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

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 `wp = wait(&status);` Έστω ότι μετά την κλήση της από τη διεργασία με PID `1000` ισχύει `wp == 1011`, `WIFSIGNALED(status) == 1`, `WTERMSIG(status) == 3`. Έστω ότι η `1000` καθόρισε την τύχη της `1011`. Τι συνέβη στη διεργασία `1011` και με ποια κλήση (έστω σε γλώσσα C) την επηρέασε η `1000`;*
4. *Τι συμβαίνει σε μια διεργασία που καλεί την `read()` σε άδειο pipe όταν είναι ανοιχτό το άκρο εγγραφής;*

1. 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`.
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 τιμή του `status` δείχνει ότι η διεργασία τερματίστηκε λόγω παραλαβής του σήματος `3`. Η `1000` σκότωσε το παιδί της, `1011` τρέχοντας `kill(1011, 3)`, ή ισοδύναμα `kill(1011, SIGQUIT)` στο Linux, όπου το `SIGQUIT` είναι το σήμα `3`.
4. Μπλοκάρει μέχρι κάποια διεργασία να γράψει στο pipe, ή να κλείσουν όλοι οι περιγραφητές που αναφέρονται στο άκρο εγγραφής.

β. (12%) *Σχεδιάστε το δένδρο διεργασιών στην τελική του μορφή, όταν δηλαδή όλες οι διεργασίες έχουν φτάσει σε μόνιμη κατάσταση. Εξηγήστε συνοπτικά πώς προκύπτει. Λύστε πιθανή κατάσταση συναγωνισμού (race) στη γραμμή 18, θεωρώντας ότι τουλάχιστον τρεις διεργασίες φτάνουν στο σημείο που σημειώνεται ως `/* A */`, περνώντας το `read()` με τη σειρά γέννησής τους.*

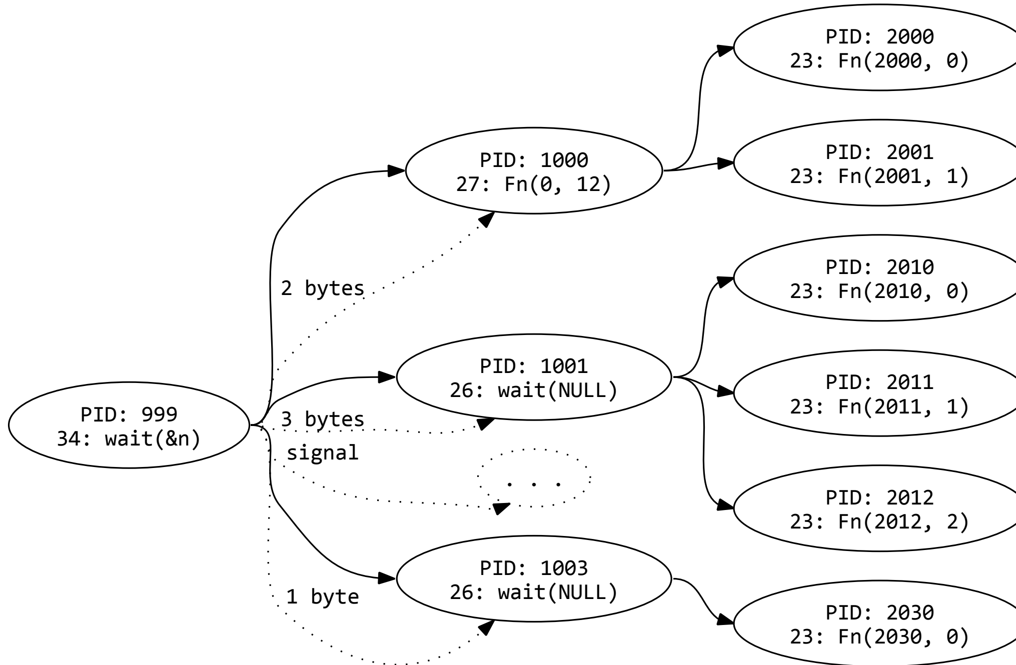
γ. (4%) *Σε κάθε κόμβο του δέντρου διεργασιών επισημάνετε: (i) την κλήση συστήματος ή συνάρτηση μέσα στην οποία βρίσκεται ο Program Counter της αντίστοιχης διεργασίας, (ii) τα ορίσματα με τα οποία αυτή έχει κληθεί, (iii) τη γραμμή του προγράμματος απ' όπου έγινε η κλήση της.*
Για τα ορίσματα, κάντε οποιαδήποτε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

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(6); Fn(0, -1); }
5
6   int main(void)
7   {
8       int i, j, n;
9       char c[10];
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                  n = read(pg[0], c, i + 2);
19                  /* A */
20                  for (j = 0; j < n; j++) {
21                      pid[i] = fork();
22                      if (pid[i] == 0)
23                          Fn(getpid(), j);
24                  }
25                  for (j = 0; j < i; j++)
26                      wait(NULL);
27                  Fn(i, 12);
28                  exit(2);
29          }
30      }
31
32      kill(pid[i - 2], SIGUSR1);
33      for (i = 0; i < 4; i++) {
34          wait(&n);
35          write(pg[1], c, WEXITSTATUS(n));
36      }
37      Fn(i, -1);
```

```
38        return 0;
39    }
```

Το δέντρο διεργασιών στην τελική κατάσταση φαίνεται στο ακόλουθο σχήμα.



Η αρχική διεργασία, έστω η `999` δημιουργεί 4 διεργασίες-παιδιά, έστω τις `1000-1003`, που μπλοκάρουν περιμένοντας να διαβάσουν από το άδειο pipe `pg` (γραμμή 18). Η γονική διεργασία αποστέλλει `SIGUSR1` στη `1002` (γραμμή 32), η οποία έχει κληρονομήσει τη συνάρτηση χειρισμού του σήματος `SIGUSR1`, όπως είχε τεθεί από την αρχική διεργασία (γραμμή 12), οπότε πεθαίνει με τιμή επιστροφής `6` (γραμμή 4). Ο πατέρας γράφει τόσα (6) bytes στο pipe. Η `i`-οστή διεργασία διαβάζει `i+2` bytes, δηλαδή οι `1000`, `1001`, `1003` ζητούν να διαβάσουν 2, 3, 5 bytes αντίστοιχα. Στο σημείο αυτό υπάρχει race: Δεν ξέρουμε με ποια σειρά οι διεργασίες θα διαβάσουν από το pipe. Ξέρουμε όμως ότι αυτό συμβαίνει ατομικά (λόγω του pipe) και ότι και οι τρεις ξεμπλοκάρουν και φτάνουν στο σημείο `/* A */` (δίνεται). Άρα, *σίγουρα* πρώτα διαβάζουν από 2 και 3 bytes οι `1000`, `1001` αντίστοιχα (αδιάφορη η σειρά), και έπειτα η read στην `1003` επιστρέφει έχοντας διαβάσει το μοναδικό byte που είχε απομείνει, οπότε για την `1003` ισχύει `n == 1` στη γραμμή 19. Τελικά κάθε διεργασία-παιδί φτιάχνει τόσα παιδιά όσα bytes διάβασε από το pipe και μπλοκάρει για πάντα στη γραμμή 26, αφού κανένα από τα παιδιά της δεν τερματίζει. Εξαιρείται η `1000`, η οποία δεν μπαίνει στο loop (γραμμή 25), αλλά μπλοκάρει στη μπαίνει στην `Fn()`, γραμμή 27.
Στο παραπάνω σχήμα, ο κόμβος του παιδιού `1002` υπάρχει μόνο για να φανεί η αποστολή του σήματος, η συγκεκριμένη διεργασία δεν είναι μέρος του δέντρου σε μόνιμη κατάσταση. Οι τιμές των bytes που μεταφέρονται μέσω του pipe είναι τυχαίες και δεν επηρεάζουν την εκτέλεση του προγράμματος, μόνο το πλήθος τους.

Θέμα 2 (25%)

α. (5%) *Παρακάτω δίνεται η λύση του Peterson για την υλοποίηση κλειδώματος (lock) για δύο διεργασίες στο λογισμικό. Εξηγήστε τι κάνει το συγκεκριμένο τμήμα κώδικα και γιατί η λύση αυτή δεν λειτουργεί στα συστήματα του πραγματικού κόσμου.*

```
1    do {
2        flag[me] = TRUE;
3        turn = other;
```

```
4     while (flag[other] && turn == other)
5        ;
6        //critical section
7     flag[me] = FALSE;
8     //remainder section
9  } while (TRUE);
```

Στο παραπάνω τμήμα κώδικα κάθε διεργασία δηλώνει την πρόθεσή της να μπει στο κρίσιμο τμήμα (ΚΤ) στη γραμμή 2 και παραχωρεί τη σειρά της στην άλλη διεργασία στη γραμμή 3. Στη συνέχεια ανάλογα με την τιμή των μεταβλητών `flag[]` και `turn` μία από τις 2 διεργασίες θα εισέλθει στο ΚΤ ενώ ή άλλη θα περιμένει στο βρόχο while.
Η παραπάνω λογική δεν είναι εξασφαλισμένο ότι θα λειτουργήσει σε ένα πραγματικό σύστημα, γιατί εκεί ο μεταγλωττιστής ή ο επεξεργαστής είναι δυνατόν να αναδιατάξουν εντολές που δεν έχουν εξάρτηση μεταξύ τους. Για παράδειγμα, ο μεταγλωττιστής ή ο επεξεργαστής μπορεί για λόγους βελτιστοποίησης να αναδιατάξουν τις εντολές στις γραμμές 2 και 3. Αν συμβεί αυτό και με μία αλληλουχία εκτέλεσης εντολών από τις 2 διεργασίες που φαίνεται στο παράδειγμα που ακολουθεί (επισημαίνουμε με `Pi: statement` μία εντολή που εκτελείται από τη διεργασία P_i) τότε και οι δύο διεργασίες θα εισέλθουν στο κρίσιμο τμήμα, κάτι που προφανώς αντίκειται στον ορισμό του ΚΤ.

```
1  //initially flag[1] = FALSE, flag[2] = FALSE
2  P2: turn = 1;
3  P1: turn = 2;
4  P1: flag[1] = TRUE;
5  P1: while (flag[2] && turn == 2);
6      // enters critical section because flag[2] == FALSE
7  P2: flag[2] = TRUE
8  P2: while (flag[1] && turn == 1);
9      // enters critical section because turn == 2
```

β. (10%) *Σε πολλές περιπτώσεις κατά την εκτέλεση ταυτόχρονων ή παράλληλων προγραμμάτων με πολλές διεργασίες υπάρχει η ανάγκη όλες οι διεργασίες να ξεκινήσουν την εκτέλεση κάποιου τμήματος κώδικα ταυτόχρονα: οι διεργασίες που καταφθάνουν πρώτες σε ένα σημείο του κώδικα περιμένουν όλες τις υπόλοιπες. Όταν όλες έχουν έρθει σε αυτό το σημείο, τότε η εκτέλεση της κάθε μιας συνεχίζεται από την επόμενη εντολή. Το σχήμα συγχρονισμού που υλοποιεί αυτή την απαίτηση ονομάζεται "φράγμα συγχρονισμού" (barrier). Δώστε κώδικα που υλοποιεί το barrier για N διεργασίες, χρησιμοποιώντας σημαφόρους και μία κοινή μεταβλητή.*

Η λύση για το barrier N διεργασιών δίνεται στον κώδικα που ακολουθεί. Οι διεργασίες αυξάνουν την τιμή της κοινής μεταβλητής `count` και περιμένουν στο σημαφόρο `barrier`. Όταν και η τελευταία διεργασία καταφθάσει εκεί, η τιμή της `count` γίνεται N και οι διεργασίες "απελευθερώνουν" η μία την άλλη αλυσιδωτά. Επισημαίνεται ότι η λύση αυτή δουλεύει αν οι διεργασίες δεν επανέλθουν στο σημείο "rendezvous" δηλαδή υποθέτουμε ότι το τμήμα κώδικα δεν βρίσκεται μέσα σε βρόχο.

```
1  mutex = semaphore(1);
2  barrier = semaphore(0);
3  int count = 0; //shared variable
4
5  // rendezvous
6  wait(mutex);
```

```
 7  count++;
 8  if (count == N) signal(barrier);
 9  signal(mutex);
10  wait(barrier);
11  signal(barrier);
12  //critical point
```

γ. (10%) *Περιγράψτε το πρόβλημα των αναγνωστών-εγγραφέων (readers-writers problem). Δώστε λύση που να αποφεύγει τη λιμοκτονία για τους εγγραφείς.*

Ως πρόβλημα των αναγνωστών-εγγραφέων αναφέρουμε το σχήμα συγχρονισμού κατά το οποίο στο κρίσιμο τμήμα επιτρέπεται να βρίσκονται είτε οσεσδήποτε διεργασίες πραγματοποιούν αναγνώσεις σε μία κοινή δομή δεδομένων (αναγνώστες) είτε μία μόνο διεργασία που πραγματοποιεί εγγραφές (εγγραφέας).
Λύση που να αποφεύγει τη λιμοκτονία στους εγγραφείς δίνεται στη συνέχεια. Θεωρούμε ότι υπάρχει ένα νοητό σημείο αναμονής που υλοποιείται από το σημαφόρο `waiting_room` στο οποίο ο εγγραφέας υποχρεώνει όλες τις επόμενες διεργασίες να περιμένουν όταν ο ίδιος βρίσκεται στο ΚΤ ή σε αναμονή προκειμένου να εισέλθει σε αυτό.

```
mutex  = semaphore(1);
critical = semaphore(1);
waiting_room = semaphore(1);
int readcount = 0; //shared variable
```

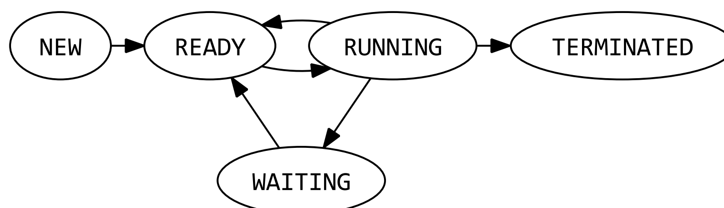
```
// reader:
wait(waiting_room);
wait(mutex);
if (++readcount == 1) wait(critical);
signal(mutex);
signal(waiting_room);

// critical section

wait(mutex);
if (--readcount == 0) signal(critical);
signal(mutex);
```

```
// writer:
wait(waiting_room);
wait(critical);
signal(waiting_room);

// critical section

signal(critical);
```

Θέμα 3 (30%)

α. (5%) *Σχεδιάστε ενδεικτικό διάγραμμα καταστάσεων διεργασίας, με τουλάχιστον τις καταστάσεις RUNNING, WAITING, READY, TERMINATED. Αναφέρετε μια αιτία για κάθε μετάβαση.*

Ένα ενδεικτικό διάγραμμα καταστάσεων διεργασίας παρουσιάζεται στο ακόλουθο σχήμα.



- NEW → READY: αποδοχή της διεργασίας για εκτέλεση στο σύστημα.
- 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.
- 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.
- READY → RUNNING: επιλογή από τον χρονοδρομολογητή για εκτέλεση στη CPU, context switch σε αυτή τη διεργασία.
- RUNNING → TERMINATED: κλήση της `exit()` από την τρέχουσα διεργασία.

β. (10%) *Σε έναν υπολογιστικό κόμβο οι χρήστες υποβάλουν εργασίες προς εκτέλεση, δεν αλληλεπιδρούν καθόλου με αυτές και στο τέλος συλλέγουν τα αποτελέσματα. Οι διαχειριστές του συγκεκριμένου συστήματος επιθυμούν να ελαχιστοποιήσουν το μέσο χρόνο αναμονής των χρηστών. Προτείνετε αλγόριθμο δρομολόγησης που να επιτυγχάνει την παραπάνω απαίτηση. Τι πληροφορία θα πρέπει να δίνουν οι χρήστες κατά την υποβολή των εργασιών τους ώστε να μπορεί να εκτελεστεί ο αλγόριθμος; Περιγράψτε επέκταση του παραπάνω αλγορίθμου που να αποφεύγει το φαινόμενο της λιμοκτονίας. Δώστε τον αλγόριθμο ή σε ψευδοκώδικα με σχόλια/επεξηγήσεις (κατά προτίμηση) ή σε φυσική γλώσσα.*

Ο αλγόριθμος που ελαχιστοποιεί το μέσο χρόνο αναμονής είναι ο SJF (Shortest Job First). Για να συμβεί αυτό ο χρήστης θα πρέπει να πληροφορεί το σύστημα για το χρόνο εκτέλεσης της εργασίας του. Άρα, κατά την υποβολή ο χρήστης θα πρέπει να δίνει στο σύστημα την πληροφορία αυτή. Ο παραπάνω αλγόριθμος διατρέχει κίνδυνο λιμοκτονίας, καθώς μεγάλες σε διάρκεια εργασίες μπορεί να αναμένουν στο διηνεκές για την εκτέλεση μικρότερων εργασιών που καταφθάνουν δυναμικά. Θα αντιμετωπίσουμε το παραπάνω πρόβλημα με την τακτική της γήρανσης (aging). Η βασική ιδέα είναι ότι όταν μία εργασία έχει παραμείνει για μεγάλο χρονικό διάστημα στο σύστημα θα επιλέγεται ανεξαρτήτως χρόνου εκτέλεσης. Ο αλγόριθμος της χρονοδρομολόγησης χρειάζεται να διατηρεί δύο δομές έτοιμων διεργασιών: Τη δομή `SJF_list` στην οποία οι εργασίες είναι ταξινομημένες κατά αύξοντα χρόνο εκτέλεσης, και τη δομή `OLD_list` που είναι μία ουρά FIFO.
Στην `SJF_list` πραγματοποιούνται λειτουργίες εισαγωγής μιας διεργασίας `p` (`SJF_list.insert(p)`), εξαγωγής της διεργασίας με τον ελάχιστο χρόνο εκτέλεσης (`SJF_list.extract_min()`), αφαίρεσης μιας διεργασίας `p` (`SJF_list.remove(p)`) και ελέγχου αν η λίστα έχει διεργασίες(`non_empty(OLD_list)`). Αντίστοιχα στη δομή `OLD_list` πραγματοποιούνται λειτουργίες εισαγωγής και διαγραφής (`OLD_list.enqueue(p)` και `OLD_list.dequeue()`) και έλεγχος αν η λίστα έχει διεργασίες. Θεωρούμε ότι κατά την υποβολή της στο σύστημα η διεργασία τοποθετείται στην δομή `SJF_list` με κλήση της `SJF_list.insert(p)` (καταλαμβάντονας την κατάλληλη θέση στη δομή). Ο αλγόριθμος χρονοδρομολόγησης σε ψευδοκώδικα είναι ο εξής:

```
1   if (non_empty(OLD_list)) { // first check aged jobs and
2      p = OLD_list.dequeue();  // dispatch them to the system
3      dispatch(p);
4   } else if (non_empty(OLD_list)) { // if there are no aged jobs apply SJF
5      p = SJF_list.extract_min();    // extract shortest job
6
7      for (q in SJF_list) {           // update ages of all remaining jobs
8         q.age++;
9         if (q.age > age_threshold) { // if age exceeds threshold
10           SJF_list.remove(q);       // remove from SJF_list
11           OLD_list.enqueue(q);      // and insert to OLD_list
12        }
13     }
```

```
14       dispatch(p);
15   }
```

γ. (15%) *Θεωρήστε ένα πρόγραμμα αναπαραγωγής μουσικής από αρχεία MP3 που εκτελείται στο κινητό σας τηλέφωνο. Το πρόγραμμα εκτελείται σε δύο διεργασίες: η P0 αποκωδικοποιεί το αρχείο MP3 σε πακέτα ήχου PCM (συνάρτηση `decode()`), τα οποία αποστέλλει μέσω σωλήνωσης (pipe `pd`) στην P1. Η P1 αναλαμβάνει να στείλει τα πακέτα στην κάρτα ήχου (συνάρτηση `play_sound()`), οπότε ακούγεται μουσική από το ηχείο. Θεωρήστε τα εξής:*

- *Κάθε τελικό πακέτο ήχου PCM είναι μήκους `L` bytes.*
- *Η `P1` μπλοκάρει κατά την εκτέλεση της `play_sound()`. Μένει σε αυτή την κατάσταση έως ότου η κάρτα ήχου έχει παίξει ολόκληρο το πακέτο (υποθέτουμε ότι η κάρτα ήχου υποστηρίζει DMA ώστε να μην εμπλέκει τη CPU).*
- *Ο buffer που χρησιμοποιεί εσωτερικά το ΛΣ για την υλοποίηση του pipe χωράει ακριβώς 3 πακέτα ήχου.*
- *Η κλήση συστήματος `write()` σε γεμάτο pipe μπλοκάρει.*
- *Η αποκωδικοποίηση απαιτεί χρόνο C για την παραγωγή ενός πακέτου μήκους L, ενώ η αναπαραγωγή του από το ηχείο απαιτεί χρόνο $P = 5 \times C$.*
- *Το τηλέφωνο διαθέτει μία CPU και τρέχει μόνο τις δύο διεργασίες. Οι `read()`, `write()` έχουν αμελητέα ανάγκη υπολογισμού και το context switch έχει αμελητέα επιβάρυνση. Το αρχείο MP3 θεωρούμε ότι είναι ήδη φορτωμένο στη μνήμη.*

Σας δίνεται ο κώδικας που εκτελούν οι δύο διεργασίες:

```
void P0(int pd[])
{
    int i; char buf[L];

    for (i = 0; i < PACKET_COUNT; i++) {
        decode(FILE, i, buf);
        write(pd[1], buf, L);
    }
}
```

```
void P1(int pd[])
{
    int i; char buf[L];

    for (i = 0; i < PACKET_COUNT; i++) {
        read(pd[0], buf, L);
        play_sound(buf, L);
    }
}
```

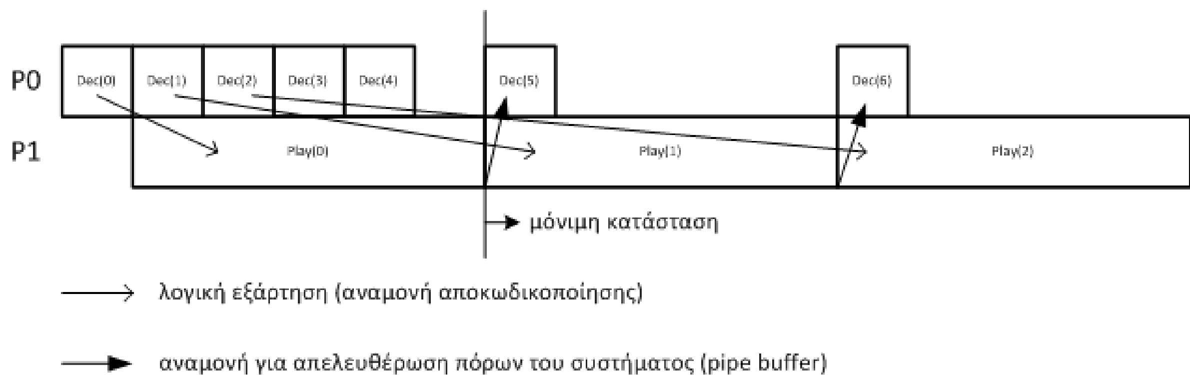
Ζητούνται τα εξής:

1. *Σε ποια κατάσταση είναι κυρίως η P1; Σε ποια γραμμή βρίσκεται όταν είναι σε αυτή την κατάσταση και πόσο χρόνο μένει εκεί κάθε φορά; Τι συμβαίνει για να φύγει από εκεί;*
2. *Ποιες καταστάσεις περνά η P0; Γιατί δεν καταλαμβάνει το 100% της CPU;*
3. *Σχεδιάστε διάγραμμα ως προς το χρόνο όπου να φαίνεται για τις δύο διεργασίες χωριστά σε ποια φάση της εκτέλεσής τους βρίσκονται ανά πάσα στιγμή. Συμβολίστε `DEC(i)`, `PLAY(i)` τις φάσεις αποκωδικοποίησης και αναπαραγωγής του `i`-οστού πακέτου, αντίστοιχα. Χρησιμοποιήστε βέλη για να δείξετε τις εξαρτήσεις ανάμεσα στις δύο φάσεις.*
4. *Ποιος είναι ο βαθμός χρησιμοποίησης της CPU στη μόνιμη κατάσταση;*
5. *Για λόγους εξοικονόμησης ενέργειας το κινητό μειώνει στο μισό το ρολόι της CPU, θεωρήστε ότι αυτό μειώνει στο μισό την ταχύτητά της. Ποιος είναι ο νέος βαθμός χρησιμοποίησης της CPU;*
6. *Στη μέση του κομματιού στέλνουμε `SIGSTOP` στην P0. Θα διακοπεί η αναπαραγωγή; Αν ναι, πότε; Σε ποια γραμμή και σε ποια κατάσταση είναι οι P0, P1 τότε;*

1. Η P1 βρίσκεται σε κατάσταση `WAITING` αφού τροφοδοτεί την κάρτα ήχου με πακέτα και παραμένει μπλοκαρισμένη για όλο το διάστημα κατά το οποίο αναπαράγεται ένα πακέτο. Από

την κατάσταση αυτή βγαίνει όταν ολοκληρωθεί η αναπαραγωγή του πακέτου και η διεργασία διαβάσει από το pipe τα δεδομένα του επόμενου πακέτου. Καθώς η ανάγνωση από το pipe διαρκεί αμελητέο χρόνο, η διεργασία θα επανέλθει αμέσως σε κατάσταση `WAITING`.

2. Η διεργασία P0 εναλλάσσεται στις καταστάσεις `RUNNING` και `WAITING`. Μετά την αποκωδικοποίηση διαδοχικών πακέτων θα γεμίσει τον buffer του pipe έτσι που η επόμενη εγγραφή θα βρει τον buffer γεμάτο και η διεργασία θα μπλοκάρει. Αυτό συμβαίνει γιατί ο ρυθμός με τον οποίο αποκωδικοποιούνται τα πακέτα είναι 5 φορές μεγαλύτερος από το ρυθμό αναπαραγωγής τους.
3. Το διάγραμμα δίνεται στη συνέχεια.



4. Όπως φαίνεται και στο διάγραμμα, ο βαθμός χρησιμοποίησης της CPU στη μόνιμη κατάσταση είναι $\frac{C}{P} = \frac{C}{5C} = 20\%$.
5. Σε αυτή την περίπτωση ο χρόνος της αποκωδικοποίησης θα διπλασιαστεί, ενώ ο χρόνος αναπαραγωγής θα μείνει σταθερός, άρα τώρα θα ισχύει $P = 2.5C$. Ο βαθμός χρησιμοποίησης θα γίνει $\frac{C}{P} = \frac{1}{2.5} = 40\%$.
6. Όπως φαίνεται στο σχήμα, κατά την αναπαραγωγή του πακέτου k η διεργασία P0 βρίσκεται μπλοκαρισμένη προσπαθώντας να γράψει το πακέτο $k+4$ στο pipe. Επομένως, έχει ήδη γράψει στο pipe τα πακέτα $k+1$, $k+2$ και $k+3$. Άρα, ακόμα και αν σταματήσει η εκτέλεσή της με το σήμα `SIGSTOP` η αναπαραγωγή δεν θα σταματήσει άμεσα, παρά μόνο όταν αναπαραχθούν και τα τρία παραπάνω πακέτα, οπότε και η P1 θα μπλοκάρει προσπαθώντας να διαβάσει από άδειο pipe.

Θέμα 4 (20%)

α. (8%) *Απαντήστε συνοπτικά, όχι πάνω από δύο-τρεις γραμμές:*

i. *Με ποιον τρόπο βοηθά η υποστήριξη modify bit από το υλικό την υλοποίηση αποδοτικού μηχανισμού για σελιδοποίηση κατ'απαίτηση από το ΛΣ;*

ii. *Τι θα γινόταν αν μπορούσε μια διεργασία χρήστη να ελέγξει τον χρονιστή (timer) του συστήματος; Πώς αποτρέπεται αυτό το ενδεχόμενο;*

i. Όταν το ΛΣ χρειάζεται να αντικαταστήσει μια σελίδα, αν το modify bit δεν είναι αναμμένο, ξέρει ότι δεν χρειάζεται να γράψει το περιεχόμενο του πλαισίου μνήμης στο δίσκο, πριν διαβάσει μια νέα σελίδα εκεί.

ii. Θα μπορούσε να τρέχει για πάντα, σταματώντας το μηχανισμό χρονοδρομολόγησης. Το ενδεχόμενο αυτό αποτρέπεται γιατί οι εντολές για χειρισμό του χρονιστή είναι προνομιούχες (privileged) κι εκτελούνται μόνο από κατάσταση επόπτη.

β. *(12%)* *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. *Η μεταβλητή* `var` *βρίσκεται σε άλλη διεύθυνση* `&var` *στον πατέρα απ'ότι στο παιδί.*
3. *Η διεύθυνση* `&var` *μιας μεταβλητής είναι φυσική διεύθυνση.*
4. *Αμέσως μετά το* `fork()`*, κάθε εγγραφή της* `var` *από το παιδί καταλήγει στην ίδια φυσική διεύθυνση με κάθε εγγραφή της* `var` *από τον πατέρα.*
5. *Η εντολή* `var = 3` *στον πατέρα προκαλεί page fault.*
6. *Η εντολή* `var = 3` *στο παιδί προκαλεί page fault.*
7. *Όταν ο Program Counter μιας διεργασίας βρεθεί σε διεύθυνση για την οποία η αντίστοιχη εγγραφή στον πίνακα σελίδων της έχει το permission bit* Execute *σβηστό, τερματίζεται πάντα με Segmentation Fault.*
8. *Όταν μια διεργασία επιχειρεί να γράψει σε διεύθυνση για την οποία δεν υπάρχει έγκυρη εγγραφή στον πίνακα σελίδων της, τερματίζεται πάντα με Segmentation Fault.*
9. *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. Ψευδές. Η μνήμη – ο χώρος εικονικών διευθύνσεων – του παιδιού είναι αντίγραφο της μνήμης – του χώρου εικονικών διευθύνσεων – του πατέρα και δημιουργείται κατά το `fork()`. Από την οπτική γωνία του παιδιού, οι μεταβλητές παραμένουν στις ίδιες (εικονικές) διευθύνσεις μνήμης, και οι δείκτες που είχαν τεθεί πριν από το `fork()` (π.χ. `p = &var`) συνεχίζουν να έχουν νόημα.
3. Ψευδές. Κάθε διεύθυνση η οποία χρησιμοποιείται σε εντολές `LOAD`/`STORE`, κάθε διεύθυνση στην οποία αναφέρεται η διεργασία, είναι εικονική διεύθυνση. Οι δείκτες αναφέρονται σε εικονικές διευθύνσεις, βλ. και προηγούμενο ερώτημα.
4. Ψευδές. Λόγω COW, ο πατέρας και το παιδί μοιράζονται πλαίσια χωρίς δικαίωμα εγγραφής. Κάθε εγγραφή προκαλεί page fault, οπότε το ΛΣ αναθέτει ένα νέο πλαίσιο στη σελίδα που το προκάλεσε. Σε αυτό το πλαίσιο καταλήγει η αρχική εγγραφή.
5. 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.
6. Ομοίως. Η κατάσταση πατέρα-παιδιού είναι συμμετρική μετά από `fork()` με COW.
7. Αληθές. Ο πυρήνας βλέπει ότι η διεργασία προσπαθεί να εκτελέσει κώδικα από απεικόνιση σελίδας χωρίς το Execute bit, οπότε στέλνει σήμα `SIGSEGV` στη διεργασία. Αν αυτή δεν το χειρίζεται, θα τερματιστεί.
8. Ψευδές. Προκαλείται page fault και το ΛΣ συμβουλεύεται το χάρτη μνήμης του για τη διεργασία. Είναι δυνατό, π.χ., η αντίστοιχη σελίδα να είναι στο δίσκο, οπότε δεν υπάρχει εγγραφή στον πίνακα σελίδων, παρόλο που η συγκεκριμένη διεύθυνση είναι έγκυρη μνήμη για τη διεργασία. Το ΛΣ θα φέρει τη σελίδα στη φυσική μνήμη, και θα ανανεώσει αναλόγως τον πίνακα σελίδων, πριν επανεκκινήσει την εντολή που προκάλεσε το page fault. Η διεργασία δεν θα καταλάβει τίποτε.
9. Αληθές. Μια διεργασία δεν έχει δικαίωμα να διαβάσει διεύθυνση σε περιοχή μνήμης για την οποία δεν είναι ενήμερο το ΛΣ. Αν δεν υπάρχει έγκυρη περιοχή στο χάρτη μνήμης για διεύθυνση στην οποία η διεργασία επιχείρησε πρόσβαση, το ΛΣ θα αποστείλει σήμα `SIGSEGV` στη διεργασία.