Lecture on Storage Systems

Storage Systems and OS Kernels

André Brinkmann

Agenda

How can we represent block devices in the kernel and process requests?

- Representation of storage systems as block devices
  - Data structures for block devices
  - Generic procedures on block devices
- Request Processing inside block device drivers

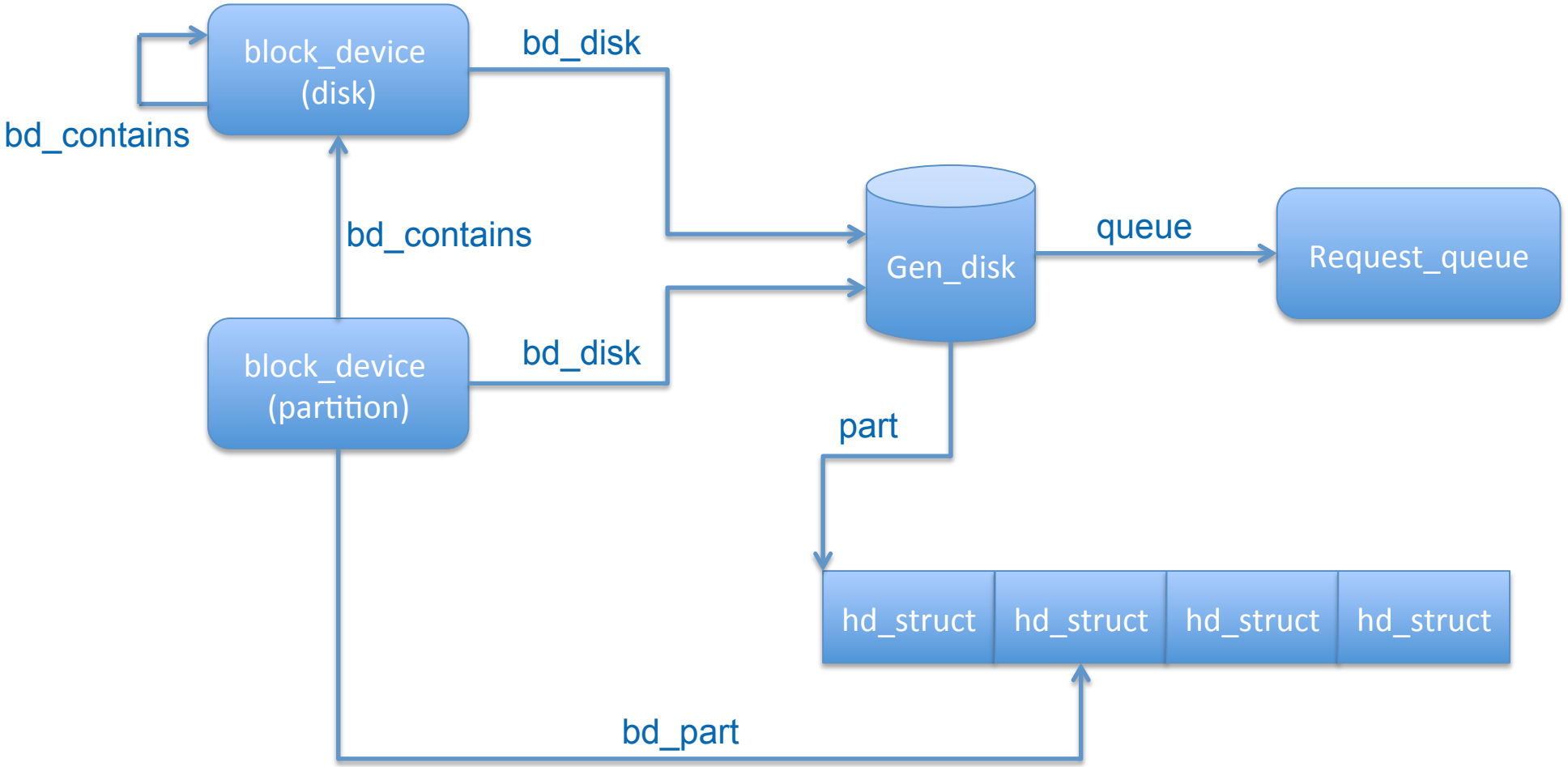
Properties of Storage Systems

- Linux kernel has to be able to work on storage devices in a uniform fashion
  - Many operations do not depend on the properties of the underlying storage system
  - It is not necessary to implement every procedure for every devices
- Requirements to abstract from system specific properties
  - Set of data structures, which include properties of the storage devices
  - Generic functions on these data structures
  - Interfaces to the kernel (e.g., virtual file system)
  - Functions, which cover specific properties of the storage system

Block Devices and Drivers

- Every storage system is represented in the kernel as a block device
- All operations on the block device are handled by the corresponding block device driver
- Block device driver is registered in the kernel by the function
  `register_blkdev (unsigned int major, const char * name)`
  - Major 0 enables operating system to choose the major number itself
  - No further parameters besides the major number and the name
  - `register_blkdev` became optional in Linux 2.6 and only allocates the major number and an entry in /proc/dev
- Individual disk is represented by `struct block_device` and `struct gendisk`

Data Structures



Block Devices: /linux/fs.h

```
struct block_device {
        dev_t                   bd_dev;  /* not a kdev_t - it's a search key */
        struct inode *          bd_inode;       /* will die */
        int                     bd_openers;
        struct semaphore        bd_sem; /* open/close mutex */
        struct semaphore        bd_mount_sem;   /* mount mutex */
        struct list_head        bd_inodes;
        void *                  bd_holder;
        int                     bd_holders;
        struct block_device *   bd_contains;
        unsigned                bd_block_size;
        struct hd_struct *      bd_part;
        /* number of times partitions within this device have been opened. */
        unsigned                bd_part_count;
        int                     bd_invalidated;
        struct gendisk *        bd_disk;
        struct list_head        bd_list;
        struct backing_dev_info *bd_inode_backing_dev_info;
        /*
         * Private data.  You must have bd_claim'ed the block_device
         * to use this.  NOTE:  bd_claim allows an owner to claim
         * the same device multiple times, the owner must take special
         * care to not mess up bd_private for that case.
         */
        unsigned long           bd_private;
};
```

Data structure to describe block devices

- Hash Key for the global management of block devices (unsigned short); encodes major / minor number
- Representation of the devices as an Inode
- bd_contains points to device, which contains this partitions (if it is a partition ...)
- Information about partition (start sector, size, ...)
- Pointer to gendisk structure

Generated at first access through

`bdev_alloc_inode()`

Generic Disks: /linux/genhd.h

```
struct gendisk {
        int major;                      /* major number of driver */
        int first_minor;
        int minors;                     /* maximum number of minors, =1 for
                                         * disks that can't be partitioned. */
        char disk_name[32];             /* name of major driver */
        struct hd_struct **part;        /* [indexed by minor] */
        int part_uevent_supress;
        struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;
        sector_t capacity;

        int flags;
        char devfs_name[64];            /* devfs crap */
        int number;                     /* more of the same */
        struct device *driverfs_dev;
        struct kobject kobj;

        struct timer_rand_state *random;
        int policy;

        atomic_t sync_io;               /* RAID */
        unsigned long stamp;
        int in_flight;
#ifdef  CONFIG_SMP
        struct disk_stats *dkstats;
#else
        struct disk_stats dkstats;
#endif
};
```

Removed in 2.6.18

Data structure to describe devices and partitions

- Includes major number / minor interval
- name of the device
- Drive operations on the device (open, release, …)
- Pointer to request queue
- Capacity in 512 byte sectors

Block Device Operations: /linux/fs.h

```
struct block_device_operations {
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned, unsigned long);
        int (*direct_access) (struct block_device *, sector_t, unsigned long *);
        int (*media_changed) (struct gendisk *);
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        struct module *owner;
};
```

IOCTL without BKL

32-64 Bit conversion

`struct block_device_operations` stores pointer to driver functions

- open(): Increments usage counter
- release(): Decrements usage counter
- ioctl(): IO-Control – Interface between driver and user space, locked by Big Kernel Lock
- direct_access(): Enables reference to data on storage system
- media_changed() – revalidate_disk(): supports removable media
- getgeo(): Returns geometry of the device

Request Queue: /linux/blkdev.h

```
struct request_queue
{
        /* Together with queue_head for cacheline sharing */
        struct list_head        queue_head;
        struct request          *last_merge;
        elevator_t              *elevator;

        /* the queue request freelist, one for reads and one for writes */
        struct request_list     rq;

        request_fn_proc         *request_fn;
        merge_request_fn        *back_merge_fn;
        merge_request_fn        *front_merge_fn;
        merge_requests_fn       *merge_requests_fn;
        make_request_fn         *make_request_fn;
        prep_rq_fn              *prep_rq_fn;
        unplug_fn               *unplug_fn;
        merge_bvec_fn           *merge_bvec_fn;
        activity_fn             *activity_fn;
        issue_flush_fn          *issue_flush_fn;
        prepare_flush_fn        *prepare_flush_fn;
        softirq_done_fn         *softirq_done_fn;

        /* Dispatch queue sorting */
        sector_t                end_sector;
        struct request          *boundary_rq;

        ....

        spinlock_t              __queue_lock;
        spinlock_t              *queue_lock;

        /* queue settings */
        unsigned long           nr_requests;    /* Max # of requests */
        unsigned int            max_sectors;
        unsigned int            max_hw_sectors;
        unsigned short          max_phys_segments;
        unsigned short          max_hw_segments;
        unsigned short          hardsect_size;
        unsigned int            max_segment_size;

        unsigned long           seg_boundary_mask;

        struct blk_queue_tag    *queue_tags;
};
```

Request queue stores requests to block device

- Pointer to a list with requests
- Pointer to elevator queue (includes Pointer to scheduler and ist functions)
- Pointer to request function for the device
- Locks to secure access to data structures
- Information about possible requests
  - Maximum number (nr_requests)
  - Maximum size (set by root)
  - Maximum size (set by block device)
  - Sektor size (multiple of 512 bytes)
- Generated by

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

or

```
blk_queue_make_request(…,
    make_request_fn *func);
```

- Second function does not include optimization through request queue)

Request Processing

- Block operations do not contain any point to functions, which allow the processing of requests
- Request processing is performed in `request()`- or `make_request()`-function
- Functions are paramaters during the generation of the request queue
  - `request()` –function works on request queue and uses optimized ordering of the scheduler
  - `make_request()`-function is used for RAM disks or logical volume managers, which do not profit from request re-ordering

- Functions are core component of request processing ➔ Optimization necessary

`request()`-function

- `request()` gets pointer to request queue as parameter
- Call to `request()` holds lock on request queue
  - Operations on request queue are secured against unintended concurrent changes
- Request Queue is accessed using the following helper functions
  - `elv_next_request()`
    - Returns pointer to next request (or NULL).
    - Request is kept in queue and marked as active
  - `blkdev_dequeue_request()`:
    - Removes request from request queue
  - `elv_requeue_request()`:
    - Re-enqueuing of a request
  - `end_request()`:
    - Calls `blkdev_dequeue_request()`

request-structure

```
struct request {
        struct list_head queuelist;
        struct list_head donelist;

        unsigned long flags;            /* see REQ_ bits below */
        sector_t sector;                /* next sector to submit */
        unsigned long nr_sectors;       /* no. of sectors left to submit */
        /* no. of sectors left to submit in the current segment */
        unsigned int current_nr_sectors;

        sector_t hard_sector;           /* next sector to complete */
        unsigned long hard_nr_sectors;  /* no. of sectors left to complete */
        /* no. of sectors left to complete in the current segment */
        unsigned int hard_cur_sectors;

        struct bio *bio;
        struct bio *biotail;

        void *elevator_private;
        void *completion_data;

        unsigned short ioprio;

        int rq_status;  /* should split this into a few status bits */
        struct gendisk *rq_disk;
        int errors;
        unsigned long start_time;

        ...

        int tag;
        char *buffer;

        int ref_count;
        request_queue_t *q;
        struct request_list *rl;

        struct completion *waiting;
        void *special;

        ...

        unsigned int timeout;
        int retries;

        /*
         * completion callback. end_io_data should be folded in with waiting
         */
        rq_end_io_fn *end_io;
        void *end_io_data;
};
```

Request contains

- Flags describing access properties (rw, barrier, file system-access)
- Access location
- Request as bio structure
- Gendisk of the device
- Completion structure
- Callback structure

request() –function I

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

- Function (for a RAM disk) only accepts ordinary file system requests
- Data transfer is implemented in `sbull_transfer()`
- Function always handles current segment of a request

request() –function II

```
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);
}
```

- Safety check, whether data is accessed over device boundaries
- Data transfer itself is a simple memcpy operation

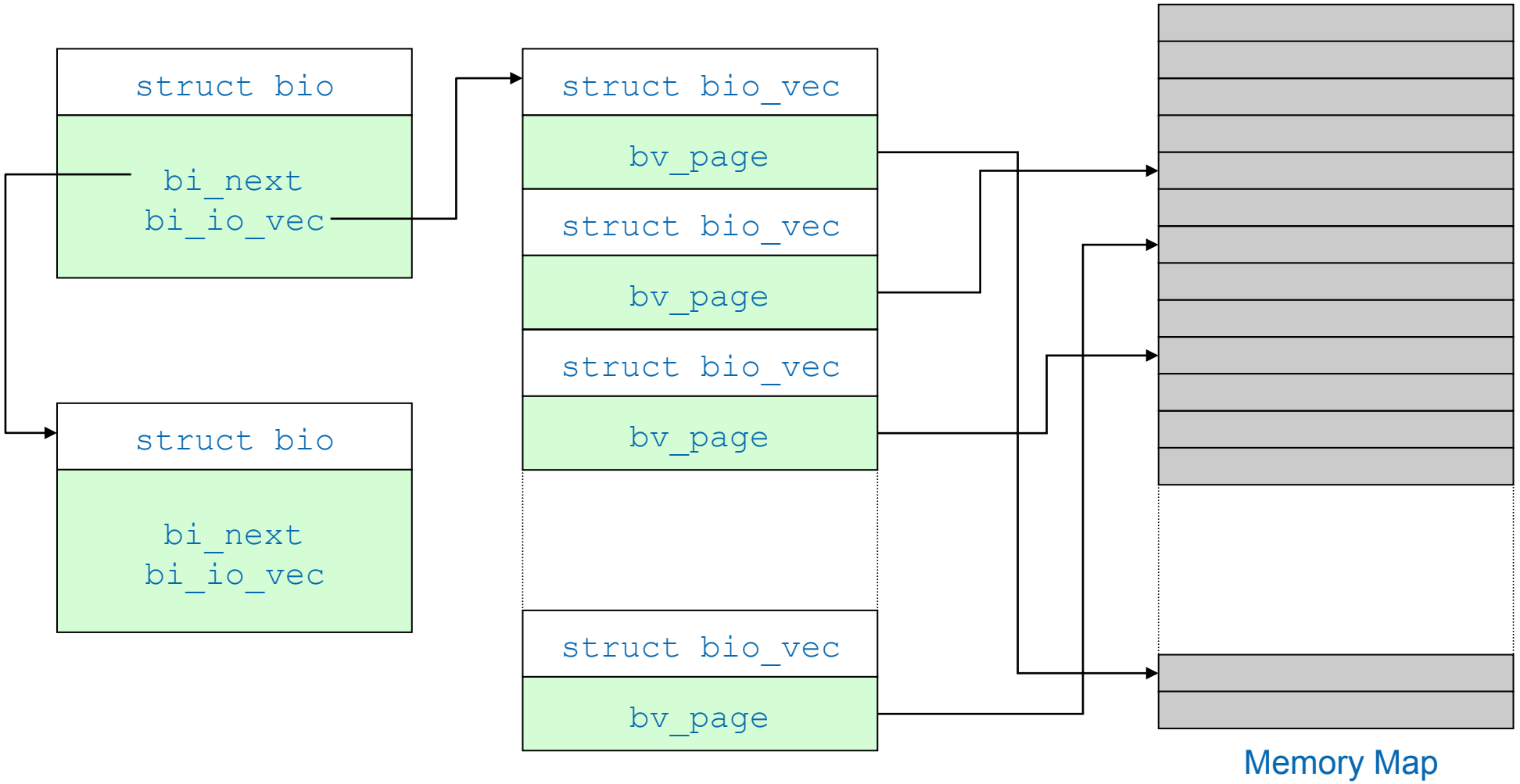
bio-structure

```
struct bio {
        sector_t                bi_sector;
        struct bio              *bi_next;       /* request queue link */
        struct block_device     *bi_bdev;
        unsigned long           bi_flags;       /* status, command, etc */
        unsigned long           bi_rw;          /* bottom bits READ/WRITE,
                                                 * top bits priority
                                                 */

        unsigned short          bi_vcnt;        /* how many bio_vec's */
        unsigned short          bi_idx;         /* current index into bvl_vec */

        /* Number of segments in this BIO after
         * physical address coalescing is performed.
         */
        unsigned short          bi_phys_segments;

        /* Number of segments after physical and DMA remapping
         * hardware coalescing is performed.
         */
        unsigned short          bi_hw_segments;

        unsigned int            bi_size;        /* residual I/O count */

        /*
         * To keep track of the max hw size, we account for the
         * sizes of the first and last virtually mergeable segments
         * in this bio
         */
        unsigned int            bi_hw_front_size;
        unsigned int            bi_hw_back_size;

        unsigned int            bi_max_vecs;    /* max bvl_vecs we can hold */

        struct bio_vec          *bi_io_vec;     /* the actual vec list */

        bio_end_io_t            *bi_end_io;
        atomic_t                bi_cnt;         /* pin count */

        void                    *bi_private;

        bio_destructor_t        *bi_destructor; /* destructor */
};
```

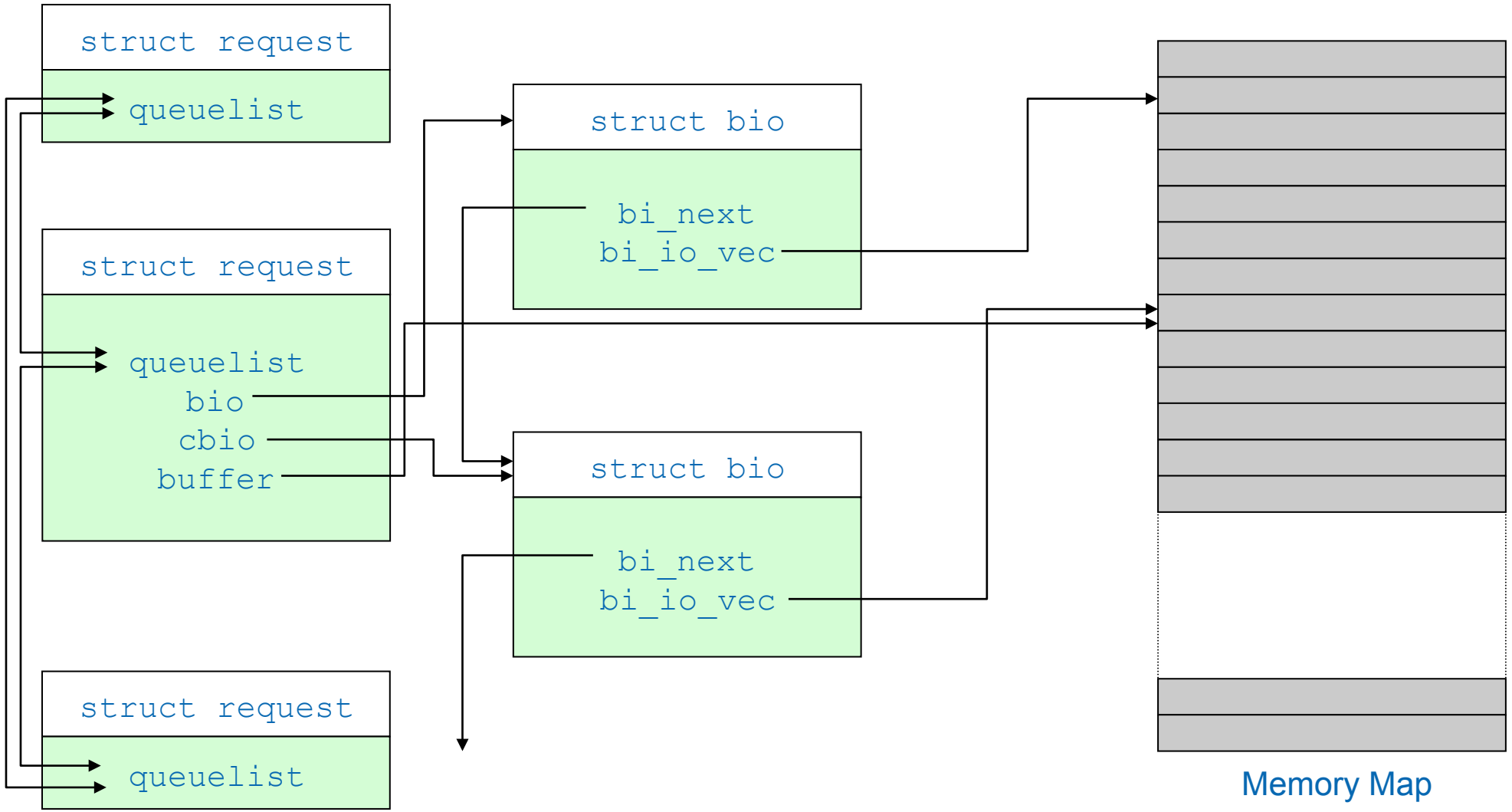
`struct bio` is core function of request processing and contains

- Sector for the next request
- Corresponding block device
- Access Type
- Can include many pages
- Number of requested sectors
- Pointer to array with set of pages for this access
- Pointer to endio()-function
- Private information and destructire

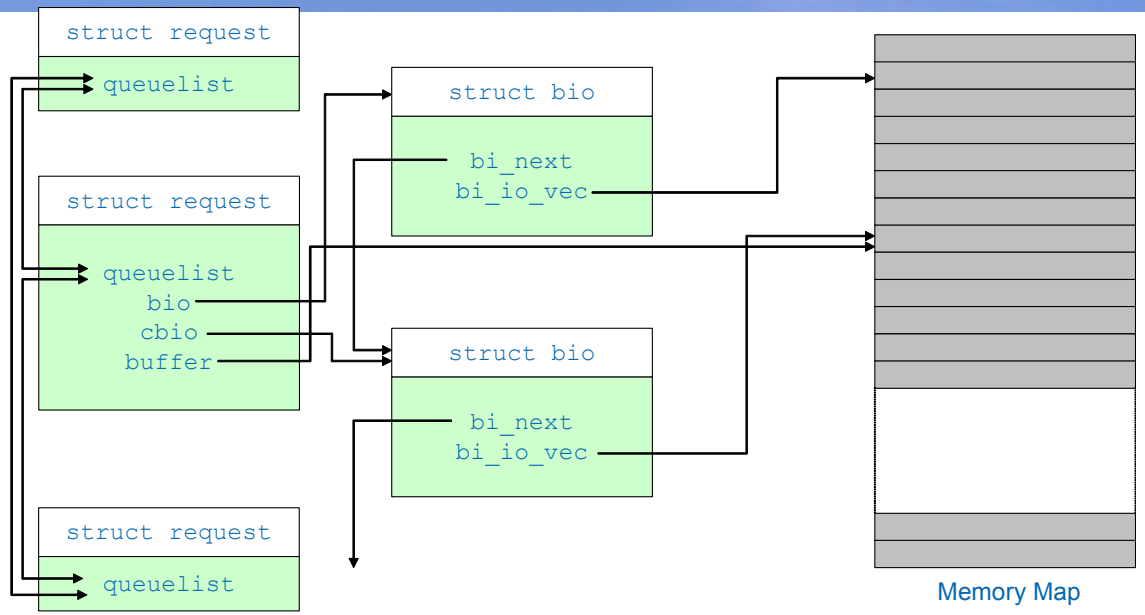
Relationship between bios



request()-function and bios I



request()-function and bios II



- Figure skips biovec-level
- `cbio` points to first bio, which has not been completely handled
- Additional helper functions:
  - `rq_for_each_bio()`: Handles each bio for a request
  - `bio_for_each_segment()`: runs through all pending pieces of a bio
  - `end_that_request_first()`: Signals the transfer of a certain amount of data and returns a value indicating whether the request as a whole was complete
  - `end_that_request_last()`: Calls post-processing

request()-function III

```
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);
        }
    }
}
```

- More complex function handling request processing for a RAM disk
- Can process multiple segments for each access
  - Stores number of segments in sectors_xferred
  - Calls end_that_request_first
  - Dequeuing of the request after all data is processed

request() – function IV

```
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;
}
```

- Functions handles each `bio` seperatley
- Updates transferred data after each round

request() –function V

```
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" */
}
```

- Process one `bio`
  - Each segment is handled by `bio_for_each_segment()`
  - Maps segments into address space
  - Call `sbull_transfer()` for each sector

API Changes

- Linux 2.6 API is far from being static
- `end_that_request_first()` and `end_that_request_last()` have been used for a long time, but usage has been confusing for driver authors
- Both functions have been replaced by single call to `blk_end_request(struct request *rq, int error, int nr_bytes);`
- Many standard housekeeping functions are automatically performed inside `blk_end_request()`
- Additionally, driver can register to callback function

Slide based on Corbet: A new block request completion API, http://lwn.net/Articles/266914

Request Processing without Request Queue

- Request queues minimize head movements of hard disks
  - Virtual devices, SSDs, or RAM disks cannot optimize head movements / do not have head
- Request queue can be initialized using `blk_queue_make_request(…, make_request_fn *func);`
- Request queue still holds information about requests, but requests are submitted to `func` without reordering

```
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;
}
```