Laborator 4 - Device drivere în Unix

Obiectivele laboratorului

- familiarizarea cu conceptul de driver al unui dispozitiv de caracter
- înţelegerea diferitelor operaţii ce pot fi efectuate asupra dispozitivelor de tip caracter
- lucrul cu cozi de aşteptare (workqueues)

Cuvinte cheie

- device node
- major
- minor
- file operations
- file
- inode
- open/release
- read/write
- put/get user
- copy from/to user
- wait queue

Materiale ajutătoare

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

Concepte generale

În UNIX dispozitivele hardware sunt accesate de utilizator prin intermediul fişierelor speciale de tip dispozitiv (device). Aceste fişiere sunt grupate în directorul `/dev`, iar apelurile de sistem `open`, `read`, `write`, `close`, `seek`, `mmap` etc. sunt redirecţionate de sistemul de operare către device driverul asociat cu dispozitivul fizic. Device driverul este o componentă a nucleului (de obicei un modul) care interacţionează cu un dispozitiv hardware.

În lumea UNIX există două categorii de fişier de dispozitiv şi, implicit, device drivere: de tip **caracter** şi de tip **bloc**. Această împărţire este făcută după viteza, volumul şi modul de organizare a datelor ce trebuie transferate de la dispozitiv către sistem şi invers. În prima categorie intră dispozitivele lente, care gestionează un volum mic de date, iar accesul la date nu necesită operaţii de căutare (seek) frecvente. Exemple sunt dispozitive cum ar fi tastatura, mouse-ul, porturile seriale, placa de sunet, joystick-ul. În general operaţiile cu aceste dispozitive (citire, scriere) se realizează secvenţial, octet cu octet. Cea de-a doua categorie cuprinde dispozitive la care volumul de date este mare, datele sunt organizate pe blocuri, operaţiile de căutare (seek) sunt frecvente. Exemple de dispozitive ce intră în această categorie sunt hard disk-urile, cdrom-urile, ram discurile, unităţile de bandă magnetică. În cazul acestor dispozitive, citirea şi scrierea se realizează la nivel de **bloc** de date.

Pentru cele două tipuri de device drivere, nucleul Linux oferă API-uri diferite. Dacă pentru dispozitivele de tip caracter apelurile de sistem ajung **direct** la device drivere, în cazul dispozitivelor de tip bloc device driverele **nu lucrează direct** cu apelurile de sistem. În cazul dispozitivelor de tip bloc, comunicaţia între user-space şi device driverul de tip bloc este intermediată de subsistemul de gestiune a fişierelor şi de subsistemul de block device. Rolul acestor subsisteme este de a pregăti device driverului resursele necesare (buffere), de a menţine în buffer cache datele recent citite şi de a reordona operaţiile de citire şi scriere din raţiuni de performanţă.

Identificator major şi minor

În UNIX, în mod tradiţional, dispozitivele aveau asociate un identificator unic, fixat. Această tradiţie se păstrează şi în Linux, deşi este posibil ca identificatorii să se aloce dinamic (din motive de compatibilitate însă, majoritatea driverelor

folosesc încă identificatori statici). Identificatorul este format din două părţi: **major** şi **minor**. Prima parte (major) identifică tipul dispozitivului (disc IDE, disc SCSI, port serial, etc.) iar cel de al doilea identifică dispozitivul (primul disc, al doilea port serial, etc.). De cele mai multe ori, majorul identifică driverul, în timp ce minorul identifică fiecare dispozitiv fizic deservit de driver. În general un driver va avea asociat un major şi va fi responsabil de toţi minorii asociaţi cu acel major.

```
# ls -la /dev/hda? /dev/ttyS?
brw-rw----  1 root disk    3,  1 2004-09-18 14:51 /dev/hda1
brw-rw----  1 root disk    3,  2 2004-09-18 14:51 /dev/hda2
crw-rw----  1 root dialout 4, 64 2004-09-18 14:52 /dev/ttyS0
crw-rw----  1 root dialout 4, 65 2004-09-18 14:52 /dev/ttyS1
```

După cum se observă din exemplul de mai sus, informaţii pentru fişiere de tip device se pot afla folosind comanda `ls`. Fişierele speciale de tip caracter sunt identificate prin caracterul **c** în prima coloană a iesirii comenzii, iar cele de tip bloc prin caracterul **b**. În coloana 5 şi 6 a rezultatului comenzii se poate observa majorul, respectiv minorul pentru fiecare dispozitiv.

Anumiţi identificatori major sunt atribuiţi în mod static dispozitivelor (în fişierul Documentation/devices.txt din sursele kernel-ului). La alegerea identificatorului pentru un nou dispozitiv se pot folosi două metode: static (se alege un număr care pare să nu fie folosit deja) sau dinamic. În `/proc/devices` se găsesc dispozitivele încărcate, împreună cu identificatorul major.

Pentru a crea un fişier de tip dispozitiv se foloseşte comanda mknod; comanda primeşte ca argumente tipul (bloc sau caracter), majorul şi minorul dispozitivului (`mknod name type major minor`). Astfel, dacă se doreşte crearea dispozitivului de tip caracter cu numele `mycdev` cu majorul `42` şi minorul `0` se foloseşte comanda:

```
# mknod /dev/mycdev c 42 0
```

Pentru a crea dispozitivului de tip block cu numele `mybdev` cu majorul `240` şi minorul `0` comanda folosită va fi:

```
# mknod /dev/mybdev b 240 0
```

În continuare ne vom referi la drivere pentru dispozitive de tip caracter.

Structuri de date importante pentru un dispozitiv de tip caracter

În kernel, un dispozitiv de tip caracter este reprezentat de structura cdev, structură folosită la înregistrarea acestuia în sistem.

Majoritatea operaţiilor cu drivere folosesc trei structuri importante: struct file_operations, struct file şi struct inode.

Structura file_operations

După cum s-a precizat, device driverele de tip caracter primesc nealterate apelurile de sistem efectuate de utilizatori asupra fişierelor de tip dispozitiv. În consecinţă, pentru implementarea unui device driver vor trebui implementate apelurile de sistem de lucru cu fişiere: `open`, `close`, `read`, `write`, `lseek`, `mmap`, etc. Aceste operaţii sunt descrise în câmpuri ale structurii file_operations [1]:

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

struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    [...]
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    [...]
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    [...]
};
```

Se poate observa că signatura funcţiei diferă de apelul de sistem pe care îl foloseşte utilizatorul. Sistemul de operare se interpune între utilizator şi device driver, pentru a simplifica implementarea în device driver.

`open` nu primeşte ca parametru calea sau diverşii parametri care controlează modul de deschidere a fişierului. În mod similar, `read`, `write`, `release`, `ioctl`, `lseek` nu primesc ca parametru un descriptor de fişier. În schimb, aceste rutine primesc ca parametri două structuri: `file` şi `inode`. Ambele structuri reprezintă un fişier, dar din perspective diferite.

Majoritatea parametrilor pentru operaţiile prezentate au semnificaţie directă:

- `file` şi `inode` identifică fişierul de tip dispozitiv;
- `size` reprezintă numărul de octeţi ce trebuie citiţi sau scrişi;
- `offset` reprezintă offsetul de unde trebuie citit sau scris (trebuie actualizat corespunzător);
- `user_buffer` reprezintă bufferul utilizatorului[2)] din care se citeşte/în care se scrie;
- `whence` reprezintă modalitatea de seek;
- `cmd` şi `arg` sunt parametrii trimişi de utilizatori la apelul `ioctl`.

Structurile "inode" şi "file"

Un inode reprezintă un fişier din punctul de vedere al sistemului de fişiere. Atribute ale unui inode sunt dimensiunea, drepturile, timpii asociaţi fişierului. Un inode identifică în mod unic un fişier într-un sistem de fişiere[3)].

Structura file reprezintă tot un fişier, dar mai aproape de punctul de vedere al utilizatorului. Dintre atributele structurii file enumerăm: inode-ul, numele fişierului, atributele de deschidere ale fişierului, poziţia în fişier. Toate fişierele deschise la un moment dat au asociate o structură file.

Pentru a înţelege diferenţele dintre `inode` şi `file`, vom folosi o analogie din programarea orientată pe obiecte: dacă vom considera un inode o clasă, atunci file-urile sunt obiecte, adică instanţe ale clasei inode. Inode-ul reprezintă imaginea statică a fişierului (**inode-ul nu are stare**), pe când **file reprezintă imaginea dinamică** a fişierului (file-ul are stare).

Revenind la device drivere, cele două entităţi au aproape întotdeauna modalităţi standard de folosire: inode-ul se foloseşte pentru a determina majorul şi minorul device-ului asupra căruia se face operaţia, iar file-ul se foloseşte pentru a determina flag-urile cu care a fost deschis fişierul dar şi pentru a memora şi accesa mai târziu date private.

Structura file conţine, printre multe câmpuri, şi:

- `f_mode`, care specifică permisiunile pentru citire (`FMODE_READ`) sau scriere (`FMODE_WRITE`);
- `f_flags`, care specifică flag-urile de deschidere a fişierului (`O_RDONLY`, `O_NONBLOCK`, `O_SYNC`, `O_APPEND`, `O_TRUNC` etc.);
- `f_op`, care specifica operaţiile asociate fişierului (pointer către structura file_operations);
- `private_data`, un pointer care poate fi folosit de programator pentru a păstra date specifice dispozitivului; pointerul va fi iniţializat la adresa unei zone de memorie alocate de programator.
- `f_pos`, offsetul în cadrul fişierului

Structura inode conţine, printre multe informaţii, un câmp `i_cdev`, care este un pointer către structura care defineşte dispozitivul de tip caracter (atunci când inode-ul referă file-ul unui dispozitiv de tip caracter).

Implementarea operaţiilor

Pentru implementarea unui device driver se recomandă crearea unei structuri care să conţină informaţii despre dispozitivul dat, informaţii utilizate în cadrul modulului. În cazul unui driver pentru un dispozitiv de tip caracter, structura va conţine un câmp de tipul struct cdev pentru a referi dispozitivul. Exemplu de mai jos foloseşte în acest sens structura `struct my_device_data`:

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

struct my_device_data {
    struct cdev cdev;
    /* my data starts here */
    //...
};

static int my_open(struct inode *inode, struct file *file)
{
    struct my_device_data *my_data =
            container_of(inode->i_cdev, struct my_device_data, cdev);

    file->private_data = my_data;
    //...
}

static int my_read(struct file *file, char __user *user_buffer, size_t size, loff_t *offset)
{
    struct my_device_data *my_data =
             (struct my_device_data *) file->private_data;
```

```
    //...
}
```

O structură de tipul `my_device_data` va conţine datele asociate unui dispozitiv. Câmpul `cdev` (de tipul struct cdev) reprezintă un dispozitiv de tip caracter şi este folosit la înregistrarea acestuia în sistem şi identificarea dispozitivului. Pointer-ul către membrul `cdev` se poate afla cu ajutorul câmpului `i_cdev` al structurii inode (cu ajutorul macro-ului container_of). În câmpul `private_data` al structurii file se pot memora informaţii la `open` care apoi sunt disponibile în rutinele `read`, `write`, `release` etc.

Înregistrarea şi deînregistrarea dispozitivelor de tip caracter

Înregistrarea/deînregistrarea unui dispozitiv se realizează prin specificarea majorului şi minorului acestuia. Tipul `dev_t` este folosit pentru a păstra identificatorii unui dispozitiv (atât majorul, cât şi minorul) şi se poate obţine cu ajutorul macro-ului MKDEV.

Pentru alocarea şi dezalocarea statică a identificatorilor unui dispozitiv, se folosesc funcţiile register_chrdev_region, respectiv unregister_chrdev_region:

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

int register_chrdev_region(dev_t first, unsigned int count, char *name);
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Este recomandat ca identificatorii de dispozitiv să fie alocaţi dinamic cu funcţia alloc_chrdev_region.

Secvenţa de mai jos rezervă `my_minor_count` dispozitive, începând de la dispozitivul cu majorul `my_major` şi minorul `my_first_minor` (dacă se depăşeşte valoare maximă pentru minor, se trece la următorul major):

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

    //...
    int err;
    err = register_chrdev_region(MKDEV(my_major, my_first_minor), my_minor_count,
                                 "my_device_driver");
    if (err != 0) {
        /* report error */
        return err;
    }
    //...
```

[4]

După atribuirea identificatorilor, dispozitivul de tip caracter va trebui iniţializat (cdev_init) şi va trebui informat nucleul de existenţa lui (cdev_add). Funcţia cdev_add trebuie apelată doar după ce dispozitivul este pregătit sa primească apeluri. Eliminarea unui dispozitiv se realizează folosind funcţia cdev_del.

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

void cdev_init(struct cdev *cdev, struct file_operations *fops);
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
void cdev_del(struct cdev *dev);
```

Următoarea secvenţă înregistrează şi iniţializează `MY_MAX_MINORS` dispozitive:

register.c

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

#define MY_MAJOR       42
#define MY_MAX_MINORS  5

struct my_device_data {
    struct cdev cdev;
    /* my data starts here */
    //...
};

struct my_device_data devs[MY_MAX_MINORS];

const struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .read = my_read,
    .write = my_write,
```

```
        .release = my_release,
        .ioctl = my_ioctl
    };

    int init_module(void)
    {
        int i, err;

        err = register_chrdev_region(MKDEV(MY_MAJOR, 0), MY_MAX_MINORS,
                                     "my_device_driver");
        if (err != 0) {
            /* report error */
            return err;
        }

        for(i = 0; i < MY_MAX_MINORS; i++) {
            /* initialize devs[i] fields */
            cdev_init(&devs[i].cdev, &my_fops);
            cdev_add(&devs[i].cdev, MKDEV(MY_MAJOR, i), 1);
        }

        return 0;
    }
```

în vreme ce următoarea secvenţă le şterge şi deînregistrează:

```
void cleanup_module(void)
{
    int i;

    for(i = 0; i < MY_MAX_MINORS; i++) {
        /* release devs[i] fields */
        cdev_del(&devs[i].cdev);
    }
    unregister_chrdev_region(MKDEV(MY_MAJOR, 0), MY_MAX_MINORS);
}
```

Observaţie: la iniţializarea structurii `my_fops` s-a folosit iniţializarea membrilor după nume, definită în standardul C99 (vezi designated initializers şi The file_operations Structure). Membrii structurii care nu apar explicit într-o astfel de iniţializare vor fi setaţi la valoarea implicită pentru tipul lor. De exemplu, după iniţializarea de mai sus, `my_fops.mmap` va fi NULL.

Accesul la spaţiul de adresă al procesului

Un driver pentru un dispozitiv este interfaţa de comunicaţie între o aplicaţie şi hardware. Drept urmare, deseori va trebui să accesăm în cadrul unui device driver date din user-space. Accesarea spaţiului de adresă al proceselor nu se poate face, însă, direct (prin dereferenţierea unui pointer din user-space). Accesarea directă a unui pointer din user-space poate duce la un comportament incorect (în funcţie de arhitectură, un pointer din user-space poate să nu fie valid sau mapat în kernel-space), un kernel oops (pointerul din user-mode poate referi o zonă de memorie care nu este rezidentă) sau probleme de securitate. Accesarea corectă a datelor din user-space se realizează prin apelarea macro-urilor/funcţiilor de mai jos:

```
#include <asm/uaccess.h>

put_user(type val, type *address);
get_user(type val, type *address);
unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);
unsigned long copy_from_user(void *to, const void __user *from, unsigned long n)
```

Toate macro-urile/funcţiile întorc 0 în caz de succes şi altă valoare în caz de eroare şi au următoarele roluri:

- put_user pune în user-space la adresa `address` valoarea `val`; tipul poate fi unul pe 8, 16, 32, 64 de biţi (tipul maxim suportat depinde de platforma hardware);
- get_user analog cu funcţia precedentă, numai că `val` va fi setată la o valoare identică cu valoarea de la adresa user-space dată prin `address`;
- copy_to_user copiază din kernel-space de la adresa referită de `from` în user-space la adresa referită de `to`, `size` octeţi;
- copy_from_user copiază din user-space de la adresa referită de `from` în kernel-space la adresa referită de `to`, size octeţi.

O secţiune uzuală de cod care lucrează cu aceste funcţii este:

```
#include <asm/uaccess.h>

if (copy_to_user(user_buffer, kernel_buffer, size))
    return -EFAULT;
else
    return SUCCESS;
```

Operaţii implementate de device drivere de tip caracter

open şi release

În funcţia `open` se realizează operaţiile de iniţializare a unui dispozitiv [5)]. În majoritatea cazurilor, aceste operaţii se referă la iniţializarea dispozitivului şi completarea datelor specifice (în cazul în care este primul apel `open`). Funcţia `release` se ocupă de eliberarea resurselor specifice dispozitivului: se dezalocă datele specifice şi se închide dispozitivul daca este ultimul apel `close`.

În cele mai multe cazuri, funcţia `open` va avea următoarea structură:

```
static int my_open(struct inode *inode, struct file *file)
{
    struct my_device_data *my_data =
            container_of(inode->i_cdev, struct my_device_data, cdev);

    /* validate access to device */
    file->private_data = my_data;

    /* initialize device */
        //..

    return 0;
}
```

O problemă care apare la implementarea funcţiei `open` este controlul accesului. Uneori este necesar ca un dispozitiv să fie deschis o singură dată la un moment dat; mai exact, nu se permite al doilea `open` înainte de `release`. Pentru a implementa această restricţie se alege o modalitate de tratare a unui apel `open` pentru un dispozitiv deja deschis: se poate întoarce o eroare (`-EBUSY`), se pot bloca apelurile `open` pâna la o operaţie de `release` sau se poate închide dispozitivul înainte de a realiza operaţia de `open`.

La apelul din user-space al funcţiilor `open` si `close` asupra dispozitivului, se vor apela operaţiile `my_open` şi `my_release` din driver. Un exemplu de apel din user-space:

```
    int fd = open("/dev/my_device", O_RDONLY);
    if (fd < 0) {
        /* handle error */
    }

    /* do work */
    //..

    close(fd);
```

read şi write

Funcţiile `read` şi `write` transferă date între dispozitiv şi user-space: funcţia `read` citeşte datele de la dispozitiv şi le transferă în user-space, în timp ce `write` citeşte datele din user-space şi le scrie pe dispozitiv. Buffer-ul primit ca parametru reprezintă un pointer în user-space, motiv pentru care este necesară folosirea funcţiilor _copy_to_user sau copy_from_user.

Valoarea întoarsă este numărul de octeţi scrişi sau citiţi. Dacă valoarea întoarsă este mai mică decât parametrul `size` (numărul de octeţi ceruţi), înseamnă că s-a realizat un transfer parţial. De cele mai multe ori, aplicaţia din user-space apelează din nou funcţia corespunzătoare apelului de sistem (read sau write) până când se transferă numărul de date cerut.

Pentru a realiza un transfer de date format din mai multe transferuri parţiale, vor trebui realizate următoarele operaţii:

- se transferă numărul maxim de octeţi posibil între buffer-ul primit ca parametru şi dispozitiv (scrierea pe dispozitiv/citirea de pe dispozitiv se va face începând de la offset-ul primit ca parametru);
- se actualizează offset-ul primit ca parametru la poziţia de la care va începe următoarea citire / scriere a datelor;
- se întoarce numărul de octeţi transferaţi.

Secvenţa de mai jos prezintă un exemplu de apel simplu al funcţiei `read`. Apelul nu actualizează câmpul `offset` astfel că tot timpul va întoarce mesajul de la începutul buffer-ului. O implementare corectă trebuie să ţină cont de parametrul `offset` şi să-l actualizeze după citire.

```
static int my_read(struct file *file, char __user *user_buffer,
                                size_t size, loff_t *offset)
{
    struct my_device_data *my_data =
             (struct my_device_data *) file->private_data;

    /* read data from device in my_data->buffer */
    if(copy_to_user(user_buffer, my_data->buffer, my_data->size))
        return -EFAULT;

    return my_data->size;
}
```

Structura funcţiei `write` este similară: citeşte date din user-space folosind funcţia copy_from_user şi le scrie pe dispozitiv.

```
static int my_write(struct file *file, const char __user *user_buffer,
                                           size_t size, loff_t * offset)
{
    // write data from user buffer into kernel buffer
    // update file offset in userspace
    // ..

}
```

La apelul funcţiilor `read` şi `write` din user-space (folosind descriptorul de fişier obţinut în urma unui apel `open`), se vor apela operaţiile `my_read` şi `my_write` din driver. Un exemplu de cod pentru user-space:

```
    if (read(fd, buffer, size) < 0) {
        /* handle error */
    }

    if (write(fd, buffer, size) < 0) {
        /* handle error */
    }
```

ioctl

Pe lângă operaţiile de `read` şi `write`, un driver are nevoie de posibilitatea de a realiza anumite operaţii de control asupra dispozitivului fizic. Aceste operaţii se realizează prin implementarea unei funcţii de tip `ioctl`:

```
static int my_ioctl (struct inode *inode, struct file *file,
                     unsigned int cmd, unsigned long arg);
```

`cmd` reprezintă comanda transmisă din user-space. Dacă la apelul din user-space se transmite un întreg, acesta poate fi accesat direct. Dacă se trasmite un buffer, valoarea `arg` va fi un pointer către acesta şi trebuie accesat prin intermediul funcţiilor `copy_to_user` sau `copy_from_user`.

Înainte de a implementa funcţia `ioctl`, vor trebui alese numerele ce corespund comenzilor. O metodă este de a alege numere consecutive începând de la 0, dar se recomandă folosirea macrodefiniţiei `_IOC(dir, type, nr, size)`[6] pentru generarea codurilor ioctl. Parametrii macrodefiniţiei sunt după cum urmează:

- `dir` reprezintă direcţia de transfer a datelor (`_IOC_NONE`, `_IOC_READ`, `_IOC_WRITE`)[7];
- `type` reprezintă numărul magic (Documentation/ioctl-number.txt);
- `nr` este numărul codului `ioctl` specific dispozitivului;
- `size` este dimensiunea datelor transferate.

În exemplul de mai jos este prezentată o implementare pentru o funcţie `ioctl`:

```
#include <asm/ioctl.h>

#define MY_IOCTL_IN _IOC(_IOC_WRITE, 'k', 1, sizeof(my_ioctl_data))

static int my_ioctl (struct inode *inode, struct file *file,
                     unsigned int cmd, unsigned long arg)
{
    struct my_device_data *my_data =
         (struct my_device_data*) file->private_data;
    my_ioctl_data mid;

    switch(cmd) {
    case MY_IOCTL_IN:
        if( copy_from_user(&mid, (my_ioctl_data *) arg,
```

```
                            sizeof(my_ioctl_data)) )
            return -EFAULT;

        /* process data and execute command */

        break;
    default:
        return -ENOTTY;
    }

    return 0;
}
```

[8)]

La apelul din user-space a funcţiei `ioctl`, se va apela funcţia `my_ioctl` a driver-ului. Un exemplu de astfel de apel în user-space:

```
    if (ioctl(fd, MY_IOCTL_IN, buffer) < 0) {
        /* handle error */
    }
```

[9)]

Sincronizare - cozi de aşteptare

Cozile de aşteptare sunt mecanisme utile în probleme de sincronizare. De multe ori este necesar ca un thread să aştepte terminarea unei operaţii, dar este de dorit ca această aşteptare să nu fie busy-wating. Folosind cozi de aşteptare şi funcţii care schimbă starea thread-ului din planificabil în neplanificabil şi invers se pot rezolva astfel de probleme. În Linux, o coadă de aşteptare este o listă în care sunt trecute procesele care aşteaptă un anumit eveniment. O coadă de aşteptare este definită cu tipul wait_queue_head_t şi poate fi folosită de funcţiile/macro-urile:

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

DECLARE_WAIT_QUEUE_HEAD(wq_name);

void init_waitqueue_head(wait_queue_head_t *q);

int wait_event(wait_queue_head_t q, int condition);

int wait_event_interruptible(wait_queue_head_t q, int condition);

int wait_event_timeout(wait_queue_head_t q, int condition, int timeout);

int wait_event_interruptible_timeout(wait_queue_head_t q, int condition, int timeout);

void wake_up(wait_queue_head_t *q);

void wake_up_interruptible(wait_queue_head_t *q);
```

Rolurile macro-urilor/funcţiilor de mai sus sunt:

- init_waitqueue_head iniţializează coada de aşteptare; dacă se doreşte iniţializarea cozii la compilare, se poate folosi macroul DECLARE_WAIT_QUEUE_HEAD;
- wait_event şi wait_event_interruptible adaugă thread-ul curent la coada de aşteptare cât timp condiţia este falsă, îi setează starea la `TASK_UNINTERRUPTIBLE` sau `TASK_INTERRUPTIBLE` şi apelează scheduler-ul pentru planificarea unui nou thread; aşteptarea va fi întreruptă atunci când un alt thread va apela funcţia wake_up;
- wait_event_timeout şi wait_event_interruptible_timeout au acelaşi efect ca funcţiile de mai sus, doar că aşteptarea poate fi întreruptă la încheierea timeout-ului primit ca parametru;
- wake_up pune toate thread-urile oprite din starea `TASK_INTERRUPTIBLE` şi `TASK_UNINTERRUPTIBLE` în starea `TASK_RUNNING`; scoate aceste thread-uri din coada de aşteptare;
- wake_up_interruptible aceeaşi acţiune, însă se folosesc doar thread-urile cu starea `TASK_INTERRUPTIBLE`.

Un exemplu simplu este cel al unui thread care aşteaptă modificarea valorii unui flag. Iniţializările se realizează prin secvenţa:

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

```
wait_queue_head_t wq;
int flag = 0;

init_waitqueue_head(&wq);
```

Un thread va aştepta ca flag-ul sa fie modificat la o valoare diferită de zero:

```
wait_event_interruptible(wq, flag !=0);
flag = 0;
```

în timp ce un alt thread va modifica valoarea flag-ului şi va trezi thread-urile care aşteaptă:

```
flag = 1;
wake_up_interruptible(&wq);
```

Quiz

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

Exerciţii

- Folosiţi arhiva de sarcini a laboratorului.
- Pentru crearea unui modul de kernel folosiţi resursele din directorul `kernel/`.
- Pentru crearea unui modul de test folosiţi resursele din directorul `user/`.
- Task-urile vor fi rezolvate succesiv prin completarea fişierului `kernel/so2_cdev.c` cu noi funcţii.
- Urmăriţi conţinutul fişierul `kernel/so2_cdev.c` şi folosiţi macro-urile definite.
- Punctaj total: **11 puncte**.

1. (**1,5 puncte**) Înregistraţi în kernel un dispozitiv de tip caracter.
   - Driver-ul va controla un singur dispozitiv cu majorul `MY_MAJOR` şi minorul `MY_MINOR` (macro-urile definite în fişierul `kernel/so2_cdev.c`
   - Creaţi fişierul de tip caracter în `/dev/so2_cdev` folosind utilitarul `mknod`.
   - **Hints:**
     - Citiţi secţiunea Identificator major şi minor din laborator.
   - Implementaţi înregistrarea şi deînregistrarea dispozitivului cu numele `so2_cdev`
   - **Hints:**
     - Nu uitaţi că driver-ul controlează un singur dispozitiv
     - Citiţi secţiunea Înregistrarea şi deînregistrarea dispozitivelor de tip caracter din laborator.
   - Afişati un mesaj după operaţiile de înregistrare, respectiv deînregistrare.
   - Inseraţi modulul în kernel şi consultaţi `/proc/devices`.
   - Extrageţi modulul din kernel.
2. (**0,5 puncte**)
   - Modificaţi modulul de mai sus pentru a încerca înregistrarea unui dispozitiv deja alocat.
   - **Hints:**
     - Consultaţi `/proc/devices` pentru a obţine un dispozitiv deja alocat.
   - Ce eroare este întoarsă de funcţia de înregistrare?
   - **Hints:**
     - Consultaţi LXR.
   - Reveniţi la configuraţia iniţială a modulului.
3. (**1 punct**)
   - Rulaţi comanda `cat` peste dispozitivul de tip caracter creat (`/dev/so2_cdev`). De ce nu funcţionează?
   - Implementaţi funcţiile `open` şi `release` în driver şi iniţializaţi dispozitivul.
   - **Hints:**
     - Prototipul operaţiilor unui device driver se găsesc în structura file_operations.
     - Citiţi secţiunile open şi release şi Înregistrarea şi deînregistrarea dispozitivelor de tip caracter din laborator.
     - Pentru început adăgaţi în structura dispozitivului doar un câmp de tipul `struct cdev`.
   - Afişaţi un mesaj în funcţiile de tip `open` şi `release`.
   - Rulaţi comanda `cat` peste dispozitiv după inserarea modulului.
   - Urmăriţi mesajele afişate de kernel după rularea comenzii `cat`.
   - Cum explicaţi mesajul de eroare al comenzii `cat`?
4. (**0.5 puncte**) Restricţionaţi accesul la dispozitiv cu variabile atomice astfel încât un singur proces să poată deschide device-ul la un moment dat, restul primind mesaj de "Device busy" (EBUSY)
   - Pentru a testa folosiţi `cat /dev/so2_cdev & cat /dev/so2_cdev`
   - **Hints:**
     - Recomandăm folosirea atomic_cmpxchg [10]. Folosiţi o variabilă de tip `atomic_t` în cadrul structurii dispozitivului.
     - Pentru a simula o utilizare îndelungată a dispozitivului apelaţi scheduler-ul:

`set_current_state(TASK_INTERRUPTIBLE)`, urmat de `schedule_timeout(1000)`
- În cadrul funcţiei de tip `release` variabila atomică de access exclusiv la deschidere trebuie resetată.

5. (**1,5 puncte**) Implementaţi funcţia `read` în driver.
   - Ignoraţi pentru moment parametrii `size` şi `offset`.
   - Păstraţi în cadrul structurii dispozitivului un buffer pe care să îl iniţializaţi la mesajul dat de macro-ul `MESSAGE` şi al cărui conţinut să îl întoarceţi la un apel `read`.
   - Testaţi folosind `cat /dev/so2_cdev`.
   - Puteţi presupune că buffer-ul utilizatorului este suficient de mare. Nu este nevoie să verificaţi validitatea argumentului `size` al funcţiei `read`.
   - **Hints:**
     - Execuţia comenzii `cat /dev/so2_cdev` nu se termină (folosiţi `Ctrl+C`).
     - Citiţi secţiunile read şi write şi Accesul la spaţiul de adresă al procesului din laborator.
6. (**1,5 puncte**) Modificaţi driver-ul anterior în aşa fel încât rularea `cat` să se termine.
   - **Hints:**
     - `cat` citeşte până la sfârşitul fişierului.
     - Sfârşitul fişierului e semnalat prin întoarcerea valorii `0` din `read`.
     - Actualizaţi şi folosiţi offset-ul primit ca parametru în funcţia `read`.
     - Citiţi secţiunea read şi write din laborator.
7. (**1 punct**) Adăugaţi posibilitatea de a scrie un mesaj care să îl înlocuiască pe cel predefinit.
   - Implementaţi funcţia `write` în driver.
   - Ignoraţi pe moment parametrul `offset`.
   - Puteţi presupune că buffer-ul driverului este suficient de mare. Nu este nevoie să verificaţi validitatea argumentului `size` al funcţiei `write`.
   - **Hints:**
     - Prototipul operaţiilor unui device driver se găsesc în structura file_operations.
     - Testaţi folosind `echo "arpeggio" > /dev/so2_cdev` (şi apoi `cat /dev/so2_cdev`).
     - Citiţi secţiunile read şi write şi Accesul la spaţiul de adresă al procesului din laborator.
8. (**1,0 puncte**) Adăugaţi operaţia ioctl `MY_IOCTL_PRINT` care să afişeze mesajul dat de macro-ul `IOCTL_MESSAGE` din driver.
   - Implementaţi funcţia `ioctl` în driver.
   - Trebuie să scrieţi un program în user-space (`user/so2_cdev_test.c` care să apeleze funcţia `ioctl` cu parametrii corespunzători.
   - **Hints:**
     - Folosiţi `printk` pentru a afişa mesajul din driver.
     - Macrodefiniţia `MY_IOCTL_PRINT` este definită în fişierul `include/so2_cdev.h` din arhiva de sarcini a laboratorului (foloseşte `_IOC` pentru a defini operaţia)
     - Citiţi secţiunile ioctl şi open şi release din laborator.
     - În fişierul de testare trebuie să deschideţi dispozitivul de tip caracter şi să apelaţi ioctl.
9. (**1,5 puncte**) Adăugaţi două operaţii `ioctl` în device driver pentru lucrul cu cozi de aşteptare.
   - Adaugaţi funcţiei `ioctl` din driver operaţiile:
     - `MY_IOCTL_DOWN` pentru a adăuga procesul într-o coadă de aşteptare
     - `MY_IOCTL_UP` pentru a scoate procesul dintr-o coadă de aşteptare
   - Modificaţi programul din user-space pentru a permite testarea.
   - **Hints:**
     - Completaţi structura dispozitivului cu un câmp de tipul `wait_queue_t` şi un flag pentru condiţia de aşteptare.
     - Nu uitaţi să iniţializaţi coada de aşteptare şi flag-ul.
     - La adăugarea procesului în coada de aşteptare, acesta va rămâne blocat în execuţie; pentru rularea comenzii de scoatere din coadă deschideţi o nouă consolă în maşina virtuală cu `Alt+F2`; puteţi reveni la consola anterioară cu `Alt+F1`. Dacă sunteţi conectaţi prin SSH la maşina virtuală deschideţi o nouă consolă.
     - Citiţi secţiunile ioctl şi Sincronizare - cozi de aşteptare din laborator.
10. (**1,0 puncte**) Adăugaţi două operaţii `ioctl` pentru modificarea mesajului asociat driver-ului.
    - Adaugaţi funcţiei `ioctl` din driver operaţiile:
      - `MY_IOCTL_SET_BUFFER` pentru scrierea unui mesaj către dispozitiv
      - `MY_IOCTL_GET_BUFFER` pentru citirea unui mesaj de la dispozitiv
    - Modificaţi programul din user-space pentru a permite testarea.
    - **Hints:**
      - Citiţi secţiunile ioctl şi Accesul la spaţiul de adresă al procesului din laborator.
      - Folosiţi buffer de lungime fixă (`BUFFER_SIZE`)

Extra

1. Implementaţi flagul `O_NONBLOCK`
   - Modificaţi programul din userspace pentru a permite testarea
   - **Hints**:
     - Dacă nu sunt date asociate device-ului şi fişierul:
       - **a** fost deschis cu `O_NONBLOCK`, citirea din el va întoarce `-EWOULDBLOCK` [11)]

    - **nu** a fost deschis cu `O_NONBLOCK`, taskul curent va fi pus în aşteptare folosind cozi de aşteptare
  - Pentru a putea debloca operaţia de `read`, renunţaţi la condiţia de exclusivitate pusă la punctul **4**.
  - Folosiţi dimensiunea datelor pe post de condiţie de deblocare din coadă.
  - Puteţi să folosiţi coada deja declarată la task-ul anterior.
  - Puteţi ignora offsetul în cadrul fişierului.
  - Modificaţi dimensiunea iniţiala a datelor reţinute la **0**, pentru a facilita testarea.
  - Adăugaţi o nouă operaţie în userspace care:
    - schimbă flagurile de deschidere (folosind `fcntl`),
    - face o citere
  - Cu ce flaguri este deschis fişierul atunci când se rulează comanda `cat /dev/so2_dev`?

Soluţii

- Soluţii exerciţii laborator 4

Resurse utile

1. Un model de device driver de tip caracter
2. Un model de program în user-space pentru a apela device driver-ul de mai sus
3. Un exemplu de character device driver
4. Linux Device Drivers, 3rd edition ? Chapter 3. Char Drivers, Chapter 6. Advanced Char Driver Operations
5. Essential Linux Device Drivers - Chapter 5. Character Drivers
6. Character Device Drivers
7. Talking to Device Files (writes and IOCTLs)
8. Linux Device Driver Dos and Don'ts

[1] struct file_operations - aici am prezentat doar o parte din operaţiile din structură, cele care se implementează în mod uzual de către device drivere
[2] user_buffer - acest buffer trebuie accesat cu funcţii speciale (pe care le vom discuta imediat) şi nu direct, pentru că este un pointer în spaţiul de adresă al procesului ce a invocat apelul de sistem
[3] nume fisier - după cum aţi observat, la atributele inode-ului nu am enumerat şi numele. Aceasta pentru că în UNIX un fişier poate avea mai multe nume, datorită link-urilor soft sau hard
[4] my_device_driver - "my_device_driver" este numele device-ului asociat cu intervalul de identificatori şi care va apărea în `/proc/devices`
[5] driver/device - după cum s-a specificat, un driver poate gestiona mai multe device-uri; funcţia init_module realizează iniţializările specifice driver-ului, pe când funcţia open pe cele specifice device-ului pentru care se apelează funcţia
[6] IOC - alternativ, se pot folosi macrodefiniţiile _IO (pentru o comandă fără parametri), _IOR (pentru o comandă de citire), _IOW (pentru o comandă de scriere) sau _IOWR (pentru o comandă de transfer bidirecţional)
[7] dir - direcţia este precizată din punct de vedere al aplicaţiei: _IOC_READ când se citeşte de pe device, iar _IOC_WRITE când se scrie pe device
[8] my_ioctl_data - `my_ioctl_data` poate fi un tip de date (`int`, `char`, etc.) sau o structura definita anterior
[9] buffer - dacă nu se transferă date între user-space şi kernel-space, parametrul `buffer` poate lipsi
[10] `atomic_cmpxchg` întoarce **vechea** valoare a variabilei atomice
[11] `-EAGAIN`