National Technical University of Athens

School of Electrical and Computer Engineering

Λειτουργικά Συστήματα Υπολογιστών (Τμήμα 1)

Άσκηση 2: *Διαχείριση Διεργασιών και Διαδεργασιακή Επικοινωνία*

ΧΡΙΣΤΟΦΟΡΟΣ ΒΑΡΔΑΚΗΣ (03118883)

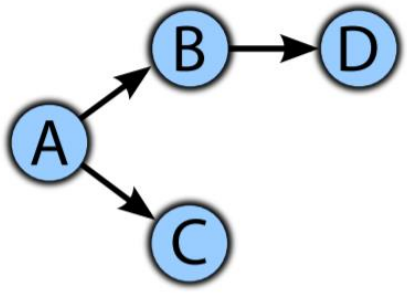
ΓΕΩΡΓΙΟΣ ΛΥΜΠΕΡΑΚΗΣ (03118881)

oslaba69

16 Απριλίου 2021

Άσκηση 1.1:

Στην πρώτη άσκηση καλούμαστε να δημιουργήσαμε το παρακάτω δέντρο διεργασιών.



Σε πρώτη φάση παρουσιάζουμε τον κώδικα που γράφτηκε για την πρώτη άσκηση.

Κώδικας Αρχείου main.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)
{
	pid_t b,c,d;
	int status;
	// Create B Child
	b = fork();
	if (b < 0) {
		perror("main: fork");
		exit(1);
	}
	if (b == 0) {
		//Create D Child
		d = fork();
		if (d < 0) {//error check
			perror("main: fork");
			exit(1);
		}
```

```
            // Body of B
            change_pname("B");
            sleep(SLEEP_PROC_SEC);
            d = wait(& status);
                  explain_wait_status(d, status);
            printf("B: Exiting...\n");
            exit(19);
      }
      // Create C Child
      c = fork();
      if (c < 0) {
            perror("main: fork");
            exit(1);
      }
      if (c == 0) {
            //Body of C
            change_pname("C");
            printf("C: Sleeping...\n");
            sleep(SLEEP_PROC_SEC);
            printf("C: Exiting...\n");
            exit(17);
      }
      //Body of A
      change_pname("A");
      printf("A: Sleeping...\n");
      sleep(SLEEP_PROC_SEC);
      c = wait(& status);
        explain_wait_status(c, status);
        d = wait(& status);
        explain_wait_status(d, status);
      printf("A: 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);
      }

      sleep(SLEEP_TREE_SEC);

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

      //show_pstree(getpid());//for question 2

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

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

      return 0;
}
```

Επίσης το Makefile που δημιουργήσαμε είναι το κάτωθεν

Makefile

```
#Compiler
cc = gcc

#Compiler Flags
CFLAGS = -Wall

#Targets
TARGET = fork clean

all: $(TARGET)

fork: main.o proc-common.o
	$(CC) $(CFLAGS) main.o proc-common.o -o fork

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

proc-common.o: proc-common.c
	$(CC) $(CFLAGS) proc-common.c -c
clean:
	rm main.o proc-common.o
```

Ενώ ,η εκτέλεση του προγράμματος έδωσε την παρακάτω έξοδο

Τερματικό

```
A: Sleeping...
D: Sleeping...
C: Sleeping...


A(19334)─┬─B(19335)──D(19337)
         └─C(19336)


D: Exiting...
C: Exiting...
My PID = 19335: Child PID = 19337 terminated normally, exit status = 13
B: Exiting...
My PID = 19334: Child PID = 19336 terminated normally, exit status = 17
My PID = 19334: Child PID = 19335 terminated normally, exit status = 19
A: Exiting...
My PID = 19333: Child PID = 19334 terminated normally, exit status = 16
```

1.

Βλέπουμε ότι τερματίζοντας την διεργασία Α ενώ το πρόγραμμα τερματίζει αμέσως οι διεργασίες παιδιά του πεθαίνουν κανονικά μετά από το χρονικό διάστημα που έχουμε ορίσει. Αυτό συμβαίνει επειδή όταν τερματιστεί η διεργασία-πατέρας οι διεργασίες παιδιά μεταφέρονται ως παιδιά της init. Αυτό μπορούμε να το εξετάσουμε τρέχοντας την εντολή pstree σε άλλο terminal αφού κάνουμε kill τον Α αλλά πριν τερματίσουν οι διεργασίες παιδιά (προφανώς χεριάζετε τροποποίηση του χρόνου ζωής τους).

2.

Πέραν των διεργασιών A - D εμφανίζεται η διεργασία fork που είναι το πρόγραμμα μας και παιδί του οποίου είναι η Α, καθώς και οι διεργασίες sh και pstree που χρησιμοποιούνται από το πρόγραμμα μας για την εκτύπωση του δέντρου διεργασιών μέσω της show_pstree (το πρόγραμμα καλεί το sh-φλοιός το οποίο με τη σειρά του καλεί την pstree).

3.

Σε περιπτώσεις συνύπαρξης πολλών χρηστών στο ίδιο υπολογιστικό σύστημα τότε ο διαχειριστής επιλέγει ένα άνω όριο για το πλήθος των διεργασιών ανά χρήστη. Ο λόγος που επιλέγεται μία τέτοια κίνηση είναι αρχικά η ασφάλεια, δηλαδή σε περίπτωση που ένας χρήστης είναι κακόβουλος και για παράδειγμα δημιουργεί αμέτρητες διεργασίες που καταλαμβάνουν αμέτρητο χώρο. Επομένως, με το όριο διεργασιών περιορίζεται η δράση ενός κακόβουλου κώδικα. Επιπρόσθετα , το υπολογιστικό σύστημα έχει πεπερασμένες δυνατότητες ως προς τους πόρους και τις επιδόσεις του. Πιο συγκεκριμένα, είναι θέμα δικαιοσύνης ένας χρήστης να το αξιοποιεί πολύ περισσότερο το υπολογιστικό σύστημα από άλλους χρήστες που θέλουν και αυτοί να εξυπηρετηθούν.

Άσκηση 1.2:

Στην τρέχων άσκηση θα δημιουργήσουμε μια αναδρομική συνάρτηση η οποία θα δημιουργεί ένα αυθαίρετο δέντρο διεργασιών.

Κώδικας Αρχείου main.c

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

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

#define SLEEP_LEAF 15
#define SLEEP_TREE_SEC 6

void print_children(struct tree_node *p){ //Print the children names of  a node

    printf("All my Children are { ");
    unsigned int i ;

    for(i = 0; i < p->nr_children; ++i){
           printf(" %s  ",(p->children + i) -> name);

    if (i< p->nr_children -1){
         printf(",");
    }
  }
       printf("} \n");
}

void Make_Tree_Proc(struct tree_node *root){
    //declaretions
    pid_t p;
    int i,status ;

    change_pname(root -> name);  //name the process
 //   printf("I am %s and I am alive !!\n",root->name);

    for(i = 0; i < root -> nr_children; ++i){ //visit every child of node

     //Say who you are and how many kids do you have
//     printf("I am %s and have [ %d ] children to make , i.e. ",root -> name,root-
>nr_children - i);
    // print_children(root);

     // call fork
     p = fork();

     //Check if there is any problem with fork implementation
     if (p < 0) {
        perror("fork");
        exit(-1);
     }
```

```
    //Code for the child process
     if ( p == 0 ) {

      //change_pname((root -> children + i) -> name);
        printf("I am %s and I am alive ! \n",(root->children + i)->name);
         if ((root -> children + i)->nr_children == 0){  //Are you a leaf?
            // printf("I am the %s Process and I am a Leaf\n", (root-> children + i)-
>name);  //say that you are a leaf!
            change_pname((root->children+i)->name);

            // printf("Process %s : I am alive !\n",(root-> children+ i)->name); //Say when
you are alive
             sleep(SLEEP_LEAF);                                          //rest
             printf("I am %s and I am dying !\n",(root-> children + i)->name); //Say when
you are done
             exit(1);
         }

         else { //Then you are a median node and call recursive the function for your next
children
           // printf("I am the %s Process and I am a Median node \n",(root->children+i)-
>name);
          Make_Tree_Proc(root -> children + i);
        }
     }
}

     for(i=0; i < root -> nr_children; ++i){
        // printf("Loop: %s hi \n",root->name);
       p = wait(&status);
       explain_wait_status(p,status);
  //     printf("Hi %s \n",root->name);
      }
     printf("I am %s and I am dying !\n",root->name);
     exit(1); //Exit the middle node
}
```

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

      if (argc != 2){ //Check if the argument is not there
            fprintf(stderr, "Usage: %s<input_tree_file>\n\n", argv[0]);
            exit(1);
      }

      printf("\n");
      printf("The fork Tree is the above:\n\n");
      root = get_tree_from_file(argv[1]);
      print_tree(root);

      printf("\n\n ---------------  Process Tree Generation  ----------------- \n\n");

      pid_t pid = fork(); //generate the first process in order to start

      if (pid < 0) //Check if there is a problem
      {
            perror("fork");
            exit(-1);
      }
        else if ( pid == 0) { //What to do the process child
           printf("I am %s and I am alive ! \n",root->name);
         Make_Tree_Proc(root);
           exit(1);
        }

    //code for the first process

    sleep(SLEEP_TREE_SEC);

    show_pstree(pid); //print process tree

    pid = wait(& status); //wait father
    explain_wait_status(pid, status); //


      return 0;
}
```

Επίσης το Makefile που δημιουργήσαμε είναι το κάτωθεν

Makefile

```
#Compiler
cc = gcc

#Compiler Flags
CFLAGS = -Wall

#Targets
TARGET = ex12 clean

all: $(TARGET)

ex12: ex12.o proc-common.o tree.o
	$(CC) $(CFLAGS) ex12.o proc-common.o tree.o  -o ex12

ex12.o: ex12.c
	$(CC) $(CFLAGS) ex12.c -c

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

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

clean:
	rm ex12.o proc-common.o tree.o
```

Ενώ ,η μεταγλώττιση και η εκτέλεση του προγράμματος φαίνεται παρακάτω

Τερματικό

```
The fork Tree is the above:

A
        B
                E
                F
        C
                H
                G
        D


 ---------------  Process Tree Generation  -----------------

I am A and I am alive !
I am B and I am alive !
I am C and I am alive !
I am E and I am alive !
I am H and I am alive !
I am D and I am alive !
I am F and I am alive !
I am G and I am alive !


A(27960)──┬─B(27961)──┬─E(27964)
          │           └─F(27965)
          ├─C(27962)──┬─G(27967)
          │           └─H(27966)
          └─D(27963)


I am E and I am dying !
I am H and I am dying !
My PID = 27961: Child PID = 27964 terminated normally, exit status = 1
My PID = 27962: Child PID = 27966 terminated normally, exit status = 1
I am D and I am dying !
I am F and I am dying !
I am G and I am dying !
My PID = 27960: Child PID = 27963 terminated normally, exit status = 1
My PID = 27961: Child PID = 27965 terminated normally, exit status = 1
My PID = 27962: Child PID = 27967 terminated normally, exit status = 1
I am B and I am dying !
I am C and I am dying !
My PID = 27960: Child PID = 27961 terminated normally, exit status = 1
My PID = 27960: Child PID = 27962 terminated normally, exit status = 1
I am A and I am dying !
My PID = 27959: Child PID = 27960 terminated normally, exit status = 1
```

Τερματικό

```
The fork Tree is the above:

A
        B
                D
                        E
                                F
                                        G
        C
                H
                        I
                                J
                                        K


 ---------------  Process Tree Generation  -----------------

I am A and I am alive !
I am B and I am alive !
I am C and I am alive !
I am D and I am alive !
I am H and I am alive !
I am I and I am alive !
I am J and I am alive !
I am K and I am alive !
I am E and I am alive !
I am F and I am alive !
I am G and I am alive !
```

```
A(19527)─┬─B(19528)──D(19530)──E(19532)──F(19536)──G(19537)
         └─C(19529)──H(19531)──I(19533)──J(19534)──K(19535)


I am K and I am dying !
My PID = 19534: Child PID = 19535 terminated normally, exit status = 1
I am J and I am dying !
My PID = 19533: Child PID = 19534 terminated normally, exit status = 1
I am I and I am dying !
My PID = 19531: Child PID = 19533 terminated normally, exit status = 1
I am H and I am dying !
My PID = 19529: Child PID = 19531 terminated normally, exit status = 1
I am C and I am dying !
My PID = 19527: Child PID = 19529 terminated normally, exit status = 1
I am G and I am dying !
My PID = 19536: Child PID = 19537 terminated normally, exit status = 1
I am F and I am dying !
My PID = 19532: Child PID = 19536 terminated normally, exit status = 1
I am E and I am dying !
My PID = 19530: Child PID = 19532 terminated normally, exit status = 1
I am D and I am dying !
My PID = 19528: Child PID = 19530 terminated normally, exit status = 1
I am B and I am dying !
My PID = 19527: Child PID = 19528 terminated normally, exit status = 1
I am A and I am dying !
My PID = 19526: Child PID = 19527 terminated normally, exit status = 1
```

1.

Όπως βλέπουμε και παραπάνω κατά τη δημιουργία των διεργασιών πρώτα εμφανίζονται τα μηνύματα των πατεράδων και έπειτα τα μηνύματα των παιδιών τους και ούτε κάθε εξής. Οπότε , παρατηρούμε αυτή την «ιεραρχική» εκτύπωση των μηνυμάτων όμως αξίζει να αναφερθεί ότι λόγω της τυχαιότητας ως προς το ποιο παιδί κάθε επιπέδου θα εκτελεστεί πρώτο με επανειλημμένη εκτέλεση του ίδιου προγράμματος βλέπουμε και διαφορετική σειρά εμφάνιση των παιδιών . Αυτό οφείλεται στην κλήση συστήματος ***fork()*** η οποία δεν μας διαβεβαιώνει για τη σειρά των διεργασιών που θα καταλάβουν την ΚΜΕ.

Κατά την διαδικασία τερματισμού πρώτα τερματίζουν οι διαδικασίες παιδιά και έπειτα παίρνουν σειρά οι πατρικές όπως παρατηρείται και από τον κώδικα της συνάρτησης που δημιουργεί το εκάστοτε δέντρο διεργασιών .

Άσκηση 1.3:

Στην τρέχουσα άσκηση επεκτείνουμε το προηγούμενο πρόγραμμα κατά τέτοιο τρόπο ώστε οι διεργασίες που δημιουργούνται να ελέγχονται με την χρήση σημάτων . Ο στόχος μας είναι οι διεργασίες να εκτυπώνουν τα μηνύματα τους κατά βάθος.

Επομένως ,στις επόμενες δύο σελίδες περιλαμβάνεται ο κώδικας που υλοποιεί το παραπάνω ζητούμενο καθώς και το Makefile και μία ενδεικτική έξοδο του προγράμματος .

Κώδικας Αρχείου main.c

```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"

//Inter-Process Communication

typedef struct tree_node* tree;

void signal_handler_cont(int sig){
        printf("Received Signal to continue %d ,PID:  %d  \n",sig,getpid() );
}


void fork_procs(tree root)
{
        pid_t * pid_array;
        pid_t p;
        int  status;
        unsigned int i ;

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

        unsigned int Num_Chil = root -> nr_children;

        if(Num_Chil) {
                pid_array = malloc(sizeof(pid_t) * Num_Chil);
                if (pid_array == NULL){
                        perror("Problem with Malloc");
                        exit(-1);
                }
        }


        for(i = 0; i < Num_Chil; ++i){

                pid_array[i] = fork();

                if (pid_array[i] < 0){ //check for a problem
                 perror("child process was unsuccessful");
                 exit(-1);
                }

                if (pid_array[i]== 0){ //code for the child-process

            change_pname((root -> children + i) -> name); //name the process and then print that you are going to sleep
            printf("PID=%ld ,name = %s is created and going to sleep for now\n",(long) getpid(), (root->children + i)->name);

            signal(SIGCONT,signal_handler_cont); //learn about your signal handler
            fork_procs(root->children + i); //call recursive the function for the next child

           }


        }

        if (Num_Chil != 0){
         wait_for_ready_children(Num_Chil); //wait for all your children to be created
        }

        raise(SIGSTOP);  //equiavelent to kill(get_pid(),SIGSTOP) //stop yourself

        printf("PID : %d with name %s , is awake\n",getpid(),root->name); //say that you have are alive

        for(i = 0; i < Num_Chil ; ++i){
                kill(pid_array[i],SIGCONT); //awake all your chidlren

        p = wait(&status);
        explain_wait_status(p, status); //check how every child has been killed

        }

        printf("PID : %d with name %s , is Exiting\n",getpid(),root->name); //say that the end is near

        exit(0);
}
```

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

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

	if (argc < 2){
		fprintf(stderr, "Usage: %s <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);
	explain_wait_status(pid, status);

	return 0;
}
```

Makefile

```
#Compiler
CC  = gcc

#Compiler Flags
CFLAGS = -Wall

#Targets
TARGET = ex21 clean

all: $(TARGET)

ex21: ex21.o proc-common.o tree.o
	$(CC) $(CFLAGS) ex21.o proc-common.o tree.o -o ex21

ex21.o: ex21.c
	$(CC) $(CFLAGS) ex21.c -c

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

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

clean:
	rm ex21.o proc-common.o tree.o
```

Τερματικό

```
PID=28088 ,name = B is created and going to sleep for now
PID=28089 ,name = C is created and going to sleep for now
PID=28092 ,name = H is created and going to sleep for now
PID=28094 ,name = G is created and going to sleep for now
My PID = 28089: Child PID = 28092 has been stopped by a signal, signo = 19
PID=28090 ,name = D is created and going to sleep for now
My PID = 28087: Child PID = 28090 has been stopped by a signal, signo = 19
My PID = 28089: Child PID = 28094 has been stopped by a signal, signo = 19
My PID = 28087: Child PID = 28089 has been stopped by a signal, signo = 19
PID=28093 ,name = F is created and going to sleep for now
My PID = 28088: Child PID = 28093 has been stopped by a signal, signo = 19
PID=28091 ,name = E is created and going to sleep for now
My PID = 28088: Child PID = 28091 has been stopped by a signal, signo = 19
My PID = 28087: Child PID = 28088 has been stopped by a signal, signo = 19
My PID = 28086: Child PID = 28087 has been stopped by a signal, signo = 19


A(28087)─┬─B(28088)─┬─E(28091)
         │          └─F(28093)
         ├─C(28089)─┬─G(28094)
         │          └─H(28092)
         └─D(28090)

PID : 28087 with name A , is awake
Received Signal to continue 18 ,PID:  28088
PID : 28088 with name B , is awake
Received Signal to continue 18 ,PID:  28091
PID : 28091 with name E , is awake
PID : 28091 with name E , is Exiting
My PID = 28088: Child PID = 28091 terminated normally, exit status = 0
Received Signal to continue 18 ,PID:  28093
PID : 28093 with name F , is awake
PID : 28093 with name F , is Exiting
My PID = 28088: Child PID = 28093 terminated normally, exit status = 0
PID : 28088 with name B , is Exiting
My PID = 28087: Child PID = 28088 terminated normally, exit status = 0
Received Signal to continue 18 ,PID:  28089
PID : 28089 with name C , is awake
Received Signal to continue 18 ,PID:  28092
PID : 28092 with name H , is awake
PID : 28092 with name H , is Exiting
My PID = 28089: Child PID = 28092 terminated normally, exit status = 0
Received Signal to continue 18 ,PID:  28094
PID : 28094 with name G , is awake
PID : 28094 with name G , is Exiting
My PID = 28089: Child PID = 28094 terminated normally, exit status = 0
PID : 28089 with name C , is Exiting
My PID = 28087: Child PID = 28089 terminated normally, exit status = 0
Received Signal to continue 18 ,PID:  28090
PID : 28090 with name D , is awake
PID : 28090 with name D , is Exiting
My PID = 28087: Child PID = 28090 terminated normally, exit status = 0
PID : 28087 with name A , is Exiting
My PID = 28086: Child PID = 28087 terminated normally, exit status = 0
```

Τερματικό

```
PID=19696 ,name = B is created and going to sleep for now
PID=19697 ,name = C is created and going to sleep for now
PID=19698 ,name = D is created and going to sleep for now
PID=19699 ,name = H is created and going to sleep for now
PID=19700 ,name = E is created and going to sleep for now
PID=19701 ,name = I is created and going to sleep for now
PID=19702 ,name = F is created and going to sleep for now
PID=19703 ,name = J is created and going to sleep for now
PID=19704 ,name = G is created and going to sleep for now
My PID = 19702: Child PID = 19704 has been stopped by a signal, signo = 19
My PID = 19700: Child PID = 19702 has been stopped by a signal, signo = 19
My PID = 19698: Child PID = 19700 has been stopped by a signal, signo = 19
PID=19705 ,name = K is created and going to sleep for now
My PID = 19696: Child PID = 19698 has been stopped by a signal, signo = 19
My PID = 19703: Child PID = 19705 has been stopped by a signal, signo = 19
My PID = 19701: Child PID = 19703 has been stopped by a signal, signo = 19
My PID = 19695: Child PID = 19696 has been stopped by a signal, signo = 19
My PID = 19699: Child PID = 19701 has been stopped by a signal, signo = 19
My PID = 19697: Child PID = 19699 has been stopped by a signal, signo = 19
My PID = 19695: Child PID = 19697 has been stopped by a signal, signo = 19
My PID = 19694: Child PID = 19695 has been stopped by a signal, signo = 19


A(19695)─┬─B(19696)──D(19698)──E(19700)──F(19702)──G(19704)
         └─C(19697)──H(19699)──I(19701)──J(19703)──K(19705)
```

```
PID : 19695 with name A , is awake
Received Signal to continue 18 ,PID:  19696
PID : 19696 with name B , is awake
Received Signal to continue 18 ,PID:  19698
PID : 19698 with name D , is awake
Received Signal to continue 18 ,PID:  19700
PID : 19700 with name E , is awake
Received Signal to continue 18 ,PID:  19702
PID : 19702 with name F , is awake
Received Signal to continue 18 ,PID:  19704
PID : 19704 with name G , is awake
PID : 19704 with name G , is Exiting
My PID = 19702: Child PID = 19704 terminated normally, exit status = 0
PID : 19702 with name F , is Exiting
My PID = 19700: Child PID = 19702 terminated normally, exit status = 0
PID : 19700 with name E , is Exiting
My PID = 19698: Child PID = 19700 terminated normally, exit status = 0
PID : 19698 with name D , is Exiting
My PID = 19696: Child PID = 19698 terminated normally, exit status = 0
PID : 19696 with name B , is Exiting
My PID = 19695: Child PID = 19696 terminated normally, exit status = 0
Received Signal to continue 18 ,PID:  19697
PID : 19697 with name C , is awake
Received Signal to continue 18 ,PID:  19699
PID : 19699 with name H , is awake
Received Signal to continue 18 ,PID:  19701
PID : 19701 with name I , is awake
Received Signal to continue 18 ,PID:  19703
PID : 19703 with name J , is awake
Received Signal to continue 18 ,PID:  19705
PID : 19705 with name K , is awake
PID : 19705 with name K , is Exiting
My PID = 19703: Child PID = 19705 terminated normally, exit status = 0
PID : 19703 with name J , is Exiting
My PID = 19701: Child PID = 19703 terminated normally, exit status = 0
PID : 19701 with name I , is Exiting
My PID = 19699: Child PID = 19701 terminated normally, exit status = 0
PID : 19699 with name H , is Exiting
My PID = 19697: Child PID = 19699 terminated normally, exit status = 0
PID : 19697 with name C , is Exiting
My PID = 19695: Child PID = 19697 terminated normally, exit status = 0
PID : 19695 with name A , is Exiting
My PID = 19694: Child PID = 19695 terminated normally, exit status = 0
```

1.

Στην προηγούμενη άσκηση ο χρόνος ζωής κάθε διεργασίας που γεννιότανε ήταν καθορισμένος από την παράμετρο που θα δίναμε στη συνάρτηση ***sleep()*** η οποία αναστέλλει για χρόνο ίσο με την τιμή της παραμέτρου την διαδικασία . Επομένως, δεν μπορούσαμε να ελέγξουμε την ακριβή διαδικασία δημιουργίας των διεργασιών πέρα από τον χρόνο που είναι ζωντανές.

Αντίθετα, η χρήση σημάτων μας επιτρέπει να δημιουργήσουμε μία ενδοεπικοινωνία ασύγχρονη μεταξύ των διεργασιών ώστε να ελέγχουμε μέσω της αποστολής κατάλληλων σημάτων το πότε μια διεργασία δημιουργείται και πότε ολοκληρώνεται.

2.

Η κλήση της συνάρτησης ***wait_for_children()*** εξασφαλίζει ότι η πατρική διεργασία θα συνεχίσει την εκτέλεση της εφόσον όλα της τα παιδιά – διεργασίες έχουν αναστείλει τη λειτουργία τους (έχει αλλάξει το status τους).

Σε περίπτωση παράλειψης χρήσης της παραπάνω συνάρτησης τότε η διεργασία – πατέρας όταν φτάσει στο σημείο να στείλει σήμα σε κάθε παιδί που έχει δημιουργήσει και περιμένει να συνεχίσει , υπάρχει η πιθανότητα/κίνδυνος το παιδί να μην έχει προλάβει να εκτελέσει την εντολή ***raise(SIGSTOP)*** με επακόλουθο το σήμα από την πατρική προς την γονική διαδικασία να αγνοηθεί.

Άσκηση 1.4:

Στην τελευταία άσκηση εφαρμόζουμε το δέντρο διεργασιών ώστε να υπολογίζουμε αριθμητικές παραστάσεις.

Κώδικας Αρχείου main.c

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

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



void Make_Tree_Proc(struct tree_node *root,int pipfd[2]){
    //declaretions
    int pip1[2];
    int pip2[2];
    int input1, input2;
    pid_t p1, p2;

    int i,status ;
    pipe (pip1);
    pipe (pip2);

    change_pname(root -> name);  //name the process
    int nr_of_children = root->nr_children;

    if (nr_of_children == 2){//middle node
        p1 = fork();
        if (p1 == 0){
            change_pname((root->children) -> name);  //name the process
            Make_Tree_Proc(root->children, pip1);
        }

        p2 = fork();

        if (p2 == 0){//child process
            change_pname((root->children+1) -> name);  //name the process
            Make_Tree_Proc(root->children+1, pip2);//recursive call
        }
       //read child 1 from pipe1

       if(read(pip1[0], &input1, sizeof(input1))!= sizeof(input1)){
              perror("read from pipe");
              exit(-1);
       }

        close(pip1[0]);//close reading end of pipe1

       //read child 2 from pie2
        if(read(pip2[0], &input2, sizeof(input2))!= sizeof(input2)){
              perror("read from pipe");
              exit(-1);
       }

        close(pip2[0]);//close reading end of pipe2
```

```
      int res;
      //calculate result
        if (!strcmp(root->name, "*")){
            res = input1*input2;
        }
        else if (!strcmp(root->name, "+")) {
            res = input1+input2;
        }
        else {//if given anything other than * or +
            printf("UNKNOWN OPERATOR\n");
            exit(-1);
        }
      //write result on writing end of father pipe
        if(write(pipfd[1],&res,sizeof(res))!= sizeof(res)){
            perror ("Write to pipe");
            exit (-1);
      }
      close(pipfd[1]); //close writing end of pipe fd

    }
    else {//leaf node
        int val = atoi(root->name);
      //write result on writing end of father pipe
        if(write(pipfd[1], &val, sizeof(val))!= sizeof(val)){
            perror("Write to pipe");
            exit(-1);
      }
      close(pipfd[1]);
        exit(1);//Exit Leaf Node
    }



   for(i=0; i < root -> nr_children; ++i){//wait for all your children to change status
         p1 = wait(&status);
         explain_wait_status(p1,status);
   }

   exit(1); //Exit the middle node
}
```

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

      if (argc != 2){ //Check if the argument is not there
            fprintf(stderr, "Usage: %s<input_tree_file>\n\n", argv[0]);
            exit(1);
      }
      printf("The fork Tree is the above:\n\n");
      root = get_tree_from_file(argv[1]);
      print_tree(root);

      printf("\n\n ---------------  Process Tree Generation  -----------------
\n\n");
    int pip[2];
    pipe(pip);
    int res;
      pid_t pid = fork(); //generate the first process in order to start

      if (pid < 0) //Check if there is a problem
      {
            perror("fork");
            exit(-1);
      }
         else if ( pid == 0) { //What to do the process child
             Make_Tree_Proc(root,pip);
             exit(17);
         }

    //code for the first process

    read(pip[0], &res, sizeof(res));
    close(pip[1]);

    //show_pstree(pid); //print process tree

    pid = wait(& status); //wait father
    explain_wait_status(pid, status); //
    printf("\n\nRESULT = %d\n\n",res);

      return 0;
}
```

Makefile

```
#Compiler
cc = gcc

#Compiler Flags
CFLAGS = -Wall

#Targets
TARGET =  optree clean

all: $(TARGET)

optree: main.o proc-common.o tree.o
	$(CC) $(CFLAGS) main.o proc-common.o tree.o -o optree

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

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

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

clean:
	rm main.o proc-common.o tree.o
```

Τερματικό

```
The fork Tree is the above:

+
        10
        *
                +
                        5
                        7
                4


 ---------------  Process Tree Generation  -----------------

My PID = 28148: Child PID = 28151 terminated normally, exit status = 1
My PID = 28147: Child PID = 28149 terminated normally, exit status = 1
My PID = 28145: Child PID = 28146 terminated normally, exit status = 1
My PID = 28148: Child PID = 28150 terminated normally, exit status = 1
My PID = 28147: Child PID = 28148 terminated normally, exit status = 1
My PID = 28145: Child PID = 28147 terminated normally, exit status = 1
My PID = 28144: Child PID = 28145 terminated normally, exit status = 1


RESULT = 58
```

Τερματικό

```
The fork Tree is the above:

*
       2
       +
               3
               *
                       4
                       +
                               5
                               *
                                       6
                                       +
                                               7
                                               8


 ---------------  Process Tree Generation  -----------------

My PID = 19949: Child PID = 19950 terminated normally, exit status = 1
My PID = 19947: Child PID = 19948 terminated normally, exit status = 1
My PID = 19945: Child PID = 19946 terminated normally, exit status = 1
My PID = 19943: Child PID = 19944 terminated normally, exit status = 1
My PID = 19949: Child PID = 19951 terminated normally, exit status = 1
My PID = 19941: Child PID = 19942 terminated normally, exit status = 1
My PID = 19939: Child PID = 19940 terminated normally, exit status = 1
My PID = 19947: Child PID = 19949 terminated normally, exit status = 1
My PID = 19945: Child PID = 19947 terminated normally, exit status = 1
My PID = 19943: Child PID = 19945 terminated normally, exit status = 1
My PID = 19941: Child PID = 19943 terminated normally, exit status = 1
My PID = 19939: Child PID = 19941 terminated normally, exit status = 1
My PID = 19938: Child PID = 19939 terminated normally, exit status = 1


RESULT = 766
```

1.

Στη συγκεκριμένη υλοποίηση χρησιμοποιούμε 2 σωληνώσεις ανά πράξη. Όμως θα μπορούσε να υλοποιηθεί εύκολα και με μία σωλήνωση, στην οποία κάθε παιδί θα έγραφε το αποτέλεσμά του. Απαραίτητη όμως προϋπόθεση για να γίνει αυτό είναι η αντιμεταθετή ιδιότητα των πράξεων, διότι η διεργασία πατέρας δεν γνωρίζει σε ποιο από τα δύο παιδιά αντιστοιχεί η απάντηση που διαβάζει από τη σωλήνωση. Για να υλοποιηθεί μία μη αντιμεταθετική πράξη θα έπρεπε το κάθε παιδί μαζί με τη πληροφορία να γράψει στη σωλήνωση και κάποιο αναγνωριστικό, για να αναγνωρίσει ο πατέρας σε ποιο παιδί αντιστοιχεί η απάντηση. Αυτό όμως περιπλέκει αρκετά τη συγκεκριμένη υλοποίηση και δεν υπάρχει κάποιο πρακτικό πλεονέκτημα έναντι της χρήσης δύο σωληνώσεων.

2.

Σε ένα σύστημα πολλαπλών πυρήνων η αποτίμηση από δέντρο διεργασιών υπερτερεί έναντι της αποτίμησης από μία μόνο διεργασία, διότι επιτρέπει στο σύστημα την παράλληλη εκτέλεση πράξεων που δεν είναι εξαρτημένες. Έτσι βελτιώνετε σημαντικά ο χρόνος εκτέλεσης. Πιο συγκεκριμένα, αν υποθέσουμε ένα πλήρες δυαδικό δέντρο τότε ένα μόνο-επεξεργαστικό σύστημα θα εκτελέσει πρώτα το δεξί και έπειτα το αριστερό υπό δέντρο. Αντίθετα, αν υποθέσουμε ότι σύστημα μας διαθέτει και δεύτερη ΚΜΕ τότε κάθε επεξεργαστής θα επισκεφτεί το κάθε υπό δέντρο και θα έχουμε αποτίμηση της παράστασης στο μισό χρόνο από όταν έχουμε μία και μοναδική ΚΜΕ.

Προαιρετικές Ερωτήσεις:

1.

Το πλεονέκτημα του παράλληλου υπολογισμού έναντι της σειριακής εκτέλεσης σε ένα σύστημα πολλαπλών επεξεργαστών είναι η δυνατότητα του να εκτελεί ταυτόχρονα περισσότερες από μία πράξεις με αποτέλεσμα επιτάχυνση του υπολογισμού. Για να είναι ταχύτερος ο υπολογισμός πρέπει να γίνονται όσο το δυνατό περισσότερες πράξεις παράλληλα. Ένα παράδειγμα της παραπάνω περίπτωσης είναι ο υπολογισμός πράξεων που σχηματίζουν ένα πλήρες δυαδικό δέντρο. Σε αυτή τη περίπτωση ο αριθμός των παράλληλων πράξεων είναι μέγιστος. Αντίθετη περίπτωση αποτελεί ένα εκφυλισμένο δέντρο , όπως το δέντρο της πράξης 2(3+(4(5+6*(7+8)))))

Σε αυτή την περίπτωση επειδή ο αριθμός των παράλληλων πράξεων είναι ελάχιστος, η απόδοση του παράλληλου υπολογισμού του δεν απέχει ουσιαστικά από τον σειριακό.

2.

Όπως είδαμε και στο προηγούμενο ερώτημα, η παράλληλη εκτέλεση περισσότερων πράξεων από τον αριθμό των επεξεργαστών του συστήματος δεν είναι αποδοτική. Το φαινόμενο αυτό γίνετε ιδιαίτερα εμφανές στα κατώτερα επίπεδα του δέντρου διεργασιών (αν αυτό δεν είναι εκφυλισμένο) όπου ο αριθμός των παράλληλων πράξεων είναι αυξημένος. Η υβριδική υλοποίηση επιτρέπει στο σύστημα να εκτελεί στο σημείο αυτό λιγότερες πράξεις παράλληλα, ελευθερώνοντας έτσι πόρους του συστήματος. Σημαντικός παράγοντας για την απόδοση της συγκεκριμένης υλοποίησης αποτελεί ο καθορισμός του βάθους μ κάτω από το οποίο οι πράξεις εκτελούνται σειριακά. Στην περίπτωση που το μ είναι πολύ μεγάλο, οι παράλληλες πράξεις ενδέχεται να μην μειωθούν επαρκώς ούτως ώστε να είναι διαχειρίσιμες από το σύστημα. Αντίθετα αν είναι πολύ μικρό οι πράξεις θα αρχίσουν να εκτελούνται σειριακά ενώ το σύστημα θα μπορούσε να τις εκτελέσει παράλληλα, γεγονός που επηρεάζει όπως είδαμε τον χρόνο εκτέλεσης. Έτσι συμπεραίνουμε ότι η βέλτιστη τιμή του βάθους μ είναι συνάρτηση του αριθμού των επεξεργαστών του συστήματος και του σχήματος του δέντρου διεργασιών.