Εργαστήριο Λειτουργικών Συστημάτων
2ή Εργαστηριακή Άσκηση

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Η/Υ
2020/2021 - 7ο Εξάμηνο - Ομάδα 16

Όνοματεπώνυμο :	Ανδριόπουλος Κωνσταντίνος	Βόσινας Κωνσταντίνος
ΑΜ :	03116023	03116435

Συναρτήσεις οδηγού συσκευής χαρακτήρων

```
static int lunix_chrdev_state_needs_refresh(struct
lunix_chrdev_state_struct *state)
```

Με αυτήν την βοηθητική συνάρτηση ελέγχουμε αν το πεδίο *state->buf_data* περιέχει τα πιο πρόσφατα δεδομένα που έχουν φτάσει από την συσκευή. Συγκεκριμένα, ελέγχουμε για ισότητα τα πεδία *state->buf_timestamp* και *state->sensors[state->type]->last_update* (ανανεώνεται όταν έρχονται νέες μετρήσεις από την συσκευή και καλείται η *lunix_sensor_update()* του lunix_sensors.c). Αν τα πεδία διαφέρουν, τότε το state χρειάζεται ανανέωση.

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct
*state)
```

Με αυτήν την βοηθητική συνάρτηση ανανεώνουμε το state. Παρακάτω παρατίθενται τα σημαντικότερα κομμάτια του κώδικα της συνάρτησης:

```
struct lunix_sensor_struct *sensor;

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

uint32_t newdata, last_update;
long lookup = 0;

spin_lock(&sensor->lock);
newdata = sensor->msr_data[state->type]->values[0];
last_update = sensor->msr_data[state->type]->last_update;
spin_unlock(&sensor->lock);
```

Αρχικά, καλώντας την *lunix_chrdev_state_needs_refresh()*, ελέγχουμε αν το state χρειάζεται ανανέωση. Αν δεν χρειάζεται, η συνάρτηση επιστρέφει με τιμή *-EAGAIN*. Ειδάλλως, με την εντολή *spin_lock(&sensor->lock)*, αποκτάμε πρόσβαση στην δομή sensor. Ύστερα, φορτώνουμε τα νέα δεδομένα και το timestamp τους στις μεταβλητές newdata και last_update αντίστοιχα, και απελευθερώνουμε το lock με την εντολή spin_unlock(&sensor->lock). Με τον τρόπο αυτό, ξοδεύουμε την λιγότερη δυνατή ώρα μέσα στο critical section, ώστε η *lunix_sensor_update()* να μπορεί να αποκτήσει πρόσβαση στην δομή sensor, δίχως καθυστέρηση, αν έρθουν νέα δεδομένα από την συσκευή.

```
state->buf_timestamp = last_update;

switch (state->type) {
      case BATT:
            lookup = lookup_voltage[newdata];
            break;
      case TEMP:
            lookup = lookup_temperature[newdata];
            break;
      case LIGHT:
            lookup = lookup_light[newdata];
            break;
      case N_LUNIX_MSR:
            return -EFAULT;
}
```

Στην συνέχεια, ανανεώνουμε το πεδίο *state->last_update* (με την τιμή της μεταβλητής *last_update*) και ανάλογα με την τιμή του *state->type*, διαβάζουμε από τον αντίστοιχο lookup table και αποθηκεύουμε την τιμή στην μεταβλητή *lookup*.

```
long decimal, fractional;

decimal = lookup / 1000;
fractional = lookup % 1000;
if (lookup < 0){
      state->buf_lim = sprintf(state->buf_data, "-%ld.%ld\n", (-1)*decimal,
(-1)*fractional);
} else {
      state->buf_lim = sprintf(state->buf_data, "%ld.%ld\n", decimal,
fractional);
}
```

Ύστερα, φέρνουμε σε δεκαδική μορφή τα δεδομένα που διαβάσαμε (μεταβλητές *decimal* και *fractional*) και με την εντολή *state->buf_lim = sprintf(state->buf_data, "%ld.%ld\n", decimal, fractional)* ανανεώνουμε τα πεδία *buf_data* και *buf_lim* του state.

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

Η συνάρτηση που υλοποιεί την *open()* syscall. Παρακάτω, φαίνονται αναλυτικότερα τα κυριότερα μέρη της:

```
unsigned int minorNum, type, sensorNum;

minorNum = iminor(inode);

type = minorNum % 8;
sensorNum = minorNum / 8; //Number of the sensor device
```

Από το *inode struct* του αρχείου εξάγουμε τον minor number. Όπως αναφέρεται και στον οδηγό της εργαστηριακής άσκησης, στον minor number περιέχεται πληροφορία για τον αριθμό του sensor αλλά και για τον τύπο της μέτρησης. Τις πληροφορίες αυτές τις εξάγουμε και τις αποθηκεύουμε στις μεταβλητές *sensorNum* και *type* αντίστοιχα.

```
struct lunix_chrdev_state_struct *state;
state = kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);
```

Ύστερα, με χρήση της συνάρτησης *kmalloc()* κάνουμε allocate μνήμη για ένα νέο *lunix_chrdev_state_struct* το οποίο **θα διατηρεί το state του reading session** για τις διεργασίες που έχουν πρόσβαση στο *struct file *filp* του αρχείου. **Οι διεργασίες αυτές μπορεί να είναι η ίδια διεργασία που καλεί την open() καθώς επίσης και παιδιά αυτής, τα οποία κληρονομούν τα resources της**.

```
state->type = type;
state->sensor = &lunix_sensors[sensorNum];
state->buf_lim = 0;
state->buf_timestamp = 0;
sema_init(&state->lock, 1);
```

Ύστερα, αρχικοποιούμε τα πεδία του *struct lunix_chrdev_state_struct state*. Πιο συγκεκριμένα, αρχικοποιούμε τα πεδία *state->buf_lim* και *state->buf_timestamp* με τιμή 0, αρχικοποιούμε το πεδίο *state->type* με το *type* που εξάγαμε από το minur number και αρχικοποιούμε τον δείκτη *state->sensor* με την διεύθυνση του *lunix_sensors[sensorNum]*, όπου το *lunix_sensors* έχει δημιουργηθεί από την συνάρτηση *lunix_module_init()* στο *lunix-module.c*. Ακόμα, με την εντολή *sema_init(&state->lock, 1)* αρχικοποιούμε τον semaphore του state στην τιμή 1. Με αυτόν τον τρόπο, διασφαλίζουμε ότι θα έχει πρόσβαση στο state μόνο μία διεργασία κάθε φορά.

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

Αφού αρχικοποιήσουμε το state, το αποθηκεύουμε στο πεδίο *filp->private_data*, όπως φαίνεται παραπάνω, ώστε να χρησιμοποιηθεί μελλοντικά από διεργασίες που έχουν πρόσβαση σε αυτό το filp.

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

Η συνάρτηση *release()* καλείται κάθε φορά που ένα ανοιχτό αρχείο κλείνει. Στην περίπτωση που πολλαπλές διεργασίες έχουν πρόσβαση στο ανοιχτό αρχείο (π.χ. λόγω *fork()*), καλείται όταν η τελευταία από αυτές κλείσει το αρχείο. Επομένως απελευθερώνουμε τη μνήμη που δεσμεύσαμε για τα private_data του ανοιχτού αρχείου με την εντολή *kfree(filp->private_data)*.

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

Η συνάρτηση αυτή υλοποιεί την κλήση συστήματος read(). Καλείται κάθε φορά που γίνεται διάβασμα από αρχείο (π.χ. *cat lunix0-batt*). Συγκεκριμένα, ζητείται να διαβάσουμε από το cnt bytes, ξεκινόντας από τη θέση *f_pos* και να τα μεταφέρουμε στο user space buffer *usrbuf*.

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

if(down_interruptible(&state->lock))
return -ERESTARTSYS;
```

Αρχικά, λαμβάνουμε το state από το ανοιχτό αρχείο με την εντολή *state = filp->private_data,* και με βάση αυτή βρίσκουμε τον αισθητήρα που γίνεται το διάβασμα. Αρχικά, κλειδώνουμε το σημαφόρο της *state* με την εντολή *down_interruptible(&state->lock)* (αν δε μπορούμε να τον αποκτήσουμε, επιστρέφουμε κατάλληλο μήνυμα).

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

        up(&state->lock);

        if(wait_event_interruptible(sensor->wq,
lunix_chrdev_state_needs_refresh(state)))
            return -ERESTARTSYS;
         if(down_interruptible(&state->lock))
            return -ERESTARTSYS;
    }
}
```

Στη συνέχεια, ελέγχουμε αν υπάρχουν δεδομένα προς ανάγνωση. Αν τα περιεχόμενα του *f_pos* είναι 0, αυτό υποδεικνύει πως δεν υπάρχουν διαθέσιμα δεδομένα στο αρχείο, οπότε η διεργασία πρέπει να "κοιμηθεί". Καλείται η συνάρτηση *lunix_chrdev_state_update(state)* και αν ούτε αυτή δείξει πως υπάρχουν νέα δεδομένα, απελευθερώνουμε το σημαφόρο της κατάστασης με *up(&state->lock)*, και καλούμε την *wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state))*. Αυτή η συνάρτηση τοποθετεί όλες τις διεργασίες σε μια ουρά αναμονής του sensor μέχρι να έρθουν νέα δεδομένα. Όταν γίνει κάποιο update από την *lunix_sensor_update*, θα ξυπνήσει όλες τις διεργασίες στην ουρά με την εντολή *wake_up_interruptible(&s->wq),* ενώ ταυτόχρονα θα ικανοποιείται η συνθήκη *lunix_chrdev_state_needs_refresh(state),* οπότε οι διεργασίες θα συνεχίσουν. Σε περίπτωση που κατά τη διάρκεια αναμονής έρθει κάποια διακοπή, θα επιστρέψουμε *-ERESTARTSYS*, υποδυκνείοντας πως πρέπει να χειριστεί από τα ανώτερα στρώματα. Τέλος, προσπαθούμε άλλη μια φορά να κλειδώσουμε το σημαφόρο του state, αφού μπορεί κάποια άλλη διεργασία να έχει προλάβει να το κάνει. Αν το καταφέρουμε, προχωράμε στο διάβασμα των δεδομένων.

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

*f_pos += cnt;
ret = cnt;
if(state->buf_lim == *f_pos) *f_pos = 0;
out:
      up(&state->lock);
      return ret;
}
```

Αρχικά, ελέγχουμε ότι υπάρχουν διαθέσιμα cnt bytes προς ανάγνωση. Τα διαθέσιμα προς ανάγνωση bytes είναι *(buf_lim - *f_pos)*, όπου *buf_lim* ο συνολικός αριθμός bytes μέσα στον buffer. Αν η τιμή είναι μικρότερη από το cnt, θέτουμε *cnt = (size_t) state->buf_lim - *f_pos*, ώστε να διαβάσουμε το μέγιστο πλήθος bytes που μπορούμε. Ύστερα, μεταφέρουμε τα cnt σε πλήθος δεδομένα από τη θέση μνήμης *state->buf_data + *f_pos* στο χώρο χρήστη, μέσω της εντολής *copy_to_user(usrbuf, state->buf_data + *f_pos, cnt)*. Στη συνέχεια, αυξάνουμε το **f_pos* κατά cnt για να υποδείξουμε την θέση από την οποία θα αρχίσει η επόμενη μέτρηση. Αν φτάσουμε στο τέλος του buffer *(state->buf_lim == *f_pos)*, θέτουμε **f_pos = 0*, υποδεικνύοντας πως δεν υπάρχουν άλλα δεδομένα διαθέσιμα προς ανάγνωση. Τέλος, επιστρέφουμε το πλήθος των bytes που διαβάστηκαν.

```
int lunix_chrdev_init(void)
```

Η συνάρτηση αρχικοποίησης του οδηγού συσκευής. Εκτελείται με την κλήση της συνάρτησης insmod, η οποία εισάγει τον driver στον πυρήνα του λειτουργικού. Αρχικά, εκτελούμε την εντολή *cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops)*, η οποία αρχικοποιεί τη συσκευή χαρακτήρων και τη συνδέει με τη δομή file_operations *lunix_chrdev_fops*. Με την εντολή *dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0)* ορίζουμε την συσκευή με τον κατάλληλο major number και minor number = 0. Τέλος, με την εντολή *register_chrdev_region(dev_no, lunix_minor_cnt, "Lunix-Sensors"),* δεσμεύουμε τον κατάλληλο αριθμό device numbers για τον driver μας, ο οποίος είναι lunix_minor_cnt.

```
void lunix_chrdev_destroy(void)
```

Καλείται κάθε φορά που αφαιρούμε το module από τον πυρήνα του linux. Παίρνουμε τον κατάλληλο αριθμό συσκευής με την *dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0)* και με χρήση της εντολής *cdev_del(&lunix_chrdev_cdev)* αφαιρούμε τη συσκευή από τον πυρήνα. Τέλος, με την εντολή *unregister_chrdev_region(dev_no, lunix_minor_cnt)*, απελευθερώνουμε τους minor numbers που είχαμε κρατήσει κατά την αρχικοποίηση του module.