ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
http://www.cslab.ece.ntua.gr

Λειτουργικά Συστήματα

7ο εξάμηνο, Ακαδημαϊκό Έτος 2013-2014

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 (15%)

Αντιμετωπίζουμε το πρόβλημα συγχρονισμού του νηπιαγωγείου ("Kindergarten"), όπου οι κανονισμοί επιβάλλουν να είναι πάντα παρών ένας δάσκαλος ανά `R` παιδιά, όπου `R` ακέραιος αριθμός (π.χ. αναλογία παιδιών/δασκάλων 3:1), ώστε να εξασφαλίζεται η σωστή επίβλεψη των παιδιών. Επιπλέον, το σχήμα μας επιθυμούμε να έχει τα παρακάτω χαρακτηριστικά:

1. *Αν ένας δάσκαλος αποπειραθεί να αποχωρήσει αλλά αυτό δεν είναι δυνατό, θα πρέπει να επιστρέφει στα καθήκοντά του (και να μην "κολλάει" αναμένοντας πότε θα ικανοποιηθεί η συνθήκη εξόδου).*
2. *Κάθε γονέας θα πρέπει να μπορεί να εισέρχεται στο χώρο του νηπιαγωγείου προκειμένου να ελέγξει αν τηρείται ο κανονισμός.*

Δώστε σχήμα συγχρονισμού των διεργασιών `Teacher`, `Child` και `Parent` που ικανοποιεί τις παραπάνω απαιτήσεις ορίζοντας τις συναρτήσεις `*_enter()`, `*_exit()` και `verify_compliance()` που χρησιμοποιούνται στη συνέχεια, ή αντικαθιστώντας τις κλήσεις τους με κατάλληλο κώδικα εντός των `Teacher`, `Child` και `Parent`. Μπορείτε να χρησιμοποιήσετε κλειδώματα, σημαφόρους και μοιραζόμενες μεταβλητές (mutexes/locks, semaphores, shared variables).

```
void Teacher()
{
    for (;;) {
        teacher_enter();
        ...critical section...
        teach();
        teacher_exit();
        go_home();
    }
}
```

```
void Child()
{
    for (;;) {
        child_enter();
        ...critical section...
        learn();
        child_exit();
        go_home();
    }
}
```

```
void Parent()
{
    for (;;) {
        parent_enter();
        ...critical section...
        verify_compliance();
        parent_exit();
        go_home();
    }
}
```

Για τη λύση του προβλήματος συγχρονισμού θα χρησιμοποιήσουμε τις κοινές μεταβλητές `teacher`, `child` και `room` που μετρούν τον αριθμό των δασκάλων, των παιδιών και του ελεύθερου χώρου για παιδιά αντίστοιχα. Θα χρησιμοποιήσουμε το μοναδιαίο σημαφόρο `mutex` για να ελέγξουμε την

πρόσβαση στα κοινά δεδομένα και το μοναδιαίο σημαφόρο door που προσομοιώνει τη λειτουργία της πόρτας εισόδου του σταθμού.
Κάθε δάσκαλος κατά την είσοδό του ενημερώνει τους κατάλληλους μετρητές και στην περίπτωση που η έλευσή του είναι αυτή που θα επιτρέψει σε νέα παιδιά να εισέλθουν "ανοίγει την πόρτα εισόδου". Για να εξέλθει ο δάσκαλος ελέγχει αν ισχύουν οι συνθήκες προκειμένου να φύγει και ανάλογα, είτε φεύγει, είτε επιστρέφει στα καθήκοντά του.
Τα παιδιά με τη σειρά τους αναμένουν στην πόρτα εισόδου μέχρι να μπορούν να εισέλθουν. Στη συνέχεια κάνουν έναν περαιτέρω έλεγχο αν υπάρχει χώρος (γιατί, π.χ. στο μεσοδιάστημα μπορεί να έφυγε κάποιος δάσκαλος) και αν υπάρχει χώρος εισέρχονται, αν όχι ξαναπεριμένουν στην πόρτα. Σε περίπτωση που μετά την είσοδό τους εξακολουθεί να υπάρχει ελεύθερος χώρος, φροντίζουν να "αφήσουν την πόρτα ανοιχτή".

```
shared int teacher = 0, child = 0;
mutex semaphore(1);
door semaphore(0);

void Teacher()
{
    for (;;) {
        wait(mutex);
        teacher++;
        room += R;
        if (room == R)
            signal(door);
        signal(mutex);

        for (;;) {
            teach();
            wait(mutex);
            if (room >= R) {
                teacher--;
                room -= R;
                signal(mutex);
                break;
            }
            signal(mutex);
        }
        go_home();
    }
}
```

```
void Child()
{
   for (;;) {
        for (;;) {
            wait(door);
            wait(mutex);
            if (room) {
                child++;
                room--;
                if (room)
                  signal(door);
                signal(mutex);
                break;
            }
            signal(mutex);
        }

        learn();
        wait(mutex);
        child--;
        room++;
        signal(mutex);

        go_home();
   }
}
```

```
void Parent()
{
    for (;;) {
        wait(mutex);
        if ((teacher - 1) * R >= child)
            printf("Regulation respected\n");
        else
            printf("Regulation violated\n");
        signal(mutex);
        go_home();
    }
}
```

Θέμα 2 (20%)

Δίνεται το πρόγραμμα C σε περιβάλλον UNIX που ακολουθεί μετά τα ζητούμενα.
Θεωρήστε ότι οι κλήσεις συστήματος δεν αποτυγχάνουν, η `Fn()` δεν επιστρέφει ποτέ, η `Fn()` δεν εκτελεί κλήσεις συστήματος, κάθε νέα διεργασία κληρονομεί ακριβώς τον τρόπο χειρισμού σημάτων του πατέρα της και τέλος ότι οι κλήσεις συστήματος διακόπτονται από εισερχόμενα σήματα.
Δεδομένου ότι η `Fn()` δεν επιστρέφει ποτέ, οι διεργασίες που δημιουργεί το πρόγραμμα έρχονται σε μόνιμη *κατάσταση: το δέντρο διεργασιών μένει σταθερό για πάντα και κάθε διεργασία είναι μέσα σε συγκεκριμένη συνάρτηση ή κλήση συστήματος.*

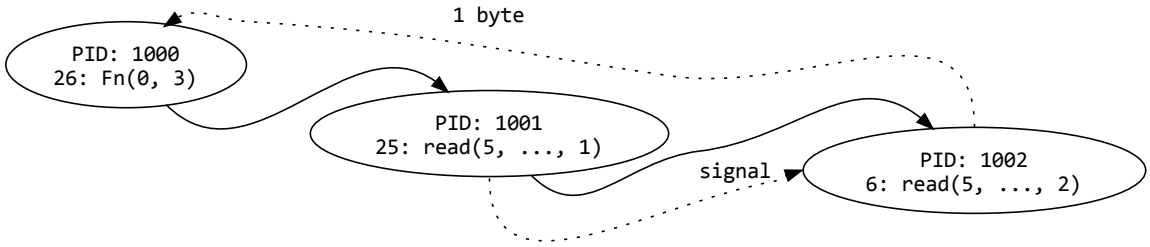
α. (12%) *Σχεδιάστε το δέντρο διεργασιών στην* τελική *του μορφή, όταν δηλαδή όλες οι διεργασίες έχουν φτάσει σε* μόνιμη *κατάσταση. Εξηγήστε* συνοπτικά *πώς προκύπτει.*

β. (4%) *Σε κάθε κόμβο του δέντρου διεργασιών επισημάνετε: (i) την κλήση συστήματος ή συνάρτηση μέσα στην οποία βρίσκεται ο Program Counter της αντίστοιχης διεργασίας, (ii) τα ορίσματα με τα οποία αυτή έχει κληθεί, (iii) τη γραμμή του προγράμματος απ' όπου έγινε η κλήση της.*
Για τα ορίσματα, κάντε οποιαδήποτε υπόθεση χρειάζεστε για νούμερα που δεν γνωρίζετε, π.χ. PIDs που ανατίθενται από το σύστημα στις νέες διεργασίες.

γ. (4%) *Συμπληρώστε το δέντρο διεργασιών ώστε να φαίνεται η διαδιεργασιακή επικοινωνία: για κάθε μεταφορά δεδομένων ή αποστολή σήματος που συνέβη, σχεδιάστε ένα διακεκομμένο βέλος από τη διεργασία-αποστολέα στη διεργασία-παραλήπτη. Πάνω στο βέλος γράψτε την τιμή που μεταφέρεται κάθε φορά.*

```
1   int n, pd[2];
2   char c;
3   pid_t p;
4
5   void h(int s)
6   { write(pd[1], &p, 2); read(pd[0], &p, 2); Fn(pd[0], pd[1]); }
7
8   int main(void)
9   {
10      pipe(pd);
11      signal(SIGUSR1, h);
12      n = pd[1];
13
14      p = fork();
15      if (p == 0) {
16          pipe(pd);
17          pd[1] = n;
18          p = fork();
19      } else
20          p = 0;
21
22      if (p)
23          kill(p, SIGUSR1);
24
25      read(pd[0], &c, 1);
26      Fn(p, pd[0]);
27      wait(&status);
28      Fn(4, 2);
29      return 0;
30  }
```

Το δέντρο διεργασιών, η θέση του PC για κάθε διεργασία και κάθε διαδιεργασιακή επικοινωνία φαίνεται στο ακόλουθο σχήμα.



Η αρχική διεργασία (έστω με PID `1000`) δημιουργεί ένα pipe (έστω `pipe0`), προσβάσιμο μέσω των περιγραφητών αρχείου `3`, `4`. Θέτει `n = 4`. Δημιουργεί παιδί (`1001`), γρ. 14. Θέτει `p = 0`, γρ. 20, προσπερνά την γρ. 22, μπλοκάρει σε ανάγνωση από το `pipe0` στη γρ. 25.
Η `1001` δημιουργείται, γρ. 14, δημιουργεί ένα δεύτερο pipe (έστω `pipe1`), προσβάσιμο μέσω περιγραφητών `5`, `6`. Συνεχίζει να έχει ανοιχτό το `pipe0`. Οι τιμές `5`, `6`, γράφονται στον πίνακα `pd[2]`, πανωγράφοντας τις υπάρχουσες. Έπειτα, χρησιμοποιεί την μεταβλητή `n` για να θέσει το `pd[1]` ίσο με τον περιγραφητή του άκρου εγγραφής του `pipe0`, `pd[1] = 4`. Η κατάσταση των `pipe0`, `pipe1` δεν επηρεάζεται. Δημιουργεί παιδί (έστω `1002`), γρ. 18 και του στέλνει σήμα `SIGUSR1`, γρ. 23. Τέλος μπλοκάρει σε `read()` από τον περιγραφητή `5`, δηλ. από το `pipe1`, γρ. 25.
Η `1002` δημιουργείται, γρ. 22 και λαμβάνει το `SIGUSR1`. Μπορεί να έχει προλάβει να μπλοκάρει στο `read()` από το `pipe1`, μπορεί και όχι. Δεν μας ενδιαφέρει, δίνεται ότι τα εισερχόμενα σήματα διακόπτουν τις κλήσεις συστήματος. Εκτελεί τον signal handler (γρ. 6), γράφει 2 bytes στο `pipe0` (λόγω της γρ. 17), επιχειρεί να διαβάσει 2 bytes από το `pipe1` και μπλοκάρει για πάντα.
Τέλος, η `1000` ξυπνάει, και μένει για πάντα μέσα στην `Fn(0, 3)`.
Υπενθύμιση: Κάθε διεργασία στο UNIX κληρονομεί ανοιχτούς τους περιγραφητές αρχείου `0` (stdin), `1` (stdout), `2` (stderr) από τον πατέρα της. Οπότε, δεν περιμένουμε οι τιμές αυτές να χρησιμοποιούνται για τους περιγραφητές αρχείων των pipes, χωρίς να περιγράφεται ρητά προηγούμενο κλείσιμό τους με `close()`.

Θέμα 3 (40%)

α. (20%) *Ο κώδικας που σας δίνεται στη συνέχεια βρίσκει τους πρώτους αριθμούς σε δεδομένο διάστημα `[low, high]`. Το πρόγραμμα αυτό εκτελείται από τη διεργασία Α σε σύστημα με έναν επεξεργαστή και ΛΣ που υποστηρίζει εικονική μνήμη με σελιδοποίηση κατ' απαίτηση (demand paging).*

```
1   int main(int argc, char *argv[])
2   {
3       long i, j, prime, low, high, counter = 0;
4       ...
5       low = atol(argv[1]);
6       high = atol(argv[2]);
7       ...
8       for (i = low; i <= high; i += 2) {
9           prime = 1;
10          for (j = 3; j < sqrt(i) + 1; j += 2)
11              if (i % j == 0) {
12                  prime = 0;
13                  break;
14          }
```

```
15          if (prime) {
16              counter++;
17              printf("%ld is a prime number\n", i);
18          }
19      }
20
21      printf("%ld prime numbers were found in interval [%ld, %ld]\n",
22          counter, low, high);
23
24      return 0;
25  }
```

i. *Κατά την εκτέλεση του κυρίως αλγορίθμου (γραμμές 8 – 19) επισημάνετε τα σημεία στον κώδικα στα οποία μπορεί να κληθεί ο χρονοδρομολογητής αν έχουμε διακοπτή (preemptive) ή μη-διακοπτή (non-preemptive) πολιτική χρονοδρομολόγησης. (3%)*
ii. *Προτείνετε αλλαγές στον κώδικα του προγράμματος ώστε κατά την εκτέλεση των συγκεκριμένων γραμμών σε σύστημα με μη-διακοπτή χρονοδρομολόγηση, το πρόγραμμά σας να μονοπωλήσει τον επεξεργαστή, δηλαδή να ολοκληρωθεί η εκτέλεση των γραμμών χωρίς το πρόγραμμα να αφήσει τον επεξεργαστή. Η έξοδος του προγράμματος θα πρέπει να είναι ακριβώς η ίδια μετά τις αλλαγές που θα κάνετε. (5%)*
iii. *Πώς αναμένετε να αλλάξει η ταχύτητα εκτέλεσης αν επιτύχετε τη μονοπώληση και πού θα οφείλονται τυχόν αλλαγές; (4%)*
iv. *Είναι δυνατόν να επιτευχθεί η μονοπώληση για οποιοδήποτε εύρος του διαστήματος `[low, high]` ; (4%)*
v. *Θεωρήστε ότι στο σύστημα υπάρχει άλλη μία διεργασία (έστω B) έτοιμη προς εκτέλεση. Αν επιτύχετε την παραπάνω μονοπώληση σχολιάστε πώς θα μεταβληθούν για το συγκεκριμένο χρονικό διάστημα της μονοπώλησης οι μετρικές: ρυθμός διεκπεραίωσης (throughput) και χρόνος αναμονής (waiting time) για τη διεργασία A, τη διεργασία B και το σύστημα συνολικά. Δικαιολογήστε την απάντησή σας. (4%)*

i. Αν έχουμε διακοπτή πολιτική χρονοδρομολόγησης ο χρονοδρομολογητής μπορεί να κληθεί σε οποιοδήποτε σημείο. Αν έχουμε μη-διακοπτή πολιτική, ο χρονοδρομολογητής μπορεί να κληθεί μέσα από κάποιο system call, δηλαδή στη γραμμή 17, αφού η κλήση της `printf()` μπορεί να οδηγήσει σε κλήση της `write()`.
ii. Για να επιτύχουμε το στόχο μας θα πρέπει να αφαιρέσουμε την κλήση συστήματος μέσα από το συγκεκριμένο μπλοκ κώδικα. Προκειμένου να διατηρήσουμε την ίδια έξοδο, μπορούμε να φυλάξουμε την τιμή κάθε πρώτου αριθμού που βρίσκει το πρόγραμμά μας σε μία δομή (π.χ. ένα πίνακα) και να κάνουμε τις εκτυπώσεις όλες μαζί στο τέλος.
iii. Η ταχύτητα εκτέλεσης θα αυξηθεί για το συγκεκριμένο μπλοκ κώδικα μιας και ο επεξεργαστής θα χρησιμοποιείται αποκλειστικά από τη συγκεκριμένη διεργασία. Επίσης, εξοικονομούμε και το κόστος από τις εναλλαγές περιβάλλοντος (context switch). Τέλος, μπορούμε να αναμένουμε και βελτίωση στο χρόνο από τις μαζικές εκτυπώσεις κατά την τελική φάση της εκτύπωσης (π.χ. κατασκευάζοντας μεγάλα string εκτύπωσης και μειώνοντας τις κλήσεις συστήματος).
iv. Για μεγάλο εύρος `[low, high]` η πρόσβαση στη δομή που θα κρατάει τους πρώτους αριθμούς θα οδηγήσει σε page faults, οπότε εκεί θα εμπλακεί ο χρονοδρομολογητής και θα σταματήσει η μονοπώληση.
v. Ο ρυθμός διεκπεραίωσης θα αυξηθεί για τη διεργασία A, θα μηδενιστεί για τη διεργασία B και θα αυξηθεί συνολικά για το σύστημα εφόσον θα αποφεύγεται το κόστος του context switch. Ο χρόνος αναμονής θα μηδενιστεί για τη διεργασία A, θα αυξηθεί για τη διεργασία B και θα μειωθεί για το σύστημα συνολικά (ομοίως επειδή αποφεύγεται ο χρόνος αναμονής του context switch).

β. (20%) *Αγοράσατε σύστημα με πολυπύρηνο επεξεργαστή και θέλετε να εκτελέσετε το πρόγραμμα υπολογισμού των πρώτων αριθμών του προηγούμενου ερωτήματος παράλληλα. Καταστρώστε ενδεικτικό παράλληλο πρόγραμμα:*

i. Με πολλαπλές διεργασίες στο μοντέλο του UNIX (`fork()` / `wait()`).
ii. Με πολλαπλά νήματα ακολουθώντας τη λογική των POSIX threads.

Η έξοδος και των δύο εκδόσεων του παράλληλου προγράμματος θα πρέπει να συμφωνεί με αυτή του σειριακού προγράμματος ως προς τα αποτελέσματα που προκύπτουν και όχι απαραίτητα ως προς της σειρά εμφάνισής τους.
ΣΗΜΕΙΩΣΗ: Δεν καλείστε να αναπαράγετε ακριβώς τις κλήσεις συναρτήσεων σε κάθε περίπτωση, αλλά να χρησιμοποιήσετε τους αντίστοιχους μηχανισμούς που σας παρέχονται.

Στη συνέχεια δίνονται πλήρη προγράμματα που υλοποιούν τα ζητούμενα του ερωτήματος (χωρίς ελέγχους σφαλμάτων, ορισμάτων, κλπ). Η ορθή επίλυση του θέματος δεν απαιτεί τη λεπτομέρεια και ακρίβεια των προγραμμάτων που ακολουθούν.

Λύση με fork() / wait():

```
1  /*
2   * parallel prime numbers with fork() / wait()
3   *
4   */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <math.h>
9  #include <unistd.h>
10 #include <sys/types.h>
11 #include <sys/wait.h>
12
13 long int count_primes(long int low, long int high);
14
15 int main(int argc, char *argv[])
16 {
17
18     long int i, tmp, low, high, *p_low, *p_high, chunk, counter = 0;
19     int procs, **pd, pid, status;
20     char buf[32];
21
22     low = atol(argv[1]);
23     high = atol(argv[2]);
24     procs = atoi(argv[3]);
25
26     p_low = malloc(procs * sizeof(long int));
27     p_high = malloc(procs * sizeof(long int));
28     pd = malloc(procs * sizeof(int *));
29
30     for (i = 0; i < procs; i++)
31         pd[i] = malloc(2 * sizeof(int));
32
33     chunk = (high - low) / procs;
34
35     p_low[0] = low;
36
37     for (i = 0; i < procs; i++) {
38
39         if (i < procs - 1){
40             p_high[i] = p_low[i] + chunk;
41             p_low[i+1] = p_high[i] + 1;
42         } else
43             p_high[i] = high;
44
45         pipe(pd[i]);
46
47         pid = fork();
48
49         if (pid) {
50             write(pd[i][1], &p_low[i], sizeof(p_low[i]));
```

```
51              write(pd[i][1], &p_high[i], sizeof(p_high[i]));
52          } else {
53              read(pd[i][0], &low, sizeof(low));
54              read(pd[i][0], &high, sizeof(high));
55              counter = count_primes(low, high);
56              write(pd[i][1], &counter, sizeof(counter));
57              exit(0);
58          }
59      }
60
61      for (i = 0; i < procs; i++) {
62          wait(&status);
63      }
64
65      for (i = 0; i < procs; i++) {
66          read(pd[i][0], &tmp, sizeof(tmp));
67          counter += tmp;
68      }
69
70      fprintf(stdout,"%ld prime numbers were found in interval %ld %ld\n",
71          counter, low, high);
72
73      return 0;
74  }
75
76  long int count_primes(long int low, long int high)
77  {
78      long int i, j, prime, counter = 0;
79
80      for (i = low; i <= high; i += 2) {
81          prime = 1;
82          for (j = 3; j < sqrt(i) + 1; j += 2)
83              if (i % j == 0) {
84                  prime = 0;
85                  break;
86          }
87          if (prime) {
88              counter++;
89              fprintf(stdout,"%ld is a prime number\n", i);
90          }
91      }
92
93      return counter;
94  }
```

Λύση με POSIX threads:

```
1  /*
2   * parallel prime numbers with POSIX threads
3   *
4   */
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <math.h>
8  #include <pthread.h>
9
10 typedef struct t_args {
11    long int low, high;
12    long int counter;
13 } thread_args;
14
15 void *count_primes_t(void *args);
16
17 int main(int argc, char *argv[])
18 {
19     pthread_t *thr;
20     thread_args *thr_args;
21
22     long int i, low, high, *p_low, *p_high, chunk, counter = 0;
23     int threads;
24
25     low = atol(argv[1]);
26     high = atol(argv[2]);
27     threads = atoi(argv[3]);
28
29     p_low = malloc(threads * sizeof(long int));
30     p_high = malloc(threads * sizeof(long int));
31
32     thr = malloc(threads * sizeof(pthread_t));
33     thr_args = malloc(threads * sizeof(thread_args));
34
35     chunk = (high - low) / threads;
36
37     p_low[0] = low;
38
39     for (i = 0; i < threads; i++) {
40
41         if (i < threads - 1){
42             p_high[i] = p_low[i] + chunk;
43             p_low[i+1] = p_high[i] + 1;
44         }
45         else p_high[i] = high;
46
47         thr_args[i].low = p_low[i];
48         thr_args[i].high = p_high[i];
49
50         pthread_create(&thr[i], NULL, count_primes_t, &thr_args[i]);
```

```
51      }
52
53      for (i = 0; i < threads; i++) {
54          pthread_join(thr[i], NULL);
55          counter += thr_args[i].counter;
56      }
57
58      fprintf(stdout,"%ld prime numbers were found in interval %ld %ld\n",
59          counter, low, high);
60
61      return 0;
62  }
63
64  void *count_primes_t(void *args)
65  {
66     thread_args *my_args;
67     long int i, j, my_low, my_high, counter = 0;
68     int prime;
69
70     my_args = args;
71     my_low = my_args->low;
72     my_high = my_args->high;
73
74     for(i = my_low; i <= my_high; i += 2) {
75        prime = 1;
76        for (j = 3; j < sqrt(i) + 1; j += 2)
77           if (i % j == 0) {
78              prime = 0;
79              break;
80           }
81        if (prime) {
82              counter++;
83              fprintf(stdout,"%ld is a prime number\n", i);
84         }
85     }
86
87     my_args->counter = counter;
88     pthread_exit(NULL);
89  }
```

Θέμα 4 (25%)

α. (10%) *Έστω ΛΣ με υποστήριξη σελιδοποίησης κατ'απαίτηση (demand paging), μοιραζόμενης μνήμης για διαδιεργασιακή επικοινωνία και CoW για τη δημιουργία νέων διεργασιών.*

i. *Μια διεργασία εκτελεί `fork()`. Τι παθαίνουν οι εγγραφές του πίνακα σελίδων της;*
ii. *Περιγράψτε ένα σενάριο όπου αμέσως μετά το `fork()` υπάρχουν ακόμη σελίδες με το bit* Write *αναμμένο στον πίνακα σελίδων της γονικής διεργασίας. Γιατί συμβαίνει αυτό;*
iii. *Περιγράψτε ένα σενάριο όπου μια διεργασία έχει δικαίωμα εγγραφής σε μεταβλητή `var` χωρίς να είναι αναμμένο το bit* Write *στην αντίστοιχη εγγραφή του πίνακα σελίδων.*
iv. *Υπάρχει περίπτωση μια διεργασία να έχει δικαίωμα εγγραφής σε μεταβλητή `var` και να μην έχει αναμμένο το bit* Write *στην αντίστοιχη εγγραφή του χάρτη μνήμης;*

v. *Υπάρχει περίπτωση μια διεργασία να εκτελέσει `open()` σε αρχείο στο δίσκο και να μην πάει σε WAITING κατά την εκτέλεση κλήσεων `read()` από αυτόν τον περιγραφητή;*
vi. *Υπάρχει περίπτωση μια διεργασία να πάει σε WAITING χωρίς να δεχτεί κάποιο σήμα και χωρίς να εκτελέσει κλήση συστήματος;*

i. Η διεργασία χάνει το δικαίωμα WRITE σε κάθε σελίδα. Ο σκοπός είναι να μπορούν οι διεργασίες να μοιράζονται πλαίσια φυσικής μνήμης, διατηρώντας όμως την ψευδαίσθηση ότι κάθε διεργασία έχει τον δικό της, ιδιωτικό χώρο μνήμης. Οπότε, δεν μπορεί πλέον η διεργασία να έχει απευθείας δικαίωμα εγγραφής στα κοινά πλαίσια, πρέπει να προκληθεί page fault και να το χειριστεί το ΛΣ, ώστε να της ανατεθεί ένα νέο, ιδιωτικό πλαίσιο στο οποίο θα έχει δικαίωμα WRITE.
ii. Η γονική διεργασία έχει ρητά ζητήσει (π.χ. με `mmap(..., MAP_SHARED, ...)`) περιοχές μοιραζόμενης μνήμης με τα παιδιά της. Οπότε, οι εγγραφές της στις αντίστοιχες σελίδες πρέπει να είναι ορατές από τις διεργασίες-παιδιά της, οπότε για αυτές τις σελίδες το bit WRITE μένει αναμμένο.
iii. Είναι το κλασικό σενάριο CoW. Θα συμβεί page fault και το ΛΣ θα αναθέσει νέο πλαίσιο, αφού συμβουλευτεί τον χάρτη μνήμης.
iv. Όχι. Το ΛΣ διατηρεί κάθε δικαίωμα πρόσβασης στον χάρτη μνήμης. Αν το bit WRITE δεν είναι αναμμένο, η διεργασία δεν έχει δικαίωμα εγγραφής και θα δεχτεί σήμα `SIGSEGV` (Segmentation fault).
v. Ναι. Η ίδια ή άλλη διεργασία έχει ήδη διαβάσει τα συγκεκριμένα blocks του αρχείου, οπότε αυτά έχουν ήδη έρθει στην cache που τηρεί το λειτουργικό στη RAM, οπότε η διεργασία μένει RUNNING κι ο πυρήνας κάνει `memcpy()` τα δεδομένα από την cache στον userspace buffer.
vi. Ναι. Στην περίπτωση page fault όπου πρέπει να έρθουν σελίδες από το δίσκο.

β. (5%) *Έστω κατάλογος `/u` ιδιοκτησίας `user2` στον οποίο όλοι οι χρήστες έχουν δικαίωμα εγγραφής χωρίς περιορισμούς, αρχείο `/u/prv1` που ανήκει στο χρήστη `user1`, και διεργασία με PID `1234` του `user2`. Οι δύο χρήστες εκτελούν κατά σειρά τις εξής εντολές, με επιτυχία:*

- *(χρήστης `user1`) `chmod ugo=r /u/prv1`: Όλοι οι χρήστες έχουν πρόσβαση μόνο για ανάγνωση*
- *(χρήστης `user2`) `ln /u/prv1 /u/prv2`: Δημιουργία νέου hard link `prv2` στο `prv1`*

Απαντήστε στα ακόλουθα, με σύντομη αιτιολόγηση:

i. *Τι θα γίνει αν η διεργασία `1234` καλέσει `open("/u/prv1", O_WRONLY);`*
ii. *Τι θα γίνει αν η διεργασία `1234` καλέσει `open("/u/prv2", O_RDONLY);`*
iii. *Τι δικαιώματα έχει ο `user2` στο `prv1` και στο `prv2`;*
iv. *Τι θα γίνει αν ο `user2` επιχειρήσει να εκτελέσει `rm prv2`;*
v. *Τι θα γίνει αν ο `user2` επιχειρήσει να εκτελέσει `rm prv1`;*

Για να απαντηθούν τα παρακάτω χρειάζεται να λάβουμε υπόψη ότι:

- Τα μεταδεδομένα ενός αρχείου (μέγεθος, ιδιοκτήτης, δικαιώματα πρόσβασης κλπ) τηρούνται στο *i-node* του.
- Τα *ονόματα* (hard links) που δείχνουν προς το συγκεκριμένο αρχείο τηρούνται στους αντίστοιχους καταλόγους.
- Δημιουργία νέου hard link προς υπάρχον i-node συνεπάγεται μόνο δημιουργία ονόματος, τα μεταδεδομένα του i-node δεν επηρεάζονται, εκτός από κάποιον reference counter που μετρά το πλήθος των links.

Οπότε:

i. Το `/u/prv1` και το `/u/prv2` είναι hard links στο ίδιο i-node άρα έχουν κοινό ιδιοκτήτη και κοινά permissions. Και τα δύο ανήκουν στον `user1`, με δικαίωμα μόνο ανάγνωσης για όλους. Άρα η συγκεκριμένη κλήση αποτυγχάνει, η `open()` επιστρέφει `-1` και θέτει `errno = EACCES` (Access denied).

ii. Η κλήση επιτυγχάνει κι επιστρέφει έναν περιγραφητή αρχείου.
iii. Ακριβώς τα ίδια και στα δύο: Το αρχείο ανήκει στον `user1` κι ο `user2` έχει δικαίωμα μόνο για ανάγνωση.
iv. Η εντολή `rm` εκτελεί την κλήση συστήματος `unlink()` για να διαγράψει το συγκεκριμένο όνομα. Η διαγραφή του ονόματος επηρεάζει τον *κατάλογο*, όχι το i-node. Από τη στιγμή που *κάθε* χρήστης έχει δικαίωμα εγγραφής στον `/u`, ο `user2` θα μπορέσει να διαγράψει το όνομα του αρχείου στο οποίο δεν έχει δικαίωμα εγγραφής. Δείτε και ερώτηση 6.17 στο `http://www.faqs.org/faqs/linux/faq/part4/`.
v. Ομοίως. Η διαγραφή του ονόματος θα ολοκληρωθεί.

γ. (10%) *Η διεργασία `1234` έχει ανοίξει το `/u/prv2` και εκτελεί συνεχόμενες κλήσεις `read()` για να διαβάσει ευαίσθητα δεδομένα του `user1` και να τα εμφανίσει στην οθόνη.*
Ο `user1` επιθυμεί να σταματήσει την πρόσβαση της `1234` στο αρχείο. Ποιες από τις παρακάτω μεθόδους ή συνδυασμός τους θα δουλέψουν και γιατί; (5%)

i. *Εκτέλεση της εντολής `chmod go-r prv1`, ώστε να έχει αποκλειστική πρόσβαση στο `prv1`.*
ii. *Εκτέλεση της εντολής `chmod go-r prv2`, ώστε να έχει αποκλειστική πρόσβαση στο `prv2`.*
iii. *Εκτέλεση της εντολής `kill -9 1234`.*
iv. *Διαγραφή των αρχείων `prv1`, `prv2`, με την εντολή `rm prv1 prv2`.*

Περιγράψτε τρόπο με τον οποίο ο `user1` μπορεί σίγουρα να σταματήσει τις αναγνώσεις της `1234`. (5%)
Για να απαντηθούν τα παρακάτω, χρειάζεται να λάβουμε υπόψη ότι:

- Ο πυρήνας μελετά τα δικαιώματα ενός αρχείου κι αποφασίζει αν θα επιτρέψει ή όχι την πρόσβαση κατά την εκτέλεση της `open()`.
- Από τη στιγμή που μια διεργασία έχει έγκυρο περιγραφητή, έχει ανοιχτό αρχείο προς συγκεκριμένο i-node, αυτό δεν επηρεάζεται από αλλαγή των δικαιωμάτων του αρχείου στο δίσκο.

Άρα κανείς συνδυασμός των προτεινόμενων ενεργειών δεν θα δουλέψει:

i. Δεν θα επηρεάσει τη διεργασία `1234`. Έχει ήδη ανοιχτό αρχείο.
ii. Δεν θα επηρεάσει τη διεργασία `1234`. Έχει ήδη ανοιχτό αρχείο.
iii. Η διεργασία της εντολής `kill` εκτελείται με τα δικαιώματα του `user1` και δεν έχει δικαίωμα να στείλει μήνυμα στην `1234` που ανήκει στον `user2`.
iv. Η διαγραφή των ονομάτων θα επηρεάσει τον reference counter πάνω στο i-node, αλλά όχι το ανοιχτό αρχείο που διατηρεί η `1234`. Όταν το ανοιχτό αρχείο κλείσει, το i-node θα απελευθερωθεί.

Αυτό που σίγουρα θα δουλέψει, είναι ο `user1` είναι να πετσοκόψει (truncate) το αρχείο, ανοίγοντάς το με `open("/u/prv1", O_WRONLY | O_TRUNC)`. Ουσιαστικά έχει πρόσβαση στο i-node μέσω υπάρχοντος ονόματος και μηδενίζει το μήκος του. Κάθε επόμενη `read()` της `1234` θα επιστρέφει EOF. Προαιρετικά, μπορεί προηγουμένως να αντιγράψει τα δεδομένα του αρχείου σε δικό του ιδιωτικό.