ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

2η Άσκηση

Οδηγός ασύρματου δικτύου αισθητήρων στο λειτουργικό σύστημα Linux.

Καναβάκης Ελευθέριος, ΑΜ: 03114180
Τσολίσου Δάφνη, ΑΜ: 03109752

Μάης 2018

Εισαγωγή

Έγινε υλοποίηση του οδήγου χαρακτήρων Lunix:TNG ο οποίος διαχειρίζεται την επεξεργασία των δεδομένων που υπάρχουν στους κατάλληλους buffers και την αποστολή αυτών στη διεργασία χρήστη που έκανε την αίτηση.

Για να πραγματοποιηθούν αυτά:

- συμπληρώσαμε τα system calls που χρειάζονταν (read, open, release).
- προσθέσαμε μηχανισμό επεξεργασίας των τιμών που στέλνουν οι συσκευές ώστε να εμφανίζονται ως δεκαδικοί αριθμοί στον χρήστη και έλεγχο ύπαρξης νέων δεδομένων.
- φτιάξαμε μηχανισμούς κλειδώματος για προστασία των κρίσιμων σημείων του κώδικα όταν υπάρχουν πολλές διεργασίες που προσπαθούν να τα πειράξουν.

Λεπτομέρειες υλοποίησης

Η συνάρτηση lunix_chrdev_init: Αρχικοποιεί τον οδηγό της συσκευής χαρακτήρων όταν τον προσθέτουμε στο kernel ως εξής:

1. δημιουργεί τον τύπο dev_t, ο οποίος θα αντιπροσωπεύει τη συσκευή, με την μακροεντολή MKDEV(int major, int minor) συνδυάζοντας τους αριθμούς major = 60 και minor = 0 σε έναν 32bit αριθμό.
2. δεσμεύει μια σειρά από minor numbers, συνολικά 128, με τη συνάρτηση register_chrdev_region(dev_t dev, unsigned int count, char *name).
3. προσθέτει την συσκευή στο σύστημα με την εντολή cdev_add (struct cdev *p, dev_t dev, unsigned count) και την κάνει ενεργή αμέσως. Σε περίπτωση αποτυχίας καλείτε η unregister_chrdev_region(dev_t dev, unsigned int count) η οποία αποδεσμεύει τους αριθμούς που ζήτησε.

Εντελώς ανάλογα γίνεται η καταστροφή της συσκευής από την συνάρτηση lunix_chrdev_destroy() η οποία την αφαιρεί από το σύστημα καλώντας με τη σειρά της τη συνάρτηση cdev_del(struct cdev *p) και αποδεσμεύει τους αριθμούς που είχε δημιουργήσει.

Η συνάρτηση lunix_chrdev_open: Καλείται όποτε ο χρήστης ζητά να μάθει τις τιμές που στέλνει κάποια μέτρηση (aka συσκευή) κάποιου αισθητήρα. Οι λειτουργίες της είναι:

1. βρίσκει το minor number που αντιστοιχεί στη συσκευή που ζήτησε ο χρήστης.
2. δεσμεύει χώρο στη μνήμη για ιδιωτική δομή δεδομένων της συγκεκριμένης συσκευής στην οποία κρατάει:
    - τον τύπο της συσκευής που ζήτησε ο χρήστης,
    - τον αισθητήρα που ζήτησε ο χρήστης,

- έναν buffer για το διάβασμα των δεδομένων,
- ένα timestamp για τον έλεγχο ύπαρξης νέων δεδομένων,
- ένα semaphore για το κλείδωμα κρίσιμων σημείων.

3. συσχετίζει τη συσκευή με τον δείκτη ανοιχτού αρχείου filp.

Η συνάρτηση lunix_chrdev_read: Ανοίγει το ειδικό αρχείο της συσκευής που ζήτησε ο χρήστης. Καλώντας δύο άλλες συναρτήσεις που υλοποιήσαμε, την lunix_chrdev_refresh και την lunix_chrdev_update γίνεται έλεγχος της ύπαρξης νέων δεδομένων και επεξεργασία τους αντίστοιχα. Τα τελικά αποτελέσματα αποστέλλονται στον χρήστη. Πιο συγκεκριμένα η συνάρτηση:

1. κλειδώνει μέσω της συνάρτησης down_interruptible(*semaphore) μόλις εκκινεί για κάποια διεργασία ώστε να εμποδίσει άλλες διεργασίες να τρέξουν ταυτόχρονα. Ξεκλειδώνει όταν τελειώνει ή όταν δεν υπάρχουν νέα δεδομένα και ξανακλειδώνει μόλις ανανεωθούν.

2. ελέγχει την ύπαρξη νέων δεδομένων μέσω της συνάρτησης lunix_chrdev_refresh η οποία συγκρίνει την τιμή του timestamp της εκάστοτε ιδιωτικής δομής με την τιμή του χρόνου τελευταίας ανανέωσης του κατάλληλου buffer. Αν δεν υπάρχουν νέα δεδομένα κοιμίζει την διεργασία μέσω της συνάρτησης wait_event_interruptible(wq, condition), προσθέτοντάς τη σε ουρά αναμονής (μία για κάθε συσκευή), εώς ότου έρθουν νέα δεδομένα ή λάβει κάποιο άλλο σήμα (πχ SIGKILL).

3. Επεξεργάζεται τα δεδομένα μέσω της συνάρτησης lunix_chrdev_update.

4. Στέλνει τα δεδομένα στον χρήστη μέσω της συνάρτησης του πυρήνα copy_to_user.

5. Ελέγχει αν ο δείκτης θέσης ανάγνωσης του αρχείου *f_pos ξεπέρασε την τιμή του buf_lim, δηλαδή του μεγέθους των δεδομένων που υπήρχαν στο τελευταίο update, και τον επαναφέρει στο 0 για να συνέχισει την προσπάθεια ανάγνωσης δεδομένων.

Η συνάρτηση lunix_chrdev_update: Λαμβάνει τα δεδομένα που υπάρχουν στον buffer που αντιστοιχεί στη συσκευή που ζήτησε η διεργασία χρήστη και τα μετατρέπει σε δεκαδικούς αριθμούς οι οποίοι στέλνονται πίσω. Τρέχει σε κατάσταση lock αφού η read την καλεί ενώ έχει κλειδώσει.

1. Χρησιμοποιεί spinlocks έτσι ώστε να εμποδίζει τυχόν interrupts την ώρα που ανανεώνει την τιμή του timestamp και παίρνει τα δεδομένα από τον buffer.

2. Ανάλογα με το ποια συσκευή ζήτησε ο χρήστης, το οποίο είναι αποθηκευμένο στην τρέχουσα ιδιωτική δομή, ελέγχεται ο αντίστοιχος πίνακας από το lookup_table με όρισμα την τιμή που διαβάστηκε από τον buffer και επιστρέφεται ο αριθμός ο οποίος κατόπιν μετρέπεται σε δεκαδικό και στέλνεται στη διεργασία χρήστη.

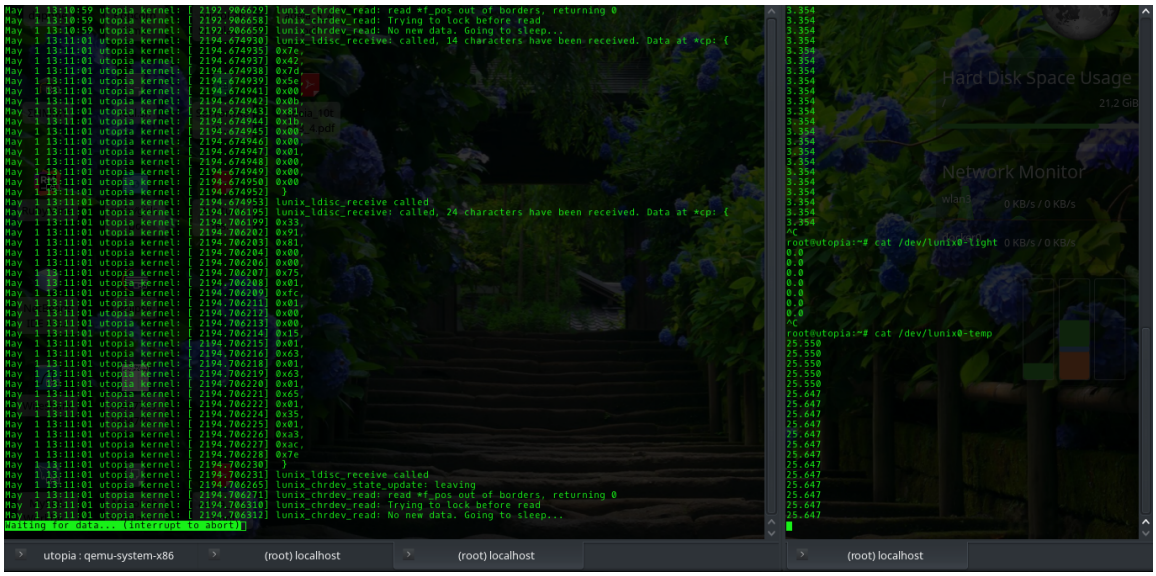
Στιγμιότυπα εκτέλεσης

Η εικόνα 1 δείχνει τα μηνύματα του πυρήνα καθώς εκτελείτε το πρόγραμμα του οδηγού και την έξοδο στην διεργασία χρήστη *cat /dev/lunix0-temp*.

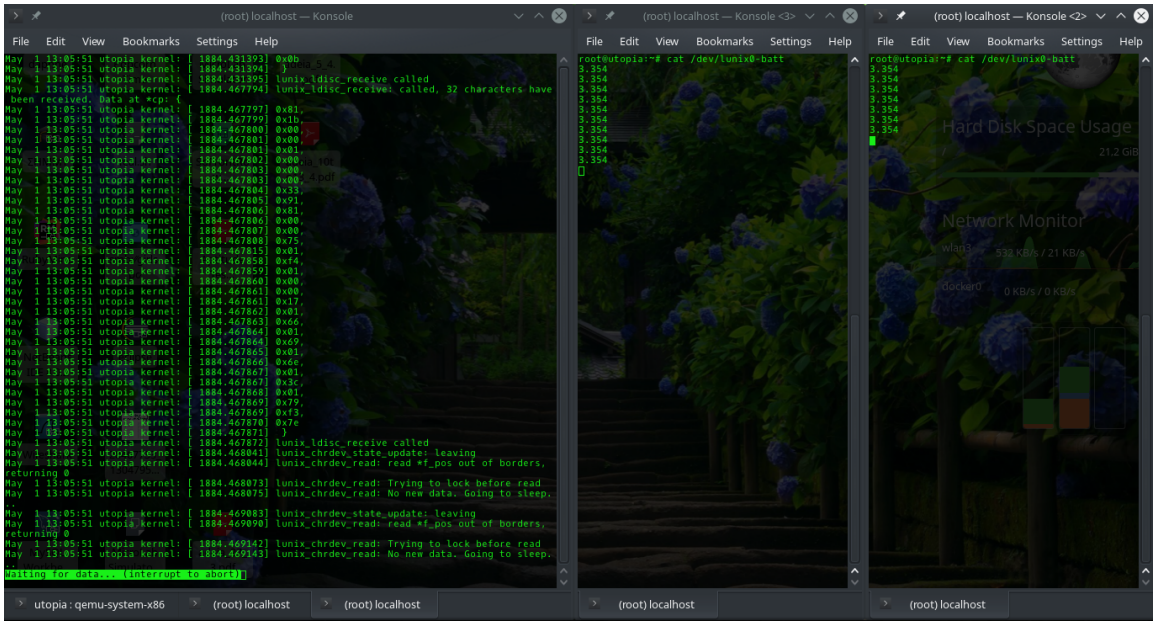
Η εικόνα 2 δείχνει τα μηνύματα του πυρήνα και τα αποτελέσματα που βλέπει ο χρήστης όταν τρέχουν δύο διεργασίες που διαβάζουν το ίδιο ειδικό αρχείο *cat /dev/lunix0-batt*. Έτσι φαίνεται ότι ο μηχανισμός κλειδώματος δουλεύει σωστά.

Τέλος η εικόνα 3 δείχνει τα μηνύματα του πυρήνα και τα αποτελέσματα που βλέπει ο χρήστης όταν τρέχουν τρεις διεργασίες, μία για κάθε ειδικό αρχείο του αισθητήρα 0.

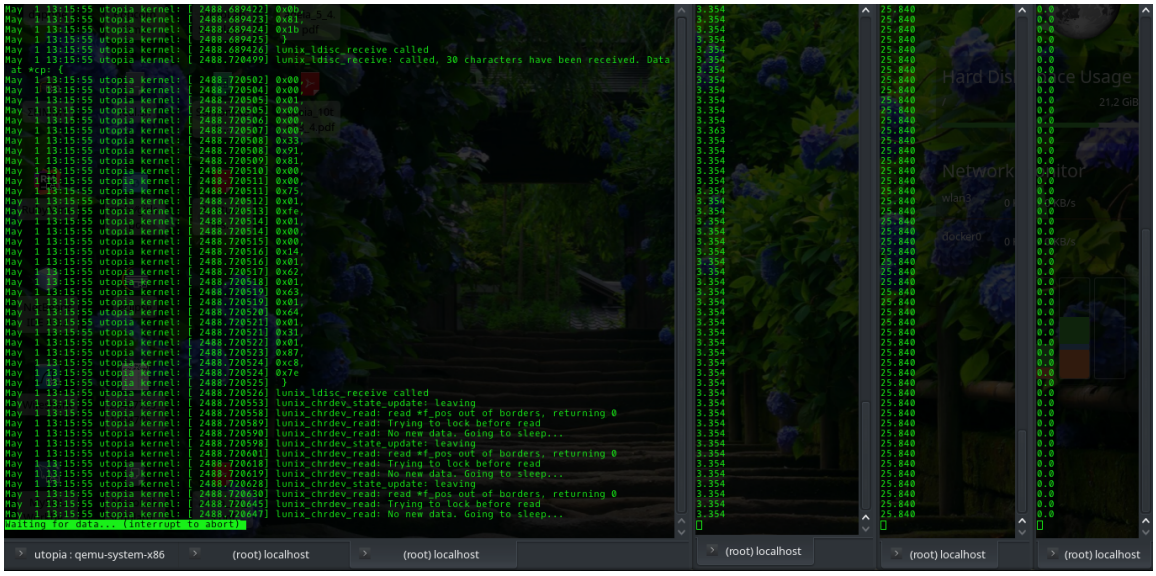
Σημείωση: Τα graphics είναι scalable και μπορείτε να κάνετε όσο zoom θέλετε.



Εικόνα 1: Στιγμιότυπο εκτέλεσης της cat /dev/lunix0-temp.



Εικόνα 2: Στιγμιότυπο ταυτόχρονης εκτέλεσης δύο εντολών cat /dev/lunix0-batt.



Εικόνα 3: Στιγμιότυπο ταυτόχρονης εκτέλεσης των εντολών cat /dev/lunix0-batt, cat /dev/lunix0-temp και cat /dev/lunix0-light.

Πηγαίος Κώδικας

lunix-chrdev.c

```
#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>
```

```c
#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;
        uint32_t state_time , sensor_time;

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

        sensor_time=sensor->msr_data[state->type]->last_update;  // time of last sensor measurement

        state_time=state->buf_timestamp;            // time of last measurement cached on state

        if(sensor_time > state_time ) return 1 ;  // measurements are out of date ;

        else return 0;  // measurements are up to date
}

/*
 * 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)
{
        uint16_t val ; // val is the measurement that we want to be updated
        long final ;   // val has to be looked up in the the lookup tables
        long ameros,dmeros;
        //int wr=0;
        struct lunix_sensor_struct *sensor;
        WARN_ON ( !(sensor = state->sensor));

        spin_lock(&sensor->lock);          // we use spinlocks cause we are in interrupt mode

        // entering critical section

        state->buf_timestamp=sensor->msr_data[state->type]->last_update;
        val=sensor->msr_data[state->type]->values[0];

        // leaving critical section

        spin_unlock(&sensor->lock);                 // release spinlock
        //debug("val= %d\n",val);
        // lookup the appropriate trasformation in the given tables

         if(state->type==0){

          // measurement is for battery
          final = lookup_voltage[val];
        }
        else if(state->type==1) {

          // measurement is for temperature
          final=lookup_temperature[val];
        }
        else {
```

```
          // measurement is for light
          final=lookup_light[val];
        }
        // final is in form xxyyy
        ameros=final/1000;    // ameros=xx
        dmeros=final%1000;    // dmeros=yyy
        //memcpy(state->buf_data,&final,sizeof final);

        sprintf(state->buf_data,"%ld.%ld\n",ameros, dmeros) ;
        state->buf_lim = strlen(state->buf_data);

        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)
{
        unsigned int minor ;
        //unsigned int major=imajor(inode);
        struct lunix_chrdev_state_struct *state;
        int ret,sensor,device;

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

        /* Allocate a new Lunix character device private state structure */
        state = kzalloc(sizeof(struct lunix_chrdev_state_struct) , GFP_KERNEL);
        if(!state) {
              printk(KERN_ERR "Failed to allocate memory for struct state in open\n");
              return -ENOMEM ; // out of memory
        }
        minor = iminor(inode) ;           // get the minor number of the sensor
        sensor = minor/8 ;                // sensor value is between 0 and 15
        device = minor % 8 ;             // device value is 1 for batt , 2 for temp and 3 for light

        debug("Creating state for sensor %d and device %d\n",sensor,device);

        state->type=device;
        state->buf_lim=0;
        state->buf_timestamp=0;
        state->sensor=&lunix_sensors[sensor];
        sema_init(&state->lock,1);
        filp->private_data=state;

        debug("Private state structure allocated successfully.\n");

out:
        debug("leaving, with ret = %d\n", ret);
        return ret;
}

// release is being called when a user process calls close for file filp
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
        // private data must be freed before kernel closes filp
        kfree(filp->private_data);
        return 0;
}

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

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

        struct lunix_sensor_struct *sensor;
        struct lunix_chrdev_state_struct *state;

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

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

        debug("Trying to lock before read\n");
        /*Lock before trying to read new data*/
        if (down_interruptible(&state->lock)){
                return -ERESTARTSYS;
        }
        /*
         * 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) {
                while (lunix_chrdev_state_needs_refresh(state) == 0) {//nothing to read
                    /* The process needs to sleep */
                    up(&state->lock);//unlock

                    if (filp->f_flags & O_NONBLOCK)
                        return -EAGAIN;//try again flag

                    debug("No new data. Going to sleep...\n");
                    if (wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state)==1))
                        return -ERESTARTSYS;//signal the fs layer to handle it

                    /* otherwise loop, but first reacquire the lock */
                    if (down_interruptible(&state->lock))
                        return -ERESTARTSYS;
                }
                //now get the data
                lunix_chrdev_state_update(state);
        }

        /* Determine the number of cached bytes to copy to userspace */
        if (*f_pos + cnt > state->buf_lim)
            cnt = state->buf_lim - *f_pos;

        if (copy_to_user(usrbuf, &state->buf_data + *f_pos, cnt)){
            ret = -EFAULT;
            debug( " problem with copy to user \n");
            goto out;
        }

        *f_pos += cnt;
        ret = cnt;
        if (*f_pos >= state->buf_lim){
            //rewind
            *f_pos = 0;
            debug("read *f_pos out of borders, returning 0\n");
        }


out:
        /* Unlock*/
        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;                // return value
        dev_t dev_no;
        unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;   //lunix_minor_cnt=sensor*8

        // initialize character device
        debug("initializing character device\n");
        cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
        lunix_chrdev_cdev.owner = THIS_MODULE;
        lunix_chrdev_cdev.ops = &lunix_chrdev_fops;
        debug("lunix_minor_cnt = %d\n",lunix_minor_cnt);
        dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
        // assigning major number to the new device driver
        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 adds a char device to the system

        /*
         *  first argument is --> the cdev structure for the device,
         *  second argument is --> device number for which this device is responsible
         *  third argument is --> number of consecutive minor numbers corresponding to the device
            in our case each sensor will take three consecutive 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;
}

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");
}
```