Αναφορά 2ης Άσκησης: Οδηγός Ασύρματου Δικτύου Αισθητήρων στο Λειτουργικό Σύστημα Linux

Ηλιακοπούλου Νικολέτα Μαρκέλα 03116111
Σαρτζετάκη Χριστίνα 03116193
Εργαστήριο Λειτουργικών Συστημάτων
Χειμερινό εξάμηνο 19-20

Εισαγωγή – Σκοπός άσκησης

Στα πλαίσια της εργαστηριακής άσκησης αυτής, κληθήκαμε να ολοκληρώσουμε την κατασκευή του Lunix:TNG, ενός απλού οδηγού συσκευής για το λειτουργικό σύστημα Linux. Έχοντας στην κατοχή μας το σύνολο των αρχείων lunix-tng-helpcode, κατανοήσαμε τη ροή της επεξεργασίας των πρωτογενών δεδομένων που λαμβάνουμε από τους αισθητήρες (line-discipline, protocol, sensor buffers), και επικεντρωθήκαμε στο επίπεδο εξαγωγής δεδομένων στο χώρο χρήστη υλοποιώντας μια σειρά από συσκευές χαρακτήρων.

Η συνάρτηση init

Στη συνάρτηση αυτή πρώτα αρχικοποιούμε τη συσκευή (cdev_init), συνδέοντάς την με τη δομή file_operations που περιέχει τις λειτουργίες που αναλαμβάνει να εκτελέσει ο οδηγός που υλοποιούμε (μέθοδοι της συσκευής: open, release, read, etc). Στη συνέχεια αναθέτουμε major number στη συσκευή μέσω της MKDEV και δεσμεύουμε εύρος από minor numbers μέσω της register_chrdev_region, ώστε τα ειδικά ονόματα των αρχείων της συσκευής να συνδέονται με τις παραπάνω μεθόδους. Τέλος, ενημερώνουμε τον πυρήνα για τη νέα συσκευή μέσω της cdev_add.

```
int lunix_chrdev_init(void)
{
	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, "LunixSensors");
	if (ret < 0) {
		debug("failed to register region, ret = %d\n", ret);
		goto out;
```

```
	}
	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;
}
```

Η συνάρτηση open

Αρχικά, μετά από χρήση της nonseekable_open για σειριακή ανάγνωση δεδομένων, θέλουμε να συσχετίσουμε το ανοικτό αρχείο με τον αντίστοιχο αισθητήρα παίρνοντας τον minor number από το inode struct του (iminor). Στη συνέχεια δεσμεύουμε χώρο στη μνήμη για το private state struct, αντίστοιχο με το μέγεθος ενός lunix_chrdev_state_struct, με κλήση της kzalloc, και αναθέτουμε τιμές στα πεδία του (στο sensor το struct που αντιστοιχεί στον αριθμό του αισθητήρα, στο type το είδος της μέτρησης, στο timestamp αρχική τιμή 0, στο lock (σημαφόρο) αρχική κατάσταση unlock). Τέλος, αποθηκεύουμε στο πεδίο private_data του file struct το παραπάνω state struct που κατασκευάσαμε.

```
static int lunix_chrdev_open(struct inode *inode, struct file *filp)
{
	struct lunix_chrdev_state_struct *state;
	int ret;
	unsigned int minor, type, sensor_no;

	debug("entering\n");
	ret = -ENODEV;
	if ((ret = nonseekable_open(inode, filp)) < 0)
		goto out;

	minor=iminor(inode);
	sensor_no=minor/8;
	type=minor%8;
```

```
	state = kzalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);
	state->type=type;
	state->sensor=&lunix_sensors[sensor_no];
	state->buf_timestamp=0;
	sema_init(&state->lock, 1);
	filp->private_data = state;
	ret=0;

out:
	debug("leaving, with ret = %d\n", ret);
	return ret;
}
```

Η συνάρτηση release

Εδώ αποδεσμεύουμε τη μνήμη που δεσμεύσαμε κατά το άνοιγμα (open) της συσκευής μέσω της kfree.

```
static int lunix_chrdev_release(struct inode *inode, struct file *filp)
{
	kfree(filp->private_data);
	return 0;
}
```

Η συνάρτηση read

Αρχικά, ανακτούμε το state struct από το private_data του file struct, και το sensor struct από το πεδίο sensor του state. Βάζουμε τον σημαφόρο σε κατάσταση lock (down_interruptible) και στο σημείο αυτό όποια άλλη διεργασία προσπαθήσει να αποκτήσει το κλείδωμα θα κοιμηθεί. Αν το διάβασμα έχει οριστεί να γίνει από την αρχή του αρχείου (*fpos == 0) δηλαδή να πάρει νέα μέτρηση, θα ελέγξουμε για ενημερώσεις στα δεδομένα. Όσο δεν υπάρχουν νέα δεδομένα (η update επιστρέφει - EAGAIN), η διεργασία πρέπει να κοιμηθεί εώς ότου αυτά καταφτάσουν. Αυτό γίνεται μέσω της wait_event_interruptible που βάζει τη διεργασία σε wait_queue, αφού έχουμε ξεκλειδώσει το σημαφόρο έχοντας πια φύγει από το κρίσιμο τμήμα. Όταν το δεύτερο όρισμα της κλήσης αυτής (refresh) γίνει αληθές, έχει κληθεί ο interrupt handler (lunix-sensors.c) και ξυπνάει η ουρά των διεργασιών που ήταν σε κατάσταση ύπνου. Όταν μια διεργασία ξυπνήσει, ξαναπαίρνει το κλείδωμα του σημαφόρου και ξαναψάχνει για νέα δεδομένα με την update.
Βγαίνοντας από το loop, έχει ανανεωθεί η μέτρηση στον state buffer (βλ. συνάρτηση update

παρακάτω) και είμαστε έτοιμοι να την αντιγράψουμε στον buffer του χώρου χρήστη. Πριν από αυτό είναι απαραίτητο να ελεγχθεί ότι τα *fpos και cnt που δόθηκαν είναι ορθά, δηλαδή, αν το *fpos είναι μεγαλύτερο από το limit του buffer να επιστρέφει η read 0, δηλαδή ότι δεν διάβασε τίποτα, και αν το cnt διαβάζοντας με αρχή το *fpos ξεπερνάει το limit, να τεθεί σε αυτό τιμή ίση με το μέγεθος που μπορεί να διαβάσει. Έτσι, μπορούμε να χρησιμοποιήσουμε την copy_to_user για ασφαλή αντιγραφή των δεδομένων μας στο χώρο χρήστη, και αν αυτό γίνει με επιτυχία η read θα επιστρέψει cnt, δηλαδή όσο διάβασε. Μετακινούμε το *fpos κατά cnt σε περίπτωση που η ίδια ή κάποια άλλη διεργασία θέλει να συνεχίσει να διαβάζει από εκεί που σταμάτησε η προηγούμενη ανάγνωση (αν δεν διάβασε μέχρι το τέλος). Τέλος υλοποιούμε auto-rewind mode, όπου όταν έχει διαβαστεί όλη η μέτρηση ο δείκτης μετακινείται στην αρχή του αρχείου για ανάγνωση νέας μέτρησης. Αμέσως πριν επιστρέψει, η read σηκώνει τον σημαφόρο, για να έχουν ξανά και οι άλλες διεργασίες πρόσβαση στα κρίσιμα τμήματα.

```
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);

      if(down_interruptible(&state->lock))
            return -ERESTARTSYS;
      if (*f_pos == 0) {
            while (lunix_chrdev_state_update(state) == -EAGAIN) {
                  up(&state->lock);
                  debug("going to sleep");
                  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) {
            ret = 0;
            goto out;
```

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

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

Η συνάρτηση refresh

Η συνάρτηση αυτή παίρνει μια boolean απόφαση ανάλογα με το αν έχει έρθει καινούργια μέτρηση (στην οποία περίπτωση το `sensor->msr_data[state->type]->last_update)` που ανανεώνεται μέσω interrupt ασύγχρονα είναι μεγαλύτερο από το `state->buf_timestamp` που ανανεώνουμε εμείς στην update). Έτσι λειτουργεί και ως συνθήκη αφύπνισης για την wait_event_interruptible της read.
Η refresh και η update είναι βοηθητικές μέθοδοι για τη read.

```
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;
	return 0;
}
```

Η συνάρτηση update

Εδώ ελέγχοντας αν έχει έρθει καινούργια μέτρηση (κλήση της refresh), αποφασίζουμε αν θα διαβάσουμε από το values των msr_data structs των sensors, ή αν θα επιστρέψουμε με – EAGAIN. Στην περίπτωση που χρειάζεται να διαβάσουμε, είναι απαραίτητο να χρησιμοποιήσουμε κλείδωμα spinlock για να προστατέψουμε τους sensor buffers καθώς η συνάρτηση που τους ανανεώνει τρέχει σε interrupt context (lunix_sensor_update στο lunix-sensors.c) όταν παραληφθούν δεδομένα. Χρησιμοποιούμε συγκεκριμένα την spin_lock_irq για να απενεργοποιήσουμε τις διακοπές και να τις δρομολογήσουμε, αφού ξεκλειδώσουμε το lock, ώστε να μην γίνει καν προσπάθεια για interrupt (που θα κατέληγε σε spin ούτος η αλλιώς λόγω του spinlock μου) και να σπαταληθεί επιπλέον χρόνος και πόροι της cpu σε busy waiting. Μέσα στο lock, αναθέτω σε μια τοπική μεταβλητή value το δεκαεξάμπιτο περιεχόμενο του values του msr_data struct, καθώς και ενημερώνω το timestamp με την πρόσφατη τιμή last update.
Βγαίνοντας από το κλείδωμα, θέλω να επεξεργαστώ τη μέτρηση που έλαβα ανάλογα με το είδος της. Σε αυτό θα με βοηθήσουν τα lookup tables του lunix-lookup.h που απεικονίζουν κάθε δεκαεξάμπιτο σε μία αναγνώσιμη τιμή. Αφού πάρω το ακέραιο και το δεκαδικό μέρος της, τα αποθηκεύω στον state buffer και ανανεώνω το limit του στο μέγεθος αυτής της μέτρησης. Η update επιστρέφει 0 κατά την επιτυχή ανανέωση.
Η refresh και η update είναι βοηθητικές μέθοδοι για τη read.

```
static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct
*state)
{
	struct lunix_sensor_struct *sensor;
	int final_value, value, akeraio, dekadiko;
	WARN_ON(!(sensor= state->sensor));

	debug("leaving\n");
	if(lunix_chrdev_state_needs_refresh(state)){
		spin_lock_irq(&sensor->lock);
		value = sensor->msr_data[state->type]->values[0];
		state->buf_timestamp = sensor->msr_data[state->type]-
>last_update;
		spin_unlock_irq(&sensor->lock);
	}
	else{
		goto out;
	}

	switch(state->type) {
		case BATT:
			final_value = lookup_voltage[value];
			break;
```

```
		case TEMP:
			final_value = lookup_temperature[value];
			break;
		case LIGHT:
			final_value = lookup_light[value];
			break;
		default:
			debug("lookup: sth went wrong");
			goto out;
	}

	akeraio = final_value/1000;
	dekadiko = final_value%1000;
	state->buf_lim = sprintf(state->buf_data ,"%d.%d\n",akeraio,dekadiko);

	return 0;
out:
	debug("leaving\n");
	return -EAGAIN;
}
```