Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή: Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Εργαστήριο λειτουργικών συστημάτων (7ο εξάμηνο)
2ο Εργαστήριο

Δημήτριος Καλαθάς – el18016
Δημήτριος Καλέμης – el18152

Περιγραφή άσκησης

Ζητούμενο της άσκησης είναι η υλοποίηση ενός απλού οδηγού συσκευής **LUNIX:TNG** για ένα ασύρματο σύστημα αισθητήρων σε περιβάλλον Linux. Πιο αναλυτικά, έχουμε 16 αισθητήρες που λαμβάνουν μετρήσεις για τη τάση που τους τροφοδοτεί και στοιχεία του περιβάλλοντος τους, δηλαδή θερμοκρασία και ένταση φωτισμού. Τα δεδομένα στέλνονται σε ένα σταθμό βάσης, ο οποίος συνδέεται μέσω USB (σειριακή θύρα ttyS0) σε υπολογιστικό σύστημα Linux όπου και εκτελείται και ο οδηγός. Ο οδηγός που υλοποιήσαμε, πρακτικά λειτουργεί ως μηχανισμός για την αντιμετώπιση των αισθητήρων και των μετρήσεων τους ανεξάρτητα. Παράλληλα, μας επιτρέπει την ταυτόχρονη πρόσβαση στα δεδομένα που λαμβάνονται από τους αισθητήρες καθώς και την εφαρμογή ποικίλλων πολιτικών πρόσβασης.

Αρχιτεκτονική συστήματος

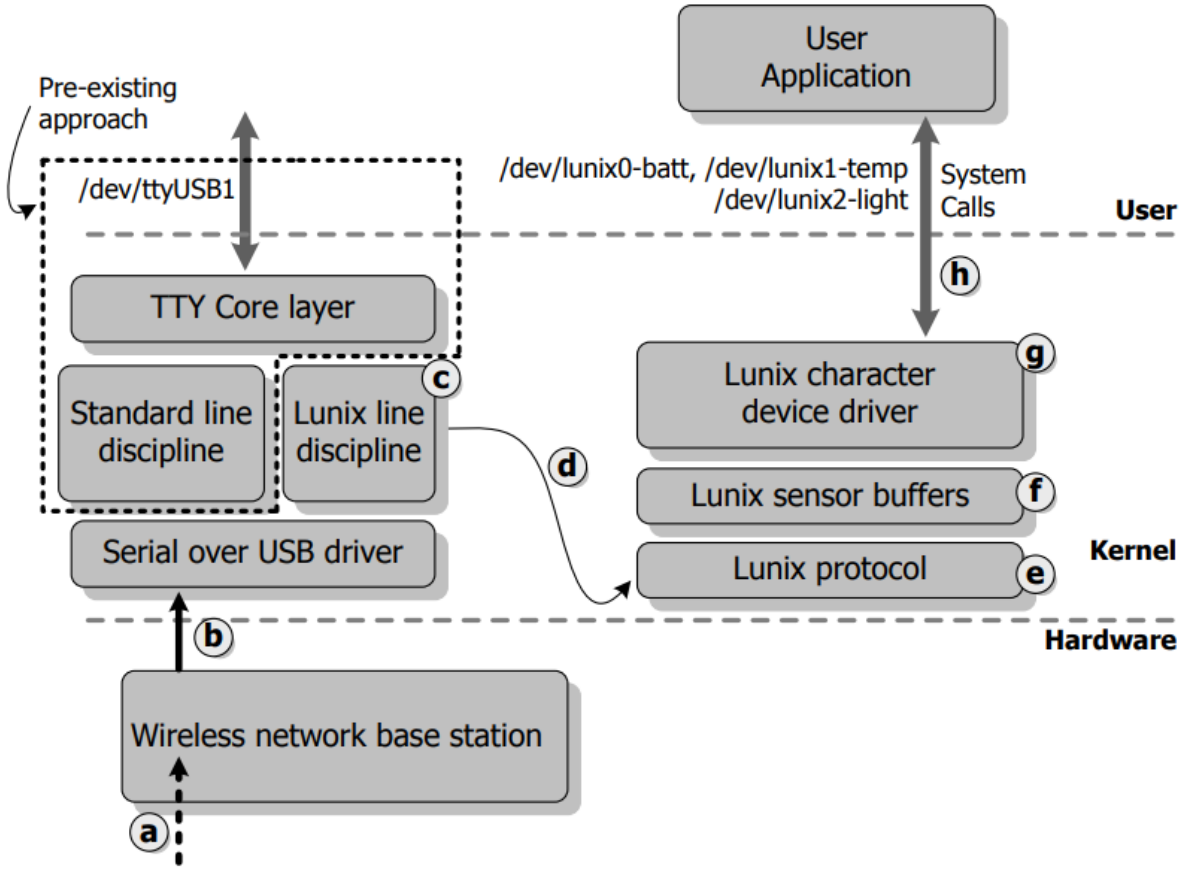
Η υλοποίηση του συστήματος οργανώνεται σε δύο κύρια μέρη:

Το πρώτο μέρος περιλαμβάνει τη συλλογή δεδομένων από το σταθμό βάσης (lunix line discipline) και την επεξεργασία τους με βάση συγκεκριμένο πρωτόκολλο (lunix protocol), με αποτέλεσμα τη διεξαγωγή και αποθήκευση των μετρούμενων μεγεθών στους κατάλληλους buffers (lunix sensor buffers).

Το δεύτερο μέρος λαμβάνει τα δεδομένα από τους ενδιάμεσους buffers και τα αναπαριστά στο χώρο χρήστη (user space) σε κατάλληλη μορφή. Αυτό είναι το κομμάτι που έπρεπε εμείς να φτιάξουμε. Πιο συγκεκριμένα, οι συναρτήσεις που υλοποιήσαμε, ικανοποιούν κλήσεις συστήματος στις οποίες θα οδηγηθεί μια διεργασία όταν εκτελεστεί πάνω στις συσκευές μας.

Τελικό σύστημα

Ο σταθμός βάσης συλλέγει τα δεδομένα των μετρήσεων των αισθητήρων και τα προωθεί μέσω της σειριακής θύρας USB στο υπολογιστικό σύστημα. Έπειτα, επεξεργάζονται από το φίλτρο γραμμής διάταξης του lunix και στέλνονται σε κατάλληλο στρώμα πρωτοκόλλου. Εδώ γίνεται η ερμηνεία των bytes των δεδομένων και στη συνέχεια αποθηκεύονται σε διαφορετικούς buffers ανά αισθητήρα. Τέλος, η συσκευή-οδηγός χαρακτήρων θα εμφανίζει στο χώρο χρήστη τα δεδομένα των buffers σε κατάλληλη μορφή. Παρακάτω, απεικονίζεται το τελικό σύστημα:



Σχήμα 3: Αρχιτεκτονική λογισμικού του υπό εξέταση συστήματος

Περιγραφή κώδικα

Η ανάπτυξη των συναρτήσεων έγινε στο αρχείο lunix-chrdev.c και προσθέσαμε μικρό κομμάτι κώδικα στο lunix-chrdev.h.

- **lunix_chrdev_Init**

```
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; //16*8

    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);

    //-----------------------------------------------//
    ret = register_chrdev_region(dev_no, lunix_minor_cnt, "Lunix:TNG");
//register a range of device numbers
    if (ret < 0)
    {
        debug("failed to register region, ret = %d\n", ret);
        goto out;
    }
    //-----------------------------------------------//
    ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt); //add a char
device to the system
    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;
}
```

Αυτή η συνάρτηση καλείται με την εκτέλεση της εντολής insmod, η οποία εισάγει ένα καινούργιο module στον πυρήνα. Επομένως, καλείται μόνο μία φορά και επεκτείνει τις δυνατότητες του πυρήνα. Στο παραπάνω κώδικα, δηλώνεται μια συσκευή χαρακτήρων με την εντολή cdev_init() και ζητάμε minor numbers για κάθε sensor με την register_chrdev_region(). Αφού τα δεσμεύσουμε, καλούμε την cdev_add() και πλέον η συσκευή μας έχει ενεργοποιηθεί και ακούει κάθε κλήση συστήματος του χώρου χρήστη.

- **lunix_chrdev_open**

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
    /* Declarations */
    unsigned int minor, sensor, type; //
    struct lunix_chrdev_state_struct *state;
    int ret;

    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>]
     */
    //--- minor = αισθητήρας * 8 + μέτρηση. ---///
    minor = iminor(inode);
    sensor = minor / 8; // 0-15
    type = minor % 8;   // 0-2
    debug("Done assosiating file with sensor %d of type %d\n", sensor, type);

    /* Allocate a new Lunix character device private state structure */
    //GFP_KERNEL    This is a normal allocation and might block.
    //This is the flag to use in process context code when it is safe to sleep
    state = kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);
    if (!state)
    {
        debug("kmalloc: could not allocate requested memory\n");
        ret = -ENOMEM;
        goto out;
    }
    state->type = type;
    state->sensor = &lunix_sensors[sensor];

    /*buffer init*/
    state->buf_lim = 0;
    state->buf_timestamp = 0;
    state->data_type = 0;

    filp->private_data = state; //points to the current state of the device
                               //stores a pointer to it for easier access
    sema_init(&state->lock, 1); //initialize semaphore with 1, refers to a
single struct file

    ret = 0;

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

Ο χρήστης καλεί τη συγκεκριμένη κλήση συστήματος για να ανοίξει ένα ειδικό αρχείο ώστε να αρχίσει την επικοινωνία με τη συσκευή. Εδώ βρίσκουμε σε ποια μέτρηση ποιανού αισθητήρα θέλει να αποκτήσει πρόσβαση, μέσω του minor number της συσκευής. Επιπλέον, δεσμεύουμε χώρο για το struct της κατάστασης της συσκευής.

- **lunix_chrdev_read**

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

    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*/
    debug("read and lock\n");
    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_update(state) == -EAGAIN)
        {
            /* ? */
            /* The process needs to sleep */
            /* See LDD3, page 153 for a hint */
            debug("nothing to read\n");
            /* nothing to read */
            up(&state->lock); /* release the lock */

            //if the file was opened with O_NONBLOCK flag return -EAGAIN
            if (filp->f_flags & O_NONBLOCK)
                return -EAGAIN;

            //sleep, if condition is FALSE, if you re woken up check condition
again and sleep or leave
            //wait_event_interruptible returns nonzero when interrupted by
signal
            if (wait_event_interruptible(sensor->wq,
lunix_chrdev_state_needs_refresh(state)))
                return -ERESTARTSYS;
```

```
            if (down_interruptible(&state->lock))
                return -ERESTARTSYS;
        }
    }
    debug("Ok, now I have fresh values\n");

    remaining_bytes = state->buf_lim - *f_pos;

    if (cnt >= remaining_bytes)
    {
        cnt = remaining_bytes;
    }

    /*
     * copy_to_user(to, from, count)
     * returns the number of bytes that couldnt copy
     */
    if (copy_to_user(usrbuf, state->buf_data + *f_pos, cnt))
    {
        ret = -EFAULT;
        goto out;
    }

    //fix the position
    *f_pos = *f_pos + cnt;
    ret = cnt;

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

out:
    /* Unlock */
    up(&state->lock);
    debug("end of reading and unlock\n");
    return ret;
}
```

Είναι η σημαντικότερη συνάρτηση για την υλοποίηση αυτού του οδηγού. Κατά την κλήση της, ο driver ελέγχει αν υπάρχουν νέα δεδομένα για το χρήστη (ο δείκτης f_pos είναι στην αρχή). Στην περίπτωση που δεν υπάρχουν, στέλνει τη διεργασία στη σειρά αναμονής του συγκεκριμένου αισθητήρα όπου «κοιμάται». Με την άφιξη νέων δεδομένων, «ξυπνάει» μαζί με τις άλλες διεργασίες που περιμένουν στην ίδια ουρά. Από την άλλη, στην περίπτωση που υπάρχουν νέα δεδομένα, στέλνουμε στο χρήστη όσα δεδομένα ζήτησε ή αν ζήτησε παραπάνω στέλνουμε όσα είναι δυνατόν. Αυτό πραγματοποιείται με τη συνάρτηση copy_to_user που μεταφέρει δεδομένα στο buffer που ζήτησε ο χρήστης με μεγαλύτερη ασφάλεια και κατάλληλους ελέγχους σε σχέση με άλλες συναρτήσεις που κάνουν παρόμοια δουλειά (πχ memcpy). Επιπρόσθετα, αν ο δείκτης f_pos φτάσει στην οριακή τιμή

state->buf_lim, τότε μηδενίζεται και ξεκινάμε από την αρχή. Αυτές οι λειτουργίες πρέπει να γίνουν με αποκλειστική πρόσβαση από μία διεργασία στη δομή του αρχείου, για αυτό και στην αρχή της συνάρτησης κλειδώνουμε το σημαφόρο (down_interruptible(&state->lock)).

- **lunix_chrdev_state_update**

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
    struct lunix_sensor_struct *sensor;
    unsigned long flags;        //spinlock's flag is an unsigned long int
    uint16_t value;             //lookup tables require uint16_t
    long int result;            //lookup tables give long int
    uint32_t current_timestamp; //defined in lunix_msr_data_struct
    int ret;

    debug("leaving\n");

    sensor = state->sensor;
    ret = -EAGAIN; //there is no data available right now, try again later

    //save the state, if locked already it is saved in flags
    //saving the state here with irqsave is redundant
    //spinlock is used here because of small code not interrupt context
    spin_lock_irqsave(&sensor->lock, flags);
    value = sensor->msr_data[state->type]->values[0];
    current_timestamp = sensor->msr_data[state->type]->last_update;
    // return to the formally state specified in flags
    spin_unlock_irqrestore(&sensor->lock, flags);

    if (lunix_chrdev_state_needs_refresh(state))
    {

        if (!state->data_type)
        {
            if (state->type == BATT)
            {
                result = lookup_voltage[value];
            }
            else if (state->type == TEMP)
            {
                result = lookup_temperature[value];
            }
            else if (state->type == LIGHT)
            {
                result = lookup_light[value];
            }
            else
```

```
            {
                debug("Internal Error: Type doesnt match one the three
available \
                                                                (BATT, TEMP,
LIGHT)");
                ret = -EMEDIUMTYPE; //wrong medium type
                goto out;
            }
            /*save formatted data in chrdev state buffer*/
            ret = 0;
            state->buf_timestamp = current_timestamp;
            state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ,
"%ld.%03ld\n", result / 1000, result % 1000);
        }
        else
        { //raw data
            debug("skipped lookup table conversion, returning raw
bytes...\n");
            ret = 0;
            state->buf_timestamp = current_timestamp;
            state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ,
"%x\n", value); //prints raw_data as hex
        }
    }
    else
    {
        ret = -EAGAIN;
    }

out:
    debug("leaving\n");
    return ret;
}
```

Η συνάρτηση update ελέγχει τα δεδομένα του buffer και αν υπάρχουν νέα δεδομένα τότε χρησιμοποιεί τους πίνακες που υπάρχουν στα αρχεία lunix-lookup.h για να μετατρέψει τα δεδομένα στη μορφή που θέλουμε. Για να έχουμε πρόσβαση στα δεδομένα των buffers, κλειδώνουμε με τη συνάρτηση spin_lock_irqsave(), η οποία φροντίζει να απενεργοποιήσει τις διακοπές και αντίστοιχα την spin_unlock_irqrestore() για να αποδεσμεύσουμε το κλείδωμα. Αυτό οφείλεται στο γεγονός ότι ο κώδικας που τρέχει σε process context ανταγωνίζεται με κώδικα που τρέχει σε interrupt context. Έτσι, λοιπόν, αποφεύγεται κάθε ενδεχόμενη κατάσταση αδιεξόδου που θα μπορούσε να εμφανιστεί στην περίπτωση που συμβεί μια διακοπή στον επεξεργαστή που τρέχει τη διεργασία.

- **lunix_chrdev_state_needs_refresh**

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct
*state)
{
    int ret = 0; //by default we are assuming that the state doesnt need
refreshing
    struct lunix_sensor_struct *sensor;

    WARN_ON(!(sensor = state->sensor));
    if (sensor->msr_data[state->type]->last_update != state->buf_timestamp)
    {
        debug("State needs refreshing!\n");
        ret = 1; //now state needs refreshing
    }

    return ret;
}
```

Ελέγχει αν τα δεδομένα που παραλάβαμε από τους buffers (sensor->msr_data[state->type]->last_update != state->buf_timestamp) είναι τα τελευταία που έχουμε αποθηκευμένα στη συσκευή. Αν όχι, σημαίνει ότι πρέπει να ανανεωθούν διότι υπάρχουν νέα δεδομένα.

- **lunix_chrdev_release**

```
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
    debug("Releasing allocated memory for private file data\n");
    kfree(filp->private_data);
    debug("Done releasing private file data, exiting now..");
    return 0;
}
```

Αυτή η κλήση συστήματος καλείται όταν τελειώνει η πρόσβαση του χρήστη στη συσκευή και ο σκοπός της είναι να απελευθερώσει όσο χώρο έχει δεσμεύσει ο driver με τη κλήση της open.

- **lunix_chrdev_ioctl**

```
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned
long arg)
{
    struct lunix_chrdev_state_struct *state;
    debug("entering\n");

    //if cmd's type is not LUNIX_IOC_MAGIC (it is 60, the major number) return
error
    if (_IOC_TYPE(cmd) != LUNIX_IOC_MAGIC)
        return -ENOTTY;
    //accept only 1 cmd
    if (_IOC_NR(cmd) > LUNIX_IOC_MAXNR)
        return -ENOTTY;

    state = filp->private_data;

    switch (cmd)
    {
    case LUNIX_IOC_DATA_TYPE_CONVERT:
        if (down_interruptible(&state->lock)) //in case multiple procs with
same fd use ioctl
            return -ERESTARTSYS;
        //if I have raw data I turn them into coocked and vice versa
        if (state->data_type)
            state->data_type = 0; //turned into coocked
        else
            state->data_type = 1; //turned into raw
        up(&state->lock);
    }

    debug("successfully changed data type transfer, now state->raw_data=%d\n",
state->date_type);
    return 0; */
}
```

Ο χρήστης με αυτή τη κλήση συστήματος επιλέγει αν τα δεδομένα εξόδου θα είναι raw ή cooked. Για το σκοπό αυτό, προσθέσαμε στο αρχείο lunix-chrdev.h το πεδίο int data_type καθώς και κάποιες σταθερές για τη χρήση εντολών στην ioctl. Όταν το data_type έχει τιμή μηδέν (data_type == 0), τα δεδομένα εμφανίζονται cooked ενώ όταν έχει τη τιμή 1, εμφανίζονται raw (δηλαδή ακατέργαστα, σε hex μορφή). Με τη προσθήκη της κλήσης συστήματος αυτής, επηρεάζεται και η **lunix_chrdev_state_update**, η οποία όπως φαίνεται και παραπάνω έχει δύο περιπτώσεις ανάλογα πως θέλουμε τα δεδομένα μας να εμφανίζονται στους buffers (χώρος χρήστη).