Block device drivers

Block devices

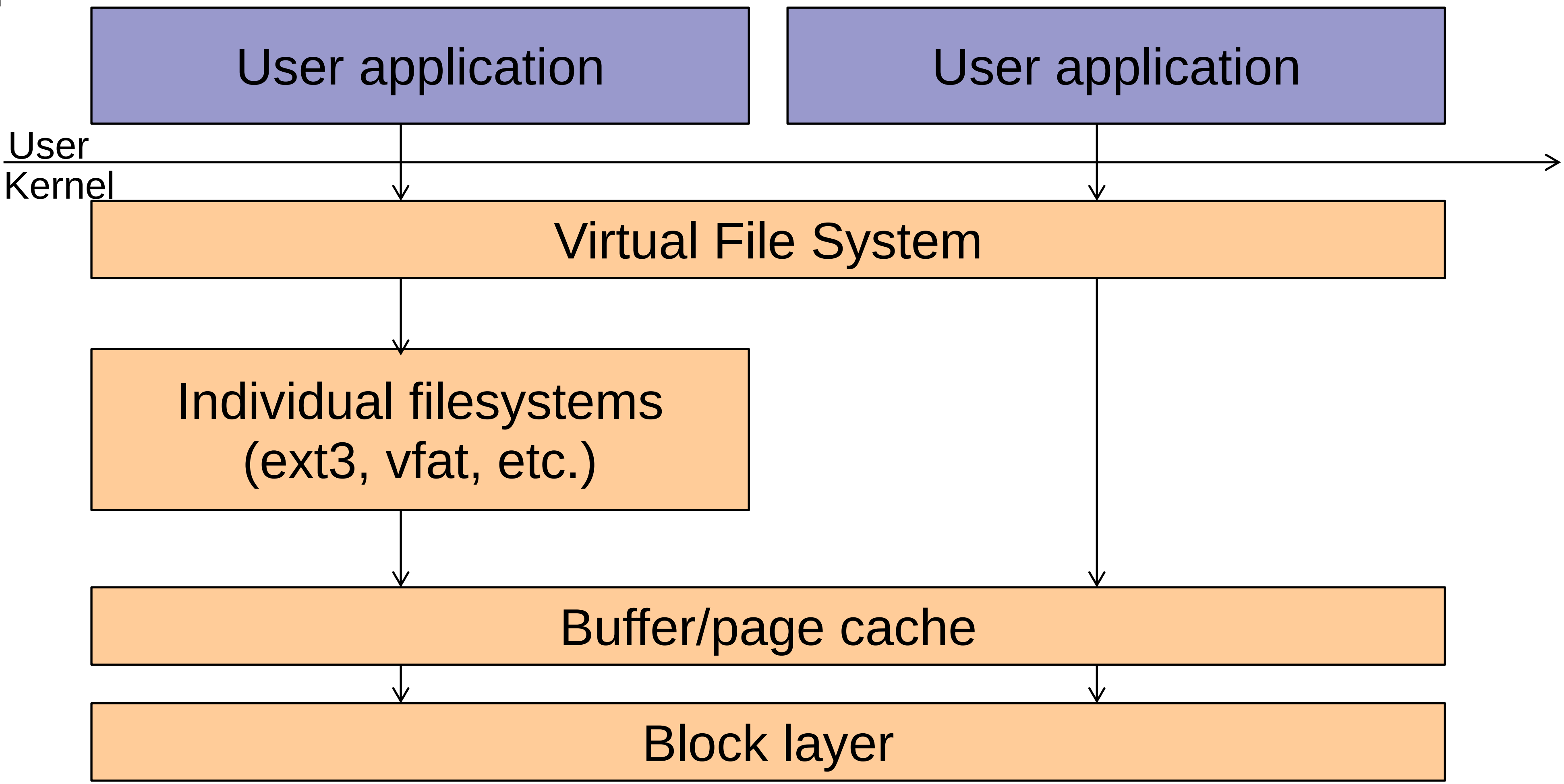
- After character devices and network devices, block devices are another important device type of any system
- Used for the storage of application code and data, and user data, they are often critical to the overall performance of the system
- A dedicated subsystem, the *block layer* is in charge of managing the block devices, together with hardware specific device drivers
- This subsystem has been completely rewritten during the 2.5 development cycle. The API covered in these slides is the one found in 2.6.x kernels.

From userspace

- From userspace, block devices are accessed using device files, usually stored in /dev/
- Created manually or dynamically by udev
- Most of the time, they store filesystems : they are not accessed directly, but rather *mounted,* so that the contents of the filesystem appears in the global hierarchy of files
- Block devices are also visible through the sysfs filesystem, in the /sys/block/ directory

```
$ ls /sys/block/
dm-0  dm-2  loop0  loop2  ram0  ram2  ram4  ram6  sda
dm-1  dm-3  loop1  loop3  ram1  ram3  ram5  ram7
```

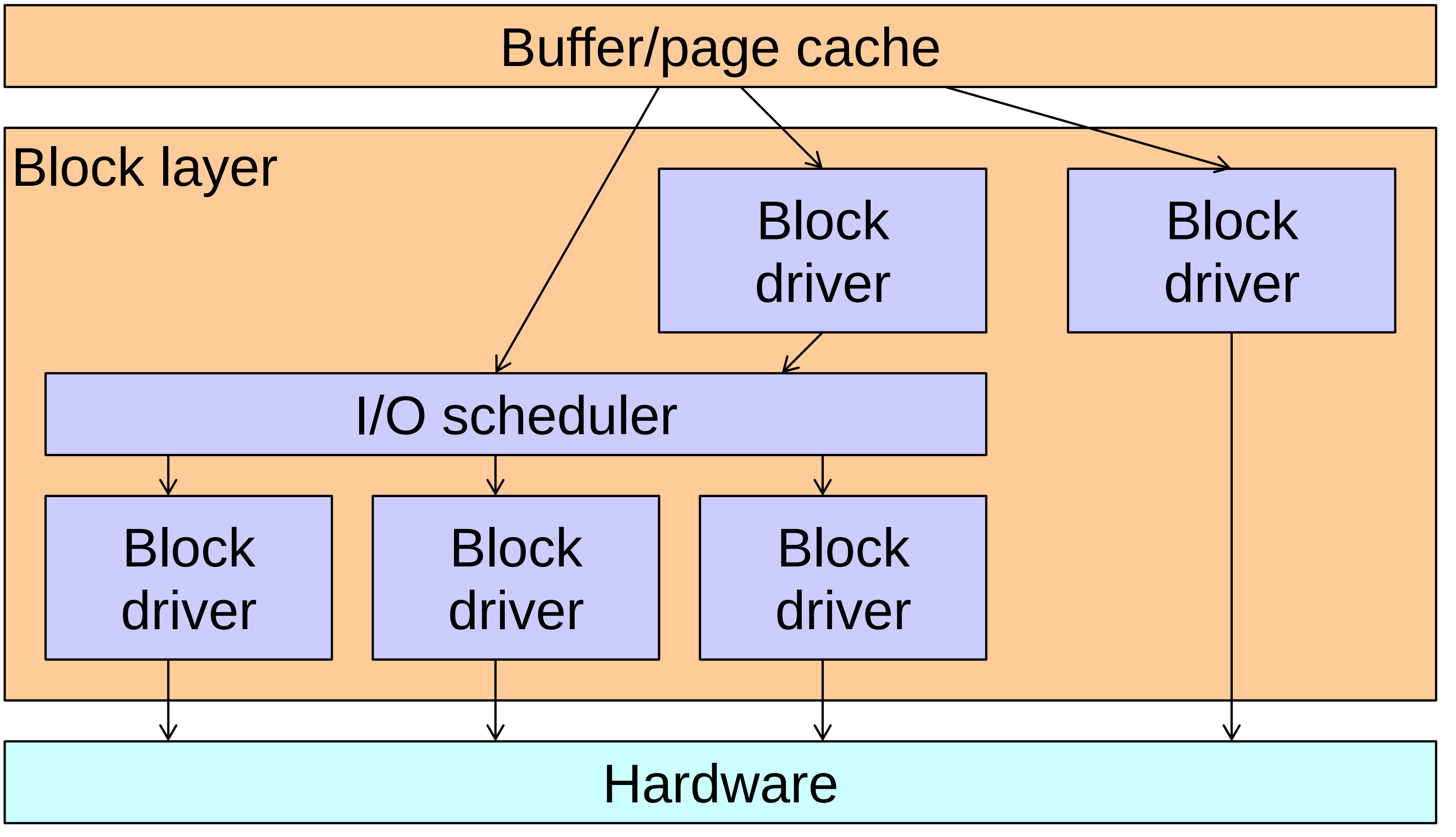
Global architecture



Global architecture (2)

- An user application can use a block device
- Through a filesystem, by reading, writing or mapping files
- Directly, by reading, writing or mapping a device file representing a block device in /dev
- In both cases, the VFS subsystem in the kernel is the entry point for all accesses
- A filesystem driver is involved if a normal file is being accessed
- The buffer/page cache of the kernel stores recently read and written portions of block devices
- It is a critical component for the performance of the system

Inside the block layer



Inside the block layer (2)

- The *block layer* allows block device drivers to receive I/O requests, and is in charge of I/O scheduling
- I/O scheduling allows to
- Merge requests so that they are of greater size
- Re-order requests so that the disk head movement is as optimized as possible
- Several I/O schedulers with different policies are available in Linux.
- A block device driver can handle the requests before or after they go through the I/O scheduler

Two main types of drivers

- Most of the block device drivers are implemented below the I/O scheduler, to take advantage of the I/O scheduling
- Hard disk drivers, CD-ROM drivers, etc.
- For some drivers however, it doesn't make sense to use the IO scheduler
- RAID and volume manager, like md
- The special loop driver
- Memory-based block devices

Available I/O schedulers

- Four I/O schedulers 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, the *Complete Fairness Queuing,* the default scheduler, tries to guarantee fairness between users of a block device
- The current scheduler for a device can be get and set in /sys/block/<dev>/queue/scheduler

Kernel options

- CONFIG_BLOCK
- Allows to selectively enable or disable the block layer. A kernel without the block layer can be useful if using MTD devices, storage over the network, or a filesystem in initramfs
- Only available if CONFIG_EMBEDDED is set
- CONFIG_IOSCHED_NOOP, CONFIG_IOSCHED_AS, CONFIG_IOSCHED_DEADLINE, CONFIG_IOSCHED_CFQ
- Allows to enable or disable different I/O schedulers. They can be compiled as module, loaded and changed dynamically, on a per-device basis.

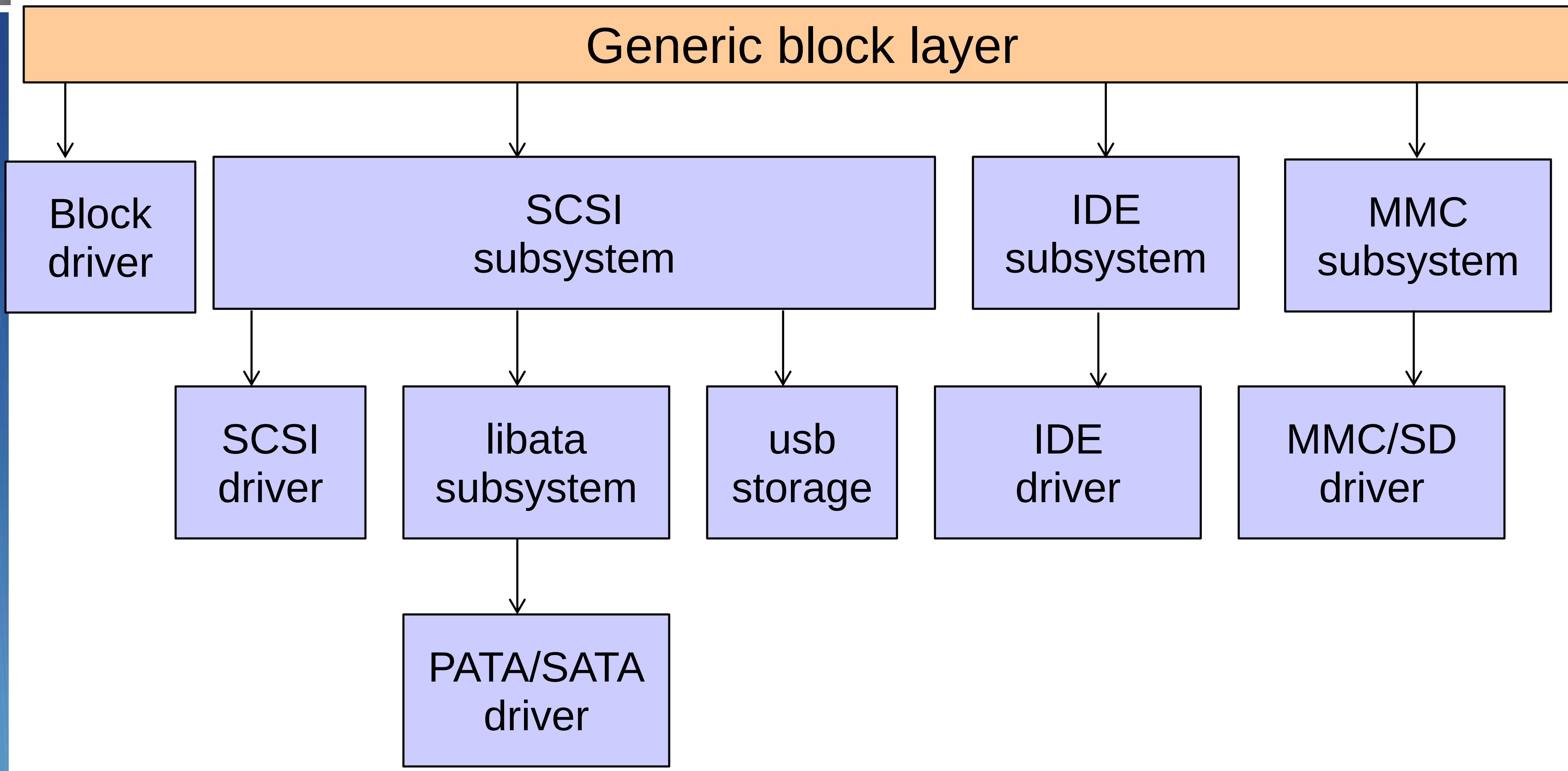
Looking at the code

- The block device layer is implemented in the block/ directory of the kernel source tree
- This directory also contains the I/O scheduler code, in the *-iosched.c files.
- A few simple block device drivers are implemented in drivers/block/, including
- 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

Implementing a block device driver

- A block device driver must implement a set of operations to be registered in the *block layer* and receive requests from the kernel
- A block device driver can directly implement this set of operation. However, as in many areas in the kernel, subsystems have been created to factorize common code of drivers for devices of the same type
- SCSI devices
- PATA/SATA devices
- MMC/SD devices
- etc.

Implementing a block device driver



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 can be 0, in which case it is dynamically allocated
- Once registered, the driver appears in /proc/devices with the other block device drivers
- Of course, at cleanup time, the major must be unregistered
- void unregister_blkdev(unsigned int major, const char *name);
- The prototypes of these functions are in <linux/fs.h>

struct gendisk

- The structure representing a single block device, defined in <linux/genhd.h>
- int major, major of the device driver
- int first_minor, minor of this device. A block device can have several minors when it is partitionned
- int minors, number of minors. 1 for non-partitionable devices
- struct block_device_operations *fops, pointer to the list of block device operations
- struct request_queue *queue, the queue of requests
- sector_t capacity, size of the block device in sectors

Initializing a disk

- **Allocate a gendisk structure**

struct gendisk *alloc_disk(int minors)

minors tells the number of minors to be allocated for this disk. Usually 1, unless your device can be partitionned

- **Allocate a request queue**

struct request_queue *blk_init_queue
    (request_fn_proc *rfn, spinlock_t *lock)

rfn is the request function (covered later). lock is a optional spinlock needed to protect the request queue against concurrent access. If NULL, a default spinlock is used

Initializing a disk (2)

- **Initialize the gendisk structure**

Fields major, first_minor, fops, disk_name and queue should at the minimum be initialized

private_data can be used to store a pointer to some private information for the disk

- **Set the capacity**

void set_capacity(struct gendisk *disk, sector_t size)

The size is a number of 512-bytes sectors. sector_t is 64 bits wide on 64 bits architectures, 32 bits on 32 bits architecture, unless CONFIG_LBD (large block devices) has been selected

Initializing a disk (3)

- **Add the disk to the system**

void add_disk(struct gendisk *disk);

The block device can now be accessed by the system, so the driver must be fully ready to handle I/O requests before calling add_disk(). I/O requests can even take place during the call to add_disk().

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);

block_device_operations

- A set of function pointers
- open() and release(), called when a device handled by the driver is opened and closed
- ioctl() for driver specific operations. unlocked_ioctl() is the non-BKL variant, and compat_ioctl() for 32 bits processes running on a 64 bits kernel
- direct_access() required for XIP support,
- media_changed() and revalidate() required for removable media support
- getgeo(), to provide geometry informations to userspace

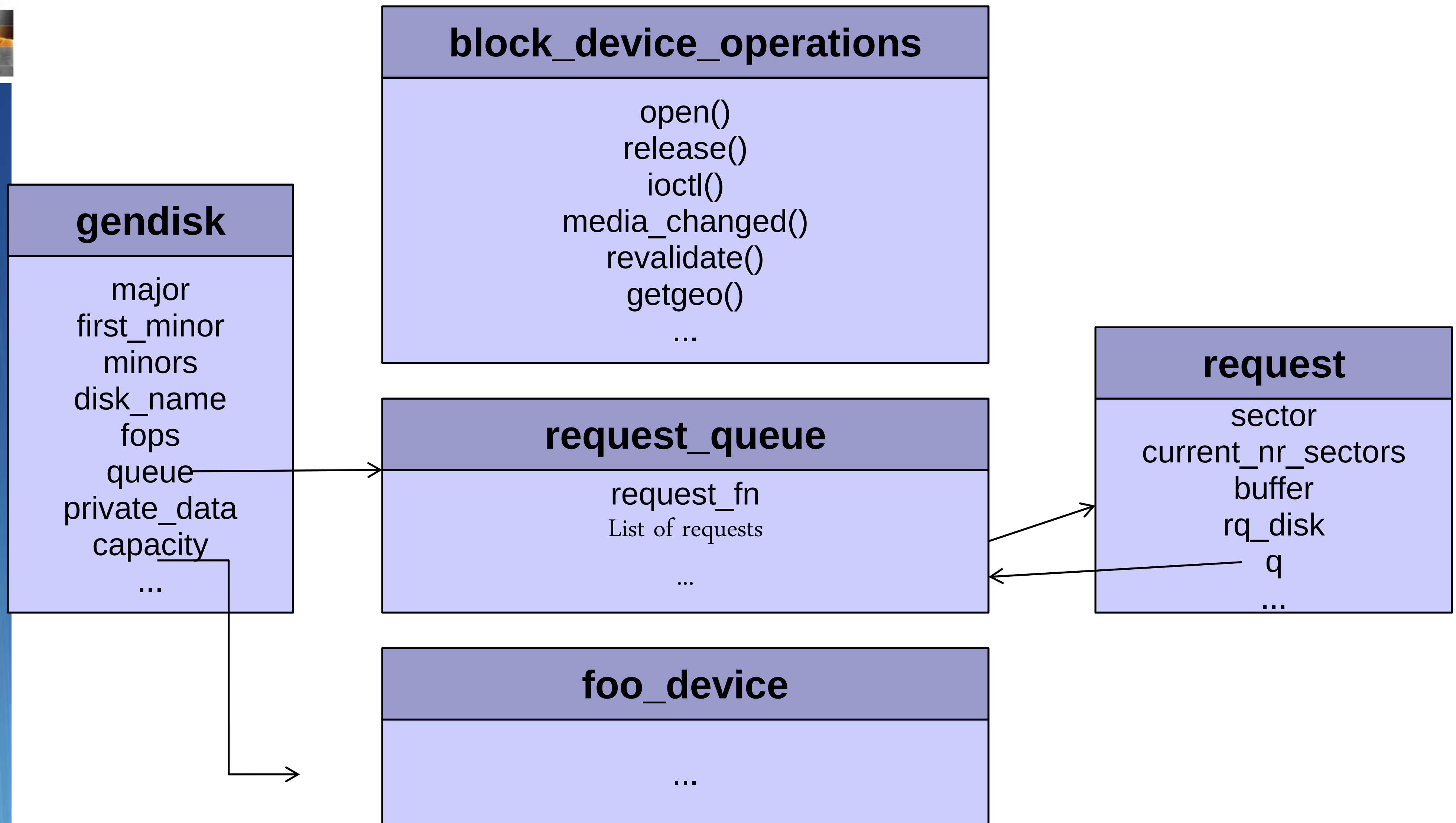
A simple request() function

```
static void foo_request(struct request_queue *q)
{
    struct request *req;
    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);
    }
}
```

A simple request() function (2)

- Information about the transfer are available in the struct request
- sector, the position in the device at which the transfer should be made
- current_nr_sectors, the number of sectors 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() is used to notify the completion of a request. __blk_end_request() must be used when the queue lock is already held

Data structures in a nutshell



Inside a request

- A request is composed of several segments, that are contiguous on the block device, but not necessarily contiguous in physical memory
- A struct request is in fact a list of struct bio
- A bio is the descriptor of an I/O request submitted to the block layer. bios are merged together in a struct request by the I/O scheduler.
- As a bio might represent several pages of data, it is composed of several struct bio_vec, each of them representing a page of memory

Inside a request (2)

request
bio
sector
nr_sectors
q
rq_disk
...

bio
bi_sector
bi_next
bi_bdev
bi_size
bi_vcnt
bi_io_vec

bio
bi_sector
bi_next
bi_bdev
bi_size
bi_vcnt
bi_io_vec

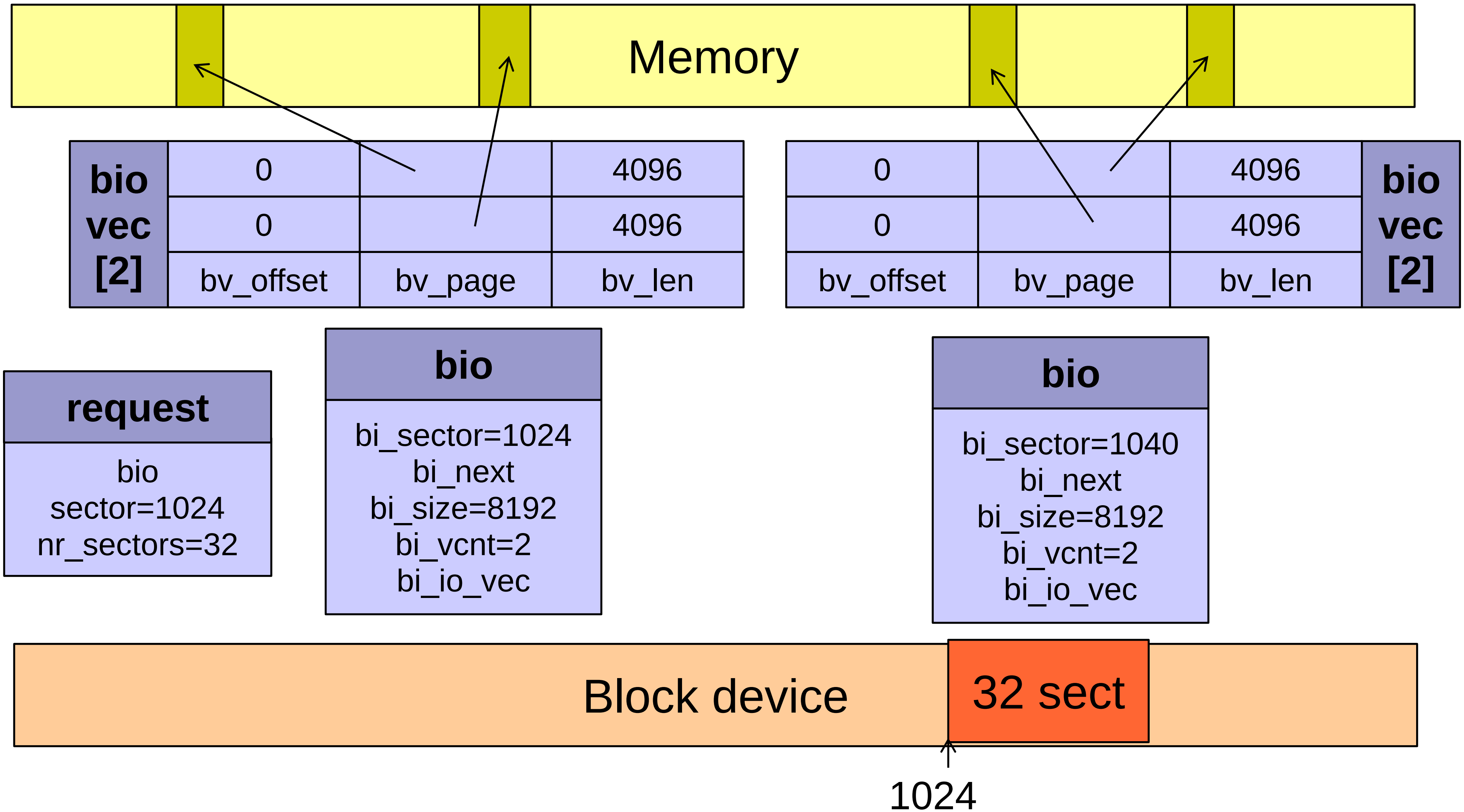
bio_vec
bv_page
bv_len
bv_offset

bio_vec
bv_page
bv_len
bv_offset

bio_vec
bv_page
bv_len
bv_offset

bio_vec
bv_page
bv_len
bv_offset

Request example



Asynchronous operations

- If you handle several requests at the same time, which is often the case when handling them in asynchronous manner, you must dequeue the requests from the queue :
void blkdev_dequeue_request(struct request *req);

- If needed, you can also put a request back in the queue :
void elv_requeue_request(struct request_queue *queue, struct request *req);

Asynchronous operations (2)

- Once the request is outside the queue, it's the responsibility of the driver to process all segments of the request
- Either by looping until blk_end_request() returns 0
- Or by using the rq_for_each_segment() macro

```
struct bio_vec *bvec;
struct req_iterator iter;
rq_for_each_segment(bvec, rq, iter)
{
        /*      rq->sector contains the current sector
                page_address(bvec->bv_page) + bvec->bv_offset points to the data
                bvec->bv_len is the length */

        rq->sector += bvec->bv_len / KERNEL_SECTOR_SIZE;
}

blk_end_request(rq, 0, rq->nr_sectors << 9);
```

DMA

- The block layer provides an helper function to « convert » a request to a scatter-gather list :

```
int blk_rq_map_sg(struct request_queue *q,
                  struct request *rq,
                  struct scatterlist *sglist)
```

- sglist must be a pointer to an array of struct scatterlist, with enough entries to hold the maximum number of segments in a request. This number is specified at queue initialization using blk_queue_max_hw_segments().
- The function returns the actual number of scatter gather list entries filled.

DMA (2)

- Once the scatterlist is generated, individual segments must be mapped at addresses suitable for DMA, using :

```
int dma_map_sg(struct device *dev,
        struct scatterlist *sglist,
        int count,
        enum dma_data_direction dir);
```

- dev is the device on which the DMA transfer will be made
- dir is the direction of the transfer (DMA_TO_DEVICE, DMA_FROM_DEVICE, DMA_BIDIRECTIONAL)
- The addresses and length of each segment can be found using sg_dma_addr() and sg_dma_len() on scatterlist entries.

DMA (3)

- After the DMA transfer completion, the segments must be unmapped, using

```
int dma_unmap_sg(struct device *dev,
                 struct scatterlist *sglist,
                 int hwcount,
                 enum dma_data_direction dir)
```

MMC / SD

Block layer

↓

MMC block device driver
CONFIG_MMC_BLOCK
drivers/mmc/card/{block,queue}.c

↓

MMC Core
CONFIG_MMC
drivers/mmc/core/

↓

MMC Host Driver
CONFIG_MMC_...
drivers/mmc/host/...

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)
- At unregistration
- void mmc_remove_host(struct mmc_host *host)
- void mmc_free_host(struct mmc_host *host)

MMC host driver (2)

- 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);
```