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 2012-2013

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 (25%)

Δίνεται το πλήρες 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. (5%) *Απαντήστε* συνοπτικά*, όχι πάνω από δύο γραμμές, στα ακόλουθα:*

1. *Η κλήση `getppid()` επιστρέφει το PID του γονέα της διεργασίας που την καλεί. Υπάρχει περίπτωση να αλλάξει η τιμή επιστροφής της από κλήση σε κλήση για μια διεργασία;*
2. *Όταν επιτυγχάνει κλήση `fork()`, τι σχέση έχει η μνήμη που βλέπει η διεργασία-παιδί με τη μνήμη της διεργασίας-πατέρα;*
3. *Τι κάνουν οι κλήσεις `kill(pid, SIGINT)` και `signal(SIGINT, handler)` στο UNIX;*
4. *Έστω διεργασίες `P0`, `P1` που εκτελούνται ταυτόχρονα. Η `P0` υπολογίζει την ψευδοτυχαία τιμή `rndval` και κάνει `*p = rndval`, όπου `p` δείκτης. Με ποιον μηχανισμό μπορεί η `P1` να διαβάσει αυτή την τιμή ως `*q`, όπου `q` δείκτης στη δική της μνήμη;*

1. Υπάρχει. Έστω η διεργασία με PID `1001`, με γονέα την `1000`. Οπότε `getppid() == 1000`. Αν η `1000` πεθάνει, τα παιδιά της υιοθετούνται από την `init`, οπότε εφεξής `getppid() == 1`.
2. Οι διεργασίες αναφέρονται ("βλέπουν") πάντα σε *εικονική* μνήμη. Η μνήμη της διεργασίας-παιδιού είναι *ιδιωτικό* αντίγραφο της μνήμης του πατέρα του όταν έγινε το `fork()`.
3. Η κλήση `kill(pid, SIGINT)` στέλνει το σήμα `SIGINT` στη διεργασία με PID `pid`. Η κλήση `signal(SIGINT, handler)` κανονίζει ώστε όταν η καλούσα διεργασία λάβει σήμα `SIGINT` να εκτελέσει τη συνάρτηση χειρισμού σήματος `handler`.
4. Οι διεργασίες πρέπει να έχουν εγκαταστήσει μοιραζόμενο τμήμα μνήμης ανάμεσά τους. Αν οι `p`, `q` είναι οι δείκτες προς τη μοιραζόμενη μνήμη για τις `P0`, `P1` αντίστοιχα, ό,τι γράφει η `P0` στο `*p` το διαβάζει η `P1` στο `*q`.

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

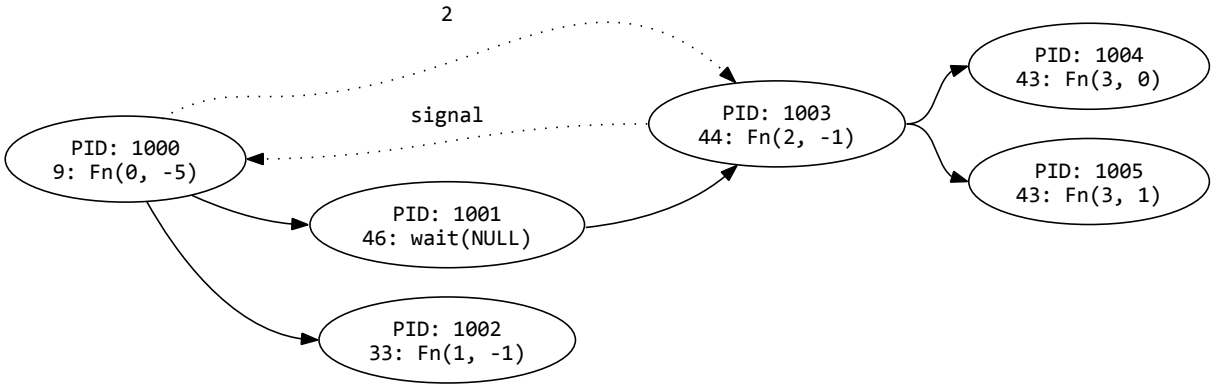
γ. (4%) *Για κάθε κόμβο του δέντρου διεργασιών, γράψτε: (i) την κλήση συστήματος ή συνάρτηση μέσα στην οποία βρίσκεται ο PC της αντίστοιχης διεργασίας, (ii) τα ορίσματα με τα οποία αυτή έχει κληθεί, (iii) τη γραμμή του προγράμματος απ' όπου έγινε η κλήση της.*
Για τα ορίσματα, κάντε οποιαδήποτε υπόθεση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

δ. (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  #include <...> /* Θεωρήστε όλες τις απαραίτητες οδηγίες #include */
2
3  void Fn(...); /* Θεωρήστε δεδομένο τον ορισμό της */
4
5  int pg[2];
6  int glob = 1;
7
8  void handler(int signum)
9  { write(pg[1], &glob, sizeof(glob)); Fn(0, -5); }
10
11 int main(void)
12 {
13     int n = 5, i;
14     pid_t p, a;
15
16     glob = 2;
17     pipe(pg);
18     signal(SIGUSR1, handler);
19
20     a = getpid();
21     p = fork();
22     p = fork();
23
24     if (a == getpid()) {
25         while (1)
26             ;
27         glob = 4; write(pg[1], &glob, sizeof(glob));
28         Fn(0, -1);
29     }
30
31     if (p == 0) {
32         if (a == getppid()) {
33             Fn(1, -1);
34             exit(3);
35         }
36         sleep(5);
37         glob = 3;
38         kill(a, SIGUSR1);
39         read(pg[0], &n, sizeof(n));
```

```
40
41          for (i = 0; i < n; i++)
42                  if (fork() == 0)
43                      Fn(3, i);
44          Fn(2, -1);
45      }
46      wait(NULL);
47  }
```

Το δέντρο διεργασιών στην τελική κατάσταση φαίνεται στο ακόλουθο σχήμα.



Η αρχική διεργασία εκτελεί `fork()` (γραμμή 21) και μετά και αυτή και το παιδί της εκτελούν εκ νέου `fork()`, οπότε γεννούν από μία διεργασία ακόμη. Οι 4 διεργασίες, έστω `1000, 1001, 1002, 1003` διακρίνονται ως εξής: Για την `1000` ισχύει `a == getpid()` (γραμμή 20), οπότε εκτελεί busy-wait στις 25-26. Για τις `1002`, `1003` ισχύει `p == 0`, ενώ η `1001` καταλήγει στη γραμμή 46. Ανάμεσα στις `1002`, `1003`, μόνο η `1002` έχει πατέρα την αρχική, οπότε καταλήγει στη γραμμή 33. Η `1003` κοιμάται για λίγο στην 36, στέλνει σήμα `SIGUSR1` στην `1000` και μπλοκάρει στο pipe. Η `1000` φεύγει από τις 25-26, μπαίνει στον handler, γράφει την τιμή `2` (γραμμή 16) στο pipe, και καταλήγει στη γραμμή 9. Τέλος, η `1003` ξεμπλοκάρει, γεννά 2 παιδιά (41-43), και καταλήγει στη 44.

Επεξήγηση της απάντησης: Οι 4 διεργασίες βρίσκονται να εκτελούν τον ίδιο κώδικα από τη γραμμή 23 και κάτω. Διακρίνονται είτε με βάση το PID τους (είναι ίδιο με της αρχικής;), είτε με το αν βρίσκονται στα φύλλα του δέντρου (`p == 0`), σε συνδυασμό με το PID του πατέρα τους.

Ότι η μεταβλητή `glob` είναι "global" στο πρόγραμμα (ορίζεται εκτός κάποιας συνάρτησης) αφορά στη γλώσσα προγραμματισμού C και μόνο: Είναι ορατή από όλα τα τμήματα του προγράμματος, *και* από τη `main()` *και* από τη `handler()`. *Δεν* είναι μοιραζόμενη μνήμη ανάμεσα στις διεργασίες, άρα η `1000` *δεν* βλέπει τις αλλαγές στη μνήμη άλλων διεργασιών (γραμμή 37).

Θέμα 2 (25%)

α. (5%) *Τι είναι σημαφόρος και ποια είναι η χρησιμότητά του; Αναφέρετε πρόβλημα συγχρονισμού που μπορεί να λύσει ο σημαφόρος και δεν μπορεί το απλό κλείδωμα (lock).*

Σημαφόρος είναι μια ακέραια μεταβλητή η οποία, αφού αρχικοποιηθεί σε ορισμένη τιμή, είναι προσβάσιμη μόνο μέσω λειτουργιών `P - wait()` και `V - signal()`. Η λειτουργία `wait()` μπλοκάρει τον καλούντα έως ότου η τιμή του σημαφόρου γίνει θετική, οπότε του επιτρέπει να προχωρήσει αφού μειώσει κατά ένα την τιμή του σημαφόρου. Ο έλεγχος της τιμής του σημαφόρου κι η μεταβολή του γίνεται ατομικά. Η λειτουργία `signal()` αυξάνει κατά ένα την τιμή του σημαφόρου. Παράδειγμα προβλήματος που λύνεται με σημαφόρους κι όχι ένα απλό κλείδωμα είναι το ordering.

β. (5%) *Δώστε τον ορισμό του σημαφόρου κατά Dijkstra. Η υλοποίηση με βάση τον ορισμό αυτό λέμε ότι οδηγεί στο πρόβλημα της "ενεργού αναμονής". Τι είναι το πρόβλημα αυτό και γιατί είναι ανεπιθύμητο;*

```
wait(S)
{
    while (S <= 0)
        ; /* no-op */
    S--;
}
```

```
signal(S)
{
    S++;
}
```

Η παραπάνω υλοποίηση οδηγεί σε ενεργό αναμονή: οι διεργασίες που περιμένουν μέσα στη `wait()` εκτελούν συνεχώς έναν άερο βρόχο `while()` έως ότου μπορέσουν να μπουν στο κρίσιμο τμήμα. Αυτό είναι ανεπιθύμητο γιατί σπαταλώνται κύκλοι του επεξεργαστή, οι οποίοι ιδανικά θα μπορούσαν να χρησιμοποιηθούν από άλλες διεργασίες που είναι ταυτόχρονα προς εκτέλεση στο σύστημα.

γ. (5%) *Δώστε υλοποίηση του σημαφόρου που αποφεύγει την ενεργό αναμονή. Επισημάνετε σε ποια σημεία έχει περιοριστεί πλέον η αναμονή.*

Για να αποφύγουμε την ενεργό αναμονή, χρειάζεται οι διεργασίες που περιμένουν σε `wait()` να *μπλοκάρουν*, απελευθερώνοντας τον επεξεργαστή. Διεργασία που πρέπει να περιμένει σε `wait()` τοποθετεί τον εαυτό της σε ουρά αναμονής, μία για κάθε σημαφόρο, και γίνεται WAITING. Η αντίστοιχη `signal()` θα ξυπνήσει μία διεργασία από την ουρά, αλλάζοντας την κατάστασή της σε READY.

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

```
wait (semaphore *S)
{
    S->value--;
    if (S->value < 0) {
        add proc to S->list;
        block();
    }
}
```

```
signal (semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove P from S->list;
        wakeup(P);
    }
}
```

Επειδή και πάλι οι `wait()`, `signal()` πρέπει να εκτελούνται ατομικά, η ενεργός αναμονή έχει περιοριστεί στο να πάρουν οι `wait()`, `signal()` το ανάλογο κλείδωμα. Η διάρκεια της ενεργού αναμονής είναι ανεξάρτητη του μεγέθους του κρίσιμου τμήματος στις διεργασίες.

δ. (10%) *Υλοποιήστε σχήμα συγχρονισμού που θα προσομοιώνει τη συμπεριφορά πελατών μιας τράπεζας ως εξής: Η τράπεζα έχει K καθίσματα στην αίθουσα αναμονής και ένα γκισέ εξυπηρέτησης. Οι πελάτες της τράπεζας μπορούν να δουν από το παράθυρο αν υπάρχουν ελεύθερα καθίσματα. Αν δεν υπάρχουν, πηγαίνουν έναν περίπατο (`take_a_walk()`) και ξαναπροσπαθούν αργότερα. Αν υπάρχουν, εισέρχονται στην αίθουσα αναμονής και προσπαθούν να εξυπηρετηθούν – ένας κάθε φορά – στο γκισέ. Ο πελάτης εξυπηρετείται καλώντας την `make_transaction()`. Χρησιμοποιήστε μοιραζόμενες μεταβλητές και σημαφόρους για τη λύση σας. Μπορείτε να δουλέψετε κάνοντας όποιες αλλαγές θεωρείτε αναγκαίες στα σημεία που υποδηλώνονται με . . . στο τμήμα κώδικα που ακολουθεί:*

```
. . .
void bank_client()
{
    while (1) {
        . . .
        if (. . .) { /* if seats available */
            . . .
            make_transaction();
            . . .
            break;
        }
        else {
            . . .
            take_a_walk();
        }
    }
    return_home();
}
```

Χρησιμοποιούμε σημαφόρο `window` ώστε κάθε πελάτης να ελέγχει ατομικά αν χωράει στην τράπεζα και αν να μπαίνει. Χρησιμοποιούμε σημαφόρο `teller` ώστε μόνο ένας πελάτης να εξυπηρετείται στο ταμείο κάθε φορά.

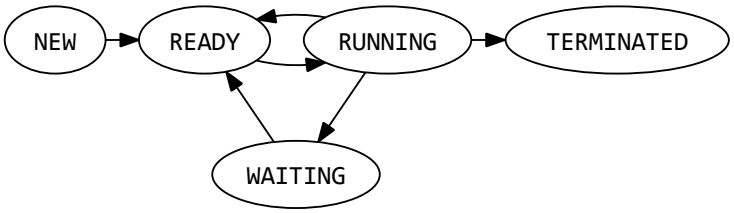
```
window = semaphore(1);
teller = semaphore(1);
shared int people_in_bank = 0;
void bank_client()
{
    while (1) {
        wait(window);
        if (people_in_bank < K) { /* if seats available */
            ++people_in_bank;
            signal(window);
            wait(teller);
            make_transaction();
            signal(teller);
            wait(window);
            --people_in_bank;
            signal(window);
            break;
        }
        else {
            signal(window);
            take_a_walk();
        }
    }
    return_home();
}
```

Επεξήγηση της απάντησης: Η χρήση σημαφόρου αρχικοποιημένου στο K δεν θα ήταν χρήσιμη σε αυτή την περίπτωση, γιατί μια διεργασία δεν μπορεί να ξέρει αν θα μπλοκάρει εκ των προτέρων, πριν εκτελέσει `wait()`.
Επίσης, η παραπάνω λύση δεν λαμβάνει υπόψη τη *σειρά* με την οποία εξυπηρετούνται οι πελάτες μέσα στην τράπεζα, είναι πιθανό π.χ. ο τελευταίος που μπήκε να πάρει πρώτος το σημαφόρο `teller`, και να εξυπηρετηθεί.

Θέμα 3 (25%)

α. (5%) *Σχεδιάστε ενδεικτικό διάγραμμα καταστάσεων διεργασίας, με τουλάχιστον τις καταστάσεις NEW, RUNNING, WAITING, READY, TERMINATED.*

Ένα ενδεικτικό διάγραμμα καταστάσεων διεργασίας παρουσιάζεται στο ακόλουθο σχήμα.



β. (10%) *Αναφέρετε μία αιτία για κάθε μετάβαση. Για κάθε μετάβαση, δώστε ενδεικτικό παράδειγμα/τμήμα κώδικα C σε UNIX στο οποίο μπορεί να συμβεί.*

- NEW → READY: αποδοχή της διεργασίας για εκτέλεση στο σύστημα. Παράδειγμα: `fork();` στη γονική διεργασία. Η διεργασία δημιουργείται (NEW) και γίνεται έτοιμη προς εκτέλεση (READY) μέσα στη `fork()`, από την οποία θα επιστρέψει όταν επιλεγεί προς εκτέλεση.
- RUNNING → READY: εκπνοή του κβάντου χρόνου διεργασίας, διακοπή της και επαναφορά της στην ουρά έτοιμων διεργασιών. Παράδειγμα: οποιαδήποτε υπολογιστικά απαιτητική αλληλουχία εντολών, π.χ: `s = 0; for (i = 0; i < 100000; i++) s += i;` Αν ο υπολογισμός απαιτεί πάνω από ένα κβάντο χρόνου, η διεργασία θα διακοπεί και θα υποστεί μετάβαση από RUNNING σε READY.
- RUNNING → WAITING: πραγματοποίηση κλήσης συστήματος που μπλοκάρει, π.χ. Ε/Ε από το δίσκο, `read(fd, buf, n);` ή εκτέλεση κλήσης `sleep(60)`.
- WAITING → READY: ολοκλήρωση διαδικασίας Ε/Ε, π.χ. ολοκλήρωση ανάγνωσης μπλοκ από το δίσκο στην κύρια μνήμη. Ο δίσκος προκαλεί διακοπή υλικού, κι η διεργασία που περίμενε τα δεδομένα μεταβαίνει από WAITING σε READY. Παράδειγμα: Η κλήση `read()` της προηγούμενης περίπτωσης, στην οποία βρίσκεται η διεργασία όταν γίνεται WAITING → READY.
- READY → RUNNING: επιλογή από τον χρονοδρομολογητή για εκτέλεση στη CPU, context switch σε αυτή τη διεργασία. Παράδειγμα: Ίδιο με των περιπτώσεων RUNNING → READY, RUNNING → WAITING. Είτε η διεργασία είχε μπλοκάρει λόγω Ε/Ε, είτε είχε διακοπεί λόγω εκπνοής του κβάντου χρόνου, τελικά θα κάνει READY → RUNNING.
- RUNNING → TERMINATED: κλήση της `exit()` από την τρέχουσα διεργασία. Στο UNIX η κατάσταση TERMINATED αντιστοιχεί σε διεργασία zombie που παραμένει στον πίνακα διεργασιών μέχρι ο πατέρας της να κάνει `wait()`. Παράδειγμα: `exit();`

γ. (5%) *Περιγράψτε τους αλγορίθμους χρονοδρομολόγησης FCFS και SJF. Αναφέρετε πλεονεκτήματα και μειονεκτήματα καθενός από τους δύο.*

Ο αλγόριθμος FCFS επιλέγει από την ουρά έτοιμων διεργασιών τη διεργασία που εισήλθε στην ουρά πρώτη. Πλεονέκτημα: Απλότητα στην υλοποίηση. Μειονέκτημα: Σε περίπτωση που έχει καταφθάσει πρώτη μία διεργασία με πολύ μεγάλο ξέσπασμα ΚΜΕ (CPU burst) τότε ο χρόνος αναμονής

των υπολοίπων διεργασιών θα είναι πολύ μεγάλος (φαινόμενο convoy). Γενικά, ο μέσος χρόνος αναμονής του αλγορίθμου FCFS μπορεί να είναι πολύ μεγάλος και με μεγάλη διακύμανση.
Ο αλγόριθμος SJF επιλέγει από την ουρά έτοιμων διεργασιών τη διεργασία με το μικρότερο επόμενο ξέσπασμα ΚΜΕ. Πλεονέκτημα: Ο αλγόριθμος οδηγεί σε μικρούς χρόνους αναμονής. Μειονεκτήματα: Είναι δύσκολη η πρόβλεψη του χρόνου του επόμενου ξεσπάσματος ΚΜΕ. Ο αλγόριθμος μπορεί να οδηγήσει μία διεργασία σε λιμοκτονία, αν η πρόβλεψη για το επόμενο ξέσπασμα ΚΜΕ της συγκεκριμένης διεργασίας είναι μονίμως μικρότερη από την πρόβλεψη για κάποια άλλη διεργασία.

δ. (5%) *Σε τι διαφέρει η διακοπτή από τη μη-διακοπτή χρονοδρομολόγηση; Αναφέρετε ένα πλεονέκτημα κι ένα μειονέκτημα για κάθε μία από τις δύο στρατηγικές. Έστω ένας υπολογιστικός κόμβος, στον οποίο χρήστες υποβάλλουν μαζικά εργασίες προς εκτέλεση. Ποια από τις δύο θα διαλέγατε για το ΛΣ που εκτελείται εκεί;*

Στη διακοπτή χρονοδρομολόγηση ο χρονοδρομολογητής καλείται μόνο όταν: α) μία διεργασία μεταβεί από κατάσταση RUNNING σε κατάσταση WAITING (π.χ. με μία κλήση συστήματος που θα οδηγήσει σε Ε/Ε) και β) Όταν μία διεργασία τερματίσει. Πλεονεκτήματα: Ευκολία στην υλοποίηση. Περιορισμένες ανάγκες για συγχρονισμό μεταξύ των διεργασιών. Μειονέκτημα: Μπορεί να οδηγήσει σε μειωμένη αποκρισιμότητα του συστήματος (π.χ. μία διεργασία που εκτελεί υπολογισμούς στη CPU επί ώρες, δεν θα αντικατασταθεί από άλλη διεργασία, με αποτέλεσμα οι υπόλοιπες διεργασίες του συστήματος να μη σημειώνουν καμμία πρόοδο).
Στη διακοπτή χρονοδρομολόγηση ο χρονοδρομολογητής καλείται επιπλέον των περιπτώσεων α) και β) και όταν: γ) συμβεί μία διακοπή υλικού (π.χ. από τον χρονιστή-timer του συστήματος) και δ) όταν μία διεργασία μεταβεί από κατάσταση WAITING σε κατάσταση READY (π.χ. μετά από την ολοκλήρωση κάποιας λειτουργίας Ε/Ε). Πλεονεκτήματα: Υψηλή αποκρισιμότητα. Μειονεκτήματα: Δυσκολία στην υλοποίηση. Ανάγκη για συγχρονισμό μεταξύ των διεργασιών, καθώς δεν είναι προβλέψιμα τα σημεία στα οποία θα γίνει εναλλαγή διεργασιών.
Σε έναν υπολογιστικό κόμβο μεγαλύτερη προτεραιότητα έχει η υψηλή χρησιμοποίηση της ΚΜΕ ώστε οι υπολογιστικές εργασίες να ολοκληρώνονται με το μεγαλύτερο δυνατό ρυθμό. Για αυτό το λόγο προσπαθούμε να περιορίσουμε τους κύκλους της ΚΜΕ που καταναλώνονται σε μη υπολογιστικό κώδικα, όπως είναι ο κώδικας του χρονοδρομολογητή. Εφόσον ο χρονοδρομολογητής καλείται σπανιότερα στη μη-διακοπτή χρονοδρομολόγηση, αυτή είναι η προτιμότερη στρατηγική για αυτή την περίπτωση υπολογιστικού συστήματος.

Θέμα 4 (25%)

α. (10%) *Έστω ΛΣ εικονικής μνήμης με σελιδοποίηση που ακολουθεί το μοντέλο* `fork()`/`exec()` *με Copy-On-Write (COW) για τη δημιουργία νέων διεργασιών. Το μέγεθος σελίδας είναι 4096 bytes. Η λίστα των ελεύθερων πλαισίων είναι* `303`, `127`, `309`, `310`, `311`, `312`, `313`, `314`, `...`
Δίνεται ο πίνακας σελίδων της υπό εκτέλεση διεργασίας `P0`*:*

page	*valid*	*perms*	*frame*
0	*i*		
1	*v*	*rx*	*128*
2	*v*	*r*	*301*
3	*v*	*rw*	*129*
4	*v*	*rw*	*126*

1. *Τι μέγεθος έχει ο εκτελέσιμος κώδικας της διεργασίας, και γιατί;*
2. *Ποιες σελίδες αποθηκεύουν* σταθερές*, π.χ. strings, για τη διεργασία; Τι θα γίνει αν επιχειρήσει να γράψει σε αυτές;*

3. *Τι page faults θα συμβούν, ποιοι θα είναι οι πίνακες σελίδων για τις `P0`, `P1`, `P2` και τι θα κάνει το ΛΣ όταν οι διεργασίες επιχειρήσουν τα εξής, το ένα μετά το άλλο; Απαντήστε για κάθε βήμα χωριστά.*

   1. *Η `P0` εκτελεί `fork()` και γεννιέται η `P1`.*
   2. *Η `P1` διαβάζει από τη διεύθυνση `8512`.*
   3. *Η `P0` γράφει στη διεύθυνση `12800`.*
   4. *Η `P1` γράφει στη διεύθυνση `12928`.*
   5. *Η `P1` εκτελεί `fork()` και γεννιέται η `P2`.*
   6. *Η `P2` γράφει στη διεύθυνση `4224`.*

Περιγράψτε επαρκώς όποιες παραδοχές κάνετε.

Δεν μας δίνεται ο χάρτης μνήμης για τη διεργασία. Υποθέτουμε την πιο απλή περίπτωση, ότι ολόκληρη η μνήμη της διεργασίας είναι ιδιωτική (δεν υπάρχουν σελίδες σημειωμένες ως `MAP_SHARED`). Οπότε:

1. Μόνο μία σελίδα έχει δικαίωμα `x`, οπότε μόνο μία σελίδα μπορεί να περιέχει εκτελέσιμο κώδικα. Άρα ο εκτελέσιμος κώδικας της διεργασίας είναι μεγέθους $\leq$4KB.
2. Οι σταθερές δεν είναι εκτελέσιμος κώδικας, και πιθανότατα αποθηκεύονται σε σελίδα χωρίς δικαίωμα εγγραφής, ώστε να εντοπίζεται προσπάθεια αλλαγής τους από λανθασμένο ή κακόβουλο κώδικα. Μόνο η σελίδα `1` ικανοποιεί τις προδιαγραφές.
3. Δίνουμε τον πίνακα σελίδων κάθε διεργασίας σε κάθε βήμα:

   1. Σε `fork()` με COW αντιγράφεται ο πίνακας σελίδων ώστε να χρησιμοποιούνται τα ίδια πλαίσια από πατέρα και παιδί, *και οι δύο* χάνουν το δικαίωμα `w` στον πίνακα σελίδων.

P0			
page	**valid**	**perms**	**frame**
0	i		
1	v	rx	128
2	v	r	301
3	v	r	129
4	v	r	126

P1			
page	**valid**	**perms**	**frame**
0	i		
1	v	rx	128
2	v	r	301
3	v	r	129
4	v	r	126

   2. Η διεύθυνση `8512` είναι στη σελίδα `2`, γιατί $\lfloor \frac{8512}{4096} \rfloor = 2$. Έχει δικαίωμα `r`, η εντολή `load` εκτελείται χωρίς να συμβεί page fault.

   3. Η διεύθυνση `12800` είναι στη σελίδα `3`. Η `P0` δεν έχει δικαίωμα `w`, συμβαίνει page fault κατά την εκτέλεση της εντολής `store`, το ΛΣ βλέπει από το χάρτη μνήμης ότι έχει γίνει COW, αναθέτει νέο πλαίσιο στη διεργασία από τα ελεύθερα, το `303`, και δίνει δικαίωμα `rw` για τη συγκεκριμένη σελίδα. Θεωρούμε ότι ο πίνακας σελίδων της `P0` δεν αλλάζει ακόμη. Θα μπορούσε, επειδή είναι η μοναδική που πλέον έχει αναφορά στο πλαίσιο `129` να αποκτήσει πρόσβαση `rw` στον πίνακα σελίδων της, ώστε να αποφευχθεί ένα ακόμη page fault. Υποθέτουμε ότι αυτό δεν συμβαίνει.

   4. Η `P1` δεν έχει δικαίωμα `w` στη σελίδα 3. Συμβαίνει page fault, είναι η μόνη που έχει δεδομένα στο πλαίσιο `129`, το ΛΣ της δίνει δικαίωμα `rw` και επανεκκινεί την εντολή `store`. Μετά τα δύο τελευταία βήματα, οι πίνακες σελίδων έχουν γίνει:

P0			
page	**valid**	**perms**	**frame**
0	i		
1	v	rx	128
2	v	r	301
3	v	rw	303
4	v	r	126

P1			
page	**valid**	**perms**	**frame**
0	i		
1	v	rx	128
2	v	r	301
3	v	rw	129
4	v	r	126

5. Έχουμε `fork()` με COW, η `P2` αποκτά πίνακα σελίδων αντίγραφο του πίνακα της `P1`, κανείς δεν έχει δικαίωμα εγγραφής:

P0			
page	**valid**	**perms**	**frame**
0	i		
1	v	rx	128
2	v	r	301
3	v	rw	303
4	v	r	126

P1			
page	**valid**	**perms**	**frame**
0	i		
1	v	rx	128
2	v	r	301
3	v	r	129
4	v	r	126

P2			
page	**valid**	**perms**	**frame**
0	i		
1	v	rx	128
2	v	r	301
3	v	r	129
4	v	r	126

6. Η `P2` επιχειρεί να γράψει στη σελίδα `1`. Δεν έχει δικαίωμα `w`, συμβαίνει page fault, το ΛΣ ξυπνά, συμβουλεύεται τον χάρτη μνήμης του, βλέπει ότι η σελίδα `1` είναι σελίδα κώδικα και η διεργασία *δεν* έχει δικαίωμα εγγραφής εκεί. Ο πυρήνας στέλνει σήμα `SIGSEGV` γιατί επιχειρείται μη επιτρεπτή πρόσβαση. Αν η διεργασία δεν το χειρίζεται, σκοτώνεται.

β. (5%)

1. *Τι είναι ο Τρέχων Κατάλογος (current working directory), για μια διεργασία; Πού κρατάει το ΛΣ την πληροφορία για τον τρέχοντα κατάλογο μιας διεργασίας;*
2. *Έστω δύο έργα/tasks `T0`, `T1` στο Linux που εκτελούνται με τρέχοντα κατάλογο τον ριζικό ("/"), ο οποίος περιέχει αρχείο `/file1` και άδειο κατάλογο `/dir1`. Το `T0` αλλάζει τον τρέχοντα κατάλογο εκτελώντας `chdir("/dir1")`. Έπειτα, το `T1` εκτελεί `open("file1", ...)`. Τι θα συμβεί αν τα `T0`, `T1` είναι (α) χωριστές διεργασίες, (β) νήματα της ίδιας διεργασίας, (γ) διεργασίες πατέρας-παιδί;*

1. Είναι ο κατάλογος από τον οποίο ξεκινά η επίλυση σχετικών μονοπατιών π.χ. `dir/file1.txt`, ή `afile.bin`, όταν η διεργασία αναφέρεται σε αυτά. Το ΛΣ κρατάει αυτή την πληροφορία στο PCB για τη διεργασία.
2. Το ερώτημα είναι αν η κλήση της `chdir()` από το T0 επηρεάζει την κλήση `open()` του T1, αν το σχετικό μονοπάτι θα επιλυθεί ξεκινώντας από το `/dir1` ή από το `/`. Τρεις περιπτώσεις: (α) κάθε διεργασία έχει το δικό της PCB. Όταν το T0 κάνει `chdir()` δεν αλλάζει τον ΤΚ του T1, οπότε η `open()` επιτυγχάνει. (β) Τα νήματα ανήκουν στην ίδια διεργασία, έχουν κοινό PCB άρα κοινό τρέχοντα κατάλογ. Η `chdir()` αλλάζει τον κοινό ΤΚ, οπότε η `open()` αποτυγχάνει με `errno = ENOENT`. (γ) Ακριβώς όπως στην περίπτωση (α), δεν υπάρχει τίποτε ιδιαίτερο, δεν μοιράζονται το PCB, άρα ούτε τον ΤΚ.

γ. (10%) *Σε σύστημα αρχείων το FCB (i-node) περιέχει μετρητή (ακέραιο αριθμό) για την υλοποίηση* μη συμβολικών *συνδέσμων (hard links).*
Απαντήστε συνοπτικά *στα εξής:*

1. *Τι συμβαίνει στα ανοιχτά αρχεία μιας διεργασίας όταν αυτή εκτελέσει `exit()`, και τι όταν σκοτωθεί από σήμα;*

2. *Ο ελεύθερος χώρος στο σύστημα αρχείων που είναι προσαρτημένο στη θέση `/mnt` είναι 3GB. Το `/mnt/data1`, `/mnt/data2` είναι hard links προς το ίδιο αρχείο, το οποίο περιέχει 2GB μη μηδενικά δεδομένα.*
   Εξηγήστε πώς μπορεί να συμβεί το εξής σενάριο:
   1. *Εκτελούμε `rm /mnt/data1`. Ο ελεύθερος χώρος στο `/mnt` παραμένει αμετάβλητος.*
   2. *Εκτελούμε `rm /mnt/data2`. Ο ελεύθερος χώρος στο `/mnt` παραμένει αμετάβλητος.*
   3. *Εκτελούμε κατάλληλη εντολή `kill`. Ο ελεύθερος χώρος στο `/mnt` γίνεται 5GB. Τι συνέβη;*
3. *Τι θα συνέβαινε στο παραπάνω σενάριο αν υπήρχε ο συμβολικός δεσμός (soft link) `/mnt/data3` → `/mnt/data2`;*

1. Όταν μια διεργασία τερματίσει εκτελώντας `exit()` όλα τα αρχεία που είχε ανοιχτά κλείνουν, με ευθύνη του πυρήνα. *Ακριβώς* το ίδιο συμβαίνει κι όταν σκοτωθεί από σήμα: ο πυρήνας απελευθερώνει όλους τους πόρους που η διεργασία κατείχε, ώστε το σύστημα να παραμένει συνεπές.
2. Το σενάριο εξηγείται ως εξής:
   1. Το `/mnt/data1` είναι hard link. Η `rm` εκτελεί κλήση συστήματος `unlink()` σβήνει το όνομα `/mnt/data1`. Υπάρχουν ακόμη hard links προς το αρχείο, το inode δεν απελευθερώνεται, ο ελεύθερος χώρος παραμένει αμετάβλητος.
   2. Το `/mnt/data2` ήταν το τελευταίο hard link. Το inode θα μπορούσε να διαγραφεί, αλλά δεν διαγράφεται γιατί μία διεργασία, έστω με PID `1001` είχε ανοίξει το αρχείο όσο είχε ακόμη όνομα να δείχνει στο inode, κάνοντας `open("/mnt/data1", ...)`.
   3. Ο χρήστης έτρεξε `kill 1001`, σκοτώνοντας τη διεργασία. Όλα τα ανοιχτά της αρχεία κλείνουν, το inode δεν έχει hard links προς αυτό και δεν αναφέρεται σε αυτό κανένα ανοιχτό αρχείο, το σύστημα αρχείων το απελευθερώνει, κι ο ελεύθερος χώρος αυξάνεται στα 5GB.
3. Δεν θα άλλαζε απολύτως τίποτε. Το `/mnt/data3` θα έμενε στο σύστημα αρχείων ως συμβολικός δεσμός που θα έδειχνε σε ανύπαρκτο όνομα (dangling link).