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 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 (25%)

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

1. *Τι κάνουν οι κλήσεις `kill(pid,SIGSTOP)` και `signal(SIGINT,handler)` στο UNIX;*
2. *Τι σημαίνει στην κλήση `n=read(fd,buf,cnt)` το όρισμα `cnt`; Τι τιμές μπορεί να έχει η μεταβλητή `n` μετά την επιστροφή της `read()`;*
3. *Τι κάνει η κλήση `pid=wait(&status)`; Έστω ότι μετά την κλήση της από τη διεργασία με PID `1000` ισχύει `pid==1002`, `WIFEXITED(status)==1`, `WEXITSTATUS(status)==101`. Τι σχέση είχαν οι διεργασίες `1000`, `1002` και ποια ήταν η τελευταία κλήση συστήματος της `1002`;*
4. *Τι παθαίνει μια διεργασία που καλεί την `read()` σε άδειο pipe όταν είναι ανοιχτό το άκρο εγγραφής;*

1. Η πρώτη στέλνει το σήμα `SIGSTOP` στη διεργασία `pid`, η δεύτερη κανονίζει ώστε όταν παραληφθεί σήμα `SIGINT` να κληθεί η συνάρτηση χειρισμού `handler` για να το χειριστεί.
2. Σημαίνει τον μέγιστο αριθμό bytes που ζητάμε να διαβαστούν από τον περιγραφητή `fd`. Σε αποτυχία της κλήσης `n==-1`, σε EOF `n==0`, αλλιώς η `n` είναι από `1` έως `cnt`.
3. Μπλοκάρει την καλούσα διεργασία μέχρι να πεθάνει ένα από τα παιδιά της. Η διεργασία `1002` ήταν παιδί της `1000`. Η τιμή του `status` δείχνει ότι τερμάτισε κανονικά με κωδικό `101`, οπότε η τελευταία κλήση της ήταν η `exit(101)`.
4. Μπλοκάρει μέχρι να γράψει κάποιος στο άκρο εγγραφής, ή να το κλείσουν όσοι το έχουν ανοιχτό.

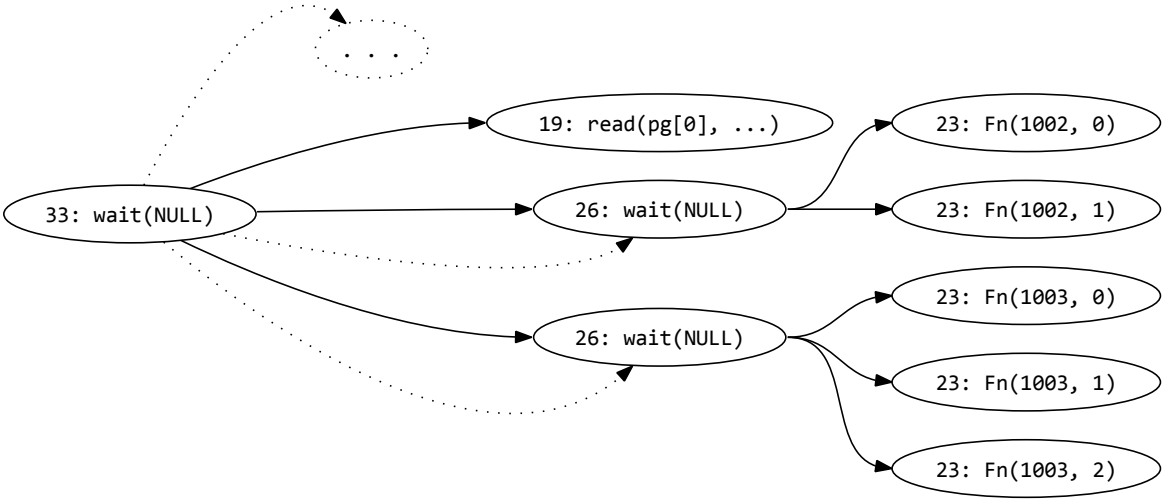
β. (12%) *Σχεδιάστε το δέντρο διεργασιών στην τελική του μορφή, όταν δηλαδή όλες οι διεργασίες έχουν φτάσει σε μόνιμη κατάσταση. Εξηγήστε συνοπτικά πώς προκύπτει.*

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   int pg[2];
2
3   void handler(int signum)
4   { exit(4); Fn(0, -1); }
5
6   int main(void)
7   {
8       int i, j, n;
9       char c[4];
10      pid_t pid[4];
11
12      signal(SIGUSR1, handler);
13
14      pipe(pg);
15      for (i = 0; i < 4; i++) {
16          pid[i] = fork();
17          if (pid[i] == 0) {
18                  if (i == 1) sleep(5);
19                  n = read(pg[0], c, 2);
20                  for (j = 0; j < i; j++) {
21                      pid[i] = fork();
22                      if (pid[i] == 0)
23                          Fn(getppid(), j);
24                  }
25                  for (j = 0; j < i; j++)
26                      wait(NULL);
27                  exit(2);
28          }
29      }
30
31      kill(pid[0], SIGUSR1);
32      for (i = 0; i < 4; i++) {
33          wait(&n);
34          write(pg[1], c, WEXITSTATUS(n));
35      }
36      Fn(i, -1);
37      return 0;
38  }
```

Το δέντρο διεργασιών στην τελική κατάσταση φαίνεται στο ακόλουθο σχήμα.



Ο πατέρας στέλνει `SIGUSR1` στο πρώτο παιδί, το οποίο πεθαίνει με τιμή επιστροφής `4`. Ο πατέρας γράφει τόσα bytes στο κοινό pipe `pg`, οπότε δύο από τα παιδιά του περνούν τη γραμμή 19, διαβάζοντας 2 bytes το καθένα. Το παιδί `i==1` χάνει το race, λόγω του sleep της γραμμής 18 και μπλοκάρει για πάντα στη γραμμή 19. Στο παραπάνω σχήμα, ο κόμβος του πρώτου παιδιού υπάρχει μόνο για να φανεί η αποστολή του σήματος, η συγκεκριμένη διεργασία δεν είναι στο δέντρο σε μόνιμη κατάσταση.

Επεξήγηση της απάντησης: Αρχικά (γραμμή 12) κανονίζεται κάθε φορά που παραλαμβάνεται σήμα `SIGUSR1` να εκτελείται η συνάρτηση χειρισμού `handler`.

Ο πατέρας κατασκευάζει ένα pipe, και 4 παιδιά. Έστω `1000`, `1001`, `1002`, `1003` τα PIDs των παιδιών. Στέλνει το SIGUSR1 στο παιδί `1000`, το οποίο εκτελεί τον `handler` και τερματίζει με κωδικό επιστροφής `4`. Τα παιδιά `1002`, `1003` τελικά μπλοκάρουν στη γραμμή 19 λόγω του άδειου pipe, ενώ το παιδί `1001` κοιμάται στη γραμμή 18. Ο πατέρας μαθαίνει τον κωδικό επιστροφής του παιδιού `1000`, οπότε γράφει 4 bytes στο pipe. Επειδή κάθε παιδί θέλει να διαβάσει 2 bytes, μόνο δύο παιδιά περνούν τη γραμμή 19. Το άλλο – υποθέτουμε το παιδί `1001` λόγω του sleep() στη 18 – θα μπλοκάρει για πάντα στη γραμμή 19. Τα `1002`, `1003` γεννούν 2 και 3 παιδιά αντίστοιχα, και μπλοκάρουν στην 26, γιατί κανένα από τα παιδιά τους δεν έχει πεθάνει. Ο πατέρας μπλοκάρει για πάντα στη γραμμή 33 στη δεύτερη επανάληψη του βρόχου (γραμμή 32), γιατί δεν πεθαίνει ποτέ δεύτερο παιδί του.

Θέμα 2 (25%)

α. (10%) *Περιγράψτε το πρόβλημα του κρίσιμου τμήματος (critical section) και το πρόβλημα της σειριοποίησης (ordering). Δώστε από ένα παράδειγμα κώδικα στο οποίο εμφανίζονται. Αναφέρετε ένα μηχανισμό που επιλύει το καθένα από αυτά.*

Το πρόβλημα του κρίσιμου τμήματος αφορά σε τμήματα κώδικα μέσα στα οποία μπορεί να βρίσκεται μόνο μία διεργασία τη φορά. Παράδειγμα: δύο διεργασίες προσπαθούν να προσπελάσουν μία κοινή μεταβλητή. Το πρόβλημα της σειριοποίησης εμφανίζεται όταν η σημασιολογία ενός προγράμματος που τρέχει σε δύο διεργασίες A, B απαιτεί συγκεκριμένη ενέργεια της A να εκτελεστεί πριν από συγκεκριμένη ενέργεια της B.

Παράδειγμα κώδικα όπου υπάρχει κρίσιμο τμήμα *και* ανάγκη σειριοποίησης:

```
shared int count;
void proc1(void)
{
    count = 0;
    ...
    count++;
    ...
}
```

```
void proc2(void)
{
    ...
    count++;
    ...
}
```

Η πρόσβαση των δύο διεργασιών στη μεταβλητή `count` για να αυξήσουν την τιμή της αποτελεί κλασική περίπτωση του προβλήματος του κρίσιμου τμήματος.
Επίσης, αν η διεργασία που εκτελεί τη ρουτίνα `proc2()` προσπελάσει τη μεταβλητή `count` πριν η διεργασία που εκτελεί τη ρουτίνα `proc1` την αρχικοποιήσει, τότε η εκτέλεση θα έχει ακαθόριστο αποτελέσμα.
Το πρόβλημα του κρίσιμου τμήματος μπορεί να αντιμετωπιστεί με τη χρήση κλειδωμάτων (locks), το πρόβλημα της σειριοποίησης με τη χρήση σημαφόρων.

β. (5%) *Υλοποιήστε σχήμα συγχρονισμού που εξασφαλίζει ότι το πολύ N διεργασίες μπορούν να εκτελούν ένα συγκεκριμένο κομμάτι κώδικα ταυτόχρονα. Μπορείτε να χρησιμοποιήσετε λειτουργίες `signal()` και `wait()` σε κατάλληλα αρχικοποιημένους σημαφόρους κι όποιες μεταβλητές μοιραζόμενες ανάμεσα στις διεργασίες χρειάζεστε. Δουλέψτε κάνοντας όποιες αλλαγές θεωρείτε αναγκαίες στα σημεία που υποδηλώνονται με ... στο τμήμα κώδικα που ακολουθεί:*

```
. . .
void work()
{
    before_work();
    . . .
    only_up_to_N_processes_do_this_work();
    . . .
    after_work();
}
```

Θα χρησιμοποιήσουμε έναν σημαφόρο αρχικοποιήμενο στην τιμή N.

```
room = semaphore(N);
void work()
{
    before_work();

    wait(room);
    only_up_to_N_processes_do_this_work();
    signal(room);

    after_work();
}
```

γ. (10%) *Υλοποιήστε σχήμα συγχρονισμού που θα προσομοιώνει τη διαδικασία επιβίβασης ταξιδιωτών σε αεροπλάνα, μέσω μιας αίθουσας αναμονής N θέσεων. Συνεχώς καταφτάνουν ταξιδιώτες, καθένας από τους οποίους εκτελεί τη συνάρτηση `traveler()`. Στην αίθουσα αναμονής μπορούν να παραβρίσκονται μέχρι N ταξιδιώτες. Κάθε ταξιδιώτης που μπαίνει στην αίθουσα αναμονής δίνει το διαβατήριό του για έλεγχο (συνάρτηση `pass_passport_check()`). Όταν συμπληρωθούν N ταξιδιώτες που έχουν περάσει τον έλεγχο διαβατηρίων, τότε όλοι μαζί εγκαταλείπουν την αίθουσα και επιβιβάζονται σε λεωφορείο (συνάρτηση `take_bus_to_plane()`), αφήνοντας την αίθουσα άδεια, διαθέσιμη για την επόμενη ομάδα ταξιδιωτών. Μπορείτε να δουλέψετε κάνοντας όποιες αλλαγές θεωρείτε αναγκαίες στα σημεία που υποδηλώνονται με ... στο τμήμα κώδικα που ακολουθεί:*

```
. . .
void traveler()
{
    . . .
    pass_passport_check();
    . . .
    take_bus_to_plane();
}
```

Οι σημαφόροι `enter`, `exit` ελέγχουν την κίνηση των επιβατών κατά την είσοδο και την έξοδο από την αίθουσα αναμονής, ο σημαφόρος `mutex` προστατεύει τη μοιραζόμενη μεταβλητή `count`.

```
enter = semaphore(N);
exit = semaphore(0);
mutex = semaphore(1);
shared int count;

void traveler()
{
    wait(enter);
    wait(mutex);
    count++;
    if (count == N)
        signal(exit);
    signal(mutex);

    pass_passport_check();

    wait(exit);
    wait(mutex);
    count--;
    if (count > 0)
        signal(exit);
    else
        for (i = 0; i < N; i++)
            signal(enter);
    signal(mutex);

    take_bus_to_plane();
}
```

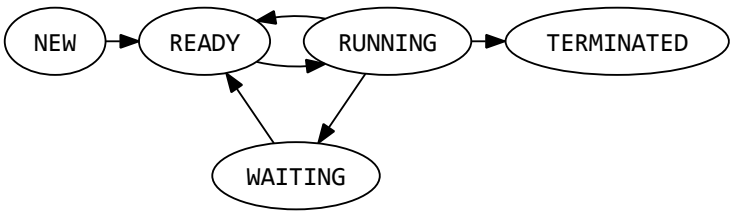
Θέμα 3 (25%)

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

Ένα ενδεικτικό διάγραμμα καταστάσεων διεργασίας παρουσιάζεται στο ακόλουθο σχήμα.

β. (5%) *Αναφέρετε μια αιτία για κάθε μετάβαση.*

- NEW → READY: αποδοχή της διεργασίας για εκτέλεση στο σύστημα.
- READY → RUNNING: επιλογή από τον χρονοδρομολογητή για εκτέλεση στη CPU, context switch σε αυτή τη διεργασία.



- RUNNING → READY: εκπνοή του κβάντου χρόνου διεργασίας, διακοπή της και επαναφορά της στην ουρά έτοιμων διεργασιών.
- RUNNING → WAITING: πραγματοποίηση κλήσης συστήματος που μπλοκάρει, π.χ. Ε/Ε από το δίσκο, ή εκτέλεση κλήσης `sleep()`.
- 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.
- 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()`.

γ. (5%) *Η τεχνολογική τάση οδηγεί σε αλγορίθμους διακοπτής χρονοδρομολόγησης που θα είναι σε θέση να διακόπτουν μια διεργασία είτε αυτή είναι σε χώρο χρήστη είτε είναι σε χώρο πυρήνα. Γιατί πιστεύετε ότι συμβαίνει αυτό; Πώς επηρεάζει η παραπάνω τάση το σχεδιασμό του συγχρονισμού;*

Η χρήση διακοπτής χρονοδρομολόγησης αυξάνει την αποκρισιμότητα του συστήματος: μια διεργασία διακόπτεται λόγω διακοπής υλικού (timer interrupt) ακόμη κι όταν δεν έχει εκτελέσει κλήση συστήματος που μπλοκάρει. Ιστορικά, η δυνατότητα αυτή υλοποιήθηκε για διακοπή διεργασίων όταν εκτελούσαν κώδικα χώρου χρήστη και μόνο. Για να βελτιωθεί περαιτέρω η αποκρισιμότητα (π.χ. εφαρμογές ήχου-εικόνας σε desktops), προστέθηκε η δυνατότητα διακοπής διεργασιών και όταν εκτελούν κώδικα στο χώρο πυρήνα ("preemptible kernel").
Η παραπάνω τάση περιπλέκει το σχεδιασμό του συγχρονισμού, καθώς δύο διεργασίες μπορεί να επιχειρήσουν πρόσβαση σε κοινές δομές δεδομένων τόσο δικές τους (χώρου χρήστη) όσο και του πυρήνα, ακόμη και σε μονοεπεξεργαστικά συστήματα.

δ. (10%) *Δίνεται το τμήμα προγράμματος C που ακολουθεί μετά το ζητούμενο.*
Περιγράψτε ένα σενάριο μετάβασης καταστάσεων της διεργασίας που τρέχει αυτόν τον κώδικα σε ΛΣ εικονικής μνήμης με σελιδοποίηση κατ'απαίτηση που εφαρμόζει (i) διακοπτή (ii) μη-διακοπτή πολιτική χρονοδρομολόγησης, με αναφορά σε συγκεκριμένες γραμμές. Θεωρήστε ότι ο χρήστης έχει δώσει `N` τέτοιο ώστε τα υπολογιστικά μέρη του κώδικα να είναι της τάξης των sec και άνω.

```
1   int main(int argc, char *argv[])
2   {
3       double *array;
4       int N, i, j;
5
6       printf("What is the size of the array?\n");
7       scanf("%d", &N);
8
9       array = malloc(N * sizeof(double));
10
11      for (i = 0; i < N; i++)
```

```
12          array[i] = i;
13
14      for (i = 0; i < N; i++)
15          for (j = 0; j < i; j++)
16              array[i] += sqrt(array[j]);
17
18      return 0;
19  }
```

Ένα δυνατό σενάριο μετάβασης καταστάσεων της διεργασίας με *διακοπτή* χρονοδρομολόγηση, θα ήταν το ακόλουθο. Για κάθε γραμμή δίνεται η (πιθανή) μετάβαση και το αίτιό της.

1. Γραμμή 6: RUNNING → WAITING → READY → RUNNING. Η διεργασία εκτελεί έξοδο, `write()`, μπορεί να μπλοκάρει.
2. Γραμμή 7: Ομοίως με την 6. Η διεργασία εκτελεί είσοδο από το χρήστη, `read()`, σίγουρα θα μπλοκάρει.
3. Γραμμή 11:RUNNING → WAITING → READY → RUNNING: Στη γραμμή 9 η διεργασία δέσμευσε (εικονική) μνήμη, στην οποία πιθανότατα δεν έχουν αποδοθεί πλαίσια φυσικής μνήμης. Στο σημείο αυτό την ακουμπάει, οπότε θα συμβεί page fault και το ΛΣ θα αναθέσει πλαίσια. Αν γι'αυτό χρειάζεται Ε/Ε (π.χ. μετακίνηση σελίδων στο δίσκο), η διεργασία πηγαίνει σε WAITING.
4. Γραμμή 16: Ομοίως με την 11, ειδικά αν εκτελούνται ταυτόχρονα άλλες διεργασίες στο σύστημα. Παρόλο που η διεργασία εκτελεί καθαρό υπολογισμό και δεν κάνει ποτέ κλήσεις συστήματος, μπορεί να πάει σε WAITING λόγω page fault κι ανάγκη αντικατάστασης σελίδας.
5. Γραμμές 14-16: RUNNING → READY → RUNNING. Επειδή ο υπολογισμός διαρκεί αρκετά δευτερόλεπτα κι έχουμε διακοπτή χρονοδρομολόγηση, η διεργασία θα διακοπεί λόγω εκπνοής κβάντου χρόνου.
6. Οποιαδήποτε γραμμή: Θεωρητικά, η διεργασία μπορεί να διακοπεί σε οποιοδήποτε σημείο, όμοια με το βρόχο 14-16, αν εκπνεύσει το κβάντο χρόνου.

Στην περίπτωση μη-διακοπτής χρονοδρομολόγησης δεν είναι δυνατές οι μεταβάσεις 5 και 6: Η διεργασία δεν θα διακοπεί από timer interrupt, παρά μόνο αν η ίδια ζητήσει Ε/Ε ή προκαλέσει page fault (software interrupt - trap).

Θέμα 4 (25%)

α. (5%) *Σε σύστημα διαχείρισης της φυσικής μνήμης, περιγράψτε το πρόβλημα του (i) εσωτερικού και του (ii) εξωτερικού κατακερματισμού. Ποιο από τα δύο επιλύει ο μηχανισμός της μετάφρασης με σελιδοποίηση και γιατί;*

Εσωτερικός κατακερματισμός είναι η ύπαρξη αχρησιμοποίητων περιοχών μνήμης *μέσα* σε περιοχές δεσμευμένες από διεργασίες, π.χ. γιατί οι διεργασίες δέσμευσαν περισσότερη μνήμη από όση τελικά χρειάστηκαν. Εξωτερικός κατακερματισμός είναι η ύπαρξη πολλών μικρών αδέσμευτων περιοχών μνήμης, οπότε δεν μπορεί να εξυπηρετηθεί αίτηση για δέσμευση μεγάλου ποσού συνεχόμενης μνήμης, παρόλο που ο συνολικά διαθέσιμος χώρος επαρκεί. Η σελιδοποίηση λύνει τον εξωτερικό κατακερματισμό: Οποιοσδήποτε αριθμός διάσπαρτων πλαισίων μπορεί να συναρμολογηθεί σε ενιαίο, συνεχή χώρο εικονικών διευθύνσεων και να αποδοθεί σε διεργασία.

β. (10%) *Έστω ΛΣ εικονικής μνήμης με σελιδοποίηση που ακολουθεί το μοντέλο `fork()`/`exec()` με COW για τη δημιουργία νέων διεργασιών. Απαντήστε αν οι ακόλουθες προτάσεις είναι αληθείς ή ψευδείς, με σύντομη αιτιολόγηση. Περιγράψτε επαρκώς όποιες παραδοχές κάνετε.*

1. *Μια διεργασία θέτει `var=5` και κάνει `fork()`. Το παιδί διαβάζει τη μεταβλητή `var` και τη βρίσκει να έχει τιμή `5`, αρκεί ο πατέρας να μην έχει προλάβει να την αλλάξει μετά το `fork()`.*

2. *Αν `fptr=&var` η διεύθυνση της `var` στον πατέρα, `cptr=&var` στο παιδί, τότε `fptr==cptr`.*
3. *Αμέσως μετά το `fork()`, κάθε ανάγνωση της `var` από τον πατέρα καταλήγει στην ίδια φυσική διεύθυνση με κάθε ανάγνωση της `var` από το παιδί.*
4. *Η εντολή `var=3` στον πατέρα προκαλεί page fault.*
5. *Η εντολή `var=3` στο παιδί προκαλεί page fault.*
6. *Όταν μια διεργασία επιχειρεί να γράψει σε διεύθυνση για την οποία η αντίστοιχη εγγραφή στον πίνακα σελίδων της έχει το permission bit* Write *σβηστό, τερματίζεται πάντα με Segmentation Fault.*
7. *Όταν μια διεργασία επιχειρεί να γράψει σε διεύθυνση για την οποία δεν υπάρχει έγκυρη εγγραφή στον πίνακα σελίδων της, τερματίζεται πάντα με Segmentation Fault.*
8. *Όταν μια διεργασία επιχειρεί να γράψει σε διεύθυνση για την οποία δεν υπάρχει έγκυρη περιοχή στον χάρτη μνήμης της, τερματίζεται πάντα με Segmentation Fault.*

1. Ψευδές. Το παιδί θα διαβάσει `var==5` ό,τι και να κάνει ο πατέρας μετά το `fork()`. Μετά το fork κάθε διεργασία ζει στο δικό της, ανεξάρτητο, απομονωμένο χώρο εικονικών διευθύνσεων.
2. Αληθές. Η μνήμη – ο χώρος εικονικών διευθύνσεων – του παιδιού είναι αντίγραφο της μνήμης – του χώρου εικονικών διευθύνσεων – του πατέρα και δημιουργείται κατά το `fork()`.
3. Αληθές. Με COW οι σελίδες και των δύο διεργασιών απεικονίζονται στο ίδιο πλαίσιο φυσικής μνήμης, μόνο με το permission bit Read αναμμένο. Υποθέτουμε ότι κανείς δεν έχει γράψει ακόμη στη `var`.
4. Αληθές για την πρώτη φορά που ο πατέρας επιχειρεί να γράψει στη `var`, ψευδές για όλες τις υπόλοιπες, οπότε έχει ιδιωτική σελίδα με δικαίωμα Write.
5. Ομοίως. Η κατάσταση πατέρα-παιδιού είναι συμμετρική μετά από `fork()` με COW.
6. Ψευδές. Είναι η περίπτωση πρώτης εκτέλεσης `var=3` από το παιδί ή τον πατέρα. Προκαλείται page fault και ανατίθεται νέα σελίδα.
7. Ψευδές. Μπορεί η αντίστοιχη σελίδα να έχει πάει στο δίσκο, οπότε δεν υπάρχει έγκυρη εγγραφή στον πίνακα σελίδων και προκαλείται page fault για να έρθει από εκεί.
8. Αληθές. Μια διεργασία δεν έχει δικαίωμα να διαβάσει διεύθυνση σε περιοχή μνήμης για την οποία δεν είναι ενήμερο το ΛΣ.

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

1. *Τι συνεπάγεται η εκτέλεση των παρακάτω λειτουργιών για τις δομές i-node που υπάρχουν στο σύστημα αρχείων και τους μετρητές που περιέχουν;*
   - *Δημιουργία νέου αρχείου*
   - *Δημιουργία νέου hard link προς υπάρχον αρχείο*
   - *Δημιουργία νέου soft link προς υπάρχον αρχείο*
2. *Ο ελεύθερος χώρος είναι F. Ο χρήστης εκτελεί `rmdata1`, το `data1` περιέχει 2GB δεδομένων. Αν ο ελεύθερος χώρος μετά είναι F', περιγράψτε ένα σενάριο όπου $F' \approx F$, ακόμη και μετά τη διαγραφή.*
3. *Δύο συστήματα αρχείων είναι προσαρτημένα (mounted) στα σημεία `/mnt1`, `/mnt2`. Με την εντολή `ls -i` βλέπουμε ότι τα αρχεία `/mnt1/a`, `/mnt2/b` έχουν το ίδιο i-node number. Αυτό σημαίνει ότι είναι hard links στο ίδιο αρχείο. Αληθές ή ψευδές;*

1. Οι αναφερόμενες λειτουργίες έχουν τις εξής παρενέργειες:
   - Δημιουργείται νέα δομή i-node, με μετρητή `cnt=1`.
   - Δεν δημιουργείται νέα δομή i-node, μόνο νέα εγγραφή στον κατάλογο που περιέχει το όνομα του νέου link, ο μετρητής του υπάρχοντος i-node γίνεται `++cnt`.

- Δημιουργείται νέα δομή i-node για το συμβολικό δεσμό, το αρχείο του συμβολικού δεσμού περιέχει το μονοπάτι (όνομα) προς το υπάρχον αρχείο, ο μετρητής του υπάρχοντος i-node δεν επηρεάζεται.

2. Το `data1` δεν είναι το μοναδικό hard link σε αυτό το i-node. Διαγραφή του `data1` μειώνει κατά ένα τον μετρητή `cnt`, αλλά επειδή ακόμη $\texttt{cnt} > 0$ το i-node δεν καταστρέφεται και τα μπλοκ δεδομένων συνολικού μεγέθους 2GB δεν απελευθερώνονται, οπότε ο ελεύθερος χώρος δεν επηρεάζεται.
3. Ψευδές. Τα ονόματα/αρχεία ανήκουν σε διαφορετικά συστήματα αρχείων, το καθένα ίσως σε διαφορετική συσκευή αποθήκευσης. Το κάθε σύστημα αρχείων έχει το δικό του διακριτό σύνολο από i-nodes κι ανεξάρτητη αρίθμηση. Η ταύτιση των αριθμών των i-nodes είναι σύμπτωση. Σχετικός σύνδεσμος: `http://www.delorie.com/djgpp/bugs/show.cgi?000342`