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

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%)

Μια επιστημονική αποστολή χρησιμοποιεί αυτόνομους αισθητήρες για να χαρτογραφήσει ένα ανεξερεύνητο υπόγειο σπήλαιο. Οι (εναέριοι) αισθητήρες κινούνται ανεξάρτητα και χρησιμοποιούν τεχνολογία LIDAR (Light Detection And Ranging) για να συλλέξουν τοπογραφικά δεδομένα για τον περιβάλλοντα χώρο τους: φωτίζουν με παλμούς laser διαφορετικού μήκους κύματος τον περιβάλλοντα χώρο, ώστε να εκτιμήσουν την απόσταση από τα γύρω τους αντικείμενα και άλλες ιδιότητές τους. Κάθε παλμός αντιστοιχεί σε διακριτή μέτρηση. Οι αισθητήρες σχηματίζουν ασύρματο δίκτυο κι αποστέλλουν τεράστιο όγκο δεδομένων σε σταθμό βάσης. Ο σταθμός συνδέεται με υπολογιστικό σύστημα που αναλαμβάνει την επεξεργασία των εισερχόμενων μετρήσεων: για κάθε μήκος κύματος κατασκευάζεται ένας τριδιάστατος χάρτης του σπηλαίου.

Θεωρήστε τον οδηγό συσκευής του σταθμού βάσης σε ΛΣ Linux. Κάθε μέτρηση χαρακτηρίζεται, ανάμεσα στα άλλα, από τον αριθμό του αισθητήρα που την κατέγραψε, το μήκος κύματος του laser που χρησιμοποιήθηκε και τις συντεταγμένες του αισθητήρα. Κάθε φορά που ο σταθμός βάσης παραλαμβάνει μια μέτρηση, προκαλείται διακοπή υλικού. Η συνάρτηση χειρισμού της διακοπής αποθηκεύει την εισερχόμενη μέτρηση σε κυκλικό απομονωτή στη μνήμη του οδηγού, αναλόγως με το μήκος κύματος στο οποίο αντιστοιχεί.

Τα δεδομένα των μετρήσεων γίνονται διαθέσιμα σε διεργασίες προς επεξεργασία μέσω ειδικών αρχείων, π.χ. `/dev/lidar/248nm`*,* `/dev/lidar/532nm`*,* `/dev/lidar/1064nm` *που υλοποιεί ο οδηγός του σταθμού βάσης ως οδηγός συσκευής χαρακτήρων.*

Από την πλευρά των διεργασιών ισχύουν τα εξής:

i. *Κάθε `read()` επιστρέφει bytes για* ακέραιο *αριθμό μετρήσεων.*
ii. *Όσο δεν υπάρχουν νέα δεδομένα μετρήσεων για το συγκεκριμένο μήκος κύματος, η διεργασία που επιχειρεί `read()` μπλοκάρει.*
iii. *Λόγω του υπολογιστικού φόρτου είναι πιθανό οι διεργασίες να μην μπορούν να ανταπεξέλθουν στο ρυθμό των εισερχομένων μετρήσεων. Ο οδηγός εξασφαλίζει ότι ποτέ δεν θα επιστραφεί στη διεργασία παλαιότερη μέτρηση (χρόνος άφιξης) από μία που της έχει ήδη επιστραφεί προς επεξεργασία.*

Δίνονται οι εξής συναρτήσεις:

- *`lidar_base_intr()`:*
  Συνάρτηση χειρισμού διακοπών του σταθμού βάσης. Αποθηκεύει την εισερχόμενη μέτρηση σε κατάλληλη δομή `lidar_buffer`.
- `get_hw_measurement(msr)`:
  Διαβάζει την εισερχόμενη μέτρηση από το hardware του σταθμού βάσης και την αποθηκεύσει στη δομή `msr`.
- `get_buf_from_wavelength(wavelength)`:
  Επιστρέφει δείκτη προς τη δομή τύπου `lidar_buffer` που αντιστοιχεί στο μήκος κύματος `wavelength`.
- `get_wavelength_from_minor(minor)`:
  Επιστρέφει το μήκος κύματος που αντιστοιχεί σε δεδομένο minor number ενός ειδικού αρχείου του οδηγού της συσκευής.

Ζητούνται τα εξής:

i. *(3%) Ποιος ο τύπος κι ο ρόλος του πεδίου `lock` της δομής `lidar_buffer`;*
ii. *(2%) Ποιος ο ρόλος του πεδίου `wq` της δομής `lidar_buffer`;*
iii. *(5%) Πώς εξασφαλίζεται η ανεξάρτητη λειτουργία των διαφορετικών ρευμάτων δεδομένων; πώς εξασφαλίζεται ότι μία διεργασία που επεξεργάζεται δεδομένα μετρήσεων στα 1064nm δεν επηρεάζεται από εισερχόμενες μετρήσεις στα 248nm;*
iv. *(5%) Μια διεργασία εκτελεί `read(fd, ...)`. Από ποιο από τα εισερχόμενα ρεύματα δεδομένων θα προέρχονται οι μετρήσεις που θα διαβάσει; πώς καθορίζεται αυτό; Σχολιάστε τη λειτουργία των `get_wavelength_from_minor()`, `get_buf_from_wavelength()`.*
v. *(10%) Ποιος ο ρόλος των πεδίων `rcnt`, `wcnt` της δομής `lidar_buffer`; Υλοποιήστε την `lidar_base_intr()`, συμπληρώνοντας τον σκελετό. Πώς ικανοποιείται η προδιαγραφή (iii) από τον κώδικά σας;*
vi. *(5%) Υλοποιήστε την `lidar_chrdev_open()`, συμπληρώνοντας τον σκελετό.*
vii. *(10%) Υλοποιήστε την `lidar_chrdev_read()`, συμπληρώνοντας τον σκελετό.*

Ο οδηγός στην τελική μορφή του, όπως προκύπτει μετά την υλοποίηση των συναρτήσεων που ζητούνται θα πρέπει να ικανοποιεί τις προδιαγραφές που αναφέρονται παραπάνω.

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

```
struct measurement {
    uint16_t wavelength;
    ...
};


struct lidar_buffer {
    ..locktype.. lock;
    uint16_t wavelength;  /* The wavelength corresponding to this buffer */
    wait_queue_head_t wq;
    uint128_t wcnt, rcnt; /* These are initialized to zero, and
                           * will never wrap. */
#define CIRC_BUF_SIZE (1024 * 1024)
    struct measurement circ_buffer[CIRC_BUF_SIZE];
};

void get_hw_measurement(struct measurement *msr);
struct lidar_buffer *get_buf_from_wavelength(uint16_t wavelength);
uint16_t get_wavelength_from_minor(unsigned int minor);

void lidar_base_intr(void)
{
    struct lidar_buffer *buf;
    struct measurement msr;

    get_hw_measurement(&msr);
    buf = get_buf_from_wavelength(msr->wavelength);
    ... lock buf ...
    memcpy(&buf->circ_buffer[buf->wcnt % CIRC_BUF_SIZE],
        &msr, sizeof(struct measurement));
    ++buf->wcnt;
    . . .
    ...unlock buf...
    . . .
}

static int lidar_chrdev_open(struct inode *inode, struct file *filp)
{
    unsigned int minor = iminor(inode);
    . . .
}

ssize_t lidar_chrdev_read(struct file *filp, char __user *usrbuf,
    size_t cnt, loff_t *f_pos)
{
    struct lidar_buffer *buf = filp->private_data;

    /* Only return whole measurements */
    . . .

    /* Retrieve measurements, block if no data available */
    . . .

    /* Eventually, i measurements are being returned */
    return i * sizeof(msr);
}
```

i. Το πεδίο `lock` είναι κλείδωμα που προστατεύει κάθε instance της δομής `lidar_buffer` από ταυτόχρονη πρόσβαση. Επειδή συναγωνίζονται γι'αυτό και process context (κά-

θε διεργασία που εκτελεί την `lidar_chrdev_read()` σε process context) και interrupt context (ο χειριστής `lidar_base_intr()`), πρέπει να είναι τύπου `spinlock_t`.

ii. Το πεδίο `wq` χρησιμοποιείται για την υλοποίηση ουράς αναμονής για τις διεργασίες που χρειάζεται να μπλοκάρουν περιμένοντας δεδομένα στην αντίστοιχη δομή `lidar_buffer`. Όταν έρθουν νέα δεδομένα για το αντίστοιχο μήκος κύματος, ο χειριστής διακοπών τα τοποθετεί στην αντίστοιχη δομή `lidar_buffer` κι ενημερώνει τις διεργασίες που περιμένουν, ξυπνώντας την ουρά αναμονής.

iii. Κάθε διακριτό ρεύμα εισερχόμενων δεδομένων καταλήγει σε ανεξάρτητη δομή `lidar_buffer` με βάση το μήκος κύματος. Ο χειριστής διακοπών `lidar_base_intr()` επιλέγει την κατάλληλη δομή (γραμμή 27) και δουλεύει μόνο με αυτή: αποθηκεύει τα εισερχόμενα δεδομένα και ξυπνάει την αντίστοιχη ουρά. Άρα, δεν υπάρχει περίπτωση διεργασία που ενδιαφέρεται για σήματα στα 1064nm να επηρεαστεί από μέτρηση στα 248nm, γιατί περιμένει σε άλλη ουρά αναμονής και διαβάζει από άλλον κυκλικό απομονωτή.

iv. Το εισερχόμενο ρεύμα δεδομένων (ουσιαστικά η δομή `lidar_buffer`) καθορίζεται από το ειδικό αρχείο με `open()` του οποίου προέκυψε ο `fd`. Κατά το `open()`, η `lidar_chrdev_open()` πρέπει να φροντίζει να καταγράφει τη δομή `lidar_buffer` ως μέλος `file->private_data` της δομής `struct file` που μόλις κατασκευάστηκε. Στη δομή αυτή καταλήγει από το minor number του ειδικού αρχείου στο οποίο εκτελείται η `open()`, μέσω των `get_wavelength_from_minor()`, για να πάρει το αντίστοιχο μήκος κύματος, και `get_buffer_from_wavelength()` για να πάρει τη δομή `lidar_buffer`.

v. Τα πεδία `wcnt`, `rcnt` καταγράφουν πόσες *μετρήσεις* έχουν αποθηκευτεί στον απομονωτή από τον interrupt handler κι έχουν διαβαστεί από διεργασίες, αντίστοιχα. Όπως αναφέρει το σχόλιο στη γραμμή 11, δεν μηδενίζονται ποτέ. Η ακριβής θέση ανάγνωσης ή εγγραφής στον κυκλικό buffer προκύπτει με πράξεις `% CIRC_BUF_SIZE`, όπως στη γραμμή 29.

```
1   void lidar_base_intr(void)
2   {
3       struct lidar_buffer *buf;
4       struct measurement msr;
5       unsigned long flags;
6
7       get_hw_measurement(&msr);
8       buf = get_buf_from_wavelength(msr->wavelength);
9       spin_lock_irqsave(&buf->lock, flags);
10      memcpy(&buf->circ_buffer[buf->wcnt % CIRC_BUF_SIZE],
11          &msr, sizeof(struct measurement));
12      ++buf->wcnt;
13      if (buf->wcnt - buf->rcnt > CIRC_BUF_SIZE)
14          ++buf->rcnt;
15      spin_unlock_irqrestore(&buf->lock, flags);
16      wake_up_interruptible(&buf->wq);
17  }
```

Η προδιαγραφή (iii) ορίζει ότι ακόμη κι αν ο ρυθμός των εισερχόμενων δεδομένων είναι μεγαλύτερος από τον ρυθμό με τον οποίο οι διεργασίες αδειάζουν τον buffer, οι μετρήσεις θα επιστρέφονται πάντα με αύξουσα σειρά άφιξης. Με άλλα λόγια, ότι στην περίπτωση buffer overrun οι διεργασίες θα αναγκαστούν να χάσουν τις μετρήσεις που δεν πρόλαβαν να ανακτήσουν, ώστε να ανταποκριθούν στον εισερχόμενο ρυθμό. Ο interrupt handler καλύπτει την προδιαγραφή μην αφήνοντας τον `buf->wcnt` να απομακρυνθεί πάνω από `CIRC_BUF_SIZE` από τον `buf->rcnt`. Αν αυτό πάει να συμ-

βεί, ο `buf->rcnt` ακολουθεί, κι οι διεργασίες χάνουν εισερχόμενες μετρήσεις για να προλάβουν.

vi. Η ακόλουθη `lidar_chrdev_open()` φροντίζει να καταγράψει τη σωστή δομή `lidar_buffer` ως `file->private_data`. Ο `buf` είναι ήδη αρχικοποιημένος και πιθανώς έχει μέσα δεδομένα, δεν πρέπει να ακουμπήσει τα πεδία του.

```
1  void lidar_chrdev_open(void)
2  {
3      unsigned int minor = iminor(inode);
4      struct lidar_buffer *buf;
5
6      buf = get_buf_from_wavelength(get_wavelength_from_minor(minor));
7      filp->private_data = buf;
8      return 0;
9  }
```

vii. Η ακόλουθη `lidar_chrdev_read()` επιστρέφει μετρήσεις στο userspace, αντλώντας από τον κλειδωμένο κυκλικό buffer. Δεν επιτρέπεται πρόσβαση στο userspace κρατώντας spinlock, οπότε κάνει ένα ενδιάμεσο αντίγραφο στη στοίβα της.

```
1  ssize_t lidar_chrdev_read(struct file *filp, char __user *usrbuf,
2      size_t cnt, loff_t *f_pos)
3  {
4      struct lidar_buffer *buf = filp->private_data;
5      int i, msrcnt, valid, will_eof;
6
7  #define MAX_MSRS_PER_READ 10
8      struct measurement msrs[MAX_MSRS_PER_READ]
9      unsigned long flags;
10
11     /*
12      * Only return whole measurements, return at least ONE measurement,
13      * do not overrun the stack-based buffer.
14      */
15     msrcnt = cnt / sizeof(struct measurement);
16     if (msrcnt == 0)
17         return -EINVAL;
18
19     if (msrcnt > MAX_MSRS_PER_READ)
20         msrcnt = MAX_MSRS_PER_READ;
21
22     spin_lock_irqsave(&buf->lock, flags);
23     while (dev->rcnt == dev->wcnt) {
24         spin_lock_irqrestore(&buf->lock, flags);
25         if (wait_event_interruptible(buf->wq, (buf->rcnt != buf->wcnt)))
26             return -ERESTARTSYS;
27         spin_lock_irsqave(&buf->lock, flags);
28     }
29
30     /* Keep buffer locked while copying to msrs[] */
31     if (msrcnt > buf->wcnt - buf->rcnt)
32         msrcnt = buf->wcnt - buf->rcnt;
33
34     for (i = 0; i < msrcnt; i++) {
35         /* Cannot copy_to_user() holding a spinlock */
36         memcpy(&msrs[i], &buf->circ_buffer[buf->rcnt % CIRC_BUF_SIZE],
37             sizeof(struct measurement));
38         ++buf->rcnt;
```

```
39      }
40
41      spin_unlock_irqrestore(&buf->lock, flags);
42
43      if (copy_to_user(usrbuf, msrs, msrcnt * sizeof(struct measurement)))
44          return -EFAULT;
45
46      /* Eventually, msrcnt measurements are being returned */
47      return msrcnt * sizeof(struct measurement);
48  }
```

Θέμα 2 (30%)

Μια κρίσιμη εφαρμογή εξυπηρετητή παρουσιάζει απρόβλεπτα προβλήματα στη λειτουργία της. Θα θέλαμε, κατά βούληση, να καταγράφουμε ένα στιγμιότυπο ολόκληρης της εικονικής μνήμης της διεργασίας, ώστε να μπορέσουμε να το μελετήσουμε προς αναζήτηση της αιτίας των προβλημάτων.

Λόγω της κρισιμότητας της εφαρμογής και της φύσης του προβλήματος έχουμε τους εξής περιορισμούς:

i. *Η λειτουργία της εφαρμογής δεν πρέπει να σταματήσει.*
ii. *Η επίδοση της εφαρμογής δεν πρέπει να επιβαρυνθεί σημαντικά (όπως όταν συνδέουμε πάνω της κάποιο εργαλείο παρακολούθησης με `ptrace()`).*
iii. *Το στιγμιότυπο θέλουμε να είναι όσο το δυνατόν συνεπές, και όχι συνοθύλευμα καταστάσεων διάσπαρτων στο χρόνο.*

θα ενσωματώσουμε τη λύση μας σε μία κατάρα Μνημογράφο.

α. (10%) *Με ποιον τρόπο ζητάμε την καταγραφή της μνήμης μιας διεργασίας; με ποιον τρόπο γίνεται η καταγραφή και πώς αποκτούμε πρόσβαση στην καταγεγραμμένη μνήμη;*

β. (15%) *Περιγράψτε τι δομές δεδομένων στον πυρήνα χρησιμοποιεί η κατάρα Μνημογράφος. σε ποια σημεία επεμβαίνει και πώς;*

γ. (5%) *Περιγράψτε πώς χειρίζεστε την υπόλοιπη κατάσταση της διεργασίας (καταχωρητές, περιγραφητές αρχείων, διαχείριση σημάτων, πολυνηματισμός, μοιραζόμενη μνήμη κ.ά.)*

α. Την καταγραφή της μνήμης την δρομολογούμε σημειώνοντας τη διεργασία με την κατάρα Μνημογράφος. Η καταγραφή της μνήμης εκτελείται κατά την επόμενη κλήση συστήματος της διεργασίας, όπου πριν την αρχική κλήση, εκτελείται η `fork()`. Η απόγονος διεργασία που προκύπτει είναι το στιγμιότυπο που ζητάμε και η πρόσβαση στη μνήμη γίνεται με τους καθιερωμένους τρόπους (πχ. core dump ή debugger).

β. Η επέμβαση στον πυρήνα γίνεται σε ένα σημείο ελέγχου (checkpoint), στην είσοδο των διεργασιών στον πυρήνα, πριν εκτελεστεί η κάθε κλήση συστήματος.

Το σημείο ελέγχου επιτελεί δύο λειτουργίες. Πρώτον, πριν καλέσει την κανονική κλήση συστήματος, ελέγχει για τη διεργασία και αν είναι σημειωμένη, καλεί την `fork()` και αίρει τη σημείωση στον πατέρα (για να μην επαναληφθεί).

Η δεύτερη λειτουργία ενεργοποιείται στην απόγονο διεργασία, φροντίζοντας να τη σταματήσει και να την επεξεργαστεί με όποιο τρόπο χρειάζεται (βλ. γ).

γ. Η προσέγγιση με `fork()`, εκτός από τη μνήμη, μας δίνει στιγμιότυπο και της υπόλοιπης κατάστασης της διεργασίας, επομένως δεν χρειάζεται ειδικό χειρισμό για την καταγραφή της. Παρ'όλ'αυτά, θα πρέπει γρήγορα να καταγράψουμε και να κλείσουμε, όλους τους περιγραφητές αρχείων, όλους τους signal handlers και όλους τους υπόλοιπους μοιραζόμενους πόρους, που ενδέχεται να επηρεάσουν την εκτέλεση του πατέρα.

Επεξήγηση της απάντησης: Ιδανικά, θα θέλαμε με μεγάλη ταχύτητα να αντικαταστήσουμε όση μνήμη είναι δυνατόν με απεικονίσεις copy-on-write και στη συνέχεια χωρίς πίεση χρόνου και χωρίς κίνδυνο αλλαγής των δεδομένων να διερευνήσουμε ή να αντιγράψουμε το στιγμιότυπο που προκύπτει.

Οι απεικονίσεις copy-on-write θα εξασφαλίσουν ότι με μέριμνα του ίδιου του λειτουργικού, όσα δεδομένα αλλάζει η διεργασία-στόχος, θα αντιγράφονται και θα απεικονίζονται σε νέες περιοχές μνήμης, ώστε το στιγμιότυπο να παραμείνει αναλλοίωτο. Επειδή η αντιγραφή αυτή γίνεται μόνο για δεδομένα που αλλάζουν, τα οποία ενδεχομένως είναι ένα μικρό ποσοστό του συνόλου, κρατάμε την κατανάλωση μνήμης και τον χρόνο αντιγραφών στο ελάχιστο.

Ο πιο εύκολος στην υλοποίηση, πιο αξιόπιστος και πιο γρήγορος τρόπος να καταγράψουμε ένα στιγμιότυπο copy-on-write της μνήμης μιας διεργασίας είναι να χρησιμοποιήσουμε τον ήδη υπάρχοντα μηχανισμό του λειτουργικού συστήματος που κάνει ακριβώς αυτό, την κλήση συστήματος `fork()`.

Επομένως, θα πρέπει να εισάγουμε ένα σημείο ελέγχου (checkpoint) στην είσοδο της διεργασίας στον πυρήνα, πριν από κάθε κλήση συστήματος, έτσι ώστε όταν το ενεργοποιήσουμε να εκτελέσει πρώτα μία κλήση `fork()` και μετά να συνεχίσει κανονικά.

Η ενεργοποίηση του σημείου ελέγχου μπορεί να γίνει με απλό έλεγχο για το άν η κατάρα Μνημογράφος είναι σημειωμένη για τη διεργασία. Όταν επιθυμούμε την καταγραφή στιγμιοτύπου μιας διεργασίας, αρκεί να τη σημειώσουμε. Στην επόμενη κλήση συστήματος θα γίνει η καταγραφή.

Μετά την καταγραφή, όμως, έχουμε 2 διεργασίες, η μία απόγονος της άλλης. Την απόγονο θα πρέπει να την σταματήσουμε έτσι ώστε να μην επηρεάσει τη διεργασία-στόχο, αλλά και να μην μεταβάλει την κατάσταση της μνήμης της, η οποία αποτελεί το στιγμιότυπό μας.

Για το λόγο αυτό, χρειαζόμαστε επέμβαση αμέσως μετά την κλήση της `fork()`. Για τη

διεργασία-πατέρα, θα πρέπει να απενεργοποιήσουμε αμέσως την κατάρα (για να μην επαναληφθεί στιγμιότυπο στην επόμενη κλήση συστήματος). Την απόγονο θα πρέπει να τη σταματήσουμε.

Πρόσβαση στη μνήμη της απογόνου θα μπορούσαμε να αποκτήσουμε με τους καθιερωμένους τρόπους, πχ. core dump ή προσάρτηση debugger με `ptrace()`.

Τέλος, με την προσέγγιση `fork()` αποκτούμε πρόσβαση όχι μόνο στη μνήμη, αλλά και στην υπόλοιπη κατάσταση της διεργασίας. Θα πρέπει όμως να λάβουμε υπόψη τον εξής κίνδυνο. Οι μοιραζόμενοι πόροι με τη διεργασία-στόχο μπορούν να επηρεάσουν τη λειτουργία της, καθιστώντας την καταγραφή στιγμιοτύπου αδιάφανη. π.χ. οι περιγραφητές αρχείων, οι απεικονίσεις στη μνήμη αρχείων, οι μοιραζόμενες περιοχές μνήμης, signal handlers, θα συνεχίσουν να λειτουργούν και να δεσμεύουν πόρους παρόλο που η διεργασία-στόχος τα έχει τερματίσει, καθώς η απόγονος κρατάει αντίγραφα. Αυτό είναι εν δυνάμει καταστροφικό για τη λειτουργία της εφαρμογής, όταν πχ. βασίζεται στο κλείσιμο και του τελευταίου περιγραφητή σε μία σωλήνωση, αφήνοντας την όποια διεργασία-αναγνώστη της σωλήνωσης να περιμένει για πάντα μια σηματοδότηση EOF που δεν θα έρθει (λόγω του παραμένοντος ανοικτού περιγραφητή της απογόνου). Ο πολυνηματισμός δεν μας επηρεάζει, γιατί η νέα διεργασία αποτελείται μόνο από ένα νήμα.

Επομένως, ιδανικά θα πρέπει αμέσως μετά την καταγραφή τους, όλοι οι μοιραζόμενοι πόροι να αποδεσμεύονται. Δηλαδή οι περιγραφητές να κλείνουν (το σημαντικότερο), οι signal handlers να απενεργοποιούνται, οι μοιραζόμενες απεικονίσεις να αποδεσμεύονται και εν τέλει, η απόγονος να τερματίζει.

θέμα 3 (30%)

α. (15%) *Απαντήστε* συνοπτικά*, δικαιολογήστε τις απαντήσεις σας:*

i. (10%) 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. (5%) Αληθές ή ψευδές; Απαντήστε με σύντομη *αιτιολόγηση.*

- *Για την ίδια διεργασία, δύο διαφορετικοί file descriptors αντιστοιχίζονται πάντα σε διαφορετικές δομές `struct file` του πυρήνα.*
- *Μια διεργασία σε κατάσταση WAITING εγγυημένα έχει εκτελέσει κλήση συστήματος και βρίσκεται μέσα στον πυρήνα.*

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.
- Ψευδές. Όταν γίνεται `fork()`, οι file descriptors του παιδιού αντιστοιχίζονται στις δομές `struct file` από τον πατέρα, καταλήγουν στο ίδιο ανοιχτό αρχείο, γι'αυτό δουλεύουν τα pipes. Ομοίως, με `dup()` μπορώ να φτιάξω file descriptors που να αντιστοιχούν στο ίδιο `struct file`.
- Ψευδές. Αν η διεργασία προκαλέσει page fault, παραμένει σε WAITING μέχρι να έρθουν τα αντίστοιχα blocks από το δίσκο. Δεν έκανε κλήση συστήματος, έκανε αναφορά στη μνήμη με load/store, η οποία προκάλεσε page fault.

β. (15%) *Δίνεται το παρακάτω πρόγραμμα.*

```
1  #include <...>
2
3  void child(void)
4  {
5      char *newargv[] = { "executable", NULL };
6      char *newenviron[] = { NULL };
7      execve(newargv[0], newargv, newenviron);
8      exit(1);
9  }
10
11 volatile int flag = 0;
12 void hndlr(int signum)
13 { flag = 1; }
14
15 int main(void)
16 {
17     pid_t p;
18
19     signal(SIGCHLD, hndlr);
20
21     p = fork();
22     if (p == 0)
23         child();
24
25     sleep(2);
26     kill(p, SIGTERM);
```

```
27      while (!flag)
28          ;
29      if (kill(p, SIGKILL) == -1) printf("SUCCESS!\n"); else printf("FAIL!\n");
30      return 0;
31  }
```

Θεωρήστε ότι οι κλήσεις συστήματος εκτός της `kill()` δεν αποτυγχάνουν.
Απαντήστε συνοπτικά στα εξής:

i. *(2%) Τι κάνει η κλήση συστήματος `signal()`;*
ii. *(3%) Υπάρχει περίπτωση το πρόγραμμα να μην τερματίσει ποτέ; Αν ναι, περιγράψτε ένα τέτοιο σενάριο, κάνοντας ό,τι υπόθεση χρειάζεται για τη συμπεριφορά του `executable`.*
iii. *(5%) Υπάρχει περίπτωση το πρόγραμμα να τερματίσει εκτυπώνοντας `FAIL`; Αν ναι, περιγράψτε ένα τέτοιο σενάριο, κάνοντας ό,τι υπόθεση χρειάζεται για τη συμπεριφορά του `executable`.*
iv. *(5%) Τι δεν κάνει σωστά αυτό το πρόγραμμα; Ποιες αλλαγές θα κάνατε στον παραπάνω κώδικα ώστε* ανεξάρτητα *από τη συμπεριφορά του `executable` το ισοδύναμο πρόγραμμα να τερματίζει εκτυπώνοντας `SUCCESS`; Δεν μπορείτε να κάνετε* καμία *παραδοχή για το εκτελέσιμο `executable`.*

i. Η κλήση `signal(SIGXXX, handler)` κανονίζει να κληθεί η συνάρτηση χειρισμού `handler` όταν παραληφθεί από τη διεργασία το σήμα `SIGXXX`.
ii. Ναι. Αν το εκτελέσιμο χειρίζεται το σήμα `SIGTERM` το οποίο αποστέλλεται στη γραμμή 26, `kill(p, SIGTERM)`, δεν θα πεθάνει. Αν δεν πεθάνει, δεν θα αποσταλεί ποτέ `SIGCHLD` στον πατέρα, δεν θα τρέξει ποτέ ο handler για να θέσει το `flag`, γραμμή 13 και το πρόγραμμα θα μείνει για πάντα κολλημένο στις γραμμές 27-28.
iii. Ναι. Το πρόγραμμα εκτυπώνει `FAIL` στη γραμμή 29 αν η `kill(p, SIGKILL)` *πετύχει*. Για να συμβεί αυτό σημαίνει ότι η διεργασία-παιδί έχει πεθάνει (ή έστειλε ρητά ένα `SIGCHLD`), οπότε ο πατέρας πέρασε τη γραμμή 28. Παρόλα αυτά, η `kill(p, SIGKILL)` θα επιτύχει, διότι το PID `p` συνεχίζει να υπάρχει στον πίνακα διεργασιών, ως zombie.
iv. Κάνει busy-wait περιμένοντας να πεθάνει το παιδί και δεν κάνει `wait()` για να συλλέξει τον κωδικό επιστροφής του. Για να τερματίζει με `SUCCESS`, χρειάζεται κλήση `wait(NULL)` μετά τη γραμμή 28. Τότε η `kill(p, SIGKILL)` θα αποτυγχάνει με `ESRCH`.

Επεξήγηση της απάντησης: Ένα παιδί που έχει πεθάνει αλλά δεν έχει ακόμη καθαριστεί από τον πατέρα, παραμένει στον πίνακα διεργασιών ως zombie, οπότε οι κλήσεις `kill(p, signal)` επιτυγχάνουν, αντί να αποτυγχάνουν με `ESRCH`. Από το manual page kill(2):

> **ESRCH** The pid or process group does not exist. Note that an existing process might be a zombie, a process which already committed termination, but has not yet been wait(2)ed for.

Τέλος, αν το `executable` είναι κακόβουλο, μπορεί να αρχίσει να στέλνει σήματα στον πατέρα, σκοτώνοντάς τον. Στην ακραία αυτή περίπτωση, ο πατέρας δεν τερματίζει εξασφαλισμένα με `SUCCESS`. Αυτό το ενδεχόμενο προλαμβάνεται με το πρόγραμμα να εκτελείται

ως `root` και να *ρίχνει* τα δικαιώματά του μέσα στο παιδί, ακριβώς πριν από το `execve()`. Έτσι ο πατέρας εκτελείται ως `root` και το παιδί ως `nobody`, π.χ., οπότε δεν μπορεί να τον επηρεάσει στέλνοντάς του σήματα. Για περισσότερες λεπτομέρειες δείτε setuid(2).