ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά για την 2η Άσκηση:

Οδηγός Ασύρματου Δικτύου Αισθητήρων στο ΛΣ Linux

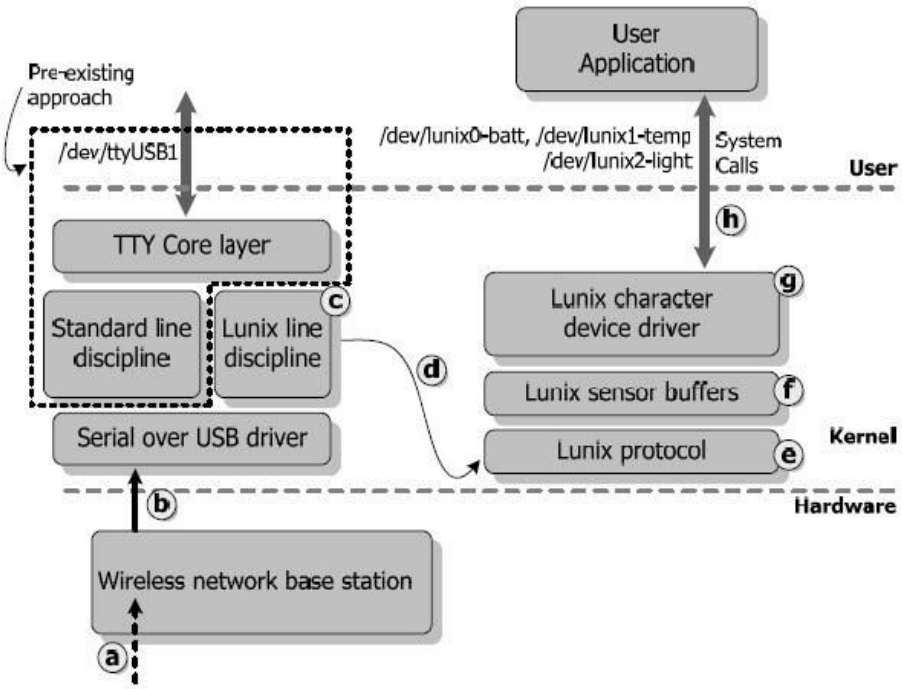
Ον/μο :Τελάλη Ειρήνη
Α.Μ.: :03113009

Ον/μο :Μπερέτσος Θεόδωρος
Α.Μ.: :03111612

Ομάδα: b27
Ημερομηνία εξέτασης: 11/5/2017
Ημερομηνία παράδοσης: 18/5/2017

1 Σκοπός

Αντικείμενο της πρώτης εργαστηριακής άσκησης είναι η υλοποίηση ενός οδηγού συσκευής (device driver) για ένα ασύρματο δίκτυο αισθητήρων κάτω από το λειτουργικό σύστημα Linux. Ο driver εισάγεται στον πυρήνα δυναμικά με το πρόγραμμα `insmod` αρχικοποιώντας όλες τις λειτουργίες του, τις οποίες θα περιγράψουμε παρακάτω. Το τελικό σύστημα θα λειτουργεί ως εξής:



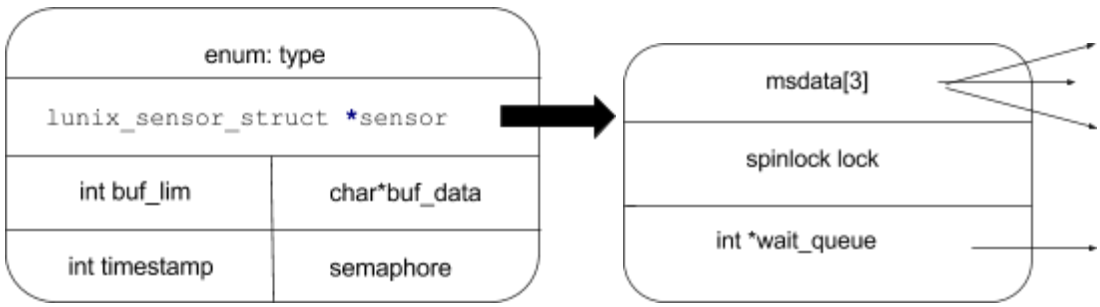
Αρχιτεκτονική λογισμικού του υπό εξέταση συστήματος

Εμείς θα υλοποιήσουμε τη ζητούμενη συσκευή χαρακτήρων **(g)**, αναλόγως με το ποιο ειδικό αρχείο χρησιμοποιεί η εφαρμογή χρήστη, θα επιτρέπει την ανάκτηση **(h)** από τους buffers των δεδομένων που αντιστοιχούν σε κάθε περίπτωση.

2 Πηγαίος κώδικας (source code)

Οι βασικές δομές με τις οποίες θα δουλέψουμε είναι:

`lunix_chrdev_state_struct` **`lunix_sensor_struct *sensor`**



lunix-chrdev.c
lunix_chrdev_state_needs_refresh

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct
*state)
{
    debug("refresh_entering\n");
    struct lunix_sensor_struct *sensor;
    WARN_ON ( !(sensor = state->sensor));

    if (!(sensor->msr_data[state->type]->last_update == state->buf_timestamp))
{
        return 1;
    }
    /* The following return is bogus, just for the stub to compile */
    debug("refresh_leaving\n");
    return 0; /* ? */
}
```

lunix_chrdev_state_update

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
    int i, type, refreshed;
    uint32_t data;
    unsigned long flags;
    long dec, fract;
    long data_value;
    long *lookup[N_LUNIX_MSR] = {lookup_voltage, lookup_temperature,
lookup_light};
    unsigned char sign, *temp_buf;
    struct lunix_sensor_struct *sensor;

    debug("update_entering\n");
    sensor = state->sensor;
    WARN_ON(!sensor);
    /* Grab the raw data quickly, hold the
     * spinlock for as little as possible.
     */
    /* Why use spinlocks? See LDD3, p. 119 */
    /*
     * Any new data available?
     */
    /* ? */
    spin_lock_irqsave(&sensor->lock, flags);
    refreshed = lunix_chrdev_state_needs_refresh(state);
    if (refreshed == 1) {
        data = sensor->msr_data[state->type]->values[0];
        state->buf_timestamp = sensor->msr_data[state->type]->last_update;
    }
    spin_unlock_irqrestore(&sensor->lock, flags);

    /* Now we can take our time to format them,
     * holding only the private state semaphore
     */
    if (refreshed == 1) {
        /*data matching*/
        data_value = lookup[state->type][data];
        /*add sign*/
            if (data_value >= 0) {
                sign = '+';
            }
        else {
            sign = '-';
        }
        dec = data_value / 1000;
```

```
        fract = data_value % 1000;
        snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ,"%c%d.%d\n", sign, dec,
fract);
        state->buf_lim = strnlen(state->buf_data, LUNIX_CHRDEV_BUFSZ);
        state->buf_data[state->buf_lim]='\0';
        state->buf_lim = strnlen(state->buf_data, LUNIX_CHRDEV_BUFSZ);
    }
    else if (state->buf_lim > 0) {
        return -EAGAIN;
        }
        else {
            return -ERESTARTSYS;
        }

    debug("update_leaving\n");
    return 0;
}
```

lunix_chrdev_open

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
    /* Declarations */
    /* ? */
    unsigned int minor_num;
    unsigned int sensor_num;
    unsigned int measurement_type;
    int ret;
    //struct lunix_sensor_struct *sensor;
    struct lunix_chrdev_state_struct *state;

    debug("open_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_num = iminor(inode);
    measurement_type = minor_num & 7;
    sensor_num =  (minor_num-measurement_type) >> 3;

    /* Allocate a new Lunix character device private state structure */
    /* ? */
    state = kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);
    if (!state) {
            /* handle error ... */
        debug("Out of memory! State struct allocation failed!\n");
        ret = -ENOMEM;
        goto out;
    }
    /*initializing state struct*/
    state->type = measurement_type;
    state->sensor = &lunix_sensors[sensor_num];
    state->buf_lim = 0;
    //state->buf_data = "";
    state->buf_timestamp = 0;
    sema_init(&state->lock, 1);

    filp->private_data = state;

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

lunix_chrdev_release

```
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
    kfree(filp->private_data);
    return 0;
}
```

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, rem;

    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("read_entering\n");

    /* Lock? */
    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 */
            up(&state->lock);/*release the lock*/
            if (filp->f_flags & O_NONBLOCK) {
                return -EAGAIN;
            }
            debug("reading: going to sleeep\n");
            if (wait_event_interruptible(sensor->wq,
lunix_chrdev_state_needs_refresh(state))) {
                return -ERESTARTSYS;/* signal: tell the fs layer to handle it
*/
            }
            /* otherwise loop, but first reacquire the lock */
            if (down_interruptible(&state->lock)) {
                return -ERESTARTSYS;
            }
        }
    }
    /* ok, data is there, trying to read something */

    /* End of file */
    /* ? */
    if (state->buf_lim == 0) {
        ret = 0;
        goto out;
    }
    /* Determine the number of cached bytes to copy to userspace */
```

```
    /* ? */
    if (*f_pos + cnt > state->buf_lim) {
        cnt = state->buf_lim - *f_pos;
    }

    /*copy_to_user returns how many bytes failed to be copied*/
    if (copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)) {
        ret = -EFAULT;
        goto out;
    }
    *f_pos += cnt;
    ret = cnt;

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

    debug("read_leaving\n");
out:
    /* Unlock? */
    up(&state->lock);
    return ret;
}
```

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;

    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, "lunix");
    if (ret < 0) {
        debug("failed to register region, ret = %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");
        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;
}
```

3 Περιγραφή υλοποίησης των ζητούμενων συναρτήσεων του `lunix-chrdev.c`

- `lunix_chrdev_init`

  Σε αυτή τη συνάρτηση προσθέσαμε μετά την υπόδειξη των σχόλιων του κώδικα τις συναρτήσεις `register_chrdev_region()` και `cdev_add()`. Σκοπός αυτών των συναρτήσεων είναι:

1. `int register_chrdev_region(dev_no, lunix_minor_cnt, "lunix")`
   Η απόκτηση των απαραίτητων major (60 όπως φαίνεται στο `lunix.h`) και minor device numbers που θα χρησιμοποιεί ο οδηγός. Εδώ `dev_no` είναι η δήλωση του major number καθώς και του πρώτου minor number. Ακόμη `lunix_minor_cnt` είναι το εύρος των minor numbers που θα χρησιμοποιήσει η συσκευή, το οποίο σε πλήθος είναι 48 (lunix_sensor_cnt×μέτρηση = 16×3), αλλά το εύρος τους είναι μεγαλύτερο γιατί κάνουμε αριστερή ολίσθηση 3 θέσεις με τη δήλωση δηλαδή ισοδύναμε πολλαπλασιασμό επί 8 – εντολή `unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3`. Τέλος `"lunix"` είναι το όνομα της συσκευής χαρακτήρων που εισάγουμε.

2. `cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt)`
   Η προσθήκη του οδηγού συσκευής χαρακτήρων στον πυρήνα και πιο συγκεκριμένα του `chrdev struct`.

- lunix_chrdev_open

  Σε αυτή τη συνάρτηση γίνεται η συσχέτιση κάθε ανοιχτού αρχείου που αφορά τον οδηγό με τους κατάλληλους major και minor numbers καθώς και η δέσμευση μνήμης (μαζί με κάποιες αρχικοποιήσεις των δεδομένων) του `struct lunix_chrdev_state_struct`. Αυτή τη μνήμη που δεσμεύουμε, φροντίζουμε να την αποθηκεύσουμε στο private_data πεδίο του file struct που αφορά τη συσκευή και αργότερα να την ελευθερώσουμε στη συνάρτηση static int lunix_chrdev_release(struct inode *inode, struct file *filp). Μια παρατήρηση για τα ορίσματα της συνάρτησης `chrdev_open` είναι ότι το `struct inode *inode` αφορά ένα αρχείο στο δίσκο ενώ το `struct file *filp` αφορά κάθε ένα ανοιχτό instance αυτού του αρχείου.

- `lunix_chrdev_state_needs_refresh`

  Σε αυτή τη συνάρτηση ελέγχουμε αν έχουν έρθει νέες μετρήσεις από τους αισθητήρες ελέγχοντας τις μεταβλητές `last_update` του `lunix_msr_data_struct` από το `lunix_sensor_struct` και του `buf_timestamp` του `lunix_chrdev_state_struct`.

- `lunix_chrdev_state_update`

  Σε αυτή τη συνάρτηση ανανεώνουμε τα δεδομένα των μετρήσεων της συσκευής χαρακτήρων όταν αυτά έρθουν ελέγχοντας το αποτέλεσμα της παραπάνω συνάρτησης `lunix_chrdev_state_needs_refresh` και ακόμη τα επεξεργαζόμαστε κατάλληλα πριν τα περάσουμε στο `userspace`.
  Πιο συγκεκριμένα αφού κάνουμε τις απαραίτητες δηλώσεις δεδομένων

που φαίνονται στον παραπάνω κώδικα το πρώτα πράγμα που κάνουμε είναι να πάρουμε τις μετρήσεις από τις δομές (struct) που είναι αποθηκευμένες. Σε αυτό το σημείο πρέπει να αναφέρουμε ότι για να γίνει με επιτυχία αυτή τη άντληση δεδομένων πρέπει να κλειδώσουμε και να ξεκλειδώσουμε το `spinlock_t lock` του `lunix_sensor_struct`. Ο λόγος που χρησιμοποιούμε `spinlocks` είναι και πιο συγκεκριμένα τις συναρτήσεις `spin_lock_irqsave(&sensor->lock, flags)` για το κλείδωμα και `spin_unlock_irqrestore(&sensor->lock, flags)` για το ξεκλείδωμα είναι για να απενεργοποιήσουμε και ενεργοποιήσουμε αντίστοιχα τις διακοπές (interrupts) κατά την επεξεργασία του συγκεκριμένου struct μιας και τα δικά του περιεχόμενα είναι αυτά που επηρεάζονται άμεσα με τον ερχομό μιας νέας μέτρησης δηλαδή ενός interrupt. Ο λόγος που χρησιμοποιούνται τα spinlocks είναι ότι θέλουμε να διαφυλάξομε αδιάβλητες τις μετρήσεις που θα πάρουμε από το sensor struct.

Κατόπιν έχουμε τις εξής περιπτώσεις:

1. Αν έχουμε νέα δεδομένα (`if (refreshed == 1)`) τότε εκτελούμε κατάλληλες μετατροπές με `div 1000` και `mod 1000` καθώς επίσης βρίσκουμε το πρόσημο της μέτρησης (`sign`). Ακόμη περνάμε τα δεδομένα στον buffer του struct lunix_chrdev_state_struct και φροντίζουμε κατάλληλα την παράμετρο buf_lim ώστε να ισούται με το μήκος της μέτρησης που θα περάσουμε στην συνάρτηση read και εν συνεχεία στο userspace.
2. Και τέλος προσαρμόζουμε κατάλληλα το αποτέλεσμα της συνάρτησης σε περίπτωση που δεν έχουμε νέα δεδομένα γιατί μας χρειάζεται στην συνάρτηση `lunix_chrdev_read`.

- `lunix_chrdev_read`

Τελευταία θα εξηγήσουμε τη λειτουργία της `chrdev_read` καθώς η λειτουργία της εξαρτάται από την κλήση των παραπάνω `chrdev_state_update` και `chrdev_state_needs_refresh.`

Πιο συγκεκριμένα στην αρχή κάνουμε κάποιες δηλώσεις βοηθητικών μεταβλητών `ssize_t ret; ssize_t rem;` και στη συνέχεια ξεκλειδώνουμε το `struct semaphore lock`; του `lunix_chrdev_state_struct`. Μετά τις απαραίτητες δηλώσεις καλούμαστε να κάνουμε μια σειρά από ελέγχους. Συγκεκριμένα αρχικά ελέγχουμε αν έχουν έρθει νέα δεδομένα από την `update` και αν όχι η διεργασία μας κοιμάται επαναληπτικά (`wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh (state))`) με συνθήκη αφύπνισης την ύπαρξη νέων δεδομένων (η αφύπνιση γίνεται από τη συνάρτηση `wake_up_interruptible(&s->wq)`; μέσα στην `lunix_sensor_update` του `lunix-sensors.c`).

Στην έξοδο από την επαναληπτική διαδικασία υπάρχουν δεδομένα. Εάν δεν είναι μηδενικά (γίνεται κατάλληλος έλεγχος) τότε αλλάζουμε τις τιμές των `f_pos*` και `cnt` και κάνουμε το ζητούμενο `copy_to_user` ώστε τα δεδομένα στο `buff_data` να περάσουν στο userspace. Η συνάρτηση επιστρέφει των αριθμό των χαρακτήρων που έγιναν `copy_to_user`.

Καθ' όλη την συνάρτηση και την μεταβολή των περιεχομένων του `chrdev_state_struct` ανοιγοκλείνουμε τον σημαφόρο του lock. Αυτό γίνεται γιατί σε περίπτωση εκτέλεσης κάποιου fork από την διεργασία μας δύο διεργασίες θα είχαν πρόσβαση στο ίδιο state_struct και αυτό θα μπορούσε να αποτελέσει πηγή προβλημάτων. Το ίδιο θα μπορούσε να προκύψει σε περίπτωση που μία διεργασία έχει περισσότερα από ένα threads.

4 Συνοπτική περιγραφή σεναρίου εκτέλεσης cat /dev/lunix{dev_nr}-{measurement}

Για την τελικό έλεγχο σωστής λειτουργίας και έλεγχο της εξόδου εκτελούμε την εντολή `cat /dev/lunix0-batt` για να δούμε την τάση της μπαταρίας στον αισθητήρα 0. Χρήση της εντολής αυτής σημαίνει μια σειρά από δράσεις για το λειτουργικό που αφορούν τις δικές μας συναρτήσεις. Αρχικά ο πυρήνας βλέπει ότι η συσκευή είναι character device αφού ως μια τέτοια έχει αρχικοποιηθεί και έτσι γνωρίζει ότι για να την επεξεργαστεί θα πρέπει να εκτελέσει συναρτήσεις που περιέχονται στην char-dev-solution.c που εμείς έχουμε γράψει. Εκτελεί λοιπόν τη δική μας `chrdev_open` και δημιουργεί ένα `chrdev_state_struct` ως ένα instance του inode για την συγκεκριμένη επεξεργασία της συσκευής όπως αυτή καλέστηκε από την `cat`. Εν συνεχεία καλεί τη `read` που χρησιμοποιεί τα πεδία αυτού του `state_struct` για την απόδοση της εξόδου. Αυτή με τη σειρά της καλεί την `update` που κάνει τους αντίστοιχους ελέγχους και περνά στη `read` τα δεδομένα για την έξοδο με όλες τις συναρτήσεις να λειτουργούν όπως εξηγήθηκε παραπάνω. Τελικά η `read` με την `copy_to_user` περνά τα δεδομένα στο userspace τα οποία και βλέπουμε στο terminal (βλ. εικόνα παρακάτω). Η διαδικασία αυτή επαναλαμβάνεται συνεχώς με αποτέλεσμα να βλέπουμε συνεχώς νέες εξόδους όσο έρχονται μετρήσεις μέχρι να σταματήσουμε την `cat` με `Ctrl+C`. Οι έξοδοι που παρουσιάστηκαν φαίνονται παρακάτω: