ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΗΛΙΑΣ ΝΤΟΝΤΟΡΟΣ
ΘΟΔΩΡΗΣ- ΙΩΑΝΝΗΣ ΚΙΤΣΗΣ

ΑΝΑΦΟΡΑ 2ης ΣΕΙΡΑΣ ΑΣΚΗΣΕΩΝ

ΑΣΚΗΣΗ 1.1

Ο κώδικας :

```
1.  #include <unistd.h>
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include <assert.h>
5.  #include <sys/types.h>
6.  #include <sys/wait.h>
7.
8.  #include "proc-common.h"
9.
10. #define SLEEP_PROC_SEC 10
11. #define SLEEP_TREE_SEC 3
12.
13. void fork_procs(void)
14. {
15.
16.     pid_t pidB, pidC, pidD;
17.     int status;
18.
19.     change_pname("A");
20.     printf("A created\n");
21.     pidB = fork();
22.     if (pidB < 0)
23.     {
24.         perror("creating B");
25.         exit(1);
26.     }
27.     if (pidB == 0)
28.     {
29.         // Node B
30.         change_pname("B");
31.         printf("B created\n");
32.         pidD = fork();
33.         if (pidD < 0)
34.         {
35.             perror("creating D");
36.             exit(1);
37.         }
38.         if (pidD == 0)
39.         {
40.             // Node D
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 (pidD > 0)
49.         {
```

```
            printf("B waiting\n");
            pidD = wait(&status);
            explain_wait_status(pidD, status);
            printf("B exiting\n");
            exit(19);
        }
    }
    if (pidB > 0)
    {
        // Node A
        pidC = fork();
        if (pidC < 0)
        {
            perror("creating C");
            exit(1);
        }
        if (pidC == 0)
        {
            // Node 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)
        {
            // Node A
            int pid_done;
            printf("A waiting\n");
            pid_done = waitpid(-1, &status, 0);
            explain_wait_status(pid_done, status);
            pid_done = waitpid(-1, &status, 0);
            explain_wait_status(pid_done, status);
            printf("A exiting\n");
            exit(16);
        }
    }
}

int main(void)
{
    pid_t pid;
    int status;

    // Create root A
    pid = fork();
    if (pid < 0)
    {
          perror("first fork");
          exit(1);
     }
     if (pid == 0)
     {
         // Child
         fork_procs();
         exit(1);
     }

     sleep(SLEEP_TREE_SEC);

     show_pstree(pid);

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

     return 0;
 }
```

Η έξοδος που παράγει :

```
1.  A created
2.  A waiting
3.  B created
4.  C created
5.  C sleeping
6.  B waiting
7.  D created
8.  D sleeping
9.
10.
11. A(2606)─┬─B(2607)──D(2609)
12.         └─C(2608)
13.
14.
15. D exiting
16. C exiting
17. My PID = 2606: Child PID = 2608 terminated normally, exit status = 17
18. My PID = 2607: Child PID = 2609 terminated normally, exit status = 13
19. B exiting
20. My PID = 2606: Child PID = 2607 terminated normally, exit status = 19
21. A exiting
22. My PID = 2605: Child PID = 2606 terminated normally, exit status = 16
```

1.1.1 Αν τερματιστεί πρόωρα η διεργασία A δίνοντας ” kill -KILL A pid”, οι διεργασίες παιδιά της θα είναι τώρα “ορφανά” και θα “υιοθετηθούν” από την init.

1.1.2 Με όρισμα τη συνάρτηση getpid(), εμφανίζεται “ολόκληρο” το δέντρο διεργασιών με ρίζα τη διεργασία ask2_1. Σο δέντρο αυτό φαίνονται οι διεργασίες sh (Standard command language interpreter), με παιδί το pstree, που καλούνται από την show_pstree.

1.1.3 Σε ένα σύστημα πολλαπλών χρηστών χωρίς όριο στο πλήθος διεργασιών, ο χρήστης θα μπορούσε να υπερφορτώσει το σύστημα με ”άπειρες” διεργασίες(με την χρήση της συνάρτησης fork()) για αυτό είναι απαραίτητος ο περιορισμός του πλήθους των διεργασιών ανά χρήστη.

ΑΣΚΗΣΗ 1.2

Ο κώδικας :

```
1.  #include <unistd.h>
2.  #include <stdlib.h>
3.  #include <stdio.h>
4.  #include <signal.h>
5.  #include <assert.h>
6.  #include <sys/wait.h>
7.  #include <fcntl.h>
8.  #include <string.h>
9.
10. #include "proc-common.h"
11. #include "tree.h"
12.
13. #define SLEEP_TREE_SEC 6
14. #define SLEEP_LEAF 10
15.
16. void forks(struct tree_node *ptr)
17. {
18.     int i = 0;
19.     int status;
20.     pid_t pid;
21.     //change process name from file
22.     change_pname(ptr->name);
23.     //create all children
24.     for (i = 0; i < ptr->nr_children; i++)
25.     {
26.         printf("i am %s and i have to create %d children\n", ptr->name, ptr->nr_children
    - i);
27.         int p = fork();
28.         //the children enter the if
29.         if (p == 0)
30.         {
31.             change_pname((ptr->children + i)->name);
32.             //the leafs enter the if
33.             if ((ptr->children + i)->nr_children == 0)
34.             {
35.                 printf("i am %s and i am going to sleep\n", (ptr->children + i)->name);
36.                 //the leafs sleep
37.                 sleep(SLEEP_LEAF);
38.                 printf("child %s woke up and exiting \n", (ptr->children + i)->name);
39.                 exit(0);
40.             }
41.             else
42.             {
43.                 forks(ptr->children + i);
44.             }
45.         }
46.     }
47.     //wait for all children to wake up and then exit
48.     for (i = 0; i < ptr->nr_children; i++)
49.     {
50.         pid = wait(&status);
51.         explain_wait_status(pid, status);
52.     }
53.     exit(0);
54. }
55.
56. int main(int argc, char **argv)
57. {
58.     struct tree_node *root;
59.     int status;
60.     id_t pid;
```

```
61.
62.     if (argc != 2)
63.     {
64.         fprintf(stderr, "Usage: ./ask2 [input file]\n");
65.         exit(1);
66.     }
67.     //opening file
68.     if (open(argv[1], O_RDONLY) < 0)
69.     {
70.         perror("opening file");
71.     }
72.     //get root from file
73.     root = get_tree_from_file(argv[1]);
74.     //create the first child
75.     pid = fork();
76.     if (pid < 0)
77.     {
78.         perror("first fork");
79.         exit(1);
80.     }
81.     //the child process enters the if
82.     if (pid == 0)
83.     {
84.         forks(root);
85.         exit(0);
86.     }
87.
88.     sleep(SLEEP_TREE_SEC);
89.     show_pstree(pid);
90.     // fro all children to wake up
91.     pid = wait(&status);
92.     explain_wait_status(pid, status);
93.     return 0;
94. }
95.
```

Η είσοδος:

```
1.  A
2.  4
3.  B
4.  C
5.  D
6.  H
7.
8.  B
9.  2
10. E
11. F
12.
13. E
14. 1
15. G
16.
17. G
18. 0
19.
20. F
21. 0
22.
23. C
24. 0
25.
26. D
27. 0
28.
```

```
29. H
30. 0
```

Η έξοδος που παράγει :

```
1.  i am A and i have to create 4 children
2.  i am A and i have to create 3 children
3.  i am A and i have to create 2 children
4.  i am C and i am going to sleep
5.  i am A and i have to create 1 children
6.  i am D and i am going to sleep
7.  i am H and i am going to sleep
8.  i am B and i have to create 2 children
9.  i am B and i have to create 1 children
10. i am F and i am going to sleep
11. i am E and i have to create 1 children
12. i am G and i am going to sleep
13.
14.
15. A(2972)─┬─B(2973)─┬─E(2977)──G(2979)
16.         │         └─F(2978)
17.         ├─C(2974)
18.         ├─D(2975)
19.         └─H(2976)
20.
21.
22. child C woke up and exiting
23. child D woke up and exiting
24. child H woke up and exiting
25. child F woke up and exiting
26. My PID = 2972: Child PID = 2974 terminated normally, exit status = 0
27. My PID = 2972: Child PID = 2975 terminated normally, exit status = 0
28. My PID = 2972: Child PID = 2976 terminated normally, exit status = 0
29. My PID = 2973: Child PID = 2978 terminated normally, exit status = 0
30. child G woke up and exiting
31. My PID = 2977: Child PID = 2979 terminated normally, exit status = 0
32. My PID = 2973: Child PID = 2977 terminated normally, exit status = 0
33. My PID = 2972: Child PID = 2973 terminated normally, exit status = 0
34. My PID = 2971: Child PID = 2972 terminated normally, exit status = 0
```

1.2.1 Τα μηνύματα έναρξης εμφανίζονται με breadth first traversing και τα μηνύματα τερματισμού αντίστροφα. Η σειρά έναρξης και καταστροφής τους είναι τυχαία . Δηλαδή ο κάθε πατέρας δημιουργεί τα παιδιά του και περιμένει αυτά να τερματίσουν. Τα παιδία με τη σειρά τους κάνουν το ίδιο μέχρι να φτάσουμε σε ένα φύλλο που μπαίνει σε sleep() μμέχρι να τελειώσει η κατασκευή ολόκληρου του δέντρου .Στη συνέχεια τα φύλλα ξεκινούν και τερματίζουν και το δέντρο αποδομείται αντίστροφα.

ΑΣΚΗΣΗ 1.3

Ο κώδικας :

```
1.  #include <unistd.h>
2.  #include <stdlib.h>
3.  #include <stdio.h>
4.  #include <signal.h>
5.  #include <assert.h>
6.  #include <sys/wait.h>
7.  #include <fcntl.h>
8.  #include <string.h>
9.  #include "proc-common.h"
10. #include "tree.h"
11.
12. void fork_procs(struct tree_node *ptr)
13. {
14.     int i = 0;
15.     int status;
16.     pid_t childPid[ptr->nr_children];
17.     pid_t pid;
18.     // change name of the process from file
19.     change_pname(ptr->name);
20.     // create children
21.     for (i = 0; i < ptr->nr_children; i++)
22.     {
23.         childPid[i] = fork();
24.         // the children go in the if
25.         if (childPid[i] == 0)
26.         {
27.             change_pname((ptr->children + i)->name);
28.             // if it is a leaf it goes in the if
29.             if ((ptr->children + i)->nr_children == 0)
30.             {
31.                 // the leafs stop
32.                 raise(SIGSTOP);
33.                 printf("%s\n", (ptr->children + i)->name);
34.                 exit(0);
35.             }
36.             else
37.             {
38.                 // if it is a parent it goes in the if to create all the children
39.                 fork_procs(ptr->children + i);
40.             }
41.         }
42.     }
43.
44.     raise(SIGSTOP);
45.     // wake up all the kids
46.     i = 0;
47.     for (i = 0; i < ptr->nr_children; i++)
48.     {
49.         kill(childPid[i], SIGCONT);
50.         // wait for the kids to finish
51.         pid = wait(&status);
52.         explain_wait_status(pid, status);
53.     }
54.
55.     printf("%s\n", ptr->name);
56.
57.     exit(0);
58. }
59.
60. int main(int argc, char **argv)
61. {
```

```
62.     struct tree_node *root;
63.     pid_t pid;
64.     int status;
65.
66.     if (argc != 2)
67.     {
68.         fprintf(stderr, "Usage: ./ask3 [input file]\n");
69.         exit(1);
70.     }
71.
72.     if (open(argv[1], O_RDONLY) < 0)
73.     {
74.         perror("Error opening");
75.     }
76.
77.     root = get_tree_from_file(argv[1]);
78.     // create root
79.     pid = fork();
80.     if (pid < 0)
81.     {
82.         perror("Error first fork");
83.         exit(2);
84.     }
85.     // the child process goes in the if
86.     if (pid == 0)
87.     {
88.         fork_procs(root);
89.         exit(0);
90.     }
91.     // wait for all children to stop
92.     wait_for_ready_children(1);
93.     show_pstree(pid);
94.     // send continue signal to root
95.     kill(pid, SIGCONT);
96.
97.     pid = wait(&status);
98.     explain_wait_status(pid, status);
99.     return 0;
100.  }
```

Η είσοδος :

```
1.  A
2.  3
3.  B
4.  C
5.  D
6.
7.  B
8.  2
9.  E
10. F
11.
12. E
13. 1
14. G
15.
16. G
17. 0
18.
19. F
20. 0
21.
22. C
23. 0
24.
```

```
25. D
26. 1
27. H
28.
29. H
30. 0
```

Η έξοδος που παράγει :

```
1.  My PID = 3223: Child PID = 3224 has been stopped by a signal, signo = 19
2.
3.
4.  A(3224)─┬─B(3225)─┬─E(3228)──G(3232)
5.          │         └─F(3229)
6.          ├─C(3226)
7.          └─D(3227)──H(3230)
8.
9.
10. G
11. My PID = 3228: Child PID = 3232 terminated normally, exit status = 0
12. E
13. My PID = 3225: Child PID = 3228 terminated normally, exit status = 0
14. F
15. My PID = 3225: Child PID = 3229 terminated normally, exit status = 0
16. B
17. My PID = 3224: Child PID = 3225 terminated normally, exit status = 0
18. C
19. My PID = 3224: Child PID = 3226 terminated normally, exit status = 0
20. H
21. My PID = 3227: Child PID = 3230 terminated normally, exit status = 0
22. D
23. My PID = 3224: Child PID = 3227 terminated normally, exit status = 0
24. A
25. My PID = 3223: Child PID = 3224 terminated normally, exit status = 0
```

1.3.1 Με την χρήση σημάτων γλιτώνουμε χρόνο αφού δεν περιμένουμε λόγω της sleep() .Επιπλέον ,με τα σήματα πετυχαίνουμε συγχρονισμό και έτσι η διαδικασία διάσχισης του δέντρου είναι αυτή τη φορά ντετερμινιστική.

1.3.2 Η wait_for_ready_children() περιμένει μμέχρι ο αριθμός παιδιών που παίρνει ως όρισμα να αλλάξει κατάσταση. Η χρήση της μέσα στο for loop εξασφαλίζει την κατά βάθος (DFS) σειρά εμφάνισης των μηνυμάτων. Με την παράλειψη της δεν θα υπήρχε συγχρονισμός και η σειρά δημιουργία των διεργασιών του δέντρου θα ήταν τυχαία.

ΑΣΚΗΣΗ 1.4

Ο κώδικας :

```
1.  #include <unistd.h>
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include <assert.h>
5.  #include <sys/types.h>
6.  #include <sys/wait.h>
7.  #include <signal.h>
8.
9.  #include "tree.h"
10. #include "proc-common.h"
11.
12. #define SLEEP_TREE_SEC 1
13. #define SLEEP_LEAF 1
14. // function to calculate the results from the values sent from children
15. char calculate(char kind, char a, char b)
16. {
17.     int a_int = a - '0', b_int = b - '0';
18.     switch (kind)
19.     {
20.     case '+':
21.     {
22.         printf("result: %i\n", a_int + b_int);
23.         return a_int + b_int + '0';
24.     }
25.     case '*':
26.     {
27.         printf("result: %i\n", a_int * b_int);
28.         return a_int * b_int + '0';
29.     }
30.     }
31.     return -1;
32. }
33.
34. void fork_procs(struct tree_node *root, int fd)
35. {
36.     int i;
37.
38.
39.     printf("PID = %ld, name %s, starting\n",
40.            (long)getpid(), root->name);
41.
42.     change_pname(root->name);
43.     // leafs enter the if
44.     if (root->nr_children == 0)
45.     {
46.         printf("leaf %s created\n", root->name);
47.         // convert the name of the node to write it to the pipe
48.         sleep(SLEEP_LEAF);
49.         char cur = atoi(root->name) + '0';
50.         if (write(fd, &cur, sizeof(cur)) != sizeof(cur))
51.         {
52.             perror("Child: write to pipe");
53.             exit(1);
54.         }
55.         printf("PID = %ld, name = %s is awake and wrote to fd \n", (long)getpid(), root-
    >name);
56.         exit(10);
57.     }
58.
59.     printf("%s created\n", root->name);
60.     // save children pid and create the pipes between parents and children
```

```
61.     pid_t pid_child[root->nr_children];
62.     int pfd_child[2];
63.     printf("parent creating pipe...\n");
64.     if (pipe(pfd_child) < 0)
65.     {
66.         perror("pipe");
67.         exit(1);
68.     }
69.
70.     // create different pipes for each child and give the child the write end of the
    pipe
71.     for (i = 0; i < root->nr_children; i++)
72.     {
73.         pid_child[i] = fork();
74.         if (pid_child[i] == 0)
75.         {
76.             // close the read end of the pipe in the children processes
77.             close(pfd_child[0]);
78.             fork_procs(root->children + i, pfd_child[1]);
79.             exit(0);
80.         }
81.     }
82.     // close write end of the pipe of the parent processes
83.     close(pfd_child[1]);
84.
85.     // stop until child sends their name
86.
87.     printf("PID = %ld, name = %s is awake\n",(long)getpid(), root->name);
88.     char val[2];
89.     for (i = 0; i < root->nr_children; i++)
90.     {
91.
92.         // read from the pipe
93.         if (read(pfd_child[0], &val[i], sizeof(val[i])) != sizeof(val[i]))
94.         {
95.             perror("child: read from pipe");
96.             exit(1);
97.         }
98.         printf("parent received value from the pipe\n");
99.
100.      }
101.
102.     char result = calculate(*root->name, val[0], val[1]);
103.     // send the parent result we have from the calculate function
104.     if (write(fd, &result, sizeof(result)) != sizeof(result))
105.     {
106.
107.         perror("Child: write to pipe");
108.         exit(1);
109.     }
110.     exit(0);
111. }
112.
113. int main(int argc, char **argv)
114. {
115.     int pfd[2];
116.     pid_t pid;
117.
118.     struct tree_node *root;
119.     if (argc != 2)
120.     {
121.         fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
122.         exit(1);
123.     }
124.     root = get_tree_from_file(argv[1]);
125.
126.     printf("Parent: Creating pipe...\n");
127.     // create first pipe
128.     if (pipe(pfd) < 0)
129.     {
```

```
130.          perror("pipe");
131.          exit(1);
132.      }
133.      // create first child process
134.      pid = fork();
135.
136.      if (pid < 0)
137.      {
138.          perror("main:fork");
139.          exit(1);
140.      }
141.
142.      if (pid == 0)
143.      {
144.          fork_procs(root, pfd[1]);
145.
146.          exit(1);
147.      }
148.
149.      // display the process tree
150.      show_pstree(pid);
151.      // parent sleeps so that the children finish first
152.      sleep(SLEEP_TREE_SEC);
153.
154.      printf("parent process done\n");
155.      return 0;
156.  }
157.
```

Η είσοδος:

```
1.  +
2.  2
3.  +
4.  10
5.
6.  +
7.  2
8.  5
9.  6
10.
11. 5
12. 0
13.
14. 6
15. 0
16.
17. 10
18. 0
```

Η έξοδος που παράγει :

```
1.  Parent: Creating pipe...
2.  PID = 3304, name +, starting
3.  + created
4.  parent creating pipe...
5.  PID = 3306, name +, starting
6.  PID = 3304, name = + is awake
7.  + created
```

```
parent creating pipe...
PID = 3307, name 10, starting
leaf 10 created
PID = 3306, name = + is awake
PID = 3308, name 5, starting
leaf 5 created
PID = 3309, name 6, starting
leaf 6 created


+(3304)─┬─+(3306)─┬─5(3308)
        │         └─6(3309)
        └─10(3307)


PID = 3307, name = 10 is awake and wrote to fd
parent received value from the pipe
PID = 3308, name = 5 is awake and wrote to fd
parent received value from the pipe
PID = 3309, name = 6 is awake and wrote to fd
parent received value from the pipe
result: 11
parent received value from the pipe
result: 21
parent process done
```

1.4.1 Στη συγκεκριμένη άσκηση χρειάζεται ένα pipe για κάθε τριάδα γονιού και δύο παιδιών, και τούτο διότι η πρόσθεση και ο πολλαπλασιασμός είναι αντιμεταθετικές πράξεις. Όσον αφορά αφαιρέσεις και διαιρέσεις τότε θα έπρεπε να έχουμε ένα pipe για κάθε γονιό και ένα για κάθε παιδί, για να ξέρουμε ποιος είναι ο κάθε τελεστέος που επιστρέφουν τα παιδιά στον πατέρα.

1.4.2 Σε ένα σύστημα πολλαπλών επεξεργαστών, κάθε διεργασία πατέρα θα μπορούσε να “βρίσκεται” σε διαφορετικό πυρήνα ξεκινώντας από το επίπεδο πάνω από τα φύλλα πηγαίνοντας στη ρίζα. Έτσι οι πράξεις στο ίδιο επίπεδο αλλά σε διαφορετικό κλαδί γίνονται παράλληλα με καλύτερη περίπτωση log(t) και χειρότερη την γραμμική.