# Εργαστήριο Λειτουργικών Συστημάτων

## 2η εργαστηριακή αναφορά

Νικόλαος Παγώνας
Νικήτας Τσίννας

Αντικείμενο της παρούσας εργαστηριακής άσκησης είναι η υλοποίηση ενός οδηγού συσκευής για ένα ασύρματο δίκτυο αισθητήρων κάτω από το λειτουργικό σύστημα Linux. Ακολουθεί η υλοποίηση των μεθόδων που βρίσκονται στο αρχείο `lunix-chrdev.c`.

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

Η συνάρτηση αυτή χρησιμοποιείται ώστε να ελέγξουμε αν υπάρχει νέα διαθέσιμη μέτρηση για κάποιο ανοιχτό αρχείο που περιμένει δεδομένα από έναν αισθητήρα. Θα έχουν έρθει νέα δεδομένα όταν ισχύει η συνθήκη:

`state->buf_timestamp != sensor->msr_data[state->type]->last_update`

Συγκεκριμένα, συγκρίνουμε την `buf_timestamp` του ανοιχτού αρχείου με την `last_update` της μέτρησης του αισθητήρα που θέλουμε. Αν αυτές οι τιμές είναι ίσες, τότε η τελευταία μέτρηση έχει ήδη ληφθεί, αλλιώς χρειάζεται να γίνει update της μέτρησης.

```
1  static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
2  {
3    struct lunix_sensor_struct *sensor;
4
5    WARN_ON(!(sensor = state->sensor));
6
7    if(state->buf_timestamp != sensor->msr_data[state->type]->last_update)
8      return 1;
9
10   return 0;
11 }
```

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

Η συνάρτηση αυτή καλείται κάθε φορά που μία διεργασία χρειάζεται να διαβάσει δεδομένα από το αρχείο συσκευής αλλά εκείνης το `*f_pos` (file position) είναι 0.

Το πρώτο βήμα που εκτελείται είναι να κλειδωθεί το spinlock του αισθητήρα που αντιστοιχεί στο αρχείο συσκευής με την εντολή `spin_lock()` (γραμμή 17). Είναι σημαντικό να χρησιμοποιήσουμε spinlocks καθώς αν μία διεργασία κοιμηθεί ενώ βρίσκεται στο κρίσιμο τμήμα, τότε στο προβλεπόμενο μέλλον το κλείδωμα θα παραμένει κλειστό και αν κάποια άλλη διεργασία θέλει να αποκτήσει πρόσβαση θα πρέπει να περιμένει αρκετό χρόνο. Αυτό μπορεί να οδηγήσει στην χειρότερη περίπτωση σε deadlock του συστήματος.

Τα spinlocks αποτελούν λύση σε αυτό το πρόβλημα γιατί απαγορεύουν σε οποιαδήποτε άλλη διεργασία να χρησιμοποιήσει τον επεξεργαστή, όσο η τρέχουσα διεργασία βρίσκεται στο κρίσιμο σημείο. Το kernel preemption ελέγχεται από τον κώδικα του ίδιου του spinlock, που σημαίνει πως ο kernel απενεργοποιεί το preemption του εκάστοτε επεξεργαστή κάθε φορά που ”κρατάει” κάποιο spinlock.

Σε συνέχεια της περιγραφής της μεθόδου, όταν η διεργασία καταφέρει να μπει στο κρίσιμο σημείο, πρώτα αποθηκεύει στην `data` τα νέα δεδομένα από την συσκευή και κρατάει στην `timestamp` πότε έγινε η ενημέρωση του αισθητήρα. Αμέσως μετά ελέγχει εάν υπάρχουν καινούργια δεδομένα (αν δεν υπάρχουν καινούργια δεδομένα επιστρέφει `-EAGAIN`).

Αν υπάρχουν καινούργια δεδομένα, ακολουθεί την παρακάτω διαδικασία ώστε με σωστό τρόπο να τα αποθηκεύσει στον buffer:

1. Πρώτα, αναλόγως με το είδος της μέτρησης που έχει λάβει, μεταφράζει τα raw bytes δεδομένων σε τιμές μέσω των πινάκων `lookup_{voltage, temperature, light}` που μας έχουν δοθεί (αρχείο `lunix-lookup.h`).
2. Βρίσκουμε το ακέραιο και το δεκαδικό μέρος της τιμής που μας δόθηκε από την lookup κάνοντας τις αντίστοιχες πράξεις στις γραμμές 50 και 51 όπως φαίνεται παρακάτω στον κώδικα.
3. Τέλος, γράφουμε με την snprintf τη μέτρηση στον buffer του struct state και έπειτα ενημερώνουμε και το timestamp αυτού.

```
1  static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
2  {
3    struct lunix_sensor_struct *sensor;
4
5    WARN_ON(!(sensor = state->sensor));
6
7
8    int type = state->type;
9    uint32_t data;
10   uint32_t timestamp;
11
12   /*
13    * Grab the raw data quickly, hold the
14    * spinlock for as little as possible.
15    */
16
17   spin_lock(&sensor->lock);
18
19   data = sensor->msr_data[type]->values[0];
20   timestamp = sensor->msr_data[type]->last_update;
21
22   spin_unlock(&sensor->lock);
23
24   /*
25    * Any new data available?
26    */
27
28   if(state->buf_timestamp == timestamp)
29     return -EAGAIN;
30
31   /*
32    * Now we can take our time to format them,
33    * holding only the private state semaphore
34    */
```

```
35
36    long lookup_value;
37
38    switch(type) {
39      case BATT:
40        lookup_value = lookup_voltage[data];
41        break;
42      case TEMP:
43        lookup_value = lookup_temperature[data];
44        break;
45      case LIGHT:
46        lookup_value = lookup_light[data];
47        break;
48    }
49
50    int integer_part = lookup_value / 1000;
51    int decimal_part = lookup_value > 0 ? lookup_value % 1000 : -lookup_value % 1000;
52
53    state->buf_lim = snprintf(state->buf_data, LUNIX_CHRDEV_BUFSZ, "%d.%d\n", integer_part,
       decimal_part);
54    state->buf_timestamp = timestamp;
55
56    return 0;
57  }
```

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

Η μέθοδος `lunix_chrdev_open` χρησιμοποιείται για να ανοίξουμε ένα ειδικό αρχείο συσκευής lunix και είναι η πρώτη μέθοδος που εκτελείται σε έναν οδηγό. Στις περισσότερες περιπτώσεις η open() είναι υπεύθυνη για τις παρακάτω λειτουργίες.

1. Να ελέγξει για hardware errors (πχ η συσκευή δεν είναι έτοιμη για άνοιγμα)
2. Να αρχικοποιήσει τη συσκευή
3. Να ενημερώσει τον `f_op` pointer (αν είναι απαραίτητο)
4. Να αντιστοιχίσει κάποια δομή δεδομένων με το πεδίο `private_data` του `filp`

Θα μπορούσαμε να παραλείψουμε την υλοποίηση της μεθόδου αλλά σε εκείνη τη περίπτωση δεν θα ενημερωνόταν ο οδηγός για το άνοιγμα του αρχείου. Εμείς θέλουμε να αρχικοποιούμε την δομή state κάθε συσκευής μαζί με το άνοιγμα του αρχείου και να την συνδέουμε με το αρχείο συσκευής (4) και για αυτό την υλοποιούμε.

Με την κλήση `nonseekable_open` ο kernel ανοίγει το αρχείο συσκευής και ταυτόχρονα ενημερώνεται πως το αρχείο είναι nonseekable, δηλαδή θα πρέπει να απαγορεύει lseek μεθόδους πάνω στο αρχείο αυτό. Αν γυρίσει αρνητική τιμή τότε το άνοιγμα της συσκευής δεν ήταν επιτυχημένο και άρα θα επιστρέψουμε την τιμή `-ENODEV` δηλώνοντας έτσι πως η συσκευή δεν είναι συνδεδεμένη στο σύστημα.

Με την χρήση της μακροεντολής iminor μπορούμε να ανακτήσουμε τον minor number του αρχείου που θα μας επιτρέψει να το αντιστοιχίσουμε με τον σωστό αισθητήρα. Έτσι, αρχικοποιούμε τα πεδία του state ως εξής:

- type: Αποτελεί τον τύπο της μέτρησης (θερμοκρασία, φως ή μπαταρία). Επειδή υπάρχει χώρος για το πολύ 8 μετρήσεις, κρατάμε τα τρία τελευταία bits του minor number, ο οποίος προκύπτει ως minor $=$ $8 *$ αισθητήρας $+$ μέτρηση.

- sensor: αποθηκεύει τον αριθμό του αισθητήρα που αντιστοιχεί στην συσκευή.
- `buf_lim`: Το μέγεθος της μέτρησης. Αρχικοποιείται στην τιμή 0.
- `buf_data`: Το buffer όπου αποθηκεύονται οι μετρήσεις. Αρχικοποιείται με '\0'
- `buf_timestamp`: Το timestamp της πιο πρόσφατης μέτρησης που έχει λάβει ο buffer. Το αρχικοποιούμε με 0, ώστε η πρώτη κλήση της `lunix_state_needs_refresh` να επιστρέφει πάντα 1.

Επίσης αρχικοποιούμε τον σημαφόρο της δομής του state στην τιμή 1 (ξεκλείδωτος) ώστε να είναι έτοιμος για χρήση. Αυτό το κλείδωμα το χρειαζόμαστε για διεργασίες που μοιράζονται κοινούς file descriptors (κυρίως σε σχέση διεργασιών γονέα-παιδιού), αφού η κάθε διεργασία δημιουργεί το δικό της filp struct. Τέλος, αποθηκεύουμε το state που μόλις αρχικοποιήσαμε στο πεδίο `private_data` του filp ώστε να έχουμε εύκολη πρόσβαση σε αυτό μέσω του ανοίγματος του αρχείου συσκευής.

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

  int ret;

  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>]
   */

  unsigned int minor = iminor(inode);

  /* Allocate a new Lunix character device private state structure */

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

  /* Fill all fields of lunix_chrdev_state_struct */
  state->type = minor & 7; /* ...xxx & ...111, 3 LSB */
  state->sensor = &lunix_sensors[minor >> 3]; /* xxx... --> xxx, we "forget" 3 LSB */
  state->buf_lim = 0;
  state->buf_data[0] = '\0';
  state->buf_timestamp = 0;
  sema_init(&state->lock, 1);

  filp->private_data = state;
out:
  return ret;
}
```

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

Ο ρόλος της μεθόδου `release()` είναι ο αντίθετος της `open()`. Καλείται κάθε φορά που κλείνει ένα ανοιχτό αρχείο. Αξίζει να σημειωθεί ότι δεν καλείται σε κάθε invocation της `close()`, αφού μπορεί να υπάρχουν πολλοί

file descriptors που αναφέρονται στο ίδιο αρχείο.
Έτσι, όταν καλείται η `release()` σημαίνει πως δεν χρειαζόμαστε πλέον τα δεδομένα και την κατάσταση του αρχείου. Επομένως, απελευθερώνουμε τους δεσμευμένους πόρους μνήμης στον χώρο του πυρήνα που σχετίζονται με το `private_data` του file pointer.

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

```
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg);
```

Κανονικά, η μέθοδος `ioctl()` χρησιμοποιείται για να υλοποιήσουμε ειδικές εντολές συσκευής οι οποίες δεν αφορούν reading ή writing. Επίσης οι εντολές που υλοποιούνται σε αυτή τη μέθοδο δεν αναγνωρίζονται απαραίτητα όλες μέσω του `f_ops` table από τον kernel. Επειδή δεν την υλοποιούμε, ουσιαστικά απλώς επιστρέφουμε `-EINVAL`.

```
static long lunix_chrdev_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    /* Technically not implemented */
  return -EINVAL;
}
```

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

Η συνάρτηση αυτή καλείται κάθε φορά που κάποια διεργασία θέλει να διαβάσει έναν αριθμό bytes από κάποιο ανοιχτό αρχείο.

Αρχικά χρησιμοποιούμε τον δείκτη `private_data` για να αποκτήσουμε πρόσβαση στο `state`, δηλαδή τις πληροφορίες που αφορούν το ανοιχτό αρχείο.

Ύστερα κλειδώνουμε τον σημαφόρο της state (δηλαδή του ανοιχτού αρχείου), προκειμένου να μην μπορεί κάποιος άλλος (πχ κάποια διεργασία παιδί) να διαβάσει την ίδια στιγμή από το ανοιχτό αρχείο.

Κατόπιν ελέγχουμε αν `*f_pos == 0`, δηλαδή αν δεν υπάρχουν ακόμα διαθέσιμα δεδομένα στο ανοιχτό αρχείο. Αν η συνθήκη είναι αληθής, τότε η διεργασία πρέπει να περιμένει για να λάβει καινούργια μέτρηση. Γι' αυτό το σκοπό, καλείται η συνάρτηση `lunix_chrdev_state_update(state)`. Αν επιστρέψει `-EAGAIN` (δεν υπάρχουν νέα δεδομένα), η διεργασία πηγαίνει για ύπνο -αφού προηγουμένως ξεκλειδώσει τον σημαφόρο του ανοιχτού αρχείου-, με χρήση της

`wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state))`.

Η κλήση αυτή θα βάλει την διεργασία στην ουρά αναμονής του αισθητήρα από τον οποίο αναμένεται η αντίστοιχη μέτρηση.

Η διεργασία θα ξυπνήσει κάποια στιγμή από τον αισθητήρα (με κλήση της `wake_up_interruptible($s->wq)`, αρχείο `lunix-sensors.c`) και θα ελεγχθεί η συνθήκη `lunix_chrdev_state_needs_refresh(state)`.

Στη συνέχεια, κλειδώνεται ο σημαφόρος του ανοιχτού αρχείου και γίνεται ξανά κλήση της συνάρτησης `lunix_chrdev_state_needs_refresh(state)`. Αυτό επαναλαμβάνεται μέχρι η συνάρτηση αυτή να επιστρέψει ότι υπάρχουν καινούργια δεδομένα.

Μόλις συμβεί αυτό, υπολογίζουμε τον αριθμό bytes που θα αντιγράψουμε στον `usrbuf`. Αυτό γίνεται μέσω της `copy_to_user(usrbuf, state->buf_data, cnt)`, η οποία εξασφαλίζει την ασφαλή μεταφορά δεδομένων στο user space. Χρησιμοποιείται:

- για να αποφευχθούν τυχόν page faults (δεν επιτρέπονται στον πυρήνα, έχουν ως αποτέλεσμα τον θάνατο της διεργασίας)
- επειδή ο δείκτης του χώρου χρήστη μπορεί να μην είναι έγκυρος όταν τρέχουμε σε χώρο πυρήνα
- για λόγους ασφαλείας.

Ύστερα αυξάνουμε την τιμή του `*f_pos` κατά τον αριθμό bytes που καταφέραμε να διαβάσουμε. Αν φτάσαμε στο τέλος της μέτρησης (EOF mode, `*f_pos == state->buf_lim`) τότε κάνουμε rewind (θέτουμε `*f_pos = 0`), οπότε ερχόμαστε στην αρχική κατάσταση. Τέλος, αφήνουμε το κλείδωμα που μέχρι τώρα είχαμε, ώστε να μπορούν να το χρησιμοποιήσουν άλλες διεργασίες και επιστρέφουμε τον αριθμό bytes που διαβάστηκαν επιτυχώς.

```
1  static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt,
       loff_t *f_pos)
2  {
3    ssize_t ret;
4
5    struct lunix_sensor_struct *sensor;
6    struct lunix_chrdev_state_struct *state;
7
8    state = filp->private_data;
9    WARN_ON(!state);
10
11   sensor = state->sensor;
12   WARN_ON(!sensor);
13
14   /* Lock */
15   if (down_interruptible(&state->lock))
16     return -ERESTARTSYS;
17
18   /*
19    * If the cached character device state needs to be
20    * updated by actual sensor data (i.e. we need to report
21    * on a "fresh" measurement, do so
22    */
23   if (*f_pos == 0) {
24     while (lunix_chrdev_state_update(state) == -EAGAIN) {
25       /* The process needs to sleep */
26       up(&state->lock);
27
28       if (wait_event_interruptible(sensor->wq, lunix_chrdev_state_needs_refresh(state)))
29         return -ERESTARTSYS;
30
31       if (down_interruptible(&state->lock))
32         return -ERESTARTSYS;
33     }
34   }
35
```

```
  /* Determine the number of cached bytes to copy to userspace */
  int bytes_left = state->buf_lim - *f_pos;

  cnt = min(cnt, bytes_left);

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

  *f_pos += cnt;
  ret = cnt;

  /* Auto-rewind on EOF mode */
  if (*f_pos == state->buf_lim)
    *f_pos = 0;

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

```
static int lunix_chrdev_mmap(struct file *filp, struct vm_area_struct *vma);
```

Στην `lunix_chrdev_mmap` απλώς επιστρέφουμε `-EINVAL` διότι πρακτικά δεν την υλοποιούμε.

```
static int lunix_chrdev_mmap(struct file *filp, struct vm_area_struct *vma)
{
    /* Technically not implemented */
  return -EINVAL;
}
```

```
int lunix_chrdev_init(void);
```

Η συνάρτηση αυτή καλείται όταν εισάγουμε τον driver με την εντολή `insmod`.

Η `cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops)` αρχικοποιεί το char device και το συνδέει με τη δομή `file_operations lunix_chrdev_fops`, η οποία ως γνωστόν κρατάει τις υποστηριζόμενες λειτουργίες.

Με την εντολή `dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0)` δημιουργούμε έναν device number με major = 60 και minor = 0. Αυτός ο `dev_no` θα αποτελέσει την αρχή του range από device numbers που θα δεσμεύσουμε μέσω της εντολής `register_chrdev_region(dev_no, lunix_minor_cnt, "lunix")`.

Τέλος, με την εντολή `cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt)` προσθέτουμε στον πυρήνα 128 (= `lunix_minor_cnt`) device numbers, τα οποία αντιστοιχούν στο char device `lunix_chrdev_cdev`.

```
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;

  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");
  if (ret < 0) {
      /* failed to register region */
    goto out;
  }

  /* cdev_add */
  ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
  if (ret < 0) {
    /* failed to add character device */
    goto out_with_chrdev_region;
  }

  /* completed successfully */
  return 0;

out_with_chrdev_region:
  unregister_chrdev_region(dev_no, lunix_minor_cnt);
out:
  return ret;
}
```

```
void lunix_chrdev_destroy(void);
```

Η εντολή αυτή εκτελείται ώστε να αφαιρεθεί ο driver από τον πυρήνα με την εντολή `rmmod` (remove module).

Πρώτα υπολογίζουμε το μεγαλύτερο minor number που έχουμε δεσμεύσει (είναι το σύνολο των αισθητήρων * 8). Έπειτα με την μεταβλητή dev_no αντιστοιχίζουμε την διάδα αριθμών major-minor του πρώτου αρχείου που είχε δεσμεύσει η συσκευή χαρακτήρων κατά την αρχικοποίησή της. Αυτές τις δύο μεταβλητές τις χρειαζόμαστε για να ορίσουμε το εύρος των device numbers που θέλουμε να απελευθερώσουμε στην τελευταία εντολή.

Με την εντολή `cdev_del(lunix_chrdev_cdev)` αφαιρούμε την συσκευή χαρακτήρων από τον πυρήνα, ενώ με την εντολή `unregister_chrdev_region(dev_no, lunix_minor_cnt)` απελευθερώνουμε τους device numbers που είχαμε αρχικοποιήσει.

```
void lunix_chrdev_destroy(void)
{
  dev_t dev_no;
  unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;

  dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);

  cdev_del(&lunix_chrdev_cdev);
  unregister_chrdev_region(dev_no, lunix_minor_cnt);
}
```