Μάθημα: Λειτουργικά Συστήματα, Εαρινό 2020-21, 2η Σειρά Ασκήσεων

1η Άσκηση:

Πρόγραμμα:

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

#define SLEEP_TREE_SEC 10
#define SLEEP_TIME 20

void fork_procs(void)
{
	change_pname("A");
	printf("I am process A (%d) with parent %d\n",getpid(),getppid());

	int status;
	pid_t b_pid = fork();  //fork a child
	if (b_pid < 0) {
			perror("B_pid: fork");
			exit(1);
	}
	if (b_pid == 0)
	{
			change_pname("B");
			printf("I am process B (%d) and my parent is %d\n" ,
getpid(),getppid());
			pid_t d_pid = fork(); //fork a child
			if (d_pid<0){
				perror("D_pid: fork");
				exit(1);
			}
			if (d_pid ==0)
			{
				change_pname("D");
				printf("I am process D (%d) and my parent is
%d\n",getpid(),getppid());
```

```
                        sleep(SLEEP_TIME);
                        exit(13);
                }
                printf("B: is waiting for his child D ...\n");
                int w = wait(&status);
                explain_wait_status(w,status);
                exit(19);
        }
        pid_t c_pid = fork();
        if (c_pid < 0){
                perror("C_pid: fork");
                exit(1);
        }
        if(c_pid == 0) {
                change_pname("C");
printf("I am process C (%d) and my parent is %d\n.",getpid(),getppid());
                sleep(SLEEP_TIME);
                exit(17);
        }
        printf("A is waiting for his children B and C\n");
        int p = wait(&status);
        explain_wait_status(p,status);
        exit(16);
}
int main()
{
        pid_t pid;
        int status;

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

        show_pstree(pid);
```

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

        return 0;
}
```

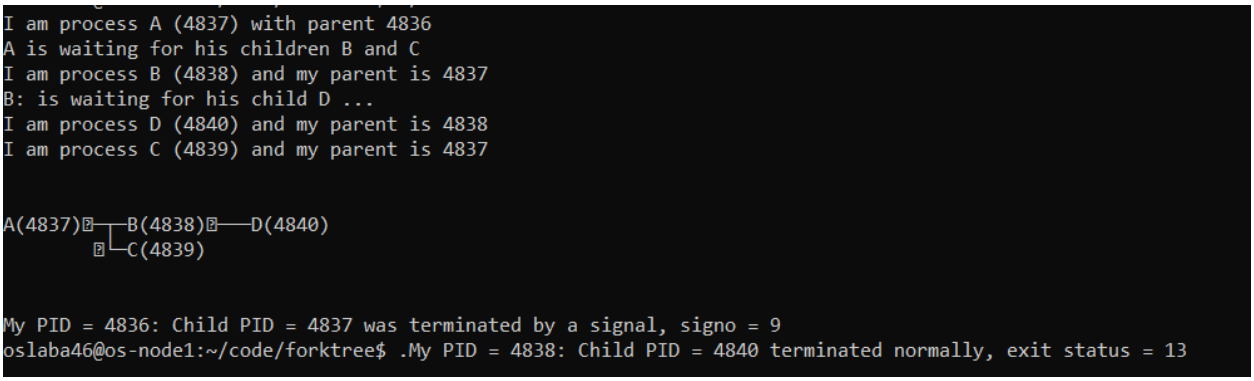
Η έξοδος του προγράμματος είναι η ακόλουθη:

```
I am process A (1385) with parent 1384
I am process B (1386) and my parent is 1385
A is waiting for his children B and C
I am process C (1387) and my parent is 1385
B: is waiting for his child D ...
I am process D (1388) and my parent is 1386


A(1385)─┬─B(1386)──D(1388)
        └─C(1387)


.My PID = 1386: Child PID = 1388 terminated normally, exit status = 13
My PID = 1385: Child PID = 1387 terminated normally, exit status = 17
My PID = 1385: Child PID = 1386 terminated normally, exit status = 19
My PID = 1384: Child PID = 1385 terminated normally, exit status = 16
```

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

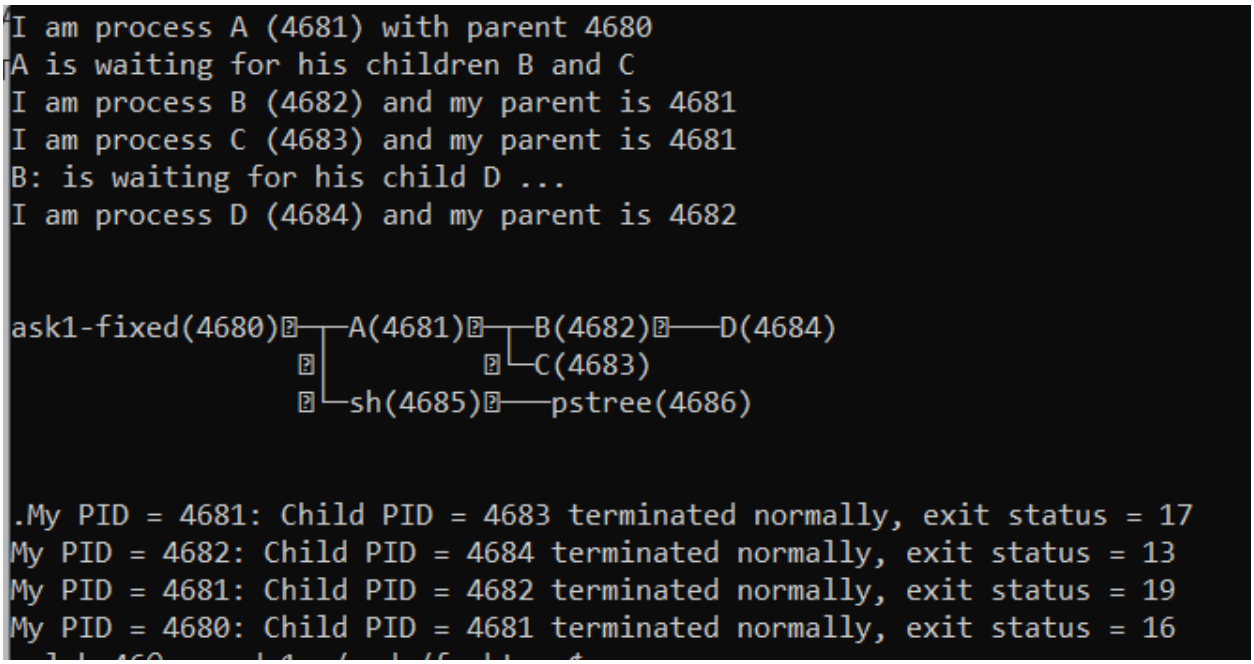
Όταν μια διεργασία πατέρας πεθαίνει πρόωρα είτε εσκεμμένα είτε όχι, πριν από την διεργασία παιδί, τότε η διεργασία παιδί γίνεται orphan, επειδή είναι running process, του οποίου ο πατέρας έχει τερματίσει. Αυτή η διεργασία παιδί, θα υιοθετηθεί αμέσως από την init, την αρχική διεργασία, δηλαδή τον πατέρα όλων. Η διεργασία init εφαρμόζει επαναλαμβανόμενα wait(), διασφαλίζοντας ότι θα τερματίσουν έτσι όλες οι διεργασίες, αλλά και ότι θα ελευθερωθεί και ο χώρος που αυτή καταλαμβάνει.
Δίνοντας την εντολή kill -KILL <pid> (Process ID της A), μέσω ενός δεύτερου terminal, τερματίζεται η διεργασία A. Ενημερωνόμαστε μέσω ενός μηνύματος ότι η διεργασία A, έχει τερματιστεί μέσω του σήματος με κωδικό 9, ο οποίος αντιστοιχεί στο σήμα kill. Βλέπουμε επίσης ότι οι διεργασίες B και D τερματίζουν κανονικά, ως παιδιά της init , και εμφανίζονται και τα αναγνωριστικά μηνύματα για την έξοδο του

D. Οι διεργασίες B και C τερματίζουν και αυτές κανονικά, αλλά δεν εμφανίζονται τα αναγνωριστικά τους μηνύματα για τον τερματισμό τους, καθώς η εντολή για να εμφανιστούν μέσω της ρουτίνας explain_wait_status, βρίσκεται μέσα στην διεργασία A, η οποία έχει τερματιστεί πρόωρα.



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

Η συνάρτηση show_pstree(node) εμφανίζει ουσιαστικά το δέντρο διεργασιών από τον κόμβο, που της αναθέτουμε στο όρισμα της. Αν σαν όρισμα, θέσουμε το pid, δηλαδή το process id της διεργασίας που μόλις κλονοποιήσαμε, της A, τότε θα πάρουμε το δέντρο με ρίζα τον κόμβο A. Αν όμως στην show_pstree θέσουμε ως όρισμα την συνάρτηση getpid(), αυτή θα έχει ως αποτέλεσμα το process id της διεργασίας που βρισκόμαστε τώρα, δηλαδή της διεργασίας main της ask2-fork και όχι του κλώνου A που δημιουργήσαμε. Αυτό, δηλαδή η επιλογή του ορίσματος getpid() στην συνάρτηση show_pstree(), θα έχει ως αποτέλεσμα να εμφανιστεί ένα δέντρο διεργασιών με ρίζα, την διεργασία που βρισκόμαστε τώρα. Θα εμφανίζονται όμως πέρα από τον κόμβο A ως παιδιά της τρέχουσας διεργασίας και κάποιες άλλες διεργασίες, μεταξύ των οποίων και η show_pstree. Αυτό συμβαίνει, επειδή το πρόγραμμα μας καλεί και άλλες διεργασίες, επομένως εμφανίζονται και αυτές ως παιδιά.



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

Πολλές φορές, για να μην επιβαρύνουμε το λειτουργικό σύστημα, μπορούμε να θέτουμε ένα άνω όριο στο πλήθος των διεργασιών που μπορεί κανείς να εκτελεί ταυτόχρονα.
Για να μην χρησιμοποιούμε ανεξέλεγκτα όλους τους πόρους του υπολογιστή
Τα λογισμικά τύπου Unix δίνουν την δυνατότητα να υπάρχει έλεγχος στο πόσες διεργασίες αντιστοιχούν σε έναν χρήστη και έτσι υπάρχει μεγαλύτερος έλεγχος των πόρων που καταναλώνονται. Σε κάθε περίπτωση όταν υπάρχουν τέτοια περιβάλλοντα, όπου πολλοί χρήστες "συνυπάρχουν", συνεπάγεται ότι θα είναι μεγάλος ο κίνδυνος, κάποιος χρήστης, να κάνει κάποιο σοβαρό λάθος και έτσι να "πέσει" ακόμα και όλο το σύστημα. Αυτό μπορεί να είναι κάποιο "fork-bomb", το οποίο δημιουργεί πάρα πολλές διεργασίες, με αποτέλεσμα να υπερφορτώσει το σύστημα και συχνά να το καταστήσει μη αποκρίσιμο.

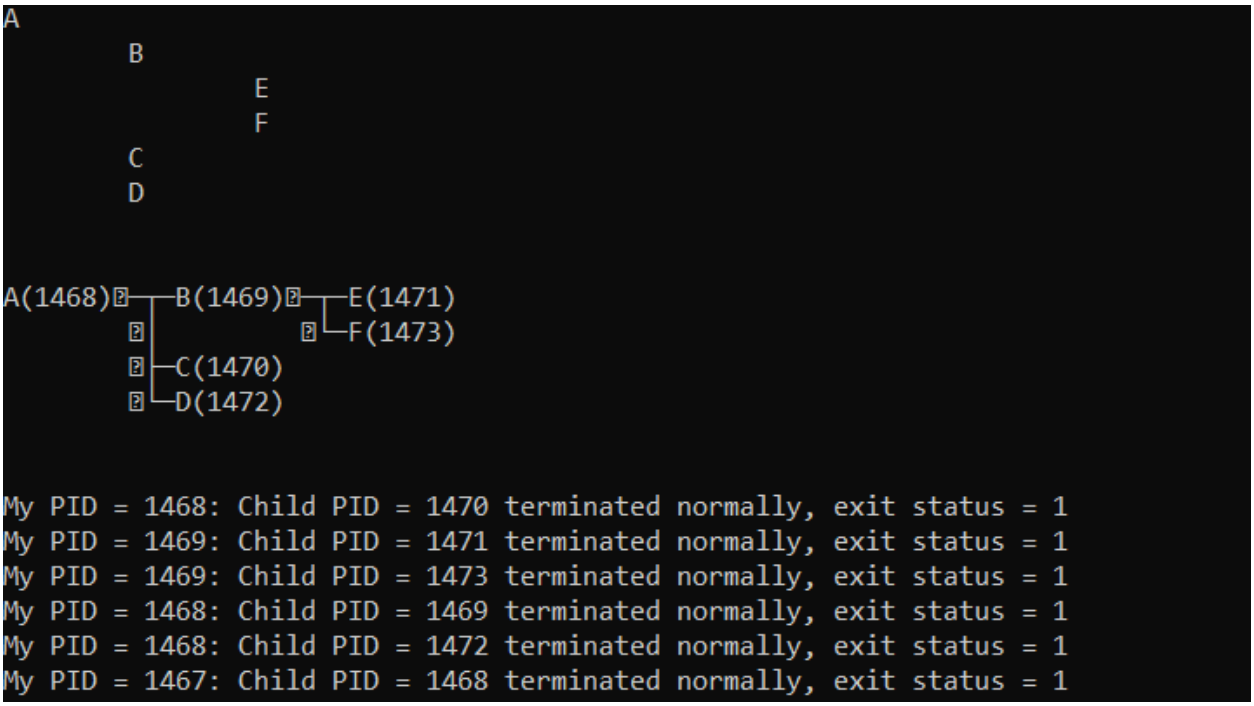
2η άσκηση:

Πρόγραμμα:

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

#define SLEEP_TREE_SEC 10
#define SLEEP_PROC_SEC 20

void child(struct tree_node *root){
	int status;
	change_pname(root->name);
	//printf("I am process: %d\n",getpid());
	if (root -> nr_children == 0){
			sleep(SLEEP_PROC_SEC);
			exit(1);
	}
	else {
			int i;
			for (i=0; i < (root->nr_children) ; i++)
			{
				pid_t n_pid = fork();
				if(n_pid <0)
				{
					perror("Child:fork");
					exit(1);
				}
				if (n_pid == 0)
				{
					child(root->children+i);
				}
			}
			int p,j;
			for (j=0;j< root -> nr_children;j++){
				p = wait(&status);
				explain_wait_status(p,status);
			}
			exit(1);
```

```
}
}

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

	int status;
	pid_t pid = fork();

	if (pid < 0) {
		perror("Root:fork");
		exit(1);
	}
	if (pid ==0) {
		child(root);
		exit(1);
	}

	sleep(SLEEP_TREE_SEC);

	show_pstree(pid);

	int w;
	w = wait(&status);
	explain_wait_status(w,status);

	return 0;
}
```

Η έξοδος του προγράμματος είναι η ακόλουθη:



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

Σε ότι αφορά τα μηνύματα έναρξης, ξεκινάει πρώτα η αρχική διεργασία, ο Α, και φυσικά δεν υπάρχει περίπτωση να δημιουργηθεί κάποιο παιδί χωρίς να δημιουργηθεί ο πατέρας, και δεν γίνεται να τερματίσει κάποιος πατέρας χωρίς να έχουν τερματίσει τα παιδιά του - στην συγκεκριμένη περίπτωση, αφού έχουμε χρησιμοποιήσει την συνάρτηση wait(), αλλιώς σε κάποια άλλη περίπτωση είναι δυνατό να τερματίσει ο πατέρας και κάποιο παιδί να μην τερματίσει και τότε αναλαμβάνει η init.
Τα μηνύματα έναρξης εμφανίζονται με σειρά που έχει καθοριστεί μέσω της υλοποίησης των κλήσεων fork εντός του προγράμματος. Η σειρά αυτή δεν ακολουθείται πάντα με ακριβώς τον ίδιο τρόπο, καθώς αυτό σχετίζεται και με τον scheduler. Ο scheduler καθορίζει και την σειρά εμφάνισης των μηνυμάτων τερματισμού.

3η άσκηση:

Πρόγραμμα:

```
#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"

pid_t pid;
pid_t n_pid;

struct tree_node *root;

void handle_cont(int sig){
	printf("Hello im: %d and I'm awake\n",
			getpid());
}
void fork_procs(struct tree_node *root){

	int status;

	printf("I'm alive, my name is: %s (%ld) and my father is: %ld\n",
				root->name, (long)getpid(), (long)getppid());
	change_pname(root->name);

	if (root -> nr_children == 0){ //WILL BE EXECUTED BY THE LEAVES
			raise(SIGSTOP);
		// printf("Hello im: %d and my namy is: %s and I'm awake\n",
getpid(), root->name);
			exit(2);
	} else{ //IF root has any children
		pid_t arr_id[root->nr_children]; //=====EVERY PARENT HAS ONE,
TO KEEP TRACK OF THE LITTLE ONES====
		int i;
			for (i=0; i < root->nr_children; i++){//===IS IT NOT
BEAUTIFUL=> THE CHILDREN ARE BORN
			arr_id[i] = fork();
			if(arr_id[i] == -1){
```

```
                        perror("Child:fork");
                        exit(1);
                }
                if (arr_id[i] ==0){ //AS LONG AS THERE ARE CHILDREN GO TO
THE NEXT ELEMENT ON children <LIST>!
                        fork_procs(root->children+i); //RUN child FOR THE
NEXT [CHILD OF ROOT] until it is LEAF
                }
                }//WHEN FOR ENDS, ALL OF THE FATHER'S CHILDREN WILL HAVE
BEEN BORN

        //=====HERIN LIES WHAT THE FATHER WILL DO======
        wait_for_ready_children(root -> nr_children);
        raise(SIGSTOP); //WILL PROCEED AFTER (SIGCONT)
        //printf("Hello im: %d and my namy is: %s, and I'm awake\n",
getpid(), root->name); //THE FATHER IS AWAKE (SIGCONT)
  //printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);

        //=====NOW THAT THE FATHER IS ALIVE, HE IS WAKING UP ALL THE
CHILDREN||AND THEN WAITS FOR THEM====

        int j;
        for (j=0;j< root -> nr_children;j++){
              kill(arr_id[j], SIGCONT);
              int p = wait(&status);
              explain_wait_status(p,status);
        }
        }
        exit(0);
}


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

{
        printf("Main's pid is: %d\n", getpid());
        int status;
        //struct tree_node *root;

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

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

        struct sigaction sa;
              // sigemptyset(&sa.sa_mask);
              sa.sa_flags= SA_RESTART;
              sa.sa_handler=&handle_cont;
              sigaction(SIGCONT, &sa, NULL);
    /* Fork root of process tree */
        pid = fork();
        if (pid < 0) {
                    perror("main: fork");
                    exit(1);
        }
        if (pid == 0) {
                    /* Child */
                    fork_procs(root);
                    exit(1);
        } else{
              wait_for_ready_children(1);
              show_pstree(pid);
              //sleep(2);
              kill(pid, SIGCONT);
              wait(&status);
              explain_wait_status(pid, status);
              return 0;
        }
}
```

Η έξοδος του προγράμματος είναι η ακόλουθη:

```
Main's pid is: 1540
I'm alive, my name is: A (1541) and my father is: 1540
I'm alive, my name is: B (1542) and my father is: 1541
I'm alive, my name is: C (1543) and my father is: 1541
My PID = 1541: Child PID = 1543 has been stopped by a signal, signo = 19
I'm alive, my name is: E (1545) and my father is: 1542
My PID = 1542: Child PID = 1545 has been stopped by a signal, signo = 19
I'm alive, my name is: D (1544) and my father is: 1541
My PID = 1541: Child PID = 1544 has been stopped by a signal, signo = 19
I'm alive, my name is: F (1546) and my father is: 1542
My PID = 1542: Child PID = 1546 has been stopped by a signal, signo = 19
My PID = 1541: Child PID = 1542 has been stopped by a signal, signo = 19
My PID = 1540: Child PID = 1541 has been stopped by a signal, signo = 19


A(1541)─┬─B(1542)─┬─E(1545)
        │         └─F(1546)
        ├─C(1543)
        └─D(1544)


Hello im: 1541 and I'm awake
Hello im: 1542 and I'm awake
Hello im: 1545 and I'm awake
My PID = 1542: Child PID = 1545 terminated normally, exit status = 2
Hello im: 1546 and I'm awake
My PID = 1542: Child PID = 1546 terminated normally, exit status = 2
My PID = 1541: Child PID = 1542 terminated normally, exit status = 0
Hello im: 1543 and I'm awake
My PID = 1541: Child PID = 1543 terminated normally, exit status = 2
Hello im: 1544 and I'm awake
My PID = 1541: Child PID = 1544 terminated normally, exit status = 2
My PID = 1540: Child PID = 1541 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 συνάρτηση sleep είναι μια κλήση συστήματος, η οποία θέτει ένα πρόγραμμα σε μια παθητική κατάσταση για ένα συγκεκριμένο χρονικό διάστημα, το οποίο το ορίζει ο προγραμματιστής και αποτελεί μεταβλητή της συνάρτησης sleep().
Όταν περάσει αυτό το χρονικό διάστημα το πρόγραμμα συνεχίζει.
Αυτό όμως δεν αποτελεί ακριβώς επικοινωνία μεταξύ των διεργασιών.
Με την χρήση σημάτων SIGSTOP/SIGCONT μπορεί να επιτευχθεί διεργασιακή επικοινωνία, η οποία είναι πιο άμεση, αφού δεν μπλέκεται ο προγραμματιστής και έτσι στέλνοντας ένα σήμα μπορεί κανείς να σταματήσει μια διεργασία και στέλνοντας ένα άλλο να την συνεχίσει.
Η χρήση της ρουτίνας sleep() αν και μας επιτρέπει να "επιβάλλουμε" στις διεργασίες μας την σειρά που επιθυμούμε, αποτελεί μια πρακτική που μερικές φορές ίσως να μην είναι επιθυμητή. Αυτό γιατί η sleep απαιτεί κάποια "πρόβλεψη" του χρόνου που χρειάζεται μια διεργασία για να ολοκληρωθεί, κάτι αρκετά δύσκολο. Σε κάθε περίπτωση το "πρόβλημα" αυτό λύνεται με τη χρήση σημάτων, όπου επιτυγχάνεται πραγματική επικοινωνία μεταξύ διεργασιών και ο χρονισμός τους είναι βέλτιστος. Να σημειωθεί ότι αυτό δεν ακυρώνει τη λειτουργία της sleep(), η οποία βρίσκει πολλές εφαρμογές στις διεργασίες.

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

Από το όνομα της ήδη μπορούμε να προϊδεάστουμε για την λειτουργία της. Η ρουτίνα wait_for_ready_children() περιμένει μέχρι τα παιδιά του πατέρα που έχουμε ορίσει να κάνουν αναστολή της εκτέλεσης τους με το σήμα SIGSTOP πριν ο ίδιος αναστείλει την λειτουργία του με SIGSTOP.

Σε περίπτωση που παραλειφθεί η ρουτίνα wait_for_ready_children(), θα δημιουργηθεί το εξής πρόβλημα:
Ας υποθέσουμε ότι δεν έχει γίνει η σωστή πρόβλεψη και δεν έχει χρησιμοποιηθεί η συνάρτηση wait_for_ready_children(), και έστω ότι έχουμε μια διεργασία πατέρα Β και δύο διεργασίες παιδιά C και D, οι οποίες εκτελούν αρκετά χρονοβόρες συναρτήσεις. Σε αυτήν την περίπτωση, ο Β, αφού δεν περιμένει τα παιδιά του κάποια στιγμή θα λάβει το σήμα raise(SIGSTOP) και θα τερματίσει πριν τα παιδιά του προλάβουν να τερματίσουν (θεωρούμε ότι είναι αρκετά χρονοβόρες διεργασίες και δεν έχουν λάβει ακόμα το σήμα SIGSTOP). Η διεργασία Β κάποια στιγμή θα λάβει το σήμα raise(SIGCONT) και θα στείλει το μήνυμα SIGCONT στα παιδιά του, σε άκυρο χρόνο βέβαια, αφού τα παιδιά δεν έχουν κάνει ακόμα raise(SIGSTOP) και η διεργασία πατέρας θα τερματίσει. Σε αυτήν την περίπτωση τα παιδιά κάποια στιγμή, όταν θα φτάσουν στο τέλος του προγράμματος τους και λάβουν το σήμα raise(SIGSTOP), δεν θα υπάρχει κάποια διεργασία πατέρας να τους στείλει σήμα SIGCONT, επομένως θα σταματήσουν χωρίς να τερματίσουν. Συμπεραίνουμε ότι η παράλειψη της συνάρτησης wait_for_ready_children(), μπορεί να οδηγήσει στον μη-τερματισμό των παιδιών. Η χρήση των σημάτων απαιτεί προσοχή στο συγκεκριμένο ζήτημα, ώστε να υπάρχει διασφάλιση ότι όλες οι διεργασίες που λαμβάνουν το σήμα SIGSTOP θα λάβουν και το σήμα SIGCONT σε σωστό χρόνο.

Άσκηση 4:

Πρόγραμμα:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include "proc-common.h"
#include "tree.h"
//for presentation check tree.c, a function is different

void child(struct tree_node *root, int ind, int fd[]){
	int status;
	change_pname(root->name); //CHANGES the name that the ptr root is
pointing to
	if (root -> nr_children == 0){
			// sleep(SLEEP_PROC_SEC);
			int te= atoi(root->name);
			//printf("My name/number is: %d\n", te);
			close(fd[2*ind]);
			if(write(fd[2*ind+1],&te, sizeof(int))<0){
			perror("Child: writing to pipe\n");
			exit(5);
			}
			close(fd[2*ind+1]);
			exit(1);
	}
	else { //CREATEEE PIPE=====
			if(pipe(&fd[2*(++ind)])<0){
			perror("Parent: creating pipe\n");
			exit(3);
			}

			int i;
			for (i=0; i < root->nr_children; i++){//ie: root->4
			pid_t n_pid = fork();
			if(n_pid == -1){
				perror("Child:fork");
				exit(1);
			}
			if (n_pid ==0){ //AS LONG AS THERE ARE CHILDREN->dive
```

```
                        child(root->children+i, ind, fd); //root stays
unchanged=====why i ?
                }
                }
                int j;
                for (j=0;j< root -> nr_children;j++){
                int p = wait(&status);
                explain_wait_status(p,status);
                }
                //printf("=========CHECK2=======\n");
                close(fd[2*ind+1]);
                int a[2];

  if(read(fd[2*ind], &a, sizeof(a))<0){
                perror("Parent: reading from child\n");
                exit(4);
                }
                close(fd[2*ind]);
                //printf("=========CHECK3=======\n");
                close(fd[2*ind-2]);
                int xr=0;
                if(*(root->name)=='+'){
                xr=a[0]+a[1];
                } else{
                xr=a[0]*a[1];
                }
                //printf("Im a father and my symbol is: %c
AAAAANDDD\nThis is what I have to say: The sum/mul is:%d \n", *root->name,
xr);
                if(write(fd[2*ind-1], &xr, sizeof(int))<0){
                perror("Father: writing the sum/mul of children\n");
                exit(3);
                }
                close(fd[2*ind-1]) ;
                //printf("=========CHECK4=======\n");
            exit(2);
      }
}

int main(int argc, char *argv[])
{
      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]);
        int d=print_tree(root);
        //printf("\n %d \n", d);

        int pfd[2*d];


        int status;
        if(pipe(pfd)<0){
                perror("Main: creating pipe\n");
                exit(9);
        }
        pid_t pid = fork();

        if (pid == -1) {
                perror("Root:fork");
 exit(1);
        }
        if (pid ==0) {
                child(root, 0, pfd);
                exit(1);
        } else{
                // sleep(SLEEP_TREE_SEC);
                // show_pstree(pid);
                int w = wait(&status);
                explain_wait_status(w,status);
                // printf("=========CHECK0=======\n");
                close(pfd[1]);
                int x;
                if(read(pfd[0], &x, sizeof(x))<0){
                        perror("Main: reading the result\n");
                        exit(9);
                }
                close(pfd[0]);
                printf("\nThe result is: %d\n", x);
                //printf("=========CHECK5=======\n");
        }
        return 0;
}
```

Έχει τροποποιηθεί μια συνάρτηση της tree.c, συγκεκριμένα η print_tree() , η οποία από void έγινε int και επιστρέφει το βάθος του δέντρου, επομένως την παραθέτουμε και αυτή για λόγους πληρότητας:

```
int print_tree(struct tree_node *root)
{
      __print_tree(root, 0);
      return(depth+1);
}
```

Η έξοδος του προγράμματος είναι η ακόλουθη:

```
+
        10
        *
                +
                        5
                        7
                4
My PID = 1719: Child PID = 1720 terminated normally, exit status = 1
My PID = 1721: Child PID = 1723 terminated normally, exit status = 1
My PID = 1722: Child PID = 1724 terminated normally, exit status = 1
My PID = 1722: Child PID = 1725 terminated normally, exit status = 1
My PID = 1721: Child PID = 1722 terminated normally, exit status = 2
My PID = 1719: Child PID = 1721 terminated normally, exit status = 2
My PID = 1718: Child PID = 1719 terminated normally, exit status = 2

The result is: 58
```

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;

Η συγκεκριμένη άσκηση έχει υλοποιηθεί με τέτοιο τρόπο, ώστε να χρειάζονται 4 σωληνώσεις και αυτό γιατί κάθε γονική διεργασία δημιουργεί μια σωλήνωση για όλα τα παιδιά της. Σε άλλη περίπτωση για κάθε παιδί θα χρειαζόταν μια ξεχωριστή σωλήνωση και άρα θα είχαμε στο σύνολο 7 σωληνώσεις. Στο

παράδειγμα αυτό μας επιτρέπεται αυτή η «εξοικονόμηση», επειδή οι αριθμητικοί τελεστές της πρόσθεσης και του πολλαπλασιασμού(+,) υποστηρίζουν την αντιμεταθετική ιδιότητα. Για τους αριθμητικούς τελεστές της αφαίρεσης και διαίρεσης (-,/), για τους οποίους δεν ισχύει η αντιμεταθετική ιδιότητα χρειάζεται μια σωλήνωση για το κάθε παιδί ώστε να εκτελούνται οι πράξεις με την σωστή σειρά που επιθυμούμε, αλλιώς το αποτέλεσμα διαφοροποιείται.*

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 με πολλούς επεξεργαστές μπορούν να εκτελεστούν πολλές διεργασίες ταυτόχρονα, οπότε η αποτίμηση της έκφρασης από δέντρο διεργασιών χρησιμοποιώντας την συνάρτηση fork() είναι πολύ πιο γρήγορη, καθώς οι διεργασίες μοιράζονται ανά τους επεξεργαστές, οπότε επειδή γίνονται ταυτόχρονα οι πράξεις, είναι αρκετά ταχύτερο από το να εκτελεί ένα τέτοιο πρόγραμμα μια και μόνο διεργασία.