Linux BIO and Device Mapper

(NetDiox R&D Team)

Authored By :

Peter Chacko, Founder & CTO

Netdiox Computing System Pvt. Ltd.
(Linux kernel research/training center)

About the Author:

Peter Chacko has been working on system/networking/storage systems development since 1994. His career spans Hard-real time/embedded systems, OS kernels like Unix and Linux, distributed storage and networking systems, virtualization, data security and related computer science disciplines. He had been working for companies like Bellcore (Bell labs spin-off), HP, IBM, US West communications and a couple of startups as a consultant in the United states for 6+ years. He holds a Masters degree in Computer science &applications(MCA), Bachelors in Physics. He is currently the founder & CTO of Bangalore-based cloudStorage startup(Sciendix data systems pvt.Ltd). He also runs NetDiox computing systems as a not- for-profitresearch center-cum-educational center on system and networking software alongside. Prior to this he was working for calsoftlabs heading all the R&D activities of storage networking business unit.
His linked-in profile is at
https://in.linkedin.com/in/peterchacko

Introduction

All Block storage operations to actual backing block devices are performed through the Block I/O layer. The fundamental data structure representing a block operation is bio (since the major block layer overhaul of 2.4/2.6 version of Linux kernel). Bio is a list of scattered segments, which are denoted by an array of vectors, where each vector represent a unit of block IO, by a page, offset and length tuple. Multiple bios can then be linked together and make up a request, which represent a member of the request queue, managed by the IO scheduler and used by the device driver of the underlying block device.

For the sake of completeness, here is the actual definition of BIO structure, as extracted from the Linux kernel resources.

The bio structure uses a vector representation pointing to an array of tuples of <page, offset, len> to describe the i/o buffer, and has various other fields describing i/o parameters and state that needs to be maintained for performing the i/o.

Notice that this representation means that a bio has no virtual address mapping at all (unlike buffer heads).

```
struct bio_vec {
      struct page     *bv_page;
      unsigned short  bv_len;
      unsigned short  bv_offset;
};
/*
 * main unit of I/O for the block layer and lower layers (ie drivers)
 */
```

```
struct bio {
    struct bio          *bi_next;    /* request queue link */
    struct block_device *bi_bdev;    /* target device */
    unsigned long   bi_flags;    /* status, command, etc */
    unsigned long    bi_rw;       /* low bits: r/w, high: priority */
    unsigned int   bi_vcnt;     /* how many bio_vec's */
    struct bvec_iter bi_iter;       /* current index into bio_vec array */
    unsigned int   bi_size;     /* total size in bytes */
    unsigned short   bi_phys_segments; /* segments after physaddr coalesce*/
    unsigned short   bi_hw_segments; /* segments after DMA remapping */
    unsigned int    bi_max;      /* max bio_vecs we can hold  used as index into pool */
    struct bio_vec  *bi_io_vec;  /* the actual vec list */
    bio_end_io_t    *bi_end_io;  /* bi_end_io (bio) */
    atomic_t     bi_cnt;      /* pin count: free when it hits zero */
    void     *bi_private;
};
```

With this multipage bio design:

- ✓ Large i/os can be sent down in one go using a bio_vec list consisting of an array of <page, offset, len> fragments (similar to the way fragments are represented in the zero-copy network code)
- ✓ Splitting of an i/o request across multiple devices (as in the case of lvm or raid) is achieved by cloning the bio (where the clone points to the same bi_io_vec array, but with the index and size accordingly modified)
- ✓ Code that traverses the req list can find all the segments of a bio by using rq_for_each_segment.

  This handles the fact that a request has multiple bios, each of which can have multiple segments.

- ✓ Drivers which can't process a large bio in one shot can use the bi_iter field to keep track of the next bio_vec entry to process.

bi_end_io() i/o callback gets called on i/o completion of the entire bio.

At a lower level, drivers build a scatter gather list from the merged bios.

The scatter gather list is in the form of an array of <page, offset, len> entries with their corresponding dma address mappings filled in at the appropriate time. As an optimization, contiguous physical pages can be covered by a single entry where <page> refers to the first page and <len> covers the range of pages (up to 16 contiguous pages could be covered this way). There is a helper routine (blk_rq_map_sg) which drivers can use to build the sg list.

Generic bio helper Routines

Traversing segments and completion units in a request

The macro rq_for_each_segment() should be used for traversing the bios in the request list (drivers should avoid directly trying to do it themselves).

Using these helpers should also make it easier to cope with block changes in the future. f time (in the case of bio, that would be after the i/o is completed).

This ensures that if part of the pool has been used up, some work (in this case i/o) must already be in progress and memory would be available when it is over. If allocating from multiple pools in the same code path, the order or hierarchy of allocation needs to be consistent, just the way one deals with multiple locks.

The bio_alloc routine also needs to allocate the bio_vec_list (bvec_alloc()) for a non-clone bio. There are the 6 pools setup for different size biovecs, so bio_alloc(gfp_mask, nr_iovecs) will allocate a vec_list of the given size from these slabs.

The bio_get() routine may be used to hold an extra reference on a bio prior to i/o submission, if the bio fields are likely to be accessed after the i/o is issued (since the bio may otherwise get freed in case i/o completion happens in the meantime).

The bio_clone() routine may be used to duplicate a bio, where the clone shares the bio_vec_list with the original bio (i.e. both point to the same bio_vec_list). This would typically be used for splitting i/o requests in lvm or md.

Following is the request structure:

```
struct request {
      struct list_head queuelist;  /* Not meant to be directly accessed by the driver.
                                      Used by q->elv_next_request_fn  */
      .
      .

      unsigned char cmd[16];           /* prebuilt command data block */
      unsigned long flags;              /* also includes earlier rq->cmd settings */
      .
      .

      sector_t sector;    /* this field is now of type sector_t instead of int  preparation for 64 bit sectors */
      .

      .
                        /* Number of scatter-gather DMA addr+len pairs after
                         * physical address coalescing is performed. */

      unsigned short nr_phys_segments;

      /* Number of scatter-gather addr+len pairs after
       * physical and DMA remapping hardware coalescing is performed.
       * This is the number of scatter-gather entries the driver
       * will actually have to deal with after DMA mapping is done. */

      unsigned short nr_hw_segments;  /* Various sector counts */
      unsigned long nr_sectors;  /* no. of sectors left: driver modifiable */
      unsigned long hard_nr_sectors;  /* block internal copy of above */
      unsigned int current_nr_sectors; /* no. of sectors left in the current segm. */
      int tag;        /* command tag associated with request */
      void *special;  /* same as before */
      char *buffer;   /* valid only for low memory buffers up to current_nr_sectors */
```

```
        .
        .
        struct bio *bio, *biotail;  /* bio list instead of bh */
        struct request_list *rl;
}
```

See the rq_flag_bits definitions for an explanation of the various flags available. Some bits are used by the block layer or i/o scheduler.

The behaviour of the various sector counts are almost the same as before, except that since we have multi-segment bios, current_nr_sectors refers to the numbers of sectors in the current segment being processed which could be one of the many segments in the current bio (i.e i/o completion unit).

The nr_sectors value refers to the total number of sectors in the whole request that remain to be transferred (no change). The purpose of the hard_xxx values is for block to remember these counts every time it hands over the request to the driver. These values are updated by block on end_that_request_first, i.e. every time the driver completes a part of the transfer and invokes block end*request helpers to mark this. The driver should not modify these values. The block layer sets up the nr_sectors and current_nr_sectors fields (based on the corresponding hard_xxx values and the number of bytes transferred) and updates it on every transfer that invokes end_that_request_first. It does the same for the buffer, bio, bio->bi_iter fields too. ent:driver modifiable */

The I/O scheduler

I/O scheduler, is implemented in two layers. Generic dispatch queue and specific I/O schedulers. Unless stated otherwise, elevator is used to refer to both parts and I/O scheduler to specific I/O schedulers.

Block layer implements generic dispatch queue in block/*.c.

The generic dispatch queue is responsible for requeueing, handling non-fs requests and all other subtleties.

Specific I/O schedulers are responsible for ordering normal filesystem requests. They can also choose to delay certain requests to improve throughput or whatever purpose. As the plural form indicates, there are multiple I/O schedulers. They can be built as modules but at least one should be built inside the kernel. Each queue can choose different one and can also change to another one dynamically.

A block layer call to the i/o scheduler follows the convention elv_xxx(). This calls elevator_xxx_fn in the elevator switch (block/elevator.c).

The functions an elevator may implement are: (* are mandatory)

elevator_merge_fn	called to query requests for merge with a bio
elevator_merge_req_fn	called when two requests get merged. the one which gets merged into the other one will be never seen by I/O scheduler again. IOW, after being merged, the request is gone.
elevator_merged_fn	called when a request in the scheduler has been involved in a merge. It is used in the deadline scheduler for example, to reposition the request if its sorting order has changed.
elevator_allow_merge_fn	called whenever the block layer determines that a bio can be merged into an existing request safely. The io scheduler may still want to stop a merge at this point if it results in some sort of conflict internally, this hook allows it to do that. Note however that two *requests* can still be merged at later time. Currently the io scheduler has no way to prevent that. It can only learn about the fact from elevator_merge_req_fn callback.

elevator_dispatch_fn* fills the dispatch queue with ready requests.

I/O schedulers are free to postpone requests by -- **INSERT** --

elevator_former_req_fn

elevator_latter_req_fn These return the request before or after the one specified in disk sort order. Used by the block layer to find merge possibilities.

elevator_completed_req_fn called when a request is completed.

elevator_may_queue_fn returns true if the scheduler wants to allow the current context to queue a new request even if it is over the queue limit. This must be used very carefully!!

elevator_set_req_fn

elevator_put_req_fn Must be used to allocate and free any elevator specific storage for a request.

elevator_activate_req_fn Called when device driver first sees a request.

I/O schedulers can use this callback to determine when actual execution of a request starts.

elevator_deactivate_req_fn Called when device driver decides to delay a request by requeueing it.

elevator_init_fn*

elevator_exit_fn Allocate and free any elevator specific storage for a queu

Now lets look at Device Mapper which is the simple block layer infrastructure for building a virtual Block driver.

Device Mapper

Introduction

Device Mapper is a virtual block device driver framework provided by Linux kernel which provides an infrastructure to filter I/O for block devices. It provides a platform for filter drivers also known as target drivers to map a BIO to multiple block devices , or to modify the BIO.

Overview

Device mapper is a new kernel framework that registers the various driver entry functions to handle block I/O with the generic block layer. Device mapper framework itself is a device driver and hence all have the characteristics of driver entry and end of io functions.These functions transform the received BIOs and pass them to corresponding functions from the target device drivers for further processing. The Device Mapper block device driver exports a set of ioctl methods, as done by a typical device driver, which are used by a userspace tool called dmsetup. This tool creates a mapping between the sectors of the virtual block device and the sectors of the real block device.When this mapping is created, a data structure (mapping device) is generated, which stores all the information about the target and the underlying block drivers. The information regarding the underlying block driver is stored in a configuration data structure in the kernel memory.

When the generic block layer receives a BIO for an I/O, the BIO is plugged into the request queue of the Device Mapper block driver. The Device Mapper driver now processes the BIO as follows.

1. The BIO is first cloned and the end-of-I/O completion handler for the cloned BIO is set to that of Device Mapper's end-of-I/O handler.
2. The targets for the BIO are searched in the list of targets, and the cloned BIO is handed over to the appropriate target implementation.
3. The target driver processes the cloned BIO and modifies the data contained by the BIO according to target driver logic. It could be a simple offset re-mapping.
4. The target driver then re-directs the BIO towards the underlying block device that was mapped by the Device Mapper layer earlier, sets an appropriate end of I/O handler for the BIO and invokes the entry method generic_make_request() for the device driver.
5. After the completion of the I/O request by the device driver, the cloned BIO's end-of-I/O handler invokes Device Mapper block driver's end-of-I/O handler, which then notifies the upper layers about the completion of I/O.

About NetDiox Computing Systems:

NetDiox is a Bangalore(India) basedcenter of research and education focusing around OS technologies, networking, distributed storage systems, virtualization and cloud computing. It has been changing the waycomputer science is taught and applied to distributed data centers. NetDiox provides high quality training and R&D project opportunity on system software, Linux kernels, virtualization , and networking technologies to brilliant BE/M.Techs/MCA/MScs who are passionate about system/storage/networking software.

Contact Us:

NetDiox R&D Team

Netdiox Computing System Pvt. Ltd.

Address: #1514, First Floor,19th Main Road,
HSR Layout Sector-1, Bangalore 102,
Karnataka, India
Contact No: 08197324604
E-mail:**info.netdiox@gmail.com**