Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2019-2020

Άσκηση 2: Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

Στοιχεία σπουδαστών:

Όνομα: Μαντζούτας Ανδρέας Α.Μ. : 03117108

Όνομα: Τσιτσής Αντώνιος Α.Μ. : 03117045

Ομάδα εργαστηρίου: Oslabc23

Εξάμηνο: 6^{ο}

Ημερομηνία: 02/06/2020

Ασκήσεις

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Για την υλοποίηση αυτής της άσκησης χρησιμοποιήσαμε το βοηθητικό αρχείο ask2-fork.c που μας δόθηκε, καθώς και τις συναρτήσεις που του αρχείου proc-common.c.

Πιο συγκεκριμένα, σκοπός της άσκησης ήταν η δημιουργία και η «φωτογράφηση» ενός δέντρου διεργασιών με 4 κόμβους. Για να καταφέρουμε να λάβουμε το στιγμιότυπο του δέντρου χρησιμοποιήθηκε η κλήση συστήματος sleep(), ώστε τα φύλλα να μην πεθάνουν για ένα προκαθορισμένο χρονικό διάστημα. Έπειτα, χρησιμοποιήσαμε τη συνάρτηση show_pstree() για να «φωτογραφήσουμε» και να εκτυπώσουμε το δέντρο. Τέλος, χρησιμοποιήσαμε διαφορετικούς κωδικούς εξόδου σε κάθε κόμβο, ώστε να είμαστε σίγουροι ότι το πρόγραμμα εκτελέστηκε σωστά.

Παρακάτω παρατίθεται ο κώδικας που υλοποιήσαμε, καθώς και ένα παράδειγμα εκτέλεσης του κώδικα:

```
#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

const char* A = "A";
const char* B = "B";
const char* C = "C";
const char* D = "D";

void fork_procs(const char *x)
{
	/*
	 * initial process is A.
	 */
	if (x == A){
	change_pname("A");
	printf("A: Sleeping...\n");
	sleep(SLEEP_PROC_SEC);
	}

	if (x  == B){
	change_pname("B");
	printf("B: Sleeping...\n");
	sleep(SLEEP_PROC_SEC);
	}

	if (x == C) {
	change_pname("C");
	printf("C: Sleeping...\n");
```

```
        sleep(SLEEP_PROC_SEC);

        printf("C: Exiting...\n");
        exit(17);
        }

        if (x == D) {
        change_pname("D");
        printf("D: Sleeping...\n");
        sleep(SLEEP_PROC_SEC);

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

int main(void)
{
        pid_t pid;
        int status;

        /* Fork root of process tree */
        pid = fork();
        if (pid < 0) {
                perror("main: fork");
        }
        else if (pid == 0) {
                /* Child */
                pid = fork();
                if (pid < 0){
                        perror("main: fork");
                }
                else if (pid ==0) {
                        //create D
                        pid = fork();
                        if (pid < 0){
                                perror("main: fork");
                        }
                        else if (pid==0){
                                fork_procs("D");
                        }
                        else{   fork_procs("B");
                                pid = wait(&status);
                                explain_wait_status(pid,status);//We need B to wait for it's child

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

                }
                else {
                        pid = fork();//C
                        if (pid < 0) {
                        perror("main: fork");
                        }
                        else if (pid == 0) {
                                fork_procs("C");
                        }
                        else {
```

```
                        fork_procs("A");
                        pid = wait(&status);
                        explain_wait_status(pid,status);
                        pid = wait(&status);
                        explain_wait_status(pid,status);//We need A to wait for both B and C

                        printf("A: Exiting...\n");
                        exit(16);
                }
        }
}
else {
/*
 * 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);
        printf("\n");
        show_pstree(getpid());

/* Wait for the root of the process tree to terminate */
pid = wait(&status);
explain_wait_status(pid, status);
}
return 0;
}
```

```
A: Sleeping...
C: Sleeping...
B: Sleeping...
D: Sleeping...


A(22217)─┬─B(22218)──D(22220)
         └─C(22219)


C: Exiting...
D: Exiting...
My PID = 22217: Child PID = 22219 terminated normally, exit status = 17
My PID = 22218: Child PID = 22220 terminated normally, exit status = 13
B: Exiting...
My PID = 22217: Child PID = 22218 terminated normally, exit status = 19
A: Exiting...
My PID = 22216: Child PID = 22217 terminated normally, exit status = 16
oslabc23@os-node1:~/andr/2$
```

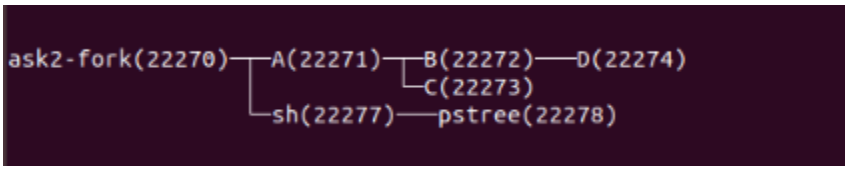
Ερωτήσεις:

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

Αν τερματίσουμε πρόωρα τη διεργασία Α, στην περίπτωση που οι διεργασίες-παιδία δεν έχουν πεθάνει ακόμα, τότε αυτές θα ανατεθούν στην εργασία init, την κεντρική διεργασία του συστήματος.

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

Κάνοντας αυτήν την αλλαγή ξανατρέξαμε το πρόγραμμα και λάβαμε την εξής έξοδο:



Παρατηρούμε ότι αν δώσουμε στην συνάρτηση show_pstree το getpid(), αντί για pid, δημιουργείται ένα νέο δέντρο, οπού, όπως φαίνεται, η διεργασία-ρίζα του δέντρου είναι η main μας. Τέλος, υπάρχει ένα ακόμα υπόδεντρο, το οποίο περιέχει διεργασίες που χρησιμοποιούνται για την εμφάνιση του δέντρου.

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

Σε ένα υπολογιστικό σύστημα, ο διαθέσιμος χρόνος διαμοιράζεται ισόποσα σε όλες τις διεργασίες. Αυτό σημαίνει, ότι αν οι διεργασίες γίνουν πάρα πολλές, υπάρχει ο κίνδυνος να καταρρεύσει το σύστημα, λόγω των μεγάλων απαιτήσεων. Επομένως, για λόγους ασφαλείας, σε υπολογιστικά συστήματα πολλαπλών χρηστών, ο διαχειριστής θέτει όρια στο πλήθος των διεργασιών ανά χρήστη.

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Σε αυτήν την άσκηση κληθήκαμε να δημιουργήσουμε ένα αυθαίρετο δέντρο διεργασιών. Για αυτό το λόγο χρησιμοποιήσαμε μία αναδρομική συνάρτηση που καλούταν για κάθε κόμβο του δέντρου, δημιουργούσε τις διεργασίες-παιδία του κόμβου και περίμενε μέχρι να πεθάνουν τα παιδία του ή σε περίπτωση που ήταν φύλλο, αδρανοποιούταν με την κλήση sleep() για κάποιο χρονικό διάστημα. Ουσιαστικά, χρησιμοποιήσαμε δύο περιπτώσεις:

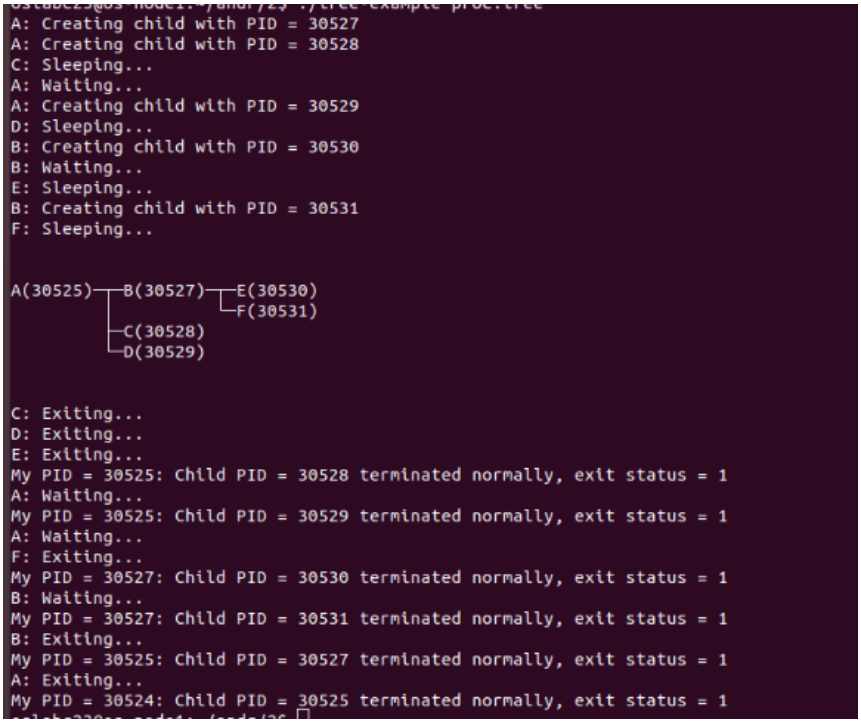
- Στην περίπτωση που ο κόμβος είναι φύλλο, δηλ. δεν έχει παιδία, θα εκτελεί την κλήση συστήματος sleep για κάποιο χρονικό διάστημα
- Στην περίπτωση που έχει παιδιά, θα κάνει fork() για να δημιουργήσει τα παιδία του με ένα for loop, θα περιμένει να πεθάνουν όλα(wait()) και έπειτα θα τερματιστεί.

Επομένως, η διαδικασία που ακολουθήσαμε ήταν πρώτα να δημιουργήσουμε το root του δέντρου και έπειτα να καλέσουμε την συνάρτηση που αναφέραμε για να δημιουργήσει το υπόλοιπο δέντρο.

Παρατίθεται ο κώδικας της άσκησης και ένα παράδειγμα εκτέλεσης με είσοδο του αρχείο proc.tree που μας δόθηκε.

```
#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_PROC_SEC  10
#define SLEEP_TREE_SEC  3

void fork_procs( struct tree_node *root){

	change_pname(root->name);
	int status;
	int i=0;

	if (root->nr_children ==0){ //If it doesn't have children case
		printf("%s: Sleeping...\n", root->name);
		sleep (SLEEP_PROC_SEC);

		printf("%s: Exiting...\n", root->name);
		exit(1);
	}
```

```
        else{ //        Case it has children
                pid_t pid;
                for (i = 0; i < root->nr_children; i++) { //For each child...
                        pid = fork();
                        if (pid < 0){
                                perror("procs: fork");
                                exit(1);
                        }
                        if (pid==0) {
                                printf("%s: Creating child with PID = %ld\n", root->name, (long)getpid());
                                fork_procs(root->children+i);
                                exit(1);
                        }
                }

                for (i = 0; i < root->nr_children; i++){
                        printf("%s: Waiting...\n", root->name);
                       pid = wait(&status);
                        explain_wait_status(pid,status);
                }
                printf("%s: Exiting...\n", root->name);
        }
}

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

        root = get_tree_from_file(argv[1]);
        pid= fork();//make root

        if (pid< 0) {
                perror("main: fork");
        }

        else if (pid == 0) {
                fork_procs(root);
                exit(1);
        }

        show_pstree(pid);
        pid = wait(&status);
        explain_wait_status(pid, status);
        return 0;
}
```



Ερωτήσεις:

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

Κατά τη διάρκεια των δοκιμών, «τρέξαμε» τον κώδικα περισσότερες από μία φορές. Παρατηρήσαμε ότι τα μηνύματα δεν εμφανίζονταν στην ίδια θέση πάντα. Αυτό σημαίνει ότι υπάρχει μία «τυχαιότητα» στην εμφάνιση των μηνυμάτων,
Αυτό ήταν αναμενόμενο εν μέρει, γιατί οι διεργασίες δεν δημιουργούνται ή τερματίζονται με προκαθορισμένη σειρά, αλλά εξαρτούνται από παράγοντες όπως η ύπαρξη ή μη παιδιών, ο αριθμός των παιδιών(, αφού κάθε διεργασία περιμένει να τερματιστούν τα παιδιά της, πριν τερματιστεί η ίδια). Επομένως, επειδή εμείς δεν έχουμε παρέμβει με κάποιο τρόπο στη σειρά έναρξης ή τερματισμού των διεργασιών-κόμβων, αυτή είναι τυχαία, άρα και η εμφάνιση των μηνυμάτων είναι τυχαία.

1.3 Αποστολή και χειρισμός σημάτων

Σε αυτήν την άσκηση, υλοποιήσαμε το ίδιο ζητούμενο με την προηγούμενη. Αυτή τη φορά όμως, θέλαμε να εξασφαλίσουμε ότι δεν θα είναι τυχαία η σειρά δημιουργίας και τερματισμού των διεργασιών. Ουσιαστικά, θέλαμε να ορίσουμε το δέντρο να δημιουργείται με τη σειρά μιας DFS διάσχισης. Για την επίτευξη αυτού, κάθε φορά που φτάναμε σε κάποιο φύλλο, το φύλλο έκανε raise SIGSTOP, και ανέστελλε τη λειτουργία του έως ότου λάβει σήμα SIGCONT. Το ίδιο κάνουν και διαδοχικά οι «γονείς», αφού δημιουργήσουν τα παιδιά τους. Έτσι, στέλνονται διαδοχικά σήματα SIGCONT από κάθε γονιό στο παιδί του, ώστε να «ξυπνήσουν» και να τερματιστούν.
Για να στείλουμε τα διαδοχικά σήματα SIGCONT, διατηρούμε σε κάθε κόμβο-γονέα έναν πίνακα από PID, ο οποίος περιέχει τα pid των κόμβων που καλείται να στείλει SIGCONT- δηλαδή των παιδιών του.
Παραθέτουμε τον κώδικα και η εκτέλεσή του με είσοδο το proc.tree.

```
#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){

	change_pname(root->name); //gia na allaksoume to id ka8e diadikasias sto epi8ymhto
//onoma
	int i;
	int status;
	pid_t pid2[root->nr_children]; //hold the pid's
	if (root->nr_children == 0){ //if the node has no children
		printf("Process %s started.",root->name);
		printf("Process %s: Waiting for SIGCONT\n",root->name);
		raise(SIGSTOP);
		printf("Process %s: Exiting\n",root->name);
		exit(0);
	}
	else{
		//if the node has children
		pid_t pid;
		printf("Process %s started creating children\n",root->name);
		for (i=0;i<root->nr_children;i++){
			pid=fork();
			 if (pid<0){
				perror("fork_procs: fork\n");
				exit(1);
```

```
                }
                if (pid == 0){
                        fork_procs(root->children+i);
                        printf("Process %s : Exiting\n",root->name);
                        exit(0);
                }
                else{
                        pid2[i]=pid;
                }
                wait_for_ready_children(1); //waits for every child
        }

        printf("PID= %ld, name = %s is waiting for SIGCONT\n",(long)getpid(),root->name);
        raise(SIGSTOP);
        printf("PID = %ld, name = %s is awake\n",
        (long)getpid(), root->name);
        //Send SIGCONT to every different pid in the pid2 table
        for(i=0;i<root->nr_children;i++){
                kill(pid2[i],SIGCONT);
                wait(&status);
                explain_wait_status(pid2[i],status);
        }
        exit(0);
    }
}

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]);

    /* Fork root of process tree */
    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
        /* Child */
        /* root of my tree */
        /*for every child I enter this if and
        decide in let_it_fork if it is a father of another child
        to fork again until I reach leaves*/
```

```
        fork_procs(root);
        exit(0);
    }
    /*
     * 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);
    kill(pid, SIGCONT);
    /* Wait for the root of the process tree to terminate */
    wait(&status);
    explain_wait_status(pid, status);
    return 0;
}
```

```
Process A started creating children
Process B started creating children
Process E started.Process E: Waiting for SIGCONT
My PID = 30666: Child PID = 30667 has been stopped by a signal, signo = 19
Process F started.Process F: Waiting for SIGCONT
My PID = 30666: Child PID = 30668 has been stopped by a signal, signo = 19
PID= 30666, name = B is waiting for SIGCONT
My PID = 30665: Child PID = 30666 has been stopped by a signal, signo = 19
Process C started.Process C: Waiting for SIGCONT
My PID = 30665: Child PID = 30669 has been stopped by a signal, signo = 19
Process D started.Process D: Waiting for SIGCONT
My PID = 30665: Child PID = 30670 has been stopped by a signal, signo = 19
PID= 30665, name = A is waiting for SIGCONT
My PID = 30664: Child PID = 30665 has been stopped by a signal, signo = 19


A(30665)─┬─B(30666)─┬─E(30667)
         │          └─F(30668)
         ├─C(30669)
         └─D(30670)


PID = 30665, name = A is awake
PID = 30666, name = B is awake
Process E: Exiting
My PID = 30666: Child PID = 30667 terminated normally, exit status = 0
Process F: Exiting
My PID = 30666: Child PID = 30668 terminated normally, exit status = 0
My PID = 30665: Child PID = 30666 terminated normally, exit status = 0
Process C: Exiting
My PID = 30665: Child PID = 30669 terminated normally, exit status = 0
Process D: Exiting
My PID = 30665: Child PID = 30670 terminated normally, exit status = 0
My PID = 30664: Child PID = 30665 terminated normally, exit status = 0
```

Ερωτήσεις:

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

Η χρήση της εντολής sleep() μέχρι τώρα ήταν απαραίτητη για τον συγχρονισμό των διεργασιών, ώστε να μένουν σε αναστολή για λίγη ώρα και να έχουμε καλύτερο έλεγχο της σειράς που ξυπνάνε οι κόμβοι.

Ωστόσο, τώρα που χειριστήκαμε το ζήτημα με χρήση σημάτων, αυτό δεν ήταν απαραίτητο, καθώς κάθε διεργασία έμπαινε σε αναστολή μέχρι να λάβει το κατάλληλο σήμα αφύπνισης. Επομένως, καταλήγουμε στο συμπέρασμα ότι το πλεονέκτημα των σημάτων είναι ότι τα πάντα γίνονται με προκαθορισμένη σειρά και απαλείφεται η τυχαιότητα.

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

Η συνάρτηση αυτή περιμένει έως ότου ο αριθμός των παιδιών του ορίσματος κάνει raise SIGSTOP, κι έπειτα εκτελείται η επόμενη εντολή. Εμείς την καλούμε με όρισμα 1, καθώς την καλούμε ξεχωριστά για κάθε παιδί, μέσω του πίνακα αποθήκευσης των PID των παιδιών που χρησιμοποιήσαμε. Με αυτόν τον τρόπο εξασφαλίζεται η σωστή λειτουργία της DFS διάσχισης. Αν την παραλείπαμε, θα υπήρχε ο κίνδυνος κάποια διεργασία γονιός να τερματιζόταν πριν τις αντίστοιχες διεργασίες-παιδιά, με αποτέλεσμα να μην εμφανιζόταν σωστά το δέντρο διεργασιών.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Σε αυτήν την άσκηση κληθήκαμε να υπολογίσουμε μια αριθμητική έκφραση, κάνοντας χρήση των pipes. Για να επιτευχθεί αυτό, χρησιμοποιήσαμε για κάθε κόμβο ένα νέο pipe, ώστε να επικοινωνεί με τον πατέρα του. Όταν φτάναμε σε κόμβους χωρίς παιδιά, που περιείχαν σίγουρα αριθμό, γράφαμε την τιμή στο pipe, και από αυτό θα το διάβαζε ο πατέρας, και θα αποτιμούσε την έκφραση. Επομένως, καθένα απ' τα δύο φύλλα έστελνε τον ένα εκ των 2 αριθμών, και ο πατέρας υπολόγιζε το αποτέλεσμα, και το «έστελνε» στον δικό του πατέρα, έως ότου φτάσουμε στο root και πάρουμε το συνολικό αποτέλεσμα.

Παραθέτουμε τον κώδικα και την έξοδο του προγράμματος, με είσοδο το expr.tree που μας δώθηκε.

```
#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_PROC_SEC 10

void fork_procs(struct tree_node *root, int fd)
{

	change_pname(root->name);

	//if its a leaf/number
	if(root->nr_children == 0) {
		int num = atoi(root->name);
		if ( write(fd, &num, sizeof(num) ) != sizeof(num) ) {
			perror("Pipe write Failed\n");
			exit(1);
		}
		printf("PID: %ld, %s: Wrote value: %d and exiting..\n", (long)getpid(),root-> name,
num);
		printf("%s: Sleeping...\n", root->name);
		sleep(SLEEP_PROC_SEC);
		printf("%s: Exiting...\n", root->name);
		exit(1);
	}
	else {

		//if its NOT a leaf
		pid_t pid1, pid2;
		int fd2[2];
```

```
if( pipe(fd2) < 0 ) {
        perror(" pipe");
        exit(1);
}

pid1 = fork();
if (pid1 < 0){
        perror("p1 fork");
        exit(1);
}
else if (pid1==0) {
        printf("%s: Created child  with PID=%ld\n", root->name, (long)getpid());
        fork_procs(root->children, fd2[1]);
}

pid2 = fork();
if (pid2 < 0){
        perror("p2 fork");
        exit(1);
}

else if (pid2 ==0){
        printf("%s: Created child with PID=%ld\n", root->name, (long)getpid());
        fork_procs(root->children + 1, fd2[1]);
}

int num1, num2;

int result;

if (read (fd2[0], &num1, sizeof(num1))!=sizeof(num1)) {
        perror("Read from pipe");
        exit(1);
}


if (read (fd2[0], &num2, sizeof(num2))!=sizeof(num2)) {
        perror("Read from pipe");
        exit(1);
}


if (strcmp(root->name,"+") == 0){
        result = num1 + num2;
}
else if (strcmp(root->name, "*") ==0) {
        result = num1 * num2;
}


if ( write(fd, &result, sizeof(result) ) != sizeof(result) ) {
        perror("Pipe Write");
```

```
                exit(1);
            }

             printf("Wrote %d to the pipe!\n", result);

            sleep(SLEEP_PROC_SEC);
            exit(1);
        }
}

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

        int fd[2];

        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]);

        if( pipe(fd) < 0 ){
            perror("main: pipe");
            exit(1);
        }

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

        if (pid == 0) {
            /* Child */
            change_pname(root->name);
            printf("%s: Created\n", root->name);
            close(fd[0]);
            fork_procs(root, fd[1]);
            exit(1);
        }


        sleep(SLEEP_TREE_SEC);


        show_pstree(pid);
```

```
    int result = 0;

    if ( read(fd[0], &result, sizeof(result) ) != sizeof(result) ){
        perror("main: read");
        exit(10);
    }


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

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

    return 0;
}
```

```
+: Created
+: Created child  with PID=6342
PID: 6342, 10: Wrote value: 10 and exiting..
+: Created child with PID=6343
10: Sleeping...
*: Created child  with PID=6344
*: Created child with PID=6345
PID: 6345, 4: Wrote value: 4 and exiting..
4: Sleeping...
+: Created child  with PID=6346
+: Created child with PID=6347
PID: 6346, 5: Wrote value: 5 and exiting..
5: Sleeping...
PID: 6347, 7: Wrote value: 7 and exiting..
7: Sleeping...
Wrote 12 to the pipe!
Wrote 48 to the pipe!
Wrote 58 to the pipe!


+(6341)─┬─*(6343)─┬─+(6344)─┬─5(6346)
        │         │         └─7(6347)
        │         └─4(6345)
        └─10(6342)


10: Exiting...
4: Exiting...
5: Exiting...
7: Exiting...
My PID = 6340: Child PID = 6341 terminated normally, exit status = 1
Total result is: 58
```

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;

Στη συγκεκριμένη άσκηση, σε περίπτωση ενδιάμεσου κόμβου χρησιμοποιήσαμε 2 σωληνώσεις μία που επικοινωνεί με τον πατέρα και μία σωλήνωση με τα παιδία. Στην περίπτωση φύλλου, είχαμε μόνο 1 σωλήνωση προφανώς. Κατά γενικό κανόνα, θα χρησιμοποιούσαμε 3 σωληνώσεις για κάθε ενδιάμεσο κόμβο. Αυτό στο ζήτημα μας δεν ήταν απαραίτητο ωστόσο, επειδή οι πράξεις του πολλαπλασιασμού και της πρόσθεσης είναι αντιμεταθετικές, γεγονός που σημαίνει ότι δεν μας επηρεάζει η σειρά με την οποία διαβάζουμε τους όρους απ' τη σωλήνωση. Στην περίπτωση διαίρεσης ή αφαίρεσης, βέβαια, δεν θα μπορούσε να επιτευχθεί αυτή η υλοποίηση.

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;

Στην περίπτωση ενός συστήματος πολλών επεξεργαστών, θα μπορούσε να ανατεθεί η κάθε αποτίμηση σε διαφορετικό επεξεργαστή. Αυτό δεν θα είχε ουσιαστικό κέρδος στην περίπτωση ενός μικρού δέντρου, όπως το δικό μας, ωστόσο σε μεγάλες εκφράσεις το κέρδος χρόνου θα ήταν τεράστιο.