ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Λειτουργικά Συστήματα

7ο εξάμηνο, Ακαδημαϊκή περίοδος 2017-2018

Άσκηση 2: Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

	Team: oslaba27
Ονοματεπώνυμο	Αριθμός Μητρώου
Κερασιώτης Ιωάννης	03114951
Ραφτόπουλος Ευάγγελος	03114743

Άσκηση 1.1

Πηγαίος Κώδικας:

ask2-fork1.c

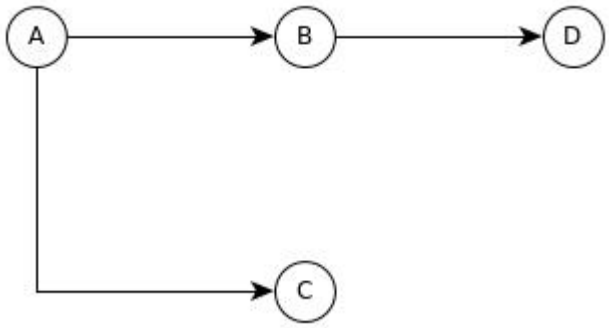
```
/*****************************************************************/
/*                          Operating Systems                    */
/*                          Second Lab Exercise                  */
/*                                                               */
/* Exercise 1.1 ask2-fork1.c                                     */
/*                                                               */
/* Team: oslaba27                                                */
/*                                                               */
/* Full Name: Kerasiotis Ioannis, Student ID: 03114951           */
/* Full Name: Raftopoulos Evangelos, Student ID: 03114743        */
/*                                                               */
/*****************************************************************/


#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

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

        change_pname("A"); /* Name of the proccess:A */
        printf("A with PID = %ld is created...\n", (long)getpid());

        pid_t pid_b, pid_c, pid_d;
        int status;
```

```
	printf( "A with PID = %ld: Creating child B...\n", (long)getpid());
	pid_b = fork(); /* Create proccess B */
	if (pid_b < 0){ /* Checking if fork() succeeded */
		perror("B: fork");
		exit(1);
	}
	if (pid_b == 0){ /* child's progress */
		change_pname("B"); /* Name of the proccess: B */
		printf("B with PID = %ld is created...\n", (long)getpid());
		printf( "B with PID = %ld: Creating child D...\n", (long)getpid());
		pid_d = fork(); /* Create proccess D */
		if (pid_d < 0){ /* Checking if fork() succeeded */
			perror("D: fork");
			exit(1);
		}
		if (pid_d == 0){ /* child's progress */
			change_pname("D"); // Name of the proccess: D
			printf("D with PID = %ld is created...\n", (long)getpid());
			/* Leaves must sleep */
			printf( "D with PID = %ld is ready to sleep...\n",(long)getpid());
			sleep(SLEEP_PROC_SEC);
			printf("D with PID = %ld is ready to terminate...\nD: Exiting...\n",
(long)getpid());
			exit(13);
		}
		/* Parent of D (B) waits to change D its status to continue this progress */
		waitpid(pid_d, &status, 0);
		explain_wait_status(pid_d, status); // … and explain the reason
		printf("B with PID = %ld is ready to terminate...\nB: Exiting...\n", (long)getpid());
		exit(19);
	}

	printf( "A with PID = %ld: Creating child C...\n", (long)getpid());
	pid_c = fork(); // Create proccess C
	if (pid_c < 0) { // Checking if fork() succeeded
		perror("C: fork");
		exit(1);
	}
	if (pid_c == 0){ // child's progress
		change_pname("C"); // Name of the proccess: C
		printf("C with PID = %ld is created...\n", (long)getpid());
		/* Leaves must sleep */
		printf("C with PID = %ld is ready to sleep...\n",(long)getpid());
		sleep(SLEEP_PROC_SEC);
		printf("C with PID = %ld is ready to terminate...\nC: Exiting...\n", (long)getpid());
		exit(17);
	}
```

```c
        /* Parent of C (A) waits to change C its status to continue this progress */
        waitpid(pid_c, &status, 0);
        explain_wait_status(pid_c, status); // … and explain the reason

        /* Parent of B (A) waits to change A its status to continue this progress */
        waitpid(pid_b, &status, 0);
        explain_wait_status(pid_b, status); // … and explain the reason

        printf("A with PID = %ld is ready to terminate...\nA: Exiting...\n", (long)getpid());
        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().
 */
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
         */

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

        /* Wait for the root of the process tree to terminate */
        waitpid(pid, &status, 0);
        explain_wait_status(pid, status);

        return 0;
}
```

Έξοδος εκτέλεσης προγράμματος:

Το πρόγραμμα ask2-fork1 δεν έχει κάποια είσοδο και σαν έξοδο το δέντρο διεργασιών του σχήματος και τα κατάλληλα μηνύματα. Το δοθέν δέντρο της εκφώνησης είναι το παρακάτω:



Η έξοδος του προγράμματος είναι η εξής:

A with PID = 3523 is created...
A with PID = 3523: Creating child B...
A with PID = 3523: Creating child C...
B with PID = 3525 is created...
B with PID = 3525: Creating child D...
C with PID = 3526 is created...
C with PID = 3526 is ready to sleep...
D with PID = 3527 is created...
D with PID = 3527 is ready to Sleep...

A(3523)─-─B(3525)———D(3527)
└-----─C(3526)

C with PID = 3526 is ready to terminate...
C: Exiting...
D with PID = 3527 is ready to terminate...
D: Exiting...
My PID = 3525: Child PID = 3527 terminated normally, exit status = 13
My PID = 3523: Child PID = 3526 terminated normally, exit status = 17
B with PID = 3525 is ready to terminate...
B: Exiting...
My PID = 3523: Child PID = 3525 terminated normally, exit status = 19
A with PID = 3523 is ready to terminate...
A: Exiting...
My PID = 3522: Child PID = 3523 terminated normally, exit status = 16

Ερωτήσεις:

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

   Σκοτώντας πρόωρα την διεργασία Α με την δοθέσα εντολή, τα παιδιά δεν πεθαίνουν αλλά "υιοθετούνται" από την init. Η init είναι η πρώτη διεργασία κατά την εκκίνηση του υπολογιστικού συστήματος που εκτελείτε στο υπόβαθρο μέχρι τον τερματισμό. Η init αναλαμβάνει τον τερματισμό των "ορφανών" διεργασιών κάνοντας συνεχόμενα wait().

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

   Από το manual της getpid() γνωρίζουμε πως η συγκεκριμένη κλήση συστήματος επιστρέφει το process ID της καλούμενης διεργασίας. Βλέποντας πως στην main γίνεται fork() δημιουργώντας ένα παιδί που στην συνέχεια του δίνεται το όνομα A. Παρόλα αυτά η διεργασία που εκτελεί το κομμάτι του κώδικα που καλείται η show_pstree είναι η ask4-fork1. Επομένως το δέντρο διεργασιών που θα δούμε είναι το παρακάτω.

   ```
   ask2-fork1(18207)─┬─A(18208)─┬─B(18210)──D(18212)
                     │          └─C(18211)
                     └─sh(18209)──pstree(18213)
   ```

   Εκτός από την ask2-fork1, που ήταν αναμενόμενη, βλέπω δύο ακόμα διεργασίες: την sh και την pstree που είναι παιδί της. Επειδή η συνάρτηση show_pstree κάνει χρήση της συνάρτησης system καλόντας την pstree. Επομένως η ask2-fork1 καλεί και την διεργασία του bash η οποία με την σειρά της καλεί την pstree.

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

   Οι διαχειριστές θέτουν όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει για το λόγο πως κάποιο πρόγραμμα θα μπορούσε να δημιουργεί συνεχώς νέες διεργασίες, οι οποίες θα απασχολούσαν πόρους του συστήματος. Είναι αξιοσημείωτο να αναφέρουμε ότι το παραπάνω είναι γνωστό ως επίθεση Form Bomb (ή Rabbit Virus ή Wabbit). Συγκεκριμένα, χρησιμοποιούντας τις παρακάτω εντολές

   ```
   while(1){
       fork();
   }
   ```

   Δημιουργούμε διεργασίες με εκθετικό ρυθμό, απασχολόντας χρόνο και μνήμη του επεξεργαστή.

Άσκηση 1.2

Πηγαίος Κώδικας:

ask2-fork2.c

```
/*****************************************************************/
/*                          Operating Systems                    */
/*                          Second Lab Exercise                  */
/*                                                               */
/* Exercise 1.2 ask2-fork2.c                                     */
/*                                                               */
/* Team: oslaba27                                                */
/*                                                               */
/* Full Name: Kerasiotis Ioannis, Student ID: 03114951           */
/* Full Name: Raftopoulos Evangelos, Student ID: 03114743        */
/*                                                               */
/*****************************************************************/

#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"

#define SLEEP_TREE_SEC  3
#define SLEEP_PROC_SEC 10

void fork_procs(struct tree_node *t){

        int i;

        printf("PID = %ld, name %s, starting...\n",  (long)getpid(), t->name);
        change_pname(t→name); // change name of proccess

        if (t->nr_children == 0){ // if it is leaf
                sleep(SLEEP_PROC_SEC); // leaves must sleep
                printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n", t->name,
(long)getpid(), t->name);
                exit(0);
        }
        else{
```

```
		int status;
		pid_t pid[t->nr_children];

		for(i=0; i<t->nr_children; i++){ // loop to create a proccess for all children

			pid[i] = fork(); // create new proccess
			if (pid[i] < 0) { // check if fork() succeeded
				perror("proc: fork");
				exit(1);
			}
			if (pid[i] == 0) {
				/* Child */
				fork_procs(t→children+i); // recursively navigate the tree
				exit(1);
			}

		}
		for (i=0; i<t->nr_children; i++){
			waitpid(pid[i], &status, 0); // wait every children to change status
			explain_wait_status(pid[i], status); //… and explain the reason
		}
	}
	printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n", t->name, (long)getpid(), t-
>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[]){
	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]);
//      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
         */
        /* 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);

        /* Wait for the root of the process tree to terminate */
        waitpid(pid, &status, 0);
        explain_wait_status(pid, status);

        return 0;
}
```

Έξοδος εκτέλεσης προγράμματος:

Το πρόγραμμα παίρνει ένα αρχείο που αναπαραστά ένα δέντρο με την παρακάτω μορφή.

nodeName
numberChildren
childNo1
childNo2
...
childNoN

Για παράδειγμα, το δέντρο στην άσκηση 1 θα ήταν το εξής:

```
A
2
B
C

B
1
D

C
0
```

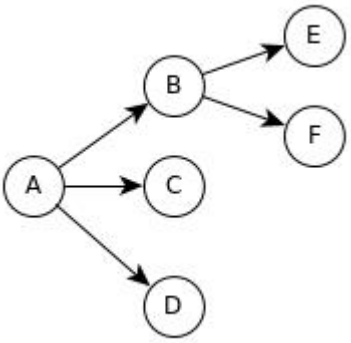
Το πρόγραμμα θα δημιουργεί της διεργασίες, θα τις μετονομάζει και αφού κοιμίζει τα φύλλα να εμφανίζει το δέντρο διεργασιών. Για τα παραπάνω ενημερώνει με αντίστοιχα μηνύματα.

Παρακάτω θα παρατεθούν οι έξοδοι του προγράμματος από δύο δέντρα:

- proc.tree

```
# file that defines the tree
# lines starting with '#' are comments
# . each block of lines defines a node
# . each node is defined as:
#    1st line:         name of node
#    2nd line:         number of children
#    subsequent lines: name(s) of children
# . blocks are seperated with empty lines
# . no comments are allowed within a block
# . nodes must be placed in a DFS order

A
3
B
C
D

B
2
E
F

E
0

F
0

C
0

D
0
```

Το παραπάνω δέντρο εκτιμάτε να έχει ως έξοδο διεργασιών:



Η έξοδος του προγράμματος φαίνεται στην παρακάτω εικόνα:

PID = 17284, name A, starting...
PID = 17285, name B, starting...
PID = 17286, name C, starting...
PID = 17287, name D, starting...
PID = 17288, name E, starting...
PID = 17289, name F, starting...

```
A(17284)─┬─B(17285)─┬─E(17288)
         │          └─F(17289)
         ├─C(17286)
         └─D(17287)
```

C with PID = 17286 is ready to terminate...
C: Exiting...
D with PID = 17287 is ready to terminate...
D: Exiting...
E with PID = 17288 is ready to terminate...
E: Exiting...
F with PID = 17289 is ready to terminate...
F: Exiting...
My PID = 17285: Child PID = 17288 terminated normally, exit status = 0
My PID = 17285: Child PID = 17289 terminated normally, exit status = 0
B with PID = 17285 is ready to terminate...
B: Exiting...
My PID = 17284: Child PID = 17285 terminated normally, exit status = 0
My PID = 17284: Child PID = 17286 terminated normally, exit status = 0
My PID = 17284: Child PID = 17287 terminated normally, exit status = 0
A with PID = 17284 is ready to terminate...
A: Exiting...
My PID = 17283: Child PID = 17284 terminated normally, exit status = 0

- example.tree

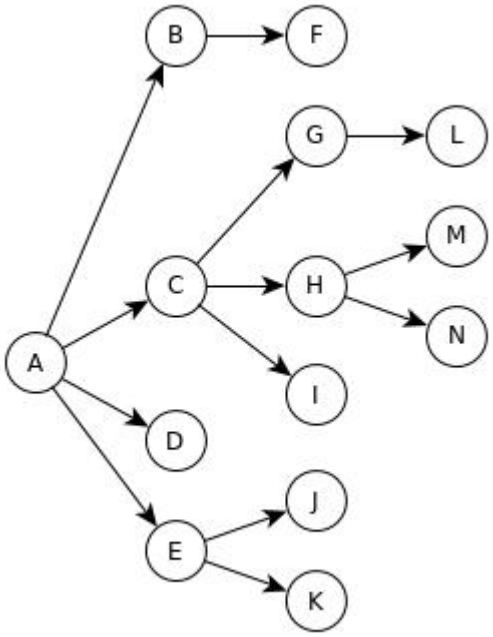
```
# file that defines the tree
# lines starting with '#' are comments
# . each block of lines defines a node
# . each node is defined as:
#    1st line:         name of node
#    2nd line:         number of children
#    subsequent lines: name(s) of children
# . blocks are seperated with empty lines
# . no comments are allowed within a block
# . nodes must be placed in a DFS order

A
4
B
C
D
E

B
1
F

F
0

C
3
G
H
I

G
2
L
M

L
0

M
0

H
1
N

N
0

I
0

D
0

E
```

2
J
K

J
0

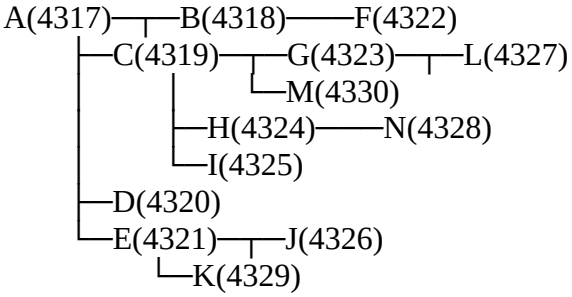
K
0

Επομένως το δέντρο θα είναι το παρακάτω:



Και επιβεβαιώνεται από την έξοδο του προγράμματος:

PID = 4317, name A, starting...
PID = 4318, name B, starting...
PID = 4319, name C, starting...
PID = 4320, name D, starting...
PID = 4321, name E, starting...
PID = 4323, name G, starting...
PID = 4324, name H, starting...
PID = 4322, name F, starting...
PID = 4329, name K, starting...
PID = 4330, name M, starting...
PID = 4327, name L, starting...
PID = 4326, name J, starting...
PID = 4328, name N, starting...
PID = 4325, name I, starting...



D with PID = 4320 is ready to terminate...
D: Exiting...
F with PID = 4322 is ready to terminate...
F: Exiting...
K with PID = 4329 is ready to terminate...
M with PID = 4330 is ready to terminate...
L with PID = 4327 is ready to terminate...
K: Exiting...
M: Exiting...
L: Exiting...
N with PID = 4328 is ready to terminate...
N: Exiting...
My PID = 4318: Child PID = 4322 terminated normally, exit status = 0
B with PID = 4318 is ready to terminate...
B: Exiting...
I with PID = 4325 is ready to terminate...
I: Exiting...
My PID = 4323: Child PID = 4327 terminated normally, exit status = 0
My PID = 4323: Child PID = 4330 terminated normally, exit status = 0
G with PID = 4323 is ready to terminate...
G: Exiting...
My PID = 4324: Child PID = 4328 terminated normally, exit status = 0
H with PID = 4324 is ready to terminate...
H: Exiting...
J with PID = 4326 is ready to terminate...
J: Exiting...
My PID = 4319: Child PID = 4323 terminated normally, exit status = 0
My PID = 4317: Child PID = 4318 terminated normally, exit status = 0
My PID = 4319: Child PID = 4324 terminated normally, exit status = 0
My PID = 4321: Child PID = 4326 terminated normally, exit status = 0
My PID = 4319: Child PID = 4325 terminated normally, exit status = 0
C with PID = 4319 is ready to terminate...
C: Exiting...
My PID = 4321: Child PID = 4329 terminated normally, exit status = 0
E with PID = 4321 is ready to terminate...
E: Exiting...
My PID = 4317: Child PID = 4319 terminated normally, exit status = 0
My PID = 4317: Child PID = 4320 terminated normally, exit status = 0

My PID = 4317: Child PID = 4321 terminated normally, exit status = 0
A with PID = 4317 is ready to terminate...
A: Exiting...
My PID = 4316: Child PID = 4317 terminated normally, exit status = 0

Ερωτήσεις:

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

   Το πρόγραμμα αρχικά δημιουργεί τους κόμβους -και όπως είναι λογικό – αρχίζοντας από τους γονείς και εκ τως υστέρων δημιουργεί και τα παιδιά-φύλλα, επομένως η δημιουργία γίνεται με top-down λογική. Αντίθετα, τερματισμός γίνεται με bottom-up λογική, καθώς θέλουμε να τερματιστούν πρώτα τα φύλλα και μετά οι εσωτερικοί κόμβοι, διότι θέλουμε να πεθάνουν ομαλά τα παιδιά πόδια και να μην μείνουν ορφανά.

Άσκηση 1.3

Πηγαίος Κώδικας:

ask2-fork3.c

```
/*****************************************************************/
/*                          Operating Systems                    */
/*                          Second Lab Exercise                  */
/*                                                               */
/* Exercise 1.4 ask2-fork4.c                                     */
/*                                                               */
/* Team: oslaba27                                                */
/*                                                               */
/* Full Name: Kerasiotis Ioannis, Student ID: 03114951           */
/* Full Name: Raftopoulos Evangelos, Student ID: 03114743        */
/*                                                               */
/*****************************************************************/

#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 *t){
        /*
         * Start
         */

        int i;

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

        if(t->nr_children == 0) { //if it is leaf

                raise(SIGSTOP); // sleep
                printf("PID = %ld, name = %s is awake\n",(long)getpid(), t->name);
        }
        else{

                int status[t->nr_children];
```

```
                pid_t pid[t->nr_children];

                for (i=0; i<t->nr_children; i++){  // loop to create a proccess for all children

                        pid[i]=fork(); //create proccess
                        if (pid[i] < 0) { // check if fork() succeeded
                                perror("proc: fork");
                                exit(1);
                        }
                        if (pid[i] == 0) {
                                /* Child */
                                fork_procs(t->children+i);
                                exit(1);
                        }

                        wait_for_ready_children(1); //wait child to sleep

                }

                raise(SIGSTOP); // sleep
                printf("PID = %ld, name = %s is awake\n", (long)getpid(), t->name);

                for(i=0 ; i<t->nr_children ; i++){
                        kill(pid[i],SIGCONT); // wake up children
                        waitpid(pid[i], &status[i], 0); //wait to change status
                        explain_wait_status(pid[i],status[i]); // … and explain
                }
        }
        /*
         * Exit
         */
        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[])
{
```

```c
        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]);

        /* 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
         */
        /* 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 */
        wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

Έξοδος εκτέλεσης προγράμματος:

Όπως και στην προηγούμενη άσκηση θα χρησιμοποιήσουμε τα δέντρα proc.tree και example.tree.

- proc.tree

```
PID = 6701, name A, starting...
PID = 6702, name B, starting...
PID = 6703, name E, starting...
My PID = 6702: Child PID = 6703 has been stopped by a signal, signo = 19
PID = 6704, name F, starting...
My PID = 6702: Child PID = 6704 has been stopped by a signal, signo = 19
My PID = 6701: Child PID = 6702 has been stopped by a signal, signo = 19
PID = 6705, name C, starting...
My PID = 6701: Child PID = 6705 has been stopped by a signal, signo = 19
PID = 6706, name D, starting...
My PID = 6701: Child PID = 6706 has been stopped by a signal, signo = 19
My PID = 6700: Child PID = 6701 has been stopped by a signal, signo = 19
```

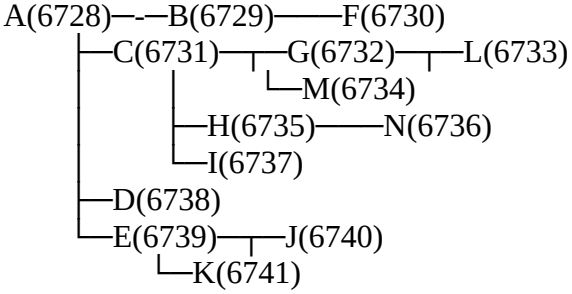
```
A(6701)─┬─B(6702)─┬─E(6703)
        │         └─F(6704)
        ├─C(6705)
        └─D(6706)
```

```
PID = 6701, name = A is awake
PID = 6702, name = B is awake
PID = 6703, name = E is awake
My PID = 6702: Child PID = 6703 terminated normally, exit status = 0
PID = 6704, name = F is awake
My PID = 6702: Child PID = 6704 terminated normally, exit status = 0
My PID = 6701: Child PID = 6702 terminated normally, exit status = 0
PID = 6705, name = C is awake
My PID = 6701: Child PID = 6705 terminated normally, exit status = 0
PID = 6706, name = D is awake
My PID = 6701: Child PID = 6706 terminated normally, exit status = 0
My PID = 6700: Child PID = 6701 terminated normally, exit status = 0
```

- example.tree

```
PID = 6728, name A, starting...
PID = 6729, name B, starting...
PID = 6730, name F, starting...
My PID = 6729: Child PID = 6730 has been stopped by a signal, signo = 19
My PID = 6728: Child PID = 6729 has been stopped by a signal, signo = 19
PID = 6731, name C, starting...
PID = 6732, name G, starting...
```

PID = 6733, name L, starting...
My PID = 6732: Child PID = 6733 has been stopped by a signal, signo = 19
PID = 6734, name M, starting...
My PID = 6732: Child PID = 6734 has been stopped by a signal, signo = 19
My PID = 6731: Child PID = 6732 has been stopped by a signal, signo = 19
PID = 6735, name H, starting...
PID = 6736, name N, starting...
My PID = 6735: Child PID = 6736 has been stopped by a signal, signo = 19
My PID = 6731: Child PID = 6735 has been stopped by a signal, signo = 19
PID = 6737, name I, starting...
My PID = 6731: Child PID = 6737 has been stopped by a signal, signo = 19
My PID = 6728: Child PID = 6731 has been stopped by a signal, signo = 19
PID = 6738, name D, starting...
My PID = 6728: Child PID = 6738 has been stopped by a signal, signo = 19
PID = 6739, name E, starting...
PID = 6740, name J, starting...
My PID = 6739: Child PID = 6740 has been stopped by a signal, signo = 19
PID = 6741, name K, starting...
My PID = 6739: Child PID = 6741 has been stopped by a signal, signo = 19
My PID = 6728: Child PID = 6739 has been stopped by a signal, signo = 19
My PID = 6727: Child PID = 6728 has been stopped by a signal, signo = 19



PID = 6728, name = A is awake
PID = 6729, name = B is awake
PID = 6730, name = F is awake
My PID = 6729: Child PID = 6730 terminated normally, exit status = 0
My PID = 6728: Child PID = 6729 terminated normally, exit status = 0
PID = 6731, name = C is awake
PID = 6732, name = G is awake
PID = 6733, name = L is awake
My PID = 6732: Child PID = 6733 terminated normally, exit status = 0
PID = 6734, name = M is awake
My PID = 6732: Child PID = 6734 terminated normally, exit status = 0
My PID = 6731: Child PID = 6732 terminated normally, exit status = 0
PID = 6735, name = H is awake
PID = 6736, name = N is awake
My PID = 6735: Child PID = 6736 terminated normally, exit status = 0
My PID = 6731: Child PID = 6735 terminated normally, exit status = 0

PID = 6737, name = I is awake
My PID = 6731: Child PID = 6737 terminated normally, exit status = 0
My PID = 6728: Child PID = 6731 terminated normally, exit status = 0
PID = 6738, name = D is awake
My PID = 6728: Child PID = 6738 terminated normally, exit status = 0
PID = 6739, name = E is awake
PID = 6740, name = J is awake
My PID = 6739: Child PID = 6740 terminated normally, exit status = 0
PID = 6741, name = K is awake
My PID = 6739: Child PID = 6741 terminated normally, exit status = 0
My PID = 6728: Child PID = 6739 terminated normally, exit status = 0
My PID = 6727: Child PID = 6728 terminated normally, exit status = 0

Ερωτήσεις:

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

   Στις προηγούμενες ασκήσες αδρανοποιούσαμε τα παιδιά-φύλλα ώστε να εξασφαλίσουμε χρόνο πριν "πεθάνουν" ώστε να προλάβουν να αποτυπωθούν από την pstree που καλείται στην συνάρτηση show_pstree(). Με την χρήση σημάτων αποφεύγουμε να ρυθμίζουμε αυθέραιτα τον χρόνο όπως κάναμε στην 1.2 με την sleep(), πετυχαίνοντας καλύτερο συγχρονισμό. Συγκεκριμένα μόλις "κοιμηθούν" με την *raise(SIGSTOP)* και αφού εμφανίσει το δέντρο στέλνει μήνυμα στα παιδιά με το kill(pid, SIGCONT) να ξυπνήσουν και να συνεχίσουν εώς ότου τερματιστούν.

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

   Η wait_for_ready_children() είναι μια συνάρτηση που αναστέλλει την λειτουργία της μέχρι τα παιδιά της να αδρανοποιηθούν. Ελέγχει ουσιαστικά τη σημαία WIFSTOPPED, που δείχνει αν το παιδί έχει σταματήσει και σε αντίθετη περίπτωση εμφανίζει μήνυμα για τον «ξαφνικό» θάνατο του παιδιού και κάνει exit(). Η χρήση της εξασφαλίζει πως όλα τα παιδιά είναι ζωντανά και έχουν σταματήσει, άρα το δέντρο διεργασιών θα εμφανιστεί σωστά και επίσης ότι θα έχουμε πλήρη γνώση αν κάποιο παιδί πέθανε πριν την ώρα του. Αν δε τη χρησιμοποιήσουμε, τότε ο πατέρας αφού ξυπνήσει θα περιμένει το θάνατο του παιδιού του, που έχει ήδη προηγηθεί, επομένως θα παραμένει αδρανή η διεργασία.

Άσκηση 1.4

Πηγαίος Κώδικας:

ask2-fork4.c

```
/*****************************************************************/
/*                          Operating Systems                    */
/*                          Second Lab Exercise                  */
/*                                                               */
/* Exercise 1.4 ask2fork4.c                                      */
/*                                                               */
/* Team: oslaba27                                                */
/*                                                               */
/* Full Name: Kerasiotis Ioannis, Student ID: 03114951           */
/* Full Name: Raftopoulos Evangelos, Student ID: 03114743        */
/*                                                               */
/*****************************************************************/

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

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

#define SLEEP_TREE_SEC  3
#define SLEEP_PROC_SEC 10

void fork_procs(struct tree_node *t, int parentPipe[]) {
        int i;

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

        if (t->nr_children == 0) {     // if it is leaf
                int value;

                sleep(SLEEP_PROC_SEC);  //sleep
                if (close(parentPipe[0]) < 0){
                        perror("Close");
                        exit(1);
                }
```

```
                value = atoi(t->name);    // convert chat to int
                if (write(parentPipe[1], &value, sizeof(int)) < 0){    //write and check if at pipe the value
                        perror("Write");
                        exit(1);
                }

                printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n", t->name,
(long)getpid(), t->name);
                exit(0);
        }
        else {
                int numbers[2];
                int myPipe[2];   //a pipe for reading and a pipe for writing
                if (pipe(myPipe)){
                        perror("Failed pipe");
                }

                int status;
                pid_t pid;

                for(i=0; i<2; i++){       // ask2-fork2.c make the proccess' tree recursively
                        pid = fork();
                        if (pid < 0) {
                                perror("proc: fork");
                                exit(1);
                        }
                        if (pid == 0) {
                                /* Child */
                                fork_procs(t->children+i, myPipe);
                                exit(1);
                        }
                }

                if (close(myPipe[1]) < 0){
                        perror("Close");
                        exit(1);
                }
                for (i=0; i<2; i++) {
                        if (read(myPipe[0], numbers+i, sizeof(int)) < 0){   //read nambers from pipe
                                perror("Read");
                                exit(1);
                        }
                }

                int result;
                if (!strcmp(t->name,"+")) result = numbers[0]+numbers[1];  // check operator
                else result = numbers[0]*numbers[1];                    // calculation
```

```
                if (write(parentPipe[1], &result, sizeof(int))<0){         // write result
                        perror("Write");
                        exit(1);
                }

                for (i=0; i<2; i++) {
                        waitpid(pid, &status, 0);
                        explain_wait_status(pid, status);
                }
        }
        printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n", t->name, (long)getpid(), t-
>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[]){
        pid_t pid;
        int status, pipe_fd[2];
        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);

        if (pipe(pipe_fd)) {
                perror("Failed pipe");
        }

        /* Fork root of process tree */
        pid = fork();
        if (pid < 0) {
```

```
                perror("main: fork");
                exit(1);
        }
        if (pid == 0) {
                /* Child */
                fork_procs(root, pipe_fd);
                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);

        int result;
        if (read(pipe_fd[0], &result, sizeof(int)) < 0){
                perror("Read");
                exit(1);
        }

        /* Wait for the root of the process tree to terminate */
        waitpid(pid, &status, 0);
        explain_wait_status(pid, status);

        printf("\nThe result is %d\n", result);

        return 0;
}
```

Έξοδος εκτέλεσης προγράμματος:

Στην άσκηση αυτή, παίρνουμε ως είσοδο ένα δέντρο που απεικονίζει μια μαθηματική έκφραση. Το αποτέλεσμα της πράξης εμφανίζεται ως έξοδος του προγράμματος στο stdout.

Συγκεκριμένα:

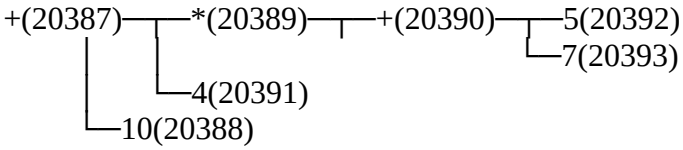
- expr.tree

Το δέντρο που δίνεται είναι το παρακάτω:

```
# file that defines the tree
# lines starting with '#' are comments
# . each block defines a node
# . each node is defined as:
#    1st line:         name of node
#    2nd line:         number of children
#    subsequent lines: name(s) of children
# . blocks are seperated with empty lines
# . no comments are allowed within a block
# . nodes must be placed in a DFS order

+
2
10
*

10
0

*
2
+
4

+
2
5
7

5
0

7
0

4
0
```

Και το αποτέλεσμα είναι το παρακάτω:

PID = 20387, name +, starting...
PID = 20388, name 10, starting...
PID = 20389, name *, starting...
PID = 20390, name +, starting...
PID = 20391, name 4, starting...
PID = 20392, name 5, starting...
PID = 20393, name 7, starting...



10 with PID = 20388 is ready to terminate...
10: Exiting...
4 with PID = 20391 is ready to terminate...
4: Exiting...
5 with PID = 20392 is ready to terminate...
5: Exiting...
7 with PID = 20393 is ready to terminate...
7: Exiting...
My PID = 20390: Child PID = 20393 terminated normally, exit status = 0
My PID = 20390: Child PID = 20393 terminated normally, exit status = 0
+ with PID = 20390 is ready to terminate...
+: Exiting...
My PID = 20389: Child PID = 20391 terminated normally, exit status = 0
My PID = 20389: Child PID = 20391 terminated normally, exit status = 0
* with PID = 20389 is ready to terminate...
*: Exiting...
My PID = 20387: Child PID = 20389 terminated normally, exit status = 0
My PID = 20387: Child PID = 20389 terminated normally, exit status = 0
+ with PID = 20387 is ready to terminate...
+: Exiting...
My PID = 20386: Child PID = 20387 terminated normally, exit status = 0

The result is 58

Το οποίο είναι λογικό αφού:

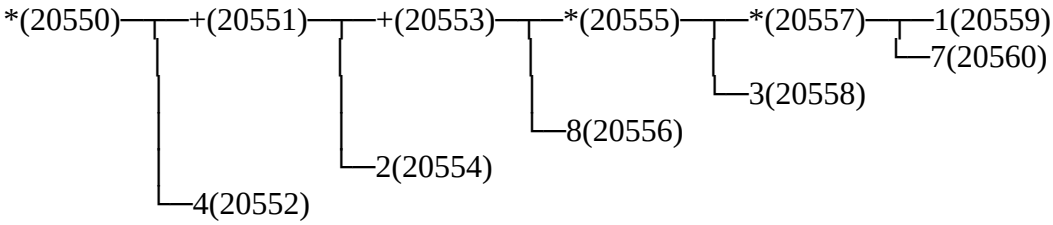
$$(10+(4\cdot(5+7)))=(10+(4\cdot 12))=(10+48)=58$$

- exp_example.tree

```
# file that defines the tree
# lines starting with '#' are comments
# . each block defines a node
# . each node is defined as:
#    1st line:         name of node
#    2nd line:         number of children
#    subsequent lines: name(s) of children
# . blocks are seperated with empty lines
# . no comments are allowed within a block
# . nodes must be placed in a DFS order

*
2
+
4

+
2
+
2

+
2
*
8

*
2
*
3

*
2
1
7

1
0

7
0

3
0

8
0

2
0

4
0
```

Με έξοδο:

```
PID = 20550, name *, starting...
PID = 20551, name +, starting...
PID = 20552, name 4, starting...
PID = 20553, name +, starting...
PID = 20554, name 2, starting...
PID = 20555, name *, starting...
PID = 20556, name 8, starting...
PID = 20557, name *, starting...
PID = 20558, name 3, starting...
PID = 20559, name 1, starting...
PID = 20560, name 7, starting...
```




```
4 with PID = 20552 is ready to terminate...
4: Exiting...
2 with PID = 20554 is ready to terminate...
2: Exiting...
8 with PID = 20556 is ready to terminate...
8: Exiting...
3 with PID = 20558 is ready to terminate...
3: Exiting...
1 with PID = 20559 is ready to terminate...
1: Exiting...
7 with PID = 20560 is ready to terminate...
7: Exiting...
My PID = 20555: Child PID = 20558 terminated normally, exit status = 0
My PID = 20555: Child PID = 20558 terminated normally, exit status = 0
* with PID = 20555 is ready to terminate...
*: Exiting...
My PID = 20553: Child PID = 20556 terminated normally, exit status = 0
My PID = 20557: Child PID = 20560 terminated normally, exit status = 0
My PID = 20553: Child PID = 20556 terminated normally, exit status = 0
My PID = 20557: Child PID = 20560 terminated normally, exit status = 0
+ with PID = 20553 is ready to terminate...
+: Exiting...
* with PID = 20557 is ready to terminate...
*: Exiting...
My PID = 20551: Child PID = 20554 terminated normally, exit status = 0
My PID = 20551: Child PID = 20554 terminated normally, exit status = 0
```

+ with PID = 20551 is ready to terminate...
+: Exiting...
My PID = 20550: Child PID = 20552 terminated normally, exit status = 0
My PID = 20550: Child PID = 20552 terminated normally, exit status = 0
* with PID = 20550 is ready to terminate...
*: Exiting...
My PID = 20549: Child PID = 20550 terminated normally, exit status = 0

The result is 124

Επίσης λογικό διότι:

$$(4\cdot(2+(8+(3\cdot(1\cdot7)))))=(4\cdot(2+(8+(3\cdot7))))=(4\cdot(2+(8+21)))=(4\cdot(2+29))=(4\cdot31)=124$$

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 άσκηση γίνεται η χρήση μία σωλήνωσης. Αυτό γίνεται για τον λόγο ότι το δέντρο διασχίζεται κατάβάθος και φτάνονταςστα φύλλα με το μεγαλύτερο βάθος αρχίζουν να γίνονται οι πράξεις της πρόσθεσης και του πολλαπλασιασμούς, που είναι αντιμεταθετικές πράξεις. Στην περίπτωση που γινόντουσαν πράξεις όπως αφαίρεση και διαίρεση που δεν ισχύει η αντιμεταθετική ιδιότητα θα χρειαζόντουσαν τουλάχιστον δύο pipes.

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 τέτοιο σύστημα οι διεργασίες θα εκτελούνταν παράλληλα, που σημαίνει πως στην περίπτωση που έχουμε 4 επεξεργαστές, στην πρώτη είσοδο expr.tree θα εκτελούνταν στον πρώτο πηρήνα θα γινόταν η πράξη (5+7), ενώ στο δεύτερο 4(5+4) και στο τρίτο η πράξη 10+4(5+7), κερδίζοντας περισσότερο από τον μισό χρόνο.