Linux Kernel Hacking Free Course, 3rd edition

R. Gioiosa
University of Rome “Tor Vergata”

Drivers for block devices

March 22, 2006

Index

- How to handle block devices
- Block device data structures
- a RAMDISK-based block device driver
- Using the RAMDISK block device

Class of devices

From previous lectures:

char device Data can be read as byte streams and random accesses are uncommon

block device slow, disk-like devices in which data can be accessed in blocks; random accesses are common

Block device drivers

Block device drivers are, usually, much more complex than character device drivers because the kernel must somehow speed up the slow disk data transfer.

- reads are quick enough only if they are sequential
- seek operations are time consuming because they require to move the disk heads
- writes are generally non blocking, so they can be optimized (*write back*)
- The kernel uses complex data structure (*disk cache*) and different scheduling algorithms to reduce the disk latency

All of these items must be considered when writing a block device driver.

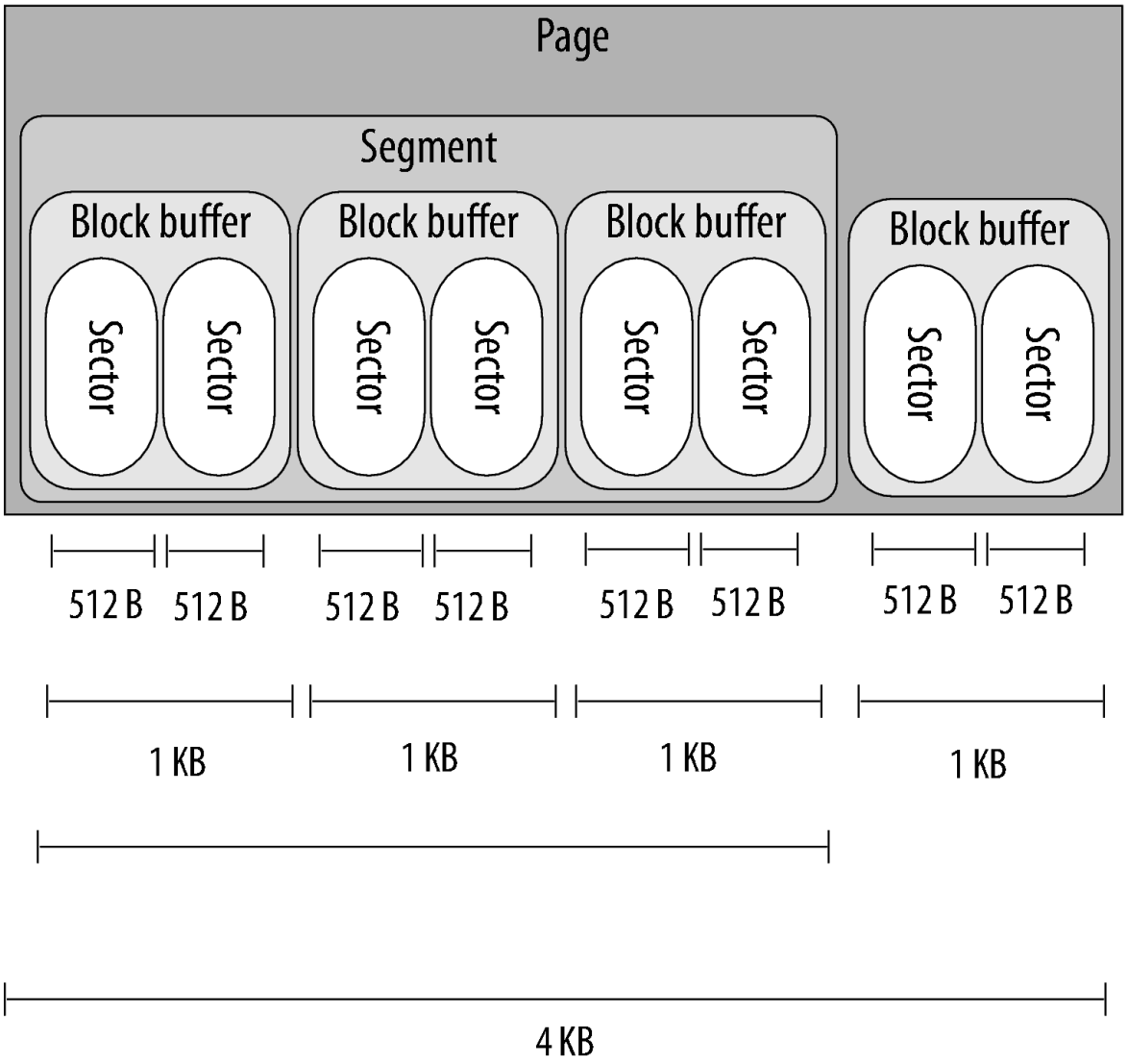
Data unit

Moving single bytes between disk and RAM is expensive, thus groups of bytes are transfered at once between RAM and disk.

sector Minimum set of bytes that can be transfered or addressed within a single disk I/O operation (usually 512 bytes)

block Amount of data that can be transfered within a single VFS operation (between 512 and `page_size`)

page Data into the page cache are stored in pages (4096 bytes)

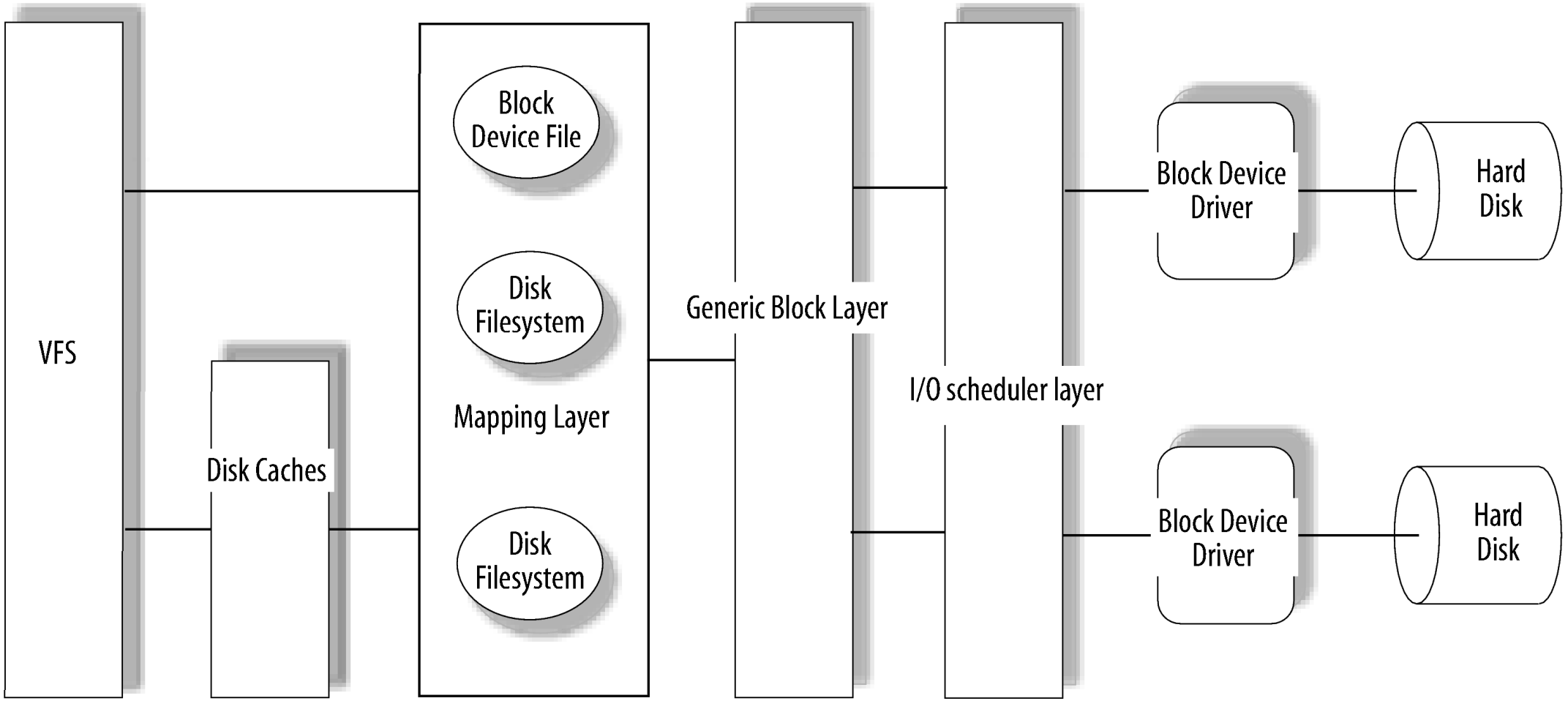

Page
Segment
Block buffer
Block buffer
Block buffer
Block buffer
Sector
Sector
Sector
Sector
Sector
Sector
Sector
Sector
512 B
512 B
512 B
512 B
512 B
512 B
512 B
512 B
1 KB
1 KB
1 KB
1 KB
4 KB

Block device driver operations

A block device driver has to:

1. Provide the file operations used by programs in order to interact with the block device (`open()`, `release()`, `ioctl()`,...). This is the higher part of the driver. Note that no `read()` or `write()` methods are provided because they are part of the filesystem interface (`ext3`, `vfat`, etc...)
2. Handle the disk accesses by programming the I/O controller. This is the lower part of the driver.

The higher part interfaces with the VFS and with the Generic Block Layer; the lower part interfaces with the kernel I/O Scheduler and with the hardware.


VFS
Disk Caches
Block Device File
Disk Filesystem
Mapping Layer
Disk Filesystem
Generic Block Layer
I/O scheduler layer
Block Device Driver
Hard Disk
Block Device Driver
Hard Disk

Bldex: a simple RAMDISK based block device

Now it's time to implement a virtual block device driver based on a RAMDISK. The required steps are:

1. Definition of data structures
2. Driver registration
3. Obtaining a `gendisk` object
4. Implement the driver's methods
5. Handle the request queue

Bldex: step 1, data definition

Bldex has a private data structure where all the information related to the device are stored.

```
struct bldex_desc_t {
        int major;
        int size_in_sectors;
        u8 *data;
        int users;
        spinlock_t lock;
        struct request_queue *queue;
        struct gendisk *gd;
};
```

Bldex: step 2, registration

A block device driver must register itself by specifying a `major number` and a `name` for the disk.

```
register_blkdev(major,name)
```

Giving zero as `major` will allocate the major number dynamically. The function

```
unregister_blkdev()
```

unregisters the device driver.

Gendisk

Each block device is handled as if it were a disk, independently from its technology.

The `gendisk` object stores all information related to device handling, such as the major and minor numbers, the disk capacity, etc.

From the developer's point of view, a single `gendisk` object is required, which represents the whole disk.

Bldex: step 3, obtaining the gendisk object

The function

```
alloc_disk(max_minor)
```

allocates a new `gendisk` object for the block device driver. Then, the new device must be activated using:

```
add_disk()
```

At this point the driver must be ready to handle I/O requests for the disk because `add_disk()` issues read requests while accessing the partition table.

Bldex: step 4, methods (1)

User programs interact with the block device by means of its file operations; the driver should provide at least:

- `open()`, `release()`
- `ioctl()` in order to implement all the operations that cannot be implemented as `read`/`write`
- `media_changed()`, `revalidate_disk()` if the device can be removed

Bldex: step 4, methods (2)

Furthermore the driver must also implement a transfer function triggered by the read and write operations.

All these methods are stored into a `block_device_operations` table that can be accessed by the `fops` field into the `gendisk` object.

Request queue (1)

I/O requests are queued into the block device request queue, ordered by disk sector position. When attempting to insert a new I/O request, the request queue is examined:

1. if there is already a request that can be merged with the new one, the old request is enlarged so that both the old and the new request will be handled together.
2. if there is no request that can be enlarged, a new request is added to the queue.

A request must be refer to contiguous disk sectors however it may encompass scattered data buffers in memory.

Request queue (2)

Each request in the request queue is a `struct request` table that stores lot of information, for instance:

- the current sector (i.e., the next sector that will be transfered)
- how many sectors can be transfered together with the current sector (all these sectors are related to the same buffer in RAM)
- the memory address where data have to be written (`read()`) or read (`write()`)
- the transfer direction (to/from the disk)

Bldex: step 5, the strategy routine

A strategy routine handles the request stored in the device request queue; this routine is provided when the queue is allocated:

```
blk_init_queue(routine,spin_lock)
```

The `spin_lock` is used by the strategy routine to lock the queue when fetching the next request to be served.

Strategy routines are usually based on the elevator strategy: once a transfer has started in one direction, further requests in the same direction will be privileged.

The function `blk_cleanup_queue()` removes the request queue.