btrfs: fix deadlock with concurrent chunk allocations involving syste…

Browse files

```
…m chunks

When a task attempting to allocate a new chunk verifies that there is not
currently enough free space in the system space_info and there is another
task that allocated a new system chunk but it did not finish yet the
creation of the respective block group, it waits for that other task to
finish creating the block group. This is to avoid exhaustion of the system
chunk array in the superblock, which is limited, when we have a thundering
herd of tasks allocating new chunks. This problem was described and fixed
by commit eafa4fd ("btrfs: fix exhaustion of the system chunk array
due to concurrent allocations").

However there are two very similar scenarios where this can lead to a
deadlock:

1) Task B allocated a new system chunk and task A is waiting on task B
   to finish creation of the respective system block group. However before
   task B ends its transaction handle and finishes the creation of the
   system block group, it attempts to allocate another chunk (like a data
   chunk for an fallocate operation for a very large range). Task B will
   be unable to progress and allocate the new chunk, because task A set
   space_info->chunk_alloc to 1 and therefore it loops at
   btrfs_chunk_alloc() waiting for task A to finish its chunk allocation
   and set space_info->chunk_alloc to 0, but task A is waiting on task B
   to finish creation of the new system block group, therefore resulting
   in a deadlock;

2) Task B allocated a new system chunk and task A is waiting on task B to
   finish creation of the respective system block group. By the time that
   task B enter the final phase of block group allocation, which happens
   at btrfs_create_pending_block_groups(), when it modifies the extent
   tree, the device tree or the chunk tree to insert the items for some
   new block group, it needs to allocate a new chunk, so it ends up at
   btrfs_chunk_alloc() and keeps looping there because task A has set
   space_info->chunk_alloc to 1, but task A is waiting for task B to
   finish creation of the new system block group and release the reserved
   system space, therefore resulting in a deadlock.

In short, the problem is if a task B needs to allocate a new chunk after
it previously allocated a new system chunk and if another task A is
currently waiting for task B to complete the allocation of the new system
chunk.

Unfortunately this deadlock scenario introduced by the previous fix for
the system chunk array exhaustion problem does not have a simple and short
fix, and requires a big change to rework the chunk allocation code so that
chunk btree updates are all made in the first phase of chunk allocation.
And since this deadlock regression is being frequently hit on zoned
filesystems and the system chunk array exhaustion problem is triggered
in more extreme cases (originally observed on PowerPC with a node size
of 64K when running the fallocate tests from stress-ng), revert the
changes from that commit. The next patch in the series, with a subject
of "btrfs: rework chunk allocation to avoid exhaustion of the system
chunk array" does the necessary changes to fix the system chunk array
exhaustion problem.

Reported-by: Naohiro Aota <naohiro.aota@wdc.com>
Link: https://lore.kernel.org/linux-btrfs/20210621015922.ewgbffxuawia7liz@naota-xeon/
Fixes: eafa4fd ("btrfs: fix exhaustion of the system chunk array due to concurrent allocations")
CC: stable@vger.kernel.org # 5.12+
Tested-by: Shin'ichiro Kawasaki <shinichiro.kawasaki@wdc.com>
Tested-by: Naohiro Aota <naohiro.aota@wdc.com>
Signed-off-by: Filipe Manana <fdmanana@suse.com>
Tested-by: David Sterba <dsterba@suse.com>
Signed-off-by: David Sterba <dsterba@suse.com>
```

master

v6.1 … v5.14-rc2

fdmanana authored and **kdave** committed on Jul 7, 20211 parent 5f93e77 commit 1cb3db1cf383a3c7dbda1aa0ce748b0958759947

Showing **3 changed files** with **1 addition** and **69 deletions**.

Split Unified

58 🟥🟥🟥 fs/btrfs/block-group.c

```
   */
  void check_system_chunk(struct btrfs_trans_handle *trans, u64 type)
  {
-         struct btrfs_transaction *cur_trans = trans->transaction;
          struct btrfs_fs_info *fs_info = trans->fs_info;
          struct btrfs_space_info *info;
          u64 left;
          lockdep_assert_held(&fs_info->chunk_mutex);

          info = btrfs_find_space_info(fs_info, BTRFS_BLOCK_GROUP_SYSTEM);
- again:
          spin_lock(&info->lock);
          left = info->total_bytes - btrfs_space_info_used(info, true);
          spin_unlock(&info->lock);

          if (left < thresh) {
                  u64 flags = btrfs_system_alloc_profile(fs_info);
-                 u64 reserved = atomic64_read(&cur_trans->chunk_bytes_reserved);
-
-                 /*
-                  * If there's not available space for the chunk tree (system
-                  * space) and there are other tasks that reserved space for
-                  * creating a new system block group, wait for them to complete
-                  * the creation of their system block group and release excess
-                  * reserved space. We do this because:
-                  *
-                  * *) We can end up allocating more system chunks than necessary
-                  *    when there are multiple tasks that are concurrently
-                  *    allocating block groups, which can lead to exhaustion of
-                  *    the system array in the superblock;
-                  *
-                  * *) If we allocate extra and unnecessary system block groups,
-                  *    despite being empty for a long time, and possibly forever,
-                  *    they end not being added to the list of unused block groups
-                  *    because that typically happens only when deallocating the
-                  *    last extent from a block group - which never happens since
-                  *    we never allocate from them in the first place. The few
-                  *    exceptions are when mounting a filesystem or running scrub,
-                  *    which add unused block groups to the list of unused block
-                  *    groups, to be deleted by the cleaner kthread.
-                  *    And even when they are added to the list of unused block
-                  *    groups, it can take a long time until they get deleted,
-                  *    since the cleaner kthread might be sleeping or busy with
-                  *    other work (deleting subvolumes, running delayed iputs,
-                  *    defrag scheduling, etc);
-                  *
-                  * This is rare in practice, but can happen when too many tasks
-                  * are allocating blocks groups in parallel (via fallocate())
-                  * and before the one that reserved space for a new system block
-                  * group finishes the block group creation and releases the space
-                  * reserved in excess (at btrfs_create_pending_block_groups()),
-                  * other tasks end up here and see free system space temporarily
-                  * not enough for updating the chunk tree.
-                  *
-                  * We unlock the chunk mutex before waiting for such tasks and
-                  * lock it again after the wait, otherwise we would deadlock.
-                  * It is safe to do so because allocating a system chunk is the
-                  * first thing done while allocating a new block group.
-                  */
-                 if (reserved > trans->chunk_bytes_reserved) {
-                         const u64 min_needed = reserved - thresh;
-
-                         mutex_unlock(&fs_info->chunk_mutex);
-                         wait_event(cur_trans->chunk_reserve_wait,
-                            atomic64_read(&cur_trans->chunk_bytes_reserved) <=
-                            min_needed);
-                         mutex_lock(&fs_info->chunk_mutex);
-                         goto again;
-                 }

                  /*
                   * Ignore failure to create system chunk. We might end up not
                  ret = btrfs_block_rsv_add(fs_info->chunk_root,
                                            &fs_info->chunk_block_rsv,
                                            thresh, BTRFS_RESERVE_NO_FLUSH);
-                 if (!ret) {
-                         atomic64_add(thresh, &cur_trans->chunk_bytes_reserved);
+                 if (!ret)
                          trans->chunk_bytes_reserved += thresh;
-                 }
          }
  }

```

5 fs/btrfs/transaction.c

```
  void btrfs_trans_release_chunk_metadata(struct btrfs_trans_handle *trans)
  {
          struct btrfs_fs_info *fs_info = trans->fs_info;
-         struct btrfs_transaction *cur_trans = trans->transaction;

          if (!trans->chunk_bytes_reserved)
                  return;

          btrfs_block_rsv_release(fs_info, &fs_info->chunk_block_rsv,
                                  trans->chunk_bytes_reserved, NULL);
-         atomic64_sub(trans->chunk_bytes_reserved, &cur_trans->chunk_bytes_reserved);
-         cond_wake_up(&cur_trans->chunk_reserve_wait);
          trans->chunk_bytes_reserved = 0;
  }

          spin_lock_init(&cur_trans->dropped_roots_lock);
          INIT_LIST_HEAD(&cur_trans->releasing_ebs);
          spin_lock_init(&cur_trans->releasing_ebs_lock);
-         atomic64_set(&cur_trans->chunk_bytes_reserved, 0);
-         init_waitqueue_head(&cur_trans->chunk_reserve_wait);
          list_add_tail(&cur_trans->list, &fs_info->trans_list);
          extent_io_tree_init(fs_info, &cur_trans->dirty_pages,
                          IO_TREE_TRANS_DIRTY_PAGES, fs_info->btree_inode);
```

7 fs/btrfs/transaction.h

```

          spinlock_t releasing_ebs_lock;
```

```
        struct list_head releasing_ebs;
-
-       /*
-        * The number of bytes currently reserved, by all transaction handles
-        * attached to this transaction, for metadata extents of the chunk tree.
-        */
-       atomic64_t chunk_bytes_reserved;
-       wait_queue_head_t chunk_reserve_wait;
 };

 #define __TRANS_FREEZABLE       (1U << 0)
```

0 comments on commit `1cb3db1`

Please sign in to comment.