Στράτη Θεώνη Μαρία - 03118186
Τσαρμποπούλου Μαργαρίτα Ελένη - 03118848

# Αναφορά για την 2η εργαστηριακή άσκηση

Σε αυτή την άσκηση υλοποιησαμε εναν οδηγό συσκευής (driver) για ένα ασύρματο δίκτυο αισθητήρων υπό το λειτουργικό σύστημα Linux.
Το δίκτυο αυτό αποτελείται από επιμέρους αισθητήρες σε συνδεσμολογία mesh ώστε όλοι οι αισθητήρες να αποδίδουν δεδομένα στον κεντρικό σταθμό βάσης. Κάθε ένας από αυτούς λαμβάνει μετρήσεις για τα μεγέθη, φωτεινότητα, τάση της μπαταρίας και θερμοκρασία περιβάλλοντος.
Κατά την διάρκεια της ανάπτυξης του οδηγού, χρησιμοποιήσαμε εικονικό μηχάνημα QEMU-KVM το οποίο αρχικοποιείται με το βοηθητικό utopia.sh bash script που μας δίνεται. Ο σταθμός βάσης λαμβάνει τα πακέτα από τους αισθητήρες και τα αποστέλλει μέσω USB στο εργαστήριο. Η διασύνδεση αυτή υλοποιείται με κύκλωμα serial over USB.
Επειδή δεν είναι δυνατό να διαθέτει κάθε ένας από εμάς ένα τέτοιο δίκτυο αισθητήρων, στο εργαστήριο της σχολής υλοποιείται μια τέτοια διάταξη και τα αποτελέσματα όλων των μετρήσεων εμφανίζονται μαζικά στην εικονική σειριακή συσκευή /dev/ttyUSB1 του υπολογιστή του εργαστηρίου. Στον ίδιο υπολογιστή εγκαθίσταται TCP/IP server ο οποίος μεταδίδει τις μετρήσεις και στη συνέχεια, μέσω του utopia.sh, όλες αυτές ανακατευθύνονται στη S0 σειριακή θύρα του QEMU μηχανήματος (dev/ttyS0).

{Τελος υλοποιησαμε σύστημα εξαγωγής των δεδομένων από την σειριακή θύρα σε ένα σύνολο από επιμέρους συσκευές χαρακτήρων, για τις οποίες έχουμε ορίσει σύμβαση για την ονομασία τους, οι οποίες παράγονται από το βοηθητικό script lunix_dev_nodes.sh σύμφωνα με την ίδια σύμβαση}

## Αρχιτεκτονική του Λογισμικού του Συστήματος

Το σύστημα οργανώνεται σε δύο κύρια μέρη: Το πρώτο αναλαμβάνει τη συλλογή των δεδομένων από το σταθμό βάσης (Lunix line discipline) και την επεξεργασία τους με συγκεκριμένο πρωτόκολλο (Lunix protocol), έτσι ώστε να εξαχθούν οι τιμές των μετρούμενων μεγεθών, οι οποίες κρατούνται σε κατάλληλες δομές προσωρινής αποθήκευσης στη μνήμη (Lunix sensor buffers). Το δεύτερο μέρος αναλαμβάνει να παραλάβει τα δεδομένα από τους προσωρινούς buffers και να τα εξάγει στο χώρο χρήστη στην κατάλληλη μορφή, υλοποιώντας μια σειρά από συσκευές χαρακτήρων.
Το τελικό σύστημα θα λαμβάνει τα δεδομένα των μετρήσεων από το σταθμό βάσης. Αυτά
θα προωθούνται μέσω USB στο υπολογιστικό σύστημα, θα παραλαμβάνονται από ένα φίλτρο, τη διάταξη γραμμής του Lunix, η οποία θα τα προωθεί στο κατάλληλο στρώμα ερμηνείας του πρωτοκόλλου των αισθητήρων. Το συγκεκριμένο είναι υπεύθυνο για την ερμηνεία των bytes
των πακέτων και την αποθήκευση τους στους αντίστοιχους buffers του κάθε αισθητήρα. Ο device driver ανάλογα με το αρχείο που χρησιμοποιείται στο χώρο χρήστη θα ανακτά τα δεδομένα από τους buffers και θα τα παρουσιάζει στην κατάλληλη μορφή.

## SUMMARY

**(0)** Αρχικά φορτώνουμε τον οδηγό στον πυρήνα με την εντολή insmod **./lunix.ko** και καλείται η lunix_module_init() αρχικοποιώντας τους 16 αισθητήρες με return 0, αφού δεν υπάρχουν για την ώρα νέα δεδομένα.
=> lunix_sensors = kzalloc(sizeof(*lunix_sensors) * lunix_sensor_cnt, GFP_KERNEL);
Δέσμευση μνήμης για τις δομές sensor που αποθηκεύουν τα δεδομένα από το line discipline

=> lunix_protocol_init(&lunix_protocol_state);
Αρχικοποίηση της μηχανής καταστάσεων του πρωτοκόλλου του οδηγού μας =>
Αρχικοποίηση τιμών των αισθητήρων (lunix_sensor_struct), της διάταξης γραμμής
**(1)** Δημιουργούμε τους κόμβους(nodes) για καθένα από τους αισθητήρες με χρήση της εντολής **./lunix_dev_nodes.sh** (αρκεί να τους δημιουργήσουμε μια φορά).
Το identifier των αρχείων αυτών είναι το major και το minor number τους. Το major number χρησιμοποιείται από τον kernel για να προσδιορίσει τον οδηγό συσκευής που αφορά ο κόμβος αυτός. Στη συνεχεια επιλέξαμε τον major number 60 ο οποίος είναι δεσμευμένος για πειραματική χρήση. Για τον minor number επιλέγουμε την τιμή : minor = αισθητήρας* 8 + μέτρηση, όπου μέτρηση = τάση μπαταρίας(0), θερμοκρασία(1) και φωτεινότητα(2)
Δημιουργούμε 1 node για την θύρα S0 με minor και major numbers 4 και 64 αντίστοιχα και 3 nodes για καθέναν από τους 16 αισθητήρες με major number 60 και minor sensor*8+value, όπου value {0,1,2}.
**(2)** Στη συνέχεια, τρέχοντας **./lunix-attach /dev/ttyS0**, προσαρτούμε τον driver στη συσκευή S0, η οποία περιέχει όλα τα δεδομένα που χρειαζόμαστε και έτσι αρχίζουν να γεμίζουν οι buffers. Καλείται η lunix_ldisc_receive(), η οποία βρίσκει πόσοι νέοι χαρακτήρες αντιστοιχούν στον κάθε αισθητήρα και εκείνη με τη σειρά της καλεί την lunix_protocol_received_buf() η οποια ελέγχει για εισερχόμενα δεδομένα για ενημέρωση του μηχανήματος κατάστασης πρωτοκόλλου. Αναλόγως το state, καλείται η lunix_protocol_parse_state() η οποια αναλυει το πακέτο εισόδου σύμφωνα στην τρέχουσα κατάσταση, δηλαδη ελέγχει πόσα bytes έχουν διαβαστεί, αν έχει γίνει overflow στον buffer και τερματίζεται με return 1 όταν έχουν διαβαστεί όλα τα δεδομένα καλώντας την set_state() η οποια ρυθμιζει γρήγορα την τρέχουσα κατάσταση και την lunix_protocol_show_packet() η οποια εμφανίζει τα περιεχόμενα ενός εισερχόμενου πακέτου XMesh που έχουν ληφθεί μέχρι στιγμής.
Αργότερα, γίνεται ξανά open με τη lunix_ldisc_open(), για να αναζητήσουμε νέα δεδομένα. Αυτό σημαίνει ότι η line discipline είναι προσαρτημένη στο terminal. Καμία άλλη κλήση στην line discipline για αυτό το tty δεν θα πραγματοποιηθεί μέχρι να ολοκληρωθεί με επιτυχία.
**(3)** Μετά τρέχοντας cat **dev/lunix{x}-{batt,temp,light}** καλειται η αντίστοιχη, από το struct lunix_chrdev_fops(), συνάρτηση, δηλαδή η lunix_chrdev_open()
**(4)** Καλείται η read() επί του αρχείου, η οποία μέσω της δομής lunix_chrdev_fops αντιστοιχεί στην συνάρτηση lunix_chrdev_read() η οποια αν δεν υπάρχουν διαθέσιμα δεδομένα στο ανοιχτό αρχείο
**(5)** καλεί την lunix_chrdev_state_update() η οποία είναι υπεύθυνη για την ενημέρωση των τιμών των μετρήσεων από τους αισθητήρες καθώς και την μορφοποίηση των δεδομένων έτσι ώστε να μπορούν να διαβάζονται σε μορφή χαρακτήρων.
**(6)** Ελέγχει αν υπάρχουν δεδομένα καλώντας την lunix_chrdev_state_needs_refresh() και παίρνει το spinlock του σένσορα για να πάρει τα δεδομένα χωρίς να μπορούν να αλλάξουν
**(7)** και υπολογίζει πόσα bytes να επιστρέψει στον χρήστη και τα αντιγράφει στον καθορισμένο buffer της καταστασης της συσκευής
**(8)** control c στην ./lunix-attach /dev/ttyS0 => καλειται η lunix_ldisc_close()
**(9)** control c στην cat dev/lunix{x}-{batt,temp,light} => καλειται η lunix_chrdev_release()
**(10)** rmmod lunix => καλείται η lunix_module_cleanup() του αρχείου lunix-module.c η οποία καλεί την υλοποιημένη στο lunix-chrdev.c lunix_chrdev_destroy(); η οποία καθαριζει όποιο υπόλειμμα στη μνήμη απο το kernel module μας

# SYSTEM CALLS

**int lunix_chrdev_init (void):**

–Καλείται μόνο κατά την εκτέλεση της εντολής insmod που εισάγει ένα καινούριο module στον κώδικα του πυρήνα

Αρχικοποιούμε τη συσκευή χαρακτήρων με την κλήση της cdev_init() και δεσμεύουμε στον πυρήνα 8 minor numbers για κάθε sensor (128). Στη συνεχεια με την MKDEV() η οποία παίρνει σαν ορίσματα ενα major και ενα minor number και κάνοντας μια λογική πράξη δημιουργεί την νέα συσκευή, η οποία είναι τύπου dev_t. Έπειτα καλώντας την συνάρτηση register_chrdev_region() κάνουμε register το character device μας προκειμένου να μπορούμε να αναφερθούμε σε αυτό με major και minor numbers.

Αφού τα δεσμεύσουμε καλώντας την cdev_add() ενεργοποιούμε αυτές τις συσκευές και στο εξής η συσκευή μας είναι ¨ζωντανή' και ακούει' σε όποια κλήση συστήματος του χώρου χρήστη

Μετά την κλήση καθεμίας απο τις συναρτήσεις αυτές ελέγχουμε για πιθανό σφάλμα και αν υπάρχει ενημερώνουμε με κατάλληλο μήνυμα και κάνουμε την ανάλογη ενέργεια, οπως κανουμε και στις επομενες κλησεις μας.

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

–Καλείται από τον χρήστη κάθε φορά που μία διεργασία ανοίγει ένα ειδικό αρχείο (struct inode) της συσκευής χαρακτήρων για την έναρξη της "επικοινωνίας" του με τη συσκευή. Αυτή η συνάρτηση συσχετίζει το ανοιχτό αρχείο με το σωστό αισθητήρα ανάλογα με το minor number του inode που παίρνει σαν όρισμα και αρχικοποιει μια καινούρια δομή state της συσκευής με τις κατάλληλες τιμές.

Αρχικά καλείται η nonseekable_open() για να γίνουν οι κατάλληλες αρχικοποιήσεις και να ενημερωθεί ο πυρήνας οτι η llseek λειτουργία δεν υποστηρίζεται.

Μετα αρχικοποιουνται οι τιμες του state σχετικες με το είδος του αισθητήρα, του σημαφόρου για παράλληλη πρόσβαση στο struct και αντιστοίχηση του με τον κατάλληλο sensor buffer

state->type = minor%8; κλπ…

Στη συνεχεια δεσμευεται χώρος μνήμης για το lunix_chrdev_state_struct που αποτελεί buffer με την τελευταία μέτρηση και αντιστοιχεί σε κάθε νέο άνοιγμα αρχείου προκειμένου να καθίσταται δυνατή η προσπέλαση του ίδιου αισθητήρα από διαφορετικές διεργασίες.

kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL);

Έπειτα βάζουμε στα πεδία buf_lim και buff_timestamp του dev_state struct τη τιμή 0 αφού όταν ανοίγουμε το ειδικό αρχείο δεν έχει γίνει κάποιο update και ο buffer είναι αρχικά άδειος(τα πεδία αυτά θα τα χρησιμοποιήσουμε στις επόμενες συναρτήσεις)

Αντιστοίχηση του ανοιχτού αρχείου που δίνεται από τον δείκτη παράμετρο *filp με το κατάλληλο state (αρα και sensor) μέσω του πεδίου private_data για μελλοντική χρήση χωρίς να αναζητάμε τους major και minor numbers.

filp->private_data = state

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

–Είναι ένας γρήγορος τρόπος να ελέγξουμε αν υπάρχουν νέα δεδομένα χωρίς να χρειαζόμαστε κλείδωμα.

Ελέγχει αν το timestamp που ειναι αποθηκευμενο στην δομη του state ειναι το ειναι το ιδιο με αυτο το οποιο εχει γραφτει στο κατωτερο επιπεδο του lunix_sensors, δηλαδή την τιμή της μεταβλητής buf_timestamp του ανοιχτού αρχείου με την τιμή της μεταβλητής last_update του είδους μέτρησης του αισθητήρα που επιθυμούμε

Αν είναι τότε η κλήση της επιστρέφει 0 αλλιώς 1.
Αν αυτές οι δύο τιμές είναι ίσες τότε η τελευταία μέτρηση του αισθητήρα έχει ήδη ληφθεί και δεν υπάρχει ακόμα νέα διαθέσιμη μέτρηση άρα επιστρεφει 0. Διαφορετικά έχει γίνει λήψη νέας μέτρησης από τον επιθυμητό αισθητήρα και επιστρεφει 1.
Καλειται οταν καποια διεργασια βρισκεται στις update και read ως συνθήκη για το αν θα κοιμηθει ή οχι

**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 μία διεργασία θέλει να διαβάσει κάποια bytes από το περιεχόμενο ενός ανοιχτού αρχείου.
Αρχικά ανακτουμε τη δομή lunix_chrdev_state_struct και λαμβάνουμε τον state buffer από το πεδίο private_data του filp και από αυτόν τον αντίστοιχό sensor struct και τον τύπο του αισθητήρα.
{Η δομή είναι κοινή για διεργασίες που έχουν προκύψει από κάποιο fork() καθως κληρονομούν τον fd της open και για το λόγο αυτό θα κάνουμε χρήση σημαφόρου για να κλειδώσουμε όταν μια διεργασία μπεί στο κρίσιμο τμήμα.
(δηλαδη, είναι πιθανό μια διεργασία που έχει μπει στο κρίσιμο τμήμα να ‘φάει’ κάποιο σήμα και τότε η εκτέλεση θα κολλήσει αφού θα πέσω σε deadlock απο το οποίο δεν θα μπορώ να βγω)
Στη συνεχεια που γίνεται πρόσβαση στο κρίσιμο τμήμα εκτελείται η συνάρτηση down_interruptible() η οποία δεσμεύει το σημαφόρο του αντίστοιχου ανοιχτού αρχείου και όλες οι άλλες διεργασίες που επιθυμούν να μπουν κλειδώνονται έξω δηλαδή μπλοκάρουν. }
Μετα κοιτάμε αν το αρχείο έχει ανοιχτεί για να διαβαστεί από την αρχή με την if(*fpos==0)
{η fpos δείχνει πόσα bytes έχουν διαβαστεί εώς τώρα από τον χρήστη
Ανανεώνουμε τα δεδομένα μας μόνο όταν το *f_pos είναι 0, είναι γιατί περιμένουμε ο χρήστης να διαβάσει πρώτα όλα τα προηγούμενα
}
Εφόσον αυτό ισχύει, καλούμε σε μια while loop την lunix_chrdev_update() η οποια επιστρέφει -EAGAIN αν δεν υπάρχουν διαθέσιμα νέα δεδομένα και τότε ο σημαφόρος ξεκλειδώνει, η διεργασία ‘πέφτει για ύπνο’ ώστε να μην χρησιμοποιείται την CPU χωρίς λόγο μέχρι να έρθουν νέα δεδομένα και μπαίνει σε μια ουρά προτεραιότητας το οποιο γίνεται με τη συνάρτηση wait_event_interruptible().
Κοιμάται εως ότου να ληφθούν δεδομένα και η needs refresh επιστρέψει ότι υπάρχουν νέα δεδομένα.
Ετσι, λαμβάνουμε με ασφάλεια, και χωρίς να καταναλώνουμε πολλούς πόρους, τα δεδομένα που χρειάζεται να πάρουμε.
Αν τώρα έχει γίνει update των δεδομένων και είναι πλέον σε κατάλληλη μορφή για να μεταφερθούν στο user space, δηλαδή στο user space buffer ελέγχω αρχικά αν ο χρήστης ζήτησε λιγότερα δεδομένα από αυτά που έχω διαθέσιμα και βάζω τον κατάλληλο αριθμό στη μεταβλητή count(επιτρέπω στο χρήστη να ζητήσει και λιγότερα δεδομένα από τα διαθέσιμα). Η μεταφορά των ζητούμενων δεδομένων στο user space buffer γίνεται με κλήση της συνάρτησης copy_to_user(). Η συνάρτηση αυτή αν δεν καταφέρει τελικά να γράψει στο user space buffer τα δεδομένα που ζητήθηκαν τότε επιστρέφει θετική τιμή, η οποία ισούται με τον αριθμό των bytes που δεν κατάφερε να αντιγράψει και τερματίζει με -EFAULT. Αν όμως η μεταφορά των ζητούμενων δεδομένων έγινε επιτυχώς, η copy_to_user() επιστρέφει μηδέν και στο σημείο αυτό θα πρέπει να γίνει και ανανέωση του περιεχομένου της μεταβλητής f_pos(αν έχω φτάσει στο τέλος του buffer πάω από την αρχή ξανά όπως φαίνεται και στο κώδικα). Στο τέλος αποδεσμεύεται ο σημαφόρος αφού πλέον η διεργασία έχει εξέλθει από το κρίσιμο τμήμα.

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

Καλείται κάθε φορά που μια διεργασία θέλει να διαβάσει από ένα ανοιχτό αρχείο, αλλά δεν υπάρχουν καθόλου διαθέσιμα δεδομένα. Παίρνει τα καινούρια δεδομένα και τα μεταφέρει κατάλληλα μορφοποιημένα στον buffer του state ώστε να είναι έτοιμα να περαστούν στο χώρο χρήστη.
{Επειδή μπορεί να αλλάζει την κατάσταση των buffers πρεπει να αποκλείσουμε οποιοδήποτε race condition. Για αυτόν τον λόγο χρησιμοποιούμε εντός της συνάρτησης το κλείδωμα για το sensor struct ενώ κατά τη read() το κλείδωμα για το state buffer(). {Σχετικά με το κλείδωμα, υλοποιούμε το πρόγραμμα χρησιμοποιώντας κλήσεις spin_*. Αυτό θα μπορούσε να δημιουργήσει ζητήματα σε περίπτωση που έρθουν δύο συνεχόμενες διακοπές από το υλικό χωρίς να έχει τερματίσει η πρώτη.}
Το γεγονός αυτό αντιμετωπίζεται με τη χρήση κλήσεων spin_lock και spin_unlock ώστε να μην κολλήσει ποτέ το πρόγραμμα με interrupt που δεν μπορεί να εξυπηρετηθεί λόγω των spinlocks.
Ετσι, κλειδώνουμε με spinlock, παιρνουμε γρηγορα τα δεδομένα που ζήτησε ο χρήστης και κάνουμε update το προσωρινό timestamp}
Πιο αναλυτικα
Μετά το ξεκλείδωμα το αποθηκεύουμε στην δομή του state και επεξεργαζομαστε την τιμή που ζήτησε ο χρήστης ανάλογα αν πρόκειται για raw ή cooked δεδομένα.
Μετα καλειται η lunix_chrdev_needs_refresh() και αν αυτή επιστρέψει 0 τότε επιστρέφει -EAGAIN. Αλλιως αν επιστρέψει 1 σημαίνει ότι έχουν έρθει νέα δεδομένα δηλαδή ότι υπάρχει καινούργιο timestamp από τους sensors, διαφορετικό από αυτό που έχουμε αποθηκευμένο στο state και θα πρέπει να γίνει update. Επεξεργαζομαστε την τιμή που ζήτησε ο χρήστης ανάλογα αν πρόκειται για raw ή cooked δεδομένα. Τα cooked δεδομένα τα χειριζομαστε με αυτον τον τροπο επειδη ο πυρήνας του Linux δεν υποστηρίζει floating point πράξεις διότι η FPU δεν σώζεται αν πάω από user space σε kernel space. Για να μπορέσουμε να κάνουμε το κατάλληλο conversion αρχικά ελεγχουμε τι ειδος τιμης ζητηθηκε και επιστρέφουμε τη τιμή του πεδίου του κατάλληλου lookup_table που ορίζεται από την μεταβλητη value.
Επειτα αλλάζουμε την τιμή του buff_timestamp για να έχουμε πλέον τη νέα χρονική τιμή που έγινε update και να μπορουμε να ελέγχουμε στη συνέχεια αν πρέπει να γίνει κάποιο άλλο update ή οχι.
Επισης αλλάζουμε την τιμή του buff_limit με τα bytes κατάφερε να γράψει η snprintf() στο buffer. Χρησιμοποιουμε την snprintf() διότι είναι υλοποιημένη ώστε να γράφει στο buffer μέχρι LUNIX_CHRDEV_BUFSZ bytes και ποτέ περισσότερα
{
*Raw*: ο χρήστης λαμβάνει τα δεδομένα ακατέργαστα όπως αυτά παράγονται από τους αισθητήρες επιστρέφει στον χρήστη σαν τιμή μέτρησης το index του αντίστοιχου κελιού της μέτρησης στον lookup_table που ορίζεται στο lunix-lookup.h
*Cooked*: το mk lookup tables.c δημιουργεί 3 πίνακες, έναν για κάθε είδος μέτρησης, οι οποίοι κάνουν 1-1 αντιστοίχιση των raw δεδομένων σε cooked τιμές
}

**static int lunix_chrdev_release(struct inode *inode, struct file *filp):**
Σε ένα ανοιχτό αρχείο μπορεί να έχουν πρόσβαση παραπάνω από μία διεργασίες (όταν πχ κάνουμε fork και διαβάζουμε από ένα αρχείο ). Η συνάρτηση lunix_chrdev_release καλείται κάθε φορά που ένα ανοιχτό αρχείο κλείνει από την τελευταία διεργασία που έχει πρόσβαση σε αυτό ή όταν αυτή τερματίζει την εκτέλεση της. Στην περίπτωση αυτή τα δεδομένα και η κατάσταση του ανοιχτού αρχείου δεν μας ενδιαφέρουν πλέον. Το στιγμιότυπο της δομής file καταστρέφεται αυτόματα ενώ εμείς απελευθερώνουμε τον χώρο στη μνήμη που είχαμε δεσμεύσει για τα ειδικά δεδομένα του συγκεκριμένου αρχείου κατά το άνοιγμα του με την εντολή kfree(filp->private_data).

{kernel panic => είχαμε κάνει global το instance του chrdev state struct και στην συνάρτηση release, κάναμε free αυτό το global instance}

**void lunix_chrdev_destroy(void):**

Η συνάρτηση αυτή εκτελείται κάθε φορά που γίνεται αφαίρεση του driver από τον πυρήνα (εντολή rmmod). Απο οτι καταλαβαμε κάνει την αντίστροφη διαδικασία από την lunix_chrdev_init αφου με την εντολή cdev_del(&lunix_chrdev_cdev) διαγράφει τη συσκευή χαρακτήρων από τον πυρήνα, ενώ με τις εντολές dev_no = MKDEV(LUNIX_CHRDEV_MAJOR, 0) και unregister_chrdev_region (dev_no, lunix_minor_cnt) απελευθερώνει τους device numbers που είχε δεσμεύσει η συσκευή χαρακτήρων κατά την αρχικοποίηση της.

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

–Επιταχύνει την πρόσβαση σε δεδομένα της συσκευής από το χρήστη αφού εξαλείφει το overhead των συνεχόμενων κλήσεων συστήματος και της αντιγραφής δεδομένων από τον χώρο πυρήνα στον χώρο χρήστη

Ο χρήστης έχει πρόσβαση σε κάποιες σελίδες μνήμης που έχει πρόσβαση και ο driver της συσκευής, οπότε η διαδικασία απόκτησης δεδομένων απλοποιείται σημαντικά

Επιλέξαμε να επιστρέφει μία σελίδα στο χρήστη για να ειναι πιο απλο αφου είναι αρκετή για τα δεδομένα που θέλει ο χρήστης

Χρησιμοποιεί remap_pfn_range και επιστρέφει μία σελίδα στο χρήστη

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

Δίνει την δυνατότητα στον χρήστη να επιλέξει ανάμεσα σε 2 μορφές εξόδου των μετρήσεων που μας δίνει ο αισθητήρας

Αρχικά ελέγχεται αν ο κωδικός της εντολής που δόθηκε αφορά όντως τη συγκεκριμένη συσκευή χαρακτήρων και επίσης αν είναι μία από τις έγκυρες που επιτρέπονται

Το να επιλέξει ανάμεσα σε 2 μορφές εξόδου γίνεται μέσω του ορίσματος που μπορεί να πάρει την τιμή 0 ή 1 που αντιστοιχεί σε format εξόδου Raw ή Cooked αντίστοιχα

*Raw*: επιστρέφει στον χρήστη σαν τιμή μέτρησης το index του αντίστοιχου κελιού της μέτρησης στον lookup_table που ορίζεται στο lunix-lookup.h

*Cooked*: το mk lookup tables.c δημιουργεί 3 πίνακες, έναν για κάθε είδος μέτρησης, οι οποίοι κάνουν 1-1 αντιστοίχιση των raw δεδομένων σε cooked τιμές

επιστρέφει το αντίστοιχο entry αυτού του πίνακα σε μορφή όπως την εκφώνηση xx.yyy

Πράξεις για να βρεθούν το ακέραιο μέρος και το κλασματικό του αριθμού

# ΔΙΑΦΟΡΑ

lunix_sensor_struct (lunix.h): Μια δομή που αντιπροσωπεύει έναν αισθητήρα hardware και σελίδες που περιέχουν τις πιο πρόσφατες μετρήσεις που λάβαμε.
Τα δεδομένα που προέρχονται από τον σταθμό βάσης αφού ερμηνευθούν από το κατάλληλο πρωτόκολλο αποθηκεύονται σε προσωρινούς buffers. Κάθε τέτοιος buffer αντιστοιχεί σε έναν συγκεκριμένο αισθητήρα. Επομένως, η παραπανω δομή είναι απαραίτητη γι' αυτό το σκοπό.
-struct lunix_msr_data_struct: Ένας αριθμός σελίδων, μία για κάθε μέτρηση.
Μπορούν να αντιστοιχιστούν στον χώρο χρηστη.
-spinlock_t lock: χρησιμοποιείται για να επιβεβαιώσει τον αμοιβαίο αποκλεισμό μεταξύ της serial line discipline και του character device driver
-wait_queue_head_t wq: Μια λίστα διαδικασιών που περιμένουν να ξυπνήσουν όταν αυτός ο αισθητήρας έχει ενημερωθεί με νέα δεδομένα

lunix_chrdev_state_struct (lunix-chrdev.h): Μας πληροφορεί για το state κάθε συσκευής που ανοίγει από κάποια διεργασία. Υπάρχει ένας buffer (buf_lim) που θα αποθηκεύονται τα δεδομένα από τους sensor-buffers και από εκεί θα μεταφέρονται στο χώρο χρήστη ο οποιος μας δείχνει πόσες θέσεις του buffer έχουν χρησιμοποιηθεί. Επισης υπαρχει μια μεταβλητή που μας ενημερώνει για το ποια ήταν η τελευταία ανανέωση που έγινε στον buffer αυτόν

struct lunix_sensor_struct * lunix_sensors: Ο pointer lunix_sensors είναι ένας πίνακας από 16 pointers που ο καθένας αντιστοιχεί στην παραπάνω δομή για τον κάθε αισθητήρα. Η δομή αυτή είναι αρκετή για να μας επιτρέψει να ασχοληθούμε με το τι θα συμβεί όταν κάνουμε load το module μας στον πυρήνα

lprotocol.c: επεξεργασία των δεδομένων που έχουν συλλεχθεί από το σταθμό βάσης με συγκεκριμένο πρωτόκολλο

ldisc.c: The low-level physical driver and the tty driver handle the transfer of data to and from the hardware, while line disciplines (ldisc) are responsible for processing the data and transferring it between kernel space and user space.

WARN_ON: macro εντολή για να βρούμε bugs στο δικό μας κώδικα ή προβλήματα στο hardware. Σταματάει την εκτέλεση του προγράμματος, παράγει διαγνωστικά μηνύματα, ένα stack trace και μία module list
WARN_ON ( !(sensor = state->sensor)); -----> there is no data available right now, try again later

void * kmalloc (size_t size,gfp_t flags); -> kmalloc(sizeof(struct lunix_chrdev_state_struct), GFP_KERNEL) ---> Τα περισσότερα από τα API εκχώρησης μνήμης χρησιμοποιούν σημαίες GFP για να εκφράσουν πώς πρέπει να εκχωρηθεί αυτή η μνήμη.

void kfree (const void * objp); --> kfree(filp->private_data); --> memory allocation

int sema_init(sema_t * sp, unsigned int count, int type, void * arg); ---> sema_init(&state->lock,1);

int down_interrupt(struct semaphore *sem) ---> down_interruptible(&state->lock) —> Αυτό σημαίνει ότι εάν ο σηματοφόρος δεν είναι διαθέσιμος, η αντίστοιχη διαδικασία θα μπει στην ουρά αναμονής του semaphore.
up_interruptible(&state->lock)

void **cdev_init** (struct cdev * cdev, const struct file_operations * fops); -> cdev_init(&lunix_chrdev_cdev, &lunix_chrdev_fops); —> initialize a cdev structure (ορίζεται στην πρώτη γραμμή του lunix-chrdev.c μετά τα import)

semaphore lock: Χρησιμοποιούμε semaphore lock για την ασφάλεια των δεδομένων της δομής state σε περίπτωση που περισσότερες από μίας διεργασίες έχουν κοινό linux driver state.
Για παράδειγμα ο fd μπορεί να κληρονομηθεί από τον πατέρα στα παιδιά μετά τη fork και έτσι στην προσπάθεια του πατέρα και των παιδιών να διαβάσουν καλώντας την read προκύπτει race condition

spinlocks vs semaphore: με τα spinlocks κάνουμε busy wait πράγμα το οποίο είναι πολύ κακό και μας χαλάει πολύτιμη υπολογιστική ισχύ καθώς ο επεξεργαστής λουπάρει συνεχώς ανάμεσα σε μια εντολή jump στην ίδια διεύθυνση. Αυτό αποφεύγεται με τους σημαφόρους που βάζουν την διεργασία που αποτυχαίνει να πάρει το κλείδωμα να κάνει sleep και συνεπώς ο χρονοδρομολογητής πάει σε μία άλλη διεργασία η οποία είναι ready και της δίνει το δικαίωμα να τρέξει

spinlocks:
spin_lock&spin_lock //Παίρνουν ως όρισμα δείκτη *lock, που δείχνει ποιας μεταβλητής το spinlock κλειδώνουμε ή αποδεσμεύουμε.

*spinlock_t lock*
Κάποιος μπορεί να ρωτήσει γιατί χρησιμοποιούμε κλείδωμα και όχι κάποιο σεμαφόρο/mutex. Γιατί δε θέλουμε με τίποτα να μεταφερθούμε σε sleep mode αφού όταν ανανεώνονται τα δεδομένα στους sensor buffers βρισκόμαστε σε interrupt context και δεν υπάρχει κάποια διεργασία για να κοιμηθεί. Επομένως, τα spin locks είναι αυτά που χρειαζόμαστε γιατί χρησιμοποιούν την μέθοδο του polling για να πάρουν κάποιο lock. Πρέπει όμως να είμαστε αρκετά προσεκτικοί και να μην κρατάμε το lock για μεγάλο χρονικό διάστημα, γιατί έτσι κάποιος άλλος που προσπαθεί να το πάρει θα σπαταλάει αρκετό χρόνο από την CPU. Επιπλέον είναι απαραίτητο να απενεργοποιήσουμε τα interrupts στην περίπτωση που λαμβάνουμε καινούργια δεδομένα από τους sensor buffers. Δε θέλουμε να έρθει κάποιο interrupt που θα θέλει να κάνει ανανέωση των δεδομένων όσο έχουμε το lock γιατί έτσι οι sensors δε θα μπορέσουν ποτέ να πάρουν το κλείδωμα και θα προκύψει deadlock. 6 Αν, λοιπόν, πάνε όλα καλά και πάρουμε τα δεδομένα μας στη συνέχεια τα μορφοποιούμε κάνοντας χρήση των ειδικών πινάκων, ανανεώνουμε το timestamp και μεταφέρουμε τα δεδομένα μας στον buffer.

*spin_lock(&sensor->lock)*
Χρησιμοποιούμε spinlocks για να αποφύγουμε πιθανό race condition
Αν δεν είχαμε το κλείδωμα πιθανόν όσο διαβάζαμε τιμές (last_update, values/μετρήσεις) που αφορούν στον κάθε αισθητήρα ταυτόχρονα αυτές να ανανεώνονται και επομένως να λάβουμε λανθασμένες τιμές

Κάθε φορά που είτε διαβάζουμε είτε επεξαργαζόμαστε τα πεδία της δομής του σένσορα πρέπει να κλειδώνουμε την πόρτα μας
Επίσης, δεν θέλουμε να μπούμε σε sleep mode και για αυτο δεν χρησιμοποιούμε κάποιο σεμαφόρο/mutex αφού όταν ανανεώονται τα δεδομένα στους sensor buffers βρισκόμαστε σε interrupt context και δεν υπάρχει κάποια διεργασία για να κοιμηθεί
Τα spin locks χρησιμοποιούν την μέθοδο του polling για να πάρουν κάποιο lock
Αρχικά κλειδώνουμε το spinlock του αντίστοιχου αισθητήρα με την εντολή spin_lock(&sensor->lock), η οποία πριν το κλειδώσει απενεργοποιεί τα interrupts στη συγκεκριμένη CPU. Αυτό είναι απαραίτητο σε περίπτωση που ο driver εκτελείται σε ένα uniprocessor σύστημα με μη-διακοπτή χρονοδρομολόγηση (non-preemptive scheduling). Στην περίπτωση αυτή υπάρχει το ενδεχόμενο μία διεργασία να κλειδώσει το spinlock ενός αισθητήρα και κατά την διάρκεια εκτέλεσης του κρίσιμου τμήματος (πριν απελευθερώσει το spinlock ) να συμβεί κάποιο interrupt από τον συγκεκριμένο αισθητήρα (επειδή πχ ήρθε νέα μέτρηση). Τότε η διεργασία φεύγει από τη μοναδική CPU του συστήματος για να εκτελεστεί ο interrupt handler.
Επειδή έχουμε uniprocessor σύστημα, μη-διακοπτή χρονοδρομολόγηση και χρήση spinlock , ο interrupt handler δεν θα αφήσει ποτέ την CPU μέχρι να μπορέσει να κλειδώσει το spinlock και η συνάρτηση lunix_chrdev_state_ update δεν θα μπορέσει ποτέ να μπει στην CPU ώστε να ξεκλειδώσει το spinlock, καθώς αυτή είναι μονίμως κατειλημμένη. Έτσι καταλήγουμε σε deadlock του συστήματος.

*spin_unlock(&sensor->lock);*
Μόλις πάρουμε τα δεδομένα κάνουμε unlock το spinlock, διότι δε θέλουμε να κρατάμε σε lock για πολλή ώρα, γιατί μπορεί να το χρειαστεί και σε άλλο σημείο του κώδικα και έτσι να σπαταλάται χρόνος από τη CPU

Ο λόγος έιναι οτι τα spinlocks όταν ενα τμήμα κώδικα δεν μπορεί να πάρει το κλείδωμα επειδή είναι κατειλμμένο, δεν θα αφήσουν τη cpu αλλά θα προσπαθούν συνέχεια μέχρι να τα καταφέρουν, δηλαδή θα κάνουν busy-wait. Εμένα με βολεύει αυτό διότι η συνάρτηση που ανανεώνει τα δεδομένα τρέχει σε interupt context όταν πάρει νεά δεδομένα γεγονός το οποίο καθιστά αναγκαία τα spinlocks για να κάνω το κλειδώμα διότι αν έκανα χρήση κάποιου mutex είναι πολυ πιθανό να έπεφτα σε deadlock απο το οποίο δεν θα έφευγα ποτέ.