Οδηγός Ασύρματου Δικτύου Αισθητήρων στο λειτουργικό σύστημα Linux

Ομάδα 2
Άδωνις-Μάριος Τσεριώτης el17838
Γεώργιος-Σάββας Συρογιάννης el17140

Εργαστήριο Λειτουργικών Συστημάτων

1 Περιγραφή κώδικα και αναφορά προβλημάτων που συναντήσαμε

Αρχικά, αναρωτηθήκαμε πώς λειτουργεί ο κώδικας που μας έχει δοθεί και πώς ο ήδη υλοποιημένος κώδικας λαμβάνει δεδομένα. Μετά απο μελέτη του κώδικα, των σχολίων και της περιγραφής, καταλάβαμε ότι το physical driver στέλνει τα δεδομένα μέσω της tty port **/dev/ttyS0** και το module που φορτώνουμε στον kernel συνδέεται με την συγκεκριμένη θύρα tty με την εντολή **./lunix-attach /dev/ttyS0**. Έτσι, αρχίζει και εκτελείται η δική μας line discipline (ldisc_open()) και "ακούει" τα δεδομένα που στέλνουν οι physicals drivers. Πιο συγκεκριμένα, εκτελείται η lunix_ldisc_receive(). Αυτή η συνάρτηση, περνάει τα bytes που λαμβάνει στο πρωτόκολλο που χρησιμοποιούμε για να αποκωδικοποιήσουμε τα bytes σε πληροφορίες σχεδόν human readable (πρέπει να μεσολαβήσει η αντιστοίχιση απο τους πίνακες του lookup_tables.h). Συγκεκριμένα, στην συνάρτηση lunix_protocol_received_buf(), η οποία κάνει parse τα δεδομένα φιλτράροντας τα με τις κατάλληλες συναρτήσεις που περιγράφει το πρωτόκολλο και στο τέλος καλεί την συνάρτηση lunix_protocol_update_sensors(), η οποία με τη σειρά της καλεί την lunix_sensor_update(), η οποία ενημερώνει τα πεδία του sensor struct με τα νέα δεδομένα κρατώντας το spinlock.

Σε αυτό το σημείο, παρατηρήσαμε ότι αυτό είναι το σημείο που ο κώδικας που πρέπει να γράψουμε επικοινωνεί με την λήψη δεδομένων. Στο αρχείο **lunix_chrdev.c** πρέπει να συμπληρώσουμε τις συναρτήσεις **lunix_chrdev_open()**, **lunix_chrdev_state_needs_refresh()**, **lunix_chrdev_state_update()**, **lunix_chrdev_release()**, **lunix_chrdev_read()**, **lunix_chrdev_init()**. Αρχίσαμε απο την needs_refresh, όπου μας δόθηκε το hint ότι είναι ένας γρήγορος τρόπος να ελέγξουμε αν υπάρχουν νέα δεδομένα χωρίς να χρειαζόμαστε κλείδωμα. Επομένως, καταλάβαμε οτι θα κοιτάμε το timestamp του σένσορα και θα βλέπουμε αν είναι ίδιος με το πεδίο *buf_timestamp* του **lunix_chrdev_state_struct** που ενημερώνουμε κάθε φορά που εισάγονται νέα δεδομένα. Στη συνέχεια, υλοποιήσαμε την update η οποία ελέγχει αν υπάρχουν δεδομένα με την παραπάνω συνάρτηση και παίρνει το spinlock του σένσορα για να πάρει τα δεδομένα χωρίς να μπορούν να αλλάξουν. Λαμβάνουμε τα δεδομένα και στη συνέχεια χρησιμοποιώντας τους πίνακες lookup μορφοποιούμε τις πραγματικές τιμές όπως χρειάζεται βάζοντας υποδιαστολή 3 θέσεις αριστερά. Ενημερώνουμε τα *buf_lim*,*buf_data*. Στη συνέχεια, μας προβλημάτισε η open, αλλα λάβαμε την βοήθεια απο το βιβλίο που προτείνεται **Linux Device Drivers**. Ο τρόπος που λειτουργεί είναι οτι ελέγχουμε αν ο minor αριθμός των αρχείων αντιστοιχούν σε κάποιον απο τους sensors με βάση τον τρόπο που έχουμε αριθμήσει τα αρχεία που θέλουμε να παίρνουν τα δεδομένα σε userspace. Εφόσον είναι μέσα στα όρια επιλέγουμε το συγκεκριμένο node κάνουμε allocate χώρο και δημιουργούμε το **lunix_chrdev_state_struct** που θα κάνει reference τον συγκεκριμένο char driver. Αρχικοποιούμε το instance του char dev state struct και τον "περνάμε" στα private data του file pointer, έτσι ώστε να μπορούμε να πάρουμε τα δεδομένα στην overloaded συνάρτηση για το system call read(). Στην συνάρτηση read, ακολουθούμε πάλι το βιβλίο **Lunix Device Drivers** και τον οδηγό για τον driver scull. Στην ουσία, κοιτάμε αν το αρχείο έχει ανοιχτεί για να διαβαστεί απο την αρχή (*fpos==0). Εφόσον αυτό ισχύει, καλούμε σε μια while loop την update για να δούμε αν έχουν ληφθεί τα δεδομένα κρατώντας τον semaphore του chrdev state struct γιατί κάνουμε access τα πεδία του και θέλουμε να αποφύγουμε πιθανά race conditions που θα πάνε να πανωγράψουν άλλες διεργασίες τα δεδομένα μας. Έπειτα, η διεργασία κοιμάται εως ότου να ληφθούν δεδομένα και η needs refresh επιστρέψει οτι υπάρχουν νέα δεδομένα. Έτσι, λαμβάνουμε με ασφάλεια, και χωρίς να καταναλώνουμε πολλούς πόρους, τα δεδομένα που χρειάζεται να πάρουμε. Στη συνέχεια, πρέπει να περάσουμε τα δεδομένα στο user space χρησιμοποιώντας την συνάρτηση copy_to_user(). Αρχικά, πρέπει να βρούμε τον αριθμό των χαρακτήρων που πρέπει να γράψουμε (αυτό γίνεται βρίσκοντας το minimum μεταξύ του count που δίνεται απο τον χρήστη και απο τον αριθμό δεδομένων που έχουμε που αποθηκεύεται στο field *buf_lim* στην συνάρτηση update και έπειτα (εφόσον η copy_to_user() σπάνια αποτυγχάνει) να προσθέσουμε στο fpos τον αριθμό χαρακτήρων που γράφτηκαν. Τέλος, απελευθερώνουμε τον semaphore για να μπορούν και άλλες διεργασίες να κάνουν access δεδομένα του struct. Για να λειτουργήσουν όλα τα παραπάνω, πρέπει στην init να καλέσουμε τις cdev_init που στην ουσία συσχετίζει τις overloaded συναρτήσεις μας με το char device struct του πυρήνα. Στην συνέχεια, συσχετίζουμε αυτό το cdev struct με τα αρχεία (χρησιμοποιώντας τους major αριθμούς των αρχείων) και στο τέλος προσθέτουμε τόσα character devices όσους σένσορες υποστηρίζει ο driver μας στον πυρήνα.

Ένα σημείο που έπρεπε να προσέξουμε, είναι η συνάρτηση release όπου πρέπει να κάνουμε free τα allocations που έχουμε προηγουμένως κάνει. Εδώ συναντήσαμε ένα bug το οποίο είχε ώς αποτέλεσμα kernel panic. Συγκεκριμένα, είχαμε κάνει global το instance του chrdev state struct και στην συνάρτηση release, κάναμε free αυτό το global instance. Έτσι, όταν κλείναμε το πρώτο αρχείο που είχαμε ανοίξει, ελευθερωνόταν η μνήμη που λογικά χρησιμοποιούσαν ακόμα απο άλλα αρχειά. Το bug αυτό διορθώθηκε όταν κάναμε free το πεδίο filp-¿private_data το οποίο περιέχει μέσα το chrdev state struct όπως αναφέρεται παραπάνω.

2 Κώδικας

```
/*
 * lunix-chrdev.c
 *
 * Implementation of character devices
 * for Lunix:TNG
 *
 * Adonis Tseriotis x George Syrogiannis
 *
 */

#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 lunix_chrdev_cdev;

/*
 * Just a quick [unlocked] check to see if the cached
 * chrdev state needs to be updated from sensor measurements.
 */

static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
{
        struct lunix_sensor_struct *sensor;

        /* Start of our code */
        sensor = state->sensor;
        /* End */

        WARN_ON ( !(sensor = state->sensor));
        /* ? */

        /* Start of our code */
        if(sensor->msr_data[state->type]->last_update != state->buf_timestamp) //If sensor has new data
        {
                return 0;
        }
```

```
        else
        {
                return -ENOREF;
        }

        /* End of our code */

        /* The following return is bogus, just for the stub to compile */
        /*return 0; ? */
}

/*
 * Updates the cached state of a character device
 * based on sensor data. Must be called with the
 * character device state lock held.
 */
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
        struct lunix_sensor_struct *sensor;

        /* Start of our code */
        uint32_t measurement;
        uint32_t timestamp;
        long temp;
        int i;

        sensor = state->sensor;

        /* End of our code */

        /*
         * Grab the raw data quickly, hold the
         * spinlock for as little as possible.
         */

        /* Start of our code */

        /* Check for new data */
        if (lunix_chrdev_state_needs_refresh(state)==0)
        {
                printk(KERN_INFO "I received new data. OMG I work\n");
                spin_lock(&sensor->lock);               //Acquire the lock

                measurement = sensor->msr_data[state->type]->values[0];                 //Take the measuremen
                timestamp = sensor->msr_data[state->type]->last_update;                 //Take the timestamp

                spin_unlock(&sensor->lock);
        }
        else
                return -EAGAIN;

        /* End of our code */

        /* Grab raw data */
        /* Why use spinlocks? See LDD3, p. 119 */

        /*
```

```
         * Any new data available?
         */
        /* ? */
        /* Checking that above */

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

        /* ? */

        /* Start of our code */
        switch (state->type)
        {
        case BATT:
                temp = lookup_voltage[measurement];
                state->buf_lim = sprintf(state->buf_data,"%ld%c%ld",(temp/1000),'.',(temp%1000));
                printk(KERN_INFO "Buff lim in struct = %d\n", state->buf_lim);
                break;
        case TEMP:
                temp = lookup_temperature[measurement];
                state->buf_lim = sprintf(state->buf_data,"%ld%c%ld",(temp/1000),'.',(temp%1000));
                break;
        case LIGHT:
                temp = lookup_light[measurement];
                state->buf_lim = sprintf(state->buf_data,"%ld%c%ld",(temp/1000),'.',(temp%1000));
                break;
        default:
                printk(KERN_INFO "Something's wrong there mate: Invalid enum value\n");
                break;
        }
        state->buf_data[state->buf_lim] = '\n'; //Display every number in new line
        state->buf_lim ++;
        state->buf_timestamp = timestamp;
        for(i=0; i<state->buf_lim; i++)
        {
                printk(KERN_INFO "buf_data[%d] = %c\n",i,state->buf_data[i]);
        }
        /* End of our code */
        debug("leaving\n");
        return 0;
}

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

static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
        /* Declarations */
        /* ? */
        /* Start of our code */
        struct lunix_chrdev_state_struct *chrdev_state;
        unsigned int minor;
        unsigned int node_id;
```

```
        /* End of our code */
        int ret;

        debug("entering\n");
        ret = -ENODEV;
        if ((ret = nonseekable_open(inode, filp)) < 0) //Never fails
                goto out;

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

        /* Start of our code */
        minor = iminor(inode);

        /* Allocate a new Lunix character device private state structure */
        /* ? */
        printk(KERN_INFO "Allocating memory for the character device state struct...\n");
        /* Allocate memory */
        ret = -ENOMEM;
        chrdev_state = kzalloc(sizeof(*chrdev_state), GFP_KERNEL);
        if (!chrdev_state){
                printk(KERN_ERR "Failed to allocate memory for Lunix char device state struct\n");
                goto out;
        }

        /* Define fields */
        chrdev_state->type = minor % 8; //Mask 3 LSBs
        printk(KERN_INFO "Just so you know, the type is %d. (In lunix1-temp supposed to be 1)\n",chrdev_stat

        ret = -ENXIO; //Error no such device
        node_id = minor / 8; //Mask rest
        /* See if device requested exists */
        if(node_id >= 0 && node_id < lunix_sensor_cnt)
                chrdev_state->sensor = &lunix_sensors[node_id];
        else
        {
                printk(KERN_WARNING "Node id %d is out of bounds [maximum %d sensors]\n",
                        node_id, lunix_sensor_cnt);
                goto out;
        }

        chrdev_state->buf_lim = 0; //Initialize characters to be read
        chrdev_state->buf_timestamp = -1; //Initialize timestamp of buf to take any first data given
        sema_init(&chrdev_state->lock,1); // Initialize semaphores to value 1
        filp->private_data = chrdev_state; //Save char device state in private data of file pointer
        ret = 0;
out:
        debug("leaving, with ret = %d\n", ret);
        return ret;
}

static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
        /* ? */
        kfree(filp->private_data);
```

```
        printk(KERN_INFO "It is now RELEASED\n");
        return 0;
}

static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
        /* Why? */
        return -EINVAL;
}

static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt, loff_t *f_pos)
{
        ssize_t ret, bugfix;

        struct lunix_sensor_struct *sensor;
        struct lunix_chrdev_state_struct *state;

        state = filp->private_data;
        WARN_ON(!state);

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

        /* Lock? */

        /* Start of our code */
        if(down_interruptible(&state->lock))
                return -ERESTARTSYS;

        /* End of our code */

        /*
         * 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
         */
        bugfix=0;
        if (*f_pos == 0)
        {
                while (lunix_chrdev_state_update(state) == -EAGAIN)
                {
                        /* Fix infinite loop causing kernel meltdown k kalaa*/
                        if(bugfix++ == 2983556)
                        {
                                return -ERESTARTSYS;
                        }
                        /* ? */

                        /* Start of our code */

                        up(&state->lock);

                        printk(KERN_INFO " reading: going for a nap\n");
                        if(wait_event_interruptible(sensor->wq, (lunix_chrdev_state_needs_refresh(state) ==
                                return -ERESTARTSYS;

                        /* If no new data, loop but first reacquire the lock */
```

```
                        if(down_interruptible(&state->lock))
                                return -ERESTARTSYS;

                        /* End of our code */

                        /* The process needs to sleep */
                        /* See LDD3, page 153 for a hint */
                }
        }

        /* Right now we have the lock */

        cnt = min(cnt, (size_t)(state->buf_lim - *f_pos));        // Take minimum count from the one the us

        /* fpos is greater than buf_lim */
        if(cnt <= 0)
        {
                ret = 0;
                goto out;
        }

        printk(KERN_INFO "You are the one to copy %ld bytes to user\n",cnt);
        if(copy_to_user(usrbuf,state->buf_data,cnt))
        {
                ret = -EFAULT;
                goto out;
        }

        *f_pos += cnt; // Move f_pos to how many bytes where written
        printk("Since you've written them now increment fpos to %lld\n",*f_pos);
        if(*f_pos >= state->buf_lim)
        {
                printk("By now I've copied everything you wanted. Leave me alone\n");
                *f_pos = 0;                //Reinitialize f_pos because wanted data was written to user
        }
        /* End of file */
        /* ? */
        ret = cnt;
        /* Determine the number of cached bytes to copy to userspace */
        /* ? */

        /* Auto-rewind on EOF mode? */
        /* ? */
        /* Both implemented above */
out:
        /* Unlock? */
        printk(KERN_INFO "Exiting read with ret = %ld...\n",ret);
        up(&state->lock);
        return ret;
}

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

```
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
};

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;
        printk(KERN_ERR "I am in init...\n");
        debug("initializing character device\n");
        cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
        lunix_chrdev_cdev.owner = THIS_MODULE;

        dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
        /* ? */
        /* register_chrdev_region? */
        ret = register_chrdev_region(dev_no,lunix_minor_cnt,"kwstas");
        if (ret < 0) {
                debug("failed to register region, ret = %d\n", ret);
                printk(KERN_ERR "Omg register is wronggg: %d\n", ret);
                goto out;
        }
        /* ? */
        /* cdev_add? */
        ret = cdev_add(&lunix_chrdev_cdev,dev_no,lunix_minor_cnt);
        if (ret < 0) {
                debug("failed to add character device\n");
                printk(KERN_ERR "OMGG add is wrongg: %d\n",ret);
                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;
}

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");
}
```