Λειτουργικά Συστήματα

Άσκηση 2: Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

Ομάδα: oslabd43
Ο/Ε: Αντώνης Παπαοικονόμου 03115140
Γιάννης Πιτόσκας 03115077

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Πηγαίος κώδικας `ask2-fork.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)
{
	int status;

	change_pname("A");
	printf("A: What is my purpose?\n");
	printf("A: Waiting for children to end their games...\n");
	pid_t pid = fork();

	if (pid < 0) {
		perror("error at B");
		exit(1);
	}

	if (pid == 0) {
		change_pname("B");
		printf("B: What is my purpose?\n");
		printf("B: Waiting for children to end their games...\n");
		pid = fork();
		if (pid < 0) {
			perror("error at D");
			exit(1);
		}
		if (pid==0) {
			change_pname("D");
			printf("D: What is my purpose?\n");
			printf("D: I'm tired now... Sleeping...zzZzZz \n");
			sleep(SLEEP_PROC_SEC);
			printf("D: My job here is done!\n");
			exit(13);
		}
		pid = wait(&status);
		explain_wait_status(pid, status);
		printf("B: My job here is done!\n");
		exit(19);
```

```
        }

        pid = fork();
        if (pid < 0) {
                perror("error at C");
                exit(1);
        }
        if (pid == 0) {
                change_pname("C");
                printf("C: What is my purpose?\n");
                printf("C: I'm tired now... Sleeping...zzZzZz \n");
                sleep(SLEEP_PROC_SEC);
                printf("C: My job here is done!\n");
                exit(17);
        }

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

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

        printf("A: My job here is done!\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(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;
}
```

Έξοδος:

```
oslabd43@orion:~/Ask2$ ./ask2-fork
A: What is my purpose?
A: Waiting for children to end their games...
B: What is my purpose?
C: What is my purpose?
C: I'm tired now... Sleeping...zzZzZz
B: Waiting for children to end their games...
D: What is my purpose?
D: I'm tired now... Sleeping...zzZzZz


A(14719)─┬─B(14720)──D(14722)
         └─C(14721)


C: My job here is done!
My PID = 14719: Child PID = 14721 terminated normally, exit status = 17
D: My job here is done!
My PID = 14720: Child PID = 14722 terminated normally, exit status = 13
B: My job here is done!
My PID = 14719: Child PID = 14720 terminated normally, exit status = 19
A: My job here is done!
My PID = 14718: Child PID = 14719 terminated normally, exit status = 16
```

<u>Ερωτήσεις:</u>

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

Το «τρέξιμο» αυτής της εντολής θα έχει ως αποτέλεσμα τον πρόωρο τερματισμό της A, με τα παιδιά αυτής να μετατρέπονται πλέον σε “zombie” και να υιοθετούνται από την init(1).

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

Η `getpid()` επιστρέφει το PID του πατέρα, δηλαδή στην δικιά μας περίπτωση της διεργασίας της main του προγράμματος μας. Όπως φαίνεται και στο διάγραμμα παρακάτω, παρατηρούμε ότι εκτός από τις διεργασίες που ξεκινάνε από την `root(A)` υπάρχουν άλλες δύο επιπλέον διεργασίες, οι `sh` και `pstree`. Κατά την κλήση της `fork()` καλείται η `execl()` η οποία δημιουργεί την διεργασία `sh` και στην συνέχεια μέσω αυτής εκτυπώνεται το δέντρο μέσω της διεργασίας `pstree`.

```
ask2-fork(14913)─┬─A(14914)─┬─B(14915)──D(14917)
                 │          └─C(14916)
                 └─sh(14918)──pstree(14919)
```

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

Ο περιορισμός στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης σε υπολογιστικά συστήματα με πολλούς παράλληλους χρήστες εξασφαλίζει την «ομαλή» λειτουργία του συστήματος αυτού, αποτρέποντας το ενδεχόμενο της χρήσης όλων των πόρων του συστήματος από έναν και μόνο χρήστη. Παράλληλα, οι περιορισμοί αυτοί μπορούν να αποτρέψουν ενδεχόμενες DoS (Denial of Service) επιθέσεις με `fork bombs`.

1.2 <u>Δημιουργία αυθαίρετου δέντρου διεργασιών</u>

Πηγαίος κώδικας `ask2-tree.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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


void create_tree(struct tree_node *node) {
        change_pname(node->name);
        pid_t pid;
        int status;

        printf("Guess who's back, Mr %s\n", node->name);
        int i;
        if (node->nr_children != 0 ) {
                for (i = 0; i < node->nr_children; ++i) {
                        printf("Mr %s begins forking\n", node->name);
                        pid = fork();
                        if (pid < 0) {
                                perror("Error at fork");
                                exit(2);
                        }
                        if (pid == 0) {
                                pid_t newPID = wait(&status);
                                explain_wait_status(newPID, status);
                                create_tree(node->children);
                        }
                }
        }
        else {
                sleep(SLEEP_PROC_SEC);
                pid = wait(&status);
                explain_wait_status(pid, status);
                exit(1);
        }

        exit(1);
}



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]);     /* returns ptr to the root of the tree */
        print_tree(root);

        int status;
        pid_t pid = fork();                     /* fork the root */

        if (pid < 0) {
                perror("Error at fork");
                exit(1);
        }
        if (pid == 0) {
                create_tree(root);              /* calling the recursive function that creates the input
tree */
                exit(1);
        }

        sleep(SLEEP_TREE_SEC);
        show_pstree(pid);

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

        return 0;
}
```

Έξοδος για το αρχείο `proc.tree`:

```
oslabd43@orion:~/Ask2$ ./ask2-tree proc.tree
A
        B
                E
                F
        C
        D
A: Forking child No.1...
A: Forking child No.2...
B: Forking child No.1...
A: Forking child No.3...
A: Waiting...
C: Sleeping...
B: Forking child No.2...
B: Waiting...
D: Sleeping...
F: Sleeping...
E: Sleeping...


A(14780)─┬─B(14781)─┬─E(14784)
         │          └─F(14785)
         ├─C(14782)
         └─D(14783)


C: Exiting...
My PID = 14780: Child PID = 14782 terminated normally, exit status = 2
D: Exiting...
A: Waiting...
F: Exiting...
E: Exiting...
My PID = 14780: Child PID = 14783 terminated normally, exit status = 2
A: Waiting...
My PID = 14781: Child PID = 14784 terminated normally, exit status = 2
```

```
B: Waiting...
My PID = 14781: Child PID = 14785 terminated normally, exit status = 2
B: Exiting...
My PID = 14780: Child PID = 14781 terminated normally, exit status = 3
A: Exiting...
My PID = 14779: Child PID = 14780 terminated normally, exit status = 3
```

Ερωτήσεις:

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

Τα μηνύματα έναρξης εμφανίζονται με BFS σειρά, αφού κάθε διεργασία δημιουργεί τα παιδιά της μέσα σε ένα for loop, αυτός είναι βασικά ο τρόπος που γίνεται η διάσχιση του δέντρου κατά την δημιουργία των διεργασιών αναδρομικά.

Τα μηνύματα τερματισμού εμφανίζονται μεν με BFS σειρά επίσης εδώ, αλλά αυτό δεν είναι αυτό που συμβαίνει κατά γενική περίπτωση, καθώς στην πραγματικότητα δεν υπάρχει μια συγκεκριμένη σειρά. Ουσιαστικά η σειρά που θα εμφανιστούν τα μηνύματα τερματισμού εξαρτάται από την σειρά που θα δοθεί στις διεργασίες (χρονοπρογραμματισμός διεργασιών).

1.3 Αποστολή και χειρισμός σημάτων

Πηγαίος κώδικας `ask2-signals2.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 create_tree_signals(struct tree_node *node)
{
	change_pname(node->name);
	printf("(PID = %ld) %s: Starting...\n", (long)getpid(), node->name);

	int status;
	pid_t pid[node->nr_children];
	pid_t wait_pid;

	int i;
	for (i=0; i < node->nr_children; ++i)
	{
		pid[i] = fork();

		if (pid[i] < 0)
		{
			perror("fork");
			exit(1);
		}

		if (pid[i] == 0)
		{
			create_tree_signals(node->children+i);
		}
	}

	printf("%s: Waiting for ready children...\n", node->name);
```

```
        wait_for_ready_children(node->nr_children);        /* if nr_children = 0 => it just continues
immediately */
        raise(SIGSTOP);

        printf("(PID = %ld) %s: Woke up!!\n", (long)getpid(), node->name);

        for (i=0; i < node->nr_children; ++i)
        {
                kill(pid[i], SIGCONT);
                printf("%s: Waiting...\n", node->name);
                wait_pid = wait(&status);
                explain_wait_status(wait_pid, status);

                if (i == (node->nr_children) -1)    /* waiting for children to terminate and parent exits
*/
                {
                        printf("%s: Exiting...\n", node->name);
                        exit(2);
                }
        }

        printf("%s: Exiting...\n", node->name);
        exit(0);
}

/*
 *      use wait_for_ready_children() to wait until
 *      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);
        }

        root = get_tree_from_file(argv[1]);
        print_tree(root);
        pid = fork();
        if (pid < 0) {
                perror("first fork");
                exit(1);
        }
        if (pid == 0) {
                create_tree_signals(root);
                exit(1);
        }

        wait_for_ready_children(1);

        show_pstree(pid);

        kill(pid, SIGCONT);

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

        return 0;
}
```

Έξοδος για το αρχείο `proc.tree`:

```
oslabd43@orion:~/Ask2$ ./ask2-signals2 proc.tree
A
        B
                E
                F
        C
        D
(PID = 14808) A: Starting...
(PID = 14809) B: Starting...
A: Waiting for ready children...
(PID = 14810) C: Starting...
B: Waiting for ready children...
(PID = 14812) E: Starting...
E: Waiting for ready children...
C: Waiting for ready children...
My PID = 14808: Child PID = 14810 has been stopped by a signal, signo = 19
My PID = 14809: Child PID = 14812 has been stopped by a signal, signo = 19
(PID = 14811) D: Starting...
D: Waiting for ready children...
(PID = 14813) F: Starting...
My PID = 14808: Child PID = 14811 has been stopped by a signal, signo = 19
F: Waiting for ready children...
My PID = 14809: Child PID = 14813 has been stopped by a signal, signo = 19
My PID = 14808: Child PID = 14809 has been stopped by a signal, signo = 19
My PID = 14807: Child PID = 14808 has been stopped by a signal, signo = 19


A(14808)─┬─B(14809)─┬─E(14812)
         │          └─F(14813)
         ├─C(14810)
         └─D(14811)


(PID = 14808) A: Woke up!!
A: Waiting...
(PID = 14809) B: Woke up!!
B: Waiting...
(PID = 14812) E: Woke up!!
E: Exiting...
My PID = 14809: Child PID = 14812 terminated normally, exit status = 0
B: Waiting...
(PID = 14813) F: Woke up!!
F: Exiting...
My PID = 14809: Child PID = 14813 terminated normally, exit status = 0
B: Exiting...
My PID = 14808: Child PID = 14809 terminated normally, exit status = 2
A: Waiting...
(PID = 14810) C: Woke up!!
C: Exiting...
My PID = 14808: Child PID = 14810 terminated normally, exit status = 0
A: Waiting...
(PID = 14811) D: Woke up!!
D: Exiting...
My PID = 14808: Child PID = 14811 terminated normally, exit status = 0
A: Exiting...
My PID = 14807: Child PID = 14808 terminated normally, exit status = 2
```

Ερωτήσεις:

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

Ουσιαστικά με τη χρήση της `sleep()` στις προηγούμενες ασκήσεις κοιμίζαμε τις διεργασίες για ένα προκαθορισμένο από εμάς χρονικό διάστημα τέτοιο, ώστε να επαρκεί προκειμένου να δημιουργηθεί ολόκληρο το δέντρο διεργασιών προτού το τυπώσουμε. Ωστόσο, για ένα μεγάλο δέντρο διεργασιών αυτή η τεχνική δεν

είναι αυτή που προτιμάται. Επίσης με χρήση της `sleep()` δεν επιτυγχάνεται ακριβής συγχρονισμός των διεργασιών.

Με τη χρήση σημάτων, λοιπόν, αντί της `sleep()` , δε χρειάζεται να περιμένουμε. Πετυχαίνουμε πιο ακριβή συγχρονισμό, καθώς ξυπνάμε τις διεργασίες με SIGCONT ή τις κοιμίζουμε με SIGSTOP ακριβώς τις στιγμές που χρειάζεται η εκάστοτε διεργασία να βρίσκεται στην αντίστοιχη κατάσταση. Ουσιαστικά, με αυτόν τον τρόπο έχουμε ασύγχρονη επικοινωνία μεταξύ των διεργασιών μέσω σημάτων (π.χ. ο πατέρας στέλνει SIGCONT στο παιδί για να ξυπνήσει όταν αυτό είναι επιθυμητό).

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()` ουσιαστικά κάνει wait μέχρις ότου όλα τα παιδιά της διεργασίας που την καλεί έχουν γίνει pause. Ρίχνοντας και μία ματιά και στην υλοποίηση της προαναφερόμενης συνάρτησης μπορούμε να δούμε ότι ο έλεγχος της κατάστασης τερματισμού που πραγματοποιεί εξαρτάται από το τι επιστρέφει η `waitpid()` και από το status WIFSTOPPED το οποίο επιστρέφει μη μηδενική τιμή αν η διεργασία παιδί έχει γίνει pause.

Στο δικό μας πρόγραμμα η `wait_for_ready_children()` που καλείται στη main γίνεται με όρισμα 1 καθώς η αρχική διεργασία (πατέρας του root) περιμένει να δημιουργηθεί ολόκληρο το δέντρο ώστε αμέσως αφού δημιουργηθεί να μας το τυπώσει ολοκληρωμένο. Τώρα, μέσα στην συνάρτηση `create_tree_signals` η κλήση της `wait_for_ready_children()` γίνεται με όρισμα τον αριθμό των παιδιών της αντίστοιχης διεργασίας που την καλεί και εν συνεχεία καλείται αναδρομικά για κάθε παιδί.

Αν δεν χρησιμοποιούσαμε την `wait_for_ready_children()` ενδεχομένως να έστελνε ο πατέρας SIGCONT πριν προλάβει το παιδί να κάνει raise SIGSTOP , δηλαδή ενώ τρέχει η διεργασία παιδί (θα μπορούσε να μην είχε καν δημιουργηθεί ακόμη η διεργασία παιδί), μπορεί να του ερχόταν το signal SIGCONT το οποίο δε θα δρούσε κάπως, καθώς ήδη έτρεχε κανονικά (δεν ήταν paused) και μόλις έκανε raise SIGSTOP θα έμπαινε σε ένα αέναο pause.

Χρησιμοποιώντας την `wait_for_ready_children()` ουσιαστικά το παραπάνω πρόβλημα επιλύεται, καθώς μας εξασφαλίζει γνώση της κατάστασης του δέντρου προτού επιχειρήσουμε να κάνουμε SIGCONT σε μια διεργασία παιδί η οποία δεν είναι paused ή δεν είναι καν δημιουργημένη.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Πηγαίος κώδικας `ask2-pipe2.c`:

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


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

void create_tree_pipes(struct tree_node *node, int fd[2])
{
          change_pname(node->name);
/*        printf("(PID = %ld) %s: Starting...\n", (long)getpid(), node->name);          */

          int num;
          int status;
          pid_t pid[node->nr_children];
          pid_t wait_pid;

          /*
             int fd[2];
```

```
   if (pipe(fd) < 0)
   {
   perror("pipe");
   }
   */

int i;
for (i=0; i < node->nr_children; ++i)
{
          pid[i] = fork();

          if (pid[i] < 0)
          {
                    perror("fork");
                    exit(1);
          }

          if (pid[i] == 0)
          {
                    create_tree_pipes(node->children+i, fd);
          }
}
/*
printf("%s: Waiting for ready children...\n", node->name);
*/
wait_for_ready_children(node->nr_children);
raise(SIGSTOP);

if (node->nr_children == 0) {
          num = atoi(node->name);
          if ( write(fd[1], &num, sizeof(num)) != sizeof(num) )
          {
                    perror("write to pipe");
                    exit(1);
          }
          printf("%s: Writing %d to pipe\n", node->name, num);
}

/*        printf("(PID = %ld) %s: Woke up!!\n", (long)getpid(), node->name);  */

int afla[2];

for (i=0; i < node->nr_children; ++i)
{
          kill(pid[i], SIGCONT);
/*        printf("%s: Waiting...\n", node->name);          */
          wait_pid = wait(&status);
          explain_wait_status(wait_pid, status);

          if ( read(fd[0], &afla[i], sizeof(afla[i])) != sizeof(afla[i]) ) {
                    perror("read from pipe");
                    exit(1);
          }

}

int result;
if ( strcmp(node->name , "+") == 0 ) {
          result = afla[0] + afla[1];
          printf("Result = %d %s %d => Result = %d\n", afla[0], node->name, afla[1], result);
/*        printf("%s: Exiting...\n", node->name);          */
          if ( write(fd[1], &result, sizeof(result)) != sizeof(result) ){
                    perror("write to pipe");
                    exit(1);
          }
          exit(2);
}
else if ( strcmp(node->name , "*") == 0 ) {
          result = afla[0] * afla[1];
          printf("Result = %d %s %d => Result = %d\n", afla[0], node->name, afla[1], result);
/*        printf("%s: Exiting...\n", node->name);*/
          if ( write(fd[1], &result, sizeof(result)) != sizeof(result) ){
```

```
                        perror("write to pipe");
                        exit(1);
                }
                exit(2);

        }

        /*printf("%s: Exiting...\n", node->name);*/
        exit(0);
}


int main(int argc, char *argv[])
{
        struct tree_node *root;
        int status;
        int fd[2];
        int temp;

        if (argc != 2) {
                fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
                exit(1);
        }

        if (pipe(fd) < 0)
        {
                perror("pipe");
                exit(1);
        }

        root = get_tree_from_file(argv[1]);
        print_tree(root);

        pid_t pid = fork();

        if ( pid < 0 ) {
                perror("fork");
                exit(1);
        }

        if ( pid == 0 ) {
                create_tree_pipes(root, fd);
                exit(1);
        }

        wait_for_ready_children(1);
        show_pstree(pid);
        kill(pid, SIGCONT);
        pid = wait(&status);
        explain_wait_status(pid, status);

        if ( read(fd[0], &temp, sizeof(int))!=sizeof(int) ){
                perror("read pipe");
        }
        printf("\nThe final result is: %d\n\n", temp);
        return 0;
}
```

Έξοδος για `expr.tree`:

```
oslabd43@orion:~/Ask2$ ./ask2-pipe2 expr.tree
+
        10
        *
                +
                        5
                        7
                4
My PID = 13295: Child PID = 13296 has been stopped by a signal, signo = 19
My PID = 13297: Child PID = 13299 has been stopped by a signal, signo = 19
My PID = 13298: Child PID = 13300 has been stopped by a signal, signo = 19
```

```
My PID = 13298: Child PID = 13301 has been stopped by a signal, signo = 19
My PID = 13297: Child PID = 13298 has been stopped by a signal, signo = 19
My PID = 13295: Child PID = 13297 has been stopped by a signal, signo = 19
My PID = 13294: Child PID = 13295 has been stopped by a signal, signo = 19
```

```
+(13295)─┬─*(13297)─┬─+(13298)─┬─5(13300)
         │          │          └─7(13301)
         │          └─4(13299)
         └─10(13296)
```

```
10: Writing 10 to pipe
My PID = 13295: Child PID = 13296 terminated normally, exit status = 0
5: Writing 5 to pipe
My PID = 13298: Child PID = 13300 terminated normally, exit status = 0
7: Writing 7 to pipe
My PID = 13298: Child PID = 13301 terminated normally, exit status = 0
Result = 5 + 7 => Result = 12
My PID = 13297: Child PID = 13298 terminated normally, exit status = 2
4: Writing 4 to pipe
My PID = 13297: Child PID = 13299 terminated normally, exit status = 0
Result = 12 * 4 => Result = 48
My PID = 13295: Child PID = 13297 terminated normally, exit status = 2
Result = 10 + 48 => Result = 58
My PID = 13294: Child PID = 13295 terminated normally, exit status = 2

The final result is: 58
```

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;**

Στη συγκεκριμένη άσκηση χρειάζεται ένα pipe ανά διεργασία μέσα από το οποίο θα γίνονται τα read-write (εξαιρέσεις αποτελούν οι διεργασίες φύλλα που κάνουν μόνο write και η διεργασία που αποτελεί ρίζα του δέντρου η οποία κάνει μόνο read το τελικό αποτέλεσμα και δίνει προς εκτύπωση).

Στην ζητούμενη περίπτωση της άσκησης ασφαλώς και μπορεί κάθε γονική διεργασία να χρησιμοποιεί μόνο μία σωλήνωση για όλες τις διεργασίες παιδιά, καθώς δεν χρειάζεται να γνωρίζουμε από ποιο παιδί προέρχεται το κάθε αποτέλεσμα μιας και οι εμπλεκόμενες αριθμητικές πράξεις είναι αντιμεταθετικές (πρόσθεση, πολλαπλασιασμός).

Στην περίπτωση που είχαμε αριθμητικούς τελεστές όπως αυτός της αφαίρεσης ή αυτός της διαίρεσης δε θα μπορούσε να ένα κοινό pipe για την επικοινωνία της γονικής διεργασίας με όλες τις διεργασίες παιδιά για τον απλούστατο λόγο ότι οι δύο αυτές αριθμητικές πράξεις δεν υποστηρίζουν την ιδιότητα της αντιμεταθετικότητας και επομένως πρέπει να γνωρίζουμε από ποιο παιδί έρχεται ποιο αποτέλεσμα, γεγονός που απαιτεί να χρησιμοποιούμε ένα pipe ανά παιδί.

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, μπορούμε να έχουμε hyperthreding για τις διαφορετικές διεργασίες που δημιουργούμε ανά τον κόμβο του δέντρου μας. Με αυτόν τον τρόπο πετυχαίνουμε παράλληλη εκτέλεση τους αφού κάθε επεξεργαστής του συστήματος μας θα εκτελεί διαφορετική διεργασία και κατ' επέκταση πολύ καλύτερη τελική απόδοση (και μικρό χρόνο run time). Βέβαια, θα πρέπει κάθε επεξεργαστής να εκτελεί διεργασίες των οποίων η ικανότητα παραγωγής αποτελέσματος δεν εξαρτάται από το αποτέλεσμα άλλων εμπλεκόμενων διεργασιών που δεν έχουν ακόμη εκτελεστεί.