Εργασία Δημιουργίας Character Device Driver στο Linux Kernel
Λειτουργικά Συστήματα (ECE ΓΚ702)

University of Patras, Department of Electrical and Computer Engineering

Περιεχόμενα

1 Εισαγωγή

Στην εργασία αυτή θα γίνει μία παρουσίαση της διεπαφής με την οποία μία συσκευή επικοινωνεί με το λειτουργικό σύστημα. Θα εστιάσουμε στον τρόπο όπου ένας οδηγητής συσκευής (device driver) επικοινωνεί με το υπόλοιπο σύστημα σε περιβάλλον Linux. Τέλος, θα δημιουργήσουμε στο σύστημά μας μια εικονική συσκευή για την οποία θα φτιάξουμε έναν απλό driver μέσω του οποίου θα μπορεί ένας χρήστης να αλληλεπιδράσει με αυτή.

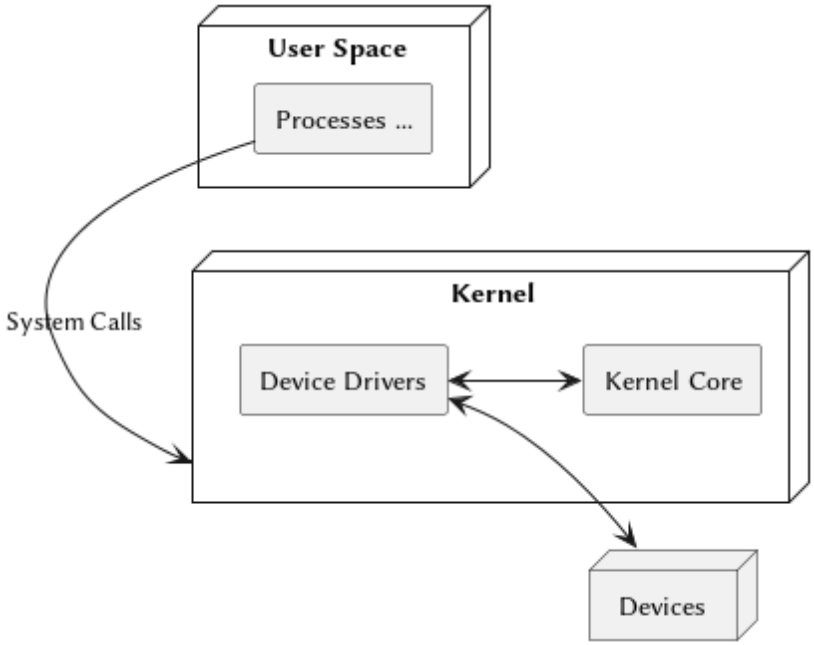
Στο λειτουργικό σύστημα UNIX, πάνω στο οποίο είναι βασισμένο το Linux υπάρχει η ιδέα του "όλα είναι ένα αρχείο". Αυτό σημαίνει πως η επικοινωνία με περιφερειακές συσκευές όπως ένα πληκτρολόγιο μπορούμε να τις μοντελοποιήσουμε στο σύστημά μας σαν ροές από bytes. Με άλλα λόγια, μπορούμε να φανταστούμε τις συσκευές σαν απλά αρχεία κειμένου, όπου το "κείμενο" σε αυτή την περίπτωση είναι τα δεδομένα της συσκευής. Προγράμματα χρήστη στη συνέχεια μπορούν να επικοινωνήσουν με τη κάθε συσκευή διαβάζοντας ή γράφοντας στο αρχείο συσκευής (device file) που της αντιστοιχεί. Η διεπαφή γίνεται μέσω κλήσεων συστήματος (system calls) όπως `open`, `read`, `write`, `close`, `lseek` κλπ. Το λειτουργικό σύστημα μεσολαβεί αυτής της επικοινωνίας εκτελώντας τις ρουτίνες που προσφέρονται από τον driver της εκάστοτε συσκευής. Ο κάθε driver είναι μέρος του kernel και μπορεί είτε να είναι ενσωματωμένος στον κώδικα του kernel είτε να φορτώνεται δυναμικά ως kernel module.

Σημείωση Μπορείτε να βρείτε περισσότερες πληροφορίες για τα system calls που είναι διαθέσιμα στο σύστημά σας διαβάζοντας τα αντίστοιχα manual pages. Χρησιμοποιήστε και την εντολή `man` (πχ `man 2 open`).

2 Αρχεία Συσκευών (Device Files)

Όπως είπαμε, κάθε φυσική συσκευή που υπάρχει στο σύστημά σας έχει ένα (ή περισσότερα) αρχείο συσκευής που της αντιστοιχεί. Μέσω αυτού μπορούν διεργασίες στο σύστημα να αλληλεπιδράσουμε με τη συσκευή. Σε UNIX-like συστήματα, οι συσκευές βρίσκονται συνήθως στον φάκελο `/dev`. Αυτό είναι μια αυθαίρετη επιλογή και υπάρχει για λόγους οργάνωσης.

Επισκεφθείτε τον φάκελο `/dev` και δείτε τα αρχεία συσκευών που υπάρχουν εκεί. Μπορείτε να τα δείτε χρησιμοποιώντας και την παρακάτω εντολή.



Σχήμα 1: Η επικοινωνία εφαρμογών χρήστη με συσκευές μέσω λειτουργικού συστήματος και των drivers.

Command Line

```
$ ls -la /dev/sd? /dev/random
crw-rw-rw- 1 root root 1,  8 Dec  3 14:05 /dev/random
brw-rw---- 1 root disk 8,  0 Dec  3 14:05 /dev/sda
brw-rw---- 1 root disk 8, 16 Dec  3 14:05 /dev/sdb
brw-rw---- 1 root disk 8, 32 Dec  3 14:05 /dev/sdc
```

Η πρώτη στήλη της εξόδου της εντολής μας δείχνει τον τύπο της συσκευής: `c` για character device και `b` για block device. Στις στήλες 5 και 6 φαίνονται οι major και minor identifiers αντίστοιχα.

Σημείωση Χωρίς να μπούμε σε πολλές λεπτομέρειες, η διαφορά μεταξύ ενός character και ενός block device είναι ο τρόπος με τον οποίο μπορούμε να έχουμε πρόσβαση στα δεδομένα της συσκευής.

- Σε ένα character device όπως ένα ποντίκι, η επικοινωνία γίνεται σε μία συνεχή ροή δεδομένων.
- Σε ένα block device όπως ένας σκληρός δίσκος, η επικοινωνία γίνεται μέσω cache και μπορεί να υπάρχει random access στα δεδομένα.

Στην εργασία αυτή θα ασχοληθούμε με τη δημιουργία ενός **character device driver**.

Μπορείτε να μπείτε εδώ για περαιτέρω μελέτη.

Οι συσκευές `sda`, `sdb`, `sdc` αντιστοιχούν στους σκληρούς δίσκους που είναι συνδεδεμένοι στο σύστημα. Η συσκευή `random` αντιστοιχεί στη γεννήτρια τυχαίων αριθμών του επεξεργαστή.

Στο σύστημά σας μπορείτε να αλληλεπιδράσετε με τις περιφερειακές σας συσκευές μέσω του συστήματος αρχείων (file system).

Δοκιμάστε να εκτελέσετε την παρακάτω εντολή:

Command Line

```
$ cat /dev/random
# random bytes ...
```

Με την παραπάνω εντολή "ζητήσατε" στο λειτουργικό σας σύστημα να ενεργοποιήσει την γεννήτρια τυχαίων αριθμών του επεξεργαστή σας και να σας επιστρέψει αυτά τα τυχαία bytes. Ενδιάμεσα αυτής της επικοινωνίας υπήρξε ένας driver ο οποίος μετέφρασε την έξοδο του επεξεργαστή σε μία μορφή που μπορεί να καταλάβει και να χρησιμοποιήσει το υπόλοιπό σας σύστημα.

3 Καταγράφοντας μια Συσκευή (Device Registering) Major & Minor Αριθμοί

Στο λειτουργικό σύστημα Linux μπορείτε χειροκίνητα να καταγράψετε ένα νέο device file στο σύστημά σας. Αυτό γίνεται μέσω της εντολής `mknod`.

Για να φτιάξετε ένα character device file μπορείτε να εκτελέσετε την εξής εντολή:

Command Line

```
$ sudo mknod /dev/mydevice c 42 0
```

Το σύμβολο `c` σημαίνει πως δημιουργούμε ένα **character** device file. Για ένα block device file θα χρησιμοποιούσαμε το σύμβολο `b`.

Τα νούμερα 42 και 0 είναι ο major και ο minor identifier αντίστοιχα του device file. Το major αριθμός παραπέμπει στον τύπο της συσκευής (π.χ MIDI controller, joystick, ποντίκι, serial port κ.α) ενώ ο minor αντιπροσωπεύει την ίδια τη συσκευή (π.χ πρώτος σκληρός δίσκος, δεύτερη οθόνη κ.α).

Ένας driver αντιστοιχεί σε ένα συγκεκριμένο major και είναι υπεύθυνος για όλες τις συσκευές που έχουν αυτό το major.

Σημείωση Μπορείτε εδώ να βρείτε επίσημο documentation πάνω στους αριθμούς major που αντιστοιχούν στον εκάστοτε τύπο περιφερειακής συσκευής. Ο αριθμός 42 τον οποίο χρησιμοποιούμε έχει δεσμευτεί για χρήση σε παραδείγματα. Συνεπώς, δεν πρόκειται κατά τη δημιουργία του driver μας να έχουμε σύγκρουση με τον major identifier μιας άλλης συσκευής.

4 Character Device στο Kernel

4.1 Πρώτα Βήματα

4.1.1 Βασικές Δομές

Στο kernel μία character device αναπαρίσταται από ένα `struct cdev`. Οι περισσότερες ενέργειες ενός driver χρησιμοποιούν τις δομές `struct file_operations`, `struct file` και `struct inode`.

Προσοχή Η δομή `struct file` στο kernel δεν έχει σχέση με τη δομή `FILE` που έχουμε δει σε user-space προγράμματα και επιστρέφεται από συναρτήσεις της βασικής βιβλιοθήκης της C όπως η `fopen`. Η δομή `file` αναπαριστά ένα ανοικτό αρχείο και εμφανίζεται μόνο σε kernel κώδικα.

4.1.2 Δομή `struct file_operations`

Η δομή `file_operations` έχει ως μέλη δείκτες σε συναρτήσεις (function pointers) ο καθένας από τους οποίους αντιστοιχεί σε ένα system call και η συνάρτηση στην οποία "δείχνει" εκτελείται όταν μία διεργασία εκτελέσει ένα system call πάνω στη συσκευή μας. Εμείς οφείλουμε, κατά τη συγγραφή του driver να ορίσουμε τις συναρτήσεις αυτές έτσι ώστε να γεφυρώσουμε μέσω του λειτουργικού συστήματος την επικοινωνία με τη συσκευή.

file operations structure

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

struct file_operations {
    struct module *owner;
    // [...]
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);

    ssize_t (*write) (struct file*, const char __user*, size_t, loff_t*);

    // [...]
    int (*open) (struct inode *, struct file *);
    // [...]
    int (*release) (struct inode *, struct file *);
    // [...]
```

Ο τρόπος με τον οποίο δημιουργούμε μια νέα συσκευή βασίζεται σε έννοιες αντικειμενοστραφούς προγραμματισμού. Φανταστείτε πως υπάρχει μία abstract κλάση `file_operations` από την οποία ο κάθε driver κληρονομεί και έπειτα ορίζει την εκάστοτε μέθοδο (`read`, `write`, ...).

Στον κώδικά σας θα δημιουργήσετε μία δομή `struct file_operations` σαν μεταβλητή. Εκεί, θα ορίσετε ως member variables τις συναρτήσεις που φτιάξατε για τη συσκευή σας.

file operations structure use

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

static int my_init(void)
{
    /* ... */

/* When initialising your device, pass in the my_fops variable. */
    cdev_init(&devs[i].cdev, &my_fops);

    /* ... */

}
```

Έτσι τελικά το λειτουργικό σύστημα, όταν μια διεργασία αλληλεπιδράσει με τη συσκευή μας, θα εκτελέσει τις συναρτήσεις που ορίσαμε.

4.1.3 Καταγράφοντας (Registering) τη Συσκευή

Η καταγραφή της συσκευής γίνεται ορίζοντας το major και minor identifier.

Η καταγραφή αυτή γίνεται με τη συνάρτηση `register_chrdev_region` και Στο τέλος της εκτέλεσης του module θα πρέπει να γίνει απ-εγγραφή της συσκευής με τη συνάρτηση `unregister_chrdev_region`.

register/unregister device

```
static int my_init(void)
{
    int err;
    // register the device
    err = register_chrdev_region(MKDEV(MY_MAJOR, MY_MINOR),
            NUM_MINORS, MODULE_NAME);

    if (err != 0) {
        pr_info("Registered!");
        return err;
    }

    // ...

    return 0;
}

static void my_exit(void)
{
    // ..

    // unregister the device
    unregister_chrdev_region(MKDEV(MY_MAJOR, MY_MINOR),
            NUM_MINORS);
}
```

4.1.4 Δεδομένα Συσκευής

Όταν φτιάχνετε έναν driver πιθανότατα θα χρειάζεται να αποθηκεύσετε δεδομένα τα οποία θα αφορούν τη συσκευή. Για αυτό, καλό είναι να οριστεί ένα struct στο οποίο θα υπάρχουν τα εξής member variables:

- Μια δομή `struct cdev` η οποία θα αντιστοιχεί στο συγκεκριμένο character device.
- Οποιαδήποτε άλλα δεδομένα αφορούν τη συσκευή.

Η δομή `my_device_data` θα περιέχει τα δεδομένα που σχετίζονται με μία συσκευή. Το μέλος `cdev` είναι η αναπαράσταση ενός character device στο kernel.

Σημείωση Το όνομα `my_device_data` το επιλέξαμε αυθαίρετα.

Όταν κάποια διεργασία εκτελέσει μια κλήση συστήματος (`open`, `read`, ...) στη συσκευή μας, το λειτουργικό σύστημα θα εκτελέσει την αντίστοιχη ρουτίνα που έχουμε ορίσει στον driver μας.

Αυτή η ρουτίνα, όταν εκτελεστεί, αυτό που "βλέπει" σαν είσοδο είναι ένας `file*` που αντιστοιχεί στη συσκευή μας, και όχι ένα `my_device_data*`.

Συνεπώς, πρέπει με κάποιον τρόπο να λάβουμε το `struct my_device_data` που αντιστοιχεί στη συσκευή.

Αυτό θα γίνει λαμβάνοντας τη δομή `struct my_device_data` μέσα από το `struct inode*` το οποίο αρχικά θα λάβουμε στη ρουτίνα `my_open` μέσω του macro `container_of`. Έπειτα, θέτουμε το μέλος `file->private_data` ώστε να δείχνει στη θέση μνήμης της δομής `my_data`. Το πεδίο `private_data` του `struct file` χρησιμοποιείται για να διατηρούμε πληροφορίες κατάστασης μεταξύ των system calls.

Η διαδικασία αυτή φαίνεται ως εξής:

get device data from inode->cdev

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

struct my_device_data {
    struct cdev cdev;
    /* device data */
    //...
};

static int my_open(struct inode *inode, struct file *file)
{
    struct my_device_data *my_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;

    my_data = (struct my_device_data *) file->private_data;

    //...
}
```

Σημείωση Η χρήση του macro `container_of` για να πάρουμε πρόσβαση στο `my_device_data` μπορεί να φαίνεται παράξενη. Εδώ είναι ένα αξιόλογο άρθρο που προσπαθεί να εξηγήσει τη λειτουργία του.

4.1.5 Αρχικοποίηση (Initialization) Συσκευής

Εφόσον έχουμε ορίσει τα major και minor identifiers, το character device πρέπει να αρχικοποιηθεί. Πρέπει, δηλαδή, να αρχικοποιήσουμε τη δομή `struct cdev` που έχουμε ορίσει στη δομή `my_device_data` Αυτό γίνεται χρησιμοποιώντας τις συναρτήσεις `cdev_init`, `cdev_add`, `cdev_del`. Αυτές οι συναρτήσεις ορίζονται στο αρχείο `linux/include/linux/cdev.h`.

Τελικά, κατά την από-φόρτωση του module πρέπει να επιστρέψουμε στο σύστημα όλους τους πόρους που έχουμε λάβει. Μεταξύ αυτών πρέπει να διαγράψουμε και να από-εγγράψουμε (unregister) τα character devices που αρχικοποιήσαμε χρησιμοποιώντας τις συναρτήσεις `cdev_del` και `unregister_chrdev_region`.

Ο τρόπος με τον οποίο μπορείτε να κάνετε register και έπειτα release τον driver σας φαίνεται παρακάτω:

device init/del

```
#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];

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;
}

void exit_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);
}
```

4.2 Επικοινωνία με μία διεργασία

Φυσικά, ο driver μας δεν θα ήταν ιδιαίτερα χρήσιμος αν δεν μπορούσε με κάποιο τρόπο να ανταλλάξει δεδομένα με τις υπόλοιπες διεργασίες του συστήματος. Για αυτό το λόγο πρέπει να έχουμε πρόσβαση σε δεδομένα που βρίσκονται σε user-space. Δεν μπορούμε όμως απλά να χρησιμοποιήσουμε pointers που δείχνουν σε θέση μνήμης του user-space, εφόσον αυτή μπορεί να μην βρίσκεται στη φυσική μνήμη τη στιγμή που τη χρησιμοποιούμε (virtual memory) αλλά και για λόγους ασφαλείας. Για αυτό το λόγο υπάρχει μια

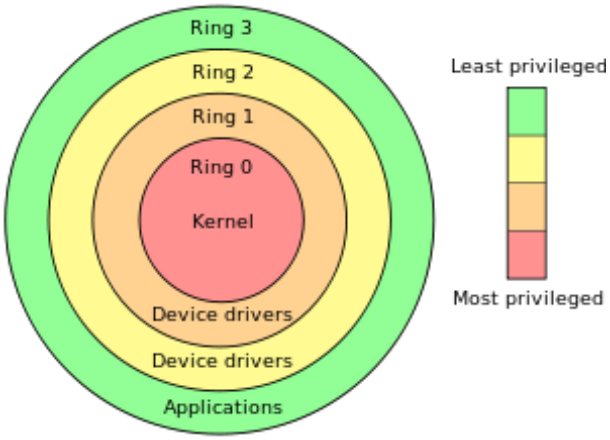
οικογένεια συναρτήσεων και macros τα οποία μας επιτρέπουν μέσα από το kernel να αλληλεπιδράμε με τη μνήμη εφαρμογών χρήστη.

move data from/to userspace

```
#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);
```

Μπορούμε να φανταστούμε πως το user-space και το kernel-space είναι χωρισμένα μεταξύ τους. Μέσω των παραπάνω συναρτήσεων μπορούμε να γεφυρώσουμε τα δεδομένα μεταξύ των δύο.

Σε ένα λειτουργικό σύστημα υπάρχουν πολλά επίπεδα ασφαλείας, από το επίπεδο στο οποίο είναι ο πυρήνας και υπάρχουν πλήρεις δικαιοδοσίες μέχρι στο επίπεδο χρήστη όπου τρέχουν οι εφαρμογές μας με μειωμένες δικαιοδοσίες.



Σχήμα 2: Τα διάφορα επίπεδα δικαιωμάτων σε ένα λειτουργικό σύστημα.

4.3 Open/Release στον Device Driver

Η συνάρτηση `my_open` αντιστοιχεί στην αρχικοποίηση της συσκευής.

Αντίστοιχα, στη συνάρτηση `release` αποδεσμεύουμε πόρους της συσκευής και τελικά την κλείνουμε.

Ενδεχομένως για μία συσκευή να πρέπει να απαγορεύσουμε ταυτόχρονη χρήση της από πολλές διεργασίες. Σε αυτή την περίπτωση θα πρέπει η συνάρτηση `my_open` να επιστρέψει τον κωδικό σφάλματος `-EBUSY`, ενημερώνοντας έτσι τον χρήστη ο οποίος κάλεσε το system call `open` πως δεν μπορεί να χρησιμοποιήσει τη συσκευή.

open/close device file from userspace (c)

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

// ...

close(fd);
```

4.4 Read/Write στον Device Driver

Όταν μία διεργασία καλέσει τα read/write system calls για το αρχείο το οποίο αντιστοιχεί στη συσκευή μας, το λειτουργικό σύστημα θα καλέσει αντίστοιχα τις ρουτίνες `my_read`/`my_write`.

Η κλήση τους σε ένα πρόγραμμα C από το user-space θα φαίνεται ως εξής:

read/write from/to device from userspace (c)

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

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

- Το read system call μεταφέρει δεδομένα από τη συσκευή σε έναν buffer που βρίσκεται σε user space. Αυτά μπορεί να είναι δεδομένα όπως το ποιο πλήκτρο πατήθηκε αν μιλάμε για έναν driver πληκτρολογίου.
- Το write system call παίρνει δεδομένα τα οποία βρίσκονται σε user-space και τα μεταφέρει στη μνήμη του kernel ώστε να τα χρησιμοποιήσει ο driver της συσκευής. Αυτά τα δεδομένα μπορεί να είναι κάποιο string που θέλουμε να εμφανιστεί στη κονσόλα. Η συνάρτηση `printf` της βασικής βιβλιοθήκης της C, από μέσα της χρησιμοποιεί το write system call για να εμφανίσει κείμενο στο command line.

Και στις δύο περιπτώσεις, εφόσον η συσκευή μας θα έχει επικοινωνία με μία διεργασία χρήστη, πρέπει τα δεδομένα να τα μεταφέρουμε και να τα λάβουμε χρησιμοποιώντας τις συναρτήσεις `copy_to_user` και `copy_from_user`.

Οι ρουτίνες που θα δημιουργήσουμε στον driver μας (`my_read`, `my_write`) θα επιστρέφουν ένα νούμερο.

- Αν επιστέψουν θετικό αριθμό, θα είναι ο αριθμός των bytes που μεταφέρθηκαν.
- Αν επιστρέψουν 0, αυτό σημαίνει πως έχουν μεταφερθεί όλα τα δεδομένα.
- Αν επιστρέψουν αρνητικό αριθμό, αυτό σηματοδοτεί κάποιο σφάλμα.

Πολλές φορές, η επικοινωνία με μία συσκευή δεν αποτελείται από μία μεγάλη μεταφορά δεδομένων αλλά από πολλές μικρές, με την καθεμία να συνεχίζει εκεί που τελείωσε η προηγούμενη. Για αυτό το λόγο οφείλουμε να λαμβάνουμε υπόψιν το `offset` το οποίο δέχεται σαν όρισμα η συνάρτηση `my_read` ή `my_write` του driver μας (παρακάτω φαίνεται το πώς).

Σημείωση: Τα ορίσματα που περνιούνται στη κάθε συνάρτηση (π.χ. `read`) είναι διαφορετικά από το αντίστοιχο system call (`ssize_t read(int fd, void *buf, size_t count)`).

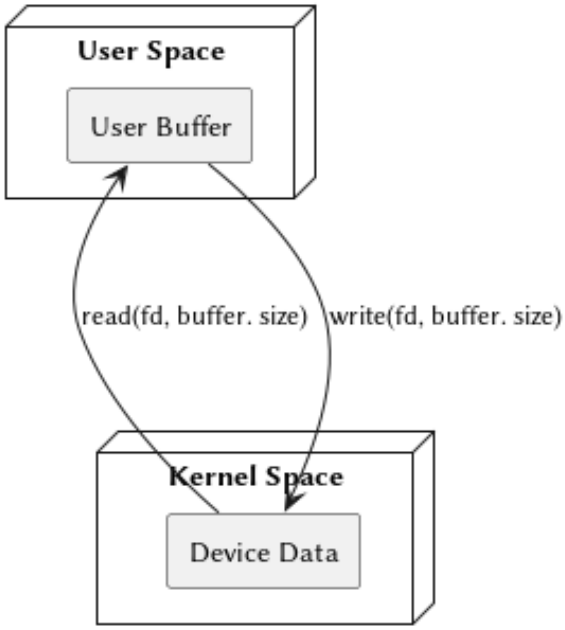
Αυτό γίνεται γιατί το λειτουργικό σύστημα μεσολαβεί και κάνει απλούστερη τη διεπαφή.

Για παράδειγμα, στο system call `read` ο χρήστης δεν χρειάζεται να περνάει κάποιο συγκεκριμένο offset. Ωστόσο, όταν πια η κλήση φτάσει στη συσκευή, έχει υπολογιστεί το offset από το οποίο θα συνεχιστεί το διάβασμα των δεδομένων. Έτσι, ο χρήστης, διαβάζοντας από μια συσκευή, βλέπει μια συνεχή ροή από bytes.

Παραδείγματα υλοποίησης των συναρτήσεων `my_read` και `my_write` όπου γίνεται μεταφορά δεδομένα μεταξύ του user-space και ενός buffer που ανήκει στη συσκευή:

read/write file operations

```
/* Driver read function */

static ssize_t my_read(struct file *file,
        char __user *user_buffer,
        size_t size, loff_t *offset)
{
    struct my_device_data *data =
        (struct my_device_data *) file->private_data;
    size_t to_read;


    /* Copy data->buffer to user_buffer using copy_to_user */
to_read = (size > data->size - *offset) ? (data->size - *offset) : size;
    if (copy_to_user(user_buffer, data->buffer + *offset, to_read) != 0)
        return -EFAULT;
    *offset += to_read;

    return to_read;
}

/* Driver write function */

static ssize_t my_write(struct file *file,
        const char __user *user_buffer,
        size_t size, loff_t *offset)
{
    struct my_device_data *data =
        (struct my_device_data *) file->private_data;


    /* Copy user_buffer to data->buffer, using copy_from_user */
    size = (*offset + size > BUFSIZ) ? (BUFSIZ - *offset) : size;
    if (copy_from_user(data->buffer + *offset, user_buffer, size) != 0)
        return -EFAULT;
    *offset += size;
    data->size = *offset;

    return size;
}
```



Σχήμα 3: Η μεταφορά δεδομένων από user-space στη συσκευή μας και το αντίστροφο.

5 Ασκήσεις

Άσκηση 1

Βρείτε στον πηγαίο κώδικα του Kernel τους ορισμούς των συμβόλων. Μπορείτε να χρησιμοποιήσετε το εργαλείο αναζήτησης LXR.

1. `container_of`
2. `struct file`
3. `struct file_operations`

Εξηγήστε τη σημασία τους.

Άσκηση 2

Μία εφαρμογή όπως ο διαχειριστής παραθύρων μας είναι υπεύθυνος για λειτουργίες όπως η εμφάνιση του κέρσορα του ποντικιού.

Όταν εμείς μετακινούμε τη φυσική συσκευή του ποντικιού βλέπουμε τον κέρσορα να μετακινείται.

Ο διαχειριστής παραθύρων δεν γνωρίζει στην πραγματικότητα τίποτα για το ποντίκι μας.

Κάθε ένα καθορισμένο χρονικό διάστημα η εφαρμογή αυτή ρωτάει το λειτουργικό σύστημα για να δει αν έχει μετακινηθεί το ποντίκι.

Το λειτουργικό σύστημα έχει μεταφράσει την είσοδο από τη συσκευή του ποντικιού σε μία ροή από bytes. Τα bytes αυτά ακολουθούν ένα συγκεκριμένο πρωτόκολλο μέσα από το οποίο περιγράφεται ουσιαστικά η κατεύθυνση στην οποία κινήθηκε το ποντίκι. Η ροή από bytes φαίνεται στο σύστημά σας ένα απλό αρχείο, το οποίο αν το διαβάσουμε θα δούμε μπορστά μας τη σειρά από bytes που περιγράφουν την κίνηση του ποντικιού μας ανα πάσα στιγμή.

Τελικά, αυτό το αρχείο το διαβάζει ο διαχειριστής παραθύρων και τελικά ζωγραφίζει το ποντίκι στη νέα θέση, όπως θα έπρεπε.

Μπορείτε και τώρα να δείτε αυτό το byte stream χρησιμοποιώντας την παρακάτω εντολή και έπειτα κουνώντας το ποντίκι σας. (Για πιο όμορφη έξοδο κατεβάστε την εφαρμογή hexdump εκτελώντας `sudo apt install hexdump`).

Command Line

```
$ sudo cat /dev/input/mice
$ sudo cat /dev/input/mice | hexdump # prettier output
```

Τί παρατηρείται;

Έπειτα, εκτελέστε το παρακάτω python script το οποίο ερμηνεύει την έξοδο της συσκευής ποντικιού σαν τριάδες από bytes.

mouse.py

```
import struct
f = open( "/dev/input/mice", "rb" );

# Open the file in the read-binary mode
while True:
  # Reads the 24 bytes
  data = f.read(3)
  # Unpack the bytes to integers
  print(struct.unpack('bbb',data))
```

Για να εκτελέσετε το script χρησιμοποιήστε την εντολή:

Command Line

```
$ sudo python mouse.py
```

Ποια είναι η έξοδος; Ερμηνεύστε την.

Σημείωση Μπορείτε να βρείτε την υλοποίηση του driver ποντικιού ο οποίος είναι ενσωματωμένος στο Linux Kernel στο αρχείο `linux/drivers/input/mouse.c`.

Άσκηση 3

Κάντε register ένα character device στο σύστημά σας χρησιμοποιώντας την εντολή `mknod`. Δώστε της major αριθμό 42, minor αριθμό 0 και όνομα `mydevice`.

Προσπαθήστε να γράψετε και να διαβάσετε στη συσκευή. Τί συμβαίνει;

Σημείωση Σε αυτή τη συσκευή στη συνέχεια θα αντιστοιχήσουμε τον driver που φτιάχνουμε.

Άσκηση 4

Φτιάξτε ένα kernel module το οποίο θα λειτουργήσει σαν driver της συσκευής που ορίσατε. Ο χρήστης θα μπορεί να διαβάσει από και να γράψει στη συσκευή. Συγκεκριμένα, όταν ο χρήστης διαβάζει από τη συσκευή θα παίρνει σαν έξοδο ένα μήνυμα που είναι αποθηκευμένο σε μνήμη που ανήκει στον πυρήνα. Όταν γράφει ο χρήστης στη συσκευή θα μπορεί να αλλάξει το μήνυμα το οποίο τυπώνεται.

Βασιστείτε στον σκελετό που δίνεται στο αρχείο `mychardev.c`.

Info: Μόλις συμπληρώσετε τον κώδικα. Κάντε τον compile και φορτώστε το module `chardev.ko` στον πυρήνα σας. Πλέον, θα μπορείτε να αλληλεπιδράσετε με την εικονική συσκευή που φτιάξατε στο προηγούμενο ερώτημα μέσω του driver.

Έπειτα, φτιάξτε ένα αρχείο σε όποια γλώσσα προγραμματισμού θέλετε που να επιδεικνύει τη λειτουργία της συσκευής σας.

6 Πηγές

- Linux Kernel Module Programming Guide
- Linux Device Drivers
- Operating System Concepts 9th Edition - Silbershatz, Galvin, Gagne (ch. 18)