Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

2η Σειρά Ασκήσεων

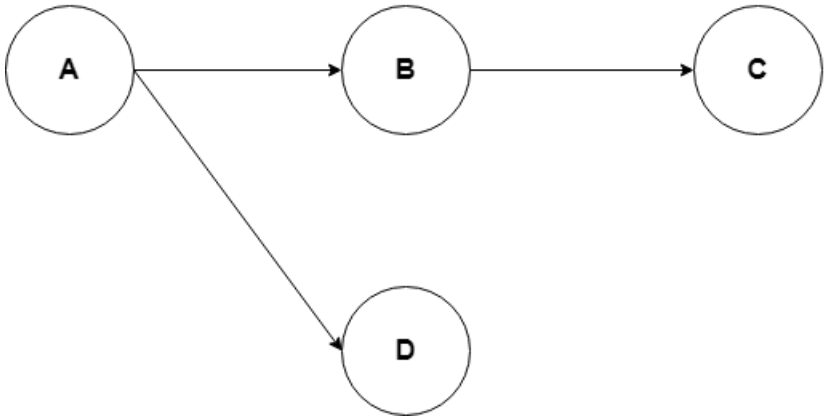
Μάθημα: Λειτουργικά Συστήματα (Τμήμα 1ο)

Εξάμηνο: 6ο

Ονοματεπώνυμα: Αλεξοπούλου Γεωργία (ΑΜ: 03120164), Γκενάκου Ζωή (ΑΜ: 03120015)

Άσκηση 2.1:

Καλούμαστε να σχεδιάσουμε έναν αλγόριθμο, ο οποίος κατασκευάζει το παρακάτω δέντρο διεργασιών:



Παρακάτω παρατίθεται ο αλγόριθμος αυτός. Η λειτουργία του εξηγείται στα σχόλια μέσα στον κώδικα:

```
#include <unistd.h>
#include <stdio.h>
```

```c
#include <stdlib.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) {
    pid_t pidB, pidC, pidD, killed_child_pid; //creating a pid
    int status;                               //variable for each node

    change_pname("A"); //change the name of the current process
                       //change_pname() is defined in proc-common.h
    printf("A: Starting...\n");
    pidB = fork();   //fork for child B
    if (pidB < 0) { //if fork() fails, return exit code 1
        perror("fork");
        exit(1);
    }
    else if (pidB == 0) { //if fork() succeeds, the current process
is a child process and its behavior is defined by the if and elif
processes
        // I'm B
        change_pname("B");
        printf("B: Starting...\n");
        pidD = fork();
        if (pidD < 0) {
            perror("fork");
            exit(1);
        }
        else if (pidD == 0) { //if fork() succeeds the current
process is the parent process
```

```
            // I'm D
            change_pname("D");
            printf("D: Starting...\n");
            printf("D: Sleeping...\n");//process D sleeps for a
designated amount of time
            sleep(SLEEP_PROC_SEC);
            printf("D: Exiting...\n");
            exit(13);//exits with exit code 13
        }
        printf("B: Waiting...\n");
        killed_child_pid = wait(&status); //process B waits for its
child process D to terminate
        explain_wait_status(killed_child_pid, status);//once D
terminates, B prints a message explaining the exit status of D
//explain_wait_status() is defined in proc-common.h
        printf("B: Exiting...\n");
        exit(19);//exits with exit code 19
    }

    pidC = fork();
    if (pidC < 0) {
        perror("fork");
        exit(1);
    } else if (pidC == 0) {
        // I'm C
        change_pname("C");
        printf("C: Starting...\n");
        printf("C: Sleeping...\n");//process C sleeps for a
designated amount of time
        sleep(SLEEP_PROC_SEC);
        printf("C: Exiting...\n");
        exit(17);//exits with exit code 17
    }

    printf("A: Waiting...\n");
    int i;
    for (i = 0; i < 2; i++) {
        killed_child_pid = wait(&status);//process A waits for its
children B and C to terminate
```

```
        explain_wait_status(killed_child_pid, status);
    }
    printf("A: Exiting...\n");//both children have terminated, exit
status 0
}

/*
 * The initial process (our program) forks the root of the process
tree (A),
 * waits for the process tree to be completely created (with
sleep()),
 * then takes a photo of it using show_pstree() and waits for A to
die.
 *
 * How to wait for the process tree to be ready?
 *  -> sleep for a few seconds and hope for the best.
 *
 * Also, the children leaf processes stay active (they sleep()) for a
few seconds
 * (longer than the sleep time of the initial process), so the user
can see the whole
 * process tree created.
 *
 * The parent processes wait until all of their children die.
*/
int main(void) {
    pid_t pid;
    int status;

    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    else if (pid == 0) {
        // I'm the child
        fork_procs();
        exit(0);
    }
```

```
    // I'm the parent
    sleep(SLEEP_TREE_SEC);
    show_pstree(pid);

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

Παρακάτω παραθέτουμε και τον source-code για το Makefile:

```
make-tree: make-tree.o proc-common.o
	gcc -o make-tree make-tree.o proc-common.o
make-tree.o: make-tree.c proc-common.h
	gcc -Wall -c make-tree.c
clean:
	rm -f make-tree.o make-tree
```

Εκτελώντας το Makefile και καλώντας τη συνάρτηση make-tree, λαμβάνουμε το παρακάτω μήνυμα εξόδου:

```
oslab33@orion:/store/homes/oslab/oslab33/ex2/2.1$ make
gcc -Wall -c make-tree.c
gcc -o make-tree make-tree.o proc-common.o
oslab33@orion:/store/homes/oslab/oslab33/ex2/2.1$ ls
ask2-fork.c  make-tree    make-tree.o    proc-common.h
Makefile     make-tree.c  proc-common.c  proc-common.o
oslab33@orion:/store/homes/oslab/oslab33/ex2/2.1$ ./make-tree
A: Starting...
A: Waiting...
B: Starting...
B: Waiting...
C: Starting...
C: Sleeping...
D: Starting...
D: Sleeping...


A(17593)─┬─B(17594)──D(17596)
         └─C(17595)


C: Exiting...
My PID = 17593: Child PID = 17595 terminated normally, exit status = 17
D: Exiting...
My PID = 17594: Child PID = 17596 terminated normally, exit status = 13
B: Exiting...
My PID = 17593: Child PID = 17594 terminated normally, exit status = 19
A: Exiting...
My PID = 17592: Child PID = 17593 terminated normally, exit status = 0
```

Ερωτήσεις:

1. ***Τι θα γίνει αν τερματίσετε πρόωρα τη διεργασία A, δίνοντας kill -KILL <pid>, όπου <pid> το Process ID της;***
   Αν τερματίσουμε πρόωρα τη διεργασία A, δίνοντας την εντολή *'kill -KILL <pid>'* , ολόκληρο το δέντρο διεργασιών με ρίζα τη διεργασία A θα τερματίσει επίσης. Αυτό συμβαίνει γιατί η εντολή *'kill -KILL'* στέλνει στο <pid> της διεργασίας ένα σήμα SIGKILL, το οποίο συνεπάγεται τον άμεσο τερματισμό της. Οι διεργασίες-παιδιά του δέντρου διεργασιών (B, C, D) λαμβάνουν ένα σήμα SIGTERM, το οποίο ειδοποιεί πως η διεργασία-πατέρας (A) έχει τερματιστεί, επομένως κι αυτές τερματίζονται άμεσα. Αν κάποια από αυτές τις διεργασίες-παιδιά έχουν δημιουργήσει δικές τους διεργασίες-παιδιά, αυτές θα μείνουν "ορφανές", και η αρχική διεργασία θα γίνει η νέα διεργασία-πατέρας τους.

```
oslab33@orion:/store/homes/oslab/oslab33/ex2/2.1$ ./make-tree
A: Starting...
A: Waiting...
B: Starting...
B: Waiting...
C: Starting...
C: Sleeping...
D: Starting...
D: Sleeping...


A(18025)─┬─B(18026)──D(18028)
         └─C(18027)


My PID = 18024: Child PID = 18025 was terminated by a signal, signo = 9
oslab33@orion:/store/homes/oslab/oslab33/ex2/2.1$ C: Exiting...
D: Exiting...
My PID = 18026: Child PID = 18028 terminated normally, exit status = 13
B: Exiting...
```

2. ***Τι θα γίνει αν κάνετε show_pstree(getpid()) αντί για show_pstree(pid) στη main(); Ποιες επιπλέον διεργασίες φαίνονται στο δέντρο και γιατί;***
   Αλλάζοντας την εντολή *'show_pstree(pid)'* σε *'show_pstree(getpid())'* στη συνάρτηση main(), τότε η εντολή *'pstree'* εκτυπώνει το δέντρο διεργασιών της κύριας διεργασίας, δηλαδή της διεργασίας A. Επομένως, το terminal θα δείξει το δέντρο διεργασιών μέχρι τη διεργασία A, αλλά δεν θα δείξει τα υπο-δέντρα με πατέρα τις διεργασίες B και C, καθώς είναι είναι παιδιά της διεργασίας A και όχι της διεργασίας-πατέρα που καλεί η *'show_pstree(getpid())'* . Επομένως, το output θα περιλαμβάνει ένα δέντρο διεργασιών με μια μόνο διεργασία (την κύρια διεργασία) με pid = 1 και τη διεργασία παιδί του (A) με τις διεργασίες παιδιά της (B και C). Τα υπο-δέντρα των κόμβων B και C δεν περιλαμβάνονται στην έξοδο. Παρακάτω φαίνεται το αποτέλεσμα της εκτέλεσης του τροποποιημένου πηγαίου κώδικα:

```
oslab33@orion:/store/homes/oslab/oslab33/ex2/2.1$ ./make-tree
A: Starting...
A: Waiting...
B: Starting...
B: Waiting...
C: Starting...
C: Sleeping...
D: Starting...
D: Sleeping...


make-tree(18839)─┬─A(18840)─┬─B(18841)──D(18843)
                 │          └─C(18842)
                 └─sh(18852)──pstree(18853)


C: Exiting...
My PID = 18840: Child PID = 18842 terminated normally, exit status = 17
D: Exiting...
My PID = 18841: Child PID = 18843 terminated normally, exit status = 13
B: Exiting...
My PID = 18840: Child PID = 18841 terminated normally, exit status = 19
A: Exiting...
My PID = 18839: Child PID = 18840 terminated normally, exit status = 0
```

3. ***Σε υπολογιστικά συστήματα πολλαπλών χρηστών, πολλές φορές ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης. Γιατί;***
   Υπάρχουν αρκετοί λόγοι για τους οποίους ένας διαχειριστής μπορεί να θέλει να περιορίσει τον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης σε ένα υπολογιστικό σύστημα πολλών χρηστών. Πιο συγκεκριμένα, κάθε διεργασία χρησιμοποιεί πόρους συστήματος (μνήμη, χρόνος CPU, I/O). Αν ένας χρήστης είχε τη δυνατότητα να δημιουργήσει άπειρο αριθμό διεργασιών, τότε θα υπήρχε το ενδεχόμενο να χρησιμοποιήσει όλους τους πόρους του συστήματος, επιβαρύνοντας την απόδοση και ενισχύοντας τα σφάλματα του συστήματος. Με την επιβολή του περιορισμού στον χρήστη, οι πόροι του συστήματος κατανέμονται ισόποσα μεταξύ των χρηστών. Ως προέκταση, ο περιορισμός αυτός μπορεί να βοηθήσει στη διατήρηση της σταθερότητας του συστήματος, αποτρέποντας εκκρεμείς διεργασίες που καταναλώνουν όλους τους διαθέσιμους πόρους και διακόπτουν τη λειτουργία του συστήματος. Θέτοντας όρια στη δημιουργία διεργασιών, ο διαχειριστής μπορεί να διασφαλίσει ότι το σύστημα παραμένει σταθερό και διαθέσιμο σε όλους τους χρήστες.

Άσκηση 2.2:

Σε αυτό το βήμα της άσκησης καλούμαστε να τροποποιήσουμε τον παραπάνω κώδικα έτσι ώστε να κατασκευάζει το δέντρο διεργασιών ενός οποιουδήποτε αρχείου εισόδου. Ο κώδικας φαίνεται

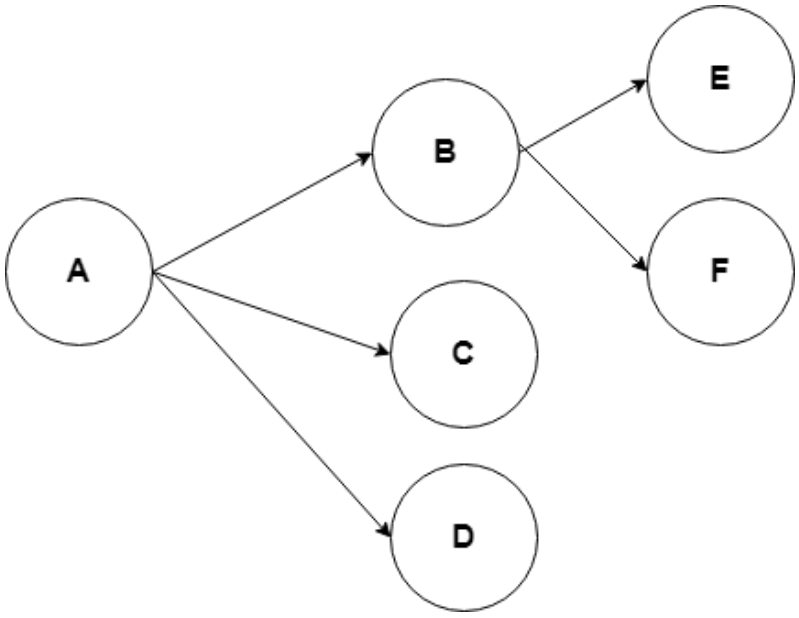
παρακάτω (η λειτουργία του εξηγείται με σχόλια):

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

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

#define LEAF_EXIT_STATUS 13
#define NON_LEAF_EXIT_STATUS 7
#define SLEEP_CHILD_SEC 10
#define SLEEP_TREE_SEC 3

void fork_procs(struct tree_node *parent) {
    pid_t pidChild;
    int i, wstatus;

    change_pname(parent->name); //change the name of the process to
the name of the node in the tree
    printf("%s: Starting...\n", parent->name);//process has started

    if (parent->nr_children == 0) { //if the node has no children,
sleep for a designated amount of time and exit
        printf("%s: Sleeping...\n", parent->name);
        sleep(SLEEP_CHILD_SEC);
        printf("%s: Exiting...\n", parent->name);
        exit(LEAF_EXIT_STATUS);
    }
    else { //if the node has children, create a new child process for
each child node in the tree
        for (i = 0; i < parent->nr_children; i++) {
            pidChild = fork();
            if (pidChild < 0) { //if the fork failed, exit code 1
                perror("fork");
                exit(1);
            }
```

```
            else if (pidChild == 0) { //if the fork was successful,
call fork_procs() using the child node as an argument
                fork_procs(parent->children + i);
            }
        }

        printf("%s: Waiting...\n", parent->name);
        for (i = 0; i < parent->nr_children; i++) { //wait for all
child processes to terminate
            pidChild = wait(&wstatus);
            if (pidChild == -1) { //if the wait fails, exit code 1
                perror("wait");
                exit(1);
            }
            explain_wait_status(pidChild, wstatus); //exit status of
each child process
        }

        printf("%s: Exiting...\n", parent->name);
        exit(NON_LEAF_EXIT_STATUS);
    }
}

int main(int argc, char **argv) //read input file name {
    pid_t pid_root, pid_wait;
    int wstatus;
    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); //print tree

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

```
        perror("main: fork");
        exit(1);
    }
    else if (pid_root == 0) {
        fork_procs(root);
    }

    /* Wait for the process tree to be created */
    sleep(SLEEP_TREE_SEC);

    /* Print the process tree with root_pid */
    show_pstree(pid_root);

    /* Wait for the root of the process tree to terminate */
    printf("Initial process: Waiting...\n");
    pid_wait = wait(&wstatus);
    if (pid_wait == -1) { //if the wait fails, exit code 1
        perror("main: wait");
        exit(1);
    }
    explain_wait_status(pid_wait, wstatus);

    printf("Initial process: All done, exiting...\n");

    return 0;
}
```

Εκτελώντας το Makefile και καλώντας τη συνάρτηση make-tree με input file το proc.tree,



λαμβάνουμε το παρακάτω μήνυμα εξόδου:

```
oslab33@orion:/store/homes/oslab/oslab33/ex2/2.2$ ./make-tree proc.tree
A
        B
                E
                F
        C
        D
A: Starting...
A: Waiting...
D: Starting...
D: Sleeping...
B: Starting...
B: Waiting...
C: Starting...
C: Sleeping...
F: Starting...
F: Sleeping...
E: Starting...
E: Sleeping...


A(21263)─┬─B(21264)─┬─E(21267)
         │          └─F(21268)
         ├─C(21265)
         └─D(21266)


Initial process: Waiting...
D: Exiting...
My PID = 21263: Child PID = 21266 terminated normally, exit status = 13
C: Exiting...
My PID = 21263: Child PID = 21265 terminated normally, exit status = 13
F: Exiting...
My PID = 21264: Child PID = 21268 terminated normally, exit status = 13
E: Exiting...
My PID = 21264: Child PID = 21267 terminated normally, exit status = 13
B: Exiting...
My PID = 21263: Child PID = 21264 terminated normally, exit status = 7
A: Exiting...
My PID = 21262: Child PID = 21263 terminated normally, exit status = 7
Initial process: All done, exiting...
```

Ερώτηση: Με ποια σειρά εμφανίζονται τα μηνύματα έναρξης και τερματισμού των διεργασιών; Γιατί;

Η σειρά με την οποία εμφανίζονται τα μηνύματα έναρξης και τερματισμού των διεργασιών εξαρτάται από τη σειρά με την οποία δημιουργούνται και τερματίζουν οι διεργασίες αυτές. Όταν μια διεργασία δημιουργείται, πρώτα εκτυπώνει το μήνυμα έναρξης (“Starting…”) κι έπειτα

εκτελεί τη διαδικασία fork() για τις διεργασίες-παιδιά της. Οι διεργασίες-παιδιά μπορούν να τυπώσουν τα δικά τους μηνύματα έναρξης πριν εκτελέσουν το κομμάτι του κώδικά τους. Όταν φτάνουμε σε μια διεργασία-φύλλο, κοιμάται για μια προκαθορισμένη περίοδο χρόνου, εκτυπώνει το μήνυμα τερματισμού του και πραγματοποιεί έξοδο με LEAF_EXIT_STATUS. Όταν μια διεργασία μη-φύλλο δημιουργήσει όλα της τα παιδιά, περιμένει αυτά να τερματίσουν καλώντας την εντολή wait(). Όταν όλα τα παιδιά έχουν τερματίσει, ο γονέας εκτυπώνει το μήνυμα τερματισμού και πραγματοποιεί έξοδο με NON_LEAF_EXIT_STATUS. Τέλος, η κύρια διεργασία περιμένει τη διεργασία-πατέρα του δέντρου να τερματίσει κι έπειτα εκτυπώνει μήνυμα τερματισμού και πραγματοποιεί έξοδο. Συνεπώς, η σειρά με την οποία τυπώνονται τα μηνύματα έναρξης και τερματισμού των διεργασιών εξαρτάται από τη δομή του δέντρου και το fork() syscall.

Άσκηση 2.3:

Αυτός ο κώδικας διαβάζεται σε μια δενδρική δομή από ένα αρχείο εισόδου, διαχωρίζει τις διεργασίες-παιδιά αναδρομικά για κάθε κόμβο στο δέντρο και εκτυπώνει μια απεικόνιση του δέντρου διεργασιών στην κονσόλα. Στη συνέχεια, ξυπνά τη ρίζα του δέντρου διεργασίας, περιμένει να τερματιστεί και εκτυπώνει την κατάσταση εξόδου του. Η συνάρτηση fork_procs είναι υπεύθυνη για τη διχοτόμηση των διεργασίες-παιδιά και εκτυπώνει επίσης ένα μήνυμα στην κονσόλα που υποδεικνύει ότι μια διεργασία ξεκινά. Η λειτουργία του κώδικα φαίνεται παράλληλα και στα επεξηγηματικά σχόλια.

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

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

/* This function forks the child processes recursively for each node
in the tree */

void fork_procs(struct tree_node *parent) {
   int i, wstatus;
   change_pname(parent->name);    // Changes the process name to the
name of the parent node
   printf("%s: Starting...\n", parent->name);  // Prints a message
indicating that the parent node is starting
```

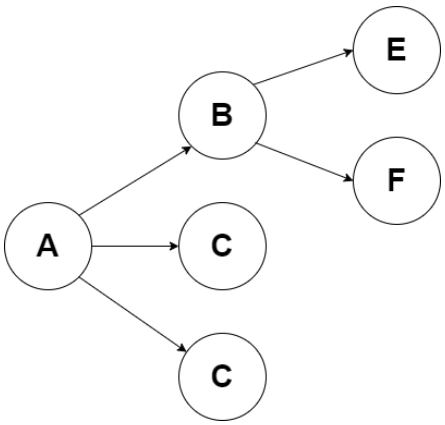
```
   if (parent->nr_children == 0) {
       if (raise(SIGSTOP) != 0) {  // Stops the process and waits for
a signal to be sent to it
           perror("raise");  // Prints an error message if there was
an error with the raise() call
           exit(1);  // Exits the process with an error code
       }
       printf("%s is awake!\n", parent->name);  // Prints a message
indicating that the parent node has woken up
       exit(13);  // Exits the process with a specific exit code
   }
   else {
       pid_t pid[parent->nr_children];
       for (i = 0; i < parent->nr_children; i++) {
           pid[i] = fork();  // Forks the process to create a child
process
           if (pid[i] < 0) {
               perror("fork");  // Prints an error message if there
was an error with the fork() call
               exit(1);  // Exits the process with an error code
           }
           else if (pid[i] == 0) {
               fork_procs(parent->children+i);  // Recursively calls
fork_procs() to fork the child processes for the current node's
children
               exit(0);  // Exits the child process with a success
exit code
           }
           else {
               change_pname(parent->name);  // Changes the process
name to the name of the parent node
           }
       }
       wait_for_ready_children(parent->nr_children);  // Waits for
all child processes to be ready
       if (raise(SIGSTOP) != 0) {  // Stops the process and waits for
a signal to be sent to it
           perror("raise");  // Prints an error message if there was
```

```
an error with the raise() call
            exit(1);  // Exits the process with an error code
        }
        printf("%s is awake!\n", parent->name);  // Prints a message
indicating that the parent node has woken up
        for (i = 0; i < parent->nr_children; i++) {
            if (kill(pid[i], SIGCONT) < 0) {  // Sends a signal to the
child process to wake it up
                perror("kill");  // Prints an error message if there
was an error with the kill() call
                exit(1);  // Exits the process with an error code
            }
            else if (wait(&wstatus) == -1) {  // Waits for the child
process to exit
                perror("wait");  // Prints an error message if there
was an error with wait() function
                exit(1); // Exits the program with a failure status
            }
            else {
                explain_wait_status(pid[i], wstatus); // Prints
information about the status of the child process
            }
        }
        exit(7); // Exits the child process with status 7
    }
}

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

    if (argc != 2) {  // Checks if the program was called with the
correct number of arguments
        fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
// Prints an error message to stderr if the program was not called
correctly
        exit(1);  // Exits the program with a failure status
    }
```

```c
    root = get_tree_from_file(argv[1]);  // Read in the tree structure from the input file
    print_tree(root);  // Print the tree structure to the console

/* Fork root of process tree */
    pid_root = fork();  // Fork the initial process to create the root of the process tree
    if (pid_root < 0) {
        perror("main: fork");  // Prints an error message to stderr if the fork() function fails
        exit(1);  // Exits the program with a failure status
    }
    else if (pid_root == 0) {  // If this is the child process
        fork_procs(root);  // Fork the child processes for the tree
        exit(1);  // Exits the child process with a failure status
    }

/* Father (initial process) */
    wait_for_ready_children(1);  // Wait for the root of the process tree to be ready

/* Print the process tree with root_pid */
    show_pstree(pid_root);  // Print a visualization of the process tree to the console

/* Wake up the root of the process tree */
    if (kill(pid_root, SIGCONT) < 0) {
        perror("main: kill");  // Prints an error message to stderr if the kill() function fails
        exit(1);  // Exits the program with a failure status
    }

/* Wait for the root of the process tree to terminate */
    pid_wait = wait(&wstatus);  // Waits for the root of the process tree to terminate
    if (pid_wait == -1) {
        perror("main: wait");  // Prints an error message to stderr if the wait() function fails
```

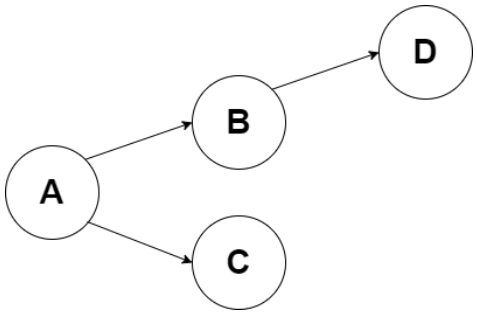
```
        exit(1);  // Exits the program with a failure status
    }

/* Print the exit status of the root process */
    explain_wait_status(pid_wait, wstatus);  // Prints information
about the status of the root process

    return 0;  // Exits the program with a success status

}
```

Προκειμένου να επιβεβαιώσουμε την λειτουργία του προγράμματος, μπορούμε να τρέξουμε το εκτλέσιμο make-tree-sig με διάφορα αρχεία εισόδου. Παρατηρούμε ότι τα μηνύματα ενεργοποίησης (… is awake!), καθώς και τα διαγνωστικά μηνύματα τερματισμού εμφανίζονται με DF τρόπο, όπως θέλαμε, ενώ η σειρά που "μπλέκονται" είναι και αυτή προκαθορισμένη. Συγκεκριμένα, για τα μηνύματα ενεργοποίησης έχουμε DF preorder traversal και για τα διαγνωστικά μηνύματα τερματισμού έχουμε DF postorder traversal. Για τα μηνύματα που εμφανίζονται πάνω από την εμφάνιση του δέντρου με την show_pstree() (μηνύματα έναρξης και διαγνωστικά μηνύματα παύσης), ισχύει ότι έχουμε αναφέρει και παραπάνω στην άσκηση 2.2 (δεν είναι ντετερμινιστική η σειρά και εξαρτάται από τον scheduler – τα μηνύματα έναρξης εμφανίζονται μάλλον πάντα με BF traversal).



proc.tree

```
oslab33@orion:~/ex2/2.3$ ./make-tree-sig proc.tree
A
        B
                E
                F
        C
        D
A: Starting...
D: Starting...
My PID = 20501: Child PID = 20504 has been stopped by a signal, signo = 19
B: Starting...
C: Starting...
My PID = 20501: Child PID = 20503 has been stopped by a signal, signo = 19
F: Starting...
My PID = 20502: Child PID = 20506 has been stopped by a signal, signo = 19
E: Starting...
My PID = 20502: Child PID = 20505 has been stopped by a signal, signo = 19
My PID = 20501: Child PID = 20502 has been stopped by a signal, signo = 19
My PID = 20500: Child PID = 20501 has been stopped by a signal, signo = 19


A(20501)─┬─B(20502)─┬─E(20505)
         │          └─F(20506)
         ├─C(20503)
         └─D(20504)


A is awake!
B is awake!
E is awake!
My PID = 20502: Child PID = 20505 terminated normally, exit status = 13
F is awake!
My PID = 20502: Child PID = 20506 terminated normally, exit status = 13
My PID = 20501: Child PID = 20502 terminated normally, exit status = 7
C is awake!
My PID = 20501: Child PID = 20503 terminated normally, exit status = 13
D is awake!
My PID = 20501: Child PID = 20504 terminated normally, exit status = 13
My PID = 20500: Child PID = 20501 terminated normally, exit status = 7
oslab33@orion:~/ex2/2.3$
```



test-ex1.tree

```
oslab33@orion:~/ex2/2.3$ ls
bad.tree       make-tree-sig.c  proc-common.h  test-ex1.tree  tree.o
Makefile       make-tree-sig.o  proc-common.o  tree.c
make-tree-sig  proc-common.c    proc.tree      tree.h
oslab33@orion:~/ex2/2.3$ ./make-tree-sig test-ex1.tree
A
        B
                D
        C
A: Starting...
C: Starting...
My PID = 21121: Child PID = 21123 has been stopped by a signal, signo = 19
B: Starting...
D: Starting...
My PID = 21122: Child PID = 21124 has been stopped by a signal, signo = 19
My PID = 21121: Child PID = 21122 has been stopped by a signal, signo = 19
My PID = 21120: Child PID = 21121 has been stopped by a signal, signo = 19


A(21121)─┬─B(21122)──D(21124)
         └─C(21123)


A is awake!
B is awake!
D is awake!
My PID = 21122: Child PID = 21124 terminated normally, exit status = 13
My PID = 21121: Child PID = 21122 terminated normally, exit status = 7
C is awake!
My PID = 21121: Child PID = 21123 terminated normally, exit status = 13
My PID = 21120: Child PID = 21121 terminated normally, exit status = 7
oslab33@orion:~/ex2/2.3$
```

Όχι καλά δομημένα αρχεία εισόδου και λάθος αριθμός ορισμάτων αντίστοιχα:

```
oslab33@orion:~/ex2/2.3$ ./make-tree-sig bad.tree
expecting: D and got EOF
oslab33@orion:~/ex2/2.3$ ./make-tree-sig bad.tree proc.tree
Usage: ./make-tree-sig <input_tree_file>
```

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

Η χρήση σημάτων για συγχρονισμό μεταξύ διεργασιών έχει πολλά πλεονεκτήματα σε σχέση με τη χρήση του sleep():

- Ταχύτερη απόκριση: Όταν χρησιμοποιούνται σήματα, η διαδικασία λήψης ειδοποιείται αμέσως όταν αποστέλλεται το σήμα, ενώ με την sleep() η διεργασία πρέπει να περιμένει για το καθορισμένο χρονικό διάστημα πριν συνεχιστεί η εκτέλεση.
- Ακριβέστερος χρονισμός: Με τα σήματα, η διαδικασία αποστολής μπορεί να καθορίσει ακριβώς πότε θα σταλεί το σήμα, ενώ με τη λειτουργία sleep() ο χρονισμός μπορεί να

επηρεαστεί από άλλους παράγοντες όπως το φόρτο του συστήματος ή οι καθυστερήσεις προγραμματισμού.

- Ευελιξία: Τα σήματα μπορούν να χρησιμοποιηθούν για διάφορους σκοπούς πέρα από το συγχρονισμό, όπως ο χειρισμός interrupts, η αναφορά σφαλμάτων ή οι ειδοποιήσεις που καθορίζονται από τον χρήστη. Το sleep() περιορίζεται μόνο σε σκοπούς συγχρονισμού.
- Ασύγχρονη λειτουργία: Τα σήματα επιτρέπουν στις διεργασίες να επικοινωνούν ασύγχρονα, πράγμα που σημαίνει ότι ο αποστολέας και ο παραλήπτης μπορούν να λειτουργούν ανεξάρτητα και με τον δικό τους ρυθμό. Η sleep() απαιτεί από τις διεργασίες να περιμένουν ένα σταθερό χρονικό διάστημα, το οποίο μπορεί να περιορίσει την ευελιξία και την ανταπόκρισή τους.

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

Ο κώδικας της συνάρτησης wait_for_ready_children() φαίνεται παρακάτω:

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

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

Η συνάρτηση wait_for_ready_children() περιμένει να ετοιμαστεί ένας συγκεκριμένος αριθμός διεργασιών-παιδιά περιμένοντας να αλλάξει η κατάστασή της οποιαδήποτε διεργασίας-παιδί και,

στη συνέχεια, ελέγχοντας εάν έχει σταματήσει. Εάν ένα παιδί δεν έχει σταματήσει, η λειτουργία θα εκτυπώσει ένα μήνυμα σφάλματος και θα βγει. Η χρήση αυτής της συνάρτησης διασφαλίζει ότι η γονική διαδικασία δεν θα συνεχιστεί έως ότου όλες οι διεργασίες-παιδιά είναι έτοιμες να προχωρήσουν.

Εάν η wait_for_ready_children() παραλειφθεί, η γονική διαδικασία μπορεί να προχωρήσει πριν να είναι έτοιμες όλες οι διεργασίες-παιδιά, κάτι που μπορεί να προκαλέσει προβλήματα συγχρονισμού και ενδεχομένως να οδηγήσει σε απροσδόκητη συμπεριφορά. Επιπλέον, η γονική διαδικασία μπορεί να προσπαθήσει να αφυπνίσει μια διεργασία-παιδί που δεν είναι ακόμη έτοιμη, κάτι που μπορεί να προκαλέσει την αποτυχία της διαδικασίας.

Επιπλέον, εξασφαλίζουμε έναν σχετικά αξιόπιστο μηχανισμό επικοινωνίας μεταξύ των διεργασιών, χωρίς ταυτόχρονη λήψη σημάτων, race conditions και χωρίς την αποστολή σημάτων σε διεργασίες που πιθανώς δεν έχουν καν δημιουργηθεί ακόμα. Π.χ. ένα πιθανό ενδεχόμενο αν δεν χρησιμοποιήσουμε την wait_for_ready_children() είναι: η διεργασία πατέρας στέλνει το σήμα SIGCONT στην διεργασία παιδί, η οποία αν και έχει δημιουργηθεί δεν έχει κάνει παύση με raise(SIGSTOP) ακόμα. Σε αυτήν την περίπτωση η διεργασία παιδί θα παραμείνει για πάντα “μπλοκαρισμένη” μετά την παύση της.

Άσκηση 2.4:

Διαμορφώνουμε πάλι τον κώδικα την 2.2, ώστε να υπολογίζει δέντρα που αναπαριστούν αριθμητικές εκφράσεις, μέσω των σωληνώσεων (pipes).

Η λειτουργία του κώδικα, εξηγείται παραπάνω στο σχόλια του κώδικα. Επιπλέον αντί για sleep(), που χρησιμοποιούσαμε στο 2.2, υλοποιήσαμε την άσκηση χρησιμοποιώντας σήματα για να μπορεί ο χρήστης να δει το πλήρες δέντρο με την show_pstree() .

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

// Required headers for the project
#include "proc-common.h"
#include "tree.h"
```

```c
#define SLEEP_PROC_SEC 1

// Recursive function that spawns child processes and computes the values
void fork_procs(struct tree_node *parent, int fdw){
    pid_t child_pid;
    int wstatus, i;

    // Changes the name of the process to the name of the current node
    change_pname(parent->name);

    // Prints a message indicating that the current node is starting
    printf("%s: Starting...\n", parent->name);

    if(parent->nr_children==0){ // If the node has no children, it's a leaf and its value is written to the pipe
        int value=atoi(parent->name);  // Converts the string value of the leaf node to an integer
        if(write(fdw, &value, sizeof(value))!=sizeof(value)){  // Writes the value to the pipe and checks for errors
            perror("write to pipe");
            exit(1);
        }
        close(fdw);  // Closes the write end of the pipe
        sleep(SLEEP_PROC_SEC);  // Sleeps for a while
        exit(13);  // Exits with status code 13
    }
    else {  // If the node has children, it's an operator and its children are recursively evaluated
        int pipefd[2];
        if(pipe(pipefd)==-1){  // Creates a pipe for inter-process communication and checks for errors
            perror("pipe");
            exit(1);
        }
        for(i=0; i<parent->nr_children; i++){  // Spawns a child process for each child of the current node
            child_pid=fork();  // Forks a new process and stores the
```

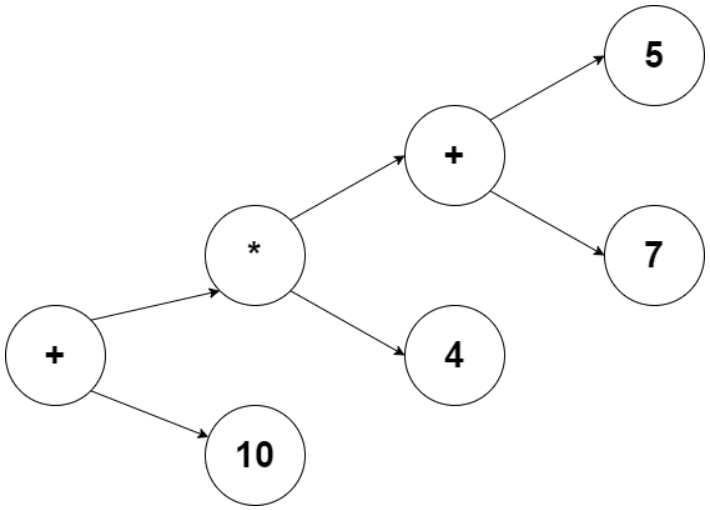
```
returned PID in child_pid
            if(child_pid<0) {  // If forking fails, prints an error
message and exits
                perror("fork");
                exit(1);
            }
            if(child_pid==0) {  // If child_pid is 0, this is the
child process and it evaluates the i-th child
                close(pipefd[0]);  // Closes the read end of the pipe
                fork_procs(parent->children+i, pipefd[1]);  //
Recursively evaluates the i-th child
            }
        }
        close(pipefd[1]);  // Closes the write end of the pipe in the
parent process
        int value[2];
        for(i=0; i<parent->nr_children; i++){  // Reads the values
computed by the children from the pipe
            if(read(pipefd[0], &value[i],
sizeof(value[i]))!=sizeof(value[i])) {
           perror("read from pipe");
           exit(1);
         }
      }
      close(pipefd[0]);
      // Computes the result of the operation according to the
operator in the parent node
      int result;
      if(!strcmp(parent->name, "+")){
           result = value[0]+value[1];
      }
      if(!strcmp(parent->name, "*")){
           result=value[0]*value[1];
      }

      // Prints the value computed by the current process
      printf("Me: %ld, i have computed: %i %s %i = %i\n",
(long)getpid(), value[0], parent->name, value[1], result);
```

```
      // Writes the result to the pipe
      if(write(fdw, &result, sizeof(result))!=sizeof(result)){
            perror("write to pipe");
            exit(1);
      }
      close(fdw);

      // Waits for the termination of all the children and prints
their exit status
      for(i=0; i<parent->nr_children; i++){
            if(wait(&wstatus)==-1){
                        perror("wait");
                        exit(1);
            }
            explain_wait_status(child_pid, wstatus);
      }

      // Exits the process with exit status 7
      exit(7);
    }
}
int main(int argc, char **argv){
    int wstatus, result;
    struct tree_node *root;
    int pipefd[2];
    pid_t pid_root;


    // Checks if the command is entered correctly, and prints usage
message if it is not
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
// Prints usage message if the command is not entered correctly
        exit(1);
    }

    // Reads the expression tree from the input file and prints it
    root = get_tree_from_file(argv[1]);
    print_tree(root);  // Prints the expression tree read from the
```

```
input file


    // Creates a pipe for inter-process communication
    if(pipe(pipefd)==-1) {
        perror("main: pipe");  // Prints an error message if pipe
creation fails
        exit(1);
    }


    // Forks a process to evaluate the expression tree and creates a
process tree
    pid_root=fork();
    if(pid_root<0) {
        perror("main: fork");  // Prints an error message if forking
fails
        exit(1);
    }

    if(pid_root==0){
        close(pipefd[0]);
        fork_procs(root, pipefd[1]);  // Starts the recursion to
evaluate the expression tree
    }
    close(pipefd[1]);


    // Reads the result of the evaluation from the pipe and prints it
    if(read(pipefd[0], &result, sizeof(result))!=sizeof(result)){
        perror("main: read from pipe");  // Prints an error message
if read from pipe fails
        exit(1);
    }

    close(pipefd[0]);

    show_pstree(pid_root);  // Shows the process tree
```

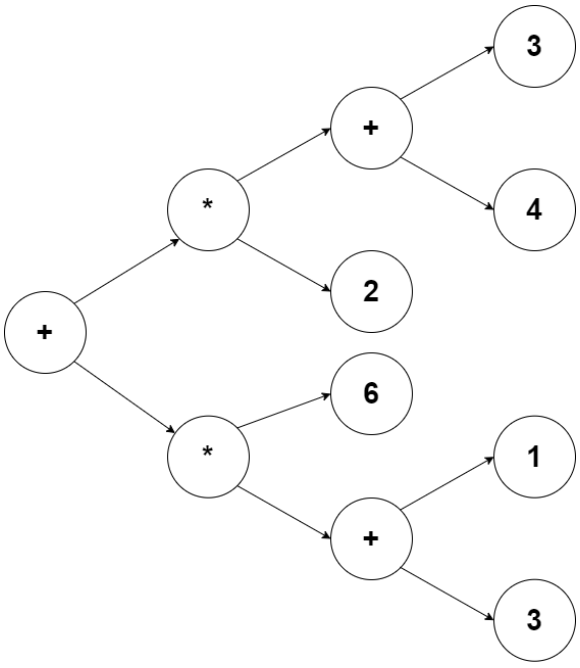
```
    // Waits for the termination of the root process and prints its
exit status
    if(wait(&wstatus)==-1){
        perror("main: wait");  // Prints an error message if the wait
for the process fails
        exit(1);
    }

    explain_wait_status(pid_root, wstatus);  // Explains the status
of the root process that has been waited for

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

    return 0;


}
```

Πάλι χρησιμοποιούμε διάφορα αρχεία εισόδου για να επαληθεύσουμε ότι η λειτουργία του προγράμματός μας είναι σωστή:



expr.tree

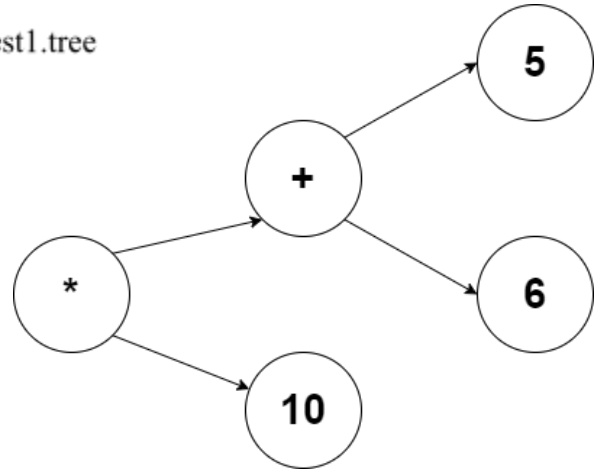
```
oslab33@orion:~/ex2/2.4$ ls
bad.tree   make-expr-tree    make-expr-tree.o  proc-common.c  proc-common.o  test1.tree  tree.c  tree.o
expr.tree  make-expr-tree.c  Makefile          proc-common.h  proc.tree      test2.tree  tree.h
oslab33@orion:~/ex2/2.4$ ./make-expr-tree expr.tree
+
        10
        *
                +
                        5
                        7
                4
+: Starting...
10: Starting...
*: Starting...
+: Starting...
4: Starting...
7: Starting...
5: Starting...
Me: 23566, i have computed: 7 + 5 = 12
Me: 23565, i have computed: 4 * 12 = 48
Me: 23563, i have computed: 10 + 48 = 58


+(23563)─┬─*(23565)─┬─+(23566)─┬─5(23568)
         │          │          └─7(23569)
         │          └─4(23567)
         └─10(23564)


My PID = 23563: Child PID = 23564 terminated normally, exit status = 13
My PID = 23566: Child PID = 23568 terminated normally, exit status = 13
My PID = 23566: Child PID = 23569 terminated normally, exit status = 13
My PID = 23565: Child PID = 23566 terminated normally, exit status = 7
My PID = 23565: Child PID = 23567 terminated normally, exit status = 13
My PID = 23563: Child PID = 23565 terminated normally, exit status = 7
My PID = 23562: Child PID = 23563 terminated normally, exit status = 7
The result of the expression is: 58
```



test2.tree

```
oslab33@orion:~/ex2/2.4$ ./make-expr-tree test2.tree
+
        *
                2
                +
                        3
                        4
        *
                6
                +
                        1
                        3
+: Starting...
*: Starting...
+: Starting...
2: Starting...
3: Starting...
*: Starting...
4: Starting...
Me: 23961, i have computed: 3 + 4 = 7
Me: 23958, i have computed: 2 * 7 = 14
6: Starting...
+: Starting...
1: Starting...
3: Starting...
Me: 23965, i have computed: 1 + 3 = 4
Me: 23959, i have computed: 6 * 4 = 24
Me: 23957, i have computed: 14 + 24 = 38


+(23957)─┬─*(23958)─┬─+(23961)─┬─3(23962)
         │          │          └─4(23963)
         │          └─2(23960)
         └─*(23959)─┬─+(23965)─┬─1(23966)
                    │          └─3(23967)
                    └─6(23964)


My PID = 23958: Child PID = 23960 terminated normally, exit status = 13
My PID = 23961: Child PID = 23962 terminated normally, exit status = 13
My PID = 23961: Child PID = 23963 terminated normally, exit status = 13
My PID = 23958: Child PID = 23961 terminated normally, exit status = 7
My PID = 23957: Child PID = 23958 terminated normally, exit status = 7
My PID = 23959: Child PID = 23964 terminated normally, exit status = 13
My PID = 23965: Child PID = 23966 terminated normally, exit status = 13
My PID = 23965: Child PID = 23967 terminated normally, exit status = 13
My PID = 23959: Child PID = 23965 terminated normally, exit status = 7
My PID = 23957: Child PID = 23959 terminated normally, exit status = 7
My PID = 23956: Child PID = 23957 terminated normally, exit status = 7
The result of the expression is: 38
```

test1.tree



```
oslab33@orion:~/ex2/2.4$ ./make-expr-tree test1.tree
*
        +
                5
                6
        10
*: Starting...
+: Starting...
6: Starting...
5: Starting...
Me: 23728, i have computed: 6 + 5 = 11
10: Starting...
Me: 23727, i have computed: 11 * 10 = 110


*(23727)─┬─+(23728)─┬─5(23730)
         │          └─6(23731)
         └─10(23729)


My PID = 23728: Child PID = 23730 terminated normally, exit status = 13
My PID = 23728: Child PID = 23731 terminated normally, exit status = 13
My PID = 23727: Child PID = 23728 terminated normally, exit status = 7
My PID = 23727: Child PID = 23729 terminated normally, exit status = 13
My PID = 23726: Child PID = 23727 terminated normally, exit status = 7
The result of the expression is: 110
oslab33@orion:~/ex2/2.4$ ./make-expr-tree bad.tree
Unexpected empty line
oslab33@orion:~/ex2/2.4$ ./make-expr-tree bad.tree test1.tree
Usage: ./make-expr-tree <input_tree_file>
```

Αντίστοιχα παραπάνω φαίνεται και η εκτέλεση με όχι καλά δομημένα αρχεία εισόδου και λάθος αριθμός ορισμάτων.

1. **Πόσες σωληνώσεις χρειάζονται στη συγκεκριμένη άσκηση ανά διεργασία; Θα μπορούσε κάθε γονική διεργασία να χρησιμοποιεί μόνο μία σωλήνωση για όλες τις διεργασίες παιδιά; Γενικά, μπορεί για κάθε αριθμητικό τελεστή να χρησιμοποιηθεί μόνο μια σωλήνωση;**
   Για αυτήν την άσκηση, αν υποθέσουμε ότι κάθε γονική διεργασία χρησιμοποιεί μόνο ένα pipeline για όλες τις διεργασίες-παιδιά της, τότε κάθε διεργασία-φύλλο (αριθμός) χρειάζεται ένα pipeline για να στείλει την τιμή του στη γονική διεργασία. Κάθε διαδικασία χωρίς φύλλα (τελεστής - αριθμητική υποέκφραση) χρειάζεται δύο pipelines, μία για να λάβει τις τιμές των παιδιών της και μία για να στείλει την τιμή του αποτελέσματός της στη μητρική της διαδικασία. Σημειώνεται πως η λειτουργία read ενός pipeline είναι blocking, δηλαδή η σωλήνωση "περιμένει" ακόμη κι αν δεν υπάρχει κάποιο περιεχόμενο να διαβαστεί, έως ότου να γραφτεί κάτι στη write λειτουργία της σωλήνωσης. Το γεγονός αυτό, μάλιστα, καθιστά μη αναγκαία την ύπαρξη κάποιου σήματος (SIGSTOP, SIGCONT) ή ακόμη και της λειτουργίας SLEEP για τη σωστή λειτουργία του προγράμματος.
   Συνεπώς, για μια αριθμητική παράσταση με n τελεστές και n+1 τιμές, θα χρειάζονταν συνολικά 2n+1 σωληνώσεις για ολόκληρη τη διαδικασία αξιολόγησης της έκφρασης.

Επίσης να σημειωθεί ότι αυτή η υπόθεση ισχύει μόνο για εκφράσεις που περιέχουν αντιμεταθετικούς τελεστές όπως πρόσθεση και πολλαπλασιασμός. Εάν υπάρχουν μη αντιμεταθετικοί τελεστές όπως η αφαίρεση και η διαίρεση, τότε κάθε γονική διεργασία θα πρέπει να χρησιμοποιεί τόσες σωλήνες όσες είναι οι θυγατρικές διεργασίες της συν ένα ακόμη για να στείλει το αποτέλεσμά της στη μητρική διεργασία.

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 σύστημα πολλαπλών επεξεργαστών, περισσότερες από μία διεργασίες μπορούν να εκτελούνται ταυτόχρονα, πράγμα που σημαίνει ότι πολλές διεργασίες μπορούν να εκτελούν υπολογισμούς ταυτόχρονα. Κατά την αποτίμηση μιας έκφρασης χρησιμοποιώντας ένα δέντρο διεργασίας, το πλεονέκτημα είναι ότι κάθε διεργασία στο δέντρο μπορεί να εκτελεστεί σε ξεχωριστό επεξεργαστή ή πυρήνα, επιτρέποντας παράλληλους υπολογισμούς.

Κατά την αξιολόγηση μιας έκφρασης χρησιμοποιώντας μια μεμονωμένη διεργασία, χρησιμοποιείται μόνο ένας επεξεργαστής ή πυρήνας, γεγονός που μπορεί να περιορίσει τη συνολική απόδοση του υπολογισμού. Με ένα δέντρο διεργασιών, ο φόρτος εργασίας μπορεί να διαιρεθεί σε πολλαπλές διεργασίες και να εκτελεστεί ταυτόχρονα, με αποτέλεσμα ταχύτερους χρόνους υπολογισμού και καλύτερη χρήση των πόρων.

Επιπλέον, η χρήση ενός δέντρου διεργασιών μπορεί επίσης να διευκολύνει τον χειρισμό σφαλμάτων και εξαιρέσεων στον υπολογισμό. Εάν παρουσιαστεί σφάλμα σε μια μεμονωμένη διεργασία, μπορεί να προκαλέσει αποτυχία ολόκληρου του υπολογισμού. Ωστόσο, με ένα δέντρο διεργασιών, τα σφάλματα μπορούν να αντιμετωπιστούν σε επίπεδο διεργασίας, επιτρέποντας στον υπολογισμό να συνεχίσει να εκτελείται σε άλλες διεργασίες.

Συνολικά, η χρήση ενός δέντρου διεργασιών μπορεί να βελτιώσει την απόδοση και την αποδοτικότητα του υπολογισμού σε ένα σύστημα πολλαπλών επεξεργαστών.