODATA;
    221                0 :         if (!EXT4_I(inode)->i_file_acl)
    222                0 :                 goto cleanup;
    223                  :         ea_idebug(inode, "reading block %u", EXT4_I(inode)->i_file_acl);
    224                0 :         bh = sb_bread(inode->i_sb, EXT4_I(inode)->i_file_acl);
    225                0 :         if (!bh)
    226                0 :                 goto cleanup;
    227                  :         ea_bdebug(bh, "b_count=%d, refcount=%d",
    228                  :                 atomic_read(&(bh->b_count)), le32_to_cpu(BHDR(bh)->h_refcount));
    229                0 :         if (ext4_xattr_check_block(bh)) {
    230                0 : bad_block:      ext4_error(inode->i_sb, __func__,
    231                  :                            "inode %lu: bad block %llu", inode->i_ino,
    232                  :                            EXT4_I(inode)->i_file_acl);
    233                0 :                 error = -EIO;
    234                0 :                 goto cleanup;
    235                  :         }
    236                0 :         ext4_xattr_cache_insert(bh);
    237                0 :         entry = BFIRST(bh);
    238                0 :         error = ext4_xattr_find_entry(&entry, name_index, name, bh->b_size, 1);
    239                0 :         if (error == -EIO)
    240                0 :                 goto bad_block;
    241                0 :         if (error)
    242                0 :                 goto cleanup;
    243                0 :         size = le32_to_cpu(entry->e_value_size);
    244                0 :         if (buffer) {
    245                0 :                 error = -ERANGE;
    246                0 :                 if (size > buffer_size)
    247                0 :                         goto cleanup;
    248                0 :                 memcpy(buffer, bh->b_data + le16_to_cpu(entry->e_value_offs),
    249                  :                        size);
    250                  :         }
    251                0 :         error = size;
    252                  :
    253                0 : cleanup:
    254                  :         brelse(bh);
    255                0 :         return error;
    256                  : }
    257                  :
    258                  : static int
    259                  : ext4_xattr_ibody_get(struct inode *inode, int name_index, const char *name,
    260                  :                      void *buffer, size_t buffer_size)
    261                0 : {
    262                  :         struct ext4_xattr_ibody_header *header;
    263                  :         struct ext4_xattr_entry *entry;
    264                  :         struct ext4_inode *raw_inode;
    265                  :         struct ext4_iloc iloc;
    266                  :         size_t size;
    267                  :         void *end;
    268                  :         int error;
    269                  :
    270                0 :         if (!(EXT4_I(inode)->i_state & EXT4_STATE_XATTR))
    271                0 :                 return -ENODATA;
    272                0 :         error = ext4_get_inode_loc(inode, &iloc);
    273                0 :         if (error)
    274                0 :                 return error;
```

```
275                0 :         raw_inode = ext4_raw_inode(&iloc);
276                0 :         header = IHDR(inode, raw_inode);
277                0 :         entry = IFIRST(header);
278                0 :         end = (void *)raw_inode + EXT4_SB(inode->i_sb)->s_inode_size;
279                0 :         error = ext4_xattr_check_names(entry, end);
280                0 :         if (error)
281                0 :                 goto cleanup;
282                0 :         error = ext4_xattr_find_entry(&entry, name_index, name,
283                  :                                       end - (void *)entry, 0);
284                0 :         if (error)
285                0 :                 goto cleanup;
286                0 :         size = le32_to_cpu(entry->e_value_size);
287                0 :         if (buffer) {
288                0 :                 error = -ERANGE;
289                0 :                 if (size > buffer_size)
290                0 :                         goto cleanup;
291                0 :                 memcpy(buffer, (void *)IFIRST(header) +
292                  :                        le16_to_cpu(entry->e_value_offs), size);
293                  :         }
294                0 :         error = size;
295                  :
296                0 : cleanup:
297                0 :         brelse(iloc.bh);
298                0 :         return error;
299                  : }
300                  :
301                  : /*
302                  :  * ext4_xattr_get()
303                  :  *
304                  :  * Copy an extended attribute into the buffer
305                  :  * provided, or compute the buffer size required.
306                  :  * Buffer is NULL to compute the size of the buffer required.
307                  :  *
308                  :  * Returns a negative error number on failure, or the number of bytes
309                  :  * used / required on success.
310                  :  */
311                  : int
312                  : ext4_xattr_get(struct inode *inode, int name_index, const char *name,
313                  :                void *buffer, size_t buffer_size)
314                0 : {
315                  :         int error;
316                  :
317                0 :         down_read(&EXT4_I(inode)->xattr_sem);
318                0 :         error = ext4_xattr_ibody_get(inode, name_index, name, buffer,
319                  :                                      buffer_size);
320                0 :         if (error == -ENODATA)
321                0 :                 error = ext4_xattr_block_get(inode, name_index, name, buffer,
322                  :                                              buffer_size);
323                0 :         up_read(&EXT4_I(inode)->xattr_sem);
324                0 :         return error;
325                  : }
326                  :
327                  : static int
328                  : ext4_xattr_list_entries(struct inode *inode, struct ext4_xattr_entry *entry,
329                  :                         char *buffer, size_t buffer_size)
330                0 : {
331                0 :         size_t rest = buffer_size;
332                  :
333                0 :         for (; !IS_LAST_ENTRY(entry); entry = EXT4_XATTR_NEXT(entry)) {
334                  :                 struct xattr_handler *handler =
335                0 :                         ext4_xattr_handler(entry->e_name_index);
336                  :
337                0 :                 if (handler) {
338                  :                         size_t size = handler->list(inode, buffer, rest,
339                  :                                                     entry->e_name,
340                0 :                                                     entry->e_name_len);
341                0 :                         if (buffer) {
342                0 :                                 if (size > rest)
343                0 :                                         return -ERANGE;
344                0 :                                 buffer += size;
345                  :                         }
346                0 :                         rest -= size;
```

```
347                :                 }
348                :         }
349              0 :         return buffer_size - rest;
350                : }
351                :
352                : static int
353                : ext4_xattr_block_list(struct inode *inode, char *buffer, size_t buffer_size)
354              0 : {
355              0 :         struct buffer_head *bh = NULL;
356                :         int error;
357                :
358                :         ea_idebug(inode, "buffer=%p, buffer_size=%ld",
359                :                   buffer, (long)buffer_size);
360                :
361              0 :         error = 0;
362              0 :         if (!EXT4_I(inode)->i_file_acl)
363              0 :                 goto cleanup;
364                :         ea_idebug(inode, "reading block %u", EXT4_I(inode)->i_file_acl);
365              0 :         bh = sb_bread(inode->i_sb, EXT4_I(inode)->i_file_acl);
366              0 :         error = -EIO;
367              0 :         if (!bh)
368              0 :                 goto cleanup;
369                :         ea_bdebug(bh, "b_count=%d, refcount=%d",
370                :                 atomic_read(&(bh->b_count)), le32_to_cpu(BHDR(bh)->h_refcount));
371              0 :         if (ext4_xattr_check_block(bh)) {
372              0 :                 ext4_error(inode->i_sb, __func__,
373                :                            "inode %lu: bad block %llu", inode->i_ino,
374                :                            EXT4_I(inode)->i_file_acl);
375              0 :                 error = -EIO;
376              0 :                 goto cleanup;
377                :         }
378              0 :         ext4_xattr_cache_insert(bh);
379              0 :         error = ext4_xattr_list_entries(inode, BFIRST(bh), buffer, buffer_size);
380                :
381              0 : cleanup:
382                :         brelse(bh);
383                :
384              0 :         return error;
385                : }
386                :
387                : static int
388                : ext4_xattr_ibody_list(struct inode *inode, char *buffer, size_t buffer_size)
389                : {
390                :         struct ext4_xattr_ibody_header *header;
391                :         struct ext4_inode *raw_inode;
392                :         struct ext4_iloc iloc;
393                :         void *end;
394                :         int error;
395                :
396              0 :         if (!(EXT4_I(inode)->i_state & EXT4_STATE_XATTR))
397              0 :                 return 0;
398              0 :         error = ext4_get_inode_loc(inode, &iloc);
399              0 :         if (error)
400              0 :                 return error;
401              0 :         raw_inode = ext4_raw_inode(&iloc);
402              0 :         header = IHDR(inode, raw_inode);
403              0 :         end = (void *)raw_inode + EXT4_SB(inode->i_sb)->s_inode_size;
404              0 :         error = ext4_xattr_check_names(IFIRST(header), end);
405              0 :         if (error)
406                :                 goto cleanup;
407              0 :         error = ext4_xattr_list_entries(inode, IFIRST(header),
408                :                                         buffer, buffer_size);
409                :
410              0 : cleanup:
411              0 :         brelse(iloc.bh);
412              0 :         return error;
413                : }
414                :
415                : /*
416                :  * ext4_xattr_list()
417                :  *
418                :  * Copy a list of attribute names into the buffer
```

```
419                :  * provided, or compute the buffer size required.
420                :  * Buffer is NULL to compute the size of the buffer required.
421                :  *
422                :  * Returns a negative error number on failure, or the number of bytes
423                :  * used / required on success.
424                :  */
425                : static int
426                : ext4_xattr_list(struct inode *inode, char *buffer, size_t buffer_size)
427              0 : {
428                :         int i_error, b_error;
429                :
430              0 :         down_read(&EXT4_I(inode)->xattr_sem);
431              0 :         i_error = ext4_xattr_ibody_list(inode, buffer, buffer_size);
432              0 :         if (i_error < 0) {
433              0 :                 b_error = 0;
434                :         } else {
435              0 :                 if (buffer) {
436              0 :                         buffer += i_error;
437              0 :                         buffer_size -= i_error;
438                :                 }
439              0 :                 b_error = ext4_xattr_block_list(inode, buffer, buffer_size);
440              0 :                 if (b_error < 0)
441              0 :                         i_error = 0;
442                :         }
443              0 :         up_read(&EXT4_I(inode)->xattr_sem);
444              0 :         return i_error + b_error;
445                : }
446                :
447                : /*
448                :  * If the EXT4_FEATURE_COMPAT_EXT_ATTR feature of this file system is
449                :  * not set, set it.
450                :  */
451                : static void ext4_xattr_update_super_block(handle_t *handle,
452                :                                           struct super_block *sb)
453                : {
454              0 :         if (EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_EXT_ATTR))
455                :                 return;
456                :
457              0 :         if (ext4_journal_get_write_access(handle, EXT4_SB(sb)->s_sbh) == 0) {
458              0 :                 EXT4_SET_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_EXT_ATTR);
459              0 :                 sb->s_dirt = 1;
460              0 :                 ext4_handle_dirty_metadata(handle, NULL, EXT4_SB(sb)->s_sbh);
461                :         }
462                : }
463                :
464                : /*
465                :  * Release the xattr block BH: If the reference count is > 1, decrement
466                :  * it; otherwise free the block.
467                :  */
468                : static void
469                : ext4_xattr_release_block(handle_t *handle, struct inode *inode,
470                :                          struct buffer_head *bh)
471              0 : {
472              0 :         struct mb_cache_entry *ce = NULL;
473              0 :         int error = 0;
474                :
475              0 :         ce = mb_cache_entry_get(ext4_xattr_cache, bh->b_bdev, bh->b_blocknr);
476              0 :         error = ext4_journal_get_write_access(handle, bh);
477              0 :         if (error)
478              0 :                 goto out;
479                :
480              0 :         lock_buffer(bh);
481              0 :         if (BHDR(bh)->h_refcount == cpu_to_le32(1)) {
482                :                 ea_bdebug(bh, "refcount now=0; freeing");
483              0 :                 if (ce)
484              0 :                         mb_cache_entry_free(ce);
485              0 :                 ext4_free_blocks(handle, inode, bh->b_blocknr, 1, 1);
486                :                 get_bh(bh);
487              0 :                 ext4_forget(handle, 1, inode, bh, bh->b_blocknr);
488                :         } else {
489              0 :                 le32_add_cpu(&BHDR(bh)->h_refcount, -1);
490              0 :                 error = ext4_handle_dirty_metadata(handle, inode, bh);
```

```
491          0 :                 if (IS_SYNC(inode))
492            :                         ext4_handle_sync(handle);
493          0 :                 vfs_dq_free_block(inode, 1);
494            :                 ea_bdebug(bh, "refcount now=%d; releasing",
495            :                           le32_to_cpu(BHDR(bh)->h_refcount));
496          0 :                 if (ce)
497          0 :                         mb_cache_entry_release(ce);
498            :         }
499          0 :         unlock_buffer(bh);
500          0 : out:
501          0 :         ext4_std_error(inode->i_sb, error);
502            :         return;
503            : }
504            :
505            : /*
506            :  * Find the available free space for EAs. This also returns the total number of
507            :  * bytes used by EA entries.
508            :  */
509            : static size_t ext4_xattr_free_space(struct ext4_xattr_entry *last,
510            :                                     size_t *min_offs, void *base, int *total)
511            : {
512          0 :         for (; !IS_LAST_ENTRY(last); last = EXT4_XATTR_NEXT(last)) {
513          0 :                 *total += EXT4_XATTR_LEN(last->e_name_len);
514          0 :                 if (!last->e_value_block && last->e_value_size) {
515          0 :                         size_t offs = le16_to_cpu(last->e_value_offs);
516          0 :                         if (offs < *min_offs)
517          0 :                                 *min_offs = offs;
518            :                 }
519            :         }
520          0 :         return (*min_offs - ((void *)last - base) - sizeof(__u32));
521            : }
522            :
523            : struct ext4_xattr_info {
524            :         int name_index;
525            :         const char *name;
526            :         const void *value;
527            :         size_t value_len;
528            : };
529            :
530            : struct ext4_xattr_search {
531            :         struct ext4_xattr_entry *first;
532            :         void *base;
533            :         void *end;
534            :         struct ext4_xattr_entry *here;
535            :         int not_found;
536            : };
537            :
538            : static int
539            : ext4_xattr_set_entry(struct ext4_xattr_info *i, struct ext4_xattr_search *s)
540          0 : {
541            :         struct ext4_xattr_entry *last;
542          0 :         size_t free, min_offs = s->end - s->base, name_len = strlen(i->name);
543            :
544            :         /* Compute min_offs and last. */
545          0 :         last = s->first;
546          0 :         for (; !IS_LAST_ENTRY(last); last = EXT4_XATTR_NEXT(last)) {
547          0 :                 if (!last->e_value_block && last->e_value_size) {
548          0 :                         size_t offs = le16_to_cpu(last->e_value_offs);
549          0 :                         if (offs < min_offs)
550          0 :                                 min_offs = offs;
551            :                 }
552            :         }
553          0 :         free = min_offs - ((void *)last - s->base) - sizeof(__u32);
554          0 :         if (!s->not_found) {
555          0 :                 if (!s->here->e_value_block && s->here->e_value_size) {
556          0 :                         size_t size = le32_to_cpu(s->here->e_value_size);
557          0 :                         free += EXT4_XATTR_SIZE(size);
558            :                 }
559          0 :                 free += EXT4_XATTR_LEN(name_len);
560            :         }
561          0 :         if (i->value) {
562          0 :                 if (free < EXT4_XATTR_SIZE(i->value_len) ||
```

```
563                :                       free < EXT4_XATTR_LEN(name_len) +
564                :                              EXT4_XATTR_SIZE(i->value_len))
565              0 :                               return -ENOSPC;
566                :               }
567                :
568              0 :               if (i->value && s->not_found) {
569                :                       /* Insert the new name. */
570              0 :                       size_t size = EXT4_XATTR_LEN(name_len);
571              0 :                       size_t rest = (void *)last - (void *)s->here + sizeof(__u32);
572              0 :                       memmove((void *)s->here + size, s->here, rest);
573              0 :                       memset(s->here, 0, size);
574              0 :                       s->here->e_name_index = i->name_index;
575              0 :                       s->here->e_name_len = name_len;
576              0 :                       memcpy(s->here->e_name, i->name, name_len);
577                :               } else {
578              0 :                       if (!s->here->e_value_block && s->here->e_value_size) {
579              0 :                               void *first_val = s->base + min_offs;
580              0 :                               size_t offs = le16_to_cpu(s->here->e_value_offs);
581              0 :                               void *val = s->base + offs;
582              0 :                               size_t size = EXT4_XATTR_SIZE(
583                :                                       le32_to_cpu(s->here->e_value_size));
584                :
585              0 :                               if (i->value && size == EXT4_XATTR_SIZE(i->value_len)) {
586                :                                       /* The old and the new value have the same
587                :                                          size. Just replace. */
588              0 :                                       s->here->e_value_size =
589                :                                               cpu_to_le32(i->value_len);
590              0 :                                       memset(val + size - EXT4_XATTR_PAD, 0,
591                :                                              EXT4_XATTR_PAD); /* Clear pad bytes. */
592              0 :                                       memcpy(val, i->value, i->value_len);
593              0 :                                       return 0;
594                :                               }
595                :
596                :                               /* Remove the old value. */
597              0 :                               memmove(first_val + size, first_val, val - first_val);
598              0 :                               memset(first_val, 0, size);
599              0 :                               s->here->e_value_size = 0;
600              0 :                               s->here->e_value_offs = 0;
601              0 :                               min_offs += size;
602                :
603                :                               /* Adjust all value offsets. */
604              0 :                               last = s->first;
605              0 :                               while (!IS_LAST_ENTRY(last)) {
606              0 :                                       size_t o = le16_to_cpu(last->e_value_offs);
607              0 :                                       if (!last->e_value_block &&
608                :                                           last->e_value_size && o < offs)
609              0 :                                               last->e_value_offs =
610                :                                                       cpu_to_le16(o + size);
611              0 :                                       last = EXT4_XATTR_NEXT(last);
612                :                               }
613                :                       }
614              0 :                       if (!i->value) {
615                :                               /* Remove the old name. */
616              0 :                               size_t size = EXT4_XATTR_LEN(name_len);
617              0 :                               last = ENTRY((void *)last - size);
618              0 :                               memmove(s->here, (void *)s->here + size,
619                :                                       (void *)last - (void *)s->here + sizeof(__u32));
620              0 :                               memset(last, 0, size);
621                :                       }
622                :               }
623                :
624              0 :               if (i->value) {
625                :                       /* Insert the new value. */
626              0 :                       s->here->e_value_size = cpu_to_le32(i->value_len);
627              0 :                       if (i->value_len) {
628              0 :                               size_t size = EXT4_XATTR_SIZE(i->value_len);
629              0 :                               void *val = s->base + min_offs - size;
630              0 :                               s->here->e_value_offs = cpu_to_le16(min_offs - size);
631              0 :                               memset(val + size - EXT4_XATTR_PAD, 0,
632                :                                      EXT4_XATTR_PAD); /* Clear the pad bytes. */
633              0 :                               memcpy(val, i->value, i->value_len);
634                :                       }
```

```
635                :         }
636              0 :         return 0;
637                : }
638                :
639                : struct ext4_xattr_block_find {
640                :         struct ext4_xattr_search s;
641                :         struct buffer_head *bh;
642                : };
643                :
644                : static int
645                : ext4_xattr_block_find(struct inode *inode, struct ext4_xattr_info *i,
646                :                       struct ext4_xattr_block_find *bs)
647              0 : {
648              0 :         struct super_block *sb = inode->i_sb;
649                :         int error;
650                :
651                :         ea_idebug(inode, "name=%d.%s, value=%p, value_len=%ld",
652                :                   i->name_index, i->name, i->value, (long)i->value_len);
653                :
654              0 :         if (EXT4_I(inode)->i_file_acl) {
655                :                 /* The inode already has an extended attribute block. */
656              0 :                 bs->bh = sb_bread(sb, EXT4_I(inode)->i_file_acl);
657              0 :                 error = -EIO;
658              0 :                 if (!bs->bh)
659              0 :                         goto cleanup;
660                :                 ea_bdebug(bs->bh, "b_count=%d, refcount=%d",
661                :                         atomic_read(&(bs->bh->b_count)),
662                :                         le32_to_cpu(BHDR(bs->bh)->h_refcount));
663              0 :                 if (ext4_xattr_check_block(bs->bh)) {
664              0 :                         ext4_error(sb, __func__,
665                :                                 "inode %lu: bad block %llu", inode->i_ino,
666                :                                 EXT4_I(inode)->i_file_acl);
667              0 :                         error = -EIO;
668              0 :                         goto cleanup;
669                :                 }
670                :                 /* Find the named attribute. */
671              0 :                 bs->s.base = BHDR(bs->bh);
672              0 :                 bs->s.first = BFIRST(bs->bh);
673              0 :                 bs->s.end = bs->bh->b_data + bs->bh->b_size;
674              0 :                 bs->s.here = bs->s.first;
675              0 :                 error = ext4_xattr_find_entry(&bs->s.here, i->name_index,
676                :                                               i->name, bs->bh->b_size, 1);
677              0 :                 if (error && error != -ENODATA)
678              0 :                         goto cleanup;
679              0 :                 bs->s.not_found = error;
680                :         }
681              0 :         error = 0;
682                :
683              0 : cleanup:
684              0 :         return error;
685                : }
686                :
687                : static int
688                : ext4_xattr_block_set(handle_t *handle, struct inode *inode,
689                :                      struct ext4_xattr_info *i,
690                :                      struct ext4_xattr_block_find *bs)
691              0 : {
692              0 :         struct super_block *sb = inode->i_sb;
693              0 :         struct buffer_head *new_bh = NULL;
694              0 :         struct ext4_xattr_search *s = &bs->s;
695              0 :         struct mb_cache_entry *ce = NULL;
696              0 :         int error = 0;
697                :
698                : #define header(x) ((struct ext4_xattr_header *)(x))
699                :
700              0 :         if (i->value && i->value_len > sb->s_blocksize)
701              0 :                 return -ENOSPC;
702              0 :         if (s->base) {
703              0 :                 ce = mb_cache_entry_get(ext4_xattr_cache, bs->bh->b_bdev,
704                :                                         bs->bh->b_blocknr);
705              0 :                 error = ext4_journal_get_write_access(handle, bs->bh);
706              0 :                 if (error)
```

```
   707          0 :                         goto cleanup;
   708          0 :                 lock_buffer(bs->bh);
   709            :
   710          0 :                 if (header(s->base)->h_refcount == cpu_to_le32(1)) {
   711          0 :                         if (ce) {
   712          0 :                                 mb_cache_entry_free(ce);
   713          0 :                                 ce = NULL;
   714            :                         }
   715            :                         ea_bdebug(bs->bh, "modifying in-place");
   716          0 :                         error = ext4_xattr_set_entry(i, s);
   717          0 :                         if (!error) {
   718          0 :                                 if (!IS_LAST_ENTRY(s->first))
   719          0 :                                         ext4_xattr_rehash(header(s->base),
   720            :                                                           s->here);
   721          0 :                                 ext4_xattr_cache_insert(bs->bh);
   722            :                         }
   723          0 :                         unlock_buffer(bs->bh);
   724          0 :                         if (error == -EIO)
   725          0 :                                 goto bad_block;
   726          0 :                         if (!error)
   727          0 :                                 error = ext4_handle_dirty_metadata(handle,
   728            :                                                                    inode,
   729            :                                                                    bs->bh);
   730          0 :                         if (error)
   731          0 :                                 goto cleanup;
   732            :                         goto inserted;
   733            :                 } else {
   734          0 :                         int offset = (char *)s->here - bs->bh->b_data;
   735            :
   736          0 :                         unlock_buffer(bs->bh);
   737          0 :                         jbd2_journal_release_buffer(handle, bs->bh);
   738          0 :                         if (ce) {
   739          0 :                                 mb_cache_entry_release(ce);
   740          0 :                                 ce = NULL;
   741            :                         }
   742            :                         ea_bdebug(bs->bh, "cloning");
   743          0 :                         s->base = kmalloc(bs->bh->b_size, GFP_NOFS);
   744          0 :                         error = -ENOMEM;
   745          0 :                         if (s->base == NULL)
   746          0 :                                 goto cleanup;
   747          0 :                         memcpy(s->base, BHDR(bs->bh), bs->bh->b_size);
   748          0 :                         s->first = ENTRY(header(s->base)+1);
   749          0 :                         header(s->base)->h_refcount = cpu_to_le32(1);
   750          0 :                         s->here = ENTRY(s->base + offset);
   751          0 :                         s->end = s->base + bs->bh->b_size;
   752            :                 }
   753            :         } else {
   754            :                 /* Allocate a buffer where we construct the new block. */
   755          0 :                 s->base = kzalloc(sb->s_blocksize, GFP_NOFS);
   756            :                 /* assert(header == s->base) */
   757          0 :                 error = -ENOMEM;
   758          0 :                 if (s->base == NULL)
   759          0 :                         goto cleanup;
   760          0 :                 header(s->base)->h_magic = cpu_to_le32(EXT4_XATTR_MAGIC);
   761          0 :                 header(s->base)->h_blocks = cpu_to_le32(1);
   762          0 :                 header(s->base)->h_refcount = cpu_to_le32(1);
   763          0 :                 s->first = ENTRY(header(s->base)+1);
   764          0 :                 s->here = ENTRY(header(s->base)+1);
   765          0 :                 s->end = s->base + sb->s_blocksize;
   766            :         }
   767            :
   768          0 :         error = ext4_xattr_set_entry(i, s);
   769          0 :         if (error == -EIO)
   770          0 :                 goto bad_block;
   771          0 :         if (error)
   772          0 :                 goto cleanup;
   773          0 :         if (!IS_LAST_ENTRY(s->first))
   774          0 :                 ext4_xattr_rehash(header(s->base), s->here);
   775            :
   776          0 : inserted:
   777          0 :         if (!IS_LAST_ENTRY(s->first)) {
   778          0 :                 new_bh = ext4_xattr_cache_find(inode, header(s->base), &ce);
```

```
779          0 :                 if (new_bh) {
780            :                         /* We found an identical block in the cache. */
781          0 :                         if (new_bh == bs->bh)
782            :                                 ea_bdebug(new_bh, "keeping");
783            :                         else {
784            :                                 /* The old block is released after updating
785            :                                    the inode. */
786          0 :                                 error = -EDQUOT;
787          0 :                                 if (vfs_dq_alloc_block(inode, 1))
788          0 :                                         goto cleanup;
789          0 :                                 error = ext4_journal_get_write_access(handle,
790            :                                                                       new_bh);
791          0 :                                 if (error)
792          0 :                                         goto cleanup_dquot;
793          0 :                                 lock_buffer(new_bh);
794          0 :                                 le32_add_cpu(&BHDR(new_bh)->h_refcount, 1);
795            :                                 ea_bdebug(new_bh, "reusing; refcount now=%d",
796            :                                         le32_to_cpu(BHDR(new_bh)->h_refcount));
797          0 :                                 unlock_buffer(new_bh);
798          0 :                                 error = ext4_handle_dirty_metadata(handle,
799            :                                                                    inode,
800            :                                                                    new_bh);
801          0 :                                 if (error)
802          0 :                                         goto cleanup_dquot;
803            :                         }
804          0 :                         mb_cache_entry_release(ce);
805          0 :                         ce = NULL;
806          0 :                 } else if (bs->bh && s->base == bs->bh->b_data) {
807            :                         /* We were modifying this block in-place. */
808            :                         ea_bdebug(bs->bh, "keeping this block");
809          0 :                         new_bh = bs->bh;
810            :                         get_bh(new_bh);
811            :                 } else {
812            :                         /* We need to allocate a new block */
813            :                         ext4_fsblk_t goal = ext4_group_first_block_no(sb,
814          0 :                                                 EXT4_I(inode)->i_block_group);
815            :                         ext4_fsblk_t block = ext4_new_meta_blocks(handle, inode,
816          0 :                                                   goal, NULL, &error);
817          0 :                         if (error)
818          0 :                                 goto cleanup;
819            :                         ea_idebug(inode, "creating block %d", block);
820            :
821          0 :                         new_bh = sb_getblk(sb, block);
822          0 :                         if (!new_bh) {
823          0 : getblk_failed:
824          0 :                                 ext4_free_blocks(handle, inode, block, 1, 1);
825          0 :                                 error = -EIO;
826          0 :                                 goto cleanup;
827            :                         }
828          0 :                         lock_buffer(new_bh);
829          0 :                         error = ext4_journal_get_create_access(handle, new_bh);
830          0 :                         if (error) {
831          0 :                                 unlock_buffer(new_bh);
832          0 :                                 goto getblk_failed;
833            :                         }
834          0 :                         memcpy(new_bh->b_data, s->base, new_bh->b_size);
835            :                         set_buffer_uptodate(new_bh);
836          0 :                         unlock_buffer(new_bh);
837          0 :                         ext4_xattr_cache_insert(new_bh);
838          0 :                         error = ext4_handle_dirty_metadata(handle,
839            :                                                            inode, new_bh);
840          0 :                         if (error)
841          0 :                                 goto cleanup;
842            :                 }
843            :         }
844            :
845            :         /* Update the inode. */
846          0 :         EXT4_I(inode)->i_file_acl = new_bh ? new_bh->b_blocknr : 0;
847            :
848            :         /* Drop the previous xattr block. */
849          0 :         if (bs->bh && bs->bh != new_bh)
850          0 :                 ext4_xattr_release_block(handle, inode, bs->bh);
```

```
851                0 :         error = 0;
852                  :
853                0 : cleanup:
854                0 :         if (ce)
855                0 :                 mb_cache_entry_release(ce);
856                  :         brelse(new_bh);
857                0 :         if (!(bs->bh && s->base == bs->bh->b_data))
858                0 :                 kfree(s->base);
859                  :
860                0 :         return error;
861                  :
862                0 : cleanup_dquot:
863                0 :         vfs_dq_free_block(inode, 1);
864                0 :         goto cleanup;
865                  :
866                0 : bad_block:
867                0 :         ext4_error(inode->i_sb, __func__,
868                  :                    "inode %lu: bad block %llu", inode->i_ino,
869                  :                    EXT4_I(inode)->i_file_acl);
870                0 :         goto cleanup;
871                  :
872                  : #undef header
873                  : }
874                  :
875                  : struct ext4_xattr_ibody_find {
876                  :         struct ext4_xattr_search s;
877                  :         struct ext4_iloc iloc;
878                  : };
879                  :
880                  : static int
881                  : ext4_xattr_ibody_find(struct inode *inode, struct ext4_xattr_info *i,
882                  :                       struct ext4_xattr_ibody_find *is)
883                0 : {
884                  :         struct ext4_xattr_ibody_header *header;
885                  :         struct ext4_inode *raw_inode;
886                  :         int error;
887                  :
888                0 :         if (EXT4_I(inode)->i_extra_isize == 0)
889                0 :                 return 0;
890                0 :         raw_inode = ext4_raw_inode(&is->iloc);
891                0 :         header = IHDR(inode, raw_inode);
892                0 :         is->s.base = is->s.first = IFIRST(header);
893                0 :         is->s.here = is->s.first;
894                0 :         is->s.end = (void *)raw_inode + EXT4_SB(inode->i_sb)->s_inode_size;
895                0 :         if (EXT4_I(inode)->i_state & EXT4_STATE_XATTR) {
896                0 :                 error = ext4_xattr_check_names(IFIRST(header), is->s.end);
897                0 :                 if (error)
898                0 :                         return error;
899                  :                 /* Find the named attribute. */
900                0 :                 error = ext4_xattr_find_entry(&is->s.here, i->name_index,
901                  :                                               i->name, is->s.end -
902                  :                                               (void *)is->s.base, 0);
903                0 :                 if (error && error != -ENODATA)
904                0 :                         return error;
905                0 :                 is->s.not_found = error;
906                  :         }
907                0 :         return 0;
908                  : }
909                  :
910                  : static int
911                  : ext4_xattr_ibody_set(handle_t *handle, struct inode *inode,
912                  :                      struct ext4_xattr_info *i,
913                  :                      struct ext4_xattr_ibody_find *is)
914                0 : {
915                  :         struct ext4_xattr_ibody_header *header;
916                0 :         struct ext4_xattr_search *s = &is->s;
917                  :         int error;
918                  :
919                0 :         if (EXT4_I(inode)->i_extra_isize == 0)
920                0 :                 return -ENOSPC;
921                0 :         error = ext4_xattr_set_entry(i, s);
922                0 :         if (error)
```

```
923          0 :                 return error;
924          0 :         header = IHDR(inode, ext4_raw_inode(&is->iloc));
925          0 :         if (!IS_LAST_ENTRY(s->first)) {
926          0 :                 header->h_magic = cpu_to_le32(EXT4_XATTR_MAGIC);
927          0 :                 EXT4_I(inode)->i_state |= EXT4_STATE_XATTR;
928            :         } else {
929          0 :                 header->h_magic = cpu_to_le32(0);
930          0 :                 EXT4_I(inode)->i_state &= ~EXT4_STATE_XATTR;
931            :         }
932          0 :         return 0;
933            : }
934            :
935            : /*
936            :  * ext4_xattr_set_handle()
937            :  *
938            :  * Create, replace or remove an extended attribute for this inode. Buffer
939            :  * is NULL to remove an existing extended attribute, and non-NULL to
940            :  * either replace an existing extended attribute, or create a new extended
941            :  * attribute. The flags XATTR_REPLACE and XATTR_CREATE
942            :  * specify that an extended attribute must exist and must not exist
943            :  * previous to the call, respectively.
944            :  *
945            :  * Returns 0, or a negative error number on failure.
946            :  */
947            : int
948            : ext4_xattr_set_handle(handle_t *handle, struct inode *inode, int name_index,
949            :                       const char *name, const void *value, size_t value_len,
950            :                       int flags)
951          0 : {
952            :         struct ext4_xattr_info i = {
953            :                 .name_index = name_index,
954            :                 .name = name,
955            :                 .value = value,
956            :                 .value_len = value_len,
957            :
958          0 :         };
959            :         struct ext4_xattr_ibody_find is = {
960            :                 .s = { .not_found = -ENODATA, },
961          0 :         };
962            :         struct ext4_xattr_block_find bs = {
963            :                 .s = { .not_found = -ENODATA, },
964          0 :         };
965            :         unsigned long no_expand;
966            :         int error;
967            :
968          0 :         if (!name)
969          0 :                 return -EINVAL;
970          0 :         if (strlen(name) > 255)
971          0 :                 return -ERANGE;
972          0 :         down_write(&EXT4_I(inode)->xattr_sem);
973          0 :         no_expand = EXT4_I(inode)->i_state & EXT4_STATE_NO_EXPAND;
974          0 :         EXT4_I(inode)->i_state |= EXT4_STATE_NO_EXPAND;
975            :
976          0 :         error = ext4_get_inode_loc(inode, &is.iloc);
977          0 :         if (error)
978          0 :                 goto cleanup;
979            :
980          0 :         if (EXT4_I(inode)->i_state & EXT4_STATE_NEW) {
981          0 :                 struct ext4_inode *raw_inode = ext4_raw_inode(&is.iloc);
982          0 :                 memset(raw_inode, 0, EXT4_SB(inode->i_sb)->s_inode_size);
983          0 :                 EXT4_I(inode)->i_state &= ~EXT4_STATE_NEW;
984            :         }
985            :
986          0 :         error = ext4_xattr_ibody_find(inode, &i, &is);
987          0 :         if (error)
988          0 :                 goto cleanup;
989          0 :         if (is.s.not_found)
990          0 :                 error = ext4_xattr_block_find(inode, &i, &bs);
991          0 :         if (error)
992          0 :                 goto cleanup;
993          0 :         if (is.s.not_found && bs.s.not_found) {
994          0 :                 error = -ENODATA;
```

```
995        0 :                 if (flags & XATTR_REPLACE)
996        0 :                         goto cleanup;
997        0 :                 error = 0;
998        0 :                 if (!value)
999        0 :                         goto cleanup;
1000         :         } else {
1001       0 :                 error = -EEXIST;
1002       0 :                 if (flags & XATTR_CREATE)
1003       0 :                         goto cleanup;
1004         :         }
1005       0 :         error = ext4_journal_get_write_access(handle, is.iloc.bh);
1006       0 :         if (error)
1007       0 :                 goto cleanup;
1008       0 :         if (!value) {
1009       0 :                 if (!is.s.not_found)
1010       0 :                         error = ext4_xattr_ibody_set(handle, inode, &i, &is);
1011       0 :                 else if (!bs.s.not_found)
1012       0 :                         error = ext4_xattr_block_set(handle, inode, &i, &bs);
1013         :         } else {
1014       0 :                 error = ext4_xattr_ibody_set(handle, inode, &i, &is);
1015       0 :                 if (!error && !bs.s.not_found) {
1016       0 :                         i.value = NULL;
1017       0 :                         error = ext4_xattr_block_set(handle, inode, &i, &bs);
1018       0 :                 } else if (error == -ENOSPC) {
1019       0 :                         if (EXT4_I(inode)->i_file_acl && !bs.s.base) {
1020       0 :                                 error = ext4_xattr_block_find(inode, &i, &bs);
1021       0 :                                 if (error)
1022       0 :                                         goto cleanup;
1023         :                         }
1024       0 :                         error = ext4_xattr_block_set(handle, inode, &i, &bs);
1025       0 :                         if (error)
1026       0 :                                 goto cleanup;
1027       0 :                         if (!is.s.not_found) {
1028       0 :                                 i.value = NULL;
1029       0 :                                 error = ext4_xattr_ibody_set(handle, inode, &i,
1030         :                                                              &is);
1031         :                         }
1032         :                 }
1033         :         }
1034       0 :         if (!error) {
1035       0 :                 ext4_xattr_update_super_block(handle, inode->i_sb);
1036       0 :                 inode->i_ctime = ext4_current_time(inode);
1037       0 :                 if (!value)
1038       0 :                         EXT4_I(inode)->i_state &= ~EXT4_STATE_NO_EXPAND;
1039       0 :                 error = ext4_mark_iloc_dirty(handle, inode, &is.iloc);
1040         :                 /*
1041         :                  * The bh is consumed by ext4_mark_iloc_dirty, even with
1042         :                  * error != 0.
1043         :                  */
1044       0 :                 is.iloc.bh = NULL;
1045       0 :                 if (IS_SYNC(inode))
1046         :                         ext4_handle_sync(handle);
1047         :         }
1048         :
1049       0 : cleanup:
1050       0 :         brelse(is.iloc.bh);
1051       0 :         brelse(bs.bh);
1052       0 :         if (no_expand == 0)
1053       0 :                 EXT4_I(inode)->i_state &= ~EXT4_STATE_NO_EXPAND;
1054       0 :         up_write(&EXT4_I(inode)->xattr_sem);
1055       0 :         return error;
1056         : }
1057         :
1058         : /*
1059         :  * ext4_xattr_set()
1060         :  *
1061         :  * Like ext4_xattr_set_handle, but start from an inode. This extended
1062         :  * attribute modification is a filesystem transaction by itself.
1063         :  *
1064         :  * Returns 0, or a negative error number on failure.
1065         :  */
1066         : int
```

```
1067                 : ext4_xattr_set(struct inode *inode, int name_index, const char *name,
1068                 :                const void *value, size_t value_len, int flags)
1069               0 : {
1070                 :         handle_t *handle;
1071               0 :         int error, retries = 0;
1072                 :
1073               0 : retry:
1074               0 :         handle = ext4_journal_start(inode, EXT4_DATA_TRANS_BLOCKS(inode->i_sb));
1075               0 :         if (IS_ERR(handle)) {
1076               0 :                 error = PTR_ERR(handle);
1077                 :         } else {
1078                 :                 int error2;
1079                 :
1080               0 :                 error = ext4_xattr_set_handle(handle, inode, name_index, name,
1081                 :                                               value, value_len, flags);
1082               0 :                 error2 = ext4_journal_stop(handle);
1083               0 :                 if (error == -ENOSPC &&
1084                 :                     ext4_should_retry_alloc(inode->i_sb, &retries))
1085               0 :                         goto retry;
1086               0 :                 if (error == 0)
1087               0 :                         error = error2;
1088                 :         }
1089                 :
1090               0 :         return error;
1091                 : }
1092                 :
1093                 : /*
1094                 :  * Shift the EA entries in the inode to create space for the increased
1095                 :  * i_extra_isize.
1096                 :  */
1097                 : static void ext4_xattr_shift_entries(struct ext4_xattr_entry *entry,
1098                 :                                      int value_offs_shift, void *to,
1099                 :                                      void *from, size_t n, int blocksize)
1100                 : {
1101               0 :         struct ext4_xattr_entry *last = entry;
1102                 :         int new_offs;
1103                 :
1104                 :         /* Adjust the value offsets of the entries */
1105               0 :         for (; !IS_LAST_ENTRY(last); last = EXT4_XATTR_NEXT(last)) {
1106               0 :                 if (!last->e_value_block && last->e_value_size) {
1107               0 :                         new_offs = le16_to_cpu(last->e_value_offs) +
1108                 :                                                         value_offs_shift;
1109               0 :                         BUG_ON(new_offs + le32_to_cpu(last->e_value_size)
1110                 :                                 > blocksize);
1111               0 :                         last->e_value_offs = cpu_to_le16(new_offs);
1112                 :                 }
1113                 :         }
1114                 :         /* Shift the entries by n bytes */
1115               0 :         memmove(to, from, n);
1116                 : }
1117                 :
1118                 : /*
1119                 :  * Expand an inode by new_extra_isize bytes when EAs are present.
1120                 :  * Returns 0 on success or negative error number on failure.
1121                 :  */
1122                 : int ext4_expand_extra_isize_ea(struct inode *inode, int new_extra_isize,
1123                 :                                struct ext4_inode *raw_inode, handle_t *handle)
1124               0 : {
1125                 :         struct ext4_xattr_ibody_header *header;
1126                 :         struct ext4_xattr_entry *entry, *last, *first;
1127               0 :         struct buffer_head *bh = NULL;
1128               0 :         struct ext4_xattr_ibody_find *is = NULL;
1129               0 :         struct ext4_xattr_block_find *bs = NULL;
1130               0 :         char *buffer = NULL, *b_entry_name = NULL;
1131                 :         size_t min_offs, free;
1132                 :         int total_ino, total_blk;
1133                 :         void *base, *start, *end;
1134               0 :         int extra_isize = 0, error = 0, tried_min_extra_isize = 0;
1135               0 :         int s_min_extra_isize = le16_to_cpu(EXT4_SB(inode->i_sb)->s_es->s_min_extra_isize);
1136                 :
1137               0 :         down_write(&EXT4_I(inode)->xattr_sem);
1138               0 : retry:
```

```
1139          0 :         if (EXT4_I(inode)->i_extra_isize >= new_extra_isize) {
1140          0 :                 up_write(&EXT4_I(inode)->xattr_sem);
1141          0 :                 return 0;
1142            :         }
1143            :
1144          0 :         header = IHDR(inode, raw_inode);
1145          0 :         entry = IFIRST(header);
1146            :
1147            :         /*
1148            :          * Check if enough free space is available in the inode to shift the
1149            :          * entries ahead by new_extra_isize.
1150            :          */
1151            :
1152          0 :         base = start = entry;
1153          0 :         end = (void *)raw_inode + EXT4_SB(inode->i_sb)->s_inode_size;
1154          0 :         min_offs = end - base;
1155          0 :         last = entry;
1156          0 :         total_ino = sizeof(struct ext4_xattr_ibody_header);
1157            :
1158          0 :         free = ext4_xattr_free_space(last, &min_offs, base, &total_ino);
1159          0 :         if (free >= new_extra_isize) {
1160          0 :                 entry = IFIRST(header);
1161          0 :                 ext4_xattr_shift_entries(entry, EXT4_I(inode)->i_extra_isize
1162            :                                          - new_extra_isize, (void *)raw_inode +
1163            :                                          EXT4_GOOD_OLD_INODE_SIZE + new_extra_isize,
1164            :                                          (void *)header, total_ino,
1165            :                                          inode->i_sb->s_blocksize);
1166          0 :                 EXT4_I(inode)->i_extra_isize = new_extra_isize;
1167          0 :                 error = 0;
1168          0 :                 goto cleanup;
1169            :         }
1170            :
1171            :         /*
1172            :          * Enough free space isn't available in the inode, check if
1173            :          * EA block can hold new_extra_isize bytes.
1174            :          */
1175          0 :         if (EXT4_I(inode)->i_file_acl) {
1176          0 :                 bh = sb_bread(inode->i_sb, EXT4_I(inode)->i_file_acl);
1177          0 :                 error = -EIO;
1178          0 :                 if (!bh)
1179          0 :                         goto cleanup;
1180          0 :                 if (ext4_xattr_check_block(bh)) {
1181          0 :                         ext4_error(inode->i_sb, __func__,
1182            :                                    "inode %lu: bad block %llu", inode->i_ino,
1183            :                                    EXT4_I(inode)->i_file_acl);
1184          0 :                         error = -EIO;
1185          0 :                         goto cleanup;
1186            :                 }
1187          0 :                 base = BHDR(bh);
1188          0 :                 first = BFIRST(bh);
1189          0 :                 end = bh->b_data + bh->b_size;
1190          0 :                 min_offs = end - base;
1191          0 :                 free = ext4_xattr_free_space(first, &min_offs, base,
1192            :                                              &total_blk);
1193          0 :                 if (free < new_extra_isize) {
1194          0 :                         if (!tried_min_extra_isize && s_min_extra_isize) {
1195          0 :                                 tried_min_extra_isize++;
1196          0 :                                 new_extra_isize = s_min_extra_isize;
1197            :                                 brelse(bh);
1198            :                                 goto retry;
1199            :                         }
1200          0 :                         error = -1;
1201          0 :                         goto cleanup;
1202            :                 }
1203            :         } else {
1204          0 :                 free = inode->i_sb->s_blocksize;
1205            :         }
1206            :
1207          0 :         while (new_extra_isize > 0) {
1208            :                 size_t offs, size, entry_size;
1209          0 :                 struct ext4_xattr_entry *small_entry = NULL;
1210            :                 struct ext4_xattr_info i = {
```

```
1211                :                           .value = NULL,
1212                :                           .value_len = 0,
1213              0 :                   };
1214                :                   unsigned int total_size;  /* EA entry size + value size */
1215                :                   unsigned int shift_bytes; /* No. of bytes to shift EAs by? */
1216              0 :                   unsigned int min_total_size = ~0U;
1217                :
1218              0 :                   is = kzalloc(sizeof(struct ext4_xattr_ibody_find), GFP_NOFS);
1219              0 :                   bs = kzalloc(sizeof(struct ext4_xattr_block_find), GFP_NOFS);
1220              0 :                   if (!is || !bs) {
1221              0 :                           error = -ENOMEM;
1222              0 :                           goto cleanup;
1223                :                   }
1224                :
1225              0 :                   is->s.not_found = -ENODATA;
1226              0 :                   bs->s.not_found = -ENODATA;
1227              0 :                   is->iloc.bh = NULL;
1228              0 :                   bs->bh = NULL;
1229                :
1230              0 :                   last = IFIRST(header);
1231                :                   /* Find the entry best suited to be pushed into EA block */
1232              0 :                   entry = NULL;
1233              0 :                   for (; !IS_LAST_ENTRY(last); last = EXT4_XATTR_NEXT(last)) {
1234              0 :                           total_size =
1235                :                           EXT4_XATTR_SIZE(le32_to_cpu(last->e_value_size)) +
1236                :                                           EXT4_XATTR_LEN(last->e_name_len);
1237              0 :                           if (total_size <= free && total_size < min_total_size) {
1238              0 :                                   if (total_size < new_extra_isize) {
1239              0 :                                           small_entry = last;
1240                :                                   } else {
1241              0 :                                           entry = last;
1242              0 :                                           min_total_size = total_size;
1243                :                                   }
1244                :                           }
1245                :                   }
1246                :
1247              0 :                   if (entry == NULL) {
1248              0 :                           if (small_entry) {
1249              0 :                                   entry = small_entry;
1250                :                           } else {
1251              0 :                                   if (!tried_min_extra_isize &&
1252                :                                       s_min_extra_isize) {
1253              0 :                                           tried_min_extra_isize++;
1254              0 :                                           new_extra_isize = s_min_extra_isize;
1255              0 :                                           goto retry;
1256                :                                   }
1257              0 :                                   error = -1;
1258              0 :                                   goto cleanup;
1259                :                           }
1260                :                   }
1261              0 :                   offs = le16_to_cpu(entry->e_value_offs);
1262              0 :                   size = le32_to_cpu(entry->e_value_size);
1263              0 :                   entry_size = EXT4_XATTR_LEN(entry->e_name_len);
1264              0 :                   i.name_index = entry->e_name_index,
1265                :                   buffer = kmalloc(EXT4_XATTR_SIZE(size), GFP_NOFS);
1266              0 :                   b_entry_name = kmalloc(entry->e_name_len + 1, GFP_NOFS);
1267              0 :                   if (!buffer || !b_entry_name) {
1268              0 :                           error = -ENOMEM;
1269              0 :                           goto cleanup;
1270                :                   }
1271                :                   /* Save the entry name and the entry value */
1272              0 :                   memcpy(buffer, (void *)IFIRST(header) + offs,
1273                :                          EXT4_XATTR_SIZE(size));
1274              0 :                   memcpy(b_entry_name, entry->e_name, entry->e_name_len);
1275              0 :                   b_entry_name[entry->e_name_len] = '\0';
1276              0 :                   i.name = b_entry_name;
1277                :
1278              0 :                   error = ext4_get_inode_loc(inode, &is->iloc);
1279              0 :                   if (error)
1280              0 :                           goto cleanup;
1281                :
1282              0 :                   error = ext4_xattr_ibody_find(inode, &i, is);
```

```
1283          0 :                     if (error)
1284          0 :                             goto cleanup;
1285            :
1286            :                     /* Remove the chosen entry from the inode */
1287          0 :                     error = ext4_xattr_ibody_set(handle, inode, &i, is);
1288            :
1289          0 :                     entry = IFIRST(header);
1290          0 :                     if (entry_size + EXT4_XATTR_SIZE(size) >= new_extra_isize)
1291          0 :                             shift_bytes = new_extra_isize;
1292            :                     else
1293          0 :                             shift_bytes = entry_size + size;
1294            :                     /* Adjust the offsets and shift the remaining entries ahead */
1295          0 :                     ext4_xattr_shift_entries(entry, EXT4_I(inode)->i_extra_isize -
1296            :                             shift_bytes, (void *)raw_inode +
1297            :                             EXT4_GOOD_OLD_INODE_SIZE + extra_isize + shift_bytes,
1298            :                             (void *)header, total_ino - entry_size,
1299            :                             inode->i_sb->s_blocksize);
1300            :
1301          0 :                     extra_isize += shift_bytes;
1302          0 :                     new_extra_isize -= shift_bytes;
1303          0 :                     EXT4_I(inode)->i_extra_isize = extra_isize;
1304            :
1305          0 :                     i.name = b_entry_name;
1306          0 :                     i.value = buffer;
1307          0 :                     i.value_len = size;
1308          0 :                     error = ext4_xattr_block_find(inode, &i, bs);
1309          0 :                     if (error)
1310          0 :                             goto cleanup;
1311            :
1312            :                     /* Add entry which was removed from the inode into the block */
1313          0 :                     error = ext4_xattr_block_set(handle, inode, &i, bs);
1314          0 :                     if (error)
1315          0 :                             goto cleanup;
1316          0 :                     kfree(b_entry_name);
1317          0 :                     kfree(buffer);
1318          0 :                     brelse(is->iloc.bh);
1319          0 :                     kfree(is);
1320          0 :                     kfree(bs);
1321            :             }
1322            :             brelse(bh);
1323          0 :             up_write(&EXT4_I(inode)->xattr_sem);
1324          0 :             return 0;
1325            :
1326          0 : cleanup:
1327          0 :             kfree(b_entry_name);
1328          0 :             kfree(buffer);
1329          0 :             if (is)
1330          0 :                     brelse(is->iloc.bh);
1331          0 :             kfree(is);
1332          0 :             kfree(bs);
1333            :             brelse(bh);
1334          0 :             up_write(&EXT4_I(inode)->xattr_sem);
1335          0 :             return error;
1336            : }
1337            :
1338            :
1339            :
1340            : /*
1341            :  * ext4_xattr_delete_inode()
1342            :  *
1343            :  * Free extended attribute resources associated with this inode. This
1344            :  * is called immediately before an inode is freed. We have exclusive
1345            :  * access to the inode.
1346            :  */
1347            : void
1348            : ext4_xattr_delete_inode(handle_t *handle, struct inode *inode)
1349    1664354 : {
1350    1664354 :             struct buffer_head *bh = NULL;
1351            :
1352    1664354 :             if (!EXT4_I(inode)->i_file_acl)
1353    1664354 :                     goto cleanup;
1354          0 :             bh = sb_bread(inode->i_sb, EXT4_I(inode)->i_file_acl);
```

```
1355                0 :         if (!bh) {
1356                0 :                 ext4_error(inode->i_sb, __func__,
1357                  :                         "inode %lu: block %llu read error", inode->i_ino,
1358                  :                         EXT4_I(inode)->i_file_acl);
1359                0 :                 goto cleanup;
1360                  :         }
1361                0 :         if (BHDR(bh)->h_magic != cpu_to_le32(EXT4_XATTR_MAGIC) ||
1362                  :             BHDR(bh)->h_blocks != cpu_to_le32(1)) {
1363                0 :                 ext4_error(inode->i_sb, __func__,
1364                  :                         "inode %lu: bad block %llu", inode->i_ino,
1365                  :                         EXT4_I(inode)->i_file_acl);
1366                0 :                 goto cleanup;
1367                  :         }
1368                0 :         ext4_xattr_release_block(handle, inode, bh);
1369                0 :         EXT4_I(inode)->i_file_acl = 0;
1370                  :
1371          1664354 : cleanup:
1372                  :         brelse(bh);
1373          1664354 : }
1374                  :
1375                  : /*
1376                  :  * ext4_xattr_put_super()
1377                  :  *
1378                  :  * This is called when a file system is unmounted.
1379                  :  */
1380                  : void
1381                  : ext4_xattr_put_super(struct super_block *sb)
1382               94 : {
1383               94 :         mb_cache_shrink(sb->s_bdev);
1384               94 : }
1385                  :
1386                  : /*
1387                  :  * ext4_xattr_cache_insert()
1388                  :  *
1389                  :  * Create a new entry in the extended attribute cache, and insert
1390                  :  * it unless such an entry is already in the cache.
1391                  :  *
1392                  :  * Returns 0, or a negative error number on failure.
1393                  :  */
1394                  : static void
1395                  : ext4_xattr_cache_insert(struct buffer_head *bh)
1396                0 : {
1397                0 :         __u32 hash = le32_to_cpu(BHDR(bh)->h_hash);
1398                  :         struct mb_cache_entry *ce;
1399                  :         int error;
1400                  :
1401                0 :         ce = mb_cache_entry_alloc(ext4_xattr_cache, GFP_NOFS);
1402                0 :         if (!ce) {
1403                  :                 ea_bdebug(bh, "out of memory");
1404                0 :                 return;
1405                  :         }
1406                0 :         error = mb_cache_entry_insert(ce, bh->b_bdev, bh->b_blocknr, &hash);
1407                0 :         if (error) {
1408                0 :                 mb_cache_entry_free(ce);
1409                0 :                 if (error == -EBUSY) {
1410                  :                         ea_bdebug(bh, "already in cache");
1411                0 :                         error = 0;
1412                  :                 }
1413                  :         } else {
1414                  :                 ea_bdebug(bh, "inserting [%x]", (int)hash);
1415                0 :                 mb_cache_entry_release(ce);
1416                  :         }
1417                  : }
1418                  :
1419                  : /*
1420                  :  * ext4_xattr_cmp()
1421                  :  *
1422                  :  * Compare two extended attribute blocks for equality.
1423                  :  *
1424                  :  * Returns 0 if the blocks are equal, 1 if they differ, and
1425                  :  * a negative error number on errors.
1426                  :  */
```

```
1427                 : static int
1428                 : ext4_xattr_cmp(struct ext4_xattr_header *header1,
1429                 :                struct ext4_xattr_header *header2)
1430                 : {
1431                 :         struct ext4_xattr_entry *entry1, *entry2;
1432                 :
1433               0 :         entry1 = ENTRY(header1+1);
1434               0 :         entry2 = ENTRY(header2+1);
1435               0 :         while (!IS_LAST_ENTRY(entry1)) {
1436               0 :                 if (IS_LAST_ENTRY(entry2))
1437               0 :                         return 1;
1438               0 :                 if (entry1->e_hash != entry2->e_hash ||
1439                 :                     entry1->e_name_index != entry2->e_name_index ||
1440                 :                     entry1->e_name_len != entry2->e_name_len ||
1441                 :                     entry1->e_value_size != entry2->e_value_size ||
1442                 :                     memcmp(entry1->e_name, entry2->e_name, entry1->e_name_len))
1443               0 :                         return 1;
1444               0 :                 if (entry1->e_value_block != 0 || entry2->e_value_block != 0)
1445               0 :                         return -EIO;
1446               0 :                 if (memcmp((char *)header1 + le16_to_cpu(entry1->e_value_offs),
1447                 :                            (char *)header2 + le16_to_cpu(entry2->e_value_offs),
1448                 :                            le32_to_cpu(entry1->e_value_size)))
1449               0 :                         return 1;
1450                 :
1451               0 :                 entry1 = EXT4_XATTR_NEXT(entry1);
1452               0 :                 entry2 = EXT4_XATTR_NEXT(entry2);
1453                 :         }
1454               0 :         if (!IS_LAST_ENTRY(entry2))
1455               0 :                 return 1;
1456               0 :         return 0;
1457                 : }
1458                 :
1459                 : /*
1460                 :  * ext4_xattr_cache_find()
1461                 :  *
1462                 :  * Find an identical extended attribute block.
1463                 :  *
1464                 :  * Returns a pointer to the block found, or NULL if such a block was
1465                 :  * not found or an error occurred.
1466                 :  */
1467                 : static struct buffer_head *
1468                 : ext4_xattr_cache_find(struct inode *inode, struct ext4_xattr_header *header,
1469                 :                       struct mb_cache_entry **pce)
1470               0 : {
1471               0 :         __u32 hash = le32_to_cpu(header->h_hash);
1472                 :         struct mb_cache_entry *ce;
1473                 :
1474               0 :         if (!header->h_hash)
1475               0 :                 return NULL;  /* never share */
1476                 :         ea_idebug(inode, "looking for cached blocks [%x]", (int)hash);
1477               0 : again:
1478               0 :         ce = mb_cache_entry_find_first(ext4_xattr_cache, 0,
1479                 :                                        inode->i_sb->s_bdev, hash);
1480               0 :         while (ce) {
1481                 :                 struct buffer_head *bh;
1482                 :
1483               0 :                 if (IS_ERR(ce)) {
1484               0 :                         if (PTR_ERR(ce) == -EAGAIN)
1485               0 :                                 goto again;
1486                 :                         break;
1487                 :                 }
1488               0 :                 bh = sb_bread(inode->i_sb, ce->e_block);
1489               0 :                 if (!bh) {
1490               0 :                         ext4_error(inode->i_sb, __func__,
1491                 :                                    "inode %lu: block %lu read error",
1492                 :                                    inode->i_ino, (unsigned long) ce->e_block);
1493               0 :                 } else if (le32_to_cpu(BHDR(bh)->h_refcount) >=
1494                 :                                    EXT4_XATTR_REFCOUNT_MAX) {
1495                 :                         ea_idebug(inode, "block %lu refcount %d>=%d",
1496                 :                                   (unsigned long) ce->e_block,
1497                 :                                   le32_to_cpu(BHDR(bh)->h_refcount),
1498                 :                                           EXT4_XATTR_REFCOUNT_MAX);
```

```
1499          0 :                 } else if (ext4_xattr_cmp(header, BHDR(bh)) == 0) {
1500          0 :                         *pce = ce;
1501          0 :                         return bh;
1502            :                 }
1503            :                 brelse(bh);
1504          0 :                 ce = mb_cache_entry_find_next(ce, 0, inode->i_sb->s_bdev, hash);
1505            :         }
1506          0 :         return NULL;
1507            : }
1508            :
1509            : #define NAME_HASH_SHIFT 5
1510            : #define VALUE_HASH_SHIFT 16
1511            :
1512            : /*
1513            :  * ext4_xattr_hash_entry()
1514            :  *
1515            :  * Compute the hash of an extended attribute.
1516            :  */
1517            : static inline void ext4_xattr_hash_entry(struct ext4_xattr_header *header,
1518            :                                          struct ext4_xattr_entry *entry)
1519            : {
1520          0 :         __u32 hash = 0;
1521          0 :         char *name = entry->e_name;
1522            :         int n;
1523            :
1524          0 :         for (n = 0; n < entry->e_name_len; n++) {
1525          0 :                 hash = (hash << NAME_HASH_SHIFT) ^
1526            :                        (hash >> (8*sizeof(hash) - NAME_HASH_SHIFT)) ^
1527            :                        *name++;
1528            :         }
1529            :
1530          0 :         if (entry->e_value_block == 0 && entry->e_value_size != 0) {
1531            :                 __le32 *value = (__le32 *)((char *)header +
1532          0 :                         le16_to_cpu(entry->e_value_offs));
1533            :                 for (n = (le32_to_cpu(entry->e_value_size) +
1534          0 :                      EXT4_XATTR_ROUND) >> EXT4_XATTR_PAD_BITS; n; n--) {
1535          0 :                         hash = (hash << VALUE_HASH_SHIFT) ^
1536            :                                (hash >> (8*sizeof(hash) - VALUE_HASH_SHIFT)) ^
1537            :                                le32_to_cpu(*value++);
1538            :                 }
1539            :         }
1540          0 :         entry->e_hash = cpu_to_le32(hash);
1541            : }
1542            :
1543            : #undef NAME_HASH_SHIFT
1544            : #undef VALUE_HASH_SHIFT
1545            :
1546            : #define BLOCK_HASH_SHIFT 16
1547            :
1548            : /*
1549            :  * ext4_xattr_rehash()
1550            :  *
1551            :  * Re-compute the extended attribute hash value after an entry has changed.
1552            :  */
1553            : static void ext4_xattr_rehash(struct ext4_xattr_header *header,
1554            :                               struct ext4_xattr_entry *entry)
1555          0 : {
1556            :         struct ext4_xattr_entry *here;
1557          0 :         __u32 hash = 0;
1558            :
1559            :         ext4_xattr_hash_entry(header, entry);
1560          0 :         here = ENTRY(header+1);
1561          0 :         while (!IS_LAST_ENTRY(here)) {
1562          0 :                 if (!here->e_hash) {
1563            :                         /* Block is not shared if an entry's hash value == 0 */
1564          0 :                         hash = 0;
1565          0 :                         break;
1566            :                 }
1567          0 :                 hash = (hash << BLOCK_HASH_SHIFT) ^
1568            :                        (hash >> (8*sizeof(hash) - BLOCK_HASH_SHIFT)) ^
1569            :                        le32_to_cpu(here->e_hash);
1570          0 :                 here = EXT4_XATTR_NEXT(here);
```

```
1571                :         }
1572              0 :         header->h_hash = cpu_to_le32(hash);
1573              0 : }
1574                :
1575                : #undef BLOCK_HASH_SHIFT
1576                :
1577                : int __init
1578                : init_ext4_xattr(void)
1579              0 : {
1580              0 :         ext4_xattr_cache = mb_cache_create("ext4_xattr", NULL,
1581                :                 sizeof(struct mb_cache_entry) +
1582                :                 sizeof(((struct mb_cache_entry *) 0)->e_indexes[0]), 1, 6);
1583              0 :         if (!ext4_xattr_cache)
1584              0 :                 return -ENOMEM;
1585              0 :         return 0;
1586                : }
1587                :
1588                : void
1589                : exit_ext4_xattr(void)
1590              0 : {
1591              0 :         if (ext4_xattr_cache)
1592              0 :                 mb_cache_destroy(ext4_xattr_cache);
1593              0 :         ext4_xattr_cache = NULL;
1594              0 : }
```