2η Ομάδα Ασκήσεων

στα Λειτουργικά Συστήματα

Ομάδα Α38:

Ιωακειμίδη Αθηνά

Α.Μ.: 03114758

Μαυρομμάτης Ιάσων

Α.Μ.: 03114771

Εξάμηνο 7ο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εθνικό Μετσόβιο Πολυτεχνείο

Άσκηση 1.1

Πηγαίος κώδικας:

```
#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, pid2, pid3;
	int status;

	change_pname("A");
	printf("A: Starting...\n");

	pid = fork();
	if (pid < 0) { perror("fork_procs: fork"); exit(1);}
	else if (pid == 0) {
		change_pname("B");
		printf("B: Starting...\n");

		pid2 = fork();
		if (pid2 < 0) { perror("fork_procs: fork"); exit(1);}
		else if (pid2 == 0) {
			change_pname("D");
			printf("D: Sleeping...\n");
			sleep(SLEEP_PROC_SEC);

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

		printf("B: Waiting...\n");

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

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

	pid3 = fork();
	if (pid3 < 0) { perror("fork_procs: fork"); exit(1);}
	else if (pid3 == 0) {
		change_pname("C");
		printf("C: Sleeping...\n");
		sleep(SLEEP_PROC_SEC);

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

```
	printf("A: Waiting...\n");

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

	pid3 = wait(&status);
	explain_wait_status(pid3, status);

	printf("A: Exiting...\n");
	exit(16);
}

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

	pid = fork();
	if (pid < 0) {
		perror("main: fork");
		exit(1);
	}
	if (pid == 0) {
		fork_procs();
		exit(1);
	}

	sleep(SLEEP_TREE_SEC);

	show_pstree(pid);

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

	return 0;
}
```

Έξοδος εκτέλεσης:

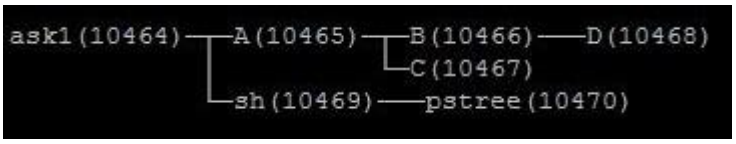
```
A: Starting...
A: Waiting...
B: Starting...
B: Waiting...
C: Sleeping...
D: Sleeping...


A(10305)─┬─B(10306)───D(10308)
         └─C(10307)


C: Exiting...
My PID = 10305: Child PID = 10307 terminated normally, exit status = 13
D: Exiting...
My PID = 10306: Child PID = 10308 terminated normally, exit status = 17
B: Exiting...
My PID = 10305: Child PID = 10306 terminated normally, exit status = 19
A: Exiting...
My PID = 10304: Child PID = 10305 terminated normally, exit status = 16
```

Απαντήσεις στις ερωτήσεις:

1. Εάν τερματίσουμε πρόωρα τη διεργασία Α, τότε τα παιδιά της θα γίνουν παιδιά της διαδικασίας init, όπου και θα συνεχίσουν να τρέχουν μέχρι να τερματιστούν.
2. Αν κάνουμε `show_pstree(getpid())` αντί για `show_pstree(pid)`, τότε το δέντρο που θα εμφανιστεί θα είναι το εξής:



   Στο δέντρο εμφανίζεται η διεργασία του προγράμματος που τρέχει μαζί με τα παιδιά της, την Α και τα παιδιά της και το shell και το παιδί της, το pstree. Αυτό συμβαίνει γιατί με την εντολή `getpid()` παίρνουμε το pid της τρέχουσας διαδικασίας.
3. Όλες οι διεργασίες για να τρέξουν απαιτούν μνήμη. Ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει κάθε χρήστης για να εξασφαλίσει ότι όλοι οι χρήστες θα μπορούν να τρέχουν τις δικές τους διαδικασίες χωρίς κάποιος από αυτούς να μονοπωλεί τη μνήμη.

Άσκηση 1.2

Πηγαίος κώδικας:

```
#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

void fork_procs(struct tree_node *root)
{
	if (root->nr_children==0) {
		/*leaf*/
		change_pname(root->name);
		printf("%s: Sleeping...\n", root->name);
		sleep(SLEEP_PROC_SEC);

		printf("%s: Exiting...\n", root->name);
		exit(2);			/*exit status: leaf->2*/
	}
	else {
		/*non-leaf*/
		pid_t pid[root->nr_children];
		int i, status;

		change_pname(root->name);
		printf("%s: Starting...\n", root->name);

		for (i=0; i<root->nr_children; i++){
			pid[i] = fork();
			if (pid[i] < 0) {
				perror("fork_procs: fork");
				exit(1);
			}
			else if (pid[i] == 0) {
				fork_procs(root->children + i);
				exit(1);
			}
		}

		printf("%s: Waiting...\n", root->name);

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

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

```
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);
	}
	root = get_tree_from_file(argv[1]);

	pid = fork();
	if (pid < 0) {
		perror("main: fork");
		exit(1);
	}
	else if (pid == 0) {
		fork_procs(root);
		exit(1);
	}

	sleep(SLEEP_TREE_SEC);

	show_pstree(pid);

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

	return 0;
}
```

Έξοδος εκτέλεσης:

```
A: Starting...
A: Waiting...
D: Sleeping...
B: Starting...
B: Waiting...
C: Sleeping...
F: Sleeping...
E: Sleeping...


A(10338)─┬─B(10339)─┬─E(10342)
         │          └─F(10343)
         ├─C(10340)
         └─D(10341)


D: Exiting...
My PID = 10338: Child PID = 10341 terminated normally, exit status = 2
C: Exiting...
F: Exiting...
My PID = 10338: Child PID = 10340 terminated normally, exit status = 2
My PID = 10339: Child PID = 10343 terminated normally, exit status = 2
E: Exiting...
My PID = 10339: Child PID = 10342 terminated normally, exit status = 2
B: Exiting...
My PID = 10338: Child PID = 10339 terminated normally, exit status = 1
A: Exiting...
My PID = 10337: Child PID = 10338 terminated normally, exit status = 1
```

Απαντήσεις στις ερωτήσεις:

1. Τα μηνύματα έναρξης και τερματισμού των διεργασιών φαίνεται να εμφανίζονται με τυχαία σειρά, διατηρώντας όμως τον κανόνα “πρώτα εμφανίζεται το μήνυμα έναρξης του γονέα και μετά του παιδιού” και “ πρώτα εμφανίζεται το μήνυμα τερματισμού του παιδιού και μετά του γονέα”. Αυτό συμβαίνει γιατί σε κάθε κόμβο, ενώ οι διεργασίες παιδιά δημιουργούνται με τη σειρά που δίνονται στο αρχείο εισόδου, δεν μπορούμε να ελέγξουμε με ποια σειρά θα εκτελεστούν από το σύστημα.
   Βλέπουμε για παράδειγμα ότι το παιδί D ξεκινάει να εκτελείται πριν από το B και, λόγω του ότι οι δύο διαδικασίες έχουν τον ίδιο χρόνο που κοιμούνται, το μήνυμα τερματισμού του D εμφανίζεται πριν το αντίστοιχο του B. Βέβαια το B περιμένει πρώτα να τερματιστούν τα παιδιά του, γι’ αυτό και πριν το δικό του μηνύματα εμφανίζονται τα μηνύματα των C, F και E, που είτε τερματίζονται πρώτα επειδή είναι και αυτά παιδιά του A (C) είτε επειδή είναι παιδιά του B (F και E).

Άσκηση 1.3

Πηγαίος κώδικας:

```
#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)
{
	if (root->nr_children==0) {
		/*leaf*/
		change_pname(root->name);
		printf("%s: Starting...\n", root->name);
		printf("%s: SigStopping...\n", root->name);
		raise(SIGSTOP);

		exit(2);			/*exit status: leaf->2*/
	}
	else {
		/*non-leaf*/
		pid_t pid[root->nr_children];
		int i;

		change_pname(root->name);
		printf("%s: Starting...\n", root->name);

		for (i=0; i<root->nr_children; i++) {
			pid[i] = fork();
			if (pid[i] < 0) {
				perror("fork_procs: fork");
				exit(1);
			}
			else if (pid[i] == 0) {
				fork_procs(root->children + i);
				exit(1);
			}
			wait_for_ready_children(1);
		}

		printf("%s: SigStopping...\n", root->name);
		raise(SIGSTOP);

		for (i=0; i<root->nr_children; i++)
			kill(pid[i], SIGCONT);

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

		exit(1);			/*exit status: non-leaf->1*/
```

```
	}
}

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

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

	wait_for_ready_children(1);

	show_pstree(pid);

	kill(pid, SIGCONT);

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

	return 0;
}
```

Έξοδος εκτέλεσης:

```
A: Starting...
B: Starting...
E: Starting...
E: SigStopping...
My PID = 10350: Child PID = 10351 has been stopped by a signal, signo = 19
F: Starting...
F: SigStopping...
My PID = 10350: Child PID = 10352 has been stopped by a signal, signo = 19
B: SigStopping...
My PID = 10349: Child PID = 10350 has been stopped by a signal, signo = 19
C: Starting...
C: SigStopping...
My PID = 10349: Child PID = 10353 has been stopped by a signal, signo = 19
D: Starting...
D: SigStopping...
My PID = 10349: Child PID = 10354 has been stopped by a signal, signo = 19
A: SigStopping...
My PID = 10348: Child PID = 10349 has been stopped by a signal, signo = 19


A(10349)─┬─B(10350)─┬─E(10351)
         │          └─F(10352)
         ├─C(10353)
         └─D(10354)


My PID = 10350: Child PID = 10351 terminated normally, exit status = 2
My PID = 10350: Child PID = 10352 terminated normally, exit status = 2
My PID = 10349: Child PID = 10350 terminated normally, exit status = 1
My PID = 10349: Child PID = 10353 terminated normally, exit status = 2
My PID = 10349: Child PID = 10354 terminated normally, exit status = 2
My PID = 10348: Child PID = 10349 terminated normally, exit status = 1
```

Απαντήσεις στις ερωτήσεις:

1. Τα πλεονεκτήματα που έχει η χρήση σημάτων είναι τα εξής :
    a. Τα σήματα μας δίνουν πληροφορίες για τις καταστάσεις των διεργασιών (π.χ. τερματίστηκε, βρίσκεται σε αναμονή κλπ) και υπάρχει επικοινωνία μεταξύ γονέων και παιδιών.
    b. Σε αντίθεση με πριν δεν χρειάζεται να χρησιμοποιήσουμε την εντολή sleep με αυθαίρετο χρόνο για να εξασφαλίσουμε ότι η διεργασία γονέας θα περιμένει τον υπολογισμό αποτελεσμάτων. (π.χ. η διεργασία της άσκησης κοιμάται για να προλάβουν να δημιουργηθούν τα παιδιά και τα παιδιά κοιμούνται για να προλάβει η ίδια να τυπώσει το δέντρο διεργασιών)
    c. Με τη χρήση των σημάτων η κάθε διεργασία γονέας μπορεί να αλλάξει την κατάσταση των παιδιών του, έτσι η εκτέλεση (έναρξη και τερματισμός) των διεργασιών γίνεται ακριβώς με τη σειρά που δίνεται στο αρχείο εισόδου.
2. Ο ρόλος της εντολής `wait_for_ready_children()` είναι να βάλει τον γονέα να περιμένει μέχρι να πάρει σήματα SGCHLD από τα παιδιά του, τα οποία και αναλύει. Αν τα σήματα είναι SIGSTOP, τότε συνεχίζεται η κανονική ροή του προγράμματος. Έτσι μας εξασφαλίζει ότι ο τερματισμός των διεργασιών-παιδιών θα συμβεί πριν τον τερματισμό του γονέα. Αν παραλείπαμε την εντολή αυτή, ο γονέας θα τερμάτιζε πριν τα παιδιά του και τα παιδιά θα γίνονταν παιδιά της διαδικασίας init.

Άσκηση 1.4

Πηγαίος κώδικας:

```
#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

void fork_procs(struct tree_node *root, int filedesc)
{
	if (root->nr_children==0) {
		/*leaf*/
		int number;
		change_pname(root->name);
		number=atoi(root->name);

		if (write(filedesc, &number, sizeof(number)) !=
				sizeof(number)){
			perror("Child: write to pipe");
			exit(1);
		}

		sleep(SLEEP_PROC_SEC);
		exit(2);				/*exit status?? leaf->2*/
	}
	else {
		/*non-leaf*/
		int i, status, pfd[2];
		int result, num[2];

		pid_t pid[root->nr_children];
		change_pname(root->name);
		char op=*(root->name);

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

		for (i=0; i<root->nr_children; i++){
			pid[i] = fork();
			if (pid[i] < 0) {
				perror("fork_procs: fork");
				exit(1);
			}
			else if ( pid[i] == 0) {
				fork_procs(root->children + i,pfd[1]);
				exit(1);
			}
		}

		for (i=0; i<root->nr_children; i++){
			if (read(pfd[0], &num[i], sizeof(num[i])) !=
					sizeof(num[i])) {
```

```
                        perror("Parent: read from pipe");
                        exit(1);
                }
        }

        switch(op){
                case '+' :
                        result=num[0]+num[1];
                        break;
                case '*' :
                        result=num[0]*num[1];
                        break;
        }

        if (write(filedesc,&result, sizeof(result)) !=
                        sizeof(result)){
                perror("Parent: write to pipe");
                exit(1);
        }

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

        exit(1);                    /*exit status??? non-leaf->1*/
    }
}

int main(int argc, char *argv[])
{
    pid_t pid;
    int ProcFiledesc[2];
    pipe(ProcFiledesc);
    int status, result;
    struct tree_node *root;

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

    root = get_tree_from_file(argv[1]);

    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    else if (pid == 0) {
        fork_procs(root,ProcFiledesc[1]);
        exit(1);
    }

    if (read(ProcFiledesc[0], &result, sizeof(result)) !=
                sizeof(result)) {
        perror("main: read from pipe");
        exit(1);
    }
```

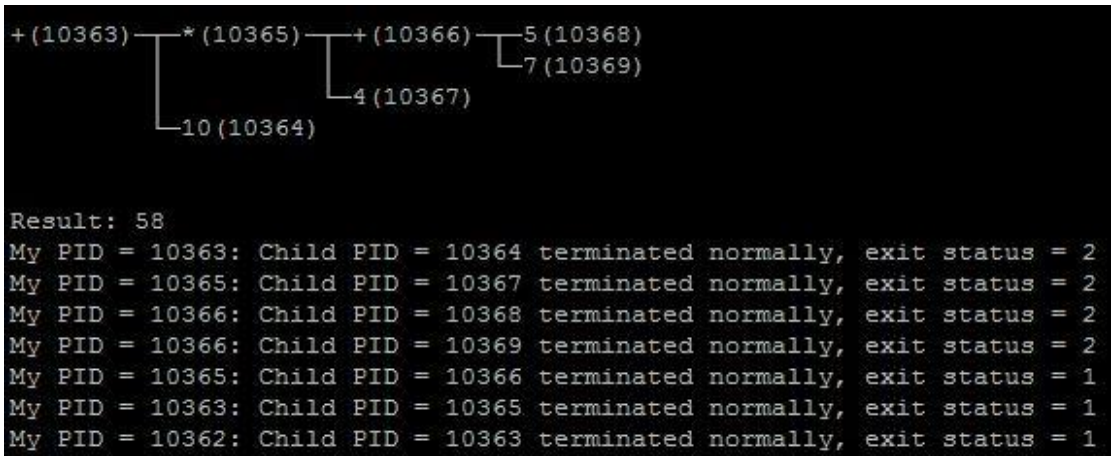
```
        sleep(SLEEP_TREE_SEC);

        show_pstree(pid);
        printf("Result: %d\n",result);

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

        return 0;
}
```

Έξοδος εκτέλεσης:



Απαντήσεις στις ερωτήσεις:

1. Στη συγκεκριμένη άσκηση κάθε γονέας χρησιμοποιεί μία σωλήνωση για να επικοινωνήσει με τα παιδιά του, άρα ανά διεργασία χρησιμοποιούνται δύο σωληνώσεις, μία για επικοινωνία με τον γονέα της και μία με τα παιδιά της. Αυτό είναι δυνατό γιατί, ενώ δεν μπορούμε να ελέγξουμε με ποια σειρά θα γράψουν στο σωλήνα τα παιδιά κάθε διεργασίας, οι πράξεις που δίνονται είναι αντιμεταθετικές οπότε το αποτέλεσμα δεν επηρεάζεται. Γενικά δεν μπορεί να χρησιμοποιηθεί μόνο μία σωλήνωση για κάθε αριθμητικό τελεστή, γιατί δεν είναι όλες οι πράξεις αντιμεταθετικές (π.χ. στην αφαίρεση και στη διαίρεση η σειρά που δίνονται οι δύο αριθμοί είναι κρίσιμη για το αποτέλεσμα της πράξης).
2. Σε ένα σύστημα πολλαπλών επεξεργαστών, η αποτίμηση μιας έκφρασης από ένα δέντρο διεργασιών μπορεί να γίνει πιο γρήγορα, αφού τα επιμέρους στοιχεία της έκφρασης υπολογίζονται παράλληλα αντί για σειριακά.