Λειτουργικά Συστήματα – Άσκηση 2

Κυριακόπουλος Γιώργος – el18153
Τζελέπης Σεραφείμ – el18849

1.1

Πηγαίος κώδικας:

```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)
{
    /* Initial process is A */
    printf("A: Initiating...\n");
    change_pname("A");

    pid_t pidB, pidC, pidD;
    int status;

    /* Child B */
    pidB = fork();

    if(pidB < 0) {
        perror("fork_procs: forkB");
```

```
    exit(1);
}
if(pidB == 0) {
    printf("B: Initiating...\n");
    change_pname("B");

    /* Child D */
    pidD = fork();

    if(pidD < 0) {
        perror("fork_procs: forkD");
        exit(1);
    }
    if(pidD == 0) {
        printf("D: Initiating...\n");
        change_pname("D");
        printf("D: Sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        exit(13);
    }

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

/* Child C */
pidC = fork();

if(pidC < 0) {
    perror("fork_procs: forkC");
    exit(1);
}
if(pidC == 0) {
    printf("C: Initiating...\n");
    change_pname("C");
    printf("C: Sleeping...\n");
    sleep(SLEEP_PROC_SEC);
    exit(17);
}

printf("A: Waiting...\n");
```

```c
    pidC = wait(&status);
    explain_wait_status(pidC, status);

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

    exit(16);
}

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 A */
        fork_procs();
        exit(1);
    }

    /* Father */
    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;
}
```

Έξοδος εκτέλεσης προγράμματος:

$ ./ask2-fork

```
A: Initiating...
A: Waiting...
B: Initiating...
C: Initiating...
C: Sleeping...
B: Waiting...
D: Initiating...
D: Sleeping...

A(871)─┬─B(872)──D(874)
       └─C(873)

My PID = 871: Child PID = 873 terminated normally, exit status = 17
My PID = 872: Child PID = 874 terminated normally, exit status = 13
My PID = 871: Child PID = 872 terminated normally, exit status = 19
My PID = 870: Child PID = 871 terminated normally, exit status = 16
```

Ερωτήσεις:

1. Εάν τερματίσουμε πρόωρα τη διεργασία A με τη χρήση του kill() system call, τότε θα εμφανιστεί το μήνυμα τερματισμού της διεργασίας A “terminated by a signal” και μετά οι διεργασίες B, C, D θα τερματιστούν επίσης, με τη διεργασία D να είναι η μόνη που θα βγάλει κάποιο μήνυμα τερματισμού “terminated normally”, καθώς οι ορφανές διεργασίες B, C θα υιοθετηθούν από την init (PID = 1), η οποία θα εκτελέσει το wait system για αυτές, μέχρι να τερματίσουν. Μπορούμε να ελέγξουμε ότι πράγματι οι διαδικασίες αυτές τερματίζουν κανονικά με τη χρήση του ps -ef command μία φορά αφού σκοτώσουμε τη διεργασία A και μία ακόμα μερικά δευτερόλεπτα μετά, όταν η D θα έχει τερματιστεί φυσιολογικά και οι διεργασίες B, C θα έχουν τερματίσει με τη βοήθεια της init, χωρίς να παράγουν κάποιο μήνυμα τερματισμού, όπως αναφέραμε και νωρίτερα, αφού αυτό αναμενόταν να το παράγει η A μέσω του wait() και explain_wait_status(), που έχει γίνει kill.

2. Οι νέες διεργασίες ask2-fork (root), sh (child) και pstree (grandchild) εμφανίζονται στο δέντρο διεργασιών σε αυτήν την περίπτωση, ως ένα δεύτερο κλαδί με το πρώτο να είναι το προηγούμενο (με χρήση της pid αντί για getpid()) με τις διεργασίες A, B, C, D. Αυτό συμβαίνει διότι η κλήση system μέσα στην show_pstree χρησιμοποιεί τη fork για να δημιουργήσει μία διεργασία παιδί που θα εκτελέσει την εντολή που ετοιμάζει η sprint (μαζί με το pstree).

3. Εάν δεν υπήρχε ένα τέτοιο όριο διεργασιών σε κάθε χρήστη, θα μπορούσε ένας να δημιουργήσει, είτε επίτηδες είτε κατά λάθος, έναν αρκετά μεγάλο αριθμό διεργασιών, ικανό να εξαντλήσει τη μνήμη του συστήματος. Μία περίπτωση αυτού είναι και τα fork bombs, τα οποία αποτρέπονται από αυτό το όριο, ειδάλλως η εκτέλεση ενός τέτοιου attack είτε θα επιβράδυνε το σύστημα, είτε θα προκαλούσε κάποιο crash, λόγω της έλλειψης πόρων.

1.2

Πηγαίος κώδικας:

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

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

#define SLEEP_PROC_SEC  5
#define SLEEP_TREE_SEC  3

void fork_procs(struct tree_node *root) {

    change_pname(root->name);
    printf("%s is initializing with PID: %ld  \n", root-
>name, (long) getpid());

    /* Parent processes with children */
    if(root->children) {
        int status;
        pid_t pid;
        int i;

        for(i = 0; i < root->nr_children; i++){
            pid = fork();

            if(pid < 0) {
                perror("fork_procs: fork");
                exit(1);
            }
            else if(pid == 0){
                /* Child calls the function recursively*/
                fork_procs(root->children + i);
            }
        }
        /* Parent waits for all its children */
        for(i = 0; i < root->nr_children; i++){
```

```
            pid = wait(&status);
            explain_wait_status(pid, status);
        }
        /* Then exit */
        exit(0);
    }

    /* Leaves with no children */
    else {
        /* Leaves sleep for 5 seconds then exit */
        sleep(SLEEP_PROC_SEC);
        exit(0);
    }
}

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

    int status;

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

    root = get_tree_from_file(argv[1]);

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

    if(p < 0) {
        perror("main: fork");
        exit(1);
    }
    if(p == 0) {
        /* Root of given tree */
        fork_procs(root);
        exit(0);
    }

    sleep(SLEEP_TREE_SEC);
```

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

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

    return 0;
}
```

Έξοδος εκτέλεσης προγράμματος:

$ ./ask2-fork proc.tree

```
A is initializing with PID: 782
B is initializing with PID: 783
C is initializing with PID: 784
D is initializing with PID: 785
E is initializing with PID: 786
F is initializing with PID: 787

A(782)─┬─B(783)─┬─E(786)
       │        └─F(787)
       ├─C(784)
       └─D(785)

My PID = 783: Child PID = 786 terminated normally, exit status = 0
My PID = 782: Child PID = 784 terminated normally, exit status = 0
My PID = 782: Child PID = 785 terminated normally, exit status = 0
My PID = 783: Child PID = 787 terminated normally, exit status = 0
My PID = 782: Child PID = 783 terminated normally, exit status = 0
My PID = 781: Child PID = 782 terminated normally, exit status = 0
```

Ερωτήσεις:

1. Οι διεργασίες δημιουργούνται κατά πλάτος, όπως είναι φανερό και από τα PIDs τους, όμως τα μηνύματα αρχικοποίησης τους δεν είναι πάντα τυπωμένα με κατά πλάτος σειρά, καθώς αυτή η σειρά είναι ουσιαστικά τυχαία και εξαρτάται από τον kernel scheduler και τη σειρά εκτέλεσης που θα διαλέξει αυτός. Κατά πλάτος σημαίνει ότι όλες οι διεργασίες του τρέχοντος

επιπέδου θα δημιουργηθούν πριν δημιουργηθούν οι διεργασίες του επόμενου επιπέδου, εξού και η σειρά δημιουργίας A μετά B, C, D και τέλος οι διεργασίες E, F.

Τα μηνύματα εξόδου τυπώνονται αρχίζοντας από τα φύλλα που εκτελούνται πρώτα και τερματίζουν, με βάση και την σειρά που επιλέγει ο scheduler, ακολουθούμενα από τον πατέρα αυτών και με όμοια λογική προς τα πάνω. Για παράδειγμα, μπορεί τα πρώτα μηνύματα εξόδου να έρθουν από τα φύλλα E, F που έχουν εκτελεστεί και τερματίσει πρώτα, στη συνέχεια από τον πατέρα τους B, μετά τα υπόλοιπα φύλλα C, D και τέλος το A.

1.3

Πηγαίος κώδικας:

```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"

void fork_procs(struct tree_node *root)
{
    change_pname(root->name);
    printf("PID = %ld, name = %s is initializing\n", (long)getpid(),
root->name);

    /* Parent processes with children */
    if(root->children) {
        /* Create an array of children's PIDs for later use */
        pid_t pid[root->nr_children];
        int i;

        for(i = 0; i < root->nr_children; i++){
            pid[i] = fork();

            if(pid[i] < 0) {
                perror("forker: fork");
                exit(1);
            }
            if(pid[i] == 0){
                /* Child calls the function recursively*/
                fork_procs(root->children + i);
            }

        }
        /* Father */
        /* Waits for all its children to be stopped */
        wait_for_ready_children(root->nr_children);
```

```c
        /* Raises SIGSTOP when all children are stopped */
        raise(SIGSTOP);

        /* Awakes when he receives SIGCONT from his parent */
        printf("PID = %ld, name = %s is awake\n", (long)getpid(), roo
t->name);
        int status;

        /* Awakes all its children one by one with use of kill(pid[i]
, SIGCONT)
        and the array of their PIDs, and waits for them to exit */
        for(i = 0; i < root->nr_children; i++){
            kill(pid[i], SIGCONT);
            pid[i] = wait(&status);
            explain_wait_status(pid[i], status);
        }
        /* Then exits himself */
        exit(0);
    }

    /* Leaves with no children */
    else {
        /* Leaves raise SIGSTOP */
        raise(SIGSTOP);

        /* Leaves receive SIGCONT from parent process, are awaken the
n exit */
        printf("PID = %ld, name = %s is awake\n", (long)getpid(), roo
t->name);
        exit(0);
    }
}

int main(int argc, char *argv[])
{
    pid_t p;
    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 */
    p = fork();

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

    /* Father */
    wait_for_ready_children(1);

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

    /* Sends SIGCONT to root process */
    kill(p, SIGCONT);

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

    return 0;
}
```

Έξοδος εκτέλεσης προγράμματος:

$ ./ask2-signals proc.tree

PID = 1057, name = A is initializing
PID = 1058, name = B is initializing
PID = 1059, name = C is initializing
My PID = 1057: Child PID = 1059 has been stopped by a signal, signo = 19
PID = 1060, name = D is initializing

```
PID = 1062, name = F is initializing
PID = 1061, name = E is initializing
My PID = 1058: Child PID = 1061 has been stopped by a signal, signo = 19
My PID = 1058: Child PID = 1062 has been stopped by a signal, signo = 19
My PID = 1057: Child PID = 1058 has been stopped by a signal, signo = 19
My PID = 1057: Child PID = 1060 has been stopped by a signal, signo = 19
My PID = 1056: Child PID = 1057 has been stopped by a signal, signo = 19

A(1057)─┬─B(1058)─┬─E(1061)
        |         └─F(1062)
        ├─C(1059)
        └─D(1060)

PID = 1057, name = A is awake
PID = 1058, name = B is awake
PID = 1061, name = E is awake
My PID = 1058: Child PID = 1061 terminated normally, exit status = 0
PID = 1062, name = F is awake
My PID = 1058: Child PID = 1062 terminated normally, exit status = 0
My PID = 1057: Child PID = 1058 terminated normally, exit status = 0
PID = 1059, name = C is awake
My PID = 1057: Child PID = 1059 terminated normally, exit status = 0
PID = 1060, name = D is awake
My PID = 1057: Child PID = 1060 terminated normally, exit status = 0
My PID = 1056: Child PID = 1057 terminated normally, exit status = 0
```

Ερωτήσεις:

1. Με τη χρήση σημάτων μπορούμε να συγχρονίσουμε τις εντολές μας ώστε να δημιουργήσουμε μια συγκεκριμένη σειρά εκτέλεσης τους, χωρίς ούτε να απαιτείται η εκτίμηση του χρόνου που απαιτούν είτε αυτές είτε άλλες εντολές για να εκτελεστούν, ούτε να ξοδέψουμε άσκοπα δευτερόλεπτα περιμένοντας (π.χ. μέσω της sleep()), ώστε να είμαστε σίγουροι ότι οι εντολές μας δεν θα αλληλεπικαλυφθούν ή ότι δεν θα εκτελεστούν πρόωρα.

2. Χρειαζόμαστε την διεργασία πατέρα να περιμένεις για όλε τις διεργασίες παιδιά της να σταματήσουν από ένα σήμα SIGSTOP πριν συνεχίσει η εκτέλεση του υπόλοιπου κώδικα. Αυτό το σήμα δεν μπορεί να ανιχνευθεί από την wait() συνάρτηση, αφού δεν μπορεί να διακρίνει πότε ένα παιδί έχει σταματήσει τη λειτουργία του. Μπορούμε να χρησιμοποιήσουμε την waitpid() συνάρτηση με την WUNTRACED flag, ώστε να πάρουμε feedback εάν ένα παιδί έχει σταματήσει. Αυτή είναι ουσιαστικά και η υλοποίηση της wait_for_ready_children() συνάρτησης και έτσι η παρουσία αυτής, ή μίας ισοδύναμης επαναληπτικής υλοποίησης με waitpid() και WUNTRACED, κρίνεται απαραίτητη για την ορθή λειτουργία του προγράμματος.

Σε περίπτωση που δεν χρησιμοποιηθεί αυτή η συνάρτηση στον κώδικα μας, οι διεργασίες γονείς δεν θα μπορέσουν να αντιληφθούν πότε τα παιδιά τους έχουν σταματήσει μέσω του σήματος SIGSTOP και θα προχωρήσουν στην εκτέλεση του υπόλοιπου κώδικα, κάτι που σημαίνει πως η show_pstree() πιθανόν να κληθεί πρόωρα, χωρίς να έχουν προλάβει όλες οι διεργασίες να δημιουργηθούν και να σταματήσουν, ώστε να ληφθεί ένα παγωμένο δέντρο διεργασιών που να τις περιέχει όλες. Επιπλέον, δεν θα εκτυπωθεί κανένα διαγνωστικό μήνυμα “stopped by a signal” στο τερματικό, καθώς αυτά τα μηνύματα τα διαχειριζόταν μέρος της wait_for_ready_children() που καλούσε την explain_wait_status(). Το υπόλοιπο πρόγραμμα ωστόσο, στην περίπτωση μας, παραμένει λειτουργικό όπως αναμένεται και τα μηνύματα ενεργοποίησης εκτυπώνονται κανονικά σε κατά βάθος σειρά και τα υπόλοιπα μηνύματα, με εξαίρεση φυσικά αυτά που αναφέραμε προηγουμένως, εκτυπώνονται επίσης.

1.4

Πηγαίος κώδικας:

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

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

#define SLEEP_PROC_SEC  5
#define SLEEP_TREE_SEC  3

void fork_procs(struct tree_node *root, int father[]) {

    change_pname(root->name);
    printf("Process (name: %s) with PID: %d is created.\n", root-
>name, getpid());

    int i, dad, number, temp;
    pid_t pid;

    /* Father processes with children */
    if(root->children) {

        int numbers[2];

        /* Create pipe between this process and its children */
        int child[2];
        if (pipe(child) < 0) {
            perror("pipe\n");
            exit(1);
        }

        /* Create children processes */
        for(i = 0; i < 2; i++){
            pid = fork();
```

```
            if(pid < 0) {
                perror("fork_procs: fork\n");
                exit(1);
            }
            else if(pid == 0){
                /* Child calls the function recursively with the file
 descriptors
                for child to parent communication */
                fork_procs(root->children + i, child);
            }
        }

        /* Father */
        for(int i = 0; i < 2; i++) {
            /* Awaits and eventually reads child's value from their p
ipe */
            if (read(child[0], &temp, sizeof(temp)) != sizeof(temp))
{
                perror("read from pipe\n");
                exit(1);
            }

            /* Stores the values of each child into an array for late
r calculations */
            numbers[i] = temp;

            /* If it has read both children's value it can close its
read end */
            if(i == 1) {
                close(child[0]);
            }
        }

        /* Check if we should add or multiply children's values and c
alculate
        value to be written on father's pipe*/
        if (!strcmp(root->name, "+")) {
            dad = numbers[0] + numbers[1];
        }
        else if(!strcmp(root->name, "*")) {
            dad = numbers[0] * numbers[1];
        }
```

```c
        /* Write value on father's pipe */
        if (write(father[1], &dad, sizeof(dad)) != sizeof(dad)) {
            perror("write from pipe\n");
            exit(1);
        }
        /* Close the write end, since the value has been written */
        close(father[1]);
        /* Then exit */
        exit(0);
    }

    else {
        /* Close father's pipe read end, since a leaf doesn't use it
*/
        close(father[0]);

        /* Convert char * to integer and write on father's pipe*/
        number = atoi(root->name);
        if (write(father[1], &number, sizeof(number)) != sizeof(numbe
r)) {
            perror("write from pipe\n");
            exit(1);
        }
        /* Close father's pipe write end, since the value has been wr
itten */
        close(father[1]);
        /* Then exit */
        exit(0);
    }
}

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

    pid_t p;
    int answer;

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

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

    /* Create pipe between ask2-fork(main) and root process */
    int child[2];
    if (pipe(child) < 0) {
        perror("pipe\n");
        exit(1);
    }

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

    if(p < 0) {
        perror("main: fork\n");
        exit(1);
    }
    if(p == 0) {
        /* Child */
        fork_procs(root, child);
        exit(0);
    }

    /* Father */
    /* Close write end, since it only reads the final answer */
    close(child[1]);

    /* Await and eventually read the final answer from the pipe and p
rint it */
    if (read(child[0], &answer, sizeof(answer)) != sizeof(answer)) {
        perror("read from pipe\n");
        exit(1);
    }
    printf("The final value of the expression tree is: %d.\n", answer
);

    return 0;
}
```

Έξοδος εκτέλεσης προγράμματος:

$ ./ask2-fork expr.tree

Process (name: +) with PID: 1931 is created.
Process (name: 10) with PID: 1932 is created.
Process (name: *) with PID: 1933 is created.
Process (name: +) with PID: 1934 is created.
Process (name: 4) with PID: 1935 is created.
Process (name: 5) with PID: 1936 is created.
Process (name: 7) with PID: 1937 is created.
The final value of the expression tree is: 58.

Ερωτήσεις:

1. Σε αυτήν την υλοποίηση μας χρησιμοποιήσαμε 1 pipe για κάθε διεργασία και για τα 2 παιδιά της με έναν πίνακα από περιγραφητές αρχείων τον οποίο περνάμε στις διεργασίες παιδιά, ώστε να μπορούν να έχουν πρόσβαση στο pipe της επικοινωνίας παιδί προς πατέρα που λαμβάνουν ως copy μέσω της fork, όσο και στο δικό τους pipe που θα δημιουργήσουν για επικοινωνία από τα παιδιά τους. Η επιλογή αυτή δουλεύει στην προκειμένη περίπτωση, επειδή δεν μας ενδιαφέρει η σειρά με την οποία τα παιδιά γράφουν και ο πατέρας διαβάζει, καθώς δεν αλλάζει το αποτέλεσμα του υπολογισμού με όποια σειρά και να γίνουν αυτά. Σε διαφορετική περίπτωση, όπως αυτή του να είχαμε τις πράξεις αφαίρεση ή διαίρεση, αυτή η υλοποίηση δεν θα δούλευε, καθώς σε αυτές τις πράξεις δεν ισχύει η αντιμεταθετική ιδιότητα. Θα χρειαζόμασταν, επομένως, μία υλοποίηση με δύο ξεχωριστά pipes, ένα για κάθε παιδί, ώστε όταν ο πατέρας διαβάζει να γνωρίζει τη σωστή σειρά με την οποία πρέπει να εκτελέσει τις πράξεις.

2. Εφόσον έχουμε στη διάθεση μας πολλαπλούς επεξεργαστές, μπορούμε να χρησιμοποιήσουμε πάνω από έναν ανά οποιαδήποτε στιγμή, ώστε να υπολογίσουμε παράλληλα τις ενδιάμεσες τιμές του δέντρου, όπως και την τελική. Με αυτόν τον τρόπο, έχουμε μειώσει αισθητά τον χρόνο που χρειάζεται η αποτίμηση μίας έκφρασης, σε σχέση με το χρόνο που θα απαιτούσε η αποτίμηση της σε ένα σύστημα με έναν επεξεργαστή και σειριακή εκτέλεση των υπολογισμών, καθώς στην περίπτωση μας εκτελούνται περισσότερες πράξεις ανά χρονική μονάδα.