ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Άσκηση 2

Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

Λειτουργικά Συστήματα

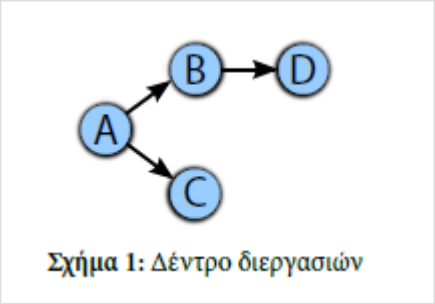
Εργαστηρική Ομάδα B02

Βαβουλιώτης Γεώργιος
Α.Μ. : 03112083
7ο εξάμηνο

Γιαννούλας Βασίλειος
Α.Μ. : 03112117
7ο εξάμηνο

Ακ. Έτος 2015-2016

1. Δημιουργία δεδομένου δέντρου διεργασιών



Σχήμα 1: Δέντρο διεργασιών

Σκοπός της άσκησης αυτής είναι η δημιουργία ενός προγράμματος που υλοποιεί το δέντρο διεγρασιών που φαίνεται στο διπλανό σχήμα. Για να παραχθεί το δέντρο διεγρασιών που φαίνεται δίπλα πρέπει οι διεγρασίες να διατηρούνται ενεργές για ορισμένο χρονικό διάστημα ώστε ο χρήστης να προλαβαίνει να δει το δέντρο. Για το σκοπό αυτό χρησιμοποιούμε τη κλήσης συστήματος(system call) sleep() για τα φύλλα του δέντρου ώστε να μην πεθάνουν για το ορισμένο χρονικό διάστημα το οποίο μας δίνετε απο την εκφώνηση, ενώ οι ενδιάμεσοι κόμβοι αναμένουν τον θάνατο των παιδίων τους για να τερματιστούν.Για τους παραπάνω λόγους μέχρι να κάνουν exit τα φύλλα, το δέντρο διεγρασιών είναι δυνατόν να παρατηρηθεί απο τον χρήστη. Παρακάτω παρατίθεται ο κώδικας της άσκησης αυτής :

```
#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


void fork_procs(void)
{
	pid_t pid;
	int status;
	change_pname("A");

	pid = fork();
	if(pid < 0){
		perror("fork_procs: fork\n");
		exit(1);
	}
	if(pid == 0){
		// A creates B
		change_pname("B");
		pid_t p;
		// B wants to create D
		p = fork();
		if (p < 0){
			//fail
			perror("fork_procs: fork\n");
			exit(1);
		}
		if (p == 0){
			//B creates D
			change_pname("D");
			printf("D: Sleeping...\n");
			sleep(SLEEP_PROC_SEC);
			printf("D: Exiting...\n");
			exit(13);
		}
		// fathers space (B)
		printf("B: Waiting...\n");
		p = wait(&status);
		explain_wait_status(p, status);
		printf("B: Exiting...\n");
		exit(19);
	}
```

```
	// fathers space (A)
	//A wants to create C
	pid = fork();
	if(pid < 0){
		perror("fork_procs: fork\n”);
		exit(1);
	}
	if(pid == 0){
		// A creates C
		change_pname("C");
		printf("C: Sleeping...\n");
		sleep(SLEEP_PROC_SEC);
		printf("C: Exiting...\n");
		exit(17);
	}
	printf("A: Waiting...\n");
	pid = wait(&status);
	explain_wait_status(pid, status);
	pid = wait(&status);
	explain_wait_status(pid, status);
	printf("A: Exiting...\n");
	exit(16);
}


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

	/* Fork root of process tree */
	pid = fork();
	if (pid < 0) {
		perror("main: fork\n”);
		exit(1);
	}
	if (pid == 0) {
		/* Child A */
		fork_procs();
		exit(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 */
	pid = wait(&status);
	explain_wait_status(pid, status);
	return 0;
}
```

Αν τρέξω τον παραπάνω κώδικα παίρνω το αποτέλεσμα που φαίνεται παρακάτω (σας το παραθέτω όπως αυτό εμφανίζεται όταν το τρέχω στο terminal):

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


A(5749)─┬─B(5750)──D(5752)
        └─C(5751)


C: Exiting...
D: Exiting...
My PID = 5749: Child PID = 5751 terminated normally, exit status = 17
My PID = 5750: Child PID = 5752 terminated normally, exit status = 13
B: Exiting...
My PID = 5749: Child PID = 5750 terminated normally, exit status = 19
A: Exiting...
```

Ερωτήσεις

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

Προφανώς η διεργασία A έχει παιδιά τα οποία όταν εμείς στείλουμε το σήμα -KILL <pid> στην A μπορεί και να μην έχουν προλάβει να πεθάνουν. Στη γενική περίπτωση όταν μια διεργασία τερματίζει και κάποια(μπορεί και όλα) απο τα παιδιά της ζούν ακόμα, τότε αυτά ανατίθενται στην init, η οποία είναι η αρχική διεργασία του συστήματος και έχει pid=1. Έτσι και στην περίπτωσή μας, αν τερματίσουμε πρόωρα την διεργασία A και κάποια(ή όλα) απο τα παιδιά της ζουν, τότε αυτά θα ανατεθούν στην init μέχρι να τερματίσουν και αυτά. Συγκεκριμένα λοιπόν αν κάποιο απο τα B και C(μπορεί και τα δυο), τα οποία είναι παιδιά της A, ζει ανατίθεται στην init, η οποία κάνει συνεχώς wait διότι είναι η αρχική διεργασία του συστήματος, μέχρι να τερματίσουν. Τέλος θα πρέπει να αναφερθεί πως τα παιδιά μιας διεργασίας που τερματίστηκε πρόωρα και δεν έχουν πεθάνει ακόμα ονομάζονται-είναι zombie.

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 show_pstree() γενικά εμφανίζει το δέντρο διεργασίων με αρχή τη διεργασία που έχει pid αυτό που δέχεται ως παράμετρο και κάτω. Συγκεκριμένα τώρα αν δώσουμε ως παράμετρο στην show_pstree() τη συνάρτηση getpid() θα δημιουργηθεί ενα διαφορετικό δέντρο διεργασιών απο αυτό που δημιουργήθηκε παραπάνω διότι η συνάρτηση getpid() θα επιστρέψει το pid της main() και πλέον το δέντρο διεργασιών που θα βλέπουμε θα έχει ρίζα την main(). Πιο αναλυτικά, θα πάρουμε το δέντρο με ρίζα την main, ένα υπόδεντρο το οποίο είναι το δέντρο διεργασιών της A και τέλος ένα άλλο υπόδεντρο που αφορά τις διεργασίες για την εμφάνιση του δέντρου διεργασιών(δεν παραθέτω το δέντρο διεργασιών που προκύπτει διότι εξήθηκε αναλυτικά η μορφή του).

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

Αν υποθέσουμε οτι σε ένα υπολογιστικό σύστημα πολλαπλών χρηστών, ο κάθε χρήστης είχε τη δυνατότητα να δημιουργήσει όσες διεργασίες αυτός επιθυμούσε, τότε επειδή ο συνολικός χρόνος ο οποίος είναι διαθέσιμος μοιράζεται δίκαια σε όλες τις διεργασίες, λόγω του μέγαλου αριθμού των διεργασιων ο χρόνος αυτός θα ήταν εξαιρετικά μικρός. Λόγω του πολύ μικρού χρόνου που θα έχει κάθε διεργασία στην διάθεση της για να εκτελέσει την λειτουργία της, το σύστημα θα κατέρρεε πλήρως για κάθε χρήστη μετά απο ορισμένο χρονικό διάστημα λόγω αδυναμίας να ανταπεξέλθει στις απαιτήσεις. Για τον λόγο αυτό σε συστήματα πολλαπλών χρηστών ο διαχειριστής θέτει όρια όσο αφορά το πλήθος των διεργασιών που κάθε χρήστης μπορεί να δημιουργήσει.

2.Δημιουργία αυθαίρετου δέντρου διεργασιών

Σκοπός της άσκησης αυτής είναι η δημιουργία ενός αυθαίρετου δέντρου διεργασιών βάσει αρχείου εισόδου που μας δίνετε. Το πρόγραμμα μας θα βασιστεί σε αναδρομική κλήση συνάρτησης, η οποία καλείται για κάθε κόμβο του δέντρου. Αν ο κόμβος του δέντρου έχει παιδιά, η συνάρτηση θα δημιουργεί τα παιδιά και θα περιμένει να τερματιστούν. Αν όχι δηλαδή αν πρόκειται για φύλλα, η συνάρτηση θα καλεί τη sleep() με προκαθορισμένο όρισμα. Παρακάτω παρατίθεται ο κώδικας της άσκησης αυτής :

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

typedef struct tree_node * nodeptr;

void fork_procs(nodeptr root)
{
     int i;
     pid_t pid;
     int status;

     change_pname(root->name);

     if( root->nr_children == 0 ){
           //leaf, same as 2i
           printf("%s: Sleeping...\n", root->name);
           sleep(SLEEP_PROC_SEC);
           printf("%s: Exiting...\n", root->name);
           exit(0);
     }
     else{
           for(i=0; i<root->nr_children; i++){
                 pid = fork();
                 if (pid < 0){
                       perror("fork_procs: fork\n”);
                       exit(1);
                 }
                 if (pid == 0){
```

```
                        root= root->children +i;
                        fork_procs(root);
                        exit(1);
                  }
            }
            for(i=0; i<root->nr_children; i++){
                  pid = wait(&status);
                  explain_wait_status(pid, status);
            }

            printf("%s: Exiting....\n", root->name);
            exit(0);
      }
}



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

      if(argc != 2) {
            printf("Usage: %s <input_tree_> \n", argv[0]);
      }
      root= get_tree_from_file(argv[1]);
      pid = fork();
      if (pid < 0) {
            perror("main: fork\n”);
            exit(1);
      }
      if (pid == 0) {
            /* Child */
            change_pname(root->name);
            print_tree(root);
            fork_procs(root);
            exit(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 */
      pid = wait(&status);
      explain_wait_status(pid, status);

      return 0;
}
```

Αν τρέξω τον παραπάνω κώδικα, δίνοντας ως όρισμα το αρχείο proc.tree παίρνω το εξής αποτέλεσμα(σας το παραθέτω όπως αυτό εμφανίζεται όταν το τρέχω στο terminal):

```
oslabd02@leykada:~$ ./runner1 proc.tree
A
        B
                E
                F
        C
        D
A : Waiting for children to exit
B : Waiting for children to exit
C: Sleeping...
D: Sleeping...
E: Sleeping...
F: Sleeping...


A(5890)─┬─B(5891)─┬─E(5894)
        │         └─F(5895)
        ├─C(5892)
        └─D(5893)


C: Exiting...
D: Exiting...
E: Exiting...
F: Exiting...
My PID = 5890: Child PID = 5892 terminated normally, exit status = 0
My PID = 5890: Child PID = 5893 terminated normally, exit status = 0
My PID = 5891: Child PID = 5894 terminated normally, exit status = 0
My PID = 5891: Child PID = 5895 terminated normally, exit status = 0
B: Exiting....
My PID = 5890: Child PID = 5891 terminated normally, exit status = 0
A: Exiting....
My PID = 5889: Child PID = 5890 terminated normally, exit status = 0
```

Ερωτήσεις

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

Θα πρέπει να τονιστεί οτι τα μηνυματά έναρξης και τερματισμού των διεργασιών εμφανιζόνται με τυχαία σειρά, δηλαδή είναι πιθανό να αλλάξει η σειρά εμφάνισης κάποιων μηνυμάτων απο εκτέλεση σε εκτέλεση. Κάτι τέτοιο είναι λογικό διότι οι διεργασίες δεν κατασκευάζονται με κάποια συγκεκριμένη σειρά αλλά καθεμία που έχει παιδιά ξεκινάει να τα δημιουργεί και όταν όλα τερματιστούν τερματίζει κι εκείνη, δηλαδή εμφανίζεται το μήνυμα *<process name>: Exiting…* . Εφόσον εμείς δεν παρεμβαίνουμε στη σειρά, η σειρά εμφάνισης των μηνυμάτων γίνεται κατά τρόπο τυχαίο. Θα πρέπει να τονιστεί πως το μόνο γενονός που δεν είναι τυχαίο στα παραπάνω μηνύματα είναι πως κάθε πατέρας πεθαίνει μετά τα παιδία του, δηλαδή ο B τερματίζει ενώ έχει τερματίσει ο E και ο F και ο A τερματίζει ενω έχει τερματίσει ο B, ο C και ο D και η σειρά αυτή δεν μπορεί να αλλάξει σε καμία εκτέλεση του προγράμματος.

3. Αποστολή και χειρισμός σημάτων

Σκοπός της άσκησης αυτής είναι εκ νέου κατασκευή ενός δέντρου διεργασιών, χρησιοποιώντας ενα αρχείο εισόδου, όπου όμως η δημιουργία των διεργασιών θα γίνει μέσω DFS ντετερμινιστικά και όχι τυχαία. Για να είναι δυνατό αυτό και να μπορεί να παρατηρηθεί το δέντρο γίνεται χρήση σημάτων. Τα φύλλα κάνουν κατευθείαν raise το σήμα SIGSTOP ενώ οι ενδιάμεσοι κόμβοι κάνουν raise το σήμα SIGSTOP αφού έχουν κάνει το ίδιο όλα τα παιδιά τους. Στη συνέχεια οι κόμβοι ξυπνάνε ένας ένας αφού λάβουν σήμα SIGCONT από τον πατέρα τους. Η αφύπνιση των κόμβων γίνεται κι αυτή κατά DFS αφού για να στείλει ένας πατέρας σήμα SIGCONT στο επόμενο παιδί του θα πρέπει το προηγούμενο παιδί να έχει κάνει το ίδιο στο δικό του υποδέντρο. Επανάληψη αυτού μας δίνει τη DFS διάσχιση που επιθυμούμε. Παρακάτω παρατίθεται ο κώδικας της άσκησης αυτής :

```
#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"

typedef struct tree_node * nodeptr;

void fork_procs(struct tree_node *root)
{
	pid_t matrix[root->nr_children]; //pid array
	int status,i;

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

	if(root->nr_children !=0){
		for(i=0; i<root->nr_children;i++){
				matrix[i]=fork();
				if(matrix[i]<0){
					perror("fork_procs: fork\n”);
					exit(1);
				}
				if( matrix[i] == 0){
					fork_procs(root->children+i);
				}
				wait_for_ready_children(1);
		}
	}

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

	if(root->nr_children != 0){
		for(i=0;i<root->nr_children;i++){
			kill(matrix[i], SIGCONT);
			wait(&status);
			explain_wait_status(matrix[i], status);
		}
	}
	//exit point
	exit(0);
}
```

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

      if (argc < 2){
            printf("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\n”);
            exit(1);
      }
      if (pid == 0) {
            /* Child */
            fork_procs(root);
            exit(1);
      }

      /* for ask2-signals */
      wait_for_ready_children(1);

      /* 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;
}
```

Αν τρέξω τον παραπάνω κώδικα, δίνοντας ως όρισμα το αρχείο proc.tree παίρνω το εξής αποτέλεσμα(σας το παραθέτω όπως αυτό εμφανίζεται όταν το τρέχω στο terminal):

```
oslabd02@leykada:~$ ./runner2 proc.tree
PID = 5900, name A, starting...
PID = 5901, name B, starting...
PID = 5902, name E, starting...
My PID = 5901: Child PID = 5902 has been stopped by a signal, signo = 19
PID = 5903, name F, starting...
My PID = 5901: Child PID = 5903 has been stopped by a signal, signo = 19
My PID = 5900: Child PID = 5901 has been stopped by a signal, signo = 19
PID = 5904, name C, starting...
My PID = 5900: Child PID = 5904 has been stopped by a signal, signo = 19
PID = 5905, name D, starting...
My PID = 5900: Child PID = 5905 has been stopped by a signal, signo = 19
My PID = 5899: Child PID = 5900 has been stopped by a signal, signo = 19


A(5900)─┬─B(5901)─┬─E(5902)
        │         └─F(5903)
        ├─C(5904)
        └─D(5905)


PID = 5900, name = A is awake
PID = 5901, name = B is awake
PID = 5902, name = E is awake
My PID = 5901: Child PID = 5902 terminated normally, exit status = 0
PID = 5903, name = F is awake
My PID = 5901: Child PID = 5903 terminated normally, exit status = 0
My PID = 5900: Child PID = 5901 terminated normally, exit status = 0
PID = 5904, name = C is awake
My PID = 5900: Child PID = 5904 terminated normally, exit status = 0
PID = 5905, name = D is awake
My PID = 5900: Child PID = 5905 terminated normally, exit status = 0
My PID = 5899: Child PID = 5900 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 sleep() την οποία χρησιμοποιούσαμε έως τώρα για να πετύχουμε το συγχρονισμό των διεργασιών, αυτό που κάνει είναι απλά να κοιμίζει τα φύλλα για κάποιο χρονικό διάστημα το οποίο της δίνεται ως όρισμα χωρίς όμως να ελέγχεται η σειρά με την οποία ξυπνάνε οι διάφοροι κόμβοι. Αντίθετα, τα σήματα έχουν handler δηλαδή ο χρήστης καθορίζει εκείνος τη σειρά δημιουργίας και αφύπνισης των διεργασιών, δηλαδή πλέον όλα γίνονται ντετερμινιστικά και όχι τυχαία. Το προφανές πλεονέλτημα της χρήσης σημάτων είναι ο ντετερμινισμός που προσφέρουν όσο αφορά τη δημιουργία και αφύπνιση διεργασιών. Συγκεκριμένα, η χρήση σημάτων είναι ιδιαίτερα χρήσιμη σε περιπτώσεις που είναι αναγκαίο κάποια διεργασία να πεθαίνει αργότερα από κάποια άλλη ή η λειτουργία μιας διεργασίας να προϋποθέτει την ύπαρξη μιας άλλης.

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

Αρχικά θα πρέπει να αναφερθεί οτι η συνάρτηση wait_for_ready_children() περιμένει έως ότου τα παιδία(για την ακρίβεια ο αριθμός των παιδιών που έχει πάρει ως όρισμα) μιας διεργασίας κάνουν raise το σήμα SIGSTOP και μετά επιτρέπει να εκτελεστεί η επόμενη εντολή. Η συνάρτηση αυτή χρησιμοποιείται στον κώδικα μας με παράμετρο 1 αφού πρέπει κάθε πατέρας πρέπει να περιμένει κάθε προηγούμενο παιδί του να σηκώσει σήμα SIGSTOP για να πάει στο επόμενο. Χρησιμοποιώντας αυτή την ιδέα καταφέρνουμε να αφυπνίσουμε τους κόμβους με σειρά DFS χρησιμοποιώντας τον πίνακα με τα pid των παιδιών τον οποίο είχαμε δημιουργήσει πιο πρίν για το σκοπό αυτό. Στον πίνακα αυτό βάζουμε κάθε φορά το pid του παιδιού, μετά κάνουμε wait_for_ready_children(1) και μετά κάνουμε το ίδιο και για το επόμενο παιδί αφού προφανώς έχει τελειώσει το προηγούμενο. Η ιδέα αυτή δουλεύει αφού οι διεργασίες-παιδιά δεν έχουν κοινή μνήμη με την διεργασία-πατέρα(σε αντίθεση με τα νήματα). Ωστόσο αν παραλείψουμε τη συνάρτηση αυτή θέτουμε σε κίνδυνο την DFS διάσχιση. Συγκεκριμένα, υπάρχει ο κίνδυνος κάποιος <πρόγονος> στο δέντρο να πεθάνει πριν από το παιδί που τον έχει ως <πρόγονο> ή ακόμα και πριν τη δημιουργία του παιδιού. Αυτό θα είχε ως αποτέλεσμα να μην είναι δυνατή η παρατήρηση του δέντρου διεργασιών.

4. Παράλληλος υπολογισμός αριθμητικής έκφρασης

Στην άσκηση αυτή πραγματοποείται ο υπολογισμός μιας αριθμητικής έκφρασης με χρήση ενός δέντρου διεργασιών και της τεχνικής της σωλήνωσης. Για την άσκηση αυτή γράφτηκαν δυο διαφορετικής φιλοσοφίας κώδικες, οι οποίοι παράγουν το ίδιο(σωστό) αποτέλεσμα.

1ος Τρόπος: Στον τρόπο αυτό χρησιμοποιήθηκε μόνο ένα νέο pipe(χωρίς χρήση σημάτων) για να επικοινωνήσουν τα παιδιά με τον πατέρα(προφανώς υπάρχει και το pipe που επικοινωνεί ο πατέρας με τον δικό του πατέρα),διότι οι πράξεις της πρόσθεσης και του πολλαπλασιασμού είναι αντιμεταθετικές και γι'αυτό δεν μας ενδιαφέρει η σειρά των τελεστέων. Συγκεκριμένα εφόσον στο δέντρο εκφράσεων κάθε νέος πατέρας έχει αναγκαστικά 2 παιδιά δημιουργώ ένα νέο πίνακα 2 θέσεων με τα pid των παιδιών του κάθε πατέρα και όπως προανέφερα ένα νέο pipe για να μπορούν να επικοινωνήσουν. Όταν φτάσω σε φύλλα γράφω την τιμή τους στο αντίστοιχο pipe, με το οποίο επικοινωνούν με τον πατέρα τους. Δηλαδή κάθε pipe θα έχει 2 τιμές τις οποίες θα πρέπει να διαβάσει ο πατέρας μια προς μια και να κάνει την αντίστοιχη πράξη, της οποίας το αποτέλεσμα γράφει στο pipe που έχει με τον πατέρα του. Κάτι τέτοιο γίνεται απλούστατα αν κάνεις δυο διαδοχικά read απο το pipe διότι ως γνωστόν το pipe είναι μια FIFO δομή. Τελικά το αρχικό pipe έχει πλέον το αποτέλεσμα της αριθμητικής έκφρασης, αφού η παραπάνω διαδικασία γίνεται επαναληπτικά για όλους τους κόμβους. Παρακάτω παρατίθεται ο κώδικας της άσκησης αυτής, ο οποίος αφορά τον πρώτο τρόπο υλοποίησης:

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

typedef struct tree_node * nodeptr;

void fork_procs(struct tree_node *root, int * fd)
{
     int result, num1, num2;
     int fd1[2];
     pid_t pid[2];
     printf("PID = %ld, name %s, starting...\n",
           (long)getpid(), root->name);
     change_pname(root->name);

     if (root->nr_children != 0) {

           if (pipe(fd1) < 0) {
                perror("Pipe error!\n");
                exit(1);
           }

           pid[0] = fork();
           if (pid[0] < 0) {
                perror("fork_procs: fork\n”);
                exit(1);
           }
           else if (pid[0] == 0) {
                close(fd1[0]);
                fork_procs(root->children, fd1);
           }
           pid[1] = fork();
           if (pid[1] < 0) {
                perror("fork_procs: fork\n”);
                exit(1);
           }
           else if (pid[1] == 0) {
                fork_procs(root->children + 1, fd1);
           }
           close(fd1[1]);

           if (read(fd1[0],&num1, sizeof(num1)) != sizeof(num1)) {
                perror("read from pipe\n”);
                exit(1);
           }
           printf("Parent with pid %ld received number %d from child with pid
                %ld\n",(long)getpid(), num1, (long)pid[0]);

           if (read(fd1[0],&num2, sizeof(num2)) != sizeof(num2))
           {
                perror("read from pipe\n");
                exit(1);
           }
           printf("Parent with pid %ld received value %d from child with pid
                %ld\n",(long)getpid(), num2,(long)pid[1]);

           close(fd1[0]); // gia asfaleia

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

```
            if (write(fd[1], &result, sizeof(result)) != sizeof(result))
            {
                  perror("write to pipe\n");
                  exit(1);
            }

            close(fd[1]);
            exit(0);
      }
      else {
            result = atoi(root->name);
            if (write(fd[1], &result, sizeof(result)) != sizeof(result))
            {
                  perror("write to pipe\n");
                  exit(1);
            }

            close(fd[1]);
            exit(0);
      }
}


int main(int argc, char *argv[])
{
      pid_t pid;
      int fd[2];
      int result;
      nodeptr root;
      if (argc < 2){
            printf("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(fd) < 0) {
            perror("Pipe error\n");
            exit(1);
      }

      /* Fork root of process tree */
      pid = fork();
      if (pid < 0) {
            perror("main: fork\n");
            exit(1);
      }
      if (pid == 0) {
            /* Child writes only*/
            close(fd[0]);
            fork_procs(root, fd);
            exit(0);
      }

      /* Father reads only*/
      close(fd[1]);

      if (read(fd[0],&result,sizeof(result)) != sizeof(result)) {
            perror("read from pipe\n");
            exit(1);
      }

      sleep(10); //gia na dw to apotelesma opws prepei
      printf("The result of the arithmetic expression is: %d\n", result);
      return 0;
}
```

Αν τρέξω τον παραπάνω κώδικα, δίνοντας ως όρισμα το αρχείο expr.tree παίρνω το εξής αποτέλεσμα(σας το παραθέτω όπως αυτό εμφανίζεται όταν το τρέχω στο terminal):

```
oslabd02@anafi:~$ gcc newiv.o proc-common.o tree.o -o lastrunner
oslabd02@anafi:~$ ./lastrunner expr.tree
+
        10
        *
                +
                        5
                        7
                4
PID = 12770, name +, starting...
PID = 12771, name 10, starting...
PID = 12772, name *, starting...
PID = 12773, name +, starting...
PID = 12774, name 4, starting...
PID = 12775, name 5, starting...
PID = 12776, name 7, starting...
Parent with pid 12770 received number 10 from child with pid12771
Parent with pid 12772 received number 4 from child with pid12773
Parent with pid 12773 received number 5 from child with pid12775
Parent with pid 12773 received value 7 from child with pid 12776
Parent with pid 12772 received value 12 from child with pid 12774
Parent with pid 12770 received value 48 from child with pid 12772
The result of the arithmetic expression is: 58
```

2ος Τρόπος: Στο δεύτερο τρόπο χρησιμοποιήθηκαν και σήματα χωρίς να το ζητάει η εκφώνηση από προσώπική επιλογή καθαρά, δηλαδή ακολούθησα το σκεπτικό της 3ης άσκησης, χρησιμοποιώντας ταυτόχρονα και την ιδέα του τρόπου υλοποίησης που φαίνεται παραπάνω.Συγκεκριμένα, ακολούθησα την ίδια ιδέα με την 3η άσκηση(δηλαδή DFS διάσχιση με χρήση πίνακα για τα pid των παιδιών χρησιμοποιώντας σήματα) απλά πρόσθεσα ενα νέο pipe για κάθε πατέρα με τα παιδιά του, αφού ένα είναι αρκετό, όπως αναφέρθηκε και στον 1ο τρόπο υλοποίησης. Στη συνέχεια η πορεία λύσης είναι η ίδια με τον 1ο τρόπο υλοποίησης(με ελάχιστες διαφορές όπως για παράδειγμα οτι ο πίνακας με τα pid στον 1ο τρόπο δηλώνεται ως pid[2] ενω στον δεύτερο τρόπο ως matrix[root->nr_children], τα οποία όμως πρακτικά είναι ίδια). Παρακάτω παρατίθεται ο κώδικας της άσκησης αυτής, ο οποίος αφορά τον δεύτερο τρόπο υλοποίησης:

```
#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"

typedef struct tree_node * nodeptr;


void fork_procs(nodeptr root, int * fd)
{
     int status,i;
     int fdn[2];
     pid_t matrix[root->nr_children];
     printf("PID = %ld, name %s, starting...\n", (long)getpid(), root->name);
     change_pname(root->name);
     int res,num1,num2,num3;

     if(root->nr_children != 0){
           if(pipe(fdn)<0){
                 perror("Pipe error\n");
                 exit(1);
           }
           pid_t pid1;
           for(i=0; i<root->nr_children;i++){
                       pid1=fork();
                       if(pid1<0){
                             perror("fork_procs: fork\n");
                             exit(1);
                       }
                       if( pid1  == 0){
                             fork_procs(root->children+i, fdn);
                             exit(0);
                       }
                       matrix[i] = pid1;
                       wait_for_ready_children(1);
           }
     }
     else{
           raise(SIGSTOP);
           close(fd[0]);
           printf("PID = %ld, name = %s is awake\n",
                  (long)getpid(), root->name);
           num1=atoi(root->name);
           if(write(fd[1],&num1,sizeof(num1)) != sizeof(num1)){
                 perror("write to pipe\n");
                 exit(1);
           }
           exit(0);
     }
```

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

	if(root->nr_children != 0){
	   for(i=0;i<root->nr_children;i++){
			kill(matrix[i], SIGCONT);
			wait(&status);
			explain_wait_status(matrix[i], status);
		}
	}

	close(fdn[1]);
	if(read(fdn[0], &num2,sizeof(num2)) != sizeof(num2)){
		perror("read from pipe\n");
		exit(1);
	}
	if(read(fdn[0], &num3, sizeof(num3)) != sizeof(num3)){
		perror("read from pipe\n");
		exit(1);
	}
	if(strcmp(root->name, "*") ==0 ){
		res=num2*num3;
	}
	else if(strcmp(root->name, "+") == 0){
		res=num2+num3;
	}
	if(write(fd[1], &res, sizeof(res)) != sizeof(res)){
		perror("write to pipe\n");
		exit(1);
	}
	exit(0);
}


int main(int argc, char *argv[])
{
	pid_t pid;
	int status, res;
	nodeptr root;
	int fd[2];

	if (argc < 2){
		printf("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("Pipe error\n”);
		exit(1);
	}

	//fork
	pid = fork();
	if (pid < 0) {
		perror("main: fork\n”);
		exit(1);
	}
	if (pid == 0) {
		// the child only writes
		close(fd[0]);
		fork_procs(root,fd);
		exit(0);
	}
	//the father reads only
	close(fd[1]);
```

```
	wait_for_ready_children(1);

	show_pstree(pid);

	kill(pid, SIGCONT);
	wait(&status);
	explain_wait_status(pid, status);
	if(read(fd[0], &res, sizeof(res)) != sizeof(res)){
		perror("read from pipe\n")	;
		exit(1);
	}
	printf("The result of the expression tree is %d\n", res);
	return 0;
}
```

Αν τρέξω τον παραπάνω κώδικα, δίνοντας ως όρισμα το αρχείο expr.tree παίρνω το εξής αποτέλεσμα(σας το παραθέτω όπως αυτό εμφανίζεται όταν το τρέχω στο terminal):

```
oslabd02@leykada:~$ ./runner3 expr.tree
PID = 5916, name +, starting...
PID = 5917, name 10, starting...
My PID = 5916: Child PID = 5917 has been stopped by a signal, signo = 19
PID = 5918, name *, starting...
PID = 5919, name +, starting...
PID = 5920, name 5, starting...
My PID = 5919: Child PID = 5920 has been stopped by a signal, signo = 19
PID = 5921, name 7, starting...
My PID = 5919: Child PID = 5921 has been stopped by a signal, signo = 19
My PID = 5918: Child PID = 5919 has been stopped by a signal, signo = 19
PID = 5922, name 4, starting...
My PID = 5918: Child PID = 5922 has been stopped by a signal, signo = 19
My PID = 5916: Child PID = 5918 has been stopped by a signal, signo = 19
My PID = 5915: Child PID = 5916 has been stopped by a signal, signo = 19


+(5916)─┬─*(5918)─┬─+(5919)─┬─5(5920)
        │         │         └─7(5921)
        │         └─4(5922)
        └─10(5917)


PID = 5916, name = + is awake
PID = 5917, name = 10 is awake
My PID = 5916: Child PID = 5917 terminated normally, exit status = 0
PID = 5918, name = * is awake
PID = 5919, name = + is awake
PID = 5920, name = 5 is awake
My PID = 5919: Child PID = 5920 terminated normally, exit status = 0
PID = 5921, name = 7 is awake
My PID = 5919: Child PID = 5921 terminated normally, exit status = 0
My PID = 5918: Child PID = 5919 terminated normally, exit status = 0
PID = 5922, name = 4 is awake
My PID = 5918: Child PID = 5922 terminated normally, exit status = 0
My PID = 5916: Child PID = 5918 terminated normally, exit status = 0
My PID = 5915: Child PID = 5916 terminated normally, exit status = 0
The result of the expression tree is 58
```

Παρατηρήσεις :

1. Προφανώς η άσκηση αυτή έχει δημιουργηθεί για να λυθεί με τον 1ο τρόπο ώστε να κατανοηθεί πως στα pipes το read και το write είναι blocking, δηλαδή δεν έχουν ανάγκη την ύπαρξη σημάτων για να παράξουν το σωστό αποτέλεσμα. Άρα, υπο αυτό το πρίσμα ο 2ος τρόπος υλοποίησης είναι κάπως περιττός και λίγο πιο δύσκολος.
2. Επίσης αντί να γράψω τον πρώτο τρόπο υλοποίησης θα μπορούσα να τροποποιήσω κατά το ελάχιστο τον 2ο τρόπο ώστε να πάρω το ίδιο αποτέλεσμα που θα έπαιρνα με τον 1ο τρόπο υλοποίησης. Ο κώδικας αυτός είναι όμοιος με αυτον του δεύτερου τρόπου υλοποίησης, με διαφορά οτι λείπουν τα σήματα SIGSTOP και SIGCONT, όπως και η κλήση της συνάρτησης wait_for_ready_children(). Επίσης πρόσθεσα και κάποια sleep() όπου ήταν αναγαίο για να μπορεί ο χρήστης να παρατηρήσει το δέντρο. Ο κώδικας φαίνεται παρακάτω:

```
#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"

typedef struct tree_node * nodeptr;

void fork_procs(nodeptr root, int * fd)
{
     int status;
     int i;
     int fdn[2];
     pid_t matrix[root->nr_children];
     printf("PID = %ld, name %s, starting...\n", (long)getpid(), root->name);
     change_pname(root->name);
     int res,num1,num2,num3;

     if(root->nr_children != 0){
           if(pipe(fdn)<0){
                 perror(“Pipe error\n”);
                 exit(1);
           }
           pid_t pid1;
           for(i=0; i<root->nr_children;i++){
                       pid1=fork();
                       if(pid1<0){
                             perror("fork_procs: fork\n”);
                             exit(1);
                       }
                       if( pid1  == 0){
                             fork_procs(root->children+i, fdn);
                             exit(0);
                       }
                       matrix[i] = pid1;
           }
     }
     else{
           close(fd[0]);
           sleep(10); // den 8a mporousa na dw to pstree xwris delay
           num1=atoi(root->name);
           if(write(fd[1],&num1,sizeof(num1)) != sizeof(num1)){
                 perror("write to pipe\n");
                 exit(1);
           }
           exit(0);
```

```
	}



	if(root->nr_children != 0){
		for(i=0;i<root->nr_children;i++){
			wait(&status);
			explain_wait_status(matrix[i], status);
		}
	}

	close(fdn[1]);

	if(read(fdn[0], &num2,sizeof(num2)) != sizeof(num2)){
		perror("read from pipe\n");
		exit(1);
	}
	if(read(fdn[0], &num3, sizeof(num3)) != sizeof(num3)){
		perror("read from pipe\n");
		exit(1);
	}
	if(strcmp(root->name, "*") ==0 ){
		res=num2*num3;
	}
	else if(strcmp(root->name, "+") == 0){
		res=num2+num3;
	}
	if(write(fd[1], &res, sizeof(res)) != sizeof(res)){
		perror("write to pipe\n");
		exit(1);
	}
	exit(0);
}


int main(int argc, char *argv[])
{
	pid_t pid;
	int status, res;
	nodeptr root;
	int fd[2];

	if (argc < 2){
		printf("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("Pipe error\n”);
		exit(1);
	}

	//fork
	pid = fork();
	if (pid < 0) {
		perror("main: fork\n”);
		exit(1);
	}
	if (pid == 0) {
		/* Child */
		// the child only writes
		close(fd[0]);
		fork_procs(root,fd);
		exit(0);
	}

	//FATHER
```

```
        //the father reads
        close(fd[1]);
        /* for ask2-signals */
        //sleep(10);
        show_pstree(pid);

        /* Wait for the root of the process tree to terminate */
        wait(&status);
        explain_wait_status(pid, status);
        if(read(fd[0], &res, sizeof(res)) != sizeof(res)){
              perror("read from pipe\n")    ;
              exit(1);
        }
        sleep(3);
        printf("The result of the expression tree is %d\n", res);
        return 0;
}
```

Αν τρέχαμε κι αυτό το κώδικα θα παίρναμε το εξής αποτέλεσμα:

```
oslabd02@anafi:~$ ./rer expr.tree
PID = 12909, name +, starting...
PID = 12911, name 10, starting...
PID = 12912, name *, starting...
PID = 12914, name 4, starting...
PID = 12913, name +, starting...
PID = 12916, name 7, starting...
PID = 12915, name 5, starting...


+(12909)─┬─*(12912)─┬─+(12913)─┬─5(12915)
         │          │          └─7(12916)
         │          └─4(12914)
         └─10(12911)


My PID = 12909: Child PID = 12911 terminated normally, exit status = 0
My PID = 12912: Child PID = 12913 terminated normally, exit status = 0
My PID = 12913: Child PID = 12915 terminated normally, exit status = 0
My PID = 12913: Child PID = 12916 terminated normally, exit status = 0
My PID = 12912: Child PID = 12914 terminated normally, exit status = 0
My PID = 12909: Child PID = 12912 terminated normally, exit status = 0
My PID = 12908: Child PID = 12909 terminated normally, exit status = 0
The result of the expression tree 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 γενική περίπτωση χρειάζονται 3 σωληνωσεις ανά διεργασία αν αυτή είναι ενδιάμεσος κόμβος ώστε να επικοινωνεί με τα δύο παιδιά της(αφού έχει αναγκαστικά 2) και με τον πατέρα της. Αν τώρα πρόκειται για φύλλο 1 σωλήνωση είναι αρκετή ώστε να επικοινωνεί με τον πατέρα του. Ωστόσο στην άσκηση μας οι ενδιάμεσοι κόμβοι είναι εφοδιασμένη μόνο την πράξη του πολλαπλασιασμού και της πρόσθεσης,οι οποίες είναι προσεταιριστικές πράξεις, άρα μπόρουμε να χρησιμοποιήσουμε αν θέλουμε 1 pipe και το τελικό αποτέλεσμα θα είναι το ίδιο. Γενικά αν είχαμε πράξεις που δεν είναι προσεταιριστικές(πχ αφαίρεση), θα έπρεπε αναγκαστικά να έχουμε ενα pipe για κάθε παιδί διότι αν διαβάσω με τυχαία σειρά τότε σίγουρα θα υποπέσω κάποια στιγμή σε λάθος, άρα δεν μπορεί να χρησιμοποιηθεί 1 pipe για κάθε αριθμητικό τελεστή.

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;

Η απότιμηση της έκφρασης μέσω δέντρου έχει προφανές πλεονέκτημα διότι εφόσον έχουμε στη διάθεση μας ενα σύστημα πολλών επεξεργαστών μπορεί κάθε κλάδος του δέντρου να μοιραστεί σε κάποιο επεξεργαστή. Αυτό θα είχε ως αποτέλεσμα να κερδίσουμε αρκετό χρόνο όσον αφορά τον υπολογισμό της έκφρασης. Το χρονικό αυτό κέρδος αυξάνεται πολύ αν για παράδειγμα έχουμε να υπολογίσουμε μια πολύ μεγάλη έκφραση, το οποίο είναι ενα σύνηθες φαινόμενο.

Παρατήρηση : Η διαδικασία με την οποία καταφέραμε να τρέξουμε τις παραπάνω ασκήσεις παρότι είναι προφανής εξηγείται παρακάτω για πληρότητα:

- Γράφουμε στο terminal την εντολή make και το Makefile το οποίο μας έχει δοθεί κάνει τα απαραίτητα compile ώστε να παράξει όλα τα object files που χρειαζόμαστε .
- Αν υποθέσουμε οτι θέλουμε να τρέξουμε την 4η άσκηση τότε πρέπει να γράψουμε επίσης gcc -Wall -c my2iv.c και παράγουμε το my2iv.o, αφού επέλεξα να γράψω την άσκηση μου σε διαφορετικό αρχείο απο το προβλεπόμενο και πρέπει το compile να το κάνω χειροκίνητα(θα μπορούσα να το είχα βάλει στο Makefile). Έπειτα κάνουμε το κατάλληλο linking αρχειών ώστε να μπορεί να εκτελεστεί ο κώδικας και παράγουμε το εκτελέσιμο αρχείο με όνομα runner3:

  gcc proc-common.o tree.o my2iv.o -o runner3

- Τέλος με χρήση της εντολής ./runner3 expr.tree τρέχω την τελευταία άσκηση.
- Προφανώς με παρόμοια διαδικασία τρέχω και τις προηγούμενες ασκήσεις απλά επέλεξα να δείξω μόνο την τελευταία.