Λειτουργικά Συστήματα, 6ο Εξάμηνο, 2022-2023

Δημουλάς Ιωάννης, 03120083,
Ματζόρι Ενρίκα Ηλιάνα, 03120143
oslab20

Άσκηση 2η

Στο τέλος παρατίθεται το Makefile για τις 4 ασκήσεις.

Άσκηση 2.1

Παρατίθεται ο κώδικας της 1ης άσκησης ask2-fork.c :

```c
#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)
{
   /*
    * initial process is A.
    */
//η πρώτη διεργασία/παιδί που δημιουργείται με το fork την λέμε Α όπου για τις
//υπόλοιπες διεργασίες θα είναι ο πατέρας
//το pid του Α θα είναι το Pid που επιστρέφει στην πρώτη πρώτη διεργασία η πρώτη
//κλήση της fork στην main
//δλδ pid_A=our_pid πριν το our_pid γίνει 0.
   change_pname("A");

   /* ... */
   //Process_B
   pid_t pidB;
   int statusB;
   pidB = fork(); //όταν αυτή θα τρέχει απ'τον πατέρα,δλδ την Α, το pid_B είναι ίσο
//με το pid του Β
   printf("pid of B = %d\n", pidB);
   if (pidB < 0) {
       perror("B: fork");
       exit(1);
   }
   if (pidB == 0) {
       change_pname("B");
       //Process_D
//για να ορίσουμε την D να είναι παιδί της Β πρέπει ο πατέρας να είναι η Β, συνεπώς
//η pidB = fork();πρέπει να έχει επιστρέψει 0, δηλαδή να "καλείται" από την Β
       pid_t pidD;
       int statusD;
       pidD = fork();
       printf("pid of D = %d\n", pidD);
```

```
        if (pidD < 0) {
            perror("D: fork");
            exit(1);
        }
        if (pidD == 0) {
            change_pname("D");
            printf("D: Sleeping...\n");
            sleep(SLEEP_PROC_SEC);
            printf("D: Exiting...\n");
            exit(13);
        }
//όμοια με την A, εφόσον η B είναι πατέρας πρέπει να την κοιμίσουμε ώστε να
//δημιουργηθεί το παιδί της, η D
        printf("B: Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
      pidD = wait(&statusD); //κανω τον πατερα να περιμενει να πεθανουν ολα τα //παιδια
του(εδώ το B έχει ενα παιδί)
        explain_wait_status(pidD, statusD);
        printf("B: Exiting...\n");
        exit(19);
    }
    //Process_C
//στην πρώτη κλήση της fork για την B δεν μπαίνει στο if(pidB == 0) οπότε //καλούμε
//άλλη μία fork
    // για να αποκτήσει η A και δεύτερο παιδί, την C
    pid_t pidC;
    int statusC;
    pidC = fork();
    printf("pid of C = %d\n", pidC);
    if (pidC < 0) {
        perror("C: fork");
        exit(1);
    }
    if (pidC == 0) {
        change_pname("C");
        printf("C: Sleeping...\n");
        sleep(SLEEP_PROC_SEC);  //κοιμίζουμε την C προκειμένου να δημιουργηθεί το
//δέντρο και  να προλάβει να εκτυπωθεί
        printf("C: Exiting...\n");
        exit(17);
    }
    //πάλι στην πρώτη κλήση της fork(απ'την A) δεν μπάινουμε στο if οπότε
    //κάνουμε τον πατέρα να κοιμηθεί ώστε να κληθεί η fork από μία απ'τις
//διεργασίες-παιδιά
    // και να μπει στο αντίστοιχο if(pid==0)
    printf("A: Sleeping...\n");
    sleep(SLEEP_PROC_SEC);
    //η πρώτη διεργασία που θα ξυπνήσει θα είναι και η πρώτη που κοίμισα, δηλαδή η A
    //κανω τον πατερα να περιμενει να πεθανουν ολα τα παιδια του
    //με το που ξυπνήσει κάποια διεργασία "πάει" στην επόμενη εντολή του sleep
    // που είναι το printf("Exiting") και επιστρέφει ένα συγκεκριμένο exit status
    pidB = wait(&statusB);
    explain_wait_status(pidB, statusB);
    pidC = wait(&statusC);
    explain_wait_status(pidC, statusC);

    /* ... */

    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?
```

```c
* 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();
   printf("our pid= %d\n", pid);
   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} */

   //κοιμίζουμε τον πατέρα ώστε να δημιουργηθεί το δέντρο,
   // ώστε όταν ξυπνήσει να είναι έτοιμο για εκτύπωση
   sleep(SLEEP_TREE_SEC);

   /* Print the process tree root at pid */
   printf("father is awake!... \n");
   show_pstree(pid);

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

   /* Wait for the root of the process tree to terminate */
   printf("father is waiting \n");
   //κανω τον πατερα να περιμενει να πεθανουν ολα τα παιδια του
   //εδώ η αρχική διεργασία πατέρας έχει μόνο το Α σαν παιδί
   //οπότε το pid που θα μας επιστρέψει η wait θα είναι το pid του Α
   pid = wait(&status);
   explain_wait_status(pid, status); //θα μας βγαλει ότι πεθανε το Α

   return 0;
}
```

Παρακάτω παρουσιάζεται η έξοδος του προγράμματος:

```
oslab20@orion:~/ask2$ ./ask2-fork
our pid= 23002
our pid= 0
pid of B = 23003
pid of C = 23004
A: Sleeping...
pid of B = 0
pid of D = 23005
B: Sleeping...
pid of C = 0
C: Sleeping...
pid of D = 0
D: Sleeping...
father is awake!...


A(23002)─┬─B(23003)──D(23005)
         └─C(23004)


father is waiting
C: Exiting...
My PID = 23002: Child PID = 23004 terminated normally, exit status = 17
D: Exiting...
My PID = 23003: Child PID = 23005 terminated normally, exit status = 13
B: Exiting...
My PID = 23002: Child PID = 23003 terminated normally, exit status = 19
A: Exiting...
My PID = 23001: Child PID = 23002 terminated normally, exit status = 16
```

ΕΡΩΤΗΣΕΙΣ:

1) **Τι θα γίνει αν τερματίσετε πρόωρα τη διεργασία Α, δίνοντας kill -KILL , όπου το Process ID της;**

   Ανοίγοντας 2 παράθυρα στο terminal(σε ubuntu), στο ένα τρέχουμε την ask2-fork και ταυτόχρονα τρέχουμε την εντολή kill -KILL 1298 ,όπου 1298 το pid της διεργασίας Α. Αυτό που παρατηρούμε στην έξοδο φαίνεται παρακάτω:

```
oslab20@orion:~/ask2$ kill -KILL 1298
```

```
oslab20@orion:~/ask2$ ./ask2-fork
our pid= 1298
our pid= 0
pid of B = 1299
pid of C = 1300
A: Sleeping...
pid of B = 0
pid of D = 1301
B: Sleeping...
pid of C = 0
C: Sleeping...
pid of D = 0
D: Sleeping...
father is awake!...


A(1298)─┬─B(1299)──D(1301)
        └─C(1300)


father is waiting
My PID = 1297: Child PID = 1298 was terminated by a signal, signo = 9
oslab20@orion:~/ask2$ C: Exiting...
D: Exiting...
My PID = 1299: Child PID = 1301 terminated normally, exit status = 13
B: Exiting...
```

Πρακτικά, η explain_wait_status(pid, status); που βρίσκεται στην main τυπώνει ότι η διεργασία Α τερματίστηκε από το KILL (9) signal που στείλαμε εμείς μέσω του λειτουργικού. Έτσι το πρόγραμμα τερματίστηκε(αφού η αρχική διεργασία παύει να περιμένει το παιδί της να πεθάνει και έκανε return 0;) και μετά οι υπόλοιπες διεργασίες συνέχιζαν να τερματίζουν χωρίς πατέρα πλέον. Αυτό συμβαίνει γιατί στην περίπτωση όπου πεθάνει πρώτα μία διεργασία-πατέρας, όλα τα ορφανά παιδιά του υιοθετούνται από τη διεργασία init(με pid=1).

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

Αν κάνουμε την αλλαγή στην main() και αντί για show_pstree(pid) γράψουμε show_pstree(getpid()) λαμβάνουμε το εξής δέντρο:

```
ask2-fork(3318)─┬─A(3319)─┬─B(3320)──D(3322)
                │         └─C(3321)
                └─sh(3323)──pstree(3324)
```

Όπως περιμέναμε, πλέον η ρίζα του δέντρου που λαμβάνει η show_pstree() σαν όρισμα για τυπώσει το δέντρο είναι η ίδια η ask2-fork, αφού αυτή είναι και η διεργασία απ' την οποία

μέσω της fork() στην main δημιουργούμε την διεργασία(της οποίας το pid λαμβάνουμε απ' την εντολή pid=fork())που κατόπιν ονομάζουμε Α και θέτουμε ως root του δέντρου μας.

Επίσης, εμφανίζονται στο δέντρο η εντολή-διεργασια pstree(που βλέπουμε στην show_pstree()), καθώς και, ο πατέρας της, η sh (standard command language interpreter) που την κάνει interpret, και οποίες καλούνται από την διαδικασία show_pstree().

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

Ο λόγος για τον οποίο είναι απαραίτητος ο καθορισμός ορίων στον αριθμό διεργασιών είναι κυρίως για λόγους ασφαλείας. Είναι πολύ εύκολο, είτε με πρόθεση (κακόβουλο λογισμικό) είτε χωρίς, να δημιουργηθεί ένας ατέρμονος βρόγχος που να δημιουργεί διαρκώς καινούριες διεργασίες. Σε ένα υπολογιστικό σύστημα πολλαπλών χρηστών, κάτι τέτοιο θα εξαντλούσε τους πόρους του λειτουργικού και θα επηρέαζε αρνητικά την απόδοση του συστήματος και την εμπειρία των υπόλοιπων χρηστών.

Άσκηση 2.2

Παρατίθεται ο κώδικας της 2ης άσκησης ask2-tree.c :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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
#include "tree.h"

//NEOS TROPOS
void fork_procs(struct tree_node *root){
   change_pname(root->name);
   int i,j;
   //φτιάχνουμε έναν πίνακα για κάθε διεργασία στην οποία αποθηκεύονται τα pid των παιδιών της
   pid_t pid_child[root->nr_children];
   int status_child;
   //δημιουργει οσα παιδιά χρειαζομαστε και μας δίνεται ότι έχει η κάθε διεργασία
   for(i=0; i<root->nr_children; i++){
       //καλούμε μία fork για την δημιουργία του εκάστοτε παιδιού και αποθηκεύουμε το pid του
       // στην θέση που του αντιστοιχεί στον πίνακα
       pid_child[i]=fork();
       printf("pid_child=%d\n",pid_child[i]); //εκτυπώνουμε το pid του παιδιού που δημιουργήθηκε
       if (pid_child[i] < 0) {
           perror("pid_child: fork");
           exit(1);
       }
       //μόλις φτιαχτούν όλα τα παιδιά (δηλαδή το for στο οποίο βρισκόμαστε τελειώσει)
       // θα βγούμε και θα κοιμίσουμε τον πατέρα, οπότε τότε θα ξεκινήσει η fork να "καλείται"
       // από το εκάστοτε παιδί και άρα θα μπαίνουμε στο if και θα ξεκινάει
       // η αναδρομή προκειμένου να δημιουργηθεί και το υπόλοιπο δέντρο
```

```
        if (pid_child[i] == 0) {
            fork_procs(root->children + i);
            printf("teleiwse h anadromh kai epistrefw\n");
            //exit(13);
        }
    }
    //κοιμίζουμε τον πατέρα ώστε να δημιουργηθεί το υπόλοιπο δέντρο
    printf("%s :Sleeping...\n",root->name);
    sleep(SLEEP_PROC_SEC);
    //αυτός που θα ξυπνήσει πρώτος είναι το root αφού τον κοιμίσαμε και πρώτο
    printf("%s is now awake...\n", root->name);
    if(root->nr_children != 0){
        for(j=0; j<root->nr_children; j++) {
        printf("j = %d\npid_child = %d\n", j, pid_child[j]);
            pid_child[j] = wait(&status_child);
            explain_wait_status(pid_child[j], status_child);
        }
    }

    printf("%s: Exiting...\n", root->name);
    exit(16);
}


int main(int argc, char *argv[])
{
    //εκτυπώνουμε το δέντρο που παίρνουμε σαν input
    struct tree_node *root;

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

    root = get_tree_from_file(argv[1]);
    print_tree(root);

    pid_t pid;
    int status;

    /* Fork root of process tree */
    pid = fork();
    printf("our pid= %d\n", pid);
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
        /* Child */
        fork_procs(root);
        printf("teleiwse na trexei h fork_procs\n");
        exit(1);
    }
    /*
     * Father
     */
    /* for ask2-signals */
    /* wait_for_ready_children(1); */

    /* for ask2-{fork, tree} */
    //κοιμίζουμε τον πατέρα ώστε να προλάβει να δημιουργηθεί το δέντρο πριν αυτός
πεθάνει
    sleep(SLEEP_TREE_SEC);

    /* Print the process tree root at pid */
    //η διεργασία-πατέρας ξυπνάει, το υπόλοιπο δέντρο όλο κοιμάται(οπότε δεν έχει
πεθάνει κανένας ακόμα)
    // και οπότε το δέντρο μπορεί να εκτυπωθεί
```

```
    printf("father is awake!... \n");
    show_pstree(pid);

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

    /* Wait for the root of the process tree to terminate */
    printf("father is waiting \n");
    pid = wait(&status); //κανω την δειργασία-πατέρα να περιμένει να πεθάνουν όλα τα
παιδιά του
    explain_wait_status(pid, status); //θα μας βγαλει ότι πεθανε το A

    return 0;
}
```

Ερωτήσεις:

1) **Με ποια σειρά εμφανίζονται τα μηνύματα έναρξης και τερματισμού των διεργασιών; γιατί;** Εκτελώντας τον κώδικα για διαφορετικά δέντρα, παρατηρούμε ότι τα μηνύματα έναρξης και τερματισμού των διεργασιών εμφανίζονται με σχετικά τυχαίο τρόπο σε κάθε εκτέλεση ακόμα και όταν επρόκειτο για το ίδιο πρόγραμμα. Αυτό εξαρτάται απ' τον χρονοδρομολογητή (scheduler) του Λειτουργικού Συστήματος. Αξίζει να σημειωθεί ότι στην περίπτωση του ενδεικτικού δέντρου της άσκησης, τα μηνύματα έναρξης εμφανίζονται σε κάθε εκτέλεση με BF τρόπο όπως φαίνεται στην παρακάτω εικόνα:

```
oslab20@orion:~/ask2$ ./ask2-tree ourtree
A
        B
                D
        C
our pid= 4972
our pid= 0
pid_child=4973
pid_child=4974
A :Sleeping...
pid_child=0
pid_child=4975
B :Sleeping...
pid_child=0
C :Sleeping...
pid_child=0
D :Sleeping...
father is awake!...


A(4972)─┬─B(4973)───D(4975)
        └─C(4974)


father is waiting
A is now awake...
B is now awake...
C is now awake...
C: Exiting...
My PID = 4972: Child PID = 4974 terminated normally, exit status = 16
D is now awake...
D: Exiting...
My PID = 4973: Child PID = 4975 terminated normally, exit status = 16
B: Exiting...
My PID = 4972: Child PID = 4973 terminated normally, exit status = 16
A: Exiting...
My PID = 4971: Child PID = 4972 terminated normally, exit status = 16
```

Άσκηση 2.3

Παρατίθεται ο κώδικας της 3ης άσκησης ask2-signals.c :

```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)
{
   /*
    * Start
    */
   printf("PID = %ld, name %s, starting...\n",
          (long)getpid(), root->name);
   change_pname(root->name);
   /* ... */
   //όμοια με την προηγούμενη άσκηση
   // φτιάχνουμε έναν πίνακα για κάθε διεργασία στην οποία αποθηκεύονται τα pid των
//παιδιών της
   int i,j;
   pid_t pid_child[root->nr_children];
   int status_child;

   //δημιουργει οσα παιδιά χρειαζομαστε και μας δίνεται ότι έχει η κάθε διεργασία
   for(i=0; i< root->nr_children; i++){
       //καλούμε μία fork για την δημιουργία του εκάστοτε παιδιού και αποθηκεύουμε
το pid του
       // στην θέση που του αντιστοιχεί στον πίνακα
       pid_child[i]= fork();
       printf("pid of child = %d\n", pid_child[i]); //εκτυπώνουμε το pid του
παιδιού που δημιουργήθηκε
       if (pid_child[i] < 0) {
           perror("pid_child: fork");
           exit(1);
       }
       //μόλις φτιαχτούν όλα τα παιδιά (δηλαδή το for στο οποίο βρισκόμαστε
τελειώσει)
       // θα βγούμε και θα πούμε στον πατέρα να περιμένει, οπότε τότε θα ξεκινήσει
η fork να "καλείται"
       // από το εκάστοτε παιδί και άρα θα μπαίνουμε στο if και θα ξεκινάει
       // η αναδρομή προκειμένου να δημιουργηθεί και το υπόλοιπο δέντρο
       if (pid_child[i] == 0) {
           fork_procs(root->children + i);
       }
   }

   /*
    * Suspend Self
    */

   //μόλις φτιαχτούν όλα τα παιδιά (δηλαδή το for στο οποίο βρισκόμαστε τελειώσει)
   // βγαίνουμε και λέμε στον πατέρα πριν πεθάνει είτε σταματήσει,
   // να περιμένει να σταματήσουν(raise SIGSTOP) όλα τα παιδιά του
   if(root->nr_children !=0) { //διεργασία-πατέρας
       //μετά απ' την wait_for_ready_children γυρνάμε στην fork
       // η οποία πλέον θα "καλείται" από τα παιδιά και άρα θα ξεκινήσει η αναδρομή
       wait_for_ready_children(root->nr_children);  //περιμένει να κάνουν raise
SIGSTOP όλα τα παιδιά του
       raise(SIGSTOP); //με το που όλα τα παιδιά σταματάνε με SIGSTOP τότε βάζουμε
τον πατέρα να κάνει το ίδιο
       //φτανουμε σε αυτό το printf την πρώτη φορά με το που στην main πούμε στο
root του δέντρου
       // να συνεχίσει(του στείλουμε SIGCONT)
```

```
        //Τις επόμενες φορές φτάνουμε εδώ όταν στείλουμε σε μία διεργασία(που είχε
παιδιά) το SIGCONT signal
        printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);
        //εχουμε τυπωσει το δέντρο στην main και τα ξυπνάμε όλα ένα-ένα ξεκινώντας
απ' τα παιδιά του root
        for (j = 0; j < root->nr_children; j++) {
            printf("j = %d\n pid_child = %d\n", j, pid_child[j]);
            //στέλνουμε SIGCONT στα παιδιά της διεργασίας στην οποία βρισκόμαστε
            kill(pid_child[j], SIGCONT); //μετά απ' αυτό ο κώδικας συνεχίζει εκεί
που κάναμε SIGSTOP τον εκάστοτε κόμβο
            pid_child[j] = wait(&status_child); //λέμε στην διεργασία-πατέρα να
περιμένει το παιδί της να πεθάνει
            explain_wait_status(pid_child[j], status_child);
        }
    } else {    //διεργασία-φύλλο
        //τα φύλλα πρέπει απλά με το που δημιουργούνται να "σηκώνουν" SIGSTOP
        // αφού δεν έχουν να περιμένουν κάποιον άλλον
        raise(SIGSTOP);
        //φτάνουμε εδώ με το που έχουμε στείλει σε μία διεργασία(χωρίς παιδιά) το
SIGCONT signal
        printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);
    }
    /*
     * Exit
     */
    printf(" %s: Exiting...\n", root->name);
    exit(16);   //βάλαμε σε όλες τις διεργασίες να επιστρέφουν το ίδιο exit status
όταν πεθαίνουν
}

/*
* 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; //to pid toy A tha einai ayto
    int status;
    struct tree_node *root;

    //εκτυπώνουμε το δέντρο που μας δίνεται ως input
    if (argc < 2){
        fprintf(stderr, "Usage: %s <tree_file>\n", argv[0]);
        exit(1);
    }

    /* Read tree into memory */
    //παίρνουμε το root
    root = get_tree_from_file(argv[1]);

    /* Fork root of process tree */
    pid = fork();
    printf("pid from fork in main is: %d\n", pid);
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
        /* Child */
```

```
        fork_procs(root);
        exit(1);
    }

    /*
     * Father
     */
    /* for ask2-signals */
    //η διεργασία-πατέρας περιμένει "όλα" του τα παιδιά να "σηκώσουν" το SIGSTOP, δηλαδή να σταμτήσουν
    // (εδώ  το "όλα "θα είναι 1 γιατί η αρχική διεργασία έχει μόνο 1 παιδί(το root)
    // που δημιουργείται με το fork και δεν "βλέπει" άμεσα όλο το δέντρ)

    wait_for_ready_children(1);
    //φτάνουμε εδώ μόλις κάνει raise SIGSTOP ο πατέρας στην fork_procs, δηλαδή το root του δέντρου,
    // οπότε τώρα όλοι οι κόμβοι-διεργασίες του δέντρου έχουν κάνει SIGSTOP
    // και άρα μπρούμε να το τυπώσουμε το δέντρο χωρίς το ενδεχόμενο κάποια διεργασία να πεθάνει

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

    //με το που το τυπώσουμε ξυπνάμε(της λέμε να κάνει resume με SIGCONT) το παιδί της διεργασίας-πατέρα,
    // το οποίο θα είναι το root του δέντρου και τότε ο κώδικας προχωράει απ'την ακριβώς
    // επόμενη γραμμή απ'αυτήν της εντολής raise(SIGSTOP); στο if(root->nr_children !=0)
    // που αφορά την διεργασία που έχει παιδιά.
    /* for ask2-signals */
    kill(pid, SIGCONT); //έτσι λέμε στο root του δέντρου να κάνει resume αφού πριν το είχαμε σταματήσει

    /* Wait for the root of the process tree to terminate */
    //στην συνέχεια λέμε στην διεργασία-πατέρα να περιμένει να πεθάνουν όλα τα παιδιά του
    // (εδώ το root είναι μόνο αλλά πρακτικά του λέμε να περιμένει να πεθάνουν όλες οι διεργασίες-κόμβοι του δέντρου)
    wait(&status);
    explain_wait_status(pid, status);

    return 0;
}
```

Η έξοδος του προγράμματος για το outree παρουσιάζεται παρακάτω:

```
oslab20@orion:~/ask2$ ./ask2-signals ourtree
pid from fork in main is: 25654
pid from fork in main is: 0
PID = 25654, name A, starting...
pid of child = 25655
pid of child = 25656
pid of child = 0
PID = 25656, name C, starting...
My PID = 25654: Child PID = 25656 has been stopped by a signal, signo = 19
pid of child = 0
PID = 25655, name B, starting...
pid of child = 25657
pid of child = 0
PID = 25657, name D, starting...
My PID = 25655: Child PID = 25657 has been stopped by a signal, signo = 19
My PID = 25654: Child PID = 25655 has been stopped by a signal, signo = 19
My PID = 25653: Child PID = 25654 has been stopped by a signal, signo = 19


A(25654)─┬─B(25655)──D(25657)
         └─C(25656)


PID = 25654, name = A is awake
j = 0
 pid_child = 25655
PID = 25655, name = B is awake
j = 0
 pid_child = 25657
PID = 25657, name = D is awake
 D: Exiting...
My PID = 25655: Child PID = 25657 terminated normally, exit status = 16
 B: Exiting...
My PID = 25654: Child PID = 25655 terminated normally, exit status = 16
j = 1
 pid_child = 25656
PID = 25656, name = C is awake
 C: Exiting...
My PID = 25654: Child PID = 25656 terminated normally, exit status = 16
 A: Exiting...
My PID = 25653: Child PID = 25654 terminated normally, exit status = 16
```

1. **Στις προηγούμενες ασκήσεις χρησιμοποιήσαμε τη sleep() για τον συγχρονισμό των διεργασιών. Τι πλεονεκτήματα έχει η χρήση σημάτων;**
   Στις προηγούμενες ασκήσεις στηριζόμασταν στην εντολή sleep() για τον συγχρονισμό των διεργασιών. Με άλλα λόγια, ορίζαμε αυθαίρετα sleeping time για την διεργασία-πατέρα και τα παιδιά του ελπίζοντας:
   α) να κοιμηθεί αρκετά ο αρχικός πατέρας έτσι ώστε να προλάβει να δημιουργηθεί το δέντρο και επομένως να το τυπώσει μόλις αυτός ξυπνήσει
   β) να κοιμηθούν αρκετά οι διεργασίες-φύλλα έτσι ώστε να μην πεθάνουν προτού τυπώσει το δέντρο ο αρχικός πατέρας(στην main)..

Προφανώς, μία τέτοια προσέγγιση εγκυμονεί τον κίνδυνο να μην οριστεί αρκετός χρόνος ύπνου για την κάθε διεργασία, συν το γεγονός ότι το πρόγραμμα αργεί πολύ, και πιθανώς άσκοπα, να τερματίσει.
Μια λύση για τα παραπάνω υλοποιήθηκε στην συγκεκριμένη άσκηση με την χρήση σημάτων(signals).

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 να κάνει την διεργασία-πατέρα που την κάλεσε να περιμένει όλα τα παιδιά της να σταματήσουν(δηλαδή να κάνουν raise SIGSTOP). Η χρήση της εξασφαλίζει ότι:
α) Θα τυπωθεί σωστά το δέντρο εφόσον κάθε διεργασία-πατέρας περιμένει τα παιδιά του να σταματήσουν προτού σταματήσει και ο ίδιος. Έτσι, αναδρομικά εξασφαλίζουμε ότι όλο το δέντρο είναι σταματημένο όταν ο αρχικός πατέρας(πάλι μέσω της wait_for_ready_children()) είναι σε θέση να το τυπώσει.
β) Αποφεύγονται περιπτώσεις άστοχης αποστολής και λήψης σημάτων μεταξύ των διεργασιών. Ένα παράδειγμα αυτού θα ήταν: μια διεργασία-πατέρας να στείλει πρόωρα σήμα SIGCONT() σε κάποιο παιδί της που ενδεχομένως να μην έχει σταματήσει ακόμα. Σε αυτήν την περίπτωση το παιδί θα παρέμενε σταματημένο για πάντα.
γ) Εξασφαλίζεται ότι η σειρά εμφάνισης των μηνυμάτων τερματισμού των διεργασιών θα γίνεται προκαθορισμένα με DF τρόπο.

Άσκηση 2.4

Παρατίθεται ο κώδικας της 4ης άσκησης ask2-pipes.c :

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

#include <sys/wait.h>

#include "proc-common.h"
#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3
#include "tree.h"

void child(int fd,struct tree_node *root){
   //βάζουμε στην διεργασία το όνομα που μας δίνεται ότι θα έχει
   change_pname(root->name);
   printf("%s: Created\n", root->name);
   //τσεκάρουμε αν το δέντρο είναι σωστό
   if((strcmp(root->name, "+") == 0) || (strcmp(root->name, "*") == 0)){
//κόμβος-τελεστής(μη τερματικός κόμβος)
       //πρέπει να έχει υποχρεωτικά ακριβώς 2 παιδιά
       if(root->nr_children != 2){
           perror("child: invalid number of children in + or *");
           exit(1);
       }
       //ΜΗ ΤΕΡΜΑΤΙΚΟΣ ΚΟΜΒΟΣ!
       int i,j;
```

```
        //φτιάχνουμε έναν πίνακα που θα αποθηκεύουμε τα 2 pid των παιδιών
του μη τερματικού κόμβου
        pid_t pid_child[2];
        int status_child;
        double result[2];
        int pfd[2];
        //φτιάχνουμε το pipe
        printf("%s: Creating pipe\n", root->name);
        if (pipe(pfd) < 0) {
            perror("pipe");
            exit(1);
        }
        //printf("pfd[0]= %d , pfd[1]= %d\n",pfd[0], pfd[1]);
        //δημιουργούμε τα 2 παιδιά της διεργασίας-πατέρα και αποθηκεύουμε
στον πίνακα τα pid του καθενός
        for(i=0; i<2; i++){
            pid_child[i]= fork();
            printf("our pid= %d\n", pid_child[i]);
            if (pid_child[i] < 0) {
                perror("pid_child: fork");
                exit(1);
            }
            if (pid_child[i] == 0) {
                //ΑΝΑΔΡΟΜΗ
                //θέλουμε το παιδί να γράψει την πληροφ αρα του δίνουμε το
write end του pipe που
                // φτιάξαμε όταν "τρέχαμε" απ' τον πατέρα για να επικοινωνεί
με τα παιδιά του
                //printf("h anadromh kaleitai gia fd=pfd[1]= %d\n", pfd[1]);
                child(pfd[1],root->children + i); //όπου βλέπει fd θα βάζει
το pfd[1] κ όπου root το root->children + i
            }
        }
        //λέμε στην διεργασία-πατέρα να περιμένει τα 2 παιδιά του να κάνουν
raise SIGSTOP
        wait_for_ready_children(2); //γυρνάει λοιπόν στο fork και μπαίνει
στο if για την αναδρομή
        raise(SIGSTOP); //αφού έχουμε σταματήσει με SIGSTOP τα παιδιά, το
κάνουμε και για τον πατέρα τους
        printf("PID = %ld, name = %s is awake\n", (long)getpid(), root-
>name);

        //έχουμε τυπώσει το δέντρο στην main και τα ξυπνάμε όλα ένα-ένα
ξεκινώντας απ' τα παιδιά του root
        for (j = 0; j < root->nr_children; j++) {
            //printf("j = %d\n pid_child = %d\n", j, pid_child[j]);
            //στέλνουμε SIGCONT στα παιδιά της διεργασίας στην οποία
βρισκόμαστε
            kill(pid_child[j], SIGCONT); //μετά απ' αυτό ο κώδικας συνεχίζει
εκεί που κάναμε SIGSTOP τον εκάστοτε κόμβο
            pid_child[j] = wait(&status_child);
            explain_wait_status(pid_child[j], status_child);
            //αφού πεθάνει το παιδί θέλουμε να κάνουμε την πράξη
```

```
            double res=0;
            //κλείνουμε το write end του pipe με το οποίο ο πατέρας
επικοινωνεί με το παιδί του
            // ώστε πλέον να μην υπάρχει κάποιο "ενεργό" fd με το οποίο ο
πατέρας να έχει ακόμα την δυνατότητα
            // να γράψει σε αυτό το pipe και δεν τερματίσει ποτέ το
πρόγραμμα
            //εδώ έχουμε βάλει την read να διαβάζει ακριβώς τους χαρακτήρες
που θέλουμε
            // οπότε ακόμη και χωρίς το close() θα τύχαινε να δούλευε αλλά
αν απλά λέγαμε να
            // διαβάσει ό,τι υπάρχει απ'το read end τότε η read θα είχε
μπλοκάρει
            close(pfd[1]);
            //ο πατέρας διαβάζει απ'το read end του pipe μέσω του οποίου
επικοινωνεί
            // με το παιδί του την τιμή που το παιδί πέρασε στο write end
πριν πεθάνει
            if (read(pfd[0], &res,sizeof(res)) != sizeof(res)) {
                perror("parent: read from pipe");
                exit(1);
            }
            printf("res=%f\n", res);
            result[j]=res;
            //printf("result[%d]=%f\n", j, res);
        }
        //μετά την for έχουμε αποθηκεύσει στο result τις τιμές των δύο
παιδιών(που πέθαναν)
        // και βρισκόμαστε στον πατέρα
        double final_res=0;
        if(strcmp(root->name, "+") == 0){
           final_res= result[0] + result[1];
        }
        else{
           final_res= result[0] * result[1];
        }

        //printf("grafoyme thn timh toy final result sto fd= %d\n", fd);
        //printf("enw to pfd[1]=%d\n", pfd[1]);
        //γράφουμε το final result στο write end του pipe που ενώνει την
διεργασία που βρισκόμαστε με τον πατέρα  του
        // και όχι στο pipe που ενώνει την διεργασία που βρισκόμαστε με τα
παιδιά του(σε εκείνη την περίπτωση
        // αντί για fd θα γράφαμε pfd[1]).
        // Εδώ δηλαδή θέλουμε να περάσουμε την τιμή του παιδιού στο ίδιο fd
που περνάμε στην αναδρομή
        if (write(fd, &final_res,sizeof(final_res)) != sizeof(final_res)) {
            perror("parent: write to pipe");
            exit(1);
        }
    }
    //ΤΕΡΜΑΤΙΚΟΣ ΚΟΜΒΟΣ!
    else{
```

```
        if(root->nr_children != 0){
            perror("child: invalid number of children in digit");
            exit(1);
        }
        //τα φύλλα πρέπει απλά με το που δημιουργούνται να "σηκώνουν"
SIGSTOP
        // αφού δεν έχουν να περιμένουν κάποιον άλλον
        raise(SIGSTOP);
        //φτάνουμε εδώ με το που έχουμε στείλει σε μία διεργασία(χωρίς
παιδιά) το SIGCONT signal
        printf("PID = %ld, name = %s is awake\n", (long)getpid(), root-
>name);
        //περνάμε την τιμή του παιδιού στο pipe που το ίδιο επικοινωνεί με
τον πατέρα
        double root_name = atof(root->name); //μετατρέπουμε τον pointer σε
double για να δουλέψει καλά η write
        //θέλουμε να γράψουμε την τιμή του παιδιού στο pipe ώστε πρακτικά το
παιδί να "στείλει" την τιμή του στον πατέρα
        //και αυτός να μπορεί να την διαβάσει απ' το read end του ίδιου pipe
        if (write(fd, &root_name,sizeof(root_name)) != sizeof(root_name)) {
            perror("child: write to pipe");
            exit(1);
        }
        close(fd);
    }
    /*
    * Exit
    */
    printf(" %s: Exiting...\n", root->name);
    exit(16);
}

int main(int argc, char *argv[]){

    //εκτυπώνουμε το δέντρο που παίρνουμε σαν input
    struct tree_node *root;

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

    pid_t p;
    int status;
    int pfd[2];

    printf("Parent: Creating pipe...\n");

    if (pipe(pfd) < 0) {
        perror("pipe");
```

```
        exit(1);
    }
    printf("Parent: Creating child...\n");
    p = fork();
    if (p < 0) {
        /* fork failed */
        perror("fork");
        exit(1);
    }
    if (p == 0) {
        /* In child process */
        child(pfd[1],root);
        /*
         * Should never reach this point,
         * child() does not return
         */
        assert(0);
    }

    wait_for_ready_children(1);
    /* Print the process tree root at pid */
    //όταν όλο το δέντρο έχει σηκώσει SIGSTOP το εκτυπώνουμε
    show_pstree(p);
    //ξυπνάμε τον πατέρα όλων
    kill(p, SIGCONT);
    /* Wait for the child to terminate */
    printf("Parent: Created child with PID = %ld, waiting for it to
terminate...\n",(long)p);
    p = wait(&status); //δεν έχει διαφορά και αν βάζαμε σκέτο wait(&status);
αφού η αρχική διεργασία έχει μόνο 1 παιδί
    explain_wait_status(p, status);
    //αφού πεθάνουν όλα τα παιδιά έχουμε το τελικό αποτέλεσμα στο pipe
    double final_val;
    if (read(pfd[0], &final_val,sizeof(final_val)) != sizeof(final_val)) {
        perror("initial parent: read from pipe");
        exit(1);
    }
    printf("FINAL RESULT: %f\n", final_val);
    printf("Parent: All done, exiting...\n");

    return 0;
}
```

Παρουσιάζεται η έξοδος του προγράμματός για το παράδειγμα της εκφώνησης:

```
oslab20@orion:~/ask2$ ./ask2-pipes given_math
*
        10
        +
                6
                5
Parent: Creating pipe...
Parent: Creating child...
*: Created
*: Creating pipe
our pid= 27003
our pid= 27004
our pid= 0
10: Created
My PID = 27002: Child PID = 27003 has been stopped by a signal, signo = 19
our pid= 0
+: Created
+: Creating pipe
our pid= 27005
our pid= 27006
our pid= 0
6: Created
our pid= 0
5: Created
My PID = 27004: Child PID = 27005 has been stopped by a signal, signo = 19
My PID = 27004: Child PID = 27006 has been stopped by a signal, signo = 19
My PID = 27002: Child PID = 27004 has been stopped by a signal, signo = 19
My PID = 27001: Child PID = 27002 has been stopped by a signal, signo = 19


*(27002)─┬─+(27004)─┬─5(27006)
         │          └─6(27005)
         └─10(27003)


Parent: Created child with PID = 27002, waiting for it to terminate...
PID = 27002, name = * is awake
PID = 27003, name = 10 is awake
 10: Exiting...
My PID = 27002: Child PID = 27003 terminated normally, exit status = 16
res=10.000000
PID = 27004, name = + is awake
PID = 27005, name = 6 is awake
 6: Exiting...
```

```
My PID = 27004: Child PID = 27005 terminated normally, exit status = 16
res=6.000000
PID = 27006, name = 5 is awake
 5: Exiting...
My PID = 27004: Child PID = 27006 terminated normally, exit status = 16
res=5.000000
 +: Exiting...
My PID = 27002: Child PID = 27004 terminated normally, exit status = 16
res=11.000000
 *: Exiting...
My PID = 27001: Child PID = 27002 terminated normally, exit status = 16
FINAL RESULT: 110.000000
Parent: All done, exiting...
```

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;

Το πόσες σωληνώσεις χρειάζεται κάθε διεργασία εξαρτάται απ' το είδος της. Στην περίπτωση των διεργασιών-φύλλων, χρειαζόμαστε μία σωλήνωση με την οποία επικοινωνεί με την διεργασία-πατέρα του Στην, δε, περίπτωση ενός μη τερματικού κόμβου(που περιέχει τελεστή), χρειαζόμαστε 2 σωληνώσεις· μία για να επικοινωνεί με τα παιδιά του και μία για να επικοινωνεί με τον πατέρα του.
Ο λόγος για τον οποίο κάθε γονική διεργασία μπορεί να χρησιμοποιεί μόνο μία σωλήνωση για να επικοινωνεί με όλα τα παιδιά της είναι επειδή ,στο συγκεκριμένο πρόγραμμα, περιέχονται μόνο αντιμεταθετικοί τελεστές(πρόσθεση και πολλαπλασιασμός) και επομένως δεν μας ενδιαφέρει η σειρά με την οποία αυτή λαμβάνει την πληροφορία των παιδιών της.
Κάτι τέτοιο δεν θα λειτουργούσε στην περίπτωση που παίζαμε και με μη αντιμεταθετικούς τελεστές. Σε αυτή την περίπτωση, κάθε γονική διεργασία θα χρειαζόταν τόσες σωληνώσεις όσες και οι διεργασίες παιδιά της συν την μία επιπλέον με την οποία θα επικοινωνεί με τον πατέρα της.

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 ένα σύστημα πολλαπλών επεξεργαστών μπορούν να εκτελεστούν περισσότερες από μία διεργασίες παράλληλα. Έτσι είναι προφανές ότι μία σχετικά μεγάλη αριθμητική έκφραση μπορεί να πραγματοποιηθεί σε λιγότερο χρόνο αν αποτιμηθεί από δέντρο διεργασιών, όπου υπάρχει η δυνατότητα υπολογισμού μαθηματικών πράξεων ταυτόχρονα, απ' ότι από μία μόνο διεργασία στην οποία όλες οι μαθηματικές πράξεις θα έπρεπε να εκτελούνται σειριακά.

Παρατίθεται το Makefile:

```
.PHONY: all clean

all: fork-example tree-example ask2-fork ask2-tree ask2-signals ask2-pipes

CC = gcc
CFLAGS = -g -Wall -O2
SHELL= /bin/bash

tree-example: tree-example.o tree.o
	$(CC) $(CFLAGS) $^ -o $@

ask2-tree: ask2-tree.o tree.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

fork-example: fork-example.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

ask2-fork: ask2-fork.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

ask2-signals: ask2-signals.o proc-common.o tree.o
	$(CC) $(CFLAGS) $^ -o $@

ask2-pipes: ask2-pipes.o proc-common.o tree.o
	$(CC) $(CFLAGS) $^ -o $@

pipe-example: pipe-example.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

%.s: %.c
	$(CC) $(CFLAGS) -S -fverbose-asm $<

%.o: %.c
	$(CC) $(CFLAGS) -c $<

%.i: %.c
	gcc -Wall -E $< | indent -kr > $@

clean:
	rm -f *.o tree-example fork-example pstree-this ask2-{fork,tree,signals,pipes}
```