Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών Ε.Μ.Π.

ΕΡΓΑΣΤΗΡΙΟ ΛΕΙΤΟΥΡΓΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

2015-2016

ΑΣΚΗΣΗ 2η

Χρυσούλα Βαρηά
ΑΜ: 03112105
Ευαγγελία-Σοφία Γεργατσούλη
ΑΜ: 03112064
Εξάμηνο:8ο

1 Οι συναρτήσεις του device driver

- *lunix_chrdev_needs_refresh*:
  Η παραπάνω συνάρτηση καλείται μέσω των lunix_chrdev_state_update, lunix_chrdev_read, δέχεται μια δομή lunix_chrdev_state_struct και ελέγχει αν τα δεδομένα της δομής χρειάζονται ανανέωση. Αυτό επιτυγχάνεται με τη σύγκριση του χρόνου της τελευταίας ανανέωσης των δεδομένων του αισθητήρα (μέσω της μεταβλητής last_update) με την μεταβλητή buf_timestamp, η οποία δηλώνει την τελευταία ενημέρωση των δεδομένων του buffer της lunix_chrdev_state_struct.

- *lunix_chrdev_state_update*:
  Η lunix_chrdev_state_update καλείται μέσω της lunix_chrdev_read και ελέγχει αν υπάρχουν νέα δεδομένα στον buffer του αισθητήρα (μέσω της μεταβλητής state->type γνωρίζουμε ποιον buffer πρέπει να ελέγξουμε) για την ενημέρωση του buffer της δομής lunix_chrdev_state_struct.
  Για τη λήψη των δεδομένων από τον αισθητήρα χρησιμοποιούνται τα spin_lock_irqsave(&sensor->lock,zflags), spin_unlock_irqrestore(&sensor->lock, zflags) για την εισαγωγή στο κρίσιμο τμήμα. Τα locks είναι απαραίτητα για να εξασφαλίζεται αμοιβαίος αποκλεισμός μεταξύ διεργασιών, οι οποίες επιχειρούν να διαβάσουν τα δεδομένα του αισθητήρα, ενώ συγχρόνως υπάρχουν νέα δεδομένα από το κατώτερο επίπεδο protocol, τα οποία πρέπει να γραφτούν στον ίδιο buffer. Χρησιμοποιούνται spin_locks και όχι σημαφόροι, επειδή σε περίπτωση interrupt context δεν υπάρχει κάποια διεργασία που "επιβλέπει" και ανιχνεύει αυτήν την κατάσταση ώστε να ξυπνήσει την εφαρμογή όταν αυτή είναι έτοιμη.
  Επισημαίνεται ότι χρησιμοποιούνται οι λειτουργίες spin_lock_irqsave, spin_unlock_irqrestore αντί των spin_lock και spin_unlock για να αντιμετωπιστεί περίπτωση deadlock αν κάποια διεργασία, η οποία έχει το lock, δεχτεί κάποιο interrupt και δεν το έχει αφήσει (επομένως απενεργοποιούμε τις διακοπές μέχρι να αφήσει το lock). Αφού η διεργασία λάβει τα δεδομένα από τον buffer του αισθητήρα, ανάλογα με την τιμή του flag επεξεργάζεται τα νέα δεδομένα ή επιστρέφει κατάλληλη τιμή (-EAGAIN αν δεν υπάρχουν νέα δεδομένα, -ERESTARTSYS αν η διεργασία έλαβε κάποιο σήμα ή έγινε κάποιο interrupt και θέλουμε να ξαναεκτελέσουμε το system call). Σχετικά με την επεξεργασία των νέων δεδομένων, μέσω της μεταβλητής state -> type, λαμβάνουμε την κατάλληλη τιμή από τον αντίστοιχο lookup_table και σχηματίζουμε το κατάλληλο float αριθμό, ο οποίος αποθηκεύεται στον buf_data[] με τη χρήση της snprintf και ενημερώνεται κατάλληλα η μεταβλητή buf_lim για το μέγεθος της νέας μέτρησης και η buf_timestamp για τη χρονική στιγμή της ενημέρωσης.

- *lunix_chrdev_open*:
  Αυτή είναι η πρώτη συνάρτηση που καλείται όταν ανοίγει ένα αρχείο της συσκευής και αναλαμβάνει τις αρχικοποιήσεις της δομής lunix_chrdev_state_struct η οποία δείχνει την τρέχουσα κατάσταση της συσκευης για κάθε φορά που ανοίγουμε ένα αρχείο. Συγκεκριμένα, αφού γίνει allocate ο κατάλληλος χώρος, αρχικοποιείται ο σημαφόρος, βρίσκουμε τον σωστό minor number της συσκευης και τον αποθηκεύουμε στη δομή, και εντοπίζεται ο κατάλληλος δείκτης στη θέση του πίνακα lunix_sensors έτσι ώστε να συνδεθεί η δομή αυτή με τον κατάλληλο sensor από τον οποίο θα λαμβάνει δεδομένα.

- *lunix_chrdev_release*:
  Η συνάρτηση αυτή καλείται στην περίπτωση που κλείσει και η τελευταία διεργασία που αναφέρεται στη δομή (σε περίπτωση που την μοιράζονται περισσότερες από μία) ώστε να απελευθερωθεί ο χώρος που δεσμεύθηκε στην open (για την δομη lunix_chrdev_state_struct), εφόσον δεν χρειάζεται πλεον.

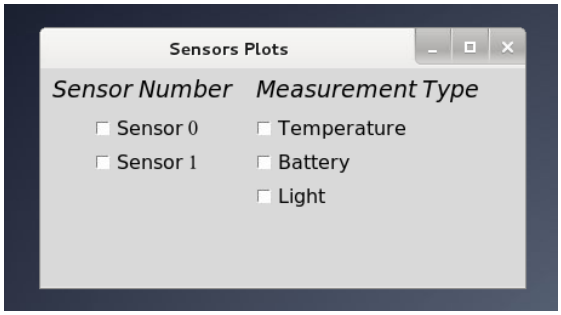
- *lunix_chrdev_read*:
  Η lunix_chrdev_read καλείται κάθε φορά που ανοίγει ένα ειδικό αρχείο του συγκεκριμένου οδηγού συσκευής. Επιστρέφει το πλήθος των bytes, τα οποία διάβασε (αν δεν υπήρξε κάποιο σφάλμα

στην ανάγνωση). Οι σημαφόροι χρησιμοποιούνται για την αντιμετώπιση races, τα οποία μπορεί να υπάρξουν σε περίπτωση μιας διεργασίας που έχει κάνει open κάποιο αρχείο και έπειτα fork ή αν είναι πολυνηματική. Στην περίπτωση που έχουν ήδη διαβαστεί τα δεδομένα του buffer (*f_pos == 0) , αν δεν υπάρχουν νέα δεδομένα, η διεργασία απελευθερώνει το σημαφόρο και αλλάζει την κατάσταση της σε waiting.
Όταν ο αισθητήρας λάβει νέα δεδομένα θα ειδοποιήσει όσες διεργασίες περιμένουν για αυτά (στο αρχείο lunix_sensors.c με την εντολη wake_up_interruptible). Αν δεν υπάρχουν δεδομένα προς ανάγνωση (EOF) επαναφέρουμε το δείκτη ανάγνωσης *f_pos. Σε περίπτωση που ζητηθούν περισσότερα bytes προς ανάγνωση από όσα είναι διαθέσιμα η read διαβάζει μόνο όσα μπορεί. Η μετάβαση των δεδομένων από τον buffer στο userspace γίνεται, με τη χρήση του copy_to_user, ενώ η ανάγνωση γίνεται από το σημείο στο οποίο έμεινε ο δείκτης ανάγνωσης την τελευταία φορά. Επομένως αν δεν διαβάστηκε ολόκληρη η μέτρηση την πρώτη φορά, την επόμενη φορά θα διαβαστούν bytes από την ίδια μέτρηση, ακόμα και αν υπάρχουν νέα δεδομένα προς ανάγνωση (δηλαδή ενημέρωση του buffer της state struct γίνεται μόνο όταν διαβάζο-νται όλα τα bytes της μέτρησης).

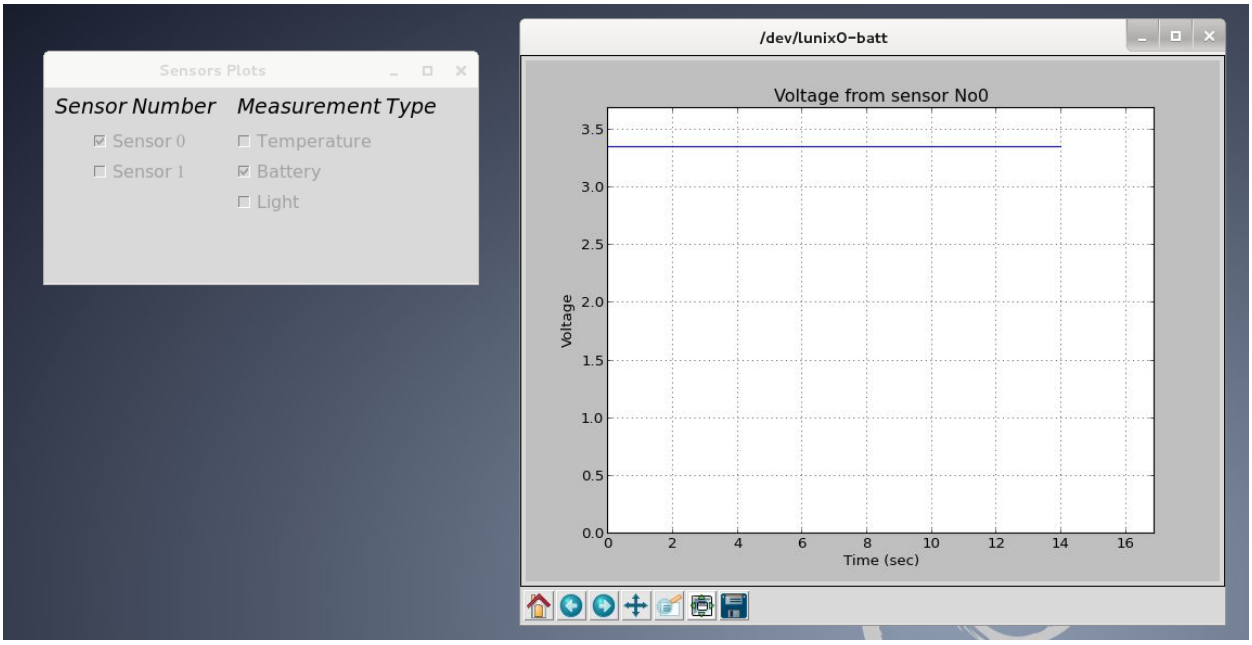
- *lunix_chrdev_init*:
  Η συνάρτηση αυτή εκτελεί τις βασικές λειτουργίες που είναι απαραίτητες για την εγγραφή της συσκευής και ουσιαστικά τη σύνδεση της με τον lunix driver. Με την MKDEV παίρνουμε τον major και minor number στην κατάλληλη μορφή, ώστε να το στείλουμε στη συνάρτηση register_chrdev_region για να δεσμεύσουμε τους συγκεκριμένους αριθμούς (ο minor είναι συνηθως 0, και ουσιαστικά ο πυρήνας ενδιαφέρεται για τον major μόνο). Αφού δεσμευθεί ο αντίστοιχος χώρος από major numbers για τη συσκευη μας, με την cdev_add ορίζουμε ότι αυτοί οι αριθμοί, θα συνδέονται με τον driver που ορίζουμε. Η cdev_init που καλείται στην αρχή, αρχικοποιεί μια δομή cdev για τη συσκευή και τη συνδέει με τη δομή file_operations.

2 Υλοποίηση UI

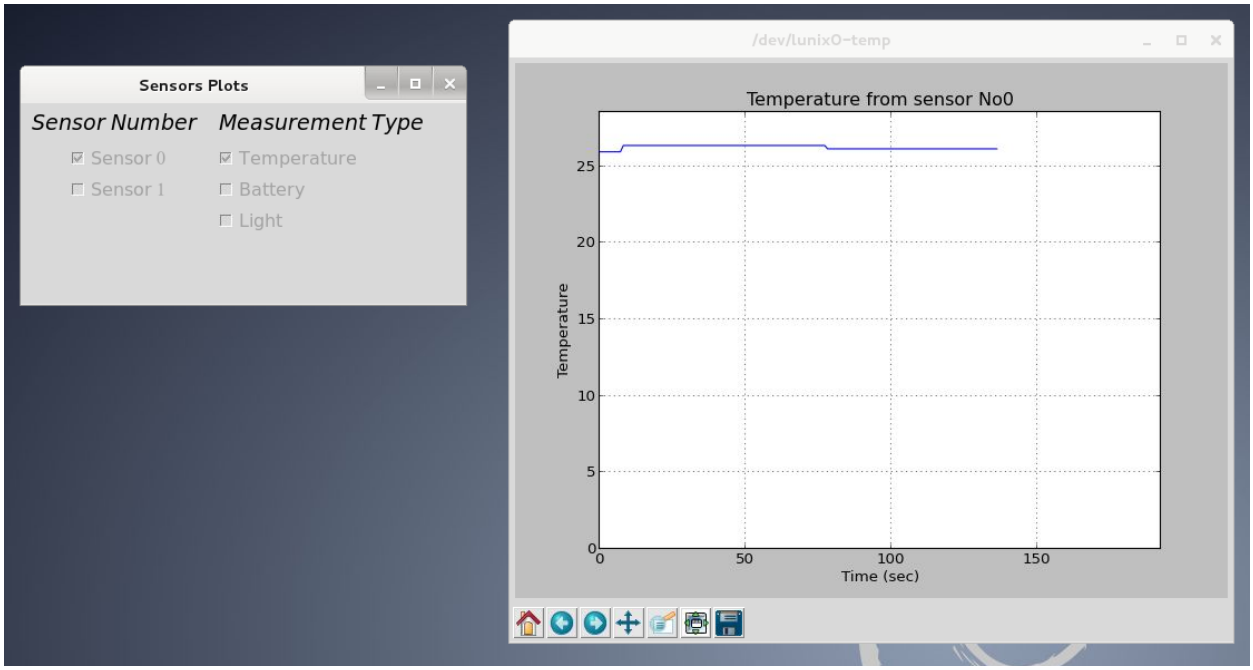
Για τις ανάγκες της συγκεκριμένης άσκησης υλοποιήθηκε user interface σε python. Στο ui αυτό, ο χρήστης μπορεί να επιλέγει από ένα menu με checkbuttons (εικόνα 1) τον αισθητήρα και το είδος της μέτρησης που θέλει να δει, και θα εμφανίζεται μια γραφική παράσταση (μέτρηση σε συνάρτηση με το χρόνο) που θα ανανεώνεται δυναμικά καθώς έρχονται νέες μετρήσεις. Υπάρχει η δυνατότητα προβολής *μίας* γραφικής παράστασης κάθε φορά, και για να επιλεγεί άλλος αισθητήρας ή είδος μέτρησης θα πρέπει πρώτα να κλείσει το παράθυρο της προηγούμενης μέτρησης που προβάλλεται. Ο άξονας y αναπαριστά το μετρούμενο μέγεθος ενώ ο άξονας x αναπαριστά τον χρόνο σε sec, και αυξάνεται σε περίπτωση που αφήσουμε τη μέτρηση να τρέχει για πολύ χρόνο (εικόνα 3). Οι μετρήσεις λαμβάνονται με ταχύτητα 1/μέτρηση/sec.
Ο πλήρης κώδικας φαίνεται στο παράρτημα 3.2.



Εικόνα 1: Το menu του ui



Εικόνα 2: Μέτρηση τάσης από τον αισθητήρα 0



Εικόνα 3: Μέτρηση για μεγάλο χρονικο διαστημα

3 Παράρτημα: Κώδικας

3.1 lunix-chrdev.c

```
/*
 * lunix-chrdev.c
 *
 * Implementation of character devices
 * for Lunix:TNG
 *
 * Chrysoula Varia
 * Evangelia Gergatsouli
 *
 */

#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/list.h>
#include <linux/cdev.h>
#include <linux/poll.h>
#include <linux/slab.h>
#include <linux/sched.h>
#include <linux/ioctl.h>
#include <linux/types.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/mmzone.h>
#include <linux/vmalloc.h>
#include <linux/spinlock.h>

#include "lunix.h"
#include "lunix-chrdev.h"
#include "lunix-lookup.h"

/*
 * Global data
 */
struct cdev lunix_chrdev_cdev;

/*
 * Just a quick [unlocked] check to see if the cached
 * chrdev state needs to be updated from sensor measurements.
 */
static int lunix_chrdev_state_needs_refresh(struct
   lunix_chrdev_state_struct *state)
{
        struct lunix_sensor_struct *sensor;
```

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

        if((sensor->msr_data[state->type]->last_update) != (state->
           buf_timestamp))
                return 1;
        else
                /* The following return is bogus, just for the stub
                   to compile */
                return 0; /* ? */
}

/*
 * Updates the cached state of a character device
 * based on sensor data. Must be called with the
 * character device state lock held.
 */
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct
    *state)
{
        struct lunix_sensor_struct *sensor;
        uint16_t data;
        uint32_t time;
        long result=0, divi_t, mod_t;
        int flag;
        unsigned char sign;
        int ret;
        unsigned long flagz;
        WARN_ON(!(sensor = state->sensor));

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

        spin_lock_irqsave(&sensor->lock, flagz);
                flag=lunix_chrdev_state_needs_refresh(state);
                if(flag){
                        data = sensor->msr_data[state->type]->values
                           [0];
                        time = sensor->msr_data[state->type]->
                           last_update;
                }
        spin_unlock_irqrestore(&sensor->lock, flagz);

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

```
         * (we already have it, from read function)
         */

        /* flag =1 means i got new data else i didnt*/
        if(flag==1){
                /*find raw data according to type of measurement*/
                switch(state->type){
                case 0:
                        result = lookup_voltage[data];
                        break;
                case 1:
                        result = lookup_temperature[data];
                        break;
                case 2:
                        result = lookup_light[data];
                        break;
                case 3:
                        flag = 2;
                        break;
                }

                /*define the sign of number*/
                if((int)result>=0)
                        sign=' ';
                else
                        sign='-';

                divi_t = result / 1000;
                mod_t = result % 1000;
                snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%c%d.%
                   d\n", sign, divi_t , mod_t);
                state->buf_data[LUNIX_CHRDEV_BUFSZ-1]='\0';

                /*find buf limit /size of measurement*/
                state->buf_lim = strnlen(state->buf_data,
                   LUNIX_CHRDEV_BUFSZ);

                state->buf_timestamp = time;
                ret = 0;
                goto update_out;
        }
        else if (flag==0){
                ret = -EAGAIN;
                goto update_out;
        }
        else{
                ret = -ERESTARTSYS;
                goto update_out;
```

```
        }

update_out:
        return ret;
}
/*************************************
 * Implementation of file operations
 * for the Lunix character device
 *************************************/

static int lunix_chrdev_open (struct inode *inode, struct file *filp)
{
        /* Declarations */
        int ret;
        unsigned int minor;
        int tmp;
        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=iminor(inode);

        /* Allocate a new Lunix character device private state
           structure
         * and initialize it
         */
        struct lunix_chrdev_state_struct *point = kmalloc(sizeof(
           struct lunix_chrdev_state_struct), GFP_KERNEL);
        if(point == 0)
                debug("Insufficient memory for state_struct in
                   chrdev_open()\n");

        filp->private_data=point;
        point->type=minor & 0x7;          /*minor=type of measurement
           (0=BATT, 1=TEMP, 2=LIGHT)*/
        point->sensor = &lunix_sensors[minor/8];
        sema_init(&point->lock, 1);
out:
        return ret;
}

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

```
{
        kfree(filp->private_data);
        debug("Device successfully closed\n");
        return 0;
}

static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd,
   unsigned long arg)
{
        return -EINVAL;
}

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 */
                        up(&state->lock); /* release the lock */

                        /*wait in wq queue until condition is true*/
                        if (wait_event_interruptible(sensor->wq, (
                           lunix_chrdev_state_needs_refresh(state) !=
                           0)))
                                return -ERESTARTSYS; /* signal: tell
                                   the fs layer to handle it */
```

```
                        /* otherwise loop, but first reacquire the
                           lock */
                        if (down_interruptible(&state->lock))
                                return -ERESTARTSYS;
                }
        }

        /* End of file */
        if(state->buf_lim == 0){
                ret = 0;
                goto unlock_out;
        }

        /* Determine the number of cached bytes to copy to userspace
           */
        if(cnt > (state->buf_lim - *f_pos)){
                cnt = (state->buf_lim )- (*f_pos);
        }

        /* Auto-rewind on EOF mode? */
        if(copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)){
                ret = -EFAULT;
                goto unlock_out;
        }
        *f_pos += cnt;
        ret = cnt;

        if(*f_pos >= state->buf_lim){
                *f_pos=0;
        }
unlock_out:
        /* Unlock */
        up(&state->lock);
        return ret;
}

static int lunix_chrdev_mmap(struct file *filp, struct vm_area_struct
    *vma)
{
        return -EINVAL;
}

static struct file_operations lunix_chrdev_fops =
{
        .owner          = THIS_MODULE,
        .open           = lunix_chrdev_open,
        .release        = lunix_chrdev_release,
        .read           = lunix_chrdev_read,
```

```
        .unlocked_ioctl = lunix_chrdev_ioctl,
        .mmap           = lunix_chrdev_mmap
};

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, range
           =16*3;

        cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
        lunix_chrdev_cdev.owner = THIS_MODULE;
        lunix_chrdev_cdev.ops = &lunix_chrdev_fops;

        dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);

        /* register_chrdev_region? */
        ret = register_chrdev_region(dev_no, range, "Lunix");
        if (ret < 0) {
                debug("failed to register region, ret = %d\n", ret);
                goto out;
        }

        /*Add the device to the system*/
        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;
}

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");
}
```

3.2 ui.py

```
#!/usr/bin/env python

from Tkinter import *
#For plot...
import os
import numpy as np
import datetime
from matplotlib import pyplot as plt
from matplotlib import animation
#...for plot

top = Tk()
#Fix title + window size
top.wm_title("Sensors␣Plots")
width=400
height=180
top.geometry('{}x{}'.format(width, height))

global fName, sSelect, mSelect
sSelect = 0
mSelect = 0
fName=""

#########--PLOT BEGIN--#############
def init():
        global line
        line.set_data([], [])
        return line,

def animate(i):
        global fName
        global xdata, ydata, ax
        fp=open(fName)
        tmpY = fp.readline()
        fp.close()
        y=float(tmpY)
        # Collect data into x and y lists
```

```
        xdata.append(i)
        ydata.append(y)
        xmin, xmax = ax.get_xlim()
        ymin, ymax = ax.get_ylim()

        #changing the xmax dynamically
        if i >= xmax:
            ax.set_xlim(xmin, xmax+(xmax/2))
            ax.figure.canvas.draw()

        #changing the ymax dynamically
        if y >= ymax:
            ax.set_ylim(ymin, y+(y/10))
            ax.figure.canvas.draw()

        line.set_data(xdata, ydata)
        return line,

#When closed plot, allow user to choose new type of plot
def handle_close(evt):
        global c0, c1, m0, m1, m2, fName, mSelect, sSelect
        c0.deselect()
        c1.deselect()
        m0.deselect()
        m1.deselect()
        m2.deselect()
        c0.config(state='normal')
        c1.config(state='normal')
        m0.config(state='normal')
        m1.config(state='normal')
        m2.config(state='normal')
        fName=""
        mSelect = 0
        sSelect = 0

#Disable buttons when user has chosen sensor and measurement
def disable_all():
        global c0, c1, m0, m1, m2
        c0.config(state='disable')
        c1.config(state='disable')
        m0.config(state='disable')
        m1.config(state='disable')
        m2.config(state='disable')

def do_plot():
        #initial max x axis
        init_xlim = 5
        init_ylim = 3
```

```
        global xdata, ydata, ax, fName
        fig = plt.figure(fName)
        fig.canvas.mpl_connect('close_event', handle_close)
        ax = plt.axes(xlim=(0, init_xlim), ylim=(0, init_ylim))

        #Create appropriate Labels
        if "temp" in fName:
                yLabel="Temperature"
        elif "batt" in fName:
                yLabel="Voltage"
        else:
                yLabel="Light"
        plt.ylabel(yLabel)
        plt.xlabel("Time (sec)")
        #And title of plot
        if "0" in fName:
                sensorNo="0"
        else:
                sensorNo="1"
        plt.title(yLabel+" from sensor No"+sensorNo)
        ax.grid()

        #Disable buttons until plot is closed
        disable_all()

        global line
        line, = ax.plot([], [], lw=1)
        xdata, ydata = [], []

        anim = animation.FuncAnimation(fig, animate, init_func=init,
           frames=200, interval=1000, blit=False)
        plt.show()
##########--PLOT END--##############

#Sensor CheckButtons functions
def sensor0Active():
        if (sensor0.get()):
                global fName, sSelect
                if sSelect == 0: #first sensor selection
                        sSelect =1
                        if fName=="":
                                fName="/dev/lunix0"
                        else:
                                fName="/dev/lunix0"+fName
                                do_plot()
                else:           #update existing sensor selection
                        fName="/dev/lunix0"
```

```
def sensor1Active():
        if (sensor1.get()):
                global fName, sSelect
                if sSelect == 0: #first selection
                        sSelect = 1
                        if fName=="":
                                fName="/dev/lunix1"
                        else:
                                fName="/dev/lunix1"+fName
                                do_plot()
                else:             #update existing one
                        fName="/dev/lunix1"

#Measurement type CheckButtons functions
def MeasTemp():
        if (Temp.get()):
                global fName, mSelect
                if mSelect == 0:
                        mSelect=1
                        if fName=="":
                                fName="-temp"
                        else:
                                fName=fName+"-temp"
                                do_plot()
                else:#update existing one
                        fName="-temp"
def MeasBatt():
        if (Batt.get()):
                global fName, mSelect
                if mSelect == 0:
                        mSelect=1
                        if fName=="":
                                fName="-batt"
                        else:
                                fName=fName+"-batt"
                                do_plot()
                else:#update existing one
                        fName="-batt"
def MeasLight():
        if (Light.get()):
                global fName, mSelect
                if mSelect == 0:
                        mSelect=1
                        if fName=="":
                                fName="-light"
                        else:
                                fName=fName+"-light"
                                do_plot()
```

```
                else:#update existing one
                        fName="-light"
#CheckButtons for Sensors
sensor0 = IntVar()
sensor1 = IntVar()

c0 = Checkbutton(top, text="Sensor 0", variable=sensor0, command=
    sensor0Active, font=('Symbol', 12))
c0.grid(row=11, column=0, columnspan=2, rowspan=1)
c1 = Checkbutton(top, text="Sensor 1", variable=sensor1, command=
    sensor1Active, font=('Symbol', 12))
c1.grid(row=12, column=0, columnspan=2, rowspan=1)


#Checkbuttons for measurement type
Temp = IntVar()
Batt = IntVar()
Light = IntVar()

w = Message(top, text="Sensor Number", width = width/2, font=('Symbol
    ', 14, 'italic')).grid(row=10, column=0, rowspan=1, columnspan=2)
w = Message(top, text="Measurement Type", width = width/2, font=('
    Symbol', 14, 'italic')).grid(row=10, column=3, rowspan=1,
    columnspan=2)

m0 = Checkbutton(top, text="Temperature", variable=Temp, command=
    MeasTemp, font=('Symbol', 12))
m0.grid(row=11, column=3, columnspan=2, rowspan=1, sticky=W)
m1 = Checkbutton(top, text="Battery", variable=Batt, command=MeasBatt
    , font=('Symbol', 12))
m1.grid(row=12, column=3, columnspan=2, rowspan=1, sticky=W)
m2 = Checkbutton(top, text="Light", variable=Light, command=MeasLight
    , font=('Symbol', 12))
m2.grid(row=13, column=3,columnspan=2, rowspan=1, sticky=W)

top.mainloop()
```