ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά 2ης Άσκησης:

Οδηγός Ασύρματου Δικτύου Αισθητήρων στο Λειτουργικό Σύστημα Linux

Ομάδα: cslaba13 :

Δράγαζης Νικόλαος , ΑΜ: 03113162

Πέτρου Γεώργιος , ΑΜ: 03113145

Εισαγωγή

Στην άσκηση αυτή υλοποιούμε έναν character devive driver για ένα δίκτυο αισθητήρων στο Linux. Συγκεκριμένα, υλοποιούμε το ανώτερο στρώμα του οδηγού που αφορά στην εξαγωγή των μετρήσεων προς το χρήστη ως ένα σύνολο από συσκευές χαρακτήρων. Επιπλέον, υλοποιήσαμε τη λειτουργία **ioctl** ώστε να μπορούμε να λαμβάνουμε τα δεδομένα και χωρίς μορφοποίηση.

Υλοποίηση των file operations

Ο κώδικας του ανωτέρου στρώματος του οδηγού μας που υλοποιεί το interface προς το χρήστη ως μία συσκευή χαρακτήρων είναι ο εξής:

```
/*
 * lunix-chrdev.c
 *
 * Implementation of character devices
 * for Lunix:TNG
 *
 * < Your name here >
 *
 */

#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)
{
	debug("entering\n");
	struct lunix_sensor_struct *sensor;

	WARN_ON ( !(sensor = state->sensor));
	/* ? */

	if(state->buf_timestamp < sensor->msr_data[state->type]->last_update){
		debug("leaving\n");
		return 1;
```

```
	}

	/* The following return is bogus, just for the stub to compile */
	debug("leaving\n");
	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;

	debug("entering\n");

	/*
	 * Grab the raw data quickly, hold the
	 * spinlock for as little as possible.
	 */
	/* ? */
	sensor = state->sensor;
	uint32_t new_data;
	uint32_t new_timestamp;
	int decoded;
	unsigned long flags;
	/* Why use spinlocks? See LDD3, p. 119 */
	/*
	 * Any new data available?
	 */
	/* ? */
	if(lunix_chrdev_state_needs_refresh(state)){
		spin_lock_irqsave(&sensor->lock, flags);
		new_data=sensor->msr_data[state->type]->values[0];
		new_timestamp=sensor->msr_data[state->type]->last_update;
		spin_unlock_irqrestore(&sensor->lock, flags);
	}else{
		debug("leaving\n");
		return -EAGAIN;
	}
	/*
	 * Now we can take our time to format them,
	 * holding only the private state semaphore
	 */

	/* ? */
	state->buf_lim = 0;
	if(state->type == TEMP)
		decoded = lookup_temperature[new_data];
	else if(state->type == BATT)
		decoded = lookup_voltage[new_data];
	else
		decoded = lookup_light[new_data];
	if(decoded < 0){
		state->buf_data[0] = '-';
		state->buf_lim++;
	}

	int n;
	if(state -> ioctl_type){
	     int div = decoded / 1000;
	     int mod = decoded % 1000;
	     n = sprintf(state->buf_data, "%d.%d\n", div, mod);
	 }else{
	     n = sprintf(state->buf_data,"0x%02x\n", new_data);
	 }

	state->buf_lim += n;
	state->buf_timestamp = new_timestamp;

	debug("leaving\n");
```

```
	return 0;
}

/************************************
 * Implementation of file operations
 * for the Lunix character device
 ************************************/

static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
	/* Declarations */
	/* ? */
	struct lunix_chrdev_state_struct *state;
	int ret;

	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>]
	 */

	/* Allocate a new Lunix character device private state structure */
	/* ? */
	state = (struct lunix_chrdev_state_struct *)kzalloc(sizeof(struct
lunix_chrdev_state_struct),GFP_KERNEL);
	state->type = iminor(inode) % 8;
	state->sensor = &lunix_sensors[iminor(inode)/8];
	state->buf_lim = 0;
	 state->ioctl_type = 0;
	sema_init(&state->lock, 1);
	filp->private_data = state;
out:
	debug("leaving, with ret = %d\n", ret);
	return ret;
}

static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
	/* ? */
	debug("entering\n");
	kfree(filp->private_data);
	debug("leaving\n");
	return 0;
}

static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
	/*The POSIX standard, states that if an inappropriate ioctl
	command has been issued, then -ENOTTY should be returned. */
    if (_IOC_TYPE(cmd) != LUNIX_IOC_MAGIC) return -ENOTTY;
    if (_IOC_NR(cmd) > LUNIX_IOC_MAXNR) return -ENOTTY;

	struct lunix_chrdev_state_struct *state;

	state = filp->private_data;

	switch(cmd) {
		case LUNIX_IOC_TYPE:
			if (down_interruptible(&state->lock))
				return -ERESTARTSYS;
			state->ioctl_type = (state->ioctl_type) ? 0 : 1;
			up (&state->lock);
			break;
		default:
			return -ENOTTY;
	}

	return 0;
}
```

```
static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt,
loff_t *f_pos)
{
	ssize_t ret;
	debug("entering\n");
	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) {
			/* ? */
			up(&state->lock);
			/* The process needs to sleep */
			/* See LDD3, page 153 for a hint */
			if(wait_event_interruptible(sensor->wq, (state->buf_timestamp <
sensor->msr_data[state->type]->last_update)))
				return -ERESTARTSYS;
			if(down_interruptible(&state->lock))
				return -ERESTARTSYS;
		}
	}

	/* End of file */
	/* ? */
	/*never happens (endless stream)*/
	if(*f_pos > state->buf_lim){
		ret=0;
		goto out;
	}

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

	if(copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)){
		ret = -EFAULT;
		goto out;
	}

	if(*f_pos + cnt == state->buf_lim) // no unused data in textual buffer (reset
pointers)
		*f_pos = state->buf_lim = 0;
	else
		*f_pos += cnt;
	ret = cnt;
	/* Auto-rewind on EOF mode? */
	/* ? */
out:
	/* Unlock? */
	up(&state->lock);
	debug("leaving\n");
	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;

       debug("initializing character device\n");
       cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
       lunix_chrdev_cdev.owner = THIS_MODULE;

       dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
       /* ? */
       /* register_chrdev_region? */
       ret = register_chrdev_region(dev_no, lunix_minor_cnt, "Lunix_TNG");
       if (ret < 0) {
              debug("failed to register region, ret = %d\n", ret);
              goto out;
       }
       /* ? */
       /* cdev_add? */
       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");
}
```

- lunix_chrdev_init()

Η συνάρτηση αυτή καλείται όταν εισάγουμε το module στον πυρήνα με την κλήση συστήματος insmod και αρχικοποιεί τον οδηγό. Συγκεκριμένα, η cdev_init() που καλείται αρχικά αρχικοποιεί το struct cdev του driver και δηλώνει τα file operations που υποστηρίζει ο driver. Το macro MKDEV δημιουργεί ένα struct dev_t που περιέχει το major number που προσδιορίζει το driver. Με τη

συνάρτηση register_chrdev_region() δηλώνουμε το εύρος των minor numbers που χρειαζόμαστε για να υποστηρίξουμε πολλαπλά devices. Τέλος, με τη cdev_add() προστίθεται ο driver στο σύστημα.

- lunix_chrdev_open()

Εδώ αρχικοποιούμε τη δομή lunix_chrdev_state_struct με βάση τη συσκευή που χρησιμοποιούμε. Για τη δέσμευση μνήμης χρησιμοποιούμε την kzalloc (αντίστοιχα την kfree στην release) ενώ αρχικοποιούμε το type και το sensor με βάση τη σύμβαση για τα minor numbers που είναι αριθμός αισθητήρα * 8 + μέτρηση.

- lunix_chrdev_read()

Η συνάρτηση αυτή καλείται σε process context όποτε μία διεργασία κάνει system call read σε κάποιο ανοιχτό αρχείο με major number που αντιστοιχεί στο major number του οδηγού μας. Οι δομές struct file των ανοιχτών αρχείων είναι διαφορετικές για διαφορετικές διεργασίες που κάνουν open στο ίδιο ειδικό αρχείο, άρα είναι διαφορετικός και ο character device buffer που έχει τα μορφοποιημένα δεδομένα. Όταν όμως δύο διεργασίες έχουν σχέση γονέας-παιδί, έχουν τα ίδια ανοιχτά αρχεία, οπότε και τους ίδιους buffers. Για αυτή την περίπτωση χρειαζόμαστε σημαφόρο για να διατηρήσουμε το συγχρονισμό πάνω στα κοινά δεδομένα. Κάθε φορά που τρέχει η read εκ μέρους κάποιας διεργασίας κλειδώνουμε από την αρχή μέχρι το τέλος ώστε να μην μπορούν να διαβάσουν πολλοί ταυτόχρονα από τον ίδιο buffer.
Η συγκεκριμένη υλοποίηση της read είναι σε blocking mode. Για να μην δημιουργηθεί deadlock, αφήνουμε το κλείδωμα κάθε φορά που "κοιμίζουμε" μία διεργασία επειδή δεν έχουμε καινούργια δεδομένα.
Αφού έρθουν δεδομένα, ελέγχουμε αρχικά αν ο δείκτης ανάγνωσης είναι μεγαλύτερος του buf_lim. Ωστόσο, με βάση την πολιτική που έχουμε ακολουθήσει για τους δείκτες f_pos και buf_lim και επειδή δεν έχουμε υλοποιήσει την lseek (οπότε ο χρήστης δεν μπορεί να μετακινήσει τον δείκτη ανάγνωσης), αυτή η συνθήκη δεν πρόκειται να ικανοποιηθεί ποτέ.
Η πολιτική που έχουμε ακολουθήσει σχετικά με το δείκτη ανάγνωσης και το buf_lim είναι η εξής: κάθε χρονική στιγμή ο character device buffer περιέχει μία μορφοποιημένη μέτρηση. Ο δείκτης buf_lim περιέχει το μέγεθος της μέτρησης και ο f_pos δείχνει στην τρέχουσα θέση ανάγνωσης στο buffer. Αν ο χρήστης ζητήσει λιγότερα δεδομένα από όσα υπάρχουν αυτή τη στιγμή στο buffer, του επιστρέφουμε όσα ζητάει και αυξάνουμε το δείκτη f_pos ώστε να δείχνει στην καινούργια θέση. Αν ο χρήστης ζητήσει περισσότερα δεδομένα από όσα έχει αυτή τη στιγμή ο buffer, επιστρέφουμε όσα έχουμε, μειώνουμε αντίστοιχα την τιμή επιστροφής ώστε να συμφωνεί με το πλήθος των επιστρεφόμενων bytes και μηδενίζουμε τους δείκτες f_pos και buf_lim.

- lunix_chrdev_update()

Η συνάρτηση αυτή καλείται από την read όποτε ο character device buffer είναι άδειος ώστε να διαβάσει μια καινούργια μέτρηση από τον sensor buffer και να την εισάγει μορφοποιημένη στον character device buffer. Η συνάρτηση ελέγχει αν υπάρχουν καινούργια δεδομένα στον sensor buffer καλώντας την lunix_chrdev_state_needs_refresh() που συγκρίνει τα timestamps στον sensor buffer και στον character device buffer. Για να προσπελάσουμε το sensor state κλειδώνουμε πρώτα με το spinlock του sensor state ώστε να υπάρχει συνέπεια στα καινούργια δεδομένα (μέτρηση και timestamp) που διαβάζουμε (διαφορετικά θα μπορούσε ενδεχομένως η sensor_update() να ανανεώσει το timestamp καθώς διαβάζουμε την μέτρηση και εν τέλει να διαβάσουμε ασυνεπή δεδομένα). Το sensor state προσπελαύνεται από την lunix_chrdev_update() σε process context και από την sensor_update() σε interrupt context. Γι αυτό το λόγο χρησιμοποιούμε spinlock για το συγχρονισμό. Τέλος, αφού διαβάσουμε την καινούργια μέτρηση την μορφοποιούμε με βάση τα lookup tables και την εισάγουμε στον character device buffer. Με το συγχρονισμό στον character device buffer δεν

έχουμε κάποιο θέμα καθώς έχουμε δεσμεύσει το σημαφόρο πρωτού καλέσουμε την lunix_chrdev_update().

Υποσημείωση:

Για το κλείδωμα του spinlock στην lunix_chrdev_update() έχουμε χρησιμοποιήσει τις συναρτήσεις spin_lock_irqsave και spin_unlock_irqrestore ώστε να αποφύγουμε το deadlock. Συγκεκριμένα, αν συμβεί interrupt και τρέξει η sensor_update(), αυτή θα προσπαθήσει να αποκτήσει το spinlock για να ανανεώσει τους sensor buffers, το οποίο όμως κρατάει η lunix_chrdev_update(). Αφού η sensor_update() τρέχει σε interrupt context και έχει "εκτοπίσει" από τη cpu την sensor_update(), που κρατάει το lock, δεν θα μπορέσει να το αποκτήσει ποτέ και θα οδηγηθούμε σε deadlock. Χρησιμοποιώντας τις παραπάνω συναρτήσεις για το κλείδωμα και ξεκλείδωμα, απενεργοποιούμε τα interrupts ενόσω η lunix_chrdev_update() κρατάει το spinlock και κρατμαε την τελευταια καταστασηη των interrupt στη μεταβήτή flags, οπότε δεν μπορεί να συμβεί αυτό το σενάριο.

- lunix_chrdev_ioctl()

Για την την υλοποίηση της λειτουργίας ioctl προσθέσαμε στο αρχείο chrdev.h τα εξής:

- ένα παραπάνω πεδίο στο **struct lunix_chrdev_state_struct** που το ονομάσαμε ioctl_type και το αρχικοποιούμε στην lunix_chrdev_open στην τιμή 0. Χρησιμοποιούμε την τιμή 0 για να δηλώσουμε στην lunix_chrdev_update ότι θα επεξεργαστούμε τα δεδομένα που παιρνουμε από το driver και θα τα εμφανίσουμε επεξεργασμένα (σε δεκαδική μορφή στο χρηστη στο userspace), ενώ χρησιμοποιούμε την τιμή 1 για να δηλώσουμε ότι τα δεδομένα θα επιστραφούν (στη συναρτηση lunix_chrdev_update) σε δεκαεξαδική μορφή όπως ακριβώς τα διαβάσαμε από το driver .

- τον ορισμό μίας ioctl command ως εξής:

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

#define LUNIX_IOC_MAXNR                 0
```

(με κόκκινο χρώμα είναι η εντολή που προσθέσαμε)

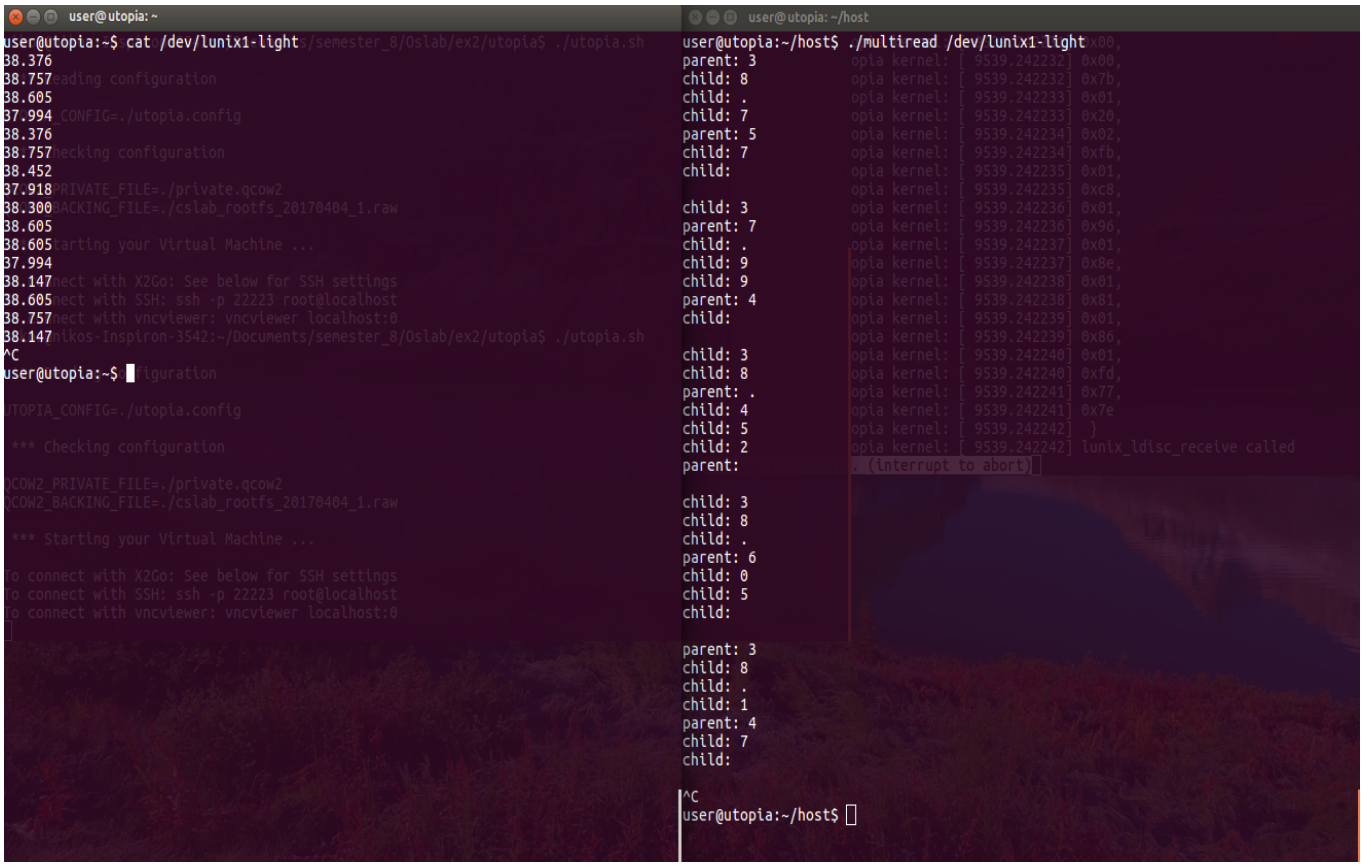
Η εναλλαγή της τιμής της μεταβλητής ioctl_type από 1 σε 0 ή από 0 σε 1 γίνεται όταν καλούμε την ioctl command LUNIX_IOC_TYPE στο πρόγραμμά μας σε userspace. Για την αλλαγή της τιμής iotcl_type στο chrdev.c υλοποιήσαμε την συνάρτηση lunix_chrdev_ioctl() όπου αρχικά ελέγχουμε αν η παράμετρος cmd έχις valid τιμές και αν όχι επιστρέφουμε –ENOTTY όπως ορίζεται στο πρότυπο posix. Στη συνέχεια ελέγχουμε αν λάβαμε ioctl command LUNIX_IOC_TYPE και αν ναι αλλάζουμε την τιμή της μεταβλητής ioctl_type αν ήταν 1 σε 0 ή το αντίστροφο. Έτσι όπως αναφέραμε και παραπάνω η συνάρτηση lunix_chrdev_update ανάλογα με την τιμή της μεταβλητής iotcl_type τυπώνει κατάλληλα τα δεδομένα.

Παράδειγμα εκτέλεσης του driver

Έχουμε υλοποιήσει ένα πρόγραμμα στο userspace, το οποίο ανοίγει κάποιο από τα ειδικά αρχεία της συσκευής χαρακτήρων, δημιουργεί ένα παιδί (κάνοντας fork()) και εν συνεχεία οι δύο διεργασίες διαβάζουν ένα χαρακτήρα τη φορά. Ο κώδικας δίνεται παρακάτω:

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

int main(int argc, char * argv[]) {

	pid_t pid;
	int fd = 0;
	size_t numRead = 0;
	char buf[1];

	if (argc < 2){
		fprintf(stderr,"Usage: ./testMultipleRead infile_path\n");
		return 1;
	}


	if ( (fd = open(argv[1], O_RDONLY)) < 0) {
		perror(argv[1]);
		return 1;
	}

	if ( (pid = fork()) < 0) {
		perror("fork");
		return 1;
	}

	if (pid != 0) {
		do {
			numRead = read(fd, buf, 1);
			if (numRead == -1){
				perror("read");
				exit(1);
			}
			printf ("parent: %c\n",buf[0]);
			sleep(3);
		} while(numRead > 0);
		perror("internal error");
		exit(1);
	}

	do {
		numRead = read(fd, buf, 1);
		   if (numRead == -1){
			perror("read");
			     exit(1);
		   }
		printf("child: %c\n",buf[0]);
		sleep(1);
	  } while(numRead > 0);

	if (close(fd) < 0) {
		perror("close");
		return 1;
	}

	return 0;
}
```

Το αποτέλεσμα της εκτέλεσης είναι το εξής:



όπου βλέπουμε ότι ο συγχρονισμός όσον αφορά τον character device buffer και τα αποτελέσματα της read είναι τα αναμενόμενα.

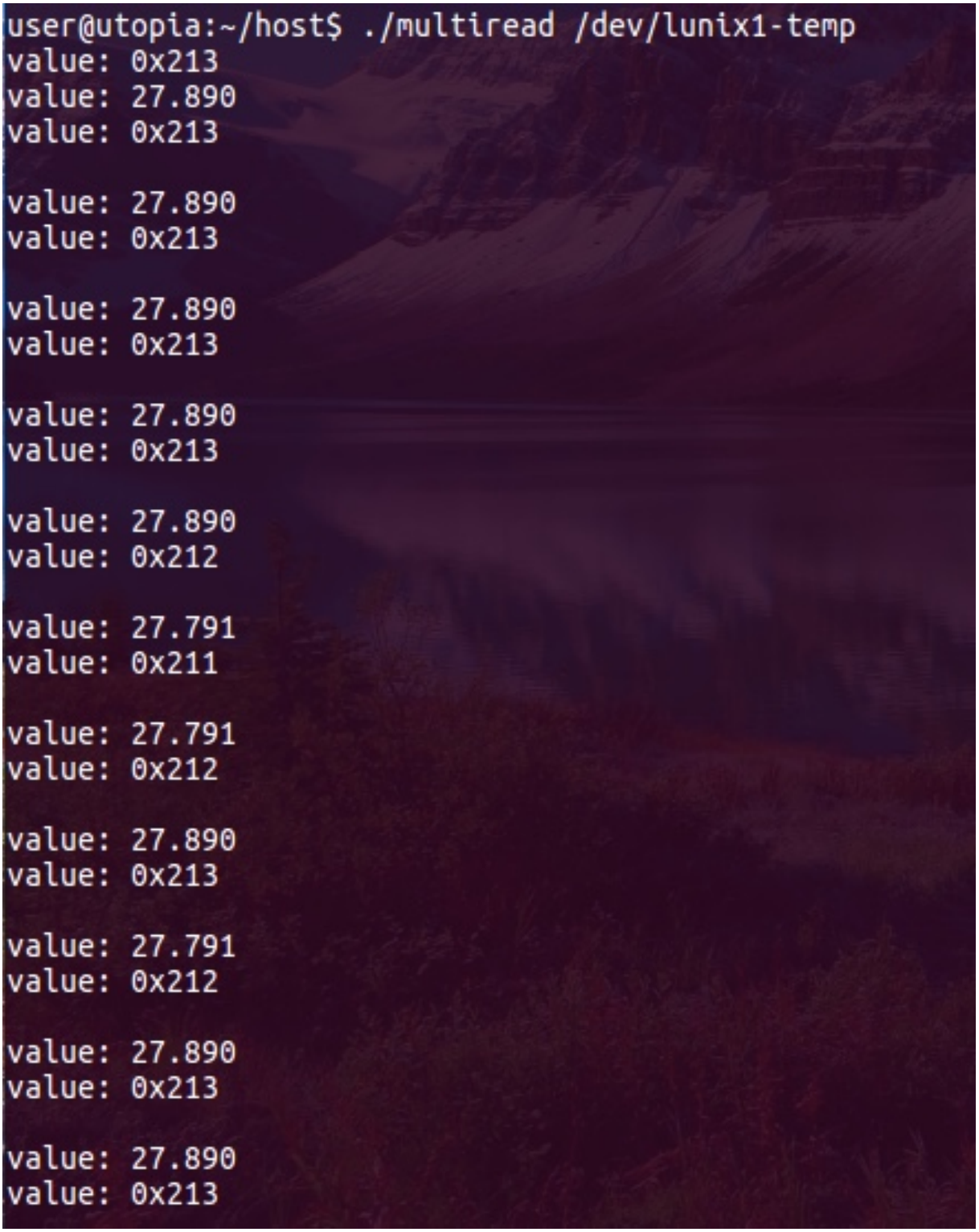
Παράδειγμα εκτέλεσης του driver χρησιμοποιώντας τη λειτουργία **ioctl**

Παρακάτω παρουσιάζεται ένα πρόγραμμα userspace όπου διαβάζει δεδομένα και καλεί την ioctl σε κάθε επανάληψη ώστε τα δεδομένα που διαβάζουμε να εναλάσσονται μεταξύ μορφοποιημένων και δεκαεξαδικών δεδομένων. Το πρόγραμμα είναι το εξής:

```
#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 "./lunix-tng-helpcode-20170404/lunix-chrdev.h"

int main(int argc, char * argv[]) {

	int fd = 0;
	size_t numRead = 0;
	char buf[10];
```

```
	if (argc < 2){
		fprintf(stderr,"Usage: ./testMultipleRead infile_path\n");
			return 1;
	}


	if ( (fd = open(argv[1], O_RDONLY)) < 0) {
		perror(argv[1]);
		return 1;
	}

		do {
			numRead = read(fd, buf, 10);
			if (numRead == -1){
				perror("read");
				exit(1);
			}
			printf ("value: %s",buf);
			//printf("ioctl\n");
			ioctl(fd, LUNIX_IOC_TYPE, NULL);
		} while(numRead > 0);

	if (close(fd) < 0) {
		perror("close");
		return 1;
	}

	return 0;
}
```

Το αποτέλεσμα της εκτέλεσης είναι το εξής:



όπου παρατηρούμε την εναλλαγή από επεξεργασμένα δεδομένα σε δεκαεξαδικά μη επεξεργασμένα δεδομένα.