Operating System Design and Implementation

Lecture 23: Block device driver

Tsung Tai Yeh

Tuesday: 3:30 – 4:20 pm
Friday: 10:10 – 12:00 pm
Classroom: EC-115

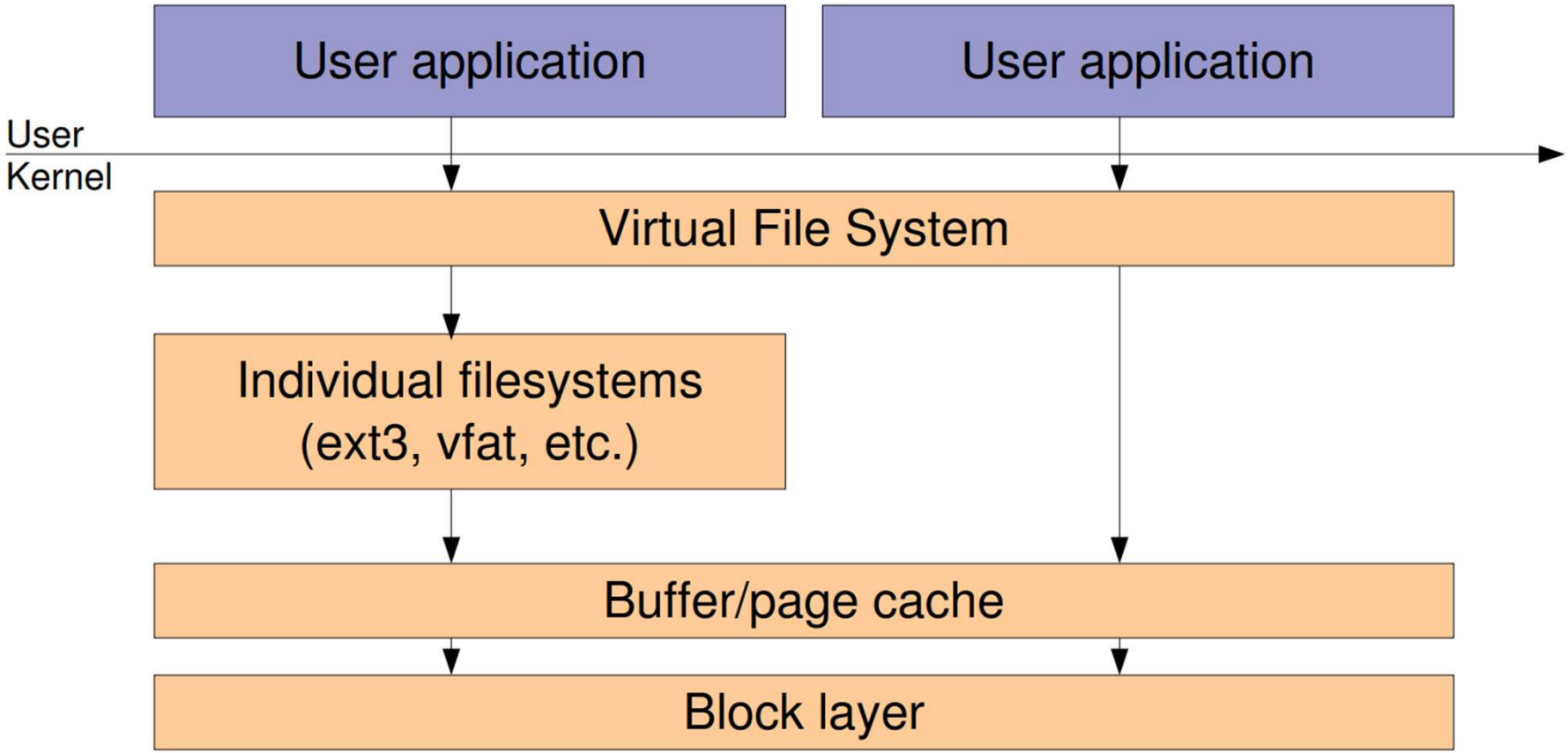
Acknowledgements and Disclaimer

- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
  Onur Mutlu, Computer architecture, ece 447, Carnegie Mellon University
- CSE 506, operating system, 2016,
  https://www.cs.unc.edu/~porter/courses/cse506/s16/slides/sync.pdf

Outline

- Block device abstraction
  - Block layer
    - I/O scheduler
    - Block driver
- The implementation of a block driver

Block device abstraction

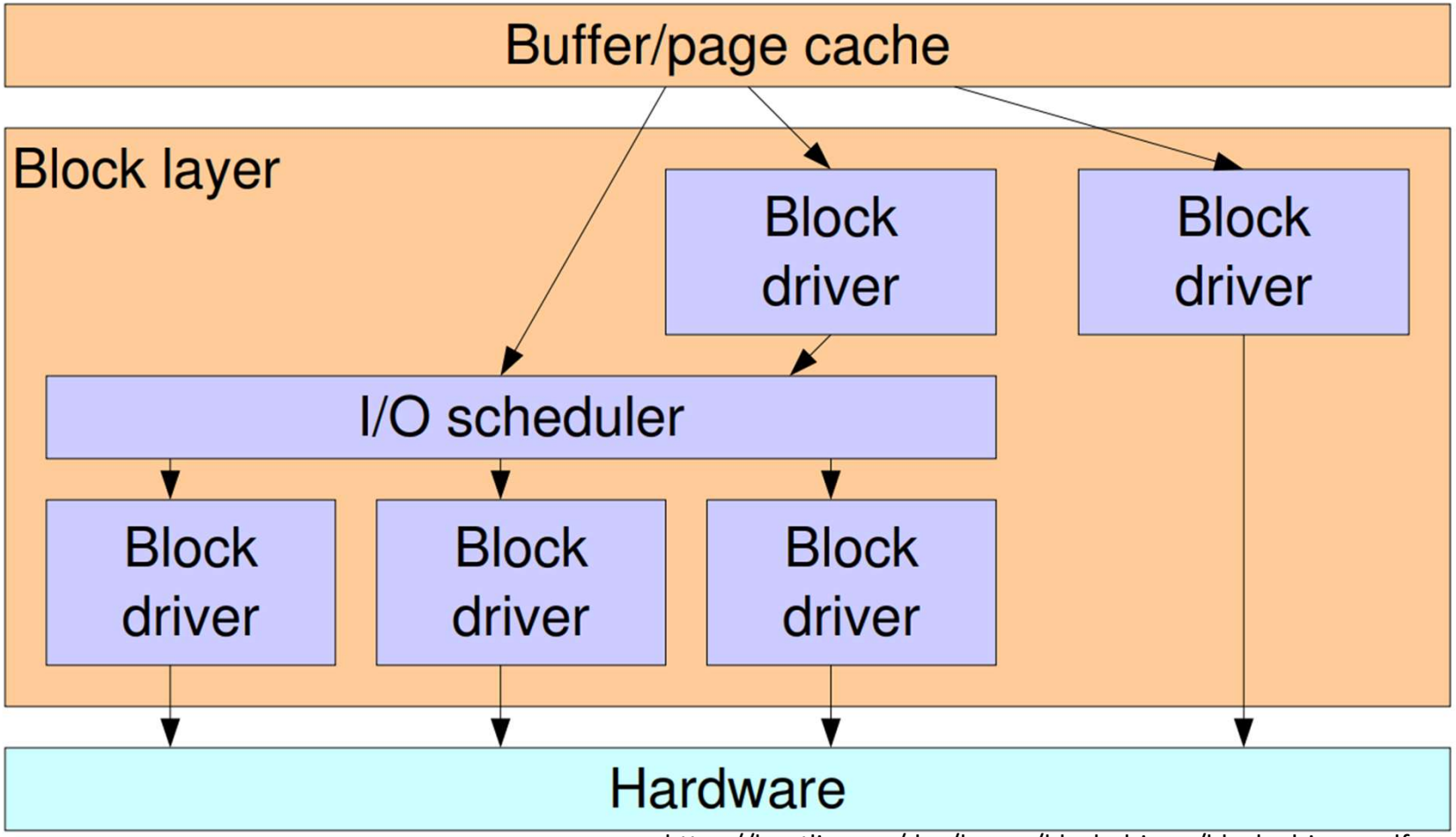


https://bootlin.com/doc/legacy/block-drivers/block_drivers.pdf

Block device abstraction

- An user application can use a block device
  - **Through a file system** -> reading, writing or mapping **files**
  - **Directly** -> reading, writing or mapping **a device file** (e.g. ‘/dev’)
- The **VFS subsystem** in the kernel is the entry point for all accesses
  - **A file system driver** is involved if a normal file is accessed
- **The buffer/page cache** of the kernel stores recently read and written portions of block devices

Inside the block layer



https://bootlin.com/doc/legacy/block-drivers/block_drivers.pdf

Inside the block layer

- **The block layer allows**
  - Block device drivers to receive I/O requests
  - In charge of I/O scheduling
- **I/O scheduling allows**
  - Merge requests so that they are of greater size
  - Re-order requests to optimize disk head movement
- Linux has several I/O schedulers with different policies

I/O schedulers

- **Four I/O scheduler in current kernels**
  - **Noop**
    - For non-disk based block devices
  - **Anticipatory**
    - Tries to anticipate what could be the next accesses
  - **Deadline**
    - Tries to guarantee that an I/O will be served within a deadline
  - **CFQ (Complete Fairness Queuing)**: The default scheduler
    - Tries to guarantee fairness between users of a block device
  - The current scheduler for a device
    - /sys/block/<dev>/queue/scheduler

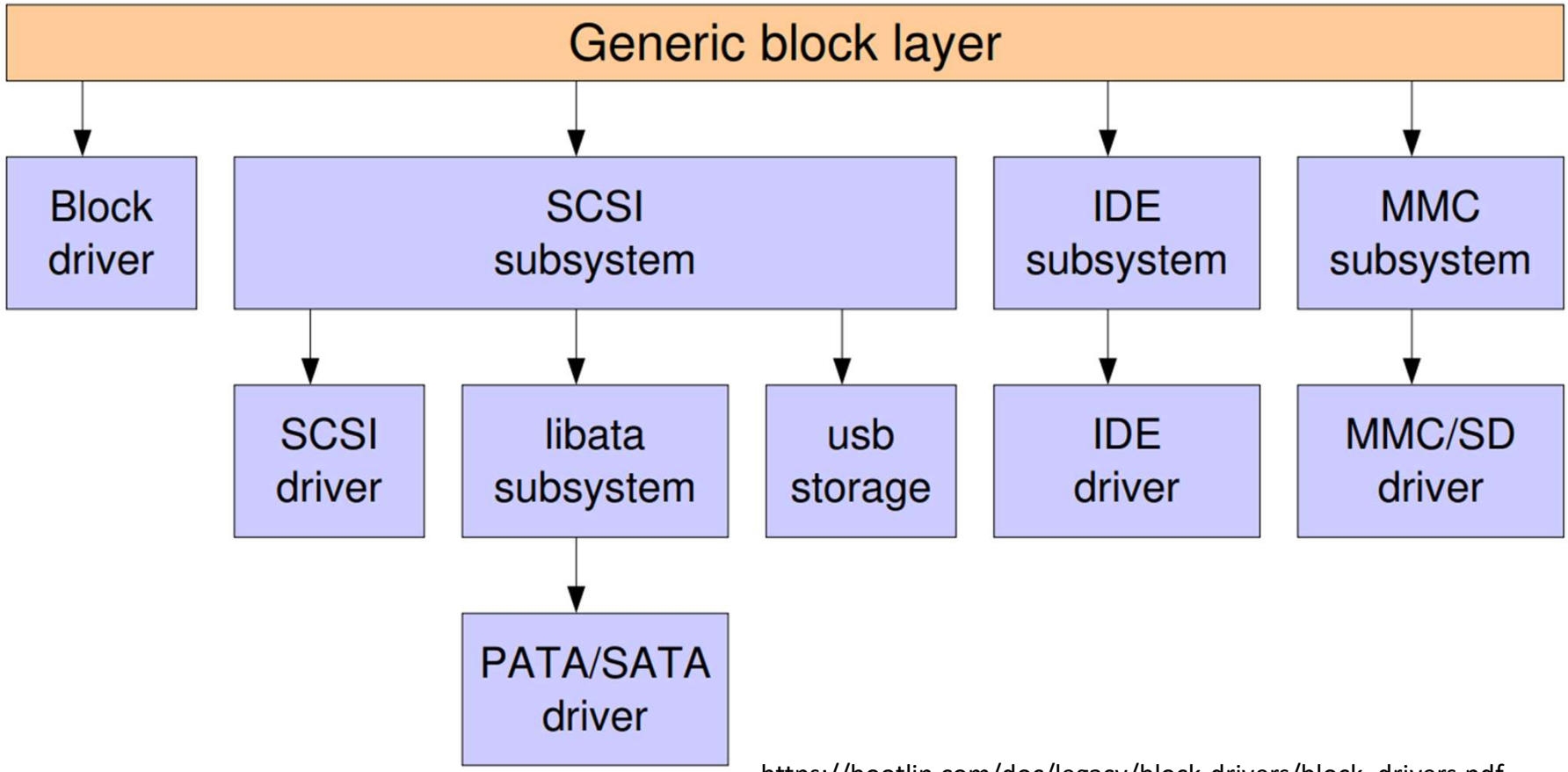
Types of drivers

- **Most of the block device drivers**
  - Implemented below the I/O scheduler to use the I/O scheduling
  - Hard disk drivers, CD-ROM drivers, etc.
- **Some drivers don’t use the I/O scheduler**
  - RAID and volume manager, like md

How to implement a block device driver ?

- **A block device driver**
  - Implement a set of operations
  - These operations must **be registered in the block layer** and receive request from the kernel
  - Sub-systems have been created to factorize common code of drivers for devices
    - SCSI devices
    - SATA devices
    - MMC/SD devices

How to implement a block device driver ?



https://bootlin.com/doc/legacy/block-drivers/block_drivers.pdf

Block device layer

- **The block device layer**
  - Implemented in the ‘block/’ directory of the kernel source tree
  - The I/O scheduler code in *-iosched.c files
- **A few simple block device drivers**
  - See drivers/block/
  - **loop.c**: the loop driver that allows to see a regular file as a block device
  - **brd.c**: a ramdisk driver
  - **nbd.c:** a network-based block device driver

Step 1: Registering the major

- **The first step in the initialization of a block device driver is**
  - The registration of the major number
  - **int register_blkdev(unsigned int major, const char *name);**
  - Major (device number) can be 0 which is dynamically allocated
  - E.g. register_blkdev(sbull_major, “sbull”);
  - Once registered, the driver appears in ‘/proc/devices’
- **Unregistered**
  - **void unregister_blkdev (unsigned int major, const char *name);**

Step 2: kmalloc

- Create the data structure of this block device
  - E.g. devices = kmalloc (ndevices * sizeof (struct sbull_dev), GFP_KERNEL);

Step 3: setup_device ()

- **Setup_device ()**
  - Add a new block device to block layer in the system
  - **Step 3.1: initialize a spin lock**
    - spin_lock_init (&dev->lock);
  - **Step 3.2: allocate a request queue and use spin lock to control the operation in the queue**
    - dev->queue = blk_init_queue (sbull_full_request, &dev->lock);
  - **Step 3.3: allocate and initialize struct gendisk**
    - dev->gd = alloc_disk (SBULL_MINORS);
    - Set_capacity (dev->gd, nsectors * (hardset_size/KERNEL_SECTOR_SIZE));

Initializing a disk

- **struct gendisk**
  - **Represents a single block device**, defined in <linux/genhd.h>
- **Allocate a gendisk structure**
  - struct gendisk ***alloc_disk**(int minors);
  - Minors tells the number of minors to be allocated in the disk
  - 1 for non-partitionable devices
- **Allocate a request queue**
  - struct **request_queue *blk_init_queue** (request_fn_proc, spinlock_t *lock)

Initializing a disk

- **Initialize the gendisk structure**
- **Set the capacity**
  - void **set_capacity** (struct gendisk *disk, sector_t size);
  - **size**: a number of 512-bytes sectors
  - sector_t is 64 bits wide on 64 bits architectures
- **Add the disk to the system**
  - void **add_disk** (struct gendisk *disk);
  - The driver must be fully ready to handle I/O requests before calling add_disk()
  - Afterward, the block device can be accessed by the system

Unregistering a disk

- **Unregister the disk**
  - void del_gendisk (struct gendisk *gp);
- **Free the request queue**
  - void blk_cleanup_queue (struct request_queue *);
- **Drop the reference taken in alloc_disk()**
  - void put_disk (struct gendisk *disk);

struct block_dev_operations

```
static struct block_device_operations sbull_ops = {
        .owner          =       THIS_MODULE,
        .open           =       sbull_open,
        .release        =       sbull_release,
        .media_change   =       sbull_release,
        .revalidate_disk =      sbull_revalidate,
        .ioctl          =       sbull_ioctll
};
```

Block device operations

- **open () and release ()**
  - Called when a device handled by the driver is opened and closed
- **ioctl ()**
  - Manipulates the underlying device parameters of special files
  - E.g. ioctl(sockfd,SIOCGIFADDR,&ifr)
- **direct_access ()**
  - required for XIP support
- **media_changed (), revalidate ()**
  - required for removable media support
- **getgeo()**
  - provides geometry information to userspace

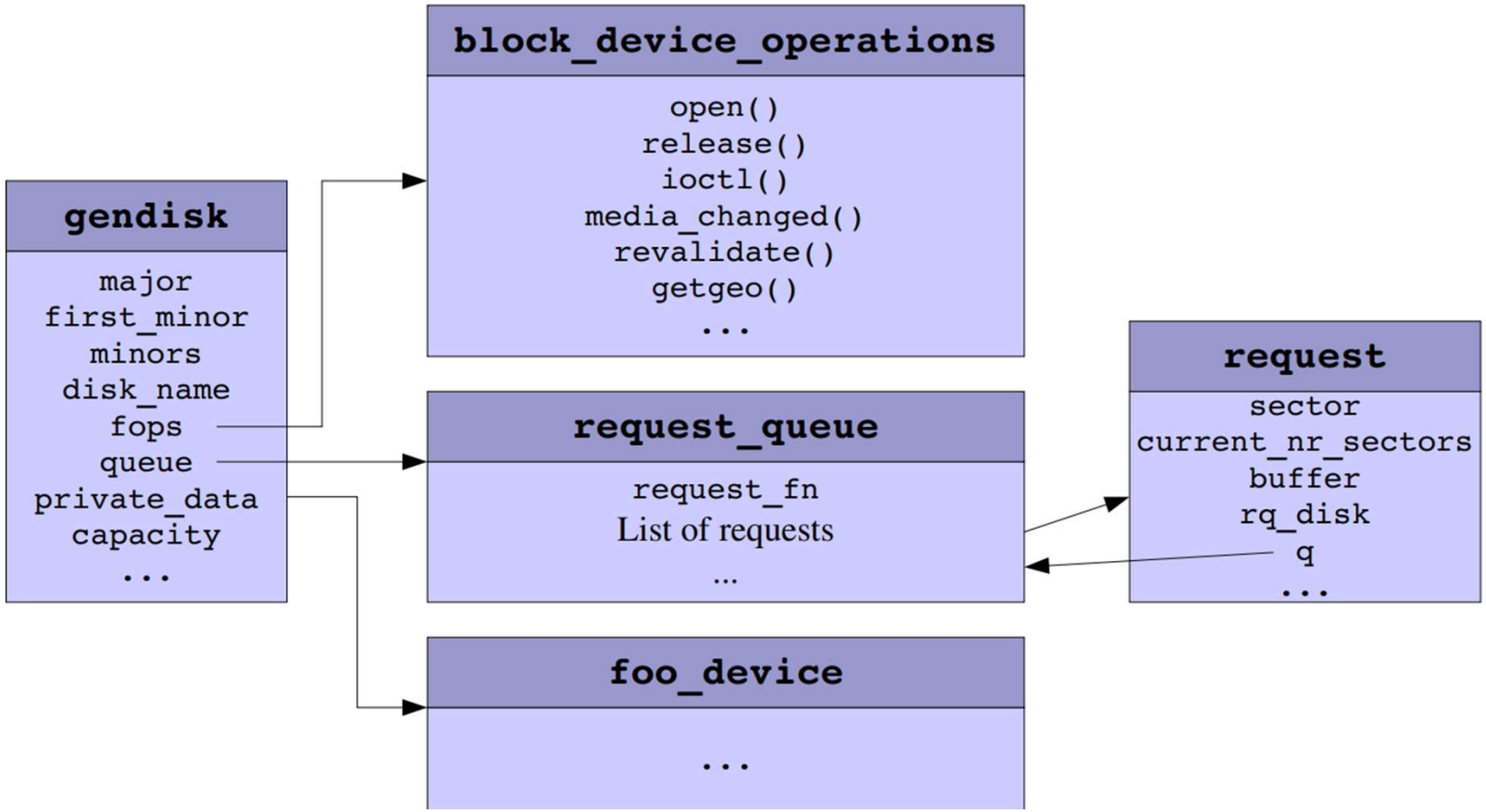
request () operations

- **struct request ()**
  - Make a request to the underlying devices
  - **sector**: the position in the device where the transfer should be made
  - **current_nr_sectors**: the number of sector to transfer
  - **buffer**: the location in memory where the data should be read or written to
  - **rq_data_dir ():** the type of transfer, either READ or WRITE
  - **_blk_end_request () or blk_end_request ()** notify the completion of a request

A simple request() example

```
static void foo_request (struct request_queue *q) {
    struct request *req;
   // elv_next_request: obtain the first non-completed request
    while ((req = elv_next_request(q)) != NULL) {
        if ( ! blk_fs_request (req) ) {
            __blk_end_request (req, 1, req->nr_sectors << 9);
            continue;
        }
        /*Do the transfer here*/
        __blk_end_request (req, 0, req->nr_sectors << 9);
    }
}
```

Data structure of a block device driver

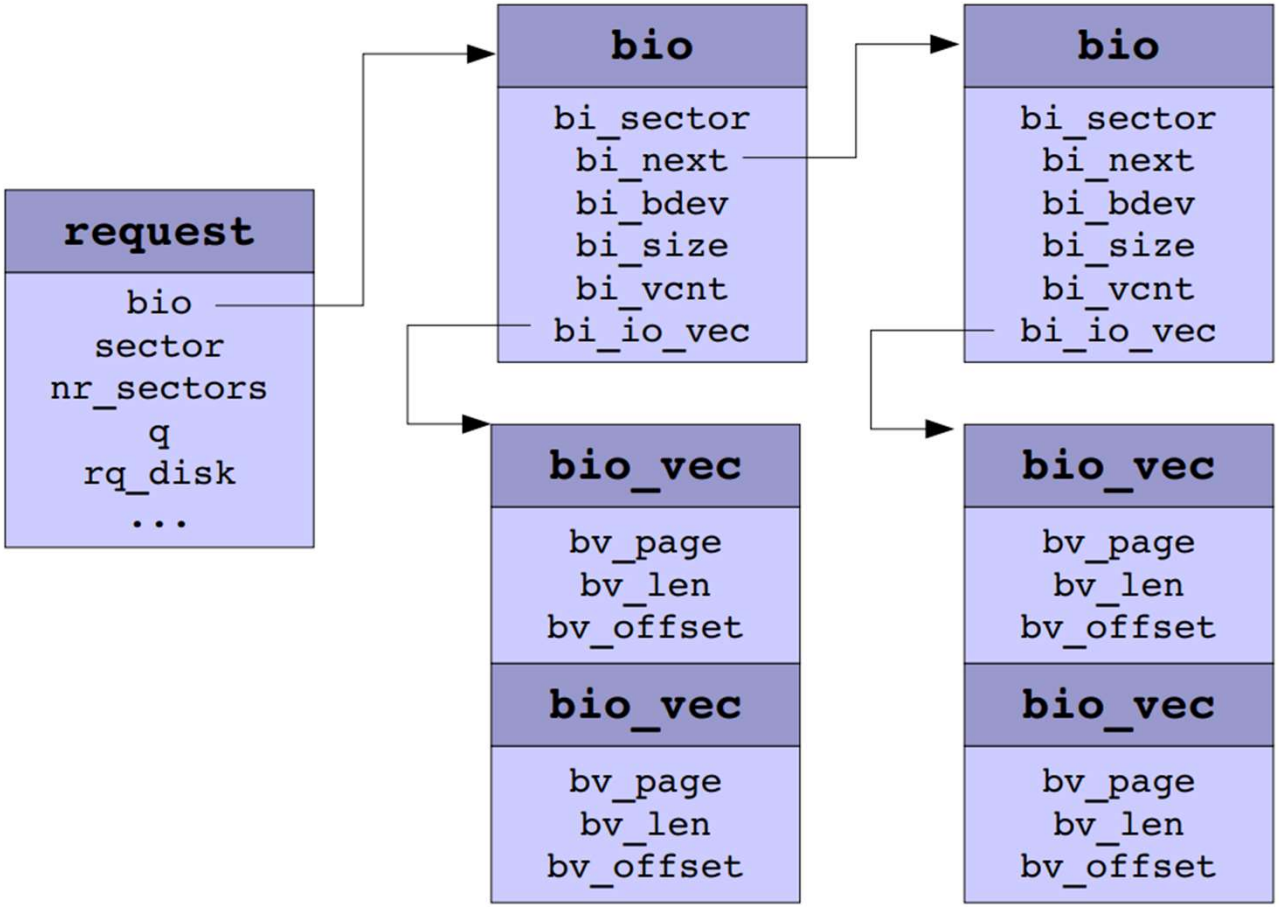


https://bootlin.com/doc/legacy/block-drivers/block_drivers.pdf

Inside a request

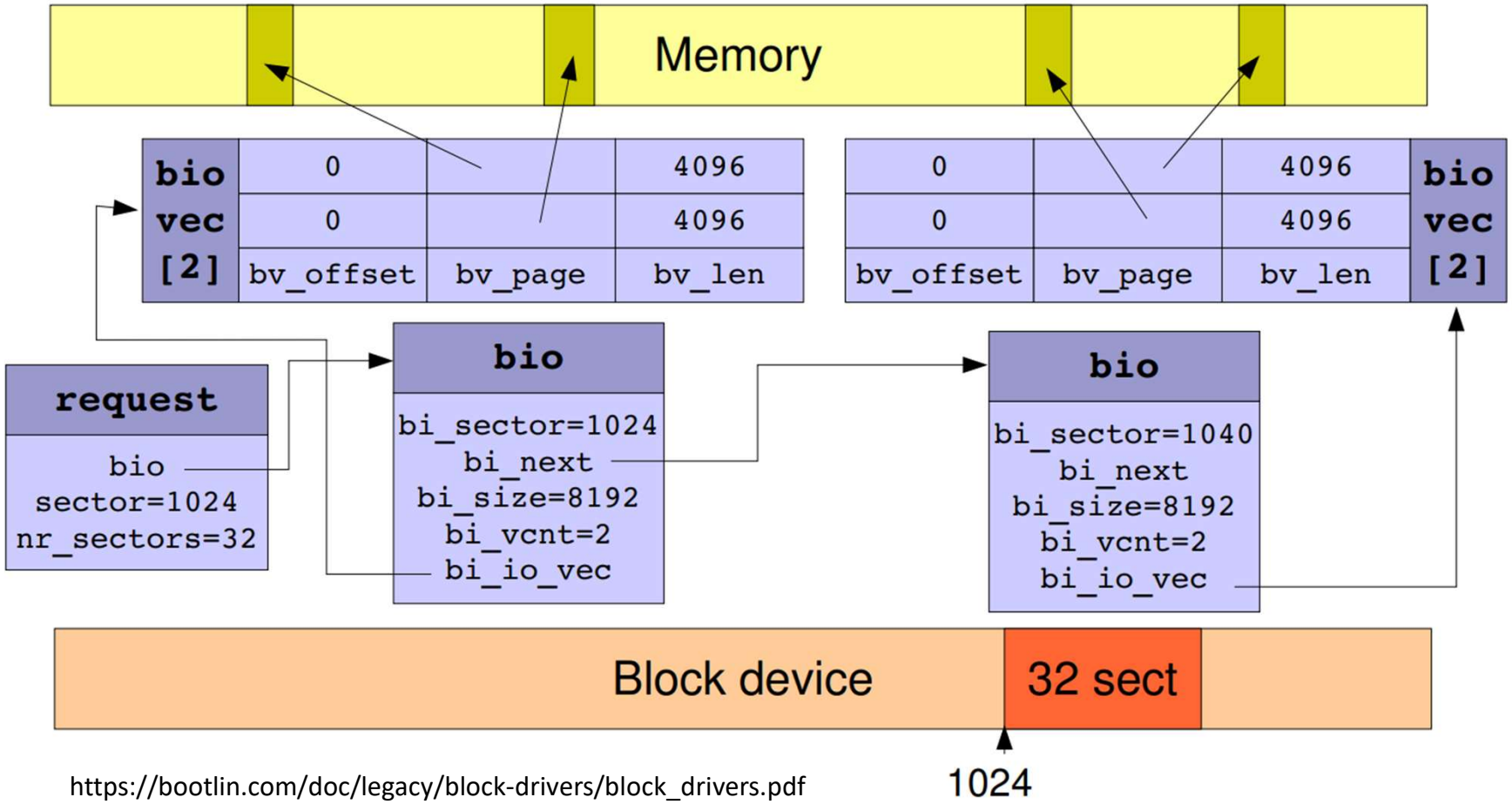
- **A request contains several segments**
  - These segments are contiguous on the block device
  - Not necessarily contiguous in physical memory
- **A *struct request* is in fact a list of *struct bio***
- **A bio**
  - **The descriptor of an I/O request** submitted to the block layer
  - The bio(s) are merged together in a *struct request* by the I/O scheduler
  - Might represent several pages of data (several struct bio_vec)
  - Each of struct bio_vec is a page of memory

Inside a request



https://bootlin.com/doc/legacy/block-drivers/block_drivers.pdf

Request example

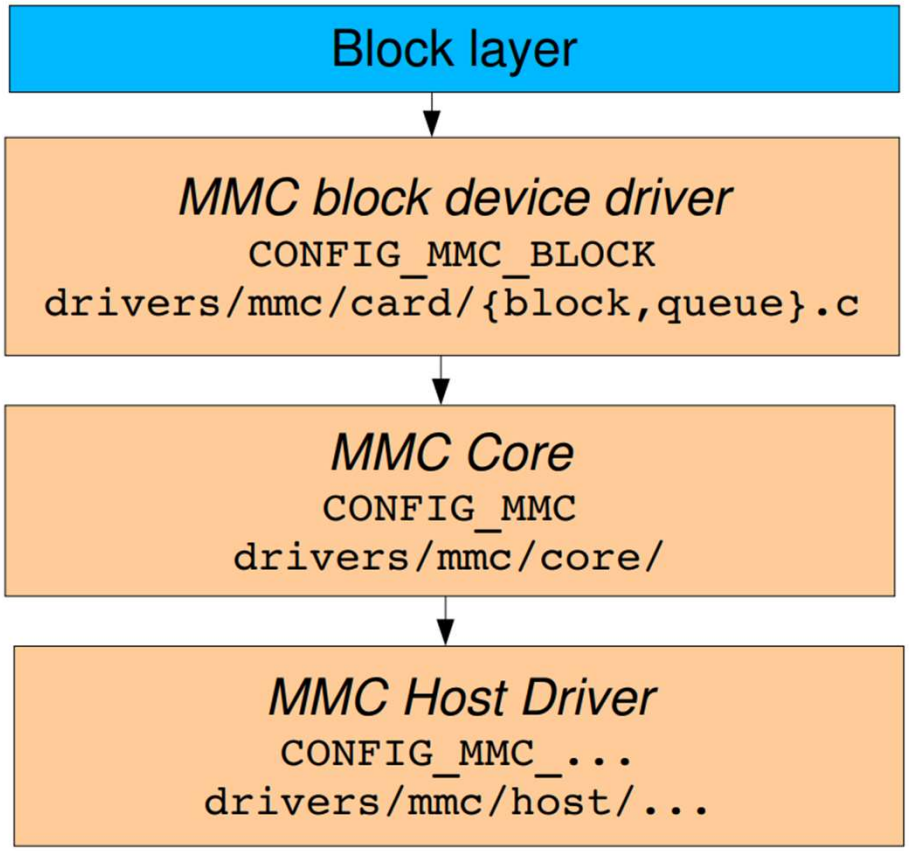


https://bootlin.com/doc/legacy/block-drivers/block_drivers.pdf

Asynchronous operations

- Asynchronous operations
  - Occurs when handling several requests at the same time
  - Dequeue the requests from the queue
  - void blkdev_dequeue_request (struct request *req);
- Put a request back in the queue
  - void elv_requeue_request (struct request_queue *queue, struct request *req);

MMC/SD



https://bootlin.com/doc/legacy/block-drivers/block_drivers.pdf

MMC host driver

- **For each host**
  - **struct mmc_host *mmc_alloc_host** (int extra, struct device *dev)
  - **Initialize struct mmc_host fields**
    - Caps, ops, max_phys_segs, max_hw_segs, max_blk_size, max_blk_count, max_req_size
  - **int mmc_add_host** (struct mmc_host *host)
- **Unregistration**
  - **void mmc_remove_host** (struct mmc_host *host)
  - **void mmc_free_host** (struct mmc_host *host)

MMC host driver

- The mmc_host->ops field points to a mmc_host_ops structure
  - **Handle an I/O request**
    - void (*request) (struct mmc_host *host, struct mmc_request *req);
  - **Set configuration settings**
    - void (*set_ios) (struct mmc_host *host, struct mmc_ios *ios);
  - **Get read-only status**
    - int (*get_ro) (struct mmc_host *host);
  - **Get the card presence status**
    - int (*get_cd) (struct mmc_host *host);

Summary

- Block layer is a middleware
  - Fetches items from the buffer cache
  - Includes block drivers and I/O scheduler
- The implementation of a block device driver
  - Step 1: registers the block device
  - Step 2: Create and allocate data structure for that device
  - Step 3: Setup device: initialize disk, allocate request queue …
- Request () operations and struct bio