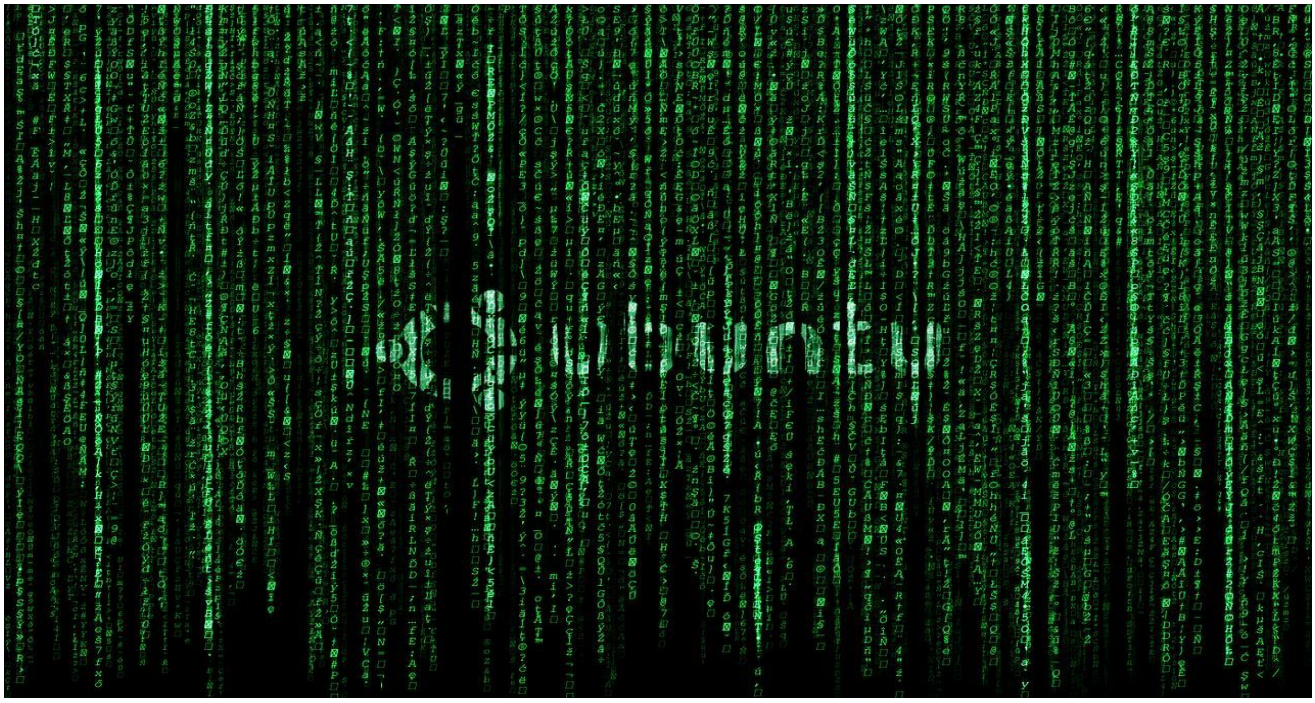
10 Μαίου 2020



ΑΚΑΔΗΜΑΪΚΟ ΕΤΟΣ : 2019-2020

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Άσκηση 2η - Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία

| Καπαρού Αλεξάνδρα (03117100)
Χλαπάνης Οδυσσέας (03117023)
Ομάδα : oslaba21

Άσκηση 2.1

➢ Πηγαίος κώδικας της 2.1

```
#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)
{
	pid_t b,c,d;
	int status;
	printf("Node %s with pid %ld starts...\n","A",(long)getpid());
	change_pname("A");
	b = fork();
	if (b < 0)
	{
		perror("fork");
		exit(EXIT_FAILURE);
	}

	if (b==0)
	{
	printf("Node %s with pid %ld starts...\n","B",(long)getpid());
	change_pname("B");
	d = fork();
	if (d < 0)
	{
		perror("fork");
		exit(EXIT_FAILURE);
	}

	if (d==0)
	{
```

```
        printf("Node %s with pid %ld starts...\n","D",(long)getpid());
        change_pname ("D");
        printf("Node D is sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("Node D is exiting...\n");
        exit (13);
        }

        printf("Parent B is waiting for child %s with pid %ld to
terminate...\n","D",(long)d);
        wait(&status);
        explain_wait_status(d, status);

        printf("Parent B: All done, exiting...\n");

        exit(19);
        }
        c = fork();

        if (c<0)
        {
                perror("fork");
                exit(EXIT_FAILURE);
        }

        if (c==0)
        {
        printf("Node %s with pid %ld starts...\n","C",(long)getpid());
        change_pname ("C");
        printf("Node C is sleeping...\n");
        sleep(SLEEP_PROC_SEC);
        printf("Node C is exiting...\n");
        exit (17);
        }

        printf("Parent A is waiting for his children to terminate...\n");
        wait(&status);
        explain_wait_status(c, status);
        wait(&status);
        explain_wait_status(b, status);

        printf("Parent A: All done, exiting...\n");
        exit(16);
}

int main(void)
{
```

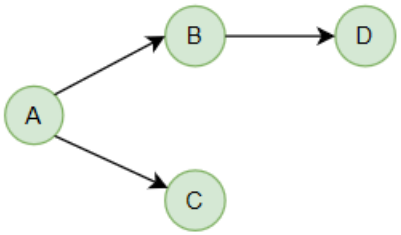
```
	pid_t pid;
	int status;

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

	/*
	 * Father
	 */

	/* for ask2-{fork, tree} */
	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;
}
```

Το Makefile που φτιάξαμε:

```
CC = gcc
CFLAGS = Wall

all: skeletos.o proc-common.o
	$(CC) -$(CFLAGS) skeletos.o proc-common.o -o skeletos

proc-common.o: proc-common.c
	$(CC) -$(CFLAGS) -c proc-common.c

skeletos.o: skeletos.c
	$(CC) -$(CFLAGS) -c skeletos.c
```

➢ Έξοδος εκτέλεσης προγραμμάτων της 2.1

- Για το δέντρο:



Έχουμε σαν έξοδο:

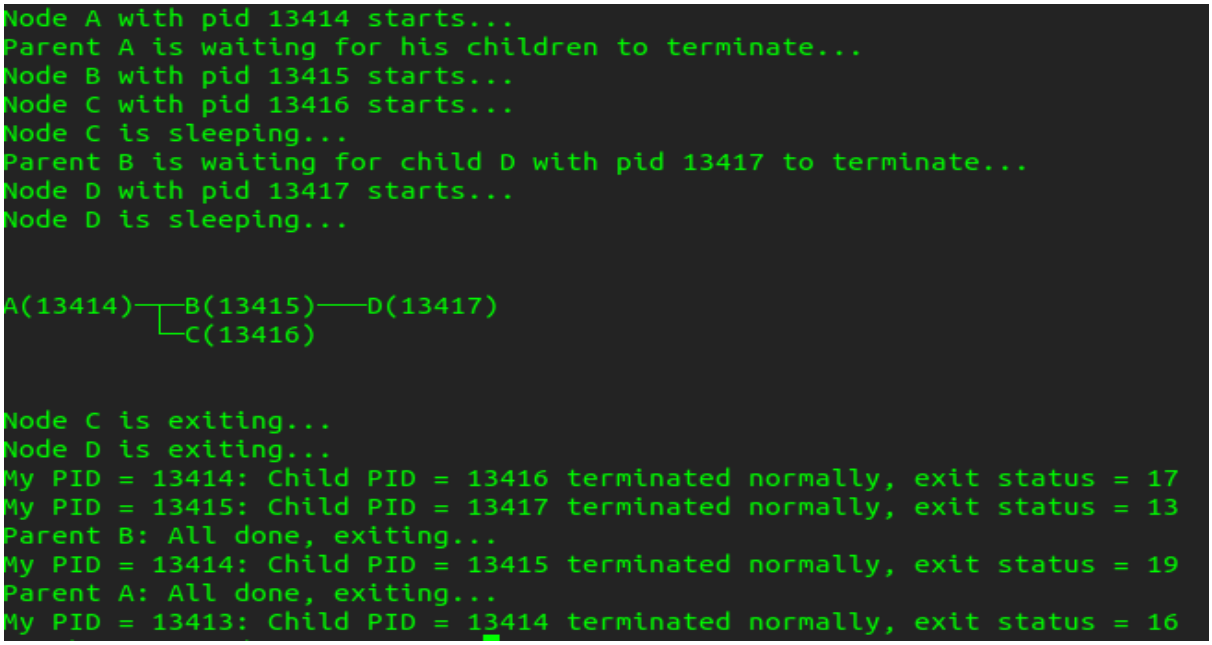
```
Node A with pid 13970 starts...
Parent A is waiting for his children to terminate...
Node B with pid 13971 starts...
Node C with pid 13972 starts...
Node C is sleeping...
Parent B is waiting for child D with pid 13973 to terminate...
Node D with pid 13973 starts...
Node D is sleeping...


A(13970)─┬─B(13971)──D(13973)
         └─C(13972)


Node C is exiting...
Node D is exiting...
My PID = 13970: Child PID = 13972 terminated normally, exit status = 17
My PID = 13971: Child PID = 13973 terminated normally, exit status = 13
Parent B: All done, exiting...
My PID = 13970: Child PID = 13971 terminated normally, exit status = 19
Parent A: All done, exiting...
My PID = 13969: Child PID = 13970 terminated normally, exit status = 16
```

➢ Ερωτήσεις της 2.1

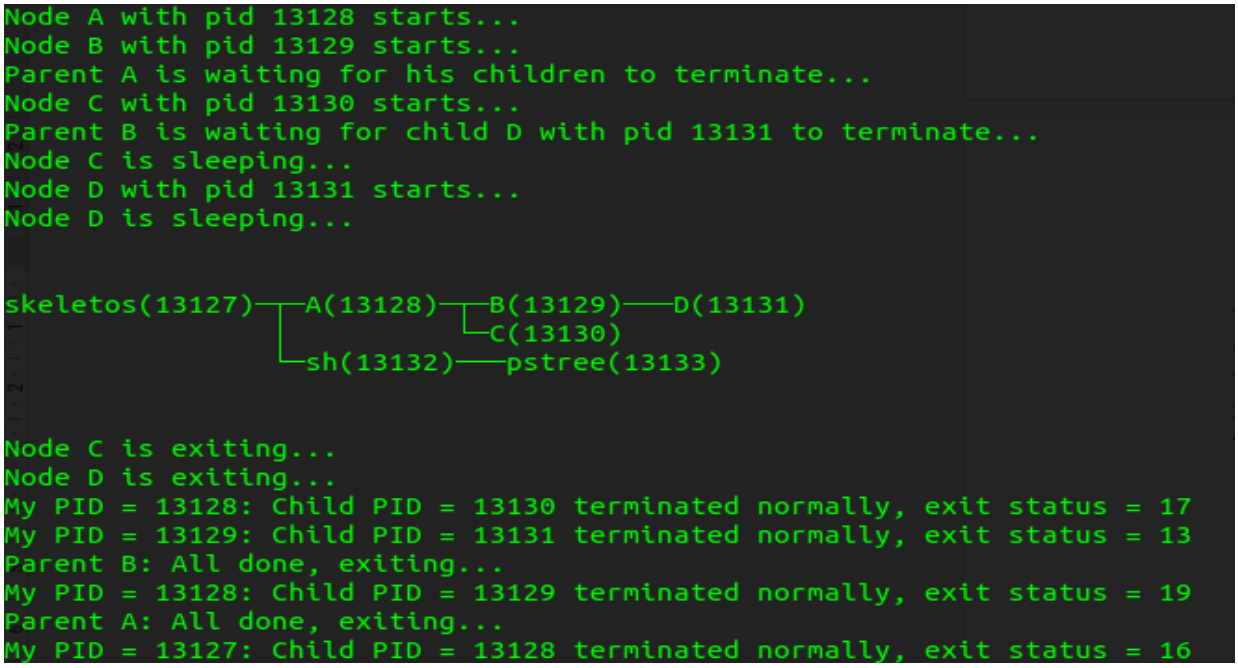
1. Η διεργασία Α είναι πατέρας των διεργασιών Β,C. Υπό κανονικές συνθήκες, πρώτα πεθαίνει μια διαδικασία-παιδί, το status της γίνεται EXIT_ZOMBIE και στην συνέχεια ο πατέρας της διαδικασίες ειδοποιείται για αυτό με ένα σήμα SIGCHLD. Τότε ο πατέρας εκτελεί την κλήση wait για να διαβάσει το exit status και τέλος αφαιρείται η zombie διαδικασία από την μνήμη. Εάν ο γονέας πεθάνει πρώτα, τότε όπως γνωρίζουμε από την θεωρία μας, οι διεργασίες Β και C γίνονται παιδιά της init με PID = 1 η οποία κάνει συνεχώς wait.

2. Παρατηρούμε ότι αλλάζοντας την εντολή show_pstree(pid) με την εντολή show_pstree(getpid())το δέντρο έχει άλλη μορφή. Για να γίνει πιο παραστατική η εξήγηση συγκρίνουμε τις δύο παρακάτω φωτογραφίες.

   Με χρήση της show_pstree(pid):

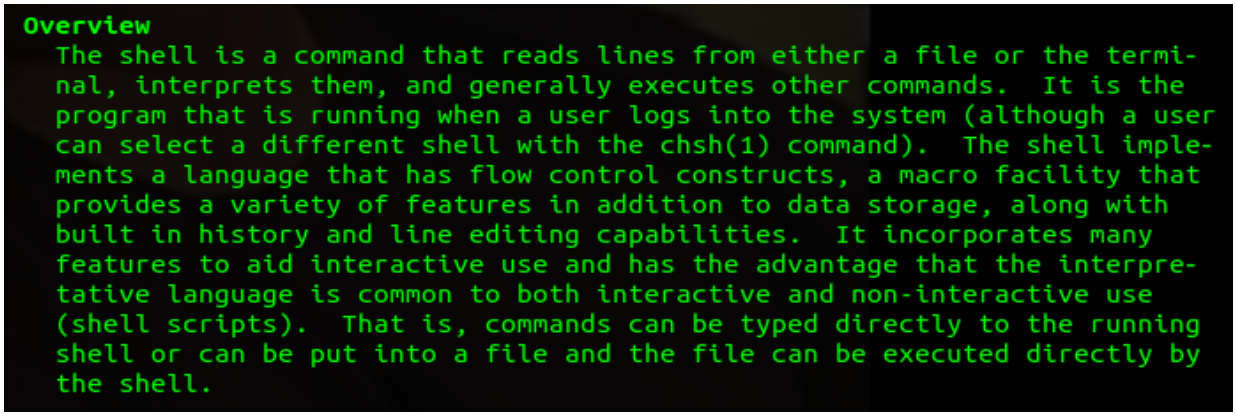


   Με χρήση της show_pstree(getpid()):



Αυτό συμβαίνει επειδή η κλήση της συνάρτησης getpid() επιστρέφει το pid της διεργασίας που πραγματοποιεί την κλήση οπότε, στην

συγκεκριμένη περίπτωση της show_pstree(getpid()), θα επιστρέψει το pid της διεργασίας της main του προγράμματος μας, δηλαδή της skeletos (εφόσον είχαμε εκτελέσει το αρχείο ./skeletos). Επομένως, με όρισμα την συνάρτηση getpid() θα εμφανιστεί ολόκληρο το δέντρο διεργασιών με ρίζα την διεργασία skeletos. Παρατηρούμε ότι εκτός από το δέντρο διεργασιών A,B,C,D υπάρχουν άλλες δύο διεργασίες, οι sh και pstree. Γράφοντας στο terminal: man sh πήραμε τα παρακάτω:



Βλέπουμε δηλαδή ότι η διεργασία skeletos κάνει fork μια διεργασία sh(shell) η οποία με την σειρά της καλεί την pstree προκειμένου να εμφανίσει το δέντρο.

3. Όταν πολλαπλοί χρήστες διαχειρίζονται ένα υπολογιστικό σύστημα, υπάρχει το ενδεχόμενο ένας χρήστης να ξεκινήσει υπερβολικά πολλές διεργασίες κάτι το οποίο θα οδηγήσει σε εξάντληση των πόρων και αστάθεια του συστήματος. Γι’αυτό το λόγο ο διαχειριστής ενός τέτοιου συστήματος πολλές φορές θέτει ένα άνω όριο στον αριθμό των διεργασιών που μπορεί ο κάθε χρήστης να εκκινήσει ώστε όταν κάποιος υπερβεί αυτό το όριο, προκειμένου να μπορέσει να δημιουργήσει καινούργιες διεργασίες, θα πρέπει αναγκαστικά να διαγράψει παλιές. Φυσικά κάτι τέτοιο μπορεί να λειτουργήσει και ως προστασία απέναντι σε denial-of-service(DoS) επιθέσεις ενάντια σε Unix-based systems. Κλασικό παράδειγμα είναι η λεγόμενη επίθεση forkbomb η οποία οδηγεί σε εξάντληση της μνήμης του συστήματος και η οποία είναι ουσιαστικά μία συνάρτηση που εκτελείται αναδρομικά, δημιουργώντας συνεχώς νέες διεργασίες-αντίγραφα της αρχικής. Αυτό έχει σαν αποτέλεσμα το σύστημα να κρασάρει και να πρέπει να γίνει reboot προκειμένου να συνεχίσει την ομαλή του

λειτουργία, καθώς για να γίνει αυτό θα πρέπει έχουν διαγραφεί όλα τα στιγμιότυπα της αρχικής διεργασίας.

Άσκηση 2.2

➢ Πηγαίος κώδικας της 2.2

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

#define SLEEP_TREE_SEC 2
#define SLEEP_PROC_SEC 6

void forkprocs (struct tree_node *node)
{
int i;
int status;
pid_t pid;
printf("Hello, this is node %s and I have one or more children \n", node-
>name);


change_pname(node->name);


for (i=0;i<node->nr_children;i++)   //we traverse all the children of the node
{
if (fork()==0)    //create a child
{
change_pname((node->children + i)->name);
if ((node->children + i)->nr_children != 0) //This node has more children
{
forkprocs(node->children+i);
}
else if ((node->children + i)->nr_children == 0) //This node doesn't have
children
{
printf("Hello, this is node %s, I am a leaf and now I will sleep\n",(node-
>children+i)->name);
```

```
sleep(SLEEP_PROC_SEC);  //put leaves to sleep
exit(0);
}
}
}


for (i=0; i<node->nr_children; i++){
      pid = wait(&status);
      explain_wait_status(pid, status);
   }
   exit(0);
}


int main(int argc, char *argv[])
{
      struct tree_node *root;
      pid_t pid;
      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 = fork();
      if (pid<0)
      {
      perror("fork");
      exit(EXIT_FAILURE);
      }
      if (pid==0)
      {
      forkprocs(root);
      }
      sleep (SLEEP_TREE_SEC);
      show_pstree (pid);
      pid = wait (&status);
      explain_wait_status (pid, status);
      return 0;
}
```

Το Makefile που φτιάξαμε:

CC = gcc
CFLAGS = Wall
all: 2.2.o proc-common.o tree.o
$(CC) -$(CFLAGS) 2.2.o proc-common.o tree.o -o 2.2

tree.o: tree.c
$(CC) -$(CFLAGS) -c tree.c

proc-common.o: proc-common.c
$(CC) -$(CFLAGS) -c proc-common.c

2.2.o: 2.2.c
$(CC) -$(CFLAGS) -c 2.2.c

➢ Έξοδος εκτέλεσης προγραμμάτων της 2.2

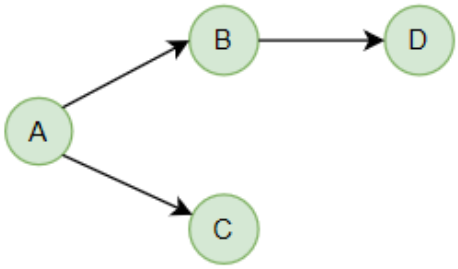
- Για το δέντρο:

A
2
B
C

B
1
D

D
0

C
0



Έχουμε σαν έξοδο:

```
Hello, this is node A and I have one or more children
Hello, this is node B and I have one or more children
Hello, this is node C, I am a leaf and now I will sleep
Hello, this is node D, I am a leaf and now I will sleep


A(14061)─┬─B(14062)──D(14064)
         └─C(14063)


My PID = 14061: Child PID = 14063 terminated normally, exit status = 0
My PID = 14062: Child PID = 14064 terminated normally, exit status = 0
My PID = 14061: Child PID = 14062 terminated normally, exit status = 0
My PID = 14060: Child PID = 14061 terminated normally, exit status = 0
```

- Για το δέντρο:

A
3
B
C
D

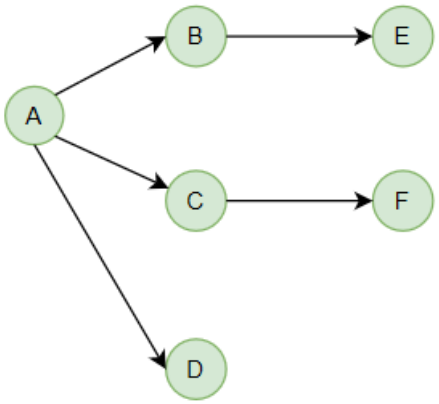
B
1
E

E
0

C
1
F

F
0

D
0

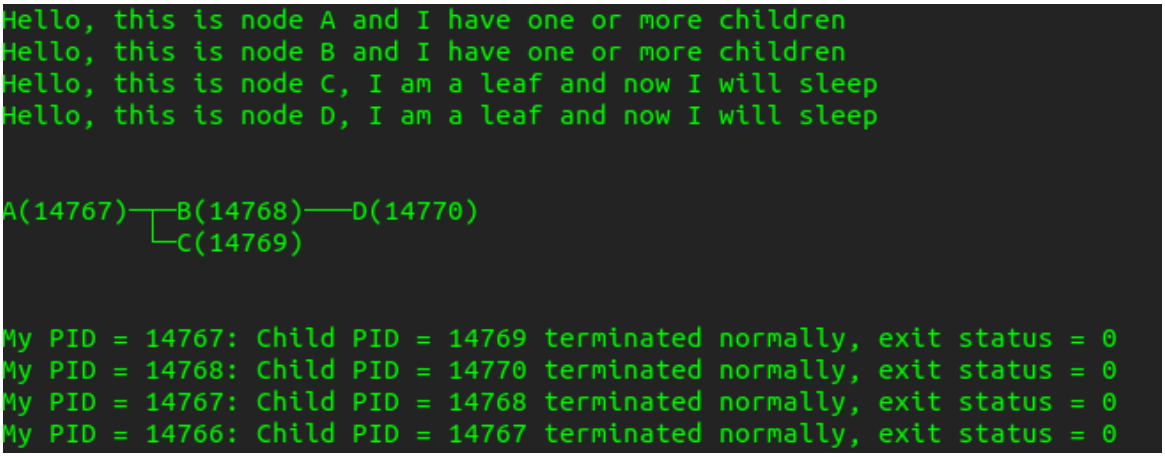


Έχουμε σαν έξοδο:

```
Hello, this is node A and I have one or more children
Hello, this is node B and I have one or more children
Hello, this is node C and I have one or more children
Hello, this is node D, I am a leaf and now I will sleep
Hello, this is node E, I am a leaf and now I will sleep
Hello, this is node F, I am a leaf and now I will sleep


A(14080)─┬─B(14081)──E(14084)
         ├─C(14082)──F(14085)
         └─D(14083)


My PID = 14080: Child PID = 14083 terminated normally, exit status = 0
My PID = 14081: Child PID = 14084 terminated normally, exit status = 0
My PID = 14082: Child PID = 14085 terminated normally, exit status = 0
My PID = 14080: Child PID = 14081 terminated normally, exit status = 0
My PID = 14080: Child PID = 14082 terminated normally, exit status = 0
My PID = 14079: Child PID = 14080 terminated normally, exit status = 0
```

➢ Ερωτήσεις της 2.2

1. Όπως βλέπουμε και από την φωτογραφία παρακάτω, τα μηνύματα έναρξης εμφανίζονται ανά επίπεδα (Breadth-First), αφού πρώτα δημιουργείται το επίπεδο 0(A),μετά το επίπεδο 1(B,C) και τέλος το επίπεδο 2(D). Όσον αφορά τα μηνύματα τερματισμού βλέπουμε ότι αυτά εμφανίζονται με ένα μίγμα των τρόπων κατά βάθος (Depth-First) και κατά πλάτος (Breadth-First). Πιο συγκεκριμένα, ξεκινάει ο τερματισμός από ένα επίπεδο στο οποίο υπάρχει κόμβος-φύλλο(εδώ ο C) και προχωρώντας στους υπόλοιπους κόμβους του επιπέδου, όπου υπάρχει κόμβος-πατέρας τερματίζονται πρώτα όλα τα παιδιά του και τελευταίος αυτός. Κάτι τέτοιο είναι αναμενόμενο εφόσον για να πραγματοποιηθεί το διάβασμα του αρχείου που περιγράφει το δέντρο πρέπει οι κόμβοι να είναι διατεταγμένοι κατά βάθος.



Άσκηση 2.3

➢ Πηγαίος κώδικας της 2.3

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

```
#define SLEEP_TREE_SEC 2
#define SLEEP_PROC_SEC 6

int got_sigusr1=0;

void sighandler(int sugnum)
{
	got_sigusr1 = 1;
}

void forkprocs (struct tree_node *node)
{
	int i;
	int status;
	pid_t pid,pid2[node->nr_children];
	printf("PID = %ld, name %s,that has children,is starting...\n", (long)getpid(),
node->name);
	change_pname(node->name);
	for (i=0;i<node->nr_children;i++)  //we traverse all the children of the node
	{
	pid2[i]=fork();
	if (pid2[i]<0)
		{
		perror("fork");
		exit(EXIT_FAILURE);
		}
	if(pid2[i]==0)//if (fork()==0)    //create a child
		{
		struct tree_node *child = node->children + i;
		//change_pname(child->name);
			if (child->nr_children != 0) //This node has more children
			{
			forkprocs(child);
			}
			else if (child->nr_children == 0) //This is a leaf!
			{
 {
			change_pname(child->name);
			raise(SIGSTOP);
			printf("PID = %ld, name = %s is a leaf and is awake\n",(long)getpid(),
child->name);
			exit(0);
			}
		}
	}//this is the parent, all children have been created

 //the parent will sleep now
```

```
        wait_for_ready_children(node->nr_children);
        printf("PID = %ld, I am parent %s and my children are ready. Now I will
sleep\n",(long)getpid(),node->name);

        raise(SIGSTOP);
        printf("PID = %ld, name = %s is awake. Now I will wake up my children.\n",
(long)getpid(), node->name);

        for (i=0; i<node->nr_children; i++){
                printf("I will wake up child with name %s.\n", (node->children + i)->name);
                kill(pid2[i],SIGCONT);
                pid = wait(&status);
                explain_wait_status(pid, status);
                printf("The child %s is ready\n", (node->children + i)->name);
        }
        printf("I am %s. All of my children are ready, now I terminate.\n",node->name);
    exit(0);
}

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

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

        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("fork");
        exit(EXIT_FAILURE);
        }
        if (pid==0)
        {
        forkprocs(root);
```

```
    }
    wait_for_ready_children(1);
    show_pstree (pid);
    kill(pid,SIGCONT);
    pid = wait (&status);
    explain_wait_status (pid, status);
    return 0;
}
```

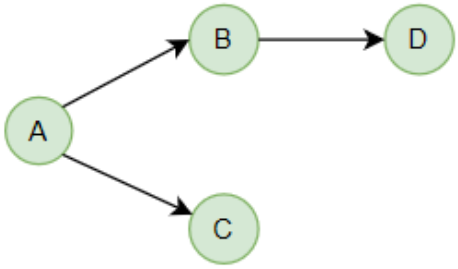
Το Makefile που φτιάξαμε:

```
CC = gcc
CFLAGS = Wall
all: 2.3.o proc-common.o tree.o
    $(CC) -$(CFLAGS) 2.3.o proc-common.o tree.o -o 2.3


tree.o: tree.c
    $(CC) -$(CFLAGS) -c tree.c


proc-common.o: proc-common.c
    $(CC) -$(CFLAGS) -c proc-common.c


2.3.o: 2.3.c
    $(CC) -$(CFLAGS) -c 2.3.c
```

➢ Έξοδος εκτέλεσης προγραμμάτων της 2.3

- Για το δέντρο:

```
A
2
B
C

B
1
D

D
0

C
0
```



Έχουμε σαν έξοδο:

```
PID = 14217, name A,that has children,is starting...
PID = 14218, name B,that has children,is starting...
I sent a signal to node C with pid 14219 to continue
My PID = 14217: Child PID = 14219 has been stopped by a signal, signo = 19
I sent a signal to node D with pid 14220 to continue
My PID = 14218: Child PID = 14220 has been stopped by a signal, signo = 19
My PID = 14217: Child PID = 14218 has been stopped by a signal, signo = 19
My PID = 14216: Child PID = 14217 has been stopped by a signal, signo = 19


A(14217)─┬─B(14218)──D(14220)
         └─C(14219)


PID = 14220, name = B is awake
My PID = 14218: Child PID = 14220 terminated normally, exit status = 0
My PID = 14217: Child PID = 14218 terminated normally, exit status = 0
PID = 14219, name = A is awake
My PID = 14217: Child PID = 14219 terminated normally, exit status = 0
My PID = 14216: Child PID = 14217 terminated normally, exit status = 0
```

- Για το δέντρο:

A
3
B
C
D

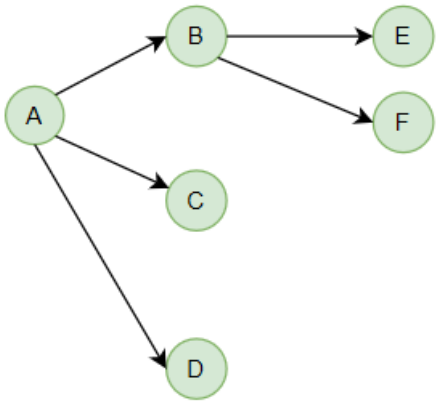
B
2
E
F

E
0

F
0

C
0

D
0



Έχουμε σαν έξοδο:

```
PID = 14175, name A,that has children,is starting...
PID = 14176, name B,that has children,is starting...
I sent a signal to node C with pid 14177 to continue
My PID = 14175: Child PID = 14177 has been stopped by a signal, signo = 19
I sent a signal to node D with pid 14178 to continue
My PID = 14175: Child PID = 14178 has been stopped by a signal, signo = 19
I sent a signal to node E with pid 14179 to continue
I sent a signal to node F with pid 14180 to continue
My PID = 14176: Child PID = 14179 has been stopped by a signal, signo = 19
My PID = 14176: Child PID = 14180 has been stopped by a signal, signo = 19
My PID = 14175: Child PID = 14176 has been stopped by a signal, signo = 19
My PID = 14174: Child PID = 14175 has been stopped by a signal, signo = 19


A(14175)─┬─B(14176)─┬─E(14179)
         │          └─F(14180)
         ├─C(14177)
         └─D(14178)


PID = 14179, name = B is awake
My PID = 14176: Child PID = 14179 terminated normally, exit status = 0
PID = 14180, name = B is awake
My PID = 14176: Child PID = 14180 terminated normally, exit status = 0
My PID = 14175: Child PID = 14176 terminated normally, exit status = 0
PID = 14177, name = A is awake
My PID = 14175: Child PID = 14177 terminated normally, exit status = 0
PID = 14178, name = A is awake
My PID = 14175: Child PID = 14178 terminated normally, exit status = 0
My PID = 14174: Child PID = 14175 terminated normally, exit status = 0
```

➢ Ερωτήσεις της 2.3

1. Χρησιμοποιώντας την sleep() ορίζαμε ένα επιθυμητό χρονικό διάστημα το οποίο θεωρούσαμε αρκετό προκειμένου να προλάβει να δημιουργηθεί το δέντρο διεργασιών. Κατά την δημιουργία μεγάλου δέντρου διεργασιών όμως βλέπουμε ότι μία τέτοια λύση δεν είναι καθόλου αποδοτική, αφενός στην περίπτωση που ορίσουμε μικρότερο χρόνο από τον απαιτούμενο, επειδή τότε το δέντρο μας δεν θα προλάβει να ολοκληρωθεί (που σε άγνωστα προς εμάς δέντρα αποτελεί σημαντικό πρόβλημα καθώς δεν θα μπορούμε να αποφανθούμε για την ορθότητα του αποτελέσματος) αφετέρου στην περίπτωση που ορίσουμε μεγαλύτερο χρόνο από τον απαιτούμενο επειδή τότε θα πρέπει αναγκαστικά να περιμένουμε μέχρι να λήξει αυτός ο χρόνος προκειμένου να συνεχίσει η εκτέλεση του προγράμματος, έχουμε δηλαδή περιττούς χρόνους αναμονής. Γι’αυτούς τους λόγους η πλέον αποτελεσματική λύση για τον συγχρονισμό των διεργασιών είναι η λύση σημάτων. Με αυτά έχουμε μεγαλύτερη εποπτεία

όσον αφορά την έναρξη και τον τερματισμό κάθε διεργασίας καθώς με την χρήση των κατάλληλων σημάτων μπορούμε να ορίσουμε ανά πάσα στιγμή αν θέλουμε να σταματήσουμε προσωρινά, να συνεχίσουμε ή και να σκοτώσουμε μια διεργασία χωρίς να χρειαστεί να περιμένουμε.

2. Διαβάζοντας το αρχείο proc-common.c όπου έχει οριστεί η συνάρτηση wait_for_ready_children() έχουμε:

```
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);
                }
        }
}
```

Βλέπουμε λοιπόν ότι η συνάρτηση αυτή παίρνει σαν όρισμα έναν αριθμό (cnt), δηλαδή τον αριθμό των παιδιών κάθε κόμβου, και με βάση αυτόν εκτελεί wait() μέχρι όλα τα παιδιά της διεργασίας που την καλεί να έχουν σταματήσει προσωρινά. Παρατηρούμε επίσης ότι έχουμε μία συνθήκη που ελέγχει αν κάποιο από τα παιδιά που εξετάζουμε έχει σταματήσει την εκτέλεση του για κάποιον άλλο λόγο πέρα κάποιο signal (αφού η εντολή WIFSTOPPED επιστρέφει true μόνο αν η υπό εξέταση διεργασία έχει σταματήσει από κάποιο σήμα). Αν έχει γίνει κάτι τέτοιο τότε πετάει ένα μήνυμα λάθους και κάνει exit. Στο δικό μας πρόγραμμα, καλέσαμε την wait_for_ready_children() μέσα στην main με παράμετρο 1 επειδή η αρχική διεργασία περιμένει να ολοκληρωθεί όλο το δέντρο ώστε στην συνέχεια να μας το τυπώσει. Μέσα στην συνάρτηση forprocs, η wait_for_ready_children() κλήθηκε με παράμετρο τον αριθμό των παιδιών της εκάστοτε διεργασίας που την καλεί, ώστε να περιμένουμε να δημιουργηθούν όλα τα παιδιά της προτού

συνεχίσει η επόμενη εντολή. Σε περίπτωση που την παραλείπαμε θα δημιουργούνταν πρόβλημα εάν παραδείγματος χάρη η διεργασία-πατέρας έστελνε στο παιδί σήμα να συνεχίσει (SIGCONT) πρωτού αυτό να σταματήσει (SIGSTOP), με αποτέλεσμα να αγνοηθεί το σήμα του πατέρα και το παιδί στην συνέχεια να παραμείνει σταματημένο, κάτι το οποίο θα εμπόδιζε την ομαλή λειτουργία του προγράμματος. Η χρήση λοιπόν της wait_for_ready_children() μας κάνει να γνωρίζουμε την κατάσταση του δέντρου πριν επιχειρήσουμε να στείλουμε SIGCONT σε μια διεργασία που μπορεί να μην έχει δημιουργηθεί ή να μην έχει γίνει pause.

Άσκηση 2.4

➢ Πηγαίος κώδικας της 2.4

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

#define SLEEP_TREE_SEC 2
#define SLEEP_PROC_SEC 6



pid_t forkprocs (struct tree_node *node,int fd[2])

{
	int i=0;
	int status;
	pid_t pid;

	int fd1[2], fd2[2]; //We need two pipes,one for the first child and one for the second

	int num1, num2;  //The two numbers of every caculation
	pid = fork();
```

```
	if (pid<0){
		perror("fork");
		exit(-1);
	}
	if (pid==0){
		change_pname(node->name);
		printf("We have created %s \n", node->name);
		if (node->nr_children==0){	//This node doesn't have children
			num1 = atoi(node->name);  //converts the string to an int
			printf("Now %s with pid %d sends %d to pipe \n", node-
>name,getpid(),num1);
			if (write(fd[1],&num1,sizeof(num1))<0){ //We send the name to the
father
				perror("write pipe");
				exit(-1);
			}
			sleep(SLEEP_PROC_SEC);
			exit(0);
		}
		if (pipe(fd1)<0){		//pipe for the first child
			perror ("pipe");
			exit(-1);
		}
		forkprocs(node->children, fd1); //for the first child

		if (pipe(fd2)<0){
			perror ("pipe");	//another pipe for the second child
			exit(-1);
		}
forkprocs(node->children+1, fd2); //for the second child
		for(i=0;i<2;i++)
		{
		pid = wait(&status);		//because we have two children
		explain_wait_status(pid, status);
		}

		if (read(fd1[0],&num1,sizeof(num1)) <0){	//read from first pipe the first
child
				perror("read pipe");
				exit(-1);
			}
		if (read(fd2[0],&num2,sizeof(num2)) <0){	//read from second pipe the
second child
				perror("read pipe");
				exit(-1);
			}
```

```
        printf("Calculation %s with pid %d reads %d and %d from pipe \n" , node->name, getpid(),num1,num2);
        i=num1;   //to save for the output
        if (node->name[0]=='+') num1=num1+num2;          //choose operation
        else if (node->name[0]=='*') num1=num1*num2;

        printf("Calculation %s with pid %d sends the result %d (%d %s %d) to pipe \n",node->name,getpid(),num1,i,node->name,num2);
        if (write(fd[1],&num1,sizeof(num1))<0){     //send the result
                perror("write pipe");
                exit(-1);
             }
        exit(0);
    }
    return pid;

}

int main(int argc, char *argv[])
{
    struct tree_node *root;
    pid_t pid;
    int fd[2];   //pipe between father and child
    int num1;
    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]);

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

    if (pipe(fd)<0)
    {
        perror ("pipe");
        return EXIT_FAILURE;
    }
    pid=forkprocs(root,fd);   //We have the pid of the root
    sleep (SLEEP_TREE_SEC);
    show_pstree (pid);
    pid = wait (&status);
```

```
        explain_wait_status (pid, status);

        if (read(fd[0],&num1,sizeof(num1))<0)
        {
        perror("read pipe");              //We read the result from the pipe
        exit(-1);
        }
        printf("The result is %d! \n",num1);
        return 0;
}
```

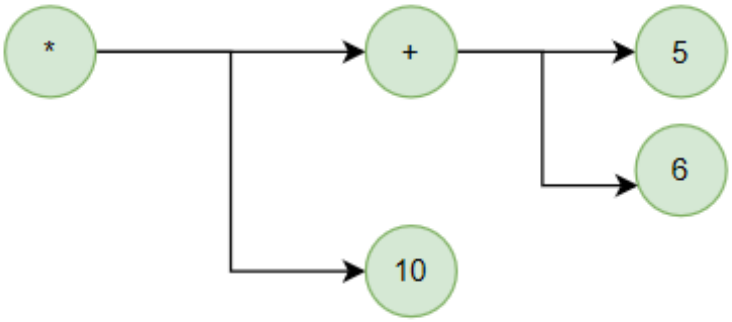
Το Makefile που φτιάξαμε:

```
CC = gcc
CFLAGS = Wall
all: 2.4.o proc-common.o tree.o
    $(CC) -$(CFLAGS) 2.4.o proc-common.o tree.o -o 2.4

tree.o: tree.c
    $(CC) -$(CFLAGS) -c tree.c

proc-common.o: proc-common.c
    $(CC) -$(CFLAGS) -c proc-common.c

2.4.o: 2.4.c
    $(CC) -$(CFLAGS) -c 2.4.c
```

➢ Έξοδος εκτέλεσης προγραμμάτων της 2.4

- Για το δέντρο:

```
*
2
10
+

10
0

+
2
5
6
```



5
0

6
0

Έχουμε σαν έξοδο:

```
We have created *
We have created 10
Now 10 with pid 7343 sends 10 to pipe
We have created +
We have created 5
Now 5 with pid 7345 sends 5 to pipe
We have created 6
Now 6 with pid 7346 sends 6 to pipe


*(7342)─┬─+(7344)─┬─5(7345)
        │         └─6(7346)
        └─10(7343)


My PID = 7342: Child PID = 7343 terminated normally, exit status = 0
My PID = 7344: Child PID = 7345 terminated normally, exit status = 0
My PID = 7344: Child PID = 7346 terminated normally, exit status = 0
Calculation + with pid 7344 reads 5 and 6 from pipe
Calculation + with pid 7344 sends the result 11 (5 + 6) to pipe
My PID = 7342: Child PID = 7344 terminated normally, exit status = 0
Calculation * with pid 7342 reads 10 and 11 from pipe
Calculation * with pid 7342 sends the result 110 (10 * 11) to pipe
My PID = 7341: Child PID = 7342 terminated normally, exit status = 0
The result is 110!
```

- Για το δέντρο:

+
2
10
***

10
0

***
2
+
4

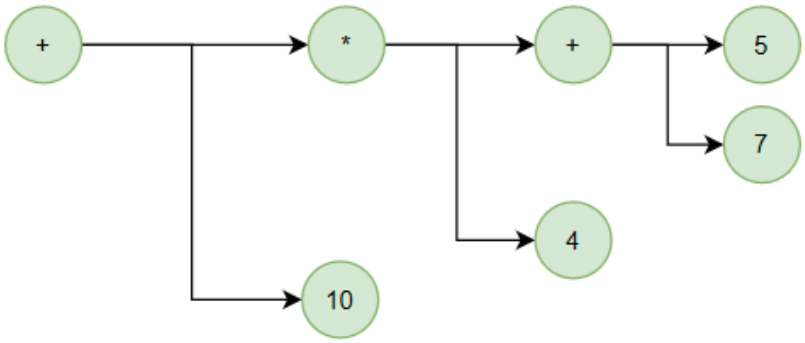
+

2
5
7

5
0

7
0

4
0



Έχουμε σαν έξοδο:

```
We have created +
We have created 10
We have created *
Now 10 with pid 7351 sends 10 to pipe
We have created +
We have created 4
Now 4 with pid 7354 sends 4 to pipe
We have created 5
We have created 7
Now 5 with pid 7355 sends 5 to pipe
Now 7 with pid 7356 sends 7 to pipe


+(7350)─┬─*(7352)─┬─+(7353)─┬─5(7355)
        │         │         └─7(7356)
        │         └─4(7354)
        └─10(7351)


My PID = 7350: Child PID = 7351 terminated normally, exit status = 0
My PID = 7352: Child PID = 7354 terminated normally, exit status = 0
My PID = 7353: Child PID = 7355 terminated normally, exit status = 0
My PID = 7353: Child PID = 7356 terminated normally, exit status = 0
Calculation + with pid 7353 reads 5 and 7 from pipe
Calculation + with pid 7353 sends the result 12 (5 + 7) to pipe
My PID = 7352: Child PID = 7353 terminated normally, exit status = 0
Calculation * with pid 7352 reads 12 and 4 from pipe
Calculation * with pid 7352 sends the result 48 (12 * 4) to pipe
My PID = 7350: Child PID = 7352 terminated normally, exit status = 0
Calculation + with pid 7350 reads 10 and 48 from pipe
Calculation + with pid 7350 sends the result 58 (10 + 48) to pipe
My PID = 7349: Child PID = 7350 terminated normally, exit status = 0
The result is 58!
```

➢ Ερωτήσεις της 2.4

1. Στην συγκεκριμένη άσκηση υλοποιήσαμε μία σωλήνωση ανά διεργασία, δηλαδή ο κάθε πατέρας έχει 2 σωληνώσεις συνολικά. Στις πράξεις της πρόσθεσης και του πολλαπλασιασμού που ισχύει η αντιμεταθετικότητα των πράξεων, επομένως δεν μας νοιάζει με ποιά σειρά θα διαβαστούν τα δεδομένα, μπορεί να υπάρξει μία σωλήνωση για όλες τις διεργασίες παιδιά. Κάτι τέτοιο φυσικά δεν θα μπορούσε να ισχύσει σε περίπτωση που έχουμε τις πράξεις της αφαίρεσης και της διαίρεσης όπου δεν χαρακτηρίζονται από προσεταιριστικότητα καθώς τότε ο πατέρας δεν θα ήταν σε θέση να γνωρίζει ποιό παιδί του έστειλε τι.

2. Σε ένα τέτοιο σύστημα η παράλληλη επεξεργασία διεργασιών επιφέρει αισθητή βελτίωση στον χρόνο επεξεργασίας των διεργασιών λόγω της παράλληλης εκτέλεσης των εντολών που δεν εξαρτώνται μεταξύ τους. Σε αντίθετη περίπτωση, αν χρησιμοποιούμε μία μόνο διεργασία τότε είμαστε αναγκασμένοι να εκτελούνται όλες οι εντολές σειριακά κάτι το οποίο οδηγεί στην αύξηση του συνολικού χρόνου.