ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Λειτουργικά Συστήματα Υπολογιστών

2η εργαστηριακή αναφορά:
Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

Διδάσκοντες:

Ν. Κοζύρης
Γ. Γκούμας

oslaba84:

Ειρήνη Δόντη
ΑΜ 03119839

6ο εξάμηνο

Αθήνα 2022

Περιεχόμενα:

1.1 Δημιουργία δεδομένου δέντρου διεργασιών

Παρακάτω, απεικονίζεται ο κώδικας που υλοποιήσαμε για αυτό το ερώτημα:

```c
#include <unistd.h> /* As ask2-fork.c */
#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

void fork_procs(void)
{
/* initial process is A.*/
        pid_t  pid_B, pid_C, pid_D;
        int status;

        change_pname("A"); /* we create node A */
        printf("Now we have node A...\n");

/* A has 2 children, B and C */

/* We create node B */

        pid_B = fork();
        if(pid_B < 0){
                perror("We have a problem with node B...");
                exit(1);
        }
        else if(pid_B == 0){ /* if node B has at least one child */
                change_pname("B"); /* we create node B */
                printf("Now we have node B...\n");

/* We create node D */

                pid_D = fork();
                if(pid_D < 0){
                        perror("We have a problem with node D...");
                        exit(1);
                }
                else if(pid_D == 0){
                        change_pname("D"); /* we create node D */
                        printf("Now we have node D...\n");
                        printf("Node D is sleeping...\n");
                        sleep(SLEEP_PROC_SEC); /* node D has no chidren, so D s
leeps */
                        printf("Node D exits\n");
                        exit(13);
                }
                else if(pid_D >0){// Check status for the leaf D
                  printf("Node B is waiting for status of node D\n");
                  pid_D = wait(&status);
                  explain_wait_status(pid_D,status);
                  printf("Node B exits\n");
                  exit(19);
```

```
                }
        }
        else if(pid_B>0){
                /* We create node C */
                pid_C = fork();
                if(pid_C < 0){
                        perror("We have a problem with node C...");
                        exit(1);
                }
                else if(pid_C == 0){
                        change_pname("C"); /* we create node C */
                        printf("Now we have node C...\n");
                        printf("Node C is sleeping...\n");
                        sleep(SLEEP_PROC_SEC); /* node C has no chidren, so C s
leeps */
                        printf("Node C exits\n");
                        exit(17);
                }
                else if(pid_C>0){//Node C is the last leaf of the tree
                       printf("Node A is waiting for status of node C\n");
                       pid_C = waitpid(-1,&status,0);
                       pid_C = waitpid(-1,&status,0); // A must exit at the end
, after B. So A needs to wait for double time
                       explain_wait_status(pid_C,status);
                       printf("Node A exits\n");
                       exit(16);
                }
        }
```

```
}

int main (void){
        pid_t p;
        int status;
        p=fork();
        if (p < 0) {
                perror("We have a problem with pid...");
                exit(1);
        }
        if (p == 0){
                fork_procs(); /* Create the tree process */
                exit(1);
        }
        sleep(SLEEP_TREE_SEC);
        show_pstree(p); /* Print process tree rooted at process with pid p */

        /* The fork() process needs Wait() process */
        p = wait(&status);
        explain_wait_status(p,status);
        return 0;
}
```

Τρέχουμε τον κώδικα του υποερωτήματος 1.1 και λαμβάνουμε στην έξοδο το παρακάτω αποτέλεσμα:

```
Now we have node A...
Now we have node B...
Node A is waiting for status of node C
Now we have node C...
Node C is sleeping...
Node B is waiting for status of node D
Now we have node D...
Node D is sleeping...


A(29957)─┬─B(29958)──D(29960)
         └─C(29959)


Node C exits
Node D exits
My PID = 29958: Child PID = 29960 terminated normally, exit status = 13
Node B exits
My PID = 29957: Child PID = 29958 terminated normally, exit status = 19
Node A exits
My PID = 29956: Child PID = 29957 terminated normally, exit status = 16
```

Ερωτήσεις:

1. Στην περίπτωση που τερματίσουμε πρόωρα τη διεργασία Α, δίνοντας **kill -KILL <pid>**, όπου <pid> το Process ID της, ο κόμβος Α καταστρέφεται χωρίς να περιμένει να τερματιστούν οι διεργασίες - παιδιά του. Έτσι, τα παιδιά του μένουν ορφανά και πλέον υιοθετούνται από την init.

2. Αλλάζουμε την εντολή show_pstree(pid) σε show_pstree(getpid()). Παρακάτω, απεικονίζεται, σε στιγμιότυπο οθόνης, το αποτέλεσμα εκτέλεσης του προγράμματος:

```
Now we have node A...
Now we have node B...
Node A is waiting for status of node C
Now we have node C...
Node C is sleeping...
Node B is waiting for status of node D
Now we have node D...
Node D is sleeping...


ex2_1_1(31552)─┬─A(31553)─┬─B(31554)──D(31556)
               │          └─C(31555)
               └─sh(31557)──pstree(31558)


Node C exits
Node D exits
My PID = 31554: Child PID = 31556 terminated normally, exit status = 13
Node B exits
My PID = 31553: Child PID = 31554 terminated normally, exit status = 19
Node A exits
My PID = 31552: Child PID = 31553 terminated normally, exit status = 16
```

Παρατηρούμε ότι πλέον έχουμε στην έξοδο όλο το δέντρο διεργασιών. Η ρίζα του δέντρου πλέον είναι η ex2_1_1 και επιπλέον περιλαμβάνεται η διεργασία sh με παιδί το pstree. Οι δύο αυτές διεργασίες καλούνται από την show_pstree.

3. Ο διαχειριστής θέτει όρια στον αριθμό των διεργασιών που μπορεί να δημιουργήσει ένας χρήστης, καθώς ένας χρήστης μπορεί να εισαγάγει στο σύστημα υπερβολικά πολλές διεργασίες με μία ατέρμονη επανάληψη που περιέχει την κλήση συστήματος fork(). Αυτό σημαίνει ότι θα δημιουργηθούν πάρα πολλές διεργασίες παιδιά, τα οποία μπορεί να μην μπορέσει να τα διαχειριστεί σωστά το πρόγραμμα.

1.2 Δημιουργία αυθαίρετου δέντρου διεργασιών

Παρακάτω, απεικονίζεται ο κώδικας που υλοποιήσαμε για αυτό το ερώτημα:

```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(struct tree_node *root)
{

        change_pname(root->name); // create the root of the tree


        // if the node has no children
        if(root->nr_children == 0){
                printf("Now we have leaf %s. It is sleeping\n",root->name); //
print the name of the leaf
                sleep(SLEEP_PROC_SEC); // now the node is sleeping
```

```c
int main(int argc, char *argv[]) //tree-example.c
{
        struct tree_node *root;
        int status;
        pid_t p;
        if (argc != 2) {
                fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
                exit(1);
        }

        root = get_tree_from_file(argv[1]); // insert filename
        print_tree(root);
        p = fork();
        if(p<0){
                perror("Problem with fork\n");
                exit(1);
        }
        else if(p == 0){
                fork_procs(root);
        }
        sleep(SLEEP_TREE_SEC);
        show_pstree(p);
        p=wait(&status);
        explain_wait_status(p,status);
        printf("Exit\n");
        return 0;
}
```

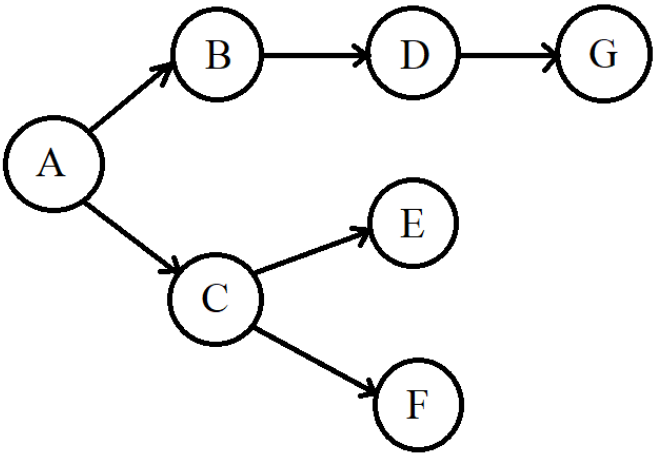
Τρέχουμε τον κώδικα του υποερωτήματος 1.2, για το δοθέν δέντρο που δίνεται στο παράδειγμα και λαμβάνουμε στην έξοδο το παρακάτω αποτέλεσμα:

```
A
        B
                D
        C
Node A has at least one child
Node B has at least one child
Node A is  waiting for all children to exit
Now we have leaf C. It is sleeping
Node B is  waiting for all children to exit
Now we have leaf D. It is sleeping


A(30608)─┬─B(30609)──D(30611)
         └─C(30610)


Exit C leaf
Exit D leaf
My PID = 30608: Child PID = 30610 terminated normally, exit status = 10
My PID = 30609: Child PID = 30611 terminated normally, exit status = 10
Exit node B
My PID = 30608: Child PID = 30609 terminated normally, exit status = 4
Exit node A
My PID = 30607: Child PID = 30608 terminated normally, exit status = 4
Exit
```

Δημιουργούμε ένα δικό μας text file, στο οποίο περιγράφουμε το παρακάτω δέντρο:



Τρέχουμε τον κώδικα για το δικό μας παράδειγμα και λαμβάνουμε τα εξής:

```
A
        B
                D
                        G
        C
                E
                F
Node A has at least one child
Node B has at least one child
Node A is  waiting for all children to exit
Node C has at least one child
Node B is  waiting for all children to exit
Node D has at least one child
Now we have leaf E. It is sleeping
Node C is  waiting for all children to exit
Now we have leaf F. It is sleeping
Node D is  waiting for all children to exit
Now we have leaf G. It is sleeping


A(32276)─┬─B(32277)──D(32279)──G(32282)
         └─C(32278)─┬─E(32280)
                    └─F(32281)
```

```
Exit E leaf
Exit F leaf
My PID = 32278: Child PID = 32280 terminated normally, exit status = 10
Exit G leaf
My PID = 32278: Child PID = 32281 terminated normally, exit status = 10
Exit node C
My PID = 32279: Child PID = 32282 terminated normally, exit status = 10
Exit node D
My PID = 32276: Child PID = 32278 terminated normally, exit status = 4
My PID = 32277: Child PID = 32279 terminated normally, exit status = 4
Exit node B
My PID = 32276: Child PID = 32277 terminated normally, exit status = 4
Exit node A
My PID = 32275: Child PID = 32276 terminated normally, exit status = 4
Exit
```

Ερωτήσεις:

1. Κάθε διεργασία δημιουργεί τις διεργασίες-παιδιά του και περιμένει έως ότου τερματίσουν. Οι διεργασίες-παιδιά επαναλαμβάνουν την ίδια διαδικασία μέχρι τα φύλλα του δέντρου να “κοιμηθούν” και συνεπώς να κατασκευαστεί το υπόλοιπο δέντρο. Αντίθετα, για τον τερματισμό, ακολουθείται η αντίστροφη διαδικασία. Πρώτα τερματίζουν όλα τα παιδιά και στο τέλος τερματίζει ο εκάστοτε “πατέρας” (και στο τέλος η ρίζα του δέντρου). Με άλλα λόγια, τα μηνύματα έναρξης των διεργασιών εμφανίζονται με BFS και του τερματισμού με τον αντίστροφο τρόπο.

1.3 Αποστολή και χειρισμός σημάτων

Παρακάτω, απεικονίζεται ο κώδικας που υλοποιήσαμε για αυτό το ερώτημα:

```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
 */
/*
 * Start
 */
void fork_procs(struct tree_node *root){ // ask2-signals.c
        int status, c; // c is for children
        pid_t p[c]; int i;
        printf("PID = %ld, name %s, starting...\n",
                        (long)getpid(), root->name);
        change_pname(root->name);

        /* ... */
```

```c
      if(root->nr_children == 0){
              /* ... */
              raise(SIGSTOP);
              printf("PID = %ld, name = %s is awake\n",
                              (long)getpid(), root->name);
              exit(0);
      }
      else { //Create tree
              c=root->nr_children;
              for(i=0; i<c; i++){
                      p[i]=fork();
                      if(p[i] == 0){
                              fork_procs(root->children +i);
                              exit(0);
                      }
                      else if(p[i] < 0){
                              perror("Error in fork process");
                              exit(1);
                      }
              }
              wait_for_ready_children(c);
              raise(SIGSTOP); //stop process
              printf("PID = %ld, name = %s is awake\n",
                              (long)getpid(), root->name);
              for(i=0; i<c; i++){
                      kill(p[i],SIGCONT);
              }
              p[i]=waitpid(p[i],&status,0);
```

```
                explain_wait_status(p[i],status);
        }
}

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

        struct sigaction sa;

        if(sigaction(SIGCHLD,&sa,NULL) < 0){
          perror("sigaction");
          exit(1);
        }

        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);

        /* 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 */
        wait(&status);
        pid=waitpid(pid,&status,0);
        explain_wait_status(pid, status);

        return 0;
}
```

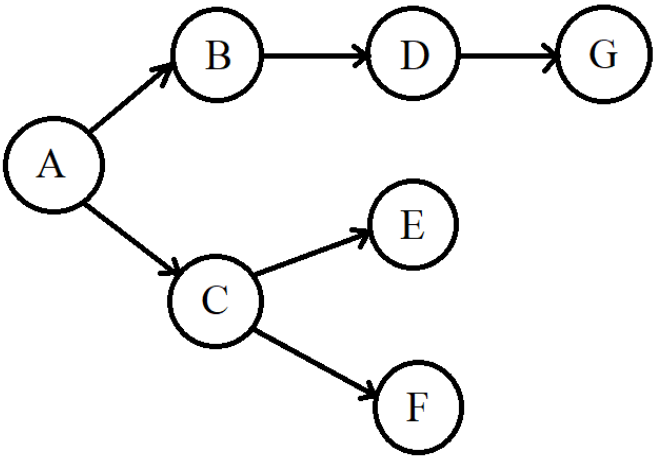
Τρέχουμε τον κώδικα του υποερωτήματος 1.3, για το δοθέν δέντρο που δίνεται στο παράδειγμα και λαμβάνουμε στην έξοδο το παρακάτω αποτέλεσμα:

```
PID = 30889, name A, starting...
PID = 30890, name B, starting...
PID = 30891, name C, starting...
My PID = 30889: Child PID = 30891 has been stopped by a signal, signo = 19
PID = 30892, name D, starting...
My PID = 30890: Child PID = 30892 has been stopped by a signal, signo = 19
My PID = 30889: Child PID = 30890 has been stopped by a signal, signo = 19
My PID = 30888: Child PID = 30889 has been stopped by a signal, signo = 19


A(30889)─┬─B(30890)──D(30892)
         └─C(30891)


PID = 30889, name = A is awake
My PID = 30889: Child PID = -1 terminated normally, exit status = 0
PID = 30890, name = B is awake
PID = 30891, name = C is awake
My PID = 30890: Child PID = -1 terminated normally, exit status = 0
PID = 30892, name = D is awake
My PID = 30888: Child PID = -1 terminated normally, exit status = 1
```

Δημιουργούμε ένα δικό μας text file, στο οποίο περιγράφουμε το παρακάτω δέντρο:



Τρέχουμε τον κώδικα για το δικό μας παράδειγμα και λαμβάνουμε τα εξής:

```
PID = 32349, name A, starting...
PID = 32350, name B, starting...
PID = 32351, name C, starting...
PID = 32352, name D, starting...
PID = 32353, name E, starting...
PID = 32354, name F, starting...
My PID = 32351: Child PID = 32353 has been stopped by a signal, signo = 19
PID = 32355, name G, starting...
My PID = 32351: Child PID = 32354 has been stopped by a signal, signo = 19
My PID = 32349: Child PID = 32351 has been stopped by a signal, signo = 19
My PID = 32352: Child PID = 32355 has been stopped by a signal, signo = 19
My PID = 32350: Child PID = 32352 has been stopped by a signal, signo = 19
My PID = 32349: Child PID = 32350 has been stopped by a signal, signo = 19
My PID = 32348: Child PID = 32349 has been stopped by a signal, signo = 19


A(32349)─┬─B(32350)──D(32352)──G(32355)
         └─C(32351)─┬─E(32353)
                    └─F(32354)
```

```
PID = 32349, name = A is awake
My PID = 32349: Child PID = -1 terminated normally, exit status = 0
PID = 32350, name = B is awake
PID = 32351, name = C is awake
My PID = 32350: Child PID = -1 terminated normally, exit status = 0
My PID = 32351: Child PID = -1 terminated normally, exit status = 0
PID = 32354, name = F is awake
PID = 32353, name = E is awake
PID = 32352, name = D is awake
PID = 32355, name = G is awake
My PID = 32348: Child PID = -1 terminated normally, exit status = 1
My PID = 32352: Child PID = 32355 terminated normally, exit status = 0
```

Ερωτήσεις:

1. Στην συγκεκριμένη άσκηση χρησιμοποιούμε σήματα αντί για τη sleep(), για τον συγχρονισμό των διεργασιών. Αυτό έχει ως αποτέλεσμα να μειώνεται ο χρόνος που απαιτείται για την εκτέλεση του προγράμματος, αφού πλέον η sleep() δεν καλείται από καμία διεργασία. Εκτός αυτού, η συνάρτηση show_pstree(pid) έχει πλέον λιγότερες πιθανότητες να αστοχήσει.
2. Η wait_for_ready_children() εξασφαλίζει τον συγχρονισμό των διεργασιών, καθώς καλεί την κλήση συστήματος wait() περιμένοντας την αλλαγή κατάστασης της εκάστοτε διεργασίας - παιδί (δηλαδή την αναστολή της λειτουργίας της). Η παράλειψή της, θα δημιουργούσε ανεξέλεγκτα διεργασίες παιδιά, με συνέπεια τη δημιουργία ορφανών διεργασιών στην περίπτωση που καταστραφεί η διεργασία - πατέρας. Γνωρίζουμε ότι, για κάθε κλήση συστήματος fork() η οποία δημιουργεί τις διεργασίες - παιδιά, χρειάζεται τουλάχιστον μία κλήση συστήματος wait() από τη διεργασία πατέρα με σκοπό τον έλεγχο της κατάστασης της διεργασίας-παιδί του.

1.4 Παράλληλος υπολογισμός αριθμητικής έκφρασης

Παρακάτω, απεικονίζεται ο κώδικας που υλοποιήσαμε για αυτό το ερώτημα:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>
#include "tree.h"
#include "proc-common.h"

/* help from given pipe-example.c*/

char calculator(char symbol, char a_char, char b_char){
        int a = a_char - '0'; int b = b_char - '0';
        switch (symbol){
                case '+': {
                                printf("The calculation of %i ", a);
                                printf( " + %i", b);
                                printf(" is equal to %i\n", a + b);
                                return a+b+'0';
                        }
                case '*': {
                                printf("The calculation of %i", a);
                                printf( " * %i", b);
                                printf(" is equal to %i\n", a*b);
                                return a*b+'0';
                        }
        }
        return -1;
}


void fork_procs(struct tree_node *root, int fd){
        int status; int i;

        if(root == NULL){ // check what happens if the node is empty
                printf("We have a problem: The node is empty \n");
                return;
        }

        printf("PID = %ld, name %s, starting...\n", (long)getpid(),root -> name
);


        change_pname(root -> name); // create the root of the tree
        // if the node has no children
        if(root -> nr_children == 0){
                printf("Create leaf %s\n",root -> name ); // print the name of
the leaf

                raise(SIGSTOP); // every leaf is ready so the parent gives the
SIGCONT

                char value;
                /* Convert root -> name from a string which represents an integ
er to an argument with value of type int */
```

```
                value = atoi(root -> name) + '0';

                if(write(fd, &value, sizeof(value)) != sizeof(value)){
                        perror("Child: write to pipe");
                        exit(1);
                }

                printf("PID = %ld, name = %s is awake and wrote to FD = %d\n",
(long)getpid(), root -> name, fd);
                exit(5);
        }

        printf("Now node %s", root -> name);
        printf(" is created\n");

        int child_pfd[2]; // create file descriptors
        printf("Parent: Creating pipe...\n");
        if(pipe(child_pfd) < 0){
                perror("pipe");
                exit(1);
        }


        // Create a pipe for each child
        pid_t child_pid[root -> nr_children];
        for(i = 0; i < root -> nr_children; i++){
                child_pid[i] = fork();
                if(child_pid[i] == 0){
                        close(child_pfd[0]);
                        fork_procs(root -> children + i, child_pfd[1]);
                        exit(1);
                }
        }
        close(child_pfd[1]);
        wait_for_ready_children(root -> nr_children);

        raise(SIGSTOP);
        printf("PID = %ld, name = %s is awake\n", (long)getpid(), root -> name)
;


        char val[2];
        for(i = 0; i < root -> nr_children; i++){
                kill(child_pid[i], SIGCONT);
                if(read(child_pfd[0], &val[i], sizeof(val[i])) != sizeof(val[i]
)){
                        perror("Child: read from pipe");
                        exit(1);
                }

                printf("Parent: received value %i from the pipe. Will now compu
te.\n", val[i]);
                child_pid[i] = waitpid(child_pid[i], &status, 0);
                explain_wait_status(child_pid[i], status);
        }
```

```
        char computed = calculator(*root -> name, val[0], val[1]);
        if(write(fd, &computed, sizeof(computed)) != sizeof(computed)){
                perror("child: write to pipe");
                exit(1);
        }
        exit(0);

}


int main(int argc, char *argv[]) //tree-example.c
{
        int pfd[2];
        struct tree_node *root;
        int status; pid_t p;
        if (argc < 2) {
                fprintf(stderr, "Usage: %s <input_tree_file>\n\n", argv[0]);
                exit(1);
        }

        root = get_tree_from_file(argv[1]);


        printf("Parent: Creating pipe...\n");

        if(pipe(pfd) < 0){
                perror("pipe");
                exit(1);
        }

        printf("Parent: Creating child...\n");

        p = fork();

        if(p<0){
                perror("fork");
                exit(1);
        }
        else if(p == 0){
                fork_procs(root, pfd[1]);
                exit(1);
        }

        wait_for_ready_children(1); // now the node is not sleeping, but is wai
ting for the children to be ready
        show_pstree(p);
        kill(p, SIGCONT );
        p=wait(&status);
        explain_wait_status(p,status);
        printf("Parent: All done, exiting...\n");
        return 0;
}
```

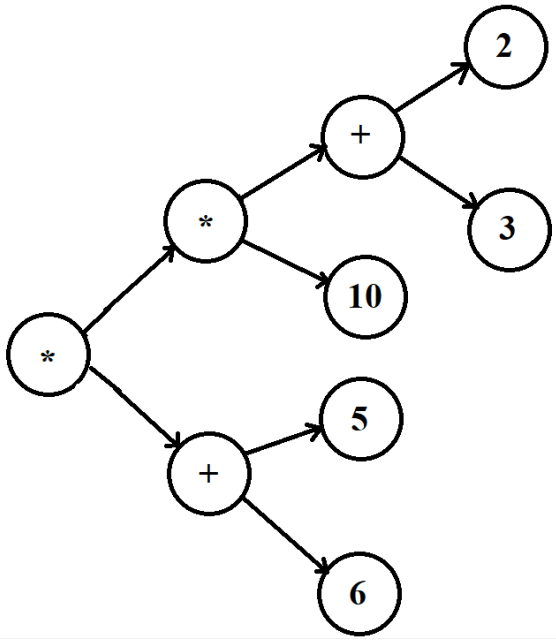
Τρέχουμε τον κώδικα του υποερωτήματος 1.4, για το δοθέν δέντρο που δίνεται στο παράδειγμα και λαμβάνουμε στην έξοδο το παρακάτω αποτέλεσμα:

```
Parent: Creating pipe...
Parent: Creating child...
PID = 32549, name *, starting...
Now node * is created
Parent: Creating pipe...
PID = 32550, name +, starting...
Now node + is created
Parent: Creating pipe...
PID = 32551, name 10, starting...
Create leaf 10
My PID = 32549: Child PID = 32551 has been stopped by a signal, signo = 19
PID = 32552, name 5, starting...
Create leaf 5
PID = 32553, name 6, starting...
Create leaf 6
My PID = 32550: Child PID = 32552 has been stopped by a signal, signo = 19
My PID = 32550: Child PID = 32553 has been stopped by a signal, signo = 19
My PID = 32549: Child PID = 32550 has been stopped by a signal, signo = 19
My PID = 32548: Child PID = 32549 has been stopped by a signal, signo = 19


*(32549)─┬─+(32550)─┬─5(32552)
         │          └─6(32553)
         └─10(32551)
```

```
PID = 32549, name = * is awake
PID = 32550, name = + is awake
PID = 32552, name = 5 is awake and wrote to FD = 7
Parent: received value 53 from the pipe. Will now compute.
My PID = 32550: Child PID = 32552 terminated normally, exit status = 5
PID = 32553, name = 6 is awake and wrote to FD = 7
Parent: received value 54 from the pipe. Will now compute.
My PID = 32550: Child PID = 32553 terminated normally, exit status = 5
The calculation of 5  + 6 is equal to 11
Parent: received value 59 from the pipe. Will now compute.
My PID = 32549: Child PID = 32550 terminated normally, exit status = 0
PID = 32551, name = 10 is awake and wrote to FD = 6
Parent: received value 58 from the pipe. Will now compute.
My PID = 32549: Child PID = 32551 terminated normally, exit status = 5
The calculation of 11 * 10 is equal to 110
My PID = 32548: Child PID = 32549 terminated normally, exit status = 0
Parent: All done, exiting...
```

Δημιουργούμε ένα δικό μας text file, στο οποίο περιγράφουμε το παρακάτω δέντρο:



Τρέχουμε τον κώδικα για το δικό μας παράδειγμα και λαμβάνουμε τα εξής:

```
Parent: Creating pipe...
Parent: Creating child...
PID = 32526, name *, starting...
Now node * is created
Parent: Creating pipe...
PID = 32527, name +, starting...
Now node + is created
Parent: Creating pipe...
PID = 32528, name *, starting...
Now node * is created
Parent: Creating pipe...
PID = 32529, name 5, starting...
Create leaf 5
PID = 32530, name 10, starting...
Create leaf 10
PID = 32531, name 6, starting...
My PID = 32527: Child PID = 32529 has been stopped by a signal, signo = 19
Create leaf 6
My PID = 32528: Child PID = 32530 has been stopped by a signal, signo = 19
PID = 32532, name +, starting...
Now node + is created
Parent: Creating pipe...
My PID = 32527: Child PID = 32531 has been stopped by a signal, signo = 19
My PID = 32526: Child PID = 32527 has been stopped by a signal, signo = 19
PID = 32533, name 3, starting...
Create leaf 3
PID = 32534, name 2, starting...
Create leaf 2
```

```
My PID = 32532: Child PID = 32533 has been stopped by a signal, signo = 19
My PID = 32532: Child PID = 32534 has been stopped by a signal, signo = 19
My PID = 32528: Child PID = 32532 has been stopped by a signal, signo = 19
My PID = 32526: Child PID = 32528 has been stopped by a signal, signo = 19
My PID = 32525: Child PID = 32526 has been stopped by a signal, signo = 19


*(32526)─┬─*(32528)─┬─+(32532)─┬─2(32534)
         │          │          └─3(32533)
         │          └─10(32530)
         └─+(32527)─┬─5(32529)
                    └─6(32531)


PID = 32526, name = * is awake
PID = 32527, name = + is awake
PID = 32529, name = 5 is awake and wrote to FD = 7
Parent: received value 53 from the pipe. Will now compute.
My PID = 32527: Child PID = 32529 terminated normally, exit status = 5
PID = 32531, name = 6 is awake and wrote to FD = 7
Parent: received value 54 from the pipe. Will now compute.
My PID = 32527: Child PID = 32531 terminated normally, exit status = 5
The calculation of 5  + 6 is equal to 11
Parent: received value 59 from the pipe. Will now compute.
My PID = 32526: Child PID = 32527 terminated normally, exit status = 0
PID = 32528, name = * is awake
PID = 32530, name = 10 is awake and wrote to FD = 7
Parent: received value 58 from the pipe. Will now compute.
My PID = 32528: Child PID = 32530 terminated normally, exit status = 5
The calculation of 10 * 5 is equal to 50
Parent: received value 98 from the pipe. Will now compute.
My PID = 32526: Child PID = 32528 terminated normally, exit status = 0
The calculation of 11 * 50 is equal to 550
My PID = 32525: Child PID = 32526 terminated normally, exit status = 0
Parent: All done, exiting...
```

Ερωτήσεις:

1. Στην συγκεκριμένη άσκηση εκτελούνται μόνο προσθέσεις και πολλαπλασιασμοί, δηλαδή μόνο αντιμεταθετικές πράξεις. Συνεπώς, χρειαζόμαστε ένα pipe για να καλυφθεί μια διεργασία (ο γονιός και τα δύο παιδιά του).
   Γενικά, δεν μπορεί, για κάθε αριθμητικό τελεστή, να χρησιμοποιηθεί μόνο μία σωλήνωση. Αν, για παράδειγμα, είχαμε αφαίρεση ή διαίρεση, δηλαδή μη αντιμεταθετικές πράξεις, θα έπρεπε με κάποιον τρόπο να διαχωρίζουμε τον αφαιρέτη από τον αφαιρετέο και τον διαιρέτη από τον διαιρετέο αντίστοιχα. Συνεπώς, θα χρειαζόμασταν άλλη μια σωλήνωση.
2. Σε ένα σύστημα πολλαπλών επεξεργαστών η αποτίμηση της έκφρασης από δέντρο διεργασιών μπορεί να εκτελεστεί σε λογαριθμικό χρόνο, καθώς υπάρχει δυνατότητα να ανατεθεί κάθε διεργασία πατέρα σε διαφορετικό πυρήνα. Οπότε, οι κόμβοι του δέντρου με το ίδιο ύψος θα διασχίζονται, με αυτόν τον τρόπο, παράλληλα.

Makefile

Το Makefile που χρησιμοποιήθηκε για τα παραπάνω υποερωτήματα, είναι το κάτωθι:

```
.PHONY: all clean

all: fork-example tree-example ask2-fork ask2-signals ex2_1_1 ex2_1_2 ex2_1_3 e
x2_1_4

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

ex2_1_1: proc-common.o ex2_1_1.o
	$(CC) $(CFLAGS) $^ -o $@
ex2_1_2: proc-common.o tree.o ex2_1_2.o
	$(CC) $(CFLAGS) $^ -o $@
ex2_1_3: proc-common.o tree.o ex2_1_3.o
	$(CC) $(CFLAGS) $^ -o $@
ex2_1_4: proc-common.o tree.o ex2_1_4.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}
```