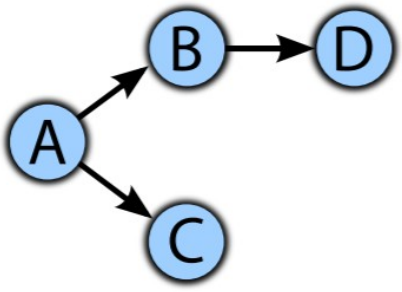
Λειτουργικά Συστήματα
6ο Εξάμηνο
Άσκηση 2: Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία
Αναφορά

Χαρδούβελης Γεώργιος-Ορέστης
El15100
6ο Εξάμηνο

Κυριάκου Αθηνά
el17405
6ο Εξάμηνο

Άσκηση 1

Σε αυτή την άσκηση καλούμαστε να κατασκευάσουμε το παρακάτω δέντρο διεργασιών.



Σχήμα 1: Δέντρο διεργασιών

Ο πηγαίος κώδικας της άσκησης έχει ως εξής:

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

#include "proc-common.h"

#define SLEEP_PROC_SEC  10  //sleeping time for the leaves
#define SLEEP_TREE_SEC  3   //sleeping time for the root

void exit_func(const char *name){
        if(strcmp(name,"A")==0) exit(16);
        else if(strcmp(name,"B")==0) exit(19);
        else if(strcmp(name,"C")==0) exit(17);
```

```
        else exit(13);
}

void fork_procs(const char *name){ //function for leaves

         change_pname(name);

        printf("%s: Sleeping...\n",name);
        sleep(SLEEP_PROC_SEC);

        printf("%s: Exiting...\n",name);
        exit_func(name);
}

/*
 * 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.*/

int main(void){

        pid_t pid;
        int status;

        printf("A: Starting...\n");
        pid = fork(); //init->A

        if (pid < 0){
                perror("main: fork");
                exit(1);
        }
        if (pid == 0){
                /* A Child */
                //fork_procs('A');

                change_pname("A");
                int pidA_B,pidA_C;
                printf("B: Starting...\n");
                pidA_B=fork(); //A->B
```

```
if(pidA_B<0){
	perror("A_B: fork");
	exit(1);
}

else if(pidA_B==0){ //do stuff for child B

	change_pname("B");
	printf("D: Starting...\n");
	int pidB_D=fork();

	if(pidB_D<0){
		perror("B_D: fork");
		exit(1);
	}

	else if(pidB_D==0){ //dostuff for the child D
		fork_procs("D");
	}

	//father B
	int statusD; //status of  process D
	printf("B: Waiting for D to finish if not already finished...\n");
	pidB_D=wait(&statusD);
	explain_wait_status(pidB_D, statusD);

	printf("B: Exiting...\n");
	exit_func("B");

}

}

printf("C: Starting...\n");
pidA_C=fork(); //A->C

if(pidA_C<0){
	perror("A_C: fork");
	exit(1);
}
else if(pidA_C==0){ //dostuff for child C
	fork_procs("C");
}
```

```
        int statusB,statusC;

        printf("A: Waiting for B to finish if not already finished...\n");
        pidA_B=waitpid(pidA_B,&statusB,0);
        explain_wait_status(pidA_B,statusB);

        printf("A: Waiting for C to finish if not already finished...\n");
        pidA_C=waitpid(pidA_C,&statusC,0);
        explain_wait_status(pidA_C,statusC);

        printf("A: Exiting...\n");
        exit_func("A");
    }

    printf("Init: Sleeping...\n");
    sleep(SLEEP_TREE_SEC);

    /* Print the process tree root at pid */
    printf("Printing the tree!\n");
    show_pstree(pid);
    //show_pstree(getpid());

    /* Wait for the root of the process tree to terminate */
    printf("Init: Waiting for A to finish if not already finished...\n");
    pid = wait(&status);
    explain_wait_status(pid, status);

    return 0;
}
```

Όπως βλέπουμε κάνουμε include και την βιβλιοθήκη proc-common που μας δίνεται. Η δημιουργία του εκτελέσιμου γίνεται μέσω του Makefile που έχει την παρακάτω μορφή:

```
all: ask2-fork

ask2-fork: ask2-fork.o proc-common.o
    gcc -o ask2-fork ask2-fork.o proc-common.o

ask2-fork.o: ask2-fork.c
    gcc -Wall -c ask2-fork.c

proc-common.o: proc-common.c
    gcc -Wall -c proc-common.c
```

clean:

rm -f ask2-fork ask2-fork.o proc-common.o

Όταν τρέξουμε το Makefile και στην συνέχεια και το έκτελέσιμο πρόγραμμα μας, έχουμε το εξής output:

```
A: Starting...
Init: Sleeping...
B: Starting...
C: Starting...
D: Starting...
A: Waiting for B to finish if not already finished...
C: Sleeping...
B: Waiting for D to finish if not already finished...
D: Sleeping...
Printing the tree!


A(26117)─┬─B(26118)──D(26120)
         └─C(26119)


Init: Waiting for A to finish if not already finished...
C: Exiting...
D: Exiting...
My PID = 26118: Child PID = 26120 terminated normally, exit status = 13
B: Exiting...
My PID = 26117: Child PID = 26118 terminated normally, exit status = 19
A: Waiting for C to finish if not already finished...
My PID = 26117: Child PID = 26119 terminated normally, exit status = 17
A: Exiting...
My PID = 26116: Child PID = 26117 terminated normally, exit status = 16
oslabdl4@os-node1:~/lab2/ex1_1$
```

Βλέπουμε πως το δέντρο που εκτυπώνεται είναι το επιθυμητό και οι κωδικοί επιστροφής κάθε συνάρτησης είναι οι ζητούμενοι από την εκφώνηση.

Γενικά κάθε “πατέρας” δημιουργεί με την συνάρτηση fork() τα παιδιά του (αναδρομικά) και στο τέλος κάνουμε sleep τα φύλλα και τυπώνουμε το δέντρο. Ύστερα από συγκεκριμένο χρονικό διάστημα συνεχίζεται η αναδρομή και τερματίζουν μία μία οι διεργασίες, αφού πρώτα τερματιστούν τα παιδιά τους.

Ερωτήσεις:

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

Στην παραπάνω περίπτωση, η διεργασία θα τερμαστιστεί πριν τερματιστούν τα παιδιά της. Έτσι, τα παιδιά της θα γίνουν παιδιά της init η οποία κάνει συνεχώς wait και το πρόγραμμα θα συνεχίσει κανονικά.

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

Εάν τρέξουμε την εντολή show_pstree(getpid()) έναντι της show_pstree(pid) στη main() τότε αντί να πάρει το pid της A διεργασίας που είχε οριστεί παραπάνω θα πάρει το pid της διεργασίας που βρίσκεται εκείνη τη στιγμή σε εκτέλεση, δηλαδή της init. Το output φαίνεται στην παρακάτω εικόνα.

```
A: Starting...
Init: Sleeping...
B: Starting...
C: Starting...
D: Starting...
A: Waiting for B to finish if not already finished...
C: Sleeping...
B: Waiting for D to finish if not already finished...
D: Sleeping...

Printing the tree!


ask2-fork(25927)─┬─A(25928)─┬─B(25929)──D(25931)
                 │          └─C(25930)
                 └─sh(25932)──pstree(25933)


Init: Waiting for A to finish if not already finished...
C: Exiting...
D: Exiting...
My PID = 25929: Child PID = 25931 terminated normally, exit status = 13
B: Exiting...
My PID = 25928: Child PID = 25929 terminated normally, exit status = 19
A: Waiting for C to finish if not already finished...
My PID = 25928: Child PID = 25930 terminated normally, exit status = 17
A: Exiting...
My PID = 25927: Child PID = 25928 terminated normally, exit status = 16
```

Έτσι πέρα από το υποδέντρο με κορυφή A η οποία είναι παιδί της init, έχουμε και το sh, που υποδηλώνει το Bourne Shell, το προεπιλεγμένο κέλυφος για Unix και λειτουργεί ως διερμηνέας της γραμμής εντολών. Το pstree που φαίνεται παρακάτω προκύπτει από την συνάρτηση show_pstree() (ο λόγος που εμφανίζει pstree είναι η κωδική λέξη echo. Αυτό φαίνετα και από την κώδικα υλοποίησης της συνάρτησης παρακάτω).

```
void
show_pstree(pid_t p)
{
        int ret;
        char cmd[1024];

        snprintf(cmd, sizeof(cmd), "echo; echo; pstree -G -c -p %ld; echo; echo",
                (long)p);
        cmd[sizeof(cmd)-1] = '\0';
        ret = system(cmd);
        if (ret < 0) {
                perror("system");
                exit(104);
        }
}
```

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

Ο λόγος που θέτονται όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης υπολογιστικού συστήματος πολλαπλών χρηστών είναι για να διασφαλιστεί η επάρκεια τον πόρων του. Κάθε θυγατρική διεργασία μίας αρχικής μπορεί είτε να χρησιμοποιεί μέρος ή το σύνολο των πόρων της πατρικής της είτε να της διαμοιράζονται πόροι απευθείας από το λειτουργικό σύστημα. Έτσι, αν ένας χρήστης δημιουργεί διεργασίες χωρίς να υπάρχει κάποιο όριο, υπάρχει κίνδυνος να ξεπεράσει τις δυνατότητες του συστήματος ή να χρησιμοποιήσει πολύ μεγάλο μέρος των πόρων εις βάρος των άλλων χρηστών.

Άσκηση 2

Σε αυτή την άσκηση ζητήθηκε η κατασκευή ενός αυθαίρετου δέντρου διεργασιών, βασισμένη σε αναδρομική συνάρτηση. Το δέντρο που δημιουργείται είναι αυτό που περιγράφεται σε ένα εξωτερικό αρχείο.

Ο πηγαίος κώδικας της άσκησης έχει ως εξής:

```
#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(const char *name)  //function for leaves
{
	change_pname(name);
	printf("%s: Sleeping...\n",name);
	sleep(SLEEP_PROC_SEC);
	printf("%s: Exiting...\n",name);
	exit(1);
}

static void fork_rec(struct tree_node *root){
	if(root->children==NULL){
		fork_procs(root->name); //if a leaf sleep
		return;
	}
	 else{
		change_pname(root->name);
		int i, pid, status;
		for(i=0; i<root->nr_children; i++){
			printf("%s: Starting...\n",(root->children+i)->name);
			pid=fork();
			if(pid<0){
				perror("fork");
				exit(1);
```

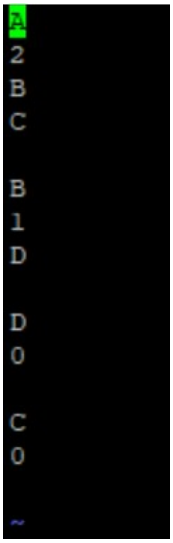
```
                }
                if(pid==0){
                        fork_rec(root->children+i);
                }
        }

        for(i=0;i<root->nr_children; i++){
                printf("%s: Waiting for %s to terminate if not already terminated...\n",root->name,
(root->children+i)->name);
                pid=wait(&status);
                explain_wait_status(pid,status);
        }
        printf("%s: Exiting...\n",root->name);
        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]); //read tree from file

    pid_t pid;
    int status;

    printf("Printing the process tree from file!\n");
    print_tree(root);

    /* Fork root of process tree */
    printf("%s: Starting...\n", root->name);
    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
```

```
            /* Child */
            fork_rec(root);
            exit(1);
        }

        /* Father */
        /* for ask2-{fork, tree} */
        printf("Init: Sleeping...\n");
        sleep(SLEEP_TREE_SEC);

        /* Print the process tree from recursive */
        printf("Printing the process tree from recursive!\n");
        show_pstree(pid);

        /* Wait for the root of the process tree to terminate */
        printf("Init: Waiting for %s to terminate if not already terminated...\n",root->name);
        pid = wait(&status);
        explain_wait_status(pid, status);

        return 0;
}
```

Ομοίως με την παραπάνω άσκηση, το Makefile έχει ως εξής:

```
all: ask2-fork

ask2-fork: ask2-fork.o tree.o proc-common.o
        gcc -o ask2-fork ask2-fork.o tree.o proc-common.o

ask2-fork.o: ask2-fork.c
        gcc -c ask2-fork.c

tree.o: tree.c
        gcc -c tree.c

proc-common.o: proc-common.c
        gcc -c proc-common.c

clean:
        rm -f tree.o ask2-fork.o proc-common.o ask2-fork
```

Έστω πως θέλουμε να δημιουργήσουμε το δέντρο που δίνεται στην εκφώνηση. Στο αρχείο (ονομαζόμενο test) περιγράφεται ως εξής:



Αφού τρέξουμε το πρόγραμμα με την εντολή ./ask2-fork test έχουμε το παρακάτω output:

```
oslabd14@os-node1:~/lab2/ex1_2$ ./ask2-fork test
Printing the process tree from file!
A
        B
                D
        C
A: Starting...
Init: Sleeping...
B: Starting...
C: Starting...
D: Starting...
A: Waiting for B to terminate if not already terminated...
C: Sleeping...
B: Waiting for D to terminate if not already terminated...
D: Sleeping...
Printing the process tree from recursive!


A(26348)─┬─B(26349)──D(26351)
         └─C(26350)


Init: Waiting for A to terminate if not already terminated...
C: Exiting...
D: Exiting...
My PID = 26348: Child PID = 26350 terminated normally, exit status = 1
A: Waiting for C to terminate if not already terminated...
My PID = 26349: Child PID = 26351 terminated normally, exit status = 1
B: Exiting...
My PID = 26348: Child PID = 26349 terminated normally, exit status = 1
A: Exiting...
My PID = 26347: Child PID = 26348 terminated normally, exit status = 1
oslabd14@os-node1:~/lab2/ex1_2$
```

Το δέντρο εκτυπώνεται αρχικά από το αρχείο με την print_tree() και στη συνέχεια βάσει της αναδρομής με την show_pstree().

Ερωτήσεις:

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

Τα μηνύματα έναρξης εμφανίζονται με την σειρά: A, B, C, D και τα μηνύματα τερματισμού C, D, B, A.

Αυτό εξηγείται βάσει της υλοποίησης της αναδρομικής συνάρτηση static void fork_rec(struct tree_node *root). Πιο συγκεκριμένα, συνάρτηση μπαίνει στο for loop όπου γίνεται fork() τόσες φορές όσες και ο αριθμός των παιδιών της καλούσας διεργασίας, αν αυτή δεν είναι φύλλο. Για

καθένα από τα παιδιά καλείται ξανά η διεργασία. Αφού τελειώσουν τα παραπάνω η συνάρτηση συνεχίζει και πραγματοποιείται wait από την καλούσα διεργασία μέχρι να τερματιστούν τα παιδιά της. Ύστερα τερματίζει και αυτή.

Στην περίπτωση που η καλούσα διεργασία είναι φύλλο, αυτή καλεί τη συνάρτηση sleep() για 10sec.

Έτσι αρχικά γίνεται δημιουργείται η A και αφού καλεστεί η συνάρτηση δημιουργείται και η B μέσω της πρώτης fork(), ενώ καλείται ξανά η συνάρτηση μας για την B. Επομένως ξεκινάει η διαδικασία για να γίνει fork() στην B και να δημιουργηθεί η D διεργασία, αλλά μέχρι να δημιουργηθεί έχουμε μπει στο 2ο loop του fork() και δημιουργείται η C διεργασία πρώτα.

Όσον αφορά τον τερματισμό, είναι φανερό πως πρώτα τερματίζουν όλα τα φύλλα, αφού οι γονείς κάνουν wait μέχρι να γίνει αυτό, ενώ για τα φύλλα η αναδρομική συνάρτηση καλεί την sleep και ύστερα τερματίζει. Έτσι τερματίζουν πρώτα τα C και D. Το C τερματίζει πριν από το D αφού δημιουργήθηκε πρώτο. Και αυτό γιατί εφόσον το sleep στα φύλλα γίνεται για το ίδιο χρονικό διάστημα, το φύλλο που δημιουργείται πρώτο θα ξυπνήσει και άρα και θα τερματίσει πρώτο. Στη συνέχει θα τερματίσει η B (εφόσον έχει ήδη τερματίσει το παιδί της) και τέλος η διεργασία / κορυφή του δέντρου A, της οποίας τον τερματισμό περιμένει η init στη main.

Άσκηση 3

Η παρούσα άσκηση αποτελεί μια επέκταση της παραπάνω, με την έννοια ότι οι διεργασίες ελέγχονται με την χρήση σημάτων. Ακόμη, τα μηνύματα ενεργοποίησης οφείλουν να εκτυπώνονται κατά βάθος.

Ο πηγαίος κώδικας της άσκησης έχει ως εξής:

```
#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) //recursive
{
	change_pname(root->name);
	if(root->children==NULL){

		printf("%s: Stopping...\n",root->name);
		raise(SIGSTOP);

		printf("%s: Continuing...\n",root->name);

		printf("%s: Exiting...\n",root->name);
		exit(1);
		return;
	}
	else{
		pid_t children_pids[root->nr_children];
		int i, pid, status;
		for(i=0;i<root->nr_children;i++){

			printf("%s: Starting...\n",(root->children+i)->name);
			pid=fork();

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

```
                }
                if(pid==0)
                        fork_procs(root->children+i);
                else{
                        children_pids[i]=pid;
                }
        }

        wait_for_ready_children(root->nr_children);

        printf("%s: Stopping...\n",root->name);
        raise(SIGSTOP);

        printf("%s: Continuing...\n",root->name);

        //waking children in DFS way
        for(i=0; i<root->nr_children; i++){
                int status;
                kill(children_pids[i],SIGCONT);
                wait(&status);
        }

        printf("%s: Exiting...\n",root->name);
        exit(1);
    }
}

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 */
    printf("%s: Starting...\n",root->name);
    pid = fork();
```

```
	if (pid < 0) {
		perror("main: fork");
		exit(1);
	}
	if (pid == 0) {
		/* Child */
		fork_procs(root); //create,sleep,wake
		exit(1);
	}

	wait_for_ready_children(1);

	/* Print the process tree root at pid */
	show_pstree(pid);

	/* for ask2-signals */
	if(kill(pid, SIGCONT)<0){ //waking A
		perror("kill");
		exit(1);
	}

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

	return 0;
}
```

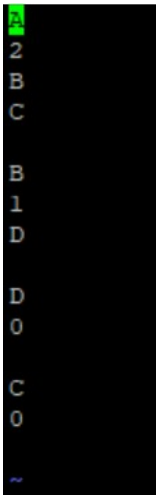
Ταυτόχρονα, ο κώδικας του Makefile είναι ο εξής:

```
all: ask2-signals

ask2-signals: ask2-signals.o tree.o proc-common.o
	gcc -o ask2-signals ask2-signals.o tree.o proc-common.o

ask2-signals.o: ask2-signals.c
	gcc -c ask2-signals.c

tree.o: tree.c
	gcc -c tree.c

proc-common.o: proc-common.c
	gcc -c proc-common.c
```

clean:

rm -f tree.o ask2-signals.o proc-common.o ask2-signals

Τρέχοντας το πρόγραμμα με input ίδιο με αυτό στην παραπάνω άσκηση, δηλαδή αυτό που φαίνεται εδώ:



Έχουμε το εξής output:

```
oslabdl4@os-node1:~/lab2/ex1_3$ ./ask2-signals test
A: Starting...
B: Starting...
C: Starting...
D: Starting...
A: Stopping...
My PID = 27098: Child PID = 27099 has been stopped by a signal, signo = 19
C: Stopping...
B: Stopping...
D: Stopping...


A(27099)─┬─B(27100)──D(27102)
         └─C(27101)


A: Continuing...
B: Continuing...
D: Continuing...
D: Exiting...
B: Exiting...
C: Continuing...
C: Exiting...
A: Exiting...
My PID = 27098: Child PID = 27099 terminated normally, exit status = 1
oslabdl4@os-node1:~/lab2/ex1_3$
```

Ερωτήσεις:

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

Στις προηγούμενες ασκήσεις για να συγχρονιστούν οι διεργασίες χρησιμοποιούσαμε την sleep() στα φύλλα, ενώ εδώ χρησιμοποιούμε σήματα. Αυτός ο τρόπος είναι σαφώς πιο ελεγχόμενος και αξιόπιστος.
Για μικρά δέντρα, η χρήση της sleep(), αν και δεν είναι τόσο αξιόπιστη, εξασφαλίζει τον σχηματισμό του συνολικού δέντρου πριν αρχίσει ο τερματισμός κάποιου φύλλου. Ωστόσο, σε πολύ μεγάλα δέντρα, ο συγκεκριμένος χρόνος για τον οποίο η sleep() σταματά την εκτέλεση της καλούμενης διεργασίας μπορεί να μην επαρκεί και έτσι να απαιτούνται δοκιμές για να προσδιοριστεί το κατάλληλο χρονικό διάστημα.
Αντιθέτως, η χρήση σημάτων μας παρέχει απόλυτα ελεγχόμενη παύση και στη συνέχεια έναρξη της εκτέλεσης μίας διεργασίας ώστε να επιτυγχάνεται το επιθυμητό αποτέλεσμα σε κάθε περίπτωση.

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

Στην άσκηση αυτή, θέλαμε αφότου δημιουργηθεί το επιθυμητό δέντρο διεργασιών, κάθε διεργασία να εξασφαλίσει την παύση της εκτέλεσης όλων των παιδιών-διεργασιών της και στη συνέχεια τη επανενεργοποίηση τους με DFS τρόπο μόλις της σταλεί το σήμα SIGCONT.

Η wait_for_ready_children() χρησιμοποιείται για να βεβαιωθούμε πως όλα τα παιδιά-διεργασίες έχουν αναστείλει την εκτέλεσή τους (με τη raise(SIGSTOP)) πριν την αναστείλει και η γονική τους. Έτσι όταν σταλεί το σήμα SIGCONT στη γονική, μπορεί συγχρονισμένα να πραγματοποιηθεί το DFS ξύπνημα όλων των παιδιών.

Εάν η συνάρτηση δε χρησιμοποιούνταν, ενδεχομένως κάποια παιδιά-διεργασίες να συνέχιζαν την εκτέλεσή τους τη στιγμή που ο γονέας τους ή τα παιδιά τους ακόμα ήταν σε αναστολή, δε θα υπήρχε δηλαδή συγχρονισμός.

Άσκηση 4

Στο 4ο μέρος της εργαστηριακής άσκησης δημιουργείται πρόγραμμα για τον παράλληλο υπολογισμό αριθμητικών εκφράσεων με χρήση δέντρων και διοχετεύσεων (pipes).

Ο πηγαίος κώδικας της άσκησης είναι:

```
#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"
#include <string.h>

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3

static void fork_rec(struct tree_node *root, int pfd_father[2]){

        change_pname(root->name);

        if(root->children==NULL){
                int value=atoi(root->name);
                write(pfd_father[1], &value, sizeof(value));
                close(pfd_father[1]); //closes the used to write
                return;
        }

        else{
                int i, pid, status,result[2];

                for(i=0; i<2; i++){
                printf("%s-%s: Creating pipe ...\n",root->name,(root->children+i)->name);

                int pfd_child[2]; //for each child make a pipe
                int value;

                if(pipe(pfd_child)<0){
                        perror("pipe");
                        exit(1);
```

```
                }

                printf("%s: Creating...\n",(root->children+i)->name);
                pid=fork();

                if(pid<0){
                        perror("fork");
                        exit(1);
                }
                if(pid==0){ //child
                        fork_rec(root->children+i,pfd_child);
                        printf("%s: Exiting...\n",(root->children+i)->name);
                        exit(1);
                }
                //father
                read(pfd_child[0], &value, sizeof(value));
                close(pfd_child[0]); //close the use for reading
                result[i]=value;
                }
                //result[2] ready father waits for children to terminate

                for(i=0;i<2; i++){
                printf("%s: Waiting for %s to terminate if not already terminated...\n",root->name,
(root->children+i)->name);

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

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

                write(pfd_father[1], &final, sizeof(final));
                close(pfd_father[1]);//close the used for writing
                printf("%s: Exiting...\n",root->name);
                exit(1);

        }
}



int main(int argc, char *argv[]){
```

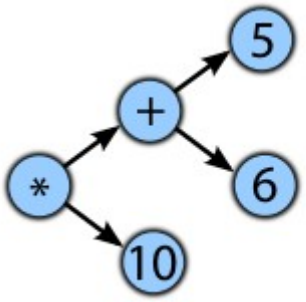
```
struct tree_node *root;
if(argc!=2){
	fprintf(stderr, "Usage: %s <input_tree_file>\n\n",argv[0]);
	exit(1);
}
root=get_tree_from_file(argv[1]); //read tree from file

/*Print the process tree from file*/
printf("Calculating this tree!\n");
print_tree(root);
pid_t pid;
int status;
int value;
int pfd[2];

printf("Init-%s: Creating pipe...\n", root->name);
if(pipe(pfd)<0){
	perror("pipe");
	exit(1);
}

printf("%s: Creating...\n",root->name);
pid = fork();
if (pid < 0) {
	perror("main: fork");
	exit(1);
}
if (pid == 0) {
	close(pfd[0]); //child is not reading
	fork_rec(root,pfd);
	exit(1);
}

/* Wait for the root of the process tree to terminate */
close(pfd[1]); //close the non used write

printf("Init: Waiting for %s to terminate if not already terminated...\n",root->name);
pid = wait(&status);
explain_wait_status(pid, status);
int final;
read(pfd[0], &final, sizeof(final));
close(pfd[0]);
printf("The result of the calculation is %d!\n",final);
return 0;
```

}

Όπως και στις προηγούμενες ασκήσεις, η δημιουργία του εκτελέσιμου γίνεται με Makefile ως εξής:

```
all: ask2-fork
ask2-fork: ask2-fork.o tree.o proc-common.o
	gcc -o ask2-fork ask2-fork.o tree.o proc-common.o
ask2-fork.o: ask2-fork.c
	gcc -c ask2-fork.c
tree.o: tree.c
	gcc -c tree.c
proc-common.o: proc-common.c
	gcc -c proc-common.c
clean:
	rm -f tree.o ask2-fork.o proc-common.o ask2-fork
```

Ένα παράδειγμα εκτέλεσης του κώδικα είναι για τον υπολογισμό της αριθμητικής έκφρασης που περιγράφεται από το δέντρο της εκφώνησης:



Σχήμα 3: Δέντρο έκφρασης $10 \times (5 + 6)$

Για τον σκοπό αυτό, δημιουργήθηκε το αρχείο test περιγραφής του δέντρου, το οποίο φαίνεται παρακάτω. Θεωρείται ότι μη τερματικοί κόμβοι είναι οι τελεστές * ή + με 2 παιδιά ενώ οι τερματικοί κόμβοι/φύλλα είναι ακέραιοι αριθμοί.

Τρέχοντας τον κώδικα με την εντολή ./ask2-fork test για αυτή την είσοδο έχουμε:

```
@os-node2: ~/lab2/ex1_4
oslabd14@os-node2:~/lab2/ex1_4$ ./ask2-fork test
Calculating this tree!
*
        +
                5
                6
        10
Init-*: Creating pipe...
*: Creating...
Init: Waiting for * to terminate if not already terminated...
*-+: Creating pipe ...
+: Creating...
+-5: Creating pipe ...
5: Creating...
+-6: Creating pipe ...
6: Creating...
5: Exiting...
+: Waiting for 5 to terminate if not already terminated...
6: Exiting...
My PID = 7500: Child PID = 7501 terminated normally, exit status = 1
+: Waiting for 6 to terminate if not already terminated...
My PID = 7500: Child PID = 7502 terminated normally, exit status = 1
+: Exiting...
*-10: Creating pipe ...
10: Creating...
*: Waiting for + to terminate if not already terminated...
10: Exiting...
My PID = 7499: Child PID = 7500 terminated normally, exit status = 1
*: Waiting for 10 to terminate if not already terminated...
My PID = 7499: Child PID = 7503 terminated normally, exit status = 1
*: Exiting...
My PID = 7498: Child PID = 7499 terminated normally, exit status = 1
The result of the calculation is 110!
oslabd14@os-node2:~/lab2/ex1_4$
```

Όπου πράγματι προκύπτει το αναμενόμενο αποτέλεσμα για το δέντρο της αριθμητικής έκφρασης (110).

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;

Στην άσκηση, για κάθε διεργασία απαιτείται μία σωλήνωση με τη γονική της και μία με κάθε παιδί της. Εφόσον η διεργασία μπορεί είτε να έχει 2 παιδιά (τελεστής/μη τερματικός κόμβος) είτε κανένα (ακέραιος αριθμός/φύλλο), απαιτούνται 3 σωληνώσεις για κάθε μη τερματικό κόμβο και 1 για κάθε τερματικό.

Οι σωληνώσεις (pipes) στα Linux είναι μία μέθοδος διαδιεργασιακής επικοινωνίας μεταξύ 2 διεργασιών. Η μία διεργασία γράφει στο ένα άκρο της σωλήνωσης (fd(1))και η άλλη διαβάζει από αυτό (fd(0)) και η επικοινωνία γίνεται μόνο προς μία κατεύθυνση.

Άρα, εφόσον με σωληνώσεις επιτυγχάνεται μονόδρομη επικοινωνία μεταξύ 2 διεργασιών, δηλαδή κάθε διεργασία μίας σωλήνωσης μπορεί είτε να διαβάζει είτε να γράφει, δεν είναι δυνατόν κάθε διεργασία να έχει μία σωλήνωση για όλα τα παιδιά της ή για κάθε τελεστή να χρησιμοποιείται μόνο μία σωλήνωση. Αν χρησιμοποιούσαμε σε αυτές τις περιπτώσεις μία σωλήνωση, δηλαδή ένα αρχείο, σε κάθε νέα εγγραφή στο άκρο εγγραφής, θα διαγράφονταν οι προηγούμενες εγγραφές και θα υπήρχε απώλεια δεδομένων προς ανάγνωση και άρα λανθασμένο αποτέλεσμα στον υπολογισμό της αριθμητικής έκφρασης.

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;

Το πλεονέκτημα είναι πώς σε ένα σύστημα πολλαπλών επεξεργαστών, οι επιμέρους αριθμητικές πράξεις που απαιτούνται για τον υπολογισμό του τελικού αποτελέσματος μπορούν να πραγματοποιηθούν από επιμέρους διεργασίες που εκτελούνται παράλληλα. Έτσι, οι επιμέρους αριθμητικές εκφράσεις/κλαδιά κάθε μη τερματικού κόμβου υπολογίζονται ταυτόχρονα και το τελικό αποτέλεσμα προκύπτει γρηγορότερα έναντι της αποτίμησης από μία μόνο διεργασία.