ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Λειτουργικών Συστημάτων

Lunix:TNG Character Device Driver

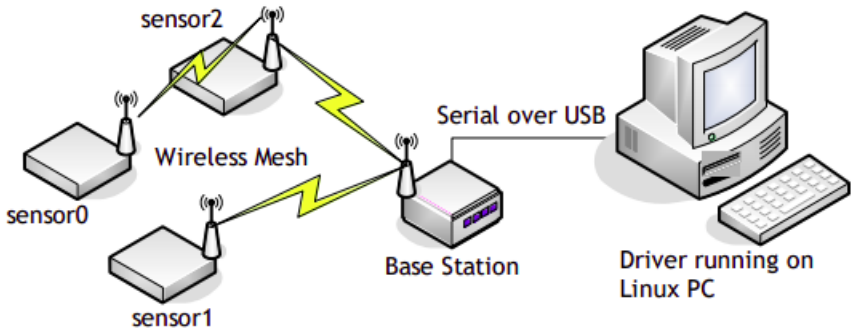
7ο Εξάμηνο - Ακ. Έτος 2019-20

Ξενίας Δημήτριος
Α.Μ.: 03115084

Ειρήνη Γραφιαδέλλη
Α.Μ.: 03114112

1 Εισαγωγή

Στόχος της συγκεκριμένης εργαστηριακής άσκησης είναι η υλοποίηση ενός οδηγού συσκευής για ένα ασύρματο δίκτυο αισθητήρων κάτω από το λειτουργικό σύστημα Linux. Το συγκεκριμένο δίκτυο διαθέτει έναν αριθμό από αισθητήρες και έναν σταθμό βάσης. Ο σταθμός βάσης συνδέεται μέσω USB με υπολογιστικό σύστημα Linux στο οποίο και εκτελείται ο ζητούμενος οδηγός συσκευής Lunix:TNG.



Σχήμα 1: Αρχιτεκτονική του υπό εξέταση συστήματος

Οι αισθητήρες αναφέρουν περιοδικά το αποτέλεσμα τριών διαφορετικών μετρήσεων: της τάσης της μπαταρίας που τους τροφοδοτεί, της θερμοκρασίας και της φωτεινότητας του χώρου που βρίσκονται. Τα δεδομένα μεταφέρονται μέσω ενός δικτύου mesh, από εναλλακτικές διαδρομές, έτσι ώστε το δίκτυο να προσαρμόζεται αυτόματα στην περίπτωση που ένας ή περισσότεροι αισθητήρες είναι εκτός της εμβέλειας του σταθμού βάσης (Σχήμα 1).

Ο σταθμός βάσης λαμβάνει πακέτα με δεδομένα μετρήσεων, τα οποία προωθεί μέσω σειριακής διασύνδεσης USB (serial over USB) στο υπολογιστικό σύστημα Linux, για το οποίο ήδη ο πυρήνας του συστήματος διαθέτει ενσωματωμένους οδηγούς (tty layer). Έτσι τα αποτελέσματα όλων των μετρήσεων όλων των αισθητήρων εμφανίζονται σε μια εικονική σειριακή θύρα, την /dev/ttyUSB1. Με τον τρόπο αυτό δεν υποστηρίζεται η ταυτόχρονη πρόσβαση από πολλές διαφορετικές διεργασίες, αφού μόνο μια έχει νόημα να ανοίγει κάθε φορά την εικονική σειριακή θύρα.

2 Λειτουργικές απαιτήσεις οδηγού συσκευής

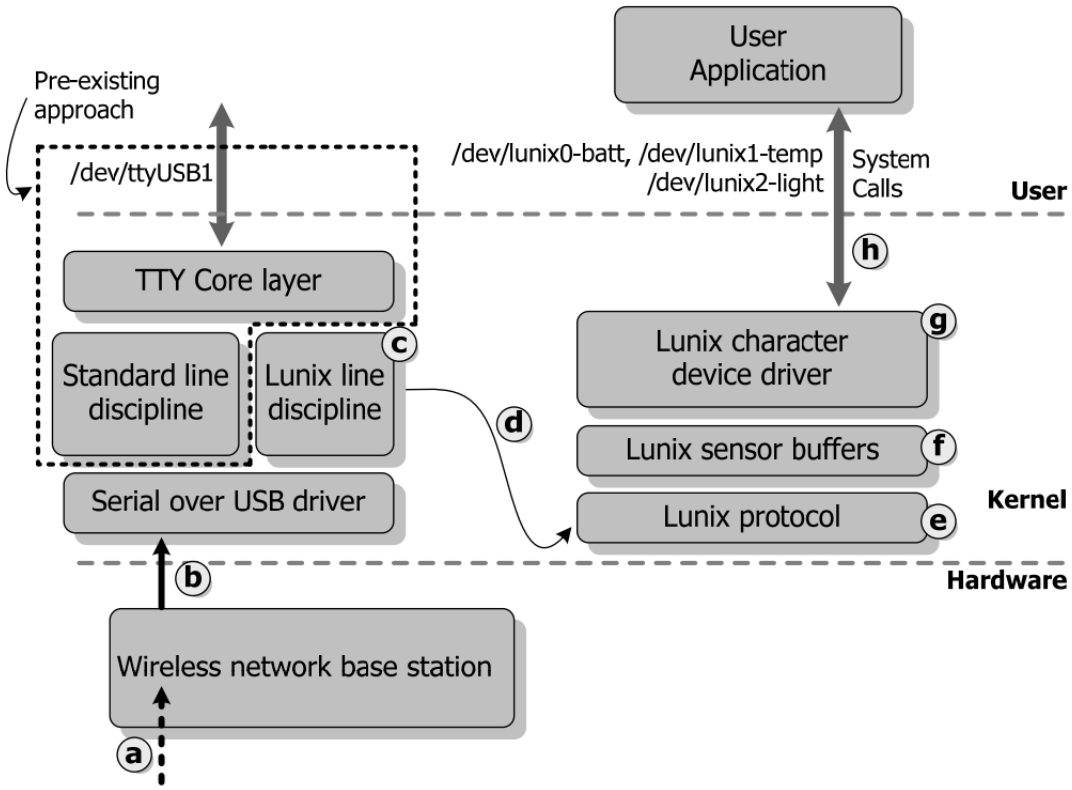
Προκύπτει λοιπόν η ανάγκη για την ανάπτυξη ενός οδηγού συσκευής ο οποίο να προσφέρει τον κατάλληλο μηχανισμό, αντιμετωπίζοντας ανεξάρτητα τους αισθητήρες και τα μετρούμενα μεγέθη, επιτρέποντας ταυτόχρονη πρόσβαση στα εισερχόμενα δεδομένα και κάνοντας δυνατή την επιβολή διαφορετικής πολιτικής από το διαχειριστή του συστήματος όσον αφορά την πρόσβαση σε αυτά.

Συγκεκριμένα, ο ζητούμενος οδηγός συσκευής χρειάζεται να προσφέρει τη δυνατότητα χωριστής πρόσβασης ανά αισθητήρα και μετρούμενο μέγεθος μέσω διαφορετικών ειδικών αρχείων. Η πρώτης προσπάθεια ανάγνωσης από ειδικό αρχείο θα επιστρέφει έναν αριθμό από bytes που θα αναπαριστούν την πιο πρόσφατη τιμή του μετρούμενου μεγέθους, σε αναγνώσιμη μορφή, μορφοποιημένη ως δεκαδικό αριθμό. Αν δεν έχουν

παραληφθεί ακόμα δεδομένα για τη συγκεκριμένη μέτρηση, ή αν γίνουν επόμενες προσπάθειες ανάγνωσης, ο οδηγός πρέπει να κοιμίζει τη διεργασία έως ότου παραληφθούν νέα δεδομένα. Έτσι, αυτή δεν θα καταναλώνει χρόνο στη CPU του συστήματος.

3 Αρχιτεκτονική Οδηγού Συσκευής

Στο Σχήμα 2 παρουσιάζεται η ήδη υπάρχουσα αρχιτεκτονική του tty layer την οποία χρησιμοποιεί ο πυρήνας ώς προεπιλεγμένο οδηγό συσκευής χαρακτήρων για σειριακές θύρες, που όπως προείπαμε κατευθύνει τα σειριακά δεδομένα μέσω USB, στην σειριακή θύρα /dev/ttyUSB1. Παράλληλα βλέπουμε πώς εμείς θα χρησιμοποιήσουμε το ήδη υπάρχον κατώτερο στρώμα του tty layer ώστε να ανακατευθύνουμε τα δεδομένα που μας δίνει σε δικιά μας αρχιτεκτονική, ώστε να γίνει η επεξεργασία και ο διαχωρισμός των δεδομένων σε διαφορετικά ειδικά αρχεία (π.χ. /dev/lunix0-temp)



Σχήμα 2: Αρχιτεκτονική λογισμικού του υπό εξέταση συστήματος

3.1 Τμήμα συλλογής και επεξεργασίας δεδομένων

Βλέπουμε λοιπόν από το Σχήμα 1 ότι ο οδηγός συσκευής χαρακτήρων που υλοποιούμε λαμβάνει αρχικά τα δεδομένα από το κατώτερο επίπεδο του tty layer, το οποίο αναλαμβάνει την απευθείας επικοινωνία με τη σειριακή συσκευή. Στην συνέχεια όμως τα δεδομένα δεν ακολουθούν την προκαθορισμένη πορεία προς το /dev/ttyUSB1. Αντιθέτως, φτιάχνουμε μια δικιά μας διάταξη γραμμής (c) η οποία λαμβάνει τα δεδομένα από τους buffers του κατώτερου στρώματος και τα προωθεί (d) στο πρωτόκολλο του δικού

μας οδηγού (e). Το πρωτόκολλο λαμβάνει τη ροή δεδομένων και εξάγει τελικά τις τιμές των μετρούμενων μεγεθών, οι οποίες και κρατούνται σε κατάλληλες δομές προσωρινής αποθήκευσης στη μνήμη (f). Έτσι λοιπόν έχουμε διαχωρίσει τα συνεχή δεδομένα σε διαφορετικούς buffers στη μνήμη, καθένας από τους οποίους περιέχει τη τελευταία χρονικά μέτρηση για ένα συγκεκριμένο αισθητήρα και για ένα συγκεκριμένο μετρούμενο μέγεθος (π.χ. μπαταρία, φωτεινότητα, θερμοκρασία).

3.2 Τμήμα εξαγωγής δεδομένων στο χώρο χρήστη

Το συγκεκριμένο τμήμα είναι και το τμήμα που μας ζητήθηκε να υλοποιήσουμε στο πλαίσιο της συγκεκριμένης εργασίας, καθώς οι υλοποιήσεις όλων των κατώτερων επιπέδων στη αρχιτεκτονική του οδηγού μάς δόθηκαν έτοιμες. Πρόκειται ουσιαστικά για το nterface μεταξύ του πυρήνα και του οδηγού συσκευής.

Το Linux παρουσιάζει σχεδόν όλες τις συσκευές ως αρχεία. Τα ειδικά αυτά αρχεία συσκευών, μέσω των οποίων είναι δυνατή η αλληλεπίδραση με συσκευές E/E αποτελούν και αυτά μέρος του συστήματος αρχείων (filesystem), όπως και τα αρχεία δεδομένων. Ο οδηγός συσκευής μοιάζει με ένα κανονικό αρχείο το οποίο μπορούμε να ανοίξουμε (open()), να κλείσουμε (close()), να διαβάσουμε (read()), και να γράψουμε. Ο πυρήνας βλέπει όλες αυτές τις λειτουργίες ως ειδικές αιτήσεις και τις μεταφράζει σε κλήσεις προς τον αντίστοιχο οδηγό συσκευής. Στην περίπτωσή μας, που αφορά έναν οδηγό συσκευής χαρακτήρων, υπάρχει σχεδόν 1-1 αντιστοιχία ανάμεσα σε κλήσεις συστήματος και κλήσεις προς τον οδηγό συσκευής.

Έτσι λοιπόν στο αρχείο lunix-chrdev.c υλοποιείται ο οδηγός συσκευής χαρακτήρων, ο οποίος πρέπει να υλοποιεί το interface του πυρήνα για τις κλήσεις συστήματος, αφού αν για παράδειγμα γίνει ένα open() στο ειδικό αρχείο /dev/lunix0-temp, τότε ο πυρήνας θα περάσει την διαδικασία του open() στη συνάρτηση open() του οδηγού συσκευής. Αντίστοιχα, αν μια εφαρμογή χρήστη κάνει read() στο /dev/lunix0-temp, τότε κάνει κλήση συστήματος προς το πυρήνα, ο πυρήνας όντας σε process context οφείλει να υλοποιήσει το read(). Επειδή όμως το read() γίνεται σε ένα ειδικό αρχείο (device file), περνάει τον έλεγχο στον οδηγό συσκευής του συγκεκριμένου αρχείου, ο οποίος είναι και υπεύθυνος για την υλοποιήση της read(). Έτσι εξηγείται γιατί το τμήμα αυτό θεωρείται το interface μεταξύ πυρήνα και οδηγού συσκευής (που είναι βέβαια και αυτός μέρος του πυρήνα). Στη συνέχεια του read(), ο οδηγός συσκευής διαβάζει τα δεδομένα από τους sensor buffers και επιστρέφει στον χρήστη τα δεδομένα που ζήτησε. Αναλυτικότερη παρουσίαση γίνεται στην Ενότητα 4.

4 Υλοποίηση του Ανώτερου Στρώματος του Οδηγού Συσκευής Χαρακτήρων Lunix:TNG

Όπως προείπαμε είναι το τμήμα εξαγωγής δεδομένων στο χώρο χρήστη και είναι το interface του πυρήνα και του οδηγού συσκευής. Όλες λοιπόν οι κλήσεις συστήματος πάνω στο ειδικό αρχείο (device file) /dev/lunix0-temp πρέπει να υλοποιούνται από αυτό το τμήμα. Η υλοποίηση βρίσκεται στο lunix-chrdev.c και κομμάτια του κώδικά του θα παρουσιαστούν στην συνέχεια.

4.1 File Operations

Εδώ γίνεται ο ορισμός της file operations structure, η οποία ουσιαστικά περιέχει pointers στις συναρτήσεις υλοποίησης του interface του πυρήνα που περιέχει ο οδηγός.

```
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
};
```

4.2 Lunix_chrdev_init()

Ο πυρήνας συνδέει τα ειδικά αρχεία με τον εκάστοτε οδηγό τους, μέσω των MAJOR και MINOR Numbers των αρχείων. Η σύνδεση αυτή των αρχείων με τον οδηγό γίνεται στην συνάρτηση **lunix_chrdev_init()** η οποία καλείται από την **lunix_module_init()** που βρίσκεται στο **lunix-module.c**. Η lunix_module_init() εκτελείται απευθείας κάθε φορά που εισάγουμε το module του οδηγού στον πυρήνα μέσω της εντολής **insmod ./lunix.ko** στο bash του συστήματος. Η ακριβής υλοποίησή της παρουσιάζεται παρακάτω:

```
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,"TNG: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;
}
```

Ουσιαστικά βάζουμε στην lunix_chrdev_cdev τα file operations που έχουμε υλοποιήσει, ορίζουμε τον owner της lunix_chrdev_cdev, δεσμεύουμε τα device number (register_chrdev_region()) και τέλος ενημερώνουμε τον πυρήνα για την σύνδεση των device numbers με τον αντίστοιχο οδηγό (cdev_add()), προσθέτοντας στον lunix_chrdev_cdev την περιοχή των device numbers που έχουμε δεσμεύσει.

4.3 Lunix_chrdev_open()

Η συνάρτηση αυτή υλοποιεί το interface του πυρήνα για την κλήση συστήματος open().

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
    /* Declarations */
    /* ? */
    unsigned int minor_number;
    struct lunix_chrdev_state_struct *private_state;
    int type;
    int ret;
    int sensor;
    debug("entering open\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_number = iminor(inode);
    type = minor_number % 8;
    sensor = minor_number / 8;
    if ( type >= N_LUNIX_MSR){
        debug("Minor number represents a type of non-existed measurement\n");
        goto out;
    }
    /* Allocate a new Lunix character device private state structure */
    /* ? */
    private_state = kzalloc(sizeof(*private_state), GFP_KERNEL);
    if(!private_state){
        printk(KERN_ERR "Failed to allocate memory for Lunix sensors\n");
        goto out;
    }
    private_state->type = type;
```

```
   private_state->buf_timestamp=0;
   private_state->buf_lim = 0;
   private_state->sensor = &lunix_sensors[sensor];
   sema_init(&private_state->lock,1);
   filp->private_data = private_state;
out:
   debug("leaving from open with ret = %d\n", ret);
   return ret;
}
```

Ουσιαστικά όταν ανοίγει ένα αρχείο δημιουργείται το struct file μέσα στον πυρήνα, το οποίο αντιπροσωπεύει το ανοιχτό αρχείο και πλέον όλες οι κλήσεις του συστήματος γίνονται πάνω σε αυτό το file. Επομένως εμείς θέλουμε στο αρχείο αυτό στα **private_data** του να αποθηκεύεται η κατάσταση του που αφορά:

- Το ποιος αισθητήρας ”ανήκει” στο file αυτό
- Το ποια μέτρηση αισθητήρα ”ανήκει” στο file αυτό
- Τον buffer που κρατάει τα δεδομένα της συγκεκριμένης μέτρησης αισθητήρα
- Και έναν σεμαφόρο με τον οποίο εξασφαλίζουμε την ατομικότητα των ”πράξεων” στα δεδομένα του state

4.4 Lunix_chredv_state_needs_refresh()

Η συνάρτηση αυτή είναι μια συνάρτηση που απλά ελέγχει εάν έχει προκύψει μια νέα μέτρηση στους buffers του αισθητήρα για την συγκεκριμένη μέτρηση που είναι υπεύθυνο το file. Ως νέα μέτρηση θεωρείται μια μέτρηση που έχει χρονική στιγμή παραλαβής μεταγενέστερη της χρονικής στιγμής της μέτρησης που είναι αποθηκευμένη στον buffer του file. Επομένως η συνάρτηση αυτή είναι μια βοηθητική συνάρτηση που καλείται από τις επόμενες συναρτήσεις και επιστρέφει 1 εάν υπάρχει καινούρια μέτρηση και 0 εάν όχι.

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct
    *state)
{
   struct lunix_sensor_struct *sensor;
   int ret;
   WARN_ON ( !(sensor = state->sensor));
   /* ? */

   /* The following return is bogus, just for the stub to compile */
   printk("Sensor time is %d\n",sensor->msr_data[state->type]->last_update);
   printk("State time is %d\n",state->buf_timestamp);
   if (state->buf_timestamp != sensor->msr_data[state->type]->last_update) ret=1;
   else ret=0;
   debug("need refresh is %d",ret);
   return ret; /* ? */
}
```

4.5 Lunix_chrdev_state_update()

Η συγκεκριμένη συνάρτηση είναι και αυτή βοηθητική συνάρτηση της read() και ουσιαστικά υλοποιεί την ενημέρωση του buffer του file, ώστε να συγχρονιστεί με την καινούρια μέτρηση που έχει προκύψει στον buffer του συγκεκριμένου αισθητήρα. Όταν λοιπόν προκύψει μια καινούρια μέτρηση, τότε δεσμεύει το spinlock του buffer του σένσορα που βρίσκεται στο ακριβώς από κάτω επίπεδο της αρχιτεκτονικής του driver, έτσι ώστε όσο αντλεί τα δεδομένα να μην γίνει μια αλλαγή στον buffer λόγω interrupt context. Αφού αντλήσει τα δεδομένα, τα μορφοποιεί από uint16_t σε ένα long αριθμό μέσω των lookup tables. Αυτό γίνεται διότι τα δεδομένα έρχονται σε 16 bit μορφή και για να μετατραπούν σε μετρήσιμα δεδομένα πρέπει να γίνουν κάποιες πράξεις κινητής υποδιαστολής. Αυτό όμως είναι απαγορευτικό σε kernel mode. Για αυτό χρησιμοποιούμε μία αντιστοίχιση για να εξάγουμε τις τιμές. Τέλος, αφού έχουμε την τιμή μέτρησης, την περνάμε σε έναν πίνακα χαρακτήρων όπου και αποθηκεύεται η πληροφορία στον buffer του file. Εάν δεν υπάρχει καινούρια πληροφορία τότε η συνάρτηση επιστρέφει -EGAIN, αλλιώς 0.

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
   struct lunix_sensor_struct *sensor;
   sensor = state->sensor;
   debug("entering state_update\n");
   uint16_t value;
   uint32_t timestamp;
   long data;
   int ret = -EAGAIN;
   /*
    * Grab the raw data quickly, hold the
    * spinlock for as little as possible.
    */
   /* ? */
   /* Why use spinlocks? See LDD3, p. 119 */

   /*
    * Any new data available?
    */
   /* ? */

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

   /* ? */
   if(lunix_chrdev_state_needs_refresh(state)){
      debug("Updating the state");
      spin_lock(&sensor->lock);
      value = sensor->msr_data[state->type]->values[0];
      timestamp = sensor->msr_data[state->type]->last_update;
      spin_unlock(&sensor->lock);

      state->buf_timestamp = timestamp;
      if (state->type == BATT ) data = lookup_voltage[value];
           else if (state->type == TEMP) data = lookup_temperature[value];
```

```
      else data = lookup_light[value];
      state->buf_lim =
          snprintf(state->buf_data,LUNIX_CHRDEV_BUFSZ,"%ld.%ld\n",data/1000,data %
          1000);
      ret = 0;

   }
   debug("leaving state update\n");
   return ret;
}
```

4.6 Lunix_chrdev_read()

Η συνάρτηση αυτή υλοποιεί την κλήση συστήματος read() και ουσιαστικά είναι υπεύθυνη για να στείλει τα δεδομένα των μετρήσεων στο χώρο χρήστη. Αποτελεί την καρδιά του driver μας. Κάθε φορά που ο χρήστης ζητάει δεδομένα από το device file τότε η read() πιάνει δουλεία. Αρχικά κλειδώνει τον σεμαφόρο που βρίσκεται στα private_data του file ώστε να μπορέσει να διασφαλίσει την ατομικότητα και να αποφευχθούν τα race conditions, όπου 2 ή περισσότερες διεργασίες θέλουν να έχουν πρόσβαση στα ίδια δεδομένα. Επειδή το διάβασμα της read() από τον buffer του file θεωρείται κρίσιμο τμήμα, πρέπει κάθε διεργασία που επιχειρεί διάβασμα στι συγκεκριμένο file να κλειδώνει τον σεμαφόρο του. Αυτό έχει σημασία σε διεργασίες που έχουν σχέση πατέρα-παιδιού, αφού μόνο τότε μια διεργασία παιδί κληρονομεί τα ανοιχτά αρχεία. Εν αντιθέσει δυο διαφορετικές διεργασίες με την open() δημιουργούν δυο διαφορετικά file και επομένως δεν συνδέονται αυτά τα δύο και η ατομικότητα είναι σίγουρη. Ωστόσο εμείς από πλευρά driver δεν ξέρουμε πώς μια διεργασία χώρου χρήστη θα έχει διαμορφωθεί και επομένως πρέπει να κρατάμε σεμαφόρους και να τους κλειδώνουμε όταν γίνονται πράξει στα στοιχεία του private_data κάθε file.

Αφού λοιπόν κλειδώσουμε τον σεμαφόρο είμαστε έτοιμοι να λάβουμε τα δεδομένα. Όμως όπως προείπαμε και στις λειτουργικές απαιτήσεις του οδηγού, μια διεργασία όταν επιθυμεί να διαβάσει από το file μια μέτρηση η οποία όμως έχει ήδη διαβαστεί από την διεργασία και δεν έχει ανανεωθεί, τότε πρέπει να ”κοιμίζουμε” τη διεργασία αυτή. Αυτό μας μειώνει κύκλους cpu που θα χάναμε εάν η διεργασία χρονοδρομολογούταν συνεχώς. Η διεργασία θα ”ξυπνήσει” όταν υπάρξει μια καινούρια μέτρηση. Αυτό ελέγχεται από το αρχείο που ανανεώνει τα περιεχόμενα των buffers των αισθητήρων όταν υπάρξει ανανέωση και ουσιαστικά υλοποιείται σε interrupt context και ξυπνάει όλες τις διεργασίες που περιμένουν στην ουρά για τον συγκεκριμένο αισθητήρα και μέτρηση.

Άρα βλέπουμε στον κώδικα ότι όσο δεν έχουμε καινούρια δεδομένα στους buffers των αισθητήρων, τότε κοιμίζουμε την διεργασία (wait_event_interruptible()). Όταν πλέον η διεργασία θα ξυπνήσει λόγω καινούριας μέτρησης, τότε κλειδώνουμε τον σεμαφόρο και ενημερώνουμε το state του file. Τέλος, περνάμε τα δεδομένα από τον buffer του file στον buffer του χώρου χρήστη, ο οποίος θα πάρει τα δεδομένα αυτά και θα τα εμφανίσει στο standard output.

Εδώ πρέπει να δοθεί προσοχή ότι η αντιγραφή των δεδομένων από χώρο πυρήνα σε χώρο χρήστη μέσω των buffers των δυο επιπέδων γίνεται μέσω της εντολής copy_to_user(). Εφόσον ο πυρήνας έχει απεριόριστη πρόσβαση στη μνήμη, θεωρεί αναξιόπιστο κάθε δεδομένο που παρέχει η διεργασία και πρέπει να ελέγχει όλες τις διευθύνσεις που αυτή περνά ως ορίσματα ώστε να γίνει ασφαλή αντιγραφή δεδομένων.

Τέλος, στη συγκεκριμένη read() φτιάχνουμε έναν μηχανισμό. Συγκεκριμένα μια read() σε χώρο χρήστη μπορεί να ζητάει λιγότερα byte από όσα είναι συνολικά το δεδομένο

στον buffer του file. Επομένως κάθε φορά που ζητάει δεδομένα, εμείς μετράμε τα πόσα δεδομένα δώσαμε. Έτσι την επόμενη φορά που μας ζητήσει κάτι, εμείς θα συνεχίσουμε από το δεδομένο που σταματήσαμε την αποστολή, ούτως ώστε τελικά να λάβει ολόκληρο το πακέτο που ξεκίνησε να διαβάζει, όση ώρα και αν μας πάρει αυτό. Μόνο όταν διαβάσει ολόκληρο το πακέτο μέτρησης, θα προχωρήσει στην ενημέρωση του state.

```
static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t
    cnt, loff_t *f_pos)
{
   ssize_t ret;
   int bytes_to_send;
   int 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);
   debug("entering read\n");
   /* Lock? */
   /*
    * 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 (down_interruptible(&state->lock)){
     return -ERESTARTSYS;
   }
   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);
       if (
           wait_event_interruptible(sensor->wq,lunix_chrdev_state_needs_refresh(state))){
         debug("BLocking the process failed\n");
         return -ERESTARTSYS;
       }
       if (down_interruptible(&state->lock)){
         return -ERESTARTSYS;
       }
     }
   }
   /* End of file */
   /* ? */

   /* Determine the number of cached bytes to copy to userspace */
   /* ? */
   //count = *f_pos;
   //debug("hello\n");
```

```
   remaining_bytes = state->buf_lim - *f_pos;
   if (cnt > remaining_bytes) bytes_to_send = remaining_bytes;
   else bytes_to_send = cnt;
   if (copy_to_user(usrbuf, state->buf_data, bytes_to_send)){
      debug("Failed to copy to user\n");
      ret = -EFAULT;
      goto out;
   }
   ret = bytes_to_send;
   /* Auto-rewind on EOF mode? */
   /* ? */
   *f_pos += bytes_to_send;
   if (*f_pos==state->buf_lim) *f_pos=0;

out:
   /* Unlock? */
   up(&state->lock);
   debug("Leaving from read with ret %ld\n",ret);
   return ret;
}
```

4.7 Lunix_chrdev_destroy()

Η συνάρτηση αυτή καλείται όταν το module αποσύρεται από τον πυρήνα μέσω της εντολής rmmod lunix.ko στο bash του συστήματος και επομένως ευθύνη της είναι η διαγραφεί των δεδομένων που έχουν δημιουργηθεί με εντολές δυναμικής δέσμευσης μνήμης.

```
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");
}
```