

**ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**  
**ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ**  
**ΥΠΟΛΟΓΙΣΤΩΝ**



**ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΤΩΝ**

(2019-2020)

*2<sup>η</sup> ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΝΑΦΟΡΑ*

Ομάδα: oslaba36

Ονοματεπώνυμο: Χρήστος Τσούφης – 03117176

Νίκος Χαράλαμπος Χαιρόπουλος – 03119507

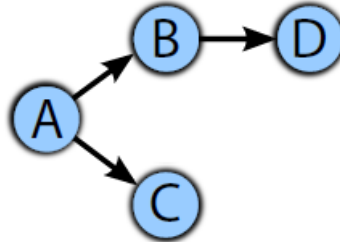
Εκτέλεση Εργαστηρίου: 06/04/2020

Σημείωση: Η άσκηση αυτή επιλύθηκε ατομικά από τον καθένα μας για την καλύτερη κατανόηση των εννοιών οπότε κρίνεται σκόπιμο να παρουσιαστούν και οι δύο προσπάθειες.

## Άσκηση 2: Διαχείριση Διεργασιών & Διαδιεργασιακή Επικοινωνία

### 1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Κατασκευάζεται πρόγραμμα το οποίο δημιουργεί το παρακάτω δέντρο διεργασιών.



Οι διεργασίες δημιουργούνται και διατηρούνται ενεργές για ορισμένο χρονικό διάστημα, ώστε ο χρήστης να έχει τη δυνατότητα παρατήρησης του δέντρου. Οι διεργασίες-φύλλα εκτελούν τη κλήση `sleep()`, ενώ οι διεργασίες σε ενδιάμεσους κόμβους περιμένουν τον τερματισμό των διεργασιών-παιδιών τους. Κάθε διεργασία εκτυπώνει κατάλληλο μήνυμα κάθε φορά που περνά από φάση σε φάση (π.χ., εκκίνηση, αναμονή για τερματισμό παιδιών, τερματισμό), ώστε να είναι δυνατή η επαλήθευση της σωστής λειτουργίας του προγράμματος. Για τον διαχωρισμό των διεργασιών, κάθε μία θα τερματίζει με διαφορετικό κωδικό επιστροφής:  $A = 16$ ,  $B = 19$ ,  $C = 17$ ,  $D = 13$ . Χρησιμοποιήθηκαν βοηθητικά αρχεία κώδικα από τον κατάλογο: `/home/oslab/code/forktree`:

- `fork-example.c`: παράδειγμα χρήσης των `fork()`, `wait()`.
- `proc-common.{h,c}`: βοηθητικές συναρτήσεις για το χειρισμό διεργασιών. Η συνάρτηση `explain_wait_status()` εκτυπώνει κατάλληλο μήνυμα ανάλογα με το επιστρεφόμενο `status` της `wait()`. Η συνάρτηση `show_pstree()` εμφανίζει το δέντρο διεργασιών με ρίζα δεδομένη διεργασία.
- `ask2-fork.c`: σκελετός του προγράμματος, απ' όπου μπορείτε να ξεκινήσετε. Κατασκευάζει μια διεργασία-παιδί (τη ρίζα του δέντρου σας), περιμένει να κατασκευαστεί το δέντρο διεργασιών (κοιμάται για λίγο) και καλεί την `show_pstree()`.

Οπότε, με βάση την `ask2-fork.c` προκύπτει ο πηγαίος κώδικας:

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

#include "proc-common.h"

#define SLEEP_PROC_SEC 10
#define SLEEP_TREE_SEC 3

/*
 * Create this process tree:
```

```

/* A--B---D
 *   `--C
 */
void fork_procs(void)
{
    /*
     * initial process is A.
     */
    int status;

    change_pname("A");
    printf("A: Process A created succesfully...\n");
    printf("A: Ready to create child B..\n");

    // Forking in order to create B
    pid_t p = fork();
    if (p < 0) {
        /* fork failed */
        perror("Forking failed");
        exit(1);
    }
    if (p == 0) {
        /*Child B process */
        change_pname("B");
        printf("B: I was created succesfully...\n");
        printf("B: Ready to create child D..\n");

        // Forking in order to create D
        p = fork();
        if (p < 0) {
            /* fork failed */
            perror("Forking failed");
            exit(1);
        }
        if (p == 0) {
            /*Child D process */
            change_pname("D");
            printf("D: I was created succesfully...\n");
            printf("D: Sleeping...\n");
            sleep(SLEEP_PROC_SEC);
            printf("D: Exiting...\n");
            exit(13);
        }
        p = wait(&status); //Node B waiting
        explain_wait_status(p, status);
        printf("B: Exiting...\n");
        exit(19);
    }
}

```

```

    // Forking in order to create C
    p = fork();
    if (p < 0) {
        /* fork failed */
        perror("Forking failed");
        exit(1);
    }
    if (p == 0) {
        /*Child C process */
        change_pname("C");
        printf("C: I was created succesfully...\n");
        printf("C: Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("C: Exiting...\n");
        exit(17);
    }

    //Wait for 2 children to terminate
    p = wait(&status);
    explain_wait_status(p, status);

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

    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);

    /* 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 */
    pid = wait(&status);
    explain_wait_status(pid, status);
    return 0;
}

```

Όπου, proc-common.c :

```

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

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

#include "proc-common.h"

void
wait_forever(void)

```

```

{
    do {
        sleep(100);
    } while (1);
}

/*
 * This function performs some not-so-useful computation.
 * Its amount is determined by the value of count.
 */
void compute(int count)
{
    long i;
    volatile long junk;

    junk = 0;
    for (i = 0; i < count * 1000000; i++) {
        junk++;
    }
}

/*
 * Changes the process name, as appears in ps or pstree,
 * using a Linux-specific system call.
 */
void
change_pname(const char *new_name)
{
    int ret;
    ret = prctl(PR_SET_NAME, new_name);
    if (ret == -1){
        perror("prctl set_name");
        exit(1);
    }
}

/*
 * This function receives an integer status value,
 * as returned by wait()/waitpid() and explains what
 * has happened to the child process.
 *
 * The child process may have:
 *     * been terminated because it received an unhandled signal (WIFSIG
NALED)
 *     * terminated gracefully using exit() (WIFEXITED)
 *     * stopped because it did not handle one of SIGTSTP, SIGSTOP, SIGT
IN, SIGTTOU
 *     (WIFSTOPPED)

```

```

*
* For every case, a relevant diagnostic is output to standard error.
*/
void
explain_wait_status(pid_t pid, int status)
{
    if (WIFEXITED(status))
        fprintf(stderr, "My PID = %ld: Child PID = %ld terminated norma
lly, exit status = %d\n",
                (long) getpid(), (long) pid, WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        fprintf(stderr, "My PID = %ld: Child PID = %ld was terminated b
y a signal, signo = %d\n",
                (long) getpid(), (long) pid, WTERMSIG(status));
    else if (WIFSTOPPED(status))
        fprintf(stderr, "My PID = %ld: Child PID = %ld has been stopped
by a signal, signo = %d\n",
                (long) getpid(), (long) pid, WSTOPSIG(status));
    else {
        fprintf(stderr, "%s: Internal error: Unhandled case, PID = %ld,
status = %d\n",
                __func__, (long) pid, status);
        exit(1);
    }
    fflush(stderr);
}

/*
* Make sure all the children have raised SIGSTOP,
* by using waitpid() with the WUNTRACED flag.
*
* This will NOT work if children use pause() to wait for SIGCONT.
*/
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 unexpe
ctedly!\n",
                    (long) p);

```

```

        exit(1);
    }
}

/*
 * Print the process tree rooted at process with PID p.
 */
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);
    }
}

/*
 * Create a shared memory area, usable by all descendants of the callin
g process.
 */
void *create_shared_memory_area(unsigned int numbytes)
{
    int pages;
    void *addr;

    if (numbytes == 0) {
        fprintf(stderr, "%s: internal error: called for numbytes == 0\n
", __func__);
        exit(1);
    }

    /* Determine the number of pages needed, round up the requested num
ber of pages */
    pages = (numbytes - 1) / sysconf(_SC_PAGE_SIZE) + 1;

    /* Create a shared, anonymous mapping for this number of pages */
    addr = mmap(NULL, pages * sysconf(_SC_PAGE_SIZE),
                PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED) {

```



```

        perror("create_shared_memory_area: mmap failed");
        exit(1);
    }

    return addr;
}

```

Και η proc-common.h:

```

#ifndef PROC_COMMON_H
#define PROC_COMMON_H

/*****
*****
* Helper Functions
*/

/* does useless computation */
void compute(int count);

/* Does nothing and never returns. */
void wait_forever(void);

/*
* Print a nice diagnostic based on {pid, status}
* as returned by wait() or waitpid().
*/
void explain_wait_status(pid_t pid, int status);

/*
* This function makes sure a number of children processes
* have raised SIGSTOP, by using waitpid() with the WUNTRACED flag.
*
* This will NOT work if children use pause() to wait for SIGCONT.
*/
void wait_for_ready_children(int cnt);

/* Change the name of the process. */
void change_pname(const char *new_name);

/* Print the process tree rooted at process with PID p. */
void show_pstree(pid_t p);

/*
* Create a shared memory area, usable by all descendants of the calling
* process.
*/

```

```
void *create_shared_memory_area(unsigned int numbytes);

#endif /* PROC_COMMON_H */
```

Και το Makefile:

```
CC = gcc

CFLAGS = Wall

all: ask2-fork.o proc-common.o

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

proc-common.o: proc-common.c

    $(CC) -$(CFLAGS) -c proc-common.c

ask2-fork.o: ask2-fork.c

    $(CC) -$(CFLAGS) -c ask2-fork.c

clean:

    rm ask2-fork.o proc-common.o ask2-fork
```

Και το output είναι:

```
oslaba36@os-nodel:~/exe2/ask21$ ./ask2-fork
A: Process A created succesfully...
A: Ready to create child B..
B: I was created succesfully...
B: Ready to create child D..
C: I was created succesfully...
C: Sleeping...
D: I was created succesfully...
D: Sleeping...

A(24331) └─ B(24332) ── D(24334)
          └─ C(24333)

C: Exiting...
D: Exiting...
My PID = 24331: Child PID = 24333 terminated normally, exit status = 17
My PID = 24332: Child PID = 24334 terminated normally, exit status = 13
B: Exiting...
My PID = 24331: Child PID = 24332 terminated normally, exit status = 19
A: Exiting...
My PID = 24330: Child PID = 24331 terminated normally, exit status = 16
oslaba36@os-nodel:~/exe2/ask21$
```

## Ερωτήσεις:

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

Αυτό που θα συμβεί αν τερματιστεί πρόωρα η διεργασία A δίνοντας `kill -KILL <A_pid>`, είναι ότι οι διεργασίες-παιδιά της θα είναι τώρα “orphan” processes (έτσι χαρακτηρίζονται οι διεργασίες των οποίων ο γονέας πεθαίνει ή σκοτώνεται πριν από τις ίδιες) και θα “υιοθετηθούν” από την `init` που κάνει συνεχώς `wait()` (η οποία είναι η πρώτη από ένα σύνολο διεργασιών που τρέχουν στο παρασκήνιο για την λειτουργία ενός Unix-based συστήματος).

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

Με την αντικατάσταση της εντολής `show_pstree(pid)` με την `show_pstree(getpid())` αυτό που συμβαίνει είναι το εξής:

Το `pid` που επιστρέφει η κλήση της `fork` από την διεργασία `ask2-fork` για την δημιουργία της διεργασίας A, είναι το 24.930, αυτό δηλαδή της A. Δίνοντάς το ως όρισμα `show_pstree(pid)` θα εμφανίσει το δέντρο διεργασιών με ρίζα το A.

```
oslab36@os-nodel:~/exe2/ask21$ ./ask2-fork
A: Process A created succesfully...
A: Ready to create child B..
B: I was created succesfully...
B: Ready to create child D..
C: I was created succesfully...
C: Sleeping...
D: I was created succesfully...
D: Sleeping...

ask2-fork(24929) — A(24930) — B(24931) — D(24933)
                  |
                  | — C(24932)
                  |
                  | — sh(24935) — pstree(24936)

C: Exiting...
D: Exiting...
My PID = 24930: Child PID = 24932 terminated normally, exit status = 17
My PID = 24931: Child PID = 24933 terminated normally, exit status = 13
B: Exiting...
My PID = 24930: Child PID = 24931 terminated normally, exit status = 19
A: Exiting...
My PID = 24929: Child PID = 24930 terminated normally, exit status = 16
oslab36@os-nodel:~/exe2/ask21$
```

Από την άλλη, η κλήση της συνάρτησης `getpid()`, επιστρέφει το `pid` της διεργασίας που πραγματοποιεί την κλήση, και στη συγκεκριμένη περίπτωση που είναι το `pid` της διεργασίας `ask2-fork` (είχαμε εκτελέσει `./ask2-fork`). Έτσι, με όρισμα τη συνάρτηση `getpid()`, εμφανίζεται ολόκληρο το δέντρο διεργασιών με ρίζα τη διεργασία `ask2-fork`.

Επίσης, επισημαίνεται ότι σε αυτό το δέντρο περιλαμβάνονται οι διεργασίες sh (standard command language interpreter), με παιδί το pstree. Παρατηρώντας το αρχείο που περιέχει την υλοποίηση της show\_pstree, προκύπτει ότι οι διεργασίες αυτές δημιουργούνται κατά την κλήση της συνάρτησης show\_pstree.


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

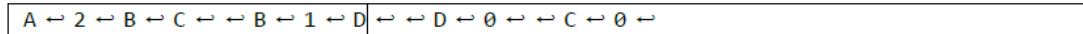
    snprintf(cmd, sizeof(cmd), "echo; echo; pstree -G -c -p %ld;
echo; echo",
             (long)p);
    cmd[sizeof(cmd)-1] = '\\0';
    ret = system(cmd);
    if (ret < 0) {
        perror("system");
        exit(104);
    }
}
```

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

Ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης διότι το σύστημα έχει πεπερασμένους πόρους. Τόσο τα blocks που καταγράφουν τα pids στη μνήμη όσο και οι απαραίτητοι πόροι (μνήμη, υπολογιστική ισχύς) που απαιτούνται για την κάθε διεργασία είναι πεπερασμένοι. Συνεπώς, για την εύρυθμη λειτουργία και την αποφυγή του ενδεχομένου να “crashάρει” το σύστημα είναι συνετό να τίθενται κάποιοι περιορισμοί αναφορικά με το πλήθος των διεργασιών ανά χρήστη. Παράδειγμα προβληματικής περίπτωσης αποτελεί η επίθεση τύπου denial-of-service fork-bomb, κατά την οποία μία διεργασία κάνει επαναλαμβανόμενα fork τον εαυτό της, επιβραδύνοντας ή “crashάροντας” το σύστημα.

## **1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών**

Κατασκευάζεται πρόγραμμα που δημιουργεί αυθαίρετα δέντρα διεργασιών βάσει αρχείου εισόδου. Το πρόγραμμα θα βασιστεί σε αναδρομική συνάρτηση, η οποία θα καλείται για κάθε κόμβο του δέντρου. Αν ο κόμβος του δέντρου έχει παιδιά, η συνάρτηση θα δημιουργεί τα παιδιά και θα περιμένει να τερματιστούν. Αν όχι, η συνάρτηση θα καλεί τη sleep() με προκαθορισμένο όρισμα. Το αρχείο εισόδου περιέχει την περιγραφή ενός δέντρου κόμβο προς κόμβο, ξεκινώντας από τη ρίζα. Για κάθε κόμβο δίνεται το όνομά του, ο αριθμός των παιδιών του και τα ονόματά τους. Η περιγραφή ενός κόμβου από τον επόμενο χωρίζεται με κενή γραμμή. Η διαδικασία επαναλαμβάνεται αναδρομικά για κάθε ένα από τα παιδιά του, ενώ οι κόμβοι πρέπει να είναι διατεταγμένοι κατά βάθος. Για παράδειγμα, το δέντρο διεργασιών του προηγούμενου σχήματος προέκυψε από το αρχείο εισόδου του παρακάτω σχήματος. Το σύμβολο  δείχνει αλλαγή γραμμής.



Σας δίνεται βιβλιοθήκη `tree.h` η οποία αναλαμβάνει να διαβάσει το αρχείο εισόδου και να κατασκευάσει την αναπαράστασή του στη μνήμη. Κάθε κόμβος ορίζεται από τη δομή `struct tree_node`, που περιέχει τον αριθμό των παιδιών `nr_children`, το όνομα του κόμβου και δείκτη προς περιοχή όπου βρίσκονται συνεχόμενα αποθηκευμένες `nr_children` δομές, μία για κάθε κόμβο-παιδί. Η βιβλιοθήκη προσφέρει δύο συναρτήσεις:

- `get_tree_from_file(const char *filename)`: διαβάζει το δέντρο, κατασκευάζει την αναπαράστασή του στη μνήμη κι επιστρέφει δείκτη στη ρίζα.
- `print_tree(struct tree_node *root)`: διατρέχει το δέντρο με ρίζα `root` και εκτυπώνει τα στοιχεία του στο standard output.

Το `tree-example.c` περιέχει παράδειγμα χρήσης της βιβλιοθήκης. Διαβάζει το αρχείο εισόδου, κατασκευάζει το δέντρο και το εμφανίζει στην οθόνη. Για να το εμφανίσει, το διατρέχει ακολουθώντας τους δείκτες `children` των δομών `struct tree_node`.

Με κατάλληλες αλλαγές στο `ask2-fork.c` προκύπτει το `ask2-tree.c` :

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

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

#define SLEEP_PROC_SEC 10
#define SLEEP_TREE_SEC 3

void fork_procs(struct tree_node *root) {
    char *name = root->name;
    change_pname(name);
    int status;

    //If process is a leaf node
    if (root->nr_children == 0) {
        printf("%s: Sleeping...\n", name);
        sleep(SLEEP_PROC_SEC);
        printf("%s: Exiting...\n", name);
        exit(getpid());
    }

    pid_t p;
    // Iterate for every child
    int i;
    for (i=0; i < root->nr_children; i++) {
```

```

        printf("%s: Ready to create child %s...\n", name,
               (root->children + i)->name);
        p = fork();
        if (p < 0) {
            /* fork failed */
            perror("Forking failed");
            exit(1);
        }

        if (p == 0) {
            /*Child process */
            printf("%s: I was created succesfully...\n",
                   (root->children + i)->name);
            fork_procs(root->children + i);
        }
    }
    for (i=0; i < root->nr_children; i++) {
        p = wait(&status); //Node B waiting
        explain_wait_status(p, status);
    }
    printf("%s: Exiting...\n", name);
    exit(getpid());
}

/*
 * 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[])
{
    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;

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

```

Όπου το tree.c είναι:

```

#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"

#define BUFF_SIZE 1024

```

```

static void
__print_tree(struct tree_node *root, int level)
{
    int i;
    for (i=0; i<level; i++)
        printf("\t");
    printf("%s\n", root->name);

    for (i=0; i < root->nr_children; i++){
        __print_tree(root->children + i, level + 1);
    }
}

void
print_tree(struct tree_node *root)
{
    __print_tree(root, 0);
}

static char *
read_line(FILE *file, char *buff, size_t buff_size)
{
    char *ret;
    size_t ret_len;

    ret = fgets(buff, buff_size, file);

    /* sanity check */
    if (ret == NULL){
        return ret;
    }

    ret_len = strlen(ret);
    if (ret_len == buff_size - 1){
        fprintf(stderr, "line too long: %s\n", buff);
        exit(1);
    }

    if (ret_len > 0)
        buff[ret_len - 1] = '\0'; /* remove \n */

    return ret;
}

static char *
read_empty_line(FILE *file, char *buff, size_t buff_size)
{

```



```

    char *ret;
    ret = read_line(file, buff, buff_size);
    if (ret != NULL && strlen(ret) != 0){
        fprintf(stderr, "expecting an empty line: %s", buff);
        exit(1);
    }

    return ret;
}

static char *
read_non_empty_line(FILE *file, char *buff, size_t buff_size)
{
    char *ret;
    ret = read_line(file, buff, buff_size);

    if (ret == NULL){
        fprintf(stderr, "unexpected EOF\n");
        exit(1);
    }

    if (strlen(ret) == 0){
        fprintf(stderr, "Unexpected empty line\n");
        exit(1);
    }

    return ret;
}

static char *
find_block_start(FILE *file, char *buff, size_t buff_size)
{
    char *line;
    for (;;) {
        line = read_line(file, buff, buff_size);
        if (line == NULL) /* EOF */
            break;
        if (strlen(line) == 0 || line[0] == '#')
            continue; /* comment or empty line */
        else
            break;
    }

    return line;
}

/*
 * recursively parse tree file, creating nodes

```

```

    */
static struct tree_node *
parse_node(FILE *file, struct tree_node *node)
{
    char buff[BUFF_SIZE], *name, *num_str;
    unsigned nr_children;
    int i;

    name = find_block_start(file, buff, BUFF_SIZE);
    if (name == NULL){ /* EOF */
        /* empty file, do nothing */
        if (node == NULL)
            return NULL;
        /* otherwise, terminate parsing */
        fprintf(stderr, "expecting: %s and got EOF\n", node->name);
        exit(1);
    }

    /* If no node given, allocate one -- this is used for root
     * If node is given, check that the names match */
    if (node == NULL){
        node = calloc(1, sizeof(struct tree_node));
        if (node == NULL){
            fprintf(stderr, "node allocation failed\n");
            exit(1);
        }
        snprintf(node->name, NODE_NAME_SIZE, "%s", name);
    } else if (strncmp(node->name, name, NODE_NAME_SIZE) != 0){
        fprintf(stderr, "nodes must be placed in a DFS order\n");
        fprintf(stderr, "expecting: %s and got: %s\n", node-
>name, name);
        exit(1);
    }

    /* read number of children */
    num_str = read_non_empty_line(file, buff, BUFF_SIZE);
    nr_children = node->nr_children = atol(num_str);

    /* allocate children */
    if (nr_children != 0){
        node->children = malloc(sizeof(struct tree_node)*nr_children);
        if (node->children == NULL){
            fprintf(stderr, "allocate children failed\n");
            exit(1);
        }
    }

    /* read children names */

```

```

    for (i=0; i<nr_children; i++){
        name = read_non_empty_line(file, buff, BUFF_SIZE);
        snprintf(node->children[i].name, NODE_NAME_SIZE, "%s", name);
        //printf("%s\n", node->children[i].name);
    }
    read_empty_line(file, buff, BUFF_SIZE);

    /* parse children */
    for (i=0; i<nr_children; i++){
        parse_node(file, &node->children[i]);
    }
    return node;
}

struct tree_node *
get_tree_from_file(const char *filename)
{
    FILE *file;
    assert(BUFF_SIZE >= NODE_NAME_SIZE);
    struct tree_node *root;

    file = fopen(filename, "r");
    if (file == NULL){
        perror(filename);
        exit(1);
    }

    root = parse_node(file, NULL);

    fclose(file);

    return root;
}

```

Και το tree.h είναι:

```

#ifndef TREE_H
#define TREE_H

/*****
*****/

/* Data structure definitions
*/
#define NODE_NAME_SIZE 16
/* tree node structure */
struct tree_node {
    unsigned    nr_children;
    char        name[NODE_NAME_SIZE];
    struct tree_node *children;
};

```

```

/*****
*****
 * Helper Functions
 */

/* returns the root node of the tree defined in a file */
struct tree_node *get_tree_from_file(const char *filename);

void print_tree(struct tree_node *root);

#endif /* TREE_H */

```

Τα proc-common.{c, h} αρχεία παραμένουν ίδια.

Είσοδος είναι το graph1.tree :

```

# file that defines the tree
# lines starting with '#' are comments
# . each block defines a node
# . each node is defined as:
#   1st line:      name of node
#   2nd line:      number of children
#   subsequent lines: name(s) of children
# . blocks are seperated with empty lines
# . no comments are allowed within a block
# . nodes must be placed in a DFS order

A
3
B
C
D

B
1
E

E
0

C
1
F

F
0

D
0

```

```
CC = gcc
CFLAGS = Wall

all: ask2-tree.o proc-common.o tree.o
    $(CC) -$(CFLAGS) ask2-tree.o proc-common.o tree.o -o ask2-tree

tree.o: tree.c
    $(CC) -$(CFLAGS) -c tree.c

proc-common.o: proc-common.c
    $(CC) -$(CFLAGS) -c proc-common.c

ask2-tree.o: ask2-tree.c
    $(CC) -$(CFLAGS) -c ask2-tree.c

clean:
    rm ask2-tree.o proc-common.o tree.o ask2-tree
```

```

oslab36@os-nodel:~/exe2/ask21$ ./ask2-tree graph1.tree
A
  B
    E
  C
    F
  D
A: Ready to create child B...
A: Ready to create child C...
B: I was created succesfully...
A: Ready to create child D...
B: Ready to create child E...
C: I was created succesfully...
C: Ready to create child F...
D: I was created succesfully...
D: Sleeping...
E: I was created succesfully...
E: Sleeping...
F: I was created succesfully...
F: Sleeping...

ask2-tree (25855) — A (25856) — B (25857) — E (25860)
                      |
                      | — C (25858) — F (25861)
                      | — D (25859)
                      |
                      — sh (25862) — pstree (25863)

D: Exiting...
E: Exiting...
My PID = 25856: Child PID = 25859 terminated normally, exit status = 3
F: Exiting...
My PID = 25857: Child PID = 25860 terminated normally, exit status = 4
B: Exiting...
My PID = 25858: Child PID = 25861 terminated normally, exit status = 5
C: Exiting...
My PID = 25856: Child PID = 25857 terminated normally, exit status = 1
My PID = 25856: Child PID = 25858 terminated normally, exit status = 2
A: Exiting...
My PID = 25855: Child PID = 25856 terminated normally, exit status = 0
oslab36@os-nodel:~/exe2/ask21$

```

### **Ερωτήσεις:**

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

Στην αρχή δημιουργείται ο πατέρας και στη συνέχεια με τη σειρά που αναγράφονται στο αρχείο εισόδου τα παιδιά του. Στο προηγούμενο παράδειγμα πρώτα A, μετά B, έπειτα C και τέλος D. Μετά το πρώτο επίπεδο όμως η σειρά δεν είναι αυστηρά καθορισμένη καθώς η κάθε διεργασία θα δημιουργήσει ανεξάρτητα από τις άλλες τα δικά της παιδιά, χωρίς κάποιο συγχρονισμό οπότε, το αποτέλεσμα δεν θα είναι με συγκεκριμένη, αυστηρά ορισμένη σειρά.

Όσον αφορά τον τερματισμό, ο κάθε πατέρας πριν κάνει exit() περιμένει να τερματιστούν όλα τα παιδιά. Εν προκειμένω, πρώτα θα τερματιστούν τα φύλλα (D, E, F). Με την χρήση καθυστερήσεων, ούτε η σειρά εξόδου είναι αυστηρά καθορισμένη, καθώς εξαρτάται από τη σειρά δημιουργίας των φύλλων, ή ισοδύναμα από το ποιας διεργασίας η διαδικασία sleep κλήθηκε νωρίτερα. Συνοψίζοντας, για ένα συγκεκριμένο level του δέντρου, οι διεργασίες που ανήκουν στο ίδιο level (ίδιο βάθος) τερματίζουν χωρίς κάποια προκαθορισμένη σειρά, ικανοποιώντας μόνο την απαίτηση να έχουν τερματίσει πρώτα όλα τα παιδιά της.

### **1.3 Αποστολή & χειρισμός σημάτων**

Επέκταση του προγράμματος της 1.2 έτσι ώστε οι διεργασίες να ελέγχονται με χρήση σημάτων, για να εκτυπώνουν τα μηνύματά τους κατά βάθος ( Depth-First).

Κάθε διεργασία δημιουργεί τις διεργασίες-παιδιά της και αναστέλλει τη λειτουργία της έως ότου λάβει κατάλληλο σήμα εκκίνησης (SIGCONT). Όταν το λάβει, εμφανίζει ενημερωτικό μήνυμα κι ενεργοποιεί διαδοχικά τις διεργασίες-παιδιά: ξυπνάει μια διεργασία, περιμένει τον τερματισμό της και εκτυπώνει ανάλογο διαγνωστικό. Η διαδικασία εξελίσσεται αναδρομικά ξεκινώντας από τη διεργασία-ρίζα του δέντρου. Η αρχική διεργασία του προγράμματος στέλνει SIGCONT στη διεργασία-ρίζα, αφού παρουσιάσει το δέντρο διεργασιών στο χρήστη. Στην περίπτωση του δέντρου του σχήματος, τα μηνύματα ενεργοποίησης εκτυπώνονται με τη σειρά A, B, D, C .

Υποδείξεις:

- Μια διεργασία αναστέλλει τη λειτουργία της με χρήση της εντολής raise(SIGSTOP).
- Πριν το κάνει αυτό, έχει βεβαιωθεί ότι όλα τα παιδιά της έχουν αναστείλει τη λειτουργία τους με χρήση της συνάρτησης wait\_for\_ready\_children() που σας δίνεται. Χρησιμοποιήστε τη συνάρτηση explain\_wait\_status() μετά από wait() ή waitpid().
- Κατά την ανάπτυξη του προγράμματος μπορείτε να στείλετε μηνύματα χειροκίνητα, με χρήση της εντολής kill από κάποιο άλλο παράθυρο. Χρησιμοποιήστε την εντολή strace -f -p <pid> για να παρακολουθήσετε τις κλήσεις συστήματος που εκτελεί η διεργασία <pid> κι όλα τα παιδιά της.
- Ξεκινήστε από τον σκελετό ask2-signals.c που σας δίνεται.

Με κατάλληλες τροποποιήσεις προκύπτει το ask2-signals.c :

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

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

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

    // Generate children
    // If node is a leaf this loop won't be taken
    int childrenPID[root->nr_children];
    int i;
    for (i=0; i < root->nr_children; i++) {
        printf("%s: Ready to create child %s...\n", root->name,
               (root->children + i)->name);
        pid_t p = fork();
        if (p < 0) {
            /* fork failed */
            perror("Forking failed");
            exit(1);
        }

        if (p == 0) {
            /*Child process */
            printf("%s: I was created succesfully...\n",
                   (root->children + i)->name);
            fork_procs(root->children + i);
        }
        // Save child pid
        childrenPID[i] = p;
        printf("Child name: %s \t PID: %d\n",
               (root->children + i)->name, childrenPID[i]);
        // DFS creation, waiting for each child to change state
        wait_for_ready_children(1);
    }
}
```

```

    }
    /* ... */

    /*
     * Suspend Self
     */
    raise(SIGSTOP);
    printf("PID = %ld, name = %s: I just woke up...\n",
        (long)getpid(), root->name);

    for (i=0; i < root->nr_children; i++) {
        printf("PID = %ld, name = %s: Trying to wake up PID: %ld\n",
            (long)getpid(), root->name, (long)childrenPID[i]);
        kill(childrenPID[i], SIGCONT);
        int diedPID = wait(&status);
        explain_wait_status(diedPID, status);
        printf("\n");
    }
    printf("\nPID = %ld, name = %s: Antio mataie toute kosme...\n",
        (long)getpid(), root->name);
    exit(getpid());
}

/*
 * 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;
}

```

Με είσοδο το αρχείο test2.tree :

```

A
2
B
C

B
1
D

D
0

C
0

```

To Makefile είναι:

```
CC = gcc
CFLAGS = Wall

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

tree.o: tree.c
    $(CC) -$(CFLAGS) -c tree.c

proc-common.o: proc-common.c
    $(CC) -$(CFLAGS) -c proc-common.c

ask2-signals.o: ask2-signals.c
    $(CC) -$(CFLAGS) -c ask2-signals.c

clean:
    rm ask2-signals.o proc-common.o tree.o ask2-signals
```

Προκύπτει output:

```
oslaba36@os-nodel:~/exe2/ask21$ ./ask2-signals test2.tree
PID = 27090, name A, starting...
A: Ready to create child B...
Child name: B    PID: 27091
B: I was created succesfully...
PID = 27091, name B, starting...
B: Ready to create child D...
Child name: D    PID: 27092
D: I was created succesfully...
PID = 27092, name D, starting...
My PID = 27091: Child PID = 27092 has been stopped by a signal, signo = 19
My PID = 27090: Child PID = 27091 has been stopped by a signal, signo = 19
A: Ready to create child C...
Child name: C    PID: 27093
C: I was created succesfully...
PID = 27093, name C, starting...
My PID = 27090: Child PID = 27093 has been stopped by a signal, signo = 19
My PID = 27089: Child PID = 27090 has been stopped by a signal, signo = 19

A(27090)└─B(27091)──D(27092)
          └─C(27093)

PID = 27090, name = A: I just woke up...
PID = 27090, name = A: Trying to wake up PID: 27091
PID = 27091, name = B: I just woke up...
PID = 27091, name = B: Trying to wake up PID: 27092
PID = 27092, name = D: I just woke up...

PID = 27092, name = D: Antio mataie toute kosme...
My PID = 27091: Child PID = 27092 terminated normally, exit status = 212

PID = 27091, name = B: Antio mataie toute kosme...
My PID = 27090: Child PID = 27091 terminated normally, exit status = 211

PID = 27090, name = A: Trying to wake up PID: 27093
PID = 27093, name = C: I just woke up...

PID = 27093, name = C: Antio mataie toute kosme...
My PID = 27090: Child PID = 27093 terminated normally, exit status = 213

PID = 27090, name = A: Antio mataie toute kosme...
My PID = 27089: Child PID = 27090 terminated normally, exit status = 210
oslaba36@os-nodel:~/exe2/ask21$
```

### Ερωτήσεις:

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

Με τη χρήση καθυστερήσεων η συμπεριφορά των διεργασιών και κατά συνέπεια η λειτουργία του προγράμματος γίνονται απρόβλεπτες, οπότε ο συγχρονισμός είναι ανέφικτος. Ενδεικτικά επισημαίνεται ότι η “φωτογράφιση” του δέντρου διεργασιών από την `show_pstree` γίνεται σχεδόν τυχαία μετά από ένα προκαθορισμένο χρόνο, στον οποίο εμείς ευελπιστούμε να έχει κατασκευαστεί ολόκληρο το δέντρο. Ένα ακόμη ζήτημα είναι πως η DFS διάσχιση του δέντρου είναι αδύνατη.

Αυτά τα ζητήματα αντιμετωπίζονται με τη χρήση σημάτων καθώς είναι ευκολότερος ο έλεγχος του προγράμματος αφού μπορεί να ελεγχθεί η συμπεριφορά της κάθε διεργασίας ανάλογα με το τι σήμα αποστέλλεται σε αυτή. Έτσι, είναι δυνατός ο πλήρης έλεγχος της κατασκευής του δέντρου και πλέον, η υλοποίηση του DFS είναι εφικτή. Τέλος, η φωτογράφιση του δέντρου από τη `show_pstree` θα γίνει σίγουρα αφού θα έχει κατασκευαστεί ολόκληρο, χωρίς να τίθεται κάποιο ρίσκο.

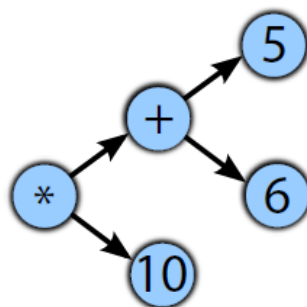
2. Ποιος ο ρόλος της `wait_for_ready_children()`; Τι εξασφαλίζει η χρήση της και τι πρόβλημα θα δημιουργούσε η παράλειψή της;

Η διεργασία `wait_for_ready_children()` περιμένει μέχρι ο αριθμός παιδιών που της δίνεται σαν όρισμα να αλλάξει κατάσταση. Η χρήση της μέσα στο `for loop` διασφαλίζει την κατά βάθος (DFS) σειρά εμφάνισης των μηνυμάτων. Συγκεκριμένα, κάθε διεργασία θα περιμένει κάποιο της παιδί να αλλάξει κατάσταση, προτού προχωρήσει στην εκτέλεση των επόμενων εντολών, επιτυγχάνοντας έτσι την DFS διάσχιση - δημιουργία του επιθυμητού δέντρου διεργασιών. Αντιθέτως, εάν παραλειπόταν, ο συγχρονισμός θα ήταν αδύνατος, και η σειρά δημιουργίας των διεργασιών του δέντρου δεν θα ήταν προκαθορισμένη. Έτσι, θα υπήρχε το ενδεχόμενο ο πατέρας να στείλει μήνυμα `SIGCONT` σε παιδί που δεν έχει κάνει `raise SIGSTOP` και να μην δημιουργηθεί εγκαίρως κάποια διεργασία.

Τέλος, γενικότερα, ο πατέρας ενός παιδιού πρέπει να καλεί τη συνάρτηση `wait()` (με τη βοήθεια της οποίας υλοποιείται η `wait_for_ready_children`) προκειμένου να αποφεύγεται η δημιουργία “zombie” παιδιών.

### 1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Επέκταση του προγράμματος της 1.2 ώστε να υπολογίζει δέντρα που αναπαριστούν αριθμητικές εκφράσεις. Για παράδειγμα, το δέντρο του παρακάτω σχήματος αναπαριστά την έκφραση  $10 \times (5+6)$ .



Θεωρήστε ότι τα αρχεία – και κατά συνέπεια τα παραγόμενα δέντρα – έχουν τους παρακάτω περιορισμούς:

- Κάθε μη τερματικός κόμβος<sup>1</sup> έχει ακριβώς δύο παιδιά, αναπαριστά τελεστή και το όνομά του είναι "+" ή "\*".
- Το όνομα κάθε τερματικού κόμβου είναι ακέραιος αριθμός.

Η αποτίμηση της έκφρασης θα γίνεται παράλληλα, δημιουργώντας μία διεργασία για κάθε κόμβο του δέντρου. Κάθε τερματική διεργασία επιστρέφει στη γονική την αριθμητική τιμή που της αντιστοιχεί. Κάθε μη-τερματική διεργασία λαμβάνει από τις διεργασίες-παιδιά τις τιμές των υπο-εκφράσεών τους, υπολογίζει την τιμή της και την επιστρέφει στη γονική της διεργασία. Η διεργασία-ρίζα επιστρέφει το συνολικό αποτέλεσμα στην αρχική διεργασία του προγράμματος, η οποία το εμφανίζει στην οθόνη.

Υποδείξεις:

- Χρησιμοποιήστε σωληνώσεις του Unix (pipes) για την επικοινωνία από τις διεργασίες-παιδιά στη γονική διεργασία.
- Κάθε διεργασία οφείλει να περιμένει τον τερματισμό των παιδιών της και να εκτυπώνει κατάλληλο διαγνωστικό, για να εντοπίζετε εγκαίρως προγραμματιστικά σφάλματα.
- Εκτυπώνετε ενδιάμεσα αποτελέσματα από κάθε διεργασία, βοηθά στο debugging.

Ο πηγαίος κώδικας της ask2-calculate.c:

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

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

#define SLEEP_PROC_SEC 10
#define SLEEP_TREE_SEC 3

void fork_procs(struct tree_node *root, int write_fd) {
    char *name = root->name;
    change_pname(name);
    printf("PID: %ld (%s): I was created succesfully...\n",
        (long)getpid(), root->name);
    //If process is a leaf node
    if (root->nr_children == 0) {
        int value = atoi(root->name);
        if(write(write_fd, &value, sizeof(value)) != sizeof(value)) {
            perror("Pipe Write Failed\n");
            exit(1);
        }
    }
}
```

```

    }
    printf("PID: %ld (%s): Wrote Value:%d on Pipe %d and is Exiting
...\\n",
        (long)getpid(), name, value, write_fd);
    sleep(SLEEP_PROC_SEC);
    exit(getpid());
}

pid_t p;
// Iterate for every child
// If node is not a leaf
int i;
int fd[2];
if (pipe(fd) < 0) {
    perror("Pipe Creation Failed");
    exit(1);
}
printf("PID: %ld (%s): Pipe [%d, %d] to children created successful
ly\\n",
        (long)getpid(), root->name, fd[0], fd[1]);

for (i = 0; i < 2; i++) {
    printf("PID: %ld (%s): Ready to create child %s...\\n",
        (long)getpid(), name, (root->children + i)->name);
    p = fork();
    if (p < 0) {
        /* fork failed */
        perror("Forking failed");
        exit(1);
    }

    if (p == 0) {
        /*Child process */
        fork_procs(root->children + i, fd[1]);
        printf("\\nWTF\\n");
    }
}

int operands[2];
int value;

for (i=0; i < 2; i++) {
    if (read(fd[0], &value, sizeof(value)) != sizeof(value)) {
        perror("Read from Pipe Failed");
        exit(1);
    }
    printf("PID: %ld (%s): Reading from pipe value no. %d : %d\\n",
        (long)getpid(), root->name, i, value);
}

```

```

        operands[i] = value;

        // p = wait(&status); //Node B waiting
        //explain_wait_status(p, status);
    }
    int result;

    if (strcmp(root->name, "+") == 0)
        result = operands[0] + operands[1];
    else
        result = operands[0] * operands[1];
    if (write(write_fd, &result, sizeof(result)) != sizeof(result)) {
        perror("Pipe Write Failed");
        exit(1);
    }
    printf("PID: %ld (%s): Wrote calculated result (%d) to Pipe %d\n",
           (long)getpid(), name, result, write_fd);
    sleep(SLEEP_PROC_SEC);
    exit(getpid());
}

/*
 * 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[])
{
    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 fd[2];

```

```

if (pipe(fd) < 0) {
    perror("Pipe Creation Failed");
    exit(1);
}
/* Fork root of process tree */
pid = fork();
if (pid < 0) {
    perror("main: fork");
    exit(1);
}
if (pid == 0) {
    /* Child */
    fork_procs(root, fd[1]);
    exit(getpid());
}
/*
 * 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(getpid());

int final_value;
/* for ask2-signals */
/* kill(pid, SIGCONT); */
if(read(fd[0], &final_value, sizeof(final_value))
    != sizeof(final_value)) {
    perror("Pipe Read Failed");
    exit(1);
}
printf("The result is %d\n", final_value);
return 0;
}

```

Με είσοδο το αρχείο myexpr.tree :

```
# file that defines the tree
# lines starting with '#' are comments
# . each block defines a node
# . each node is defined as:
#   1st line:          name of node
#   2nd line:          number of children
#   subsequent lines: name(s) of children
# . blocks are seperated with empty lines
# . no comments are allowed within a block
# . nodes must be placed in a DFS order
```

```
*
2
+
*

+
1
100

100
0

*
2
+
*

+
2
5
+

5
0

+
2
8
4

8
0

4
0

*
2
3
6

3
0

6
0
```



To Makefile είναι:

```
CC = gcc
CFLAGS = Wall

all: ask2-calculate.o proc-common.o tree.o
    $(CC) -$(CFLAGS) ask2-calculate.o proc-common.o tree.o -o
ask2-calculate

tree.o: tree.c
    $(CC) -$(CFLAGS) -c tree.c

proc-common.o: proc-common.c
    $(CC) -$(CFLAGS) -c proc-common.c

ask2-calculate.o: ask2-calculate.c
    $(CC) -$(CFLAGS) -c ask2-calculate.c

clean:
    rm ask2-calculate.o proc-common.o tree.o ask2-calculate
```

Δίνει output:

```
oslab36@os-nodel:~/exe2/ask21$ ./ask2-calculate myexpr.tree
*
+
    100
*
    +
        5
        +
            8
            4
        *
            3
            6
PID: 27633 (*): I was created succesfully...
PID: 27633 (*): Pipe [5, 6] to children created successfully
PID: 27633 (*): Ready to create child +...
PID: 27633 (*): Ready to create child *...
PID: 27634 (+): I was created succesfully...
PID: 27634 (+): Pipe [7, 8] to children created successfully
PID: 27634 (+): Ready to create child 100...
PID: 27635 (*): I was created succesfully...
PID: 27635 (*): Pipe [7, 8] to children created successfully
PID: 27635 (*): Ready to create child +...
PID: 27634 (+): Ready to create child ...
PID: 27635 (*): Ready to create child *...
PID: 27636 (100): I was created succesfully...
PID: 27637 (+): I was created succesfully...
PID: 27637 (+): Pipe [9, 10] to children created successfully
PID: 27637 (+): Ready to create child 5...
PID: 27636 (100): Wrote Value:100 on Pipe 8 and is Exiting...
PID: 27638 (): I was created succesfully...
PID: 27634 (+): Reading from pipe value no. 0 : 100
PID: 27639 (*): I was created succesfully...
PID: 27638 (): Wrote Value:0 on Pipe 8 and is Exiting...
PID: 27634 (+): Reading from pipe value no. 1 : 0
PID: 27637 (+): Ready to create child +...
PID: 27639 (*): Pipe [9, 10] to children created successfully
PID: 27634 (+): Wrote calculated result (100) to Pipe 6
PID: 27633 (*): Reading from pipe value no. 0 : 100
PID: 27639 (*): Ready to create child 3...
PID: 27640 (5): I was created succesfully...
PID: 27640 (5): Wrote Value:5 on Pipe 10 and is Exiting...
PID: 27637 (+): Reading from pipe value no. 0 : 5
PID: 27641 (+): I was created succesfully...
PID: 27641 (+): Pipe [11, 12] to children created successfully
PID: 27641 (+): Ready to create child 8...
PID: 27639 (*): Ready to create child 6...
PID: 27642 (3): I was created succesfully...
PID: 27641 (+): Ready to create child 4...
```



2. Σε ένα σύστημα πολλαπλών επεξεργαστών, μπορούν να εκτελούνται παραπάνω από μια διεργασίες παράλληλα. Σε ένα τέτοιο σύστημα, τι πλεονέκτημα μπορεί να έχει η αποτίμηση της έκφρασης από δέντρο διεργασιών, έναντι της αποτίμησης από μία μόνο διεργασία;

Σε ένα τέτοιο σύστημα μπορεί να εκτελεστεί ένα σύνολο ανεξάρτητων υπολογισμών – διεργασιών παράλληλα (κάθε υπολογισμός σε διαφορετικό πυρήνα) κι έτσι το αποτέλεσμα της συνολικής αριθμητικής έκφρασης να βρεθεί πιο γρήγορα σε σχέση με την περίπτωση που όλοι οι υπολογισμοί εκτελούνται από μια διεργασία.

### **1.5 Προαιρετικές Ερωτήσεις**

1. Έστω σύστημα με  $N$  επεξεργαστές. Ποιοι παράγοντες καθορίζουν αν ο παράλληλος υπολογισμός μια αριθμητικής έκφρασης όπως στην άσκηση θα είναι ταχύτερος από τον αντίστοιχο υπολογισμό σε μία μόνο διεργασία (σειριακή εκτέλεση).

Ο πιο συχνός λόγος που χρησιμοποιείται ο παράλληλος υπολογισμός είναι η απόδοσή του (performance) και η ταχύτητα ανταπόκρισής του (responsiveness). Υπάρχουν συγκεκριμένες συναρτήσεις που μπλοκάρουν την εκτέλεση ενός προγράμματος για ένα χρονικό διάστημα (π.χ. διάβασμα από αρχεία) οι οποίες μπορεί να μην καταναλώνουν CPU Power, αλλά συχνά καθυστερούν τη ροή του προγράμματος. Η χρήση threads σε αυτές τις περιπτώσεις είναι μια καλή πρακτική για την αντιμετώπιση αυτού του ζητήματος. Έτσι, αντί να χρησιμοποιούνται έλεγχοι για την είσοδο, έπειτα η ενσωμάτωσή τους στη ροή προγράμματος, η χειροκίνητη αλλαγή μεταξύ της τρέχουσας εισόδου και άλλων διεργασιών, ο προγραμματιστής μπορεί να επιλέξει να χρησιμοποιήσει threads και έτσι, να αφήσει το ένα thread να περιμένει το input και το άλλο να εκτελέσει π.χ. υπολογισμούς. Με άλλα λόγια, τα πολλαπλά threads επιτρέπουν την καλύτερη χρήση των διαφορετικών πόρων ενός υπολογιστή.

Εν κατακλείδι, στον πολυεπεξεργαστή ο πυρήνας αυτομάτως χρονοπρογραμματίζει καταλλήλως τα νήματα ή τις διεργασίες ώστε, αν είναι εφικτό, να εκτελούνται παράλληλα σε διαφορετικούς επεξεργαστές, ενώ στον σειριακό υπολογιστή όλα τα νήματα ή διεργασίες εκτελούνται σειριακά και ψευδοπαράλληλα στον ίδιο επεξεργαστή χωρίς να υπάρχει πραγματικός παραλληλισμός.