ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ
ΑΝΑΦΟΡΑ

Άσκηση 2

Αρετή Μέη 03120062
Ίρις-Ελευθερία Παλιατσού 03120639

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Ο κώδικας για την δημιουργία του δοσμένου δέντρου διεργασιών είναι ο εξής:

```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

//Process A:Creates B, Creates C, waits for its 2 children to terminate and then it exits with value 16
//Process B:Creates D, waits for its child to terminate and then it exits with value 19
//Process D:sleeps for a little bit and then it exits with value 13
//Process C:sleeps and then it exits with value 17
void fork_procs(void){
        int status; // initial process is A
        change_pname("A");
        printf("A created\n");

        pid_t pidB,pidC,pidD;
        pidB = fork(); //creating child B from parent A
        if (pidB < 0) {
                perror("fork");
                exit(1);
        }
        if (pidB == 0) { //in child process B
                change_pname("B");
                printf("B created\n");
                pidD = fork();
                if (pidD < 0) {
                        perror("fork");
                        exit(1);
                }
                if (pidD == 0) { //in child process D
                        change_pname("D");
                        printf("D created\n");
                        printf("D: sleeping...\n");
                        sleep(SLEEP_PROC_SEC);
                        printf("D:exiting\n");
                        exit(13);
                }
                if(pidD > 0) { //in parent process B
                        printf("B waiting\n"); // B has to wait for its child to terminate before it does
                        pidD = wait(&status);
                        explain_wait_status(pidD, status);
                        printf("B:exiting\n");
                        exit(19);
                }
```

```
        }
        if(pidB>0) { //in parent process A
                pidC = fork(); //we create child process C from parent A
                if (pidC < 0) {
                   perror("fork");
                   exit(1);
                }
                if (pidC == 0) { // in child process C
                   change_pname("C");
                   printf("C created\n");
                   printf("C: Sleeping...\n");
                   sleep(SLEEP_PROC_SEC);
                   printf("C:exiting\n");
                   exit(17);
                }
                if(pidC>0) { //in parent process A,which has to wait for its 2 children to terminate
                   printf("A waiting\n");
                   pidB=wait(&status);
                   explain_wait_status(pidB, status);
                   //Using     the waitpid() function instead of wait() helps if the parent process has multiple child processes,
                   //wait() may not know which child has terminated, and it woul
                   //need to call wait() repeatedly until all child processes have terminated.
                   //In contrast, the waitpid() function allows the parent process to specify the process ID
                   //of the child process it wants to wait for.
                   pidC=wait(&status); //waits for another children to terminate
                   explain_wait_status(pidC, status);
                   printf("A exiting ...\n");
                   exit(16);

                }
        }
}

int main(void) {
        pid_t pid;
        int status;
        // Fork root of proc 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); //the initial process sleeps and gives enough(hopefully) time to root to creat the process tree.

        /* 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;
}
```

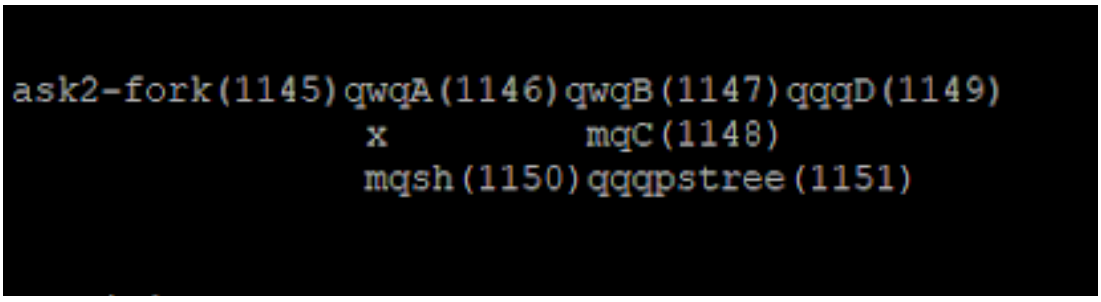
Η έξοδος εκτέλεσής του:

```
oslab25@orion:~/exercise2$ ./ask2-fork
A created
A waiting
B created
B waiting
D created
D: sleeping...
C created
C: Sleeping...


A(1259)─┬─B(1260)─┬─D(1262)
        │         └─{B}(1261)
        └─{A}(1259)


D:exiting
My PID = 1260: Child PID = 1262 terminated normally, exit status = 13
B:exiting
My PID = 1259: Child PID = 1260 terminated normally, exit status = 19
C:exiting
My PID = 1259: Child PID = 1261 terminated normally, exit status = 17
A exiting ...
My PID = 1258: Child PID = 1259 terminated normally, exit status = 16
oslab25@orion:~/exercise2$
```

Ερωτήσεις

1. Αν η διεργασία Α τερματιστεί πρόωρα, τότε τα παιδιά της γίνονται παιδιά της init(), με PID=1. Μέχρι να συμβεί αυτή η "υιοθεσία", τα παιδιά της Α βρίσκονται σε zombie state.

2.Το δέντρο διεργασιών που εμφανίζεται:



Οι επιπλέον διεργασίες είναι η αρχική διεργασία που με fork() παράγει τη Α, ask2-fork, η διεργασία sh με παιδί τη διεργασία pstree, που καλούνται από την show_pstree..Αυτό συμβαίνει γιατί εδώ ως γονική διεργασία για την εκτύπωση του δέντρου δεν δίνουμε την Α αλλά την αρχική διεργασία του προγράμματος ask2-fork, αφού το getpid() παίρνει το PID της διεργασίας αυτής.

3.Γιατί αν δημιουργηθούν ανεξέλεγκτα πολλές διεργασίες θα έχουμε πρόβλημα λόγω του πεπερασμένου χώρου, και το σύστημα ίσως υπερφορτωθεί.

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Ο πηγαίος κώδικας για την δημιουργία του αυθαίρετου δέντρου διεργασιών 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

//For the recursive making_tree function:
//We work according to 2 cases: whether our providen node has children or not.
//In case our node has no children, then its a leaf, so we don't have to recursively call the function.W
e just print a message that our process is created and sleeping, we call the sleep()
//function, and then the process exits, printing the appropriate message.
//We need to work differently when our node has children. First of all, we print the message that our pa
rent process is created, and then we call fork() in a loop that repeats as many times
//as our node's nr_children, in order to create the child processes. In that same loop we also check whe
ther the return value of fork is 0, which means that we are in a child process. That
//means that we need to recursively call the making_tree function, so that the same procedure is being f
ollowed, according to whether the process is a leaf or not.
//Then, we call wait(){we could also use waitpid()} also inside a loop that runs as many times as the nu
mber of children, because the parent HAS to wait for every child process to terminate
//before it does.Then, we print an exiting message that lets us know that the parent process is exiting.
The leaf processes exit using the value 9, and every other process exit using the
//value 10.


void making_tree(struct tree_node *root) {
    int i;
    int status;
    change_pname(root->name);
    //if node is a leaf, aka has no children
    if(root->nr_children == 0) {
            printf("%s created, sleeping\n", root->name);//%s
            sleep(SLEEP_PROC_SEC);
            printf("%s exiting!\n", root->name);
            exit(9);
    }
    //if our node has children
    printf("%s created\n", root->name);
                                                                                1,1           Top
```

```
    pid_t pid;
    for (i=0; i < root->nr_children; i++){
            pid = fork();
            if(pid < 0) { //error
                    printf("fork error");
                    exit(1);
            }
            if(pid == 0) { //in child process, recursively calling the function for child
               making_tree(root->children + i);
               exit(9);
            }
    }

    for(i=0; i<root->nr_children; i++) {
       pid = waitpid(-1, &status, 0);
       explain_wait_status(pid, status);
    }
    printf("%s is exiting\n", root->name);
    exit(10);
}

int main(int argc, char *argv[]) {
        if(argc!=2) { //1st argument:name, 2nd argument:infile that describes the tree
                fprintf(stderr, "Usage: %s <input_tree_file>\n", argv[0]);
                exit(1);

        }

        struct tree_node *root;
        root = get_tree_from_file(argv[1]);
        int status;
        print_tree(root);


        pid_t pid_root;
        pid_root = fork();
        if(pid_root < 0) { //error
                perror("main.fork");
                exit(1);
        }

        if(pid_root == 0) { // calling the making_tree function for the child
                making_tree(root);
        }

        sleep(SLEEP_TREE_SEC);
```

```
            if(pid == 0) { //in child process, recursively calling the function for child
               making_tree(root->children + i);
               exit(9);
            }
    }

    for(i=0; i<root->nr_children; i++) {
       pid = waitpid(-1, &status, 0);
       explain_wait_status(pid, status);
    }
    printf("%s is exiting\n", root->name);
    exit(10);
}

int main(int argc, char *argv[]) {
        if(argc!=2) { //1st argument:name, 2nd argument:infile that describes the tree
                 fprintf(stderr, "Usage: %s <input_tree_file>\n", argv[0]);
                 exit(1);

        }

        struct tree_node *root;
        root = get_tree_from_file(argv[1]);
        int status;
        print_tree(root);


        pid_t pid_root;
        pid_root = fork();
        if(pid_root < 0) { //error
                perror("main.fork");
                exit(1);
        }

        if(pid_root == 0) { // calling the making_tree function for the child
                making_tree(root);
        }

        sleep(SLEEP_TREE_SEC);

        show_pstree(pid_root);

        pid_root = wait(&status);
        explain_wait_status(pid_root, status);
        printf("root exiting\n");
        return 0;
}
                                                                    91,1          Bo
```

Η έξοδος εκτέλεσής του:

```
oslab25@orion:~/exercise2$ ./ask2-arbitr text2
A
        B
                D
        C
A created
C created, sleeping
B created
D created, sleeping


A(1288)qwqB(1289)qqqD(1291)
        mqC(1290)


C exiting!
My PID = 1288: Child PID = 1290 terminated normally, exit status = 9
D exiting!
My PID = 1289: Child PID = 1291 terminated normally, exit status = 9
B is exiting
My PID = 1288: Child PID = 1289 terminated normally, exit status = 10
A is exiting
My PID = 1287: Child PID = 1288 terminated normally, exit status = 10
```

Ερωτήσεις:

1. Παρατηρούμε ότι τα μηνύματα δημιουργιας και καταστροφής εμφανίζονται κατά πλάτος(BFS), που είναι λογικό καθώς αφού δημιουργηθεί ο πατέρας δημιουργεί μέσα στο for loop τα παιδιά του με χρήση της fork(). Τρέχοντας ωστόσο τον κώδικα παραπάνω από μια φορές για το παράδειγμα της εκφώνησης, παρατηρήσαμε ότι ο τρόπος που επιλέγεται ποιο παιδί θα δημιουργηθεί πρώτο, ποιο δεύτερο κλπ δεν είναι ο ίδιος κάθε φορά. Αυτό οφείλεται στο scheduling που κάνει το σύστημα για τις διεργασίες του.

1.3 Αποστολή και χειρισμός σημάτων

Το πρόγραμμα του ερωτήματος 1.2 επεκτάθηκε κάνοντας χρήση σημάτων ως εξής:

```
#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) {
        printf("PID = %ld, name %s, starting...\n",
                        (long)getpid(), root->name);
        change_pname(root->name);
        //In the for loop where fork() is called,if we used one pid and not a table, the pid
variable is overwritten in each iteration. Therefore, the kill() and explain_wait_status()
        //functions will not work correctly because they use the last value of pid. To fix th
is, we use an array to store the child process IDs and use it to refer to the correct child
        //process.
        int i, status;

        pid_t pid[root->nr_children];
        if(root->nr_children == 0) { //if our node is a leaf, we do not expand any nodes,we j
ust raise SIGSTOP.
                raise(SIGSTOP);
                printf("PID = %ld, name = %s is awake\n",
                   (long)getpid(), root->name);
                exit(0);
        }
        // our node has children of its own

          for(i=0; i<root->nr_children; i++) {
                pid[i] = fork();
                if(pid[i]<0) {
                        perror("fork error");
                        exit(1);
                }
                if(pid[i] == 0) {
                        fork_procs(root->children + i); //child node becomes the new root (pa
rent to its children)
                        exit(0);
                }
          }

          /*wait for all children to stop*/
          wait_for_ready_children(root->nr_children);
                                                                          1,1           Top
```

```
        raise(SIGSTOP); //The parent process raises the SIGSTOP signal on itself, suspendin
g its execution until it receives a SIGCONT signal. Children created but not executed

        printf("PID = %ld, name = %s is awake\n",
               (long)getpid(), root->name); //received the SIGCONT signal, the parent proces
s prints a message indicating that it is awake. Wakes up children as well

        for(i=0; i < root->nr_children; i++) { //Resuming child processes and waiting for t
heir termination
            kill(pid[i], SIGCONT);  //sends the SIGCONT signal to the process identified by
 the process ID pid[i].
            pid[i]=wait(&status);
            explain_wait_status(pid[i], status);
        }
}

/*
 * 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[]) {

        int status;
        struct tree_node *root;

        if (argc < 2){ //checks that input file has been specified as argument
                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_t pid_root = fork();
        if (pid_root < 0) {
                perror("main: fork");
                exit(1);
                                                                        86,1-8        62%
```

```
 *      the first process raises SIGSTOP.
 */

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

        int status;
        struct tree_node *root;

        if (argc < 2){ //checks that input file has been specified as argument
                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_t pid_root = fork();
        if (pid_root < 0) {
                perror("main: fork");
                exit(1);
}
        if (pid_root == 0) {
                /* Child */
                fork_procs(root);
                exit(1);
        }


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

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

        /* for ask2-signals */
        kill(pid_root, SIGCONT);

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

       wait(&status);
       explain_wait_status(pid_root, status);
       return 0;
}
                                                                  112,1
```

Η έξοδος εκτέλεσής του:

```
oslab25@orion:~/exercise2$ ./ask2-signals text2
PID = 1536, name A, starting...
PID = 1538, name C, starting...
My PID = 1536: Child PID = 1538 has been stopped by a signal, signo = 19
PID = 1537, name B, starting...
PID = 1539, name D, starting...
My PID = 1537: Child PID = 1539 has been stopped by a signal, signo = 19
My PID = 1536: Child PID = 1537 has been stopped by a signal, signo = 19
My PID = 1535: Child PID = 1536 has been stopped by a signal, signo = 19


A(1536)qwqB(1537)qqqD(1539)
        mqC(1538)


PID = 1536, name = A is awake
PID = 1537, name = B is awake
PID = 1539, name = D is awake
My PID = 1537: Child PID = 1539 terminated normally, exit status = 0
My PID = 1536: Child PID = 1537 terminated normally, exit status = 0
PID = 1538, name = C is awake
My PID = 1536: Child PID = 1538 terminated normally, exit status = 0
My PID = 1535: Child PID = 1536 terminated normally, exit status = 1
```

Ερωτήσεις

1. Αρχικά η χρήση σημάτων μας δίνει την δυνατότητα της ασύγχρονης επικοινωνίας των διεργασιών μας.Επίσης ενισχύει της απόκριση του συστήματός μας σε πραγματικό χρόνο, σε αντίθεση με την sleep(), η οποία μπορεί να εισάγει αχρείαστες καθυστερήσεις μεταξύ της εκτέλεσης των διεργασιών, και για παράδειγμα, όπως

υπάρχει και σε σχόλιο, μπορεί η show_pstree να μην προλάβει να εκτελεστεί καν.Ακόμη πολύ σημαντικό είναι το ότι με τη χρήση σημάτων είναι πολύ ευκολότερο να γίνει διαχείριση σφαλμάτων, όπως π.χ. σφάλματα κατάτμησης, δυνατότητα που η sleep() δεν παρέχει.

2. Η wait_for_ready_children() είναι συνάρτηση η οποία παίρνει ως όρισμα τον αριθμό παιδιών της διεργασίας που την καλεί, χρησιμοποιεί ένα loop για να κάνει wait για όλα τα παιδία της διεργασίας(αυτές οι κλήσεις της wait() ουσιαστικά πιάνουν τα raise(SIGSTOP) που κάθε διεργασία στέλνει στον εαυτό της). Στην συνέχεια του κώδικα ο πατέρας στέλνει και αυτός ένα SIGSTOP στον εαυτό του. Έτσι είναι δυνατόν να προκύψει η κατα βάθος(DFS) εκτύπωση των awakening μηνυμάτων.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

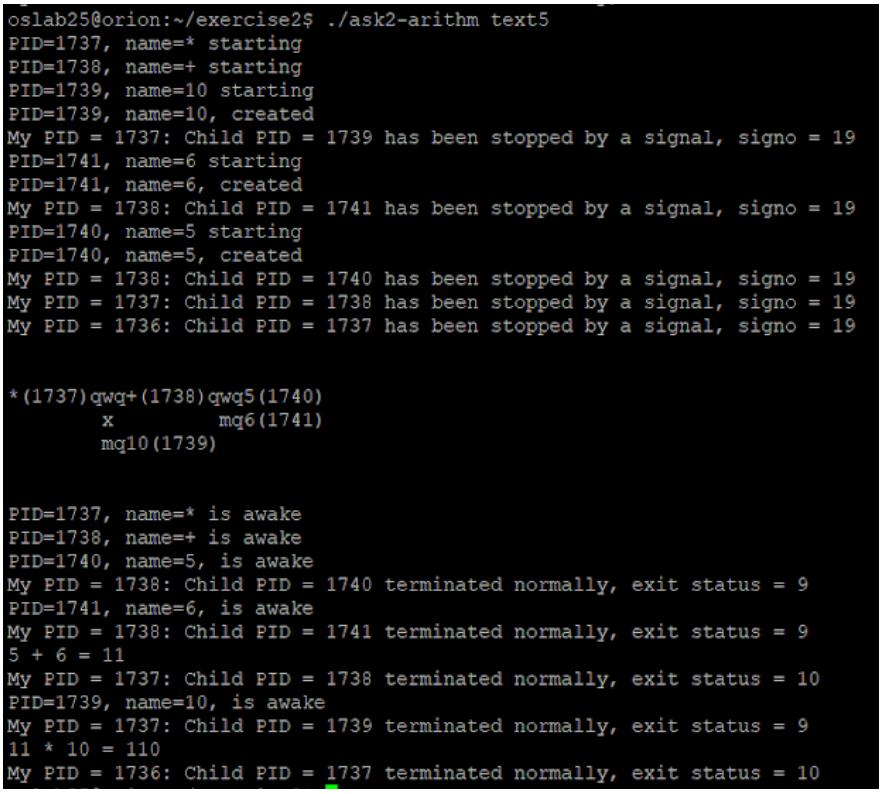
Η επέκταση του προγράμματος του ερωτήματος 1.3 για τον υπολογισμό δέντρων που αναπαριστούν αριθμητικές εκφράσεις είναι ως εξής:

```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"

int compute_result(int a, int b, char *c) {
        int result;

        char k = *c;
        if(k == '+')
                result =(a + b);
        else
                result = (a * b);
        printf("%d %s %d = %d\n", a, c, b, result);
        return result;
}

void fork_procs(struct tree_node *root, int fd) {
        printf("PID=%ld, name=%s starting\n",
                        (long)getpid(), root->name);
        change_pname(root->name);
        if(root->nr_children == 0) {
                int value = atoi(root->name);
                printf("PID=%ld, name=%s, created\n",
                                (long)getpid(), root->name);
                raise(SIGSTOP);
                if(write(fd, &value, sizeof(value))!=sizeof(value)) {
                        perror("write from pipe");
                        exit(1);

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

                exit(9); //leafs exit with value 9
        }
                int pfd[2]; // pfd[0] is for reading, pfd[1] is for writing
                int status, i;
                pid_t pid[root->nr_children];

                if (pipe(pfd) < 0) {
                    perror("pipe");
```

```
                exit(0);
        }


        for(i=0; i<root->nr_children; i++) {
                pid[i] = fork();
                if(pid[i]<0) {
                        perror("fork error");
                        exit(1);
                }
                if(pid[i] == 0) {
                        close(pfd[0]);
                        fork_procs(root->children + i, pfd[1]); //we write the resu
lt that the nodes give at pfd[1]
                        exit(10);
                }
        }
        close(pfd[1]);
        wait_for_ready_children(2);


        int Value[2]; //we will save the results of the 2 child nodes here
        raise(SIGSTOP);
        printf("PID=%ld, name=%s is awake\n",
                        (long)getpid(), root->name);

        for(i=0; i<root->nr_children; i++) {
                kill(pid[i], SIGCONT);
                if(read(pfd[0], &Value[i], sizeof(Value[i]))!=sizeof(Value[i])) {  /
/we read the 2 values from pfd[0] and we place them at Value[1] & Value[2]
                        perror("read from pipe");
                        exit(1);
                }
                pid[i] = wait(&status);
                explain_wait_status(pid[i], status);
        }
        int res = compute_result(Value[0], Value[1], root->name); //we compute the re
sult of our 3 nodes
        if(write(fd, &res, sizeof(res)) != sizeof(res)){
                perror("write to pipe");
                exit(1);
        } //we save the result in fd
        exit(10);
}

int main(int argc, char *argv[]) {
        int status;
```

```
        struct tree_node *root;

        if (argc < 2){
                fprintf(stderr, "Usage: %s <tree_file>\n", argv[0]);
                exit(1);
        }
        int pfd_root[2];
        if (pipe(pfd_root) < 0) {
                perror("pipe");
                exit(0);
        }
        root = get_tree_from_file(argv[1]);
        pid_t pid_root = fork();
        if (pid_root < 0) {
                perror("main: fork");
                exit(1);
        }
        if (pid_root == 0) {
                fork_procs(root, pfd_root[1]);
                exit(0);
        }
        wait_for_ready_children(1); //wait for the root to be ready

        show_pstree(pid_root);

        kill(pid_root, SIGCONT);

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

Εκτέλεση:



Ερωτήσεις

1. Για κάθε διεργασία που είναι μη τερματικός κόμβος χρησιμοποιείται μια μόνο σωλήνωση, στην οποία τοποθετούνται οι αριθμοί για την τέλεση της πράξης, και αυτό λόγω της αντιμεταθετικής ιδιότητας που ισχύει στη πρόσθεση και τον πολλαπλασιασμό. Στις αφαιρέσεις και τις διαιρέσεις, όπου δεν ισχύει η αντιμεταθετική ιδιότητα, δεν μπορεί να χρησιμοποιηθεί μόνο 1 σωλήνωση, καθώς η σειρά των αριθμών που χρησιμοποιούνται για την πράξη έχει σημασία.
2. Το πλεονέκτημα που προκύπτει είναι ότι οι πράξεις ίδιου επιπέδου(φύλλα, γονείς φύλλων, γονείς γονιών φύλλων κλπ)που απαιτούνται μπορούν να εκτελεστούν παράλληλα, άρα γρηγορότερα σε αντίθεση με την σειριακή εκτέλεση που εκτελεί διεργασίες την μια μετά την άλλη και είναι πολύ πιο χρονοβόρα.