Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά 2ης Εργαστηριακής Άσκησης

Φίλιππος Μαλανδράκης 03116200
Βιτάλιος Σαλής 03115751

1 - Εισαγωγή

Σε αυτή την εργαστηριακή άσκηση μας ζητείται να υλοποιήσουμε έναν οδηγό συσκευής για ένα ασύρματο δίκτυο αισθητήρων κάτω απο το λειτουργικό σύστημα Linux. Μας δίνεται ένα μεγάλο κομμάτι της υλοποίησης καθώς και ο σκελετός του οδηγού συσκευής που πρέπει να υλοποιήσουμε. Στην ενότητα 2 θα παραθέσουμε το setup του μηχανήματος στο οποίο τρέχει ο οδηγός συσκευής (ενότητα 2). Στην ενότητα 3, θα κάνουμε μια επισκόπηση της ήδη υλοποιημένης λειτουργικότητας που μας δίνεται. Στην ενότητα 4, θα παρουσιάσουμε την υλοποίηση του οδηγού συσκευής που πραγματοποιήσαμε. Τέλος, στην ενότητα 5 θα ελέγξουμε την ορθότητα του οδηγού μας.

2 - Setup

Για την υλοποίηση της άσκησης χρησιμοποιούμε ένα Virtual Machine στο okeanos-knossos. Συγκεκριμένα, το μηχάνημα μας τρέχει Debian stretch και έχει 30GB δίσκο, 16GB RAM και 8 επεξεργαστές.

Σε αυτό το μηχάνημα εγκαθιστούμε και κάνουμε setup το utopia. Επιλέγουμε να χρησιμοποιήσουμε την έκδοση buster που δόθηκε μιας και είναι η πιο καινούργια. Αφού κατεβάσουμε τον βοηθητικό κώδικα και ρυθμίσουμε το utopia τρέχουμε το *utopia.sh* που μας δόθηκε για να κάνουμε spawn το utopia virtual machine.

Το utopia virtual machine ακουει για ssh συνδέσεις στην θύρα 22223. Συνδεόμαστε και στήνουμε το sshfs συνδέοντας το filesystem του utopia με αυτό του host. Συγκεκριμένα, συνδέουμε το σημείο όπου υπάρχει ο βοηθητικός κώδικας στο host μηχάνημα. Η διαδικασία έχει ως εξής:

```
user@snf-kno:~$ ./utopia.sh
...
...
...
Utopia running, leave this instance be.
```

Σε άλλο shell:

```
user@snf-kno:~$ ssh root@localhost:22223
...
root@utopia:~# mkdir -p /home/user/host
root@utopia:~# sshfs -o allow_other \
user@10.0.2.2:/home/user/utopia/helpcode \
        /home/user/host
```

Για να κάνουμε compile τον κώδικα πηγαίνουμε στο directory του κώδικα που μας δόθηκε και τρέχουμε *make*. Επιλέγουμε να χρησιμοποιήσουμε τον kernel που έχει δοθεί μαζί με το utopia δηλαδή τον *4.19.0-6-amd64*. Αφού κάνουμε compile τον κώδικα, εισάγουμε το module, δημιουργούμε τα device nodes που θα χρειαστούμε μέσω του *lunix_dev_nodes.sh* που μας έχει δοθεί και τέλος κάνουμε attach την συσκευή */dev/ttyS0*, πάλι χρησιμοποιώντας το *lunix-attach.c* που μας έχει δοθεί:

```
root@utopia:~# cd /home/user/host
root@utopia:~# make
...
...
root@utopia:~# insmod lunix.ko
root@utopia:~# ./lunix_dev_nodes.sh
...
root@utopia:~# ./lunix-attach /dev/ttyS0
...
...
^c
root@utopia:~# rmmod lunix.ko
```

3 - Υπάρχουσα Λειτουργικότητα

Για τις ανάγκες της άσκησης μας δίνονται υλοποιημένο ένα μεγάλο μέρος της λειτουργικότητας του device driver. Σε αυτή την ενότητα θα κάνουμε μια επισκόπηση αυτής της λειτουργικότητας αρχείο προς αρχείο, επισημαίνοντας τα πιο σημαντικά χαρακτηριστικά τους.

lunix.h

Αυτό το αρχείο ορίζει τα βασικά data structures και constants που θα χρησιμοποιήθουν για την υλοποίηση του driver. Ιδιαίτερο ενδιαφέρον έχει το struct *lunix_sensor_struct* το οποίο ορίζει την δομή δεδομένων που θα χρησιμοποιηθεί για τους σένσορες, ορίζοντας ένα spinlock για ασφαλείς πράξεις στα δεδομένα του, ένα waitqueue για processes που περιμένουν να ενημερωθούν για updates και η δομή *lunix_msr_data_struct* που κρατάει τα δεδομένα του σενσορα.

mk_lookup_tables.c & lunix-lookup.h

Το αρχείο *mk_lookup_tables.c* δεν χρησιμοποιείται απο τον driver και τρέχει σε userspace. Η λειτουργία του είναι να δημιουργήσει τα lookup tables που ορίζονται στο *lunix-lookup.h*.

Χρησιμοποιούμε lookup tables διότι ο kernel δεν επιτρέπει floating point πράξεις. Αυτό συμβαίνει διότι η κατάσταση της μονάδας κινητής υποδιαστολής (FP Unit) δεν αποθηκεύεται όταν μεταφερόμαστε από user space σε kernel space. Ταυτόχρονα, κάποιες αρχιτεκτονικές δεν υποστηρίζουν floating point πράξεις στο υλικό και αντι αυτού υλοποιούνται σε userspace βιβλιοθήκες.

Εφόσον οι τιμές που επιστρέφουν οι sensors μπορούν να κωδικοποιηθούν σε 16 bits, υλοποιούμε lookup tables που μετατρέπουν τις raw μετρήσεις σε integers τα οποία μπορούμε να διαιρέσουμε με έναν αριθμό και χρησιμοποιώντας το αποτέλεσμα της διαίρεσης και το υπόλοιπο να τους μετατρέψουμε σε floating point αριθμούς.

lunix-attach.c

Ένα βοηθητικό πρόγραμμα το οποίο έχει ως σκοπό να συνδέσει τον οδηγό συσκευής μας με ένα καθορισμένο TTY, προσαρτώντας σε αυτό το line discipline που έχουμε υλοποιήσει στο *lunix-ldisc.c*.

Συγκεκριμένα, παίρνει ως παράμετρο ένα TTY device και αφού αποθηκευσεί την κατάσταση του και το line discipline που χρησιμοποιεί, αλλάζει τις ρυθμίσεις του και προσαρτά το δικό μας line discipline.

Τέλος, όταν κάνουμε release το TTY, το πρόγραμμα επαναφέρει το TTY στην προηγούμενη του κατάσταση και προσαρτά το παλιό line discipline που είχαμε αποθηκεύσει.

lunix-module.c

Ο κώδικας σε αυτό το αρχείο είναι ο πρώτος που θα τρέξει όταν εισάγουμε το kernel module μας. Είναι υπεύθυνο για το initialization των απαραίτητων data structures και γενικά της λειτουργικότητας του module.

Συγκεκριμένα, δεσμεύει μνήμη για τους σενσορες και καλεί τα initialization method τους, αρχικοποιεί το line discipline που θα χρησιμοποιήσουμε και τέλος καλεί το initialization method του driver που καλούμαστε να υλοποιήσουμε.

Τέλος, όταν αφαιρείται το module είναι υπεύθυνο για την απελευθέρωση της μνήμης που δεσμεύτηκε κατα την αρχικοποίηση του.

lunix-ldisc.{c,h}

Εδώ υλοποιείται το line discipline του driver μας. Είναι το πιο low level component της λειτουργικότητας και είναι υπεύθυνο για να λαμβάνει raw data απο το TTY layer και να τα προωθεί στο *lunix-protocol.c*.

lunix-protocol.{c,h}

Εδώ ο κώδικας υλοποιεί το πρωτόκολλο επικοινωνίας του driver μας με τους sensors. Λαμβάνει raw δεδομένα απο το line discipline και τα ομαδοποιεί σε πακέτα.

Συγκεκριμένα, υλοποιεί ένα state machine το οποίο περιέχει πληροφορία για το στάδιο συλλογής του πακέτου στο οποίο βρίσκεται. Χρησιμοποιώντας αυτό το state machine το πρωτόκολλο μπορεί να αντιληφθεί τι αντιστοιχεί στα bytes που λαμβάνει και έτσι να χτίσει το πακέτο. Όταν λάβει ολόκληρο το πακέτο, μετατρέπει τις μετρικές που έχει λάβει απο τον sensor σε unsigned 16 bit integers και τις προωθεί στο αρχείο *lunix-sensors.c*.

lunix-sensors.c

Εδώ ο κώδικας αναλαμβάνει το management των buffers που κρατάνε τις μετρικές απο τους σενσορες. Συγκεκριμένα διαχειρίζεται τα *lunix_sensor_struct* που είδαμε προηγουμένως.

Σε αυτό το αρχείο υλοποιούνται μέθοδοι για το initialization και destruction αυτών των structs. Ταυτόχρονα, υλοποιείται η μέθοδος που λαμβάνει τις μετρήσεις απο το protocol και τις αποθηκεύει στους buffers.

lunix-chrdev.{c,h}

Το *lunix-chrdev.h* ορίζει την δομή που θα χρησιμοποιήσει ο driver μας για να αποθηκεύσει μεταδεδομένα μεταξύ system calls, το *lunix_chrdev_state_struct*. Ταυτόχρονα ορίζει διάφορα consants τα οποία σχετίζονται με αυτό το struct. Με την υλοποιήση του *lunix-chrdev.c* θα ασχοληθούμε στην επόμενη ενότητα.

4 - Υλοποίηση

Σε αυτή την ενότητα παραθέτουμε την υλοποίηση του οδηγού συσκευής που μας ζητήθηκε.

Initialization & Destruction

Το πρώτο πράγμα που πρέπει να κάνει ο device driver μας είναι να προσθέσει στον kernel τον νέο driver, ζητώντας του έναν major number και ένα range απο minor numbers, ενω ταυτόχρονα του καθορίζει ποιες συναρτήσεις θα χρησιμοποιηθούν για τα διάφορα file operations. Αυτή η λειτουργικότητα υλοποιείται στα *lunix-chrdev.{c,h}* μέσα στην συνάρτηση *lunix_chrdev_init(void)*.

Αρχικά, ζητάμε απο τον kernel έναν major number και ένα range απο minor numbers. Αυτό υλοποιείται με τον ακόλουθο κώδικα:

```
dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
register_chrdev_region(dev_no, lunix_minor_cnt, name)
```

όπου η *MKDEV* αντιστοιχίζει στο *dev_no* έναν major number τιμής *LUNIX_CHRDEV_MAJOR* και έναν minor number τιμής 0.

Η *register_chrdev_region* παίρνει αυτόν τον αριθμό και τον χρησιμοποιεί για να ορίσει την πρώτη θέση του region των major-minors που θα καταλάβει ο driver μας μαζί με έναν αριθμό *lunix_minor_cnt* ο οποίος υποδηλώνει τον αριθμό των minor numbers που θα χρησιμοποιήσουμε. Τέλος, η παράμετρος *name* χρησιμοποιείται για να δώσει ένα όνομα στον driver μας ο οποίος θα εμφανίζεται στον κατάλογο */proc/devices/*.

Ο kernel αναπαριστά στο εσωτερικό του τα devices χρησιμοποιώντας το *struct cdev* structure. Όποτε για να προσθέσουμε στον kernel το device μας μαζί με τα file operations που υλοποιούν την λειτουργικότητα του, χρειάζεται να δημιουργήσουμε ένα *cdev* structure. Το δημιουργούμε και το προσθέτουμε στον kernel με τον εξής τρόπο:

```
struct cdev lunix_chrdev_cdev;
cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
```

όπου *lunix_chrdev_fops* είναι ένα instance του *file_operations* structure, το οποίο μας έχει δοθεί και ορίζει μέσω ποιων συναρτήσεων θα υλοποιηθούν τα file operations των devices μας. Τα *dev_no* και *lunix_minor_cnt* είναι τα ίδια που χρησιμοποιήθηκαν στο registration του driver.

Όταν θέλουμε να αφαιρέσουμε τον driver μας πρέπει να απελευθερώσουμε τους πόρους που καταλάβαμε στο initialization. Αυτή η λειτουργικότητα υλοποιείται στο *lunix-chrdev.{c,h}* μέσα στην συνάρτηση *lunix_chrdev_destroy(void)*.

Συγκεκριμένα, χρειάζεται να απελευθέρωσουμε τον major και τους minor numbers που καταλάβαμε και να ζητήσουμε απο τον kernel να αφαιρέσει το *cdev* structure που του είχαμε δώσει. Αυτές οι λειτουργίες υλοποιούνται ως εξής:

cdev_del(&lunix_chrdev_cdev);
unregister_chrdev_region(dev_no, lunix_minor_cnt);

Πλήρης κώδικας *lunix_chrdev_init*:

```
int lunix_chrdev_init(void)
        [12/235]
{
        /*
        * 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;
        char name[] = "Lunix:TNG";

        debug("initializing character device\n");
        /*
        * The kernel uses cdev data structures to represent char devices internally.
        * Here we initialize the lunix_chrdev_cdev pointer to cdev
        * and set its file operations to the file operations defined for this driver
        */
        cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
        lunix_chrdev_cdev.owner = THIS_MODULE;

        /* Assign a major of LUNIX_CHRDEV_MAJOR and minor of 0 */
        dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
        /*
        * Register our character device
        */
        if ((ret = register_chrdev_region(dev_no, lunix_minor_cnt, name)) < 0) {
                debug("failed to register region, ret = %d\n", ret);
                goto out;
        }
        /*
        * Now we need to let the kernel know that a new device exist by adding
        * the cdev instance we have previously initialized.
        */
        if ((ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt)) < 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;
```

```
}
```

Πλήρης κώδικας *lunix_chrdev_destroy*:

```
void lunix_chrdev_destroy(void)
{
        dev_t dev_no = 0;
        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");
}
```

Open & Close

Η συνάρτηση *lunix_chrdev_open* είναι η πρώτη που καλείται στον κόμβο συσκευής όταν γίνεται πρόσβαση σε κάποιο device, μιας και την έχουμε συσχετίσει με την μέθοδο *open* στο *file_operations* struct. Θα χρησιμοποιήσουμε την συνάρτηση *lunix_chrdev_open* για να αρχικοποιήσουμε τις διάφορες δομές που θέλουμε να συσχετίσουμε με το αρχείο που ανοίγει, καθώς και να αναγνωρίσουμε, μέσω του minor number, με ποιον αισθητηρα και με ποια μέτρηση αυτού του αισθητήρα σχετίζεται το αρχείο.

Η συνάρτηση open παίρνει δύο παραμέτρους: Έναν pointer σε *struct inode* που αντιστοιχεί στο αρχείο του file system που άνοιξε και έναν pointer σε *struct file* που αντιστοιχεί στην εσωτερική δομή δεδομένων του kernel που σχετίζεται με αρχεία.

Θα χρησιμοποιήσουμε το inode για να αναγνωρίσουμε με ποια μέτρηση ποιανού αισθητήρα σχετίζεται το αρχείο. Για να το κάνουμε αυτό θα χρησιμοποιήσουμε τον minor number του αρχείου που λαμβάνεται με την εντολή:

```
minor = iminor(inode);
```

Ο αριθμός του αισθητήρα που χρησιμοποιείται και η μέτρηση λαμβάνονται ως εξής:

```
sensor = minor / 8;
msr = minor % 8;
```

Το *struct file* που έχει δοθεί στην open, δίνεται ως παράμετρος σε κάθε συνάρτηση η οποία σχετίζεται με το συγκεκριμένο opened αρχείο. Συνεπώς, αυτό είναι το καλύτερο σημείο για να αποθηκεύσουμε πληροφορία σχετικά με το state του αρχείου. Συγκεκριμένα, η δομή file παρέχει ένα πεδίο *private_data* η οποία εξυπηρετεί αυτόν τον σκοπό. Όταν ένα αρχείο γίνεται open και δημιουργείται η δομή file αυτό το πεδίο αρχικοποιείται σε *NULL* και είναι στην ευχέρεια του προγραμματιστή τι θα του θέσει ως τιμή. Στην δική μας περίπτωση μας δίνεται το *struct lunix_chrdev_state_struct* του οποίου τα instances θα χρησιμοποιήσουμε ως την τιμή των πεδίων *private_data*.

Βλέπουμε πως το struct έχει τα αντίστοιχα πεδία:

```
struct lunix_chrdev_state_struct {
	enum lunix_msr_enum type;
	struct lunix_sensor_struct *sensor;
	int buf_lim;
	unsigned char buf_data[LUNIX_CHRDEV_BUFSZ];
	uint32_t buf_timestamp;
	struct semaphore lock;
};
```

Τα οποία θέτουμε ως εξής:

```
state = kzalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);
state->type = minor % 8;
state->sensor = &lunix_sensors[minor / 8];
state->buf_lim = 0;
state->buf_timestamp = 0;
sema_init(&state->lock, 1);
filp->private_data = state;
```

Χρησιμοποιούμε τον πίνακα *lunix-sensors* προς *lunix_sensor_struct* για να βρούμε τον κατάλληλο sensor που θα χρησιμοποιήσουμε. Αρχικοποιούμε το μέγεθος του buffer (*buf_lim*), το timestamp του τελευταίου του update (*buf_timestamp*) και τον σημαφόρο (*state->lock*).

Η πλήρης υλοποίηση της *lunix_chrdev_open* είναι:

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
	int ret;
	unsigned int minor;
	struct lunix_chrdev_state_struct *state;

	debug("entering\n");
	ret = -ENODEV;
	if ((ret = nonseekable_open(inode, filp)) < 0)
		goto out;


	state = kzalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);

	if (!state) {
		debug("failed to allocate memory");
		ret = -ENOMEM;
		goto out;
	}
```

```
        minor = iminor(inode);
        if (minor / 8 >= lunix_sensor_cnt) {
                debug("failed to find a valid sensor");
                kfree(state);
                ret = -EINVAL;
                goto out;
        }

        state->type = minor % 8;
        state->sensor = &lunix_sensors[minor / 8];
        state->buf_lim = 0;
        state->buf_timestamp = 0;
        sema_init(&state->lock, 1);

        filp->private_data = state;
out:
        debug("leaving, with ret = %d\n", ret);
        return ret;
}
```

Εν τέλει, όταν πλέον ο χρήστης δεν χρειάζεται το αρχείο, το κλείνει. Εμείς υλοποιούμε το κλείσιμο των αρχείων στην συνάρτηση *lunix_chrdev_release*. Η ευθύνη αυτής της συνάρτησης είναι η απελευθέρωση των πόρων που δεσμεύσαμε κατά το άνοιγμα του αρχείου, δηλαδή της δομής *private_data*. Συνεπώς, ο πλήρης κώδικας είναι:

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

Read

Η μέθοδος *read* είναι η καρδιά της υλοποίησης μας. Είναι υπεύθυνη για την λήψη των δεδομένων απο τους sensor buffers, την κατάλληλη μορφοποιήση τους και την επιστροφή τους στον χρήστη.

Η μέθοδος read λαμβάνει τις εξής παραμέτρους:

- *struct file *filp*: Είναι ο pointer για το *file* structure που σχετίζεται με το αρχείο. Στο πεδίο *private_data* έχουμε αποθηκεύσει τα metadata που έχουμε ορίσει για το αρχείο.
- *char __user *usrbuf*: Είναι ένας pointer σε σημείο μνήμης του userspace. Όταν δουλεύουμε με μνήμη απο userspace χρειάζεται ιδιαίτερη προσοχή. Συγκεκριμένα, θεωρείται κακή πρακτική να κάνουμε dereference αυτόν τον pointer, μιας και μπορεί να μην είναι έγκυρος όταν βρισκόμαστε σε kernel mode. Επίσης η σελίδα που αφορά αυτό το κομμάτι μνήμης, μπορεί να μην βρίσκεται στην μνήμη και να δημιουργηθεί page fault, πράγμα το οποίο δεν επιτρέπεται όσο βρισκόμαστε σε kernel mode. Τέλος, και σημαντικότερο, ο kernel δεν πρέπει να εμπιστεύεται οποιοδήποτε δεδομένο έρχεται απο userspace, γιατί αν του δοθεί malicious input ένας attacker θα

μπορούσε να αποκτήσει απεριόριστη πρόσβαση σε όλο το σύστημα (π.χ. μέσω buffer overflow).
- *size_t count*: Είναι ένας αριθμός που εκφράζει πόσα bytes ζήτησε να διαβάσει ο χρήστης.
- *loff_t *f_pos*: Έκφράζει το offset του αρχείου στο οποίο βρισκόμαστε.

Για τους λόγους που παραθέσαμε, θα χρησιμοποιήσουμε την συνάρτηση *copy_to_user* για να μεταφέρουμε δεδομένα στο userspace. Η συνάρτηση αυτή παρέχεται απο τον kernel και υλοποιεί μια ασφαλή μεταφορά δεδομένων απο kernel space σε user space. Μιας και η σελίδα στο user space μπορεί να μην βρίσκεται στην μνήμη, μπορεί να δημιουργήθει κάποιο page fault και η διεργασία μας να μπει σε sleep. Για αυτό το λόγο επιβάλλεται η χρήση μεθόδων συγχρονισμού. Μιας και βρισκόμαστε σε process context ο συγχρονισμός θα υλοποιηθεί με την χρήση σημαφόρων. Οπότε στην αρχή και στο τέλος της μεθόδου μας θα έχουμε τα ακόλουθα κομμάτια κώδικα:

```
if (down_interruptible(&state->lock)) {
      return -ERESTARTSYS;
}
...
...
up(&state->lock);
```

Η μεταβλητή *state* δείχνει στην *lunix_chrdev_state_struct* που αποθηκεύσαμε στο πεδίο *private_data* του *file* structure κατά το άνοιγμα του αρχείου.

Αμα δεν μας δοθεί το lock στέλνουμε σήμα *ERESTARTSYS* για να ενημερώσουμε τον kernel να ξαναδοκιμάσει να τρέξει την διεργασία.

Η μέθοδος read του οδηγού μας, πρέπει πάντα να επιστρέφει τις τελευταίες μετρικές του sensor απο τον οποίο ζητούνται. Συνεπώς, χρειαζόμαστε έναν αλγόριθμο ο οποίος ελέγχει αν ο buffer του *lunix_chrdev_state_struct* περιέχει τις τελευταίες μετρήσεις ή αν μας είναι διαθέσιμες καινούριες μετρήσεις. Για την υλοποίηση αυτού του αλγοριθμού χρησιμοποιούμε την συνάρτηση *lunix_chrdev_state_needs_refresh* με παράμετρο το *lunix_chrdev_state_struct*. Η τιμή επιστροφής της συνάρτησης είναι 1 αν χρειάζεται να γίνουν update τα δεδομένα του buffer, αλλιώς είναι 0. Η υλοποίηση της είναι:

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
{
        struct lunix_sensor_struct *sensor = state->sensor;
        uint16_t ret = 0;

        spin_lock(&sensor->lock);
        if (state->buf_timestamp < sensor->msr_data[state->type]->last_update) {
                ret = 1;
        }
        spin_unlock(&sensor->lock);
        return ret;
}
```

Η συνάρτηση αυτή θέλουμε να μπορεί να καλείται και απο interrupt context για λόγους που θα εξηγήσουμε παρακάτω. Συνεπώς, χρησιμοποιούμε spinlocks αντι για σημαφόρους, μιας και δεν επιτρέπεται να μπούμε σε sleep καθώς βρισκόμαστε σε interrupt context. Ένας άλλος λόγος που χρησιμοποιούμε το spinlock είναι διότι η υλοποίηση των δομών των sensors (*lunix-sensors.{c,h}*) χρησιμοποιεί το ίδιο spinlock που χρησιμοποιουμε και εμείς και βρίσκεται στην δομή *lunix_sensor_struct*.

Η λειτουργία της συνάρτησης είναι απλή. Κοιτάει αμα το timestamp που έχουμε αποθηκεύσει ως την στιγμή που έγινε τελευταία φορά update ο buffer του οδηγού μας είναι μικρότερος απο το timestamp που δείχνει την τελευταία στιγμή όπου έγινε update η μετρική του sensor που ζητάμε. Αν ναι, τότε επιστρέφουμε 1, δηλαδή χρειάζεται να κάνουμε update τον buffer του οδηγού, αλλιώς επιστρέφουμε 0.

Συνεπώς, αφού έχουμε μεθοδολογία για να αναγνωρίζουμε πότε χρειάζεται να ανανεώσουμε τις εγγραφές του buffer μας, χρειαζόμαστε τώρα έναν αλγόριθμο για να κάνει αυτή την ανανέωση. Για αυτό τον σκοπό υλοποιούμε την συνάρτηση *lunix_chrdev_needs_update(struct lunix_chrdev_state_struct *state)*. Η υλοποίηση της είναι:

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
        struct lunix_sensor_struct *sensor;
        uint16_t num = 0;
        uint32_t timestamp = 0;
        char name[10];

        sensor = state->sensor;
        spin_lock(&sensor->lock);

        num = sensor->msr_data[state->type]->values[0];
        timestamp = sensor->msr_data[state->type]->last_update;

        spin_unlock(&sensor->lock);

        switch (state->type) {
                case BATT:
                        sprintf(name, "BATT");
                        num = lookup_voltage[num];
                        break;
                case TEMP:
                        sprintf(name, "TEMP");
                        num = lookup_temperature[num];
                        break;
                case LIGHT:
                        sprintf(name, "LIGHT");
                        num = lookup_light[num];
                        break;
                case N_LUNIX_MSR:
                        break;
        }
        state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%s=%d.%d\n", name, num /
1000, num % 1000);
        state->buf_timestamp = timestamp;

        return 0;
}
```

Στην αρχή, χρησιμοποιούμε το spinlock του *lunix_sensor_struct* του sensor, ώστε να ανακτήσουμε τα τελευταία δεδομένα με ασφαλή τρόπο. Συγκεκριμένα, λαμβάνουμε την τελευταία μετρική και το timestamp της στιγμής στην οποία έγινε. Στη συνέχεια χρησιμοποιώντας τα lookup tables τα οποία μας έχουν δοθεί, λαμβάνουμε την αντιστοίχιση της μετρικής. Τέλος, ανανεώνουμε τον buffer με το formatted string που θέλουμε να εμφανίζεται στον χρήστη, χρησιμοποιώντας την μέθοδο *snprintf*. Επιλέξαμε την *snprintf* αντι της *sprintf* για την αποφυγή των buffer overflows, μιας και ένα buffer overflow μπορεί να έχει καταστροφικές συνέπειες για τον kernel. Τέλος, σημειώνουμε το timestamp της τελευταίας μετρικής και τα όρια του buffer.

Πλέον, έχουμε όλα τα εργαλεία που χρειαζόμαστε για να υλοποιήσουμε την μέθοδο read. Πρέπει να λάβουμε υπόψη δύο πράγματα:

1. Την επιστροφή όσων ή λιγότερων απο τα byte που μας ζητούνται μέσω της παραμέτρου *size_t count*.
2. Όταν δεν έχουμε κάποια καινούργια μετρική να δώσουμε στον χρήστη, η διεργασία πρέπει να μπαίνει σε sleep.

Για το πρώτο υπάρχουν οι εξής περιπτώσεις:

- *state->buf_lim <= count*: Σε αυτή την περίπτωση επιστρέφουμε όλα τα περιεχόμενα του buffer.
- *state->buf_lim > count*: Σε αυτή την περίπτωση επιστρέφουμε *count* bytes, ενω αυξάνουμε τον *f_pos* pointer κατά *count*. Ιδιαίτερη προσοχή πρέπει να δοθεί στην διατήρηση σωστής τιμής στον *f_pos* μιας και θέλουμε να τον επιστρέφουμε σε τιμή 0 (δηλαδή ζητείται καινούργια μετρική) όταν επιστρέψουμε όλα τα περιεχόμενα του τρέχοντος buffer.

Αυτό υλοποιείται ως εξής:

```
        if (state->buf_lim <= cnt) {
                lim = state->buf_lim;
                strncpy(data, state->buf_data, state->buf_lim);
        }
        else {
                for (lim = 0; lim < cnt && *f_pos + lim < state->buf_lim; ++lim) {
                        data[lim] = state->buf_data[*f_pos + lim];
                }
                *f_pos += lim;
                // end of file, we need a new metric
                if (*f_pos >= state->buf_lim) {
                        *f_pos = 0;
                }
        }
        copy_to_user(usrbuf, data, lim);
```

Για το δεύτερο, πρέπει η διεργασία να μπαίνει σε sleep όταν

1. *f_pos == 0, δηλαδή ζητείται νέα μετρική, και
2. Δεν υπάρχει νέα μετρική.

Σε αυτό το σημείο, θα να αναφέρουμε κάποιους κανόνες σχετικά με το sleep. Αρχικά, πρέπει να αποφεύγουμε να μπαίνουμε σε sleep όταν κρατάμε locks. Συγκεκριμένα, απαγορεύεται να μπαίνουμε σε sleep όσο κρατάμε spinlock. Ο kernel επιτρέπει να μπούμε σε sleep όταν κρατάμε σημαφόρο αλλά θα πρέπει να το αποφεύγουμε μιας και μπορεί να βρεθούμε σε κατάσταση deadlock. Αρα, στην περίπτωση μας, πριν μπούμε σε sleep θα απελευθερώσουμε τον σημαφόρο.

Επίσης, αφού βγούμε απο sleep mode, δεν γνωρίζουμε για πόση χρονική διάρκεια και τι άλλαξε όσο βρισκόμασταν σε sleep. Μπορεί επίσης μια άλλη διεργασία να περιμένει για το ίδιο event που μας ξύπνησε. Για αυτό το λόγο, αφού ξυπνήσουμε ελέγχουμε τις συνθήκες οι οποίες μας οδήγησαν στο να μπούμε σε sleep και δοκιμάζουμε να ξαναπάρουμε το lock.

Τέλος, δεν πρέπει να μπαίνουμε σε sleep αμα δεν γνωρίζουμε με σιγουριά πως κάποιο άλλο process θα μας ξυπνήσει.

Έχοντας κατα νου αυτούς τους κανόνες, η υλοποίηση του μηχανισμού που βάζει το process σε sleep αν ικανοποιούνται οι απαραίτητες συνθήκες και όταν ξυπνάει κάνει update τον buffer είναι:

```
        if (*f_pos == 0) {
            while (!lunix_chrdev_state_needs_refresh(state)) {
                up(&state->lock);
                if (filp->f_flags & O_NONBLOCK) {
                    return -EAGAIN;
                }
                if (wait_event_interruptible(sensor->wq, (lunix_chrdev_state_needs_refresh(state))))
{
                    return -ERESTARTSYS;
                }
                if (down_interruptible(&state->lock)) {
                    return -ERESTARTSYS;
                }
            }
            lunix_chrdev_state_update(state);
        }
```

Χρησιμοποιούμε τις συναρτήσεις *lunix_chrdev_state_needs_refresh* και *lunix_chrdev_state_update* που ορίσαμε πιο πάνω για να αναγνωρίσουμε αν χρειάζεται update και να κάνουμε το update αντίστοιχα.

Αφού απελευθερώσουμε τον σημαφόρο, ελέγχουμε αν έχει δοθεί το flag *O_NONBLOCK* όπου ο χρήστης ζητάει να μην μπει σε sleep η διεργασία και επιστρέφουμε *-EAGAIN*, δηλαδή να ξαναπροσπαθήσει μιας και σκοπεύουμε να μπούμε σε sleep.

Στη συνέχεια, χρησιμοποιούμε την *wait_event_interruptible*. Η συνάρτηση αυτή, βάζει το τρέχων process σε ένα queue απο processes που περιμένουν ένα event. Στην δική μας περίπτωση χρησιμοποιούμε το *struct wait_queue_head_t* που χρησιμοποιείται απο το *struct lunix_sensor_struct* που περιέχει τις πληροφορίες για τον sensor. Όταν ο sensor λάβει νέα δεδομένα, χρησιμοποιεί αυτό το queue για να στείλει σήμα σε όλα τα processes τα οποία

περιμένουν νέα δεδομένα και να ξυπνήσουν. Η δεύτερη παράμετρος είναι μια συνθήκη η οποία πρέπει να είναι αληθής έτσι ώστε να ξυπνήσει το process. Η αληθοτιμή αυτής της παραμέτρου ελέγχεται σε interrupt context και πρέπει να είναι σύντομη μιας και μπορεί να καλεστεί πολλές φορές.

Τέλος, αφού ξυπνήσει το process, ελέγχει αν μπορεί να ξανα αποκτήσει τον σημαφόρο και ξανα ελέγχει την συνθήκη που το οδήγησε σε sleep, και αν δεν χρειάζεται να μπει σε sleep λαμβάνει τα νέα δεδομένα.

Η πλήρης υλοποίηση της read είναι:

```
static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt, loff_t *f_pos)
{
        ssize_t ret = 0;

        struct lunix_sensor_struct *sensor;
        struct lunix_chrdev_state_struct *state;
        char data[LUNIX_CHRDEV_BUFSZ];
        ssize_t lim;

        state = filp->private_data;
        WARN_ON(!state);

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

        /* Lock state semaphore */
        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_needs_refresh(state)) {
                        up(&state->lock);
                        // if no block is specified ask to try again.
                        if (filp->f_flags & O_NONBLOCK) {
                                return -EAGAIN;
                        }
                        debug("Putting process to sleep\n");
                        // wait for event
                        if (wait_event_interruptible(sensor->wq,
(lunix_chrdev_state_needs_refresh(state)))) {
                                // We have been woken up by an event
                                // Let the upper virtual filesystem (VFS) layer handle it.
                                return -ERESTARTSYS;
                        }
                        // see if we can get the lock again
                        if (down_interruptible(&state->lock)) {
                                return -ERESTARTSYS;
                        }
                }
                // ok data is here.
```

```
                lunix_chrdev_state_update(state);
        }
        if (state->buf_lim <= cnt) {
                lim = state->buf_lim;
                strncpy(data, state->buf_data, state->buf_lim);
        }
        else {
                // copy data starting from *f_pos to *f_pos + cnt
                for (lim = 0; lim < cnt && *f_pos + lim < state->buf_lim; ++lim) {
                        data[lim] = state->buf_data[*f_pos + lim];
                }
                *f_pos += lim;
                // end of file, we need a new metric
                if (*f_pos >= state->buf_lim) {
                        *f_pos = 0;
                }
        }
        ret = lim;
        if (copy_to_user(usrbuf, data, lim)) {
                ret = -EFAULT;
                goto out;
        }
out:
        /* Unlock */
        up(&state->lock);
        return ret;
}
```

ioctl

Για το operation *ioctl* δεν προσφέρουμε κάποια λειτουργικότητα και επιστρέφουμε *-EINVAL*.

```
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
        /*
        * We return -EINVAL since we do not support any
        * feature that can fall under between the read/write spectrum
        */
        return -EINVAL;
}
```

Write

Ο οδηγός μας δεν υποστηρίζει write functionality στα character device files οπότε για το write δεν χρειάζεται να κάνουμε τίποτα.

5 - Έλεγχος Ορθότητας

Σε αυτή την ενότητα θα ελέγξουμε αν ο οδηγός μας λειτουργεί σωστά όταν πολλά προγράμματα προσπαθούν να διαβάσουν τις μετρήσεις κάποιου σένσορα.

Αρχικά τρέχουμε το ακόλουθο:

```
for i in {0..10}; do cat /dev/lunix1-light > out$i & done
```

Και παρατηρούμε πως τα αρχεία *out$i* έχουν πάντα το ίδιο περιεχόμενο, συνεπώς ο οδηγός λειτουργεί σωστά όταν πολλοί προσπαθούν να αποκτήσουν πρόσβαση σε μια μέτρηση.

Στη συνέχεια, θα δοκιμάσουμε να πάρουμε ταυτόχρονα διαφορετικές μετρήσεις:

```
for i in {0..10}; do cat /dev/lunix1-light > out$i & done
for i in {10..20}; do cat /dev/lunix1-batt > out$i & done
```

Και παρατηρούμε πως τα αρχεία out$i έχουν και τα δύο σωστό περιεχόμενο. Αν και δεν μπορούμε να θεωρήσουμε αυτές τις δοκιμές ως unit tests, είναι μια καλή ένδειξη ότι ο οδηγός λειτουργεί όπως αναμένουμε.