LCOV - code coverage report

			Found	Hit	Covera
Current view:	directory - fs/ext4 - mballoc.c (source / functions)				
Test:	kernel_2_6_31_ext4_round_3.info	**Lines:**	1973	1527	77.4
Date:	2009-10-24	**Functions:**	66	48	72.7

```
                :  /*
                :   * Copyright (c) 2003-2006, Cluster File Systems, Inc, info@clusterfs.com
                :   * Written by Alex Tomas <alex@clusterfs.com>
                :   *
                :   * This program is free software; you can redistribute it and/or modify
                :   * it under the terms of the GNU General Public License version 2 as
                :   * published by the Free Software Foundation.
                :   *
                :   * This program is distributed in the hope that it will be useful,
                :   * but WITHOUT ANY WARRANTY; without even the implied warranty of
                :   * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
                :   * GNU General Public License for more details.
                :   *
                :   * You should have received a copy of the GNU General Public Licens
                :   * along with this program; if not, write to the Free Software
                :   * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-
                :   */
                :
                :
                :  /*
                :   * mballoc.c contains the multiblocks allocation routines
                :   */
                :
                :  #include "mballoc.h"
                :  #include <trace/events/ext4.h>
                :
                :  /*
                :   * MUSTDO:
                :   *   - test ext4_ext_search_left() and ext4_ext_search_right()
                :   *   - search for metadata in few groups
                :   *
                :   * TODO v4:
                :   *   - normalization should take into account whether file is still open
                :   *   - discard preallocations if no free space left (policy?)
                :   *   - don't normalize tails
                :   *   - quota
                :   *   - reservation for superuser
                :   *
                :   * TODO v3:
                :   *   - bitmap read-ahead (proposed by Oleg Drokin aka green)
                :   *   - track min/max extents in each group for better group selection
                :   *   - mb_mark_used() may allocate chunk right after splitting buddy
                :   *   - tree of groups sorted by number of free blocks
                :   *   - error handling
                :   */
                :
                :  /*
                :   * The allocation request involve request for multiple number of blocks
                :   * near to the goal(block) value specified.
                :   *
                :   * During initialization phase of the allocator we decide to use the
                :   * group preallocation or inode preallocation depending on the size of
                :   * the file. The size of the file could be the resulting file size we
                :   * would have after allocation, or the current file size, which ever
                :   * is larger. If the size is less than sbi->s_mb_stream_request we
                :   * select to use the group preallocation. The default value of
                :   * s_mb_stream_request is 16 blocks. This can also be tuned via
                :   * /sys/fs/ext4/<partition>/mb_stream_req. The value is represented in
```

```
 59                 :  * terms of number of blocks.
 60                 :  *
 61                 :  * The main motivation for having small file use group preallocation is to
 62                 :  * ensure that we have small files closer together on the disk.
 63                 :  *
 64                 :  * First stage the allocator looks at the inode prealloc list,
 65                 :  * ext4_inode_info->i_prealloc_list, which contains list of prealloc
 66                 :  * spaces for this particular inode. The inode prealloc space is
 67                 :  * represented as:
 68                 :  *
 69                 :  * pa_lstart -> the logical start block for this prealloc space
 70                 :  * pa_pstart -> the physical start block for this prealloc space
 71                 :  * pa_len    -> lenght for this prealloc space
 72                 :  * pa_free   ->  free space available in this prealloc space
 73                 :  *
 74                 :  * The inode preallocation space is used looking at the _logical_ start
 75                 :  * block. If only the logical file block falls within the range of prealloc
 76                 :  * space we will consume the particular prealloc space. This make sure that
 77                 :  * that the we have contiguous physical blocks representing the file blocks
 78                 :  *
 79                 :  * The important thing to be noted in case of inode prealloc space is that
 80                 :  * we don't modify the values associated to inode prealloc space except
 81                 :  * pa_free.
 82                 :  *
 83                 :  * If we are not able to find blocks in the inode prealloc space and if we
 84                 :  * have the group allocation flag set then we look at the locality group
 85                 :  * prealloc space. These are per CPU prealloc list repreasented as
 86                 :  *
 87                 :  * ext4_sb_info.s_locality_groups[smp_processor_id()]
 88                 :  *
 89                 :  * The reason for having a per cpu locality group is to reduce the contention
 90                 :  * between CPUs. It is possible to get scheduled at this point.
 91                 :  *
 92                 :  * The locality group prealloc space is used looking at whether we have
 93                 :  * enough free space (pa_free) withing the prealloc space.
 94                 :  *
 95                 :  * If we can't allocate blocks via inode prealloc or/and locality group
 96                 :  * prealloc then we look at the buddy cache. The buddy cache is represented
 97                 :  * by ext4_sb_info.s_buddy_cache (struct inode) whose file offset gets
 98                 :  * mapped to the buddy and bitmap information regarding different
 99                 :  * groups. The buddy information is attached to buddy cache inode so that
100                 :  * we can access them through the page cache. The information regarding
101                 :  * each group is loaded via ext4_mb_load_buddy.  The information involve
102                 :  * block bitmap and buddy information. The information are stored in the
103                 :  * inode as:
104                 :  *
105                 :  *  {                        page                        }
106                 :  *  [ group 0 bitmap][ group 0 buddy] [group 1][ group 1]...
107                 :  *
108                 :  *
109                 :  * one block each for bitmap and buddy information.  So for each group we
110                 :  * take up 2 blocks. A page can contain blocks_per_page (PAGE_CACHE_SIZE /
111                 :  * blocksize) blocks.  So it can have information regarding groups_per_page
112                 :  * which is blocks_per_page/2
113                 :  *
114                 :  * The buddy cache inode is not stored on disk. The inode is thrown
115                 :  * away when the filesystem is unmounted.
116                 :  *
117                 :  * We look for count number of blocks in the buddy cache. If we were able
118                 :  * to locate that many free blocks we return with additional information
119                 :  * regarding rest of the contiguous physical block available
120                 :  *
121                 :  * Before allocating blocks via buddy cache we normalize the request
122                 :  * blocks. This ensure we ask for more blocks that we needed. The extra
123                 :  * blocks that we get after allocation is added to the respective prealloc
124                 :  * list. In case of inode preallocation we follow a list of heuristics
125                 :  * based on file size. This can be found in ext4_mb_normalize_request. If
126                 :  * we are doing a group prealloc we try to normalize the request to
127                 :  * sbi->s_mb_group_prealloc. Default value of s_mb_group_prealloc is
128                 :  * 512 blocks. This can be tuned via
129                 :  * /sys/fs/ext4/<partition/mb_group_prealloc. The value is represented in
```

```
130                 :  * terms of number of blocks. If we have mounted the file system with -O
131                 :  * stripe=<value> option the group prealloc request is normalized to the
132                 :  * stripe value (sbi->s_stripe)
133                 :  *
134                 :  * The regular allocator(using the buddy cache) supports few tunables.
135                 :  *
136                 :  * /sys/fs/ext4/<partition>/mb_min_to_scan
137                 :  * /sys/fs/ext4/<partition>/mb_max_to_scan
138                 :  * /sys/fs/ext4/<partition>/mb_order2_req
139                 :  *
140                 :  * The regular allocator uses buddy scan only if the request len is power of
141                 :  * 2 blocks and the order of allocation is >= sbi->s_mb_order2_reqs. The
142                 :  * value of s_mb_order2_reqs can be tuned via
143                 :  * /sys/fs/ext4/<partition>/mb_order2_req.  If the request len is equal to
144                 :  * stripe size (sbi->s_stripe), we try to search for contigous block in
145                 :  * stripe size. This should result in better allocation on RAID setups. If
146                 :  * not, we search in the specific group using bitmap for best extents. The
147                 :  * tunable min_to_scan and max_to_scan control the behaviour here.
148                 :  * min_to_scan indicate how long the mballoc __must__ look for a best
149                 :  * extent and max_to_scan indicates how long the mballoc __can__ look for a
150                 :  * best extent in the found extents. Searching for the blocks starts with
151                 :  * the group specified as the goal value in allocation context via
152                 :  * ac_g_ex. Each group is first checked based on the criteria whether it
153                 :  * can used for allocation. ext4_mb_good_group explains how the groups are
154                 :  * checked.
155                 :  *
156                 :  * Both the prealloc space are getting populated as above. So for the first
157                 :  * request we will hit the buddy cache which will result in this prealloc
158                 :  * space getting filled. The prealloc space is then later used for the
159                 :  * subsequent request.
160                 :  */
161                 :
162                 : /*
163                 :  * mballoc operates on the following data:
164                 :  *  - on-disk bitmap
165                 :  *  - in-core buddy (actually includes buddy and bitmap)
166                 :  *  - preallocation descriptors (PAs)
167                 :  *
168                 :  * there are two types of preallocations:
169                 :  *  - inode
170                 :  *    assiged to specific inode and can be used for this inode only.
171                 :  *    it describes part of inode's space preallocated to specific
172                 :  *    physical blocks. any block from that preallocated can be used
173                 :  *    independent. the descriptor just tracks number of blocks left
174                 :  *    unused. so, before taking some block from descriptor, one must
175                 :  *    make sure corresponded logical block isn't allocated yet. this
176                 :  *    also means that freeing any block within descriptor's range
177                 :  *    must discard all preallocated blocks.
178                 :  *  - locality group
179                 :  *    assigned to specific locality group which does not translate to
180                 :  *    permanent set of inodes: inode can join and leave group. space
181                 :  *    from this type of preallocation can be used for any inode. thus
182                 :  *    it's consumed from the beginning to the end.
183                 :  *
184                 :  * relation between them can be expressed as:
185                 :  *    in-core buddy = on-disk bitmap + preallocation descriptors
186                 :  *
187                 :  * this mean blocks mballoc considers used are:
188                 :  *  - allocated blocks (persistent)
189                 :  *  - preallocated blocks (non-persistent)
190                 :  *
191                 :  * consistency in mballoc world means that at any time a block is either
192                 :  * free or used in ALL structures. notice: "any time" should not be read
193                 :  * literally -- time is discrete and delimited by locks.
194                 :  *
195                 :  *  to keep it simple, we don't use block numbers, instead we count number of
196                 :  *  blocks: how many blocks marked used/free in on-disk bitmap, buddy and PA.
197                 :  *
198                 :  * all operations can be expressed as:
199                 :  *  - init buddy:                       buddy = on-disk + PAs
200                 :  *  - new PA:                           buddy += N; PA = N
```

```
 *  - use inode PA:                   on-disk += N; PA -= N
 *  - discard inode PA                buddy -= on-disk - PA; PA = 0
 *  - use locality group PA           on-disk += N; PA -= N
 *  - discard locality group PA       buddy -= PA; PA = 0
 *  note: 'buddy -= on-disk - PA' is used to show that on-disk bitmap
 *        is used in real operation because we can't know actual used
 *        bits from PA, only from on-disk bitmap
 *
 * if we follow this strict logic, then all operations above should be atomic.
 * given some of them can block, we'd have to use something like semaphores
 * killing performance on high-end SMP hardware. let's try to relax it using
 * the following knowledge:
 *  1) if buddy is referenced, it's already initialized
 *  2) while block is used in buddy and the buddy is referenced,
 *     nobody can re-allocate that block
 *  3) we work on bitmaps and '+' actually means 'set bits'. if on-disk has
 *     bit set and PA claims same block, it's OK. IOW, one can set bit in
 *     on-disk bitmap if buddy has same bit set or/and PA covers corresponded
 *     block
 *
 * so, now we're building a concurrency table:
 *  - init buddy vs.
 *    - new PA
 *      blocks for PA are allocated in the buddy, buddy must be referenced
 *      until PA is linked to allocation group to avoid concurrent buddy init
 *    - use inode PA
 *      we need to make sure that either on-disk bitmap or PA has uptodate data
 *      given (3) we care that PA-=N operation doesn't interfere with init
 *    - discard inode PA
 *      the simplest way would be to have buddy initialized by the discard
 *    - use locality group PA
 *      again PA-=N must be serialized with init
 *    - discard locality group PA
 *      the simplest way would be to have buddy initialized by the discard
 *  - new PA vs.
 *    - use inode PA
 *      i_data_sem serializes them
 *    - discard inode PA
 *      discard process must wait until PA isn't used by another process
 *    - use locality group PA
 *      some mutex should serialize them
 *    - discard locality group PA
 *      discard process must wait until PA isn't used by another process
 *  - use inode PA
 *    - use inode PA
 *      i_data_sem or another mutex should serializes them
 *    - discard inode PA
 *      discard process must wait until PA isn't used by another process
 *    - use locality group PA
 *      nothing wrong here -- they're different PAs covering different blocks
 *    - discard locality group PA
 *      discard process must wait until PA isn't used by another process
 *
 * now we're ready to make few consequences:
 *  - PA is referenced and while it is no discard is possible
 *  - PA is referenced until block isn't marked in on-disk bitmap
 *  - PA changes only after on-disk bitmap
 *  - discard must not compete with init. either init is done before
 *    any discard or they're serialized somehow
 *  - buddy init as sum of on-disk bitmap and PAs is done atomically
 *
 * a special case when we've used PA to emptiness. no need to modify buddy
 * in this case, but we should care about concurrent init
 *
 */

/*
 * Logic in few words:
 *
 *  - allocation:
 *    load group
```

```
272                 :  *    find blocks
273                 :  *    mark bits in on-disk bitmap
274                 :  *    release group
275                 :  *
276                 :  *  - use preallocation:
277                 :  *    find proper PA (per-inode or group)
278                 :  *    load group
279                 :  *    mark bits in on-disk bitmap
280                 :  *    release group
281                 :  *    release PA
282                 :  *
283                 :  *  - free:
284                 :  *    load group
285                 :  *    mark bits in on-disk bitmap
286                 :  *    release group
287                 :  *
288                 :  *  - discard preallocations in group:
289                 :  *    mark PAs deleted
290                 :  *    move them onto local list
291                 :  *    load on-disk bitmap
292                 :  *    load group
293                 :  *    remove PA from object (inode or locality group)
294                 :  *    mark free blocks in-core
295                 :  *
296                 :  *  - discard inode's preallocations:
297                 :  */
298                 :
299                 : /*
300                 :  * Locking rules
301                 :  *
302                 :  * Locks:
303                 :  *  - bitlock on a group        (group)
304                 :  *  - object (inode/locality)   (object)
305                 :  *  - per-pa lock               (pa)
306                 :  *
307                 :  * Paths:
308                 :  *  - new pa
309                 :  *    object
310                 :  *    group
311                 :  *
312                 :  *  - find and use pa:
313                 :  *    pa
314                 :  *
315                 :  *  - release consumed pa:
316                 :  *    pa
317                 :  *    group
318                 :  *    object
319                 :  *
320                 :  *  - generate in-core bitmap:
321                 :  *    group
322                 :  *        pa
323                 :  *
324                 :  *  - discard all for given object (inode, locality group):
325                 :  *    object
326                 :  *        pa
327                 :  *    group
328                 :  *
329                 :  *  - discard all for given group:
330                 :  *    group
331                 :  *        pa
332                 :  *    group
333                 :  *        object
334                 :  *
335                 :  */
336                 : static struct kmem_cache *ext4_pspace_cachep;
337                 : static struct kmem_cache *ext4_ac_cachep;
338                 : static struct kmem_cache *ext4_free_ext_cachep;
339                 : static void ext4_mb_generate_from_pa(struct super_block *sb, void *bitmap,
340                 :                                      ext4_group_t group);
341                 : static void ext4_mb_generate_from_freelist(struct super_block *sb, void *bitmap,
342                 :                                           ext4_group_t group);
```

```
343                :  static void release_blocks_on_commit(journal_t *journal, transaction_t *txn);
344                :
345                :  static inline void *mb_correct_addr_and_bit(int *bit, void *addr)
346                :  {
347                :  #if BITS_PER_LONG == 64
348                :          *bit += ((unsigned long) addr & 7UL) << 3;
349                :          addr = (void *) ((unsigned long) addr & ~7UL);
350                :  #elif BITS_PER_LONG == 32
351             -1 :          *bit += ((unsigned long) addr & 3UL) << 3;
352             -1 :          addr = (void *) ((unsigned long) addr & ~3UL);
353                :  #else
354                :  #error "how many bits you are?!"
355                :  #endif
356             -1 :          return addr;
357                :  }
358                :
359                :  static inline int mb_test_bit(int bit, void *addr)
360             -1 :  {
361                :          /*
362                :           * ext4_test_bit on architecture like powerpc
363                :           * needs unsigned long aligned address
364                :           */
365             -1 :          addr = mb_correct_addr_and_bit(&bit, addr);
366             -1 :          return ext4_test_bit(bit, addr);
367                :  }
368                :
369                :  static inline void mb_set_bit(int bit, void *addr)
370                :  {
371     1196559432 :          addr = mb_correct_addr_and_bit(&bit, addr);
372     1196559432 :          ext4_set_bit(bit, addr);
373                :  }
374                :
375                :  static inline void mb_clear_bit(int bit, void *addr)
376                :  {
377     1342143782 :          addr = mb_correct_addr_and_bit(&bit, addr);
378     1342143782 :          ext4_clear_bit(bit, addr);
379                :  }
380                :
381                :  static inline int mb_find_next_zero_bit(void *addr, int max, int start)
382      468250298 :  {
383      468250298 :          int fix = 0, ret, tmpmax;
384      468250298 :          addr = mb_correct_addr_and_bit(&fix, addr);
385      468250298 :          tmpmax = max + fix;
386      468250298 :          start += fix;
387                :
388      468250298 :          ret = ext4_find_next_zero_bit(addr, tmpmax, start) - fix;
389      468443040 :          if (ret > max)
390              0 :                  return max;
391      468443040 :          return ret;
392                :  }
393                :
394                :  static inline int mb_find_next_bit(void *addr, int max, int start)
395                :  {
396        9743927 :          int fix = 0, ret, tmpmax;
397        9743927 :          addr = mb_correct_addr_and_bit(&fix, addr);
398        9743927 :          tmpmax = max + fix;
399        9743927 :          start += fix;
400                :
401        9743927 :          ret = ext4_find_next_bit(addr, tmpmax, start) - fix;
402        9742503 :          if (ret > max)
403              0 :                  return max;
404        9742503 :          return ret;
405                :  }
406                :
407                :  static void *mb_find_buddy(struct ext4_buddy *e4b, int order, int *max)
408    -1648538238 :  {
409                :          char *bb;
410                :
411    -1648538238 :          BUG_ON(EXT4_MB_BITMAP(e4b) == EXT4_MB_BUDDY(e4b));
412    -1648493540 :          BUG_ON(max == NULL);
413                :
```

```
414    -1648120435 :         if (order > e4b->bd_blkbits + 1) {
415           7760 :                 *max = 0;
416           7760 :                 return NULL;
417                :         }
418                :
419                :         /* at order 0 we see each particular block */
420    -1648128195 :         *max = 1 << (e4b->bd_blkbits + 3);
421    -1648128195 :         if (order == 0)
422     1337797578 :                 return EXT4_MB_BITMAP(e4b);
423                :
424    -1676884250 :         bb = EXT4_MB_BUDDY(e4b) + EXT4_SB(e4b->bd_sb)->s_mb_offsets[order];
425    -1676884250 :         *max = EXT4_SB(e4b->bd_sb)->s_mb_maxs[order];
426                :
427     1309041523 :         return bb;
428                : }
429                :
430                : #ifdef DOUBLE_CHECK
431                : static void mb_free_blocks_double(struct inode *inode, struct ext4_buddy *e4b,
432                :                                   int first, int count)
433                : {
434                :         int i;
435                :         struct super_block *sb = e4b->bd_sb;
436                :
437                :         if (unlikely(e4b->bd_info->bb_bitmap == NULL))
438                :                 return;
439                :         assert_spin_locked(ext4_group_lock_ptr(sb, e4b->bd_group));
440                :         for (i = 0; i < count; i++) {
441                :                 if (!mb_test_bit(first + i, e4b->bd_info->bb_bitmap)) {
442                :                         ext4_fsblk_t blocknr;
443                :                         blocknr = e4b->bd_group * EXT4_BLOCKS_PER_GROUP(sb);
444                :                         blocknr += first + i;
445                :                         blocknr +=
446                :                             le32_to_cpu(EXT4_SB(sb)->s_es->s_first_data_block);
447                :                         ext4_grp_locked_error(sb, e4b->bd_group,
448                :                                               __func__, "double-free of inode"
449                :                                               " %lu's block %llu(bit %u in group %u)",
450                :                                               inode ? inode->i_ino : 0, blocknr,
451                :                                               first + i, e4b->bd_group);
452                :                 }
453                :                 mb_clear_bit(first + i, e4b->bd_info->bb_bitmap);
454                :         }
455                : }
456                :
457                : static void mb_mark_used_double(struct ext4_buddy *e4b, int first, int count)
458                : {
459                :         int i;
460                :
461                :         if (unlikely(e4b->bd_info->bb_bitmap == NULL))
462                :                 return;
463                :         assert_spin_locked(ext4_group_lock_ptr(e4b->bd_sb, e4b->bd_group));
464                :         for (i = 0; i < count; i++) {
465                :                 BUG_ON(mb_test_bit(first + i, e4b->bd_info->bb_bitmap));
466                :                 mb_set_bit(first + i, e4b->bd_info->bb_bitmap);
467                :         }
468                : }
469                :
470                : static void mb_cmp_bitmaps(struct ext4_buddy *e4b, void *bitmap)
471                : {
472                :         if (memcmp(e4b->bd_info->bb_bitmap, bitmap, e4b->bd_sb->s_blocksize)) {
473                :                 unsigned char *b1, *b2;
474                :                 int i;
475                :                 b1 = (unsigned char *) e4b->bd_info->bb_bitmap;
476                :                 b2 = (unsigned char *) bitmap;
477                :                 for (i = 0; i < e4b->bd_sb->s_blocksize; i++) {
478                :                         if (b1[i] != b2[i]) {
479                :                                 printk(KERN_ERR "corruption in group %u "
480                :                                        "at byte %u(%u): %x in copy != %x "
481                :                                        "on disk/prealloc\n",
482                :                                        e4b->bd_group, i, i * 8, b1[i], b2[i]);
483                :                                 BUG();
484                :                         }
```

```
485                 :                 }
486                 :         }
487                 : }
488                 :
489                 : #else
490                 : static inline void mb_free_blocks_double(struct inode *inode,
491                 :                                 struct ext4_buddy *e4b, int first, int count)
492                 : {
493                 :         return;
494                 : }
495                 : static inline void mb_mark_used_double(struct ext4_buddy *e4b,
496                 :                                         int first, int count)
497                 : {
498                 :         return;
499                 : }
500                 : static inline void mb_cmp_bitmaps(struct ext4_buddy *e4b, void *bitmap)
501                 : {
502                 :         return;
503                 : }
504                 : #endif
505                 :
506                 : #ifdef AGGRESSIVE_CHECK
507                 :
508                 : #define MB_CHECK_ASSERT(assert)                                         \
509                 : do {                                                                    \
510                 :         if (!(assert)) {                                              \
511                 :                 printk(KERN_EMERG                                     \
512                 :                         "Assertion failure in %s() at %s:%d: \"%s\"\n",     \
513                 :                         function, file, line, # assert);              \
514                 :                 BUG();                                                \
515                 :         }                                                             \
516                 : } while (0)
517                 :
518                 : static int __mb_check_buddy(struct ext4_buddy *e4b, char *file,
519                 :                                 const char *function, int line)
520                 : {
521                 :         struct super_block *sb = e4b->bd_sb;
522                 :         int order = e4b->bd_blkbits + 1;
523                 :         int max;
524                 :         int max2;
525                 :         int i;
526                 :         int j;
527                 :         int k;
528                 :         int count;
529                 :         struct ext4_group_info *grp;
530                 :         int fragments = 0;
531                 :         int fstart;
532                 :         struct list_head *cur;
533                 :         void *buddy;
534                 :         void *buddy2;
535                 :
536                 :         {
537                 :                 static int mb_check_counter;
538                 :                 if (mb_check_counter++ % 100 != 0)
539                 :                         return 0;
540                 :         }
541                 :
542                 :         while (order > 1) {
543                 :                 buddy = mb_find_buddy(e4b, order, &max);
544                 :                 MB_CHECK_ASSERT(buddy);
545                 :                 buddy2 = mb_find_buddy(e4b, order - 1, &max2);
546                 :                 MB_CHECK_ASSERT(buddy2);
547                 :                 MB_CHECK_ASSERT(buddy != buddy2);
548                 :                 MB_CHECK_ASSERT(max * 2 == max2);
549                 :
550                 :                 count = 0;
551                 :                 for (i = 0; i < max; i++) {
552                 :
553                 :                         if (mb_test_bit(i, buddy)) {
554                 :                                 /* only single bit in buddy2 may be 1 */
555                 :                                 if (!mb_test_bit(i << 1, buddy2)) {
```

```
556                                             MB_CHECK_ASSERT(
557                                                     mb_test_bit((i<<1)+1, buddy2));
558                                     } else if (!mb_test_bit((i << 1) + 1, buddy2)) {
559                                             MB_CHECK_ASSERT(
560                                                     mb_test_bit(i << 1, buddy2));
561                                     }
562                                     continue;
563                             }
564 
565                             /* both bits in buddy2 must be 0 */
566                             MB_CHECK_ASSERT(mb_test_bit(i << 1, buddy2));
567                             MB_CHECK_ASSERT(mb_test_bit((i << 1) + 1, buddy2));
568 
569                             for (j = 0; j < (1 << order); j++) {
570                                     k = (i * (1 << order)) + j;
571                                     MB_CHECK_ASSERT(
572                                             !mb_test_bit(k, EXT4_MB_BITMAP(e4b)));
573                             }
574                             count++;
575                     }
576                     MB_CHECK_ASSERT(e4b->bd_info->bb_counters[order] == count);
577                     order--;
578             }
579 
580             fstart = -1;
581             buddy = mb_find_buddy(e4b, 0, &max);
582             for (i = 0; i < max; i++) {
583                     if (!mb_test_bit(i, buddy)) {
584                             MB_CHECK_ASSERT(i >= e4b->bd_info->bb_first_free);
585                             if (fstart == -1) {
586                                     fragments++;
587                                     fstart = i;
588                             }
589                             continue;
590                     }
591                     fstart = -1;
592                     /* check used bits only */
593                     for (j = 0; j < e4b->bd_blkbits + 1; j++) {
594                             buddy2 = mb_find_buddy(e4b, j, &max2);
595                             k = i >> j;
596                             MB_CHECK_ASSERT(k < max2);
597                             MB_CHECK_ASSERT(mb_test_bit(k, buddy2));
598                     }
599             }
600             MB_CHECK_ASSERT(!EXT4_MB_GRP_NEED_INIT(e4b->bd_info));
601             MB_CHECK_ASSERT(e4b->bd_info->bb_fragments == fragments);
602 
603             grp = ext4_get_group_info(sb, e4b->bd_group);
604             buddy = mb_find_buddy(e4b, 0, &max);
605             list_for_each(cur, &grp->bb_prealloc_list) {
606                     ext4_group_t groupnr;
607                     struct ext4_prealloc_space *pa;
608                     pa = list_entry(cur, struct ext4_prealloc_space, pa_group_list);
609                     ext4_get_group_no_and_offset(sb, pa->pa_pstart, &groupnr, &k);
610                     MB_CHECK_ASSERT(groupnr == e4b->bd_group);
611                     for (i = 0; i < pa->pa_len; i++)
612                             MB_CHECK_ASSERT(mb_test_bit(k + i, buddy));
613             }
614             return 0;
615     }
616     #undef MB_CHECK_ASSERT
617     #define mb_check_buddy(e4b) __mb_check_buddy(e4b,       \
618                                             __FILE__, __func__, __LINE__)
619     #else
620     #define mb_check_buddy(e4b)
621     #endif
622 
623     /* FIXME!! need more doc */
624     static void ext4_mb_mark_free_simple(struct super_block *sb,
625                                     void *buddy, unsigned first, int len,
626                                             struct ext4_group_info *grp)
```

```
627                 : {
628         1778729 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
629                 :         unsigned short min;
630                 :         unsigned short max;
631                 :         unsigned short chunk;
632                 :         unsigned short border;
633                 :
634         3557458 :         BUG_ON(len > EXT4_BLOCKS_PER_GROUP(sb));
635                 :
636         1778728 :         border = 2 << sb->s_blocksize_bits;
637                 :
638         7065797 :         while (len > 0) {
639                 :                 /* find how many blocks can be covered since this position */
640        10574142 :                 max = ffs(first | border) - 1;
641                 :
642                 :                 /* find how many blocks of power 2 we need to mark */
643         5287071 :                 min = fls(len) - 1;
644                 :
645         5287071 :                 if (max < min)
646         3172819 :                         min = max;
647         5287071 :                 chunk = 1 << min;
648                 :
649                 :                 /* mark multiblock chunks only */
650         5287071 :                 grp->bb_counters[min]++;
651         5287071 :                 if (min > 0)
652         4404598 :                         mb_clear_bit(first >> min,
653                 :                                      buddy + sbi->s_mb_offsets[min]);
654                 :
655         5287069 :                 len -= chunk;
656         5287069 :                 first += chunk;
657                 :         }
658                 : }
659                 :
660                 : static noinline_for_stack
661                 : void ext4_mb_generate_buddy(struct super_block *sb,
662                 :                             void *buddy, void *bitmap, ext4_group_t group)
663           63148 : {
664           63148 :         struct ext4_group_info *grp = ext4_get_group_info(sb, group);
665           63148 :         unsigned short max = EXT4_BLOCKS_PER_GROUP(sb);
666           63148 :         unsigned short i = 0;
667                 :         unsigned short first;
668                 :         unsigned short len;
669           63148 :         unsigned free = 0;
670           63148 :         unsigned fragments = 0;
671           63148 :         unsigned long long period = get_cycles();
672                 :
673                 :         /* initialize buddy from bitmap which is aggregation
674                 :          * of on-disk bitmap and preallocations */
675           63148 :         i = mb_find_next_zero_bit(bitmap, max, 0);
676           63152 :         grp->bb_first_free = i;
677         1992027 :         while (i < max) {
678         1865727 :                 fragments++;
679         1865727 :                 first = i;
680         3731453 :                 i = mb_find_next_bit(bitmap, max, i);
681         1865726 :                 len = i - first;
682         1865726 :                 free += len;
683         1865726 :                 if (len > 1)
684         1778729 :                         ext4_mb_mark_free_simple(sb, buddy, first, len, grp);
685                 :                 else
686           86997 :                         grp->bb_counters[0]++;
687         1865723 :                 if (i < max)
688         1806854 :                         i = mb_find_next_zero_bit(bitmap, max, i);
689                 :         }
690           63148 :         grp->bb_fragments = fragments;
691                 :
692           63148 :         if (free != grp->bb_free) {
693               0 :                 ext4_grp_locked_error(sb, group,  __func__,
694                 :                         "EXT4-fs: group %u: %u blocks in bitmap, %u in gd",
695                 :                         group, free, grp->bb_free);
696                 :                 /*
697                 :                  * If we intent to continue, we consider group descritor
```

```
    698                 :                  * corrupt and update bb_free using bitmap value
    699                 :                  */
    700               0 :                 grp->bb_free = free;
    701                 :         }
    702                 :
    703           63148 :         clear_bit(EXT4_GROUP_INFO_NEED_INIT_BIT, &(grp->bb_state));
    704                 :
    705           63148 :         period = get_cycles() - period;
    706           63148 :         spin_lock(&EXT4_SB(sb)->s_bal_lock);
    707           63148 :         EXT4_SB(sb)->s_mb_buddies_generated++;
    708           63148 :         EXT4_SB(sb)->s_mb_generation_time += period;
    709           63148 :         spin_unlock(&EXT4_SB(sb)->s_bal_lock);
    710           63148 : }
    711                 :
    712                 : /* The buddy information is attached the buddy cache inode
    713                 :  * for convenience. The information regarding each group
    714                 :  * is loaded via ext4_mb_load_buddy. The information involve
    715                 :  * block bitmap and buddy information. The information are
    716                 :  * stored in the inode as
    717                 :  *
    718                 :  * {                        page                        }
    719                 :  * [ group 0 bitmap][ group 0 buddy] [group 1][ group 1]...
    720                 :  *
    721                 :  *
    722                 :  * one block each for bitmap and buddy information.
    723                 :  * So for each group we take up 2 blocks. A page can
    724                 :  * contain blocks_per_page (PAGE_CACHE_SIZE / blocksize)  blocks.
    725                 :  * So it can have information regarding groups_per_page which
    726                 :  * is blocks_per_page/2
    727                 :  */
    728                 :
    729                 : static int ext4_mb_init_cache(struct page *page, char *incore)
    730          102277 : {
    731                 :         ext4_group_t ngroups;
    732                 :         int blocksize;
    733                 :         int blocks_per_page;
    734                 :         int groups_per_page;
    735          102277 :         int err = 0;
    736                 :         int i;
    737                 :         ext4_group_t first_group;
    738                 :         int first_block;
    739                 :         struct super_block *sb;
    740                 :         struct buffer_head *bhs;
    741                 :         struct buffer_head **bh;
    742                 :         struct inode *inode;
    743                 :         char *data;
    744                 :         char *bitmap;
    745                 :
    746                 :         mb_debug("init page %lu\n", page->index);
    747                 :
    748          102277 :         inode = page->mapping->host;
    749          102277 :         sb = inode->i_sb;
    750          102277 :         ngroups = ext4_get_groups_count(sb);
    751          102277 :         blocksize = 1 << inode->i_blkbits;
    752          102277 :         blocks_per_page = PAGE_CACHE_SIZE / blocksize;
    753                 :
    754          102277 :         groups_per_page = blocks_per_page >> 1;
    755          102277 :         if (groups_per_page == 0)
    756           91623 :                 groups_per_page = 1;
    757                 :
    758                 :         /* allocate buffer_heads to read bitmaps */
    759          102277 :         if (groups_per_page > 1) {
    760            6566 :                 err = -ENOMEM;
    761            6566 :                 i = sizeof(struct buffer_head *) * groups_per_page;
    762            6566 :                 bh = kzalloc(i, GFP_NOFS);
    763            6566 :                 if (bh == NULL)
    764               0 :                         goto out;
    765                 :         } else
    766           95711 :                 bh = &bhs;
    767                 :
    768          102277 :         first_group = page->index * blocks_per_page / 2;
```

```
   769                :
   770                :         /* read all groups the page covers into the cache */
   771         211120 :         for (i = 0; i < groups_per_page; i++) {
   772                :                 struct ext4_group_desc *desc;
   773                :
   774         108843 :                 if (first_group + i >= ngroups)
   775              0 :                         break;
   776                :
   777         108843 :                 err = -EIO;
   778         108843 :                 desc = ext4_get_group_desc(sb, first_group + i, NULL);
   779         108843 :                 if (desc == NULL)
   780              0 :                         goto out;
   781                :
   782         108843 :                 err = -ENOMEM;
   783         217686 :                 bh[i] = sb_getblk(sb, ext4_block_bitmap(sb, desc));
   784         108843 :                 if (bh[i] == NULL)
   785              0 :                         goto out;
   786                :
   787         108843 :                 if (bitmap_uptodate(bh[i]))
   788          50984 :                         continue;
   789                :
   790          57859 :                 lock_buffer(bh[i]);
   791          57859 :                 if (bitmap_uptodate(bh[i])) {
   792              0 :                         unlock_buffer(bh[i]);
   793              0 :                         continue;
   794                :                 }
   795          57859 :                 ext4_lock_group(sb, first_group + i);
   796          57859 :                 if (desc->bg_flags & cpu_to_le16(EXT4_BG_BLOCK_UNINIT)) {
   797          33934 :                         ext4_init_block_bitmap(sb, bh[i],
   798                :                                                first_group + i, desc);
   799          33934 :                         set_bitmap_uptodate(bh[i]);
   800          33934 :                         set_buffer_uptodate(bh[i]);
   801          33934 :                         ext4_unlock_group(sb, first_group + i);
   802          33934 :                         unlock_buffer(bh[i]);
   803          33934 :                         continue;
   804                :                 }
   805          23925 :                 ext4_unlock_group(sb, first_group + i);
   806          47850 :                 if (buffer_uptodate(bh[i])) {
   807                :                         /*
   808                :                          * if not uninit if bh is uptodate,
   809                :                          * bitmap is also uptodate
   810                :                          */
   811              4 :                         set_bitmap_uptodate(bh[i]);
   812              4 :                         unlock_buffer(bh[i]);
   813              4 :                         continue;
   814                :                 }
   815          23921 :                 get_bh(bh[i]);
   816                :                 /*
   817                :                  * submit the buffer_head for read. We can
   818                :                  * safely mark the bitmap as uptodate now.
   819                :                  * We do it here so the bitmap uptodate bit
   820                :                  * get set with buffer lock held.
   821                :                  */
   822          23921 :                 set_bitmap_uptodate(bh[i]);
   823          23921 :                 bh[i]->b_end_io = end_buffer_read_sync;
   824          23921 :                 submit_bh(READ, bh[i]);
   825                :                 mb_debug("read bitmap for group %u\n", first_group + i);
   826                :         }
   827                :
   828                :         /* wait for I/O completion */
   829         211120 :         for (i = 0; i < groups_per_page && bh[i]; i++)
   830         108843 :                 wait_on_buffer(bh[i]);
   831                :
   832         102277 :         err = -EIO;
   833         211120 :         for (i = 0; i < groups_per_page && bh[i]; i++)
   834         217686 :                 if (!buffer_uptodate(bh[i]))
   835              0 :                         goto out;
   836                :
   837         102277 :         err = 0;
   838         102277 :         first_block = page->index * blocks_per_page;
   839                :         /* init the page  */
```

```
840          102277 :          memset(page_address(page), 0xff, PAGE_CACHE_SIZE);
841          228340 :          for (i = 0; i < blocks_per_page; i++) {
842                 :                  int group;
843                 :                  struct ext4_group_info *grinfo;
844                 :
845          126063 :                  group = (first_block + i) >> 1;
846          126063 :                  if (group >= ngroups)
847               0 :                          break;
848                 :
849                 :                  /*
850                 :                   * data carry information regarding this
851                 :                   * particular group in the format specified
852                 :                   * above
853                 :                   *
854                 :                   */
855          126063 :                  data = page_address(page) + (i * blocksize);
856          126063 :                  bitmap = bh[group - first_group]->b_data;
857                 :
858                 :                  /*
859                 :                   * We place the buddy block and bitmap block
860                 :                   * close together
861                 :                   */
862          126063 :                  if ((first_block + i) & 1) {
863                 :                          /* this is block of buddy */
864           63148 :                          BUG_ON(incore == NULL);
865                 :                          mb_debug("put buddy for group %u in page %lu/%x\n",
866                 :                                  group, page->index, i * blocksize);
867          126296 :                          grinfo = ext4_get_group_info(sb, group);
868           63148 :                          grinfo->bb_fragments = 0;
869          126296 :                          memset(grinfo->bb_counters, 0,
870                 :                                 sizeof(unsigned short)*(sb->s_blocksize_bits+2));
871                 :                          /*
872                 :                           * incore got set to the group block bitmap below
873                 :                           */
874           63148 :                          ext4_lock_group(sb, group);
875           63148 :                          ext4_mb_generate_buddy(sb, data, incore, group);
876           63148 :                          ext4_unlock_group(sb, group);
877           63148 :                          incore = NULL;
878                 :                  } else {
879                 :                          /* this is block of bitmap */
880           62915 :                          BUG_ON(incore != NULL);
881                 :                          mb_debug("put bitmap for group %u in page %lu/%x\n",
882                 :                                  group, page->index, i * blocksize);
883                 :
884                 :                          /* see comments in ext4_mb_put_pa() */
885           62915 :                          ext4_lock_group(sb, group);
886          125830 :                          memcpy(data, bitmap, blocksize);
887                 :
888                 :                          /* mark all preallocated blks used in in-core bitmap */
889           62915 :                          ext4_mb_generate_from_pa(sb, data, group);
890           62915 :                          ext4_mb_generate_from_freelist(sb, data, group);
891           62915 :                          ext4_unlock_group(sb, group);
892                 :
893                 :                          /* set incore so that the buddy information can be
894                 :                           * generated using this
895                 :                           */
896           62915 :                          incore = data;
897                 :                  }
898                 :          }
899                 :          SetPageUptodate(page);
900                 :
901          102277 : out:
902          102277 :          if (bh) {
903          211120 :                  for (i = 0; i < groups_per_page && bh[i]; i++)
904          108843 :                          brelse(bh[i]);
905          102277 :                  if (bh != &bhs)
906            6566 :                          kfree(bh);
907                 :          }
908          102277 :          return err;
909                 : }
910                 :
```

```
   911                :  static noinline_for_stack int
   912                :  ext4_mb_load_buddy(struct super_block *sb, ext4_group_t group,
   913                :                                           struct ext4_buddy *e4b)
   914         42777130 :  {
   915                :          int blocks_per_page;
   916                :          int block;
   917                :          int pnum;
   918                :          int poff;
   919                :          struct page *page;
   920                :          int ret;
   921                :          struct ext4_group_info *grp;
   922         42777130 :          struct ext4_sb_info *sbi = EXT4_SB(sb);
   923         42777130 :          struct inode *inode = sbi->s_buddy_cache;
   924                :
   925                :          mb_debug("load group %u\n", group);
   926                :
   927         42777130 :          blocks_per_page = PAGE_CACHE_SIZE / sb->s_blocksize;
   928         42777130 :          grp = ext4_get_group_info(sb, group);
   929                :
   930         42777130 :          e4b->bd_blkbits = sb->s_blocksize_bits;
   931         42777130 :          e4b->bd_info = ext4_get_group_info(sb, group);
   932         42777130 :          e4b->bd_sb = sb;
   933         42777130 :          e4b->bd_group = group;
   934         42777130 :          e4b->bd_buddy_page = NULL;
   935         42777130 :          e4b->bd_bitmap_page = NULL;
   936         42777130 :          e4b->alloc_semp = &grp->alloc_sem;
   937                :
   938                :          /* Take the read lock on the group alloc
   939                :           * sem. This would make sure a parallel
   940                :           * ext4_mb_init_group happening on other
   941                :           * groups mapped by the page is blocked
   942                :           * till we are done with allocation
   943                :           */
   944         42777130 :          down_read(e4b->alloc_semp);
   945                :
   946                :          /*
   947                :           * the buddy cache inode stores the block bitmap
   948                :           * and buddy information in consecutive blocks.
   949                :           * So for each group we need two blocks.
   950                :           */
   951         42785659 :          block = group * 2;
   952         42785659 :          pnum = block / blocks_per_page;
   953         42785659 :          poff = block % blocks_per_page;
   954                :
   955                :          /* we could use find_or_create_page(), but it locks page
   956                :           * what we'd like to avoid in fast path ... */
   957         42785659 :          page = find_get_page(inode->i_mapping, pnum);
   958         42793477 :          if (page == NULL || !PageUptodate(page)) {
   959            35923 :                  if (page)
   960                :                          /*
   961                :                           * drop the page reference and try
   962                :                           * to get the page with lock. If we
   963                :                           * are not uptodate that implies
   964                :                           * somebody just created the page but
   965                :                           * is yet to initialize the same. So
   966                :                           * wait for it to initialize.
   967                :                           */
   968             1386 :                          page_cache_release(page);
   969            35923 :                  page = find_or_create_page(inode->i_mapping, pnum, GFP_NOFS);
   970            28544 :                  if (page) {
   971            28544 :                          BUG_ON(page->mapping != inode->i_mapping);
   972            28544 :                          if (!PageUptodate(page)) {
   973            27155 :                                  ret = ext4_mb_init_cache(page, NULL);
   974            27155 :                                  if (ret) {
   975                0 :                                          unlock_page(page);
   976                0 :                                          goto err;
   977                :                                  }
   978            27155 :                                  mb_cmp_bitmaps(e4b, page_address(page) +
   979                :                                                 (poff * sb->s_blocksize));
   980                :                          }
   981            28544 :                          unlock_page(page);
```

```
   982                :                 }
   983                :         }
   984         42785005 :         if (page == NULL || !PageUptodate(page)) {
   985                0 :                 ret = -EIO;
   986                0 :                 goto err;
   987                :         }
   988         42792212 :         e4b->bd_bitmap_page = page;
   989         42792212 :         e4b->bd_bitmap = page_address(page) + (poff * sb->s_blocksize);
   990         42776270 :         mark_page_accessed(page);
   991                :
   992         42778530 :         block++;
   993         42778530 :         pnum = block / blocks_per_page;
   994         42778530 :         poff = block % blocks_per_page;
   995                :
   996         42778530 :         page = find_get_page(inode->i_mapping, pnum);
   997         42794042 :         if (page == NULL || !PageUptodate(page)) {
   998            31017 :                 if (page)
   999              427 :                         page_cache_release(page);
  1000            31017 :                 page = find_or_create_page(inode->i_mapping, pnum, GFP_NOFS);
  1001            27952 :                 if (page) {
  1002            27952 :                         BUG_ON(page->mapping != inode->i_mapping);
  1003            27952 :                         if (!PageUptodate(page)) {
  1004            27384 :                                 ret = ext4_mb_init_cache(page, e4b->bd_bitmap);
  1005            27384 :                                 if (ret) {
  1006                0 :                                         unlock_page(page);
  1007                0 :                                         goto err;
  1008                :                                 }
  1009                :                         }
  1010            27952 :                         unlock_page(page);
  1011                :                 }
  1012                :         }
  1013         42790660 :         if (page == NULL || !PageUptodate(page)) {
  1014                0 :                 ret = -EIO;
  1015                0 :                 goto err;
  1016                :         }
  1017         42794172 :         e4b->bd_buddy_page = page;
  1018         42794172 :         e4b->bd_buddy = page_address(page) + (poff * sb->s_blocksize);
  1019         42793175 :         mark_page_accessed(page);
  1020                :
  1021         42791413 :         BUG_ON(e4b->bd_bitmap_page == NULL);
  1022         42786306 :         BUG_ON(e4b->bd_buddy_page == NULL);
  1023                :
  1024         42789316 :         return 0;
  1025                :
  1026                0 : err:
  1027                0 :         if (e4b->bd_bitmap_page)
  1028                0 :                 page_cache_release(e4b->bd_bitmap_page);
  1029                0 :         if (e4b->bd_buddy_page)
  1030                0 :                 page_cache_release(e4b->bd_buddy_page);
  1031                0 :         e4b->bd_buddy = NULL;
  1032                0 :         e4b->bd_bitmap = NULL;
  1033                :
  1034                :         /* Done with the buddy cache */
  1035                0 :         up_read(e4b->alloc_semp);
  1036                0 :         return ret;
  1037                : }
  1038                :
  1039                : static void ext4_mb_release_desc(struct ext4_buddy *e4b)
  1040         42790512 : {
  1041         42790512 :         if (e4b->bd_bitmap_page)
  1042         42790851 :                 page_cache_release(e4b->bd_bitmap_page);
  1043         42793813 :         if (e4b->bd_buddy_page)
  1044         42793940 :                 page_cache_release(e4b->bd_buddy_page);
  1045                :         /* Done with the buddy cache */
  1046         42794662 :         if (e4b->alloc_semp)
  1047         29513971 :                 up_read(e4b->alloc_semp);
  1048         42794538 : }
  1049                :
  1050                :
  1051                : static int mb_find_order_for_block(struct ext4_buddy *e4b, int block)
  1052        954072331 : {
```

```
1053  954072331 :          int order = 1;
1054            :          void *bb;
1055            :
1056  954072331 :          BUG_ON(EXT4_MB_BITMAP(e4b) == EXT4_MB_BUDDY(e4b));
1057  954073141 :          BUG_ON(block >= (1 << (e4b->bd_blkbits + 3)));
1058            :
1059  953076378 :          bb = EXT4_MB_BUDDY(e4b);
1060         -1 :          while (order <= e4b->bd_blkbits + 1) {
1061         -1 :                  block = block >> 1;
1062         -1 :                  if (!mb_test_bit(block, bb)) {
1063            :                          /* this block is part of buddy of order 'order' */
1064  707700996 :                          return order;
1065            :                  }
1066         -1 :                  bb += 1 << (e4b->bd_blkbits - order);
1067         -1 :                  order++;
1068            :          }
1069  246701762 :          return 0;
1070            : }
1071            :
1072            : static void mb_clear_bits(void *bm, int cur, int len)
1073            : {
1074            :          __u32 *addr;
1075            :
1076    3896262 :          len = cur + len;
1077   60897619 :          while (cur < len) {
1078   57001340 :                  if ((cur & 31) == 0 && (len - cur) >= 32) {
1079            :                          /* fast path: clear whole word at once */
1080    3121123 :                          addr = bm + (cur >> 3);
1081    3121123 :                          *addr = 0;
1082    3121123 :                          cur += 32;
1083            :                          continue;
1084            :                  }
1085   53880217 :                  mb_clear_bit(cur, bm);
1086   53880234 :                  cur++;
1087            :          }
1088            : }
1089            :
1090            : static void mb_set_bits(void *bm, int cur, int len)
1091  236059858 : {
1092            :          __u32 *addr;
1093            :
1094  236059858 :          len = cur + len;
1095 1069933172 :          while (cur < len) {
1096  597813456 :                  if ((cur & 31) == 0 && (len - cur) >= 32) {
1097            :                          /* fast path: set whole word at once */
1098   61238153 :                          addr = bm + (cur >> 3);
1099   61238153 :                          *addr = 0xffffffff;
1100   61238153 :                          cur += 32;
1101   61238153 :                          continue;
1102            :                  }
1103  536575303 :                  mb_set_bit(cur, bm);
1104  536575303 :                  cur++;
1105            :          }
1106  236059858 : }
1107            :
1108            : static void mb_free_blocks(struct inode *inode, struct ext4_buddy *e4b,
1109            :                            int first, int count)
1110   11122973 : {
1111   11122973 :          int block = 0;
1112   11122973 :          int max = 0;
1113            :          int order;
1114            :          void *buddy;
1115            :          void *buddy2;
1116   11122973 :          struct super_block *sb = e4b->bd_sb;
1117            :
1118   11122973 :          BUG_ON(first + count > (sb->s_blocksize << 3));
1119   33370545 :          assert_spin_locked(ext4_group_lock_ptr(sb, e4b->bd_group));
1120            :          mb_check_buddy(e4b);
1121            :          mb_free_blocks_double(inode, e4b, first, count);
1122            :
1123   11137482 :          e4b->bd_info->bb_free += count;
```

```
1124  11137482 :         if (first < e4b->bd_info->bb_first_free)
1125    968546 :                 e4b->bd_info->bb_first_free = first;
1126           :
1127           :         /* let's maintain fragments counter */
1128  11137482 :         if (first != 0)
1129  11119377 :                 block = !mb_test_bit(first - 1, EXT4_MB_BITMAP(e4b));
1130  22274090 :         if (first + count < EXT4_SB(sb)->s_mb_maxs[0])
1131  11095851 :                 max = !mb_test_bit(first + count, EXT4_MB_BITMAP(e4b));
1132  11137578 :         if (block && max)
1133   1468024 :                 e4b->bd_info->bb_fragments--;
1134   9669554 :         else if (!block && !max)
1135   3273666 :                 e4b->bd_info->bb_fragments++;
1136           :
1137           :         /* let's maintain buddy itself */
1138 632695391 :         while (count-- > 0) {
1139 621570655 :                 block = first++;
1140 621570655 :                 order = 0;
1141           :
1142 621570655 :                 if (!mb_test_bit(block, EXT4_MB_BITMAP(e4b))) {
1143           :                         ext4_fsblk_t blocknr;
1144         0 :                         blocknr = e4b->bd_group * EXT4_BLOCKS_PER_GROUP(sb);
1145         0 :                         blocknr += block;
1146         0 :                         blocknr +=
1147           :                             le32_to_cpu(EXT4_SB(sb)->s_es->s_first_data_block);
1148         0 :                         ext4_grp_locked_error(sb, e4b->bd_group,
1149           :                                               __func__, "double-free of inode"
1150           :                                               " %lu's block %llu(bit %u in group %u)",
1151           :                                               inode ? inode->i_ino : 0, blocknr, block,
1152           :                                               e4b->bd_group);
1153           :                 }
1154 621572163 :                 mb_clear_bit(block, EXT4_MB_BITMAP(e4b));
1155 621570984 :                 e4b->bd_info->bb_counters[order]++;
1156           :
1157           :                 /* start of the buddy */
1158 621570984 :                 buddy = mb_find_buddy(e4b, order, &max);
1159           :
1160           :                 do {
1161 1228899175 :                         block &= ~1UL;
1162 1228899175 :                         if (mb_test_bit(block, buddy) ||
1163           :                                         mb_test_bit(block + 1, buddy))
1164           :                                 break;
1165           :
1166           :                         /* both the buddies are free, try to coalesce them */
1167 607336539 :                         buddy2 = mb_find_buddy(e4b, order + 1, &max);
1168           :
1169 607336692 :                         if (!buddy2)
1170      7760 :                                 break;
1171           :
1172 607328932 :                         if (order > 0) {
1173           :                                 /* for special purposes, we don't set
1174           :                                  * free bits in bitmap */
1175 297610786 :                                 mb_set_bit(block, buddy);
1176 297610786 :                                 mb_set_bit(block + 1, buddy);
1177           :                         }
1178 607328932 :                         e4b->bd_info->bb_counters[order]--;
1179 607328932 :                         e4b->bd_info->bb_counters[order]--;
1180           :
1181 607328932 :                         block = block >> 1;
1182 607328932 :                         order++;
1183 607328932 :                         e4b->bd_info->bb_counters[order]++;
1184           :
1185 607328932 :                         mb_clear_bit(block, buddy2);
1186 607328480 :                         buddy = buddy2;
1187 607328480 :                 } while (1);
1188           :         }
1189           :         mb_check_buddy(e4b);
1190  11124736 : }
1191           :
1192           : static int mb_find_extent(struct ext4_buddy *e4b, int order, int block,
1193           :                                 int needed, struct ext4_free_extent *ex)
1194  468461813 : {
```

```
1195  468461813 :          int next = block;
1196            :          int max;
1197            :          int ord;
1198            :          void *buddy;
1199            :
1200 1405385439 :          assert_spin_locked(ext4_group_lock_ptr(e4b->bd_sb, e4b->bd_group));
1201  468461626 :          BUG_ON(ex == NULL);
1202            :
1203  468502276 :          buddy = mb_find_buddy(e4b, order, &max);
1204  468456346 :          BUG_ON(buddy == NULL);
1205  468462938 :          BUG_ON(block >= max);
1206  468475933 :          if (mb_test_bit(block, buddy)) {
1207    5295167 :                  ex->fe_len = 0;
1208    5295167 :                  ex->fe_start = 0;
1209    5295167 :                  ex->fe_group = 0;
1210    5295167 :                  return 0;
1211            :          }
1212            :
1213            :          /* FIXME dorp order completely ? */
1214  463173754 :          if (likely(order == 0)) {
1215            :                  /* find actual order */
1216  463151929 :                  order = mb_find_order_for_block(e4b, block);
1217  463261708 :                  block = block >> order;
1218            :          }
1219            :
1220  463239457 :          ex->fe_len = 1 << order;
1221  463239457 :          ex->fe_start = block << order;
1222  463239457 :          ex->fe_group = e4b->bd_group;
1223            :
1224            :          /* calc difference from given start */
1225  463239457 :          next = next - ex->fe_start;
1226  463239457 :          ex->fe_len -= next;
1227  463239457 :          ex->fe_start += next;
1228            :
1229 1352826376 :          while (needed > ex->fe_len &&
1230            :                 (buddy = mb_find_buddy(e4b, order, &max))) {
1231            :
1232  852766233 :                  if (block + 1 >= max)
1233    1137670 :                          break;
1234            :
1235  851628563 :                  next = (block + 1) * (1 << order);
1236  851628563 :                  if (mb_test_bit(next, EXT4_MB_BITMAP(e4b)))
1237  425462341 :                          break;
1238            :
1239  426363441 :                  ord = mb_find_order_for_block(e4b, next);
1240            :
1241  426347462 :                  order = ord;
1242  426347462 :                  block = next >> order;
1243  426347462 :                  ex->fe_len += 1 << order;
1244            :          }
1245            :
1246  463392242 :          BUG_ON(ex->fe_start + ex->fe_len > (1 << (e4b->bd_blkbits + 3)));
1247  463286211 :          return ex->fe_len;
1248            : }
1249            :
1250            : static int mb_mark_used(struct ext4_buddy *e4b, struct ext4_free_extent *ex)
1251   13280387 : {
1252            :          int ord;
1253   13280387 :          int mlen = 0;
1254   13280387 :          int max = 0;
1255            :          int cur;
1256   13280387 :          int start = ex->fe_start;
1257   13280387 :          int len = ex->fe_len;
1258   13280387 :          unsigned ret = 0;
1259   13280387 :          int len0 = len;
1260            :          void *buddy;
1261            :
1262   13280387 :          BUG_ON(start + len > (e4b->bd_sb->s_blocksize << 3));
1263   13280287 :          BUG_ON(e4b->bd_group != ex->fe_group);
1264   39841194 :          assert_spin_locked(ext4_group_lock_ptr(e4b->bd_sb, e4b->bd_group));
1265            :          mb_check_buddy(e4b);
```

```
1266                :         mb_mark_used_double(e4b, start, len);
1267                :
1268       13280373 :         e4b->bd_info->bb_free -= len;
1269       13280373 :         if (e4b->bd_info->bb_first_free == start)
1270        2137713 :                 e4b->bd_info->bb_first_free += len;
1271                :
1272                :         /* let's maintain fragments counter */
1273       13280373 :         if (start != 0)
1274       13248152 :                 mlen = !mb_test_bit(start - 1, EXT4_MB_BITMAP(e4b));
1275       26561284 :         if (start + len < EXT4_SB(e4b->bd_sb)->s_mb_maxs[0])
1276       13229603 :                 max = !mb_test_bit(start + len, EXT4_MB_BITMAP(e4b));
1277       13280685 :         if (mlen && max)
1278        2576511 :                 e4b->bd_info->bb_fragments++;
1279       10704174 :         else if (!mlen && !max)
1280        2748922 :                 e4b->bd_info->bb_fragments--;
1281                :
1282                :         /* let's maintain buddy itself */
1283       78043041 :         while (len) {
1284       64762363 :                 ord = mb_find_order_for_block(e4b, start);
1285                :
1286       64762510 :                 if (((start >> ord) << ord) == start && len >= (1 << ord)) {
1287                :                         /* the whole chunk may be allocated at once! */
1288       37283538 :                         mlen = 1 << ord;
1289       37283538 :                         buddy = mb_find_buddy(e4b, ord, &max);
1290       37283548 :                         BUG_ON((start >> ord) >= max);
1291       37283609 :                         mb_set_bit(start >> ord, buddy);
1292       37283609 :                         e4b->bd_info->bb_counters[ord]--;
1293       37283609 :                         start += mlen;
1294       37283609 :                         len -= mlen;
1295       37283609 :                         BUG_ON(len < 0);
1296                :                         continue;
1297                :                 }
1298                :
1299                :                 /* store for history */
1300       27478972 :                 if (ret == 0)
1301        6023868 :                         ret = len | (ord << 16);
1302                :
1303                :                 /* we have to split large buddy */
1304       27478972 :                 BUG_ON(ord <= 0);
1305       27478922 :                 buddy = mb_find_buddy(e4b, ord, &max);
1306       27478948 :                 mb_set_bit(start >> ord, buddy);
1307       27478948 :                 e4b->bd_info->bb_counters[ord]--;
1308                :
1309       27478948 :                 ord--;
1310       27478948 :                 cur = (start >> ord) & ~1U;
1311       27478948 :                 buddy = mb_find_buddy(e4b, ord, &max);
1312       27478940 :                 mb_clear_bit(cur, buddy);
1313       27478932 :                 mb_clear_bit(cur + 1, buddy);
1314       27478937 :                 e4b->bd_info->bb_counters[ord]++;
1315       27478937 :                 e4b->bd_info->bb_counters[ord]++;
1316                :         }
1317                :
1318       13280678 :         mb_set_bits(EXT4_MB_BITMAP(e4b), ex->fe_start, len0);
1319                :         mb_check_buddy(e4b);
1320                :
1321       13280701 :         return ret;
1322                : }
1323                :
1324                : /*
1325                :  * Must be called under group lock!
1326                :  */
1327                : static void ext4_mb_use_best_found(struct ext4_allocation_context *ac,
1328                :                                    struct ext4_buddy *e4b)
1329       13280395 : {
1330       26560790 :         struct ext4_sb_info *sbi = EXT4_SB(ac->ac_sb);
1331                :         int ret;
1332                :
1333       13280395 :         BUG_ON(ac->ac_b_ex.fe_group != e4b->bd_group);
1334       13280444 :         BUG_ON(ac->ac_status == AC_STATUS_FOUND);
1335                :
1336       13280404 :         ac->ac_b_ex.fe_len = min(ac->ac_b_ex.fe_len, ac->ac_g_ex.fe_len);
```

```
1337       13280404 :          ac->ac_b_ex.fe_logical = ac->ac_g_ex.fe_logical;
1338       13280404 :          ret = mb_mark_used(e4b, &ac->ac_b_ex);
1339                :
1340                :          /* preallocation can change ac_b_ex, thus we store actually
1341                :           * allocated blocks for history */
1342       13280705 :          ac->ac_f_ex = ac->ac_b_ex;
1343                :
1344       13280705 :          ac->ac_status = AC_STATUS_FOUND;
1345       13280705 :          ac->ac_tail = ret & 0xffff;
1346       13280705 :          ac->ac_buddy = ret >> 16;
1347                :
1348                :          /*
1349                :           * take the page reference. We want the page to be pinned
1350                :           * so that we don't get a ext4_mb_init_cache_call for this
1351                :           * group until we update the bitmap. That would mean we
1352                :           * double allocate blocks. The reference is dropped
1353                :           * in ext4_mb_release_context
1354                :           */
1355       13280705 :          ac->ac_bitmap_page = e4b->bd_bitmap_page;
1356       13280705 :          get_page(ac->ac_bitmap_page);
1357       13280717 :          ac->ac_buddy_page = e4b->bd_buddy_page;
1358       13280717 :          get_page(ac->ac_buddy_page);
1359                :          /* on allocation we use ac to track the held semaphore */
1360       13280690 :          ac->alloc_semp =  e4b->alloc_semp;
1361       13280690 :          e4b->alloc_semp = NULL;
1362                :          /* store last allocated for subsequent stream allocation */
1363       13280690 :          if ((ac->ac_flags & EXT4_MB_HINT_DATA)) {
1364       11391802 :                  spin_lock(&sbi->s_md_lock);
1365       11391833 :                  sbi->s_mb_last_group = ac->ac_f_ex.fe_group;
1366       11391833 :                  sbi->s_mb_last_start = ac->ac_f_ex.fe_start;
1367       11391833 :                  spin_unlock(&sbi->s_md_lock);
1368                :          }
1369       13280645 : }
1370                :
1371                : /*
1372                :  * regular allocator, for general purposes allocation
1373                :  */
1374                :
1375                : static void ext4_mb_check_limits(struct ext4_allocation_context *ac,
1376                :                                  struct ext4_buddy *e4b,
1377                :                                  int finish_group)
1378      458292141 : {
1379      916584282 :          struct ext4_sb_info *sbi = EXT4_SB(ac->ac_sb);
1380      458292141 :          struct ext4_free_extent *bex = &ac->ac_b_ex;
1381      458292141 :          struct ext4_free_extent *gex = &ac->ac_g_ex;
1382                :          struct ext4_free_extent ex;
1383                :          int max;
1384                :
1385      458292141 :          if (ac->ac_status == AC_STATUS_FOUND)
1386        4152604 :                  return;
1387                :          /*
1388                :           * We don't want to scan for a whole year
1389                :           */
1390      454139537 :          if (ac->ac_found > sbi->s_mb_max_to_scan &&
1391                :                          !(ac->ac_flags & EXT4_MB_HINT_FIRST)) {
1392        3336109 :                  ac->ac_status = AC_STATUS_BREAK;
1393        3336109 :                  return;
1394                :          }
1395                :
1396                :          /*
1397                :           * Haven't found good chunk so far, let's continue
1398                :           */
1399      450803428 :          if (bex->fe_len < gex->fe_len)
1400      422245635 :                  return;
1401                :
1402       28557793 :          if ((finish_group || ac->ac_found > sbi->s_mb_min_to_scan)
1403                :                          && bex->fe_group == e4b->bd_group) {
1404                :                  /* recheck chunk's availability - we don't know
1405                :                   * when it was found (within this lock-unlock
1406                :                   * period or not) */
1407        4481434 :                  max = mb_find_extent(e4b, 0, bex->fe_start, gex->fe_len, &ex);
```

```
1408    4481435 :                 if (max >= gex->fe_len) {
1409    4481435 :                         ext4_mb_use_best_found(ac, e4b);
1410    4481434 :                         return;
1411            :                 }
1412            :         }
1413            : }
1414            :
1415            : /*
1416            :  * The routine checks whether found extent is good enough. If it is,
1417            :  * then the extent gets marked used and flag is set to the context
1418            :  * to stop scanning. Otherwise, the extent is compared with the
1419            :  * previous found extent and if new one is better, then it's stored
1420            :  * in the context. Later, the best found extent will be used, if
1421            :  * mballoc can't find good enough extent.
1422            :  *
1423            :  * FIXME: real allocation policy is to be designed yet!
1424            :  */
1425            : static void ext4_mb_measure_extent(struct ext4_allocation_context *ac,
1426            :                                    struct ext4_free_extent *ex,
1427            :                                    struct ext4_buddy *e4b)
1428  454010169 : {
1429  454010169 :         struct ext4_free_extent *bex = &ac->ac_b_ex;
1430  454010169 :         struct ext4_free_extent *gex = &ac->ac_g_ex;
1431            :
1432  454010169 :         BUG_ON(ex->fe_len <= 0);
1433  908032000 :         BUG_ON(ex->fe_len > EXT4_BLOCKS_PER_GROUP(ac->ac_sb));
1434  907990972 :         BUG_ON(ex->fe_start >= EXT4_BLOCKS_PER_GROUP(ac->ac_sb));
1435  454002303 :         BUG_ON(ac->ac_status != AC_STATUS_CONTINUE);
1436            :
1437  454015528 :         ac->ac_found++;
1438            :
1439            :         /*
1440            :          * The special case - take what you catch first
1441            :          */
1442  454015528 :         if (unlikely(ac->ac_flags & EXT4_MB_HINT_FIRST)) {
1443      18457 :                 *bex = *ex;
1444      18457 :                 ext4_mb_use_best_found(ac, e4b);
1445      18459 :                 return;
1446            :         }
1447            :
1448            :         /*
1449            :          * Let's check whether the chuck is good enough
1450            :          */
1451  453968944 :         if (ex->fe_len == gex->fe_len) {
1452    1758544 :                 *bex = *ex;
1453    1758544 :                 ext4_mb_use_best_found(ac, e4b);
1454    1758545 :                 return;
1455            :         }
1456            :
1457            :         /*
1458            :          * If this is first found extent, just store it in the context
1459            :          */
1460  452210400 :         if (bex->fe_len == 0) {
1461    7081433 :                 *bex = *ex;
1462    7081433 :                 return;
1463            :         }
1464            :
1465            :         /*
1466            :          * If new found extent is better, store it in the context
1467            :          */
1468  445128967 :         if (bex->fe_len < gex->fe_len) {
1469            :                 /* if the request isn't satisfied, any found extent
1470            :                  * larger than previous best one is better */
1471  420704159 :                 if (ex->fe_len > bex->fe_len)
1472   16414640 :                         *bex = *ex;
1473   24424808 :         } else if (ex->fe_len > gex->fe_len) {
1474            :                 /* if the request is satisfied, then we try to find
1475            :                  * an extent that still satisfy the request, but is
1476            :                  * smaller than previous one */
1477   22836055 :                 if (ex->fe_len < bex->fe_len)
1478    1322411 :                         *bex = *ex;
```

```
1479                :         }
1480                :
1481       445128967 :         ext4_mb_check_limits(ac, e4b, 0);
1482                : }
1483                :
1484                : static noinline_for_stack
1485                : int ext4_mb_try_best_found(struct ext4_allocation_context *ac,
1486                :                                           struct ext4_buddy *e4b)
1487         1668458 : {
1488         1668458 :         struct ext4_free_extent ex = ac->ac_b_ex;
1489         1668458 :         ext4_group_t group = ex.fe_group;
1490                 :         int max;
1491                 :         int err;
1492                 :
1493         1668458 :         BUG_ON(ex.fe_len <= 0);
1494         1668492 :         err = ext4_mb_load_buddy(ac->ac_sb, group, e4b);
1495         1668504 :         if (err)
1496               0 :                 return err;
1497                 :
1498         1668504 :         ext4_lock_group(ac->ac_sb, group);
1499         1668492 :         max = mb_find_extent(e4b, 0, ex.fe_start, ex.fe_len, &ex);
1500                 :
1501         1668505 :         if (max > 0) {
1502         1650023 :                 ac->ac_b_ex = ex;
1503         1650023 :                 ext4_mb_use_best_found(ac, e4b);
1504                 :         }
1505                 :
1506         1668529 :         ext4_unlock_group(ac->ac_sb, group);
1507         1668508 :         ext4_mb_release_desc(e4b);
1508                 :
1509         1668492 :         return 0;
1510                : }
1511                :
1512                : static noinline_for_stack
1513                : int ext4_mb_find_by_goal(struct ext4_allocation_context *ac,
1514                :                                  struct ext4_buddy *e4b)
1515        13272015 : {
1516        13272015 :         ext4_group_t group = ac->ac_g_ex.fe_group;
1517                 :         int max;
1518                 :         int err;
1519        26544030 :         struct ext4_sb_info *sbi = EXT4_SB(ac->ac_sb);
1520        13272015 :         struct ext4_super_block *es = sbi->s_es;
1521                 :         struct ext4_free_extent ex;
1522                 :
1523        13272015 :         if (!(ac->ac_flags & EXT4_MB_HINT_TRY_GOAL))
1524         4912977 :                 return 0;
1525                 :
1526         8359038 :         err = ext4_mb_load_buddy(ac->ac_sb, group, e4b);
1527         8364890 :         if (err)
1528               0 :                 return err;
1529                 :
1530         8364890 :         ext4_lock_group(ac->ac_sb, group);
1531         8363701 :         max = mb_find_extent(e4b, 0, ac->ac_g_ex.fe_start,
1532                 :                              ac->ac_g_ex.fe_len, &ex);
1533                 :
1534         8365433 :         if (max >= ac->ac_g_ex.fe_len && ac->ac_g_ex.fe_len == sbi->s_stripe) {
1535                 :                 ext4_fsblk_t start;
1536                 :
1537               0 :                 start = (e4b->bd_group * EXT4_BLOCKS_PER_GROUP(ac->ac_sb)) +
1538                 :                         ex.fe_start + le32_to_cpu(es->s_first_data_block);
1539                 :                 /* use do_div to get remainder (would be 64-bit modulo) */
1540               0 :                 if (do_div(start, sbi->s_stripe) == 0) {
1541               0 :                         ac->ac_found++;
1542               0 :                         ac->ac_b_ex = ex;
1543               0 :                         ext4_mb_use_best_found(ac, e4b);
1544                 :                 }
1545         8365433 :         } else if (max >= ac->ac_g_ex.fe_len) {
1546          774778 :                 BUG_ON(ex.fe_len <= 0);
1547          774751 :                 BUG_ON(ex.fe_group != ac->ac_g_ex.fe_group);
1548          774756 :                 BUG_ON(ex.fe_start != ac->ac_g_ex.fe_start);
1549          774772 :                 ac->ac_found++;
```

```
1550   774772 :                 ac->ac_b_ex = ex;
1551   774772 :                 ext4_mb_use_best_found(ac, e4b);
1552  7590655 :         } else if (max > 0 && (ac->ac_flags & EXT4_MB_HINT_MERGE)) {
1553          :                 /* Sometimes, caller may want to merge even small
1554          :                  * number of blocks to an existing extent */
1555        0 :                 BUG_ON(ex.fe_len <= 0);
1556        0 :                 BUG_ON(ex.fe_group != ac->ac_g_ex.fe_group);
1557        0 :                 BUG_ON(ex.fe_start != ac->ac_g_ex.fe_start);
1558        0 :                 ac->ac_found++;
1559        0 :                 ac->ac_b_ex = ex;
1560        0 :                 ext4_mb_use_best_found(ac, e4b);
1561          :         }
1562  8365383 :         ext4_unlock_group(ac->ac_sb, group);
1563  8364374 :         ext4_mb_release_desc(e4b);
1564          :
1565  8365798 :         return 0;
1566          : }
1567          :
1568          : /*
1569          :  * The routine scans buddy structures (not bitmap!) from given order
1570          :  * to max order and tries to find big enough chunk to satisfy the req
1571          :  */
1572          : static noinline_for_stack
1573          : void ext4_mb_simple_scan_group(struct ext4_allocation_context *ac,
1574          :                                struct ext4_buddy *e4b)
1575  4597269 : {
1576  4597269 :         struct super_block *sb = ac->ac_sb;
1577  4597269 :         struct ext4_group_info *grp = e4b->bd_info;
1578          :         void *buddy;
1579          :         int i;
1580          :         int k;
1581          :         int max;
1582          :
1583  4597269 :         BUG_ON(ac->ac_2order <= 0);
1584  6820937 :         for (i = ac->ac_2order; i <= sb->s_blocksize_bits + 1; i++) {
1585  6820907 :                 if (grp->bb_counters[i] == 0)
1586  2223666 :                         continue;
1587          :
1588  4597241 :                 buddy = mb_find_buddy(e4b, i, &max);
1589  4597384 :                 BUG_ON(buddy == NULL);
1590          :
1591  4597438 :                 k = mb_find_next_zero_bit(buddy, max, 0);
1592  4597452 :                 BUG_ON(k >= max);
1593          :
1594  4597452 :                 ac->ac_found++;
1595          :
1596  4597452 :                 ac->ac_b_ex.fe_len = 1 << i;
1597  4597452 :                 ac->ac_b_ex.fe_start = k << i;
1598  4597452 :                 ac->ac_b_ex.fe_group = e4b->bd_group;
1599          :
1600  4597452 :                 ext4_mb_use_best_found(ac, e4b);
1601          :
1602  4597384 :                 BUG_ON(ac->ac_b_ex.fe_len != ac->ac_g_ex.fe_len);
1603          :
1604  4597397 :                 if (EXT4_SB(sb)->s_mb_stats)
1605  4597426 :                         atomic_inc(&EXT4_SB(sb)->s_bal_2orders);
1606          :
1607          :                 break;
1608          :         }
1609  4597440 : }
1610          :
1611          : /*
1612          :  * The routine scans the group and measures all found extents.
1613          :  * In order to optimize scanning, caller must pass number of
1614          :  * free blocks in the group, so the routine can know upper limit.
1615          :  */
1616          : static noinline_for_stack
1617          : void ext4_mb_complex_scan_group(struct ext4_allocation_context *ac,
1618          :                                 struct ext4_buddy *e4b)
1619 13169263 : {
1620 13169263 :         struct super_block *sb = ac->ac_sb;
```

```
1621  13169263 :          void *bitmap = EXT4_MB_BITMAP(e4b);
1622           :          struct ext4_free_extent ex;
1623           :          int i;
1624           :          int free;
1625           :
1626  13169263 :          free = e4b->bd_info->bb_free;
1627  13169263 :          BUG_ON(free <= 0);
1628           :
1629  13169287 :          i = e4b->bd_info->bb_first_free;
1630           :
1631 480296015 :          while (free && ac->ac_status == AC_STATUS_CONTINUE) {
1632 453912797 :                  i = mb_find_next_zero_bit(bitmap,
1633           :                                                  EXT4_BLOCKS_PER_GROUP(sb), i);
1634 908000398 :                  if (i >= EXT4_BLOCKS_PER_GROUP(sb)) {
1635           :                          /*
1636           :                           * IF we have corrupt bitmap, we won't find any
1637           :                           * free blocks even though group info says we
1638           :                           * we have free blocks
1639           :                           */
1640         0 :                          ext4_grp_locked_error(sb, e4b->bd_group,
1641           :                                                __func__, "%d free blocks as per "
1642           :                                                "group info. But bitmap says 0",
1643           :                                                free);
1644         0 :                          break;
1645           :                  }
1646           :
1647 454000199 :                  mb_find_extent(e4b, 0, i, ac->ac_g_ex.fe_len, &ex);
1648 454009658 :                  BUG_ON(ex.fe_len <= 0);
1649 454019543 :                  if (free < ex.fe_len) {
1650         0 :                          ext4_grp_locked_error(sb, e4b->bd_group,
1651           :                                                __func__, "%d free blocks as per "
1652           :                                                "group info. But got %d blocks",
1653           :                                                free, ex.fe_len);
1654           :                          /*
1655           :                           * The number of free blocks differs. This mostly
1656           :                           * indicate that the bitmap is corrupt. So exit
1657           :                           * without claiming the space.
1658           :                           */
1659         0 :                          break;
1660           :                  }
1661           :
1662 454019543 :                  ext4_mb_measure_extent(ac, &ex, e4b);
1663           :
1664 453957441 :                  i += ex.fe_len;
1665 453957441 :                  free -= ex.fe_len;
1666           :          }
1667           :
1668  13213931 :          ext4_mb_check_limits(ac, e4b, 1);
1669  13171183 : }
1670           :
1671           : /*
1672           :  * This is a special case for storages like raid5
1673           :  * we try to find stripe-aligned chunks for stripe-size requests
1674           :  * XXX should do so at least for multiples of stripe size as well
1675           :  */
1676           : static noinline_for_stack
1677           : void ext4_mb_scan_aligned(struct ext4_allocation_context *ac,
1678           :                           struct ext4_buddy *e4b)
1679         0 : {
1680         0 :          struct super_block *sb = ac->ac_sb;
1681         0 :          struct ext4_sb_info *sbi = EXT4_SB(sb);
1682         0 :          void *bitmap = EXT4_MB_BITMAP(e4b);
1683           :          struct ext4_free_extent ex;
1684           :          ext4_fsblk_t first_group_block;
1685           :          ext4_fsblk_t a;
1686           :          ext4_grpblk_t i;
1687           :          int max;
1688           :
1689         0 :          BUG_ON(sbi->s_stripe == 0);
1690           :
1691           :          /* find first stripe-aligned block in group */
```

```
1692                0 :          first_group_block = e4b->bd_group * EXT4_BLOCKS_PER_GROUP(sb)
1693                  :                  + le32_to_cpu(sbi->s_es->s_first_data_block);
1694                0 :          a = first_group_block + sbi->s_stripe - 1;
1695                0 :          do_div(a, sbi->s_stripe);
1696                0 :          i = (a * sbi->s_stripe) - first_group_block;
1697                  :
1698                0 :          while (i < EXT4_BLOCKS_PER_GROUP(sb)) {
1699                0 :                  if (!mb_test_bit(i, bitmap)) {
1700                0 :                          max = mb_find_extent(e4b, 0, i, sbi->s_stripe, &ex);
1701                0 :                          if (max >= sbi->s_stripe) {
1702                0 :                                  ac->ac_found++;
1703                0 :                                  ac->ac_b_ex = ex;
1704                0 :                                  ext4_mb_use_best_found(ac, e4b);
1705                0 :                                  break;
1706                  :                          }
1707                  :                  }
1708                0 :                  i += sbi->s_stripe;
1709                  :          }
1710                0 : }
1711                  :
1712                  : static int ext4_mb_good_group(struct ext4_allocation_context *ac,
1713                  :                               ext4_group_t group, int cr)
1714               -1 : {
1715                  :          unsigned free, fragments;
1716                  :          unsigned i, bits;
1717               -1 :          int flex_size = ext4_flex_bg_size(EXT4_SB(ac->ac_sb));
1718               -1 :          struct ext4_group_info *grp = ext4_get_group_info(ac->ac_sb, group);
1719                  :
1720               -1 :          BUG_ON(cr < 0 || cr >= 4);
1721               -1 :          BUG_ON(EXT4_MB_GRP_NEED_INIT(grp));
1722                  :
1723               -1 :          free = grp->bb_free;
1724               -1 :          fragments = grp->bb_fragments;
1725               -1 :          if (free == 0)
1726              814 :                  return 0;
1727               -1 :          if (fragments == 0)
1728              175 :                  return 0;
1729                  :
1730               -1 :          switch (cr) {
1731                  :          case 0:
1732               -1 :                  BUG_ON(ac->ac_2order == 0);
1733                  :
1734                  :                  /* Avoid using the first bg of a flexgroup for data files */
1735               -1 :                  if ((ac->ac_flags & EXT4_MB_HINT_DATA) &&
1736                  :                      (flex_size >= EXT4_FLEX_SIZE_DIR_ALLOC_SCHEME) &&
1737                  :                      ((group % flex_size) == 0))
1738       367222264 :                          return 0;
1739                  :
1740               -1 :                  bits = ac->ac_sb->s_blocksize_bits + 1;
1741               -1 :                  for (i = ac->ac_2order; i <= bits; i++)
1742               -1 :                          if (grp->bb_counters[i] > 0)
1743                0 :                                  return 1;
1744                  :                  break;
1745                  :          case 1:
1746               -1 :                  if ((free / fragments) >= ac->ac_g_ex.fe_len)
1747        12935417 :                          return 1;
1748                  :                  break;
1749                  :          case 2:
1750       801846935 :                  if (free >= ac->ac_g_ex.fe_len)
1751         8749875 :                          return 1;
1752                  :                  break;
1753                  :          case 3:
1754         4646264 :                  return 1;
1755                  :          default:
1756                0 :                  BUG();
1757                  :          }
1758                  :
1759               -1 :          return 0;
1760                  : }
1761                  :
1762                  : /*
```

```
1763                :  * lock the group_info alloc_sem of all the groups
1764                :  * belonging to the same buddy cache page. This
1765                :  * make sure other parallel operation on the buddy
1766                :  * cache doesn't happen  whild holding the buddy cache
1767                :  * lock
1768                :  */
1769                : int ext4_mb_get_buddy_cache_lock(struct super_block *sb, ext4_group_t group)
1770          29874 : {
1771                :         int i;
1772                :         int block, pnum;
1773                :         int blocks_per_page;
1774                :         int groups_per_page;
1775          29874 :         ext4_group_t ngroups = ext4_get_groups_count(sb);
1776                :         ext4_group_t first_group;
1777                :         struct ext4_group_info *grp;
1778                :
1779          29874 :         blocks_per_page = PAGE_CACHE_SIZE / sb->s_blocksize;
1780                :         /*
1781                :          * the buddy cache inode stores the block bitmap
1782                :          * and buddy information in consecutive blocks.
1783                :          * So for each group we need two blocks.
1784                :          */
1785          29874 :         block = group * 2;
1786          29874 :         pnum = block / blocks_per_page;
1787          29874 :         first_group = pnum * blocks_per_page / 2;
1788                :
1789          29874 :         groups_per_page = blocks_per_page >> 1;
1790          29874 :         if (groups_per_page == 0)
1791          18721 :                 groups_per_page = 1;
1792                :         /* read all groups the page covers into the cache */
1793          66813 :         for (i = 0; i < groups_per_page; i++) {
1794                :
1795          36939 :                 if ((first_group + i) >= ngroups)
1796              0 :                         break;
1797          73878 :                 grp = ext4_get_group_info(sb, first_group + i);
1798                :                 /* take all groups write allocation
1799                :                  * semaphore. This make sure there is
1800                :                  * no block allocation going on in any
1801                :                  * of that groups
1802                :                  */
1803          36939 :                 down_write_nested(&grp->alloc_sem, i);
1804                :         }
1805          29874 :         return i;
1806                : }
1807                :
1808                : void ext4_mb_put_buddy_cache_lock(struct super_block *sb,
1809                :                                   ext4_group_t group, int locked_group)
1810          29874 : {
1811                :         int i;
1812                :         int block, pnum;
1813                :         int blocks_per_page;
1814                :         ext4_group_t first_group;
1815                :         struct ext4_group_info *grp;
1816                :
1817          29874 :         blocks_per_page = PAGE_CACHE_SIZE / sb->s_blocksize;
1818                :         /*
1819                :          * the buddy cache inode stores the block bitmap
1820                :          * and buddy information in consecutive blocks.
1821                :          * So for each group we need two blocks.
1822                :          */
1823          29874 :         block = group * 2;
1824          29874 :         pnum = block / blocks_per_page;
1825          29874 :         first_group = pnum * blocks_per_page / 2;
1826                :         /* release locks on all the groups */
1827          66813 :         for (i = 0; i < locked_group; i++) {
1828                :
1829          73878 :                 grp = ext4_get_group_info(sb, first_group + i);
1830                :                 /* take all groups write allocation
1831                :                  * semaphore. This make sure there is
1832                :                  * no block allocation going on in any
1833                :                  * of that groups
```

```
1834                :                  */
1835          36939 :                 up_write(&grp->alloc_sem);
1836                :         }
1837                :
1838          29874 : }
1839                :
1840                : static noinline_for_stack
1841                : int ext4_mb_init_group(struct super_block *sb, ext4_group_t group)
1842          29874 : {
1843                :
1844                :         int ret;
1845                :         void *bitmap;
1846                :         int blocks_per_page;
1847                :         int block, pnum, poff;
1848          29874 :         int num_grp_locked = 0;
1849                :         struct ext4_group_info *this_grp;
1850          29874 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
1851          29874 :         struct inode *inode = sbi->s_buddy_cache;
1852          29874 :         struct page *page = NULL, *bitmap_page = NULL;
1853                :
1854                :         mb_debug("init group %lu\n", group);
1855          29874 :         blocks_per_page = PAGE_CACHE_SIZE / sb->s_blocksize;
1856          29874 :         this_grp = ext4_get_group_info(sb, group);
1857                :         /*
1858                :          * This ensures we don't add group
1859                :          * to this buddy cache via resize
1860                :          */
1861          29874 :         num_grp_locked =  ext4_mb_get_buddy_cache_lock(sb, group);
1862          59748 :         if (!EXT4_MB_GRP_NEED_INIT(this_grp)) {
1863                :                 /*
1864                :                  * somebody initialized the group
1865                :                  * return without doing anything
1866                :                  */
1867            680 :                 ret = 0;
1868            680 :                 goto err;
1869                :         }
1870                :         /*
1871                :          * the buddy cache inode stores the block bitmap
1872                :          * and buddy information in consecutive blocks.
1873                :          * So for each group we need two blocks.
1874                :          */
1875          29194 :         block = group * 2;
1876          29194 :         pnum = block / blocks_per_page;
1877          29194 :         poff = block % blocks_per_page;
1878          29194 :         page = find_or_create_page(inode->i_mapping, pnum, GFP_NOFS);
1879          29194 :         if (page) {
1880          29194 :                 BUG_ON(page->mapping != inode->i_mapping);
1881          29194 :                 ret = ext4_mb_init_cache(page, NULL);
1882          29194 :                 if (ret) {
1883              0 :                         unlock_page(page);
1884              0 :                         goto err;
1885                :                 }
1886          29194 :                 unlock_page(page);
1887                :         }
1888          29194 :         if (page == NULL || !PageUptodate(page)) {
1889              0 :                 ret = -EIO;
1890              0 :                 goto err;
1891                :         }
1892          29194 :         mark_page_accessed(page);
1893          29194 :         bitmap_page = page;
1894          29194 :         bitmap = page_address(page) + (poff * sb->s_blocksize);
1895                :
1896                :         /* init buddy cache */
1897          29194 :         block++;
1898          29194 :         pnum = block / blocks_per_page;
1899          29194 :         poff = block % blocks_per_page;
1900          29194 :         page = find_or_create_page(inode->i_mapping, pnum, GFP_NOFS);
1901          29194 :         if (page == bitmap_page) {
1902                :                 /*
1903                :                  * If both the bitmap and buddy are in
1904                :                  * the same page we don't need to force
```

```
1905                :                  * init the buddy
1906                :                  */
1907            10650 :                 unlock_page(page);
1908            18544 :         } else if (page) {
1909            18544 :                 BUG_ON(page->mapping != inode->i_mapping);
1910            18544 :                 ret = ext4_mb_init_cache(page, bitmap);
1911            18544 :                 if (ret) {
1912                0 :                         unlock_page(page);
1913                0 :                         goto err;
1914                :                 }
1915            18544 :                 unlock_page(page);
1916                :         }
1917            29194 :         if (page == NULL || !PageUptodate(page)) {
1918                0 :                 ret = -EIO;
1919                0 :                 goto err;
1920                :         }
1921            29194 :         mark_page_accessed(page);
1922            29874 : err:
1923            29874 :         ext4_mb_put_buddy_cache_lock(sb, group, num_grp_locked);
1924            29874 :         if (bitmap_page)
1925            29194 :                 page_cache_release(bitmap_page);
1926            29874 :         if (page)
1927            29194 :                 page_cache_release(page);
1928            29874 :         return ret;
1929                : }
1930                :
1931                : static noinline_for_stack int
1932                : ext4_mb_regular_allocator(struct ext4_allocation_context *ac)
1933         13271609 : {
1934                :         ext4_group_t ngroups, group, i;
1935                :         int cr;
1936         13271609 :         int err = 0;
1937                :         int bsbits;
1938                :         struct ext4_sb_info *sbi;
1939                :         struct super_block *sb;
1940                :         struct ext4_buddy e4b;
1941                :         loff_t size, isize;
1942                :
1943         13271609 :         sb = ac->ac_sb;
1944         13271609 :         sbi = EXT4_SB(sb);
1945         13272657 :         ngroups = ext4_get_groups_count(sb);
1946         13272657 :         BUG_ON(ac->ac_status == AC_STATUS_FOUND);
1947                :
1948                :         /* first, try the goal */
1949         13278241 :         err = ext4_mb_find_by_goal(ac, &e4b);
1950         13275314 :         if (err || ac->ac_status == AC_STATUS_FOUND)
1951                :                 goto out;
1952                :
1953         12502522 :         if (unlikely(ac->ac_flags & EXT4_MB_HINT_GOAL_ONLY))
1954                0 :                 goto out;
1955                :
1956                :         /*
1957                :          * ac->ac2_order is set only if the fe_len is a power of 2
1958                :          * if ac2_order is set we also set criteria to 0 so that we
1959                :          * try exact allocation using buddy.
1960                :          */
1961         25016358 :         i = fls(ac->ac_g_ex.fe_len);
1962         12508179 :         ac->ac_2order = 0;
1963                :         /*
1964                :          * We search using buddy data only if the order of the request
1965                :          * is greater than equal to the sbi_s_mb_order2_reqs
1966                :          * You can tune it via /sys/fs/ext4/<partition>/mb_order2_req
1967                :          */
1968         12508179 :         if (i >= sbi->s_mb_order2_reqs) {
1969                :                 /*
1970                :                  * This should tell if fe_len is exactly power of 2
1971                :                  */
1972          9930200 :                 if ((ac->ac_g_ex.fe_len & (~(1 << (i - 1)))) == 0)
1973          4760728 :                         ac->ac_2order = i - 1;
1974                :         }
1975                :
```

```
1976   12508179 :          bsbits = ac->ac_sb->s_blocksize_bits;
1977            :          /* if stream allocation is enabled, use global goal */
1978   12508179 :          size = ac->ac_o_ex.fe_logical + ac->ac_o_ex.fe_len;
1979   25013891 :          isize = i_size_read(ac->ac_inode) >> bsbits;
1980   12505712 :          if (size < isize)
1981    1380978 :                  size = isize;
1982            :
1983   12505712 :          if (size < sbi->s_mb_stream_request &&
1984            :                          (ac->ac_flags & EXT4_MB_HINT_DATA)) {
1985            :                  /* TBD: may be hot point */
1986      68536 :                  spin_lock(&sbi->s_md_lock);
1987      68536 :                  ac->ac_g_ex.fe_group = sbi->s_mb_last_group;
1988      68536 :                  ac->ac_g_ex.fe_start = sbi->s_mb_last_start;
1989      68536 :                  spin_unlock(&sbi->s_md_lock);
1990            :          }
1991            :          /* Let's just scan groups to find more-less suitable blocks */
1992   12499241 :          cr = ac->ac_2order ? 0 : 1;
1993            :          /*
1994            :           * cr == 0 try to get exact allocation,
1995            :           * cr == 3  try to get anything
1996            :           */
1997            : repeat:
1998   14223400 :          for (; cr < 4 && ac->ac_status == AC_STATUS_CONTINUE; cr++) {
1999   14217166 :                  ac->ac_criteria = cr;
2000            :                  /*
2001            :                   * searching for the right group start
2002            :                   * from the goal value specified
2003            :                   */
2004   14217166 :                  group = ac->ac_g_ex.fe_group;
2005            :
2006          -1 :                  for (i = 0; i < ngroups; group++, i++) {
2007            :                          struct ext4_group_info *grp;
2008            :                          struct ext4_group_desc *desc;
2009            :
2010          -1 :                          if (group == ngroups)
2011    1746719 :                                  group = 0;
2012            :
2013            :                          /* quick check to skip empty groups */
2014          -1 :                          grp = ext4_get_group_info(sb, group);
2015          -1 :                          if (grp->bb_free == 0)
2016           0 :                                  continue;
2017            :
2018            :                          /*
2019            :                           * if the group is already init we check whether it is
2020            :                           * a good group and if not we don't load the buddy
2021            :                           */
2022          -1 :                          if (EXT4_MB_GRP_NEED_INIT(grp)) {
2023            :                                  /*
2024            :                                   * we need full data about the group
2025            :                                   * to make a good selection
2026            :                                   */
2027      29874 :                                  err = ext4_mb_init_group(sb, group);
2028      29874 :                                  if (err)
2029           0 :                                          goto out;
2030            :                          }
2031            :
2032            :                          /*
2033            :                           * If the particular group doesn't satisfy our
2034            :                           * criteria we continue with the next group
2035            :                           */
2036          -1 :                          if (!ext4_mb_good_group(ac, group, cr))
2037          -1 :                                  continue;
2038            :
2039   17767063 :                          err = ext4_mb_load_buddy(sb, group, &e4b);
2040   17767522 :                          if (err)
2041           0 :                                  goto out;
2042            :
2043            :                          ext4_lock_group(sb, group);
2044   17769475 :                          if (!ext4_mb_good_group(ac, group, cr)) {
2045            :                                  /* someone did allocation from this group */
2046            :                                  ext4_unlock_group(sb, group);
```

```
2047          2584 :                                ext4_mb_release_desc(&e4b);
2048          2584 :                                continue;
2049               :                        }
2050               :
2051      17766575 :                        ac->ac_groups_scanned++;
2052      17766575 :                        desc = ext4_get_group_desc(sb, group, NULL);
2053      17766937 :                        if (cr == 0)
2054       4597317 :                                ext4_mb_simple_scan_group(ac, &e4b);
2055      13169620 :                        else if (cr == 1 &&
2056               :                                 ac->ac_g_ex.fe_len == sbi->s_stripe)
2057             0 :                                ext4_mb_scan_aligned(ac, &e4b);
2058               :                        else
2059      13169620 :                                ext4_mb_complex_scan_group(ac, &e4b);
2060               :
2061               :                        ext4_unlock_group(sb, group);
2062      17766527 :                        ext4_mb_release_desc(&e4b);
2063               :
2064      17766007 :                        if (ac->ac_status != AC_STATUS_CONTINUE)
2065             0 :                                break;
2066               :                }
2067               :        }
2068               :
2069      12523905 :        if (ac->ac_b_ex.fe_len > 0 && ac->ac_status != AC_STATUS_FOUND &&
2070               :            !(ac->ac_flags & EXT4_MB_HINT_FIRST)) {
2071               :                /*
2072               :                 * We've been searching too long. Let's try to allocate
2073               :                 * the best chunk we've found so far
2074               :                 */
2075               :
2076       1668475 :                ext4_mb_try_best_found(ac, &e4b);
2077       1668111 :                if (ac->ac_status != AC_STATUS_FOUND) {
2078               :                        /*
2079               :                         * Someone more lucky has already allocated it.
2080               :                         * The only thing we can do is just take first
2081               :                         * found block(s)
2082               :                        printk(KERN_DEBUG "EXT4-fs: someone won our chunk\n");
2083               :                         */
2084         18344 :                        ac->ac_b_ex.fe_group = 0;
2085         18344 :                        ac->ac_b_ex.fe_start = 0;
2086         18344 :                        ac->ac_b_ex.fe_len = 0;
2087         18344 :                        ac->ac_status = AC_STATUS_CONTINUE;
2088         18344 :                        ac->ac_flags |= EXT4_MB_HINT_FIRST;
2089         18344 :                        cr = 3;
2090         18344 :                        atomic_inc(&sbi->s_mb_lost_chunks);
2091               :                        goto repeat;
2092               :                }
2093               :        }
2094      13277989 : out:
2095      13277989 :        return err;
2096               : }
2097               :
2098               : #ifdef EXT4_MB_HISTORY
2099               : struct ext4_mb_proc_session {
2100               :        struct ext4_mb_history *history;
2101               :        struct super_block *sb;
2102               :        int start;
2103               :        int max;
2104               : };
2105               :
2106               : static void *ext4_mb_history_skip_empty(struct ext4_mb_proc_session *s,
2107               :                                         struct ext4_mb_history *hs,
2108               :                                         int first)
2109             0 : {
2110             0 :        if (hs == s->history + s->max)
2111             0 :                hs = s->history;
2112             0 :        if (!first && hs == s->history + s->start)
2113             0 :                return NULL;
2114             0 :        while (hs->orig.fe_len == 0) {
2115             0 :                hs++;
2116             0 :                if (hs == s->history + s->max)
2117             0 :                        hs = s->history;
```

```
2118          0 :                 if (hs == s->history + s->start)
2119          0 :                         return NULL;
2120            :         }
2121          0 :         return hs;
2122            : }
2123            :
2124            : static void *ext4_mb_seq_history_start(struct seq_file *seq, loff_t *pos)
2125          0 : {
2126          0 :         struct ext4_mb_proc_session *s = seq->private;
2127            :         struct ext4_mb_history *hs;
2128          0 :         int l = *pos;
2129            :
2130          0 :         if (l == 0)
2131          0 :                 return SEQ_START_TOKEN;
2132          0 :         hs = ext4_mb_history_skip_empty(s, s->history + s->start, 1);
2133          0 :         if (!hs)
2134          0 :                 return NULL;
2135          0 :         while (--l && (hs = ext4_mb_history_skip_empty(s, ++hs, 0)) != NULL);
2136          0 :         return hs;
2137            : }
2138            :
2139            : static void *ext4_mb_seq_history_next(struct seq_file *seq, void *v,
2140            :                                       loff_t *pos)
2141          0 : {
2142          0 :         struct ext4_mb_proc_session *s = seq->private;
2143          0 :         struct ext4_mb_history *hs = v;
2144            :
2145          0 :         ++*pos;
2146          0 :         if (v == SEQ_START_TOKEN)
2147          0 :                 return ext4_mb_history_skip_empty(s, s->history + s->start, 1);
2148            :         else
2149          0 :                 return ext4_mb_history_skip_empty(s, ++hs, 0);
2150            : }
2151            :
2152            : static int ext4_mb_seq_history_show(struct seq_file *seq, void *v)
2153          0 : {
2154            :         char buf[25], buf2[25], buf3[25], *fmt;
2155          0 :         struct ext4_mb_history *hs = v;
2156            :
2157          0 :         if (v == SEQ_START_TOKEN) {
2158          0 :                 seq_printf(seq, "%-5s %-8s %-23s %-23s %-23s %-5s "
2159            :                                 "%-5s %-2s %-5s %-5s %-5s %-6s\n",
2160            :                            "pid", "inode", "original", "goal", "result", "found",
2161            :                             "grps", "cr", "flags", "merge", "tail", "broken");
2162          0 :                 return 0;
2163            :         }
2164            :
2165          0 :         if (hs->op == EXT4_MB_HISTORY_ALLOC) {
2166          0 :                 fmt = "%-5u %-8u %-23s %-23s %-23s %-5u %-5u %-2u "
2167            :                         "%-5u %-5s %-5u %-6u\n";
2168          0 :                 sprintf(buf2, "%u/%d/%u@%u", hs->result.fe_group,
2169            :                         hs->result.fe_start, hs->result.fe_len,
2170            :                         hs->result.fe_logical);
2171          0 :                 sprintf(buf, "%u/%d/%u@%u", hs->orig.fe_group,
2172            :                         hs->orig.fe_start, hs->orig.fe_len,
2173            :                         hs->orig.fe_logical);
2174          0 :                 sprintf(buf3, "%u/%d/%u@%u", hs->goal.fe_group,
2175            :                         hs->goal.fe_start, hs->goal.fe_len,
2176            :                         hs->goal.fe_logical);
2177          0 :                 seq_printf(seq, fmt, hs->pid, hs->ino, buf, buf3, buf2,
2178            :                                 hs->found, hs->groups, hs->cr, hs->flags,
2179            :                                 hs->merged ? "M" : "", hs->tail,
2180            :                                 hs->buddy ? 1 << hs->buddy : 0);
2181          0 :         } else if (hs->op == EXT4_MB_HISTORY_PREALLOC) {
2182          0 :                 fmt = "%-5u %-8u %-23s %-23s %-23s\n";
2183          0 :                 sprintf(buf2, "%u/%d/%u@%u", hs->result.fe_group,
2184            :                         hs->result.fe_start, hs->result.fe_len,
2185            :                         hs->result.fe_logical);
2186          0 :                 sprintf(buf, "%u/%d/%u@%u", hs->orig.fe_group,
2187            :                         hs->orig.fe_start, hs->orig.fe_len,
2188            :                         hs->orig.fe_logical);
```

```
2189                0 :                 seq_printf(seq, fmt, hs->pid, hs->ino, buf, "", buf2);
2190                0 :         } else if (hs->op == EXT4_MB_HISTORY_DISCARD) {
2191                0 :                 sprintf(buf2, "%u/%d/%u", hs->result.fe_group,
2192                  :                         hs->result.fe_start, hs->result.fe_len);
2193                0 :                 seq_printf(seq, "%-5u %-8u %-23s discard\n",
2194                  :                                     hs->pid, hs->ino, buf2);
2195                0 :         } else if (hs->op == EXT4_MB_HISTORY_FREE) {
2196                0 :                 sprintf(buf2, "%u/%d/%u", hs->result.fe_group,
2197                  :                         hs->result.fe_start, hs->result.fe_len);
2198                0 :                 seq_printf(seq, "%-5u %-8u %-23s free\n",
2199                  :                                     hs->pid, hs->ino, buf2);
2200                  :         }
2201                0 :         return 0;
2202                  : }
2203                  :
2204                  : static void ext4_mb_seq_history_stop(struct seq_file *seq, void *v)
2205                0 : {
2206                0 : }
2207                  :
2208                  : static struct seq_operations ext4_mb_seq_history_ops = {
2209                  :         .start  = ext4_mb_seq_history_start,
2210                  :         .next   = ext4_mb_seq_history_next,
2211                  :         .stop   = ext4_mb_seq_history_stop,
2212                  :         .show   = ext4_mb_seq_history_show,
2213                  : };
2214                  :
2215                  : static int ext4_mb_seq_history_open(struct inode *inode, struct file *file)
2216                0 : {
2217                0 :         struct super_block *sb = PDE(inode)->data;
2218                0 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2219                  :         struct ext4_mb_proc_session *s;
2220                  :         int rc;
2221                  :         int size;
2222                  :
2223                0 :         if (unlikely(sbi->s_mb_history == NULL))
2224                0 :                 return -ENOMEM;
2225                0 :         s = kmalloc(sizeof(*s), GFP_KERNEL);
2226                0 :         if (s == NULL)
2227                0 :                 return -ENOMEM;
2228                0 :         s->sb = sb;
2229                0 :         size = sizeof(struct ext4_mb_history) * sbi->s_mb_history_max;
2230                0 :         s->history = kmalloc(size, GFP_KERNEL);
2231                0 :         if (s->history == NULL) {
2232                0 :                 kfree(s);
2233                0 :                 return -ENOMEM;
2234                  :         }
2235                  :
2236                0 :         spin_lock(&sbi->s_mb_history_lock);
2237                0 :         memcpy(s->history, sbi->s_mb_history, size);
2238                0 :         s->max = sbi->s_mb_history_max;
2239                0 :         s->start = sbi->s_mb_history_cur % s->max;
2240                0 :         spin_unlock(&sbi->s_mb_history_lock);
2241                  :
2242                0 :         rc = seq_open(file, &ext4_mb_seq_history_ops);
2243                0 :         if (rc == 0) {
2244                0 :                 struct seq_file *m = (struct seq_file *)file->private_data;
2245                0 :                 m->private = s;
2246                  :         } else {
2247                0 :                 kfree(s->history);
2248                0 :                 kfree(s);
2249                  :         }
2250                0 :         return rc;
2251                  :
2252                  : }
2253                  :
2254                  : static int ext4_mb_seq_history_release(struct inode *inode, struct file *file)
2255                0 : {
2256                0 :         struct seq_file *seq = (struct seq_file *)file->private_data;
2257                0 :         struct ext4_mb_proc_session *s = seq->private;
2258                0 :         kfree(s->history);
2259                0 :         kfree(s);
```

```
2260                0 :         return seq_release(inode, file);
2261                  : }
2262                  :
2263                  : static ssize_t ext4_mb_seq_history_write(struct file *file,
2264                  :                                         const char __user *buffer,
2265                  :                                         size_t count, loff_t *ppos)
2266                0 : {
2267                0 :         struct seq_file *seq = (struct seq_file *)file->private_data;
2268                0 :         struct ext4_mb_proc_session *s = seq->private;
2269                0 :         struct super_block *sb = s->sb;
2270                  :         char str[32];
2271                  :         int value;
2272                  :
2273                0 :         if (count >= sizeof(str)) {
2274                0 :                 printk(KERN_ERR "EXT4-fs: %s string too long, max %u bytes\n",
2275                  :                                         "mb_history", (int)sizeof(str));
2276                0 :                 return -EOVERFLOW;
2277                  :         }
2278                  :
2279                0 :         if (copy_from_user(str, buffer, count))
2280                0 :                 return -EFAULT;
2281                  :
2282                0 :         value = simple_strtol(str, NULL, 0);
2283                0 :         if (value < 0)
2284                0 :                 return -ERANGE;
2285                0 :         EXT4_SB(sb)->s_mb_history_filter = value;
2286                  :
2287                0 :         return count;
2288                  : }
2289                  :
2290                  : static struct file_operations ext4_mb_seq_history_fops = {
2291                  :         .owner          = THIS_MODULE,
2292                  :         .open           = ext4_mb_seq_history_open,
2293                  :         .read           = seq_read,
2294                  :         .write          = ext4_mb_seq_history_write,
2295                  :         .llseek         = seq_lseek,
2296                  :         .release        = ext4_mb_seq_history_release,
2297                  : };
2298                  :
2299                  : static void *ext4_mb_seq_groups_start(struct seq_file *seq, loff_t *pos)
2300                0 : {
2301                0 :         struct super_block *sb = seq->private;
2302                  :         ext4_group_t group;
2303                  :
2304                0 :         if (*pos < 0 || *pos >= ext4_get_groups_count(sb))
2305                0 :                 return NULL;
2306                0 :         group = *pos + 1;
2307                0 :         return (void *) ((unsigned long) group);
2308                  : }
2309                  :
2310                  : static void *ext4_mb_seq_groups_next(struct seq_file *seq, void *v, loff_t *pos)
2311                0 : {
2312                0 :         struct super_block *sb = seq->private;
2313                  :         ext4_group_t group;
2314                  :
2315                0 :         ++*pos;
2316                0 :         if (*pos < 0 || *pos >= ext4_get_groups_count(sb))
2317                0 :                 return NULL;
2318                0 :         group = *pos + 1;
2319                0 :         return (void *) ((unsigned long) group);
2320                  : }
2321                  :
2322                  : static int ext4_mb_seq_groups_show(struct seq_file *seq, void *v)
2323                0 : {
2324                0 :         struct super_block *sb = seq->private;
2325                0 :         ext4_group_t group = (ext4_group_t) ((unsigned long) v);
2326                  :         int i;
2327                  :         int err;
2328                  :         struct ext4_buddy e4b;
2329                  :         struct sg {
2330                  :                 struct ext4_group_info info;
```

```
2331                :                 unsigned short counters[16];
2332                :         } sg;
2333                :
2334              0 :         group--;
2335              0 :         if (group == 0)
2336              0 :                 seq_printf(seq, "#%-5s: %-5s %-5s %-5s "
2337                :                                 "[ %-5s %-5s %-5s %-5s %-5s %-5s %-5s "
2338                :                                   "%-5s %-5s %-5s %-5s %-5s %-5s %-5s ]\n",
2339                :                            "group", "free", "frags", "first",
2340                :                            "2^0", "2^1", "2^2", "2^3", "2^4", "2^5", "2^6",
2341                :                            "2^7", "2^8", "2^9", "2^10", "2^11", "2^12", "2^13");
2342                :
2343              0 :         i = (sb->s_blocksize_bits + 2) * sizeof(sg.info.bb_counters[0]) +
2344                :                 sizeof(struct ext4_group_info);
2345              0 :         err = ext4_mb_load_buddy(sb, group, &e4b);
2346              0 :         if (err) {
2347              0 :                 seq_printf(seq, "#%-5u: I/O error\n", group);
2348              0 :                 return 0;
2349                :         }
2350                :         ext4_lock_group(sb, group);
2351              0 :         memcpy(&sg, ext4_get_group_info(sb, group), i);
2352                :         ext4_unlock_group(sb, group);
2353              0 :         ext4_mb_release_desc(&e4b);
2354                :
2355              0 :         seq_printf(seq, "#%-5u: %-5u %-5u %-5u [", group, sg.info.bb_free,
2356                :                         sg.info.bb_fragments, sg.info.bb_first_free);
2357              0 :         for (i = 0; i <= 13; i++)
2358              0 :                 seq_printf(seq, " %-5u", i <= sb->s_blocksize_bits + 1 ?
2359                :                                 sg.info.bb_counters[i] : 0);
2360              0 :         seq_printf(seq, " ]\n");
2361                :
2362              0 :         return 0;
2363                : }
2364                :
2365                : static void ext4_mb_seq_groups_stop(struct seq_file *seq, void *v)
2366              0 : {
2367              0 : }
2368                :
2369                : static struct seq_operations ext4_mb_seq_groups_ops = {
2370                :         .start  = ext4_mb_seq_groups_start,
2371                :         .next   = ext4_mb_seq_groups_next,
2372                :         .stop   = ext4_mb_seq_groups_stop,
2373                :         .show   = ext4_mb_seq_groups_show,
2374                : };
2375                :
2376                : static int ext4_mb_seq_groups_open(struct inode *inode, struct file *file)
2377              0 : {
2378              0 :         struct super_block *sb = PDE(inode)->data;
2379                :         int rc;
2380                :
2381              0 :         rc = seq_open(file, &ext4_mb_seq_groups_ops);
2382              0 :         if (rc == 0) {
2383              0 :                 struct seq_file *m = (struct seq_file *)file->private_data;
2384              0 :                 m->private = sb;
2385                :         }
2386              0 :         return rc;
2387                :
2388                : }
2389                :
2390                : static struct file_operations ext4_mb_seq_groups_fops = {
2391                :         .owner          = THIS_MODULE,
2392                :         .open           = ext4_mb_seq_groups_open,
2393                :         .read           = seq_read,
2394                :         .llseek         = seq_lseek,
2395                :         .release        = seq_release,
2396                : };
2397                :
2398                : static void ext4_mb_history_release(struct super_block *sb)
2399                : {
2400             94 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2401                :
```

```
2402              94 :         if (sbi->s_proc != NULL) {
2403              94 :                 remove_proc_entry("mb_groups", sbi->s_proc);
2404              94 :                 if (sbi->s_mb_history_max)
2405              94 :                         remove_proc_entry("mb_history", sbi->s_proc);
2406                 :         }
2407              94 :         kfree(sbi->s_mb_history);
2408                 : }
2409                 :
2410                 : static void ext4_mb_history_init(struct super_block *sb)
2411                 : {
2412              94 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2413                 :         int i;
2414                 :
2415              94 :         if (sbi->s_proc != NULL) {
2416              94 :                 if (sbi->s_mb_history_max)
2417              94 :                         proc_create_data("mb_history", S_IRUGO, sbi->s_proc,
2418                 :                                          &ext4_mb_seq_history_fops, sb);
2419              94 :                 proc_create_data("mb_groups", S_IRUGO, sbi->s_proc,
2420                 :                                  &ext4_mb_seq_groups_fops, sb);
2421                 :         }
2422                 :
2423              94 :         sbi->s_mb_history_cur = 0;
2424              94 :         spin_lock_init(&sbi->s_mb_history_lock);
2425              94 :         i = sbi->s_mb_history_max * sizeof(struct ext4_mb_history);
2426              94 :         sbi->s_mb_history = i ? kzalloc(i, GFP_KERNEL) : NULL;
2427                 :         /* if we can't allocate history, then we simple won't use it */
2428                 : }
2429                 :
2430                 : static noinline_for_stack void
2431                 : ext4_mb_store_history(struct ext4_allocation_context *ac)
2432       232424280 : {
2433       464848560 :         struct ext4_sb_info *sbi = EXT4_SB(ac->ac_sb);
2434                 :         struct ext4_mb_history h;
2435                 :
2436       232424280 :         if (sbi->s_mb_history == NULL)
2437               0 :                 return;
2438                 :
2439       232424280 :         if (!(ac->ac_op & sbi->s_mb_history_filter))
2440        11785369 :                 return;
2441                 :
2442       220638911 :         h.op = ac->ac_op;
2443       220638911 :         h.pid = current->pid;
2444       220638911 :         h.ino = ac->ac_inode ? ac->ac_inode->i_ino : 0;
2445       220638911 :         h.orig = ac->ac_o_ex;
2446       220638911 :         h.result = ac->ac_b_ex;
2447       220638911 :         h.flags = ac->ac_flags;
2448       220638911 :         h.found = ac->ac_found;
2449       220638911 :         h.groups = ac->ac_groups_scanned;
2450       220638911 :         h.cr = ac->ac_criteria;
2451       220638911 :         h.tail = ac->ac_tail;
2452       220638911 :         h.buddy = ac->ac_buddy;
2453       220638911 :         h.merged = 0;
2454       220638911 :         if (ac->ac_op == EXT4_MB_HISTORY_ALLOC) {
2455        13279531 :                 if (ac->ac_g_ex.fe_start == ac->ac_b_ex.fe_start &&
2456                 :                     ac->ac_g_ex.fe_group == ac->ac_b_ex.fe_group)
2457          838760 :                         h.merged = 1;
2458        13279531 :                 h.goal = ac->ac_g_ex;
2459        13279531 :                 h.result = ac->ac_f_ex;
2460                 :         }
2461                 :
2462       220638911 :         spin_lock(&sbi->s_mb_history_lock);
2463       222875670 :         memcpy(sbi->s_mb_history + sbi->s_mb_history_cur, &h, sizeof(h));
2464       222875670 :         if (++sbi->s_mb_history_cur >= sbi->s_mb_history_max)
2465          222837 :                 sbi->s_mb_history_cur = 0;
2466       222875670 :         spin_unlock(&sbi->s_mb_history_lock);
2467                 : }
2468                 :
2469                 : #else
2470                 : #define ext4_mb_history_release(sb)
2471                 : #define ext4_mb_history_init(sb)
2472                 : #endif
```

```
2473                :
2474                :
2475                : /* Create and initialize ext4_group_info data for the given group. */
2476                : int ext4_mb_add_groupinfo(struct super_block *sb, ext4_group_t group,
2477                :                           struct ext4_group_desc *desc)
2478          1719040 : {
2479                :         int i, len;
2480          1719040 :         int metalen = 0;
2481          1719040 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2482                :         struct ext4_group_info **meta_group_info;
2483                :
2484                :         /*
2485                :          * First check if this group is the first of a reserved block.
2486                :          * If it's true, we have to allocate a new table of pointers
2487                :          * to ext4_group_info structures
2488                :          */
2489          1719040 :         if (group % EXT4_DESC_PER_BLOCK(sb) == 0) {
2490            42708 :                 metalen = sizeof(*meta_group_info) <<
2491                :                         EXT4_DESC_PER_BLOCK_BITS(sb);
2492            85416 :                 meta_group_info = kmalloc(metalen, GFP_KERNEL);
2493            42708 :                 if (meta_group_info == NULL) {
2494                0 :                         printk(KERN_ERR "EXT4-fs: can't allocate mem for a "
2495                :                                "buddy group\n");
2496                0 :                         goto exit_meta_group_info;
2497                :                 }
2498            85416 :                 sbi->s_group_info[group >> EXT4_DESC_PER_BLOCK_BITS(sb)] =
2499                :                         meta_group_info;
2500                :         }
2501                :
2502                :         /*
2503                :          * calculate needed size. if change bb_counters size,
2504                :          * don't forget about ext4_mb_generate_buddy()
2505                :          */
2506          1719040 :         len = offsetof(typeof(**meta_group_info),
2507                :                        bb_counters[sb->s_blocksize_bits + 2]);
2508                :
2509          3438080 :         meta_group_info =
2510                :                 sbi->s_group_info[group >> EXT4_DESC_PER_BLOCK_BITS(sb)];
2511          1719040 :         i = group & (EXT4_DESC_PER_BLOCK(sb) - 1);
2512                :
2513          1719040 :         meta_group_info[i] = kzalloc(len, GFP_KERNEL);
2514          1719040 :         if (meta_group_info[i] == NULL) {
2515                0 :                 printk(KERN_ERR "EXT4-fs: can't allocate buddy mem\n");
2516                0 :                 goto exit_group_info;
2517                :         }
2518          1719040 :         set_bit(EXT4_GROUP_INFO_NEED_INIT_BIT,
2519                :                 &(meta_group_info[i]->bb_state));
2520                :
2521                :         /*
2522                :          * initialize bb_free to be able to skip
2523                :          * empty groups without initialization
2524                :          */
2525          1719040 :         if (desc->bg_flags & cpu_to_le16(EXT4_BG_BLOCK_UNINIT)) {
2526          1325447 :                 meta_group_info[i]->bb_free =
2527                :                         ext4_free_blocks_after_init(sb, group, desc);
2528                :         } else {
2529           393593 :                 meta_group_info[i]->bb_free =
2530                :                         ext4_free_blks_count(sb, desc);
2531                :         }
2532                :
2533          1719040 :         INIT_LIST_HEAD(&meta_group_info[i]->bb_prealloc_list);
2534          1719040 :         init_rwsem(&meta_group_info[i]->alloc_sem);
2535          1719040 :         meta_group_info[i]->bb_free_root.rb_node = NULL;;
2536                :
2537                : #ifdef DOUBLE_CHECK
2538                :         {
2539                :                 struct buffer_head *bh;
2540                :                 meta_group_info[i]->bb_bitmap =
2541                :                         kmalloc(sb->s_blocksize, GFP_KERNEL);
2542                :                 BUG_ON(meta_group_info[i]->bb_bitmap == NULL);
2543                :                 bh = ext4_read_block_bitmap(sb, group);
```

```
2544                :                 BUG_ON(bh == NULL);
2545                :                 memcpy(meta_group_info[i]->bb_bitmap, bh->b_data,
2546                :                        sb->s_blocksize);
2547                :                 put_bh(bh);
2548                :         }
2549                : #endif
2550                :
2551          1719040 :         return 0;
2552                :
2553                0 : exit_group_info:
2554                :         /* If a meta_group_info table has been allocated, release it now */
2555                0 :         if (group % EXT4_DESC_PER_BLOCK(sb) == 0)
2556                0 :                 kfree(sbi->s_group_info[group >> EXT4_DESC_PER_BLOCK_BITS(sb)]);
2557                0 : exit_meta_group_info:
2558                0 :         return -ENOMEM;
2559                : } /* ext4_mb_add_groupinfo */
2560                :
2561                : /*
2562                :  * Update an existing group.
2563                :  * This function is used for online resize
2564                :  */
2565                : void ext4_mb_update_group_info(struct ext4_group_info *grp, ext4_grpblk_t add)
2566                0 : {
2567                0 :         grp->bb_free += add;
2568                0 : }
2569                :
2570                : static int ext4_mb_init_backend(struct super_block *sb)
2571               94 : {
2572               94 :         ext4_group_t ngroups = ext4_get_groups_count(sb);
2573                :         ext4_group_t i;
2574                :         int metalen;
2575               94 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2576               94 :         struct ext4_super_block *es = sbi->s_es;
2577                :         int num_meta_group_infos;
2578                :         int num_meta_group_infos_max;
2579                :         int array_size;
2580                :         struct ext4_group_info **meta_group_info;
2581                :         struct ext4_group_desc *desc;
2582                :
2583                :         /* This is the number of blocks used by GDT */
2584              188 :         num_meta_group_infos = (ngroups + EXT4_DESC_PER_BLOCK(sb) -
2585                :                                 1) >> EXT4_DESC_PER_BLOCK_BITS(sb);
2586                :
2587                :         /*
2588                :          * This is the total number of blocks used by GDT including
2589                :          * the number of reserved blocks for GDT.
2590                :          * The s_group_info array is allocated with this value
2591                :          * to allow a clean online resize without a complex
2592                :          * manipulation of pointer.
2593                :          * The drawback is the unused memory when no resize
2594                :          * occurs but it's very low in terms of pages
2595                :          * (see comments below)
2596                :          * Need to handle this properly when META_BG resizing is allowed
2597                :          */
2598               94 :         num_meta_group_infos_max = num_meta_group_infos +
2599                :                                 le16_to_cpu(es->s_reserved_gdt_blocks);
2600                :
2601                :         /*
2602                :          * array_size is the size of s_group_info array. We round it
2603                :          * to the next power of two because this approximation is done
2604                :          * internally by kmalloc so we can have some more memory
2605                :          * for free here (e.g. may be used for META_BG resize).
2606                :          */
2607               94 :         array_size = 1;
2608             1344 :         while (array_size < sizeof(*sbi->s_group_info) *
2609                :                num_meta_group_infos_max)
2610             1156 :                 array_size = array_size << 1;
2611                :         /* An 8TB filesystem with 64-bit pointers requires a 4096 byte
2612                :          * kmalloc. A 128kb malloc should suffice for a 256TB filesystem.
2613                :          * So a two level scheme suffices for now. */
2614              188 :         sbi->s_group_info = kmalloc(array_size, GFP_KERNEL);
```

```
2615          94 :          if (sbi->s_group_info == NULL) {
2616           0 :                  printk(KERN_ERR "EXT4-fs: can't allocate buddy meta group\n");
2617           0 :                  return -ENOMEM;
2618             :          }
2619          94 :          sbi->s_buddy_cache = new_inode(sb);
2620          94 :          if (sbi->s_buddy_cache == NULL) {
2621           0 :                  printk(KERN_ERR "EXT4-fs: can't get new inode\n");
2622           0 :                  goto err_freesgi;
2623             :          }
2624         188 :          EXT4_I(sbi->s_buddy_cache)->i_disksize = 0;
2625             :
2626          94 :          metalen = sizeof(*meta_group_info) << EXT4_DESC_PER_BLOCK_BITS(sb);
2627       42802 :          for (i = 0; i < num_meta_group_infos; i++) {
2628       42708 :                  if ((i + 1) == num_meta_group_infos)
2629          94 :                          metalen = sizeof(*meta_group_info) *
2630             :                                  (ngroups -
2631             :                                          (i << EXT4_DESC_PER_BLOCK_BITS(sb)));
2632       85416 :                  meta_group_info = kmalloc(metalen, GFP_KERNEL);
2633       42708 :                  if (meta_group_info == NULL) {
2634           0 :                          printk(KERN_ERR "EXT4-fs: can't allocate mem for a "
2635             :                                  "buddy group\n");
2636           0 :                          goto err_freemeta;
2637             :                  }
2638       42708 :                  sbi->s_group_info[i] = meta_group_info;
2639             :          }
2640             :
2641     1719134 :          for (i = 0; i < ngroups; i++) {
2642     1719040 :                  desc = ext4_get_group_desc(sb, i, NULL);
2643     1719040 :                  if (desc == NULL) {
2644           0 :                          printk(KERN_ERR
2645             :                                  "EXT4-fs: can't read descriptor %u\n", i);
2646           0 :                          goto err_freebuddy;
2647             :                  }
2648     1719040 :                  if (ext4_mb_add_groupinfo(sb, i, desc) != 0)
2649           0 :                          goto err_freebuddy;
2650             :          }
2651             :
2652          94 :          return 0;
2653             :
2654             : err_freebuddy:
2655           0 :          while (i-- > 0)
2656           0 :                  kfree(ext4_get_group_info(sb, i));
2657           0 :          i = num_meta_group_infos;
2658             : err_freemeta:
2659           0 :          while (i-- > 0)
2660           0 :                  kfree(sbi->s_group_info[i]);
2661           0 :          iput(sbi->s_buddy_cache);
2662           0 : err_freesgi:
2663           0 :          kfree(sbi->s_group_info);
2664           0 :          return -ENOMEM;
2665             : }
2666             :
2667             : int ext4_mb_init(struct super_block *sb, int needs_recovery)
2668          94 : {
2669          94 :          struct ext4_sb_info *sbi = EXT4_SB(sb);
2670             :          unsigned i, j;
2671             :          unsigned offset;
2672             :          unsigned max;
2673             :          int ret;
2674             :
2675          94 :          i = (sb->s_blocksize_bits + 2) * sizeof(unsigned short);
2676             :
2677          94 :          sbi->s_mb_offsets = kmalloc(i, GFP_KERNEL);
2678          94 :          if (sbi->s_mb_offsets == NULL) {
2679           0 :                  return -ENOMEM;
2680             :          }
2681             :
2682          94 :          i = (sb->s_blocksize_bits + 2) * sizeof(unsigned int);
2683          94 :          sbi->s_mb_maxs = kmalloc(i, GFP_KERNEL);
2684          94 :          if (sbi->s_mb_maxs == NULL) {
2685           0 :                  kfree(sbi->s_mb_offsets);
```

```
2686                0 :                 return -ENOMEM;
2687                  :         }
2688                  :
2689                  :         /* order 0 is regular bitmap */
2690               94 :         sbi->s_mb_maxs[0] = sb->s_blocksize << 3;
2691               94 :         sbi->s_mb_offsets[0] = 0;
2692                  :
2693               94 :         i = 1;
2694               94 :         offset = 0;
2695               94 :         max = sb->s_blocksize << 2;
2696                  :         do {
2697             1190 :                 sbi->s_mb_offsets[i] = offset;
2698             1190 :                 sbi->s_mb_maxs[i] = max;
2699             1190 :                 offset += 1 << (sb->s_blocksize_bits - i);
2700             1190 :                 max = max >> 1;
2701             1190 :                 i++;
2702             1190 :         } while (i <= sb->s_blocksize_bits + 1);
2703                  :
2704                  :         /* init file for buddy data */
2705               94 :         ret = ext4_mb_init_backend(sb);
2706               94 :         if (ret != 0) {
2707                0 :                 kfree(sbi->s_mb_offsets);
2708                0 :                 kfree(sbi->s_mb_maxs);
2709                0 :                 return ret;
2710                  :         }
2711                  :
2712               94 :         spin_lock_init(&sbi->s_md_lock);
2713               94 :         spin_lock_init(&sbi->s_bal_lock);
2714                  :
2715               94 :         sbi->s_mb_max_to_scan = MB_DEFAULT_MAX_TO_SCAN;
2716               94 :         sbi->s_mb_min_to_scan = MB_DEFAULT_MIN_TO_SCAN;
2717               94 :         sbi->s_mb_stats = MB_DEFAULT_STATS;
2718               94 :         sbi->s_mb_stream_request = MB_DEFAULT_STREAM_THRESHOLD;
2719               94 :         sbi->s_mb_order2_reqs = MB_DEFAULT_ORDER2_REQS;
2720               94 :         sbi->s_mb_history_filter = EXT4_MB_HISTORY_DEFAULT;
2721               94 :         sbi->s_mb_group_prealloc = MB_DEFAULT_GROUP_PREALLOC;
2722                  :
2723               94 :         sbi->s_locality_groups = alloc_percpu(struct ext4_locality_group);
2724               94 :         if (sbi->s_locality_groups == NULL) {
2725                0 :                 kfree(sbi->s_mb_offsets);
2726                0 :                 kfree(sbi->s_mb_maxs);
2727                0 :                 return -ENOMEM;
2728                  :         }
2729             1786 :         for_each_possible_cpu(i) {
2730                  :                 struct ext4_locality_group *lg;
2731              752 :                 lg = per_cpu_ptr(sbi->s_locality_groups, i);
2732              752 :                 mutex_init(&lg->lg_mutex);
2733             8272 :                 for (j = 0; j < PREALLOC_TB_SIZE; j++)
2734             7520 :                         INIT_LIST_HEAD(&lg->lg_prealloc_list[j]);
2735              752 :                 spin_lock_init(&lg->lg_prealloc_lock);
2736                  :         }
2737                  :
2738                  :         ext4_mb_history_init(sb);
2739                  :
2740               94 :         if (sbi->s_journal)
2741               94 :                 sbi->s_journal->j_commit_callback = release_blocks_on_commit;
2742                  :
2743               94 :         printk(KERN_INFO "EXT4-fs: mballoc enabled\n");
2744               94 :         return 0;
2745                  : }
2746                  :
2747                  : /* need to called with the ext4 group lock held */
2748                  : static void ext4_mb_cleanup_pa(struct ext4_group_info *grp)
2749                  : {
2750                  :         struct ext4_prealloc_space *pa;
2751                  :         struct list_head *cur, *tmp;
2752          1719040 :         int count = 0;
2753                  :
2754          1720095 :         list_for_each_safe(cur, tmp, &grp->bb_prealloc_list) {
2755             1055 :                 pa = list_entry(cur, struct ext4_prealloc_space, pa_group_list);
2756             1055 :                 list_del(&pa->pa_group_list);
```

```
2757      1055 :                 count++;
2758      1055 :                 kmem_cache_free(ext4_pspace_cachep, pa);
2759           :         }
2760           :         if (count)
2761           :                 mb_debug("mballoc: %u PAs left\n", count);
2762           :
2763           : }
2764           :
2765           : int ext4_mb_release(struct super_block *sb)
2766        94 : {
2767        94 :         ext4_group_t ngroups = ext4_get_groups_count(sb);
2768           :         ext4_group_t i;
2769           :         int num_meta_group_infos;
2770           :         struct ext4_group_info *grinfo;
2771        94 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
2772           :
2773        94 :         if (sbi->s_group_info) {
2774   1719134 :                 for (i = 0; i < ngroups; i++) {
2775   1719040 :                         grinfo = ext4_get_group_info(sb, i);
2776           : #ifdef DOUBLE_CHECK
2777           :                         kfree(grinfo->bb_bitmap);
2778           : #endif
2779           :                         ext4_lock_group(sb, i);
2780           :                         ext4_mb_cleanup_pa(grinfo);
2781           :                         ext4_unlock_group(sb, i);
2782   1719040 :                         kfree(grinfo);
2783           :                 }
2784       188 :                 num_meta_group_infos = (ngroups +
2785           :                                         EXT4_DESC_PER_BLOCK(sb) - 1) >>
2786           :                         EXT4_DESC_PER_BLOCK_BITS(sb);
2787     42802 :                 for (i = 0; i < num_meta_group_infos; i++)
2788     42708 :                         kfree(sbi->s_group_info[i]);
2789        94 :                 kfree(sbi->s_group_info);
2790           :         }
2791        94 :         kfree(sbi->s_mb_offsets);
2792        94 :         kfree(sbi->s_mb_maxs);
2793        94 :         if (sbi->s_buddy_cache)
2794        94 :                 iput(sbi->s_buddy_cache);
2795        94 :         if (sbi->s_mb_stats) {
2796       376 :                 printk(KERN_INFO
2797           :                        "EXT4-fs: mballoc: %u blocks %u reqs (%u success)\n",
2798           :                                 atomic_read(&sbi->s_bal_allocated),
2799           :                                 atomic_read(&sbi->s_bal_reqs),
2800           :                                 atomic_read(&sbi->s_bal_success));
2801       564 :                 printk(KERN_INFO
2802           :                       "EXT4-fs: mballoc: %u extents scanned, %u goal hits, "
2803           :                                 "%u 2^N hits, %u breaks, %u lost\n",
2804           :                                 atomic_read(&sbi->s_bal_ex_scanned),
2805           :                                 atomic_read(&sbi->s_bal_goals),
2806           :                                 atomic_read(&sbi->s_bal_2orders),
2807           :                                 atomic_read(&sbi->s_bal_breaks),
2808           :                                 atomic_read(&sbi->s_mb_lost_chunks));
2809        94 :                 printk(KERN_INFO
2810           :                        "EXT4-fs: mballoc: %lu generated and it took %Lu\n",
2811           :                                 sbi->s_mb_buddies_generated++,
2812           :                                 sbi->s_mb_generation_time);
2813       282 :                 printk(KERN_INFO
2814           :                        "EXT4-fs: mballoc: %u preallocated, %u discarded\n",
2815           :                                 atomic_read(&sbi->s_mb_preallocated),
2816           :                                 atomic_read(&sbi->s_mb_discarded));
2817           :         }
2818           :
2819        94 :         free_percpu(sbi->s_locality_groups);
2820           :         ext4_mb_history_release(sb);
2821           :
2822        94 :         return 0;
2823           : }
2824           :
2825           : /*
2826           :  * This function is called by the jbd2 layer once the commit has finished,
2827           :  * so we know we can free the blocks that were released with that commit.
```

```
2828                :  */
2829                : static void release_blocks_on_commit(journal_t *journal, transaction_t *txn)
2830          42653 : {
2831          42653 :         struct super_block *sb = journal->j_private;
2832                :         struct ext4_buddy e4b;
2833                :         struct ext4_group_info *db;
2834          42653 :         int err, count = 0, count2 = 0;
2835                :         struct ext4_free_data *entry;
2836                :         ext4_fsblk_t discard_block;
2837                :         struct list_head *l, *ltmp;
2838                :
2839        3276890 :         list_for_each_safe(l, ltmp, &txn->t_private_list) {
2840        3234237 :                 entry = list_entry(l, struct ext4_free_data, list);
2841                :
2842                :                 mb_debug("gonna free %u blocks in group %u (0x%p):",
2843                :                          entry->count, entry->group, entry);
2844                :
2845        3234237 :                 err = ext4_mb_load_buddy(sb, entry->group, &e4b);
2846                :                 /* we expect to find existing buddy because it's pinned */
2847        3234237 :                 BUG_ON(err != 0);
2848                :
2849        3234237 :                 db = e4b.bd_info;
2850                :                 /* there are blocks to put in buddy to make them really free */
2851        3234237 :                 count += entry->count;
2852        3234237 :                 count2++;
2853        3234237 :                 ext4_lock_group(sb, entry->group);
2854                :                 /* Take it out of per group rb tree */
2855        3234237 :                 rb_erase(&entry->node, &(db->bb_free_root));
2856        3234237 :                 mb_free_blocks(NULL, &e4b, entry->start_blk, entry->count);
2857                :
2858        3234237 :                 if (!db->bb_free_root.rb_node) {
2859                :                         /* No more items in the per group rb tree
2860                :                          * balance refcounts from ext4_mb_free_metadata()
2861                :                          */
2862         895564 :                         page_cache_release(e4b.bd_buddy_page);
2863         895564 :                         page_cache_release(e4b.bd_bitmap_page);
2864                :                 }
2865        3234237 :                 ext4_unlock_group(sb, entry->group);
2866        9702711 :                 discard_block = (ext4_fsblk_t) entry->group * EXT4_BLOCKS_PER_GROUP(sb)
2867                :                         + entry->start_blk
2868                :                         + le32_to_cpu(EXT4_SB(sb)->s_es->s_first_data_block);
2869        3234237 :                 trace_ext4_discard_blocks(sb, (unsigned long long)discard_block,
2870                :                                           entry->count);
2871        3234237 :                 sb_issue_discard(sb, discard_block, entry->count);
2872                :
2873        3234237 :                 kmem_cache_free(ext4_free_ext_cachep, entry);
2874        3234237 :                 ext4_mb_release_desc(&e4b);
2875                :         }
2876                :
2877                :         mb_debug("freed %u blocks in %u structures\n", count, count2);
2878          42653 : }
2879                :
2880                : int __init init_ext4_mballoc(void)
2881              0 : {
2882              0 :         ext4_pspace_cachep =
2883                :                 kmem_cache_create("ext4_prealloc_space",
2884                :                                   sizeof(struct ext4_prealloc_space),
2885                :                                   0, SLAB_RECLAIM_ACCOUNT, NULL);
2886              0 :         if (ext4_pspace_cachep == NULL)
2887              0 :                 return -ENOMEM;
2888                :
2889              0 :         ext4_ac_cachep =
2890                :                 kmem_cache_create("ext4_alloc_context",
2891                :                                   sizeof(struct ext4_allocation_context),
2892                :                                   0, SLAB_RECLAIM_ACCOUNT, NULL);
2893              0 :         if (ext4_ac_cachep == NULL) {
2894              0 :                 kmem_cache_destroy(ext4_pspace_cachep);
2895              0 :                 return -ENOMEM;
2896                :         }
2897                :
2898              0 :         ext4_free_ext_cachep =
```

```
2899                :                 kmem_cache_create("ext4_free_block_extents",
2900                :                                   sizeof(struct ext4_free_data),
2901                :                                   0, SLAB_RECLAIM_ACCOUNT, NULL);
2902              0 :         if (ext4_free_ext_cachep == NULL) {
2903              0 :                 kmem_cache_destroy(ext4_pspace_cachep);
2904              0 :                 kmem_cache_destroy(ext4_ac_cachep);
2905              0 :                 return -ENOMEM;
2906                :         }
2907              0 :         return 0;
2908                : }
2909                :
2910                : void exit_ext4_mballoc(void)
2911              0 : {
2912                :         /*
2913                :          * Wait for completion of call_rcu()'s on ext4_pspace_cachep
2914                :          * before destroying the slab cache.
2915                :          */
2916              0 :         rcu_barrier();
2917              0 :         kmem_cache_destroy(ext4_pspace_cachep);
2918              0 :         kmem_cache_destroy(ext4_ac_cachep);
2919              0 :         kmem_cache_destroy(ext4_free_ext_cachep);
2920              0 : }
2921                :
2922                :
2923                : /*
2924                :  * Check quota and mark choosed space (ac->ac_b_ex) non-free in bitmaps
2925                :  * Returns 0 if success or error code
2926                :  */
2927                : static noinline_for_stack int
2928                : ext4_mb_mark_diskspace_used(struct ext4_allocation_context *ac,
2929                :                             handle_t *handle, unsigned int reserv_blks)
2930      220365684 : {
2931      220365684 :         struct buffer_head *bitmap_bh = NULL;
2932                :         struct ext4_super_block *es;
2933                :         struct ext4_group_desc *gdp;
2934                :         struct buffer_head *gdp_bh;
2935                :         struct ext4_sb_info *sbi;
2936                :         struct super_block *sb;
2937                :         ext4_fsblk_t block;
2938                :         int err, len;
2939                :
2940      220365684 :         BUG_ON(ac->ac_status != AC_STATUS_FOUND);
2941      219676781 :         BUG_ON(ac->ac_b_ex.fe_len <= 0);
2942                :
2943      220032578 :         sb = ac->ac_sb;
2944      220032578 :         sbi = EXT4_SB(sb);
2945      220032578 :         es = sbi->s_es;
2946                :
2947                :
2948      220032578 :         err = -EIO;
2949      220032578 :         bitmap_bh = ext4_read_block_bitmap(sb, ac->ac_b_ex.fe_group);
2950      221747831 :         if (!bitmap_bh)
2951              0 :                 goto out_err;
2952                :
2953      221747831 :         err = ext4_journal_get_write_access(handle, bitmap_bh);
2954      221757459 :         if (err)
2955              0 :                 goto out_err;
2956                :
2957      221757459 :         err = -EIO;
2958      221757459 :         gdp = ext4_get_group_desc(sb, ac->ac_b_ex.fe_group, &gdp_bh);
2959      220511319 :         if (!gdp)
2960              0 :                 goto out_err;
2961                :
2962                :         ext4_debug("using block group %u(%d)\n", ac->ac_b_ex.fe_group,
2963                :                         ext4_free_blks_count(sb, gdp));
2964                :
2965      220511319 :         err = ext4_journal_get_write_access(handle, gdp_bh);
2966      222268997 :         if (err)
2967              0 :                 goto out_err;
2968                :
2969      444537994 :         block = ac->ac_b_ex.fe_group * EXT4_BLOCKS_PER_GROUP(sb)
```

```
2970                :                 + ac->ac_b_ex.fe_start
2971                :                 + le32_to_cpu(es->s_first_data_block);
2972                :
2973      222268997 :         len = ac->ac_b_ex.fe_len;
2974      222268997 :         if (!ext4_data_block_valid(sbi, block, len)) {
2975              0 :                 ext4_error(sb, __func__,
2976                :                            "Allocating blocks %llu-%llu which overlap "
2977                :                            "fs metadata\n", block, block+len);
2978                :                 /* File system mounted not to panic on error
2979                :                  * Fix the bitmap and repeat the block allocation
2980                :                  * We leak some of the blocks here.
2981                :                  */
2982              0 :                 ext4_lock_group(sb, ac->ac_b_ex.fe_group);
2983              0 :                 mb_set_bits(bitmap_bh->b_data, ac->ac_b_ex.fe_start,
2984                :                             ac->ac_b_ex.fe_len);
2985              0 :                 ext4_unlock_group(sb, ac->ac_b_ex.fe_group);
2986              0 :                 err = ext4_handle_dirty_metadata(handle, NULL, bitmap_bh);
2987              0 :                 if (!err)
2988              0 :                         err = -EAGAIN;
2989                :                 goto out_err;
2990                :         }
2991                :
2992      222597017 :         ext4_lock_group(sb, ac->ac_b_ex.fe_group);
2993                : #ifdef AGGRESSIVE_CHECK
2994                :         {
2995                :                 int i;
2996                :                 for (i = 0; i < ac->ac_b_ex.fe_len; i++) {
2997                :                         BUG_ON(mb_test_bit(ac->ac_b_ex.fe_start + i,
2998                :                                            bitmap_bh->b_data));
2999                :                 }
3000                :         }
3001                : #endif
3002      222492017 :         mb_set_bits(bitmap_bh->b_data, ac->ac_b_ex.fe_start,ac->ac_b_ex.fe_len);
3003      222682568 :         if (gdp->bg_flags & cpu_to_le16(EXT4_BG_BLOCK_UNINIT)) {
3004          29837 :                 gdp->bg_flags &= cpu_to_le16(~EXT4_BG_BLOCK_UNINIT);
3005          29837 :                 ext4_free_blks_set(sb, gdp,
3006                :                                    ext4_free_blocks_after_init(sb,
3007                :                                    ac->ac_b_ex.fe_group, gdp));
3008                :         }
3009      222682568 :         len = ext4_free_blks_count(sb, gdp) - ac->ac_b_ex.fe_len;
3010      222099974 :         ext4_free_blks_set(sb, gdp, len);
3011      222321586 :         gdp->bg_checksum = ext4_group_desc_csum(sbi, ac->ac_b_ex.fe_group, gdp);
3012                :
3013      222861231 :         ext4_unlock_group(sb, ac->ac_b_ex.fe_group);
3014      222832929 :         percpu_counter_sub(&sbi->s_freeblocks_counter, ac->ac_b_ex.fe_len);
3015                :         /*
3016                :          * Now reduce the dirty block count also. Should not go negative
3017                :          */
3018      222814759 :         if (!(ac->ac_flags & EXT4_MB_DELALLOC_RESERVED))
3019                :                 /* release all the reserved blocks if non delalloc */
3020      208345857 :                 percpu_counter_sub(&sbi->s_dirtyblocks_counter, reserv_blks);
3021                :         else {
3022       14468902 :                 percpu_counter_sub(&sbi->s_dirtyblocks_counter,
3023                :                                    ac->ac_b_ex.fe_len);
3024                :                 /* convert reserved quota blocks to real quota blocks */
3025       14468130 :                 vfs_dq_claim_block(ac->ac_inode, ac->ac_b_ex.fe_len);
3026                :         }
3027                :
3028      222023967 :         if (sbi->s_log_groups_per_flex) {
3029                :                 ext4_group_t flex_group = ext4_flex_group(sbi,
3030      443434312 :                                                           ac->ac_b_ex.fe_group);
3031      221717156 :                 atomic_sub(ac->ac_b_ex.fe_len,
3032                :                            &sbi->s_flex_groups[flex_group].free_blocks);
3033                :         }
3034                :
3035      222855993 :         err = ext4_handle_dirty_metadata(handle, NULL, bitmap_bh);
3036      222644660 :         if (err)
3037              0 :                 goto out_err;
3038      222644660 :         err = ext4_handle_dirty_metadata(handle, NULL, gdp_bh);
3039                :
3040      222770056 : out_err:
```

```
3041  222770056 :          sb->s_dirt = 1;
3042            :          brelse(bitmap_bh);
3043  222795613 :          return err;
3044            : }
3045            :
3046            : /*
3047            :  * here we normalize request for locality group
3048            :  * Group request are normalized to s_strip size if we set the same via mount
3049            :  * option. If not we set it to s_mb_group_prealloc which can be configured via
3050            :  * /sys/fs/ext4/<partition>/mb_group_prealloc
3051            :  *
3052            :  * XXX: should we try to preallocate more than the group has now?
3053            :  */
3054            : static void ext4_mb_normalize_group_request(struct ext4_allocation_context *ac)
3055            : {
3056      68536 :          struct super_block *sb = ac->ac_sb;
3057      68536 :          struct ext4_locality_group *lg = ac->ac_lg;
3058            :
3059      68536 :          BUG_ON(lg == NULL);
3060      68536 :          if (EXT4_SB(sb)->s_stripe)
3061          0 :                  ac->ac_g_ex.fe_len = EXT4_SB(sb)->s_stripe;
3062            :          else
3063      68536 :                  ac->ac_g_ex.fe_len = EXT4_SB(sb)->s_mb_group_prealloc;
3064            :          mb_debug("#%u: goal %u blocks for locality group\n",
3065            :                  current->pid, ac->ac_g_ex.fe_len);
3066            : }
3067            :
3068            : /*
3069            :  * Normalization means making request better in terms of
3070            :  * size and alignment
3071            :  */
3072            : static noinline_for_stack void
3073            : ext4_mb_normalize_request(struct ext4_allocation_context *ac,
3074            :                                  struct ext4_allocation_request *ar)
3075   13274370 : {
3076            :          int bsbits, max;
3077            :          ext4_lblk_t end;
3078            :          loff_t size, orig_size, start_off;
3079            :          ext4_lblk_t start, orig_start;
3080   26548740 :          struct ext4_inode_info *ei = EXT4_I(ac->ac_inode);
3081            :          struct ext4_prealloc_space *pa;
3082            :
3083            :          /* do normalize only data requests, metadata requests
3084            :             do not need preallocation */
3085   13274370 :          if (!(ac->ac_flags & EXT4_MB_HINT_DATA))
3086    1888105 :                  return;
3087            :
3088            :          /* sometime caller may want exact blocks */
3089   11386265 :          if (unlikely(ac->ac_flags & EXT4_MB_HINT_GOAL_ONLY))
3090          0 :                  return;
3091            :
3092            :          /* caller may indicate that preallocation isn't
3093            :           * required (it's a tail, for example) */
3094   11383443 :          if (ac->ac_flags & EXT4_MB_HINT_NOPREALLOC)
3095          0 :                  return;
3096            :
3097   11383443 :          if (ac->ac_flags & EXT4_MB_HINT_GROUP_ALLOC) {
3098            :                  ext4_mb_normalize_group_request(ac);
3099            :                  return ;
3100            :          }
3101            :
3102   11314907 :          bsbits = ac->ac_sb->s_blocksize_bits;
3103            :
3104            :          /* first, let's learn actual file size
3105            :           * given current request is allocated */
3106   11314907 :          size = ac->ac_o_ex.fe_logical + ac->ac_o_ex.fe_len;
3107   11314907 :          size = size << bsbits;
3108   22631507 :          if (size < i_size_read(ac->ac_inode))
3109    1026709 :                  size = i_size_read(ac->ac_inode);
3110            :
3111            :          /* max size of free chunks */
```

```
3112  11316597 :          max = 2 << bsbits;
3113           :
3114           : #define NRL_CHECK_SIZE(req, size, max, chunk_size)      \
3115           :                  (req <= (size) || max <= (chunk_size))
3116           :
3117           :          /* first, try to predict filesize */
3118           :          /* XXX: should this table be tunable? */
3119  11316597 :          start_off = 0;
3120  11316597 :          if (size <= 16 * 1024) {
3121    155431 :                  size = 16 * 1024;
3122  11161166 :          } else if (size <= 32 * 1024) {
3123    361834 :                  size = 32 * 1024;
3124  10799332 :          } else if (size <= 64 * 1024) {
3125    811110 :                  size = 64 * 1024;
3126   9988222 :          } else if (size <= 128 * 1024) {
3127   1966087 :                  size = 128 * 1024;
3128   8022135 :          } else if (size <= 256 * 1024) {
3129   1941729 :                  size = 256 * 1024;
3130   6080406 :          } else if (size <= 512 * 1024) {
3131   2229178 :                  size = 512 * 1024;
3132   3851228 :          } else if (size <= 1024 * 1024) {
3133   2091110 :                  size = 1024 * 1024;
3134   1760118 :          } else if (NRL_CHECK_SIZE(size, 4 * 1024 * 1024, max, 2 * 1024)) {
3135   1504233 :                  start_off = ((loff_t)ac->ac_o_ex.fe_logical >>
3136           :                                                  (21 - bsbits)) << 21;
3137   1504233 :                  size = 2 * 1024 * 1024;
3138    255885 :          } else if (NRL_CHECK_SIZE(size, 8 * 1024 * 1024, max, 4 * 1024)) {
3139     20939 :                  start_off = ((loff_t)ac->ac_o_ex.fe_logical >>
3140           :                                                      (22 - bsbits)) << 22;
3141     20939 :                  size = 4 * 1024 * 1024;
3142    234946 :          } else if (NRL_CHECK_SIZE(ac->ac_o_ex.fe_len,
3143           :                                          (8<<20)>>bsbits, max, 8 * 1024)) {
3144    234946 :                  start_off = ((loff_t)ac->ac_o_ex.fe_logical >>
3145           :                                                      (23 - bsbits)) << 23;
3146    234946 :                  size = 8 * 1024 * 1024;
3147           :          } else {
3148         0 :                  start_off = (loff_t)ac->ac_o_ex.fe_logical << bsbits;
3149         0 :                  size      = ac->ac_o_ex.fe_len << bsbits;
3150           :          }
3151  11316597 :          orig_size = size = size >> bsbits;
3152  11316597 :          orig_start = start = start_off >> bsbits;
3153           :
3154           :          /* don't cover already allocated blocks in selected range */
3155  11316597 :          if (ar->pleft && start <= ar->lleft) {
3156   8016778 :                  size -= ar->lleft + 1 - start;
3157   8016778 :                  start = ar->lleft + 1;
3158           :          }
3159  11316597 :          if (ar->pright && start + size - 1 >= ar->lright)
3160      1796 :                  size -= start + size - ar->lright;
3161           :
3162  11316597 :          end = start + size;
3163           :
3164           :          /* check we don't cross already preallocated blocks */
3165           :          rcu_read_lock();
3166  22646589 :          list_for_each_entry_rcu(pa, &ei->i_prealloc_list, pa_inode_list) {
3167           :                  ext4_lblk_t pa_end;
3168           :
3169      6917 :                  if (pa->pa_deleted)
3170         0 :                          continue;
3171      6917 :                  spin_lock(&pa->pa_lock);
3172      6917 :                  if (pa->pa_deleted) {
3173         0 :                          spin_unlock(&pa->pa_lock);
3174           :                          continue;
3175           :                  }
3176           :
3177      6917 :                  pa_end = pa->pa_lstart + pa->pa_len;
3178           :
3179           :                  /* PA must not overlap original request */
3180      6917 :                  BUG_ON(!(ac->ac_o_ex.fe_logical >= pa_end ||
3181           :                          ac->ac_o_ex.fe_logical < pa->pa_lstart));
3182           :
```

```
3183                :                 /* skip PA normalized request doesn't overlap with */
3184           6917 :                 if (pa->pa_lstart >= end) {
3185           1529 :                         spin_unlock(&pa->pa_lock);
3186                :                         continue;
3187                :                 }
3188           5388 :                 if (pa_end <= start) {
3189           2157 :                         spin_unlock(&pa->pa_lock);
3190                :                         continue;
3191                :                 }
3192           3231 :                 BUG_ON(pa->pa_lstart <= start && pa_end >= end);
3193                :
3194           3231 :                 if (pa_end <= ac->ac_o_ex.fe_logical) {
3195           3037 :                         BUG_ON(pa_end < start);
3196           3037 :                         start = pa_end;
3197                :                 }
3198                :
3199           3231 :                 if (pa->pa_lstart > ac->ac_o_ex.fe_logical) {
3200            194 :                         BUG_ON(pa->pa_lstart > end);
3201            194 :                         end = pa->pa_lstart;
3202                :                 }
3203           3231 :                 spin_unlock(&pa->pa_lock);
3204                :         }
3205                :         rcu_read_unlock();
3206       11316158 :         size = end - start;
3207                :
3208                :         /* XXX: extra loop to check we really don't overlap preallocations */
3209                :         rcu_read_lock();
3210       22645777 :         list_for_each_entry_rcu(pa, &ei->i_prealloc_list, pa_inode_list) {
3211                :                 ext4_lblk_t pa_end;
3212           6917 :                 spin_lock(&pa->pa_lock);
3213           6917 :                 if (pa->pa_deleted == 0) {
3214           6917 :                         pa_end = pa->pa_lstart + pa->pa_len;
3215           6917 :                         BUG_ON(!(start >= pa_end || end <= pa->pa_lstart));
3216                :                 }
3217           6917 :                 spin_unlock(&pa->pa_lock);
3218                :         }
3219                :         rcu_read_unlock();
3220                :
3221       11315785 :         if (start + size <= ac->ac_o_ex.fe_logical &&
3222                :                         start > ac->ac_o_ex.fe_logical) {
3223                0 :                 printk(KERN_ERR "start %lu, size %lu, fe_logical %lu\n",
3224                :                         (unsigned long) start, (unsigned long) size,
3225                :                         (unsigned long) ac->ac_o_ex.fe_logical);
3226                :         }
3227       11315785 :         BUG_ON(start + size <= ac->ac_o_ex.fe_logical &&
3228                :                         start > ac->ac_o_ex.fe_logical);
3229       22632866 :         BUG_ON(size <= 0 || size > EXT4_BLOCKS_PER_GROUP(ac->ac_sb));
3230                :
3231                :         /* now prepare goal request */
3232                :
3233                :         /* XXX: is it better to align blocks WRT to logical
3234                :          * placement or satisfy big request as is */
3235       11319769 :         ac->ac_g_ex.fe_logical = start;
3236       11319769 :         ac->ac_g_ex.fe_len = size;
3237                :
3238                :         /* define goal start in order to merge */
3239       11319769 :         if (ar->pright && (ar->lright == (start + size))) {
3240                :                 /* merge to the right */
3241           1602 :                 ext4_get_group_no_and_offset(ac->ac_sb, ar->pright - size,
3242                :                                              &ac->ac_f_ex.fe_group,
3243                :                                              &ac->ac_f_ex.fe_start);
3244           1602 :                 ac->ac_flags |= EXT4_MB_HINT_TRY_GOAL;
3245                :         }
3246       11319769 :         if (ar->pleft && (ar->lleft + 1 == start)) {
3247                :                 /* merge to the left */
3248        8359885 :                 ext4_get_group_no_and_offset(ac->ac_sb, ar->pleft + 1,
3249                :                                              &ac->ac_f_ex.fe_group,
3250                :                                              &ac->ac_f_ex.fe_start);
3251        8363917 :                 ac->ac_flags |= EXT4_MB_HINT_TRY_GOAL;
3252                :         }
3253                :
```

```
3254                :         mb_debug("goal: %u(was %u) blocks at %u\n", (unsigned) size,
3255                :                 (unsigned) orig_size, (unsigned) start);
3256                : }
3257                :
3258                : static void ext4_mb_collect_stats(struct ext4_allocation_context *ac)
3259                : {
3260     442155664  :         struct ext4_sb_info *sbi = EXT4_SB(ac->ac_sb);
3261                :
3262     221077832  :         if (sbi->s_mb_stats && ac->ac_g_ex.fe_len > 1) {
3263      15041057  :                 atomic_inc(&sbi->s_bal_reqs);
3264      15041026  :                 atomic_add(ac->ac_b_ex.fe_len, &sbi->s_bal_allocated);
3265      15040330  :                 if (ac->ac_o_ex.fe_len >= ac->ac_g_ex.fe_len)
3266       4661514  :                         atomic_inc(&sbi->s_bal_success);
3267      15040344  :                 atomic_add(ac->ac_found, &sbi->s_bal_ex_scanned);
3268      15041160  :                 if (ac->ac_g_ex.fe_start == ac->ac_b_ex.fe_start &&
3269                :                         ac->ac_g_ex.fe_group == ac->ac_b_ex.fe_group)
3270       1304328  :                         atomic_inc(&sbi->s_bal_goals);
3271      15040903  :                 if (ac->ac_found > sbi->s_mb_max_to_scan)
3272       1676348  :                         atomic_inc(&sbi->s_bal_breaks);
3273                :         }
3274                :
3275     221077111  :         ext4_mb_store_history(ac);
3276                : }
3277                :
3278                : /*
3279                :  * use blocks preallocated to inode
3280                :  */
3281                : static void ext4_mb_use_inode_pa(struct ext4_allocation_context *ac,
3282                :                                  struct ext4_prealloc_space *pa)
3283     195708963  : {
3284                :         ext4_fsblk_t start;
3285                :         ext4_fsblk_t end;
3286                :         int len;
3287                :
3288                :         /* found preallocated blocks, use them */
3289     195708963  :         start = pa->pa_pstart + (ac->ac_o_ex.fe_logical - pa->pa_lstart);
3290     195708963  :         end = min(pa->pa_pstart + pa->pa_len, start + ac->ac_o_ex.fe_len);
3291     195708963  :         len = end - start;
3292     195708963  :         ext4_get_group_no_and_offset(ac->ac_sb, start, &ac->ac_b_ex.fe_group,
3293                :                                      &ac->ac_b_ex.fe_start);
3294     194743730  :         ac->ac_b_ex.fe_len = len;
3295     194743730  :         ac->ac_status = AC_STATUS_FOUND;
3296     194743730  :         ac->ac_pa = pa;
3297                :
3298     194743730  :         BUG_ON(start < pa->pa_pstart);
3299     195224678  :         BUG_ON(start + len > pa->pa_pstart + pa->pa_len);
3300     196251070  :         BUG_ON(pa->pa_free < len);
3301     194590351  :         pa->pa_free -= len;
3302                :
3303                :         mb_debug("use %llu/%u from inode pa %p\n", start, len, pa);
3304     194590351  : }
3305                :
3306                : /*
3307                :  * use blocks preallocated to locality group
3308                :  */
3309                : static void ext4_mb_use_group_pa(struct ext4_allocation_context *ac,
3310                :                                  struct ext4_prealloc_space *pa)
3311                : {
3312      21609364  :         unsigned int len = ac->ac_o_ex.fe_len;
3313                :
3314      21609364  :         ext4_get_group_no_and_offset(ac->ac_sb, pa->pa_pstart,
3315                :                                      &ac->ac_b_ex.fe_group,
3316                :                                      &ac->ac_b_ex.fe_start);
3317      21602092  :         ac->ac_b_ex.fe_len = len;
3318      21602092  :         ac->ac_status = AC_STATUS_FOUND;
3319      21602092  :         ac->ac_pa = pa;
3320                :
3321                :         /* we don't correct pa_pstart or pa_plen here to avoid
3322                :          * possible race when the group is being loaded concurrently
3323                :          * instead we correct pa later, after blocks are marked
3324                :          * in on-disk bitmap -- see ext4_mb_release_context()
```

```
3325                 :          * Other CPUs are prevented from allocating from this pa by lg_mutex
3326                 :          */
3327                 :         mb_debug("use %u/%u from group pa %p\n", pa->pa_lstart-len, len, pa);
3328                 : }
3329                 :
3330                 : /*
3331                 :  * Return the prealloc space that have minimal distance
3332                 :  * from the goal block. @cpa is the prealloc
3333                 :  * space that is having currently known minimal distance
3334                 :  * from the goal block.
3335                 :  */
3336                 : static struct ext4_prealloc_space *
3337                 : ext4_mb_check_group_pa(ext4_fsblk_t goal_block,
3338                 :                        struct ext4_prealloc_space *pa,
3339                 :                        struct ext4_prealloc_space *cpa)
3340                 : {
3341                 :         ext4_fsblk_t cur_distance, new_distance;
3342                 :
3343      21682390 :         if (cpa == NULL) {
3344      21526709 :                 atomic_inc(&pa->pa_count);
3345      21538279 :                 return pa;
3346                 :         }
3347        155681 :         cur_distance = abs(goal_block - cpa->pa_pstart);
3348        155681 :         new_distance = abs(goal_block - pa->pa_pstart);
3349                 :
3350        155681 :         if (cur_distance < new_distance)
3351         96198 :                 return cpa;
3352                 :
3353                 :         /* drop the previous reference */
3354         59483 :         atomic_dec(&cpa->pa_count);
3355         59481 :         atomic_inc(&pa->pa_count);
3356         59479 :         return pa;
3357                 : }
3358                 :
3359                 : /*
3360                 :  * search goal blocks in preallocated space
3361                 :  */
3362                 : static noinline_for_stack int
3363                 : ext4_mb_use_preallocated(struct ext4_allocation_context *ac)
3364     219651348 : {
3365                 :         int order, i;
3366     439302696 :         struct ext4_inode_info *ei = EXT4_I(ac->ac_inode);
3367                 :         struct ext4_locality_group *lg;
3368     219651348 :         struct ext4_prealloc_space *pa, *cpa = NULL;
3369                 :         ext4_fsblk_t goal_block;
3370                 :
3371                 :         /* only data can be preallocated */
3372     219651348 :         if (!(ac->ac_flags & EXT4_MB_HINT_DATA))
3373       1887643 :                 return 0;
3374                 :
3375                 :         /* first, try per-file preallocation */
3376                 :         rcu_read_lock();
3377     436334335 :         list_for_each_entry_rcu(pa, &ei->i_prealloc_list, pa_inode_list) {
3378                 :
3379                 :                 /* all fields in this condition don't change,
3380                 :                  * so we can skip locking for them */
3381     185630628 :                 if (ac->ac_o_ex.fe_logical < pa->pa_lstart ||
3382                 :                         ac->ac_o_ex.fe_logical >= pa->pa_lstart + pa->pa_len)
3383                 :                         continue;
3384                 :
3385                 :                 /* found preallocated blocks, use them */
3386     187762407 :                 spin_lock(&pa->pa_lock);
3387     186757677 :                 if (pa->pa_deleted == 0 && pa->pa_free) {
3388     187307451 :                         atomic_inc(&pa->pa_count);
3389     187455369 :                         ext4_mb_use_inode_pa(ac, pa);
3390     185656846 :                         spin_unlock(&pa->pa_lock);
3391     187935481 :                         ac->ac_criteria = 10;
3392                 :                         rcu_read_unlock();
3393     187935481 :                         return 1;
3394                 :                 }
3395           0     :                 spin_unlock(&pa->pa_lock);
```

```
3396                :         }
3397                :         rcu_read_unlock();
3398                :
3399                :         /* can we use group allocation? */
3400       32913551 :         if (!(ac->ac_flags & EXT4_MB_HINT_GROUP_ALLOC))
3401       11314677 :                 return 0;
3402                :
3403                :         /* inode may have no locality group for some reason */
3404       21598874 :         lg = ac->ac_lg;
3405       21598874 :         if (lg == NULL)
3406              0 :                 return 0;
3407       43197748 :         order  = fls(ac->ac_o_ex.fe_len) - 1;
3408       21598874 :         if (order > PREALLOC_TB_SIZE - 1)
3409                :                 /* The max size of hash table is PREALLOC_TB_SIZE */
3410              0 :                 order = PREALLOC_TB_SIZE - 1;
3411                :
3412       64796622 :         goal_block = ac->ac_g_ex.fe_group * EXT4_BLOCKS_PER_GROUP(ac->ac_sb) +
3413                :                      ac->ac_g_ex.fe_start +
3414                :                      le32_to_cpu(EXT4_SB(ac->ac_sb)->s_es->s_first_data_block);
3415                :         /*
3416                :          * search for the prealloc space that is having
3417                :          * minimal distance from the goal block.
3418                :          */
3419      231458027 :         for (i = order; i < PREALLOC_TB_SIZE; i++) {
3420                :                 rcu_read_lock();
3421      465028684 :                 list_for_each_entry_rcu(pa, &lg->lg_prealloc_list[i],
3422                :                                         pa_inode_list) {
3423       22650903 :                         spin_lock(&pa->pa_lock);
3424       22647373 :                         if (pa->pa_deleted == 0 &&
3425                :                                         pa->pa_free >= ac->ac_o_ex.fe_len) {
3426                :
3427       21693956 :                                 cpa = ext4_mb_check_group_pa(goal_block,
3428                :                                                              pa, cpa);
3429                :                         }
3430       22658939 :                         spin_unlock(&pa->pa_lock);
3431                :                 }
3432                :                 rcu_read_unlock();
3433                :         }
3434       21609741 :         if (cpa) {
3435                :                 ext4_mb_use_group_pa(ac, cpa);
3436       21533933 :                 ac->ac_criteria = 20;
3437       21533933 :                 return 1;
3438                :         }
3439          68536 :         return 0;
3440                : }
3441                :
3442                : /*
3443                :  * the function goes through all block freed in the group
3444                :  * but not yet committed and marks them used in in-core bitmap.
3445                :  * buddy must be generated from this bitmap
3446                :  * Need to be called with the ext4 group lock held
3447                :  */
3448                : static void ext4_mb_generate_from_freelist(struct super_block *sb, void *bitmap,
3449                :                                            ext4_group_t group)
3450                : {
3451                :         struct rb_node *n;
3452                :         struct ext4_group_info *grp;
3453                :         struct ext4_free_data *entry;
3454                :
3455          62915 :         grp = ext4_get_group_info(sb, group);
3456          62915 :         n = rb_first(&(grp->bb_free_root));
3457                :
3458          62915 :         while (n) {
3459              0 :                 entry = rb_entry(n, struct ext4_free_data, node);
3460              0 :                 mb_set_bits(bitmap, entry->start_blk, entry->count);
3461              0 :                 n = rb_next(n);
3462                :         }
3463                :         return;
3464                : }
3465                :
3466                : /*
```

```
3467                :  * the function goes through all preallocation in this group and marks them
3468                :  * used in in-core bitmap. buddy must be generated from this bitmap
3469                :  * Need to be called with ext4 group lock held
3470                :  */
3471                : static noinline_for_stack
3472                : void ext4_mb_generate_from_pa(struct super_block *sb, void *bitmap,
3473                :                               ext4_group_t group)
3474           62915 : {
3475           62915 :         struct ext4_group_info *grp = ext4_get_group_info(sb, group);
3476                :         struct ext4_prealloc_space *pa;
3477                :         struct list_head *cur;
3478                :         ext4_group_t groupnr;
3479                :         ext4_grpblk_t start;
3480           62915 :         int preallocated = 0;
3481           62915 :         int count = 0;
3482                :         int len;
3483                :
3484                :         /* all form of preallocation discards first load group,
3485                :          * so the only competing code is preallocation use.
3486                :          * we don't need any locking here
3487                :          * notice we do NOT ignore preallocations with pa_deleted
3488                :          * otherwise we could leave used blocks available for
3489                :          * allocation in buddy when concurrent ext4_mb_put_pa()
3490                :          * is dropping preallocation
3491                :          */
3492          126106 :         list_for_each(cur, &grp->bb_prealloc_list) {
3493             138 :                 pa = list_entry(cur, struct ext4_prealloc_space, pa_group_list);
3494             138 :                 spin_lock(&pa->pa_lock);
3495             138 :                 ext4_get_group_no_and_offset(sb, pa->pa_pstart,
3496                :                                              &groupnr, &start);
3497             138 :                 len = pa->pa_len;
3498             138 :                 spin_unlock(&pa->pa_lock);
3499             138 :                 if (unlikely(len == 0))
3500               0 :                         continue;
3501             138 :                 BUG_ON(groupnr != group);
3502             138 :                 mb_set_bits(bitmap, start, len);
3503             138 :                 preallocated += len;
3504             138 :                 count++;
3505                :         }
3506                :         mb_debug("prellocated %u for group %u\n", preallocated, group);
3507           62915 : }
3508                :
3509                : static void ext4_mb_pa_callback(struct rcu_head *head)
3510         9190721 : {
3511                :         struct ext4_prealloc_space *pa;
3512         9190721 :         pa = container_of(head, struct ext4_prealloc_space, u.pa_rcu);
3513         9190721 :         kmem_cache_free(ext4_pspace_cachep, pa);
3514         9190323 : }
3515                :
3516                : /*
3517                :  * drops a reference to preallocated space descriptor
3518                :  * if this was the last reference and the space is consumed
3519                :  */
3520                : static void ext4_mb_put_pa(struct ext4_allocation_context *ac,
3521                :                            struct super_block *sb, struct ext4_prealloc_space *pa)
3522       216932297 : {
3523                :         ext4_group_t grp;
3524                :         ext4_fsblk_t grp_blk;
3525                :
3526       434016777 :         if (!atomic_dec_and_test(&pa->pa_count) || pa->pa_free != 0)
3527                :                 return;
3528                :
3529                :         /* in this short window concurrent discard can set pa_deleted */
3530         1332712 :         spin_lock(&pa->pa_lock);
3531         1332231 :         if (pa->pa_deleted == 1) {
3532               0 :                 spin_unlock(&pa->pa_lock);
3533                :                 return;
3534                :         }
3535                :
3536         1332231 :         pa->pa_deleted = 1;
3537         1332231 :         spin_unlock(&pa->pa_lock);
```

```
3538                :
3539       1332427  :          grp_blk = pa->pa_pstart;
3540                :          /*
3541                :           * If doing group-based preallocation, pa_pstart may be in the
3542                :           * next group when pa is used up
3543                :           */
3544       1332427  :          if (pa->pa_type == MB_GROUP_PA)
3545         54944  :                  grp_blk--;
3546                :
3547       1332427  :          ext4_get_group_no_and_offset(sb, grp_blk, &grp, NULL);
3548                :
3549                :          /*
3550                :           * possible race:
3551                :           *
3552                :           *  P1 (buddy init)                     P2 (regular allocation)
3553                :           *                                      find block B in PA
3554                :           *  copy on-disk bitmap to buddy
3555                :           *                                      mark B in on-disk bitmap
3556                :           *                                      drop PA from group
3557                :           *  mark all PAs in buddy
3558                :           *
3559                :           * thus, P1 initializes buddy with B available. to prevent this
3560                :           * we make "copy" and "mark all PAs" atomic and serialize "drop PA"
3561                :           * against that pair
3562                :           */
3563       1332405  :          ext4_lock_group(sb, grp);
3564       1332591  :          list_del(&pa->pa_group_list);
3565       1332691  :          ext4_unlock_group(sb, grp);
3566                :
3567       1332750  :          spin_lock(pa->pa_obj_lock);
3568       1332657  :          list_del_rcu(&pa->pa_inode_list);
3569       1332657  :          spin_unlock(pa->pa_obj_lock);
3570                :
3571       1332725  :          call_rcu(&(pa)->u.pa_rcu, ext4_mb_pa_callback);
3572                : }
3573                :
3574                : /*
3575                :  * creates new preallocated space for given inode
3576                :  */
3577                : static noinline_for_stack int
3578                : ext4_mb_new_inode_pa(struct ext4_allocation_context *ac)
3579       9123964  : {
3580       9123964  :          struct super_block *sb = ac->ac_sb;
3581                :          struct ext4_prealloc_space *pa;
3582                :          struct ext4_group_info *grp;
3583                :          struct ext4_inode_info *ei;
3584                :
3585                :          /* preallocate only when found space is larger then requested */
3586       9123964  :          BUG_ON(ac->ac_o_ex.fe_len >= ac->ac_b_ex.fe_len);
3587       9124058  :          BUG_ON(ac->ac_status != AC_STATUS_FOUND);
3588       9124290  :          BUG_ON(!S_ISREG(ac->ac_inode->i_mode));
3589                :
3590       9124058  :          pa = kmem_cache_alloc(ext4_pspace_cachep, GFP_NOFS);
3591       9124297  :          if (pa == NULL)
3592             0  :                  return -ENOMEM;
3593                :
3594       9124297  :          if (ac->ac_b_ex.fe_len < ac->ac_g_ex.fe_len) {
3595                :                  int winl;
3596                :                  int wins;
3597                :                  int win;
3598                :                  int offs;
3599                :
3600                :                  /* we can't allocate as much as normalizer wants.
3601                :                   * so, found space must get proper lstart
3602                :                   * to cover original request */
3603        418011  :                  BUG_ON(ac->ac_g_ex.fe_logical > ac->ac_o_ex.fe_logical);
3604        418001  :                  BUG_ON(ac->ac_g_ex.fe_len < ac->ac_o_ex.fe_len);
3605                :
3606                :                  /* we're limited by original request in that
3607                :                   * logical block must be covered any way
3608                :                   * winl is window we can move our chunk within */
```

```
3609     418002 :                winl = ac->ac_o_ex.fe_logical - ac->ac_g_ex.fe_logical;
3610            :
3611            :                /* also, we should cover whole original request */
3612     418002 :                wins = ac->ac_b_ex.fe_len - ac->ac_o_ex.fe_len;
3613            :
3614            :                /* the smallest one defines real window */
3615     418002 :                win = min(winl, wins);
3616            :
3617     418002 :                offs = ac->ac_o_ex.fe_logical % ac->ac_b_ex.fe_len;
3618     418002 :                if (offs && offs < win)
3619          2 :                        win = offs;
3620            :
3621     418002 :                ac->ac_b_ex.fe_logical = ac->ac_o_ex.fe_logical - win;
3622     418002 :                BUG_ON(ac->ac_o_ex.fe_logical < ac->ac_b_ex.fe_logical);
3623     418009 :                BUG_ON(ac->ac_o_ex.fe_len > ac->ac_b_ex.fe_len);
3624            :        }
3625            :
3626            :        /* preallocation can change ac_b_ex, thus we store actually
3627            :         * allocated blocks for history */
3628    9124288 :        ac->ac_f_ex = ac->ac_b_ex;
3629            :
3630    9124288 :        pa->pa_lstart = ac->ac_b_ex.fe_logical;
3631   18248576 :        pa->pa_pstart = ext4_grp_offs_to_block(sb, &ac->ac_b_ex);
3632    9124288 :        pa->pa_len = ac->ac_b_ex.fe_len;
3633    9124288 :        pa->pa_free = pa->pa_len;
3634    9124288 :        atomic_set(&pa->pa_count, 1);
3635    9124288 :        spin_lock_init(&pa->pa_lock);
3636    9124288 :        INIT_LIST_HEAD(&pa->pa_inode_list);
3637    9124288 :        INIT_LIST_HEAD(&pa->pa_group_list);
3638    9124288 :        pa->pa_deleted = 0;
3639    9124288 :        pa->pa_type = MB_INODE_PA;
3640            :
3641            :        mb_debug("new inode pa %p: %llu/%u for %u\n", pa,
3642            :                        pa->pa_pstart, pa->pa_len, pa->pa_lstart);
3643            :        trace_ext4_mb_new_inode_pa(ac, pa);
3644            :
3645    9123728 :        ext4_mb_use_inode_pa(ac, pa);
3646    9123710 :        atomic_add(pa->pa_free, &EXT4_SB(sb)->s_mb_preallocated);
3647            :
3648   18248840 :        ei = EXT4_I(ac->ac_inode);
3649   18248840 :        grp = ext4_get_group_info(sb, ac->ac_b_ex.fe_group);
3650            :
3651    9124420 :        pa->pa_obj_lock = &ei->i_prealloc_lock;
3652    9124420 :        pa->pa_inode = ac->ac_inode;
3653            :
3654    9124420 :        ext4_lock_group(sb, ac->ac_b_ex.fe_group);
3655    9124160 :        list_add(&pa->pa_group_list, &grp->bb_prealloc_list);
3656    9124312 :        ext4_unlock_group(sb, ac->ac_b_ex.fe_group);
3657            :
3658    9124407 :        spin_lock(pa->pa_obj_lock);
3659    9124163 :        list_add_rcu(&pa->pa_inode_list, &ei->i_prealloc_list);
3660    9124167 :        spin_unlock(pa->pa_obj_lock);
3661            :
3662    9124036 :        return 0;
3663            : }
3664            :
3665            : /*
3666            :  * creates new preallocated space for locality group inodes belongs to
3667            :  */
3668            : static noinline_for_stack int
3669            : ext4_mb_new_group_pa(struct ext4_allocation_context *ac)
3670      68159 : {
3671      68159 :        struct super_block *sb = ac->ac_sb;
3672            :        struct ext4_locality_group *lg;
3673            :        struct ext4_prealloc_space *pa;
3674            :        struct ext4_group_info *grp;
3675            :
3676            :        /* preallocate only when found space is larger then requested */
3677      68159 :        BUG_ON(ac->ac_o_ex.fe_len >= ac->ac_b_ex.fe_len);
3678      68159 :        BUG_ON(ac->ac_status != AC_STATUS_FOUND);
3679      68159 :        BUG_ON(!S_ISREG(ac->ac_inode->i_mode));
```

```
3680                :
3681           68159 :          BUG_ON(ext4_pspace_cachep == NULL);
3682           68159 :          pa = kmem_cache_alloc(ext4_pspace_cachep, GFP_NOFS);
3683           68159 :          if (pa == NULL)
3684               0 :                  return -ENOMEM;
3685                :
3686                :          /* preallocation can change ac_b_ex, thus we store actually
3687                :           * allocated blocks for history */
3688           68159 :          ac->ac_f_ex = ac->ac_b_ex;
3689                :
3690          136318 :          pa->pa_pstart = ext4_grp_offs_to_block(sb, &ac->ac_b_ex);
3691           68159 :          pa->pa_lstart = pa->pa_pstart;
3692           68159 :          pa->pa_len = ac->ac_b_ex.fe_len;
3693           68159 :          pa->pa_free = pa->pa_len;
3694           68159 :          atomic_set(&pa->pa_count, 1);
3695           68159 :          spin_lock_init(&pa->pa_lock);
3696           68159 :          INIT_LIST_HEAD(&pa->pa_inode_list);
3697           68159 :          INIT_LIST_HEAD(&pa->pa_group_list);
3698           68159 :          pa->pa_deleted = 0;
3699           68159 :          pa->pa_type = MB_GROUP_PA;
3700                :
3701                :          mb_debug("new group pa %p: %llu/%u for %u\n", pa,
3702                :                          pa->pa_pstart, pa->pa_len, pa->pa_lstart);
3703                :          trace_ext4_mb_new_group_pa(ac, pa);
3704                :
3705                :          ext4_mb_use_group_pa(ac, pa);
3706           68159 :          atomic_add(pa->pa_free, &EXT4_SB(sb)->s_mb_preallocated);
3707                :
3708          136318 :          grp = ext4_get_group_info(sb, ac->ac_b_ex.fe_group);
3709           68159 :          lg = ac->ac_lg;
3710           68159 :          BUG_ON(lg == NULL);
3711                :
3712           68159 :          pa->pa_obj_lock = &lg->lg_prealloc_lock;
3713           68159 :          pa->pa_inode = NULL;
3714                :
3715           68159 :          ext4_lock_group(sb, ac->ac_b_ex.fe_group);
3716           68159 :          list_add(&pa->pa_group_list, &grp->bb_prealloc_list);
3717           68159 :          ext4_unlock_group(sb, ac->ac_b_ex.fe_group);
3718                :
3719                :          /*
3720                :           * We will later add the new pa to the right bucket
3721                :           * after updating the pa_free in ext4_mb_release_context
3722                :           */
3723           68159 :          return 0;
3724                : }
3725                :
3726                : static int ext4_mb_new_preallocation(struct ext4_allocation_context *ac)
3727                : {
3728                :          int err;
3729                :
3730         9192364 :          if (ac->ac_flags & EXT4_MB_HINT_GROUP_ALLOC)
3731           68159 :                  err = ext4_mb_new_group_pa(ac);
3732                :          else
3733         9124205 :                  err = ext4_mb_new_inode_pa(ac);
3734         9192163 :          return err;
3735                : }
3736                :
3737                : /*
3738                :  * finds all unused blocks in on-disk bitmap, frees them in
3739                :  * in-core bitmap and buddy.
3740                :  * @pa must be unlinked from inode and group lists, so that
3741                :  * nobody else can find/use it.
3742                :  * the caller MUST hold group/inode locks.
3743                :  * TODO: optimize the case when there are no in-core structures yet
3744                :  */
3745                : static noinline_for_stack int
3746                : ext4_mb_release_inode_pa(struct ext4_buddy *e4b, struct buffer_head *bitmap_bh,
3747                :                          struct ext4_prealloc_space *pa,
3748                :                          struct ext4_allocation_context *ac)
3749         7845381 : {
3750         7845381 :          struct super_block *sb = e4b->bd_sb;
```

```
3751  7845381 :         struct ext4_sb_info *sbi = EXT4_SB(sb);
3752          :         unsigned int end;
3753          :         unsigned int next;
3754          :         ext4_group_t group;
3755          :         ext4_grpblk_t bit;
3756          :         unsigned long long grp_blk_start;
3757          :         sector_t start;
3758  7845381 :         int err = 0;
3759  7845381 :         int free = 0;
3760          :
3761  7845381 :         BUG_ON(pa->pa_deleted == 0);
3762  7845621 :         ext4_get_group_no_and_offset(sb, pa->pa_pstart, &group, &bit);
3763  7846236 :         grp_blk_start = pa->pa_pstart - bit;
3764  7846236 :         BUG_ON(group != e4b->bd_group && pa->pa_len != 0);
3765  7846285 :         end = bit + pa->pa_len;
3766          :
3767  7846285 :         if (ac) {
3768  7845853 :                 ac->ac_sb = sb;
3769  7845853 :                 ac->ac_inode = pa->pa_inode;
3770  7845853 :                 ac->ac_op = EXT4_MB_HISTORY_DISCARD;
3771          :         }
3772          :
3773 15724617 :         while (bit < end) {
3774  7884446 :                 bit = mb_find_next_zero_bit(bitmap_bh->b_data, end, bit);
3775  7885265 :                 if (bit >= end)
3776     7065 :                         break;
3777 15754977 :                 next = mb_find_next_bit(bitmap_bh->b_data, end, bit);
3778 15753554 :                 start = group * EXT4_BLOCKS_PER_GROUP(sb) + bit +
3779          :                                 le32_to_cpu(sbi->s_es->s_first_data_block);
3780          :                 mb_debug("    free preallocated %u/%u in group %u\n",
3781          :                                 (unsigned) start, (unsigned) next - bit,
3782          :                                 (unsigned) group);
3783  7876777 :                 free += next - bit;
3784          :
3785  7876777 :                 if (ac) {
3786  7876938 :                         ac->ac_b_ex.fe_group = group;
3787  7876938 :                         ac->ac_b_ex.fe_start = bit;
3788  7876938 :                         ac->ac_b_ex.fe_len = next - bit;
3789  7876938 :                         ac->ac_b_ex.fe_logical = 0;
3790  7876938 :                         ext4_mb_store_history(ac);
3791          :                 }
3792          :
3793  7877261 :                 trace_ext4_mb_release_inode_pa(ac, pa, grp_blk_start + bit,
3794          :                                                next - bit);
3795  7877886 :                 mb_free_blocks(pa->pa_inode, e4b, bit, next - bit);
3796  7878332 :                 bit = next + 1;
3797          :         }
3798  7847236 :         if (free != pa->pa_free) {
3799        0 :                 printk(KERN_CRIT "pa %p: logic %lu, phys. %lu, len %lu\n",
3800          :                         pa, (unsigned long) pa->pa_lstart,
3801          :                         (unsigned long) pa->pa_pstart,
3802          :                         (unsigned long) pa->pa_len);
3803        0 :                 ext4_grp_locked_error(sb, group,
3804          :                                       __func__, "free %u, pa_free %u",
3805          :                                       free, pa->pa_free);
3806          :                 /*
3807          :                  * pa is already deleted so we use the value obtained
3808          :                  * from the bitmap and continue.
3809          :                  */
3810          :         }
3811  7847236 :         atomic_add(free, &sbi->s_mb_discarded);
3812          :
3813  7846597 :         return err;
3814          : }
3815          :
3816          : static noinline_for_stack int
3817          : ext4_mb_release_group_pa(struct ext4_buddy *e4b,
3818          :                                 struct ext4_prealloc_space *pa,
3819          :                                 struct ext4_allocation_context *ac)
3820    12160 : {
3821    12160 :         struct super_block *sb = e4b->bd_sb;
```

```
3822                :         ext4_group_t group;
3823                :         ext4_grpblk_t bit;
3824                :
3825           12160 :         if (ac)
3826           12160 :                 ac->ac_op = EXT4_MB_HISTORY_DISCARD;
3827                :
3828                :         trace_ext4_mb_release_group_pa(ac, pa);
3829           12160 :         BUG_ON(pa->pa_deleted == 0);
3830           12160 :         ext4_get_group_no_and_offset(sb, pa->pa_pstart, &group, &bit);
3831           12160 :         BUG_ON(group != e4b->bd_group && pa->pa_len != 0);
3832           12160 :         mb_free_blocks(pa->pa_inode, e4b, bit, pa->pa_len);
3833           12160 :         atomic_add(pa->pa_len, &EXT4_SB(sb)->s_mb_discarded);
3834                :
3835           12160 :         if (ac) {
3836           12160 :                 ac->ac_sb = sb;
3837           12160 :                 ac->ac_inode = NULL;
3838           12160 :                 ac->ac_b_ex.fe_group = group;
3839           12160 :                 ac->ac_b_ex.fe_start = bit;
3840           12160 :                 ac->ac_b_ex.fe_len = pa->pa_len;
3841           12160 :                 ac->ac_b_ex.fe_logical = 0;
3842           12160 :                 ext4_mb_store_history(ac);
3843                :         }
3844                :
3845           12160 :         return 0;
3846                : }
3847                :
3848                : /*
3849                :  * releases all preallocations in given group
3850                :  *
3851                :  * first, we need to decide discard policy:
3852                :  * - when do we discard
3853                :  *   1) ENOSPC
3854                :  * - how many do we discard
3855                :  *   1) how many requested
3856                :  */
3857                : static noinline_for_stack int
3858                : ext4_mb_discard_group_preallocations(struct super_block *sb,
3859                :                                      ext4_group_t group, int needed)
3860              0 : {
3861              0 :         struct ext4_group_info *grp = ext4_get_group_info(sb, group);
3862              0 :         struct buffer_head *bitmap_bh = NULL;
3863                :         struct ext4_prealloc_space *pa, *tmp;
3864                :         struct ext4_allocation_context *ac;
3865                :         struct list_head list;
3866                :         struct ext4_buddy e4b;
3867                :         int err;
3868              0 :         int busy = 0;
3869              0 :         int free = 0;
3870                :
3871                :         mb_debug("discard preallocation for group %u\n", group);
3872                :
3873              0 :         if (list_empty(&grp->bb_prealloc_list))
3874              0 :                 return 0;
3875                :
3876              0 :         bitmap_bh = ext4_read_block_bitmap(sb, group);
3877              0 :         if (bitmap_bh == NULL) {
3878              0 :                 ext4_error(sb, __func__, "Error in reading block "
3879                :                                         "bitmap for %u", group);
3880              0 :                 return 0;
3881                :         }
3882                :
3883              0 :         err = ext4_mb_load_buddy(sb, group, &e4b);
3884              0 :         if (err) {
3885              0 :                 ext4_error(sb, __func__, "Error in loading buddy "
3886                :                                         "information for %u", group);
3887                :                 put_bh(bitmap_bh);
3888              0 :                 return 0;
3889                :         }
3890                :
3891              0 :         if (needed == 0)
3892              0 :                 needed = EXT4_BLOCKS_PER_GROUP(sb) + 1;
```

```
3893                 :
3894                 :         INIT_LIST_HEAD(&list);
3895               0 :         ac = kmem_cache_alloc(ext4_ac_cachep, GFP_NOFS);
3896               0 :         if (ac)
3897               0 :                 ac->ac_sb = sb;
3898               0 : repeat:
3899                 :         ext4_lock_group(sb, group);
3900               0 :         list_for_each_entry_safe(pa, tmp,
3901                 :                                  &grp->bb_prealloc_list, pa_group_list) {
3902               0 :                 spin_lock(&pa->pa_lock);
3903               0 :                 if (atomic_read(&pa->pa_count)) {
3904               0 :                         spin_unlock(&pa->pa_lock);
3905               0 :                         busy = 1;
3906               0 :                         continue;
3907                 :                 }
3908               0 :                 if (pa->pa_deleted) {
3909               0 :                         spin_unlock(&pa->pa_lock);
3910                 :                         continue;
3911                 :                 }
3912                 :
3913                 :                 /* seems this one can be freed ... */
3914               0 :                 pa->pa_deleted = 1;
3915                 :
3916                 :                 /* we can trust pa_free ... */
3917               0 :                 free += pa->pa_free;
3918                 :
3919               0 :                 spin_unlock(&pa->pa_lock);
3920                 :
3921               0 :                 list_del(&pa->pa_group_list);
3922               0 :                 list_add(&pa->u.pa_tmp_list, &list);
3923                 :         }
3924                 :
3925                 :         /* if we still need more blocks and some PAs were used, try again */
3926               0 :         if (free < needed && busy) {
3927               0 :                 busy = 0;
3928                 :                 ext4_unlock_group(sb, group);
3929                 :                 /*
3930                 :                  * Yield the CPU here so that we don't get soft lockup
3931                 :                  * in non preempt case.
3932                 :                  */
3933               0 :                 yield();
3934               0 :                 goto repeat;
3935                 :         }
3936                 :
3937                 :         /* found anything to free? */
3938               0 :         if (list_empty(&list)) {
3939               0 :                 BUG_ON(free != 0);
3940                 :                 goto out;
3941                 :         }
3942                 :
3943                 :         /* now free all selected PAs */
3944               0 :         list_for_each_entry_safe(pa, tmp, &list, u.pa_tmp_list) {
3945                 :
3946                 :                 /* remove from object (inode or locality group) */
3947               0 :                 spin_lock(pa->pa_obj_lock);
3948               0 :                 list_del_rcu(&pa->pa_inode_list);
3949               0 :                 spin_unlock(pa->pa_obj_lock);
3950                 :
3951               0 :                 if (pa->pa_type == MB_GROUP_PA)
3952               0 :                         ext4_mb_release_group_pa(&e4b, pa, ac);
3953                 :                 else
3954               0 :                         ext4_mb_release_inode_pa(&e4b, bitmap_bh, pa, ac);
3955                 :
3956               0 :                 list_del(&pa->u.pa_tmp_list);
3957               0 :                 call_rcu(&(pa)->u.pa_rcu, ext4_mb_pa_callback);
3958                 :         }
3959                 :
3960               0 : out:
3961                 :         ext4_unlock_group(sb, group);
3962               0 :         if (ac)
3963               0 :                 kmem_cache_free(ext4_ac_cachep, ac);
```

```
3964                0 :          ext4_mb_release_desc(&e4b);
3965                  :          put_bh(bitmap_bh);
3966                0 :          return free;
3967                  : }
3968                  :
3969                  : /*
3970                  :  * releases all non-used preallocated blocks for given inode
3971                  :  *
3972                  :  * It's important to discard preallocations under i_data_sem
3973                  :  * We don't want another block to be served from the prealloc
3974                  :  * space when we are discarding the inode prealloc space.
3975                  :  *
3976                  :  * FIXME!! Make sure it is valid at all the call sites
3977                  :  */
3978                  : void ext4_discard_preallocations(struct inode *inode)
3979         38286672 : {
3980         38286672 :          struct ext4_inode_info *ei = EXT4_I(inode);
3981         38286672 :          struct super_block *sb = inode->i_sb;
3982         38286672 :          struct buffer_head *bitmap_bh = NULL;
3983                  :          struct ext4_prealloc_space *pa, *tmp;
3984                  :          struct ext4_allocation_context *ac;
3985         38286672 :          ext4_group_t group = 0;
3986                  :          struct list_head list;
3987                  :          struct ext4_buddy e4b;
3988                  :          int err;
3989                  :
3990         38286672 :          if (!S_ISREG(inode->i_mode)) {
3991                  :                  /*BUG_ON(!list_empty(&ei->i_prealloc_list));*/
3992          1353294 :                  return;
3993                  :          }
3994                  :
3995                  :          mb_debug("discard preallocation for inode %lu\n", inode->i_ino);
3996                  :          trace_ext4_discard_preallocations(inode);
3997                  :
3998                  :          INIT_LIST_HEAD(&list);
3999                  :
4000         36938781 :          ac = kmem_cache_alloc(ext4_ac_cachep, GFP_NOFS);
4001         36916305 :          if (ac) {
4002         36924586 :                  ac->ac_sb = sb;
4003         36924586 :                  ac->ac_inode = inode;
4004                  :          }
4005         36916305 : repeat:
4006                  :          /* first, collect all pa's in the inode */
4007         36916305 :          spin_lock(&ei->i_prealloc_lock);
4008         81768927 :          while (!list_empty(&ei->i_prealloc_list)) {
4009          7843934 :                  pa = list_entry(ei->i_prealloc_list.next,
4010                  :                                  struct ext4_prealloc_space, pa_inode_list);
4011          7843934 :                  BUG_ON(pa->pa_obj_lock != &ei->i_prealloc_lock);
4012          7846021 :                  spin_lock(&pa->pa_lock);
4013         15683154 :                  if (atomic_read(&pa->pa_count)) {
4014                  :                          /* this shouldn't happen often - nobody should
4015                  :                           * use preallocation while we're discarding it */
4016                0 :                          spin_unlock(&pa->pa_lock);
4017                0 :                          spin_unlock(&ei->i_prealloc_lock);
4018                0 :                          printk(KERN_ERR "uh-oh! used pa while discarding\n");
4019                0 :                          WARN_ON(1);
4020                0 :                          schedule_timeout_uninterruptible(HZ);
4021                0 :                          goto repeat;
4022                  :
4023                  :                  }
4024          7841577 :                  if (pa->pa_deleted == 0) {
4025          7841577 :                          pa->pa_deleted = 1;
4026          7841577 :                          spin_unlock(&pa->pa_lock);
4027          7846490 :                          list_del_rcu(&pa->pa_inode_list);
4028          7846490 :                          list_add(&pa->u.pa_tmp_list, &list);
4029                  :                          continue;
4030                  :                  }
4031                  :
4032                  :                  /* someone is deleting pa right now */
4033                0 :                  spin_unlock(&pa->pa_lock);
4034                0 :                  spin_unlock(&ei->i_prealloc_lock);
```

```
4035                :
4036                :                 /* we have to wait here because pa_deleted
4037                :                  * doesn't mean pa is already unlinked from
4038                :                  * the list. as we might be called from
4039                :                  * ->clear_inode() the inode will get freed
4040                :                  * and concurrent thread which is unlinking
4041                :                  * pa from inode's list may access already
4042                :                  * freed memory, bad-bad-bad */
4043                :
4044                :                 /* XXX: if this happens too often, we can
4045                :                  * add a flag to force wait only in case
4046                :                  * of ->clear_inode(), but not in case of
4047                :                  * regular truncate */
4048              0 :                 schedule_timeout_uninterruptible(HZ);
4049              0 :                 goto repeat;
4050                :         }
4051       36973805 :         spin_unlock(&ei->i_prealloc_lock);
4052                :
4053       44782158 :         list_for_each_entry_safe(pa, tmp, &list, u.pa_tmp_list) {
4054        7841687 :                 BUG_ON(pa->pa_type != MB_INODE_PA);
4055        7841681 :                 ext4_get_group_no_and_offset(sb, pa->pa_pstart, &group, NULL);
4056                :
4057        7842563 :                 err = ext4_mb_load_buddy(sb, group, &e4b);
4058        7845965 :                 if (err) {
4059              0 :                         ext4_error(sb, __func__, "Error in loading buddy "
4060                :                                         "information for %u", group);
4061              0 :                         continue;
4062                :                 }
4063                :
4064        7845965 :                 bitmap_bh = ext4_read_block_bitmap(sb, group);
4065        7841031 :                 if (bitmap_bh == NULL) {
4066              0 :                         ext4_error(sb, __func__, "Error in reading block "
4067                :                                         "bitmap for %u", group);
4068              0 :                         ext4_mb_release_desc(&e4b);
4069              0 :                         continue;
4070                :                 }
4071                :
4072        7841031 :                 ext4_lock_group(sb, group);
4073        7845892 :                 list_del(&pa->pa_group_list);
4074        7846257 :                 ext4_mb_release_inode_pa(&e4b, bitmap_bh, pa, ac);
4075        7846574 :                 ext4_unlock_group(sb, group);
4076                :
4077        7846577 :                 ext4_mb_release_desc(&e4b);
4078                :                 put_bh(bitmap_bh);
4079                :
4080        7846574 :                 list_del(&pa->u.pa_tmp_list);
4081        7846479 :                 call_rcu(&(pa)->u.pa_rcu, ext4_mb_pa_callback);
4082                :         }
4083       36940471 :         if (ac)
4084       36947030 :                 kmem_cache_free(ext4_ac_cachep, ac);
4085                : }
4086                :
4087                : /*
4088                :  * finds all preallocated spaces and return blocks being freed to them
4089                :  * if preallocated space becomes full (no block is used from the space)
4090                :  * then the function frees space in buddy
4091                :  * XXX: at the moment, truncate (which is the only way to free blocks)
4092                :  * discards all preallocations
4093                :  */
4094                : static void ext4_mb_return_to_preallocation(struct inode *inode,
4095                :                                             struct ext4_buddy *e4b,
4096                :                                             sector_t block, int count)
4097                : {
4098              0 :         BUG_ON(!list_empty(&EXT4_I(inode)->i_prealloc_list));
4099                : }
4100                : #ifdef MB_DEBUG
4101                : static void ext4_mb_show_ac(struct ext4_allocation_context *ac)
4102                : {
4103                :         struct super_block *sb = ac->ac_sb;
4104                :         ext4_group_t ngroups, i;
4105                :
```

```
4106                 :         printk(KERN_ERR "EXT4-fs: Can't allocate:"
4107                 :                         " Allocation context details:\n");
4108                 :         printk(KERN_ERR "EXT4-fs: status %d flags %d\n",
4109                 :                         ac->ac_status, ac->ac_flags);
4110                 :         printk(KERN_ERR "EXT4-fs: orig %lu/%lu/%lu@%lu, goal %lu/%lu/%lu@%lu, "
4111                 :                         "best %lu/%lu/%lu@%lu cr %d\n",
4112                 :                         (unsigned long)ac->ac_o_ex.fe_group,
4113                 :                         (unsigned long)ac->ac_o_ex.fe_start,
4114                 :                         (unsigned long)ac->ac_o_ex.fe_len,
4115                 :                         (unsigned long)ac->ac_o_ex.fe_logical,
4116                 :                         (unsigned long)ac->ac_g_ex.fe_group,
4117                 :                         (unsigned long)ac->ac_g_ex.fe_start,
4118                 :                         (unsigned long)ac->ac_g_ex.fe_len,
4119                 :                         (unsigned long)ac->ac_g_ex.fe_logical,
4120                 :                         (unsigned long)ac->ac_b_ex.fe_group,
4121                 :                         (unsigned long)ac->ac_b_ex.fe_start,
4122                 :                         (unsigned long)ac->ac_b_ex.fe_len,
4123                 :                         (unsigned long)ac->ac_b_ex.fe_logical,
4124                 :                         (int)ac->ac_criteria);
4125                 :         printk(KERN_ERR "EXT4-fs: %lu scanned, %d found\n", ac->ac_ex_scanned,
4126                 :                 ac->ac_found);
4127                 :         printk(KERN_ERR "EXT4-fs: groups: \n");
4128                 :         ngroups = ext4_get_groups_count(sb);
4129                 :         for (i = 0; i < ngroups; i++) {
4130                 :                 struct ext4_group_info *grp = ext4_get_group_info(sb, i);
4131                 :                 struct ext4_prealloc_space *pa;
4132                 :                 ext4_grpblk_t start;
4133                 :                 struct list_head *cur;
4134                 :                 ext4_lock_group(sb, i);
4135                 :                 list_for_each(cur, &grp->bb_prealloc_list) {
4136                 :                         pa = list_entry(cur, struct ext4_prealloc_space,
4137                 :                                         pa_group_list);
4138                 :                         spin_lock(&pa->pa_lock);
4139                 :                         ext4_get_group_no_and_offset(sb, pa->pa_pstart,
4140                 :                                                      NULL, &start);
4141                 :                         spin_unlock(&pa->pa_lock);
4142                 :                         printk(KERN_ERR "PA:%lu:%d:%u \n", i,
4143                 :                                                        start, pa->pa_len);
4144                 :                 }
4145                 :                 ext4_unlock_group(sb, i);
4146                 :
4147                 :                 if (grp->bb_free == 0)
4148                 :                         continue;
4149                 :                 printk(KERN_ERR "%lu: %d/%d \n",
4150                 :                        i, grp->bb_free, grp->bb_fragments);
4151                 :         }
4152                 :         printk(KERN_ERR "\n");
4153                 : }
4154                 : #else
4155                 : static inline void ext4_mb_show_ac(struct ext4_allocation_context *ac)
4156                 : {
4157                 :         return;
4158                 : }
4159                 : #endif
4160                 :
4161                 : /*
4162                 :  * We use locality group preallocation for small size file. The size of the
4163                 :  * file is determined by the current size or the resulting size after
4164                 :  * allocation which ever is larger
4165                 :  *
4166                 :  * One can tune this size via /sys/fs/ext4/<partition>/mb_stream_req
4167                 :  */
4168                 : static void ext4_mb_group_or_file(struct ext4_allocation_context *ac)
4169                 : {
4170       438345656 :         struct ext4_sb_info *sbi = EXT4_SB(ac->ac_sb);
4171       219172828 :         int bsbits = ac->ac_sb->s_blocksize_bits;
4172                 :         loff_t size, isize;
4173                 :
4174       219172828 :         if (!(ac->ac_flags & EXT4_MB_HINT_DATA))
4175                 :                 return;
4176                 :
```

```
4177  219561937 :          size = ac->ac_o_ex.fe_logical + ac->ac_o_ex.fe_len;
4178  440115388 :          isize = i_size_read(ac->ac_inode) >> bsbits;
4179  220553451 :          size = max(size, isize);
4180            :
4181            :          /* don't use group allocation for large files */
4182  220553451 :          if (size >= sbi->s_mb_stream_request)
4183            :                  return;
4184            :
4185   21597316 :          if (unlikely(ac->ac_flags & EXT4_MB_HINT_GOAL_ONLY))
4186            :                  return;
4187            :
4188   21596549 :          BUG_ON(ac->ac_lg != NULL);
4189            :          /*
4190            :           * locality group prealloc space are per cpu. The reason for having
4191            :           * per cpu locality group is to reduce the contention between block
4192            :           * request from multiple CPUs.
4193            :           */
4194   21600699 :          ac->ac_lg = per_cpu_ptr(sbi->s_locality_groups, raw_smp_processor_id());
4195            :
4196            :          /* we're going to use group allocation */
4197   21600699 :          ac->ac_flags |= EXT4_MB_HINT_GROUP_ALLOC;
4198            :
4199            :          /* serialize all allocations in the group */
4200   21600699 :          mutex_lock(&ac->ac_lg->lg_mutex);
4201            : }
4202            :
4203            : static noinline_for_stack int
4204            : ext4_mb_initialize_context(struct ext4_allocation_context *ac,
4205            :                                  struct ext4_allocation_request *ar)
4206  219503640 : {
4207  219503640 :          struct super_block *sb = ar->inode->i_sb;
4208  219503640 :          struct ext4_sb_info *sbi = EXT4_SB(sb);
4209  219503640 :          struct ext4_super_block *es = sbi->s_es;
4210            :          ext4_group_t group;
4211            :          unsigned int len;
4212            :          ext4_fsblk_t goal;
4213            :          ext4_grpblk_t block;
4214            :
4215            :          /* we can't allocate > group size */
4216  219503640 :          len = ar->len;
4217            :
4218            :          /* just a dirty hack to filter too big requests  */
4219  219503640 :          if (len >= EXT4_BLOCKS_PER_GROUP(sb) - 10)
4220          0 :                  len = EXT4_BLOCKS_PER_GROUP(sb) - 10;
4221            :
4222            :          /* start searching from the goal */
4223  219503640 :          goal = ar->goal;
4224  438714837 :          if (goal < le32_to_cpu(es->s_first_data_block) ||
4225            :                          goal >= ext4_blocks_count(es))
4226         13 :                  goal = le32_to_cpu(es->s_first_data_block);
4227  219503640 :          ext4_get_group_no_and_offset(sb, goal, &group, &block);
4228            :
4229            :          /* set up allocation goals */
4230            :          memset(ac, 0, sizeof(struct ext4_allocation_context));
4231  219172828 :          ac->ac_b_ex.fe_logical = ar->logical;
4232  219172828 :          ac->ac_status = AC_STATUS_CONTINUE;
4233  219172828 :          ac->ac_sb = sb;
4234  219172828 :          ac->ac_inode = ar->inode;
4235  219172828 :          ac->ac_o_ex.fe_logical = ar->logical;
4236  219172828 :          ac->ac_o_ex.fe_group = group;
4237  219172828 :          ac->ac_o_ex.fe_start = block;
4238  219172828 :          ac->ac_o_ex.fe_len = len;
4239  219172828 :          ac->ac_g_ex.fe_logical = ar->logical;
4240  219172828 :          ac->ac_g_ex.fe_group = group;
4241  219172828 :          ac->ac_g_ex.fe_start = block;
4242  219172828 :          ac->ac_g_ex.fe_len = len;
4243  219172828 :          ac->ac_flags = ar->flags;
4244            :
4245            :          /* we have to define context: we'll we work with a file or
4246            :           * locality group. this is a policy, actually */
4247            :          ext4_mb_group_or_file(ac);
```

```
4248                :
4249                :         mb_debug("init ac: %u blocks @ %u, goal %u, flags %x, 2^%d, "
4250                :                         "left: %u/%u, right %u/%u to %swritable\n",
4251                :                         (unsigned) ar->len, (unsigned) ar->logical,
4252                :                         (unsigned) ar->goal, ac->ac_flags, ac->ac_2order,
4253                :                         (unsigned) ar->lleft, (unsigned) ar->pleft,
4254                :                         (unsigned) ar->lright, (unsigned) ar->pright,
4255                :                         atomic_read(&ar->inode->i_writecount) ? "" : "non-");
4256        220174869 :         return 0;
4257                :
4258                : }
4259                :
4260                : static noinline_for_stack void
4261                : ext4_mb_discard_lg_preallocations(struct super_block *sb,
4262                :                                   struct ext4_locality_group *lg,
4263                :                                   int order, int total_entries)
4264             3040 : {
4265             3040 :         ext4_group_t group = 0;
4266                :         struct ext4_buddy e4b;
4267                :         struct list_head discard_list;
4268                :         struct ext4_prealloc_space *pa, *tmp;
4269                :         struct ext4_allocation_context *ac;
4270                :
4271                :         mb_debug("discard locality group preallocation\n");
4272                :
4273                :         INIT_LIST_HEAD(&discard_list);
4274             3040 :         ac = kmem_cache_alloc(ext4_ac_cachep, GFP_NOFS);
4275             3040 :         if (ac)
4276             3040 :                 ac->ac_sb = sb;
4277                :
4278             3040 :         spin_lock(&lg->lg_prealloc_lock);
4279            25310 :         list_for_each_entry_rcu(pa, &lg->lg_prealloc_list[order],
4280                :                                                 pa_inode_list) {
4281            12655 :                 spin_lock(&pa->pa_lock);
4282            25310 :                 if (atomic_read(&pa->pa_count)) {
4283                :                         /*
4284                :                          * This is the pa that we just used
4285                :                          * for block allocation. So don't
4286                :                          * free that
4287                :                          */
4288              495 :                         spin_unlock(&pa->pa_lock);
4289                :                         continue;
4290                :                 }
4291            12160 :                 if (pa->pa_deleted) {
4292                0 :                         spin_unlock(&pa->pa_lock);
4293                :                         continue;
4294                :                 }
4295                :                 /* only lg prealloc space */
4296            12160 :                 BUG_ON(pa->pa_type != MB_GROUP_PA);
4297                :
4298                :                 /* seems this one can be freed ... */
4299            12160 :                 pa->pa_deleted = 1;
4300            12160 :                 spin_unlock(&pa->pa_lock);
4301                :
4302            12160 :                 list_del_rcu(&pa->pa_inode_list);
4303            12160 :                 list_add(&pa->u.pa_tmp_list, &discard_list);
4304                :
4305            12160 :                 total_entries--;
4306            12160 :                 if (total_entries <= 5) {
4307                :                         /*
4308                :                          * we want to keep only 5 entries
4309                :                          * allowing it to grow to 8. This
4310                :                          * mak sure we don't call discard
4311                :                          * soon for this list.
4312                :                          */
4313             3040 :                         break;
4314                :                 }
4315                :         }
4316             3040 :         spin_unlock(&lg->lg_prealloc_lock);
4317                :
4318            15200 :         list_for_each_entry_safe(pa, tmp, &discard_list, u.pa_tmp_list) {
```

```
4319                :
4320          12160 :                   ext4_get_group_no_and_offset(sb, pa->pa_pstart, &group, NULL);
4321          12160 :                   if (ext4_mb_load_buddy(sb, group, &e4b)) {
4322              0 :                           ext4_error(sb, __func__, "Error in loading buddy "
4323                :                                   "information for %u", group);
4324              0 :                           continue;
4325                :                   }
4326          12160 :                   ext4_lock_group(sb, group);
4327          12160 :                   list_del(&pa->pa_group_list);
4328          12160 :                   ext4_mb_release_group_pa(&e4b, pa, ac);
4329          12160 :                   ext4_unlock_group(sb, group);
4330                :
4331          12160 :                   ext4_mb_release_desc(&e4b);
4332          12160 :                   list_del(&pa->u.pa_tmp_list);
4333          12160 :                   call_rcu(&(pa)->u.pa_rcu, ext4_mb_pa_callback);
4334                :           }
4335           3040 :           if (ac)
4336           3040 :                   kmem_cache_free(ext4_ac_cachep, ac);
4337           3040 : }
4338                :
4339                : /*
4340                :  * We have incremented pa_count. So it cannot be freed at this
4341                :  * point. Also we hold lg_mutex. So no parallel allocation is
4342                :  * possible from this lg. That means pa_free cannot be updated.
4343                :  *
4344                :  * A parallel ext4_mb_discard_group_preallocations is possible.
4345                :  * which can cause the lg_prealloc_list to be updated.
4346                :  */
4347                :
4348                : static void ext4_mb_add_n_trim(struct ext4_allocation_context *ac)
4349                : {
4350      21545650 :           int order, added = 0, lg_prealloc_count = 1;
4351      21545650 :           struct super_block *sb = ac->ac_sb;
4352      21545650 :           struct ext4_locality_group *lg = ac->ac_lg;
4353      21545650 :           struct ext4_prealloc_space *tmp_pa, *pa = ac->ac_pa;
4354                :
4355      43091300 :           order = fls(pa->pa_free) - 1;
4356      21545650 :           if (order > PREALLOC_TB_SIZE - 1)
4357                :                   /* The max size of hash table is PREALLOC_TB_SIZE */
4358              0 :                   order = PREALLOC_TB_SIZE - 1;
4359                :           /* Add the prealloc space to lg */
4360                :           rcu_read_lock();
4361      43269732 :           list_for_each_entry_rcu(tmp_pa, &lg->lg_prealloc_list[order],
4362                :                                   pa_inode_list) {
4363        101681 :                   spin_lock(&tmp_pa->pa_lock);
4364        101681 :                   if (tmp_pa->pa_deleted) {
4365              0 :                           spin_unlock(&tmp_pa->pa_lock);
4366                :                           continue;
4367                :                   }
4368        101681 :                   if (!added && pa->pa_free < tmp_pa->pa_free) {
4369                :                           /* Add to the tail of the previous entry */
4370          3688 :                           list_add_tail_rcu(&pa->pa_inode_list,
4371                :                                             &tmp_pa->pa_inode_list);
4372          3688 :                           added = 1;
4373                :                           /*
4374                :                            * we want to count the total
4375                :                            * number of entries in the list
4376                :                            */
4377                :                   }
4378        101681 :                   spin_unlock(&tmp_pa->pa_lock);
4379        101681 :                   lg_prealloc_count++;
4380                :           }
4381      21520720 :           if (!added)
4382      21522367 :                   list_add_tail_rcu(&pa->pa_inode_list,
4383                :                                     &lg->lg_prealloc_list[order]);
4384                :           rcu_read_unlock();
4385                :
4386                :           /* Now trim the list to be not more than 8 elements */
4387      21526138 :           if (lg_prealloc_count > 8) {
4388          3040 :                   ext4_mb_discard_lg_preallocations(sb, lg,
4389                :                                             order, lg_prealloc_count);
```

```
4390                :                 return;
4391                :         }
4392                :         return ;
4393                : }
4394                :
4395                : /*
4396                :  * release all resource we used in allocation
4397                :  */
4398                : static int ext4_mb_release_context(struct ext4_allocation_context *ac)
4399      220905494 : {
4400      220905494 :         struct ext4_prealloc_space *pa = ac->ac_pa;
4401      220905494 :         if (pa) {
4402      217027748 :                 if (pa->pa_type == MB_GROUP_PA) {
4403                :                         /* see comment in ext4_mb_use_group_pa() */
4404       21597246 :                         spin_lock(&pa->pa_lock);
4405       21592506 :                         pa->pa_pstart += ac->ac_b_ex.fe_len;
4406       21592506 :                         pa->pa_lstart += ac->ac_b_ex.fe_len;
4407       21592506 :                         pa->pa_free -= ac->ac_b_ex.fe_len;
4408       21592506 :                         pa->pa_len -= ac->ac_b_ex.fe_len;
4409       21592506 :                         spin_unlock(&pa->pa_lock);
4410                :                 }
4411                :         }
4412      219636265 :         if (ac->alloc_semp)
4413       13278322 :                 up_read(ac->alloc_semp);
4414      219630085 :         if (pa) {
4415                :                 /*
4416                :                  * We want to add the pa to the right bucket.
4417                :                  * Remove it from the list and while adding
4418                :                  * make sure the list to which we are adding
4419                :                  * doesn't grow big.  We need to release
4420                :                  * alloc_semp before calling ext4_mb_add_n_trim()
4421                :                  */
4422      217120994 :                 if ((pa->pa_type == MB_GROUP_PA) && likely(pa->pa_free)) {
4423       21545159 :                         spin_lock(pa->pa_obj_lock);
4424       21543308 :                         list_del_rcu(&pa->pa_inode_list);
4425       21543308 :                         spin_unlock(pa->pa_obj_lock);
4426                :                         ext4_mb_add_n_trim(ac);
4427                :                 }
4428      217109266 :                 ext4_mb_put_pa(ac, ac->ac_sb, pa);
4429                :         }
4430      221017492 :         if (ac->ac_bitmap_page)
4431       13277001 :                 page_cache_release(ac->ac_bitmap_page);
4432      221019094 :         if (ac->ac_buddy_page)
4433       13274667 :                 page_cache_release(ac->ac_buddy_page);
4434      221024639 :         if (ac->ac_flags & EXT4_MB_HINT_GROUP_ALLOC)
4435       21556948 :                 mutex_unlock(&ac->ac_lg->lg_mutex);
4436                :         ext4_mb_collect_stats(ac);
4437      222634468 :         return 0;
4438                : }
4439                :
4440                : static int ext4_mb_discard_preallocations(struct super_block *sb, int needed)
4441                : {
4442              0 :         ext4_group_t i, ngroups = ext4_get_groups_count(sb);
4443                :         int ret;
4444              0 :         int freed = 0;
4445                :
4446                :         trace_ext4_mb_discard_preallocations(sb, needed);
4447              0 :         for (i = 0; i < ngroups && needed > 0; i++) {
4448              0 :                 ret = ext4_mb_discard_group_preallocations(sb, i, needed);
4449              0 :                 freed += ret;
4450              0 :                 needed -= ret;
4451                :         }
4452                :
4453              0 :         return freed;
4454                : }
4455                :
4456                : /*
4457                :  * Main entry point into mballoc to allocate blocks
4458                :  * it tries to use preallocation first, then falls back
4459                :  * to usual allocation
4460                :  */
```

```
4461                 : ext4_fsblk_t ext4_mb_new_blocks(handle_t *handle,
4462                 :                                 struct ext4_allocation_request *ar, int *errp)
4463       220386244 : {
4464                 :         int freed;
4465       220386244 :         struct ext4_allocation_context *ac = NULL;
4466                 :         struct ext4_sb_info *sbi;
4467                 :         struct super_block *sb;
4468       220386244 :         ext4_fsblk_t block = 0;
4469       220386244 :         unsigned int inquota = 0;
4470       220386244 :         unsigned int reserv_blks = 0;
4471                 :
4472       220386244 :         sb = ar->inode->i_sb;
4473       220386244 :         sbi = EXT4_SB(sb);
4474                 :
4475                 :         trace_ext4_request_blocks(ar);
4476                 :
4477                 :         /*
4478                 :          * For delayed allocation, we could skip the ENOSPC and
4479                 :          * EDQUOT check, as blocks and quotas have been already
4480                 :          * reserved when data being copied into pagecache.
4481                 :          */
4482       441114260 :         if (EXT4_I(ar->inode)->i_delalloc_reserved_flag)
4483        14463860 :                 ar->flags |= EXT4_MB_DELALLOC_RESERVED;
4484                 :         else {
4485                 :                 /* Without delayed allocation we need to verify
4486                 :                  * there is enough free blocks to do block allocation
4487                 :                  * and verify allocation doesn't exceed the quota limits.
4488                 :                  */
4489       206093280 :                 while (ar->len && ext4_claim_free_blocks(sbi, ar->len)) {
4490                 :                         /* let others to free the space */
4491              10 :                         yield();
4492              10 :                         ar->len = ar->len >> 1;
4493                 :                 }
4494       207912828 :                 if (!ar->len) {
4495              10 :                         *errp = -ENOSPC;
4496              10 :                         return 0;
4497                 :                 }
4498       207912818 :                 reserv_blks = ar->len;
4499       622757939 :                 while (ar->len && vfs_dq_alloc_block(ar->inode, ar->len)) {
4500               0 :                         ar->flags |= EXT4_MB_HINT_NOPREALLOC;
4501               0 :                         ar->len--;
4502                 :                 }
4503       206574015 :                 inquota = ar->len;
4504       206574015 :                 if (ar->len == 0) {
4505               0 :                         *errp = -EDQUOT;
4506               0 :                         goto out3;
4507                 :                 }
4508                 :         }
4509                 :
4510       221037875 :         ac = kmem_cache_alloc(ext4_ac_cachep, GFP_NOFS);
4511       219254427 :         if (!ac) {
4512               0 :                 ar->len = 0;
4513               0 :                 *errp = -ENOMEM;
4514               0 :                 goto out1;
4515                 :         }
4516                 :
4517       219254427 :         *errp = ext4_mb_initialize_context(ac, ar);
4518       221760775 :         if (*errp) {
4519               0 :                 ar->len = 0;
4520               0 :                 goto out2;
4521                 :         }
4522                 :
4523       221760775 :         ac->ac_op = EXT4_MB_HISTORY_PREALLOC;
4524       221760775 :         if (!ext4_mb_use_preallocated(ac)) {
4525        13277609 :                 ac->ac_op = EXT4_MB_HISTORY_ALLOC;
4526        13277609 :                 ext4_mb_normalize_request(ac, ar);
4527        13279387 : repeat:
4528                 :                 /* allocate space in core */
4529        13279387 :                 ext4_mb_regular_allocator(ac);
4530                 :
4531                 :                 /* as we've just preallocated more space than
```

```
4532                :                  * user requested orinally, we store allocated
4533                :                  * space in a special descriptor */
4534         13280118 :                 if (ac->ac_status == AC_STATUS_FOUND &&
4535                :                                 ac->ac_o_ex.fe_len < ac->ac_b_ex.fe_len)
4536                :                         ext4_mb_new_preallocation(ac);
4537                :         }
4538        222641370 :         if (likely(ac->ac_status == AC_STATUS_FOUND)) {
4539        221146678 :                 *errp = ext4_mb_mark_diskspace_used(ac, handle, reserv_blks);
4540        221010567 :                 if (*errp ==  -EAGAIN) {
4541                :                         /*
4542                :                          * drop the reference that we took
4543                :                          * in ext4_mb_use_best_found
4544                :                          */
4545                0 :                         ext4_mb_release_context(ac);
4546                0 :                         ac->ac_b_ex.fe_group = 0;
4547                0 :                         ac->ac_b_ex.fe_start = 0;
4548                0 :                         ac->ac_b_ex.fe_len = 0;
4549                0 :                         ac->ac_status = AC_STATUS_CONTINUE;
4550                0 :                         goto repeat;
4551        221010567 :                 } else if (*errp) {
4552                0 :                         ac->ac_b_ex.fe_len = 0;
4553                0 :                         ar->len = 0;
4554                :                         ext4_mb_show_ac(ac);
4555                :                 } else {
4556        442021134 :                         block = ext4_grp_offs_to_block(sb, &ac->ac_b_ex);
4557        221010567 :                         ar->len = ac->ac_b_ex.fe_len;
4558                :                 }
4559                :         } else {
4560                0 :                 freed  = ext4_mb_discard_preallocations(sb, ac->ac_o_ex.fe_len);
4561                0 :                 if (freed)
4562                0 :                         goto repeat;
4563                0 :                 *errp = -ENOSPC;
4564                0 :                 ac->ac_b_ex.fe_len = 0;
4565                0 :                 ar->len = 0;
4566                :                 ext4_mb_show_ac(ac);
4567                :         }
4568                :
4569        221010567 :         ext4_mb_release_context(ac);
4570                :
4571        222060942 : out2:
4572        222060942 :         kmem_cache_free(ext4_ac_cachep, ac);
4573        219956965 : out1:
4574        219956965 :         if (inquota && ar->len < inquota)
4575                0 :                 vfs_dq_free_block(ar->inode, inquota - ar->len);
4576        219956965 : out3:
4577        219956965 :         if (!ar->len) {
4578                0 :                 if (!EXT4_I(ar->inode)->i_delalloc_reserved_flag)
4579                :                         /* release all the reserved blocks if non delalloc */
4580                0 :                         percpu_counter_sub(&sbi->s_dirtyblocks_counter,
4581                :                                                 reserv_blks);
4582                :         }
4583                :
4584                :         trace_ext4_allocate_blocks(ar, (unsigned long long)block);
4585                :
4586        220231602 :         return block;
4587                : }
4588                :
4589                : /*
4590                :  * We can merge two free data extents only if the physical blocks
4591                :  * are contiguous, AND the extents were freed by the same transaction,
4592                :  * AND the blocks are associated with the same group.
4593                :  */
4594                : static int can_merge(struct ext4_free_data *entry1,
4595                :                         struct ext4_free_data *entry2)
4596                : {
4597          4703627 :         if ((entry1->t_tid == entry2->t_tid) &&
4598                :             (entry1->group == entry2->group) &&
4599                :             ((entry1->start_blk + entry1->count) == entry2->start_blk))
4600           662042 :                 return 1;
4601          4041585 :         return 0;
4602                : }
```

```
4603                :
4604                : static noinline_for_stack int
4605                : ext4_mb_free_metadata(handle_t *handle, struct ext4_buddy *e4b,
4606                :                       struct ext4_free_data *new_entry)
4607     3896279    : {
4608                :         ext4_grpblk_t block;
4609                :         struct ext4_free_data *entry;
4610     3896279    :         struct ext4_group_info *db = e4b->bd_info;
4611     3896279    :         struct super_block *sb = e4b->bd_sb;
4612     3896279    :         struct ext4_sb_info *sbi = EXT4_SB(sb);
4613     3896279    :         struct rb_node **n = &db->bb_free_root.rb_node, *node;
4614     3896279    :         struct rb_node *parent = NULL, *new_node;
4615                :
4616     3896279    :         BUG_ON(!ext4_handle_valid(handle));
4617     3896279    :         BUG_ON(e4b->bd_bitmap_page == NULL);
4618     3896279    :         BUG_ON(e4b->bd_buddy_page == NULL);
4619                :
4620     3896279    :         new_node = &new_entry->node;
4621     3896279    :         block = new_entry->start_blk;
4622                :
4623     3896279    :         if (!*n) {
4624                :                 /* first free block exent. We need to
4625                :                    protect buddy cache from being freed,
4626                :                  * otherwise we'll refresh it from
4627                :                  * on-disk bitmap and lose not-yet-available
4628                :                  * blocks */
4629      895564    :                 page_cache_get(e4b->bd_buddy_page);
4630      895562    :                 page_cache_get(e4b->bd_bitmap_page);
4631                :         }
4632    11419298    :         while (*n) {
4633     7523019    :                 parent = *n;
4634     7523019    :                 entry = rb_entry(parent, struct ext4_free_data, node);
4635     7523019    :                 if (block < entry->start_blk)
4636     3996460    :                         n = &(*n)->rb_left;
4637     3526559    :                 else if (block >= (entry->start_blk + entry->count))
4638     3526559    :                         n = &(*n)->rb_right;
4639                :                 else {
4640           0    :                         ext4_grp_locked_error(sb, e4b->bd_group, __func__,
4641                :                                               "Double free of blocks %d (%d %d)",
4642                :                                               block, entry->start_blk, entry->count);
4643           0    :                         return 0;
4644                :                 }
4645                :         }
4646                :
4647                :         rb_link_node(new_node, parent, n);
4648     3896279    :         rb_insert_color(new_node, &db->bb_free_root);
4649                :
4650                :         /* Now try to see the extent can be merged to left and right */
4651     3896275    :         node = rb_prev(new_node);
4652     3896275    :         if (node) {
4653     2298360    :                 entry = rb_entry(node, struct ext4_free_data, node);
4654     2298360    :                 if (can_merge(entry, new_entry)) {
4655      469324    :                         new_entry->start_blk = entry->start_blk;
4656      469324    :                         new_entry->count += entry->count;
4657      469324    :                         rb_erase(node, &(db->bb_free_root));
4658      469324    :                         spin_lock(&sbi->s_md_lock);
4659      469324    :                         list_del(&entry->list);
4660      469324    :                         spin_unlock(&sbi->s_md_lock);
4661      469324    :                         kmem_cache_free(ext4_free_ext_cachep, entry);
4662                :                 }
4663                :         }
4664                :
4665     3896275    :         node = rb_next(new_node);
4666     3896276    :         if (node) {
4667     2405267    :                 entry = rb_entry(node, struct ext4_free_data, node);
4668     2405267    :                 if (can_merge(new_entry, entry)) {
4669      192718    :                         new_entry->count += entry->count;
4670      192718    :                         rb_erase(node, &(db->bb_free_root));
4671      192718    :                         spin_lock(&sbi->s_md_lock);
4672      192718    :                         list_del(&entry->list);
4673      192718    :                         spin_unlock(&sbi->s_md_lock);
```

```
4674          192718 :                        kmem_cache_free(ext4_free_ext_cachep, entry);
4675                 :                }
4676                 :        }
4677                 :        /* Add the extent to transaction's private list */
4678         3896276 :        spin_lock(&sbi->s_md_lock);
4679         3896279 :        list_add(&new_entry->list, &handle->h_transaction->t_private_list);
4680         3896279 :        spin_unlock(&sbi->s_md_lock);
4681         3896279 :        return 0;
4682                 : }
4683                 :
4684                 : /*
4685                 :  * Main entry point into mballoc to free blocks
4686                 :  */
4687                 : void ext4_mb_free_blocks(handle_t *handle, struct inode *inode,
4688                 :                          ext4_fsblk_t block, unsigned long count,
4689                 :                          int metadata, unsigned long *freed)
4690         3896279 : {
4691         3896279 :        struct buffer_head *bitmap_bh = NULL;
4692         3896279 :        struct super_block *sb = inode->i_sb;
4693         3896279 :        struct ext4_allocation_context *ac = NULL;
4694                 :        struct ext4_group_desc *gdp;
4695                 :        struct ext4_super_block *es;
4696                 :        unsigned int overflow;
4697                 :        ext4_grpblk_t bit;
4698                 :        struct buffer_head *gd_bh;
4699                 :        ext4_group_t block_group;
4700                 :        struct ext4_sb_info *sbi;
4701                 :        struct ext4_buddy e4b;
4702         3896279 :        int err = 0;
4703                 :        int ret;
4704                 :
4705         3896279 :        *freed = 0;
4706                 :
4707         3896279 :        sbi = EXT4_SB(sb);
4708         3896279 :        es = EXT4_SB(sb)->s_es;
4709         7792557 :        if (block < le32_to_cpu(es->s_first_data_block) ||
4710                 :            block + count < block ||
4711                 :            block + count > ext4_blocks_count(es)) {
4712               1 :                ext4_error(sb, __func__,
4713                 :                           "Freeing blocks not in datazone - "
4714                 :                           "block = %llu, count = %lu", block, count);
4715               0 :                goto error_return;
4716                 :        }
4717                 :
4718                 :        ext4_debug("freeing block %llu\n", block);
4719                 :        trace_ext4_free_blocks(inode, block, count, metadata);
4720                 :
4721         3896279 :        ac = kmem_cache_alloc(ext4_ac_cachep, GFP_NOFS);
4722         3896278 :        if (ac) {
4723         3896279 :                ac->ac_op = EXT4_MB_HISTORY_FREE;
4724         3896279 :                ac->ac_inode = inode;
4725         3896279 :                ac->ac_sb = sb;
4726                 :        }
4727                 :
4728         3896278 : do_more:
4729         3896278 :        overflow = 0;
4730         3896278 :        ext4_get_group_no_and_offset(sb, block, &block_group, &bit);
4731                 :
4732                 :        /*
4733                 :         * Check to see if we are freeing blocks across a group
4734                 :         * boundary.
4735                 :         */
4736         7792554 :        if (bit + count > EXT4_BLOCKS_PER_GROUP(sb)) {
4737               0 :                overflow = bit + count - EXT4_BLOCKS_PER_GROUP(sb);
4738               0 :                count -= overflow;
4739                 :        }
4740         3896277 :        bitmap_bh = ext4_read_block_bitmap(sb, block_group);
4741         3896276 :        if (!bitmap_bh) {
4742               0 :                err = -EIO;
4743               0 :                goto error_return;
4744                 :        }
```

```
4745  3896276 :         gdp = ext4_get_group_desc(sb, block_group, &gd_bh);
4746  3896278 :         if (!gdp) {
4747        0 :                 err = -EIO;
4748        0 :                 goto error_return;
4749        : :         }
4750        : :
4751 11688835 :         if (in_range(ext4_block_bitmap(sb, gdp), block, count) ||
4752        : :             in_range(ext4_inode_bitmap(sb, gdp), block, count) ||
4753        : :             in_range(block, ext4_inode_table(sb, gdp),
4754        : :                      EXT4_SB(sb)->s_itb_per_group) ||
4755        : :             in_range(block + count - 1, ext4_inode_table(sb, gdp),
4756        : :                      EXT4_SB(sb)->s_itb_per_group)) {
4757        : :
4758        0 :                 ext4_error(sb, __func__,
4759        : :                            "Freeing blocks in system zone - "
4760        : :                            "Block = %llu, count = %lu", block, count);
4761        : :                 /* err = 0. ext4_std_error should be a no op */
4762        0 :                 goto error_return;
4763        : :         }
4764        : :
4765        : :         BUFFER_TRACE(bitmap_bh, "getting write access");
4766  3896279 :         err = ext4_journal_get_write_access(handle, bitmap_bh);
4767  3896276 :         if (err)
4768        0 :                 goto error_return;
4769        : :
4770        : :         /*
4771        : :          * We are about to modify some metadata.  Call the journal APIs
4772        : :          * to unshare ->b_data if a currently-committing transaction is
4773        : :          * using it
4774        : :          */
4775        : :         BUFFER_TRACE(gd_bh, "get_write_access");
4776  3896276 :         err = ext4_journal_get_write_access(handle, gd_bh);
4777  3896275 :         if (err)
4778        0 :                 goto error_return;
4779        : : #ifdef AGGRESSIVE_CHECK
4780        : :         {
4781        : :                 int i;
4782        : :                 for (i = 0; i < count; i++)
4783        : :                         BUG_ON(!mb_test_bit(bit + i, bitmap_bh->b_data));
4784        : :         }
4785        : : #endif
4786  3896275 :         if (ac) {
4787  3896276 :                 ac->ac_b_ex.fe_group = block_group;
4788  3896276 :                 ac->ac_b_ex.fe_start = bit;
4789  3896276 :                 ac->ac_b_ex.fe_len = count;
4790  3896276 :                 ext4_mb_store_history(ac);
4791        : :         }
4792        : :
4793  3896273 :         err = ext4_mb_load_buddy(sb, block_group, &e4b);
4794  3896279 :         if (err)
4795        0 :                 goto error_return;
4796 11688836 :         if (metadata && ext4_handle_valid(handle)) {
4797        : :                 struct ext4_free_data *new_entry;
4798        : :                 /*
4799        : :                  * blocks being freed are metadata. these blocks shouldn't
4800        : :                  * be used until this transaction is committed
4801        : :                  */
4802  3896278 :                 new_entry  = kmem_cache_alloc(ext4_free_ext_cachep, GFP_NOFS);
4803  3896271 :                 new_entry->start_blk = bit;
4804  3896271 :                 new_entry->group  = block_group;
4805  3896271 :                 new_entry->count = count;
4806  3896271 :                 new_entry->t_tid = handle->h_transaction->t_tid;
4807        : :
4808  3896271 :                 ext4_lock_group(sb, block_group);
4809  3896262 :                 mb_clear_bits(bitmap_bh->b_data, bit, count);
4810  3896279 :                 ext4_mb_free_metadata(handle, &e4b, new_entry);
4811        : :         } else {
4812        : :                 /* need to update group_info->bb_free and bitmap
4813        : :                  * with group lock held. generate_buddy look at
4814        : :                  * them with group_lock_held
4815        : :                  */
```

```
4816          1 :                     ext4_lock_group(sb, block_group);
4817          0 :                     mb_clear_bits(bitmap_bh->b_data, bit, count);
4818          0 :                     mb_free_blocks(inode, &e4b, bit, count);
4819          0 :                     ext4_mb_return_to_preallocation(inode, &e4b, block, count);
4820            :             }
4821            :
4822    3896279 :             ret = ext4_free_blks_count(sb, gdp) + count;
4823    3896276 :             ext4_free_blks_set(sb, gdp, ret);
4824    3896278 :             gdp->bg_checksum = ext4_group_desc_csum(sbi, block_group, gdp);
4825    3896278 :             ext4_unlock_group(sb, block_group);
4826    3896279 :             percpu_counter_add(&sbi->s_freeblocks_counter, count);
4827            :
4828    3896279 :             if (sbi->s_log_groups_per_flex) {
4829    7792556 :                     ext4_group_t flex_group = ext4_flex_group(sbi, block_group);
4830    3896278 :                     atomic_add(count, &sbi->s_flex_groups[flex_group].free_blocks);
4831            :             }
4832            :
4833    3896280 :             ext4_mb_release_desc(&e4b);
4834            :
4835    3896279 :             *freed += count;
4836            :
4837            :             /* We dirtied the bitmap block */
4838            :             BUFFER_TRACE(bitmap_bh, "dirtied bitmap block");
4839    3896279 :             err = ext4_handle_dirty_metadata(handle, NULL, bitmap_bh);
4840            :
4841            :             /* And the group descriptor block */
4842            :             BUFFER_TRACE(gd_bh, "dirtied group descriptor block");
4843    3896277 :             ret = ext4_handle_dirty_metadata(handle, NULL, gd_bh);
4844    3896276 :             if (!err)
4845    3896279 :                     err = ret;
4846            :
4847    3896276 :             if (overflow && !err) {
4848          0 :                     block += count;
4849          0 :                     count = overflow;
4850            :                     put_bh(bitmap_bh);
4851            :                     goto do_more;
4852            :             }
4853    3896276 :             sb->s_dirt = 1;
4854    3896276 : error_return:
4855            :             brelse(bitmap_bh);
4856    3896277 :             ext4_std_error(sb, err);
4857    3896277 :             if (ac)
4858    3896277 :                     kmem_cache_free(ext4_ac_cachep, ac);
4859            :             return;
4860            : }
```