Assignment 2: Information Assuring Block Device Driver
CS 519
Mandanna Thekkada, Karan Desai

Abstract
The main idea being our solution is that we can create a virtual driver that will wrap the original driver and we will be monitoring the request queue of the original driver for IO requests and then use this information to check for changes that the specific IO operation will possibly make. These changes are logged with the kernel.

Summary of Workflow
First we started reading about Linux drivers from the reference Linux Device Drivers and various online resources. Starting with basic module setup, coding a basic driver to Block and even Char Drivers. Most of the code that we wrote will resemble the ideas presented in this book.

We ended up reading and learning much more than we had expected, starting from the very scratch and learning some complex ideas about how block IO works and what happens at every single step. We read a great deal about the virtual file systems, about how the operating system coalesces multiple IO requests to the same block device into a single request and thus learning further.

The main aspects of the scheme that we are implementing is described briefly

INITIALIZATION AND REQUEST QUEUE

- The internal representation of the device is done which declares the block device, the queue and other pointers. Which is the struct my_device

- The initiation function declares the size, allocates the memory and initiates the sync locks.

  static int __init my_init(void)

- Once the basic structure of our device is ready we need the reference to the original device that is obtained by the following function :

  Device.blkdev=blkdev_get_by_path(name,FMODE_READ|FMODE_WRITE|FMODE_EXCL,&Device);

  Here Device is our own implementation of a block device and *name* is a command line parameter that contains the entire filepath of the device to be monitored.

Eg:- like */dev/sda*

Here we get the queue for the actual device driver by calling the function

bdev_get_queue(Device.blkdev);

Here blkdev is the return value from ***blkdev_get_by_path***

- We copy the *gendisk's queue* with all of the values from the queue's (just returned by bdev_get_queue) by using the following

  Device.Queue = blk_alloc_queue(GFP_KERNEL);

  We need to have our own queue beforehand so that our device driver is able to receive requests.

  Since our device driver is a virtual device driver, we need to tell the queue that whenever a request comes in to call a function

  blk_queue_make_request(Device.Queue, bd_make_request);

- Now we can refer to the request Queue of the original driver and have what all we need to achieve our goal

- The registration, allocation, initialization of the gendisk structure, geometry of the device, etc. all get done in the remaining portion of the init module.

ACCESSING THE DATA OF EACH IO REQUEST

The Queue of the original device that we have obtained is made of instances of struct request. Each request structure represents one block I/O request although they may have been formed by the merger of multiple IO requests.

A request structure has a multitude of fields but it is primarily made of a linked list of structures of type struct bio. The bio structure is a low-level description of a portion of a block IO request.

How the Bio Structure Works -

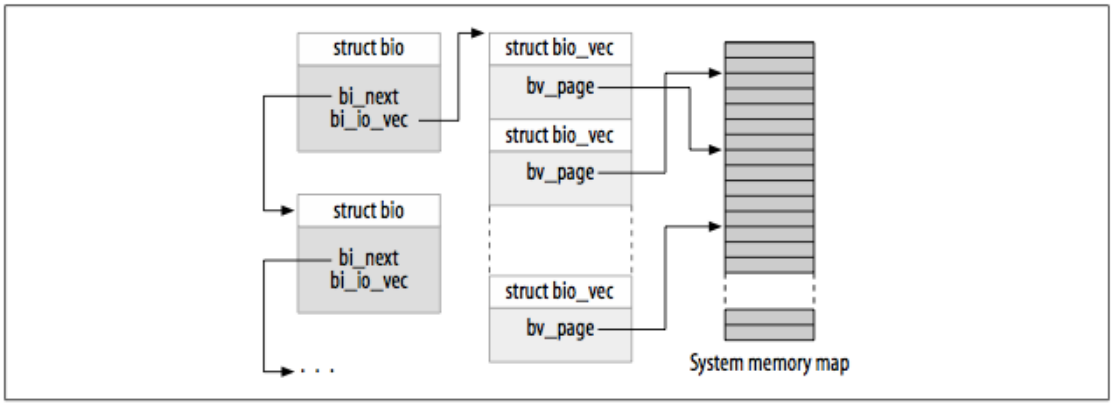

Figure 16-1. The bio structure

The bio structure is defined in <linux/bio.h>. It contains everything that the driver needs to know to perform the required IO operation.

The main part of the bio structure is the bio_io_vec array.

We can step through the bio_io_vec array to get access structures of type bio_vec which gives us information about the required data from each page from which data is to be transferred. This is given by the following fields:

- Offset – which is the offset in the page from where the data transfer begins.
- len - number of bytes to be transferred

Working directly with the bio structure is not preferred. Hence, to walk through the bio_io_vec array we use the following macro :

```
bio_for_each_segment(bvec, bio, i)
```

Here 'i' is the current segment number.

We use loop to compare pre and post images of IO being performed. The device driver that currently handles the current bio structure would be ours. For every bio_vec structure we perform (monitoring/comparison) via a call to the **my_*transfer*** function and then we submit the bio to the original request queue of the device being monitored. We also listen for the completion status of the bio structure via the call to the bio_endio function.

The copying from the bio and comparing is done as described below. We will trace it from the request function.

- First we create our own request function

  static void bd_make_request (struct request_queue *q, struct bio *bio)

- Apart from doing the generic work, what this function does is assign the device driver you got from *blkdev_get_by_path* to the bio received.

  bio->bi_bdev = Device.blkdev;

- Then we run a Kernel Thread inside which we call our ***bookkeeping function.***

- In the Bookkeeping function we clone the received bio and perform a copy of the same transfer on our virtual device

  struct bio * temp_bio=bio_clone(bio,GFP_KERNEL);

- The **my_bd_xfer_bio** is called which is where all the work regarding the **__bio_kmap_atomic** is done. This is the low-level function allows you to directly map the buffer found in a given **bio_vec**, as indicated by the index i. An atomic kmap is created.

- We then call the **my_transfer** function where all the work of IO request handling is being done and also regarding the comparison and logging. The pre and post images are compared and the changes are logged.

  Note that we are only paying attention to the write IO commands because they are the only ones that will make any changes.

  In the **my_transfer** we create a temporary buffer. We copy the data from the old device onto this temp_buffer and compare

  memcpy(temp_buffer1, dev->data + offset, nbytes)

  strcmp(temp_buffer1,buffer)

  The temp_buffer is compared with the buffer that we currently have with the device and any changes are logged onto the system. All of this is being done only when the IO request is a write.

Once all these functions are executed we do

submit_bio(bio_data_dir(bio), bio);

followed by -

bio_endio(bio,status);

This function is executed only after all the monitoring on the bio is done. The bio is now resubmitted back to the original device for the actual IO request to be performed.

In the exit function which is **__exit my_exit(void),** we delete the gendisk, unregister the device, free the queue and stop the kernel thread

COMPILING THE CODE

Use the following commands to compile and installing:

- make
- sudo insmod iablockdrv.ko name="/dev/sda" start_sector=3 range_sector=2
- dmesg (to check the kernel buffer)

Note the three run time parameters that have to be given

CONCLUSION

We did implement a block driver that satisfies the purpose. We were able to learn a lot from this assignment and got a very clear picture of the concepts. Though there is a lot of room for improvement we were not able to fully implement some of the features that were needed. For example we could have used dequeue-enqueue functions on the request queue to avoid the race conditions that will occur but then would have to deal with the overhead.

REFERENCES

[1] http://lwn.net/Kernel/LDD3/
[2] http://blog.superpat.com/2010/05/04/a-simple-block-driver-for-linux-kernel-2-6-31/
[3] http://free-electrons.com/doc/block_drivers.pdf
[4] http://www.linuxjournal.com/article/2890