ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
`http://www.cslab.ece.ntua.gr`

Εργαστήριο Λειτουργικών Συστημάτων

8ο εξάμηνο, Ακαδημαϊκή περίοδος 2012-2013

Κανονική Εξέταση – Λύσεις

Το παρόν περιγράφει πλήρη λύση των θεμάτων, με σύντομες απαντήσεις. Για να βοηθήσει στην καλύτερη κατανόηση των απαντήσεων, προσφέρει αναλυτική επεξήγησή τους, η οποία δεν ήταν απαραίτητη για να θεωρείται τέλεια η λύση.

Θέμα 1 (40%)

Μια κατασκοπευτική οργάνωση επιθυμεί να παρακολουθεί το σύνολο των κλήσεων που πραγματοποιούνται μέσω παρόχων τηλεφωνίας. Τα τηλεφωνικά κέντρα κάθε παρόχου τηλεφωνίας διασυνδέονται με την οργάνωση μέσω εξειδικευμένου υλικού. Το υλικό προωθεί πληροφορίες που αφορούν σε γεγονότα (events) για κάθε συνδιάλεξη και μπορεί κατ'επιλογή να προγραμματιστεί ώστε να προωθεί και το περιεχόμενο επιλεγμένων συνδιαλέξεων (κλήσεις "ειδικού ενδιαφέροντος"), ως δείγματα PCM. Γεγονότα αποτελούν η έναρξη κι η λήξη μιας τηλεφωνικής συνδιάλεξης. Κάθε γεγονός χαρακτηρίζεται από τον αριθμό του καλούντα, τον αριθμό του καλούμενου, ένα αναγνωριστικό του παρόχου, και από ένα μοναδικό αναγνωριστικό της συγκεκριμένης κλήσης, μήκους 32 bits.

Το υλικό διασυνδέεται μέσω κατάλληλης διεπαφής με υπολογιστικό σύστημα, το οποίο αναλαμβάνει την καταγραφή κι επεξεργασία των εισερχόμενων γεγονότων και του ήχου.

Θεωρήστε τον οδηγό συσκευής για το υλικό καταγραφής σε ΛΣ Linux. Τα δεδομένα γίνονται διαθέσιμα σε διεργασίες προς επεξεργασία μέσω ειδικού αρχείου συσκευής χαρακτήρων `/dev/prism`*, το οποίο υλοποιεί ο οδηγός της συσκευής.*

Υποθέτουμε ότι στο υπολογιστικό σύστημα εκτελούνται μία ή περισσότερες διεργασίες, οι οποίες αναλύουν εισερχόμενα γεγονότα κλήσεων. Αν ανάμεσα σε αυτά εντοπιστεί κλήση ειδικού ενδιαφέροντος, μια διεργασία ζητά μέσω του οδηγού από το υλικό την ηχητική καταγραφή της και καταγράφει το περιεχόμενό της σε αρχείο στο δίσκο.

Το υλικό παρέχει πακέτα τα οποία είτε περιγράφουν ένα νέο γεγονός κλήσης, είτε περιέχουν ηχητικά δεδομένα για μια κλήση. Το υλικό προκαλεί διακοπή διακοπή κάθε φορά που ένα τέτοιο πακέτο είναι διαθέσιμο. Ο οδηγός αποθηκεύει προσωρινά τα εισερχόμενα δεδομένα σε κατάλληλο κυκλικό απομονωτή, αναλόγως του είδους τους.

Από την πλευρά των διεργασιών ισχύουν τα εξής:

1. *Κάθε διεργασία που ανοίγει το ειδικό αρχείο ανακτά ένα ή περισσότερα εισερχόμενα γεγονότα κλήσεων (δομή `struct prism_event`) με κάθε εκτέλεση της `read()`.*
2. *Ο οδηγός δεν θα επιστρέψει ποτέ σε διεργασία ένα γεγονός που έχει ήδη επιστραφεί από οποιαδήποτε προηγούμενη κλήση `read()`. Αν δεν υπάρχουν δεδομένα προς επιστροφή, η `read()` μπλοκάρει.*
3. *Όταν κριθεί αναγκαίο, μια διεργασία ζητά την καταγραφή μιας συνομιλίας με χρήση κατάλληλης κλήσης `ioctl()` σε ανοιχτό αρχείο της συσκευής. Επόμενες κλήσεις `read()` στο αρχείο αυτό δεν επιστρέφουν πλέον γεγονότα κλήσεων, αλλά bytes (δείγματα PCM) που κωδικοποιούν τον ήχο της συνομιλίας. Μόνο ένα αρχείο μπορεί να λαμβάνει δεδομένα για συγκεκριμένη κλήση κάθε φορά. Σε αυτόν τον τρόπο λειτουργίας, μια διεργασία μπορεί να διαβάσει οποιονδήποτε αριθμό από bytes.*
4. *Η καταγραφή ηχητικών δεδομένων μιας κλήσης τερματίζεται όταν η κλήση ολοκληρωθεί ή όταν γεμίσει ο αντίστοιχος κυκλικός απομονωτής. Όταν ο οδηγός σταματήσει την καταγραφή κι ο απομονωτής αδειάσει, κάθε επόμενη κλήση `read()` επιστρέφει EOF.*
5. *Αν οι διεργασίες δεν μπορούν να αντεπεξέλθουν στον εισερχόμενο ρυθμό γεγονότων, ο οδηγός εξασφαλίζει ότι ποτέ δεν θα επιστραφεί σε διεργασία παλαιότερο γεγονός από ένα που έχει ήδη επιστραφεί προς επεξεργασία, κρατώντας τα πιο πρόσφατα.*
6. *Το ρεύμα εισερχόμενων γεγονότων δεν επιστρέφει ποτέ EOF.*

Δίνονται τα εξής:

- `prism_intr()`:
  Συνάρτηση χειρισμού διακοπών υλικού. Αναλόγως του είδους του πακέτου που λαμβάνεται από το υλικό, αποθηκεύει ένα νέο εισερχόμενο γεγονός κλήσης ή ηχητικά δεδομένα σε κατάλληλη δομή `prism_buffer`.
- `get_hw_prism_packet(pkt)`:
  Διαβάζει ένα νέο εισερχόμενο πακέτο από το hardware και το αποθηκεύει στη δομή `pkt`.
- `start_hw_recording(call_id)`:
  Προγραμματίζει το υλικό να καταγράφει ήχο από την συνομιλία `call_id`.
- `alloc_buf_for_call(call_id)`:
  Δεσμεύει κι επιστρέφει μια νέα δομή `prism_buffer`, την οποία συσχετίζει με την κλήση με αναγνωριστικό `call_id`. Επιστρέφει `NULL` σε περίπτωση σφάλματος, π.χ. υπάρχει ήδη δομή που σχετίζεται με τη συγκεκριμένη κλήση.
- `get_buf_for_call(call_id)`:
  Επιστρέφει δείκτη στη δομή `prism_buffer` που σχετίζεται με την κλήση με αναγνωριστικό `call_id`, ή `NULL` αν δεν υπάρχει τέτοια δομή.
- `free_buf_for_call(call_id)`:
  Απελευθερώνει τη δομή `prism_buffer` που σχετίζεται με την κλήση με αναγνωριστικό `call_id`. Δεν πρέπει να κληθεί για αναγνωριστικό κλήσης για το οποίο δεν υπάρχει δομή `prism_buffer`.

- `event_buf`:
  Καθολικός απομονωτής που κρατά εισερχόμενα γεγονότα κλήσεων.

Ζητούνται τα εξής:

i. *(5%) Ποιες οντότητες χρησιμοποιούν ταυτόχρονα στιγμιότυπα της δομής `struct prism_buffer`; Τι πρόβλημα δημιουργεί αυτό; Προσθέστε ο,τιδήποτε χρειάζεται στον κώδικα ώστε να αποφευχθεί το πρόβλημα κι η δομή να χρησιμοποιείται με ασφαλή τρόπο.*
ii. *(5%) Πότε δεσμεύεται και πότε απελευθερώνεται μια δομή `struct prism_buffer`;*
iii. *(5%) Χωρίς να δώσετε κώδικα, περιγράψτε τη λειτουργία της `prism_chrdev_ioctl()`. Ποια η χρησιμότητά της και ποιες δομές δεδομένων επηρεάζει κατά τη λειτουργία της; Τι προβλήματα μπορεί να αντιμετωπίσει και πώς τα χειρίζεται;*
iv. *(5%) Πώς είναι δυνατό να κατανεμηθεί η επεξεργασία των ηχητικών δεδομένων μίας κλήσης σε δύο διεργασίες;*
v. *(5%) Υλοποιήστε την `prism_intr()` συμπληρώνοντας το σκελετό. Πώς ο κώδικάς σας ικανοποιεί την προδιαγραφή 5; Υπάρχει περίπτωση να μην μπορεί να χειριστεί εισερχόμενο πακέτο ήχου; Αν ναι, δώστε ένα σενάριο.*
vi. *(10%) Υλοποιήστε την `prism_chrdev_read()`, συμπληρώνοντας τον σκελετό. Πώς εντοπίζει αν πρέπει να επιστρέψει γεγονότα κλήσεων ή bytes ήχου; Πότε επιστρέφει EOF και πώς το εντοπίζει;*
vii. *(5%) Για κάθε ένα από τα ακόλουθα, απαντήστε αν εκτελείται σε interrupt ή process context: (α) γραμμή 52, (β) γραμμή 71, (γ) συνάρτηση `get_buf_for_call()`, (δ) συνάρτηση `alloc_buf_for_call()`.*

Αν το χρειαστείτε, μπορείτε να προσθέσετε νέες δομές ή μεταβλητές, πεδία σε υπάρχουσες δομές, ή νέες συναρτήσεις, αρκεί να περιγράψετε με ακρίβεια τη λειτουργία τους.

```
1  #define NUMBER_LEN 10 /* Maximum length of a phone number */
2  #define PCM_SAMPLES_LEN 128 /* Length of PCM data in a single sound packet */
3
4  struct prism_event {
5      uint32_t type; /* one of CALL_START, CALL_END */
6      uint32_t call_id;
7      uint32_t provider_id;
8      char calling_number[NUMBER_LEN];
9      char called_number[NUMBER_LEN];
10     ...
11 }; /* Assume length of struct is 128 bytes */
12
13 struct prism_sound {
14     uint32_t type; /* must be CALL_SOUND */
15     uint32_t call_id;
16     char data[PCM_SAMPLES_LEN];
17 };
18
19 union prism_packet {
20     uint32_t type; /* one of CALL_START, CALL_END, CALL_SOUND */
21     struct prism_event event;
22     struct prism_sound sound;
23 };
```

```
24
25  struct prism_buffer {
26      wait_queue_head_t wq;
27      uint128_t wcnt, rcnt; /* These are initialized to zero, and
28                             * will never wrap. */
29  #define CIRC_BUF_SIZE (1024 * 1024)
30      char circ_buffer[CIRC_BUF_SIZE];
31      ...
32  };
33
34  void get_hw_prism_packet(union prism_packet *pkt);
35  void start_hw_recording(uint32_t call_id);
36  struct prism_buffer *alloc_buf_for_call(uint32_t call_id);
37  struct prism_buffer *free_buf_for_call(uint32_t call_id);
38  struct prism_buffer *get_buf_for_call(uint32_t call_id);
39
40  struct prism_buffer event_buf; /* Buffer holding call events */
41
42  static void prism_intr(void)
43  {
44      struct prism_buffer *buf;
45      union prism_packet pkt;
46
47      get_hw_prism_packet(&pkt);
48      if (pkt->type == CALL_START_EVENT || pkt-> type == CALL_END_EVENT) {
49          /* hw packet contains information on a call event */
50          buf = &event_buf;
51          ...
52          memcpy(&buf->circ_buffer[buf->wcnt % CIRC_BUF_SIZE],
53              &pkt.event, sizeof(struct prism_event));
54          buf->wcnt += sizeof(struct prism_event);
55          ...
56      } else if (pkt->type == CALL_SOUND) {
57          /* hw packet contains call sound data */
58          buf = get_buf_for_call(pkt.sound.call_id);
59          ...
60          memcpy(&buf->circ_buffer[buf->wcnt % CIRC_BUF_SIZE],
61                 &pkt.sound.data, PCM_SAMPLES_LEN);
62          buf->wcnt += PCM_SAMPLES_LEN;
63          ...
64      } else
65          printk(KERN_ERR "Internal error: Received hw packet of unknown type");
66      ...
67  }
68
69  static int prism_chrdev_open(struct inode *inode, struct file *filp)
70  {
71      filp->private_data = NULL; /* Initially return call events */
72      ...
73  }
74
75  static int prism_chrdev_release(struct inode *inode, struct file *filp)
76  {
77      ...
78  }
79
80  static long prism_chrdev_ioctl(struct file *filp, unsigned int cmd,
81      unsigned long arg)
82  {
83      /*
```

```
84          * Implement PRISM_IOC_GETCALLDATA,
85          * manipulate filp->private_data accordingly
86          */
87         ...
88     }
89
90     static ssize_t prism_chrdev_read(struct file *filp, char __user *usrbuf,
91         size_t cnt, loff_t *offp)
92     {
93         struct prism_buffer *buf = filp->private_data;
94
95         /* Determine whether to return events or sound data */
96         /* If returning call events, only return whole events */
97         ...
98
99         /* Retrieve data, block appropriately, or return EOF */
100        ...
101    }
```

i. Οι διεργασίες που καλούν `read()` για να βγάλουν δεδομένα από ένα `prism_buffer` συναγωνίζονται η μία την άλλη και τη συνάρτηση χειρισμού διακοπών, η οποία αποθηκεύει εισερχόμενα δεδομένα μέσα σε δομές `prism_buffer`. Για να αποφευχθεί το πρόβλημα συγχρονισμού, θα χρησιμοποιήσουμε ένα spinlock ανά `prism_buffer`, προσθέτοντας πεδίο `spinlock_t lock` και παίρνοντας το κλείδωμα σε κάθε σημείο όπου γίνεται πρόσβαση. Στην περίπτωση του interrupt handler, κλειδώνουμε με `spin_lock_irqsave()` στις γραμμές 51, 59 και ξεκλειδώνουμε με `spin_lock_irqrestore()` στις γραμμές 55, 63. Βλ. παρακάτω για τη `read()`.

ii. Μια δομή `struct prism_buffer` δεσμεύεται όταν μια διεργασία εντοπίσει στο ρεύμα εισερχομένων γεγονότων κλήση ειδικού ενδιαφέροντος, οπότε ζητάει την καταγραφή δεδομένων με κατάλληλη κλήση `ioctl()`. Η δομή *δεν* απελευθερώνεται όταν έρθει το γεγονός λήξης της αντίστοιχης συνδιάλεξης, καθώς είναι πιθανό να μην έχουν ανακτηθεί ακόμη όλα τα δεδομένα ήχου από διεργασία, αλλά όταν κλείσουν όλοι οι περιγραφητές αρχείου στην αντίστοιχη δομή `struct file` και αυτή πρέπει να απελευθερωθεί (συνάρτηση `prism_chrdev_release()`).

iii. Η `prism_chrdev_ioctrl()` δεσμεύει μια νέα δομή `struct prism_buffer`, ώστε το ανοιχτό αρχείο για το οποίο κλήθηκε να ξεκινήσει να επιστρέφει bytes ηχητικών δεδομένων για συγκεκριμένη κλήση. Χρησιμοποιεί την κλήση `alloc_buf_for_call()` για να δεσμεύσει έναν νέο απομονωτή, τον οποίο συσχετίζει με το ανοιχτό αρχείο μέσω του δείκτη `filp->private_data`. Μπορεί να αντιμετωπίσει το πρόβλημα η κλήση `alloc_buf_for_call()` να αποτύχει (π.χ., η συγκεκριμένη κλήση να καταγράφεται *ήδη*), οπότε επιστρέφει κατάλληλο κωδικό λάθους, π.χ. `EINVAL` (Invalid Argument), ή `EBUSY` (Device or Resource Busy).

iv. Δύο διαφορετικά αρχεία δεν είναι δυνατό να λαμβάνουν ηχητικά δεδομένα της ίδιας κλήσης (επιβάλλεται από την `ioctl()` με βάση την προδιαγραφή 3). Οπότε οι δύο διεργασίες πρέπει να αναφέρονται στο *ίδιο* ανοιχτό αρχείο, στο ίδιο υφιστάμενο `struct file`. Ένας τρόπος να επιτευχθεί αυτό είναι η μία διεργασία να είναι παιδί της άλλης, με `fork()`, οπότε κληρονομεί το ανοιχτό αρχείο.

v. Ακολουθεί η υλοποίηση της `prism_intr()`. Ο κώδικας ικανοποιεί την προδιαγραφή 5 με το να προχωράει το δείκτη `buf->rcnt` μόνο όταν οι διεργασίες δεν διαβάζουν αρκετά γρήγορα εισερχόμενα γεγονότα κι ο απομονωτής `event_buf` γεμίσει, ουσιαστικά πετώντας παλιότερα δεδομένα. Υπάρχει περίπτωση να μην μπορεί να χειριστεί

εισερχόμενο πακέτο ήχου, γιατί πλέον δεν υπάρχει buffer συσχετισμένος με τη συγκεκριμένη κλήση. Αυτό μπορεί να συμβεί αν μια διεργασία ζητήσει να καταγραφεί μια συνομιλία και στη συνέχεια κλείσει πρόωρα το ανοιχτό αρχείο, πριν από την ολοκλήρωσή της.

```
1   static void prism_intr(void)
2   {
3       struct prism_buffer *buf = NULL, *buf2;
4       union prism_packet pkt;
5       unsigned long flags, flags2;
6
7       get_hw_prism_packet(&pkt);
8       if (pkt->type == CALL_START_EVENT || pkt->type == CALL_END_EVENT) {
9           /* hw packet contains information on a call event */
10          buf = &event_buf;
11          spin_lock_irqsave(&buf->lock, flags);
12          memcpy(&buf->circ_buffer[buf->wcnt % CIRC_BUF_SIZE],
13              &pkt.event, sizeof(struct prism_event));
14          buf->wcnt += sizeof(struct prism_event);
15          if (bur->wcnt - buf->rcnt > CIRC_BUF_SIZE)
16              buf->rcnt += sizeof(struct prism_event); /* Προδιαγραφή 5 */
17          if (pkt->type == CALL_END_EVENT) {
18              buf2 = get_buf_for_call(pkt.sound.call_id);
19              if (buf2) {
20                  spin_lock_irqsave(&buf2->lock, flags2);
21                  buf2->terminated = 1;
22                  wake_up_interruptible(&buf2->wq);
23                  spin_unlock_irqrestore(&buf2->lock, flags2);
24              }
25          }
26          spin_unlock_irqrestore(&buf->lock, flags);
27      } else if (pkt->type == CALL_SOUND) {
28          /* hw packet contains call sound data */
29          buf = get_buf_for_call(pkt.sound.call_id);
30          spin_lock_irqsave(&buf->lock, flags);
31          if (buf->wcnt - buf->rcnt == CIRC_BUF_SIZE)
32              buf->terminated = 1;  /* Προδιαγραφή 4 */
33          if (!buf->terminated) {
34              memcpy(&buf->circ_buffer[buf->wcnt % CIRC_BUF_SIZE],
35                     &pkt.sound.data, PCM_SAMPLES_LEN);
36              buf->wcnt += PCM_SAMPLES_LEN;
37          }
38          spin_unlock_irqrestore(&buf->lock, flags);
39      } else
40        printk(KERN_ERR "Internal error: Received hw packet of unknown type");
41      if (buf)
42          wake_up_interruptible(&buf->wq);
43  }
```

vi. Ακολουθεί η υλοποίηση της `prism_chrdev_read()`. Η συνάρτηση εντοπίζει αν πρέπει να επιστρέψει γεγονότα κλήσεων ή bytes ήχου ελέγχοντας αν ο δείκτης `filp->private_data` δείχνει στον καθολικό buffer `event_buf` ή σε νέο `struct prism_buffer` που έχει δεσμευτεί με αίτηση `ioctl()`. Επιστρέφει `EOF` όταν ο buffer δεν περιέχει πλέον δεδομένα *και* η αντίστοιχη κλήση έχει τερματιστεί. Είναι ευθύνη του χειριστή διακοπών να θέσει ανάλογη σημαία `terminated`, με την οποία επεκτείνουμε τη δομή, όταν εντοπίσει εισερχόμενο γεγονός τερματισμού κλήσης, ή όταν ο αντίστοιχος απομονωτής γεμίσει (προδιαγραφή 4).

```
 1  static ssize_t prism_chrdev_read(struct file *filp, char __user *usrbuf,
 2      size_t cnt, loff_t *offp)
 3  {
 4      struct prism_buffer *buf = filp->private_data;
 5      unsigned long flags;
 6      int evcnt = 0;
 7
 8  #define MAX_CNT_PER_READ 1024
 9      char tmpbuf[MAX_CNT_PER_READ];
10
11      /* Determine whether to return events or sound data */
12      if (buf == &event_buf) {
13          /* If returning call events, only return whole events */
14          evcnt = cnt / sizeof(struct prism_event);
15          if (evcnt == 0)
16              return -EINVAL;
17          cnt = evcnt * sizeof(struct prism_event);
18      }
19
20      if (cnt > MAX_CNT_PER_READ)
21          cnt = MAX_CNT_PER_READ;
22
23      /* Retrieve data, block appropriately, or return EOF */
24      spin_lock_irsqsave(&buf->lock, flags);
25      while (buf->rcnt == buf->cnt && !buf->terminated) {
26          spin_unlock_irqrestore(&buf->lock, flags);
27          if (wait_event_interruptible(buf->wq,
28                  (buf->rcnt != buf->wcnt || buf->terminated)))
29              ret = -ERESTARTSYS;
30          spin_lock_irqsave(&buf->lock, flags);
31      }
32
33      /* Return EOF if buffer is empty and call has terminated */
34      if (buf->rcnt == buf->wcnt && buf->terminated) {
35          spin_unlock_irqrestore(&buf->lock, flags);
36          return 0;
37      }
38
39      for (i = 0; i < cnt; ++i, ++buf->rcnt)
40          tmpbuf[i] = buf->circ_buffer[buf->rcnt & CIRC_BUF_SIZE];
41      spin_unlock_irqrestore(&buf->lock, flags);
42
43      if (copy_to_user(usrbuf, tmpbuf, cnt))
44          return -EFAULT;
45
46      return cnt;
47  }
```

vii. (α) interrupt context, είναι μέσα στον interrupt handler, (β) process context, είναι μέσα σε συνάρτηση που υλοποιεί μέθοδο του `struct file`, (γ) interrupt context, δεδομένου ότι η χρησιμοποιείται μόνο από τον interrupt handler κι η `prism_chrdev_read()` χρησιμοποιεί απευθείας τον απομονωτή μέσω του `filp->private_data`, (δ) process context, δεδομένου ότι την καλεί μόνο η `prism_chrdev_ioctl()`.

Λεπτό σημείο: Υπάρχει πιθανό race ανάμεσα στη *χρήση* ενός buffer από τον interrupt handler και την *απελευθέρωσή* του, από την `prism_chrdev_release()`: θα μπορούσε η διεργασία να κλείσει το ανοιχτό αρχείο και να απελευθερώσει τον buffer, ενώ ο interrupt

handler ακόμη τον χρησιμοποιεί. Δεν υπάρχει αντίστοιχο race με την `prism_chrdev_read()` γιατί ο πυρήνας εξασφαλίζει ότι δεν υπάρχει περίπτωση να κληθεί η `prism_chrdev_release()` ενώ υπάρχουν ακόμη file descriptors διεργασιών που αναφέρονται σε αυτό το `struct file`. Ένας τρόπος επίλυσης του προβλήματος είναι χρήση reference count πάνω σε κάθε δομή `prism_buffer` και συνάρτησης `put_buf_for_call()`, συμπληρωματικής της `get_buf_for_call()`. Εναλλακτικά, θα μπορούσε να χρησιμοποιηθεί μοναδικό spinlock, το οποίο θα εξασφάλιζε ότι η `prism_chrdev_release()` δεν θα μπορούσε να εκτελεστεί παράλληλα με τον interrupt handler.

Θέμα 2 (30%)

α. *(5%)* *Έστω N διεργασίες (N: άρτιος), οι οποίες τρέχουν το πρόγραμμα `prog1` και ανοίγουν (`open()`) το ίδιο ειδικό αρχείο (`/dev/somedevice`). Πόσα στιγμιότυπα της δομής `struct file` δημιουργεί ο πυρήνας; Στη συνέχεια οι $\frac{N}{2}$ διεργασίες εκτελούν την κλήση συστήματος `fork()` και οι υπόλοιπες $\frac{N}{2}$ διεργασίες εκτελούν την κλήση συστήματος `execve("prog2", ...)`. Τελικά, πόσες διεργασίες εκτελούν το `prog1` και πόσες εκτελούν το `prog2`; Πόσα στιγμιότυπα της δομής `struct file` που σχετίζονται με το `/dev/somedevice` υπάρχουν στον πυρήνα; Έστω ότι στην αρχή καμία διεργασία δεν είχε ανοιχτό το `/dev/somedevice`.*

Ο πυρήνας δημιουργεί N στιγμιότυπα της δομής `struct file` (ένα για κάθε διεργασία). Τελικά, το πλήθος των διεργασιών που εκτελούν το `prog1` είναι N (οι $N/2$ αρχικές συν $N/2$ μετά το `fork()`) και οι διεργασίες που εκτελούν το `prog2` είναι $N/2$ (αυτές που εκτέλεσαν `execve("prog2", ...)`. Τα στιγμιότυπα της δομής `struct file` που σχετίζονται με το `/dev/somedevice` παραμένουν N, καθώς μετά από κάθε `fork()`, η διεργασία-παιδί μοιράζεται τις δομές `struct file` με τη διεργασία-πατέρα.

β. *(25%)* *Θεωρούμε μια εικονική συσκευή χαρακτήρων (`/dev/mult`), η οποία υλοποιεί την πράξη του πολλαπλασιασμού μιας παραμέτρου που περνάει ο χρήστης μέσω κλήσης συστήματος `ioctl()` με τη σταθερά 2. Θέλουμε την υποστήριξη της συσκευής σε περιβάλλον εικονικών μηχανών και αποφασίζουμε την υλοποίηση του οδηγού συσκευής με χρήση του VirtIO split-driver model (frontend/backend). Σας δίνεται μια πρώτη προσπάθεια υλοποίησης του frontend, όπως εκτελείται στον πυρήνα του guest, η οποία εμφανίζει προβλήματα. Επίσης δίνεται το πρόγραμμα χώρου χρήστη `test.c` που χρησιμοποιεί το `/dev/mult`.*

Ισχύουν τα ακόλουθα:

1. *Η λειτουργικότητα της συσκευής υλοποιείται πλήρως από το backend μέσα στο userspace του host.*
2. *Τα δύο μέρη του οδηγού (frontend/backend) επικοινωνούν μέσω μοναδικής virtqueue.*
3. *Το frontend υποστηρίζει μόνο ένα minor number και όχι περισσότερα.*

Ζητούνται τα εξής:

i. *(2%) Πότε κοιμούνται οι διεργασίες; Πότε και ποιος τις ξυπνάει; Πόσες διεργασίες ξυπνάνε κάθε φορά;*

ii. *(2%) Για κάθε μία από τις συναρτήσεις που σας δίνονται, αναφέρετε αν εκτελούνται σε process ή interrupt context.*
iii. *(3%) Τι συμβαίνει στην εικονική μηχανή αφού το μέρος backend γράψει στην virtqueue και ποια συνάρτηση του frontend στον guest το χειρίζεται;*
iv. *(3%) Τι θα συμβεί αν έρθει νέα απάντηση από το backend, χωρίς να έχει απορροφηθεί η προηγούμενη από μία διεργασία;*
v. *(5%) Εκτελούμε ταυτόχρονα `./test 5`, `./test 9`, και η έξοδος κάθε εντολής είναι `18`, `10`, αντίστοιχα. Τι συνέβη; Ποιο είναι το πρόβλημα και πού οφείλεται;*
vi. *(5%) Με ποιον τρόπο θα μπορούσε το μέρος backend να υποδηλώνει ποια απάντηση αντιστοιχεί σε ποια αίτηση; Έστω ότι περνάμε το PID της διεργασίας που εκτέλεσε την `ioctl()` ως αναγνωριστικό της αίτησης, τι πρόβλημα θα δημιουργούσε αυτό; Μπορείτε να προτείνετε καταλληλότερο αναγνωριστικό;* Υπόδειξη: *Χρησιμοποιήστε το πεδίο `tag` της δομής `struct mult_buffer`.*
vii. *(5%) Περιγράψτε μια λύση, χωρίς να δώσετε πλήρη κώδικα, προκειμένου οι διεργασίες να μπορούν με ασφάλεια να εκτελούν ταυτόχρονες προσβάσεις στο `/dev/mult`. Υποθέστε ότι το μέρος backend του οδηγού έχει υλοποιηθεί ήδη και συμπεριφέρεται όπως χρειάζεστε.*

```
1   /* The struct that is being exchanged via the virtqueue. */
2   struct mult_buffer {
3       uint32_t number;
4       uint32_t result;
5
6       uint64_t tag; /* This may prove useful as a request id */
7   };
8
9   /* Global variables for the "mult" virtio device */
10  struct mult_device {
11      struct virtqueue *vq;
12
13      /* The buffer that the host sent us */
14      struct mult_buffer mult_buffer;
15
16      /* Has the above buffer been delivered to any process yet? */
17      bool buff_delivered;
18
19      /* Processes that wait for an answer from the host */
20      wait_queue_head_t wq;
21  };
22
23  struct mult_device *mult_device; /* Assume proper initialization */
24
25  static int mult_chrdev_open(struct inode *inode, struct file *filp)
26  {
27      ...
28      filp->private_data = kzalloc(sizeof(struct mult_buffer), GFP_KERNEL);
29      ...
30  }
31
32  static int mult_chrdev_release(struct inode *inode, struct file *filp)
33  {
34      ...
35      kfree(filp->private_data);
36  }
37
38  bool device_has_data(struct mult_buffer *buf)
```

```
39  {
40      bool ret;
41
42      ret = false;
43
44      /* if another process has taken the buffer return false */
45      if (mult_device->buff_delivered)
46          return false;
47
48      memcpy(buf, mult_device->mult_buffer, sizeof(struct mult_buffer));
49      mult_device->buff_delivered = true;
50
51      return ret;
52  }
53
54  static long mult_chrdev_ioctl(struct file *filp, unsigned int cmd,
55      unsigned long arg)
56  {
57      long ret = 0;
58      struct mult_buffer *buf = (struct mult_buffer *)filp->private_data;
59
60      ret = copy_from_user(buf, (void __user *)arg,
61          sizeof(struct mult_buffer)));
62      if (ret) {
63          ret = -EFAULT;
64          goto out;
65      }
66
67      switch (cmd) {
68      case MULTIPLY:
69          ...
70          /* send buffer and notify host */
71          virtqueue_send_buf(vq, buf);
72          virtqueue_kick(vq);
73          ...
74
75          /* sleep until host sends us the reply */
76          if (!device_has_data(buf)) {
77              if (filp->f_flags & O_NONBLOCK)
78              return -EAGAIN;
79
80              ret = wait_event_interruptible(wq, device_has_data(buf));
81
82              if (ret < 0)
83                  goto out;
84          }
85
86          break;
87
88      default:
89          ret = -EINVAL;
90          goto out;
91      }
92
93      if (copy_to_user((void __user *)arg, buf, sizeof(struct mult_buffer)))
94          ret = -EFAULT;
95
96  out:
97      return ret;
98  }
```

```
 99
100  static void in_intr(struct virtqueue *vq)
101  {
102      struct mult_buffer *buf = mult_device->mult_buffer;
103      struct virtqueue *vq = mult_device->vq;
104      wait_queue_head_t wq = mult_device->wq;
105
106      ...
107      virtqueue_get_buf(vq, buf);
108      mult_device->buff_delivered = false;
109      wake_up_interruptible(wq);
110      ...
111  }
112
```

Ακολουθεί ο κώδικας του `test.c`*:*

```
 1  int main (int argc, char *argv[])
 2  {
 3      int fd;
 4      struct mult_buffer buf;
 5
 6      fd = open("/dev/mult", O_RDWR);
 7      if (fd < 0) {
 8          perror("open"); exit(1);
 9      }
10
11      buf.number = atoi(argv[1]);
12      if (ioctl(fd, MULTIPLY, buf) < 0) {
13          perror("ioctl"); exit(1);
14      }
15
16      printf("%u\n", buf.result);
17      return 0;
18  }
```

i. Στο σενάριο που δίνεται μία διεργασία εκτελεί κλήση συστήματος `ioctl()` με τα σχετικά ορίσματα προκειμένου να εκμεταλλευτεί τη λειτουργικότητα της συσκευής, δηλαδή τον πολλαπλασιασμό. Κατά την κλήση αυτή εκτελείται κώδικας πυρήνα στο εικονικό μηχάνημα, ο οποίος στέλνει μέσω της virtqueue δεδομένα και ενημερώνει τον host (`virtqueue_kick()`). Στη συνέχεια, η διεργασία που εκτελεί την `ioctl()` κοιμάται μέσα στον πυρήνα (`wait_event_interruptible()`, γρ. 80) περιμένοντας η συσκευή να αποκτήσει δεδομένα. Όταν γίνει το σχετικό interrupt, θα κληθεί ο interrupt handler του οδηγού (`in_intr()`, γρ. 100), ο οποίος θα αλλάξει την κατάσταση των διεργασιών που κοιμούνται στην ουρά `wq` σε READY. Μόλις κάθε μία από αυτές τις διεργασίες μπει για να εκτελεστεί στον επεξεργαστή (βάσει των αποφάσεων χρονοδρομολόγησης), θα εκτελέσει τη συνάρτηση `device_has_data()` και αν χρειαστεί (αν η `device_has_data()` επιστρέψει false), θα ξανακοιμηθεί.
ii. Οι συναρτήσεις `mult_chrdev_open()`, `mult_chrdev_release()`, `device_has_data()` και `mult_chrdev_ioctl()` εκτελούνται σε process context, ενώ η συνάρτηση `in_intr()` εκτελείται σε interrupt context.
iii. Η εικονική μηχανή δέχεται (εικονική) διακοπή υλικού, την οποία χειρίζεται ο οδηγός (συνάρτηση `in_intr()`, γρ. 100). Η διακοπή προκαλείται μετά από την εγγραφή των σχετικών δεδομένων από το backend στη virtqueue.

iv. Στην περίπτωση όπου στον χρόνο που μεσολαβεί ανάμεσα στην αποστολή δύο διαδοχικών πακέτων από τον host δεν έχει επιλεγεί προς εκτέλεση μία από τις διεργασίες που περιμένουν αυτές τις απαντήσεις (ώστε να απορροφηθεί το πρώτο πακέτο) τότε το πρώτο πακέτο θα χαθεί (το νέο θα το πανωγράψει) με αποτέλεσμα κάποια διεργασία να περιμένει για πάντα μία απάντηση η οποία δε θα έρθει ποτέ. Μια λύση για αυτό το πρόβλημα θα ήταν η αντικατάσταση του πεδίου `mult_buffer` της δομής `mult_device` από έναν κυκλικό απομονωτή με δομές `mult_buffer` ο οποίος θα κρατάει τις πιο πρόσφατες απαντήσεις από τον host.

v. Αυτό που συνέβη είναι ότι η μία διεργασία πήρε την απάντηση που αντιστοιχούσε στην άλλη. Το πρόβλημα είναι ότι ο οδηγός έχει γραφτεί με τέτοιο τρόπο ώστε να μην είναι εφικτός ο διαχωρισμός των πακέτων που στέλνει ο host μέσω της virtqueue.

vi. Για το διαχωρισμό των απαντήσεων που λαμβάνει ο guest από τον host θα έπρεπε να χρησιμοποιήσουμε κάποιο είδος αναγνωριστικού το οποίο πρέπει να είναι μοναδικό για κάθε διεργασία, μέσα στη δομή `mult_buffer`. Αν χρησιμοποιηθεί για το σκοπό αυτό το PID της κάθε διεργασίας τότε θα είχαμε πρόβλημα με τη χρήση των νημάτων σε χώρο χρήστη (π.χ. pthreads), όπου πολλά νήματα μοιράζονται το ίδιο PID. Μια λύση θα ήταν να αριθμούμε τις κλήσεις `ioctl()` και να κρατάμε αυτό τον αριθμό ως αναγνωριστικό για να διακρίνουμε διαφορετικές κλήσεις `ioctl()`.

vii. Η λύση που προτάθηκε στο προηγούμενο ερώτημα υλοποιείται με τις παρακάτω αλλαγές στον κώδικα που δίνεται για τον οδηγό:

```
1   /* The struct that is being exchanged via the virtqueue. */
2   struct mult_buffer {
3       ...
4       uint64_t tag; /* This may prove useful as a request id */
5   };
6
7   /* Global variables for the "mult" virtio device */
8   struct mult_device {
9       ...
10  --> uint64_t next_tag; /* Initialized to 0 */
11  };
12
13  struct mult_device *mult_device; /* Assume proper initialization */
14
15  bool device_has_data(struct mult_buffer *buf)
16  {
17      bool ret;
18
19      ret = false;
20
21      /* if another process has taken the buffer return false */
22      if (mult_device->buff_delivered)
23          return false;
24
25  --> /* Check if the buffer is for the calling process. */
26  --> if (buf->tag != mult_device->mult_buffer->tag)
27  -->     return false;
28
29      ...
30  }
31
32  static long mult_chrdev_ioctl(struct file *filp, unsigned int cmd,
33      unsigned long arg)
34  {
```

```
35      long ret = 0;
36      struct mult_buffer *buf = (struct mult_buffer *)filp->private_data;
37
38      ret = copy_from_user(buf, (void __user *)arg,
39          sizeof(struct mult_buffer)));
40      if (ret) {
41          ret = -EFAULT;
42          goto out;
43      }
44
45  --> /* Get a tag number */
46  --> /* Should have a lock here */
47  --> buf->tag = mult_device->next_tag++;
48
49      switch (cmd) {
50      case MULTIPLY:
51          ...
52      default:
53          ...
54      }
55
56      ...
57  }
58
```

Θέμα 3 (30%)

α. (8%) *Απαντήστε* περιληπτικά *στα εξής:*

i. *Τι είναι ένα UNIX domain socket; Ποια οικογένεια διευθύνσεων χρησιμοποιεί και τι μορφή έχουν οι διευθύνσεις σε αυτή την οικογένεια;*
ii. *Πώς συγκρίνεται ένα UNIX domain socket με ένα Internet (IPv4) socket;*
iii. *Έστω πρόγραμμα-πελάτης, το οποίο επιχειρεί να συνδεθεί μέσω ενός (α) UNIX domain, (β) TCP/IP socket σε μια διεύθυνση. Ποια κλήση συστήματος θα χρησιμοποιήσει στη μία και στην άλλη περίπτωση; Πώς αλλάζει η δήλωση της κλήσης συστήματος (πλήθος και τύπος ορισμάτων) για τα δύο sockets, και πώς διακρίνονται;*

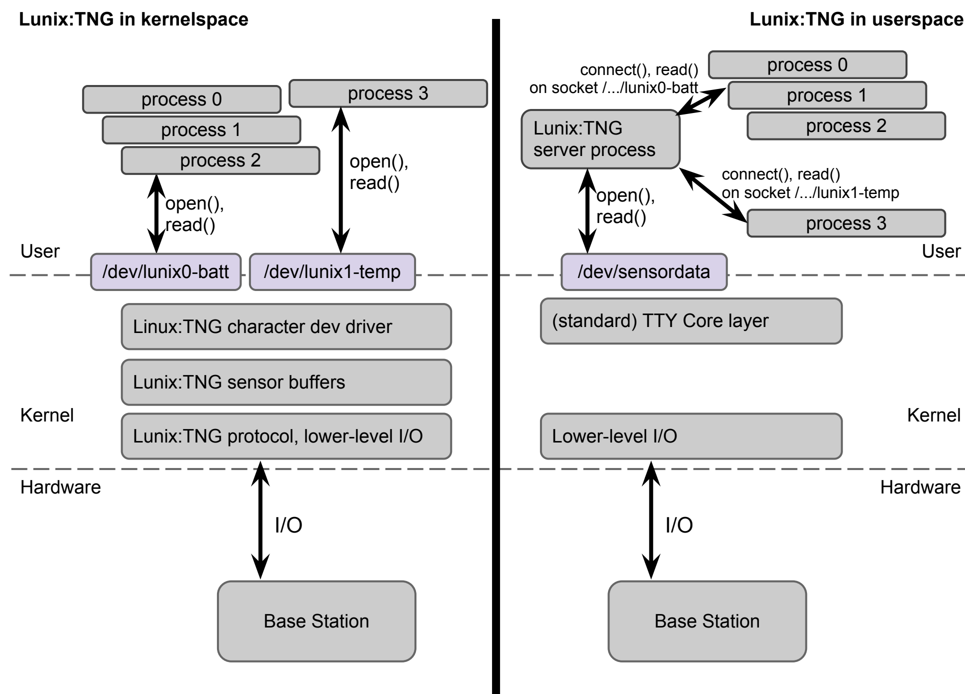
i. Είναι ένας μηχανισμός διαδιεργασιακής επικοινωνίας ανάμεσα σε διεργασίες που εκτελούνται στο ίδιο σύστημα. Από την πλευρά των διεργασιών ένα UNIX domain socket εμφανίζεται ως περιγραφητής ανοιχτού αρχείου. Χρησιμοποιεί τον χώρο ονομάτων `PF_UNIX`, συνώνυμο του `PF_LOCAL` και τον αντίστοιχο χώρο διευθύνσεων `AF_UNIX`/`AF_LOCAL`. Οι διευθύνσεις είναι ονόματα αρχείων (pathnames) στο τοπικό σύστημα αρχείων.
ii. Είναι και τα δύο sockets και χρησιμοποιούνται μέσω του BSD Sockets API. Τα Unix domain sockets μπορούν να χρησιμοποιηθούν μόνο για επικοινωνία ανάμεσα σε διεργασίες στον ίδιο υπολογιστή, ενώ τα IPv4 sockets για τη σύνδεση διεργασιών σε διαφορετικά μηχανήματα μέσω δικτύου IP.
iii. Και στις δύο περιπτώσεις θα χρησιμοποιήσει την κλήση συστήματος `connect()`. Η δήλωση της κλήσης συστήματος είναι ανεξάρτητη του είδους του socket και δεν αλλάζει, παίρνει πάντα όρισμα `struct sockaddr *addr`. Το πρώτο πεδίο του `struct sockaddr`, το `sa_family`, καθορίζει αν ο δείκτης ερμηνεύεται ως `struct sockaddr_in *` (IPv4 socket), `sockaddr_un *` (UNIX domain socket), ή άλλο.

β. ***(22%)*** *Θεωρήστε σενάριο όπου τιμές διακριτών μετρήσεων από πολλούς αισθητήρες, (`sensor0-temp`, `sensor0-batt`, . . ., `sensorN-temp`, `sensorN-batt`) είναι προσβάσιμο μέσω μοναδικού ρεύματος δεδομένων από συσκευή χαρακτήρων `/dev/sensordata`. Επιθυμούμε να μπορούμε να έχουμε απομονωμένη, ελεγχόμενη, ταυτόχρονη πρόσβαση από πολλές διεργασίες στα δεδομένα των αισθητήρων, σε σενάριο παρόμοιο με αυτό του Lunix:TNG, χωρίς την προσθήκη νέου οδηγού συσκευής στον πυρήνα. Σκιαγραφήστε υλοποίηση που το επιτυγχάνει εξ ολοκλήρου στο χώρο χρήστη, βασιζόμενη σε κεντρική διεργασία που εκτελείται με δικαίωμα root. Στα επόμενα θα συγκρίνετε την προσέγγισή σας με την προσέγγιση του Lunix:TNG, η οποία βασίζεται σε οδηγό συσκευής που φιλτράρει τα εισερχόμενα δεδομένα και τα παρουσιάζει μέσω χωριστών ειδικών αρχείων. Απαντήστε περιληπτικά στα εξής:*

- *(5%) Ποιο μηχανισμό επικοινωνίας επιλέγετε; Με ποιον τρόπο μπορούν οι διεργασίες να έχουν πρόσβαση στις τιμές που τις αφορούν;*
- *(5%) Σχεδιάστε δύο διαγράμματα, ένα για κάθε σχεδίαση, στο οποίο να φαίνεται το υλικό, οι χώροι πυρήνα και χρήστη, και το πού βρίσκεται ο υπό εκτέλεση κώδικας κάθε φορά.*
- *(5%) Με ποιον μηχανισμό εξασφαλίζεται η απομονωμένη πρόσβαση στα δεδομένα και πώς επιβάλλονται διαφορετικά δικαιώματα πρόσβασης στη μία και στην άλλη περίπτωση; Π.χ. πώς εξασφαλίζεται ότι μόνο ο χρήστης `user1` θα έχει δικαιώματα στις μετρήσεις του αισθητήρα `sensor0`;*
- *(7%) Πώς συγκρίνεται η σχεδίασή σας με τη σχεδίαση του Lunix:TNG; Αναφέρετε ένα πλεονέκτημα κι ένα μειονέκτημα της κάθε προσέγγισης.*

Γενικά: η κεντρική διεργασία χώρου χρήστη που εκτελείται με δικαίωμα root θα είναι η μόνη που θα έχει πρόσβαση στο μοναδικό ρεύμα δεδομένων `/dev/sensordata`. Θα φροντίζει να διαχωρίζει τα δεδομένα, και να επιβάλλει ελεγχόμενη πρόσβαση σε αυτά. Κάθε άλλη διεργασία θα πρέπει να μιλάει με αυτή, μέσω κατάλληλου μηχανισμού διαδιεργασιακής επικοινωνίας.

- Επιλέγουμε το μηχανισμό των UNIX domain sockets. Είναι η καταλληλότερη επιλογή όταν πολλές διεργασίες στο UNIX επιθυμούν να μιλήσουν με κεντρική διεργασία-εξυπηρετητή. Παραδείγματα χρήσης στον πραγματικό κόσμο: Στο σύστημα X Window, όταν και μια γραφική εφαρμογή (π.χ. ο firefox), και ο X11 Server, που χειρίζεται οθόνη-/πληκτρολόγιο/ποντίκι τρέχουν στο ίδιο σύστημα, επικοινωνούν μέσω UNIX domain sockets, βλ. `http://www.x.org/archive/X11R6.8.0/doc/Xorg.1.html`. Ομοίως, πελάτες της βάσης MySQL μπορούν να συνδεθούν τοπικά στον MySQL server και να εκτελέσουν queries μέσω UNIX domain socket, βλ. `https://dev.mysql.com/doc/refman/5.0/en/connecting.html`.
  Τρόπος πρόσβασης: Κάθε διεργασία εκτελεί `socket(PF_UNIX, ...)`, `connect(...)` σε κατάλληλο socket ανάλογα με τον αισθητήρα και τη μέτρηση στην οποία θέλει να έχει πρόσβαση, και `read()` για την επικοινωνία.
- Φαίνεται σε χωριστό σχήμα.
- Υπάρχουν πολλοί διαφορετικοί τρόποι. Θα μπορούσαμε να χρησιμοποιήσουμε μοναδικό socket, οπότε το βάρος της πιστοποίησης των εισερχόμενων διεργασιών θα έπεφτε στην κεντρική διεργασία, π.χ. με υλοποίηση μηχανισμού username/password από το μηδέν. Επιλέγουμε να επαναχρησιμοποιήσουμε το μηχανισμό ελεγχόμενης πρόσβασης του UNIX: χρησιμοποιούμε διακριτά UNIX domain sockets, ένα για κάθε ζεύγος αισθητήρα-μέτρησης, με διακριτή διεύθυνση (pathname) στο σύστημα αρχείων, π.χ. `/tmp/lunix-tng-sockets/{lunix0-batt, lunix0-temp, ...}`, και είτε αναθέτουμε διαφορετικά δικαιώματα πρόσβασης (με `chown`, `chmod`) στο καθένα, είτε στους (διακριτούς)



καταλόγους όπου βρίσκονται. Έτσι μπορούμε να διακρίνουμε τις διεργασίες με βάση το user ή το group id τους. Για να εξασφαλίσουμε ότι μόνο ο χρήστης `user1` έχει δικαίωμα στις μετρήσεις του αισθητήρα `sensor0`, ένας τρόπος είναι να τον κάνουμε κάτοχο όλων των σχετικών sockets, με `chown`, και να του δώσουμε αποκλειστική πρόσβαση σε αυτά, π.χ. με `chmod 0600`. Ομοίως, αν χρησιμοποιούμε την υλοποίηση του Lunix:TNG στον πυρήνα μπορούμε να αλλάξουμε τα δικαιώματα πρόσβασης στα ειδικά αρχεία `/dev/lunix*`.

- Στη νέα σχεδίαση το φιλτράρισμα των εισερχόμενων δεδομένων υλοποιείται στο χώρο χρήστη. Πλεονεκτήματα: ευκολότερη υλοποίηση, δυνατότητα χρήσης userspace debuggers (π.χ. μπορώ να τρέξω `strace` ή `gdb` πάνω στην κεντρική διεργασία), δυνατότητα εκτέλεσης υπολογισμών κινητής υποδιαστολής, χρήσης βιβλιοθηκών. Πιθανό προγραμματιστικό σφάλμα δεν επηρεάζει τη σταθερότητα του συνολικού συστήματος. Μειονεκτήματα: Απαίτηση για χρήση UNIX domain sockets, ενώ πριν είχε πρόσβαση οποιοδήποτε πρόγραμμα μπορούσε να εκτελέσει `open()`/`read()`, π.χ. `cat`, μεγαλύτερη υπολογιστική επιβάρυνση (overhead) ειδικά αν ο ρυθμός των εισερχόμενων δεδομένων είναι πολύ μεγάλος, ανάγκη για συνεχή αντιγραφή δεδομένων ανάμεσα στους χώρους πυρήνα/χρήστη.