

Introduction to Linux Block Drivers

Hao-Ran Liu



Sectors and blocks

- Sector

- The basic unit of data transfer for the hardware device
- Kernel expects a 512-byte sector. If you use a different hardware sector size, scale the kernel's sector numbers accordingly

- Block

- A group of adjacent bytes involved in an I/O operation
- Often 4096 bytes, can vary depending on the architecture and the exact filesystem being used



Block driver registration

```
int register_blkdev(unsigned int major, const char *name);
```

- Allocating a dynamic major number if requested
- Creating an entry in `/proc/devices`

```
int unregister_blkdev(unsigned int major, const char *name);
```

- In the 2.6 kernel, the call to `register_blkdev` is entirely optional
- A separate registration interface to register disk drives and block device operations



Block device operations

```
int (*open)(struct inode *inode, struct file *filp)
int (*release)(struct inode *inode, struct file *filp)
```

Called whenever the device is opened and closed. A block driver might spin up the device, lock the door (for removable media) in the open operation

```
int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd,
             unsigned long arg)
int (*media_changed)(struct gendisk *gd)
```

Check if the user has changed the media in the drive, returning a nonzero value if so

```
int (*revalidate_disk)(struct gendisk *gd)
```

This function is called in response to a media change. It gives the driver a chance to perform whatever work is required to make the new media ready for use

* Request function, register elsewhere, handles the actual read or write of data.



The gendisk structure

- the kernel's representation of an individual disk device
 - The kernel also uses **gendisk** structures to represent partitions

```
int major; int first_minor; int minors;
```

The first field is the major number of the device driver. A drive must use at least one minor number. A partitionable drive has one minor number for each possible partition. If minors = 16, it allows for the “full disk” device and 15 partitions

```
char disk_name[32];
```

The name of the disk device. It shows up in `/proc/partitions` and `sysfs`

```
struct block_device_operations *fops;  
struct request_queue *queue;
```

Structure used by the kernel to manage I/O requests for this device



The gendisk structure (cont.)

```
sector_t capacity;
```

The capacity of this drive, in 512-byte sectors

```
void *private_data;
```

Block drivers may use this field for a pointer to their own internal data



The gendisk API

```
struct gendisk *alloc_disk(int minors);
```

Allocation of the `struct gendisk` can only be done through this function. **Minors** is the number of minor numbers this disk uses

```
void del_gendisk(struct gendisk *gd);
```

Invalidates stuff in the gendisk and normally removes the final reference to the gendisk

```
void add_disk(struct gendisk *gd);
```

This function makes the disk available to the system. As soon as you call **add_disk**, the disk is “live” and its methods can be called at any time. So you should not call **add_disk** until your driver is completely initialized and ready to respond to requests on that disk.

Sbull – A real example

- The *sbull* driver implements a set of in-memory virtual disk drives
- You can download the example from O'Reilly's website

Sbull allows a major number to be specified at compile or module load time. If no number is specified, one is allocated dynamically.

```
sbull_major = register_blkdev(sbull_major, "sbull");
if (sbull_major <= 0) {
    printk(KERN_WARNING "sbull: unable to get major number\n");
    return -EBUSY;
}
```




Describing the sbull device

- The sbull device is described by an internal structure

```
struct sbull_dev {  
    int size;                /* Device size in bytes */  
    u8 *data;                /* The data array */  
    short users;             /* How many users */  
    short media_change;      /* Flag a media change? */  
    spinlock_t lock;         /* For mutual exclusion */  
    struct request_queue *queue; /* The device request queue */  
    struct gendisk *gd;       /* The gendisk structure */  
    struct timer_list timer;  /* For simulated media changes */  
};
```



Initialization of the sbull_dev

Basic initialization and allocation of the underlying memory

```
memset (dev, 0, sizeof (struct sbull_dev));
dev->size = nsectors*hardsect_size;
dev->data = vmalloc(dev->size);
if (dev->data == NULL) {
    printk (KERN_NOTICE "vmalloc failure.\n");
    return;
}
spin_lock_init(&dev->lock);

/* ... */

dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

Allocation of the request queue, **sbull_request** is our request function – the function that actually performs block read and write requests. The spinlock is provided by the driver because, often, the request queue and other driver data structures fall within the same critical section.

Initialization of the `sbull_dev`

(cont.)

Allocate, initialize and install the corresponding `gendisk` structure. `SBULL_MINORS` is the number of minor numbers each *sbull* device supports. The name of the disk is set such that the first one is *sbulla*, the second *sbullb*, and so on. Once everything is set up, we finish with a call to `add_disk`. Chances are that several of our methods will have been called for that disk by the time `add_disk` returns, so we take care to make that call the very last step in the initialization of our device.

```
dev->gd = alloc_disk(SBULL_MINORS);
if (! dev->gd) {
    printk (KERN_NOTICE "alloc_disk failure\n");
    goto out_vfree;
}
dev->gd->major = sbull_major;
dev->gd->first_minor = which*SBULL_MINORS;
dev->gd->fops = &sbull_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
snprintf (dev->gd->disk_name, 32, "sbull%c", which + 'a');
set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
```

A note on sector sizes

- The kernel always expresses itself in 512-byte sectors, but not all hardware uses that sector size. Thus, it is necessary to translate all sector numbers accordingly.

```
blk_queue_hardsect_size(dev->queue, hardsect_size);
```

Use this function to inform the kernel of the sector size your device supports. The hardware sector size is a parameter in the request queue. The *sbull* device exports a **hardsect_size** parameter that can be used to change the “hardware” sector size of the device.



A feature of the *sbull* device

- *Sbull* pretends to be a removable device
 - Whenever the last user closes the device, a 30-second timer is set; if the device is not opened during that time, the contents of the device are cleared, and the kernel will be told that the media has been changed

Sbull's block device operations

-- open()

The function maintains a count of users and calls **del_timer_sync** to remove the “media removal” timer. **check_disk_change** is a kernel function, which calls driver’s **media_changed** function to check if a removable media has been changed. In that case, it invalidates all buffer cache entries and calls driver’s **revalidate_disk** function.

```
static int sbull_open(struct inode *inode, struct file *filp)
{
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;

    del_timer_sync(&dev->timer);
    filp->private_data = dev;
    spin_lock(&dev->lock);
    if (! dev->users)
        check_disk_change(inode->i_bdev);
    dev->users++;
    spin_unlock(&dev->lock);
    return 0;
}
```

Sbull's block device operations

-- release()

The function, in contrast, decrement the user count and, if indicated, start the media removal timer. In a driver that handles a real hardware device, the **open** and **release** methods would set the state of the driver and hardware accordingly. A block device is opened when user space programs access the device directly (**mkfs**, **fdisk**, **fsck**) or when a partition on it is mounted.

```
static int sbull_release(struct inode *inode, struct file *filp)
{
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;

    spin_lock(&dev->lock);
    dev->users--;
    if (!dev->users) {
        dev->timer.expires = jiffies + INVALIDATE_DELAY;
        add_timer(&dev->timer);
    }
    spin_unlock(&dev->lock);
    return 0;
}
```

Sbull's block device operations

-- media_changed() & revalidate_disk()

If you are writing a driver for a nonremovable device, you can safely omit these methods. Both of these functions are called by **check_disk_change**. When a device is opened and the removable media has changed, the kernel will reread the partition table and start over with the device.

```
int sbull_media_changed(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;
    return dev->media_change;
}

int sbull_revalidate(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;

    if (dev->media_change) {
        dev->media_change = 0;
        memset (dev->data, 0, dev->size);
    }
    return 0;
}
```


Sbull's block device operations

-- `ioctl()`

- The higher-level block subsystem code intercepts a number of `ioctl` commands before your driver ever gets to see them
- *Sbull* `ioctl` method handles only one command – a request for the device's geometry
 - The kernel is not concerned with a block device's geometry; it sees it simply as a linear array of sectors
 - But certain user-space utilities still expect to be able to query a disk's geometry
 - Eg. the *fdisk* tool depends on cylinder information and does not function properly if that information is not available

Sbull's block device operations

-- ioctl() (cont.)

```
int sbull_ioctl (struct inode *inode, struct file *filp,
                unsigned int cmd, unsigned long arg)
{
    long size; struct hd_geometry geo;
    struct sbull_dev *dev = filp->private_data;

    switch(cmd) {
        case HDIO_GETGEO:
            /* Get geometry: since we are a virtual device, we have to make
             * up something plausible.  So we claim 16 sectors, four heads,
             * and calculate the corresponding number of cylinders.  We set the
             * start of data at sector four.
             */
            size = dev->size/KERNEL_SECTOR_SIZE;
            geo.cylinders = (size & ~0x3f) >> 6;
            geo.heads = 4; geo.sectors = 16; geo.start = 4;
            if (copy_to_user((void __user *) arg, &geo, sizeof(geo)))
                return -EFAULT;
            return 0;
    }
    return -ENOTTY; /* unknown command */
}
```

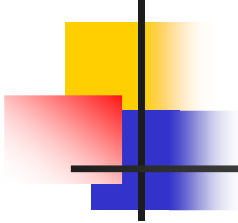


Request processing

-- request function

```
void request(request_queue_t *queue);
```

- The place where the real work gets done
- Does not need to complete all of the requests on the queue before it returns
 - But it must make a start on these requests and ensure that they are all, eventually, processed by the driver
- Invocation of the request function is (usually) entirely asynchronous with respect to the actions of any user-space process



Request processing

-- request queue

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

- Every device (usually) needs a request queue because:
 - Actual transfers to and from a disk can take place far away from the time the kernel requests them
 - Kernel needs the flexibility to schedule each transfer at the most propitious moment, grouping together requests that affect sectors close together on the disk (I/O scheduling)
- Whenever the request function is called, the queue lock is held by the kernel.
 - It prevents the kernel from queueing any other requests for your device
 - You may want to consider dropping the lock while the request function runs

Sbull's request function

-- a simple request method

This represents a block I/O request

It obtains the first incomplete request on the queue and returns NULL when there are no requests. It does not remove the request from the queue.

```
static void sbull_request(request_queue_t *q)
{
    struct request *req;

    while ((req = elv_next_request(q)) != NULL) {
        struct sbull_dev *dev = req->rq_disk->private_data;
        if (!blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        sbull_transfer(dev, req->sector, req->current_nr_sectors,
                      req->buffer, rq_data_dir(req));
        end_request(req, 1);
    }
}
```

Exclude non-filesystem request because we don't know how to handle it

The index of the beginning sector on our device (in 512-byte sector)

The number of (512-byte) sectors to be transferred

`sbull_transfer(dev, req->sector, req->current_nr_sectors, req->buffer, rq_data_dir(req));`

The request has been processed successfully

A pointer to the buffer to or from which the data should be transferred

The direction of the transfer from the request (0 = read)

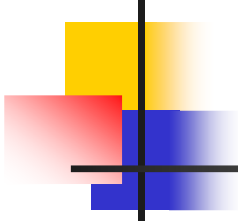
Sbull's request function

-- sbull_transfer()

```
static void sbull_transfer(struct sbull_dev *dev, unsigned long sector,
                          unsigned long nsect, char *buffer, int write)
{
    unsigned long offset = sector*KERNEL_SECTOR_SIZE;
    unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;

    if ((offset + nbytes) > dev->size) {
        printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n",
offset, nbytes);
        return;
    }
    if (write)
        memcpy(dev->data + offset, buffer, nbytes);
    else
        memcpy(buffer, dev->data + offset, nbytes);
}
```

Problems with the simple request function



- Executes requests synchronously, only 1 requests at a time
 - Some devices are capable of having numerous requests outstanding at the same time
- The largest single transfer never exceed the size of a single page



Request queue

- A queue for keeping block I/O requests
- Stores parameters that describe what kinds of requests the device is able to service
 - Maximum size
 - Maximum number of segments per request
 - Hardware sector size, alignment requirements
- A plug-in interface allowing the use of multiple I/O schedulers
 - Improve I/O performance by accumulating and sorting requests
 - Merge of adjacent requests



Queue creation and deletion functions

```
request_queue_t *blk_init_queue(request_fn_proc *request, spinlock_t *lock);
```

Create and initialize a request queue. The arguments are the request function for this queue and a spinlock that controls access to the queue. This function allocates memory and can fail because of this; you should always check the return value before attempting to use the queue

```
void blk_cleanup_queue(request_queue_t *);
```

Return a request queue to the system. After this call, your driver sees no more requests from the given queue and should not reference it again



Queueing functions

The queue lock must be hold before calling these functions

```
struct request *elv_next_request(request_queue_t *queue);
```

This function returns the next request to process or NULL if no more requests remain to be processed. The request returned is left on the queue but marked as being active; this mark prevents the I/O scheduler from attempting to merge other requests with this one

```
void blk_dequeue_request(struct request *req);
```

Remove a request from a queue. If your driver operates on multiple requests from the same queue simultaneously, it must dequeue them in this manner

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

Put a dequeued request back on the queue



Queue control functions

```
void blk_stop_queue(request_queue_t *queue);  
void blk_start_queue(request_queue_t *queue);
```

If your device has reached a state where it can handle no more outstanding commands, you can call **blk_stop_queue** to prevent the request function from being called until you call **blk_start_queue** to restart queue operations

```
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
```

This function tells the kernel the highest physical address to which your device can perform DMA. If a request comes in containing a reference to memory above the limit, a bounce buffer will be used for the operation. You can use these predefined symbols:

BLK_BOUNCE_HIGH : bounce all highmem pages.

BLK_BOUNCE_ANY : don't bounce anything

BLK_BOUNCE_ISA : bounce pages above ISA DMA boundary



Queue control functions (cont.)

```
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
```

These functions set parameters describing the requests that can be satisfied by this device.

blk_queue_max_sectors set the maximum size of any request in (512-byte) sectors; the default is 255. **blk_queue_max_phys_segments** and **blk_queue_max_hw_segments** both control how many physical segments (nonadjacent areas in system memory) may be contained within a single request. The first limit would be the largest sized scatter list the driver could handle, and the second limit would be the largest number of address/length pairs the host adapter can actually give as once to the device.



Queue control functions (cont.)

```
void blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
```

Some devices cannot handle requests that cross a particular size memory boundary. For example, if your device cannot handle requests that cross a 4-MB boundary, pass in a mask of **0x3ffffff**. The default mask is **0xffffffff**

```
void blk_queue_dma_alignment(request_queue_t *queue, int mask);
```

Tells the kernel the memory alignment constraints your device imposes on DMA transfers. All requests are created with the given alignment, and the length of the request also matches the alignment. The default mask is **0x1ff**, which causes all requests to be aligned on 512-byte boundaries

```
void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
```

Tells the kernel about your device's hardware sector size. All requests generated by the kernel are a multiple of this size and are properly aligned. All communications between the block layer and the driver continues to be expressed in 512-byte sectors, however.



The anatomy of a request

- Each **request** structure represents one block I/O request; it is:
 - A set of segments, each of which corresponds to one in-memory buffer
 - A set of consecutive sectors on the block device
 - Implemented as a linked list of **bio** structures with some information for the driver to keep track of its position as it works through the request



The **bio** -- uppermost interface used by filesystems

- **bio** is issued by filesystems, virtual memory, or a system call to read or write a block device
- It may be merged into an existing **request** structure or put into a newly created one



The **bio** structure

```
struct block_device *bi_bdev; sector_t bi_sector;
```

The block device to be read/write; The first (512-byte) sector to be transferred for this **bio**

```
unsigned int bi_size;
```

The size of the data to be transferred, in bytes. This macro **bio_sectors(bio)** returns the size of a **bio** in sectors

```
unsigned long bi_flags; unsigned long bi_rw;
```

Sets of flags describing the **bio**. The least significant bit of **bi_rw** is set if this is a write request. Use **bio_data_dir(bio)** to query the read/write flag

```
unsigned short bio_phys_segments; unsigned short bio_hw_segments;
```

The number of physical segments contained within this BIO and the number of segments seen by the hardware after DMA mapping is done, respectively

The bio structure (cont.)

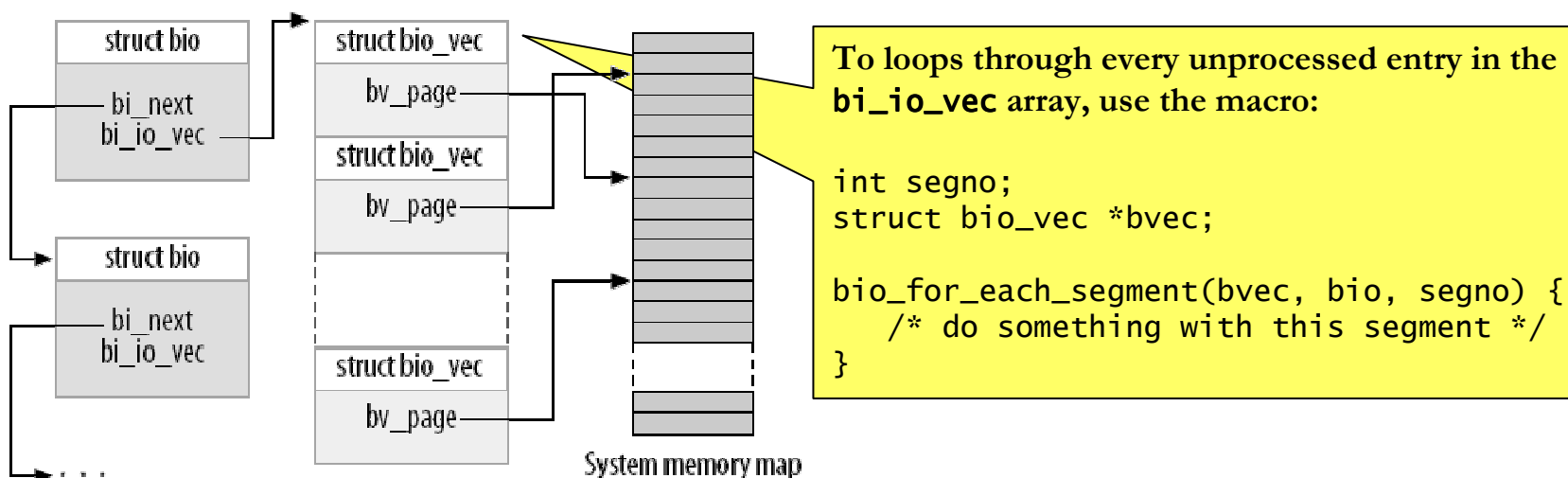
```
struct bio_vec *bi_io_vec;
```

```
struct bio_vec {
    struct page *bv_page;
    unsigned int bv_len;    // bytes to be transferred
    unsigned int bv_offset; // starting at bv_offset
}
```

An array of data structures indicating memory locations from which data is read or write

```
unsigned short bi_vcnt; unsigned short bi_idx;
```

The number of I/O vectors in **bi_io_vec**; Current I/O position in **bi_io_vec**



Mapping the buffer of a **bio**

- To access the pages in a **bio** directly, make sure that they have a proper kernel virtual address
 - Pages in high memory are not addressable

map the i-th buffer in **bi_io_vec** array

atomic kmap slot

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
```

Use these functions to ensure that the buffer in a given **bio** is addressable. An atomic kmap is created and a kernel virtual address is returned. The caller cannot sleep while the mapping is in used.

map current buffer as indicated in **bio->bi_idx**



Macros to read the current state of a `bio`

```
struct page *bio_page(struct bio *bio);
```

Returns a pointer to the **page** structure representing the page to be transferred next

```
int bio_offset(struct bio *bio);
```

Return the offset within the page for the data to be transferred

```
int bio_cur_sectors(struct bio *bio);
```

Returns the number of sectors to be transferred out of the current page

```
char *bio_data(struct bio *bio);
```

Returns a kernel logical address pointing to the data to be transferred. Note that if the page in question is in high memory, calling this function is a bug. By default, the block subsystem does not pass high-memory buffers to your driver, but if you have changed that setting with `blk_queue_bounce_limit`, you probably should not be using `bio_data`

The request structure

```
sector_t hard_sector;  
unsigned long hard_nr_sectors;  
unsigned int hard_cur_sectors;
```

These fields are for use only within the block subsystem; drivers should not make use of them

hard_sector is the first sector that has not been transferred. **hard_nr_sectors** is the total number of sectors yet to transfer. **hard_cur_sectors** is the number of sectors remaining in the current **bio**

```
struct bio *bio;
```

The linked list of **bio** structures for this request. Use **rq_for_each_bio** to traverse the list

```
char *buffer;
```

The simple driver example earlier use this field to find the buffer for the transfer. It equals to the result of calling **bio_data** on the current **bio**



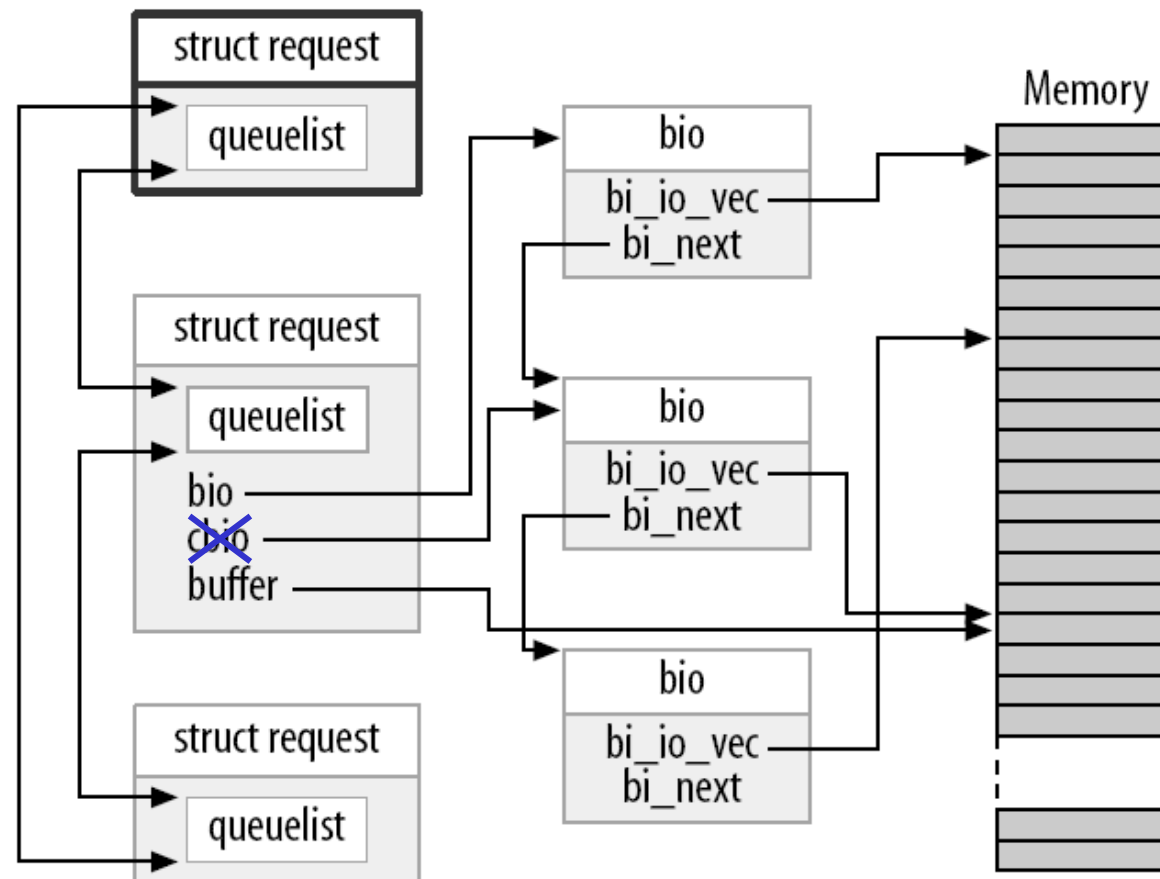
The request structure (cont.)

```
unsigned short nr_phys_segments;
```

Number of distinct segments after adjacent pages have been merged

```
struct list_head queuelist;
```

The linked list structure that links the request into the request queue. if the request is removed from the queue with `blkdev_dequeue_request`, you may use this list head for other purpose





Barrier requests

- Block layer reorders requests before submitting them to the device drivers to improve I/O performance
- But some applications require that certain I/O operations complete before the others
 - Journaling filesystems, relational databases
- The solution is barrier request. if a request is marked with `REQ_HARDBARRIER` flag, it must be written to the drive before any following request is initiated



Barrier request control functions

```
void blk_queue_ordered(request_queue_t *queue, int flag);
```

Inform the block layer that your driver implements barrier requests. In case a power failure occurs when the critical data is still sitting in the drive's cache, your driver must take steps to force the drive to actually write the data to the media

```
int blk_barrier_rq(struct request *req);
```

If this macro returns a nonzero value, the request is a barrier request



Nonretryable requests

- If the macro returns a nonzero value on a failed request, your driver should simply abort the request instead of retrying it

```
int blk_noretry_request(struct request *req);
```



Request completion functions

```
int end_that_request_first(struct request *req, int success, int count);
```

Tell the block code that your driver has completed transferring some or all of the sectors in an I/O request. **count** is the number of sectors transferred starting from where you last left off. If the I/O was successful, pass **success** as 1. The return value indicates if all sectors in this request have been transferred or not

```
void end_that_request_last(struct request *req);
```

wakeup whoever is waiting for the completion of the request and recycles the **request** structure

end_request function

This function is called in *sbull's* request function

```
void end_request(struct request *req, int uptodate)
{
    if (!end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
        add_disk_randomness(req->rq_disk);
        blkdev_dequeue_request(req);
        end_that_request_last(req);
    }
}
```

When all sectors in the request have been transferred, we dequeue the request from request queue and recycle it

contribute entropy to the system's random number pool. It should be called only if the disk's I/O completion time is truly random

Work directly with the bio – Replace `sbul_request` function

```
static void sbull_full_request(request_queue_t *q)
{
    struct request *req;
    int sectors_xferred;
    struct sbull_dev *dev = q->queuedata;

    while ((req = elv_next_request(q)) != NULL) {
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        sectors_xferred = sbull_xfer_request(dev, req);
        if (! end_that_request_first(req, 1, sectors_xferred)) {
            blkdev_dequeue_request(req);
            end_that_request_last(req);
        }
    }
}
```



Work directly with the `bio` (cont.)

```
static int sbull_xfer_request(struct sbull_dev *dev, struct request *req)
{
    struct bio *bio;
    int nsect = 0;

    rq_for_each_bio(bio, req) {
        sbull_xfer_bio(dev, bio);
        nsect += bio->bi_size/KERNEL_SECTOR_SIZE;
    }
    return nsect;
}
```

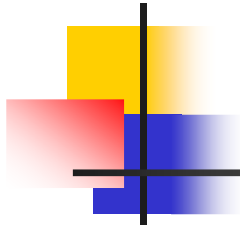


Work directly with the `bio` (cont.)

```
static int sbull_xfer_bio(struct sbull_dev *dev, struct bio *bio)
{
    int i;
    struct bio_vec *bvec;
    sector_t sector = bio->bi_sector;

    /* Do each segment independently. */
    bio_for_each_segment(bvec, bio, i) {
        char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
        sbull_transfer(dev, sector, bio_cur_sectors(bio),
                      buffer, bio_data_dir(bio) == WRITE);
        sector += bio_cur_sectors(bio);
        __bio_kunmap_atomic(bio, KM_USER0);
    }
    return 0; /* Always "succeed" */
}
```

Prepare a scatterlist for DMA transfer



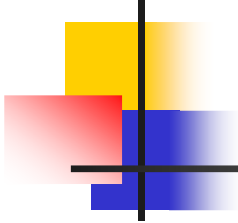
```
int blk_rq_map_sg(request_queue_t *q, struct request *rq,
                  struct scatterlist *sg);
```

Map a request to scatterlist, return number of **sg** entries setup. The returned scatterlist can then be passed to **dma_map_sg**. Caller must make sure **sg** can hold **rq->nr_phys_segments** entries. Segments that are adjacent in memory will be coalesced prior to insertion into the scatterlist. If you do not want to coalesce adjacent segments, clear the bit **QUEUE_FLAG_CLUSTER** in **q->queue_flags**



The problem of queueing requests

- The purpose having request queue
 - Optimizing the order of requests
 - Stalling requests to allow an anticipated request to arrive
- Some devices does not benefit from these optimizations
 - Memory-based device like RAM disks, flash drives
 - Virtual disks created by RAID or LVM



Overriding the default make request function

- Every request queue keeps a function pointer to its make request function, which is invoked when the kernel submit a **bio** to the request queue
- Override the default make request function **__make_request** to avoid reordering and stalling of requests



Designing your make request function

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

The prototype of the make request function. In this function, we can put the **bio** into a request in the request queue, transfer the **bio** directly by walking through the **bio_vec**, or redirect it to another device. **Returns a nonzero value when you want to redirect the **bio** to other device.** It will cause the **bio** to be submitted again. So a “stacking” driver can modify the **bi_dev** to point to a difference device, change the starting sector value, and return.

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

Signal completion directly to the creator of the **bio**. **bytes** is the number of bytes you have transferred so far. It can be less than the number of bytes represented by the **bio** as a whole. If an error is encountered and the request cannot be completed, you can signal an error by providing a nonzero value like **-EIO** for **error** parameter

Sbull without queuing requests

```
static int sbull_make_request(request_queue_t *q, struct bio *bio)
{
    struct sbull_dev *dev = q->queuedata;
    int status;

    status = sbull_xfer_bio(dev, bio);
    bio_endio(bio, bio->bi_size, status);
    return 0;
}
```

Never call **bio_endio** from a regular request function; that job is handled by **end_that_request_first** instead.

Sbull without queuing requests (cont.)

This differs from `blk_init_queue` in that it does not actually set up the queue to hold requests

```
dev->queue = blk_alloc_queue(GFP_KERNEL);
if (dev->queue == NULL)
    goto out_vfree;
blk_queue_make_request(dev->queue, sbull_make_request);
```

Change the make request function of a request queue



Reference

- For detailed information, refer to
 - Linux Device Drivers, 3rd edition, Chapter 16, Block Drivers