ΕΘΝΙΚΟ ΜΕΤΣΟΒΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ: ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΜΑΘΗΜΑ: Λειτουργικά Συστήματα

2η Εργαστηριακή Άσκηση

ΟΝΟΜΑΤΕΠΩΝΥΜΟ ΜΕΛΩΝ :

ΣΤΑΜΑΤΗΣ ΑΛΕΞΑΝΔΡΟΠΟΥΛΟΣ (03117060)

ΠΟΛΥΤΙΜΗ-ΑΝΝΑ ΓΚΟΤΣΗ (03117201)

ΑΘΗΝΑ

2020

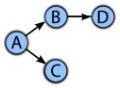
Στόχος της δεύτερης εργαστηριακής άσκησης ήταν η ενασχόληση με τη διαχείριση εργασιών καθώς και με την μεταξύ τους επικοινωνία. Η άσκηση αποτελείται από δύο μέρη. Στο πρώτο μέρος, που περιλαμβάνει τις ασκήσεις 1,2 υλοποιήσαμε πρόγραμμα κατασκευής δεδομένου δέντρου διεργασιών (Άσκηση 1) επεκτείνοντάς το για οποιοδήποτε δέντρο, βάσει αρχείου εισόδου (Άσκηση 2). Στο δεύτερο μέρος ενεργοποιήσαμε με συγκεκριμένη σειρά τις διεργασίες, χρησιμοποιώντας σήματα (Άσκηση 3) και εκτελέσαμε παράλληλο υπολογισμό μιας αριθμητικής έκφρασης, χρησιμοποιώντας σωληνώσεις για τη διάδοση των ενδιάμεσων αποτελεσμάτων (Άσκηση 4). Τονίζεται ότι για την υλοποίηση όλων των ερωτημάτων και προκειμένου να μην αλλάζουμε κάθε φορά το Makefile σε κάθε ερώτημα, κατασκευάσαμε για τις ασκήσεις 1,2,3,4 τα makefile.one, makefile.two, makefile.three, makefile.four αντίστοιχα και όλα αυτά τα συμπεριλάβαμε σε ένα Makefile το οποίο είναι το εξής :

```
one:
	make -f makefile.one

two:
	make -f makefile.two
three:
	make -f makefile.three
four:
	make -f makefile.four
```

Έτσι ανάλογα με την άσκηση που υλοποιούμε, εκτελούμε την κατάλληλη εντολή make one ή make two ή make three ή make four.

Ερώτημα 1.1

Στην άσκηση αυτή δημιουργήσαμε ένα δένδρο διεργασιών το οποίο έχει την παρακάτω μορφή



Συγκεκριμένα, οι διεργασίες - φυλλά εκτελούν τη κλήση sleep(), ενώ αυτές που αφορούν ενδιάμεσους κόμβους αναμένουν τον τερματισμό των διεργασιών-παιδιών τους. Όταν μια διεργασία αλλάζει από τη κατάσταση στην οποία βρίσκεται, σε μία διαφορετική εκτυπώνει το κατάλληλο μήνυμα : εκκίνηση, αναμονή για τερματισμό παιδιών, τερματισμό, με σκόπο την ορθή λειτουργία του προγράμματος.

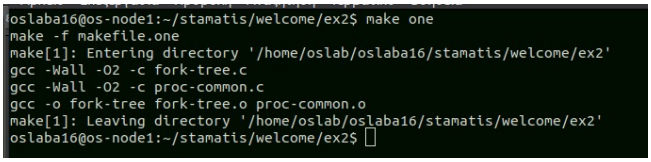
Τονίζεται ότι κάθε φορά που μία διεργασία τερματίζεται, εμφανίζει διαφορετικό κωδικό επιστροφής και συγκεκριμένα:

- Κωδικός επιστροφής της A= 16
- Κωδικός επιστροφής της B= 19
- Κωδικός επιστροφής της C= 17
- Κωδικός επιστροφής της D= 13

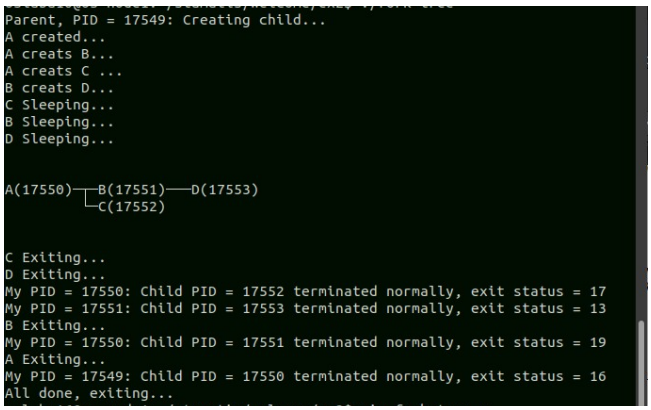
Για την υλοποίηση του ερωτήματος αυτό χρησιμοποιήσαμε το βοηθητικά αρχεία proc-common.{h,c} και βασιστήκαμε στη μορφή του κώδικα που περιγράφεται από το ask2-fork. Τονίζεται ότι πρώτα αντιγράψαμε τα αρχεία αυτά από το directory *home/*oslab/code/forktree στο directory *home/oslab/oslaba16/exc2*. Ο κώδικας περιέχεται στο αρχείο fork-tree.c . Προκειμένου να δημιουργηθεί το εκτελέσιμο αρχείο fork-tree και τα κατάλληλα object files δημιουργήσαμε το εξής Makefile:

```
#a simple Makefile

CC= gcc
CFLAGS= -Wall -O2

all:fork-tree

fork-tree: fork-tree.o proc-common.o
	$(CC) -o fork-tree fork-tree.o proc-common.o
fork-tree.o: fork-tree.c
	$(CC) $(CFLAGS) -c fork-tree.c
proc-common.o: proc-common.c
	$(CC) $(CFLAGS) -c proc-common.c

clean:
	rm -f fork-tree fork-tree.o proc-common.o
~
```

Το makefile αυτό το ονομάσαμε makefile.one (για τους λόγους που αναφέρονται παραπάνω) και το τρέχουμε πληκτρολογώντας στο command line την εντολή make one. Έτσι εμφανίζεται το παρακάτω αποτέλεσμα:



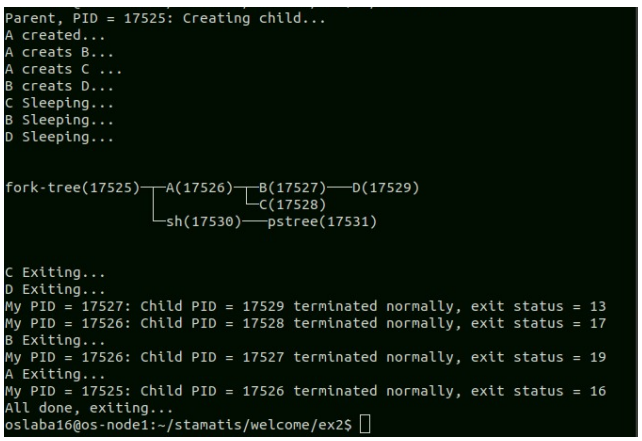
Τρέχοντας το εκτελέσιμο αρχείο με την εντολή ./fork-tree και εμφανίζεται το παρακάτω αποτέλεσμα:



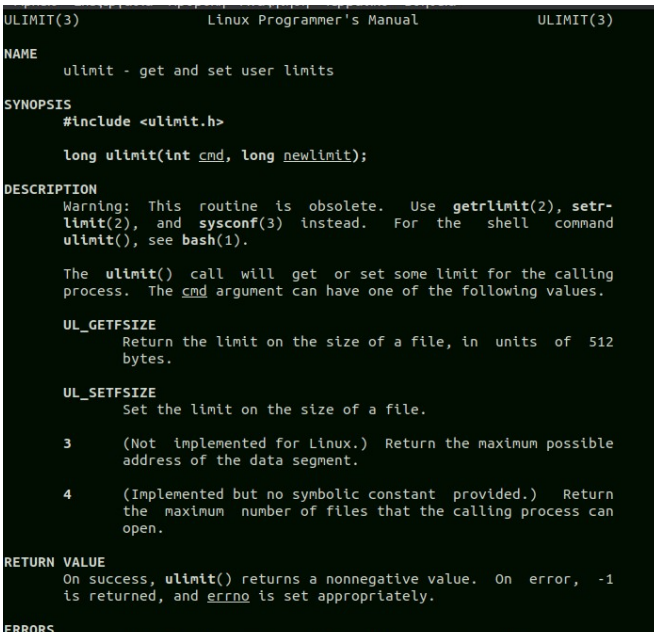
Ερωτήσεις:

1. Δίνοντας στον φλοιό την εντολή τερματισμού της διαεργασίας A kill –KILL <pid> ή και τερματίζοντας την διεργασία Α μέσα στο πρόγραμμα με κάποιο σήμα SIGKILL, η διεργασία θα λάβει το σήμα και θα τερματίσει την λειτουργία της, χωρίς να περιμένει τον τερματισμό των παιδιών της. Έτσι όλες οι διεργασίες παιδιά αυτής, όταν τερματίζουν δεν θα υπάρχει η διεργασία πατέρας τους ώστε να στείλουν σε αυτή το κατάλληλο μήνυμα τερματισμού (το οποίο λαμβάνει η διεργασία πατέρας κάνοντας wait()). Το αποτέλεσμα είναι να εξακολουθούν να υπάρχουν στον κατάλογο διεργασιών, αφού δεν έχει γίνει wait σε αυτές. Οι ονομαζόμενες αυτές διεργασίες zombie (διεργασίες που έχουν τερματίσει και δεν έχει γίνει wait σε αυτές), εφόσον έχουμε τερματίσει την διεργασία πατέρα τους, θα πρέπει με κάποιο τρόπο να αφαιρεθούν από τον κατάλογο των διεργασίων. Η πολιτική που ακολουθείται στα unix λειτουργικά συστήματα σε τέτοιες περιπτώσεις, είναι τα παιδιά zombies να γίνουν παιδιά μιας διεργασίας με pid =1 (init), η οποία κάνει διάρκως wait().

2. Αν γράψουμε την εντολή show_pstree(getpid()) αντί για την εντολή show_pstree(pid) θα εμφανιστεί το δέντρο μαζι με την διεργασία του προγράμματός μας (fork-tree), η οποία κάνει fork μια διεργασία shell η οποία μαζι με τη σειρά της καλεί την pstree που εκτυπώνει το ζητούμενο δένδρο, όπως φαίνεται στο παρακάτω σχήμα:



3. Σε υπολογιστικά συστήματα πολλαπλών χρηστών, πολλές φορές ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης. Αυτό είναι λογικό, αφού αν ο διαχειριστής δεν έχει βάλει όριο στον αριθμό διεργασιών που μπορεί να δημιουργήσει ένας χρήστης τότε είναι πιθανό να εξαντληθούν οι πόροι του μηχανήματος λόγω κακής χρήσης, καθιστώντας το αχρησιμοποίητο από τους υπόλοιπους χρήστες. Προκειμένου ο διαχειριστής να θέσει όριο στον αριθμό των διεργασιών χρησιμοποιείται η εντολή ulimit από το manual της οποίας βλέπουμε ότι :



```
ULIMIT(3)              Linux Programmer's Manual             ULIMIT(3)

NAME
       ulimit - get and set user limits

SYNOPSIS
       #include <ulimit.h>

       long ulimit(int cmd, long newlimit);

DESCRIPTION
       Warning:  This  routine  is  obsolete.   Use  getrlimit(2), setr-
       limit(2),  and  sysconf(3)  instead.   For  the   shell   command
       ulimit(), see bash(1).

       The  ulimit()  call  will  get  or set some limit for the calling
       process.  The cmd argument can have one of the following values.

       UL_GETFSIZE
              Return the limit on the size of a file, in  units  of  512
              bytes.

       UL_SETFSIZE
              Set the limit on the size of a file.

       3      (Not  implemented for Linux.)  Return the maximum possible
              address of the data segment.

       4      (Implemented but no symbolic constant  provided.)   Return
              the  maximum  number of files that the calling process can
              open.

RETURN VALUE
       On success, ulimit() returns a nonnegative value.  On  error,  -1
       is returned, and errno is set appropriately.

ERRORS
```

Τονίζεται ότι τα όρια που θέσει ο διαχειριστής πρέπει να είναι κατάλληλα ορισμένα, αφού σε περίπτωση λανθασμένης ρύθμισης, τα όρια αυτά μπορεί να προκαλέσουν απροσδόκητες ή ανεπιθύμητες συμπεριφορές, επηρεάζοντας πολλές φορές αρνητικά την απόδοση.

Ερώτημα 1.2

Στην άσκηση αυτή στόχος ήταν η κατασκευή προγράμματος που θα δημιουργεί αυθαίρετα δένδρα διεργασιών βάσει ενός αρχείου εισόδου. Το πρόγραμμα που υλοποιήσαμε βασίζεται στην αναδρομική κλήση της συνάρτησης void rec(struct tree_node *ptr) η οποία καλείται για κάθε κόμβο του δένδρου. Με βάση την υλοποίησή της και σύμφωνα με την εκφώνηση, αν ο κόμβος του δέντρου έχει παιδιά, η συνάρτηση θα δημιουργεί τα παιδιά και περιμένει να τερματιστούν. Αν δεν έχει, η συνάρτηση καλεί τη sleep() με όρισμα το LEAF_SLEEP , το οποίο έχει οριστεί στην αρχή του κώδικα ίσο με 10, με την εντολή *#define LEAF_SLEEP 10* . Το αρχείο εισόδου με το οποίο καλείται το εκτελέσιμο μας πρόγραμμα μέσω της εντολής ./ask2_2_tree bad.tree ακολουθεί τη δομή που περιγράφεται στην εκφώνηση.

Για την υλοποίηση του ερωτήματος αυτό χρησιμοποιήσαμε το βοηθητικά αρχεία proc-common.{h,c}, tree.{c,h} και βασιστήκαμε στη μορφή του κώδικα που περιγράφεται από το tree-example.c.O. Προκειμένου να δημιουργηθεί το εκτελέσιμο αρχείο fork-tree και τα κατάλληλα object files δημιουργήσαμε το εξής Makefile:

```
CC = gcc
CFLAGS = Wall

all: ask2_2_tree.o proc-common.o tree.o
	$(CC) -$(CFLAGS) ask2_2_tree.o proc-common.o tree.o -o ask2_2_tree

tree.o: tree.c
	$(CC) -$(CFLAGS) -c tree.c

proc-common.o: proc-common.c
	$(CC) -$(CFLAGS) -c proc-common.c

ask2_2_tree.o: ask2_2_tree.c
	$(CC) -$(CFLAGS) -c ask2_2_tree.c

clean:
	rm ask2_2_tree.o proc-common.o tree.o ask2_2_tree
~
```

Το makefile αυτό το ονομάσαμε makefile.two (για τους λόγους που αναφέρονται παραπάνω) και το τρέχουμε πληκτρολογώντας στο command line την εντολή make two. Έτσι εμφανίζεται το παρακάτω αποτέλεσμα:

```
oslaba16@os-node1:~/stamatis/welcome/ex2$ make two
make -f makefile.two
make[1]: Entering directory '/home/oslab/oslaba16/stamatis/welcome/ex2'
gcc -Wall -c ask2_2_tree.c
gcc -Wall -c proc-common.c
gcc -Wall -c tree.c
gcc -Wall ask2_2_tree.o proc-common.o tree.o -o ask2_2_tree
make[1]: Leaving directory '/home/oslab/oslaba16/stamatis/welcome/ex2'
```

Τρέχοντας το εκτελέσιμο αρχείο με την εντολή ./ask2_2_tree bad.tree και εμφανίζεται το παρακάτω αποτέλεσμα:

```
oslaba16@os-node1:~/stamatis/welcome/ex2$ ./ask2_2_tree bad.tree
Creating the following process tree:
A
        B
                E
                F
        C
        D
A: Forks 3 children
Specifically A: is forking now child with B
A: Forks 2 children
Specifically A: is forking now child with C
B: Forks 2 children
Specifically B: is forking now child with E
A: Forks 1 children
Specifically A: is forking now child with D
B: Forks 1 children
Specifically B: is forking now child with F


A(17794)─┬─B(17795)─┬─E(17797)
         │          └─F(17799)
         ├─C(17796)
         └─D(17798)


My PID = 17794: Child PID = 17796 terminated normally, exit status = 0
My PID = 17795: Child PID = 17797 terminated normally, exit status = 0
My PID = 17794: Child PID = 17798 terminated normally, exit status = 0
My PID = 17795: Child PID = 17799 terminated normally, exit status = 0
My PID = 17794: Child PID = 17795 terminated normally, exit status = 0
My PID = 17793: Child PID = 17794 terminated normally, exit status = 0
```

Ερωτήσεις:

1. Όπως προκύπτει από την εκτέλεση του προγράμματος, τα μηνύματα έναρξης εμφανίζονται κατά βάθος bfs (breadth-first search).Αυτό συμβαίνει επειδή κάθε διεργασία δημιουργεί με ένα for τα παιδιά της. Από την άλλη, τα μηνύματα τερματισμού δεν παρουσιάζουν κάποιο μοτίβο, καθώς η σειρά εμφάνισης τους εξαρτάται από τη σειρά που θα δώσει ο χειριστής στις διεργασίες.

Ερώτημα 1.3

Ζητούμενο της άσκησης αποτελούσε η επέκταση του ερωτήματος 1.2, με την προσθήκη σημάτων για τον έλεγχο των διεργασιών, προκειμένου οι διεργασίες του δημιουργούμενου δέντρου διεργασιών να τυπώνουν τα μηνύματα έναρξης και τερματισμού κατά βάθος.

Αρχικά, το κύριο πρόγραμμα διαβάζει το ζητούμενο δέντρο από το δοσμένο στο όρισμα argv[1] αρχείο, και δημιουργεί την αναπαράστασή του στην μνήμη (get_tree_from_file(argv[1])), τυπώνει το ζητούμενο δέντρο (print_tree(root)), δημιουργεί με fork() την διεργασία-ρίζα το δέντρου και περιμένει αυτή να αναστείλει την λειτουργία της.

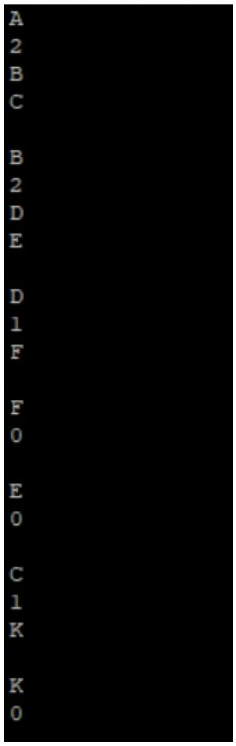
Έτσι, κάθε φορά που γεννιέται μια διεργασία ρίζας ή ενδιάμεσου κόμβου, δημιουργεί με την κλήση συστήματος fork() το πρώτο της παιδί, περιμένει αυτό να αναστείλει την λειτουργία του και στην συνέχεια δημιουργεί το επόμενο παιδί της (αν έχει επόμενο παιδί). Ο λόγος που ακολουθούνται οι ενέργειες με αυτή την σειρά είναι ότι αν η διεργασία πατέρας δημιουργούσε το δεύτερο παιδί της πριν βεβαιωθεί ότι το πρώτο έχει δημιουργήσει τα δικά του παιδιά (αν έχει) και έχει αναστείλει την λειτουργία του, τότε οι δύο διεργασίες παιδία της θα έτρεχαν παράλληλα και θα δημιουργούσαν τα δικά τους παιδία κοκ, κι έτσι δεν θα ήταν εξασφαλισμένη η έναρξη των διεργασιών των επομένων επιπέδων του δέντρου κατά DFS.

Κάθε φορά δε που γεννιέται μια διεργασία φύλλο του δέντρου, αυτή αναστέλλει απευθείας την λειτουργία της (δεν έχει παιδιά για να περιμένει).

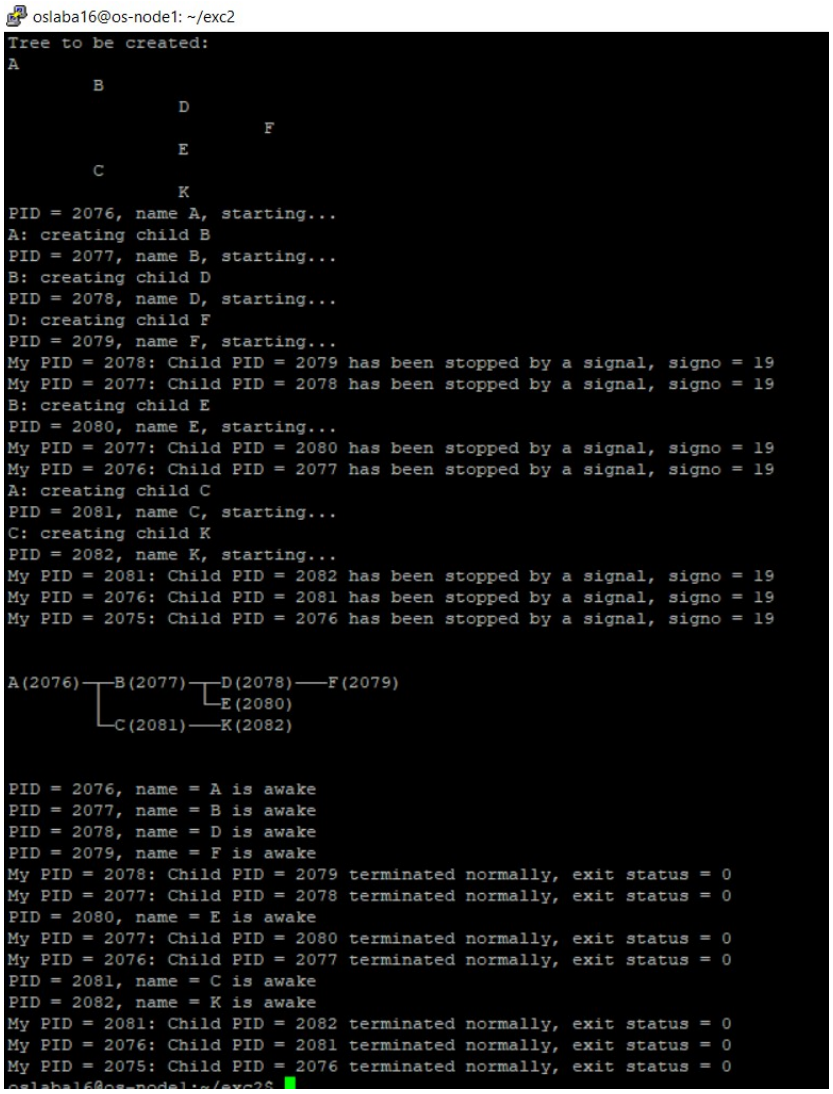
Έτσι, δημιουργείται κατά DFS σειρά το δέντρο διεργασιών, και όλες οι διαδικασίες αναστέλλουν η μία μετά την άλλη την λειτουργία τους. Όταν και η διεργασία της ρίζας του δέντρου αναστείλει την λειτουργία της, η αρχική διεργασία πατέρας του προγράμματος που εκτελεί wait_for_ready_children(1) λαμβάνει το κατάλληλο σήμα, και είναι πια έτοιμη να συνεχίσει, τυπώνοντας το δέντρο διεργασιών και στέλνοντας σήμα στην διεργασία –ρίζα να «ξυπνήσει». Αυτή ξυπνά, στέλνει μήνυμα στο πρώτο παιδί της να ξυπνήσει, περιμένει τον τερματισμό του και μαθαίνει για ποιο λόγο τερμάτισε, τυπώνει κατάλληλο μήνυμα και στην συνέχεια ξυπνά το επόμενο παιδί (αν έχει). Εφόσον για να τερματίσει το πρώτο παιδί της πρέπει να έχουν τερματίσει πρώτα τα δικά του παιδία (όταν ξυπνήσει το παιδί εκτελεί τα ίδια βήματα για τα δικά του παιδία, αυτά για τα δικά τους, κοκ), εξασφαλίζεται ο τερματισμός τον διεργασιών κατά βάθος.

Έτσι για αρχείο εισόδου: (“ input.txt “)

oslaba16@os-node1



προκύπτει το εξής output:



Παρατηρούμε ότι τα μηνύματα έναρξης, τα μηνύματα αναστολής λειτουργίας, τα μηνύματα επανέναρξης και τα μηνύματα λήξης τυπώνονται πράγματι κατά βάθος, καθώς χρησιμοποιούμε τα σήματα SIGSTOP, SIGCONT με τρόπο που να το εξασφαλίζει, μέσω της μεθόδου που περιγράφηκε παραπάνω. (Συνοπτικά: Δημουργούμε ένα δέυτερο παιδί μιας διεργσίας μόνο αφού το πρώτο έχει δημιουργήσει το υπόδεντρο του και έχει αναστείλει την λειτουργία του, και ξυπνάμε το δεύτερο παιδί μόνο αφού έχει ξυπνήσει κατά DFS όλο το υπόδεντρο το πρώτου παιδιού).

Ερωτήσεις:

1. Με την χρήση της sleep() , δεν μπορούμε να ελέγξουμε την πορεία και των συγχρονισμό των διεργασιών, αφού αυτές, μετά την δημιουργία τους λειτουργούν πλήρως αυτόματα. Έτσι, η σειρά με την οποία αυτές θα εκτελέσουν τις λειτουργία τους και θα τερματίσουν εξαρτάται από την σειρά που θα λάβουν από τον χειριστή. Αντίθετα, η χρήση των σημάτων παρέχει την δυνατότητα επικοινωνίας μεταξύ

διεργασίων και επομένως και την δυνατότητα ελέγχου της ροής αυτών, με την παύση και έναρξη τους ή με τον τερματισμό τους από άλλες διεργασίες. Έτσι, μπορούμε με τα κατάλληλα σήματα να ελέγξουμε την σειρά εκτέλεσης των εργασιών (θέτοντας τες σε παύση), αντί να την αφήσουμε εξαρτώμενη μόνο από τον χειριστή. Γενικότερα, τα σήματα παρέχουν την δυνατότητα διαεργασιακής επικοινωνίας με αποστολή και λήψη μηνυμάτων προς μια άλλη διεργασία ή προς την ιδία δειργασία, που σχετίζονται με στοιχεία της λειτουργίας μιας διεργσίας, την πορεία ή τον λόγω τερματισμού της (π.χ. Το σήματα SIGSEGV δίνει το μήνυμα κάποιου segmentation fault).

2. Η συνάρτηση wait_for_ready_children() λαμβάνει ως παράμετρο των αριθμό των παιδιών του παρόντος κόμβου και εξασφαλίζει ότι ο παρόντας κόμβος θα περιμένει μέχρι να λάβει τόσα σήματα αναστολής λειτουργίας ενός παιδιού όσα και τα παιδιά του, μέχρι δηλαδή να βεβαιωθεί πως όλα τα παιδιά του έχουν αναστείλει την λειτουργία τους. Η παράλειψη της εντολής αυτής, θα μπορούσε να έχει τις εξής συνέπειες: Αρχικά, όταν μια διεργασία πρέπει να δημιουργήσει δύο παιδιά, αφού δημιουργήσει το πρώτο, να μην περιμένει αυτό να σταματήσει να λειτουργεί έχοντας δημιουργήσει το δικό του υπόδεντρο, αλλά να προχωρήσει κατευθείαν στην δημιουργία του δεύτερου παιδιού με αποτέλεσμα οι δύο διεργασίες να δημιουργούν παιδιά παράλληλα και όχι κατά βάθος. Έτσι, προκύπτει τελικά και μία δεύτερη συνέπεια: Μια διεργασία μπορεί να δημιουργήσει τα παιδιά της και να αναστείλει την λειτουργία της προτού βεβαιωθεί ότι όλα τα κατώτερα επίπεδα του δέντρου έχουν δημιουργηθεί. Έτσι, όταν κληθεί η συνάρτηση show_ptree() στο κύριο πρόγραμμα για την εμφάνισει του δέντρου δειργασιών, μπορεί να έχει δημιουργηθεί από μία έως όλες οι διεργασίες, και δεν μπορούμε να είμαστε σίγουροι ποιο δέντρο διεργασιών θα τυπωθεί.

Ερώτημα 1.4

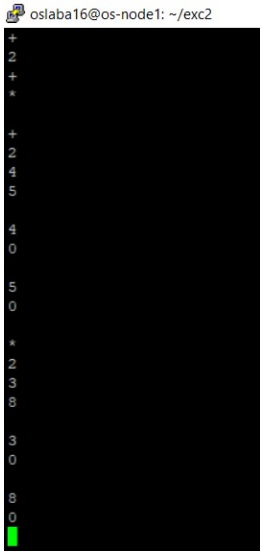
Στόχο το ερωτήματος αποτελούσε η επέκταση του ερωτήματος 1.2 για την κατασκευή ενός δέντρου διεργασιών το οποίο υπολογίζει κάποια αριθμητική παράσταση κατά BFS. Προκειμένου να επιτευχθεί κάτι τέτοιο είναι απαραίτητο κάθε παιδί να στέλνει στην διεργασία πατέρα του κάποια αριθμητική τιμή. Συγκεκριμένα, οι διεργασίες φύλλα στέλνουν την τιμή του αριθμού που θα αποτελέσει το ένα όρισμα της πράξης που αυτός θα εκτελέσει, και ο πατέρας λαμβάνοντας τα δύο ορίσματα (κάθε ενδιάμεσος κόμβος έχει δύο παιδιά) εκτελεί την αριθμητική πράξη που ορίζει το όνομα του (root->name) και στέλνει το αποτέλεσμα στην δική του διεργασία πατέρα, ώστε να χρησιμοποιοηθεί από αυτόν σε επόμενο υπολογισμό. Η διαδικασία ακολουθείται με τον τρόπο αυτό από τα φύλλα μέχρι τη ρίζα, η οποία και υπολογίζει το τελικό αποτέλεσμα της αριθμητικής παράστασης.

Για να επιτευχθεί η επικοινωνία αυτή των διεργασιών, χρησιμοποιούνται σωληνώσεις του UNIX (pipes). Κάθε σωλήνωση διαθέτει δύο άκρα, στα οποία στο ένα γράφει το παιδί τον κατάλληλο αριθμό, και στο άλλο διαβάζει τον αριθμό ο πατέρας, προκειμένου να εκτελέσει την αριθμητική πράξη. Έτσι, κάθε διεργασία που δεν είναι φύλλο περιμένει τον τερματισμό ενός παιδιού της, και όταν αυτό τερματίσει, διαβάζει από το άκρο ανάγνωσης της σωλήνωσης τον αριθμό που αυτό

της έστειλε. Εκτελεί ύστερα την ίδια διαδικασία για το δεύτερο παιδί της και αφού έχει λάβει τους δύο αριθμούς, εκτελεί την κατάλληλη πράξη και γράφει στο δικό της άκρο εγγραφής για την επικοινωνία με την διεργασία πατέρα της και ύστερα τερματίζει. Την ίδια διαδικασία εκτελεί ο πατέρας αυτής κοκ.

Παράλληλα, χρησιμοποιούμε τα σήματα SIGSTOP, SIGCONT, προκειμένου να εξασφαλίσουμε, όπως στο προηγούμενο ερώτημα, ότι το δέντρο διεργασιών θα τυπωθεί σωστά από την show_ptree(). Ωστόσο, εδώ δεν μας ενδιαφέρει η δημιουργία του δέντρου κατά DFS ούτε και η επανεκκίνηση των διεργασιών κατά βάθος, κι επομένως κάθε διεργασία δημιουργεί τα παιδιά της διαδοχικά και αργότερα όταν αυτή επανεκκινήσει τα ξυπνάει το ένα μετά το άλλο κατευθείαν, προκειμένου να επιτρέχουμε στις διεργασίες παιδιά να τρέχουν παράλληλα, υπολογίζοντας τις αριθμητικές πράξεις των κατώτερων επιπέδων του δέντρου. Άλλωστε, αφού τερματίσουν τα παιδιά βρίσκονται σε κατάσταση zombie μέχρι να εκτελεστεί waitpid() και να ληφθεί από την διεργασία πατέρα τους το σήμα τερματισμού τους.

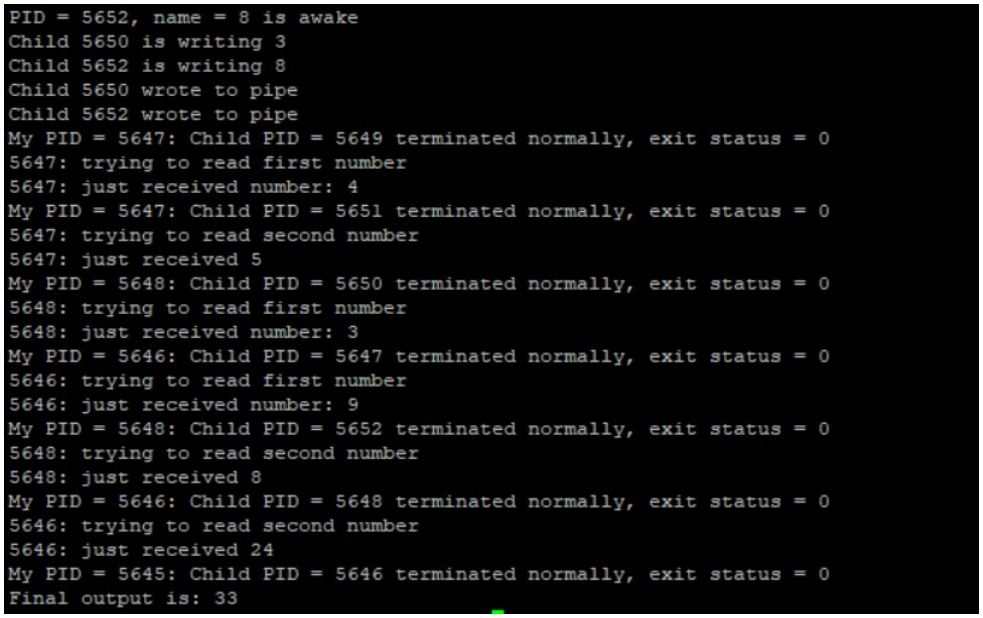
Έτσι, για το αρχείο εισόδου (“input3.txt”):



Προκύπτει:

oslaba16@os-node1: ~/exc2

```
oslaba16@os-node1:~/exc2$ vim input3.txt
oslaba16@os-node1:~/exc2$ ./ask2_4_tree input3.txt
+
        +
                4
                5
        *
                3
                8
PID = 5646, name +, starting...
+: creating child +
+: creating child *
PID = 5647, name +, starting...
+: creating child 4
PID = 5648, name *, starting...
*: creating child 3
+: creating child 5
PID = 5649, name 4, starting...
*: creating child 8
PID = 5650, name 3, starting...
My PID = 5647: Child PID = 5649 has been stopped by a signal, signo = 19
PID = 5651, name 5, starting...
My PID = 5648: Child PID = 5650 has been stopped by a signal, signo = 19
PID = 5652, name 8, starting...
My PID = 5647: Child PID = 5651 has been stopped by a signal, signo = 19
My PID = 5646: Child PID = 5647 has been stopped by a signal, signo = 19
My PID = 5648: Child PID = 5652 has been stopped by a signal, signo = 19
My PID = 5646: Child PID = 5648 has been stopped by a signal, signo = 19
My PID = 5645: Child PID = 5646 has been stopped by a signal, signo = 19


+(5646)─┬─*(5648)─┬─3(5650)
        │         └─8(5652)
        └─+(5647)─┬─4(5649)
                  └─5(5651)


PID = 5646, name = + is awake
PID = 5647, name = + is awake
PID = 5649, name = 4 is awake
PID = 5651, name = 5 is awake
Child 5649 is writing 4
Child 5651 is writing 5
PID = 5648, name = * is awake
Child 5649 wrote to pipe
Child 5651 wrote to pipe
PID = 5650, name = 3 is awake
PID = 5652, name = 8 is awake
Child 5650 is writing 3
Child 5652 is writing 8
```



Παρατηρούμε πως πράγματι οι διεργασίες δύο διαφορετικών υποδέντρων εκτελούνται παράλληλα.

Ερωτήσεις:

1.Κάθε ενδιάμεση διεργασία χειρίζεται δύο σωληνώσεις. Μία αυτή που της επιτρέπει την επικοινωνία με την διεργασία πατέρα, για την οποία διαθέτει το άκρο εγγραφής, και μία για την επικοινωνία με τα παιδία της, για την οποία διαθέτει το άκρο ανάγνωσης. Έτσι, η διεργασία χρησιμοποιεί την ίδια σωλήνωση για τα δύο παιδία της, και άρα για κάθε αριθμητικό τελεστή χρησιμοποιείται μόνο μία σωλήνωση, στο άκρο εγγραφής της οποίας γράφουν και τα δύο παιδιά, και τα αποτελέσματα διαβάζει η διεργασία πατέρας με δύο αναγνώσεις καθορισμένου μήκους, μία για κάθε παιδί.

2. Σε ένα σύστημα πολλών επεξεργαστών είναι εφικτό διαφορετικές διεργασίες να τρέχουν παράλληλα σε διαφορετικούς επεξεργαστές. Αυτό σημαίνει, ότι, δύο ή περισσότερα υπόδεντρα ενός δέντρου διεργασιών μπορούν να εκτελούν παράλληλα τους υπολογισμούς των επιπέδων κατωτέρων από αυτά με αποτέλεσμα τελικά πολύ μεγαλύτερη ταχύτητα αφού η εξαγωγή ενός αποτελέσματος ενός υποδέντρου εξαρτάται μόνο από τον παράλληλο και πάλι υπολιγμό του υποδέντρου του. Αντίθετα, όταν η αριθμητική έκφραση υπολογίζεται από μία μόνο διεργασία, ο υπολογισμός γίνεται σειριακά, και επομένως ο απαιτούμενος χρόνος είναι μεγαλύτερος, αφού κάθε πράξη εκτελείται μόνο αφού ολοκληρωθεί η προηγούμενή της.

Οι κώδικες για τα παραπάνω ερωτήματα είναι:

Κώδικας για το 1.1 (fork-tree.c)

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

#include <sys/wait.h>

#include "proc-common.h"

#define SLEEP_SEC 5
#define LEAF_SLEEP 10
void child(){
	//initial process is A
	pid_t B,C,D, p;
	int status;
	fprintf(stderr,"A creats B...\n");
	B=fork();
	if (B<0){
		perror("Error when in B");
		exit(1);
	}
	else if (B==0){
		fprintf(stderr, "B creats D...\n");
		D=fork();
		if(D<0){
			perror("Error when in D\n");
			exit(1);
		}
		else if(D==0){
			change_pname("D");
			fprintf(stderr, "D Sleeping...\n");
			sleep(LEAF_SLEEP);
			fprintf(stderr, "D Exiting...\n");
			exit(13);
		}
		else{
			change_pname("B");
			fprintf(stderr,"B Sleeping...\n");
			p=wait(&status);
			explain_wait_status(D,status);
			sleep(LEAF_SLEEP);
			printf("B Exiting...\n");
			exit(19);
		}
	}
 else{
		fprintf(stderr,"A creats C ...\n");
```

```
		C=fork();
		if(C<0){
			perror("When in C/n");
			exit(1);
		}
		else if(C==0){
			change_pname("C");
			fprintf(stderr, "C Sleeping...\n");
			sleep(LEAF_SLEEP);
			fprintf(stderr,"C Exiting...\n");
			exit(17);
		}
		else{
			change_pname("A");
			p=wait(&status);
			explain_wait_status(p,status);
			p=wait(&status);
			explain_wait_status(p,status);
			sleep(LEAF_SLEEP);
			printf("A Exiting...\n");
			exit(16);
		}
	}
}
int main(void)
{
	pid_t p;
	int status;

	fprintf(stderr, "Parent, PID = %ld: Creating child...\n",
			(long)getpid());
	p = fork();
	if (p < 0) {
		/* fork failed */
		perror("fork");
		exit(1);
	}
	if (p == 0) {
		printf("A created...\n");
		child();
		exit(1);
	}

	//	change_pname("father");
	/*
	 * In parent process. Wait for the child to terminate
	 * and report its termination status.
	 */
	sleep(SLEEP_SEC);
	show_pstree(p);
	p = wait(&status);
	explain_wait_status(p, status);
```

```
	printf("All done, exiting...\n");

	return 0;
}
```

Κώδικας για το 1.2 (ask2_2_tree.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <fcntl.h>
#include "tree.h"
#include "proc-common.h"

#define LEAF_SLEEP 10
#define SLEEP_SEC 5

void rec(struct tree_node *ptr){
	pid_t child;
	change_pname(ptr->name);
	int i;
	for(i=0;i<ptr->nr_children;i++){
		printf("%s: Forks %d children\n",ptr->name,ptr->nr_children-i);
		printf("Specifically %s: is forking now child with %s \n",ptr->name,ptr-
>children[i].name);
		child=fork();
		if (child<0){
			perror("fork");
			exit(1);
		}
		if(child==0){
		//	printf("Child with %ld created\n",getpid());
			rec(ptr->children+i);
			exit(1);
		}
	}

		for (i = 0; i < ptr->nr_children; ++i) {
		int status;
		pid_t pid;
		pid=wait(&status);
		explain_wait_status(pid, status);

	}
	if(ptr->nr_children==0) sleep(LEAF_SLEEP);
	exit(0);
```

```
}
int main(int argc,char *argv[]){
	struct tree_node *root;
	pid_t p;
	int status;

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

	root=get_tree_from_file(argv[1]);
	printf("Creating the following process tree: \n");
	print_tree(root);

	//Create the first child
	p=fork();
	if(p<0){
		perror("Error in the main\n");
		exit(1);
	}
	else if(p==0){
		rec(root);
		exit(1);
	}
	else {
		sleep(SLEEP_SEC);
		show_pstree(p);
		p=wait(&status);
		explain_wait_status(p,status);
		return 0;
	}
}
```

Κώδικας για το 1.3 (ask2-signals.c)

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

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

```
void fork_procs(struct tree_node *root)
{

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

	int i;
	pid_t pid[root->nr_children];
	for (i=0; i<root->nr_children; i++) {
		printf("%s: creating child %s\n", root->name, (root->children+i)->name);
		pid[i]=fork();
		if (pid[i]<0) { //error in fork
			perror(root->name);
			exit(1);
		}
		if (pid[i]==0) { //is child
			fork_procs(root->children+i);
			exit(1);
		}

	//is father:
	wait_for_ready_children(1); //wait for all childrem to suspend themselves
	}
	raise(SIGSTOP); //Suspend self until SIGCONT is received
	printf("PID = %ld, name = %s is awake\n",(long)getpid(), root->name); //print message after
proccess is resumed

	/* ... */
	for (i=0; i<root->nr_children; i++){
		kill(pid[i], SIGCONT); //send signal to child to wake up
		pid_t pid;
		int status;
		pid=wait(&status); //wait for child to exit
		explain_wait_status(pid, status);  //print reason for child's exit
		//repeat for next child
	}

	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[])
{
	pid_t pid;
	int status;
	struct tree_node *root;

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

	/* Read tree into memory */
	root = get_tree_from_file(argv[1]);
	printf("Tree to be created:\n");
	print_tree(root);
	/* Fork root of process tree */
	pid = fork();
	if (pid < 0) {
		perror("main: fork");
		exit(1);
	}
	if (pid == 0) {
		/* Child */
		fork_procs(root);
		exit(1);
	}

	//Father:
	wait_for_ready_children(1); //wait for root of tree to suspend itself

	//Print the process tree root at pid:
	show_pstree(pid);

	kill(pid, SIGCONT); //send signal to root of tree to resume

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

	return 0;
}
```

Κώδικας για το 1.4(ask2_4_tree.c)

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

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

void fork_procs(struct tree_node *root, int pip[])
{
	int pfd[2]; //create array to become pipe

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

	int i;
	if (pipe(pfd)<0) {  //create pipe
		perror("error creating pipe");
		exit(1);
	}
	pid_t pid[root->nr_children];
	for (i=0; i<root->nr_children; i++) { //create children (if not leaf)
		printf("%s: creating child %s\n", root->name, (root->children+i)->name);
		pid[i]=fork();
		if (pid[i]<0) { //error in fork()
			perror(root->name);
			exit(1);
		}
		if (pid[i]==0) {  //child
			fork_procs(root->children+i, pfd);
			exit(1);
		}
	}
	wait_for_ready_children(root->nr_children);

	/*
	 * Suspend Self
	 */
	raise(SIGSTOP);
	printf("PID = %ld, name = %s is awake\n",(long)getpid(), root->name);
 int first, second;
	int  result;
	if (root->nr_children==2){
		kill(pid[0], SIGCONT); //wake proccesses
		kill(pid[1], SIGCONT);
		pid_t p;
		int status;
		p=wait(&status); //wait for child to exit
		explain_wait_status(p, status);  //explain reason for child's exit
		//kill(pid[1], SIGCONT);
		printf("%d: trying to read first number\n", getpid());
		if (read(pfd[0], &first, sizeof(first))!=sizeof(first)) { //read first arguement from pipe
			perror("error read from pipe from first child");
			exit(1);
		}
		printf("%d: just received number: %d\n",getpid(), first);
		//kill(pid[1], SIGCONT);
```

```
            p=wait(&status); //wait for second child to exit
            explain_wait_status(p, status); //print reason for child's exit
            printf("%d: trying to read second number\n", getpid());
            if (read(pfd[0],&second, sizeof(second))!=sizeof(second)) {  //read second arguement
from pipe
                  perror("error reading from pipe from second child");
                  exit(1);
            }
            printf("%d: just received %d\n",getpid(),  second);
            if (*(root->name)=='+') //determine operator, calculate value
                  result=first+second;
            else if (*(root->name)=='*')
                  result=first*second;
            if (write(pip[1], &result, sizeof(result))!=sizeof(result)){ //send result to father
                  perror("error writing my result to pipe");
                  exit(1);
            }

      }
 //leaf:
      if (root->nr_children==0){
            printf("Child %d is writing %s\n",getpid(), root->name);
            int value=atoi(root->name);
            if (write(pip[1], &value, sizeof(value))!=sizeof(value)){
                  perror("write to pipe from proccess");
                  exit(1);
            }
            printf("Child %d wrote to pipe\n", getpid());
            exit(0);
      }
      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[])
{
      int pfd[2];
      pid_t pid;
      int status;
      int value;
      struct tree_node *root;
```

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

	/* Read tree into memory */
	root = get_tree_from_file(argv[1]);
	print_tree(root); //print tree to be created
	if (pipe(pfd)<0) { //create pipe
		perror("initial pipe");
		exit(1);
	}

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

	/*
	 * Father
	 */
	wait_for_ready_children(1); //wake for root of tree to suspend itself

	/* Print the process tree root at pid */
	show_pstree(pid);

	kill(pid, SIGCONT); //wake root proccess

	/* Wait for the root of the process tree to terminate */
	wait(&status);
	explain_wait_status(pid, status);
	if (read(pfd[0],&value,sizeof(value))!=sizeof(value)){
		perror("read from initial pipe");
		exit(1);
	}
	printf("Final output is: %d\n",value);

	return 0;
}
```