Άσκηση 2

open(struct inode *inode, struct file *filp)

Χρήσιμες Δομές

struct inode: Είναι η δομή με την οποία αναπαρίστανται στον πυρήνα τα αρχεία του file system. Περιλαμβάνει δύο πεδία που για εμάς που γράφουμε driver για character device έχουν ενδιαφέρον. Αυτά είναι τα ***dev_t i_rdev*** και ***struct cdev *i_cdev***.
Το ***i_rdev*** έχει τον αριθμό που αντιστοιχεί στη συσκευή χαρακτήρων που αναπαριστά το αρχείο. Υπενθυμίζεται πως ο αριθμός αυτός είναι 32 bits. Από αυτά, τα 12 πιο σημαντικά bits αντιστοιχούν στο ***major number*** της συσκευής, που είναι ενδεικτικός του driver που τη διαχειρίζεται. Τα υπόλοιπα 20 αντιστοιχούν στο ***minor number***. Για την άσκηση με το ***LUNIX*** τα 3 λιγότερο σημαντικά bits χρησιμοποιήθηκαν για την περιγραφεί του τύπου της μέτρησης, ενώ τα 17 πιο σημαντικά για τον αισθητήρα προέλευσης.
Το ***i_cdev*** είναι ένας δείκτης σε μία δομή τύπου ***cdev***. Οι δομές αυτές αποτελούν τον τρόπο με τον οποίο ο πυρήνας περιγράφει εσωτερικά τους drivers για συσκευές χαρακτήρων.
Στην άσκηση, για να πάρουμε τα major και minor numbers, χρησιμοποιούμε τις μακροεντολές:

unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);

struct file: Η δομή αυτή χρησιμοποιείται από τον πυρήνα για την αναπαράσταση ενός ανοιχτού αρχείου μίας διεργασίας. Το πεδίο της που μας ενδιαφέρει είναι το ***private_data***. Το πεδίο αυτό περιλαμβάνει πληροφορίες για την κατάσταση του ανοιχτού αρχείου. Για την άσκησή μας, το συγκεκριμένο πεδίο έδειχνε σε μία δομή τύπου:

```
struct lunix_chrdev_state_struct {
        enum lunix_msr_enum type; // τύπος μέτρησης
        struct lunix_sensor_struct *sensor;

        /* A buffer used to hold cached textual info */
        int buf_lim; // πλήθος χαρακτήρων που αναπαριστούν την τιμή της μέτρησης
        unsigned char buf_data[LUNIX_CHRDEV_BUFSZ]; // buffer με τιμή μέτρησης
        uint32_t buf_timestamp; // χρόνος τελευταίας ενημέρωσης

        struct semaphore lock; /* σημαφόρος για να μην πειράζουν τη συσκευή
χαρακτήρων πολλά νήματα της ίδιας διεργασίας ή συγγενικές της (αφού
κληρονομούνται ανοιχτά αρχεία) */
};
```

Συγκεκριμένα, βρίσκαμε σε ποια δομή τύπου ***lunix_sensor_struct*** αντιστοιχούσε στον αισθητήρα από τον οποίο θέλαμε να πάρουμε μετρήσεις βάζαμε τη διεύθυνσή της στο αντίστοιχο πεδίο της δομής. Επιπλέον γινόντουσαν και κάποιες αρχικοποιήσεις για τα υπόλοιπα πεδία της δομής που φαίνονται παρακάτω. Φυσιολογική ολοκλήρωση των προηγούμενων οδηγούσε στην επιστροφή της τιμής 0.

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

	debug("entering open phase\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>]
	 */
	dev_minor_num = iminor(inode);
	sensor = &lunix_sensors[dev_minor_num/8];
	type = dev_minor_num%8;

	/* Allocate a new Lunix character device private state structure */
	state = (struct lunix_chrdev_state_struct*) vmalloc(sizeof(struct
lunix_chrdev_state_struct));
	if (!state) {
		printk(KERN_ERR "Failed to allocate memory for Lunix character device
state structure\n");
		ret = -ENOMEM;
		goto out;
	}
	state->sensor = sensor;
	state->type = type;
	state->buf_lim = 0;
	sema_init(&state->lock, 1);
	filp->private_data = state;
out:
	debug("leaving, with ret = %d\n", ret);
	return ret;
}
```

read(struct file *filp, char __user *usrbuf, size_t cnt, loff_t *f_pos)

Χρήσιμες Δομές

char __user *usrbuf: Ένας buffer με τον οποίο αλληλεπιδρούμε με το χώρο χρήστη. Σε αυτόν αντιγράφουμε μετρήσεις που θέλουμε να εμφανιστούν στην οθόνη. Οι μετρήσεις είναι γραμμένες υπό μορφή ακολουθιών χαρακτήρων (εξ' ου και το char *). Επιπλέον, επειδή πρόκειται για δομή χώρου χρήστη, απαγορεύεται να την κάνουμε απευθείας dereference στο χώρο πυρήνα. Για το λόγο αυτό, προκειμένου να αντιγράψουμε τα δεδομένα που θέλουμε, χρησιμοποιούμε την εντολή:

```
if (copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)) {
        debug("user gave invalid adress");
        ret = -EFAULT;
        goto out;
}
```

Η ***copy_to_user*** είναι μία ***memcpy*** για τον πυρήνα. Η διαφορά της από τη συμβατική ***memcpy*** είναι ότι πραγματοποιεί ελέγχους σχετικά με το αν η διεύθυνση που έχουμε στο ***usrbuf*** ανήκει στο χρήστη που κάνει ***read***. Εάν δεν ανήκει, η συνάρτηση αποτυγχάνει, δίνοντας ως τιμή επιστροφής το πλήθος των bytes που δεν αντιγράφτηκαν (δηλαδή όλα). Θεωρούμε πως αυτή είναι και η μοναδική περίπτωση αποτυχίας. Σε κάθε άλλη, η τιμή επιστροφής είναι 0. Δεδομένου ότι, θεωρητικά, υπάρχει περίπτωση να κοιμηθεί η διεργασία που κάνει ***copy_to_user***, δεν είναι καλό να χρησιμοποιούνται ***spinlocks*** για την προστασία της εκτέλεσης τμήματος κώδικα που την περιλαμβάνει.

f_pos: Το ***f_pos*** είναι ένας δείκτης που η τιμή του δηλώνει την θέση μέσα στο ειδικό αρχείο όπου βρισκόμαστε (σε bytes). Αν είναι 0, σημαίνει πως βρισκόμαστε στην αρχή νέας μέτρησης, οπότε πρέπει να γίνει ενημέρωση. Αν βρισκόμαστε στο τέλος του αρχείου (πράγμα αδύνατο υπό κανονικές συνθήκες, αφού έχουμε απλά ένα ρεύμα από μετρήσεις που θεωρητικά δεν τελειώνει), έχουμε ****f_pos >= state->buf_lim*sizeof(unsigned char)*** (***sizeof(unsigned char)*** κανονικά έχει τιμή 1, οπότε θα μπορούσε θεωρητικά να παραλειφθεί). Τα προηγούμενα φαίνονται στον παρακάτω κώδικα:

```
if (*f_pos == 0) {
        while (lunix_chrdev_state_update(state) == -EAGAIN) {
                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;
            }
        }
}

if (*f_pos >= state->buf_lim*sizeof(unsigned char)) {
        *f_pos = 0;
        ret = 0;
        goto out;
}
```

Επειδή έχουμε αλληλεπίδραση με τα ***sensor structs*** για να διαβάσουμε δεδομένα από αυτά, πρόκειται για κρίσιμο τμήμα. Γι' αυτό και φαίνεται η χρήση σημαφόρων σε διάφορα σημεία. Όταν η διαδικασία πρόκειται να κοιμηθεί, καθώς περιμένει νέα δεδομένα, δεν χρειάζεται να κρατήσει το σημαφόρο. Υλοποιείται τόσο ***blocking*** όσο και ***non-blocking I/O***. Χρησιμοποιούνται οι interruptible εκδοχές των ***wait_event*** και ***down_interruptible***, προκειμένου με ***CTRL+C*** να επιστρέφει ο έλεγχος στο χρήστη.

size_t cnt: Δηλώνει το πλήθος των bytes που θέλουμε να διαβάσουμε από τη συσκευή. Σε περίπτωση που ζητείται να διαβαστούν περισσότερα από αυτά που είναι διαθέσιμα, γίνεται ξανά υπολογισμός του με την ακόλουθη μέθοδο:

```
if (*f_pos + cnt > state->buf_lim*sizeof(unsigned char)) {
        cnt = state->buf_lim*sizeof(unsigned char) - *f_pos;
        debug("more bytes requested than existing, new cnt is %d", (int) cnt);
}
```

Ο συνολικός κώδικας είναι:

```
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);

        debug("about to read from sensor\n");
        if (down_interruptible(&state->lock)){
```

```
                return -ERESTARTSYS;
        }

        if (*f_pos == 0) {
                while (lunix_chrdev_state_update(state) == -EAGAIN) {
                        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;
                        }
                }
        }

        if (*f_pos >= state->buf_lim*sizeof(unsigned char)) {
                *f_pos = 0;
                ret = 0;
                goto out;
        }

        if (*f_pos + cnt > state->buf_lim*sizeof(unsigned char)) {
                cnt = state->buf_lim*sizeof(unsigned char) - *f_pos;
        }

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

        *f_pos += cnt;
        ret = cnt;
        if (*f_pos == state->buf_lim*sizeof(unsigned char)) {
                *f_pos = 0;
        }

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

Τα περισσότερα από τα παραπάνω (υπολογισμός cnt κλπ) ισχύουν εντελώς αντίστοιχα και στην περίπτωση που ζητείται να υλοποιηθεί η write αντί της read.

*needs_refresh(struct lunix_chrdev_state_struct *state)*

Συγκρίνονται timestamps για να διαπιστωθεί αν χρειάζεται ανανέωση.

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

        WARN_ON ( !(sensor = state->sensor));
        if (state->buf_timestamp != sensor->msr_data[state->type]->last_update) {
                return 1;
        }
        return 0;
}
```

*update(struct lunix_chrdev_state_struct *state)*

Το μόνο που έχει ενδιαφέρον είναι ότι με το sensor_struct αλληλεπιδρά και η ***sensors_update*** (που φαίνεται πιο κάτω), η οποία εκτελείται σε interrupt context. Επομένως, χρειάζεται να χρησιμοποιήσουμε ***spinlocks***. Κατά τα άλλα, έχουμε μετατροπή από αριθμό σε συμβολοσειρά, που μπορούσε να γίνει και πιο γρήγορα (shit happens :P).

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
{
	struct lunix_sensor_struct *sensor;
	uint32_t value, timestamp;
	int i;
	long lookup_table_value, integer_part, decimal_part;
	unsigned long flags;

	debug("start of update function\n");

	if (!lunix_chrdev_state_needs_refresh(state)) {
		return -EAGAIN;
	}

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

	spin_lock_irqsave(&sensor->lock, flags);
	value = sensor->msr_data[state->type]->values[0];
	timestamp = sensor->msr_data[state->type]->last_update;
	spin_unlock_irqrestore(&sensor->lock, flags);

	if (state->type == 0) lookup_table_value = lookup_voltage[value];
	else if (state->type == 1) lookup_table_value = lookup_temperature[value];
	else lookup_table_value = lookup_light[value];

	state->buf_lim = 0;

	if (lookup_table_value < 0) {
		state->buf_data[state->buf_lim++] = '-';
		lookup_table_value = -lookup_table_value;
	}

	integer_part = lookup_table_value/1000;
	decimal_part = lookup_table_value%1000;

	if (integer_part == 0) {
```

```
        state->buf_data[state->buf_lim++] = '0';
        i = 0;
    }
    else {
        i = 1;
        while (integer_part >= i) i *= 10;
        i /= 10;
    }
    while (i != 0) {
        state->buf_data[state->buf_lim++] = (unsigned char) (integer_part/i + 48);
        integer_part %= i;
        i /= 10;
    }

    state->buf_data[state->buf_lim++] = '.';

    i = 1;
    state->buf_lim += 3;
    while (i <= 3) {
        state->buf_data[state->buf_lim - i] = (unsigned char) (decimal_part%10 +
48);
        decimal_part /= 10;
        i++;
    }

    state->buf_data[state->buf_lim++] = '\n';
    state->buf_data[state->buf_lim] = '\0';
    state->buf_timestamp = timestamp;

    return 0;
}
```

chrdev_init

Καλείται κάθε φορά που γίνεται insmod το kernel object του driver.

```
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);
	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 with dev_no %d and range %d\n", (int)
dev_no, (int) lunix_minor_cnt);
	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;
}
```

sensors_update(struct lunix_sensor_struct *s, uint16_t batt, uint16_t temp, uint16_t light)

Η συνάρτηση αυτή εκτελείται σε ***interrupt context*** (γι' αυτό και χρησιμοποιούνται spinlocks). Πρόκειται για τη συνάρτηση που ενημερώνει τους αισθητήρες. Αφού ολοκληρωθεί η ενημέρωση, ξυπνάνε όλες οι διεργασίες που περιμένουν να διαβάσουν δεδομένα.

```
void lunix_sensor_update(struct lunix_sensor_struct *s, uint16_t batt, uint16_t temp,
uint16_t light)
{
        spin_lock(&s->lock);

        /*
         * Update the raw values and the relevant timestamps.
         */
        s->msr_data[BATT]->values[0] = batt;
        s->msr_data[TEMP]->values[0] = temp;
        s->msr_data[LIGHT]->values[0] = light;

        s->msr_data[BATT]->magic = s->msr_data[TEMP]->magic = s->msr_data[LIGHT]-
>magic = LUNIX_MSR_MAGIC;
        s->msr_data[BATT]->last_update = s->msr_data[TEMP]->last_update = s-
>msr_data[LIGHT]->last_update = get_seconds();

        spin_unlock(&s->lock);

        /*
         * And wake up any sleepers who may be waiting on
         * fresh data from this sensor.
         */
        wake_up_interruptible(&s->wq);
}
```

Θέμα 2 (Επαναληπτική 2016)

Εδώ δίνεται μία λύση του θέματος που αναφέρεται παραπάνω.

i) ***static ssize_t write(struct file *filp, const char __user *buf, ssize_t len, loff_t *pos)***
{
```
        int i;
        ssize_t ret;
        unsigned long flags;
#define TMP_SIZE 1024
        char tmp_buf[TMP_SIZE];

        struct cache_device *cache;

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

        if (len > TMP_SIZE) {
                len = TMP_SIZE;
        }

        spin_lock_irqsave(&cache->lock, flags);
        while (cache->cnt == BUF_SIZE) {
                if (!cache->flush_in_progress) {
                        cache->flush_in_progress = 1;
                        initiate_flush(cache);
                }
                while (cache->flush_in_progress) {
                        spin_unlock_irqrestore(&cache->lock, flags);
                        if (wait_event_interruptible(cache->wq, (cache->cnt != BUF_SIZE) &&
(!cache->flush_in_progress))) {
                                return -ERESTARTSYS;
                        }
                        spin_lock_irqsave(&cache->lock, flags);
                }
        }
        if (len > BUF_SIZE - cache->cnt) {
                len = BUF_SIZE - cache->cnt;
        }
        spin_unlock_irqrestore(&cache->lock, flags);

        if (copy_from_user(tmp, buf, len)) {
                return -EFAULT;
```

```
        }

        spin_lock_irqsave(&cache->lock, flags);
        for (i = 0; i < len; i++) {
                cache->buf[i + cache->cnt] = tmp_buf[i];
        }
        cache->cnt += len;
        spin_unlock_irqrestore(&cache->lock, flags);

        ret = len;
        return ret;
}
```

Προς αποφυγή της χρήσης spinlock για την ***copy_from_user***, χρησιμοποιήσαμε έναν temporary buffer. Ο buffer αυτός δεν είναι κοινόχρηστος (σε αντίθεση με την ***cache struct***) οπότε δεν χρειάζεται προστασία. Αντιγράφουμε σε αυτόν τα δεδομένα από το χώρο χρήστη και, δεδομένου ότι αυτός έχει οριστεί στον πυρήνα, μπορούμε μετά να τον κάνουμε dereference, προκειμένου να αντιγράψουμε ό,τι δεδομένα χρειαζόμαστε.

ii)
```
void flush_completed(unsigned int intr_mask)
{
        unsigned long flags;

        if (!intr_mask) spin_lock(&cache->lock);
        else spin_lock_irqsave(&cache->lock, flags);
        cache->cnt = 0;
        cache->flush_in_progress = 0;
        wake_up_interruptible(&cache->wq);
        if (!intr_mask) spin_unlock(&cache->lock);
        else spin_unlock_irqrestore(&cache->lock, flags);
}
```

Ο ρόλος του ***intr_mask*** δεν είναι πολύ ξεκάθαρος με βάση την εκφώνηση, οπότε θεωρήθηκε πως παίζει το ρόλο που έχουν συνήθως οι μάσκες διακοπών: δηλώνει αν πρέπει ή όχι να είναι ενεργές οι διακοπές όταν εκτελείται ο κώδικας. Ανάλογα, λοιπόν, με την τιμή του, επιλέγουμε αν θα χρησιμοποιήσουμε ***spin_lock*** ή ***spin_lock_irqsave***. Κατά τα άλλα, θεωρούμε πως η δομή ***cache*** έχει οριστεί σε τέτοιο σημείο ώστε η ***flush_completed*** να τη "βλέπει" (πχ global στο ίδιο αρχείο κώδικα). Αλλιώς, η συνάρτηση θα χρειαζόταν ένα επιπλέον όρισμα.

Σε αυτό το σημείο, πρέπει να κάνουμε και κάποιες σημαντικές παρατηρήσεις για την δομή ***cache***. Η δομή αυτή είναι στο πεδίο ***private_data***. Το συγκεκριμένο πεδίο χρησιμοποιείται για την περιγραφεί της κατάστασης ενός ανοιχτού αρχείου από τον πυρήνα. Στην περίπτωση της ***read*** που υλοποιήσαμε στην άσκηση μέσα στο εξάμηνο, κάθε διεργασία που διάβαζε από τους αισθητήρες είχε δική της τέτοια δομή (αφού γινόταν

allocate κατά την ***open***). Η μόνη εξαίρεση που υπήρχε ήταν άμα είχαμε συγγενικές διεργασίες. Εδώ, από την άλλη μεριά, θέλουμε την ίδια δομή ***cache*** να την βλέπουν και μη συγγενικές διεργασίες. Ο λόγος είναι απλός. Η ***cache*** (το φυσικό στοιχείο, όχι η δομή) είναι μία. Αν χρησιμοποιούσαμε διαφορετική δομή ανά διεργασία για να την περιγράψουμε, θα κατέληγε η μία διεργασία να κάνει overwrite τα δεδομένα της άλλης. Επομένως, η δομή αυτή θα πρέπει να έχει γίνει allocate σε κατάλληλο σημείο, ώστε όλες οι διεργασίες να χρησιμοποιούν την ίδια δομή (όπως γίνεται με τα ***sensor structs*** στην εργαστηριακή άσκηση, που είναι τα ίδια για όλες τις διεργασίες, βλ. ***lunix-sensors.c*** στον βοηθητικό κώδικα της άσκησης).

iii) Το κλείδωμα lock είναι ***spinlock***. Ο λόγος που έγινε η συγκεκριμένη επιλογή είναι ότι κάποια από τα πεδία που προστατεύει (το ***flush_in_progress*** και το ***cnt***) προσπελάζονται τόσο από κώδικα που εκτελείται σε ***process context*** (τη ***write***), όσο και από κώδικα που εκτελείται σε ***interrupt context*** (τη ***flush_completed***).
Δύο περιπτώσεις όπου η απουσία του lock θα δημιουργούσε πρόβλημα είναι οι ακόλουθες:
1) Αν δεν υπήρχε το lock κατά τον έλεγχο της συνθήκης ***cache->cnt == BUF_SIZE*** μπορεί η διεργασία να έχανε τον επεξεργαστή, να εκτελούταν ο κώδικας της ***flush_completed*** και μετά η διεργασία να κατέληγε να κοιμάται χωρίς να αναμένεται να υπάρχει κάποιος να την ξυπνήσει.
2) Αν δεν υπήρχε το lock στο τμήμα που γράφει στην cache, θα μπορούσε να χάσει μία διεργασία τον επεξεργαστή προτού αλλάξει η τιμή του cnt, με αποτέλεσμα να γίνουν overwrite αυτά που έγραψε από άλλη διεργασία.

iv) Το πεδίο αυτό μετράει το πλήθος των bytes που έχουν γραφτεί στην ***cache***. Δεδομένου ότι τα αντικείμενα τύπου ***char*** αναπαρίστανται με ***1 byte*** το καθένα, το ***cnt*** ουσιαστικά εκφράζει το πλήθος των θέσεων του ***buf*** που έχουν γραφτεί. Τροποποιείται από την ***flush_completed***, όταν η ***cache*** αδειάζει. Επιπλέον, τροποποείται και από τον κώδικα της ***write*** όταν γράφονται περισσότερα bytes.

v) Αποτρέπεται με χρήση του πεδίου ***flush_in_progress***. Όταν αυτό έχει τιμή 1, μία διεργασία κοιμάται απλά, χωρίς να κάνει ***initiate_flush*** ξανά.