Λειτουργικά Συστήματα

7ο εξάμηνο, Ακαδημαϊκή περίοδος 2017-2018

Άσκηση 2:

Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

oslabc19

Χαράλαμπος Κάρδαρης
A.M. 03114074
bkardaris@hotmail.com

Μιχάλης Παπαδόπουλλος
A.M. 03114702
mechatr0nic.cy@gmail.com

Άσκηση 1:

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  5

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */
void fork_procs(void)
{
	int status;
	pid_t b, c, d;
	/*
	 * initial process is A.
	 */

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

	b = fork();
	if (b < 0) {
		perror("main: fork");
		exit(EXIT_FAILURE);
	}
	if (b == 0) {
		/* Child */
		printf("PID = %ld, name %s, starting...\n",
			(long)getpid(), "B");
		change_pname("B");

		d = fork ();
		if (d < 0)  { perror ("fork"); exit (EXIT_FAILURE); }
		if (d == 0) {
			printf("PID = %ld, name %s, starting...\n",
			(long)getpid(), "D");
```

```
                change_pname ("D");

                printf("D: Sleeping...\n");
                sleep(SLEEP_PROC_SEC);

                printf("D: exiting...\n");
                exit (13 /* 'D' */);
        }
        printf("B: Waiting for child %ld...\n", (long)d);
        wait (&status);
        explain_wait_status (d, status);

        printf("B: exiting...\n");
        exit(19 /* B */);
    }

    c = fork ();
    if (c < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (c == 0) {
        printf("PID = %ld, name %s, starting...\n",
                (long)getpid(), "C");
        change_pname ("C");

        printf("C: sleeping...\n");
        sleep(SLEEP_PROC_SEC + 2);

        printf("C: exiting...\n");
        exit (17 /* C */);
    }


    printf("A: Waiting for children...\n");
    //wait_for_ready_children (2);
    wait (&status);
    explain_wait_status (b, status);

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

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

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
```

```
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */
int main(void)
{
        pid_t pid;
        int status;

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

        /* for ask2-{fork, tree} */
        sleep(SLEEP_TREE_SEC);

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


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

        return 0;
}
```

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

Η διεργασία A, που είναι πατέρας των B,C θα λαβει το σήμα SIGKILL.
Οι διεργασίες B,C αποκαλούνται `zombie` μιας και ο πατέρας τους έχει πεθάνει χωρίς να περιμένει για τα παιδιά του να τελειώσουν και να καθαρίσει το process table με κλήση συστήματος wait().
Σε συστήματα UNIX, οι διεργασίες zombie, υιοθετούνται απο την διεργασία init (pid = 1) η οποία εκτελεί περιοδικά wait().

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

Το αποτέλεσμα της `show_pstree (getpid())` φαίνεται πιο κάτω:

Συγκεκριμένα, βλέπουμε ότι η κύρια διεργασία, δημιουργεί ακόμα ενα παιδί καλώντας την show_pstree().
Η συνάρτηση αυτή, καλεί την `system()`.

```
PID = 30647, name A, starting...
PID = 30648, name B, starting...
A: Waiting for children...
PID = 30649, name C, starting...
C: sleeping...
B: Waiting for child 30650...
PID = 30650, name D, starting...
D: Sleeping...


ex21(30646)─┬─A(30647)─┬─B(30648)──D(30650)
            │          └─C(30649)
            └─sh(30679)──pstree(30680)


D: exiting...
My PID = 30648: Child PID = 30650 terminated normally, exit status = 13
B: exiting...
My PID = 30647: Child PID = 30648 terminated normally, exit status = 19
C: exiting...
My PID = 30647: Child PID = 30649 terminated normally, exit status = 17
A: Exiting...
My PID = 30646: Child PID = 30647 terminated normally, exit status = 16
```

$ man 3 system
The system() library function uses fork(2) to create a child process that executes the shell command specified in command using execl(3) as follows: execl("/bin/sh", "sh", "-c", command, (char *) 0);

Σύμφωνα με το manual, η κλήση `system ()` καλέι την `fork ()` και μέσω της `execl ()` φορτώνει στη μνήμη της το κέλυφος */bin/*sh το οποίο με τη σειρά του εκτελέι το command `"echo; echo; pstree -G -c -p <pid>; echo; echo"` και γι' αυτό βλέπουμε ως παιδί της τη διεργασία που τυπώνει το δένδρο διεργασιών `pstree` (η φιλοσοφία των UNIX είναι να κάνει fork/execve για την εκτέλεση καποιου command απο το κέλυφος)

Όταν κληθεί `show_pstree (pid)` αντι για `show_pstree (getpid())` το δένδρο διεργασιών τυπώνεται με ρίζα (root) την διεργασία A, οπότε βλεπουμε το ακολουθο output:

```
PID = 13865, name A, starting...
PID = 13866, name B, starting...
A: Waiting for children...
B: Waiting for child 13868...
PID = 13868, name D, starting...
D: Sleeping...
PID = 13867, name C, starting...
C: sleeping...


A(13865)─┬─B(13866)──D(13868)
         └─C(13867)


D: exiting...
My PID = 13866: Child PID = 13868 terminated normally, exit status = 13
B: exiting...
My PID = 13865: Child PID = 13866 terminated normally, exit status = 19
C: exiting...
My PID = 13865: Child PID = 13867 terminated normally, exit status = 17
A: Exiting...
My PID = 13859: Child PID = 13865 terminated normally, exit status = 16
```

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

Ο εκάστοτε διαχειριστής καποιου συστήματος, πρέπει να εξασφαλίσει ότι κανένας χρήστης δε θα μπορεί να δημιουργήσει αυθαίρετο αριθμό απο διεργασίες – αφήνοντας ανοικτή την περίπτωση κάποιος χρήστης εκούσια ή ακούσια να κάνει κατάχρηση των διαθέσιμων system resources με αποτέλεσμα το σύστημα να μην είναι stable. Ένα χαρακτηριστικό παράδειγμα αποτελεί η επίθεση forkbomb η οποία δημιουργεί συνεχώς αντίγραφα του εαυτού της με σκοπό να κάνει ενα σύστημα να `κρασάρει` (DoS : Denial of Service attack).

Κλασσικό παράδειγμα σε Bash: **:(){ :|: & };:**
Δημιουργία συνάρτησης με όνομα `**:**` η οποία καλέι τον εαυτό της και με pipe στέλνει το output στον εαυτό της. Δηλαδή σε κάθε κλήση της δημιουργούνται 2 instances της ιδιας διεργασίας, χωρίς να τερματίζονται, εως οτου το συστημα ξεμείνει απο resources.

Άσκηση 2:

Κώδικας

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

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

#define SLEEP_PROC_SEC 6
#define SLEEP_TREE_SEC 2

pid_t *pid;
extern errno;

void creatProcTree (struct tree_node *node)
{

	int i, status;
	pid = (pid_t *)malloc (sizeof (pid_t) * node->nr_children);

	printf("%s: Hello, my pid is %ld\n", node->name, (long)getpid());
	change_pname (node->name);	// change name
	for (i = 0; i < node->nr_children; i++) {
		pid[i] = fork ();
		if (pid[i] < 0) { exit (EXIT_FAILURE); }
		else if (pid[i] == 0) { creatProcTree (&node->children[i]); }
	}

	while (i--) {
		errno = 0;
		if ( -1 == waitpid (pid[i], &status, WUNTRACED))
			fprintf(stderr, "(%ld): %s\n", (long)pid[i], strerror(errno));
	}

	if (node->nr_children == 0)  // put leaves for sleep
		sleep(SLEEP_PROC_SEC);

   printf("%s: Goodbye (%ld)\n", node->name, (long)getpid());
   exit(0);

	return;
}
```

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

	pid = fork();
	switch (pid) {
		case -1:
			perror ("fork");
			exit (EXIT_FAILURE);

		case 0:
			puts ("Creating process tree, recursively by BFS");
			creatProcTree (root);
			break;

		default:
			printf ("parent process: %d\n", getpid());
			break;
	}

	sleep (SLEEP_TREE_SEC);
	show_pstree (pid);

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

	return 0;
}
```

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

Το output του προγραμματός μας με είσοδο το αρχείο `proc.tree` φαίνεται πιο κάτω:

```
parent process: 26699
Creating process tree, recursively by BFS
A: Hello, my pid is 26700
B: Hello, my pid is 26701
C: Hello, my pid is 26702
D: Hello, my pid is 26703
E: Hello, my pid is 26704
F: Hello, my pid is 26705


A(26700)─┬─B(26701)─┬─E(26704)
         │          └─F(26705)
         ├─C(26702)
         └─D(26703)


C: Goodbye (26702)
D: Goodbye (26703)
E: Goodbye (26704)
F: Goodbye (26705)
B: Goodbye (26701)
A: Goodbye (26700)
My PID = 26699: Child PID = 26700 terminated normally, exit status = 0
```

Η δημιουργία των διεργασιών γίνεται με BFS (Breadth First Search) καθώς διατρέχουμε το δεντρο διεργασιών εισόδου κατα επίπεδα, καλώντας την `creatProcTree ()` αναδρομικά για κάθε διεργασία-παιδί.

Άσκηση 3:

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)
{
	int i;
	/*
	 * Start
	 */
	printf("PID = %ld, name %s, starting...\n",
			(long)getpid(), root->name);
	change_pname(root->name);

	/* My code for forks */
	pid_t *pid;

	pid = malloc(sizeof(pid_t) * root->nr_children);
	if (pid == NULL) exit (1);

	/*We suppose that every process creates all subprocesses
	without waiting for them to suspend individually. Then
	waits for all of them.
	*/
	for( /* int */ i = 0; i < root->nr_children; i++){
		pid[i] = fork();
		if (pid[i] < 0) {
			perror("fork");
			exit(1);
		}
		if (pid[i] == 0) { // create processes by BFS traversal
			/* Child */
			fork_procs(root->children + i);
			exit(1);
		}
	}

	wait_for_ready_children(root->nr_children);
```

```
	/*
	 * Suspend Self
	 */
	raise(SIGSTOP); // put current process to sleep by sending SIGSTOP to the current process id
CONTINUE:
	printf("PID = %ld, name = %s is awake\n",
		(long)getpid(), root->name);

	/* ... */
	int status;

	for(/* int */ i = 0; i < root->nr_children; i++){
		kill(pid[i], SIGCONT); // send awake signal to children => foreach (child) { goto
CONTINUE; }
		pid[i] = wait(&status);
		explain_wait_status(pid[i], status);
	}

	/*
	 * Exit
	 */
	free (pid);
	exit(0);
}

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */

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

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

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

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

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

	/* for ask2-{fork, tree} */
	/* sleep(SLEEP_TREE_SEC); */

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

	/* for ask2-signals */
	kill(pid, SIGCONT);

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

	return 0;
}
```

Δημιουργούμε το δέντρο διεργασιών με BFS traversal.
Στη συνέχεια, αναστέλλουμε τη λειτουργία κάθε διεργασίας, περιμένοντας πρώτα τα παιδιά της, με την `wait_for_ready_children (root->nr_children);`

```
PID = 5406, name A, starting...
PID = 5407, name B, starting...
PID = 5408, name C, starting...
PID = 5409, name D, starting...
PID = 5410, name E, starting...
My PID = 5406: Child PID = 5408 has been stopped by a signal, signo = 19
My PID = 5406: Child PID = 5409 has been stopped by a signal, signo = 19
My PID = 5407: Child PID = 5410 has been stopped by a signal, signo = 19
PID = 5411, name F, starting...
My PID = 5407: Child PID = 5411 has been stopped by a signal, signo = 19
My PID = 5406: Child PID = 5407 has been stopped by a signal, signo = 19
My PID = 5405: Child PID = 5406 has been stopped by a signal, signo = 19


A(5406)─┬─B(5407)─┬─E(5410)
        │         └─F(5411)
        ├─C(5408)
        └─D(5409)


PID = 5406, name = A is awake
PID = 5407, name = B is awake
PID = 5410, name = E is awake
My PID = 5407: Child PID = 5410 terminated normally, exit status = 0
PID = 5411, name = F is awake
My PID = 5407: Child PID = 5411 terminated normally, exit status = 0
My PID = 5406: Child PID = 5407 terminated normally, exit status = 0
PID = 5408, name = C is awake
My PID = 5406: Child PID = 5408 terminated normally, exit status = 0
PID = 5409, name = D is awake
My PID = 5406: Child PID = 5409 terminated normally, exit status = 0
My PID = 5405: Child PID = 5406 terminated normally, exit status = 0
```

Ακολούθως, στέλνουμε σήμα SIGCONT με DFS traversal.
Έτσι βλέπουμε πως `ξυπνάνε` οι διεργασίες με την ακολουθη σειρα:
A – B – E – F – C – D, αντι για A – B – C – D – E – F, που θα είχαμε σε περιπτωση BFS traversal.

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

Στις ασκήσεις 1 και 2 που έγινε χρήση της sleep(), χρειάστηκε να περιμένουμε ένα χρονικό διάστημα, που εμείς ορίζουμε εκ των προτέρων, το οποίο υποθέτουμε ότι θα είναι αρκετό για να δημιουργηθεί ολόκληρο το δέντρο διεργασιών, πριν το τυπώσουμε.
Είναι φανερό πως για μεγάλα δέντρα διεργασιών αυτή η μέθοδος δεν ανταποκρίνεται όπως θα θέλαμε, καθώς κάνει το προγραμμά μας να φαίνεται πως `λαγγαρει`.
Με τη χρήση σημάτων, έχουμε περισσότερο έλεγχο στο συγχρονισμο των διεργασιών και το πρόγραμμά μας τρέχει χωρίς περιττούς χρόνους αναμονής, όπως στην περίπτωση που καναμε χρηση της sleep(). Έτσι αυξάνουμε το responsiveness του προγραμματός μας και γενικά αποκτά καλύτερο scalability.

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

```c
void
wait_for_ready_children(int cnt)
{
    int i;
    pid_t p;
    int status;

    for (i = 0; i < cnt; i++) {
        /* Wait for any child, also get status for stopped children */
        p = waitpid(-1, &status, WUNTRACED);
        explain_wait_status(p, status);
        if (!WIFSTOPPED(status)) {
            fprintf(stderr, "Parent: Child with PID %ld has died unexpectedly!\n",
                (long)p);
            exit(1);
        }
    }
}
```

Όπως βλέπουμε και απο τον ορισμό της συνάρτησης που βρίσκεται στο αρχείο `proc-commons.h`, η συνάρτηση αυτή, εκτελεί την waitpid() για οποιοδήποτε αριθμό διεργασίας (pid = -1) τόσες φορές όσες δηλωνει η παράμετρος `cnt` (η παράμετρος `cnt` είναι ο αριθμός των παιδιών).

Σχετικά με το macro `WIFSTOPPED( )` :
WIFSTOPPED(*status*)

returns true if the child process was stopped by delivery of a signal; this is only possible if the call was done using **WUNTRACED** or when the child is being traced.

Δηλαδή, ελέγχει αν όλα τα παιδιά της εκάστοτε διεργασίας που κάλεσε την `wait_for_ready_children()` πως έχουν γίνει STOPPED απο κάποιο signal, και δεν έχουν σταματήσει την εκτελεσή τους για κάποιο άλλο (unexpected) λογο.

Άσκηση 4:

Κώδικας

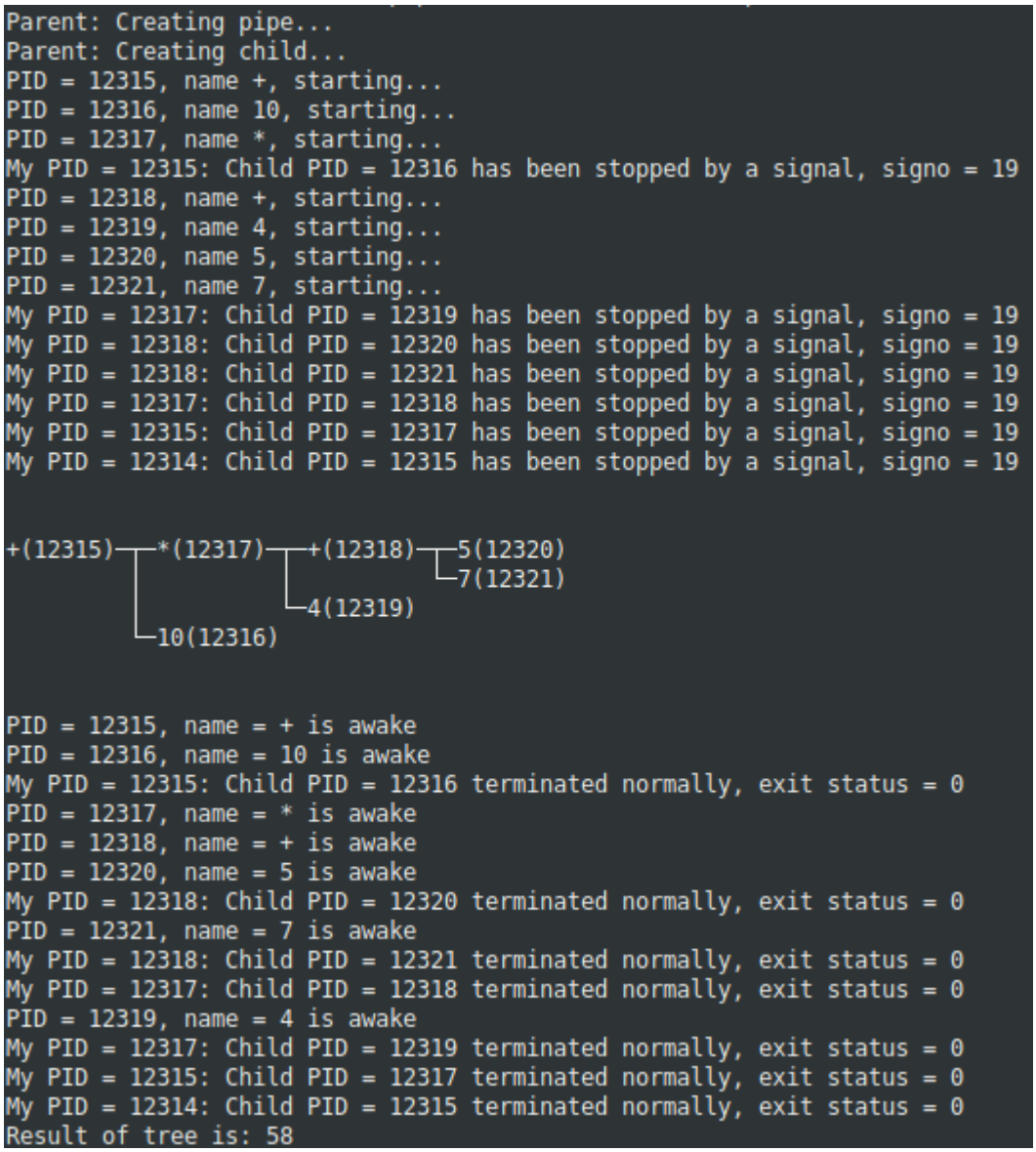
```
/*
 * pipe-example.c
 */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <assert.h>
#include <string.h>

#include <sys/wait.h>

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

void fork_procs(struct tree_node *root, int fd)
{
	int i;
	/*
	 * Start
	 */
	printf("PID = %ld, name %s, starting...\n",
			(long)getpid(), root->name);
	change_pname(root->name);

	/* ... */
	pid_t *pid;
	int pfd[2];

	pid = malloc(sizeof(pid_t) * root->nr_children);
	if (pid == NULL) exit (1);

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

	/*We suppose that every process creates all subprocesses
	without waiting for them to suspend individually. Then
	waits for all of them.
	*/
	for(/* int */ i = 0; i < root->nr_children; i++){
		pid[i] = fork();
		if (pid[i] < 0) {
```

```
                perror("fork");
                exit(1);
        }
        if (pid[i] == 0) {
                /* Child */
                fork_procs(root->children + i, pfd[1]);
                exit(1);
        }
}

wait_for_ready_children(root->nr_children);

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

/* ... */
int status, result;

if (root->nr_children == 0){
        result = atoi(root->name);
        // use writep from ex1
        if (write(fd, &result, sizeof(result)) != sizeof(result)) {
                perror("parent: write to pipe");
                exit(1);
        }
}
else {
        if (strcmp(root->name, "+") == 0){
                result = 0;
                for(/* int */ i = 0; i < root->nr_children; i++){
                        kill(pid[i], SIGCONT);

                        int temp;
                        if (read(pfd[0], &temp, sizeof(temp)) != sizeof(temp)) {
                                perror("parent: write to pipe");
                                exit(1);
                        }
                        result += temp;

                        wait(&status);
                        explain_wait_status(pid[i], status);
                }
        }
        else if (strcmp(root->name, "*") == 0){
                result = 1;
```

```
                        for(/* int */ i = 0; i < root->nr_children; i++){
                                kill(pid[i], SIGCONT); // deliver SIGCONT signal to process pid[i]

                                int temp;
                                if (read(pfd[0], &temp, sizeof(temp)) != sizeof(temp)) {
                                        perror("parent: write to pipe");
                                        exit(1);
                                }
                                result *= temp;

                                pid[i] = wait(&status);
                                explain_wait_status(pid[i], status);
                        }
                }
                if (write(fd, &result, sizeof(result)) != sizeof(result)) {
                        perror("parent: write to pipe");
                        exit(1);
                }
        }


        // close pipe fd and in children

        /*
         * Exit
         */
        free (pid);
        exit(0);
}

int main(int argc, char *argv[])
{
        pid_t p;
        int pfd[2];             // pipe fd[0,1]
        int status;
        int result;
        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]);

        printf("Parent: Creating pipe...\n");
        if (pipe(pfd) < 0) {
                perror("pipe");
```

```
		exit(1);
	}

	printf("Parent: Creating child...\n");

	p = fork();
	if (p < 0) {
		/* fork failed */
		perror("main: fork");
		exit(1);
	}
	if (p == 0) {
		/* Child */
		fork_procs(root, pfd[1]);
		exit(1);
	}

	/*
	 * In parent process.
	 */

	wait_for_ready_children(1);

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

	// deliver signal SIGCONT to proces p
	kill(p, SIGCONT);

	/* Read result from the pipe */
	if (read(pfd[0], &result, sizeof(result)) != sizeof(result)) {
		perror("parent: write to pipe");
		exit(1);
	}

	/* Wait for the root of the process tree to terminate */
	p = wait(&status);
	explain_wait_status(p, status);

	printf("Result of tree is: %d\n", result);

	return 0;
}
```

Η εκτέλεση του προγράμματος με το δοθέν αρχείο δίνει:



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;

Όπως φαίνεται και στον κώδικα, έχουμε κάνει υλοποίηση με ένα pipe ανά διεργασία. Αυτό γενικά είναι δυνατό γιατί τα writes μέσα στο pipe γίνονται ατομικά ,το οποίο σημαίνει ότι αν δύο η περισσότερα παιδιά προσπαθούν να γράψουν μέσα στο ίδιο pipe δεν θα υπάρχει περίπτωση να υπάρξει πρόβλημα με μπερδεμένα δεδομένα (εναλλάξ δεδομένα από τα διαφορετικά παιδιά).

Όσον αφορά βέβαια τη δική μας υλοποίηση, επειδή την κάναμε με βάση την άσκηση 3 και έχουμε επικονωνία με σήματα, εξασφαλίζουμε ότι κάθε φορά κάθε process parent περιμένει ανα πάσα στιγμή δεδομένα από ένα μόνο process child (τα SIGCONT στέλνονται με Depth First τρόπο). Οπότε σε κάθε περίπτωση είμαστε καλυμμένοι.

Η υλοποίση μας βέβαια υστερεί σε παραλληλοποίηση. Πράγματι επειδή αφαιρούμε τη δυνατότητα να “τρέξουν” πολλά παιδιά απο τον ίδιο γονιό ταυτόχρονα, δεν θα κερδίζαμε απολύτως τίποτα αν τρέχαμε το πρόγραμμα σε ένα πολυεπεξεργαστικό σύστημα.

Όσον αφορά την δημιουργία ενός pipe για κάθε πράξη, δηλαδή 2 pipes για όλο το πρόγραμμα, αυτό δεν είναι δυνατό να δουλέψει για λόγους που σχετίζονται με την προτεραιότητα των πράξεων. Παρόλα αυτά αν το πρόγραμμα εκτελόυσε μία μόνο πράξη λόγω προσεταιριστικότητας θα μπορούσε θεωρητικά να δουλέψει/

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;

Γενικά σε ένα τέτοιο σύστημα μπορούν να τρέξουν ταυτόχρονα περισσότερες διεργασίες αν δεν έχουν μεταξύ τους εξάρτηση. Στο συγκεκριμένο πρόγραμμα, αυτές είναι οι διεργασίες παιδιά κάθε γονιού. Αυτό σημαίνει ότι κάνοντας τις απαραίτητες τροποποιήσεις στο πρόγραμμα (δέντρο διεργασιών και επίτρεψη ταυτόχρονου τρεξίματος των παιδιών) μπορούμε στον ίδιο χρόνο να καάνουμε περισσότερες πράξεις και τελικά να έχουμε πιο γρήγορο υπολογισμό. Αντίθετα, αν δεν έχουμε ένα τέτοιο σύστημα ή μια κατάλληλη οργάνωση του προγράμματος σε δέντρο διεργασιών, αναγκαζόμαστε να ακολουθούμε το ρυθμό ενός επεξεργαστή και της σειράς του κώδικα.

ΤΕΛΟΣ ΑΝΑΦΟΡΑΣ