Λειτουργικά Συστήματα – Άσκηση 2η

ΑΓΓΕΛΟΣ ΣΤΑΗΣ 03117435, ΣΩΚΡΑΤΗΣ ΠΟΥΤΑΣ 03117054 (Ομάδα Oslabb08)

Άσκηση 1.1 – Δημιουργία δεδομένου δέντρου διεργασιών

Κώδικας

```
#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

void child_D(void){
	change_pname("D");
	printf("D starting\n");
	printf("D: Sleeping...\n");
	sleep(SLEEP_PROC_SEC);
	printf("D: Exiting...\n");
	exit(13);
}
void child_C(void){
	change_pname("C");
	printf("C starting\n");
	printf("C: Sleeping...\n");
	sleep(SLEEP_PROC_SEC);

	printf("C: Exiting...\n");
	exit(17);
}
void child_B(void){
	printf("B starting\n");
	change_pname("B");

	int statusD;
	pid_t pD = fork();
	if(pD < 0){
			perror("B: fork");
			exit(1);
	}
	else if(pD == 0){
		child_D();
```

```
	}

	printf("B waiting for children to terminate\n");

	pD = wait(&statusD);
	explain_wait_status(pD, statusD);

	printf("B: Exiting...\n");
	exit(19);
}
void fork_procs(void)
{
	/*
	* initial process is A.
	*/
	printf("A starting\n");
	change_pname("A");
	int statusB, statusC;
	pid_t pB, pC;
	pC = fork();
	if(pC < 0){
			perror("A: fork");
			exit(1);
	}
	else if(pC == 0){
		child_C();
	}
	pB = fork();
	if(pB < 0){
			perror("A: fork");
			exit(1);
		}
	else if(pB == 0){
		child_B();
	}
	printf("A waiting for children to terminate\n");
	pB = wait(&statusB);
	pC = wait(&statusC);
	explain_wait_status(pB, statusB);
	explain_wait_status(pC, statusC);
	printf("A: Exiting...\n");
	exit(16);
}
int main(void)
{
	pid_t pid;
	int status;

	/* Fork root of process tree */
```

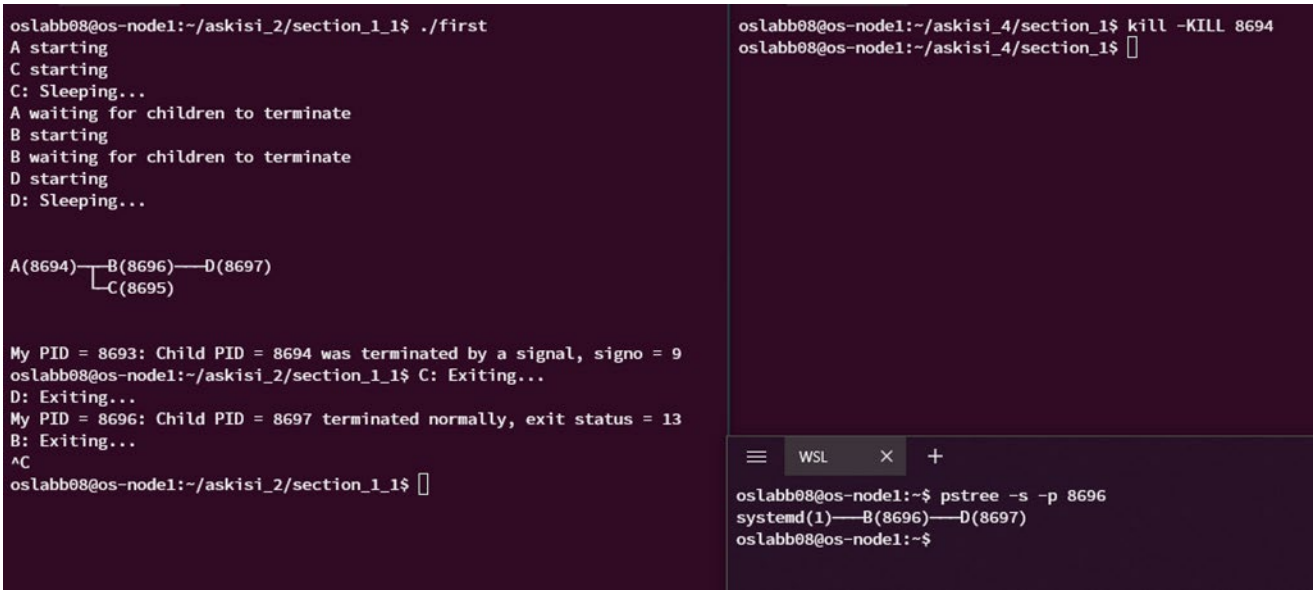
```
        pid = fork();
        if (pid < 0) {
                    perror("main: fork");
                    exit(1);
        }
        if (pid == 0) {
                    /* Child */
                    fork_procs();
                    exit(1);
        }
        /*
        * Father
        */
        sleep(SLEEP_TREE_SEC);
        /* Print the process tree root at pid */
        show_pstree(pid);
        /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);
        return 0;
}
```

Έξοδος Εκτέλεσης για proc.tree

```
A starting
A waiting for children to terminate
B starting
B waiting for children to terminate
C starting
C: Sleeping...
D starting
D: Sleeping...

A(4915)─┬─B(4917)──D(4918)
        └─C(4916)

C: Exiting...
D: Exiting...
My PID = 4917: Child PID = 4918 terminated normally, exit status = 13
B: Exiting...
My PID = 4915: Child PID = 4916 terminated normally, exit status = 17
My PID = 4915: Child PID = 4917 terminated normally, exit status = 19
A: Exiting...
My PID = 4914: Child PID = 4915 terminated normally, exit status = 16
```

Ερωτήσεις

1. Όταν η διεργασία Α τερματίζεται πρόωρα (πριν τερματιστούν τα παιδιά της) τότε τα ορφανά πλέον παιδιά αποκτούν το PPID μιας ειδικής διεργασίας συστήματος. Παραδοσιακά, αυτή η διεργασία είναι η init με pid = 1. Παρατίθεται και το αντίστοιχο στιγμιότυπο εκτέλεσης.



2. Αντικαθιστώντας την εντολή **show_pstree(pid)** με **show_pstree(getpid())** το αποτέλεσμα είναι να τυπώνεται το δέντρο διεργασιών με ρίζα τον γονέα της διεργασίας Α, επειδή εντός αυτής της διεργασίας εκτελείται η εντολή **getpid()** ενώ στο δέντρο διεργασιών φαίνεται και η δημιουργία της διεργασίας pstree.

Έξοδος Εκτέλεσης

```
A: starting
A: waiting for children to terminate
B: starting
B: waiting for children to terminate
C: starting
C: Sleeping...
D: starting
D: Sleeping...
first(16324)─┬─A(16325)─┬─B(16327)──D(16328)
             │          └─C(16326)
             └─sh(16329)──pstree(16330)
C: Exiting...
D: Exiting...
My PID = 16327: Child PID = 16328 terminated normally, exit status = 13
B: Exiting...
My PID = 16325: Child PID = 16326 terminated normally, exit status = 17
```

```
My PID = 16325: Child PID = 16327 terminated normally, exit status = 19
A: Exiting...
My PID = 16324: Child PID = 16325 terminated normally, exit status = 16
```

3. Ο περιορισμός του πλήθους των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης είναι πιθανό να γίνεται για λόγους δίκαιης κατανομής των πόρων του συστήματος σε όλους τους χρήστες ώστε να μην μπορεί κάποιος χρήστης να μονοπωλεί τον χρόνο του επεξεργαστή δημιουργώντας πάρα πολλές διεργασίες και καθυστερώντας τους υπόλοιπους χρήστες. Επιπρόσθετα επειδή ο χρόνος του επεξεργαστή ισομοιράζεται σε πολλές διεργασίες αν ήταν δυνατό κάθε χρήστης να δημιουργήσει πάρα πολλές η κάθε μια θα περιοριζόταν σε ένα υπερβολικά μικρό χρονικό παράθυρο εκτέλεσης με αποτέλεσμα να μην ολοκληρώνουν τη λειτουργία τους. Τέλος ένας ακόμη λόγος θα ήταν για προστασία από κακόβουλο λογισμικό που προσπαθεί να δημιουργήσει πάρα πολλές διεργασίες προκαλώντας κατάρρευση του συστήματος.

Άσκηση 1.2 - Δημιουργία αυθαίρετου δέντρου διεργασιών

Κώδικας

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

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3
typedef struct tree_node * tree_node_ptr;

void fork_process(tree_node_ptr ptr)
{
      printf("%s: Created...\n",ptr->name);
      change_pname(ptr->name); //Allazei to onoma tis diergasias
      if((ptr->nr_children)==0)
    {
        printf("%s: Sleeping...\n",ptr->name);
        sleep(SLEEP_PROC_SEC);
        printf("%s: Exiting...\n",ptr->name);
        exit(0);
    }
    //Periptosi opou exei paidia
    pid_t p[(ptr->nr_children)+1]; //Pinakas me ta pid ton paidion tou ekastote
komvou
```

```
    int status;
    for(int i=1;i<=ptr->nr_children;i++)
        {
            p[i]=fork();
            if(p[i] < 0)
            {
                perror("fork");
                exit(0);
            }
            else if(p[i] == 0)
                fork_process(ptr->children+i-1);
        }
    printf("%s: Waiting for children to terminate...\n",ptr->name);
    for(int i=1;i<=ptr->nr_children;i++)
        {
            p[i] = wait(&status);
            explain_wait_status(p[i], status);
        }
      printf("%s: Exiting...\n",ptr->name);
      exit(0);
}

int main(int argc, char *argv[])
{
      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]); //Pairnei diekti sto root tou dentrou
      print_tree(root); //Typonoume to dentro gia na doume an to bgazei sosta sto
telos
    pid_t pid;
      int status;
    if (root==NULL)
    {
        printf("Tree is empty. Exiting...\n");
        return(0);
    }
      /* Fork root of process tree */
      pid = fork();
      if (pid < 0)
    {
            perror("main: fork");
            exit(1);
      }
      if (pid == 0)
    {
```

```
            /* Child */
            fork_process(root);
            exit(1);
      }
      /*Father */
      sleep(SLEEP_TREE_SEC);
      /* Print the process tree root at pid */
      show_pstree(pid);
      /* Wait for the root of the process tree to terminate */
      pid = wait(&status);
      explain_wait_status(pid, status);
      return 0;
}
```

Έξοδος Εκτέλεσης για proc.tree

Παρατίθενται 2 από τις πιθανές εξόδους του προγράμματος όπου η σειρά εμφάνισης του μηνύματος B: Waiting for children to terminate... και των μηνυμάτων δημιουργίας και ύπνου των κόμβων E και F αλλάζει αναλόγως με την εκτέλεση.

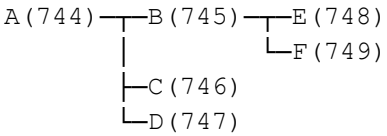
Έξοδος A

```
A
      B
            E
            F
      C
      D
A: Created...
B: Created...
A: Waiting for children to terminate...
C: Created...
C: Sleeping...
D: Created...
D: Sleeping...
E: Created...
E: Sleeping...
B: Waiting for children to terminate...
F: Created...
F: Sleeping...

A(729)─┬─B(730)─┬─E(733)
       │        └─F(734)
       ├─C(731)
       └─D(732)

C: Exiting...
D: Exiting...
```

```
E: Exiting...
F: Exiting...
My PID = 729: Child PID = 731 terminated normally, exit status = 0
My PID = 729: Child PID = 732 terminated normally, exit status = 0
My PID = 730: Child PID = 733 terminated normally, exit status = 0
My PID = 730: Child PID = 734 terminated normally, exit status = 0
B: Exiting...
My PID = 729: Child PID = 730 terminated normally, exit status = 0
A: Exiting...
My PID = 728: Child PID = 729 terminated normally, exit status = 0
```

Έξοδος Β

```
A
      B
            E
            F
      C
      D
A: Created...
B: Created...
A: Waiting for children to terminate...
C: Created...
C: Sleeping...
D: Created...
D: Sleeping...
B: Waiting for children to terminate...
E: Created...
E: Sleeping...
F: Created...
F: Sleeping...
```




```
C: Exiting...
D: Exiting...
E: Exiting...
F: Exiting...
My PID = 744: Child PID = 746 terminated normally, exit status = 0
My PID = 744: Child PID = 747 terminated normally, exit status = 0
My PID = 745: Child PID = 748 terminated normally, exit status = 0
My PID = 745: Child PID = 749 terminated normally, exit status = 0
B: Exiting...
My PID = 744: Child PID = 745 terminated normally, exit status = 0
A: Exiting...
My PID = 743: Child PID = 744 terminated normally, exit status = 0
```

Ερωτήσεις

1. Τα μηνύματα έναρξης εμφανίζονται με σειρά διάσχισης κατά πλάτος δηλαδή για την προηγούμενη έξοδο A,B,C,D,E,F. Αυτό συμβαίνει καθώς κατά αυτή τη σειρά δημιουργούνται οι αντίστοιχες διεργασίες από το πρόγραμμα.
   Τα μηνύματα τερματισμού εμφανίζονται αρχικά για τα φύλλα με αυτά που βρίσκονται σε μικρότερο επίπεδο σε σχέση με τη ρίζα να προηγούνται και στη συνέχεια για τους υπόλοιπους κόμβους με αυτούς που βρίσκονται σε μεγαλύτερο επίπεδο να προηγούνται. Συγκεκριμένα για τη προηγούμενη έξοδο η σειρά είναι C,D,E,F,B,A. Αυτό συμβαίνει καθώς τα φύλλα που βρίσκονται σε μικρότερο επίπεδο «κοιμήθηκαν» νωρίτερα άρα θα ξυπνήσουν και θα κάνουν έξοδο νωρίτερα, ενώ οι υπόλοιποι κόμβοι περιμένουν κατά σειρά τα φύλλα-παιδιά τους να τερματίσουν με αυτούς που είναι σε μεγαλύτερο επίπεδο εύλογα να τερματίζουν πρώτοι.

Άσκηση 1.3 - Αποστολή και χειρισμός σημάτων

Κώδικας

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

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3
typedef struct tree_node * tree_node_ptr;

void fork_process(tree_node_ptr ptr)
{
    printf("PID = %ld, name %s, starting...\n",
                (long)getpid(), ptr->name);
      change_pname(ptr->name); //Allazei to onoma tis diergasias
      if((ptr->nr_children)==0)
    {
            printf("PID = %ld, name = %s is sleeping\n",(long)getpid(),ptr->name);
            raise(SIGSTOP); //Anastellei ti leitourgia tis
            printf("PID = %ld, name = %s is awake\n",(long)getpid(),ptr->name);
            printf("PID = %ld, name = %s exiting\n",(long)getpid(),ptr->name);
            exit(0);
```

```
      }
    //Periptosi opou exei paidia
    pid_t p[(ptr->nr_children)+1]; //Pinakas pid olon ton paidion tou komvou
    int status;
    //Dimiourgia paidion
    for(int i=1;i<=ptr->nr_children;i++)
        {
            p[i]=fork();
            if(p[i] < 0)
            {
                perror("fork");
                exit(0);
            }
            else if(p[i] == 0)
                fork_process(ptr->children+i-1);
        }
      wait_for_ready_children(ptr->nr_children); //Perimenei na anastiloun ti
      //leitourgia tous ola ta paidia
      printf("PID = %ld, name = %s is sleeping\n",(long)getpid(),ptr->name);
      raise(SIGSTOP); //Anastellei ti leitourgia tis
      if (signal(SIGCONT,sighandler)<0)
      {
            printf("Could not establish sighandler");
            exit(1);
      }
      printf("PID = %ld, name = %s is awake\n",(long)getpid(),ptr->name);
      for(int i=1;i<=ptr->nr_children;i++)
    {
            kill(p[i], SIGCONT);
        p[i] = wait(&status);
        explain_wait_status(p[i], status);
    }
      printf("%s: Exiting...\n",ptr->name);
      exit(0);
}

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_process(root);
		exit(1);
	}
	wait_for_ready_children(1);
	/* 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);
	return 0;
}
```

Έξοδος Εκτέλεσης για proc.tree

```
PID = 5502, name A, starting...
PID = 5503, name B, starting...
PID = 5504, name C, starting...
PID = 5504, name = C is sleeping
My PID = 5502: Child PID = 5504 has been stopped by a signal, signo = 19
PID = 5505, name D, starting...
PID = 5505, name = D is sleeping
My PID = 5502: Child PID = 5505 has been stopped by a signal, signo = 19
PID = 5506, name E, starting...
PID = 5506, name = E is sleeping
PID = 5507, name F, starting...
My PID = 5503: Child PID = 5506 has been stopped by a signal, signo = 19
PID = 5507, name = F is sleeping
My PID = 5503: Child PID = 5507 has been stopped by a signal, signo = 19
PID = 5503, name = B is sleeping
My PID = 5502: Child PID = 5503 has been stopped by a signal, signo = 19
PID = 5502, name = A is sleeping
My PID = 5501: Child PID = 5502 has been stopped by a signal, signo = 19

A(5502)─┬─B(5503)─┬─E(5506)
        │         └─F(5507)
        ├─C(5504)
        └─D(5505)
```

```
PID = 5502, name = A is awake
PID = 5503, name = B is awake
PID = 5506, name = E is awake
PID = 5506, name = E exiting
My PID = 5503: Child PID = 5506 terminated normally, exit status = 0
PID = 5507, name = F is awake
PID = 5507, name = F exiting
My PID = 5503: Child PID = 5507 terminated normally, exit status = 0
B: Exiting...
My PID = 5502: Child PID = 5503 terminated normally, exit status = 0
PID = 5504, name = C is awake
PID = 5504, name = C exiting
My PID = 5502: Child PID = 5504 terminated normally, exit status = 0
PID = 5505, name = D is awake
PID = 5505, name = D exiting
My PID = 5502: Child PID = 5505 terminated normally, exit status = 0
A: Exiting...
My PID = 5501: Child PID = 5502 terminated normally, exit status = 0
```

Ερωτήσεις

1. Η χρήση σημάτων σε σχέση με τη sleep εξασφαλίζει έναν πιο αποτελεσματικό και ντετερμινιστικό έλεγχο του προγραμματιστή στην αναστολή λειτουργίας και το τερματισμό των διεργασιών με την επιθυμητή σειρά και στην επιθυμητή χρονική στιγμή για τη κάθε μία.
2. Η λειτουργία της wait_for_ready_children είναι να περιμένει να αναστείλουν τη λειτουργία τους όλες οι διεργασίες-παιδιά της διεργασίας που την τρέχει λαμβάνοντας όλα τα σήματα SIGCHLD αλλαγής κατάστασης για καθένα από τα παιδία της μέσω της χρήσης της waitpid και τυπώνοντας αντίστοιχο μήνυμα.
   Χρησιμοποιώντας τη wait_for_ready_children εξασφαλίζεται ότι μια διεργασία δε θα αναστείλει τη λειτουργία της πριν αναστείλουν τη λειτουργία τους τα παιδιά της. Συνεπώς όταν έχουμε φτάσει στην αρχική διεργασία θα έχει εκτελεστεί με τη σωστή σειρά η αναδρομική διαδικασία αναστολής λειτουργίας όλων των διεργασιών.
   Παραλείποντας τη είναι πιθανό να μη ακολουθηθεί η παραπάνω αναδρομική διαδικασία καθώς μόλις μια διεργασία δημιουργεί τα παιδιά της θα αναστέλλει κατευθείαν τη λειτουργία της, άρα δε μπορούμε ντετερμινιστικά να γνωρίζουμε τη σειρά αναστολής λειτουργίας των διεργασιών του δέντρου. Επίσης ενδεχομένως να μην έχει προλάβει να δημιουργηθεί πλήρως το δέντρο διεργασιών αφού η κύρια διεργασία αμέσως μόλις δημιουργήσει τη ρίζα θα εκτελεί την show_pstree, με συνέπεια τη λανθασμένη εκτύπωση του. Τέλος αν το δέντρο διεργασιών είναι αρκετά μεγάλο πιθανώς όταν οι διεργασίες κοντά στη ρίζα λαμβάνουν σήμα «αφύπνισης» οι διαδικασίες σε μεγαλύτερο επίπεδο να μην έχουν προλάβει να αναστείλουν τη λειτουργία τους.

Άσκηση 1.4 - Παράλληλος Υπολογισμός Αριθμητικής Έκφρασης

Κώδικας

```
#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"

int computation(int a, int b, char op)
{
	if(op == '+') return a + b;
	else if(op == '*') return a * b;
}
void fork_procs(struct tree_node *ptr, int pipe_to_parent)
{
	/*
	* Start
	*/
	printf("PID = %ld, name %s, starting...\n",
	                    (long)getpid(), ptr->name);
	change_pname(ptr->name);

	if((ptr->nr_children)==0)
	{
			//to fyllo koimatai
			raise(SIGSTOP);

			//to fyllo jypna

			int res = atoi(ptr->name);

			if (write(pipe_to_parent, &res, sizeof(res)) != sizeof(res)) {
			perror("leaf: write to pipe");
			exit(1);
			}

			printf("%s: Exiting...\n",ptr->name);
			exit(res);
	}
	//Periptosi opou exei paidia
	int pipe_to_child[2];
	if (pipe(pipe_to_child) < 0) {
			perror("pipe");
			exit(1);
	}
	pid_t p[(ptr->nr_children)+1]; //Pinakas pou krataei ta pid olon ton paidion
tou ekastote komvou
```

```
	int status;
	p[1]=fork();
	if(p[1] < 0)
	{
		perror("fork");
		exit(0);
	}
	else if(p[1] == 0)
		fork_procs(ptr->children, pipe_to_child[1]); //children take pipe[1] to
write

	p[2]=fork();
	if(p[2] < 0)
	{
		perror("fork");
		exit(0);
	}
	else if(p[2] == 0)
		fork_procs(ptr->children+1, pipe_to_child[1]); //children take pipe[1]
to write
	wait_for_ready_children(ptr->nr_children);
	/*
	* Suspend Self
	*/
	raise(SIGSTOP);

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

	//parent sets children in motion one-by-one
	int a, b;
	//child 1
	kill(p[1], SIGCONT);
	wait(&status);
	explain_wait_status(p[1], status);
	if (read(pipe_to_child[0], &a, sizeof(a)) != sizeof(a)) {
			perror("parent: read from pipe");
			exit(1);
	}
	//child 2
	kill(p[2], SIGCONT);
	wait(&status);
	explain_wait_status(p[2], status);
	if (read(pipe_to_child[0], &b, sizeof(b)) != sizeof(b)) {
			perror("parent: read from pipe");
			exit(1);
	}
	int ret = computation(a, b, *(ptr->name));

	if (write(pipe_to_parent, &ret, sizeof(ret)) != sizeof(ret)) {
			perror("node: write to pipe");
			exit(1);
	}
	printf("%s: Exiting...\n",ptr->name);
	/*
	* Exit
```

```
	*/
	exit(ret);
}
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]);
	int pipe_root[2];
	printf("root: Creating pipe...\n");
	if (pipe(pipe_root) < 0) {
			perror("pipe");
			exit(1);
	}
	/* Fork root of process tree */
	pid = fork();
	if (pid < 0) {
			perror("main: fork");
			exit(1);
	}
	if (pid == 0) {
			/* Child */
			fork_procs(root, pipe_root[1]);
			exit(1);
	}
	/* for ask2-signals */
	wait_for_ready_children(1);
	/* 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);
	int final;
	if (read(pipe_root[0], &final, sizeof(final)) != sizeof(final)) {
			perror("root: read from pipe");
			exit(1);
	}
	printf("final result = %d\n", final);

	return 0;
}
```

Έξοδος Εκτέλεσης για expr.tree

```
	root: Creating pipe...
```

```
PID = 3059, name +, starting...
PID = 3061, name *, starting...
PID = 3060, name 10, starting...
My PID = 3059: Child PID = 3060 has been stopped by a signal, signo = 19
PID = 3063, name 4, starting...
My PID = 3061: Child PID = 3063 has been stopped by a signal, signo = 19
PID = 3062, name +, starting...
PID = 3065, name 7, starting...
My PID = 3062: Child PID = 3065 has been stopped by a signal, signo = 19
PID = 3064, name 5, starting...
My PID = 3062: Child PID = 3064 has been stopped by a signal, signo = 19
My PID = 3061: Child PID = 3062 has been stopped by a signal, signo = 19
My PID = 3059: Child PID = 3061 has been stopped by a signal, signo = 19
My PID = 3058: Child PID = 3059 has been stopped by a signal, signo = 19
+(3059)─┬─*(3061)─┬─+(3062)─┬─5(3064)
        │         │         └─7(3065)
        │         └─4(3063)
        └─10(3060)
PID = 3059, name = + is awake
10: Exiting...
My PID = 3059: Child PID = 3060 terminated normally, exit status = 10
PID = 3061, name = * is awake
PID = 3062, name = + is awake
5: Exiting...
My PID = 3062: Child PID = 3064 terminated normally, exit status = 5
7: Exiting...
My PID = 3062: Child PID = 3065 terminated normally, exit status = 7
+: Exiting...
My PID = 3061: Child PID = 3062 terminated normally, exit status = 12
4: Exiting...
My PID = 3061: Child PID = 3063 terminated normally, exit status = 4
*: Exiting...
My PID = 3059: Child PID = 3061 terminated normally, exit status = 48
+: Exiting...
My PID = 3058: Child PID = 3059 terminated normally, exit status = 58
final result = 58
```

Ερωτήσεις

1. Όπως φαίνεται και στον παραπάνω κώδικα, μπορεί να χρησιμοποιηθεί μόνο μία σωλήνωση ανά γονική διεργασία (κόμβοι του δέντρου που πραγματοποιούν μία αριθμητική πράξη). Αυτό που συμβαίνει είναι ότι τα παιδιά ενός τέτοιου κόμβου μοιράζονται το άκρο εγγραφής στη σωλήνωση, αλλά δεν γράφουν ταυτόχρονα σε αυτό. Αφού ξυπνήσει ο γονέας ξυπνάει το πρώτο παιδί, το οποίο γράφει στο pipe. Ο γονέας περιμένει το παιδί να τερματιστεί (με wait) και μετά διαβάζει από την σωλήνωση. Αφού, λοιπόν, έχει διαβάσει τον πρώτο τελεστέο ξυπνά και το δεύτερο παιδί και η διαδικασία επαναλαμβάνεται. Με αυτό τον τρόπο, τα παιδιά χρησιμοποιούν το ίδιο pipe χωρίς να δημιουργείται σύγχυση, επειδή το κάνουν σειριακά.
2. Σε ένα σύστημα στο οποίο η αποτίμηση μιας αριθμητικής έκφρασής γίνεται από διαφορετικούς επεξεργαστές, το σημαντικό πλεονέκτημα που προσφέρεται είναι η εξοικονόμηση χρόνου, αφού ανεξάρτητοι υπολογισμοί (υποδέντρα του δέντρου εργασιών) γίνονται παράλληλα.