# Laborator 8 - Device drivere de tip bloc. Subsistemul de I/O.

## Obiectivele laboratorului

- dobândirea de cunoştinţe legate de funcţionarea subsistemului de I/O pe Linux respectiv Windows
- acomodarea cu structurile şi funcţiile de lucru cu dispozitive de tip bloc
- obţinerea unor deprinderi de bază de utilizare a API-ului pentru dispozitive de tip bloc prin rezolvarea exerciţiilor

## Cuvinte cheie

- dispozitive de tip block
- subsistemul de I/O
- înregistrare/deînregistrare
- `struct gendisk`
- sector
- `struct block_device_operations`
- cereri ? `struct request`
- cozi de cereri ? `struct request_queue`
- prelucrarea unei cereri
- `struct bio`, `submit_bio`
- `FILE_OBJECT`
- `IoGetDeviceObjectPointer/ObDereferenceObject`
- `IoBuildSynchronousFsdRequest`
- `IoCallDriver`

## Materiale ajutătoare

- Slide-uri de suport pentru laborator
- SO2 Reference Card

## Noţiuni generale

Dispozitivele de tip bloc se caracterizează prin accesul aleator la date organizate în blocuri de dimensiune fixă. Exemple de astfel de dispozitive sunt hard disk drive-urile, CD-ROM drive-urile, RAM disk-urile etc. Viteza dispozitivelor de tip bloc este, în general, mult mai ridicată decât a celor de tip caracter, iar performanţa acestora este, de asemenea, importantă. Acesta fiind motivul pentru care **nucleul Linux tratează diferit** cele două tipuri de dispozitive (dispune de un API specializat). **Nucleul Windows, prin intermediul I/O managerului oferă o interfaţă unificată** pentru tratarea dispozitivelor

Lucrul cu dispozitive de tip bloc este, astfel, mai complicat decât lucrul cu cele de tip caracter. Dispozitivele de tip caracter au o singură poziţie curentă, în timp ce dispozitivele de tip bloc trebuie să se poată mişca la orice poziţie din dispozitiv pentru a asigura accesul aleator la date. Pentru a simplifica lucrul cu dispozitivele de tip bloc nucleul Linux pune la dispoziţia programatorului un întreg subsistem denumit subsistemul block I/O (sau *block layer*).

Din perspectiva nucleului, cea mai mică unitate logică de adresare este **blocul**. Cu toate că dispozitivul fizic poate fi adresat la nivel de **sector**, nucleul efectuează toate operaţiile cu discuri folosind blocuri. Întrucât cea mai mică unitate de adresare fizică este sectorul, dimensiunea blocului trebuie să fie un multiplu al dimensiunii sectorului. În plus, dimensiunea blocului trebuie să fie o putere a lui 2 şi nu poate depăşi dimensiunea unei pagini. Dimensiunea blocului poate varia în funcţie de sistemul de fişiere folosit, cele mai frecvente valori fiind 512 bytes, 1 kilobyte şi 4 kilobytes.

Spre deosebire de Linux, nucleul Windows nu face distincţie între dispozitivele de tip caracter şi cele de tip bloc. Rămân astfel valabile toate noţiunile prezentate în laboratorul 5 referitor la device driverele din Windows. În plus faţă de cele spuse în laboratorul 5 există utilitarul WinObj care permite vizualizarea driverelor încărcate la un moment dat în sistem. În `\Device` se pot observa numele dispozitivelor aşa cum au fost ele înregistrate, iar în `\GLOBAL??` sunt precizate numele

symlink-urilor aferente.

# Device drivere de tip bloc în Linux

## Înregistrarea unui dispozitiv de tip bloc

Pentru înregistrare se foloseşte funcţia register_blkdev [1].

Pentru deînregistrarea unui dispozitiv de tip bloc se foloseşte funcţia unregister_blkdev.

În versiunea 2.6 a kernel-ului Linux apelul funcţiei register_blkdev este opţional. Singurele operaţii efectuate de această funcţie sunt alocarea dinamică a unui major (dacă este apelată cu valoarea `0` pentru argumentul `major`) şi crearea unei intrări în `/proc/devices`. În versiunile viitoare de kernel este posibil să fie eliminată; cu toate acestea majoritatea driverelor încă o apelează.

Ca de obicei, apelul funcţiei de înregistrare se realizează în funcţia de iniţializare a modulului, iar apelul funcţiei de deînregistrare în funcţia de ieşire a modulului. Un scenariu obişnuit este prezentat în continuare:

```
#include <linux/fs.h>

#define MY_BLOCK_MAJOR           240
#define MY_BLKDEV_NAME          "mybdev"

static int my_block_init(void)
{
        int status;

        status = register_blkdev(MY_BLOCK_MAJOR, MY_BLKDEV_NAME);
        if (status < 0) {
                printk(KERN_ERR "unable to register mybdev block device\n");
                return -EBUSY;
        }
        //...
}

static void my_block_exit(void)
{
        //...
        unregister_blkdev(MY_BLOCK_MAJOR, MY_BLKDEV_NAME);
}
```

## Înregistrarea unui disc

Cu toate că funcţia register_blkdev obţine un major, nu pune la dispoziţia sistemului un dispozitiv (disc). Pentru crearea şi utilizarea de dispozitive de tip bloc (discuri) se foloseşte o interfaţă specializată definită în linux/genhd.h.

Funcţiile utile definite în linux/genhd.h sunt cele de înregistrare/alocare a unui disc, de adăugare a acestuia în sistem şi de deînregistrare/dezalocare a discului.

Funcţia alloc_disk este folosită pentru alocarea unui disc, iar funcţia del_gendisk este utilizată pentru dezalocarea acestuia. Adăugarea discului în sistem se realizează cu ajutorul funcţiei add_disk.

Funcţiile alloc_disk şi add_disk se folosesc de obicei în funcţia de iniţializare a modulului, iar funcţia del_gendisk în funcţia de ieşire a modulului.

```
#include <linux/fs.h>
#include <linux/genhd.h>

#define MY_BLOCK_MINORS  1

static struct my_block_dev {
        struct gendisk *gd;
        //...
} dev;

static int create_block_device(struct my_block_dev *dev)
{
        dev->gd = alloc_disk(MY_BLOCK_MINORS);
        //...
        add_disk(dev->gd);
}
```

```
static int my_block_init(void)
{
        //...
        create_block_device(&dev);
}

static void delete_block_device(struct my_block_dev *dev)
{
        if (dev->gd)
                del_gendisk(dev->gd);
        //...
}

static void my_block_exit(void)
{
        delete_block_device(&dev);
        //...
}
```

Ca şi la dispozitivele de tip caracter, se recomanda folosirea unei structuri de tipul `my_block_dev` în care să se regăsească elemente importante ce descriu dispozitivul de tip bloc.

Trebuie reţinut faptul că imediat după apelul funcţiei add_disk (de fapt chiar încă din timpul apelului) discul este activ şi metodele sale pot fi apelate la orice moment de timp. Ca urmare această funcţie nu trebuie apelată înainte ca driverul să fie complet iniţializat şi gata să răspundă cererilor adresate discului înregistrat.

Se observă că structura de bază în lucrul cu dispozitive de tip bloc (discuri) este structura struct gendisk.

După un apel del_gendisk este posibil ca structura struct gendisk să continue să existe (şi operaţiile asupra dispozitivului să fie apelate în continuare), în cazul în care există utilizatori ai acesteia (s-a apelat o operaţie `open` asupra dispozitivului, dar încă nu a fost apelată operaţia `release` asociată). O soluţie este păstrarea numărului de utilizatori ai dispozitivului şi apelarea funcţiei del_gendisk numai atunci când nu există utilizatori ai acestuia.

### Structura gendisk

Structura struct gendisk reţine informaţiile referitoare la un disc. După cum s-a afirmat şi mai sus, o astfel de structură se obţine în urma apelului alloc_disk şi trebuie completată înainte de a fi transmisă funcţiei add_disk.

Structura struct gendisk are următoarele câmpuri importante:

- `major`, `first_minor`, `minors`, care descriu identificatorii folosiţi de disc; un disc trebuie să aibă cel puţin un minor; dacă discul permite operaţia de partiţionare, trebuie alocat un minor pentru fiecare partiţie posibilă
- `disk_name`, care reprezintă numele discului, aşa cum apare în `/proc/partitions` şi în `sysfs` (`/sys/block`)
- `fops`, care reprezintă operaţiile asociate discului
- `queue`, care reprezintă coada de cereri
- `capacity`, care reprezintă capacitatea discului în sectoare de 512 octeţi; se iniţializează folosind funcţia set_capacity
- `private_data`, care reprezintă un pointer către datele private

Un exemplu de completare a unei structuri struct gendisk este prezentat în continuare:

```
#include <linux/genhd.h>
#include <linux/fs.h>
#include <linux/blkdev.h>

#define NR_SECTORS                      1024

#define KERNEL_SECTOR_SIZE              512

static struct my_block_dev {
       //...
       spinlock_t lock;                /* For mutual exclusion */
       struct request_queue *queue;    /* The device request queue */
       struct gendisk *gd;             /* The gendisk structure */
       //...
} dev;

static int create_block_device(struct my_block_dev *dev)
{
        ...
        /* Initialize the gendisk structure */
        dev->gd = alloc_disk(MY_BLOCK_MINORS);
        if (!dev->gd) {
                printk (KERN_NOTICE "alloc_disk failure\n");
```

```
                return -ENOMEM;
        }

        dev->gd->major = MY_BLOCK_MAJOR;
        dev->gd->first_minor = 0;
        dev->gd->fops = &my_block_ops;
        dev->gd->queue = dev->queue;
        dev->gd->private_data = dev;
        snprintf (dev->gd->disk_name, 32, "myblock");
        set_capacity(dev->gd, NR_SECTORS);

        add_disk(dev->gd);

        return 0;
}

static int my_block_init(void)
{
        int status;
        //...
        status = create_block_device(&dev);
        if (status < 0)
                return status;
        //...
}

static void delete_block_device(struct my_block_dev *dev)
{
        if (dev->gd) {
                del_gendisk(dev->gd);
        }
        //...
}

static void my_block_exit(void)
{
        delete_block_device(&dev);
        //...
}
```

[2)]

După cum s-a precizat, kernel-ul consideră un disc ca fiind un vector de sectoare cu dimensiunea 512 octeţi. În realitate, dispozitivele pot avea altă dimensiune a sectorului. Pentru a lucra cu aceste dispozitive, trebuie informat kernel-ul asupra dimensiunii reale a unui sector, şi pentru toate operaţiile trebuie făcute conversiile necesare.

Pentru a informa kernel-ul asupra dimensiunii sectorului dispozitivului, trebuie setat un parametru al cozii de cereri, imediat după ce coada este alocată folosind funcţia blk_queue_logical_block_size. Toate cererile generate de kernel vor fi multiplu de aceasta dimensiune a sectorului şi vor fi aliniate corespunzător. Totuşi, comunicaţia între dispozitiv şi driver se va realiza în continuare în sectoare de 512 octeţi, astfel încât trebuie făcută conversia de fiecare dată (un exemplu de astfel de conversie este la apelul funcţiei set_capacity în codul de mai sus).

### Structura block_device_operations

La fel cum pentru un dispozitiv de tip caracter trebuiau completate operaţiile din struct file_operations, şi pentru un dispozitiv de tip bloc trebuie completate operaţiile din block_device_operations. Asocierea operaţiilor se realizează prin intermediul câmpului `fops` din structura struct gendisk.

Câteva din câmpurile structurii block_device_operations sunt prezentate în continuare:

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

Operaţiile `open` şi `release` sunt apelate direct din userspace de către utilitare de partiţionare, creare de sisteme de fişiere sau verificare de sisteme de fişiere. La o operaţie mount se apelează `open` direct din kernelspace, identificatorul de

fişier fiind reţinut de către kernel. Un driver de tip bloc nu poate face diferenţa între apelurile `open` din userspace şi cele din kernelspace.

Un exemplu de utilizare a celor două funcţii este prezentat în continuare:

```
#include <linux/fs.h>
#include <linux/genhd.h>

static struct my_block_dev {
        //...
        struct gendisk * gd;
        //...
} dev;

static int my_block_open(struct block_device *bdev, fmode_t mode)
{
        //...

        return 0;
}

static int my_block_release(struct gendisk *gd, fmode_t mode)
{
        //...

        return 0;
}

struct block_device_operations my_block_ops = {
        .owner = THIS_MODULE,
        .open = my_block_open,
        .release = my_block_release
};

static int create_block_device(struct my_block_dev *dev)
{
        //....
        dev->gd->fops = &my_block_ops;
        dev->gd->private_data = dev;
        //...
}
```

Este de remarcat că nu există operaţiile `read` şi `write`. Aceste operaţii sunt efectuate de funcţia `request` asociata cu coada de cereri a discului.

## Cozi de cereri

Driverele de tip bloc folosesc cozi de cereri pentru a păstra cererile block I/O care urmează să fie procesate. O coadă de cereri este reprezentată de structura struct request_queue. Coada de cereri este formată dintr-o lista dublu înlănţuită de cereri şi informaţia de control asociată. Cererile sunt adăugate la coadă de cod kernel de nivel mai înalt (de exemplu sistemele de fişiere). Cât timp coada de cereri nu este vidă, driver-ul asociat cozii va trebui să extragă prima cerere din coadă şi să o transmită dispozitivului de tip bloc asociat. Fiecare element din lista de cereri este o cerere reprezentată de tipul struct request.

Cozile de cereri implementează o interfaţă care permite utilizarea mai multor planificatoare I/O (I/O schedulers). Un planificator trebuie să sorteze cererile şi să le prezinte driver-ului intr-o ordine care să maximizeze performanţa. De asemenea planificatorul se ocupă de combinarea cererilor adiacente (care se referă la sectoare adiacente de pe disc).

### Crearea şi ştergerea cozii de cereri

O coadă de cereri este creată cu ajutorul funcţiei blk_init_queue şi este ştearsă cu ajutorul funcţiei blk_cleanup_queue. Antetele acestor funcţii se găsesc în linux/blkdev.h.

Un exemplu de folosire a acestor funcţii este următorul:

```
#include <linux/fs.h>
#include <linux/genhd.h>
#include <linux/blkdev.h>

static struct my_block_dev {
        //...
        struct request_queue *queue;
        //...
} dev;
```

```
static void my_block_request(struct request_queue *q);
//...

static int create_block_device(struct my_block_dev *dev)
{
        /* Initialize the I/O queue */
        spin_lock_init(&dev->lock);
        dev->queue = blk_init_queue(my_block_request, &dev->lock);
        if (dev->queue == NULL)
                goto out_err;
        blk_queue_logical_block_size(dev->queue, KERNEL_SECTOR_SIZE);
        dev->queue->queuedata = dev;
        //...

out_err:
        return -ENOMEM;
}

static int my_block_init(void)
{
        int status;
        //...
        status = create_block_device(&dev);
        if (status < 0)
                return status;
        //...
}

static void delete_block_device(struct block_dev *dev)
{
        //...
        if (dev->queue)
                blk_cleanup_queue(dev->queue);
}

static void my_block_exit(void)
{
        delete_block_device(&dev);
        //...
}
```

Funcţia blk_init_queue primeşte ca prim argument un pointer la funcţia de prelucrare a cererilor pentru dispozitiv (de tipul request_fn_proc). În exemplul de mai sus, funcţia este `my_block_request`. Parametrul `lock` este un spinlock (iniţializat de către driver) pe care kernel-ul îl deţine în timpul apelului funcţiei `request` pentru a asigura accesul exclusiv la coada de cereri. Acest spinlock se poate folosi şi în alte funcţii ale driver-ului, pentru a proteja accesul la date partajate cu funcţia `request`.

Ca parte a iniţializării cozii de cereri se poate configura câmpul `queuedata`, care este echivalent cu câmpul `private_data` din alte structuri.

#### Funcţii utile pentru prelucrarea cozilor de cereri

Funcţia de tipul request_fn_proc este utilizată pentru tratarea cererilor de lucru cu dispozitivul de tip bloc. Această funcţie este echivalentul funcţiilor de citire şi scriere întâlnite la dispozitivele de tip caracter. Funcţia primeşte ca argument coada de cereri asociată dispozitivului şi poate folosi diverse funcţii pentru prelucrarea cererilor din coadă.

Funcţiile utilizate pentru prelucrarea cererilor din coadă, descrise mai jos, sunt:

- blk_peek_request - obţine o referinţă la prima cerere din coadă; cererea trebuie pornită folosind blk_start_request;
- blk_start_request - extrage cererea din coadă şi o porneşte pentru procesare; în general funcţia primeşte ca referinţă un pointer la o cerere întors de blk_peek_request;
- blk_fetch_request - obţine prima cerere din coadă (folosind blk_peek_request) şi o porneşte (folosind blk_start_request);
- blk_requeue_request - pentru reinserarea cererii în coadă.

Înainte de a apela aceste funcţii, trebuie obţinut spinlock-ul asociat cozii. Dacă aceste funcţii sunt apelate în funcţia de tip request_fn_proc, spinlock-ul este deja deţinut.

### Cereri pentru dispozitive de tip bloc

O cerere pentru un dispozitiv de tip bloc este descrisă de structura struct request.

Câmpurile structurii struct request includ:

- `cmd_flags`, o serie de flag-uri printre care şi direcţia (citire sau scriere); pentru a afla direcţia se foloseşte macrodefiniţia rq_data_dir, care returnează 0 pentru o cerere de citire şi 1 pentru o cerere de scriere pe dispozitiv;
- `__sector`, primul sector al cererii de transfer, valoare exprimată ca multiplu de dimensiunea sectorului(512 bytes); dacă sectorul dispozitivului are altă dimensiune, trebuie facută conversia corespunzătoare; pentru accesarea acestui câmp se foloseşte macro-ul blk_rq_pos;
- `__data_len`, numărul total de octeţi de transferat; pentru accesarea acestui câmp se foloseşte macro-ul blk_rq_bytes;
  - în general, se vor transfera date din bio-ul curent; dimensiunea datelor se obţine cu ajutorul macro-ului blk_rq_cur_bytes ;
- `buffer`, un pointer către datele de transferat; acest pointer reprezintă o adresă virtuală din kernel şi poate fi accesată direct de către driver;
- `bio`, o listă dinamică de structuri bio care reprezintă cererea; acest câmp se accesează cu ajutorul macrodefiniţiei rq_for_each_segment; transferul datelor unei cereri se realizează fie folosind câmpul `buffer`, fie câmpul `bio` [3].

#### Crearea unei cereri

Cererile de citire/scriere sunt create de nivelurile de cod superioare subsistemului de I/O din nucleu. De obicei, subsistemul care creează cereri pentru dispozitive de tip bloc este subsistemul de gestiune a fişierelor. Subsistemul de I/O acţionează ca intermediar între subsistemul de gestiune a fişierelor şi driverul de dispozitiv de tip bloc. Principalele operaţii care intră în responsabilitatea subsistemului de I/O sunt adăugarea cererilor în coada de cereri a dispozitivului de tip bloc specific şi sortarea şi comasarea cererilor (*sorting and merging*) din considerente de performanţă.

#### Terminarea unei cereri

Când driverul a terminat de transferat toate sectoarele dintr-o cerere către/dinspre dispozitiv, trebuie să informeze subsistemul de I/O prin apelarea funcţiei blk_end_request. Dacă lock-ul aferent cozii de cereri este deja obţinut, se poate folosi funcţia __blk_end_request.

În situaţia în care driverul doreşte să încheie cererea chiar dacă nu a transferat toate sectoarele aferente acesteia, poate apela respectiv funcţiile blk_end_request_all sau __blk_end_request_all. Funcţia __blk_end_request_all se apelează dacă lock-ul aferent cozii de cereri este deja obţinut.

#### Prelucrarea cererilor

Partea centrală a unui driver de tip bloc este metoda de tip request_fn_proc. În exemplele anterioare, funcţia care satisfăcea acest rol era `my_block_request`. După cum s-a precizat şi în secţiunea Crearea şi ştergerea cozii de cereri, această funcţie este ataşată driverului prin apelarea blk_init_queue.

Această metodă este apelată atunci când kernel-ul consideră că driverul trebuie să proceseze cereri de I/O. Metoda trebuie să pornească procesarea cererilor din coada, dar nu este obligată să le şi termine, cererile putând fi terminate din alte părţi ale driverului.

Parametrul `lock`, transmis la crearea unei cozi de cereri, reprezintă un spinlock pe care kernel-ul îl deţine atunci când execută metoda `request`. Din acest motiv metoda `request` rulează în context atomic şi trebuie să respecte regulile pentru cod atomic (nu trebuie să apeleze funcţii care pot duce la sleep, etc.). Acest lock asigură şi faptul că nu vor fi adăugate în coada alte cereri pentru device în timp ce se execută metoda request.

Apelarea funcţiei de prelucrare a cozii de cereri este asincronă relativ la acţiunile oricărui proces din userspace şi nu trebuie făcute presupuneri privind procesul în contextul căruia rulează. De asemenea nu trebuie presupus că buffer-ul oferit de o cerere este din kernelspace sau userspace, orice operaţie care accesează userspace-ul fiind eronata.

În continuare este prezentată una dintre cele mai simple metode de tip request_fn_proc:

```
static void my_block_request(struct request_queue *q)
{
        struct request *rq;
        struct my_block_dev *dev = q->queuedata;

        while (1) {
                rq = blk_fetch_request(q);
                if (rq == NULL)
                        break;

                if (!blk_fs_request(rq)) {
                        printk (KERN_NOTICE "Skip non-fs request\n");
```

```
                        __blk_end_request_all(rq, -EIO);
                        continue;
                }

                /* do work */
                ...

                __blk_end_request_all(rq, 0);
        }
}
```

Funcţia `my_block_request` conţine un ciclu while de parcurgere a cererilor din coada de cereri transmisă ca argument. Operaţiile realizate în cadrul acestui ciclu sunt:

1. Se citeşte prima cerere din coada folosind blk_fetch_request. Aşa cum a fost descrisă aici, funcţia `blk_fetch_request` obţine primul element din coada de cereri şi porneşte cererea.
   - Dacă funcţia întoarce `NULL` s-a ajuns la sfârşitul cozii de cereri (nu mai este nici o cerere de prelucrat) şi se iese din funcţie.
2. Un dispozitiv de tip bloc poate primi cereri care nu transferă blocuri de date (operaţii low-level asupra discului, instrucţiuni referitoare la moduri speciale de accesare a dispozitivului). Majoritatea driverelor nu ştiu cum să trateze aceste cereri şi întorc eroare. Funcţia blk_fs_request verifică dacă o cerere este din partea sistemului de fişiere (dacă este o cerere de transfer de blocuri de date) şi în caz contrar termină cererea cu eroare.
   - Terminarea cu eroare se realizează prin apelul funcţiei __blk_end_request_all cu al doilea argument `-EIO`.
3. Se prelucrează cererea conform nevoilor dispozitivului aferent.
4. Se încheie cererea. În cazul de faţă se apelează funcţia __blk_end_request_all pentru a încheiea complet cererea. Dacă toate sectoarele cererii au fost prelucrate, se foloseşte funcţia __blk_end_request.

## Structura bio

Fiecare structură struct request reprezintă o cerere block I/O, dar poate proveni din combinarea mai multor cereri independente de la un nivel mai înalt. Sectoarele ce trebuie transferate pentru o cerere pot fi dispersate în memoria principală, dar întotdeauna corespund unui set de sectoare consecutive de pe dispozitiv. Cererea este reprezentată ca o mulţime de segmente, fiecare corespunzând unui buffer din memorie. Kernel-ul poate combina cereri care se referă la sectoare adiacente, dar nu va combina cereri de scriere cu cereri de citire într-o singură structură struct request.

O structură struct request este implementată ca o listă înlănţuită de structuri bio împreună cu informaţii care permit driver-ului să-şi retina poziţia curentă în timp ce procesează cererea.

Structura bio este o descriere low-level a unei porţiuni dintr-o cerere block I/O.

```
struct bio {
        sector_t                bi_sector;      /* device address in 512 byte
                                                   sectors */
        struct block_device     *bi_bdev;
        unsigned long           bi_rw;          /* bottom bits READ/WRITE,
                                                 * top bits priority
                                                 */
        //...
        unsigned int            bi_size;        /* residual I/O count */

        //...
        struct bio_vec          *bi_io_vec;     /* the actual vec list */
        bio_end_io_t            *bi_end_io;
        atomic_t                bi_cnt;         /* pin count */

        void                    *bi_private;
        //...
};
```

La rândul ei, structura bio conţine un vector `bi_io_vec` de tipul struct bio_vec. Acesta este format din paginile individuale din memoria fizică ce trebuie transferate. Pentru a parcurge o structura `bio`, trebuie parcurs acest vector de structuri struct bio_vec şi transferate datele din fiecare pagină fizică.

### Crearea unei structuri bio

Pentru crearea unei structuri bio se pot folosi două funcţii:

- bio_alloc - alocă spaţiu pentru o nouă structură; structura trebuie iniţializată;
- bio_clone - realizează o copie a unei structuri bio existente; structura nou obţinută este iniţializată cu valorile câmpurilor structurii clonate.

Ambele funcţii întorc o nouă structură bio.

#### Transmiterea unei structuri bio

De obicei o structură bio este creată de nivelurile superioare ale kernel-ului (de obicei sistemul de fişiere). O structură astfel creată este apoi transmisă subsistemului de I/O care adună mai multe structuri bio într-o cerere.

Pentru transmiterea unei structuri bio către driverul dispozitivului I/O asociat se foloseşte funcţia submit_bio. Funcţia primeşte ca argument o structură bio iniţializată care va fi adăugată unei cereri din coada de cereri a unui dispozitiv I/O. Din acea coadă de cereri va putea fi prelucrată de driverul dispozitivului I/O cu o funcţie specializată. Primul argument al funcţiei `submit_bio` este un flag ce specifică dacă este vorba de o operaţie de scriere (`1`) sau de citire (`0`) (acest flag se va combina cu valoarea curentă a `bio->bi_rw`).

#### Aşteptarea încheierii unei structuri bio

Transmiterea unei structuri bio unui driver are ca efect adăugarea acesteia într-o cerere din coada de cereri de unde va fi ulterior prelucrată. Astfel, în momentul în care funcţia submit_bio se întoarce nu se garantează încheierea prelucrării structurii.

Dacă se doreşte să se aştepte încheierea prelucrării unei structuri bio, va trebui folosit câmpul bi_end_io al structurii. În acest câmp se precizează funcţia care va fi apelată la încheierea prelucrării structurii bio. De obicei, funcţia se foloseşte împreună cu o structură completion. Structura completion încapsulează o coadă de aşteptare şi o condiţie de aşteptare.

Un exemplu uzual este prezent în drivers/md/md.c. Partea relevantă este prezentată în extrasul de cod de mai jos:

```
static void bi_complete(struct bio *bio, int error)
{
        complete((struct completion*)bio->bi_private);
}

/* functie de prelucrare a bio */
int sync_page_io(struct block_device *bdev, sector_t sector, int size,
                   struct page *page, int rw)
{
        struct bio *bio = bio_alloc(GFP_NOIO, 1);
        struct completion event;
        //...
        init_completion(&event);
        bio->bi_private = &event;
        bio->bi_end_io = bi_complete;
        submit_bio(rw, bio);
        wait_for_completion(&event);
        //...
}
```

După cum se poate observa, câmpul bi_private al structurii este folosit pentru a reţine pointer-ul la structura completion. Acest pointer va putea fi folosit în funcţia apelată la încheierea bio-ului la apelul complete.

#### Iniţializarea unei structuri bio

După ce o structură bio a fost alocată şi înainte de a fi transmisă, trebuie iniţializată.

Iniţializarea structurii presupune completarea câmpurilor importante. După cum s-a precizat anterior, câmpul bi_end_io este folosit pentru a preciza funcţia apelată la încheierea prelucrării structurii. Câmpul bi_private este folosit pentru a stoca date utile ce pot fi accesate în funcţia bi_end_io.

Câmpul bi_rw specifică dacă este vorba de o operaţie de scriere (`1`) sau de citire (`0`).

Tot din exemplul din drivers/md/md.c se poate observa iniţializarea celorlaltor câmpuri utile:

```
int sync_page_io(struct block_device *bdev, sector_t sector, int size,
                   struct page *page, int rw)
{
       struct bio *bio = bio_alloc(GFP_NOIO, 1);
       //...
       bio->bi_bdev = bdev;
       bio->bi_sector = sector;
       bio_add_page(bio, page, size, 0);
       //...
```

În extrasul de mai sus se specifică dispozitivul de tip bloc către care va fi trimis bio-ul, sectorul de început şi conţinutul.

Conţinutul unui bio este dat de o pagină.

**Atenţie**: Câmpul `size` al apelului bio_add_page trebuie să fie multiplu de dimensiunea sectorului dispozitivului.

O pagină poate fi alocată folosind apelul alloc_page.

#### Folosirea conţinutului unei structuri bio

Pentru folosirea conţinutului unei structuri bio, paginile de suport ale structurii trebuie mapate în spaţiul de adresă nucleu de unde vor putea fi accesate. Pentru mapare/demapare se folosesc macrourile __bio_kmap_atomic şi __bio_kunmap_atomic [4)].

Un exemplu tipic de folosire este:

```
static void my_block_transfer(struct my_block_dev *dev, size_t start, size_t len, char *buffer, int dir);


static int my_xfer_bio(struct my_block_dev *dev, struct bio *bio)
{
   int i;
   struct bio_vec *bvec;
   size_t start = bio->bi_sector * KERNEL_SECTOR_SIZE;

   /* Do each segment independently. */
   bio_for_each_segment(bvec, bio, i) {
       char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);

       /* process mapped buffer */
       my_block_transfer(dev, start, bio_cur_bytes(bio), buffer, bio_data_dir(bio) == WRITE);

       start += bio_cur_bytes(bio);
       __bio_kunmap_atomic(buffer, KM_USER0);
   }
    return 0;
}
```

Se pot stoca informaţii în buffer-ul mapat sau se pot extrage informaţii.

Următoarele câmpuri ale structurii `bio` sunt accesate fie direct, fie cu ajutorul unor macrodefiniţii:

- `bi_sector`, primul sector de transferat, exprimat în sectoare de 512 octeţi
- bio_sectors, pentru obţinerea numărului total de sectoare de transferat din cererea `bio` curentă
- bio_cur_sectors, pentru obţinerea numărului de sectoare de transferat din pagina curentă
- bio_data_dir, pentru a determina dacă este o cerere de citire (`0`) sau de scriere (`1`)
- bio_for_each_segment, pentru a parcurge vectorul de structuri struct bio_vec ce conţine paginile de memorie fizică dintr-o structură `bio`.

În cazul în care se folosesc cozile de cereri şi se doreşte prelucrarea cererilor la nivel de structură `bio`, se va folosi macrodefiniţia rq_for_each_segment în locul bio_for_each_segment. Această macrodefiniţie parcurge fiecare segment din fiecare structură `bio` a unei cereri `struct request` şi actualizează o structură struct req_iterator. Structura struct req_iterator conţine structura curentă `bio` şi indicele segmentului curent.

#### Eliberarea unei structuri bio

După ce un subsistem al nucleului foloseşte o structură bio va trebui să elibereze referinţa către aceasta. Acest lucru se realizează cu ajutorul funcţiei bio_put. Funcţia bio_put este apelată şi în exemplul anterior din drivers/md/md.c.

#### Configurarea unei cozi de cerere la nivel de bio

Cu ajutorul funcţiei blk_init_queue se putea specifica o funcţie care să fie folosită pentru prelucrarea cererilor transmise driverului. Funcţia primea ca argument coada de cereri şi realiza prelucrări la nivel de structuri request.

Dacă, din motive de flexibilitate, se doreşte specificarea unei funcţii care să realizeze prelucrări la nivel de bio, trebuie folosită funcţia blk_queue_make_request în conjuncţie cu funcţia blk_alloc_queue. Mai jos este prezentat un exemplu tipic de iniţializare a unei funcţii de prelucrare la nivel de bio:

```
// semnatura functiei de tratare
static int my_make_request(struct request_queue *q, struct bio *bio);
```

```
// ...
// crearea cozii
dev->queue = blk_alloc_queue (GFP_KERNEL);
if (dev->queue == NULL) {
        printk (KERN_ERR "cannot allocate block device queue\n");
        return -ENOMEM;
}
// inregistrarea functiei de tratare
blk_queue_make_request (dev->queue, my_make_request);
dev->queue->queuedata = dev;
```

Funcţia `my_make_request` este de tipul _make_request_fn. Un exemplu de folosire a unei astfel de funcţii se găseşte în drivers/md/raid1.c.

În cazul în care se foloseşte această metodă, nu se mai folosesc cozile de cereri, fiecare cerere fiind reprezentă de o structură `bio`. Astfel, transferul de date se reduce la parcurgerea fiecărei structuri `bio` după cum a fost prezentat mai sus şi semnalarea terminării prelucrării acesteia cu ajutorul funcţiei bio_endio.

# Lucrul cu dispozitive în kernel-ul Windows

În Windows un dispozitiv este definit printr-un nume şi două structuri importante:

- DEVICE_OBJECT reprezintă dispozitivul fizic sau logic;
- FILE_OBJECT reprezintă dispozitivul în user-mode.

## Numele dispozitivelor

Numele dispozitivelor în nucleul Windows încep cu şirul `\Device\`. Exemple sunt:

- `\Device\CdRom0`
- `\Device\Harddisk1`
- `\Device\Harddisk2\Partition3`

Dispozitivele care sunt create de un driver non-WDM (Windows Driver Model) vor trebui referite cu ajutorul unui nume MS-DOS de forma `\DosDevices\DeviceName`. Exemple de nume MS-DOS sunt:

- primul disc din sistem va fi referit de numele `\DosDevices\PhysicalDisk0`;
- portul serial are numele `\DosDevices\COM1`;
- drive-ul C are numele `\DosDevices\C:`.

Dispozitivele cu nume MS-DOS sunt referite din userspace cu un nume care începe cu `\\.\`. Astfel, pentru deschiderea primului disc din sistem se va folosi un apel de forma:

```
file = CreateFile("\\.\PhysicalDisk0",
    GENERIC READ | GENERIC WRITE,
    0,
    NULL,
    OPEN_EXISTING,
    0,
    NULL);
```

Se poate crea un alt nume pentru un dispozitiv cu ajutorul funcţiei IoCreateSymbolicLink. Astfel, pentru a crea o legătură simbolică de la `\DosDevices\So2RulesDevice` la `\Device\So2RulesDevice` se foloseşte următoarea secvenţă de cod:

```
UNICODE_STRING DeviceName;
UNICODE_STRING DosDeviceName;
NTSTATUS status;

RtlInitUnicodeString(&DeviceName, L"\Device\So2RulesDevice");
RtlInitUnicodeString(&DosDeviceName, L"\DosDevices\So2RulesDevice");
status = IoCreateSymbolicLink(&DosDeviceName, &DeviceName);
if (!NT_SUCCESS(status)) {
       /* Symbolic link creation failed.  Handle error appropriately. */
}
```

După cum se observă, numele dispozitivelor sunt şiruri Unicode.

## Crearea şi ştergerea unui dispozitiv

Un dispozitiv (sau mai bine spus un obiect dispozitiv) este creat cu ajutorul funcţiei IoCreateDevice. Funcţia întoarce în ultimul argument primit un pointer la structura DEVICE_OBJECT alocată şi iniţializată de kernel.

Un exemplu de folosire este următorul:

```
static struct PSO_RULES_EXTENSION  {
   PDEVICE_OBJECT pDevice;
 } dev;

   //...
   /* PDRIVER_OBJECT driver is initialized in the driver */
   UNICODE_STRING deviceName;
   PUNICODE_STRING symlinkName;
   NTSTATUS status;

   RtlInitUnicodeString(&deviceName, L"\Device\So2RulesDevice");
   status = IoCreateDevice(
               driver,
               sizeof(PSO_RULES_EXTENSION),
               &deviceName,
               FILE_DEVICE_DISK,
               FILE_DEVICE_SECURE_OPEN,
               FALSE,
               dev->pDevice);
   //...

   /* access from user mode using "\.\So2RulesDevice" */
   RtlInitUnicodeString(symlinkName, L"\??\So2RulesDevice");
   status = IoCreateSymbolicLink(symlinkName, &deviceName);
   //...
```

Un dispozitiv (obiect dispozitiv) este eliminat cu ajutorul funcţiei IoDeleteDevice.

## Deschiderea şi închiderea unui dispozitiv

Deschiderea unui dispozitiv înseamnă obţinerea unei referinţe la obiectul asociat dispozitivului. Acest lucru se realizează cu ajutorul funcţiei IoGetDeviceObjectPointer.

Închiderea unui dispozitiv înseamnă ştergerea referinţei către obiectul asociat dispozitivului şi se realizează cu ajutorul funcţiei generice ObDereferenceObject.

Un exemplu de folosire a funcţiilor IoGetDeviceObjectPointer şi ObDereferenceObject este următorul:

```
UNICODE_STRING deviceName;
PDEVICE_OBJECT pDevice;
PFILE_OBJECT pFile;
NTSTATUS status;

RtlInitUnicodeString(&deviceName, L"\Device\So2RulesDevice");
status = IoGetDeviceObjectPointer(
        &deviceName,
        GENERIC_READ | GENERIC_WRITE,
        &pFile,
        &pDevice);

//...

ObDereferenceObject(pFile);

//...
```

## Crearea unui IRP

Ca şi structurile bio, IRP-urile sunt create de nivelurile superioare ale nucleului (de exemplu sisteme de fişiere) şi sunt transmise apoi subsistemului de I/O.

O structură IRP poate fi creată cu ajutorul funcţiei IoBuildSynchronousFsdRequest, aşa cum se poate observa şi din exemplul de mai jos:

```
#define SECTOR_SIZE             512
#define NR_SECTORS              1

PDEVICE_OBJECT deviceObject;
CHAR buffer[NR_SECTORS  * SECTOR_SIZE];
LARGE_INTEGER offset;
PIRP irp;
```

```
KEVENT irpEvent;
IO_STATUS_BLOCK ioStatus;
NTSTATUS status;


/* init offset to first sector to be read */
/* init irpEvent */
//...

irp = IoBuildSynchronousFsdRequest(
        IRP_MJ_WRITE,    /* write IRP */
        deviceObject,
        buffer,
        NR_SECTORS  * SECTOR_SIZE,
        &offset,
        &irpEvent,
        &ioStatus);

if (irp == NULL)  {
        status = STATUS_INSUFFICIENT_RESOURCES;
        /* Handle error */
}
//...
```

### Transmiterea unui IRP

Un IRP este transmis driverului asociat dispozitivului cu ajutorul funcţiei IoCallDriver. Funcţia este asemănătoare funcţiei submit_bio din kernel-ul Linux.

### Aşteptarea încheierii unui IRP

La fel ca submit_bio, funcţia IoCallDriver doar planifică IRP-ul pentru prelucrare, nu aşteaptă execuţia sa.

Pentru aşteptarea încheierii prelucrării unui IRP se foloseşte obiectul event folosit în momentul creării IRP-ului (obiectul `event` dat ca parametru funcţiei IoBuildSynchronousFsdRequest). Funcţia de aşteptare folosită va fi KeWaitForSingleObject.

## Quiz

Pentru auto-evaluare înainte de laborator răspundeţi la întrebările de quiz.

## Exerciţii

- Folosiţi arhiva de sarcini a laboratorului.
- Punctaj total: **11 puncte**
- Punctajul maxim obţinut este de **10 puncte**. Bonusul poate recupera lipsa de activitate de la alte laboratoare.

### Linux

- Intraţi în directorul `lin/` din arhiva de sarcini a laboratorului.
- Punctaj total: **7.5 puncte**
- Recomandăm folosirea unei console de lucru prin SSH (`ssh -l root spook.local`) şi o consolă cu `netconsole` (shortcut pe `Desktop`).

1. (**1 punct**) Intraţi în directorul `1-2-3-ram-disk/`.
   - Analizaţi conţinutul fişierului `ram-disk.c`.
   - Completaţi funcţiile de iniţializare şi ieşire a modulului astfel încât să permită înregistrarea, respectiv deînregistrarea unui dispozitiv de tip bloc.
   - **Hint**:
     - Consultaţi secţiunea Înregistrarea unui dispozitiv de tip bloc.
     - Nu omiteţi să verificaţi valoarea întoarsă de funcţia de înregistrare şi, în caz de eroare, să întoarceţi cod de eroare.
     - Parcurgeţi comentariile marcate cu `TODO 1`.
   - Folosiţi macrodefiniţiile existente (`MY_BLOCK_MAJOR`, `MY_BLKDEV_NAME`).
   - Compilaţi modulul.
   - Încărcaţi modulul în kernel.

- Verificaţi că dispozitivul a fost încărcat cu succes consultând `/proc/devices/`.
- Descărcaţi modulul din kernel.
- Verificaţi că a fost deînregistrat consultând `/proc/devices`.
- Definiţi pentru macroul `MY_BLOCK_MAJOR` valoarea `1`.
- Compilaţi şi încărcaţi modulul în kernel.
- De ce eşuează încărcarea modulului?
- Restauraţi valoarea `240` pentru macroul `MY_BLOCK_MAJOR`.

2. (**1.5 puncte**) Extindeţi modulul de la pasul anterior.
   - Scopul acestui exerciţiu este să adăugaţi un disc asociat driverului.
   - Analizaţi fişierul `ram-disk.c`: macrodefiniţii, structura `struct my_block_dev`, funcţiile existente.
   - Decomentaţi apelurile de funcţii `create_block_device` şi `delete_block_device`.
   - **Hint**:
     - Consultaţi secţiunile Înregistrarea unui disc, Cozi de cereri şi Cereri pentru dispozitive de tip bloc.
     - Parcurgeţi comentariile marcate cu `TODO 2`.
   - Completaţi funcţia `my_block_request` pentru prelucrarea cozii de cereri.
     - Nu faceţi o prelucrare efectivă a cererii.
     - Afişaţi mesajul `"request received"` şi următoarele informaţii: sectorul de start, dimensiunea totală, dimensiunea datelor din bio-ul curent, direcţia de tratare.
   - **Hint**:
     - Folosiţi __blk_end_request_all pentru încheierea prelucrării cererii.
   - Inseraţi modulul în kernel.
     - Folosiţi `dmesg` pentru a observa un mesaj transmis de modul. În momentul adăugării dispozitivului se transmite o cerere către acesta.
     - Verificaţi prezenţa `/dev/myblock`.
   - Folosiţi comanda `echo "abc" > /dev/myblock` pentru a genera cereri de scriere.
   - Descărcaţi modulul din kernel.
     - Reverificaţi prezenţa `/dev/myblock`.
3. (**1.5 puncte**) Extindeţi modulul de la pasul anterior.
   - Scopul acestui exerciţiu este să creaţi un RAM disc: cererile către dispozitiv vor rezulta în citiri/scrieri într-o zonă de memorie.
     - Zona de memorie aferentă `dev->data` este alocată folosind vmalloc [5)].
     - Pentru dezalocare se foloseşte vfree.
   - Creaţi o funcţie `my_block_transfer` care să scrie/citească informaţia din cerere în/din zona de memorie.
     - Funcţia va fi apelată pentru fiecare cerere din cadrul funcţiei de prelucrarea a cozii de cereri: `my_block_request`.
     - Pentru a scrie/citi în/din zona de memorie folosiţi `memcpy`.
   - **Hints**:
     - Folosiţi câmpurile structurii request pentru determinarea informaţiei de scris/citit. O descriere a macro-urilor utile este în secţiunea Cereri pentru dispozitive de tip bloc
     - Atenţie la caţi bytes scrieţi/citiţi. Ce macro trebuie să folosiţi pentru a acest scop?
     - Informaţii utile se găsesc în exemplul de block device driver din Linux Device Drivers
     - Consultaţi secţiunea Cereri pentru dispozitive de tip bloc.
     - Parcurgeţi comentariile marcate cu `TODO 3`.
   - Pentru testare folosiţi fişierul `ram-disk-test.c`. Îl puteţi compila cu ajutorul fişierului `Makefile.test`.
   - **Hints**:
     - Pentru compilarea testului folosiţi `make -f Makefile.test`.
     - Pentru rulare folosiţi `./ram-disk-test`.
       - Există posibilitatea ca unele teste să pice din cauza nesincronizării datelor transmise (flush).
4. (**2 puncte**) Intraţi în directorul `4-5-relay-disk/`.
   - Scopul exerciţiul este să citiţi datele de pe discul `PHYSICAL_DISK_NAME` direct din kernel.
   - Analizaţi conţinutul fişierului `relay-disk.c`.
   - Urmăriţi comentariile marcate cu `TODO 4`.
   - Completaţi funcţiile `open_disk` şi `close_disk`.
   - **Hint**:
     - Folosiţi funcţiile open_bdev_exclusive şi close_bdev_exclusive. Un exemplu de utilizare găsiţi în drivers/mtd/devices/block2mtd.c.
       - Deschideţi dispozitivul read-write (`FMODE_READ | FMODE_WRITE`).
       - Ca owner folosiţi modulul curent (`THIS_MODULE`).
   - Completaţi funcţiile `send_test_bio` şi `bi_complete`.
   - **Hint**:
     - Consultaţi secţiunea Structura bio.
   - În funcţia `send_test_bio` va trebui să creaţi un nou bio pe care să-l completaţi, să-l submiteţi şi să-l aşteptaţi.
   - **Citiţi** primul sector al discului.
   - **Hints**:
     - Primul sector al discului este sectorul cu index `0`.
     - Pentru aşteptare puteţi folosi o structură completion.
     - Consultaţi secţiunea Aşteptarea încheierii unei structuri bio.
   - În funcţia `bi_complete` afişaţi primii 3 octeţi din datele citite de bio şi notificaţi procesul care aşteaptă.
   - **Hints**:

    - Folosiţi formatul `"%02x"` la `printk` pentru afişarea datelor.
    - Folosiţi macrourile `__bio_kmap_atomic`, respectiv `__bio_kunmap_atomic`.
  - Pentru testare folosiţi scriptul `test-relay-disk`.
  - **Hints**:
    - Folosiţi comanda `./test-relay-disk`.
    - Scriptul scrie `"abc"` la începutul discului `PHYSICAL_DISK_NAME`.
    - În urma rulării, modulul va afişa `61 62 63` (valorile hexazecimale corespunzătoare).
5. (**1.5 puncte**) Completaţi modulul anterior pentru scrierea unui mesaj (`BIO_WRITE_MESSAGE`) pe disc.
  - Urmăriţi comentariile marcate cu `TODO 5`.
  - Actualizaţi funcţia `send_test_bio` pentru a primi ca argument tipul operaţiei (citire sau scriere).
    - Apelaţi în `relay_init` funcţia pentru citire iar în `relay_exit` funcţia pentru scriere.
  - **Hint**:
    - Recomandăm folosirea macro-urilor `BIO_DIR_READ` şi `BIO_DIR_WRITE`.
  - În cadrul funcţiei `send_test_bio`, dacă operaţia este de scriere, completaţi buffer-ul aferent bio-ului cu mesajul `BIO_WRITE_MESSAGE`.
  - Rulaţi scriptul `test-relay-disk` pentru testare.
    - Scriptul va afişa la ieşirea standard mesajul `"read from /dev/sdb: 64 65 66"`.

#### Extra

1. Adăugaţi, în cadrul implementării ramdisk-ului (directorul `1-2-3-ram-disk`) suport pentru prelucrarea cererilor din coada de cereri la nivel de bio.
  - Urmăriţi comentariile marcate cu `TODO BONUS`.
  - Configuraţi macro-ul `USE_BIO_TRANSFER` la valoarea 1.
  - Implementaţi funcţia `my_xfer_request`.
  - **Hints**:
    - Folosiţi `rq_for_each_segment` pentru a parcurge structurile `bio` ale fiecărei `cereri` şi structurile `bio_vec` ale fiecărui `bio`.
    - Folosiţi macrourile `__bio_kmap_atomic`, respectiv `__bio_kunmap_atomic` pentru maparea paginilor fiecărui bio şi accesarea bufferelor acestuia.
    - Pentru transferul efectiv, apelaţi funcţia `my_block_transfer` implementată la exerciţiul anterior.
  - Pentru testare folosiţi fişierul de test `ram-disk-test.c`.

### Windows

- Intraţi în directorul `win/` din arhiva de sarcini a laboratorului.
- Punctaj total: **3.5 puncte**

1. (**2.5 puncte**) Intraţi în directorul `relay-disk/`.
  - Scopul exerciţiul este să citiţi datele de pe discul `PHYSICAL_DISC_NAME` direct din kernel.
  - Analizaţi conţinutul fişierului `relay-disk.c`. Observaţi conţinutul structurii `SO2_PHYSICAL_DEVICE`.
  - Urmăriţi comentariile marcate cu `TODO 1`.
  - Completaţi funcţiile `OpenPhysicalDisk` şi `ClosePhysicalDisk`.
  - **Hint**:
    - Folosiţi funcţiile IoGetDeviceObjectPointer şi ObDereferenceObject.
    - Consultaţi secţiunea `Lucrul cu dispozitive în kernel-ul Windows`.
  - Completaţi funcţia `SendTestIrp`.
  - Va trebui să creaţi un nou IRP pe care să-l completaţi, să-l submiteţi şi să-l aşteptaţi.
    - **Citiţi** primul sector al discului.
  - **Hints**:
    - Primul sector al discului este sectorul cu offset `0`.
    - Folosiţi câmpul `QuadPart` al structurii `LARGE_INTEGER` pentru iniţializarea variabilei `offset`.
    - Folosiţi funcţiile IoBuildSynchronousFsdRequest şi IoCallDriver.
    - Consultaţi secţiunile `Crearea unui IRP`, `Transmiterea unui IRP` şi `Aşteptarea încheierii unui IRP`.
    - Pentru aşteptare se foloseşte un `event`.
  - În funcţia `SendTestIrp` afişaţi primii 3 octeţi din datele citite de IRP.
  - **Hint**:
    - Folosiţi `"%02x"` pentru afişarea datelor.
  - Pentru testare folosiţi fişierul `relay-disk-test.c`. Îl puteţi compila folosind fişierul `NMakefile.test`.
  - **Hint**:
    - Pentru compilarea testului folosiţi `nmake -f NMakefile.test`.
    - După rularea testului încărcaţi şi descărcaţi modulul din kernel.
    - În urma rulării modulul va afişa `61 62 63` (valorile hexazecimale corespunzătoare).
2. (**1 punct**) Completaţi modulul anterior pentru scrierea unui mesaj (`IRP_WRITE_MESSAGE`) pe disc.
  - Urmăriţi comentariile marcate cu `TODO 2`.

- Actualizaţi funcţia `SendTestIrp` pentru a primi ca argument tipul operaţiei (citire sau scriere).
- Apelaţi în `DriverEntry` funcţia pentru citire iar în `DriverUnload` funcţia pentru scriere.
- Hint:
  - Recomandăm folosirea valorilor `IRP_MJ_READ` şi `IRP_MJ_WRITE`.
- În cadrul funcţiei `SendTestIrp`, dacă operaţia este de scriere, completaţi buffer-ul aferent IRP-ului cu mesajul `IRP_WRITE_MESSAGE`.
- Rulaţi executabilul `relay-disk-test` pentru testare.
- Executabilul va afişa ieşirea standard mesajul `"read from \\.\PhysicalDrive1: 64 65 66"`.

### Soluţii

- Soluţii exerciţii laborator 8

## Resurse utile

### Linux

1. Linux Device Drivers 3rd Edition, Chapter 16. Block Drivers
2. Linux Kernel Development, Second Edition ? Chapter 13. The Block I/O Layer
3. A simple block driver
4. The gendisk interface
5. The bio structure
6. Request Queues
7. Documentation/ block/request.txt - Struct request documentation
8. Documentation/block/biodoc.txt - Notes on the Generic Block Layer
9. drivers/block/brd.c - RAM backed block disk driver
10. Linux Block Device Architecture
11. I/O Schedulers

### Windows

1. The Windows 2000 Device Driver Book, Second Edition ? Chapter 15. Layered Drivers
2. Programming the Microsoft Windows Driver Model, Second Edition - Chapter 5. The I/O Request Packet - The "Standard Model" for IRP Processing
3. The NT I/O Manager
4. Different ways of handling IRPs
5. Attaching the Filter Device Object to the Target Device Object
6. Creating IRPs for Lower-Level Drivers

[1)] Pentru mai multe detalii despre alegerea identificatorilor *major* şi *minor*, consultaţi laboratorul 4
[2)] Despre structura struct request_queue şi operaţiile pentru prelucrarea cozilor de cereri se va discuta în secţiunea Cozi de cereri
[3)] Despre structura bio şi operaţiile asociate se va discuta în secţiunea Structura bio
[4)] Mai multe informaţii despre maparea atomică a memoriei găsiţi în Linux Device Drivers 3rd Edition, Chapter 15. Memory Mapping and DMA - The Memory Map and Struct Page
[5)] Pentru simplitate, se foloseşte funcţia vmalloc pentru alocarea vectorului în memorie; funcţia vmalloc alocă o zonă de memorie contiguă în spaţiul de adrese virtuale, iar funcţia vfree o dealocă. Mai multe despre alocarea memoriei în kernel găsiţi în Linux Device Drivers 3rd Edition, Chapter 8. Allocating memory