ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΑΣΚΗΣΗ 4: ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗ

Φοιτητές

Αθανασίου Ιωάννης , Α.Μ.:03117041

Καραβαγγέλης Αθανάσιος , Α.Μ.:03117022

ΜΕΡΟΣ 1ο

Source code:

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>

#include <sys/wait.h>
#include <sys/types.h>

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


/* Compile-time parameters. */
#define SCHED_TQ_SEC 2                    /* time quantum */
#define TASK_NAME_SZ 60                    /* maximum size for a task's name */


// to next afora thn oura, deikths sto struct ths epomenhs diergasias pou tha xronodromologhthei

struct process {

        pid_t mypid;
        struct process *next;
        char *name;

};



/*Definition of my list nodes*/

struct process *head=NULL;
struct process *newnode=NULL;
struct process *last=NULL;



// o SIGALRM handler kaleitai gia otan teleiwnei to kvanto xronou, afou tote o xronodromologhths stelnei
SIGALRM, ara tote prepei na skotwsoume thn diergasia pou trexei twra
// dhladh thn diergasia pou einai sto head
```

```
static void sigalrm_handler(int signum) {
        kill(head->mypid,SIGSTOP);
}


// o SIGCHLD handler kaleitai gia otan fernoume allh diergasia sto head ths ouras, dhl allh diergasia gia ektelesh

static void sigchld_handler(int signum) {

        pid_t p;
        int status;

        while( 1 ) {

                //perimene opoiodhpote paidi
                p=waitpid(-1,&status,WNOHANG|WUNTRACED);

                //error sthn waitpid
                if (p<0){
                        perror("waitpid");
                        exit(1);
                }

                //no child has changed its state
                if(p==0) break;            /////////

                //dhladh ean to p>0 pou shmainei oti h waitpid ekane return to pid tou child whose state has
chenged
                explain_wait_status(p,status);

                //me vash to status thanatou ths p, pairnoume dyo kyries periptwseis


                if ( WIFEXITED(status) | WIFSIGNALED(status) )    {                  //if finished REMOVE it from the list

                        struct process * temp;
                        struct process* current;
                        temp=head;


                        while (temp != NULL) {
                                if (temp->mypid==p && temp==head) {                        //if temp == head -> got to delete
the head

                                        if (temp->next == NULL){               //an yparxei mono to head sthn oura
MESW AUTOU TOU SHMEIOU THA GINEI H EJODOS APO TO PROGRAMMA
                                                free(temp);
                                                printf("I am done\n");     //telos h douleia
                                                exit(0);
                                        }

                                        else{
                                                head=temp->next;       //an yparxoyn ki alla, head = to pisw tou
                                                free(temp);
                                        }
                                }


                                else if (temp->mypid==p && temp==last) {                //if TEMP == last -> apla to
diagrafoume kanontas to NULL kai auto kai to epomeno tou
                                        current = head;

                                        while (current->next->next != NULL) {
                                                current = current->next;
                                        }
```

```
                    free( temp );
                    last = current;
                    last->next = NULL;
                }

                else if (temp->mypid==p) {                          //if temp == random process of
the queue

                    current = head;

                    while (current->next != temp) {
                        current = current->next;
                    }

                    current->next = temp->next;
                    temp->next = NULL;
                    free (temp);

                }

                else {
                    temp = temp -> next;
                    continue;
                }

            }
        }

        if ( WIFSTOPPED(status) ) {             // if stopped by the scheduler, SIGALRM ->
sigalrm_hanlder() -> SIGSTOP sent to the head
            // (only the head is being stopped by the scheduler)
            // bring the head->next to the head
// move the head to the last->next
            last->next=head;
            last=head;

            struct process * temp;

            temp=head;
            head=head->next;                        // to head na ginei to epomeno tou

            temp->next=NULL;                // to palio head = temp, na exei next = null

        }

        alarm(SCHED_TQ_SEC);                            //set the alarm-counter to the time quantum
        //printf("%s\n",running->name);
        kill(head->mypid,SIGCONT);                      // so we send a SIGCONT to the head procces, so
that it continues from its pause
        // and after the alarm time has passed, is stops again because of the sigalrm_handler(int signum)

    }

}


static void install_signal_handlers(void) {

    sigset_t sigset;
    struct sigaction sa;

    sa.sa_handler = sigchld_handler;
    sa.sa_flags = SA_RESTART;
```

```
        // we make the set of the flags that will be in the mask
        // os the sigaction struct

        sigemptyset(&sigset);                           // make it empty
        sigaddset(&sigset, SIGCHLD);                    // add SIGCHLD, SIGALRM
        sigaddset(&sigset, SIGALRM);

        sa.sa_mask = sigset;

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

        sa.sa_handler = sigalrm_handler;
        if (sigaction(SIGALRM, &sa, NULL) < 0) {
                perror("sigaction: sigalrm");
                exit(1);
        }

        /*
         * Ignore SIGPIPE, so that write()s to pipes
         * with no reader do not result in us being killed,
         * and write() returns EPIPE instead.
         */
        if (signal(SIGPIPE, SIG_IGN) < 0) {
                perror("signal: sigpipe");
                exit(1);
        }
}



int main(int argc, char *argv[]) {

        int nproc;          // number of processes that we will organise

        /*
         * For each of argv[1] to argv[argc - 1],
         * create a new child process, add it to the process list.
         */

        char executable[] = "prog";                                     // name of the executable which will replace
the process
        char *newargv[] = { executable, NULL, NULL, NULL };         // pointer to the table with the executable's
arguments (must end with NULL)
        char *newenviron[] = { NULL };                                  // pointer to environment variables


        nproc = argc-1;                                                   // number of proccesses

        if (nproc == 0) {
                fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                exit(1);
        }

        pid_t mypid;
        int i;

        /*Fork and create the list*/
        for (i=0;i<nproc;i++) {

                mypid = fork();                                                         // create the process for the
current executable
```

```
        if (mypid<0){                                                       // fork error
            printf("Error with forks\n");
        }

        if(mypid==0){                                                        // code of the child
process

            raise(SIGSTOP);                                                  // pauses, waits for the
parent to send the SIGCONT

            printf("I am %s, PID = %ld\n", argv[0], (long)getpid());

            printf("About to replace myself with the executable %s...\n", executable);

            sleep(2);

            execve(executable, newargv, newenviron);                        // replace current procces
with the executable using arguments and environment variables
            //

            /* because execve() only returns on error */
            perror("execve");
            exit(1);

        }

        else{                                                                    // parent's code
after child's fork


            if (i==0){                                                           // i=0 -> first process
was just forked, so it's the head

                head = (struct process *) malloc(sizeof(struct process));     // create space for its struct
                if (head==NULL) printf("Error with malloc\n");

                head->mypid=mypid;                                               // this process is
the head and the last process of the list-queue
                head->next=NULL;
                head->name=argv[i+1];                            // name of the process = name of the
exec

                last=head;

            }

            else{                                                                // no first process
was just made by the scheduler process

                newnode=(struct process *) malloc(sizeof(struct process));      // make space for its struct
using malloc
                if (newnode==NULL) printf("Error with malloc\n");

                newnode->mypid=mypid;
                newnode->next=NULL;
                newnode->name=argv[i+1];

                last->next=newnode;                        // connect it to the end of the list-queue
                last=newnode;                               // make it the last process of the list-queue

            }

        }
        }

        // after all the nproc processes have been forked, the father-scheduler waits for them

        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(nproc);

        /* Install SIGALRM and SIGCHLD handlers. */
        install_signal_handlers();

        /*Set the alarm on*/
        alarm(SCHED_TQ_SEC);

        /*Start the first process*/
        kill(head->mypid,SIGCONT);

        // loop forever    until we exit from inside a signal handler. -> from line 83
        while (pause())
                ;

        /* Unreachable */
        fprintf(stderr, "Internal error: Reached unreachable point\n");
        return 1;
}
```

Γενικά:

Ο χρονοδρομολογητής αφού ρυθμίσει κάθε φορά ποια διεργασία θα μπει στο head της λίστας-ουράς και άρα θα ξεκινήσει να εκτελείται, χρησιμοποιεί την κλήση συστήματος alarm() ώστε να αρχίσει η αντίστροφη μέτρηση του κβάντου χρόνου για την εκτέλεση αυτής της διεργασίας. Όταν εκπνεύσει το κβάντο χρόνου αυτό, τότε η υπό-εκτέλεση διεργασία δέχεται το σήμα SIGALRM. Ενεργοποιείται ο αντίστοιχος χειριστής

```
sigalrm_handler(int signum)
```

ο οποίος στέλνει στην διεργασία κεφαλή το σήμα SIGSTOP. Η διεργασία παιδί στέλνει το σήμα SIGCHLD στον πατέρα της (scheduler) οπότε ενεργοποιείται ο χειριστής .

```
sigchld_handler(int signum)
```

Αυτός, αφού εξετάσει αν η διεργασία διακόπηκε με σήμα SIGSTOP ή με σήμα SIGKILL μέσω ειδικών σημαιών

```
if (WIFEXITED(status) || WIFSIGNALED(status))

if (WIFSTOPPED(status))
```

Ρυθμίζει κατάλληλα την ουρά των διεργασιών, ξεκινάει τον μετρητή

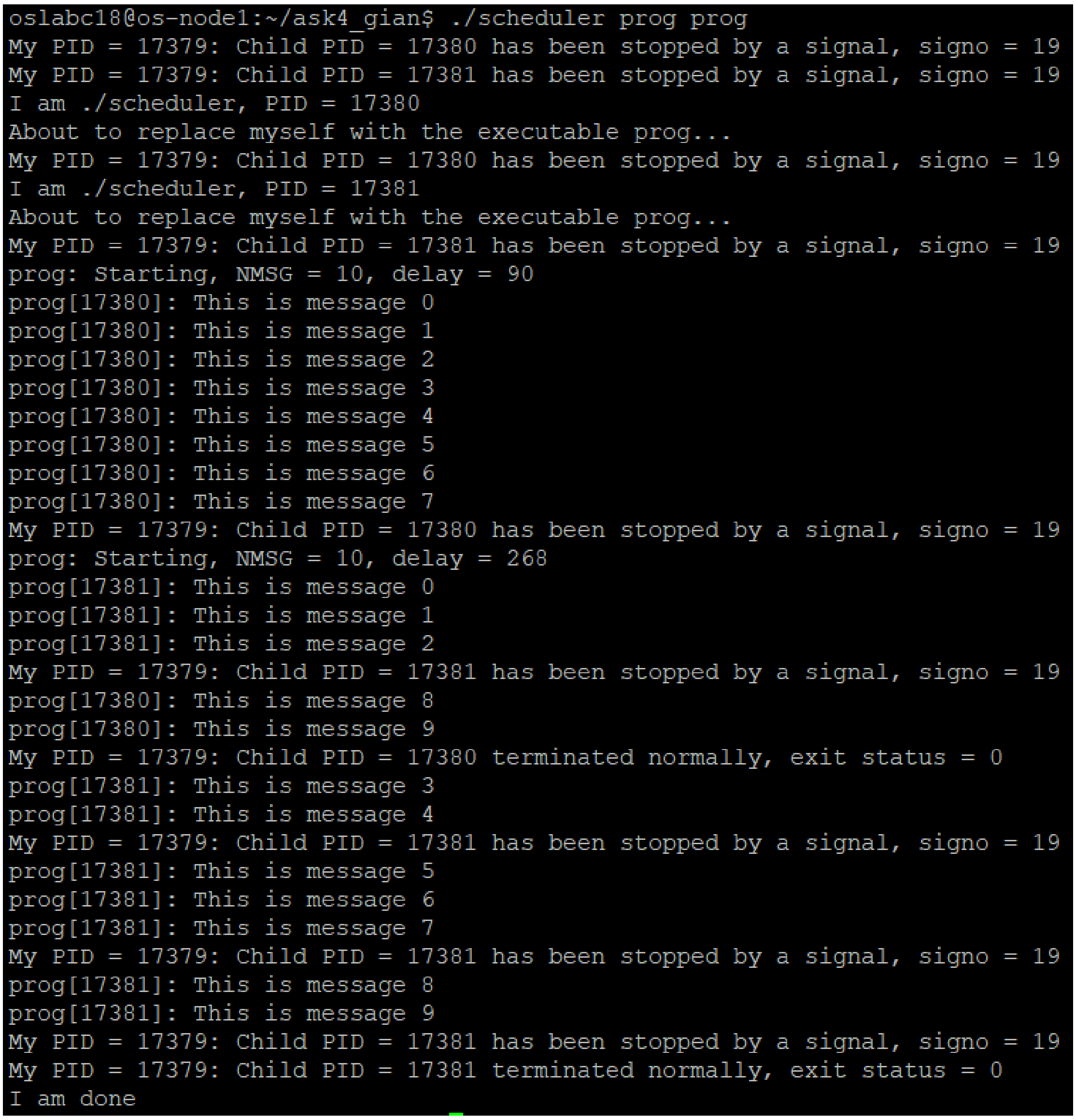
```
alarm(SCHED_TQ_SEC);
```

Και δίνει το σήμα SIGCONT στην διεργασία στο head

```
   kill(head->mypid,SIGCONT);
```

Η διαδικασία αυτή επαναλαμβάνεται μέχρι να αδειάσει η ουρά.

Ένα παράδειγμα εκτέλεσης του προγράμματός μας



```
oslabc18@os-node1:~/ask4_gian$ ./scheduler prog prog
My PID = 17379: Child PID = 17380 has been stopped by a signal, signo = 19
My PID = 17379: Child PID = 17381 has been stopped by a signal, signo = 19
I am ./scheduler, PID = 17380
About to replace myself with the executable prog...
My PID = 17379: Child PID = 17380 has been stopped by a signal, signo = 19
I am ./scheduler, PID = 17381
About to replace myself with the executable prog...
My PID = 17379: Child PID = 17381 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 10, delay = 90
prog[17380]: This is message 0
prog[17380]: This is message 1
prog[17380]: This is message 2
prog[17380]: This is message 3
prog[17380]: This is message 4
prog[17380]: This is message 5
prog[17380]: This is message 6
prog[17380]: This is message 7
My PID = 17379: Child PID = 17380 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 10, delay = 268
prog[17381]: This is message 0
prog[17381]: This is message 1
prog[17381]: This is message 2
My PID = 17379: Child PID = 17381 has been stopped by a signal, signo = 19
prog[17380]: This is message 8
prog[17380]: This is message 9
My PID = 17379: Child PID = 17380 terminated normally, exit status = 0
prog[17381]: This is message 3
prog[17381]: This is message 4
My PID = 17379: Child PID = 17381 has been stopped by a signal, signo = 19
prog[17381]: This is message 5
prog[17381]: This is message 6
prog[17381]: This is message 7
My PID = 17379: Child PID = 17381 has been stopped by a signal, signo = 19
prog[17381]: This is message 8
prog[17381]: This is message 9
My PID = 17379: Child PID = 17381 has been stopped by a signal, signo = 19
My PID = 17379: Child PID = 17381 terminated normally, exit status = 0
I am done
```

4.1.1. Τι συμβαίνει αν το σήμα SIGALRM έρθει ενώ εκτελείται η συνάρτηση χειρισμού του σήματος SIGCHLD ή το αντίστροφο; Πώς αντιμετωπίζει ένας πραγματικός χρονοδρομολογητής χώρου πυρήνα ανάλογα ενδεχόμενα και πώς η δική σας υλοποίηση; Υπόδειξη: μελετήστε τη συνάρτηση install_signal_handlers() που δίνεται.

Η διαδικασία που θα ακολουθηθεί στην περίπτωση αυτή είναι η εξής: όταν εκτελείται ο handler της SIGCHLD, τότε δεν θα εκτελεστεί η συνάρτηση χειρισμού του SIGALRM ακόμη και αν ληφθεί τέτοιο σήμα. Αυτό συμβαίνει καθώς στην install_signal_handlers() έχουμε ορίσει μέσω μάσκας να μπλοκάρεται το σήμα SIGALRM όταν εκτελείται το τμήμα κώδικα του SIGCHLD handler. Αντίστοιχα, έχει οριστεί και στην αντίθετη περίπτωση, όταν δηλαδή εκτελείται ο SIGALRM handler και ληφθεί σήμα SIGCHLD.

Όσον αφορά έναν πραγματικό χρονοδρομολογητή, αυτός λειτουργεί με διακοπές και δε βασίζεται σε σήματα, τα οποία μπορεί να φανούν αναξιόπιστα. Με αυτή την τακτική, έχουμε καλύτερη και πιο άμεση απόκριση, αφού με το που γίνει μια διακοπή, θα εκτελεστεί αμέσως η ρουτίνα εξυπηρέτησής της, ενώ στη περίπτωσή μας, τα σήματα ενδέχεται να έχουν καθυστερήσεις αφού ακόμη και αυτά χρονοδρομολογούνται. Αυτός είναι ο κύριος λόγος που χρησιμοποιούμε διακοπές αντί για σήματα στους πραγματικούς χρονοδρομολογητές.

4.1.2. Κάθε φορά που ο χρονοδρομοληγητής λαμβάνει σήμα SIGCHLD, σε ποια διεργασία παιδί περιμένετε να αναφέρεται αυτό; Τι συμβαίνει αν λόγω εξωτερικού παράγοντα (π.χ. αποστολή SIGKILL) τερματιστεί αναπάντεχα μια οποιαδήποτε διεργασία-παιδί;

Το σήμα SIGCHLD στέλνεται από μία διεργασία στον πατέρα της όταν αυτή δέχεται σήμα που την θέτει σε παύση ή την σκοτώνει.

Ένα τέτοιο σήμα εδώ μπορεί να έρθει σε μία διεργασία όταν αυτή ολοκληρώσει την εκτέλεση της επιτυχημένα, όταν διακόπτεται αναπάντεχα από άλλη διεργασία ή όταν διακοπεί από τον χρονοδρομολογητή μετά το πέρας του κβάντου χρόνου.

Άρα το σήμα μπορεί να αναφέρεται σε οπιαδήποτε διεργασία της ουράς.

Πιο συγκεκριμένα, στη συνάρτηση “sigchld_handler” θεωρούμε δύο μεγάλες περιπτώσεις, τις:

- If (WIFEXITED(status) || WIFSIGNALED(status)) :

Στην περίπτωση αυτή υπάγεται μία διεργασία όταν τερματίζεται φυσιολογικά επειδή ολοκλήρωσε την λειτουργία της (αυτό μπορεί να συμβεί μόνο όταν βρίσκεται στην κεφαλή της λίστας), είτε όταν “σκοτώνεται” αναπάντεχα από μία άλλη διεργασία (αυτό μπορεί να συμβεί όταν βρίσκεται σε οποιοδήποτε σημείο της λίστας).

Τότε, αφαιρούμε τη διεργασία τελείως από τη λίστα.

- If (WIFSTOPPED(status)):

Στην δεύτερη περίπτωση υπάγεται μία διεργασία όταν η λειτουργία της έρχεται σε παύση. Αυτό εδώ συμβαίνει μέσω του χρονοδρομολογητή, όταν η διεργασία βρίσκεται στην κεφαλή της λίστας για χρονικό διάστημα μεγαλύτερο του κβάντου χρόνου. Τότε, επιλέγουμε η διεργασία να τοποθετηθεί στο τέλος της λίστας.

4.1.3. Γιατί χρειάζεται ο χειρισμός δύο σημάτων για την υλοποίηση του χρονοδρομολογητή; θα μπορούσε ο χρονοδρομοληγητής να χρησιμοποιεί μόνο το σήμα SIGALRM για να σταματά την τρέχουσα διεργασία και να ξεκινά την επόμενη; Τι ανεπιθύμητη συμπεριφορά θα μπορούσε να εμφανίζει μια τέτοια υλοποίηση; Υπόδειξη: Η παραλαβή του σήματος SIGCHLD εγγυάται ότι η τρέχουσα διεργασία έλαβε το σήμα SIGSTOP και έχει σταματήσει.

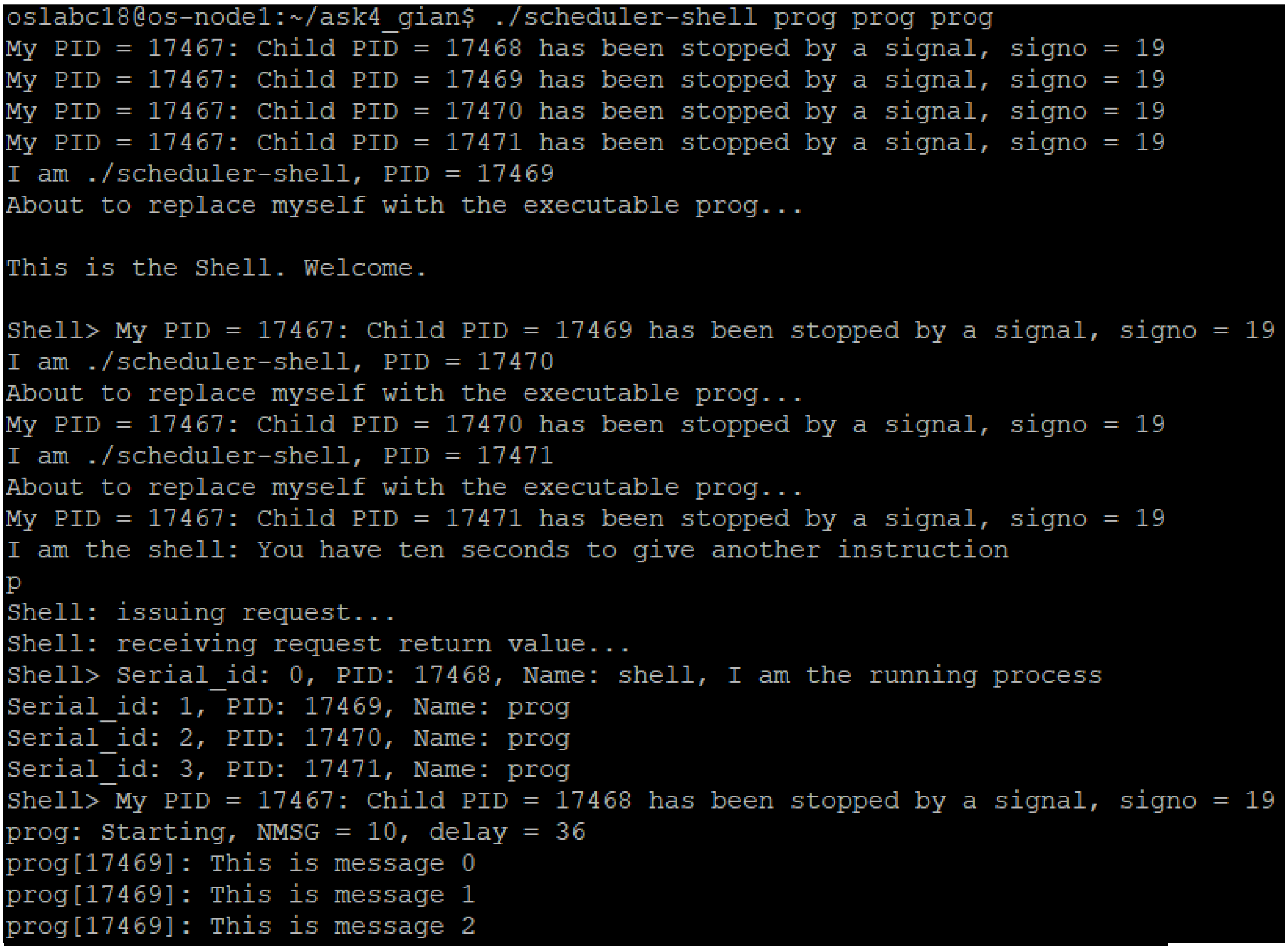
Στην άσκηση χρησιμοποιήσαμε το σήμα SIGCHLD προκειμένου η διεργασία-παιδί να ενημερώσει άμεσα τον πατέρα-χρονοδρομολογητή ότι άλλαξε η κατάστασή της.

Το σήμα SIGALRM χρησιμοποιήθηκε από τον πατέρα-χρονοδρομολογητή για να θέσει σε παύση την διεργασία-παιδί όταν αυτή εκτελείται για χρονικό διάστημα μεγαλύτερο του κβάντου χρόνου.

Έτσι, αν χρησιμοποιούσαμε μόνο τον χειρισμό του SIGALRM, στην περίπτωση που μία διεργασία τερματιζόταν αναπάντεχα μέσω του σήματος SIGKILL ή ολοκλήρωνε ομαλά την λειτουργία της, ο χρονοδρομοληγητής δεν θα μπορούσε να την αφαιρέσει επί τόπου από την λίστα. Θα μπορούσαμε να υλοποιούμε τις περιστροφές σύμφωνα με το σχήμα round-robin, και κάθε φορά να ελέγχουμε την κατάσταση της διεργασίας στην κεφαλή <u>μετά το πέρας του κβάντου χρόνου</u>. Η ανάγκη αυτή για αναμονή λήξης του κβάντου χρόνου σε κάθε περίπτωση θα οδηγούσε σε πιο αργή απόκριση του συστήματος.

ΜΕΡΟΣ 2^{ο}

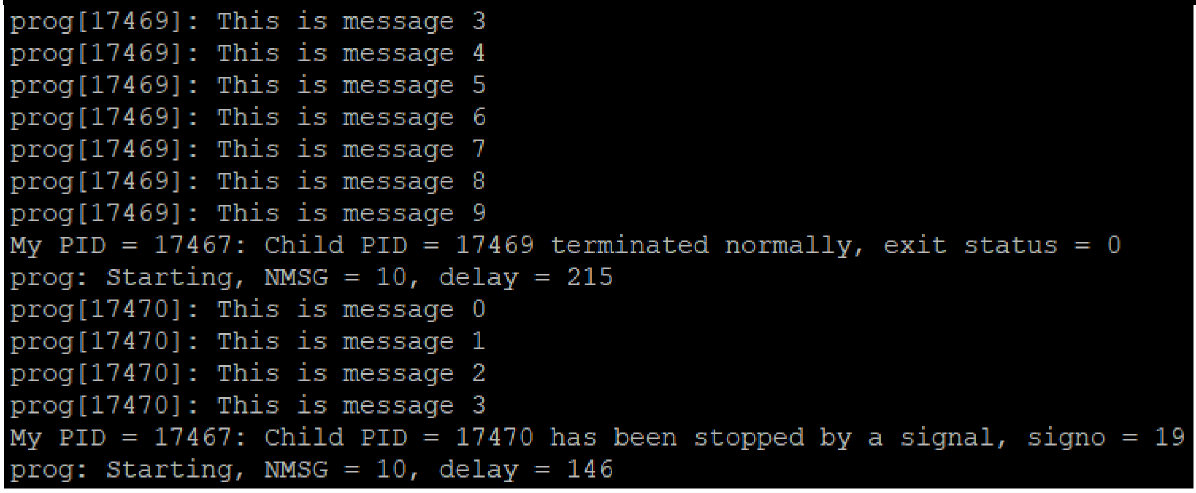
Ένα παράδειγμα εκτέλεσης του προγράμματός μας



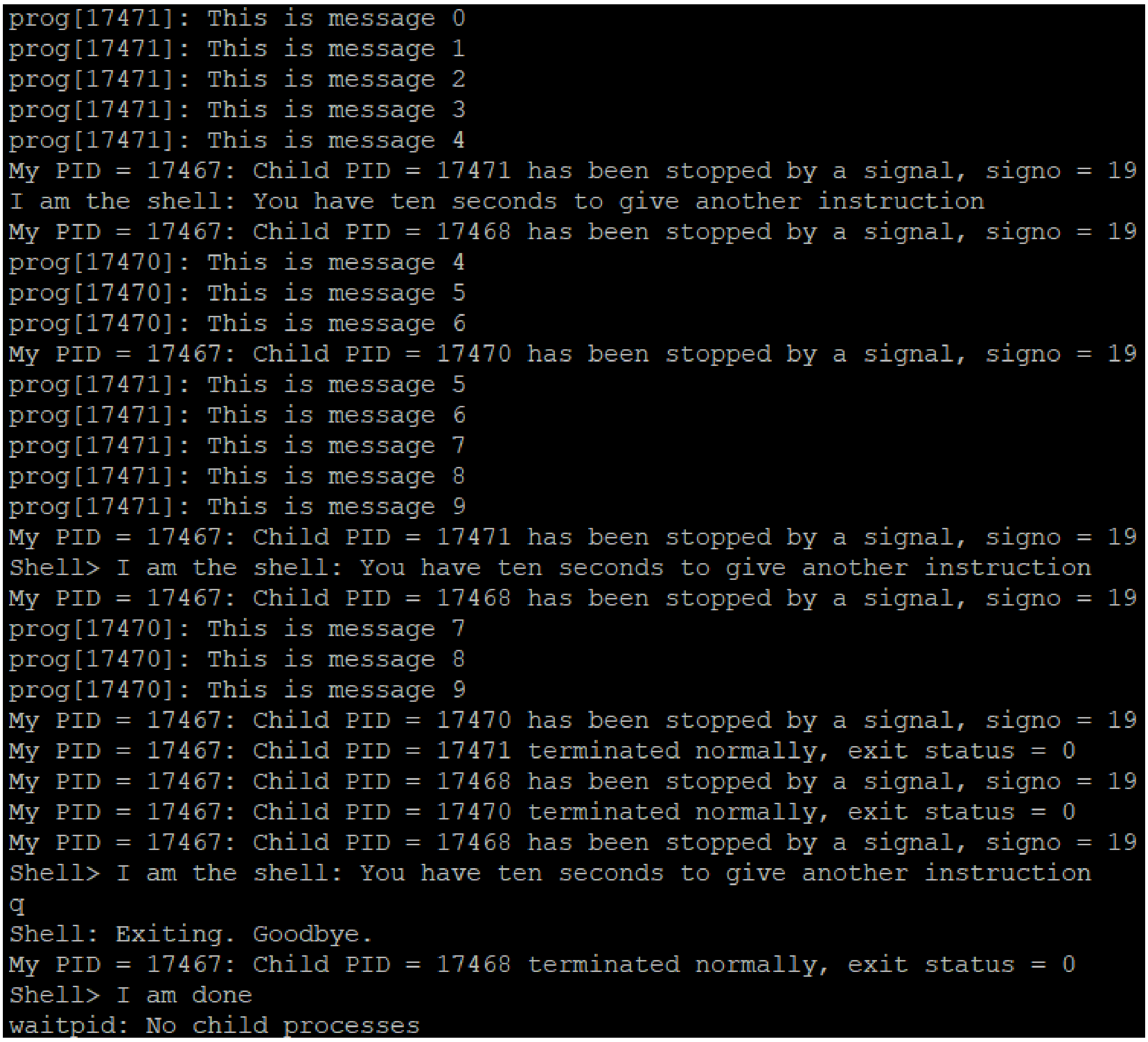
```
oslabc18@os-node1:~/ask4_gian$ ./scheduler-shell prog prog prog
My PID = 17467: Child PID = 17468 has been stopped by a signal, signo = 19
My PID = 17467: Child PID = 17469 has been stopped by a signal, signo = 19
My PID = 17467: Child PID = 17470 has been stopped by a signal, signo = 19
My PID = 17467: Child PID = 17471 has been stopped by a signal, signo = 19
I am ./scheduler-shell, PID = 17469
About to replace myself with the executable prog...

This is the Shell. Welcome.

Shell> My PID = 17467: Child PID = 17469 has been stopped by a signal, signo = 19
I am ./scheduler-shell, PID = 17470
About to replace myself with the executable prog...
My PID = 17467: Child PID = 17470 has been stopped by a signal, signo = 19
I am ./scheduler-shell, PID = 17471
About to replace myself with the executable prog...
My PID = 17467: Child PID = 17471 has been stopped by a signal, signo = 19
I am the shell: You have ten seconds to give another instruction
p
Shell: issuing request...
Shell: receiving request return value...
Shell> Serial_id: 0, PID: 17468, Name: shell, I am the running process
Serial_id: 1, PID: 17469, Name: prog
Serial_id: 2, PID: 17470, Name: prog
Serial_id: 3, PID: 17471, Name: prog
Shell> My PID = 17467: Child PID = 17468 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 10, delay = 36
prog[17469]: This is message 0
prog[17469]: This is message 1
prog[17469]: This is message 2
```



```
prog[17469]: This is message 3
prog[17469]: This is message 4
prog[17469]: This is message 5
prog[17469]: This is message 6
prog[17469]: This is message 7
prog[17469]: This is message 8
prog[17469]: This is message 9
My PID = 17467: Child PID = 17469 terminated normally, exit status = 0
prog: Starting, NMSG = 10, delay = 215
prog[17470]: This is message 0
prog[17470]: This is message 1
prog[17470]: This is message 2
prog[17470]: This is message 3
My PID = 17467: Child PID = 17470 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 10, delay = 146
```



```
prog[17471]: This is message 0
prog[17471]: This is message 1
prog[17471]: This is message 2
prog[17471]: This is message 3
prog[17471]: This is message 4
My PID = 17467: Child PID = 17471 has been stopped by a signal, signo = 19
I am the shell: You have ten seconds to give another instruction
My PID = 17467: Child PID = 17468 has been stopped by a signal, signo = 19
prog[17470]: This is message 4
prog[17470]: This is message 5
prog[17470]: This is message 6
My PID = 17467: Child PID = 17470 has been stopped by a signal, signo = 19
prog[17471]: This is message 5
prog[17471]: This is message 6
prog[17471]: This is message 7
prog[17471]: This is message 8
prog[17471]: This is message 9
My PID = 17467: Child PID = 17471 has been stopped by a signal, signo = 19
Shell> I am the shell: You have ten seconds to give another instruction
My PID = 17467: Child PID = 17468 has been stopped by a signal, signo = 19
prog[17470]: This is message 7
prog[17470]: This is message 8
prog[17470]: This is message 9
My PID = 17467: Child PID = 17470 has been stopped by a signal, signo = 19
My PID = 17467: Child PID = 17471 terminated normally, exit status = 0
My PID = 17467: Child PID = 17468 has been stopped by a signal, signo = 19
My PID = 17467: Child PID = 17470 terminated normally, exit status = 0
My PID = 17467: Child PID = 17468 has been stopped by a signal, signo = 19
Shell> I am the shell: You have ten seconds to give another instruction
q
Shell: Exiting. Goodbye.
My PID = 17467: Child PID = 17468 terminated normally, exit status = 0
Shell> I am done
waitpid: No child processes
```

Source code:

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>

#include <sys/wait.h>
#include <sys/types.h>

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

/* Compile-time parameters. */
#define SCHED_TQ_SEC 2                          /* time quantum */
#define TASK_NAME_SZ 60                          /* maximum size for a task's name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */
```

```
struct process {
         pid_t mypid;
         struct process *next;
         char *name;
        int serial_id;

};

struct process *head = NULL;
struct process *last = NULL;
struct process *running = NULL;
struct process *newnode = NULL;

int nproc = 0;
int serial_id_counter = 0;



/* Print a list of all tasks currently being scheduled.     */

static void sched_print_tasks(void) {
        struct process *temp;
        temp = head;
        running = head;
        while (temp != NULL) {
                printf("Serial_id: %d, PID: %d, Name: %s",
                                  temp->serial_id, temp->mypid, temp->name);
                if (temp->serial_id == running->serial_id) {
                        printf(", I am the running process\n");
                }
                else {
                        printf("\n");
                }
                temp = temp->next;
        }

}


/* Send SIGKILL to a task determined by the value of its
  * scheduler-specific id.
  */
static int sched_kill_task_by_id(int id) {

                struct process *temp;
                temp = head;

                while (temp != NULL) {
                        if (temp->serial_id == id) {
                                kill(temp->mypid, SIGKILL);
                                return 0;
                        }
                        else {
                                temp = temp->next;
                        }
                }

                return -ENOSYS;
}
```

```
/* Create a new task.    */
static void sched_create_task(char *executable) {

    pid_t my_pid;

    char *newargv[] = {executable, NULL, NULL, NULL};
    char *newenviron[] = {NULL};

    my_pid = fork();

    if (my_pid < 0) {
        printf("Error with forks\n");
    }
    if (my_pid == 0) {          // child's code
     raise(SIGSTOP);
        printf("I am %s, PID = %ld\n", executable, (long)getpid());
        sleep(2);
        execve(executable, newargv, newenviron);

        perror("execve");     // because it only returns on error
        exit(1);
    }
    else {          // scheduler's code

        newnode = (struct process *)malloc(sizeof(struct process));        // create space for its struct
     if (newnode==NULL) printf("Error with malloc\n");

     printf("-->Process with PID = %ld was just created.\n", (long)my_pid);

        newnode->mypid = my_pid;

        newnode->name=(char*)malloc(strlen(executable)+1);
          strcpy(newnode->name,executable);

        newnode->next = NULL;

        newnode->serial_id = serial_id_counter;
        serial_id_counter = serial_id_counter+1;

        last->next = newnode;
        last = newnode;

        nproc = nproc+1;

    }

}


/* Process requests by the shell.    */
static int process_request(struct request_struct *rq) {

        switch (rq->request_no) {
                case REQ_PRINT_TASKS:
                        sched_print_tasks();
                        return 0;

                case REQ_KILL_TASK:
                        return sched_kill_task_by_id(rq->task_arg);
                case REQ_EXEC_TASK:
                        sched_create_task(rq->exec_task_arg);
                        return 0;

                default:
                        return -ENOSYS;
        }
}


/*
 * SIGALRM handler
 */
static void sigalrm_handler(int signum) {
        kill(head->mypid, SIGSTOP);
}


/*
 * SIGCHLD handler
 */
static void sigchld_handler(int signum) {

        pid_t p;
    int status;

    while( 1 ) {

        //perimene opoiodhpote paidi
        p=waitpid(-1,&status,WNOHANG|WUNTRACED);

        //error sthn waitpid
        if (p<0) {
          perror("waitpid");
          exit(1);
    }

          //no child has changed its state
        if(p==0) break;         /////////

        //dhladh ean to p>0 pou shmainei oti h waitpid ekane return to pid tou child whose state has chenged
          explain_wait_status(p,status);

        //me vash to status thanatou ths p, pairnoume dyo kyries periptwseis
          if ( WIFEXITED(status) || WIFSIGNALED(status) )    {           //if finished REMOVE it from the list

                struct process* temp;
                struct process* current;
                temp=head;

                while (temp != NULL) {

                    if (temp->mypid==p && temp==head) {              //if temp == head -> got to delete
the head
                                if (temp->next == NULL){          //an yparxei mono
to head sthn oura         MESW AUTOU TOU SHMEIOU THA GINEI H EJODOS APO TO PROGRAMMA
                                    free(temp);
                                    printf("I am done\n");     //telos h douleia
                                    //exit(0);
                        }
                        else{
                            head=temp->next;        //an yparxoyn ki alla, head = to pisw tou
                            free(temp);
                        }
                }

                else if (temp->mypid==p && temp==last) {                //if TEMP == last -> apla to
diagrafoume kanontas to NULL kai auto kai to epomeno tou
                    current = head;
                    while (current->next->next != NULL) {
                                current = current->next;
                    }
                       free( temp );
                    last = current;
                    last->next = NULL;
              }

                else if (temp->mypid==p) {                                      //if temp == random process of
the queue
                        current = head;
                    while (current->next != temp) {
                        current = current->next;
                    }
                    current->next = temp->next;
                    temp->next = NULL;
                    free (temp);

              }

                else {                                                      // iterate till you find the temp that
matches
                    temp = temp -> next;
                    continue;
                }
            }
        }

        if ( WIFSTOPPED(status) ) {                       // if stopped by the scheduler, SIGALRM ->
sigalrm_hanlder() -> SIGSTOP sent to the head
                                                                  // (only the head is being stopped by the scheduler)
                                                                         // bring the head->next to the head
// move the head to the last->next
          last->next=head;
            last=head;

            struct process * temp;

          temp=head;
          head=head->next;                              // to head na ginei to epomeno tou

            temp->next=NULL;                        // to palio head = temp, na exei next = null

       }


       if ( WIFSTOPPED(status) ) {                  // take care of the alarm time

          if (head->serial_id == 0) {
              printf("I am the shell: You have ten seconds to give another instruction\nShell> ");
```

```
                 alarm(5*SCHED_TQ_SEC);
                 kill(head->mypid,SIGCONT);
            }

            else {
                 alarm(SCHED_TQ_SEC);        // set the alarm-counter to the time quantum
                 kill(head->mypid,SIGCONT);  // so we send a SIGCONT to the head procces, so that it
continues from its pause
                                              // and after the alarm time has passed, is stops again
because of the sigalrm_handler(int signum)
            }

        }

        else {
            alarm(SCHED_TQ_SEC);             // set the alarm-counter to the time quantum
            kill(head->mypid,SIGCONT);       // so we send a SIGCONT to the head procces, so that it
continues from its pause
                                              // and after the alarm time has passed, is stops again
because of the sigalrm_handler(int signum)
        }

     }

}


/* Disable delivery of SIGALRM and SIGCHLD. */
static void signals_disable(void) {

        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);

        //              how=union    the set
        if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
                perror("signals_disable: sigprocmask");
                exit(1);
        }
}


/* Enable delivery of SIGALRM and SIGCHLD.    */
static void signals_enable(void) {

        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        //              how=remove      the set
        if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
                perror("signals_enable: sigprocmask");
                exit(1);
        }
}
```

```
/* Install two signal handlers.
  * One for SIGCHLD, one for SIGALRM.
  * Make sure both signals are masked when one of them is running.
  */
static void install_signal_handlers(void) {

        sigset_t sigset;
        struct sigaction sa;

        sa.sa_handler = sigchld_handler;
        sa.sa_flags = SA_RESTART;
        sigemptyset(&sigset);
        sigaddset(&sigset, SIGCHLD);
        sigaddset(&sigset, SIGALRM);
        sa.sa_mask = sigset;
        if (sigaction(SIGCHLD, &sa, NULL) < 0) {
                perror("sigaction: sigchld");
                exit(1);
        }

        sa.sa_handler = sigalrm_handler;
        if (sigaction(SIGALRM, &sa, NULL) < 0) {
                perror("sigaction: sigalrm");
                exit(1);
        }

        /*
          * Ignore SIGPIPE, so that write()s to pipes
          * with no reader do not result in us being killed,
          * and write() returns EPIPE instead.
          */
        if (signal(SIGPIPE, SIG_IGN) < 0) {
                perror("signal: sigpipe");
                exit(1);
        }
}



static void do_shell(char *executable, int wfd, int rfd) {

        char arg1[10], arg2[10];
        char *newargv[] = { executable, NULL, NULL, NULL };
        char *newenviron[] = { NULL };

        // write to arg1,2 with format="%05d" from wfd, rfd
        sprintf(arg1, "%05d", wfd);
        sprintf(arg2, "%05d", rfd);
        newargv[1] = arg1;
        newargv[2] = arg2;

        raise(SIGSTOP);         // wait for the parent to send SIGCONT
        execve(executable, newargv, newenviron);        //execute the executable with arguments from array
pointed to by newargv

        /* execve() only returns on error */
        perror("scheduler: child: execve");
        exit(1);
}
```

```
/* Create a new shell task.
 *
 * The shell gets special treatment:
 * two pipes are created for communication and passed
 * as command-line arguments to the executable.
 */
static void sched_create_shell(char *executable, int *request_fd, int *return_fd) {

        pid_t p;
        int pfds_rq[2], pfds_ret[2];            // 2 pipes with these fds


        if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
                perror("pipe");
                exit(1);
        }

        p = fork();
        if (p < 0) {
                perror("scheduler: fork");
                exit(1);
        }

        if (p == 0) {
                /* Child */
                close(pfds_rq[0]);
                close(pfds_ret[1]);

                do_shell(executable, pfds_rq[1], pfds_ret[0]);
                assert(0);
        }
        /* Parent */
        close(pfds_rq[1]);
        close(pfds_ret[0]);
        *request_fd = pfds_rq[0];
        *return_fd = pfds_ret[1];

        // head <- shell
        // last <- head

        head->serial_id = 0;
        head->mypid = p;
        head->name = "shell";
        head->next = NULL;



}


static void shell_request_loop(int request_fd, int return_fd) {

        int ret;
        struct request_struct rq;

        /*
         * Keep receiving requests from the shell.
         */
        for (;;) {
```

```
        //              from        to      size
        if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
            perror("scheduler: read from shell");
            fprintf(stderr, "Scheduler: giving up on shell request processing.\n");
            break;
        }

        signals_disable();
        ret = process_request(&rq);       // apokwdikopoiei me vash to rq->request_no kai kalei thn
antistoixh func apo tis
        signals_enable();                 // sched_print_tasks, sched_kill_task_by_id,
sched_create_task


        //              to          from    size
        if (write(return_fd, &ret, sizeof(ret)) != sizeof(ret)) {
            perror("scheduler: write to shell");
            fprintf(stderr, "Scheduler: giving up on shell request processing.\n");
            break;
        }
    }
}



int main(int argc, char *argv[]) {

    int nproc;

    /* Two file descriptors for communication with the shell */
    static int request_fd, return_fd;

    head=(struct process *)malloc(sizeof(struct process));
    last = head;

    /* Create the shell. */
    sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
    /* TODO: add the shell to the scheduler's tasks */

    /*
     * For each of argv[1] to argv[argc - 1],
     * create a new child process, add it to the process list.
     */

    nproc = argc-1; /* number of proccesses goes here */

    if (nproc==0) {
        fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
        exit(1);
    }

    char executable[] = "prog";
    char *newargv[] = {executable, NULL, NULL, NULL};
    char *newenviron[] = {NULL};

    pid_t my_pid;

    int i;
    for (i=0; i<nproc; i++) {
        my_pid = fork();

        if (my_pid<0) {
            printf("Error with fork\n");
        }

        if (my_pid == 0) {                  // child's code

            raise(SIGSTOP);             // and wait for scheduler's SIGCONT
            printf("I am %s, PID = %ld\n", argv[0], (long)getpid());
            printf("About to replace myself with the executable %s...\n", executable);
            sleep(2);

            execve (executable, newargv, newenviron);
            //because execve only returns on error
            perror("execve");
            exit(1);

        }

        else {                              // father's code
                                             // the first time that this part is executed, the list has
only the shell on the head

            newnode = (struct process*) malloc(sizeof(struct process));

            if (newnode == NULL) {
                printf("Error with malloc\n");
            }

            newnode->mypid = my_pid;
            newnode->name = argv[i+1]; // i executable's name
            newnode->next = NULL;
            newnode->serial_id = i+1;

            last->next = newnode;
            last = newnode;

        }


    }


    /* Wait for all children to raise SIGSTOP before exec()ing. */
    wait_for_ready_children(nproc);

    /* Install SIGALRM and SIGCHLD handlers. */
    install_signal_handlers();


    alarm(SCHED_TQ_SEC);       // begin the alarm countdown

    kill(head->mypid, SIGCONT);        //continue the head process (shell)

    shell_request_loop(request_fd, return_fd);

    /* Now that the shell is gone, just loop forever
     * until we exit from inside a signal handler.
```

```
         */
        while (pause())
                ;

        /* Unreachable */
        fprintf(stderr, "Internal error: Reached unreachable point\n");
        return 1;
}
```

4.2.1. Όταν και ο φλοιός υφίσταται χρονοδρομολόγηση, ποια εμφανίζεται πάντοτε ως τρέχουσα διεργασία στη λίστα διεργασιών (εντολή 'p'); Θα μπορούσε να μη συμβαίνει αυτό; Γιατί;

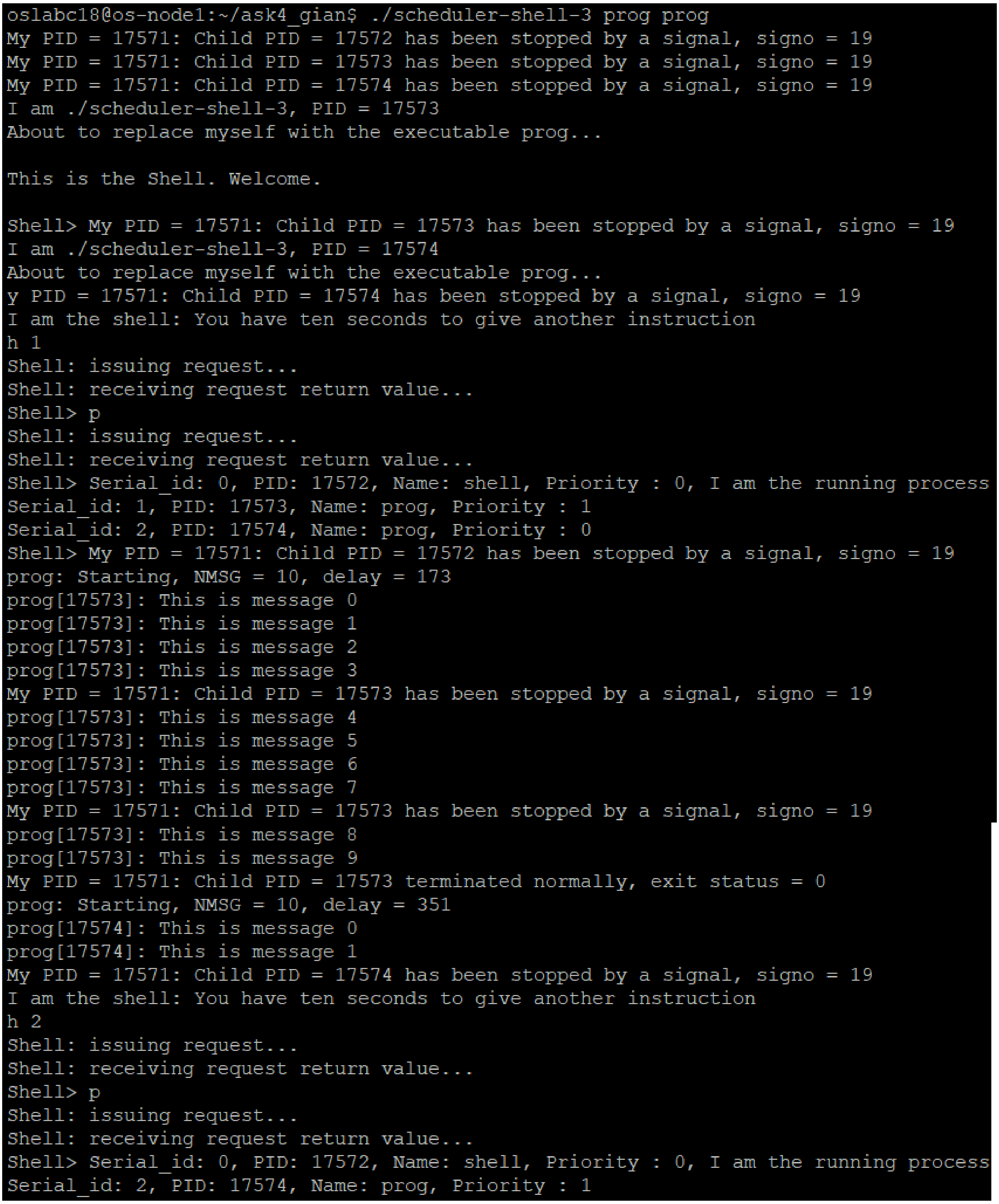
Πάντα όταν εκτελούμε την εντολή 'p' ως τρέχουσα διαδικασία εμφανίζεται να είναι ο φλοιός (serial_id:0).Η ακολουθία συμβάντων που περιγράφεται παραπάνω είναι φυσιολογική με βάση την υλοποίησή μας. Η εκτύπωση των διεργασιών γίνεται μόνο όταν η τρέχουσα διεργασία είναι ο φλοιός. Αυτό συμβαίνει διότι η εντολή 'p' δίνεται μόνο όταν στο head της λίστας μας είναι ο φλοιός ως τρέχουσα διεργασία. Τη στιγμή εκείνη απενεργοποιούνται και τα λοιπά σήματα.

4.2.2. Γιατί είναι αναγκαίο να συμπεριλάβετε κλήσεις signals_disable(), _enable() γύρω από την συνάρτηση υλοποίησης αιτήσεων του φλοιού; Υπόδειξη: Η συνάρτηση υλοποίησης αιτήσεων του φλοιού μεταβάλλει δομές όπως η ουρά εκτέλεσης των διεργασιών

Οι συναρτήσεις signals_disable() και signals_enable() χρησιμοποιούνται για την απενεργοποίηση και ενεργοποίηση των σημάτων αντίστοιχα. Αυτές είναι απαραίτητο να χρησιμοποιηθούν ώστε να διασφαλίσουμε ότι όσο εξυπηρετούνται οι αιτήσεις στο φλοιό δε θα γίνει χειρισμός άλλου σήματος. Συνεπώς, εξασφαλίζεται ότι δε θα διαφοροποιηθούν οι δομές που χρησιμοποιούνται τη δεδομένη στιγμή. Αν δεν υπήρχαν οι συναρτήσεις αυτές και γινόταν κανονικά ο χειρισμός άλλων σημάτων θα ήταν πιθανό να τροποποιηθεί η λίστα διεργασιών μας και να οδηγούσε τον χρονοδρομολογητή σε λάθος αποτέλεσμα.

Μέρος 3ο

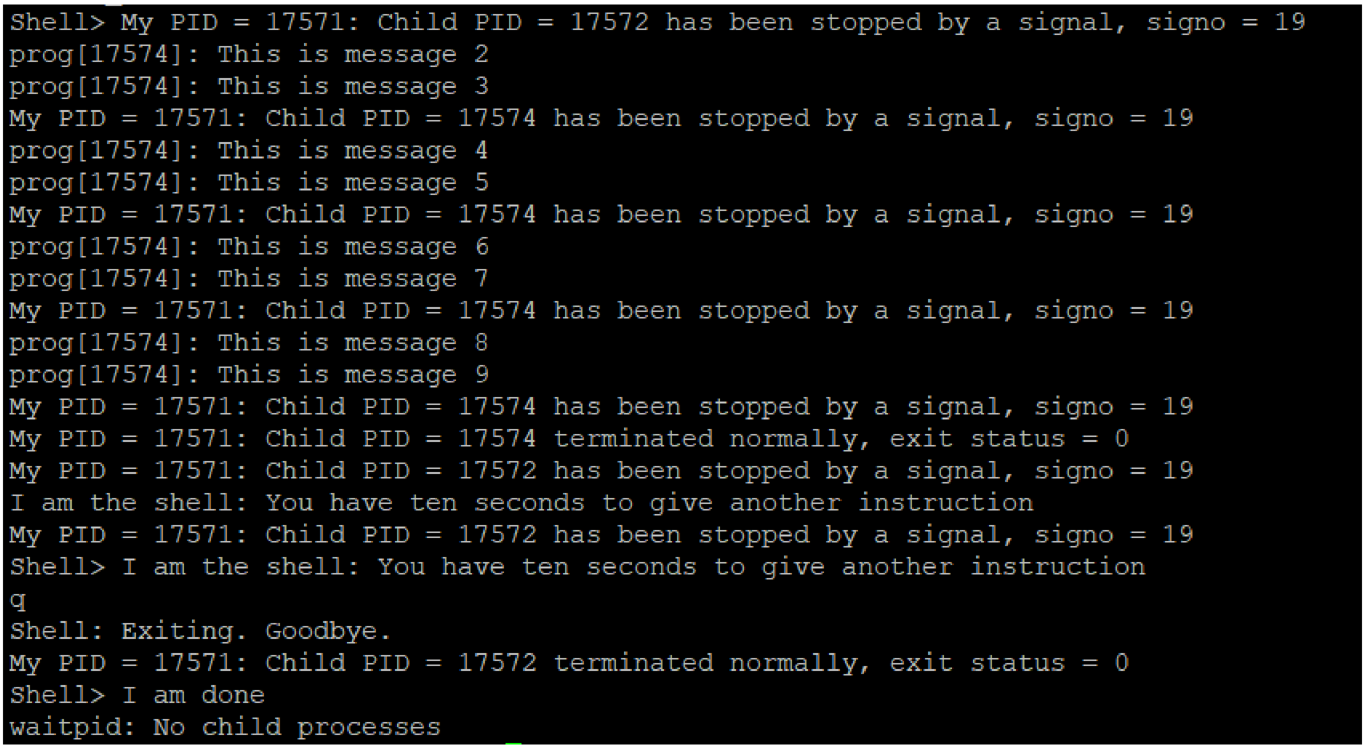
Ένα παράδειγμα εκτέλεσης του προγράμματος μας



```
oslabc18@os-node1:~/ask4_gian$ ./scheduler-shell-3 prog prog
My PID = 17571: Child PID = 17572 has been stopped by a signal, signo = 19
My PID = 17571: Child PID = 17573 has been stopped by a signal, signo = 19
My PID = 17571: Child PID = 17574 has been stopped by a signal, signo = 19
I am ./scheduler-shell-3, PID = 17573
About to replace myself with the executable prog...

This is the Shell. Welcome.

Shell> My PID = 17571: Child PID = 17573 has been stopped by a signal, signo = 19
I am ./scheduler-shell-3, PID = 17574
About to replace myself with the executable prog...
y PID = 17571: Child PID = 17574 has been stopped by a signal, signo = 19
I am the shell: You have ten seconds to give another instruction
h 1
Shell: issuing request...
Shell: receiving request return value...
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Shell> Serial_id: 0, PID: 17572, Name: shell, Priority : 0, I am the running process
Serial_id: 1, PID: 17573, Name: prog, Priority : 1
Serial_id: 2, PID: 17574, Name: prog, Priority : 0
Shell> My PID = 17571: Child PID = 17572 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 10, delay = 173
prog[17573]: This is message 0
prog[17573]: This is message 1
prog[17573]: This is message 2
prog[17573]: This is message 3
My PID = 17571: Child PID = 17573 has been stopped by a signal, signo = 19
prog[17573]: This is message 4
prog[17573]: This is message 5
prog[17573]: This is message 6
prog[17573]: This is message 7
My PID = 17571: Child PID = 17573 has been stopped by a signal, signo = 19
prog[17573]: This is message 8
prog[17573]: This is message 9
My PID = 17571: Child PID = 17573 terminated normally, exit status = 0
prog: Starting, NMSG = 10, delay = 351
prog[17574]: This is message 0
prog[17574]: This is message 1
My PID = 17571: Child PID = 17574 has been stopped by a signal, signo = 19
I am the shell: You have ten seconds to give another instruction
h 2
Shell: issuing request...
Shell: receiving request return value...
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Shell> Serial_id: 0, PID: 17572, Name: shell, Priority : 0, I am the running process
Serial_id: 2, PID: 17574, Name: prog, Priority : 1
```



```
Shell> My PID = 17571: Child PID = 17572 has been stopped by a signal, signo = 19
prog[17574]: This is message 2
prog[17574]: This is message 3
My PID = 17571: Child PID = 17574 has been stopped by a signal, signo = 19
prog[17574]: This is message 4
prog[17574]: This is message 5
My PID = 17571: Child PID = 17574 has been stopped by a signal, signo = 19
prog[17574]: This is message 6
prog[17574]: This is message 7
My PID = 17571: Child PID = 17574 has been stopped by a signal, signo = 19
prog[17574]: This is message 8
prog[17574]: This is message 9
My PID = 17571: Child PID = 17574 has been stopped by a signal, signo = 19
My PID = 17571: Child PID = 17574 terminated normally, exit status = 0
My PID = 17571: Child PID = 17572 has been stopped by a signal, signo = 19
I am the shell: You have ten seconds to give another instruction
My PID = 17571: Child PID = 17572 has been stopped by a signal, signo = 19
Shell> I am the shell: You have ten seconds to give another instruction
q
Shell: Exiting. Goodbye.
My PID = 17571: Child PID = 17572 terminated normally, exit status = 0
Shell> I am done
waitpid: No child processes
```

Source code:

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>

#include <sys/wait.h>
#include <sys/types.h>

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

/* Compile-time parameters. */
#define SCHED_TQ_SEC 2                /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */


struct process {
        pid_t mypid;
        struct process *next;
        char *name;
        int serial_id;
        int priority;
};

struct process *head = NULL;
struct process *last = NULL;
struct process *running = NULL;
struct process *newnode = NULL;

int nproc = 0;
int serial_id_counter = 0;



/* Print a list of all tasks currently being scheduled.     */

static void sched_print_tasks(void) {
    struct process *temp;
    temp = head;
    running = head;
    while (temp != NULL) {
        printf("Serial_id: %d, PID: %d, Name: %s, Priority : %d",
                 temp->serial_id, temp->mypid, temp->name,temp->priority);
        if (temp->serial_id == running->serial_id) {
            printf(", I am the running process\n");
        }
        else {
            printf("\n");
        }
        temp = temp->next;
    }

}


/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int sched_kill_task_by_id(int id) {

        struct process *temp;
        temp = head;

        while (temp != NULL) {
            if (temp->serial_id == id) {
                kill(temp->mypid, SIGKILL);
                return 0;
            }
            else {
                temp = temp->next;
            }
        }

        return -ENOSYS;
}

/*fix the priorities*/
static int sched_set_high_p(int id){
     struct process * temp = head;
     //struct process * prev = NULL;
     //set the priority of a process to high and put it in the head of my list
        while (temp!=NULL){
                if (temp->serial_id == id ){
                        temp->priority=1;
              return 0;
                }
                else
                        if(temp->next!=NULL){
                                temp=temp->next;
                        }
              else{
                                break;
              }
        }
        return -ENOSYS;
}

static int sched_set_low_p(int id){
     struct process * temp=head;
        while (temp!=NULL){
                if (temp->serial_id == id ){
                       temp->priority=0;
                       return 0;
                }
                else
                        if(temp->next!=NULL)
                                temp=temp->next;
                        else
                                break;
        }
        return -ENOSYS;

}


/* Create a new task.     */
static void sched_create_task(char *executable) {

    pid_t my_pid;

    char *newargv[] = {executable, NULL, NULL, NULL};
    char *newenviron[] = {NULL};

    my_pid = fork();

    if (my_pid < 0) {
        printf("Error with forks\n");
    }
    if (my_pid == 0) {          // child's code
     raise(SIGSTOP);
        printf("I am %s, PID = %ld\n", executable, (long)getpid());
        sleep(2);
        execve(executable, newargv, newenviron);

        perror("execve");     // because it only returns on error
        exit(1);
    }
    else {          // scheduler's code

        newnode = (struct process *)malloc(sizeof(struct process));        // create space for its struct
     if (newnode==NULL) printf("Error with malloc\n");

     printf("-->Process with PID = %ld was just created.\n", (long)my_pid);

        newnode->mypid = my_pid;

        newnode->name=(char*)malloc(strlen(executable)+1);
          strcpy(newnode->name,executable);

        newnode->next = NULL;
```

```
      newnode->priority=0;

        newnode->serial_id = serial_id_counter;
        serial_id_counter = serial_id_counter+1;

        last->next = newnode;
        last = newnode;

        nproc = nproc+1;

      }

}


/* Process requests by the shell.     */
static int process_request(struct request_struct *rq) {

        switch (rq->request_no) {
                case REQ_PRINT_TASKS:
                        sched_print_tasks();
                        return 0;

                case REQ_KILL_TASK:
                        return sched_kill_task_by_id(rq->task_arg);

                case REQ_EXEC_TASK:
                        sched_create_task(rq->exec_task_arg);
                        return 0;

          case REQ_HIGH_TASK:
              sched_set_high_p(rq->task_arg);
              return 0;

          case REQ_LOW_TASK:
              sched_set_low_p(rq->task_arg);
              return 0;

                default:
                        return -ENOSYS;
        }
}


/*
 * SIGALRM handler
 */
static void sigalrm_handler(int signum) {
        kill(head->mypid, SIGSTOP);
}


/*
 * SIGCHLD handler
 */
static void sigchld_handler(int signum) {

        pid_t p;
     int status;
```

```
        int flag = 0;
        while( 1 ) {

              //perimene opoiodhpote paidi
              p=waitpid(-1,&status,WNOHANG|WUNTRACED);

              //error sthn waitpid
              if (p<0) {
                perror("waitpid");
                exit(1);
        }

                //no child has changed its state
              if(p==0) break;              /////////

              //dhladh ean to p>0 pou shmainei oti h waitpid ekane return to pid tou child whose state has chenged
                explain_wait_status(p,status);

              //me vash to status thanatou ths p, pairnoume dyo kyries periptwseis
                if ( WIFEXITED(status) || WIFSIGNALED(status) )    {                  //if finished REMOVE it from the list

                        struct process* temp;
                        struct process* current;
                        temp=head;

                        while (temp != NULL) {

                                if (temp->mypid==p && temp==head) {                        //if temp == head -> got to delete
the head
                                                                              if (temp->next == NULL){               //an yparxei mono
to head sthn oura                  MESW AUTOU TOU SHMEIOU THA GINEI H EJODOS APO TO PROGRAMMA
                                                          free(temp);
                                                          printf("I am done\n");      //telos h douleia
                                                          //exit(0);
                                                }
                                                else{
                                                        head=temp->next;        //an yparxoyn ki alla, head = to pisw tou
                                                        free(temp);
                                                }
                                }

                                else if (temp->mypid==p && temp==last) {                //if TEMP == last -> apla to
diagrafoume kanontas to NULL kai auto kai to epomeno tou
                                        current = head;
                                        while (current->next->next != NULL) {
                                                              current = current->next;
                                        }
                                              free( temp );
                                        last = current;
                                        last->next = NULL;
                                        }

                                else if (temp->mypid==p) {                                    //if temp == random process of the
queue
                                                current = head;
                                        while (current->next != temp) {
                                                current = current->next;
                                        }
                                        current->next = temp->next;
                                        temp->next = NULL;
```

```
                    free (temp);

                    }

                else {                                                  // iterate till you find the temp that
matches
                    temp = temp -> next;
                    continue;
                }
                break;
            }
            temp=head;
            struct process *current1 = head;
            //struct process * prev = NULL;
                     //search for high priorities
                     while (temp!=NULL){
                        if (temp->priority!=1){ //move this node to the tail
                            last->next=head;
                                    last=head;
                                    head=head->next;
                                    last->next=NULL;
                                    temp=head;
                                    if (temp == current1)
                        break;
                                    continue;
                            }
                            else{
                                  flag=1;
                                 break;
                            }
            }

        }

        if ( WIFSTOPPED(status) ) {                    // if stopped by the scheduler, SIGALRM ->
sigalrm_hanlder() -> SIGSTOP sent to the head
                                                          // (only the head is being stopped by the scheduler)
// bring the head->next to the head                                                                        //
move the head to the last->next

            if(head==NULL) {
                printf("Empty List\n");
                exit(0);
            }
            if(head->next != NULL) {

                    last->next=head;
                last=head;
                head = head->next;     //TO HEAD NA GINEI TO EPOMENO TOU
                last->next = NULL;
                struct process * current1;
                struct process * temp;

                current1 = head;
                    temp=head;
                    //SEARCH FOR PRIORITY = HIGH
                while (temp!=NULL){
                    if (temp->priority!=1){ //move this node to the end
                        last->next=head;
                        last=head;
                        head=head->next;
                        last->next=NULL;
                        temp=head;
                        if (temp == current1)
                            break;
                        continue;
                    }
                    else{
                        flag=1;
                        break;
                    }
                }
            }
        }


        if ( WIFSTOPPED(status) ) {                // take care of the alarm time

            if (head->serial_id == 0 && (head->priority == 1 || flag == 0)) {
                printf("I am the shell: You have ten seconds to give another instruction\nShell> ");
                alarm(5*SCHED_TQ_SEC);
                kill(head->mypid,SIGCONT);
            }

            else {
                alarm(SCHED_TQ_SEC);               // set the alarm-counter to the time quantum
                kill(head->mypid,SIGCONT);    // so we send a SIGCONT to the head procces, so that it
continues from its pause
                                                    // and after the alarm time has passed, is stops again
because of the sigalrm_handler(int signum)
            }

        }

        else {
            alarm(SCHED_TQ_SEC);                   // set the alarm-counter to the time quantum
            kill(head->mypid,SIGCONT);            // so we send a SIGCONT to the head procces, so that it
continues from its pause
                                                    // and after the alarm time has passed, is stops again
because of the sigalrm_handler(int signum)
        }

    }

}


/* Disable delivery of SIGALRM and SIGCHLD. */
static void signals_disable(void) {

        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);

        //                    how=union    the set
        if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
                perror("signals_disable: sigprocmask");
                exit(1);
        }
}


/* Enable delivery of SIGALRM and SIGCHLD.     */
static void signals_enable(void) {

        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        //                          how=remove       the set
        if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
                perror("signals_enable: sigprocmask");
                exit(1);
        }
}


/* Install two signal handlers.
  * One for SIGCHLD, one for SIGALRM.
  * Make sure both signals are masked when one of them is running.
  */
static void install_signal_handlers(void) {

        sigset_t sigset;
        struct sigaction sa;

        sa.sa_handler = sigchld_handler;
        sa.sa_flags = SA_RESTART;
        sigemptyset(&sigset);
        sigaddset(&sigset, SIGCHLD);
        sigaddset(&sigset, SIGALRM);
        sa.sa_mask = sigset;
        if (sigaction(SIGCHLD, &sa, NULL) < 0) {
                perror("sigaction: sigchld");
                exit(1);
        }

        sa.sa_handler = sigalrm_handler;
        if (sigaction(SIGALRM, &sa, NULL) < 0) {
                perror("sigaction: sigalrm");
                exit(1);
        }

        /*
          * Ignore SIGPIPE, so that write()s to pipes
          * with no reader do not result in us being killed,
          * and write() returns EPIPE instead.
          */
        if (signal(SIGPIPE, SIG_IGN) < 0) {
                perror("signal: sigpipe");
                exit(1);
        }
}



static void do_shell(char *executable, int wfd, int rfd) {
        char arg1[10], arg2[10];
        char *newargv[] = { executable, NULL, NULL, NULL };
        char *newenviron[] = { NULL };

        // write to arg1,2 with format="%05d" from wfd, rfd
        sprintf(arg1, "%05d", wfd);
        sprintf(arg2, "%05d", rfd);
        newargv[1] = arg1;
        newargv[2] = arg2;

        raise(SIGSTOP);         // wait for the parent to send SIGCONT
        execve(executable, newargv, newenviron);        //execute the executable with arguments from array
pointed to by newargv

        /* execve() only returns on error */
        perror("scheduler: child: execve");
        exit(1);
}


/* Create a new shell task.
  *
  * The shell gets special treatment:
  * two pipes are created for communication and passed
  * as command-line arguments to the executable.
  */
static void sched_create_shell(char *executable, int *request_fd, int *return_fd) {

        pid_t p;
        int pfds_rq[2], pfds_ret[2];            // 2 pipes with these fds


        if (pipe(pfds_rq) < 0 | | pipe(pfds_ret) < 0) {
                perror("pipe");
                exit(1);
        }

        p = fork();
        if (p < 0) {
                perror("scheduler: fork");
                exit(1);
        }

        if (p == 0) {
                /* Child */
                close(pfds_rq[0]);
                close(pfds_ret[1]);

                do_shell(executable, pfds_rq[1], pfds_ret[0]);
                assert(0);
        }
        /* Parent */
        close(pfds_rq[1]);
        close(pfds_ret[0]);
        *request_fd = pfds_rq[0];
        *return_fd = pfds_ret[1];

        // head <- shell
        // last <- head
```

```
        head->serial_id = 0;
        head->mypid = p;
        head->name = "shell";
        head->next = NULL;
     head->priority = 0;



}


static void shell_request_loop(int request_fd, int return_fd) {

        int ret;
        struct request_struct rq;

        /*
          * Keep receiving requests from the shell.
          */
        for (;;) {

               //          from        to      size
                if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
                        perror("scheduler: read from shell");
                        fprintf(stderr, "Scheduler: giving up on shell request processing.\n");
                        break;
                }

                signals_disable();
                ret = process_request(&rq);          // apokwdikopoiei me vash to rq->request_no kai kalei thn
antistoixh func apo tis
                signals_enable();                      // sched_print_tasks, sched_kill_task_by_id,
sched_create_task


                //           to         from      size
                if (write(return_fd, &ret, sizeof(ret)) != sizeof(ret)) {
                        perror("scheduler: write to shell");
                        fprintf(stderr, "Scheduler: giving up on shell request processing.\n");
                        break;
                }
        }
}



int main(int argc, char *argv[]) {

        int nproc;

        /* Two file descriptors for communication with the shell */
        static int request_fd, return_fd;

        head=(struct process *)malloc(sizeof(struct process));
        last = head;

        /* Create the shell. */
        sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
        /* TODO: add the shell to the scheduler's tasks */
```

```
    /*
     * For each of argv[1] to argv[argc - 1],
     * create a new child process, add it to the process list.
     */

    nproc = argc-1; /* number of proccesses goes here */

    if (nproc==0) {
        fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
        exit(1);
    }

    char executable[] = "prog";
    char *newargv[] = {executable, NULL, NULL, NULL};
    char *newenviron[] = {NULL};

    pid_t my_pid;

    int i;
    for (i=0; i<nproc; i++) {

        my_pid = fork();

        if (my_pid<0) {
            printf("Error with fork\n");
        }

        if (my_pid == 0) {              // child's code

            raise(SIGSTOP);             // and wait for scheduler's SIGCONT
            printf("I am %s, PID = %ld\n", argv[0], (long)getpid());
            printf("About to replace myself with the executable %s...\n", executable);
            sleep(2);

            execve (executable, newargv, newenviron);
            //because execve only returns on error
            perror("execve");
            exit(1);

        }

        else {                          // father's code
                                          // the first time that this part is executed, the list has
only the shell on the head

            newnode = (struct process*) malloc(sizeof(struct process));

            if (newnode == NULL) {
                printf("Error with malloc\n");
            }

            newnode->mypid = my_pid;
            newnode->name = argv[i+1]; // i executable's name
            newnode->next = NULL;
            newnode->serial_id = i+1;
       newnode->priority = 0;

            last->next = newnode;
            last = newnode;
```

```
                }

        }

        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(nproc);

        /* Install SIGALRM and SIGCHLD handlers. */
        install_signal_handlers();


        alarm(SCHED_TQ_SEC);        // begin the alarm countdown

        kill(head->mypid, SIGCONT);          //continue the head process (shell)

        shell_request_loop(request_fd, return_fd);

        /* Now that the shell is gone, just loop forever
          * until we exit from inside a signal handler.
          */
        while (pause())
                ;

        /* Unreachable */
        fprintf(stderr, "Internal error: Reached unreachable point\n");
        return 1;
}
```

4.3.1.Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας.

Ένα σενάριο λιμοκτονίας για τον χρονοδρομολόγητή μας θα μπορούσε να είναι το εξής: Αν έχουμε μία ή περισσότερες διεργασίες οι οποίες έχουν Low Priority και δεν αλλάξει για αυτή/αυτές ποτέ η προτεραιότητα και πάντοτε έχουμε διαδικασίες με High Priority. Σε αυτό το ενδεχόμενο οι διαδικασίες με low priority σύμφωνα με τη λειτουργία του scheduler δε θα εκτελεστούν ποτέ ,αφού γνωρίζουμε πως όσο υπάρχουν διεργασίες με priority αυτές θα εκτελούνται πρώτες. Προφανώς σε μία τέτοια υλοποίηση σαν τη δική μας κάτι τέτοιο είναι υψηλά ανεπιθύμητο. Λύση στο πρόβλημα αυτό γενικά αποτελούν αλγόριθμοι που ελέγχουν αν μία διεργασία αναμένει την εκτέλεση της για μεγάλο χρονικό διάστημα. Στην περίπτωση μας , χρήσιμη θα ήταν η εισαγωγή μιας μεταβλητής ,έστω το όνομά της: timer, η οποία θα λειτουργούσε ως χρονόμετρο για τη συγκεκριμένη διεργασία. Με τη χρήση αυτής θα μπορούσαμε να μορφοποιήσουμε την υλοποίησή μας κατάλληλα ώστε ο timer να μην γίνεται να υπερβεί μια προκαθορισμένη τιμή από τον προγραμματιστή χωρίς να εκτελεστεί η διεργασία του. Αυτό θα συμβαίνει ανεξάρτητα του αν η διεργασία είναι high ή low priority και έτσι θα αντιμετωπιστεί αυτό το σενάριο λιμοκτονίας.