Laboratorium 9: „Sterowniki urządzeń blokowych” (dwa zajęcia)

dr inż. Arkadiusz Chrobot

18 maja 2024

Spis treści

Wprowadzenie **1**

1. Przetwarzanie operacji blokowych **1**

1.1. Warstwa operacji blokowych 2

2. Sterowniki urządzeń blokowych **3**

3. Opis API sterowników urządzeń blokowych **3**

4. Przykład **12**

5. Konfiguracja `Udev` **16**

6. Obsługa urządzenia **17**

Zadania **17**

Wprowadzenie

Niniejsza instrukcja dotyczy tworzenia sterowników dla urządzeń blokowych. Obsługa tych urządzeń jest bardziej złożona od obsługi urządzeń znakowych, gdyż od jej efektywności zależy wydajności całego systemu komputerowego. Urządzenia te przesyłają informację porcjami, których wielkość jest wielokrotnością rozmiaru sektora i umożliwiają swobodny dostęp do danych. Charakterystyka przetwarzania żądań wejścia-wyjścia związanych z urządzeniami blokowymi przedstawiona jest w rozdziale 1 Jądro Linuksa posiada rozbudowaną infrastrukturę, która związana jest wyłącznie z obsługą urządzeń blokowych. Jest ona opisana w podrozdziale 1.1. Rozdział 2 krótko przedstawia specyfikę działania sterowników urządzenia blokowego. Rozdział 3 przedstawia API służące do tworzenia tych sterowników, a rozdział 4 kod przykładowego sterownika urządzenia blokowego. Dwa kolejne rozdziały (5 oraz 6) opisują sposób konfiguracji i użycia przykładowego sterownika. Instrukcja kończy się listą zadań do samodzielnego wykonania w ramach zajęć laboratoryjnych.

1. Przetwarzanie operacji blokowych

Podobnie jak w przypadku urządzeń znakowych żądanie wykonania operacji wejścia-wyjścia związanej z urządzeniem blokowym jest inicjowane przez proces lub wątek przestrzeni użytkownika przy pomocy któregoś z wywołań systemowych, takich jak `read()` i `write()`, a następnie jest przetwarzane przez Wirtualny System Plików, który je rozpoznaje i kieruje do sterownika rzeczywistego systemu plików związanego z danych urządzeniem blokowym (rys. 1). Ten sterownik najpierw sprawdza rodzaj żądania. Jeśli jest to zapis, to umieszcza on zapisywane dane w buforze, oznaczając ten bufor jako przeznaczony do późniejszego zapisu na nośnik. Jeśli jednak jest to odczyt, to sterownik sprawdza, czy żądane dane są już w którymś z buforów. Jeśli tak jest, to dane te przenoszone są do przestrzeni użytkownika i operacja jest kończona. Jeśli odczytywanych danych nie ma w buforach, lub bufor zawierający zmodyfikowane dane musi zostać zapisany na nośnik, to jądro rozpoczyna właściwą operację wejścia wyjścia kierując żądanie jej wykonania do Warstwy Operacji Blokowych (ang. *Block Layer*), gdzie najpierw tworzone są struktury `bio`[1] opisujące tę operację. To co się dzieje dalej jest zależne od budowy urządzenia blokowego, do którego skierowane jest żądanie i konstrukcji jego sterownika. Możliwe są trzy tryby realizacji operacji blokowej:

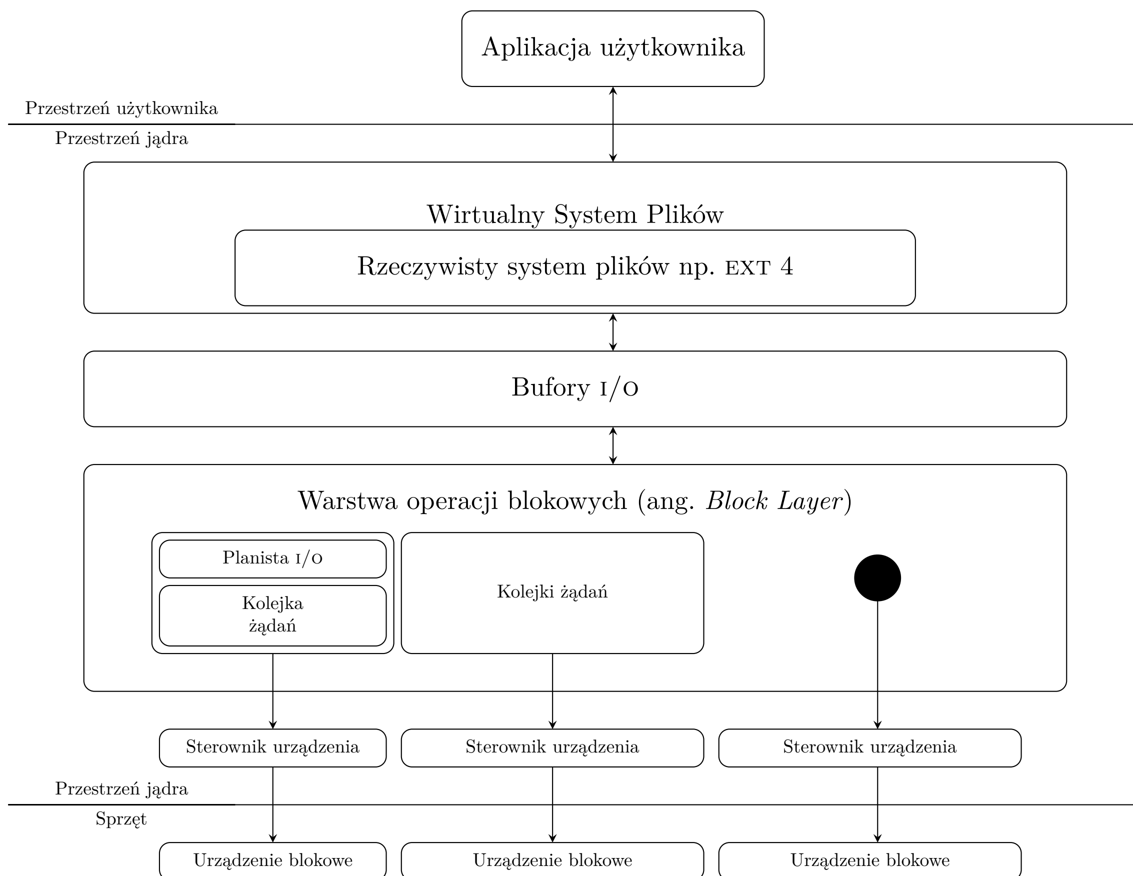
bez kolejki sterownik otrzymuje bezpośrednio strukturę `bio` i wykonuje opisaną przez nią operację; taki tryb pracy jest przeznaczony dla urządzeń oferujących stały czas dostępu do każdej lokacji na ich nośniku, np. dla pamięci USB,

[1] Ich nazwa jest skrótem od angielskich słów *Block Input/Output Operation*, czyli operacja wejścia-wyjścia.

z pojedynczą kolejką struktury `bio` łączone są w większe struktury opisujące żądania, które następnie są umieszczane w kolejce i szeregowane przez planistę wejścia-wyjścia; dopiero uporządkowana kolejka trafia do sterownika, który realizuje operacje opisane przez poszczególne żądania, które są w niej zawarte; ten tryb obsługi przeznaczony jest dla takich urządzeń jak dyski twarde i optyczne.

z wieloma kolejkami ten tryb przeznaczony jest dla systemów wieloprocesorowych, wyposażonych w urządzenia SSD (ang. *Solid State Device*); każdy procesor umieszcza żądania wykonania operacji wejścia-wyjścia w swojej prywatnej kolejce; następnie te żądania są przenoszone do kolejek, którymi dysponuje sterownik urządzenia; ich liczba zależna jest od możliwości urządzenia SSD, które on obsługuje; z tych ostatnich kolejek sterownik je zdejmuje i realizuje równolegle.

Ostatni z opisanych trybów obsługi nie będzie bardziej szczegółowo opisywany w tej instrukcji. Informacje mu poświęcone można znaleźć np. na stronie: `https://www.thomas-krenn.com/en/wiki/Linux_Multi-Queue_Block_IO_Queueing_Mechanism_(blk-mq)`

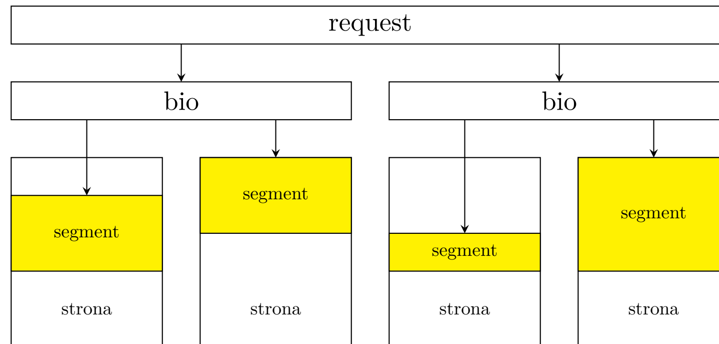


Rysunek 1: Przetwarzanie żądań I/O związanych z urządzeniami blokowymi (na podstawie `http://free-electrons.com/doc/legacy/block-drivers/block_drivers.pdf` oraz `https://www.thomas-krenn.com/de/wikiDE/images/5/50/Linux-storage-stack-diagram_v4.0.svg`)

1.1. Warstwa operacji blokowych

Jądro systemu Linux dysponuje wydzielonym podsystemem obsługi operacji na urządzeniach blokowych (w skrócie: operacji blokowych), który nazywany jest Warstwą Operacji Blokowych lub, krócej Warstwą Blokową. Celem wprowadzenia takiego podsystemu było zapewnienie efektywności wykonywania operacji blokowych. W tej warstwie tworzone są struktury `bio`, typu `struct bio`, które opisują na niskim poziomie operacje blokowe. Pojedyncza taka struktura opisuje operację, która dotyczy tworzących na nośniku ciągły obszar, przyległych do siebie sektorów, ale związaną z buforami w pamięci operacyjnej,

które niekoniecznie muszą tworzyć obszar ciągły. W takiej operacji bufory nie muszą także brać udziału w całości. Ich fragmenty, które faktycznie są używane, nazywają się `segmentami`. O ile bufor ma maksymalny rozmiar pojedynczej strony pamięci i jest to jego najczęściej spotykana wielkość, to segment jest od niego zawsze mniejszy. Pojedynczy segment jest opisany strukturą typu `bio_vec`. Pojedyncza struktura `bio` zawiera listę takich segmentów. Struktury `bio` są grupowane w struktury opisujące pojedyncze żądanie typu `struct request`, a te z kolei są umieszczane w kolejce żądań typu `struct request_queue`. Te zależności są zilustrowane na rysunku 2. W tym podrozdziale pominięte są informacje na temat roli



Rysunek 2: Powiązania między strukturami danych związanymi z warstwą operacji blokowych

Warstwy Operacji Blokowych w trybie obsługi z użyciem wielu kolejek.

2. Sterowniki urządzeń blokowych

Sterowniki urządzeń blokowych, podobnie jaki ich odpowiedniki dla urządzeń znakowych, dostarczają definicji funkcji, które są wywoływane w trakcie realizacji żądań pochodzących od procesów i wątków użytkownika. Urządzenia, które one obsługują mogą posiadać sektory o różnej wielkości, ale wewnętrznie jądro systemu przyjmuje, że sektor ma rozmiar 512 bajtów. Na każdym takim urządzeniu osadzony jest system plików. Niektóre z tych urządzeń mogą dodatkowo być podzielone na partycje. Aby jeden sterownik mógł obsłużyć kilka partycji musi, podobnie jak sterowniki urządzeń znakowych, pozyskać numer główny i numery poboczne (po jednym dla każdej partycji). W tej instrukcji zostanie przedstawiony sterownik obsługujący pseudo urządzenie blokowe, które jest RAM-dyskiem. Oznacza to, że jest ono na tyle wydajne, iż funkcje obsługującego je sterownika nie muszą przełączać w stan oczekiwania procesów/wątków użytkownika, które je aktywowały, gdyż urządzenie dostarcza potrzebne dane bardzo szybko. W przypadku rzeczywistych urządzeń blokowych taka konieczność może jednak wystąpić, podobnie jak konieczność reagowania na sygnały wysyłane do procesów/wątków użytkownika. W instrukcji nie będzie także opisane zagadnienie obsługi transmisji DMA, która często jest wykorzystywana w rzeczywistych urządzeniach blokowych. Kody źródłowe innych sterowników urządzeń blokowych, zarówno rzeczywistych, jak i pseudo urządzeń można znaleźć wśród kodu źródłowego jądra, w katalogu `drivers/block`. Na szczególną uwagę zasługuje kod źródłowy sterownika `null_blk`. Jest to sterownik służący do oceny wydajności opisanych wcześniej trybów obsługi operacji blokowych i może być skonfigurowanym do pracy w każdym z nich. Opis jego użycia znajduje się w katalogu z dokumentacją kodu źródłowego jądra.

3. Opis API sterowników urządzeń blokowych

Opis API sterowników urządzeń blokowych zaczniemy od najważniejszych struktur danych. Listing 1 zawiera definicję typu strukturalnego `struct gendisk`, który służy do tworzenia struktur zawierających charakterystykę urządzenia blokowego. Innymi słowy, ta struktura jest odpowiednikiem struktury typu `struct cdev` używanej w przypadku urządzeń znakowych.

Listing 1: Typ strukturalny struct gendisk

```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;
};
```

Typu `struct gendisk` jest zdefiniowany w pliku nagłówkowym o nazwie `linux/genhd.h`. Najważniejsze pola zawarte w strukturze tego typu to:

major - zawiera numer główny przypisany sterownikowi,

firstminor - zawiera pierwszy numer poboczny przypisany sterownikowi,

minors - liczba numerów pobocznych przypisanych sterownikowi,

disk_name - ciąg znaków będących nazwą sterownika (max. 31 znaków),

fops - wskaźnik na strukturę typu `struct block_device_operations`, będącą tablicą metod dla urządzeń blokowych,

queue - wskaźnik na kolejkę żądań,

private_data - wskaźnik typu `void *` na obszar pamięci zawierający dane prywatne sterownika,

flags - pole, w którym wartości poszczególnych bitów określają flagi opisujące zachowanie obsługiwanego urządzenia blokowego. Wartości tych bitów mogą być zmieniane za pomocą operatorów bitowych i np. następujących stałych:

GENHD_FL_REMOVABLE - określa urządzenie z wymiennym nośnikiem,

GENHD_FL_MEDIA_CHANGE_NOTIFY - sterownik będzie generował powiadomienie o zmianie nośnika w urządzeniu,

GENHD_FL_SUPPRESS_PARTITION_INFO - sterownik nie będzie przekazywał do systemu plików `procfs` informacji o partycjach umieszczonych na urządzeniu.

Istnieją jeszcze inne stałe związane z tym polem, ale nie będą one opisywane w tej instrukcji.

Listing 2: Typ strukturalny `struct block_device_operations`

```
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 zawiera definicję typu strukturalnego `struct block_device_operations`. Określa on struktury, które są odpowiednikami struktur typu `struct file_operations`, ale przeznaczonymi ściśle dla urządzeń blokowych. Warto zwrócić uwagę, że wśród wskaźników na funkcję będących polami struktury typu `block_device_operations` nie ma wymienionych `write()` i `read()`. To dlatego, że te metody nie są używane przez urządzenia blokowe. Programista piszący sterownik urządzenia blokowego nie musi definiować wszystkich metod, które mogą wskazywać pola opisywanej struktury. W najprostszym przypadku może ograniczyć się do przypisania jednie wartości polu `owner`. Tą wartością jest wskaźnik na strukturę typu `struct module` zwracany przez makro `THIS_MODULE`. Jeśli urządzenie blokowej jest bardziej złożone w obsłudze, to może się okazać konieczne zdefiniowanie pozostałych metod. Oto opis zadań, które wykonują niektóre z nich:

`int (*open) (struct block_device *, fmode_t)` - metoda wskazywana przez ten wskaźnik ma takie samo znaczenie, jak w przypadku urządzeń znakowych, czyli może odpowiadać za inicjacje struktury danych sterownika, włączenie urządzenia blokowego i inne czynności przygotowujące urządzenie do działania,

`void (*release) (struct gendisk *, fmode_t)` - metoda wskazywana przez ten wskaźnik ma takie samo znaczenie, jak w przypadku urządzeń znakowych, czyli odpowiada za czynności finalizujące obsługę urządzenia blokowego,

`int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long)` - podobny wskaźnik istnieje także dla urządzeń znakowych; zadaniem tej metody, która jest przez niego wskazywana jest wykonywanie operacji na urządzeniu blokowym, które nie są typowymi operacjami na pliku urządzenia.

`int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long)` - metoda wskazywana przez ten wskaźnik pełni taką samą rolę, jak `ioctl()`, ale przeznaczona jest dla 32-bitowych aplikacji wykonywanych na 64-bitowym systemie operacyjnym,

`long (*direct_access)(struct block_device *, sector_t, void __pmem **, unsigned long *pfn)` metoda wskazywana przez ten wskaźnik jest wywoływana wtedy, gdy aplikacja użytkownika zażąda bezpośredniego dostępu do danych na nośniku urządzenia, tzn. za pośrednictwem pliku urządzenia i z pominięciem systemu plików osadzonego na urządzeniu blokowym,

`unsigned int (*check_events) (struct gendisk *disk, unsigned int clearing);` - metoda wskazywana przez ten wskaźnik związana jest z obsługą zdarzeń dotyczących obsługiwanego przez sterownik urządzenia blokowego; między innymi zastępuje metodę `media_changed()`, która sprawdzała, czy nośnik w urządzeniu o wymiennym nośniku (np. DVD lub CD) zostały zmieniony,

`int (*revalidate_disk) (struct gendisk *)` - metoda wskazywana przez ten wskaźnik jest wywoływana przez jądra systemu wtedy, gdy zostanie odnotowana zmiana nośnika w urządzeniu blokowym o wymiennym nośniku,

`int (*getgeo)(struct block_device *, struct hd_geometry *)` - metoda wskazywana przez ten wskaźnik jest uruchamiana z poziomu wywołania systemowego `ioctl()`. Zwraca ona przez swój drugi parametr informacje o „geometrii” urządzenia blokowego, czyli liczbie jego głowic, cylindrów i sektorów. Zdefiniowanie tej funkcji jest konieczne, jeśli nośnik urządzenia ma być dzielony na partycje.

Listing 3 zawiera definicję typu `struct request`, czyli określającego strukturę opisującą żądanie wejścia-wyjścia.

Listing 3: Typ strukturalny `struct request`

```
1  struct request {
2          struct list_head queuelist;
3          union {
4                  struct call_single_data csd;
5                  unsigned long fifo_time;
6          };
7  
8          struct request_queue *q;
9          struct blk_mq_ctx *mq_ctx;
10 
11         u64 cmd_flags;
12         unsigned cmd_type;
13         unsigned long atomic_flags;
14 
15         int cpu;
16 
17         /* the following two fields are internal, NEVER access directly */
18         unsigned int __data_len;        /* total data len */
19         sector_t __sector;              /* sector cursor */
20 
21         struct bio *bio;
22         struct bio *biotail;
23 
24         /*
25          * The hash is used inside the scheduler, and killed once the
26          * request reaches the dispatch list. The ipi_list is only used
27          * to queue the request for softirq completion, which is long
28          * after the request has been unhashed (and even removed from
29          * the dispatch list).
30          */
31         union {
32                 struct hlist_node hash; /* merge hash */
33                 struct list_head ipi_list;
34         };
35 
36         /*
```

```
37          * The rb_node is only used inside the io scheduler, requests
38          * are pruned when moved to the dispatch queue. So let the
39          * completion_data share space with the rb_node.
40          */
41         union {
42                 struct rb_node rb_node; /* sort/lookup */
43                 void *completion_data;
44         };
45 
46         /*
47          * Three pointers are available for the IO schedulers, if they need
48          * more they have to dynamically allocate it.  Flush requests are
49          * never put on the IO scheduler. So let the flush fields share
50          * space with the elevator data.
51          */
52         union {
53                 struct {
54                         struct io_cq            *icq;
55                         void                    *priv[2];
56                 } elv;
57 
58                 struct {
59                         unsigned int            seq;
60                         struct list_head        list;
61                         rq_end_io_fn            *saved_end_io;
62                 } flush;
63         };
64 
65         struct gendisk *rq_disk;
66         struct hd_struct *part;
67         unsigned long start_time;
68 #ifdef CONFIG_BLK_CGROUP
69         struct request_list *rl;                /* rl this rq is alloced from */
70         unsigned long long start_time_ns;
71         unsigned long long io_start_time_ns;    /* when passed to hardware */
72 #endif
73         /* Number of scatter-gather DMA addr+len pairs after
74          * physical address coalescing is performed.
75          */
76         unsigned short nr_phys_segments;
77 #if defined(CONFIG_BLK_DEV_INTEGRITY)
78         unsigned short nr_integrity_segments;
79 #endif
80 
81         unsigned short ioprio;
82 
83         void *special;          /* opaque pointer available for LLD use */
84 
85         int tag;
86         int errors;
87 
88         /*
89          * when request is used as a packet command carrier
90          */
91         unsigned char __cmd[BLK_MAX_CDB];
92         unsigned char *cmd;
93         unsigned short cmd_len;
94 
95         unsigned int extra_len; /* length of alignment and padding */
96         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.
         */
        rq_end_io_fn *end_io;
        void *end_io_data;

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

Większość z tych pól nie jest obsługiwana bezpośrednio przez sterowniki urządzeń blokowych, ale warto zwrócić uwagę na następujące pola:

`q` - pole pozwalające umieścić strukturę w kolejce żądań,

`cmd_type` - pole to określa rodzaj żądania, które opisuje struktura typu `struct request`; może być odczytywane bezpośrednio; jeśli żądanie związane jest z wykonaniem operacji wejścia-wyjścia, to wartość tego pola jest równa stałej `REQ_TYPE_FS`; struktura typu `struct request` może także opisywać żądania, które nie są bezpośrednio związane operacjami na systemie plików, jak np. żądania związane z obsługą urządzeń z interfejsem SCSI,

`bio` - pole wskazujące na pierwszą strukturę `bio` wchodzącą w skład żądania,

`biotail` - pole wskazujące ostatnią strukturę `bio` wchodzącą w skład żądania; pozostałe struktury, znajdujące się między tymi wskazywanymi przez oba opisywane pola, są połączone w kolejkę,

`ioprio` - pole określające priorytet żądania,

`next_rq` - pole wskazujące na następną strukturę typu `struct request` w kolejce żądań.

Proszę także zauważyć, że niektóre pola z tej struktury są wykorzystywane przez planistów I/O.

Listing 4 zawiera z kolei definicję typu strukturalnego `struct bio`, który opisuje wspomniane wcześniej struktury `bio`.

Listing 4: Typ strukturalny `struct bio`

```
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];
};
```

Podobnie, jak w przypadku struktury typu `struct request` większość z tych pól nie jest obsługiwana bezpośrednio przez sterownik urządzenia blokowego, ale ważnymi polami z punktu widzenia programisty tworzącego taki sterownik są:

`bi_rw` - pole, którego starsze bity określają priorytet operacji, a młodsze kierunek (zapis lub odczyt),

`bi_iter` - pole zawierające informacje pozwalające iterować po tablicy segmentów zawartej w strukturze `bio`,

`bi_vnct` - pole określające liczbę segmentów związanych ze strukturą `bio`,

`bi_io_vec` - tablica segmentów związanych z daną strukturą `bio`.

Wspomniana wcześniej tablica segmentów składa się z elementów typu `struct bio_vec`, którego definicję zawiera listing 5.

Listing 5: Typ strukturalny `struct bio_vec`

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

Znaczenia poszczególnych pól tej struktury są następujące:

`bv_page` - pole zawiera adres struktury określającej ramkę, w której znajduje się strona zawierająca bufor dla operacji wejścia-wyjścia oraz segment będący częścią tego bufora; pole nie zawiera adresu wirtualnego strony, gdyż może ona należeć do pamięci wysokiej,

`bv_len` - pole określa rozmiar segmentu,

`bv_offset` - pole określa przesunięcie względem początku strony, od którego zaczyna się segment.

Listing 6 zawiera definicję typu strukturalnego `bvec_iter`, który określa struktury przechowujące dane związane z iterowaniem po tablicy segmentów należącej do struktury `bio`. Tego typu jest pole `bi_iter` w tej strukturze.

Listing 6: Typ strukturalny `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 */
};
```

Pola tej struktury mają następujące znaczenie:

`bi_sector` - numer początkowego sektora, którego dotyczy wykonywana operacja blokowa,

`bi_size` - liczba operacji wejścia-wyjścia, które pozostały do wykonania,

`bi_idx` - indeks elementu tablicy segmentów, który jest bieżąco przetwarzany,

`bi_bvec_done` - liczba przetworzonych bajtów w bieżącym segmencie.

Funkcje i makra używane przez sterowniki urządzeń blokowych zadeklarowane lub zdefiniowane są w następujących plikach nagłówkowych: `linux/genhd.h`, `linux/bio.h`, `linux/bkldev.h` i `linux/fs.h`.

Niezależnie od tego, czy sterownik obsługuje urządzenie blokowe w trybie bezkolejkowym, czy z użyciem pojedynczej kolejki żądań, to korzysta on z następujących makr i funkcji zdefiniowanych w plikach nagłówkowych:

`int register_blkdev(unsigned int major, const char *name)` - funkcja, która rejestruje nowe urządzenie blokowe, rezerwując mu numer główny. Jako pierwszy argument wywołania przyjmuje ona numer główny, który programista chce zarezerwować. Jeśli ten argument ma wartość `0`, to funkcja zarezerwuje pierwszy wolny numer główny i zwróci jego wartość. Jest to liczba naturalna z zakresu od `1` do `255`. Jako drugi argument funkcja przyjmuje ciąg znaków będący unikatową nazwą urządzenia blokowego. Jeśli rejestracja się nie powiedzie, to funkcja zwraca liczbę ujemną lub zero.

`void unregister_blkdev(unsigned int, const char *)` - funkcja wyrejestrowuje urządzenie blokowe. Jako argumenty wywołania przyjmuje numer główny przydzielony urządzeniu i unikatową nazwę urządzenia.

struct gendisk *alloc_disk(int minors) - funkcja alokuje pamięć na strukturę typu `struct gendisk`. Jako argument wywołania przyjmuje ona liczbę numerów pobocznych, jaką będzie posługiwał się sterownik. Funkcja zwraca adres utworzonej struktury typu `struct gendisk` lub wartość `NULL` w przypadku niepowodzenia. W przypadku, gdy urządzenie podzielone jest na partycje, to dla każdej partycji tworzony jest osobna taka struktura.

void add_disk(struct gendisk *disk) - funkcja, która rejestruje wypełnioną wcześniej strukturę typu `struct gendisk` w systemie. W efekcie tej rejestracji powstają odpowiednie katalogi i pliki w systemie plików `sysfs` i wysyłane są odpowiednie powiadomienia do przestrzeni użytkownika, w odpowiedzi na które `udev` tworzy pliki urządzeń blokowych dla sterownika. W przypadku, kiedy urządzenie jest podzielone na partycje, to dla każdej partycji należy zarejestrować taką strukturę.

void del_gendisk(struct gendisk *disk) - funkcja wyrejestrowuje z systemu strukturę typu `struct gendisk`, której adres zostanie jej przekazany jako argument wywołania. Skutkiem jej wywołania jest wysłanie powiadomień do przestrzeni użytkownika, które nakazują `udev` usunięcie pliku urządzenia blokowego oraz usunięcie odpowiednich plików i katalogów z systemu plików `sysfs`.

void put_disk(struct gendisk *disk) - funkcja ta zmniejsza wartość licznika odwołań do struktury `struct gendisk`, której adres został jej przekazany jako argument wywołania. Jeśli ten licznik się wyzeruje, to funkcja zwolni pamięć przeznaczoną na tę strukturę.

void set_capacity(struct gendisk *disk, sector_t size) - funkcja `inline`, która pozwala ustawić pojemność urządzenia blokowego. Jako pierwszy argument wywołania przyjmuje ona adres struktury typu `struct gendisk` związanej z urządzeniem blokowym, a jako drugi rozmiar tego urządzenia wyrażony liczbą sektorów.

sector_t get_capacity(struct gendisk *disk) - funkcja `inline`, która pozwala odczytać pojemność urządzenia blokowego wyrażoną w sektorach. Jako argument wywołania przyjmuje ona adres struktury typu `struct gendisk`.

Do obsługi struktur `bio` zarówno w trybie bezkolejkowym, jak i z pojedynczą kolejką żądań, przydatne są następujące funkcje i makra:

bio_end_sector(bio) - makro, które zwraca numer ostatniego sektora, którego dotyczy operacja blokowa opisywana przez strukturę `bio` przekazaną mu jako argument.

void bio_endio(struct bio *bio) - funkcja, która sygnalizuje, że operacja opisywana przez strukturę `bio` zakończyła się powodzeniem.

void bio_io_error(struct bio *bio) - funkcja, która sygnalizuje, że operacja opisywana przez strukturę `bio` zakończyła się niepowodzeniem.

bio_data_dir(bio) - makro, które zwraca liczbę informującą, czy operacja opisywana przez przekazana mu przez argument strukturę `bio` jest operacją zapisu, czy odczytu. W pierwszym przypadku ta liczba jest równa stałej `WRITE`, a w drugim `READ`.

bio_for_each_segment(bvl, bio, iter) - makro pozwalające iterować po segmentach związanych ze strukturą `bio`. Przyjmuje ono trzy argumenty. Pierwszym jest struktura typu `bio_vec`, do której będą zapisywane wartości kolejnych elementów tablicy `bi_io_vec`. Drugim argumentem jest struktura `bio`, do której należy wspomniana tablica. Trzecim argumentem jest struktura typu `struct bvec_iter`.

struct request_queue *blk_alloc_queue(gfp_t gfp_mask) - funkcja, która przydziela pamięć na kolejkę żądań. Ta pamięć musi zastać zarezerwowana przez sterownik, nawet jeśli pracuje on w trybie bezkolejkowym. Jako argument wywołania ta funkcja przyjmuje znacznik typu związany z przydziałem pamięci.

void blk_cleanup_queue(struct request_queue *q) - funkcja, która zwalnia pamięć przydzieloną na kolejkę żądań, której adres jest jej przekazany jako argument wywołania.

W trybie obsługi urządzenia blokowego z jedną kolejką pomocne są następujące makra i funkcje:

struct request_queue *blk_init_queue(request_fn_proc *, spinlock_t *) - funkcja, która przydziela pamięć na kolejkę żądań i ją inicjuje. Zwraca ona adres utworzonej kolejki lub `NULL` w przypadku niepowodzenia. Przydzieloną kolejkę można zwolnić za pomocą wywołania funkcji `blk_cleanup_queue()`. Opisywana funkcja przyjmuje dwa argumenty wywołania. Pierwszym jest adres funkcji, która będzie zdejmowała z kolejki i przetwarzała żądania. Drugim jest adres rygla pętlowego, który będzie chronił tę kolejkę przed współbieżnym dostępem. Funkcja przetwarzająca żądania musi mieć następujący prototyp:

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

Do jej implementacji pomocne będą funkcje opisane dalej.

struct request *blk_fetch_request(struct request_queue *q) - funkcja, zdejmuje z czoła kolejki żądań pojedyncze żądanie. Zwraca adres struktury typu `struct request` opisującej to żądanie lub `NULL` jeśli kolejka jest pusta.

rq_data_dir(rq) - makro, którego wartość określa, czy operacja, której dotyczy żądanie jest operacją zapisu, czy odczytu. Zwraca ono takie same wartości jak `bio_data_dir`.

rq_for_each_segment(bvl, _rq, _iter) - makro, które służy do iterowania po segmentach związanych z danym żądaniem. Takie same argumenty jak `bio_for_each_segment`, za wyjątkiem środkowego. Tym argumentem jest, w przypadku tego makra, struktura typu `struct request`.

void blk_complete_request(struct request *req) - funkcja, która sygnalizuje zakończenie przetwarzania żądania. Przyjmuje ona jako argument wywołania wskaźnik na strukturę typu `struct request`, która opisuje to żądanie.

W trybie obsługi bezkolejkowej konieczne jest jeszcze wywołania następującej funkcji:

void blk_queue_make_request(struct request_queue *, make_request_fn *) - funkcja ta rejestruje funkcję, która będzie przetwarzała struktury `bio` trafiające do sterownika. Jako pierwszy argument wywołana przyjmuje ona adres kolejki żądań, na którą pamięć została przydzielona przy pomocy funkcji `blk_alloc_queue()`. Jako drugi argument wywołania przyjmuje ona wskaźnik na funkcję przetwarzającą struktury `bio`, która musi posiadać następujący prototyp:

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

Funkcja ta powinna zwracać jako swą wartość stałą `BLK_QC_T_NONE`.

Opisane wcześniej podprogramy nie wyczerpują całego spektrum funkcji i makr, które związane są z obsługą operacji blokowych przez sterowniki urządzeń. Stanowią one jednak niezbędne minimum do tego, aby zaimplementować działający sterownik.

4. Przykład

Listing 7 zawiera kod źródłowy prostego sterownika, który obsługuje RAM-dysk w trybie bezkolejkowym.

Listing 7: Sterownik pseudo urządzenia blokowego

```
1  #include<linux/module.h>
2  #include<linux/genhd.h>
3  #include<linux/vmalloc.h>
4  #include<linux/fs.h>
5  #include<linux/bio.h>
6  #include<linux/blkdev.h>
7
8  #define DEVICE_SIZE 4*1024*1024
9
10 static int sector_size = 512;
11 static int major = 0;
12 static struct sbd_struct
```

```
13 {
14         struct gendisk *gd;
15         void *memory;
16 } sbd_dev;
17
18 static inline int transfer_single_bio(struct bio *bio)
19 {
20         struct bvec_iter iter;
21         struct bio_vec vector;
22         sector_t sector = bio->bi_iter.bi_sector;
23         bool wirte = bio_data_dir(bio) == WRITE;
24
25         bio_for_each_segment(vector,bio,iter) {
26                 unsigned int len = vector.bv_len;
27                 void *addr = kmap(vector.bv_page);
28                 if(wirte)
29                         memcpy(sbd_dev.memory+sector*sector_size,addr+vector.bv_offset,len);
30                 else
31                         memcpy(addr+vector.bv_offset,sbd_dev.memory+sector*sector_size,len);
32                 kunmap(addr);
33                 sector += len >> 9;
34         }
35         return 0;
36 }
37
38 static blk_qc_t make_request(struct request_queue *q, struct bio *bio)
39 {
40         int result=0;
41
42         if(bio_end_sector(bio)>get_capacity(bio->bi_bdev->bd_disk))
43                 goto mrerr0;
44
45         result = transfer_single_bio(bio);
46         if(unlikely(result!=0))
47                 goto mrerr0;
48
49         bio_endio(bio);
50         return BLK_QC_T_NONE;
51 mrerr0:
52         bio_io_error(bio);
53         return BLK_QC_T_NONE;
54 }
55
56 static struct block_device_operations block_methods = {
57         .owner = THIS_MODULE
58 };
59
60
61 static int __init sbd_constructor(void)
62 {
63         sbd_dev.memory = vmalloc(DEVICE_SIZE);
64         if(!sbd_dev.memory) {
65                 pr_alert("Memory allocation error!\n");
66                 goto ier1;
67         }
68         sbd_dev.gd = alloc_disk(1);
69         if(!sbd_dev.gd) {
70                 pr_alert("General disk structure allocation error!\n");
71                 goto ier2;
72         }
```

```c
73          major = register_blkdev(major,"sbd");
74          if(major<=0) {
75                  pr_alert("Major number allocation error!\n");
76                  goto ier3;
77          }
78          pr_info("[sbd] Major number allocated: %d.\n",major);
79          sbd_dev.gd->major = major;
80          sbd_dev.gd->first_minor = 0;
81          sbd_dev.gd->fops = &block_methods;
82          sbd_dev.gd->private_data = NULL;
83          sbd_dev.gd->flags|=GENHD_FL_SUPPRESS_PARTITION_INFO;
84          strcpy(sbd_dev.gd->disk_name,"sbd");
85          set_capacity(sbd_dev.gd,(DEVICE_SIZE)>>9);
86          sbd_dev.gd->queue = blk_alloc_queue(GFP_KERNEL);
87          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_info("[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 pseudo block device.");
120 MODULE_VERSION("1.0");
```

Wiersze 1-6 zawierają instrukcje preprocesora, które włączają do kodu sterownika pliki nagłówkowe. Większość z nich została wymieniona wcześniej w instrukcji. Plik nagłówkowy o nazwie `module.h` zawiera podprogramy związane z obsługą modułów jądra, a plik `vmalloc.h` został dołączony ze względu na zadeklarowane w nim funkcje pozwalające przydzielić i zwolnić ciągły obszar pamięci wirtualnej. Wiersz nr 8 zawiera definicję stałej określającej pojemność RAM-dysku w bajtach. Wynosi ona 4 MiB. W wierszu nr 10 została zadeklarowana i zainicjowana zmienna określająca wielkość pojedynczego sektora, czyli 512 B. Wiersz nr 11 zawiera deklarację zmiennej, w której będzie przechowywany numer główny. Jej początkowa wartość będzie wynosiła `0`, gdyż chcemy, aby funkcja `register_blkdev()` przydzieliła sterownikowi pierwszy wolny numer główny. Wiersze 12-16 zawierają definicje typu strukturalnego, a dodatkowo wiersz nr 16 zawiera deklarację struktury tego typu. Jest to prywatna struktura sterownika.

Zawiera ona dwa pola wskaźnikowe. W polu `gd` przechowywany będzie adres struktury typu `struct gendisk`. RAM-dysk będzie miał jedną partycję, stąd wystarczy tylko jedna taka struktura. Drugie pole, o nazwie `memory` będzie zawierało adres obszaru pamięci, który będzie stanowił „nośnik" RAM-dysku.

Wiersze 18-36 zawierają definicję funkcji `transfer_single_bio()`, która zgodnie ze swoją nazwą, odpowiedzialna jest za wykonanie operacji blokowej opisanej strukturą `bio`, której adres otrzymuje przez swój parametr. W wierszach 20-23 zawarte są deklaracje zmiennych lokalnych tej funkcji. Wiersz nr 20 zawiera deklarację zmiennej `iter`, która wykorzystywana jest przez makro `bio_for_each_segment`. Z kolei wiersz nr 21 zawiera deklarację zmiennej `vector`, która będzie zawierała informacje o bieżąco przetwarzanym segmencie należącym do struktury `bio`. Wiersz nr 22 zawiera deklarację zmiennej `sector`, która będzie zawierała numer sektora początkowego obszaru nośnika, który na zostać zapisany lub odczytany. Jest ona inicjowana numerem sektor początkowego dla całej struktury `bio`. W wierszu nr 23 zadeklarowana jest zmienna typu `bool`, która będzie określała rodzaj realizowanej operacji, czyli zapis lub odczyt. Przypisywana jest jej wartość wyrażenia, które porównuje wynik makra `bio_data_dir` zastosowanego dla przetwarzanej struktury `bio` ze stałą `WIRTE`. Jeśli ta wartość będzie wynosiła `true`, to będzie realizowana operacja zapisu, a jeśli `false`, to odczytu. Wiersze 25-34 zawierają pętlę zrealizowaną przy pomocy makra `bio_for_each_segment`, która iteruje po wszystkich segmentach związanych ze struktura `bio` i je przetwarza. Wspomniane makro zapisuje informację na temat bieżącego segmentu w zmiennej `vector`. W wierszu nr 16 zadeklarowana jest zmienna `len`, której przypisywana jest wielkość sektora, wyrażoną w bajtach. W wierszu nr 27 zadeklarowana jest zmienna `addr`, której przypisywany jest adres wirtualny strony, na której znajduje się przetwarzany segment. Ponieważ ta strona może należeć do pamięci wysokiej, to adres ten uzyskiwany jest ze struktury typu `struct page` za pomocą funkcji `kmap()`, a po przetworzeniu segmentu jest on zwalniany za pomocą funkcji `kunmap()`. W wierszu nr 28 funkcja sprawdza, czy ma być wykonana operacja odczytu, czy operacja zapisu i w zależności od wyniku kopiuje dane z segmentu na nośnik lub odwrotnie, przy użyciu funkcji `memcpy()`. Ilość tych danych jest określona wartością zmiennej `len`. Adres początku segmentu jest wyznaczany jako suma adresu początku strony i przesunięcia zapisanego w polu `bv_offset` struktury opisującej bieżąco przetwarzany segmentu. Początek fragmentu obszaru w pamięci będącej nośnikiem RAM-dysku, na którym ma być przeprowadzona operacja blokowa, wyznaczany jest jako suma adresu początku „nośnika" i sektora początkowego pomnożonego przez wielkość pojedynczego sektora. Po zakończeniu odpowiedniego kopiowania numer sektora początkowego jest zwiększany o liczbę przekopiowanych bajtów podzielną przez rozmiar pojedynczego sektora (wiersz nr 33). Jest to konieczne, aby prawidłowo zaadresować kolejny fragment „nośnika", który będzie powiązany z kolejnym przetwarzanym segmentem. Ponieważ w tym przypadku liczba bajtów jest dzielona bez reszty przez `512`, to celem zwiększenia jej wydajności stosowany jest operator bitowego przesunięcia w prawo, zamiast zwykłego operatora dzielenia ($512 = 2^9$). Funkcja zwraca wartość `0` i kończy swe działanie w wierszu nr 35.

Wiersze 38-54 zawierają definicję funkcji `make_request()`, która przetwarza struktury `bio` otrzymywane przez sterownik z Warstwy Operacji Blokowych. W wierszu nr 40 zadeklarowana jest zmienna lokalna tej funkcji, która będzie przechowywała wynik wykonania funkcji `transfer_single_bio()`. Jej wartość początkowa ustalana jest na `0`. W wierszu nr 42 sprawdzane jest, czy operacja opisywana przez strukturę `bio` nie wykracza poza rozmiar urządzenia blokowego. Sprawdzenie to polega na odczytaniu numeru sektora końcowego zaangażowanego w tę operację i porównaniu go z pojemnością tego urządzenia, wyrażoną w sektorach i odczytaną ze struktury typu `struct gendisk` wskazywanej pośrednio przez jedno z pól struktury `bio`. Jeśli wynik tego sprawdzenia byłby pozytywny, to sterowanie jest przekazywane do wierszy 51-53, gdzie wywoływana jest funkcja sygnalizująca błąd przetwarzania struktury `bio` (wiersz nr 52) i funkcja `make_request()` kończy swe działanie zwracając odpowiednią wartość. Jeśli jednak warunek z wiersza nr 42 nie jest spełniony, to wywoływana jest funkcja `transfer_single_bio()`, której wynik jest zapisywany do zmiennej `result`. Następnie w wierszu nr 46 sprawdzane jest, czy ten wynik jest różny od zera. Jeśli tak by było, to sterowanie jest przenoszone do wierszy 51-53 funkcji. Makro `unlikely` używane jest do oznaczania warunków, dla których jest małe prawdopodobieństwo, że będą spełnione. W opisywany przypadku, ze względu na obecną konstrukcję funkcji `transfer_single_bio()` jest to w ogóle niemożliwe. Jeśli warunek z wiersza nr 46 nie jest spełniony, to funkcja wywołuje funkcję sygnalizującą pomyślne zakończenie przetwarzania struktury `bio` (wiersz nr 49) i kończy swe działanie zwracając odpowiednią wartość.

Wiersze 56-58 zawierają deklarację i kod inicjujący zmienną o nazwie `block_methods` typu strukturalnego `block_device_operations`. Ponieważ nie potrzebujemy definicji żadnej z metod wskazywanych przez tę strukturę, to sterownik inicjuje jedynie pole `owner` tej struktury.

Wiersze 61-101 zawierają konstruktor, czyli funkcję inicjującą działanie sterownika. W wierszu nr 63 przydzielany jest obszar pamięci, który będzie „nośnikiem" obsługiwanego urządzenia blokowego. Adres początkowy tego obszaru jest zapisywany w polu `memory` struktury `sbd_dev`. Ponieważ ten obszar pamięci nie musi być fizycznie ciągły, wystarczy aby był wirtualnie ciągły, to do jego przydziału jest używana funkcja `vmalloc()`. Rozmiar tego obszaru wyznacza stała `DEVICE_SIZE`. Jeśli przydział pamięci na „nośnik" zakończy się niepowodzeniem, to konstruktor zakończy swe działanie umieszczając komunikat o wyjątku w buforze jądra i sygnalizując niepowodzenie (wiersz 65 i 66). W wierszu nr 68 przydzielana jest pamięć na strukturę typu `struct gen_disk`. Ponieważ urządzenie będzie miało tylko jedną partycję, to przydzielana jako argument wywołania funkcji `alloc_disk()` przekazywana jest liczba `1`, określająca, że będzie potrzebny tylko jeden numer poboczny. Jeśli ten przydział się nie powiedzie, to konstruktor umieści w buforze jądra odpowiedni komunikat i przekaże sterowanie do kodu, który zwolni wcześniej przydzielone zasoby. Wierszu nr 73 pozyskiwany jest numer główny dla urządzenia i zapisywany w zmiennej o nazwie `major`. Ponownie, jeśli przydział tego numeru by się nie powiódł to funkcja przekazuje sterowanie do kodu zwalniającego wcześniej przydzielone zasoby. W wierszu nr 78 funkcja umieszcza komunikat o przydzielonym numerze głównym w buforze jądra. W wierszach 79-91 inicjowane są odpowiednie pola struktury typu `struc gen_disk`. W polu `major` tej struktury zapisywany jest numer główny (wiersz nr 79), a w polu `first_minor` zapisywany jest pierwszy i jedyny numer poboczny, którego wartość wynosi `0` (wiersz nr 80). Do pola `fops` zapisywany jest adres struktury `block_methods` zaiwierającej wskaźniki na metody wywoływane dla pliku urządzenia blokowego. W wierszu nr 82 inicjowany jest wskaźnik na obszar pamięci przechowujący prywatne dane sterownika. Ponieważ taki obszar nie będzie wykorzystywany przez ten sterownik, to temu wskaźnikowi przypisywana jest wartość `NULL`. W polu `flags` ustawiana jest flaga, która wyłącza umieszczanie informacji o partycjach w odpowiednim pliku w systemie `procfs` (wiersz nr 83). W wierszu nr 84 polu `disk_name` przypisywany jest adres łańcuch znaków będącego nazwą urządzenia. Jest to `"sbd"`, czyli skrót od angielskich wyrazów *simple block device* - proste urządzenie blokowe. W wierszu nr 85 określają jest pojemność obsługiwanego urządzenia. Jest to wielkość wyrażona w sektorach, stąd jako drugi argument wywołania funkcji `set_capacity()` przekazywana jest wartość wyrażenia dzielącego rozmiar RAM-dysku przez rozmiar pojedynczego sektora. Następnie, w wierszu nr 86 przydzielana jest pamięć na kolejkę żądań, której adres jest zapisywany w polu `queue`. Jeśli ten przydział się nie powiedzie, to sterownik umieści odpowiedni komunikat w buforze jądra i przekaże sterownie do kodu zwalniającego wcześniej przydzielone zasoby. Do wywołania funkcji `blk_alloc_queue()` przekazywany jest znacznik `GFP_KERNEL`, co oznacza, że będzie wykonywany zwykły przydział pamięci w przestrzeni jądra. Jeśli się on nie powiedzie, to konstruktor umieści w buforze jądra odpowiedni komunikat i przekaże sterowanie do kodu zwalniającego wcześniej przydzielone zasoby. W wierszu nr 91 konstruktor rejestruje funkcję `make_request()` jako tę, która będzie realizowała przetwarzanie struktur `bio`, a następnie umieszcza w buforze jądra komunikat, że struktura typu `struct gen_disk` została zainicjowana. W wierszu nr 93 ta struktura dodawana jest do systemu i konstruktor kończy swe działania zwracając wartość `0`. Wiersze 95-102 zawierają kod zwalniający zasoby, jeśli nie powiodą się przydziały innych zasobów. W kolejności są zwalniane: numer główny, pamięć na strukturę typu `struct gen_disk` i pamięć będąca „nośnikiem" urządzenia blokowego. Ostatnia instrukcja wykonywana w ramach tego kodu, to zakończenie działania konstruktora kodem wyjątku wskazującym na problemy z przydziałem pamięci (wiersz nr 102).

Wiersze 105-112 zawierają destruktor, czyli funkcję finalizującą działanie sterownika. W wierszu nr 107 wyrejestrowywana jest struktura typu `struct gen_disk`. W wierszu nr 108 jest zwalniana pamięć przeznaczona na kolejkę żądań. W wierszu nr 109 zwalniany jest numer główny, a w wierszu nr 110 usuwana jest struktura typu `struct gen_disk`. Na koniec, w wierszu nr 111 zwalniana jest pamięć będąca „nośnikiem" urządzenia.

5. Konfiguracja `Udev`

Listing 8 zawiera treść pliku konfiguracyjnego o nazwie `42-sbd.rules` dla demona `udevd`, który należy umieścić w katalogu `/etc/udev/rules.d/`. Dzięki niemu użytkownik o nazwie `pi`[2] będzie mógł wykonywać na pliku urządzenia, bez użycia polecenia `sudo` takie operacje jak tworzenie systemu plików, czy podział na partycje. Sposób konstruowania treści takiego pliku został opisany w instrukcji dotyczącej sterowników urządzeń znakowych.

[2]Plik został przygotowany dla dystrybucji Raspbian przeznaczonej dla komputera Rasbperry Pi.

Listing 8: Plik konfiguracyjny dla demona udevd

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

6. Obsługa urządzenia

Obsługa urządzeń blokowych z przestrzeni użytkownika jest bardziej skomplikowana, niż obsługa urządzeń znakowych. Po załadowaniu sterownika poleceniem `insmod` tworzony jest w katalogu (zazwyczaj) `/dev` plik urządzenia blokowego o nazwie `sbd`. Najpierw należy utworzyć system plików dla takiego urządzenia przy użyciu któregoś z poleceń `mkfs`. Najlepiej będzie w tym przypadku użyć polecenia tworzącego system plików `ext2`, gdyż jego struktury zużywają stosunkowo mało miejsca na nośniku, z uwagi na to, że nie stosuje on księgowania (ang. *journalizing*). System ten można utworzyć na urządzeniu poleceniem `mkfs.ext2 /dev/sbd`. Jeśli jest ono wykonywane z poziomu użytkownika nieuprzywilejowanego i jeśli system `udev` nie został odpowiednio skonfigurowany, to może okazać się konieczne poprzedzenie tego polecenia poleceniem `sudo`. Po założeniu na urządzeniu blokowym systemu plików możemy je zamontować, czyli udostępnić użytkownikowi w określonym katalogu. Takim katalogiem może być `/mnt` lub jeden z jego podkatalogów, jeśli takie istnieją. Montowania możemy dokonać poleceniem `mount /dev/sbd /mnt`. Jeśli jest ono wykonywane z poziomu użytkownika nieuprzywilejowanego, to trzeba będzie je poprzedzić poleceniem `sudo`. Na tak zamontowanym urządzeniu operacje takie jak zakładanie, usuwanie, odczytywanie i zapisywanie plików i katalogów może przeprowadzać jedynie użytkownik uprzywilejowany. Jeśli chcielibyśmy, aby inni użytkownicy też mogli je wykonywać, to konieczne będzie przekazanie do polecenia `mount` dodatkowych opcji, opisanych w podręczniku. Zamontowane urządzenie można zdemontować przy pomocy polecenia `umount`. W przypadku opisywanego urządzenia wywołanie tego polecenia będzie miało postać: `umount /mnt`. Ponownie, jeśli jest ono wykonywane z poziomu użytkownika nieuprzywilejowanego, to konieczne będzie poprzedzenie go poleceniem `sudo`. Demontaż urządzenia blokowego jest konieczny, aby możliwe było usunięcie modułu poleceniem `rmmod`. Warto zaznaczyć, że po zdemontowaniu urządzania, a przed usunięciem modułu dane zapisane na urządzeniu blokowym nie są tracone i można uzyskać do nich dostęp po ponownym zamontowaniu urządzenia.

Zdania

1. `[2 punkty]` Zmień kod źródłowy modułu z listingu 7 tak, aby nie korzystał on z instrukcji `goto`.
2. `[4 punktów]` Zmień kod źródłowy modułu z listingu 7, tak aby po załadowaniu do jądra tworzył on dwie partycje. Konieczne będzie utworzenie dwóch struktur typu `struct gen_disk` oraz inne zmiany.
3. `[6 punktów]` Zmień kod źródłowy modułu z listingu 7 tak, aby wykonywał on obsługę urządzenia blokowego w trybie pojedynczej kolejki żądań.
4. `[2 punkty]` Zmień kod źródłowy modułu z listingu 7 tak, aby pojemność RAM-dysku była ustalana za pomocą parametru podczas ładowania sterownika do jądra systemu.
5. `[4 punktów]` Zmień kod źródłowy modułu z listingu 7, tak aby możliwe było tworzenie partycji na urządzeniu blokowym przy użyciu polecenia `cfdisk`. Konieczne będzie utworzenie definicji metody `getgeo()` oraz inne zmiany.
6. `[6 punktów]` Zmień kod źródłowy modułu z listingu 7, tak aby zamiast ciągłego obszaru pamięci używał on jako „nośnika” urządzenia listy, której elementami będą strony pamięci. Wykorzystaj w tym celu implementację dwukierunkowej listy cyklicznej udostępnianej przez jądro systemu.