Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

Άσκηση 2:

Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

ΟΜΑΔΑ: oslab100

Ασπρογέρακας Ιωάννης (03118942)

Κανατάς Άγγελος-Νικόλαος (03119169)

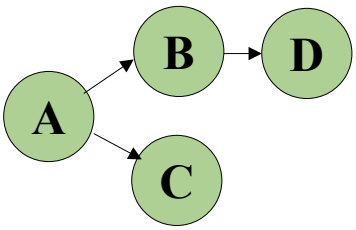
ΜΕΡΟΣ 1^{Ο} (Διαχείριση διεργασιών)

Σε αυτό το μέρος, δημιουργούμε **δέντρα διεργασιών** ως εξής:

- Στην ***Άσκηση 1.1*** δημιουργούμε ένα δεδομένο δέντρο διεργασιών που είναι *hard-coded* στον κώδικά μας.
- Στην ***Άσκηση 1.2*** επεκτείνουμε τον κώδικά μας για την δημιουργία αυθαίρετου δέντρου διεργασιών βάσει αρχείου εισόδου.

Άσκηση 1.1 (Δημιουργία δεδομένου δέντρου διεργασιών)

Το δέντρο διεργασιών που θα δημιουργήσουμε είναι αυτό που φαίνεται παρακάτω:



Ο **πηγαίος κώδικας (source code)** για την άσκηση φαίνεται παρακάτω:

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

#include "../helpers/proc-common.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*
 * create this process tree:
 * A-+-B---D
 *   `-C
 */
void fork_procs(void)
{
        /*
         * root of the process tree is A
         */

        //i'm A
        pid_t Bpid, Cpid, Dpid, killed_child_pid;      //we can use only one pid_t variable
        int wstatus;                                   //but for comprehension reasons we
        change_pname("A"); //for pstree                //do this
        printf("A: Starting...\n");
        Bpid=fork();
        if(Bpid<0) {
                perror("fork");
                exit(1);
        }
        if(Bpid==0) { //i'm  B
                change_pname("B");
                printf("B: Starting...\n");
                Dpid=fork();
                if(Dpid<0) {
                        perror("fork");
                        exit(1);
                }
                if(Dpid==0) { //i'm D                   //the format of the code resembles
                        change_pname("D");             //DF traversal of the process tree
                        printf("D: Starting...\n");    //, but the real sequence in which the
                        printf("D: Sleeping...\n");    //processes start, wait and exit is
                        sleep(SLEEP_PROC_SEC);         //random-like and depends on the OS
                        printf("D: Exiting...\n");     //scheduler
                        exit(13);
                }
                printf("B: Waiting...\n"); //B waiting for D to die
                Dpid=wait(&wstatus);
```

```
                if(Dpid==-1) {
                        perror("wait");
                        exit(1);
                }
                explain_wait_status(Dpid, wstatus);
                printf("B: Exiting...\n");
                exit(19);
        }
        Cpid=fork();
        if(Cpid<0) {
                perror("fork");
                exit(1);
        }
        if(Cpid==0) { //i'm C
                change_pname("C");
                printf("C: Starting...\n");
                printf("C: Sleeping...\n");
                sleep(SLEEP_PROC_SEC);
                printf("C: Exiting...\n");
                exit(17);
        }
        printf("A: Waiting...\n"); //A waiting for B and C to die
        int i;
        for(i=0; i<2; i++) {
                killed_child_pid=wait(&wstatus);
                if(killed_child_pid==-1) {
                        perror("wait");
                        exit(1);
                }
                explain_wait_status(killed_child_pid, wstatus);
        }
        printf("A: Exiting...\n");
        exit(16);
}

/*
 * the initial process (our program) forks the root of the process tree (A),
 * waits for the process tree to be completely created (with sleep()),
 * then takes a photo of it using show_pstree() and waits for A to die.
 *
 * how to wait for the process tree to be ready?
 *  ->sleep for a few seconds, hope for the best.
 *
 * also the children leaf processes stay active (they sleep()) for a few seconds (>sleep
 * time of the initial process), so the user can see the whole process tree created.
 *
 * the parent processes wait until all of their children die.
*/

int main(void)
{
        pid_t pid_root;
        int wstatus;

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

```
                fork_procs();
        }

        /*
         * father (initial process)
         */
        sleep(SLEEP_TREE_SEC);

        /* print the process tree with root having pid_root */
        show_pstree(pid_root);

        /* wait for the root of the process tree to terminate */
        printf("initial process: Waiting...\n");
        pid_root=wait(&wstatus);
        if(pid_root==-1) {
                perror("main: wait");
                exit(1);
        }
        explain_wait_status(pid_root, wstatus);
        printf("initial process: All done, exiting...\n");

        return 0;
}
```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Παρακάτω φαίνεται και το **Makefile** που έχουμε φτιάξει:

```
.PHONY: hardcoded-process-tree run clean


hardcoded-process-tree: hardcoded-process-tree.o ../helpers/proc-common.o
        gcc -o hardcoded-process-tree hardcoded-process-tree.o ../helpers/proc-common.o

hardcoded-process-tree.o: hardcoded-process-tree.c ../helpers/proc-common.h
        gcc -Wall -c hardcoded-process-tree.c

run:
        ./hardcoded-process-tree

clean:
        rm -f hardcoded-process-tree.o hardcoded-process-tree
```

Η διαδικασία μεταγλώττισης και σύνδεσης ώστε να δημιουργήσουμε το εκτελέσιμο **hardcoded-process-tree** καθώς και η έξοδος εκτέλεσής του είναι η εξής (στο directory **helpers** έχουμε όλα τα βοηθητικά .c/.o/.h αρχεία που μας έχουν δοθεί):

```
oslaba100@os-node2:~/ex2_a/1.1$ ls
Makefile  hardcoded-process-tree.c
oslaba100@os-node2:~/ex2_a/1.1$ make
gcc -Wall -c hardcoded-process-tree.c
gcc -o hardcoded-process-tree hardcoded-process-tree.o ../helpers/proc-common.o
oslaba100@os-node2:~/ex2_a/1.1$ ls
Makefile  hardcoded-process-tree  hardcoded-process-tree.c  hardcoded-process-tree.o
oslaba100@os-node2:~/ex2_a/1.1$ ./hardcoded-process-tree
A: Starting...
B: Starting...
A: Waiting...
C: Starting...
C: Sleeping...
B: Waiting...
D: Starting...
D: Sleeping...


A(22544)─┬─B(22545)──D(22547)
         └─C(22546)


initial process: Waiting...
C: Exiting...
D: Exiting...
My PID = 22544: Child PID = 22546 terminated normally, exit status = 17
My PID = 22545: Child PID = 22547 terminated normally, exit status = 13
B: Exiting...
My PID = 22544: Child PID = 22545 terminated normally, exit status = 19
A: Exiting...
My PID = 22543: Child PID = 22544 terminated normally, exit status = 16
initial process: All done, exiting...
```

Αξίζει να σημειώσουμε ότι η σειρά με την οποία εμφανίζονται τα μηνύματα έναρξης, αναμονής και τερματισμού των διεργασιών είναι ψιλό τυχαία, διαφέρει από εκτέλεση σε εκτέλεση και εξαρτάται από τον **χρονοδρομολογητή (scheduler)** του λειτουργικού συστήματος (αυτό θα το αναλύσουμε και παρακάτω στην ***Άσκηση 1.2***). Π.χ. για μία άλλη εκτέλεση του παραπάνω προγράμματος στο ίδιο μηχάνημα έχουμε:

```
oslaba100@os-node2:~/ex2_a/1.1$ make run
./hardcoded-process-tree
A: Starting...
A: Waiting...
B: Starting...
C: Starting...
C: Sleeping...
B: Waiting...
D: Starting...
D: Sleeping...


A(22614)─┬─B(22615)──D(22617)
         └─C(22616)


initial process: Waiting...
C: Exiting...
D: Exiting...
My PID = 22614: Child PID = 22616 terminated normally, exit status = 17
My PID = 22615: Child PID = 22617 terminated normally, exit status = 13
B: Exiting...
My PID = 22614: Child PID = 22615 terminated normally, exit status = 19
A: Exiting...
My PID = 22613: Child PID = 22614 terminated normally, exit status = 16
initial process: All done, exiting...
```

Παρατηρούμε, επίσης, ότι μάλλον πάντα τα μηνύματα έναρξης εμφανίζονται με **BF** τρόπο.

1.

Αν τερματίσουμε πρόωρα τη διεργασία **A**, δίνοντας **`kill -KILL <pid>`** (από ένα άλλο παράθυρο του terminal μας), όπου **`<pid>`** το Process ID της, τότε έχουμε (εκτελούμε τα παρακάτω στο περιβάλλον μας - Ubuntu):

Γνωρίζουμε θεωρητικά ότι όταν ένα parent process πεθάνει πρώτα από τα "παιδιά" του, η διεργασία **init** (PID=1) κληρονομεί όλα τα "ορφανά" children processes του, η οποία κάνει συνεχώς *wait()*. Παρατηρούμε, λοιπόν, παραπάνω το κατάλληλο μήνυμα τερματισμού της διεργασίας A (η διεργασία τερματίστηκε από το **KILL (9)** signal που στείλαμε μέσω του λειτουργικού), ενώ παρακάτω φαίνεται με περισσότερη λεπτομέρεια τι συμβαίνει πρακτικά:

(Για να μπορούμε να δούμε το δέντρο διεργασιών που φαίνεται παραπάνω πειράξαμε στον κώδικά μας την χρονική διάρκεια που οι διεργασίες-φύλλα κοιμούνται).

Παρατηρούμε, λοιπόν, ότι οι διεργασίες B και C "υιοθετούνται" από την **systemd(7031)** που είναι παιδί της **init** (*systemd(1)*), παρά από την init που αναφέραμε παραπάνω θεωρητικά. Αυτό είναι αρκετά ενδιαφέρον και συμβαίνει διότι θεωρούμε την init ανά **session** του χρήστη (?).

2.

Τώρα, τροποποιούμε την main() έτσι ώστε αντί για show_pstree(pid_root) να κάνουμε **show_pstree(getpid())**. Έτσι, έχουμε το παρακάτω δέντρο διεργασιών:

```
hardcoded-proce(1404136)─┬─A(1404137)─┬─B(1404138)───D(1404140)
                         │            └─C(1404139)
                         └─sh(1404141)───pstree(1404142)
```

Παρατηρούμε, λοιπόν, ότι τώρα στο δέντρο εμφανίζονται και οι διεργασίες **sh** και **pstree**. Η ρίζα του είναι το πρόγραμμά μας (αφού κάναμε *show_pstree(getpid())*), που δημιουργεί το δεδομένο δέντρο διεργασιών της άσκησης που αναφέραμε παραπάνω. Εκτός από αυτό, καλώντας την **show_pstree()**, εμφανίζει το δέντρο διεργασιών με ρίζα μία δεδομένη διεργασία.

Για να καταλάβουμε τι συμβαίνει βλέπουμε τον κώδικα της συνάρτησης **show_pstree()**:

```
void
show_pstree(pid_t p)
{
        int ret;
        char cmd[1024];

        snprintf(cmd, sizeof(cmd), "echo; echo; pstree -G -c -p %ld; echo; echo",
                (long)p);
        cmd[sizeof(cmd)-1] = '\0';
        ret = system(cmd);
        if (ret < 0) {
                perror("system");
                exit(104);
        }
}
```

Αυτό που κάνει είναι να γράφει στον buffer cmd την εντολή **"echo; echo; pstree -G -c -p <pid>; echo; echo"**, η οποία έπειτα μέσω του buffer μπαίνει σαν όρισμα στην **system()**. Αυτό που κάνει η system() είναι να εκτελεί ένα shell command που της δίνεται ως όρισμα με τον κάτωθι τρόπο:

```
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 *) NULL);
```

Επομένως, η εντολή-διεργασία **pstree()** προστίθεται στο δέντρο διεργασιών, καθώς και ο **sh** (standard command language interpreter) που την κάνει interpret.

3.

Σε υπολογιστικά συστήματα πολλαπλών χρηστών, πολλές φορές ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης. Αυτό συμβαίνει, διότι ένας κακόβουλος χρήστης θα μπορούσε να "γεννάει" συνέχεια διεργασίες με εκθετικό ρυθμό σε έναν ατέρμονο βρόχο π.χ., κάτι που θα καταλήξει στην εξάντληση της CPU και στον κορεσμό του πίνακα διεργασιών του λειτουργικού συστήματος - αν γεμίσει ο πίνακας διεργασιών τότε δεν μπορεί να δημιουργηθεί άλλη διεργασία και επομένως να αποκριθεί το σύστημα σε κάποια είσοδο (στα Linux η fork() είναι υλοποιημένη με μία τεχνική *copy-on-write* και επομένως η μόνη "ποινή" για το υπολογιστικό σύστημα είναι ο χρόνος και η μνήμη που απαιτείται για την δημιουργία της δομής του παιδιού, την αντιγραφή του **πίνακα σελίδων** από τον πατέρα στο παιδί κ.λπ., οπότε γενικά δεν φτάνει σε κορεσμό η μνήμη του συστήματος με την επαναλαμβανόμενη δημιουργία διεργασιών).

Αυτό είναι γνωστό και ως **fork bomb (rabbit virus)** που είναι ένα είδος **DoS** attack. Ένα παράδειγμα ενός fork bomb στα Linux είναι: "`:(){ :|:& };:`". Έτσι, λοιπόν, ένας τρόπος να αποτρέψουμε μία τέτοια επίθεση είναι ο περιορισμός του αριθμού των διεργασιών που μπορεί να έχει ένας χρήστης.

Άσκηση 1.2 (Δημιουργία αυθαίρετου δέντρου διεργασιών)

Τώρα, επεκτείνουμε τον παραπάνω κώδικα για την δημιουργία αυθαίρετου δέντρου διεργασιών βάσει αρχείου εισόδου. Το format του αρχείου εισόδου περιγράφεται στο αρχείο "**proc.tree**".

Ο **πηγαίος κώδικας (source code)** της άσκησης φαίνεται παρακάτω:

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

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

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3


void fork_procs(struct tree_node *root)
{
        pid_t child_pid;
        int wstatus;
        change_pname(root->name); //for pstree
        printf("%s: Starting...\n", root->name);

        //current node is a leaf process (base case)
        if(root->nr_children==0) {
                printf("%s: Sleeping...\n", root->name);
                sleep(SLEEP_PROC_SEC);
                printf("%s: Exiting...\n", root->name);
                exit(13); //we assume exit status=13 for leaf processes
        } else { //current node is not a leaf process
                int i;
```

```
		for(i=0; i < root->nr_children; i++) {
			child_pid=fork();
			if(child_pid<0) {
				perror("fork");
				exit(1);
			}
			if(child_pid==0) { //i'm the child
				fork_procs(root->children+i); //recursive call
			}
		}
		//parent process waiting for all of its children to die
		printf("%s: Waiting...\n", root->name);
		for(i=0; i < root->nr_children; i++) {
			child_pid=wait(&wstatus);
			if(child_pid==-1) {
				perror("wait");
				exit(1);
			}
			explain_wait_status(child_pid, wstatus);
		}
		printf("%s: Exiting...\n", root->name);
		exit(7); //we assume exit status=7 for non-leaf processes
	}
}

/*
 * the initial process (our program) forks the root of the process tree,
 * waits for the process tree to be completely created (with sleep()),
 * then takes a photo of it using show_pstree() and waits for the root to die.
 *
 * how to wait for the process tree to be ready?
 *  ->sleep for a few seconds, hope for the best.
 *
 * also the children leaf processes stay active (they sleep()) for a few seconds (>sleep
 * time of the initial process), so the user can see the whole process tree created.
 *
 * the parent processes wait until all of their children die.
*/

int main(int argc, char **argv)
{
	pid_t pid_root;
	int wstatus;
	struct tree_node *root;

	if(argc!=2) { //check if arguments are OK
		fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
		exit(1);
	}

	root=get_tree_from_file(argv[1]); //the infile describes the process tree
	print_tree(root);

	/* fork root of process tree */
	pid_root=fork();
	if(pid_root<0) {
		perror("main: fork");
		exit(1);
	}
	if(pid_root==0) {
		/* child (root of process tree) */
		fork_procs(root);
```

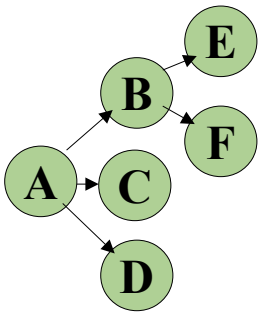
```
	}

	/*
	 * father (initial process)
	 */
	sleep(SLEEP_TREE_SEC);

	/* print the process tree with root having pid_root */
	show_pstree(pid_root);

	/* wait for the root of the process tree to terminate */
	printf("initial process: Waiting...\n");
	pid_root=wait(&wstatus);
	if(pid_root==-1) {
		perror("main: wait");
		exit(1);
	}
	explain_wait_status(pid_root, wstatus);
	printf("initial process: All done, exiting...\n");

	return 0;
}
```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Παρακάτω φαίνεται και το **Makefile** που έχουμε φτιάξει:

```
.PHONY: process-tree clean

process-tree: process-tree.o ../helpers/proc-common.o ../helpers/tree.o
	gcc -o process-tree process-tree.o ../helpers/proc-common.o ../helpers/tree.o

process-tree.o: process-tree.c ../helpers/proc-common.h ../helpers/tree.h
	gcc -Wall -c process-tree.c

ex1:
	./process-tree proc.tree

ex2:
	./process-tree test.tree  #1.1 example

ex3:
	./process-tree bad.tree

ex4:
	./process-tree proc.tree bad.tree

clean:
	rm -f process-tree.o process-tree
```

Η διαδικασία μεταγλώττισης και σύνδεσης ώστε να δημιουργήσουμε το εκτελέσιμο **process-tree** καθώς και η έξοδος εκτέλεσής του ενδεικτικά για διάφορα αρχεία εισόδου είναι η εξής:

- 


```
oslaba100@os-node1:~/ex2_a/1.2$ ls
Makefile  bad.tree  proc.tree  process-tree.c  test.tree
oslaba100@os-node1:~/ex2_a/1.2$ make
gcc -Wall -c process-tree.c
gcc -o process-tree process-tree.o ../helpers/proc-common.o ../helpers/tree.o
oslaba100@os-node1:~/ex2_a/1.2$ ls
Makefile  bad.tree  proc.tree  process-tree  process-tree.c  process-tree.o  test.tree
oslaba100@os-node1:~/ex2_a/1.2$ make ex1
./process-tree proc.tree
A
        B
                E
                F
        C
        D
A: Starting...
B: Starting...
C: Starting...
A: Waiting...
C: Sleeping...
D: Starting...
D: Sleeping...
E: Starting...
B: Waiting...
E: Sleeping...
F: Starting...
F: Sleeping...


A(27867)─┬─B(27868)─┬─E(27871)
         │          └─F(27872)
         ├─C(27869)
         └─D(27870)


initial process: Waiting...
C: Exiting...
D: Exiting...
E: Exiting...
F: Exiting...
My PID = 27867: Child PID = 27869 terminated normally, exit status = 13
My PID = 27867: Child PID = 27870 terminated normally, exit status = 13
My PID = 27868: Child PID = 27871 terminated normally, exit status = 13
My PID = 27868: Child PID = 27872 terminated normally, exit status = 13
B: Exiting...
My PID = 27867: Child PID = 27868 terminated normally, exit status = 7
A: Exiting...
My PID = 27866: Child PID = 27867 terminated normally, exit status = 7
initial process: All done, exiting...
```

- Δέντρο διεργασιών ***Άσκησης 1.1***:

```
oslaba100@os-node1:~/ex2_a/1.2$ make ex2
./process-tree test.tree
A
        B
                D
        C
A: Starting...
B: Starting...
A: Waiting...
C: Starting...
C: Sleeping...
B: Waiting...
D: Starting...
D: Sleeping...


A(27878)─┬─B(27879)──D(27881)
         └─C(27880)


initial process: Waiting...
C: Exiting...
D: Exiting...
My PID = 27878: Child PID = 27880 terminated normally, exit status = 13
My PID = 27879: Child PID = 27881 terminated normally, exit status = 13
B: Exiting...
My PID = 27878: Child PID = 27879 terminated normally, exit status = 7
A: Exiting...
My PID = 27877: Child PID = 27878 terminated normally, exit status = 7
initial process: All done, exiting...
```

• Όχι καλά δομημένο αρχείο εισόδου και λάθος αριθμός ορισμάτων αντίστοιχα:

```
oslaba100@os-node1:~/ex2_a/1.2$ make ex3
./process-tree bad.tree
expecting: D and got EOF
Makefile:16: recipe for target 'ex3' failed
make: *** [ex3] Error 1
oslaba100@os-node1:~/ex2_a/1.2$ make ex4
./process-tree proc.tree bad.tree
Usage: ./process-tree <input_tree_file>

Makefile:19: recipe for target 'ex4' failed
make: *** [ex4] Error 1
```

1.

Όπως αναφέραμε και στην ***Άσκηση 1.1*** η σειρά με την οποία εμφανίζονται τα μηνύματα έναρξης, αναμονής και τερματισμού των διεργασιών είναι ψιλό τυχαία, διαφέρει από εκτέλεση σε εκτέλεση και εξαρτάται από τον **χρονοδρομολογητή (scheduler)** του λειτουργικού συστήματος.

Π.χ. για μία άλλη εκτέλεση του προγράμματος με αρχείο εισόδου το *"proc.tree"* στο ίδιο μηχάνημα έχουμε:

```
./process-tree proc.tree
A
        B
                E
                F
        C
        D
A: Starting...
B: Starting...
A: Waiting...
C: Starting...
C: Sleeping...
D: Starting...
D: Sleeping...
B: Waiting...
E: Starting...
E: Sleeping...
F: Starting...
F: Sleeping...


A(1459024)─┬─B(1459025)─┬─E(1459028)
           │            └─F(1459029)
           ├─C(1459026)
           └─D(1459027)


initial process: Waiting...
C: Exiting...
E: Exiting...
D: Exiting...
F: Exiting...
My PID = 1459025: Child PID = 1459028 terminated normally, exit status = 13
My PID = 1459024: Child PID = 1459026 terminated normally, exit status = 13
My PID = 1459024: Child PID = 1459027 terminated normally, exit status = 13
My PID = 1459025: Child PID = 1459029 terminated normally, exit status = 13
B: Exiting...
My PID = 1459024: Child PID = 1459025 terminated normally, exit status = 7
A: Exiting...
My PID = 1459023: Child PID = 1459024 terminated normally, exit status = 7
initial process: All done, exiting...
```

Παρατηρούμε, επίσης, ότι τα μηνύματα έναρξης των διεργασιών εμφανίζονται μάλλον πάντα με **BF** τρόπο. Αυτό συμβαίνει μάλλον λόγω της πολιτικής που ακολουθεί ο scheduler και του τρόπου προφανώς που έχουμε γράψει τον κώδικά μας. Η σειρά των μηνυμάτων τερματισμού των διεργασιών είναι ψιλό τυχαία και εξαρτάται από τον scheduler.

ΜΕΡΟΣ 2^{Ο} (Διαδιεργασιακή επικοινωνία)

Σε αυτό το μέρος:

- Στην ***Άσκηση 1.3*** επεκτείνουμε τον κώδικα της 1.2, έτσι ώστε να ενεργοποιούνται με συγκεκριμένη σειρά οι διεργασίες (DF), χρησιμοποιώντας **σήματα**.
- Στην ***Άσκηση 1.4*** επεκτείνουμε τον κώδικα της 1.2, με σκοπό την εκτέλεση παράλληλου υπολογισμού αριθμητικής έκφρασης, χρησιμοποιώντας **σωληνώσεις** για την διάδοση των ενδιαμέσων αποτελεσμάτων.

Άσκηση 1.3 (Αποστολή και χειρισμός σημάτων)

Τώρα, θα επεκτείνουμε τον παραπάνω κώδικα της ***Άσκησης 1.2***, έτσι ώστε οι διεργασίες να ελέγχονται μέσω σημάτων για να εκτυπώνουν μηνύματα ενεργοποίησης *κατά βάθος* (**Depth-First**).

Ο **πηγαίος κώδικας (source code)** για την άσκηση φαίνεται παρακάτω:

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

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

void fork_procs(struct tree_node *root)
{
	int wstatus;
	change_pname(root->name); //for pstree
	printf("%s: Starting...\n", root->name);

	//current node is a leaf process (base case)
	if(root->nr_children==0) {
		if(raise(SIGSTOP)!=0) {
			perror("raise");
			exit(1);
		}
		printf("%s is awake!\n", root->name);
		exit(13); //we assume exit status=13 for leaf processes
	} else { //current node is not a leaf process
		pid_t pid[root->nr_children]; //now we must remember our children to
		int i;                        //wake them up later
		for(i=0; i < root->nr_children; i++) {
			pid[i]=fork();
			if(pid[i]<0) {
				perror("fork");
				exit(1);
			}
			if(pid[i]==0) { //i'm the child
				fork_procs(root->children+i); //recursive call
			}
		}
		/* wait for all children to stop, then stop yourself */
		wait_for_ready_children(root->nr_children);
		if(raise(SIGSTOP)!=0) {
```

```
                perror("raise");
                exit(1);
        }                                       //this recursively leads to DF
        printf("%s is awake!\n", root->name); //ordering of the awake messages!
        /* wake up each child and then wait for it to die :( */
        for(i=0; i < root->nr_children; i++) {
                if(kill(pid[i], SIGCONT)<0) {
                        perror("kill");
                        exit(1);
                }
                if(wait(&wstatus)==-1) {         //the ordering of the
                        perror("wait");          //diagnostic terminate
                        exit(1);                 //messages is deterministic
                }                               //now (DF postorder traversal)!
                explain_wait_status(pid[i], wstatus);
        }
        exit(7); //we assume exit status=7 for non-leaf processes
    }
}

/*
 * the initial process (our program) forks the root of the process tree,
 * waits for the process tree to be completely created (with wait_for_ready_children())
 * ,then takes a photo of it using show_pstree() and after it wakes up the root of the
 * process tree waits for it to die.
 *
 * how to wait for the process tree to be ready?
 *  ->use wait_for_ready_children() to wait until
 *    all of the children processes recursively are
 *    created.
 *
 * every leaf process raises SIGSTOP and waits to continue (print awake message and
 * then exit).
 *
 * every non-leaf process creates its children processes, waits for all children
 * to stop, then stops with raise(SIGSTOP) until it receives SIGCONT signal, prints
 * awake message and wakes up each child then waiting for it to die. This recursively
 * leads to DF ordering of the awaking messages.
 */

int main(int argc, char **argv)
{
        pid_t pid_root;
        int wstatus;
        struct tree_node *root;

        if(argc!=2) { //check if arguments are OK
                fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
                exit(1);
        }

        root = get_tree_from_file(argv[1]); //the infile describes the process tree
        print_tree(root);

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

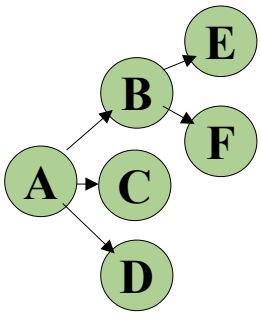
```
                fork_procs(root);
        }

        /*
         * father (initial process)
         */
        wait_for_ready_children(1);

        /* print the process tree with root having pid_root */
        show_pstree(pid_root);

        /* wake up the root of the process tree */
        if(kill(pid_root, SIGCONT)<0) {
                perror("main: kill");
                exit(1);
        }

        /* wait for the root of the process tree to terminate */
        if(wait(&wstatus)==-1) {
                perror("main: wait");
                exit(1);
        }
        explain_wait_status(pid_root, wstatus);

        return 0;
}
```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Παρακάτω φαίνεται και το **Makefile** που έχουμε φτιάξει:

```
.PHONY: process-tree-sig clean

process-tree-sig: process-tree-sig.o ../helpers/proc-common.o ../helpers/tree.o
        gcc -o process-tree-sig process-tree-sig.o ../helpers/proc-common.o ../helpers/tree.o

process-tree-sig.o: process-tree-sig.c ../helpers/proc-common.h ../helpers/tree.h
        gcc -Wall -c process-tree-sig.c

ex1:
        ./process-tree-sig proc.tree

ex2:
        ./process-tree-sig test.tree    #1.1 example

ex3:
        ./process-tree-sig bad.tree

ex4:
        ./process-tree-sig proc.tree bad.tree

clean:
        rm -f process-tree-sig.o process-tree-sig
```

Η διαδικασία μεταγλώττισης και σύνδεσης ώστε να δημιουργήσουμε το εκτελέσιμο **process-tree-sig** καθώς και η έξοδος εκτέλεσής του ενδεικτικά για διάφορα αρχεία εισόδου είναι η εξής:

- 


```
oslaba100@os-node2:~/ex2_a/1.3$ ls
Makefile  bad.tree  proc.tree  process-tree-sig.c  test.tree
oslaba100@os-node2:~/ex2_a/1.3$ make
gcc -Wall -c process-tree-sig.c
gcc -o process-tree-sig process-tree-sig.o ../helpers/proc-common.o ../helpers/tree.o
oslaba100@os-node2:~/ex2_a/1.3$ make ex1
./process-tree-sig proc.tree
A
        B
                E
                F
        C
        D
A: Starting...
B: Starting...
C: Starting...
D: Starting...
My PID = 1611: Child PID = 1613 has been stopped by a signal, signo = 19
My PID = 1611: Child PID = 1614 has been stopped by a signal, signo = 19
F: Starting...
E: Starting...
My PID = 1612: Child PID = 1615 has been stopped by a signal, signo = 19
My PID = 1612: Child PID = 1616 has been stopped by a signal, signo = 19
My PID = 1611: Child PID = 1612 has been stopped by a signal, signo = 19
My PID = 1610: Child PID = 1611 has been stopped by a signal, signo = 19


A(1611)─┬─B(1612)─┬─E(1615)
        │         └─F(1616)
        ├─C(1613)
        └─D(1614)


A is awake!
B is awake!
E is awake!
My PID = 1612: Child PID = 1615 terminated normally, exit status = 13
F is awake!
My PID = 1612: Child PID = 1616 terminated normally, exit status = 13
My PID = 1611: Child PID = 1612 terminated normally, exit status = 7
C is awake!
My PID = 1611: Child PID = 1613 terminated normally, exit status = 13
D is awake!
My PID = 1611: Child PID = 1614 terminated normally, exit status = 13
My PID = 1610: Child PID = 1611 terminated normally, exit status = 7
```

- Δέντρο διεργασιών ***Άσκησης 1.1***:

```
oslaba100@os-node1:~/ex2_a/1.3$ ls
Makefile  bad.tree  proc.tree  process-tree-sig.c  test.tree
oslaba100@os-node1:~/ex2_a/1.3$ make
gcc -Wall -c process-tree-sig.c
gcc -o process-tree-sig process-tree-sig.o ../helpers/proc-common.o ../helpers/tree.o
oslaba100@os-node1:~/ex2_a/1.3$ make ex2
./process-tree-sig test.tree    #1.1 example
A
        B
                D
        C
A: Starting...
B: Starting...
C: Starting...
My PID = 3639: Child PID = 3641 has been stopped by a signal, signo = 19
D: Starting...
My PID = 3640: Child PID = 3642 has been stopped by a signal, signo = 19
My PID = 3639: Child PID = 3640 has been stopped by a signal, signo = 19
My PID = 3638: Child PID = 3639 has been stopped by a signal, signo = 19


A(3639)─┬─B(3640)──D(3642)
        └─C(3641)


A is awake!
B is awake!
D is awake!
My PID = 3640: Child PID = 3642 terminated normally, exit status = 13
My PID = 3639: Child PID = 3640 terminated normally, exit status = 7
C is awake!
My PID = 3639: Child PID = 3641 terminated normally, exit status = 13
My PID = 3638: Child PID = 3639 terminated normally, exit status = 7
```

- Όχι καλά δομημένο αρχείο εισόδου και λάθος αριθμός ορισμάτων αντίστοιχα:

```
oslaba100@os-node1:~/ex2_a/1.3$ ls
Makefile  bad.tree  proc.tree  process-tree-sig.c  test.tree
oslaba100@os-node1:~/ex2_a/1.3$ make
gcc -Wall -c process-tree-sig.c
gcc -o process-tree-sig process-tree-sig.o ../helpers/proc-common.o ../helpers/tree.o
oslaba100@os-node1:~/ex2_a/1.3$ make ex3
./process-tree-sig bad.tree
expecting: D and got EOF
Makefile:16: recipe for target 'ex3' failed
make: *** [ex3] Error 1
oslaba100@os-node1:~/ex2_a/1.3$ make ex4
./process-tree-sig proc.tree bad.tree
Usage: ./process-tree-sig <input_tree_file>

Makefile:19: recipe for target 'ex4' failed
make: *** [ex4] Error 1
```

Τώρα, παρατηρούμε ότι τα μηνύματα ενεργοποίησης (… ***is awake!***), καθώς και τα διαγνωστικά μηνύματα τερματισμού εμφανίζονται με **DF** τρόπο, όπως θέλαμε, ενώ η σειρά που "μπλέκονται" είναι και αυτή προκαθορισμένη. Συγκεκριμένα, για τα μηνύματα ενεργοποίησης έχουμε **DF preorder traversal** και για τα διαγνωστικά μηνύματα τερματισμού έχουμε **DF postorder traversal**. Για τα μηνύματα που εμφανίζονται πάνω από την εμφάνιση του δέντρου με την show_pstree() (μηνύματα έναρξης και διαγνωστικά μηνύματα παύσης), ισχύει ότι έχουμε αναφέρει και παραπάνω στην άσκηση 1.2 (δεν είναι ντετερμινιστική η σειρά και εξαρτάται από τον scheduler – τα μηνύματα έναρξης εμφανίζονται μάλλον πάντα με BF traversal).

1.

Στις προηγούμενες ασκήσεις (1.1 και 1.2) χρησιμοποιήσαμε τη **sleep()** για τον συγχρονισμό των διεργασιών. Τώρα, τον συγχρονισμό των διεργασιών τον υλοποιούμε βάσει της διαδιεργασιακής επικοινωνίας και πιο συγκεκριμένα με την αποστολή και τον χειρισμό σημάτων. Ας θυμηθούμε ότι προηγουμένως με την sleep() ορίζαμε αυθαίρετα ***sleeping time*** με συγκεκριμένο τρόπο (*sleeping_time_leaf_process>sleeping_time_initial_process*), όπου ελπίζαμε να είναι αρκετά ώστε ο χρήστης να μπορέσει να δει το πλήρες δέντρο διεργασιών μετά την κλήση της show_pstree().

Προφανώς, αυτός ο τρόπος δεν είναι και τόσο ακριβής, σε αντίθεση με τον συγχρονισμό που επιτυγχάνεται με την χρήση σημάτων. Αρχικά, όπως αναφέραμε και παραπάνω, με την χρήση σημάτων μπορούμε να ελέγξουμε κατά κάποιο τρόπο τις διεργασίες. Επιπλέον, παρατηρούμε ότι το πρόγραμμά μας αποκρίνεται γρήγορα σε αντίθεση με την αποκρισιμότητα που έχουμε χρησιμοποιώντας sleep(). Επιπλέον, με την αύξηση του μεγέθους του δέντρου διεργασιών, απαιτείται, αν υλοποιούμε τον συγχρονισμό με χρήση sleep(), η κατάλληλη τροποποίηση των *sleeping time*, με αποτέλεσμα όσο μεγαλύτερο είναι το μέγεθος του δέντρου τόσο μικρότερη είναι η αποκρισιμότητα του προγράμματος μας.

Έτσι, με την χρήση σημάτων το πρόγραμμά μας είναι πιο:

- **robust**
- **scalable**
- **responsive**

- **controllable**, όσων αφορά τον έλεγχο των διεργασιών.

2.

Ο κώδικας της συνάρτησης **wait_for_ready_children()** φαίνεται παρακάτω:

```
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);
                }
        }
}
```

Αυτό που κάνει ουσιαστικά είναι να περιμένει όλα τα παιδιά της γονικής διεργασίας που κάλεσε την συνάρτηση να σταματήσουν. Με την χρήση της επιτυγχάνονται τα εξής:

- Ο χρήστης μπορεί να δει το πλήρες δέντρο διεργασιών, αφού κάθε non-leaf process περιμένει τα παιδιά της να σταματήσουν (και επομένως να δημιουργηθούν) και έπειτα σταματάει και το ίδιο, κάτι που αναδρομικά οδηγεί στο να έχουμε το πλήρες δέντρο διεργασιών έτοιμο και "μπλοκαρισμένο" να τυπωθεί από το πρόγραμμά μας με την show_pstree(), η οποία μετά κάνει "trigger" την ρίζα του δέντρου διεργασιών και αναδρομικά το "ξεμπλοκάρει", έτσι ώστε να το ελέγξουμε με συγκεκριμένο τρόπο.
- Εξασφαλίζουμε έναν σχετικά αξιόπιστο μηχανισμό επικοινωνίας μεταξύ των διεργασιών, χωρίς ταυτόχρονη λήψη σημάτων, race conditions και χωρίς την αποστολή σημάτων σε διεργασίες που πιθανώς δεν έχουν καν δημιουργηθεί ακόμα. Π.χ. ένα πιθανό ενδεχόμενο αν δεν χρησιμοποιήσουμε την *wait_for_ready_children()* είναι: η διεργασία πατέρας στέλνει το σήμα **SIGCONT** στην διεργασία παιδί, η οποία αν και έχει δημιουργηθεί δεν έχει κάνει παύση με **raise(SIGSTOP)** ακόμα. Σε αυτήν την περίπτωση η διεργασία παιδί θα παραμείνει για πάντα "μπλοκαρισμένη" μετά την παύση της.

Άσκηση 1.4 (Παράλληλος υπολογισμός αριθμητικής έκφρασης)

Τέλος, επεκτείνουμε τον κώδικα της ***Άσκησης 1.2***, ώστε να υπολογίζει δέντρα που αναπαριστούν αριθμητικές εκφράσεις, μέσω των **σωληνώσεων (pipes)**. Η αποτίμηση της έκφρασης θα γίνεται παράλληλα, δημιουργώντας μία διεργασία για κάθε κόμβο του δέντρου.

Ο **πηγαίος κώδικας (source code)** για την άσκηση φαίνεται παρακάτω:

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

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

#define SLEEP_PROC_SEC 1

void fork_procs(struct tree_node *root, int fdw)
{
        pid_t child_pid;
        int wstatus;
        change_pname(root->name); //for pstree
        printf("%s: Starting...\n", root->name);

        //current node is a leaf - number
        if(root->nr_children==0) {
                int value=atoi(root->name); //write to parent
                if(write(fdw, &value, sizeof(value))!=sizeof(value)) {
                        perror("write to pipe");
                        exit(1);
                }                       //leaf processes sleep so the user can
                close(fdw);             //see the whole expression tree created
                sleep(SLEEP_PROC_SEC); //with show_pstree()
                exit(13); //we assume exit status=13 for leaf processes
        } else { //current node is operator - non leaf - subexpression
                int i;
                int pipefd[2];          //we need only one pipe because operators
                if(pipe(pipefd)==-1) { //'+' and '*' are commutative
                        perror("pipe");
                        exit(1);
                }
                for(i=0; i < root->nr_children; i++) {
                        child_pid=fork();
                        if(child_pid<0) {
                                perror("fork");
                                exit(1);
                        }
                        if(child_pid==0) { //i'm the child
                                close(pipefd[0]); /* recursive call */
                                fork_procs(root->children+i, pipefd[1]);
                        }
                }
                close(pipefd[1]);
                int value[2]; //wait for all children values
                for(i=0; i < root->nr_children; i++) {
                        if(read(pipefd[0], &value[i], sizeof(value[i]))!
=sizeof(value[i])) {
                                perror("read from pipe");
                                exit(1);
                        }
                }
                close(pipefd[0]);
                int result; //compute the result of the subexpression
                if(!strcmp(root->name, "+")) { //'+' operator
                        result=value[0]+value[1];
                }
                if(!strcmp(root->name, "*")) { //'*' operator
                        result=value[0]*value[1];
                }
```

```
                printf("Me: %ld, i have computed: %i %s %i = %i\n",
(long)getpid(), value[0], root->name, value[1], result);
                if(write(fdw, &result, sizeof(result))!=sizeof(result)) {
//write to parent
                        perror("write to pipe");
                        exit(1);
                }
                close(fdw);

                /* parent process waiting for all of its children to die :( */
                for(i=0; i < root->nr_children; i++) {
                        if(wait(&wstatus)==-1) {
                                perror("wait");
                                exit(1);
                        }
                        explain_wait_status(child_pid, wstatus);
                }
                exit(7); //we assume exit status=7 for non-leaf processes
        }
}

/*
 * we will parallel compute the arithmetic expression depicted in the expression
 * tree.
 *
 * every leaf process-number returns to its parent (operator) its value.
 *
 * every non-leaf process-operator-subexpression takes the values of its
 * children-operands, computes the result based on what operator it is
 * (+ or * here) and returns the result to its parent. This recursively
 * leads to the evaluation of the arithmetic expression in the initial
 * process.
 *
 * our approach is based on that if a process attemps to read from an empty
 * pipe, then read will block until data is available.
 */

int main(int argc, char **argv)
{
        int result; //the result of the expression depicted in the expression
tree
        int pipefd[2];
        pid_t pid_root;
        int wstatus;
        struct tree_node *root;

        if(argc!=2) { //check if arguments are OK
                fprintf(stderr, "Usage: %s <input_expr_tree_file>\n\n",
argv[0]);
                exit(1);
        }

        root = get_tree_from_file(argv[1]); //the infile describes the
expression tree
        print_tree(root);

        if(pipe(pipefd)==-1) {
                perror("main: pipe");
                exit(1);
        }

        /* fork root of process tree */
        pid_root=fork();
        if(pid_root<0) {
```

```
                perror("main: fork");
                exit(1);
        }
        if(pid_root==0) {
                /* child (root of process-expression tree) */
                close(pipefd[0]); //children write to parent
                fork_procs(root, pipefd[1]);
        }

        /*
         * father (initial process)
         */
        close(pipefd[1]); //parent reads from children

        if(read(pipefd[0], &result, sizeof(result))!=sizeof(result)) {
                perror("main: read from pipe");
                exit(1);        //read waits until someone writes to the
        }                       //pipe (this means that the initial
        close(pipefd[0]);       //process doesn't need to sleep())

        /* print the expression tree with root having pid_root */
        show_pstree(pid_root);

        /* wait for the root of the expression tree to terminate */
        if(wait(&wstatus)==-1) {
                perror("main: wait");
                exit(1);
        }
        explain_wait_status(pid_root, wstatus);

        printf("The result of the expression is: %i\n", result);

        return 0;
}
```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

(Θα μπορούσαμε αντί για **sleep()** να χρησιμοποιήσουμε **σήματα** για να μπορεί ο χρήστης να δει το πλήρες δέντρο με την *show_pstree()* - η λύση αυτή είναι ανεβασμένη και στον orion αλλά εδώ παραθέτουμε και αναλύουμε μόνο την λύση με sleep()).

Παρακάτω φαίνεται και το **Makefile** που έχουμε φτιάξει:

```
.PHONY: comp-expr-tree clean

comp-expr-tree: comp-expr-tree.o ../helpers/proc-common.o ../helpers/tree.o
	gcc -o comp-expr-tree comp-expr-tree.o ../helpers/proc-common.o ../helpers/tree.o

comp-expr-tree.o: comp-expr-tree.c ../helpers/proc-common.h ../helpers/tree.h
	gcc -Wall -c comp-expr-tree.c

ex1:
	./comp-expr-tree expr.tree   #10+4*(5+7)

ex2:
	./comp-expr-tree test.tree   #10*(5+6)

ex3:
	./comp-expr-tree bad.tree

ex4:
	./comp-expr-tree expr.tree test.tree

clean:
	rm -f comp-expr-tree.o comp-expr-tree
```

Η διαδικασία μεταγλώττισης και σύνδεσης ώστε να δημιουργήσουμε το εκτελέσιμο **comp-expr-tree** καθώς και η έξοδος εκτέλεσής του ενδεικτικά για διάφορα αρχεία εισόδου είναι η εξής:

- **10+4*(5+7)**:

```
oslaba100@os-node1:~/ex2/1.4$ ls
Makefile              bad.tree          expr.tree           test.tree
alt-comp-expr-tree.c  comp-expr-tree.c  nonparallel-comp.c
oslaba100@os-node1:~/ex2/1.4$ make
gcc -Wall -c comp-expr-tree.c
gcc -o comp-expr-tree comp-expr-tree.o ../helpers/proc-common.o ../helpers/tree.o
oslaba100@os-node1:~/ex2/1.4$ make ex1
./comp-expr-tree expr.tree   #10+4*(5+7)
+
        10
        *
                +
                        5
                        7
                4
+: Starting...
10: Starting...
*: Starting...
+: Starting...
4: Starting...
5: Starting...
7: Starting...
Me: 1165, i have computed: 5 + 7 = 12
Me: 1164, i have computed: 4 * 12 = 48
Me: 1162, i have computed: 10 + 48 = 58


+(1162)─┬─*(1164)─┬─+(1165)─┬─5(1167)
        │         │         └─7(1168)
        │         └─4(1166)
        └─10(1163)


My PID = 1162: Child PID = 1164 terminated normally, exit status = 13
My PID = 1164: Child PID = 1166 terminated normally, exit status = 13
My PID = 1165: Child PID = 1168 terminated normally, exit status = 13
My PID = 1165: Child PID = 1168 terminated normally, exit status = 13
My PID = 1164: Child PID = 1166 terminated normally, exit status = 7
My PID = 1162: Child PID = 1164 terminated normally, exit status = 7
My PID = 1161: Child PID = 1162 terminated normally, exit status = 7
The result of the expression is: 58
```

- Όχι καλά δομημένο αρχείο εισόδου και λάθος αριθμός ορισμάτων αντίστοιχα:

```
oslaba100@os-node1:~/ex2/1.4$ make ex3
./comp-expr-tree bad.tree
Unexpected empty line
Makefile:16: recipe for target 'ex3' failed
make: *** [ex3] Error 1
oslaba100@os-node1:~/ex2/1.4$ make ex4
./comp-expr-tree expr.tree test.tree
Usage: ./comp-expr-tree <input_expr_tree_file>

Makefile:19: recipe for target 'ex4' failed
make: *** [ex4] Error 1
```

- **10*(5+6)**:

```
oslaba100@os-node1:~/ex2/1.4$ make ex2
./comp-expr-tree test.tree   #10*(5+6)
*
        +
                5
                6
        10
*: Starting...
+: Starting...
10: Starting...
5: Starting...
6: Starting...
Me: 1336, i have computed: 5 + 6 = 11
Me: 1335, i have computed: 10 * 11 = 110


*(1335)─┬─+(1336)─┬─5(1338)
        │         └─6(1339)
        └─10(1337)


My PID = 1335: Child PID = 1337 terminated normally, exit status = 13
My PID = 1336: Child PID = 1339 terminated normally, exit status = 13
My PID = 1336: Child PID = 1339 terminated normally, exit status = 13
My PID = 1335: Child PID = 1337 terminated normally, exit status = 7
My PID = 1334: Child PID = 1335 terminated normally, exit status = 7
The result of the expression is: 110
```

- **2*(3+4)+6*(1+3)**:

```
./comp-expr-tree testagain.tree #2*(3+4)+6*(1+3)
+
        *
                2
                +
                        3
                        4
        *
                6
                +
                        1
                        3
+: Starting...
*: Starting...
*: Starting...
2: Starting...
6: Starting...
+: Starting...
+: Starting...
3: Starting...
1: Starting...
4: Starting...
3: Starting...
Me: 3797, i have computed: 3 + 4 = 7
Me: 3793, i have computed: 2 * 7 = 14
Me: 3798, i have computed: 1 + 3 = 4
Me: 3794, i have computed: 6 * 4 = 24
Me: 3792, i have computed: 14 + 24 = 38


+(3792)─┬─*(3793)─┬─+(3797)─┬─3(3799)
        │         │         └─4(3801)
        │         └─2(3795)
        └─*(3794)─┬─+(3798)─┬─1(3800)
                  │         └─3(3802)
                  └─6(3796)


My PID = 3793: Child PID = 3797 terminated normally, exit status = 13
My PID = 3794: Child PID = 3798 terminated normally, exit status = 13
My PID = 3797: Child PID = 3801 terminated normally, exit status = 13
My PID = 3798: Child PID = 3802 terminated normally, exit status = 13
My PID = 3797: Child PID = 3801 terminated normally, exit status = 13
My PID = 3798: Child PID = 3802 terminated normally, exit status = 13
My PID = 3793: Child PID = 3797 terminated normally, exit status = 7
My PID = 3794: Child PID = 3798 terminated normally, exit status = 7
My PID = 3792: Child PID = 3794 terminated normally, exit status = 7
My PID = 3792: Child PID = 3794 terminated normally, exit status = 7
My PID = 3791: Child PID = 3792 terminated normally, exit status = 7
The result of the expression is: 38
```

1.

Οι παραπάνω αριθμητικές εκφράσεις περιέχουν μόνο **αντιμεταθετικούς τελεστές (“+” και “*”)**. Έτσι, κάθε γονική διεργασία χρησιμοποιεί μόνο μία σωλήνωση για όλες τις διεργασίες παιδιά της, αφού δεν την ενδιαφέρει η σειρά με την οποία επιστρέφουν οι διεργασίες παιδιά τις τιμές τους σε αυτήν, λόγω της αντιμεταθετικότητας των τελεστών. Έτσι, κάθε διεργασία φύλλο (αριθμός) χρησιμοποιεί μία σωλήνωση - για να στείλει την τιμή της στην γονική διεργασία - τελεστή) - , ενώ κάθε non leaf διεργασία (τελεστής - αριθμητική υπο-έκφραση) χρησιμοποιεί δύο σωληνώσεις, μία για να λάβει τις τιμές των παιδιών της και μία για να στείλει την τιμή του αποτελέσματός της στην γονική της διεργασία. Αν θέλουμε να υπολογίζουμε και αριθμητικές εκφράσεις που περιέχουν **μη αντιμεταθετικούς τελεστές (‘-’ και ‘÷’)**, τότε κάθε γονική διεργασία θα πρέπει να χρησιμοποιεί τόσες σωληνώσεις όσες και ο αριθμός των διεργασιών παιδιών της (+ άλλη μία για να στείλει το αποτέλεσμά της στην διεργασία πατέρα της).

2.

Έστω ότι έχουμε ένα σύστημα πολλαπλών επεξεργαστών. Σε αυτό μπορούν να εκτελούνται παραπάνω από μία διεργασίες παράλληλα. Έτσι, για μία σχετικά μεγάλη αριθμητική έκφραση, η χρονική πολυπλοκότητα παράλληλης αποτίμησης της από δέντρο διεργασιών είναι μικρότερη έναντι της αποτίμησης της από μία μόνο διεργασία σειριακά, καθώς μπορούν ταυτόχρονα να αποτιμούνται υπο-εκφράσεις της αριθμητικής έκφρασης.