# Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά στη δεύτερη εργαστηριακή άσκηση:
Lunix:TNG


**Ομάδα oslab 37**

**Παπαδόπουλος Κωνσταντίνος, Α.Μ.: 03120152**

**Μέη Αρετή, Α.Μ.:03120062**

# Εισαγωγή:

Στην παρούσα αναφορά, θα αναλυθεί ο κώδικας που μας ζητήθηκε να συμπληρώσουμε στο αρχείο chrdev.c, προκειμένου να καταστεί λειτουργικός ο οδηγός συσκευής χαρακτήρων Lunix:TNG.

Αρχικά αναφέρουμε πως το cdev είναι μια δομή του Kernel(περιέχεται στο inode), που αξιοποιείται για την αναπαράσταση συσκευών χαρακτήρων. Χρησιμοποιείται προκειμένου να γίνουν προσβάσιμες από τον πυρήνα οι κλήσεις που ορίζονται στο file_operations struct. Από την άλλη, το struct file_operations ορίζει τους pointers που πραγματοποιούν την σύνδεση μεταξύ των system calls που πραγματοποιεί ο χρήστης στο userspace(π.χ. read(), open()) και στις διαφορετικές κλήσεις του character device(π.χ. lunix_chrdev_open(), lunix_chrdev_read()), των οποίων τον κώδικα καλούμαστε να συμπληρώσουμε.

# Κώδικας

```
247 static struct file_operations lunix_chrdev_fops =
248 {
249         .owner                  = THIS_MODULE,
250         .open           = lunix_chrdev_open,
251         .release        = lunix_chrdev_release,
252         .read           = lunix_chrdev_read,
253         .unlocked_ioctl = lunix_chrdev_ioctl,
254         .mmap           = lunix_chrdev_mmap
255 };
```

Σε αυτό το struct βλέπουμε ποιες συναρτήσεις του οδηγού έχουμε να κατασκευάσουμε προκειμένου να λειτουργεί σωστά και σύμφωνα με τις πρακτικές.

**int lunix_chrdev_init(void):**

Ξεκινάμε με την συνάρτηση αρχικοποίησης (initialization)

```
257 int lunix_chrdev_init(void)
258 {
259         /*
260         * Register the character device with the kernel, asking for
261         * a range of minor numbers (number of sensors * 8 measurements / sensor)
262         * beginning with LINUX_CHRDEV_MAJOR:0
263         */
264         int ret;
265         dev_t dev_no;
266         unsigned int lunix_minor_cnt = lunix_sensor_cnt << 3;
267
268         debug("Startring initialization\n");
269
270         cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops);
271
272
273         lunix_chrdev_cdev.owner = THIS_MODULE;
274
275
276         // Create a device number using the major and minor numbers
277         dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0);
278
279         // Reserving a range of character device numbers
280         ret = register_chrdev_region(dev_no, lunix_minor_cnt, "lunix");
281         if (ret < 0) {
282           debug("failed to register region, ret = %d\n", ret);
283           goto out;
284         }
285
286         ret = cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt);
287         if (ret < 0) {
288           debug("failed to add character device\n");
289           goto out_with_chrdev_region;
290         }
291
292         debug("Init completed successfully\n");
293         return 0;
```

Αρχικά ορίζοντας ως lunix_minor_cnt ένα range για τα minor numbers, ενώ το είδος της κάθε μέτρησης με βάση την εκφώνηση θα καθορίζεται από τα 3 LSBs, οπότε και θα έχουμε:

lunix_minor_cnt = lunix_sensor_cnt << 3.

Στη συνέχεια, καλώντας την cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops), αρχικοποιούμε τη συσκευή χαρακτήρων και πραγματοποιούμε την σύνδεση μεταξύ του global struct μας cdev και του struct file_operations(δηλαδή στο πεδίο const struct file_operations *ops περνιούνται οι lunix_chrdev_fops).

Ορίζουμε επίσης το data type dev_no, με minor_number=0 και major_number = 60(έχει γίνει define σε διαφορετικό σημείο του κώδικα), ο οποίος σηματοδοτεί την

έναρξη του range των character device numbers, με βάση το range των minor numbers που ορίσαμε παραπάνω.

Έπειτα, καλώντας την register_chrdev_region(dev_no, lunix_minor_cnt, "lunix"), δεσμεύουμε την περιοχή αυτή. Αν αυτή η κλήση αποτύχει, βλέπουμε μήνυμα σφάλματος.

Με την cdev_add(&lunix_chrdev_cdev, dev_no, lunix_minor_cnt), συνδέουμε αυτή την περιοχή των character device numbers με το cdev μας, κάνοντάς τον έτσι προσβάσιμο μέσω των συγκεκριμένων minor numbers και του major number. Αν αυτό αποτύχει έχουμε και πάλι μήνυμα σφάλματος.

### int lunix_chrdev_open(struct inode *inode, struct file *filp):

Έπειτα η κλήση συστήματος που ανοίγει ένα οδηγό, όπως ανοιίγουμε κάποιο αρχείο (open).

```
116 static int lunix_chrdev_open(struct inode *inode, struct file *filp)
117 {
118         struct lunix_chrdev_state_struct *chrdev_state;
119         unsigned int sensor, type;
120         int ret;
121
122         //debug("entering open: %d\n", sensor);
123         ret = -ENODEV;
124
125         /* Write operation is not permitted */
126         if ((filp->f_flags & O_WRONLY) || (filp->f_flags & O_RDWR))
127           return -EPERM;
128
129         /* Device does not support seeking */
130         if ((ret = nonseekable_open(inode, filp)) < 0)
131           goto out;
132         /*
133         * Associate this open file with the relevant sensor based on
134         * the minor number of the device node [/dev/sensor<NO>-<TYPE>]
135         */
136         sensor = iminor(inode) / 8;
137         type = iminor(inode) % 8;
138
139         /* Allocate a new Lunix character device private state structure */
140         if (!(chrdev_state = kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL))) {
141           printk(KERN_ERR "Failed to allocate memory for character device private state\n");
142           ret = -ENOMEM;
143           goto out;
144         }
145
146         /* Initialization of the device private state structure */
147         chrdev_state->type = type;
148         chrdev_state->sensor = &lunix_sensors[sensor];
149         chrdev_state->buf_lim = 0;
150         chrdev_state->buf_timestamp = 0;
151         sema_init(&chrdev_state->lock, 1);
152
153         /* Save the device state */
154         filp->private_data = chrdev_state;
155         debug("character device state initialized successfully\n");
156
157
158
159         ret = 0;
160 out:
161         debug("leaving, with ret = %d\n", ret);
162         return ret;
163 }
```

Αρχικά, παρατηρούμε τις 2 παραμέτρους της συγκεκριμένης κλήσης:

- struct inode *inode: Είναι ένα inode object το οποίο είναι συνδεδεμένο με το file το οποίο επιχειρούμε να ανοίξουμε και περιέχει πληροφορίες για αυτό, όπως τα major και minor numbers.
- struct file *filp : Είναι ένας pointer στη δομή του Kernel file, και πάλι περιέχει πληροφορίες για το αρχείο που επιχειρούμε να ανοίξουμε, όπως τα file_operations, καθώς και τον pointer private_data.

Έπειτα ορίζουμε το chrdev_state, το οποίο είναι μια μεταβλητή στην οποία θα αποθηκεύονται πληροφορίες για τους αισθητήρες και τα δεδομένα που παίρνουμε από αυτούς για κάθε ξεχωριστό είδος μέτρησης.

Με το if statement:

```
if ((filp->f_flags & O_WRONLY) || (filp->f_flags & O_RDWR))
    return -EPERM;
```

, βεβαιωνόμαστε πως απαγορεύεται οποιαδήποτε προσπάθεια να γράψουμε πάνω στο file, ενώ με το:

```
/* Device does not support seeking */
if ((ret = nonseekable_open(inode, filp)) < 0)
    goto out;
```

, βεβαιωνόμαστε πως η συσκευή μας δεν είναι seekable, δηλαδή ο file pointer δεν μπορεί να μετακινηθεί από μια διεργασία σε κάποια αυθαίρετη θέση μέσα στο αρχείο μετά από μια κλήση read().

Στο σημείο αυτό, προκύπτει η ανάγκη να χρησιμοποιήσουμε πληροφορίες από τα structs inode & filp, προκειμένου να αρχικοποιήσουμε τα πεδία του chrdev_state, μέσω του οποίου θα κρατάμε τις πληροφορίες που παίρνουμε από τους αισθητήρες για κάθε μία από τις μετρήσεις μπαταρίας, πίεσης ή φωτός.

Γνωρίζουμε πως ισχύει, λόγω εκφώνησης:

minor = 8*sensor + type,

άρα αρχικοποιούμε τις μεταβλητές sensor kai type αναλόγως:

```
sensor = iminor(inode) / 8;
type = iminor(inode) % 8;
```

Στη συνέχεια, δεσμεύουμε δυναμικά τον απαραίτητο χώρο για το lunix_chrdev_state, μέσω της συνάρτησης του Kernel kmalloc() (χώρος, ο οποίος θα αποδεσμευτεί όταν θα γίνει release και η τελευταία αναφορά σε κάποιο ανοιχτό αρχείο, μέσω της kfree() στην lunix_chrdev_release()).Και πάλι σε περίπτωση αποτυχίας, τυπώνεται μήνυμα λάθους.

Έπειτα παίρνουμε ένα-ένα τα πεδία του private character device state, και τα αρχικοποιούμε.

Προσέχουμε πως η μεταβλητή sensor είναι int, και πως ο πραγματικός sensor ο οποίος θα ανατεθεί στο πεδίο chrdev_state->sensor προέρχεται από τον πίνακα lunix_sensors που έχει οριστεί στο header file lunix.h, και ως index χρησιμοποιούμε το sensor που βρήκαμε παραπάνω. Επίσης, θέτουμε αρχικά τον semaphore του state available, ώστε να μπορεί οποιαδήποτε διεργασία προλάβει να τον κάνει acquire και να ανανεώσει τα δεδομένα που κρατάμε για τις μετρήσεις χωρίς να μπορεί κάποια άλλη διεργασία να παρέμβει(αποφυγή race conditions).

Έπειτα αποθηκεύουμε αυτό το chrdev_state ως το private_data του filp.

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

Η πιο απαιτητική και ωστόσο χρήσιμη κλήση συστήματος, αναλαμβάνει να διαβάσει κάποια byte από τον οδηγίο και να μετακινήσει το pointer.

```c
static ssize_t lunix_chrdev_read(struct file *filp, char __user *usrbuf, size_t cnt, loff_t *f_pos)
{
    ssize_t ret = 0;
    struct lunix_sensor_struct *sensor;
    struct lunix_chrdev_state_struct *state;
    size_t bytes_to_copy;
    state = filp->private_data;
    sensor = state->sensor;
    debug("entering read");
    if (down_interruptible(&state->lock)) {
        /* Handle the case where down_interruptible was interrupted by a signal */
        ret = -ERESTARTSYS;  // the system call should be restarted because it was interrupted by a signal.
        goto out;
    }
    /*
    * If the cached character device state needs to be
    * updated by actual sensor data (i.e., we need to report
    * on a "fresh" measurement), do so
    */
    if (*f_pos == 0) { //buffer empty
        if (lunix_chrdev_state_update(state) == -EAGAIN) { //den xreiazetai refresh, den exei ginei kapoio update, ara nothing to read
            /* The process needs to sleep */
            up(&state->lock);
            /*If, instead, the buffer is empty, we must sleep. Before we can do that,
however, we must drop the device semaphore; if we were to sleep holding it, no
writer would ever have the opportunity to wake us up.*/
            if (wait_event_interruptible(sensor->wq,lunix_chrdev_state_needs_refresh(state) )) {
                ret = -ERESTARTSYS;
                goto out;
            }
            if (down_interruptible(&state->lock))return -ERESTARTSYS;
        }
    }

    debug( "efyge apo if f_pos==0");
    /* Determine the number of cached bytes to copy to userspace */
    bytes_to_copy = MIN(cnt, (size_t)(state->buf_lim - *f_pos));
    debug( "min:%lu", bytes_to_copy);
    /* Copy data to userspace */
    if (copy_to_user(usrbuf, state->buf_data + *f_pos, bytes_to_copy)) {
        ret = -EFAULT; // Error copying data to userspace
        goto out;
    }
    /* Update file position and return the number of bytes read */
    *f_pos += bytes_to_copy;
    ret = bytes_to_copy;
    /* Auto-rewind on EOF mode */
    if (*f_pos == state->buf_lim)
    *f_pos = 0;
out:
    debug("leavinggg");
    up(&state->lock);
    return ret;
}

```

Ξεκινάμε ορίζοντας έναν sensor pointer, που θα μας βοηθήσει να έχουμε πρόσβαση στον αισθητήρα από τον οποίο θα λαμβάνουμε τα δεδομένα, καθώς και έναν pointer στο state, μέσω του οποίου θα αποθηκεύουμε τις μετρήσεις που λαμβάνουμε από τον αισθητήρα. Ακόμη, έχουμε μια μεταβλητή bytes_to_copy, όπου αποθηκεύεται ο αριθμός των Bytes που αντιγράφονται.

Στη συνέχεια δεσμεύουμε τον semaphore για το chrdev state< προκειμένου να πάρουμε τα δεδομένα από τον buffer του state και να τα μεταφέρουμε στο userspace, αποφεύγοντας τα race conditions. Με άλλα λόγια το διάβασμα.δεδομένων πρέπει να γίνεται ατομικά για κάθε διεργασία με κοινό fd (δηλαδή που έχουν γονική σχέση), ώστε να διαβάζει η κάθε μια τα δεδομένα που θέλει και να προχωράει τον pointer, διαφορετικά, πολλές διεργασίες είτε θα διαβάζουν τα ίδια δεδομένα, και δε θα προχωράει καλά ο pointer και ενδεχωμένως να βγεί εκτός ορίων προκαλώντας seg-fault.

Έπειτα τσεκάρουμε αν υπάρχουν δεδομένα, με τα οποία θα επικαιροποιήσουμε τον buffer—pointer. Αν δεν υπάρχουν, τότε η τρέχουσα διεργασία θα πρέπει, αφού πρώτα απελευθερώσει τον semaphore, να κοιμηθεί. Ο λόγος που απαιτείται η απελευθέρωση του semaphore είναι γιατί σε περίπτωση που δεν έρθουν νέα δεδομένα, αλλά η τρέχουσα διεργασία κοιμάται κρατώντας τον semaphore, όλες οι διεργασίες θα περιμένουν στο kernel-space κατι το οποίο πρέπει να μην γίνεται.

Έπειτα μολις έρθουν νέα δεδομένα, μια διεργασία ξυπνάει και παίρνει τον semaphore, ενώ οι υπόλοιπες περιμένουν στο while loop μέχρι να τελειώσει και να είναι ο σεμαφλορος ξανά διαθέσιμος.

Έπειτα, έχοντας πλέον bytes να αντιγράψουμε, κάνουμε determine τον αριθμό των Bytes προς αντιγραφή. Θα πάρουμε το minimum(που έχει οριστεί σε macro #define παραπάνω στον κώδικα) μεταξύ των cnt, (size_t)(state->buf_lim - *f_pos) όπου cnt είναι παράμετρος που περνιέται στην συγκεκριμένη κλήση, ενώ το (size_t)(state->buf_lim - *f_pos) είναι ουσιαστικά ο αριθμός των bytes που έχουν έρθει από τον αισθητηρα. Αυτό συμβαίνει επειδή μπορεί ο αριθμός των byte που έχουν έρθει από τον αισθητήρα μπορεί να είναι μικρότερος από την παράμετρο cnt, και αν παίρναμε το cnt τότε πιθανότατα θα βγαίναμε εκτός των ορίων του πίνακα.

Στη συνέχεια, κάνουμε update το File positions με βάση τη bytes_to_copy μεταβλητή, ενώ έχουμε και το τσεκάρισμα για το αν φτάσαμε στο End of File.

Τέλος, η current process απελευθερώνουμε τον semaphore του state, ώστε να γίνει available για άλλες διεργασίες να τον καταλάβουν και να τροποποιήσουν το state.

### static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state):

Απαραίτητη συνάρτηση προκειμένου να δουλέψει σωστά η read() και η unix_chrdev_state_update().

```
39 static int lunix_chrdev_state_needs_refresh(struct lunix_chrdev_state_struct *state)
40 {
41
42         struct lunix_sensor_struct *sensor;
43         int type;
44         uint32_t last_update_of_sensor;
45         uint32_t last_update_of_buffer;
46
47          sensor = state->sensor;
48          type = state->type;
49
50         last_update_of_sensor= sensor->msr_data[type]->last_update;
51
52         last_update_of_buffer = state->buf_timestamp;
53         debug("last update of senor %d",last_update_of_sensor );
54         //debug("%u",last_update_of_sensor);
55         debug("last update of buffer %d",last_update_of_buffer);
56         return (last_update_of_sensor > last_update_of_buffer);
57 }
```

Η βοηθητική συνάρτηση, παίρνει ως παράμετρο το state προς ανανέωση, και γυρνάει 1 (true) όταν αποφαινόμαστε πως χρειάζεται να ανανεωθούν τα δεδομένα του

buffer του state(τα οποία γενικά λαμβάνονται από τους αισθητήρες), αλλιώς επιστρέφει 0. Τα δεδομένα αυτά χρειάζονται ανανέωση όταν η τελευταία τροποποίηση του sensor, η οποία υποδηλώνεται από το last_update field του sensor, είναι μεγαλύτερη(δηλαδή πιο πρόσφατη) από την τελευταία τροποποίηση του buffer, που υποδηλώνεται από το buf_timestamp field του state.

### static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state):

Απαραίτητη συνάρτηση προκειμένου να δουλέψει σωστά η read().

```
64 static int lunix_chrdev_state_update(struct lunix_chrdev_state_struct *state)
65 {
66         struct lunix_sensor_struct *sensor;
67         uint32_t data;
68          uint32_t time;
69          int ret;
70
71          WARN_ON(!(sensor = state->sensor));
72
73         /*
74         * Any new data available?
75         */
76
77         if(lunix_chrdev_state_needs_refresh(state)) {
78           /*
79            * Grab the raw data quickly, hold the
80            * spinlock for as little as possible.
81            */
82           spin_lock_irq(&sensor->lock);
83           data = sensor->msr_data[state->type]->values[0];
84           time = sensor->msr_data[state->type]->last_update;
85           spin_unlock_irq(&sensor->lock);
86
87           /* Why use spinlocks?  acquires a spin lock and disables interrupts on the local CPU. It is used to protect critical sections of code from concurrent access by multiple processors or
   interrupt handlers. any code must, while holding a spinlock, be atomic and  cannot sleep*/
88
89
90           /*
91            * Now we can take our time to format them,
92            * holding only the private state semaphore
93            */
94
95           if(state->type == TEMP) state->buf_lim = sprintf(state->buf_data, "%ld.%03ldC\t", lookup_temperature[data]/1000, lookup_temperature[data]%1000);
96           else if(state->type == BATT) state->buf_lim = sprintf(state->buf_data, "%ld.%02ldpercent\t", lookup_voltage[data]/100, lookup_voltage[data]%100);
97           else if(state->type == LIGHT) state->buf_lim = sprintf(state->buf_data, "%ldCd\t", lookup_light[data]);
98           state->buf_timestamp = time;
99           ret = 0;
100         }
101         else {
102                 ret = -EAGAIN;
103         }
104
105        debug("leaving update\n");
106        return ret;
107 }
```

Βοηθητική συνάρτηση πραγματοποιεί την αλλαγή των δεδομένων στον buffer με βάση τις νέες μετρήσεις που έρχονται στον sensor, εφόσον η lunix_chrdev_state_need_refresh μας λέει πως τα δεδομένα χρήζουν ανανέωσης. Όπως τονίζεται, θα πρέπει ο σεμαφόρος του chdev state στο σημείο που η lunix_chrdev_state_update καλείται να είναι ήδη acquired από μια διεργασία, προκειμένου μόνο αυτή να μπορεί να επεξεργάζεται και να τροποποιεί τα πεδία του, χωρίς καμία άλλη να έχει ταυτόχρονα πρόσβαση σε αυτά και να τα αλλάζει, ώστε να αποφευχθούν πιθανά race conditions.

Έχουμε έναν sensor pointer, sensor, που θα αξιοποιηθεί για να γίνουν προσβάσιμες οι μετρήσεις στους αισθητήρες με τους οποίους θα ανανεώσουμε τον buffer του state, η μεταβλητή data, στην οποία θα ανατεθούν οι καθαρές μετρήσεις τις οποίες θα μορφοποιήσουμε, ώστε να τις αποθηκεύσουμε στον buffer και την μεταβλητή time για να αποθηκεύσουμε την τελευταία τροποποίηση στο timestamp του αισθητήρα.

Έτσι, εάν όντως ο buffer μας χρειάζεται ανανέωση, αρχικά κάνουμε acquice το spinclock του sensor(χρησιμοποιούμε spinlocks αντί για σημαφόρους, γιατί θέλουμε να μην μπορεί καμία άλλη διεργασία ή hardware interrupt να διακόψει τη local CPU, προκειμένου να ολοκληρωθεί σωστά η λήψη δεδομένων από τους σένσορες). Λαμβάνουμε στη συνέχεια τα δεδομένα μας, ανανεώνουμε κατάλληλα το time και απελευθερώνουμε το spinlock.

Έπειτα, αποθηκεύουμε στον buffer τα raw data, αφού πρώτα τα υποβάλλουμε σε μια μικρή τροποποίηση  ανάλογα με τον τύπο της μέτρησης, προκειμένου να είναι πιο ευανάγνωστα. Τέλος, ανανεώνουμε και το buf_timestamp, το οποίο κωδικοποιεί την τελευταία επικαιροποίηση του buffer του state.

### static int lunix_chrdev_release(struct inode *inode, struct file *filp):

Εδώ πραγματοποιείται η αποδέσμευση του χώρου που καταναλώνει το private character device state, που δεσμεύτηκε μέσω της kmalloc() στην κλήση lunix_chrdev_open(). Η αποδέσμευση του χώρου αυτού γίνεται μέσω της kfree().

```
165 static int lunix_chrdev_release(struct inode *inode, struct file *filp)
166 {
167         kfree(filp->private_data); //ousiastika diagrafoume to state mesa sto file pointer
168         return 0;
169 }
```

# Δοκιμή

Προκειμένου να τεστάρουμε την σωστή λειτουργία του driver, αρχικά δοκιμάσαμε τα:

cat /dev/lunix2-temp, dd if=/dev/lunix0-temp bs=N for N 1,2,5,256, τα οποία λειτουργούσαν όπως έπρεπε, δηλαδή το πρώτο εμφάνιζε το περιεχόμενο του οδηγού κανονικά, ενώ η δεύτερη εντολή εμφάνιζε ανάλογα το μέγεθος του N, ολο και περισσότερα δεδομένα με τη σειρά (όπως θα έπρεπε δηλαδή να διαβάζονται τα δεδομένα απο διεργασίες που έχουν γονική σχέση).

Έπειτα δημιουργήσαμε τα δικά μας tests ένα εκ των οποίων παρουσιάζεται από κάτω:

Πρόκειται για ένα .c source file, μέσα στο οποίο καλώντας την fork() δημιουργούμε από μια γονική διεργασία πολλά παιδιά-διεργασίες, τα οποία κληρονομώντας τον ίδιο file descriptor για το file(το οποίο περιέχει τις συνεχώς ανανεωμένες μετρήσεις) από τη διεργασία-γονέα, μπαίνουν και διαβάζουν ταυτόχρονα. Σκοπός αυτού του source file είναι να ελέγξουμε πως, μέσω του semaphore για το state και των spinlocks για τους αισθητήρες, πραγματοποιείται συγχρονισμός των διεργασιών και πως τα αποτελέσματα των μετρήσεων που αναμένουμε εκτυπώνονται σωστά.

Καλούμε, λοιπόν, την fork() από τη διεργασία πατέρα μέσα σε ένα for loop, το οποίο επαναλαμβάνεται όσα και τα παιδιά που θέλουμε να φτιάξουμε. Όσον αφορά τον κώδικα των διεργασιών-παιδιών, αυτά καλούν την συνάρτηση readFromFile(),στην οποία, μέσα σε ένα ατέρμονο loop, προσπαθούν να κάνουν access στον κοινό fd, και να διαβάσουν ταυτόχρονα από εκεί. Έτσι τεστάρουμε εάν για πολλαπλές διεργασίες τόσο τα spinlocks όσο και τα semaphores λειτουργούν όπως πρέπει.

Ακολουθεί ο κώδικας, τα αποτελέσματα του οποίου ήταν τα αναμενόμενα, δηλαδή η κάθε διεργασία παιδί — όποια προλάβει — διαβάζει και τυπώνει δεδομένα ανάλογα με το μέγεθος του BUFFER_SIZE που έχουμε ορίσει.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "lunix-chrdev.h"

#define FILE_PATH "/dev/lunix0-temp"
#define BUFFER_SIZE 1024
#define NUM_CHILDREN 200
void readFromFile(int childNumber, int fd) {
        int file_descriptor=fd;
        char buffer[BUFFER_SIZE];

        // Open the file with read-only access


        if (file_descriptor == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
        }

        // Read and print the contents of the file
        ssize_t bytesRead = read(file_descriptor, buffer, sizeof(buffer) - 1);

        if (bytesRead == -1) {
        perror("Error reading file");
        close(file_descriptor);
        exit(EXIT_FAILURE);
        }

        buffer[bytesRead] = '\0'; // Null-terminate the string

        printf("Child %d - Content of %s: %s\n", childNumber, FILE_PATH, buffer);

        // Close the file descriptor
        close(file_descriptor);
}

int main() {
        pid_t pid;
        int fd = open(FILE_PATH, O_RDONLY);

        for (int i = 1; i <= NUM_CHILDREN; ++i) {
        pid = fork();

        if (pid == -1) {
                perror("Error forking");
                exit(EXIT_FAILURE);
        } else if (pid == 0) {
                // Child process
                readFromFile(i, fd);
                exit(EXIT_SUCCESS);
        }

        }

        // Parent process waits for all child processes to finish
        for (int i = 0; i < NUM_CHILDREN; ++i) {
        wait(NULL);
        }

        return 0;
}

```