Λειτουργικά Συστήματα Υπολογιστών

2η Εργαστηριακή Αναφορά

Θοδωρής Παπαρρηγόπουλος el18040
Ορφέας Φιλιππόπουλος el18082

Άσκηση 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 fork_procs(void)
{
        /*
         *         * initial process is A.
         *                 */

        change_pname("A");
        printf("A: Waiting for children...\n");
        pid_t p, pid;
        int statusA;
        p = fork();
        if (p < 0) {
                perror("main: fork");
                exit(1);
        }
        if (p == 0) {
                change_pname("B");
                int statusB;
                printf("B: Waiting for child...\n");
                pid_t p1;
                p1 = fork();
                if (p1<0) {
                        perror("main: fork");
                        exit(1);
```

```
                }
                if (p1 == 0) {
                        change_pname("D");
                        printf("D: Sleeping...\n");
                        sleep(SLEEP_PROC_SEC);
                        printf("D: Exiting...\n");
                        exit(13);
                }
                p1 = wait(&statusB);
                explain_wait_status(p1, statusB);
                printf("B: Exiting...\n");
                exit(19);
        }
        else  {
                pid = fork();
                if (pid < 0) {
                        perror("main: fork");
                        exit(1);
                }
                if (pid == 0) {
                        change_pname("C");
                        printf("C: Sleeping...\n");
                        sleep(SLEEP_PROC_SEC);
                        printf("C: Exiting...\n");
                        exit(17);
                }
        }

        p = wait(&statusA);
        explain_wait_status(p, statusA);
        pid = wait(&statusA);
        explain_wait_status(pid, statusA);
        /* ... */

        printf("A: Exiting...\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;
}
```

Output:

```
oslaba33@os-node1:~/second_lab/forktree$ oslaba33@os-node1:~/second_lab/forktree$ ./1__1 proc.tree
A: Waiting for children...
B: Waiting for child...
C: Sleeping...
D: Sleeping...


A(15049)─┬─B(15050)─┬─D(15052)
         └─C(15051)


C: Exiting...
My PID = 15049: Child PID = 15051 terminated normally, exit status = 17
D: Exiting...
My PID = 15050: Child PID = 15052 terminated normally, exit status = 13
B: Exiting...
My PID = 15049: Child PID = 15050 terminated normally, exit status = 19
A: Exiting...
My PID = 15048: Child PID = 15049 terminated normally, exit status = 16
```

Ερωτήσεις

1) Όταν τερματίζουμε πρόωρα την διεργασία Α υπάρχουν 2 ενδεχόμενα. Αρχικά, υπάρχει το ενδεχόμενο να μην είχαμε προλάβει να δημιουργήσουμε τα παιδιά. Τότε, σε αυτή την περίπτωση τερματίζει και το πρόγραμμα. Στην 2η περίπτωση απλά τα παιδιά αντί για παιδιά του Α θα γίνουν παιδιά της init.

2) Η getpid() θα μας επιστρέψει το pid της main διεργασίας. Συνεπώς, το δέντρο που θα μας εμφανίζει θα περιλαμβάνει και την Α. Το δέντρο επιπλέον θα περιλαμβάνει και μερικά system calls τα οποία δημιουργούνται με την main, η sh (shell) και κάνει execv ώστε η pstree να εκτελεστεί.

3) Συνήθως, οι διαχειριστές θέτουν όρια στον συνολικό αριθμό διεργασιών ενός χρήστη. Αυτό συμβαίνει καθώς αν αφήσεις ανεξέλεγκτο τον χρήστη είναι πολύ πιθανόν να γεμίσει η μνήμη και να crashari το λειτουργικό. Έτσι, γίνεται τόσο για θέματα ασφάλειας όσο και για θέματα ορθότητας.

Άσκηση 2

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"
#include "tree.h"

#define SLEEP_PROC_SEC  5
#define SLEEP_TREE_SEC  3



void make_me(struct tree_node *root){//, int mypid){


        change_pname(root->name);
//      printf("pid: %d\n", mypid);
//      printf("%s, waiting all children", root->name);


        if(root->nr_children == 0){
                printf("%s am a leaf\n", root->name);
                change_pname(root->name);
                printf("%s: Is sleeping...\n", root->name);
                sleep(SLEEP_PROC_SEC);
                printf("%s: exiting...\n", root->name);
                exit(0);
        }

        printf("%s, waiting all children\n", root->name);

        //change_pname(root->name);

        int number = root->nr_children;

        pid_t p;

        int i = 0;

        for(i = 0; i<number; i++){
                p = fork();
                if(p < 0){
                        perror("fork");
                        exit(1);
```

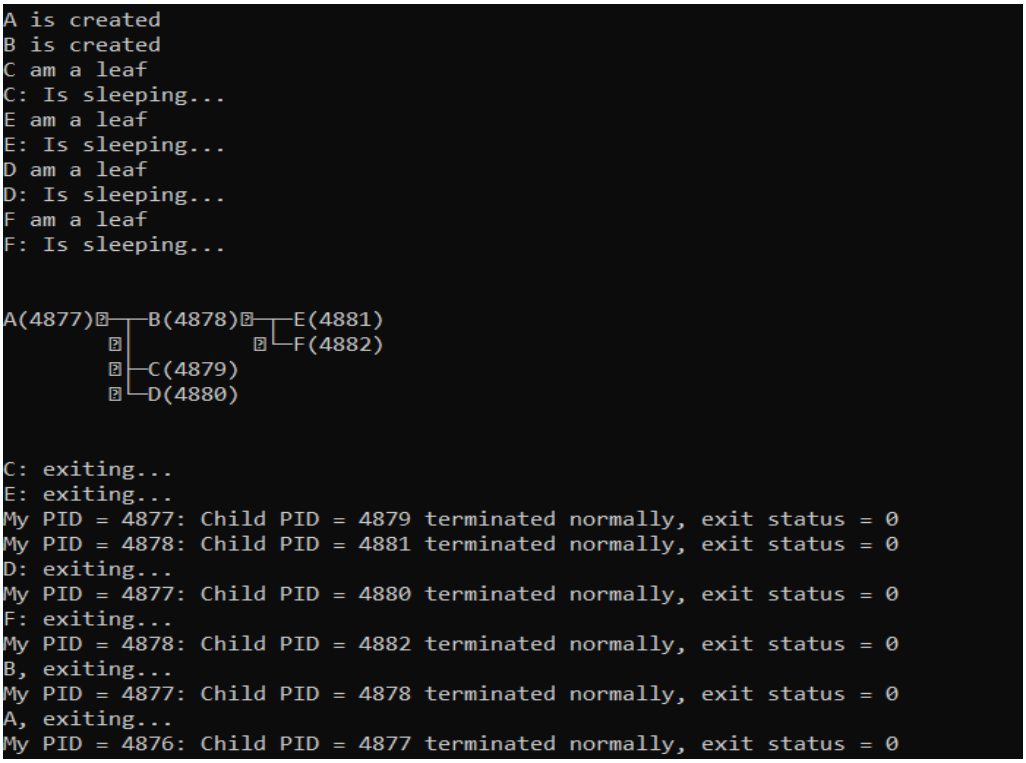
```
                }
                if(p == 0){
                        make_me(root->children + i);//, getpid());

                }
        }

        int status;

        for(i = 0; i<number; i++){
                p = wait(&status);
                explain_wait_status(p, status);
        }

        printf("%s, exiting...\n", root->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]);
        if(root == NULL) return 0;
      print_tree(root);

        //sleep(SLEEP_TREE_SEC);

        pid_t p, mypid;
        p = fork();
        if (p < 0){
                perror("fork");
                exit(1);
        }
        else if (p == 0){
                mypid = getpid();
                make_me(root);//, mypid);
        }


        sleep(SLEEP_TREE_SEC);
```

```
        show_pstree(getpid()+1);

        int status;
        p = wait(&status);
        explain_wait_status(p, status);

    return 0;
}
```

Ερωτήσεις

1) **Output:**

Από το screenshot:



παρατηρούμε ότι:

Πρώτα δημιουργείται το A. Έπειτα το B και C, μετά το παιδί του B(E) μετά το D και στη συνέχεια το άλλο παιδί του B(F). Παρόλο που θα περιμέναμε κάτι του στυλ ABCDEF ή ABEFCD λάβαμε ABCEDF. Αυτό έχει να κάνει πρωτίστως με το πως τοποθετεί ο scheduler τις διεργασίες και είναι κάπως τυχαίος ο τρόπος που αποφασίζεται ποιο θα τοποθετηθεί πρώτα (τυχαιότητα scheduler).

Άσκηση 3

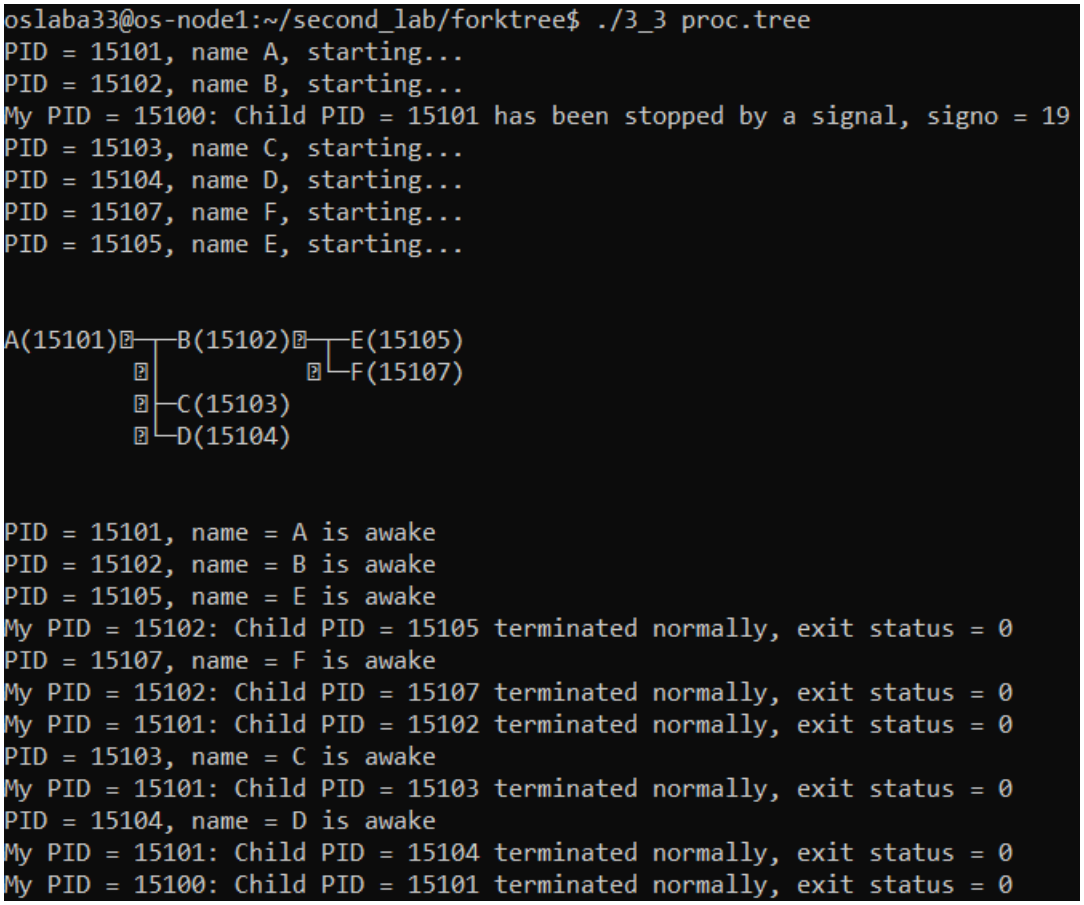
Πηγαίος Κώδικας

```
#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"

void fork_procs(struct tree_node *root){
	/*
	 * Start
	 */
	printf("PID = %ld, name %s, starting...\n",(long)getpid(), root->name);
	change_pname(root->name);
	int i=0, status, p1;
	if (root->nr_children == 0) {
		raise(SIGSTOP);
		printf("PID = %ld, name = %s is awake\n", (long)getpid(), root->name);
		exit(0);
	}
	//wow
	pid_t p[root->nr_children];
	for (i=0; i < (root->nr_children); i++) {
		p[i] = fork();
		if (p[i] < 0) {
			perror("main: fork");
			exit(1);
		}
		if (p[i] == 0) {
			fork_procs(root->children + i);
		}
	}
	/* ... */
	   wait_for_ready_children(root->nr_children);
	/*
	 * 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(p[i], SIGCONT);
		p1 = wait(&status);
		explain_wait_status(p1, status);
	}
	/* ... */
```

```
	/*
	 * Exit
	 */
	exit(0);
}

/*
 * 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(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_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);

	/* Wait for the root of the process tree to terminate */
	wait(&status);
	explain_wait_status(pid, status);

	return 0;
}
```

Output:



(Παρατηρούμε ότι οι διεργασίες γίνονται awake με dfs-order (ABEFCD).)

Ερωτήσεις

1) Αντί της sleep() χρησιμοποιήσαμε τα σήματα. Αυτό μας εξασφάλισε καλύτερο συγχρονισμό μεταξύ των διεργασιών μας, ενώ με τη sleep() η διεργασία περιμένει κάποιον χρόνο(που εδώ συγκεκριμένα εμείς ορίζουμε) και έτσι δεν είμαστε σίγουροι (πάντα) ότι γίνονται όλα όσα θέλουμε (με τη σειρά που θέλουμε). Επιπλέον, δεν έχουμε ευελιξία στο τι μπορούμε να κάνουμε. Με τη χρήση σημάτων εξασφαλίζουμε ότι με το που ολοκληρωθεί μια διεργασία ενημερώνονται οι άλλες

διεργασίες που επικοινωνούν άμεσα μαζί της (πατέρας στη συγκεκριμένη περίπτωση) με αποτέλεσμα να πετυχαίνουμε το συγχρονισμό. Επιπροσθέτως με χρήση σημάτων μπορούμε να κάνουμε αρκετά πιο περίπλοκα πράγματα - εφαρμογές που με χρήση της sleep είναι αδύνατο να γίνουν.

2) Η συνάρτηση wait_for_ready_children() εξασφαλίζει ότι θα περιμένει ο πατέρας πρώτα τα παιδιά του να κάνουν SIGSTOP (θα έχουν αναστείλει τη λειτουργία τους δηλαδή). Αυτό εξασφαλίζει πως οι διεργασίες μας είναι συγχρονισμένες δεδομένου ότι αν ο πατέρας λάβει SIGCONT πρέπει να ενημερώσει αντίστοιχα και τα παιδιά του (και με την wait_for_ready_children() θα ξέρει πως έχουν σταματήσει για να τους το στείλει αλλιώς αυτό θα δημιουργούσε προβλήματα). Εάν έλλειπε αυτή η εντολή τότε ο πατέρας δεν θα περίμενε όλα τα παιδιά του να αναστείλουν την λειτουργία τους με αποτέλεσμα να έχουμε ως πιθανό ενδεχόμενο το να στείλει SIGCONT ο πατέρας στο παιδί ενώ το παιδί δεν έχει κάνει ακόμα SIGSTOP (δηλαδή να βρίσκεται ποιο κάτω στο κώδικα του παιδιού και να μην έχει προλάβει να εκτελεστεί η εντολή αυτή) με αποτέλεσμα το SIGCONT αυτό που έστειλε ο πατέρας να πάει χαμένο ενώ όταν το παιδί κάνει SIGSTOP πρακτικά να “freezarei” εκεί, γεγονός προφανώς που είναι ανεπιθύμητο.

Άσκηση 4

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"
#include "tree.h"

//#define SLEEP_PROC_SEC  10
//#define SLEEP_TREE_SEC  3


void fork_procs(struct tree_node *root, int pfd) {
	int i=0;
	//printf("%s is created...\n", root->name);
	change_pname(root->name);
	if ((root->nr_children) == 0) {
		int l;
		l = atoi(root->name);
//		printf("Value is: %d\n", l);
		if (write(pfd, &l, sizeof(l)) != sizeof(l)) {
			perror("Child: write to pipe\n");
			exit(1);
		}
		close(pfd);
		exit(0);
	}
	else {
		//printf("%s is waiting for all children...\n", root->name);
		//pid_t pid;
		int status;
		int res[2];
		pid_t pids[2];
		int pfd1[2];
		int pfd2[2];
		if(pipe(pfd1) < 0) {
			perror("Pipe\n");
			exit(1);
		}
		if(pipe(pfd2) < 0){
			perror("Pipe\n");
			exit(1);
		}
		// for one pipe for all children if wait is inside (as it is) for loop and only pfd1 is used
		for (i=0; i < (root->nr_children); i++) {
			//int pfd1[2];
```

```
//                  if (pipe(pfd1) < 0) {
//                        exit(1);
//                  }
//                  printf("root->name: %s,my %dth child, pfd[0]: %d, pfd[1]: %d\n", root->name, i,
pfd1[0], pfd1[1]);
                    pids[i] = fork();
                    if (pids[i] < 0) {
                          perror("main: fork");
                          exit(1);
                    }
                    if (pids[i] == 0) {
                          if(i==0){
                                close(pfd1[0]);
                                fork_procs(root->children + i, pfd1[1]);
                          }
                          else {
                                close(pfd2[0]);
                                fork_procs(root->children + i, pfd2[1]);
                          }
                    }

                    //pid = wait(&status);//not necesairy, read freezes the program anyway
                    //explain_wait_status(pid, status);
                    //if (read(pfd1[0], &res[i], sizeof(res[i])) != sizeof(res[i])) {
                    //      perror("Pipe\n");
                    //      exit(1);
                    //} // the programm freezes so there is no chance both childs will write, if no exit
//after write better solution is comment out wait 4 lines above
              }
              close(pfd1[1]);//father closes write file discriptors
              close(pfd2[1]);

              for(i = 0; i <root->nr_children; i++){
                    pids[i] = wait(&status);
                    explain_wait_status(pids[i], status);
              }
//            int i = 0;
              if (read(pfd1[0], &res[0], sizeof(res[0])) != sizeof(res[0])) {
                    perror("Pipe\n");
                    exit(1);
              }
              if(read(pfd2[0], &res[1], sizeof(res[1])) != sizeof(res[1])){
                    perror("Pipe\n");
                    exit(1);
              }
//we can also read the pipe after we create all the children (having one pipe for all of //them)
//            close(pfd1[1]);


              if ((root->name[0]) == '+') {
                    res[0] += res[1];
              }
```

```
            else {
                res[0] *= res[1];
            }
            printf("Result is: %d\n", res[0]);
            if (write(pfd, &res[0], sizeof(res[0])) != sizeof(res[0])) {
                perror("Pipe");
                exit(1);
            }

            //printf("%s is exiting...\n", root->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]);
        print_tree(root);
        pid_t p;
        int status, result;
        int pfd[2];
        if (pipe(pfd) < 0) {
                perror("pipe");
                exit(1);
        }
        p = fork();
        if (p < 0) {
                perror("main: fork");
                exit(1);
        }
        if (p == 0) {
                fork_procs(root, pfd[1]);
                //exit(1);
        }
        close(pfd[1]);
        if (read(pfd[0], &result, sizeof(result)) != sizeof(result)) {
                perror("Father read from pipe...\n");
                exit(1);
        }
        p = wait(&status);
        explain_wait_status(p, status);
        printf("Result is: %d\n", result);
        exit(0);
}
```

Output:

```
oslaba33@os-node1:~/second_lab/forktree$ ./4_4 expr.tree
+
        10
        *
                +
                        5
                        7
                4
My PID = 15174: Child PID = 15175 terminated normally, exit status = 0
My PID = 15176: Child PID = 15177 terminated normally, exit status = 0
My PID = 15177: Child PID = 15179 terminated normally, exit status = 0
My PID = 15177: Child PID = 15180 terminated normally, exit status = 0
Result is: 12
My PID = 15176: Child PID = 15178 terminated normally, exit status = 0
Result is: 48
My PID = 15174: Child PID = 15176 terminated normally, exit status = 0
Result is: 58
My PID = 15173: Child PID = 15174 terminated normally, exit status = 0
Result is: 58
```

(Παρατηρούμε ότι όντως το αποτέλεσμα πρέπει να είναι 10 + 4 * (5 + 7) = 58.)

Ερωτήσεις

1) Στη συγκεκριμένη περίπτωση χρησιμοποιούμε τόσα pipes όσα και τα παιδιά του κάθε πατέρα (2 δηλαδή). Στη συγκεκριμένη περίπτωση αν χρησιμοποιούσαμε 1 pipe για όλα τα παιδιά δεν θα είχαμε κάποιο θέμα καθώς έχουμε μόνο πολ/σμο και πρόσθεση που έχουν την αντιμεταθετική ιδιότητα. Από την άλλη εάν είχαμε και διαίρεση και αφαίρεση επειδή δεν θα ήμασταν σίγουροι για τη σειρά που θα έμπαιναν τα νούμερα (που θα τα έκαναν write τα παιδιά στο pipe) προς πράξη στο pipe (και η διαίρεση και η αφαίρεση δεν έχουν την αντιμεταθετική ιδιότητα). Βέβαια και αυτό το πρόβλημα λύνεται και είναι σε σχόλια στον παραπάνω κώδικα (μέσα στη for που δημιουργώ τα παιδιά:

```
//pid = wait(&status);//not necesairy, read freezes the program anyway
//explain_wait_status(pid, status);
//if (read(pfd1[0], &res[i], sizeof(res[i])) != sizeof(res[i])) {
//      perror("Pipe\n");
//      exit(1);
//}
```

Προφανώς για να τρέξει πρέπει να γίνουν κατάλληλες τροποποιήσεις καθώς το στιγμιότυπο του προγράμματος είναι προσαρμοσμένο για να τρέξει στη περίπτωση που έχω 2 Pipes)

Με αυτό το κώδικα κάνοντας wait-read (ή και σκέτο read, καθώς και αυτό freezarei το πρόγραμμα μέχρι να γραφτεί κάτι στο Pipe) διαδοχικά τα παιδιά λύνεται το παραπάνω πρόβλημα καθώς διαβάζουμε πρώτα το πρώτο παιδί, κάνουμε wait, στη συνέχεια διαβάζουμε το 2ο παιδί κλπ με αποτέλεσμα το αποτέλεσμα των παιδιών να μπαίνει διαδοχικά στο Pipe. Με αυτό το τρόπο χρησιμοποιούμε και ένα Pipe και μπορούμε να καλύψουμε και τις περιπτώσεις όπου έχουμε και διαίρεση και αφαίρεση. Βέβαια δεν είναι αποδοτικό καθώς εξαφανίζεται η έννοια του ταυτοχρονισμού.