ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Λειτουργικών Συστημάτων
Ακαδημαϊκή περίοδος 2014-2015

Άσκηση 1:
Οδηγός Ασύρματου Δικτύου Αισθητήρων στο λειτουργικό σύστημα Linux

Ομάδα a10:
Ρέτζος Ραφαήλ AM: 3110662
Μάραντος Χαράλαμπος AM: 3110794

Θα περιγράψουμε συνοπτικά τα βασικά βήματα και τα σημεία του οδηγού συσκευής για το ασύρματο δίκτυο αισθητήρων κάτω από το λειτουργικό σύστημα Linux που υλοποιήσαμε τα οποία θεωρούμε ότι πρέπει να αναφέρουμε:

Τα system calls και οι λειτουργίες που υλοποιήσαμε βρίσκονται στο αρχείο lunix-chrdev.c και αφορά την συσκευή χαρακτήρων:

● Αρχικά στην συνάρτηση lunix-chrdev-init αρικοποιούμε τις συσκευές και τον driver. κάνουμε **register** τις συσκευές μας ξεκινόντας από minor number dev_no και ζητώντας lunix_minor_cnt αριθμούς δηλαδή lunix_sensor_cnt << 3 όσοι είναι οι sensors δια 8 (8 sensors κάθε συσκευή) δηλαδή 16 συσκευές και δίνοντας όνομα “lunix_tng”. Έτσι ο πυρήνας ξέρει ότι ο driver αυτός είναι γι αυτές τις συσκευές:

```
    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;
     }
```

Επίσης στην συνέχεια **προσθέτουμε τις συσκευές μας** στην “λίστα” με τις **συσκευές χαρακτήρων** που διατηρεί ο πυρήνας:

```
    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;
     }
```

● Έπειτα κοιτάξαμε την open. Στο userspace θα “φαίνονται” οι σένσορες σαν “αρχεία” στον κατάλογο /dev/ και το πρώτο πράγμα που θα κάνει ένα userspace πρόγραμμα είναι η **κλήση συστήματος open**. Στην open φτιάχνουμε (για κάθε process που θα ανοίγει το αρχείο) **ένα “προσωπικό” private_state** του lunix_chrdev_state_struct. Έπειτα από το inode που στέλνει τις πληροφορίες ο πυρήνας παίρνουμε με την συνάρτηση iminor τον minor αριθμό που αντιστοιχεί στον συγκεκριμμένο sensor. Κάνουμε **allocate μνήμη** για το struct και δίνουμε τις πληροφορίες: Τον τύπο (**type**) του sensor που ισούται με τον minor αριθμό mod 8 (8 διαφορετικές μετρήσεις κάθε ένα), το ποιός **sensor** είναι στη θέση minor number διά 8 του lunix_sensors που αρχικοποιείται στο lunix.h και κάνουμε **αρχικοποίηση του σημαφόρου** που θα χρειαστεί όπως θα δούμε παρακάτω για συγχρονισμό μεταξύ διεργασιών με το ίδιο state (πχ fork). Τέλος όλα τα παραπάνω **τα περνάμε στο file pointer** που διαχειρίζεται ο πυρήνας και που θα στέλνεται στα system calls (read κλπ):

```
    struct lunix_chrdev_state_struct *private_state;
    ...
    private_state = kzalloc(sizeof(struct lunix_chrdev_state_struct),
GFP_KERNEL);
    if(!private_state) {
      ret = -ENOMEM; /*Error code 12: Out of memory*/
      printk(KERN_ERR "Memory allocation failed\n");
      goto out;
    }
    private_state->type = minor_num%8;
    private_state->sensor = &lunix_sensors[minor_num>>3];
    init_MUTEX(&(private_state->lock));
```

```
    filp->private_data = private_state;
```
(όπου `init_MUTEX(LOCK)` κάνουμε define το `sema_init (LOCK, 1))`

● Το επόμενο βήμα αφορά την **read**. Η read παίρνει ως **όρισμα τον file pointer** οπότε από αυτόν **παίρνουμε το state** της συγκεκριμένης διεργασίας που επιχειρεί το read μέσω του **filp->private_data**. Έπειτα από το **sensor** του struct αυτού παίρνουμε τον sensor για τον οποίο ζητούνται οι τιμές:

```
    state = filp->private_data;
    WARN_ON(!state);
    sensor = state->sensor;
    WARN_ON(!sensor);
```

Κατόπιν **κλειδώνουμε με χρήση του σημαφόρου του state**. Αυτός ο σημαφόρος αφορά **συγχρονισμό μεταξύ διεργασιών με το ίδιο state** (αντίστοιχα το ίδιο fd στην open). Ένα παράδειγμα τέτοιων διεργασιών είναι μεταξύ γονέων παιδιών (**fork**). Όταν το fork γίνεται μετά την open αυτές οι διεργασίες κληρονομούν τα ίδια χαρακτηριστικά (ίδιο fd σε userspace άρα ίδιο state σε kernelspace). Δεν αφορά διαφορετικές διεργασίες που κάνανε διαφορετικά open γιατί αυτές θα έχουν η κάθε μια το δικό της state.

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

Μετά το κλείδωμα ελέγχουμε το f_pos. Το **f_pos** δείχνει την **τρέχουσα θέση** στο αρχείο (πχ στο lunix1-temp). Αν είναι **0** πάει να πει ότι δεν υπολείπεται κάποια προηγούμενη τιμή αλλά **χρειαζόμαστε μια καινούρια**. Έτσι μπαίνουμε σε ένα **loop** όπου καλούμε την συνάρτηση **update** (θα την δούμε παρακάτω) η οποία θα πάρει τα δεδομένα μας. Όσο αυτή επιστρέφει **-EAGAIN** (“Try again”) άρα δεν έχουμε νέα δεδομένα συνεχίζουμε το loop ως εξής: Αν έχουμε από userspace καλέσει read **non_blocking** εφ όσον δεν υπάρχει νέα τιμή η read επιστρέφει **-EAGAIN** (“Try again”). Αν όχι τότε πρέπει να περιμένει να έρθει νέα τιμή. Αφού ξεκλειδώσει ο σημαφόρος, αυτό θα γίνεται με την συνάρτηση: **wait_event_interruptible(state->sensor->wq,lunix_chrdev_state_needs_refresh(state))** Όλες οι διεργασίες που περιμένουν (βρίσκονται στο **wq (wait queue)** στο sensor struct του συγκεκριμένου sensor θα περιμένουν μέχρι να έρθουν νέα δεδομένα. Όταν αυτά έρθουν θα κληθεί μέσω της **lunix_protocol_update_sensors** (στο lunix-protocol.c) η **lunix_sensor_update** (στο lunix-sensors.c) η οποία αφού περάσει τις νέες τιμές στους sensor buffers καλεί την **wake_up_interruptible(&s->wq)** που στέλνει σήμα να ξυπνήσουν όλες οι διεργασίες στην ουρά. Αυτές **όταν ξυπνήσουν ελέγχουν** με την συνάρτηση **lunix_chrdev_state_needs_refresh(state)** αν οι νέες τιμές ήρθαν (επιστρέφει 1 θα δούμε στο επόμενο βήμα πως) και αν ναι τότε συνεχίζουν. Έπειτα έχουμε και πάλι κλείδωμα επειδή **ΟΛΕΣ οι διεργασίες ξυπνάνε** και **θα τεθεί θέμα συγχρονισμού** σε αυτές με το ίδιο state **όποια “προλάβει” θα πάρει τον σημαφόρο**, update για να πάρει τιμές κλπ:

```
    if (*f_pos == 0) {
          printk("Needs a new value\n");
          while ((ret = lunix_chrdev_state_update(state)) == -EAGAIN) {
          up(&state->lock);
          if (filp->f_flags & O_NONBLOCK)
              return -EAGAIN;
          if(wait_event_interruptible(state->sensor>wq,lunix_
               chrdev_state_needs_refresh(state))){
```

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

Έπειτα ανάλογα με την τρέχουσα θέση του f_pos και το πόσοι χαρακτήρες ζητήθηκαν στην read (cnt) υπολογίζουμε πόσους χαρακτήρες θα “στείλουμε”. Αν χωράει ή όχι όλη η τρέχουσα τιμή. Το μήκος της είναι το state->buf_lim που ενημερώνουμε στην update για κάθε νέα τιμή (παρακάτω βήμα).

```
      if(*f_pos + cnt < state->buf_lim) {
            cnt = state->buf_lim - cnt - *f_pos;
      } else {
        cnt = state->buf_lim - *f_pos;
      }
```

Για να στείλουμε τα δεδομένα στον χρήστη χρησιμοποιούμε την **copy_to_user** για λόγους **ασφαλείας**. Επειδή ο χρήστης και ο πυρήνας **δεν “βλέπουν” την ίδια διευθυνσιοδότηση στην μνήμη** (καθώς προφανώς ο χρήστης δεν θα πρέπει να μπορεί να γράψει πχ πάνω στον πυρήνα) και για να μπορεί να ελέγχει ο πυρήνας τους δείκτες του χρήστη (πχ μπορεί να υπάρχει πρόβλημα segmentation fault).

```
      if(copy_to_user(usrbuf, state->buf_data, cnt)){
            printk("Copy_to_user failed \n");
            ret = -EFAULT;
            goto out;
      }
      ret = cnt;
```

Αφού τα cnt δεδομένα σταλούν **ανανεώνουμε** την τρέχουσα θέση στο αρχείο **f_pos** κι αν γράψαμε όλη την τιμή κάνουμε το f_pos=0 ώστε σε επόμενη read να γίνει update. Τέλος ξεκλειδώνουμε και επιστρέφουμε το πόσα στοιχεία στείλαμε (πόσα στοιχεία διαβάζει ο χρήστης) αλλιώς κάποιο σφάλμα αν υπάρχει:

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

● Στην **lunix_chrdev_state_needs_refresh ελέγχουμε** απλά αν το state->buf_timestamp στο οποίο στην update αποθηκεύουμε την **χρονική στιγμή** που έρχεται η τελευταία τιμή είναι μικρότερο από το sensor->msr_data[state->type]->**last_update** στο οποίο στην lunix_sensor_update του lunix-sensors.c αποθηκεύεται η χρονική στιγμή που έρχονται τα δεδομένα και αν είναι σημαίνει ότι **ήρθαν νέα δεδομένα** άρα επιστρέφει 1 αλλιώς 0:

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

● Στην update αρχικοποιούμε ότι μεταβλητές χρειάζονται για να αποθηκεύσουμε τις τιμές μας και έπειτα κάνουμε **spin lock** για να πάρουμε τιμές. Θα χρησιμοποιήσουμε την **spin_lock_irqsave**(&sensor->lock,flags) που **απενεργοποιεί τις διακοπές διατηρώντας ένα flag για την ανάκτησή τους**. Γιατί χρησιμοποιούμε spinlocks και γιατί αυτο: Όταν η συσκευή έχει νέα δεδομένα κάνει **διακοπή** και ο **handler** πάει στην **update** στο lunix-protocols.c όπου παίρνει τα δεδομένα και τα βάζει μέσω της update στο lunix-sensors.c στους sensor buffers. Μέσα στην συνάρτηση εκείνη έχει επίσης **spinlock (interrupt context)**. Εμείς (ο user) κάνουμε system call read καλείται οι update που θέλει να διαβάσει τους sensor buffers **(process context)**. Εδώ υπάρχει θέμα **συγχρονισμού**. Τα spinlocks είναι **busy-waits** όπου ελέγχει συνεχώς αν ο άλλος το άφησε και αυτό γίνεται γιατί δεν μπορεί ο πυρήνας να κάνει sleep την συσκευή καθώς η συσκευή δεν είναι διεργασία! Εμείς όταν παίρνουμε αυτό το spinlock **απενεργοποιούμε τις διακοπές** γιατί αν το χαμε πάρει και εκείνη την ώρα γινόταν διακοπές ο πυρήνας στον handler θα αναλάμβανε και θα κόλλαγε αιώνια στο spinlock!!! Η irqsave διατηρεί την κατάσταση των flags σε περίπτωση που είχαν απενεργοποιηθεί οι διακοπές για άλλο λόγο να μην τις ενεργοποιήσει έτσι κι αλλιώς μετά αλλά να επαναφερει απλα'την προηγούμενη κατάσταση όποια κι αν ήταν:

```
    struct lunix_sensor_struct *sensor;
    uint32_t time_stamp;
    uint32_t magic_no;
    uint32_t val;
    long lookup_value;
    int buf_pos = 0;
    int num;
    unsigned long flags;
    sensor = state->sensor;
    spin_lock_irqsave(&sensor->lock,flags);
    magic_no = sensor->msr_data[state->type]->magic;
    time_stamp = sensor->msr_data[state->type]->last_update;
    val = sensor->msr_data[state->type]->values[0];
    spin_unlock_irqrestore(&sensor->lock,flags);
```

Έπειτα ελέγχουμε την περίπτωση σφαλμάτων αν δεν διαβάσαμε το magic (-EFAULT) ή αν δεν έχουμε νέα τιμή (-EAGAIN) αλλιώς παίρνουμε τις τιμές μας από τους πίνακες **lookup** από το lunix-lookup.h και **“σπάμε” τον αριθμό** σε **ψηφίο ψηφίο** που αποθηκεύουμε ως **char** (ascii num '0' + το ψηφίο) σε κάθε θέση του buffer (buf_data) στο state. Τέλος περνάμε το πλήθος chars της τιμής στο **buf_lim** του state και τον χρόνο που ήρθε η τιμή στο **buf_timestamp**:

```
    if(magic_no != LUNIX_MSR_MAGIC || val >=65536){
            return -EFAULT;
    }
    if( time_stamp <= state->buf_timestamp){
            return -EAGAIN;
    }
    if (state->type == BATT)
            lookup_value = lookup_voltage[val];
```

```
    else if (state->type == TEMP)
          lookup_value = lookup_temperature[val];
    else if (state->type == LIGHT)
          lookup_value = lookup_light[val];
    else
          return -EFAULT;
    if(lookup_value < 0) {
          lookup_value *= -1;
          state->buf_data[0] = '-';
          buf_pos = 1;
    }
    num = lookup_value/1000;
    if (num/10 != 0)
               state->buf_data[buf_pos++] = '0' + num/10;
    state->buf_data[buf_pos++] = '0' + num%10;
    state->buf_data[buf_pos++] = '.';
    num = lookup_value%1000;
    state->buf_data[buf_pos++] = '0' + num/100;
    if (num%10 == 0){
          if ((num/10)%10 != 0)
                state->buf_data[buf_pos++] = '0' + (num/10)%10;
    }
    else {
      state->buf_data[buf_pos++] = '0' + (num/10)%10;
      state->buf_data[buf_pos++] = '0' + num%10;
    }
    state->buf_data[buf_pos++] = '\n';
    state->buf_lim = buf_pos;
    state->buf_timestamp = time_stamp;
    return 0;
```