Λειτουργικά Συστήματα

2η αναφορά

Ονοματεπώνυμο	Εμμανουηλίδης Εμμανουήλ	Λίτσος Ιωάννης
Αριθμός Μητρώου	03119435	03119135
Ομάδα	oslab61	

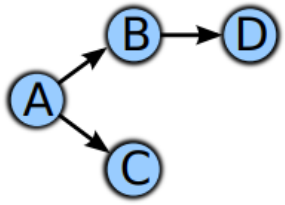
Άσκηση 2:

ΔΙΑΧΕΙΡΙΣΗ ΔΙΕΡΓΑΣΙΩΝ ΚΑΙ ΔΙΑΔΙΕΡΓΑΣΙΑΚΗ ΕΠΙΚΟΙΝΩΝΙΑ

ΑΣΚΗΣΕΙΣ

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Σε αυτή την άσκηση ζητείται η κατασκευή προγράμματος που δημιουργεί το παρακάτω δέντρο διεργασιών :



Για να ξεχωρίζουμε τις διεργασίες κάθε μία τερματίζει με διαφορετικό κωδικό επιστροφής : A = 16, B = 19, C = 17, D = 13. Ο κώδικας της άσκησης είναι ο ακόλουθος:

```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. */

        pid_t B, C, D;
        int status;

        change_pname("A");
        printf("A -> Created\n");
```

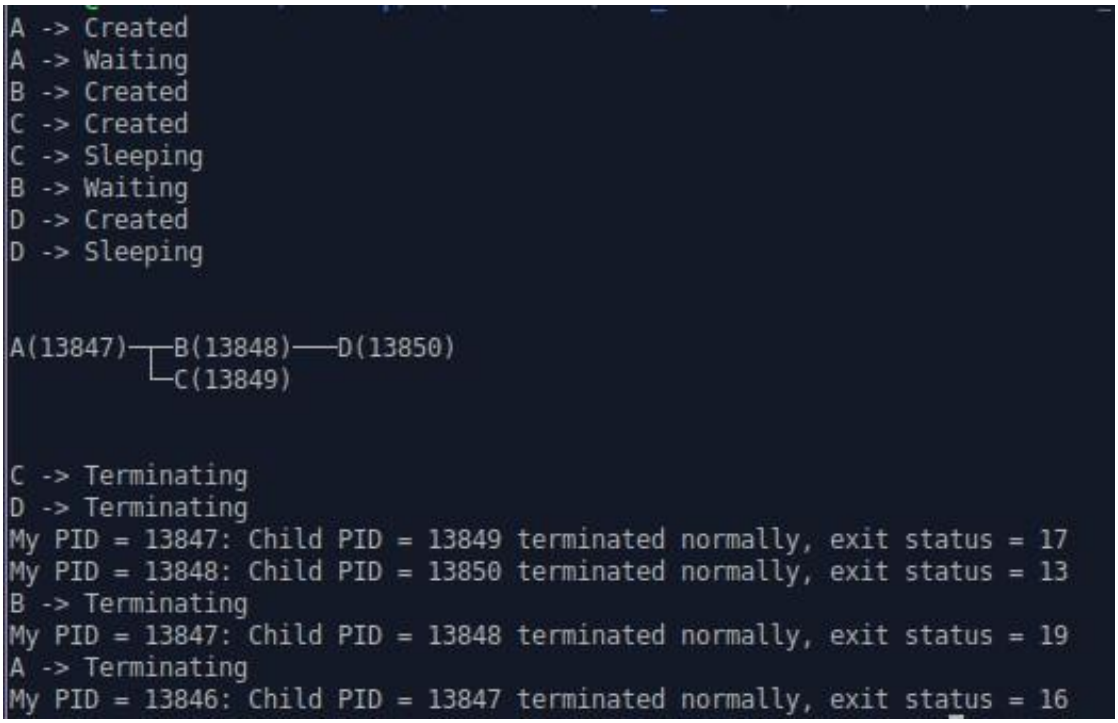
```c
        /* we create B */
        B = fork();
        if(B < 0) {
            /* B fork failed */
            perror("B fork failed");
            exit(1);
        }
        else if(B == 0) {
            /* In B process */
            change_pname("B");
            printf("B -> Created\n");

            /* we create D */
            D = fork();
            if(D < 0) {
                /* D fork failed */
                perror("D fork failed");
                exit(1);
            }
             else if(D == 0) {
                /* In D process */
                change_pname("D");
                printf("D -> Created\n");
                printf("D -> Sleeping\n");
                sleep(SLEEP_PROC_SEC);
                printf("D -> Terminating\n");
                exit(13);
            }
            else {
                /* In B process */
                printf("B -> Waiting\n");
                D = wait(&status);
                explain_wait_status(D, status);
                printf("B -> Terminating\n");
                exit(19);
            }
        }
        else {
            /* In A process */

            /* we create C */
            C = fork();
            if(C < 0) {
                /* C fork failed */
                perror("C fork failed");
```

```
                exit(1);
            }
            else if(C == 0) {
                /* In C process */
                change_pname("C");
                printf("C -> Created\n");
                printf("C -> Sleeping\n");
                sleep(SLEEP_PROC_SEC);
                printf("C -> Terminating\n");
                exit(17);
            }
            else {
                /* In A process */

                printf("A -> Waiting\n");
                B = wait(&status);
                explain_wait_status(B, status);
                C = wait(&status);
                explain_wait_status(C, status);
                 printf("A -> Terminating\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.
 */

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

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

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

        return 0;
}
```

Στη main καλούμε τη κλήση συστήματος fork() για τη δημιουργία της διεργασίας-ρίζας του δέντρου διεργασιών και το παιδί της main δηλαδή η ρίζα καλεί την συνάρτηση fork_procs() που δημιουργεί το ζητούμενο δέντρο διεργασιών . Έπειτα, στη main καλούμε τη sleep και στη συνέχεια τις wait και explain_wait_status ώστε να περιμένει να τελειώσει η διεργασία ρίζα και τέλος να τερματίσει.

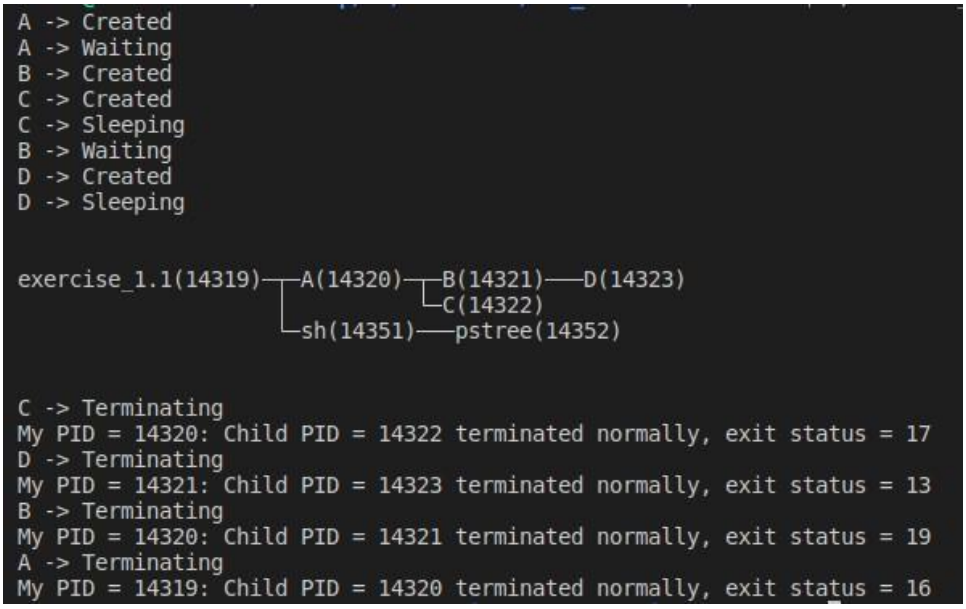
Η συνάρτηση fork_procs αρχικά δημιουργεί την διεργασία B , στη συνέχεια η διεργασία B δημιουργεί την D και παράλληλα η A δημιουργεί την C. Κάθε φορά φροντίζουμε ώστε να τυπώνονται κατάλληλα μηνύματα (‘’created” , “sleeping”, “terminated”). Επίσης, κάθε διεργασία φύλλο καλεί την sleep() προτού κάνει exit() και τερματίσει, ενώ οι διεργασίες μη φύλλα καλούν τις wait και explain_wait_status και αφού βεβαιωθούν ότι έχουν τερματίσει όλα τα παιδιά τους τερματίζουν και αυτές. Εκτελώντας τον παραπάνω κώδικα παίρνουμε την παρακάτω έξοδο:



Όπως βλέπουμε πρώτα τερματίζουν οι διεργασίες D και C , έπειτα η B και τέλος η A.

ΕΡΩΤΗΣΕΙΣ

1) Τερματίζοντας πρόωρα την διεργασία A , οι διεργασίες παιδιά της που δεν έχουν τερματίσει θα μείνουν ορφανές . Έτσι, θα γίνουν παιδιά της init (PID =1) που καλεί συνεχώς την wait().
2) Αν κάνουμε show_pstree(getpid()) παίρνουμε την παρακάτω έξοδο:



Η getpid() επιστρέφει το process id της τρέχουσας διεργασίας. Έτσι, αν καλέσουμε show_pstree(getpid()) θα τυπωθεί ολόκληρο το δέντρο διεργασιών που ξεκινάει με την exercise_1.1 ως ρίζα . Στο δέντρο περιλαμβάνονται οι διεργασίες sh (standard command language interpreter), με παιδί το pstree, που καλούνται από την show_pstree.

3) Σε υπολογιστικά συστήματα πολλαπλών χρηστών ο αριθμός των διαθέσιμων πόρων είναι πεπερασμένος . Έτσι, αν δεν τεθούν όρια στον αριθμό διεργασιών που μπορεί να δημιουργήσει ένας χρήστης θα υπάρχει κίνδυνος υπερφόρτωσης του συστήματος , αν για παράδειγμα μία κακόβουλη διεργασία έκανε συνεχώς fork() δημιουργώντας διεργασίες αέναα.

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Στην άσκηση αυτή ζητείται η κατασκευή προγράμματος που δημιουργεί αυθαίρετα δέντρα διεργασιών βάσει αρχείου εισόδου. Ο κώδικας της συγκεκριμένης άσκησης είναι ο εξής:

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

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

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

void fork_tree(struct tree_node *root) {

    /* exit(17) -> leaves
       exit(42) -> parents */

    change_pname(root->name);
    int i, status;

    /* case 1 -> Current node is a leaf */
    if(root->nr_children == 0) {
```

```
        printf("Leaf %s -> Created\n", root->name);
        printf("Leaf %s -> Sleeping\n", root->name);
        sleep(SLEEP_PROC_SEC);
        printf("Leaf %s -> Terminating\n", root->name);
        exit(17);
    }
    else {
        /* case 2 -> Current node has child(ren) */

        printf("Node %s -> Created\n", root->name);
        pid_t child;

        for(i = 0; i < root->nr_children; i++) {
            child = fork();
            if(child < 0) {
                /* child fork failed */
                perror("child fork failed");
                exit(1);
            }
            else if(child == 0) {
                /* In child process */
                fork_tree(root->children + i);
            }
            /* else continue to next iteration or exit loop */
        }

        /* In father process */

        printf("Node %s -> Waiting\n", root->name);
        for(i = 0; i < root->nr_children; i++) {
            child = wait(&status);
            explain_wait_status(child, status);
        }
        printf("Node %s -> Terminating\n", root->name);
        exit(42);
    }
}


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

        /* Checking for valid input */
        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;

        pid = fork();
        if(pid < 0) {
            perror("main: fork");
            exit(1);
        }
        if(pid == 0) {
            /* Child */
            fork_tree(root);
            exit(1);
        }

        /* Father */

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

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

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

        return 0;
}
```

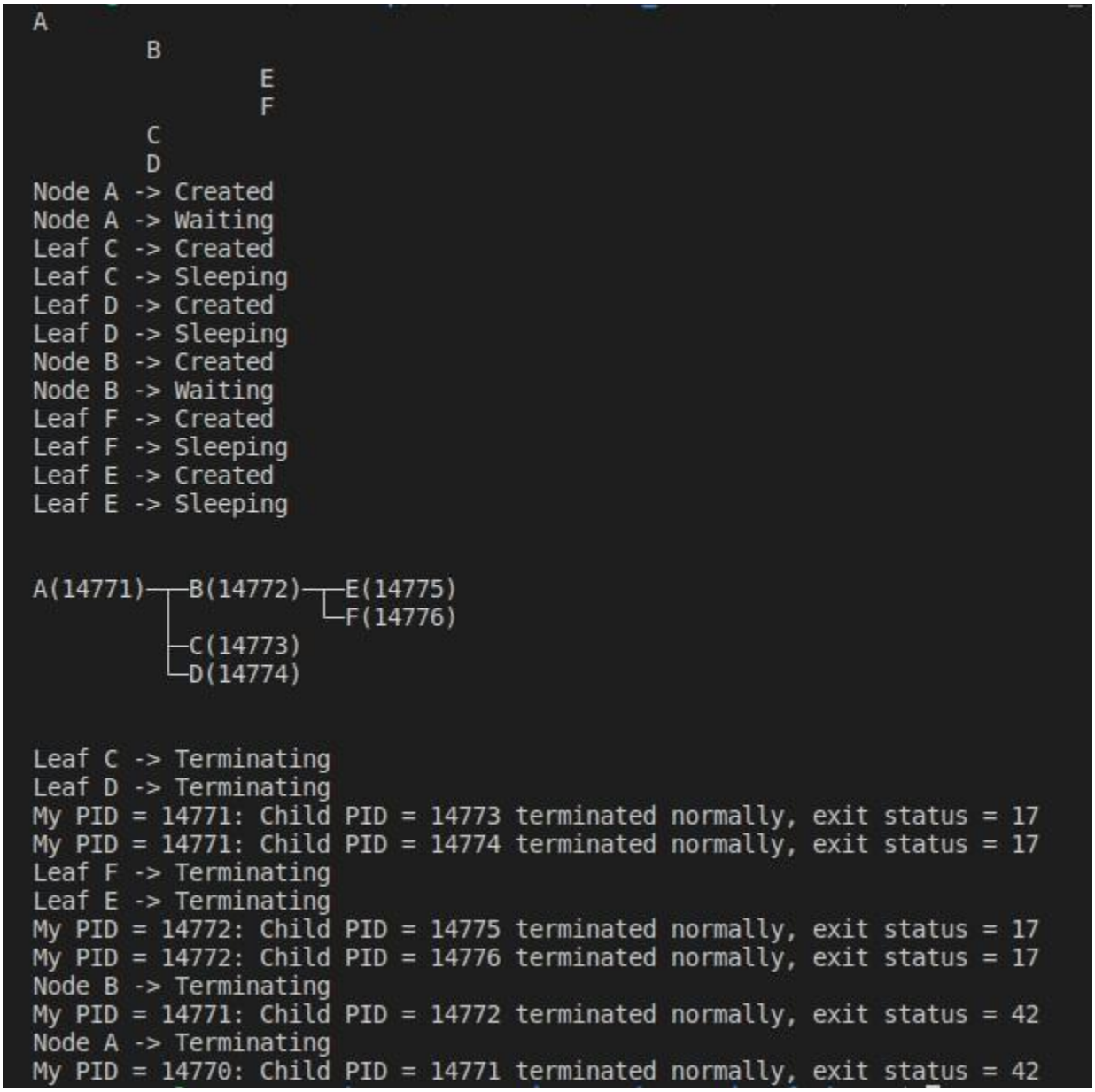
Το πρόγραμμα ακολουθεί παρόμοια λογική με την προηγούμενη άσκηση , αλλά βασίζεται στην αναδρομική συνάρτηση fork_tree() η οποία καλείται για κάθε κόμβο του δέντρου και κάνει τα εξής:

1. Ελέγχει αν η τρέχουσα διεργασία είναι φύλλο και αν είναι τυπώνει τα κατάλληλα μηνύματα , κάνει sleep() και exit(17).
2. Αν δεν είναι φύλλο τότε κάνει fork για κάθε παιδί της.

3. Στη συνέχεια, καλείται αναδρομικά η συνάρτηση fork_tree για το κάθε παιδί της διεργασίας πατέρα που έκανε fork, ενώ παράλληλα η διεργασία που έκανε fork κάνει wait και explain_wait_status για κάθε παιδί της και τέλος κάνει exit(42).

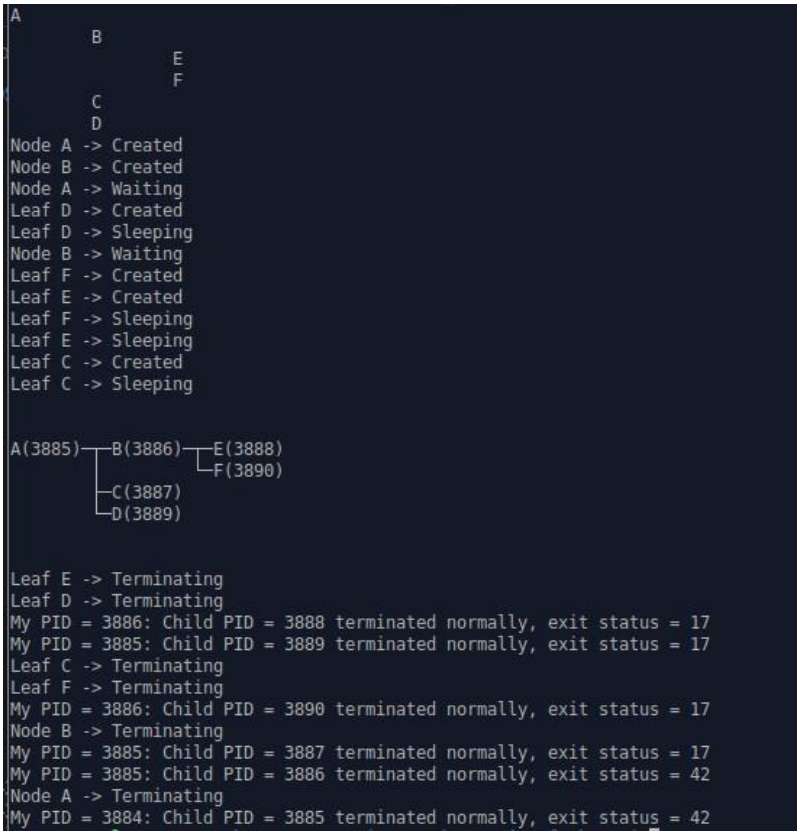
Στη main ελέγχουμε για σωστό αριθμό ορισμάτων και καλούμε την συνάρτηση get_tree_from_file ώστε να πάρουμε το ζητούμενο δέντρο διεργασιών από το αρχείο εισόδου.

Εκτελούμε το πρόγραμμα με αρχείο εισόδου το proc.tree και έχουμε το παρακάτω output:



Ερώτηση

1) Στο παραπάνω παράδειγμα τα μηνύματα έναρξης εμφανίζονται με BFS(Breadth First Search) σειρά. Ωστόσο, η σειρά έναρξης και τερματισμού των διεργασιών δεν είναι καθορισμένη και εξαρτάται από τον scheduler. Για παράδειγμα, όταν ξανατρέξαμε το ίδιο πρόγραμμα σε άλλο σύστημα πήραμε την ακόλουθη έξοδο:



Αυτό συμβαίνει διότι με τη χρήση των καθυστερήσεων sleep() δεν μπορούμε να είμαστε βέβαιοι για το ποια θα είναι η σειρά έναρξης και τερματισμού των διεργασιών. Για παράδειγμα, η διεργασία Α μπορεί να δημιουργήσει πρώτα την Β και άλλες φορές πρώτα την C Αυτό που γνωρίζουμε είναι ότι:

- Ο κάθε πατέρας δημιουργεί τα παιδιά του και περιμένει αυτά να τερματίσουν
- Εφόσον η συνάρτηση είναι αναδρομική τα παιδιά κάνουν το ίδιο
- Όταν βρεθεί μια διεργασία φύλλο κάνει sleep μέχρι να κατασκευαστεί ολόκληρο το δέντρο
- Έπειτα τα φύλλα τερματίζουν
- Στη συνέχεια τερματίζουν οι γονείς των φύλλων και διαδικασία συνεχίζεται μέχρι να τερματίσει και η διαδικασία ρίζα

1.3 Αποστολή και χειρισμός σημάτων

Στη συγκεκριμένη άσκηση επεκτείνουμε το πρόγραμμα της 1.2 έτσι ώστε οι διεργασίες να ελέγχονται με χρήση σημάτων, για να εκτυπώνουν τα μηνύματα τους κατά βάθος(Depth-First). Ο κώδικας της άσκησης είναι ο ακόλουθος:

```
#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_DFS(struct tree_node *root)
{
        /*
         * Start
         */

        int i, status;

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

        /* case 1 -> Current node is a leaf */
        if(root->nr_children == 0) {

            /* Suspend Self */
            raise(SIGSTOP); /* stop */

            /* after awakening */
            printf("PID = %ld, name = %s is awake\n",
                    (long)getpid(), root->name);
            exit(0);
        }
        else {
            /* case 2 -> Current node has child(ren) */
```

```
            /* kids -> array with PIDs of the children of current node */
            pid_t kids[root->nr_children];

            for(i = 0; i < root-> nr_children; i++) {
                kids[i] = fork();
                if(kids[i] < 0) {
                    /* child fork failed */
                    perror("child fork failed");
                    exit(1);
                }
                else if(kids[i] == 0) {
                    /* In child process */
                    fork_DFS(root->children + i);
                }
            }
            /* In father process */

            /* wait for ALL children to stop */
            wait_for_ready_children(root->nr_children);

            raise(SIGSTOP); /* stop */

            /* after awakening */
            printf("PID = %ld, name = %s is awake\n",
                (long)getpid(), root->name);

            for(i = 0; i < root->nr_children; i++) {
                /* wake up children */
                kill(kids[i], SIGCONT);
                printf("PID = %ld, name = %s is waiting\n",
                    (long)getpid(), root->name);
                kids[i] = wait(&status);
                explain_wait_status(kids[i], status);
            }
            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().
 *
 * 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]);
        print_tree(root);

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

        /*
         * Father
         */

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

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

        /* for ask2-signals */
        kill(pid, SIGCONT);   /* send SIGCONT signal to root process */

        /* Wait for the root of the process tree to terminate */
```

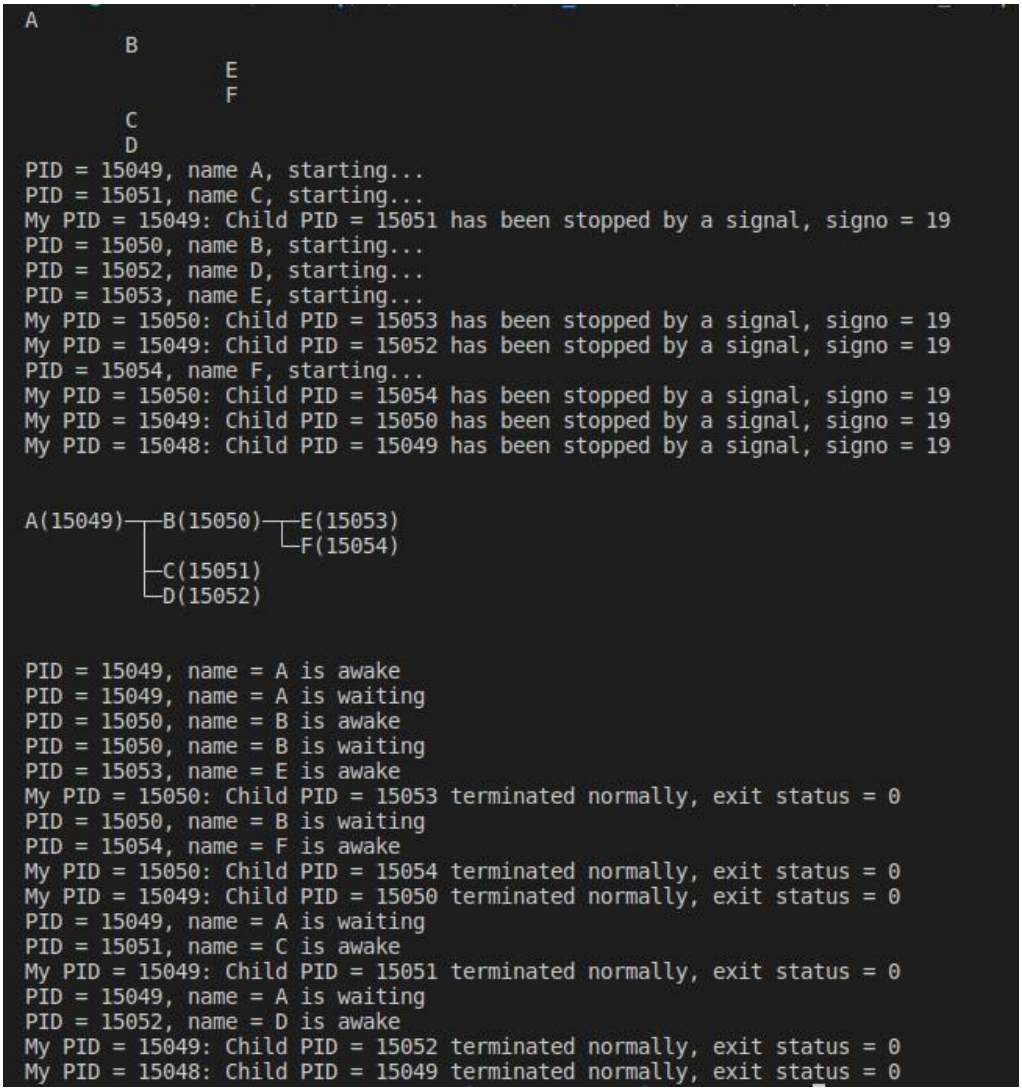
```
        pid = wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

Αρχικά, η συνάρτηση main κάνει τις ίδιες λειτουργίες με πριν αλλά αντί για τη sleep καλεί την wait_for_ready_children(1) και την kill(pid, SIGCONT) . Η wait_for_ready_children() μας βεβαιώνει ότι όλα τα παιδιά έχουν αναστείλει την λειτουργία τους , ενώ η kill(pid, SIGCONT) ,όπου pid το pid της διεργασίας ρίζας , στέλνει κατάλληλο σήμα εκκίνησης στη διεργασία ρίζα.

Στην αναδρομική συνάρτηση fork_DFS :

- Αρχικά τυπώνουμε το μήνυμα εκκίνησης της διεργασίας
- Ελέγχουμε αν η διεργασία είναι φύλλο και αν είναι καλείται η raise(SIGSTOP) ώστε να ανασταλεί η λειτουργία της και περιμένει ένα σήμα εκκίνησης SIGCONT από τον πατέρα της . Όταν το λάβει ξυπνάει , τυπώνει αντίστοιχο μήνυμα και κάνει exit(0)
- Αν δεν είναι φύλλο τότε κάνει fork() για κάθε παιδί της
- Καλείται η συνάρτηση fork_DFS αναδρομικά για κάθε παιδί ,ενώ η ίδια διεργασία καλεί την wait_for_ready_children περιμένοντας όλα τα παιδιά να αναστείλουν τη λειτουργία τους. Όταν συμβεί αυτό καλείται η raise(SIGSTOP) ώστε να ανασταλεί η λειτουργία της και περιμένει σήμα εκκίνησης SIGCONT από το πατέρα. Όταν το λάβει ξυπνάει και στέλνει σε κάθε παιδί της σήμα εκκίνησης SIGCONT. Για το σκοπό αυτό αποθηκεύουμε τα PIDs όλων των παιδιών της κάθε διεργασίας μέσω του πίνακα kids. Αφού στείλει τα σήματα εκκίνησης για κάθε παιδί κάνει wait και explain_wait_status και τέλος κάνει exit(0).

Εκτελούμε το παραπάνω πρόγραμμα με αρχείο εισόδου το proc.tree και προκύπτει το παρακάτω output:



Όπως βλέπουμε από την παραπάνω έξοδο τα μηνύματα αφύπνισης awake εμφανίζονται με τη σειρά A,B,E,F,C,D , δηλαδή κατά βάθος (DFS).

Ερωτήσεις

1) Αρχικά, ο κώδικας εκτελείται ταχύτερα διότι οι διεργασίες φύλλα δεν χρειάζεται να καθυστερούν καλώντας την sleep() . Επίσης, η σειρά διάσχισης του δέντρου είναι πάντα ίδια λόγω του συγχρονισμού που πετυχαίνουμε με τα σήματα. Αυτό σημαίνει ότι για παράδειγμα στο παραπάνω δέντρο η σειρά διάσχισης θα είναι πάντα η A,B,E,F,C,D , ενώ αν χρησιμοποιούσαμε την sleep αυτό δε θα ίσχυε. Τέλος, είμαστε βέβαιοι ότι όταν θα κληθεί η show_pstree θα τυπώσει το σωστό δέντρο χωρίς κάποιο σφάλμα. Αντίθετα, στη περίπτωση της sleep , αν είχαμε ορίσει διαφορετικό χρόνο καθυστέρησης, θα μπορούσαν ορισμένες διεργασίες να είχαν τερματίσει πριν την κλήση της show_pstree και έτσι θα τυπωνόταν λάθος δέντρο.
2) Η wait_for_ready_children μάς διαβεβαιώνει ότι όλα τα παιδιά έχουν αναστείλει τη λειτουργία τους. Αφού βεβαιωθούμε γι'αυτό τότε καλείται η raise(SIGSTOP) . Όταν ξυπνήσει η διεργασία πατέρας, ξυπνάει όλα της τα παιδιά. Έτσι, εξασφαλίζουμε ότι η σειρά αφύπνισης είναι DFS. Αν την παραλείπαμε τότε η διεργασία πατέρας θα ξυπνούσε όλα τα παιδιά που θα είχε εκείνη τη στιγμή. Αν κάποιο παιδί δεν είχε προλάβει να αναστείλει την λειτουργία του και το έκανε μετά δε θα μπορούσε να λάβει ξανά σήμα εκκίνησης και θα κοιμόταν για πάντα.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Σε αυτήν την άσκηση ζητείται η επέκταση του προγράμματος της 1.2 ώστε να υπολογίζονται δέντρα που αναπαριστούν αριθμητικές εκφράσεις. Ο κώδικας είναι ο ακόλουθος :

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

#include <sys/prctl.h>
#include <sys/wait.h>

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


int compute_value(int x, int y, char operator) {
    if(operator == '*')
        return x * y;
    else
        return x + y;
}
```

```c
void fork_pipe(struct tree_node *root, int pfd) {

    if(root->nr_children > 2) {
        perror("More than two children -> Wrong input!");
        exit(1);
    }

    int i, status;

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

    /* case 1 -> current node is a leaf (integer) */
    if(root->nr_children == 0) {

        printf("Value = %s\n", root->name);
        int value = atoi(root->name);  /* convert to integer */

        /* leaf performs write */
        if ( write(pfd, &value, sizeof(value)) != sizeof(value) ) {
            printf("%s -> Write failed\n", root->name);
            exit(1);
        }
        printf("Leaf with PID: %ld ,Name: %s wrote %i in pipe\n",
(long)getpid(),root->name, value);
        printf("Leaf with PID: %ld ,Name: %s -> Terminating\n", (long)getpid(), root-
>name);
        exit(0);
    }
    else {
        /* case 2 -> current node has child(ren) */

        /* kids -> array with PIDs of the children of current node */
        pid_t kids[root->nr_children];

        /* Parent Creating pipe */
        int kid_pfd[2];

        printf("Parent %ld ,Name: %s: Creating pipe...\n", (long)getpid(), root-
>name);
        if (pipe(kid_pfd) < 0) {
            perror("pipe");
            exit(1);
        }
        for(i = 0; i < root->nr_children; i++) {
            kids[i] = fork();
            if(kids[i] < 0) {
                /* child fork failed */
```

```
                perror("child fork failed");
                exit(1);
            }
            else if (kids[i] == 0) {
                /* In child process */
                close(kid_pfd[0]);  /*  child closes reading edge */
                fork_pipe(root->children + i, kid_pfd[1]); /* child writes */
            }
        }
        /* In parent process */

        close(kid_pfd[1]);  /* parent closes writing edge */
        /* children's values */
        int values[2];

        for(i = 0; i < root->nr_children; i++) {

            if( read(kid_pfd[0], &values[i], sizeof(values[i])) != sizeof(values[i])
){
                printf("%s -> Read failed\n", root->name);
                exit(1);
            }
            printf("PID = %ld, name = %s is waiting\n", (long)getpid(), root->name);
            kids[i] = wait(&status);
            explain_wait_status(kids[i], status);
        }
        int new_value = compute_value(values[0], values[1], *(root->name));
        printf("PID = %ld, new value = %i\n", (long)getpid(), new_value);

        /* perform write */
        if ( write(pfd, &new_value, sizeof(new_value)) != sizeof(new_value) ) {
            printf("%s -> Write failed\n", root->name);
            exit(1);
        }
        printf("Node with PID: %ld , Name: %s ,wrote %i in pipe\n", (long)getpid(),
root->name, new_value);
        printf("Node with PID: %ld, Name %s -> Terminating\n", (long)getpid(), root-
>name);
        exit(0);
    }
}



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

```c
    /* Checking for valid input */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
        exit(1);
    }

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

    /* Parent Creating pipe */

    printf("Parent %ld: Creating pipe...\n", (long)getpid());
    if (pipe(pfd) < 0) {
        perror("pipe");
        exit(1);
    }

    printf("Parent %ld: Creating child...\n", (long)getpid());
    pid = fork();
    if (pid < 0) {
        perror("main:fork");
        exit(1);
    }
    if (pid == 0) {
        /* In child process */
        fork_pipe(root, pfd[1]);  /* child writes */
        /*
         * Should never reach this point,
         * fork_pipe() does not return
         */
        printf("Child should never reach this point\n");
        assert(0);
    }

    /*
     * In parent process.
     */

    int final_value;
    if( read(pfd[0], &final_value, sizeof(final_value)) != sizeof(final_value) ){
        printf("Read failed\n");
        exit(1);
    }


    close(pfd[1]);  /* parent closes writing edge */


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

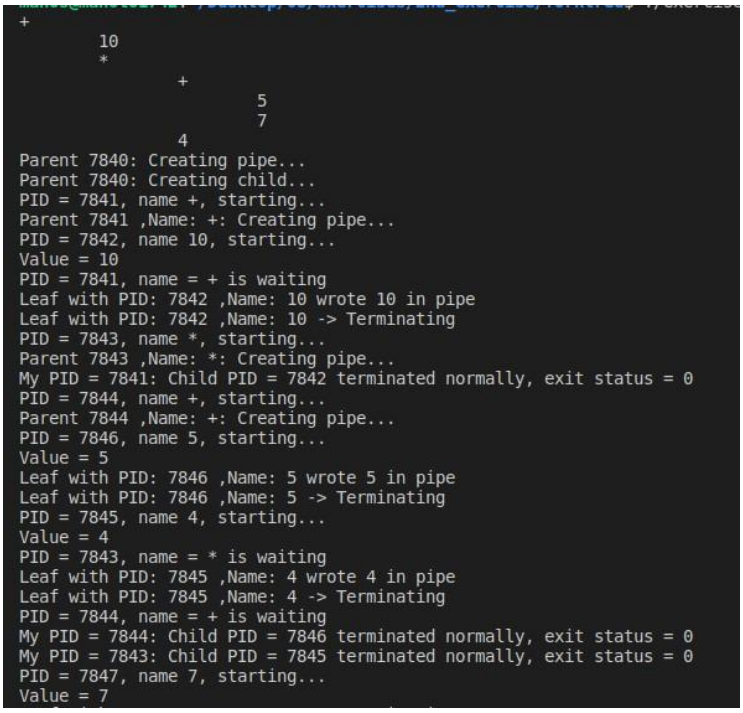
```
    explain_wait_status(pid, status);

    printf("Final Value = %i\n", final_value);
    printf("All done, exiting...\n");

    return 0;
}
```

Γνωρίζουμε ότι κάθε μη τερματικός κόμβος έχει ακριβώς δύο παιδιά, αναπαριστά τελεστή και το όνομα του είναι “+” ή “*”, ενώ κάθε τερματικός κόμβος είναι ακέραιος αριθμός. Εμείς θέλουμε οι διεργασίες φύλλα να γράφουν στον πατέρα την τιμή τους , ο πατέρας να εφαρμόζει τον τελεστή του στις τιμές των παιδιών του , να υπολογίζει το αποτέλεσμα και να το επιστρέφει στον δικό του πατέρα. Δηλαδή θέλουμε τα παιδιά να γράφουν στο πατέρα . Η main είναι παρόμοια με πριν και καλεί την αναδρομική συνάρτηση fork_pipe . Η fork_pipe κάνει τα εξής:

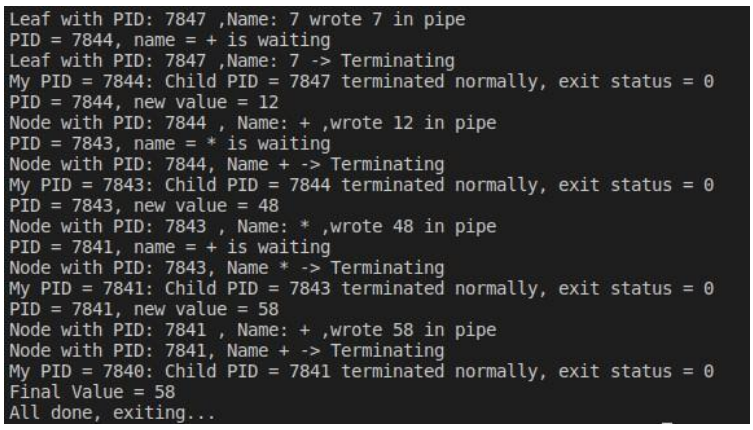
- Αρχικά, ελέγχει αν η τρέχουσα διεργασία έχει περισσότερα από δύο παιδιά, πράγμα που σημαίνει ότι έχουμε λάθος αρχείο εισόδου και τυπώνει κατάλληλο μήνυμα .
- Ελέγχει αν η τρέχουσα διεργασία είναι φύλλο και αν είναι καλεί την write με παραμέτρους το pid της διεργασίας πατέρα και την τιμή της και τα γράφει μέσω της σωλήνωσης στο πατέρα.
- Αν δεν είναι παιδί φύλλο τότε δημιουργεί τον σωλήνα (pipe) και κάνει fork για κάθε παιδί . Στη συνέχεια, κάθε παιδί κλείνει το άκρο ανάγνωσης και για κάθε παιδί καλείται η αναδρομική συνάρτηση fork_pipe , ενώ παράλληλα ο πατέρας κλείνει το άκρο εγγραφής και διαβάζει τις τιμές των παιδιών του από το pipe. Αφού διαβάσει τις τιμές των παιδιών καλεί την συνάρτηση compute_value η οποία ανάλογα με τον τελεστή της διεργασίας κάνει το κατάλληλο υπολογισμό και το αποτέλεσμα το επιστρέφει στον δικό του πατέρα.

Σημείωση: Στο παραπάνω πρόγραμμα δεν χρησιμοποιήθηκαν σήματα ούτε η εντολή sleep() διότι οι κλήσεις συστήματος read() και write() μας εξασφαλίζουν ότι το πρόγραμμα θα εκτελεστεί σωστά. Πιο συγκεκριμένα, η read() όταν κληθεί θα μπλοκάρει μέχρι να υπάρξει τουλάχιστον ένα byte για να διαβάσει(εκτός αν προκύψει error). Όταν βρει τουλάχιστον ένα byte θα διαβάσει όσα περισσότερα μπορεί και μετά θα επιστρέψει. Αντίστοιχα, λειτουργεί και η write() .Άρα, μέσω των read και write εξασφαλίζεται η επιθυμητή καθυστέρηση και έτσι οι λειτουργίες γίνονται με τη σωστή σειρά και το πρόγραμμα λειτουργεί με τον επιθυμητό τρόπο.

Εκτελούμε το παραπάνω πρόγραμμα με αρχείο εισόδου το expr.tree και προκύπτει το παρακάτω output:



```
+
        10
        *
                +
                        5
                        7
                4
Parent 7840: Creating pipe...
Parent 7840: Creating child...
PID = 7841, name +, starting...
Parent 7841 ,Name: +: Creating pipe...
PID = 7842, name 10, starting...
Value = 10
PID = 7841, name = + is waiting
Leaf with PID: 7842 ,Name: 10 wrote 10 in pipe
Leaf with PID: 7842 ,Name: 10 -> Terminating
PID = 7843, name *, starting...
Parent 7843 ,Name: *: Creating pipe...
My PID = 7841: Child PID = 7842 terminated normally, exit status = 0
PID = 7844, name +, starting...
Parent 7844 ,Name: +: Creating pipe...
PID = 7846, name 5, starting...
Value = 5
Leaf with PID: 7846 ,Name: 5 wrote 5 in pipe
Leaf with PID: 7846 ,Name: 5 -> Terminating
PID = 7845, name 4, starting...
Value = 4
PID = 7843, name = * is waiting
Leaf with PID: 7845 ,Name: 4 wrote 4 in pipe
Leaf with PID: 7845 ,Name: 4 -> Terminating
PID = 7844, name = + is waiting
My PID = 7844: Child PID = 7846 terminated normally, exit status = 0
My PID = 7843: Child PID = 7845 terminated normally, exit status = 0
PID = 7847, name 7, starting...
Value = 7
```



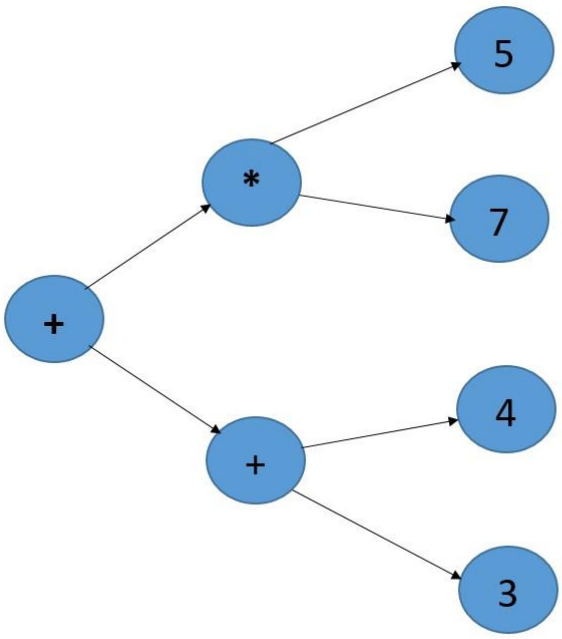
```
Leaf with PID: 7847 ,Name: 7 wrote 7 in pipe
PID = 7844, name = + is waiting
Leaf with PID: 7847 ,Name: 7 -> Terminating
My PID = 7844: Child PID = 7847 terminated normally, exit status = 0
PID = 7844, new value = 12
Node with PID: 7844 , Name: + ,wrote 12 in pipe
PID = 7843, name = * is waiting
Node with PID: 7844, Name + -> Terminating
My PID = 7843: Child PID = 7844 terminated normally, exit status = 0
PID = 7843, new value = 48
Node with PID: 7843 , Name: * ,wrote 48 in pipe
PID = 7841, name = + is waiting
Node with PID: 7843, Name * -> Terminating
My PID = 7841: Child PID = 7843 terminated normally, exit status = 0
PID = 7841, new value = 58
Node with PID: 7841 , Name: + ,wrote 58 in pipe
Node with PID: 7841, Name + -> Terminating
My PID = 7840: Child PID = 7841 terminated normally, exit status = 0
Final Value = 58
All done, exiting...
```

Όπως βλέπουμε παραπάνω τα παιδιά 5 και 7 γράφουν τις τιμές τους στον πατέρα «+» , ο οποίος εφαρμόζει τον τελεστή πρόσθεσης και γράφει 12 στον πατέρα «*». Ο πατέρας «*» διαβάζει το 12 και το 4 από τα παιδιά του και εφαρμόζει τον τελεστή του γινομένου και γράφει στο pipe του πατέρα ρίζα την τιμή 48 . Ο πατέρας ρίζα διαβάζει το 48 και το 10 από τα pipes των παιδιών του και εφαρμόζει τον τελεστή πρόσθεσης και γράφει στο pipe της main την τιμή 58 , όπως θα έπρεπε.

Ερωτήσεις

1) Εάν ανοίγαμε μία σωλήνωση για κάθε παιδί στο συγκεκριμένο παράδειγμα θα χρειαζόντουσαν 7 σωληνώσεις .Ωστόσο, εφόσον η πρόσθεση και ο πολλαπλασιασμός είναι αντιμεταθετικές πράξεις μπορούμε να έχουμε μόνο ένα pipe για κάθε τριάδα γονιού και παιδιών. Τότε θα χρειαζόμασταν (όπως βλέπουμε και στη παραπάνω έξοδο) 4 σωληνώσεις. Αν είχαμε αφαιρέσεις και διαιρέσεις οι οποίες δεν είναι αντιμεταθετικές θα χρειαζόταν ένα pipe για κάθε ζεύγος πατέρα-παιδιού, προκειμένου να ξέρουμε από ποιο παιδί λάβαμε τι. Αν για παράδειγμα στο «5+7» είχαμε αφαίρεση θα έπρεπε να ξέρουμε αν θα έπρεπε να εκτελέσουμε «5-7» ή «7-5».

2) Σε αυτή τη περίπτωση εφόσον μπορούν να εκτελούνται παραπάνω από μία διεργασίες παράλληλα, θα μπορούσαν πράξεις που ανήκουν σε διαφορετικά κλαδιά του δέντρου και βρίσκονται στο ίδιο επίπεδο να εκτελούνται παράλληλα και με αυτό τον τρόπο θα μπορούσαμε να πετύχουμε μεγαλύτερη ταχύτητα. Για παράδειγμα στο ακόλουθο δέντρο οι λειτουργίες 5*7 και 4+3 θα μπορούσαν να εκτελεστούν παράλληλα εξοικονομώντας χρόνο.



Ωστόσο, στη χειρότερη περίπτωση δεν εξοικονομούμε χρόνο . Για παράδειγμα στο δέντρο του παραδείγματος που τρέξαμε η διεργασία 4 δεν μπορεί να εκτελεστεί παράλληλα με την διεργασία «+» διότι η «+» πρέπει να περιμένει πρώτα για τα παιδιά της.

Makefile

```
.PHONY: all clean

all: fork-example tree-example ask2-fork ask2-signals exercise_1.1
exercise_1.2 exercise_1.3 exercise_1.4 exercise_1.42

CC = gcc
CFLAGS = -g -Wall -O2
SHELL= /bin/bash

tree-example: tree-example.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

fork-example: fork-example.o proc-common.o
    $(CC) $(CFLAGS) $^ -o $@

ask2-fork: ask2-fork.o proc-common.o
    $(CC) $(CFLAGS) $^ -o $@

ask2-signals: ask2-signals.o proc-common.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

exercise_1.1: exercise_1.1.o proc-common.o
    $(CC) $(CFLAGS) $^ -o $@

exercise_1.2: exercise_1.2.o proc-common.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

exercise_1.3: exercise_1.3.o proc-common.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

exercise_1.4: exercise_1.4.o proc-common.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

exercise_1.42: exercise_1.42.o proc-common.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

%.s: %.c
```

```
    $(CC) $(CFLAGS) -S -fverbose-asm $<

%.o: %.c
    $(CC) $(CFLAGS) -c $<

%.i: %.c
    gcc -Wall -E $< | indent -kr > $@

clean:
    rm -f *.o tree-example fork-example pstree-this ask2-
{fork,tree,signals,pipes}
```

Προαιρετικές ερωτήσεις

3.1 Οι παράγοντες που καθορίζουν αν ο παράλληλος υπολογισμός μιας αριθμητικής έκφρασης θα είναι ταχύτερος από τον αντίστοιχο υπολογισμό σε μία μόνο διεργασία (σειριακή εκτέλεση) είναι οι εξής:

1) Η μορφή του δέντρου εισόδου και σε ποιο βαθμό αυτή υποστηρίζει την παραλληλία στον υπολογισμό των αριθμητικών εκφράσεων. Προκειμένου να αξιοποιήσουμε αυτή τη παραλληλία θέλουμε να υπάρχουν στο ίδιο επίπεδο πολλοί κόμβοι ώστε να μπορούν να εκτελούνται παράλληλα και να επιτυγχάνονται ταχύτεροι υπολογισμοί. Σε περίπτωση που δεν συμβαίνει αυτό ορισμένοι επεξεργαστές μπορεί να χρειαστεί να μείνουν αδρανείς περιμένοντας τα αποτελέσματα άλλων επεξεργαστών με αποτέλεσμα να μειώνεται η απόδοση του συστήματος.
2) Η χρονική καθυστέρηση που απαιτείται για την επικοινωνία μεταξύ των πυρήνων και τη μεταφορά δεδομένων.
3) Η πολυπλοκότητα των ζητούμενων υπολογισμών. Σε περίπτωση που εμφανισθούν απαιτητικοί χρονικά υπολογισμοί είναι προτιμότερο να έχουμε πολυπύρηνο σύστημα επεξεργαστών .

3.2 Με την υβριδική υλοποίηση αξιοποιούμε τα πλεονεκτήματα της παράλληλης επεξεργασίας ελαχιστοποιώντας ταυτόχρονα τα μειονεκτήματα της. Για να το πετύχουμε αυτό εντοπίζουμε από ποιο βάθος μ του δέντρου και μετά είναι πιο αποδοτική η σειριακή εκτέλεση. Συνήθως, αυτό το βάθος βρίσκεται κοντά στις διεργασίες φύλλα όπου έχουν μείνει λίγες διεργασίες για την παραγωγή του τελικού αποτελέσματος . Ωστόσο, πρέπει να λάβουμε υπόψη μας ότι όσο αυξάνεται το βάθος μ του δέντρου τόσο αυξάνεται και το πλήθος των κόμβων σε ένα επίπεδο και άρα αυξάνεται και η απαίτηση σε πυρήνες και πόρους. Συνεπώς, από ένα σημείο και μετά η περαιτέρω αύξηση του βάθους μ έχει περισσότερο κόστος παρά κέρδος.