Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
6^{Ο} Εξάμηνο: Λειτουργικά Συστήματα 2022-23
Δανάη Σπέντζου (03120237)
Νεκτάριος Μπούμπαλος (03120441)

Άσκηση 2.1: Δημιουργία δεδομένου δέντρου διεργασιών

```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;
        pid_t p, pidB, pidC, pidD;
        fprintf(stderr, "A creating B...\n");
        pidB=fork();

        if (pidB<0){
                perror("main: fork"); exit(1);
        }
        if (pidB == 0){
                fprintf(stderr, "B creating D...\n");
                pidD=fork();
                if (pidD < 0) {
                        perror("main: fork");
                        exit(1);
                }
                if (pidD == 0) {
                        change_pname("D");
                        printf("D: Sleeping...\n");
                        sleep(SLEEP_PROC_SEC);
                        printf("D: Exiting...\n");
                        exit(13); //exits D
                }
                else{
                        change_pname("B");
                        p = wait(&status);
                        explain_wait_status(pidD, status);
                        printf("B: Exiting...\n");
                        exit(19); //exits B
                }
        }
        else{
                fprintf(stderr, "A creating C...\n");
                pidC=fork();
                if (pidC < 0) {
                        perror("main: fork");
                        exit(1);
                }
                if (pidC == 0) {
                        change_pname("C");
                        printf("C: Sleeping...\n");
                        sleep(SLEEP_PROC_SEC);
                        printf("C: Exiting...\n");
                        exit(17); //exits D
                }
                else{
                        change_pname("A");
                        p = wait(&status);
                        explain_wait_status(p, status);
                        p = wait(&status);
                        explain_wait_status(p, status);
                        printf("A: Exiting...\n");
                        exit(16); //exits B


        }
}
```

```c
/*
 * 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;
}
```

```
oslab162@orion:~/exercise2/forktree$ make ask2-fork
gcc -g -Wall -O2 -c ask2-fork.c
gcc -g -Wall -O2 ask2-fork.o proc-common.o -o ask2-fork
oslab162@orion:~/exercise2/forktree$ ./ask2-fork
A creating B...
A creating C...
B creating D...
C: Sleeping...
D: Sleeping...


A(6717)─┬─B(6718)──D(6720)
        └─C(6719)


C: Exiting...
My PID = 6717: Child PID = 6719 terminated normally, exit status = 17
D: Exiting...
My PID = 6718: Child PID = 6720 terminated normally, exit status = 13
B: Exiting...
My PID = 6717: Child PID = 6718 terminated normally, exit status = 19
A: Exiting...
My PID = 6716: Child PID = 6717 terminated normally, exit status = 16
```

Ερωτήσεις:

1. Αν τερματίσουμε πρόωρα τη διεργασία A δίνοντας kill -KILL <pid> όπου pid το process id της τότε θα τερματιστούν ακαριαία όλες οι διεργασίες του δέντρου (όπου root A) χωρίς να πραγματοποιηθούν οι απαραίτητες διεργασίες που γίνονται πριν από κάθε έξοδό (clean up), οπότε μπορεί να προκληθούν προβλήματα στο σύστημα.
2. Αν κάνουμε show_pstree(getpid()) αντί για show_pstree(pid) στη main() τότε θα εκτυπωθεί ένα δέντρο που θα ξεκινά από τη διεργασία που δημιουργείται από τον κώδικα της main() και θα περιλαμβάνει τις διεργασίες που δημιουργούνται από την fork_procs().

   Επιπλέον διεργασίες που φαίνονται στο δέντρο:

   - **ask2-fork** process: είναι το parent process του προγράμματος. Όταν το πρόγραμμα εκτελείται το shell εκκινεί τη διεργασία ask2-fork και αυτή η διεργασία δημιουργεί ένα αντίγραφο της ρίζας του δέντρου διεργασιών (fork: process tree). Στη συνέχεια, η διεργασία ask2-fork περιμένει τη ρίζα να τερματίσει το πρόγραμμα, να εκτυπώσει το δέντρο και τέλος τερματίζεται.
   - **Sh process**: είναι το shell που χρησιμοποιείται για την εκτέλεση του προγράμματος. Εκκινεί μια νέα διεργασία για να εκτελέσει το πρόγραμμα, η διεργασία γίνεται η διεργασία ask2-fork και περιμένει να τερματίσει. Μόλις η διεργασία ask2-fork τερματίσει, το shell συνεχίζει και εκτυπώνει μια νέα γραμμή εντολών.

```
oslab162@orion:~/exercise2/forktree$ ./ask2-fork
A creating B...
A creating C...
B creating D...
C: Sleeping...
D: Sleeping...
```

```
ask2-fork(25893)─┬─A(25894)─┬─B(25895)──D(25897)
                 │          └─C(25896)
                 └─sh(25900)──pstree(25901)
```

```
C: Exiting...
My PID = 25894: Child PID = 25896 terminated normally, exit status = 17
D: Exiting...
My PID = 25895: Child PID = 25897 terminated normally, exit status = 13
B: Exiting...
My PID = 25894: Child PID = 25895 terminated normally, exit status = 19
A: Exiting...
My PID = 25893: Child PID = 25894 terminated normally, exit status = 16
```

3. Σε υπολογιστικά συστήματα πολλαπλών χρηστών, πολλές φορές ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης. Όταν ο χρήστης δημιουργεί πολλές διεργασίες τότε επιβαρύνει σημαντικά την μνήμη, τον επεξεργαστή, την ασφάλεια του συστήματος και προκαλεί προβλήματα στην απόδοση του και στην απόκριση σε άλλους χρήστες.

Άσκηση 2.2 : Δημιουργία αυθαίρετου δέντρου διεργασιών

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

#define SLEEP_TREE_SEC 3
#define SLEEP_CHILD 10

/* Question 2.2 */


void forks(struct tree_node *);

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

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

        root = get_tree_from_file(argv[1]);

        //create root
        pid = fork();
        if(pid < 0){
                perror("Error");
                exit(0);
        }
        if(pid == 0){
                forks(root);
                exit(0);
        }

        sleep(SLEEP_TREE_SEC);
        show_pstree(pid);

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

        return 0;
}

void forks(struct tree_node *t){
        int i=0;
        int status;
        pid_t pid;

        change_pname(t->name);

        while(i < t->nr_children){
                printf("Node %s has to generate %d children\n", t->name, t->nr_children - i);
                if(!fork()){
                        change_pname((t->children + i)->name);
                        if((t->children + i)->nr_children == 0){
                                printf("%s:Sleeping...\n", (t->children + i)->name);
                                sleep(SLEEP_CHILD);
                                printf("%s:Exiting now...\n", (t->children + i)->name);
                                exit(0);
                        }
                        else{
                                forks(t->children+i);
                        }
                }
                i++;
        }
        for (i=0; i<t->nr_children; i++){
                pid = wait(&status);
                explain_wait_status(pid, status);
        }
        exit(0);
}
```

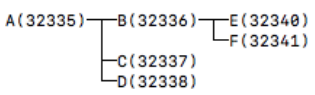
```
oslab162@orion:~/exercise2/forktree$ ./tree-example proc.tree
Node A has to generate 3 children
Node A has to generate 2 children
Node A has to generate 1 children
D:Sleeping...
Node B has to generate 2 children
Node B has to generate 1 children
F:Sleeping...
C:Sleeping...
E:Sleeping...


A(31012)─┬─B(31013)─┬─E(31016)
         │          └─F(31017)
         ├─C(31014)
         └─D(31015)


D:Exiting now...
My PID = 31012: Child PID = 31015 terminated normally, exit status = 0
F:Exiting now...
My PID = 31013: Child PID = 31017 terminated normally, exit status = 0
C:Exiting now...
E:Exiting now...
My PID = 31012: Child PID = 31014 terminated normally, exit status = 0
My PID = 31013: Child PID = 31016 terminated normally, exit status = 0
My PID = 31012: Child PID = 31013 terminated normally, exit status = 0
My PID = 31011: Child PID = 31012 terminated normally, exit status = 0
```

Τα μηνύματα έναρξης και τερματισμού εκτυπώνονται με τη σειρά που δημιουργούνται και τερματίζονται από τη συνάρτηση forks(). Για τα μηνύματα έναρξης, αυτά ξεκινάνε από τη ρίζα του δέντρου(A) και στη συνέχεια στα παιδιά της με τη σειρά που αυτά δημιουργούνται, αντίστοιχα για τα μηνύματα τερματισμού. Η σειρές αυτές οφείλονται κυρίως στη συνάρτηση fork() και στον τρόπο που εκτελεί κάθε φορά τις νέες διεργασίες, δηλαδή δημιουργεί αντίγραφα και συνεχίζει κάθε φορά από το σημείο που είχε βρισκόταν η αρχική.

Άσκηση 2.3: Αποστολή και χειρισμός σημάτων

```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"

#define SLEEP_TREE_SEC 1

/* Question 2.3 */

int got_signal=0;

void sighandler(){
        got_signal=1;
}

void fork_procs(struct tree_node *root){

        int i=0, status;
        pid_t pid, cpid[root->nr_children];

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

        while(i < root->nr_children){
                cpid[i]=fork();
                if(cpid[i]==0){
                        change_pname((root->children + i)->name);
                        if((root->children + i)->nr_children == 0){
                                signal(SIGCONT, sighandler);
                                raise(SIGSTOP);
                                printf("%s\n", (root->children + i)->name);
                                exit(0);
                        }
                        else
                                fork_procs(root->children + i);
                }
                i++; //next child
        }

        signal(SIGCONT, sighandler);
        raise(SIGSTOP);
        printf("PID = %ld, name = %s is awake\n",(long)getpid(), root->name);

        i=0;
        while(i < root->nr_children){
                kill(cpid[i], SIGCONT);
                pid = wait(&status);
                explain_wait_status(pid, status);
                i++;
        }

        printf("%s\n", (root->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_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;
}
```

```
oslab162@orion:~/exercise2/forktree$ vi ask2-signals.c
oslab162@orion:~/exercise2/forktree$ make ask2-signals
gcc -g -Wall -O2 -c ask2-signals.c
gcc -g -Wall -O2 ask2-signals.o proc-common.o tree.o -o ask2-signals
oslab162@orion:~/exercise2/forktree$ ./ask2-signals proc.tree
PID = 32335, name A, starting...
My PID = 32334: Child PID = 32335 has been stopped by a signal, signo = 19
PID = 32336, name B, starting...
```




```
PID = 32335, name = A is awake
PID = 32336, name = B is awake
E
My PID = 32336: Child PID = 32340 terminated normally, exit status = 0
F
My PID = 32336: Child PID = 32341 terminated normally, exit status = 0
B
My PID = 32335: Child PID = 32336 terminated normally, exit status = 0
C
My PID = 32335: Child PID = 32337 terminated normally, exit status = 0
D
My PID = 32335: Child PID = 32338 terminated normally, exit status = 0
A
My PID = 32334: Child PID = 32335 terminated normally, exit status = 0
```

Ερωτήσεις:

1. Στις προηγούμενες ασκήσεις χρησιμοποιήσαμε τη sleep() για τον συγχρονισμό των διεργασιών. Όμως, η χρήση σημάτων πλεονεκτεί σε σύγκριση με τη πρώτη μέθοδο, διότι δεν απαιτεί την εκτίμηση χρονικών διαστημάτων αναμονής, αλλά η εκκίνηση και ο τερματισμός πραγματοποιούνται αυτόματα με την εκτέλεση εντολών και οι διαφορετικές διεργασίες μπορούν έτσι να επικοινωνούν μεταξύ τους μέσα από τα σήματα και να συντονίζονται. Παρ ’όλα αυτά, τα σήματα είναι αρκετά ευάλωτα ως προς τη σειρά προτεραιότητας τους και απαιτούν προσοχή στη χρήση τους.

2. Η wait_for_ready_children() χρησιμοποιείται για τον συγχρονισμό των διεργασιών ενός δέντρου. Επιστρέφει όταν οι διεργασίες όλων των παιδιών έχουν δημιουργηθεί και είναι έτοιμες να τερματιστούν, όπου καλείται η εντολή raise(SIGSTOP). Αν παραληφθεί, τότε υπάρχει πιθανότητα οι διεργασίες να τερματίζονται με την raise(SIGSTOP) πριν ακόμη ολοκληρωθούν για όλα τα παιδιά του δέντρου και η συμπεριφορά του προγράμματος να είναι απρόβλεπτη, αφού οι διεργασίες δε θα είναι συγχρονισμένες.

Άσκηση 2.4: Παράλληλος υπολογισμός αριθμητικής έκφρασης

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

#define SLEEP_CHILD  1
#define SLEEP_TREE_SEC  0.1



void math_fork(struct tree_node *root, int fd1)
{
        pid_t pid;
        int status;

        change_pname(root->name);
        if(root->nr_children==0)
        {
                int child_value=atoi(root->name);
                if(write(fd1,&child_value,sizeof(child_value))!=sizeof(child_value)){
                        perror("write to pipe");
                        exit(1);
                }

                close(fd1);
                sleep(SLEEP_CHILD);
                exit(1);
        }
        else{
                int pipe_fd[2];
                if(pipe(pipe_fd)==-1){
                        perror("pipe");
                        exit(1);
                }
                int i;
                for(i=0;i<root->nr_children;i++){
                        pid=fork();
                        if(pid<0)
                        {
                                perror("fork");
                                exit(1);
                        }
                        else if(pid==0){

                                close(pipe_fd[0]);
                                math_fork(root->children+i,pipe_fd[1]);
                                }
                }
                close(pipe_fd[1]);

                int value[2];
                for(i=0;i<root->nr_children;i++){
                        if(read(pipe_fd[0], &value[i],sizeof(value[i]))!=sizeof(value[i])){
                                perror("read from pipe");
```

```
                        exit(1);
                }

        }

        int result;
        if(strcmp(root->name,"+")==0){
                result=value[0]+value[1];}
        else if(strcmp(root->name,"*")==0){
                result=value[0]*value[1];
        }

        if(write(fd1,&result,sizeof(result))!=sizeof(result)){
                perror("write to pipe");
                exit(1);
        }
        close(fd1);
        for(i=0;i<root->nr_children;i++){
                if(wait(&status)==-1){
                        perror("wait");
                        exit(1);
                }
                explain_wait_status(pid,status);
        }
        exit(1);


    }

}

int main(int argc, char **argv){

    struct tree_node *root;
    pid_t pid;

    int status, result, fd[2];

    if(argc != 2){
            fprintf(stderr,"Usage: ./math-fork [input file]\n");
            exit(1);
    }
    if(open(argv[1], O_RDONLY) < 0)
            perror("tree-file error");

    root = get_tree_from_file(argv[1]);

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

    /* Fork root of process tree */
    pid = fork();
    if (pid < 0) {
            perror("main: fork");
            exit(2);
    }
    if (pid == 0) {
            /* Child */
            close(fd[0]);
            math_fork(root,fd[1]);
            exit(0);
     }

     /* Father */
     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;
}
```

```
oslab162@orion:~/exercise2/forktree$ ./math-fork expr.tree

+(3607)─┬─*(3610)─┬─+(3612)─┬─5(3614)
        │         │         └─7(3615)
        │         └─4(3613)
        └─10(3609)

My PID = 3607: Child PID = 3610 terminated normally, exit status = 1
My PID = 3610: Child PID = 3613 terminated normally, exit status = 1
My PID = 3612: Child PID = 3615 terminated normally, exit status = 1
My PID = 3612: Child PID = 3615 terminated normally, exit status = 1
My PID = 3610: Child PID = 3613 terminated normally, exit status = 1
My PID = 3607: Child PID = 3610 terminated normally, exit status = 1
My PID = 3606: Child PID = 3607 terminated normally, exit status = 1
Result: 58
```

Ερωτήσεις:

1. Η συγκεκριμένη άσκηση χρησιμοποιεί μία σωλήνωση. Η γονική διεργασία δημιουργεί μία σωλήνωση και αυτή χρησιμοποιείται για να πάρει το τελικό αποτέλεσμα από το παιδί της που εκτελεί τον υπολογισμό του μαθηματικού δέντρου.

   Μία γονική διεργασία δε θα μπορούσε να χρησιμοποιήσει μόνο μία σωλήνωση για όλα τα παιδιά της. Αυτό οφείλεται στο γεγονός ότι οι σωληνώσεις είναι αμφίδρομες και κάθε φορά περιλαμβάνουν έναν συγγραφέα που κάνει write και έναν αναγνώστη που κάνει read αφού πρώτα τερματιστεί η write(). Αν μια διεργασία προσπαθήσει να γράψει στη σωλήνωση ενώ κάποια άλλη διεργασία διαβάζει από αυτήν τη σωλήνωση, τότε μπορεί να προκληθεί σύγκρουση. Για αυτό το λόγο, κάθε παιδί πρέπει να έχει τη δική του σωλήνωση με τη γονική διεργασία, προκειμένου να αποφεύγονται παρόμοιες συγκρούσεις και να διασφαλίζεται η σωστή επικοινωνία μεταξύ των διεργασιών.

   Γενικά για κάθε αριθμητικό τελεστή μπορεί να χρησιμοποιηθεί μόνο μια σωλήνωση. Αυτό οφείλεται στο γεγονός ότι η τιμή που επιστρέφεται από την κάθε διεργασία περιέχει όλη την απαραίτητη πληροφορία που απαιτείται για την εκτέλεση της αντίστοιχης αριθμητικής πράξης. Έτσι, ο πατέρας μπορεί να χρησιμοποιήσει αυτές τις τιμές για να εκτελέσει την αντίστοιχη αριθμητική πράξη, χωρίς την ανάγκη νέων σωληνώσεων.

2. Σε ένα σύστημα πολλαπλών επεξεργαστών που μπορούν να εκτελούνται παραπάνω από μια διεργασίες παράλληλα, η αποτίμηση μιας έκφρασης από δέντρο διεργασιών μπορεί να έχει το πλεονέκτημα των παράλληλων υπολογισμών. Αν υπάρχουν αρκετοί επεξεργαστές στο σύστημα, μπορούν να εκτελούνται πολλές διεργασίες του δέντρου ταυτόχρονα, αυξάνοντας την απόδοση του συστήματος. Αντίθετα, η επεξεργασία κάθε διεργασίας γίνεται σειριακά και ο επεξεργαστής πρέπει να περιμένει το τέλος της μίας πράξης για να προχωρήσει στην επόμενη.