2^{η} Αναφορά στα Λειτουργικά Συστήματα

Ιωάννης Αλεξόπουλος (03117001)

Αλέξανδρος Κυριακάκης (03112163)

June 6, 2020

1 Σειρά

1.1 Άσκηση

Source Code

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

#include "proc-common.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */
void fork_procs(void)
{
 /*
  * initial process is A.
  */
 pid_t B_pid, C_pid, D_pid;
 int status;
 change_pname("A");
 printf("A created\n");
 B_pid = fork();
 if (B_pid < 0) {
  perror("fork creating tree");
```

```
30      exit(1);
31     }
32     if (B_pid == 0) {
33      change_pname("B");
34      printf("B created\n");
35      D_pid = fork();
36      if (D_pid < 0) {
37       perror("fork creating tree");
38       exit(1);
39      }
40      if (D_pid == 0) {
41       change_pname("D");
42       printf("D created\n");
43       printf("D sleeping\n");
44       sleep(SLEEP_PROC_SEC);
45       printf("D exiting\n");
46       exit(13);
47      }
48      if (D_pid > 0) {
49       printf("B waiting\n");
50       D_pid = wait(&status);
51       explain_wait_status(D_pid, status);
52       printf("B exiting\n");
53       exit(19);
54      }
55     }
56     if (B_pid > 0) {
57      C_pid = fork();
58      if (C_pid < 0) {
59       perror("fork creating tree");
60       exit(1);
61      }
62      if (C_pid == 0) {
63       change_pname("C");
64       printf("C created\n");
65       printf("C sleeping\n");
66       sleep(SLEEP_PROC_SEC);
67       printf("C exiting\n");
68       exit(17);
69      }
70      if (C_pid > 0) {
71       printf("A waiting\n");
72       C_pid = waitpid(-1, &status, 0);
73       explain_wait_status(C_pid, status);
74       C_pid = waitpid(-1, &status, 0);
75       explain_wait_status(C_pid, status);
76       printf("A exiting\n");
77       exit(16);
```

```
  }
 }
}

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */
int main(void)
{
 pid_t pid;
 int status;

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

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

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

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

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

```

```
126   /* Wait for the root of the process tree to terminate */
127   pid = wait(&status);
128   explain_wait_status(pid, status);
129
130   return 0;
131 }
```

Output

```
A created
B created
A waiting
C created
C sleeping
B waiting
D created
D sleeping


A(18877)     B     (18878)     D     (18880)
             C    (18879)


C exiting
D exiting
My PID = 18878: Child PID = 18880 terminated normally, exit status = 13
B exiting
My PID = 18877: Child PID = 18879 terminated normally, exit status = 17
My PID = 18877: Child PID = 18878 terminated normally, exit status = 19
A exiting
My PID = 18876: Child PID = 18877 terminated normally, exit status = 16
```

1.1.1 Kill

Αν τερματιστεί πρόωρα η διεργασία A δίνοντας "kill -KILL A_pid", οι διεργασίες παιδιά της θα είναι τώρα "orphan" processes, και θα "υιοθετηθούν" από την init.

1.1.2 getpid()

```
A created
B created
A waiting
C created
C sleeping
B waiting
D created
D sleeping


ask2_1(18914)     A     (18915)     B     (18916)     D     (18918)
                                C    (18917)
                  sh    (18919)     pstree     (18920)


C exiting
D exiting
My PID = 18915: Child PID = 18917 terminated normally, exit status = 17
```

```
My PID = 18916: Child PID = 18918 terminated normally, exit status = 13
B exiting
My PID = 18915: Child PID = 18916 terminated normally, exit status = 19
A exiting
My PID = 18914: Child PID = 18915 terminated normally, exit status = 16
```

Με όρισμα τη συνάρτηση getpid(), εμφανίζεται "ολόκληρο" το δέντρο διεργασιών με ρίζα τη διεργασία ask2_1. Το δέντρο αυτό περιλαμβάνονται οι διεργασίες sh (standard command language interpreter), με παιδί το pstree, που καλούνται απο την show_pstree.

1.1.3 Resources

Σε ένα σύστημα πολλαπλών χρηστών χωρίς όριο στο πλήθος διεργασιών, δεδομένου των πεπερασμένων πόρων, ένας κακόβουλος ή αφελής χρήστης θα μπορούσε να υπερφωρτόσει το σύστημα με "άπειρες" διεργασίες. Αυτό εύκολα συμβαίνει με χρήση της συνάρτησης fork() σε ένα ατέρμονο βρόγχο. Έτσι, είναι απαραίτητος ο περιορισμός του πλήθους των διεργασιών ανα χρήστη.

1.2 Άσκηση

Source Code

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "tree.h"
#include "proc-common.h"

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

/*
 * 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.
 */
/* Leaves exit with 5, Non-Leaves exit with 2 */
void fork_procs (struct tree_node *root)
{
 int i;
 int status;
 change_pname(root->name);
```

```c
 /* if current node is a leaf */
 if (root->nr_children == 0) {
  printf("Leaf %s created and sleeping!\n",root->name);
  sleep(SLEEP_PROC_SEC);
  printf("Leaf %s exiting!\n", root->name);
  exit(5);
 }
 /* if current node is not a leaf */
 printf("Node %s created\n", root->name);
 pid_t pid_child;
 for (i = 0; i < root->nr_children; i++) {
  pid_child = fork();
  if (pid_child == 0) {
   /* call recursive function for child */
   fork_procs (root->children + i);
  }
 }
 /* wait for all children to exit */
 printf("Node %s waiting for all children to exit!\n",root->name);
 for (i = 0; i < root->nr_children; i++) {
  pid_child = waitpid(-1, &status, 0);
  explain_wait_status(pid_child, status);
 }
 printf("Node %s exiting\n", root->name);
 exit(2);

}


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

 pid_t pid;
 int status;
 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 = fork();
 if (pid < 0) {
  perror("main:fork");
  exit(1);
 }
 if (pid == 0) {
  fork_procs(root);
 }
```

```
79   sleep(SLEEP_TREE_SEC);
80   show_pstree(pid);
81   pid  = wait(&status);
82   explain_wait_status(pid, status);
83   printf("Root: All done, exiting...\n");
84
85   return 0;
86  }
```

Output

```
A
  B
    E
    F
  C
  D
Node A created
Node B created
Node A waiting for all children to exit!
Leaf C created and sleeping!
Leaf D created and sleeping!
Leaf E created and sleeping!
Node B waiting for all children to exit!
Leaf F created and sleeping!


A(19096)     B     (19097)     E     (19100)
                           F    (19101)
             C    (19098)
             D    (19099)


Leaf C exiting!
Leaf D exiting!
Leaf E exiting!
My PID = 19096: Child PID = 19098 terminated normally, exit status = 5
Leaf F exiting!
My PID = 19096: Child PID = 19099 terminated normally, exit status = 5
My PID = 19097: Child PID = 19100 terminated normally, exit status = 5
My PID = 19097: Child PID = 19101 terminated normally, exit status = 5
Node B exiting
My PID = 19096: Child PID = 19097 terminated normally, exit status = 2
Node A exiting
My PID = 19095: Child PID = 19096 terminated normally, exit status = 2
Root: All done, exiting...
```

1.2.1 BFS

Τα μυνήματα δημιουργίας εμφανίζονται με breadth first traversing και τερματισμού αντίστροφα, με αξιοσημίωτο το γεγονός ότι δεν είναι ντετερμινιστική η σειρά δημιουργίας και καταστροφής τους αφού αυτό εξαρτάται απο τον τρόπο που το σύστημα διαχειρίζεται τις διεργασίες του (scheduling).
Πιο συγκεκριμένα έκαστος "πατέρας" δημιουργεί τα παιδιά του και περιμένει αυτά να τερματήσουν. Τα παιδία με τη σειρά τους κάνουν το ίδιο εώς ότου κάποιο από αυτά είναι φύλλο που μπαίνει σε sleep() μέχρι να τελειώσει η κατασκευή ολόκληρου του δέντρου (εξαιρετικά χρονοβόρο). Έπειτα τα φύλλα ξεκινούν και τερματίζουν και το δέντρο αποδομίται αντίστροφα, αλλά όχι με deterministic τρόπο, αφού

δεν υπάρχει συγχρονισός μεταξύ δημιουργίας/θανάτου των διεργασιών που βρίσκονται στο ίδιο επίπεδο του δέντρου.

1.3 Άσκηση

Source Code

```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 "proc-common.h"
#include "tree.h"
void fork_procs(struct tree_node *root)
{
 /*
  * Start
  */
 int i, status, children;
 printf("PID = %ld, name %s, starting...\n",
   (long)getpid(), root->name);
 change_pname(root->name);
 if (root->nr_children == 0) {
  raise(SIGSTOP);
  printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);
  exit(0);
 }
 /* create proccess tree*/
 else {
  children = root->nr_children;
  pid_t pid[children];
  for (i = 0; i < children; i++) {
   pid[i] = fork();
   if (pid[i] == 0) {
    fork_procs(root->children + i);
    exit(0);
   }
   if (pid[i] < 0) {
    perror("fork in fork_procs");
    exit(1);
   }
  }
  /*wait for all children to stop*/
  wait_for_ready_children(children);
  /*stop*/
  raise(SIGSTOP);
```

```
43    printf("*PID = %ld, name = %s is awake\n", (long)getpid(), root->name);
44    /* wake up each child and wait for it to exit leading to DFS messages */
45    for (i = 0; i < children; i++) {
46  //    printf("children of proccess with id %ld : %d\n", (long)getpid(), children);
47      /* wake up first child */
48      kill(pid[i], SIGCONT);
49  //    printf("proccess with id %ld sent SIGCONT to child with id %ld\n",
    ↪   (long)getpid(), pid[i]);
50      /* wait for it to exit */
51      pid[i] = waitpid(pid[i], &status, 0);
52      explain_wait_status(pid[i], status);
53    }
54   }
55  }
56
57  /*
58   * The initial process forks the root of the process tree,
59   * waits for the process tree to be completely created,
60   * then takes a photo of it using show_pstree().
61   *
62   * How to wait for the process tree to be ready?
63   * In ask2-{fork, tree}:
64   *      wait for a few seconds, hope for the best.
65   * In ask2-signals:
66   *      use wait_for_ready_children() to wait until
67   *      the first process raises SIGSTOP.
68   */
69
70  int main(int argc, char *argv[])
71  {
72   pid_t pid;
73   int status;
74   struct tree_node *root;
75   struct sigaction sa;
76   if (sigaction(SIGCHLD, &sa, NULL) < 0) {
77    perror("sigaction");
78    exit(1);
79   }
80
81   if (argc < 2){
82    fprintf(stderr, "Usage: %s <tree_file>\n", argv[0]);
83    exit(1);
84   }
85
86   /* Read tree into memory */
87   root = get_tree_from_file(argv[1]);
88
89   /* 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);
 pid = waitpid(pid, &status, 0);
 explain_wait_status(pid, status);
 /* Wait for the root of the process tree to terminate */


 return 0;
}
```

Output

```
PID = 19370, name A, starting...
PID = 19371, name B, starting...
PID = 19372, name C, starting...
PID = 19373, name D, starting...
My PID = 19370: Child PID = 19372 has been stopped by a signal, signo = 19
PID = 19374, name E, starting...
My PID = 19370: Child PID = 19373 has been stopped by a signal, signo = 19
PID = 19375, name F, starting...
My PID = 19371: Child PID = 19374 has been stopped by a signal, signo = 19
My PID = 19371: Child PID = 19375 has been stopped by a signal, signo = 19
My PID = 19370: Child PID = 19371 has been stopped by a signal, signo = 19
My PID = 19369: Child PID = 19370 has been stopped by a signal, signo = 19


A(19370)     B     (19371)     E     (19374)
                           F    (19375)
```

```
            C   (19372)
            D   (19373)


*PID = 19370, name = A is awake
*PID = 19371, name = B is awake
PID = 19374, name = E is awake
My PID = 19371: Child PID = 19374 terminated normally, exit status = 0
PID = 19375, name = F is awake
My PID = 19371: Child PID = 19375 terminated normally, exit status = 0
My PID = 19370: Child PID = 19371 terminated normally, exit status = 0
PID = 19372, name = C is awake
My PID = 19370: Child PID = 19372 terminated normally, exit status = 0
PID = 19373, name = D is awake
My PID = 19370: Child PID = 19373 terminated normally, exit status = 0
My PID = 19369: Child PID = 19370 terminated normally, exit status = 1
```

1.3.1 Signal Advantages

Με την χρήση σημάτων έχουμε δύο βασικά προτερήματα:

1. Αρχικά, το χρονικό πλεονέκτημα είναι τεράστιο, αφού δεν χρειάζεται καμιά διεργασία να καλέσει την sleep().

2. Έπειτα, δεδομένου του συγχρονισού που πετυχένουμε με τα σήματα, είναι deterministic η διαδικασία διάσχισης του δέντρου, καθώς μας δίδεται και η επιλογή για depth first traversing αντι breadth first traversing.

3. Τέλος, εκμηδενίζεται και η πιθανότητα αστοχίας της συνάρτησης show_pstree(pid).

1.3.2 wait_for_ready_children()

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 διασφαλίζει την κατά βάθος (DFS) σειρά εμφάνισης των μηνυμάτων. Εάν παραλειπόταν, ο συγχρονισμός δε θα ήταν δυνατός, και η σειρά δημιουργία των διεργασιών του δέντρου θα ήταν μη προκαθορισμένη. Γενικά, ο πατέρας κάθε παιδιού πρέπει να καλεί τη συνάρτηση wait() μέσω της wait_for_ready_children προκειμένου να αποφεύγεται η δημιουργία "orphan" παιδιών.

1.4 Άσκηση

Source Code

```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"
#include <signal.h>

/*
```

```c
 * 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.
 */

char mycompute (char kind, char a, char b){
 int a_int = a - '0', b_int = b - '0';
 switch (kind){
 case '+': {printf("I will execute %i + %i with result %i !\n",a_int,b_int,a_int+b_int); return a_int + b_int + '0';}
 case '*': {printf("I will execute %i * %i with result %i !\n",a_int,b_int,a_int*b_int); return a_int * b_int + '0';}
 }
 return -1;
}

void fork_procs (struct tree_node *root, int fd){
 int i;
 int status;
 // If NULL return ERROR
 if (root == NULL) {
  printf("empty node");
  return;
 }

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

 change_pname(root->name);
 if (root->nr_children == 0) {
  printf("Leaf %s created\n",root->name);
  // We stop here and wait for parent to give SIGCONT after all leafs created
  raise(SIGSTOP);
  // root->name is something like string so we cast it to int (char)
  char cur = atoi(root->name) + '0';
  // then we write it to the pipe
  if (write(fd, &cur,
   sizeof(cur)) != sizeof(cur)) {
   perror("Child: write to pipe");
   exit(1);
  }
```

```
58      printf("PID = %ld, name = %s is awake and wrote to FD = %d \n",(long)getpid(),
        ↪  root->name, fd);
59      exit(5);
60    }
61
62    printf("%s created\n", root->name);
63    // Save the children PID's
64    pid_t pid_child[root->nr_children];
65    // and we create the same ammount of file discriptors
66    int pfd_child[2];
67    printf("Parent: Creating pipe...\n");
68    if (pipe(pfd_child) < 0) {
69      perror("pipe");
70      exit(1);
71    }
72
73    // for every child we create a pipe and give the write-end to child
74    for (i = 0; i < root->nr_children; i++) {
75      pid_child[i] = fork();
76      if (pid_child[i] == 0){
77        close(pfd_child[0]);
78        fork_procs (root->children + i, pfd_child[1]);
79        exit(1);
80      }
81    }
82    close(pfd_child[1]);
83    wait_for_ready_children(root->nr_children);
84    // we wait for leafs to send back their "names"
85    raise(SIGSTOP);
86    printf("PID = %ld, name = %s is awake\n",
87      (long)getpid(), root->name);
88    // Char type is a hack for saving ints bigger than 0-9
89    char val[2];
90    for (i = 0; i < root->nr_children; i++) {
91      // Its necesary to re-start children before reading
92      kill(pid_child[i], SIGCONT);
93      // we read from the pipe
94      if (read(pfd_child[0], &val[i], sizeof(val[i])) != sizeof(val[i])) {
95        perror("child: read from pipe");
96        exit(1);
97      }
98      printf("Parent: received value %i from the pipe. Will now compute.\n",
        ↪  val[i]);
99
100     pid_child[i] = waitpid(pid_child[i], &status, 0);
101     explain_wait_status(pid_child[i], status);
102   }
```

```
103   // Now root->name is a "special char" (int + '0') in order to transfer chars but
      ↪  with all the info
104   char computed = mycompute(*root->name, val[0], val[1]);
105   // then we write to parent using pipe
106   if (write(fd, &computed,
107    sizeof(computed)) != sizeof(computed)) {
108    perror("Child: write to pipe");
109    exit(1);
110   }
111   exit(0);
112
113  }
114
115
116  int main(int argc, char **argv) {
117
118   int pfd[2];
119   pid_t pid;
120   int status;
121   struct tree_node *root;
122   if (argc < 2) {
123    fprintf(stderr, "Usage: %s <input_tree_file>\n\n",argv[0]);
124    exit(1);
125   }
126   root = get_tree_from_file(argv[1]);
127   printf("Parent: Creating pipe...\n");
128   if (pipe(pfd) < 0) {
129    perror("pipe");
130    exit(1);
131   }
132
133   printf("Parent: Creating child...\n");
134   pid = fork();
135   if (pid < 0) {
136    perror("main:fork");
137    exit(1);
138   }
139   if (pid == 0) {
140    fork_procs(root,pfd[1]);
141    exit(1);
142   }
143   wait_for_ready_children(1);
144   show_pstree(pid);
145   kill(pid, SIGCONT);
146   pid = wait(&status);
147   explain_wait_status(pid, status);
148   printf("Parent: All done, exiting...\n");
149   return 0;
```

```
}
```

Output

```
Parent: Creating pipe...
Parent: Creating child...
PID = 19575, name +, starting...
+ created
Parent: Creating pipe...
PID = 19576, name 10, starting...
Leaf 10 created
PID = 19577, name *, starting...
My PID = 19575: Child PID = 19576 has been stopped by a signal, signo = 19
* created
Parent: Creating pipe...
PID = 19578, name +, starting...
+ created
Parent: Creating pipe...
PID = 19579, name 4, starting...
Leaf 4 created
My PID = 19577: Child PID = 19579 has been stopped by a signal, signo = 19
PID = 19580, name 5, starting...
Leaf 5 created
My PID = 19578: Child PID = 19580 has been stopped by a signal, signo = 19
PID = 19581, name 7, starting...
Leaf 7 created
My PID = 19578: Child PID = 19581 has been stopped by a signal, signo = 19
My PID = 19577: Child PID = 19578 has been stopped by a signal, signo = 19
My PID = 19575: Child PID = 19577 has been stopped by a signal, signo = 19
My PID = 19574: Child PID = 19575 has been stopped by a signal, signo = 19


+(19575)         *(19577)         +(19578)     5     (19580)
                                      7   (19581)
                          4   (19579)
            10   (19576)


PID = 19575, name = + is awake
PID = 19576, name = 10 is awake and wrote to FD = 6
Parent: received value 58 from the pipe. Will now compute.
My PID = 19575: Child PID = 19576 terminated normally, exit status = 5
PID = 19577, name = * is awake
PID = 19578, name = + is awake
PID = 19580, name = 5 is awake and wrote to FD = 8
Parent: received value 53 from the pipe. Will now compute.
My PID = 19578: Child PID = 19580 terminated normally, exit status = 5
PID = 19581, name = 7 is awake and wrote to FD = 8
Parent: received value 55 from the pipe. Will now compute.
My PID = 19578: Child PID = 19581 terminated normally, exit status = 5
I will execute 5 + 7 with result 12 !
Parent: received value 60 from the pipe. Will now compute.
My PID = 19577: Child PID = 19578 terminated normally, exit status = 0
PID = 19579, name = 4 is awake and wrote to FD = 7
Parent: received value 52 from the pipe. Will now compute.
My PID = 19577: Child PID = 19579 terminated normally, exit status = 5
I will execute 12 * 4 with result 48 !
Parent: received value 96 from the pipe. Will now compute.
My PID = 19575: Child PID = 19577 terminated normally, exit status = 0
I will execute 10 + 48 with result 58 !
```

```
My PID = 19574: Child PID = 19575 terminated normally, exit status = 0
Parent: All done, exiting...
```

1.4.1 File Descriptors

Επειδή χρησιμοποιούμε μόνο προσθέσεις και πολλαπλασιασμούς, που είναι αντιμεταθετικές πράξεις, δεν χρειάζονται παραπάνω από ένα pipe για κάθε τριάδα γονιού και δύο παιδιών. Εάν έπρεπε να εκτελέσουμε αφαιρέσεις και διαιρέσεις, που είναι πράξεις μη αντιμεταθετικές, τότε θα έπρεπε να έχουμε δύο pipes, δηλαδή ένα pipe για κάθε γονιό και παιδί, προκειμένου να γνωρίζουμε ποιος είναι ο κάθε τελεστέος που επιστρέφουν τα παιδιά διεργασίες στον πατέρα.

1.4.2 Multiprocessing

Στην περίπτωση που διαθέτουμε multicore σύστημα, θα μπορούσαμε να αναθέσουμε κάθε διεργασία "πατέρα" σε διαφορετικό πυρήνα ξεκινόντας απο το επίπεδο πάνω από τα φύλλα και κινούμενοι προς τη ρίζα. Έτσι οι πράξεις ίδιου επιπέδου αλλά διαφορετικού "κλαδιού" γίνονται παράλληλα, με καλύτερη περίπτωση (ισσοζυγισμένο AVL tree) $\log_2(t)$ όπου t ο χρόνος για να γίνουν οι πράξεις γραμμικά. Βέβαια δεν παύει στη χειρότερη περίπτωση να είναι να κάνει τόσο χρόνο όσο ο γραμμικός υπολογισμός.

2 Makefile

```
.PHONY: all clean

all: fork-example tree-example ask2-fork ask2_3 ask2_1 ask2_2 ask2_4

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

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

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

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

ask2_1: ask2_1.o proc-common.o
 $(CC) $(CFLAGS) $^ -o $@

ask2_3: ask2_3.o proc-common.o tree.o
 $(CC) $(CFLAGS) $^ -o $@

ask2_2: ask2_2.o tree.o proc-common.o
 $(CC) $(CFLAGS) $^ -o $@

ask2_4: ask2_4.o tree.o proc-common.o
 $(CC) $(CFLAGS) $^ -o $@
```

```
29
30 %.s: %.c
31  $(CC) $(CFLAGS) -S -fverbose-asm $<
32
33 %.o: %.c
34  $(CC) $(CFLAGS) -c $<
35
36 %.i: %.c
37  gcc -Wall -E $< | indent -kr > $@
38
39 clean:
40  rm -f *.o tree-example fork-example pstree-this ask2-{fork,tree,signals,pipes}
```