Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

2η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Ημερομηνία Παράδοσης : 01/12/2017

Ονοματεπώνυμο	**Αριθμός Μητρώου**
Αλεξάκης Κωνσταντίνος	03114086
Βασιλάκης Εμμανουήλ	03114167

Εξάμηνο	**Ακαδημαϊκό Έτος**
7ο	2017-2018

1 Δημιουργία δεδομένου δέντρου διεργασιών

1.1 Πηγαίος 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

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */

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

void Cfork_procs() {
        change_pname("C");
        printf("C : Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("C : Exiting...\n");
        exit(17);
}

 void Bfork_procs(){
        change_pname("B");
        printf("B : Starting...\n");

        pid_t pid;
        int status;

        /* Fork D */
        pid = fork();
        if (pid < 0) {
                perror("B: fork");
                exit(1);
        }
        if (pid == 0) {
                /* Child */
                Dfork_procs();
                assert(0);
        }
```

```
        printf("B: Waiting for children...\n");
        pid = wait(&status);
        explain_wait_status(pid, status);

        printf("B : Exiting...\n");
        exit(19);

}

void fork_procs() {
        /*
         * initial process is A.
         */
        change_pname("A");
        printf("A: Starting..\n");

        pid_t pid;
        int status;

        /* Fork B */
        pid = fork();
        if (pid < 0) {
                perror("A: fork");
                exit(1);
        }
        if (pid == 0) {
                /* Child */
                Bfork_procs();
                assert(0);
        }

        /* Fork C */
        pid = fork();
        if (pid < 0) {
                perror("A: fork");
                exit(1);
        }
        if (pid == 0) {
                /* Child */
                Cfork_procs();
                assert(0);
        }

        printf("A: Waiting for children...\n");
        pid = wait(&status);
        explain_wait_status(pid, status);
        pid = wait(&status);
        explain_wait_status(pid, status);

        printf("A: Exiting...\n");
        exit(16);
}
```

```
int main() {
        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();
                assert(0);
        }

        /*
         * Father
         */
        sleep(SLEEP_TREE_SEC);

        /* Print the process tree root at pid */
        show_pstree(pid);
        //show_pstree(getpid());
        /* Wait for the root of the process tree to terminate */
        pid = wait(&status);
        explain_wait_status(pid, status);
        return 0;
}
```

1.2 Έξοδος εκτέλεσης του προγράμματος

Αν τρέξουμε το εκτελέσιμο out1, θα λάβουμε την ακόλουθη έξοδο:

```
manolis@hp-255-g5:~/Desktop/Temp/1.1$ ./out1
A: Starting..
B : Starting...
A: Waiting for children...
C : Sleeping...
B: Waiting for children...
D : Sleeping...


A(9738)─┬─B(9739)──D(9741)
        └─C(9740)


D : Exiting...
C : Exiting...
My PID = 9739: Child PID = 9741 terminated normally, exit status = 13
B : Exiting...
My PID = 9738: Child PID = 9739 terminated normally, exit status = 19
My PID = 9738: Child PID = 9740 terminated normally, exit status = 17
A: Exiting...
My PID = 9737: Child PID = 9738 terminated normally, exit status = 16
```

1.3 Ερωτήσεις

1. *Τι θα γίνει αν τερματίσετε πρόωρα τη διεργασία Α, δίνοντας kill -KILL <pid>, όπου <pid> το Process ID της;*

   Αφού η διεργασία Α τερματίζεται πρόωρα ,τότε όλες οι διεργασίες–παιδιά της, που δεν έχουν πεθάνει, θα γίνουν παιδιά της init (PID=1) , η οποία κάνει περιοδικά wait().

2. *Τι θα γίνει αν κάνετε show_pstree(getpid()) αντί για show_pstree(pid) στη main(); Ποιες επιπλέον διεργασίες φαίνονται στο δέντρο και γιατί;*

```
manolis@hp-255-g5:~/Desktop/Temp/1.1$ ./out1
A: Starting..
B : Starting...
A: Waiting for children...
C : Sleeping...
B: Waiting for children...
D : Sleeping...


out1(9933)─┬─A(9934)─┬─B(9935)──D(9937)
           │         └─C(9936)
           └─sh(9938)──pstree(9939)


D : Exiting...
C : Exiting...
My PID = 9935: Child PID = 9937 terminated normally, exit status = 13
My PID = 9934: Child PID = 9936 terminated normally, exit status = 17
B : Exiting...
My PID = 9934: Child PID = 9935 terminated normally, exit status = 19
A: Exiting...
My PID = 9933: Child PID = 9934 terminated normally, exit status = 16
```

   Εκτελώντας την εντολή *show_pstree(getpid())*, θα εμφανιστεί επιπλέον και ο πατέρας της διεργασίας Α. Στην περίπτωση μας, πατέρας της διεργασίας Α είναι το κυρίως πρόγραμμα και το όνομα του είναι το όνομα του εκτελέσιμου. Ο λόγος που συμβαίνει το παραπάνω είναι γιατί καλείται η συνάρτηση *show_pstree()* με όρισμα το pid του γονέα της Α.

   Ακόμα, παρατηρούμε ότι στο δέντρο εμφανίζονται δύο διεργασίες (η sh και η pstree). Το κυρίως πρόγραμμα δημιουργεί πρώτα την sh και αυτή "γεννά" την pstree, με σκοπό την εμφάνιση του δεδομένου δεντρου από την κλήση της *show_pstree()*. Αυτό συμβαίνει επειδή η *show_pstree()* εκτελεί την εντολή φλοιού pstree, επομένως πρέπει να δημιουργηθεί μία νέα διεργασία φλοιού αρχικά, η οποία στη συνέχεια με *fork* και *execve* θα εκτελέσει την pstree.

3. *Σε υπολογιστικά συστήματα πολλαπλών χρηστών, πολλές φορές ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης. Γιατί;*

   Αν κάποιος χρήστης δεν έχει όρια ως προς τον αριθμό των διεργασιών που μπορεί να δημιουργήσει, τότε υπάρχει η πιθανότητα ο όγκος των διεργασιών που δημιούργησε να είναι τόσο μεγάλος που παρεμποδίζει την φυσιολογική λειτουργία των διεργασιών των υπολοίπων χρηστών. Ο λόγος είναι ότι η *fork()* δημιουργεί ένα ακριβές αντίγραφο στη μνήμη της διεργασίας – πατέρα, αποθηκεύοντας δεδομένα, στοίβα, κείμενο κλπ. Επομένως, αν δημιουργηθούν διεργασίες πάνω από ένα όριο, θα δημιουργηθεί πρόβλημα στην μνήμη και συνεπώς στην λειτουργία των διεργασιών των υπόλοιπων χρηστών.

2 Δημιουργία αυθαίρετου δέντρου διεργασιών

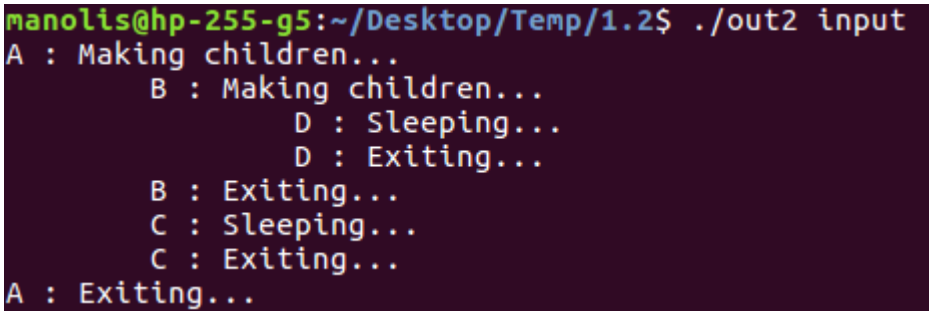
2.1 Πηγαίος κώδικας

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

#include "tree.h"

#define SLEEP_PROC_SEC  2

void leaf_process(char *name, int level){
        int i;
        for (i=0; i<level; i++)         printf("\t");
        printf("%s : Sleeping...\n", name);
        sleep(SLEEP_PROC_SEC);
        for (i=0; i<level; i++)         printf("\t");
        printf("%s : Exiting...\n", name);
        exit(10);
}

void generate_process(struct tree_node *root, int level){
        if (root->nr_children == 0) leaf_process(root->name, level);

        pid_t pid;
        int i, status;

        for (i=0; i<level; i++)         printf("\t");
        printf("%s : Making children...\n", root->name);

        for (i=0; i < root->nr_children; i++){
                pid = fork();
                if (pid == -1) {
                        printf("fork unsuccessful\n");
                        exit(1);
                }
                else if (pid == 0) generate_process(root->children + i, level + 1);
                wait(&status);     // depth first dhmiourgia paidion
        }
        //for (i=0; i < root->nr_children; i++) wait(&status);  // enallaktika ftiaxno ola ta paidia sthn arxh kai
                                                                //perimeno na gyrisoun
        for (i=0; i<level; i++) printf("\t");
        printf("%s : Exiting...\n", root->name);
        exit(11);
}

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]);
        int status;
        pid_t pid;
        pid = fork();
        if (pid == -1) {
                printf("fork unsuccessful\n");
                exit(1);
        }
        else if (pid == 0) generate_process(root, 0);
        wait(&status);
        return 0;
}
```

2.2 Έξοδος εκτέλεσης του προγράμματος

Αν τρέξουμε το εκτελέσιμο out2 με είσοδο αυτήν που δίνεται στην εκφώνηση, θα λάβουμε την ακόλουθη έξοδο:



2.3 Ερωτήσεις

1. *Με ποια σειρά εμφανίζονται τα μηνύματα έναρξης και τερματισμού των διεργασιών; Γιατί;*

   Η σειρά που εμφανίζονται τα μηνύματα είναι κατά βάθος, αφού προτού ένας πατέρας δημιουργήσει ένα δεύτερο παιδί περιμένει να έχει τερματίσει το 1ο παιδί ήδη. Δηλαδή, μετά από κάθε *fork* υπάρχει ένα *wait*.

3 Αποστολή και χειρισμός σημάτων

3.1 Πηγαίος 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 "tree.h"
#include "proc-common.h"

void leaf_procs(struct tree_node *root){
	printf("PID = %ld, name %s, is starting...\n", (long)getpid(), root->name);
	change_pname(root->name);
	raise(SIGSTOP);
	printf("PID = %ld, name = %s is awake and exiting\n", (long)getpid(), root->name);
	exit(0);
}

void fork_procs(struct tree_node *root) {
	if (root->nr_children == 0) leaf_procs(root);

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

	pid_t pid;
	int status, i;
	pid_t children[root->nr_children];
	for (i=0; i < root->nr_children; i++){
		pid = fork();
		if (pid == -1) {
			printf("fork unsuccessful\n");
			exit(1);
		}
		else if (pid == 0) fork_procs(root->children + i);
		children[i] = pid;
		wait_for_ready_children(1);				//depth first dhmiourgia
	}

	//wait_for_ready_children(root->nr_children);	//an den me noiazei h seira dhmiourgias

	/* Suspend Self */
	raise(SIGSTOP);
	printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);

	for (i=0; i < root->nr_children; i++) {
		kill(children[i], SIGCONT);
	//}
	//for (i=0; i < root->nr_children; i++) {
		pid = wait(&status);
		explain_wait_status(pid, status);			//depth first awakening
	}
```

```
        printf("PID = %ld, name = %s is exiting\n", (long)getpid(), root->name);
        /* Exit */
        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);
        }
        else if (pid == 0) {
                /* Child */
                fork_procs(root);
                assert(0);
        }

        /* Father */
        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 */
        pid = wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

3.2 Έξοδος εκτέλεσης του προγράμματος

Αν τρέξουμε το εκτελέσιμο out3 με είσοδο αυτήν που χρησιμοποιήσαμε στο προηγούμενο ερώτημα, θα λάβουμε την ακόλουθη έξοδο:

```
manolis@hp-255-g5:~/Desktop/Temp/1.3$ ./out3 input
PID = 10527, name A, is starting...
PID = 10528, name B, is starting...
PID = 10529, name D, is starting...
My PID = 10528: Child PID = 10529 has been stopped by a signal, signo = 19
My PID = 10527: Child PID = 10528 has been stopped by a signal, signo = 19
PID = 10530, name C, is starting...
My PID = 10527: Child PID = 10530 has been stopped by a signal, signo = 19
My PID = 10526: Child PID = 10527 has been stopped by a signal, signo = 19


A(10527)─┬─B(10528)──D(10529)
         └─C(10530)


PID = 10527, name = A is awake
PID = 10528, name = B is awake
PID = 10529, name = D is awake
PID = 10529, name = D is exiting
My PID = 10528: Child PID = 10529 terminated normally, exit status = 0
PID = 10528, name = B is exiting
My PID = 10527: Child PID = 10528 terminated normally, exit status = 0
PID = 10530, name = C is awake
PID = 10530, name = C is exiting
My PID = 10527: Child PID = 10530 terminated normally, exit status = 0
PID = 10527, name = A is exiting
My PID = 10526: Child PID = 10527 terminated normally, exit status = 0
```

3.3 Ερωτήσεις

1. *Στις προηγούμενες ασκήσεις χρησιμοποιήσαμε τη sleep() για τον συγχρονισμό των διεργασιών. Τι πλεονεκτήματα έχει η χρήση σημάτων;*

   Η χρήση σημάτων μας εξασφαλίζει την σωστή εμφάνιση του δέντρου διεργασιών, αφού γνωρίζουμε πως για να εκτελεστεί η συνάρτηση *show_pstree* πρέπει να έχουν δημιουργηθεί όλες οι διεργασίες παιδιά και μετά να στείλουν σήμα στον γονέα τους αναδρομικά. Με την *sleep* δεν έχουμε εξασφαλίσει την σωστή εμφάνιση του δέντρου και πρέπει να ψάξουμε και να βρούμε τι όρισμα θα πρέπει να έχει η *sleep* των φύλλων και της αρχικής διεργασίας για να έχουμε την σωστή αναπαράστασή του.

2. *Ποιος ο ρόλος της wait_for_ready_children(); Τι εξασφαλίζει η χρήση της και τι πρόβλημα θα δημιουργούσε η παράλειψή της;*

   Η *wait_for_ready_children* διακόπτει την ροή της διεργασίας – γονέα, μέχρις ότου όλα τα παιδιά αυτής έχουν κάνει *raise(SIGSTOP)*. Εφόσον γνωρίζουμε ότι μία διεργασία θα κάνει *raise(SIGSTOP)* μόνο εάν το υποδέντρο της έχει δημιουργηθεί πλήρως, με την χρήση της εντολής εξασφαλίζουμε αναδρομικά την ορθή δημιουργία ολόκληρου του δέντρου διεργασιών. Αν δεν υπήρχε αυτή η εντολή, τότε θα μπορούσε ένας κόμβος – γονέας να κάνει *raise(SIGSTOP)* προτού δημιουργηθεί πλήρως το υποδέντρο του, προκαλώντας παρόλα αυτά την εντύπωση στον γονέα του ότι το υποδέντρο έχει δημιουργηθεί φυσιολογικά και έτσι να μην έχει δημιουργηθεί το ζητούμενο δέντρο διεργασιών την στιγμή της εκτέλεσης της συνάρτησης *show_pstree*.

4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

4.1 Πηγαίος κώδικας

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

#include "proc-common.h"
#include "tree.h"

#define SLEEP_PROC_SEC  2

void leaf_process(char *name, int fd){
        printf("%s : Writing to pipe...\n", name);
        int num = atoi(name);
        if (write(fd, &num, sizeof(num)) != sizeof(num)) {
                perror("child: write to pipe");
                exit(1);
        }
        close(fd);
        exit(num);
}

void child_ps(struct tree_node *root, int level, int fd) {
        int i;
        for (i=0; i<level; i++)         printf("\t");

        if (root->nr_children == 0) leaf_process(root->name, fd);

        pid_t pid;
        int pfd[2];
        int status;

        printf("%s: Creating pipe...\n", root->name);
        if (pipe(pfd) < 0) {
                perror("pipe");
                exit(1);
        }
        for (i=0; i<level; i++)         printf("\t");
        printf("%s: Creating childen...\n", root->name);
        for (i=0; i<2; i++){
                pid = fork();
                if (pid < 0) {
                        /* fork failed */
                        perror("fork");
                        exit(1);
                }
                if (pid == 0) {
                        close(fd);
                        close(pfd[0]);      //does not use those, uses only new writing end
                        child_ps(root->children + i, level + 1, pfd[1]);
```

```
                        assert(0);
                }
        }

        /* Father */
        close(pfd[1]);      //does not write here

        pid = wait(&status);
        explain_wait_status(pid, status);
        pid = wait(&status);
        explain_wait_status(pid, status);   //den xreiazontai

        int num1, num2;
        if (read(pfd[0], &num1, sizeof(num1)) != sizeof(num1)) {
                perror("Parent: read from pipe");
                exit(1);
        }

        if (read(pfd[0], &num2, sizeof(num2)) != sizeof(num2)) {
                perror("Parent: read from pipe");
                exit(1);
        }

        close(pfd[0]);      //read everything
        int result;
        if (strcmp(root->name, "+") == 0) result = num1 + num2;
        else result = num1 * num2;

        if (write(fd, &result, sizeof(result)) != sizeof(result)) {
                perror("child: write to pipe");
                exit(1);
        }
        close(fd);
        for (i=0; i<level; i++) printf("\t");
        printf("%s : Exiting...\n", root->name);
        exit(result);
}

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]);

        int pfd[2];
        int status;
        pid_t pid;

        printf("Main: Creating pipe...\n");
        if (pipe(pfd) < 0) {
                perror("pipe");
                exit(1);
        }
```

```
	printf("Main: Creating root...\n");
	pid = fork();
	if (pid < 0) {
		/* fork failed */
		perror("Main: fork");
		exit(1);
	}
	if (pid == 0) {
		/* In child process */
		close(pfd[0]);	//does not read
		child_ps(root, 0, pfd[1]);
		assert(0);
	}
	close(pfd[1]);	//does not write
	int result;
	printf("Main: Created root with PID = %ld, waiting for it to terminate...\n", (long)pid);
	pid = wait(&status);
	explain_wait_status(pid, status);
	if (read(pfd[0], &result, sizeof(result)) != sizeof(result)) {
		perror("Main: read from pipe");
		exit(1);
	}
	close(pfd[0]);
	printf("\n\nResult = %d\n", result);
	return 0;
}
```

4.2 Έξοδος εκτέλεσης του προγράμματος

Αν τρέξουμε το εκτελέσιμο out4 με είσοδο κατάλληλη ώστε να περιγράφουμε το δέντρο της εκφώνησης, θα λάβουμε την ακόλουθη έξοδο:

```
manolis@hp-255-g5:~/Desktop/Temp/1.4$ ./out4 expr.tree
Main: Creating pipe...
Main: Creating root...
Main: Created root with PID = 10877, waiting for it to terminate...
+: Creating pipe...
+: Creating childen...
        10 : Writing to pipe...
        *: Creating pipe...
        *: Creating childen...
                +: Creating pipe...
                +: Creating childen...
                4 : Writing to pipe...
My PID = 10877: Child PID = 10878 terminated normally, exit status = 10
                        7 : Writing to pipe...
My PID = 10879: Child PID = 10881 terminated normally, exit status = 4
My PID = 10880: Child PID = 10883 terminated normally, exit status = 7
                        5 : Writing to pipe...
My PID = 10880: Child PID = 10882 terminated normally, exit status = 5
                + : Exiting...
My PID = 10879: Child PID = 10880 terminated normally, exit status = 12
        * : Exiting...
My PID = 10877: Child PID = 10879 terminated normally, exit status = 48
+ : Exiting...
My PID = 10876: Child PID = 10877 terminated normally, exit status = 58


Result = 58
```

4.3 Ερωτήσεις

1. *Πόσες σωληνώσεις χρειάζονται στη συγκεκριμένη άσκηση ανά διεργασία; Θα μπορούσε κάθε γονική διεργασία να χρησιμοποιεί μόνο μία σωλήνωση για όλες τις διεργασίες παιδιά; Γενικά, μπορεί για κάθε αριθμητικό τελεστή να χρησιμοποιηθεί μόνο μια σωλήνωση;*

   Για τον υπολογισμό της αριθμητικής έκφρασης αρκούν μία σωλήνωση ανά διεργασία – γονέα, δηλαδή μία σωλήνωση ανά σύμβολο, επομένως ο λόγος του πλήθους των σωληνώσεων προς το πλήθος των διεργασιών είναι περίπου 0,5 αφού το πλήθος των φύλλων ενός δυαδικού δέντρου είναι περίπου το μισό του πλήθους όλων των κόμβων του δέντρου, άρα οι διεργασίες – γονείς είναι το 50% των διεργασιών μας. Κάθε διεργασία – αριθμητικό σύμβολο δημιουργεί από μία νέα σωλήνωση, την οποία χρησιμοποιούν από κοινού τα δύο παιδιά της για να γράψουν το αποτέλεσμά τους. Αυτό γίνεται επειδή οι πράξεις της πρόσθεσης και του πολλαπλασιασμού έχουν την αντιμεταθετική ιδιότητα και έτσι δεν έχει διαφορά στο αποτέλεσμα η σειρά των αριθμών. Στην περίπτωση της

αφαίρεσης και της διαίρεσης, που χρειάζεται να ξέρουμε τον πρώτο και τον δεύτερο αριθμό αυστηρά, θα μπορούσε να χρησιμοποιηθούν είτε δύο σωληνώσεις, είτε μία πάλι και να την χειριστούμε με κατάλληλα σήματα, ώστε να περιμένουμε πρώτα τον έναν αριθμό και ύστερα τον δεύτερο.

2. *Σε ένα σύστημα πολλαπλών επεξεργαστών, μπορούν να εκτελούνται παραπάνω από μια διεργασίες παράλληλα. Σε ένα τέτοιο σύστημα, τι πλεονέκτημα μπορεί να έχει η αποτίμηση της έκφρασης από δέντρο διεργασιών, έναντι της αποτίμησης από μία μόνο διεργασία;*

   Σε ένα σύστημα πολλαπλών επεξεργαστών, είναι δυνατόν κάθε επεξεργαστής να αποτιμά από μία αριθμητική έκφραση – παιδί, και να στέλνει το αποτέλεσμα του εκάστοτε παιδιού στον πατέρα. Σε αυτήν την περίπτωση, αποτιμούνται παράλληλα οι δύο εκφράσεις, με αποτέλεσμα την μείωση του χρόνου που χρειάζεται για την αποτίμηση της παράστασης σε σύγκριση με τον χρόνο που θα έκανε μία μόνο διεργασία η οποία εκτελείται από έναν μόνο επεξεργαστή.