Λειτουργικά Συστήματα
2η Εργαστηριακή Αναφορά

Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2022-2023

Ομάδα: oslab35

Μέλη

Καμπουγέρης Χαράλαμπος | A.M: el20098

Κουστένης Χρίστος | A.M: el20227

Μέρος 1ο : Διαχείριση διεργασιών

1.1. ΔΗΜΙΟΥΡΓΙΑ ΔΕΔΟΜΕΝΟΥ ΔΕΝΤΡΟΥ ΔΙΕΡΓΑΣΙΩΝ

Ο πηγαίος κώδικας της άσκησης είναι ο εξής:

```c
#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

/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */
/**/

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

    /* ... */
```

```
    int statusA;
    change_pname("A");
    printf("Creating... B\n");

    pid_t pA = fork();
    if (pA < 0)
    {
        perror("A: fork");
        exit(16);
    }
    if (pA == 0)
    {
        int statusB;
        change_pname("B");
        printf("Creating... D\n");
        pid_t pB = fork();
        if (pB < 0)
        {
            perror("B: fork");
            exit(19);
        }
        if (pB == 0)
        {
            change_pname("D");
            printf("D: Sleeping...\n");
            sleep(SLEEP_PROC_SEC);
            printf("D: Exiting...\n");
            exit(13);
        }
        printf("B: Waiting...\n");
        pB = wait(&statusB);
        explain_wait_status(pB, statusB);
        printf("B: Exiting...\n");
```

```
        exit(19);
    }
    printf("Creating... C\n");
    pid_t pA_2 = fork();
    if (pA_2 == -1)
    {
        perror("A: fork");
        exit(16);
    }
    if (pA_2 == 0)
    {
        change_pname("C");
        printf("C: Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("C: Exiting...\n");
        exit(17);
    }
    // A waiting for its 2 children
    //  for (int i = 0; i < 2; ++i)
    //{
    for (int i = 0; i < 2; ++i)
    {
        printf("A: Waiting...\n");
        pA_2 = wait(&statusA);
        explain_wait_status(pA_2, statusA);
    }
    //  }
    printf("A: Exiting...\n");
    exit(16);
}

int main()
{
```

```c
    pid_t p;
    int status;
    /* Fork root of process tree */
    printf("Creating... A\n");
    p = fork();
    if (p < 0)
    {
        perror("main: fork");
        exit(1);
    }

    if (p == 0)
    {
        /* Child */
        fork_procs();
        exit(1);
    }

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

    /* Print the process tree root at pid */

    show_pstree(getpid());

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

    return 0;
```

```
}
```

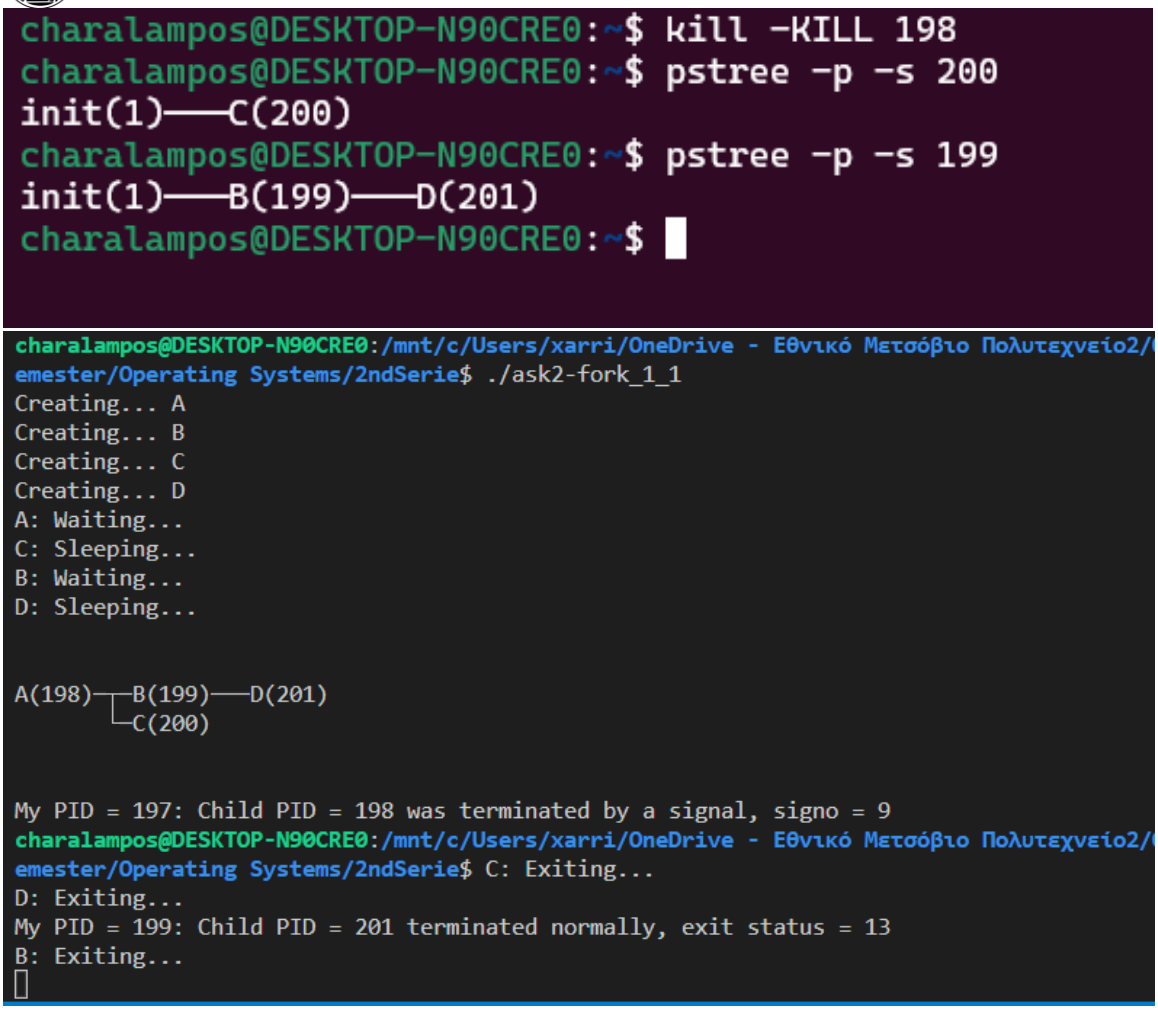
Επίσης παρακάτω παραθέτουμε το Makefile το οποίο δεν αλλάζει σε όλη τη διάρκεια της άσκησης:

```
.PHONY: all clean

all: ask2-fork_1_1 ask2-tree_1_2 ask2-signals_1_3 ask2-pipes_1_4

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

ask2-fork_1_1: ask2-fork_1_1.o proc-common.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

ask2-tree_1_2: ask2-tree_1_2.o proc-common.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

ask2-signals_1_3: ask2-signals_1_3.o proc-common.o tree.o
    $(CC) $(CFLAGS) $^ -o $@

ask2-pipes_1_4: ask2-pipes_1_4.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 pstree-this ask2-fork_1_1,ask2-tree_1_2,ask2-signals_1_3,ask2-pipes_1_4

```

Ερωτήσεις:

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

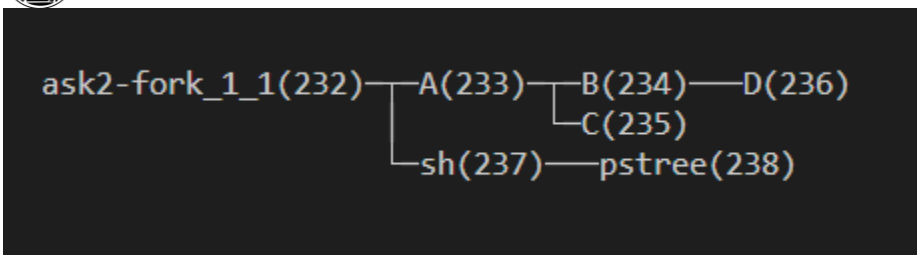
Αν τερματίσουμε πρόωρα τη διεργασία A, δίνοντας kill -KILL (από ένα άλλο παράθυρο του terminal μας) τότε έχουμε το εξής στα terminal:



Γνωρίζουμε ότι όταν ένα parent process πεθάνει πρώτα από τα "παιδιά" του, η διεργασία init (PID=1) κληρονομεί όλα τα "ορφανά" children processes του, η οποία κάνει συνεχώς wait(). Παρατηρούμε λοιπόν, ότι η διεργασία A τερματίστηκε από το KILL (9) με signal που στείλαμε από άλλο terminal. Επίσης βλέπουμε στο άλλο terminal ότι οι διεργασίες παιδιά της A κληρονομούνται από την init(1).

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

Τώρα, τροποποιούμε την main() έτσι ώστε αντί για show_pstree(pid_root) να κάνουμε show_pstree(getpid()). Παρατηρούμε λοιπόν, ότι τώρα στο δέντρο εμφανίζονται και οι διεργασίες sh και pstree. Η ρίζα του είναι το πρόγραμμά μας (αφού κάναμε show_pstree(getpid())), που δημιουργεί το δεδομένο δέντρο διεργασιών της άσκησης που αναφέραμε παραπάνω. Εκτός από αυτό, καλώντας την show_pstree(), εμφανίζει το δέντρο διεργασιών με ρίζα μία δεδομένη διεργασία.



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

Σε υπολογιστικά συστήματα πολλαπλών χρηστών, πολλές φορές ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ο κάθε χρήστης. Αυτό γίνεται για λόγους επίδοσης και ασφάλειας του συστήματος. Παράδειγμα θα μπορούσε ένας κακόβουλος χρήστης να γεννάει συνεχώς διεργασίες σε έναν ατέρμον βρόχο (σε εκθετικό ρυθμό). Αυτό θα οδηγούσε την CPU σε εξάντληση και τον πίνακα διεργασιών σε κορεσμό με αποτέλεσμα να μην μπορέσει να δημιουργηθεί άλλη διεργασία και ο υπολογιστής να μην ανταποκρίνεται. Αυτό είναι γνωστό και ως fork bomb.

Περιορίζοντας τον αριθμό των διεργασιών που έχει ο κάθε χρήστης, αποτρέπουμε τέτοια σοβαρά προβλήματα.

1.2. ΔΗΜΙΟΥΡΓΙΑ ΑΥΘΑΙΡΕΤΟΥ ΔΕΝΤΡΟΥ ΔΙΕΡΓΑΣΙΩΝ

Ο πηγαίος κώδικας της άσκησης είναι ο εξής:

```c
#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
```

```c
/*
 * Create this process tree:
 * A-+-B---D
 *   `-C
 */

void fork_procs(struct tree_node *root)
{
    pid_t pid;
    int status;
    int i;
    change_pname(root->name);
    printf("%s: Creating...\n", root->name);

    if (root->nr_children == 0) // if the process is leaf
    {
        printf("%s: Sleeping...\n", root->name);
        sleep(SLEEP_PROC_SEC);
        printf("%s: Exiting...\n", root->name);
        exit(10);
    }
    for (i = 0; i < root->nr_children; ++i)
    {
        pid = fork();
        if (pid < 0)
        {
            fprintf(stderr, "%s : fork", root->name);
            exit(1);
        }
        if (pid == 0)
        {
            fork_procs(root->children + i);
        }
```

```
    }
    printf("%s: Waiting...\n", root->name);
    for (int i = 0; i < root->nr_children; i++)
    {
        pid = wait(&status);
        if (pid < -1)
        {
            perror("wait");
            exit(1);
        }
        explain_wait_status(pid, status);
    }
    printf("%s : Exiting..\n", root->name);
    exit(11);
}


/*
 * The initial process forks th
 e root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-{fork, tree}:
 *      wait for a few seconds, hope for the best.
 * In ask2-signals:
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */

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

```
    pid_t pid;
    int status;
    struct tree_node *root;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
        exit(1);
    }

    root = get_tree_from_file(argv[1]);
    print_tree(root);

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

    sleep(SLEEP_TREE_SEC);

    /* Print the process tree root at pid */

    show_pstree(pid);

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

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


    return 0;
}
```

Ερώτηση:

1)Με ποια σειρά εμφανίζονται τα μηνύματα έναρξης και τερματισμού των διεργασιών; γιατί;

Η σειρά με την οποία εμφανίζονται τα μηνύματα έναρξης, αναμονής και τερματισμού των διεργασιών είναι τυχαία, διαφέρει από εκτέλεση σε εκτέλεση και εξαρτάται από τον χρονοδρομολογητή (scheduler) του λειτουργικού συστήματος. Παρακάτω βλέπουμε 2 εκτελέσεις και επαληθεύουμε τη διαφορετική σειρά αποτελεσμάτων κάθε φορά.

```
charalampos@DESKTOP-N90CRE0:/mnt/c/Users/xarri/OneDrive - Εθνικό Μετσόβιο Πολυτεχνείο2/
emester/Operating Systems/2ndSerie$ ./ask2-tree_1_2 proc.tree
A
        B
                E
                F
        C
        D
A: Creating...
B: Creating...
C: Creating...
C: Sleeping...
E: Creating...
B: Waiting...
A: Waiting...
F: Creating...
D: Creating...
E: Sleeping...
F: Sleeping...
D: Sleeping...


A(649)──B(650)──E(652)
        │       └─F(654)
        ├─C(651)
        └─D(653)


C: Exiting...
My PID = 649: Child PID = 651 terminated normally, exit status = 10
E: Exiting...
My PID = 650: Child PID = 652 terminated normally, exit status = 10
D: Exiting...
F: Exiting...
My PID = 649: Child PID = 653 terminated normally, exit status = 10
My PID = 650: Child PID = 654 terminated normally, exit status = 10
B : Exiting..
My PID = 649: Child PID = 650 terminated normally, exit status = 11
A : Exiting..
My PID = 648: Child PID = 649 terminated normally, exit status = 11
```

```
charalampos@DESKTOP-N90CRE0:/mnt/c/Users/xarri/OneDrive - Εθνικό Μετσόβιο Πολυτεχνείο2/
emester/Operating Systems/2ndSerie$ ./ask2-tree_1_2 proc.tree
A
        B
                E
                F
        C
        D
A: Creating...
B: Creating...
C: Creating...
C: Sleeping...
E: Creating...
A: Waiting...
D: Creating...
E: Sleeping...
D: Sleeping...
B: Waiting...
F: Creating...
F: Sleeping...


A(668)─┬─B(669)─┬─E(671)
       │        └─F(673)
       ├─C(670)
       └─D(672)


C: Exiting...
My PID = 668: Child PID = 670 terminated normally, exit status = 10
D: Exiting...
E: Exiting...
My PID = 668: Child PID = 672 terminated normally, exit status = 10
My PID = 669: Child PID = 671 terminated normally, exit status = 10
F: Exiting...
My PID = 669: Child PID = 673 terminated normally, exit status = 10
B : Exiting..
My PID = 668: Child PID = 669 terminated normally, exit status = 11
A : Exiting..
My PID = 667: Child PID = 668 terminated normally, exit status = 11
```

Μέρος 2ο : Διαδιεργαστική επικοινωνία

1.3.ΑΠΟΣΤΟΛΗ ΚΑΙ ΧΕΙΡΙΣΜΟΣ ΣΗΜΑΤΩΝ

Ο πηγαίος κώδικας της άσκησης είναι ο εξής:

```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"
```

```c
void fork_procs(struct tree_node *root)
{
    /*
     * Start
     */
    printf("PID = %ld, name %s, starting...\n",(long)getpid(), root->name);
    change_pname(root->name);
    if(root->nr_children == 0)
    {
        printf("%s: Stoping...\n", root->name);
        raise(SIGSTOP);
        printf("PID = %ld, name = %s is awake\n",(long)getpid(), root->name);
        exit(20);
    }
    pid_t pid[root->nr_children]; // We create an array where the parent-process will save
the children's pids
    for (int i = 0; i < root->nr_children; ++i)
    {
        pid[i] = fork();
        if (pid[i] < 0)
        {
            fprintf(stderr, "%s : fork", root->name);
            exit(1);
        }
        if (pid[i] == 0)
        {
            fork_procs(root->children + i);
        }
    }
    /*........*/
    printf("%s: Waiting for my children to stop...", root->name);
    wait_for_ready_children(root->nr_children);
    /*
     * Suspend Self
     */
    printf("%s: Stoping...\n", root->name);
    raise(SIGSTOP);
    /* ... */
    printf("PID = %ld, name = %s is awake\n",(long)getpid(), root->name);

    int status;
    for (int i = 0; i < root->nr_children; ++i)
    {
        kill(pid[i], SIGCONT); //let's send a wake up signal to each of our children
        pid[i] = wait(&status);
```

```
        explain_wait_status(pid[i],status);
    }
    /*
     * Exit
     */

    exit(0);
}

/*
 * The initial process forks the root of the process tree,
 * waits for the process tree to be completely created,
 * then takes a photo of it using show_pstree().
 *
 * How to wait for the process tree to be ready?
 * In ask2-signals:_
 *      use wait_for_ready_children() to wait until
 *      the first process raises SIGSTOP.
 */

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

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

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

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

```
        exit(1);
    }

    /*
     * Father
     */
    /* for ask2-signals */
    wait_for_ready_children(1); // the father of the root process waits for its child

    /* Print the process tree root at pid */
    show_pstree(pid);
    printf("");
    /* for ask2-signals */
    if (kill(pid, SIGCONT) == -1)
    {
        perror("kill");
        exit(1);
    }

    /* Wait for the root of the process tree to terminate */
    if (wait(&status) == -1)
    {
        perror("wait");
        exit(1);
    }
    explain_wait_status(pid, status);

    return 0;
}
```

Ερωτήσεις:

1) Στις προηγούμενες ασκήσεις χρησιμοποιήσαμε τη sleep() για τον συγχρονισμό

των διεργασιών. Τι πλεονεκτήματα έχει η χρήση σημάτων;

Με τη χρήση του sleep() στις προηγούμενες διεργασίες επιδιώξαμε να καθυστερήσουμε αρκετά την έξοδο των «φύλλων» διεργασιών του δέντρου που μας δινόταν σε κάθε περίπτωση ώστε να προλάβει να σχηματιστεί ολόκληρο το δένδρο διεργασιών και να το δει ο χρήστης με την κλήση της show_pstree(). Με την sleep() αναστείλαμε προσωρινά την πρόοδο εκτέλεσης των διεργασιών για κάποια δευτερόλεπτα που ορίσαμε αυθαίρετα και ήταν αρκετά για παραδείγματα μικρών δένδρων. Ωστόσο, εκτός από την αχρείαστη καθυστέρηση που εισάγουμε στην εκτέλεση του προγράμματος μας αφού δεν εκμεταλλευόμαστε πλήρως την υπολογιστική ισχύ που

διαθέτουμε, σε μεγαλύτερα δένδρα θα πρέπει να εισάγουμε μεγαλύτερα χρονικά διαστήματα sleep() για να εκτελεστεί το πρόγραμμα σωστά άρα η καθυστέρηση θα γίνει ανυπόφορη.

Χρησιμοποιώντας τα σήματα πετυχαίνουμε μια ντετερμινιστική επίλυση του προγράμματος αφού δεν στηριζόμαστε σε τυχαίες καθυστερήσεις που εισάγουμε με την ελπίδα να είναι επαρκείς για την σωστή εκτέλεση του κώδικα. Ακόμη, το πρόγραμμα εκτελείται με ροή, αφού η κάθε διεργασία ξέρει ακριβώς πότε θα σταματήσει(με χρήση της SIGSTOP)και πότε θα συνεχίσει (με χρήση της SIGCONT) και άρα ταχύτατα ακόμη και για μεγάλα δένδρα. Επίσης, δε χρειάζεται να κάνουμε αλλαγές στον κώδικα όπως θα χρειαζόταν στο sleeping time για μεγαλύτερα δένδρα. Έτσι, η show_pstree() θα λειτουργεί πάντα ορθά. Τέλος, με τη χρήση σημάτων το δέντρο διεργασιών είναι απόλυτα ελεγχόμενο αφού έχουμε πλέον τη δυνατότητα οποιασδήποτε μορφής διάσχισης των κόμβων-διεργασιών(όπως η DFS που ζητείται στην άσκηση).

Συνεπώς, πετυχαίνουμε με τα σήματα έναν πιο αποοτελεσματικό συγχρονισμό των διεργασιών του δένδρου μέσω της διαδεργαστικής επικοινωνίας που προσφέρουν και κερδίζουμε ταχύτητα εκτέλεσης, γενικότερη χρήση του κώδικα, εξασφάλιση σωστής εκτέλεσης της show_pstree() και έλεγχο διάσχισης κόμβων.

Η έξοδος του προγράμματος όπου φαίνεται η DFS προσπέλαση των κόμβων έχοντας ως είσοδο το proc.tree που δίνεται φαίνεται παρακάτω:

```
chris@LAPTOP-TK5Q3T95:/mnt/c/Users/koust/ECE/6th Semester/OS/os-lab-exer2/ex2$ ./ask2-signals_1_3 proc.tree
PID = 237, name A, starting...
PID = 238, name B, starting...
PID = 239, name C, starting...
C: Stoping...
PID = 240, name D, starting...
My PID = 237: Child PID = 239 has been stopped by a signal, signo = 19
D: Stoping...
PID = 241, name E, starting...
E: Stoping...
My PID = 237: Child PID = 240 has been stopped by a signal, signo = 19
PID = 242, name F, starting...
My PID = 238: Child PID = 241 has been stopped by a signal, signo = 19
F: Stoping...
My PID = 238: Child PID = 242 has been stopped by a signal, signo = 19
B: Waiting for my children to stop...B: Stoping...
My PID = 237: Child PID = 238 has been stopped by a signal, signo = 19
A: Waiting for my children to stop...A: Stoping...
My PID = 236: Child PID = 237 has been stopped by a signal, signo = 19


A(237)─┬─B(238)─┬─E(241)
       │        └─F(242)
       ├─C(239)
       └─D(240)


PID = 237, name = A is awake
PID = 238, name = B is awake
PID = 241, name = E is awake
My PID = 238: Child PID = 241 terminated normally, exit status = 20
PID = 242, name = F is awake
My PID = 238: Child PID = 242 terminated normally, exit status = 20
My PID = 237: Child PID = 238 terminated normally, exit status = 0
PID = 239, name = C is awake
My PID = 237: Child PID = 239 terminated normally, exit status = 20
PID = 240, name = D is awake
My PID = 237: Child PID = 240 terminated normally, exit status = 20
My PID = 236: Child PID = 237 terminated normally, exit status = 0
```

2. Ποιος ο ρόλος της wait_for_ready_children(); Τι εξασφαλίζει η χρήση της

και τι πρόβλημα θα δημιουργούσε η παράλειψή της;

Για να κατανοήσουμε πλήρως τη λειτουργία της wait_for_ready_children() ανατρέχουμε στο αρχείο proc-common.c .

```
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);
        }
    }
}
```

Διαπιστώνουμε ότι με τη συνάρτηση αυτή η γονική διεργασία που την καλεί περιμένει την αλλαγή κατάστασης του πλήθους των παιδιών-διεργασιών που της δίνεται σαν όρισμα. Συγκεκριμένα, τα παιδιά μιας γονικής διεργασίας θα αλλάξουν κατάσταση στον κώδικα μας όταν σταματήσουν με χρήση του σήματος SIGSTOP. Το system call waitpid() έχει στην πρώτη παράμετρο του -1 που σημαίνει ότι περιμένει για οποιοσδήποτε παιδί-διεργασία να αλλάξει κατάσταση ενώ το flag WUNTRACED εξασφαλίζει τον εντοπισμό διεργασίας-παιδί που θα σταματήσει και χρειάζεται για την μετέπειτα κλήση της WIFSTOPPED που επιστρέφει true αν μία διεργασία-παιδί σταμάτησε με κλήση σήματος που είναι το επιθυμητό. Αλλιώς, εμφανίζεται αντίστοιχο μήνυμα σφάλματος.

Αναλύοντας τον κώδικα της wait_for_ready_children() καταλήγουμε στο συμπέρασμα ότι η χρήση της είναι αναγκαία . Τέλος, είναι φανερό πως απαιτείται η κλήση της από τη γονική διεργασία πριν το σταμάτημα της με raise(SIGSTOP) αφού, η γονική διεργασία δεσμεύεται να εξασφαλίζει το σταμάτημα όλων παιδιών της αναδρομικά πριν η ίδια παύσει την εκτέλεση της.

Η παράλειψη της χρήσης της wait_for_ready_children() δε θα επέτρεπε τον συγχρονισμό

1.4.ΠΑΡΑΛΛΗΛΟΣ ΥΠΟΛΟΓΙΣΜΟΣ ΑΡΙΘΜΗΤΙΚΗΣ ΕΚΦΡΑΣΗΣ

Ο πηγαίος κώδικας της άσκησης αυτής είναι ο εξής:

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

```c
#include <sys/wait.h>
#include <string.h>
#include <signal.h>

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

void fork_procs(struct tree_node *root, int fd)
{
    int status;
    change_pname(root->name);
    printf("%s(%ld) is created...\n", root->name, (long)getpid());

    // If the node is a leaf
    if (root->nr_children == 0)
    {
        printf("The leaf node %s is created...\n", root->name);
        int value = atoi(root->name);
        if (write(fd, &value, sizeof(value)) != sizeof(value))
        {
            perror("Leaf : write");
            exit(1);
        }
        close(fd);
        raise(SIGSTOP);
        exit(10);
    }

    int pfd[2];
    pid_t pid[root->nr_children]; // pid[2] for us
    printf("%s creating a pipe...\n", root->name);
    if (pipe(pfd) < 0)
    {
        perror("pipe");
        exit(1);
    }
    /*If the parent wants to receive data from the child, it should close pfd[1],
    and the child should close pfd[0]. If the parent wants to send data to the child, it
should close fd[0], and the child should close fd[1].
    Since descriptors are shared between the parent and child, we should always be sure to
close the end of pipe we aren't concerned with. On a technical note, the EOF will never be
returned if the unnecessary ends of the pipe are not explicitly closed. */
    int value[root->nr_children];
    for (int i = 0; i < root->nr_children; ++i)
    {
        pid[i] = fork();
```

```
        if (pid[i] < 0)
        {
            fprintf(stderr, "%s : fork\n", root->name);
            exit(1);
        }
        if (pid[i] == 0)
        {
            close(pfd[0]);
            fork_procs(root->children + i, pfd[1]);
            exit(10);
        }
    }
    close(pfd[1]);
    for (int i = 0; i < root->nr_children; ++i)
    {
        if (read(pfd[0], &value[i], sizeof(value[i])) != sizeof(value[i]))
        {
            perror("read");
            exit(1);
        }
    }

    close(pfd[0]);
    int result;
    if (!strcmp(root->name, "+"))
    {
        result = value[0] + value[1];
        printf("Node %ld : %d + %d = %d\n", (long)getpid(), value[0], value[1], result);
    }

    if (!strcmp(root->name, "*"))
    {
        result = value[0] * value[1];
        printf("Node %ld : %d * %d = %d\n", (long)getpid(), value[0], value[1], result);
    }
    if (write(fd, &result, sizeof(result)) != sizeof(result))
    { // write to parent
        perror("write to pipe");
        exit(1);
    }
    close(fd);
    raise(SIGSTOP); // then stops

    // Let's wake up our children
    for (int i = 0; i < root->nr_children; ++i)
    {
```

```
        kill(pid[i], SIGCONT);
        pid[i] = wait(&status);
        explain_wait_status(pid[i], status);
        // printf("My child with PID : %d sent me %d\n", pid[i], value[i]);
    }
    exit(0);
}

int main(int argc, char *argv[])
{
    pid_t pid;
    int status;
    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]);
    int pfd[2];
    if (pipe(pfd) < 0)
    {
        perror("pipe");
        exit(1);
    }

    int value;
    pid = fork();
    if (pid < 0)
    {
        perror("main: fork");
        exit(1);
    }
    if (pid == 0)
    {
        close(pfd[0]);
        fork_procs(root, pfd[1]);
        exit(0);
    }

    close(pfd[1]);

    if (read(pfd[0], &value, sizeof(value)) != sizeof(value))
    {
        perror("read");
```

```
        exit(1);
    }

    close(pfd[0]);

    /* Print the process tree root at pid */
    show_pstree(pid);
    printf("\n\n");
    if (kill(pid, SIGCONT) == -1)
    {
        perror("kill");
        exit(1);
    }

    /* Wait for the root of the process tree to terminate */
    if (wait(&status) == -1)
    {
        perror("wait");
        exit(1);
    }
    explain_wait_status(pid, status);
    printf("Done... Final result is: %d\n", value);

    return 0;
}
```

και ακολουθεί η έξοδος στο τερματικό που επιεβεβαιώνει την ορθή λειτουργία του:

```
chris@LAPTOP-TK5Q3T95:/mnt/c/Users/koust/ECE/6th Semester/OS/os-lab-exer2/ex2$ ./ask2-pipes_1_4 expr.tree
+(317) is created...
+ creating a pipe...
10(318) is created...
The leaf node 10 is created...
*(319) is created...
* creating a pipe...
+(320) is created...
+ creating a pipe...
4(321) is created...
The leaf node 4 is created...
5(322) is created...
The leaf node 5 is created...
7(323) is created...
The leaf node 7 is created...
Node 320 : 5 + 7 = 12
Node 319 : 4 * 12 = 48
Node 317 : 10 + 48 = 58


+(317)─┬─*(319)─┬─+(320)─┬─5(322)
       │        │        └─7(323)
       │        └─4(321)
       └─10(318)



My PID = 317: Child PID = 318 terminated normally, exit status = 10
My PID = 320: Child PID = 322 terminated normally, exit status = 10
My PID = 320: Child PID = 323 terminated normally, exit status = 10
My PID = 319: Child PID = 320 terminated normally, exit status = 0
My PID = 319: Child PID = 321 terminated normally, exit status = 10
My PID = 317: Child PID = 319 terminated normally, exit status = 0
My PID = 316: Child PID = 317 terminated normally, exit status = 0
Done... 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;

Οι αριθμητικες πράξεις που εκτελούνται στα δέντρα αριθμητικών εκφράσεων στη συγκεκριμένη άσκηση είναι η πρόσθεση «+» και ο πολλαπλασιασμός «*» που είναι αντιμεταθετικές. Έτσι, είναι εφικτή η χρήση μοναδικής σωλήνωσης από κάθε γονική διεργασία αφού δεν παίζει ρόλο η σειρά επιστροφής της αποτίμησης του κάθε κόμβου-παιδιού. Επομένως, κάθε διεργασία που αντιστοιχεί σε μη τερματικό κόμβο διαθέτει δύο σωληνώσεις: Μία για την επιστροφή της αποτίμησης της αξίας της στη γονική της και μία για τη λήψη των τιμών από τις διεργασίες-παιδιά της. Από την άλλη οι τερματικοί κόμβοι, χρησιμοποιούν μία μόνο σωλήνωση για να επιστρέψουν την τιμή τους στη γονική διεργασία.

Στην περίπτωση εκτέλεσης μη αντιμεταθετικών πράξεων όπως διαίρεσης «/» και αφαίρεσης «-» θα ήταν απαραίτητο η γονική διεργασία να αφιερώσει ένα pipe() για κάθε παιδί της ώστε να μπορούμε να γνωρίζουμε ποιος τελεστέος θα είναι το κάθε παιδί-διεργασία.

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;

Ένα σύστημα πολλαπλών επεξεργαστών μπορεί να εξασφαλίζει μεγαλύτερη ταχύτητα και αποδοτικότητα. Συγκεκριμένα, τα οφέλη είναι:

- Παραλληλισμός : Διαιρώντας σε επιμέρους υποεκφράσεις μία έκφραση αξιοποιούμε καλύτερα του πόρους του συστήματος και χρησιμοποιούμε πιο αποδοτικά του πολλαπλούς επεξεργαστές.
- Διαμοιρασμός φόρτου εργασίας των διεργασιών.
- Ανοχή σε σφάλαματα του συστήματος: Αν μία διεργασία αποτύχει και σταματήσει απροσδόκητα τότε οι άλλες διεργασίες συνεχίζουν να εκτελούν τις λειτουργίες τους εξασφαλίζοντας ένα λειτουργικό σύστημα στο σύνολο παρά την ύπαρξη μεμονομένων προβλημάτων.