LCOV - code coverage report

Current view: directory - fs/ext4 - super.c (source / functions)		Found	Hit	Coverage
Test: kernel_2_6_31_ext4_round_3.info	Lines:	1787	699	39.1 %
Date: 2009-10-24	Functions:	80	41	51.2 %

```
        :/*
        : *  linux/fs/ext4/super.c
        : *
        : * Copyright (C) 1992, 1993, 1994, 1995
        : * Remy Card (card@masi.ibp.fr)
        : * Laboratoire MASI - Institut Blaise Pascal
        : * Universite Pierre et Marie Curie (Paris VI)
        : *
        : *  from
        : *
        : *  linux/fs/minix/inode.c
        : *
        : *  Copyright (C) 1991, 1992  Linus Torvalds
        : *
        : *  Big-endian to little-endian byte-swapping/bitmaps by
        : *        David S. Miller (davem@caip.rutgers.edu), 1995
        : */
        :
        :#include <linux/module.h>
        :#include <linux/string.h>
        :#include <linux/fs.h>
        :#include <linux/time.h>
        :#include <linux/vmalloc.h>
        :#include <linux/jbd2.h>
        :#include <linux/slab.h>
        :#include <linux/init.h>
        :#include <linux/blkdev.h>
        :#include <linux/parser.h>
        :#include <linux/smp_lock.h>
        :#include <linux/buffer_head.h>
        :#include <linux/exportfs.h>
        :#include <linux/vfs.h>
        :#include <linux/random.h>
        :#include <linux/mount.h>
        :#include <linux/namei.h>
        :#include <linux/quotaops.h>
        :#include <linux/seq_file.h>
        :#include <linux/proc_fs.h>
        :#include <linux/ctype.h>
        :#include <linux/log2.h>
        :#include <linux/crc16.h>
        :#include <asm/uaccess.h>
        :
        :#include "ext4.h"
        :#include "ext4_jbd2.h"
        :#include "xattr.h"
        :#include "acl.h"
        :
        :#define CREATE_TRACE_POINTS
        :#include <trace/events/ext4.h>
        :
        :static int default_mb_history_length = 1000;
        :
        :module_param_named(default_mb_history_length, default_mb_history_length,
        :                   int, 0644);
        :MODULE_PARM_DESC(default_mb_history_length,
        :                 "Default number of entries saved for mb_history");
        :
        :struct proc_dir_entry *ext4_proc_root;
        :static struct kset *ext4_kset;
        :
        :static int ext4_load_journal(struct super_block *, struct ext4_super_block *,
        :                             unsigned long journal_devnum);
        :static int ext4_commit_super(struct super_block *sb, int sync);
        :static void ext4_mark_recovery_complete(struct super_block *sb,
        :                                        struct ext4_super_block *es);
        :static void ext4_clear_journal_err(struct super_block *sb,
        :                                   struct ext4_super_block *es);
        :static int ext4_sync_fs(struct super_block *sb, int wait);
        :static const char *ext4_decode_error(struct super_block *sb, int errno,
        :                                     char nbuf[16]);
        :static int ext4_remount(struct super_block *sb, int *flags, char *data);
        :static int ext4_statfs(struct dentry *dentry, struct kstatfs *buf);
        :static int ext4_unfreeze(struct super_block *sb);
```

```
     75                :  static void ext4_write_super(struct super_block *sb);
     76                :  static int ext4_freeze(struct super_block *sb);
     77                :
     78                :
     79                :  ext4_fsblk_t ext4_block_bitmap(struct super_block *sb,
     80                :                                 struct ext4_group_desc *bg)
     81        239018831 :  {
     82        481475742 :          return le32_to_cpu(bg->bg_block_bitmap_lo) |
     83                :                  (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT ?
     84                :                   (ext4_fsblk_t)le32_to_cpu(bg->bg_block_bitmap_hi) << 32 : 0);
     85                :  }
     86                :
     87                :  ext4_fsblk_t ext4_inode_bitmap(struct super_block *sb,
     88                :                                 struct ext4_group_desc *bg)
     89         17397000 :  {
     90         38232080 :          return le32_to_cpu(bg->bg_inode_bitmap_lo) |
     91                :                  (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT ?
     92                :                   (ext4_fsblk_t)le32_to_cpu(bg->bg_inode_bitmap_hi) << 32 : 0);
     93                :  }
     94                :
     95                :  ext4_fsblk_t ext4_inode_table(struct super_block *sb,
     96                :                                struct ext4_group_desc *bg)
     97       1819523582 :  {
     98       -652482052 :          return le32_to_cpu(bg->bg_inode_table_lo) |
     99                :                  (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT ?
    100                :                   (ext4_fsblk_t)le32_to_cpu(bg->bg_inode_table_hi) << 32 : 0);
    101                :  }
    102                :
    103                :  __u32 ext4_free_blks_count(struct super_block *sb,
    104                :                                struct ext4_group_desc *bg)
    105        230150063 :  {
    106        463390046 :          return le16_to_cpu(bg->bg_free_blocks_count_lo) |
    107                :                  (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT ?
    108                :                   (__u32)le16_to_cpu(bg->bg_free_blocks_count_hi) << 16 : 0);
    109                :  }
    110                :
    111                :  __u32 ext4_free_inodes_count(struct super_block *sb,
    112                :                                struct ext4_group_desc *bg)
    113                -1 :  {
    114                -1 :          return le16_to_cpu(bg->bg_free_inodes_count_lo) |
    115                :                  (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT ?
    116                :                   (__u32)le16_to_cpu(bg->bg_free_inodes_count_hi) << 16 : 0);
    117                :  }
    118                :
    119                :  __u32 ext4_used_dirs_count(struct super_block *sb,
    120                :                                struct ext4_group_desc *bg)
    121          3246237 :  {
    122          9582394 :          return le16_to_cpu(bg->bg_used_dirs_count_lo) |
    123                :                  (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT ?
    124                :                   (__u32)le16_to_cpu(bg->bg_used_dirs_count_hi) << 16 : 0);
    125                :  }
    126                :
    127                :  __u32 ext4_itable_unused_count(struct super_block *sb,
    128                :                                struct ext4_group_desc *bg)
    129          9132646 :  {
    130         18265292 :          return le16_to_cpu(bg->bg_itable_unused_lo) |
    131                :                  (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT ?
    132                :                   (__u32)le16_to_cpu(bg->bg_itable_unused_hi) << 16 : 0);
    133                :  }
    134                :
    135                :  void ext4_block_bitmap_set(struct super_block *sb,
    136                :                             struct ext4_group_desc *bg, ext4_fsblk_t blk)
    137                0 :  {
    138                0 :          bg->bg_block_bitmap_lo = cpu_to_le32((u32)blk);
    139                0 :          if (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT)
    140                0 :                  bg->bg_block_bitmap_hi = cpu_to_le32(blk >> 32);
    141                0 :  }
    142                :
    143                :  void ext4_inode_bitmap_set(struct super_block *sb,
    144                :                             struct ext4_group_desc *bg, ext4_fsblk_t blk)
    145                0 :  {
    146                0 :          bg->bg_inode_bitmap_lo  = cpu_to_le32((u32)blk);
    147                0 :          if (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT)
    148                0 :                  bg->bg_inode_bitmap_hi = cpu_to_le32(blk >> 32);
    149                0 :  }
    150                :
    151                :  void ext4_inode_table_set(struct super_block *sb,
    152                :                            struct ext4_group_desc *bg, ext4_fsblk_t blk)
    153                0 :  {
    154                0 :          bg->bg_inode_table_lo = cpu_to_le32((u32)blk);
    155                0 :          if (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT)
    156                0 :                  bg->bg_inode_table_hi = cpu_to_le32(blk >> 32);
    157                0 :  }
    158                :
    159                :  void ext4_free_blks_set(struct super_block *sb,
    160                :                            struct ext4_group_desc *bg, __u32 count)
    161        226048966 :  {
    162        226048966 :          bg->bg_free_blocks_count_lo = cpu_to_le16((__u16)count);
    163        226048966 :          if (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT)
    164                0 :                  bg->bg_free_blocks_count_hi = cpu_to_le16(count >> 16);
```

```
   165  226048966 : }
   166            :
   167            : void ext4_free_inodes_set(struct super_block *sb,
   168            :                           struct ext4_group_desc *bg, __u32 count)
   169   10976927 : {
   170   10976927 :         bg->bg_free_inodes_count_lo = cpu_to_le16((__u16)count);
   171   10976927 :         if (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT)
   172          0 :                 bg->bg_free_inodes_count_hi = cpu_to_le16(count >> 16);
   173   10976927 : }
   174            :
   175            : void ext4_used_dirs_set(struct super_block *sb,
   176            :                           struct ext4_group_desc *bg, __u32 count)
   177    1353076 : {
   178    1353076 :         bg->bg_used_dirs_count_lo = cpu_to_le16((__u16)count);
   179    1353076 :         if (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT)
   180          0 :                 bg->bg_used_dirs_count_hi = cpu_to_le16(count >> 16);
   181    1353076 : }
   182            :
   183            : void ext4_itable_unused_set(struct super_block *sb,
   184            :                           struct ext4_group_desc *bg, __u32 count)
   185    8475822 : {
   186    8475822 :         bg->bg_itable_unused_lo = cpu_to_le16((__u16)count);
   187    8475822 :         if (EXT4_DESC_SIZE(sb) >= EXT4_MIN_DESC_SIZE_64BIT)
   188          0 :                 bg->bg_itable_unused_hi = cpu_to_le16(count >> 16);
   189    8475822 : }
   190            :
   191            : /*
   192            :  * Wrappers for jbd2_journal_start/end.
   193            :  *
   194            :  * The only special thing we need to do here is to make sure that all
   195            :  * journal_end calls result in the superblock being marked dirty, so
   196            :  * that sync() will call the filesystem's write_super callback if
   197            :  * appropriate.
   198            :  */
   199            : handle_t *ext4_journal_start_sb(struct super_block *sb, int nblocks)
   200 -2099583818 : {
   201            :         journal_t *journal;
   202            :
   203 -2099583818 :         if (sb->s_flags & MS_RDONLY)
   204          0 :                 return ERR_PTR(-EROFS);
   205            :
   206            :         /* Special case here: if the journal has aborted behind our
   207            :          * backs (eg. EIO in the commit thread), then we still need to
   208            :          * take the FS itself readonly cleanly. */
   209 -2099583818 :         journal = EXT4_SB(sb)->s_journal;
   210 -2099583818 :         if (journal) {
   211 -2099583818 :                 if (is_journal_aborted(journal)) {
   212          0 :                         ext4_abort(sb, __func__, "Detected aborted journal");
   213          0 :                         return ERR_PTR(-EROFS);
   214            :                 }
   215 -2099583818 :                 return jbd2_journal_start(journal, nblocks);
   216            :         }
   217            :         /*
   218            :          * We're not journaling, return the appropriate indication.
   219            :          */
   220          0 :         current->journal_info = EXT4_NOJOURNAL_HANDLE;
   221          0 :         return current->journal_info;
   222            : }
   223            :
   224            : /*
   225            :  * The only special thing we need to do here is to make sure that all
   226            :  * jbd2_journal_stop calls result in the superblock being marked dirty, so
   227            :  * that sync() will call the filesystem's write_super callback if
   228            :  * appropriate.
   229            :  */
   230            : int __ext4_journal_stop(const char *where, handle_t *handle)
   231 -2112567660 : {
   232            :         struct super_block *sb;
   233            :         int err;
   234            :         int rc;
   235            :
   236 -2112567660 :         if (!ext4_handle_valid(handle)) {
   237            :                 /*
   238            :                  * Do this here since we don't call jbd2_journal_stop() in
   239            :                  * no-journal mode.
   240            :                  */
   241          0 :                 current->journal_info = NULL;
   242          0 :                 return 0;
   243            :         }
   244 -2112567660 :         sb = handle->h_transaction->t_journal->j_private;
   245 -2112567660 :         err = handle->h_err;
   246 -2112567660 :         rc = jbd2_journal_stop(handle);
   247            :
   248 -2090457778 :         if (!err)
   249 -2087905261 :                 err = rc;
   250 -2090457778 :         if (err)
   251          0 :                 __ext4_std_error(sb, where, err);
   252 -2083723223 :         return err;
   253            : }
   254            :
```

```
   255                : void ext4_journal_abort_handle(const char *caller, const char *err_fn,
   256                :                                struct buffer_head *bh, handle_t *handle, int err)
   257              0 : {
   258                :         char nbuf[16];
   259              0 :         const char *errstr = ext4_decode_error(NULL, err, nbuf);
   260                :
   261              0 :         BUG_ON(!ext4_handle_valid(handle));
   262                :
   263                :         if (bh)
   264                :                 BUFFER_TRACE(bh, "abort");
   265                :
   266              0 :         if (!handle->h_err)
   267              0 :                 handle->h_err = err;
   268                :
   269              0 :         if (is_handle_aborted(handle))
   270              0 :                 return;
   271                :
   272              0 :         printk(KERN_ERR "%s: aborting transaction: %s in %s\n",
   273                :                caller, errstr, err_fn);
   274                :
   275                :         jbd2_journal_abort_handle(handle);
   276                : }
   277                :
   278                : /* Deal with the reporting of failure conditions on a filesystem such as
   279                :  * inconsistencies detected or read IO failures.
   280                :  *
   281                :  * On ext2, we can store the error state of the filesystem in the
   282                :  * superblock.  That is not possible on ext4, because we may have other
   283                :  * write ordering constraints on the superblock which prevent us from
   284                :  * writing it out straight away; and given that the journal is about to
   285                :  * be aborted, we can't rely on the current, or future, transactions to
   286                :  * write out the superblock safely.
   287                :  *
   288                :  * We'll just use the jbd2_journal_abort() error code to record an error in
   289                :  * the journal instead.  On recovery, the journal will compain about
   290                :  * that error until we've noted it down and cleared it.
   291                :  */
   292                :
   293                : static void ext4_handle_error(struct super_block *sb)
   294              0 : {
   295              0 :         struct ext4_super_block *es = EXT4_SB(sb)->s_es;
   296                :
   297              0 :         EXT4_SB(sb)->s_mount_state |= EXT4_ERROR_FS;
   298              0 :         es->s_state |= cpu_to_le16(EXT4_ERROR_FS);
   299                :
   300              0 :         if (sb->s_flags & MS_RDONLY)
   301              0 :                 return;
   302                :
   303              0 :         if (!test_opt(sb, ERRORS_CONT)) {
   304              0 :                 journal_t *journal = EXT4_SB(sb)->s_journal;
   305                :
   306              0 :                 EXT4_SB(sb)->s_mount_flags |= EXT4_MF_FS_ABORTED;
   307              0 :                 if (journal)
   308              0 :                         jbd2_journal_abort(journal, -EIO);
   309                :         }
   310              0 :         if (test_opt(sb, ERRORS_RO)) {
   311              0 :                 ext4_msg(sb, KERN_CRIT, "Remounting filesystem read-only");
   312              0 :                 sb->s_flags |= MS_RDONLY;
   313                :         }
   314              0 :         ext4_commit_super(sb, 1);
   315              0 :         if (test_opt(sb, ERRORS_PANIC))
   316              0 :                 panic("EXT4-fs (device %s): panic forced after error\n",
   317                :                         sb->s_id);
   318                : }
   319                :
   320                : void ext4_error(struct super_block *sb, const char *function,
   321                :                 const char *fmt, ...)
   322              0 : {
   323                :         va_list args;
   324                :
   325              0 :         va_start(args, fmt);
   326              0 :         printk(KERN_CRIT "EXT4-fs error (device %s): %s: ", sb->s_id, function);
   327              0 :         vprintk(fmt, args);
   328              0 :         printk("\n");
   329              0 :         va_end(args);
   330                :
   331              0 :         ext4_handle_error(sb);
   332              0 : }
   333                :
   334                : static const char *ext4_decode_error(struct super_block *sb, int errno,
   335                :                                      char nbuf[16])
   336              0 : {
   337              0 :         char *errstr = NULL;
   338                :
   339              0 :         switch (errno) {
   340                :         case -EIO:
   341              0 :                 errstr = "IO failure";
   342              0 :                 break;
   343                :         case -ENOMEM:
   344              0 :                 errstr = "Out of memory";
```

```
345        0 :                 break;
346          :         case -EROFS:
347        0 :                 if (!sb || EXT4_SB(sb)->s_journal->j_flags & JBD2_ABORT)
348        0 :                         errstr = "Journal has aborted";
349          :                 else
350        0 :                         errstr = "Readonly filesystem";
351          :                 break;
352          :         default:
353          :                 /* If the caller passed in an extra buffer for unknown
354          :                  * errors, textualise them now.  Else we just return
355          :                  * NULL. */
356        0 :                 if (nbuf) {
357          :                         /* Check for truncated error codes... */
358        0 :                         if (snprintf(nbuf, 16, "error %d", -errno) >= 0)
359        0 :                                 errstr = nbuf;
360          :                 }
361          :                 break;
362          :         }
363          :
364        0 :         return errstr;
365          : }
366          :
367          : /* __ext4_std_error decodes expected errors from journaling functions
368          :  * automatically and invokes the appropriate error response.  */
369          :
370          : void __ext4_std_error(struct super_block *sb, const char *function, int errno)
371        0 : {
372          :         char nbuf[16];
373          :         const char *errstr;
374          :
375          :         /* Special case: if the error is EROFS, and we're not already
376          :          * inside a transaction, then there's really no point in logging
377          :          * an error. */
378        0 :         if (errno == -EROFS && journal_current_handle() == NULL &&
379          :             (sb->s_flags & MS_RDONLY))
380        0 :                 return;
381          :
382        0 :         errstr = ext4_decode_error(sb, errno, nbuf);
383        0 :         printk(KERN_CRIT "EXT4-fs error (device %s) in %s: %s\n",
384          :                sb->s_id, function, errstr);
385          :
386        0 :         ext4_handle_error(sb);
387          : }
388          :
389          : /*
390          :  * ext4_abort is a much stronger failure handler than ext4_error.  The
391          :  * abort function may be used to deal with unrecoverable failures such
392          :  * as journal IO errors or ENOMEM at a critical moment in log management.
393          :  *
394          :  * We unconditionally force the filesystem into an ABORT|READONLY state,
395          :  * unless the error response on the fs has been set to panic in which
396          :  * case we take the easy way out and panic immediately.
397          :  */
398          :
399          : void ext4_abort(struct super_block *sb, const char *function,
400          :                 const char *fmt, ...)
401        0 : {
402          :         va_list args;
403          :
404        0 :         va_start(args, fmt);
405        0 :         printk(KERN_CRIT "EXT4-fs error (device %s): %s: ", sb->s_id, function);
406        0 :         vprintk(fmt, args);
407        0 :         printk("\n");
408        0 :         va_end(args);
409          :
410        0 :         if (test_opt(sb, ERRORS_PANIC))
411        0 :                 panic("EXT4-fs panic from previous error\n");
412          :
413        0 :         if (sb->s_flags & MS_RDONLY)
414        0 :                 return;
415          :
416        0 :         ext4_msg(sb, KERN_CRIT, "Remounting filesystem read-only");
417        0 :         EXT4_SB(sb)->s_mount_state |= EXT4_ERROR_FS;
418        0 :         sb->s_flags |= MS_RDONLY;
419        0 :         EXT4_SB(sb)->s_mount_flags |= EXT4_MF_FS_ABORTED;
420        0 :         if (EXT4_SB(sb)->s_journal)
421        0 :                 jbd2_journal_abort(EXT4_SB(sb)->s_journal, -EIO);
422          : }
423          :
424          : void ext4_msg (struct super_block * sb, const char *prefix,
425          :                const char *fmt, ...)
426      365 : {
427          :         va_list args;
428          :
429      365 :         va_start(args, fmt);
430      365 :         printk("%sEXT4-fs (%s): ", prefix, sb->s_id);
431      365 :         vprintk(fmt, args);
432      365 :         printk("\n");
433      365 :         va_end(args);
434      365 : }
```

```
   435                :
   436                : void ext4_warning(struct super_block *sb, const char *function,
   437                :                   const char *fmt, ...)
   438             10 : {
   439                :         va_list args;
   440                :
   441             10 :         va_start(args, fmt);
   442             10 :         printk(KERN_WARNING "EXT4-fs warning (device %s): %s: ",
   443                :                sb->s_id, function);
   444             10 :         vprintk(fmt, args);
   445             10 :         printk("\n");
   446             10 :         va_end(args);
   447             10 : }
   448                :
   449                : void ext4_grp_locked_error(struct super_block *sb, ext4_group_t grp,
   450                :                            const char *function, const char *fmt, ...)
   451                : __releases(bitlock)
   452                : __acquires(bitlock)
   453              0 : {
   454                :         va_list args;
   455              0 :         struct ext4_super_block *es = EXT4_SB(sb)->s_es;
   456                :
   457              0 :         va_start(args, fmt);
   458              0 :         printk(KERN_CRIT "EXT4-fs error (device %s): %s: ", sb->s_id, function);
   459              0 :         vprintk(fmt, args);
   460              0 :         printk("\n");
   461              0 :         va_end(args);
   462                :
   463              0 :         if (test_opt(sb, ERRORS_CONT)) {
   464              0 :                 EXT4_SB(sb)->s_mount_state |= EXT4_ERROR_FS;
   465              0 :                 es->s_state |= cpu_to_le16(EXT4_ERROR_FS);
   466              0 :                 ext4_commit_super(sb, 0);
   467              0 :                 return;
   468                :         }
   469                :         ext4_unlock_group(sb, grp);
   470              0 :         ext4_handle_error(sb);
   471                :         /*
   472                :          * We only get here in the ERRORS_RO case; relocking the group
   473                :          * may be dangerous, but nothing bad will happen since the
   474                :          * filesystem will have already been marked read/only and the
   475                :          * journal has been aborted.  We return 1 as a hint to callers
   476                :          * who might what to use the return value from
   477                :          * ext4_grp_locked_error() to distinguish beween the
   478                :          * ERRORS_CONT and ERRORS_RO case, and perhaps return more
   479                :          * aggressively from the ext4 function in question, with a
   480                :          * more appropriate error code.
   481                :          */
   482                :         ext4_lock_group(sb, grp);
   483                :         return;
   484                : }
   485                :
   486                : void ext4_update_dynamic_rev(struct super_block *sb)
   487             94 : {
   488             94 :         struct ext4_super_block *es = EXT4_SB(sb)->s_es;
   489                :
   490             94 :         if (le32_to_cpu(es->s_rev_level) > EXT4_GOOD_OLD_REV)
   491             94 :                 return;
   492                :
   493              0 :         ext4_warning(sb, __func__,
   494                :                      "updating to rev %d because of new feature flag, "
   495                :                      "running e2fsck is recommended",
   496                :                      EXT4_DYNAMIC_REV);
   497                :
   498              0 :         es->s_first_ino = cpu_to_le32(EXT4_GOOD_OLD_FIRST_INO);
   499              0 :         es->s_inode_size = cpu_to_le16(EXT4_GOOD_OLD_INODE_SIZE);
   500              0 :         es->s_rev_level = cpu_to_le32(EXT4_DYNAMIC_REV);
   501                :         /* leave es->s_feature_*compat flags alone */
   502                :         /* es->s_uuid will be set by e2fsck if empty */
   503                :
   504                :         /*
   505                :          * The rest of the superblock fields should be zero, and if not it
   506                :          * means they are likely already in use, so leave them alone.  We
   507                :          * can leave it up to e2fsck to clean up any inconsistencies there.
   508                :          */
   509                : }
   510                :
   511                : /*
   512                :  * Open the external journal device
   513                :  */
   514                : static struct block_device *ext4_blkdev_get(dev_t dev, struct super_block *sb)
   515                : {
   516                :         struct block_device *bdev;
   517                :         char b[BDEVNAME_SIZE];
   518                :
   519              0 :         bdev = open_by_devnum(dev, FMODE_READ|FMODE_WRITE);
   520              0 :         if (IS_ERR(bdev))
   521                :                 goto fail;
   522              0 :         return bdev;
   523                :
   524              0 : fail:
```

```
    525                0 :          ext4_msg(sb, KERN_ERR, "failed to open journal device %s: %ld",
    526                  :                   __bdevname(dev, b), PTR_ERR(bdev));
    527                0 :          return NULL;
    528                  : }
    529                  :
    530                  : /*
    531                  :  * Release the journal device
    532                  :  */
    533                  : static int ext4_blkdev_put(struct block_device *bdev)
    534                  : {
    535                0 :          bd_release(bdev);
    536                0 :          return blkdev_put(bdev, FMODE_READ|FMODE_WRITE);
    537                  : }
    538                  :
    539                  : static int ext4_blkdev_remove(struct ext4_sb_info *sbi)
    540                0 : {
    541                  :          struct block_device *bdev;
    542                0 :          int ret = -ENODEV;
    543                  :
    544                0 :          bdev = sbi->journal_bdev;
    545                0 :          if (bdev) {
    546                0 :                   ret = ext4_blkdev_put(bdev);
    547                0 :                   sbi->journal_bdev = NULL;
    548                  :          }
    549                0 :          return ret;
    550                  : }
    551                  :
    552                  : static inline struct inode *orphan_list_entry(struct list_head *l)
    553                  : {
    554                0 :          return &list_entry(l, struct ext4_inode_info, i_orphan)->vfs_inode;
    555                  : }
    556                  :
    557                  : static void dump_orphan_list(struct super_block *sb, struct ext4_sb_info *sbi)
    558                  : {
    559                  :          struct list_head *l;
    560                  :
    561                0 :          ext4_msg(sb, KERN_ERR, "sb orphan head is %d",
    562                  :                   le32_to_cpu(sbi->s_es->s_last_orphan));
    563                  :
    564                0 :          printk(KERN_ERR "sb_info orphan list:\n");
    565                0 :          list_for_each(l, &sbi->s_orphan) {
    566                0 :                   struct inode *inode = orphan_list_entry(l);
    567                0 :                   printk(KERN_ERR "  "
    568                  :                          "inode %s:%lu at %p: mode %o, nlink %d, next %d\n",
    569                  :                          inode->i_sb->s_id, inode->i_ino, inode,
    570                  :                          inode->i_mode, inode->i_nlink,
    571                  :                          NEXT_ORPHAN(inode));
    572                  :          }
    573                  : }
    574                  :
    575                  : static void ext4_put_super(struct super_block *sb)
    576               94 : {
    577               94 :          struct ext4_sb_info *sbi = EXT4_SB(sb);
    578               94 :          struct ext4_super_block *es = sbi->s_es;
    579                  :          int i, err;
    580                  :
    581               94 :          lock_super(sb);
    582               94 :          lock_kernel();
    583               94 :          if (sb->s_dirt)
    584               86 :                   ext4_commit_super(sb, 1);
    585                  :
    586               94 :          ext4_release_system_zone(sb);
    587               94 :          ext4_mb_release(sb);
    588               94 :          ext4_ext_release(sb);
    589               94 :          ext4_xattr_put_super(sb);
    590               94 :          if (sbi->s_journal) {
    591               94 :                   err = jbd2_journal_destroy(sbi->s_journal);
    592               94 :                   sbi->s_journal = NULL;
    593               94 :                   if (err < 0)
    594                0 :                            ext4_abort(sb, __func__,
    595                  :                                       "Couldn't clean up the journal");
    596                  :          }
    597               94 :          if (!(sb->s_flags & MS_RDONLY)) {
    598               94 :                   EXT4_CLEAR_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER);
    599               94 :                   es->s_state = cpu_to_le16(sbi->s_mount_state);
    600               94 :                   ext4_commit_super(sb, 1);
    601                  :          }
    602               94 :          if (sbi->s_proc) {
    603               94 :                   remove_proc_entry(sb->s_id, ext4_proc_root);
    604                  :          }
    605               94 :          kobject_del(&sbi->s_kobj);
    606                  :
    607            42802 :          for (i = 0; i < sbi->s_gdb_count; i++)
    608            42708 :                   brelse(sbi->s_group_desc[i]);
    609               94 :          kfree(sbi->s_group_desc);
    610              188 :          if (is_vmalloc_addr(sbi->s_flex_groups))
    611                0 :                   vfree(sbi->s_flex_groups);
    612                  :          else
    613               94 :                   kfree(sbi->s_flex_groups);
    614               94 :          percpu_counter_destroy(&sbi->s_freeblocks_counter);
```

```
   615                94 :          percpu_counter_destroy(&sbi->s_freeinodes_counter);
   616                94 :          percpu_counter_destroy(&sbi->s_dirs_counter);
   617                94 :          percpu_counter_destroy(&sbi->s_dirtyblocks_counter);
   618                94 :          brelse(sbi->s_sbh);
   619                   : #ifdef CONFIG_QUOTA
   620               282 :          for (i = 0; i < MAXQUOTAS; i++)
   621               188 :                  kfree(sbi->s_qf_names[i]);
   622                   : #endif
   623                   :
   624                   :          /* Debugging code just in case the in-memory inode orphan list
   625                   :           * isn't empty.  The on-disk one can be non-empty if we've
   626                   :           * detected an error and taken the fs readonly, but the
   627                   :           * in-memory list had better be clean by this point. */
   628               188 :          if (!list_empty(&sbi->s_orphan))
   629                   :                  dump_orphan_list(sb, sbi);
   630               188 :          J_ASSERT(list_empty(&sbi->s_orphan));
   631                   :
   632                94 :          invalidate_bdev(sb->s_bdev);
   633                94 :          if (sbi->journal_bdev && sbi->journal_bdev != sb->s_bdev) {
   634                   :                  /*
   635                   :                   * Invalidate the journal device's buffers.  We don't want them
   636                   :                   * floating about in memory - the physical journal device may
   637                   :                   * hotswapped, and it breaks the `ro-after' testing code.
   638                   :                   */
   639                 0 :                  sync_blockdev(sbi->journal_bdev);
   640                 0 :                  invalidate_bdev(sbi->journal_bdev);
   641                 0 :                  ext4_blkdev_remove(sbi);
   642                   :          }
   643                94 :          sb->s_fs_info = NULL;
   644                   :          /*
   645                   :           * Now that we are completely done shutting down the
   646                   :           * superblock, we need to actually destroy the kobject.
   647                   :           */
   648                94 :          unlock_kernel();
   649                94 :          unlock_super(sb);
   650                94 :          kobject_put(&sbi->s_kobj);
   651                94 :          wait_for_completion(&sbi->s_kobj_unregister);
   652                94 :          kfree(sbi->s_blockgroup_lock);
   653                94 :          kfree(sbi);
   654                94 : }
   655                   :
   656                   : static struct kmem_cache *ext4_inode_cachep;
   657                   :
   658                   : /*
   659                   :  * Called inside transaction, so use GFP_NOFS
   660                   :  */
   661                   : static struct inode *ext4_alloc_inode(struct super_block *sb)
   662          13086279 : {
   663                   :          struct ext4_inode_info *ei;
   664                   :
   665          13086279 :          ei = kmem_cache_alloc(ext4_inode_cachep, GFP_NOFS);
   666          13086278 :          if (!ei)
   667                 0 :                  return NULL;
   668                   :
   669          13086278 :          ei->vfs_inode.i_version = 1;
   670          13086278 :          ei->vfs_inode.i_data.writeback_index = 0;
   671          13086278 :          memset(&ei->i_cached_extent, 0, sizeof(struct ext4_ext_cache));
   672          13086278 :          INIT_LIST_HEAD(&ei->i_prealloc_list);
   673          13086278 :          spin_lock_init(&ei->i_prealloc_lock);
   674                   :          /*
   675                   :           * Note:  We can be called before EXT4_SB(sb)->s_journal is set,
   676                   :           * therefore it can be null here.  Don't check it, just initialize
   677                   :           * jinode.
   678                   :           */
   679          13086278 :          jbd2_journal_init_jbd_inode(&ei->jinode, &ei->vfs_inode);
   680          13086278 :          ei->i_reserved_data_blocks = 0;
   681          13086278 :          ei->i_reserved_meta_blocks = 0;
   682          13086278 :          ei->i_allocated_meta_blocks = 0;
   683          13086278 :          ei->i_delalloc_reserved_flag = 0;
   684          13086278 :          spin_lock_init(&(ei->i_block_reservation_lock));
   685                   :
   686          13086278 :          return &ei->vfs_inode;
   687                   : }
   688                   :
   689                   : static void ext4_destroy_inode(struct inode *inode)
   690          13086268 : {
   691          26172536 :          if (!list_empty(&(EXT4_I(inode)->i_orphan))) {
   692                 0 :                  ext4_msg(inode->i_sb, KERN_ERR,
   693                   :                           "Inode %lu (%p): orphan list check failed!",
   694                   :                           inode->i_ino, EXT4_I(inode));
   695                 0 :                  print_hex_dump(KERN_INFO, "", DUMP_PREFIX_ADDRESS, 16, 4,
   696                   :                                 EXT4_I(inode), sizeof(struct ext4_inode_info),
   697                   :                                 true);
   698                 0 :                  dump_stack();
   699                   :          }
   700          13086268 :          kmem_cache_free(ext4_inode_cachep, EXT4_I(inode));
   701          13086271 : }
   702                   :
   703                   : static void init_once(void *foo)
   704           5935608 : {
```

```
   705  5935608 :          struct ext4_inode_info *ei = (struct ext4_inode_info *) foo;
   706          :
   707  5935608 :          INIT_LIST_HEAD(&ei->i_orphan);
   708          : #ifdef CONFIG_EXT4_FS_XATTR
   709  5935608 :          init_rwsem(&ei->xattr_sem);
   710          : #endif
   711  5935608 :          init_rwsem(&ei->i_data_sem);
   712  5935608 :          inode_init_once(&ei->vfs_inode);
   713  5935608 : }
   714          :
   715          : static int init_inodecache(void)
   716          : {
   717        0 :          ext4_inode_cachep = kmem_cache_create("ext4_inode_cache",
   718          :                                                sizeof(struct ext4_inode_info),
   719          :                                                0, (SLAB_RECLAIM_ACCOUNT|
   720          :                                                   SLAB_MEM_SPREAD),
   721          :                                                init_once);
   722        0 :          if (ext4_inode_cachep == NULL)
   723        0 :                  return -ENOMEM;
   724        0 :          return 0;
   725          : }
   726          :
   727          : static void destroy_inodecache(void)
   728          : {
   729        0 :          kmem_cache_destroy(ext4_inode_cachep);
   730          : }
   731          :
   732          : static void ext4_clear_inode(struct inode *inode)
   733 13086272 : {
   734 13086272 :          ext4_discard_preallocations(inode);
   735 26172550 :          if (EXT4_JOURNAL(inode))
   736 26172544 :                  jbd2_journal_release_jbd_inode(EXT4_SB(inode->i_sb)->s_journal,
   737          :                                                 &EXT4_I(inode)->jinode);
   738 13086277 : }
   739          :
   740          : static inline void ext4_show_quota_options(struct seq_file *seq,
   741          :                                            struct super_block *sb)
   742          : {
   743          : #if defined(CONFIG_QUOTA)
   744   241088 :          struct ext4_sb_info *sbi = EXT4_SB(sb);
   745          :
   746   241088 :          if (sbi->s_jquota_fmt)
   747        0 :                  seq_printf(seq, ",jqfmt=%s",
   748          :                  (sbi->s_jquota_fmt == QFMT_VFS_OLD) ? "vfsold" : "vfsv0");
   749          :
   750   241088 :          if (sbi->s_qf_names[USRQUOTA])
   751        0 :                  seq_printf(seq, ",usrjquota=%s", sbi->s_qf_names[USRQUOTA]);
   752          :
   753   241088 :          if (sbi->s_qf_names[GRPQUOTA])
   754        0 :                  seq_printf(seq, ",grpjquota=%s", sbi->s_qf_names[GRPQUOTA]);
   755          :
   756   241088 :          if (sbi->s_mount_opt & EXT4_MOUNT_USRQUOTA)
   757        0 :                  seq_puts(seq, ",usrquota");
   758          :
   759   241088 :          if (sbi->s_mount_opt & EXT4_MOUNT_GRPQUOTA)
   760        0 :                  seq_puts(seq, ",grpquota");
   761          : #endif
   762          : }
   763          :
   764          : /*
   765          :  * Show an option if
   766          :  *  - it's set to a non-default value OR
   767          :  *  - if the per-sb default is different from the global default
   768          :  */
   769          : static int ext4_show_options(struct seq_file *seq, struct vfsmount *vfs)
   770   241089 : {
   771          :          int def_errors;
   772          :          unsigned long def_mount_opts;
   773   241089 :          struct super_block *sb = vfs->mnt_sb;
   774   241089 :          struct ext4_sb_info *sbi = EXT4_SB(sb);
   775   241089 :          struct ext4_super_block *es = sbi->s_es;
   776          :
   777   241089 :          def_mount_opts = le32_to_cpu(es->s_default_mount_opts);
   778   241089 :          def_errors     = le16_to_cpu(es->s_errors);
   779          :
   780   241089 :          if (sbi->s_sb_block != 1)
   781        0 :                  seq_printf(seq, ",sb=%llu", sbi->s_sb_block);
   782   241089 :          if (test_opt(sb, MINIX_DF))
   783        0 :                  seq_puts(seq, ",minixdf");
   784   241089 :          if (test_opt(sb, GRPID) && !(def_mount_opts & EXT4_DEFM_BSDGROUPS))
   785        0 :                  seq_puts(seq, ",grpid");
   786   241089 :          if (!test_opt(sb, GRPID) && (def_mount_opts & EXT4_DEFM_BSDGROUPS))
   787        0 :                  seq_puts(seq, ",nogrpid");
   788   241089 :          if (sbi->s_resuid != EXT4_DEF_RESUID ||
   789          :              le16_to_cpu(es->s_def_resuid) != EXT4_DEF_RESUID) {
   790        0 :                  seq_printf(seq, ",resuid=%u", sbi->s_resuid);
   791          :          }
   792   241089 :          if (sbi->s_resgid != EXT4_DEF_RESGID ||
   793          :              le16_to_cpu(es->s_def_resgid) != EXT4_DEF_RESGID) {
   794        2 :                  seq_printf(seq, ",resgid=%u", sbi->s_resgid);
```

```
    795                 :         }
    796          241087 :         if (test_opt(sb, ERRORS_RO)) {
    797               0 :                 if (def_errors == EXT4_ERRORS_PANIC ||
    798                 :                     def_errors == EXT4_ERRORS_CONTINUE) {
    799               0 :                         seq_puts(seq, ",errors=remount-ro");
    800                 :                 }
    801                 :         }
    802          241087 :         if (test_opt(sb, ERRORS_CONT) && def_errors != EXT4_ERRORS_CONTINUE)
    803               0 :                 seq_puts(seq, ",errors=continue");
    804          241087 :         if (test_opt(sb, ERRORS_PANIC) && def_errors != EXT4_ERRORS_PANIC)
    805               0 :                 seq_puts(seq, ",errors=panic");
    806          241087 :         if (test_opt(sb, NO_UID32) && !(def_mount_opts & EXT4_DEFM_UID16))
    807               0 :                 seq_puts(seq, ",nouid32");
    808          241087 :         if (test_opt(sb, DEBUG) && !(def_mount_opts & EXT4_DEFM_DEBUG))
    809               0 :                 seq_puts(seq, ",debug");
    810          241087 :         if (test_opt(sb, OLDALLOC))
    811               0 :                 seq_puts(seq, ",oldalloc");
    812                 : #ifdef CONFIG_EXT4_FS_XATTR
    813          241087 :         if (test_opt(sb, XATTR_USER) &&
    814                 :                 !(def_mount_opts & EXT4_DEFM_XATTR_USER))
    815               0 :                 seq_puts(seq, ",user_xattr");
    816          241087 :         if (!test_opt(sb, XATTR_USER) &&
    817                 :             (def_mount_opts & EXT4_DEFM_XATTR_USER)) {
    818               0 :                 seq_puts(seq, ",nouser_xattr");
    819                 :         }
    820                 : #endif
    821                 : #ifdef CONFIG_EXT4_FS_POSIX_ACL
    822          241087 :         if (test_opt(sb, POSIX_ACL) && !(def_mount_opts & EXT4_DEFM_ACL))
    823               0 :                 seq_puts(seq, ",acl");
    824          241087 :         if (!test_opt(sb, POSIX_ACL) && (def_mount_opts & EXT4_DEFM_ACL))
    825               0 :                 seq_puts(seq, ",noacl");
    826                 : #endif
    827          241087 :         if (sbi->s_commit_interval != JBD2_DEFAULT_MAX_COMMIT_AGE*HZ) {
    828             963 :                 seq_printf(seq, ",commit=%u",
    829                 :                            (unsigned) (sbi->s_commit_interval / HZ));
    830                 :         }
    831          241087 :         if (sbi->s_min_batch_time != EXT4_DEF_MIN_BATCH_TIME) {
    832               0 :                 seq_printf(seq, ",min_batch_time=%u",
    833                 :                            (unsigned) sbi->s_min_batch_time);
    834                 :         }
    835          241087 :         if (sbi->s_max_batch_time != EXT4_DEF_MAX_BATCH_TIME) {
    836               0 :                 seq_printf(seq, ",max_batch_time=%u",
    837                 :                            (unsigned) sbi->s_min_batch_time);
    838                 :         }
    839                 :
    840                 :         /*
    841                 :          * We're changing the default of barrier mount option, so
    842                 :          * let's always display its mount state so it's clear what its
    843                 :          * status is.
    844                 :          */
    845          241087 :         seq_puts(seq, ",barrier=");
    846          241088 :         seq_puts(seq, test_opt(sb, BARRIER) ? "1" : "0");
    847          241089 :         if (test_opt(sb, JOURNAL_ASYNC_COMMIT))
    848             322 :                 seq_puts(seq, ",journal_async_commit");
    849          241088 :         if (test_opt(sb, NOBH))
    850               0 :                 seq_puts(seq, ",nobh");
    851          241088 :         if (test_opt(sb, I_VERSION))
    852             106 :                 seq_puts(seq, ",i_version");
    853          241088 :         if (!test_opt(sb, DELALLOC))
    854           43355 :                 seq_puts(seq, ",nodelalloc");
    855                 :
    856                 :
    857          241088 :         if (sbi->s_stripe)
    858               0 :                 seq_printf(seq, ",stripe=%lu", sbi->s_stripe);
    859                 :         /*
    860                 :          * journal mode get enabled in different ways
    861                 :          * So just print the value even if we didn't specify it
    862                 :          */
    863          241088 :         if (test_opt(sb, DATA_FLAGS) == EXT4_MOUNT_JOURNAL_DATA)
    864             318 :                 seq_puts(seq, ",data=journal");
    865          240770 :         else if (test_opt(sb, DATA_FLAGS) == EXT4_MOUNT_ORDERED_DATA)
    866          240444 :                 seq_puts(seq, ",data=ordered");
    867             326 :         else if (test_opt(sb, DATA_FLAGS) == EXT4_MOUNT_WRITEBACK_DATA)
    868             326 :                 seq_puts(seq, ",data=writeback");
    869                 :
    870          241088 :         if (sbi->s_inode_readahead_blks != EXT4_DEF_INODE_READAHEAD_BLKS)
    871               0 :                 seq_printf(seq, ",inode_readahead_blks=%u",
    872                 :                            sbi->s_inode_readahead_blks);
    873                 :
    874          241088 :         if (test_opt(sb, DATA_ERR_ABORT))
    875               0 :                 seq_puts(seq, ",data_err=abort");
    876                 :
    877          241088 :         if (test_opt(sb, NO_AUTO_DA_ALLOC))
    878            2440 :                 seq_puts(seq, ",noauto_da_alloc");
    879                 :
    880                 :         ext4_show_quota_options(seq, sb);
    881                 :
    882          241088 :         return 0;
    883                 : }
    884                 :
```

```
    885                 : static struct inode *ext4_nfs_get_inode(struct super_block *sb,
    886                 :                                         u64 ino, u32 generation)
    887               0 : {
    888                 :         struct inode *inode;
    889                 :
    890               0 :         if (ino < EXT4_FIRST_INO(sb) && ino != EXT4_ROOT_INO)
    891               0 :                 return ERR_PTR(-ESTALE);
    892               0 :         if (ino > le32_to_cpu(EXT4_SB(sb)->s_es->s_inodes_count))
    893               0 :                 return ERR_PTR(-ESTALE);
    894                 :
    895                 :         /* iget isn't really right if the inode is currently unallocated!!
    896                 :          *
    897                 :          * ext4_read_inode will return a bad_inode if the inode had been
    898                 :          * deleted, so we should be safe.
    899                 :          *
    900                 :          * Currently we don't know the generation for parent directory, so
    901                 :          * a generation of 0 means "accept any"
    902                 :          */
    903               0 :         inode = ext4_iget(sb, ino);
    904               0 :         if (IS_ERR(inode))
    905               0 :                 return ERR_CAST(inode);
    906               0 :         if (generation && inode->i_generation != generation) {
    907               0 :                 iput(inode);
    908               0 :                 return ERR_PTR(-ESTALE);
    909                 :         }
    910                 :
    911               0 :         return inode;
    912                 : }
    913                 :
    914                 : static struct dentry *ext4_fh_to_dentry(struct super_block *sb, struct fid *fid,
    915                 :                                         int fh_len, int fh_type)
    916               0 : {
    917               0 :         return generic_fh_to_dentry(sb, fid, fh_len, fh_type,
    918                 :                                     ext4_nfs_get_inode);
    919                 : }
    920                 :
    921                 : static struct dentry *ext4_fh_to_parent(struct super_block *sb, struct fid *fid,
    922                 :                                         int fh_len, int fh_type)
    923               0 : {
    924               0 :         return generic_fh_to_parent(sb, fid, fh_len, fh_type,
    925                 :                                     ext4_nfs_get_inode);
    926                 : }
    927                 :
    928                 : /*
    929                 :  * Try to release metadata pages (indirect blocks, directories) which are
    930                 :  * mapped via the block device.  Since these pages could have journal heads
    931                 :  * which would prevent try_to_free_buffers() from freeing them, we must use
    932                 :  * jbd2 layer's try_to_free_buffers() function to release them.
    933                 :  */
    934                 : static int bdev_try_to_free_page(struct super_block *sb, struct page *page,
    935                 :                                  gfp_t wait)
    936          797154 : {
    937          797154 :         journal_t *journal = EXT4_SB(sb)->s_journal;
    938                 :
    939          797154 :         WARN_ON(PageChecked(page));
    940          797200 :         if (!page_has_buffers(page))
    941               0 :                 return 0;
    942          797200 :         if (journal)
    943          797200 :                 return jbd2_journal_try_to_free_buffers(journal, page,
    944                 :                                                         wait & ~__GFP_WAIT);
    945               0 :         return try_to_free_buffers(page);
    946                 : }
    947                 :
    948                 : #ifdef CONFIG_QUOTA
    949                 : #define QTYPE2NAME(t) ((t) == USRQUOTA ? "user" : "group")
    950                 : #define QTYPE2MOPT(on, t) ((t) == USRQUOTA?((on)##USRJQUOTA):((on)##GRPJQUOTA))
    951                 :
    952                 : static int ext4_write_dquot(struct dquot *dquot);
    953                 : static int ext4_acquire_dquot(struct dquot *dquot);
    954                 : static int ext4_release_dquot(struct dquot *dquot);
    955                 : static int ext4_mark_dquot_dirty(struct dquot *dquot);
    956                 : static int ext4_write_info(struct super_block *sb, int type);
    957                 : static int ext4_quota_on(struct super_block *sb, int type, int format_id,
    958                 :                          char *path, int remount);
    959                 : static int ext4_quota_on_mount(struct super_block *sb, int type);
    960                 : static ssize_t ext4_quota_read(struct super_block *sb, int type, char *data,
    961                 :                                size_t len, loff_t off);
    962                 : static ssize_t ext4_quota_write(struct super_block *sb, int type,
    963                 :                                 const char *data, size_t len, loff_t off);
    964                 :
    965                 : static struct dquot_operations ext4_quota_operations = {
    966                 :         .initialize     = dquot_initialize,
    967                 :         .drop           = dquot_drop,
    968                 :         .alloc_space    = dquot_alloc_space,
    969                 :         .reserve_space  = dquot_reserve_space,
    970                 :         .claim_space    = dquot_claim_space,
    971                 :         .release_rsv    = dquot_release_reserved_space,
    972                 :         .get_reserved_space = ext4_get_reserved_space,
    973                 :         .alloc_inode    = dquot_alloc_inode,
    974                 :         .free_space     = dquot_free_space,
```

```
975                 :         .free_inode     = dquot_free_inode,
976                 :         .transfer       = dquot_transfer,
977                 :         .write_dquot    = ext4_write_dquot,
978                 :         .acquire_dquot  = ext4_acquire_dquot,
979                 :         .release_dquot  = ext4_release_dquot,
980                 :         .mark_dirty     = ext4_mark_dquot_dirty,
981                 :         .write_info     = ext4_write_info,
982                 :         .alloc_dquot    = dquot_alloc,
983                 :         .destroy_dquot  = dquot_destroy,
984                 : };
985                 :
986                 : static struct quotactl_ops ext4_qctl_operations = {
987                 :         .quota_on       = ext4_quota_on,
988                 :         .quota_off      = vfs_quota_off,
989                 :         .quota_sync     = vfs_quota_sync,
990                 :         .get_info       = vfs_get_dqinfo,
991                 :         .set_info       = vfs_set_dqinfo,
992                 :         .get_dqblk      = vfs_get_dqblk,
993                 :         .set_dqblk      = vfs_set_dqblk
994                 : };
995                 : #endif
996                 :
997                 : static const struct super_operations ext4_sops = {
998                 :         .alloc_inode    = ext4_alloc_inode,
999                 :         .destroy_inode  = ext4_destroy_inode,
1000                :         .write_inode    = ext4_write_inode,
1001                :         .dirty_inode    = ext4_dirty_inode,
1002                :         .delete_inode   = ext4_delete_inode,
1003                :         .put_super      = ext4_put_super,
1004                :         .sync_fs        = ext4_sync_fs,
1005                :         .freeze_fs      = ext4_freeze,
1006                :         .unfreeze_fs    = ext4_unfreeze,
1007                :         .statfs         = ext4_statfs,
1008                :         .remount_fs     = ext4_remount,
1009                :         .clear_inode    = ext4_clear_inode,
1010                :         .show_options   = ext4_show_options,
1011                : #ifdef CONFIG_QUOTA
1012                :         .quota_read     = ext4_quota_read,
1013                :         .quota_write    = ext4_quota_write,
1014                : #endif
1015                :         .bdev_try_to_free_page = bdev_try_to_free_page,
1016                : };
1017                :
1018                : static const struct super_operations ext4_nojournal_sops = {
1019                :         .alloc_inode    = ext4_alloc_inode,
1020                :         .destroy_inode  = ext4_destroy_inode,
1021                :         .write_inode    = ext4_write_inode,
1022                :         .dirty_inode    = ext4_dirty_inode,
1023                :         .delete_inode   = ext4_delete_inode,
1024                :         .write_super    = ext4_write_super,
1025                :         .put_super      = ext4_put_super,
1026                :         .statfs         = ext4_statfs,
1027                :         .remount_fs     = ext4_remount,
1028                :         .clear_inode    = ext4_clear_inode,
1029                :         .show_options   = ext4_show_options,
1030                : #ifdef CONFIG_QUOTA
1031                :         .quota_read     = ext4_quota_read,
1032                :         .quota_write    = ext4_quota_write,
1033                : #endif
1034                :         .bdev_try_to_free_page = bdev_try_to_free_page,
1035                : };
1036                :
1037                : static const struct export_operations ext4_export_ops = {
1038                :         .fh_to_dentry = ext4_fh_to_dentry,
1039                :         .fh_to_parent = ext4_fh_to_parent,
1040                :         .get_parent = ext4_get_parent,
1041                : };
1042                :
1043                : enum {
1044                :         Opt_bsd_df, Opt_minix_df, Opt_grpid, Opt_nogrpid,
1045                :         Opt_resgid, Opt_resuid, Opt_sb, Opt_err_cont, Opt_err_panic, Opt_err_ro,
1046                :         Opt_nouid32, Opt_debug, Opt_oldalloc, Opt_orlov,
1047                :         Opt_user_xattr, Opt_nouser_xattr, Opt_acl, Opt_noacl,
1048                :         Opt_auto_da_alloc, Opt_noauto_da_alloc, Opt_noload, Opt_nobh, Opt_bh,
1049                :         Opt_commit, Opt_min_batch_time, Opt_max_batch_time,
1050                :         Opt_journal_update, Opt_journal_dev,
1051                :         Opt_journal_checksum, Opt_journal_async_commit,
1052                :         Opt_abort, Opt_data_journal, Opt_data_ordered, Opt_data_writeback,
1053                :         Opt_data_err_abort, Opt_data_err_ignore, Opt_mb_history_length,
1054                :         Opt_usrjquota, Opt_grpjquota, Opt_offusrjquota, Opt_offgrpjquota,
1055                :         Opt_jqfmt_vfsold, Opt_jqfmt_vfsv0, Opt_quota, Opt_noquota,
1056                :         Opt_ignore, Opt_barrier, Opt_nobarrier, Opt_err, Opt_resize,
1057                :         Opt_usrquota, Opt_grpquota, Opt_i_version,
1058                :         Opt_stripe, Opt_delalloc, Opt_nodelalloc,
1059                :         Opt_block_validity, Opt_noblock_validity,
1060                :         Opt_inode_readahead_blks, Opt_journal_ioprio
1061                : };
1062                :
1063                : static const match_table_t tokens = {
1064                :         {Opt_bsd_df, "bsddf"},
```

```
1065                :          {Opt_minix_df, "minixdf"},
1066                :          {Opt_grpid, "grpid"},
1067                :          {Opt_grpid, "bsdgroups"},
1068                :          {Opt_nogrpid, "nogrpid"},
1069                :          {Opt_nogrpid, "sysvgroups"},
1070                :          {Opt_resgid, "resgid=%u"},
1071                :          {Opt_resuid, "resuid=%u"},
1072                :          {Opt_sb, "sb=%u"},
1073                :          {Opt_err_cont, "errors=continue"},
1074                :          {Opt_err_panic, "errors=panic"},
1075                :          {Opt_err_ro, "errors=remount-ro"},
1076                :          {Opt_nouid32, "nouid32"},
1077                :          {Opt_debug, "debug"},
1078                :          {Opt_oldalloc, "oldalloc"},
1079                :          {Opt_orlov, "orlov"},
1080                :          {Opt_user_xattr, "user_xattr"},
1081                :          {Opt_nouser_xattr, "nouser_xattr"},
1082                :          {Opt_acl, "acl"},
1083                :          {Opt_noacl, "noacl"},
1084                :          {Opt_noload, "noload"},
1085                :          {Opt_nobh, "nobh"},
1086                :          {Opt_bh, "bh"},
1087                :          {Opt_commit, "commit=%u"},
1088                :          {Opt_min_batch_time, "min_batch_time=%u"},
1089                :          {Opt_max_batch_time, "max_batch_time=%u"},
1090                :          {Opt_journal_update, "journal=update"},
1091                :          {Opt_journal_dev, "journal_dev=%u"},
1092                :          {Opt_journal_checksum, "journal_checksum"},
1093                :          {Opt_journal_async_commit, "journal_async_commit"},
1094                :          {Opt_abort, "abort"},
1095                :          {Opt_data_journal, "data=journal"},
1096                :          {Opt_data_ordered, "data=ordered"},
1097                :          {Opt_data_writeback, "data=writeback"},
1098                :          {Opt_data_err_abort, "data_err=abort"},
1099                :          {Opt_data_err_ignore, "data_err=ignore"},
1100                :          {Opt_mb_history_length, "mb_history_length=%u"},
1101                :          {Opt_offusrjquota, "usrjquota="},
1102                :          {Opt_usrjquota, "usrjquota=%s"},
1103                :          {Opt_offgrpjquota, "grpjquota="},
1104                :          {Opt_grpjquota, "grpjquota=%s"},
1105                :          {Opt_jqfmt_vfsold, "jqfmt=vfsold"},
1106                :          {Opt_jqfmt_vfsv0, "jqfmt=vfsv0"},
1107                :          {Opt_grpquota, "grpquota"},
1108                :          {Opt_noquota, "noquota"},
1109                :          {Opt_quota, "quota"},
1110                :          {Opt_usrquota, "usrquota"},
1111                :          {Opt_barrier, "barrier=%u"},
1112                :          {Opt_barrier, "barrier"},
1113                :          {Opt_nobarrier, "nobarrier"},
1114                :          {Opt_i_version, "i_version"},
1115                :          {Opt_stripe, "stripe=%u"},
1116                :          {Opt_resize, "resize"},
1117                :          {Opt_delalloc, "delalloc"},
1118                :          {Opt_nodelalloc, "nodelalloc"},
1119                :          {Opt_block_validity, "block_validity"},
1120                :          {Opt_noblock_validity, "noblock_validity"},
1121                :          {Opt_inode_readahead_blks, "inode_readahead_blks=%u"},
1122                :          {Opt_journal_ioprio, "journal_ioprio=%u"},
1123                :          {Opt_auto_da_alloc, "auto_da_alloc=%u"},
1124                :          {Opt_auto_da_alloc, "auto_da_alloc"},
1125                :          {Opt_noauto_da_alloc, "noauto_da_alloc"},
1126                :          {Opt_err, NULL},
1127                : };
1128                :
1129                : static ext4_fsblk_t get_sb_block(void **data)
1130                : {
1131                :          ext4_fsblk_t    sb_block;
1132             94 :          char            *options = (char *) *data;
1133                :
1134             94 :          if (!options || strncmp(options, "sb=", 3) != 0)
1135             94 :                  return 1;       /* Default location */
1136                :
1137              0 :          options += 3;
1138                :          /* TODO: use simple_strtoll with >32bit ext4 */
1139              0 :          sb_block = simple_strtoul(options, &options, 0);
1140              0 :          if (*options && *options != ',') {
1141              0 :                  printk(KERN_ERR "EXT4-fs: Invalid sb specification: %s\n",
1142                :                         (char *) *data);
1143              0 :                  return 1;
1144                :          }
1145              0 :          if (*options == ',')
1146              0 :                  options++;
1147              0 :          *data = (void *) options;
1148                :
1149              0 :          return sb_block;
1150                : }
1151                :
1152                : #define DEFAULT_JOURNAL_IOPRIO (IOPRIO_PRIO_VALUE(IOPRIO_CLASS_BE, 3))
1153                :
1154                : static int parse_options(char *options, struct super_block *sb,
```

```
1155                :                           unsigned long *journal_devnum,
1156                :                           unsigned int *journal_ioprio,
1157                :                           ext4_fsblk_t *n_blocks_count, int is_remount)
1158               94 : {
1159               94 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
1160                :         char *p;
1161                :         substring_t args[MAX_OPT_ARGS];
1162               94 :         int data_opt = 0;
1163                :         int option;
1164                : #ifdef CONFIG_QUOTA
1165                :         int qtype, qfmt;
1166                :         char *qname;
1167                : #endif
1168                :
1169               94 :         if (!options)
1170               24 :                 return 1;
1171                :
1172              310 :         while ((p = strsep(&options, ",")) != NULL) {
1173                :                 int token;
1174              240 :                 if (!*p)
1175               36 :                         continue;
1176                :
1177              204 :                 token = match_token(p, tokens, args);
1178              204 :                 switch (token) {
1179                :                 case Opt_bsd_df:
1180                0 :                         clear_opt(sbi->s_mount_opt, MINIX_DF);
1181                0 :                         break;
1182                :                 case Opt_minix_df:
1183                0 :                         set_opt(sbi->s_mount_opt, MINIX_DF);
1184                0 :                         break;
1185                :                 case Opt_grpid:
1186                0 :                         set_opt(sbi->s_mount_opt, GRPID);
1187                0 :                         break;
1188                :                 case Opt_nogrpid:
1189                0 :                         clear_opt(sbi->s_mount_opt, GRPID);
1190                0 :                         break;
1191                :                 case Opt_resuid:
1192                0 :                         if (match_int(&args[0], &option))
1193                0 :                                 return 0;
1194                0 :                         sbi->s_resuid = option;
1195                0 :                         break;
1196                :                 case Opt_resgid:
1197                0 :                         if (match_int(&args[0], &option))
1198                0 :                                 return 0;
1199                0 :                         sbi->s_resgid = option;
1200                0 :                         break;
1201                :                 case Opt_sb:
1202                :                         /* handled by get_sb_block() instead of here */
1203                :                         /* *sb_block = match_int(&args[0]); */
1204                :                         break;
1205                :                 case Opt_err_panic:
1206                0 :                         clear_opt(sbi->s_mount_opt, ERRORS_CONT);
1207                0 :                         clear_opt(sbi->s_mount_opt, ERRORS_RO);
1208                0 :                         set_opt(sbi->s_mount_opt, ERRORS_PANIC);
1209                0 :                         break;
1210                :                 case Opt_err_ro:
1211                0 :                         clear_opt(sbi->s_mount_opt, ERRORS_CONT);
1212                0 :                         clear_opt(sbi->s_mount_opt, ERRORS_PANIC);
1213                0 :                         set_opt(sbi->s_mount_opt, ERRORS_RO);
1214                0 :                         break;
1215                :                 case Opt_err_cont:
1216                0 :                         clear_opt(sbi->s_mount_opt, ERRORS_RO);
1217                0 :                         clear_opt(sbi->s_mount_opt, ERRORS_PANIC);
1218                0 :                         set_opt(sbi->s_mount_opt, ERRORS_CONT);
1219                0 :                         break;
1220                :                 case Opt_nouid32:
1221                0 :                         set_opt(sbi->s_mount_opt, NO_UID32);
1222                0 :                         break;
1223                :                 case Opt_debug:
1224                0 :                         set_opt(sbi->s_mount_opt, DEBUG);
1225                0 :                         break;
1226                :                 case Opt_oldalloc:
1227                0 :                         set_opt(sbi->s_mount_opt, OLDALLOC);
1228                0 :                         break;
1229                :                 case Opt_orlov:
1230                9 :                         clear_opt(sbi->s_mount_opt, OLDALLOC);
1231                9 :                         break;
1232                : #ifdef CONFIG_EXT4_FS_XATTR
1233                :                 case Opt_user_xattr:
1234                0 :                         set_opt(sbi->s_mount_opt, XATTR_USER);
1235                0 :                         break;
1236                :                 case Opt_nouser_xattr:
1237                0 :                         clear_opt(sbi->s_mount_opt, XATTR_USER);
1238                0 :                         break;
1239                : #else
1240                :                 case Opt_user_xattr:
1241                :                 case Opt_nouser_xattr:
1242                :                         ext4_msg(sb, KERN_ERR, "(no)user_xattr options not supported");
1243                :                         break;
1244                : #endif
```

```
1245                :  #ifdef CONFIG_EXT4_FS_POSIX_ACL
1246                :                  case Opt_acl:
1247              0 :                          set_opt(sbi->s_mount_opt, POSIX_ACL);
1248              0 :                          break;
1249                :                  case Opt_noacl:
1250              0 :                          clear_opt(sbi->s_mount_opt, POSIX_ACL);
1251              0 :                          break;
1252                :  #else
1253                :                  case Opt_acl:
1254                :                  case Opt_noacl:
1255                :                          ext4_msg(sb, KERN_ERR, "(no)acl options not supported");
1256                :                          break;
1257                :  #endif
1258                :                  case Opt_journal_update:
1259                :                          /* @@@ FIXME */
1260                :                          /* Eventually we will want to be able to create
1261                :                             a journal file here.  For now, only allow the
1262                :                             user to specify an existing inode to be the
1263                :                             journal file. */
1264              0 :                          if (is_remount) {
1265              0 :                                  ext4_msg(sb, KERN_ERR,
1266                :                                           "Cannot specify journal on remount");
1267              0 :                                  return 0;
1268                :                          }
1269              0 :                          set_opt(sbi->s_mount_opt, UPDATE_JOURNAL);
1270              0 :                          break;
1271                :                  case Opt_journal_dev:
1272              0 :                          if (is_remount) {
1273              0 :                                  ext4_msg(sb, KERN_ERR,
1274                :                                           "Cannot specify journal on remount");
1275              0 :                                  return 0;
1276                :                          }
1277              0 :                          if (match_int(&args[0], &option))
1278              0 :                                  return 0;
1279              0 :                          *journal_devnum = option;
1280              0 :                          break;
1281                :                  case Opt_journal_checksum:
1282             24 :                          set_opt(sbi->s_mount_opt, JOURNAL_CHECKSUM);
1283             24 :                          break;
1284                :                  case Opt_journal_async_commit:
1285             12 :                          set_opt(sbi->s_mount_opt, JOURNAL_ASYNC_COMMIT);
1286             12 :                          set_opt(sbi->s_mount_opt, JOURNAL_CHECKSUM);
1287             12 :                          break;
1288                :                  case Opt_noload:
1289              0 :                          set_opt(sbi->s_mount_opt, NOLOAD);
1290              0 :                          break;
1291                :                  case Opt_commit:
1292             36 :                          if (match_int(&args[0], &option))
1293              0 :                                  return 0;
1294             36 :                          if (option < 0)
1295              0 :                                  return 0;
1296             36 :                          if (option == 0)
1297              0 :                                  option = JBD2_DEFAULT_MAX_COMMIT_AGE;
1298             36 :                          sbi->s_commit_interval = HZ * option;
1299             36 :                          break;
1300                :                  case Opt_max_batch_time:
1301              0 :                          if (match_int(&args[0], &option))
1302              0 :                                  return 0;
1303              0 :                          if (option < 0)
1304              0 :                                  return 0;
1305              0 :                          if (option == 0)
1306              0 :                                  option = EXT4_DEF_MAX_BATCH_TIME;
1307              0 :                          sbi->s_max_batch_time = option;
1308              0 :                          break;
1309                :                  case Opt_min_batch_time:
1310              0 :                          if (match_int(&args[0], &option))
1311              0 :                                  return 0;
1312              0 :                          if (option < 0)
1313              0 :                                  return 0;
1314              0 :                          sbi->s_min_batch_time = option;
1315              0 :                          break;
1316                :                  case Opt_data_journal:
1317             12 :                          data_opt = EXT4_MOUNT_JOURNAL_DATA;
1318             12 :                          goto datacheck;
1319                :                  case Opt_data_ordered:
1320             12 :                          data_opt = EXT4_MOUNT_ORDERED_DATA;
1321             12 :                          goto datacheck;
1322                :                  case Opt_data_writeback:
1323             12 :                          data_opt = EXT4_MOUNT_WRITEBACK_DATA;
1324             36 :                  datacheck:
1325             36 :                          if (is_remount) {
1326              0 :                                  if ((sbi->s_mount_opt & EXT4_MOUNT_DATA_FLAGS)
1327                :                                          != data_opt) {
1328              0 :                                          ext4_msg(sb, KERN_ERR,
1329                :                                                   "Cannot change data mode on remount");
1330              0 :                                          return 0;
1331                :                                  }
1332                :                          } else {
1333             36 :                                  sbi->s_mount_opt &= ~EXT4_MOUNT_DATA_FLAGS;
1334             36 :                                  sbi->s_mount_opt |= data_opt;
```

```
1335                :                         }
1336                :                         break;
1337                :                 case Opt_data_err_abort:
1338              0 :                         set_opt(sbi->s_mount_opt, DATA_ERR_ABORT);
1339              0 :                         break;
1340                :                 case Opt_data_err_ignore:
1341              0 :                         clear_opt(sbi->s_mount_opt, DATA_ERR_ABORT);
1342              0 :                         break;
1343                :                 case Opt_mb_history_length:
1344              0 :                         if (match_int(&args[0], &option))
1345              0 :                                 return 0;
1346              0 :                         if (option < 0)
1347              0 :                                 return 0;
1348              0 :                         sbi->s_mb_history_max = option;
1349              0 :                         break;
1350                : #ifdef CONFIG_QUOTA
1351                :                 case Opt_usrjquota:
1352              0 :                         qtype = USRQUOTA;
1353              0 :                         goto set_qf_name;
1354                :                 case Opt_grpjquota:
1355              0 :                         qtype = GRPQUOTA;
1356              0 : set_qf_name:
1357              0 :                         if (sb_any_quota_loaded(sb) &&
1358                :                             !sbi->s_qf_names[qtype]) {
1359              0 :                                 ext4_msg(sb, KERN_ERR,
1360                :                                        "Cannot change journaled "
1361                :                                        "quota options when quota turned on");
1362              0 :                                 return 0;
1363                :                         }
1364              0 :                         qname = match_strdup(&args[0]);
1365              0 :                         if (!qname) {
1366              0 :                                 ext4_msg(sb, KERN_ERR,
1367                :                                         "Not enough memory for "
1368                :                                         "storing quotafile name");
1369              0 :                                 return 0;
1370                :                         }
1371              0 :                         if (sbi->s_qf_names[qtype] &&
1372                :                             strcmp(sbi->s_qf_names[qtype], qname)) {
1373              0 :                                 ext4_msg(sb, KERN_ERR,
1374                :                                         "%s quota file already "
1375                :                                         "specified", QTYPE2NAME(qtype));
1376              0 :                                 kfree(qname);
1377              0 :                                 return 0;
1378                :                         }
1379              0 :                         sbi->s_qf_names[qtype] = qname;
1380              0 :                         if (strchr(sbi->s_qf_names[qtype], '/')) {
1381              0 :                                 ext4_msg(sb, KERN_ERR,
1382                :                                         "quotafile must be on "
1383                :                                         "filesystem root");
1384              0 :                                 kfree(sbi->s_qf_names[qtype]);
1385              0 :                                 sbi->s_qf_names[qtype] = NULL;
1386              0 :                                 return 0;
1387                :                         }
1388              0 :                         set_opt(sbi->s_mount_opt, QUOTA);
1389              0 :                         break;
1390                :                 case Opt_offusrjquota:
1391              0 :                         qtype = USRQUOTA;
1392              0 :                         goto clear_qf_name;
1393                :                 case Opt_offgrpjquota:
1394              0 :                         qtype = GRPQUOTA;
1395              0 : clear_qf_name:
1396              0 :                         if (sb_any_quota_loaded(sb) &&
1397                :                             sbi->s_qf_names[qtype]) {
1398              0 :                                 ext4_msg(sb, KERN_ERR, "Cannot change "
1399                :                                         "journaled quota options when "
1400                :                                         "quota turned on");
1401              0 :                                 return 0;
1402                :                         }
1403                :                         /*
1404                :                          * The space will be released later when all options
1405                :                          * are confirmed to be correct
1406                :                          */
1407              0 :                         sbi->s_qf_names[qtype] = NULL;
1408              0 :                         break;
1409                :                 case Opt_jqfmt_vfsold:
1410              0 :                         qfmt = QFMT_VFS_OLD;
1411              0 :                         goto set_qf_format;
1412                :                 case Opt_jqfmt_vfsv0:
1413              0 :                         qfmt = QFMT_VFS_V0;
1414              0 : set_qf_format:
1415              0 :                         if (sb_any_quota_loaded(sb) &&
1416                :                             sbi->s_jquota_fmt != qfmt) {
1417              0 :                                 ext4_msg(sb, KERN_ERR, "Cannot change "
1418                :                                         "journaled quota options when "
1419                :                                         "quota turned on");
1420              0 :                                 return 0;
1421                :                         }
1422              0 :                         sbi->s_jquota_fmt = qfmt;
1423              0 :                         break;
1424                :                 case Opt_quota:
```

```
1425                :                 case Opt_usrquota:
1426              0 :                         set_opt(sbi->s_mount_opt, QUOTA);
1427              0 :                         set_opt(sbi->s_mount_opt, USRQUOTA);
1428              0 :                         break;
1429                :                 case Opt_grpquota:
1430              0 :                         set_opt(sbi->s_mount_opt, QUOTA);
1431              0 :                         set_opt(sbi->s_mount_opt, GRPQUOTA);
1432              0 :                         break;
1433                :                 case Opt_noquota:
1434              0 :                         if (sb_any_quota_loaded(sb)) {
1435              0 :                                 ext4_msg(sb, KERN_ERR, "Cannot change quota "
1436                :                                         "options when quota turned on");
1437              0 :                                 return 0;
1438                :                         }
1439              0 :                         clear_opt(sbi->s_mount_opt, QUOTA);
1440              0 :                         clear_opt(sbi->s_mount_opt, USRQUOTA);
1441              0 :                         clear_opt(sbi->s_mount_opt, GRPQUOTA);
1442              0 :                         break;
1443                : #else
1444                :                 case Opt_quota:
1445                :                 case Opt_usrquota:
1446                :                 case Opt_grpquota:
1447                :                         ext4_msg(sb, KERN_ERR,
1448                :                                 "quota options not supported");
1449                :                         break;
1450                :                 case Opt_usrjquota:
1451                :                 case Opt_grpjquota:
1452                :                 case Opt_offusrjquota:
1453                :                 case Opt_offgrpjquota:
1454                :                 case Opt_jqfmt_vfsold:
1455                :                 case Opt_jqfmt_vfsv0:
1456                :                         ext4_msg(sb, KERN_ERR,
1457                :                                 "journaled quota options not supported");
1458                :                         break;
1459                :                 case Opt_noquota:
1460                :                         break;
1461                : #endif
1462                :                 case Opt_abort:
1463              0 :                         sbi->s_mount_flags |= EXT4_MF_FS_ABORTED;
1464              0 :                         break;
1465                :                 case Opt_nobarrier:
1466              0 :                         clear_opt(sbi->s_mount_opt, BARRIER);
1467              0 :                         break;
1468                :                 case Opt_barrier:
1469             36 :                         if (match_int(&args[0], &option)) {
1470              0 :                                 set_opt(sbi->s_mount_opt, BARRIER);
1471              0 :                                 break;
1472                :                         }
1473             36 :                         if (option)
1474             18 :                                 set_opt(sbi->s_mount_opt, BARRIER);
1475                :                         else
1476             18 :                                 clear_opt(sbi->s_mount_opt, BARRIER);
1477                :                         break;
1478                :                 case Opt_ignore:
1479                :                         break;
1480                :                 case Opt_resize:
1481              0 :                         if (!is_remount) {
1482              0 :                                 ext4_msg(sb, KERN_ERR,
1483                :                                         "resize option only available "
1484                :                                         "for remount");
1485              0 :                                 return 0;
1486                :                         }
1487              0 :                         if (match_int(&args[0], &option) != 0)
1488              0 :                                 return 0;
1489              0 :                         *n_blocks_count = option;
1490              0 :                         break;
1491                :                 case Opt_nobh:
1492              0 :                         set_opt(sbi->s_mount_opt, NOBH);
1493              0 :                         break;
1494                :                 case Opt_bh:
1495              0 :                         clear_opt(sbi->s_mount_opt, NOBH);
1496              0 :                         break;
1497                :                 case Opt_i_version:
1498              8 :                         set_opt(sbi->s_mount_opt, I_VERSION);
1499              8 :                         sb->s_flags |= MS_I_VERSION;
1500              8 :                         break;
1501                :                 case Opt_nodelalloc:
1502             11 :                         clear_opt(sbi->s_mount_opt, DELALLOC);
1503             11 :                         break;
1504                :                 case Opt_stripe:
1505              0 :                         if (match_int(&args[0], &option))
1506              0 :                                 return 0;
1507              0 :                         if (option < 0)
1508              0 :                                 return 0;
1509              0 :                         sbi->s_stripe = option;
1510              0 :                         break;
1511                :                 case Opt_delalloc:
1512             15 :                         set_opt(sbi->s_mount_opt, DELALLOC);
1513             15 :                         break;
1514                :                 case Opt_block_validity:
```

```
1515        0 :                         set_opt(sbi->s_mount_opt, BLOCK_VALIDITY);
1516        0 :                         break;
1517          :                 case Opt_noblock_validity:
1518        0 :                         clear_opt(sbi->s_mount_opt, BLOCK_VALIDITY);
1519        0 :                         break;
1520          :                 case Opt_inode_readahead_blks:
1521        0 :                         if (match_int(&args[0], &option))
1522        0 :                                 return 0;
1523        0 :                         if (option < 0 || option > (1 << 30))
1524        0 :                                 return 0;
1525        0 :                         if (!is_power_of_2(option)) {
1526        0 :                                 ext4_msg(sb, KERN_ERR,
1527          :                                          "EXT4-fs: inode_readahead_blks"
1528          :                                          " must be a power of 2");
1529        0 :                                 return 0;
1530          :                         }
1531        0 :                         sbi->s_inode_readahead_blks = option;
1532        0 :                         break;
1533          :                 case Opt_journal_ioprio:
1534        0 :                         if (match_int(&args[0], &option))
1535        0 :                                 return 0;
1536        0 :                         if (option < 0 || option > 7)
1537        0 :                                 break;
1538        0 :                         *journal_ioprio = IOPRIO_PRIO_VALUE(IOPRIO_CLASS_BE,
1539          :                                                             option);
1540        0 :                         break;
1541          :                 case Opt_noauto_da_alloc:
1542        8 :                         set_opt(sbi->s_mount_opt,NO_AUTO_DA_ALLOC);
1543        8 :                         break;
1544          :                 case Opt_auto_da_alloc:
1545        9 :                         if (match_int(&args[0], &option)) {
1546        1 :                                 clear_opt(sbi->s_mount_opt, NO_AUTO_DA_ALLOC);
1547        1 :                                 break;
1548          :                         }
1549        8 :                         if (option)
1550        8 :                                 clear_opt(sbi->s_mount_opt, NO_AUTO_DA_ALLOC);
1551          :                         else
1552        0 :                                 set_opt(sbi->s_mount_opt,NO_AUTO_DA_ALLOC);
1553          :                         break;
1554          :                 default:
1555        0 :                         ext4_msg(sb, KERN_ERR,
1556          :                                "Unrecognized mount option \"%s\" "
1557          :                                "or missing value", p);
1558        0 :                         return 0;
1559          :                 }
1560          :         }
1561          : #ifdef CONFIG_QUOTA
1562       70 :         if (sbi->s_qf_names[USRQUOTA] || sbi->s_qf_names[GRPQUOTA]) {
1563        0 :                 if ((sbi->s_mount_opt & EXT4_MOUNT_USRQUOTA) &&
1564          :                      sbi->s_qf_names[USRQUOTA])
1565        0 :                         clear_opt(sbi->s_mount_opt, USRQUOTA);
1566          :
1567        0 :                 if ((sbi->s_mount_opt & EXT4_MOUNT_GRPQUOTA) &&
1568          :                      sbi->s_qf_names[GRPQUOTA])
1569        0 :                         clear_opt(sbi->s_mount_opt, GRPQUOTA);
1570          :
1571        0 :                 if ((sbi->s_qf_names[USRQUOTA] &&
1572          :                                 (sbi->s_mount_opt & EXT4_MOUNT_GRPQUOTA)) ||
1573          :                     (sbi->s_qf_names[GRPQUOTA] &&
1574          :                                 (sbi->s_mount_opt & EXT4_MOUNT_USRQUOTA))) {
1575        0 :                         ext4_msg(sb, KERN_ERR, "old and new quota "
1576          :                                         "format mixing");
1577        0 :                         return 0;
1578          :                 }
1579          :
1580        0 :                 if (!sbi->s_jquota_fmt) {
1581        0 :                         ext4_msg(sb, KERN_ERR, "journaled quota format "
1582          :                                         "not specified");
1583        0 :                         return 0;
1584          :                 }
1585          :         } else {
1586       70 :                 if (sbi->s_jquota_fmt) {
1587        0 :                         ext4_msg(sb, KERN_ERR, "journaled quota format "
1588          :                                         "specified with no journaling "
1589          :                                         "enabled");
1590        0 :                         return 0;
1591          :                 }
1592          :         }
1593          : #endif
1594       70 :         return 1;
1595          : }
1596          :
1597          : static int ext4_setup_super(struct super_block *sb, struct ext4_super_block *es,
1598          :                             int read_only)
1599       94 : {
1600       94 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
1601       94 :         int res = 0;
1602          :
1603       94 :         if (le32_to_cpu(es->s_rev_level) > EXT4_MAX_SUPP_REV) {
1604        0 :                 ext4_msg(sb, KERN_ERR, "revision level too high, "
```

```
1605                :                           "forcing read-only mode");
1606              0 :                   res = MS_RDONLY;
1607                :           }
1608             94 :           if (read_only)
1609              0 :                   return res;
1610             94 :           if (!(sbi->s_mount_state & EXT4_VALID_FS))
1611              0 :                   ext4_msg(sb, KERN_WARNING, "warning: mounting unchecked fs, "
1612                :                           "running e2fsck is recommended");
1613             94 :           else if ((sbi->s_mount_state & EXT4_ERROR_FS))
1614              0 :                   ext4_msg(sb, KERN_WARNING,
1615                :                           "warning: mounting fs with errors, "
1616                :                           "running e2fsck is recommended");
1617             94 :           else if ((__s16) le16_to_cpu(es->s_max_mnt_count) >= 0 &&
1618                :                    le16_to_cpu(es->s_mnt_count) >=
1619                :                    (unsigned short) (__s16) le16_to_cpu(es->s_max_mnt_count))
1620              0 :                   ext4_msg(sb, KERN_WARNING,
1621                :                           "warning: maximal mount count reached, "
1622                :                           "running e2fsck is recommended");
1623             94 :           else if (le32_to_cpu(es->s_checkinterval) &&
1624                :                   (le32_to_cpu(es->s_lastcheck) +
1625                :                           le32_to_cpu(es->s_checkinterval) <= get_seconds()))
1626              0 :                   ext4_msg(sb, KERN_WARNING,
1627                :                           "warning: checktime reached, "
1628                :                           "running e2fsck is recommended");
1629             94 :           if (!sbi->s_journal)
1630              0 :                   es->s_state &= cpu_to_le16(~EXT4_VALID_FS);
1631             94 :           if (!(__s16) le16_to_cpu(es->s_max_mnt_count))
1632              0 :                   es->s_max_mnt_count = cpu_to_le16(EXT4_DFL_MAX_MNT_COUNT);
1633             94 :           le16_add_cpu(&es->s_mnt_count, 1);
1634             94 :           es->s_mtime = cpu_to_le32(get_seconds());
1635             94 :           ext4_update_dynamic_rev(sb);
1636             94 :           if (sbi->s_journal)
1637             94 :                   EXT4_SET_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER);
1638                :
1639             94 :           ext4_commit_super(sb, 1);
1640             94 :           if (test_opt(sb, DEBUG))
1641              0 :                   printk(KERN_INFO "[EXT4 FS bs=%lu, gc=%u, "
1642                :                           "bpg=%lu, ipg=%lu, mo=%04x]\n",
1643                :                           sb->s_blocksize,
1644                :                           sbi->s_groups_count,
1645                :                           EXT4_BLOCKS_PER_GROUP(sb),
1646                :                           EXT4_INODES_PER_GROUP(sb),
1647                :                           sbi->s_mount_opt);
1648                :
1649             94 :           if (EXT4_SB(sb)->s_journal) {
1650            188 :                   ext4_msg(sb, KERN_INFO, "%s journal on %s",
1651                :                           EXT4_SB(sb)->s_journal->j_inode ? "internal" :
1652                :                           "external", EXT4_SB(sb)->s_journal->j_devname);
1653                :           } else {
1654              0 :                   ext4_msg(sb, KERN_INFO, "no journal");
1655                :           }
1656             94 :           return res;
1657                : }
1658                :
1659                : static int ext4_fill_flex_info(struct super_block *sb)
1660             77 : {
1661             77 :           struct ext4_sb_info *sbi = EXT4_SB(sb);
1662             77 :           struct ext4_group_desc *gdp = NULL;
1663                :           ext4_group_t flex_group_count;
1664                :           ext4_group_t flex_group;
1665             77 :           int groups_per_flex = 0;
1666                :           size_t size;
1667                :           int i;
1668                :
1669             77 :           if (!sbi->s_es->s_log_groups_per_flex) {
1670              0 :                   sbi->s_log_groups_per_flex = 0;
1671              0 :                   return 1;
1672                :           }
1673                :
1674             77 :           sbi->s_log_groups_per_flex = sbi->s_es->s_log_groups_per_flex;
1675             77 :           groups_per_flex = 1 << sbi->s_log_groups_per_flex;
1676                :
1677                :           /* We allocate both existing and potentially added groups */
1678            154 :           flex_group_count = ((sbi->s_groups_count + groups_per_flex - 1) +
1679                :                           ((le16_to_cpu(sbi->s_es->s_reserved_gdt_blocks) + 1) <<
1680                :                                   EXT4_DESC_PER_BLOCK_BITS(sb))) / groups_per_flex;
1681             77 :           size = flex_group_count * sizeof(struct flex_groups);
1682             77 :           sbi->s_flex_groups = kzalloc(size, GFP_KERNEL);
1683             77 :           if (sbi->s_flex_groups == NULL) {
1684              0 :                   sbi->s_flex_groups = vmalloc(size);
1685              0 :                   if (sbi->s_flex_groups)
1686              0 :                           memset(sbi->s_flex_groups, 0, size);
1687                :           }
1688             77 :           if (sbi->s_flex_groups == NULL) {
1689              0 :                   ext4_msg(sb, KERN_ERR, "not enough memory for "
1690                :                           "%u flex groups", flex_group_count);
1691              0 :                   goto failed;
1692                :           }
1693                :
1694        1545037 :           for (i = 0; i < sbi->s_groups_count; i++) {
```

```
1695  1544960 :                 gdp = ext4_get_group_desc(sb, i, NULL);
1696          :
1697  3089920 :                 flex_group = ext4_flex_group(sbi, i);
1698  1544960 :                 atomic_set(&sbi->s_flex_groups[flex_group].free_inodes,
1699          :                            ext4_free_inodes_count(sb, gdp));
1700  1544960 :                 atomic_set(&sbi->s_flex_groups[flex_group].free_blocks,
1701          :                            ext4_free_blks_count(sb, gdp));
1702  1544960 :                 atomic_set(&sbi->s_flex_groups[flex_group].used_dirs,
1703          :                            ext4_used_dirs_count(sb, gdp));
1704          :         }
1705          :
1706       77 :         return 1;
1707        0 : failed:
1708        0 :         return 0;
1709          : }
1710          :
1711          : __le16 ext4_group_desc_csum(struct ext4_sb_info *sbi, __u32 block_group,
1712          :                             struct ext4_group_desc *gdp)
1713  238568732 : {
1714  238568732 :         __u16 crc = 0;
1715          :
1716  238568732 :         if (sbi->s_es->s_feature_ro_compat &
1717          :             cpu_to_le32(EXT4_FEATURE_RO_COMPAT_GDT_CSUM)) {
1718  238155460 :                 int offset = offsetof(struct ext4_group_desc, bg_checksum);
1719  238155460 :                 __le32 le_group = cpu_to_le32(block_group);
1720          :
1721  238155460 :                 crc = crc16(~0, sbi->s_es->s_uuid, sizeof(sbi->s_es->s_uuid));
1722  239158431 :                 crc = crc16(crc, (__u8 *)&le_group, sizeof(le_group));
1723  239156536 :                 crc = crc16(crc, (__u8 *)gdp, offset);
1724  239164277 :                 offset += sizeof(gdp->bg_checksum); /* skip checksum */
1725          :                 /* for checksum of struct ext4_group_desc do the rest...*/
1726  239164277 :                 if ((sbi->s_es->s_feature_incompat &
1727          :                      cpu_to_le32(EXT4_FEATURE_INCOMPAT_64BIT)) &&
1728          :                     offset < le16_to_cpu(sbi->s_es->s_desc_size))
1729        0 :                         crc = crc16(crc, (__u8 *)gdp + offset,
1730          :                                     le16_to_cpu(sbi->s_es->s_desc_size) -
1731          :                                         offset);
1732          :         }
1733          :
1734  239577549 :         return cpu_to_le16(crc);
1735          : }
1736          :
1737          : int ext4_group_desc_csum_verify(struct ext4_sb_info *sbi, __u32 block_group,
1738          :                                 struct ext4_group_desc *gdp)
1739    1922299 : {
1740    1922299 :         if ((sbi->s_es->s_feature_ro_compat &
1741          :              cpu_to_le32(EXT4_FEATURE_RO_COMPAT_GDT_CSUM)) &&
1742          :             (gdp->bg_checksum != ext4_group_desc_csum(sbi, block_group, gdp)))
1743        0 :                 return 0;
1744          :
1745    1922299 :         return 1;
1746          : }
1747          :
1748          : /* Called at mount-time, super-block is locked */
1749          : static int ext4_check_descriptors(struct super_block *sb)
1750         94 : {
1751         94 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
1752         94 :         ext4_fsblk_t first_block = le32_to_cpu(sbi->s_es->s_first_data_block);
1753          :         ext4_fsblk_t last_block;
1754          :         ext4_fsblk_t block_bitmap;
1755          :         ext4_fsblk_t inode_bitmap;
1756          :         ext4_fsblk_t inode_table;
1757         94 :         int flexbg_flag = 0;
1758          :         ext4_group_t i;
1759          :
1760         94 :         if (EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_FLEX_BG))
1761         77 :                 flexbg_flag = 1;
1762          :
1763          :         ext4_debug("Checking group descriptors");
1764          :
1765    1719134 :         for (i = 0; i < sbi->s_groups_count; i++) {
1766    1719040 :                 struct ext4_group_desc *gdp = ext4_get_group_desc(sb, i, NULL);
1767          :
1768    1719040 :                 if (i == sbi->s_groups_count - 1 || flexbg_flag)
1769    3089954 :                         last_block = ext4_blocks_count(sbi->s_es) - 1;
1770          :                 else
1771     174063 :                         last_block = first_block +
1772          :                                 (EXT4_BLOCKS_PER_GROUP(sb) - 1);
1773          :
1774    1719040 :                 block_bitmap = ext4_block_bitmap(sb, gdp);
1775    1719040 :                 if (block_bitmap < first_block || block_bitmap > last_block) {
1776        0 :                         ext4_msg(sb, KERN_ERR, "ext4_check_descriptors: "
1777          :                                "Block bitmap for group %u not in group "
1778          :                                "(block %llu)!", i, block_bitmap);
1779        0 :                         return 0;
1780          :                 }
1781    1719040 :                 inode_bitmap = ext4_inode_bitmap(sb, gdp);
1782    1719040 :                 if (inode_bitmap < first_block || inode_bitmap > last_block) {
1783        0 :                         ext4_msg(sb, KERN_ERR, "ext4_check_descriptors: "
1784          :                                "Inode bitmap for group %u not in group "
```

```
1785                :                                "(block %llu)!", i, inode_bitmap);
1786              0 :                                return 0;
1787                :                        }
1788        1719040 :                        inode_table = ext4_inode_table(sb, gdp);
1789        1719040 :                        if (inode_table < first_block ||
1790                :                            inode_table + sbi->s_itb_per_group - 1 > last_block) {
1791              0 :                                ext4_msg(sb, KERN_ERR, "ext4_check_descriptors: "
1792                :                                       "Inode table for group %u not in group "
1793                :                                       "(block %llu)!", i, inode_table);
1794              0 :                                return 0;
1795                :                        }
1796                :                        ext4_lock_group(sb, i);
1797        1719040 :                        if (!ext4_group_desc_csum_verify(sbi, i, gdp)) {
1798              0 :                                ext4_msg(sb, KERN_ERR, "ext4_check_descriptors: "
1799                :                                         "Checksum for group %u failed (%u!=%u)",
1800                :                                         i, le16_to_cpu(ext4_group_desc_csum(sbi, i,
1801                :                                             gdp)), le16_to_cpu(gdp->bg_checksum));
1802              0 :                                if (!(sb->s_flags & MS_RDONLY)) {
1803                :                                        ext4_unlock_group(sb, i);
1804              0 :                                        return 0;
1805                :                                }
1806                :                        }
1807                :                        ext4_unlock_group(sb, i);
1808        1719040 :                        if (!flexbg_flag)
1809         174080 :                                first_block += EXT4_BLOCKS_PER_GROUP(sb);
1810                :                }
1811                :
1812             94 :                ext4_free_blocks_count_set(sbi->s_es, ext4_count_free_blocks(sb));
1813             94 :                sbi->s_es->s_free_inodes_count =cpu_to_le32(ext4_count_free_inodes(sb));
1814             94 :                return 1;
1815                : }
1816                :
1817                : /* ext4_orphan_cleanup() walks a singly-linked list of inodes (starting at
1818                :  * the superblock) which were deleted from all directories, but held open by
1819                :  * a process at the time of a crash.  We walk the list and try to delete these
1820                :  * inodes at recovery time (only with a read-write filesystem).
1821                :  *
1822                :  * In order to keep the orphan inode chain consistent during traversal (in
1823                :  * case of crash during recovery), we link each inode into the superblock
1824                :  * orphan list_head and handle it the same way as an inode deletion during
1825                :  * normal operation (which journals the operations for us).
1826                :  *
1827                :  * We only do an iget() and an iput() on each inode, which is very safe if we
1828                :  * accidentally point at an in-use or already deleted inode.  The worst that
1829                :  * can happen in this case is that we get a "bit already cleared" message from
1830                :  * ext4_free_inode().  The only reason we would point at a wrong inode is if
1831                :  * e2fsck was run on this filesystem, and it must have already done the orphan
1832                :  * inode cleanup for us, so we can safely abort without any further action.
1833                :  */
1834                : static void ext4_orphan_cleanup(struct super_block *sb,
1835                :                                 struct ext4_super_block *es)
1836             94 : {
1837             94 :                unsigned int s_flags = sb->s_flags;
1838             94 :                int nr_orphans = 0, nr_truncates = 0;
1839                : #ifdef CONFIG_QUOTA
1840                :                int i;
1841                : #endif
1842             94 :                if (!es->s_last_orphan) {
1843             94 :                        jbd_debug(4, "no orphan inodes to clean up\n");
1844                :                        return;
1845                :                }
1846                :
1847              0 :                if (bdev_read_only(sb->s_bdev)) {
1848              0 :                        ext4_msg(sb, KERN_ERR, "write access "
1849                :                                        "unavailable, skipping orphan cleanup");
1850              0 :                        return;
1851                :                }
1852                :
1853              0 :                if (EXT4_SB(sb)->s_mount_state & EXT4_ERROR_FS) {
1854              0 :                        if (es->s_last_orphan)
1855              0 :                                jbd_debug(1, "Errors on filesystem, "
1856                :                                          "clearing orphan list.\n");
1857              0 :                        es->s_last_orphan = 0;
1858              0 :                        jbd_debug(1, "Skipping orphan recovery on fs with errors.\n");
1859                :                        return;
1860                :                }
1861                :
1862              0 :                if (s_flags & MS_RDONLY) {
1863              0 :                        ext4_msg(sb, KERN_INFO, "orphan cleanup on readonly fs");
1864              0 :                        sb->s_flags &= ~MS_RDONLY;
1865                :                }
1866                : #ifdef CONFIG_QUOTA
1867                :                /* Needed for iput() to work correctly and not trash data */
1868              0 :                sb->s_flags |= MS_ACTIVE;
1869                :                /* Turn on quotas so that they are updated correctly */
1870              0 :                for (i = 0; i < MAXQUOTAS; i++) {
1871              0 :                        if (EXT4_SB(sb)->s_qf_names[i]) {
1872              0 :                                int ret = ext4_quota_on_mount(sb, i);
1873              0 :                                if (ret < 0)
1874              0 :                                        ext4_msg(sb, KERN_ERR,
```

```
1875                :                                                       "Cannot turn on journaled "
1876                :                                                       "quota: error %d", ret);
1877                :                       }
1878                :               }
1879                : #endif
1880                :
1881              0 :               while (es->s_last_orphan) {
1882                :                       struct inode *inode;
1883                :
1884              0 :                       inode = ext4_orphan_get(sb, le32_to_cpu(es->s_last_orphan));
1885              0 :                       if (IS_ERR(inode)) {
1886              0 :                               es->s_last_orphan = 0;
1887              0 :                               break;
1888                :                       }
1889                :
1890              0 :                       list_add(&EXT4_I(inode)->i_orphan, &EXT4_SB(sb)->s_orphan);
1891                :                       vfs_dq_init(inode);
1892              0 :                       if (inode->i_nlink) {
1893              0 :                               ext4_msg(sb, KERN_DEBUG,
1894                :                                       "%s: truncating inode %lu to %lld bytes",
1895                :                                       __func__, inode->i_ino, inode->i_size);
1896              0 :                               jbd_debug(2, "truncating inode %lu to %lld bytes\n",
1897                :                                         inode->i_ino, inode->i_size);
1898              0 :                               ext4_truncate(inode);
1899              0 :                               nr_truncates++;
1900                :                       } else {
1901              0 :                               ext4_msg(sb, KERN_DEBUG,
1902                :                                       "%s: deleting unreferenced inode %lu",
1903                :                                       __func__, inode->i_ino);
1904              0 :                               jbd_debug(2, "deleting unreferenced inode %lu\n",
1905                :                                         inode->i_ino);
1906              0 :                               nr_orphans++;
1907                :                       }
1908              0 :                       iput(inode);  /* The delete magic happens here! */
1909                :               }
1910                :
1911                : #define PLURAL(x) (x), ((x) == 1) ? "" : "s"
1912                :
1913              0 :               if (nr_orphans)
1914              0 :                       ext4_msg(sb, KERN_INFO, "%d orphan inode%s deleted",
1915                :                               PLURAL(nr_orphans));
1916              0 :               if (nr_truncates)
1917              0 :                       ext4_msg(sb, KERN_INFO, "%d truncate%s cleaned up",
1918                :                               PLURAL(nr_truncates));
1919                : #ifdef CONFIG_QUOTA
1920                :               /* Turn quotas off */
1921              0 :               for (i = 0; i < MAXQUOTAS; i++) {
1922              0 :                       if (sb_dqopt(sb)->files[i])
1923              0 :                               vfs_quota_off(sb, i, 0);
1924                :               }
1925                : #endif
1926              0 :               sb->s_flags = s_flags; /* Restore MS_RDONLY status */
1927                : }
1928                :
1929                : /*
1930                :  * Maximal extent format file size.
1931                :  * Resulting logical blkno at s_maxbytes must fit in our on-disk
1932                :  * extent format containers, within a sector_t, and within i_blocks
1933                :  * in the vfs.  ext4 inode has 48 bits of i_block in fsblock units,
1934                :  * so that won't be a limiting factor.
1935                :  *
1936                :  * Note, this does *not* consider any metadata overhead for vfs i_blocks.
1937                :  */
1938                : static loff_t ext4_max_size(int blkbits, int has_huge_files)
1939                : {
1940                :               loff_t res;
1941             94 :               loff_t upper_limit = MAX_LFS_FILESIZE;
1942                :
1943                :               /* small i_blocks in vfs inode? */
1944             94 :               if (!has_huge_files || sizeof(blkcnt_t) < sizeof(u64)) {
1945                :                       /*
1946                :                        * CONFIG_LBDAF is not enabled implies the inode
1947                :                        * i_block represent total blocks in 512 bytes
1948                :                        * 32 == size of vfs inode i_blocks * 8
1949                :                        */
1950             17 :                       upper_limit = (1LL << 32) - 1;
1951                :
1952                :                       /* total blocks in file system block size */
1953             17 :                       upper_limit >>= (blkbits - 9);
1954             17 :                       upper_limit <<= blkbits;
1955                :               }
1956                :
1957                :               /* 32-bit extent-start container, ee_block */
1958             94 :               res = 1LL << 32;
1959             94 :               res <<= blkbits;
1960             94 :               res -= 1;
1961                :
1962                :               /* Sanity check against vm- & vfs- imposed limits */
1963             94 :               if (res > upper_limit)
1964             77 :                       res = upper_limit;
```

```
1965                :
1966             94 :         return res;
1967                : }
1968                :
1969                : /*
1970                :  * Maximal bitmap file size.  There is a direct, and {,double-,triple-}indirect
1971                :  * block limit, and also a limit of (2^48 - 1) 512-byte sectors in i_blocks.
1972                :  * We need to be 1 filesystem block less than the 2^48 sector limit.
1973                :  */
1974                : static loff_t ext4_max_bitmap_size(int bits, int has_huge_files)
1975                : {
1976             94 :         loff_t res = EXT4_NDIR_BLOCKS;
1977                :         int meta_blocks;
1978                :         loff_t upper_limit;
1979                :         /* This is calculated to be the largest file size for a dense, block
1980                :          * mapped file such that the file's total number of 512-byte sectors,
1981                :          * including data and all indirect blocks, does not exceed (2^48 - 1).
1982                :          *
1983                :          * __u32 i_blocks_lo and _u16 i_blocks_high represent the total
1984                :          * number of 512-byte sectors of the file.
1985                :          */
1986                :
1987             94 :         if (!has_huge_files || sizeof(blkcnt_t) < sizeof(u64)) {
1988                :                 /*
1989                :                  * !has_huge_files or CONFIG_LBDAF not enabled implies that
1990                :                  * the inode i_block field represents total file blocks in
1991                :                  * 2^32 512-byte sectors == size of vfs inode i_blocks * 8
1992                :                  */
1993             17 :                 upper_limit = (1LL << 32) - 1;
1994                :
1995                :                 /* total blocks in file system block size */
1996             17 :                 upper_limit >>= (bits - 9);
1997                :
1998                :         } else {
1999                :                 /*
2000                :                  * We use 48 bit ext4_inode i_blocks
2001                :                  * With EXT4_HUGE_FILE_FL set the i_blocks
2002                :                  * represent total number of blocks in
2003                :                  * file system block size
2004                :                  */
2005             77 :                 upper_limit = (1LL << 48) - 1;
2006                :
2007                :         }
2008                :
2009                :         /* indirect blocks */
2010             94 :         meta_blocks = 1;
2011                :         /* double indirect blocks */
2012             94 :         meta_blocks += 1 + (1LL << (bits-2));
2013                :         /* tripple indirect blocks */
2014             94 :         meta_blocks += 1 + (1LL << (bits-2)) + (1LL << (2*(bits-2)));
2015                :
2016             94 :         upper_limit -= meta_blocks;
2017             94 :         upper_limit <<= bits;
2018                :
2019             94 :         res += 1LL << (bits-2);
2020             94 :         res += 1LL << (2*(bits-2));
2021             94 :         res += 1LL << (3*(bits-2));
2022             94 :         res <<= bits;
2023             94 :         if (res > upper_limit)
2024             16 :                 res = upper_limit;
2025                :
2026             94 :         if (res > MAX_LFS_FILESIZE)
2027              0 :                 res = MAX_LFS_FILESIZE;
2028                :
2029             94 :         return res;
2030                : }
2031                :
2032                : static ext4_fsblk_t descriptor_loc(struct super_block *sb,
2033                :                                    ext4_fsblk_t logical_sb_block, int nr)
2034                : {
2035          42708 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2036                :         ext4_group_t bg, first_meta_bg;
2037          42708 :         int has_super = 0;
2038                :
2039          42708 :         first_meta_bg = le32_to_cpu(sbi->s_es->s_first_meta_bg);
2040                :
2041          42708 :         if (!EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_META_BG) ||
2042                :             nr < first_meta_bg)
2043          42708 :                 return logical_sb_block + nr + 1;
2044              0 :         bg = sbi->s_desc_per_block * nr;
2045              0 :         if (ext4_bg_has_super(sb, bg))
2046              0 :                 has_super = 1;
2047                :
2048              0 :         return (has_super + ext4_group_first_block_no(sb, bg));
2049                : }
2050                :
2051                : /**
2052                :  * ext4_get_stripe_size: Get the stripe size.
2053                :  * @sbi: In memory super block info
2054                :  *
```

```
2055                :  * If we have specified it via mount option, then
2056                :  * use the mount option value. If the value specified at mount time is
2057                :  * greater than the blocks per group use the super block value.
2058                :  * If the super block value is greater than blocks per group return 0.
2059                :  * Allocator needs it be less than blocks per group.
2060                :  *
2061                :  */
2062                : static unsigned long ext4_get_stripe_size(struct ext4_sb_info *sbi)
2063                : {
2064               94 :         unsigned long stride = le16_to_cpu(sbi->s_es->s_raid_stride);
2065                :         unsigned long stripe_width =
2066               94 :                         le32_to_cpu(sbi->s_es->s_raid_stripe_width);
2067                :
2068               94 :         if (sbi->s_stripe && sbi->s_stripe <= sbi->s_blocks_per_group)
2069                0 :                 return sbi->s_stripe;
2070                :
2071               94 :         if (stripe_width <= sbi->s_blocks_per_group)
2072               94 :                 return stripe_width;
2073                :
2074                0 :         if (stride <= sbi->s_blocks_per_group)
2075                0 :                 return stride;
2076                :
2077                0 :         return 0;
2078                : }
2079                :
2080                : /* sysfs supprt */
2081                :
2082                : struct ext4_attr {
2083                :         struct attribute attr;
2084                :         ssize_t (*show)(struct ext4_attr *, struct ext4_sb_info *, char *);
2085                :         ssize_t (*store)(struct ext4_attr *, struct ext4_sb_info *,
2086                :                          const char *, size_t);
2087                :         int offset;
2088                : };
2089                :
2090                : static int parse_strtoul(const char *buf,
2091                :                 unsigned long max, unsigned long *value)
2092                : {
2093                :         char *endp;
2094                :
2095                0 :         while (*buf && isspace(*buf))
2096                0 :                 buf++;
2097                0 :         *value = simple_strtoul(buf, &endp, 0);
2098                0 :         while (*endp && isspace(*endp))
2099                0 :                 endp++;
2100                0 :         if (*endp || *value > max)
2101                0 :                 return -EINVAL;
2102                :
2103                0 :         return 0;
2104                : }
2105                :
2106                : static ssize_t delayed_allocation_blocks_show(struct ext4_attr *a,
2107                :                                               struct ext4_sb_info *sbi,
2108                :                                               char *buf)
2109                0 : {
2110                0 :         return snprintf(buf, PAGE_SIZE, "%llu\n",
2111                :                         (s64) percpu_counter_sum(&sbi->s_dirtyblocks_counter));
2112                : }
2113                :
2114                : static ssize_t session_write_kbytes_show(struct ext4_attr *a,
2115                :                                          struct ext4_sb_info *sbi, char *buf)
2116                0 : {
2117                0 :         struct super_block *sb = sbi->s_buddy_cache->i_sb;
2118                :
2119                0 :         return snprintf(buf, PAGE_SIZE, "%lu\n",
2120                0 :                         (part_stat_read(sb->s_bdev->bd_part, sectors[1]) -
2121                :                          sbi->s_sectors_written_start) >> 1);
2122                : }
2123                :
2124                : static ssize_t lifetime_write_kbytes_show(struct ext4_attr *a,
2125                :                                           struct ext4_sb_info *sbi, char *buf)
2126                0 : {
2127                0 :         struct super_block *sb = sbi->s_buddy_cache->i_sb;
2128                :
2129                0 :         return snprintf(buf, PAGE_SIZE, "%llu\n",
2130                :                         sbi->s_kbytes_written +
2131                0 :                         ((part_stat_read(sb->s_bdev->bd_part, sectors[1]) -
2132                :                           EXT4_SB(sb)->s_sectors_written_start) >> 1));
2133                : }
2134                :
2135                : static ssize_t inode_readahead_blks_store(struct ext4_attr *a,
2136                :                                           struct ext4_sb_info *sbi,
2137                :                                           const char *buf, size_t count)
2138                0 : {
2139                :         unsigned long t;
2140                :
2141                0 :         if (parse_strtoul(buf, 0x40000000, &t))
2142                0 :                 return -EINVAL;
2143                :
2144                0 :         if (!is_power_of_2(t))
```

```
2145          0 :                 return -EINVAL;
2146            :
2147          0 :         sbi->s_inode_readahead_blks = t;
2148          0 :         return count;
2149            : }
2150            :
2151            : static ssize_t sbi_ui_show(struct ext4_attr *a,
2152            :                            struct ext4_sb_info *sbi, char *buf)
2153          0 : {
2154          0 :         unsigned int *ui = (unsigned int *) (((char *) sbi) + a->offset);
2155            :
2156          0 :         return snprintf(buf, PAGE_SIZE, "%u\n", *ui);
2157            : }
2158            :
2159            : static ssize_t sbi_ui_store(struct ext4_attr *a,
2160            :                             struct ext4_sb_info *sbi,
2161            :                             const char *buf, size_t count)
2162          0 : {
2163          0 :         unsigned int *ui = (unsigned int *) (((char *) sbi) + a->offset);
2164            :         unsigned long t;
2165            :
2166          0 :         if (parse_strtoul(buf, 0xffffffff, &t))
2167          0 :                 return -EINVAL;
2168          0 :         *ui = t;
2169          0 :         return count;
2170            : }
2171            :
2172            : #define EXT4_ATTR_OFFSET(_name,_mode,_show,_store,_elname) \
2173            : static struct ext4_attr ext4_attr_##_name = {                  \
2174            :         .attr = {.name = __stringify(_name), .mode = _mode },  \
2175            :         .show   = _show,                                       \
2176            :         .store  = _store,                                      \
2177            :         .offset = offsetof(struct ext4_sb_info, _elname),      \
2178            : }
2179            : #define EXT4_ATTR(name, mode, show, store) \
2180            : static struct ext4_attr ext4_attr_##name = __ATTR(name, mode, show, store)
2181            :
2182            : #define EXT4_RO_ATTR(name) EXT4_ATTR(name, 0444, name##_show, NULL)
2183            : #define EXT4_RW_ATTR(name) EXT4_ATTR(name, 0644, name##_show, name##_store)
2184            : #define EXT4_RW_ATTR_SBI_UI(name, elname)       \
2185            :         EXT4_ATTR_OFFSET(name, 0644, sbi_ui_show, sbi_ui_store, elname)
2186            : #define ATTR_LIST(name) &ext4_attr_##name.attr
2187            :
2188            : EXT4_RO_ATTR(delayed_allocation_blocks);
2189            : EXT4_RO_ATTR(session_write_kbytes);
2190            : EXT4_RO_ATTR(lifetime_write_kbytes);
2191            : EXT4_ATTR_OFFSET(inode_readahead_blks, 0644, sbi_ui_show,
2192            :                  inode_readahead_blks_store, s_inode_readahead_blks);
2193            : EXT4_RW_ATTR_SBI_UI(inode_goal, s_inode_goal);
2194            : EXT4_RW_ATTR_SBI_UI(mb_stats, s_mb_stats);
2195            : EXT4_RW_ATTR_SBI_UI(mb_max_to_scan, s_mb_max_to_scan);
2196            : EXT4_RW_ATTR_SBI_UI(mb_min_to_scan, s_mb_min_to_scan);
2197            : EXT4_RW_ATTR_SBI_UI(mb_order2_req, s_mb_order2_reqs);
2198            : EXT4_RW_ATTR_SBI_UI(mb_stream_req, s_mb_stream_request);
2199            : EXT4_RW_ATTR_SBI_UI(mb_group_prealloc, s_mb_group_prealloc);
2200            :
2201            : static struct attribute *ext4_attrs[] = {
2202            :         ATTR_LIST(delayed_allocation_blocks),
2203            :         ATTR_LIST(session_write_kbytes),
2204            :         ATTR_LIST(lifetime_write_kbytes),
2205            :         ATTR_LIST(inode_readahead_blks),
2206            :         ATTR_LIST(inode_goal),
2207            :         ATTR_LIST(mb_stats),
2208            :         ATTR_LIST(mb_max_to_scan),
2209            :         ATTR_LIST(mb_min_to_scan),
2210            :         ATTR_LIST(mb_order2_req),
2211            :         ATTR_LIST(mb_stream_req),
2212            :         ATTR_LIST(mb_group_prealloc),
2213            :         NULL,
2214            : };
2215            :
2216            : static ssize_t ext4_attr_show(struct kobject *kobj,
2217            :                               struct attribute *attr, char *buf)
2218          0 : {
2219          0 :         struct ext4_sb_info *sbi = container_of(kobj, struct ext4_sb_info,
2220            :                                                 s_kobj);
2221          0 :         struct ext4_attr *a = container_of(attr, struct ext4_attr, attr);
2222            :
2223          0 :         return a->show ? a->show(a, sbi, buf) : 0;
2224            : }
2225            :
2226            : static ssize_t ext4_attr_store(struct kobject *kobj,
2227            :                                struct attribute *attr,
2228            :                                const char *buf, size_t len)
2229          0 : {
2230          0 :         struct ext4_sb_info *sbi = container_of(kobj, struct ext4_sb_info,
2231            :                                                 s_kobj);
2232          0 :         struct ext4_attr *a = container_of(attr, struct ext4_attr, attr);
2233            :
2234          0 :         return a->store ? a->store(a, sbi, buf, len) : 0;
```

```
2235                 : }
2236                 :
2237                 : static void ext4_sb_release(struct kobject *kobj)
2238              94 : {
2239              94 :         struct ext4_sb_info *sbi = container_of(kobj, struct ext4_sb_info,
2240                 :                                                 s_kobj);
2241              94 :         complete(&sbi->s_kobj_unregister);
2242              94 : }
2243                 :
2244                 :
2245                 : static struct sysfs_ops ext4_attr_ops = {
2246                 :         .show   = ext4_attr_show,
2247                 :         .store  = ext4_attr_store,
2248                 : };
2249                 :
2250                 : static struct kobj_type ext4_ktype = {
2251                 :         .default_attrs  = ext4_attrs,
2252                 :         .sysfs_ops      = &ext4_attr_ops,
2253                 :         .release        = ext4_sb_release,
2254                 : };
2255                 :
2256                 : static int ext4_fill_super(struct super_block *sb, void *data, int silent)
2257                 :                                 __releases(kernel_lock)
2258                 :                                 __acquires(kernel_lock)
2259              94 : {
2260                 :         struct buffer_head *bh;
2261              94 :         struct ext4_super_block *es = NULL;
2262                 :         struct ext4_sb_info *sbi;
2263                 :         ext4_fsblk_t block;
2264              94 :         ext4_fsblk_t sb_block = get_sb_block(&data);
2265                 :         ext4_fsblk_t logical_sb_block;
2266              94 :         unsigned long offset = 0;
2267              94 :         unsigned long journal_devnum = 0;
2268                 :         unsigned long def_mount_opts;
2269                 :         struct inode *root;
2270                 :         char *cp;
2271                 :         const char *descr;
2272              94 :         int ret = -EINVAL;
2273                 :         int blocksize;
2274                 :         unsigned int db_count;
2275                 :         unsigned int i;
2276                 :         int needs_recovery, has_huge_files;
2277                 :         int features;
2278                 :         __u64 blocks_count;
2279                 :         int err;
2280              94 :         unsigned int journal_ioprio = DEFAULT_JOURNAL_IOPRIO;
2281                 :
2282              94 :         sbi = kzalloc(sizeof(*sbi), GFP_KERNEL);
2283              94 :         if (!sbi)
2284               0 :                 return -ENOMEM;
2285                 :
2286              94 :         sbi->s_blockgroup_lock =
2287                 :                 kzalloc(sizeof(struct blockgroup_lock), GFP_KERNEL);
2288              94 :         if (!sbi->s_blockgroup_lock) {
2289               0 :                 kfree(sbi);
2290               0 :                 return -ENOMEM;
2291                 :         }
2292              94 :         sb->s_fs_info = sbi;
2293              94 :         sbi->s_mount_opt = 0;
2294              94 :         sbi->s_resuid = EXT4_DEF_RESUID;
2295              94 :         sbi->s_resgid = EXT4_DEF_RESGID;
2296              94 :         sbi->s_inode_readahead_blks = EXT4_DEF_INODE_READAHEAD_BLKS;
2297              94 :         sbi->s_sb_block = sb_block;
2298             940 :         sbi->s_sectors_written_start = part_stat_read(sb->s_bdev->bd_part,
2299                 :                                                       sectors[1]);
2300                 :
2301              94 :         unlock_kernel();
2302                 :
2303                 :         /* Cleanup superblock name */
2304             188 :         for (cp = sb->s_id; (cp = strchr(cp, '/'));)
2305               0 :                 *cp = '!';
2306                 :
2307              94 :         blocksize = sb_min_blocksize(sb, EXT4_MIN_BLOCK_SIZE);
2308              94 :         if (!blocksize) {
2309               0 :                 ext4_msg(sb, KERN_ERR, "unable to set blocksize");
2310               0 :                 goto out_fail;
2311                 :         }
2312                 :
2313                 :         /*
2314                 :          * The ext4 superblock will not be buffer aligned for other than 1kB
2315                 :          * block sizes.  We need to calculate the offset from buffer start.
2316                 :          */
2317              94 :         if (blocksize != EXT4_MIN_BLOCK_SIZE) {
2318               0 :                 logical_sb_block = sb_block * EXT4_MIN_BLOCK_SIZE;
2319               0 :                 offset = do_div(logical_sb_block, blocksize);
2320                 :         } else {
2321              94 :                 logical_sb_block = sb_block;
2322                 :         }
2323                 :
2324              94 :         if (!(bh = sb_bread(sb, logical_sb_block))) {
```

```
2325         0 :                 ext4_msg(sb, KERN_ERR, "unable to read superblock");
2326         0 :                 goto out_fail;
2327           :         }
2328           :         /*
2329           :          * Note: s_es must be initialized as soon as possible because
2330           :          *       some ext4 macro-instructions depend on its value
2331           :          */
2332        94 :         es = (struct ext4_super_block *) (((char *)bh->b_data) + offset);
2333        94 :         sbi->s_es = es;
2334        94 :         sb->s_magic = le16_to_cpu(es->s_magic);
2335        94 :         if (sb->s_magic != EXT4_SUPER_MAGIC)
2336         0 :                 goto cantfind_ext4;
2337        94 :         sbi->s_kbytes_written = le64_to_cpu(es->s_kbytes_written);
2338           :
2339           :         /* Set defaults before we parse the mount options */
2340        94 :         def_mount_opts = le32_to_cpu(es->s_default_mount_opts);
2341        94 :         if (def_mount_opts & EXT4_DEFM_DEBUG)
2342         0 :                 set_opt(sbi->s_mount_opt, DEBUG);
2343        94 :         if (def_mount_opts & EXT4_DEFM_BSDGROUPS)
2344         0 :                 set_opt(sbi->s_mount_opt, GRPID);
2345        94 :         if (def_mount_opts & EXT4_DEFM_UID16)
2346         0 :                 set_opt(sbi->s_mount_opt, NO_UID32);
2347           : #ifdef CONFIG_EXT4_FS_XATTR
2348        94 :         if (def_mount_opts & EXT4_DEFM_XATTR_USER)
2349         0 :                 set_opt(sbi->s_mount_opt, XATTR_USER);
2350           : #endif
2351           : #ifdef CONFIG_EXT4_FS_POSIX_ACL
2352        94 :         if (def_mount_opts & EXT4_DEFM_ACL)
2353         0 :                 set_opt(sbi->s_mount_opt, POSIX_ACL);
2354           : #endif
2355        94 :         if ((def_mount_opts & EXT4_DEFM_JMODE) == EXT4_DEFM_JMODE_DATA)
2356         0 :                 sbi->s_mount_opt |= EXT4_MOUNT_JOURNAL_DATA;
2357        94 :         else if ((def_mount_opts & EXT4_DEFM_JMODE) == EXT4_DEFM_JMODE_ORDERED)
2358         0 :                 sbi->s_mount_opt |= EXT4_MOUNT_ORDERED_DATA;
2359        94 :         else if ((def_mount_opts & EXT4_DEFM_JMODE) == EXT4_DEFM_JMODE_WBACK)
2360         0 :                 sbi->s_mount_opt |= EXT4_MOUNT_WRITEBACK_DATA;
2361           :
2362        94 :         if (le16_to_cpu(sbi->s_es->s_errors) == EXT4_ERRORS_PANIC)
2363         0 :                 set_opt(sbi->s_mount_opt, ERRORS_PANIC);
2364        94 :         else if (le16_to_cpu(sbi->s_es->s_errors) == EXT4_ERRORS_CONTINUE)
2365        94 :                 set_opt(sbi->s_mount_opt, ERRORS_CONT);
2366           :         else
2367         0 :                 set_opt(sbi->s_mount_opt, ERRORS_RO);
2368           :
2369        94 :         sbi->s_resuid = le16_to_cpu(es->s_def_resuid);
2370        94 :         sbi->s_resgid = le16_to_cpu(es->s_def_resgid);
2371        94 :         sbi->s_commit_interval = JBD2_DEFAULT_MAX_COMMIT_AGE * HZ;
2372        94 :         sbi->s_min_batch_time = EXT4_DEF_MIN_BATCH_TIME;
2373        94 :         sbi->s_max_batch_time = EXT4_DEF_MAX_BATCH_TIME;
2374        94 :         sbi->s_mb_history_max = default_mb_history_length;
2375           :
2376        94 :         set_opt(sbi->s_mount_opt, BARRIER);
2377           :
2378           :         /*
2379           :          * enable delayed allocation by default
2380           :          * Use -o nodelalloc to turn it off
2381           :          */
2382        94 :         set_opt(sbi->s_mount_opt, DELALLOC);
2383           :
2384        94 :         if (!parse_options((char *) data, sb, &journal_devnum,
2385           :                            &journal_ioprio, NULL, 0))
2386         0 :                 goto failed_mount;
2387           :
2388        94 :         sb->s_flags = (sb->s_flags & ~MS_POSIXACL) |
2389           :                 ((sbi->s_mount_opt & EXT4_MOUNT_POSIX_ACL) ? MS_POSIXACL : 0);
2390           :
2391        94 :         if (le32_to_cpu(es->s_rev_level) == EXT4_GOOD_OLD_REV &&
2392           :             (EXT4_HAS_COMPAT_FEATURE(sb, ~0U) ||
2393           :              EXT4_HAS_RO_COMPAT_FEATURE(sb, ~0U) ||
2394           :              EXT4_HAS_INCOMPAT_FEATURE(sb, ~0U)))
2395         0 :                 ext4_msg(sb, KERN_WARNING,
2396           :                        "feature flags set on rev 0 fs, "
2397           :                        "running e2fsck is recommended");
2398           :
2399           :         /*
2400           :          * Check feature flags regardless of the revision level, since we
2401           :          * previously didn't change the revision level when setting the flags,
2402           :          * so there is a chance incompat flags are set on a rev 0 filesystem.
2403           :          */
2404        94 :         features = EXT4_HAS_INCOMPAT_FEATURE(sb, ~EXT4_FEATURE_INCOMPAT_SUPP);
2405        94 :         if (features) {
2406         0 :                 ext4_msg(sb, KERN_ERR,
2407           :                        "Couldn't mount because of "
2408           :                        "unsupported optional features (%x)",
2409           :                        (le32_to_cpu(EXT4_SB(sb)->s_es->s_feature_incompat) &
2410           :                        ~EXT4_FEATURE_INCOMPAT_SUPP));
2411         0 :                 goto failed_mount;
2412           :         }
2413        94 :         features = EXT4_HAS_RO_COMPAT_FEATURE(sb, ~EXT4_FEATURE_RO_COMPAT_SUPP);
2414        94 :         if (!(sb->s_flags & MS_RDONLY) && features) {
```

```
2415         0 :                 ext4_msg(sb, KERN_ERR,
2416           :                         "Couldn't mount RDWR because of "
2417           :                         "unsupported optional features (%x)",
2418           :                         (le32_to_cpu(EXT4_SB(sb)->s_es->s_feature_ro_compat) &
2419           :                         ~EXT4_FEATURE_RO_COMPAT_SUPP));
2420         0 :                 goto failed_mount;
2421           :         }
2422        94 :         has_huge_files = EXT4_HAS_RO_COMPAT_FEATURE(sb,
2423           :                                             EXT4_FEATURE_RO_COMPAT_HUGE_FILE);
2424           :         if (has_huge_files) {
2425           :                 /*
2426           :                  * Large file size enabled file system can only be
2427           :                  * mount if kernel is build with CONFIG_LBDAF
2428           :                  */
2429           :                 if (sizeof(root->i_blocks) < sizeof(u64) &&
2430           :                                 !(sb->s_flags & MS_RDONLY)) {
2431           :                         ext4_msg(sb, KERN_ERR, "Filesystem with huge "
2432           :                                         "files cannot be mounted read-write "
2433           :                                         "without CONFIG_LBDAF");
2434           :                         goto failed_mount;
2435           :                 }
2436           :         }
2437        94 :         blocksize = BLOCK_SIZE << le32_to_cpu(es->s_log_block_size);
2438           :
2439        94 :         if (blocksize < EXT4_MIN_BLOCK_SIZE ||
2440           :             blocksize > EXT4_MAX_BLOCK_SIZE) {
2441         0 :                 ext4_msg(sb, KERN_ERR,
2442           :                        "Unsupported filesystem blocksize %d", blocksize);
2443         0 :                 goto failed_mount;
2444           :         }
2445           :
2446        94 :         if (sb->s_blocksize != blocksize) {
2447           :                 /* Validate the filesystem blocksize */
2448        80 :                 if (!sb_set_blocksize(sb, blocksize)) {
2449         0 :                         ext4_msg(sb, KERN_ERR, "bad block size %d",
2450           :                                         blocksize);
2451         0 :                         goto failed_mount;
2452           :                 }
2453           :
2454           :                 brelse(bh);
2455        80 :                 logical_sb_block = sb_block * EXT4_MIN_BLOCK_SIZE;
2456        80 :                 offset = do_div(logical_sb_block, blocksize);
2457        80 :                 bh = sb_bread(sb, logical_sb_block);
2458        80 :                 if (!bh) {
2459         0 :                         ext4_msg(sb, KERN_ERR,
2460           :                                 "Can't read superblock on 2nd try");
2461         0 :                         goto failed_mount;
2462           :                 }
2463        80 :                 es = (struct ext4_super_block *)(((char *)bh->b_data) + offset);
2464        80 :                 sbi->s_es = es;
2465        80 :                 if (es->s_magic != cpu_to_le16(EXT4_SUPER_MAGIC)) {
2466         0 :                         ext4_msg(sb, KERN_ERR,
2467           :                                 "Magic mismatch, very weird!");
2468         0 :                         goto failed_mount;
2469           :                 }
2470           :         }
2471           :
2472       188 :         sbi->s_bitmap_maxbytes = ext4_max_bitmap_size(sb->s_blocksize_bits,
2473           :                                                       has_huge_files);
2474       188 :         sb->s_maxbytes = ext4_max_size(sb->s_blocksize_bits, has_huge_files);
2475           :
2476        94 :         if (le32_to_cpu(es->s_rev_level) == EXT4_GOOD_OLD_REV) {
2477         0 :                 sbi->s_inode_size = EXT4_GOOD_OLD_INODE_SIZE;
2478         0 :                 sbi->s_first_ino = EXT4_GOOD_OLD_FIRST_INO;
2479           :         } else {
2480        94 :                 sbi->s_inode_size = le16_to_cpu(es->s_inode_size);
2481        94 :                 sbi->s_first_ino = le32_to_cpu(es->s_first_ino);
2482       188 :                 if ((sbi->s_inode_size < EXT4_GOOD_OLD_INODE_SIZE) ||
2483           :                     (!is_power_of_2(sbi->s_inode_size)) ||
2484           :                     (sbi->s_inode_size > blocksize)) {
2485         0 :                         ext4_msg(sb, KERN_ERR,
2486           :                                 "unsupported inode size: %d",
2487           :                                 sbi->s_inode_size);
2488         0 :                         goto failed_mount;
2489           :                 }
2490        94 :                 if (sbi->s_inode_size > EXT4_GOOD_OLD_INODE_SIZE)
2491        92 :                         sb->s_time_gran = 1 << (EXT4_EPOCH_BITS - 2);
2492           :         }
2493           :
2494        94 :         sbi->s_desc_size = le16_to_cpu(es->s_desc_size);
2495        94 :         if (EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_64BIT)) {
2496         0 :                 if (sbi->s_desc_size < EXT4_MIN_DESC_SIZE_64BIT ||
2497           :                     sbi->s_desc_size > EXT4_MAX_DESC_SIZE ||
2498           :                     !is_power_of_2(sbi->s_desc_size)) {
2499         0 :                         ext4_msg(sb, KERN_ERR,
2500           :                                 "unsupported descriptor size %lu",
2501           :                                 sbi->s_desc_size);
2502         0 :                         goto failed_mount;
2503           :                 }
2504           :         } else
```

```
2505       94 :                 sbi->s_desc_size = EXT4_MIN_DESC_SIZE;
2506          :
2507       94 :         sbi->s_blocks_per_group = le32_to_cpu(es->s_blocks_per_group);
2508       94 :         sbi->s_inodes_per_group = le32_to_cpu(es->s_inodes_per_group);
2509      188 :         if (EXT4_INODE_SIZE(sb) == 0 || EXT4_INODES_PER_GROUP(sb) == 0)
2510          :                 goto cantfind_ext4;
2511          :
2512       94 :         sbi->s_inodes_per_block = blocksize / EXT4_INODE_SIZE(sb);
2513       94 :         if (sbi->s_inodes_per_block == 0)
2514        0 :                 goto cantfind_ext4;
2515       94 :         sbi->s_itb_per_group = sbi->s_inodes_per_group /
2516          :                                         sbi->s_inodes_per_block;
2517      188 :         sbi->s_desc_per_block = blocksize / EXT4_DESC_SIZE(sb);
2518       94 :         sbi->s_sbh = bh;
2519       94 :         sbi->s_mount_state = le16_to_cpu(es->s_state);
2520      188 :         sbi->s_addr_per_block_bits = ilog2(EXT4_ADDR_PER_BLOCK(sb));
2521      188 :         sbi->s_desc_per_block_bits = ilog2(EXT4_DESC_PER_BLOCK(sb));
2522          :
2523      470 :         for (i = 0; i < 4; i++)
2524      376 :                 sbi->s_hash_seed[i] = le32_to_cpu(es->s_hash_seed[i]);
2525       94 :         sbi->s_def_hash_version = es->s_def_hash_version;
2526       94 :         i = le32_to_cpu(es->s_flags);
2527       94 :         if (i & EXT2_FLAGS_UNSIGNED_HASH)
2528        0 :                 sbi->s_hash_unsigned = 3;
2529       94 :         else if ((i & EXT2_FLAGS_SIGNED_HASH) == 0) {
2530          : #ifdef __CHAR_UNSIGNED__
2531          :                 es->s_flags |= cpu_to_le32(EXT2_FLAGS_UNSIGNED_HASH);
2532          :                 sbi->s_hash_unsigned = 3;
2533          : #else
2534        0 :                 es->s_flags |= cpu_to_le32(EXT2_FLAGS_SIGNED_HASH);
2535          : #endif
2536        0 :                 sb->s_dirt = 1;
2537          :         }
2538          :
2539       94 :         if (sbi->s_blocks_per_group > blocksize * 8) {
2540        0 :                 ext4_msg(sb, KERN_ERR,
2541          :                        "#blocks per group too big: %lu",
2542          :                        sbi->s_blocks_per_group);
2543        0 :                 goto failed_mount;
2544          :         }
2545       94 :         if (sbi->s_inodes_per_group > blocksize * 8) {
2546        0 :                 ext4_msg(sb, KERN_ERR,
2547          :                        "#inodes per group too big: %lu",
2548          :                        sbi->s_inodes_per_group);
2549        0 :                 goto failed_mount;
2550          :         }
2551          :
2552       94 :         if (ext4_blocks_count(es) >
2553          :                     (sector_t)(~0ULL) >> (sb->s_blocksize_bits - 9)) {
2554        0 :                 ext4_msg(sb, KERN_ERR, "filesystem"
2555          :                        " too large to mount safely");
2556          :                 if (sizeof(sector_t) < 8)
2557          :                         ext4_msg(sb, KERN_WARNING, "CONFIG_LBDAF not enabled");
2558        0 :                 goto failed_mount;
2559          :         }
2560          :
2561       94 :         if (EXT4_BLOCKS_PER_GROUP(sb) == 0)
2562        0 :                 goto cantfind_ext4;
2563          :
2564          :         /* check blocks count against device size */
2565       94 :         blocks_count = sb->s_bdev->bd_inode->i_size >> sb->s_blocksize_bits;
2566      188 :         if (blocks_count && ext4_blocks_count(es) > blocks_count) {
2567        0 :                 ext4_msg(sb, KERN_WARNING, "bad geometry: block count %llu "
2568          :                        "exceeds size of device (%llu blocks)",
2569          :                        ext4_blocks_count(es), blocks_count);
2570        0 :                 goto failed_mount;
2571          :         }
2572          :
2573          :         /*
2574          :          * It makes no sense for the first data block to be beyond the end
2575          :          * of the filesystem.
2576          :          */
2577      188 :         if (le32_to_cpu(es->s_first_data_block) >= ext4_blocks_count(es)) {
2578        0 :                 ext4_msg(sb, KERN_WARNING, "bad geometry: first data"
2579          :                         "block %u is beyond end of filesystem (%llu)",
2580          :                         le32_to_cpu(es->s_first_data_block),
2581          :                         ext4_blocks_count(es));
2582        0 :                 goto failed_mount;
2583          :         }
2584      188 :         blocks_count = (ext4_blocks_count(es) -
2585          :                         le32_to_cpu(es->s_first_data_block) +
2586          :                         EXT4_BLOCKS_PER_GROUP(sb) - 1);
2587       94 :         do_div(blocks_count, EXT4_BLOCKS_PER_GROUP(sb));
2588       94 :         if (blocks_count > ((uint64_t)1<<32) - EXT4_DESC_PER_BLOCK(sb)) {
2589        0 :                 ext4_msg(sb, KERN_WARNING, "groups count too large: %u "
2590          :                        "(block count %llu, first data block %u, "
2591          :                        "blocks per group %lu)", sbi->s_groups_count,
2592          :                        ext4_blocks_count(es),
2593          :                        le32_to_cpu(es->s_first_data_block),
2594          :                        EXT4_BLOCKS_PER_GROUP(sb));
```

```
    2595          0 :                 goto failed_mount;
    2596            :         }
    2597         94 :         sbi->s_groups_count = blocks_count;
    2598        282 :         db_count = (sbi->s_groups_count + EXT4_DESC_PER_BLOCK(sb) - 1) /
    2599            :                    EXT4_DESC_PER_BLOCK(sb);
    2600        188 :         sbi->s_group_desc = kmalloc(db_count * sizeof(struct buffer_head *),
    2601            :                                     GFP_KERNEL);
    2602         94 :         if (sbi->s_group_desc == NULL) {
    2603          0 :                 ext4_msg(sb, KERN_ERR, "not enough memory");
    2604          0 :                 goto failed_mount;
    2605            :         }
    2606            :
    2607            : #ifdef CONFIG_PROC_FS
    2608         94 :         if (ext4_proc_root)
    2609         94 :                 sbi->s_proc = proc_mkdir(sb->s_id, ext4_proc_root);
    2610            : #endif
    2611            :
    2612         94 :         bgl_lock_init(sbi->s_blockgroup_lock);
    2613            :
    2614      42802 :         for (i = 0; i < db_count; i++) {
    2615      85416 :                 block = descriptor_loc(sb, logical_sb_block, i);
    2616      85416 :                 sbi->s_group_desc[i] = sb_bread(sb, block);
    2617      42708 :                 if (!sbi->s_group_desc[i]) {
    2618          0 :                         ext4_msg(sb, KERN_ERR,
    2619            :                                "can't read group descriptor %d", i);
    2620          0 :                         db_count = i;
    2621          0 :                         goto failed_mount2;
    2622            :                 }
    2623            :         }
    2624         94 :         if (!ext4_check_descriptors(sb)) {
    2625          0 :                 ext4_msg(sb, KERN_ERR, "group descriptors corrupted!");
    2626          0 :                 goto failed_mount2;
    2627            :         }
    2628         94 :         if (EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_FLEX_BG))
    2629         77 :                 if (!ext4_fill_flex_info(sb)) {
    2630          0 :                         ext4_msg(sb, KERN_ERR,
    2631            :                                "unable to initialize "
    2632            :                                "flex_bg meta info!");
    2633          0 :                         goto failed_mount2;
    2634            :                 }
    2635            :
    2636         94 :         sbi->s_gdb_count = db_count;
    2637         94 :         get_random_bytes(&sbi->s_next_generation, sizeof(u32));
    2638         94 :         spin_lock_init(&sbi->s_next_gen_lock);
    2639            :
    2640         94 :         err = percpu_counter_init(&sbi->s_freeblocks_counter,
    2641            :                                   ext4_count_free_blocks(sb));
    2642         94 :         if (!err) {
    2643         94 :                 err = percpu_counter_init(&sbi->s_freeinodes_counter,
    2644            :                                           ext4_count_free_inodes(sb));
    2645            :         }
    2646         94 :         if (!err) {
    2647         94 :                 err = percpu_counter_init(&sbi->s_dirs_counter,
    2648            :                                           ext4_count_dirs(sb));
    2649            :         }
    2650         94 :         if (!err) {
    2651         94 :                 err = percpu_counter_init(&sbi->s_dirtyblocks_counter, 0);
    2652            :         }
    2653         94 :         if (err) {
    2654          0 :                 ext4_msg(sb, KERN_ERR, "insufficient memory");
    2655          0 :                 goto failed_mount3;
    2656            :         }
    2657            :
    2658         94 :         sbi->s_stripe = ext4_get_stripe_size(sbi);
    2659            :
    2660            :         /*
    2661            :          * set up enough so that it can read an inode
    2662            :          */
    2663        282 :         if (!test_opt(sb, NOLOAD) &&
    2664            :             EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_HAS_JOURNAL))
    2665         94 :                 sb->s_op = &ext4_sops;
    2666            :         else
    2667          0 :                 sb->s_op = &ext4_nojournal_sops;
    2668         94 :         sb->s_export_op = &ext4_export_ops;
    2669         94 :         sb->s_xattr = ext4_xattr_handlers;
    2670            : #ifdef CONFIG_QUOTA
    2671         94 :         sb->s_qcop = &ext4_qctl_operations;
    2672         94 :         sb->dq_op = &ext4_quota_operations;
    2673            : #endif
    2674         94 :         INIT_LIST_HEAD(&sbi->s_orphan); /* unlinked but open files */
    2675         94 :         mutex_init(&sbi->s_orphan_lock);
    2676         94 :         mutex_init(&sbi->s_resize_lock);
    2677            :
    2678         94 :         sb->s_root = NULL;
    2679            :
    2680        188 :         needs_recovery = (es->s_last_orphan != 0 ||
    2681            :                           EXT4_HAS_INCOMPAT_FEATURE(sb,
    2682            :                                     EXT4_FEATURE_INCOMPAT_RECOVER));
    2683            :
    2684            :         /*
```

```
2685                :          * The first inode we look at is the journal inode.  Don't try
2686                :          * root first: it may be modified in the journal!
2687                :          */
2688              188 :         if (!test_opt(sb, NOLOAD) &&
2689                :             EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_HAS_JOURNAL)) {
2690               94 :                 if (ext4_load_journal(sb, es, journal_devnum))
2691                0 :                         goto failed_mount3;
2692              188 :                 if (!(sb->s_flags & MS_RDONLY) &&
2693                :                     EXT4_SB(sb)->s_journal->j_failed_commit) {
2694                0 :                         ext4_msg(sb, KERN_CRIT, "error: "
2695                :                                "ext4_fill_super: Journal transaction "
2696                :                                "%u is corrupt",
2697                :                                EXT4_SB(sb)->s_journal->j_failed_commit);
2698                0 :                         if (test_opt(sb, ERRORS_RO)) {
2699                0 :                                 ext4_msg(sb, KERN_CRIT,
2700                :                                          "Mounting filesystem read-only");
2701                0 :                                 sb->s_flags |= MS_RDONLY;
2702                0 :                                 EXT4_SB(sb)->s_mount_state |= EXT4_ERROR_FS;
2703                0 :                                 es->s_state |= cpu_to_le16(EXT4_ERROR_FS);
2704                :                         }
2705                0 :                         if (test_opt(sb, ERRORS_PANIC)) {
2706                0 :                                 EXT4_SB(sb)->s_mount_state |= EXT4_ERROR_FS;
2707                0 :                                 es->s_state |= cpu_to_le16(EXT4_ERROR_FS);
2708                0 :                                 ext4_commit_super(sb, 1);
2709                0 :                                 goto failed_mount4;
2710                :                         }
2711                :                 }
2712                0 :         } else if (test_opt(sb, NOLOAD) && !(sb->s_flags & MS_RDONLY) &&
2713                :                    EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER)) {
2714                0 :                 ext4_msg(sb, KERN_ERR, "required journal recovery "
2715                :                        "suppressed and not mounted read-only");
2716                0 :                 goto failed_mount4;
2717                :         } else {
2718                0 :                 clear_opt(sbi->s_mount_opt, DATA_FLAGS);
2719                0 :                 set_opt(sbi->s_mount_opt, WRITEBACK_DATA);
2720                0 :                 sbi->s_journal = NULL;
2721                0 :                 needs_recovery = 0;
2722                0 :                 goto no_journal;
2723                :         }
2724                :
2725               94 :         if (ext4_blocks_count(es) > 0xffffffffULL &&
2726                :             !jbd2_journal_set_features(EXT4_SB(sb)->s_journal, 0, 0,
2727                :                                        JBD2_FEATURE_INCOMPAT_64BIT)) {
2728                0 :                 ext4_msg(sb, KERN_ERR, "Failed to set 64-bit journal feature");
2729                0 :                 goto failed_mount4;
2730                :         }
2731                :
2732               94 :         if (test_opt(sb, JOURNAL_ASYNC_COMMIT)) {
2733               12 :                 jbd2_journal_set_features(sbi->s_journal,
2734                :                                 JBD2_FEATURE_COMPAT_CHECKSUM, 0,
2735                :                                 JBD2_FEATURE_INCOMPAT_ASYNC_COMMIT);
2736               82 :         } else if (test_opt(sb, JOURNAL_CHECKSUM)) {
2737               12 :                 jbd2_journal_set_features(sbi->s_journal,
2738                :                                 JBD2_FEATURE_COMPAT_CHECKSUM, 0, 0);
2739               12 :                 jbd2_journal_clear_features(sbi->s_journal, 0, 0,
2740                :                                 JBD2_FEATURE_INCOMPAT_ASYNC_COMMIT);
2741                :         } else {
2742               70 :                 jbd2_journal_clear_features(sbi->s_journal,
2743                :                                 JBD2_FEATURE_COMPAT_CHECKSUM, 0,
2744                :                                 JBD2_FEATURE_INCOMPAT_ASYNC_COMMIT);
2745                :         }
2746                :
2747                :         /* We have now updated the journal if required, so we can
2748                :          * validate the data journaling mode. */
2749               94 :         switch (test_opt(sb, DATA_FLAGS)) {
2750                :         case 0:
2751                :                 /* No mode set, assume a default based on the journal
2752                :                  * capabilities: ORDERED_DATA if the journal can
2753                :                  * cope, else JOURNAL_DATA
2754                :                  */
2755               58 :                 if (jbd2_journal_check_available_features
2756                :                     (sbi->s_journal, 0, 0, JBD2_FEATURE_INCOMPAT_REVOKE))
2757               58 :                         set_opt(sbi->s_mount_opt, ORDERED_DATA);
2758                :                 else
2759                0 :                         set_opt(sbi->s_mount_opt, JOURNAL_DATA);
2760                :                 break;
2761                :
2762                :         case EXT4_MOUNT_ORDERED_DATA:
2763                :         case EXT4_MOUNT_WRITEBACK_DATA:
2764               24 :                 if (!jbd2_journal_check_available_features
2765                :                     (sbi->s_journal, 0, 0, JBD2_FEATURE_INCOMPAT_REVOKE)) {
2766                0 :                         ext4_msg(sb, KERN_ERR, "Journal does not support "
2767                :                                "requested data journaling mode");
2768                0 :                         goto failed_mount4;
2769                :                 }
2770                :         default:
2771                :                 break;
2772                :         }
2773               94 :         set_task_ioprio(sbi->s_journal->j_task, journal_ioprio);
2774                :
```

```
2775          94 : no_journal:
2776             :
2777          94 :         if (test_opt(sb, NOBH)) {
2778           0 :                 if (!(test_opt(sb, DATA_FLAGS) == EXT4_MOUNT_WRITEBACK_DATA)) {
2779           0 :                         ext4_msg(sb, KERN_WARNING, "Ignoring nobh option - "
2780             :                                 "its supported only with writeback mode");
2781           0 :                         clear_opt(sbi->s_mount_opt, NOBH);
2782             :                 }
2783             :         }
2784             :         /*
2785             :          * The jbd2_journal_load will have done any necessary log recovery,
2786             :          * so we can safely mount the rest of the filesystem now.
2787             :          */
2788             :
2789          94 :         root = ext4_iget(sb, EXT4_ROOT_INO);
2790          94 :         if (IS_ERR(root)) {
2791           0 :                 ext4_msg(sb, KERN_ERR, "get root inode failed");
2792           0 :                 ret = PTR_ERR(root);
2793           0 :                 goto failed_mount4;
2794             :         }
2795          94 :         if (!S_ISDIR(root->i_mode) || !root->i_blocks || !root->i_size) {
2796           0 :                 iput(root);
2797           0 :                 ext4_msg(sb, KERN_ERR, "corrupt root inode, run e2fsck");
2798           0 :                 goto failed_mount4;
2799             :         }
2800          94 :         sb->s_root = d_alloc_root(root);
2801          94 :         if (!sb->s_root) {
2802           0 :                 ext4_msg(sb, KERN_ERR, "get root dentry failed");
2803           0 :                 iput(root);
2804           0 :                 ret = -ENOMEM;
2805           0 :                 goto failed_mount4;
2806             :         }
2807             :
2808          94 :         ext4_setup_super(sb, es, sb->s_flags & MS_RDONLY);
2809             :
2810             :         /* determine the minimum size of new large inodes, if present */
2811          94 :         if (sbi->s_inode_size > EXT4_GOOD_OLD_INODE_SIZE) {
2812          92 :                 sbi->s_want_extra_isize = sizeof(struct ext4_inode) -
2813             :                                                      EXT4_GOOD_OLD_INODE_SIZE;
2814          92 :                 if (EXT4_HAS_RO_COMPAT_FEATURE(sb,
2815             :                                                EXT4_FEATURE_RO_COMPAT_EXTRA_ISIZE)) {
2816          75 :                         if (sbi->s_want_extra_isize <
2817             :                             le16_to_cpu(es->s_want_extra_isize))
2818           0 :                                 sbi->s_want_extra_isize =
2819             :                                         le16_to_cpu(es->s_want_extra_isize);
2820          75 :                         if (sbi->s_want_extra_isize <
2821             :                             le16_to_cpu(es->s_min_extra_isize))
2822           0 :                                 sbi->s_want_extra_isize =
2823             :                                         le16_to_cpu(es->s_min_extra_isize);
2824             :                 }
2825             :         }
2826             :         /* Check if enough inode space is available */
2827          94 :         if (EXT4_GOOD_OLD_INODE_SIZE + sbi->s_want_extra_isize >
2828             :                                                         sbi->s_inode_size) {
2829           0 :                 sbi->s_want_extra_isize = sizeof(struct ext4_inode) -
2830             :                                                      EXT4_GOOD_OLD_INODE_SIZE;
2831           0 :                 ext4_msg(sb, KERN_INFO, "required extra inode space not"
2832             :                         "available");
2833             :         }
2834             :
2835          94 :         if (test_opt(sb, DATA_FLAGS) == EXT4_MOUNT_JOURNAL_DATA) {
2836          12 :                 ext4_msg(sb, KERN_WARNING, "Ignoring delalloc option - "
2837             :                         "requested data journaling mode");
2838          12 :                 clear_opt(sbi->s_mount_opt, DELALLOC);
2839          82 :         } else if (test_opt(sb, DELALLOC))
2840          71 :                 ext4_msg(sb, KERN_INFO, "delayed allocation enabled");
2841             :
2842          94 :         err = ext4_setup_system_zone(sb);
2843          94 :         if (err) {
2844           0 :                 ext4_msg(sb, KERN_ERR, "failed to initialize system "
2845             :                         "zone (%d)\n", err);
2846           0 :                 goto failed_mount4;
2847             :         }
2848             :
2849          94 :         ext4_ext_init(sb);
2850          94 :         err = ext4_mb_init(sb, needs_recovery);
2851          94 :         if (err) {
2852           0 :                 ext4_msg(sb, KERN_ERR, "failed to initalize mballoc (%d)",
2853             :                         err);
2854           0 :                 goto failed_mount4;
2855             :         }
2856             :
2857          94 :         sbi->s_kobj.kset = ext4_kset;
2858          94 :         init_completion(&sbi->s_kobj_unregister);
2859          94 :         err = kobject_init_and_add(&sbi->s_kobj, &ext4_ktype, NULL,
2860             :                                    "%s", sb->s_id);
2861          94 :         if (err) {
2862           0 :                 ext4_mb_release(sb);
2863           0 :                 ext4_ext_release(sb);
2864           0 :                 goto failed_mount4;
```

```
2865                :         };
2866                :
2867               94 :         EXT4_SB(sb)->s_mount_state |= EXT4_ORPHAN_FS;
2868               94 :         ext4_orphan_cleanup(sb, es);
2869               94 :         EXT4_SB(sb)->s_mount_state &= ~EXT4_ORPHAN_FS;
2870               94 :         if (needs_recovery) {
2871                0 :                 ext4_msg(sb, KERN_INFO, "recovery complete");
2872                0 :                 ext4_mark_recovery_complete(sb, es);
2873                :         }
2874               94 :         if (EXT4_SB(sb)->s_journal) {
2875               94 :                 if (test_opt(sb, DATA_FLAGS) == EXT4_MOUNT_JOURNAL_DATA)
2876               12 :                         descr = " journalled data mode";
2877               82 :                 else if (test_opt(sb, DATA_FLAGS) == EXT4_MOUNT_ORDERED_DATA)
2878               70 :                         descr = " ordered data mode";
2879                :                 else
2880               12 :                         descr = " writeback data mode";
2881                :         } else
2882                0 :                 descr = "out journal";
2883                :
2884               94 :         ext4_msg(sb, KERN_INFO, "mounted filesystem with%s", descr);
2885                :
2886               94 :         lock_kernel();
2887               94 :         return 0;
2888                :
2889                0 : cantfind_ext4:
2890                0 :         if (!silent)
2891                0 :                 ext4_msg(sb, KERN_ERR, "VFS: Can't find ext4 filesystem");
2892                :         goto failed_mount;
2893                :
2894                0 : failed_mount4:
2895                0 :         ext4_msg(sb, KERN_ERR, "mount failed");
2896                0 :         ext4_release_system_zone(sb);
2897                0 :         if (sbi->s_journal) {
2898                0 :                 jbd2_journal_destroy(sbi->s_journal);
2899                0 :                 sbi->s_journal = NULL;
2900                :         }
2901                0 : failed_mount3:
2902                0 :         if (sbi->s_flex_groups) {
2903                0 :                 if (is_vmalloc_addr(sbi->s_flex_groups))
2904                0 :                         vfree(sbi->s_flex_groups);
2905                :                 else
2906                0 :                         kfree(sbi->s_flex_groups);
2907                :         }
2908                0 :         percpu_counter_destroy(&sbi->s_freeblocks_counter);
2909                0 :         percpu_counter_destroy(&sbi->s_freeinodes_counter);
2910                0 :         percpu_counter_destroy(&sbi->s_dirs_counter);
2911                0 :         percpu_counter_destroy(&sbi->s_dirtyblocks_counter);
2912                0 : failed_mount2:
2913                0 :         for (i = 0; i < db_count; i++)
2914                0 :                 brelse(sbi->s_group_desc[i]);
2915                0 :         kfree(sbi->s_group_desc);
2916                0 : failed_mount:
2917                0 :         if (sbi->s_proc) {
2918                0 :                 remove_proc_entry(sb->s_id, ext4_proc_root);
2919                :         }
2920                : #ifdef CONFIG_QUOTA
2921                0 :         for (i = 0; i < MAXQUOTAS; i++)
2922                0 :                 kfree(sbi->s_qf_names[i]);
2923                : #endif
2924                0 :         ext4_blkdev_remove(sbi);
2925                :         brelse(bh);
2926                0 : out_fail:
2927                0 :         sb->s_fs_info = NULL;
2928                0 :         kfree(sbi->s_blockgroup_lock);
2929                0 :         kfree(sbi);
2930                0 :         lock_kernel();
2931                0 :         return ret;
2932                : }
2933                :
2934                : /*
2935                :  * Setup any per-fs journal parameters now.  We'll do this both on
2936                :  * initial mount, once the journal has been initialised but before we've
2937                :  * done any recovery; and again on any subsequent remount.
2938                :  */
2939                : static void ext4_init_journal_params(struct super_block *sb, journal_t *journal)
2940               94 : {
2941               94 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2942                :
2943               94 :         journal->j_commit_interval = sbi->s_commit_interval;
2944               94 :         journal->j_min_batch_time = sbi->s_min_batch_time;
2945               94 :         journal->j_max_batch_time = sbi->s_max_batch_time;
2946                :
2947               94 :         spin_lock(&journal->j_state_lock);
2948               94 :         if (test_opt(sb, BARRIER))
2949               76 :                 journal->j_flags |= JBD2_BARRIER;
2950                :         else
2951               18 :                 journal->j_flags &= ~JBD2_BARRIER;
2952               94 :         if (test_opt(sb, DATA_ERR_ABORT))
2953                0 :                 journal->j_flags |= JBD2_ABORT_ON_SYNCDATA_ERR;
2954                :         else
```

```
2955       94 :                 journal->j_flags &= ~JBD2_ABORT_ON_SYNCDATA_ERR;
2956       94 :         spin_unlock(&journal->j_state_lock);
2957       94 : }
2958          :
2959          : static journal_t *ext4_get_journal(struct super_block *sb,
2960          :                                    unsigned int journal_inum)
2961       94 : {
2962          :         struct inode *journal_inode;
2963          :         journal_t *journal;
2964          :
2965       94 :         BUG_ON(!EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_HAS_JOURNAL));
2966          :
2967          :         /* First, test for the existence of a valid inode on disk.  Bad
2968          :          * things happen if we iget() an unused inode, as the subsequent
2969          :          * iput() will try to delete it. */
2970          :
2971       94 :         journal_inode = ext4_iget(sb, journal_inum);
2972       94 :         if (IS_ERR(journal_inode)) {
2973        0 :                 ext4_msg(sb, KERN_ERR, "no journal found");
2974        0 :                 return NULL;
2975          :         }
2976       94 :         if (!journal_inode->i_nlink) {
2977        0 :                 make_bad_inode(journal_inode);
2978        0 :                 iput(journal_inode);
2979        0 :                 ext4_msg(sb, KERN_ERR, "journal inode is deleted");
2980        0 :                 return NULL;
2981          :         }
2982          :
2983       94 :         jbd_debug(2, "Journal inode found at %p: %lld bytes\n",
2984          :                   journal_inode, journal_inode->i_size);
2985       94 :         if (!S_ISREG(journal_inode->i_mode)) {
2986        0 :                 ext4_msg(sb, KERN_ERR, "invalid journal inode");
2987        0 :                 iput(journal_inode);
2988        0 :                 return NULL;
2989          :         }
2990          :
2991       94 :         journal = jbd2_journal_init_inode(journal_inode);
2992       94 :         if (!journal) {
2993        0 :                 ext4_msg(sb, KERN_ERR, "Could not load journal inode");
2994        0 :                 iput(journal_inode);
2995        0 :                 return NULL;
2996          :         }
2997       94 :         journal->j_private = sb;
2998       94 :         ext4_init_journal_params(sb, journal);
2999       94 :         return journal;
3000          : }
3001          :
3002          : static journal_t *ext4_get_dev_journal(struct super_block *sb,
3003          :                                        dev_t j_dev)
3004        0 : {
3005          :         struct buffer_head *bh;
3006          :         journal_t *journal;
3007          :         ext4_fsblk_t start;
3008          :         ext4_fsblk_t len;
3009          :         int hblock, blocksize;
3010          :         ext4_fsblk_t sb_block;
3011          :         unsigned long offset;
3012          :         struct ext4_super_block *es;
3013          :         struct block_device *bdev;
3014          :
3015        0 :         BUG_ON(!EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_HAS_JOURNAL));
3016          :
3017        0 :         bdev = ext4_blkdev_get(j_dev, sb);
3018        0 :         if (bdev == NULL)
3019        0 :                 return NULL;
3020          :
3021        0 :         if (bd_claim(bdev, sb)) {
3022        0 :                 ext4_msg(sb, KERN_ERR,
3023          :                         "failed to claim external journal device");
3024        0 :                 blkdev_put(bdev, FMODE_READ|FMODE_WRITE);
3025        0 :                 return NULL;
3026          :         }
3027          :
3028        0 :         blocksize = sb->s_blocksize;
3029        0 :         hblock = bdev_logical_block_size(bdev);
3030        0 :         if (blocksize < hblock) {
3031        0 :                 ext4_msg(sb, KERN_ERR,
3032          :                         "blocksize too small for journal device");
3033        0 :                 goto out_bdev;
3034          :         }
3035          :
3036        0 :         sb_block = EXT4_MIN_BLOCK_SIZE / blocksize;
3037        0 :         offset = EXT4_MIN_BLOCK_SIZE % blocksize;
3038        0 :         set_blocksize(bdev, blocksize);
3039        0 :         if (!(bh = __bread(bdev, sb_block, blocksize))) {
3040        0 :                 ext4_msg(sb, KERN_ERR, "couldn't read superblock of "
3041          :                         "external journal");
3042        0 :                 goto out_bdev;
3043          :         }
3044          :
```

```
3045         0 :          es = (struct ext4_super_block *) (((char *)bh->b_data) + offset);
3046         0 :          if ((le16_to_cpu(es->s_magic) != EXT4_SUPER_MAGIC) ||
3047           :              !(le32_to_cpu(es->s_feature_incompat) &
3048           :                EXT4_FEATURE_INCOMPAT_JOURNAL_DEV)) {
3049         0 :                  ext4_msg(sb, KERN_ERR, "external journal has "
3050           :                                         "bad superblock");
3051           :                  brelse(bh);
3052           :                  goto out_bdev;
3053           :          }
3054           :
3055         0 :          if (memcmp(EXT4_SB(sb)->s_es->s_journal_uuid, es->s_uuid, 16)) {
3056         0 :                  ext4_msg(sb, KERN_ERR, "journal UUID does not match");
3057           :                  brelse(bh);
3058           :                  goto out_bdev;
3059           :          }
3060           :
3061         0 :          len = ext4_blocks_count(es);
3062         0 :          start = sb_block + 1;
3063           :          brelse(bh);     /* we're done with the superblock */
3064           :
3065         0 :          journal = jbd2_journal_init_dev(bdev, sb->s_bdev,
3066           :                                          start, len, blocksize);
3067         0 :          if (!journal) {
3068         0 :                  ext4_msg(sb, KERN_ERR, "failed to create device journal");
3069         0 :                  goto out_bdev;
3070           :          }
3071         0 :          journal->j_private = sb;
3072         0 :          ll_rw_block(READ, 1, &journal->j_sb_buffer);
3073         0 :          wait_on_buffer(journal->j_sb_buffer);
3074         0 :          if (!buffer_uptodate(journal->j_sb_buffer)) {
3075         0 :                  ext4_msg(sb, KERN_ERR, "I/O error on journal device");
3076         0 :                  goto out_journal;
3077           :          }
3078         0 :          if (be32_to_cpu(journal->j_superblock->s_nr_users) != 1) {
3079         0 :                  ext4_msg(sb, KERN_ERR, "External journal has more than one "
3080           :                                         "user (unsupported) - %d",
3081           :                           be32_to_cpu(journal->j_superblock->s_nr_users));
3082         0 :                  goto out_journal;
3083           :          }
3084         0 :          EXT4_SB(sb)->journal_bdev = bdev;
3085         0 :          ext4_init_journal_params(sb, journal);
3086         0 :          return journal;
3087           :
3088         0 : out_journal:
3089         0 :          jbd2_journal_destroy(journal);
3090         0 : out_bdev:
3091           :          ext4_blkdev_put(bdev);
3092         0 :          return NULL;
3093           : }
3094           :
3095           : static int ext4_load_journal(struct super_block *sb,
3096           :                              struct ext4_super_block *es,
3097           :                              unsigned long journal_devnum)
3098        94 : {
3099           :          journal_t *journal;
3100        94 :          unsigned int journal_inum = le32_to_cpu(es->s_journal_inum);
3101           :          dev_t journal_dev;
3102        94 :          int err = 0;
3103           :          int really_read_only;
3104           :
3105        94 :          BUG_ON(!EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_HAS_JOURNAL));
3106           :
3107        94 :          if (journal_devnum &&
3108           :              journal_devnum != le32_to_cpu(es->s_journal_dev)) {
3109         0 :                  ext4_msg(sb, KERN_INFO, "external journal device major/minor "
3110           :                                  "numbers have changed");
3111         0 :                  journal_dev = new_decode_dev(journal_devnum);
3112           :          } else
3113       188 :                  journal_dev = new_decode_dev(le32_to_cpu(es->s_journal_dev));
3114           :
3115        94 :          really_read_only = bdev_read_only(sb->s_bdev);
3116           :
3117           :          /*
3118           :           * Are we loading a blank journal or performing recovery after a
3119           :           * crash?  For recovery, we need to check in advance whether we
3120           :           * can get read-write access to the device.
3121           :           */
3122        94 :          if (EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER)) {
3123         0 :                  if (sb->s_flags & MS_RDONLY) {
3124         0 :                          ext4_msg(sb, KERN_INFO, "INFO: recovery "
3125           :                                          "required on readonly filesystem");
3126         0 :                          if (really_read_only) {
3127         0 :                                  ext4_msg(sb, KERN_ERR, "write access "
3128           :                                                  "unavailable, cannot proceed");
3129         0 :                                  return -EROFS;
3130           :                          }
3131         0 :                          ext4_msg(sb, KERN_INFO, "write access will "
3132           :                                          "be enabled during recovery");
3133           :                  }
3134           :          }
```

```
3135                :
3136               94 :          if (journal_inum && journal_dev) {
3137                0 :                  ext4_msg(sb, KERN_ERR, "filesystem has both journal "
3138                :                         "and inode journals!");
3139                0 :                  return -EINVAL;
3140                :          }
3141                :
3142               94 :          if (journal_inum) {
3143               94 :                  if (!(journal = ext4_get_journal(sb, journal_inum)))
3144                0 :                          return -EINVAL;
3145                :          } else {
3146                0 :                  if (!(journal = ext4_get_dev_journal(sb, journal_dev)))
3147                0 :                          return -EINVAL;
3148                :          }
3149                :
3150               94 :          if (journal->j_flags & JBD2_BARRIER)
3151               76 :                  ext4_msg(sb, KERN_INFO, "barriers enabled");
3152                :          else
3153               18 :                  ext4_msg(sb, KERN_INFO, "barriers disabled");
3154                :
3155              188 :          if (!really_read_only && test_opt(sb, UPDATE_JOURNAL)) {
3156                0 :                  err = jbd2_journal_update_format(journal);
3157                0 :                  if (err)  {
3158                0 :                          ext4_msg(sb, KERN_ERR, "error updating journal");
3159                0 :                          jbd2_journal_destroy(journal);
3160                0 :                          return err;
3161                :                  }
3162                :          }
3163                :
3164               94 :          if (!EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER))
3165               94 :                  err = jbd2_journal_wipe(journal, !really_read_only);
3166               94 :          if (!err)
3167               94 :                  err = jbd2_journal_load(journal);
3168                :
3169               94 :          if (err) {
3170                0 :                  ext4_msg(sb, KERN_ERR, "error loading journal");
3171                0 :                  jbd2_journal_destroy(journal);
3172                0 :                  return err;
3173                :          }
3174                :
3175               94 :          EXT4_SB(sb)->s_journal = journal;
3176               94 :          ext4_clear_journal_err(sb, es);
3177                :
3178               94 :          if (journal_devnum &&
3179                :              journal_devnum != le32_to_cpu(es->s_journal_dev)) {
3180                0 :                  es->s_journal_dev = cpu_to_le32(journal_devnum);
3181                :
3182                :                  /* Make sure we flush the recovery flag to disk. */
3183                0 :                  ext4_commit_super(sb, 1);
3184                :          }
3185                :
3186               94 :          return 0;
3187                : }
3188                :
3189                : static int ext4_commit_super(struct super_block *sb, int sync)
3190              274 : {
3191              274 :          struct ext4_super_block *es = EXT4_SB(sb)->s_es;
3192              274 :          struct buffer_head *sbh = EXT4_SB(sb)->s_sbh;
3193              274 :          int error = 0;
3194                :
3195              274 :          if (!sbh)
3196                0 :                  return error;
3197              274 :          if (buffer_write_io_error(sbh)) {
3198                :                  /*
3199                :                   * Oh, dear.  A previous attempt to write the
3200                :                   * superblock failed.  This could happen because the
3201                :                   * USB device was yanked out.  Or it could happen to
3202                :                   * be a transient write error and maybe the block will
3203                :                   * be remapped.  Nothing we can do but to retry the
3204                :                   * write and hope for the best.
3205                :                   */
3206                0 :                  ext4_msg(sb, KERN_ERR, "previous I/O error to "
3207                :                         "superblock detected");
3208                :                  clear_buffer_write_io_error(sbh);
3209                :                  set_buffer_uptodate(sbh);
3210                :          }
3211              274 :          es->s_wtime = cpu_to_le32(get_seconds());
3212              548 :          es->s_kbytes_written =
3213             2740 :                  cpu_to_le64(EXT4_SB(sb)->s_kbytes_written +
3214                :                              ((part_stat_read(sb->s_bdev->bd_part, sectors[1]) -
3215                :                                EXT4_SB(sb)->s_sectors_written_start) >> 1));
3216              548 :          ext4_free_blocks_count_set(es, percpu_counter_sum_positive(
3217                :                                     &EXT4_SB(sb)->s_freeblocks_counter));
3218              548 :          es->s_free_inodes_count = cpu_to_le32(percpu_counter_sum_positive(
3219                :                                     &EXT4_SB(sb)->s_freeinodes_counter));
3220              274 :          sb->s_dirt = 0;
3221                :          BUFFER_TRACE(sbh, "marking dirty");
3222              274 :          mark_buffer_dirty(sbh);
3223              274 :          if (sync) {
3224              274 :                  error = sync_dirty_buffer(sbh);
```

```
3225              274 :                   if (error)
3226                0 :                           return error;
3227                  :
3228              274 :                   error = buffer_write_io_error(sbh);
3229              274 :                   if (error) {
3230                0 :                           ext4_msg(sb, KERN_ERR, "I/O error while writing "
3231                  :                                   "superblock");
3232                  :                           clear_buffer_write_io_error(sbh);
3233                  :                           set_buffer_uptodate(sbh);
3234                  :                   }
3235                  :           }
3236              274 :           return error;
3237                  : }
3238                  :
3239                  : /*
3240                  :  * Have we just finished recovery?  If so, and if we are mounting (or
3241                  :  * remounting) the filesystem readonly, then we will end up with a
3242                  :  * consistent fs on disk.  Record that fact.
3243                  :  */
3244                  : static void ext4_mark_recovery_complete(struct super_block *sb,
3245                  :                                         struct ext4_super_block *es)
3246                0 : {
3247                0 :           journal_t *journal = EXT4_SB(sb)->s_journal;
3248                  :
3249                0 :           if (!EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_HAS_JOURNAL)) {
3250                0 :                   BUG_ON(journal != NULL);
3251                  :                   return;
3252                  :           }
3253                0 :           jbd2_journal_lock_updates(journal);
3254                0 :           if (jbd2_journal_flush(journal) < 0)
3255                0 :                   goto out;
3256                  :
3257                0 :           if (EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER) &&
3258                  :               sb->s_flags & MS_RDONLY) {
3259                0 :                   EXT4_CLEAR_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER);
3260                0 :                   ext4_commit_super(sb, 1);
3261                  :           }
3262                  :
3263                0 : out:
3264                0 :           jbd2_journal_unlock_updates(journal);
3265                  : }
3266                  :
3267                  : /*
3268                  :  * If we are mounting (or read-write remounting) a filesystem whose journal
3269                  :  * has recorded an error from a previous lifetime, move that error to the
3270                  :  * main filesystem now.
3271                  :  */
3272                  : static void ext4_clear_journal_err(struct super_block *sb,
3273                  :                                    struct ext4_super_block *es)
3274               94 : {
3275                  :           journal_t *journal;
3276                  :           int j_errno;
3277                  :           const char *errstr;
3278                  :
3279               94 :           BUG_ON(!EXT4_HAS_COMPAT_FEATURE(sb, EXT4_FEATURE_COMPAT_HAS_JOURNAL));
3280                  :
3281               94 :           journal = EXT4_SB(sb)->s_journal;
3282                  :
3283                  :           /*
3284                  :            * Now check for any error status which may have been recorded in the
3285                  :            * journal by a prior ext4_error() or ext4_abort()
3286                  :            */
3287                  :
3288               94 :           j_errno = jbd2_journal_errno(journal);
3289               94 :           if (j_errno) {
3290                  :                   char nbuf[16];
3291                  :
3292                0 :                   errstr = ext4_decode_error(sb, j_errno, nbuf);
3293                0 :                   ext4_warning(sb, __func__, "Filesystem error recorded "
3294                  :                                "from previous mount: %s", errstr);
3295                0 :                   ext4_warning(sb, __func__, "Marking fs in need of "
3296                  :                                "filesystem check.");
3297                  :
3298                0 :                   EXT4_SB(sb)->s_mount_state |= EXT4_ERROR_FS;
3299                0 :                   es->s_state |= cpu_to_le16(EXT4_ERROR_FS);
3300                0 :                   ext4_commit_super(sb, 1);
3301                  :
3302                0 :                   jbd2_journal_clear_err(journal);
3303                  :           }
3304               94 : }
3305                  :
3306                  : /*
3307                  :  * Force the running and committing transactions to commit,
3308                  :  * and wait on the commit.
3309                  :  */
3310                  : int ext4_force_commit(struct super_block *sb)
3311            13486 : {
3312                  :           journal_t *journal;
3313            13486 :           int ret = 0;
3314                  :
```

```
3315          13486 :          if (sb->s_flags & MS_RDONLY)
3316              0 :                  return 0;
3317                :
3318          13486 :          journal = EXT4_SB(sb)->s_journal;
3319          13486 :          if (journal)
3320          13486 :                  ret = ext4_journal_force_commit(journal);
3321                :
3322          13486 :          return ret;
3323                : }
3324                :
3325                : static void ext4_write_super(struct super_block *sb)
3326              0 : {
3327              0 :          lock_super(sb);
3328              0 :          ext4_commit_super(sb, 1);
3329              0 :          unlock_super(sb);
3330              0 : }
3331                :
3332                : static int ext4_sync_fs(struct super_block *sb, int wait)
3333            657 : {
3334            657 :          int ret = 0;
3335                :          tid_t target;
3336                :
3337                :          trace_ext4_sync_fs(sb, wait);
3338            657 :          if (jbd2_journal_start_commit(EXT4_SB(sb)->s_journal, &target)) {
3339            255 :                  if (wait)
3340            210 :                          jbd2_log_wait_commit(EXT4_SB(sb)->s_journal, target);
3341                :          }
3342            657 :          return ret;
3343                : }
3344                :
3345                : /*
3346                :  * LVM calls this function before a (read-only) snapshot is created.  This
3347                :  * gives us a chance to flush the journal completely and mark the fs clean.
3348                :  */
3349                : static int ext4_freeze(struct super_block *sb)
3350              0 : {
3351              0 :          int error = 0;
3352                :          journal_t *journal;
3353                :
3354              0 :          if (sb->s_flags & MS_RDONLY)
3355              0 :                  return 0;
3356                :
3357              0 :          journal = EXT4_SB(sb)->s_journal;
3358                :
3359                :          /* Now we set up the journal barrier. */
3360              0 :          jbd2_journal_lock_updates(journal);
3361                :
3362                :          /*
3363                :           * Don't clear the needs_recovery flag if we failed to flush
3364                :           * the journal.
3365                :           */
3366              0 :          error = jbd2_journal_flush(journal);
3367              0 :          if (error < 0) {
3368              0 :          out:
3369              0 :                  jbd2_journal_unlock_updates(journal);
3370              0 :                  return error;
3371                :          }
3372                :
3373                :          /* Journal blocked and flushed, clear needs_recovery flag. */
3374              0 :          EXT4_CLEAR_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER);
3375              0 :          error = ext4_commit_super(sb, 1);
3376              0 :          if (error)
3377              0 :                  goto out;
3378              0 :          return 0;
3379                : }
3380                :
3381                : /*
3382                :  * Called by LVM after the snapshot is done.  We need to reset the RECOVER
3383                :  * flag here, even though the filesystem is not technically dirty yet.
3384                :  */
3385                : static int ext4_unfreeze(struct super_block *sb)
3386              0 : {
3387              0 :          if (sb->s_flags & MS_RDONLY)
3388              0 :                  return 0;
3389                :
3390              0 :          lock_super(sb);
3391                :          /* Reset the needs_recovery flag before the fs is unlocked. */
3392              0 :          EXT4_SET_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_RECOVER);
3393              0 :          ext4_commit_super(sb, 1);
3394              0 :          unlock_super(sb);
3395              0 :          jbd2_journal_unlock_updates(EXT4_SB(sb)->s_journal);
3396              0 :          return 0;
3397                : }
3398                :
3399                : static int ext4_remount(struct super_block *sb, int *flags, char *data)
3400              0 : {
3401                :          struct ext4_super_block *es;
3402              0 :          struct ext4_sb_info *sbi = EXT4_SB(sb);
3403              0 :          ext4_fsblk_t n_blocks_count = 0;
3404                :          unsigned long old_sb_flags;
```

```
3405                :         struct ext4_mount_options old_opts;
3406                :         ext4_group_t g;
3407              0 :         unsigned int journal_ioprio = DEFAULT_JOURNAL_IOPRIO;
3408                :         int err;
3409                : #ifdef CONFIG_QUOTA
3410                :         int i;
3411                : #endif
3412                :
3413              0 :         lock_kernel();
3414                :
3415                :         /* Store the original options */
3416              0 :         lock_super(sb);
3417              0 :         old_sb_flags = sb->s_flags;
3418              0 :         old_opts.s_mount_opt = sbi->s_mount_opt;
3419              0 :         old_opts.s_resuid = sbi->s_resuid;
3420              0 :         old_opts.s_resgid = sbi->s_resgid;
3421              0 :         old_opts.s_commit_interval = sbi->s_commit_interval;
3422              0 :         old_opts.s_min_batch_time = sbi->s_min_batch_time;
3423              0 :         old_opts.s_max_batch_time = sbi->s_max_batch_time;
3424                : #ifdef CONFIG_QUOTA
3425              0 :         old_opts.s_jquota_fmt = sbi->s_jquota_fmt;
3426              0 :         for (i = 0; i < MAXQUOTAS; i++)
3427              0 :                 old_opts.s_qf_names[i] = sbi->s_qf_names[i];
3428                : #endif
3429              0 :         if (sbi->s_journal && sbi->s_journal->j_task->io_context)
3430              0 :                 journal_ioprio = sbi->s_journal->j_task->io_context->ioprio;
3431                :
3432                :         /*
3433                :          * Allow the "check" option to be passed as a remount option.
3434                :          */
3435              0 :         if (!parse_options(data, sb, NULL, &journal_ioprio,
3436                :                            &n_blocks_count, 1)) {
3437              0 :                 err = -EINVAL;
3438              0 :                 goto restore_opts;
3439                :         }
3440                :
3441              0 :         if (sbi->s_mount_flags & EXT4_MF_FS_ABORTED)
3442              0 :                 ext4_abort(sb, __func__, "Abort forced by user");
3443                :
3444              0 :         sb->s_flags = (sb->s_flags & ~MS_POSIXACL) |
3445                :                 ((sbi->s_mount_opt & EXT4_MOUNT_POSIX_ACL) ? MS_POSIXACL : 0);
3446                :
3447              0 :         es = sbi->s_es;
3448                :
3449              0 :         if (sbi->s_journal) {
3450              0 :                 ext4_init_journal_params(sb, sbi->s_journal);
3451              0 :                 set_task_ioprio(sbi->s_journal->j_task, journal_ioprio);
3452                :         }
3453                :
3454              0 :         if ((*flags & MS_RDONLY) != (sb->s_flags & MS_RDONLY) ||
3455                :                 n_blocks_count > ext4_blocks_count(es)) {
3456              0 :                 if (sbi->s_mount_flags & EXT4_MF_FS_ABORTED) {
3457              0 :                         err = -EROFS;
3458              0 :                         goto restore_opts;
3459                :                 }
3460                :
3461              0 :                 if (*flags & MS_RDONLY) {
3462                :                         /*
3463                :                          * First of all, the unconditional stuff we have to do
3464                :                          * to disable replay of the journal when we next remount
3465                :                          */
3466              0 :                         sb->s_flags |= MS_RDONLY;
3467                :
3468                :                         /*
3469                :                          * OK, test if we are remounting a valid rw partition
3470                :                          * readonly, and if so set the rdonly flag and then
3471                :                          * mark the partition as valid again.
3472                :                          */
3473              0 :                         if (!(es->s_state & cpu_to_le16(EXT4_VALID_FS)) &&
3474                :                             (sbi->s_mount_state & EXT4_VALID_FS))
3475              0 :                                 es->s_state = cpu_to_le16(sbi->s_mount_state);
3476                :
3477              0 :                         if (sbi->s_journal)
3478              0 :                                 ext4_mark_recovery_complete(sb, es);
3479                :                 } else {
3480                :                         int ret;
3481              0 :                         if ((ret = EXT4_HAS_RO_COMPAT_FEATURE(sb,
3482                :                                     ~EXT4_FEATURE_RO_COMPAT_SUPP))) {
3483              0 :                                 ext4_msg(sb, KERN_WARNING, "couldn't "
3484                :                                        "remount RDWR because of unsupported "
3485                :                                        "optional features (%x)",
3486                :                                       (le32_to_cpu(sbi->s_es->s_feature_ro_compat) &
3487                :                                        ~EXT4_FEATURE_RO_COMPAT_SUPP));
3488              0 :                                 err = -EROFS;
3489              0 :                                 goto restore_opts;
3490                :                         }
3491                :
3492                :                         /*
3493                :                          * Make sure the group descriptor checksums
3494                :                          * are sane.  If they aren't, refuse to remount r/w.
```

```
3495                :                          */
3496              0 :                         for (g = 0; g < sbi->s_groups_count; g++) {
3497                :                                 struct ext4_group_desc *gdp =
3498              0 :                                         ext4_get_group_desc(sb, g, NULL);
3499                :
3500              0 :                                 if (!ext4_group_desc_csum_verify(sbi, g, gdp)) {
3501              0 :                                         ext4_msg(sb, KERN_ERR,
3502                :                                "ext4_remount: Checksum for group %u failed (%u!=%u)",
3503                :                                 g, le16_to_cpu(ext4_group_desc_csum(sbi, g, gdp)),
3504                :                                                le16_to_cpu(gdp->bg_checksum));
3505              0 :                                         err = -EINVAL;
3506              0 :                                         goto restore_opts;
3507                :                                 }
3508                :                         }
3509                :
3510                :                         /*
3511                :                          * If we have an unprocessed orphan list hanging
3512                :                          * around from a previously readonly bdev mount,
3513                :                          * require a full umount/remount for now.
3514                :                          */
3515              0 :                         if (es->s_last_orphan) {
3516              0 :                                 ext4_msg(sb, KERN_WARNING, "Couldn't "
3517                :                                        "remount RDWR because of unprocessed "
3518                :                                        "orphan inode list.  Please "
3519                :                                        "umount/remount instead");
3520              0 :                                 err = -EINVAL;
3521              0 :                                 goto restore_opts;
3522                :                         }
3523                :
3524                :                         /*
3525                :                          * Mounting a RDONLY partition read-write, so reread
3526                :                          * and store the current valid flag.  (It may have
3527                :                          * been changed by e2fsck since we originally mounted
3528                :                          * the partition.)
3529                :                          */
3530              0 :                         if (sbi->s_journal)
3531              0 :                                 ext4_clear_journal_err(sb, es);
3532              0 :                         sbi->s_mount_state = le16_to_cpu(es->s_state);
3533              0 :                         if ((err = ext4_group_extend(sb, es, n_blocks_count)))
3534              0 :                                 goto restore_opts;
3535              0 :                         if (!ext4_setup_super(sb, es, 0))
3536              0 :                                 sb->s_flags &= ~MS_RDONLY;
3537                :                 }
3538                :         }
3539              0 :         ext4_setup_system_zone(sb);
3540              0 :         if (sbi->s_journal == NULL)
3541              0 :                 ext4_commit_super(sb, 1);
3542                :
3543                : #ifdef CONFIG_QUOTA
3544                :         /* Release old quota file names */
3545              0 :         for (i = 0; i < MAXQUOTAS; i++)
3546              0 :                 if (old_opts.s_qf_names[i] &&
3547                :                     old_opts.s_qf_names[i] != sbi->s_qf_names[i])
3548              0 :                         kfree(old_opts.s_qf_names[i]);
3549                : #endif
3550              0 :         unlock_super(sb);
3551              0 :         unlock_kernel();
3552              0 :         return 0;
3553                :
3554              0 : restore_opts:
3555              0 :         sb->s_flags = old_sb_flags;
3556              0 :         sbi->s_mount_opt = old_opts.s_mount_opt;
3557              0 :         sbi->s_resuid = old_opts.s_resuid;
3558              0 :         sbi->s_resgid = old_opts.s_resgid;
3559              0 :         sbi->s_commit_interval = old_opts.s_commit_interval;
3560              0 :         sbi->s_min_batch_time = old_opts.s_min_batch_time;
3561              0 :         sbi->s_max_batch_time = old_opts.s_max_batch_time;
3562                : #ifdef CONFIG_QUOTA
3563              0 :         sbi->s_jquota_fmt = old_opts.s_jquota_fmt;
3564              0 :         for (i = 0; i < MAXQUOTAS; i++) {
3565              0 :                 if (sbi->s_qf_names[i] &&
3566                :                     old_opts.s_qf_names[i] != sbi->s_qf_names[i])
3567              0 :                         kfree(sbi->s_qf_names[i]);
3568              0 :                 sbi->s_qf_names[i] = old_opts.s_qf_names[i];
3569                :         }
3570                : #endif
3571              0 :         unlock_super(sb);
3572              0 :         unlock_kernel();
3573              0 :         return err;
3574                : }
3575                :
3576                : static int ext4_statfs(struct dentry *dentry, struct kstatfs *buf)
3577          76374 : {
3578          76374 :         struct super_block *sb = dentry->d_sb;
3579          76374 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
3580          76374 :         struct ext4_super_block *es = sbi->s_es;
3581                :         u64 fsid;
3582                :
3583          76374 :         if (test_opt(sb, MINIX_DF)) {
3584              0 :                 sbi->s_overhead_last = 0;
```

```
3585     152748 :         } else if (sbi->s_blocks_last != ext4_blocks_count(es)) {
3586         59 :                 ext4_group_t i, ngroups = ext4_get_groups_count(sb);
3587         59 :                 ext4_fsblk_t overhead = 0;
3588            :
3589            :                 /*
3590            :                  * Compute the overhead (FS structures).  This is constant
3591            :                  * for a given filesystem unless the number of block groups
3592            :                  * changes so we cache the previous value until it does.
3593            :                  */
3594            :
3595            :                 /*
3596            :                  * All of the blocks before first_data_block are
3597            :                  * overhead
3598            :                  */
3599         59 :                 overhead = le32_to_cpu(es->s_first_data_block);
3600            :
3601            :                 /*
3602            :                  * Add the overhead attributed to the superblock and
3603            :                  * block group descriptors.  If the sparse superblocks
3604            :                  * feature is turned on, then not all groups have this.
3605            :                  */
3606    1055419 :                 for (i = 0; i < ngroups; i++) {
3607    1055360 :                         overhead += ext4_bg_has_super(sb, i) +
3608            :                                 ext4_bg_num_gdb(sb, i);
3609            :                         cond_resched();
3610            :                 }
3611            :
3612            :                 /*
3613            :                  * Every block group has an inode bitmap, a block
3614            :                  * bitmap, and an inode table.
3615            :                  */
3616         59 :                 overhead += ngroups * (2 + sbi->s_itb_per_group);
3617         59 :                 sbi->s_overhead_last = overhead;
3618         59 :                 smp_wmb();
3619         59 :                 sbi->s_blocks_last = ext4_blocks_count(es);
3620            :         }
3621            :
3622      76374 :         buf->f_type = EXT4_SUPER_MAGIC;
3623      76374 :         buf->f_bsize = sb->s_blocksize;
3624      76374 :         buf->f_blocks = ext4_blocks_count(es) - sbi->s_overhead_last;
3625     229122 :         buf->f_bfree = percpu_counter_sum_positive(&sbi->s_freeblocks_counter) -
3626            :                        percpu_counter_sum_positive(&sbi->s_dirtyblocks_counter);
3627      76374 :         ext4_free_blocks_count_set(es, buf->f_bfree);
3628     152748 :         buf->f_bavail = buf->f_bfree - ext4_r_blocks_count(es);
3629     152748 :         if (buf->f_bfree < ext4_r_blocks_count(es))
3630          0 :                 buf->f_bavail = 0;
3631      76374 :         buf->f_files = le32_to_cpu(es->s_inodes_count);
3632     152748 :         buf->f_ffree = percpu_counter_sum_positive(&sbi->s_freeinodes_counter);
3633      76374 :         es->s_free_inodes_count = cpu_to_le32(buf->f_ffree);
3634      76374 :         buf->f_namelen = EXT4_NAME_LEN;
3635     229122 :         fsid = le64_to_cpup((void *)es->s_uuid) ^
3636            :                le64_to_cpup((void *)es->s_uuid + sizeof(u64));
3637      76374 :         buf->f_fsid.val[0] = fsid & 0xFFFFFFFFUL;
3638      76374 :         buf->f_fsid.val[1] = (fsid >> 32) & 0xFFFFFFFFUL;
3639            :
3640      76374 :         return 0;
3641            : }
3642            :
3643            : /* Helper function for writing quotas on sync - we need to start transaction
3644            :  * before quota file is locked for write. Otherwise the are possible deadlocks:
3645            :  * Process 1                         Process 2
3646            :  * ext4_create()                     quota_sync()
3647            :  *   jbd2_journal_start()                   write_dquot()
3648            :  *   vfs_dq_init()                         down(dqio_mutex)
3649            :  *     down(dqio_mutex)                    jbd2_journal_start()
3650            :  *
3651            :  */
3652            :
3653            : #ifdef CONFIG_QUOTA
3654            :
3655            : static inline struct inode *dquot_to_inode(struct dquot *dquot)
3656            : {
3657          0 :         return sb_dqopt(dquot->dq_sb)->files[dquot->dq_type];
3658            : }
3659            :
3660            : static int ext4_write_dquot(struct dquot *dquot)
3661          0 : {
3662            :         int ret, err;
3663            :         handle_t *handle;
3664            :         struct inode *inode;
3665            :
3666          0 :         inode = dquot_to_inode(dquot);
3667          0 :         handle = ext4_journal_start(inode,
3668            :                                     EXT4_QUOTA_TRANS_BLOCKS(dquot->dq_sb));
3669          0 :         if (IS_ERR(handle))
3670          0 :                 return PTR_ERR(handle);
3671          0 :         ret = dquot_commit(dquot);
3672          0 :         err = ext4_journal_stop(handle);
3673          0 :         if (!ret)
3674          0 :                 ret = err;
```

```
3675                0 :         return ret;
3676                  : }
3677                  :
3678                  : static int ext4_acquire_dquot(struct dquot *dquot)
3679                0 : {
3680                  :         int ret, err;
3681                  :         handle_t *handle;
3682                  :
3683                0 :         handle = ext4_journal_start(dquot_to_inode(dquot),
3684                0 :                                     EXT4_QUOTA_INIT_BLOCKS(dquot->dq_sb));
3685                0 :         if (IS_ERR(handle))
3686                0 :                 return PTR_ERR(handle);
3687                0 :         ret = dquot_acquire(dquot);
3688                0 :         err = ext4_journal_stop(handle);
3689                0 :         if (!ret)
3690                0 :                 ret = err;
3691                0 :         return ret;
3692                  : }
3693                  :
3694                  : static int ext4_release_dquot(struct dquot *dquot)
3695                0 : {
3696                  :         int ret, err;
3697                  :         handle_t *handle;
3698                  :
3699                0 :         handle = ext4_journal_start(dquot_to_inode(dquot),
3700                0 :                                     EXT4_QUOTA_DEL_BLOCKS(dquot->dq_sb));
3701                0 :         if (IS_ERR(handle)) {
3702                  :                 /* Release dquot anyway to avoid endless cycle in dqput() */
3703                0 :                 dquot_release(dquot);
3704                0 :                 return PTR_ERR(handle);
3705                  :         }
3706                0 :         ret = dquot_release(dquot);
3707                0 :         err = ext4_journal_stop(handle);
3708                0 :         if (!ret)
3709                0 :                 ret = err;
3710                0 :         return ret;
3711                  : }
3712                  :
3713                  : static int ext4_mark_dquot_dirty(struct dquot *dquot)
3714                0 : {
3715                  :         /* Are we journaling quotas? */
3716                0 :         if (EXT4_SB(dquot->dq_sb)->s_qf_names[USRQUOTA] ||
3717                  :             EXT4_SB(dquot->dq_sb)->s_qf_names[GRPQUOTA]) {
3718                0 :                 dquot_mark_dquot_dirty(dquot);
3719                0 :                 return ext4_write_dquot(dquot);
3720                  :         } else {
3721                0 :                 return dquot_mark_dquot_dirty(dquot);
3722                  :         }
3723                  : }
3724                  :
3725                  : static int ext4_write_info(struct super_block *sb, int type)
3726                0 : {
3727                  :         int ret, err;
3728                  :         handle_t *handle;
3729                  :
3730                  :         /* Data block + inode block */
3731                0 :         handle = ext4_journal_start(sb->s_root->d_inode, 2);
3732                0 :         if (IS_ERR(handle))
3733                0 :                 return PTR_ERR(handle);
3734                0 :         ret = dquot_commit_info(sb, type);
3735                0 :         err = ext4_journal_stop(handle);
3736                0 :         if (!ret)
3737                0 :                 ret = err;
3738                0 :         return ret;
3739                  : }
3740                  :
3741                  : /*
3742                  :  * Turn on quotas during mount time - we need to find
3743                  :  * the quota file and such...
3744                  :  */
3745                  : static int ext4_quota_on_mount(struct super_block *sb, int type)
3746                  : {
3747                0 :         return vfs_quota_on_mount(sb, EXT4_SB(sb)->s_qf_names[type],
3748                  :                                   EXT4_SB(sb)->s_jquota_fmt, type);
3749                  : }
3750                  :
3751                  : /*
3752                  :  * Standard function to be called on quota_on
3753                  :  */
3754                  : static int ext4_quota_on(struct super_block *sb, int type, int format_id,
3755                  :                          char *name, int remount)
3756                0 : {
3757                  :         int err;
3758                  :         struct path path;
3759                  :
3760                0 :         if (!test_opt(sb, QUOTA))
3761                0 :                 return -EINVAL;
3762                  :         /* When remounting, no checks are needed and in fact, name is NULL */
3763                0 :         if (remount)
3764                0 :                 return vfs_quota_on(sb, type, format_id, name, remount);
```

```
3765                :
3766              0 :         err = kern_path(name, LOOKUP_FOLLOW, &path);
3767              0 :         if (err)
3768              0 :                 return err;
3769                :
3770                :         /* Quotafile not on the same filesystem? */
3771              0 :         if (path.mnt->mnt_sb != sb) {
3772              0 :                 path_put(&path);
3773              0 :                 return -EXDEV;
3774                :         }
3775                :         /* Journaling quota? */
3776              0 :         if (EXT4_SB(sb)->s_qf_names[type]) {
3777                :                 /* Quotafile not in fs root? */
3778              0 :                 if (path.dentry->d_parent != sb->s_root)
3779              0 :                         ext4_msg(sb, KERN_WARNING,
3780                :                                 "Quota file not on filesystem root. "
3781                :                                 "Journaled quota will not work");
3782                :         }
3783                :
3784                :         /*
3785                :          * When we journal data on quota file, we have to flush journal to see
3786                :          * all updates to the file when we bypass pagecache...
3787                :          */
3788              0 :         if (EXT4_SB(sb)->s_journal &&
3789                :             ext4_should_journal_data(path.dentry->d_inode)) {
3790                :                 /*
3791                :                  * We don't need to lock updates but journal_flush() could
3792                :                  * otherwise be livelocked...
3793                :                  */
3794              0 :                 jbd2_journal_lock_updates(EXT4_SB(sb)->s_journal);
3795              0 :                 err = jbd2_journal_flush(EXT4_SB(sb)->s_journal);
3796              0 :                 jbd2_journal_unlock_updates(EXT4_SB(sb)->s_journal);
3797              0 :                 if (err) {
3798              0 :                         path_put(&path);
3799              0 :                         return err;
3800                :                 }
3801                :         }
3802                :
3803              0 :         err = vfs_quota_on_path(sb, type, format_id, &path);
3804              0 :         path_put(&path);
3805              0 :         return err;
3806                : }
3807                :
3808                : /* Read data from quotafile - avoid pagecache and such because we cannot afford
3809                :  * acquiring the locks... As quota files are never truncated and quota code
3810                :  * itself serializes the operations (and noone else should touch the files)
3811                :  * we don't have to be afraid of races */
3812                : static ssize_t ext4_quota_read(struct super_block *sb, int type, char *data,
3813                :                                size_t len, loff_t off)
3814              0 : {
3815              0 :         struct inode *inode = sb_dqopt(sb)->files[type];
3816              0 :         ext4_lblk_t blk = off >> EXT4_BLOCK_SIZE_BITS(sb);
3817              0 :         int err = 0;
3818              0 :         int offset = off & (sb->s_blocksize - 1);
3819                :         int tocopy;
3820                :         size_t toread;
3821                :         struct buffer_head *bh;
3822              0 :         loff_t i_size = i_size_read(inode);
3823                :
3824              0 :         if (off > i_size)
3825              0 :                 return 0;
3826              0 :         if (off+len > i_size)
3827              0 :                 len = i_size-off;
3828              0 :         toread = len;
3829              0 :         while (toread > 0) {
3830              0 :                 tocopy = sb->s_blocksize - offset < toread ?
3831                :                                 sb->s_blocksize - offset : toread;
3832              0 :                 bh = ext4_bread(NULL, inode, blk, 0, &err);
3833              0 :                 if (err)
3834              0 :                         return err;
3835              0 :                 if (!bh)        /* A hole? */
3836              0 :                         memset(data, 0, tocopy);
3837                :                 else
3838              0 :                         memcpy(data, bh->b_data+offset, tocopy);
3839                :                 brelse(bh);
3840              0 :                 offset = 0;
3841              0 :                 toread -= tocopy;
3842              0 :                 data += tocopy;
3843              0 :                 blk++;
3844                :         }
3845              0 :         return len;
3846                : }
3847                :
3848                : /* Write to quotafile (we know the transaction is already started and has
3849                :  * enough credits) */
3850                : static ssize_t ext4_quota_write(struct super_block *sb, int type,
3851                :                                 const char *data, size_t len, loff_t off)
3852              0 : {
3853              0 :         struct inode *inode = sb_dqopt(sb)->files[type];
3854              0 :         ext4_lblk_t blk = off >> EXT4_BLOCK_SIZE_BITS(sb);
```

```
3855                0 :          int err = 0;
3856                0 :          int offset = off & (sb->s_blocksize - 1);
3857                  :          int tocopy;
3858                0 :          int journal_quota = EXT4_SB(sb)->s_qf_names[type] != NULL;
3859                0 :          size_t towrite = len;
3860                  :          struct buffer_head *bh;
3861                0 :          handle_t *handle = journal_current_handle();
3862                  :
3863                0 :          if (EXT4_SB(sb)->s_journal && !handle) {
3864                0 :                  ext4_msg(sb, KERN_WARNING, "Quota write (off=%llu, len=%llu)"
3865                  :                          " cancelled because transaction is not started",
3866                  :                          (unsigned long long)off, (unsigned long long)len);
3867                0 :                  return -EIO;
3868                  :          }
3869                0 :          mutex_lock_nested(&inode->i_mutex, I_MUTEX_QUOTA);
3870                0 :          while (towrite > 0) {
3871                0 :                  tocopy = sb->s_blocksize - offset < towrite ?
3872                  :                          sb->s_blocksize - offset : towrite;
3873                0 :                  bh = ext4_bread(handle, inode, blk, 1, &err);
3874                0 :                  if (!bh)
3875                0 :                          goto out;
3876                0 :                  if (journal_quota) {
3877                0 :                          err = ext4_journal_get_write_access(handle, bh);
3878                0 :                          if (err) {
3879                  :                                  brelse(bh);
3880                  :                                  goto out;
3881                  :                          }
3882                  :                  }
3883                  :                  lock_buffer(bh);
3884                0 :                  memcpy(bh->b_data+offset, data, tocopy);
3885                0 :                  flush_dcache_page(bh->b_page);
3886                0 :                  unlock_buffer(bh);
3887                0 :                  if (journal_quota)
3888                0 :                          err = ext4_handle_dirty_metadata(handle, NULL, bh);
3889                  :                  else {
3890                  :                          /* Always do at least ordered writes for quotas */
3891                0 :                          err = ext4_jbd2_file_inode(handle, inode);
3892                0 :                          mark_buffer_dirty(bh);
3893                  :                  }
3894                  :                  brelse(bh);
3895                0 :                  if (err)
3896                0 :                          goto out;
3897                0 :                  offset = 0;
3898                0 :                  towrite -= tocopy;
3899                0 :                  data += tocopy;
3900                0 :                  blk++;
3901                  :          }
3902                0 : out:
3903                0 :          if (len == towrite) {
3904                0 :                  mutex_unlock(&inode->i_mutex);
3905                0 :                  return err;
3906                  :          }
3907                0 :          if (inode->i_size < off+len-towrite) {
3908                0 :                  i_size_write(inode, off+len-towrite);
3909                0 :                  EXT4_I(inode)->i_disksize = inode->i_size;
3910                  :          }
3911                0 :          inode->i_mtime = inode->i_ctime = CURRENT_TIME;
3912                0 :          ext4_mark_inode_dirty(handle, inode);
3913                0 :          mutex_unlock(&inode->i_mutex);
3914                0 :          return len - towrite;
3915                  : }
3916                  :
3917                  : #endif
3918                  :
3919                  : static int ext4_get_sb(struct file_system_type *fs_type, int flags,
3920                  :                        const char *dev_name, void *data, struct vfsmount *mnt)
3921               94 : {
3922               94 :          return get_sb_bdev(fs_type, flags, dev_name, data, ext4_fill_super,mnt);
3923                  : }
3924                  :
3925                  : static struct file_system_type ext4_fs_type = {
3926                  :          .owner          = THIS_MODULE,
3927                  :          .name           = "ext4",
3928                  :          .get_sb         = ext4_get_sb,
3929                  :          .kill_sb        = kill_block_super,
3930                  :          .fs_flags       = FS_REQUIRES_DEV,
3931                  : };
3932                  :
3933                  : #ifdef CONFIG_EXT4DEV_COMPAT
3934                  : static int ext4dev_get_sb(struct file_system_type *fs_type, int flags,
3935                  :                           const char *dev_name, void *data,struct vfsmount *mnt)
3936                0 : {
3937                0 :          printk(KERN_WARNING "EXT4-fs (%s): Update your userspace programs "
3938                  :                 "to mount using ext4\n", dev_name);
3939                0 :          printk(KERN_WARNING "EXT4-fs (%s): ext4dev backwards compatibility "
3940                  :                 "will go away by 2.6.31\n", dev_name);
3941                0 :          return get_sb_bdev(fs_type, flags, dev_name, data, ext4_fill_super,mnt);
3942                  : }
3943                  :
3944                  : static struct file_system_type ext4dev_fs_type = {
```

```
3945                :         .owner          = THIS_MODULE,
3946                :         .name           = "ext4dev",
3947                :         .get_sb         = ext4dev_get_sb,
3948                :         .kill_sb        = kill_block_super,
3949                :         .fs_flags       = FS_REQUIRES_DEV,
3950                : };
3951                : MODULE_ALIAS("ext4dev");
3952                : #endif
3953                :
3954                : static int __init init_ext4_fs(void)
3955              0 : {
3956                :         int err;
3957                :
3958              0 :         err = init_ext4_system_zone();
3959              0 :         if (err)
3960              0 :                 return err;
3961              0 :         ext4_kset = kset_create_and_add("ext4", NULL, fs_kobj);
3962              0 :         if (!ext4_kset)
3963              0 :                 goto out4;
3964              0 :         ext4_proc_root = proc_mkdir("fs/ext4", NULL);
3965              0 :         err = init_ext4_mballoc();
3966              0 :         if (err)
3967              0 :                 goto out3;
3968                :
3969              0 :         err = init_ext4_xattr();
3970              0 :         if (err)
3971              0 :                 goto out2;
3972              0 :         err = init_inodecache();
3973              0 :         if (err)
3974              0 :                 goto out1;
3975              0 :         err = register_filesystem(&ext4_fs_type);
3976              0 :         if (err)
3977              0 :                 goto out;
3978                : #ifdef CONFIG_EXT4DEV_COMPAT
3979              0 :         err = register_filesystem(&ext4dev_fs_type);
3980              0 :         if (err) {
3981              0 :                 unregister_filesystem(&ext4_fs_type);
3982              0 :                 goto out;
3983                :         }
3984                : #endif
3985              0 :         return 0;
3986                : out:
3987                :         destroy_inodecache();
3988              0 : out1:
3989              0 :         exit_ext4_xattr();
3990              0 : out2:
3991              0 :         exit_ext4_mballoc();
3992              0 : out3:
3993              0 :         remove_proc_entry("fs/ext4", NULL);
3994              0 :         kset_unregister(ext4_kset);
3995              0 : out4:
3996              0 :         exit_ext4_system_zone();
3997              0 :         return err;
3998                : }
3999                :
4000                : static void __exit exit_ext4_fs(void)
4001              0 : {
4002              0 :         unregister_filesystem(&ext4_fs_type);
4003                : #ifdef CONFIG_EXT4DEV_COMPAT
4004              0 :         unregister_filesystem(&ext4dev_fs_type);
4005                : #endif
4006                :         destroy_inodecache();
4007              0 :         exit_ext4_xattr();
4008              0 :         exit_ext4_mballoc();
4009              0 :         remove_proc_entry("fs/ext4", NULL);
4010              0 :         kset_unregister(ext4_kset);
4011              0 :         exit_ext4_system_zone();
4012              0 : }
4013                :
4014                : MODULE_AUTHOR("Remy Card, Stephen Tweedie, Andrew Morton, Andreas Dilger, Theodore Ts'o and others");
4015                : MODULE_DESCRIPTION("Fourth Extended Filesystem");
4016                : MODULE_LICENSE("GPL");
4017                : module_init(init_ext4_fs)
4018                : module_exit(exit_ext4_fs)
```