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

Κανονική Εξέταση – Λύσεις

Το παρόν περιγράφει πλήρη λύση τον 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, όταν μια διεργασία δημιουργείται 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 εκτελεί συγκεκριμένη συνάρτηση ή κλήση συστήματος.

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

i. *Τι κάνει η κλήση συστήματος `pipe()` στο UNIX; Πόσες και τι τύπου τιμές δίνει στο πρόγραμμα που την καλεί; Με ποιον τρόπο επιστρέφει αυτές τις τιμές και πού τις γράφει;*

ii. *Τι παθαίνει μια διεργασία που καλεί την `read()` σε άδειο pipe όταν είναι ανοιχτό το άκρο εγγραφής;*

iii. *Τι επιστρέφει η `read()` σε κάποιον που διαβάζει από άδειο pipe όταν κανείς δεν έχει πλέον ανοιχτό το άκρο εγγραφής;*

iv. *Τι επιστρέφουν οι κλήσεις `getpid()`, `getppid()` στο UNIX; Υπάρχει περίπτωση οι τιμές που επιστρέφουν να αλλάξουν για την ίδια διεργασία κατά τη διάρκεια της ζωής της;*

i. Κατασκευάζει ένα νέο pipe στο χώρο πυρήνα. Επιστρέφει δύο περιγραφητές αρχείων, που είναι μη μηδενικοί ακέραιοι: έναν περιγραφητή για το άκρο ανάγνωσης, έναν για το άκρο εγγραφής. Τους γράφει σε δύο συνεχόμενες θέσεις μνήμης, ξεκινώντας από τον δείκτη `p` που δέχεται ως όρισμα, ώστε να είναι προσβάσιμοι ως `p[0]`, `p[1]`, αντίστοιχα.

ii. Μπλοκάρει μέχρι να γράψει κάποιος, ή να καταρρεύσει το άκρο εγγραφής.

iii. Επιστρέφει τιμή `0`, δηλώνοντας κατάσταση EOF.

iv. Επιστρέφουν το PID της ίδιας της διεργασίας και της γονικής της, αντίστοιχα. Η επιστροφή της `getpid()` δεν αλλάζει ποτέ, η επιστροφή της `getppid()` μπορεί να αλλάξει αν ο γονέας μιας διεργασίας πεθάνει κι αυτή υιοθετηθεί από την `init`.

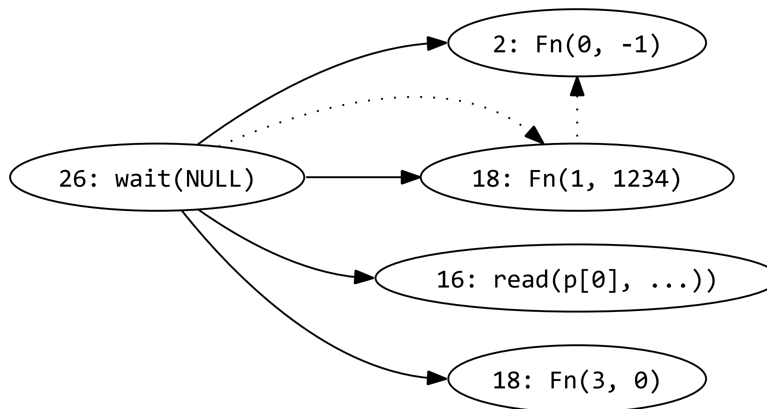
β. (12%) *Για την τελική κατάσταση των διεργασιών του προγράμματος: σχεδιάστε το δέντρο διεργασιών τότε. Εξηγήστε συνοπτικά πώς προκύπτει.*

γ. (4%) *Για κάθε κόμβο του δέντρου, 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  void handler(int signum)
2  { Fn(0, -1); }
3
4  int main(void)
5  {
6      int i, p[2], writefd[4];
7      pid_t pid[4];
8
9      signal(SIGUSR1, handler);
10
11     for (i = 0; i < 4; i++) {
12         pipe(p);
13         pid[i] = fork();
14         if (pid[i] == 0) {
15             close(p[1]);
16             read(p[0], &pid[i], sizeof(pid_t));
17             if (pid[i] > 0) kill(pid[i], SIGUSR1);
18             Fn(i, pid[i]);
19         }
20         close(p[0]);
21         writefd[i] = p[1];
22     }
23
24     write(writefd[1], &pid[0], sizeof(pid_t));
25     close(writefd[3]);
26     wait(NULL);
27     write(writefd[2], &pid[1], sizeof(pid_t));
28     Fn(-1, -2);
29     return 0;
30 }
```

Το δέντρο διεργασιών στην τελική κατάσταση φαίνεται στο ακόλουθο σχήμα.



Ο πατέρας κατασκευάζει 4 παιδιά, ένα pipe για το κάθε παιδί. Κρατάει ανοιχτά μόνο τα άκρα εγγραφής, στον πίνακα `writefd[]`. Κάθε παιδί μπλοκάρει, μέχρι να διαβάσει ένα PID από το αντίστοιχο pipe, στο οποίο στέλνει `SIGUSR1`. Ο πατέρας ζητά από το παιδί `1` να σκοτώσει το `0`, το `2` μένει για πάντα μπλοκαρισμένο, και το `3` ξεμπλοκάρει γιατί ο πατέρας κλείνει το άκρο εγγραφής και δεν μπορεί κανείς πια να γράψει στο pipe. Διαδιεργασιακή επικοινωνία: Ο πατέρας στέλνει το pid του `0`, έστω `1234` στο παιδί `1`, το `1` σκοτώνει το `0`.
Επεξήγηση της απάντησης: Αρχικά (γραμμή 9) κανονίζεται κάθε φορά που παραλαμβάνεται σήμα `SIGURG` να εκτελείται η συνάρτηση χειρισμού `handler`.
Η αρχική διεργασία εκτελεί 4 `fork()`, δημιουργώντας ισάριθμες διεργασίες-παιδιά (γραμμή 13, μέσα στο for loop, γραμμή 11). Πριν από κάθε δημιουργία παιδιού κατασκευάζει ένα νέο pipe με άκρα ανάγνωσης/εγγραφής τα `p[0]`, `p[1]`, αντίστοιχα. Μετά από κάθε `fork()` ο πατέρας κλείνει το άκρο ανάγνωσης και κρατά το άκρο εγγραφής στην αντίστοιχη θέση πίνακα `writefd[]`. Επίσης, κρατά το PID του νέου παιδιού σε αντίστοιχη θέση του πίνακα `pid[]`. Τελικά, μετά το loop (γραμμή 23) οι πίνακες `pid[]`, `writefd[]` περιέχουν τα PIDs των παιδιών και τους περιγραφητές για τα άκρα εγγραφής των pipes, αντίστοιχα.
Κάθε νέο παιδί, αφού κλείσει το άκρο εγγραφής του αντίστοιχου pipe, μπλοκάρει με `read()` στο άκρο ανάγνωσης ζητώντας να διαβάζει στην αντίστοιχη θέση του πίνακα `pid[]` τόσα bytes όσο ο τύπος `pid_t`. Αυτή η ενέργεια δεν επηρεάζει καθόλου τον πατέρα, αφού δεν μοιράζονται μνήμη. Επιπλέον, η αρχική τιμή της θέσης `pid[i]` είναι μηδέν, αφού η `fork()` στο παιδί επιστρέφει `0`.
Ο πατέρας περνά στο παιδί `1` το PID του παιδιού `0` (έστω `1234`) στη γραμμή 24. Στη συνέχεια κλείνει τον περιγραφητή `writefd[3]`, οπότε το άκρο εγγραφής του pipe `3` καταρρέει. Τέλος, μπλοκάρει για πάντα στη `wait()` (γραμμή 26), γιατί κανένα παιδί δεν πεθαίνει μέχρι να φτάσει το σύστημα σε μόνιμη κατάσταση.
Το παιδί `1` ξυπνά, στέλνει `SIGUSR1` στο παιδί `0` (γραμμή 17) και μπλοκάρει για πάντα σε κλήση της `Fn(1, 1234)`. Το παιδί `0` ήταν μπλοκαρισμένο στη γραμμή 16. Με την παραλαβή του `SIGUSR1` εκτελεί τον `handler()` και μπλοκάρει για πάντα στην `Fn()`, γραμμή 2. Το παιδί `3` ξυπνά, γιατί η `read()` στη γραμμή 1 επιστρέφει EOF, αφού κανείς δεν μπορεί πια να γράψει στο άδειο pipe `3`. Η τιμή `pid[3]` έχει παραμείνει `0`, οπότε μπλοκάρει σε κλήση `Fn(3, 0)`, γραμμή 18. Τέλος, το παιδί `2` δεν έχει κανέναν να το ξυπνήσει ποτέ, και μένει για πάντα στη `read()`.
Η μόνη διαδιεργασιακή επικοινωνία που ολοκληρώνεται είναι η μεταφορά του PID του παιδιού `0`, έστω `1234` από την αρχική διεργασία στο παιδί `1`, και η αποστολή του σήματος `SIGUSR1` από το παιδί `1` στο παιδί `0`.

Θέμα 2 (25%)

α. (10%) *Σε ένα γραφείο μπορούν να βρίσκονται είτε οι εργαζόμενοι που δουλεύουν εκεί, είτε η καθαρίστρια που επιμελείται το χώρο. Οι εργαζόμενοι εργάζονται μέσα στο χώρο (`work_for_a_while()`) και κάνουν περιοδικά διάλειμμα στην αυλή (`take_a_break()`). Η καθαρίστρια δεν μπαίνει ποτέ στο*

γραφείο όσο υπάρχουν υπάλληλοι εκεί· αντίστοιχα, οι εργαζόμενοι δεν μπαίνουν στο γραφείο όσο η καθαρίστρια καθαρίζει.
Οι εργαζόμενοι αναπαρίστανται από διεργασίες που εκτελούν τη διαδικασία `worker()`, η καθαρίστρια αναπαρίσταται από διεργασία που εκτελεί τη διαδικασία `cleaner()`.

```
void worker()
{
    for (;;) {
        . . .
        work_for_a_while();
        . . .
        take_a_break();
    }
}
```

```
void cleaner()
{
    for (;;) {
        . . .
        clean_the_room();
        . . .
        take_a_break();
    }
}
```

Στα σημεία που υποδηλώνονται με ". . ." ζητείται να υλοποιήσετε σχήμα συγχρονισμού ώστε να εξασφαλίζεται ο τρόπος εργασίας που περιγράφηκε. Μπορείτε να χρησιμοποιήσετε κλήσεις `signal()` και `wait()` σε κατάλληλα αρχικοποιημένους σημαφόρους κι όποιες μεταβλητές μοιραζόμενες ανάμεσα στις διεργασίες χρειάζεστε.
Στη λύση που δώσατε, υπάρχει ενδεχόμενο λιμοκτονίας; Σχολιάστε.

Η κατάσταση που περιγράφεται είναι ισοδύναμη με το γνωστό πρόβλημα συγχρονισμού αναγνωστών και εγγραφέων.

```
mutex = semaphore(1);
empty = semaphore(1);
int worker_count = 0;
```

```
void worker()
{

    for (;;) {
        wait(mutex);
        worker_count++;
        if (worker_count == 1)
            wait(empty);
        signal(mutex);

        work_for_a_while();

        wait(mutex);
        worker_count--;
        if (worker_count == 0)
            signal(empty);
        signal(mutex);

        take_a_break();
    }
}
```

```
void cleaner()
{

    for (;;) {
        wait(empty);
        clean_the_room();
        signal(empty);
        take_a_break();
    }
}
```

Η λύση αυτή ενέχει τον κίνδυνο να λιμοκτονήσει η καθαρίστρια περιμένοντας να αδειάσει ο χώρος εργασίας, αν συνεχώς καταφτάνουν νέοι εργαζόμενοι.

β. (5%) *Το γραφείο του προηγούμενου ερωτήματος διαθέτει ακριβώς N θέσεις εργασίας. Τροποποιήστε το σχήμα συγχρονισμού του προηγούμενου ερωτήματος, ώστε το πολύ N εργαζόμενοι να μπορούν να βρίσκονται μέσα στο χώρο εργασίας, να εκτελούν δηλαδή την `work_for_a_while()` ταυτόχρονα.*

Θα χρησιμοποιήσουμε έναν επιπλέον σημαφόρο (`available_desks`), αρχικοποιημένο στην τιμή N, ο οποίος θα καταγράφει των αριθμό των ελεύθερων θέσεων εργασίας στο χώρο:

```
mutex = semaphore(1);
empty_room = semaphore(1);
available_desks = semaphore(N);
int worker_count = 0;
```

```
void worker()
{

    for (;;) {
        wait(mutex);
        worker_count++;
        if (worker_count == 1)
            wait(empty_room);
        signal(mutex);

        wait(available_desks);
        work_for_a_while();
        signal(available_desks);

        wait(mutex);
        worker_count--;
        if (worker_count == 0)
            signal(empty_room);
        signal(mutex);

         take_a_break();
    }
}
```

```
void cleaner()
{

    for (;;) {
        wait(empty_room);
        clean_the_room();
        signal(empty_room);
        take_a_break();
    }
}
```

γ. (10%) *Επειδή οι ανάγκες καθαρισμού αυξήθηκαν, προσλαμβάνονται περισσότερες καθαρίστριες, οι οποίες μπορούν να εργάζονται ταυτόχρονα στον ίδιο χώρο. Και πάλι, μόνο όταν δεν υπάρχουν εργαζόμενοι εκεί.*
Τροποποιήστε κατάλληλα το σχήμα συγχρονισμού του προηγούμενου ερωτήματος, ώστε στο χώρο να μπορούν να βρίσκονται ταυτόχρονα είτε το πολύ N εργαζόμενοι είτε το πολύ M καθαρίστριες.

Χρησιμοποιούμε έναν επιπλέον σημαφόρο (`available_brooms`) για να καταγράφουμε τον αριθμό των διαθέσιμων σκουπών για χρήση από τις καθαρίστριες. Ο κώδικας της διαδικασίας `cleaner()` είναι πλέον συμμετρικός με τον κώδικα της διαδικασίας `worker()`:

```
mutex1 = semaphore(1);
mutex2 = semaphore(1);
empty_room = semaphore(1);
available_desks = semaphore(N);
available_brooms = semaphore(M);
int worker_count = 0;
int cleaner_count = 0;
```

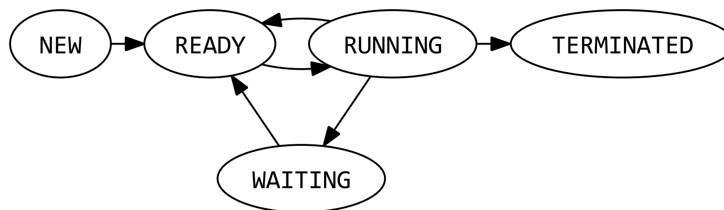
```
void worker()
{
    for (;;) {
        wait(mutex1);
        worker_count++;
        if (worker_count == 1)
            wait(empty_room);
        signal(mutex1);

        wait(available_desks);
        work_for_a_while();
        signal(available_desks);

        wait(mutex1);
        worker_count--;
        if (worker_count == 0)
            signal(empty_room);
        signal(mutex1);

        take_a_break();
    }
}
```

```
void cleaner()
{
    for (;;) {
        wait(mutex2);
        cleaner_count++;
        if (cleaner_count == 1)
            wait(empty_room);
        signal(mutex2);

        wait(available_brooms);
        clean_the_room();
        signal(available_brooms);

        wait(mutex2);
        cleaner_count--;
        if (cleaner_count == 0)
            signal(empty_room);
        signal(mutex2);

        take_a_break();
    }
}
```

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%) *Σε ποιες μεταβάσεις λαμβάνεται απόφαση χρονοδρομολόγησης από το ΛΣ; Πού διαφέρει ένας διακοπτός (preemptive) από ένα μη-διακοπτό/συνεργατικό (non-preemptive/cooperative) αλγόριθμο χρονοδρομολόγησης σε σχέση με το πότε παίρνονται αποφάσεις χρονοδρομολόγησης; Σχολιάστε τη συμπεριφορά τους όταν συμβαίνουν διακοπές λογισμικού και υλικού (software και hardware interrupts).*

Μη-διακοπτός: μόνο σε RUNNING → WAITING ή RUNNING → TERMINATED. Διακοπτός: και σε μεταβάσεις RUNNING → READY, WAITING → READY. Ο μη-διακοπτός αποφασίζει μόνο όταν συμβεί software interrupt (system call ή άλλο trap), ο διακοπτός αποφασίζει κι όταν συμβεί hardware interrupt (π.χ. από τον timer ή τον δίσκο).
Επεξήγηση της απάντησης: Γενικά, μπορεί να λαμβάνεται απόφαση σε οποιαδήποτε από τις μεταβάσεις RUNNING → WAITING, RUNNING → READY, WAITING → READY, RUNNING → TERMINATED.
Ένας μη-διακοπτός (non-preemptive/cooperative) χρονοδρομολογητής δίνει τη CPU σε μια διεργασία μέχρι *εκείνη* να θελήσει να την απελευθερώσει, είτε να εκτελέσει δηλαδή μια κλήση συστήματος που μπλοκάρει, είτε να τερματίσει. Αυτό σημαίνει ότι λαμβάνει απόφαση χρονοδρομολόγησης μόνο στις μεταβάσεις RUNNING → WAITING, RUNNING → TERMINATED. Οι μεταβάσεις αυτές αντιστοιχούν σε διακοπές *λογισμικού*: η τρέχουσα διεργασία προκαλεί trap για κλήση συστήματος, ή page fault.
Ένας διακοπτός (preemptive) χρονοδρομολογητής ενεργοποιείται και στις μεταβάσεις RUNNING → READY, WAITING → READY. Οι μεταβάσεις αυτές αντιστοιχούν σε διακοπές *υλικού*, από εξωτερικούς παράγοντες: ο χρονιστής διακόπτει τη CPU γιατί το κβάντο χρόνου τελείωσε, ή ο δίσκος διακόπτει τη CPU γιατί μια μεταφορά ολοκληρώθηκε, αντίστοιχα.
Στην περίπτωση αυτή, η διακοπή υλικού μπορεί να οδηγήσει σε αλλαγή της τρέχουσας διεργασίας, ακόμη κι αν εκείνη δεν ήταν διατεθειμένη να αφήσει εκείνη τη στιγμή τη CPU.

δ. (5%) *Αναφέρετε ένα πλεονέκτημα κι ένα μειονέκτημα για κάθε μία από τις δύο στρατηγικές (διακοπτή χρονοδρομολόγηση ή μη-διακοπτή). Ποια από τις δύο θα διαλέγατε για ένα ΛΣ με γραφικό περιβάλλον που εκτελεί εφαρμογές που έχουν συχνή αλληλεπίδραση με το χρήστη;*

Μη-διακοπτή χρονοδρομολόγηση: απλούστερη στην υλοποίηση, χωρίς ειδικό υλικό (χρονιστής), πρόβλημα αν μία διεργασία μονοπωλεί τη CPU. Διακοπτή: εξασφαλισμένη κατανομή χρόνου, πιο πολύπλοκη στην υλοποίηση, ανάγκη συγχρονισμού για πρόσβαση σε κοινά δεδομένα. Θα επιλέγαμε διακοπτή χρονοδρομολόγηση για καλύτερη αποκρισιμότητα του συστήματος.
Επεξήγηση της απάντησης: Η μη-διακοπτή/συνεργατική χρονοδρομολόγηση είναι απλούστερη στην υλοποίηση και δεν απαιτεί ειδικό υλικό (χρονιστή), ωστόσο μια κακογραμμένη ή κακόβουλη διεργασία μπορεί να μονοπωλεί τη CPU και να καταστρέψει την αποκρισιμότητα του συστήματος, μην επιτρέποντας σε άλλες να εκτελεστούν. Η διακοπτή χρονοδρομολόγηση επιβάλλει το μηχανισμό χρονοδρομολόγησης σε κάθε περίπτωση, αλλά είναι περισσότερο πολύπλοκη στην υλοποίηση και εισάγει κόστος λόγω της ανάγκης συγχρονισμού της πρόσβασης σε κοινά δεδομένα.
Στην περίπτωση που περιγράφεται ταιριάζει διακοπτή στρατηγική χρονοδρομολόγησης, ώστε να εξασφαλιστεί ότι το σύστημα θα έχει προβλέψιμο χρόνο απόκρισης στις ενέργειες του χρήστη κι αυτός θα απολαμβάνει καλή υπηρεσία.

ε. (5%) *Δουλεύετε στον editor `vi` σε μονοεπεξεργαστικό σύστημα UNIX, ενώ μεταγλωττίζετε μεγάλο πρόγραμμα. Πιέζετε ένα πλήκτρο. Η ενέργειά σας αυτή μπορεί να προκαλέσει context switch*

όταν το ΛΣ χρησιμοποιεί (i) διακοπτό (ii) μη-διακοπτό αλγόριθμο χρονοδρομολόγησης; Αν ναι, περιγράψτε με βήματα τι συμβαίνει στο σύστημα και καταλήγει σε context switch.

Πατάμε το πλήκτρο, γίνεται διακοπή υλικού, ξεκινά ο interrupt handler, ο `vi` γίνεται READY. Σε διακοπτή χρονοδρομολόγηση μπορεί να σταματήσει η τρέχουσα διεργασία (έστω ο μεταγλωττιστής) και να γίνει context switch στο `vi`. Σε μη-διακοπτή θα γίνει context switch αργότερα, μόνο όταν ο μεταγλωττιστής μπλοκάρει ή τερματίσει.

Επεξήγηση της απάντησης: Υποθέτουμε ότι ο μεταγλωττιστής είναι η τρέχουσα διεργασία όταν πιέζουμε ένα πλήκτρο. Το πληκτρολόγιο προκαλεί διακοπή υλικού, η οποία εξυπηρετείται από κώδικα πυρήνα του ΛΣ και προκαλεί μετάβαση WAITING $\rightarrow$ READY για τη διεργασία του `vi`. Στην περίπτωση μη-διακοπτής χρονοδρομολόγησης δεν λαμβάνεται εδώ απόφαση χρονοδρομολόγησης, κι η τρέχουσα διεργασία συνεχίζει να εκτελείται ανεπηρέαστη. Όταν εκτελέσει μια κλήση συστήματος που μπλοκάρει, π.χ. χρειαστεί δεδομένα από το δίσκο, οπότε θα έχουμε μετάβαση RUNNING $\rightarrow$ WAITING, τότε θα επιλεγεί μια νέα διεργασία από όσες είναι READY προς εκτέλεση. Υποθέτουμε ότι θα είναι ο `vi`, οπότε θα γίνει context switch προς αυτόν. Αν έχουμε διακοπτή χρονοδρομολόγηση, με το που ο `vi` περάσει σε READY θα ληφθεί απόφαση χρονοδρομολόγησης, οπότε μπορεί να γίνει άμεσα context switch από τη διεργασία του μεταγλωττιστή προς αυτόν.

Θέμα 4 (25%)

α. (15%) *Μια διεργασία εκτελεί το ακόλουθο τμήμα κώδικα σε σύστημα UNIX εικονικής μνήμης με σελιδοποίηση. Θεωρήστε ότι οι σελίδες που περιέχουν τον κώδικά της και τις μεταβλητές `i`, `n`, `sum`, `len`, `total`, είναι πάντοτε στη φυσική μνήμη. Καμία κλήση δεν αποτυγχάνει. Υπάρχουν κι άλλες διεργασίες στο σύστημα κι ο αλγόριθμος χρονοδρομολόγησης είναι RR. Το σύστημα έχει 16GB φυσικής μνήμης.*

```
1   long i, n, sum, len, total = 1024 * 1048576;
2   char *p, *buf = malloc(total);
3
4   fd = open("a_file_larger_than_1GB", O_RDONLY);
5   p = buf; len = total;
6   while (len > 0) { /* 1GB total */
7       n = read(fd, p, 1048576); /* 1MB */
8       p += n; len -= n;
9   }
10
11  for (sum = 0, i = 0; i < total; i++) sum += i;
12  sleep(60);
13  for (i = 0; i < total; i++) buf[i]++;
```

Απαντήστε συνοπτικά, όχι πάνω από δύο-τρεις γραμμές, στις παρακάτω ερωτήσεις για τη συμπεριφορά της διεργασίας. Κάντε όποιες ρεαλιστικές υποθέσεις χρειάζεστε, αρκεί να τις καταγράψετε. Σε κάθε περίπτωση δικαιολογήστε θετική απάντησή σας με αντίστοιχο σενάριο/παράδειγμα.

i. *Ποιες κλήσεις συστήματος εκτελεί η διεργασία στις γραμμές 4–9; Γίνεται να υποστεί μετάβαση RUNNING $\rightarrow$ WAITING κατά την εκτέλεσή τους;*

ii. *Γίνεται να μην πάει σε WAITING καθ' όλη την εκτέλεση των 4–9;*

iii. *Γίνεται να μην παραμείνει RUNNING καθ' όλη την εκτέλεση της 11;*

iv. *Γίνεται να παραμείνει RUNNING κατά την εκτέλεση της 12;*

v. *Ποιες κλήσεις συστήματος εκτελεί στη γραμμή 13; Ποιες είναι όλες οι δυνατές μεταβάσεις κατά την εκτέλεση της 13; Είναι δυνατή μετάβαση RUNNING → WAITING; Γιατί; Θεωρήστε καταστάσεις RUNNING, WAITING, READY, TERMINATED.*

i. Εκτελεί κλήσεις `open()` και `read()`. Ναι, αν πρέπει το ΛΣ να ζητήσει να έρθουν δεδομένα από το δίσκο. Μέχρι να έρθουν, θα είναι WAITING.

ii. Ναι, αν ό,τι χρειάζεται από το δίσκο είναι στην block/page cache που τηρεί το ΛΣ στη RAM.

iii. Ναι, γιατί έχουμε διακοπτή χρονοδρομολόγηση (Round-Robin) και μπορεί να πάει σε READY, όταν εκπνεύσει το κβάντο χρόνου.

iv. Αποκλείεται, η `sleep()` μπλοκάρει.

v. Καμία. Μπορεί να πάει RUNNING → READY, READY → RUNNING, RUNNING → WAITING, WAITING → READY, αν εκπνεύσει το κβάντο χρόνου ή κάνει page fault κατά την πρόσβαση στον `buf[]`.

Επεξήγηση των απάντησεων:

i. Εκτελεί τις κλήσεις συστήματος `open()` και `read()`. Ναι, είναι πολύ πιθανό να υποστεί μετάβαση RUNNING → WAITING, γιατί π.χ, το ΛΣ χρειάζεται να φέρει από το δίσκο καταλόγους του συστήματος αρχείων για να ολοκληρώσει το `open()`, ή να φέρει μπλοκ με τα περιεχόμενα του αρχείου, για να ολοκληρώσει αιτήσεις `read()`. Στην περίπτωση αυτή, η διεργασία είναι WAITING για όσο διαρκεί η λειτουργία Ε/Ε.

ii. Ναι, μπορεί να μην πάει ποτέ σε WAITING. Το ΛΣ αποθηκεύει προσωρινά σε κρυφή μνήμη στη RAM (block/page cache) δεδομένα του δίσκου, ώστε να αποφεύγει συχνές λειτουργίες Ε/Ε. Επειδή το σύστημα έχει πολλή φυσική μνήμη (16GB), είναι πιθανό, ειδικά αν δεν είναι η πρώτη φορά που τρέχει αυτό το πρόγραμμα, το 1GB δεδομένων που χρειάζεται να είναι ήδη στην cache.

iii. Παρόλο που δεν εκτελεί καμία κλήση συστήματος, ούτε υπάρχει περίπτωση να υποστεί page fault, αφού θεωρούμε ότι όλες οι σελίδες που ακουμπά στη γραμμή 11 είναι στη φυσική μνήμη, δεν είναι σίγουρο ότι θα παραμείνει RUNNING καθ'όλη την εκτέλεσή της. Αφού έχουμε χρονοδρομολόγηση RR μαζί με άλλες διεργασίες, αν η CPU έχει ρολόι της τάξης των GHz είναι σίγουρο ότι η εκτέλεση της 11 θα πάρει χρόνο της τάξης του sec, κάποια κβάντα χρόνου. Οπότε η διεργασία θα υποστεί μεταβάσεις RUNNING → READY → RUNNING.

iv. Αποκλείται. Κλήση της `sleep()` συνεπάγεται μετάβαση σε WAITING.

v. Δεν εκτελεί καμία κλήση συστήματος. Ωστόσο, επειδή (i) η εκτέλεση της 13 διαρκεί κάποια κβάντα χρόνου (ii) δεν είναι εξασφαλισμένο ότι όλες οι σελίδες για το 1GB στο οποίο δείχνει ο `buf` θα είναι στη φυσική μνήμη, μπορεί να συμβούν οι μεταβάσεις RUNNING → READY, READY → RUNNING, RUNNING → WAITING, WAITING → READY. Οι δύο πρώτες λόγω εκπνοής κβάντου χρόνου, οι δύο τελευταίες λόγω page fault και εκτέλεσης λειτουργίας Ε/Ε για την μεταφορά των απαιτούμενων σελίδων στην κύρια μνήμη.

β. (10%) *Σε ΛΣ με σελιδοποίηση κατ'απαίτηση (demand paging), επιχειρούμε να τρέξουμε με λάθος παραμέτρους ένα πολύ μεγάλο εκτελέσιμο, της τάξης των 400MB. Όταν το εκτελέσιμο τρέξει, εκτυπώνει μήνυμα λάθους για τις παραμέτρους κι η διεργασία τερματίζει. Τα μπλοκ του δίσκου που το περιέχουν δεν έχουν προσπελαστεί ποτέ από την εκκίνηση του ΛΣ. Το ΛΣ δεν γνωρίζει τίποτε για την εσωτερική οργάνωση του κώδικα του εκτελέσιμου. Ο δίσκος υποστηρίζει ρυθμό μεταφοράς 20MB/s προς την κύρια μνήμη.*

i. *Πόσο χρόνο τουλάχιστον θα χρειαζόταν η ανάγνωση του εκτελέσιμου από το δίσκο στην κύρια μνήμη;*

ii. *Περιγράψτε μηχανισμό του ΛΣ με τον οποίο μια διεργασία μπορεί να προσπελάσει δεδομένα αρχείου στο δίσκο, χωρίς την εκτέλεση κλήσεων συστήματος `read()`/`write()`.*

iii. *Περιγράψτε σενάριο όπου το εκτελέσιμο τρέχει, παράγει το μήνυμα λάθους και τερματίζει, σε πολύ λιγότερο από μερικά δευτερόλεπτα.*

i. Τουλάχιστον $t = \frac{400MB}{20MB/s} = 20s$.

ii. Με *απεικόνιση* του αρχείου στην εικονική μνήμη, η διεργασία κάνει Ε/Ε με εντολές load/store του επεξεργαστή, χωρίς system call. Ζητάει απεικόνιση (με `mmap()`) και κάνει load ή store κάτι στην ανάλογη περιοχή της μνήμης. Τότε θα συμβεί page fault και το ΛΣ θα φέρει τη σελίδα που χρειάζεται στη μνήμη.

iii. Όπως στο (ii), ο φορτωτής απεικονίζει το εκτελέσιμο στη μνήμη, χωρίς καμία σελίδα του στη μνήμη. Με το που πάει ο PC στην πρώτη εντολή, θα γίνει page fault. Ενώ το πρόγραμμα τρέχει, θα έρθουν στη μνήμη μόνο οι λίγες, απαραίτητες σελίδες που χρειάζονται για να τυπωθεί το μήνυμα λάθους, οπότε μόνο ένα πολύ μικρό μέρος του προγράμματος θα έρθει από τον δίσκο.

Επεξήγηση της απάντησης:

i. Ο δίσκος μεταφέρει με 20MB/s, το αρχείο είναι 400MB, χρειάζεται τουλάχιστον $t = \frac{400MB}{20MB/s} = 20s$.

ii. Μια διεργασία μπορεί να χρησιμοποιήσει το μηχανισμό *απεικόνισης* (mapping) αρχείων στην εικονική μνήμη, ώστε να έχει πρόσβαση στα δεδομένα τους μέσω εντολών load/store χωρίς την πραγματοποίηση κλήσεων συστήματος. Αρχικά χρειάζεται να εγκαταστήσει την απεικόνιση, ζητώντας από το ΛΣ με μια κλήση `mmap()` να τροποποιήσει κατάλληλα το χάρτη μνήμης της. Το αρχείο απεικονίζεται σε συγκεκριμένη περιοχή εικονικής μνήμης, ξεκινώντας π.χ. από τη διεύθυνση `vaddr`. Ένα load στη διεύθυνση `vaddr + offset` προκαλεί page fault: ενεργοποιείται η ρουτίνα εξυπηρέτησης σφάλματος σελίδας του ΛΣ, η οποία αναθέτει ένα νέο πλαίσιο μνήμης στη διεργασία, το οποίο γεμίζει με τη σελίδα του αρχείου η οποία περιέχει το `offset`. Εφεξής, η διεργασία μπορεί να έχει πρόσβαση στο `offset` και τα δεδομένα που περιέχονται στην ίδια σελίδα χωρίς page fault. Αν θελήσει να κάνει load ή store σε σελίδα που αντιστοιχεί σε άλλο μέρος του αρχείου, γίνεται πάλι page fault κι η διαδικασία επαναλαμβάνεται.

iii. Δεν χρειάζεται να φορτωθεί ολόκληρο το εκτελέσιμο στη μνήμη για να ξεκινήσει η εκτέλεση της διεργασίας. Η εκτέλεση μπορεί να ξεκινήσει έχοντας απεικονισμένο το εκτελέσιμο στη μνήμη, χωρίς ούτε μία σελίδα του να βρίσκεται στη φυσική μνήμη. Με το που τεθεί ο PC στο σημείο εκκίνησης του εκτελέσιμου, θα προκληθεί page fault και θα έρθει η πρώτη σελίδα του στη μνήμη. Αυτό θα γίνεται συνεχώς καθώς ο PC ακουμπά σελίδες που δεν έχουν ακόμη έγκυρη μετάφραση στον πίνακα σελίδων. Υποθέτοντας ότι το πρόγραμμα πολύ σύντομα θα

τερματίσει, γιατί έχει κληθεί χωρίς τα σωστά ορίσματα, κι ότι για να γίνει αυτό ενεργοποιείται ένα πολύ μικρό μέρος του, καταλήγουμε ότι πολύ λίγες σελίδες θα χρειαστεί να έρθουν τελικά στη φυσική μνήμη. Ο συνολικός χρόνος μεταφοράς τους καθορίζει και το χρόνο εκτέλεσης του προγράμματος.