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 2011-2012

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%)

Το πείραμα ATLAS είναι ένα από τα πειράματα που εκτελούνται στον Μεγάλο Επιταχυντή Αδρονίων, LHC. Καθώς δέσμες σωματιδίων συγκρούονται, ανιχνευτές παράγουν τεράστιο όγκο δεδομένων.

Θεωρήστε πειραματική διάταξη υλικού και αντίστοιχου οδηγού συσκευής στο Linux. Το υλικό αποτελείται από τους ανιχνευτές (`0` έως `detector_cnt - 1`) και μια διάταξη περιοδικής παραγωγής διακοπών υλικού (διακοπές `SAMPLER`). Με κάθε διακοπή `SAMPLER`, ο οδηγός συλλέγει τις μετρήσεις όλων των ανιχνευτών και τις καταγράφει σε δικό του κυκλικό απομονωτή στη μνήμη. Διεργασίες χώρου χρήστη απορροφούν τα δεδομένα αυτά μέσω συσκευής χαρακτήρων `/dev/atlas-device` για περαιτέρω ανάλυση και αρχειοθέτηση.

Μας ενδιαφέρουν μετρήσεις μόνο κατά τη διάρκεια επιλεγμένων γεγονότων *(events), π.χ. συγκρούσεων με συγκεκριμένα χαρακτηριστικά, οπότε είναι πιθανό να έχει δημιουργηθεί ένα ως τώρα άγνωστο σωματίδιο. Επομένως, ξεκινάμε και σταματάμε την καταγραφή και απορρόφηση των δεδομένων ώστε να συμπίπτουν με τα γεγονότα.*

Ένας από τους ανιχνευτές (`trigger_detector_id`) έχει ειδικό ρόλο: κάθε φορά που ανανεώνεται η τιμή του, το υλικό προκαλεί διακοπή `TRIGGER`. Όταν η τιμή αυτή ξεπεράσει κάποιο όριο, ο οδηγός θεωρεί ότι γεγονός είναι σε εξέλιξη, και ξεκινά την καταγραφή κι απορρόφηση. Η καταγραφή συνεχίζεται μέχρι το τέλος του γεγονότος ή την πλήρωση του κυκλικού απομονωτή γιατί οι διεργασίες δεν τον πρόλαβαν.

Από την πλευρά των διεργασιών, κάθε `read()` επιστρέφει bytes για ακέραιο *αριθμό (όχι κατ'ανάγκη διαδοχικών) μετρήσεων τύπου `struct measurement` από το ρεύμα των εισερχόμενων δεδομένων. Αν δεν υπάρχουν δεδομένα, η `read()` μπλοκάρει.*

Το ρεύμα τερματίζεται όταν ο οδηγός σταματήσει την καταγραφή κι ο απομονωτής αδειάσει. Στην περίπτωση αυτή, το αμέσως επόμενο `read()` από οποιονδήποτε επιστρέφει `EOF` και τα υπόλοιπα μπλοκάρουν έως την έναρξη του επόμενου γεγονότος.

Δίνονται τα εξής:

- *`get_hw_measurement(dev, detectorid, msr):`*
  Διαβάζει τη μέτρηση του ανιχνευτή `detectorid` της συσκευής `dev` στη δομή `msr`.
- `event_in_progress(triggermsr)`:
  Επιστρέφει αληθές αν η μέτρηση `triggermsr` του ανιχνευτή `trigger_detector_id` δείχνει ότι γεγονός είναι σε εξέλιξη.
- `start_recording(dev)`:
  Ενεργοποιεί διακοπές `SAMPLER` και περιοδική καταγραφή εισερχομένων μετρήσεων στον κυκλικό απομονωτή για τη συσκευή.
- `stop_recording(dev)`:
  Απενεργοποιεί διακοπές `SAMPLER` και σταματά την καταγραφή μετρήσεων.
- `is_recording(dev)`:
  Επιστρέφει αληθές αν γίνεται καταγραφή δεδομένων.
- `store_measurement(dev, msr)`:
  *Αποθηκεύει τη μέτρηση `msr` στον κυκλικό απομονωτή για τη συσκευή `dev`. Επιστρέφει αληθές αν ο απομονωτής έχει γεμίσει (*overflow*).*
- `retrieve_measurement(dev, msr)`:
  Διαβάζει από τον κυκλικό απομονωτή της συσκευής `dev` μια νέα μέτρηση `msr`, επιστρέφει ψευδές αν δεν υπάρχουν έγκυρα δεδομένα.
- `intr(interrupt_mask)`:
  Εξυπηρετεί τις διακοπές υλικού της συσκευής. Στο όρισμα `interrupt_mask` δίνονται οι διακοπές οι οποίες συνέβησαν με τη μορφή αληθών bit.

Όλες οι συναρτήσεις που δέχονται όρισμα τον περιγραφητή της συσκευής `dev`, την κλειδώνουν για να εκτελέσουν τη λειτουργία τους και την απελευθερώνουν.

```
1  struct measurement { ...  }; /* 128 bytes */
2
3  struct atlas_device {
4      ..locktype.. lock;
5      int detector_cnt, detector_trigger_id;
6      int recording;
7      wait_queue_head_t wq;
8      uint128_t wcnt, rcnt; /* These are initialized to zero, and
9                             * will never wrap. */
10     . . .
11 } atlas;
12
13 int store_measurement(struct atlas_device *dev, struct measurement *msr);
14 int retrieve_measurement(struct atlas_device *dev, struct measurement *msr)
15 void get_hw_measurement(struct atlas_device *dev, uint32_t detectorid,
16                         struct measurement *msr);
17 unsigned int get_hw_interrupt_mask(struct atlas_device *dev);
```

```
18  void set_hw_interrupt_mask(struct atlas_device *dev, unsigned int mask);
19  int event_in_progress(struct measurement *triggermsr);
20  int is_recording(struct atlas_device *dev);
21  void start_recording(struct atlas_device *dev);
22
23  void stop_recording(struct atlas_device *dev)
24  {
25      unsigned long flags;
26      unsigned int mask;
27
28      ... lock dev ...
29      dev->recording = 0;
30      . . .
31
32      /* Disable periodic sampling interrupts */
33      mask = get_hw_interrupt_mask(dev);
34      set_hw_interrupt_mask(dev, mask & ~(1 << INTR_SAMPLER_SHIFT));
35      wake_up_interruptible(&dev->wq);
36      ... unlock dev ...
37  }
38
39  void intr(unsigned int mask)
40  {
41      int i, overflow;
42      struct measurement msr;
43      struct atlas_device *dev = &atlas;
44
45      if (mask & (1 << INTR_TRIGGER_SHIFT)) {
46          get_hw_measurement(dev, dev->detector_trigger_id, &msr);
47          if (event_in_progress(&msr) && !is_recording(dev))
48              start_recording(dev);
49          if (!event_in_progress(&msr) && recording(dev))
50              stop_recording(dev);
51      }
52
53      if (mask & (1 << INTR_SAMPLER_SHIFT)) {
54          for (i = 0; i < dev->detector_cnt; i++) {
55              get_hw_measurement(dev, i, &msr);
56              overflow = store_measurement(dev, &msr);
57              if (overflow) {
58                  stop_recording(dev);
59                  return;
60              }
61          }
62          wake_up_interruptible(&dev->wq);
63      }
64  }
65
66  ssize_t atlas_chrdev_read(struct file *filp, char __user *usrbuf,
67      size_t cnt, loff_t *f_pos)
68  {
69      struct atlas_device *dev = filp->private_data;
70
71      /* Only return whole measurements */
72      . . .
73
74      /* Retrieve measurements, block if no data available,
75       * return EOF on event termination or buffer overflow. */
76
77      /* Eventually, i measurements are being returned */
78      return i * sizeof(msr);
79  }
```

Ζητούνται τα εξής:

i. *(3%) Ποιος ο τύπος κι ο ρόλος του πεδίου `lock`;*
ii. *(2%) Ποιος ο ρόλος του πεδίου `wq`;*
iii. *(25%) Υλοποιήστε την `atlas_chrdev_read()`, συμπληρώνοντας τον σκελετό.*
iv. *(10%) Υλοποιήστε την `start_recording()`, συμπληρώνοντας τον σκελετό.*

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

i. Το πεδίο `lock` είναι κλείδωμα που προστατεύει τη δομή `atlas_device` από ταυτόχρονη πρόσβαση. Ο τύπος του πρέπει να είναι `spinlock_t`, γιατί το κλειδώνει τόσο κώδικας που τρέχει σε process context, π.χ. `retrieve_measurement()` όσο και σε interrupt context, π.χ. `intr()`→`stop_recording()`).
ii. Το πεδίο `wq` είναι μια ουρά αναμονής. Τοποθετούνται εκεί διεργασίες οι οποίες εκτελούν μετάβαση RUNNING→WAITING και περιμένουν δεδομένα από το `/dev/atlas`, π.χ. μπλοκάρουν σε `read()` περιμένοντας να ξεκινήσει η καταγραφή δεδομένων.
iii. Ο απλούστερος τρόπος υλοποίησης της `atlas_chrdev_read()` είναι χρησιμοποιώντας την `retrieve_measurement()` που μας δίνεται, χωρίς ανάγκη για κλείδωμα της συσκευής, αφού η `retrieve_measurement()` φροντίζει να την κλειδώνει κατά την προσπέλαση:

```
1  ssize_t atlas_chrdev_read(struct file *filp, char __user *usrbuf,
2      size_t cnt, loff_t *f_pos)
3  {
4    struct atlas_device *dev = filp->private_data;
5    int i, msrcnt, valid, will_eof;
6    struct measurement msr;
7    unsigned long flags;
8
9    /* Only return whole measurements, return at least ONE measurement */
10   msrcnt = cnt / sizeof(struct measurement);
11   if (msrcnt == 0)
12     return -EINVAL;
13
14   for (i = 0; i < msrcnt; i++) {
15     valid = retrieve_measurement(dev, &msr);
16     if (i == 0 && !valid) {
17       if (!is_recording(dev)) {
18         spin_lock_irqsave(&dev->lock, flags);
19         if ((will_eof = dev->at_eof))
20           dev->at_eof = 0;
21         spin_unlock_irqrestore(&dev->lock, flags);
22         if (will_eof)
23           return 0;
24       }
25       if (wait_event_interruptible(dev->wq,
26         !is_recording(dev) || (valid = retrieve_measurement(dev, &msr))))
```

```
27          return -ERESTARTSYS;
28      }
29      if (!valid)
30        break;
31      if (copy_to_user(usrbuf + i * sizeof(msr), &msr, sizeof(msr)))
32        return -EFAULT;
33    }
34    /* Eventually, i measurements are being returned */
35    return i * sizeof(msr);
36  }
```

Εμπλουτίζουμε τη δομή `struct atlas_device` με το flag `at_eof`, το οποίο τίθεται από την `stop_recording()` όταν σταματήσει η καταγραφή. Με τον τρόπο αυτό εξασφαλίζουμε ότι το EOF είναι άλλο ένα δεδομένο που καταναλώνεται από την `atlas_chrdev_read()`.

iv. Η `start_recording()` κλειδώνει τη συσκευή, θέτει τη σημαία `recording`, ενεργοποιεί τις διακοπές `SAMPLER` από το υλικό και ξυπνά όσες διεργασίες περιμένουν στην ουρά αναμονής `wq` για να λάβουν δεδομένα:

```
1   void start_recording(struct atlas_device *dev)
2   {
3       unsigned long flags;
4       unsigned int mask;
5
6       spin_lock_irqsave(&dev->lock, flags);
7       dev->recording = 1;
8
9       /* Enable periodic sampling interrupts */
10      mask = get_hw_interrupt_mask(dev);
11      set_hw_interrupt_mask(dev, mask | (1 << INTR_SAMPLER_SHIFT));
12
13      wake_up_interruptible(&dev->wq);
14      spin_unlock_irqrestore(&dev->lock, flags);
15  }
16
```

Θέμα 2 (30%)

Θέλουμε να υλοποιήσουμε μια κατάρα Αναγνωριστή [Α], η οποία παρακολουθεί την αλληλουχία των κλήσεων συστήματος μιας διεργασίας. Εάν η αλληλουχία αυτή ανταποκρίνεται σε μία προκαθορισμένη πρότυπη ακολουθία [Π], ενεργοποιεί ("οπλίζει") μια άλλη κατάρα [Κ] για τη διεργασία.

Σε κάθε κλήση συστήματος αντιστοιχίζεται ένα αναγνωριστικό (πχ `read`), και η πρότυπη ακολουθία [Π] σχηματίζεται από τέτοια αναγνωριστικά.

Η κατάρα [Κ] και το πρότυπο [Π] επιλέγονται από το διαχειριστή την ώρα που ενεργοποιεί την κατάρα Αναγνωριστή.

α. (10%) *Προτείνετε ένα σχεδιασμό για την κατάρα Α και απαντήστε:*

i. *Ποιές δομές στην υποδομή κατάρων και στον πυρήνα χρησιμοποιείτε για την κατάρα [Κ] και το πρότυπο [Π];*
ii. *Πώς επεμβαίνετε στις κλήσεις συστήματος, πώς είναι και πώς καλείται το checkpoint σας;*

β. (10%) *Θέλουμε να αποκτήσουμε με τη βοήθεια της κατάρας Αναγνωριστή, ένα φλοιό από τον οποίο όλα τα προγράμματα που εκτελούμε δεν θα μπορούν να εκτελούν άλλα προγράμματα.*

Περιγράψτε πώς θα χρησιμοποιήσετε την κατάρα Αναγνωριστή [Α], την ακολουθία [Π], και τη λειτουργικότητα της κατάρας [Κ].

γ. (10%) *Έστω ένα δικτυακό πρόγραμμα εξυπηρετητή ο οποίος συγχρονίζει την ώρα των πελατών οι οποίοι συνδέονται σε αυτόν με πρωτόκολλο TCP. Το πρόγραμμα αυτό, στην αρχή της εκτέλεσής του ανοίγει μόνο ένα αρχείο με* `open()`*, αυτό με τη ρύθμιση του σε ποια πόρτα TCP να ακούει. Για λόγους ασφαλείας, θέλουμε να απαγορέψουμε στο πρόγραμμα να ανοίξει οποιοδήποτε άλλο αρχείο μέσω της* `open()`*.*

Περιγράψτε πώς θα χρησιμοποιήσετε την κατάρα Αναγνωριστή [Α], την ακολουθία [Π], και τη λειτουργικότητα της κατάρας [Κ].

α. Προσθέτουμε στο PCB την Κ, την Π, και έναν ακέραιο δείκτη στη θέση της Π η οποία έχει αναγνωρισθεί.

Σε κάθε διαθέσιμη κλήση συστήματος ελέγχουμε αν το επόμενο στοιχείο του δείκτη στην Π ανταποκρίνεται στην κλήση. Αν ανταποκρίνεται, προχωράμε το δείκτη. Αν όχι, τότε ο δείκτης μένει στάσιμος. Αν ο δείκτης φτάσει στο τέλος της Π, τότε απενεργοποιούμε την Α και ενεργοποιούμε την Κ. Ο στάσιμος δείκτης εξασφαλίζει την αναγνώριση προτύπων ανεξάρτητα από τις ενδιάμεσες κλήσεις.

Επεξήγηση της απάντησης:

```
1       struct target_curse {
2               curse_id_t      curse_id;
3               curse_data_t    curse_data;
4       };
5
6       struct matcher_data {
7               cusrse_data_t   target_curse,
8               unsigned int    pattern_index,
9               unsigned int    pattern_length,
10              syscall_id_t    *syscall_pattern;
11      };
12
13      typedef unsigned int syscall_id_t;
14              ή
15      typedef char syscall_id_t[8];
```

Προσθέτουμε σε κάθε system call που μας ενδιαφέρει (ή στον αποκωδικοποιητή που τα καλεί) ένα checkpoint:

```
1       void matcher_checkpoint(syscall_id_t id) {
2               struct matcher_data *matcher_data = &current->matcher_data;
3               unsigned int index = matcher_data->pattern_index;
4               syscall_id_t *pattern = matcher_data->pattern;
5
6               if (pattern[index] != id)
7                       return;
8
9               index += 1;
10              if (index >= matcher_data->patter_length) {
11                      curse_task(current, matcher_data->target_curse);
12                      lift_task(current, matcher_id);
13              }
14              matcher_data->pattern_index = index;
15      }
```

Σημαντικές σχεδιαστικές επιλογές:

Οικομενικά ή ανά διεργασία δεδομένα της Α. Η καλύτερη απόφαση είναι τα δεδομένα της Α να είναι ανά διεργασία. Αν είναι οικουμενικά, τότε αναγκαστικά η Α από δυναμική κατάρα με 2 ορίσματα γίνεται μια στατική κατάρα με δεδομένα τα Π και Κ.

Επίσης δεν κερδίζουμε σε χώρο με το να κάνουμε τα δεδομένα της Α οικουμενικά, εφόσον αυτά θα μπορούσαν να είναι αφενός δυναμικά δεσμευόμενα, και αφετέρου μοιραζόμενα ανά πολλές διεργασίες.

Απόρριψη, οπισθοκύληση, ή στάση της αναγνωρισμένης ακολουθίας Όσο κλήσεις συστήματος ανταποκρίνονται στο Πρότυπο, τότε η αναγνωρισμένη ακολουθία μεγαλώνει ως υπο-ακολουθία του. Όταν προκύψει μια κλήση που δεν ανταποκρίνεται στο επόμενο στοιχείο του Προτύπου υπάρχουν 3 ενδεχόμενα:

(α) Να απορριφθεί ολόκληρη η αναγνωρισμένη ακολουθία

Τελευταία Κλήση:	`close`
Πρότυπο:	`open`, `read`, `close`, `open`, `read`, `write`, `close`
Αναγνώριση-Πριν:	`open`, `read`, `close`, `open`, `read` ← [`close`]
Αναγνώριση-Μετά:	

(β) Να κυλήσει προς την αρχή της η αναγνωρισμένη ακολουθία, απορρίπτοντας τα αρχικά της στοιχεία, εως ότου είτε αδειάσει, είτε αντιστοιχηθεί εκ νέου στο πρότυπο

Πρότυπο:	`open`, `read`, `close`, `open`, `read`, `write`, `close`
Αναγνώριση-Πριν:	`open`, `read`, `close`, `open`, `read` ← [`close`]
Αναγνώριση-Μετά:	`open`, `read`, `close`

(γ) Να παραμείνει στάσιμη η αναγνωρισμένη ακολουθία

Πρότυπο:	`open`, `read`, `close`, `open`, `read`, `write`, `close`
Αναγνώριση-Πριν:	`open`, `read`, `close`, `open`, `read` ← [`close`]
Αναγνώριση-Μετά:	`open`, `read`, `close`, `open`, `read`

Η επιλογή (α) είναι ακατάλληλη για την επιβολή περιορισμών σε διεργασίες γιατί είναι εύκολο να αποφύγουμε την αναγνώριση παρεμβάλλοντας κλήσεις συστήματος.

Ακόμη και αν θεωρούμε ότι η διεργασίες συνεργάζονται, υπάρχει ο κίνδυνος να χαθούν ακολουθίες. Η επιλογή (β) λύνει το 2ο αυτό πρόβλημα της (α) αλλά όχι το πρώτο.

Η επιλογή (γ) είναι η καταλληλότερη για την επιβολή περιορισμών, καθώς μια διεργασία δεν μπορεί να αποφύγει την αναγνώριση ενός προτύπου με κανένα τρόπο παρά μόνο με τη μή εκτέλεσή του.

Ακύρωση η μη της Α μετά τον οπλισμό της Κ Όταν το πρότυπο της Α αναγνωρισθεί εξ ολοκλήρου και ενεργοποιηθεί η Κ για τη διεργασία, προτιμότερο είναι να απενεργοποιηθεί η Α, δεδομένου ότι η μόνη επίδραση που μπορεί να έχει στη διεργασία είναι να ενεργοποιήσει την Κ, η οποία είναι ήδη ενεργοποιημένη.

Υποστήριξη αναδρομικής κλήσης της Α Ανάλογα αν μαζί με την ταυτότητα της κατάρα Κ αποθηκεύουμε και ένα δείκτη στα τυχόντα δεδομένα αρχικοποίησής της, τότε η αναδρομική κλήση της Α υποστηρίζεται.

Δεν απαιτείται όμως από τα παρόντα σενάρια, και θα ήταν καλύτερο η όποια αναδρομική λειτουργικότητα να υλοποιηθεί στη γλώσσα αναγνώρισης προτύπων της Α.

β. Ρίχνουμε την Α στο φλοιό με πρότυπο Π = [`fork`, `execve`, `fork`, `execve`] και κατάρα Κ που στέλνει ακαριαία `SIGKILL` στη διεργασία.

Επεξήγηση της απάντησης: Το ιστορικό των κλήσεων συστήματος ενός φλοιού που εκτελεί προγράμματα είναι, και των προγραμμάτων που εκτελεί είναι:

Φλοιός:	[...`fork`...[not `execve`]...`fork`...]
Πρόγραμμα επιτρεπόμενο:	[`fork`...`execve`...`fork`...`fork`...]
Πρόγραμμα επιτρεπόμενο:	[`fork`...`execve`...[not `fork`]...`execve`]
Πρόγραμμα μη επιτρεπόμενο:	[`fork`...`execve`...`fork`...`execve`]

Ο φλοιός κάνει `fork()`, `execve()` για κάθε πρόγραμμα που τρέχει. Επίσης μπορεί να κάνει `fork()` για δική του χρήση. Ο φλοιός ποτέ δεν έχει `execve()` στην ιστορία του, εκτός από το αρχικό δικό του, γιατί αλλιώς θα σταματούσε (εξ ορισμού!) να είναι φλοιός και θα ήταν εκτελούμενο πρόγραμμα.

Κάθε εκτελούμενο πρόγραμμα έχει στην ιστορία του ένα `fork()`, `execve()`, από την εκτέλεσή του από το φλοιό. Από εκεί και πέρα μπορεί να κάνει όσα fork θέλει (π.χ. multiprocess ή multithreaded πρόγραμμα), όσα `execve()` θέλει (π.χ. `wrapper.sh`→ `app-launcher`→`app.bin`) αλλά δε θα πρέπει να μπορεί να κάνει `execve()` μετά από `fork()`.

Επομένως η ακολουθία [`fork`, `execve`, `fork`, `execve`], δεν πρόκειται ποτέ να αναγνωρισθεί στον φλοιό, λόγω του `execve`, ενώ θα αναγνωρίζει πάντοτε τα προγράμματα που κάνουν `fork()`→`execve()`.

γ. Ρίχνουμε στον φλοιό που εκτελεί τον εξυπηρετητή, την Α με πρότυπο Π = [`listen`] και κατάρα Κ που αποτρέπει κάθε κλήση `open()`.

Επεξήγηση της απάντησης: Η συνθήκη ασφαλείας είναι να μην μπορεί να ανοιχθεί κανένα αρχείο αφού έχει ανοιχθεί το αρχείο των ρυθμίσεων.

Άν το αρχείο αυτό είναι το μοναδικό που ανοίγει ο εξυπηρετητής τότε μπορούμε να ρυθμίσουμε το πρότυπο Π = [`open`].

Εδώ όμως σιωπηρά απαιτούμε το πρόγραμμα του εξυπηρετητή να μην χρησιμοποιεί καμία δυναμική βιλιοθήκη, γιατί αυτές θα πρέπει να ανοιχθούν για να συνδεθούν με το πρόγραμμα, αφαιρώντας τη δυνατότητά μας να καταλάβουμε ποια open είναι εκείνη του αρχείου ρυθμίσεων.

Το πρόγραμμα όμως ενδέχεται να μην μπορεί ή να μη βολεύει να γίνει στατικό (πχ. βλ. `gethostbyname()`).

Αντί να μετράμε τις `open()` χρησιμοποιούμε ένα καλύτερο κριτήριο. γνωρίζουμε ότι η `open()` του αρχείου ρυθμίσεων θα γίνει πριν την `listen()`, γιατί το `listen()` απαιτεί πρότερο `bind()`, το οποίο με τη σειρά του απαιτεί τον αριθμό της πόρτας από το αρχείο ρυθμίσεων.

Άρα αν αποτρέψουμε κάθε `open()` μετά το `listen()`, το open του αρχείου ρυθμίσεων είναι ασφαλές. Ταυτοχρόνως, κάθε `open()` πριν το `listen()` είναι ασφαλές, καθώς πριν το `listen()` είναι αδύνατο να υπάρξει δικτυακή επικοινωνία.

Θέμα 3 (30%)

α. (20%) *Απαντήστε* συνοπτικά *στα εξής:*

i. *Ποιος περιγραφητής αρχείου είναι η καθιερωμένη είσοδος για μια διεργασία;*
ii. *Υπάρχει περίπτωση μια διεργασία να κάνει `read()`, `write()` σε περιγραφητή αρχείου από το δίσκο, χωρίς να έχει κάνει η ίδια το αντίστοιχο `open()`;*
iii. *Αληθές ή ψευδές; Όταν γίνεται `fork()`, οι file descriptors του παιδιού αντιστοιχίζονται στις ίδιες δομές `struct file` με τους file descriptors του πατέρα.*
iv. *Τι συμβαίνει σε μια διεργασία όταν εκτελέσει επιτυχώς `execve()`;*
v. *Τι συμβαίνει στα ανοιχτά αρχεία μιας διεργασίας όταν εκτελέσει επιτυχώς `execve()`;*
vi. *Λύστε την εξής πρόκληση (εκτελέσιμο `challenge`):*

```
1       int main(void)
2       {
3           off_t a, b;
4
5           a = lseek(42, 0, SEEK_CUR); /* Find out current file position */
6           sleep(10);
7           b = lseek(42, 0, SEEK_CUR);
8           if (a == b) FAIL(); else SUCCESS();
9           return 0;
10      }
11
```

Γράψτε πρόγραμμα που να εκτελεί το `challenge`, όπου κι όπως θέλετε, ώστε να επιτυγχάνει. Κάθε κλήση συστήματος του εκτελέσιμου πρέπει να είναι επιτυχής.

i. Ο περιγραφητής `0`.
ii. Ναι, μπορεί να έχει κληρονομήσει το ανοιχτό αρχείο από το `fork()` που τη δημιούργησε. **Παράδειγμα:** Μια `main()` που το μόνο που κάνει είναι `printf("Hello, World\n")` κάνει `write()` στον περιγραφητή `1`, χωρίς ποτέ να έκανε το αντίστοιχο `open()`.
iii. Αληθές. Οι περιγραφητές αρχείου του παιδιού αντιστοιχίζονται στις ίδιες δομές `struct file` με τους file descriptors του πατέρα. **Παράδειγμα:** Αν ο πατέρας κάνει `pipe()` και `fork()`, το παιδί έχει file descriptors που δείχνουν στα ίδια άκρα του pipe, δηλαδή στα ίδια `struct files`.

iv. Το πρόγραμμά της αντικαθίσταται από το εκτελέσιμο που ορίζεται στην `execve()`. Όλες οι απεικονίσεις μνήμης καταστρέφονται.
v. Τίποτε, μένουν ως έχουν, γι'αυτό η `printf()` της `main()` συνεχίζει να λειτουργεί. **Λεπτομέρεια:** Όσα αρχεία είναι σημειωμένα με τη σημαία close-on-exec (`FD_CLOEXEC`) κλείνουν.
vi. Εκμεταλλευόμαστε ότι ο πατέρας και το παιδί μοιράζονται το ίδιο `struct file` όταν το παιδί κληρονομεί file descriptors με `fork()`.

```
1   int main(int argc, char *argv[])
2   {
3       int a;
4       pid_t p;
5
6       a = open("kalhmera", O_RDONLY);
7       dup2(a, 42)
8       p = fork();
9
10      if (p > 0) {
11              sleep(3);
12              lseek(a, 10, SEEK_SET) < 0); /* 10 != 0 */
13              wait(NULL);
14      }
15
16      if (p == 0)
17              execve("challenge", argv, NULL);
18
19      return 0;
20  }
```

Επεξήγηση: Η πρόκληση έχει δύο μέρη: (i) το `challenge` χρησιμοποιεί τον περιγραφητή `42` χωρίς ποτέ να έκανε `open()`, (ii) για να πετύχει πρέπει στο διάστημα των 10s από τη γραμμή 5 στη γραμμή 7 να έχει αλλάξει η θέση του δείκτη ανάγνωσης-εγγραφής στο αρχείο, χωρίς η ίδια να κάνει τίποτε (π.χ. `read()`, `lseek()`) για να τον επηρεάσει.
Για το πρώτο μέρος, ο πατέρας φροντίζει με `dup2()` το ίδιο `struct file` να είναι προσβάσιμο και μέσω του fd που επέστρεψε η `open()` και μέσω του fd `42`.
Για το δεύτερο μέρος, εκμεταλλευόμαστε ότι ο fd που κληρονομείται από το παιδί (`challenge`) αντιστοιχίζεται στην ίδια δομή `struct file`, η οποία περιέχει τον δείκτη ανάγνωσης-εγγραφής στο πεδίο `f_pos`. Ο πατέρας μετακινεί το δείκτη από το `0` στο `10` όσο το παιδί κοιμάται.

β. (10%) *Απαντήστε* συνοπτικά, *δικαιολογήστε τις απαντήσεις σας:*

i. Σε ποιες από τις παρακάτω δομές εφαρμόζεται τεχνική Copy-on-Write και γιατί;

- *Σώμα Ελέγχου Διεργασίας (`struct task_struct`)*
- *Διαπιστευτήρια χρήστη (`struct cred`)*

- *Σελίδες που αποτελούν το σωρό (heap) μιας διεργασίας*

ii. Θεωρήστε το παρακάτω πρόγραμμα:

```
1           unsigned long t, t0;
2           ...
3
4           c = connect(s, (struct sockaddr *)&c_addr, &c_addr_len);
5           t0 = 0;
6           for (;;) {
7                   t = time(NULL);
8                   if (t > t0) {
9                           write(c, &t, sizeof(t));
10                          t0 = t;
11                  }
12          }
13
```

Τι λειτουργία κάνει το πρόγραμμα; τι προβλήματα βλέπετε στην παραπάνω υλοποίησή της, και πώς θα τα διορθώνατε;
Υπόδειξη: *Η `time(NULL)` επιστρέφει τον αριθμό των δευτερολέπτων που έχουν περάσει από τη χρονική στιγμή Epoch, 1970-01-01 00:00:00 +0000 (UTC).*

i. Copy-on-Write εφαρμόζεται στο `struct cred` και στις σελίδες του σωρού, όχι στο `struct task_struct`. Το `struct task_struct` γράφεται από *κάθε* διεργασία, τα διαπιστευτήρια κι οι σελίδες δεν υφίστανται πιθανότατα μεταβολή καθώς συμβαίνουν απανωτά `fork()`.
Επεξήγηση της απάντησης: Τεχνική Copy-on-Write εφαρμόζεται σε δομές που γράφονται σπάνια από διεργασίες, παρόλο που κάθε μία κληρονομεί το δικό της αντίγραφο και σε λογικό επίπεδο έχει τη δική της, ιδιωτική δομή, την οποία μπορεί να γράψει οποτεδήποτε. Με τον τρόπο αυτό γλιτώνουμε χώρο, για την τήρηση των δομών, και χρόνο, για την αντιγραφή τους. Δεν έχει νόημα το Copy-on-write για τη δομή `struct task_struct` γιατί είναι σίγουρο ότι για κάθε διεργασία πανωγράφεται: αλλάζει το PID σε σχέση με τη γονική. Αντίθετα, το `struct cred` πιθανότατα μένει ίδιο, η νέα διεργασία δεν θα αλλάξει τα διαπιστευτήριά της. Για τις σελίδες του σωρού, γλιτώνουμε πολύ χρόνο και χώρο αν κατ'απαίτηση δημιουργούμε ιδιωτικά αντίγραφα, αφού μια νέα διεργασία πιθανότατα θα αλλάξει ένα μικρό ποσοστό από αυτές, ή απλώς θα κάνει αμέσως `execve()`.

ii. Το πρόγραμμα συνδέεται σε έναν απομακρυσμένο υπολογιστή, στον οποίο στέλνει περιοδικά, κάθε δευτερόλεπτο, την τρέχουσα ώρα ως απόσταση σε δευτερόλεπτα από τη στιγμή Epoch. Είναι μια αδρή υλοποίηση ενός network time server. Έχει δύο κύρια προβλήματα:
   - Δεν ελέγχει την τιμή επιστροφής των κλήσεων συστήματος. Ειδικά κλήσεις οπως η `connect()`, που εμπλέκουν απομακρυσμένα μηχανήματα και το δίκτυο είναι πολύ πιθανό να αποτύχουν.
   - Κάνει busy wait περιμένοντας να περάσει το ένα δευτερόλεπτο: Αντί να κοιμηθεί με `sleep()`, οπότε παθαίνει μετάβαση RUNNING→WAITING κι αφήνει τον επεξεργαστή, κάνει busy wait εκτελώντας συνεχώς `time()` μέχρι να αλλάξει η τιμή

επιστροφής της. Με κατάλληλη ρύθμιση του χρόνου της `sleep()` σε λίγο μικρότερο του απαιτούμενου μπορεί να προσεγγιστεί η επιθυμητή ακρίβεια.