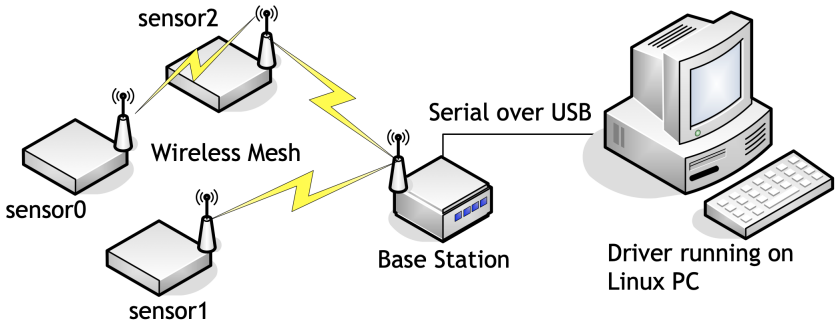
2η Εργαστηριακή Άσκηση Λειτουργικών Συστημάτων

Νικόλαος Οικονόμου: 03120014
Πέτρος Αυγερίνος: 03115074

1 Εισαγωγή

Σκοπός της συγκεκριμένης άσκησης είναι η δημιουργία ενός character device driver ο οποίος θα λειτουργεί πάνω σε ένα δίκτυο αισθητήρων, θα λαμβάνει μετρήσεις και θα τις περνάει στο userspace. Το δίκτυο αισθητήρων καταλήγει σε ένα σταθμό βάσης ο οποίος είναι συνδεδεμένος με USB και έτσι γίνεται η διασύνδεση των αισθητήρων με το character device driver. Για κάθε αισθητήρα ο οδηγός θα πρέπει να μεταφράζει την μέτρηση αυτή σε διαφορετική συσκευή και να επιτελεί κλήσεις συστήματος όπως open(), read(), ioctl() και άλλες. Ο character device driver υλοποιήθηκε σε εικονική μηχανή QEMU-KVM επομένως δεν υπάρχει πραγματική διεπαφή μέσω USB αλλά εικονική.



Σχήμα 1: Αρχιτεκτονική του υπό εξέταση συστήματος

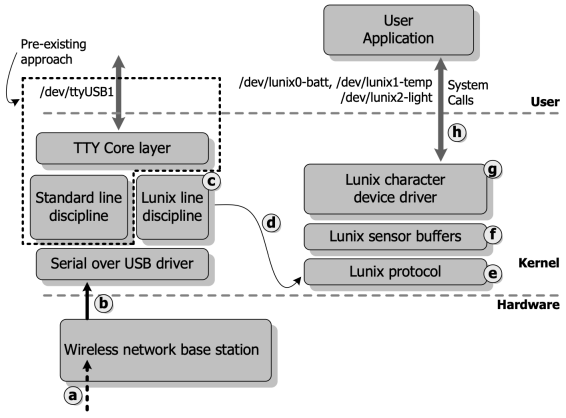
2 Αρχιτεκτονική

2.1 Αρχική Αρχιτεκτονική

Το σύστημα λαμβάνει μέσω του ασύρματου δικτύου τις μετρήσεις στο USB driver, της μεταφέρει σε μία διάταξη γραμμής και από εκεί στο ανάλογο TTY, για υπολογιστή εργαστηρίου το TTY είναι το /dev/ttyUSB, για εικονική μηχανή το TTY είναι το /dev/ttyS1.

2.2 Η δική μας Αρχιτεκτονική

Η υλοποίηση μας διαφέρει σημαντικά από την παραπάνω. Η ανάληψη δεδομένων γίνεται ξανά από μία διάταξη γραμμής**(lunix line discipline)**, αλλά αυτή τη φορά παρεμβάλλεται μία διαδικασία πρωτοκόλλου η **(lunix protocol)**οποία θα επεξεργαστεί τα δεδομένα ώστε να τοποθετηθούν στα κατάλληλα buffers. Η επεξεργασία αυτή γίνεται σε επίπεδο δικτυακού πακέτου όπου εξάγουμε την σημαντική πληροφορία και την τοποθετούμε στους **lunix sensor buffers**. Σε αυτό το σημείο είμαστε έτοιμοι να περάσουμε τις μετρήσεις των αισθητήρων στο userspace ώστε να γίνουν ορατές από τον user.



Σχήμα 2: Αρχιτεκτονική λογισμικού του υπό εξέταση συστήματος

3 Υλοποίηση

3.1 int lunix_chrdev_init()

Ξεκινήσαμε υλοποιώντας την αρχικοποίηση των συσκευών και την ανάθεσή τους στον πυρήνα των Linux. Αυτό γίνεται με χρήση ενός struct τύπου **cdev**, το οποίο είναι η εσωτερική αναπαράσταση του πυρήνα για ένα character device, πιο συγκεκριμένα του struct **lunix_chrdev_cdev**. Η παρακάτω συνάρτηση αναθέτει τα ορισμένα file operations στο συγκεκριμένο struct και αρχικοποιεί την δομή.

```
cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
```

Σε αυτό το σημείο θέλουμε να αναθέσουμε minor numbers στο συγκεκριμένο device driver και ως major number για πειραματική χρήση θα χρησιμοποιήσουμε τον 60.

```
register_chrdev_region(dev_no, lunix_minor_cnt, "lunix");
```

Με την παρακάτω εντολή κάνουμε εγγραφή του οδηγού συσκευής στον πυρήνα.

```
cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
```

3.2 static int lunix_chrdev_open()

Με αυτό το file operation 'ανοίγουμε' το αρχείο για επόμενα operations. Μέσω του inode μπορούμε να βρούμε το minor number, επομένως τον αισθητήρα και τον τύπο του. Θα λάβουμε μνήμη με ασφαλή τρόπο για τη δομή μας και θα κάνουμε τις αναθέσεις στη δομή αυτή ώστε να περαστεί στη δομή filp για να χρησιμοποιηθεί στις επόμενες συναρτήσεις.

Το memory allocation γίνεται ως εξής:

```
lunix_chrdev_state = kzalloc(sizeof(*lunix_chrdev_state), GFP_KERNEL);
```

Η ανάθεση της δομής μας ως εξής:

```
lunix_chrdev_state->sensor = &lunix_sensors[sensor];
lunix_chrdev_state->type = type;
lunix_chrdev_state->buf_lim = 0;
lunix_chrdev_state->buf_timestamp = 0;
```

και το πέρασμα τη δομής στο filp ως εξής:

```
filp->private_data = lunix_chrdev_state;
```

3.3 static ssize_t lunix_chrdev_read()

Για την υλοποίηση του read θα χρειαστούμε δύο βοηθητικές συναρτήσεις:

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state);
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state);
```

1. Η πρώτη εξυπηρετεί στην αναγνώριση της ανάγκης για refresh, δηλαδή αν τα δεδομένα που έχουμε στο buffer μας είναι διαφορετικά από την τελευταία μέτρηση του αισθητήρα μας.
2. Η δεύτερη εξυπηρετεί στην ανανέωση της τιμής αυτής με κατάλληλη επεξεργασία. Στην συγκεκριμένη συνάρτηση γίνεται χρήση ενός lookup πίνακα ο οποίος δέχεται bytes και στέλνει πίσω τιμές στη δεκαδική τους μορφή.

3.3.1 Λίγα λόγια για την update

Για την αποφυγή κάποιου race condition κάνουμε χρήση spinlocks, αφού διαβάζονται και ανανεώνονται οι τιμές στους sensor buffers, επιθυμούμε να μην προκύψει κάποιο conflict για να μην προκύψουν ανεπιθύμητες τιμές για συγκεκριμένα timestamps.

1. Χρησιμοποιούμε spinlocks έναντι semaphores για την αποφυγή sleep mode.
2. Κρατάμε λίγο το lock για να διατηρήσουμε καλή ροή σε όλους όσους το ζητούν χωρίς κάποιος να σπαταλάει χρόνο και πόρους από τη CPU.
3. Απενεργοποιούμε τα interrupts στην ανανέωση των δεδομένων από τους sensor buffers ώστε να μην προκύψει κάποιο deadlock.

3.3.2 Η υλοποίηση της read

Αρχικά θα ελέγξουμε για την κατάσταση του σεμαφόρο, αν είναι ελεύθερος θα συνεχίσουμε αλλιώς θα τοποθετηθούμε σε waiting queue για τον σεμαφόρο. Θα εργαστούμε με την χρήση του offset του αρχείου βλέποντας τη θέση του στο αρχείο, την ποσότητα που ζητείται από τον χρήστη να διαβαστεί και την ποσότητα της μέτρησης για να ελέγξουμε τι θα διαβαστεί τελικά.
Σαφώς αν δεν υπάρχει τίποτα στο αρχείο για να διαβαστεί περιμένουμε μέχρι να προκύψει κάτι για διάβασμα σε ένα waiting queue. Αυτή η διαδικασία είναι interruptable από την ανανέωση των sensor buffers. Κάθε διεργασία που 'πάει για ύπνο' πρέπει να αφήσει το lock. Με το interrupt ελέγχεται αν έχουμε διαφορετική μέτρηση από το refresh και η διεργασία συνεχίζει τη λειτουργία της.
Ελέγχουμε αν τα ζητούμενα bytes από τον χρήστη είναι περισσότερα από την θέση του offset και τα bytes της τελευταίας μέτρησης, αν είναι τα διορθώνουμε και περνάμε την πληροφορία στο χρήστη. Προσθέτουμε ύστερα τον αριθμό που διαβάσαμε στο offset και όταν διαβαστούν όλα τα δεδομένα μηδενίζουμε το offset.

3.4 static long lunix_chrdev_ioctl()

Σε αυτό το σημείο υλοποιήσαμε μία πολύ απλή εντολή που με κλήση ioctl() στη συσκευή μας και cmd το LUNIX_IOC_EXAMPLE, στέλνει στο log του πυρήνα ένα μήνυμα. Αρχικά ελέγχουμε αν το cmd που λάβαμε είναι ορθό, και ύστερα ελέγχουμε αν είναι ένα απο τα προκαθορισμένα για τα οποία γνωρίζουμε πως να αντιμετωπίσουμε και πράττουμε ανάλογα.

4 Testing

Αρχικά θα χρειαστούν τα παρακάτω για οποιαδήποτε λειτουργία του character device driver:

```
root@lunix# make
root@lunix# ./lunix-attach /dev/ttyS1
```

4.1 Testing της read()

Για τον έλεγχο λειτουργίας της read σε ένα νέο τερματικό κάνουμε cat οποιασδήποτε συσκευής από αυτές που δημιουργήσαμε:

```
root@lunix# cat /dev/lunix{0..15}-{batt,light,temp}
```

Η συσκευή λειτουργεί σωστά.

4.2 Testing της ioctl()

Για να ελέγξουμε την λειτουργία της ioctl() σε ένα νέο τερματικό ανοίξαμε το log του πυρήνα με την εντολή:

```
root@lunix# less /var/log/kern.log
```

Με το παρακάτω script στέλνουμε μια κλήση συστήματος ioctl() σε μία από τις συσκευές μας και κοιτούμε ταυτόχρονα το log για επιτυχία ή αποτυχία της κλήσης μας:

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include "lunix-chrdev.h"

int main(int argc, char **argv) {
    int fd = open(argv[1], O_RDONLY);
        if (fd < 0) {
                perror("open failed");
                exit(EXIT_FAILURE);
        }

        if (ioctl(fd, LUNIX_IOC_EXAMPLE,NULL) < 0) {
                perror("ioctl failed");
                exit(EXIT_FAILURE);
        }

        close(fd);

        exit(EXIT_SUCCESS);
}
```

4.3 Debugging

Για την διόρθωσει λαθών παρακολουθούσαμε το log του πυρήνα και τις κλήσεις συστήματων που γινόντουσαν στα binaries κατά τη διάρκεια του testing.

5 Κώδικας

Παρακάτω δίνεται ο κώδικας για το αρχείο lunix-chrdev.c:

```c
/*
 * lunix-chrdev.c
 *
 * Implementation of character devices
 * for Lunix:TNG
 *
 * < Your name here >
 *
 */

#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/list.h>
#include <linux/cdev.h>
#include <linux/poll.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/ioctl.h>
#include <linux/types.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mmzone.h>
#include <linux/vmalloc.h>
#include <linux/spinlock.h>

#include "lunix.h"
#include "lunix-chrdev.h"
#include "lunix-lookup.h"

/*
 * Global data
 */
// struct cdev:
struct cdev lunix_chrdev_cdev;

/*
 * Just a quick [unlocked] check to see if the cached
 * chrdev state needs to be updated from sensor measurements.
 */
/*
 * Declare a prototype so we can define the "unused" attribute and keep
 * the compiler happy. This function is not yet used, because this helpcode
 * is a stub.
 */

// State need Refresh
//                                refresh         timestamp
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
{
        struct lunix_sensor_struct *sensor;
```

```
        WARN_ON ( !(sensor = state->sensor));
        /* ? */
        debug("exiting state need refresh");
        return state->buf_timestamp != sensor->msr_data[state->type]->last_update;
}

/*
 * Updates the cached state of a character device
 * based on sensor data. Must be called with the
 * character device state lock held.
 */


// Update State
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
        struct lunix_sensor_struct *sensor = state->sensor;
        long int proper_data;
        uint32_t raw_time = sensor->msr_data[state->type]->last_update;
        uint32_t raw_value = sensor->msr_data[state->type]->values[0];
        char sign;
        debug("entering state update");


        // debug("leaving\n");

        /*
         * Grab the raw data quickly, hold the
         * spinlock for as little as possible.
         */
        /* ? */
        /* Why use spinlocks? See LDD3, p. 119 */

        /*
         * Any new data available?
         */
        /* ? */
        if (lunix_chrdev_state_needs_refresh(state)) {
                spin_lock_irq(&sensor->lock);
                raw_time = sensor->msr_data[BATT]->last_update;
                raw_value = sensor->msr_data[state->type]->values[0];
                spin_unlock_irq(&sensor->lock);
        }
        else {
                // -EAGAIN = no data available right now, try again later
                return -EAGAIN;
        }

        /*
         * Now we can take our time to format them,
         * holding only the private state semaphore
         */

        switch(state->type){
                case BATT:
                        proper_data = lookup_voltage[raw_value];
                        break;
                case TEMP:
                        proper_data = lookup_temperature[raw_value];
                        break;
                case LIGHT:
                        proper_data = lookup_light[raw_value];
                        break;
                default:
                        return -EAGAIN;
        }

        if (proper_data >= 0) {
                sign='+';
        }
        else {
                sign='-';
                proper_data = (-1)*proper_data;
        }


        state->buf_lim = snprintf(state->buf_data,
                        LUNIX_CHRDEV_BUFSZ, "%c%ld,%ld",
                        sign, proper_data/1000,
                        proper_data%1000);
        state->buf_timestamp = raw_time;

        /* ? */

        debug("leaving\n");
        return 0;
}

/*************************************
 * Implementation of file operations
 * for the Lunix character device
 *************************************/

// Open System Call
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
        /* Declarations */
        /* ? */
        //        lunix-chrdev.h                        struct
        //
        //
        //                                                        struct
        struct lunix_chrdev_state_struct *lunix_chrdev_state;
        int ret;
        int minor = iminor(inode);
        int sensor = minor >> 3;
        int type = minor%8; //??

        debug("entering\n");
        ret = -ENODEV;
        if ((ret = nonseekable_open(inode, filp)) < 0)
                goto out;

        /*
         * Associate this open file with the relevant sensor based on
         * the minor number of the device node [/dev/sensor<NO>-<TYPE>]
         */

        /* Allocate a new Lunix character device private state structure */
        /* ? */
```

```
        //                                      allocate
        //              struct state
        lunix_chrdev_state = kzalloc(sizeof(*lunix_chrdev_state), GFP_KERNEL);
        //      GFP_KERNEL flag                                 allocation                                      process
        //
        //                      struct
        //                              minor
        //      struct                          :
        //          -type {BATT, TEMP, LIGHT, N_LUNIX_MSR}
        //          -sensor,                                                                                    sensors
        //                          minor/8.
        //          -buf_lim
        //          -buf_data
        //          -lock
        //          -buf_timestamp
        //          -raw_data

        lunix_chrdev_state->sensor = &lunix_sensors[sensor];
        lunix_chrdev_state->type = type;
        lunix_chrdev_state->buf_lim = 0;
        lunix_chrdev_state->buf_timestamp = 0;
        // struct file: void *private_data;
        // The open system call sets this pointer to NULL before calling the open method for the driver.
        // The driver is free to make its own use of the field or to ignore it.
        // The driver can use the field to point to allocated data, but then must free memory
        // in the release method before the file structure is destroyed by the kernel.
        // private_data is a useful resource for preserving state information across system calls
        // and is used by most of our sample modules.
        filp->private_data = lunix_chrdev_state;

        // Initialize lock
        sema_init(&lunix_chrdev_state->lock,1);
out:
        debug("leaving, with ret = %d\n", ret);
        return ret;
}

// Release System Call (free allocated memory of file)
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
        /* ? */
        kfree(filp->private_data);
        return 0;
}

// Device Specific Commands
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
        // File descriptor not associated with character special device, or the request does not apply
        // to the kind of object the file descriptor references. (ENOTTY)
        int ret = -ENOTTY;
        int retry = -ERESTARTSYS;
        struct lunix_chrdev_state_struct *state;

        // https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch05.html
        if(_IOC_TYPE(cmd) != LUNIX_IOC_MAGIC) return ret;
        if(_IOC_NR(cmd) > LUNIX_IOC_MAXNR) return ret;
        state = filp->private_data;

        switch(cmd) {
                case LUNIX_IOC_EXAMPLE:
                        if(down_interruptible(&state->lock)) return retry;
                        debug("in LUNIX_IOC_EXAMPLE area");
                        up(&state->lock);
                        break;
                default: return ret;
        }
        /* Why? */
        // return -EINVAL; Invalid Argument
        debug("ioctl done my guy ");
        return 0;
}

static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt, loff_t *f_pos)
{
        ssize_t ret = 0;
        int retry = -ERESTARTSYS;

        struct lunix_sensor_struct *sensor;
        struct lunix_chrdev_state_struct *state = filp->private_data;
        WARN_ON(!state);

        sensor = state->sensor;
        WARN_ON(!sensor);

        /* Lock? */
        if(down_interruptible(&state->lock)){
                debug("down interruptable failed in read");
                return retry;
        }
        /*
         * If the cached character device state needs to be
         * updated by actual sensor data (i.e. we need to report
         * on a "fresh" measurement, do so
         */
        if (*f_pos == 0) {
                // Nothing to read
                while (lunix_chrdev_state_update(state) == -EAGAIN) {
                        /* ? */
                        /* The process needs to sleep */
                        /* See LDD3, page 153 for a hint */
                        up(&state->lock);
                        if (wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state))){
                                debug("wait interruptable failed in read");
                                return -ERESTARTSYS;
                        }
                        if (down_interruptible(&state->lock)){
                                debug("down interruptable failed in read");
                                return -ERESTARTSYS;
                        }
                }
        }

        /* End of file */
        /* ? */

        /* Determine the number of cached bytes to copy to userspace */
        /* ? */
        //          user                                                                                        ,
        //
        if (*f_pos + cnt >= state->buf_lim) {
```

```c
                cnt = state->buf_lim - *f_pos;
        }

        if (copy_to_user(usrbuf, state->buf_data, cnt)) {
                ret=-EFAULT;
                goto out;
        }

        //                                                      offset
        *f_pos += cnt;
        ret = cnt;


        /* Auto-rewind on EOF mode? */
        /* ? */
        if (*f_pos == state->buf_lim){*f_pos = 0;}

out:
        /* Unlock? */
        up(&state->lock);
        return ret;
}

// Mmap
static int lunix_chrdev_mmap(struct file *filp, struct vm_area_struct *vma)
{
        return -EINVAL;
}

//                              cdev
static struct file_operations lunix_chrdev_fops =
{
        .owner          = THIS_MODULE,
        .open           = lunix_chrdev_open,
        .release        = lunix_chrdev_release,
        .read           = lunix_chrdev_read,
        .unlocked_ioctl = lunix_chrdev_ioctl,
        .mmap           = lunix_chrdev_mmap
};

// Initialization
int lunix_chrdev_init(void)
{
        /*
         * Register the character device with the kernel, asking for
         * a range of minor numbers (number of sensors * 8 measurements / sensor)
         * beginning with LINUX_CHRDEV_MAJOR:0
         */
        int ret;
        dev_t dev_no;
        unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

        //              sensors * 2^3                                   minor numbers

        debug("initializing character device\n");
        cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
        //                                      cdev
        lunix_chrdev_cdev.owner = THIS_MODULE;

        dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
        //              device ID
        /* ? */
        /* register_chrdev_region? */
        //                                              device numbers
        //                                                      (               Major Number),
        //                                  ,
        ret = register_chrdev_region(dev_no, lunix_minor_cnt, "lunix");
        if (ret < 0) {
                debug("failed to register region, ret = %d\n", ret);
                goto out;
        }
        /* ? */
        /* cdev_add? */
        //                                              cdev_add
        //              :       cdev,
        //                  minor numbers.
        ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
        if (ret < 0) {
                debug("failed to add character device\n");
                goto out_with_chrdev_region;
        }
        debug("completed successfully\n");
        return 0;

out_with_chrdev_region:
        unregister_chrdev_region(dev_no, lunix_minor_cnt);
out:
        return ret;
}

// Destroy
void lunix_chrdev_destroy(void)
{
        dev_t dev_no;
        unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

        debug("entering\n");
        dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
        cdev_del(&lunix_chrdev_cdev);
        unregister_chrdev_region(dev_no, lunix_minor_cnt);
        debug("leaving\n");
}
```