Αναφορά της δεύτερης εργαστηριακής άσκησης

Ανθοπούλου Φαίδρα-Αναστασία 03118818

Πευκιανάκης Κωνσταντίνος 03114897

Όλες οι προσθήκες κώδικα που χρειάστηκε να κάνουμε, βρίσκονται στο αρχείο chrdev.c. Οι συναρτήσεις που υλοποιήσαμε ήταν οι lunix_chrdev_state_needs_refresh, lunix_chrdev_state_update, lunix_chrdev_open, lunix_chrdev_release, lunix_chrdev_read, lunix_setup_cdev, και lunix_chrdev_init, ενώ οι υπόλοιπες ήταν ήδη έτοιμες. Ακολουθεί η κάθε συνάρτηση, και από κάτω μια επεξήγηση για τον τρόπο σκέψης μας για την υλοποίησή της, καθώς και τυχόντα προβλήματα που αντιμετωπίσαμε.

lunix_chrdev_state_needs_refresh:

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
{
struct lunix_sensor_struct *sensor;
        WARN_ON ( !(sensor = state->sensor));
if (state->buf_timestamp == sensor->msr_data[state->type]->last_update)
return 0;
return 1;
}
```

Η συνάρτηση αυτή έχει ως σκοπό να μας ενημερώνει αν τα δεδομένα που βρίσκονται στο struct state είναι παλιά και χρειάζονται ανανέωση. Για να αποφασίσουμε αν χρειάζονται ανανέωση, ελέγχουμε αν το πεδίο buf_timestamp του state, που μας δίνει την χρονική στιγμή την οποία λήφθηκαν τα δεδομένα που βρίσκονται στο state, είναι ίδιο με το πεδίο last update των δεδομένων (msr_data) των sensor, που μας δίνει την τελευταία χρονική στιγμή που δόθηκαν δεδομένα από τους αισθητήρες. Εάν πράγματι είναι ίδια τότε τα δεδομένα στο state είναι επίκαιρα (return 0, δεν χρειαζόμαστε ανανέωση), ενώ αν δεν είναι ίδια τότε έχουν έρθει καινούργια δεδομένα που σημαίνει ότι χρειαζόμαστε ανανέωση (return 1).

lunix_chrdev_state_update:

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
struct lunix_sensor_struct *sensor;
debug("leaving\n");

/*
 * Grab the raw data quickly, hold the
 * spinlock for as little as possible.
 */

sensor=state->sensor;

        /*
 * Any new data available?
 */

if (!lunix_chrdev_state_needs_refresh(state))
                return -EAGAIN;
uint32_t rawdata;

spin_lock(&sensor->lock);
state->buf_timestamp=sensor->msr_data[state->type]->last_update;
rawdata=sensor->msr_data[state->type]->values[0];
spin_unlock(&sensor->lock);

/* Why use spinlocks? See LDD3, p. 119 */


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

long mydata;
```

```
switch (state->type){
case BATT: mydata=lookup_voltage[rawdata]; break;
case TEMP: mydata=lookup_temperature[rawdata]; break;
case LIGHT: mydata=lookup_light[rawdata]; break;
}
int integer_part = mydata / 1000;
int decimal_part = mydata % 1000;

state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%d.%d\n", integer_part, decimal_part);

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

Ο ρόλος της update είναι η λήψη και η επεξεργασία των νέων δεδομένων. Για να την υλοποιήσουμε, αρχικά αν υπάρχουν νέα δεδομένα (ελέγχουμε με την refresh) παίρνουμε το καινούργιο buf_timestamp και τα δεδομένα από τους sensors που τα αποθηκεύουμε στην μεταβλητή rawdata, ενώ κρατάμε ένα spinlock για όσο διαρκεί η απόκτηση των δεδομένων, έτσι ώστε να αποφύγουμε κάποιο race condition σε περίπτωση που χρησιμοποιούνται threads. Αφότου ελευθερώσουμε το spinlock (δεν πρέπει να το κρατάμε για πολύ γιατί όσες άλλες διαδικασίες το χρειάζονται θα προσπαθούν συνεχώς να το πάρουν και θα καταναλώνουν πόρους), θα πρέπει να κάνουμε την επεξεργασία των δεδομένων που λάβαμε. Έτσι ανάλογα με την τιμή του πεδίου type του struct state (BATT, TEMP, ή LIGHT) επεξεργαζόμαστε την μεταβλητή rawdata κατάλληλα με την βοήθεια των συναρτήσεων που βρίσκονται στα mk_lookup_tables.c και lunix-lookup.h και αποθηκεύουμε την νέα τιμή στην μεταβλητή mydata. Εκτελώντας τις κατάλληλες πράξεις υπολογίζουμε το ακέραιο και το δεκαδικό μέρος της μεταβλητής, και τα μεταφέρουμε στο πεδίο buf_data του state μέσω της συνάρτησης snprintf, ενώ ταυτόχρονα ενημερώνεται και το buf_lim που είναι το μέγεθος του buf_data.

Υλοποιώντας αυτήν την συνάρτηση συναντήσαμε ένα πρόβλημα. Αρχικά όταν κατά τον έλεγχο της refresh βλέπαμε ότι δεν είχαμε νέα δεδομένα, χρησιμοποιούσαμε μια εντολή goto out, όμως έτσι δεν δούλευε το πρόγραμμα. Αντί αυτού χρησιμοποιήσαμε το return -EAGAIN, το οποίο δούλεψε.

lunix_chrdev_open:

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

```
{

   /* Declarations */
   int minor = iminor(inode);
   int type = minor & 7; /* check if type = 0 ,1 ,2*/
   int sensor_index = (minor - type) >> 3; /*sensor index = (minor - type) / 8 */

   /* Allocate a new Lunix character device private state structure */
   struct lunix_chrdev_state_struct *dev = (struct lunix_chrdev_state_struct*)
kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);

    /*
    * Associate this open file with the relevant sensor based on
    * the minor number of the device node [/dev/sensor<NO>-<TYPE>]
    */

   dev->type = type;
   dev->sensor = lunix_sensors + sensor_index;
   dev->buf_lim = 1;
   dev->buf_data[0] = '\0';
   dev->buf_timestamp = 0;
   sema_init(&dev->lock, 1);

   int ret;

   debug("entering\n");
   ret = -ENODEV;

   if ((ret = nonseekable_open(inode, filp)) < 0)
      goto out;

   filp->private_data = dev;

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

Ο σκοπός της συνάρτησης αυτής είναι να κάνουμε τις απαραίτητες αρχικοποιήσεις όσον αναφορά τα minor numbers . Αρχικά παίρνουμε το minor number του κάθε sensor, και από αυτό με bitmasking που αφήνει μόνο τα 3 τελευταία bits εξάγουμε την τιμή του πεδίου type, ενώ ο sensor index θα είναι ο minor χωρίς τα 3 τελευταία bits. Μετά με την kmalloc δίνουμε χώρο για ένα καινούργιο struct που το ονομάζουμε dev και είναι τύπου state struct, ενώ αμέσως μετά αρχικοποιούμε τα πεδία του καταλλήλως. Δηλαδή μέσω του dev αρχικοποιούμε το state struct του κάθε sensor. Τέλος, αποθηκεύουμε το dev στα private_data του filp, για να βρει μετά η read.

lunix_chrdev_release:

```
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
debug("Freeing the resources allocated in filp->private_data");
kfree(filp->private_data);
return 0;
```

Εδώ απλά απελευθερώνω τον χώρο μνήμης που έπιασα για το state της κάθε συσκευής, μέσω της εντολής kfree, όταν αυτό πλέον δεν μου χρειάζεται να υπάρχει.

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;
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? */
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);
if (filp->f_flags & O_NONBLOCK)
                        return -EAGAIN;

                //PDEBUG("\"%s\" reading: going to sleep\n", current->comm);

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

if (down_interruptible(&state->lock))
                        return -ERESTARTSYS;
}
}

/* End of file */
if(*f_pos==state->buf_lim){
ret = 0;
*f_pos = 0; /* Auto-rewind on EOF mode? */
goto out;
}

/* Determine the number of cached bytes to copy to userspace */

if(cnt > state->buf_lim)
cnt = state->buf_lim;

if (copy_to_user(usrbuf, state->buf_data, cnt)) {
```

```
                ret = -EFAULT;
                goto out;
    }

        *f_pos += cnt;
        ret = cnt;
*f_pos = 0;

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

Ο ρόλος της συνάρτησης read είναι να εμφανίζει στον χρήστη τα δεδομένα που θέλουμε, και να επιστρέφει το μέγεθος των δεδομένων αυτών. Αρχικά, παίρνουμε τον σηματοφορέα. Μετά, εφόσον βρισκόμαστε στην αρχή των δεδομένων που θέλουμε να μεταφέρουμε (το buff_data του state), και όσο δεν υπάρχουν νέα δεδομένα για να κάνουμε update αφήνουμε τον σεμαφόρο. Κοιμίζουμε την διαδικασία και περιμένουμε έως ότου να χρειάζεται refresh, και τότε ξαναπαίρνουμε τον σεμαφόρο. Εάν βρισκόμαστε εξαρχής στο τέλος του αρχείου, τότε κάνουμε rewind στο αρχείο και επιστρέφουμε 0 γιατί δεν διαβάσαμε τίποτα. Αλλιώς, αν ο αριθμός χαρακτήρων που ζητάμε να διαβαστεί είναι μεγαλύτερος των δεδομένων που έχουμε τότε ορίζουμε πως θα διαβαστούν μόνο όσα δεδομένα έχουμε. Τότε χρησιμοποιώντας την copy_to_user (μεγαλύτερη ασφάλεια από την memcopy γιατί ελέγχει τα δικαιώματα πρόσβασης της διαδικασίας στην μνήμη) δίνουμε cnt bytes από τα περιεχόμενα της μεταβλητής buf_data. Ύστερα, αυξάνουμε την θέση που βρισκόμαστε στο αρχείο κατά cnt, λέμε ότι πρέπει να επιστραφεί το cnt, και επιστρέφουμε στην αρχή του αρχείου. Τέλος, απελευθερώνουμε τον σημαφόρο.

Εδώ υπήρξε ένα πρόβλημα, αρχικά δεν χρησιμοποιούσαμε το *f_pos = 0; προς το τέλος της read, που είχε ως αποτέλεσμα η cat να βγάζει μόνο ένα αποτέλεσμα και να σταματάει. Έτσι, προσθέσαμε αυτό το rewind κάθε φορά που διαβάζουμε κάτι έτσι ώστε η cat να επιστρέφει δεδομένα διαρκώς.

lunix_setup_cdev:

```
static void lunix_setup_cdev (struct cdev *dev, int index) {
int err, devno = MKDEV(LUNIX_CHRDEV_MAJOR, index);
cdev_init(dev, &lunix_chrdev_fops);
```

```
(*dev).owner = THIS_MODULE;
(*dev).ops = &lunix_chrdev_fops;
err = cdev_add(dev, devno, 1);
if (err)
printk(KERN_NOTICE "Error %d adding lunix%d", err, index);
}
```

Αυτή είναι μια βοηθητική συνάρτηση που υλοποιήσαμε με την βοήθεια του LDD (κεφάλαιο 3, σελίδα 57), στην ουσία αντί να κάνουμε κατευθείαν cdev_add στην init, χρησιμοποιούμε αυτήν την συνάρτηση, η οποία αρχικοποιεί και προσθέτει κάθε συσκευή στο σύστημα. Αυτό γίνεται μέσω των cdev_init και cdev_add, και ορίζοντας τα υπόλοιπα πεδία κατάλληλα (το device number, devno, προκύπτει από την MKDEV με όρισμα το ajor και το minor number της κάθε σσυσκευής, ενώ ορίζουμε ως owner του dev αυτό το MODULE και λέμε ότι τα operations είναι αυτά που ορίζονται στο lunix_chrdev_fops).

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");

/* cdev_add? */
```

```
if (ret < 0) {
debug("failed to register region, ret = %d\n", ret);
goto out;
}

int i, j;
for(i = 0; i < lunix_sensor_cnt; i++)
for(j = 0; j < 3; j++){
int minor = i * 8 + j;
lunix_setup_cdev(&lunix_chrdev_cdev, minor);
}

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 έχει ως σκοπό να κάνει register κάθε συσκευή στον πυρήνα και να πάρει minor number για κάθε μία, όπως και να κάνει intialise και add για κάθε συσκευή (το οποίο το κάνουμε καλώντας την lunix_setup_cdev). Το register το κάνουμε μέσω της register_chrdev_region, ενώ καλούμε την lunix_setup_cdev για κάθε sensor έτσι ώστε να κάνουμε το cdev_init και το cdev_add (αν και κάποιες από τις εντολές που γράψαμε στην lunix_setup_cdev ακολουθώντας το LDD τις είχατε ήδη γράψει εσείς στην init).