ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΛΕΙΤΟΥΡΓΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ – 7ο ΕΞΑΜΗΝΟ

Δημητρίου Αγγελική – ΑΜ: 03117106

Τζομάκα Αφροδίτη – ΑΜ: 03117107

Ομάδα 9

2η Εργαστηριακή Άσκηση

ΕΙΣΑΓΩΓΗ – ΣΤΟΧΟΣ ΤΗΣ ΑΣΚΗΣΗΣ:

Στην εργαστηριακή άσκηση αυτή θα υλοποιηθεί οδηγός συσκευής για ένα ασύρματο δίκτυο αισθητήρων κάτω από το λειτουργικό σύστημα Linux. Το συγκεκριμένο δίκτυο διαθέτει έναν αριθμό από αισθητήρες και ένα σταθμό βάσης ο οποίος συνδέεται μέσω USB με υπολογιστικό σύστημα Linux στο οποίο και θα εκτελείται ο ζητούμενος οδηγός συσκευής - Lunix:TNG. Οι αισθητήρες αναφέρουν περιοδικά το αποτέλεσμα τριών διαφορετικών μετρήσεων: της τάσης της μπαταρίας που τους τροφοδοτεί, της θερμοκρασίας και της φωτεινότητας του χώρου όπου βρίσκονται. Ο σταθμός βάσης λαμβάνει πακέτα με δεδομένα μετρήσεων, τα οποία προωθεί μέσω διασύνδεσης USB στο υπολογιστικό σύστημα. Η διασύνδεση υλοποιείται με κύκλωμα Serial over USB, για το οποίο ήδη ο πυρήνας διαθέτει ενσωματωμένους οδηγούς, οπότε τα δεδομένα όλων των μετρήσεων όλων των αισθητήρων εμφανίζονται σε μία εικονική σειριακή θύρα, /dev/ttyUSB1.

Για την προσομοίωση της παραπάνω λειτουργίας για τον καθένα μας, σε μηχάνημα της σχολής εγκαταστάθηκε εξυπηρετητής TCP/IP, ο οποίος εκπέμπει συνεχώς μετρήσεις. Ο εξυπηρετητής αυτός είναι προσβάσιμος μέσω της θύρας TCP 49152. Το utopia.sh, ανακατευθύνει αυτόματα την πρώτη σειριακή θύρα του εικονικού μηχανήματος QEMU-KVM (/dev/ttyS0) στη θύρα TCP 49152. Οπότε στο δικό μας εικονικό μηχάνημα οι μετρήσεις εμφανίζονταν στην σειριακή θύρα/dev/ttyS0, χωρίς να υπάρχει καμία πρακτική διαφορά.

Ουσιαστικά από εμάς ζητείται η κατασκευή του οδηγού συσκευής χαρακτήρων (character device driver), ο οποίος θα λαμβάνει τα δεδομένα των μετρήσεων από το δίκτυο αισθητήρων και θα τα εξάγει στο χώρο χρήστη σε διαφορετικές συσκευές, ανάλογα με το είδος της μέτρησης και τον αισθητήρα απ' όπου προέρχεται. Το αρχείο lunix_dev_nodes.sh υλοποιεί τη δημιουργία των 16(αισθητήρες)*3(μετρήσεις) = 48 συσκευών που χρειαζόμαστε (π.χ., για τον πρώτο αισθητήρα οι /dev/lunix0-batt, /dev/lunix0-temp και /dev/lunix0-light).

ΠΕΡΙΓΡΑΦΗ ΤΩΝ ΑΝΩΤΕΡΩΝ ΕΠΙΠΕΔΩΝ

Ξεκινάμε εισάγοντας το module του LunixTNG στον πυρήνα μέσω της κλήσης της insmod και εκτελώντας την εντολή dmesg παρατηρούμε ότι καλείται η συνάρτηση lunix_module_init( ) του αρχείου lunix-module.c. Η ίδια αρχικοποιεί ορισμένες δομές δεδομένων όπως η lunix_sensor_struct, για την οποία δεσμεύει ένα πίνακα *lunix_sensors 16 τέτοιων δομών (όσοι και οι αισθητήρες) για γράψιμο και διάβασμα των κατάλληλων δεδομένων από όλα τα επιμέρους αρχεία. Στη συνέχεια καλούνται οι lunix_protocol_init( ), lunix_sensor_init( ), lunix_ldisc_init( ), lunix_chrdev_init( ) οι οποίες εκτελούν αρχικοποιήσεις σχετικές με τις δομές που περιλαμβάνουν.

Επόμενο βήμα είναι η δημιουργία των συσκευών μέσω του lunix_dev_nodes.sh και προσάρτηση της θύρας ttyS0 μέσω του lunix-attach. Ξανατρέχουμε dmesg και αυτήν την φορά βλέπουμε την κλήση της συνάρτησης lunix_ldisc_receive( ) που βρίσκεται αντίστοιχα στο αρχείο lunix-ldisc.c. Η ίδια στο σώμα της καλεί την lunix_protocol_received_buff και εκείνη με τη σειρά της την lunix_protocol_update_sensors με σκοπό την πακετοποίηση των raw δεδομένων σύμφωνα με τις συμβάσεις του πρωτοκόλλου και την ανανέωση με αυτά των δομών των αισθητήρων. Για το λόγο αυτό καλείται η lunix_sensor_update και τέλος η lunix_chrdev_update διαβάζει τα σωστά δεδομένα από το πεδίο value των μετρήσεων των αισθητήρων.

ΥΛΟΠΟΙΗΣΗ ΣΥΣΚΕΥΗΣ ΧΑΡΑΚΤΗΡΩΝ LUNIX – TNG

Παρακάτω παρουσιάζονται οι συναρτήσεις του αρχείου lunix-chrdev.c οι οποίες απαιτούσαν την δική μας προσθήκη κώδικα για την ορθή λειτουργία της συσκευής. Σκοπός ουσιαστικά είναι η σωστή εξαγωγή των δεδομένων από τους buffers των αισθητήρων προς την εφαρμογή του χρήστη φροντίζοντας παράλληλα τόσο για την ταυτόχρονη πρόσβαση από πολλές διεργασίες όσο και για τον περιορισμό των δικαιωμάτων ανά αρχείο, ανά αισθητήρα, ανά μέτρηση.

Σημειώνεται πώς τα σχόλια που ξεκινάνε με «//» είναι δική μας προσθήκη και ξεχωρίζουν από τα δοθέντα που ξεκινάνε με «/*». Επεξηγούν τη λειτουργία και το σκοπό κάθε επιμέρους εντολής που χρησιμοποιήθηκε.

- lunix_chrdev_init( ): Η συνάρτηση αυτή χρησιμοποιήθηκε για την εγγραφή και την προσθήκη κάθε συσκευής στον οδηγό μέσω των συναρτήσεων register_chrdev_region και cdev_add αντίστοιχα. Η πρώτη καταχωρεί τη συσκευή δεσμεύοντας έναν major number, μια σειρά minor numbers και ένα όνομα για αυτήν. Η δεύτερη προσθέτει στο σύστημα και θέτει σε λειτουργία τη συσκευή. Προφανώς υπάρχει και η αντίστοιχη συνάρτηση destroy που περιέχει τα cdev_delete και uregister_chrdev_region απλά δεν χρειάστηκε κάποια προσθήκη από εμάς.

```
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? */
    //Register a range of device numbers
    //dev_no: the first in the desired range of device numbers; must include the major number
    //lunix minor cnt: the number of consecutive device numbers required
    //"lunix": the name of the device or driver
    //Return value is zero on success, a negative error code on failure
    ret = register_chrdev_region(dev_no,lunix_minor_cnt,"lunix");
    if (ret < 0)
    {
        debug("failed to register region, ret = %d\n", ret);
        goto out;
    }
    /* ? */
    /* cdev add? */
    //Add a char device to the system
    //lunix_chrdev_cdev: the cdev structure for the device
    //dev_no: the first device number for which this device is responsible
    //lunix_minor_cnt: the number of consecutive minor numbers corresponding to this device
    //Adds the device represented by lunix chrdev cdev to the system, making it live
immediately. A negative error code is returned on failure.
    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;
}
```

- lunix_chrdev_open( ): Η open είναι η πρώτη ρουτίνα που υλοποιήσαμε και στην οποία δείχνει το struct file_operations. Γενικά, ελέγχει για τυχόν σφάλματα που έχουν να κάνουν με το εκάστοτε αρχείο της συσκευής, αρχικοποιεί όλα τα απαραίτητα πεδία και δεσμεύει την κατάλληλη μνήμη. Σημειώνεται ότι στο πεδίο private_data του ανοιχτού αποθηκεύεται όλη η πληροφορία για την αρχική κατάσταση της συσκευής χαρακτήρων η οποία μετέπειτα θα περάσει και στα διάφορα system calls, κάτι το οποίο περιγράφεται εκτενώς στα σχόλια.

```
static int lunix chrdev open(struct inode *inode, struct file *filp)
{
    /* Declarations */
    /* ? */
    int ret;
    int minor num;
    struct lunix chrdev state struct *prst;
```

```
    debug("entering\n");
    ret = -ENODEV;                                  //Initialize with no such device
    if ((ret = nonseekable open(inode, filp)) < 0) //Nonseekable open clears the bits that
specify the access mode of the open file
//This call marks the given filp as being nonseekable //the kernel never allows an lseek call
on such a file to succeed. By marking the file in //this way, you can also be assured that no
attempts will be made to seek //the file by way of the pread and pwrite system calls
        goto out;           //if that fails no device was found, cannot open, leave.

    /*
     * Associate this open file with the relevant sensor based on
     * the minor number of the device node [/dev/sensor<NO>-<TYPE>]
     */
    minor_num = iminor(inode); //use macro iminor to get inodes minor number

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

    prst = kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL); //GFP_KERNEL -> both
background and direct reclaim are allowed
//and the default page allocator behavior is used
//underlines that memory is allocated on behalf of user and may sleep
    if (!prst)
    {
        ret = -ENOMEM;
        goto out;
    }

    prst->type = minor_num & 7; //Minor number = 8*sensor_num + msr_type
                                //so mask 3 LSBs to find the msr type

    prst->sensor = &lunix_sensors[minor_num >> 3]; //The bits in front indicate the number of
the sensor

    prst->buf_timestamp = 0; //At the initialisation of the device time = 0

    sema init(&prst->lock, 1); //Semaphore value = 1 in order to acquire access for the first
proccess
//1 is the initial value to assign to a semaphore

    filp->private_data = prst; //private_data is set to null by open sys_call
                               //Info about the state of the chrdev is stored there
                               //Preserve data across sys_calls
    debug("Data allocated\n");
out:
    debug("leaving, with ret = %d\n", ret);
    return ret;
}
```

- lunix_chrdev_release( ): Η συνάρτηση αυτή απελευθερώνει τον δείκτη private_data κατά το κλείσιμο του αρχείου καθώς δεν θα είναι πλέον open file και δεν θα θέλαμε ξεκρέμαστο pointer.

```
static int lunix chrdev release(struct inode *inode, struct file *filp)
{
    /* ? */
    kfree(filp->private_data); //Free the private_data pointer
    return 0;
}
```

- lunix_chrdev_ioctl( ): Η προαιρετική αυτή συνάρτηση υλοποιεί λειτουργίες input/output που δεν υποστηρίζονται από τις κλασσικές κλήσεις συστήματος. Μιας και δεν υλοποιήθηκε, οφείλει να επιστρέφει invalid argument. Το ίδιο ισχύει και για την lunix_chrdev_mmap( ).

```
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
   /* Why? */
   //If the device doesn't provide an ioctl method
   //the system call returns an  error  for  any  request  that  isn't  predefined
   return -EINVAL; //invalid argument
}
```

- lunix_chrdev_read( ): Εδώ υλοποιείται η επιστροφή των δεδομένων στον χρήστη. Αρχικά ανακτούνται οι πληροφορίες τόσο για την παρούσα κατάσταση μέσω του πεδίου private_data του δοθέντος ανοικτού αρχείου, όσο και για τον αντίστοιχο αισθητήρα. Στη συνέχεια αφού εξασφαλιστεί ότι μόνο μία διεργασία εκ του συνόλου των διεργασιών που μοιράζονται το ίδιο struct file (πχ πατέρας – παιδί) θα αποκτήσει πρόσβαση στα δεδομένα, δηλαδή θα αποκτήσει τον σημαφόρο, ελέγχεται η θέση από την οποία θα γίνει το read. Αν βρισκόμαστε στην αρχή θα πρέπει να κάνουμε τους απαιτούμενους ελέγχους για την ύπαρξη ή όχι καινούριων δεδομένων φροντίζοντας τη συμπεριφορά των διεργασιών στην δεύτερη περίπτωση (απελευθέρωση σημαφόρου, εξασφάλιση blocking λειτουργίας, διακοπτόμενος ύπνος, ανάκτηση σημαφόρου). Σημειώνεται ότι όσο κοιμούνται οι διεργασίες με την wait_event_interruptible, μέσω της wake_up_interruptible της lunix_sensor_update( ) ξυπνάνε όλες όσες βρίσκονται στο waiting queue και ελέγχουν την συνθήκη ανανέωσης δεδομένων. Αν ισχύει μόνο η σωστή διεργασία θα λάβει εν τέλει τον σημαφόρο μέσω της down_interruptible. Αν βρισκόμαστε στο τέλος τότε επιστρέφουμε το 0 ενώ διαφορετικά, ξεκινάμε τη διαδικασία του διαβάσματος. Πιο συγκεκριμένα, καθορίζεται ο αριθμός δεδομένων που θα διαβαστούν, τα δεδομένα αυτά μεταφέρονται στους buffers του χρήστη και ανανεώνεται κατάλληλα το offset μέσα στον buffer δεδομένων της κατάστασης που διαβάζουμε. Τιμή επιστροφής τίθεται ο αριθμός των δεδομένων που διαβάστηκαν και αν φτάσαμε στο τέλος επιστρέφουμε το offset 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;
   WARN_ON(!state);

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

   /* Lock? */
   if (down_interruptible(&state->lock)) //for down_interruptible
       return -ERESTARTSYS;              //If the semaphore is successfully acquired, this
function returns 0.
                                         //If semaphore is not available the respective
process will be put on the semaphore wait-queue
                                         //Task state of the proccess will change to
TASK_INTERRUPTIBLE (sleeping until some condition becomes true, signal or hw interrupt)
                                         //Scheduler will be invoked to run another process
                                         //If the sleep is interrupted by a signal, this
function will return -EINTR, and the syscall will be restarted
```

```
                                        //semaphore is only acquired when process in front of
wq

    /*
     * 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
     */

    //f pos = 0  means we start from the beginning so we have a new measurment
    if (*f_pos == 0)
    {
        while (lunix_chrdev_state_update(state) == -EAGAIN) //EAGAIN = no available data, try
later
        {
            up(&state->lock);               //release semaphore
                                            //it needs to be released because if process sleeps
holding it, no one will be able to wake us up
            if (filp->f_flags & O_NONBLOCK) //if in non-blocking mode, try again
                return -EAGAIN;

            //interruptible sleep until state needs refresh
            //wait event interruptible ensures the proper and expected reaction to signals,
            //which could have been responsible for waking up the process
            if (wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state)))
                return -ERESTARTSYS;

            if (down interruptible(&state->lock)) //try to acquire lock like in the beginning
                return -ERESTARTSYS;
        }
    }

    /* End of file */
    if (*f_pos >= state->buf_lim)
    {
        return 0;
    }

    /* Determine the number of cached bytes to copy to userspace */
    if (cnt > state->buf lim - *f pos)                       //if we need to read more bytes
than available in buffer
        cnt = state->buf_lim - *f_pos;                      //force to read as many bytes as
we can
    if (copy to user(usrbuf, state->buf data + *f pos, cnt)) //write to users buffer cnt amount
of bytes from our current reading point
    {                                                       //which is buf_data offseted by
f_pos
        ret = -EFAULT;
        goto out;
    }

    *f pos += cnt; //update open files offset by cnt
    ret = cnt;     //read function must return num of bytes read

    /* Auto-rewind on EOF mode? */
    if (*f pos >= state->buf lim)
    {
        *f_pos = 0;
    }
out:
    /* Unlock? */
    up(&state->lock);
    return ret;
}
```

- lunix_chrdev_state_needs_refresh( ): Πρόκειται για βοηθητική συνάρτηση ώστε να γνωρίζουμε εάν τα δεδομένα μέτρησης της ιδιωτικής κατάστασης που αποθηκεύτηκε στο πεδίο private_data του ανοιχτού αρχείου πρέπει να ανανεωθούν ή όχι. Επιστρέφει 1 εάν ναι και μηδέν αν όχι. Ελέγχει αν έχει συμβεί ανανέωση των δεδομένων στη δομή του αισθητήρα της κατάστασης σε

μεταγενέστερη χρονική στιγμή από αυτήν που είναι αποθηκευμένη στο πεδίο buf_timestamp. Σημειώνεται πως το πεδίο last_update αλλάζει λόγω καινούριας μέτρησης των αισθητήρων, ενώ το buf_timestamp ανανεώνεται από εμάς. Δεδομένου ότι η συνάρτηση αυτή διαβάζει απευθείας δεδομένα του αισθητήρα, θα πρέπει να καλείται μόνο όταν η διεργασία που την καλεί έχει πάρει το spinlock του αισθητήρα.

```
static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
{
   struct lunix sensor struct *sensor;

   WARN_ON(!(sensor = state->sensor));
   //If the condition above is true (error) WARN() prints the contents of the registers and
the stack Trace.

   //Function returns 1 if state needs refreshing or 0 if not
   //The state needs to be refreshed only when an update to the measurement
   //data has happened later than the timestamp of the given state
   return sensor->msr_data[state->type]->last_update > state->buf_timestamp;
}
```

- lunix_chrdev_state_update( ): Πρόκειται για τη συνάρτηση που ανανεώνει τα πεδία της κατάστασης του οδηγού συσκευής χαρακτήρων όταν αυτό είναι αναγκαίο (νέα διαθέσιμα δεδομένα). Λόγω της λειτουργίας της η update οφείλει να καλείται από διεργασία αναγνώστη μόνο αφού αποκτήσει σημαφόρο. Η ανανέωση των δεδομένων χωρίζεται σε δύο στάδια, την απόκτηση τους από τον αισθητήρα (επεξεργασμένα δεδομένα από την update του lunix protocol) και την αποθήκευση τους στον buffer της ιδιωτικής κατάστασης. Δεδομένου ότι η needs_refresh επιστρέψει 1, η ανάκτηση δεδομένων πρέπει να γίνει γρήγορα κρατώντας το spinlock του αισθητήρα ώστε να μην ενοχληθεί από κάποια διακοπή. Όταν τελειώσει αυτή η διαδικασία γυρνάμε στο ίδιο interrupt state που είχαμε πριν αποκτηθεί το spinlock και πλέον μπορούμε χωρίς βιασύνη να κάνουμε format στα δεδομένα με τον τρόπο που αναλυτικά περιγράφεται στα σχόλια του κώδικα.

```
static int lunix chrdev state update(struct lunix chrdev state struct *state)
{
   struct lunix sensor struct *sensor;
   unsigned long flags;
   uint32_t value;
   long conv msr;
   debug("leaving\n");
   sensor = state->sensor;     //Retrieve states sensor to refer to
   /*
    * Grab the raw data quickly, hold the
    * spinlock for as little as possible.
    */
   spin lock irqsave(&sensor->lock, flags);
   /* Why use spinlocks? See LDD3, p. 119 */
   //We use spinlocks because they are fast. Also there is no need for context switching.
   //Spinlocks are used in code that should not sleep (ie intr_handlers) to avoid deadlocks.
   //spin lock irqsave disables interrupts (on the local processor only) before taking the
spinlock.
   //Because a spinlock can be taken by code that runs in (hardware or software) interrupt
context (sensors),
   //we must use it. The previous interrupt state is stored in flags.

   /*
    * Any new data available?
    */
   if (!lunix chrdev state needs refresh(state))
   {
```

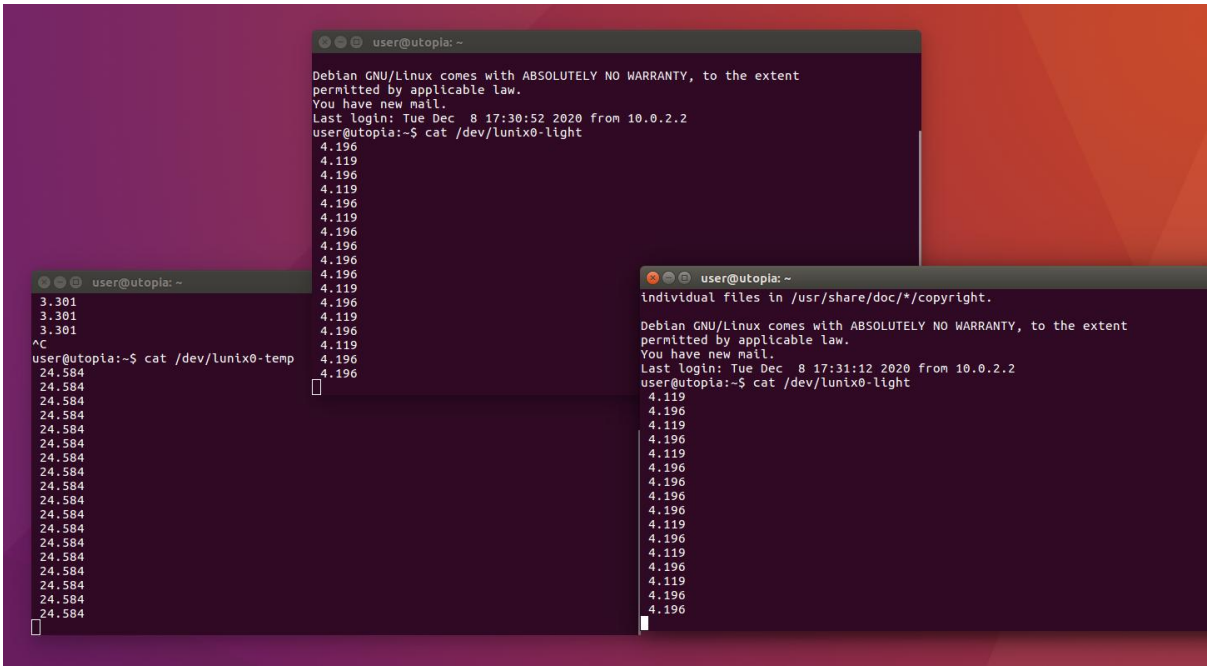
```
        spin_unlock_irqrestore(&sensor->lock, flags);    //If not, must give back the lock and
enable interrupts
        return -EAGAIN;                                  //Return the corresponding value for no
data, try again
    }

    //NOW I GO TO JOB, NOW I GO TO JOB.
    state->buf timestamp = sensor->msr data[state->type]->last update;    //Update the state
timestamp with the time of sensor last update
    value = sensor->msr data[state->type]->values[0];                     //Retrieve value of
sensor (lunix_sensor_update)
    spin_unlock_irqrestore(&sensor->lock, flags);                         //Again, we must give
back the lock and enable interrupts

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

    if (state->type == BATT)
    {
        conv msr = lookup voltage[value];
    }
    else if (state->type == TEMP)
    {
        conv msr = lookup temperature[value];
    }
    else
    {
        conv msr = lookup light[value];
    }

    //Update states date with the given format +-xxyyy(from lookup tables) ---> +-xx,yyy with
div & mod
    state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%c%ld.%03ld\n", conv_msr >=
0 ? ' ' : '-', conv msr / 1000, conv msr % 1000);

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

ΕΛΕΓΧΟΣ ΟΡΘΗΣ ΛΕΙΤΟΥΡΓΙΑΣ

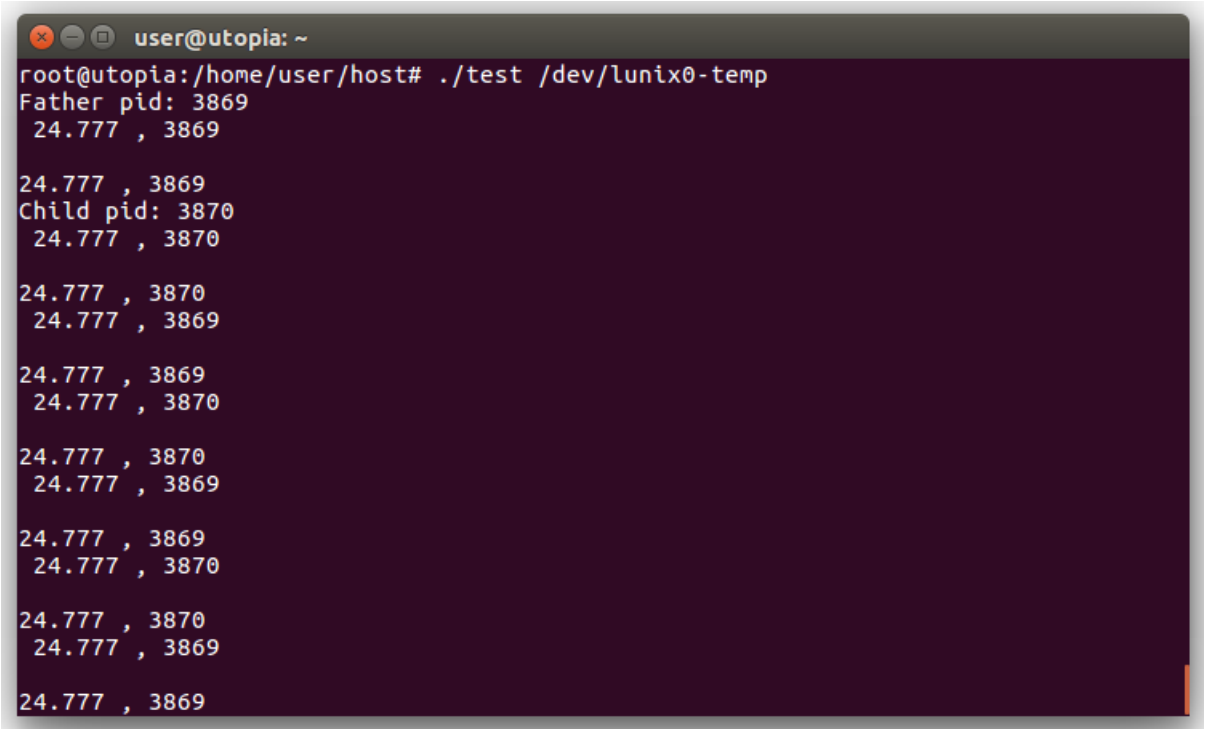
Αρχικά ελέγχθηκε η ορθή λειτουργία του οδηγού με χρήση της εντολής cat από διαφορετικά τερματικά, διαβάζοντας ταυτόχρονα και από ίδιους αισθητήρες και βλέπουμε ότι τα αποτελέσματα είναι τα αναμενόμενα. Δηλαδή κάθε διεργασία έχει πρόσβαση στις μετρήσεις και δεν παρουσιάζονται προβλήματα/παγώματα στην εκτέλεση του προγράμματος.

Για τον περαιτέρω έλεγχο της ορθότητας του οδηγού, φτιάξαμε και ένα userspace program το οποίο ουσιαστικά κάνει fork ένα παιδί και διαβάζει τα δεδομένα των μετρήσεων της συσκευής που δίνεται στο input. Παρατηρούμε πως αν και ο πατέρας και το παιδί έχουν πρόσβαση στο ίδιο fd και προσπαθούν να διαβάσουν και οι δύο από αυτό τα αποτελέσματα και πάλι είναι σωστά. Αναλυτικότερα, καταφέρνουν να διαβάσουν και οι δύο μετρήσεις εναλλάξ χωρίς να μπερδεύονται.

Τα αποτελέσματα αυτά φαίνονται στις εικόνες παρακάτω μαζί με τον κώδικα του userspace προγράμματος.



Αποτελέσματα cat



Αποτελέσματα μέσω του προγράμματος χρήστη

```c
#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 fd, status;
    pid_t pid;
    size_t bytes_read2;
    char buffer2[7];

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

    fd = open(argv[1], O_RDONLY);
    if (fd < 0)
    {
        perror(argv[1]);
        return 1;
    }
    printf("Father pid: %d\n", getpid());
    pid = fork();

    if (pid == 0)
    {
        printf("Child pid: %d\n", getpid());
    }
    do
    {
        bytes_read2 = read(fd, &buffer2, 7);
        if (bytes_read2 < 0)
        {
            perror("read");
            exit(1);
        }

        printf("%s , %d\n", buffer2, getpid());
    } while (bytes_read2 > 0);

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

    wait(&status);

    return 0;
}
```

Κώδικας userspace προγράμματος