4η Ομάδα Ασκήσεων

ΣΤΑ ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Ομάδα Α38:

Ιωακειμίδη Αθηνά

Α.Μ.: 03114758

Μαυρομμάτης Ιάσων

Α.Μ.: 03114771

Εξάμηνο 7^{ο}

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εθνικό Μετσόβιο Πολυτεχνείο

Άσκηση 1

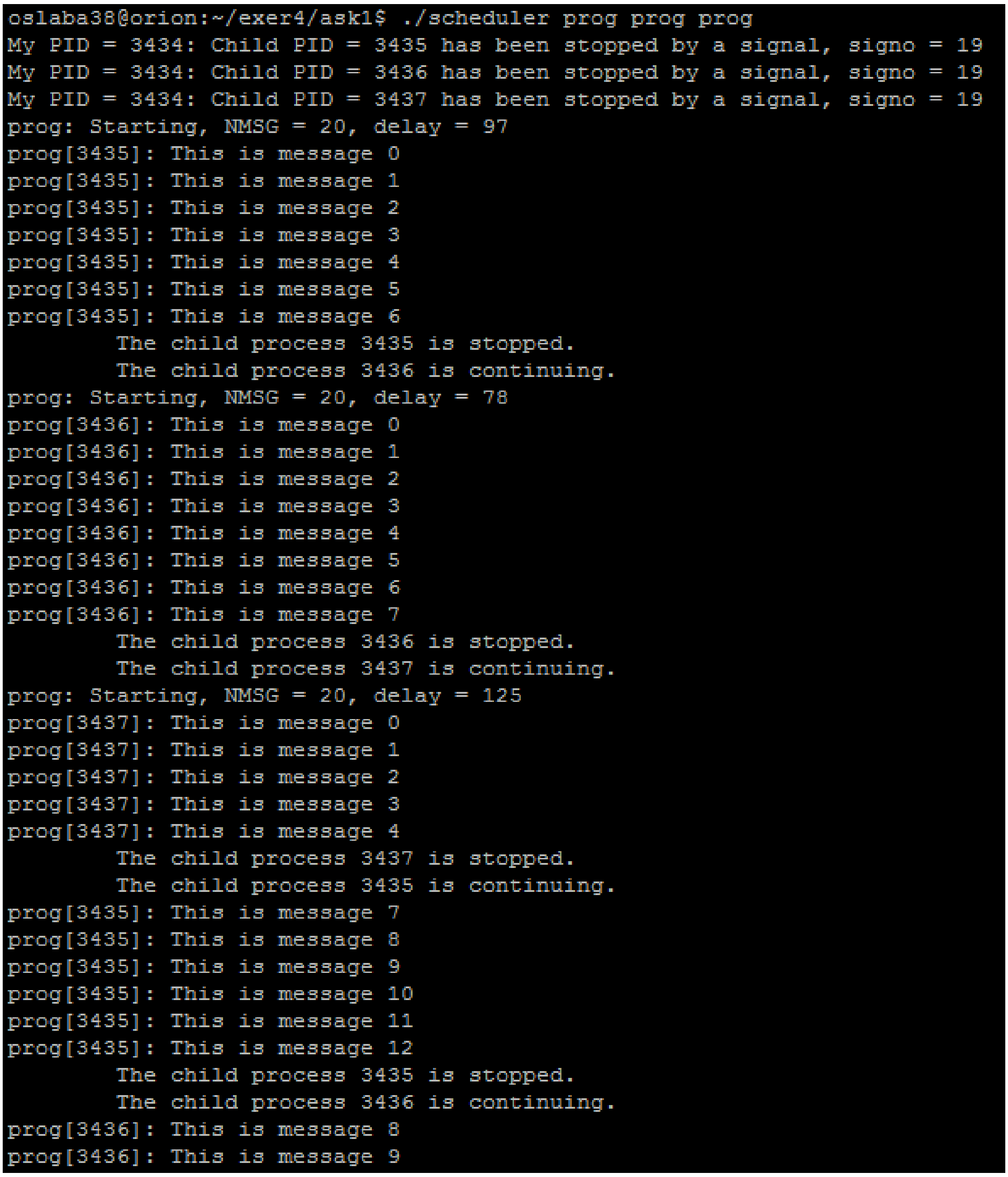
Πηγαίος κώδικας:

```
int * process;
int * active;
int nproc, current_proc;

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        kill(process[current_proc], SIGSTOP);
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        pid_t p;
        int i, status;

        for(;;) {
                p = waitpid(-1, &status, WUNTRACED | WNOHANG);

                if(p == 0) break;

                process[current_proc] = (int) p;

                if (WIFEXITED(status) || WIFSIGNALED(status)) {
                        printf("        The child process %d is
                                     terminated.\n", process[current_proc]);
                        active[current_proc] = 0;
                }

                if (WIFSTOPPED(status)) printf("        The child process
                                %d is stopped.\n", process[current_proc]);

                for (i=0; i<nproc; i++) if (active[i] == 1) break
                if (i == nproc) {
                        printf("        All child processes were
                                                        terminated.\n");
                        exit(0);
                }

                do {
                        current_proc = ((current_proc + 1) % nproc);
                } while (active[current_proc] == 0);

                printf("        The child process %d is continuing.\n",
                                                process[current_proc]);

                alarm(SCHED_TQ_SEC);
                kill(process[current_proc], SIGCONT);
        }
}
```

(Παρεμβάλλεται η `install_signal_handlers`, η οποία παραμένει ίδια με το αρχικό αρχείο και γι’ αυτό δεν παρατίθεται.)

```
int child(char *executable)
{
        char *newargv[] = { executable, NULL, NULL, NULL };
        char *newenviron[] = { NULL };

        raise(SIGSTOP);         //don't start unless said so

        execve(executable, newargv, newenviron);

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

int main(int argc, char *argv[])
{
        int i;
        pid_t p;

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

        nproc = argc-1;
        current_proc = 0;
        process = malloc(sizeof(int)*nproc);
        active = malloc(sizeof(int)*nproc);

        for (i=0; i<nproc; i++) {
                p = fork();
                if (p < 0) {perror("fork"); exit(1);}
                else if (p == 0) {
                        child(argv[i+1]);
                exit(1);
                } else {
                        process[i] = (int) p;       //add to process list
                        active[i] = 1;              //the process is active
                }
        }

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

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

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

        alarm(SCHED_TQ_SEC);
        kill(process[0], SIGCONT);      //start with process 0.

        /* loop forever  until we exit from inside a signal handler. */
        while (pause())
                ;

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

Έξοδος εκτέλεσης:



```
oslaba38@orion:~/exer4/ask1$ ./scheduler prog prog prog
My PID = 3434: Child PID = 3435 has been stopped by a signal, signo = 19
My PID = 3434: Child PID = 3436 has been stopped by a signal, signo = 19
My PID = 3434: Child PID = 3437 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 20, delay = 97
prog[3435]: This is message 0
prog[3435]: This is message 1
prog[3435]: This is message 2
prog[3435]: This is message 3
prog[3435]: This is message 4
prog[3435]: This is message 5
prog[3435]: This is message 6
        The child process 3435 is stopped.
        The child process 3436 is continuing.
prog: Starting, NMSG = 20, delay = 78
prog[3436]: This is message 0
prog[3436]: This is message 1
prog[3436]: This is message 2
prog[3436]: This is message 3
prog[3436]: This is message 4
prog[3436]: This is message 5
prog[3436]: This is message 6
prog[3436]: This is message 7
        The child process 3436 is stopped.
        The child process 3437 is continuing.
prog: Starting, NMSG = 20, delay = 125
prog[3437]: This is message 0
prog[3437]: This is message 1
prog[3437]: This is message 2
prog[3437]: This is message 3
prog[3437]: This is message 4
        The child process 3437 is stopped.
        The child process 3435 is continuing.
prog[3435]: This is message 7
prog[3435]: This is message 8
prog[3435]: This is message 9
prog[3435]: This is message 10
prog[3435]: This is message 11
prog[3435]: This is message 12
        The child process 3435 is stopped.
        The child process 3436 is continuing.
prog[3436]: This is message 8
prog[3436]: This is message 9
```

Απαντήσεις στις ερωτήσεις:

1. Εάν ένα σήμα SIGALRM έρθει ενώ εκτελείται η συνάρτηση χειρισμού του σήματος SIGCHLD, τότε το SIGALRM θα περιμένει έως ότου τελειώσει η εξυπηρέτηση του SIGCHLD και μετά θα εξυπηρετηθεί και αυτό. Αυτό συμβαίνει γιατί στη συνάρτηση `install_signal_handlers()` δημιουργείται μια μάσκα, η οποία αποτρέπει την εξυπηρέτηση δεύτερου σήματος όταν υπάρχει άλλο που εξυπηρετείται.
   Ένας πραγματικός χρονοδρομολογητής χώρου πυρήνα θα υλοποιούσε την ίδια λειτουργία σε επίπεδο hardware.
2. Κάθε σήμα SIGCHLD που λαμβάνει ο χρονοδρομολογητής αναφέρεται στη διεργασία-παιδί που τρέχει τη στιγμή λήψης του σήματος. Με τη λήψη του σήματος η `sigchld_handler()` θα βρει το pid της τρέχουσας διαδικασίας και θα δράσει αναλόγως.

Εάν μια διεργασία-παιδί τερματιστεί αναπάντεχα, τότε η `sigchld_handler()` θα την αφαιρέσει από τη λίστα των ενεργών διεργασιών.

3. Ο χειρισμός δύο σημάτων είναι απαραίτητος για την ορθή λειτουργία του προγράμματος γιατί έτσι εξασφαλίζουμε ότι η τρέχουσα διαδικασία έχει σταματήσει για να μπορούμε να ξεκινήσουμε την επόμενη. Στην υλοποίησή μας η έναρξη της καινούριας διαδικασίας γίνεται μετά τη λήψη του σήματος SIGCHLD που μας εγγυάται ότι η προηγούμενη διαδικασία έχει σταματήσει.
   Εάν χρησιμοποιούσαμε μόνο το σήμα SIGALRM τότε δεν θα μπορούσαμε να γνωρίζουμε αν ή πότε τερματίστηκε επιτυχώς η προηγούμενη διαδικασία και θα υπήρχε περίπτωση είτε το σύστημα να μένει άπραγο περιμένοντας είτε να τρέχουν δύο διαδικασίες παράλληλα.

Άσκηση 2

Πηγαίος κώδικας:

```
#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 */

typedef struct process_info
{
      pid_t PID;
      int id;
      char name[TASK_NAME_SZ];
      struct process_info * next;
} proc;

static int nproc;
static proc * extra_proc;
static proc * current_proc;

/* Insert a process to the process list. */
static void
ins_proc(int id, pid_t p, char * name)    {
    proc * extra_proc = (struct process_info *) malloc(sizeof(struct
process_info));

      extra_proc->id = id;
      extra_proc->PID = p;
      strcpy(extra_proc->name,name);

      extra_proc->next = current_proc->next;
      current_proc->next = extra_proc;
      current_proc = extra_proc;
}

/* Delete a process from the process list. */
static void
del_proc(int id)
{
      extra_proc = current_proc;
      while (extra_proc->next->id != id) extra_proc = extra_proc-
>next;

      extra_proc->next = extra_proc->next->next;
}

static void
```

```
child(char * executable)
{
        char * newargv[] = {executable, NULL, NULL, NULL};
        char * newenviron[] = {NULL};

        raise(SIGSTOP);         //don't start unless said so

        execve(executable, newargv, newenviron);

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

/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
        extra_proc = current_proc;

        printf("          Current process: PID: %d, id: %d, name:
%s\n", extra_proc->PID, extra_proc->id, extra_proc->name);

        while(extra_proc->next != current_proc){
                extra_proc = extra_proc->next;
                printf("          Process: PID: %d, id: %d, name: %s\n",
extra_proc->PID, extra_proc->id, extra_proc->name);
        }
}

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

        while (extra_proc->id != id){
                extra_proc = extra_proc->next;

                if(extra_proc == current_proc) return -1;
        }

        printf("          The child process %d with id=%d was
terminated.\n", extra_proc->PID, id);

        kill(extra_proc->PID, SIGKILL);
        return id;
}

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

        pid_t p = fork();
        if (p < 0) {perror("fork"); exit(1);}
        else if (p == 0) {
                child(executable);
        }
```

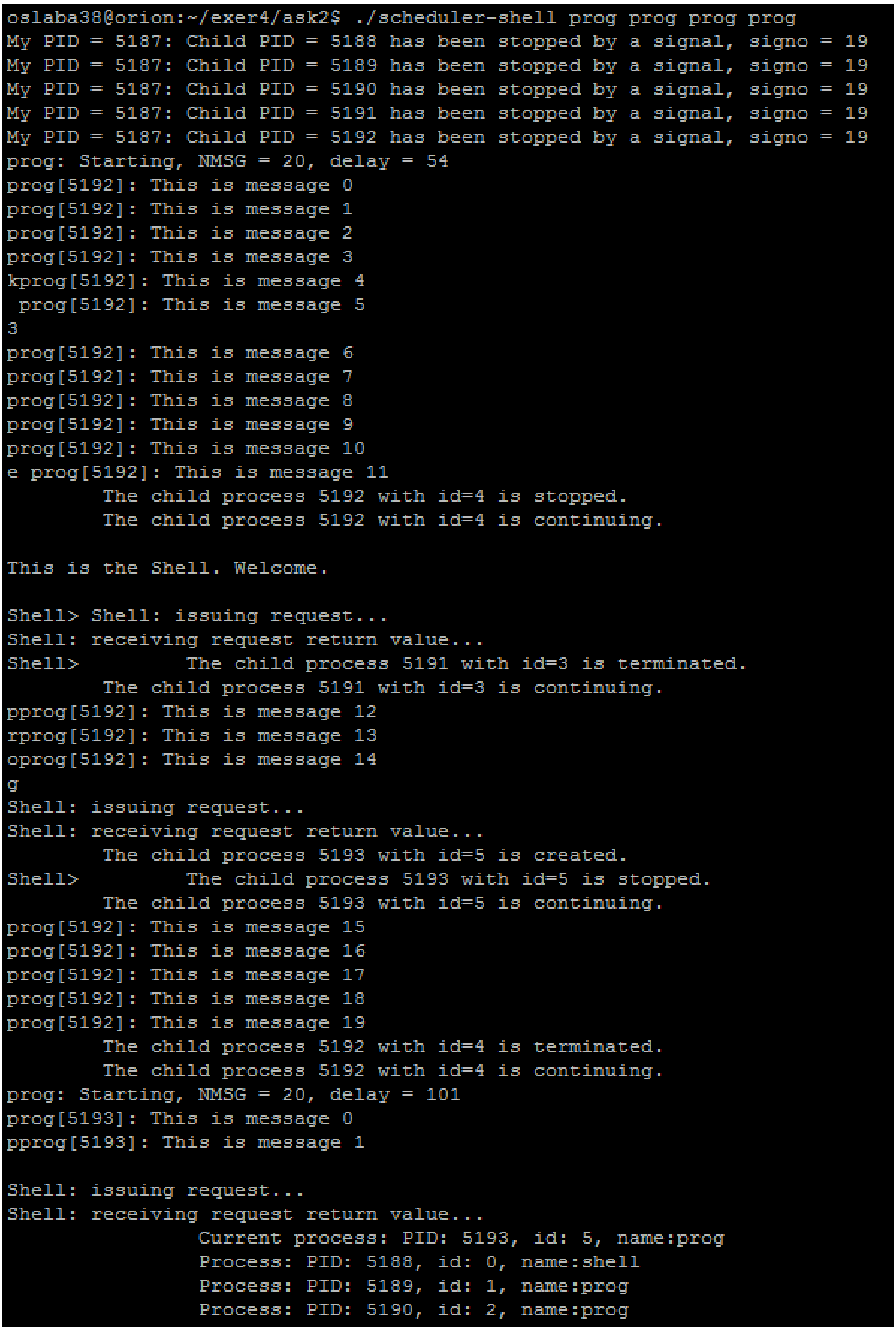
```
      else{
            printf("        The child process %d with id=%d is 
created.\n", p, nproc);
            ins_proc(nproc-1, p, executable);         //add to 
process list
      }
}

/* 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(current_proc->PID, SIGSTOP);       //your time is up, stop
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
      pid_t p;
      int status;

      for(;;){
            p = waitpid(-1, &status, WUNTRACED | WNOHANG);     //wait 
for the process that was lastly signaled or stopped

            if(p == 0) break;   //if you can't find any, break

            while(current_proc->PID != p) current_proc = 
current_proc->next;

            if (WIFEXITED(status)  || WIFSIGNALED(status)) {
                  printf("        The child process %d with id=%d is 
terminated.\n", current_proc->PID, current_proc->id);

                  if(current_proc->next == current_proc){
                        printf("        All child processes were
terminated.\n");
                        exit(0);
                  }

                  del_proc(current_proc->id);
      //delete process from process list
            }

            if(WIFSTOPPED(status)){
                  printf("        The child process %d with id=%d is
stopped.\n",  current_proc->PID, current_proc->id);        //SIGSTOP
            }

            printf("        The child process %d with id=%d is
continuing.\n", current_proc->PID, current_proc->id);

            current_proc = current_proc->next;
            alarm(SCHED_TQ_SEC);
            kill(current_proc->PID, SIGCONT);
      }
}

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

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

        sprintf(arg1, "%05d", wfd);
        sprintf(arg2, "%05d", rfd);
        newargv[1] = arg1;
        newargv[2] = arg2;

        raise(SIGSTOP);
        execve(executable, newargv, newenviron);

        /* 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];

        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 */
        current_proc=(struct process_info *) malloc(sizeof(struct
process_info));
        current_proc->PID = p;
        strcpy(current_proc->name, SHELL_EXECUTABLE_NAME);

        current_proc->next = current_proc;

        close(pfds_rq[1]);
        close(pfds_ret[0]);
        *request_fd = pfds_rq[0];
        *return_fd = pfds_ret[1];
}

static void
shell_request_loop(int request_fd, int return_fd)
{
        int ret;
        struct request_struct rq;

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

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

        /* 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.
         */

        pid_t p;
        int i;

        nproc = argc;

        for (i=1; i < nproc; i++){
                p = fork();

                if (p < 0) {perror("fork"); exit(1);}
                else if (p == 0){
                        child(argv[i]);
                }
                else{
                        ins_proc(i, p, argv[i]);            //add to
process list
                }
        }

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

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

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

        alarm(SCHED_TQ_SEC);
        kill(current_proc->PID, SIGCONT);

        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;
}
```

Έξοδος εκτέλεσης:

Η έξοδος της εκτέλεσης όταν ο χρήστης πληκτρολογεί τις παρακάτω εντολές:

```
k 3
e prog
p
```



```
oslaba38@orion:~/exer4/ask2$ ./scheduler-shell prog prog prog prog
My PID = 5187: Child PID = 5188 has been stopped by a signal, signo = 19
My PID = 5187: Child PID = 5189 has been stopped by a signal, signo = 19
My PID = 5187: Child PID = 5190 has been stopped by a signal, signo = 19
My PID = 5187: Child PID = 5191 has been stopped by a signal, signo = 19
My PID = 5187: Child PID = 5192 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 20, delay = 54
prog[5192]: This is message 0
prog[5192]: This is message 1
prog[5192]: This is message 2
prog[5192]: This is message 3
kprog[5192]: This is message 4
 prog[5192]: This is message 5
3
prog[5192]: This is message 6
prog[5192]: This is message 7
prog[5192]: This is message 8
prog[5192]: This is message 9
prog[5192]: This is message 10
e prog[5192]: This is message 11
        The child process 5192 with id=4 is stopped.
        The child process 5192 with id=4 is continuing.

This is the Shell. Welcome.

Shell> Shell: issuing request...
Shell: receiving request return value...
Shell>         The child process 5191 with id=3 is terminated.
        The child process 5191 with id=3 is continuing.
pprog[5192]: This is message 12
rprog[5192]: This is message 13
oprog[5192]: This is message 14
g
Shell: issuing request...
Shell: receiving request return value...
        The child process 5193 with id=5 is created.
Shell>         The child process 5193 with id=5 is stopped.
        The child process 5193 with id=5 is continuing.
prog[5192]: This is message 15
prog[5192]: This is message 16
prog[5192]: This is message 17
prog[5192]: This is message 18
prog[5192]: This is message 19
        The child process 5192 with id=4 is terminated.
        The child process 5192 with id=4 is continuing.
prog: Starting, NMSG = 20, delay = 101
prog[5193]: This is message 0
pprog[5193]: This is message 1

Shell: issuing request...
Shell: receiving request return value...
                Current process: PID: 5193, id: 5, name:prog
                Process: PID: 5188, id: 0, name:shell
                Process: PID: 5189, id: 1, name:prog
                Process: PID: 5190, id: 2, name:prog
```

Απαντήσεις στις ερωτήσεις:

1. Με την εντολή ‘p’ τυπώνεται στην οθόνη η λίστα με τις ενεργές διαδικασίες. Ως τρέχουσα διαδικασία εμφανίζεται ο φλοιός, αφού αυτός πάντα είναι που διαβάζει την εντολή του χρήστη και την εκτελεί.
   Δεν θα μπορούσε να μη συμβαίνει αυτό γιατί στην υλοποίησή μας ο φλοιός είναι μέσα στη λίστα των ενεργών διεργασιών.
2. Είναι απαραίτητο να απενεργοποιούμε τη λήψη σημάτων κατά την εκτέλεσή των εντολών που δίνει ο χρήστης στο φλοιό, γιατί αυτές οι εντολές μεταβάλουν την ουρά εκτέλεσης των διεργασιών. Στην περίπτωση που δεν το κάναμε αυτό θα υπήρχε πρόβλημα για παράδειγμα εάν κάποια διεργασία τερμάτιζε και ταυτόχρονα ο φλοιός προσπαθούσε να την σκοτώσει, ή αν προσπαθούσε να σκοτώσει την επόμενη από αυτήν.

Άσκηση 3

Πηγαίος κώδικας:

```
#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 */

typedef struct process_info
{
      pid_t PID;
      int id;
      char name[TASK_NAME_SZ];
      char priority;
      struct process_info * next;
} proc;

static int nproc;
static proc * extra_proc;
static proc * current_proc;

/* Insert a process to the process list. */
static void
ins_proc(int id, pid_t p, char * name)
{
    proc * extra_proc = (struct process_info *) malloc(sizeof(struct
process_info));

      extra_proc->id = id;
      extra_proc->PID = p;
      strcpy(extra_proc->name,name);
      extra_proc->priority = 'l';

      extra_proc->next = current_proc->next;
      current_proc->next = extra_proc;
      current_proc = extra_proc;
}

/* Delete a process from the process list. */
static void
del_proc(int id)
{
      extra_proc = current_proc;

      while (extra_proc->next->id != id) extra_proc = extra_proc-
>next;
```

```
        extra_proc->next = extra_proc->next->next;
}

static void
child(char * executable)
{
        char * newargv[] = {executable, NULL, NULL, NULL};
        char * newenviron[] = {NULL};

        raise(SIGSTOP);         //don't start unless said so

        execve(executable, newargv, newenviron);

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

/* Print a list of all tasks currently being scheduled. */
static void
sched_print_tasks(void)
{
        extra_proc = current_proc;

        char * prior="";
        if(extra_proc->priority == 'h') prior = "high";
        else prior = "low";

        printf("          Current process: PID: %d, id: %d, name: %s,
priority: %s\n", extra_proc->PID, extra_proc->id, extra_proc->name,
prior);

        while(extra_proc->next != current_proc){
                extra_proc = extra_proc->next;
                if(extra_proc->priority =='h') prior = "high";
                else prior = "low";

                printf("          Process: PID: %d, id: %d, name: %s,
priority: %s\n", extra_proc->PID, extra_proc->id, extra_proc->name,
prior);
        }
}

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

        while (extra_proc->id != id){
                extra_proc = extra_proc->next;

                if(extra_proc == current_proc) return -1;
        }

        printf("          The child process %d with id=%d was
terminated.\n", extra_proc->PID, id);

        kill(extra_proc->PID, SIGKILL);
        return id;
```

```
}

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

      pid_t p = fork();
      if (p < 0) {perror("fork"); exit(1);}
      else if (p == 0){
            child(executable);
      }
      else{
            printf("        The child process %d with id=%d is
created.\n", p, nproc-1);
            ins_proc(nproc-1, p, executable);
      }
}

static int
sched_high_priority(int id)
{
      extra_proc = current_proc;

      while (extra_proc->id != id){
            extra_proc = extra_proc->next;
            if(extra_proc == current_proc) return -1;
      }

      extra_proc->priority = 'h';
      printf("        The child process %d with id=%d now has high
priority.\n", extra_proc->PID, extra_proc->id);

      return(extra_proc->id);
}

static int
sched_low_priority(int id)
{
      extra_proc = current_proc;
      while (extra_proc->id != id){
            extra_proc = extra_proc->next;
            if(extra_proc == current_proc) return -1;
      }

      extra_proc->priority = 'l';
      printf("        The child process %d with id=%d now has low
priority.\n", extra_proc->PID, extra_proc->id);

      return(extra_proc->id);
}

/* 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:
                  return sched_high_priority(rq->task_arg);

            case REQ_LOW_TASK:
                  return sched_low_priority(rq->task_arg);

            default:
                  return -ENOSYS;
      }
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
      kill(current_proc->PID, SIGSTOP);      //your time is up, stop
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
      pid_t p;
      int status;

      for(;;){
            p = waitpid(-1, &status, WUNTRACED | WNOHANG);

            if(p == 0) break;

            while(current_proc->PID != p) current_proc =
current_proc->next;

            if (WIFEXITED(status)  || WIFSIGNALED(status)) {
                  printf("        The child process %d with id=%d is
terminated.\n", current_proc->PID, current_proc->id);

                  if(current_proc->next == current_proc){
                        printf("        All child processes were
terminated.\n");
                        exit(0);
                  }

                  del_proc(current_proc->id);
                  current_proc=current_proc->next;
            }

            if (WIFSTOPPED(status)){
                  printf("        The child process %d with id=%d is
stopped.\n",  current_proc->PID, current_proc->id);       //SIGSTOP
            }
            extra_proc = current_proc->next;
            while(extra_proc->priority != 'h'){
                  extra_proc = extra_proc->next;
                  if(extra_proc == current_proc){
                        if(current_proc->priority == 'h') break;
                        else {
                              extra_proc = extra_proc->next;
                              break;
                        }
                  }
            }

            current_proc = extra_proc;

            char * prior;
            if(current_proc->priority == 'l') prior = "low";
            else prior = "high";

            printf("        The child process %d with id=%d and
priority=%d is continuing.\n", current_proc->PID, current_proc->id,
prior);

            alarm(SCHED_TQ_SEC);
            kill(current_proc->PID, SIGCONT);
      }
}

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

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

      sprintf(arg1, "%05d", wfd);
      sprintf(arg2, "%05d", rfd);
      newargv[1] = arg1;
      newargv[2] = arg2;

      raise(SIGSTOP);
      execve(executable, newargv, newenviron);

      /* 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)
```

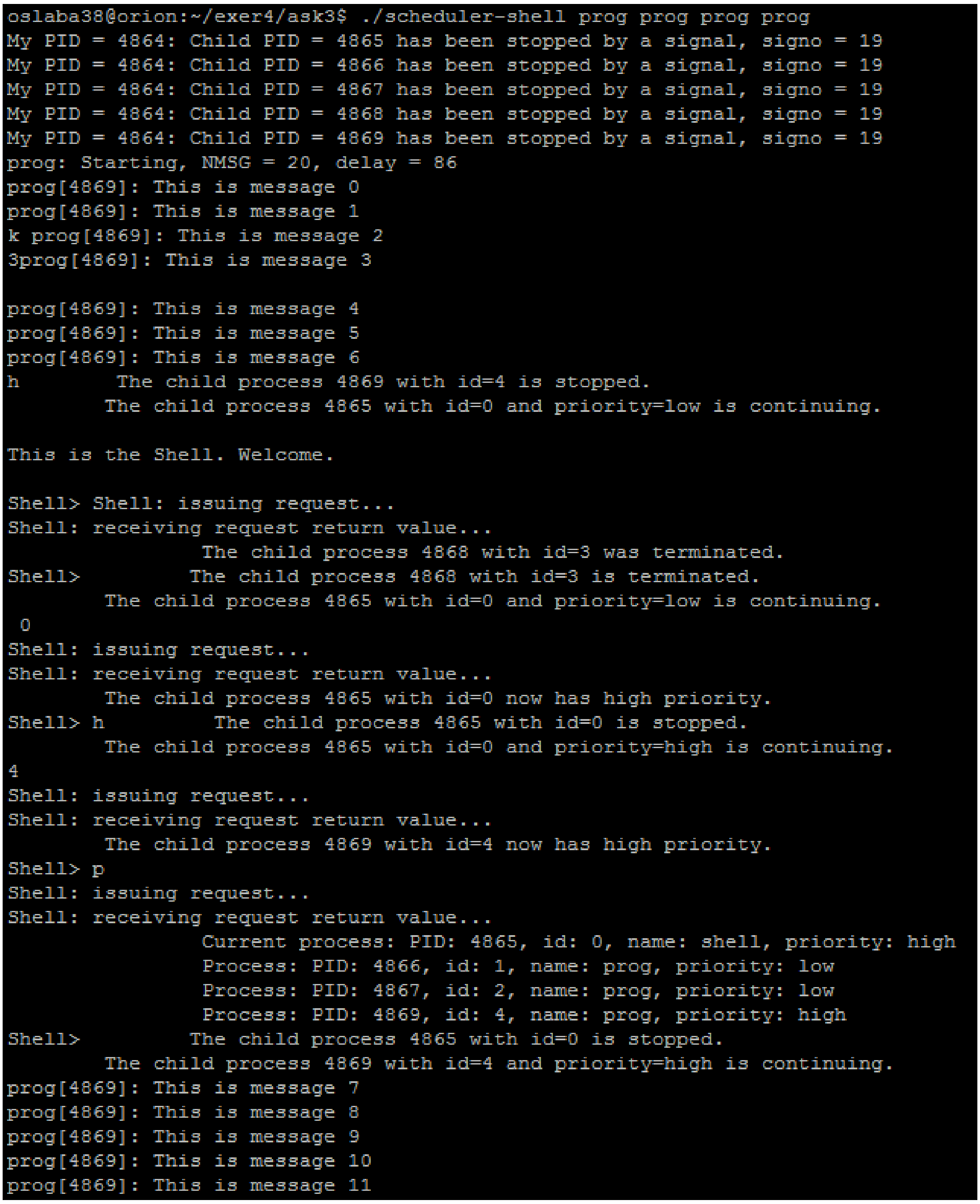
```
{
	pid_t p;
	int pfds_rq[2], pfds_ret[2];

	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 */
	current_proc = (struct process_info *) malloc(sizeof(struct
process_info));
	current_proc->PID = p;
	strcpy(current_proc->name, SHELL_EXECUTABLE_NAME);
	current_proc->priority = 'l';
	current_proc->next = current_proc;

	close(pfds_rq[1]);
	close(pfds_ret[0]);
	*request_fd = pfds_rq[0];
	*return_fd = pfds_ret[1];
}

static void
shell_request_loop(int request_fd, int return_fd)
{
	int ret;
	struct request_struct rq;

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

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

      /* 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.
       */

      pid_t p;
      int i;

      nproc = argc;

      for (i=1; i < nproc; i++){
            p = fork();
            if (p < 0) {perror("fork"); exit(1);}
            else if (p == 0){
                  child(argv[i]);
            }
            else{
                  ins_proc(i, p, argv[i]);
            }
      }

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

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

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

      alarm(SCHED_TQ_SEC);
      kill(current_proc->PID, SIGCONT);

      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;
}
```

Έξοδος εκτέλεσης:

Η έξοδος της εκτέλεσης όταν ο χρήστης πληκτρολογεί τις παρακάτω εντολες:

```
k 3
h 0
h 4
p
```



```
oslaba38@orion:~/exer4/ask3$ ./scheduler-shell prog prog prog prog
My PID = 4864: Child PID = 4865 has been stopped by a signal, signo = 19
My PID = 4864: Child PID = 4866 has been stopped by a signal, signo = 19
My PID = 4864: Child PID = 4867 has been stopped by a signal, signo = 19
My PID = 4864: Child PID = 4868 has been stopped by a signal, signo = 19
My PID = 4864: Child PID = 4869 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 20, delay = 86
prog[4869]: This is message 0
prog[4869]: This is message 1
k prog[4869]: This is message 2
3prog[4869]: This is message 3

prog[4869]: This is message 4
prog[4869]: This is message 5
prog[4869]: This is message 6
h        The child process 4869 with id=4 is stopped.
        The child process 4865 with id=0 and priority=low is continuing.

This is the Shell. Welcome.

Shell> Shell: issuing request...
Shell: receiving request return value...
                The child process 4868 with id=3 was terminated.
Shell>         The child process 4868 with id=3 is terminated.
        The child process 4865 with id=0 and priority=low is continuing.
 0
Shell: issuing request...
Shell: receiving request return value...
        The child process 4865 with id=0 now has high priority.
Shell> h         The child process 4865 with id=0 is stopped.
        The child process 4865 with id=0 and priority=high is continuing.
4
Shell: issuing request...
Shell: receiving request return value...
        The child process 4869 with id=4 now has high priority.
Shell> p
Shell: issuing request...
Shell: receiving request return value...
                Current process: PID: 4865, id: 0, name: shell, priority: high
                Process: PID: 4866, id: 1, name: prog, priority: low
                Process: PID: 4867, id: 2, name: prog, priority: low
                Process: PID: 4869, id: 4, name: prog, priority: high
Shell>         The child process 4865 with id=0 is stopped.
        The child process 4869 with id=4 and priority=high is continuing.
prog[4869]: This is message 7
prog[4869]: This is message 8
prog[4869]: This is message 9
prog[4869]: This is message 10
prog[4869]: This is message 11
```

Απαντήσεις στις ερωτήσεις:

1. Εάν υπάρχουν διεργασίες υψηλής προτεραιότητας που είτε παίρνουν πολύ χρόνο είτε είναι πολλές, τότε οι διεργασίες χαμηλής προτεραιότητας μπορεί να μην τρέξουν ποτέ. Αυτό θα ήταν καταστροφικό στην περίπτωση που οι διεργασίες χαμηλής προτεραιότητας αυτές ήταν απαραίτητες για την ομαλή λειτουργία του συστήματος.