ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΕΡΓΑΣΤΗΡΙΟ ΛΕΙΤΟΥΡΓΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
7ο Εξάμηνο
2η Εργαστηριακή Άσκηση

Ομάδα oslab45
Προεστάκη Χριστίνα 03118877
Παπαδούλης Γεώργιος 03118003

Οδηγός Ασύρματου Δικτύου Αισθητήρων στο Λειτουργικό Σύστημα Linux

Αρχικοποίηση του οδηγού

Για την αρχικοποίηση του οδηγού ενσωματώνει ο διαχειριστής (root) στον πυρήνα το τμήμα κώδικα πυρήνα του οδηγού μέσω της εντολής insmod ./lunix.ko.
Με την εντολή αυτή καλείται από το αρχείο lunix-module.c η συνάρτηση:

__init lunix_module_init()

η οποία με τη σειρά της καλεί τις συναρτήσεις:

lunix_protocol_init (από το αρχείο lunix-protocol.c)
lunix_sensor_init (από το αρχείο lunix-sensor.c)
lunix_ldisc_init() (από το αρχείο lunix-ldisc.c)
lunix_chrdev_init() (από το αρχείο lunix-chrdev.c)

που αποσκοπούν στην αρχικοποίηση του οδηγού Lunix:TNG.

Συγκεκριμένα:

- Η lunix_protocol_init(struct lunix_protocol_state_struct *) αρχικοποιεί τη δομή lunix_protocol_state_struct η οποία περιγράφει τη μηχανή καταστάσεων που είναι υπεύθυνη για την επεξεργασία των πακέτων δεδομένων που λαμβάνονται από το τμήμα του κατώτερου οδηγού συσκευής και προωθούνται από τη διάταξη γραμμής.

- Η lunix_sensor_init(struct lunix_sensor_struct *) αρχικοποιεί τη δομή lunix_sensor_struct η οποία αποτελείται από ένα spinlock, μια ουρά αναμονής και τη δομή lunix_msr_data_struct η οποία είναι ο προσωρινός χώρος αποθήκευσης των 16bit (χωρίς πρόσημο) αριθμητικών ποσοτήτων που εξάγονται αφού τα δεδομένα περάσουν από το protocol.
- Η lunix_ldisc_init() εγκαθιστά τη διάταξη γραμμής του Lunix:TNG μέσω της κλήσης tty_register_ldisc() η οποία συνδέει τους δείκτες συναρτήσεων της δομής tty_ldisc_ops με τις συναρτήσεις που υλοποιήσαμε για την διάταξη γραμμής του Lunix:TNG.

- Η lunix_chrdev_init() ζητήθηκε να υλοποιηθεί από εμάς και είναι υπεύθυνη να αρχικοποιήσει τη συσκευή χαρακτήρων (char device) του οδηγού μας.

```
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); //... bits a->ma, b->mi
    /*
        Αποθηκεύει τον αριθμό τον device numbers που μας απασχολούν
     */
    /* register_chrdev_region? */
    ret = register_chrdev_region(dev_no, lunix_minor_cnt, "lunixTNG");
    if (ret < 0) {
```

```
        debug("failed to register region, ret = %d\n", ret);
        goto out;
    }
    /*
        add a char device to the system
        δίνει την δυνατότητα στον πυρήνα να καλέσει τα operations
    */
    /* 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;
}
```

Αρχικά η συνάρτηση μας καλεί την cdev_init (που βρίσκεται στο αρχείο linux/fs/char_dev.c) η οποία αρχικοποιεί το struct cdev της δομής inode που κατασκευάζεται από τον πυρήνα για κάθε αρχείο (και είναι μοναδικό για κάθε αρχείο). Μέσα στη δομή αυτή υπάρχει ένας δείκτης προς το struct file_operations (που περιέχει δείκτες προς τις ρουτίνες του οδηγού) που παίρνει την τιμή του δεύτερου ορίσματος (cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops)). Με αυτόν τον τρόπο, συνδέουμε τα ειδικά αρχεία που κατασκευάζουμε (διακρίνονται μέσω του major number) με τις υλοποιήσεις των συναρτήσεων που μπορούν να εφαρμοστούν σε αυτά.

Έπειτα, μέσω της register_chrdev_region(dev_no, lunix_minor_cnt, "lunixTNG") δεσμεύει τα device numbers για τα ειδικά αρχεία. Όλα τα ειδικά αρχεία μας έχουν τον ίδιο major number καθώς αυτός αναφέρεται στον οδηγό συσκευής και διαφορετικό minor number που σε συνδυασμό με τον major number καθορίζουν ακριβώς την συσκευή. Τα minor number που αναθέτουμε ξεκινούν από το 0 (dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0))και φτάνουν μέχρι την τιμή lunix_minor_cnt η οποία προκύπτει από το γινόμενο των αισθητήρων και των μετρήσεων που μπορεί να έχει ο καθένας (εδώ 16 αισθητήρες που ο καθένας μπορεί να πάρει μέχρι 8 μετρήσεις).

Τέλος με την συνάρτηση cdev_add ενημερώνουμε τον πυρήνα για τη νέα συσκευή χαρακτήρων.

Αξίζει να τονίσουμε ότι όλη η διαδικασία της αρχικοποίησης γίνεται σε process context καθώς καλείται από την διεργασία insmod που εκτελεί ο διαχειριστής (root).

Περιγραφή ροής δεδομένων από το hardware μέχρι τον χρήστη

Σε interrupt context, το hardware μας ενημερώνει ότι έχουν έρθει νέα δεδομένα, τα οποία η lunix_ldisc_receive «διαβάζει» από τον tty_buffer και καλεί την lunix_protocol_received buf, η οποία εφαρμόζει το πρωτόκολλο, αλλάζοντας το state του lunix_protocol_state_struct. Εφόσον εφαρμοστεί το πρωτόκολλο στα δεδομένα, αυτά αποθηκεύονται στον πίνακα packet και καλείται η lunix_protocol_update_sensors, η οποία ανάλογα με τον σένσορα τον οποίο αφορούν τα δεδομένα, καλεί την lunix_sensor_update. Επιπλέον, αποθηκεύει στις μεταβλητές batt, temp, light τις κατάλληλες τιμές από το packet. Στη συνέχεια, οι τιμές των batt, temp, light αποθηκεύονται στην πρώτη θέση του πίνακα values για κάθε ένα από τα lunix_msr_data_struct των 3 μεταβλητών (msr_data[BATT/TEMP/LIGHT]). Για να αποφευχθεί πιθανό race condition κατά την αποθήκευση των μεταβλητών αυτών στον πίνακα msr_data χρησιμοποιούμε spinlock. Η επιλογή του spinlock αντί semaphore γίνεται γιατί η λήψη νέων δεδομένων γίνεται σε interrupt context. Αυτό σημαίνει ότι η διαδικασία αυτή δεν εκτελείται από κάποια διεργασία ώστε να μπορεί να “κοιμηθεί” (όπως συμβαίνει με τη χρήση των semaphores), αλλά από κάποιο interrupt handler. Όταν ολοκληρωθεί η εγγραφή των δεδομένων, καλείται η wake_up_interruptible, η οποία ξυπνάει την ουρά διεργασιών που κοιμάται, για το συγκεκριμένο sensor. Στην ουρά των διεργασιών του κάθε σένσορα wq, περιμένουν κάποιες διεργασίες σε sleep οι οποίες περιμένουν να διαβάσουν καινούρια δεδομένα για να τυπώσουν στο χρήστη.

Ο χρήστης αλληλεπιδρά με τα δεδομένα μέσω των κλήσεων συστήματος open πάνω σε ένα ειδικό αρχείο και read πάνω σε ένα ανοικτό ειδικό αρχείο. Όταν εκτελείται η open ο πυρήνας αναγνωρίζει από το major number του ειδικού αρχείο σε ποιον οδηγό αντιστοιχεί (που βρίσκονται μέσα στο struct inode που έχει δημιουργηθεί όταν κατασκευάστηκε το αρχείο) και συνδέει τη δομή file που δημιουργείται με τη δομή file_operations που περιέχει τους δεικτες προς τις υλοποιήσεις των συναρτήσεων του οδηγού μας. Επομένως, με την κλήση κάποιας συνάρτησης πάνω στο ειδικό αρχείο εκτελείται η συγκεκριμένη υλοποίησή της του οδηγού μας μέσω του δείκτη του file_operations (π.χ. Όταν ο χρήστης καλεί τη κλήση συστήματος read μέσω αυτής της διαδικασίας καλείται η συνάρτηση lunix_chrdev_read).

- **lunix_chrdev_open**

Κάθε φορά που ο χρήστης ανοίγει ένα αρχείο (κλήση συστήματος open) δημιουργείται μία δομή struct file, η οποία αντιπροσωπεύει το ανοιχτό αρχείο και περνιέται σαν όρισμα στις υπόλοιπες συναρτήσεις.
Στην lunix_chrdev_open(struct inode *inode, struct file *filp) αρχικά συνδέεται το πεδίο f_op της δομής file με την κατάλληλη δομή file_operations μέσω της δομής inode (filp->f_op = inode->i_cdev->ops). Επίσης από το struct inode παίρνουμε το minor_number της συσκευής με την εντολή minor_number = iminor(inode). Από το minor_number βρίσκουμε τον αριθμό του sensor και τον τύπο μέτρησης με βάση την σύμβαση που έχει ακολουθηθεί. Στη συνέχεια, δεσμεύουμε χώρο για το struct private state ώστε να αποθηκεύσουμε αυτές τις πληροφορίες. Τα πεδία του struct είναι τα εξής:

- Sensor: αριθμός sensor
- Type: τύπος μέτρησης
- Timestamp: ώρα τελευταίας ανανέωσης δεδομένων του πίνακα buf_data
- Buf_lim: δείκτης στον πίνακα buf_data
- Lock: semaphore

Τα timestamp και buf_lim αρχικοποιούνται σε 0 με το άνοιγμα του αρχείου επειδή με το άνοιγμα δεν έχει γίνει ανανέωση των δεδομένων ακόμα και ούτε έχουμε αρχίσει να διαβάζουμε από τον πίνακα (buf_data). Ακόμα, αρχικοποιούμε τον σεμαφόρο σε 1 - unlocked. Αυτή η δομή αποθηκεύεται στο πεδίο private_data του αρχείου ώστε οι υπόλοιπες συναρτήσεις να έχουν πρόσβαση σε αυτά τα δεδομένα.

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)

{

    /* Declarations */

    /* ? */

    int ret, no, type, minor_number;

    struct lunix_chrdev_state_struct *private_state;

    debug("entering\n");

    ret = -ENODEV;

    if ((ret = nonseekable_open(inode, filp)) < 0)
```

```
        goto out;


    //connect the f_op of file with the ops of inode struct
    filp->f_op = (inode->i_cdev)->ops;
    /*
     * 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);
    no = minor_number / 8;
    type = minor_number % 8;
/*
    if(type >= N_LUNIX_MSR ){ //types : {0: batt , 1: temp, 2: light}
        ret -ENODEV;  //ERROR: wrong minor number
        goto out;
    }
*/
    /* Allocate a new Lunix character device private state structure */
    /* ? */
    private_state = kmalloc(sizeof(*private_state),GFP_KERNEL);
    private_state->type = type;
    private_state->sensor= lunix_sensors+no;
    private_state->buf_timestamp=0;
```

```
    private_state->buf_lim=0;

    sema_init(&(private_state->lock), 1);



    filp->private_data = private_state; //for access from other fun

out:

    debug("leaving, with ret = %d\n", ret);

    return ret;

}
```

- **lunix_chrdev_state_needs_refresh**

Σκοπός της lunix_chrdev_state_needs_refresh είναι να ελέγξει αν το τελευταίο update των sensors (στο πίνακα msr_data[]) έχει γίνει μετά το τελευταίο update του πίνακα buf_data. Ουσιαστικά, ελέγχει αν το δεδομένα που έχουμε στο buf_data και θα τυπώσουμε στον χρήστη είναι τα ανανεωμένα δεδομένα που μας έχει στείλει το hardware. Αν δεν είναι, τότε χρειάζεται να γίνει το refresh και επιστρέφεται 0.

```
static   int   lunix_chrdev_state_needs_refresh(struct   lunix_chrdev_state_struct
*state)

{

    struct lunix_sensor_struct *sensor;



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

    /* ? */

    //refreshed when
```

```
        if(sensor->msr_data[state->type]->last_update  /*sensor  last  update*/>
state->buf_timestamp /* chrdev_data_buf last update*/)

        return 1;

    return 0;

}
```

- **lunix_chrdev_state_update**

Σκόπος της lunix_chrdev_state_update είναι να ανανεώσει τα δεδομένα, αν αυτό είναι αναγκαίο. Αρχικά, αφού κλειδώσουμε το spinlock η τιμή του πίνακα msr_data[type] αποθηκεύεται στο tempor_data και στη συνέχεια ξεκλειδώνουμε το spinlock. Η χρήση κλειδώματος είναι απαραίτητη για να μη γίνει το εξής race condition, να μη γραφτούν νέα δεδομένα τη στιγμή που εμείς γράφουμε τα προηγούμενα στο tempor_data. Χρησιμοποιούμε συγκεκριμένα spinlock διότι η παραλαβή δεδομένων από το hardware γίνεται σε interrupt context και συνεπώς δεν υπάρχει κάποια διεργασία για να κοιμηθεί, όπως στους semaphores. Επιπλέον, για να ενεργοποιήσουμε τις διακοπές χρησιμοποιούμε spin_lock_irq_save. Στη συνέχεια ελέγχουμε με την lunix_chrdev_state_refresh αν τα δεδομένα αυτά είναι καινούρια. Αν είναι συνεχίζουμε στην επεξεργασία τους. Αναλόγως με τον τύπο της μέτρησης, αντιστοιχίζω την 16bit που έλαβα με την κατάλληλη τιμή της μορφής xxyyy και την αποθηκεύω στο buf_data. Τέλος, ενημερώνω το buf_timestamp ώστε να έχει την ανανεωμένη τιμή του last_update.

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
    struct lunix_sensor_struct *sensor;
    uint32_t tempor_data, div, mod, last_update;
    unsigned long flags;

    debug("leaving\n");

    /*
     * Grab the raw data quickly hold the
     * spinlock for as little as possible.
     */
    /* ? */
    sensor = state->sensor;
    spin_lock_irqsave(&sensor->lock, flags);
```

```
    tempor_data = sensor->msr_data[state->type]->values[0];
    last_update = sensor->msr_data[state->type]->last_update;
    spin_unlock_irqrestore(&sensor->lock, flags);
    /* Why use spinlocks? See LDD3, p. 119 */

    /*
     * Any new data available?
     */
    /* ? */
    if(!lunix_chrdev_state_needs_refresh(state)){
        return -EAGAIN;
    }

    /*
     * Now we can take our time to format them,
     * holding only the private state semaphore
     */
    state->buf_timestamp = last_update;
    switch(state->type){
            case BATT:
                tempor_data = lookup_voltage[tempor_data];
                break;
            case TEMP:
                tempor_data = lookup_temperature[tempor_data];
                break;
            case LIGHT:
                tempor_data = lookup_light[tempor_data];
                break;
            default:
                debug("ERROR_update: Wrong type");
                return -1;
        }

    div = tempor_data / 1000; //morfi einai xx.yyy
    mod = tempor_data % 1000;

    //writing the data to state->buf_data
    if(((state->buf_lim) = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ,
"%lu.%lu\n", (unsigned long)div, (unsigned long)mod)) > LUNIX_CHRDEV_BUFSZ){
```

```
        debug("ERROR_update: Buffer overflow");
        return -1;
    }

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

- **lunix_chrdev_read**

Αρχικά η read βρίσκει τη δομή lunix_chrdev_state_struct μέσω του πεδίο private_data του file. Πριν ζητήσει να ανανεωθούν τα δεδομένα που έχουν σταλεί από τους αισθητήρες (τα οποία αποθηκεύονται προσωρινά στο msr_data του sensor) μέσω της εντολής lunix_chrdev_state_update, κλειδώνει το semaphore(mutex) με την εντολή down_interruptible. (Χρησιμοποιείται η down_interruptible ώστε να μπορεί να ξεκλειδώσει ο semaphore και μέσω σημάτων.) Αυτό συμβαίνει γιατί η συνάρτηση update αν δεν επιστρέψει -EAGAIN γράφει πάνω στο πίνακα state->buf_data, οπότε αν παραπάνω από μία διεργασία προσπαθήσουν να την καλέσουν ταυτόχρονα θα δημιουργηθεί ένα race condition. Αυτό το race condition επηρεάζει μόνο διεργασίες που έχουν γονεϊκή σχέση μεταξύ τους καθώς το struct lunix_chrdev_state_struct είναι μοναδικό για κάθε ένα ανοιχτό αρχείο (αφού διαφορετικά ανοιχτά αρχεία αντιπροσωπεύονται από διαφορετικά struct file). Όμως διεργασίες που έχουν τη σχέση parent-child οι οποίες δημιουργούνται μέσω της fork() μοιράζονται τα ίδια ανοιχτά αρχεία. Έτσι ο ρόλος του semaphore στη read και στην update είναι για να αποτρέψει τα race condition που δημιουργούνται σε διεργασίες που συνδέονται με γονεϊκή σχέση μεταξύ τους.
Ελέγχουμε αν βρισκόμαστε στην αρχή των δεδομένων (*f_pos == 0) και αν ναι εξετάζουμε με την lunix_chrdev_state_update αν τα δεδομένα που βρίσκονται στο χώρο προσωρινής αποθήκευσης (msr_data) είναι καινούργια (ανανεωμένα). Αν δεν είναι, επιστρέφει -EAGAIN αλλιώς ανανεώνει τον buf_data με τα καινούργια δεδομένα.
Αν δεν υπάρχουν νέα δεδομένα τότε η διεργασία πρέπει να “κοιμηθεί” (αν δεν είναι ενεργοποιημένη η σημαία O_NONBLOCK) και να συνεχίσει όταν αυτά είναι έτοιμα. Πριν κοιμήσουμε την διεργασία χρειάζεται να ξεκλειδώσουμε το semaphore ώστε να επιτρέπουμε στις άλλες (με γονεϊκή σχέση) διεργασίες να εκτελέσουν την read. Έπειτα, προσθέτουμε την διεργασία στην ουρά αναμονής του αισθητήρα με την εντολή wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state)).

Η διεργασία “ξυπνάει” από τη συνάρτηση lunix_sensor_update και καλεί την lunix_chrdev_state_needs_refresh για να ελέγξει είναι καινούργια.

Όταν τα δεδομένα είναι καινούργια (η update δεν επιστρέφει -EAGAIN) τότε η read μέσω της συνάρτησης copy_to_user μεταφέρει τα δεδομένα από το buf_data σε έναν buffer που μπορεί να έχει πρόσβαση ο χρήστης και να διαβάσει από αυτον.
Τέλος, ανανεώνουμε το δείκτη που μας δείχνει σε ποιο σημείο του πίνακα buf_data βρισκόμαστε και αν έφτασε στο τέλος των δεδομένων (buf_lim) τον βάζουμε να δείχνει στην αρχή.

```
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; //state->trexousa katastasi tis siskevis
    WARN_ON(!state);

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

    /* 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) {
            /* ? */
            up(&(state->lock));
            if(filp->f_flags & O_NONBLOCK)
```

```
                return -EAGAIN;
            //debug("reading: going to sleep");
            /* The process needs to sleep */
            /* See LDD3, page 153 for a hint */
            if(wait_event_interruptible(sensor->wq,
lunix_chrdev_state_needs_refresh(state)))
                return -ERESTARTSYS;
            if(down_interruptible(&state->lock))
                return -ERESTARTSYS;
        }
    }
    /* End of file */
    /* ? */
    //buf_lim -> shows EOF
    if(state->buf_lim == 0){ //nothing to read - EOF
        ret = 0; //return the number of bytes that we read
        goto out;
    }
    /* Determine the number of cached bytes to copy to userspace */
    /* ? */
    if(cnt > state->buf_lim - *f_pos)
        cnt = state->buf_lim - *f_pos;
//  cnt = min(cnt, (state->buf_lim - *f_pos)); //cnt
    if(copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)){
        up(&state->lock);
        return -EFAULT;
    }
    /* Auto-rewind on EOF mode? */
    /* ? */
    *f_pos += cnt;
    if(*f_pos >= state->buf_lim){
        *f_pos=0;
    }
    ret = cnt; //cnt is the bytes that we read

out:
    /* Unlock? */
    up(&state->lock);
    return ret;
```

```
}
```

- **lunix_chrdev_release**

Η lunix_chrdev_release απελευθερώνει τη μνήμη που δεσμεύθηκε για την αποθήκευση του πεδίου private_data της δομής file.

```
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
    /* ? */
    //deallocate from private data the private_state
    kfree(filp->private_data);
    return 0;
}
```

Δοκιμή της λειτουργίας του οδηγού

Παρακάτω, παρατίθεται κώδικας που χρησιμοποιούμε για να ελέγξουμε την ορθή λειτουργία του οδηγού για τις γονεϊκές διεργασίες.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>

int main(){
    int fd;
    pid_t pid;
    char buffer[99999];
    fd = open("/dev/lunix0-batt", O_RDONLY);
    pid = fork();
    if(pid==0){
        while(1){
        read(fd, &buffer,6);
        printf("I'm the child\n");
```

```
        write(1, &buffer,6);
        }
    }
    else{
        while(1){
        read(fd, &buffer, 6);
        printf("I'm the parent\n");
        write(1, &buffer, 6);

        }
    }

    return 0;
}
```