ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαστήριο Λειτουργικών Συστημάτων

8ο εξάμηνο, Ροή Υ, ΗΜΜΥ

Υλοποίηση ενός οδηγού συσκευής για τον πυρήνα του Linux

Σταυρακάκης Χρήστος, 03106165
Τσιούρης Ιωάννης, 03106193

Η/Μ Παράδοσης: **26 Απριλίου 2010**

Αθήνα, 2010

1 Εισαγωγή

Στην συγκεκριμένη άσκηση υλοποιούμε έναν οδηγό συσκευής (`Device Driver`) για μια εικονική συσκευή χαρακτήρων στον πυρήνα του `Linux`. Η συσκευή αυτή προσομοιώνεται στη μνήμη RAM του υπολογιστή. Τα δεδομένα που διαβάζονται και γράφονται προς τη συσκευή θα αποθηκεύονται και διαβάζονται από έναν χώρο μνήμης τον οποίο δεσμεύει ο driver της.

Ο οδηγός αυτός θα υλοποιηθεί ως ένα `kernel module` στον πυρήνα του Linux. Τα modules αποτελούν τμήματα κώδικα τα οποία μπορούν να φορτωθούν στον πυρήνα του Linux κατά τον χρόνο εκτέλεσης (`runtime`). Κάθε module αποτελεί object code το οποίο γίνεται linked με τον πυρήνα κατά το χρόνο εκτέλεσης. Έτσι ο πηγαίος κώδικας του πυρήνα παραμένει όσο μικρότερος γίνεται επιτρέποντας στους χρήστες να φορτώσουν δυναμικά όσες λειτουργίες επιθυμούν. Οι περισσότεροι οδηγοί συσκευής, όπως και ο δικός μας, είναι υλοποιημένοι σαν kernel modules.

2 Οδηγός συσκευής

Όπως ήδη αναφέραμε ο οδηγός μας αφορά μια εικονική συσκευή η οποία προσομοιώνεται στη μνήμη του υπολογιστή μας. Η συσκευή αυτή θα είναι μια συσκευή χαρακτήρων (`Character Device`), δηλαδή μία συσκευή η οποία μπορεί να προσπελαστεί σαν μια ροή από byte, ακριβώς όπως και ένα αρχείο. Στο εξής θα ονομάζουμε αυτή τη συσκευή καθώς και τον οδηγό της **lunix**

Οι λειτουργίες που υποστηρίζονται από τον οδηγό που υλοποιούμε είναι οι εξής:

- **open**
- **release**
- **write**
- **read**
- **lseek**
- **ioctl**
- **mmap**

2.1 Συσκευή και Μοντέλο μνήμης

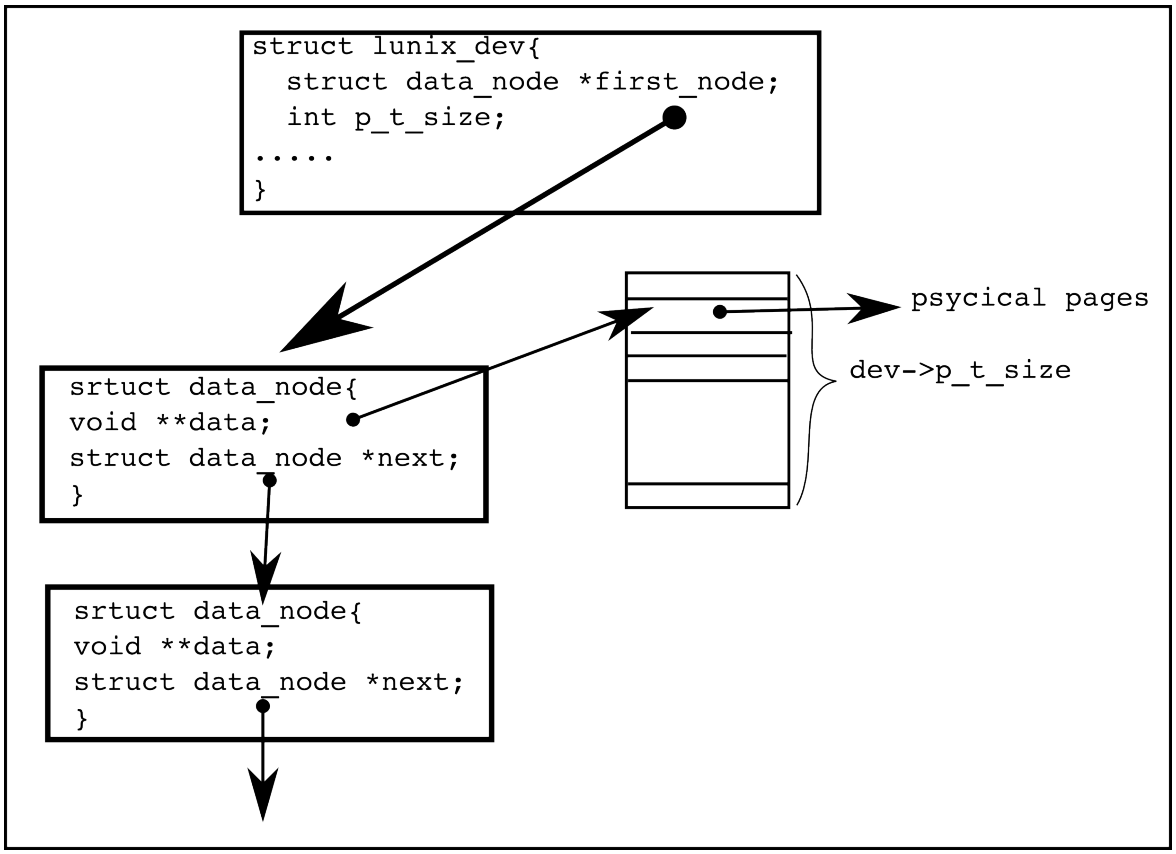
Καθώς τα δεδομένα που διαβάζονται και γράφονται προς τη συσκευή μας ουσιαστικά διαβάζονται και γράφονται χρειάστηκε να καθορίσουμε πως θα γίνεται η ανάθεση της μνήμης στη συσκευή.

Ο ευκολότερος τρόπος ήταν ο οδηγός μας να δεσμεύει από την αρχή έναν σταθερού μεγέθους χώρο μνήμης με δυνατότητα να καθορίζει ο χρήστης το μέγεθος του. Καθώς όμως η προσέγγιση της στατικής δέσμευσης μνήμης είναι και αρκετά δαπανηρή για τη μνήμη μας και θέτει όρια στις εφαρμογές μας, επιλέξαμε η μνήμη του οδηγού μας να δεσμεύεται δυναμικά. Όσο περισσότερα γράφονται στην συσκευή, τόσο περισσότερη μνήμη τις ανατίθεται.

Η ανάθεση μνήμης γίνεται ανά φυσικές σελίδες(pages) της μνήμης με χρήση της συνάρτησης $\_\_get\_\_free\_\_pages$(unsigned int flag, unsigned long order). Σαν flag χρησιμοποιούμε το GFP_KERNEL, για να γίνει ανάθεση σελίδων στο χώρο πυρήνα. Το όρισμα order καθορίζει πόσες σελίδες θα δεσμεύσουμε. Σε κάθε κλήση δεσμεύουμε 2^{order} σελίδες.

Η συσκευή μας μοντελοποιείται ως μια δομή `struct lunix_dev` η οποία έχει διάφορες πληροφορίες που αφορούν τη συσκευή. Μία από αυτές είναι ο δείκτης `first_node` σε μια δομή `struct data_node`, που αποτελεί το πρώτο στοιχείο μιας λίστας από data_node. Κάθε data_node περιέχει ένα δείκτη σε ένα πίνακα από `p_t_size` δείκτες. Κάθε ένας από αυτούς τους δείκτες δείχνει σε 2^{p_order} φυσικές σελίδες τις οποίες δεσμεύουμε με την __get_free_pages. Τόσο το `p_t_size` όσο και

το p_order ορίζονται μέσα στη δομή lunix_dev. Όλα αυτά θα γίνουν πιο κατανοητά με το επόμενο σχήμα 1:



Σχήμα 1: Μοντέλο Μνήμης Lunix

Επίσης οι δομές lunix_dev και data_node ορίζονται στο lunix2k10.h και είναι οι εξής :

```
struct data_node {
    void **data;
    struct data_node *next;
};


struct lunix_dev {
    struct data_node *first_node;
    size_t size;
    int p_order;
    int p_t_size;
    int vmas;
    struct semaphore sem;
    struct cdev mydev;
};
```

Το πεδίο struct semaphore sem αποτελεί έναν σημαφόρο ο οποίος χρησιμοποείται για την αντιμετώπιση των προβλημάτων ταυτόχρονης πρόσβασης στην συσκευή(για παράδειγμα όταν δύο διαφορετικές διεργασίες έχουν χρησιμοποιούν τη συσκευή). Οι κύριες εντολές που αφορούν τον σημαφόρο είναι οι:

- down_interruptible(&dev->sem) η οποία παίρνει το σημαφόρο. Το interruptible χρησιμοποιείται έτσι ώστε η συσκευή να είναι αποκρίσιμη σε σήματα(linux signals).
- up(&dev->sem) η οποία απελευθερώνει το σημαφόρο.

2.2 Αρχικοποίηση Module

Κατά την εισαγωγή του module πρέπει να γίνουν διάφορες εργασίες που αφορούν τον οδηγό. Αυτές τις εργασίες θα αναλάβει η συνάρτηση:

```
static int __init lunix_init(void)
```

Το __init δηλώνει ότι η συνάρτηση θα χρησιμοποιηθεί μόνο κατά την εισαγωγή του module οπότε μετά μπορεί να απορριφθεί από τον module_loader. Επίσης δηλώνουμε στον module_loader ότι αυτή η συνάρτηση θα χρησιμοποιηθεί κατά την αρχικοποίηση με χρήση του ειδικού macro module_init:

```
module_init(lunix_init);
```

Κατά την αρχικοποίηση του οδηγού πραγματοποιούνται οι εξής λειτουργίες:

1. Η δέσμευση ενός device number τύπου dev_t. Ο device number αποτελείται από τον κατάλληλο major και minor number. Κάθε character device έχει τον δικό του device number. Ο major number δηλώνει τον driver που είναι συσχετισμένος με τη συσκευή και ο minor number χρησιμοποιείται από τον πυρήνα για να αναφέρεται στην συγκεκριμένη συσκευή.

   Στην υλοποίηση μας θα χρησιμοποιήσουμε δυναμική δέσμευση major number έτσι ώστε να αποφύγουμε προβλήματα από conflicts. Αυτό το επιτυγχάνουμε με τον εξής κώδικα:

   ```
   dev_t dev = 0;
   err = alloc_chrdev_region(&dev, lunix_minor, 1, "lunix");
   ```

2. Δέσμευση χώρου μνήμης για τη δομή lunix_dev καθώς και αρχικοποίηση των μεταβλητών της. Ενδιαφέρον παρουσιάζει μόνο η αρχικοποίηση του σημαφόρου που υπάρχει στη δομή και χρησιμοποιείται για την αποφυγή race conditions. Ο σημαφόρος δηλώνεται και αρχικοποιείται σε τιμή 1 με την `INIT_MUTEX(&dev->sem)`.

3. Τέλος πρέπει να δηλώσουμε στον πυρήνα την συσκευή μας ως ένα `struct cdev` το οποίο χρησιμοποιείται από τον πυρήνα για να αναφέρεται στην συσκευή. Οπότε αρχικόποιούμε τη δομή cdev που υπάρχει μέσα στο lunix_dev και στην συνέχεια τη δηλώνουμε:

   ```
   cdev_init(&device->mydev, &lunix_fops);
   err = cdev_add(&device->mydev, dev, 1);
   ```

   Κατά την αρχικοποίηση της δομής περνάμε ένα δείκτη στη δομή lunix_ops η οποία αποτελεί μια δομή τύπου file_operations και περιέχει δείκτες στις συνάρτησεις που υλοποιούν τις λειτουργίες που αναφέρονται στη δομή και επικαλύπτουν αυτές του λειτουργικού.. Η δομή αυτή είναι η εξής:

   ```
   static const struct file_operations lunix_fops = {
       .owner      = THIS_MODULE,
       .open       = lunix_open,
       .release    = lunix_release,
       .read       = lunix_read,
       .write      = lunix_write,
       .llseek     = lunix_llseek,
   ```

```
        .ioctl      = lunix_ioctl,
        .mmap       = lunix_mmap,
};
```

2.3 Υλοποίηση λειτουργιών

Στην συνέχεια θα παρουσιάσουμε συνοπτικά την υλοποίηση των βασικών μας λειτουργιών. Θα αναφερθούμε μόνο στα σημεία που έχουν ενδιαφέρον καθώς ο κώδικας τους μαζί με τα σχόλια είναι αρκετά κατανοητός και μπορεί να βρεθεί στο παράρτημα. XXX

2.3.1 open

Η μέθοδος open υλοποιείται στον οδηγό μας από την μέθοδο:

```
static int
lunix_open(struct inode *inode, struct file *file)
```

Αρχικά χρειάζεται να εξάγουμε από τον δομή inode την συσκευή(lunix_dev) στην οποία αναφέρεται για να την εισάγουμε στο πεδίο file->private_data έτσι ώστε να μπορούμε να το χρησιμοποιήσουμε στις υπόλοιπες μεθόδους. Αυτό επιτυγχάνεται με την μακροεντολή του πυρήνα container_of:

```
  struct lunix_dev *dev;
  dev = container_of(inode->i_cdev, struct lunix_dev, mydev);
  file->private_data = dev;
```

Ο δεύτερος κύριος ρόλος της open είναι να ανανεώνει τον δείκτη f_pos ή να μηδενίζει τη συσκευή, ανάλογα με τα flag που έχει ανοιχτεί. Τα flags αυτά περιέχονται στο file->f_flags.

Αν το flag είναι το O_APPEND χρειάζεται απλά να θέσουμε τον f_pos στο τρέχων μέγεθος της συσκευής: $file->f\_pos = dev->size$;

Αν το flag είναι O_TRUNC χρειάζεται να "μηδενίσουμε" τη συσκευή. Αυτό απαιτεί να ελευθερώσουμε όλη τη μνήμη την οποία έχουμε δεσμεύσει και να θέσουμε το dev->size στο 0. Διατρέχουμε λοιπόν όλη τη λίστα των data_node και ελευθερώνουμε όσους πίνακες δεικτών και σελίδες στο δίσκο έχουμε δεσμεύσει.

2.3.2 release

Η μέθοδος release υλοποιείται από τον οδηγό μας με τη μέθοδο:

```
static int
lunix_release(struct inode *inode, struct file *file)
```

Στην υλοποίηση μας δεν επιτελεί κάποια ιδιαίτερη λειτουργία.

2.3.3 write

Η μέθοδος write υλοποιείται από τον οδηγό μας με τη μέθοδο:

```
static ssize_t
lunix_write(struct file *file, const char __user *buff, size_t count, loff_t
    *ppos)
```

Η μέθοδος αυτή χρησιμοποιείται για τη μεταφορά δεδομένων από τον χρήση στη συσκευή. Τα δεδομένα υπάρχουν στον buffer που δείχνει ο buff και πρέπει να αντιγραφούν count bytes στη συσκευή, αρχίζοντας από τη θέση ppos.

Σύμφωνα με το μοντέλο μνήμης πρέπει να βρούμε σε ποια σελίδα αντιστοιχεί ο δείκτης ppos. Για να το βρούμε αυτό ακολουθούμε την εξής διαδικασία.

1. Αρχικά βρίσκουμε πόσα bytes καταλαμβάνουν συνολικά οι σελίδες που αντιστοιχούν σε έναν data_node. Αυτά είναι:
   $dtn\_size = (PAGE\_SIZE << dev- > p\_order) * dev- > p\_size$;

2. Στη συνέχεια βρίσκουμε μέσα σε ποιο από τα data_node βρίσκεται η σελίδα που περιέχει την ppos. Αυτό το βρίσκουμε διαιρώντας το ppos με το dtn->size. Υπολογίζουμε και το node_off το οποίο θα χρησιμοποιηθεί στη συνέχεια.

```
node_index = ((long) (*ppos) / dtn_size);
node_off =  ((long)(*ppos) % dtn_size);
```

3. Τέλος υπολογίζουμε σε ποια από τις σελίδες που υπάρχουν στον πίνακα του data_node "πέφτει" το ppos, καθώς και ποιο είναι το offset μέσα στην σελίδα στο οποίο αντιστοιχεί η διεύθυνση ppos.

```
table_index = (node_off / (PAGE_SIZE<<dev->p_order));
table_off = (node_off % (PAGE_SIZE<<dev->p_order));
```

Έπειτα ακολουθούμε τη λίστα με τα data_node μέχρι να φτάσουμε στον node_index node. Αν οι ενδιάμεσοι data_node δεν υπάρχουν τους δημιουργούμε(μόνο τους node και όχι τον πίνακα και τι σελίδες). Έτσι δημιουργούνται κενά στη συσκευή. Μόλις βρούμε τον κατάλληλο data_node ελέγχουμε αν υπάρχει ο πίνακας δεικτών και αν δεν υπάρχει τον δημιουργούμε. Έπειτα ελέγχουμε αν υπάρχει σελίδα στην κατάλληλη θέση του πίνακα, και αν δεν υπάρχει την δεσμεύουμε.

```
    while( node_index > 0){
        if(!dtn->next){
            printk(KERN_NOTICE "lunix_write: Allocating more nodes on index
                %d\n", node_index);
            dtn->next = kmalloc(sizeof(struct data_node), GFP_KERNEL);
            if (!dtn->next)
                goto out;
            memset(dtn->next, 0 , sizeof(struct data_node));
        }
        dtn = (dtn->next) ;
        node_index--;
    }

    if(!dtn->data){
        printk(KERN_NOTICE "lunix_write: Allocating a page table\n");
        dtn->data = kmalloc (dev->p_t_size * sizeof(void*), GFP_KERNEL);
        if (!dtn->data)
            goto out;
        memset(dtn->data, 0 , dev->p_t_size * sizeof(void*));
    }

    if(!dtn->data[table_index]){
        printk(KERN_NOTICE "lunix_write: Allocating a page\n");
        dtn->data[table_index] = __get_free_pages(GFP_KERNEL, dev->p_order);
        if (!(dtn->data[table_index]))
            goto out;
        memset(dtn->data[table_index], 0 , PAGE_SIZE<<dev->p_order);
    }
```

Στη συνέχεια ελέγχουμε αν το count είναι μεγαλύτερο από το χώρο που απομένει μέσα στη σελίδα. Αν είναι μεγαλύτερο τότε γράφουμε μόνο όσα bytes χωράνε μέσα στη σελίδα και αγνοούμε

τα υπόλοιπα. Αυτό είναι μια κακή πρακτική αλλά αποφύγαμε να υλοποιήσουμε το να συνεχίζεται η εγγραφή σε επόμενες σελίδες για να είναι πιο αναγνώσιμος ο κώδικας.

Τέλος μένει να μεταφέρουμε τα bytes με χρήση της $copy\_from\_user$ καθώς και να ανανεώσουμε τους δείκτες *ppos και dev->size.

```
if (count > ((PAGE_SIZE<<dev->p_order) - table_off))
        count = ((PAGE_SIZE<<dev->p_order) - table_off);

    if(copy_from_user( ((dtn->data[table_index]) + table_off) , buff, count)){
        ret = -EFAULT;
        goto out;
    }
    *ppos += count;
    if (*ppos > dev->size)
        dev->size = *ppos;
```

2.3.4 read

Η μέθοδος release υλοποιείται από τον οδηγό μας με τη μέθοδο:

```
static ssize_t
lunix_read(struct file *file, char __user *buff, size_t count, loff_t *ppos)
```

Η υλοποίηση της μοιάζει πολύ με τη μέθοδο write οπότε δε θα αναφερθούμε σε λεπτομέρειες. Απλά να σημειώσουμε το εξής: Όπως ήδη αναφέραμε με τη write δημιουργούνται κενά στο χώρο μνήμης της συσκευής. Όταν η read προσπαθήσει να διαβάσει σε κάποιο από αυτά τα κενά της συσκευής, δεν διαβάζει τίποτα και επιστρέφει 0 στο χρήστη. Μια άλλη προσέγγιση θα ήταν να επέστρεφε στο χρήστη τόσα μηδενικά όσα αντιστοιχούσαν στον κενό χώρο.

Επίσης η read δεν μπορεί να διαβάσει έξω από το μέγεθος της συσκευής. Όταν ο χρήστης προσπαθήσει να διαβάσει περισσότερο χώρο από όσο απομένει στη συσκευή τότε διαβάζει μόνο τον εναπομείναντα χώρο. Επίσης σε αντιστοιχία με την write, η read διαβάζει το πολύ όσο χώρο αντιστοιχεί στις σελίδες που έχουν δεσμευθεί και υπάρχουν στον δείκτη μιας θέση του πίνακα δεικτών. Οι αναφερθέντες έλεγχοι είναι οι εξής:

```
if (*ppos >= dev->size)
    goto out;
if (*ppos + count > dev->size)
    count = dev->size - *ppos;

if (count > ((PAGE_SIZE<<dev->p_order) - table_off))
    count = ((PAGE_SIZE<<dev->p_order) - table_off);
```

2.3.5 lseek

```
loff_t
lunix_llseek(struct file *filp, loff_t offset, int whence)
```

Η επομένη μέθοδος υλοποιεί τα system call lseek και llseek. Όταν αυτή η μέθοδος δεν είναι υλοποιημένη, ο πυρήνας τις υλοποιεί μεταβάλλοντας τον δείκτη file->f_pos. Η μέθοδος μας, ανάλογα με το flag whence, θέτει σε μια μεταβλητή pos τύπου loff_t pos, την θέση η οποία θέλουμε να μπει στο file->f_pos, την θέτουμε και την επιστρέφουμε:

```
switch(whence){
    case SEEK_SET:  // Set to this offset
```

```
        printk(KERN_NOTICE "From lseek");
        pos = offset;
        break;

    case SEEK_CUR:  // Set with offset to current
        pos = filp->f_pos + offset;
        break;

    case SEEK_END:  // Set with offset to size of device..
        pos = dev->size + offset;
        break;

    default:
        return -EINVAL;

    if (pos < 0) return -EINVAL;

    filp->f_pos = pos;
    return pos;
}
```

2.3.6 ioctl

Το system call ioctl υλοποιείται από τον οδηγό μας με τη μέθοδο:

```
int lunix_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
    unsigned long arg)
```

Η μέθοδος αυτή προσφέρει στους χρήστες επιπλέον λειτουργικότητα από αυτήν των read και write. Συγκεκριμένα χρησιμοποιείται για να προσφέρει στους χρήστες τη δυνατότητα να έχουν κάποιο έλεγχο στη συσκευή (πχ καθορισμό baud rate ή eject media), καθώς και τον καθορισμό κάποιων από τις παραμέτρους της.

Στη δικιά μας υλοποίηση η lunix_ioctl προσφέρει μόνο τη δυνατότητα να διαβάζουμε και να θέτουμε την παράμετρο size της συσκευής (dev->size).

Το ioctl αποτελεί ένα system call το οποίο δεν αποτελεί μία δεδομένη, αλλά πολλές διαφορετικές λειτουργίες. Το ποια λειτουργία θα εκτελεστεί καθορίζεται από το όρισμα cmd. Το όρισμα arg αποτελεί ένα όρισμα το οποίο αποκτά σημασία ανάλογα με τη cmd. Είτε είναι κάποιος τύπος δεδομένων ή ένας δείκτης. Με αυτόν τον τρόπο μπορούμε να περάσουμε και να λάβουμε από την ioctl "αυθαίρετα" ορίσματα/αποτελέσματα. Για να είναι ποιο εύκολη η χρήση της εντολής συνήθως δίνονται συμβολικά ονόματα για τις αριθμητικές τιμές του cmd οι οποίες προσδιορίζονται στη βιβλιοθήκη.

Οι αριθμητικές τιμές που αντιστοιχούν σε κάθε λειτουργία πρέπει να είναι μοναδικές στο σύστημα έτσι ώστε να αποφεύγονται λάθη όπου θα δοθεί κάποια σωστά ορισμένη λειτουργία σε κάποια συσκευή η οποία δεν την υποστηρίζει. Για αυτό το λόγο οι αριθμητικές τιμές για κάθε λειτουργία χωρίζονται στα εξής πεδία:

- `Magic Number` με μέγεθος 8 bits. Είναι ένας μοναδικός αριθμός που επιλέγεται και χρησιμοποιείται για όλη τη συσκευή. Για την επιλογή του πρέπει να ελέγξουμε τους υπόλοιπους μαγικούς αριθμούς που χρησιμοποιούνται στο σύστημα
- `Number` με μέγεθος 8 bits. Αύξων αριθμός για κάθε λειτουργία.
- `Direction` ορίζει την κατεύθυνση ροής των δεδομένων. Η κατεύθυνση αυτή ορίζεται από την πλευρά του της εφαρμογής. Για παράδειγμα η τιμή _IOC_READ χρησιμοποιείται για διάβασμα από την πλευρά του χρήστη, δηλαδή για γράψιμο από τη συσκευή στο χώρο χρήστη. Οι υπόλοιπες επιλογές είναι οι _IOC_WRITE και _IOC_NONE(no data transfer).

- `Size`: Το μέγεθος των δεδομένων που εμπλέκονται. Το πεδίο αυτό εξαρτάται από την αρχιτεκτονική και δεν είναι υποχρεωτικό.

Για την ευκολότερη δήλωση προσφέρεται μια σειρά από μακροεντολές, όπου type ο Magic Number, nr το Number και datatype ο τύπος:

- _IO(type,nr): χωρίς ορίσματα
- _IOR(type, nr ,datatype): διάβασμα από driver αντικειμένου datatype
- _IOW(type, nr, datatype): γράψιμο στον driver αντικειμένου datatype
- _IORW(type, nr, datatype): διάβασμα/γράψιμο

Για τον οδηγό μας λοιπόν έχουμε τις εξής δηλώσεις:

```
#define LUNIX_IOC_MAGIC 'k'
#define LUNIX_IOC_GETSIZE    _IOW(LUNIX_IOC_MAGIC, 0, int *)
#define LUNIX_IOC_SETSIZE    _IOR(LUNIX_IOC_MAGIC, 1, int *)
```

Στη μέθοδο μας τώρα πρέπει να κάνουμε τους εξής ελέγχους:

- Το ότι το όρισμα cmd έχει σαν magic number τον magic number της συσκευής:

```
if (_IOC_TYPE(cmd) != LUNIX_IOC_MAGIC) return -ENOTTY;
```

- Ο δεύτερος έλεγχος αφορά το όρισμα arg. Καθώς στην ioctl μας το όρισμα αυτό είναι δείκτης σε int, ο οδηγός πρέπει να ελέγξει αν ο δείκτης αυτός είναι έγκυρος και αν υπάρχουν δικαιώματα εγγραφής ή ανάγνωσης. Αυτό το επιτυγχάνουμε με τη συνάρτηση `int access_ok`. Αξίζει να σημειώσουμε ότι το return value της συνάρτησης είναι αντίθετο από το συνηθισμένο. Δηλαδή επιστρέφει 1 για επιτυχία και 0 διαφορετικά.Ο έλεγχος λοιπόν είναι ο εξής:

```
if (_IOC_DIR(cmd) & _IOC_READ)
        ret = !access_ok(VERIFY_WRITE, (void __user *)arg,
            _IOC_SIZE(cmd));
    else if (_IOC_DIR(cmd) & _IOC_WRITE)
        ret = !access_ok(VERIFY_READ, (void __user *)arg,
            _IOC_SIZE(cmd));
    if (ret)
        return -EFAULT;
```

Τέλος για το διάβασμα και γράψιμο χρησιμοποιούνται οι συναρτήσεις __put_user και __get_user οι οποίες είναι οι αντίστοιχες των copy_from_user και copy_to_user αλλά optimized για συγκεκριμένους τύπους δεδομένων. Οπότε οι αλλαγές για size της συσκευής γίνονται ως εξής:

```
switch (cmd){
    case LUNIX_IOC_GETSIZE:
        dev = filp->private_data;
        ret = __put_user(dev->size, (int __user *)arg);
        break;
    case LUNIX_IOC_SETSIZE:
        dev = filp->private_data;;
        ret = __get_user(dev->size, (int __user *)arg);
        break;
    default:
        ret = -ENOTTY;;
}
```

2.3.7 mmap

Η τελευταία μέθοδος μας είναι η lunix_mmap η οποία αφορά το system call mmap το οποίο χρησιμοποιείται για να γίνεται map χώρος μνήμης της συσκευής στο χώρο χρήστη. Κάθε κλήση στην mmap δημιουργεί μια περιοχή εικονικών διευθύνσεων(Virtual Memory Area). Για τον λόγο αυτό το λειτουργικό διατηρεί μια δομή για κάθε VMA, την `vm_area_struct` η οποία ορίζεται στο `linux/mm_types.h`.

Αυτή η δομή περιέχει έναν δείκτη σε μια δομή`struct vm_operations_struct`, η οποία ορίζεται στο `linux/mm.h>`,και η οποία περιέχει pointers σε συναρτήσεις που υλοποιούν λειτουργίες που χρειάζονται για την διαχείριση του χώρου μνήμης του χρήστη. Οι κύριες λειτουργίες που χρειάζονται και υλοποιούμε είναι οι εξής:

- ```
  void (*open)(struct vm_area_struct * area);
  ```

  Χρησιμοποιείται για την αρχικοποίηση της VMA, και καλείται κάθε φορά που γίνεται μια νέα αναφορά στην VMA, εκτός από την πρώτη φορά που γίνεται μέσω της mmap.

- ```
  void (*close)(struct vm_area_struct * area);
  ```

  Καλείται κάθε φορά που καταστρέφεται ένα VMA.

- ```
  int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);
  ```

  Η συνάρτηση αυτή είναι η σημαντικότερη συνάρτηση. Κάθε φορά που το πρόγραμμα χρήστη κάνει μια αναφορά σε μια σελίδα που ανήκει στη VMA αλλά δεν υπάρχει στη μνήμη καλείται η fault.

Αρχικά λοιπόν ορίζομαι τη δομή μας vm_operations_struct:

```
struct vm_operations_struct lunix_vm_ops = {
    .open = lunix_vma_open,
    .close = lunix_vma_close,
    .fault = lunix_vma_fault,
};
```

Η μέθοδος lunix_mmap δεν έχει ιδιαίτερες λειτουργίες να εκτελέσει καθώς αυτές θα εκτελούνται από την fault. Ο κύριος ρόλος της είναι να αρχικοποιήσει στο vma τον δείκτη vm_ops και να τον θέσει στο δικό μας lunix__ops καθώς και να θέσει στα vm_flags την τιμή VM_RESERVER που αποτρέπει τον πυρήνα από τα να βγάζει τις σελίδες από τη μνήμη. Επίσης καλεί την lunix_vm_open καθώς όπως αναφέραμε δεν καλείται αυτόματα την πρώτη φορά. Επίσης η mmap λειτουργεί στον οδηγό μας μόνο όταν το order είναι 0. Αυτό συμβαίνει γιατί το η μέθοδος fault αναλαμβάνει τον έλεγχο 1 σελίδας. Θα μπορούσαμε να το διορθώσουμε αυτό αλλά θα δημιουργούταν αρκετά προβλήματα τον page reference counter που διατηρεί το λειτουργικό σύστημα για να ελευθερώνει σελίδες. Έχουμε λοιπόν τα εξής:

```
int lunix_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct inode *inode = filp->f_dentry->d_inode;
        struct lunix_dev *dev = filp->private_data;
        /* refuse to map if order is not 0 */
        if (dev->p_order)
        return -ENODEV;

        vma->vm_ops = &lunix_vm_ops;
```
```

```
        vma->vm_flags |= VM_RESERVED;
        vma->vm_private_data = filp->private_data; //save the dev for later use
        lunix_vma_open(vma);    //for the first map
        return 0;
}
```

Η μέθοδοι open και close αναλαμβάνουν μόνο να αυξάνουν και να μειώνουν τον μετρητή dev->vmas ο οποίος αποτελεί έναν μετρητή των ενεργών mappings.

Η κύρια συνάρτηση όπως αναφέραμε είναι η fault (nopage σε παλαιότερες εκδόσεις).

```
int lunix_vma_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
```

Αυτή καλείται κάθε φορά που γίνεται ένα page fault στον mapped χώρο του χρήστη

Η μέθοδος αυτή δέχεται ένα pointer σε vm_area_struct και έναν pointer σε μια δομή vm_fault που περιλαμβάνει διάφορες πληροφορίες και είναι η εξής:

```
struct vm_fault {
    unsigned int flags;             /* FAULT_FLAG_xxx flags */
    pgoff_t pgoff;                  /* Logical page offset based on vma */
    void __user *virtual_address;   /* Faulting virtual address */

    struct page *page;              /* ->fault handlers should return a
                                     * page here, unless VM_FAULT_NOPAGE
                                     * is set (which is also implied by
                                     * VM_FAULT_ERROR).
                                     */
```

Το pgoff είναι το offset της σελίδας που προκάλεσε το σφάλμα μέσα στο VMA. Το virtual_address αποτελεί που προσπέλασε ο χρήστης. Η fault πρέπει να εντοπίσει την σελίδα που λείπει και να την βάλει στη θέση που δείχνει ο pointer *page. Αν κάτι πάει στραβά, ενημερώνει τον πυρήνα θέτοντας το κατάλληλο error code στο πεδίο flags.

Η fault μας εντοπίζει τη διεύθυνση της σελίδας με βάση το offset. Στην συνέχεια χρησιμοποιώντας τη συνάρτηση `virt_to_page` παίρνει τη έναν pointer στη δομή page που δείχνει η διεύθυνση. Αφού αυξήσει τον reference counter της σελίδας με την get_page , τη θέτει στη vmf->pages και επιστρέφει:

```
page = virt_to_page(dtn->data[table_offset]);
    if (!page) goto out;
    get_page(page); //counter
    vmf->page = page;
```

3 Αφαίρεση Module

Κατά την αφαίρεση του module πρέπει να γίνουν διάφορες εργασίες. Το ποια συνάρτηση θα κληθεί κατά την αφαίρεση του module δηλώνονται με το macro:

```
```

Η συνάρτηση που αναλαμβάνει τις εργασίες ”καθαρισμού” είναι η lunix_exit. Αυτή χρειάζεται να αφαιρέσει τη συσκευή από το σύστημα με κλήση της cdev_del καθώς και να αποδεσμεύσει όσο χώρο έχει δεσμεύσει η συσκευή:

```
static void lunix_exit(void)
{
    dev_t dev = MKDEV(lunix_major, lunix_minor);

    if(device){
```

```
        cdev_del(&device->mydev);
        lunix_reset_dev(device);
        kfree(device);
    }

    unregister_chrdev_region(dev, 1);
    printk(KERN_ALERT "Goodbye\n");
}
```