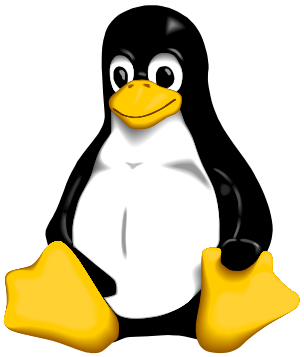
ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Οδηγός Ασύρματου Δικτύου Αισθητήρων στο Λειτουργικό Σύστημα Linux

Μάρτιος 2018



Έκδοση εγγράφου: 27/3/2018 12:17:00 πμ
Επιμέλεια: Ε. Κούκης, vkoukis@cslab.ece.ntua.gr

1. Εισαγωγή

Το παρόν φυλλάδιο περιέχει τις βασικές πληροφορίες που θα χρειαστείτε για την κατασκευή του Lunix:TNG, ενός απλού οδηγού συσκευής για το λειτουργικό σύστημα Linux, στο πλαίσιο της εργαστηριακής σας άσκησης. Στόχος του είναι η παρουσίαση των βασικών σχεδιαστικών αρχών του πυρήνα και του τρόπου αλληλεπίδρασής του με τους οδηγούς συσκευών (device drivers). Πολλές από τις πληροφορίες που περιέχει προέρχονται από το βιβλίο "Linux Device Drivers", 3rd Edition, του εκδοτικού οίκου O'Reilly (βιβλιογραφική αναφορά [1]). Το βιβλίο είναι διαθέσιμο ελεύθερα στο δίκτυο και αποτελεί μια εξαιρετική πηγή γνώσης σχετικής με την κατασκευή οδηγών για τον πυρήνα του Linux και τον ίδιο τον πυρήνα γενικότερα. Οι πληροφορίες που περιέχονται στο παρόν φυλλάδιο είναι ακριβείς για την έκδοση 2.6.37.4 του πυρήνα Linux.

2. Οργάνωση ενός σύγχρονου λειτουργικού συστήματος

2.1. Χώροι πυρήνα και χρήστη

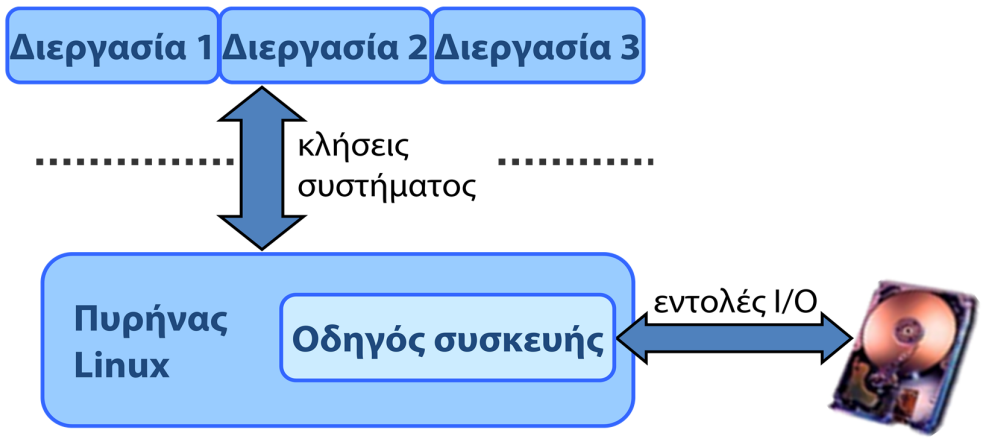
Το Linux είναι ένα σύγχρονο λειτουργικό σύστημα ανοιχτού κώδικα, που βασίζεται στις βασικές σχεδιαστικές αρχές και την παράδοση του Unix. Αναπτύχθηκε αρχικά από τον Linus Torvalds για τον επεξεργαστή i386 της Intel αλλά έχει μεταφερθεί έως σήμερα σε πολλές διαφορετικές αρχιτεκτονικές (IA-64, MIPS, Alpha, SPARC, PowerPC κ.ά.).

Το Linux προσφέρει τη δυνατότητα ταυτόχρονης *προστατευμένης* εκτέλεσης πολλών διαφορετικών διεργασιών, που ανήκουν σε διαφορετικούς χρήστες, είναι δηλαδή multi-tasking και multi-user. Με τον όρο «προστατευμένη» εκτέλεση εννοείται ότι κάθε διεργασία εκτελείται σε απομόνωση από τις υπόλοιπες, σε δικό της εικονικό χώρο διευθύνσεων και χωρίς καμία δυνατότητα άμεσης πρόσβασης στο υλικό (hardware) του υπολογιστικού συστήματος. Έτσι, πιθανή δυσλειτουργία της δεν επηρεάζει την εξέλιξη των υπολοίπων που εκτελούνται στο σύστημα. Επιπλέον, κάθε διεργασία ανήκει σε συγκεκριμένο χρήστη, οπότε έχει και ανάλογα δικαιώματα πρόσβασης σε πόρους (π.χ. αποθηκευμένα δεδομένα στο σύστημα αρχείων, περιφερειακές συσκευές) του συστήματος.

Το λειτουργικό σύστημα διαχειρίζεται τους πόρους του συστήματος και συντονίζει την πρόσβαση των διεργασιών σε αυτούς. Το Linux είναι ένα μονολιθικό λειτουργικό

σύστημα, όπως και τα περισσότερα συστήματα που βασίζονται στο Unix: οι περισσότερες λειτουργίες του υλοποιούνται σε ένα πρόγραμμα, που ονομάζεται *πυρήνας* του Λ.Σ., το οποίο εκτελείται ολόκληρο σε ένα ενιαίο χώρο διευθύνσεων και έχει πλήρη, ελεύθερη πρόσβαση στο υλικό *(προνομιούχος κατάσταση* – privileged mode). Ο πυρήνας του Linux αναλαμβάνει το σύνολο σχεδόν των λειτουργιών που χαρακτηρίζουν ένα σύγχρονο Λ.Σ.: Διαχείριση CPU, χρονοδρομολόγηση διεργασιών, διαχείριση μνήμης, διαχείριση συσκευών I/O, διαχείριση συστημάτων αρχείων, διαδικτύωση. Στον πυρήνα του Linux βρίσκονται επίσης οι οδηγοί συσκευών (device drivers), που επιτρέπουν τον έλεγχο των διάφορων περιφερειακών συσκευών.

Γίνεται έτσι φανερό ότι, υπάρχουν δύο εντελώς χωριστά *επίπεδα εκτέλεσης* κώδικα, που αναφέρονται τόσο σε διαφορετικούς χώρους μνήμης όσο και σε διαφορετικά δικαιώματα πρόσβασης στο υλικό: από τη μία είναι ο *χώρος χρήστη* (userspace), στον οποίο ζουν και εκτελούνται οι διεργασίες χρήστη και από την άλλη είναι ο *χώρος πυρήνα* (kernelspace), στον οποίο ζει ο κώδικας του πυρήνα. Η διάκριση των δύο χώρων και των διαφορετικών δικαιωμάτων πρόσβασης επιβάλλεται με μηχανισμούς της CPU.



Σχήμα 1: Οι κλήσεις συστήματος ως μηχανισμός επικοινωνίας χώρων πυρήνα-χρήστη

Εφόσον ο κώδικας στο χώρο πυρήνα εκτελείται τόσο περιορισμένα, είναι αναγκαίος ένας μηχανισμός μέσω του οποίου μια διεργασία μπορεί να ζητήσει από τον πυρήνα να ολοκληρώσει εκ μέρους της λειτουργίες που η ίδια δεν έχει δικαίωμα να πραγματοποιήσει: Αυτός είναι ο μηχανισμός των *κλήσεων συστήματος* (system calls), ο οποίος λειτουργεί ως εξής: κατά την εκτέλεσή της, η διεργασία ετοιμάζει και υποβάλλει κατάλληλη αίτηση στο λειτουργικό σύστημα. Τότε, η CPU ξεκινά να

εκτελεί κώδικα πυρήνα, ο οποίος ελέγχει την ορθότητα των δεδομένων εισόδου και – αν η διεργασία έχει τα ανάλογα δικαιώματα – πραγματοποιεί τη ζητούμενη λειτουργία (π.χ. λειτουργίες I/O με το υλικό) και επιστρέφει τα αποτελέσματά της πίσω στον χώρο χρήστη (Σχήμα 1). Ο έλεγχος τότε επανέρχεται σε κώδικα της διεργασίας.

Ας δούμε ένα παράδειγμα: μια διεργασία χρειάζεται να ανοίξει το αρχείο `/home/user/file1` στο δίσκο για ανάγνωση. Εκτελεί λοιπόν την κλήση συστήματος `open("/home/user/file1", O_RDONLY)`, οπότε ο έλεγχος περνάει στον χώρο πυρήνα. Ο πυρήνας εκτελεί μια σειρά ελέγχων ανάμεσα στους οποίους είναι ότι το αρχείο πράγματι υπάρχει και ότι ο χρήστης με τα δικαιώματα του οποίου τρέχει η διεργασία έχει δικαίωμα ανάγνωσής του, κατασκευάζει κατάλληλες δομές στη μνήμη του («ανοίγει» το αρχείο) και επιστρέφει μια αριθμητική τιμή ("file descriptor") μέσω της οποίας η διεργασία μπορεί πλέον να αναφέρεται σε αυτό.

Έως τώρα είχατε ασχοληθεί περισσότερο με τον προγραμματισμό σε χώρο χρήστη. Στην άσκηση αυτή καλείστε να γράψετε κώδικα προορισμένο να εκτελείται σε χώρο πυρήνα, ως τμήμα ενός οδηγού για μια πραγματική περιφερειακή συσκευή. Ο κώδικας αυτός θα ενσωματώνεται στον πυρήνα του Linux ως *kernel module*.

2.2. Linux kernel modules

Το Linux υποστηρίζει τη δυναμική εισαγωγή νέου κώδικα στο χώρο διευθύνσεων του πυρήνα, μέσω του μηχανισμού των *τμημάτων πυρήνα* (kernel modules). Τα modules είναι τμήματα κώδικα τα οποία ενσωματώνονται με τον πυρήνα κατά τη λειτουργία του και επεκτείνουν τις δυνατότητές του. Έτσι, είναι δυνατό για παράδειγμα ο οδηγός συσκευής για τη μονάδα δισκέτας να φορτωθεί ως module στον πυρήνα, μόλις υπάρξει ανάγκη να ανακτηθούν τα περιεχόμενα μιας δισκέτας και να αφαιρεθεί αργότερα, όταν πλέον δεν είναι απαραίτητος.

Εφόσον κάθε τμήμα κώδικα που βρίσκεται στον πυρήνα έχει ελεύθερη πρόσβαση στη μνήμη και στο hardware, είναι ανάγκη ο κώδικας του πυρήνα να είναι όσον το δυνατό απαλλαγμένος από προγραμματιστικά λάθη, εφόσον δεν υπάρχει δίχτυ προστασίας: Σε αντίθεση με κώδικα που εκτελείται ως μία διεργασία κάτω από ένα Λ.Σ., όπου πιθανή δυσλειτουργία οδηγεί στον τερματισμό της διεργασίας από το Λ.Σ. χωρίς να

επηρεάζονται οι υπόλοιπες[1], εάν δυσλειτουργήσει κώδικας του πυρήνα, οι συνέπειες μπορεί να είναι από το να «κολλήσει» ο υπολογιστής έως το να υπάρξει σοβαρή βλάβη στο σύστημα αρχείων ή σε συσκευές υλικού. Αυτός είναι και ο λόγος για τον οποίο η εισαγωγή kernel modules στο χώρο μνήμης του πυρήνα επιτρέπεται μόνο στο διαχειριστή του συστήματος, το χρήστη root.

3. Οδηγοί συσκευών στο Linux

3.1. Ρόλος των οδηγών συσκευών

Ένα σύγχρονο λειτουργικό σύστημα χρειάζεται να υποστηρίξει πολλές διαφορετικές κατηγορίες περιφερειακών συσκευών (π.χ. κάρτες ήχου, κάρτες γραφικών, προσαρμογείς δικτύου, ελεγκτές μέσων αποθήκευσης) αλλά και πολλές διαφορετικές συσκευές σε κάθε κατηγορία (π.χ. προσαρμογείς δικτύου διαφορετικών κατασκευαστών). Κάθε μία από αυτές ελέγχεται πιθανά με διαφορετικές εντολές, οπότε χρειάζεται ακριβής γνώση του τρόπου προγραμματισμού της για τη σωστή λειτουργία της και ειδικός κώδικας για το χειρισμό της.

Προφανώς δεν είναι δυνατό να απαιτούνται αλλαγές στον κώδικα των εφαρμογών των χρηστών για να υποστηρίζονται νέες περιφερειακές συσκευές που αναπτύσσονται και κυκλοφορούν. Ευτυχώς, το λειτουργικό σύστημα *αποκρύπτει* τις λεπτομέρειες του ελέγχου των συσκευών, πίσω από τον μηχανισμό των κλήσεων συστήματος. Ο κώδικας οργανώνεται σε επίπεδα, με τις εφαρμογές να χρησιμοποιούν υψηλότερου επιπέδου *αφαιρέσεις* (abstractions), όπως «αρχείο», ή «TCP/IP socket» για να επικοινωνήσουν με το έξω κόσμο.

Ωστόσο και ο ίδιος ο κώδικας του πυρήνα είναι σε μεγάλο βαθμό ανεξάρτητος από τη συσκευή που χρησιμοποιείται κάθε φορά, ακριβώς γιατί ο πυρήνας είναι οργανωμένος σε στρώματα: κάποια είναι ανεξάρτητα από το υλικό (π.χ. ο χρονοδρομολογητής, τα συστήματα αρχείων, η υποστήριξη TCP/IP), άλλα όχι.

Οι λεπτομέρειες του προγραμματισμού και ελέγχου μιας συσκευής περικλείονται μέσα σε τμήμα κώδικα που ονομάζεται *οδηγός συσκευής* και έχει συγκεκριμένο τρόπο (προγραμματιστική διεπαφή – interface) επικοινωνίας με τον υπόλοιπο πυρήνα. Έτσι μπορεί να αντικατασταθεί όταν υπάρξει ανάγκη (π.χ. όταν προστεθεί μια κάρτα

[1] Στο σημείο αυτό αξίζει να αναφερθεί το αγαπημένο μήνυμα όλων: “Segmentation fault”

δικτύου στο σύστημα που προγραμματίζεται με διαφορετικό τρόπο), χωρίς να χρειάζονται αλλαγές στα υπόλοιπα τμήματα του πυρήνα (π.χ. στο στρώμα δικτύου IP).

Ωστόσο, δεν μπορούν *όλοι* οι οδηγοί συσκευών να αντιμετωπιστούν με τον ίδιο τρόπο από τον πυρήνα, γιατί δεν ταιριάζει η ίδια προγραμματιστική διεπαφή σε όλες τις συσκευές. Έχει νόημα να αντιμετωπίζονται όλες οι κάρτες δικτύου Ethernet με τον ίδιο τρόπο (σε τελική ανάλυση, όλες στέλνουν και λαμβάνουν πλαίσια Ethernet προς και από το καλώδιο δικτύου), αλλά δεν μπορούν να μπουν στην ίδια κατηγορία με έναν σκληρό δίσκο που διαβάζει και γράφει τυχαία μπλοκ ή με ένα πληκτρολόγιο.

Ο πυρήνας λοιπόν ομαδοποιεί τους οδηγούς συσκευών σε ένα μικρό αριθμό από κατηγορίες. Μέσα σε μία κατηγορία, όλοι οι οδηγοί υλοποιούν το ίδιο interface και αντιμετωπίζονται με όμοιο τρόπο. Έτσι, για να προστεθεί υποστήριξη για μια νέα συσκευή υλικού, χρειάζεται να επιλεγεί μία από τις υπάρχουσες κατηγορίες, στην οποία ταιριάζει καλύτερα η λειτουργία της και να γραφεί κατάλληλος οδηγός συσκευής για το αντίστοιχο interface. Τα παρακάτω θα γίνουν καλύτερα αντιληπτά στη συνέχεια, όπου αναλύονται οι βασικότεροι τύποι οδηγών συσκευών στο Linux.

3.2. Κατηγορίες οδηγών στο Linux

Ο πυρήνας του Linux διακρίνει τρεις κύριες κατηγορίες οδηγών συσκευών (βλ. και [1], σελ. 6). Για κάθε μία προβλέπεται ανάλογο interface, δηλαδή ένας συνδυασμός δομών δεδομένων και λειτουργιών (ουσιαστικά, συναρτήσεων C) που χρειάζεται να υλοποιεί ο οδηγός.

- *Συσκευές χαρακτήρων:* Στην κατηγορία αυτή ανήκουν συσκευές που αντιμετωπίζονται από τον πυρήνα ως ακολουθίες χαρακτήρων. Η προσπέλαση σε αυτές γίνεται με ένα interface που θυμίζει πρόσβαση σε αρχεία στο δίσκο: οι κυριότερες λειτουργίες που υλοποιούνται από τον οδηγό συσκευής είναι η λειτουργία «διάβασε δεδομένα», αντίστοιχη της `read()` από αρχείο, η «γράψε δεδομένα», αντίστοιχη της `write()`, και η «μετακίνησε το σημείο ανάγνωσης/εγγραφής», αντίστοιχη της `lseek()`. Οι κυριότερες συσκευές υλικού που ταιριάζουν σε αυτό το μοντέλο είναι σειριακές και παράλληλες θύρες επικοινωνίας, ποντίκια, τερματικά χρηστών, κάρτες ήχου, ταινίες για αντίγραφα εφεδρείας. Τι πραγματικά σημαίνουν οι λειτουργίες ανάγνωσης/εγγραφής εξαρτάται από την ίδια τη συσκευή: «διάβασε/γράψε δεδομένα» από/προς μία σειριακή θύρα σημαίνει αποστολή και λήψη bytes,

«διάβασε/γράψε δεδομένα» από και προς μία κάρτα ήχου σημαίνει «κάνε δειγματοληψία του σήματος από το μικρόφωνο» και «αυτά τα bytes είναι δείγματα PCM, στείλε το αντίστοιχο αναλογικό σήμα στα ηχεία», αντίστοιχα.

- *Συσκευές μπλοκ:* Πρόκειται για συσκευές στις οποίες η βασική μονάδα I/O είναι το *μπλοκ* δεδομένων, μια ομάδα σταθερού αριθμού bytes. Κάθε συσκευή μοντελοποιείται ως μια συλλογή από αριθμημένα μπλοκ, στα οποία μπορεί να γίνει αναφορά με τυχαία σειρά. Το interface με τον πυρήνα είναι *εντελώς* διαφορετικό από αυτό των συσκευών χαρακτήρα: ο πυρήνας χρησιμοποιεί τον οδηγό για να πραγματοποιήσει δύο κύριες λειτουργίες: τη λειτουργία «διάβασε το μπλοκ με αριθμό `n` και γράψε τα δεδομένα του στη θέση μνήμης `buf`» και τη λειτουργία «πάρε τα δεδομένα που βρίσκονται στη θέση μνήμης `buf` και γράψε τα στο μπλοκ `n`». Ως συσκευές μπλοκ μοντελοποιούνται συνήθως σκληροί δίσκοι ATA/SATA/SCSI και μονάδες CD/DVD-ROM/DVD-RW.

- *Συσκευές δικτύου:* Ως συσκευές δικτύου μοντελοποιούνται περιφερειακά τα οποία μπορούν να στείλουν και να λάβουν πακέτα δεδομένων από και προς άλλα υπολογιστικά συστήματα. Η κατηγορία αυτή περιλαμβάνει πραγματικές, φυσικές, κάρτες δικτύου (π.χ. `eth0`, `eth1` για τον πρώτο και δεύτερο προσαρμογέα δικτύου Ethernet) αλλά και εικονικούς προσαρμογείς δικτύου, όπως τους `lo` και `ppp0` που χρησιμοποιούνται για πακέτα που το μηχάνημα στέλνει στον εαυτό του και για πακέτα που ανταλλάσσονται πάνω από σειριακές γραμμές μέσω του πρωτοκόλλου PPP, αντίστοιχα.

3.3. Πρόσβαση σε συσκευές μέσω ειδικών αρχείων

Οι δύο πρώτες κατηγορίες της προηγούμενης παραγράφου, οι συσκευές χαρακτήρων και οι συσκευές μπλοκ δηλαδή, είναι προσβάσιμες από προγράμματα χρήστη απευθείας, μέσω ειδικών αρχείων («κόμβων») στο σύστημα αρχείων (βλ. και [1], σελ. 43). Όπως δηλαδή μια εφαρμογή μπορεί να ανοίξει το αρχείο `/home/users/file1` για ανάγνωση και εγγραφή, οπότε τα δεδομένα ανταλλάσσονται με το σύστημα αρχείων, έτσι μπορεί να ανοίξει και μια σειρά από ειδικά αρχεία, ώστε να χρησιμοποιήσει μια συσκευή χαρακτήρων ή μια συσκευή μπλοκ απευθείας.

Τα ειδικά αυτά αρχεία δεν περιέχουν δηλαδή δεδομένα που αποθηκεύονται στο σύστημα αρχείων, αλλά αποτελούν πύλες πρόσβασης σε συσκευές του συστήματος, μέσω των καθιερωμένων κλήσεων συστήματος `open()`, `read()`, `write()`, `close()`

κλπ. Λόγω της ειδικής τους σημασίας βρίσκονται συνήθως αποθηκευμένα με κατάλληλα ονόματα κάτω από τον κατάλογο `/dev`. Αυτό που τα κάνει ειδικά ωστόσο δεν είναι ούτε η θέση τους, ούτε το όνομά τους, παρόλο που συνήθως αυτό ακολουθεί συγκεκριμένη σύμβαση ώστε να ξέρουν οι εφαρμογές πού μπορούν να τα βρουν. Αν έχετε εγκατεστημένο ένα σύστημα Linux, τώρα είναι μια καλή ώρα να δείτε τα περιεχόμενα του καταλόγου `/dev`:[2]

```
$ ls –l /dev
crw-rw---- 1 root audio   14,  3 2008-05-21 20:53 /dev/dsp
crw-rw-rw- 1 root root     1,  3 2008-05-21 20:52 /dev/null
crw-rw-rw- 1 root root     1,  8 2008-05-21 20:52 /dev/random
crw------- 1 root root     4,  1 2008-05-21 22:42 /dev/tty1
crw-rw---- 1 root dialout  4, 64 2008-05-21 20:52 /dev/ttyS0
crw-rw---- 1 root dialout  4, 65 2008-05-21 20:52 /dev/ttyS1
… πολλές ανάλογες εγγραφές…
brw-rw---- 1 root disk     8,  0 2008-05-21 20:53 /dev/sda
brw-rw---- 1 root disk     8,  1 2008-05-21 20:53 /dev/sda1
brw-rw---- 1 root disk     8,  2 2008-05-21 20:53 /dev/sda2
brw-rw---- 1 root disk     8, 16 2008-05-21 20:53 /dev/sdb
brw-rw---- 1 root disk     8, 17 2008-05-21 20:53 /dev/sdb1
```

Χαμηλώστε την ένταση των ηχείων στο ελάχιστο και δοκιμάστε να δώσετε μια εντολή όπως αυτή:

```
$ cat /vmlinuz >/dev/dsp
^C
```

για να ακούσετε το αποτέλεσμα του να γράφετε στο ειδικό αρχείο της κάρτας ήχου τυχαία δείγματα από ένα αρχείο, στη συγκεκριμένη περίπτωση τον κώδικα του πυρήνα που τρέχετε (αρχείο `/vmlinuz`). Με Ctrl-C μπορείτε να διακόψετε την εκτέλεσή της.

Η πρώτη στήλη του καταλόγου της `ls` περιέχει το γράμμα ‘`c`’ αν πρόκειται για κόμβους που αντιστοιχούν σε συσκευές χαρακτήρων ή το γράμμα ‘`b`’ αν πρόκειται για κόμβους που αντιστοιχούν σε συσκευές μπλοκ. Για παράδειγμα, το αρχείο `ttyS0` αντιστοιχεί στην πρώτη σειριακή θύρα του συστήματος, ενώ το αρχείο `/dev/sdb` στο δεύτερο δίσκο SCSI. Σε αντίθεση με ό,τι συμβαίνει για συνηθισμένα αρχεία, για αυτά τα αρχεία (τους *κόμβους συσκευών*, device nodes) η `ls` δεν εμφανίζει μέγεθος, καθώς

[2] Με **έντονα γράμματα** εμφανίζονται οι χαρακτήρες που πληκτρολογεί ο χρήστης. Προτείνεται να πειραματιστείτε με τις εντολές που περιγράφονται και στο δικό σας σύστημα.

δεν αντιστοιχούν σε αποθηκευμένα δεδομένα. Αντίθετα, εμφανίζει δύο αριθμούς, χωρισμένους με κόμμα.

Οι αριθμοί αυτοί και όχι το όνομα του αρχείου, καθορίζουν ποια συσκευή αφορά κάθε κόμβος και ονομάζονται major και minor number αντίστοιχα. Ο major number χρησιμοποιείται από τον πυρήνα για να προσδιορίσει τον οδηγό συσκευής που αφορά ο συγκεκριμένος κόμβος. Για παράδειγμα, όλοι οι δίσκοι SCSI αντιστοιχούν στον ίδιο major number, τον `8`. Ο minor number δεν αφορά τον κώδικα του πυρήνα αλλά τον ίδιο τον οδηγό της συσκευής και χρησιμοποιείται ώστε να μπορεί να διακρίνει ποια συγκεκριμένη συσκευή από όλες όσες μπορεί να χειριστεί αφορά το συγκεκριμένο αρχείο. Έτσι, παρόλο που και το `/dev/sda` και το `/dev/sdb` αντιστοιχούν στον ίδιο οδηγό, τον οδηγό σκληρών δίσκων SCSI, το sda έχει minor `0` και αναφέρεται στον πρώτο δίσκο, ενώ το sdb έχει minor `16` και αναφέρεται στον δεύτερο.

Όπως και οι υπόλοιποι οδηγοί, έτσι και το Lunix:TNG θα χρησιμοποιήσει ένα συγκεκριμένο major number. Ωστόσο, η επιλογή δεν μπορεί να γίνει αυθαίρετα, εφόσον πολλοί χρησιμοποιούνται ήδη. Στο αρχείο `Documentation/devices.txt` στον κώδικα του πυρήνα, κάτω από τον κατάλογο `/usr/src/linux`, υπάρχει μια λίστα με την τρέχουσα αντιστοιχία. Ορισμένοι αριθμοί είναι δεσμευμένοι για πειραματική χρήση, όπως ο `60`, τον οποίο προτείνεται να χρησιμοποιήσετε. Οι αριθμοί που χρησιμοποιούνται κάθε στιγμή από τον πυρήνα φαίνονται με χρήση της εντολής `cat /proc/devices`.

Τα ειδικά αρχεία κατασκευάζονται με χρήση της εντολής `mknod`, οπότε καθορίζεται το είδος τους (character ή block node) και οι major/minor numbers που τους αντιστοιχούν. Το Lunix:TNG θα χρησιμοποιεί περισσότερα του ενός ειδικά αρχεία, ένα για κάθε αισθητήρα του δικτύου και είδος μέτρησης που υποστηρίζεται. Σε κάθε ζεύγος {αριθμός αισθητήρα, μέτρηση} χρειάζεται να ανατεθεί συγκεκριμένο όνομα αρχείου και αντίστοιχος minor number. Προτείνεται τα 3 λιγότερο σημαντικά bits κάθε minor number να καθορίζουν το είδος της μέτρησης [οι μετρήσεις αριθμούνται με τη σειρά «τάση μπαταρίας = 0, θερμοκρασία, φωτεινότητα» και υπάρχει χώρος για το πολύ 8 μετρήσεις ανά αισθητήρα], ενώ τα υπόλοιπα τον αριθμό του αισθητήρα. Έτσι η τιμή του minor number προκύπτει ως `minor = αισθητήρας * 8 + μέτρηση`.

Το αρχείο `/dev/lunix0-batt` που αντιστοιχεί στην τάση της μπαταρίας του πρώτου αισθητήρα φτιάχνεται με την εντολή:

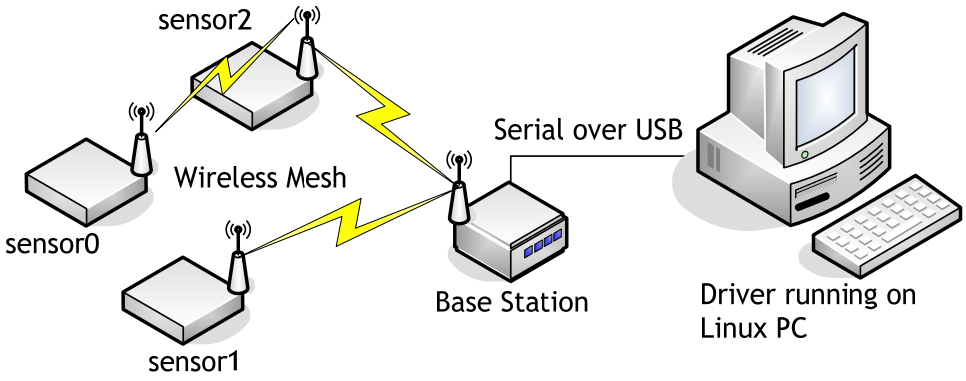
```
# mknod /dev/lunix0-batt c 60 0
```

Ενώ το αρχείο `/dev/lunix1-temp` που αντιστοιχεί στη θερμοκρασία του δεύτερου θα είχε minor number `9`. Σας δίνεται το script `lunix_dev_nodes.sh`, το οποίο κατασκευάζει αυτόματα τους ανάλογους κόμβους με βάση αυτή τη σύμβαση, για 16 αισθητήρες.

4. Περισσότερα για την εργαστηριακή άσκηση

4.1. Αρχιτεκτονική συστήματος ασύρματου δικτύου αισθητήρων

Αντικείμενο της εργαστηριακής άσκησης είναι η υλοποίηση ενός οδηγού συσκευής για ένα ασύρματο δίκτυο αισθητήρων κάτω από το λειτουργικό σύστημα Linux. Το ασύρματο δίκτυο με το οποίο θα εργαστείτε διαθέτει έναν αριθμό από αισθητήρες (ασύρματες κάρτες Crossbow MPR2400CA με αισθητήρες τάσης, θερμοκρασίας και φωτεινότητας MDA100CB) και ένα σταθμό βάσης (κάρτα MPR2400CA και διασύνδεση USB MIB520CB). Ο σταθμός βάσης συνδέεται μέσω USB με υπολογιστικό σύστημα Linux στο οποίο και θα εκτελείται ο ζητούμενος οδηγός συσκευής.



Σχήμα 2: Αρχιτεκτονική του υπό εξέταση συστήματος

Οι αισθητήρες αναφέρουν περιοδικά το αποτέλεσμα τριών διαφορετικών μετρήσεων: της τάσης της μπαταρίας που τους τροφοδοτεί, της θερμοκρασίας και της φωτεινότητας του χώρου όπου βρίσκονται. Οι αισθητήρες οργανώνονται σε ένα πλέγμα (mesh), έτσι ώστε τα δεδομένα τους να μεταφέρονται από εναλλακτικές διαδρομές και το δίκτυο να προσαρμόζεται αυτόματα στην περίπτωση όπου ένας οι περισσότεροι αισθητήρες είναι εκτός της εμβέλειας του σταθμού βάσης (Σχήμα 2). Αυτό δεν χρειάζεται να σας απασχολήσει, οι αλγόριθμοι που χρειάζονται έχουν ήδη υλοποιηθεί σε κατάλληλο ενσωματωμένο λογισμικό που εκτελείται από τους ίδιους τους αισθητήρες και τους επιτρέπει να ανακαλύπτουν ο ένας τον άλλο.

Ο σταθμός βάσης λαμβάνει πακέτα με δεδομένα μετρήσεων, τα οποία προωθεί μέσω διασύνδεσης USB στο υπολογιστικό σύστημα. Η διασύνδεση υλοποιείται με κύκλωμα Serial over USB, με το τσιπ FTDI FT2232C για το οποίο ο πυρήνας διαθέτει ενσωματωμένους οδηγούς. Στην περίπτωση αυτή (η "pre-existing approach" στο Σχήμα 3), δε γίνεται καμία διάκριση ανάμεσα στα διαφορετικά πακέτα δεδομένων: τα δεδομένα όλων των μετρήσεων όλων των αισθητήρων εμφανίζονται σε μία εικονική σειριακή θύρα, `/dev/ttyUSB1`. Επιπλέον, δεν υποστηρίζεται ταυτόχρονη πρόσβαση από πολλές διαφορετικές διεργασίες· μόνο μία έχει νόημα να ανοίγει κάθε φορά την εικονική σειριακή θύρα. Χρειάζεται λοιπόν η ανάπτυξη ενός οδηγού συσκευής ο οποίος να προσφέρει κατάλληλο *μηχανισμό*, αντιμετωπίζοντας ανεξάρτητα τους αισθητήρες και τα μετρούμενα μεγέθη, επιτρέποντας ταυτόχρονη πρόσβαση στα εισερχόμενα δεδομένα και κάνοντας δυνατή την επιβολή διαφορετικής *πολιτικής* από το διαχειριστή του συστήματος όσον αφορά την πρόσβαση σε αυτά.

4.2. Λειτουργικές απαιτήσεις οδηγού συσκευής

Ο ζητούμενος οδηγός συσκευής χρειάζεται να προσφέρει τη δυνατότητα χωριστής πρόσβασης ανά αισθητήρα και μετρούμενο μέγεθος μέσω διαφορετικών ειδικών αρχείων. Το πλήθος των απαιτούμενων αρχείων θα είναι ίσο με τον αριθμό των υποστηριζόμενων αισθητήρων επί τον αριθμό των υποστηριζόμενων μετρήσεων, που είναι τρεις: τάση μπαταρίας, θερμοκρασία, φωτεινότητα. Κάθε ζεύγος {αριθμός μετρητή, μέτρηση} θα αντιστοιχιστεί σε διαφορετικό ειδικό αρχείο, με διαφορετικό minor number, όπως περιγράφεται στην §3.3.

Η πρώτη προσπάθεια ανάγνωσης από ειδικό αρχείο θα επιστρέφει έναν αριθμό από bytes που θα αναπαριστούν την πιο πρόσφατη τιμή του μετρούμενου μεγέθους, σε αναγνώσιμη μορφή, μορφοποιημένη ως δεκαδικό αριθμό. Αν δεν έχουν ακόμη παραληφθεί δεδομένα για τη συγκεκριμένη μέτρηση, ή αν γίνουν επόμενες προσπάθειες ανάγνωσης, ο οδηγός θα πρέπει να κοιμίζει τη διεργασία (βλ. 7.3) έως ότου παραληφθούν νέα δεδομένα. Έτσι, αυτή δεν θα καταναλώνει χρόνο στην CPU του συστήματος.

Ορίστε ένα παράδειγμα χρήσης του συστήματος, με σχολιασμό:

Ο απλός χρήστης "user", αφού κάνει login μεταγλωττίζει τον κώδικα του module δίνοντας την εντολή `make`:

[αν ο πηγαίος κώδικας του πυρήνα για τον οποίο γίνεται η μεταγλώττιση είναι σε ασυνήθιστη θέση, χρειάζεται να τεθεί η μεταβλητή περιβάλλοντος KERNELDIR]

```
user@machine:~/lunix-tng$ export KERNELDIR=~/utopia/linux-2.6.37.4
user@machine:~/lunix-tng$ make
gcc -Wall -Werror -m32 -o mk_lookup_tables mk_lookup_tables.c -lm
./mk_lookup_tables >lunix-lookup.h
make -C /home/user/utopia/linux-2.6.37.4 M=/home/user/lunix-tng modules
make[1]: Entering directory `/home/user/utopia/linux-2.6.37.4'
  CC [M]  /home/user/lunix-tng/lunix-module.o
  CC [M]  /home/user/lunix-tng/lunix-chrdev.o
  CC [M]  /home/user/lunix-tng/lunix-ldisc.o
  CC [M]  /home/user/lunix-tng/lunix-protocol.o
  CC [M]  /home/user/lunix-tng/lunix-sensors.o
  LD [M]  /home/user/lunix-tng/lunix.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/user/lunix-tng/lunix.mod.o
  LD [M]  /home/user/lunix-tng/lunix.ko
make[1]: Leaving directory `/home/user/utopia/linux-2.6.37.4'
gcc -Wall -Werror -o lunix-attach lunix-attach.c
```

Στη συνέχεια, αποκτά δικαιώματα διαχειριστή (`su`), για να φορτώσει το module (`insmod`), να δημιουργήσει μια σειρά από ειδικά αρχεία της συσκευής (`lunix_dev_nodes.sh`) και να ενεργοποιήσει την είσοδο δεδομένων από τη σειριακή θύρα USB /dev/ttyUSB1 (`lunix-attach`):

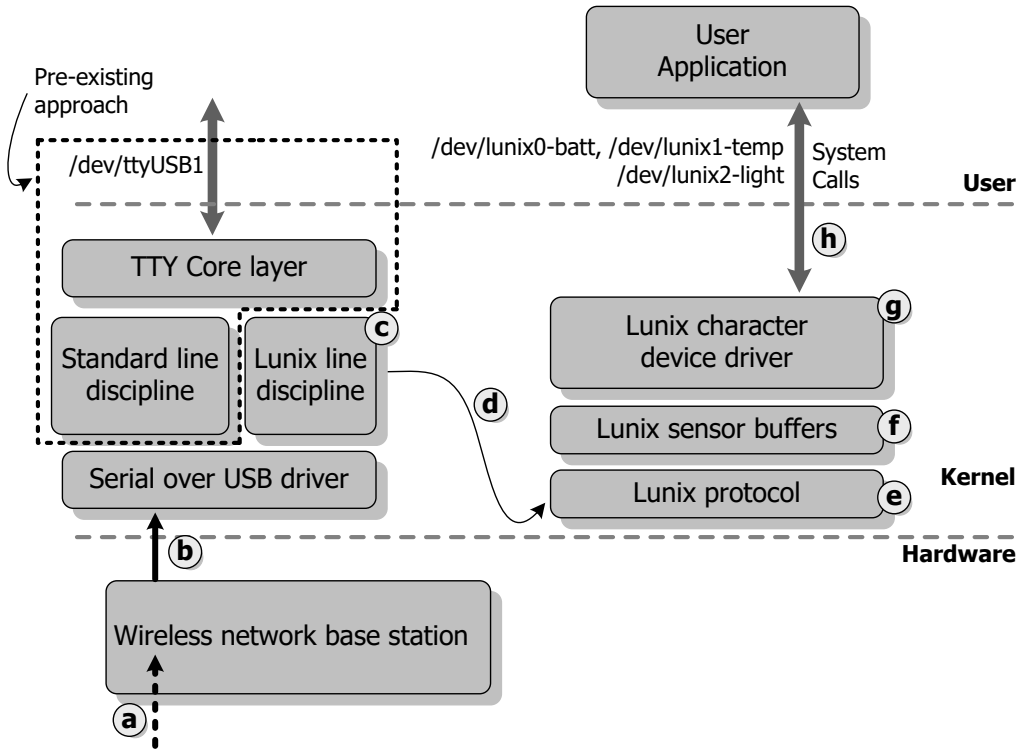
```
user@machine:~/lunix-tng$ su –
Password:
machine:~# cd ~user/lunix-tng
machine:/home/user/lunix-tng# insmod ./lunix.ko
machine:/home/user/lunix-tng# ./lunix_dev_nodes.sh
machine:/home/user/lunix-tng# ./lunix-attach /dev/ttyUSB1
tty_open: looking for lock
tty_open: trying to open /dev/ttyUSB1
tty_open: /dev/ttyUSB1 (fd=3) Line discipline set on /dev/ttyUSB1,
press ^C to release the TTY...
```

Τέλος, ζητά τη θερμοκρασία του 3ου αισθητήρα.

```
user@utopia:~$ cat /dev/lunix2-temp
```
[αν ο αισθητήρας είναι κλειστός, εδώ η cat φαίνεται να έχει κολλήσει]
```
  27.791
```
[μικρή παύση ανάμεσα σε κάθε γραμμή, ανάλογα με τη συχνότητα ανανέωσης των δεδομένων]
```
  27.791
  27.693
  27.791
^C
```

4.3. Προτεινόμενη αρχιτεκτονική οδηγού συσκευής

Η προτεινόμενη αρχιτεκτονική του υπό σχεδίαση συστήματος παρουσιάζεται στο ακόλουθο σχήμα. Το παρακάτω αποτελεί μια πλήρη περιγραφή της αρχιτεκτονικής του και συμπεριλαμβάνει τόσο τα τμήματα που σας δίνονται ήδη υλοποιημένα όσο και το τμήμα που θα πρέπει εσείς να κατασκευάσετε (τα τμήματα **(g)** και **(h)** του παρακάτω σχήματος). Η περιγραφή γίνεται με σκοπό την ευκολότερη ανάγνωση του κώδικα που σας δίνεται και την καλύτερη κατανόηση του πλαισίου στο οποίο θα ενταχθεί η δουλειά σας.



Σχήμα 3: Αρχιτεκτονική λογισμικού του υπό εξέταση συστήματος

Το σύστημα οργανώνεται σε δύο κύρια μέρη: Το πρώτο (τμήματα **(c)**, **(d)**, **(e)** του σχήματος) αναλαμβάνει τη *συλλογή* των δεδομένων από το σταθμό βάσης και την επεξεργασία τους με συγκεκριμένο πρωτόκολλο, έτσι ώστε να εξαχθούν οι τιμές των μετρούμενων μεγεθών, οι οποίες και κρατούνται σε κατάλληλες δομές προσωρινής αποθήκευσης (buffers) στη μνήμη (τμήμα **(f)**). Το δεύτερο (τμήματα **(g)**, **(h)**), αναλαμβάνει την *εξαγωγή* των δεδομένων στο χώρο χρήστη, υλοποιώντας μια σειρά από συσκευές χαρακτήρων. Τα δεδομένα προέρχονται από ανάγνωση των χώρων ενδιάμεσης αποθήκευσης.

Το τελικό σύστημα θα λειτουργεί ως εξής: Τα δεδομένα των μετρήσεων αφού **(a)** λαμβάνονται από το σταθμό βάσης **(b)** θα προωθούνται μέσω USB στο υπολογιστικό σύστημα. Στη συνέχεια όμως δεν θα ακολουθούν τη συνήθη πορεία τους, προς το `/dev/ttyUSB1` αλλά **(c)** θα παραλαμβάνονται από ένα φίλτρο, τη διάταξη γραμμής (line discipline) του Lunix, η οποία **(d)** θα τα προωθεί σε κατάλληλο στρώμα ερμηνείας του πρωτοκόλλου των αισθητήρων **(e)**. Το τμήμα αυτό ερμηνεύει το περιεχόμενο των πακέτων και αποθηκεύει τα δεδομένα των μετρήσεων σε διαφορετικούς buffers ανά αισθητήρα **(f)**. Η ζητούμενη συσκευή χαρακτήρων **(g)**, αναλόγως με το ποιο ειδικό αρχείο χρησιμοποιεί η εφαρμογή χρήστη, θα επιτρέπει την ανάκτηση **(h)** από τους buffers των δεδομένων που αντιστοιχούν σε κάθε περίπτωση.

Η υλοποίηση των τμημάτων **(b)**, **(c)**, **(d)**, **(e)**, **(f)** σας δίνεται έτοιμη, ως τμήματα κώδικα μαζί με τις κατάλληλες δομές δεδομένων που ορίζουν τις διεπαφές (interfaces) ανάμεσά τους. Από εσάς ζητείται η υλοποίηση του **(g)** και ο έλεγχος **(h)** της σωστής λειτουργίας του συστήματος, μέσω ενός προγράμματος χρήστη.

Στη συνέχεια αναλύεται ο ρόλος καθενός από τα παραπάνω τμήματα του λογισμικού και ο τρόπος αλληλεπίδρασής του με τα υπόλοιπα.

5. Τμήμα εξαγωγής δεδομένων στο χώρο χρήστη

5.1. Interface συσκευών χαρακτήρων

Όπως είπαμε και προηγούμενα, το Linux παρουσιάζει σχεδόν όλες τις συσκευές ως αρχεία. Τα ειδικά αρχεία συσκευών, μέσω του οποίου είναι δυνατή η αλληλεπίδραση με συσκευές Ε/Ε αποτελούν και αυτά μέρος του συστήματος αρχείων (filesystem), όπως και τα αρχεία δεδομένων. Ο οδηγός συσκευής μοιάζει με ένα κανονικό αρχείο το οποίο μπορείτε να ανοίξετε (`open()`), κλείσετε (`close()`), διαβάσετε (`read()`) και να γράψετε (`write()`). Ο πυρήνας βλέπει αυτές τις λειτουργίες ως ειδικές αιτήσεις και τις μεταφράζει σε κλήσεις προς τον αντίστοιχο οδηγό συσκευής. Στην περίπτωση των συσκευών χαρακτήρων (όχι όμως και των συσκευών μπλοκ, όπου το interface είναι εντελώς διαφορετικό), υπάρχει σχεδόν 1-1 αντιστοιχία ανάμεσα σε κλήσεις συστήματος και κλήσεις προς τον οδηγό συσκευής.

Όταν για παράδειγμα ανοίξουμε (`open()`) ένα ειδικό αρχείο, ο πυρήνας ψάχνει την αντίστοιχη συνάρτηση στον κώδικα του οδηγού και την εκτελεί. Όμως πώς καταφέρνει και βρίσκει την κατάλληλη συνάρτηση για κάθε συσκευή; Το καταφέρνει,

διότι κάθε οδηγός για συσκευή χαρακτήρων υλοποιεί μια συγκεκριμένη διεπαφή (interface) προς τον πυρήνα. Το interface αυτό περιγράφεται στη συνέχεια.

Όλοι οι οδηγοί παρέχουν ένα σύνολο από ρουτίνες. Κάθε οδηγός παρέχει μια δομή `struct file_operations` που περιέχει δείκτες στις ρουτίνες αυτές. Κατά την αρχικοποίηση του οδηγού, αυτή η δομή "αγκιστρώνεται" σε ένα πίνακα που γνωρίζει ο πυρήνας. Κάθε φορά που γίνεται αίτηση για άνοιγμα ενός ειδικού αρχείου, ο αριθμός major του αρχείου χρησιμοποιείται από τον πυρήνα ως δείκτης σε αυτόν τον πίνακα, ώστε να εντοπίσει τον κατάλληλο οδηγό συσκευής και την ανάλογη δομή `file_operations`.

Εκτός από τη δομή `file_operations`, πολύ σημαντικό ρόλο στην αλληλεπίδραση του οδηγού συσκευής με τον πυρήνα παίζει και η δομή `struct file`. Η δομή αυτή αναπαριστά κάποιο ανοιχτό αρχείο μιας διεργασίας μέσα στον κώδικα του πυρήνα. Ο πυρήνας δημιουργεί μια νέα δομή `file` κάθε φορά που μια διεργασία εκτελεί την κλήση συστήματος `open()` για να ανοίξει κάποιο αρχείο.

Μια δομή `file` δημιουργείται και όταν μια διεργασία ανοίξει ειδικό αρχείο που αναφέρεται στο Lunix:TNG. Η δομή αυτή αντιστοιχίζεται, μέσω ενός δείκτη, με τη δομή `file_operations` που έχει δηλώσει ο οδηγός συσκευής στον πυρήνα. Έτσι, κάθε φορά που η διεργασία ζητά την εκτέλεση μιας κλήσης συστήματος επάνω στο ανοιχτό αρχείο (π.χ. της `read()` για την ανάγνωση δεδομένων), ο πυρήνας βρίσκει τη δομή `file` που αντιστοιχεί στο ανοιχτό αρχείο, ακολουθεί το σύνδεσμο προς τη δομή `file_operations` του οδηγού συσκευής, ακολουθεί το δείκτη της δομής που αναφέρεται στη λειτουργία `read` και φτάνει έτσι στην ρουτίνα του οδηγού συσκευής, την οποία και εκτελεί. Η παραπάνω διαδικασία θα γίνει περισσότερο κατανοητή όταν παρουσιαστούν τα πεδία των δομών `file_operations` και `file`.

5.2. Λειτουργίες αρχείου - Η δομή file_operations

Στη συνέχεια παρουσιάζονται τα πεδία της δομής `file_operations` (βλ. και [1], σελ. 49) Κάθε πεδίο της δομής αυτής αντιστοιχεί σε μία λειτουργία που αναλαμβάνει να επιτελέσει ο οδηγός συσκευής που υλοποιούμε, όταν του ζητηθεί από άλλα τμήματα του πυρήνα. Η συσκευή αναγνωρίζεται εσωτερικά στον πυρήνα μέσω της δομής `file` που προκύπτει από άνοιγμα ενός ειδικού αρχείου, π.χ. του `/dev/sensor1-light` από κάποια διεργασία. Το ειδικό αρχείο έχει κατασκευαστεί εκ των προτέρων, όπως αναλύεται στην §3.3.

Τα πεδία της δομής `file_operations` ονομάζονται συχνά και «μέθοδοι» της συσκευής, σύμφωνα με την ορολογία του αντικειμενοστρεφούς προγραμματισμού, θεωρώντας ότι ο πυρήνας ζητά από τον οδηγό συσκευής την εκτέλεση μιας λειτουργίας καλώντας την αντίστοιχη μέθοδο του αντικειμένου «ανοιχτό αρχείο».

Η δομή `file_operations` αποτελείται από δείκτες σε συναρτήσεις και ορίζεται στο `<linux/fs.h>`. Στη συνέχεια παρουσιάζονται περιληπτικά τα σημαντικότερα από τα πεδία της. Αν ο οδηγός συσκευής δεν υποστηρίζει τη συγκεκριμένη λειτουργία και δεν παρέχει ρουτίνα που την υλοποιεί, ο δείκτης της δομής `file_operations` είναι `NULL` και η συμπεριφορά του πυρήνα εξαρτάται από το είδος της λειτουργίας που δεν παρέχεται.

Η εκτέλεση μιας συνάρτησης θεωρείται επιτυχής όταν η επιστρεφόμενη τιμή είναι μηδέν, ενώ αρνητική επιστρεφόμενη τιμή υποδηλώνει κάποιο σφάλμα. Μπορεί να επιστραφεί και θετικός αριθμός μερικές φορές, οπότε θα επισημαίνεται στην περιγραφή της λειτουργίας.

- `loff_t llseek (struct file *, loff_t, int);`

Η μέθοδος `llseek()` χρησιμοποιείται για να αλλάξουμε την θέση ανάγνωσης / εγγραφής σε ένα αρχείο και η νέα τιμή επιστρέφεται ως μια θετική τιμή. Τα λάθη σηματοδοτούνται από αρνητική τιμή. Αν η συνάρτηση δεν ορίζεται στον οδηγό (υπάρχει `NULL` στην ανάλογη θέση στη δομή `struct file_operations`), ο πυρήνας προσφέρει μια προκαθορισμένη (default) υλοποίηση της `llseek()`, σύμφωνα με την οποία οι λειτουργίες seek ως προς το τέλος του αρχείου αποτυγχάνουν, διαφορετικά επιτυγχάνουν αλλάζοντας τον μετρητή θέσης στη δομή `file`.

- `ssize_t read(struct file *, char __user *, size_t, loff_t *);`

Χρησιμοποιείται για να διαβάζει δεδομένα από τη συσκευή. Αν ο οδηγός συσκευής δεν υποστηρίζει αυτή τη λειτουργία, οπότε ο δείκτης στη θέση αυτή είναι `NULL`, η κλήση συστήματος `read()` αποτυγχάνει με την τιμή `-EINVAL` (“Invalid argument”). Μια μη αρνητική επιστρεφόμενη τιμή φανερώνει το πλήθος των bytes που διαβάστηκαν επιτυχώς, ενώ η τιμή `0` υποδηλώνει ότι η ανάγνωση έφτασε στο τέλος του αρχείου (EOF).

Η μέθοδος αυτή θα είναι η καρδιά της υλοποίησής σας. Χρειάζεται να λαμβάνει δεδομένα από τους sensor buffers, να φροντίζει για τη σωστή μορφοποίησή τους και

να τα περνά στο χώρο χρήστη. Για τον τρόπο με τον οποίο υλοποιείται και τις συμβάσεις που ακολουθούνται, δείτε τα αντίστοιχα σχόλια στον κώδικα που θα σας δοθεί και την υλοποίηση της αντίστοιχης `read()` του scull στο [1].

- `ssize_t write(struct file *, const char __user *, size_t, loff_t *);`

Στέλνει δεδομένα στην συσκευή. Αν δεν παρέχεται από τον οδηγό συσκευής, επιστρέφεται `-EINVAL` στο πρόγραμμα που κάλεσε την κλήση συστήματος `write()`. Η επιστρεφόμενη τιμή, αν δεν είναι αρνητική, φανερώνει το πλήθος των bytes που γράφτηκαν επιτυχώς.

- `int unlocked_ioctl(struct file *, unsigned int, unsigned long);`

Η κλήση συστήματος `ioctl()` προσφέρει ένα τρόπο για να δώσουμε ειδικές εντολές για τη συσκευή υλικού, που δεν είναι μπορούν να παρασταθούν ούτε ως ανάγνωση ούτε ως εγγραφή δεδομένων . Για παράδειγμα, μια τέτοια εντολή θα ήταν η μορφοποίηση της δισκέτας για τη συσκευή `/dev/fd0`, ή η μεταβολή του baud rate για μια σειριακή θύρα (`/dev/ttyS0`). Μερικές εντολές ioctl ικανοποιούνται άμεσα από τον πυρήνα, χωρίς να κληθεί ο οδηγός συσκευής. Αν ο οδηγός δεν προσφέρει την `ioctl()`, τότε η κλήση συστήματος επιστρέφει `-EINVAL`. Μια μη αρνητική επιστρεφόμενη τιμή περνά αμετάβλητη στον καλούντα για να φανερώσει την επιτυχία της κλήσης.

- `int open (struct inode *, struct file *)`

Αν και αυτή η συνάρτηση είναι η πρώτη που καλείται στον κόμβο συσκευής, ο οδηγός συσκευής δεν χρειάζεται να ορίσει μια τέτοια μέθοδο. Στην περίπτωση αυτή το άνοιγμα συσκευής θεωρείται πάντα επιτυχές, αλλά ο οδηγός μας δεν ενημερώνεται ποτέ. Εμείς θα ορίζουμε τη συνάρτηση αυτή, επειδή θέλουμε να ενημερώνεται ο οδηγός μας για το άνοιγμα της συσκευής, ώστε να μας δίνεται η ευκαιρία να κάνουμε αρχικοποιήσεις σε δομές δεδομένων του οδηγού. Το όρισμα τύπου `struct inode` αφορά στο ειδικό αρχείο που χρησιμοποιήθηκε κατά την κλήση συστήματος. Με χρήση των μακροεντολών

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

μπορούμε να ανακτήσουμε τον major και minor number του αρχείου, ώστε το ανοιχτό αρχείο που θα προκύψει να αφορά συγκεκριμένο αισθητήρα και μέτρηση του δικτύου.

Στην περίπτωσή που εξετάζετε, η open οφείλει να δεσμεύει χώρο για τη δομή τύπου lunix_chrdev_state_struct που περιγράφει την τρέχουσα κατάσταση της συσκευής και να τη συνδέει με τη δομή file μέσω του δείκτη private_data (βλ. παρακάτω).

- `void (*release) (struct inode *, struct file *);`

Αυτή η μέθοδος καλείται όταν μια διεργασία χρήστη εκτελέσει την κλήση συστήματος close() για το κλείσιμο του ανοιχτού αρχείου της συσκευής, οπότε καταστρέφεται η αντίστοιχη δομή file. Όμοια με την open(), δεν χρειάζεται να οριστεί μέθοδος release() και θα είναι πάντα επιτυχές το κλείσιμο. Στην δική μας περίπτωση, θα ορίζουμε συνάρτηση για το δείκτη release, για να ενημερωνόμαστε όταν το πρόγραμμα που είχε ανοίξει τη συσκευή μας κλείσει το ανοιχτό αρχείο και να απελευθερώνουμε τους αντίστοιχους πόρους (δεσμευμένη μνήμη).

Η μορφή της δομής file_operations για το Lunix:TNG θα είναι κάπως έτσι:

```
static struct file_operations lunix_chrdev_fops =
{
        .owner   = THIS_MODULE,
        .open    = lunix_chrdev_open,
        .release = lunix_chrdev_release,
        .read    = lunix_chrdev_read
};
```

5.3. Ανοιχτά αρχεία - Η δομή file

Η δομή struct file, που ορίζεται στο <linux/fs.h>, (βλ. [1] σελ. 53) είναι η δεύτερη σημαντικότερη δομή που χρησιμοποιείται στους οδηγούς συσκευών. Προσέξτε ότι ένα file δεν έχει σχέση με τα FILEs των προγραμμάτων userspace. Η δομή FILE ορίζεται στην βιβλιοθήκη της C και δεν εμφανίζεται ποτέ σε κώδικα στον πυρήνα. Ένα struct file, από την άλλη, είναι μια δομή του πυρήνα και ποτέ δεν εμφανίζεται στα προγράμματα των χρηστών.

Η δομή file αναπαριστά ένα «ανοιχτό αρχείο». Δημιουργείται από τον πυρήνα κατά την εκτέλεση της open και περνιέται σε κάθε συνάρτηση που λειτουργεί στο file, μέχρι το κλείσιμό του (κλήση συστήματος close()). Μετά το κλείσιμο του αρχείου, ο πυρήνας αποδεσμεύει την αντίστοιχη δομή file. Ένα "open file" είναι διαφορετικό από ένα "disk file", που αναπαρίσταται από ένα struct inode.

Τα πεδία της δομής file αναπαριστούν την κατάσταση ενός αρχείου ανοιχτού για κάποια διεργασία χρήστη, δηλαδή τα δικαιώματα πρόσβασης της διεργασίας στο αρχείο, τη θέση του δείκτη ανάγνωσης / εγγραφής κλπ. Τα σημαντικότερα από αυτά είναι:

- `mode_t f_mode;`

Τα bits του πεδίου f_mode καθορίζουν το είδος της πρόσβασης που επιτρέπεται να έχει η διεργασία στο ανοιχτό αρχείο. Για παράδειγμα, αν το αρχείο έχει ανοιχτεί για ανάγνωση, το bit FMODE_READ θα είναι 1 και η παράσταση f_mode & FMODE_READ θα είναι μη μηδενική.

- `loff_t f_pos;`

Στο πεδίο αυτό αποθηκεύεται η τρέχουσα θέση του δείκτη ανάγνωσης / εγγραφής. Ο τύπος loff_t είναι μια τιμή εύρους 64bit (long long στην ορολογία του gcc). Ο οδηγός μπορεί να διαβάσει την τιμή αυτή αν χρειάζεται να μάθει την τρέχουσα θέση. Η μέθοδος llseek, εφόσον ορίζεται, αναλαμβάνει να ανανεώνει την τιμή f_pos. Οι μέθοδοι read και write πρέπει την ανανεώνουν όταν μεταφέρουν δεδομένα.

- `unsigned short f_flags;`

Τα flags αυτά καθορίζουν κάποιες άλλες ιδιότητες πρόσβασης, όπως O_RDONLY, O_NONBLOCK, και O_SYNC. Ένας οδηγός ελέγχει συνήθως για το flag για τις λειτουργίες nonblocking, ενώ τα άλλα σπάνια χρησιμοποιούνται. Όλα τα flags ορίζονται στο αρχείο επικεφαλίδας <linux/fcntl.h>.

- `struct file_operations *f_op;`

Αυτό είναι το πεδίο της δομής file που επιτρέπει την αντιστοίχιση του ανοιχτού αρχείου με συγκεκριμένη δομή file_operations, άρα και με συγκεκριμένο οδηγό συσκευής. Ο πυρήνας αρχικοποιεί τον δείκτη αυτό κατά το άνοιγμα και τον ακολουθεί όποτε χρειάζεται να καλέσει κάποια συνάρτηση για να εκτελέσει κάποια

λειτουργία επάνω στο ανοιχτό αρχείο. Δεν προβλέπεται να χρησιμοποιηθεί άμεσα από τον οδηγό συσκευής μας.

- `void *private_data;`

Η κλήση συστήματος `open()` θέτει το δείκτη αυτό σε `NULL` πριν την κλήση της μεθόδου `open` του οδηγού. Ο οδηγός είναι ελεύθερος να κάνει χρήση του δείκτη αυτού όπως θεωρεί καλύτερο, ακόμα και να τον αγνοήσει εντελώς. Χρησιμοποιείται συνήθως ώστε να αντιστοιχιστεί το ανοιχτό αρχείο σε ιδιωτικές δομές δεδομένων του οδηγού συσκευής. Ο δείκτης μπορεί να χρησιμοποιηθεί ώστε να δείχνει σε δεσμευμένα δεδομένα, αλλά αυτά πρέπει να απελευθερωθούν από τη μέθοδο `release` πριν ο πυρήνας καταστρέψει τη δομή `file`. Ο δείκτης `private_data` είναι ένας πολύ χρήσιμος τρόπος για να κρατάμε πληροφορίες για την κατάσταση της συσκευής (state information) ανάμεσα στις διάφορες κλήσεις συστήματος.

Στην περίπτωσή που εξετάζετε, θα χρησιμοποιείται ώστε να δείχνει σε δομή τύπου `lunix_chrdev_state_struct`, η οποία περιγράφει την τρέχουσα κατάσταση της συσκευής.

Η πραγματική δομή `struct file` διαθέτει μερικά ακόμη πεδία, αλλά δεν μας είναι χρήσιμα. Θυμίζουμε ότι όποιος θέλει να μάθει περισσότερα, μπορεί να κοιτάξει στο αρχείο `<linux/fs.h>` όπου ορίζεται η δομή.

6. Τμήμα συλλογής και επεξεργασίας δεδομένων

Στο κεφάλαιο αυτό περιγράφεται ο τρόπος με τον οποίο έχει ήδη σχεδιαστεί και υλοποιηθεί το τμήμα συλλογής και επεξεργασίας δεδομένων, το οποίο θα παρέχει τις τιμές των μετρούμενων μεγεθών στον οδηγό συσκευής χαρακτήρων που κατασκευάζετε.

6.1. Συλλογή δεδομένων από σειριακές θύρες

Κατά τη σχεδίαση του Lunix:TNG προκύπτει η ανάγκη *συλλογής* δεδομένων που προέρχονται από το σταθμό βάσης και προώθησής προς το τμήμα επεξεργασίας του πρωτοκόλλου, έτσι ώστε να είναι δυνατή η εξαγωγή και προσωρινή αποθήκευση των μετρούμενων τιμών.

Όταν ένας οδηγός συσκευής γράφεται από μηδενική βάση, η ανάγκη αυτή καλύπτεται με άμεση, απευθείας αλληλεπίδραση με το υλικό. Υπάρχουν δύο τρόποι να γίνει αυτό: Ο πρώτος τρόπος είναι *σύγχρονος:* όταν ο οδηγός της συσκευής χρειαστεί νέα δεδομένα αποστέλλει εντολές Ε/Ε στο υλικό. Αυτό τις επεξεργάζεται ενώ ο επεξεργαστής περιμένει και τελικά το υλικό επιστρέφει τις τιμές που του ζητήθηκαν. Ο δεύτερος τρόπος είναι *ασύγχρονος* και είναι αυτός που υποστηρίζεται από τις περισσότερες συσκευές σήμερα, ακριβώς γιατί δεν απαιτεί άμεση και συνεχή εμπλοκή του επεξεργαστή: όταν υπάρχουν διαθέσιμες νέες τιμές (π.χ. όταν πατηθεί ένα πλήκτρο στο πληκτρολόγιο), ενεργοποιείται κατάλληλο σήμα διακοπής (interrupt) προς τον επεξεργαστή. Ο πυρήνας ξεκινά να επεξεργάζεται τη διακοπή και την προωθεί στον οδηγό συσκευής, ο οποίος έχει ζητήσει να την εξυπηρετεί. Ο οδηγός εξετάζει τη συσκευή και αντιγράφει σε χώρους προσωρινής αποθήκευσης τις νέες πληροφορίες.

Στην περίπτωση του Lunix:TNG όμως, δεν υπάρχει λόγος να ξεκινήσουμε να γράφουμε κώδικα από το μηδέν για το σταθμό βάσης του δικτύου. Ο λόγος είναι ότι αυτός βασίζεται σε ένα γνωστό τσιπ Serial over USB, για το οποίο ήδη υπάρχει βασική υποστήριξη από τον πυρήνα. Αυτό σημαίνει ότι *ήδη* έχει γραφτεί κώδικας που συνεργάζεται με το υποσύστημα USB του πυρήνα, έτσι ώστε να μπορεί να ανακαλύψει τη συσκευή, να λάβει δεδομένα από αυτή και να τα εμφανίσει σε μία *εικονική* σειριακή θύρα (περισσότερα για οδηγούς συσκευών USB μπορείτε να βρείτε στο [1], κεφ. 13). Επιπλέον, δεν αποκλείεται άλλοι, παρόμοιοι σταθμοί βάσης που χρησιμοποιούν το ίδιο πρωτόκολλο να συνδέονται μέσω μιας πραγματικής σειριακής σύνδεσης στο μηχάνημα αντί για USB, οπότε τα δεδομένα τους εμφανίζονται σε μια συσκευή όπως η `/dev/ttyS0`. Ο οδηγός μας δεν έχει λόγο να μην μπορεί να υποστηρίξει και αυτή την περίπτωση, εφόσον το πρωτόκολλο επικοινωνίας (η μορφή των bytes που τελικά μεταφέρονται) είναι ουσιαστικά το ίδιο.

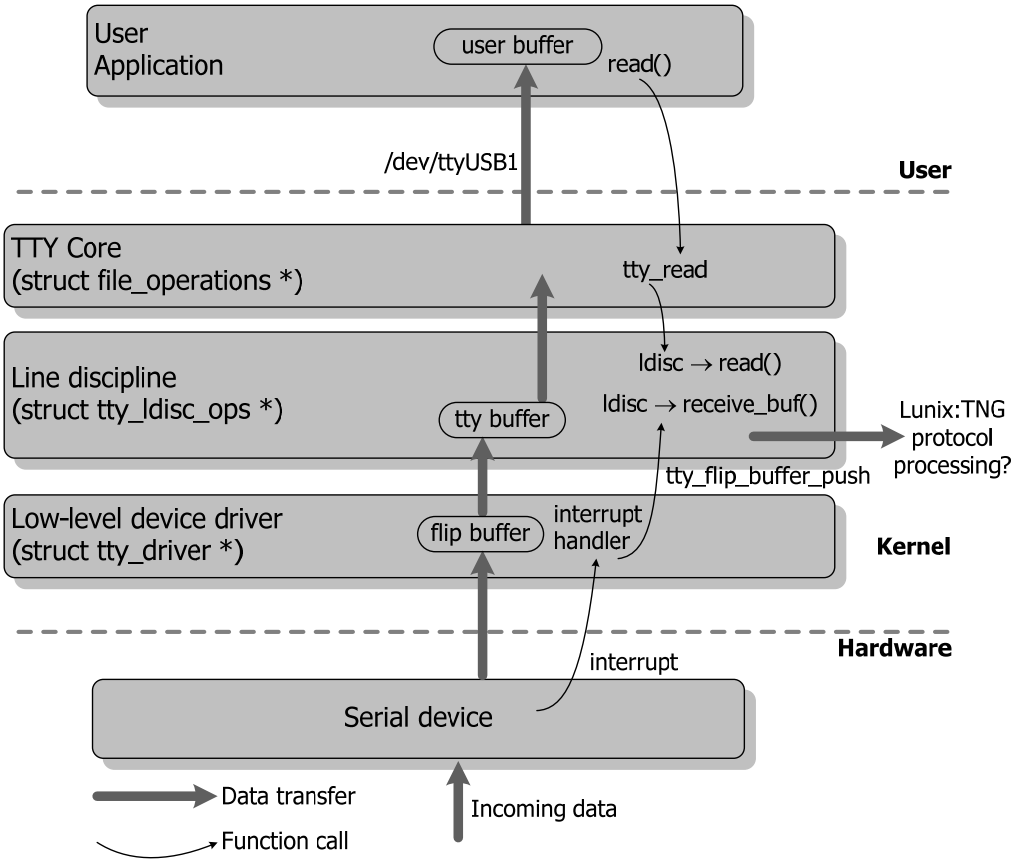
Αυτό λοιπόν που χρειάζεται είναι ένας *γενικός* τρόπος το Lunix:TNG να λαμβάνει δεδομένα από συσκευές εισόδου όπως η `/dev/ttyS0` και η `/dev/ttyUSB1` βασιζόμενο σε ήδη υπάρχοντες οδηγούς για σειριακές και USB θύρες που υπάρχουν μέσα στον πυρήνα. Για να το πετύχουμε αυτό χρειάζεται να δούμε περισσότερο διεξοδικά το ρόλο του *στρώματος TTY*, του στρώματος δηλαδή του πυρήνα που υποστηρίζει τερματικές συσκευές, όπως είναι οι σειριακές θύρες.

6.2. Το στρώμα TTY του πυρήνα του Linux

Ο όρος "TTY" είναι μια σύντμηση του "TeleTYpewriter" και αναφέρεται σε πραγματικά αρχαίες συσκευές, τα τηλέτυπα, [2] οι οποίες χρησιμοποιούνταν ως τερματικά για είσοδο εντολών σε μηχανήματα UNIX και παρουσίαση αποτελεσμάτων. Ως θύρα TTY χαρακτηρίζεται κάθε θύρα που μπορεί να υποστηρίξει τερματικό, στην οποία μπορούμε να κάνουμε login και να δώσουμε εντολές στο σύστημα.

Σε ένα σύγχρονο σύστημα Linux οι θύρες TTY συμπεριλαμβάνουν τις σειριακές θύρες (`ttyS0`, `ttyS1`), τις σειριακές πάνω από USB (`ttyUSB0`, `ttyUSB1`), τις κονσόλες του συστήματος (`tty0`, `tty1`, `tty2`) αλλά και εικονικές κονσόλες που δημιουργούνται όταν για παράδειγμα ανοίγετε ένα νέο παράθυρο τερματικού (π.χ. `xterm`, `konsole`, `gnome-terminal`) κάτω από τα X. Με την εντολή `tty` μπορείτε να δείτε τη θύρα στην οποία έχετε συνδεθεί κάθε φορά.

Όλες οι θύρες TTY υφίστανται ενιαία διαχείριση από τον πυρήνα, καθώς χρειάζεται να υποστηρίζουν ένα σύνολο από κοινά χαρακτηριστικά. Υπάρχει οργάνωση σε στρώματα: στο κατώτερο στρώμα υπάρχει κώδικας (ο *οδηγός TTY*) που μιλάει άμεσα με το υλικό (π.χ. το τσιπ που χειρίζεται τη σειριακή θύρα), ενώ στο ανώτερο στρώμα υπάρχει κώδικας που υλοποιεί τη διεπαφή συσκευής χαρακτήρων, ώστε οι TTYs να εμφανίζονται ως συσκευές χαρακτήρων στο σύστημα (Σχήμα 4).



Σχήμα 4: Αρχιτεκτονική του στρώματος TTY του Linux

Το στρώμα TTY του πυρήνα χωρίζεται σε τρία κύρια μέρη: Το τμήμα "TTY core", το τμήμα της *διάταξης* γραμμής (line discipline) και το τμήμα του κατώτερου οδηγού συσκευής. Το τμήμα του κατώτερου οδηγού συσκευής αναλαμβάνει την απευθείας επικοινωνία με τη σειριακή συσκευή, π.χ. το χειρισμό σημάτων διακοπής, αλλά δε χρειάζεται να γνωρίζει τίποτε για το πώς η συσκευή αποθηκεύει προσωρινά δεδομένα στα ανώτερα επίπεδα ή υποστηρίζει λειτουργίες τερματικού. Το τμήμα TTY core φροντίζει ώστε οι σειριακές συσκευές να εμφανίζονται ως συσκευές χαρακτήρων στο σύστημα.

Το τμήμα που μας ενδιαφέρει περισσότερο είναι αυτό της line discipline: λειτουργεί ως φίλτρο και φροντίζει για την προσωρινή αποθήκευση των δεδομένων. Το σημαντικό είναι ότι μπορεί να αλλάξει. Διαφορετικές διατάξεις μπορούν να επιλεγούν αναλόγως με τη χρήση που πρόκειται να έχουν τα δεδομένα. Η συνηθισμένη line discipline ονομάζεται `N_TTY` και απλά περνά δεδομένα στο αντίστοιχο `/dev/tty`.

Ας δούμε περιληπτικά τι συμβαίνει στη συνηθισμένη περίπτωση όπου μια διεργασία ζητά να διαβάσει δεδομένα από μια σειριακή θύρα, έστω τη `/dev/ttyUSB1`. Η διαδικασία της παραλαβής των δεδομένων λειτουργεί ασύγχρονα σε σχέση με τη διαδικασία ανάγνωσης/κατανάλωσής τους από τη διεργασία, οπότε χρειάζεται τα δεδομένα που λαμβάνονται να αποθηκεύονται προσωρινά έως ότου ζητηθούν.

Όταν παραληφθούν δεδομένα από τη συσκευή, αυτή προκαλεί ένα σήμα διακοπής, το οποίο εξυπηρετείται από τον κατώτερο οδηγό TTY. Τα εισερχόμενα δεδομένα αντιγράφονται μέσα σε έναν από δύο χώρους που ονομάζονται flip buffers και χρησιμοποιούνται εναλλάξ: κάθε φορά ένας από τους δύο είναι διαθέσιμος για χρήση από τη συσκευή, ενώ ο άλλος προωθείται προς τη line discipline. Όταν αυτός που χρησιμοποιείται γεμίσει, οι buffers αλλάζουν ρόλους ("flip").

Όταν ένας flip buffer γεμίσει, ο κατώτερος οδηγός καλεί την συνάρτηση `tty_flip_buffer_push()` (αρχείο `drivers/tty/tty_buffer.c` του κώδικα του πυρήνα) η οποία με τη σειρά της περνά τα δεδομένα στη διάταξη γραμμής αντιγράφοντάς τα στον TTY buffer (`flush_to_ldisc()`). Η διάταξη γραμμής ενημερώνεται ότι νέα δεδομένα έχουν φτάσει όταν καλείται η μέθοδος `ldisc→receive_buf()` για να τα επεξεργαστεί. Τέλος, τα δεδομένα ανακτώνται από την `tty_read()` η οποία χρησιμοποιεί την `ldisc→read()` για να τα αντιγράψει από τον TTY buffer στο χώρο χρήστη.

Η διαδικασία που μόλις περιγράψαμε ανταποκρίνεται στη συνηθισμένη περίπτωση όπου τα δεδομένα προωθούνται απευθείας στο χώρο χρήστη. Ωστόσο, με χρήση διαφορετικών διατάξεων γραμμής είναι δυνατό να επεκταθεί η λειτουργία του συστήματος. Για παράδειγμα, χρησιμοποιείτε διαφορετική διάταξη γραμμής κάθε φορά που συνδέεστε μέσω PPP στο δίκτυο: Αντί για τη συνηθισμένη διάταξη `N_TTY`, παρεμβάλλεται η διάταξη `N_PPP`, η οποία αντί να περνά τα δεδομένα στο χώρο χρήστη, τα ομαδοποιεί σε πακέτα δικτύου και τα περνά στο δικτυακό υποσύστημα του πυρήνα. Λίστα των προκαθορισμένων διατάξεων υπάρχει στο αρχείο `<linux/tty.h>`.

6.3. Η διάταξη γραμμής του Lunix:TNG

Ο απλούστερος τρόπος ώστε το υπό κατασκευή σύστημα να έχει πρόσβαση στα πακέτα αποστέλλονται από το σταθμό βάσης, είναι να κατασκευαστεί μια νέα διάταξη γραμμής, η "Lunix:TNG line discipline", η οποία θα συλλαμβάνει τα δεδομένα εισόδου και θα τα προωθεί όχι απευθείας στο χώρο χρήστη αλλά σε κατάλληλο τμήμα

επεξεργασίας του πρωτοκόλλου των αισθητήρων, το οποίο και θα εξάγει τις τιμές των μετρούμενων μεγεθών.

Η διάταξη αυτή έχει ήδη υλοποιηθεί, ο κώδικάς της σας δίνεται στα αρχεία `lunix-ldisc.h` και `lunix-ldisc.c`. Η συνάρτηση αρχικοποίησης του `lunix-ldisc.c` εγκαθιστά τη νέα διάταξη μέσω της κλήσης `tty_register_ldisc()`, έτσι ώστε αυτή να είναι διαθέσιμη στο υπόλοιπο σύστημα. Το σύνολο της απαιτούμενης λειτουργικότητας υλοποιείται στην `lunix_ldisc_receive_buf()`, η οποία προωθεί τα εισερχόμενα δεδομένα στο τμήμα υλοποίησης του πρωτοκόλλου επικοινωνίας. Επιπλέον, παρέχεται ο κώδικας του μικρού βοηθητικού προγράμματος `lunix-attach.c` το οποίο ενεργοποιεί τη συγκεκριμένη διάταξη σε κάποια σειριακή θύρα του συστήματος, μέσω κατάλληλης κλήσης `ioctl()`. Με χρήση αυτού του εργαλείου, ορίζεται έτσι ποια θύρα θα χρησιμοποιείται από το σύστημα για παραλαβή δεδομένων από τους αισθητήρες.

Περισσότερα για την οργάνωση του στρώματος TTY του Linux μπορείτε να βρείτε στο [5] και στο κεφ. 18 του [1].

6.4. Επεξεργασία των πακέτων και εξαγωγή μετρήσεων

Το τμήμα επεξεργασίας του πρωτοκόλλου επικοινωνίας των αισθητήρων του δικτύου υλοποιείται στα αρχεία `lunix-protocol.h` και `lunix-protocol.c` που σας δίνονται. Η επεξεργασία των πακέτων βασίζεται σε μία απλή μηχανή καταστάσεων, η οποία γνωρίζει πόσα bytes δεδομένων χρειάζεται να περιμένει σε κάθε φάση της επεξεργασίας, έως ότου συμπληρωθεί ένα πλήρες πακέτο μετρήσεων.

Οι εισερχόμενοι χαρακτήρες παρέχονται στον κώδικα επεξεργασίας του πρωτοκόλλου μέσω της `lunix_protocol_received_buf()` που καλείται από τη `lunix_ldisc_receive_buf()`. Οι τιμές των μετρούμενων μεγεθών εξάγονται ως αριθμητικές ποσότητες των 16 bit χωρίς πρόσημο (τύπος `uint16_t` της C) και αποθηκεύονται σε χώρους προσωρινής αποθήκευσης. Ο κώδικας που υλοποιεί τη δέσμευση και διαχείρισή τους περιέχεται στο αρχείο `lunix-sensors.c`, ενώ ο ορισμός της δομής δεδομένων που τους περιγράφει (`struct lunix_sensor_struct`) βρίσκεται στο αρχείο `lunix.h`.

Οι ποσότητες αυτές χρειάζεται να μετατραπούν με βάση συγκεκριμένες συναρτήσεις που δίνονται από τον κατασκευαστή των αισθητήρων έτσι ώστε να εξαχθούν προσημασμένες δεκαδικές τιμές κατάλληλες για παρουσίαση και καταγραφή. Η

μετατροπή αυτή χρειάζεται λειτουργίες κινητής υποδιαστολής, για παράδειγμα διαίρεση, πολλαπλασιασμό και υπολογισμό λογαρίθμων για ποσότητες τύπου `float`. Ωστόσο, η σχεδίαση του πυρήνα του Linux καθιστά απαγορευτική την εκτέλεση πράξεων κινητής υποδιαστολής από κώδικα του kernelspace. Οι λόγοι είναι κυρίως δύο:

- Η κατάσταση της μονάδας εκτέλεσης υπολογισμών κινητής υποδιαστολής (Floating Point Unit – FPU) δε σώζεται και δεν επαναφέρεται όταν ο επεξεργαστής περνά από την εκτέλεση χώρου χρήστη στην εκτέλεση χώρου πυρήνα. Έτσι, αν ο πυρήνας αλλάξει την τιμή καταχωρητών της είναι σχεδόν σίγουρο ότι θα επηρεαστεί η ορθότητα της εκτέλεσης προγραμμάτων χρήστη.

- Ορισμένες από τις αρχιτεκτονικές στις οποίες έχει μεταφερθεί ο πυρήνας (π.χ. η αρχική αρχιτεκτονική i386 και η αρχιτεκτονική ARM) δεν υποστηρίζουν καν εντολές κινητής υποδιαστολής στο υλικό, αλλά αυτές αναλαμβάνονται από βιβλιοθήκες που τις υλοποιούν μέσω πράξεων ακεραίων, οι οποίες δε συμπεριλαμβάνονται στον πυρήνα.

Υπάρχουν δύο πιθανές λύσεις για το πρόβλημα που δημιουργείται:

- Να προωθούνται στο userspace απευθείας οι 16-bit ποσότητες, χωρίς καμία μεταβολή. Να θεωρείται ευθύνη του προγράμματος χρήστη η μετατροπή τους σε τιμές αναγνώσιμες από το χρήστη. Το πρόγραμμα είναι ελεύθερο να χρησιμοποιήσει χωρίς περιορισμό υπολογισμούς κινητής υποδιαστολής.

- Εφόσον με 16 bits κωδικοποιούνται το πολύ 65536 καταστάσεις, μπορεί η απεικόνιση των τιμών αυτών σε αναγνώσιμες τιμές να υπολογιστεί εκ των προτέρων σε πίνακες αναζήτησης (lookup tables). Μια δεκαδική τιμή σταθερής ακρίβειας, της μορφής ±xx.yyy μπορεί να αποθηκεύεται ως ο ακέραιος ±xxyyyy, οπότε χρειάζεται απλά μια διαίρεση και μια πράξη υπολοίπου για τον υπολογισμό του δεκαδικού και του ακέραιου μέρους της.

Σας δίνεται κώδικας που υλοποιεί τη δεύτερη λύση. Το αρχείο `mk_lookup_tables.c` περιέχει τις κατάλληλες συναρτήσεις ώστε να υπολογίσει τις απεικονίσεις όλου του εύρους μη προσημασμένων τιμών από `0x0000` έως `0xFFFF` και για τρία μετρούμενα μεγέθη. Το αποτέλεσμα αποθηκεύεται σε τρεις πίνακες αναζήτησης, στο αρχείο `lunix-lookup.h`.

7. Ζητήματα υλοποίησης

Στο κεφάλαιο αυτό παρουσιάζονται ορισμένα τεχνικά ζητήματα που προκύπτουν κατά τον προγραμματισμό σε χώρο πυρήνα. Επειδή αυτά εμφανίζονται πολύ συχνά και αφορούν το σύνολο σχεδόν των οδηγών συσκευών, ο πυρήνας προσφέρει υπηρεσίες (μακροεντολές και συναρτήσεις) που αναλαμβάνουν την επίλυσή τους με γενικό τρόπο, έτσι ώστε να περιορίζεται το να ξαναγράφεται κώδικας που κάνει ουσιαστικά την ίδια δουλειά και να μειώνεται ο κίνδυνος για προγραμματιστικά λάθη.

7.1. Ανάγνωση 16-bit και 32-bit τιμών από συσκευές υλικού

Ο πυρήνας του Linux έχει μεταφερθεί σε πολλές αρχιτεκτονικές. Κάποιες από αυτές είναι τύπου little-endian, κάποιες είναι big-endian. Η διαφορά έγκειται στον τρόπο που οργανώνονται στη μνήμη ακέραιες ποσότητες μεγαλύτερες του ενός byte (πχ. 16-bit και 32-bit unsigned integers): οι little-endian αρχιτεκτονικές τοποθετούν τα λιγότερο σημαντικά bytes σε μικρότερες διεύθυνσεις στη μνήμη, ενώ οι big-endian αρχιτεκτονικές τοποθετούν τα περισσότερο σημαντικά bytes σε μικρότερες διευθύνσεις.

Τα πακέτα που προέρχονται από το σταθμό βάσης είναι κωδικοποιημένα σε little-endian μορφή. Εφόσον δεν είναι δυνατό να γνωρίζουμε τη μορφή του συγκεκριμένου επεξεργαστή όπου εκτελείται ο κώδικας του οδηγού, χρειάζεται κάθε φορά που γίνεται επεξεργασία των εισερχόμενων δεδομένων να μετατρέπονται από little-endian στη φυσική μορφή του συστήματος (αν αυτό είναι little-endian δε γίνεται καμία μετατροπή). Για το σκοπό αυτό ο πυρήνας προσφέρει μια σειρά μακροεντολών, όπως οι `cpu_to_le32()` και `le32_to_cpu()`. Ο κώδικας του `lunix-protocol.c` τις χρησιμοποιεί ώστε να λειτουργεί ανεξάρτητα της υφιστάμενης αρχιτεκτονικής (δείτε τη συνάρτηση `uint16_from_packet()`) και τη σελ. 293 του [1] για περισσότερες πληροφορίες.

7.2. Πρόσβαση σε δεδομένα χώρου χρήστη

Η υλοποίηση της λειτουργίας `read()` που θα κατασκευάσετε, χρειάζεται να αντιγράφει δεδομένα προς απομονωτές χώρου χρήστη, ώστε να επιστρέφει τα δεδομένα που ζητήθηκαν από τη διεργασία. Εφόσον ο πυρήνας έχει απεριόριστη πρόσβαση στη μνήμη, θεωρεί αναξιόπιστο κάθε δεδομένο που παρέχει η διεργασία και πρέπει να ελέγχει όλες τις διευθύνσεις που αυτή περνά ως ορίσματα.

Για την πρόσβαση σε απομονωτές χώρου χρήστη ο πυρήνας προσφέρει δύο ειδικές διαδικασίες, τις `copy_from_user()` και `copy_to_user()` οι οποίες αντικαθιστούν την `memcpy()` και κάνουν ασφαλή αντιγραφή δεδομένων. Επιστρέφουν τον αριθμό των bytes που δεν μπόρεσαν να αντιγράψουν. Αν αυτός είναι μη μηδενικός, σημαίνει ότι η διεργασία προσπάθησε να κάνει πρόσβαση εκτός ορίων με δείκτη που είχε μη επιτρεπτή τιμή. Στην περίπτωση αυτή επιστρέφεται κατάλληλος κωδικός λάθους που τελικά καταλήγει στην αποστολή ενός σήματος "Segmentation Fault":

```
if (copy_to_user(usrbuf, kernel_addr, cnt))
        return -EFAULT;
```

Περισσότερα για την πρόσβαση σε δεδομένα χώρου χρήστη μπορείτε να διαβάσετε στη σελ. 64 του [1].

7.3. Μπλοκάρισμα/επανεκκίνηση διεργασιών

Ένα από τα σημαντικότερα προβλήματα που έχει να αντιμετωπίσει ο οδηγός συσκευής χαρακτήρων σας είναι τι θα συμβαίνει όταν δεν υπάρχουν νέα διαθέσιμα δεδομένα μετρήσεων και κάποια διεργασία εκτελέσει την κλήση συστήματος `read()`. Στην περίπτωση αυτή, θα πρέπει ο οδηγός να *κοιμίζει* τη διεργασία: αυτό σημαίνει ότι η διεργασία αλλάζει κατάσταση, δεν είναι πλέον «έτοιμη» προς εκτέλεση και ο χρονοδρομολογητής θα διαλέξει κάποια άλλη να εκτελεστεί στη θέση της. Ενώ είναι μπλοκαρισμένη, η διεργασία πρέπει να μετακινείται σε χωριστή ουρά διεργασιών, ανάλογα με τον αισθητήρα απ' όπου περιμένει να φτάσουν δεδομένα.

Η συμπληρωματική διαδικασία, η διαδικασία αφύπνισης, γίνεται όταν κάποιος αισθητήρας αποστείλει μετρήσεις. Αφού ανανεωθούν οι σχετικές τιμές στους sensor buffers, το τμήμα συλλογής δεδομένων του Lunix:TNG ξυπνά όσες διεργασίες κοιμόντουσαν περιμένοντας τα αποτελέσματα.

Ο κώδικας του πυρήνα ορίζει τον τύπο `wait_queue_head_t` για να περιγράψει μια ουρά αναμονής διεργασιών. Για κάθε αισθητήρα χρησιμοποιείται μια τέτοια ουρά. Όταν η `read()` διαπιστώσει ότι δεν υπάρχουν νέα δεδομένα και η διαδικασία πρέπει να κοιμηθεί, χρησιμοποιεί την κλήση `wait_event_interruptible()` ώστε η τρέχουσα διεργασία να μπλοκάρει στην ανάλογη ουρά.

Για ένα παράδειγμα χρήσης της `wait_event_interruptible()` δείτε το τμήμα κώδικα στη σελ. 153 του [1].

7.4. Process και Interrupt Context

Στον πυρήνα του Linux διακρίνονται δύο *περιβάλλοντα εκτέλεσης* (execution contexts). Ένα τμήμα κώδικα μπορεί να εκτελείται είτε σε περιβάλλον διεργασίας (process context) είτε σε περιβάλλον διακοπής (interrupt context). Στην πρώτη περίπτωση ο πυρήνας εκτελείται εκ μέρους μιας συγκεκριμένης διεργασίας, η οποία έχει πραγματοποιήσει μια κλήση συστήματος. Στη δεύτερη περίπτωση ο κώδικας του πυρήνα εκτελείται ως αποτέλεσμα μιας διακοπής υλικού, οπότε δεν υπάρχει διεργασία σχετική με την εκτέλεσή του. Το περιβάλλον εκτέλεσης επηρεάζει το είδος των λειτουργιών που μπορεί να εκτελέσει ο κώδικας. Σε process context ο πυρήνας μπορεί να *κοιμηθεί*, οπότε η κατάσταση της διεργασίας αλλάζει και ο χρονοδρομολογητής επιλέγει κάποια άλλη για εκτέλεση στη CPU. Αντίθετα, σε interrupt context απαγορεύεται η εκτέλεση κώδικα που μπορεί να κοιμηθεί, όπως είναι η πρόσβαση σε δεδομένα χώρου χρήστη (`copy_from_user()`, `copy_to_user()`) ή το κλείδωμα σηματοφορέων. Περισσότερα για τα περιβάλλοντα εκτέλεσης μπορείτε να βρείτε στη σελ. 118 του [1] και στο [4].

7.5. Κρίσιμα τμήματα στο χώρο πυρήνα

Ένα από τα σημαντικότερα προβλήματα που αντιμετωπίζει κανείς όταν γράφει για τον πυρήνα του Linux είναι η ορθή εκτέλεση του κώδικά του σε ένα περιβάλλον πολυεπεξεργασίας, όπου μπορεί να εκτελείται σε περισσότερους από έναν επεξεργαστές ταυτόχρονα και να ικανοποιεί τις αιτήσεις περισσοτέρων της μίας διεργασιών. Συχνά υπάρχουν τμήματα κώδικα (*κρίσιμα* τμήματα) τα οποία χρειάζεται να εκτελούνται αδιαίρετα και μόνο σε έναν επεξεργαστή, γιατί έχουν πρόσβαση σε μοιραζόμενα δεδομένα.

Οι δύο σημαντικότεροι μηχανισμοί που προσφέρει το Linux για την υλοποίηση κρίσιμων τμημάτων είναι οι σηματοφορείς (semaphores) και τα spinlocks.

Σηματοφορείς έχετε υλοποιήσει και χρησιμοποιήσει ήδη από το χώρο χρήστη. Στον κώδικα πυρήνα αναπαρίστανται από τη δομή `struct semaphore`. Αρχικοποιούνται με χρήση της `sema_init()`, ενώ η πράξη P, κλείδωμα του σηματοφορέα υλοποιείται από τη συνάρτηση `down()` και η πράξη V, απελευθέρωση του σηματοφορέα, από τη συνάρτηση `up()`.

Αν ο σηματοφορέας δεν είναι διαθέσιμος κατά την εκτέλεση της `down()`, η διεργασία εκ μέρους της οποίας εκτελείται ο κώδικας πυρήνα θα μπλοκαριστεί, θα *κοιμηθεί*, έως

ότου εκτελεστεί η πράξη `up()` σε αυτόν. Αυτό σημαίνει ότι η `down()` επιτρέπεται *μόνο* σε process context: η χρήση σηματοφορέων δεν είναι κατάλληλη για να κλειδωθούν δεδομένα για τα οποία ανταγωνίζεται κώδικας που τρέχει σε interrupt context, καθώς στην περίπτωση αυτή *δεν υπάρχει* διεργασία η οποία θα κοιμηθεί σε ανάλογη ουρά αναμονής. Αντί της `down()` καλό είναι να χρησιμοποιείτε, αν δεν υπάρχει ιδιαίτερος λόγος, τη συνάρτηση `down_interruptible()` στον κώδικά σας. Η διαφορά της είναι ότι η διεργασία ξυπνά όχι μόνο αν κάποιος ξεκλειδώσει το σηματοφορέα, αλλά και αν δεχθεί κάποιο σήμα. Έτσι, δεν εμφανίζεται κολλημένη για πάντα και μπορεί να δεχθεί σήματα, πχ. το σήμα `SIGINT` πατώντας Ctrl-C.

Η λειτουργία των σηματοφορέων βασίζεται στο χρονοδρομολογητή (scheduler) του συστήματος: όταν κάποια διεργασία καλέσει την `down()` σε κλειδωμένο σηματοφορέα, κοιμάται. Αντίθετα, στην περίπτωση των spinlocks, ένα τμήμα κώδικα που δεν μπορεί να πάρει το κλείδωμα γιατί είναι κατειλημμένο θα προσπαθεί συνεχώς έως ότου τα καταφέρει χωρίς να αφήσει τη CPU ("spins" σε busy-wait loop). Αυτό κάνει τα spinlocks κατάλληλα για χρήση σε interrupt context, όπου για παράδειγμα μια δομή δεδομένων μπορεί να μεταβάλλεται και από κώδικα που τρέχει σε process context και από έναν interrupt handler (π.χ. ο TTY buffer προστατεύεται με spinlock, γιατί γίνεται πρόσβαση σε αυτόν από interrupt context). Οι κύριες συναρτήσεις χειρισμού spinlocks είναι η `spin_lock()`, `spin_unlock()`, `spin_lock_init()`.

Στην περίπτωση του υπό κατασκευή οδηγού, οι Lunix sensor buffers προστατεύονται με spinlocks. Αυτό συμβαίνει γιατί η συνάρτηση που τους ανανεώνει τρέχει σε interrupt context, όταν παραληφθούν δεδομένα από τη σειριακή θύρα.

Περισσότερα για τα κλειδώματα στον πυρήνα και τους περιορισμούς που επιβάλλουν μπορείτε να βρείτε στο κεφ. 5 του [1].

8. Πρακτικά ζητήματα

8.1. Κατασκευή του οδηγού

Έχοντας διαβάσει όλα τα παραπάνω, ίσως αναρωτιέστε από ποιο σημείο θα πρέπει να αρχίσετε για την κατασκευή του ζητούμενου οδηγού.

Η ανάπτυξη θα γίνει μέσα σε εικονική μηχανή, στην οποία θα έχετε δικαιώματα διαχειριστή (root). Για τη συγγραφή και μεταγλώττιση του κώδικα θα κάνετε login ως απλοί χρήστες και θα έχετε ορισμένα τερματικά ανοιχτά ως root ώστε να μπορείτε να

παρακολουθείτε το αρχείο καταγραφής του πυρήνα και να προσθαφαιρείτε το module σας.

Ξεκινήστε από τον κώδικα που σας δίνεται. Ο κώδικας που χρειάζεται να προσθέσετε εντάσσεται στα αρχεία `lunix-chrdev.h` και `lunix-chrdev.c`, τα οποία περιέχουν ήδη ένα σύντομο σκελετό. Υπάρχουν σχόλια σε σημεία που πρέπει να κάνετε αλλαγές ή να προσθέσετε κώδικα, ενώ πολύ χρήσιμη θα βρείτε την υλοποίηση του `scull` και των παραλλαγών του στο [1].

Οπότε, διαβάζετε τμήματα του υπάρχοντα κώδικα, ξεκινάτε να προσθέτετε τον δικό σας και μεταγλωττίζετε με χρήση της εντολής `make` και του παρεχόμενου `Makefile`.

Με την εντολή `dmesg` μπορείτε να δείτε τα τελευταία μηνύματα του πυρήνα (πολύ χρήσιμη για debugging με διάσπαρτες κλήσεις `printk()` / `debug()` για να μπορείτε να παρακολουθείτε πώς τρέχει ο κώδικάς σας). Για να παρακολουθείτε το αντίστοιχο αρχείο καταγραφής των μηνυμάτων του πυρήνα καθώς προστίθενται νέες εγγραφές σε αυτό, τρέξτε `less /var/log/kern.log` και πιέστε `Shift-f`.

Τέλος, υπάρχει το βασισμένο στο Web εργαλείο LXR [3], το οποίο είναι ιδανικό για εξερεύνηση του κώδικα του πυρήνα και για αναφορά σχετικά με τα ορίσματα και τον τρόπο χρήσης των υπηρεσιών που προσφέρει.

8.2. Έλεγχος σωστής λειτουργίας του οδηγού

Για τον έλεγχο της σωστής λειτουργίας του οδηγού σας, μπορείτε να ακολουθήσετε τη διαδικασία που περιγράφεται στο παράδειγμα της §4.2, με χρήση της `cat` στα ειδικά αρχεία. Για να ελέγξετε ότι όλα λειτουργούν σωστά όταν πολλαπλές διεργασίες προσπαθούν να έχουν πρόσβαση στα δεδομένα, μπορείτε να ανοίξετε διαφορετικά τερματικά και να τρέξετε πολλές φορές της `cat` ταυτόχρονα, στο ίδιο ή σε διαφορετικά αρχεία.

Χρησιμοποιώντας τις υπηρεσίες του οδηγού που κατασκευάσατε, μπορείτε να χτίσετε στο χώρο χρήστη περισσότερο πολύπλοκες δυνατότητες, για παράδειγμα ένα πρόγραμμα το οποίο θα λαμβάνει περιοδικά, θα εμφανίζει και θα καταγράφει σε αρχεία τις μετρούμενες τιμές.

8.3. Λειτουργία του οδηγού σε περιβάλλον QEMU-KVM

8.3.1. Χρήση σε μηχάνημα του εργαστηρίου

Το παράδειγμα της §4.2 αναφέρεται σε χρήση του οδηγού σε φυσικό σύστημα, όπου ο σταθμός βάσης εμφανίζεται συνδεδεμένος σε συσκευή Serial-over-USB, `/dev/ttyUSB1`. Για τις ανάγκες του εργαστηρίου, η ανάπτυξη γίνεται σε περιβάλλον εικονικής μηχανής QEMU-KVM. Στην περίπτωση αυτή, αν είστε σε μηχάνημα του εργαστηρίου και έχετε πρόσβαση σε πραγματικό αισθητήρα, χρειάζεται να *απεικονιστεί* η θύρα `/dev/ttyUSB1` στην εικονική πρώτη σειριακή θύρα του QEMU. Οπότε, το παράδειγμα ισχύει όπως είναι, ο οδηγός συσκευής δεν μπορεί να καταλάβει ότι εκτελείται μέσα σε εικονική μηχανή, αρκεί να χρησιμοποιηθεί η `/dev/ttyS0` αντί της `/dev/ttyUSB1`.

Η απεικόνιση της `/dev/ttyUSB1` του host στην πρώτη σειριακή του VM γίνεται με χρήση της παραμέτρου `-chardev tty,id=sensors0,path=/dev/ttyUSB1` στη γραμμή εντολών του script που εκτελεί το QEMU-KVM. Από τη στιγμή αυτή, κάθε πρόσβαση στην εικονική σειριακή `/dev/ttyS0` οδηγεί σε πρόσβαση στη θύρα `/dev/ttyUSB1`.

8.3.2. Χρήση σε μηχάνημα εκτός εργαστηρίου

Μέρος της ανάπτυξης του οδηγού πιθανότατα θα γίνει σε μηχανήματα εκτός εργαστηρίου, όπου δεν υπάρχει διαθέσιμος αισθητήρας, συνδεδεμένος στη θύρα `/dev/ttyUSB1`. Παράδειγμα τέτοιου μηχανήματος είναι εικονική μηχανή QEMU-KVM σε προσωπικό σας μηχάνημα, ή στο cloud ~okeanos. Προς διευκόλυνσή σας, έχει εγκατασταθεί στο μηχάνημα `cerberus.cslab.ece.ntua.gr` εξυπηρετητής TCP/IP, ο οποίος εκπέμπει συνεχώς μετρήσεις από σταθμό βάσης μόνιμα συνδεδεμένο στο συγκεκριμένο σύστημα. Ο εξυπηρετητής είναι προσβάσιμος στην θύρα TCP `49152`. Οπότε, στην πλευρά του πελάτη χρειάζεται να συνδέσετε, μέσω δικτύου, την εικονική σειριακή θύρα του QEMU με τον εξυπηρετητή που εκπέμπει πακέτα δεδομένων από το δίκτυο αισθητήρων. Το script `utopia.sh` που σας δίνεται, ανακατευθύνει αυτόματα την πρώτη σειριακή θύρα του εικονικού μηχανήματος (`/dev/ttyS0`) στη θύρα TCP `cerberus.cslab.ece.ntua.gr:49152`, χωρίς να

απαιτείται κάποια άλλη ενέργεια από εσάς. Δείτε τις παραμέτρους `-chardev`, `-device` στη γραμμή που ξεκινά τη διεργασία του QEMU.

9. Πιθανές επεκτάσεις

Έχοντας υλοποιήσει τον οδηγό συσκευής όπως περιγράφεται στα προηγούμενα, θα έχετε ένα πλήρως λειτουργικό σύστημα για την ανάκτηση μορφοποιημένων τιμών των μετρήσεων διαφορετικών αισθητήρων, βασισμένο στην κλήση συστήματος `read()`.

Δύο πιθανές βελτιώσεις/επεκτάσεις που μπορείτε να εξερευνήσετε για το συγκεκριμένο σύστημα, αν το επιθυμείτε, είναι:

- *Υποστήριξη κλήσεων ioctl() για τη μεταβολή της συμπεριφοράς του οδηγού:* Μπορείτε να ενσωματώσετε νέες εντολές `ioctl()` ώστε να αλλάζετε δυναμικά τη συμπεριφορά του οδηγού. Ένα παράδειγμα είναι να υποστηρίζεται εναλλαγή ανάμεσα σε “raw”/”cooked” τρόπους λειτουργίας της συσκευής χαρακτήρων: ο “cooked” τρόπος θα περνάει στο userspace μορφοποιημένες τις μετρούμενες τιμές, όπως περιγράφεται στα προηγούμενα, ενώ ο “raw” τρόπος θα περνάει χωρίς καμία μεταβολή τις 16-bit ποσότητες που επιστρέφονται από τους αισθητήρες, ώστε η επεξεργασία τους [π.χ. απεικόνιση σε δεκαδικές ποσότητες] να γίνεται από κατάλληλο πρόγραμμα χώρου χρήστη.

- *Υποστήριξη επικοινωνίας χώρων πυρήνα-χρήστη χωρίς κλήση συστήματος read(), με memory-mapped I/O.*

  Η χρήση κλήσεων `read()` για την επικοινωνία του χώρου χρήστη με το χώρο πυρήνα δεν είναι ο μοναδικός τρόπος με τον οποίο μια διεργασία μπορεί να ανακτήσει τις μετρήσεις των αισθητήρων. Μία εναλλακτική μέθοδος, η οποία δεν εισάγει το κόστος της πραγματοποίησης της κλήσης συστήματος και της αντιγραφής των δεδομένων από το χώρο πυρήνα στο χώρο χρήστη, είναι η *απεικόνιση* των sensor buffers απευθείας στον χώρο εικονικών διευθύνσεων μιας διεργασίας, μόνο για ανάγνωση. Με τον τρόπο αυτό, η διεργασία θα μπορεί να διαβάζει *απευθείας* τις τιμές που χρειάζεται από τους sensor buffers που βρίσκονται μέσα στον πυρήνα, και να έχει πρόσβαση στις πιο πρόσφατες μετρήσεις, στην αρχική, μη μορφοποιημένη εκδοχή τους.

Η απεικόνιση θα εγκαθίσταται με χρήση της κλήσης συστήματος `mmap()` στο ειδικό αρχείο και πρέπει να αφορά μόνο το μετρούμενο μέγεθος και τον αισθητήρα που αντιστοιχούν στο αρχείο αυτό, ώστε το σύστημα να παραμένει ασφαλές. Τι επιπτώσεις έχει αυτό στον τρόπο με τον οποίο είναι οργανωμένοι οι sensor buffers; [*Υπόδειξη:* η απεικόνιση μνήμης αφορά πάντοτε ακέραιο αριθμό σελίδων μνήμης].

Περισσότερα για την υλοποίηση της μεθόδου `mmap()` από τη συσκευή σας μπορείτε να βρείτε στη σελ. 422 του [1].

10. Αναφορές - Βιβλιογραφία

[1] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, "Linux Device Drivers", 3rd Edition, O'Reilly, διαθέσιμο ελεύθερα στο http://lwn.net/Kernel/LDD3/.

[2] "Teleprinter", Wikipedia article, http://en.wikipedia.org/wiki/Teleprinter/.

[3] "the Linux Cross Reference", http://lxr.linux.no/.

[4] Matthew Wilcox, "I'll Do It Later: Softirqs, Tasklets, Bottom Halves, Task Queues, Work Queues and Timers", Linux.Conf.Au 2003, http://www.wil.cx/matthew/lca2003/paper.pdf.

[5] Alessandro Rubini, "Serial Drivers", http://www.linux.it/~rubini/docs/serial/serial.html