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 2015-2016

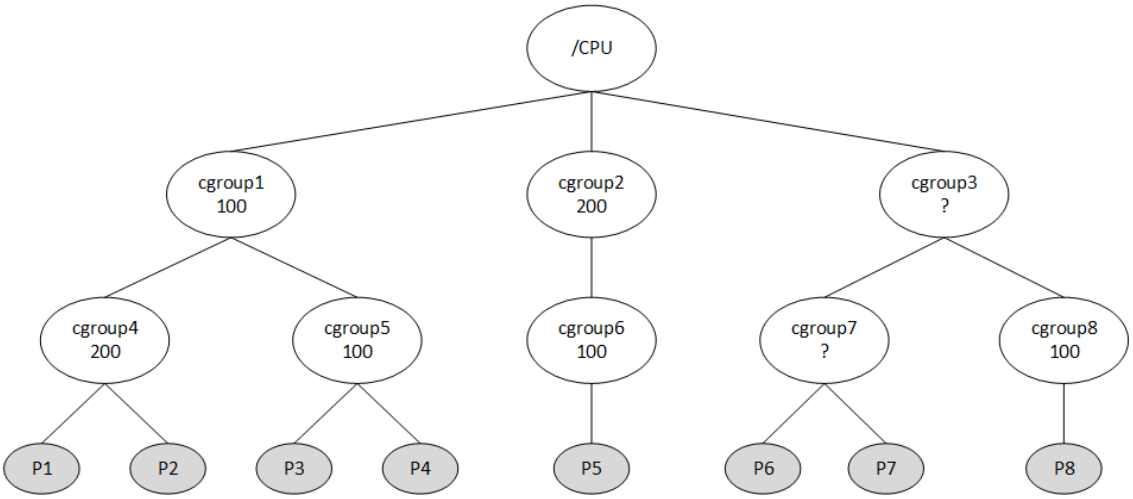
Επαναληπτική Εξέταση

Διάρκεια: 2 ώρες, 45 λεπτά

Η εξέταση πραγματοποιείται με ανοιχτά βιβλία και σημειώσεις.

Θέμα 1 (25%)

α (5%). Σας δίνεται η ακόλουθη ιεραρχία ενός υποσυστήματος κεντρικής υπολογιστικής μονάδας (CPU subsystem).



i. Να υπολογιστούν οι τιμές των *cpu.shares* των *cgroup3*, *cgroup7* έτσι ώστε οι διεργασίες P6, P8 να καταναλώνουν το 1/6 της συνολικής διαθέσιμης υπολογιστικής ισχύος έκαστη.

ii. Να υπολογίσετε την υπολογιστική ισχύ που καταναλώνουν οι υπόλοιπες διεργασίες που απεικονίζονται στο άνωθεν δένδρο (P1, P2, P3, P4, P5, P7).

Θεωρήστε πως όλες οι διεργασίες μπορούν να καταναλώσουν τη μέγιστη διαθέσιμη υπολογιστική ισχύ.

β (10%). Έστω κάποιο πειραματικό σενάριο κατά το οποίο δύο διεργασίες, **A** και **B** τρέχουν ταυτόχρονα στο λειτουργικό σύστημα *linux* με ενεργοποιημένο το μηχανισμό των *cgroups*. Και οι δύο διεργασίες απαιτείται να εκτελεστούν για το ίδιο ακριβώς χρονικό διάστημα **T** κατά το οποίο η διεργασία **B** παρουσιάζει πλήρως επιθετικό προφίλ (τείνει να καταναλώνει όλο το διαθέσιμο χρόνο CPU), ενώ η διεργασία **A** χαρακτηρίζεται από δύο φάσεις επεξεργασίας, χρόνου **T1** και **T2** αντίστοιχα. Κατά τη διάρκεια της πρώτης φάσης (**T1**), λόγω αναμονής εισόδου δεδομένων, η **A** καταναλώνει ποσοστό **A1** του χρόνου της CPU, ενώ κατά τη διάρκεια της δεύτερης φάσης (**T2**), η **A** εμφανίζει επίσης **επιθετικό προφίλ** ως προς την απαίτηση χρόνου CPU, καταναλώνοντας όσο περισσότερο χρόνο CPU της επιτρέπεται.

Στόχος μας λοιπόν είναι, αντιμετωπίζοντας τη διεργασία **B** ως **ελαστική**, η **A** να καταναλώσει κατά μέσο όρο ποσοστό **ακριβώς M1** του συνολικού χρόνου CPU καθόλη τη διάρκεια των δύο φάσεών της (T = T1 + T2).

Βάσει της παραπάνω περιγραφής σας ζητείται:

1. Να δώσετε τις ρυθμίσεις που πρέπει να γίνουν στο υποσύστημα `cgroup cpu` καθώς και τις χρονικές στιγμές που πρέπει αυτές να λάβουν χώρα.
2. Να περιγράφει ο μηχανισμός ο οποίος σε περίπτωση που ο ανωτέρω στόχος θεωρείται ανέφικτος, να παράγει την αντίστοιχη προειδοποίηση.

γ (10%). Σας ζητείται η υλοποίηση ενός ελαστικού μηχανισμού διαμοιρασμού υπολογιστικού χρόνου κεντρικής μονάδας επεξεργασίας (CPU), ο οποίος:

- θα διασφαλίζει κάποιο ελάχιστο ποσοστό χρόνου εκτέλεσης (εκφρασμένου σε χιλιοστά CPU) για κάθε εφαρμογή.
- σε περίπτωση που είναι δυνατόν, θα πριμοδοτεί το χρόνο που μπορεί να καταναλώσει κάθε διεργασία ανάλογα με τον εκτιμώμενο χρόνο ολοκλήρωσής της. Έτσι, αν π.χ. η εφαρμογή A υπολογίζεται πως απαιτεί λιγότερο χρόνο εκτέλεσης από την εφαρμογή B, τότε είναι επιθυμητό να λάβει περισσότερο χρόνο CPU από την τελευταία, **μόνο για το ποσοστό πέρα από τα εξασφαλισμένα ελάχιστα και των δύο**. Πιο γενικά, σε σύστημα με ωφέλιμο χρόνο υπολογιστικής ισχύος `CPU_TIME`, όπου συνυπάρχουν n διεργασίες $(\delta_1, \delta_2, \ldots, \delta_n)$ με ελάχιστα εγγυημένα ποσοστά υπολογιστικής ισχύος $p_1, p_2, \ldots, p_n$ και με εκτιμώμενους χρόνους εκτέλεσης $t_1, t_2, \ldots, t_n$, θέλουμε η καθεμία να λαμβάνει χρόνο (σε χιλιοστά CPU):

$$T_i = \begin{cases} p_i, & n = 1 \\ p_i + \frac{1}{n-1}\left(1 - \frac{t_i}{\sum_{j=1}^{n} t_j}\right) \times \left(CPU\_TIME - \sum_{j=1}^{n} p_j\right), & n > 1 \end{cases} \qquad i \in [1, n]$$

Να υλοποιηθεί ένα policy module που να διαβάζει από το stdin είσοδο μορφοποιημένη ως εξής:

- στην πρώτη γραμμή έναν ακέραιο που αντιστοιχεί στη συνολική διαθέσιμη υπολογιστική ισχύ (σε χιλιοστά CPU).
- σε κάθε μία από τις επόμενες το `pid` μιας διεργασίας, έναν **ακέραιο** που θα εκφράζει το ελάχιστο εγγυημένο ποσοστό χρόνου υπολογιστικής ισχύος σε χιλιοστά CPU (CPU shares), καθώς και έναν **ακέραιο** που να υποδηλώνει τον εκτιμώμενο χρόνο εκτέλεσης (σε ms) της διεργασίας:

  ```
  <CPU_TIME>
  <task_pid_1>:<minimum_shares_1>:<estimated_completion_time_1>
  ...
  <task_pid_n>:<minimum_shares_n>:<estimated_completion_time_n>
  ```

  και να έχει έξοδο *(stdout)* της μορφής:

  ```
  <task_pid_1>:<cpu.shares_1>
  ...
  <task_pid_n>:<cpu.shares_n>
  ```

  ή το αντίστοιχο **μήνυμα λάθους** σε περίπτωση μη επάρκειας πόρων για την εξυπηρέτηση του αιτήματος.

Θέμα 2 (40%)

Σας ζητείται να υλοποιήσετε οδηγό για ειδική συσκευή χαρακτήρων, η οποία χρησιμοποιείται για αποδοτική αποθήκευση δεδομένων. Η συσκευή έχει την εξής ιδιότητα: η εγγραφή ενός MB δεδομένων απαιτεί τον ίδιο χρόνο με την εγγραφή ενός byte. Για να εκμεταλλευτείτε αυτό το χαρακτηριστικό, αποφασίσατε να προσθέσετε ένα ενδιάμεσο επίπεδο κρυφής μνήμης (cache). Συγκεκριμένα, τα δεδομένα αποθηκεύονται αρχικά σε έναν προσωρινό buffer (1MB) στη μνήμη και όταν αυτός γεμίσει, τα δεδομένα μεταφέρονται στη συσκευή (flush).

Ο οδηγός υλοποιεί μία συσκευή χαρακτήρων `/dev/cache`. Ισχύουν τα ακόλουθα:

- Η κλήση συστήματος `write()` λειτουργεί ως εξής: γράφει στον προσωρινό buffer όσα bytes χωράνε τη δεδομένη στιγμή και επιστρέφει την κατάλληλη τιμή, χωρίς να προχωρήσει σε κάποια άλλη ενέργεια.
- Σε περίπτωση που η `write()` βρει τον buffer γεμάτο, εκκινεί την διαδικασία εγγραφής των δεδομένων του προσωρινού buffer στην συσκευή (flush) και κοιμάται σε

ουρά αναμονής, μέχρι να ολοκληρωθεί η εγγραφή (flush) και να υπάρξει διαθέσιμος χώρος στον buffer.

- Όταν ολοκληρωθεί η εγγραφή των δεδομένων στη συσκευή (flush), η συσκευή παράγει διακοπή υλικού. Η αντίστοιχη συνάρτηση χειρισμού διακοπών (`flush_completed()`) μαρκάρει τον buffer ως κενό και ξυπνάει τις διεργασίες που περιμένουν στη σχετική ουρά αναμονής.
- Μία διεργασία μπορεί να καλέσει την `initiate_flush()` **αν και μόνο αν** δε βρίσκεται σε εξέλιξη λειτουργία flush.

Δίνεται επίσης:

- `void initiate_flush(struct cache_device *dev)`:
  Προγραμματίζει τη συσκευή να εκκινήσει τη λειτουργία αποθήκευσης (flush) και επιστρέφει αμέσως (non-blocking).

Δίνεται ο σκελετός του οδηγού. Μπορείτε να θεωρήσετε ότι οι κλήσεις αντιγραφής μνήμης επιτυγχάνουν πάντα:

```
struct cache_device {
#define BUF_SIZE (1024*1024)  /* 1MB */
        char buf[BUF_SIZE];
        unsigned int cnt;
        int flush_in_progress;

        ...locktype... lock;

        wait_queue_head_t wq;
} cache_dev;

void flush_completed(unsigned int intr_mask)
{
    . . .
}

static ssize_t cache_write(struct file *filp, const char __user *buf,
                           size_t len, loff_t *pos)
{
    . . .
}
```

Ζητούνται τα εξής:

i. (14%) Υλοποιήστε την `cache_write()`.
ii. (10%) Υλοποιήστε την `flush_completed()`.
iii. (6%) Τι τύπου είναι το κλείδωμα `lock` της δομής `cache_dev` και γιατί; Τι προστατεύει και από ποιες οντότητες; Αναφέρατε δύο περιπτώσεις κατά τις οποίες η απουσία κλειδώματος θα δημιουργούσε πρόβλημα.
iv. (4%) Τι μετράει το πεδίο `cnt` της δομής `cache_dev`; Από ποια/ες οντότητα/ες τροποποιείται και σε ποια/ες περίπτωση/περιπτώσεις;
v. (6%) Πώς αποτρέπεται στον κώδικά σας η κλήση της `initiate_flush()`, ενώ ήδη βρίσκεται σε εξέλιξη λειτουργία flush;

Θέμα 3 (35%)

Θεωρούμε συσκευή χαρακτήρων `/dev/crypto`, η οποία χρησιμοποιείται για την κρυπτογράφηση ή αποκρυπτογράφηση ενός ρεύματος χαρακτήρων εισόδου. Η συσκευή δέχεται ως είσοδο έναν πίνακα χαρακτήρων και επιστρέφει τον αντίστοιχο κρυπτογραφημένο/αποκρυπτογραφημένο πίνακα χαρακτήρων. Στοχεύουμε στην υποστήριξη της συσκευής σε περιβάλλον εικονικών μηχανών και αποφασίζουμε την υλοποίηση του οδηγού συσκευής στο QEMU/KVM σε ΛΣ Linux με χρήση του VirtIO split-driver model (frontend/backend). Σας δίνεται σκελετός υλοποίησης του frontend μέρους, όπως εκτελείται στο χώρο πυρήνα του guest και του backend μέρους, όπως εκτελείται στο χώρο χρήστη του host.

Επίσης, δίνονται τα ακόλουθα:

- Η κλήση συστήματος `ioctl()` της συσκευής δέχεται δύο διαφορετικά ορίσματα `cmd`, ENCRYPT και DECRYPT για κρυπτογράφηση και αποκρυπτογράφηση αντίστοιχα της εισόδου.
- Η `ioctl()` στη γραμμή 18 του backend επιτυγχάνει πάντα.

Ζητούνται τα εξής:

i. (10%) Υλοποιήστε τη `virtio_crypto_ioctl()` συμπληρώνοντας το σκελετό.
ii. (5%) Κάνατε χρήση των `copy_from_user()`/`copy_to_user()`; Αν ναι, αναφέρετε ένα λόγο για τον οποίο δεν μπορεί να χρησιμοποιηθεί απευθείας η `memcpy()` αντί των προαναφερθεισών συναρτήσεων για αντιγραφή δεδομένων από το χώρο χρήστη. Αναφέρετε συνοπτικά το σκοπό που εξυπηρετούν οι `copy_from_user()`/`copy_to_user()` για κάθε σημείο που τις χρησιμοποιήσατε.
iii. (5%) Υλοποιήστε τη `vq_crypto_callback()` συμπληρώνοντας το σκελετό.

```
1  /* The struct that is being exchanged via the virtqueue. */
2  struct crypto_buffer {
3      char *input;           /* The input string. */
4      char *output;          /* The output string. */
5      unsigned int len;      /* The length of the input. */
6      char *key;             /* The key used for encryption/decryption. */
7      unsigned int key_len;  /* The length of the key. */
8  };
9
10 struct crypto_device {
11     struct virtqueue *vq;
12     struct semaphore vq_lock;
13 } crypto_dev;
14
15 static long virtio_crypto_ioctl(struct file *filp, unsigned int cmd,
16                                 unsigned long arg)
17 {
```

```
18      struct scatterlist cmd_sg, cbuf_sg, cbuf_input_sg, cbuf_output_sg,
19                              cbuf_key_sg, *sgs[5];
20      struct crypto_device *c_dev = &crypto_dev;
21      struct crypto_buffer *cbuf;
22      char *input_ptr, *output_ptr;
23      unsigned int len;
24      long ret = 0;
25
26      /* Fetch all necessary data from userspace. */
27      ... ??? ...
28
29      switch (cmd) {
30      case ENCRYPT:
31      case DECRYPT:
32          sg_init_one(&cmd_sg, &cmd, sizeof(cmd));
33          sgs[0] = &cmd_sg;
34          ... ??? ...
35
36          /* Send sgs and notify the host. */
37          down_interruptible(&c_dev->vq_lock);
38          virtqueue_add_sgs(c_dev->vq, sgs, 4, 1, cbuf, GFP_ATOMIC);
39          virtqueue_kick(vq);
40
41          /* Spin on the virtqueue until the buffer is back. */
42          while (virtqueue_get_buf(c_dev->vq, &len) == NULL)
43              /* do nothing */;
44          up(&c_dev->vq_lock);
45
46          break;
47
48      default:
49          return -EINVAL;
50      }
51
52      /* Copy all necessary data back to userspace. */
53      ... ??? ...
54
55      return ret;
56  }
57
58  static void crypto_recv(struct virtqueue *vq)
59  {
60      /* Nothing to do here. */
61  }
62
63  static int virtcrypto_probe(struct virtio_device *vdev)
64  {
65      ...
```

```
66      struct crypto_device *c_dev = &crypto_dev;
67      c_dev->vq = virtio_find_single_vq(vdev, crypto_recv, "crypto-vq");
68      ...
69  }
```

Ακολουθεί η συνάρτηση χειρισμού στην πλευρά του host:

```
1   void vq_crypto_callback(VirtIODevice *vdev, VirtQueue *vq)
2   {
3       VirtQueueElement elem;
4       struct crypto_buffer *cbuf;
5       char *input, *output, *key;
6       int ret;
7       int fd; /* This is an open instance of /dev/crypto on the host. */
8       char *input_saved, *output_saved, *key_saved;
9       unsigned int cmd;
10
11      if (!virtqueue_pop(vq, &elem))
12          return;
13
14      cmd = *elem.out_sg[0];
15      ... ??? ...
16
17      /* ioctl() to the host device driver, it is always successful. */
18      ret = ioctl(fd, cmd, cbuf);
19      ... ??? ...
20
21      virtqueue_push(vq, &elem, 0);
22      virtio_notify(vdev, vq);
23  }
```

Στη συνέχεια σας δίνεται μία δεύτερη υλοποίηση για το frontend μέρος του οδηγού. Συγκεκριμένα, οι γραμμές 36–44 και 58–61 της πρώτης υλοποίησης έχουν αντικατασταθεί αντίστοιχα με τις γραμμές 19–31 και 42–55 της δεύτερης υλοποίησης. Επίσης, έχουν προστεθεί στη δομή `crypto_device` δύο πεδία: κατάλληλη δομή (`buf_pool`) για αποθήκευση των απομονωτών που λαμβάνονται από τον host, καθώς και το `wq`. Επιπλέον, έχουν χρησιμοποιηθεί οι ακόλουθες συναρτήσεις:

- `void add_buf_to_pool(struct crypto_buffer *buf)`: Προσθέτει τον απομονωτή `buf` στη δομή `buf_pool` του `crypto_dev`. Χρησιμοποιεί τα κατάλληλα κλειδώματα.
- `bool is_this_my_buf(struct crypto_buffer *buf)`: Επιστρέφει αληθές αν ο απομονωτής `buf` βρίσκεται στη δομή `buf_pool`, αλλιώς ψευδές. Χρησιμοποιεί τα κατάλληλα κλειδώματα.

Το backend μέρος του οδηγού παραμένει το ίδιο.

Τέλος, δίνεται από τις προδιαγραφές της φυσικής συσκευής ότι η διαδικασία της κρυπτογράφησης / αποκρυπτογράφησης είναι χρονοβόρα και για αυτό το λόγο η συσκευή διαθέτει `N` υπολογιστικούς πυρήνες, οι οποίοι μπορούν να λειτουργούν παράλληλα για διαφορετικές εισόδους.

Υπόδειξη: Η συνάρτηση `virtqueue_add_sgs()` δέχεται ως 5ο όρισμα μία διεύθυνση, η οποία λειτουργεί ως αναγνωριστικό (token) για τον κάθε buffer. Αυτή η διεύθυνση επιστρέφεται από την `virtqueue_get_buf()` για τον αντίστοιχο buffer.

Απαντήστε στα ακόλουθα:

iv. (3%) Ποια από τις δύο υλοποιήσεις θα επιλέγατε, ώστε να εκμεταλλευτείτε την παραλληλία που σας προσφέρει το υλικό και γιατί;
v. (7%) Τι προκαλεί η κλήση της `virtio_notify()` στο εικονικό μηχάνημα; Ποια είναι η επόμενη συνάρτηση που καλείται στον πυρήνα του guest ως αποτέλεσμα της `virtio_notify()` και γιατί; Σε τι context (process/interrupt) εκτελείται; Η `virtio_notify()` απαιτείται και στις δύο υλοποιήσεις, σε μία από τις δύο ή σε καμμία και γιατί;
vi. (5%) Για κάθε μία από τις δύο υλοποιήσεις αναφέρετε σε τι κατάσταση (process state) βρίσκεται μια διεργασία ενώ περιμένει την παραλαβή δεδομένων από την virtqueue.

```
1  struct crypto_device {
2      struct virtqueue *vq;
3      spinlock_t vq_lock;
4      ... buffer_pool; /* Stores the buffers received from host. */
5      wait_queue_head_t wq;
6  } crypto_dev;
7
8
9  static long virtio_crypto_ioctl(struct file *filp, unsigned int cmd,
10                                  unsigned long arg)
11 {
12     ...
13
14     switch (cmd) {
15     case ENCRYPT:
16     case DECRYPT:
17         ...
18
19         /* send sg and notify the host */
20         spin_lock_irqsave(&c_dev->vq_lock, flags);
21         virtqueue_add_sgs(c_dev->vq, sgs, 4, 1, cbuf, GFP_ATOMIC);
22         virtqueue_kick(vq);
23         spin_unlock_irqrestore(&c_dev->vq_lock, flags);
24
25         while (!is_this_mybuf(cbuf)) {
26             ret = wait_event_interruptible(c_dev->wq, is_this_mybuf(cbuf));
27             if (ret < 0) {
```

```
28                  ret = -EIO;
29                  goto out_with_cbuf;
30              }
31          }
32
33          break;
34
35      default:
36          return -EINVAL;
37      }
38
39      ...
40  }
41
42  static void crypto_recv(struct virtqueue *vq)
43  {
44      struct crypto_device *c_dev = &crypto_dev;
45      struct crypto_buffer *new_buf;
46      unsigned int len;
47      unsigned long flags;
48
49      spin_lock_irqsave(&c_dev->vq_lock, flags);
50      new_buf = virtqueue_get_buf(vq, &len);
51      add_buf_to_pool(new_buf);
52      spin_unlock_irqrestore(&c_dev->vq_lock, flags);
53
54      wake_up_interruptible(wq);
55  }
```