ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ - ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Λειτουργικά Συστήματα

Ομάδα 118:

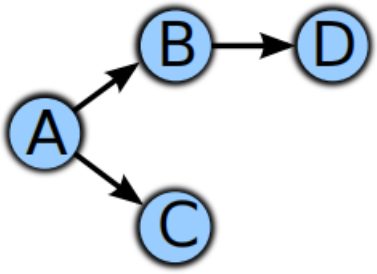
Ρέππας Ευστράτιος (03120002) – Τζουρμανά Ελευθερία (03119927)

Άσκηση 2:

Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Για την άσκηση 1.1 ζητείται η δημιουργία του παρακάτω δέντρο διεργασιών. Το δέντρο διεργασιών που δημιουργείται καθώς και ο αντίστοιχος κώδικας φαίνονται παρακάτω.



- Ο πηγαίος κώδικας της άσκησης φαίνεται παρακάτω:

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

```c
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "proc-common.h"

#define SLEEP_PROC_SEC  4
#define SLEEP_TREE_SEC  3

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */
void fork_procs(void) {
        /*
     * initial process is A.
     */
        change_pname("A"); //changes the process name to A
        printf("A: Starting...\n");
        /* ... */

        pid_t pid;
        int status, i;

        pid = fork();//fork the next  procedure; we create a second process with exactly the same code and pcb with the
        // single difference of pid_new=0. The new procedure runs after the fork command, just like the parent.

        if (pid < 0) { //error message
                perror("B: fork");
                exit(1);
        }
        if (pid == 0) { //this is accessable by the child process (pid=0) as the parent ignores this if statement
                //Everything the child does is in this code
                /* Child */
                change_pname("B"); //change the process name to B
                printf("B: Starting...\n");

                pid = fork(); //we now fork the next process (D), following the same principles

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

```
                if (pid == 0) {
                        /* D Child */
                        change_pname("D");
                        printf("D: Sleeping...\n");
                        sleep(SLEEP_PROC_SEC); //D process is a leaf
process and so it sleeps
                        printf("D: Exiting...\n"); //D exits with code 13
                        exit(13);
                }

                //now back to the B (pid>0) process, as it ignores the if
statement
                printf("B: Waiting...\n");
                pid = wait(&status); //B waits for the first child to
finish
                printf("B: Exiting...\n"); //then B exits with code 19
                exit(19);

        }//from now on this will be viewed by the A process (as the others
terminate before that)
        pid = fork(); //Now we run C, with A being the parent node

        if (pid < 0) {
                perror("C: fork");
                exit(1);
        }
        if (pid == 0) {
                /* Child */
                change_pname("C");
                printf("C: Starting...\n");
                printf("C: Sleeping...\n");
                sleep(SLEEP_PROC_SEC); //C process is a leaf process and so
it sleeps
                printf("C: Exiting...\n");//C exits with code 17
                exit(17);
        }

        printf("A: Waiting...\n");
        for (i = 0; i < 2; i++)
                pid = wait(&status);//A waits for both of its children to
finish
        printf("A: Exiting...\n");//A exits with code 16
        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;
}
```

- Το τελικό Makefile φαίνεται παρακάτω και περιλαμβάνει και τις υπόλοιπες ασκήσεις:

```
.PHONY: all clean

all: fork-example tree-example ask2-fork ask2-signals ask22 ask24 help
askhelp

CC = gcc
CFLAGS = -g -Wall -O2
SHELL= /bin/bash

askhelp: askhelp.o tree.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

help: help.o tree.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

ask22: ask22.o tree.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

ask24: ask24.o tree.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

tree-example: tree-example.o tree.o
	$(CC) $(CFLAGS) $^ -o $@

fork-example: fork-example.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

ask2-fork: ask2-fork.o proc-common.o
	$(CC) $(CFLAGS) $^ -o $@

ask2-signals: ask2-signals.o proc-common.o tree.o
	$(CC) $(CFLAGS) $^ -o $@

%.s: %.c
	$(CC) $(CFLAGS) -S -fverbose-asm $<

%.o: %.c
	$(CC) $(CFLAGS) -c $<

%.i: %.c
	gcc -Wall -E $< | indent -kr > $@

clean:
	rm -f *.o tree-example fork-example pstree-this ask2-
{fork,tree,signals,pipes}
```

- Η έξοδος εκτέλεσης για την άσκηση 1.1 είναι η παρακάτω:

```
oslab118@orion:~/ask2$ ./ask2-fork
A: Starting...
A: Waiting...
C: Starting...
C: Sleeping...
B: Starting...
B: Waiting...
D: Sleeping...


A(21755)─┬─B(21756)──D(21758)
         └─C(21757)


C: Exiting...
My PID = 21755: Child PID = 21757 terminated normally, exit status = 17
D: Exiting...
My PID = 21756: Child PID = 21758 terminated normally, exit status = 13
B: Exiting...
My PID = 21755: Child PID = 21756 terminated normally, exit status = 19
A: Exiting...
My PID = 21754: Child PID = 21755 terminated normally, exit status = 16
```

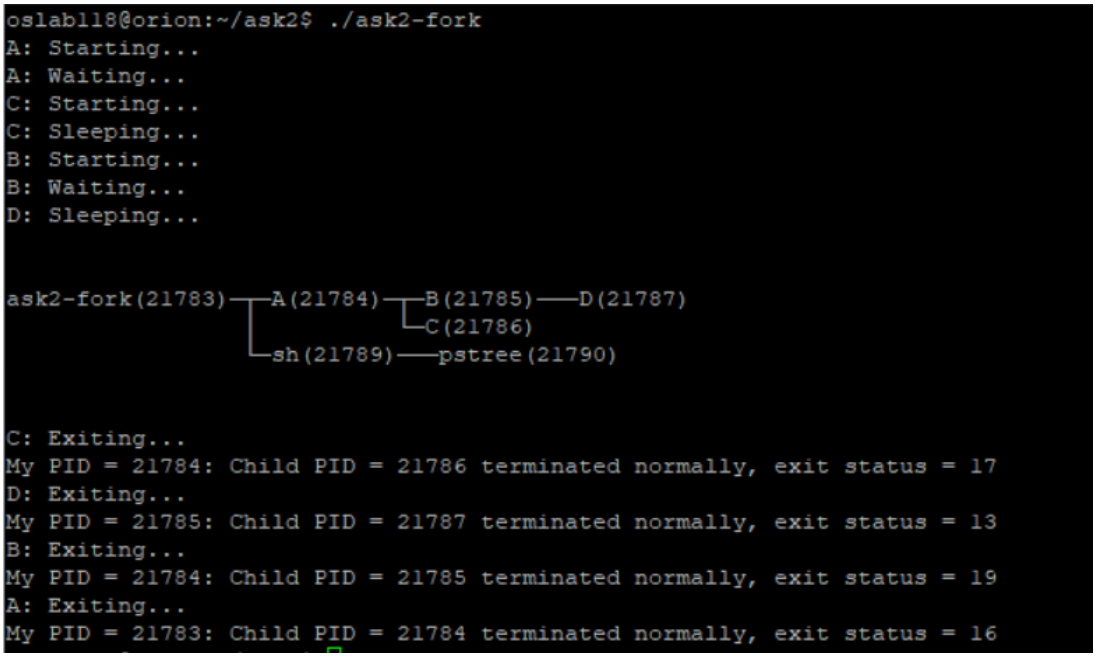
Ερωτήσεις:

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

Αν ο γονέας πεθάνει πρώτος από τα παιδιά του, η διεργασία παιδί γίνεται init (PID=1) και κάνει συνεχώς wait για να μαζέψει όλα τα zombie διεργασιών που για κάποιο λόγο δεν υπάρχει η γονική διεργασία στο σύστημα.

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

Αντικαθιστούμε στον κώδικα μας όπου *show_pstree(pid) το show_pstree(getpid()) και η έξοδος εκτέλεσης φαίνεται παρακάτω. Οι* πρόσθετες διεργασίες που εμφανίζονται στο δέντρο θα είναι η διεργασία φλοιού (shell process) και οποιεσδήποτε άλλες διεργασίες που εκτελούνται στο τερματικό ή στη γραμμή εντολών όπου εκτελέσαμε το πρόγραμμα. Αυτό συμβαίνει επειδή η getpid() επιστρέφει το process ID της τρέχουσας διεργασίας. Επομένως, η χρήση του show_pstree(getpid()) δείχνει ολόκληρο το δέντρο διεργασίας που περιλαμβάνει την κύρια διεργασία και οποιεσδήποτε άλλες διεργασίες που εκτελούνται στο ίδιο τερματικό ή γραμμή εντολών. Το δέντρο αυτό περιλαμβάνονται οι διεργασίες sh (standard command language interpreter), με παιδί το pstree, που καλούνται απο την show pstree.



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

Σε ένα σύστημα πολλαπλών χρηστών χωρίς όριο στο πλήθος διεργασιών, δεδομένου των πεπερασμένων πόρων, ένας χρήστης θα μπορούσε να υπερφορτώσει το σύστημα με ”άπειρες” διεργασίες με χρήση της συνάρτησης fork() σε ένα ατέρμονο βρόγχο. Κάτι τέτοιο θα μπορούσε να οδηγήσει στην εξάντληση της CPU. Για να αποφευχθεί κάτι τέτοιο θέτεται ένας περιορισμός του αριθμού των διεργασιών ανά χρήστη.

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Υλοποιούμε πρόγραμμα που βασίζεται σε αναδρομική συνάρτηση και δημιουργεί αυθαίρετα δέντρα διεργασιών βάσει αρχείου εισόδου.

- Ο πηγαίος κώδικας της άσκησης φαίνεται παρακάτω:

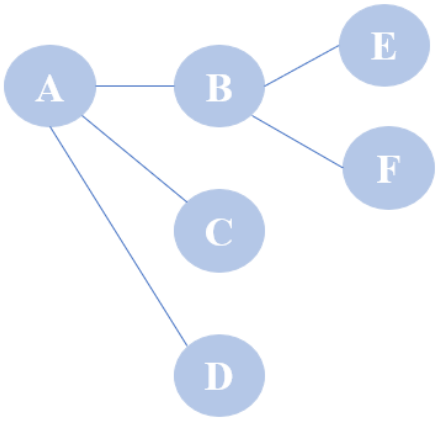
```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

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

#define SLEEP_PROC_SEC  10
#define SLEEP_TREE_SEC  3



void ask22(struct tree_node *node){
    pid_t pid;
    int status, i=0;

    change_pname(node->name); //set the process name to the corresponding
value for the current process
    printf("%s: Starting...\n", node->name);
    //printf("A: Sleeping...\n");
    /* ... */
    for(i=0; i<node->nr_children; i++) { //fork each one of the child
processes a node has.
        // If it has no children, it shouldn't enter the loop
        pid = fork();

        if (pid < 0) { //error message
            perror("fork");
            exit(1);
        }
        if (pid == 0) { //this recursively calls this function for each
child. The parent ignores it and continues the loop
            ask22(node->children + i); //repeat for the next child
(parent+1)
        }
    }

    if(i==0){ //if it has no children (aka leaf)
        printf("%s: Sleeping...\n",node->name);
        sleep(SLEEP_PROC_SEC); //I want you to sleep
```

```
    }
    else { //If it has children
        for(i=0; i < node->nr_children; i++) { //for each one
            pid=wait(&status); //wait for it to finish
            explain_wait_status(pid, status);
        }
    }
    printf("%s: Exiting...\n",node->name); //exit successfully
    exit(0);
}

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

    if (argc != 2) { //error message for correct usage
        fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
        exit(1);
    }
    pid = fork();
    if (pid < 0) {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) {
        /* Child */
        root=get_tree_from_file(argv[1]); //reads the tree file
        ask22(root); //calls the recursive function
        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;
}
```

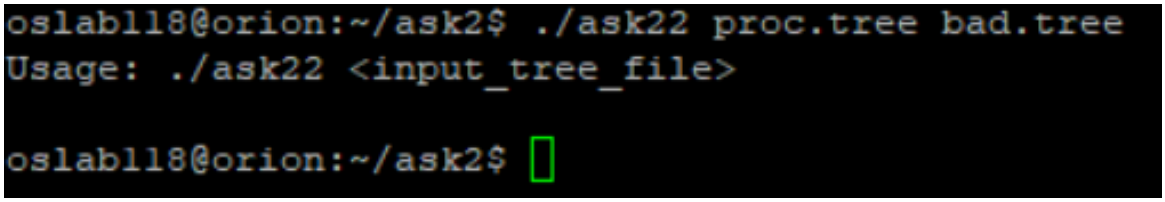
- Η έξοδος εκτέλεσης για την άσκηση 1.2 είναι η παρακάτω:

Παράδειγμα εκτέλεσης για το παρακάτω δέντρο το οποίο βρίσκεται στο file “proc.tree”.




```
oslab118@orion:~/ask2$ ./ask22 proc.tree
A: Starting...
C: Starting...
D: Starting...
D: Sleeping...
C: Sleeping...
B: Starting...
E: Starting...
E: Sleeping...
F: Starting...
F: Sleeping...


A(25578)─┬─B(25579)─┬─E(25582)
         │          └─F(25583)
         ├─C(25580)
         └─D(25581)


D: Exiting...
C: Exiting...
My PID = 25578: Child PID = 25580 terminated normally, exit status = 0
My PID = 25578: Child PID = 25581 terminated normally, exit status = 0
E: Exiting...
My PID = 25579: Child PID = 25582 terminated normally, exit status = 0
F: Exiting...
My PID = 25579: Child PID = 25583 terminated normally, exit status = 0
B: Exiting...
My PID = 25578: Child PID = 25579 terminated normally, exit status = 0
A: Exiting...
My PID = 25577: Child PID = 25578 terminated normally, exit status = 0
```

Παράδειγμα εκτέλεσης για λάθος αριθμό ορισμάτων:



Ερωτήσεις:

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

Έχουμε παρατηρήσει ότι η σειρά με την οποία εμφανίζονται τα μηνύματα έναρξης και τερματισμού των διεργασιών είναι τυχαία και μάλιστα διαφέρει από εκτέλεση σε εκτέλεση. Η σειρά εμφάνισης εξαρτάται από τον χρονοδρομολογητή (scheduler) του λειτουργικού συστήματος. Συγκεκριμένα, ο χρονοδρομολογητής αναλαμβάνει την επιλογή της διεργασίας που θα χρησιμοποιήσει τον επεξεργαστή καθώς και την αλλαγή της διεργασίας που εκτελείται στον επεξεργαστή. Επομένως, ακόμη και αν το ίδιο πρόγραμμα εκτελείται πολλές φορές, η σειρά με την οποία ξεκινούν και τερματίζονται οι διεργασίες μπορεί να διαφέρει.

1.3 Αποστολή και χειρισμός σημάτων

Επεκτείνουμε το πρόγραμμα της § 1.2 έτσι ώστε οι διεργασίες να ελέγχονται με χρήση σημάτων, για να εκτυπώνουν τα μηνύματά τους κατά βάθος (Depth-First).

- Ο πηγαίος κώδικας της άσκησης φαίνεται παρακάτω:

```
#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("%s, starting... (PID = %ld)\n", root->name, (long)
getpid());
        change_pname(root->name); //set the process name to the
corresponding value for the current process

        /* ... */
        pid_t help;
        int status, i = 0;
        pid_t pid[root->nr_children];

        for (i = 0; i < root->nr_children; i++) { //for each of the
children (if it has none it should skip this)

                pid[i] = fork(); //fork each child

                if (pid[i] < 0) { //error message
                        perror("fork");
                        exit(1);
                }
                if (pid[i] == 0) { //this code is for the child
                        fork_procs(root->children + i); //calls recursively
the same function for the child (parent+1)
                }
        }

        if (i != 0) {
```

```
                wait_for_ready_children(root->nr_children); //wait for all
the children to suspend
        }

                /*
   * Suspend Self
   */
                printf("%s suspending... (PID = %ld)\n", root->name, (long)
getpid());
                raise(SIGSTOP); //suspend yourself by broadcasting a
sigstop flag. This stops the running of the program until it receives the
sigcont flag
                //now if we continue from now on we have received a sigcont
flag by the parent
                printf("%s is awake! (PID = %ld)\n", root->name, (long)
getpid());


        for(i=0; i<root->nr_children; i++){
                kill(pid[i], SIGCONT); //for each child as specified by its
pid send the sigcont to continue its execution
                help=wait(&status); //wait for each child to finish (one at
a time)
                explain_wait_status(help, status);
        }

        /*
   * Exit
   */
        printf("%s, exiting... (PID = %ld)\n", root->name, (long)
getpid()); //exit succesfully
        exit(0);
}

//Basically we create all the processes and before they execute their
function we stop them.
// After all processes are created, we start waking them up one by one


/*
 * 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){ //error message for worng usage
                fprintf(stderr, "Usage: %s <tree_file>\n", argv[0]);
                exit(1);
        }

        /* Read tree into memory */
         root = get_tree_from_file(argv[1]); //read the tree from the file

        /* Fork root of process tree */
        pid = fork();

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

        /*
     * Father
     */
        /* for ask2-signals */
        wait_for_ready_children(1); //we wait until the first process of
the tree is suspended

        /* for ask2-{fork, tree} */
        /* sleep(SLEEP_TREE_SEC); */

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

        /* for ask2-signals */
        kill(pid, SIGCONT); //we then wake the first process and the rest
is history

        /* Wait for the root of the process tree to terminate */
```

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

        return 0;
}
```

- Η έξοδος εκτέλεσης για την άσκηση 1.3 είναι η παρακάτω:

Παράδειγμα εκτέλεσης για το δέντρο το οποίο βρίσκεται στο file “proc.tree”

```
oslab118@orion:~/ask2$ ./ask2-signals proc.tree
A, starting... (PID = 25798)
D, starting... (PID = 25801)
D suspending... (PID = 25801)
My PID = 25798: Child PID = 25801 has been stopped by a signal, signo = 19
B, starting... (PID = 25799)
C, starting... (PID = 25800)
C suspending... (PID = 25800)
My PID = 25798: Child PID = 25800 has been stopped by a signal, signo = 19
F, starting... (PID = 25803)
F suspending... (PID = 25803)
My PID = 25799: Child PID = 25803 has been stopped by a signal, signo = 19
E, starting... (PID = 25802)
E suspending... (PID = 25802)
My PID = 25799: Child PID = 25802 has been stopped by a signal, signo = 19
B suspending... (PID = 25799)
My PID = 25798: Child PID = 25799 has been stopped by a signal, signo = 19
A suspending... (PID = 25798)
My PID = 25797: Child PID = 25798 has been stopped by a signal, signo = 19


A(25798)─┬─B(25799)─┬─E(25802)
         │          └─F(25803)
         ├─C(25800)
         └─D(25801)


A is awake! (PID = 25798)
B is awake! (PID = 25799)
E is awake! (PID = 25802)
E, exiting... (PID = 25802)
My PID = 25799: Child PID = 25802 terminated normally, exit status = 0
F is awake! (PID = 25803)
F, exiting... (PID = 25803)
My PID = 25799: Child PID = 25803 terminated normally, exit status = 0
B, exiting... (PID = 25799)
My PID = 25798: Child PID = 25799 terminated normally, exit status = 0
C is awake! (PID = 25800)
C, exiting... (PID = 25800)
My PID = 25798: Child PID = 25800 terminated normally, exit status = 0
D is awake! (PID = 25801)
D, exiting... (PID = 25801)
My PID = 25798: Child PID = 25801 terminated normally, exit status = 0
A, exiting... (PID = 25798)
My PID = 25797: Child PID = 25798 terminated normally, exit status = 0
oslab118@orion:~/ask2$
```

Ερωτήσεις:

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

Αρχικά, η χρήση σημάτων πλεονεκτεί έναντι της sleep() για τον συγχρονισμό των διεργασιών. Η sleep() απαιτεί ένα σταθερό χρονικό διάστημα να περάσει πριν συνεχιστεί η διαδικασία, γεγονός που καταναλώνει σημαντικούς πόρους της CPU. Αντίθετα, τα σήματα δεν καταναλώνουν καθόλου χρόνο της CPU ενώ περιμένουν να φτάσει το σήμα. Επομένως, όσο μεγαλύτερο είναι το μέγεθος του δέντρου τόσο μικρότερη είναι η αποκρισιμότητα του προγράμματος μας. Επιπλέον, τα σήματα μπορούν να χρησιμοποιηθούν επιπλέον για την επικοινωνία μεταξύ διεργασιών. Τέλος, με την χρήση σημάτων μπορούμε να ελέγξουμε κατά κάποιο τρόπο τις διεργασίες.

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

```
/*
 * Make sure all the children have raised SIGSTOP,
 * by using waitpid() with the WUNTRACED flag.
 *
 * This will NOT work if children use pause() to wait for SIGCONT.
 */
void
wait_for_ready_children(int cnt)
{
        int i;
        pid_t p;
        int status;

        for (i = 0; i < cnt; i++) {
                /* Wait for any child, also get status for stopped children
*/
                p = waitpid(-1, &status, WUNTRACED);
                explain_wait_status(p, status);
                if (!WIFSTOPPED(status)) {
                        fprintf(stderr, "Parent: Child with PID %ld has
died unexpectedly!\n",
                                (long)p);
                        exit(1);
                }
        }
}
```

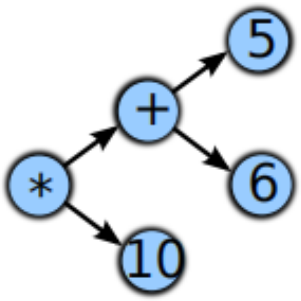
Ο ρόλος της συνάρτηση wait_for_ready_children() είναι να περιμένει όλα τα παιδιά να σταματήσουν. Χρησιμοποιεί την κλήση συστήματος waitpid() με flag

WUNTRACED, η οποία επιστρέφει το PID του παιδιού που άλλαξε κατάσταση. Εάν η διεργασία του παιδιού διακόπηκε από το σήμα SIGSTOP, η μακροεντολή WIFSTOPPED() επιστρέφει true και ο βρόχος συνεχίζει να περιμένει για την επόμενη διεργασία. Αν το παιδί τερματιστεί απροσδόκητα, εκτυπώνεται μήνυμα σφάλματος.

Η συνάρτηση διασφαλίζει ότι όλες οι διεργασίες των παιδιών έχουν σταματήσει πριν επιτραπεί από την γονική διεργασία να συνεχίσουν. Επιπλέον, η χρήση της διασφαλίζει την κατά βάθος (DFS) σειρά εμφάνισης των μηνυμάτων. Κάτι τέτοιο είναι χρήσιμο όταν η γονική διαδικασία χρειάζεται να συγχρονίσει πολλές διεργασίες, να αποφευχθεί η δημιουργία «orphan», εξασφαλίζοντας αξιόπιστη επικοινωνία μεταξύ διεργασιών χωρίς πιθανά race conditions.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Επεκτείνουμε τον κώδικα του ερωτήματος § 1.2 ώστε να υπολογίζει δέντρα που αναπαριστούν αριθμητικές εκφράσεις. Για παράδειγμα, το δέντρο που αναπαριστά την έκφραση 10 × (5 + 6).



- Ο πηγαίος κώδικας της άσκησης φαίνεται παρακάτω:

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

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

//we do not need the sleep times unless in order to print the tree
//there is no need for them if we just want the result


void math(struct tree_node *root, int fd_parent){
    pid_t pid;
    int i=0, fd_child[2][2], buffer[2];

    change_pname(root->name); //set the process name to the corresponding
value for the current process
    printf("%s: Starting...\n", root->name);


    for(i=0; i<root->nr_children; i++) {  //for each of the children (if it
has none it should skip this)

        if (pipe(fd_child[i]) < 0) { //create a pipe (one for each child)
while printing an error message if it fails.
            // Since each node has exactly two children, we need two pipes
per parent (hence the 2d array)
            perror("pipe failed");
            exit(1);
        }
```

```
        pid = fork(); //fork each child

        if (pid < 0) { //error message
            perror("fork");
            exit(1);
        }
        if (pid == 0) { //this code is for the child
            close(fd_child[i][0]); //since we won't be using the child to
read from the parent, we close the corresponding fd
            math(root->children + i, fd_child[i][1]); //call the function
recursively for the next chilod process,
            //passing the fd to write as an argument
        }
        close(fd_child[i][1]); //if we reach this line, this is by the
parent node and therefore we close the write
        // fd towards the child since we won't be using it
    }

    if(i==0){ //if the process has no children (aka leaf)
        printf("%s: Writing...\n",root->name);
        int num = atoi(root->name); // convert the char type name of the
node to number (since you are a child you have a number as a name)
        write(fd_parent,&num,sizeof(num)); //write the number to the fd as
passed by through the function by the parent
        close(fd_parent); //we finished writing so we close the fd
        //we do not need to sleep as this is just in order to print the
tree as explained
    }
    else {
        for(i=0; i < root->nr_children; i++) { //for each of the two
children
            //there is no need to use wait() since the read() system call
waits for something to be written in the fd to execute
            read(fd_child[i][0],&buffer[i],sizeof(buffer[i])); //read the
number as written by the child to the corresponding fd
            //and write it to the buffer. The buffer is a 1x2 array where
each column should contain the value of a child
            printf("%s: Reading %d... \n",root->name, buffer[i]);
            close(fd_child[i][0]); //we finished reading so we close the fd
        }
        int rslt;
        if (*root->name=='+'){ //if your name is +, add the two values you
read
            rslt = buffer[0]+buffer[1];}

        else if (*root->name=='*'){ //if your name is *, multiply the two
values you read
```

```
            rslt = buffer[0]*buffer[1];}
        else {
            perror("invalid node name");
            exit(1);
        }

        write(fd_parent,&rslt,sizeof(rslt)); //write them for your parent
node to read
        close(fd_parent); //we finished writing so we close the fd
    }
    //exit successfully
    exit(0);


}


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

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

    if (pipe(fd_root) < 0) { //create  a pipe for the root node of the
tree. It should only be used to pass the result
        perror("pipe failed");
        exit(1);
    }

    pid = fork(); //fork the root node

    if (pid < 0) { //error message
        perror("main: fork");
        exit(1);
    }
    if (pid == 0) { //for the root node
        /* Child */
        root=get_tree_from_file(argv[1]); //read the tree from the file
        math(root, fd_root[1]); //call the recursive function
    }

    /*
     * Father
     */
```

```
    /* Wait for the root of the process tree to terminate */
    read(fd_root[0],&result,sizeof(result)); //read the result as provided
by the root node
    close(fd_root[0]); //close all the remaining file descriptors
    close(fd_root[1]);
    printf("The final result is %d \n", result); //print the result

    return 0;
}
```

- Η έξοδος εκτέλεσης για αρχείο εισόδου το δέντρο που υλοποιεί την εξής πράξη:

  10+4*(5+7):

```
oslab118@orion:~/ask2$ ./ask24 expr.tree
+: Starting...
*: Starting...
10: Starting...
10: Writing...
+: Reading 10...
+: Starting...
4: Starting...
4: Writing...
7: Starting...
7: Writing...
5: Starting...
5: Writing...
+: Reading 5...
+: Reading 7...
*: Reading 12...
*: Reading 4...
+: Reading 48...
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;

Στην άσκηση αυτή δημιουργούμε μια σωλήνωση (μια για κάθε παιδί). Αν ο κάθε κόμβος έχει ακριβώς 2 παιδιά τότε χρειαζόμαστε 2 σωληνώσεις ανά γονέα. Δεδομένου ότι οι συγκεκριμένες αριθμητικές εκφράσεις περιέχουν τους αντιμεταθετικούς τελεστές της πρόσθεσης και του πολλαπλασιασμού, η κάθε γονική διεργασία θα μπορούσε να χρησιμοποιεί μόνο μια σωλήνωση για όλες τις διεργασίες

παιδιά της. Το συγκεκριμένου ισχύει καθώς δεν μας επηρεάζει η σειρά με την οποία επιστρέφουν οι διεργασίες παιδιά τις τιμές τους σε αυτήν, λόγω της αντιμεταθετικότητας των τελεστών. Ωστόσο, δεν ισχύει για κάθε αριθμητικό τελεστή και ειδικά για τις πράξεις της αφαίρεσης και της διαίρεσης προκειμένου να γνωρίζουμε ποιος είναι ο κάθε τελεστέος που επιστρέφουν τα παιδιά στον γονέα.

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;

Δεδομένου ότι σε ένα σύστημα πολλαπλών επεξεργαστών μπορούν να εκτελούνται παραπάνω από μία διεργασίες παράλληλα, πράξεις που βρίσκονται στο ίδιο επίπεδο αλλά σε διαφορετικά «μονοπάτια» γίνονται παράλληλα. Αυτό μειώνει τον χρόνο που χρειάζεται προκειμένου να υπολογιστεί οι αριθμητική έκφραση.