LCOV - code coverage report

Current view: directory - fs/ext4 - inode.c (source / functions)		Found	Hit	Coverage
Test: kernel_2_6_31_ext4_round_3.info	Lines:	1934	1284	66.4 %
Date: 2009-10-24	Functions:	82	65	79.3 %

```
       1                :  /*
       2                :   *  linux/fs/ext4/inode.c
       3                :   *
       4                :   * Copyright (C) 1992, 1993, 1994, 1995
       5                :   * Remy Card (card@masi.ibp.fr)
       6                :   * Laboratoire MASI - Institut Blaise Pascal
       7                :   * Universite Pierre et Marie Curie (Paris VI)
       8                :   *
       9                :   *  from
      10                :   *
      11                :   *  linux/fs/minix/inode.c
      12                :   *
      13                :   *  Copyright (C) 1991, 1992  Linus Torvalds
      14                :   *
      15                :   *  Goal-directed block allocation by Stephen Tweedie
      16                :   *      (sct@redhat.com), 1993, 1998
      17                :   *  Big-endian to little-endian byte-swapping/bitmaps by
      18                :   *        David S. Miller (davem@caip.rutgers.edu), 1995
      19                :   *  64-bit file support on 64-bit platforms by Jakub Jelinek
      20                :   *      (jj@sunsite.ms.mff.cuni.cz)
      21                :   *
      22                :   *  Assorted race fixes, rewrite of ext4_get_block() by Al Viro, 2000
      23                :   */
      24                :
      25                :  #include <linux/module.h>
      26                :  #include <linux/fs.h>
      27                :  #include <linux/time.h>
      28                :  #include <linux/jbd2.h>
      29                :  #include <linux/highuid.h>
      30                :  #include <linux/pagemap.h>
      31                :  #include <linux/quotaops.h>
      32                :  #include <linux/string.h>
      33                :  #include <linux/buffer_head.h>
      34                :  #include <linux/writeback.h>
      35                :  #include <linux/pagevec.h>
      36                :  #include <linux/mpage.h>
      37                :  #include <linux/namei.h>
      38                :  #include <linux/uio.h>
      39                :  #include <linux/bio.h>
      40                :
      41                :  #include "ext4_jbd2.h"
      42                :  #include "xattr.h"
      43                :  #include "acl.h"
      44                :  #include "ext4_extents.h"
      45                :
      46                :  #include <trace/events/ext4.h>
      47                :
      48                :  #define MPAGE_DA_EXTENT_TAIL 0x01
      49                :
      50                :  static inline int ext4_begin_ordered_truncate(struct inode *inode,
      51                :                                                loff_t new_size)
      52                :  {
      53          2017890 :          return jbd2_journal_begin_ordered_truncate(
      54                :                                          EXT4_SB(inode->i_sb)->s_journal,
      55                :                                          &EXT4_I(inode)->jinode,
      56                :                                          new_size);
      57                :  }
      58                :
```

```
                :  * Test whether an inode is a fast symlink.
                :  */
                : static int ext4_inode_is_fast_symlink(struct inode *inode)
                : {
                :         int ea_blocks = EXT4_I(inode)->i_file_acl ?
              0 :                 (inode->i_sb->s_blocksize >> 9) : 0;
                :
              0 :         return (S_ISLNK(inode->i_mode) && inode->i_blocks - ea_blocks == 0);
                : }
                :
                : /*
                :  * The ext4 forget function must perform a revoke if we are freeing data
                :  * which has been journaled.  Metadata (eg. indirect blocks) must be
                :  * revoked in all cases.
                :  *
                :  * "bh" may be NULL: a metadata block may have been freed from memory
                :  * but there may still be a record of it in the journal, and that record
                :  * still needs to be revoked.
                :  *
                :  * If the handle isn't valid we're not journaling, but we still need to
                :  * call into ext4_journal_revoke() to put the buffer head.
                :  */
                : int ext4_forget(handle_t *handle, int is_metadata, struct inode *inode,
                :                 struct buffer_head *bh, ext4_fsblk_t blocknr)
      153755336 : {
                :         int err;
                :
      153755336 :         might_sleep();
                :
                :         BUFFER_TRACE(bh, "enter");
                :
      153752950 :         jbd_debug(4, "forgetting bh %p: is_metadata = %d, mode %o, "
                :                   "data mode %x\n",
                :                   bh, is_metadata, inode->i_mode,
                :                   test_opt(inode->i_sb, DATA_FLAGS));
                :
                :         /* Never use the revoke function if we are doing full data
                :          * journaling: there is no need to, and a V1 superblock won't
                :          * support it.  Otherwise, only skip the revoke on un-journaled
                :          * data blocks. */
                :
      461045999 :         if (test_opt(inode->i_sb, DATA_FLAGS) == EXT4_MOUNT_JOURNAL_DATA ||
                :             (!is_metadata && !ext4_should_journal_data(inode))) {
      152877687 :                 if (bh) {
                :                         BUFFER_TRACE(bh, "call jbd2_journal_forget");
          179 :                         return ext4_journal_forget(handle, bh);
                :                 }
      152877508 :                 return 0;
                :         }
                :
                :         /*
                :          * data!=journal && (is_metadata || should_journal_data(inode))
                :          */
                :         BUFFER_TRACE(bh, "call ext4_journal_revoke");
        876306 :         err = ext4_journal_revoke(handle, blocknr, bh);
        877105 :         if (err)
              0 :                 ext4_abort(inode->i_sb, __func__,
                :                            "error %d when attempting revoke", err);
                :         BUFFER_TRACE(bh, "exit");
        877105 :         return err;
                : }
                :
                : /*
                :  * Work out how many blocks we need to proceed with the next chunk of a
                :  * truncate transaction.
                :  */
                : static unsigned long blocks_for_truncate(struct inode *inode)
                : {
```

```
130                 :         ext4_lblk_t needed;
131                 :
132         1664364 :         needed = inode->i_blocks >> (inode->i_sb->s_blocksize_bits - 9);
133                 :
134                 :         /* Give ourselves just enough room to cope with inodes in which
135                 :          * i_blocks is corrupt: we've seen disk corruptions in the past
136                 :          * which resulted in random data in an inode which looked enough
137                 :          * like a regular file for ext4 to try to delete it.  Things
138                 :          * will go a bit crazy if that happens, but at least we should
139                 :          * try not to panic the whole kernel. */
140         1664364 :         if (needed < 2)
141          656857 :                 needed = 2;
142                 :
143                 :         /* But we need to bound the transaction so we don't overflow the
144                 :          * journal. */
145         1664364 :         if (needed > EXT4_MAX_TRANS_DATA)
146          691770 :                 needed = EXT4_MAX_TRANS_DATA;
147                 :
148         4993092 :         return EXT4_DATA_TRANS_BLOCKS(inode->i_sb) + needed;
149                 : }
150                 :
151                 : /*
152                 :  * Truncate transactions can be complex and absolutely huge.  So we need to
153                 :  * be able to restart the transaction at a conventient checkpoint to make
154                 :  * sure we don't overflow the journal.
155                 :  *
156                 :  * start_transaction gets us a new handle for a truncate transaction,
157                 :  * and extend_transaction tries to extend the existing one a bit.  If
158                 :  * extend fails, we need to propagate the failure up and restart the
159                 :  * transaction in the top-level truncate loop. --sct
160                 :  */
161                 : static handle_t *start_transaction(struct inode *inode)
162                 : {
163                 :         handle_t *result;
164                 :
165              18 :         result = ext4_journal_start(inode, blocks_for_truncate(inode));
166               9 :         if (!IS_ERR(result))
167               9 :                 return result;
168                 :
169               0 :         ext4_std_error(inode->i_sb, PTR_ERR(result));
170               0 :         return result;
171                 : }
172                 :
173                 : /*
174                 :  * Try to extend this transaction for the purposes of truncation.
175                 :  *
176                 :  * Returns 0 if we managed to create more room.  If we can't create more
177                 :  * room, and the transaction must be restarted we return 1.
178                 :  */
179                 : static int try_to_extend_transaction(handle_t *handle, struct inode *inode)
180               9 : {
181               9 :         if (!ext4_handle_valid(handle))
182               0 :                 return 0;
183               9 :         if (ext4_handle_has_enough_credits(handle, EXT4_RESERVE_TRANS_BLOCKS+1))
184               9 :                 return 0;
185               0 :         if (!ext4_journal_extend(handle, blocks_for_truncate(inode)))
186               0 :                 return 0;
187               0 :         return 1;
188                 : }
189                 :
190                 : /*
191                 :  * Restart the transaction associated with *handle.  This does a commit,
192                 :  * so before we call here everything must be consistently dirtied against
193                 :  * this transaction.
194                 :  */
195                 : static int ext4_journal_test_restart(handle_t *handle, struct inode *inode)
196               0 : {
197               0 :         BUG_ON(EXT4_JOURNAL(inode) == NULL);
198               0 :         jbd_debug(2, "restarting handle %p\n", handle);
199               0 :         return ext4_journal_restart(handle, blocks_for_truncate(inode));
200                 : }
```

```
   201                :
   202                : /*
   203                :  * Called at the last iput() if i_nlink is zero.
   204                :  */
   205                : void ext4_delete_inode(struct inode *inode)
   206        1664355 : {
   207                :         handle_t *handle;
   208                :         int err;
   209                :
   210        1664355 :         if (ext4_should_order_data(inode))
   211                :                 ext4_begin_ordered_truncate(inode, 0);
   212        1664355 :         truncate_inode_pages(&inode->i_data, 0);
   213                :
   214        1664355 :         if (is_bad_inode(inode))
   215              0 :                 goto no_delete;
   216                :
   217        3328710 :         handle = ext4_journal_start(inode, blocks_for_truncate(inode)+3);
   218        1664355 :         if (IS_ERR(handle)) {
   219              0 :                 ext4_std_error(inode->i_sb, PTR_ERR(handle));
   220                :                 /*
   221                :                  * If we're going to skip the normal cleanup, we still need to
   222                :                  * make sure that the in-core orphan linked list is properly
   223                :                  * cleaned up.
   224                :                  */
   225              0 :                 ext4_orphan_del(NULL, inode);
   226              0 :                 goto no_delete;
   227                :         }
   228                :
   229        1664355 :         if (IS_SYNC(inode))
   230                :                 ext4_handle_sync(handle);
   231        1664355 :         inode->i_size = 0;
   232        1664355 :         err = ext4_mark_inode_dirty(handle, inode);
   233        1664354 :         if (err) {
   234              0 :                 ext4_warning(inode->i_sb, __func__,
   235                :                              "couldn't mark inode dirty (err %d)", err);
   236              0 :                 goto stop_handle;
   237                :         }
   238        1664354 :         if (inode->i_blocks)
   239        1662895 :                 ext4_truncate(inode);
   240                :
   241                :         /*
   242                :          * ext4_ext_truncate() doesn't reserve any slop when it
   243                :          * restarts journal transactions; therefore there may not be
   244                :          * enough credits left in the handle to remove the inode from
   245                :          * the orphan list and set the dtime field.
   246                :          */
   247        1664355 :         if (!ext4_handle_has_enough_credits(handle, 3)) {
   248              0 :                 err = ext4_journal_extend(handle, 3);
   249              0 :                 if (err > 0)
   250              0 :                         err = ext4_journal_restart(handle, 3);
   251              0 :                 if (err != 0) {
   252              0 :                         ext4_warning(inode->i_sb, __func__,
   253                :                                      "couldn't extend journal (err %d)", err);
   254              0 :                 stop_handle:
   255              0 :                         ext4_journal_stop(handle);
   256              0 :                         goto no_delete;
   257                :                 }
   258                :         }
   259                :
   260                :         /*
   261                :          * Kill off the orphan record which ext4_truncate created.
   262                :          * AKPM: I think this can be inside the above `if'.
   263                :          * Note that ext4_orphan_del() has to be able to cope with the
   264                :          * deletion of a non-existent orphan - this is because we don't
   265                :          * know if ext4_truncate() actually created an orphan record.
   266                :          * (Well, we could do this if we need to, but heck - it works)
   267                :          */
   268        1664355 :         ext4_orphan_del(handle, inode);
   269        1664355 :         EXT4_I(inode)->i_dtime       = get_seconds();
   270                :
   271                :         /*
```

```
   272                :          * One subtle ordering requirement: if anything has gone wrong
   273                :          * (transaction abort, IO errors, whatever), then we can still
   274                :          * do these next steps (the fs will already have been marked as
   275                :          * having errors), but we can't free the inode if the mark_dirty
   276                :          * fails.
   277                :          */
   278          1664355 :         if (ext4_mark_inode_dirty(handle, inode))
   279                :                 /* If that failed, just do the required in-core inode clear. */
   280                0 :                 clear_inode(inode);
   281                :         else
   282          1664355 :                 ext4_free_inode(handle, inode);
   283          1664355 :         ext4_journal_stop(handle);
   284          1664350 :         return;
   285                0 : no_delete:
   286                0 :         clear_inode(inode);     /* We must guarantee clearing of inode... */
   287                : }
   288                :
   289                : typedef struct {
   290                :         __le32  *p;
   291                :         __le32  key;
   292                :         struct buffer_head *bh;
   293                : } Indirect;
   294                :
   295                : static inline void add_chain(Indirect *p, struct buffer_head *bh, __le32 *v)
   296                : {
   297         10758375 :         p->key = *(p->p = v);
   298         10758375 :         p->bh = bh;
   299                : }
   300                :
   301                : /**
   302                :  *      ext4_block_to_path - parse the block number into array of offsets
   303                :  *      @inode: inode in question (we are only interested in its superblock)
   304                :  *      @i_block: block number to be parsed
   305                :  *      @offsets: array to store the offsets in
   306                :  *      @boundary: set this non-zero if the referred-to block is likely to be
   307                :  *             followed (on disk) by an indirect block.
   308                :  *
   309                :  *      To store the locations of file's data ext4 uses a data structure common
   310                :  *      for UNIX filesystems - tree of pointers anchored in the inode, with
   311                :  *      data blocks at leaves and indirect blocks in intermediate nodes.
   312                :  *      This function translates the block number into path in that tree -
   313                :  *      return value is the path length and @offsets[n] is the offset of
   314                :  *      pointer to (n+1)th node in the nth one. If @block is out of range
   315                :  *      (negative or too large) warning is printed and zero returned.
   316                :  *
   317                :  *      Note: function doesn't find node addresses, so no IO is needed. All
   318                :  *      we need to know is the capacity of indirect blocks (taken from the
   319                :  *      inode->i_sb).
   320                :  */
   321                :
   322                : /*
   323                :  * Portability note: the last comparison (check that we fit into triple
   324                :  * indirect block) is spelled differently, because otherwise on an
   325                :  * architecture with 32-bit longs and 8Kb pages we might get into trouble
   326                :  * if our filesystem had 8Kb blocks. We might use long long, but that would
   327                :  * kill us on x86. Oh, well, at least the sign propagation does not matter -
   328                :  * i_block would have to be negative in the very beginning, so we would not
   329                :  * get there at all.
   330                :  */
   331                :
   332                : static int ext4_block_to_path(struct inode *inode,
   333                :                               ext4_lblk_t i_block,
   334                :                               ext4_lblk_t offsets[4], int *boundary)
   335          6650883 : {
   336          6650883 :         int ptrs = EXT4_ADDR_PER_BLOCK(inode->i_sb);
   337         13301766 :         int ptrs_bits = EXT4_ADDR_PER_BLOCK_BITS(inode->i_sb);
   338          6650883 :         const long direct_blocks = EXT4_NDIR_BLOCKS,
   339          6650883 :                 indirect_blocks = ptrs,
   340          6650883 :                 double_blocks = (1 << (ptrs_bits * 2));
   341          6650883 :         int n = 0;
   342          6650883 :         int final = 0;
```

```
343                :
344                :         if (i_block < 0) {
345                :                 ext4_warning(inode->i_sb, "ext4_block_to_path", "block < 0");
346         6650883 :         } else if (i_block < direct_blocks) {
347         2776572 :                 offsets[n++] = i_block;
348         2776572 :                 final = direct_blocks;
349         3874311 :         } else if ((i_block -= direct_blocks) < indirect_blocks) {
350         2517783 :                 offsets[n++] = EXT4_IND_BLOCK;
351         2517783 :                 offsets[n++] = i_block;
352         2517783 :                 final = ptrs;
353         1356528 :         } else if ((i_block -= indirect_blocks) < double_blocks) {
354         1356528 :                 offsets[n++] = EXT4_DIND_BLOCK;
355         1356528 :                 offsets[n++] = i_block >> ptrs_bits;
356         1356528 :                 offsets[n++] = i_block & (ptrs - 1);
357         1356528 :                 final = ptrs;
358               0 :         } else if (((i_block -= double_blocks) >> (ptrs_bits * 2)) < ptrs) {
359               0 :                 offsets[n++] = EXT4_TIND_BLOCK;
360               0 :                 offsets[n++] = i_block >> (ptrs_bits * 2);
361               0 :                 offsets[n++] = (i_block >> ptrs_bits) & (ptrs - 1);
362               0 :                 offsets[n++] = i_block & (ptrs - 1);
363               0 :                 final = ptrs;
364                :         } else {
365               0 :                 ext4_warning(inode->i_sb, "ext4_block_to_path",
366                :                              "block %lu > max in inode %lu",
367                :                              i_block + direct_blocks +
368                :                              indirect_blocks + double_blocks, inode->i_ino);
369                :         }
370         6650769 :         if (boundary)
371         6650760 :                 *boundary = final - 1 - (i_block & (ptrs - 1));
372         6650769 :         return n;
373                : }
374                :
375                : static int __ext4_check_blockref(const char *function, struct inode *inode,
376                :                                  __le32 *p, unsigned int max)
377           12962 : {
378           12962 :         __le32 *bref = p;
379                :         unsigned int blk;
380                :
381          330812 :         while (bref < p+max) {
382          304888 :                 blk = le32_to_cpu(*bref++);
383          492798 :                 if (blk &&
384                :                     unlikely(!ext4_data_block_valid(EXT4_SB(inode->i_sb),
385                :                                                     blk, 1))) {
386               0 :                         ext4_error(inode->i_sb, function,
387                :                                    "invalid block reference %u "
388                :                                    "in inode #%lu", blk, inode->i_ino);
389               0 :                         return -EIO;
390                :                 }
391                :         }
392           12962 :         return 0;
393                : }
394                :
395                :
396                : #define ext4_check_indirect_blockref(inode, bh)                         \
397                :         __ext4_check_blockref(__func__, inode, (__le32 *)(bh)->b_data,  \
398                :                               EXT4_ADDR_PER_BLOCK((inode)->i_sb))
399                :
400                : #define ext4_check_inode_blockref(inode)                                \
401                :         __ext4_check_blockref(__func__, inode, EXT4_I(inode)->i_data,   \
402                :                               EXT4_NDIR_BLOCKS)
403                :
404                : /**
405                :  *      ext4_get_branch - read the chain of indirect blocks leading to data
406                :  *      @inode: inode in question
407                :  *      @depth: depth of the chain (1 - direct pointer, etc.)
408                :  *      @offsets: offsets of pointers in inode/indirect blocks
409                :  *      @chain: place to store the result
410                :  *      @err: here we store the error value
411                :  *
412                :  *      Function fills the array of triples <key, p, bh> and returns %NULL
413                :  *      if everything went OK or the pointer to the last filled triple
```

```
                :  *      (incomplete one) otherwise. Upon the return chain[i].key contains
                :  *      the number of (i+1)-th block in the chain (as it is stored in memory,
                :  *      i.e. little-endian 32-bit), chain[i].p contains the address of that
                :  *      number (it points into struct inode for i==0 and into the bh->b_data
                :  *      for i>0) and chain[i].bh points to the buffer_head of i-th indirect
                :  *      block for i>0 and NULL for i==0. In other words, it holds the block
                :  *      numbers of the chain, addresses they were taken from (and where we can
                :  *      verify that chain did not change) and buffer_heads hosting these
                :  *      numbers.
                :  *
                :  *      Function stops when it stumbles upon zero pointer (absent block)
                :  *              (pointer to last triple returned, *@err == 0)
                :  *      or when it gets an IO error reading an indirect block
                :  *              (ditto, *@err == -EIO)
                :  *      or when it reads all @depth-1 indirect blocks successfully and finds
                :  *      the whole chain, all way to the data (returns %NULL, *err == 0).
                :  *
                :  *      Need to be called with
                :  *      down_read(&EXT4_I(inode)->i_data_sem)
                :  */
                : static Indirect *ext4_get_branch(struct inode *inode, int depth,
                :                                  ext4_lblk_t  *offsets,
                :                                  Indirect chain[4], int *err)
        6650835 : {
        6650835 :         struct super_block *sb = inode->i_sb;
        6650835 :         Indirect *p = chain;
                :         struct buffer_head *bh;
                :
        6650835 :         *err = 0;
                :         /* i_data is not going away, no lock needed */
        6650835 :         add_chain(chain, NULL, EXT4_I(inode)->i_data + *offsets);
        6650835 :         if (!p->key)
        2452172 :                 goto no_block;
        7871904 :         while (--depth) {
        8214741 :                 bh = sb_getblk(sb, le32_to_cpu(p->key));
        4107367 :                 if (unlikely(!bh))
              0 :                         goto failure;
                :
        4107391 :                 if (!bh_uptodate_or_lock(bh)) {
            568 :                         if (bh_submit_read(bh) < 0) {
                :                                 put_bh(bh);
                :                                 goto failure;
                :                         }
                :                         /* validate block references */
            568 :                         if (ext4_check_indirect_blockref(inode, bh)) {
                :                                 put_bh(bh);
                :                                 goto failure;
                :                         }
                :                 }
                :
        4107540 :                 add_chain(++p, bh, (__le32 *)bh->b_data + *++offsets);
                :                 /* Reader: end */
        4107540 :                 if (!p->key)
         434299 :                         goto no_block;
                :         }
        3764530 :         return NULL;
                :
              0 : failure:
              0 :         *err = -EIO;
        2886471 : no_block:
        2886471 :         return p;
                : }
                :
                : /**
                :  *      ext4_find_near - find a place for allocation with sufficient locality
                :  *      @inode: owner
                :  *      @ind: descriptor of indirect block.
                :  *
                :  *      This function returns the preferred place for block allocation.
                :  *      It is used when heuristic for sequential allocation fails.
                :  *      Rules are:
```

```
485                 :  *        + if there is a block to the left of our position - allocate near it.
486                 :  *        + if pointer will live in indirect block - allocate near that block.
487                 :  *        + if pointer will live in inode - allocate in the same
488                 :  *          cylinder group.
489                 :  *
490                 :  * In the latter case we colour the starting block by the callers PID to
491                 :  * prevent it from clashing with concurrent allocations for a different inode
492                 :  * in the same block group.   The PID is used here so that functionally related
493                 :  * files will be close-by on-disk.
494                 :  *
495                 :  *      Caller must make sure that @ind is valid and will stay that way.
496                 :  */
497                 : static ext4_fsblk_t ext4_find_near(struct inode *inode, Indirect *ind)
498                 : {
499          259184 :         struct ext4_inode_info *ei = EXT4_I(inode);
500          259184 :         __le32 *start = ind->bh ? (__le32 *) ind->bh->b_data : ei->i_data;
501                 :         __le32 *p;
502                 :         ext4_fsblk_t bg_start;
503                 :         ext4_fsblk_t last_block;
504                 :         ext4_grpblk_t colour;
505                 :         ext4_group_t block_group;
506          518368 :         int flex_size = ext4_flex_bg_size(EXT4_SB(inode->i_sb));
507                 :
508                 :         /* Try to find previous block */
509          277177 :         for (p = ind->p - 1; p >= start; p--) {
510          177131 :                 if (*p)
511          159138 :                         return le32_to_cpu(*p);
512                 :         }
513                 :
514                 :         /* No such thing, so let's try location of indirect block */
515          100046 :         if (ind->bh)
516               0 :                 return ind->bh->b_blocknr;
517                 :
518                 :         /*
519                 :          * It is going to be referred to from the inode itself? OK, just put it
520                 :          * into the same cylinder group then.
521                 :          */
522          100046 :         block_group = ei->i_block_group;
523          100046 :         if (flex_size >= EXT4_FLEX_SIZE_DIR_ALLOC_SCHEME) {
524               0 :                 block_group &= ~(flex_size-1);
525               0 :                 if (S_ISREG(inode->i_mode))
526               0 :                         block_group++;
527                 :         }
528          200092 :         bg_start = ext4_group_first_block_no(inode->i_sb, block_group);
529          300138 :         last_block = ext4_blocks_count(EXT4_SB(inode->i_sb)->s_es) - 1;
530                 :
531                 :         /*
532                 :          * If we are doing delayed allocation, we don't need take
533                 :          * colour into account.
534                 :          */
535          200092 :         if (test_opt(inode->i_sb, DELALLOC))
536          100046 :                 return bg_start;
537                 :
538               0 :         if (bg_start + EXT4_BLOCKS_PER_GROUP(inode->i_sb) <= last_block)
539               0 :                 colour = (current->pid % 16) *
540                 :                         (EXT4_BLOCKS_PER_GROUP(inode->i_sb) / 16);
541                 :         else
542               0 :                 colour = (current->pid % 16) * ((last_block - bg_start) / 16);
543               0 :         return bg_start + colour;
544                 : }
545                 :
546                 : /**
547                 :  *      ext4_find_goal - find a preferred place for allocation.
548                 :  *      @inode: owner
549                 :  *      @block:  block we want
550                 :  *      @partial: pointer to the last triple within a chain
551                 :  *
552                 :  *      Normally this function find the preferred place for block allocation,
553                 :  *      returns it.
554                 :  */
555                 : static ext4_fsblk_t ext4_find_goal(struct inode *inode, ext4_lblk_t block,
```

```
556                 :                                       Indirect *partial)
557                 : {
558                 :         /*
559                 :          * XXX need to get goal block from mballoc's data structures
560                 :          */
561                 :
562          259184 :         return ext4_find_near(inode, partial);
563                 : }
564                 :
565                 : /**
566                 :  *      ext4_blks_to_allocate: Look up the block map and count the number
567                 :  *      of direct blocks need to be allocated for the given branch.
568                 :  *
569                 :  *      @branch: chain of indirect blocks
570                 :  *      @k: number of blocks need for indirect blocks
571                 :  *      @blks: number of data blocks to be mapped.
572                 :  *      @blocks_to_boundary:  the offset in the indirect block
573                 :  *
574                 :  *      return the total number of blocks to be allocate, including the
575                 :  *      direct and indirect blocks.
576                 :  */
577                 : static int ext4_blks_to_allocate(Indirect *branch, int k, unsigned int blks,
578                 :                                  int blocks_to_boundary)
579                 : {
580          259184 :         unsigned int count = 0;
581                 :
582                 :         /*
583                 :          * Simple case, [t,d]Indirect block(s) has not allocated yet
584                 :          * then it's clear blocks on that path have not allocated
585                 :          */
586          259184 :         if (k > 0) {
587                 :                 /* right now we don't handle cross boundary allocation */
588           77190 :                 if (blks < blocks_to_boundary + 1)
589           77190 :                         count += blks;
590                 :                 else
591               0 :                         count += blocks_to_boundary + 1;
592           77190 :                 return count;
593                 :         }
594                 :
595          181994 :         count++;
596         1441859 :         while (count < blks && count <= blocks_to_boundary &&
597                 :                 le32_to_cpu(*(branch[0].p + count)) == 0) {
598         1259865 :                 count++;
599                 :         }
600          181994 :         return count;
601                 : }
602                 :
603                 : /**
604                 :  *      ext4_alloc_blocks: multiple allocate blocks needed for a branch
605                 :  *      @indirect_blks: the number of blocks need to allocate for indirect
606                 :  *                      blocks
607                 :  *
608                 :  *      @new_blocks: on return it will store the new block numbers for
609                 :  *      the indirect blocks(if needed) and the first direct block,
610                 :  *      @blks:  on return it will store the total number of allocated
611                 :  *              direct blocks
612                 :  */
613                 : static int ext4_alloc_blocks(handle_t *handle, struct inode *inode,
614                 :                              ext4_lblk_t iblock, ext4_fsblk_t goal,
615                 :                              int indirect_blks, int blks,
616                 :                              ext4_fsblk_t new_blocks[4], int *err)
617          259161 : {
618                 :         struct ext4_allocation_request ar;
619                 :         int target, i;
620          259161 :         unsigned long count = 0, blk_allocated = 0;
621          259161 :         int index = 0;
622          259161 :         ext4_fsblk_t current_block = 0;
623          259161 :         int ret = 0;
624                 :
625                 :         /*
626                 :          * Here we try to allocate the requested multiple blocks at once,
```

```
627                :          * on a best-effort basis.
628                :          * To build a branch, we should allocate blocks for
629                :          * the indirect blocks(if not allocated yet), and at least
630                :          * the first direct block of this branch.  That's the
631                :          * minimum number of blocks need to allocate(required)
632                :          */
633                :         /* first we try to allocate the indirect blocks */
634         259161 :         target = indirect_blks;
635         595511 :         while (target > 0) {
636          77190 :                 count = target;
637                :                 /* allocating blocks for indirect blocks and direct blocks */
638          77190 :                 current_block = ext4_new_meta_blocks(handle, inode,
639                :                                                      goal, &count, err);
640          77189 :                 if (*err)
641              0 :                         goto failed_out;
642                :
643          77189 :                 target -= count;
644                :                 /* allocate blocks for indirect blocks */
645         231570 :                 while (index < indirect_blks && count) {
646          77192 :                         new_blocks[index++] = current_block++;
647          77192 :                         count--;
648                :                 }
649          77189 :                 if (count > 0) {
650                :                         /*
651                :                          * save the new block number
652                :                          * for the first direct block
653                :                          */
654              0 :                         new_blocks[index] = current_block;
655              0 :                         printk(KERN_INFO "%s returned more blocks than "
656                :                                               "requested\n", __func__);
657              0 :                         WARN_ON(1);
658              0 :                         break;
659                :                 }
660                :         }
661                :
662         259160 :         target = blks - count ;
663         259160 :         blk_allocated = count;
664         259160 :         if (!target)
665             11 :                 goto allocated;
666                :         /* Now allocate data blocks */
667                :         memset(&ar, 0, sizeof(ar));
668         259142 :         ar.inode = inode;
669         259142 :         ar.goal = goal;
670         259142 :         ar.len = target;
671         259142 :         ar.logical = iblock;
672         259142 :         if (S_ISREG(inode->i_mode))
673                :                 /* enable in-core preallocation only for regular files */
674         226340 :                 ar.flags = EXT4_MB_HINT_DATA;
675                :
676         259142 :         current_block = ext4_mb_new_blocks(handle, &ar, err);
677                :
678         259125 :         if (*err && (target == blks)) {
679                :                 /*
680                :                  * if the allocation failed and we didn't allocate
681                :                  * any blocks before
682                :                  */
683              0 :                 goto failed_out;
684                :         }
685         259125 :         if (!*err) {
686         259145 :                 if (target == blks) {
687                :                         /*
688                :                          * save the new block number
689                :                          * for the first direct block
690                :                          */
691         259134 :                         new_blocks[index] = current_block;
692                :                 }
693         259145 :                 blk_allocated += ar.len;
694                :         }
695         259136 : allocated:
696                :         /* total number of blocks allocated for direct blocks */
697         259136 :         ret = blk_allocated;
```

```
698          259136 :         *err = 0;
699          259136 :         return ret;
700                0 : failed_out:
701                0 :         for (i = 0; i < index; i++)
702                0 :                 ext4_free_blocks(handle, inode, new_blocks[i], 1, 0);
703                0 :         return ret;
704                  : }
705                  :
706                  : /**
707                  :  *      ext4_alloc_branch - allocate and set up a chain of blocks.
708                  :  *      @inode: owner
709                  :  *      @indirect_blks: number of allocated indirect blocks
710                  :  *      @blks: number of allocated direct blocks
711                  :  *      @offsets: offsets (in the blocks) to store the pointers to next.
712                  :  *      @branch: place to store the chain in.
713                  :  *
714                  :  *      This function allocates blocks, zeroes out all but the last one,
715                  :  *      links them into chain and (if we are synchronous) writes them to disk.
716                  :  *      In other words, it prepares a branch that can be spliced onto the
717                  :  *      inode. It stores the information about that chain in the branch[], in
718                  :  *      the same format as ext4_get_branch() would do. We are calling it after
719                  :  *      we had read the existing part of chain and partial points to the last
720                  :  *      triple of that (one with zero ->key). Upon the exit we have the same
721                  :  *      picture as after the successful ext4_get_block(), except that in one
722                  :  *      place chain is disconnected - *branch->p is still zero (we did not
723                  :  *      set the last link), but branch->key contains the number that should
724                  :  *      be placed into *branch->p to fill that gap.
725                  :  *
726                  :  *      If allocation fails we free all blocks we've allocated (and forget
727                  :  *      their buffer_heads) and return the error value the from failed
728                  :  *      ext4_alloc_block() (normally -ENOSPC). Otherwise we set the chain
729                  :  *      as described above and return 0.
730                  :  */
731                  : static int ext4_alloc_branch(handle_t *handle, struct inode *inode,
732                  :                              ext4_lblk_t iblock, int indirect_blks,
733                  :                              int *blks, ext4_fsblk_t goal,
734                  :                              ext4_lblk_t *offsets, Indirect *branch)
735          259160 : {
736          259160 :         int blocksize = inode->i_sb->s_blocksize;
737          259160 :         int i, n = 0;
738          259160 :         int err = 0;
739                  :         struct buffer_head *bh;
740                  :         int num;
741                  :         ext4_fsblk_t new_blocks[4];
742                  :         ext4_fsblk_t current_block;
743                  :
744          259160 :         num = ext4_alloc_blocks(handle, inode, iblock, goal, indirect_blks,
745                  :                                 *blks, new_blocks, &err);
746          259084 :         if (err)
747                0 :                 return err;
748                  :
749          259084 :         branch[0].key = cpu_to_le32(new_blocks[0]);
750                  :         /*
751                  :          * metadata blocks and data blocks are allocated.
752                  :          */
753          336277 :         for (n = 1; n <= indirect_blks;  n++) {
754                  :                 /*
755                  :                  * Get buffer_head for parent block, zero it out
756                  :                  * and set the pointer to new one, then send
757                  :                  * parent to disk.
758                  :                  */
759          154385 :                 bh = sb_getblk(inode->i_sb, new_blocks[n-1]);
760           77193 :                 branch[n].bh = bh;
761           77193 :                 lock_buffer(bh);
762                  :                 BUFFER_TRACE(bh, "call get_create_access");
763           77193 :                 err = ext4_journal_get_create_access(handle, bh);
764           77193 :                 if (err) {
765                0 :                         unlock_buffer(bh);
766                0 :                         brelse(bh);
767                0 :                         goto failed;
768                  :                 }
```

```
769                :
770         154386 :                 memset(bh->b_data, 0, blocksize);
771          77193 :                 branch[n].p = (__le32 *) bh->b_data + offsets[n];
772          77193 :                 branch[n].key = cpu_to_le32(new_blocks[n]);
773          77193 :                 *branch[n].p = branch[n].key;
774          77193 :                 if (n == indirect_blks) {
775          77190 :                         current_block = new_blocks[n];
776                :                         /*
777                :                          * End of chain, update the last new metablock of
778                :                          * the chain to point to the new allocated
779                :                          * data blocks numbers
780                :                          */
781         969178 :                         for (i = 1; i < num; i++)
782         891988 :                                 *(branch[n].p + i) = cpu_to_le32(++current_block);
783                :                 }
784                :                 BUFFER_TRACE(bh, "marking uptodate");
785                :                 set_buffer_uptodate(bh);
786          77193 :                 unlock_buffer(bh);
787                :
788                :                 BUFFER_TRACE(bh, "call ext4_handle_dirty_metadata");
789          77193 :                 err = ext4_handle_dirty_metadata(handle, inode, bh);
790          77193 :                 if (err)
791              0 :                         goto failed;
792                :         }
793         259085 :         *blks = num;
794         259085 :         return err;
795              0 : failed:
796                :         /* Allocation failed, free what we already allocated */
797              0 :         for (i = 1; i <= n ; i++) {
798                :                 BUFFER_TRACE(branch[i].bh, "call jbd2_journal_forget");
799              0 :                 ext4_journal_forget(handle, branch[i].bh);
800                :         }
801              0 :         for (i = 0; i < indirect_blks; i++)
802              0 :                 ext4_free_blocks(handle, inode, new_blocks[i], 1, 0);
803                :
804              0 :         ext4_free_blocks(handle, inode, new_blocks[i], num, 0);
805                :
806              0 :         return err;
807                : }
808                :
809                : /**
810                :  * ext4_splice_branch - splice the allocated branch onto inode.
811                :  * @inode: owner
812                :  * @block: (logical) number of block we are adding
813                :  * @chain: chain of indirect blocks (with a missing link - see
814                :  *      ext4_alloc_branch)
815                :  * @where: location of missing link
816                :  * @num:   number of indirect blocks we are adding
817                :  * @blks:  number of direct blocks we are adding
818                :  *
819                :  * This function fills the missing link and does all housekeeping needed in
820                :  * inode (->i_blocks, etc.). In case of success we end up with the full
821                :  * chain to new block and return 0.
822                :  */
823                : static int ext4_splice_branch(handle_t *handle, struct inode *inode,
824                :                               ext4_lblk_t block, Indirect *where, int num,
825                :                               int blks)
826         259108 : {
827                :         int i;
828         259108 :         int err = 0;
829                :         ext4_fsblk_t current_block;
830                :
831                :         /*
832                :          * If we're splicing into a [td]indirect block (as opposed to the
833                :          * inode) then we need to get write access to the [td]indirect block
834                :          * before the splice.
835                :          */
836         259108 :         if (where->bh) {
837                :                 BUFFER_TRACE(where->bh, "get_write_access");
838          78057 :                 err = ext4_journal_get_write_access(handle, where->bh);
839          78135 :                 if (err)
```

```
840              0 :                         goto err_out;
841                :         }
842                :         /* That's it */
843                :
844         259186 :         *where->p = where->key;
845                :
846                :         /*
847                :          * Update the host buffer_head or inode to point to more just allocated
848                :          * direct blocks blocks
849                :          */
850         259186 :         if (num == 0 && blks > 1) {
851         144212 :                 current_block = le32_to_cpu(where->key) + 1;
852        1394007 :                 for (i = 1; i < blks; i++)
853        1249795 :                         *(where->p + i) = cpu_to_le32(current_block++);
854                :         }
855                :
856                :         /* We are done with atomic stuff, now do the rest of housekeeping */
857                :         /* had we spliced it onto indirect block? */
858         259186 :         if (where->bh) {
859                :                 /*
860                :                  * If we spliced it onto an indirect block, we haven't
861                :                  * altered the inode.  Note however that if it is being spliced
862                :                  * onto an indirect block at the very end of the file (the
863                :                  * file is growing) then we *will* alter the inode to reflect
864                :                  * the new i_size.  But that is not done here - it is done in
865                :                  * generic_commit_write->__mark_inode_dirty->ext4_dirty_inode.
866                :                  */
867          78152 :                 jbd_debug(5, "splicing indirect only\n");
868                :                 BUFFER_TRACE(where->bh, "call ext4_handle_dirty_metadata");
869          78152 :                 err = ext4_handle_dirty_metadata(handle, inode, where->bh);
870          78146 :                 if (err)
871              0 :                         goto err_out;
872                :         } else {
873                :                 /*
874                :                  * OK, we spliced it into the inode itself on a direct block.
875                :                  */
876         181034 :                 ext4_mark_inode_dirty(handle, inode);
877         181034 :                 jbd_debug(5, "splicing direct\n");
878                :         }
879         259180 :         return err;
880                :
881              0 : err_out:
882              0 :         for (i = 1; i <= num; i++) {
883                :                 BUFFER_TRACE(where[i].bh, "call jbd2_journal_forget");
884              0 :                 ext4_journal_forget(handle, where[i].bh);
885              0 :                 ext4_free_blocks(handle, inode,
886                :                                  le32_to_cpu(where[i-1].key), 1, 0);
887                :         }
888              0 :         ext4_free_blocks(handle, inode, le32_to_cpu(where[num].key), blks, 0);
889                :
890              0 :         return err;
891                : }
892                :
893                : /*
894                :  * The ext4_ind_get_blocks() function handles non-extents inodes
895                :  * (i.e., using the traditional indirect/double-indirect i_blocks
896                :  * scheme) for ext4_get_blocks().
897                :  *
898                :  * Allocation strategy is simple: if we have to allocate something, we will
899                :  * have to go the whole way to leaf. So let's do it before attaching anything
900                :  * to tree, set linkage between the newborn blocks, write them if sync is
901                :  * required, recheck the path, free and repeat if check fails, otherwise
902                :  * set the last missing link (that will protect us from any truncate-generated
903                :  * removals - all blocks on the path are immune now) and possibly force the
904                :  * write on the parent block.
905                :  * That has a nice additional property: no special recovery from the failed
906                :  * allocations is needed - we simply release blocks and do not touch anything
907                :  * reachable from inode.
908                :  *
909                :  * `handle' can be NULL if create == 0.
910                :  *
```

```
   911                :  * return > 0, # of blocks mapped or allocated.
   912                :  * return = 0, if plain lookup failed.
   913                :  * return < 0, error case.
   914                :  *
   915                :  * The ext4_ind_get_blocks() function should be called with
   916                :  * down_write(&EXT4_I(inode)->i_data_sem) if allocating filesystem
   917                :  * blocks (i.e., flags has EXT4_GET_BLOCKS_CREATE set) or
   918                :  * down_read(&EXT4_I(inode)->i_data_sem) if not allocating file system
   919                :  * blocks.
   920                :  */
   921                : static int ext4_ind_get_blocks(handle_t *handle, struct inode *inode,
   922                :                                ext4_lblk_t iblock, unsigned int maxblocks,
   923                :                                struct buffer_head *bh_result,
   924                :                                int flags)
   925        6650714 : {
   926        6650714 :         int err = -EIO;
   927                :         ext4_lblk_t offsets[4];
   928                :         Indirect chain[4];
   929                :         Indirect *partial;
   930                :         ext4_fsblk_t goal;
   931                :         int indirect_blks;
   932        6650714 :         int blocks_to_boundary = 0;
   933                :         int depth;
   934        6650714 :         int count = 0;
   935        6650714 :         ext4_fsblk_t first_block = 0;
   936                :
   937        6650714 :         J_ASSERT(!(EXT4_I(inode)->i_flags & EXT4_EXTENTS_FL));
   938        6650749 :         J_ASSERT(handle != NULL || (flags & EXT4_GET_BLOCKS_CREATE) == 0);
   939        6650827 :         depth = ext4_block_to_path(inode, iblock, offsets,
   940                :                                    &blocks_to_boundary);
   941                :
   942        6650834 :         if (depth == 0)
   943              0 :                 goto out;
   944                :
   945        6650834 :         partial = ext4_get_branch(inode, depth, offsets, chain, &err);
   946                :
   947                :         /* Simplest case - block found, no allocation needed */
   948        6650942 :         if (!partial) {
   949        3764530 :                 first_block = le32_to_cpu(chain[depth - 1].key);
   950                :                 clear_buffer_new(bh_result);
   951        3764529 :                 count++;
   952                :                 /*map more blocks*/
   953        7624992 :                 while (count < maxblocks && count <= blocks_to_boundary) {
   954                :                         ext4_fsblk_t blk;
   955                :
   956          95934 :                         blk = le32_to_cpu(*(chain[depth-1].p + count));
   957                :
   958          95934 :                         if (blk == first_block + count)
   959          95934 :                                 count++;
   960                :                         else
   961              0 :                                 break;
   962                :                 }
   963                :                 goto got_it;
   964                :         }
   965                :
   966                :         /* Next simple case - plain lookup or failed read of indirect block */
   967        2886412 :         if ((flags & EXT4_GET_BLOCKS_CREATE) == 0 || err == -EIO)
   968                :                 goto cleanup;
   969                :
   970                :         /*
   971                :          * Okay, we need to do block allocation.
   972                :          */
   973         259184 :         goal = ext4_find_goal(inode, iblock, partial);
   974                :
   975                :         /* the number of blocks need to allocate for [d,t]indirect blocks */
   976         259184 :         indirect_blks = (chain + depth) - partial - 1;
   977                :
   978                :         /*
   979                :          * Next look up the indirect map to count the totoal number of
   980                :          * direct blocks to allocate for this branch.
   981                :          */
```

```
    982          518368 :          count = ext4_blks_to_allocate(partial, indirect_blks,
    983                 :                                        maxblocks, blocks_to_boundary);
    984                 :          /*
    985                 :           * Block out ext4_truncate while we alter the tree
    986                 :           */
    987          259184 :          err = ext4_alloc_branch(handle, inode, iblock, indirect_blks,
    988                 :                                  &count, goal,
    989                 :                                  offsets + (partial - chain), partial);
    990                 :
    991                 :          /*
    992                 :           * The ext4_splice_branch call will free and forget any buffers
    993                 :           * on the new chain if there is a failure, but that risks using
    994                 :           * up transaction credits, especially for bitmaps where the
    995                 :           * credits cannot be returned.  Can we handle this somehow?  We
    996                 :           * may need to return -EAGAIN upwards in the worst case.  --sct
    997                 :           */
    998          259088 :          if (!err)
    999          259088 :                  err = ext4_splice_branch(handle, inode, iblock,
   1000                 :                                           partial, indirect_blks, count);
   1001                 :          else
   1002               0 :                  goto cleanup;
   1003                 :
   1004                 :          set_buffer_new(bh_result);
   1005         4023715 : got_it:
   1006         4023715 :          map_bh(bh_result, inode->i_sb, le32_to_cpu(chain[depth-1].key));
   1007         4023699 :          if (count > blocks_to_boundary)
   1008                 :                  set_buffer_boundary(bh_result);
   1009         4023699 :          err = count;
   1010                 :          /* Clean up and exit */
   1011         4023699 :          partial = chain + depth - 1;    /* the whole chain */
   1012                 : cleanup:
   1013        10835664 :          while (partial > chain) {
   1014                 :                  BUFFER_TRACE(partial->bh, "call brelse");
   1015         4184638 :                  brelse(partial->bh);
   1016         4184737 :                  partial--;
   1017                 :          }
   1018                 :          BUFFER_TRACE(bh_result, "returned");
   1019         6651026 : out:
   1020         6651026 :          return err;
   1021                 : }
   1022                 :
   1023                 : qsize_t ext4_get_reserved_space(struct inode *inode)
   1024               0 : {
   1025                 :          unsigned long long total;
   1026                 :
   1027               0 :          spin_lock(&EXT4_I(inode)->i_block_reservation_lock);
   1028               0 :          total = EXT4_I(inode)->i_reserved_data_blocks +
   1029                 :                  EXT4_I(inode)->i_reserved_meta_blocks;
   1030               0 :          spin_unlock(&EXT4_I(inode)->i_block_reservation_lock);
   1031                 :
   1032               0 :          return total;
   1033                 : }
   1034                 : /*
   1035                 :  * Calculate the number of metadata blocks need to reserve
   1036                 :  * to allocate @blocks for non extent file based file
   1037                 :  */
   1038                 : static int ext4_indirect_calc_metadata_amount(struct inode *inode, int blocks)
   1039                 : {
   1040         2458088 :          int icap = EXT4_ADDR_PER_BLOCK(inode->i_sb);
   1041                 :          int ind_blks, dind_blks, tind_blks;
   1042                 :
   1043                 :          /* number of new indirect blocks needed */
   1044         2458088 :          ind_blks = (blocks + icap - 1) / icap;
   1045                 :
   1046         2458088 :          dind_blks = (ind_blks + icap - 1) / icap;
   1047                 :
   1048         2458088 :          tind_blks = 1;
   1049                 :
   1050         2458088 :          return ind_blks + dind_blks + tind_blks;
   1051                 : }
   1052                 :
```

```
1053                 : /*
1054                 :  * Calculate the number of metadata blocks need to reserve
1055                 :  * to allocate given number of blocks
1056                 :  */
1057                 : static int ext4_calc_metadata_amount(struct inode *inode, int blocks)
1058       803755250 : {
1059       803755250 :         if (!blocks)
1060        11696313 :                 return 0;
1061                 :
1062       792058937 :         if (EXT4_I(inode)->i_flags & EXT4_EXTENTS_FL)
1063       789600849 :                 return ext4_ext_calc_metadata_amount(inode, blocks);
1064                 :
1065         2458088 :         return ext4_indirect_calc_metadata_amount(inode, blocks);
1066                 : }
1067                 :
1068                 : static void ext4_da_update_reserve_space(struct inode *inode, int used)
1069        13817635 : {
1070        27635270 :         struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);
1071                 :         int total, mdb, mdb_free;
1072                 :
1073        13817635 :         spin_lock(&EXT4_I(inode)->i_block_reservation_lock);
1074                 :         /* recalculate the number of metablocks still need to be reserved */
1075        13821107 :         total = EXT4_I(inode)->i_reserved_data_blocks - used;
1076        13821107 :         mdb = ext4_calc_metadata_amount(inode, total);
1077                 :
1078                 :         /* figure out how many metablocks to release */
1079        27639876 :         BUG_ON(mdb > EXT4_I(inode)->i_reserved_meta_blocks);
1080        13824088 :         mdb_free = EXT4_I(inode)->i_reserved_meta_blocks - mdb;
1081                 :
1082        13824088 :         if (mdb_free) {
1083                 :                 /* Account for allocated meta_blocks */
1084        24172178 :                 mdb_free -= EXT4_I(inode)->i_allocated_meta_blocks;
1085                 :
1086                 :                 /* update fs dirty blocks counter */
1087        12086089 :                 percpu_counter_sub(&sbi->s_dirtyblocks_counter, mdb_free);
1088        12085938 :                 EXT4_I(inode)->i_allocated_meta_blocks = 0;
1089        12085938 :                 EXT4_I(inode)->i_reserved_meta_blocks = mdb;
1090                 :         }
1091                 :
1092                 :         /* update per-inode reservations */
1093        27647874 :         BUG_ON(used  > EXT4_I(inode)->i_reserved_data_blocks);
1094        13816431 :         EXT4_I(inode)->i_reserved_data_blocks -= used;
1095        13816431 :         spin_unlock(&EXT4_I(inode)->i_block_reservation_lock);
1096                 :
1097                 :         /*
1098                 :          * free those over-booking quota for metadata blocks
1099                 :          */
1100        13825611 :         if (mdb_free)
1101        12087749 :                 vfs_dq_release_reservation_block(inode, mdb_free);
1102                 :
1103                 :         /*
1104                 :          * If we have done all the pending block allocations and if
1105                 :          * there aren't any writers on the inode, we can discard the
1106                 :          * inode's preallocations.
1107                 :          */
1108        25523606 :         if (!total && (atomic_read(&inode->i_writecount) == 0))
1109        11690626 :                 ext4_discard_preallocations(inode);
1110        13823617 : }
1111                 :
1112                 : static int check_block_validity(struct inode *inode, sector_t logical,
1113                 :                                 sector_t phys, int len)
1114       321742558 : {
1115       643485116 :         if (!ext4_data_block_valid(EXT4_SB(inode->i_sb), phys, len)) {
1116                 :                 ext4_error(inode->i_sb, "check_block_validity",
1117                 :                            "inode #%lu logical block %llu mapped to %llu "
1118                 :                            "(size %d)", inode->i_ino,
1119                 :                            (unsigned long long) logical,
1120                 :                            (unsigned long long) phys, len);
1121                 :                 WARN_ON(1);
1122                 :                 return -EIO;
1123                 :         }
```

```
1124  321685547 :         return 0;
1125            : }
1126            :
1127            : /*
1128            :  * The ext4_get_blocks() function tries to look up the requested blocks,
1129            :  * and returns if the blocks are already mapped.
1130            :  *
1131            :  * Otherwise it takes the write lock of the i_data_sem and allocate blocks
1132            :  * and store the allocated blocks in the result buffer head and mark it
1133            :  * mapped.
1134            :  *
1135            :  * If file type is extents based, it will call ext4_ext_get_blocks(),
1136            :  * Otherwise, call with ext4_ind_get_blocks() to handle indirect mapping
1137            :  * based files
1138            :  *
1139            :  * On success, it returns the number of blocks being mapped or allocate.
1140            :  * if create==0 and the blocks are pre-allocated and uninitialized block,
1141            :  * the result buffer head is unmapped. If the create ==1, it will make sure
1142            :  * the buffer head is mapped.
1143            :  *
1144            :  * It returns 0 if plain look up failed (blocks have not been allocated), in
1145            :  * that casem, buffer head is unmapped
1146            :  *
1147            :  * It returns the error in case of allocation failure.
1148            :  */
1149            : int ext4_get_blocks(handle_t *handle, struct inode *inode, sector_t block,
1150            :                     unsigned int max_blocks, struct buffer_head *bh,
1151            :                     int flags)
1152 1109850276 : {
1153            :         int retval;
1154            :
1155            :         clear_buffer_mapped(bh);
1156            :         clear_buffer_unwritten(bh);
1157            :
1158            :         /*
1159            :          * Try to see if we can get the block without requesting a new
1160            :          * file system block.
1161            :          */
1162 1133410505 :         down_read((&EXT4_I(inode)->i_data_sem));
1163 1112394515 :         if (EXT4_I(inode)->i_flags & EXT4_EXTENTS_FL) {
1164 1106002855 :                 retval =  ext4_ext_get_blocks(handle, inode, block, max_blocks,
1165            :                                 bh, 0);
1166            :         } else {
1167    6391660 :                 retval = ext4_ind_get_blocks(handle, inode, block, max_blocks,
1168            :                                              bh, 0);
1169            :         }
1170 1123736876 :         up_read((&EXT4_I(inode)->i_data_sem));
1171            :
1172 1120754113 :         if (retval > 0 && buffer_mapped(bh)) {
1173            :                 int ret = check_block_validity(inode, block,
1174  102237174 :                                                bh->b_blocknr, retval);
1175  102235278 :                 if (ret != 0)
1176          0 :                         return ret;
1177            :         }
1178            :
1179            :         /* If it is only a block(s) look up */
1180 1120752234 :         if ((flags & EXT4_GET_BLOCKS_CREATE) == 0)
1181  900895350 :                 return retval;
1182            :
1183            :         /*
1184            :          * Returns if the blocks have already allocated
1185            :          *
1186            :          * Note that if blocks have been preallocated
1187            :          * ext4_ext_get_block() returns th create = 0
1188            :          * with buffer head unmapped.
1189            :          */
1190  219856884 :         if (retval > 0 && buffer_mapped(bh))
1191     435234 :                 return retval;
1192            :
1193            :         /*
1194            :          * When we call get_blocks without the create flag, the
```

```
1195                :          * BH_Unwritten flag could have gotten set if the blocks
1196                :          * requested were part of a uninitialized extent.  We need to
1197                :          * clear this flag now that we are committed to convert all or
1198                :          * part of the uninitialized extent to be an initialized
1199                :          * extent.  This is because we need to avoid the combination
1200                :          * of BH_Unwritten and BH_Mapped flags being simultaneously
1201                :          * set on the buffer_head.
1202                :          */
1203                :         clear_buffer_unwritten(bh);
1204                :
1205                :         /*
1206                :          * New blocks allocate and/or writing to uninitialized extent
1207                :          * will possibly result in updating i_data, so we take
1208                :          * the write lock of i_data_sem, and call get_blocks()
1209                :          * with create == 1 flag.
1210                :          */
1211     219831465 :         down_write((&EXT4_I(inode)->i_data_sem));
1212                :
1213                :         /*
1214                :          * if the caller is from delayed allocation writeout path
1215                :          * we have already reserved fs blocks for allocation
1216                :          * let the underlying get_block() function know to
1217                :          * avoid double accounting
1218                :          */
1219     221470135 :         if (flags & EXT4_GET_BLOCKS_DELALLOC_RESERVE)
1220      13826254 :                 EXT4_I(inode)->i_delalloc_reserved_flag = 1;
1221                :         /*
1222                :          * We need to check for EXT4 here because migrate
1223                :          * could have changed the inode type in between
1224                :          */
1225     221470135 :         if (EXT4_I(inode)->i_flags & EXT4_EXTENTS_FL) {
1226     221210950 :                 retval =  ext4_ext_get_blocks(handle, inode, block, max_blocks,
1227                :                                               bh, flags);
1228                :         } else {
1229        259185 :                 retval = ext4_ind_get_blocks(handle, inode, block,
1230                :                                              max_blocks, bh, flags);
1231                :
1232        518321 :                 if (retval > 0 && buffer_new(bh)) {
1233                :                         /*
1234                :                          * We allocated new blocks which will result in
1235                :                          * i_data's format changing.  Force the migrate
1236                :                          * to fail by clearing migrate flags
1237                :                          */
1238        259161 :                         EXT4_I(inode)->i_flags = EXT4_I(inode)->i_flags &
1239                :                                                  ~EXT4_EXT_MIGRATE;
1240                :                 }
1241                :         }
1242                :
1243     219787207 :         if (flags & EXT4_GET_BLOCKS_DELALLOC_RESERVE)
1244      13816518 :                 EXT4_I(inode)->i_delalloc_reserved_flag = 0;
1245                :
1246                :         /*
1247                :          * Update reserved blocks/metadata blocks after successful
1248                :          * block allocation which had been deferred till now.
1249                :          */
1250     219787207 :         if ((retval > 0) && (flags & EXT4_GET_BLOCKS_UPDATE_RESERVE_SPACE))
1251      13820044 :                 ext4_da_update_reserve_space(inode, retval);
1252                :
1253     219793060 :         up_write((&EXT4_I(inode)->i_data_sem));
1254     220275378 :         if (retval > 0 && buffer_mapped(bh)) {
1255                :                 int ret = check_block_validity(inode, block,
1256     221117832 :                                                bh->b_blocknr, retval);
1257     219745167 :                 if (ret != 0)
1258             0 :                         return ret;
1259                :         }
1260     220201119 :         return retval;
1261                : }
1262                :
1263                : /* Maximum number of blocks we map for direct IO at once. */
1264                : #define DIO_MAX_BLOCKS 4096
1265                :
```

```
1266                :int ext4_get_block(struct inode *inode, sector_t iblock,
1267                :                   struct buffer_head *bh_result, int create)
1268       243817678 :{
1269       243817678 :        handle_t *handle = ext4_journal_current_handle();
1270       243817678 :        int ret = 0, started = 0;
1271       243817678 :        unsigned max_blocks = bh_result->b_size >> inode->i_blkbits;
1272                 :        int dio_credits;
1273                 :
1274       243817678 :        if (create && !handle) {
1275                 :                /* Direct IO write... */
1276          276738 :                if (max_blocks > DIO_MAX_BLOCKS)
1277               0 :                        max_blocks = DIO_MAX_BLOCKS;
1278          276738 :                dio_credits = ext4_chunk_trans_blocks(inode, max_blocks);
1279          276738 :                handle = ext4_journal_start(inode, dio_credits);
1280          276738 :                if (IS_ERR(handle)) {
1281               0 :                        ret = PTR_ERR(handle);
1282               0 :                        goto out;
1283                 :                }
1284          276738 :                started = 1;
1285                 :        }
1286                 :
1287       243817678 :        ret = ext4_get_blocks(handle, inode, iblock, max_blocks, bh_result,
1288                 :                              create ? EXT4_GET_BLOCKS_CREATE : 0);
1289       244869409 :        if (ret > 0) {
1290       245168079 :                bh_result->b_size = (ret << inode->i_blkbits);
1291       245168079 :                ret = 0;
1292                 :        }
1293       244869409 :        if (started)
1294          276732 :                ext4_journal_stop(handle);
1295       244869415 :out:
1296       244869415 :        return ret;
1297                 :}
1298                 :
1299                 :/*
1300                 : * `handle' can be NULL if create is zero
1301                 : */
1302                 :struct buffer_head *ext4_getblk(handle_t *handle, struct inode *inode,
1303                 :                                ext4_lblk_t block, int create, int *errp)
1304        60706071 :{
1305                 :        struct buffer_head dummy;
1306        60706071 :        int fatal = 0, err;
1307        60706071 :        int flags = 0;
1308                 :
1309        60706071 :        J_ASSERT(handle != NULL || create == 0);
1310                 :
1311        60706222 :        dummy.b_state = 0;
1312        60706222 :        dummy.b_blocknr = -1000;
1313                 :        buffer_trace_init(&dummy.b_history);
1314        60706222 :        if (create)
1315          847582 :                flags |= EXT4_GET_BLOCKS_CREATE;
1316        60706222 :        err = ext4_get_blocks(handle, inode, block, 1, &dummy, flags);
1317                 :        /*
1318                 :         * ext4_get_blocks() returns number of blocks mapped. 0 in
1319                 :         * case of a HOLE.
1320                 :         */
1321        60706925 :        if (err > 0) {
1322        60706733 :                if (err > 1)
1323               0 :                        WARN_ON(1);
1324        60706129 :                err = 0;
1325                 :        }
1326        60706321 :        *errp = err;
1327        60706321 :        if (!err && buffer_mapped(&dummy)) {
1328                 :                struct buffer_head *bh;
1329       121413615 :                bh = sb_getblk(inode->i_sb, dummy.b_blocknr);
1330        60706676 :                if (!bh) {
1331               0 :                        *errp = -EIO;
1332               0 :                        goto err;
1333                 :                }
1334        60706676 :                if (buffer_new(&dummy)) {
1335          847582 :                        J_ASSERT(create != 0);
1336          847582 :                        J_ASSERT(handle != NULL);
```

```
1337                :
1338                :                         /*
1339                :                          * Now that we do not always journal data, we should
1340                :                          * keep in mind whether this should always journal the
1341                :                          * new buffer as metadata.  For now, regular file
1342                :                          * writes use ext4_get_block instead, so it's not a
1343                :                          * problem.
1344                :                          */
1345         847582 :                         lock_buffer(bh);
1346                :                         BUFFER_TRACE(bh, "call get_create_access");
1347         847582 :                         fatal = ext4_journal_get_create_access(handle, bh);
1348         847582 :                         if (!fatal && !buffer_uptodate(bh)) {
1349        1695054 :                                 memset(bh->b_data, 0, inode->i_sb->s_blocksize);
1350                :                                 set_buffer_uptodate(bh);
1351                :                         }
1352         847582 :                         unlock_buffer(bh);
1353                :                         BUFFER_TRACE(bh, "call ext4_handle_dirty_metadata");
1354         847582 :                         err = ext4_handle_dirty_metadata(handle, inode, bh);
1355         847582 :                         if (!fatal)
1356         847582 :                                 fatal = err;
1357                :                 } else {
1358                :                         BUFFER_TRACE(bh, "not a new buffer");
1359                :                 }
1360       60706676 :                 if (fatal) {
1361              0 :                         *errp = fatal;
1362              0 :                         brelse(bh);
1363              0 :                         bh = NULL;
1364                :                 }
1365       60706676 :                 return bh;
1366                :         }
1367              0 : err:
1368              0 :         return NULL;
1369                : }
1370                :
1371                : struct buffer_head *ext4_bread(handle_t *handle, struct inode *inode,
1372                :                                ext4_lblk_t block, int create, int *err)
1373       60314460 : {
1374                :         struct buffer_head *bh;
1375                :
1376       60314460 :         bh = ext4_getblk(handle, inode, block, create, err);
1377       60315229 :         if (!bh)
1378              0 :                 return bh;
1379       60315229 :         if (buffer_uptodate(bh))
1380       60308687 :                 return bh;
1381           6153 :         ll_rw_block(READ_META, 1, &bh);
1382           6153 :         wait_on_buffer(bh);
1383           6153 :         if (buffer_uptodate(bh))
1384           6153 :                 return bh;
1385              0 :         put_bh(bh);
1386              0 :         *err = -EIO;
1387              0 :         return NULL;
1388                : }
1389                :
1390                : static int walk_page_buffers(handle_t *handle,
1391                :                              struct buffer_head *head,
1392                :                              unsigned from,
1393                :                              unsigned to,
1394                :                              int *partial,
1395                :                              int (*fn)(handle_t *handle,
1396                :                                        struct buffer_head *bh))
1397     1374884053 : {
1398                :         struct buffer_head *bh;
1399                :         unsigned block_start, block_end;
1400     1374884053 :         unsigned blocksize = head->b_size;
1401     1374884053 :         int err, ret = 0;
1402                :         struct buffer_head *next;
1403                :
1404     1374884053 :         for (bh = head, block_start = 0;
1405     -156905855 :              ret == 0 && (bh != head || !block_start);
1406     1388293335 :              block_start = block_end, bh = next) {
1407     1388178007 :                 next = bh->b_this_page;
```

```
1408  1388178007 :                 block_end = block_start + blocksize;
1409  1388178007 :                 if (block_end <= from || block_start >= to) {
1410     1261159 :                         if (partial && !buffer_uptodate(bh))
1411           0 :                                 *partial = 1;
1412             :                         continue;
1413             :                 }
1414  1386916848 :                 err = (*fn)(handle, bh);
1415  1387032176 :                 if (!ret)
1416  1387040567 :                         ret = err;
1417             :         }
1418  1374999381 :         return ret;
1419             : }
1420             :
1421             : /*
1422             :  * To preserve ordering, it is essential that the hole instantiation and
1423             :  * the data write be encapsulated in a single transaction.  We cannot
1424             :  * close off a transaction and start a new one between the ext4_get_block()
1425             :  * and the commit_write().  So doing the jbd2_journal_start at the start of
1426             :  * prepare_write() is the right place.
1427             :  *
1428             :  * Also, this function can nest inside ext4_writepage() ->
1429             :  * block_write_full_page(). In that case, we *know* that ext4_writepage()
1430             :  * has generated enough buffer credits to do the whole page.  So we won't
1431             :  * block on the journal in that case, which is good, because the caller may
1432             :  * be PF_MEMALLOC.
1433             :  *
1434             :  * By accident, ext4 can be reentered when a transaction is open via
1435             :  * quota file writes.  If we were to commit the transaction while thus
1436             :  * reentered, there can be a deadlock - we would be holding a quota
1437             :  * lock, and the commit would never complete if another thread had a
1438             :  * transaction open and was blocking on the quota lock - a ranking
1439             :  * violation.
1440             :  *
1441             :  * So what we do is to rely on the fact that jbd2_journal_stop/journal_start
1442             :  * will _not_ run commit under these circumstances because handle->h_ref
1443             :  * is elevated.  We'll still have enough credits for the tiny quotafile
1444             :  * write.
1445             :  */
1446             : static int do_journal_get_write_access(handle_t *handle,
1447             :                                        struct buffer_head *bh)
1448     8567696 : {
1449    17200154 :         if (!buffer_mapped(bh) || buffer_freed(bh))
1450           0 :                 return 0;
1451     8636748 :         return ext4_journal_get_write_access(handle, bh);
1452             : }
1453             :
1454             : static int ext4_write_begin(struct file *file, struct address_space *mapping,
1455             :                             loff_t pos, unsigned len, unsigned flags,
1456             :                             struct page **pagep, void **fsdata)
1457   208425048 : {
1458   208425048 :         struct inode *inode = mapping->host;
1459             :         int ret, needed_blocks;
1460             :         handle_t *handle;
1461   208425048 :         int retries = 0;
1462             :         struct page *page;
1463             :         pgoff_t index;
1464             :         unsigned from, to;
1465             :
1466             :         trace_ext4_write_begin(inode, pos, len, flags);
1467             :         /*
1468             :          * Reserve one block more for addition to orphan list in case
1469             :          * we allocate blocks but write fails for some reason
1470             :          */
1471   208596572 :         needed_blocks = ext4_writepage_trans_blocks(inode) + 1;
1472   208100693 :         index = pos >> PAGE_CACHE_SHIFT;
1473   208100693 :         from = pos & (PAGE_CACHE_SIZE - 1);
1474   208100693 :         to = from + len;
1475             :
1476   208100693 : retry:
1477   211238390 :         handle = ext4_journal_start(inode, needed_blocks);
1478   211259676 :         if (IS_ERR(handle)) {
```

```
1479          0 :                 ret = PTR_ERR(handle);
1480          0 :                 goto out;
1481            :         }
1482            :
1483            :         /* We cannot recurse into the filesystem as the transaction is already
1484            :          * started */
1485  211259676 :         flags |= AOP_FLAG_NOFS;
1486            :
1487  211259676 :         page = grab_cache_page_write_begin(mapping, index, flags);
1488  209474812 :         if (!page) {
1489          0 :                 ext4_journal_stop(handle);
1490          0 :                 ret = -ENOMEM;
1491          0 :                 goto out;
1492            :         }
1493  209474812 :         *pagep = page;
1494            :
1495  209474812 :         ret = block_write_begin(file, mapping, pos, len, flags, pagep, fsdata,
1496            :                                 ext4_get_block);
1497            :
1498  418456946 :         if (!ret && ext4_should_journal_data(inode)) {
1499    8572986 :                 ret = walk_page_buffers(handle, page_buffers(page),
1500            :                                 from, to, NULL, do_journal_get_write_access);
1501            :         }
1502            :
1503  209066063 :         if (ret) {
1504         10 :                 unlock_page(page);
1505         10 :                 page_cache_release(page);
1506            :                 /*
1507            :                  * block_write_begin may have instantiated a few blocks
1508            :                  * outside i_size.  Trim these off again. Don't need
1509            :                  * i_size_read because we hold i_mutex.
1510            :                  *
1511            :                  * Add inode to orphan list in case we crash before
1512            :                  * truncate finishes
1513            :                  */
1514         10 :                 if (pos + len > inode->i_size && ext4_can_truncate(inode))
1515         10 :                         ext4_orphan_add(handle, inode);
1516            :
1517         10 :                 ext4_journal_stop(handle);
1518         10 :                 if (pos + len > inode->i_size) {
1519         10 :                         ext4_truncate(inode);
1520            :                         /*
1521            :                          * If truncate failed early the inode might
1522            :                          * still be on the orphan list; we need to
1523            :                          * make sure the inode is removed from the
1524            :                          * orphan list in that case.
1525            :                          */
1526         10 :                         if (inode->i_nlink)
1527         10 :                                 ext4_orphan_del(NULL, inode);
1528            :                 }
1529            :         }
1530            :
1531  208989451 :         if (ret == -ENOSPC && ext4_should_retry_alloc(inode->i_sb, &retries))
1532          0 :                 goto retry;
1533  208989451 : out:
1534  208989451 :         return ret;
1535            : }
1536            :
1537            : /* For write_end() in data=journal mode */
1538            : static int write_end_fn(handle_t *handle, struct buffer_head *bh)
1539    8615928 : {
1540   17256454 :         if (!buffer_mapped(bh) || buffer_freed(bh))
1541          0 :                 return 0;
1542            :         set_buffer_uptodate(bh);
1543    8613610 :         return ext4_handle_dirty_metadata(handle, NULL, bh);
1544            : }
1545            :
1546            : static int ext4_generic_write_end(struct file *file,
1547            :                                   struct address_space *mapping,
1548            :                                   loff_t pos, unsigned len, unsigned copied,
1549            :                                   struct page *page, void *fsdata)
```

```
1550       201142339 : {
1551       201142339 :         int i_size_changed = 0;
1552       201142339 :         struct inode *inode = mapping->host;
1553       201142339 :         handle_t *handle = ext4_journal_current_handle();
1554                 :
1555       201142339 :         copied = block_write_end(file, mapping, pos, len, copied, page, fsdata);
1556                 :
1557                 :         /*
1558                 :          * No need to use i_size_read() here, the i_size
1559                 :          * cannot change under us because we hold i_mutex.
1560                 :          *
1561                 :          * But it's important to update i_size while still holding page lock:
1562                 :          * page writeout could otherwise come in and zero beyond i_size.
1563                 :          */
1564       201853179 :         if (pos + copied > inode->i_size) {
1565       198921982 :                 i_size_write(inode, pos + copied);
1566       199891516 :                 i_size_changed = 1;
1567                 :         }
1568                 :
1569       405645426 :         if (pos + copied >  EXT4_I(inode)->i_disksize) {
1570                 :                 /* We need to mark inode dirty even if
1571                 :                  * new_i_size is less that inode->i_size
1572                 :                  * bu greater than i_disksize.(hint delalloc)
1573                 :                  */
1574       201282430 :                 ext4_update_i_disksize(inode, (pos + copied));
1575       201833963 :                 i_size_changed = 1;
1576                 :         }
1577       203374246 :         unlock_page(page);
1578       202006742 :         page_cache_release(page);
1579                 :
1580                 :         /*
1581                 :          * Don't mark the inode dirty under page lock. First, it unnecessarily
1582                 :          * makes the holding time of page lock longer. Second, it forces lock
1583                 :          * ordering of page lock and transaction start for journaling
1584                 :          * filesystems.
1585                 :          */
1586       201584245 :         if (i_size_changed)
1587       200949751 :                 ext4_mark_inode_dirty(handle, inode);
1588                 :
1589       199322175 :         return copied;
1590                 : }
1591                 :
1592                 : /*
1593                 :  * We need to pick up the new inode size which generic_commit_write gave us
1594                 :  * `file' can be NULL - eg, when called from page_symlink().
1595                 :  *
1596                 :  * ext4 never places buffers on inode->i_mapping->private_list.  metadata
1597                 :  * buffers are managed internally.
1598                 :  */
1599                 : static int ext4_ordered_write_end(struct file *file,
1600                 :                                   struct address_space *mapping,
1601                 :                                   loff_t pos, unsigned len, unsigned copied,
1602                 :                                   struct page *page, void *fsdata)
1603       201575633 : {
1604       201575633 :         handle_t *handle = ext4_journal_current_handle();
1605       201575633 :         struct inode *inode = mapping->host;
1606       201575633 :         int ret = 0, ret2;
1607                 :
1608                 :         trace_ext4_ordered_write_end(inode, pos, len, copied);
1609       202516575 :         ret = ext4_jbd2_file_inode(handle, inode);
1610                 :
1611       202516575 :         if (ret == 0) {
1612       202594706 :                 ret2 = ext4_generic_write_end(file, mapping, pos, len, copied,
1613                 :                                                         page, fsdata);
1614       199471985 :                 copied = ret2;
1615       199471985 :                 if (pos + len > inode->i_size && ext4_can_truncate(inode))
1616                 :                         /* if we have allocated more blocks and copied
1617                 :                          * less. We will have blocks allocated outside
1618                 :                          * inode->i_size. So truncate them
1619                 :                          */
1620             0 :                         ext4_orphan_add(handle, inode);
```

```
1621  199207513 :                 if (ret2 < 0)
1622          0 :                         ret = ret2;
1623            :         }
1624  199129382 :         ret2 = ext4_journal_stop(handle);
1625  202255010 :         if (!ret)
1626  202431780 :                 ret = ret2;
1627            :
1628  202255010 :         if (pos + len > inode->i_size) {
1629          0 :                 ext4_truncate(inode);
1630            :                 /*
1631            :                  * If truncate failed early the inode might still be
1632            :                  * on the orphan list; we need to make sure the inode
1633            :                  * is removed from the orphan list in that case.
1634            :                  */
1635          0 :                 if (inode->i_nlink)
1636          0 :                         ext4_orphan_del(NULL, inode);
1637            :         }
1638            :
1639            :
1640  200015973 :         return ret ? ret : copied;
1641            : }
1642            :
1643            : static int ext4_writeback_write_end(struct file *file,
1644            :                                     struct address_space *mapping,
1645            :                                     loff_t pos, unsigned len, unsigned copied,
1646            :                                     struct page *page, void *fsdata)
1647          0 : {
1648          0 :         handle_t *handle = ext4_journal_current_handle();
1649          0 :         struct inode *inode = mapping->host;
1650          0 :         int ret = 0, ret2;
1651            :
1652            :         trace_ext4_writeback_write_end(inode, pos, len, copied);
1653          0 :         ret2 = ext4_generic_write_end(file, mapping, pos, len, copied,
1654            :                                                         page, fsdata);
1655          0 :         copied = ret2;
1656          0 :         if (pos + len > inode->i_size && ext4_can_truncate(inode))
1657            :                 /* if we have allocated more blocks and copied
1658            :                  * less. We will have blocks allocated outside
1659            :                  * inode->i_size. So truncate them
1660            :                  */
1661          0 :                 ext4_orphan_add(handle, inode);
1662            :
1663          0 :         if (ret2 < 0)
1664          0 :                 ret = ret2;
1665            :
1666          0 :         ret2 = ext4_journal_stop(handle);
1667          0 :         if (!ret)
1668          0 :                 ret = ret2;
1669            :
1670          0 :         if (pos + len > inode->i_size) {
1671          0 :                 ext4_truncate(inode);
1672            :                 /*
1673            :                  * If truncate failed early the inode might still be
1674            :                  * on the orphan list; we need to make sure the inode
1675            :                  * is removed from the orphan list in that case.
1676            :                  */
1677          0 :                 if (inode->i_nlink)
1678          0 :                         ext4_orphan_del(NULL, inode);
1679            :         }
1680            :
1681          0 :         return ret ? ret : copied;
1682            : }
1683            :
1684            : static int ext4_journalled_write_end(struct file *file,
1685            :                                      struct address_space *mapping,
1686            :                                      loff_t pos, unsigned len, unsigned copied,
1687            :                                      struct page *page, void *fsdata)
1688    8633085 : {
1689    8633085 :         handle_t *handle = ext4_journal_current_handle();
1690    8633085 :         struct inode *inode = mapping->host;
1691    8633085 :         int ret = 0, ret2;
```

```
1692  8633085 :          int partial = 0;
1693          :          unsigned from, to;
1694          :          loff_t new_i_size;
1695          :
1696          :          trace_ext4_journalled_write_end(inode, pos, len, copied);
1697  8635484 :          from = pos & (PAGE_CACHE_SIZE - 1);
1698  8635484 :          to = from + len;
1699          :
1700  8635484 :          if (copied < len) {
1701        0 :                  if (!PageUptodate(page))
1702        0 :                          copied = 0;
1703        0 :                  page_zero_new_buffers(page, from+copied, to);
1704          :          }
1705          :
1706  8635484 :          ret = walk_page_buffers(handle, page_buffers(page), from,
1707          :                                  to, &partial, write_end_fn);
1708  8647016 :          if (!partial)
1709          :                  SetPageUptodate(page);
1710  8648749 :          new_i_size = pos + copied;
1711  8648749 :          if (new_i_size > inode->i_size)
1712  6419308 :                  i_size_write(inode, pos+copied);
1713  8648928 :          EXT4_I(inode)->i_state |= EXT4_STATE_JDATA;
1714  8648928 :          if (new_i_size > EXT4_I(inode)->i_disksize) {
1715  6419215 :                  ext4_update_i_disksize(inode, new_i_size);
1716  6391126 :                  ret2 = ext4_mark_inode_dirty(handle, inode);
1717  6393499 :                  if (!ret)
1718  6401147 :                          ret = ret2;
1719          :          }
1720          :
1721  8623212 :          unlock_page(page);
1722  8627666 :          page_cache_release(page);
1723  8619805 :          if (pos + len > inode->i_size && ext4_can_truncate(inode))
1724          :                  /* if we have allocated more blocks and copied
1725          :                   * less. We will have blocks allocated outside
1726          :                   * inode->i_size. So truncate them
1727          :                   */
1728        0 :                  ext4_orphan_add(handle, inode);
1729          :
1730  8619805 :          ret2 = ext4_journal_stop(handle);
1731  8572455 :          if (!ret)
1732  8580533 :                  ret = ret2;
1733  8572455 :          if (pos + len > inode->i_size) {
1734        0 :                  ext4_truncate(inode);
1735          :                  /*
1736          :                   * If truncate failed early the inode might still be
1737          :                   * on the orphan list; we need to make sure the inode
1738          :                   * is removed from the orphan list in that case.
1739          :                   */
1740        0 :                  if (inode->i_nlink)
1741        0 :                          ext4_orphan_del(NULL, inode);
1742          :          }
1743          :
1744  8627193 :          return ret ? ret : copied;
1745          : }
1746          :
1747          : static int ext4_da_reserve_space(struct inode *inode, int nrblocks)
1748  795471652 : {
1749  795471652 :          int retries = 0;
1750 1590943304 :          struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);
1751  795471652 :          unsigned long md_needed, mdblocks, total = 0;
1752          :
1753          :          /*
1754          :           * recalculate the amount of metadata blocks to reserve
1755          :           * in order to allocate nrblocks
1756          :           * worse case is one extent per block
1757          :           */
1758  795471652 : repeat:
1759  795471652 :          spin_lock(&EXT4_I(inode)->i_block_reservation_lock);
1760  800806509 :          total = EXT4_I(inode)->i_reserved_data_blocks + nrblocks;
1761  800806509 :          mdblocks = ext4_calc_metadata_amount(inode, total);
1762  792541518 :          BUG_ON(mdblocks < EXT4_I(inode)->i_reserved_meta_blocks);
```

```
1763                :
1764      790244735 :         md_needed = mdblocks - EXT4_I(inode)->i_reserved_meta_blocks;
1765      790244735 :         total = md_needed + nrblocks;
1766                :
1767                :         /*
1768                :          * Make quota reservation here to prevent quota overflow
1769                :          * later. Real quota accounting is done at pages writeout
1770                :          * time.
1771                :          */
1772     1600993658 :         if (vfs_dq_reserve_block(inode, total)) {
1773              0 :                 spin_unlock(&EXT4_I(inode)->i_block_reservation_lock);
1774              0 :                 return -EDQUOT;
1775                :         }
1776                :
1777      810748923 :         if (ext4_claim_free_blocks(sbi, total)) {
1778              0 :                 spin_unlock(&EXT4_I(inode)->i_block_reservation_lock);
1779              0 :                 if (ext4_should_retry_alloc(inode->i_sb, &retries)) {
1780              0 :                         yield();
1781              0 :                         goto repeat;
1782                :                 }
1783              0 :                 vfs_dq_release_reservation_block(inode, total);
1784              0 :                 return -ENOSPC;
1785                :         }
1786      790149584 :         EXT4_I(inode)->i_reserved_data_blocks += nrblocks;
1787      790149584 :         EXT4_I(inode)->i_reserved_meta_blocks = mdblocks;
1788                :
1789      790149584 :         spin_unlock(&EXT4_I(inode)->i_block_reservation_lock);
1790      811935531 :         return 0;       /* success */
1791                : }
1792                :
1793                : static void ext4_da_release_space(struct inode *inode, int to_free)
1794       16844235 : {
1795       33688470 :         struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);
1796                :         int total, mdb, mdb_free, release;
1797                :
1798       16844235 :         if (!to_free)
1799       16635936 :                 return;         /* Nothing to release, exit */
1800                :
1801         208299 :         spin_lock(&EXT4_I(inode)->i_block_reservation_lock);
1802                :
1803         208311 :         if (!EXT4_I(inode)->i_reserved_data_blocks) {
1804                :                 /*
1805                :                  * if there is no reserved blocks, but we try to free some
1806                :                  * then the counter is messed up somewhere.
1807                :                  * but since this function is called from invalidate
1808                :                  * page, it's harmless to return without any action
1809                :                  */
1810              0 :                 printk(KERN_INFO "ext4 delalloc try to release %d reserved "
1811                :                                 "blocks for inode %lu, but there is no reserved "
1812                :                                 "data blocks\n", to_free, inode->i_ino);
1813              0 :                 spin_unlock(&EXT4_I(inode)->i_block_reservation_lock);
1814                :                 return;
1815                :         }
1816                :
1817                :         /* recalculate the number of metablocks still need to be reserved */
1818         208311 :         total = EXT4_I(inode)->i_reserved_data_blocks - to_free;
1819         208311 :         mdb = ext4_calc_metadata_amount(inode, total);
1820                :
1821                :         /* figure out how many metablocks to release */
1822         416634 :         BUG_ON(mdb > EXT4_I(inode)->i_reserved_meta_blocks);
1823         208319 :         mdb_free = EXT4_I(inode)->i_reserved_meta_blocks - mdb;
1824                :
1825         208319 :         release = to_free + mdb_free;
1826                :
1827                :         /* update fs dirty blocks counter for truncate case */
1828         208319 :         percpu_counter_sub(&sbi->s_dirtyblocks_counter, release);
1829                :
1830                :         /* update per-inode reservations */
1831         416630 :         BUG_ON(to_free > EXT4_I(inode)->i_reserved_data_blocks);
1832         208316 :         EXT4_I(inode)->i_reserved_data_blocks -= to_free;
1833                :
```

```
1834     416632 :         BUG_ON(mdb > EXT4_I(inode)->i_reserved_meta_blocks);
1835     208319 :         EXT4_I(inode)->i_reserved_meta_blocks = mdb;
1836     208319 :         spin_unlock(&EXT4_I(inode)->i_block_reservation_lock);
1837            :
1838     208305 :         vfs_dq_release_reservation_block(inode, release);
1839            : }
1840            :
1841            : static void ext4_da_page_release_reservation(struct page *page,
1842            :                                              unsigned long offset)
1843            : {
1844   16844269 :         int to_release = 0;
1845            :         struct buffer_head *head, *bh;
1846   16844269 :         unsigned int curr_off = 0;
1847            :
1848   16844269 :         head = page_buffers(page);
1849   16844223 :         bh = head;
1850            :         do {
1851   20535408 :                 unsigned int next_off = curr_off + bh->b_size;
1852            :
1853   41070862 :                 if ((offset <= curr_off) && (buffer_delay(bh))) {
1854     208319 :                         to_release++;
1855            :                         clear_buffer_delay(bh);
1856            :                 }
1857   20535427 :                 curr_off = next_off;
1858   20535427 :         } while ((bh = bh->b_this_page) != head);
1859   16844242 :         ext4_da_release_space(page->mapping->host, to_release);
1860            : }
1861            :
1862            : /*
1863            :  * Delayed allocation stuff
1864            :  */
1865            :
1866            : struct mpage_da_data {
1867            :         struct inode *inode;
1868            :         sector_t b_blocknr;             /* start block number of extent */
1869            :         size_t b_size;                  /* size of extent */
1870            :         unsigned long b_state;          /* state of the extent */
1871            :         unsigned long first_page, next_page;    /* extent of pages */
1872            :         struct writeback_control *wbc;
1873            :         int io_done;
1874            :         int pages_written;
1875            :         int retval;
1876            : };
1877            :
1878            : /*
1879            :  * mpage_da_submit_io - walks through extent of pages and try to write
1880            :  * them with writepage() call back
1881            :  *
1882            :  * @mpd->inode: inode
1883            :  * @mpd->first_page: first page of the extent
1884            :  * @mpd->next_page: page after the last page of the extent
1885            :  *
1886            :  * By the time mpage_da_submit_io() is called we expect all blocks
1887            :  * to be allocated. this may be wrong if allocation failed.
1888            :  *
1889            :  * As pages are already locked by write_cache_pages(), we can't use it
1890            :  */
1891            : static int mpage_da_submit_io(struct mpage_da_data *mpd)
1892   17295074 : {
1893            :         long pages_skipped;
1894            :         struct pagevec pvec;
1895            :         unsigned long index, end;
1896   17295074 :         int ret = 0, err, nr_pages, i;
1897   17295074 :         struct inode *inode = mpd->inode;
1898   17295074 :         struct address_space *mapping = inode->i_mapping;
1899            :
1900   17295074 :         BUG_ON(mpd->next_page <= mpd->first_page);
1901            :         /*
1902            :          * We need to start from the first_page to the next_page - 1
1903            :          * to make sure we also write the mapped dirty buffer_heads.
1904            :          * If we look at mpd->b_blocknr we would only be looking
```

```
1905                :          * at the currently mapped buffer_heads.
1906                :          */
1907         17293004 :         index = mpd->first_page;
1908         17293004 :         end = mpd->next_page - 1;
1909                :
1910                :         pagevec_init(&pvec, 0);
1911         95824180 :         while (index <= end) {
1912         78519642 :                 nr_pages = pagevec_lookup(&pvec, mapping, index, PAGEVEC_SIZE);
1913         78528589 :                 if (nr_pages == 0)
1914                0 :                         break;
1915       1000369067 :                 for (i = 0; i < nr_pages; i++) {
1916        925080887 :                         struct page *page = pvec.pages[i];
1917                :
1918        925080887 :                         index = page->index;
1919        925080887 :                         if (index > end)
1920          3253190 :                                 break;
1921        921827697 :                         index++;
1922                :
1923        921827697 :                         BUG_ON(!PageLocked(page));
1924        921843287 :                         BUG_ON(PageWriteback(page));
1925                :
1926        921841539 :                         pages_skipped = mpd->wbc->pages_skipped;
1927        921841539 :                         err = mapping->a_ops->writepage(page, mpd->wbc);
1928        921840478 :                         if (!err && (pages_skipped == mpd->wbc->pages_skipped))
1929                :                                 /*
1930                :                                  * have successfully written the page
1931                :                                  * without skipping the same
1932                :                                  */
1933        812631925 :                                 mpd->pages_written++;
1934                :                         /*
1935                :                          * In error case, we have to continue because
1936                :                          * remaining pages are still locked
1937                :                          * XXX: unlock and re-dirty them?
1938                :                          */
1939        921840478 :                         if (ret == 0)
1940        921838085 :                                 ret = err;
1941                :                 }
1942                :                 pagevec_release(&pvec);
1943                :         }
1944         17304538 :         return ret;
1945                : }
1946                :
1947                : /*
1948                :  * mpage_put_bnr_to_bhs - walk blocks and assign them actual numbers
1949                :  *
1950                :  * @mpd->inode - inode to walk through
1951                :  * @exbh->b_blocknr - first block on a disk
1952                :  * @exbh->b_size - amount of space in bytes
1953                :  * @logical - first logical block to start assignment with
1954                :  *
1955                :  * the function goes through all passed space and put actual disk
1956                :  * block numbers into buffer heads, dropping BH_Delay and BH_Unwritten
1957                :  */
1958                : static void mpage_put_bnr_to_bhs(struct mpage_da_data *mpd, sector_t logical,
1959                :                                  struct buffer_head *exbh)
1960         13821890 : {
1961         13821890 :         struct inode *inode = mpd->inode;
1962         13821890 :         struct address_space *mapping = inode->i_mapping;
1963         13821890 :         int blocks = exbh->b_size >> inode->i_blkbits;
1964         13821890 :         sector_t pblock = exbh->b_blocknr, cur_logical;
1965                :         struct buffer_head *head, *bh;
1966                :         pgoff_t index, end;
1967                :         struct pagevec pvec;
1968                :         int nr_pages, i;
1969                :
1970         13821890 :         index = logical >> (PAGE_CACHE_SHIFT - inode->i_blkbits);
1971         13821890 :         end = (logical + blocks - 1) >> (PAGE_CACHE_SHIFT - inode->i_blkbits);
1972         13821890 :         cur_logical = index << (PAGE_CACHE_SHIFT - inode->i_blkbits);
1973                :
1974                :         pagevec_init(&pvec, 0);
1975                :
```

```
1976       80995205 :         while (index <= end) {
1977                :                 /* XXX: optimize tail */
1978       67169377 :                 nr_pages = pagevec_lookup(&pvec, mapping, index, PAGEVEC_SIZE);
1979       67169086 :                 if (nr_pages == 0)
1980              0 :                         break;
1981      873298313 :                 for (i = 0; i < nr_pages; i++) {
1982      808116535 :                         struct page *page = pvec.pages[i];
1983                :
1984      808116535 :                         index = page->index;
1985      808116535 :                         if (index > end)
1986        1990543 :                                 break;
1987      806125992 :                         index++;
1988                :
1989      806125992 :                         BUG_ON(!PageLocked(page));
1990      806129506 :                         BUG_ON(PageWriteback(page));
1991      806133965 :                         BUG_ON(!page_has_buffers(page));
1992                :
1993      806132484 :                         bh = page_buffers(page);
1994      806133669 :                         head = bh;
1995                :
1996                :                         /* skip blocks out of the range */
1997                :                         do {
1998      806379741 :                                 if (cur_logical >= logical)
1999      806128645 :                                         break;
2000         251096 :                                 cur_logical++;
2001         251096 :                         } while ((bh = bh->b_this_page) != head);
2002                :
2003                :                         do {
2004      812347418 :                                 if (cur_logical >= logical + blocks)
2005         351016 :                                         break;
2006                :
2007      811996406 :                                 if (buffer_delay(bh) ||
2008                :                                                 buffer_unwritten(bh)) {
2009                :
2010      811996402 :                                         BUG_ON(bh->b_bdev != inode->i_sb->s_bdev);
2011                :
2012      811996392 :                                         if (buffer_delay(bh)) {
2013                :                                                 clear_buffer_delay(bh);
2014      811991956 :                                                 bh->b_blocknr = pblock;
2015                :                                         } else {
2016                :                                                 /*
2017                :                                                  * unwritten already should have
2018                :                                                  * blocknr assigned. Verify that
2019                :                                                  */
2020                :                                                 clear_buffer_unwritten(bh);
2021              4 :                                                 BUG_ON(bh->b_blocknr != pblock);
2022                :                                         }
2023                :
2024              0 :                                 } else if (buffer_mapped(bh))
2025              0 :                                         BUG_ON(bh->b_blocknr != pblock);
2026                :
2027      811991960 :                                 cur_logical++;
2028      811991960 :                                 pblock++;
2029      811991960 :                         } while ((bh = bh->b_this_page) != head);
2030                :                 }
2031                :                 pagevec_release(&pvec);
2032                :         }
2033       13825828 : }
2034                :
2035                :
2036                : /*
2037                :  * __unmap_underlying_blocks - just a helper function to unmap
2038                :  * set of blocks described by @bh
2039                :  */
2040                : static inline void __unmap_underlying_blocks(struct inode *inode,
2041                :                                              struct buffer_head *bh)
2042                : {
2043       13822700 :         struct block_device *bdev = inode->i_sb->s_bdev;
2044                :         int blocks, i;
2045                :
2046       13822700 :         blocks = bh->b_size >> inode->i_blkbits;
```

```
2047  825821460 :          for (i = 0; i < blocks; i++)
2048  811998561 :                  unmap_underlying_metadata(bdev, bh->b_blocknr + i);
2049            : }
2050            :
2051            : static void ext4_da_block_invalidatepages(struct mpage_da_data *mpd,
2052            :                                           sector_t logical, long blk_cnt)
2053          0 : {
2054            :          int nr_pages, i;
2055            :          pgoff_t index, end;
2056            :          struct pagevec pvec;
2057          0 :          struct inode *inode = mpd->inode;
2058          0 :          struct address_space *mapping = inode->i_mapping;
2059            :
2060          0 :          index = logical >> (PAGE_CACHE_SHIFT - inode->i_blkbits);
2061          0 :          end   = (logical + blk_cnt - 1) >>
2062            :                              (PAGE_CACHE_SHIFT - inode->i_blkbits);
2063          0 :          while (index <= end) {
2064          0 :                  nr_pages = pagevec_lookup(&pvec, mapping, index, PAGEVEC_SIZE);
2065          0 :                  if (nr_pages == 0)
2066          0 :                          break;
2067          0 :                  for (i = 0; i < nr_pages; i++) {
2068          0 :                          struct page *page = pvec.pages[i];
2069          0 :                          index = page->index;
2070          0 :                          if (index > end)
2071          0 :                                  break;
2072          0 :                          index++;
2073            :
2074          0 :                          BUG_ON(!PageLocked(page));
2075          0 :                          BUG_ON(PageWriteback(page));
2076          0 :                          block_invalidatepage(page, 0);
2077            :                          ClearPageUptodate(page);
2078          0 :                          unlock_page(page);
2079            :                  }
2080            :          }
2081            :          return;
2082            : }
2083            :
2084            : static void ext4_print_free_blocks(struct inode *inode)
2085          0 : {
2086          0 :          struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);
2087          0 :          printk(KERN_EMERG "Total free blocks count %lld\n",
2088            :                              ext4_count_free_blocks(inode->i_sb));
2089          0 :          printk(KERN_EMERG "Free/Dirty block details\n");
2090          0 :          printk(KERN_EMERG "free_blocks=%lld\n",
2091            :                              (long long)percpu_counter_sum(&sbi->s_freeblocks_counter));
2092          0 :          printk(KERN_EMERG "dirty_blocks=%lld\n",
2093            :                              (long long)percpu_counter_sum(&sbi->s_dirtyblocks_counter));
2094          0 :          printk(KERN_EMERG "Block reservation details\n");
2095          0 :          printk(KERN_EMERG "i_reserved_data_blocks=%u\n",
2096            :                              EXT4_I(inode)->i_reserved_data_blocks);
2097          0 :          printk(KERN_EMERG "i_reserved_meta_blocks=%u\n",
2098            :                              EXT4_I(inode)->i_reserved_meta_blocks);
2099            :          return;
2100            : }
2101            :
2102            : /*
2103            :  * mpage_da_map_blocks - go through given space
2104            :  *
2105            :  * @mpd - bh describing space
2106            :  *
2107            :  * The function skips space we know is already mapped to disk blocks.
2108            :  *
2109            :  */
2110            : static int mpage_da_map_blocks(struct mpage_da_data *mpd)
2111   17301301 : {
2112            :          int err, blks, get_blocks_flags;
2113            :          struct buffer_head new;
2114   17301301 :          sector_t next = mpd->b_blocknr;
2115   17301301 :          unsigned max_blocks = mpd->b_size >> mpd->inode->i_blkbits;
2116   34602602 :          loff_t disksize = EXT4_I(mpd->inode)->i_disksize;
2117   17301301 :          handle_t *handle = NULL;
```

```
2118                :
2119                :         /*
2120                :          * We consider only non-mapped and non-allocated blocks
2121                :          */
2122       17301301 :         if ((mpd->b_state  & (1 << BH_Mapped)) &&
2123                :                 !(mpd->b_state & (1 << BH_Delay)) &&
2124                :                 !(mpd->b_state & (1 << BH_Unwritten)))
2125        3476557 :                 return 0;
2126                :
2127                :         /*
2128                :          * If we didn't accumulate anything to write simply return
2129                :          */
2130       13824744 :         if (!mpd->b_size)
2131              0 :                 return 0;
2132                :
2133       13824744 :         handle = ext4_journal_current_handle();
2134       13824744 :         BUG_ON(!handle);
2135                :
2136                :         /*
2137                :          * Call ext4_get_blocks() to allocate any delayed allocation
2138                :          * blocks, or to convert an uninitialized extent to be
2139                :          * initialized (in the case where we have written into
2140                :          * one or more preallocated blocks).
2141                :          *
2142                :          * We pass in the magic EXT4_GET_BLOCKS_DELALLOC_RESERVE to
2143                :          * indicate that we are on the delayed allocation path.  This
2144                :          * affects functions in many different parts of the allocation
2145                :          * call path.  This flag exists primarily because we don't
2146                :          * want to change *many* call functions, so ext4_get_blocks()
2147                :          * will set the magic i_delalloc_reserved_flag once the
2148                :          * inode's allocation semaphore is taken.
2149                :          *
2150                :          * If the blocks in questions were delalloc blocks, set
2151                :          * EXT4_GET_BLOCKS_DELALLOC_RESERVE so the delalloc accounting
2152                :          * variables are updated after the blocks have been allocated.
2153                :          */
2154       13825751 :         new.b_state = 0;
2155       13825751 :         get_blocks_flags = (EXT4_GET_BLOCKS_CREATE |
2156                :                             EXT4_GET_BLOCKS_DELALLOC_RESERVE);
2157       13825751 :         if (mpd->b_state & (1 << BH_Delay))
2158       13826024 :                 get_blocks_flags |= EXT4_GET_BLOCKS_UPDATE_RESERVE_SPACE;
2159       13825751 :         blks = ext4_get_blocks(handle, mpd->inode, next, max_blocks,
2160                :                                &new, get_blocks_flags);
2161       13820676 :         if (blks < 0) {
2162              0 :                 err = blks;
2163                :                 /*
2164                :                  * If get block returns with error we simply
2165                :                  * return. Later writepage will redirty the page and
2166                :                  * writepages will find the dirty page again
2167                :                  */
2168              0 :                 if (err == -EAGAIN)
2169              0 :                         return 0;
2170                :
2171              0 :                 if (err == -ENOSPC &&
2172                :                     ext4_count_free_blocks(mpd->inode->i_sb)) {
2173              0 :                         mpd->retval = err;
2174              0 :                         return 0;
2175                :                 }
2176                :
2177                :                 /*
2178                :                  * get block failure will cause us to loop in
2179                :                  * writepages, because a_ops->writepage won't be able
2180                :                  * to make progress. The page will be redirtied by
2181                :                  * writepage and writepages will again try to write
2182                :                  * the same.
2183                :                  */
2184              0 :                 printk(KERN_EMERG "%s block allocation failed for inode %lu "
2185                :                                   "at logical offset %llu with max blocks "
2186                :                                   "%zd with error %d\n",
2187                :                                   __func__, mpd->inode->i_ino,
2188                :                                   (unsigned long long)next,
```

```
2189                :                                       mpd->b_size >> mpd->inode->i_blkbits, err);
2190              0 :                     printk(KERN_EMERG "This should not happen.!! "
2191                :                                                 "Data will be lost\n");
2192              0 :                     if (err == -ENOSPC) {
2193              0 :                             ext4_print_free_blocks(mpd->inode);
2194                :                     }
2195                :                     /* invalidate all the pages */
2196              0 :                     ext4_da_block_invalidatepages(mpd, next,
2197                :                                     mpd->b_size >> mpd->inode->i_blkbits);
2198              0 :                     return err;
2199                :             }
2200       13820676 :             BUG_ON(blks == 0);
2201                :
2202       13819437 :             new.b_size = (blks << mpd->inode->i_blkbits);
2203                :
2204       13819437 :             if (buffer_new(&new))
2205       13822700 :                     __unmap_underlying_blocks(mpd->inode, &new);
2206                :
2207                :             /*
2208                :              * If blocks are delayed marked, we need to
2209                :              * put actual blocknr and drop delayed bit
2210                :              */
2211       13819636 :             if ((mpd->b_state & (1 << BH_Delay)) ||
2212                :                 (mpd->b_state & (1 << BH_Unwritten)))
2213       13826336 :                     mpage_put_bnr_to_bhs(mpd, next, &new);
2214                :
2215       27647760 :             if (ext4_should_order_data(mpd->inode)) {
2216       22805576 :                     err = ext4_jbd2_file_inode(handle, mpd->inode);
2217       11403405 :                     if (err)
2218              0 :                             return err;
2219                :             }
2220                :
2221                :             /*
2222                :              * Update on-disk size along with block allocation.
2223                :              */
2224       13825114 :             disksize = ((loff_t) next + blks) << mpd->inode->i_blkbits;
2225       27653523 :             if (disksize > i_size_read(mpd->inode))
2226       22633647 :                     disksize = i_size_read(mpd->inode);
2227       27651148 :             if (disksize > EXT4_I(mpd->inode)->i_disksize) {
2228       13782034 :                     ext4_update_i_disksize(mpd->inode, disksize);
2229       13782585 :                     return ext4_mark_inode_dirty(handle, mpd->inode);
2230                :             }
2231                :
2232          43540 :             return 0;
2233                : }
2234                :
2235                : #define BH_FLAGS ((1 << BH_Uptodate) | (1 << BH_Mapped) | \
2236                :                 (1 << BH_Delay) | (1 << BH_Unwritten))
2237                :
2238                : /*
2239                :  * mpage_add_bh_to_extent - try to add one more block to extent of blocks
2240                :  *
2241                :  * @mpd->lbh - extent of blocks
2242                :  * @logical - logical number of the block in the file
2243                :  * @bh - bh of the block (used to access block's state)
2244                :  *
2245                :  * the function is used to collect contig. blocks in same state
2246                :  */
2247                : static void mpage_add_bh_to_extent(struct mpage_da_data *mpd,
2248                :                                    sector_t logical, size_t b_size,
2249                :                                    unsigned long b_state)
2250       921833300 : {
2251                :             sector_t next;
2252       921833300 :             int nrblocks = mpd->b_size >> mpd->inode->i_blkbits;
2253                :
2254                :             /* check if thereserved journal credits might overflow */
2255      1843666600 :             if (!(EXT4_I(mpd->inode)->i_flags & EXT4_EXTENTS_FL)) {
2256         3342547 :                     if (nrblocks >= EXT4_MAX_TRANS_DATA) {
2257                :                             /*
2258                :                              * With non-extent format we are limited by the journal
2259                :                              * credit available.  Total credit needed to insert
```

```
2260                :                          * nrblocks contiguous blocks is dependent on the
2261                :                          * nrblocks.  So limit nrblocks.
2262                :                          */
2263              2 :                         goto flush_it;
2264        3342545 :                 } else if ((nrblocks + (b_size >> mpd->inode->i_blkbits)) >
2265                :                                 EXT4_MAX_TRANS_DATA) {
2266                :                         /*
2267                :                          * Adding the new buffer_head would make it cross the
2268                :                          * allowed limit for which we have journal credit
2269                :                          * reserved. So limit the new bh->b_size
2270                :                          */
2271              0 :                         b_size = (EXT4_MAX_TRANS_DATA - nrblocks) <<
2272                :                                                 mpd->inode->i_blkbits;
2273                :                         /* we will do mpage_da_submit_io in the next loop */
2274                :                 }
2275                :         }
2276                :         /*
2277                :          * First block in the extent
2278                :          */
2279      921833298 :         if (mpd->b_size == 0) {
2280       13824447 :                 mpd->b_blocknr = logical;
2281       13824447 :                 mpd->b_size = b_size;
2282       13824447 :                 mpd->b_state = b_state & BH_FLAGS;
2283       13824447 :                 return;
2284                :         }
2285                :
2286      908008851 :         next = mpd->b_blocknr + nrblocks;
2287                :         /*
2288                :          * Can we merge the block to our big extent?
2289                :          */
2290      908008851 :         if (logical == next && (b_state & BH_FLAGS) == mpd->b_state) {
2291      908005630 :                 mpd->b_size += b_size;
2292      908005630 :                 return;
2293                :         }
2294                :
2295           3223 : flush_it:
2296                :         /*
2297                :          * We couldn't merge the block to our extent, so we
2298                :          * need to flush current  extent and start new one
2299                :          */
2300           3223 :         if (mpage_da_map_blocks(mpd) == 0)
2301          16359 :                 mpage_da_submit_io(mpd);
2302          16366 :         mpd->io_done = 1;
2303          16366 :         return;
2304                : }
2305                :
2306                : static int ext4_bh_delay_or_unwritten(handle_t *handle, struct buffer_head *bh)
2307    -1994936293 : {
2308             -1 :         return (buffer_delay(bh) || buffer_unwritten(bh)) && buffer_dirty(bh);
2309                : }
2310                :
2311                : /*
2312                :  * __mpage_da_writepage - finds extent of pages and blocks
2313                :  *
2314                :  * @page: page to consider
2315                :  * @wbc: not used, we just follow rules
2316                :  * @data: context
2317                :  *
2318                :  * The function finds extents of pages and scan them for all blocks.
2319                :  */
2320                : static int __mpage_da_writepage(struct page *page,
2321                :                                 struct writeback_control *wbc, void *data)
2322      922017542 : {
2323      922017542 :         struct mpage_da_data *mpd = data;
2324      922017542 :         struct inode *inode = mpd->inode;
2325                :         struct buffer_head *bh, *head;
2326                :         sector_t logical;
2327                :
2328      922017542 :         if (mpd->io_done) {
2329                :                 /*
2330                :                  * Rest of the page in the page_vec
```

```
      :                 * redirty then and skip then. We will
      :                 * try to to write them again after
      :                 * starting a new transaction
      :                 */
    0 :                redirty_page_for_writepage(wbc, page);
    0 :                unlock_page(page);
    0 :                return MPAGE_DA_EXTENT_TAIL;
      :        }
      :        /*
      :         * Can we merge this page to current extent?
      :         */
922017542 :        if (mpd->next_page != page->index) {
      :                /*
      :                 * Nope, we can't. So, we map non-allocated blocks
      :                 * and start IO on them using writepage()
      :                 */
 10208126 :                if (mpd->next_page != mpd->first_page) {
   242220 :                        if (mpage_da_map_blocks(mpd) == 0)
   242195 :                                mpage_da_submit_io(mpd);
      :                        /*
      :                         * skip rest of the page in the page_vec
      :                         */
   242212 :                        mpd->io_done = 1;
   242212 :                        redirty_page_for_writepage(wbc, page);
   242219 :                        unlock_page(page);
   242224 :                        return MPAGE_DA_EXTENT_TAIL;
      :                }
      :
      :                /*
      :                 * Start next extent of pages ...
      :                 */
  9965906 :                mpd->first_page = page->index;
      :
      :                /*
      :                 * ... and blocks
      :                 */
  9965906 :                mpd->b_size = 0;
  9965906 :                mpd->b_state = 0;
  9965906 :                mpd->b_blocknr = 0;
      :        }
      :
921775322 :        mpd->next_page = page->index + 1;
921775322 :        logical = (sector_t) page->index <<
      :                  (PAGE_CACHE_SHIFT - inode->i_blkbits);
      :
921775322 :        if (!page_has_buffers(page)) {
    0 :                mpage_add_bh_to_extent(mpd, logical, PAGE_CACHE_SIZE,
      :                                       (1 << BH_Dirty) | (1 << BH_Uptodate));
    0 :                if (mpd->io_done)
    0 :                        return MPAGE_DA_EXTENT_TAIL;
      :        } else {
      :                /*
      :                 * Page with regular buffer heads, just add all dirty ones
      :                 */
921754448 :                head = page_buffers(page);
921760064 :                bh = head;
      :                do {
930225553 :                        BUG_ON(buffer_locked(bh));
      :                        /*
      :                         * We need to try to allocate
      :                         * unmapped blocks in the same page.
      :                         * Otherwise we won't make progress
      :                         * with the page in ext4_writepage
      :                         */
930264643 :                        if (ext4_bh_delay_or_unwritten(NULL, bh)) {
921870944 :                                mpage_add_bh_to_extent(mpd, logical,
      :                                                       bh->b_size,
      :                                                       bh->b_state);
921854554 :                                if (mpd->io_done)
 16367 :                                        return MPAGE_DA_EXTENT_TAIL;
  8389636 :                        } else if (buffer_dirty(bh) && (buffer_mapped(bh))) {
```

```
      :                                 /*
      :                                  * mapped dirty buffer. We need to update
      :                                  * the b_state because we look at
      :                                  * b_state in mpage_da_map_blocks. We don't
      :                                  * update b_size because if we find an
      :                                  * unmapped buffer_head later we need to
      :                                  * use the b_state flag of that buffer_head.
      :                                  */
  7048875 :                                 if (mpd->b_size == 0)
  7015965 :                                         mpd->b_state = bh->b_state & BH_FLAGS;
      :                         }
930228002 :                         logical++;
930228002 :                 } while ((bh = bh->b_this_page) != head);
      :         }
      :
921762513 :         return 0;
      : }
      :
      : /*
      :  * This is a special get_blocks_t callback which is used by
      :  * ext4_da_write_begin().  It will either return mapped block or
      :  * reserve space for a single block.
      :  *
      :  * For delayed buffer_head we have BH_Mapped, BH_New, BH_Delay set.
      :  * We also have b_blocknr = -1 and b_bdev initialized properly
      :  *
      :  * For unwritten buffer_head we have BH_Mapped, BH_New, BH_Unwritten set.
      :  * We also have b_blocknr = physicalblock mapping unwritten extent and b_bdev
      :  * initialized properly.
      :  */
      : static int ext4_da_get_block_prep(struct inode *inode, sector_t iblock,
      :                                   struct buffer_head *bh_result, int create)
792282763 : {
792282763 :         int ret = 0;
792282763 :         sector_t invalid_block = ~((sector_t) 0xffff);
      :
-1918119007 :         if (invalid_block < ext4_blocks_count(EXT4_SB(inode->i_sb)->s_es))
          0 :                 invalid_block = ~0;
      :
792282763 :         BUG_ON(create == 0);
805609743 :         BUG_ON(bh_result->b_size != inode->i_sb->s_blocksize);
      :
      :         /*
      :          * first, we need to know whether the block is allocated already
      :          * preallocated blocks are unmapped but should treated
      :          * the same as allocated blocks.
      :          */
793341781 :         ret = ext4_get_blocks(NULL, inode, iblock, 1,  bh_result, 0);
1608621298 :         if ((ret == 0) && !buffer_delay(bh_result)) {
      :                 /* the block isn't (pre)allocated yet, let's reserve space */
      :                 /*
      :                  * XXX: __block_prepare_write() unmaps passed block,
      :                  * is it OK?
      :                  */
804297702 :                 ret = ext4_da_reserve_space(inode, 1);
811530225 :                 if (ret)
      :                         /* not enough space to reserve */
          0 :                         return ret;
      :
811530225 :                 map_bh(bh_result, inode->i_sb, invalid_block);
      :                 set_buffer_new(bh_result);
      :                 set_buffer_delay(bh_result);
  1691553 :         } else if (ret > 0) {
  2665872 :                 bh_result->b_size = (ret << inode->i_blkbits);
  2665872 :                 if (buffer_unwritten(bh_result)) {
      :                         /* A delayed write to unwritten bh should
      :                          * be marked new and mapped.  Mapped ensures
      :                          * that we don't do get_block multiple times
      :                          * when we write to the same offset and new
      :                          * ensures that we do proper zero out for
      :                          * partial write.
```

```
2473                :                          */
2474                :                         set_buffer_new(bh_result);
2475                :                         set_buffer_mapped(bh_result);
2476                :                 }
2477          2665872 :                 ret = 0;
2478                :         }
2479                :
2480        814464627 :         return ret;
2481                : }
2482                :
2483                : /*
2484                :  * This function is used as a standard get_block_t calback function
2485                :  * when there is no desire to allocate any blocks.  It is used as a
2486                :  * callback function for block_prepare_write(), nobh_writepage(), and
2487                :  * block_write_full_page().  These functions should only try to map a
2488                :  * single block at a time.
2489                :  *
2490                :  * Since this function doesn't do block allocations even if the caller
2491                :  * requests it by passing in create=1, it is critically important that
2492                :  * any caller checks to make sure that any buffer heads are returned
2493                :  * by this function are either all already mapped or marked for
2494                :  * delayed allocation before calling nobh_writepage() or
2495                :  * block_write_full_page().  Otherwise, b_blocknr could be left
2496                :  * unitialized, and the page write functions will be taken by
2497                :  * surprise.
2498                :  */
2499                : static int noalloc_get_block_write(struct inode *inode, sector_t iblock,
2500                :                                    struct buffer_head *bh_result, int create)
2501              0 : {
2502              0 :         int ret = 0;
2503              0 :         unsigned max_blocks = bh_result->b_size >> inode->i_blkbits;
2504                :
2505              0 :         BUG_ON(bh_result->b_size != inode->i_sb->s_blocksize);
2506                :
2507                :         /*
2508                :          * we don't want to do block allocation in writepage
2509                :          * so call get_block_wrap with create = 0
2510                :          */
2511              0 :         ret = ext4_get_blocks(NULL, inode, iblock, max_blocks, bh_result, 0);
2512              0 :         if (ret > 0) {
2513              0 :                 bh_result->b_size = (ret << inode->i_blkbits);
2514              0 :                 ret = 0;
2515                :         }
2516              0 :         return ret;
2517                : }
2518                :
2519                : static int bget_one(handle_t *handle, struct buffer_head *bh)
2520              0 : {
2521                :         get_bh(bh);
2522              0 :         return 0;
2523                : }
2524                :
2525                : static int bput_one(handle_t *handle, struct buffer_head *bh)
2526              0 : {
2527                :         put_bh(bh);
2528              0 :         return 0;
2529                : }
2530                :
2531                : static int __ext4_journalled_writepage(struct page *page,
2532                :                                        struct writeback_control *wbc,
2533                :                                        unsigned int len)
2534              0 : {
2535              0 :         struct address_space *mapping = page->mapping;
2536              0 :         struct inode *inode = mapping->host;
2537                :         struct buffer_head *page_bufs;
2538              0 :         handle_t *handle = NULL;
2539              0 :         int ret = 0;
2540                :         int err;
2541                :
2542              0 :         page_bufs = page_buffers(page);
2543              0 :         BUG_ON(!page_bufs);
```

```
2544                0 :         walk_page_buffers(handle, page_bufs, 0, len, NULL, bget_one);
2545                  :         /* As soon as we unlock the page, it can go away, but we have
2546                  :          * references to buffers so we are safe */
2547                0 :         unlock_page(page);
2548                  :
2549                0 :         handle = ext4_journal_start(inode, ext4_writepage_trans_blocks(inode));
2550                0 :         if (IS_ERR(handle)) {
2551                0 :                 ret = PTR_ERR(handle);
2552                0 :                 goto out;
2553                  :         }
2554                  :
2555                0 :         ret = walk_page_buffers(handle, page_bufs, 0, len, NULL,
2556                  :                                 do_journal_get_write_access);
2557                  :
2558                0 :         err = walk_page_buffers(handle, page_bufs, 0, len, NULL,
2559                  :                                 write_end_fn);
2560                0 :         if (ret == 0)
2561                0 :                 ret = err;
2562                0 :         err = ext4_journal_stop(handle);
2563                0 :         if (!ret)
2564                0 :                 ret = err;
2565                  :
2566                0 :         walk_page_buffers(handle, page_bufs, 0, len, NULL, bput_one);
2567                0 :         EXT4_I(inode)->i_state |= EXT4_STATE_JDATA;
2568                0 : out:
2569                0 :         return ret;
2570                  : }
2571                  :
2572                  : /*
2573                  :  * Note that we don't need to start a transaction unless we're journaling data
2574                  :  * because we should have holes filled from ext4_page_mkwrite(). We even don't
2575                  :  * need to file the inode to the transaction's list in ordered mode because if
2576                  :  * we are writing back data added by write(), the inode is already there and if
2577                  :  * we are writing back data modified via mmap(), noone guarantees in which
2578                  :  * transaction the data will hit the disk. In case we are journaling data, we
2579                  :  * cannot start transaction directly because transaction start ranks above page
2580                  :  * lock so we have to do some magic.
2581                  :  *
2582                  :  * This function can get called via...
2583                  :  *   - ext4_da_writepages after taking page lock (have journal handle)
2584                  :  *   - journal_submit_inode_data_buffers (no journal handle)
2585                  :  *   - shrink_page_list via pdflush (no journal handle)
2586                  :  *   - grab_page_cache when doing write_begin (have journal handle)
2587                  :  *
2588                  :  * We don't do any block allocation in this function. If we have page with
2589                  :  * multiple blocks we need to write those buffer_heads that are mapped. This
2590                  :  * is important for mmaped based write. So if we do with blocksize 1K
2591                  :  * truncate(f, 1024);
2592                  :  * a = mmap(f, 0, 4096);
2593                  :  * a[0] = 'a';
2594                  :  * truncate(f, 4096);
2595                  :  * we have in the page first buffer_head mapped via page_mkwrite call back
2596                  :  * but other bufer_heads would be unmapped but dirty(dirty done via the
2597                  :  * do_wp_page). So writepage should write the first block. If we modify
2598                  :  * the mmap area beyond 1024 we will again get a page_fault and the
2599                  :  * page_mkwrite callback will do the block allocation and mark the
2600                  :  * buffer_heads mapped.
2601                  :  *
2602                  :  * We redirty the page if we have any buffer_heads that is either delay or
2603                  :  * unwritten in the page.
2604                  :  *
2605                  :  * We can get recursively called as show below.
2606                  :  *
2607                  :  *      ext4_writepage() -> kmalloc() -> __alloc_pages() -> page_launder() ->
2608                  :  *              ext4_writepage()
2609                  :  *
2610                  :  * But since we don't do any block allocation we should not deadlock.
2611                  :  * Page also have the dirty flag cleared so we don't get recurive page_lock.
2612                  :  */
2613                  : static int ext4_writepage(struct page *page,
2614                  :                           struct writeback_control *wbc)
```

```
2615     1357714420 : {
2616     1357714420 :         int ret = 0;
2617                :         loff_t size;
2618                :         unsigned int len;
2619                :         struct buffer_head *page_bufs;
2620     1357714420 :         struct inode *inode = page->mapping->host;
2621                :
2622                :         trace_ext4_writepage(inode, page);
2623     1357731864 :         size = i_size_read(inode);
2624     1357731864 :         if (page->index == size >> PAGE_CACHE_SHIFT)
2625       20236671 :                 len = size & ~PAGE_CACHE_MASK;
2626                :         else
2627     1337495193 :                 len = PAGE_CACHE_SIZE;
2628                :
2629     1357731864 :         if (page_has_buffers(page)) {
2630     1357768337 :                 page_bufs = page_buffers(page);
2631     1357781325 :                 if (walk_page_buffers(NULL, page_bufs, 0, len, NULL,
2632                :                                       ext4_bh_delay_or_unwritten)) {
2633                :                         /*
2634                :                          * We don't want to do  block allocation
2635                :                          * So redirty the page and return
2636                :                          * We may reach here when we do a journal commit
2637                :                          * via journal_submit_inode_data_buffers.
2638                :                          * If we don't have mapping block we just ignore
2639                :                          * them. We can also reach here via shrink_page_list
2640                :                          */
2641      334659685 :                         redirty_page_for_writepage(wbc, page);
2642      334657636 :                         unlock_page(page);
2643      334657659 :                         return 0;
2644                :                 }
2645                :         } else {
2646                :                 /*
2647                :                  * The test for page_has_buffers() is subtle:
2648                :                  * We know the page is dirty but it lost buffers. That means
2649                :                  * that at some moment in time after write_begin()/write_end()
2650                :                  * has been called all buffers have been clean and thus they
2651                :                  * must have been written at least once. So they are all
2652                :                  * mapped and we can happily proceed with mapping them
2653                :                  * and writing the page.
2654                :                  *
2655                :                  * Try to initialize the buffer_heads and check whether
2656                :                  * all are mapped and non delay. We don't want to
2657                :                  * do block allocation here.
2658                :                  */
2659              0 :                 ret = block_prepare_write(page, 0, len,
2660                :                                           noalloc_get_block_write);
2661              0 :                 if (!ret) {
2662              0 :                         page_bufs = page_buffers(page);
2663                :                         /* check whether all are mapped and non delay */
2664              0 :                         if (walk_page_buffers(NULL, page_bufs, 0, len, NULL,
2665                :                                               ext4_bh_delay_or_unwritten)) {
2666              0 :                                 redirty_page_for_writepage(wbc, page);
2667              0 :                                 unlock_page(page);
2668              0 :                                 return 0;
2669                :                         }
2670                :                 } else {
2671                :                         /*
2672                :                          * We can't do block allocation here
2673                :                          * so just redity the page and unlock
2674                :                          * and return
2675                :                          */
2676              0 :                         redirty_page_for_writepage(wbc, page);
2677              0 :                         unlock_page(page);
2678              0 :                         return 0;
2679                :                 }
2680                :                 /* now mark the buffer_heads as dirty and uptodate */
2681              0 :                 block_commit_write(page, 0, len);
2682                :         }
2683                :
2684     1023017597 :         if (PageChecked(page) && ext4_should_journal_data(inode)) {
2685                :                 /*
```

```
2686                :                  * It's mmapped pagecache.  Add buffers and journal it.  There
2687                :                  * doesn't seem much point in redirtying the page here.
2688                :                  */
2689                :                 ClearPageChecked(page);
2690              0 :                 return __ext4_journalled_writepage(page, wbc, len);
2691                :         }
2692                :
2693     2046039405 :         if (test_opt(inode->i_sb, NOBH) && ext4_should_writeback_data(inode))
2694              0 :                 ret = nobh_writepage(page, noalloc_get_block_write, wbc);
2695                :         else
2696     1023017597 :                 ret = block_write_full_page(page, noalloc_get_block_write,
2697                :                                             wbc);
2698                :
2699     1023094215 :         return ret;
2700                : }
2701                :
2702                : /*
2703                :  * This is called via ext4_da_writepages() to
2704                :  * calulate the total number of credits to reserve to fit
2705                :  * a single extent allocation into a single transaction,
2706                :  * ext4_da_writpeages() will loop calling this before
2707                :  * the block allocation.
2708                :  */
2709                :
2710                : static int ext4_da_writepages_trans_blocks(struct inode *inode)
2711                : {
2712       34984818 :         int max_blocks = EXT4_I(inode)->i_reserved_data_blocks;
2713                :
2714                :         /*
2715                :          * With non-extent format the journal credit needed to
2716                :          * insert nrblocks contiguous block is dependent on
2717                :          * number of contiguous block. So we will limit
2718                :          * number of contiguous block to a sane value
2719                :          */
2720       34984818 :         if (!(inode->i_flags & EXT4_EXTENTS_FL) &&
2721                :             (max_blocks > EXT4_MAX_TRANS_DATA))
2722        4772229 :                 max_blocks = EXT4_MAX_TRANS_DATA;
2723                :
2724       34984818 :         return ext4_chunk_trans_blocks(inode, max_blocks);
2725                : }
2726                :
2727                : static int ext4_da_writepages(struct address_space *mapping,
2728                :                               struct writeback_control *wbc)
2729       28648752 : {
2730                :         pgoff_t index;
2731       28648752 :         int range_whole = 0;
2732       28648752 :         handle_t *handle = NULL;
2733                :         struct mpage_da_data mpd;
2734       28648752 :         struct inode *inode = mapping->host;
2735                :         int no_nrwrite_index_update;
2736       28648752 :         int pages_written = 0;
2737                :         long pages_skipped;
2738       28648752 :         int range_cyclic, cycled = 1, io_done = 0;
2739       28648752 :         int needed_blocks, ret = 0, nr_to_writebump = 0;
2740       57297504 :         struct ext4_sb_info *sbi = EXT4_SB(mapping->host->i_sb);
2741                :
2742                :         trace_ext4_da_writepages(inode, wbc);
2743                :
2744                :         /*
2745                :          * No pages to write? This is mainly a kludge to avoid starting
2746                :          * a transaction for special inodes like journal inode on last iput()
2747                :          * because that could violate lock ordering on umount
2748                :          */
2749       28648817 :         if (!mapping->nrpages || !mapping_tagged(mapping, PAGECACHE_TAG_DIRTY))
2750       12263759 :                 return 0;
2751                :
2752                :         /*
2753                :          * If the filesystem has aborted, it is read-only, so return
2754                :          * right away instead of dumping stack traces later on that
2755                :          * will obscure the real source of the problem.  We test
2756                :          * EXT4_MF_FS_ABORTED instead of sb->s_flag's MS_RDONLY because
```

```
2757                :          * the latter could be true if the filesystem is mounted
2758                :          * read-only, and in that case, ext4_da_writepages should
2759                :          * *never* be called, so if that ever happens, we would want
2760                :          * the stack trace.
2761                :          */
2762       16384940 :         if (unlikely(sbi->s_mount_flags & EXT4_MF_FS_ABORTED))
2763              0 :                 return -EROFS;
2764                :
2765                :         /*
2766                :          * Make sure nr_to_write is >= sbi->s_mb_stream_request
2767                :          * This make sure small files blocks are allocated in
2768                :          * single attempt. This ensure that small files
2769                :          * get less fragmented.
2770                :          */
2771       16388589 :         if (wbc->nr_to_write < sbi->s_mb_stream_request) {
2772         146771 :                 nr_to_writebump = sbi->s_mb_stream_request - wbc->nr_to_write;
2773         146771 :                 wbc->nr_to_write = sbi->s_mb_stream_request;
2774                :         }
2775       16388589 :         if (wbc->range_start == 0 && wbc->range_end == LLONG_MAX)
2776        9345911 :                 range_whole = 1;
2777                :
2778       16388589 :         range_cyclic = wbc->range_cyclic;
2779       16388589 :         if (wbc->range_cyclic) {
2780       16174633 :                 index = mapping->writeback_index;
2781       16174633 :                 if (index)
2782        6919012 :                         cycled = 0;
2783       16174633 :                 wbc->range_start = index << PAGE_CACHE_SHIFT;
2784       16174633 :                 wbc->range_end  = LLONG_MAX;
2785       16174633 :                 wbc->range_cyclic = 0;
2786                :         } else
2787         213956 :                 index = wbc->range_start >> PAGE_CACHE_SHIFT;
2788                :
2789       16388589 :         mpd.wbc = wbc;
2790       16388589 :         mpd.inode = mapping->host;
2791                :
2792                :         /*
2793                :          * we don't want write_cache_pages to update
2794                :          * nr_to_write and writeback_index
2795                :          */
2796       16388589 :         no_nrwrite_index_update = wbc->no_nrwrite_index_update;
2797       16388589 :         wbc->no_nrwrite_index_update = 1;
2798       16388589 :         pages_skipped = wbc->pages_skipped;
2799                :
2800                : retry:
2801       35580341 :         while (!ret && wbc->nr_to_write > 0) {
2802                :
2803                :                 /*
2804                :                  * we  insert one extent at a time. So we need
2805                :                  * credit needed for single extent allocation.
2806                :                  * journalled mode is currently not supported
2807                :                  * by delalloc
2808                :                  */
2809       34985152 :                 BUG_ON(ext4_should_journal_data(inode));
2810       34981044 :                 needed_blocks = ext4_da_writepages_trans_blocks(inode);
2811                :
2812                :                 /* start a new transaction*/
2813       34987943 :                 handle = ext4_journal_start(inode, needed_blocks);
2814       34988169 :                 if (IS_ERR(handle)) {
2815              0 :                         ret = PTR_ERR(handle);
2816              0 :                         printk(KERN_CRIT "%s: jbd2_start: "
2817                :                                "%ld pages, ino %lu; err %d\n", __func__,
2818                :                                wbc->nr_to_write, inode->i_ino, ret);
2819              0 :                         dump_stack();
2820              0 :                         goto out_writepages;
2821                :                 }
2822                :
2823                :                 /*
2824                :                  * Now call __mpage_da_writepage to find the next
2825                :                  * contiguous region of logical blocks that need
2826                :                  * blocks to be allocated by ext4.  We don't actually
2827                :                  * submit the blocks for I/O here, even though
```

```
2828                :                  * write_cache_pages thinks it will, and will set the
2829                :                  * pages as clean for write before calling
2830                :                  * __mpage_da_writepage().
2831                :                  */
2832        34988169 :                 mpd.b_size = 0;
2833        34988169 :                 mpd.b_state = 0;
2834        34988169 :                 mpd.b_blocknr = 0;
2835        34988169 :                 mpd.first_page = 0;
2836        34988169 :                 mpd.next_page = 0;
2837        34988169 :                 mpd.io_done = 0;
2838        34988169 :                 mpd.pages_written = 0;
2839        34988169 :                 mpd.retval = 0;
2840        34988169 :                 ret = write_cache_pages(mapping, wbc, __mpage_da_writepage,
2841                :                                         &mpd);
2842                :                 /*
2843                :                  * If we have a contigous extent of pages and we
2844                :                  * haven't done the I/O yet, map the blocks and submit
2845                :                  * them for I/O.
2846                :                  */
2847        34984264 :                 if (!mpd.io_done && mpd.next_page != mpd.first_page) {
2848        17043494 :                         if (mpage_da_map_blocks(&mpd) == 0)
2849        17036230 :                                 mpage_da_submit_io(&mpd);
2850        17045066 :                         mpd.io_done = 1;
2851        17045066 :                         ret = MPAGE_DA_EXTENT_TAIL;
2852                :                 }
2853        34985836 :                 wbc->nr_to_write -= mpd.pages_written;
2854                :
2855        34985836 :                 ext4_journal_stop(handle);
2856                :
2857        34980201 :                 if ((mpd.retval == -ENOSPC) && sbi->s_journal) {
2858                :                         /* commit the transaction which would
2859                :                          * free blocks released in the transaction
2860                :                          * and try again
2861                :                          */
2862              0 :                         jbd2_journal_force_commit_nested(sbi->s_journal);
2863              0 :                         wbc->pages_skipped = pages_skipped;
2864              0 :                         ret = 0;
2865        34980201 :                 } else if (ret == MPAGE_DA_EXTENT_TAIL) {
2866                :                         /*
2867                :                          * got one extent now try with
2868                :                          * rest of the pages
2869                :                          */
2870        17298801 :                         pages_written += mpd.pages_written;
2871        17298801 :                         wbc->pages_skipped = pages_skipped;
2872        17298801 :                         ret = 0;
2873        17298801 :                         io_done = 1;
2874        17681400 :                 } else if (wbc->nr_to_write)
2875                :                         /*
2876                :                          * There is no more writeout needed
2877                :                          * or we requested for a noblocking writeout
2878                :                          * and we found the device congested
2879                :                          */
2880        17681895 :                         break;
2881                :         }
2882        18277084 :         if (!io_done && !cycled) {
2883         1893446 :                 cycled = 1;
2884         1893446 :                 index = 0;
2885         1893446 :                 wbc->range_start = index << PAGE_CACHE_SHIFT;
2886         1893446 :                 wbc->range_end  = mapping->writeback_index - 1;
2887         1893446 :                 goto retry;
2888                :         }
2889        16383638 :         if (pages_skipped != wbc->pages_skipped)
2890              0 :                 printk(KERN_EMERG "This should not happen leaving %s "
2891                :                                   "with nr_to_write = %ld ret = %d\n",
2892                :                                   __func__, wbc->nr_to_write, ret);
2893                :
2894                :         /* Update index */
2895        16382191 :         index += pages_written;
2896        16382191 :         wbc->range_cyclic = range_cyclic;
2897        16382191 :         if (wbc->range_cyclic || (range_whole && wbc->nr_to_write > 0))
2898                :                 /*
```

```
2899                :                  * set the writeback_index so that range_cyclic
2900                :                  * mode will write it back later
2901                :                  */
2902         16381764 :                 mapping->writeback_index = index;
2903                :
2904         16382191 : out_writepages:
2905         16382191 :         if (!no_nrwrite_index_update)
2906         16382191 :                 wbc->no_nrwrite_index_update = 0;
2907         16382191 :         wbc->nr_to_write -= nr_to_writebump;
2908                :         trace_ext4_da_writepages_result(inode, wbc, ret, pages_written);
2909         16380092 :         return ret;
2910                : }
2911                :
2912                : #define FALL_BACK_TO_NONDELALLOC 1
2913                : static int ext4_nonda_switch(struct super_block *sb)
2914                : {
2915                :         s64 free_blocks, dirty_blocks;
2916        827862363 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2917                :
2918                :         /*
2919                :          * switch to non delalloc mode if we are running low
2920                :          * on free block. The free block accounting via percpu
2921                :          * counters can get slightly wrong with percpu_counter_batch getting
2922                :          * accumulated on each CPU without updating global counters
2923                :          * Delalloc need an accurate free block accounting. So switch
2924                :          * to non delalloc when we are near to error range.
2925                :          */
2926       1663145308 :         free_blocks  = percpu_counter_read_positive(&sbi->s_freeblocks_counter);
2927       1674268147 :         dirty_blocks = percpu_counter_read_positive(&sbi->s_dirtyblocks_counter);
2928        838985202 :         if (2 * free_blocks < 3 * dirty_blocks ||
2929                :                 free_blocks < (dirty_blocks + EXT4_FREEBLOCKS_WATERMARK)) {
2930                :                 /*
2931                :                  * free block count is less that 150% of dirty blocks
2932                :                  * or free blocks is less that watermark
2933                :                  */
2934                0 :                 return 1;
2935                :         }
2936        849191164 :         return 0;
2937                : }
2938                :
2939                : static int ext4_da_write_begin(struct file *file, struct address_space *mapping,
2940                :                                loff_t pos, unsigned len, unsigned flags,
2941                :                                struct page **pagep, void **fsdata)
2942        827862363 : {
2943        827862363 :         int ret, retries = 0;
2944                :         struct page *page;
2945                :         pgoff_t index;
2946                :         unsigned from, to;
2947        827862363 :         struct inode *inode = mapping->host;
2948                :         handle_t *handle;
2949                :
2950        827862363 :         index = pos >> PAGE_CACHE_SHIFT;
2951        827862363 :         from = pos & (PAGE_CACHE_SIZE - 1);
2952        827862363 :         to = from + len;
2953                :
2954       1666847565 :         if (ext4_nonda_switch(inode->i_sb)) {
2955            27274 :                 *fsdata = (void *)FALL_BACK_TO_NONDELALLOC;
2956            27274 :                 return ext4_write_begin(file, mapping, pos,
2957                :                                         len, flags, pagep, fsdata);
2958                :         }
2959        838957928 :         *fsdata = (void *)0;
2960                :         trace_ext4_da_write_begin(inode, pos, len, flags);
2961        838673118 : retry:
2962                :         /*
2963                :          * With delayed allocation, we don't log the i_disksize update
2964                :          * if there is delayed block allocation. But we still need
2965                :          * to journalling the i_disksize update if writes to the end
2966                :          * of file which has an already mapped buffer.
2967                :          */
2968        847634980 :         handle = ext4_journal_start(inode, 1);
2969        848062760 :         if (IS_ERR(handle)) {
```

```
      2970               0 :                  ret = PTR_ERR(handle);
      2971               0 :                  goto out;
      2972                 :          }
      2973                 :          /* We cannot recurse into the filesystem as the transaction is already
      2974                 :           * started */
      2975       848062760 :          flags |= AOP_FLAG_NOFS;
      2976                 :
      2977       848062760 :          page = grab_cache_page_write_begin(mapping, index, flags);
      2978       834069035 :          if (!page) {
      2979               0 :                  ext4_journal_stop(handle);
      2980               0 :                  ret = -ENOMEM;
      2981               0 :                  goto out;
      2982                 :          }
      2983       834069035 :          *pagep = page;
      2984                 :
      2985       834069035 :          ret = block_write_begin(file, mapping, pos, len, flags, pagep, fsdata,
      2986                 :                                  ext4_da_get_block_prep);
      2987       832775454 :          if (ret < 0) {
      2988               0 :                  unlock_page(page);
      2989               0 :                  ext4_journal_stop(handle);
      2990               0 :                  page_cache_release(page);
      2991                 :                  /*
      2992                 :                   * block_write_begin may have instantiated a few blocks
      2993                 :                   * outside i_size.  Trim these off again. Don't need
      2994                 :                   * i_size_read because we hold i_mutex.
      2995                 :                   */
      2996               0 :                  if (pos + len > inode->i_size)
      2997               0 :                          ext4_truncate(inode);
      2998                 :          }
      2999                 :
      3000       830111955 :          if (ret == -ENOSPC && ext4_should_retry_alloc(inode->i_sb, &retries))
      3001               0 :                  goto retry;
      3002       830111955 : out:
      3003       830111955 :          return ret;
      3004                 : }
      3005                 :
      3006                 : /*
      3007                 :  * Check if we should update i_disksize
      3008                 :  * when write to the end of file but not require block allocation
      3009                 :  */
      3010                 : static int ext4_da_should_update_i_disksize(struct page *page,
      3011                 :                                             unsigned long offset)
      3012                 : {
      3013                 :          struct buffer_head *bh;
      3014       831933399 :          struct inode *inode = page->mapping->host;
      3015                 :          unsigned int idx;
      3016                 :          int i;
      3017                 :
      3018       831933399 :          bh = page_buffers(page);
      3019       835891377 :          idx = offset >> inode->i_blkbits;
      3020                 :
      3021       842304904 :          for (i = 0; i < idx; i++)
      3022         6413527 :                  bh = bh->b_this_page;
      3023                 :
      3024      1679946699 :          if (!buffer_mapped(bh) || (buffer_delay(bh)) || buffer_unwritten(bh))
      3025       833456853 :                  return 0;
      3026         5410422 :          return 1;
      3027                 : }
      3028                 :
      3029                 : static int ext4_da_write_end(struct file *file,
      3030                 :                              struct address_space *mapping,
      3031                 :                              loff_t pos, unsigned len, unsigned copied,
      3032                 :                              struct page *page, void *fsdata)
      3033       841822953 : {
      3034       841822953 :          struct inode *inode = mapping->host;
      3035       841822953 :          int ret = 0, ret2;
      3036       841822953 :          handle_t *handle = ext4_journal_current_handle();
      3037                 :          loff_t new_i_size;
      3038                 :          unsigned long start, end;
      3039       841822953 :          int write_mode = (int)(unsigned long)fsdata;
      3040                 :
```

```
3041  841822953 :          if (write_mode == FALL_BACK_TO_NONDELALLOC) {
3042      27264 :                  if (ext4_should_order_data(inode)) {
3043      27264 :                          return ext4_ordered_write_end(file, mapping, pos,
3044            :                                                        len, copied, page, fsdata);
3045          0 :                  } else if (ext4_should_writeback_data(inode)) {
3046          0 :                          return ext4_writeback_write_end(file, mapping, pos,
3047            :                                                        len, copied, page, fsdata);
3048            :                  } else {
3049          0 :                          BUG();
3050            :                  }
3051            :          }
3052            :
3053            :          trace_ext4_da_write_end(inode, pos, len, copied);
3054  843365121 :          start = pos & (PAGE_CACHE_SIZE - 1);
3055  843365121 :          end = start + copied - 1;
3056            :
3057            :          /*
3058            :           * generic_write_end() will run mark_inode_dirty() if i_size
3059            :           * changes.  So let's piggyback the i_disksize mark_inode_dirty
3060            :           * into that.
3061            :           */
3062            :
3063  843365121 :          new_i_size = pos + copied;
3064  843365121 :          if (new_i_size > EXT4_I(inode)->i_disksize) {
3065  838867275 :                  if (ext4_da_should_update_i_disksize(page, end)) {
3066    5408621 :                          down_write(&EXT4_I(inode)->i_data_sem);
3067    5410535 :                          if (new_i_size > EXT4_I(inode)->i_disksize) {
3068            :                                  /*
3069            :                                   * Updating i_disksize when extending file
3070            :                                   * without needing block allocation
3071            :                                   */
3072    5408991 :                                  if (ext4_should_order_data(inode))
3073    4193517 :                                          ret = ext4_jbd2_file_inode(handle,
3074            :                                                                     inode);
3075            :
3076    5407722 :                                  EXT4_I(inode)->i_disksize = new_i_size;
3077            :                          }
3078    5409266 :                          up_write(&EXT4_I(inode)->i_data_sem);
3079            :                          /* We need to mark inode dirty even if
3080            :                           * new_i_size is less that inode->i_size
3081            :                           * bu greater than i_disksize.(hint delalloc)
3082            :                           */
3083    5410363 :                          ext4_mark_inode_dirty(handle, inode);
3084            :                  }
3085            :          }
3086  850294962 :          ret2 = generic_write_end(file, mapping, pos, len, copied,
3087            :                                                        page, fsdata);
3088  823317578 :          copied = ret2;
3089  823317578 :          if (ret2 < 0)
3090          0 :                  ret = ret2;
3091  823317578 :          ret2 = ext4_journal_stop(handle);
3092  830159788 :          if (!ret)
3093  845631114 :                  ret = ret2;
3094            :
3095  830159788 :          return ret ? ret : copied;
3096            : }
3097            :
3098            : static void ext4_da_invalidatepage(struct page *page, unsigned long offset)
3099   16844265 : {
3100            :          /*
3101            :           * Drop reserved blocks
3102            :           */
3103   16844265 :          BUG_ON(!PageLocked(page));
3104   16844208 :          if (!page_has_buffers(page))
3105          0 :                  goto out;
3106            :
3107            :          ext4_da_page_release_reservation(page, offset);
3108            :
3109   16844228 : out:
3110   16844228 :          ext4_invalidatepage(page, offset);
3111            :
```

```
3112                :         return;
3113                : }
3114                :
3115                : /*
3116                :  * Force all delayed allocation blocks to be allocated for a given inode.
3117                :  */
3118                : int ext4_alloc_da_blocks(struct inode *inode)
3119              0 : {
3120              0 :         if (!EXT4_I(inode)->i_reserved_data_blocks &&
3121                :             !EXT4_I(inode)->i_reserved_meta_blocks)
3122              0 :                 return 0;
3123                :
3124                :         /*
3125                :          * We do something simple for now.  The filemap_flush() will
3126                :          * also start triggering a write of the data blocks, which is
3127                :          * not strictly speaking necessary (and for users of
3128                :          * laptop_mode, not even desirable).  However, to do otherwise
3129                :          * would require replicating code paths in:
3130                :          *
3131                :          * ext4_da_writepages() ->
3132                :          *    write_cache_pages() ---> (via passed in callback function)
3133                :          *        __mpage_da_writepage() -->
3134                :          *           mpage_add_bh_to_extent()
3135                :          *           mpage_da_map_blocks()
3136                :          *
3137                :          * The problem is that write_cache_pages(), located in
3138                :          * mm/page-writeback.c, marks pages clean in preparation for
3139                :          * doing I/O, which is not desirable if we're not planning on
3140                :          * doing I/O at all.
3141                :          *
3142                :          * We could call write_cache_pages(), and then redirty all of
3143                :          * the pages by calling redirty_page_for_writeback() but that
3144                :          * would be ugly in the extreme.  So instead we would need to
3145                :          * replicate parts of the code in the above functions,
3146                :          * simplifying them becuase we wouldn't actually intend to
3147                :          * write out the pages, but rather only collect contiguous
3148                :          * logical block extents, call the multi-block allocator, and
3149                :          * then update the buffer heads with the block allocations.
3150                :          *
3151                :          * For now, though, we'll cheat by calling filemap_flush(),
3152                :          * which will map the blocks, and start the I/O, but not
3153                :          * actually wait for the I/O to complete.
3154                :          */
3155              0 :         return filemap_flush(inode->i_mapping);
3156                : }
3157                :
3158                : /*
3159                :  * bmap() is special.  It gets used by applications such as lilo and by
3160                :  * the swapper to find the on-disk block of a specific piece of data.
3161                :  *
3162                :  * Naturally, this is dangerous if the block concerned is still in the
3163                :  * journal.  If somebody makes a swapfile on an ext4 data-journaling
3164                :  * filesystem and enables swap, then they may get a nasty shock when the
3165                :  * data getting swapped to that swapfile suddenly gets overwritten by
3166                :  * the original zero's written out previously to the journal and
3167                :  * awaiting writeback in the kernel's buffer cache.
3168                :  *
3169                :  * So, if we see any bmap calls here on a modified, data-journaled file,
3170                :  * take extra steps to flush any blocks which might be in the cache.
3171                :  */
3172                : static sector_t ext4_bmap(struct address_space *mapping, sector_t block)
3173       37951535 : {
3174       37951535 :         struct inode *inode = mapping->host;
3175                :         journal_t *journal;
3176                :         int err;
3177                :
3178       37951535 :         if (mapping_tagged(mapping, PAGECACHE_TAG_DIRTY) &&
3179                :                         test_opt(inode->i_sb, DELALLOC)) {
3180                :                 /*
3181                :                  * With delalloc we want to sync the file
3182                :                  * so that we can make sure we allocate
```

```
3183                :                  * blocks for file
3184                :                  */
3185              0 :                 filemap_write_and_wait(mapping);
3186                :         }
3187                :
3188      113854511 :         if (EXT4_JOURNAL(inode) && EXT4_I(inode)->i_state & EXT4_STATE_JDATA) {
3189                :                 /*
3190                :                  * This is a REALLY heavyweight approach, but the use of
3191                :                  * bmap on dirty files is expected to be extremely rare:
3192                :                  * only if we run lilo or swapon on a freshly made file
3193                :                  * do we expect this to happen.
3194                :                  *
3195                :                  * (bmap requires CAP_SYS_RAWIO so this does not
3196                :                  * represent an unprivileged user DOS attack --- we'd be
3197                :                  * in trouble if mortal users could trigger this path at
3198                :                  * will.)
3199                :                  *
3200                :                  * NB. EXT4_STATE_JDATA is not set on files other than
3201                :                  * regular files.  If somebody wants to bmap a directory
3202                :                  * or symlink and gets confused because the buffer
3203                :                  * hasn't yet been flushed to disk, they deserve
3204                :                  * everything they get.
3205                :                  */
3206                :
3207              0 :                 EXT4_I(inode)->i_state &= ~EXT4_STATE_JDATA;
3208              0 :                 journal = EXT4_JOURNAL(inode);
3209              0 :                 jbd2_journal_lock_updates(journal);
3210              0 :                 err = jbd2_journal_flush(journal);
3211              0 :                 jbd2_journal_unlock_updates(journal);
3212                :
3213              0 :                 if (err)
3214              0 :                         return 0;
3215                :         }
3216                :
3217       37951535 :         return generic_block_bmap(mapping, block, ext4_get_block);
3218                : }
3219                :
3220                : static int ext4_readpage(struct file *file, struct page *page)
3221           2061 : {
3222           2061 :         return mpage_readpage(page, ext4_get_block);
3223                : }
3224                :
3225                : static int
3226                : ext4_readpages(struct file *file, struct address_space *mapping,
3227                :                struct list_head *pages, unsigned nr_pages)
3228         165230 : {
3229         165230 :         return mpage_readpages(mapping, pages, nr_pages, ext4_get_block);
3230                : }
3231                :
3232                : static void ext4_invalidatepage(struct page *page, unsigned long offset)
3233       25036886 : {
3234       50073772 :         journal_t *journal = EXT4_JOURNAL(page->mapping->host);
3235                :
3236                :         /*
3237                :          * If it's a full truncate we just forget about the pending dirtying
3238                :          */
3239       25036886 :         if (offset == 0)
3240                :                 ClearPageChecked(page);
3241                :
3242       25036892 :         if (journal)
3243       25036892 :                 jbd2_journal_invalidatepage(journal, page, offset);
3244                :         else
3245              0 :                 block_invalidatepage(page, offset);
3246       25036888 : }
3247                :
3248                : static int ext4_releasepage(struct page *page, gfp_t wait)
3249      984477926 : {
3250     1968955852 :         journal_t *journal = EXT4_JOURNAL(page->mapping->host);
3251                :
3252      984477926 :         WARN_ON(PageChecked(page));
3253      986413668 :         if (!page_has_buffers(page))
```

```
3254                0 :                 return 0;
3255        984945317 :         if (journal)
3256        984945317 :                 return jbd2_journal_try_to_free_buffers(journal, page, wait);
3257                  :         else
3258                0 :                 return try_to_free_buffers(page);
3259                  : }
3260                  :
3261                  : /*
3262                  :  * If the O_DIRECT write will extend the file then add this inode to the
3263                  :  * orphan list.  So recovery will truncate it back to the original size
3264                  :  * if the machine crashes during the write.
3265                  :  *
3266                  :  * If the O_DIRECT write is intantiating holes inside i_size and the machine
3267                  :  * crashes then stale disk data _may_ be exposed inside the file. But current
3268                  :  * VFS code falls back into buffered path in that case so we are safe.
3269                  :  */
3270                  : static ssize_t ext4_direct_IO(int rw, struct kiocb *iocb,
3271                  :                               const struct iovec *iov, loff_t offset,
3272                  :                               unsigned long nr_segs)
3273          1390541 : {
3274          1390541 :         struct file *file = iocb->ki_filp;
3275          1390541 :         struct inode *inode = file->f_mapping->host;
3276          1390541 :         struct ext4_inode_info *ei = EXT4_I(inode);
3277                  :         handle_t *handle;
3278                  :         ssize_t ret;
3279          1390541 :         int orphan = 0;
3280          1390541 :         size_t count = iov_length(iov, nr_segs);
3281                  :
3282          1390541 :         if (rw == WRITE) {
3283           553943 :                 loff_t final_size = offset + count;
3284                  :
3285           553943 :                 if (final_size > inode->i_size) {
3286                  :                         /* Credits for sb + inode write */
3287           276738 :                         handle = ext4_journal_start(inode, 2);
3288           276738 :                         if (IS_ERR(handle)) {
3289                0 :                                 ret = PTR_ERR(handle);
3290                0 :                                 goto out;
3291                  :                         }
3292           276738 :                         ret = ext4_orphan_add(handle, inode);
3293           276738 :                         if (ret) {
3294                0 :                                 ext4_journal_stop(handle);
3295                0 :                                 goto out;
3296                  :                         }
3297           276738 :                         orphan = 1;
3298           276738 :                         ei->i_disksize = inode->i_size;
3299           276738 :                         ext4_journal_stop(handle);
3300                  :                 }
3301                  :         }
3302                  :
3303          2781081 :         ret = blockdev_direct_IO(rw, iocb, inode, inode->i_sb->s_bdev, iov,
3304                  :                                  offset, nr_segs,
3305                  :                                  ext4_get_block, NULL);
3306                  :
3307          1390540 :         if (orphan) {
3308                  :                 int err;
3309                  :
3310                  :                 /* Credits for sb + inode write */
3311           276738 :                 handle = ext4_journal_start(inode, 2);
3312           276738 :                 if (IS_ERR(handle)) {
3313                  :                         /* This is really bad luck. We've written the data
3314                  :                          * but cannot extend i_size. Bail out and pretend
3315                  :                          * the write failed... */
3316                0 :                         ret = PTR_ERR(handle);
3317                0 :                         goto out;
3318                  :                 }
3319           276738 :                 if (inode->i_nlink)
3320           276738 :                         ext4_orphan_del(handle, inode);
3321           276738 :                 if (ret > 0) {
3322           276738 :                         loff_t end = offset + ret;
3323           276738 :                         if (end > inode->i_size) {
3324           276738 :                                 ei->i_disksize = end;
```

```
3325                :                                 i_size_write(inode, end);
3326                :                                 /*
3327                :                                  * We're going to return a positive `ret'
3328                :                                  * here due to non-zero-length I/O, so there's
3329                :                                  * no way of reporting error returns from
3330                :                                  * ext4_mark_inode_dirty() to userspace.  So
3331                :                                  * ignore it.
3332                :                                  */
3333           276738 :                                 ext4_mark_inode_dirty(handle, inode);
3334                :                         }
3335                :                 }
3336           276738 :                 err = ext4_journal_stop(handle);
3337           276738 :                 if (ret == 0)
3338                0 :                         ret = err;
3339                :         }
3340          1390543 : out:
3341          1390543 :         return ret;
3342                : }
3343                :
3344                : /*
3345                :  * Pages can be marked dirty completely asynchronously from ext4's journalling
3346                :  * activity.  By filemap_sync_pte(), try_to_unmap_one(), etc.  We cannot do
3347                :  * much here because ->set_page_dirty is called under VFS locks.  The page is
3348                :  * not necessarily locked.
3349                :  *
3350                :  * We cannot just dirty the page and leave attached buffers clean, because the
3351                :  * buffers' dirty state is "definitive".  We cannot just set the buffers dirty
3352                :  * or jbddirty because all the journalling code will explode.
3353                :  *
3354                :  * So what we do is to mark the page "pending dirty" and next time writepage
3355                :  * is called, propagate that into the buffers appropriately.
3356                :  */
3357                : static int ext4_journalled_set_page_dirty(struct page *page)
3358                0 : {
3359                :         SetPageChecked(page);
3360                0 :         return __set_page_dirty_nobuffers(page);
3361                : }
3362                :
3363                : static const struct address_space_operations ext4_ordered_aops = {
3364                :         .readpage               = ext4_readpage,
3365                :         .readpages              = ext4_readpages,
3366                :         .writepage              = ext4_writepage,
3367                :         .sync_page              = block_sync_page,
3368                :         .write_begin            = ext4_write_begin,
3369                :         .write_end              = ext4_ordered_write_end,
3370                :         .bmap                   = ext4_bmap,
3371                :         .invalidatepage         = ext4_invalidatepage,
3372                :         .releasepage            = ext4_releasepage,
3373                :         .direct_IO              = ext4_direct_IO,
3374                :         .migratepage            = buffer_migrate_page,
3375                :         .is_partially_uptodate  = block_is_partially_uptodate,
3376                : };
3377                :
3378                : static const struct address_space_operations ext4_writeback_aops = {
3379                :         .readpage               = ext4_readpage,
3380                :         .readpages              = ext4_readpages,
3381                :         .writepage              = ext4_writepage,
3382                :         .sync_page              = block_sync_page,
3383                :         .write_begin            = ext4_write_begin,
3384                :         .write_end              = ext4_writeback_write_end,
3385                :         .bmap                   = ext4_bmap,
3386                :         .invalidatepage         = ext4_invalidatepage,
3387                :         .releasepage            = ext4_releasepage,
3388                :         .direct_IO              = ext4_direct_IO,
3389                :         .migratepage            = buffer_migrate_page,
3390                :         .is_partially_uptodate  = block_is_partially_uptodate,
3391                : };
3392                :
3393                : static const struct address_space_operations ext4_journalled_aops = {
3394                :         .readpage               = ext4_readpage,
3395                :         .readpages              = ext4_readpages,
```

```
3396                :         .writepage              = ext4_writepage,
3397                :         .sync_page              = block_sync_page,
3398                :         .write_begin            = ext4_write_begin,
3399                :         .write_end              = ext4_journalled_write_end,
3400                :         .set_page_dirty         = ext4_journalled_set_page_dirty,
3401                :         .bmap                   = ext4_bmap,
3402                :         .invalidatepage         = ext4_invalidatepage,
3403                :         .releasepage            = ext4_releasepage,
3404                :         .is_partially_uptodate  = block_is_partially_uptodate,
3405                : };
3406                :
3407                : static const struct address_space_operations ext4_da_aops = {
3408                :         .readpage               = ext4_readpage,
3409                :         .readpages              = ext4_readpages,
3410                :         .writepage              = ext4_writepage,
3411                :         .writepages             = ext4_da_writepages,
3412                :         .sync_page              = block_sync_page,
3413                :         .write_begin            = ext4_da_write_begin,
3414                :         .write_end              = ext4_da_write_end,
3415                :         .bmap                   = ext4_bmap,
3416                :         .invalidatepage         = ext4_da_invalidatepage,
3417                :         .releasepage            = ext4_releasepage,
3418                :         .direct_IO              = ext4_direct_IO,
3419                :         .migratepage            = buffer_migrate_page,
3420                :         .is_partially_uptodate  = block_is_partially_uptodate,
3421                : };
3422                :
3423                : void ext4_set_aops(struct inode *inode)
3424         12388393 : {
3425         30468073 :         if (ext4_should_order_data(inode) &&
3426                :                 test_opt(inode->i_sb, DELALLOC))
3427          8106162 :                 inode->i_mapping->a_ops = &ext4_da_aops;
3428          4282231 :         else if (ext4_should_order_data(inode))
3429          1867357 :                 inode->i_mapping->a_ops = &ext4_ordered_aops;
3430          4843621 :         else if (ext4_should_writeback_data(inode) &&
3431                :                  test_opt(inode->i_sb, DELALLOC))
3432          1214368 :                 inode->i_mapping->a_ops = &ext4_da_aops;
3433          1200506 :         else if (ext4_should_writeback_data(inode))
3434               11 :                 inode->i_mapping->a_ops = &ext4_writeback_aops;
3435                :         else
3436          1200495 :                 inode->i_mapping->a_ops = &ext4_journalled_aops;
3437         12388393 : }
3438                :
3439                : /*
3440                :  * ext4_block_truncate_page() zeroes out a mapping from file offset `from'
3441                :  * up to the end of the block which corresponds to `from'.
3442                :  * This required during truncate. We need to physically zero the tail end
3443                :  * of that block so it doesn't yield old data if the file is later grown.
3444                :  */
3445                : int ext4_block_truncate_page(handle_t *handle,
3446                :                 struct address_space *mapping, loff_t from)
3447                : {
3448                :         ext4_fsblk_t index = from >> PAGE_CACHE_SHIFT;
3449                :         unsigned offset = from & (PAGE_CACHE_SIZE-1);
3450                :         unsigned blocksize, length, pos;
3451                :         ext4_lblk_t iblock;
3452                :         struct inode *inode = mapping->host;
3453                :         struct buffer_head *bh;
3454                :         struct page *page;
3455                :         int err = 0;
3456                :
3457                :         page = find_or_create_page(mapping, from >> PAGE_CACHE_SHIFT,
3458                :                                    mapping_gfp_mask(mapping) & ~__GFP_FS);
3459                :         if (!page)
3460                :                 return -EINVAL;
3461                :
3462                :         blocksize = inode->i_sb->s_blocksize;
3463                :         length = blocksize - (offset & (blocksize - 1));
3464                :         iblock = index << (PAGE_CACHE_SHIFT - inode->i_sb->s_blocksize_bits);
3465                :
3466                :         /*
```

```
3467                :          * For "nobh" option,  we can only work if we don't need to
3468                :          * read-in the page - otherwise we create buffers to do the IO.
3469                :          */
3470              0 :         if (!page_has_buffers(page) && test_opt(inode->i_sb, NOBH) &&
3471                :             ext4_should_writeback_data(inode) && PageUptodate(page)) {
3472              0 :                 zero_user(page, offset, length);
3473              0 :                 set_page_dirty(page);
3474              0 :                 goto unlock;
3475                :         }
3476                :
3477              0 :         if (!page_has_buffers(page))
3478              0 :                 create_empty_buffers(page, blocksize, 0);
3479                :
3480                :         /* Find the buffer that contains "offset" */
3481              0 :         bh = page_buffers(page);
3482              0 :         pos = blocksize;
3483              0 :         while (offset >= pos) {
3484              0 :                 bh = bh->b_this_page;
3485              0 :                 iblock++;
3486              0 :                 pos += blocksize;
3487                :         }
3488                :
3489              0 :         err = 0;
3490              0 :         if (buffer_freed(bh)) {
3491                :                 BUFFER_TRACE(bh, "freed: skip");
3492              0 :                 goto unlock;
3493                :         }
3494                :
3495              0 :         if (!buffer_mapped(bh)) {
3496                :                 BUFFER_TRACE(bh, "unmapped");
3497              0 :                 ext4_get_block(inode, iblock, bh, 0);
3498                :                 /* unmapped? It's a hole - nothing to do */
3499              0 :                 if (!buffer_mapped(bh)) {
3500                :                         BUFFER_TRACE(bh, "still unmapped");
3501              0 :                         goto unlock;
3502                :                 }
3503                :         }
3504                :
3505                :         /* Ok, it's mapped. Make sure it's up-to-date */
3506              0 :         if (PageUptodate(page))
3507              0 :                 set_buffer_uptodate(bh);
3508                :
3509              0 :         if (!buffer_uptodate(bh)) {
3510              0 :                 err = -EIO;
3511              0 :                 ll_rw_block(READ, 1, &bh);
3512              0 :                 wait_on_buffer(bh);
3513                :                 /* Uhhuh. Read error. Complain and punt. */
3514              0 :                 if (!buffer_uptodate(bh))
3515              0 :                         goto unlock;
3516                :         }
3517                :
3518              0 :         if (ext4_should_journal_data(inode)) {
3519                :                 BUFFER_TRACE(bh, "get write access");
3520              0 :                 err = ext4_journal_get_write_access(handle, bh);
3521              0 :                 if (err)
3522              0 :                         goto unlock;
3523                :         }
3524                :
3525              0 :         zero_user(page, offset, length);
3526                :
3527                :         BUFFER_TRACE(bh, "zeroed end of block");
3528                :
3529              0 :         err = 0;
3530              0 :         if (ext4_should_journal_data(inode)) {
3531              0 :                 err = ext4_handle_dirty_metadata(handle, inode, bh);
3532                :         } else {
3533              0 :                 if (ext4_should_order_data(inode))
3534              0 :                         err = ext4_jbd2_file_inode(handle, inode);
3535              0 :                 mark_buffer_dirty(bh);
3536                :         }
3537                :
```

```
3538                0 : unlock:
3539                0 :         unlock_page(page);
3540                0 :         page_cache_release(page);
3541                0 :         return err;
3542                  : }
3543                  :
3544                  : /*
3545                  :  * Probably it should be a library function... search for first non-zero word
3546                  :  * or memcmp with zero_page, whatever is better for particular architecture.
3547                  :  * Linus?
3548                  :  */
3549                  : static inline int all_zeroes(__le32 *p, __le32 *q)
3550                  : {
3551                0 :         while (p < q)
3552                0 :                 if (*p++)
3553                0 :                         return 0;
3554                0 :         return 1;
3555                  : }
3556                  :
3557                  : /**
3558                  :  *      ext4_find_shared - find the indirect blocks for partial truncation.
3559                  :  *      @inode:   inode in question
3560                  :  *      @depth:   depth of the affected branch
3561                  :  *      @offsets: offsets of pointers in that branch (see ext4_block_to_path)
3562                  :  *      @chain:   place to store the pointers to partial indirect blocks
3563                  :  *      @top:     place to the (detached) top of branch
3564                  :  *
3565                  :  *      This is a helper function used by ext4_truncate().
3566                  :  *
3567                  :  *      When we do truncate() we may have to clean the ends of several
3568                  :  *      indirect blocks but leave the blocks themselves alive. Block is
3569                  :  *      partially truncated if some data below the new i_size is refered
3570                  :  *      from it (and it is on the path to the first completely truncated
3571                  :  *      data block, indeed).  We have to free the top of that path along
3572                  :  *      with everything to the right of the path. Since no allocation
3573                  :  *      past the truncation point is possible until ext4_truncate()
3574                  :  *      finishes, we may safely do the latter, but top of branch may
3575                  :  *      require special attention - pageout below the truncation point
3576                  :  *      might try to populate it.
3577                  :  *
3578                  :  *      We atomically detach the top of branch from the tree, store the
3579                  :  *      block number of its root in *@top, pointers to buffer_heads of
3580                  :  *      partially truncated blocks - in @chain[].bh and pointers to
3581                  :  *      their last elements that should not be removed - in
3582                  :  *      @chain[].p. Return value is the pointer to last filled element
3583                  :  *      of @chain.
3584                  :  *
3585                  :  *      The work left to caller to do the actual freeing of subtrees:
3586                  :  *              a) free the subtree starting from *@top
3587                  :  *              b) free the subtrees whose roots are stored in
3588                  :  *                      (@chain[i].p+1 .. end of @chain[i].bh->b_data)
3589                  :  *              c) free the subtrees growing from the inode past the @chain[0].
3590                  :  *                      (no partially truncated stuff there).  */
3591                  :
3592                  : static Indirect *ext4_find_shared(struct inode *inode, int depth,
3593                  :                                   ext4_lblk_t offsets[4], Indirect chain[4],
3594                  :                                   __le32 *top)
3595                  : {
3596                  :         Indirect *partial, *p;
3597                  :         int k, err;
3598                  :
3599                0 :         *top = 0;
3600                  :         /* Make k index the deepest non-null offest + 1 */
3601                0 :         for (k = depth; k > 1 && !offsets[k-1]; k--)
3602                  :                 ;
3603                0 :         partial = ext4_get_branch(inode, k, offsets, chain, &err);
3604                  :         /* Writer: pointers */
3605                0 :         if (!partial)
3606                0 :                 partial = chain + k-1;
3607                  :         /*
3608                  :          * If the branch acquired continuation since we've looked at it -
```

```
3609                :          * fine, it should all survive and (new) top doesn't belong to us.
3610                :          */
3611              0 :         if (!partial->key && *partial->p)
3612                :                 /* Writer: end */
3613                :                 goto no_top;
3614              0 :         for (p = partial; (p > chain) && all_zeroes((__le32 *) p->bh->b_data, p->p); p--)
3615                :                 ;
3616                :         /*
3617                :          * OK, we've found the last block that must survive. The rest of our
3618                :          * branch should be detached before unlocking. However, if that rest
3619                :          * of branch is all ours and does not grow immediately from the inode
3620                :          * it's easier to cheat and just decrement partial->p.
3621                :          */
3622              0 :         if (p == chain + k - 1 && p > chain) {
3623              0 :                 p->p--;
3624                :         } else {
3625              0 :                 *top = *p->p;
3626                :                 /* Nope, don't do this in ext4.  Must leave the tree intact */
3627                : #if 0
3628                :                 *p->p = 0;
3629                : #endif
3630                :         }
3631                :         /* Writer: end */
3632                :
3633              0 :         while (partial > p) {
3634              0 :                 brelse(partial->bh);
3635              0 :                 partial--;
3636                :         }
3637              0 : no_top:
3638              0 :         return partial;
3639                : }
3640                :
3641                : /*
3642                :  * Zero a number of block pointers in either an inode or an indirect block.
3643                :  * If we restart the transaction we must again get write access to the
3644                :  * indirect block for further modification.
3645                :  *
3646                :  * We release `count' blocks on disk, but (last - first) may be greater
3647                :  * than `count' because there can be holes in there.
3648                :  */
3649                : static void ext4_clear_blocks(handle_t *handle, struct inode *inode,
3650                :                               struct buffer_head *bh,
3651                :                               ext4_fsblk_t block_to_free,
3652                :                               unsigned long count, __le32 *first,
3653                :                               __le32 *last)
3654              9 : {
3655                :         __le32 *p;
3656              9 :         if (try_to_extend_transaction(handle, inode)) {
3657              0 :                 if (bh) {
3658                :                         BUFFER_TRACE(bh, "call ext4_handle_dirty_metadata");
3659              0 :                         ext4_handle_dirty_metadata(handle, inode, bh);
3660                :                 }
3661              0 :                 ext4_mark_inode_dirty(handle, inode);
3662              0 :                 ext4_journal_test_restart(handle, inode);
3663              0 :                 if (bh) {
3664                :                         BUFFER_TRACE(bh, "retaking write access");
3665              0 :                         ext4_journal_get_write_access(handle, bh);
3666                :                 }
3667                :         }
3668                :
3669                :         /*
3670                :          * Any buffers which are on the journal will be in memory. We
3671                :          * find them on the hash table so jbd2_journal_revoke() will
3672                :          * run jbd2_journal_forget() on them.  We've already detached
3673                :          * each block from the file, so bforget() in
3674                :          * jbd2_journal_forget() should be safe.
3675                :          *
3676                :          * AKPM: turn on bforget in jbd2_journal_forget()!!!
3677                :          */
3678            117 :         for (p = first; p < last; p++) {
3679            108 :                 u32 nr = le32_to_cpu(*p);
```

```
3680        108 :                   if (nr) {
3681            :                           struct buffer_head *tbh;
3682            :
3683         48 :                           *p = 0;
3684         96 :                           tbh = sb_find_get_block(inode->i_sb, nr);
3685         48 :                           ext4_forget(handle, 0, inode, tbh, nr);
3686            :                   }
3687            :           }
3688            :
3689          9 :           ext4_free_blocks(handle, inode, block_to_free, count, 0);
3690          9 : }
3691            :
3692            : /**
3693            :  * ext4_free_data - free a list of data blocks
3694            :  * @handle:     handle for this transaction
3695            :  * @inode:      inode we are dealing with
3696            :  * @this_bh:    indirect buffer_head which contains *@first and *@last
3697            :  * @first:      array of block numbers
3698            :  * @last:       points immediately past the end of array
3699            :  *
3700            :  * We are freeing all blocks refered from that array (numbers are stored as
3701            :  * little-endian 32-bit) and updating @inode->i_blocks appropriately.
3702            :  *
3703            :  * We accumulate contiguous runs of blocks to free.  Conveniently, if these
3704            :  * blocks are contiguous then releasing them at one time will only affect one
3705            :  * or two bitmap blocks (+ group descriptor(s) and superblock) and we won't
3706            :  * actually use a lot of journal space.
3707            :  *
3708            :  * @this_bh will be %NULL if @first and @last point into the inode's direct
3709            :  * block pointers.
3710            :  */
3711            : static void ext4_free_data(handle_t *handle, struct inode *inode,
3712            :                            struct buffer_head *this_bh,
3713            :                            __le32 *first, __le32 *last)
3714          9 : {
3715          9 :           ext4_fsblk_t block_to_free = 0;    /* Starting block # of a run */
3716          9 :           unsigned long count = 0;            /* Number of blocks in the run */
3717          9 :           __le32 *block_to_free_p = NULL;     /* Pointer into inode/ind
3718            :                                                  corresponding to
3719            :                                                  block_to_free */
3720            :           ext4_fsblk_t nr;                    /* Current block # */
3721            :           __le32 *p;                          /* Pointer into inode/ind
3722            :                                                  for current block */
3723            :           int err;
3724            :
3725          9 :           if (this_bh) {                          /* For indirect block */
3726            :                   BUFFER_TRACE(this_bh, "get_write_access");
3727          0 :                   err = ext4_journal_get_write_access(handle, this_bh);
3728            :                   /* Important: if we can't update the indirect pointers
3729            :                    * to the blocks, we can't free them. */
3730          0 :                   if (err)
3731          0 :                           return;
3732            :           }
3733            :
3734        117 :           for (p = first; p < last; p++) {
3735        108 :                   nr = le32_to_cpu(*p);
3736        108 :                   if (nr) {
3737            :                           /* accumulate blocks to free if they're contiguous */
3738         48 :                           if (count == 0) {
3739          9 :                                   block_to_free = nr;
3740          9 :                                   block_to_free_p = p;
3741          9 :                                   count = 1;
3742         39 :                           } else if (nr == block_to_free + count) {
3743         39 :                                   count++;
3744            :                           } else {
3745          0 :                                   ext4_clear_blocks(handle, inode, this_bh,
3746            :                                                     block_to_free,
3747            :                                                     count, block_to_free_p, p);
3748          0 :                                   block_to_free = nr;
3749          0 :                                   block_to_free_p = p;
3750          0 :                                   count = 1;
```

```
3751                :                         }
3752                :                 }
3753                :         }
3754                :
3755              9 :         if (count > 0)
3756              9 :                 ext4_clear_blocks(handle, inode, this_bh, block_to_free,
3757                :                                   count, block_to_free_p, p);
3758                :
3759              9 :         if (this_bh) {
3760                :                 BUFFER_TRACE(this_bh, "call ext4_handle_dirty_metadata");
3761                :
3762                :                 /*
3763                :                  * The buffer head should have an attached journal head at this
3764                :                  * point. However, if the data is corrupted and an indirect
3765                :                  * block pointed to itself, it would have been detached when
3766                :                  * the block was cleared. Check for this instead of OOPSing.
3767                :                  */
3768              0 :                 if ((EXT4_JOURNAL(inode) == NULL) || bh2jh(this_bh))
3769              0 :                         ext4_handle_dirty_metadata(handle, inode, this_bh);
3770                :                 else
3771              0 :                         ext4_error(inode->i_sb, __func__,
3772                :                                    "circular indirect block detected, "
3773                :                                    "inode=%lu, block=%llu",
3774                :                                    inode->i_ino,
3775                :                                    (unsigned long long) this_bh->b_blocknr);
3776                :         }
3777                : }
3778                :
3779                : /**
3780                :  *      ext4_free_branches - free an array of branches
3781                :  *      @handle: JBD handle for this transaction
3782                :  *      @inode: inode we are dealing with
3783                :  *      @parent_bh: the buffer_head which contains *@first and *@last
3784                :  *      @first: array of block numbers
3785                :  *      @last:  pointer immediately past the end of array
3786                :  *      @depth: depth of the branches to free
3787                :  *
3788                :  *      We are freeing all blocks refered from these branches (numbers are
3789                :  *      stored as little-endian 32-bit) and updating @inode->i_blocks
3790                :  *      appropriately.
3791                :  */
3792                : static void ext4_free_branches(handle_t *handle, struct inode *inode,
3793                :                                struct buffer_head *parent_bh,
3794                :                                __le32 *first, __le32 *last, int depth)
3795              0 : {
3796                :         ext4_fsblk_t nr;
3797                :         __le32 *p;
3798                :
3799              0 :         if (ext4_handle_is_aborted(handle))
3800              0 :                 return;
3801                :
3802              0 :         if (depth--) {
3803                :                 struct buffer_head *bh;
3804              0 :                 int addr_per_block = EXT4_ADDR_PER_BLOCK(inode->i_sb);
3805              0 :                 p = last;
3806              0 :                 while (--p >= first) {
3807              0 :                         nr = le32_to_cpu(*p);
3808              0 :                         if (!nr)
3809              0 :                                 continue;               /* A hole */
3810                :
3811                :                         /* Go read the buffer for the next level down */
3812              0 :                         bh = sb_bread(inode->i_sb, nr);
3813                :
3814                :                         /*
3815                :                          * A read failure? Report error and clear slot
3816                :                          * (should be rare).
3817                :                          */
3818              0 :                         if (!bh) {
3819              0 :                                 ext4_error(inode->i_sb, "ext4_free_branches",
3820                :                                            "Read failure, inode=%lu, block=%llu",
3821                :                                            inode->i_ino, nr);
```

```
3822        0 :                         continue;
3823          :                 }
3824          :
3825          :                 /* This zaps the entire block.  Bottom up. */
3826          :                 BUFFER_TRACE(bh, "free child branches");
3827        0 :                 ext4_free_branches(handle, inode, bh,
3828          :                                 (__le32 *) bh->b_data,
3829          :                                 (__le32 *) bh->b_data + addr_per_block,
3830          :                                 depth);
3831          :
3832          :                 /*
3833          :                  * We've probably journalled the indirect block several
3834          :                  * times during the truncate.  But it's no longer
3835          :                  * needed and we now drop it from the transaction via
3836          :                  * jbd2_journal_revoke().
3837          :                  *
3838          :                  * That's easy if it's exclusively part of this
3839          :                  * transaction.  But if it's part of the committing
3840          :                  * transaction then jbd2_journal_forget() will simply
3841          :                  * brelse() it.  That means that if the underlying
3842          :                  * block is reallocated in ext4_get_block(),
3843          :                  * unmap_underlying_metadata() will find this block
3844          :                  * and will try to get rid of it.  damn, damn.
3845          :                  *
3846          :                  * If this block has already been committed to the
3847          :                  * journal, a revoke record will be written.  And
3848          :                  * revoke records must be emitted *before* clearing
3849          :                  * this block's bit in the bitmaps.
3850          :                  */
3851        0 :                 ext4_forget(handle, 1, inode, bh, bh->b_blocknr);
3852          :
3853          :                 /*
3854          :                  * Everything below this this pointer has been
3855          :                  * released.  Now let this top-of-subtree go.
3856          :                  *
3857          :                  * We want the freeing of this indirect block to be
3858          :                  * atomic in the journal with the updating of the
3859          :                  * bitmap block which owns it.  So make some room in
3860          :                  * the journal.
3861          :                  *
3862          :                  * We zero the parent pointer *after* freeing its
3863          :                  * pointee in the bitmaps, so if extend_transaction()
3864          :                  * for some reason fails to put the bitmap changes and
3865          :                  * the release into the same transaction, recovery
3866          :                  * will merely complain about releasing a free block,
3867          :                  * rather than leaking blocks.
3868          :                  */
3869        0 :                 if (ext4_handle_is_aborted(handle))
3870        0 :                         return;
3871        0 :                 if (try_to_extend_transaction(handle, inode)) {
3872        0 :                         ext4_mark_inode_dirty(handle, inode);
3873        0 :                         ext4_journal_test_restart(handle, inode);
3874          :                 }
3875          :
3876        0 :                 ext4_free_blocks(handle, inode, nr, 1, 1);
3877          :
3878        0 :                 if (parent_bh) {
3879          :                         /*
3880          :                          * The block which we have just freed is
3881          :                          * pointed to by an indirect block: journal it
3882          :                          */
3883          :                         BUFFER_TRACE(parent_bh, "get_write_access");
3884        0 :                         if (!ext4_journal_get_write_access(handle,
3885          :                                                            parent_bh)){
3886        0 :                                 *p = 0;
3887          :                                 BUFFER_TRACE(parent_bh,
3888          :                                 "call ext4_handle_dirty_metadata");
3889        0 :                                 ext4_handle_dirty_metadata(handle,
3890          :                                                            inode,
3891          :                                                            parent_bh);
3892          :                         }
```

```
3893                :                         }
3894                :                 }
3895                :         } else {
3896                :                 /* We have reached the bottom of the tree. */
3897                :                 BUFFER_TRACE(parent_bh, "free data blocks");
3898              0 :                 ext4_free_data(handle, inode, parent_bh, first, last);
3899                :         }
3900                : }
3901                :
3902                : int ext4_can_truncate(struct inode *inode)
3903        1662913 : {
3904        1662913 :         if (IS_APPEND(inode) || IS_IMMUTABLE(inode))
3905              0 :                 return 0;
3906        1662913 :         if (S_ISREG(inode->i_mode))
3907        1007503 :                 return 1;
3908         655410 :         if (S_ISDIR(inode->i_mode))
3909         655410 :                 return 1;
3910              0 :         if (S_ISLNK(inode->i_mode))
3911              0 :                 return !ext4_inode_is_fast_symlink(inode);
3912              0 :         return 0;
3913                : }
3914                :
3915                : /*
3916                :  * ext4_truncate()
3917                :  *
3918                :  * We block out ext4_get_block() block instantiations across the entire
3919                :  * transaction, and VFS/VM ensures that ext4_truncate() cannot run
3920                :  * simultaneously on behalf of the same inode.
3921                :  *
3922                :  * As we work through the truncate and commmit bits of it to the journal there
3923                :  * is one core, guiding principle: the file's tree must always be consistent on
3924                :  * disk.  We must be able to restart the truncate after a crash.
3925                :  *
3926                :  * The file's tree may be transiently inconsistent in memory (although it
3927                :  * probably isn't), but whenever we close off and commit a journal transaction,
3928                :  * the contents of (the filesystem + the journal) must be consistent and
3929                :  * restartable.  It's pretty simple, really: bottom up, right to left (although
3930                :  * left-to-right works OK too).
3931                :  *
3932                :  * Note that at recovery time, journal replay occurs *before* the restart of
3933                :  * truncate against the orphan inode list.
3934                :  *
3935                :  * The committed inode has the new, desired i_size (which is the same as
3936                :  * i_disksize in this case).  After a crash, ext4_orphan_cleanup() will see
3937                :  * that this inode's truncate did not complete and it will again call
3938                :  * ext4_truncate() to have another go.  So there will be instantiated blocks
3939                :  * to the right of the truncation point in a crashed ext4 filesystem.  But
3940                :  * that's fine - as long as they are linked from the inode, the post-crash
3941                :  * ext4_truncate() run will find them and release them.
3942                :  */
3943                : void ext4_truncate(struct inode *inode)
3944        1662905 : {
3945                :         handle_t *handle;
3946        1662905 :         struct ext4_inode_info *ei = EXT4_I(inode);
3947        1662905 :         __le32 *i_data = ei->i_data;
3948        1662905 :         int addr_per_block = EXT4_ADDR_PER_BLOCK(inode->i_sb);
3949        1662905 :         struct address_space *mapping = inode->i_mapping;
3950                :         ext4_lblk_t offsets[4];
3951                :         Indirect chain[4];
3952                :         Indirect *partial;
3953        1662905 :         __le32 nr = 0;
3954                :         int n;
3955                :         ext4_lblk_t last_block;
3956        1662905 :         unsigned blocksize = inode->i_sb->s_blocksize;
3957                :
3958        1662905 :         if (!ext4_can_truncate(inode))
3959              0 :                 return;
3960                :
3961        3325802 :         if (ei->i_disksize && inode->i_size == 0 &&
3962                :             !test_opt(inode->i_sb, NO_AUTO_DA_ALLOC))
3963        1662896 :                 ei->i_state |= EXT4_STATE_DA_ALLOC_CLOSE;
```

```
3964                :
3965          1662906 :         if (EXT4_I(inode)->i_flags & EXT4_EXTENTS_FL) {
3966          1662897 :                 ext4_ext_truncate(inode);
3967          1662895 :                 return;
3968                :         }
3969                :
3970                9 :         handle = start_transaction(inode);
3971                9 :         if (IS_ERR(handle))
3972                0 :                 return;         /* AKPM: return what? */
3973                :
3974                9 :         last_block = (inode->i_size + blocksize-1)
3975                :                                         >> EXT4_BLOCK_SIZE_BITS(inode->i_sb);
3976                :
3977                9 :         if (inode->i_size & (blocksize - 1))
3978                0 :                 if (ext4_block_truncate_page(handle, mapping, inode->i_size))
3979                0 :                         goto out_stop;
3980                :
3981                9 :         n = ext4_block_to_path(inode, last_block, offsets, NULL);
3982                9 :         if (n == 0)
3983                0 :                 goto out_stop;  /* error */
3984                :
3985                :         /*
3986                :          * OK.  This truncate is going to happen.  We add the inode to the
3987                :          * orphan list, so that if this truncate spans multiple transactions,
3988                :          * and we crash, we will resume the truncate when the filesystem
3989                :          * recovers.  It also marks the inode dirty, to catch the new size.
3990                :          *
3991                :          * Implication: the file must always be in a sane, consistent
3992                :          * truncatable state while each transaction commits.
3993                :          */
3994                9 :         if (ext4_orphan_add(handle, inode))
3995                0 :                 goto out_stop;
3996                :
3997                :         /*
3998                :          * From here we block out all ext4_get_block() callers who want to
3999                :          * modify the block allocation tree.
4000                :          */
4001                9 :         down_write(&ei->i_data_sem);
4002                :
4003                9 :         ext4_discard_preallocations(inode);
4004                :
4005                :         /*
4006                :          * The orphan list entry will now protect us from any crash which
4007                :          * occurs before the truncate completes, so it is now safe to propagate
4008                :          * the new, shorter inode size (held for now in i_size) into the
4009                :          * on-disk inode. We do this via i_disksize, which is the value which
4010                :          * ext4 *really* writes onto the disk inode.
4011                :          */
4012                9 :         ei->i_disksize = inode->i_size;
4013                :
4014                9 :         if (n == 1) {           /* direct blocks */
4015                9 :                 ext4_free_data(handle, inode, NULL, i_data+offsets[0],
4016                :                                i_data + EXT4_NDIR_BLOCKS);
4017                9 :                 goto do_indirects;
4018                :         }
4019                :
4020                0 :         partial = ext4_find_shared(inode, n, offsets, chain, &nr);
4021                :         /* Kill the top of shared branch (not detached) */
4022                0 :         if (nr) {
4023                0 :                 if (partial == chain) {
4024                :                         /* Shared branch grows from the inode */
4025                0 :                         ext4_free_branches(handle, inode, NULL,
4026                :                                            &nr, &nr+1, (chain+n-1) - partial);
4027                0 :                         *partial->p = 0;
4028                :                         /*
4029                :                          * We mark the inode dirty prior to restart,
4030                :                          * and prior to stop.  No need for it here.
4031                :                          */
4032                :                 } else {
4033                :                         /* Shared branch grows from an indirect block */
4034                :                         BUFFER_TRACE(partial->bh, "get_write_access");
```

```
4035           0 :                         ext4_free_branches(handle, inode, partial->bh,
4036             :                                            partial->p,
4037             :                                            partial->p+1, (chain+n-1) - partial);
4038             :                 }
4039             :         }
4040             :         /* Clear the ends of indirect blocks on the shared branch */
4041           0 :         while (partial > chain) {
4042           0 :                 ext4_free_branches(handle, inode, partial->bh, partial->p + 1,
4043             :                                    (__le32*)partial->bh->b_data+addr_per_block,
4044             :                                    (chain+n-1) - partial);
4045             :                 BUFFER_TRACE(partial->bh, "call brelse");
4046           0 :                 brelse(partial->bh);
4047           0 :                 partial--;
4048             :         }
4049           9 : do_indirects:
4050             :         /* Kill the remaining (whole) subtrees */
4051           9 :         switch (offsets[0]) {
4052             :         default:
4053           9 :                 nr = i_data[EXT4_IND_BLOCK];
4054           9 :                 if (nr) {
4055           0 :                         ext4_free_branches(handle, inode, NULL, &nr, &nr+1, 1);
4056           0 :                         i_data[EXT4_IND_BLOCK] = 0;
4057             :                 }
4058             :         case EXT4_IND_BLOCK:
4059           9 :                 nr = i_data[EXT4_DIND_BLOCK];
4060           9 :                 if (nr) {
4061           0 :                         ext4_free_branches(handle, inode, NULL, &nr, &nr+1, 2);
4062           0 :                         i_data[EXT4_DIND_BLOCK] = 0;
4063             :                 }
4064             :         case EXT4_DIND_BLOCK:
4065           9 :                 nr = i_data[EXT4_TIND_BLOCK];
4066           9 :                 if (nr) {
4067           0 :                         ext4_free_branches(handle, inode, NULL, &nr, &nr+1, 3);
4068           0 :                         i_data[EXT4_TIND_BLOCK] = 0;
4069             :                 }
4070             :         case EXT4_TIND_BLOCK:
4071             :                 ;
4072             :         }
4073             :
4074           9 :         up_write(&ei->i_data_sem);
4075           9 :         inode->i_mtime = inode->i_ctime = ext4_current_time(inode);
4076           9 :         ext4_mark_inode_dirty(handle, inode);
4077             :
4078             :         /*
4079             :          * In a multi-transaction truncate, we only make the final transaction
4080             :          * synchronous
4081             :          */
4082           9 :         if (IS_SYNC(inode))
4083             :                 ext4_handle_sync(handle);
4084           9 : out_stop:
4085             :         /*
4086             :          * If this was a simple ftruncate(), and the file will remain alive
4087             :          * then we need to clear up the orphan record which we created above.
4088             :          * However, if this was a real unlink then we were called by
4089             :          * ext4_delete_inode(), and we allow that function to clean up the
4090             :          * orphan info for us.
4091             :          */
4092           9 :         if (inode->i_nlink)
4093           0 :                 ext4_orphan_del(handle, inode);
4094             :
4095           9 :         ext4_journal_stop(handle);
4096             : }
4097             :
4098             : /*
4099             :  * ext4_get_inode_loc returns with an extra refcount against the inode's
4100             :  * underlying buffer_head on success. If 'in_mem' is true, we have all
4101             :  * data in memory that is needed to recreate the on-disk version of this
4102             :  * inode.
4103             :  */
4104             : static int __ext4_get_inode_loc(struct inode *inode,
4105             :                                 struct ext4_iloc *iloc, int in_mem)
```

```
4106    1455390821 : {
4107               :         struct ext4_group_desc  *gdp;
4108               :         struct buffer_head      *bh;
4109    1455390821 :         struct super_block      *sb = inode->i_sb;
4110               :         ext4_fsblk_t            block;
4111               :         int                     inodes_per_block, inode_offset;
4112               :
4113    1455390821 :         iloc->bh = NULL;
4114   -1384185654 :         if (!ext4_valid_inum(sb, inode->i_ino))
4115             0 :                 return -EIO;
4116               :
4117   -1384185654 :         iloc->block_group = (inode->i_ino - 1) / EXT4_INODES_PER_GROUP(sb);
4118    1455390821 :         gdp = ext4_get_group_desc(sb, iloc->block_group, NULL);
4119    1430431708 :         if (!gdp)
4120             0 :                 return -EIO;
4121               :
4122               :         /*
4123               :          * Figure out the offset within the block group inode table
4124               :          */
4125   -1434103880 :         inodes_per_block = (EXT4_BLOCK_SIZE(sb) / EXT4_INODE_SIZE(sb));
4126   -1434103880 :         inode_offset = ((inode->i_ino - 1) %
4127               :                         EXT4_INODES_PER_GROUP(sb));
4128    1430431708 :         block = ext4_inode_table(sb, gdp) + (inode_offset / inodes_per_block);
4129   -1434414824 :         iloc->offset = (inode_offset % inodes_per_block) * EXT4_INODE_SIZE(sb);
4130               :
4131    1428780663 :         bh = sb_getblk(sb, block);
4132    1428780663 :         if (!bh) {
4133             0 :                 ext4_error(sb, "ext4_get_inode_loc", "unable to read "
4134               :                            "inode block - inode=%lu, block=%llu",
4135               :                            inode->i_ino, block);
4136             0 :                 return -EIO;
4137               :         }
4138    1428780663 :         if (!buffer_uptodate(bh)) {
4139        870741 :                 lock_buffer(bh);
4140               :
4141               :                 /*
4142               :                  * If the buffer has the write error flag, we have failed
4143               :                  * to write out another inode in the same block.  In this
4144               :                  * case, we don't have to read the block because we may
4145               :                  * read the old inode data successfully.
4146               :                  */
4147        870741 :                 if (buffer_write_io_error(bh) && !buffer_uptodate(bh))
4148               :                         set_buffer_uptodate(bh);
4149               :
4150        870741 :                 if (buffer_uptodate(bh)) {
4151               :                         /* someone brought it uptodate while we waited */
4152             8 :                         unlock_buffer(bh);
4153             8 :                         goto has_buffer;
4154               :                 }
4155               :
4156               :                 /*
4157               :                  * If we have all information of the inode in memory and this
4158               :                  * is the only valid inode in the block, we need not read the
4159               :                  * block.
4160               :                  */
4161        870733 :                 if (in_mem) {
4162               :                         struct buffer_head *bitmap_bh;
4163               :                         int i, start;
4164               :
4165        860193 :                         start = inode_offset & ~(inodes_per_block - 1);
4166               :
4167               :                         /* Is the inode bitmap in cache? */
4168       1720386 :                         bitmap_bh = sb_getblk(sb, ext4_inode_bitmap(sb, gdp));
4169        860193 :                         if (!bitmap_bh)
4170             0 :                                 goto make_io;
4171               :
4172               :                         /*
4173               :                          * If the inode bitmap isn't in cache then the
4174               :                          * optimisation may end up performing two reads instead
4175               :                          * of one, so skip it.
4176               :                          */
```

```
    4177          860193 :                         if (!buffer_uptodate(bitmap_bh)) {
    4178               3 :                                 brelse(bitmap_bh);
    4179               3 :                                 goto make_io;
    4180                 :                         }
    4181        10370356 :                         for (i = start; i < start + inodes_per_block; i++) {
    4182         9510303 :                                 if (i == inode_offset)
    4183          860059 :                                         continue;
    4184        17300488 :                                 if (ext4_test_bit(i, bitmap_bh->b_data))
    4185             137 :                                         break;
    4186                 :                         }
    4187          860190 :                         brelse(bitmap_bh);
    4188          860190 :                         if (i == start + inodes_per_block) {
    4189                 :                                 /* all other inodes are free, so skip I/O */
    4190         1720106 :                                 memset(bh->b_data, 0, bh->b_size);
    4191                 :                                 set_buffer_uptodate(bh);
    4192          860053 :                                 unlock_buffer(bh);
    4193          860053 :                                 goto has_buffer;
    4194                 :                         }
    4195                 :                 }
    4196                 :
    4197           10680 : make_io:
    4198                 :                 /*
    4199                 :                  * If we need to do any I/O, try to pre-readahead extra
    4200                 :                  * blocks from the inode table.
    4201                 :                  */
    4202           10680 :                 if (EXT4_SB(sb)->s_inode_readahead_blks) {
    4203                 :                         ext4_fsblk_t b, end, table;
    4204                 :                         unsigned num;
    4205                 :
    4206           10680 :                         table = ext4_inode_table(sb, gdp);
    4207                 :                         /* s_inode_readahead_blks is always a power of 2 */
    4208           10680 :                         b = block & ~(EXT4_SB(sb)->s_inode_readahead_blks-1);
    4209           10680 :                         if (table > b)
    4210             409 :                                 b = table;
    4211           10680 :                         end = b + EXT4_SB(sb)->s_inode_readahead_blks;
    4212           10680 :                         num = EXT4_INODES_PER_GROUP(sb);
    4213           10680 :                         if (EXT4_HAS_RO_COMPAT_FEATURE(sb,
    4214                 :                                                        EXT4_FEATURE_RO_COMPAT_GDT_CSUM))
    4215           10663 :                                 num -= ext4_itable_unused_count(sb, gdp);
    4216           10680 :                         table += num / inodes_per_block;
    4217           10680 :                         if (end > table)
    4218             400 :                                 end = table;
    4219          352826 :                         while (b <= end)
    4220          342146 :                                 sb_breadahead(sb, b++);
    4221                 :                 }
    4222                 :
    4223                 :                 /*
    4224                 :                  * There are other valid inodes in the buffer, this inode
    4225                 :                  * has in-inode xattrs, or we don't have this inode in memory.
    4226                 :                  * Read the block from disk.
    4227                 :                  */
    4228                 :                 get_bh(bh);
    4229           10680 :                 bh->b_end_io = end_buffer_read_sync;
    4230           10680 :                 submit_bh(READ_META, bh);
    4231           10680 :                 wait_on_buffer(bh);
    4232           10680 :                 if (!buffer_uptodate(bh)) {
    4233               0 :                         ext4_error(sb, __func__,
    4234                 :                                    "unable to read inode block - inode=%lu, "
    4235                 :                                    "block=%llu", inode->i_ino, block);
    4236               0 :                         brelse(bh);
    4237               0 :                         return -EIO;
    4238                 :                 }
    4239                 :         }
    4240    1429660495 : has_buffer:
    4241    1429660495 :         iloc->bh = bh;
    4242    1429660495 :         return 0;
    4243                 : }
    4244                 :
    4245                 : int ext4_get_inode_loc(struct inode *inode, struct ext4_iloc *iloc)
    4246    1430643729 : {
    4247                 :         /* We have all inode data except xattrs in memory here. */
```

```
4248  1430643729 :         return __ext4_get_inode_loc(inode, iloc,
4249             :                 !(EXT4_I(inode)->i_state & EXT4_STATE_XATTR));
4250             : }
4251             :
4252             : void ext4_set_inode_flags(struct inode *inode)
4253    13086182 : {
4254    13086182 :         unsigned int flags = EXT4_I(inode)->i_flags;
4255             :
4256    13086182 :         inode->i_flags &= ~(S_SYNC|S_APPEND|S_IMMUTABLE|S_NOATIME|S_DIRSYNC);
4257    13086182 :         if (flags & EXT4_SYNC_FL)
4258           0 :                 inode->i_flags |= S_SYNC;
4259    13086182 :         if (flags & EXT4_APPEND_FL)
4260           0 :                 inode->i_flags |= S_APPEND;
4261    13086182 :         if (flags & EXT4_IMMUTABLE_FL)
4262           0 :                 inode->i_flags |= S_IMMUTABLE;
4263    13086182 :         if (flags & EXT4_NOATIME_FL)
4264           0 :                 inode->i_flags |= S_NOATIME;
4265    13086182 :         if (flags & EXT4_DIRSYNC_FL)
4266           0 :                 inode->i_flags |= S_DIRSYNC;
4267    13086182 : }
4268             :
4269             : /* Propagate flags from i_flags to EXT4_I(inode)->i_flags */
4270             : void ext4_get_inode_flags(struct ext4_inode_info *ei)
4271  1426054014 : {
4272  1426054014 :         unsigned int flags = ei->vfs_inode.i_flags;
4273             :
4274  1426054014 :         ei->i_flags &= ~(EXT4_SYNC_FL|EXT4_APPEND_FL|
4275             :                         EXT4_IMMUTABLE_FL|EXT4_NOATIME_FL|EXT4_DIRSYNC_FL);
4276  1426054014 :         if (flags & S_SYNC)
4277           0 :                 ei->i_flags |= EXT4_SYNC_FL;
4278  1426054014 :         if (flags & S_APPEND)
4279           0 :                 ei->i_flags |= EXT4_APPEND_FL;
4280  1426054014 :         if (flags & S_IMMUTABLE)
4281           0 :                 ei->i_flags |= EXT4_IMMUTABLE_FL;
4282  1426054014 :         if (flags & S_NOATIME)
4283           0 :                 ei->i_flags |= EXT4_NOATIME_FL;
4284  1426054014 :         if (flags & S_DIRSYNC)
4285           0 :                 ei->i_flags |= EXT4_DIRSYNC_FL;
4286  1426054014 : }
4287             :
4288             : static blkcnt_t ext4_inode_blocks(struct ext4_inode *raw_inode,
4289             :                                   struct ext4_inode_info *ei)
4290             : {
4291             :         blkcnt_t i_blocks ;
4292     3773613 :         struct inode *inode = &(ei->vfs_inode);
4293     3773613 :         struct super_block *sb = inode->i_sb;
4294             :
4295     3773613 :         if (EXT4_HAS_RO_COMPAT_FEATURE(sb,
4296             :                                 EXT4_FEATURE_RO_COMPAT_HUGE_FILE)) {
4297             :                 /* we are using combined 48 bit field */
4298     3761305 :                 i_blocks = ((u64)le16_to_cpu(raw_inode->i_blocks_high)) << 32 |
4299             :                                         le32_to_cpu(raw_inode->i_blocks_lo);
4300     3761305 :                 if (ei->i_flags & EXT4_HUGE_FILE_FL) {
4301             :                         /* i_blocks represent file system block size */
4302           2 :                         return i_blocks  << (inode->i_blkbits - 9);
4303             :                 } else {
4304     3761303 :                         return i_blocks;
4305             :                 }
4306             :         } else {
4307       12308 :                 return le32_to_cpu(raw_inode->i_blocks_lo);
4308             :         }
4309             : }
4310             :
4311             : struct inode *ext4_iget(struct super_block *sb, unsigned long ino)
4312     4917331 : {
4313             :         struct ext4_iloc iloc;
4314             :         struct ext4_inode *raw_inode;
4315             :         struct ext4_inode_info *ei;
4316             :         struct buffer_head *bh;
4317             :         struct inode *inode;
4318             :         long ret;
```

```
4319                :          int block;
4320                :
4321          4917331 :          inode = iget_locked(sb, ino);
4322          4917410 :          if (!inode)
4323                0 :                  return ERR_PTR(-ENOMEM);
4324          4917410 :          if (!(inode->i_state & I_NEW))
4325          1143797 :                  return inode;
4326                :
4327          3773613 :          ei = EXT4_I(inode);
4328                :
4329          3773613 :          ret = __ext4_get_inode_loc(inode, &iloc, 0);
4330          3773613 :          if (ret < 0)
4331                0 :                  goto bad_inode;
4332          3773613 :          bh = iloc.bh;
4333          3773613 :          raw_inode = ext4_raw_inode(&iloc);
4334          3773613 :          inode->i_mode = le16_to_cpu(raw_inode->i_mode);
4335          3773613 :          inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
4336          3773613 :          inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
4337          7547226 :          if (!(test_opt(inode->i_sb, NO_UID32))) {
4338          3773613 :                  inode->i_uid |= le16_to_cpu(raw_inode->i_uid_high) << 16;
4339          3773613 :                  inode->i_gid |= le16_to_cpu(raw_inode->i_gid_high) << 16;
4340                :          }
4341          3773613 :          inode->i_nlink = le16_to_cpu(raw_inode->i_links_count);
4342                :
4343          3773613 :          ei->i_state = 0;
4344          3773613 :          ei->i_dir_start_lookup = 0;
4345          3773613 :          ei->i_dtime = le32_to_cpu(raw_inode->i_dtime);
4346                :          /* We now have enough fields to check if the inode was active or not.
4347                :           * This is needed because nfsd might try to access dead inodes
4348                :           * the test is that same one that e2fsck uses
4349                :           * NeilBrown 1999oct15
4350                :           */
4351          3773613 :          if (inode->i_nlink == 0) {
4352                0 :                  if (inode->i_mode == 0 ||
4353                :                      !(EXT4_SB(inode->i_sb)->s_mount_state & EXT4_ORPHAN_FS)) {
4354                :                          /* this inode is deleted */
4355                0 :                          brelse(bh);
4356                0 :                          ret = -ESTALE;
4357                0 :                          goto bad_inode;
4358                :                  }
4359                :                  /* The only unlinked inodes we let through here have
4360                :                   * valid i_mode and are being read by the orphan
4361                :                   * recovery code: that's fine, we're about to complete
4362                :                   * the process of deleting those. */
4363                :          }
4364          3773613 :          ei->i_flags = le32_to_cpu(raw_inode->i_flags);
4365          3773613 :          inode->i_blocks = ext4_inode_blocks(raw_inode, ei);
4366          3773613 :          ei->i_file_acl = le32_to_cpu(raw_inode->i_file_acl_lo);
4367          3773613 :          if (EXT4_HAS_INCOMPAT_FEATURE(sb, EXT4_FEATURE_INCOMPAT_64BIT))
4368                0 :                  ei->i_file_acl |=
4369                :                          ((__u64)le16_to_cpu(raw_inode->i_file_acl_high)) << 32;
4370          3773613 :          inode->i_size = ext4_isize(raw_inode);
4371          3773613 :          ei->i_disksize = inode->i_size;
4372          3773613 :          inode->i_generation = le32_to_cpu(raw_inode->i_generation);
4373          3773613 :          ei->i_block_group = iloc.block_group;
4374          3773613 :          ei->i_last_alloc_group = ~0;
4375                :          /*
4376                :           * NOTE! The in-memory inode i_data array is in little-endian order
4377                :           * even on big-endian machines: we do NOT byteswap the block numbers!
4378                :           */
4379         60377800 :          for (block = 0; block < EXT4_N_BLOCKS; block++)
4380         56604187 :                  ei->i_data[block] = raw_inode->i_block[block];
4381          3773613 :          INIT_LIST_HEAD(&ei->i_orphan);
4382                :
4383          7547226 :          if (EXT4_INODE_SIZE(inode->i_sb) > EXT4_GOOD_OLD_INODE_SIZE) {
4384          3773609 :                  ei->i_extra_isize = le16_to_cpu(raw_inode->i_extra_isize);
4385          7547218 :                  if (EXT4_GOOD_OLD_INODE_SIZE + ei->i_extra_isize >
4386                :                      EXT4_INODE_SIZE(inode->i_sb)) {
4387                0 :                          brelse(bh);
4388                0 :                          ret = -EIO;
4389                0 :                          goto bad_inode;
```

```
4390                :                 }
4391        3773609 :                 if (ei->i_extra_isize == 0) {
4392                :                         /* The extra space is currently unused. Use it. */
4393              0 :                         ei->i_extra_isize = sizeof(struct ext4_inode) -
4394                :                                             EXT4_GOOD_OLD_INODE_SIZE;
4395                :                 } else {
4396                :                         __le32 *magic = (void *)raw_inode +
4397                :                                         EXT4_GOOD_OLD_INODE_SIZE +
4398        3773609 :                                         ei->i_extra_isize;
4399        3773609 :                         if (*magic == cpu_to_le32(EXT4_XATTR_MAGIC))
4400              0 :                                 ei->i_state |= EXT4_STATE_XATTR;
4401                :                 }
4402                :         } else
4403              4 :                 ei->i_extra_isize = 0;
4404                :
4405        7547226 :         EXT4_INODE_GET_XTIME(i_ctime, inode, raw_inode);
4406        7547226 :         EXT4_INODE_GET_XTIME(i_mtime, inode, raw_inode);
4407        7547226 :         EXT4_INODE_GET_XTIME(i_atime, inode, raw_inode);
4408        3773613 :         EXT4_EINODE_GET_XTIME(i_crtime, ei, raw_inode);
4409                :
4410        3773613 :         inode->i_version = le32_to_cpu(raw_inode->i_disk_version);
4411        7547226 :         if (EXT4_INODE_SIZE(inode->i_sb) > EXT4_GOOD_OLD_INODE_SIZE) {
4412        3773609 :                 if (EXT4_FITS_IN_INODE(raw_inode, ei, i_version_hi))
4413        3773609 :                         inode->i_version |=
4414                :                         (__u64)(le32_to_cpu(raw_inode->i_version_hi)) << 32;
4415                :         }
4416                :
4417        3773613 :         ret = 0;
4418        3773613 :         if (ei->i_file_acl &&
4419                :             ((ei->i_file_acl <
4420                :               (le32_to_cpu(EXT4_SB(sb)->s_es->s_first_data_block) +
4421                :                EXT4_SB(sb)->s_gdb_count)) ||
4422                :              (ei->i_file_acl >= ext4_blocks_count(EXT4_SB(sb)->s_es)))) {
4423              2 :                 ext4_error(sb, __func__,
4424                :                            "bad extended attribute block %llu in inode #%lu",
4425                :                            ei->i_file_acl, inode->i_ino);
4426              0 :                 ret = -EIO;
4427              0 :                 goto bad_inode;
4428        3773611 :         } else if (ei->i_flags & EXT4_EXTENTS_FL) {
4429        3761217 :                 if (S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) ||
4430                :                     (S_ISLNK(inode->i_mode) &&
4431                :                      !ext4_inode_is_fast_symlink(inode)))
4432                :                         /* Validate extent which is part of inode */
4433        3761216 :                         ret = ext4_ext_check_inode(inode);
4434          12394 :         } else if (S_ISREG(inode->i_mode) || S_ISDIR(inode->i_mode) ||
4435                :                    (S_ISLNK(inode->i_mode) &&
4436                :                     !ext4_inode_is_fast_symlink(inode))) {
4437                :                 /* Validate block references which are part of inode */
4438          12394 :                 ret = ext4_check_inode_blockref(inode);
4439                :         }
4440        3773611 :         if (ret) {
4441              0 :                 brelse(bh);
4442              0 :                 goto bad_inode;
4443                :         }
4444                :
4445        3773611 :         if (S_ISREG(inode->i_mode)) {
4446        3773486 :                 inode->i_op = &ext4_file_inode_operations;
4447        3773486 :                 inode->i_fop = &ext4_file_operations;
4448        3773486 :                 ext4_set_aops(inode);
4449            125 :         } else if (S_ISDIR(inode->i_mode)) {
4450            125 :                 inode->i_op = &ext4_dir_inode_operations;
4451            125 :                 inode->i_fop = &ext4_dir_operations;
4452              0 :         } else if (S_ISLNK(inode->i_mode)) {
4453              0 :                 if (ext4_inode_is_fast_symlink(inode)) {
4454              0 :                         inode->i_op = &ext4_fast_symlink_inode_operations;
4455              0 :                         nd_terminate_link(ei->i_data, inode->i_size,
4456                :                                 sizeof(ei->i_data) - 1);
4457                :                 } else {
4458              0 :                         inode->i_op = &ext4_symlink_inode_operations;
4459              0 :                         ext4_set_aops(inode);
4460                :                 }
```

```
4461                0 :         } else if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode) ||
4462                  :                    S_ISFIFO(inode->i_mode) || S_ISSOCK(inode->i_mode)) {
4463                0 :                 inode->i_op = &ext4_special_inode_operations;
4464                0 :                 if (raw_inode->i_block[0])
4465                0 :                         init_special_inode(inode, inode->i_mode,
4466                  :                            old_decode_dev(le32_to_cpu(raw_inode->i_block[0])));
4467                  :                 else
4468                0 :                         init_special_inode(inode, inode->i_mode,
4469                  :                            new_decode_dev(le32_to_cpu(raw_inode->i_block[1])));
4470                  :         } else {
4471                0 :                 brelse(bh);
4472                0 :                 ret = -EIO;
4473                0 :                 ext4_error(inode->i_sb, __func__,
4474                  :                            "bogus i_mode (%o) for inode=%lu",
4475                  :                            inode->i_mode, inode->i_ino);
4476                0 :                 goto bad_inode;
4477                  :         }
4478          3773613 :         brelse(iloc.bh);
4479          3773611 :         ext4_set_inode_flags(inode);
4480          3773612 :         unlock_new_inode(inode);
4481          3773610 :         return inode;
4482                  :
4483                0 : bad_inode:
4484                0 :         iget_failed(inode);
4485                0 :         return ERR_PTR(ret);
4486                  : }
4487                  :
4488                  : static int ext4_inode_blocks_set(handle_t *handle,
4489                  :                                  struct ext4_inode *raw_inode,
4490                  :                                  struct ext4_inode_info *ei)
4491                  : {
4492       1430288015 :         struct inode *inode = &(ei->vfs_inode);
4493       1430288015 :         u64 i_blocks = inode->i_blocks;
4494       1430288015 :         struct super_block *sb = inode->i_sb;
4495                  :
4496       1430288015 :         if (i_blocks <= ~0U) {
4497                  :                 /*
4498                  :                  * i_blocks can be represnted in a 32 bit variable
4499                  :                  * as multiple of 512 bytes
4500                  :                  */
4501       1430288015 :                 raw_inode->i_blocks_lo   = cpu_to_le32(i_blocks);
4502       1430288015 :                 raw_inode->i_blocks_high = 0;
4503       1430288015 :                 ei->i_flags &= ~EXT4_HUGE_FILE_FL;
4504       1430288015 :                 return 0;
4505                  :         }
4506                0 :         if (!EXT4_HAS_RO_COMPAT_FEATURE(sb, EXT4_FEATURE_RO_COMPAT_HUGE_FILE))
4507                0 :                 return -EFBIG;
4508                  :
4509                0 :         if (i_blocks <= 0xffffffffffffULL) {
4510                  :                 /*
4511                  :                  * i_blocks can be represented in a 48 bit variable
4512                  :                  * as multiple of 512 bytes
4513                  :                  */
4514                0 :                 raw_inode->i_blocks_lo   = cpu_to_le32(i_blocks);
4515                0 :                 raw_inode->i_blocks_high = cpu_to_le16(i_blocks >> 32);
4516                0 :                 ei->i_flags &= ~EXT4_HUGE_FILE_FL;
4517                  :         } else {
4518                0 :                 ei->i_flags |= EXT4_HUGE_FILE_FL;
4519                  :                 /* i_block is stored in file system block size */
4520                0 :                 i_blocks = i_blocks >> (inode->i_blkbits - 9);
4521                0 :                 raw_inode->i_blocks_lo   = cpu_to_le32(i_blocks);
4522                0 :                 raw_inode->i_blocks_high = cpu_to_le16(i_blocks >> 32);
4523                  :         }
4524                0 :         return 0;
4525                  : }
4526                  :
4527                  : /*
4528                  :  * Post the struct inode info into an on-disk inode location in the
4529                  :  * buffer-cache.  This gobbles the caller's reference to the
4530                  :  * buffer_head in the inode location struct.
4531                  :  *
```

```
4532                :  * The caller must have write access to iloc->bh.
4533                :  */
4534                : static int ext4_do_update_inode(handle_t *handle,
4535                :                                 struct inode *inode,
4536                :                                 struct ext4_iloc *iloc)
4537     1451885169 : {
4538     1451885169 :         struct ext4_inode *raw_inode = ext4_raw_inode(iloc);
4539     1451885169 :         struct ext4_inode_info *ei = EXT4_I(inode);
4540     1451885169 :         struct buffer_head *bh = iloc->bh;
4541     1451885169 :         int err = 0, rc, block;
4542                :
4543                :         /* For fields not not tracking in the in-memory inode,
4544                :          * initialise them to zero for new inodes. */
4545     1451885169 :         if (ei->i_state & EXT4_STATE_NEW)
4546       18625144 :                 memset(raw_inode, 0, EXT4_SB(inode->i_sb)->s_inode_size);
4547                :
4548     1451885169 :         ext4_get_inode_flags(ei);
4549     1430288015 :         raw_inode->i_mode = cpu_to_le16(inode->i_mode);
4550    -1434391266 :         if (!(test_opt(inode->i_sb, NO_UID32))) {
4551     1430288015 :                 raw_inode->i_uid_low = cpu_to_le16(low_16_bits(inode->i_uid));
4552     1430288015 :                 raw_inode->i_gid_low = cpu_to_le16(low_16_bits(inode->i_gid));
4553                : /*
4554                :  * Fix up interoperability with old kernels. Otherwise, old inodes get
4555                :  * re-used with the upper 16 bits of the uid/gid intact
4556                :  */
4557     1430288015 :                 if (!ei->i_dtime) {
4558     1427791161 :                         raw_inode->i_uid_high =
4559                :                                 cpu_to_le16(high_16_bits(inode->i_uid));
4560     1427791161 :                         raw_inode->i_gid_high =
4561                :                                 cpu_to_le16(high_16_bits(inode->i_gid));
4562                :                 } else {
4563        2496854 :                         raw_inode->i_uid_high = 0;
4564        2496854 :                         raw_inode->i_gid_high = 0;
4565                :                 }
4566                :         } else {
4567              0 :                 raw_inode->i_uid_low =
4568                :                         cpu_to_le16(fs_high2lowuid(inode->i_uid));
4569              0 :                 raw_inode->i_gid_low =
4570                :                         cpu_to_le16(fs_high2lowgid(inode->i_gid));
4571              0 :                 raw_inode->i_uid_high = 0;
4572              0 :                 raw_inode->i_gid_high = 0;
4573                :         }
4574     1430288015 :         raw_inode->i_links_count = cpu_to_le16(inode->i_nlink);
4575                :
4576      -13267358 :         EXT4_INODE_SET_XTIME(i_ctime, inode, raw_inode);
4577       -6974214 :         EXT4_INODE_SET_XTIME(i_mtime, inode, raw_inode);
4578              -1 :         EXT4_INODE_SET_XTIME(i_atime, inode, raw_inode);
4579    -1408657402 :         EXT4_EINODE_SET_XTIME(i_crtime, ei, raw_inode);
4580                :
4581     1430288015 :         if (ext4_inode_blocks_set(handle, raw_inode, ei))
4582              0 :                 goto out_brelse;
4583     1430288015 :         raw_inode->i_dtime = cpu_to_le32(ei->i_dtime);
4584                :         /* clear the migrate flag in the raw_inode */
4585     1430288015 :         raw_inode->i_flags = cpu_to_le32(ei->i_flags & ~EXT4_EXT_MIGRATE);
4586    -1434391266 :         if (EXT4_SB(inode->i_sb)->s_es->s_creator_os !=
4587                :             cpu_to_le32(EXT4_OS_HURD))
4588     1436103122 :                 raw_inode->i_file_acl_high =
4589                :                         cpu_to_le16(ei->i_file_acl >> 32);
4590     1430288015 :         raw_inode->i_file_acl_lo = cpu_to_le32(ei->i_file_acl);
4591     1430288015 :         ext4_isize_set(raw_inode, ei->i_disksize);
4592     1430288015 :         if (ei->i_disksize > 0x7fffffffULL) {
4593      617137641 :                 struct super_block *sb = inode->i_sb;
4594     1234275328 :                 if (!EXT4_HAS_RO_COMPAT_FEATURE(sb,
4595                :                                 EXT4_FEATURE_RO_COMPAT_LARGE_FILE) ||
4596                :                                 EXT4_SB(sb)->s_es->s_rev_level ==
4597                :                                 cpu_to_le32(EXT4_GOOD_OLD_REV)) {
4598                :                         /* If this is the first large file
4599                :                          * created, add a flag to the superblock.
4600                :                          */
4601       10349529 :                         err = ext4_journal_get_write_access(handle,
4602                :                                         EXT4_SB(sb)->s_sbh);
```

```
4603                0 :                         if (err)
4604                0 :                                 goto out_brelse;
4605                0 :                         ext4_update_dynamic_rev(sb);
4606                0 :                         EXT4_SET_RO_COMPAT_FEATURE(sb,
4607                  :                                 EXT4_FEATURE_RO_COMPAT_LARGE_FILE);
4608                0 :                         sb->s_dirt = 1;
4609                  :                         ext4_handle_sync(handle);
4610                0 :                         err = ext4_handle_dirty_metadata(handle, inode,
4611                  :                                 EXT4_SB(sb)->s_sbh);
4612                  :                 }
4613                  :         }
4614       1419938486 :         raw_inode->i_generation = cpu_to_le32(inode->i_generation);
4615       1419938486 :         if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode)) {
4616                0 :                 if (old_valid_dev(inode->i_rdev)) {
4617                0 :                         raw_inode->i_block[0] =
4618                  :                                 cpu_to_le32(old_encode_dev(inode->i_rdev));
4619                0 :                         raw_inode->i_block[1] = 0;
4620                  :                 } else {
4621                0 :                         raw_inode->i_block[0] = 0;
4622                0 :                         raw_inode->i_block[1] =
4623                  :                                 cpu_to_le32(new_encode_dev(inode->i_rdev));
4624                0 :                         raw_inode->i_block[2] = 0;
4625                  :                 }
4626                  :         } else
4627               -1 :                 for (block = 0; block < EXT4_N_BLOCKS; block++)
4628               -1 :                         raw_inode->i_block[block] = ei->i_data[block];
4629                  :
4630       1419938486 :         raw_inode->i_disk_version = cpu_to_le32(inode->i_version);
4631       1419938486 :         if (ei->i_extra_isize) {
4632       1456804115 :                 if (EXT4_FITS_IN_INODE(raw_inode, ei, i_version_hi))
4633       1457898887 :                         raw_inode->i_version_hi =
4634                  :                         cpu_to_le32(inode->i_version >> 32);
4635       1456804115 :                 raw_inode->i_extra_isize = cpu_to_le16(ei->i_extra_isize);
4636                  :         }
4637                  :
4638                  :         BUFFER_TRACE(bh, "call ext4_handle_dirty_metadata");
4639       1419938486 :         rc = ext4_handle_dirty_metadata(handle, inode, bh);
4640       1451755825 :         if (!err)
4641       1452034583 :                 err = rc;
4642       1451755825 :         ei->i_state &= ~EXT4_STATE_NEW;
4643                  :
4644       1451755825 : out_brelse:
4645       1451755825 :         brelse(bh);
4646       1454277299 :         ext4_std_error(inode->i_sb, err);
4647       1454277299 :         return err;
4648                  : }
4649                  :
4650                  : /*
4651                  :  * ext4_write_inode()
4652                  :  *
4653                  :  * We are called from a few places:
4654                  :  *
4655                  :  * - Within generic_file_write() for O_SYNC files.
4656                  :  *   Here, there will be no transaction running. We wait for any running
4657                  :  *   trasnaction to commit.
4658                  :  *
4659                  :  * - Within sys_sync(), kupdate and such.
4660                  :  *   We wait on commit, if tol to.
4661                  :  *
4662                  :  * - Within prune_icache() (PF_MEMALLOC == true)
4663                  :  *   Here we simply return.  We can't afford to block kswapd on the
4664                  :  *   journal commit.
4665                  :  *
4666                  :  * In all cases it is actually safe for us to return without doing anything,
4667                  :  * because the inode has been copied into a raw inode buffer in
4668                  :  * ext4_mark_inode_dirty().  This is a correctness thing for O_SYNC and for
4669                  :  * knfsd.
4670                  :  *
4671                  :  * Note that we are absolutely dependent upon all inode dirtiers doing the
4672                  :  * right thing: they *must* call mark_inode_dirty() after dirtying info in
4673                  :  * which we are interested.
```

```
4674                 :  *
4675                 :  * It would be a bug for them to not do this.  The code:
4676                 :  *
4677                 :  *      mark_inode_dirty(inode)
4678                 :  *      stuff();
4679                 :  *      inode->i_size = expr;
4680                 :  *
4681                 :  * is in error because a kswapd-driven write_inode() could occur while
4682                 :  * `stuff()' is running, and the new i_size will be lost.  Plus the inode
4683                 :  * will no longer be on the superblock's dirty inode list.
4684                 :  */
4685                 : int ext4_write_inode(struct inode *inode, int wait)
4686          33760955 : {
4687          33760955 :         if (current->flags & PF_MEMALLOC)
4688                 0 :                 return 0;
4689                 :
4690          33760955 :         if (ext4_journal_current_handle()) {
4691                 0 :                 jbd_debug(1, "called recursively, non-PF_MEMALLOC!\n");
4692                 0 :                 dump_stack();
4693                 0 :                 return -EIO;
4694                 :         }
4695                 :
4696          33760955 :         if (!wait)
4697          33747469 :                 return 0;
4698                 :
4699             13486 :         return ext4_force_commit(inode->i_sb);
4700                 : }
4701                 :
4702                 : /*
4703                 :  * ext4_setattr()
4704                 :  *
4705                 :  * Called from notify_change.
4706                 :  *
4707                 :  * We want to trap VFS attempts to truncate the file as soon as
4708                 :  * possible.  In particular, we want to make sure that when the VFS
4709                 :  * shrinks i_size, we put the inode on the orphan list and modify
4710                 :  * i_disksize immediately, so that during the subsequent flushing of
4711                 :  * dirty pages and freeing of disk blocks, we can guarantee that any
4712                 :  * commit will leave the blocks being flushed in an unused state on
4713                 :  * disk.  (On recovery, the inode will get truncated and the blocks will
4714                 :  * be freed, so we have a strong guarantee that no future commit will
4715                 :  * leave these blocks visible to the user.)
4716                 :  *
4717                 :  * Another thing we have to assure is that if we are in ordered mode
4718                 :  * and inode is still attached to the committing transaction, we must
4719                 :  * we start writeout of all the dirty pages which are being truncated.
4720                 :  * This way we are sure that all the data written in the previous
4721                 :  * transaction are already on disk (truncate waits for pages under
4722                 :  * writeback).
4723                 :  *
4724                 :  * Called with inode->i_mutex down.
4725                 :  */
4726                 : int ext4_setattr(struct dentry *dentry, struct iattr *attr)
4727                 6 : {
4728                 6 :         struct inode *inode = dentry->d_inode;
4729                 6 :         int error, rc = 0;
4730                 6 :         const unsigned int ia_valid = attr->ia_valid;
4731                 :
4732                 6 :         error = inode_change_ok(inode, attr);
4733                 6 :         if (error)
4734                 0 :                 return error;
4735                 :
4736                 6 :         if ((ia_valid & ATTR_UID && attr->ia_uid != inode->i_uid) ||
4737                 :                 (ia_valid & ATTR_GID && attr->ia_gid != inode->i_gid)) {
4738                 :                 handle_t *handle;
4739                 :
4740                 :                 /* (user+group)*(old+new) structure, inode write (sb,
4741                 :                  * inode block, ? - but truncate inode update has it) */
4742                 4 :                 handle = ext4_journal_start(inode, 2*(EXT4_QUOTA_INIT_BLOCKS(inode->i_sb)+
4743                 0 :                                                     EXT4_QUOTA_DEL_BLOCKS(inode->i_sb))+3);
4744                 1 :                 if (IS_ERR(handle)) {
```

```
4745           0 :                         error = PTR_ERR(handle);
4746           0 :                         goto err_out;
4747             :                 }
4748           1 :                 error = vfs_dq_transfer(inode, attr) ? -EDQUOT : 0;
4749           1 :                 if (error) {
4750           0 :                         ext4_journal_stop(handle);
4751           0 :                         return error;
4752             :                 }
4753             :                 /* Update corresponding info in inode so that everything is in
4754             :                  * one transaction */
4755           1 :                 if (attr->ia_valid & ATTR_UID)
4756           1 :                         inode->i_uid = attr->ia_uid;
4757           1 :                 if (attr->ia_valid & ATTR_GID)
4758           1 :                         inode->i_gid = attr->ia_gid;
4759           1 :                 error = ext4_mark_inode_dirty(handle, inode);
4760           1 :                 ext4_journal_stop(handle);
4761             :         }
4762             :
4763           6 :         if (attr->ia_valid & ATTR_SIZE) {
4764           0 :                 if (!(EXT4_I(inode)->i_flags & EXT4_EXTENTS_FL)) {
4765           0 :                         struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);
4766             :
4767           0 :                         if (attr->ia_size > sbi->s_bitmap_maxbytes) {
4768           0 :                                 error = -EFBIG;
4769           0 :                                 goto err_out;
4770             :                         }
4771             :                 }
4772             :         }
4773             :
4774           6 :         if (S_ISREG(inode->i_mode) &&
4775             :             attr->ia_valid & ATTR_SIZE && attr->ia_size < inode->i_size) {
4776             :                 handle_t *handle;
4777             :
4778           0 :                 handle = ext4_journal_start(inode, 3);
4779           0 :                 if (IS_ERR(handle)) {
4780           0 :                         error = PTR_ERR(handle);
4781           0 :                         goto err_out;
4782             :                 }
4783             :
4784           0 :                 error = ext4_orphan_add(handle, inode);
4785           0 :                 EXT4_I(inode)->i_disksize = attr->ia_size;
4786           0 :                 rc = ext4_mark_inode_dirty(handle, inode);
4787           0 :                 if (!error)
4788           0 :                         error = rc;
4789           0 :                 ext4_journal_stop(handle);
4790             :
4791           0 :                 if (ext4_should_order_data(inode)) {
4792           0 :                         error = ext4_begin_ordered_truncate(inode,
4793             :                                                             attr->ia_size);
4794           0 :                         if (error) {
4795             :                                 /* Do as much error cleanup as possible */
4796           0 :                                 handle = ext4_journal_start(inode, 3);
4797           0 :                                 if (IS_ERR(handle)) {
4798           0 :                                         ext4_orphan_del(NULL, inode);
4799           0 :                                         goto err_out;
4800             :                                 }
4801           0 :                                 ext4_orphan_del(handle, inode);
4802           0 :                                 ext4_journal_stop(handle);
4803           0 :                                 goto err_out;
4804             :                         }
4805             :                 }
4806             :         }
4807             :
4808           6 :         rc = inode_setattr(inode, attr);
4809             :
4810             :         /* If inode_setattr's call to ext4_truncate failed to get a
4811             :          * transaction handle at all, we need to clean up the in-core
4812             :          * orphan list manually. */
4813           6 :         if (inode->i_nlink)
4814           6 :                 ext4_orphan_del(NULL, inode);
4815             :
```

```
4816         6 :         if (!rc && (ia_valid & ATTR_MODE))
4817         1 :                 rc = ext4_acl_chmod(inode);
4818           :
4819         6 : err_out:
4820         6 :         ext4_std_error(inode->i_sb, error);
4821         6 :         if (!error)
4822         6 :                 error = rc;
4823         6 :         return error;
4824           : }
4825           :
4826           : int ext4_getattr(struct vfsmount *mnt, struct dentry *dentry,
4827           :                  struct kstat *stat)
4828  29544061 : {
4829           :         struct inode *inode;
4830           :         unsigned long delalloc_blocks;
4831           :
4832  29544061 :         inode = dentry->d_inode;
4833  29544061 :         generic_fillattr(inode, stat);
4834           :
4835           :         /*
4836           :          * We can't update i_blocks if the block allocation is delayed
4837           :          * otherwise in the case of system crash before the real block
4838           :          * allocation is done, we will have i_blocks inconsistent with
4839           :          * on-disk file blocks.
4840           :          * We always keep i_blocks updated together with real
4841           :          * allocation. But to not confuse with user, stat
4842           :          * will return the blocks that include the delayed allocation
4843           :          * blocks for this file.
4844           :          */
4845  29510139 :         spin_lock(&EXT4_I(inode)->i_block_reservation_lock);
4846  29691207 :         delalloc_blocks = EXT4_I(inode)->i_reserved_data_blocks;
4847  29691207 :         spin_unlock(&EXT4_I(inode)->i_block_reservation_lock);
4848           :
4849  29970117 :         stat->blocks += (delalloc_blocks << inode->i_sb->s_blocksize_bits)>>9;
4850  29970117 :         return 0;
4851           : }
4852           :
4853           : static int ext4_indirect_trans_blocks(struct inode *inode, int nrblocks,
4854           :                                       int chunk)
4855           : {
4856           :         int indirects;
4857           :
4858           :         /* if nrblocks are contiguous */
4859   1334365 :         if (chunk) {
4860           :                 /*
4861           :                  * With N contiguous data blocks, it need at most
4862           :                  * N/EXT4_ADDR_PER_BLOCK(inode->i_sb) indirect blocks
4863           :                  * 2 dindirect blocks
4864           :                  * 1 tindirect block
4865           :                  */
4866   1334365 :                 indirects = nrblocks / EXT4_ADDR_PER_BLOCK(inode->i_sb);
4867   1334365 :                 return indirects + 3;
4868           :         }
4869           :         /*
4870           :          * if nrblocks are not contiguous, worse case, each block touch
4871           :          * a indirect block, and each indirect block touch a double indirect
4872           :          * block, plus a triple indirect block
4873           :          */
4874         0 :         indirects = nrblocks * 2 + 1;
4875         0 :         return indirects;
4876           : }
4877           :
4878           : static int ext4_index_trans_blocks(struct inode *inode, int nrblocks, int chunk)
4879           : {
4880 247625594 :         if (!(EXT4_I(inode)->i_flags & EXT4_EXTENTS_FL))
4881   1334365 :                 return ext4_indirect_trans_blocks(inode, nrblocks, chunk);
4882 246291229 :         return ext4_ext_index_trans_blocks(inode, nrblocks, chunk);
4883           : }
4884           :
4885           : /*
4886           :  * Account for index blocks, block groups bitmaps and block group
```

```
4887                :  * descriptor blocks if modify datablocks and index blocks
4888                :  * worse case, the indexs blocks spread over different block groups
4889                :  *
4890                :  * If datablocks are discontiguous, they are possible to spread over
4891                :  * different block groups too. If they are contiugous, with flexbg,
4892                :  * they could still across block group boundary.
4893                :  *
4894                :  * Also account for superblock, inode, quota and xattr blocks
4895                :  */
4896                : int ext4_meta_trans_blocks(struct inode *inode, int nrblocks, int chunk)
4897      245657740 : {
4898      493283334 :         ext4_group_t groups, ngroups = ext4_get_groups_count(inode->i_sb);
4899                :         int gdpblocks;
4900                :         int idxblocks;
4901      247625594 :         int ret = 0;
4902                :
4903                :         /*
4904                :          * How many index blocks need to touch to modify nrblocks?
4905                :          * The "Chunk" flag indicating whether the nrblocks is
4906                :          * physically contiguous on disk
4907                :          *
4908                :          * For Direct IO and fallocate, they calls get_block to allocate
4909                :          * one single extent at a time, so they could set the "Chunk" flag
4910                :          */
4911      247271585 :         idxblocks = ext4_index_trans_blocks(inode, nrblocks, chunk);
4912                :
4913      247271585 :         ret = idxblocks;
4914                :
4915                :         /*
4916                :          * Now let's see how many group bitmaps and group descriptors need
4917                :          * to account
4918                :          */
4919      247271585 :         groups = idxblocks;
4920      247271585 :         if (chunk)
4921       35971071 :                 groups += 1;
4922                :         else
4923      211300514 :                 groups += nrblocks;
4924                :
4925      247271585 :         gdpblocks = groups;
4926      247271585 :         if (groups > ngroups)
4927              0 :                 groups = ngroups;
4928      494543170 :         if (groups > EXT4_SB(inode->i_sb)->s_gdb_count)
4929              0 :                 gdpblocks = EXT4_SB(inode->i_sb)->s_gdb_count;
4930                :
4931                :         /* bitmaps and block group descriptor blocks */
4932      247271585 :         ret += groups + gdpblocks;
4933                :
4934                :         /* Blocks for super block, inode, quota and xattr blocks */
4935      494543170 :         ret += EXT4_META_TRANS_BLOCKS(inode->i_sb);
4936                :
4937      247271585 :         return ret;
4938                : }
4939                :
4940                : /*
4941                :  * Calulate the total number of credits to reserve to fit
4942                :  * the modification of a single pages into a single transaction,
4943                :  * which may include multiple chunks of block allocations.
4944                :  *
4945                :  * This could be called via ext4_write_begin()
4946                :  *
4947                :  * We need to consider the worse case, when
4948                :  * one new block per extent.
4949                :  */
4950                : int ext4_writepage_trans_blocks(struct inode *inode)
4951      209925778 : {
4952      211559671 :         int bpp = ext4_journal_blocks_per_page(inode);
4953                :         int ret;
4954                :
4955      211559671 :         ret = ext4_meta_trans_blocks(inode, bpp, 0);
4956                :
4957                :         /* Account for data blocks for journalled mode */
```

```
4958  209957827 :         if (ext4_should_journal_data(inode))
4959    9213171 :                 ret += bpp;
4960  209957827 :         return ret;
4961            : }
4962            :
4963            : /*
4964            :  * Calculate the journal credits for a chunk of data modification.
4965            :  *
4966            :  * This is called from DIO, fallocate or whoever calling
4967            :  * ext4_get_blocks() to map/allocate a chunk of contigous disk blocks.
4968            :  *
4969            :  * journal buffers for data blocks are not included here, as DIO
4970            :  * and fallocate do no need to journal data buffers.
4971            :  */
4972            : int ext4_chunk_trans_blocks(struct inode *inode, int nrblocks)
4973   35260337 : {
4974   35260337 :         return ext4_meta_trans_blocks(inode, nrblocks, 1);
4975            : }
4976            :
4977            : /*
4978            :  * The caller must have previously called ext4_reserve_inode_write().
4979            :  * Give this, we know that the caller already has write access to iloc->bh.
4980            :  */
4981            : int ext4_mark_iloc_dirty(handle_t *handle,
4982            :                          struct inode *inode, struct ext4_iloc *iloc)
4983 1423275500 : {
4984 1423275500 :         int err = 0;
4985            :
4986 -1448416296 :         if (test_opt(inode->i_sb, I_VERSION))
4987            :                 inode_inc_iversion(inode);
4988            :
4989            :         /* the do_update_inode consumes one bh->b_count */
4990 1423275500 :         get_bh(iloc->bh);
4991            :
4992            :         /* ext4_do_update_inode() does jbd2_journal_dirty_metadata */
4993 1455056032 :         err = ext4_do_update_inode(handle, inode, iloc);
4994 1451138885 :         put_bh(iloc->bh);
4995 1451506016 :         return err;
4996            : }
4997            :
4998            : /*
4999            :  * On success, We end up with an outstanding reference count against
5000            :  * iloc->bh.  This _must_ be cleaned up later.
5001            :  */
5002            :
5003            : int
5004            : ext4_reserve_inode_write(handle_t *handle, struct inode *inode,
5005            :                          struct ext4_iloc *iloc)
5006 1427474376 : {
5007            :         int err;
5008            :
5009 1427474376 :         err = ext4_get_inode_loc(inode, iloc);
5010 1454370814 :         if (!err) {
5011            :                 BUFFER_TRACE(iloc->bh, "get_write_access");
5012 1455937966 :                 err = ext4_journal_get_write_access(handle, iloc->bh);
5013 1443440126 :                 if (err) {
5014          0 :                         brelse(iloc->bh);
5015          0 :                         iloc->bh = NULL;
5016            :                 }
5017            :         }
5018 1441872974 :         ext4_std_error(inode->i_sb, err);
5019 1441872974 :         return err;
5020            : }
5021            :
5022            : /*
5023            :  * Expand an inode by new_extra_isize bytes.
5024            :  * Returns 0 on success or negative error number on failure.
5025            :  */
5026            : static int ext4_expand_extra_isize(struct inode *inode,
5027            :                                    unsigned int new_extra_isize,
5028            :                                    struct ext4_iloc iloc,
```

```
5029                :                                       handle_t *handle)
5030              0 : {
5031                :         struct ext4_inode *raw_inode;
5032                :         struct ext4_xattr_ibody_header *header;
5033                :         struct ext4_xattr_entry *entry;
5034                :
5035              0 :         if (EXT4_I(inode)->i_extra_isize >= new_extra_isize)
5036              0 :                 return 0;
5037                :
5038              0 :         raw_inode = ext4_raw_inode(&iloc);
5039                :
5040              0 :         header = IHDR(inode, raw_inode);
5041              0 :         entry = IFIRST(header);
5042                :
5043                :         /* No extended attributes present */
5044              0 :         if (!(EXT4_I(inode)->i_state & EXT4_STATE_XATTR) ||
5045                :                 header->h_magic != cpu_to_le32(EXT4_XATTR_MAGIC)) {
5046              0 :                 memset((void *)raw_inode + EXT4_GOOD_OLD_INODE_SIZE, 0,
5047                :                         new_extra_isize);
5048              0 :                 EXT4_I(inode)->i_extra_isize = new_extra_isize;
5049              0 :                 return 0;
5050                :         }
5051                :
5052                :         /* try to expand with EAs present */
5053              0 :         return ext4_expand_extra_isize_ea(inode, new_extra_isize,
5054                :                                           raw_inode, handle);
5055                : }
5056                :
5057                : /*
5058                :  * What we do here is to mark the in-core inode as clean with respect to inode
5059                :  * dirtiness (it may still be data-dirty).
5060                :  * This means that the in-core inode may be reaped by prune_icache
5061                :  * without having to perform any I/O.  This is a very good thing,
5062                :  * because *any* task may call prune_icache - even ones which
5063                :  * have a transaction open against a different journal.
5064                :  *
5065                :  * Is this cheating?  Not really.  Sure, we haven't written the
5066                :  * inode out, but prune_icache isn't a user-visible syncing function.
5067                :  * Whenever the user wants stuff synced (sys_sync, sys_msync, sys_fsync)
5068                :  * we start and wait on commits.
5069                :  *
5070                :  * Is this efficient/effective?  Well, we're being nice to the system
5071                :  * by cleaning up our inodes proactively so they can be reaped
5072                :  * without I/O.  But we are potentially leaving up to five seconds'
5073                :  * worth of inodes floating about which prune_icache wants us to
5074                :  * write out.  One way to fix that would be to get prune_icache()
5075                :  * to do a write_super() to free up some memory.  It has the desired
5076                :  * effect.
5077                :  */
5078                : int ext4_mark_inode_dirty(handle_t *handle, struct inode *inode)
5079     1423256650 : {
5080                :         struct ext4_iloc iloc;
5081    -1448453996 :         struct ext4_sb_info *sbi = EXT4_SB(inode->i_sb);
5082                :         static unsigned int mnt_count;
5083                :         int err, ret;
5084                :
5085     1423256650 :         might_sleep();
5086     1433847736 :         err = ext4_reserve_inode_write(handle, inode, &iloc);
5087    -1433129019 :         if (ext4_handle_valid(handle) &&
5088                :             EXT4_I(inode)->i_extra_isize < sbi->s_want_extra_isize &&
5089                :             !(EXT4_I(inode)->i_state & EXT4_STATE_NO_EXPAND)) {
5090                :                 /*
5091                :                  * We need extra buffer credits since we may write into EA block
5092                :                  * with this same handle. If journal_extend fails, then it will
5093                :                  * only result in a minor loss of functionality for that inode.
5094                :                  * If this is felt to be critical, then e2fsck should be run to
5095                :                  * force a large enough s_min_extra_isize.
5096                :                  */
5097              0 :                 if ((jbd2_journal_extend(handle,
5098                :                                EXT4_DATA_TRANS_BLOCKS(inode->i_sb))) == 0) {
5099              0 :                         ret = ext4_expand_extra_isize(inode,
```

```
5100                :                                                       sbi->s_want_extra_isize,
5101                :                                                       iloc, handle);
5102              0 :                               if (ret) {
5103              0 :                                       EXT4_I(inode)->i_state |= EXT4_STATE_NO_EXPAND;
5104              0 :                                       if (mnt_count !=
5105                :                                               le16_to_cpu(sbi->s_es->s_mnt_count)) {
5106              0 :                                               ext4_warning(inode->i_sb, __func__,
5107                :                                               "Unable to expand inode %lu. Delete"
5108                :                                               " some EAs or run e2fsck.",
5109                :                                               inode->i_ino);
5110              0 :                                               mnt_count =
5111                :                                                 le16_to_cpu(sbi->s_es->s_mnt_count);
5112                :                                       }
5113                :                               }
5114                :                       }
5115                :               }
5116     1435527040 :               if (!err)
5117     1426723938 :                       err = ext4_mark_iloc_dirty(handle, inode, &iloc);
5118     1457203330 :               return err;
5119                : }
5120                :
5121                : /*
5122                :  * ext4_dirty_inode() is called from __mark_inode_dirty()
5123                :  *
5124                :  * We're really interested in the case where a file is being extended.
5125                :  * i_size has been changed by generic_commit_write() and we thus need
5126                :  * to include the updated inode in the current transaction.
5127                :  *
5128                :  * Also, vfs_dq_alloc_block() will always dirty the inode when blocks
5129                :  * are allocated to the file.
5130                :  *
5131                :  * If the inode is marked synchronous, we don't honour that here - doing
5132                :  * so would cause a commit on atime updates, which we don't bother doing.
5133                :  * We handle synchronous inodes at the highest possible level.
5134                :  */
5135                : void ext4_dirty_inode(struct inode *inode)
5136     1112178837 : {
5137     1112178837 :               handle_t *current_handle = ext4_journal_current_handle();
5138                :               handle_t *handle;
5139                :
5140     1112178837 :               if (!ext4_handle_valid(current_handle)) {
5141              0 :                       ext4_mark_inode_dirty(current_handle, inode);
5142              0 :                       return;
5143                :               }
5144                :
5145     1103435827 :               handle = ext4_journal_start(inode, 2);
5146     1104714831 :               if (IS_ERR(handle))
5147              0 :                       goto out;
5148     1104714831 :               if (current_handle &&
5149                :                       current_handle->h_transaction != handle->h_transaction) {
5150                :                       /* This task has a transaction open against a different fs */
5151              0 :                       printk(KERN_EMERG "%s: transactions do not match!\n",
5152                :                              __func__);
5153                :               } else {
5154     1104714831 :                       jbd_debug(5, "marking dirty.  outer handle=%p\n",
5155                :                                 current_handle);
5156     1104714831 :                       ext4_mark_inode_dirty(handle, inode);
5157                :               }
5158     1098755054 :               ext4_journal_stop(handle);
5159                : out:
5160                :               return;
5161                : }
5162                :
5163                : #if 0
5164                : /*
5165                :  * Bind an inode's backing buffer_head into this transaction, to prevent
5166                :  * it from being flushed to disk early.  Unlike
5167                :  * ext4_reserve_inode_write, this leaves behind no bh reference and
5168                :  * returns no iloc structure, so the caller needs to repeat the iloc
5169                :  * lookup to mark the inode dirty later.
5170                :  */
```

```
5171                 : static int ext4_pin_inode(handle_t *handle, struct inode *inode)
5172                 : {
5173                 :         struct ext4_iloc iloc;
5174                 :
5175                 :         int err = 0;
5176                 :         if (handle) {
5177                 :                 err = ext4_get_inode_loc(inode, &iloc);
5178                 :                 if (!err) {
5179                 :                         BUFFER_TRACE(iloc.bh, "get_write_access");
5180                 :                         err = jbd2_journal_get_write_access(handle, iloc.bh);
5181                 :                         if (!err)
5182                 :                                 err = ext4_handle_dirty_metadata(handle,
5183                 :                                                                  inode,
5184                 :                                                                  iloc.bh);
5185                 :                         brelse(iloc.bh);
5186                 :                 }
5187                 :         }
5188                 :         ext4_std_error(inode->i_sb, err);
5189                 :         return err;
5190                 : }
5191                 : #endif
5192                 :
5193                 : int ext4_change_inode_journal_flag(struct inode *inode, int val)
5194               0 : {
5195                 :         journal_t *journal;
5196                 :         handle_t *handle;
5197                 :         int err;
5198                 :
5199                 :         /*
5200                 :          * We have to be very careful here: changing a data block's
5201                 :          * journaling status dynamically is dangerous.  If we write a
5202                 :          * data block to the journal, change the status and then delete
5203                 :          * that block, we risk forgetting to revoke the old log record
5204                 :          * from the journal and so a subsequent replay can corrupt data.
5205                 :          * So, first we make sure that the journal is empty and that
5206                 :          * nobody is changing anything.
5207                 :          */
5208                 :
5209               0 :         journal = EXT4_JOURNAL(inode);
5210               0 :         if (!journal)
5211               0 :                 return 0;
5212               0 :         if (is_journal_aborted(journal))
5213               0 :                 return -EROFS;
5214                 :
5215               0 :         jbd2_journal_lock_updates(journal);
5216               0 :         jbd2_journal_flush(journal);
5217                 :
5218                 :         /*
5219                 :          * OK, there are no updates running now, and all cached data is
5220                 :          * synced to disk.  We are now in a completely consistent state
5221                 :          * which doesn't have anything in the journal, and we know that
5222                 :          * no filesystem updates are running, so it is safe to modify
5223                 :          * the inode's in-core data-journaling state flag now.
5224                 :          */
5225                 :
5226               0 :         if (val)
5227               0 :                 EXT4_I(inode)->i_flags |= EXT4_JOURNAL_DATA_FL;
5228                 :         else
5229               0 :                 EXT4_I(inode)->i_flags &= ~EXT4_JOURNAL_DATA_FL;
5230               0 :         ext4_set_aops(inode);
5231                 :
5232               0 :         jbd2_journal_unlock_updates(journal);
5233                 :
5234                 :         /* Finally we can mark the inode as dirty. */
5235                 :
5236               0 :         handle = ext4_journal_start(inode, 1);
5237               0 :         if (IS_ERR(handle))
5238               0 :                 return PTR_ERR(handle);
5239                 :
5240               0 :         err = ext4_mark_inode_dirty(handle, inode);
5241                 :         ext4_handle_sync(handle);
```

```
5242                0 :          ext4_journal_stop(handle);
5243                0 :          ext4_std_error(inode->i_sb, err);
5244                  :
5245                0 :          return err;
5246                  : }
5247                  :
5248                  : static int ext4_bh_unmapped(handle_t *handle, struct buffer_head *bh)
5249                0 : {
5250                0 :          return !buffer_mapped(bh);
5251                  : }
5252                  :
5253                  : int ext4_page_mkwrite(struct vm_area_struct *vma, struct vm_fault *vmf)
5254                0 : {
5255                0 :          struct page *page = vmf->page;
5256                  :          loff_t size;
5257                  :          unsigned long len;
5258                0 :          int ret = -EINVAL;
5259                  :          void *fsdata;
5260                0 :          struct file *file = vma->vm_file;
5261                0 :          struct inode *inode = file->f_path.dentry->d_inode;
5262                0 :          struct address_space *mapping = inode->i_mapping;
5263                  :
5264                  :          /*
5265                  :           * Get i_alloc_sem to stop truncates messing with the inode. We cannot
5266                  :           * get i_mutex because we are already holding mmap_sem.
5267                  :           */
5268                0 :          down_read(&inode->i_alloc_sem);
5269                0 :          size = i_size_read(inode);
5270                0 :          if (page->mapping != mapping || size <= page_offset(page)
5271                  :              || !PageUptodate(page)) {
5272                  :                   /* page got truncated from under us? */
5273                  :                   goto out_unlock;
5274                  :          }
5275                0 :          ret = 0;
5276                0 :          if (PageMappedToDisk(page))
5277                0 :                   goto out_unlock;
5278                  :
5279                0 :          if (page->index == size >> PAGE_CACHE_SHIFT)
5280                0 :                   len = size & ~PAGE_CACHE_MASK;
5281                  :          else
5282                0 :                   len = PAGE_CACHE_SIZE;
5283                  :
5284                0 :          if (page_has_buffers(page)) {
5285                  :                   /* return if we have all the buffers mapped */
5286                0 :                   if (!walk_page_buffers(NULL, page_buffers(page), 0, len, NULL,
5287                  :                                          ext4_bh_unmapped))
5288                0 :                           goto out_unlock;
5289                  :          }
5290                  :          /*
5291                  :           * OK, we need to fill the hole... Do write_begin write_end
5292                  :           * to do block allocation/reservation.We are not holding
5293                  :           * inode.i__mutex here. That allow * parallel write_begin,
5294                  :           * write_end call. lock_page prevent this from happening
5295                  :           * on the same page though
5296                  :           */
5297                0 :          ret = mapping->a_ops->write_begin(file, mapping, page_offset(page),
5298                  :                          len, AOP_FLAG_UNINTERRUPTIBLE, &page, &fsdata);
5299                0 :          if (ret < 0)
5300                0 :                   goto out_unlock;
5301                0 :          ret = mapping->a_ops->write_end(file, mapping, page_offset(page),
5302                  :                          len, len, page, fsdata);
5303                0 :          if (ret < 0)
5304                0 :                   goto out_unlock;
5305                0 :          ret = 0;
5306                0 : out_unlock:
5307                0 :          if (ret)
5308                0 :                   ret = VM_FAULT_SIGBUS;
5309                0 :          up_read(&inode->i_alloc_sem);
5310                0 :          return ret;
5311                  : }
```