ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΛΕΙΤΟΥΡΓΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

(2020-2021)

2η Εργαστηριακή Άσκηση

"Οδηγός Ασύρματου Δικτύου Αισθητήρων στο Λειτουργικό Σύστημα Linux"

Ημερομηνία Επίδειξης:

- 3/12/2020 | 12:45 – 13:45

Ομάδα:

- 40

Ονοματεπώνυμο, Α.Μ., στοιχεία επικοινωνίας:

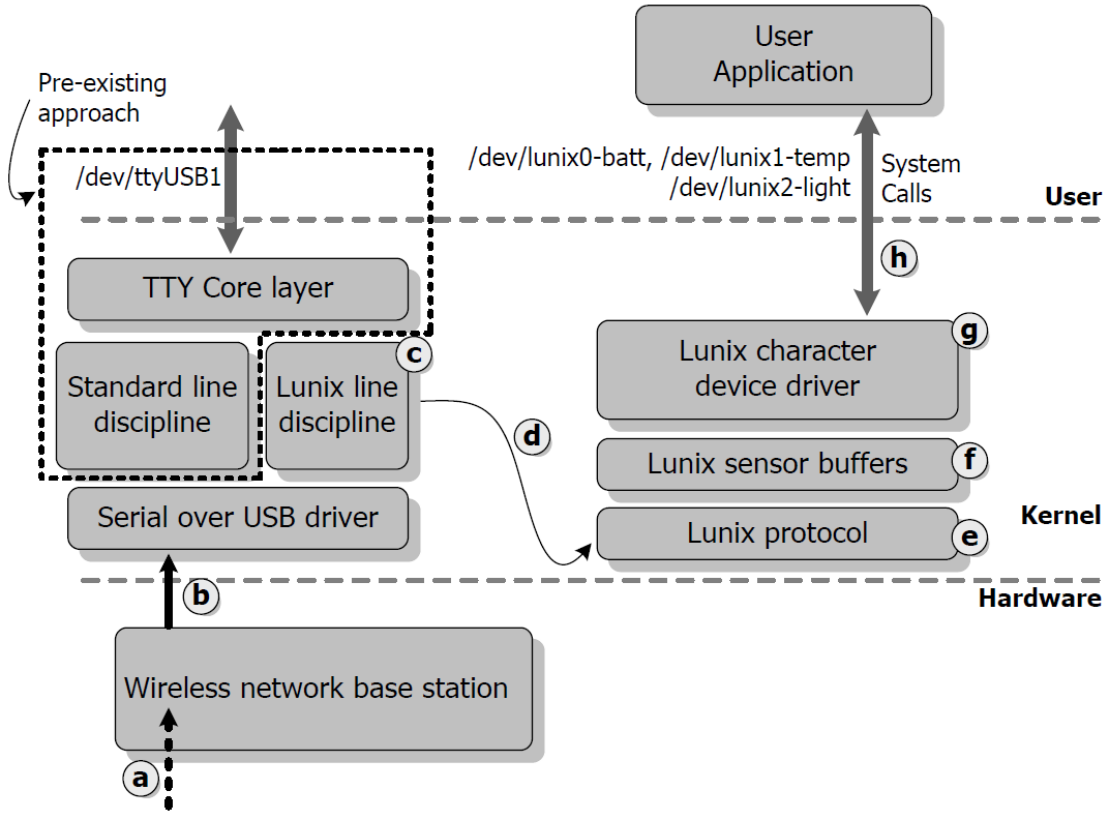
- Τόφαλος Φίλιππος
  - 03117087
  - el17087@mail.ntua.gr
- Χρήστος Τσούφης
  - 03117176
  - el17176@mail.ntua.gr

Εισαγωγή

Στη συγκεκριμένη εργαστηριακή άσκηση ζητείται η υλοποίηση ενός οδηγού συσκευής (driver) για ένα ασύρματο δίκτυο αισθητήρων υπό το λειτουργικό σύστημα `Linux`. Το δίκτυο αυτό αποτελείται από επιμέρους αισθητήρες σε συνδεσμολογία mesh ώστε όλοι οι αισθητήρες να αποδίδουν δεδομένα στον κεντρικό σταθμό βάσης. Κάθε ένας από αυτούς λαμβάνει μετρήσεις για τρία μεγέθη, τη φωτεινότητα, την τάση της μπαταρίας και τη θερμοκρασία περιβάλλοντος. Κατά την διάρκεια της ανάπτυξης του οδηγού, χρησιμοποιήσαμε εικονικό μηχάνημα `QEMU-KVM` το οποίο αρχικοποιείται με το βοηθητικό utopia.sh bash script που μας δίνεται. Ο σταθμός βάσης λαμβάνει τα πακέτα από τους αισθητήρες και τα αποστέλλει μέσω USB στο εργαστήριο. Η διασύνδεση αυτή υλοποιείται με κύκλωμα serial over USB.

Επειδή δεν είναι δυνατό να διαθέτει κάθε ένας ένα τέτοιο δίκτυο αισθητήρων, στο εργαστήριο της σχολής υλοποιείται μια τέτοια διάταξη και τα αποτελέσματα όλων των μετρήσεων εμφανίζονται μαζικά στην εικονική σειριακή συσκευή `/dev/ttyUSB1` του υπολογιστή του εργαστηρίου. Στον ίδιο υπολογιστή εγκαθίσταται TCP/IP server ο οποίος μεταδίδει τις μετρήσεις και στη συνέχεια, μέσω του utopia.sh, όλες αυτές ανακατευθύνονται στη S0 σειριακή θύρα του `QEMU` μηχανήματος (`dev/ttyS0`). Τελικό ζητούμενο αποτελεί η υλοποίηση συστήματος εξαγωγής των δεδομένων από την σειριακή θύρα σε ένα σύνολο από επιμέρους συσκευές χαρακτήρων, για τις οποίες έχουμε ορίσει σύμβαση για την ονομασία τους, οι οποίες παράγονται από το βοηθητικό `script lunix_dev_nodes.sh` σύμφωνα με την ίδια σύμβαση.

Πιο εποπτικά, η υλοποίηση είναι η παρακάτω:



Περιγραφή & Ανάλυση Κώδικα

Έστω ότι έχει γίνει ήδη η υλοποίηση του οδηγού και πρέπει να φορτώσει ο πυρήνας με την εντολή `insmod()`. Αυτή πυροδοτεί την εκτέλεση της συνάρτησης `lunix_module_init()` που υλοποιείται στο αρχείο `lunix-module.c` και εκτελείται λόγω της εκτέλεσης της εντολής `module_init(lunix_module_init)` η οποία ορίζει ποια συνάρτηση θα κληθεί κατά τη φόρτωση. Η `lunix_module_init` είναι υπεύθυνη για ορισμένες λειτουργίες αρχικοποίησης του οδηγού χαρακτήρων και των απαραίτητων δομών του. Συγκεκριμένα:

```
lunix_sensors = kzalloc(sizeof(*lunix_sensors) * lunix_sensor_cnt, GFP_KERNEL);
```

Δέσμευση μνήμης για τις δομές sensor που αποθηκεύουν τα δεδομένα από το `line discipline` (στάδιο f σχήμα 1). Ο πίνακας `lunix_sensors[]` είναι global για κάθε αρχείο της υλοποίησής και δηλώνεται στο αρχείο επικεφαλίδας `lunix.h` .

Έπειτα, η παρακάτω συνάρτηση

```
lunix_protocol_init(&lunix_protocol_state);
```

υλοποιείται στο αρχείο `lunix_protocol.c` και ευθύνεται για την αρχικοποίηση της μηχανής καταστάσεων του πρωτοκόλλου του οδηγού.

```
    /* Initialize all sensors. On exit, si_done is the index of the last
     * successfully initialized sensor.
     */
    for (si_done = -1; si_done < lunix_sensor_cnt - 1; si_done++) {
        debug("initializing sensor %d\n", si_done + 1);
        ret = lunix_sensor_init(&lunix_sensors[si_done + 1]);
        debug("initialized sensor %d, ret = %d\n", si_done + 1, ret);
        if (ret < 0) {
            goto out_with_sensors;
        }
    }
    /*
     * Initialize the Lunix line discipline
     */
    if ((ret = lunix_ldisc_init()) < 0)
        goto out_with_sensors;

    /*
     * Initialize the Lunix character device
     */
    if ((ret = lunix_chrdev_init()) < 0)
        goto out_with_ldisc;

    return 0;
```

Επίσης, γίνεται αρχικοποίηση τιμών των αισθητήρων (`lunix_sensor_struct`), της διάταξης γραμμής καθώς και κλήση της `lunix_chrdev_init()` η οποία υλοποιείται στο αρχείο `lunix-chrdev.c`.

Ανάλυση του κώδικα στο αρχείο `lunix-chrdev.c`

init()

Αρχικά, καλείται η `lunix_chrdev_init()`. Ο κώδικάς της ακολουθεί:

```
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;
    /* For every sensor we want at least 3 minor numbers,
       the measurement info is contained at the 3 LSB
       of the minor number, and the region must be
       consecutive, so we end up with 16 << 3 minor numbers */
    unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

    debug("initializing character device\n");

    /* We initialize the global cdev structure, specifying
       the file operations right above to be used */
    cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
    lunix_chrdev_cdev.owner = THIS_MODULE;

    /* Produce a Device ID for the pair (Major = 60, Minor = 0) */
    dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);

    /* ! */
    /* register_chrdev_region? */
    /* We register the wanted range, starting from (Major = 60, Minor = 0), up
       to (Major = 60, Minor = 16 << 3) */
    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? */
    /* After the above registration, we are ready to add the
       char device for the corresponding cdev structure and
       the defined range */
    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;
}
```

Αυτή η συνάρτηση καλείται μόνο μια φορά κατά την εκτέλεση της εντολής `insmod` που εισάγει ένα καινούριο `module` στον κώδικα του πυρήνα. Όσον αφορά τις συσκευές χαρακτήρων και την ονομασία τους, όπως αναφέρθηκε και παραπάνω το `script lunix_dev_nodes.sh` είναι υπεύθυνο για την δημιουργία των αρχείων που χρησιμοποιήθηκε για την ανάγνωση των μετρήσεων. Πιο συγκεκριμένα για κάθε `sensor` (16 στο σύνολο), δημιουργούνται τρία αρχεία τα οποία αναφέρονται στη μέτρηση μπαταρίας, θερμοκρασίας και φωτεινότητας αντίστοιχα. Το `identifier` των αρχείων αυτών είναι το `major` και το `minor number` τους. Το `major number` χρησιμοποιείται από τον `kernel` για να προσδιορίσει τον οδηγό συσκευής που αφορά ο κόμβος αυτός. Επιλέγεται ο `major number 60` ο οποίος είναι δεσμευμένος για πειραματική χρήση. Για τον `minor number` επιλέγεται η τιμή : `minor` = αισθητήρας * 8 + μέτρηση, όπου μέτρηση = τάση μπαταρίας(0), θερμοκρασία(1) και φωτεινότητα(2).

Πρώτα, μέσω της κλήσης `cdev_init()` αρχικοποιείται η συσκευή χαρακτήρων με τη αντίστοιχη δομή των `file operations`. Στη συνέχεια δεσμεύεται στον πυρήνα ένα `range` από `minor numbers`, συγκεκριμένα απαιτούνται 16 αισθητήρες με οχτώ μετρήσεις ο καθένας οπότε 128 `minor numbers`. Τέλος ενεργοποιούνται αυτές τις συσκευές μέσω της `cdev_add()` κλήσης.

open()

Έπειτα, γίνεται κλήση `open` σε ειδικό αρχείο `dev/lunix{x}-{batt, temp, light}`. Κατά το άνοιγμα ενός τέτοιου αρχείου καλείται η αντίστοιχη, από το `struct lunix_chrdev_fops`, συνάρτηση, δηλαδή η `lunix_chrdev_open()`. Ο κώδικάς της δίνεται παρακάτω:

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
    /* Declarations */
    /* ! */
    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>]
     */

    /* Allocate a new Lunix character device private state structure */
    /* ! */
    /* We perform the allocation using kmalloc */
    state = kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);
    if (!state) {
        /* If the allocation failed, return the suitable errno value */
        ret = -ENOMEM;
        goto out;
    }

    /* TYPE: We acquire the type of measurement from the
     * last 3 bits of the minor number */
    state->type = iminor(inode) & 0b111;
    /* SENSOR: Associate the file with the corresponding
     * sensor struct defined in lunix.h */
    state->sensor = &lunix_sensors[iminor(inode) >> 3];
    /* BUF_LIM: Initially no measurement is cahced.
     * This is not really necessary */
    state->buf_lim = 0;
    /* LOCK: Initialize the semaphore with value equal
     * to 1. This is the same as init_MUTEX */
    sema_init(&state->lock, 1);
    /* BUF_TIMESTAMP: It is important to be initialized
     * to 0, because the sensor MSR DATA is initialized
     * as a zeroed page */
    state->buf_timestamp = 0;
```

```
    state->switch_raw = 0;

    /* Assign the state structure to the private_data field
     * of the given file pointer */
    filp->private_data = state;
out:
    debug("leaving, with ret = %d\n", ret);
    return ret;
}
```

Όπως φαίνεται παραπάνω, η συγκεκριμένη συνάρτηση ευθύνεται για τα εξής:

- Δέσμευση χώρου μνήμης για το `lunix_chrdev_state_struct` που αποτελεί `buffer` με την τελευταία μέτρηση και αντιστοιχεί σε κάθε νέο άνοιγμα αρχείου προκειμένου να καθίσταται δυνατή η προσπέλαση του ίδιου αισθητήρα από διαφορετικές διεργασίες.

```
state = kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);
```

- Αρχικοποίηση τιμών του state σχετικών με το είδος του αισθητήρα, του σημαφόρου για παράλληλη πρόσβαση στο `struct` και αντιστοίχιση του με τον κατάλληλο `sensor buffer`.

```
state->type = iminor(inode) & 0b111;
/* SENSOR: Associate the file with the corresponding
 * sensor struct defined in lunix.h */
state->sensor = &lunix_sensors[iminor(inode) >> 3];
/* BUF_LIM: Initially no measurement is cahced.
 * This is not really necessary */
state->buf_lim = 0;
/* LOCK: Initialize the semaphore with value equal
 * to 1. This is the same as init_MUTEX */
sema_init(&state->lock, 1);
/* BUF_TIMESTAMP: It is important to be initialized
 * to 0, because the sensor MSR DATA is initialized
 * as a zeroed page */
state->buf_timestamp = 0;
state->switch_raw = 0;
```

- Αντιστοίχιση του ανοιχτού αρχείου που δίνεται από τον δείκτη παράμετρο `*filp` με το κατάλληλο `state` (και κατ' επέκταση `sensor`) μέσω του πεδίου `private_data` για μελλοντική χρήση χωρίς να απαιτείται η αναζήτηση των `major` και `minor numbers`.

```
/* Assign the state structure to the private_data field
 * of the given file pointer */
filp->private_data = state;
```

read()

Στη συνέχεια καλείται η `read()` επί του αρχείου, η οποία μέσω της δομής `lunix_chrdev_fops` αντιστοιχεί στην συνάρτηση `lunix_chrdev_read()`. Η συγκεκριμένη συνάρτηση αποτελεί τον κύριο κορμό της υλοποίησης. Ο κώδικάς της δίνεται παρακάτω:

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

    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? */
    /* We try to acquire the semaphore (used in case many processes
     * try to read the same sensor measurement) */
    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) {
            /* ! */
            /* "Release" the semaphore before sleeping */
            up(&state->lock);
            /* If non-blocking is requested by the process,
             * we return a -EAGAIN errno value and avoid
             * putting the process to sleep */
            if (filp->f_flags & O_NONBLOCK)
                return -EAGAIN;
            /* Add the process to the waiting queue while no new data is
             * available */
            /* LINK: https://stackoverflow.com/questions/9254395/ */

            if (wait_event_interruptible(sensor-
>wq, lunix_chrdev_state_needs_refresh(state)))
```

```
                return -ERESTARTSYS;

            /* Re-acquire the lock before continuing */
            if (down_interruptible(&state->lock))
                return -ERESTARTSYS;
            /* The process needs to sleep */
            /* See LDD3, page 153 for a hint */
        }
        /* Σε αυτό το σημείο είχε προστεθεί μια ανούσια κλήση lunix_chrdev_state_upd
ate(state)
         * η οποία δεν χρειαζόταν εφόσον στην συνθήκη της while γίνεται η ανανέωση *
/
    }
    /* End of file */
    /* ! */
    /* We initially assume that zero bytes are going to be read */
    ret = 0;
    /* Unlikely: If the f_pos value exceeds the buf_lim (what we
     * actually have to copy), reset it and return */
    if (*f_pos > state->buf_lim) {
        *f_pos = 0;
        goto out;
    }

    /* Determine the number of cached bytes to copy to userspace */
    /* ! */
    /* If more bytes than those available are requested, redefine
     * the cnt value */
    if (*f_pos + cnt > state->buf_lim)
        cnt = state->buf_lim - *f_pos;

    /* cnt indicates how many bytes will be transfered to the user */
    ret = cnt;

    if (copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)) {
        /* Νομίζοντας ότι είχα αφήσει το return -EFAULT, είχα
         * μια εντολή up(&state->lock);. Κάνοντας την αλλαγή σε goto
         * αυτό αποτελούσε πρόβλημα γιατί θα τρέχαμε ξανά την ίδια
         * εντολή πριν την return */
        /* Indicate that a bad address was given */
        ret = -EFAULT;
        goto out;
    }

    /* Increase the f_pos by cnt for it to be ready at the next call */
    *f_pos += cnt;

    /* Auto-rewind on EOF mode? */
```

```
    /* ! */
    /* If we reached at the end of the buffer, reset f_pos */
    if (*f_pos == state->buf_lim)
        *f_pos = 0;
out:
    /* ! */
    /* Unlock? */
    /* Release the seamphore to be used by the next process
     * probably waiting */
    up(&state->lock);
    return ret;
}
```

Αρχικά, λαμβάνεται ο state buffer από το πεδίο `private_data` του `filp` και από αυτόν τον αντίστοιχο `sensor struct` και τον τύπο του αισθητήρα. Στη συνέχεια, λαμβάνεται το `lock` του `state buffer` ώστε να τροποποιηθεί κατάλληλα με τη νεότερη τιμή, αν αυτή υπάρχει, με χρήση της `down_interruptible()` ώστε να μπορεί να ξυπνήσει η διεργασία και με σήματα. Στη συνέχεια ελέγχεται η θέση του δείκτη `f_pos` ώστε να μάθει σε ποιο σημείο σταμάτησε η προηγούμενη μέτρηση. Αν αυτή δεν είναι μηδενική, τότε υπάρχουν ήδη δεδομένα στο πεδίο `state->buf_data` οπότε επιστρέφουν αυτά φροντίζοντας να είναι γνωστό το κατάλληλο `offset` του `string` από το οποίο θα διαβάσει. Αν πάλι ξεκινάει καινούρια μέτρηση, ελέγχεται αν ο `state buffer` έχει τα πιο πρόσφατα δεδομένα. Πιο συγκεκριμένα χρησιμοποιείται η `lunix_chrdev_state_update()` η οποία, αν δεν υπάρχουν νέα δεδομένα, δηλαδή δεν έγινε κάποια αλλαγή στον `state buffer`, επιστρέφει `-EAGAIN`. Περιμένοντας δεδομένα, θα πρέπει να κοιμίζεται η διεργασία που ζητά δεδομένα μη διαθέσιμα, οπότε και χρησιμοποιείται η εξής κλήση:

```
if (wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state)))
                return -ERESTARTSYS;
```

Αναλυτικά, τοποθετείται στην ουρά αναμονής κάθε αισθητήρα η τρέχουσα διεργασία μέχρι να ικανοποιηθεί η λογική έκφραση της δεύτερης παραμέτρου. Το `lunix_chrdev_state_needs_refresh(state)` αποτελεί ένα `shortcut` που ελέγχει, όπως θα περιγραφεί και στη συνέχεια, αν χρειάζεται να γίνει `update`, δηλαδή αν ο `sensor buffer` έχει πιο πρόσφατα δεδομένα από τον `state buffer`. Όταν τελικά βρεθούν νέα δεδομένα, τότε επιστρέφεται `-ERESTARTSYS` ώστε να επανεκκινήσει την `read()` και να λάβει τα νέα δεδομένα. Χρησιμοποιείται η interruptible έκδοση της εντολής ώστε να υπάρχει η δυνατότητα αποστολής σημάτων κατά τη διάρκεια του `waiting` της διεργασίας (π.χ. Ctrl + C για τερματισμό της διεργασίας).

Αφού τελικά ληφθούν τα νεότερα δεδομένα, πλέον υπολογίζεται το `offset` από το οποίο θα ξεκινήσει η επιστροφή χαρακτήρων του `state->buf_data` και στη συνέχεια εκτελείται η `copy_to_user()` ώστε να μεταφερθούν σε `userspace` χώρο μνήμης οι επιθυμητοί χαρακτήρες. Παράλληλα, φροντίζεται να δοθεί η σωστή τιμή στο `*f_pos` ώστε επόμενες αναγνώσεις παιδιών που μοιράζονται το ίδιο `file struct` να ξεκινήσουν από το σωστό σημείο ενώ επιστρέφεται ο αριθμός των bytes που τελικά διαβάστηκαν (`ret`).

update()

Μετά, η `lunix_chrdev_read` καλεί την `lunix_chrdev_state_update` η οποία είναι υπεύθυνη για την ενημέρωση των τιμών των μετρήσεων από τους αισθητήρες καθώς και την μορφοποίηση των δεδομένων έτσι ώστε να μπορούν να διαβάζονται σε μορφή χαρακτήρων. Επειδή ακριβώς δύναται να αλλάζει την κατάσταση των `buffers` πρέπει να ληφθεί υπόψιν ο αποκλεισμός οποιουδήποτε `race condition`. Για αυτόν τον λόγο χρησιμοποιείται εντός της συνάρτησης το κλείδωμα για το `sensor struct` ενώ κατά τη καλούσα `read()` το κλείδωμα για το `state buffer()`. Σχετικά με το κλείδωμα, υλοποιείται το πρόγραμμα χρησιμοποιώντας κλήσεις `spin_*`. Αυτό θα μπορούσε να δημιουργήσει ζητήματα σε περίπτωση που έρθουν δύο συνεχόμενες διακοπές από το υλικό χωρίς να έχει τερματίσει η πρώτη. Το γεγονός αυτό αντιμετωπίζεται με τη χρήση κλήσεων `spin_lock_irqsave` και `spin_unlock_irqsave` ώστε να μην κολλήσει ποτέ το πρόγραμμα με `interrupt` που δεν μπορεί να εξυπηρετηθεί λόγω των `spinlocks`. Αν η συνάρτηση κάνει πράγματι `update` την τιμή της μέτρησης, επιστρέφει 1 αλλιώς αν δεν χρειάζεται `update`, επιστρέφει `-EAGAIN`. Ακολουθεί ο κώδικας της `lunix_chrdev_update()`:

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
    struct lunix_sensor_struct *sensor;
    /* ! */
    uint16_t value;
    long int temp = 0;
    unsigned long flags;

    debug("leaving\n");
    /*
     * Grab the raw data quickly, hold the
     * spinlock for as little as possible.
     */

    /* ! */
    WARN_ON ( !(sensor = state->sensor) );
    /* Acquire the sensor lock */
    /* NOTE: We use spin_lock_irqsave() beacuse it is possible that new data
     * may be sent from the same sensor, causing an interrupt and putting this
     * function on hold while we call the sequence: lunix_ldisc_receive() ->
     * lunix_protocol_received_buf() -> lunix_protocol_update_sensors() ->
     * lunix_sensor_update() and thus trying to lock again the sensor
     * spinlock. In practice, this is not that possible, because the frequency
     * with which new data is sent from the same sensor is low enough to not
     * cause this sort of aformentioned trouble */
    spin_lock_irqsave(&sensor->lock, flags);

    /* Why use spinlocks? See LDD3, p. 119 */
    /*
     * Any new data available?
     */
```

```
    /* ! */
    /* We examine the result of lunix_chrdev_state_needs_refresh() */
    if (lunix_chrdev_state_needs_refresh(state)) {
        /* Grab the value for the specific sensor */
        value = sensor->msr_data[state->type]->values[0];
        /* Update the timestamp (while locked) */
        state->buf_timestamp = sensor->msr_data[state->type]->last_update;
    } else {
        /* We must not forget to unlock the spinlock */
        spin_unlock_irqrestore(&sensor->lock, flags);
        /* We return the value indicated by lunix_chrdev_read()
         * EAGAIN = "there is no data available right now, try
         * again later" */
        return -EAGAIN;
    }

    /* Restore the sensor lock */
    spin_unlock_irqrestore(&sensor->lock, flags);

    /*
     * Now we can take our time to format them,
     * holding only the private state semaphore
     */
    /* Look at the lookup table in lunix-lookup.h, for the
     * measurement specified by state->type */
    if (state->switch_raw) {
        temp = value;
        goto out;
    }

    switch (state->type) {
        case BATT  : temp = lookup_voltage[value];  break;
        case TEMP  : temp = lookup_temperature[value];  break;
        case LIGHT : temp = lookup_light[value];    break;
        case N_LUNIX_MSR : /*This case is unlikely to occur */ ;
         /* default    : [IT MAY BE BETTER TO RETURN WITH AN ERROR] */
    }

out:    /* Format the value acquired from the lookup table. If it is a LIGHT
     * value, no decimal point is required */
    format_value(state, temp, state->type == LIGHT || state-
>switch_raw ? 0 : DEFAULT_DOT_POS);
    /* Εδώ είχε γίνει λάθος τοποθέτηση της εντολής
     * state->buf_timestamp = sensor->msr_data[state->type]->last_update;
     * αντί για την τοποθέτηση στο critical path που ορίζει το spinlock */
    debug("leaving\n");
    return 0;
}
```

Κρατώντας το `state` κλείδωμα, αρχικά ελέγχεται μέσω της συνάρτησης `lunix_chrdev_state_needs_refresh()` αν πράγματι υπάρχει κάποια καινούργια μέτρηση στον αισθητήρα. Ο τρόπος με τον οποίο γίνεται αυτός ο έλεγχος αναλύεται παρακάτω. Δεδομένου ότι υπάρχει μία νέα μέτρηση, λαμβάνεται το sensor κλείδωμα και μπαίνει σε κομμάτι κρίσιμου κώδικα (`spin_lock_irqsave(&sensor->lock)`). Στη συνέχεια αποθηκεύονται οι τρέχουσες τιμές του αισθητήρα για τον οποίο υπάρχει νέα μέτρηση:

```
/* Grab the value for the specific sensor */
value = sensor->msr_data[state->type]->values[0];
```

Έπειτα η διαδικασία μορφοποίησης των δεδομένων και αποθήκευσης τους στο `state` ξεκινά με το `timestamps` της τελευταίας ενημέρωσης:

```
/* Update the timestamp (while locked) */
state->buf_timestamp = sensor->msr_data[state->type]->last_update;
```

Αφού γίνουν τα παραπάνω, φεύγει από το κρίσιμο κομμάτι κώδικα οπότε καλείται η `spin_unlock_irqrestore(&sensor->lock)` και ανάλογα με το είδος της μέτρησης που πήρε, ανατρέχει στο αντίστοιχο `lookup_table`. Τα `lookup_tables` που ορίζονται στο `lunix-lookup.h` χρησιμοποιούνται προκειμένου να μετατραπούν οι τιμές των μετρήσεων σε `human readable` μορφή. Αφού ληφθεί η τιμή, μορφοποιείται με την συνάρτηση `format_value`:

```
static void format_value (
    /* The state struct to update */ struct lunix_chrdev_state_struct *state,
    /* Value to convert to string and format */ long int value,
    /* Dot position, starting from the most right digit */  int dot
    ) {
    int i, s;
    /* Copy the value from the lookup table into the buffer */
    s = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%ld", value);
    /* Position the decimal point if needed */
    if (dot != 0) {
        for (i = 0; i < dot; i++)
            state->buf_data[s - i] = state->buf_data[s - i - 1];
        state->buf_data[s - i] = '.';
    }
    /* Add the new line character at the end */
    state->buf_data[s + (dot != 0)] = '\n';
    /* Update the buffer limit to be ready for copy_to_user() */
    state->buf_lim = s + 1 + (dot != 0);
}
```

refresh()

Ακολούθως, παρουσιάζεται η `lunix_chrdev_state_needs_refresh`. Αυτή καλείται από την `state_update` και σκοπό έχει να αποφανθεί αν υπάρχει καινούργια μέτρηση για τον αισθητήρα. Για να το πετύχει αυτό συγκρίνονται τα δύο πεδία `timestamps sensor->msr_data[state->type]->last_update` και `state->buf_timestamp`. Τα πεδία αυτά περιέχουν τη πληροφορία για την ώρα μέτρησης της κάθε μέτρησης σε μία πολύ συνηθισμένη αριθμητική μορφή για το λειτουργικό σύστημα `Linux`, το `Linux epoch`, δηλαδή την διαφορά σε `seconds` από την 1η Ιανουαρίου του 1970. Η υλοποίηση:

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
{
    /* We declare the (boolean) return value */
    int ret;
    struct lunix_sensor_struct *sensor;
    WARN_ON ( !(sensor = state->sensor) );
    /* We simply compare the sensor timestamp with the most recent one stored
       in the measurement state struct */
    ret = (state->buf_timestamp != sensor->msr_data[state->type]->last_update);
    /* The following return was bogus but not anymore but equal to the result of the
       above comparison */
    return ret;
}
```

release()

Ύστερα, παρουσιάζεται και η `release` συνάρτηση η οποία απελευθερώνει απλά το `state` αφού για τα υπόλοιπα αναλαμβάνουν άλλες συναρτήσεις του `lunix-module.c`.

```
static int lunix_chrdev_release(struct inode *inode, struct file *filp){
    /* Deallocate the memory used for the private_data structure */
    kfree(filp->private_data);
    return 0;
}
```

destroy()

Η υλοποίησή της δίνεται έτοιμη και είναι η παρακάτω:

```
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");
}
```

Για λόγους πληρότητας δίνεται μια συνοπτική περιγραφή της. Αρχικά παίρνει το `dev_number` της συσκευής που είναι επιθυμητό να αφαιρεθεί και έπειτα καλείται η συνάρτηση `cdev_del()` η οποία αφαιρεί τη συσκευή χαρακτήρων που της δίνεται ως όρισμα (στην πράξη της δίνεται το `cdev struct` που πρέπει να αφαιρεθεί). Στο σημείο αυτό θα πρέπει να κληθεί και η συνάρτηση `unregister_chrdev_region()` για να αποδεσμεύσει τον χώρο που είχε δεσμευτεί για τους αριθμούς με τους οποίους θα γινόταν αναφορά στις αντίστοιχες συσκευές.

Επεκτάσεις

ioctl()

Από τις επεκτάσεις, έχει υλοποιηθεί η `lunix_chrdev_ioctl`. Ο κώδικάς της είναι ο ακόλουθος:

```
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long ar
g)
{
    int retval = 0;
    struct lunix_chrdev_state_struct *state;
    state = filp->private_data;

    switch(cmd) {
        case LUNIX_IOC_SWITCH:
            if (!capable(CAP_SYS_ADMIN))
                return -EPERM;
            if (down_interruptible(&state->lock))
                return -ERESTARTSYS;
            state->switch_raw = !state->switch_raw;
            up(&state->lock);
            break;
        default:
            return -ENOTTY;
    }

    // return -EINVAL;
    return retval;
}
```

Η `ioctl()` είναι μια απλή κλήση συστήματος που δίνει την δυνατότητα στον χρήστη να επιλέξει ανάμεσα σε 2 μορφές εξόδου των μετρήσεων που δίνει ο αισθητήρας. Αυτό γίνεται μέσω του ορίσματος που μπορεί να πάρει την τιμή που αντιστοιχεί στο format της εξόδου `Raw` ή `Cooked`. Στην πρώτη περίπτωση, ο χρήστης λαμβάνει τα δεδομένα ακατέργαστα, όπως αυτά παράγονται από τους αισθητήρες. Στην δεύτερη περίπτωση, έχει ήδη γραφεί ένα πρόγραμμα, το `mk_lookup_tables.c`, το οποίο δημιουργεί 3 πίνακες, έναν για κάθε είδος μέτρησης, οι οποίοι κάνουν 1-1 αντιστοίχιση των `raw data` σε `cooked` τιμές. Ύστερα, αρκούν πολύ απλές πράξεις για να βρεθούν το ακέραιο και το κλασματικό μέρος του αριθμού.

Τέλος, κατά την κλήση του `rmmod` εκτελείται μία σειρά από συναρτήσεις. Αναλυτικότερα, καλείται η `lunix_module_cleanup()` του αρχείου `lunix-module.c` η οποία καλεί την υλοποιημένη στο `lunix-chrdev.c` `lunix_chrdev_destroy()` η οποία καθαρίζει όποιο υπόλειμμα υπάρχει στη μνήμη από το `kernel module`.

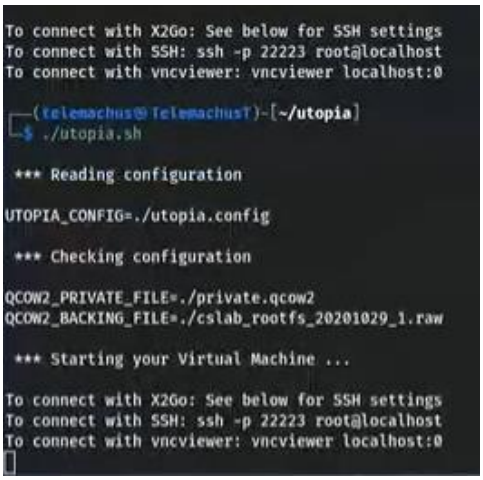
Additions in header file

Επιπλέον, σημειώνεται ότι προστέθηκαν κάποιες γραμμές κώδικα στο αρχείο `lunix_chrdev.h` ώστε να υλοποιηθεί κλήση συστήματος.

```
struct lunix_chrdev_state_struct {
    enum lunix_msr_enum type;
    struct lunix_sensor_struct *sensor;

    /* A buffer used to hold cached textual info */
    int buf_lim;
    unsigned char buf_data[LUNIX_CHRDEV_BUFSZ];
    uint32_t buf_timestamp;

    struct semaphore lock;

    int switch_raw;

    /*
     * Fixme: Any mode settings? e.g. blocking vs. non-blocking
     */
};

/*
 * Function prototypes
 */
int lunix_chrdev_init(void);
void lunix_chrdev_destroy(void);

#endif  /* __KERNEL__ */

#include <linux/ioctl.h>

/*
 * Definition of ioctl commands
 */
#define LUNIX_IOC_MAGIC         LUNIX_CHRDEV_MAJOR
//#define LUNIX_IOC_EXAMPLE     _IOR(LUNIX_IOC_MAGIC, 0, void *)
#define LUNIX_IOC_SWITCH        _IO(LUNIX_IOC_MAGIC, 0)

#define LUNIX_IOC_MAXNR         0

#endif  /* _LUNIX_H */
```

Δοκιμή της υλοποίησης

Σύνδεση με το Virtual Machine:

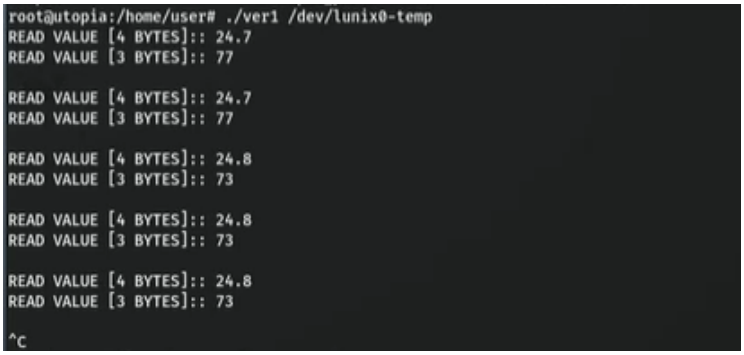


Εκτέλεση κώδικα:

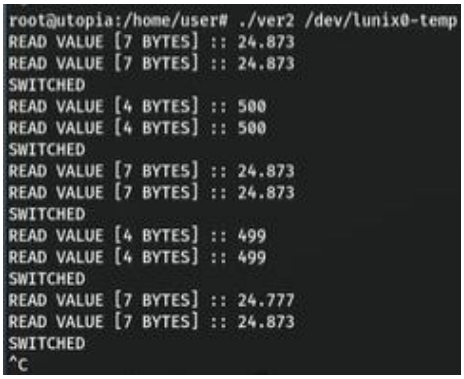
Σε ένα terminal εκτελώντας για παράδειγμα την εντολή `cat /dev/lunix0-temp`, εμφανίζονται ενδεικτικές τιμές από τον αισθητήρα θερμοκρασίας. Ομοίως συμβαίνει και από τους υπόλοιπους όπως φαίνεται παρακάτω.

Επίσης, ενδεικτικά έχουν δημιουργηθεί κάποια δοκιμαστικά αρχεία για να ελεγχθεί ότι λειτουργεί σωστά ο οδηγός. Με λίγα λόγια, γίνεται `read` στο αρχείο που έχει δημιουργηθεί και εκτελώντας στο `terminal` την εντολή όπως φαίνεται παρακάτω, αρχικά διαβάζονται 4 byte και μετά 3 byte για να ελεγχθεί ότι το `f_pos` δουλεύει κανονικά.

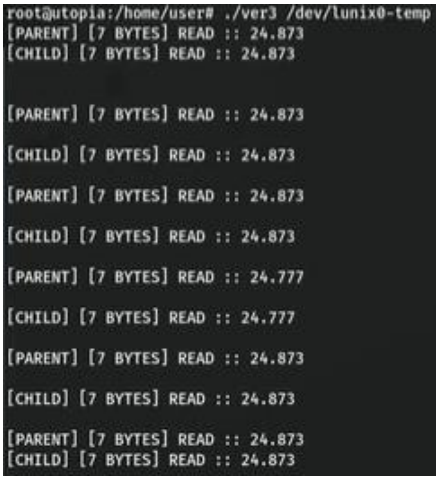
Και με το δοκιμαστικό κώδικα, εμφανίζεται το παρακάτω:



Στη συνέχεια, όπως φαίνεται και παρακάτω, εκτελείται από τις επεκτάσεις το `ioctl` ώστε να δείχνει τα `raw data` αντί για τις μορφοποιημένες τιμές. Σημειώνεται ότι αυτό δουλεύει μόνο από `root`.



Τέλος φαίνεται μια εκτέλεση της `fork` σε πατέρα – παιδί ώστε να ελεγχθεί ότι λειτουργεί κανονικά.

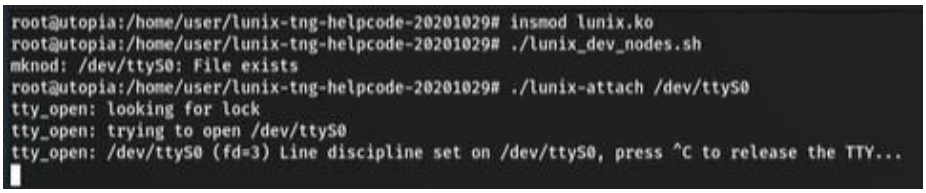


<u>Ανάλυση & Επεξήγηση κώδικα:</u>

Γενικά ως προς την <u>εγκατάσταση</u>:

Αρχικά εκτελείται η εντολή `make`. Έπειτα εισάγεται το `module`. Στη συνέχεια, φτιάχνονται τα `nodes`, με την αντίστοιχη περιοχή των `major` & `minor numbers`. Τέλος, γίνεται `attach` στο `ttyS0` ώστε να εγκατασταθεί ο `line discipline` και να στέλνει στο `module` ό,τι χρειάζεται.

Ο `driver` τρέχει όταν εκτελείται `insmod`. Αυτό που ισχύει πιο συγκεκριμένα είναι ότι έχει εγκατασταθεί, είναι λειτουργικός αλλά αν δεν εγκατασταθεί το `line discipline` δεν μπορεί να πάρει τις μετρήσεις γιατί αυτό γίνεται από το `lunix-attach` προκειμένου να πάρει τις πληροφορίες από το `S0` αυτά που στέλνονται μέσω `TCP`.



Ορθότητα Υλοποίησης:

Προκειμένου να ελεγχθεί η σωστή λειτουργία του `driver` μας, υλοποιήσαμε ένα πρόγραμμα του οποίου ο κώδικας φαίνεται παρακάτω:

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

#include <sys/ioctl.h>
#include "lunix-tng-helpcode-20201029/lunix-chrdev.h"

int main(){
    int fd, sz, i, p;
    char *c = (char *) calloc(100, sizeof(char));

    fd = open("/dev/lunix0-temp", O_RDONLY);

    if(fd < 0){
        perror("OPEN");
        exit(1);
    }

    i = 0;
    while (1){
        i++;
        sz = read(fd, c, 1000);
        c[sz] = '\0';
        printf("READ VALUE [%d BYTES] :: %s", sz, c);
        if(i%2 == 0){
            ioctl(fd, LUNIX_IOC_SWIRCH);
        }
    }
}
```