Λειτουργικά Συστήματα Υπολογιστών
2η Εργαστηριακή Άσκηση

Συνεργάτες:

- **Καλλάς Κωσταντίνος Α.Μ:03112057**
- **Τζίνης Ευθύμιος Α.Μ:03112007**

Για την πιο ευανάγνωστη μορφή της εργασίας μας απαντάμε σειριακά σε όλες τις ασκήσεις και προκειμένου να μην παρεμβάλλουμε τεράστια κομμάτια κώδικα αναφερόμαστε σε αυτά σε όλες τις ασκήσεις με αναφορές που παραπέμπουν στο τελευταίο κομμάτι της αναφοράς που περιλαμβάνει όλους τους κώδικες.
Τα υπογραμμισμένα και bold είναι αναφορές στο τμήμα της αναφοράς με τους κώδικες.

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Σε αυτή την άσκηση φτιάξαμε ένα πρόγραμμα όπου χειροκίνητα φτάχνει το ζητούμενο δένδρο διεργασιών που δείχνεται στην εικόνα της εκφώνησης της άσκησης.
Σε αυτό το πρόγραμμα χρησιμοποιήσαμε στα waiting κάποιο χρόνο όπου φαίνεται στα definitions στην αρχή του κώδικά μας και ουσιαστικά χρησιμοποιείται για τον έλεγχο των διεργασιών μέσω του χρόνου.
Επιπλέον, όλες οι διεργασίες κατευθείαν μόλις ξεκινούν κοιμούνται επομένως εκτυπώνουν τα αντίστοιχα μηνύματα.

Ο κώδικας εργασίας είναι ο **Code.1.1**

Για την έξοδο στο παράδειγμα της εκφώνησης εκτελούμε στο shell:

```
oslabf04@amorgos:~/Exercises/Exercise2/Erwthma1$ make ask2-fork
gcc -g -Wall -O2 -c ask2-fork.c
gcc -g -Wall -O2 ask2-fork.o proc-common.o -o ask2-fork
```

Και παίρνουμε την έξοδο:

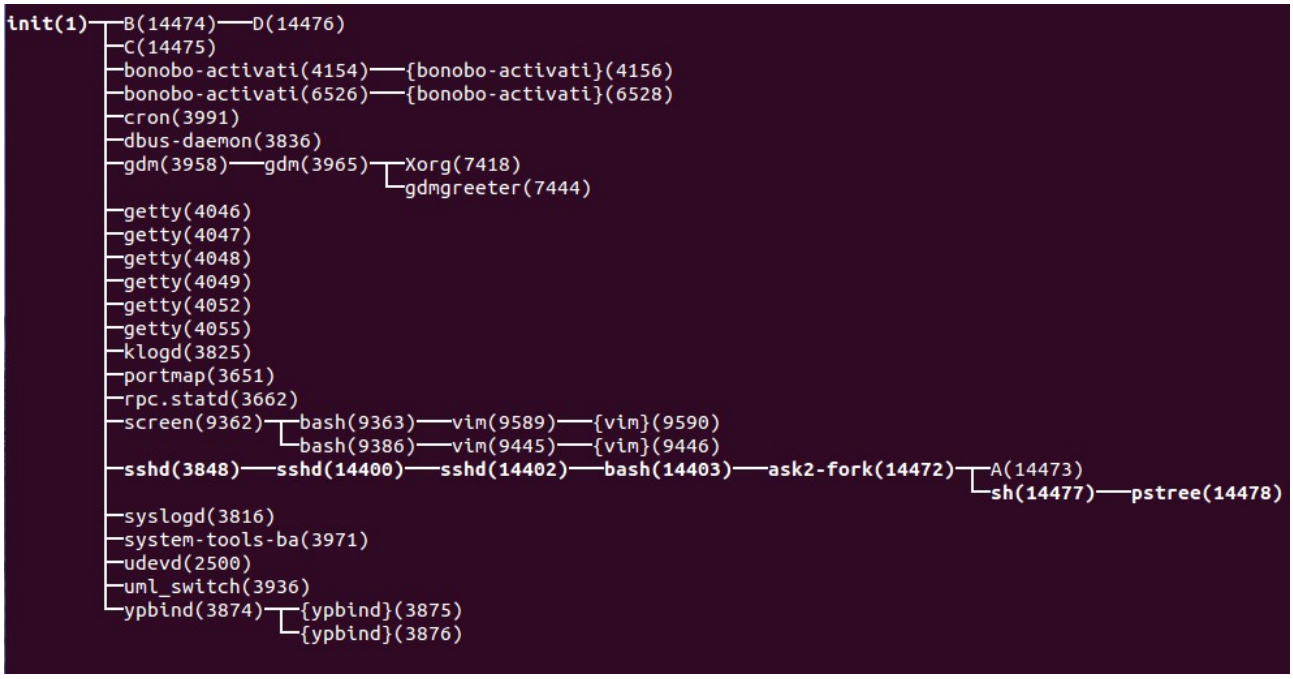
```
oslabf04@amorgos:~/Exercises/Exercise2/Erwthma1$ ./ask2-fork
A: Sleeping...
B: Sleeping...
C: Sleeping...
D: Sleeping...


A(15751)─┬─B(15752)──D(15754)
         └─C(15753)


C: Exiting...
D: Exiting...
My PID = 15752: Child PID = 15754 terminated normally, exit status = 13
B: Exiting...
My PID = 15751: Child PID = 15752 terminated normally, exit status = 19
My PID = 15751: Child PID = 15753 terminated normally, exit status = 17
A: Exiting...
My PID = 15750: Child PID = 15751 terminated normally, exit status = 16
```

Ερωτήσεις:

1. Τι θα γίνει αν τερματίσετε πρόωρα τη διεργασία A, δίνοντας kill -KILL <pid>, όπου <pid> το Process ID της;

Τροποποιούμε ελάχιστα τον κώδικά μας όπως φαίνεται παρακάτω προκειμένου να σκοτώσουμε πρόωρα την γονική διεργασία Α με την χρήση της kill() . Στον κώδικα **Code.1.1** απλά αφαιρούμε τα σχόλια μπροστά από τις γραμμές που έχουν σχόλια της μορφής //**.

Αν τρέξουμε τώρα πάλι τον κώδικά μας με το ίδιο δέντρο διεργασιών παίρνουμε στην έξοδο τυπώνοντας το δέντρο της init με την χρήση του show_pstree(1) την παρακάτω έξοδο:

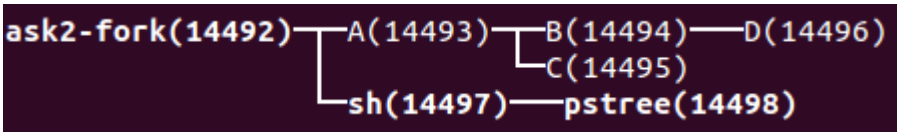


Αυτό σημαίνει απλά ότι επειδή η γονική διεργασία Α πέθανε πρόωρα χωρίς να αναμένει τον τερματισμό των παιδιών της, τα παιδιά της γίνονται zombie (δηλαδή όλο το υπόλοιπο δέντρο) υϊοθετούνται από την init που κάνει μονίμως wait. Όταν τα παιδιά τερματίσουν φεύγουν από την Init και τελικά και από την μνήμη.

2.Τι θα γίνει αν κάνετε show_pstree(getpid()) αντί για show_pstree(pid) στη main() ; Ποιες επιπλέον διεργασίες φαίνονται στο δέντρο και γιατί;

Με την χρήση του show_pstree(getpid()) αντί για show_pstree(pid) στη main() παίρνουμε το δέντρο που έχει ώς αρχική ρίζα την main Και όχι την Α. Αυτό συμβαίνει διότι όταν κάνουμε getpid() σε μία διεργασία μας επιστρέφει το pid της διεργασίας αυτής ενώ το pid αναφέρεται στο pid της Α η οποία έχει δημιουργηθεί με fork() από την main.

Παρακάτω βλέπουμε το δέντρο που εκτυπώνεται και εύκολα βλέπουμε τις παραπάνω διεργασίες που υπάρχουν στο δέντρο λόγω της main (πιο συγκεκριμένα έχουε ότι μέσω της main καλούμε την show_pstree η οποία καλεί τις διεργασίες sh και pstree):



3.Σε υπολογιστικά συστήματα πολλαπλών χρηστών, πολλές φορές ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης. Γιατί;

Προφανώς ο διαχειριστής προκειμένου να εξασφαλίσει την ακεραιότητα του συστήματος θα απαγορέψει έναν πολύ μεγάλο αριθμό διεργασιών που δημιουργούνται μέσω της fork()
Για παράδειγμα ένας κακόβουλος user θα μπορούσε να γράψει το εξής πρόγραμμα:

```
while(1)
{
        pid=fork()
}
```

Και να κρασάρει όλο το σύστημα αν ο διαχειριστής τον αφήσει να το κάνει καθώς θα δημιουργεί συνεχώς διεργασίες που αυτές με την σειρά τους άλλες διεργασίες και αυτός ο κύκλος θα σταματήσει μόνο όταν πέσει το σύστημα. (Όταν δεν υπάρχουν άλλοι πόροι)

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Με ακριβώς παρόμοια διαδικασία με πριν φτιάχνουμε και αυτό το πρόγραμμα το οποίο χρησιμοποιεί έλεγχο των διεργασιών μέσω χρονοκαθυστερήσεων απλά χρησιμοποιούμε τις δομές και τις συναρτήσεις που μας δίνονται στην βιβλιοθήκη.

Ο κώδικας εργασίας είναι στο τμήμα κώδικα **Code.1.2**

Ενώ όταν τρέξουμε το πρόγραμμά μας με όρισμα το proc.tree που μας δίνεται τότε παίρνουμε την εξής έξοδο:

```
oslabf04@ithaki:~/Exercises/Exercise2/Erwthma2$ ./ask2-fork proc.tree
A: Sleeping...
B: Sleeping...
C: Sleeping...
D: Sleeping...
E: Sleeping...
F: Sleeping...


A(14507)─┬─B(14508)─┬─E(14511)
         │          └─F(14512)
         ├─C(14509)
         └─D(14510)


C: Exiting...
D: Exiting...
E: Exiting...
F: Exiting...
My PID = 14508: Child PID = 14511 terminated normally, exit status = 0
My PID = 14508: Child PID = 14512 terminated normally, exit status = 0
B: Exiting...
My PID = 14507: Child PID = 14508 terminated normally, exit status = 0
My PID = 14507: Child PID = 14509 terminated normally, exit status = 0
My PID = 14507: Child PID = 14510 terminated normally, exit status = 0
A: Exiting...
My PID = 14506: Child PID = 14507 terminated normally, exit status = 0
A
        B
                E
                F
        C
        D
```

Ερωτήσεις:

1. Με ποια σειρά εμφανίζονται τα μηνύματα έναρξης και τερματισμού των διεργασιών; Γιατί;

Όπως φαίνεται και από τον κώδικά μας η σειρά έναρξης και τερματισμού των διεργασιών ουσιαστικά ελέγχεται μόνο από τον timer της κάθε διεργασίας. Αυτό επιτρέπει στον επεξεργαστή να επέμβει μη ντετερμινιστικά προς τον χρήστη για την επιλογή ποιάς διεργασίας θα εκτελεστεί σε μια συγκεκριμένη χρονική στιγμή. Το παράδειγμα που μας είχε δωθεί ήταν πολύ μικρό και έτσι δεν αναδείχθηκε η όλη εξάρτηση από τον χρονοδρομολογητή του επεξεργαστή μας. Τα πράγματα σε αυτή την περίπτωση θα μπορούσαν να "ξεφύγουν τελείως" από τον έλεγχο του προγραμματιστή, αν είχαμε πολλούς επεξεργαστές όπου ο καθένας θα ήτσαν ελεύθερος να τρέξει όποια διεργασία θέλει.

Φτιάξαμε ένα δικό μας δένδρο και τρέξαμε το συγκεκριμένο πρόγραμμα για να αναδείξουμε τον μη ντετερμινισμό του επεξεργαστή που διατυπώνεται παραπάνω:

```
oslabf04@lemnos:~/Exercises/Exercise2/Erwthma2$ ./ask2-fork my_tree.tree
A: Sleeping...
A1: Sleeping...
A3: Sleeping...
A2: Sleeping...
B3: Sleeping...
R: Sleeping...
B2: Sleeping...
B4: Sleeping...
B1: Sleeping...
C1: Sleeping...
X: Sleeping...
Y: Sleeping...
C2: Sleeping...
D1: Sleeping...
Z: Sleeping...
Y1: Sleeping...
Y2: Sleeping...
Y3: Sleeping...


A(12460)─┬─A1(12461)─┬─B1(12464)─┬─C1(12470)──D1(12474)
         │           │           └─C2(12472)
         │           ├─B2(12466)
         │           ├─B3(12467)
         │           └─B4(12468)─┬─X(12469)
         │                       ├─Y(12471)──Y1(12475)──Y2(12476)──Y3(12477)
         │                       └─Z(12473)
         ├─A2(12462)──R(12465)
         └─A3(12463)


A3: Exiting...
B2: Exiting...
C2: Exiting...
D1: Exiting...
Y3: Exiting...
My PID = 12470: Child PID = 12474 terminated normally, exit status = 0
C1: Exiting...
```

1.3 Αποστολή και χειρισμός σημάτων

Σε αυτή την άσκηση προκειμένου να έχουμε καλύτερο χειρισμό των διεργασιών από την προηγούμενη μέθοδο με τον timer χρησιμοποιούμε σήματα προκειμένου ένας γονέας να μπορεί να πεθάνει μόνο όταν όλα του τα παιδιά έχουν πεθάνει και κατά συνέπεια και τα παιδιά του με τον ίδιο τρόπο. Έτσι υλοποιείται η Depth First εκτύπωση των μηνυμάτων των διεργασιών όταν αυτές πεθαίνουν. Προφανώς πάλι όταν δημιουργούμε τις διεργασίες υπάρχει μη ντετερμινισμός στο ποιά θα δημιουργηθεί πρώτη αφού αυτό πάλει εξαρτάται από την προτεραιότητα πού θα δώσει ο χρονοδρομολογητής του επεξεργαστή σε κάθε διεργασία παιδί.

Ο κώδικας εργασίας μας είναι στο τμήμα κώδικα **Code.1.3.**

Όταν τρέξουμε το πρόγραμμά μας για το δοθέν δέντρο της εκφώνησης παίρνουμε το εξής αποτέλεσμα:

```
oslabf04@lemnos:~/Exercises/Exercise2/Erwthma3$ ./ask3-signals proc.tree
PID = 12636, name A, starting...
PID = 12639, name D, starting...
PID = 12638, name C, starting...
PID = 12637, name B, starting...
My PID = 12636: Child PID = 12638 has been stopped by a signal, signo = 19
My PID = 12636: Child PID = 12639 has been stopped by a signal, signo = 19
PID = 12640, name E, starting...
My PID = 12637: Child PID = 12640 has been stopped by a signal, signo = 19
PID = 12641, name F, starting...
My PID = 12637: Child PID = 12641 has been stopped by a signal, signo = 19
My PID = 12636: Child PID = 12637 has been stopped by a signal, signo = 19
My PID = 12635: Child PID = 12636 has been stopped by a signal, signo = 19


A(12636)─┬─B(12637)─┬─E(12640)
         │          └─F(12641)
         ├─C(12638)
         └─D(12639)


PID = 12636, name = A is awake
PID = 12637, name = B is awake
PID = 12640, name = E is awake
E: Adios Amigos...
My PID = 12637: Child PID = 12640 terminated normally, exit status = 0
PID = 12641, name = F is awake
F: Adios Amigos...
My PID = 12637: Child PID = 12641 terminated normally, exit status = 0
B: Adios Amigos...
My PID = 12636: Child PID = 12637 terminated normally, exit status = 0
PID = 12638, name = C is awake
C: Adios Amigos...
My PID = 12636: Child PID = 12638 terminated normally, exit status = 0
PID = 12639, name = D is awake
D: Adios Amigos...
My PID = 12636: Child PID = 12639 terminated normally, exit status = 0
A: Adios Amigos...
My PID = 12635: Child PID = 12636 terminated normally, exit status = 0
```

Ερωτήσεις:

1. Στις προηγούμενες ασκήσεις χρησιμοποιήσαμε τη sleep() για τον συγχρονισμό των διεργασιών. Τι πλεονεκτήματα έχει η χρήση σημάτων;

Η χρήση των σημάτων μας επιτρέπει να έχουμε καλύτερο χειρισμό των διεργασιών αφού κάθε παιδί προκειμένου να εκτελεστεί περιμένει σήμα SIGCONT από τον πατέρα αφού αυτό είναι σε sleep mode αφού έχει εκτελέσει raise(SIGSTOP). Επιπλέον, αυτή η τεχνική μας επιτρέπει όσο μεγάλο και αν είναι το δένδρο να μην χρειάζεται να υπολογίζουμε τους timer που θα θέσουμε όπως στην προηγούμενη τεχνική. Γενικότερα η όλη διαδικασία τείνει να γίνει τελείως ντετερμινιστική σε αντίθεση με την προηγούμενη.

2. Ποιος ο ρόλος της wait_for_ready_children(); Τι εξασφαλίζει η χρήση της και τι πρόβλημα θα δημιουργούσε η παράλειψή της;

Ο ρόλος της wait_for_ready_children() είναι ότι κάθε πατέρας για να πεθάνει πρέπει να περιμένει να πεθάνουν όλα του τα παιδιά και ουσιαστικά είναι ο λόγος που οι διεργασίες εμφανίζουν τα μηνυμάτα τους σε Depth First σειρά. Ουσιαστικά για να πεθάνει οποιοσδήποτε κόμβος του δένδρου θα πρέπει όλοι οι κόμβοι που βρίσκονται κάτω από αυτόν ιεραρχικά στο δέντρο να πεθάνουν. Αν την είχαμε παραλείψει ουσιαστικά δεν θα υπήρχε ο περιορισμός που αναφέραμε προθύστερα και κατά συνέπεια ένας κόμβος θα μπορούσε να πεθάνει πριν πεθάνουν όλα του τα παιδιά τα οποία περιμένουν να πάρουν σήμα SIGCONT για να σκοτώσουν τα παιδιά τους ή να πεθάνουν αν δεν έχουν. Συνεπώς κάποια μέρη του δένδρου θα υϊοθετηθούν από την init αφού θα γίνουν zombie.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Σε αυτήτ την άσκηση χρησιμοποιήσαμε pipes για την επικοινωνία μεταξύ των διεργασιών γονέων και παιδιών.

Ο κώδικας εργασίας μας είναι στο τμήμα κώδικα **Code.1.4**

Μετά την εκτέλεση του προγράμματός μας με το δοθέν δένδρο παίρνουμε την παρακάτω έξοδο:

```
oslabf04@lemnos:~/Exercises/Exercise2/Erwthma4$ ./ask4-signals expr.tree
Eimai fyllo: 10
Eimai fyllo: 4
Eimai fyllo: 5
Eimai fyllo: 7
This is my result: 5 + 7 = 12
This is my result: 4 * 12 = 48
This is my result: 10 + 48 = 58
This is the final result: 58
+
        10
        *
                +
                        5
                        7
                4
```

Ερωτήσεις:

1. Πόσες σωληνώσεις χρειάζονται στη συγκεκριμένη άσκηση ανά διεργασία; Θα μπορούσε κάθε γονική διεργασία να χρησιμοποιεί μόνο μία σωλήνωση για όλες τις διεργασίες παιδιά; Γενικά, μπορεί για κάθε αριθμητικό τελεστή να χρησιμοποιηθεί μόνο μια σωλήνωση;

Στην συγκεκριμένη άσκηση όπως ζητείται αλλά και στην υλοποιήσή μας χρησιμοποιήσαμε μόνο μια σωλήνωση από πατέρα σε παιδιά. Δηλαδή τα 2 παιδιά μπορούν να γράψουν στο ίδιο pipe για να στείλουν τα αποτελέσματά τους στον κοινό τους πατέρα. Αυτό οφείλεται στην αντιμεταθετικότητα των πράξεων του πολλαπλασιασμού και της πρόσθεσης που μας επιτρέπει να το υλοποιούμε κατά αυτόν τον τρόπο. Προφανώς αν είχαμε άλλους τελεστές όπως (-,/,^) θα μας ενδιέφερε η σειρά των τελεστέων και θα χρειαζόμασταν απαραίτητα 2 διαφορετικά pipes για να γράψουν τα 2 παιδιά στον πατέρα τα αποτελέσματά τους και αυτός να τα υπολογίσει με την σωστή σειρά. Εξαίρεση αποτελεί το γεγονός αν χρησιμοποιήσουμε μόνο ένα pipe αλλά μαζί με το αποτέλεσμα να στέλνουμε ένα διαχωριστικό που υποδηλώνει την προέλευση του αποτελέσματος. Για παράδειγμα θα μπορούσαμε να έχουμε τελεστές (-,/,^) αλλά μαζί με το αποτέλεσμα να στέλναμε και το pid του εκάστοτε παιδιού για να γνωρίζει ο πατέρας ποιά είναι η ορθή σειρά των τελεστέων.

2. Σε ένα σύστημα πολλαπλών επεξεργαστών, μπορούν να εκτελούνται παραπάνω από μια διεργασίες παράλληλα. Σε ένα τέτοιο σύστημα, τι πλεονέκτημα μπορεί να έχει η αποτίμηση της έκφρασης από δέντρο διεργασιών, έναντι της αποτίμησης από μία μόνο διεργασία;

Σε συστήματα πολλαπλών επεξεργαστών γνωρίζουμε ότι κάθε επεξεργαστής μπορεί να εκτελέσει κάθε στιγμή μόνο μία διεργασία κατά συνέπεια σπάζοντας τον υπολογισμό σε επιμέρους διεργασίες ο συνολικός υπολογισμός μπορεί να γίνει παράλληλα χωρίς να περιμένει κάποιος επεξεργαστής τον άλλον για να τελειώσει την δική του διεργασία. Αν αποτιμούσαμε όλη την έκφραση με μία μόνο διεργασία θα απαγορεύαμε την καλύτερη αξιοποίηση του hardware διότι όλη αυτή η διεργασία θα εκτελούταν σε έναν μόνο επεξεργαστή, αγνοώντας όλους τους άλλους διαθέσιμους πόρους.

3 Προαιρετικές ερωτήσεις

1. Έστω σύστημα με N επεξεργαστές. Ποιοι παράγοντες καθορίζουν αν ο παράλληλος υπολογισμός μιας αριθμητικής έκφρασης όπως στην άσκηση θα είναι ταχύτερος από τον αντίστοιχο υπολογισμό σε μία μόνο διεργασία (σειριακή εκτέλεση).

Η ταχύτητα του παράλληλου υπολογισμού μίας έκφρασης καθορίζεται από τα εξής:

1. Αν το N=1 τότε προφανώς δεν έχει διαφορά αφού κάθε φορά ο επεξεργαστής εκτελέι μία μόνο διεργασία και ουσιαστικά είναι σαν να έχουμε σειριακή εκτέλεση.
2. Η μορφή του δένδρου μπορεί να μας απαγορέψει να χρησιμοποιήσουμε ευέλικτα τον παράλληλο προγραμματισμό αφού για παράδειγμα αν το δέντρο έχει την μορφή μιάς λίστας τότε θα εξαναγκάσει τους επεξεργαστές να δουλέψουν σαν να ήταν σειριακή εκτέλεση.

Τμήμα κώδικα:

Code.1.1

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */
void fork_procs(void)
{
        pid_t pid;
        int status;
        /*
         * initial process is A.
         */

        change_pname("A");
        printf("A: Sleeping...\n");

        pid = fork();
        if(pid < 0) {
                perror("A: fork");
                exit(1);
        }
        if (pid == 0) {
                /*Child B*/
                change_pname("B");
                printf("B: Sleeping...\n");

                pid = fork();
                if(pid < 0) {
                        perror("B: fork");
                        exit(1);
                }
                if (pid == 0) {
                        /*Child D*/
                        change_pname("D");
                        printf("D: Sleeping...\n");

                        sleep(SLEEP_PROC_SEC);
                        printf("D: Exiting...\n");
                        exit(13);
                }

                /*Waiting for them kids */
                sleep(SLEEP_PROC_SEC);

                pid = wait(&status);
                explain_wait_status(pid, status);

                printf("B: Exiting...\n");
                exit(19);
        }

        pid = fork();
        if(pid < 0) {
                perror("C: fork");
```

```
                exit(1);
        }
        if (pid == 0) {
                /*Child C*/
                change_pname("C");
                printf("C: Sleeping...\n");

                /*Waiting for them kids */
                sleep(SLEEP_PROC_SEC);
                printf("C: Exiting...\n");
                exit(17);
        }
         //we kill procedure A
       //from ssh from another window kill -KILL getpid();
//**         kill(getpid(), SIGKILL);
//**       exit(1);         /* ... */
        sleep(SLEEP_PROC_SEC);

        /*Ena gia kathe paidi*/

        pid = wait(&status);
        explain_wait_status(pid,status);

        pid = wait(&status);
        explain_wait_status(pid, status);

        printf("A: Exiting...\n");
        exit(16);
}

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */
int main(void)
{
        pid_t pid;
        int status;

        /* Fork root of process tree */
        pid = fork();
        if (pid < 0) {
                perror("main: fork");
                exit(1);
        }
        if (pid == 0) {
                /* Child */
                fork_procs();
                exit(1);
        }

        /*
         * Father
         */
```

```
        /* for ask2-signals */
        /* wait_for_ready_children(1); */

        /* for ask2-{fork, tree} */
        sleep(SLEEP_TREE_SEC);
        //init procedure tree
//**             show_pstree(1);
        /* Print the process tree root at pid */
        show_pstree(pid);

        /* for ask2-signals */
        /* kill(pid, SIGCONT); */

        /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

Code.1.2

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"
#include "tree.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3



void
create_process_tree(struct tree_node *root)
{

        pid_t pid;
        int status;
        int i;

        change_pname(root->name);
        printf("%s: Sleeping...\n", root->name);

        for (i=0; i < root->nr_children; i++){

                /* Fork node of process tree */
                pid = fork();
                if (pid < 0) {
                        perror( root->name);
                        exit(1);
                }
                if (pid == 0) {
                        /* Child */
                        create_process_tree(root->children + i);
                        exit(1);
                }
        }
```

```
	sleep(SLEEP_PROC_SEC);

	for (i=0; i< root->nr_children; i++){
		/* Wait for the children of the node to terminate */
		pid = wait(&status);
		explain_wait_status(pid, status);

	}

	printf("%s: Exiting...\n", root->name);
	exit(0);

}


/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */
int main(int argc, char *argv[])
{

	struct tree_node *root;
	pid_t pid;
	int status;
	if (argc != 2) {
		fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
		exit(1);
	}

	root = get_tree_from_file(argv[1]);

	pid = fork();
	if (pid < 0) {
		perror("main: fork");
		exit(1);
	}
	if (pid == 0) {
		/* Child */
		create_process_tree(root);
		exit(1);
	}

	/*
	 * Father
	 */
	/* for ask2-signals */
	/* wait_for_ready_children(1); */

	/* for ask2-{fork, tree} */
	sleep(SLEEP_TREE_SEC);

	/* Print the process tree root at pid */
```

```
        show_pstree(pid);

        /* for ask2-signals */
        /* kill(pid, SIGCONT); */
 /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);


        print_tree(root);

        return 0;

}
```

Code.1.3

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"
#include "tree.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*void sig_handler(int signo)
{
  if (signo == SIGSTOP)
    printf("o apo katw mou stamathse\n");
}*/

void
fork_process_tree(struct tree_node *root)
{

        pid_t pid;
        int status;
        int i;
        //we have to save the pid's of the childs in order to come back to them
        pid_t *chil_pids=(pid_t *)malloc((root->nr_children)*sizeof(pid_t));

        printf("PID = %ld, name %s, starting...\n", (long)getpid(), root->name);
        change_pname(root->name);

        for (i=0; i < root->nr_children; i++){

                /* Fork node of process tree */
                pid = fork();
                if (pid < 0) {
                        perror( root->name);
                        exit(1);
                }

                chil_pids[i]=pid;
        }
```

```
	//When we are sure about all of our children that they have used raise()
	wait_for_ready_children(root->nr_children);

	//signal(SIGSTOP,sig_handler);
	//Suspend self, send signal in parent process to let it know
	raise(SIGSTOP);

	//when it comes back
	printf("PID = %ld, name = %s is awake\n",(long)getpid(), root->name);

	/*Kill all the children of the node to terminate */
	for (i=0; i < root->nr_children; i++){
		kill(chil_pids[i], SIGCONT);
		pid = wait(&status);
		explain_wait_status(pid, status);
	}

	printf("%s: Adios Amigos...\n", root->name);
	exit(0);
}


/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
*      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */
int main(int argc, char *argv[])
{

	struct tree_node *root;
	pid_t pid;
	int status;

	if (argc != 2) {
		fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
		exit(1);
	}

	root = get_tree_from_file(argv[1]);

	pid = fork();
	if (pid < 0) {
		perror("main: fork");
		exit(1);
	}
	if (pid == 0) {
		/* Child */
		fork_process_tree(root);
		exit(1);
	}

	/*
	 * Father
	 */
```

```
        /* for ask2-signals */
        wait_for_ready_children(1);

        /* for ask2-{fork, tree} */
        //sleep(SLEEP_TREE_SEC);

        /* Print the process tree root at pid */
        show_pstree(pid);
 /* for ask2-signals */
        kill(pid, SIGCONT);

        /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);


        print_tree(root);

        return 0;

}
```

Code.1.4

```
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"
#include "tree.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*void sig_handler(int signo)
{
  if (signo == SIGSTOP)
    printf("o apo katw mou stamathse\n");
}*/

void
fork_process_tree(struct tree_node *root, int dadspipe)
{

          pid_t pid;
        int status;
          int i, result, temp1, temp2;
          int childrenspipe[2];
          //we have to save the pid's of the childs in order to come back to them
          pid_t *chil_pids=(pid_t *)malloc((root->nr_children)*sizeof(pid_t));
        change_pname(root->name);

          //if the process has a +, * then it creates a pipe for its children to //write inside
          if(root->nr_children > 0)
          {
                  if (pipe (childrenspipe))
                  {
                          perror ("Pipe failed.\n");
```

```
                        exit(1);
                }
        }
        for (i=0; i < root->nr_children; i++){

                /* Fork node of process tree */
        pid = fork();
        if (pid < 0) {
                perror( root->name);
                exit(1);
        }
        if (pid == 0) {
                    /* Child */
                        close (childrenspipe[0]);
//We close the writing side of the      childrens pipe for the children
                fork_process_tree(root->children + i, childrenspipe[1]);
                        exit(1);
        }

                chil_pids[i]=pid;
        }
        close(childrenspipe[1]);

        //When we are sure about all of our children that they have used raise()
      //wait_for_ready_children(root->nr_children);


        /*Kill all the children of the node to terminate */
        for (i=0; i < root->nr_children; i++){
                pid = wait(&status);
        //explain_wait_status(pid, status);
        }

        //


        //if the currest process has data number it just writes on its pipe end

//else if its a +, * it reads the pipe that it has created and writes on its //fathers pipe, and then
closes the children pipe
        if(root->nr_children == 0)
        {
                result =(int)atoi(root->name);
                printf("Eimai fyllo: %d\n", result);
        }
        else
        {
                if( read(childrenspipe[0],&temp1, sizeof(temp1) ) < 0){perror("Read Pipe");
exit(1); }
                if( read(childrenspipe[0],&temp2, sizeof(temp2) ) < 0){perror("Read Pipe");
exit(1); }
        if(root->name[0] == '+')
                {
                        result = temp1 + temp2;
                }
                else
                {
                        result = temp1 * temp2;
                }
                printf("This is my result: %d %c %d = %d\n", temp1, root->name[0], temp2,
result);
```

```
        }

        if( write(dadspipe, &result, sizeof(result)) < 0){
                perror("Dads Pipe is not good");exit(1);
        }
    exit(0);
}


/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */
int main(int argc, char *argv[])
{

        struct tree_node *root;
        pid_t pid;
        int status;
        int mainpipe[2], result;

    if (argc != 2) {
            fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
            exit(1);
    }

    root = get_tree_from_file(argv[1]);

        if (pipe (mainpipe))
    {
            perror ("MainPipe failed.\n");
            exit(1);
    }


        pid = fork();
    if (pid < 0) {
            perror("main: fork");
            exit(1);
    }
    if (pid == 0) {
            /* Child */
                close(mainpipe[0]);
            fork_process_tree(root, mainpipe[1]);
            exit(1);
    }
        close(mainpipe[1]);
    /*
     * Father
     */
    /* for ask2-signals */
    //wait_for_ready_children(1);

    /* for ask2-{fork, tree} */
```

```
	//sleep(SLEEP_TREE_SEC);

	/* Print the process tree root at pid */
	/* for ask2-signals */
	//kill(pid, SIGCONT);

	/* Wait for the root of the process tree to terminate */
	pid = wait(&status);
	//explain_wait_status(pid, status);
	   if(read(mainpipe[0], &result, sizeof(result)) < 0){
	            perror("Main Pipe read not Good"); exit(1);
	   }
	   printf("This is the final result: %d\n", result);
	print_tree(root);

	return 0;

}
```