ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Λειτουργικά Συστήματα

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2020-2021

Ονοματεπώνυμο	Αριθμός Μητρώου
Γουλόπουλος Δημήτριος	03107627
Σιαφάκας Ξενοφών	03115753

oslaba116

Άσκηση 2: Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

Άσκηση 2.1 Δημιουργία δεδομένου δέντρου διεργασιών

```
#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

void fork_procs(void)
{
    change_pname("A");
    printf("A with PID = %ld is created...\n", (long) getpid());
    pid_t pid_b, pid_c, pid_d;
    int status;
    printf("A with PID = %ld: Creating child B...\n", (long) getpid());
    pid_b = fork();
    if (pid_b < 0) { perror("B: fork"); exit(1); }

        if (pid_b == 0)
        {
            change_pname("B");
            printf("B with PID = %ld is created...\n", (long) getpid());
            printf("B with PID = %ld: Creating child D...\n", (long) getpid());
            pid_d = fork();
            if (pid_d < 0) { perror("D: fork"); exit(1); }

                if (pid_d == 0)
                {
                    change_pname("D");
                    printf("D with PID = %ld is created...\n", (long) getpid());
                    printf("D with PID = %ld is ready to Sleep...\n", (long) getpid());
                    sleep(SLEEP_PROC_SEC);
                    printf("D with PID = %ld is ready to terminate...\nD: Exiting...\n", (long) getpid());
                    exit(13);
                }

            waitpid(pid_d, &status, 0);
            explain_wait_status(pid_d, status);
            printf("B with PID = %ld is ready to terminate...\nB: Exiting...\n", (long) getpid());
            exit(19);
        }
```

```
    printf("A with PID = %ld: Creating child C...\n", (long) getpid());
    pid_c = fork();
    if (pid_c < 0) { perror("C: fork"); exit(1); }

        // PROCESS C
        if (pid_c == 0)  // an eimai to paidi
        {
            change_pname("C");
            printf("C with PID = %ld is created...\n", (long) getpid());
            printf("C with PID = %ld is ready to sleep...\n", (long) getpid());
            sleep(SLEEP_PROC_SEC);
            printf("C with PID = %ld is ready to terminate...\nC: Exiting...\n", (long) getpid());
            exit(17);
        }

    waitpid(pid_c, &status, 0);
    explain_wait_status(pid_c, status);
    waitpid(pid_b, &status, 0);
    explain_wait_status(pid_b, status);
    printf("A with PID = %ld is ready to terminate...\nA: Exiting...\n", (long) getpid());
    exit(16);
}

int main(void)
{
    pid_t pid;
    int status;
    pid = fork();
    if (pid < 0) { perror("main: fork"); exit(1); }
    if (pid == 0)
    {
         fork_procs();
         exit(1);
    }
    show_pstree(pid);
    waitpid(pid, &status, 0);
    explain_wait_status(pid, status);
    return 0;
}
```

Ερωτήσεις:

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

Η διεργασία Α, που είναι πατέρας των Β, C θα λαβει το σήμα SIGKILL.
Οι διεργασίες Β, C δεν πεθαίνουν αλλά αποκαλούνται "zombie" ή ορφανές μιας και ο πατέρας τους έχει πεθάνει χωρίς να περιμένει για τα παιδιά του να τελειώσουν και να καθαρίσει το process table με κλήση συστήματος wait(). Σε συστήματα UNIX, οι διεργασίες zombie, υιοθετούνται απο την διεργασία init (pid = 1) η οποία εκτελεί περιοδικά wait() και αναλαμβάνει τον τερματισμό των ορφανών διεργασιών. Η init είναι η πρώτη διεργασία κατά την εκκίνηση του υπολογιστικού συστήματος που εκτελείται στο υπόβαθρο μέχρι τον τερματισμό του.

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

Από το manual της getpid() γνωρίζουμε πως η συγκεκριμένη κλήση συστήματος επιστρέφει το processID της καλούμενης διεργασίας.

Βλέποντας πως στην main γίνεται fork(), δημιουργώντας ένα παιδί που στην συνέχεια του δίνεται το όνομα Α.

Παρόλα αυτά η διεργασία που εκτελεί το κομμάτι του κώδικα που καλείται η show_pstree είναι η ask4-fork1.

Επομένως, το δέντρο διεργασιών που θα δούμε είναι το παρακάτω:

```
ask2-fork1(18207)─┬─A(18208)─┬─B(18210)───D(18212)
|                 └─C(18211)
└─sh(18209)───pstree(18213)
```

Εκτός από την ask2-fork1, που ήταν αναμενόμενη, βλέπω δύο ακόμα διεργασίες: την sh και τηνpstree που είναι παιδί της.

Επειδή η συνάρτηση show_pstree κάνει χρήση της συνάρτησηςsystemκαλώντας την pstree. Επομένως η ask2-fork1 καλεί και την διεργασία του bash η οποίαμε την σειρά της καλεί την pstree.

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

Ο εκάστοτε διαχειριστής καποιου συστήματος, πρέπει να εξασφαλίσει ότι κανένας χρήστης δε θα μπορεί να δημιουργήσει αυθαίρετο αριθμό απο διεργασίες – αφήνοντας ανοικτή την περίπτωση κάποιος χρήστης εκούσια ή ακούσια να κάνει κατάχρηση των διαθέσιμων system resources με αποτέλεσμα το σύστημα να μην είναι stable. Ένα χαρακτηριστικό παράδειγμα αποτελεί η επίθεση forkbomb (ή RabbitVirus ή Wabbit) η οποία δημιουργεί συνεχώς αντίγραφα του εαυτού της με σκοπό να κάνει ενα σύστημα να κρασάρει (DoS : Denial of Service attack).

Συγκεκριμένα, γράφοντας:

```
while(1)  {fork();}
```

δημιουργούμε συνεχώς νέες διεργασίες.

Άσκηση 2.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "tree.h"
#include "proc-common.h"
#define SLEEP_TREE_SEC 3
#define SLEEP_PROC_SEC 10

void fork_procs(struct tree_node *t)
{
    int i;
    printf("PID = %ld, name %s, starting...\n", (long) getpid(), t->name);
    change_pname(t->name);
    if (t->nr_children == 0)
    {
      sleep(SLEEP_PROC_SEC);
        printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n",
           t->name, (long) getpid(), t->name);
        exit(0);
    }
    else
    {
        int status;
        pid_t pid[t->nr_children];
        for (i = 0; i < t->nr_children; i++)
        {
            pid[i] = fork();
            if (pid[i] < 0) { perror("proc: fork"); exit(1); }
            if (pid[i] == 0)
            {
                fork_procs(t->children + i);
                exit(1);
            }
        }
        for (i = 0; i < t->nr_children; i++)
        {
            waitpid(pid[i], &status, 0);
            explain_wait_status(pid[i], status);
        }
    }
    printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n",
             t->name, (long) getpid(), t->name);
    exit(0);
}
```

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

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

    root = get_tree_from_file(argv[1]);

    pid = fork();
    if (pid < 0) { perror("main: fork"); exit(1); }

    if (pid == 0)
    {
        fork_procs(root);
        exit(1);
    }

    sleep(SLEEP_TREE_SEC);

    show_pstree(pid);
    waitpid(pid, &status, 0);
    explain_wait_status(pid, status);
    return 0;
}
```

Ερωτήσεις:

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

Το πρόγραμμα αρχικά δημιουργεί τους κόμβους αρχίζοντας από τους γονείς και εκ τως υστέρων δημιουργεί και τα παιδιά-φύλλα, επομένως η δημιουργία γίνεται με top-down λογική.

Αντίθετα, ο τερματισμός γίνεται με bottom-up λογική, καθώς θέλουμε να τερματιστούν πρώτα τα φύλλα και μετά οι εσωτερικοί κόμβοι, διότι θέλουμε να πεθάνουν ομαλά τα παιδιά και να μην μείνουν ορφανά.

```
#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 *t)
{
    int i;
    printf("PID = %ld, name %s, starting...\n", (long) getpid(), t->name);
    change_pname(t->name);
    if (t->nr_children == 0)
    {
        raise(SIGSTOP);
        printf("PID = %ld, name = %s is awake\n", (long) getpid(), t->name);
    }
    else
    {
        int status[t->nr_children];
        pid_t pid[t->nr_children];
        for (i = 0; i < t->nr_children; i++)
        {
            pid[i] = fork();
            if (pid[i] < 0) { perror("proc: fork"); exit(1); }
            if (pid[i] == 0)
            {
                fork_procs(t->children + i);
                exit(1);
            }
            wait_for_ready_children(1);
        }
        raise(SIGSTOP);
        printf("PID = %ld, name = %s is awake\n", (long) getpid(), t->name);
        for (i = 0; i < t->nr_children; i++)
        {
            kill(pid[i], SIGCONT);   // SIGnal CONTinue
            waitpid(pid[i], &status[i], 0);
            explain_wait_status(pid[i], status[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<tree_file>\n", argv[0]);
        exit(1);
    }
    root = get_tree_from_file(argv[1]);
    pid = fork();
    if (pid < 0) { perror("main: fork"); exit(1); }
    if (pid == 0)       { fork_procs(root); exit(1); }
    wait_for_ready_children(1);
    show_pstree(pid);
    kill(pid, SIGCONT);
    wait(&status);
    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;

Στις προηγούμενες ασκήσεις αδρανοποιούσαμε τα παιδιά-φύλλα ώστε να εξασφαλίσουμε χρόνο πριν "πεθάνουν" ώστε να προλάβουν να αποτυπωθούν από την pstree που καλείται στηνσυνάρτηση show_pstree().Με τη χρήση σημάτων, έχουμε περισσότερο έλεγχο στο συγχρονισμο των διεργασιών και το πρόγραμμά μας τρέχει χωρίς περιττούς χρόνους αναμονής. Με την χρήση σημάτων αποφεύγουμε να ρυθμίζουμε αυθαίρετα τον χρόνο όπως κάναμε στην 1.2 με την sleep(), πετυχαίνοντας καλύτερο συγχρονισμό. Συγκεκριμένα μόλις "κοιμηθούν" με την raise(SIGSTOP) και αφού εμφανιστεί το δέντρο στέλνεται μήνυμα στα παιδιά με το kill(pid, SIGCONT) να ξυπνήσουν και να συνεχίσουν εώς ότου τερματιστούν.

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

Η wait_for_ready_children() είναι μια συνάρτηση που αναστέλλει την λειτουργία της διεργασίας μέχρι τα παιδιά της να αδρανοποιηθούν. Ελέγχει ουσιαστικά τη σημαία WIFSTOPPED. Η συνάρτηση αυτή, εκτελεί την waitpid() για οποιοδήποτε αριθμό διεργασίας (pid = -1) τόσες φορές όσες δηλωνει η παράμετρος cnt (ο αριθμός των παιδιών). Η χρήση της εξασφαλίζει πως όλα τα παιδιά είναι ζωντανά και έχουν σταματήσει, άρα το δέντρο διεργασιών θα εμφανιστεί σωστά. Δηλαδή, ελέγχει αν όλα τα παιδιά της εκάστοτε διεργασίας που κάλεσε την wait_for_ready_children() έχουν γίνει STOPPED απο κάποιο signal, και δεν έχουν σταματήσει την εκτελεσή τους για κάποιο άλλο (unexpected) λογο.

Άσκηση 2.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include "tree.h"
#include "proc-common.h"
#define SLEEP_TREE_SEC 3
#define SLEEP_PROC_SEC 10

void fork_procs(struct tree_node *t, int parentPipe[])
{
    int i;
    printf("PID = %ld, name %s, starting...\n", (long) getpid(), t->name);
    change_pname(t->name);
    if (t->nr_children == 0)
    {
        int value;
        sleep(SLEEP_PROC_SEC);
        if (close(parentPipe[0]) < 0)
            { perror("Close"); exit(1); }
        value = atoi(t->name);
        if (write(parentPipe[1], &value, sizeof(int)) < 0) { perror("Write"); exit(1); }
        printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n",
                t->name, (long) getpid(), t->name);
        exit(0);
    }
    else
    {
        int numbers[2];
        int myPipe[2];
        if (pipe(myPipe)) { perror("Failed pipe"); }
        int status;
        pid_t pid;
        for (i = 0; i < 2; i++)
        {
            pid = fork();
            if (pid < 0) { perror("proc: fork"); exit(1); }
            if (pid == 0)
            {
                fork_procs(t->children + i, myPipe);
                exit(1);
            }
        }
```

```
        if (close(myPipe[1]) < 0)
             { perror("Close"); exit(1); }
        for (i = 0; i < 2; i++)
        {
            if (read(myPipe[0], numbers + i, sizeof(int)) < 0)
                { perror("Read"); exit(1); }
        }
        int result;
        if (!strcmp(t->name, "+")) result = numbers[0] + numbers[1];
        else                       result = numbers[0] * numbers[1];
        if (write(parentPipe[1], &result, sizeof(int)) < 0)
        {
            perror("Write");
            exit(1);
        }
        for (i = 0; i < 2; i++)
        {
            waitpid(pid, &status, 0);
            explain_wait_status(pid, status);
        }
    }
    printf("%s with PID = %ld is ready to terminate...\n%s: Exiting...\n",
             t->name, (long) getpid(), t->name);
    exit(0);
}

int main(int argc, char *argv[])
{
    pid_t pid;
    int status, pipe_fd[2];
    struct tree_node * root;
    if (argc < 2)  { fprintf(stderr, "Usage: %s<tree_file>\n", argv[0]); exit(1); }
    root = get_tree_from_file(argv[1]);
    if (pipe(pipe_fd))     { perror("Failed pipe"); }
    pid = fork();
    if (pid < 0) { perror("main: fork"); exit(1); }
    if (pid == 0)
    {
        fork_procs(root, pipe_fd);
        exit(1);
    }
    sleep(SLEEP_TREE_SEC);
    show_pstree(pid);
    int result;
    if (read(pipe_fd[0], &result, sizeof(int)) < 0)   { perror("Read"); exit(1); }
    wait(&status);
    explain_wait_status(pid, status);
    printf("\nThe result is %d\n", result);
    return 0;
}
```

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 γίνεται η χρήση μίας σωλήνωσης. Όπως φαίνεται και στον κώδικα, έχουμε κάνει υλοποίηση με ένα pipe ανά διεργασία. Αυτό γενικά είναι δυνατό γιατί τα writes μέσα στο pipe γίνονται ατομικά,το οποίο σημαίνει ότι αν δύο η περισσότερα παιδιά προσπαθούν να γράψουν μέσα στο ίδιο pipe δεν θα υπάρχει περίπτωση να υπάρξει πρόβλημα με μπερδεμένα δεδομένα (εναλλάξ δεδομένα από τα διαφορετικά παιδιά). Το δέντρο διασχίζεται κατά βάθος και φτάνοντας στα φύλλα με το μεγαλύτερο βάθος αρχίζουν να γίνονται οι πράξεις της πρόσθεσης και του πολλαπλασιασμού που είναι αντιμεταθετικές πράξεις. Στην περίπτωση που γίνονταν πράξεις όπως αφαίρεση και διαίρεση, που δεν ισχύει η αντιμεταθετική ιδιότητα, θα χρειάζονταν τουλάχιστον δύο pipes.

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;

Γενικά σε ένα τέτοιο σύστημα μπορούν να τρέξουν ταυτόχρονα περισσότερες διεργασίες αν δεν έχουν μεταξύ τους εξάρτηση. Στο συγκεκριμένο πρόγραμμα, αυτές είναι οι διεργασίες παιδιά κάθε γονιού. Αυτό σημαίνει ότι κάνοντας τις απαραίτητες τροποποιήσεις στο πρόγραμμα (δέντρο διεργασιών και επίτρεψη ταυτόχρονου τρεξίματος των παιδιών) μπορούμε στον ίδιο χρόνο να καάνουμε περισσότερες πράξεις και τελικά να έχουμε πιο γρήγορο υπολογισμό. Αντίθετα, αν δεν έχουμε ένα τέτοιο σύστημα ή μια κατάλληλη οργάνωση του προγράμματος σε δέντρο διεργασιών, αναγκαζόμαστε να ακολουθούμε το ρυθμό ενός επεξεργαστή και της σειράς του κώδικα.