ΣΗΜΜΥ ΕΜΠ
Λειτουργικά Συστήματα / 2η εργαστηριακή αναφορά
Νικόλαος Πηγαδάς / el18445
Γιώργος Αγγέλης / el18030
6ο εξάμηνο/ Ακαδημαϊκό έτος 2020-2021

OS αναφορά 2

Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

2.1 Κατασκευή δεδομένου δέντρου διεργασιών

Πηγαίος κώδικας

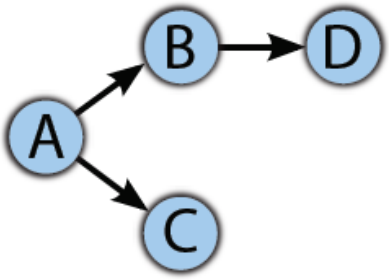
```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */
void fork_procs(void){
	/*
	 * initial process is A.
	 */
	int status;
	pid_t  pb, pc, pd;
	printf("A is being created...\n");
	printf("A creating: B...\n");
	pb = fork();

	if (pb < 0) {
		perror("pb: fork");
		exit(1);
	}
	else if (pb == 0) {
		/* Node B exists and creates node D*/
		pd = fork();
		if (pd < 0) {
			perror("pd: fork");
			exit(1);
		}
		else if (pd == 0) {
			/* D exists and is named */
			printf("B creating: D...\n");
			change_pname("D");
```

```
                printf("D: Sleeping...\n");
                sleep(SLEEP_PROC_SEC);
                printf("D: Exiting...\n");
                exit(13);
        }

        /*Edw mpainei to B meta to fork poy dhmioyrgei to D*/
                change_pname("B");
                /*To B elegxei to paidi toy, to D*/
                int p = wait(&status);
                explain_wait_status(p, status);
                printf("B: Sleeping...\n");
        //      sleep(SLEEP_PROC_SEC);, einai peritto logw wait
                printf("B: Exiting...\n");
                exit(19);

    }
    else{
        pc = fork();
        /*Apo edw synexizei to C*/
        if (pc < 0) {
                perror("pc: fork");
                exit(1);
        }
        else if (pc == 0) {
                /* C is being created and is named */
                printf("A creating: C...\n");
                change_pname("C");
                printf("C: Sleeping...\n");
                sleep(SLEEP_PROC_SEC);
                printf("C: Exiting...\n");
                exit(17);
        }
        /*Edw tha mpei to A meta to fork pou dhmioyrgei to B*/
                change_pname("A");
                /*Kanei 2 wait gia ta dyo toy paidia*/
//              raise( SIGKILL);
                int p = wait(&status);
                explain_wait_status(p, status);
                p = wait(&status);
                explain_wait_status(p, status);
                printf("A: Sleeping...\n");
        //      sleep(SLEEP_PROC_SEC);, einai peritto logw wait
                printf("A: Exiting...\n");
                exit(16);

    }
}

/*
 * 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);
        }

        /*
         * Father
         */
        /* for ask2-signals */
        /* wait_for_ready_children(1); */
        /* for ask2-{fork, tree} */
        sleep(SLEEP_TREE_SEC);
//       show_pstree(1);
        /* Print the process tree root at pid */
        show_pstree(pid);
        show_pstree(getpid());
        /* for ask2-signals */
        /* kill(pid, SIGCONT); */
        /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

Έξοδος και σύντομες απαντήσεις στις ερωτήσεις

Δημιουργία του ζητούμενου δέντρου:



Κωδικοί επιστροφής διεργασιών
A = 16, B = 19, C = 17, D = 13

Έξοδος ζητούμενου προγράμματος

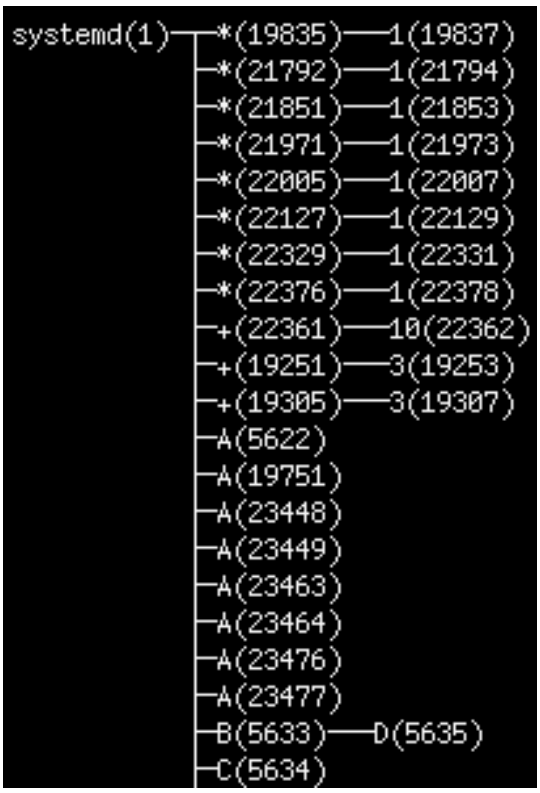
```
oslaba77@os-node1:~/code/forktree1.0/forktree$ ./ask2-fork1
A is being created...
A creating: B...
A creating: C...
B creating: D...
C: Sleeping...
D: Sleeping...


A(9359)─┬─B(9360)───D(9362)
        └─C(9361)


D: Exiting...
C: Exiting...
My PID = 9360: Child PID = 9362 terminated normally, exit status = 13
B: Sleeping...
B: Exiting...
My PID = 9359: Child PID = 9361 terminated normally, exit status = 17
My PID = 9359: Child PID = 9360 terminated normally, exit status = 19
A: Sleeping...
A: Exiting...
My PID = 9358: Child PID = 9359 terminated normally, exit status = 16
```

Σχόλιο για τις ασκήσεις 2.1, 2.2: Οι συναρτήσεις sleep διαδραμάτισαν καθοριστικό ρόλο στην δημιουργία του δέντρου, καθώς χωρίς αυτές, οι διεργασίες θα τερμάτιζαν χωρίς να έχουν προλάβει να αποτυπωθούν στο δέντρο διεργασιών. Το μειονέκτημα αυτής της μεθόδου, όμως, είναι ότι για να διασφαλίσουμε την ολοκληρωμένη αποτύπωση του δοσμένου δέντρου διεργασιών πρέπει κάθε φορά να προσαρμόζουμε κατάλληλα τα δευτερόλεπτα που θα κοιμάται κάθε κόμβος. Δηλαδή, η ορθότητα του προγράμματος αυτού εξαρτάται από τα διαφορετικά test cases. Το γεγονός αυτό φανερώνει την χρησιμότητα της συνάρτησης wait_children που χρησιμοποιείται σε επόμενη άσκηση.

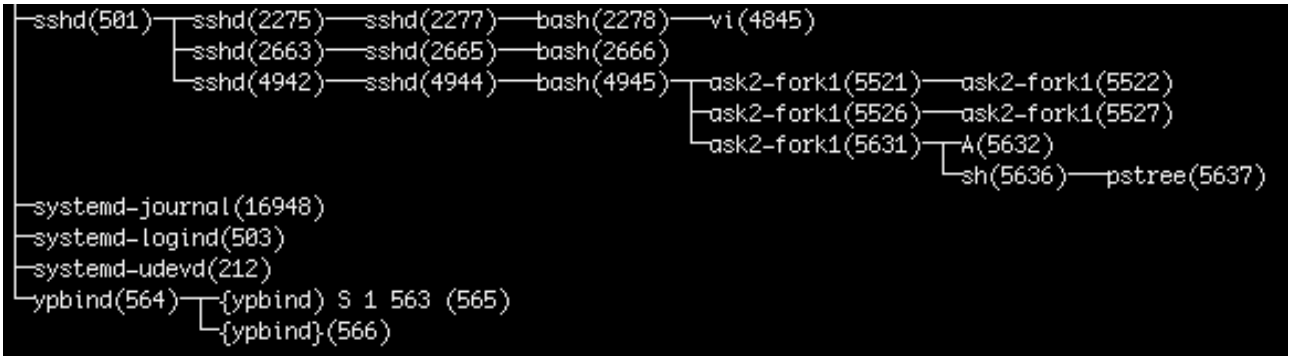
Απάντηση 1ης ερώτησης

Έξοδος με πρόωρο raise(SIGKILL) της διεργασίας A και εκτύπωση δέντρου με την χρήση της εντολής show_pstree(1), για την εκτύπωση έως και τις αρχικές διεργασίες. Η δεύτερη εντολή χρησιμοποιείται, επειδή με τον πρόωρο θάνατο του father/root A, τα παιδιά της B και C γίνονται zombies και γίνονται παιδιά της αρχικής διεργασίας systemd(1). Η systemd είναι μια διεργασία που όταν καλείται με PID = 1 αποκτά ρόλο αντίστοιχο αυτού της init, δηλαδή διαχειρίζεται την μνήμη και τις διεργασίες των χρηστών.

Ενδιαφέροντα τμήματα του εμφανιζόμενου δέντρου διεργασιών:



…
ενδιάμεσες διεργασίες
…



Παρατηρούμε ότι με τον θάνατο της A, η δημιουργία του δέντρου παγώνει και εμφανίζει μόνο τη ρίζα A (κάτω στιγμιότυπο). Ταυτόχρονα, όμως, έχουν δημιουργηθεί τα παιδιά του μέσω fork, πριν ο A πεθάνει. Τα παιδιά αυτά, λοιπόν, είναι zombies.
Όπως αναμένουμε από την θεωρία και βλέπουμε στο πάνω στιγμιότυπο του δέντρου, τα B και C έγιναν πράγματι παιδιά μιας αρχικής διεργασίας και όπως φαίνεται παρακάτω έκλεισαν.

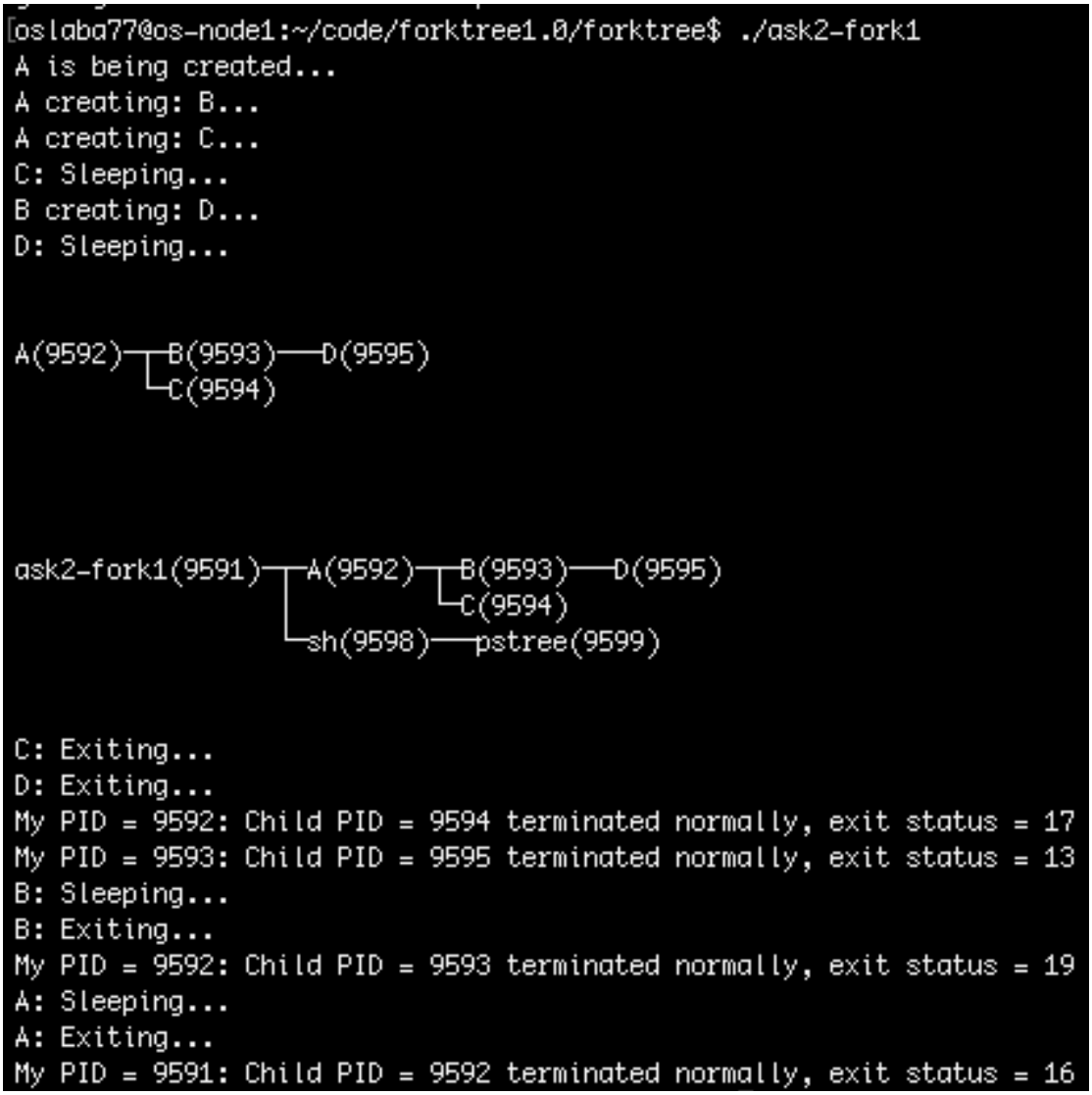
```
oslaba77@os-node1:~/code/forktree1.0/forktree$ D: Exiting...
C: Exiting...
My PID = 5633: Child PID = 5635 terminated normally, exit status = 13
B: Sleeping...
B: Exiting...
```

Σημείωση: Η αρχική διεργασία systemd(1) κάνει συνεχόμενα wait, περιμένοντας τις διεργασίες B και C να ολοκληρωθούν. Από την άλλη, τα παιδιά των C και B, εν προκειμένω το παιδί D του B, συνεχίζει κανονικά την λειτουργία του, εφόσον ο πατέρας του δεν έπαθε κάτι. Δηλαδή τα παιδιά των zombies δεν επηρεάζονται.

Συνοψίζοντας, αν αντί για την A γινόταν SIGKILL μιας άλλης διεργασίας, θα γινόντουσαν τα αντίστοιχα βήματα αντιμετώπισης αυτού του γεγονότος από το Λειτουργικό Συστήμα.

Απάντηση 2ης ερώτησης

Έξοδος, όταν ο πατέρας περικλείει και την εντολή show_pstree(getpid())



Με τη χρήση αυτής της εντολής εμφανίζεται δέντρο με ρίζα το pid μιας διεργασίας πριν τη διεργασία Α. Η προηγούμενη διεργασία που ξεκίνησε και στην οποία υπάγεται το δέντρο μας, είναι η διεργασία εκτέλεσης τους εκτελέσιμου προγράμματος που δημιουργήσαμε (ask2-fork1), έννοια συνυφασμένη με την συνάρτηση main(). Έτσι, δημιουργείται το δέντρο με ρίζα το εκτελέσιμο και τα παιδιά του. Αξίζει να σημειωθεί εδώ ότι εκτός από το Α, έχει παιδί και άλλη διεργασία, την sh με pid = 9598, η οποία έχει παιδί την pstree με pid = 9599. Εφόσον αυτές οι διεργασίες ανήκουν στο δέντρο διεργασιών του εκτελέσιμού μας, αποτελούν προφανώς διεργασίες που ανοίγουν και λειτουργούν στο πρασκήνιο, κατά την εκτέλεση του προγράμματος. Άρα, η μορφή του δέντρου είναι η αναμενόμενη, αφού μερικές διεργασίες που λειτουργούν στο παρασκήνιο, όπως η show_pstree, παραμένουν ανοιχτές μέχρι το τέλος του προγράμματος.

Απάντηση 3ης ερώτησης

Σε υπολογιστικά συστήματα πολλαπλών χρηστών, πολλές φορές ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης, γιατί αν κάποιος χρήστης άθελά του προξενήσει την συνεχή δημιουργία διεργασιών, θα πέσει το σύστημα για όλους τους χρήστες. Μάλιστα, συνήθως τίθενται όρια και για έναν χρήστη, γιατί το ενδεχόμενο για κάτι τέτοιο δεν παύει να υπάρχει. Τέλος, αξίζει να αναφέρουμε ότι η τακτική αυτή αποτρέπει κακόβουλους χρήστες από το να ρίξουν το σύστημα με ασταμάτητα fork.

Σημείωση: Πέρα από την ακεραιότητα του συστήματος υπάρχει και άλλο πλεονέκτημα. Ειδικότερα, ξέρουμε ότι ο χρόνος που παρέχει το ΛΣ στις διεργασίες να δεσμεύσουν το hardware είναι συγκεκριμένος, οι λεγόμενες χρονοσχισμές. Η μοιρασιά του χρόνου αυτού στις διεργασίες γίνεται με τον βέλτιστο τρόπο και όχι απαραίτητα ισότιμα (π.χ. αν μια διεργασία περιμένει δεδομένα από την μνήμη, ενώ έχει δεσμεύσει ταυτόχρονα τον επεξεργαστή, θα μπει στην ουρά διεργασιών σε αναμονή και όχι σε εκτέλεση. Αυτό γίνεται γιατί αλλιώς θα χάναμε πολλούς κύκλους λειτουργίας του επεξεργαστή χωρίς λόγο (η μνήμη είναι πολλές τάξεις μεγέθους πιο αργή από τον επεξεργαστή)). Και πάλι, όμως, οι χρονοσχισμές κάθε δεργασίας γίνονται πολύ μικρές, οπότε το σύστημα γίνεται εξαιρετικά αργό, γιατί θα είμαστε κοντά στο ανώτατο όριο των συνολικών διεργασιών. Καταληκτικά, ένας σχεδιαστής ΛΣ πρέπει να έχει στο μυαλό του, όχι μόνο ανώτατο όριο όλων των διεργασιών, αλλά και ένα ξεχωριστό για κάθε διεργασία και αναλόγως την διεργασία, για την αποφυγή αργής λειτουργίας του Η/Υ.
(Όπως για παράδειγμα όταν μια καρτέλα ενός browser καταναλώνει σχετικά μεγάλη ισχύ. Όταν περνάει ένα συγκεκριμένο χρονικό όριο, η διεργασία της σταματά και εμφανίζεται ένα μήνυμα που μας ενημερώνει ότι πρέπει να γίνει reload για να την ανοίξουμε ξανά).

2.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

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


static void __crocs(struct tree_node *root, int level){

        change_pname(root->name);
        printf("We are in %s\n", root->name);
        printf("%s has %d children and is patient\n", root->name, root->nr_children);
        pid_t pid;
        int status, i;
        /*Apo anadromi o root komvos einai goneas opote elegxoyme ta paidia toy, an exei*/
        for(i=0; i < root->nr_children; i++){
                pid = fork();
                if (pid < 0){
                        perror("fork");
                        exit(1);
                }
                /*Anadromi: An yparxei paidi kai einai mia xara, kalese thn synarthsh gia to paidi*/
                else if (pid == 0) {
                        __crocs(root->children+i, level+1);
                }
        }
        /*Efoson exoyn ginei ta fork, kanoyme META wait gia ta paidia ayta*/
        if(root->children > 0){
        //      printf("%s has %d children and is patient\n", root->name, root->nr_children);
                for (i=0; i < root->nr_children; i++){
                        pid = wait(&status);
                        explain_wait_status(pid, status);
                }
        }
        /*An den exei paidia, as koimithei gia na ftiaksoyme to dentro diergasiwn*/
        else{
                printf("I, %s, am sleeping\n", root->name);
                sleep(SLEEP_PROC_SEC);
        }
        printf("Goodbye %s, you will be missed.\n", root->name);
        exit(level);
}


void crocs(struct tree_node *root){
        __crocs(root, 1);
}


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

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

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

        pid_t pid;
        int status;
```

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

        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;
}
```

Έξοδος και σύντομη απαντηση στις ερωτήσεις

```
A
        B
                E
                F
        C
        D
We are in A
A has 3 children and is patient
We are in B
B has 2 children and is patient
We are in C
C has 0 children and is patient
I, C, am sleeping
We are in D
D has 0 children and is patient
I, D, am sleeping
We are in E
E has 0 children and is patient
I, E, am sleeping
We are in F
F has 0 children and is patient
I, F, am sleeping


A(16585)─┬─B(16586)─┬─E(16589)
         │          └─F(16590)
         ├─C(16587)
         └─D(16588)


Goodbye C, you will be missed.
My PID = 16585: Child PID = 16587 terminated normally, exit status = 2
Goodbye D, you will be missed.
My PID = 16585: Child PID = 16588 terminated normally, exit status = 2
Goodbye E, you will be missed.
Goodbye F, you will be missed.
My PID = 16586: Child PID = 16589 terminated normally, exit status = 3
My PID = 16586: Child PID = 16590 terminated normally, exit status = 3
Goodbye B, you will be missed.
My PID = 16585: Child PID = 16586 terminated normally, exit status = 2
Goodbye A, you will be missed.
My PID = 16584: Child PID = 16585 terminated normally, exit status = 1
```

Σχόλια

Περιγραφή ροής της υλοποίησης

Το πρώτο fork γίνεται στην main και δημιουργεί την ρίζα του δέντρου και μετά καλείται η αναδρομική συνάρτηση που δημιουργεί τους υπόλοιπους κόμβους, διατεταγμένους κατά βάθος.

Απάντηση της ερώτησης

Παρατηρούμε ότι τα μηνύματα έναρξης των διεργασιών εμφανίζονται ανά επίπεδο, κάτι που είναι αναμενόμενο, ωστόσο τα μηνύματα τερματισμού των διεργασιών, πέραν του ότι ακολουθούν τον βασικό κανόνα να μην τερματίζεται μια διεργασία γονέας αν δεν έχει τερματιστεί το παιδί της, εμφανίζονται με σειρά μη αναμενόμενη. Μάλιστα, η σειρά με την οποία εμφανίζονται τα μηνύματα τερματισμού διαφέρει από λειτουργικό σύστημα σε λειτουργικό σύστημα. Ειδικότερα, στο συγκεκριμένο παράδειγμα φαίνεται ότι τερματίζουν πρώτα οι κόμβοι χωρίς παιδιά, οι οποίοι ξεκινούν και πρώτοι (δηλαδή έχουμε C, D, E, F, B, A). Αυτό που αλλάζει από σύστημα σε σύστημα είναι ποιες διεργασίες θα μπουν πρώτες στην ουρά διεργασιών προς εκτέλεση (π.χ. θα μπορούσαμε να έχουμε E, F, C, D, B, A). Τέλος, το αποτέλεσμα επηρεάζεται από τους χρόνους που έχουμε ορίσει για τις κλήσεις της sleep.

2.3 Αποστολή και χειρισμός σημάτων

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"

static void __fork_procs(struct tree_node *root, int level)
{
        int i;
```

```
        pid_t pid_ar[root->nr_children];
        /*
         * Start
         */
        printf("PID = %ld, name %s, starting...\n", (long)getpid(), root->name);
        change_pname(root->name);

        /* ... */
        /*Apo anadromi o root komvos einai goneas opote elegxoyme ta paidia toy, an exei*/
        for(i = 0; i < root->nr_children; i++){
                pid_ar[i] = fork();
                if (pid_ar[i] < 0){
                        perror("fork");
                        exit(1);
                }
                else if(pid_ar[i] == 0){
                        __fork_procs(root->children + i, level+1);
                }
        }

        /*Edw o komvos kanei wait gia ola ta paidia toy, prin stamathsei o idios*/
        wait_for_ready_children(root->nr_children);

        /*
         * Suspend Self
         */

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

        /* ... */
        /*Edw oles koimountai, ektos apo ta fylla. Ksypname ta fylla  me DFS seira, anadromika,
        gia na kataskeyastei to dentro*/
        for(i = 0; i < root->nr_children; i++){
                kill(pid_ar[i], SIGCONT);
                int status, pid_status = wait(&status);
                explain_wait_status(pid_status, status);
        }
        /*
         * Exit
         */
        printf("PID = %ld, name = %s is exiting\n", (long)getpid(), root->name);
        exit(level);
}

void fork_procs(struct tree_node *root){
        __fork_procs(root, 1);
}
/*
 * 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;
}
```

Περιγραφή λύσης

Στο δέντρο οι διεργασίες-κόμβοι λαβάνουν σήμα να ξυπνήσουν και τερματίζουν ως διεργασίες σε σειρά DFS postorder, όπως φαίνεται και από τις εξόδους που έχουμε παραθέσει. Αυτό επιτυγχάνεται μέσω βοηθητικού πίνακα, στον οποίο κρατάμε τα pids των παιδιών κάθε κόμβου. Μέσα σε ένα for στον κώδικα, κάθε γονέας ξυπνάει ένα παιδί την φορά (μέσω SIGCONT) και μετά το περιμένει να τελειώσει. Με αυτόν τον τρόπο, φτάνουμε στα φύλλα και μετά πάμε προς τα πίσω, σε κάθε υποδέντρο και μετά στην ρίζα. Έτσι, επιτυγχάνεται το επιθυμητό αποτέλεσμα.

Το προαναφερθές for:

```
for(i = 0; i < root->nr_children; i++){
        kill(pid_ar[i], SIGCONT);
        int status, pid_status = wait(&status);
        explain_wait_status(pid_status, status);
}
```

Περιγραφή ροής της υλοποίησης

Το πρώτο fork γίνεται στην main και δημιουργεί την ρίζα του δέντρου και μετά καλείται η αναδρομική συνάρτηση που δημιουργεί τους υπόλοιπους κόμβους. Όλες οι διαδικασίες-κόμβοι του δέντρου περιμένουν τα παιδιά τους και σταματάνε την λειτουργία τους, μέσω της raise(SIGSTOP). Ταυτόχρονα και ο πρώτος πατέρας, ο πατέρας της ρίζας, που βρίσκεται στην main, περιμένει το παιδί του. Μόλις είναι έτοιμη η ρίζα της στέλνει σήμα kill(pid, SIGCONT) για να συνεχίσει την λειτουργία της. Κατόπιν, περιμένει πάλι το παιδί του - ρίζα του δέντρου μας. Ταυτόχρονα, η ρίζα, συνεχίζοντας την λειτουργία της, στέλνει σήμα στα kill(pid, SIGCONT) στα παιδιά της και τα περιμένει, αυτά αναδρομικά στα παιδιά τους και τα περιμένουν, και αυτή η διαδικασία συνεχίζεται αναδρομικά μέχρι τα φύλλα. Τα τελευταία δεν έχουν παιδιά, οπότε ολοκληρώνονται οι διεργασίες τους (μία την φορά) και αναδρομικά συνεχίζουν οι γονείς τους που τα περίμεναν. Μόλις τελειώσουν τα υποδέντρα, φτάνουμε στην ρίζα. Τελειώνει η διεργασία της ρίζας και μετά του πατέρα της.

Απάντηση 1ης ερώτησης

Η εμφάνιση του δέντρου διεργασιών γίνεται όταν είναι παγωμένο το δέντρο και αυτό ακριβώς είναι το πλεονέκτημα των σημάτων με την wait_for_ready_children, έναντι της συνάρτησης sleep. Ειδικότερα, εμείς ελέγχουμε την ροή εκτέλεσης των διεργασιών, αφού μπορούμε να τις σταματήσουμε και να τις συνεχίσουμε κατά βούληση -με την βοήθεια της

wait_for_ready_children- και όχι να τις ‘’κοιμίζουμε’’, ελπίζοντας ότι έχουμε ορίσει την κατάλληλη χρονική διάρκεια για κάθε κόμβο. Δηλαδή, πλέον, το πρόγραμμα έχει σωστή ορθότητα αυτούσιο, για οποιοδήποτε ζητούμενο δέντρο.

Απάντηση 2ης ερώτησης

Όπως αναφέρθηκε παραπάνω, ο ρόλος της wait_for_ready_children είναι να βοηθά στην σωστή ροή του προγράμματος, αφού μας επιτρέπει να περιμένουμε τα παιδιά ενός κόμβου να πεθάνουν, πριν πεθάνει ο ίδιος. Καθιστά εφικτό, να παγώνουμε τους γονείς μέχρι να τελειώσουν τα παιδιά τους και τελικά να πετύχουμε την DFS postorder με τον τρόπο που εξηγήθηκε παραπάνω. Χωρίς αυτήν, υπάρχει ο κίνδυνος κάποιος πρόγονος στο δέντρο να πεθάνει πριν από κάποιον απόγονό του. Αυτό θα είχε ως αποτέλεσμα να μην είναι δυνατή η παρατήρηση ολοκληρωμένου του δέντρου διεργασιών.

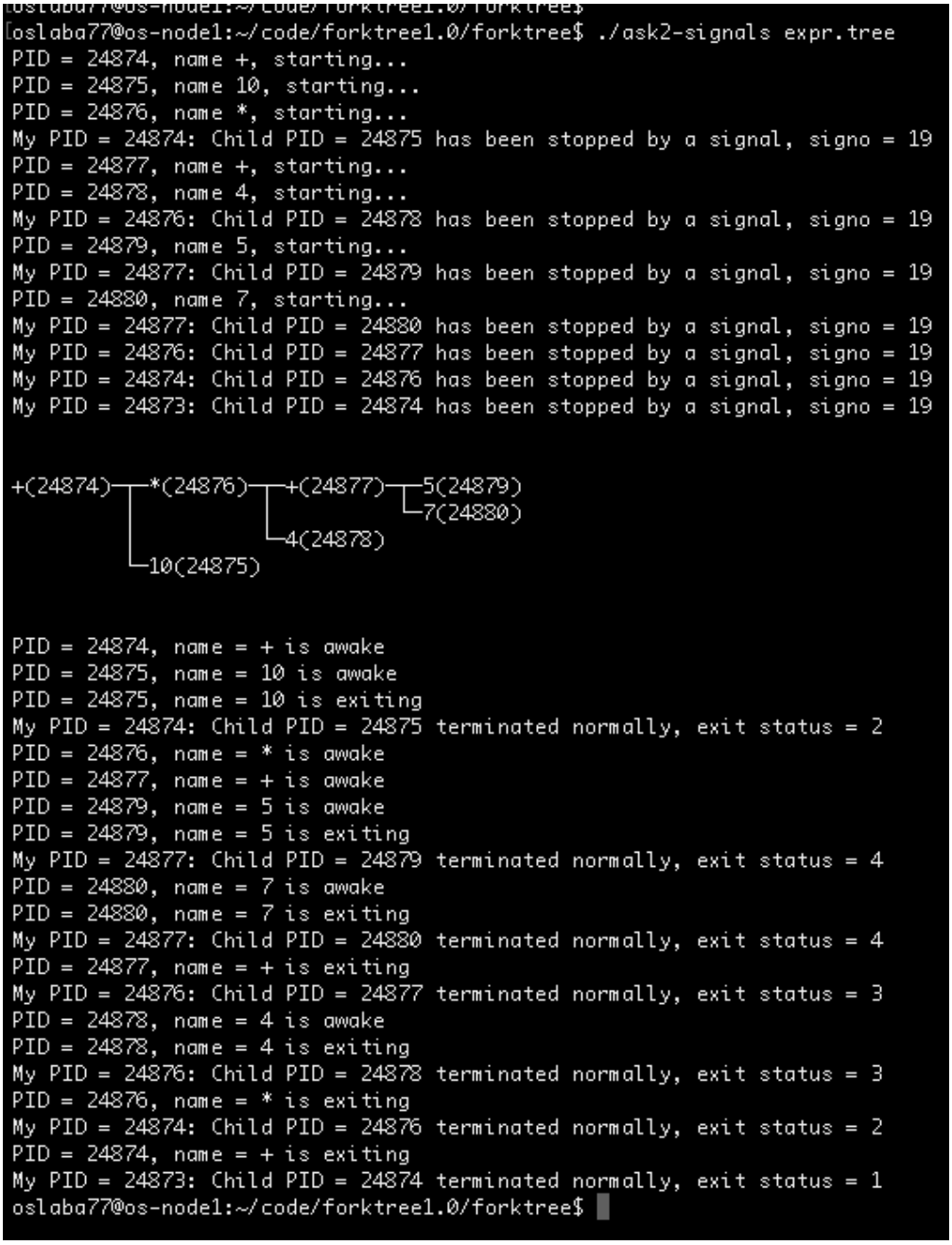
<u>Έξοδος και σύντομες απαντήσεις στις ερωτήσεις</u>

Σημείωση: Τα exit status είναι τα επίπεδα από των παιδιών, των οποίων τερματίζεται η λειτουργία.

Τρέξιμο αρχείου proc.tree

```
oslaba77@os-node1:~/code/forktree1.0/forktree$ ./ask2-signals proc.tree
PID = 24850, name A, starting...
PID = 24851, name B, starting...
PID = 24852, name C, starting...
My PID = 24850: Child PID = 24852 has been stopped by a signal, signo = 19
PID = 24853, name D, starting...
My PID = 24850: Child PID = 24853 has been stopped by a signal, signo = 19
PID = 24854, name E, starting...
My PID = 24851: Child PID = 24854 has been stopped by a signal, signo = 19
PID = 24855, name F, starting...
My PID = 24851: Child PID = 24855 has been stopped by a signal, signo = 19
My PID = 24850: Child PID = 24851 has been stopped by a signal, signo = 19
My PID = 24849: Child PID = 24850 has been stopped by a signal, signo = 19


A(24850)─┬─B(24851)─┬─E(24854)
         │          └─F(24855)
         ├─C(24852)
         └─D(24853)


PID = 24850, name = A is awake
PID = 24851, name = B is awake
PID = 24854, name = E is awake
PID = 24854, name = E is exiting
My PID = 24851: Child PID = 24854 terminated normally, exit status = 3
PID = 24855, name = F is awake
PID = 24855, name = F is exiting
My PID = 24851: Child PID = 24855 terminated normally, exit status = 3
PID = 24851, name = B is exiting
My PID = 24850: Child PID = 24851 terminated normally, exit status = 2
PID = 24852, name = C is awake
PID = 24852, name = C is exiting
My PID = 24850: Child PID = 24852 terminated normally, exit status = 2
PID = 24853, name = D is awake
PID = 24853, name = D is exiting
My PID = 24850: Child PID = 24853 terminated normally, exit status = 2
PID = 24850, name = A is exiting
My PID = 24849: Child PID = 24850 terminated normally, exit status = 1
oslaba77@os-node1:~/code/forktree1.0/forktree$
```

Τρέξιμο αρχείου expr.tree



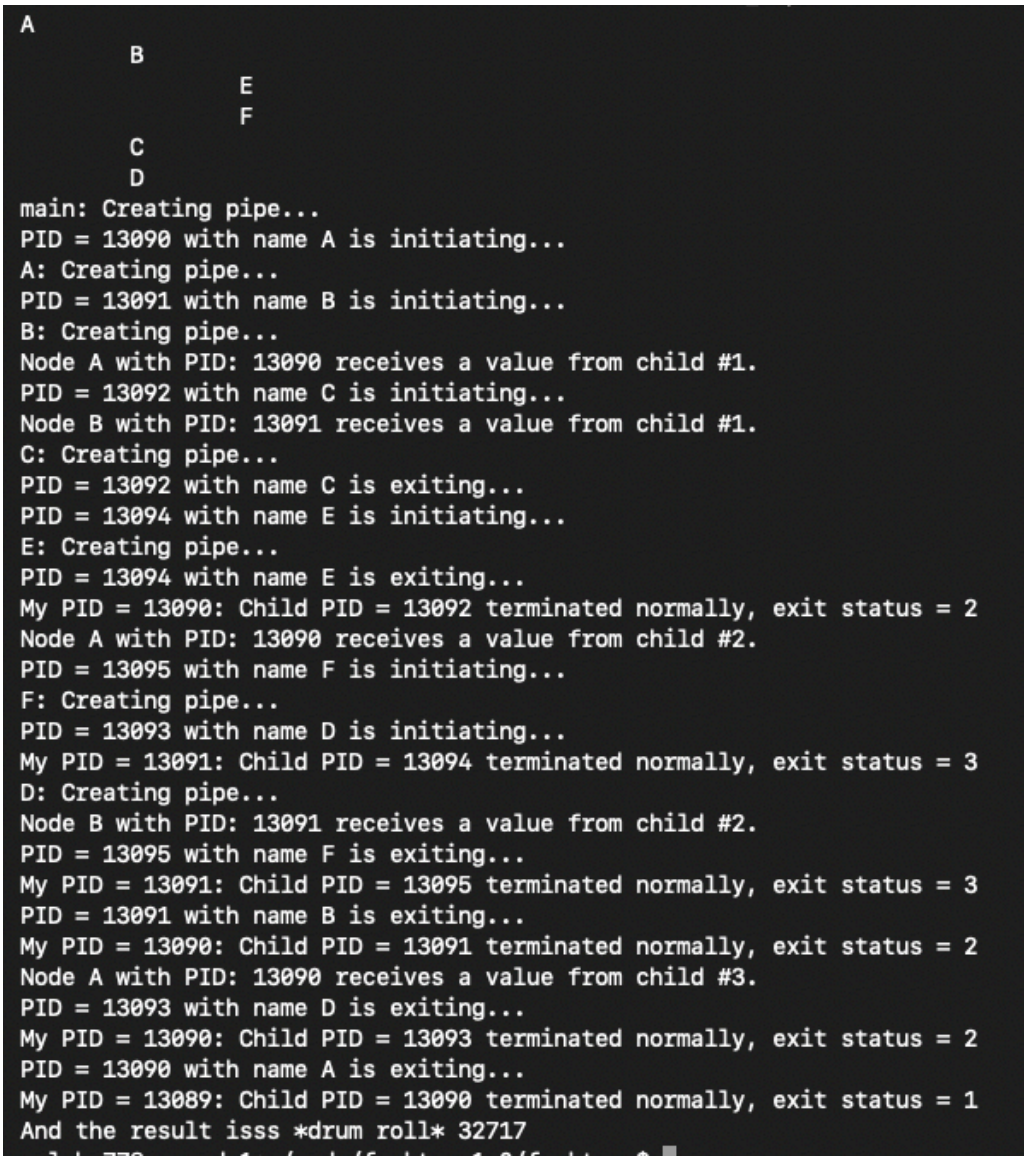
2.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Πηγαίος κώδικας

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

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


static void croccs(int fd, struct tree_node *root, int level)
{
        pid_t pid;
        int i, pfd[2], status, val, timi[2];

        printf("PID = %ld with name %s is initiating...\n", (long)getpid(), root->name);
        change_pname(root->name);
        //2 children -> einai telestis
        //An to ena einai telesths kai to allo arithmos tote synexizeis ston telesti
        //An exeis dyo arithmous einai fylla, tote mesw tou  gonea tous kane thn praksi


        /*-------------------------------------------------pipe creation-----------------------------------------------------*/
        printf("%s: Creating pipe...\n", root->name);

        /*************************************************  APOTYXIA  *************************************************/

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

        /*************************************************  EPITYXIA  *************************************************/

        /*-----------------Edw den mpainoun ta fylla kai mesw ths fork dhmiourgeitai to paidi-------------------------------*/
        for (i=0; i < root->nr_children; i++) {
                /*-----------------------------------------------fork--------------------------------------------------------*/
                pid = fork();

                /*-----------------------------------------------fork-error--------------------------------------------------*/
                if (pid < 0) {
                        perror("fork_procs: fork");
                        exit(1);
                }

                /*-----------------------------------------------fork-child--------------------------------------------------*/
                else if (pid == 0) {
                        /*----recursive call of croccs using pfd[1] because as a child it will have to write into the pipe---*/
                        croccs(pfd[1], root->children + i, level + 1);
                }

        }

        /*-Edw oloi oi komvoi ektos apo ta fylla, diavazoyn tis times twn paidiwn tous kai tis apothikeuoun ston pinaka timi-*/
        for (i=0; i < root->nr_children; i++){
                printf("Node %s with PID: %ld receives a value from child #%d.\n", root->name, (long)getpid(), i+1);

                /*-----------------------father reads value from the same pipe both children use------------------------------*/
                if (read(pfd[0], &val, sizeof(val)) != sizeof(val)) {
                        perror("parent: read from pipe");
                        exit(1);
                }

                timi[i]= val;

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

        }


        /*-----------------------------------------------komvos-fyllo------------------------------------------------------*/
        if (((root->name[0]) != '+') && ((root->name[0]) != '*'))
                sscanf(root->name, "%d", &val);

        /*-------------------------------------komvos-telestis-ektelesi_praksewn---------------------------------------------*/
        else if ((root->name[0])  == '+') {
                val = timi[0] + timi[1];
        }
```

```
        else if ((root->name[0]) == '*') {
                val = timi[0] * timi[1];
        }

        /*-----------------------------------------------------write in pipe-----------------------------------------------------*/
        if (write(fd, &val, sizeof(val)) != sizeof(val)) {
                perror("child: write to pipe");
                exit(1);
        }


        /*-----------------------------------------------------exit-----------------------------------------------------*/
        printf("PID = %ld with name %s is exiting...\n", (long)getpid(), root->name);

        /*-----------------------------------------------------close pipes-----------------------------------------------------*/
        close(fd);
        close(pfd[0]);
        /*-----------------------------------------------------exit level-----------------------------------------------------*/
        exit(level);

}


void crocs(int fd, struct tree_node *root)
{
        croccs(fd, root, 1);
}

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

{
        pid_t pid;
        struct tree_node *root;
        int pfd[2], status, ans;
        /*   pfd[0] -- read   */
        /*   pfd[1] -- write  */

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


        /*-----------------------------------------------------read tree-----------------------------------------------------*/
        root = get_tree_from_file(argv[1]);

        /*-----------------------------------------------------print tree-----------------------------------------------------*/
        print_tree(root);

        /*-----------------------------------------------------pipe creation-----------------------------------------------------*/
        printf("main: Creating pipe...\n");

        /*************************************************  APOTYXIA  *************************************************/

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

        /*************************************************  EPITYXIA  *************************************************/

        /*-----------------------------------------------------fork-----------------------------------------------------*/
        pid = fork();

        /*-----------------------------------------------------fork-error-----------------------------------------------------*/
        if (pid < 0) {
                perror("main: fork");
                exit(1);
        }

        /*-----------------------------------------------------fork-child-----------------------------------------------------*/
        if (pid == 0) {
                crocs(pfd[1],root);
                /*stelnei thn akrh grapsimatos pfd[1] tou pipe sthn synartisi wste h riza tou tree pou periexei telesti na kanei
                ton teleytaio ypologismo kai na epistrepsei to teliko apotelesma sto father kai autos na to diavasei mesa apo to
                to  allo akro tou pipe*/
                exit(1);
        }

        /*-----------------------------------------------------fork-father-----------------------------------------------------*/

        /*------------------------------father reads final result from the tree's root-----------------------------------------*/
        if (read(pfd[0], &ans, sizeof(ans)) != sizeof(ans)) {
                perror("parent: read from pipe");
                exit(1);
        }

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

        /*-----------------------------------------------Print the result-----------------------------------------------------*/
        printf("And the result isss *drum roll* %d\n", ans);
        return 0;

}
```

Έξοδος και σύντομες απαντήσεις στις ερωτήσεις



```
A
        B
                E
                F
        C
        D
main: Creating pipe...
PID = 13090 with name A is initiating...
A: Creating pipe...
PID = 13091 with name B is initiating...
B: Creating pipe...
Node A with PID: 13090 receives a value from child #1.
PID = 13092 with name C is initiating...
Node B with PID: 13091 receives a value from child #1.
C: Creating pipe...
PID = 13092 with name C is exiting...
PID = 13094 with name E is initiating...
E: Creating pipe...
PID = 13094 with name E is exiting...
My PID = 13090: Child PID = 13092 terminated normally, exit status = 2
Node A with PID: 13090 receives a value from child #2.
PID = 13095 with name F is initiating...
F: Creating pipe...
PID = 13093 with name D is initiating...
My PID = 13091: Child PID = 13094 terminated normally, exit status = 3
D: Creating pipe...
Node B with PID: 13091 receives a value from child #2.
PID = 13095 with name F is exiting...
My PID = 13091: Child PID = 13095 terminated normally, exit status = 3
PID = 13091 with name B is exiting...
My PID = 13090: Child PID = 13091 terminated normally, exit status = 2
Node A with PID: 13090 receives a value from child #3.
PID = 13093 with name D is exiting...
My PID = 13090: Child PID = 13093 terminated normally, exit status = 2
PID = 13090 with name A is exiting...
My PID = 13089: Child PID = 13090 terminated normally, exit status = 1
And the result isss *drum roll* 32717
```

Σχόλια:

Ακολουθούμε το ίδιο μοτίβο δημιουργίας δέντρου διεργασιών μέσω αναδρομής, όπως και στις προηγούμενες ασκήσεις. Η λύση μας βασίζεται στη δημιουργία μίας σωλήνωσης μεταξύ γονέα-παιδιών, στην οποία γράφουν τα δύο παιδιά και διαβάζει ο γονέας. Ο γονέας λαμβάνει την πληροφορία με τυχαία σειρά, όμως στην περίπτωση της πρόσθεσης και του πολλαπλασιασμού, αυτό δεν είναι πρόβλημα αφού πρόκειται για αριθμητικές πράξεις, για τις οποίες ισχύει η αντιμεταθετική ιδιότητα.

Απάντηση 1ης ερώτησης

Στην υλοποίηση της άσκησης χρησιμοποιήσαμε μία σωλήνωση για κάθε γονέα με τα παιδιά του, δηλαδή μια σωλήνωση για την εκτέλεση μιας πράξης. Συγκεκριμένα, τα παιδιά μέσω μίας σωλήνωσης στέλνουν την πληροφορία στον γονέα. Το ότι οι πράξεις που κληθήκαμε να χειριστούμε είναι πρόσθεση και πολλαπλασιασμός μας επιτρέπει να έχουμε μία σωλήνωση ανά πράξη, καθώς σε αυτήν την υλοποίηση η πληροφορία από τα παιδιά - φύλλα έρχεται με τυχαία σειρά, αλλά ισχύει η αντιμεταθετική ιδιότητα. Σε περίπτωση που είχαμε αφαίρεση, διαίρεση ή mod θα έπρεπε να παραλλάξουμε την υλοποίησή μας, δημιουργώντας μια σωλήνωση ανά παιδί.

Απάντηση 2ης ερώτησης

Με την προϋπόθεση ότι το δέντρο είναι πυκνό έως έναν βαθμό, η αποτίμηση της έκφρασης από δέντρο διεργασιών σε σύστημα πολλαπλών επεξεργαστών μας δίνει τη δυνατότητα πολλές διεργασίες να εκτελούνται παράλληλα. Έτσι, εξοικονομούμε πολύτιμο χρόνο, αφού με τη δημιουργία κατάλληλων σωληνώσεων αποκτούμε την πληροφορία που επιθυμούμε σε πολύ πιο σύντομο χρόνο.

Προαιρετική ερώτηση 1

Η ταχύτητα παράλληλου υπολογισμού εξαρτάται από τα εξής:

<u>Ποιοτικά χαρακτηριστικά</u>

1) Τα χαρακτηριστικά των επεξεργαστών που δουλεύουν παράλληλα. Πυρήνες που διαθέτουν αυτοί και ο επεξεργαστής που είναι μόνος του.
2) Ανάλογα με τις δυνατότητες του κάθε επεξεργαστή και τον αριθμό των επεξεργαστών που διαθέτουμε, πρέπει να αναθέτουμε κατάλληλα τους υπολογισμούς

Ποσοτικά χαρακτηριστικά

1) Τη μορφή του δέντρου. Θέλουμε πυκνά δέντρα και παράλληλους επεξεργαστες να υπολογίζουν τα υποδέντρα. Στην χειρότερη περίπτωση έχουμε λίστα, οπότε ο επεξεργαστής κάθε επιπέδου περιμένει αυτόν κατώτερου επιπέδου, οπότε έχουμε πάλι σειριακή εκτέλεση, όπως θα είχαμε και για N = 1 επεξεργαστή.
2) Το πως θα κάνουμε καταμερισμό εργασίας για διάφορες πιθανές περιπτώσεις δέντρων.

Οι πρώτες περιπτώσεις (1, 1) των δύο ειδών χαρακτηριστικών είναι και αυτές που μπορούν να καταστήσουν παράλληλους επεξεργαστές πιο αργούς από έναν που λειτουργεί σειριακά. Οι άλλες δύο (2, 2) περιπτώσεις, και βέλτιστα να μην υλοποιηθούν, και πάλι καθιστούν γρηγορότερους τους N επεξεργαστές, αν δηλαδή τα χαρακτηριστικά των N και του μονού επεξεργαστή είναι παρόμοια και το δέντρο σχετικά πυκνό, κάποιος θα επέλεγε τους N επεξεργαστές.

Προαιρετική ερώτηση 2

Η προτεινόμενη υβριδική υλοποίηση συνδυάζει τη σειριακή με την παράλληλη υλοποίηση που είδαμε πριν.

Σε περίπτωση μόνο σειριακής υλοποίησης χάνουμε πολύτιμο χρόνο από διεργασίες που θα μπορούσαν να είχαν ήδη γίνει.

Σε περίπτωση μόνο παράλληλης υλοποίησης και για μεγάλα δέντρα αριθμητικών πράξεων υπάρχει ο κίνδυνος να χρησιμοποιήσουμε μεγάλο αριθμό επεξεργαστών και υπολογιστικής ισχύος, ο οποίος ενδέχεται να είναι απαραίτητος για τις υπόλοιπες λειτουργίες του συστήματος μας.

Συνεπώς, η υβριδική υλοποίηση του παράλληλου υπολογισμού αριθμητικής έκφρασης δίνει την ευχέρεια στον δημιουργό της να επιλέξει το βάθος μέχρι το οποίο θα γίνονται παράλληλες διεργασίες, και άρα θα αξιοποιούνται περισσότεροι επεξεργαστές. Πρακτικά η παράμετρος «μ» εξασφαλίζει ότι από το σημείο που ορίζει και έπειτα οι διεργασίες θα γίνονται σειριακά, και εφόσον το «μ» συνδέεται με σχέση αναλογίας με τον αριθμό των επεξεργαστών που έχουν

χρησιμοποιηθεί μέχρι στιγμής καταλαβαίνουμε ότι όσο το «μ» αυξάνεται τόσο περισσότεροι επεξεργαστές μπορούν να χρησιμοποιηθούν και άρα τόσο πιο γρήγορη είναι η υλοποίηση.

Έτσι η βέλτιστη τιμή της μεταβλητής «μ» εξαρτάται τόσο από τη διαθέσιμη για τον αριθμητικό υπολογισμό υπολογιστική ισχύ, όσο και από το χρόνο που έχουμε στη διάθεση μας μέχρι να λάβουμε το ζητούμενο αποτέλεσμα.

ΤΕΛΟΣ