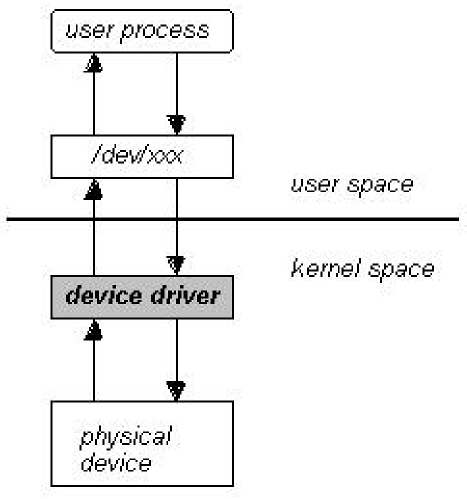
stackbd: Stacking a block device over another block device

stackbd is a virtual block device that acts as the front-end of another block device, such as a USB memory stick or a loop device. It passes I/O requests to the underlying device, while it prints the requests information for debugging purposes. Potentially it can also modify the requests.

The stacked block device (*stackbd*) is based on the code of the Linux Device Mapper, a block device in the Linux kernel that is supported by RedHat and is used to create logical volumes or, in other words, modify I/O requests' address values and target devices.

stackbd, for now, does not modify the requests. It acts as a sniffer and prints, for each request, whether its read/write, the block address, the number of pages, and the total size in bytes.

Other than debugging purposes, this simple device is a great way to learn about block device programming in the Linux Kernel.



Download source and build

First of all, it is best to work on a virtual machine since kernel hacking can cause the operating system to get stuck and a virtual machine is much faster to reboot.

Download the source (Or check out as Git or SVN) code from my GitHub:

https://github.com/OrenKishon/stackbd

Build the kernel module in the "module" directory and the user side util in directory "util".

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ make
```

```
oren@oren-VirtualBox:~/stackbd/trunk/util $ make
```

If you experience compile errors, please describe it in the comments.

Create a loop device for testing

We need a device to act as the underlying "real" device. The easiest thing to do is to create a loop device based on a file in the file system.

Create a 100 MB file, *disk_file*, that will be used as the device storage:

```
oren@oren-VirtualBox:~ $ dd if=/dev/urandom of=disk_file bs=1024 count=100000
```

Setup a loop device, */dev/loop0*, over this file:

```
oren@oren-VirtualBox:~ $ sudo losetup /dev/loop0 file_dev
```

Confirm that the device has been created with size 200,000 (512 bytes blocks):

```
oren@oren-VirtualBox~/stackbd/trunk/module $ sudo blockdev --getsize
/dev/loop0
200000
```

Notice that the loop device does not persist after a reboot, so once the file *disk_file* is created, only the *losetup*command needs to be repeated after a reboot.

Follow Kernel debug prints

The *stackbd* module prints debug messages using the *printk* command, so we need to follow them by tailing the *syslog* file. Open a second shell window, and move it so you can see the messages in parallel to executing commands in the first shell window. In the second terminal:

```
oren@oren-VirtualBox:~ $ tail -f /var/log/syslog
```

All of the following commands in this post should yield debug messages in this file.

Initialize and stack the device

Insert the module into the Kernel. This will only create the new device */dev/stackbd0* but will not associate it with another device:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ sudo insmod ./stackbd.ko
```

Use the user-side util to have *stackbd* open the loop device. It uses *ioctl* commands to control the kernel module:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ sudo ../util/stackbd_util
/dev/loop0
```

Confirm that the new device, */dev/stackbd0*, exists and it has the same size as the underlying device:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ sudo blockdev --getsize
/dev/stackbd0
```

200000

The messages in *syslog* for the two above commands should look like:

```
Oct 29 10:06:04 oren-VirtualBox kernel: [  753.775151] stackbd: init done
Oct 29 10:06:10 oren-VirtualBox kernel: [  759.671404]
Oct 29 10:06:10 oren-VirtualBox kernel: [  759.671404] *** DO IT!!!!!!! ***
Oct 29 10:06:10 oren-VirtualBox kernel: [  759.671404]
Oct 29 10:06:10 oren-VirtualBox kernel: [  759.671417] Opened /dev/loop0
Oct 29 10:06:10 oren-VirtualBox kernel: [  759.671425] stackbd: Device real
capacity: 200000
Oct 29 10:06:10 oren-VirtualBox kernel: [  759.671427] stackbd: Max sectors:
255
Oct 29 10:06:10 oren-VirtualBox kernel: [  759.671461] stackbd: done
initializing successfully
```

Mount the device and use it

First, create a directory, *mnt*, in the home directory for the mount. It needs to be done once, since it remains after reboots:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ mkdir ~/mnt
```

Create a file system, of type *ext4* for the example, on the device:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ sudo mkfs.ext4 /dev/stackbd0
```

Mount the file system on directory *mnt*:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ sudo mount -t ext4 /dev/stackbd0
~/mnt/
```

Give read and write permissions for non-root user on the mount point:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ sudo chmod -R 777 ~/mnt/
```

Create a file and write to it in the device. Afterwards, read the file.

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ echo test > ~/mnt/1.txt
```

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ cat ~/mnt/1.txt
```

test

During the above operations, view the debug prints detailing the I/O requests. An example:

```
Oct 29 10:09:58 oren-VirtualBox kernel: [  987.765196] stackbd: make request
read  block 544  #pages 1  total-size 1024
Oct 29 10:10:00 oren-VirtualBox kernel: [  989.880441] stackbd: make request
write block 586  #pages 31 total-size 126976
Oct 29 10:10:00 oren-VirtualBox kernel: [  989.880616] stackbd: make request
write block 834  #pages 29 total-size 117760
Oct 29 10:10:00 oren-VirtualBox kernel: [  989.880742] stackbd: make request
write block 1064 #pages 31 total-size 126976
Oct 29 10:10:00 oren-VirtualBox kernel: [  989.880840] stackbd: make request
write block 1312 #pages 30 total-size 119808
```

Unmount and remove device

In order to re-test the device, for instance after code modification, it can be removed and than remounted.

Unmount the file system:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ sudo umount /dev/stackbd0
```

Remove the module, which will remove the device */dev/stackbd0*:

```
oren@oren-VirtualBox:~/stackbd/trunk/module $ sudo rmmod stackbd
```

Interesting kernel code snippets

Opening the underlying block device inside this block device, using its path (in the example here, the path is */dev/loop0*). The functions used to open a block device are `lookup_dev()`, `bdget()` and `blkdev_get()`:

```
    struct block_device *bdev_raw = lookup_bdev(dev_path);
    printk("Opened %s\n", dev_path);
    if (IS_ERR(bdev_raw))
    {
        printk("stackbd: error opening raw device <%lu>\n", PTR_ERR(bdev_raw));
        return NULL;
    }
    if (!bdget(bdev_raw->bd_dev))
    {
        printk("stackbd: error bdget()\n");
        return NULL;
    }
    if (blkdev_get(bdev_raw, STACKBD_BDEV_MODE, &stackbd))
    {
        printk("stackbd: error blkdev_get()\n");
        bdput(bdev_raw);
        return NULL;
    }
    return bdev_raw;
```

The actual remapping of an I/O request from this block device to the underlying block device. The function `trace_block_bio_remap()` simply modifies the request's target device and address, and sends the request to the other device's queue (using `generic_make_request()`):

```
static void stackbd_io_fn(struct bio *bio)
{
    bio->bi_bdev = stackbd.bdev_raw;

    trace_block_bio_remap(bdev_get_queue(stackbd.bdev_raw), bio,
            bio->bi_bdev->bd_dev, bio->bi_sector);

    /* No need to call bio_endio() */
    generic_make_request(bio);
}
```

The block device queue function. Block devices process requests asynchronously (unlike char devices). They define a request callback and register it with the queue. The kernel calls this callback for I/O. This function acts as a producer thread as it only adds the I/O request to an internal list (`struct bio list`) and does not handle it. It signals the other thread which acts as a consumer to actually perform the I/O.

```
static void stackbd_make_request(struct request_queue *q, struct bio *bio)
{
    spin_lock_irq(&stackbd.lock);
    if (!stackbd.bdev_raw)
    {
        printk("stackbd: Request before bdev_raw is ready, aborting\n");
        goto abort;
    }
    if (!stackbd.is_active)
    {
        printk("stackbd: Device not active yet, aborting\n");
        goto abort;
    }
    bio_list_add(&stackbd.bio_list, bio);
    wake_up(&req_event);
    spin_unlock_irq(&stackbd.lock);

    return;

abort:
    spin_unlock_irq(&stackbd.lock);
    printk("<%p> Abort request\n\n", bio);
    bio_io_error(bio);
}
```

The block device “consumer” thread function – waits for a signal from the “producer” thread, which is the actual queue thread, that a request has been added to the list. `wait_event_interruptible()` is the function that sleep-waits for the queue thread to signal it to wake up.

```
static int stackbd_threadfn(void *data)
{
    struct bio *bio;
    while (!kthread_should_stop())
    {
        /* wake_up() is after adding bio to list. No need for condition */
        wait_event_interruptible(req_event, kthread_should_stop() ||
                !bio_list_empty(&stackbd.bio_list));

        spin_lock_irq(&stackbd.lock);
        if (bio_list_empty(&stackbd.bio_list))
        {
            spin_unlock_irq(&stackbd.lock);
            continue;
        }

        bio = bio_list_pop(&stackbd.bio_list);
        spin_unlock_irq(&stackbd.lock);

        stackbd_io_fn(bio);
    }
    return 0;
}
```

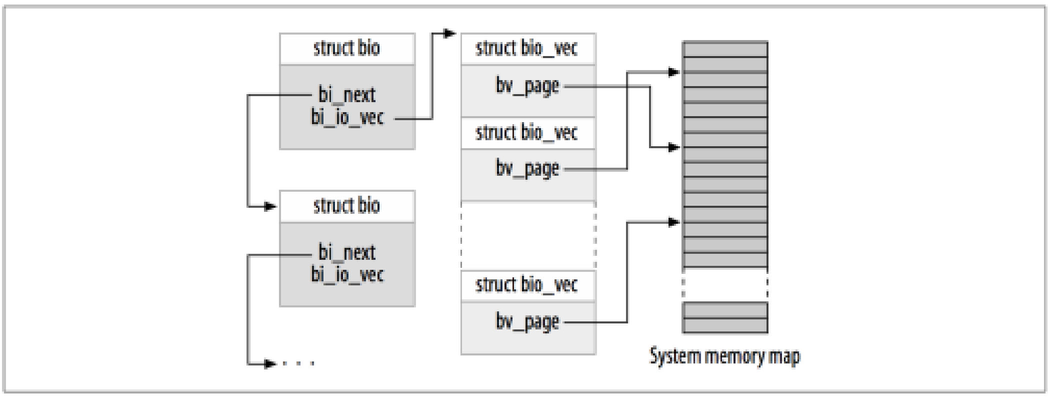


Figure 16-1. The bio structure

Figure taken from Linux Device Drivers book

Future development

The next modification I intend to do is to create a counter of I/O requests for each block. The implementation will be to take some of the underlying device's capacity for storing metadata blocks that will hold the counters for each block. If there will be two 4 bytes counters, for read and for write, for each block, then for every 512 byte block there will be 512 / 8 = 64 bytes of metadata. It means that for every 8 blocks there will be a metadata block.

The requests' addresses will be need to be modified to some different constant offset and every request will be followed by two requests to the underlying device: one that will actually write/read the data and one that will modify the counters blocks.