ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά 2ης Εργαστηρικής Άσκησης

Ιάσων - Λάζαρος Παπαγεωργίου | Α.Μ: 03114034
Γρηγόρης Θανάσουλας | Α.Μ: 03114131

Άσκηση 1

Πήραμε ως βάση την ask2-fork.c και την εμπλουτίσαμε όπως φαίνεται παρακάτω:

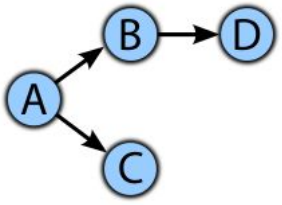
```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

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

  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");

```

```
42          // Forking in order to create D
43          p = fork();
44          if (p < 0) {
45                /* fork failed */
46                perror("Forking failed");
47                exit(1);
48          }
49          if (p == 0) {
50                /*Child  D process */
51                change_pname("D");
52                printf("D: I was created succesfully...\n");
53                printf("D: Sleeping...\n");
54                sleep(SLEEP_PROC_SEC);
55                printf("D: Exiting...\n");
56                exit(13);
57          }
58          p = wait(&status); //Node B waiting
59          explain_wait_status(p, status);
60          printf("B: Exiting...\n");
61          exit(19);
62    }
63    // Forking in order to create C
64    p = fork();
65    if (p < 0) {
66          /* fork failed */
67          perror("Forking failed");
68          exit(1);
69    }
70    if (p == 0) {
71          /*Child  C process */
72          change_pname("C");
73          printf("C: I was created succesfully...\n");
74          printf("C: Sleeping...\n");
75          sleep(SLEEP_PROC_SEC);
76          printf("C: Exiting...\n");
77          exit(17);
78    }
79
80    //Wait for 2 children to terminate
81    p = wait(&status);
82    explain_wait_status(p, status);
83
84    p = wait(&status);
85    explain_wait_status(p, status);
86
```

```c
  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(getpid());

  /* for ask2-signals */
```

```
132   /* kill(pid, SIGCONT); */
133
134   /* Wait for the root of the process tree to terminate */
135   pid = wait(&status);
136   explain_wait_status(pid, status);
137   return 0;
138 }
```

Έτσι για το δέντρο διεργασιών:



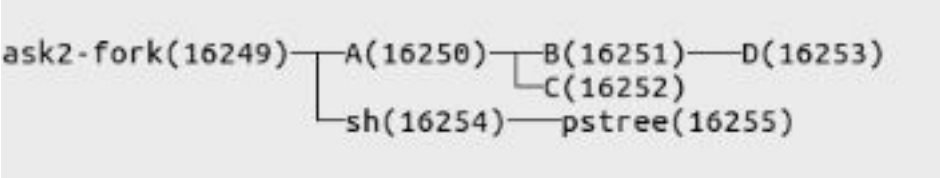
Έχουμε έξοδο:

```
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(19267)─┬─B(19268)──D(19270)
         └─C(19269)


C: Exiting...
D: Exiting...
My PID = 19267: Child PID = 19269 terminated normally, exit status = 17
My PID = 19268: Child PID = 19270 terminated normally, exit status = 13
B: Exiting...
My PID = 19267: Child PID = 19268 terminated normally, exit status = 19
A: Exiting...
My PID = 19266: Child PID = 19267 terminated normally, exit status = 16
```

Ερωτήσεις

1) Αν τερματιστεί πρόωρα η διεργασία A δίνοντας kill -KILL <A_pid>, οι διεργασίες παιδιά της θα είναι τώρα “**orphan**” processes (χαρακτηρισμός που δίνεται σε διεργασίες των οποίων ο γονέας πεθαίνει η σκοτώνεται πριν από τις ίδιες), και θα “υιοθετηθούν” από την **init** (η πρώτη από ένα σύνολο διεργασιών που τρέχουν στο παρασκήνιο για τη λειτουργία ενός Unix-based συστήματος).

2) Εάν αντί για show_pstree(pid) κάνουμε show_pstree(getpid()) θα εμφανιστεί το εξής:



Το pid που επιστρέφει η κλήση της fork από την διεργασία ask2-fork για την δημιουργία της διεργασίας Α, είναι το 16250, αυτό δηλαδή της Α. Δίνοντάς το ως όρισμα show_pstree(pid) θα εμφανίσει το δέντρο διεργασιών με ρίζα το Α.

Η κλήση της συνάρτησης getpid(), επιστρέφει το pid της διεργασίας που πραγματοποιεί την κλήση, και στη συγκεκριμένη περίπτωση είναι το pid της διεργασίας ask2-fork (είχαμε εκτελέσει ./ask2-fork). Συνεπώς, με όρισμα τη συνάρτηση getpid(), εμφανίζεται “ολόκληρο” το δέντρο διεργασιών με ρίζα τη διεργασία ask2-fork. Αξίζει να σημειώσουμε ότι στο δέντρο αυτό περιλαμβάνονται οι διεργασίες sh (standard command language interpreter), με παιδί το pstree. Εξετάζοντας το αρχείο που περιέχει την υλοποίηση της show_pstree, συμπεραίνουμε πως οι διεργασίες αυτές δημιουργούνται κατά την κλήση της συνάρτησης show_pstree.

```
116 void  show_pstree(pid_t p)
117 {
118     int ret;
119     char cmd[1024];
120
121     snprintf(cmd, sizeof(cmd), "echo; echo; pstree -G -c -p %ld; echo;echo",
122         (long)p);
123     cmd[sizeof(cmd)-1] = '\0';
124     ret = system(cmd);
125     if (ret < 0) {
126         perror("system");
127         exit(104);
128     }
129 }
```

3) Αυτό συμβαίνει διότι το σύστημα έχει πεπερασμένους πόρους. Τόσο τα blocks που 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 “κρασάρει” το σύστημα είναι συνετό να τίθενται κάποιοι περιορισμοί αναφορικά με το πλήθος των διεργασιών ανά χρήστη.

Ένα τέτοιο παράδειγμα προβληματικής περίπτωσης, είναι η επίθεση τύπου denial-of-serrvice **fork-bomb,** κατά την οποία μία διεργασίας κάνει επαναλαμβανόμενα fork τον εαυτό της, επιβραδύνοντας ή κρασάροντας το σύστημα.

Άσκηση 2

Και στην δεύτερη άσκηση χρησιμοποίησαμε το ask2-fork.c που μας δινόταν ως σκελετό και κάναμε τις απαραίτητες προσθήκες, καταλήγοντας στο ask2-signals.c :

```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);
```

```
40                fork_procs(root->children + i);
41          }
42          // Save child pid
43          childrenPID[i] = p;
44          printf("Child name: %s \t PID: %d\n",
45                (root->children + i)->name, childrenPID[i]);
46          // DFS creation, waiting for each child to change state
47          wait_for_ready_children(1);
48    }
49    /* ... */
50
51    /*
52    * Suspend Self
53    */
54    raise(SIGSTOP);
55    printf("PID = %ld, name = %s: I just woke up...\n",

56          (long)getpid(), root->name);
57
58    for (i=0; i < root->nr_children; i++) {
59          printf("PID = %ld, name = %s: Trying to wake up PID: %ld\n",
60                (long)getpid(), root->name, (long)childrenPID[i]);
61          kill(childrenPID[i], SIGCONT);
62          int diedPID = wait(&status);
63          explain_wait_status(diedPID, status);
64          printf("\n");
65    }
66    printf("\nPID = %ld, name = %s: Antio mataie toute kosme...\n",
67    (long)getpid(), root->name);
68    exit(getpid());
69 }
70
71 /*
72  * The initial process forks the root of the process tree,
73  * waits for the process tree to be completely created,
74  * then takes a photo of it using show_pstree().
75  *
76  * How to wait for the process tree to be ready?
77  * In ask2-{fork, tree}:
78  *      wait for a few seconds, hope for the best.
79  * In ask2-signals:
80  *      use wait_for_ready_children() to wait until
81  *      the first process raises SIGSTOP.
82  */
83
```

```c
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);
```

```
128
129   return 0;
130 }
```

Έτσι, για είσοδο που περιγράφεται από το graph1.tree

```
 1 # file that defines the tree
 2 # lines starting with '#' are comments
 3 # . each block defines a node
 4 # . each node is defined as:
 5 # 1st line:          name of node
 6 # 2nd line:          number of children
 7 # subsequent lines: name(s) of children
 8 # . blocks are seperated with empty lines
 9 # . no comments are allowed within a block
10 # . nodes must be placed in a DFS order
11
12 A
13 3
14 B
15 C
16 D
17
18 B
19 1
20 E
21
22 E
23 0
24
25 C
26 1
27 F
28
29 F
30 0
31
32 D
33 0
34
```

Έχουμε έξοδο:

```
A
        B
                E
        C
                F
        D
A: Ready to create child B...
A: Ready to create child C...
A: Ready to create child D...
B: I was created succesfully...
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(16871)─┬─A(16872)─┬─B(16873)──E(16876)
                 │          ├─C(16874)──F(16877)
                 │          └─D(16875)
                 └─sh(16878)──pstree(16879)


D: Exiting...
E: Exiting...
My PID = 16872: Child PID = 16875 terminated normally, exit status = 235
F: Exiting...
My PID = 16873: Child PID = 16876 terminated normally, exit status = 236
B: Exiting...
My PID = 16874: Child PID = 16877 terminated normally, exit status = 237
C: Exiting...
My PID = 16872: Child PID = 16873 terminated normally, exit status = 233
My PID = 16872: Child PID = 16874 terminated normally, exit status = 234
A: Exiting...
My PID = 16871: Child PID = 16872 terminated normally, exit status = 232
```

Ερωτήσεις

1) Δημιουργείται πρώτα ο πατέρας και μετά με τη σειρά που αναγράφονται στο αρχείο εισόδου τα παιδιά του. Στο προηγούμενο παράδειγμα πρώτα A, στη συνέχεια B, C και D. Η σειρά μετά το πρώτο επίπεδο όμως **δεν είναι αυστηρά καθορισμένη** καθώς η κάθε διεργασία θα δημιουργήσει **ανεξάρτητα** από τις άλλες τα δικά της παιδιά, χωρίς κάποιο συγχρονισμό. Έτσι, το αποτέλεσμα υπολείπεται συγκεκριμένης, αυστηρά ορισμένης σειράς.

   Πριν κάνει exit(), ο κάθε πατέρας περιμένει να τερματιστούν όλα του τα παιδιά. Στο εν λόγω παράδειγμα, πρώτα θα τερματιστούν τα φύλλα (εδώ D,E,F). Επειδή όμως χρησιμοποιούμε καθυστερήσεις, ούτε η σειρά της εξόδου είναι αυστηρά καθορισμένη, καθώς εξαρτάται από τη σειρά δημιουργίας των φύλλων, ή ισοδύναμα από το ποιας διεργασίας η sleep κλήθηκε νωρίτερα. Γενικεύοντας, για ένα συγκεκριμένο level του

δέντρου, οι διεργασίες που ανήκουν στο ίδιο level (ίδιο βάθος) τερματίζουν **χωρίς κάποια προκαθορισμένη σειρά, ικανοποιώντας μονάχα την απαίτηση να έχουν τερματίσει πρώτα όλα τα παιδιά της.**

Άσκηση 3

Επεκτείναμε την 1.2 ώστε οι διεργασίες να ελέγχονται με τη χρήση σημάτων. Έχουμε το ask2-signals.c:

```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) {
```

```
 37                /*Child  process */
 38                printf("%s: I was created succesfully...\n",
 39                      (root->children + i)->name);
 40                fork_procs(root->children + i);
 41          }
 42          // Save child pid
 43          childrenPID[i] = p;
 44          printf("Child name: %s \t PID: %d\n",
 45                (root->children + i)->name, childrenPID[i]);
 46          // DFS creation, waiting for each child to change state
 47          wait_for_ready_children(1);
 48    }
 49    /* ... */
 50
 51    /*
 52    * Suspend Self
 53    */
 54    raise(SIGSTOP);
 55    printf("PID = %ld, name = %s: I just woke up...\n",
 56          (long)getpid(), root->name);
 57
 58    for (i=0; i < root->nr_children; i++) {
 59          printf("PID = %ld, name = %s: Trying to wake up PID:
%ld\n",
 60                (long)getpid(), root->name, (long)childrenPID[i]);
 61          kill(childrenPID[i], SIGCONT);
 62          int diedPID = wait(&status);
 63          explain_wait_status(diedPID, status);
 64          printf("\n");
 65    }
 66    printf("\nPID = %ld, name = %s: Antio mataie toute kosme...\n",
 67    (long)getpid(), root->name);
 68    exit(getpid());
 69 }
 70
 71 /*
 72  * The initial process forks the root of the process tree,
 73  * waits for the process tree to be completely created,
 74  * then takes a photo of it using show_pstree().
 75  *
 76  * How to wait for the process tree to be ready?
 77  * In ask2-{fork, tree}:
 78  *      wait for a few seconds, hope for the best.
 79  * In ask2-signals:
 80  *      use wait_for_ready_children() to wait until
```

```c
 *      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 */
```

```
126   wait(&status);
127   explain_wait_status(pid, status);
128
129   return 0;
130 }
```

Έτσι, με είσοδο το αρχείο test2.tree:

```
 1 A
 2 2
 3 B
 4 C
 5
 6 B
 7 1
 8 D
 9
10 D
11 0
12
13 C
14 0
15
```

Έχουμε έξοδο:

```
PID = 17439, name A, starting...
A: Ready to create child B...
Child name: B    PID: 17440
B: I was created succesfully...
PID = 17440, name B, starting...
B: Ready to create child D...
Child name: D    PID: 17441
D: I was created succesfully...
PID = 17441, name D, starting...
My PID = 17440: Child PID = 17441 has been stopped by a signal, signo = 19
My PID = 17439: Child PID = 17440 has been stopped by a signal, signo = 19
A: Ready to create child C...
Child name: C    PID: 17442
C: I was created succesfully...
PID = 17442, name C, starting...
My PID = 17439: Child PID = 17442 has been stopped by a signal, signo = 19
My PID = 17438: Child PID = 17439 has been stopped by a signal, signo = 19


A(17439)─┬─B(17440)──D(17441)
         └─C(17442)


PID = 17439, name = A: I just woke up...
PID = 17439, name = A: Trying to wake up PID: 17440
PID = 17440, name = B: I just woke up...
PID = 17440, name = B: Trying to wake up PID: 17441
PID = 17441, name = D: I just woke up...

PID = 17441, name = D: Antio mataie toute kosme...
My PID = 17440: Child PID = 17441 terminated normally, exit status = 33


PID = 17440, name = B: Antio mataie toute kosme...
My PID = 17439: Child PID = 17440 terminated normally, exit status = 32

PID = 17439, name = A: Trying to wake up PID: 17442
PID = 17442, name = C: I just woke up...

PID = 17442, name = C: Antio mataie toute kosme...
My PID = 17439: Child PID = 17442 terminated normally, exit status = 34


PID = 17439, name = A: Antio mataie toute kosme...
My PID = 17438: Child PID = 17439 terminated normally, exit status = 31
```

Ερωτήσεις

1) Όπως φάνηκε και από τις προηγούμενες ασκήσεις, με **τη χρήση καθυστερήσεων η συμπεριφορά των διεργασιών και κατά συνέπεια η λειτουργία του προγράμματος καθίστανται απρόβλεπτες, άρα και ο συγχρονισμός είναι ανέφικτος**. Για παράδειγμα, 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. Άλλο πρόβλημα είναι πως η DFS διάσχιση του δέντρου είναι αδύνατη.

Με τη χρήση σημάτων είναι πολύ ευκολότερος ο έλεγχος του προγράμματος, καθώς μπορεί να ελεγχθεί η συμπεριφορά της κάθε διεργασίας ανάλογα με το τι σήμα αποστέλλεται σε αυτή. Έτσι μπορούμε να έχουμε πλήρη έλεγχο της κατασκευής του δέντρου και η υλοποίηση του DFS είναι εφικτή. Επιπλέον, η φωτογράφιση του δέντρου από την show_pstree θα γίνει σίγουρα αφού θα έχει κατασκευαστεί ολόκληρο, χωρίς να τίθεται κάποιο ρίσκο.

2) **Η wait_for_ready_children() περιμένει μέχρι ο αριθμός παιδιών που της δίνεται σαν όρισμα να αλλάξει κατάσταση. Η χρήση της μέσα στο for loop όπως φαίνεται στον κώδικά μας παραπάνω 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 δέντρου διεργασιών που επιθυμούσαμε

   Εάν παραλειπόταν, ο συγχρονισμός δε θα ήταν δυνατός, και η σειρά δημιουργία των διεργασιών του δέντρου θα ήταν μη προκαθορισμένη.

   Τέλος, στη γενικότερη περίπτωση, 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.

Άσκηση 4

Ο πηγαίος κώδικας της ask2-calculate.c είναι :

```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);
```

```
 20   //If process is a leaf node
 21   if (root->nr_children == 0) {
 22         int value = atoi(root->name);
 23         if(write(write_fd, &value, sizeof(value)) != sizeof(value))
{
 24               perror("Pipe Write Failed\n");
 25               exit(1);
 26         }
 27         printf("PID: %ld (%s): Wrote Value:%d on Pipe %d and is
Exiting...\n",
 28               (long)getpid(), name, value, write_fd);
 29         sleep(SLEEP_PROC_SEC);
 30         exit(getpid());
 31   }
 32
 33   pid_t p;
 34   // Iterate for every child
 35   // If node is not a leaf
 36   int i;
 37   int fd[2];
 38   if (pipe(fd) < 0) {
 39         perror("Pipe Creation Failed");
 40         exit(1);
 41   }
 42   printf("PID: %ld (%s): Pipe [%d, %d] to children created
successfully\n",
 43               (long)getpid(), root->name, fd[0], fd[1]);
 44
 45   for (i = 0; i < 2; i++) {
 46         printf("PID: %ld (%s): Ready to create child %s...\n",
 47               (long)getpid(), name, (root->children + i)->name);
 48         p = fork();
 49         if (p < 0) {
 50               /* fork failed */
 51               perror("Forking failed");
 52               exit(1);
 53         }
 54
 55         if (p == 0) {
 56               /*Child  process */
 57               fork_procs(root->children + i, fd[1]);
 58               printf("\nWTF\n");
 59         }
60    }
 61
```

```
 62    int operands[2];
 63    int value;
 64
 65    for (i=0; i < 2; i++) {
 66          if (read(fd[0], &value, sizeof(value)) != sizeof(value)) {
 67                perror("Read from Pipe Failed");
 68                exit(1);
 69          }
 70          printf("PID: %ld (%s): Reading from pipe value no. %d :
%d\n",
 71                (long)getpid(), root->name, i, value);
 72          operands[i] = value;
 73
 74          // p = wait(&status); //Node B waiting
 75          //explain_wait_status(p, status);
 76    }
 77    int result;
 78
 79    if (strcmp(root->name, "+") == 0)
 80          result = operands[0] + operands[1];
 81    else
 82          result = operands[0] * operands[1];
 83    if (write(write_fd, &result, sizeof(result)) != sizeof(result)) {
 84          perror("Pipe Write Failed");
 85          exit(1);
 86    }
 87    printf("PID: %ld (%s): Wrote calculated result (%d) to Pipe %d\n",
 88                (long)getpid(), name, result, write_fd);
 89    sleep(SLEEP_PROC_SEC);
 90    exit(getpid());
 91 }
 92
 93 /*
 94  * The initial process forks the root of the process tree,
 95  * waits for the process tree to be completely created,
 96  * then takes a photo of it using show_pstree().
 97  *
 98  * How to wait for the process tree to be ready?
 99  * In ask2-{fork, tree}:
100  *      wait for a few seconds, hope for the best.
101  * In ask2-signals:
102  *      use wait_for_ready_children() to wait until
103  *      the first process raises SIGSTOP.
104  */
105 int main(int argc, char *argv[])
```

```c
{
    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); */
```

```
150   if(read(fd[0], &final_value, sizeof(final_value))
151               != sizeof(final_value)) {
152         perror("Pipe Read Failed");
153         exit(1);
154   }
155   printf("The result is %d\n", final_value);
156   return 0;
157 }
```

Με είσοδο

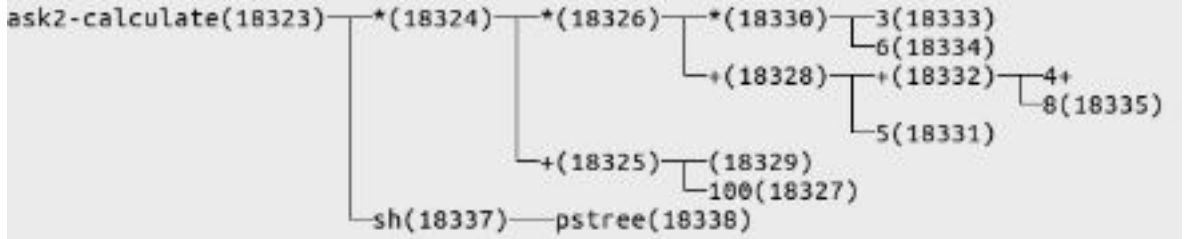
```
 1 # file that defines the tree
 2 # lines starting with '#' are comments
 3 # . each block defines a node
 4 # . each node is defined as:
 5 # 1st line:         name of node
 6 # 2nd line:         number of children
 7 # subsequent lines: name(s) of children
 8 # . blocks are seperated with empty lines
 9 # . no comments are allowed within a block
10 # . nodes must be placed in a DFS order
11
12 *
13 2
14 +
15 *
16
17 +
18 1
19 100
20
21 100
22 0
23
24 *
25 2
26 +
27 *
28
29 +
30 2
31 5
32 +
33
34 5
```

```
35 0
36
37 +
38 2
39 8
40 4
41
42 8
43 0

44
45 4
46 0
47
48 *
49 2
50 3
51 6
52
53 3
54 0
55
56 6
57 0
58
```

Έχουμε αποτέλεσμα:

```
*
        +
                100
        *
                +
                        5
                        +
                                8
                                4
                *
                        3
                        6
PID: 18324 (*): I was created succesfully...
PID: 18324 (*): Pipe [5, 6] to children created successfully
PID: 18324 (*): Ready to create child +...
PID: 18324 (*): Ready to create child *...
PID: 18325 (+): I was created succesfully...
PID: 18325 (+): Pipe [7, 8] to children created successfully
PID: 18325 (+): Ready to create child 100...
PID: 18326 (*): I was created succesfully...
PID: 18326 (*): Pipe [7, 8] to children created successfully
PID: 18326 (*): Ready to create child +...
PID: 18325 (+): Ready to create child ...
PID: 18326 (*): Ready to create child *...
PID: 18327 (100): I was created succesfully...
PID: 18328 (+): I was created succesfully...
PID: 18327 (100): Wrote Value:100 on Pipe 8 and is Exiting...
PID: 18328 (+): Pipe [9, 10] to children created successfully
PID: 18328 (+): Ready to create child 5...
PID: 18325 (+): Reading from pipe value no. 0 : 100
PID: 18329 (): I was created succesfully...
PID: 18330 (*): I was created succesfully...
PID: 18329 (): Wrote Value:0 on Pipe 8 and is Exiting...
PID: 18325 (+): Reading from pipe value no. 1 : 0
PID: 18330 (*): Pipe [9, 10] to children created successfully
PID: 18325 (+): Wrote calculated result (100) to Pipe 6
PID: 18328 (+): Ready to create child +...
PID: 18324 (*): Reading from pipe value no. 0 : 100
PID: 18330 (*): Ready to create child 3...
PID: 18331 (5): I was created succesfully...
PID: 18331 (5): Wrote Value:5 on Pipe 10 and is Exiting...
PID: 18328 (+): Reading from pipe value no. 0 : 5
PID: 18332 (+): I was created succesfully...
PID: 18330 (*): Ready to create child 6...
PID: 18332 (+): Pipe [11, 12] to children created successfully
PID: 18333 (3): I was created succesfully...
PID: 18332 (+): Ready to create child 8...
PID: 18333 (3): Wrote Value:3 on Pipe 10 and is Exiting...
PID: 18330 (*): Reading from pipe value no. 0 : 3
PID: 18334 (6): I was created succesfully...
PID: 18332 (+): Ready to create child 4...
PID: 18334 (6): Wrote Value:6 on Pipe 10 and is Exiting...
PID: 18335 (8): I was created succesfully...
PID: 18330 (*): Reading from pipe value no. 1 : 6
PID: 18330 (*): Wrote calculated result (18) to Pipe 8
PID: 18326 (*): Reading from pipe value no. 0 : 18
PID: 18335 (8): Wrote Value:8 on Pipe 12 and is Exiting...
PID: 18332 (+): Reading from pipe value no. 0 : 8
PID: 18336 (4): I was created succesfully...
PID: 18336 (4): Wrote Value:4 on Pipe 12 and is Exiting...
```

```
PID: 18332 (+): Reading from pipe value no. 1 : 4
PID: 18332 (+): Wrote calculated result (12) to Pipe 10
PID: 18328 (+): Reading from pipe value no. 1 : 12
PID: 18328 (+): Wrote calculated result (17) to Pipe 8
PID: 18326 (*): Reading from pipe value no. 1 : 17
PID: 18326 (*): Wrote calculated result (306) to Pipe 6
PID: 18324 (*): Reading from pipe value no. 1 : 306
PID: 18324 (*): Wrote calculated result (30600) to Pipe 4
```




```
The result is 30600
```

Ερωτήσεις

1) Επειδή στην συγκεκριμένη άσκηση χρησιμοποιούμε μόνο **προσθέσεις και πολλαπλασιασμούς, που είναι πράξεις αντιμεταθετικές, δεν χρειάζονται παραπάνω από ένα pipe για κάθε τριάδα γονιού - 2 παιδιών.**

   Εάν βέβαια έπρεπε να εκτελέσουμε αφαιρέσεις και διαιρέσεις, που είναι πράξεις μη αντιμεταθετικές, τότε -για την πιο straightforward προσέγγιση- θα έπρεπε να έχουμε 2 pipes, δηλαδή ένα pipe για κάθε γονιό και παιδί, προκειμένου να γνωρίζουμε ποιος είναι ο κάθε τελεστέος (αφαιρετέος ή αφαιρέτης, διαιρετέος ή διαιρέτης) που επιστρέφουν τα παιδιά διεργασίες στον πατέρα (αφού A - B είναι διάφορο του B - A).

   Σημείωση*: Για την γενική περίπτωση, όπου έχουμε διαίρεση και αφαίρεση, με μία διαφορετική προσέγγιση σκεφτήκαμε το εξής: Ενδεχομένως θα μπορούσαμε να διατηρήσουμε το ένα pipe για κάθε τριάδα γονιός - 2 παιδιά, διαμέσου του οποίου θα περνούσαν τα παιδιά ένα struct στον γονέα, με κατάλληλα δεδομένα εντός του struct για το αν είναι το αριστερό ή το δεξί παιδί που έγραψε την τιμή-αποτέλεσμα στο pipe. Αυτό σαφώς προϋποθέτει την τροποποίηση του κώδικα ώστε κατά το fork να περνιούνται αντίστοιχες μεταβλητές - παράμετροι στις διεργασίες παιδιά για το αν είναι αριστερό παιδί ή δεξί του πατέρα.*

2) Σε ένα σύστημα με πολλούς επεξεργαστές μπορούμε να εκτελέσουμε ένα σύνολο ανεξάρτητων υπολογισμών - διεργασιών παράλληλα (κάθε υπολογισμός σε διαφορετικό πυρήνα) και έτσι το αποτέλεσμα της συνολικής αριθμητικής έκφρασης να βρεθεί πιο γρήγορα σε σχέση με την περίπτωση που μία διεργασία εκτελεί όλους τους υπολογισμούς