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

Επαναληπτική 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%)

Συμμετέχετε σε ομάδα που σχεδιάζει και υλοποιεί ενσωματωμένο σύστημα Linux το οποίο ελέγχει την αναπαραγωγή μουσικής στο ηχοσύστημα MEGA BLASTER. Ο ήχος αναπαριστάται ως μονοφωνικό ρεύμα δειγμάτων PCM, 8-bit, με ρυθμό δειγματοληψίας 22kHz.

Θεωρήστε τον οδηγό συσκευής για το υλικό αναπαραγωγής ήχου ("κάρτα ήχου") σε ΛΣ Linux. Από την πλευρά των διεργασιών ισχύουν τα εξής:

1. *Μια διεργασία μπορεί να παίξει μουσική ανοίγοντας ειδικό αρχείο συσκευής χαρακτήρων `/dev/audio` και γράφοντας σε αυτό.*
2. *Δεν επιτρέπονται περισσότερα του ενός ανοιχτά αρχεία από το `/dev/audio`. Προσπάθεια για δεύτερο άνοιγμά του πρέπει να επιστρέφει κωδικό λάθους `EBUSY` (Device or Resource Busy).*

Το υλικό προσφέρει μόνο δύο καταχωρητές προσβάσιμους από τον οδηγό:

- *Καταχωρητής `SND_HW_READY`, πλάτους `char`: Έχει τιμή `1` όταν η κάρτα ήχου μπορεί να δεχτεί δείγμα PCM προς αναπαραγωγή. Γίνεται `0` για όσο διάστημα η κάρτα ήχου είναι απασχολημένη.*
- *Καταχωρητής `SND_HW_PCM`, πλάτους `char`: Ο οδηγός γράφει εδώ ένα 8-bit δείγμα PCM προς αναπαραγωγή, αρκεί να ισχύει `SND_HW_READY == 1`.*

```
1   struct sound_dev_struct {
2       struct semaphore lock;
3       ...
4   } sound;
5
6   void hw_output_pcm(char val)
7   {
8       while ( !inb(SND_HW_READY))
9           ;
10      outb(val, SND_HW_PCM);
11  }
12
13  void hw_output_buffer(const char *buf, int len)
14  {
15      int i;
16
17      for (i = 0; i < len; i++)
18          hw_output_pcm(buf[i]);
19  }
20
21  int snd_chrdev_open(struct inode *inode, struct file *filp)
22  {
23          filp->private_data = &sound;
24          ...
25  }
26
27  static ssize_t snd_chrdev_write(struct file *filp, const char __user  *usrbuf,
28      size_t cnt, loff_t *f_pos)
29  {
30  #define WRITE_BUF_SZ 3072
31      char tmpbuf[WRITE_BUF_SZ];
32      struct sound_dev_struct *sd = filp->private_data;
33      ...
34      if (down_interruptible(&sd->lock))
35          return -ERESTARTSYS;
36      ...
37      hw_output_buffer(tmpbuf, n);
38      ...
39      up(&sd->lock);
40      ...
41      return cnt;
42  }
```

Σας δίνεται σκελετός αρχικής υλοποίησης του οδηγού, ο οποίος περιέχει:

- `inb(port)`, `outb(val, port)`:
  Μακροεντολές που χρησιμοποιούνται για πρόσβαση στους καταχωρητές υλικού. Κάνουν Ε/Ε από και προς το χώρο διευθύνσεων Ε/Ε ("ports") εκτελώντας απευθείας τις αντίστοιχες ειδικές εντολές του επεξεργαστή, π.χ. `IN`, `OUT` *για τον x86.*
- `hw_output_pcm(val)`, `hw_output_buffer(buf, len)`:
  Συναρτήσεις οι οποίες χρησιμοποιούν τις `inb()`, `outb()` *για να στείλουν στην κάρτα ήχου ένα δείγμα PCM κι έναν ολόκληρο απομονωτή με δείγματα PCM, αντίστοιχα.*
- `snd_chrdev_write(...)`:
  Η μέθοδος `write()` *του οδηγού συσκευής, η οποία βασίζεται στην* `hw_output_buffer()`.

Θεωρήστε ότι ο οδηγός εκτελείται σε σύστημα Linux όπου δεν θα υποστεί ποτέ context switch μια διεργασία όταν βρίσκεται σε κατάσταση πυρήνα (non-preemptible kernel).

Κάντε όποιες επεμβάσεις επιθυμείτε στο σκελετό, αρκεί να τις περιγράψετε με ακρίβεια. Ζητούνται τα εξής:

i. *(3%) Υλοποιήστε την `snd_chrdev_open()`. Πώς καλύπτει ο οδηγός σας την προδιαγραφή 2;*
ii. *(5%) Περιγράψτε σενάριο που οδηγεί στο να έχουν δύο διαφορετικές διεργασίες πρόσβαση σε ανοιχτό αρχείο από το `/dev/audio`, ταυτόχρονα. Πώς εξασφαλίζεται η ασφαλής χρήση του οδηγού σας αν κι οι δύο επιχειρήσουν `write()` και τι θα ακουστεί από τα ηχεία; Γιατί το πεδίο `lock` είναι τύπου `struct semaphore`;*
iii. *(3%) Σε τι κατάσταση βρίσκεται μια διεργασία ακριβώς όταν εκτελεί κλήση συστήματος `write()` στον οδηγό σας και από ποιες καταστάσεις περνάει μέχρι την επιστροφή της `write()`;*
iv. *(4%) Πόσο χρόνο χρειάζεται η `hw_output_buffer()` για να στείλει στην κάρτα ήχου έναν απομονωτή μήκους 33kB; Η υλοποίηση της `snd_chrdev_write()` με χρήση της `hw_output_buffer()` είναι ακατάλληλη για χρήση σε πραγματικό σύστημα, για να υποστηρίξει π.χ. διεργασία `mp3player`. Γιατί;*

i. Η υλοποίηση της `snd_chrdev_open()` ακολουθεί. Ο οδηγός καλύπτει την προδιαγραφή με μια σημαία `opened` στη συσκευή, η οποία τίθεται την πρώτη φορά που κάποια διεργασία ανοίγει τη συσκευή. Η σημαία προστατεύεται από το κλείδωμα `lock`, ως μέρος της δομής `struct sound_dev_struct`. Υποθέτουμε ότι η `snd_chrdev_release()` επαναφέρει τη σημαία στο `0` όταν απελευθερώνεται το αντίστοιχο `struct file`.
ii. Μια διεργασία ανοίγει το `/dev/audio` και μετά κάνει `fork()`. Αυτή και το παιδί της μοιράζονται το ίδιο ανοιχτό αρχείο (`struct file`). Ο οδηγός εξασφαλίζει ασφαλή χρήση κλειδώνοντας το κρίσιμο τμήμα (κλήση της `hw_output_buffer()` από την `write()`) με σημαφόρο `lock`. Το πεδίο `lock` είναι σημαφόρος γιατί μόνο κλήσεις της `write()` συναγωνίζονται γι'αυτό, οπότε κάθε προσπάθεια κλειδώματός του γίνεται από process context, οπότε ο καλών μπορεί να κοιμηθεί. Αν δύο διεργασίες επιχειρήσουν `write()` ταυτόχρονα, πρώτα θα ακουστεί ολόκληρος ο buffer της μίας, και μετά ο buffer της άλλης, γιατί μία από τις δύο θα μπλοκάρει προσπαθώντας να κλειδώσει το `lock`.
iii. Αρχικά είναι RUNNING, ακριβώς γιατί εκτελεί κλήση συστήματος. Μπορεί να πάει σε WAITING περιμένοντας να μπει στο κρίσιμο τμήμα (γρ. 34) ή κατά την `copy_from_user()` που υποθέτουμε πώς συμβαίνει για να γεμίσει τον τοπικό `tmpbuf[]`. Μέσα στο κρίσιμο τμήμα, η `hw_output_val()` εκτελεί busy waiting μέχρις ότου το υλικό είναι έτοιμο για να δεχτεί το επόμενο δείγμα PCM. Άρα η αλληλουχία `snd_chrdev_write()` →`hw_output_buffer()`→`hw_output_val()` οδηγεί στο να *παραμείνει* η διεργασία RUNNING, κάνοντας busy-waiting μέσα στον πυρήνα. Επειδή ο πυρήνας είναι non-preemptive, αποκλείεται να γίνει READY όσο είναι μέσα σε αυτόν.
iv. Ο buffer έχει μήκος 33.000 δείγματα. Υποθέτουμε ότι όταν ένα δείγμα δοθεί στην κάρτα ήχου (γρ. 10) αυτή μένει `SND_HW_READY == 0` για όσο χρόνο απαιτεί η αναπαραγωγή του, δηλαδή $1/22000$sec, διάστημα κατά το οποίο η `hw_output_buffer()` κάνει busy waiting. Άρα, χρειάζεται 1.5sec για να στείλει ολόκληρο τον απομονωτή. Η υλοποίηση είναι ακατάλληλη γιατί κρατάει κατειλημμένο τον επεξεργαστή εκτελώντας busy wait μέσα στον πυρήνα. Σε μονοεπεξεργαστικό σύστημα, αυτό θα σήμαινε ότι

για να μπορεί μια διεργασία να παίζει συνεχώς μουσική, θα έπρεπε να κρατάει το 100% του επεξεργαστή, χωρίς καμία άλλη να μπορεί να εκτελεστεί.

```
1  int snd_chrdev_open(struct inode *inode, struct file *filp)
2  {
3      struct sound_dev_struct *sd = &sound;
4
5      if (down_interruptible(&sd->lock))
6          return -ERESTARTSYS;
7      if (sd->opened) {
8          up(&sd->lock);
9          return -EBUSY;
10     }
11     sd->opened = 1;
12     up(&sd->lock);
13
14     filp->private_data = sd;
15
16     return 0;
17 }
```

Λόγω ακαταλληλότητας της υλοποίησης, η ομάδα σας αποφασίζει να προχωρήσει σε επέκταση του υλικού ώστε να υποστηρίζει διακοπές. *Η κάρτα ήχου προκαλεί διακοπή υλικού κάθε φορά που ο καταχωρητής `SND_HW_READY` μεταβαίνει `0` $\longrightarrow$ `1`. Επιπλέον, η σχεδίαση του οδηγού αλλάζει ώστε η υλοποίηση της μεθόδου `write()` να εκτελεί ενδιάμεση αποθήκευση των δεδομένων σε κυκλικό απομονωτή. Τα δεδομένα προωθούνται από τον κυκλικό απομονωτή στην κάρτα ήχου όταν χρειάζεται.*

```
1  struct sound_dev_struct {
2      struct semaphore lock;
3      uint128_t wcnt, rcnt;  /* TOTAL number of bytes read/written, from/to the
4                                circular buffer. They are initialized to zero,
5                                and will never wrap. */
6  #define CIRC_BUF_SIZE (1024 * 1024)
7      char circ_buffer[CIRC_BUF_SIZE];
8      ...
9  } sound;
10
11 static void snd_hw_intr(void)
12 {
13     struct sound_dev_struct *sd = &sound;
14     ...
15     if (...the circular buffer contains data...)
16         if (inb(SND_HW_READY))
17             hw_output_pcm(...)
18     ...
19 }
20
21 static ssize_t snd_chrdev_write(struct file *filp, const char __user  *usrbuf,
22     size_t cnt, loff_t *f_pos)
23 {
24     ...
25 }
26
```

Σας δίνεται βελτιωμένος σκελετός υλοποίησης του οδηγού, ο οποίος περιέχει συνάρτηση χειρισμού διακοπών υλικού `snd_hw_intr()` *και κυκλικό απομονωτή (circular buffer) με χρήση πεδίων* `rcnt`, `wcnt`, `circ_buf`.

Ζητούνται τα εξής:

v. *(5%) Υλοποιήστε την συνάρτηση χειρισμού διακοπών υλικού* `snd_hw_intr()`. *Κάντε όσες αλλαγές/προσθήκες χρειάζονται στη δομή* `sound_dev_struct` *ώστε να καλύπτεται η νέα σχεδίαση.*
vi. *(10%) Υλοποιήστε πλήρως τη μέθοδο* `snd_chrdev_write()`. *Στον οδηγό σας, όταν η κάρτα ήχου είναι ανενεργή, ποιο μέρος του προκαλεί την έναρξη της αναπαραγωγής ήχου;*
vii. *(5%) Σε τι κατάσταση βρίσκεται μια διεργασία ακριβώς όταν εκτελεί κλήση συστήματος* `write()` *στον οδηγό σας και από ποιες καταστάσεις περνάει μέχρι την επιστροφή της* `write()`*;*
viii. *(5%) Με ποιον τρόπο βοηθάει η υποστήριξη διακοπών την καλύτερη λειτουργία του οδηγού σας; Με τι ρυθμό γίνονται διακοπές υλικού;* Πολύ συνοπτικά, *προτείνετε πιθανή αλλαγή στο υλικό και στον οδηγό ώστε να βελτιωθεί κι άλλο η λειτουργία του συστήματος.*

Η γενική ιδέα είναι: Η `snd_chrdev_write()` αντιγράφει τα δεδομένα στον κυκλικό απομονωτή και επιστρέφει αμέσως. Στη συνέχεια ο χειριστής διακοπών μπορεί να τροφοδοτεί την κάρτα ήχου με αυτά, δείγμα-δείγμα, κάθε φορά που έχουμε μετάβαση `0` $\longrightarrow$`1` για τον καταχωρητή `SND_HW_READY`. Αν δεν υπάρχει διαθέσιμος χώρος στον απομονωτή, η `snd_chrdev_write()` κοιμίζει τη διεργασία μέχρις ότου δημιουργηθεί χώρος. Αυτό είναι η λογική επιλογή: *Σίγουρα* μια διεργασία μπορεί να παραγάγει δεδομένα ήχου πολύ γρηγορότερα απ'ότι η κάρτα ήχου τα επεξεργάζεται (22.000 bytes/sec), οπότε στο ενδιάμεσο η διεργασία είναι WAITING κι ο επεξεργαστής ελεύθερος. Οπότε:

v. Η υλοποίηση της `snd_hw_intr()` ακολουθεί. Αλλάζουμε το κλείδωμα σε spinlock, δεδομένου ότι και η `snd_chrdev_write()` και ο interrupt handler συναγωνίζονται για αυτό και προσθέτουμε δομή waitqueue, ώστε ο interrupt handler να μπορεί να ξυπνά τις διεργασίες που κοιμούνται περιμένοντας να απελευθερωθεί χώρος στον κυκλικό buffer.
vi. Η υλοποίηση της `snd_chrdev_write()` ακολουθεί. Όταν η κάρτα ήχου είναι ενεργή (παίζει κάποιο δείγμα, `SND_HW_READY == 0`), ο χειριστής διακοπών είναι η οντότητα που θα ενεργοποιηθεί όταν γίνει `SND_HW_READY == 1` για να την τροφοδοτήσει με το επόμενο δείγμα. Αρχικά όμως, η κάρτα είναι `SND_HW_READY == 1` και η `snd_chrdev_write()` είναι αυτή που γράφει το πρώτο δείγμα, για να ξεκινήσει τη διαδικασία.
vii. Όταν εκτελεί τη `write()` είναι RUNNING. Αν υπάρχει διαθέσιμος χώρος στον κυκλικό απομονωτή, παραμένει RUNNING, αντιγράφει τα δείγματα προς αναπαραγωγή κι επιστρέφει. Αν δεν υπάρχει, πηγαίνει σε WAITING (γρ. 68) μέχρι να απελευθερωθεί χώρος, οπότε την ξυπνά ο interrupt handler (γρ. 12), οπότε ξαναγίνεται RUNNING για να γράψει στον κυκλικό buffer.
viii. Δεδομένου ότι κάθε δείγμα χρειάζεται $1/22000$sec για την αναπαραγωγή του, η υποστήριξη διακοπών επιτρέπει στον επεξεργαστή να ασχολείται με άλλα πράγματα στο

ενδιάμεσο, αντί να κάνει busy wait πάνω στον καταχωρητή `SND_HW_READY`. Και πάλι ωστόσο, έχουμε μία διακοπή ανά δείγμα, 22.000 διακοπές/sec. Το πρόβλημα είναι ότι και πάλι πρέπει να εμπλέκεται ο επεξεργαστής για *κάθε* δείγμα, ώστε να το αντιγράφει στην κάρτα (`hw_output_val()`, Programmable I/O – PIO). Η σχεδίαση αυτή δεν μπορεί να κλιμακωθεί σε πολλές κάρτες και μεγαλύτερο ρυθμό δειγματοληψίας (ποιότητα CD). Η λύση είναι η ίδια η κάρτα να υποστηρίζει να διαβάζει απευθείας τα δείγματα από τη μνήμη (Direct Memory Access - DMA), οπότε θα προκαλεί μόνο μία διακοπή όταν έχει εξαντλήσει έναν ολόκληρο buffer, π.χ. 64kB.

```
1   static void snd_hw_intr(void)
2   {
3   #define BYTES_IN_BUFFER(sd)    ((sd)->wcnt - (sd)->rcnt)
4       struct sound_dev_struct *sd = &sound;
5       unsigned long flags;
6
7       spin_lock_irqsave(&sd->lock, flags);
8       if (BYTES_IN_BUFFER(sd)) { /* buffer contains data */
9           if (inb(SND_HW_READY)) {
10              hw_output_pcm(sd->circ_buffer[sd->rcnt % CIRC_BUF_SIZE]);
11              ++sd->rcnt;
12              wake_up_interruptible(&sd->wq);
13          } else {
14              /*
15               * lost the race with write(), the sound card will cause
16               * another interrupt when SND_HW_READY becomes 1 again.
17               */
18          }
19       }
20      spin_lock_irqrestore(&sd->lock, flags);
21  }
22
23  static ssize_t snd_chrdev_write(struct file *filp, const char __user  *usrbuf,
24      size_t cnt, loff_t *f_pos)
25  {
26  #define WRITE_BUF_SZ 3072
27  #define FREE_BYTES_IN_BUFFER(sd)    (CIRC_BUF_SIZE - BYTES_IN_BUFFER(sd))
28
29      size_t i, free;
30      char tmpbuf[WRITE_BUF_SZ];
31      unsigned long flags;
32      struct sound_dev_struct *sd = filp->private_data;
33
34      /* Bring up to WRITE_BUF_SZ bytes into the buffer on the stack */
35      if (cnt > WRITE_BUF_SZ)
36          cnt = WRITE_BUF_SZ;
37      if (copy_from_user(tmpbuf, userbuf, cnt))
38          return -EFAULT;
39
40      /* Lock the device structure, copy data into the circular buffer */
41  retry:
42      spin_lock_irqave(&sd->lock, flags);
43      /* Got room in the buffer? */
44      free = FREE_BYTES_IN_BUFFER(sd);
45      if (!free)
46          goto wait_retry;
47
```

```
48        if (cnt > free)
49            cnt = free;
50
51        for (i = 0; i < cnt; i++) {
52            sd->circ_buffer[sd->wcnt % CIRC_BUF_SIZE] = tmpbuf[i];
53            ++sd->wcnt;
54        }
55
56        /* If the sound card is not already playing, start it */
57        if (inb(SND_HW_READY)) {
58            hw_output_pcm(sd->circ_buffer[sd->rcnt % CIRC_BUF_SIZE]);
59            ++sd->rcnt;
60            wake_up_interruptible(&sd->wq);
61        }
62        spin_unlock_irqrestore(&sd->lock, flags);
63
64        return i;
65
66    wait_retry:
67        spin_unlock_irqrestore(&sd->lock, flags);
68        if (wait_event_interruptible(sd->wq, FREE_BYTES_IN_BUFFER(sd)))
69            return -ERESTARTSYS;
70        goto retry;
71    }
```

Σχετικοί σύνδεσμοι:
Το υλικό που περιγράφεται είναι πολύ κοντά στην πραγματική κάρτα ήχου Sound Blaster, η οποία κυριαρχούσε στην αγορά στις αρχές της δεκαετίας του '90. Ακόμη και σήμερα, ο πυρήνας του Linux περιέχει οδηγό συσκευής για 8-bit SB. Δείτε:

- Wikipedia:
  `https://en.wikipedia.org/wiki/Sound_Blaster`
- Οδηγίες προγραμματισμού, που περιλαμβάνουν busy waiting, interrupt-based PIO, και DMA:
  `http://the.earth.li/~tfm/oldpage/sb_wave.html`
- Οδηγός 8-bit Sound Blaster στο Linux:
  `http://lxr.free-electrons.com/source/sound/isa/sb/sb8_main.c`
- Χαμηλού επιπέδου περιγραφή των I/O ports στον x86:
  `http://www.cs.mun.ca/~paul/cs3725/material/web/notes/node28.html`

Θέμα 2 (35%)

Συμμετέχετε σε ομάδα που σχεδιάζει και υλοποιεί ένα εξειδικευμένο υποσύστημα αποδοτικής επικοινωνίας ανάμεσα σε διεργασίες που εκτελούνται σε διαφορετικές εικονικές μηχανές (VMs), επάνω στο ίδιο φυσικό μηχάνημα. Οι απαιτήσεις αναφέρουν ρητά ότι το μοναδικό σενάριο χρήσης του συστήματος αφορά ακριβώς δύο εικονικές μηχανές (στο ίδιο φυσικό μηχάνημα) και ακριβώς δύο διεργασίες (μία διεργασία σε κάθε εικονικό μηχάνημα), οι οποίες επικοινωνούν μέσω αμφίδρομου, αξιόπιστου ρεύματος χαρακτήρων που παρέχεται από τον μηχανισμό σας.

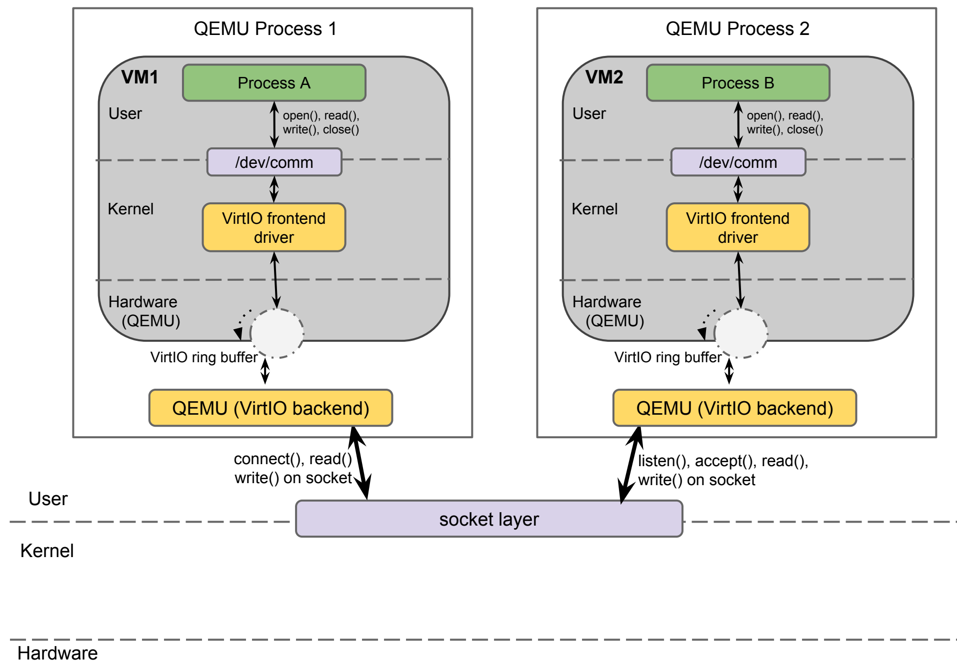
Η πλατφόρμα εικονικοποίησης που θα χρησιμοποιήσετε είναι QEMU/KVM σε ΛΣ Linux.

Στο σύστημα Εισόδου/Εξόδου θα χρησιμοποιήσετε το VirtIO split-driver model (frontend-backend). Οι διεργασίες μέσα στα VMs χρησιμοποιούν κλήσεις `open()`, `read()`, `write()`, `close()` για τη χρήση του μηχανισμού. Οι εικονικές μηχανές QEMU χρησιμοποιούν UNIX domain sockets για διαδιεργασιακή επικοινωνία.

Απαντήστε στα ακόλουθα:

i. *(10%) Περιγράψτε συνοπτικά την αρχιτεκτονική που προτείνετε. Σχεδιάστε ένα σχήμα στο οποίο θα φαίνονται τα διαφορετικά κομμάτια του συστήματός σας και η κατανομή τους στο λογισμικό και στο υλικό του φυσικού και των εικονικών μηχανημάτων. Δείξτε την επικοινωνία μεταξύ τους.*
ii. *(10%) Περιγράψτε με ακρίβεια ένα παράδειγμα χρήσης του συστήματός σας, για την αποστολή ενός μηνύματος (π.χ., "ping") από τη μία διεργασία στην άλλη. Περιγράψτε την αλληλουχία κλήσεων, όλα τα διαφορετικά στρώματα από τα οποία περνάει το μήνυμα και τον τρόπο επικοινωνίας ανάμεσά τους, στο εικονικό και στο φυσικό μηχάνημα. Πώς ενεργοποιείται το επόμενο τμήμα του μονοπατιού σε κάθε περίπτωση;*
iii. *(5%) Έστω ότι η διεργασία A του εικονικού μηχανήματος VM1 στέλνει ένα block δεδομένων στη διεργασία B του εικονικού μηχανήματος VM2. Πόσες αντιγραφές αυτού του block θα γίνουν σε ολόκληρο το μονοπάτι A⟶B και από ποιο μέρος σε ποιο, κάθε φορά;*
iv. *(5%) Ενώ βρίσκεστε στη φάση παράδοσης και έχετε ήδη υλοποιήσει το σύστημα και τις διεργασίες ελέγχου (test cases), σας ειδοποιούν επειγόντως ότι άλλαξε μια βασική απαίτηση του συστήματος: οι εικονικές μηχανές θα βρίσκονται σε δύο διαφορετικά φυσικά μηχανήματα, τα οποία συνδέονται δικτυακά μεταξύ τους. Χρειάζεται να κάνετε αλλαγές στην αρχιτεκτονική του συστήματός σας και γιατί; Αν ναι, ποιες είναι αυτές και ποια μέρη του συστήματος θα μείνουν ανεπηρέαστα; Τα test cases χρειάζονται αλλαγή και γιατί;*
v. *(5%) Στην περίπτωση όπου τα VMs είναι σε διαφορετικά φυσικά μηχανήματα και ανταλλάσσονται ευαίσθητα δεδομένα ανάμεσα στις διεργασίες, προκύπτει η ανάγκη για κρυπτογράφησή τους. Σε ποιο/α μέρος/η του συστήματος θα προσθέτατε τη λειτουργικότητα αυτή και πώς επηρεάζονται τα test cases;*

i. Στο ακόλουθο σχήμα φαίνεται η προτεινόμενη αρχιτεκτονική.
   Το σύστημα υλοποιείται ως οδηγός συσκευής στον πυρήνα της εικονικής μηχανής ("frontend") και στο hardware της εικονικής μηχανής, μέσα στη διεργασία του QEMU, στο χώρο χρήστη του φυσικού μηχανήματος ("backend"). Τα μέρη frontend, backend μπορούν να ανταλλάσσουν δεδομένα μέσω ουράς σε μνήμη μοιραζόμενη ανάμεσά τους ("virtqueue"). Τα δύο backends επικοινωνούν μεταξύ τους μέσω UNIX domain sockets. Η χρήση τους εμπλέκει τον πυρήνα του φυσικού μηχανήματος, καθώς από την πλευρά του δύο διεργασίες QEMU επικοινωνούν μέσω UNIX domain sockets εκτελώντας κλήσεις συστήματος `connect()`, `listen()`, `bind()`, `accept()`, `read()`, `write()`, `close()`.
ii. Στα επόμενα υποθέτουμε ότι η διεργασία A στέλνει σύντομο μήνυμα, π.χ. "ping" στη B. Υποθέτουμε ότι το VM2 αρχικοποιεί το UNIX domain socket και είναι έτοιμο να δεχτεί συνδέσεις (`listen()`, `accept()`), ενώ το VM1 συνδέεται σε αυτό (`connect()`). Αρχικά, η B ανοίγει με `open("/dev/comm", ...)` το ειδικό αρχείο της συσκευής μέσα



στο VM. Ο οδηγός (frontend) περνά κατάλληλο μήνυμα ελέγχου στο backend με το οποίο ζητά την αρχικοποίηση της επικοινωνίας, μέσω της virtqueue. Η B μπλοκάρει (WAITING) μέσα στην `open()`\. Το backend αρχικοποιεί το UNIX domain socket (`socket()`, `bind()`, `listen()`) και μπλοκάρει μέχρι να συνδεθεί το VM1 εκεί. Ομοίως, η διεργασία A ανοίγει το ειδικό αρχείο του οδηγού μέσα στο VM, ο οδηγός περνά αντίστοιχο μήνυμα ελέγχου στο backend του VM1 και το VM1 συνδέεται στο UNIX domain socket του VM2 (`connect()`. Όταν αυτό συμβεί, η `connect()` του VM1 επιστρέφει, η `accept()` του VM2 επιστρέφει και τα δύο backends ενημερώνουν τα αντίστοιχα frontends μέσω της virtqueue (virtual interrupt στο VM), οπότε οι A, B γίνονται READY και οι κλήσεις `open()` τελικά επιστρέφουν. Για την αποστολή του μηνύματος, η A εκτελεί `write(fd, "ping", 4)`, το frontend γράφει το μήνυμα στη virtqueue, ενεργοποιεί το backend του VM1, το οποίο κάνει `write()` στο socket. Αυτό ενεργοποιεί το "ξεμπλοκάρισμα" της αντίστοιχης κλήσης `read()` του VM2, το backend γράφει τα δεδομένα στη virtqueue του VM2, προκαλεί virtual interrupt, το frontend το χειρίζεται και αντιγράφει τα εισερχόμενα δεδομένα σε kernelspace buffer στο VM2. Αν η διεργασία B έχει ήδη μπλοκάρει εκτελώντας `read(fd, ...)`, το frontend την κάνει READY ώστε να επεξεργαστεί το εισερχόμενο μήνυμα.
Στα προηγούμενα υποθέτουμε ότι η υλοποίηση του backend στο VM χρησιμοποιεί χωριστό νήμα το οποίο μπλοκάρει σε κλήση `read()` από το UNIX domain socket.
Ακολουθούν τα τμήματα του μονοπατιού A⟶B. Ο τρόπος ενεργοποίησης του επόμενου τμήματος σημειώνεται μέσα στις παρενθέσεις: i Διεργασία A → frontend VM1 (system call), ii frontend VM1 → backend VM1, μέσω virtqueue (`virtqueue_kick()`, εγγραφή σε memory-mapped I/O space που προκαλεί trap στο QEMU), iii backend VM1 → πυρήνας host (`write()` στο UNIX domain socket), iv πυρήνας host → backend VM2 (ξύπνημα νήματος QEMU από πυρήνα host), v backend VM2 → frontend VM2 (σήμα προς QEMU thread, εικονικό interrupt μέσα στο VM), vi frontend VM2 → διεργασία B (ξύπνημα διεργασίας B από πυρήνα guest).

iii. Στο πρώτο μισό της διαδρομής γίνεται ένα αντίγραφο από userspace της A στο ker-

nelspace του VM1 και ένα αντίγραφο από τη virtqueue του VM1 (userspace του QEMU process της VM1) προς το kernelspace του host (buffers του UNIX domain socket). Στο μονοπάτι από το frontend (virtqueue) στο backend δε γίνεται αντιγραφή, αλλά περνιέται η διεύθυνση του buffer, καθώς η διεργασία QEMU έχει απευθείας πρόσβαση στις αντίστοιχες θέσεις μνήμης. Στο δεύτερο μισό της διαδρομής (VM2) γίνονται τα ίδια αντίγραφα, σε αντίστροφη σειρά.

iv. Αποφασίζουμε να χρησιμοποιήσουμε TCP/IP sockets αντί για UNIX domain sockets για την επικοινωνία ανάμεσα στα δύο backends. Οι απαιτούμενες αλλαγές περιορίζονται μόνο στον κώδικα του backend, το οποίο πλέον αρχικοποιεί `PF_INET` sockets αντί για `PF_UNIX` sockets και διευθύνσεις TCPv4 (διεύθυνση IPv4, αριθμός θύρας TCP) αντί για ονόματα αρχείων στο τοπικό filesystem. Ο κώδικας του frontend και των διεργασιών παραμένει ίδιος.

v. Ομοίως, επεκτείνουμε τον κώδικα του backend ώστε να κρυπτογραφεί και να αποκρυπτογραφεί τα δεδομένα πριν και μετά το πέρασμά τους από τη σύνδεση TCP, αντίστοιχα. Ο κώδικας που εκτελείται μέσα στο VM (frontend, διεργασίες) δεν ξέρει τίποτε για την αλλαγή και παραμένει ίδιος. Η επικοινωνία backend $\rightarrow$ frontend και frontend $\rightarrow$ διεργασίας δεν επηρεάζεται από την αλλαγή του υφιστάμενου τρόπου επικοινωνίας.

Θέμα 3 (25%)

α. (15%) *Ένας φίλος σας, σας τηλεφωνεί γιατί το παιχνίδι `pongman` στο σύστημα του έχει ”κρεμάσει” (hang), δηλαδή δεν αποκρίνεται πλέον σε είσοδο από το χρήστη (ποντίκι, πληκτρολόγιο). Το σύστημα έχει έναν επεξεργαστή και τρέχει Linux.*

Απαντήστε στα εξής:

i. *(3%) Με την καθοδήγησή σας, εκτελεί την εντολή `strace -p 1234`, όπου `1234` το PID του `pongman`. Η εντολή δεν παράγει τίποτε στην έξοδό της. Τι σημαίνει αυτό για τη διεργασία `1234`;* Υπόδειξη: *Θεωρήστε ότι κατά την εκκίνησή της η `strace` εκτυπώνει την κλήση συστήματος μέσα στην οποία μπορεί να βρίσκεται η διεργασία.*

ii. *(3%) Σε τι κατάσταση (process state) βρίσκεται η διεργασία `1234`; Πόσο είναι το CPU load του μηχανήματος;*

iii. *(2%) Δώστε* σύντομο *ενδεικτικό κώδικα σε Assembly που θα μπορούσε να έχει συμπεριφορά ανάλογη με αυτή του `pongman` εκείνη τη στιγμή. Θα μπορούσε η δυσλειτουργία της διεργασίας `1234` να κάνει μη-αποκρίσιμο ολόκληρο το σύστημα; Γιατί;*

i. Η `strace` δεν παράγει τίποτε στην έξοδό της, γιατί η διεργασία `1234` δεν ήταν μέσα σε system call όταν ξεκίνησε η `strace` και δεν εκτελεί κανένα system call. Αυτό σημαίνει ότι η διεργασία εκτελεί αποκλειστικά υπολογισμό και δεν τελειώνει ποτέ το CPU burst. Πιθανότατα έχει bug και έχει κολλήσει σε κάποιο infinite loop.
Σχετικός σύνδεσμος:
`https://serverfault.com/questions/190643/question-about-no-output-from-strace`, με την διευκρίνηση ότι η άσκηση θεωρεί ότι η `strace` δείχνει αρχικά και το τρέχον system call.

ii. Η διεργασία εκτελεί CPU burst, η κατάστασή της εναλλάσσεται από RUNNING σε READY και αντίστροφα. Το CPU load είναι 100%.
iii. Ως προς το ΛΣ, η διεργασία εκτελεί το ισοδύναμο του `L1: JMP L1`. Το infinite loop στο οποίο έχει πέσει η `1234` δεν επηρεάζει το υπόλοιπο σύστημα γιατί η `1234` διακόπτεται κανονικά από timer interrupts, τρέχει ο χρονοδρομολογητής κι επιτρέπει σε άλλες διεργασίες να τρέξουν.

Μια άλλη μέρα:

iv. *(2%) Ο φίλος σας παρατηρεί ότι το pongman, ενώ λειτουργεί κανονικά, εμφανίζει μικρές καθυστερήσεις κατά την εκτέλεσή του (lag), γιατί παράλληλα τρέχει ένα πρόγραμμα συμπίεσης σε MP3, μια υπολογιστικά απαιτητική διαδικασία. Γιατί έχει lag το pongman;*
v. *(5%) Για να διορθώσει την κατάσταση, ο φίλος σας αναθέτει τη μέγιστη δυνατή απόλυτη προτεραιότητα χρονοδρομολόγησης στη διεργασία του pongman. Τι θα συμβεί αν η διεργασία δυσλειτουργήσει όπως στα (i) – (iii); Τι πρέπει να έχει κάνει ο φίλος σας για να μπορεί να αντιμετωπίσει αυτό το ενδεχόμενο;*

iv. Το pongman έχει lag γιατί συναγωνίζεται για χρόνο στη CPU με τα νήματα του MP3 encoder, που είναι μονίμως σε CPU burst, οπότε μπορεί να πρέπει να περιμένει αρκετά κβάντα χρόνου για να τρέξει.
v. Αν το pongman έχει μέγιστη απόλυτη προτεραιότητα, καμία άλλη διεργασία δεν μπορεί να τη διακόψει, συμπεριλαμβανομένου του φλοιού του χρήστη, οπότε όλο το σύστημα θα είναι μη-αποκρίσιμο. Μια λύση θα ήταν ο φίλος σας να κρατά *πάντοτε* έναν φλοιό με μεγαλύτερη προτεραιότητα από κάθε άλλη διεργασία, ώστε να έχει τον έλεγο (π.χ. να τη σκοτώσει) αν χρειαστεί.
Σχετικός σύνδεσμος:
`https://www.gnu.org/software/libc/manual/html_node/Absolute-Priority.html`

β. (10%) *Εκτελούμε την εντολή*
`ls -l | grep string > file1`
στο φλοιό του UNIX.

Απαντήστε στα εξής:

i. *(2%) Σε ποιον περιγραφητή αρχείου γράφει η διεργασία του `ls`; Από ποιον περιγραφητή αρχείου διαβάζει η διεργασία του `grep`;*
ii. *(3%) Ποια διεργασία ανοίγει το αρχείο `file1` και με ποιες παραμέτρους;*
iii. *(5%) Μια δική σας διεργασία επιθυμεί να εκτελέσει την εντολή `ls` και να αποκτήσει τη συνολική έξοδό της στη μνήμη της, π.χ., σε απομονωτή `buf[]`. Περιγράψτε την αλληλουχία κλήσεων συστήματος για να το επιτύχει αυτό, χωρίς κανενός είδους ενδιάμεση αποθήκευση στο σύστημα αρχείων. Αν σας εξυπηρετεί, δώστε ψευδοκώδικα.*

i. Η διεργασία του `ls` γράφει στον περιγραφητή `1` (καθιερωμένη έξοδος) κι η διεργασία του `grep` διαβάζει από τον περιγραφητή `0` (καθιερωμένη είσοδος).

ii. Η διεργασία του φλοιού ανοίγει το `file1`, αφού κάνει ανάλυση της γραμμής εντολών και δει ότι ο χρήστης ζήτησε ανακατεύθυνση της εξόδου του `grep` στο αρχείο αυτό. Εκτελεί την κλήση `open()` με παραμέτρους `flags == O_WRONLY | O_TRUNC | O_CREAT` και `mode == 0666`.

iii. Η διεργασία δημιουργεί ένα pipe (`pipe()`) και γεννά ένα παιδί (`fork()`). Στο παιδί, ανακατευθύνει (`dup2()`) τον περιγραφητή `1` ώστε να δείχνει στο ανοιχτό άκρο εγγραφής του pipe και στη συνέχεια εκτελεί την `ls`, με `execve()`. Η `ls` γράφει μέσα στο pipe, ενώ ο πατέρας εκτελεί επαναλαμβανόμενα `read()` από το άκρο ανάγνωσης ώστε να λαμβάνει τα δεδομένα εξόδου της και να τα τοποθετεί στον `buf[]`.

Για πληρέστερη κατανόηση της απάντησης, ακολουθεί πλήρες πρόγραμμα στο UNIX που ικανοποιεί το ζητούμενο. Για απλότητα, το πρόγραμμα θεωρεί ότι η έξοδος της ls δεν θα ξεπερνά το 1KB.

```
1   #include <unistd.h>
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <errno.h>
5
6   void die(char *str)
7   {
8       perror(str);
9       exit(1);
10  }
11
12  int main(void)
13  {
14      pid_t pid;
15      int pd[2];
16      size_t n, total, rem;
17      char *p, buf[1024]; /* Hardcoded limits are ugly. */
18      char *argv[] = { "/bin/ls", NULL };
19      char *envp[] = { NULL };
20
21      if (pipe(pd) < 0)
22          die("pipe");
23
24      pid = fork();
25      if (pid < 0)
26          die("fork");
27      if (pid == 0) {
28          close(pd[0]);
29          if (dup2(pd[1], 1) < 0)
30              die("dup2");
31          execve("/bin/ls", argv, envp);
32          die("execve(\"/bin/ls\") failed");
33      }
34
35      /* Parent */
36      close(pd[1]);
37      p = buf;
38      rem = sizeof(buf);
39      n = total = 0;
40      while (rem) {
41          n = read(pd[0], p, rem);
```

```
42          if (n < 0)
43              die("read");
44          if (n == 0)
45              break;
46          p += n;
47          total += n;
48          rem -= n;
49      }
50      if (!rem) {
51          close(pd[0]); /* This causes delivery of SIGPIPE to child */
52          fprintf(stderr, "Out of space in buf[]\n");
53      }
54      wait(NULL);
55      fprintf(stderr, "Output of /bin/ls follows:\n");
56      n = write(1, buf, total);
57      if (n < 0)
58          die("write");
59      if (n != total)
60        fprintf(stderr, "Will not handle partial write, %d != %d, use insist_write().\n",
61              n, total);
62
63      return 0;
64  }
```