Laboratory 9: "Block device drivers" (two weeks)

Arkadiusz Chrobot, PhD
Karol Tomaszewski, PhD

June 7, 2024

Contents

Introduction

This document refers to creating drivers for block devices. Handling this group of devices is more complex than handling of character devices, because the efficiency of the entire computer system depends on the efficiency of this kind of devices.

These devices send information in portions whose size is a multiple of the sector size and allow free access to data. The characteristics of processing I/O requests related to block devices are presented in Chapter 1. The Linux kernel has an extensive infrastructure that is only associated with block device support. It is described in subsection 1.1. Chapter 2 briefly introduces the specifics of how block device drivers work. Chapter 3 presents the API used to create these drivers, and chapter 4 contains the source code for an example block device driver. The next two chapters (5 and 6) describe how to configure and use the sample driver. The guide ends with a list of tasks to be carried out as part of the laboratory.

1. Processing block operations

Similarly to character devices, the request for an I/O operation associated with a block device is initiated by a user space process or thread using one of the system calls, such as `read()` and `write()`, and then it is processed by Virtual File System that recognizes them and redirects them to the driver of the real file system associated with the given block device (Fig. 1). This driver first checks the type of request. If it is a write operation, the driver puts the stored data into the buffer, marking the buffer as intended for later writing to the medium. However, if it is a read operation, the driver checks if the requested data is already in any of the buffers. If so, this data is transferred to the user space and the operation is completed. If the read data is not in the buffers, or the buffer containing the modified data must be written to the medium, then the kernel initiates the proper I/O operation by redirecting a request to be performed in the *Block Operations Layer* where `bio`[1] structures are created at first to describe this operation. What happens next depends on the construction of the block device to which the request is redirected and the design of its driver. Three modes of block operation are possible:

without a queue the driver receives the `bio` structure directly and performs the operation described by it; this mode is intended for devices offering constant access time to each location on their medium, e.g. for USB memory,

with a single queue the `bio` structures are combined into larger structures describing requests, which are then placed in a queue and scheduled by an I/O scheduler; only the ordered queue goes to the driver, which performs the operations described by individual requests contained inside; this mode is intended for devices such as hard drives and optical disks.

[1]This name is an abbreviation of the *Block Input/Output Operation*.

with multiple queues this mode is designed for multiprocessor systems equipped with SSD drives; each processor places a requests for I/O in its private queue; then these requests are transferred to the queues of the device driver; their number depends on the capabilities of the SSD device that it supports; from these queues, the driver removes the requests and performs them in parallel.

The last of these operating modes will not be described in more detail in this document. Information about it can be found, e.g. at: `https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq)`

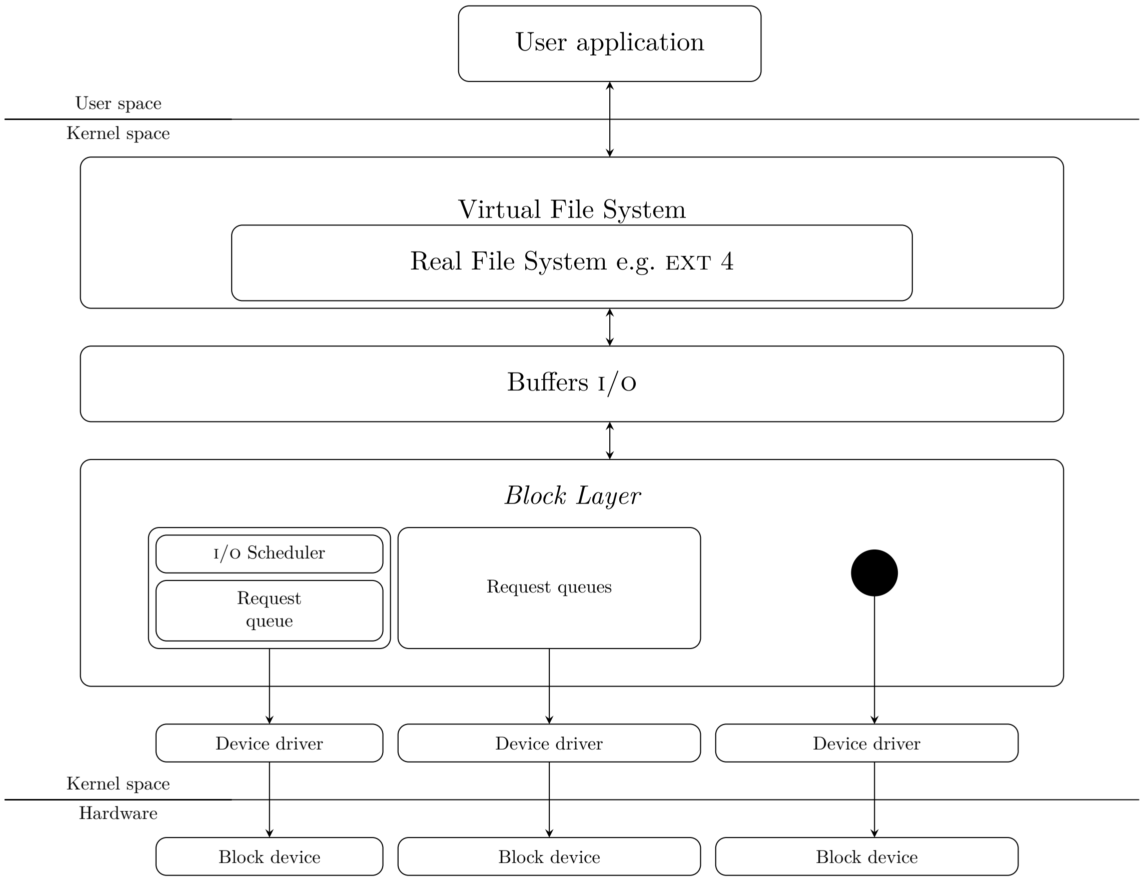

Figure 1: Processing of the I/O requests related to block devices (based on `http://free-electrons.com/doc/legacy/block-drivers/block_drivers.pdf` and `https://www.thomas-krenn.com/de/wikiDE/images/5/50/Linux-storage-stack-diagram_v4.0.svg`)

1.1. Block Operations Layer

The Linux kernel has a separate subsystem for handling operations on block devices (or block operations for short), which is called the Block Operations Layer or, more specifically, the Block Layer. The purpose of introducing such a subsystem was to ensure the efficiency of block operations. This layer creates `bio` structures of type `struct bio`, which describe block operations at a low level. A single such structure describes an operation that relates to contiguous sectors forming a contiguous area on the media, but associated with buffers in the operational memory, which may not necessarily form a contiguous area. In this operation, the buffers also do not have to participate in full. The fragments that are actually used are called `segments`. While the buffer has the maximum size of a single page of memory and it is its most common size, the segment is always smaller than that. A single segment is described with a structure of type `bio_vec`. A single `bio` structure contains a list of such segments. The `bio` structures are grouped into structures describing a single request of type `struct request`, and these are placed in the request queue of type `struct request_queue`. These dependencies are illustrated in Figure 2.

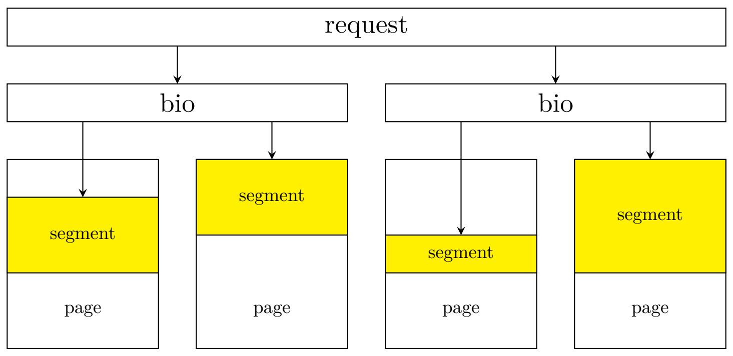

Figure 2: Dependencies between data structures associated with the block operation layer

This subsection does not cover the role of the Block Layer in a multiple queue operation mode.

2. Block device drivers

Block device drivers, similar as for character devices, provide definitions of functions that are called when requests from user processes/threads are processed. The devices they support may have sectors of different sizes, but internally the kernel assumes that the sector is 512 bytes in size. A file system is embedded in each such device. Some of these devices may additionally be partitioned. To make it possible for a single driver to handle several partitions, it must obtain the major number and minor numbers (one for each partition), just like with the character device driver. This guide will present a driver that supports a pseudo-block device that is a RAM-drive. This means that the device is efficient enough, so the functions of the driver do not have to put the user processes/threads that activated them in the waiting state, because the device provides the necessary data very quickly. In the case of real block devices, however, this need may arise, as well as the need to respond to signals sent to user processes/threads. The guide will not describe the DMA transmission support, which is often used in real block devices. The source codes for other block device drivers, both real and pseudo-devices, can be found among the kernel source code in the `drivers/block` directory. Particularly noteworthy is the source code for the `null_blk` driver. It is a driver used to assess the efficiency of the block operation modes described earlier and can be configured to work in each of them. Description of its use is in the kernel source code documentation directory.

3. Description of block device drivers API

The description of API for block device drivers should begin with the most important data structures. Listing 1 contains the definition of the `struct gendisk` type which is used to create structures that contain block device characteristics. In other words, this structure is equivalent to a `struct cdev` structure used for character devices.

Listing 1: The `struct gendisk` structure type

```c
struct gendisk {
        /* major, first_minor and minors are input parameters only,
         * don't use directly.  Use disk_devt() and disk_max_parts().
         */
        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[DISK_NAME_LEN];  /* name of major driver */
        char *(*devnode)(struct gendisk *gd, umode_t *mode);

        unsigned int events;            /* supported events */
        unsigned int async_events;      /* async events, subset of all */

        /* Array of pointers to partitions indexed by partno.
         * Protected with matching bdev lock but stat and other
         * non-critical accesses use RCU.  Always access through
         * helpers.
         */
        struct disk_part_tbl __rcu *part_tbl;
        struct hd_struct part0;

        const struct block_device_operations *fops;
        struct request_queue *queue;
        void *private_data;

        int flags;
        struct device *driverfs_dev;  // FIXME: remove
        struct kobject *slave_dir;

        struct timer_rand_state *random;
        atomic_t sync_io;               /* RAID */
        struct disk_events *ev;
#ifdef  CONFIG_BLK_DEV_INTEGRITY
        struct kobject integrity_kobj;
#endif  /* CONFIG_BLK_DEV_INTEGRITY */
        int node_id;
};
```

The `struct gendisk` type is defined in the `linux/genhd.h` header file. The most important fields included in this structure type are:

`major` — contains the major number assigned to the driver,

`firstminor` — contains the first minor number assigned to the driver,

`minors` — the amount of minor numbers assigned to the driver,

`disk_name` — a character string that is the name of the driver (max. 31 characters),

`fops` — a pointer to a structure of type `struct block_device_operations` which is an array of methods for block devices,

`queue` — a pointer to the request queue,

`private_data` — a pointer of type `void *` on the memory area containing private data of the driver,

`flags` — a field in which the values of particular bits specify flags describing the behavior of the supported block device. The values of these bits can be changed using bit operators and e.g. the following constants:

  `GENHD_FL_REMOVABLE` — specifies a device with removable media,

  `GENHD_FL_MEDIA_CHANGE_NOTIFY` — the driver will generate a notification about a change of media in the device,

  `GENHD_FL_SUPPRESS_PARTITION_INFO` — the driver will not transfer information about partitions located on the device to the `procfs` file system.

  There are also other constants associated with this field, but they will not be described in this document.

Listing 2: The `struct block_device_operations` structure type

```c
struct block_device_operations {
	int (*open) (struct block_device *, fmode_t);
	void (*release) (struct gendisk *, fmode_t);
	int (*rw_page)(struct block_device *, sector_t, struct page *, int rw);
	int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
	int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
	long (*direct_access)(struct block_device *, sector_t, void __pmem **,
			unsigned long *pfn);
	unsigned int (*check_events) (struct gendisk *disk,
				      unsigned int clearing);
	/* ->media_changed() is DEPRECATED, use ->check_events() instead */
	int (*media_changed) (struct gendisk *);
	void (*unlock_native_capacity) (struct gendisk *);
	int (*revalidate_disk) (struct gendisk *);
	int (*getgeo)(struct block_device *, struct hd_geometry *);
	/* this callback is with swap_lock and sometimes page table lock held */
	void (*swap_slot_free_notify) (struct block_device *, unsigned long);
	struct module *owner;
	const struct pr_ops *pr_ops;
};
```

Listing 2 contains the definition of the `struct block_device_operations` structure type. It specifies structures that are equivalent to `struct file_operations`, but dedicated strictly for block devices. It is worth noting that among the function pointers being the fields of the `block_device_operations` type structure, the `write()` and `read()` are not listed. This is because these methods are not used by block devices. A programmer writing a block device driver does not have to define all the methods pointed by the fields of the described structure. In the simplest case, it can be limited to assigning only the value to the `owner` field. This value is the pointer to a structure of type `struct module` returned by `THIS_MODULE` macro. If the block device is more complex to use, then it may be necessary to define other methods. A brief description of tasks performed by some of them can be found below:

`int (*open) (struct block_device *, fmode_t)` — the method indicated by this pointer has the same meaning as in the case of character devices, i.e. it can be responsible for the initialization of the driver data structure, enabling the block device and other activities preparing the device for operation,

`void (*release) (struct gendisk *, fmode_t)` — the method indicated by this pointer has the same meaning as in the case of character devices, i.e. it is responsible for the operations that finalize block device handling,

`int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long)` — a similar pointer also exists for character devices; the indicated method is responsible for performing operations on a block device that are not typical operations on a device file.

`int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long)` — the method indicated by this pointer performs the same role as `ioctl()`, but is intended for 32-bit applications executed on a 64-bit operating system,

`long (*direct_access)(struct block_device *, sector_t, void __pmem **, unsigned long *pfn)` — the method indicated to by this pointer is called when the user application requests direct access to data on the device media, i.e. via the device file and bypassing the file system embedded in the block device,

`unsigned int (*check_events) (struct gendisk *disk, unsigned int clearing);` — the method indicated by this pointer is associated with handling events related to the block device supported by the driver; among others, it replaces the `media_changed()` method, which checked if the media medium on the device with removable media (e.g. DVD or CD) has been changed,

`int (*revalidate_disk) (struct gendisk *)` — the method indicated to by this pointer is called by the system kernel when a change of media is discovered in a removable media block device,

`int (*getgeo)(struct block_device *, struct hd_geometry *)` — the method indicated to by this pointer is called from the `ioctl()` system call. It returns (by its second call argument) the information about the ”geometry” of the block device, i.e. the number of its heads, cylinders and sectors. Defining this function is necessary if the device media is meant to be partitioned.

Listing 3 contains a definition of the `struct request` type, which is the structure describing the I/O request.

Listing 3: Definition of the `struct request`

```
struct request {
        struct list_head queuelist;
        union {
                struct call_single_data csd;
                unsigned long fifo_time;
        };

        struct request_queue *q;
        struct blk_mq_ctx *mq_ctx;

        u64 cmd_flags;
        unsigned cmd_type;
        unsigned long atomic_flags;

        int cpu;

        /* the following two fields are internal, NEVER access directly */
        unsigned int __data_len;        /* total data len */
        sector_t __sector;              /* sector cursor */

        struct bio *bio;
        struct bio *biotail;

        /*
         * The hash is used inside the scheduler, and killed once the
         * request reaches the dispatch list. The ipi_list is only used
         * to queue the request for softirq completion, which is long
         * after the request has been unhashed (and even removed from
         * the dispatch list).
         */
        union {
                struct hlist_node hash; /* merge hash */
                struct list_head ipi_list;
        };

        /*
         * The rb_node is only used inside the io scheduler, requests
         * are pruned when moved to the dispatch queue. So let the
         * completion_data share space with the rb_node.
         */
        union {
                struct rb_node rb_node; /* sort/lookup */
                void *completion_data;
        };

        /*
         * Three pointers are available for the IO schedulers, if they need
```

```
	 * more they have to dynamically allocate it.  Flush requests are
	 * never put on the IO scheduler. So let the flush fields share
	 * space with the elevator data.
	 */
	union {
		struct {
			struct io_cq		*icq;
			void			*priv[2];
		} elv;

		struct {
			unsigned int		seq;
			struct list_head	list;
			rq_end_io_fn		*saved_end_io;
		} flush;
	};

	struct gendisk *rq_disk;
	struct hd_struct *part;
	unsigned long start_time;
#ifdef CONFIG_BLK_CGROUP
	struct request_list *rl;		/* rl this rq is alloced from */
	unsigned long long start_time_ns;
	unsigned long long io_start_time_ns;    /* when passed to hardware */
#endif
	/* Number of scatter-gather DMA addr+len pairs after
	 * physical address coalescing is performed.
	 */
	unsigned short nr_phys_segments;
#if defined(CONFIG_BLK_DEV_INTEGRITY)
	unsigned short nr_integrity_segments;
#endif

	unsigned short ioprio;

	void *special;		/* opaque pointer available for LLD use */

	int tag;
	int errors;

	/*
	 * when request is used as a packet command carrier
	 */
	unsigned char __cmd[BLK_MAX_CDB];
	unsigned char *cmd;
	unsigned short cmd_len;

	unsigned int extra_len;	/* length of alignment and padding */
	unsigned int sense_len;
	unsigned int resid_len;	/* residual count */
	void *sense;

	unsigned long deadline;
	struct list_head timeout_list;
	unsigned int timeout;
	int retries;

	/*
	 * completion callback.
	 */
```

```c
        rq_end_io_fn *end_io;
        void *end_io_data;

        /* for bidi */
        struct request *next_rq;
};
```

Most of these fields are not directly supported by block device drivers. However the following fields are worth to be noted:

`q` — field allowing to place the structure in the request queue,

`cmd_type` — this field specifies the type of request which is described by the `struct request` structure; it can be read directly; if the request is related to an I/O operation, the value of this field is equal to the `REQ_TYPE_FS` constant; the `struct request` structure can also describe requests that are not directly related to operations on the file system, such as requests related to support for devices with the SCSI interface,

`bio` — field indicating the first `bio` structure included in the request,

`biotail` — a field indicating the last `bio` structure included in the request; the remaining structures located between those indicated by both described fields are connected in a queue,

`ioprio` — field specifying priority of the request,

`next_rq` — a field that points to the next `struct request` structure in the request queue.

Please also note that some of the fields in this structure are used by I/O schedulers.

Listing 4, contains a definition of the structure of type `struct bio`, which describes the `bio` structures mentioned earlier.

Listing 4: Definition of the struct bio

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

        struct bvec_iter        bi_iter;

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

        /*
         * To keep track of the max segment size, we account for the
         * sizes of the first and last mergeable segments in this bio.
         */
        unsigned int            bi_seg_front_size;
        unsigned int            bi_seg_back_size;

        atomic_t                __bi_remaining;

        bio_end_io_t            *bi_end_io;

        void                    *bi_private;
#ifdef CONFIG_BLK_CGROUP
	/*
	 * Optional ioc and css associated with this bio.  Put on bio
	 * release.  Read comment on top of bio_associate_current().
	 */
	struct io_context	*bi_ioc;
	struct cgroup_subsys_state *bi_css;
#endif
	union {
#if defined(CONFIG_BLK_DEV_INTEGRITY)
		struct bio_integrity_payload *bi_integrity; /* data integrity */
#endif
	};

	unsigned short		bi_vcnt;	/* how many bio_vec's */

	/*
	 * Everything starting with bi_max_vecs will be preserved by bio_reset()
	 */

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

	atomic_t		__bi_cnt;	/* pin count */

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

	struct bio_set		*bi_pool;

	/*
	 * We can inline a number of vecs at the end of the bio, to avoid
	 * double allocations for a small number of bio_vecs. This member
	 * MUST obviously be kept at the very end of the bio.
	 */
	struct bio_vec		bi_inline_vecs[0];
};
```

Similar to the `struct request` structure, most of these fields are not directly handled by the block device driver, but the important fields from the point of view of the programmer creating the driver are:

`bi_rw` — a field whose older bits determine the priority of operations and the younger bits define the operations direction (write or read),

`bi_iter` — a field containing information that allows to iterate over the array of segments contained in the `bio` structure,

`bi_vnct` — a field specifying the number of segments related to the `bio` structure,

`bi_io_vec` — array of segments related to a given `bio` structure.

The aforementioned segment table consists of `struct bio_vec` elements whose definition is contained in listing 5.

Listing 5: Definition of the `struct bio_vec`

```c
struct bio_vec {
	struct page	*bv_page;
	unsigned int	bv_len;
	unsigned int	bv_offset;
};
```

The meaning of the individual fields of this structure are as follows:

bv_page — the field contains the address of the structure that defines the frame in which the page containing the buffer for input-output operations and the segment that is part of this buffer is located; the field does not contain the virtual address of the page, because it may belong to high memory,

bv_len — the field specifies the size of the segment,

bv_offset — the field specifies the offset from the beginning of the page from which the segment begins.

Listing 6 contains the definition of the structure of type `bvec_iter`, which specifies structures that store data related to iterating over a segment table belonging to the `bio` structure. The `bi_iter` field of the `bio` structure is of this type.

Listing 6: Definition of the `struct bvec_iter`

```
struct bvec_iter {
        sector_t                bi_sector;      /* device address in 512 byte
                                                   sectors */
        unsigned int            bi_size;        /* residual I/O count */

        unsigned int            bi_idx;         /* current index into bvl_vec */

        unsigned int            bi_bvec_done;   /* number of bytes completed in
                                                   current bvec */
};
```

The fields in this structure have the following meanings:

bi_sector — the number of the starting sector to which the block operation relates,

bi_size — the number of I/O operations remaining to be performed,

bi_idx — index of the segment table element that is currently being processed,

bi_bvec_done — number of bytes processed in the current segment.

The functions and macros used by block device drivers are declared or defined in the following header files: `linux/genhd.h`, `linux/bio.h`, `linux/bkldev.h` and `linux/fs.h`.

Regardless of whether the driver supports a block device with or without using a request queue, it uses the following macros and functions defined in those header files:

int register_blkdev(unsigned int major, const char *name) — a function that registers a new block device by reserving its major number. As the first call argument, it takes the major number that the programmer wants to reserve. If this argument is `0`, the function will reserve the first available major number and return its value. It will be a natural number between `1` and `255`. As a second argument, the function takes a string that is a unique name for the block device. If registration fails, the function returns a negative value or zero.

void unregister_blkdev(unsigned int, const char *) — function unregisters a block device. As the call arguments, it takes the major number assigned to the device and the unique device name.

struct gendisk *alloc_disk(int minors) — the function allocates memory to `struct gendisk`. As a call argument, it takes the amount of minor numbers that the driver will use. The function returns the address of the `struct gendisk` structure that was created, or `NULL` if it fails. If the device is partitioned, a separate such structure is created for each partition.

void add_disk(struct gendisk *disk) — a function that registers a previously initiated `struct gendisk` structure in the system. As a result of this registration, appropriate directories and files are created in the `sysfs` file system and appropriate notifications are sent to the user space, in response to which `udev` creates block device files for the driver. In case the device is partitioned, the one such structure must be registered for each partition.

`void del_gendisk(struct gendisk *disk)` — the function unregisters a `struct gendisk` structure from the system, whose address will be given as the call argument. The result of its invocation is to send notifications to the user space that require `udev` to remove the block device file and to remove the appropriate files and directories from the `sysfs` file system.

`void put_disk(struct gendisk *disk)` — this function reduces the value of the reference count to the `struct gendisk` structure whose address was given as the call argument. If this counter value will reach zero, the function will release memory allocated to this structure.

`void set_capacity(struct gendisk *disk, sector_t size)` — an `inline` function that allows to set the capacity of a block device. As the first call argument, it takes the `struct gendisk` structure address associated with the block device, and as the second argument it takes the size of this device expressed in the number of sectors.

`sector_t get_capacity(struct gendisk *disk)` — an `inline` function that allows to read the capacity of a block device expressed in sectors. It takes the `struct gendisk` structure address as its call argument.

The following functions and macros are useful for handling the `bio` structures in a single request queue mode as well as without the request queue:

`bio_end_sector(bio)` — a macro that returns the number of the last sector affected by the block operation described by the `bio` structure given to it as a call argument.

`void bio_endio(struct bio *bio)` — a function that indicates that the operation described by the `bio` structure has finished successfully.

`void bio_io_error(struct bio *bio)` — a function that indicates that the operation described by the `bio` structure has failed.

`bio_data_dir(bio)` — a macro that returns a value indicating whether the operation described by the `bio` structure given as a call argument is a write or read operation. In the first case the returned value is equal to the `WRITE` constant and in the second case it is equal to `READ`.

`bio_for_each_segment(bvl, bio, iter)` — a macro that allows to iterate over the segments associated with the `bio` structure. It takes three call arguments. The first argument is a structure of type `bio_vec`, into which the values subsequent elements of the `bi_io_vec` array will be stored. The second argument is the `bio` structure to which the array belongs. The third argument is a structure of type `struct bvec_iter`.

`struct request_queue *blk_alloc_queue(gfp_t gfp_mask)` — a function that allocates memory to the request queue. This memory must be reserved by the driver, even if it works in a mode without a queue. As a call argument, this function takes a type tag associated with memory allocation.

`void blk_cleanup_queue(struct request_queue *q)` — a function that releases the memory allocated to the request queue whose address is given to it as a call argument.

In block device mode with a single queue, the following macros and functions are helpful:

`struct request_queue *blk_init_queue(request_fn_proc *, spinlock_t *)` — a function that allocates memory to the request queue and initializes it. It returns the address of the queue created or `NULL` in case of failure. The allocated queue can be released by calling the `blk_cleanup_queue()` function. This function takes two call arguments. The first is the address of the function that will remove and process the requests from the queue. The second is the loop lock address that will protect this queue from concurrent access. The request processing function must have the following prototype:

```
void request_fn_proc(struct request_queue *q);
```

The functions described below will be helpful for its implementation.

`struct request *blk_fetch_request(struct request_queue *q)` — a function that removes a single request from the request queue. It returns the address of a `struct request` structure describing this request or `NULL` if the queue is empty.

`rq_data_dir(rq)` — a macro whose value determines whether the operation being requested is a write or read operation. It returns the same values as `bio_data_dir`.

`rq_for_each_segment(bvl, _rq, _iter)` — a macro that is used to iterate over the segments associated with a given request. It takes the same call arguments as `bio_for_each_segment`, except for the middle one. In case of this macro this argument is a structure of type `struct request`.

`void blk_complete_request(struct request *req)` — a function that signals when the request has been processed. It takes as a call argument a pointer to a `struct request` structure that describes the request.

In mode without a queue it is still necessary to call the following function:

`void blk_queue_make_request(struct request_queue *, make_request_fn *)` — this function registers a function that will process `bio` structures incoming into the driver. As the first call argument it takes the address of the request queue to which memory has been allocated using the `blk_alloc_queue()` function. As a second argument, it takes a pointer to a function that is meant to process the `bio` structures. This function must have the following prototype:

```
blk_qc_t (make_request_fn) (struct request_queue *q, struct bio *bio);
```

This function should return `BLK_QC_T_NONE` as its constant value.

The subprograms described earlier do not cover the entire spectrum of functions and macros that are associated with handling of block operations by device drivers. However, they are the minimum necessary to implement a working driver.

4. Example

Listing 7 contains the source code of a simple driver that handles the RAM-disk in mode without a queue.

Listing 7: Block pseudo-device driver

```c
#include<linux/module.h>
#include<linux/genhd.h>
#include<linux/vmalloc.h>
#include<linux/fs.h>
#include<linux/bio.h>
#include<linux/blkdev.h>

#define DEVICE_SIZE 4*1024*1024

static int sector_size = 512;
static int major = 0;
static struct sbd_struct
{
        struct gendisk *gd;
        void *memory;
} sbd_dev;

static inline int transfer_single_bio(struct bio *bio)
{
        struct bvec_iter iter;
        struct bio_vec vector;
        sector_t sector = bio->bi_iter.bi_sector;
        bool wirte = bio_data_dir(bio) == WRITE;

        bio_for_each_segment(vector,bio,iter) {
                unsigned int len = vector.bv_len;
                void *addr = kmap(vector.bv_page);
                if(wirte)
                        memcpy(sbd_dev.memory+sector*sector_size,addr+vector.bv_offset,len);
                else
                        memcpy(addr+vector.bv_offset,sbd_dev.memory+sector*sector_size,len);
                kunmap(addr);
                sector += len >> 9;
        }
        return 0;
}

static blk_qc_t make_request(struct request_queue *q, struct bio *bio)
{
        int result=0;

        if(bio_end_sector(bio)>get_capacity(bio->bi_disk))
                goto mrerr0;

        result = transfer_single_bio(bio);
        if(unlikely(result!=0))
                goto mrerr0;

        bio_endio(bio);
        return BLK_QC_T_NONE;
mrerr0:
        bio_io_error(bio);
        return BLK_QC_T_NONE;
}

static struct block_device_operations block_methods = {
        .owner = THIS_MODULE
};


static int __init sbd_constructor(void)
{
        sbd_dev.memory = vmalloc(DEVICE_SIZE);
        if(!sbd_dev.memory) {
                pr_alert("Memory allocation error!\n");
                goto ier1;
        }
        sbd_dev.gd = alloc_disk(1);
        if(!sbd_dev.gd) {
                pri_alert("General disk structure allocation error!\n");
                goto ier2;
        }
        major = register_blkdev(major,"sbd");
        if(major<=0) {
                pr_alert("Major number allocation error!\n");
                goto ier3;
        }
        pr_info("[sbd] Major number allocated: %d.\n",major);
        sbd_dev.gd->major = major;
        sbd_dev.gd->first_minor = 0;
        sbd_dev.gd->fops = &block_methods;
        sbd_dev.gd->private_data = NULL;
        sbd_dev.gd->flags|=GENHD_FL_SUPPRESS_PARTITION_INFO;
        strcpy(sbd_dev.gd->disk_name,"sbd");
        set_capacity(sbd_dev.gd,(DEVICE_SIZE)>>9);
        sbd_dev.gd->queue = blk_alloc_queue(GFP_KERNEL);
        if(!sbd_dev.gd->queue) {
```

```
88                  pr_alert("Request queue allocation error!\n");
89                  goto ier4;
90          }
91          blk_queue_make_request(sbd_dev.gd->queue,make_request);
92          pr_infor("[sbd] Gendisk initialized.\n");
93          add_disk(sbd_dev.gd);
94          return 0;
95  ier4:
96          unregister_blkdev(major,"sbd");
97  ier3:
98          put_disk(sbd_dev.gd);
99  ier2:
100         vfree(sbd_dev.memory);
101 ier1:
102         return -ENOMEM;
103 }
104
105 static void __exit sbd_desctructor(void)
106 {
107         del_gendisk(sbd_dev.gd);
108         blk_cleanup_queue(sbd_dev.gd->queue);
109         unregister_blkdev(major,"sbd");
110         put_disk(sbd_dev.gd);
111         vfree(sbd_dev.memory);
112 }
113
114 module_init(sbd_constructor);
115 module_exit(sbd_desctructor);
116
117 MODULE_LICENSE("GPL");
118 MODULE_AUTHOR("Arkadiusz Chrobot <a.chrobot@tu.kielce.pl>");
119 MODULE_DESCRIPTION("A block pseudo-device.");
120 MODULE_VERSION("1.0");
```

Lines 1–6 contain preprocessor instructions that include header files in the driver code. Most of them were mentioned earlier in this guide. The header file named `module.h` contains subprograms related to the support of kernel modules, and the `vmalloc.h` file has been included due to its functions allowing to allocate and release a continuous area of virtual memory. Line 8 contains a definition of a constant variable that specifies the RAM-disk capacity in bytes. It is equal to 4 MiB. In line 10, a variable was declared and initiated specifying the size of a single sector, i.e. 512 B. Line 11 contains the variable declaration in which the major number will be stored. Its initial value will be `0`, because we want the `register_blkdev()` function to give the driver the first available major number. Lines 12–16 contain definitions of the structural type, and additionally line 16 contains declaration of a structure of this type. This is the private driver structure. It contains two pointer fields. The `gd` field will contain the `struct gendisk` structure address. The RAM-disk will have one partition, hence only one structure is needed. The second field, called `memory`, will contain the address of the memory area, which will be the ”medium” of the RAM-disk.

Lines 18–36 contain the definition of the `transfer_single_bio()` function, which is responsible for the execution of the block operation described by the `bio` structure, whose address is given by its call argument. In lines 20–23, there are local variable declarations for this function. Line 20 contains the declaration of the `iter` variable, which is used by the `bio_for_each_segment`. Line 21 contains a declaration of the `vector` variable, which will contain information about the currently processed segment belonging to the `bio` structure. Line 22 contains the declaration of the `sector` variable, which will contain the sector number of the starting area of the medium to be written or read. It is initiated by the starting sector number of the entire `bio` structure. In line 23, a variable of the `bool` type is declared, which will specify the type of operation being carried out, i.e. writing or reading. It is assigned with an expression value that compares the result of the `bio_data_dir` macro used for the processed `bio` structure with the `WIRTE` constant. If this value is `true`, then write operation will be performed, and if it is `false`, read operation will be performed. Lines 25–34 contain a loop implemented using the `bio_for_each_segment` macro, which iterates over all segments related to the `bio` structure and processes them. This macro saves information about the current segment in the `vector` variable. In line 16, the variable `len` is declared, which is assigned with the size of the sector, expressed as the number of bytes. In line 27 the variable `addr` is declared, which is assigned with the virtual address of the page on which the processed segment is located. Because this page may belong to high memory, this address is obtained from a `struct page` structure using the `kmap()` function, and after the segment is processed it is released using the `kunmap()` function. In line 28, the function checks whether the read or the write operation is to be performed and, depending on the result, copies data from the segment to the medium or vice versa, using the `memcpy()` function. The amount of this data is determined by the value of the variable `len`. The segment start address is determined as the sum of the page beginning address and the offset value saved in the `bv_offset` field of the structure describing the currently processed segment. The beginning of a fragment of the memory that is the RAM-disk medium on which the block operation is to be performed is determined as the sum of the address of the beginning of the medium and the initial sector multiplied by the size of a single sector. After the corresponding copy is completed, the starting sector number is increased by the number of bytes copied divided by the size of a single sector (line 33). This is necessary to correctly address the next fragment of medium that will be associated with the next processed segment. Because in this case the number of bytes is divided by `512`, the bitwise right shift operator is used to increase its efficiency instead of the usual divide operator ($512 = 2^9$). The function returns `0` and terminates in line 35.

Lines 38–54 contain a definition of the `make_request()` function, which processes the `bio` structures obtained by the driver from the Block Operation Layer. In line 40 the local variable of this function is declared, which will store the result of the `transfer_single_bio()` function. Its initial value is set to `0`. In line 42, it is checked if the operation described by the `bio` structure does not exceed the size of the block device. This check consists in reading the sector number involved in this operation and comparing it with the capacity of this device, expressed in number of sectors. This capacity is read from a `struct gendisk` structure pointed indirectly by one of the fields of the `bio` structure. If the result of this check was positive, then control is passed to lines 51–53, where the function signaling the error of processing the `bio` structure is called (line 52) and the function `make_request()` ends its operation returning the appropriate value. However, if the condition from line 42 is not met, the `transfer_single_bio()` function is called, the result of which is written to the `result` variable. Then in line 46 it is checked if this result is different from zero. If so, then the control is transferred to function lines 51–53. The `unlikely` macro is used to indicate conditions that are unlikely to be met. In this case, due to the current structure of the `transfer_single_bio()` function, this is not possible at all. If the condition from line 46 is not met, the current function calls the function signaling successful completion of the `bio` structure processing (line 49) and terminates its operation by returning the appropriate value.

Lines 56–58 contain the declaration and initialization of a variable called `block_methods` of the `block_device_operations` structural type. Because we do not need a definition of any of the methods indicated by this structure, the driver only initializes the `owner` field of this structure.

Lines 61–101 contain a constructor, i.e. a function that initiates driver operation. In line 63, a memory area is allocated that will be the "medium" of the supported block device. The starting address of this area is saved in the `memory` field of the `sbd_dev` structure. Because this memory area does not have to be physically continuous, it is enough for it to be virtually continuous, thus the `vmalloc()` function is used to allocate it. The size of this area is determined by the `DEVICE_SIZE` constant. If the allocation of memory to "medium" fails, the constructor terminates by placing an exception message in the kernel buffer and signaling failure (lines 65 and 66). In line 68, memory is allocated for a `struct gen_disk` structure. Since the device will have only one partition, the value `1` is given as the call argument to `alloc_disk()` function, specifying that only one minor number will be needed. If this assignment fails, the constructor will put the appropriate message in the kernel buffer and pass control to the code, which will release previously allocated resources. Line 73 is the major number for the device and is stored in a variable named `major`. Again, if the allocation of this number fails, the function transfers control to the code releasing the previously allocated resources. In line 78 the function puts the message about the assigned major number in the kernel buffer. In lines 79–91, the appropriate `struct gen_disk` structure fields are initialized. The major number is saved in the `major` field of this structure (line 79), and the first and only minor number, whose value is `0`, is stored in the `first_minor` field (line 80). The address of the `block_methods` structure containing pointers to the block device file methods is stored in the `fops` field.

In line 82, a pointer to the memory area that stores the private data of the driver is initialized. Because this area will not be used by described driver, this pointer is assigned with a `NULL` value. A flag that disables placing partition information in the appropriate file in the `procfs` system is set in the `flags` field (line 83). In line 84, the `disk_name` field is assigned with a string being the name of the device. This is `"sbd"`, which is an abbreviation of *simple block device*. In line 85 the capacity of the supported device is specified. This is the size expressed in number of sectors, hence the second argument to the `set_capacity()` function call is the value of the expression dividing the size of the RAM-disk by the size of a single sector. Then, on line 86, memory is allocated to the request queue whose address is stored in the `queue` field. If this assignment fails, the driver will put the appropriate message in the kernel buffer and transfer control to the code releasing previously allocated resources. The `GFP_KERNEL` tag is given to the `blk_alloc_queue()` function call, which means that the regular kernel space allocation will be done. If it fails, the constructor will put an appropriate message in the kernel buffer and pass control to the code releasing previously allocated resources. On line 91, the constructor registers the `make_request()` function as the one that will be processing the `bio` structures, and then places a message in the kernel buffer that the `struct gen_disk` structure has been initialized. In line 93 this structure is added to the system and the constructor finishes his operations returning the value `0`. Lines 95–102 contain the source code that releases resources if other resource allocations fail. The following are released in order: major number, memory for `struct gen_disk` and memory which is the ”medium” of the block device. The last instruction executed under this code is the end of the constructor's operation with an exception code indicating problems with memory allocation (line 102).

Lines 105–112 contain a destructor, i.e. a function that finalizes the operation of the driver. In line 107, a `struct gen_disk` structure is deregistered. At line 108, memory for the request queue is released. On line 109, the major number is released, and on line 110, the `struct gen_disk` structure is deleted. Finally, in line 111, the memory that is the ”medium” of the device is released.

5. Udev configuration

Listing 8 contains the contents of a configuration file named `42-sbd.rules` for the `udevd` daemon, which should be placed in the `/etc/udev/rules.d/` directory. Thanks to it, the user named `pi`[2] will be able to perform operations on the device file, without using the `sudo` command, such as creating a file system or partitioning. The way of constructing the content of such a file has been described in the guide about character device drivers.

Listing 8: Configuration file for the udevd daemon

```
KERNEL=="sbd", NAME="sbd", OWNER="pi", MODE="0660"
```

6. Handling the device

Handling of the block devices in user space is more complicated than handling of the character devices. After loading the driver with the `insmod` command, a block device file called `sbd` is created in the (usually) `/dev` directory. First, it is necessary to create a file system for such a device using one of the `mkfs` commands. It is best to use the command to create the `ext2` file system in this case, because its structures consume relatively little space on the media, since it does not use *journalizing*. This system can be created on the device with the `mkfs.ext2 /dev/sbd` command. If it is performed by an unprivileged user and if `udev` has not been properly configured, it may be necessary to precede this command with the `sudo` command. After creating the file system on the block device, we can mount it, i.e. make it available to the user in a specific directory. Such a directory can be `/mnt` or one of its subdirectories, if there any. Mounting can be done with the `mount /dev/sbd /mnt` command. If it is performed by an unprivileged user, it may require to precede it with the `sudo` command. Only a privileged user can perform operations such as creating, deleting, reading and storing files and directories on a device mounted in such way. If we would like also other users to be able to perform those operations, we will need to give additional options to the `mount` command, described in the system manual. The mounted device can be unmounted using

[2]The file has been prepared for the Raspbian distribution for the Rasbperry Pi computer.

the `umount` command. In case of the described device this command will be: `umount /mnt`. Again, if it is done from an unprivileged user level, it will require to precede it with the `sudo` command. Unmounting of the block device is necessary so that the module can be removed with the `rmmod` command. It is worth noting that after unmounting the device and before removing the module, the data saved on the block device is not lost and can be accessed after mounting the device again.

Exercises

1. `[3 points]` Change the source code of the module from Listing 7 so that it does not use the `goto` instruction and remain it's functionality.
2. `[5 points]` Change the source code of the module from Listing 7 so that when it loads into the kernel it creates two partitions. It will be necessary to create two `struct gen_disk` structures among the other changes.
3. `[7 points]` Change the source code of the module from Listing 7 so that it performs block device support in single request queue mode.
4. `[3 points]` Change the source code of the module from Listing 7 so that the RAM-disk capacity is determined by the parameter when loading the driver into the system kernel.
5. `[5 points]` Change the source code of the module from Listing 7 so that it is possible to create partitions on the block device using the `cfdisk` command. It will be necessary to create a `getgeo()` method definition among the other changes.
6. `[7 points]` Change the source code of the module from Listing 7 so that instead of using a continuous memory area as a device ”medium” it will use the list, whose elements will be memory pages. Use the implementation of the bidirectional cyclic list provided by the kernel for this purpose.