ΕΡΓΑΣΤΗΡΙΟ ΛΕΙΤΟΥΡΓΙΚΩΝ
ΑΝΑΦΟΡΑ 2ης ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ

ΓΟΥΛΙΑΜΟΥ ΜΑΡΙΑ-ΕΘΕΛ
ΝΤΟΥΡΟΣ ΕΥΑΓΓΕΛΟΣ

30 Νοεμβρίου 2019

1 Περιγραφή άσκησης

Σε αυτήν την άσκηση ολοκληρώσαμε έναν οδηγό συσκευής για τους αισθητήρες Lunix:TNG. Ουσιαστικά υλοποιήσαμε το κομμάτι που είναι πιο κοντά στο χώρο χρήστη, το lunix-chrdev.c , που είναι ένας character device driver.

Το κομμάτι αυτό του οδηγού, υλοποιεί την επικοινωνία με τον χρήστη, υλοποιώντας τις κλήσεις συστήματος:

- **open()**
- **release()**
- **read()**
- **ioctl()**
- **mmap()**

Ώς συσκευή είχαμε 16 ίδιους αισθητήρες που λάμβαναν μετρήσεις με τρόπο περιοδικό (θερμοκρασία, ένταση φωτός, τάση μπαταρίας) και εξέπεμπαν τα δεδομένα σε ένα σταθμό. Αυτός τα έστελνε μέσω σειριακής θύρας σε ένα μηχάνημα του εργαστηρίου. Ύστερα αυτά αποστέλλονταν μέσω δικτύου σε δικιά μας εικονική μηχανή. Αυτή ήταν ρυθμισμένη να τα περνά αυτόματα στη σειριακή θύρα ttyS0.

Η σειριακή μας θύρα, λοιπόν, φαινόταν να δέχεται -περιοδικά- bytes από όλους τους αισθητήρες και όλες τις μετρήσεις ανακατεμένα.

Η πρόκληση, λοιπόν, ήταν να ξεχωρίσουμε αυτά τα δεδομένα, να τα αντιστοιχίσουμε σε πραγματικές τιμές, και να λύσουμε θέματα race conditions μεταξύ των διαφόρων διεργασιών που θα προσπαθούν να τα λάβουν.

Εκτός από τις βασικές κλήσεις συστήματος υλοποιήσαμε την ioctl και την mmap, όπως εξηγείται παρακάτω.

2 Περιγραφή κώδικα

Παρακάτω περιγράφεται συνοπτικά η ανάπτυξη των συναρτήσεων που στελέχωσαν το αρχειο lunix-chrdev.c.

refresh()
Αυτή η συνάρτηση καλείται μόνο για να ελέγξει αν το timestamp που είναι αποθηκευμένο στην δομή του state είναι το ίδιο με αυτό το οποίο έχει γραφεί στο κατώτερο επίπεδο του lunix-sensors. Αν είναι τότε η κλήση της επιστρέφει 0, αλλιώς 1. Καλείται όταν κάποια διεργασία βρίσκεται στην read, ως συνθήκη για το άν θα κοιμηθεί ή όχι.

update()
Αυτή η κλήση γίνεται μέσα στην κλήση συστήματος της read όταν ο user ζητάει να διαβάσει κάποια bytes. Για την ανανέωση των δεδομένων ελέγχεται αν η χρονική στιγμή της τελευταίας ανανέωσης είναι μικρότερη από τη χρονικη στιγμή της τελευταίας ανανέωσης των τιμών των sensor buffers. Τότε θα πρέπει να κλειδώσουμε το spinlock , να πάρουμε γρηγορα τα δεδομένα που ζήτησε ο χρήστης (temperature/light/battery voltage) και να κάνουμε update το προσωρινό timestamp. Μετά το ξεκλείδωμα θα το αποθηκεύσουμε στην δομή του state και θα επεξεργαστούμε την τιμή που ζήτησε ο χρήστης ανάλογα αν πρόκειται για raw ή coocked δεδομένα, όπως εξηγείται παρακάτω στην περιγραφή της κλήσης ioctl .

open()
Ο χρήστης καλεί τη συγκεκριμένη κλήση συστήματος για να ανοίξει ένα ειδικό αρχείο ώστε να αρχίσει την επικοινωνία του με τη συσκευή. Εδώ βρίσκουμε σε ποιά μέτρηση ποιανού αισθητήρα θέλει να αποκτήσει πρόσβαση, μέσω του minor number της συσκευής. Επίσης δεσμεύουμε χώρο για το struct της κατάστασης της συσκευής.

release()
Αυτή η κλήση συστήματος καλείται όταν τελειώνει η πρόσβαση του χρήστη στη συσκευή και σκοπός της είναι απλά να ελευθερώσει όσο χώρο έχει δεσμέυσει ο driver.

read()
Η read είναι η βασική κλήση συστήματος της υλοποιησής μας. Όταν καλείται, ο driver ελέγχει αν υπάρχουν νέα δεδομένα για τον χρήστη (αν έχει τελειώσει με το διάβασμα των παλιών). Αν ναι, περιμένει να τα λάβει σε μια ουρά (εκτός από την περίπτωση που η συσκευή έχει ανοιχτεί με O_NONBLOCK flag, στην οποία επιστρέφει κατευθείαν στον χρήστη). Για την ανανέωση των δεδομένων καλεί τη συνάρτηση update. Ύστερα υπολογίζει πόσα bytes να επιστρέψει στον χρήστη και τα αντιγράφει στον καθορισμένο buffer της καταστασης της συσκευής.

init()
Αυτή η συνάρτηση καλείται μόνο μία φορά κατά την εκτέλεση της εντολής insmod που εισάγει ένα καινούριο module στον κώδικα του πυρήνα. Όπως φαίνεται και στον κώδικα αυτής της κλήσης συστήματος, ουσιαστικά δηλώνεται μια συσκευή χαρακτήρων με την κλήση της cdev_init() και ζητάμε 8 minor numbers για κάθε sensor , με την register_chrdev_region. Αφού τα δεσμέυσουμε καλούμε την cdev_add και στο εξής η συσκευή μας είναι ¨ζωντανή' και ¨ακούει' σε όποια κλήση συστήματος του χώρου χρήστη.

ioctl()
Η ioctl() είναι μια απλή κλήση συστήματος που δίνει την δυνατότητα στον χρήστη να επιλέξει ανάμεσα σε 2 μορφές εξόδου των μετρήσεων που μας δίνει ο αισθητήρας. Αυτό γίνεται μέσω του ορίσματος που μπορεί να πάρει την τιμή 0 ή 1 που αντιστοιχεί σε format εξόδου Raw ή Cooked αντίστοιχα. Στην πρώτη περίπτωση ο χρήστης λαμβάνει τα δεδομένα ακατέργαστα όπως αυτά παράγονται από τους αισθητήρες. Για τη δεύτερη περίπτωση, έχει γραφεί για εμάς 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 δεδομένων σε coocked τιμές. Ύστερα, αρκούν πολύ απλές πράξεις για να βρεθούν το ακέραιο μέρος και το κλασματικό του αριθμού.

Αλλαγή του header file
Προστέθηκαν κάποιες γραμμές κώδικα στο αρχείο lunix-chrdev.h ώστε να υλοποιηθεί η παραπάνω κλήση συστήματος.

testIoctl.c
Ένα απλό πρόγραμμα, για τον έλεγχο της σωστής λειτουργίας της κλήσης ioctl, που δέχεται ως ορίσματα το αρχείο προς ανάγνωση και 0 ή 1 ανάλογα αν θέλουμε coocked ή raw δεδομένα, αντίστοιχα.

mmap()
Επίσης υλοποιήσαμε και την κλήση συστήματος mmap για τη συσκευή μας. Η mmap επιταχύνει την πρόσβαση σε δεδομένα της συσκευής από το χρήστη αφού εξαλείφει το overhead των συνεχόμενων κλήσεων συστήματος και της αντιγραφής δεδομένων από τον χώρο πυρήνα στον χώρο χρήστη. Επείτα από επιτυχή χρήση της, ο χρήστης έχει πρόσβαση σε κάποιες σελίδες μνήμης που έχει πρόσβαση και ο driver της συσκευής, οπότε η διαδικασία απόκτησης δεδομένων απλοποιείται σημαντικά. Για την απλότητα της υλοποιήσής μας επιλέξαμε να επιστρέφει μία σελίδα στο χρήστη. Αν αναλογιστούμε ότι αυτή είναι αρκετή για τα δεδομένα που θέλει ο χρήστης, η υλοποίηση δεν εισάγει κανένα περιορισμό.

testMmap.c
Ένα απλό πρόγραμμα, για τον έλεγχο της σωστής λειτουργίας της κλήσης mmap , που δέχεται ως όρισμα το αρχείο προς ανάγνωση.

3 Κώδικας

refresh()

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
{
    int ret = 0; //by default we are assuming that the state doesnt need refreshing
    struct lunix_sensor_struct *sensor;

    WARN_ON ( !(sensor = state->sensor));
    if (sensor->msr_data[state->type]->last_update != state->buf_timestamp)
    {
        debug("State needs refreshing!\n");
        ret = 1; //now state needs refreshing
    }

    return ret;
}
```

update()

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
    struct lunix_sensor_struct *sensor;
    unsigned long flags;        //spinlock's flag is an unsigned long int
    uint16_t value;             //lookup tables require uint16_t
    long int result;            //lookup tables give long int
    uint32_t current_timestamp; //defined in lunix_msr_data_struct
    int ret;

    debug("entering\n");
    sensor = state->sensor;
    ret = -EAGAIN; //there is no data available right now, try again later

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

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

    /*
     * Any new data available?
     *
     * Now we can take our time to format them, holding only the private state
     * semaphore. This is implemented in open syscall.
     */
    if(sensor->msr_data[state->type]->last_update > state->buf_timestamp)
    {
        if (!state->raw_data)
        {   //coocked data
            if (state->type == BATT)
            {
                result = lookup_voltage[value];
            }
            else if (state->type == TEMP)
```

```
            {
                result = lookup_temperature[value];
            }
            else if (state->type == LIGHT)
            {
                result = lookup_light[value];
            }
            else
            {
                debug("Internal Error: Type doesnt match one the three available \
                                                        (BATT, TEMP, LIGHT)");
                ret = -EMEDIUMTYPE;     //wrong medium type
                goto out;
            }

            /*save formatted data in chrdev state buffer*/
            ret = 0;
            state->buf_timestamp = current_timestamp;
            state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, \
                                "%ld.%03ld\n", result/1000, result%1000);
        }
        else{   //raw data
            debug("skipped lookup table conversion, returning raw bytes...\n");
            ret = 0;
            state->buf_timestamp = current_timestamp;
            state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, \
                                "%x\n", value); //prints raw_data as hex
        }
    }
    else
    {
        ret = -EAGAIN;
    }

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

open()

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

    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>]
     */
    minor = MINOR(inode->i_rdev);    //LDD3 page 55
    sensor = minor/8;                // 0-15
    type = minor%8;                  // 0-2
    debug("Done assosiating file with sensor %d of type %d\n", sensor, type);

    /* Allocate a new Lunix character device private state structure */

    //GFP_KERNEL    This is a normal allocation and might block.
    //This is the flag to use in process context code when it is safe to sleep
    state = kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);
    state->type = type;
    state->sensor = &lunix_sensors[sensor];
```

```
    /*buffer init*/
    state->buf_lim = 0;
    state->buf_timestamp = 0;
    filp->private_data = state; //points to the current state of the device
                               //stores a pointer to it for easier access
    state->raw_data = 0;       //by default, in coocked data mode

    sema_init(&state->lock,1);  //initialize semaphore with 1, refers to a single struct file
    ret = 0;
out:
    debug("leaving, with ret = %d\n", ret);
    return ret;
}
```

release()

```
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
    debug("Releasing allocated memory for private file data\n");
    kfree(filp->private_data);
    debug("Done releasing private file data, exiting now..");
    return 0;
}
```

read()

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

    struct lunix_sensor_struct *sensor;
    struct lunix_chrdev_state_struct *state;

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

    sensor = state->sensor;
    WARN_ON(!sensor);
    debug("entering!\n");

    /*
     * Lock, in case processes with the same fd
     */
    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 the file was opened with O_NONBLOCK flag return -EAGAIN
            if (filp->f_flags & O_NONBLOCK)
                return -EAGAIN;
            //sleep, if condition is TRUE, if you re woken up check condition again and sleep or leave
            if (wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state)))
                //wait_event_interruptible returns nonzero when interrupted by signal
                return -ERESTARTSYS;
            //now grab again the lock because you woke up, and continue the read
            if (down_interruptible(&state->lock))
```

```
                return -ERESTARTSYS;
            }
        }
        debug("Ok, now I have fresh values\n");

        /* Determine the number of cached bytes to copy to userspace */
        remaining_bytes = state->buf_lim - *f_pos;

        if (cnt >= remaining_bytes)
        {
            cnt = remaining_bytes;
        }

        /*
         * copy_to_user(to, from, count)
         * returns the number of bytes that couldnt copy
         */
        if (copy_to_user(usrbuf, state->buf_data + *f_pos, cnt))
        {
            ret = -EFAULT;
            goto out;
        }

        //fix the position
        *f_pos = *f_pos + cnt;
        ret = cnt;

        /* Auto-rewind on EOF mode */
        if (*f_pos == state->buf_lim)
            *f_pos = 0;

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

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");

        //Initialize the struct. Give it the file operations.The device responds to multiple numbers.
        cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
        lunix_chrdev_cdev.owner = THIS_MODULE;

        /*
         * register_chrdev_region
         * Give multiple device numbers to the same device.
         * "The device responds to multiple device numbers."
         */
        dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0); //the first device number to which the device responds

        ret = register_chrdev_region(dev_no, lunix_minor_cnt, "Lunix:TNG");
        if (ret < 0) {
            debug("failed to register region, ret = %d\n", ret);
            goto out;
        }
        debug("device registered successfully\n");
```

```
    /* cdev_add: From now on the device is "alive" and shall listen to method requests */
    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;
}
```

destroy()

```
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");
}
```

ioctl()

```
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    struct lunix_chrdev_state_struct *state;
    debug("entering\n");

    //if cmd's type is not LUNIX_IOC_MAGIC (it is 60, the major number) return error
    if (_IOC_TYPE(cmd) != LUNIX_IOC_MAGIC) return -ENOTTY;
    //accept only 1 cmd
    if (_IOC_NR(cmd) > LUNIX_IOC_MAXNR) return -ENOTTY;

    state = filp->private_data;

    switch (cmd)
    {
        case LUNIX_IOC_DATA_TYPE_CONVERT:
            if (down_interruptible(&state->lock)) //in case multiple procs with same fd use ioctl
                return -ERESTARTSYS;
            //if I have raw data I turn them into coocked and vice versa
            if (state->raw_data)
                state->raw_data = 0; //turned into coocked
            else
                state->raw_data = 1; //turned into raw
            up(&state->lock);

    }

    debug("successfully changed data type transfer, now state->raw_data=%d\n", state->raw_data);
    return 0;
}
```

Additions in header file

```
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;

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



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

#define LUNIX_IOC_MAXNR           1
```

testIoctl.c

```
/*
 * This program tests the ioctl method
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <string.h>

#include "lunix-chrdev.h"

int main(int argc, char **argv)
{
    int mode, fd, status;
    pid_t pid;
    size_t bytes_read;
    char buffer;

    if (argc != 3)
    {
        fprintf(stderr, "Usage: ./testIoctl /dev/lunixX-XXXX Y[0/1], 1:for raw_data\n");
        return 1;
    }

    mode = atoi(argv[2]);
    if ((mode != 0) & (mode !=1))
    {
        fprintf(stderr,"Third argument has to be 0 or 1, 1 for raw_data\n");
        return 1;
    }

    fd = open(argv[1],O_RDONLY);
```

```
    if (fd < 0)
    {
        perror(argv[1]);
        return 1;
    }

    if (mode)
    {
        printf("changed to raw bytes mode\n");
        if (ioctl(fd, LUNIX_IOC_DATA_TYPE_CONVERT, NULL)) //change to raw bytes
        {
            perror("ioctl");
            return 1;
        }
    }

    pid = fork();
    if (pid == 0)
    {
        do
        {
            bytes_read = read(fd, &buffer, 1);
            if (bytes_read < 0)
            {
                perror("read");
                exit(1);
            }

            printf("%c", buffer);
        } while (bytes_read > 0);

        //Unreachable
        perror("reached non reachable point, internal error");
        exit(0);
    }

    wait(&status);

    return 0;
}
```

mmap

```
/*
 * Open and close methods of mmap. Not nessesary is this implementation though.
 */
void simple_vma_open(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "simple VMA open, virt %lx, phys %lx\n", \
            vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void simple_vma_close(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "simple VMA close.\n");
}

static struct vm_operations_struct simple_remap_vm_ops =
{
    .close = simple_vma_close,
    .open = simple_vma_open,
};

static int lunix_chrdev_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct lunix_chrdev_state_struct *state;
    struct lunix_sensor_struct *sensor;
    struct page *kernel_page;
    unsigned long long *kernel_page_address;
```

```
    state = filp->private_data;
    sensor = state->sensor;

    //get a pointer to the virtual address's associated struct page
    kernel_page = virt_to_page(sensor->msr_data[state->type]->values);
    //get the virtual address of the struct page
    kernel_page_address = page_address(kernel_page);
    //get the pgysical address of the above virtual one
    vma->vm_pgoff = __pa(kernel_page_address) >> PAGE_SHIFT;

    //map the page to the userspace
    if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, \
                        vma->vm_end - vma->vm_start,\
                        vma->vm_page_prot))
        return -EAGAIN;

    //unnessesary, just written for code wholeness, it could be deleted.
    vma->vm_ops = &simple_remap_vm_ops;
    simple_vma_open(vma);
```

testMmap.c

```
/*
 * A program to test mmap method
 */
#include <unistd.h>
#include <stdlib.h>
#include <inttypes.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>

struct lunix_msr_data_struct {
    uint32_t magic;
    uint32_t last_update;
    uint32_t values[];
};

int main(int argc, char **argv)
{
    int fd;
    struct lunix_msr_data_struct *addr;
    unsigned int buf_timestamp, my_timestamp = 0;

    if (argc != 2)
    {
        fprintf(stderr, "Usage: ./testMmap /dev/lunixX-XXXX\n");
        return 1;
    }

    fd = open(argv[1], O_RDONLY);
    if (fd < 0)
    {
        perror("open");
        return 1;
    }
    printf("Opened special file %s with fd: %d\n", argv[1], fd);

    //ask for something smaller or equal to a page size.
    //then by default one page will returns
    //note that the drivers mmap is implemented to return only one page
    addr = mmap(NULL, 20, PROT_READ, MAP_PRIVATE, fd, 0);
    if (addr == MAP_FAILED)
    {
        perror("mmap");
        return 1;
    }

    while(1)
```

```
    {
        buf_timestamp = addr->last_update;
        if(buf_timestamp != my_timestamp)
        {
            my_timestamp = buf_timestamp;
            printf("%x\n", addr->values[0]);
        }
    }
    return 0;
}
```