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 2013-2014

Κανονική Εξέταση – Λύσεις

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

Θέμα 1 (40%)

Συμμετέχετε σε ομάδα που σχεδιάζει και υλοποιεί το νέο μηχανισμό INTERCOM για διαδιεργασιακή επικοινωνία στο Linux. Ο μηχανισμός προσφέρει μοναδικό μονόδρομο κανάλι δεδομένων, για το οποίο ισχύουν τα εξής:

1. *Το κανάλι είναι διαθέσιμο μέσω ειδικού αρχείου συσκευής χαρακτήρων `/dev/intercom`.*
2. *Το κανάλι έχει δύο άκρα, το άκρο ανάγνωσης (**R**) και το άκρο εγγραφής (**W**).*
3. *Μια διεργασία αποκτά πρόσβαση στο **R** ανοίγοντας το ειδικό αρχείο μόνο για ανάγνωση, πρόσβαση στο **W** ανοίγοντας το ειδικό αρχείο μόνο για εγγραφή, και πρόσβαση και στα δύο ανοίγοντας το ειδικό αρχείο για ανάγνωση και εγγραφή.*
4. *Δεδομένα που γράφονται στο **W**, μέσω της κλήσης συστήματος `write()`, παραμένουν στο κανάλι μέχρι να διαβαστούν από το **R**, μέσω της κλήσης συστήματος `read()`. Τα δεδομένα επιστρέφονται με τη σειρά που γράφτηκαν, το κανάλι είναι ένα ρεύμα χαρακτήρων.*
5. *Το άνοιγμα του ειδικού αρχείου για εγγραφή μπλοκάρει μέχρι να υπάρξουν αναγνώστες, και αντίστροφα.*
6. *Κλήσεις `read()` μπλοκάρουν, αν το κανάλι δεν περιέχει δεδομένα.*
7. *Κλήσεις `read()` επιστρέφουν `0` (EOF), αν δεν υπάρχουν πλέον εγγραφείς.*
8. *Κλήσεις `write()` μπλοκάρουν, αν το κανάλι δεν μπορεί να δεχτεί δεδομένα.*

```
1  struct intercom_dev_struct {
2      uint128_t wcnt, rcnt; /* TOTAL number of bytes read/written, from/to the
3                               circular buffer. They are initialized to zero,
4                               and will never wrap. */
5  #define CIRC_BUF_SIZE (4096 * 1024)
6      char circ_buffer[CIRC_BUF_SIZE];
7      ...
8  } intercom;
9
10 int intercom_open(struct inode *inode, struct file *filp)
11 {
12         int m = filp->f_mode;
13         ...
14         filp->private_data = &intercom;
15 }
16
17 static ssize_t intercom_read(struct file *filp, char __user *usrbuf,
18     size_t cnt, loff_t *f_pos)
19 {
20     struct intercom_device *dev = filp->private_data;
21     ...
22     return ret;
23 }
```

Σας δίνεται ο σκελετός του οδηγού που υλοποιεί το `/dev/intercom`. Κάντε όποιες επεμβάσεις επιθυμείτε, αρκεί να τις περιγράψετε με ακρίβεια.

Ζητούνται εξής:

i. *(3%) Τι χρειάζεται να κάνουν δύο διεργασίες που εκτελούνται ήδη στο σύστημα για να μπορέσουν να επικοινωνήσουν; Τι σχέση πρέπει να έχουν μεταξύ τους; Πώς γίνεται να δοθεί αποκλειστική πρόσβαση στο κανάλι επικοινωνίας σε συγκεκριμένο χρήστη;*
ii. *(2%) Δύο διεργασίες μεταφέρουν 1GB δεδομένων πάνω από το κανάλι επικοινωνίας. Ποιες είναι οι απαιτήσεις σε μνήμη RAM ή/και χώρο στο δίσκο από το σύστημα;*
iii. *(3%) Με ποιον υπάρχοντα μηχανισμό διαδιεργασιακής επικοινωνίας του Linux μοιάζει ο μηχανισμός INTERCOM; Πού διαφέρει ο INTERCOM από τον υπάρχοντα μηχανισμό του Linux;* Υπόδειξη: *Μια διεργασία επιχειρεί να γράψει 32 bytes. Ποιες είναι οι δυνατές τιμές επιστροφής της `write()` στη μία και στην άλλη περίπτωση;*
iv. *(7%) Υλοποιήστε την `intercom_open()`. Πώς καλύπτονται οι προδιαγραφές 3, 5;*
v. *(8%) Υλοποιήστε τη `read()`. Περιγράψτε* πολύ *συνοπτικά τη λειτουργία της `write()`, αντίστοιχα με τη `read()` σας, χωρίς να δώσετε κώδικα ή ψευδοκώδικα. Πώς καλύπτει ο οδηγός σας τις προδιαγραφές 4, 6, 7;*
vi. *(2%) Ποιες οντότητες επιχειρούν ταυτόχρονη πρόσβαση σε δομές του οδηγού και πώς εξασφαλίζεται η ορθή λειτουργία του;*

Σημείωση: *Θεωρήστε ότι η δομή `struct file` έχει πεδίο `f_mode`, το οποίο παίρνει τιμές που είναι συνδυασμός των bits `FMODE_READ`, `FMODE_WRITE`.*

i. Χρειάζονται να ανοίξουν το `/dev/intercom`, η μία για ανάγνωση κι η άλλη για εγγραφή. Δεν χρειάζεται να έχουν κάποια ειδική σχέση. Ο διαχειριστής μπορεί να αλλάξει τον κάτοχο του αρχείου `/dev/intercom` ώστε να είναι ο συγκεκριμένος χρήστης

και να επιτρέψει πρόσβαση μόνο στον κάτοχο: (`chown user /dev/intercom`, `chmod 600 /dev/intercom`).

ii. Ο οδηγός χρειάζεται τόση RAM (μνήμη πυρήνα) όσα δεδομένα είναι κάθε φορά προσωρινά αποθηκευμένα στο κανάλι, δηλαδή `CIRC_BUF_SIZE`, 4MB. Το πόσα GB δεδομένων μπορούν να μεταφέρονται συνολικά είναι ανεξάρτητο από το μέγεθος της RAM του συστήματος ή από το μέγεθος του δίσκου του, αφού τα δεδομένα δεν περνάνε καν από το δίσκο.

iii. Με τα named pipes. Διαφέρουν ως προς τις εγγυήσεις ατομικότητας: τα pipes και τα named pipes εγγυώνται ότι εγγραφές μεγέθους μικρότερου του `PIPE_BUF` γίνονται ατομικά, σε αντίθεση με τον μηχανισμό INTERCOM. Αλλιώς: Ένα `write()` των 32 bytes στο `/dev/intercom` μπορεί να επιστρέψει από `1` έως `32`, ενώ το named pipe εγγυάται ότι το `write()` θα μπλοκάρει μέχρι να μπορεί να επιστρέψει με τιμή επιστροφής `32`. Επίσης, στο INTERCOM δεν υπάρχει πρόβλεψη για αποστολή `SIGPIPE` κατά την εκτέλεση `write()` όταν δεν υπάρχουν πλέον αναγνώστες, και επιστροφή τιμής `EPIPE` αν η διεργασία χειρίζεται το `SIGPIPE`.

iv. Η υλοποίησή της `open()` ακολουθεί. Η προδιαγραφή 3 καλύπτεται από τον ίδιο τον πυρήνα, ο οποίος δεν επιτρέπει την εκτέλεση της `write()` σε αρχείο ανοιχτό με `O_RDONLY` ή το αντίστροφο [1] [2]. Η `read()` ανακτά δεδομένα από το **R**, η `write()` αποθηκεύει στο **W**. Η προδιαγραφή 5 καλύπτεται με μετρητές `readers`, `writers` (ενεργοί αναγνώστες και εγγραφείς), `open_r_count`, `open_w_count` (αριθμός κλήσεων `open()` για ανάγνωση και εγγραφή αντίστοιχα που έχουν πραγματοποιηθεί ως τώρα) και ουρά αναμονής `openwq`, στην οποία περιμένουν οι διεργασίες που πρέπει να κοιμηθούν γιατί θέλουν να ανοίξουν το αρχείο για εγγραφή και δεν υπάρχουν αναγνώστες ή το αντίστροφο. Για πληρότητα δίνουμε και ενδεικτική υλοποίηση της `release()`, η οποία μειώνει τις τιμές των `readers`, `writers`.

Λεπτό σημείο: Η πολύ απλούστερη υλοποίηση χωρίς `open_{r,w}_count`, όπου η `open()` αποφασίζει με βάση την τιμή `readers` ή `writers` αν μπορεί να συνεχίσει έχει race: Είναι δυνατό ενώ περιμένουμε έναν εγγραφέα (`writers == 0`), μια διεργασία να ανοίξει το ειδικό αρχείο, να γράψει σε αυτό και ναι το κλείσει, πριν χρονοδρομολογηθεί η `open()` που περιμένει. Όταν τελικά τρέξει, θα δει `writers == 0` όπως πριν και θα συνεχίσει να μπλοκάρει, ενώ ήρθε εγγραφέας και μάλιστα υπάρχουν δεδομένα που περιμένουν.

Λεπτό σημείο: Αν η διεργασία δεχτεί σήμα ενώ είναι σε αναμονή στην `open()`, πρέπει να ξανακλειδώσουμε και να καθαρίσουμε τη δομή πριν επιστρέψουμε. Αν στο ενδιάμεσο έχει εκπληρωθεί η συνθήκη που περιμέναμε, επιστρέφουμε επιτυχώς, όχι `-ERESTARTSYS`.

```
1   static int intercom_open(struct inode *inode, struct file *filp)
2   {
3       int sigpending;
4       int r = (filp->f_mode & FMODE_READ);
5       int w = (filp->f_mode & FMODE_WRITE);
6       int blockval, *blockp;
7       struct intercom_dev_struct *dev = &intercom;
8
9       nonseekable_open(inode, filp);
10      filp->private_data = dev;
11
12      down(&dev->lock);
13      if (r) {
```

[1] `http://lxr.free-electrons.com/source/fs/read_write.c#L419`

[2] `http://lxr.free-electrons.com/source/fs/read_write.c#L520`

```
14          ++dev->readers;
15          ++dev->open_r_count;
16      }
17      if (w) {
18          ++dev->writers;
19          ++dev->open_w_count;
20      }
21      sigpending = 0;
22      /* Should we wait for a reader, or a writer? */
23      blockp = NULL;
24      if (!dev->readers) {
25          blockp = &dev->open_r_count;
26          blockval = dev->open_r_count;
27      }
28      if (!dev->writers) {
29          blockp = &dev->open_w_count;
30          blockval = dev->open_w_count;
31      }
32      /* Block until a change in the count we care about */
33      while (blockp && blockval == *blockp) {
34          /* If woken up due to a signal, clean up before returning */
35          if (sigpending) {
36              if (r) --dev->readers;
37              if (w) --dev->writers;
38              up(&dev->lock);
39              return -ERESTARTSYS;
40          }
41          up(&dev->lock);
42          if (wait_event_interruptible(dev->openwq, blockval != *blockp))
43              sigpending = 1;
44          down(&dev->lock);
45      }
46      up(&dev->lock);
47      wake_up_interruptible(&dev->openwq);
48      return 0;
49  }
50
51  static int intercom_release(struct inode *inode, struct file *filp)
52  {
53      struct intercom_dev_struct *dev = filp->private_data;
54
55      down(&dev->lock);
56      if (filp->f_mode & FMODE_READ)
57          --dev->readers;
58      if (filp->f_mode & FMODE_WRITE) {
59          --dev->writers;
60          wake_up_interruptible(&dev->rwq); /* EOF? */
61      }
62      up(&dev->lock);
63
64      return 0;
65  }
```

v. Η υλοποίησή της `read()` ακολουθεί. Η `write()` αντιγράφει δεδομένα από το χώρο χρήστη στον κυκλικό buffer, η `read()` βγάζει δεδομένα από τον κυκλικό buffer και τα αντιγράφει στο χώρο χρήστη. Αν ο κυκλικός buffer είναι γεμάτος, η `write()` μπλοκάρει σε ουρά `wwq`. Ομοίως, αν ο κυκλικός buffer είναι άδειος, αν υπάρχουν εγγραφείς η `read()` μπλοκάρει σε ουρά `rwq`, αν δεν υπάρχουν, επιστρέφει `0`. Τα παραπάνω καλύπτουν τις προδιαγραφές 4, 6, 7. Η υλοποίηση της `release()` ξυπνάει την `rwq` ώστε να

επιστρέφεται EOF σε διεργασία που ήδη περιμένει για δεδομένα.

vi. Μόνο διεργασίες επιχειρούν πρόσβαση στην κοινή δομή `intercom`, τα πεδία και τον κυκλικό της buffer. Έτσι, η δομή αρκεί να προστατεύεται από σημαφόρο `lock`.

```
1   #define BYTES_IN_BUFFER(dev) ((dev)->wcnt - ((dev)->rcnt))
2
3   static ssize_t intercom_read(struct file *filp, char __user *usrbuf,
4       size_t cnt, loff_t *f_pos)
5   {
6       ssize_t ret;
7       int i, oldrcnt;
8       char *buf;
9       struct intercom_dev_struct *dev = filp->private_data;
10
11      if (cnt == 0)
12          return 0;
13  #define MAX_CNT_PER_READ CIRC_BUF_SIZE /* A sane value */
14      if (cnt > MAX_CNT_PER_READ)
15          cnt = MAX_CNT_PER_READ;
16
17      down(&dev->lock);
18      while (BYTES_IN_BUFFER(dev) == 0) {
19          if (dev->writers == 0) {
20              ret = 0; /* EOF */
21              goto out;
22          }
23          up(&dev->lock);
24          if (wait_event_interruptible(dev->rwq,
25                                       (BYTES_IN_BUFFER(dev) ||
26                                        dev->writers == 0)))
27              return -ERESTARTSYS;
28          down(&dev->lock);
29      }
30      if (cnt > BYTES_IN_BUFFER(dev))
31          cnt = BYTES_IN_BUFFER(dev);
32
33      /*
34       * Take care of wrapping.
35       * Transfer through intermediate buffer for simplicity.
36       */
37      buf = kmalloc(cnt, GFP_KERNEL);
38      if (!buf) {
39          ret = -ENOMEM;
40          goto out;
41      }
42      oldrcnt = dev->rcnt;
43      for (i = 0; i < cnt; i++)
44          buf[i] = dev->circ_buffer[(dev->rcnt++) % CIRC_BUF_SIZE];
45      if (copy_to_user(usrbuf, buf, cnt)) {
46          dev->rcnt = oldrcnt;
47          ret = -EFAULT;
48          goto out_with_buf;
49      }
50      ret = cnt;
51
52  out_with_buf:
53      kfree(buf);
```

```
54  out:
55      up(&dev->lock);
56      if (ret > 0)
57          wake_up_interruptible(&dev->wwq);
58      return ret;
59  }
```

```
1   #define TIMERHZ   10       /* timer() gets called 10 times/sec */
2   #define TOKENMAX  2097152
3   #define RATE      10485760 /* 10 MB/s */
4
5   struct intercom_dev_struct {
6       ...
7       int tokens;     /* Initialized to TOKENMAX */
8   };
9
10  static void timer(void)
11  {
12          struct intercom_dev_struct *dev = &intercom;
13
14          ...
15          dev->tokens += RATE / TIMERHZ;
16          if (dev->tokens >= TOKENMAX)
17              dev->tokens = TOKENMAX;
18          ...
19  }
```

Σε δεύτερη φάση, σας ζητείται η επέκταση του INTERCOM ώστε να έχει τη δυνατότητα ελέγχου *του ρυθμού με τον οποίο περνάνε δεδομένα από μέσα του σε μόνιμη κατάσταση (rate limiting) ώστε να έχει δεδομένη τιμή, π.χ.* `RATE` *= 10MB/s. Επιθυμούμε να κάνουμε αλλαγές στο μέρος* `read()` *του οδηγού, έτσι ώστε να μην επιτρέπει την ανάκτηση δεδομένων από το κανάλι με ρυθμό μεγαλύτερο του* `RATE`*.*

Σας δίνεται η επέκταση στο σκελετό του οδηγού που ακολουθεί. Διακοπές χρονιστή προκαλούν την εκτέλεση της συνάρτησης `timer()` *με συχνότητα* `TIMERHZ` *φορές/sec.*

i. *(5%) Προτείνετε τρόπο με τον οποίο τα διάφορα μέρη του οδηγού μπορούν να συνεργαστούν ώστε να επιτευχθεί ο ζητούμενος έλεγχος του ρυθμού μεταφοράς. Περιγράψτε την τροποποιημένη συμπεριφορά της* `read()`*.*
ii. *(4%) Υλοποιήστε τη συνάρτηση* `read()`*.*
iii. *(3%) Υλοποιήστε τη συνάρτηση* `timer()`*. Ποιες οντότητες επιχειρούν ταυτόχρονη πρόσβαση σε δομές του οδηγού και πώς εξασφαλίζεται η ορθή λειτουργία του;*
iv. *(3%) Αρκεί η αλλαγή του κώδικα μόνο της* `read()` *για την επίτευξη του στόχου; Πώς θα φαίνεται το κανάλι από την πλευρά μιας διεργασίας που εκτελεί* `write()`*;*

i. Θα υλοποιήσουμε αλγόριθμο περιορισμού του ρυθμού μεταφοράς δεδομένων από τη `read()` με βάση τα tokens τα οποία περιοδικά τοποθετεί στη μεταβλητή `tokencnt` η `timer()` κάθε φορά που τρέχει. Η `read()` αφαιρεί τόσα tokens όσα bytes αντιγράφει στο χώρο χρήστη, αν τα tokens επαρκούν. Αλλιώς, μεταφέρει τόσα bytes όσα tokens

υπάρχουν. Αν δεν υπάρχουν πλέον tokens, μπλοκάρει στην ουρά `rwq`. Η `timer()` ξυπνά την ουρά `rwq` ώστε οι διεργασίες που περιμένουν να έχουν την ευκαιρία να αποτιμήσουν εκ νέου την κατάσταση. Η τιμή `TOKENMAX` ρυθμίζει τη μέγιστη ριπή δεδομένων που μπορεί να συμβεί σε μεταβατική κατάσταση, αν το κανάλι έχει μείνει αρκετό καιρό σε αδράνεια και έχει συσσωρεύσει tokens. Η ριπή σταματά όταν εξαντληθούν τα tokens.

ii. Η υλοποίηση της `read()` ακολουθεί. Οι διαφορές είναι στον περιορισμό του `cnt` με βάση το τρέχον πλήθος tokens, στη συνθήκη για την οποία βάζει την καλούσα διεργασία σε κατάσταση αναμονής και στη μείωση του αριθμού των tokens πριν από την επιστροφή δεδομένων στο χρήστη.

```
1   static ssize_t intercom_read(struct file *filp, char __user *usrbuf,
2       size_t cnt, loff_t *f_pos)
3   {
4       int i, oldrcnt;
5       ssize_t ret;
6       char *buf;
7       struct intercom_dev_struct *dev = filp->private_data;
8       unsigned long flags;
9
10      if (cnt == 0)
11          return 0;
12  #define MAX_CNT_PER_READ CIRC_BUF_SIZE /* A sane value */
13      if (cnt > MAX_CNT_PER_READ)
14          cnt = MAX_CNT_PER_READ;
15
16      down(&dev->lock);
17      while (BYTES_IN_BUFFER(dev) == 0 || dev->tokens == 0) {
18          if (BYTES_IN_BUFFER(dev) == 0 && dev->writers == 0) {
19              ret = 0; /* EOF */
20              goto out;
21          }
22          up(&dev->lock);
23          if (wait_event_interruptible(
24                  dev->rwq,
25                  ((BYTES_IN_BUFFER(dev) && dev->tokens) ||
26                    dev->writers == 0)))
27              return -ERESTARTSYS;
28          down(&dev->lock);
29      }
30      if (cnt > BYTES_IN_BUFFER(dev))
31          cnt = BYTES_IN_BUFFER(dev);
32
33      spin_lock_irqsave(&dev->tokenlck, flags);
34      if (cnt > dev->tokens)
35          cnt = dev->tokens;
36      spin_unlock_irqrestore(&dev->tokenlck, flags);
37      /*
38       * Take care of wrapping.
39       * Transfer through intermediate buffer for simplicity.
40       */
41      buf = kmalloc(cnt, GFP_KERNEL);
42      if (!buf) {
43          ret = -ENOMEM;
44          goto out;
45      }
46      oldrcnt = dev->rcnt;
47      for (i = 0; i < cnt; i++)
```

```
48          buf[i] = dev->circ_buffer[(dev->rcnt++) % CIRC_BUF_SIZE];
49      if (copy_to_user(usrbuf, buf, cnt)) {
50          dev->rcnt = oldrcnt;
51          ret = -EFAULT;
52          goto out_with_buf;
53      }
54      ret = cnt;
55
56  out_with_buf:
57      kfree(buf);
58  out:
59      if (ret > 0) {
60          spin_lock_irqsave(&dev->tokenlck, flags);
61          dev->tokens -= cnt;
62          spin_unlock_irqrestore(&dev->tokenlck, flags);
63      }
64      up(&dev->lock);
65      if (ret > 0)
66          wake_up_interruptible(&dev->wwq);
67      return ret;
68  }
```

iii. Η υλοποίηση της `timer()` ακολουθεί. Κάθε φορά που τρέχει, προσθέτει τόσα tokens όσα χρειάζεται για να διατηρηθεί ο ζητούμενος ρυθμός `RATE` και ξυπνάει την ουρά `rwq`. Η `timer()` εκτελείται σε interrupt context, αλλά ακουμπά μόνο το πεδίο `tokens`. Αντί να αλλάξουμε το κλείδωμα όλης της δομής, προσθέτουμε το spinlock `tokenlck`, το οποίο παίρνει και η `read()` κάθε φορά που χρειάζεται να αφαιρέσει tokens κατά τη μεταφορά δεδομένων προς το χώρο χρήστη.

```
1   static void timer(void)
2   {
3       struct intercom_dev_struct *dev = &intercom;
4
5       spin_lock(&dev->tokenlck);
6       dev->tokens += rate / TIMERHZ;
7       if (dev->tokens > TOKENMAX)
8           dev->tokens = TOKENMAX;
9       spin_unlock(&dev->tokenlck);
10      wake_up_interruptible(&dev->rwq);
11  }
```

iv. Ναι, αρκεί. Για να περιορίσουμε το ρυθμό με τον οποίο περνάνε δεδομένα μέσα από το κανάλι, αρκεί να περιορίσουμε το ρυθμό με τον οποίο *βγαίνουν* δεδομένα από αυτό. Μια διεργασία που κάνει `write()` θα δει ότι οι πρώτες-πρώτες κλήσεις στη `write( )` επιστρέφουν γρήγορα, γιατί ακόμη γεμίζουν τον κυκλικό buffer, ενώ οι επόμενες μπλοκάρουν, καθώς ο ρυθμός με τον οποίο αδειάζει ο buffer είναι πολύ μικρότερος από τον ρυθμό με τον οποίο γεμίζει, οπότε θα τον βλέπουν συνεχώς γεμάτο. Στη μόνιμη κατάσταση, ο ρυθμός με τον οποίο οι `write()` μπορούν να βάζουν δεδομένα στον buffer είναι ίσος με τον ρυθμό με τον οποίο οι κλήσεις `read()` τον αδειάζουν.

Σχετικοί σύνδεσμοι:
Ο μηχανισμός που περιγράφεται είναι πολύ κοντά στα named pipes του UNIX. Η επέκτασή τους ώστε να υποστηρίζουν έλεγχο του ρυθμού μεταφοράς βασίζεται στον αλγόριθμο Token Bucket.
Δείτε:

- Linux Journal:
  `http://www.linuxjournal.com/article/2156`
- Wikipedia:
  `https://en.wikipedia.org/wiki/Token_bucket`
- Juniper Networks:
  `http://goo.gl/2v2S01`

Θέμα 2 (35%)

Εργάζεστε σε έναν πάροχο υπηρεσιών cloud, προσφέροντας στους πελάτες σας εικονικές μηχανές QEMU. Για λόγους ασφαλείας, οι πελάτες σας έχουν πλήρη διαχειριστική πρόσβαση μέσα στις εικονικές μηχανές, αλλά καμία πρόσβαση στους φυσικούς κόμβους στους οποίους εκτελούνται οι μηχανές. Οι πελάτες σας χρησιμοποιούν τις εικονικές μηχανές για να προσφέρουν με τη σειρά τους δικτυακές υπηρεσίες, π.χ. HTTPS και SSH servers. Οι υπηρεσίες αυτές απαιτούν για τη λειτουργία τους πρόσβαση σε αξιόπιστη ακολουθία τυχαίων αριθμών.

Στο Linux για το σκοπό αυτό υπάρχει το ειδικό αρχείο `/dev/random`, το οποίο μια διεργασία μπορεί να ανοίξει και να διαβάσει τυχαία bytes. Το `/dev/random` συλλέγει εντροπία από το σύστημα, παρατηρώντας εξωτερικά γεγονότα όπως την κίνηση του ποντικιού και τα χρονικά διαστήματα ανάμεσα σε πατήματα πλήκτρων. Επειδή ο ρυθμός με τον οποίο παράγονται τυχαίοι αριθμοί είναι πολύ μικρός, το πολύ μερικά Mbit/s, συνεχόμενες κλήσεις `read()` στη συσκευή `/dev/random` μπλοκάρουν έως ότου το σύστημα έχει συλλέξει αρκετή εντροπία ώστε να μπορεί να συνεχίσει.

Το πρόβλημα με την παραγωγή τυχαίων αριθμών σε εικονικές μηχανές είναι η έλλειψη καλών πηγών εντροπίας, γιατί δεν υπάρχουν οι αντίστοιχες φυσικές συσκευές. Η συσκευή `/dev/random` μπορεί να μιλά και με γεννήτριες τυχαίων αριθμών στο υλικό ("hardware random number generators"). Οι συσκευές αυτές παρακολουθούν φυσικά φαινόμενα (π.χ. θερμικό θόρυβο) για να εξάγουν μια ακολουθία τυχαίων αριθμών. Ως cloud provider, έχετε προνοήσει να τοποθετήσετε σε κάθε φυσικό server από μία τέτοια συσκευή παραγωγής τυχαίων αριθμών σε υλικό.

Για να κάνει διαθέσιμες αυτές τις συσκευές στις εικονικές μηχανές, η ομάδα σας έφτιαξε έναν οδηγό με χρήση του VirtIO split-driver model (frontend/backend). Ο οδηγός αυτός επιτρέπει στις εικονικές μηχανές να χρησιμοποιούν τις συσκευές που είναι συνδεδεμένες στους φυσικούς εξυπηρετητές για να αποκτούν τυχαίους αριθμούς. Το μόνο που χρειάζεται να κάνει μια διεργασία μέσα στην εικονική μηχανή είναι να ανοίξει το ειδικό αρχείο χαρακτήρων `/dev/virtio-random` και να διαβάσει από αυτό. Στο backend, ο οδηγός σας χρησιμοποιεί το αρχείο `/dev/random` του host για να λάβει δεδομένα από τη γεννήτρια τυχαίων αριθμών.

Σας δίνεται σκελετός για το frontend του οδηγού, το οποίο υλοποιεί έναν οδηγό χαρακτήρων στο ΛΣ Linux.

Ζητούνται τα εξής:

i. (2%) Για κάθε μία από τις συναρτήσεις που σας δίνονται, αναφέρετε αν εκτελούνται σε

process ή interrupt context.

ii. *(5%) Γιατί ο οδηγός χρησιμοποιεί την `copy_to_user()` και όχι απευθείας τη `memcpy()`; Περιγράψτε με ακρίβεια ένα σενάριο όπου ένας κακόβουλος χρήστης εικονικής μηχανής `VM1` θα μπορούσε να εκμεταλλευτεί ενδεχόμενη χρήση της `memcpy()` αντί της `copy_to_user()`, και τις συνέπειες του. Πώς θα επηρεαζόταν η `VM1` και πώς οι υπόλοιπες εικονικές μηχανές στον ίδιο φυσικό κόμβο;*

iii. *(4%) Πότε χρησιμοποιούνται `spinlocks` και πότε `semaphores` για την προστασία δομών στη μνήμη; Ο οδηγός που σας δίνεται χρησιμοποιεί τους σωστούς μηχανισμούς κλειδώματος; Εξηγήστε την απάντησή σας.*

iv. *(5%) Σε τι κατάσταση (process state) βρίσκεται μια διεργασία ενώ περιμένει την παραλαβή δεδομένων από την virtqueue; Τι επιπτώσεις έχει αυτό στην απόδοση του εικονικού συστήματος;*

v. *(10%) Σκιαγραφήστε σχεδίαση του οδηγού στην οποία οι διεργασίες περνάνε σε κατάσταση αναμονής έως ότου έρθουν τα δεδομένα που χρειάζονται. Ποια μέρη του οδηγού επηρεάζονται και με ποιον τρόπο;*

vi. *(4%) Υπάρχει περίπτωση μια κακόβουλη διεργασία στη `VM1` που διαβάζει συνεχώς από το `/dev/virtio-random` να προκαλέσει πρόβλημα στο υπόλοιπο σύστημα; Πώς θα επηρεαζόντουσαν διεργασίες στη `VM1` και πώς στις υπόλοιπες εικονικές μηχανές;*

vii. *(5%) Για να προστατευτείτε από αυτό το ενδεχόμενο, επεκτείνετε τον οδηγό ώστε το frontend να* περιορίζει *το ρυθμό με τον οποίο ζητά τυχαίους αριθμούς από το backend. Αρκεί αυτό ώστε να εξασφαλίσετε τη δίκαιη χρήση των φυσικών γεννητριών τυχαίων αριθμών; Αν όχι, τι πρότεινετε;*

i. Οι συναρτήσεις `rng_chrdev_read()`, `find_vq()` και `virtcons_probe()` εκτελούνται σε process context, η συνάρτηση `vq_has_data()` εκτελείται σε interrupt context. Η `virtcons_probe()` πρέπει να εκτελείται σε process context γιατί δεσμεύει μνήμη με όρισμα `GFP_KERNEL` στην `kmalloc()`, η `find_vq()` καλείται από την `virtcons_probe()` που τρέχει σε process context.

ii. Χρησιμοποιεί την `copy_to_user()` γιατί επιθυμεί να αντιγράψει δεδομένα στο χώρο χρήστη και δεν μπορεί να εμπιστευτεί το δείκτη `usrbuf` ακολουθώντας τον απευθείας, γιατί η διεργασία αποφασίζει ελεύθερα την τιμή του. Σενάριο: Η διεργασία περνά δείκτη `NULL`. Ο πυρήνας προκαλεί page fault (NULL pointer dereference) κατά το `memcpy()` που δεν μπορεί να χειριστεί. Αλλιώς: Η διεργασία μπορεί να περάσει όποιον δείκτη επιθυμεί κι ο πυρήνας θα πανωγράψει το περιεχόμενο της μνήμης του στο συγκεκριμένο σημείο. Έτσι, η διεργασία μπορεί να πανωγράψει το uid (user id) μέσα στο PCB της ώστε να φαίνεται ότι εκτελείται με τα δικαιώματα του `root`. Στην περίπτωση αυτή, η `VM1` θα ερχόταν υπό τον έλεγχο της συγκεκριμένης διεργασίας. Οι υπόλοιπες εικονικές μηχανές δεν επηρεάζονται, γιατί η διεργασία παραμένει εγκλωβισμένη στο εικονικό υλικό.

iii. Γενικά χρησιμοποιούνται `semaphores` όταν επιχειρεί να πάρει το lock μόνο κώδικας που τρέχει σε process context, οπότε μπορεί να κοιμηθεί αν δεν είναι διαθέσιμο, ενώ `spinlocks` όταν για το lock ανταγωνίζεται και κώδικας που τρέχει σε interrupt context, ο οποίος δεν μπορεί να κοιμηθεί. Ο κώδικας που μας δίνεται χρησιμοποιεί semaphores για να κλειδώσει τη virtqueue χωρίς πρόβλημα, γιατί μόνο process context, η `rng_chrdev_read()`, εκτελεί πρόσβαση στη virtqueue.

iv. Είναι RUNNING (ή και READY αν ο πυρήνας είναι preemptible) στο loop των γραμμών 46-47. Αυτό καταστρέφει την απόδοση της CPU, γιατί την κρατά κατειλημμένη

και δεν την αφήνει ελεύθερη να εκτελέσει χρήσιμο κώδικα ενώ η διεργασία περιμένει απάντηση από το εικονικό hardware.

v. Θα χρησιμοποιούσαμε μια waitqueue στην οποία θα κοιμόντουσαν οι διεργασίες αν δεν υπήρχε έτοιμη απάντηση στη virtqueue (`virtqueue_get_buf()`). Η `vq_has_data()` θα έτρεχε σε interrupt context όταν υπήρχε απάντηση και θα ξυπνούσε τη συγκεκριμένη ουρά. Ο έλεγχος με `virtqueue_get_buf()` μπορεί να συνεχίσει να γίνεται σε process context από τις διεργασίες.

vi. Ναι, μια κακόβουλη διεργασία μπορεί να διαβάζει συνεχώς το `/dev/virtio-random`. Επειδή αναγνώσεις του `/dev/virtio-random` καταλήγουν σε αναγνώσεις του `/dev/random` του host από το backend μέσα στο QEMU process που αντιστοιχεί στη VM1, αυτό θα οδηγούσε σε εξάντληση της διαθέσιμης εντροπίας από το `/dev/random` του host. Αυτό θα είχε ως αποτέλεσμα διεργασίες σε *κάθε* άλλη εικονική μηχανή να μπλοκάρουν σε αναγνώσεις στο `/dev/virtio-random` τους.

vii. Το frontend εκτελείται μέσα στην εικονική μηχανή, ως μέρος του εκάστοτε ΛΣ, π.χ. του Linux kernel και *δεν* είναι έμπιστο. Δεν μπορεί να αποκλείσει κανείς ένας κακόβουλος χρήστης να αντικαταστήσει με ό,τι θέλει τον κώδικα μέσα στην εικονική μηχανή, αλλάζοντας κατά βούληση τη συμπεριφορά του. Για το λόγο αυτό, για να είναι αξιόπιστη η όποια πολιτική περιορισμού του ρυθμού παροχής τυχαίων αριθμών πρέπει να εφαρμόζεται με αλλαγή στο έμπιστο *backend*, μέσα στον κώδικα του QEMU, ανεξάρτητα από το frontend.

Σχετικοί σύνδεσμοι:
Ο μηχανισμός πρόσβασης μέσω VirtIO σε γεννήτριες τυχαίων αριθμών σε υλικό (RNG) υπάρχει ήδη στο QEMU και ονομάζεται VirtIO RNG. Η τεκμηρίωσή του κάνει ειδική αναφορά στην ανάγκη περιορισμού του ρυθμού ανάκτησης τυχαίων αριθμών, για να αποφευχθεί εξάντληση της εντροπίας του φυσικού μηχανήματος: "This restricts the data sent to the guest at 1KB per second. This is useful to not let a guest starve the host of entropy."
Δείτε:

- VirtIO RNG QEMU feature:
  `http://wiki.qemu-project.org/Features/VirtIORNG`
- About Random Numbers and Virtual Machines:
  `http://log.amitshah.net/2013/01/about-random-numbers-and-virtual-machines/`
- LCE: Don't play dice with random numbers:
  `https://lwn.net/Articles/525459/`

```
1  struct rng_device {
2      struct list_head list;      /* Next rng device in the list,
3                                     head is in the rngdrvdata struct */
4      struct virtio_device *vdev; /* The virtio device we are
5                                     associated with */
6      struct virtqueue *vq;
7      struct semaphore vq_sem;    /* Semaphore to protect the virtqueue */
8      unsigned int minor;         /* The minor number of the device */
9  };
10
11 static ssize_t rng_chrdev_read(struct file *filp, char __user *usrbuf,
12     size_t cnt, loff_t *f_pos)
13 {
14     int ret, err, len;
```

```
15      struct rng_open_file *rngof = filp->private_data;
16      struct rng_device *rngdev = rngof->rngdev;
17      struct scatterlist syscall_type_sg, read_cnt_sg, buffer_sg, *sgs[3];
18      unsigned int syscall_type = VIRTIO_RNG_SYSCALL_READ;
19      char *kern_buffer;
20      size_t read_cnt;
21
22      if (cnt == 0)
23          return 0;
24  #define MAX_CNT_PER_READ 65536
25      if (cnt > MAX_CNT_PER_READ)
26          cnt = MAX_CNT_PER_READ;
27      read_cnt = cnt;
28
29      kern_buffer = kzalloc(cnt, GFP_KERNEL);
30      if (!kern_buffer) {
31          ret = -ENOMEM;
32          goto out;
33      }
34
35      sg_init_one(&syscall_type_sg, &syscall_type, sizeof(syscall_type));
36      sgs[0] = &syscall_type_sg;
37      sg_init_one(&read_cnt_sg, &read_cnt, sizeof(read_cnt));
38      sgs[1] = &read_cnt_sg;
39      sg_init_one(&buffer_sg, kern_buffer, cnt * sizeof(*kern_buffer));
40      sgs[2] = &buffer_sg;
41
42      down(&rngdev->vq_sem);
43      err = virtqueue_add_sgs(rngdev->vq, sgs, 1, 2,
44          &syscall_type_sg, GFP_ATOMIC);
45      virtqueue_kick(rngdev->vq);
46      while (virtqueue_get_buf(rngdev->vq, &len) == NULL)
47          /* do nothing */;
48      up(&rngdev->vq_sem);
49
50      ret = copy_to_user(usrbuf, kern_buffer, read_cnt);
51      if (ret) {
52          ret = -EFAULT;
53          goto out_with_buf;
54      }
55
56      ret = read_cnt;
57
58  out_with_buf:
59      kfree(kern_buffer);
60  out:
61      return ret;
62  }
63
64  static void vq_has_data(struct virtqueue *vq)
65  {
66  }
67
68  static struct virtqueue *find_vq(struct virtio_device *vdev)
69  {
70      int err;
71      struct virtqueue *vq;
72
73      vq = virtio_find_single_vq(vdev, vq_has_data, "rng-vq");
74      if (IS_ERR(vq)) {
```

```
75          debug("Could not find vq");
76          vq = NULL;
77      }
78
79      return vq;
80  }
81
82  static int virtcons_probe(struct virtio_device *vdev)
83  {
84      int ret = 0;
85      struct rng_device *rngdev;
86
87      rngdev = kzalloc(sizeof(*rngdev), GFP_KERNEL);
88      if (!rngdev) {
89          ret = -ENOMEM;
90          goto out;
91      }
92
93      rngdev->vdev = vdev;
94      vdev->priv = rngdev;
95
96      rngdev->vq = find_vq(vdev);
97      if (!(rngdev->vq)) {
98          ret = -ENXIO;
99          goto out;
100     }
101
102     sema_init(&rngdev->vq_sem, 1);
103
104     /* Grab the next minor number and put the device in the driver's list */
105     spin_lock_irq(&rngdrvdata.lock);
106     rngdev->minor = rngdrvdata.next_minor++;
107     list_add_tail(&rngdev->list, &rngdrvdata.devs);
108     spin_unlock_irq(&rngdrvdata.lock);
109     debug("Got minor = %u", rngdev->minor);
110
111 out:
112     return ret;
113 }
```

Θέμα 3 (25%)

Θεωρήστε έναν περιηγητή ιστού στο Linux, όπως ο Firefox, ο οποίος υποστηρίζει plugins *γραμμένα από τρίτους για την υποστήριξη διαφορετικών μορφών αρχείου ήχου, π.χ.* `mp3`*. Θεωρήστε ότι τα plugins διανέμονται ως μοιραζόμενες βιβλιοθήκες και υλοποιούν συναρτήσεις*

- `int encode_snd(char *in, int ilen, char *out, int *olen);`
- `int decode_snd(char *in, int ilen, char *out, int *olen);`

Έστω ότι ο χρήστης `user` *εγκαθιστά το plugin* `p1` *για αναπαραγωγή αρχείων ήχου* `mp5` *στον Firefox. Απαντήστε στα εξής:*

i. *(2%) Μέσα σε ποια διεργασία εκτελείται ο κώδικας της* `decode_snd()` *και με τα δικαιώματα ποιου χρήστη;*

ii. *(2%) Τι θα συμβεί στον Firefox αν ο κώδικας του `p1` έχει κάποιο προγραμματιστικό σφάλμα, π.χ. λόγω bug κάνει προσπέλαση εκτός ορίων κάποιου πίνακα;*
iii. *(3%) Περιγράψτε πολύ συνοπτικά πώς θα μπορούσε ο Firefox να προστατευτεί από τέτοια δυσλειτουργία.*
iv. *(3%) Έστω ότι ο κώδικας του `p1` είναι* κακόβουλος. *Περιγράψτε την αλληλουχία κλήσεων συστήματος που μπορεί να εκτελέσει ώστε να αποστείλει δικτυακά το περιεχόμενο του αρχείου `/home/user/secret_passwd` σε επιτιθέμενο.*

i. Υποθέτοντας ότι ένα νήμα του Firefox καλεί απευθείας την `decode_snd()`, ο κώδικάς της εκτελείται μέσα στη διεργασία του Firefox με τα δικαιώματα της διεργασίας, δηλαδή τα δικαιώματα του χρήστη που έτρεξε τη συγκεκριμένη διεργασία.
ii. Η διεργασία που τρέχει αυτόν τον κώδικα (ο Firefox) θα δεχτεί σήμα `SIGSEGV` και αν δεν το χειριστεί θα τερματιστεί από τον πυρήνα. Γενικά, κώδικας που εκτελείται μέσα στη διεργασία έχει πλήρη πρόσβαση στη μνήμη και τους υπόλοιπους πόρους της διεργασίας.
iii. Μπορεί να δημιουργεί χωριστή διεργασία-παιδί μέσα στην οποία θα τρέχει το plugin και να επικοινωνεί μαζί της μέσω καθορισμένου μηχανισμού IPC, π.χ. pipes. Στην περίπτωση αυτή η διεργασία-παιδί θα τερματιστεί χωρίς να επηρεαστεί η γονική διεργασία, η οποία μπορεί να εμφανίσει κατάλληλο διαγνωστικό (”το plugin πέθανε”), και να συνεχίσει τη λειτουργία της. Ο Firefox στα Windows κάνει ακριβώς αυτό, εκτελώντας τα plugins σε χωριστή διεργασία με εκτελέσιμο `plugin-container.exe`.
iv. Ακόμη κι αν ο Firefox χρησιμοποιεί χωρίστή διεργασία-παιδί, αυτή εκτελείται με τα ίδια δικαιώματα με τη γονική διεργασία. Αν ο κώδικας είναι κακόβουλος, μπορεί να κάνει `open("secret_passwd", ...)` και `read()` για να αποκτήσει ευαίσθητη πληροφορία και στη συνέχεια `socket(AF_INET, SOCK_STREAM, ...)`, `connect()` σε προκαθορισμένο προορισμό και `write()` για να την παραδώσει μέσω TCP/IP σε υπολογιστή όπου υποθέτουμε ότι ακούει ο δημιουργός του.

Θεωρήστε ότι ο πυρήνας του Linux επεκτείνεται ώστε να περιέχει κλήση συστήματος `start_secure()`. Από τη στιγμή που μια διεργασία εκτελέσει αυτή την κλήση, κάθε *επόμενη κλήση της εκτός των `read()`, `write()`, `exit()` αποτυγχάνει με κωδικό σφάλματος `EPERM` (Permission Denied).*

v. *(6%) Περιγράψτε αναλυτικά επέκταση του μηχανισμού εκτέλεσης των plugins με χρήση της `start_secure()` έτσι ώστε ο χρήστης να προστατεύεται και από κακόβουλο κώδικα της περίπτωσης (iv). Δώστε ψευδοκώδικα σε C για το βελτιωμένο πλαίσιο εκτέλεσης του plugin. Θεωρήστε ήδη διαθέσιμα τα σύμβολα `encode_snd()`, `decode_snd()`.*
vi. *(4%) Έστω ένα νήμα του Firefox που πριν καλούσε απευθείας την `decode_snd()`. Τώρα, στο νέο πλαίσιο που ορίσατε, τι πρέπει να κάνει; Πώς μπορεί στο συγκεκριμένο πλαίσιο ο Firefox να αποκωδικοποιεί δύο αρχεία ήχου ταυτόχρονα;*
vii. *(5%) Έστω ότι ο Firefox έχει ήδη στη μνήμη του πίνακα `mp5data[LEN]` που περιέχει ηχητικά δεδομένα προς αναπαραγωγή. Περιγράψτε βήμα προς βήμα τον τρόπο με τον οποίο ένα νήμα θα χρησιμοποιήσει το plugin στο σενάριό σας, ώστε να μπορέσει να τα αποσυμπιέσει και να τα αναπαραγάγει από τα ηχεία. Θεωρήστε ότι ο Firefox έχει ήδη ανοιχτό περιγραφητή αρχείου `audiofd`. Εκεί, μπορεί να κάνει `write()` ασυμπίεστα*

δείγματα ήχου PCM για να ακουστούν από τα ηχεία. Οι κλήσεις `write()` μπορεί να μπλοκάρουν.

Μπορείτε να κάνετε οποιαδήποτε λογική παραδοχή για τη λειτουργία των `encode_snd()`, `decode_snd()` χρειάζεστε, αρκεί να την περιγράψετε συγκεκριμένα.

i. Η γενική ιδέα είναι: κάθε φορά που ο Firefox επιθυμεί να χρησιμοποιήσει ένα plugin θα δημιουργεί μια νέα διεργασία-παιδί. Η διεργασία-παιδί θα απαγορεύει στον εαυτό της οποιαδήποτε άλλη κλήση συστήματος εκτός των `read()`, `write()`, `exit()` εκτελώντας `start_secure()` και μετά μπορεί να καλέσει ελεύθερα οποιαδήποτε μη-έμπιστη συνάρτηση. Το σενάριο του (iv) δεν είναι πλέον λειτουργικό, γιατί η διεργασία που εκτελεί την `decode_snd()` δεν μπορεί *καν* να εκτελέσει `open()` ή `socket()`: αν προσπαθήσει η κλήση θα αποτύχει με `EPERM`. Η όποια επικοινωνία με τον έξω κόσμο γίνεται μέσω προκατασκευασμένων pipes ή ανάλογων μηχανισμών IPC, που φροντίζει ο Firefox να δημιουργεί πριν από το `fork()`. Στο ακόλουθο, υποθέτουμε ότι ο Firefox επιθυμεί να αποκωδικοποιεί δεδομένα mp5.
   Για απλότητα στον χειρισμό των buffers, υποθέτουμε ότι η μέγιστη είσοδος και έξοδος της `decode_snd()` είναι μεγέθους `MAXSZ`, καθώς και ότι η `decode_snd()` μπορεί να δεχτεί είσοδο δεδομένων οποιουδήποτε μήκους χωρίς περιορισμούς στοίχισης (alignment). Αν αυτά δεν ισχύουν, θα μπορούσαμε να επεκτείνουμε το πρωτόκολλο χρήσης των pipes ώστε ο Firefox να περνάει συγκεκριμένο header με το μέγεθος του μηνύματος κάθε φορά, αντί για raw δεδομένα. Τέλος, δεν κάνουμε έλεγχο λαθών στις κλήσεις συστήματος.

```
1   pipe(sendp);
2   pipe(recvp);
3   if (fork() == 0) {
4       /* Plugin process */
5       close(recvp[0]);
6       close(sendp[1]);
7       start_secure();
8       for (;;) {
9           n = read(sendp[0], buf, MAXSZ);
10          if (n == 0)
11              exit();
12          decode_snd(buf, n, outbuf, &olen);
13          write(recvp[1], outbuf, olen);
14      }
15  }
16  /* Firefox process */
17  close(recvp[1]);
18  close(sendp[0]);
19  /* ... use sendp and recvp pipes to communicate with plugin ... */
```

ii. Πρέπει να γράφει τα δεδομένα προς αποκωδικοποίηση στο pipe `sendp` και να διαβάζει τα δεδομένα που αποτελούν το αποτέλεσμα της αποκωδικοποίησης από το pipe `recvp`. Για την απομονωμένη αποκωδικοποίηση δύο διαφορετικών αρχείων, μπορεί να δημιουργήσει δύο διαφορετικές διεργασίες-στιγμιότυπα του plugin, με διακριτό ζεύγος `sendp`/`recvp` για την κάθε μία. Οι διεργασίες θα λειτουργούν παράλληλα, απομονωμένα, χωρίς δυσλειτουργία της μίας να μπορεί να επηρεάσει την άλλη.

iii. Στα ακόλουθα υποθέτουμε ότι ένα νήμα αναλαμβάνει και να προωθήσει τα δεδομένα προς αποκωδικοποίηση στο plugin και να στείλει στην κάρτα ήχου τα δεδομένα που λαμβάνει από το plugin. Το νήμα βασίζεται στη `select()` για να ξέρει πότε μπορεί να ανταλλάξει (διαβάσει/γράψει) δεδομένα με τα pipes και το `audiofd` χωρίς να μπλοκάρει. Χρησιμοποιεί τον πίνακα `rawdata[MAXSZ]` για να αποθηκεύει προσωρινά τα δεδομένα που λαμβάνει κάθε φορά από το plugin πριν τα προωθήσει στην κάρτα ήχου και τις μεταβλητές `mp5i`, `mp5rem` (τρέχουσα θέση αποκωδικοποίησης, πλήθος bytes που απομένουν στον πίνακα `mp5data[]`, αντίστοιχα) και `rawi`, `rawrem` (τρέχουσα θέση αναπαραγωγής, πλήθος bytes που απομένουν στον πίνακα `rawdata[]`, αντίστοιχα).
Ιδανικά, το νήμα πρέπει να κρατάει λογαριασμό και για τις τρεις λειτουργίες (αποστολή κωδικοποιημένων δεδομένων προς το plugin, παραλαβή αποκωδικοποιημένων δεδομένων από το plugin και αποστολή τους στην κάρτα ήχου) και να κάνει `read()`/`write()` μόνο όταν ξέρει ότι δεν θα μπλοκάρει.
Αναλυτικά, σε ψευδοκώδικα C που υποθέτουμε ότι υποστηρίζει πράξεις με σύνολα:

```
1   mp5rem = LEN;
2   mp5i = rawi = rawrem = 0;
3   for (;;) {
4       /* Prepare read, write fd sets for select */
5       read_set = [];
6       write_set = [];
7       if (mp5rem > 0) /* Need to send mp5-encoded data to plugin? */
8           write_set += sendp[1];
9       if (rawrem > 0) /* Need to write to sound card fd? */
10          write_set += audiofd;
11      if (rawrem == 0) /* Need to receive raw data from plugin? */
12          read_set += recvp[0];
13      /* Call select(). This modifies read_set, write_set in place */
14      select(&read_set, &write_set);
15      if (recvp[0] in read_set) {
16          rawi = 0;
17          rawrem = read(recvp[0], rawdata, MAXSZ);
18          if (rawrem == 0)   /* EOF from plugin, we're done */
19              break;
20      }
21      if (sendp[1] in write_set) {
22          n = write(sendp[1], mp5data[mp5i], mp5rem);
23          mp5i += n;
24          mp5rem -= n;
25          if (mp5rem == 0) {
26              /* Close pipe end, so plugin receives EOF and exit()s */
27              close(sendp[1]);
28          }
29      }
30      if (audiofd in write_set) {
31          n = write(audiofd, rawdata[rawi], rawrem);
32          rawi += n;
33          rawrem -= n;
34      }
35  }
```

Σχετικοί σύνδεσμοι:
Ο μηχανισμός που περιγράφεται υπάρχει ήδη στον πυρήνα του Linux, ονομάζεται seccomp (secure computing) και κατασκευάστηκε αρχικά ακριβώς γι'αυτό το σκοπό, τη δυνατότητα εκτέλεσης μη-έμπιστου κώδικα.

Δείτε:

- Securely renting out your CPU with Linux:
  `http://lwn.net/Articles/120647/`
- Seccomp and sandboxing:
  `http://lwn.net/Articles/332974/`
- Χρήση του seccomp από τον Google Chome για περιορισμό του Flash:
  `http://scarybeastsecurity.blogspot.gr/2012/07/chrome-20-on-linux-and-flash-sandboxing.html`
- Wikipedia:
  `https://en.wikipedia.org/wiki/Seccomp`