BLOCK IO AND BLOCK DEVICE DRIVERS

Important Notice : Courseware - Legal

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

Table of Contents

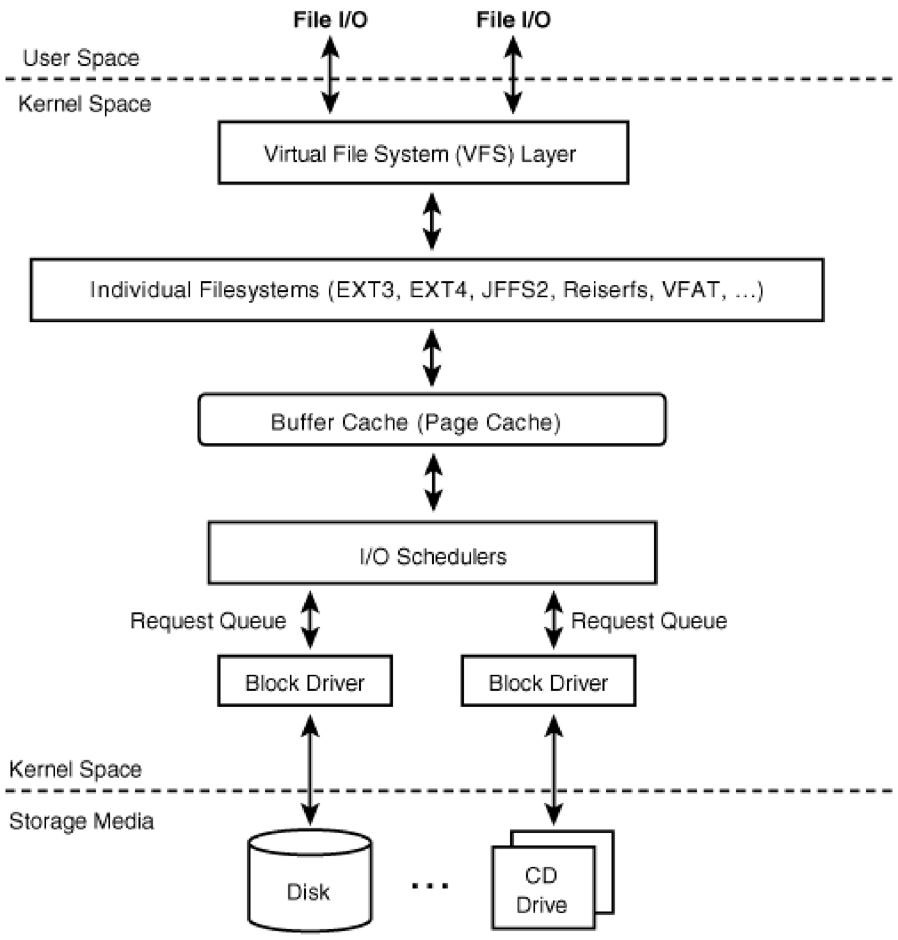
Block Device Drivers [1]

[1] *This material extracted from "Understanding the Linux Kernel" 3rd Ed., Bovet & Cesati, O'Reilly. All rights with above.*

Introduction

This chapter deals with I/O drivers for block devices, i.e., for disks of every kind. The key aspect of a block device is the disparity between the time taken by the CPU and buses to read or write data and the speed of the disk hardware. Block devices have very high average access times. Each operation requires several milliseconds to complete, mainly because << *it's an electro-mechanical device* >> the disk controller must move the heads on the disk surface to reach the exact position where the data is recorded. However, when the heads are correctly placed, data transfer can be sustained at rates of tens of megabytes per second.

The organization of Linux block device handlers is quite involved. We won't be able to discuss in detail all the functions that are included in the block I/O subsystem of the kernel; however, we'll outline the general software architecture.

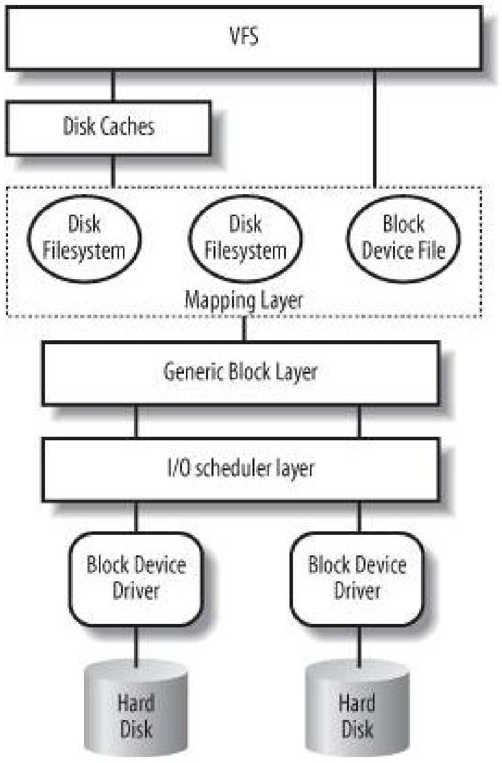


Block I/O on Linux.

<< Source: Essential Linux Device Drivers book >>

Block Devices Handling

Each operation on a block device driver involves a large number of kernel components; the most important ones are shown in Figure below.



Kernel components affected by a block device operation

Let us suppose, for instance, that a process issued a *read()* system call on some disk file - we'll see that *write* requests are handled essentially in the same way. Here is what the kernel typically does to service the process request:

1. The service routine of the *read(2)* system call activates a suitable VFS function, passing to it a file descriptor and an offset inside the file. The Virtual Filesystem is the upper layer of the block device handling architecture, and it provides a common file model adopted by all filesystems supported by Linux.

2. The VFS function determines if the requested data is already available (in the Disk Cache; <<*same as the Page Cache* >>) and, if necessary, how to perform the read operation. Sometimes there is no

need to access the data on disk, because the kernel keeps in RAM the data most recently read from - or written to - a block device.

3. Let's assume that the kernel must read the data from the block device << *a page cache 'miss'* >>, thus it must determine the physical location of that data. To do this, the kernel relies on the mapping layer, which typically executes two steps:

a. It determines the block size of the filesystem including the file and computes the extent of the requested data in terms of file block numbers . Essentially, the file is seen as split in many blocks, and the kernel determines the numbers (indices relative to the beginning of file) of the blocks containing the requested data.

b. Next, the mapping layer invokes a filesystem-specific function that accesses the file's disk inode and determines the position of the requested data on disk in terms of logical block numbers.

<<
The (virtual) function is called '*fiemap*' and is a member of the *inode_operations* data structure. For example, for the ext2 filesystem implementation, it is '*ext2_fiemap*' (as can be seen below):

```
const struct inode_operations ext2_file_inode_operations = {
        .truncate       = ext2_truncate,
#ifdef CONFIG_EXT2_FS_XATTR
        .setxattr       = generic_setxattr,
        .getxattr       = generic_getxattr,
        .listxattr      = ext2_listxattr,
        .removexattr    = generic_removexattr,
#endif
        .setattr        = ext2_setattr,
        .permission     = ext2_permission,
        .fiemap         = ext2_fiemap,
};
```
>>

Essentially, the disk is seen as split in blocks, and the kernel determines the numbers (indices relative to the beginning of the disk or partition) corresponding to the blocks storing the requested data. Because a file may be stored in nonadjacent blocks on disk, a data structure stored in the disk inode maps each file block number to a logical block number.[*]

[*] However, if the read access was done on a raw block device file, the mapping layer does not invoke a filesystem-specific method; rather, it translates the offset in the block device file to a position inside the disk or disk partition corresponding to the device file.

4. The kernel can now issue the read operation on the block device. It makes use of the ***generic block layer***, which **starts the I/O operations** that transfers the requested data. In general, each I/O operation involves a group of blocks that are adjacent on disk *(? - rather, logically adjacent)*. Because the requested data is not necessarily adjacent on disk, the generic block layer might start several I/O operations. ***Each I/O operation is represented by a "block I/O" (in short, "bio") structure, which collects all information needed by the lower components to satisfy the request.***

The generic block layer hides the peculiarities of each hardware block device, thus offering an abstract view of the block devices. Because almost all block devices are disks, the generic block layer also provides some general data structures that describe "disks" and "disk partitions." We will discuss the generic block layer and the bio structure in the section *"The Generic Block Layer"* later in this chapter.

5. Below the generic block layer, the **"I/O scheduler"** **sorts the pending I/O data transfer requests according to predefined kernel policies.** The purpose of the scheduler *is to group requests of data – by sorting and merging block I/O requests - that lie near each other* on the physical medium. We will describe this component in the section "The I/O Scheduler" later in this chapter.

6. Finally, the ***block device drivers*** **take care of the actual data transfer by sending suitable commands to the hardware interfaces of the disk controllers**. We will explain the overall organization of a generic block device driver in the section "Block Device Drivers" later in this chapter.

…

As you can see, there are many kernel components that are concerned with data stored in block devices; each of them **manages the disk data using chunks of different length**:

- The controllers of the hardware block devices transfer data in chunks of fixed length called "**sectors**." Therefore, the I/O scheduler and the block device drivers must manage sectors of data.
- The Virtual Filesystem, the mapping layer, and the filesystems group the disk data in logical units called "**blocks**" (or “block buffers”). A block corresponds to the minimal disk storage unit inside a filesystem.
  <<
  FAQ. How big is a “block” on the filesystem?
  A. For the Linux extended filesystems (ext2|3|4), we can see it like this:

  ```
  $ sudo tune2fs -l /dev/sda9 |grep 'Block size'
  Block size:               4096
  $
  ```
  >>
- As we will see shortly, block device drivers should be able to cope with "**segments**" of data: each segment is a **memory page** or a portion of a memory page - including chunks of data that are physically adjacent on disk.
- The disk (page) caches work on "**pages**" of disk data, each of which fits in a page frame.
- The generic block layer glues together all the upper and lower components, thus it knows about sectors, blocks, segments and pages of data.

Even if there are many different chunks of data, they usually share the same physical RAM cells. For instance, Figure 14-2 shows the layout of a 4,096-byte page. The upper kernel components see the page as composed of four block buffers (“blocks”) of 1,024 bytes each. The last three blocks of the page are being transferred by the block device driver, thus they are inserted in a segment covering the last 3,072 bytes of the page. The hard disk controller considers the segment as composed of six 512-byte sectors.

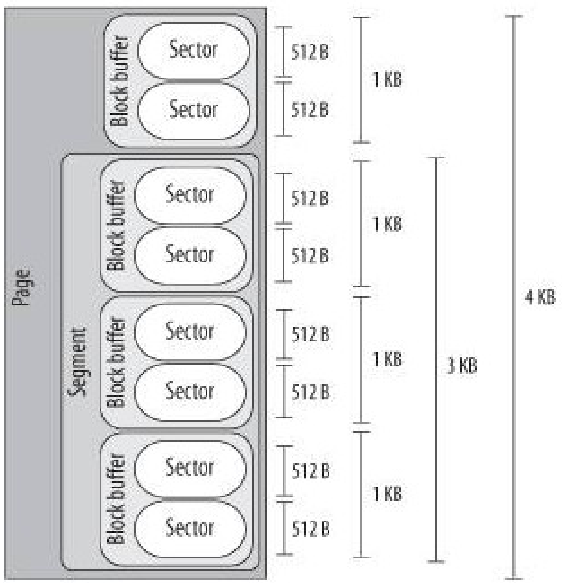
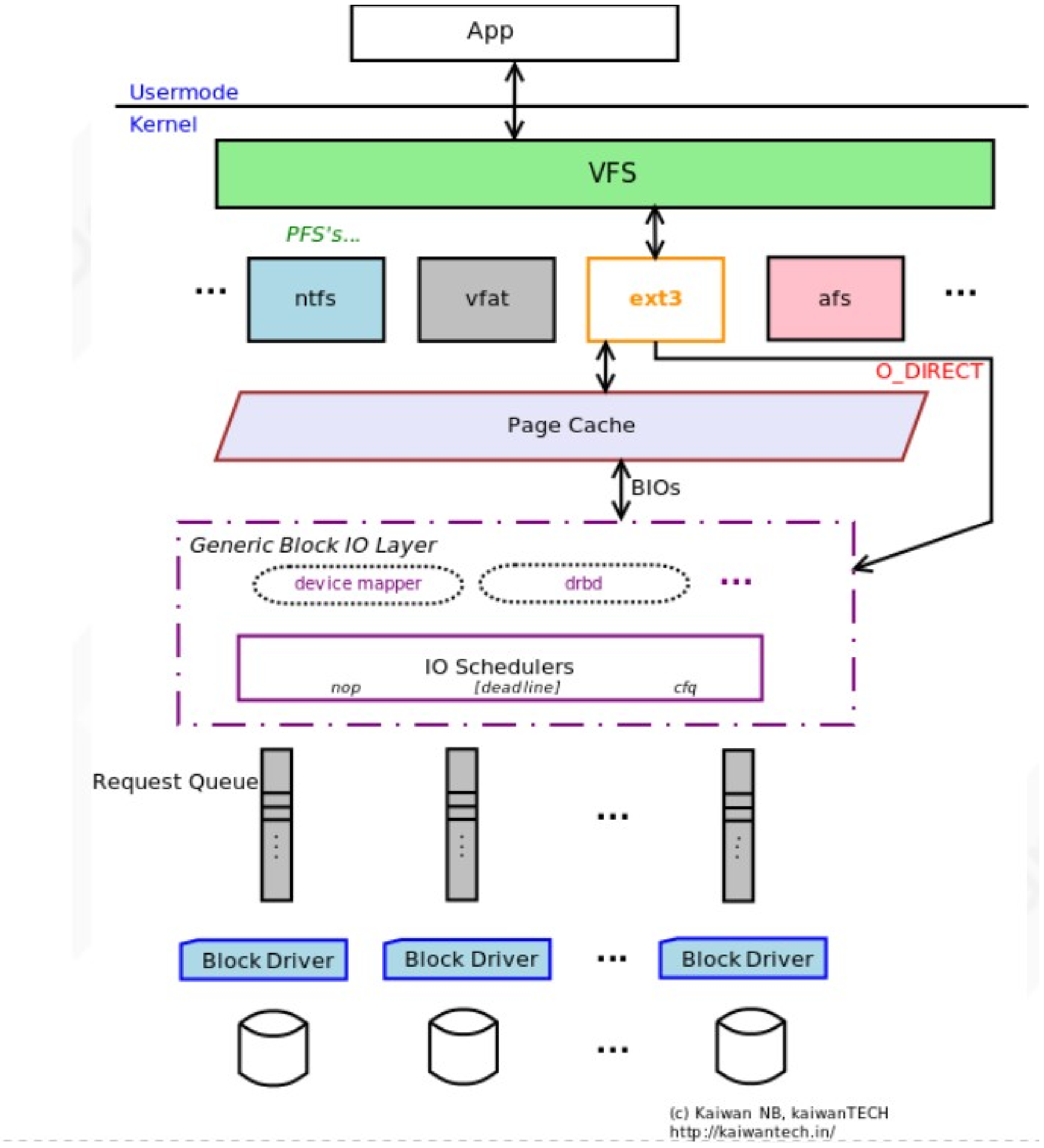


Figure 14-2. Typical layout of a page including disk data

[FYI, more details on “Sectors, Blocks & Segments” can be seen in Appendix A.]

The Generic Block IO Layer

The generic block layer is a kernel component that handles the requests for all block devices in the system.



Thanks to its functions, the kernel may easily:

- Put data buffers in high memory - the page frame(s) will be mapped in the kernel linear address space only when the CPU must access the data, and will be unmapped right after. << *This is called using "bounce" buffers.* >>
- Implement - with some additional effort - a *"zero-copy"* schema, where disk data is directly put in the User Mode address space without being copied to kernel memory first; essentially, the buffer used by the kernel for the I/O transfer lies in a page frame mapped in the User Mode linear address space of a process.
- Manage logical volumes - such as those used by LVM (the Logical Volume Manager) and RAID (Redundant Array of Inexpensive Disks): several disk partitions, even on different block devices, can be seen as a single partition. (Also see coverage on Device Mapper (dm) below).
- Exploit the advanced features of the most recent disk controllers, such as large onboard disk caches, enhanced DMA capabilities, onboard scheduling of the I/O transfer requests, and so on.

SIDEBAR :: More detailed I/O Path *[FYI / OPTIONAL]*

Linux makes available the option of using a "device mapper" (and multi-path IO – Linux's 2.6.13 implementation is called DM-Multipathing DM-MPIO).

The device mapper essentially provides a layering mechanism for block devices.
From the Wikipedia entry:

The **device mapper** is Linux kernel's framework for mapping physical block devices onto higher-level virtual block devices. It forms the foundation of LVM2, software RAIDs and dm-crypt disk encryption, and offers additional features such as file system snapshots.[1]

Device mapper works by passing data from a virtual block device, which is provided by the device mapper itself, to another block device. Data can be also modified in transition, which is performed, for example, in the case of device mapper providing disk encryption or simulation of unreliable hardware behavior.

Features

Functions provided by the device mapper include linear, striped and error mappings, as well as crypt and multipath targets. For example, two disks may be concatenated into one logical volume with a pair of linear mappings, one for each disk. As another example, crypt target encrypts the data passing through the specified device, by using the Linux kernel's Crypto API.[1]

[...]

Usage

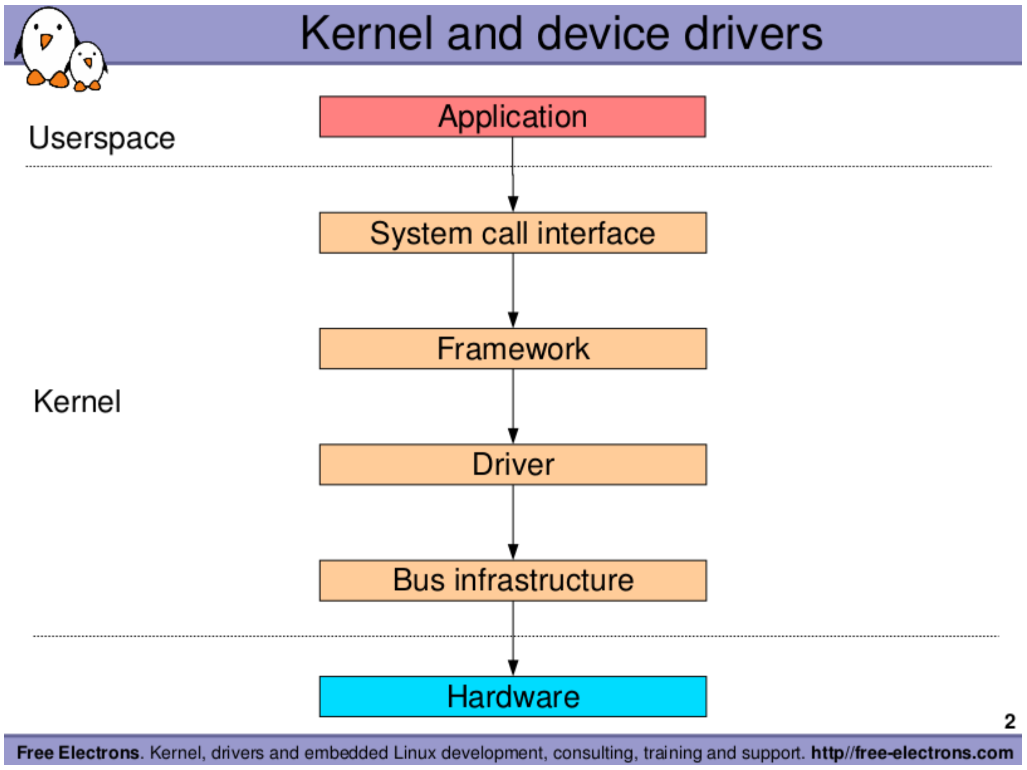
Applications (like LVM2 and EVMS) that need to create new mapped devices talk to the device mapper via the *libdevmapper.so* shared library, which in turn issues ioctls to the */dev/mapper/control* device node.[4] Configuration of the device mapper can be also examined and configured interactively —or from shell scripts—by using the dmsetup(8) utility.[5][6]

Both of these two userspace components have their source code maintained alongside the LVM2 source.[7]

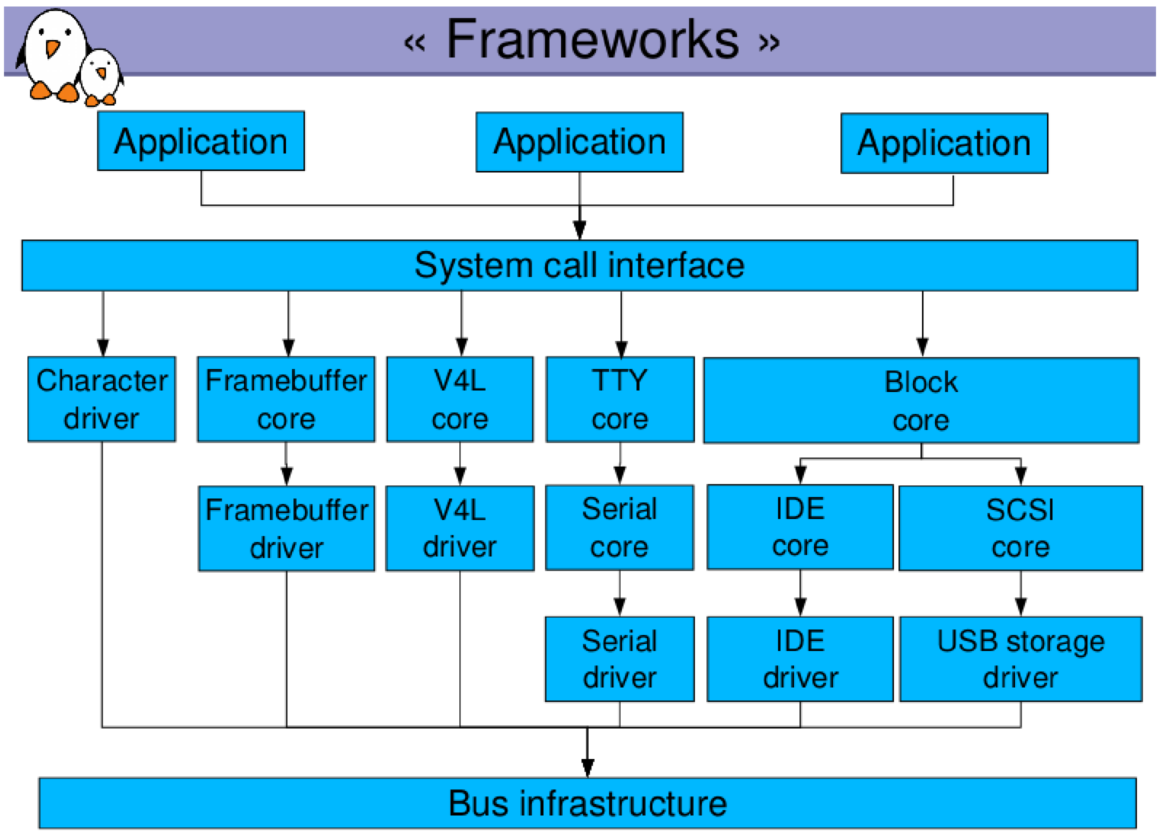
<<Useful: RHEL 7 LVM : "APPENDIX A. THE DEVICE MAPPER" >>

<< For mapping targets and projects that rely on the device mapper, see the wikipedia entry >>

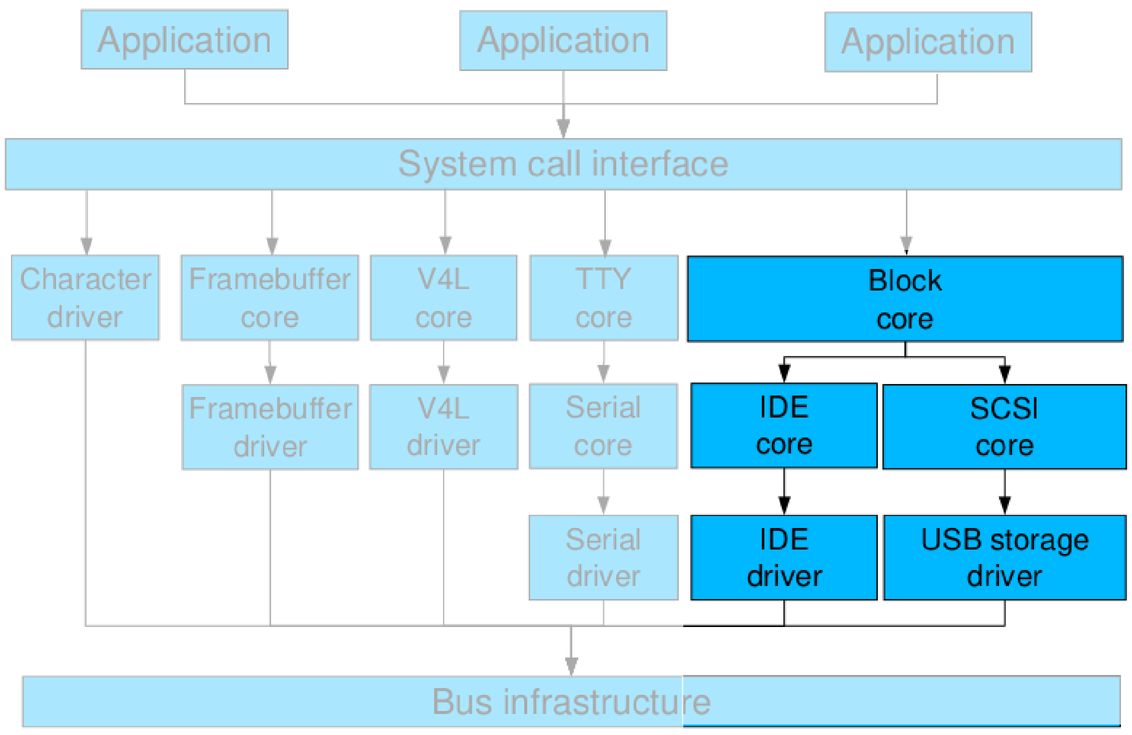
<<
SIDEBAR :: Linux Driver Model : Driver Framework
Ref/Src: http://free-electrons.com/


Kernel and device drivers
Userspace
Application
System call interface
Framework
Kernel
Driver
Bus infrastructure
Hardware
2
Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. http//free-electrons.com

- Many device drivers are not implemented directly as character drivers
- They are implemented under a « framework », specific to a given device type (framebuffer, V4L, serial, etc.)
  - The framework allows to factorize the common parts of drivers for the same type of devices
  - From userspace, they are still seen as character devices by the applications
  - The framework allows to provide a coherent userspace interface (ioctl, etc.) for every type of device, regardless of the driver
- The device drivers rely on the « bus infrastructure » to enumerate the devices and communicate with them.



The part we're dealing with here is the Block layer (and only part of it!):

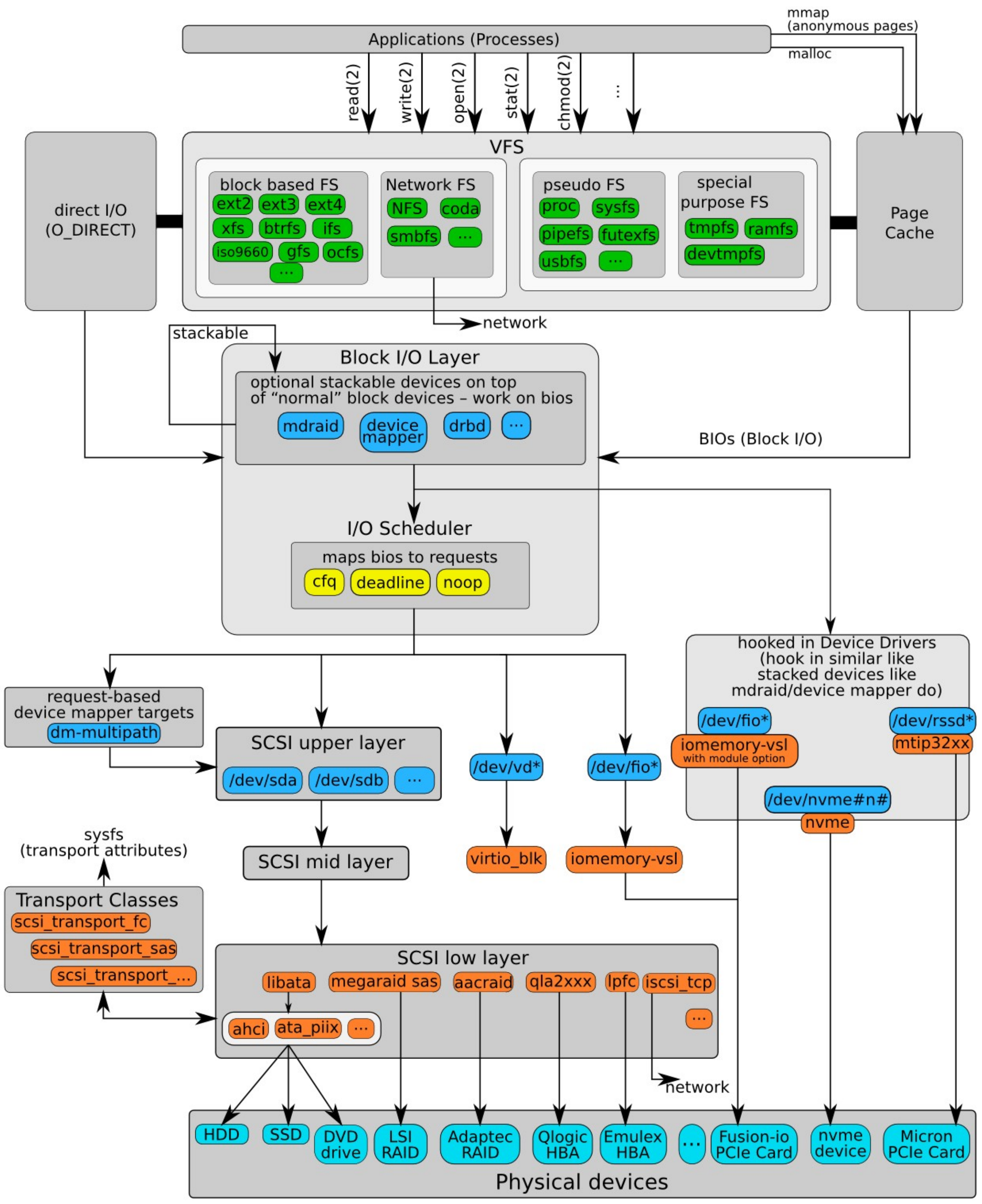


>>

"As the old computer science adage goes, all problems in computer science are created by, and solved by, adding an extra layer of abstraction." - a *DZone article*.

<<
The FTSE – Fundamental Theorem of Software Engineering ***:***

The **fundamental theorem of software engineering** (**FTSE**) is a term originated by Andrew Koenig to describe a remark by Butler Lampson attributed to the late David J. Wheeler: *"We can solve any problem by introducing an extra level of indirection."*
>>


Applications (Processes)
mmap (anonymous pages)
malloc
read(2)
write(2)
open(2)
stat(2)
chmod(2)
...
VFS
direct I/O (O_DIRECT)
block based FS
ext2 ext3 ext4 xfs btrfs ifs iso9660 gfs ocfs ...
Network FS
NFS coda smbfs ...
pseudo FS
proc sysfs pipefs futexfs usbfs ...
special purpose FS
tmpfs ramfs devtmpfs
Page Cache
network
stackable
Block I/O Layer
optional stackable devices on top of "normal" block devices – work on bios
mdraid device mapper drbd ...
BIOs (Block I/O)
I/O Scheduler
maps bios to requests
cfq deadline noop
hooked in Device Drivers (hook in similar like stacked devices like mdraid/device mapper do)
/dev/fio* iomemory-vsl with module option
/dev/rssd* mtip32xx
/dev/nvme#n# nvme
request-based device mapper targets
dm-multipath
SCSI upper layer
/dev/sda /dev/sdb ...
/dev/vd* virtio_blk
/dev/fio* iomemory-vsl
sysfs (transport attributes)
SCSI mid layer
Transport Classes
scsi_transport_fc scsi_transport_sas scsi_transport_...
SCSI low layer
libata megaraid sas aacraid qla2xxx lpfc iscsi_tcp ...
ahci ata_piix ...
network
HDD SSD DVD drive LSI RAID Adaptec RAID Qlogic HBA Emulex HBA ... Fusion-io PCIe Card nvme device Micron PCIe Card
Physical devices

The position of the device mapper targets within various layers of the Linux kernel's I/O stack. Source: Evgeny Budilovsky (April 2013). "Kernel-Based Mechanisms for High-Performance I/O" (PDF). Tel Aviv University. p. 8. Retrieved 2014-12-28.

Source

DM-Multipathing (**DM-MPIO**) provides input-output (I/O) fail-over and load-balancing within Linux for block devices.[1][2][3] By utilizing device-mapper, multipathd provides the host-side logic to use multiple paths of a redundant network to provide continuous availability and higher bandwidth connectivity between the host server and the block-level device.[4] DM-MPIO handles the rerouting of block I/O to an alternate path in the event of a path failure. DM-MPIO can also balance the I/O load across all of the available paths that are typically utilized in Fibre Channel (FC) and iSCSI SAN environments.[5] DM-MPIO is based on the device mapper, which provides the basic framework that maps one block device onto another.

[...]

<< Source: "Essential Linux Device Drivers" by S Venkateswaran, Prentice Hall >>

I/O Schedulers

Block devices suffer seek times, the latency to move the disk head from its existing position to the disk sector of interest. The main goal of an I/O scheduler is to increase system throughput by minimizing these seek times. To achieve this, I/O schedulers maintain the request queue in sorted order according to the disk sectors associated with the constituent requests. New requests are inserted into the queue such that this order is maintained. If an existing request in the queue is associated with an adjacent disk sector, the new request is merged with it. Because of these properties, I/O schedulers bear an operational resemblance to elevators - they schedule requests in a single direction until the last requester in the line is serviced.

The I/O scheduler in 2.4 kernels implemented a straightforward version of this algorithm and was called the *Linus elevator*. This turned out to be inadequate under real-world conditions, however, and was replaced in the 2.6 kernel by a suite of four schedulers: Deadline, Anticipatory, Complete Fair Queuing (CFQ), and Noop. The scheduler used by default is ~~Anticipatory~~ ~~Deadline~~ CFQ, but this can be changed during kernel configuration or by changing the value of */sys/block/[disk]/queue/scheduler*. (Replace [disk] with hda if you are using an IDE disk, for example.) Table 14.2 briefly describes Linux I/O schedulers.

```
# cat /sys/class/block/sda/queue/scheduler
noop deadline [cfq]
#
```

Table 14.2. Linux I/O Schedulers

I/O Scheduler	Description
Linus elevator	Straightforward implementation of the standard merge-and-sort I/O scheduling (in the 2.4 kernel tree) algorithm.

Deadline	In addition to what the *Linus elevator* does, the *Deadline scheduler* associates expiration times with each request in order to ensure that a burst of requests to the same disk region do not starve requests to regions that are farther away.
	Moreover, read operations are granted more priority than write operations because user processes usually block until their read requests complete. The *Deadline scheduler* thus ensures that each I/O request is serviced within a time limit, which is important for some database loads.
	By default, the expire time of read requests is 500 milliseconds, while the expire time for write requests is 5 seconds
	<< tunable via */sys/block/[disk]/queue/iosched/*
	[Older]:
	/sys/block/[disk]/queue/iosched/read_expire and
	/sys/block/[disk]/queue/iosched/write_expire
	`# cat /sys/class/block/sda/queue/iosched/read_expire`
	`500`

	`# cat /sys/class/block/sda/queue/iosched/write_expire`
	`5000`
	`#`
	>>

Anticipatory	Similar to the *Deadline* scheduler, except that after servicing read requests, the *Anticipatory scheduler* waits for a predetermined amount of time anticipating further requests (125ms for reads, 250ms for writes). This scheduling technique is suited for workstation/interactive loads.
Complete Fair Queuing (CFQ)	Similar to the *Linus elevator*, except that the *CFQ scheduler* maintains one request queue per originating process, rather than one generic queue. This ensures that each process (or process group) gets a fair portion of the I/O and prevents one process from starving others. << Recent: CFQ is the default kernel I/O scheduler; kernel config: CONFIG_DEFAULT_IOSCHED="cfq" >>
Noop	The *Noop scheduler* doesn't spend time traversing the request queue searching for optimal insertion points. Instead, it simply adds new requests to the tail of the request queue. This scheduler is thus ideal for semiconductor storage media that have no moving parts and, hence, no seek latencies. An example is a Disk-On-Module (DOM), which internally uses flash memory.
[Blk-mq] multiqueue support; prereq for the new ***BFQ – Budget Fair Queuing – IO scheduler***	(All above : now called legacy block layer; modern: blk-mq). Recent kernels : 4.16 onwards; BFQ: ELC-2018 Video, Paolo Valente, Linaro *Very* promising; no real difference in IO latencies experienced with or without IO load ! Imp for Android as well (blk-mq support in MMC). *Res:* *Linux Kernel Documentation* The multiqueue block layer, LWN, Jun 2013 ; Linux Multi-Queue Block IO Queueing Mechanism (blk-mq), Thomas-Krenn, *Jun 2015;* ArchLinux Wiki ; *How to enable and use the BFQ scheduler? [SX]*

[FYI: S Benchmark is a storage IO benchmark suite; iostress – simple IO stress utility]

At a conceptual level, I/O scheduling resembles process scheduling. Whereas I/O scheduling provides an illusion to processes that they own the disk, process scheduling gives processes the illusion that they own the CPU. Both I/O and process schedulers on Linux have undergone extensive changes in recent times.

To select the IO scheduler at boot time, use the argument '*elevator=deadline*'.

'noop', 'as' and 'cfq' (the default) are also available. IO schedulers are assigned globally at boot time only presently. For details, see:
Documentation/block/switching-sched.txt

Block Driver Data Structures and Methods

Let's now shift focus to the main topic of this chapter, block device drivers. In this section, we take a look at the important data structures and driver methods that you are likely to encounter while implementing a block device driver. We use these structures and methods in the next section when we implement a block driver for a fictitious storage controller.

The following are the main block driver data structures:

1. The kernel represents a disk using the gendisk (short for generic disk) structure defined in *include/linux/genhd.h* :

```
struct gendisk {
  int major;            /* Device major number */
  int first_minor;      /* Starting minor number */
  int minors;           /* Maximum number of minors. You have one
                           minor number per disk partition */

  char disk_name[32];    /* Disk name */

  /* ... */

  struct block_device_operations *fops;      /* Block device
                   operations. Described soon. */

  struct request_queue *queue;            /* The request queue
                  associated with this disk. Discussed next. */
/* ... */

};
```

2. The I/O request queue associated with each block driver is described using the *request_queue* structure defined in *include/linux/blkdev.h*. This is a big structure, but its only constituent field that you might use is the *request* structure, which is described next.

3. Each request in a *request_queue* is represented using a *request* structure defined in *include/linux/blkdev.h*:

```
struct request {
  /* ... */
  struct request_queue *q;   /* The container request queue */
  /* ... */
  sector_t sector;            /* Sector from which data access
                                 is requested */
  /* ... */
  unsigned long nr_sectors;   /* Number of sectors left to
                                 submit */
  /* ... */
  struct bio *bio;            /* The associated bio. Discussed soon. */
  /* ... */
  char *buffer;               /* The buffer for data transfer */
  /* ... */
  struct request *next_rq;    /* Next request in the queue */
};
```

4. *block_device_operations* is the block driver counterpart of the *file_operations* structure used by character drivers. It contains the following entry points associated with a block driver:

- Standard methods such as *open(), release(), and ioctl()*
- Specialized methods such as *media_changed()* and *revalidate_disk()* that support removable block devices.

block_device_operations is defined as follows in *include/linux/fs.h*:

```
struct block_device_operations {
  int (*open) (struct inode *, struct file *);          /* Open */
  int (*release) (struct inode *, struct file *);       /* Close */
  int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
                                                        /* I/O Control */
  /* ... */
  int (*media_changed) (struct ge ndisk *);    /* Check if media is
                                          available or ejected */
  int (*revalidate_disk) (struct gendisk *);   /* Gear up for newly
                                                inserted media */
  /* ... */
};
```

5. When we looked at the request structure, we saw that it was associated with a bio. A *bio* structure is a low-level description of block I/O operations at page-level granularity.

<< Discussed in detail in the next section. Read that and return here... >>

Block data is internally represented as an I/O vector using an array of *bio_vec* structures. Each element of the *bio_vec* array is made up of a *(page, page_offset, length)* tuple that describes a segment of the I/O block. Maintaining I/O requests as a vector of pages brings several advantages, including a leaner implementation and efficient scatter/gather.

Before ending this section, let's briefly look at block driver entry points. Block drivers are broadly built using three types of methods:

- The usual initialization and exit methods.
- Methods that are part of the *block_device_operations* described previously.
- A *request* method. Block drivers, unlike char devices, do not support *read()/write()* methods for data transfer. Instead, they perform disk access using a special routine called the *request* method.

…

The Bio Structure

The **core** data structure of the generic block layer is a **descriptor of an ongoing I/O block device operation called a *bio***. Each bio essentially includes an *identifier for a disk storage area* - the initial sector number and the number of sectors included in the storage area - and one or more segments describing the memory areas involved in the I/O operation. A bio is implemented by the **bio data structure**, whose fields are listed in Table 14-1.

<< Recall that (stated earlier): "Each I/O operation is represented by a "block I/O" (in short, "bio") structure, which collects all information needed by the lower components to satisfy the request." >>

Table 14-1. The fields of the bio structure

Type	**Field**	**Description**
sector_t	bi_sector	First sector on disk of block I/O operation
struct bio *	bi_next	Link to the next bio in the request queue
struct gendisk *	bi_disk	The disk the request is associated with
~~struct block_device *~~	~~bi_bdev~~	~~Pointer to block device descriptor~~
unsigned long	bi_flags	Bio status flags
unsigned long	bi_rw	I/O operation flags
unsigned short	bi_vcnt	Number of segments in the bio's bio_vec array (how many bio_vec's)
unsigned short	bi_idx	Current index in the bio's bio_vec array of segments
unsigned short	bi_phys_segments	Number of physical segments of the bio after merging
~~unsigned short~~	~~bi_hw_segments~~	~~Number of hardware segments after merging~~
unsigned int	bi_size	Bytes (yet) to be transferred
~~unsigned int~~	~~bi_hw_front_size~~	~~Used by the hardware segment merge algorithm~~
~~unsigned int~~	~~bi_hw_back_size~~	~~Used by the hardware segment merge algorithm~~

unsigned int	bi_seg_front_size	To keep track of the max segment size, we account for the sizes of the first and last mergeable segments in this bio.
unsigned int	bi_seg_back_size	
unsigned int	bi_max_vecs	Maximum allowed number of segments in the bio's bio_vec array
struct bio_vec *	bi_io_vec	Pointer to the bio's bio_vec array of segments (the actual vec list)
bio_end_io_t *	bi_end_io	Method invoked at the end of bio's I/O operation
atomic_t	bi_cnt	Reference counter for the bio
void *	bi_private	Pointer used by the generic block layer and the I/O completion method of the block device driver
bio_destructor_t *	bi_destructor	Destructor method (usually bio_destructor()) invoked when the bio is being freed

<<
From include/linux/blk_types.h ~~bio.h~~ :

--snip--

```
/*
 * main unit of I/O for the block layer and lower layers (ie drivers and
 * stacking drivers)
 */
struct bio {
        struct bio              *bi_next;       /* request queue link */
        struct gendisk         *bi_disk;
        unsigned int            bi_opf;         /* bottom bits req flags,
                                                 * top bits REQ_OP. Use
                                                 * accessors.
                                                 */
        unsigned short          bi_flags;       /* status, etc and bvec pool number */
        unsigned short          bi_ioprio;
        unsigned short          bi_write_hint;
        blk_status_t            bi_status;
        u8                      bi_partno;

         /* Number of segments in this BIO after
         * physical address coalescing is performed.
         */
        unsigned int            bi_phys_segments;
	/*
	 * To keep track of the max segment size, we account for the
	 * sizes of the first and last mergeable segments in this bio.
	 */
	unsigned int		bi_seg_front_size;
	unsigned int		bi_seg_back_size;

	struct bvec_iter	bi_iter;

	atomic_t		__bi_remaining;
	bio_end_io_t		*bi_end_io;

	void			*bi_private;
#ifdef CONFIG_BLK_CGROUP
	/*
	 * Optional ioc and css associated with this bio.  Put on bio
	 * release.  Read comment on top of bio_associate_current().
	 */
	struct io_context	   *bi_ioc;
	struct cgroup_subsys_state *bi_css;
#ifdef CONFIG_BLK_DEV_THROTTLING_LOW
	void			*bi_cg_private;
	struct blk_issue_stat	bi_issue_stat;
#endif
#endif

	union {
#if defined(CONFIG_BLK_DEV_INTEGRITY)
		struct bio_integrity_payload *bi_integrity; /* data integrity */
#endif
	};

	unsigned short		bi_vcnt;	/* how many bio_vec's */
			<< number of segments in the bio's bio_vec array >>
	/*
	 * Everything starting with bi_max_vecs will be preserved by bio_reset()
	 */
	unsigned short		bi_max_vecs;	/* max bvl_vecs we can hold */
	atomic_t		__bi_cnt;	/* pin count */
	struct bio_vec		*bi_io_vec;	/* the actual vec list */
	struct bio_set		*bi_pool;

	/*
	 * We can inline a number of vecs at the end of the bio, to avoid
	 * double allocations for a small number of bio_vecs. This member
	 * MUST obviously be kept at the very end of the bio.
	 */
	struct bio_vec		bi_inline_vecs[0];
```

```
};
```

***Each segment* in a bio is represented by a `bio_vec` data structure**, whose fields are listed in Table 14-2. The `bi_io_vec` field of the bio points to the first element **of an array of `bio_vec` data structures**, while the `bi_vcnt` field stores the current number of elements in the array.

Table 14-2. The fields of the bio_vec structure

Type	**Field**	**Description**
struct page *	bv_page	Pointer to the page descriptor of the segment's page frame
unsigned int	bv_len	Length of the segment in bytes
unsigned int	bv_offset	Offset of the segment's data in the page frame

```
struct bio_vec {                    << represents a segment in a bio >>
        struct page   *bv_page;
        unsigned int bv_len;
        unsigned int bv_offset;
};
```

[P. T. O.]

Block I/O Layer - Block Device Driver - Important Data Structures

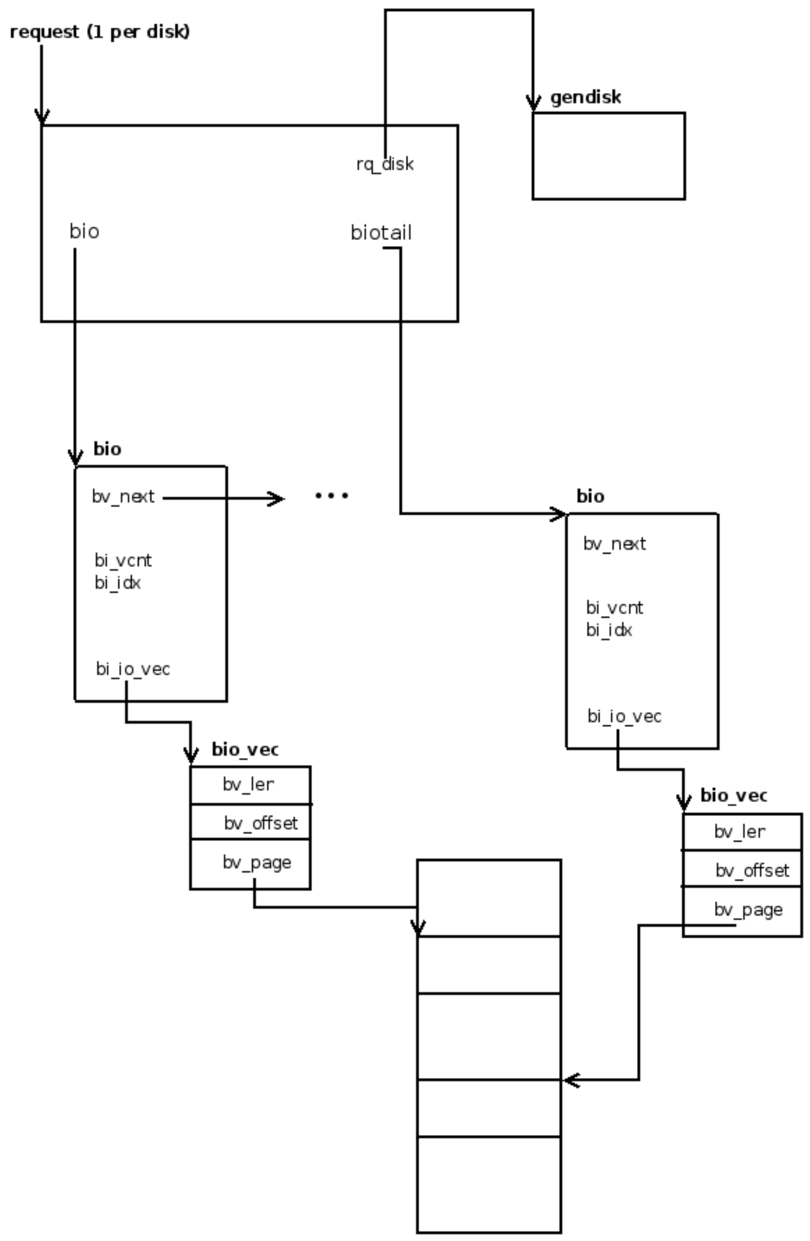


Figure: The request structure (struct request in include/linux/blkdev.h).

<< FYI, one can find the definition of
struct request : in *include/linux/blkdev.h*
struct gendisk : in *include/linux/genhd.h*
struct bio : in *include/linux/blk_types.h*
>>

The contents of a bio descriptor keep changing during the block I/O operation. For instance, if the block device driver cannot perform the whole data transfer with one scatter-gather DMA operation, the *bi_idx* field is updated to keep track of the first segment in the bio that is yet to be transferred. To iterate over the segments of a bio - starting from the current segment at index *bi_idx* - a device driver can execute the **bio_for_each_segment** macro.

<< Recent: now there's no *bo_idx ;* rather, the *bio_for_each_segment* is implemented using the *bi_iter* field >>.

When the generic block layer starts a new I/O operation, it allocates a new *bio* structure by invoking the *bio_alloc( )* function. Usually, bios are allocated through the slab allocator, but the kernel also keeps a small memory pool of bios to be used when memory is scarce (see the section "Memory Pools" in Chapter 8). The kernel also keeps a memory pool for the bio_vec structures - after all, it would not make sense to allocate a bio without being able to allocate the segment descriptors to be included in the bio.

Correspondingly, the *bio_put( )* function decrements the reference counter (bi_cnt) of a bio and, if the counter becomes zero, it releases the *bio* structure and the related *bio_vec* structures.

FYI, for further information on this topic, please refer:

- *Ch 16 "Block Drivers" from"Linux Device Drivers" 3rd Ed., Corbet, Rubini & Hartman, O'Reilly.*
- *Ch 14 "Block Drivers" from"Essential Linux Device Drivers" S Venkateswaran, Prentice Hall.*

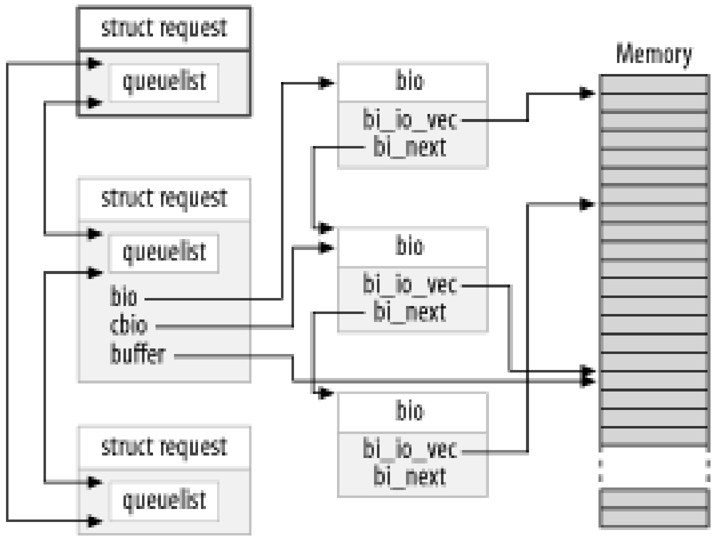
<<

NOTE:-
The block driver layer underwent several changes which showed up from the 2.6.31 kernel. Hence, the older sample block driver - sbull – from the LDD3 book no longer works as advertised.

A simpler updated, ported, working! version, can be found here:
http://blog.superpat.com/2010/05/04/a-simple-block-driver-for-linux-kernel-2-6-31/

A slightly enhanced version of the 'sbd' block driver above is provided on the particpant courseware and demonstrated by the instructor.

>>



Block requests and DMA

If you are working on a high-performance block driver, chances are you will be using DMA for the actual data transfers. A block driver can certainly step through the bio structures, as described above, create a DMA mapping for each one, and pass the result to the device. There is an easier way, however, if your device can do scatter/gather I/O. The function:

```
 int blk_rq_map_sg(request_queue_t *queue,
struct request *req, struct scatterlist *list);
```

fills in the given list with the full set of segments from the given request. Segments that are adjacent in memory are coalesced prior to insertion into the scatterlist, so you need not try to detect them yourself. The return value is the number of entries in the list. The function also passes back, in its third argument, a scatterlist suitable for passing to *dma_map_sg*. (See the section “Scatter-gather mappings” in Chapter 15 for more information on dma_map_sg.)

Your driver must allocate the storage for the scatterlist before calling *blk_rq_map_sg*. The list must be able to hold at least as many entries as the request has physical segments; the *struct request* field *nr_phys_segments* holds that count, which will not exceed the maximum number of physical segments specified with *blk_queue_max_phys_segments*.

If you do not want *blk_rq_map_sg* to coalesce adjacent segments, you can change the default behavior with a call such as:

```
clear_bit(QUEUE_FLAG_CLUSTER, &queue->queue_flags);
```

Some SCSI disk drivers mark their request queue in this way, since they do not benefit from the coalescing of requests.

<<
SG-DMA:

A real driver that does scatter-gather (SG-)DMA : the Disk Array driver for Compaq SMART2 Controllers.
Code: *drivers/block/cpqarray.c:do_ida_request()* function.

For context, see "Essential Linux Device Drivers", Venkateswaran,
Ch 14 "Block Drivers" sec "Advanced Topics" pg 484.

Code View

```
File : [kernel 3.10.24 ] : drivers/block/cpqarray.c
...
<< Note: in interrupt context here ... >>
/*
 * Get a request and submit it to the controller.
 * This routine needs to grab all the requests it possibly can from the
 * req Q and submit them.  Interrupts are off (and need to be off) when you
 * are in here (either via the dummy do_ida_request functions or by being
 * called from the interrupt handler
 */
static void do_ida_request(struct request_queue *q)
{
    ctlr_info_t *h = q->queuedata;
    cmdlist_t *c;
    struct request *creq;
    struct scatterlist tmp_sg[SG_MAX];  << SG_MAX = 32 >>
    int i, dir, seg;

queue_next:
    creq = blk_peek_request(q);
    if (!creq)                        << no more request queued? Start IO... >>
        goto startio;

    BUG_ON(creq->nr_phys_segments > SG_MAX);

    if ((c = cmd_alloc(h,1)) == NULL)
        goto startio;

    blk_start_request(creq);   << Start request processing on the driver
            Dequeue @req and start timeout timer on it. This hands off the
            request to the driver. >>

    c->ctlr = h->ctlr;
    c->hdr.unit = (drv_info_t *)(creq->rq_disk->private_data) - h->drv;
    c->hdr.size = sizeof(rblk_t) >> 2;
    c->size += sizeof(rblk_t);

    c->req.hdr.blk = blk_rq_pos(creq);  << the current sector >>
    c->rq = creq;
DBGPX(
    printk("sector=%d, nr_sectors=%u\n",
           blk_rq_pos(creq), blk_rq_sectors(creq));
);
    sg_init_table(tmp_sg, SG_MAX);
    seg = blk_rq_map_sg(q, creq, tmp_sg); << map a request to
        scatterlist, return number of sg entries setup. Caller must make
        sure sg can hold rq->nr_phys_segments entries >>

    /* Now do all the DMA Mappings */
    if (rq_data_dir(creq) == READ)
        dir = PCI_DMA_FROMDEVICE;
    else
        dir = PCI_DMA_TODEVICE;
    for( i=0; i < seg; i++)  << loop around the scatterlist, setting up
               the DMA descriptors and mapping each page in the list >>
    {
        c->req.sg[i].size = tmp_sg[i].length;
        c->req.sg[i].addr = (__u32) pci_map_page(h->pci_dev,
                        << pci_map_page is a wrapper over dma_map_page >>
                         sg_page(&tmp_sg[i]),
                         tmp_sg[i].offset,
                         tmp_sg[i].length, dir);
    }
DBGPX(  printk("Submitting %u sectors in %d segments\n", blk_rq_sectors(creq),
seg); );
    c->req.hdr.sg_cnt = seg;
    c->req.hdr.blk_cnt = blk_rq_sectors(creq); << sectors left in the
                                                   entire request >>
    c->req.hdr.cmd = (rq_data_dir(creq) == READ) ? IDA_READ : IDA_WRITE;
    c->type = CMD_RWREQ;

    /* Put the request on the tail of the request queue */
    addQ(&h->reqQ, c);
    h->Qdepth++;
    if (h->Qdepth > h->maxQsinceinit)
        h->maxQsinceinit = h->Qdepth;

    goto queue_next;

startio:
    start_io(h);   << The comment in the driver's start_io() function says:
 * start_io submits everything on a controller's request queue
 * and moves it to the completion queue.
 *
 * Interrupts had better be off if you're in here >>
}
```

>>

An experiment with regard to Block IO with the kernel tracer Ftrace

Visualize a file being copied from a generic hard disk onto an ext4 formatted SD-MMC card.

What is the exact kernel code path? Is the generic VFS code going to run? The ext4 filesystem layer code? What about the generic Block layer code? The underlying block driver code?

All these questions are well answered by having the patience to setup (not that hard, actually!) and trace the kernel code paths in great detail with the superb Ftrace tool!

The essential snippet from a shell script:

```
...
# Global settings
EV_CLASSES="-e block -e fs -e ext4 -e filemap -e vfs -e hda -e scsi"
FN_NOT_TRACE="do_page_fault -n __do_page_fault"
BUFFER_SZ_KB_PCPU=9000
...
run_trace_cmd_2()
{
#--- CMD 2
CMD2="sync"
FTRC_REPORT2=./blk-mmc-report.sync_cmd.txt
CMD2_RAW_TRC_FILE=cmd2.dat

taskset 01 trace-cmd record ${EV_CLASSES} -v -e *vm* -n ${FN_NOT_TRACE}  -p
 function_graph -b ${BUFFER_SZ_KB_PCPU} -i -o ${CMD2_RAW_TRC_FILE} -F ${CMD2}

rm -f ${FTRC_REPORT2}
# -l => latency format
setup_report_header ${FTRC_REPORT2}
trace-cmd report -l -i ${CMD2_RAW_TRC_FILE} >> ${FTRC_REPORT2}
...
```

Sample Output – the report file (for the "sync" command):

```
...
    sync-25918   0d... 186650.476588: funcgraph_entry:                   |
scsi_init_io() {
    sync-25918   0d... 186650.476589: funcgraph_entry:                   |
scsi_init_sgtable() {
    sync-25918   0d... 186650.476589: funcgraph_entry:                   |
scsi_alloc_sgtable() {
    sync-25918   0d... 186650.476589: funcgraph_entry:                   |
scsi_sg_alloc() {
    sync-25918   0d... 186650.476589: funcgraph_entry:                   |
mempool_alloc() {
    sync-25918   0d... 186650.476590: funcgraph_entry:                   |
mempool_alloc_slab() {
    sync-25918   0d... 186650.476590: funcgraph_entry:        0.230 us   |
kmem_cache_alloc();
    sync-25918   0d... 186650.476591: funcgraph_exit:         0.856 us   |
}
```

```
    sync-25918   0d... 186650.476591: funcgraph_exit:        1.481 us   |
}
...
```

As the above output is difficult to interpret (due to long line lengths that wrap), we use awk to eliminate the date-timestamp and the “funcgraph_*” fields. Unfortunately, doing this trivially with awk also results in our losing the correct space-based indentation. Oh, well. The result:

```
# awk '{print $1, $2, $5, $6, $7, $8}' ./tmp1
-----------------------------------------------------------------
TASK/PID   CPU  DURATION FUNCTION CALLS
           ||          |                   |
           |/----=> cpu#                   |
           ||_-----=> irqs-off             |
           ||/ _----=> need-resched        |
           |||/ _---=> hardirq/softirq     |
           ||||/ _--=> preempt-depth       |
           |||||/                          |
...
sync-25918 0d... | scsi_init_io() {
sync-25918 0d... | scsi_init_sgtable() {
sync-25918 0d... | scsi_alloc_sgtable() {
sync-25918 0d... | scsi_sg_alloc() {
sync-25918 0d... | mempool_alloc() {
sync-25918 0d... | mempool_alloc_slab() {
sync-25918 0d... 0.230 us | kmem_cache_alloc();
sync-25918 0d... 0.856 us | }
sync-25918 0d... 1.481 us | }
sync-25918 0d... 2.092 us | }
sync-25918 0d... 0.143 us | memset();
sync-25918 0d... 3.496 us | }
sync-25918 0d... | blk_rq_map_sg() {
sync-25918 0d... 0.095 us | __blk_segment_map_sg();
sync-25918 0d... 0.811 us | }
sync-25918 0d... 5.523 us | }
sync-25918 0d... 6.241 us | }
sync-25918 0d... + 15.772 us |
sync-25918 0d... 0.085 us | scsi_prep_return();
sync-25918 0d... + 17.146 us |
sync-25918 0d... + 20.669 us |
sync-25918 0d... | blk_start_request() {
sync-25918 0d... 0.117 us | blk_dequeue_request();
sync-25918 0d... | blk_add_timer() {
sync-25918 0d... | __blk_add_timer() {
sync-25918 0d... | round_jiffies_up() {
sync-25918 0d... 0.083 us | round_jiffies_common();
sync-25918 0d... 0.703 us | }
sync-25918 0d... 1.323 us | }
sync-25918 0d... 1.943 us | }
sync-25918 0d... 3.228 us | }
sync-25918 0d... 0.083 us | _raw_spin_unlock();
sync-25918 0d... 0.081 us | _raw_spin_lock();
sync-25918 0.... 0.110 us | scsi_init_cmd_errh();
sync-25918 0.... | scsi_dispatch_cmd() {
sync-25918 0.... 0.084 us | scsi_log_send();
sync-25918 0.... [FAILED TO PARSE] host_no=0
sync-25918 0.... | ata_scsi_queuecmd() {
sync-25918 0.... 0.087 us | _raw_spin_lock_irqsave();
sync-25918 0d... | ata_scsi_find_dev() {
sync-25918 0d... | __ata_scsi_find_dev() {
sync-25918 0d... 0.080 us | ata_find_dev.part.16();
sync-25918 0d... 0.691 us | }
sync-25918 0d... 1.316 us | }
sync-25918 0d... 0.118 us | ata_qc_new_init();
sync-25918 0d... 0.080 us | ata_sg_init();
sync-25918 0d... | ata_scsi_rw_xlat() {
sync-25918 0d... 0.111 us | ata_build_rw_tf();
sync-25918 0d... 0.824 us | }
sync-25918 0d... | ahci_pmp_qc_defer() {
sync-25918 0d... 0.081 us | ata_std_qc_defer();
sync-25918 0d... 0.698 us | }
sync-25918 0d... | ata_qc_issue() {
sync-25918 0d... 0.083 us | page_address();
sync-25918 0d... | nommu_map_sg() {
sync-25918 0d... 0.085 us | check_addr();
sync-25918 0d... 0.754 us | }
sync-25918 0d... | ahci_qc_prep() {
sync-25918 0d... 0.108 us | ata_tf_to_fis();
sync-25918 0d... 0.766 us | }
sync-25918 0d... 0.097 us | ahci_qc_issue();
sync-25918 0d... 4.217 us | }
sync-25918 0d... 0.120 us | _raw_spin_unlock_irqrestore();
sync-25918 0.... + 12.432 us |
sync-25918 0.... + 14.127 us |
sync-25918 0.... 0.085 us | _raw_spin_lock_irq();
sync-25918 0d... | blk_peek_request() {
sync-25918 0d... 0.120 us | deadline_dispatch_requests();
sync-25918 0d... 0.781 us | }
...
#
```

Ftrace, additional Resources

[LWN, Steven Rostedt]

Debugging the kernel using Ftrace - part 1
Debugging the kernel using Ftrace – part 2
Secrets of the Ftrace function tracer

(Cgroup) Subsystems and Tunable Parameters

Subsystems are kernel modules that are aware of cgroups. Typically, they are resource controllers that allocate varying levels of system resources to different cgroups. However, subsystems could be programmed for any other interaction with the kernel where the need exists to treat different groups of processes differently.

The application programming interface (API) to develop new subsystems is documented in cgroups.txt in the kernel documentation, installed on your system at */usr/share/doc/kernel-doc-kernel-version/Documentation/cgroups/* (provided by the kernel-doc package). The latest version of the cgroups documentation is also available on line at
http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt . Note, however, that the features in the latest documentation might not match those available in the kernel installed on your system.

State objects that contain the subsystem parameters for a cgroup are represented as pseudofiles within the cgroup virtual file system. These pseudo-files can be manipulated by shell commands or their equivalent system calls.

For example, *cpuset.cpus* is a pseudo-file that specifies which CPUs a cgroup is permitted to access. If /cgroup/cpuset/webserver is a cgroup for the web server that runs on a system, and the following command is executed:

```
# echo 0,2 > /cgroup/cpuset/webserver/cpuset.cpus
```

The value 0,2 is written to the cpuset.cpus pseudofile and therefore limits any tasks whose PIDs are listed in */cgroup/cpuset/webserver/tasks* to use only CPU 0 and CPU 2 on the system.
...

Block I/O Subsystem Tuning

See the section on " this excellent RedHat "Resource Management Guide" for a very detailed perspective on managing resource allocators or Subsystems and Tuning via Cgroups.
[Of course, it's specific to RHEL6; still very useful.]

https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch-Subsystems_and_Tunable_Parameters.html#sec-blkio

…

3.1. blkio

The Block I/O (blkio) subsystem controls and monitors access to I/O on block devices by tasks in cgroups. Writing values to some of these pseudofiles limits access or bandwidth, and reading values from some of these pseudofiles provides information on I/O operations.

The blkio subsystem offers two policies for controlling access to I/O:

Proportional weight division — implemented in the Completely Fair Queuing I/O scheduler, this policy allows you to set weights to specific cgroups. This means that each cgroup has a set percentage (depending on the weight of the cgroup) of all I/O operations reserved. For more information, refer to Section 3.1.1, "Proportional Weight Division Tunable Parameters"

I/O throttling (Upper limit) — this policy is used to set an upper limit for the number of I/O operations performed by a specific device. This means that a device can have a limited rate of read or write operations. For more information, refer to Section 3.1.2, "I/O Throttling Tunable Parameters".

!NOTE!

Currently, the Block I/O subsystem does not work for buffered write operations. It is primarily targeted at direct I/O, although it works for buffered read operations.

…

Appendix A :: Sectors, Blocks and Segments

<< ULK3 >>

In this chapter we describe the lower kernel components that handle the block devices - generic block layer, I/O scheduler, and block device drivers - thus we focus our attention on sectors, blocks, and segments.

14.1.1. Sectors

To achieve acceptable performance, hard disks and similar devices transfer several adjacent bytes at once. Each data transfer operation for a block device acts on a group of adjacent bytes called a sector. In the following discussion, we say that groups of bytes are adjacent when they are recorded on the disk surface in such a manner that a *single seek operation can access them*. Although the physical geometry of a disk is usually very complicated, the hard disk controller accepts commands that refer to the disk as a large array of sectors.

In most disk devices, the size of a sector is 512 bytes, although there are devices that use larger sectors (1,024 and 2,048 bytes). Notice that the **sector** should be considered as the *basic (atomic) unit of data transfer*; it is never possible to transfer less than one sector, although most disk devices are capable of transferring several adjacent sectors at once.

In Linux, the size of a sector is conventionally set to 512 bytes; if a block device uses larger sectors, the corresponding low-level block device driver will do the necessary conversions. Thus, a group of data stored in a block device is identified on disk by its position - the index of the first 512-byte sector - and its length as number of 512-byte sectors. Sector indices are stored in 32- or 64-bit variables of type `sector_t`.

14.1.2. Blocks

While the sector is the basic unit of data transfer for the hardware devices, the **block** is the *basic (atomic) unit of data transfer for the VFS* and, consequently, for the filesystems. For example, when the kernel accesses the contents of a file, it must first read from disk a block containing the disk inode of the file (see the section "Inode Objects" in Chapter 12). This block on disk corresponds to one or more adjacent sectors, which are looked at by the VFS as a single data unit.

In Linux, the block size must be a power of 2 and cannot be larger than a page frame. Moreover, it must be a multiple of the sector size, because each block must include an integral number of sectors. Therefore, on the x86 architecture, the permitted block sizes are 512, 1024, 2048, and 4096 bytes.

The block size is not specific to a block device. When creating a disk-based filesystem, the administrator may select the proper block size. Thus, several partitions on the same disk might make use of different block sizes. Furthermore, each read or write operation issued on a block device file is a "raw" access that bypasses the disk-based filesystem; the kernel executes it by using blocks of largest size (4,096 bytes).

Each block requires its own block buffer, which is a RAM memory area used by the kernel to store the block's content. When the kernel reads a block from disk, it fills the corresponding block buffer with the values obtained from the hardware device; similarly, when the kernel writes a block on disk, it updates the corresponding group of adjacent bytes on the hardware device with the actual values of the associated block buffer. The size of a block buffer always matches the size of the corresponding block.

<<
With regard to the above, see the output of the “free” command:

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:          2026       1456        569          0         52        951
-/+ buffers/cache:        452       1573
Swap:          996          0        995
$
```

It helps shows us how the “buffers” item differs from the “cached” item heading; “buffers” relates to RAM (page frames) used by the kernel to hold a block's content; “cached” refers to any and all RAM taken up by kernel caches (for performance) – the dcache, icache, page (disk) cache, etc. When file I/O is high (like when copying large amounts of data), the “buffers” memory increases.
>>

--*snip*--

14.1.3. Segments

We know that each disk I/O operation consists of transferring the contents of some adjacent sectors from - or to - some RAM locations. **In almost all cases, the data transfer is directly performed by the disk controller with a DMA operation** (see the section "Direct Memory Access (DMA)" in Chapter 13). The block device driver simply triggers the data transfer by sending suitable commands to the disk controller; once the data transfer is finished, the controller raises an interrupt to notify the block device driver.

The data transferred by a single DMA operation must belong to sectors that are adjacent on disk. This is a physical constraint: a disk controller that allows DMA transfers to non-adjacent sectors would have a poor transfer rate, because moving a read/write head on the disk surface is quite a slow operation.

Older disk controllers support "simple" DMA operations only: in each such operation, data is transferred from or to memory cells that are physically contiguous in RAM. **Recent disk controllers, however, may also support the so-called *scatter-gather DMA transfers* : in each such operation, the data can be transferred from or to several noncontiguous memory areas.**

For each scatter-gather DMA transfer, the block device driver must send to the disk controller:

- The initial disk sector number and the total number of sectors to be transferred
- A list of descriptors of memory areas, each of which consists of an address and a length.

The disk controller takes care of the whole data transfer; for instance, in a read operation the controller fetches the data from the adjacent disk sectors and scatters it into the various memory areas.

To make use of scatter-gather DMA operations, block device drivers must handle the data in units called *segments*. A segment is simply a memory page - or a portion of a memory page - that includes the data of some adjacent disk sectors. Thus, a scatter-gather DMA operation may involve several segments at once.

Notice that a block device driver does not need to know about blocks, block sizes, and block buffers. Thus, even if a segment is seen by the higher levels as a page composed of several block buffers, the block device driver does not care about it.

As we'll see, the generic block layer can merge different segments if the corresponding page frames happen to be contiguous in RAM and the corresponding chunks of disk data are adjacent on disk. The larger memory area resulting from this *merge operation* is called **physical segment**.

Yet another merge operation is allowed on architectures that handle the mapping between bus addresses and physical addresses through a dedicated bus circuitry (the IO-MMU; see the section "Direct Memory Access (DMA)" in Chapter 13). The memory area resulting from this kind of merge operation is called hardware segment. Because we will focus on the x86 architecture, which has no such dynamic mapping between bus addresses and physical addresses, we will assume in the rest of this chapter that hardware segments always coincide with physical segments.