Immutable biovecs and biovec iterators

Kent Overstreet <kmo@daterainc.com>

As of 3.13, biovecs should never be modified after a bio has been submitted. Instead, we have a new struct bvec_iter which represents a range of a biovec - the iterator will be modified as the bio is completed, not the biovec.

More specifically, old code that needed to partially complete a bio would update bi_sector and bi_size, and advance bi_idx to the next biovec. If it ended up partway through a biovec, it would increment bv_offset and decrement bv_len by the number of bytes completed in that biovec.

In the new scheme of things, everything that must be mutated in order to partially complete a bio is segregated into struct bvec_iter: bi_sector, bi_size and bi_idx have been moved there; and instead of modifying bv_offset and bv_len, struct bvec_iter has bi_bvec_done, which represents the number of bytes completed in the current bvec.

There are a bunch of new helper macros for hiding the gory details - in particular, presenting the illusion of partially completed biovecs so that normal code doesn't have to deal with bi_bvec_done.

- Driver code should no longer refer to biovecs directly; we now have bio_iovec() and bio_iter_iovec() macros that return literal struct biovecs, constructed from the raw biovecs but taking into account bi_bvec_done and bi_size.

  bio_for_each_segment() has been updated to take a bvec_iter argument instead of an integer (that corresponded to bi_idx); for a lot of code the conversion just required changing the types of the arguments to bio_for_each_segment().
- Advancing a bvec_iter is done with bio_advance_iter(); bio_advance() is a wrapper around bio_advance_iter() that operates on bio->bi_iter, and also advances the bio integrity's iter if present.

  There is a lower level advance function - bvec_iter_advance() - which takes a pointer to a biovec, not a bio; this is used by the bio integrity code.

As of 5.12 bvec segments with zero bv_len are not supported.

What's all this get us?

Having a real iterator, and making biovecs immutable, has a number of advantages:

- Before, iterating over bios was very awkward when you weren't processing exactly one bvec at a time - for example, bio_copy_data() in block/bio.c, which copies the contents of one bio into another. Because the biovecs wouldn't necessarily be the same size, the old code was tricky convoluted - it had to walk two different bios at the same time, keeping both bi_idx and and offset into the current biovec for each.

  The new code is much more straightforward - have a look. This sort of pattern comes up in a lot of places; a lot of drivers were essentially open coding bvec iterators before, and having common implementation considerably simplifies a lot of code.
- Before, any code that might need to use the biovec after the bio had been completed (perhaps to copy the data somewhere else, or perhaps to resubmit it somewhere else if there was an error) had to save the entire bvec array - again, this was being done in a fair number of places.
- Biovecs can be shared between multiple bios - a bvec iter can represent an arbitrary range of an existing biovec, both starting and ending midway through biovecs. This is what enables efficient splitting of arbitrary bios. Note that this means we _only_ use bi_size to determine when we've reached the end of a bio, not bi_vcnt - and the bio_iovec() macro takes bi_size into account when constructing biovecs.
- Splitting bios is now much simpler. The old bio_split() didn't even work on bios with more than a single bvec! Now, we can efficiently split arbitrary size bios - because the new bio can share the old bio's biovec.

  Care must be taken to ensure the biovec isn't freed while the split bio is still using it, in case the original bio completes first, though. Using bio_chain() when splitting bios helps with this.
- Submitting partially completed bios is now perfectly fine - this comes up occasionally in stacking block drivers and various code (e.g. md and bcache) had some ugly workarounds for this.

  It used to be the case that submitting a partially completed bio would work fine to _most_ devices, but since accessing the raw bvec array was the norm, not all drivers would respect bi_idx and those would break. Now, since all drivers _must_ go through the bvec iterator - and have been audited to make sure they are - submitting partially completed bios is perfectly fine.

Other implications:

- Almost all usage of bi_idx is now incorrect and has been removed; instead, where previously you would have used bi_idx you'd now use a bvec_iter, probably passing it to one of the helper macros.

I.e. instead of using bio_iovec_idx() (or bio->bi_iovec[bio->bi_idx]), you now use bio_iter_iovec(), which takes a bvec_iter and returns a literal struct bio_vec - constructed on the fly from the raw biovec but taking into account bi_bvec_done (and bi_size).

- bi_vcnt can't be trusted or relied upon by driver code - i.e. anything that doesn't actually own the bio. The reason is twofold: firstly, it's not actually needed for iterating over the bio anymore - we only use bi_size. Secondly, when cloning a bio and reusing (a portion of) the original bio's biovec, in order to calculate bi_vcnt for the new bio we'd have to iterate over all the biovecs in the new bio - which is silly as it's not needed.

  So, don't use bi_vcnt anymore.

- The current interface allows the block layer to split bios as needed, so we could eliminate a lot of complexity particularly in stacked drivers. Code that creates bios can then create whatever size bios are convenient, and more importantly stacked drivers don't have to deal with both their own bio size limitations and the limitations of the underlying devices. Thus there's no need to define ->merge_bvec_fn() callbacks for individual block drivers.

Usage of helpers:

- The following helpers whose names have the suffix of *_all* can only be used on non-BIO_CLONED bio. They are usually used by filesystem code. Drivers shouldn't use them because the bio may have been split before it reached the driver.

```
bio_for_each_segment_all()
bio_for_each_bvec_all()
bio_first_bvec_all()
bio_first_page_all()
bio_last_bvec_all()
```

- The following helpers iterate over single-page segment. The passed 'struct bio_vec' will contain a single-page IO vector during the iteration:

  ```
  bio_for_each_segment()
  bio_for_each_segment_all()
  ```

- The following helpers iterate over multi-page bvec. The passed 'struct bio_vec' will contain a multi-page IO vector during the iteration:

  ```
  bio_for_each_bvec()
  bio_for_each_bvec_all()
  rq_for_each_bvec()
  ```