Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών υπολογιστών

Εξάμηνο 6ο

Μάθημα: Λειτουργικά Συστήματα

Μέλη Ομάδας: Μπουσμπουκέα Γεωργία el19059
Μιχελάκης Παναγιώτης el19119

Έτος: 2022 - 2023

2η Εργαστηριακή Άσκηση

Άσκηση 1.1

Παραθέτουμε παρακάτω τον πηγαίο κώδικα της άσκησης 1.1 .

```
#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
 *
 */

// D is a leaf process thus it sleeps and then exits
void fork_procsD(void){
        printf("D:Initiating...\n");
        change_pname("D");
        printf("D:Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("D:Exiting...\n");
        exit(13);
}
```

```
// C is a leaf process thus it sleeps and then exits
void fork_procsC(void){
        printf("C:Initiating...\n");
        change_pname("C");
        printf("C:Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("C:Exiting...\n");
        exit(17);
}
// B is not a leaf process thus it forks and waits for child to die
void fork_procsB(void){
        change_pname("B");
        printf("B:Initiating...\n");
        pid_t pidD;
        int status;
        pidD=fork();
        if(pidD<0){
                perror("B:fork");
                exit(1);
        }
        if(pidD==0){
                fork_procsD();
                exit(1);

        }
        if(pidD>0){
                printf("B:Waiting...\n");
                pidD= wait(&status);
                explain_wait_status(pidD, status);
                printf("B:Exiting...\n");
                exit(19);
        }
}

// A is not a leaf process thus it forks and waits for children to die
void fork_procsA(void)
{
        printf("A:Initiating...\n");
        pid_t pidB, pidC;
        int status1, status2;
        /*
         * initial process is A.
         */

        change_pname("A");
        pidB = fork();
        if(pidB<0)
        {
                perror("A:fork");
                exit(1);
        }
        if(pidB==0)
```

```
        {
                fork_procsB();
                exit(1);
        }
        if(pidB > 0)
        {
                printf("A: Waiting...\n");
                pidC=fork();
                if(pidC<0){
                        perror("A:fork");
                        exit(1);
                }
                if(pidC==0){
                        fork_procsC();
                        exit(1);
                }
                if(pidC>0){
                        pidC = wait(&status1);
                        pidB = wait(&status2);
                        explain_wait_status(pidC, status1);
                        explain_wait_status(pidB, status2);
                        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 pidA;
        int status;

        /* Fork root of process tree */
        pidA = fork();
        if (pidA < 0) {
                perror("main: fork");
                exit(1);
        }
        if (pidA == 0) {
                /* Child */
```

```
                fork_procsA();
                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(pidA);
        /*show_pstree(getpid());*/



        /* for ask2-signals */
        /* kill(pid, SIGCONT); */

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

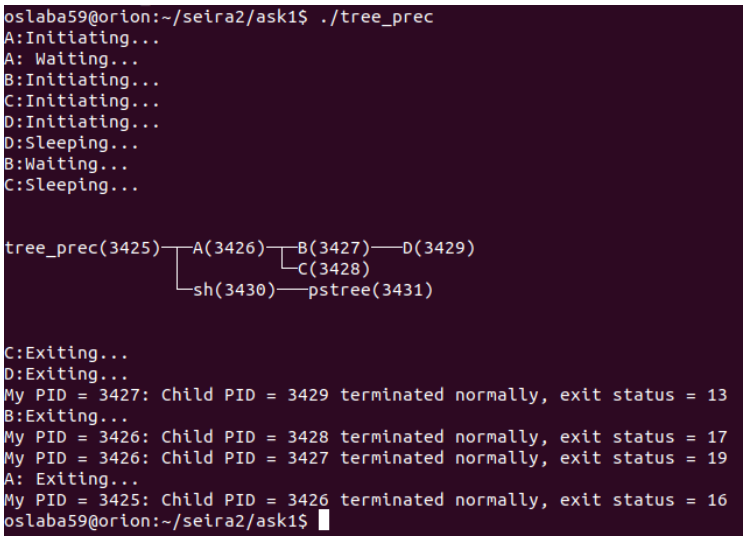
Η έξοδος του προγράμματος φαίνεται παρακάτω:

```
oslaba59@orion:~/seira2/ask1$ ./tree
A:Initiating...
A: Waiting...
B:Initiating...
B:Waiting...
C:Initiating...
C:Sleeping...
D:Initiating...
D:Sleeping...



A(2182)─┬─B(2183)──D(2185)
        └─C(2184)


C:Exiting...
D:Exiting...
My PID = 2183: Child PID = 2185 terminated normally, exit status = 13
B:Exiting...
My PID = 2182: Child PID = 2184 terminated normally, exit status = 17
My PID = 2182: Child PID = 2183 terminated normally, exit status = 19
A: Exiting...
My PID = 2181: Child PID = 2182 terminated normally, exit status = 16
oslaba59@orion:~/seira2/ask1$
```

Ερωτήσεις:

1) Αν τερματιστεί πρόωρα η διεργασία Α δίνοντας **“kill -KILL <pid>”** τότε όλες ο διεργασίες παιδιά της θα μείνουν “ορφανές” (orphan processes) και θα υιοθετηθούν από τη διεργασία init.

2) Η έξοδος του προγράμματος αν αντικαταστήσουμε το **pidA** με το **getpid()** φαίνεται παρακάτω.



Παρατηρούμε ότι με την αλλαγή αυτή εμφανίζονται τρεις παραπάνω διεργασίες. Η πρώτη, η tree_prec είναι η κύριο διεργασία του προγράμματος, ο γονέας του Α, ήτοι, της ρίζας του δέντρου.

Οι άλλες δύο διεργασίες που εμφανίζονται είναι οι sh και η pstree. Η πρώτη είναι η shell process, το κέλυφος/τερματικό που χρησιμοποιούμε για να επικοινωνούμε με το λειτουργικό σύστημα των LINUX. Η δεύτερη είναι η διεργασία pstree η οποίο δημιουργήθηκε με fork() στην shell process έτσι ώστε να τρέξει η συνάρτηση pstree().

3) Σε ένα υπολογιστικό σύστημα πολλαπλών χρηστών, εάν ο διαχειριστής δεν θέσει κάποιο όριο σχετικά με το πλήθος των διεργασίων τότε κάποιος από τους χρήστες θα μπορούσε να δημιουργήσει υπερβολικά πολλές διεργασίες και να υπερκεράσει τους πόρους του συστήματος. Αυτό μπορεί να γίνει πολύ εύκολα μέσω της χρήσης της συνάρτησης fork() μέσα σε έναν ατέρμονα βρόγχο.

Άσκηση 1.2

Παραθέτουμε παρακάτω τον πηγαίο κώδικα της άσκησης 1.2 .

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

#define SLEEP_PROC_SEC 3
```

```
void create_process_tree(struct tree_node* node)
{
int i = 0;
if(node==NULL)
{
return;
}
change_pname(node->name);
pid_t child[node->nr_children];

printf("%s: initiating...\n", node->name);

if(node->nr_children == 0)
{
printf("%s: sleeping...\n", node->name);
sleep(SLEEP_PROC_SEC);
printf("%s: exiting....\n",node->name);
exit(0);
}else
{
do{
pid_t ch;
ch = fork();
if(ch<0)
{
        perror("error:fork");
        exit(1);
}
if(ch==0)
{
        create_process_tree(node->children + i);
}
if(ch>0)
{
child[i] = ch;
++i;

}

}while(i < node->nr_children);

printf("%s: waiting....\n",node->name);
int status;
int j;
for(j=0;j<node->nr_children;++j)
{
pid_t chi = child[j];
waitpid(chi, &status,0);
explain_wait_status(chi, status);
}

printf("%s: exiting....\n",node->name);
```

```
exit(2);
}
}

int main(int argc, char **argv)
{
        struct tree_node *root;
        pid_t pid;
        int status;

        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=fork();
        if(pid<0){
                perror("main: fork");
                exit(1);
        }
        if (pid==0) create_process_tree(root);
        pid= wait(&status);
        explain_wait_status(pid, status);
        return 0;
}
```

Η έξοδος του συστήματος για δύο διαφορετικά αρχεία εισόδου φαίνεται παρακάτω.

```
oslaba59@orion:~/seira2/ask2$ ./tree_creator input
A
        B
                F
        C
                G
                        H
        D
                J
        E
A: initiating...
D: initiating...
A: waiting....
E: initiating...
E: sleeping...
D: waiting....
C: initiating...
B: initiating...
C: waiting....
J: initiating...
J: sleeping...
B: waiting....
G: initiating...
F: initiating...
F: sleeping...
G: waiting....
H: initiating...
H: sleeping...
E: exiting....
J: exiting....
My PID = 3280: Child PID = 3282 terminated normally, exit status = 0
D: exiting....
F: exiting....
H: exiting....
My PID = 3283: Child PID = 3285 terminated normally, exit status = 0
G: exiting....
My PID = 3279: Child PID = 3283 terminated normally, exit status = 2
C: exiting....
My PID = 3278: Child PID = 3284 terminated normally, exit status = 0
B: exiting....
My PID = 3277: Child PID = 3278 terminated normally, exit status = 2
My PID = 3277: Child PID = 3279 terminated normally, exit status = 2
My PID = 3277: Child PID = 3280 terminated normally, exit status = 2
My PID = 3277: Child PID = 3281 terminated normally, exit status = 0
A: exiting....
My PID = 3276: Child PID = 3277 terminated normally, exit status = 2
oslaba59@orion:~/seira2/ask2$
```

```
oslaba59@orion:~/seira2/ask2$ ./tree_creator input2
A
        B
                E
                F
        C
                G
        D
                H
A: initiating...
B: initiating...
A: waiting....
C: initiating...
E: initiating...
E: sleeping...
B: waiting....
C: waiting....
D: initiating...
D: waiting....
G: initiating...
G: sleeping...
H: initiating...
H: sleeping...
F: initiating...
F: sleeping...
E: exiting....
My PID = 3308: Child PID = 3311 terminated normally, exit status = 0
G: exiting....
H: exiting....
F: exiting....
My PID = 3309: Child PID = 3313 terminated normally, exit status = 0
C: exiting....
My PID = 3308: Child PID = 3312 terminated normally, exit status = 0
B: exiting....
My PID = 3310: Child PID = 3314 terminated normally, exit status = 0
D: exiting....
My PID = 3307: Child PID = 3308 terminated normally, exit status = 2
My PID = 3307: Child PID = 3309 terminated normally, exit status = 2
My PID = 3307: Child PID = 3310 terminated normally, exit status = 2
A: exiting....
My PID = 3306: Child PID = 3307 terminated normally, exit status = 2
oslaba59@orion:~/seira2/ask2$
```

Ερωτήσεις:

1) Η σειρά με την οποία εμφανίζονται τα μηνύματα έναρξης ακολουθεί δομή Top-down BFS αλλά στο εσωτερικό του κάθε επίπεδου η σειρά είναι τυχαία και εξαρτάται από το λειτουργικό και το δρομολογητή του. Το ίδιο ισχύει και για τα μηνύματα τερματισμού εκτός του ότι αυτά εμφανίζονται με Bottom-up BFS.

Άσκηση 1.3

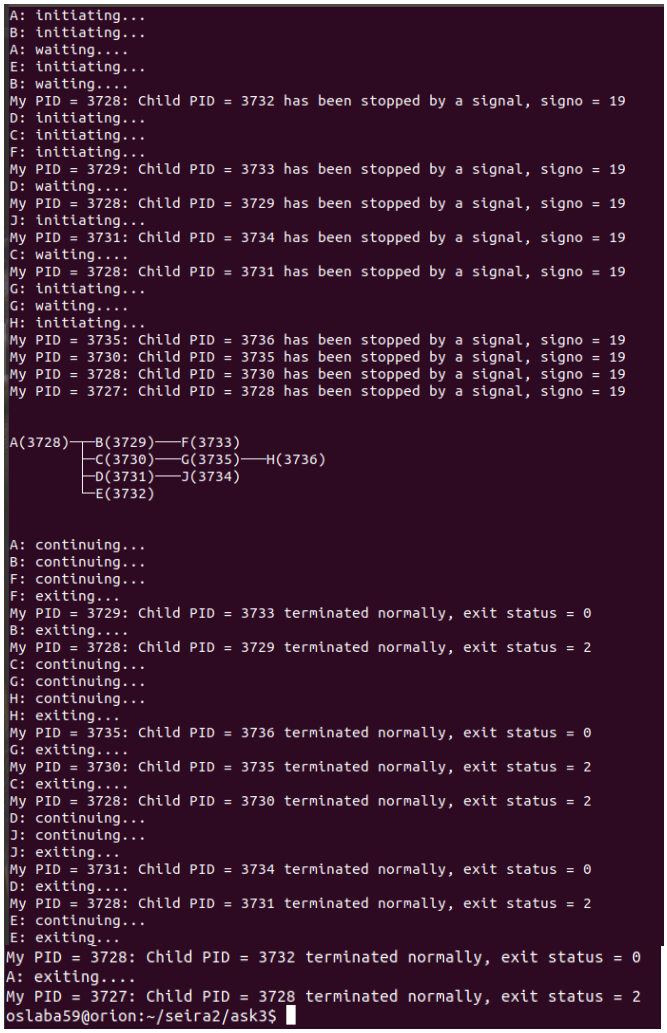
Παραθέτουμε παρακάτω τον πηγαίο κώδικα της άσκησης 1.3 .

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

void create_process_tree(struct tree_node* node)
{
int i = 0, status;
if(node == NULL)
{
return;
}
//pid_t child[node->nr_children];
change_pname(node->name);
printf("%s: initiating... \n", node->name);

if(node->nr_children == 0)
{
//printf("%s: stopping...\n", node->name);
//sleep(SLEEP_PROC_SEC);
raise(SIGSTOP);
printf("%s: continuing... \n",node->name);
printf("%s: exiting...\n",node->name);
exit(0);
}else
{
pid_t child[node->nr_children];
do{
pid_t ch;
ch = fork();
if(ch<0)
{
        perror("error: fork");
        exit(1);
}
```

```
if(ch==0)
{
        create_process_tree(node->children + i);
}
if(ch>0)
{
child[i] = ch;
++i;
}
}while(i < node->nr_children);

printf("%s: waiting....\n",node->name);
int j;
/*for(j=0;j<node->nr_children;++j)
{
pid_t chi = child[j];
int status;
waitpid(chi, &status,0);
}*/
wait_for_ready_children(node->nr_children);
//printf("%s: stopping...\n", node->name);
raise(SIGSTOP);
printf("%s: continuing...\n",node->name);
for(j=0;j<node->nr_children;++j)
{
        kill(child[j], SIGCONT);
//}
//for(j=0;j<node->nr_children;++j){
        child[j]=waitpid(child[j], &status,0);
        explain_wait_status(child[j], status);
}
printf("%s: exiting....\n",node->name);
exit(2);
}
}
int main(int argc, char **argv)
{
        struct tree_node *root;
        pid_t pid;
        int status;

        if (argc != 2) {
                fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
                exit(1);
        }
        root = get_tree_from_file(argv[1]);
        pid = fork();
        //c/hange_pname(root->name);
        //printf("%s: initiating...", root->name);
        if(pid<0)
        {
                perror("main:fork");
```

```
                exit(1);
        }
        if(pid==0)
        {
                create_process_tree(root);

        }
        //printf("%s: waiting...", root->name);
        //Father
        wait_for_ready_children(1);

        //print the process tree root at pid
        show_pstree(pid);

        //send the signal to continue
        kill(pid,SIGCONT);
        pid=wait(&status);
        explain_wait_status(pid, status);
//      print_tree(root);
//      create_process_tree(root);
        return 0;
}
```

Η έξοδος του προγράμματος για ένα ενδεικτικό αρχείο εισόδου φαίνεται παρακάτω.



Ερωτήσεις:

1) Υπάρχουν τρία κύριο πλεονεκτήματα στη χρήση σημάτων για το συγχρονισμό των διεργασιών έναντι της χρήσης της συνάρτησης sleep() .

- Καταρχάς, κερδίζουμε χρόνο εφόσον καμία συνάρτηση πλέον δεν χρησιμοποιεί τη συνάρτηση sleep() με όρισμα ένα αυθαίρετο – και συνήθως πολύ μεγαλύτερο του απαιτούμενου – χρονικά διάστημα.

- Με τη χρήση σημάτων, μας δίνεται ένας βαθμός ελευθερίας σχετικά με τη διάσχιση του δέντρου καθώς μπορούμε να πετύχουμε και Breadth-First-Traversal αλλά και Depth-First-Traversal.

- Τέλος, με τη χρήση σημάτων, εκμηδενίζεται η πιθανότητα να αστοχήσει η συνάρτηση **show_pstree(pid)** .

Άσκηση 1.4

Παραθέτουμε παρακάτω τον πηγαίο κώδικα της άσκησης 1.4 .

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

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

#define SLEEP_TREE_SEC 0.1
#define SLEEP_LEAF 1
void forks(struct tree_node *, int);

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

        if(argc != 2){
                fprintf(stderr,"Usage: ./arithmetical-expression-tree [input file]\n");
                exit(1);
        }

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

```
	if(open(argv[1], O_RDONLY) < 0){
		perror("tree-file error");
	}

	root = get_tree_from_file(argv[1]);

	pid = fork();
	if(pid < 0){
		perror("Initial fork error");
		exit(2);
	}
	if(pid == 0){
		close(fd[0]);
		forks(root, fd[1]);
		exit(0);
	}
	sleep(SLEEP_TREE_SEC);
	show_pstree(pid);
	close(fd[1]);
	if(read(fd[0], &result, sizeof(int)) != sizeof(int)){
		perror("read pipe");
	}

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

	printf("Result: %d\n", result);
	return 0;
}

void forks(struct tree_node *ptr, int pd){
	int i = 0;
	int status, result, *number, temp;
	pid_t pid;

	change_pname(ptr->name);
	int fd[2*ptr->nr_children];

	while(i < ptr->nr_children){
		if(pipe(fd + 2*i) < 0){
			perror("pipe creation");
			exit(1);
		}

		if(!fork()){
			change_pname((ptr->children + i)->name);
			if((ptr->children + i)->nr_children == 0){
				sleep(SLEEP_LEAF);
				int num = atoi((ptr->children + i)->name);

				close(fd[2*i]);
				if(write(fd[2*i + 1], &num, sizeof(int)) != sizeof(int)){
```

```
                        perror("write pipe");
                    }

                    exit(0);
                }
                else{
                    forks(ptr->children + i, fd[2*i + 1]);

                }
            }
            i++;
        }
        number = (int *) malloc (ptr->nr_children * sizeof(int));
        for(i = 0; i < ptr->nr_children; i++){
            if(read(fd[2*i], &temp, sizeof(temp)) != sizeof(temp)){
                perror("read pipe");
            }
            *(number + i)= temp;
        }

        switch(strcmp(ptr->name,"+")) {
            case 0:
                result = *(number + 1) + *number;
                break;
            default:
                result = *(number + 1) * *number;
        }

        if(write(pd, &result, sizeof(int)) != sizeof(int)){
            perror("write pipe");
        }
        for (i=0; i<ptr->nr_children; i++){
            pid = wait(&status);
            explain_wait_status(pid, status);
        }
        exit(0);
}
```

Η έξοδος του προγράμματος για το αρχείο εισόδου της εκφώνησης φαίνεται παρακάτω .

```
oslaba59@orion:~/seira2/ask4$ ./result expr.tree


+(4416)─┬─*(4419)─┬─+(4420)─┬─5(4422)
        │         │         └─7(4423)
        │         └─4(4421)
        └─10(4418)


My PID = 4420: Child PID = 4422 terminated normally, exit status = 0
My PID = 4420: Child PID = 4423 terminated normally, exit status = 0
My PID = 4419: Child PID = 4420 terminated normally, exit status = 0
My PID = 4419: Child PID = 4421 terminated normally, exit status = 0
My PID = 4416: Child PID = 4418 terminated normally, exit status = 0
My PID = 4416: Child PID = 4419 terminated normally, exit status = 0
My PID = 4415: Child PID = 4416 terminated normally, exit status = 0
Result: 58
oslaba59@orion:~/seira2/ask4$
```

Ερωτήσεις:

1) Εφόσον στη συγκεκριμένη άσκηση πραγματευόμαστε μονάχα προσθέσεις και πολλαπλασιασμούς, που είναι αντιμεταθετικές πράξεις, είναι δυνατόν να χρησιμοποιήσουμε και ένα μόνο pipe για κάθε τριάδα γονιού και δύο παιδιών. Εάν έπρεπε να εκτελέσουμε αφαιρέσεις και διαιρέσεις που είναι πράξεις μη αντιμεταθετικές, τότε θα έπρεπε να έχομε δύο pipes: ένα σωλήνα για κάθε γονιό και ένα για κάθε παιδί. Αυτό θα μας εξασφάλιζε το ποιος είναι ο κάθε τελεστέος που επιστρέφουν τα παιδιά διεργασίες στον πατέρα.

2) Σε ένα σύστημα πολλών επεξεργαστών έχουμε τη δυνατότητα να αναθέσουμε κάθε διεργασία "πατέρα" σε διαφορετικό πυρήνα ξεκινώντας από το επίπεδο αμέσως πάνω από τα φύλλα και κινούμενοι προς τη ρίζα του δέντρου. Έτσι, οι πράξεις ίδιου επιπέδου αλλά διαφορετικού "κλάδιού" γίνονται παράλληλα, με καλύτερη περίπτωση (ισοζυγισμένο AVL tree) log2(t) όπου t ο χρόνος για να γίνουν οι πράξεις γραμμικά. Βέβαια, στη χειρότερη περίπτωση, αυτή του εκφυλισμένου δέντρου, ο χρόνος που απαιτείται είναι ίσος με τον χρόνο για τον γραμμικό υπολογισμό (με έναν πυρήνα) .