Λειτουργικά Συστήματα – Άσκηση 4η

ΑΓΓΕΛΟΣ ΣΤΑΗΣ 03117435, ΣΩΚΡΑΤΗΣ ΠΟΥΤΑΣ 03117054 (Ομάδα Oslabb08)

Άσκηση 1.1 – Υλοποίηση χρονοδρομολογητή κυκλικής επαναφοράς στο χώρο χρήστη

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 EXEC_CHAR_LIM 10

struct Node {
      struct Node *next;
      int id;
      int pid;
      char name [EXEC_CHAR_LIM]; //onoma ektelesimou
};
struct Queue {
      struct Node *front;
      struct Node *last;
      unsigned int size;
      unsigned int next_id;
};

void init(struct Queue *q) {
      q->front = NULL;
      q->last = NULL;
      q->size = 0;
      q->next_id=1;
}
struct Node * front(struct Queue *q) {
      return q->front;
}
int next_id(struct Queue *q){
      return q->next_id;
}
void increase_next_id(struct Queue *q){
      q->next_id++;
}
void pop(struct Queue *q) {
      q->size--;
      struct Node *tmp = q->front;
      q->front = q->front->next;
      free(tmp);
}
void push(struct Queue *q, int id, int pid, char exe []) {
      q->size++;
      if (q->front == NULL) {  //If the queue was empty
            q->front = (struct Node *) malloc(sizeof(struct Node));
            q->front->id = id;
            q->front->pid= pid;
            strcpy(q->front->name,exe);
            q->front->next = NULL;
            q->last = q->front;
      }
      else {
            q->last->next = (struct Node *) malloc(sizeof(struct Node));
            q->last->next->id= id;
            q->last->next->next = NULL;
            q->last->next->pid= pid;
            strcpy(q->last->next->name,exe);
            q->last = q->last->next;
      }
}
void print_queue(struct Queue *q){
      struct Node *tmp = q->front;
      printf("Processes queue: ");
      while (tmp != NULL) {
            printf("%d (pid:%d, name:%s) --> ", tmp->id,tmp->pid,tmp->name);
            tmp = tmp->next;
      }
      printf("NULL\n");
}
//Pairnei gia orisma to pid tis diergasias pou tha afairethei
void remove_from_queue(struct Queue *q, int pid)
{
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
diagrafis
      if(curr != NULL)
      {
            if(curr->pid == pid){//node to be removed is front
                  q->front = q->front->next;
                  free(curr);
                  q->size--;
                  return;
            }
      }
      struct Node *prev = q->front;
      curr = curr->next;
      while(curr != NULL)
      {
            if(curr->pid == pid)
                  break;
            else{
```

```c
                  curr = curr->next;
                  prev = prev->next;
            }
      }
      if(curr == NULL) return;
      else{ //Brethike o komvos diagrafis
            prev->next = curr->next;
            curr->next = NULL;
            free(curr);
            q->size--;
            //next 3 lines are to restore q->last
            struct Node *tmp = q->front;
            while(tmp->next != NULL) tmp = tmp->next;
            q->last = tmp;
            return;
      }
}
int empty(struct Queue *q) {
      return q->size == 0;
}
//Euresi tou pid tis diergasias apo to id
int find_pid(struct Queue *q, int id)
{
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
      if(curr != NULL){
            if(curr->id == id){//node is front
                  return curr->pid;
            }
      }
      curr = curr->next;
      while(curr != NULL){
            if(curr->id == id) break;
            else curr = curr->next;
      }
      if(curr == NULL) return -1;
      else//Brethike o komvos
            return curr->pid;
}

struct Node * find_node_from_pid(struct Queue *q, int search_pid)
{
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
      while(curr != NULL){
            if(curr->pid == search_pid) break;
            else curr = curr->next;
      }
      if(curr == NULL) return NULL;
      else//Brethike o komvos
            return curr;
}

struct Queue q;

/*
 * SIGALRM handler
 */
//Stamataei ti diergasia kefali tis ouras
```

```
static void
sigalrm_handler(int signum)
{
      // printf("alarm handler used\n");
      pid_t to_stop = (front(&q)->pid);
      printf("Scheduler: Stopping (id:%d, pid:%d, name:%s).\n",
                  front(&q)->id,front(&q)->pid,front(&q)->name);
      kill(to_stop, SIGSTOP);
}

/*
 * SIGCHLD handler
 */
//Kalitai kathe fora pou lambanetai sima termatismou-pausis opoioudipote paidiou
static void
sigchld_handler(int signum)
{
      // printf("sigchld handler used\n");
      pid_t p;
      int status;

      if (signum != SIGCHLD) {
            fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n",
                  signum);
            exit(1);
      }

      /*
       * Something has happened to one of the children.
       * We use waitpid() with the WUNTRACED flag, instead of wait(), because
       * SIGCHLD may have been received for a stopped, not dead child.
       *
       * A single SIGCHLD may be received if many processes die at the same time.
       * We use waitpid() with the WNOHANG flag in a loop, to make sure all
       * children are taken care of before leaving the handler.
       */

      for (;;) {
            p = waitpid(-1, &status, WUNTRACED | WNOHANG);
            if (p < 0) {
                  perror("waitpid");
                  exit(1);
            }
            if (p == 0)
                  break;

            explain_wait_status(p, status);
            // if (WIFEXITED(status) || WIFSIGNALED(status))
            if (WIFEXITED(status)) {
                  /* A child has died normally */
                  printf("Scheduler: (id:%d, pid:%d, name:%s) terminated
normally.\n",
                  find_node_from_pid(&q,p)->id,find_node_from_pid(&q,p)-
>pid,find_node_from_pid(&q,p)->name);
                  pop(&q); // remove child from queue
            }
            if(WIFSIGNALED(status)){
                  //a child has died by signal
                  printf("Scheduler: (id:%d, pid:%d, name:%s) has been killed.\n",
                              find_node_from_pid(&q,p)-
>id,find_node_from_pid(&q,p)->pid,find_node_from_pid(&q,p)->name);
                  if(p != (front(&q)->pid)){
                        remove_from_queue(&q, p);
                        if(empty(&q)){
                              fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                              exit(1);
                        }
                        return;
                  }
            }
            if (WIFSTOPPED(status)) {
                  /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
                  if(p==front(&q)->pid){
                        push(&q,front(&q)->id,front(&q)->pid,front(&q)->name);
                        pop(&q); // remove process from queue
                  //move stopped process at the end of the queue
                  }
            }
            if(empty(&q)){
                  fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                  exit(1);
            }
            alarm(SCHED_TQ_SEC); //set alarm
            printf("Scheduler: Activating (id:%d, pid:%d, name:%s).\n",
                        front(&q)->id,front(&q)->pid,front(&q)->name);
            kill((front(&q)->pid), SIGCONT); //send SIGCONT to next process in
queue
      }
}

/* 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; //a collection of values for signals
                               //that are used in various system calls.
      struct sigaction sa; //Orizei to struct sigaction sa
      sa.sa_handler = sigchld_handler; //Periptosi SIGCHLD
      sa.sa_flags = SA_RESTART;
      sigemptyset(&sigset); //Creates empty sigset
      sigaddset(&sigset, SIGCHLD);
      sigaddset(&sigset, SIGALRM);
      sa.sa_mask = sigset; //O handler blockarei pros to paron ta simata SIGCHILD
kai SIGALRM otan ekteleitai
      if (sigaction(SIGCHLD, &sa, NULL) < 0) //Gia lipsi simatos SIGCHLD ektelese
to handler mas
      {
            perror("sigaction: sigchld");
            exit(1);
      }
      sa.sa_handler = sigalrm_handler; //Periptosi SIGALRM
      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;
      init(&q); //Dimiourgei mia keni oura
      /*
       * For each of argv[1] to argv[argc - 1],
       * create a new child process, add it to the process list.
       */
      //Einai ta programmata pou tha xronodromologithoun
       int i;
       for(i = 1; i < argc; i++)
       {
             char *executable = argv[i];
             char *newargv[] = { argv[i], NULL, NULL, NULL }; //oi parametroi pou
tha parei to execve,

//me to proto na einai to onoma tou ektelesimou
             char *newenviron[] = { NULL };
             pid_t p = fork();      //gia kathe programma pou dinetai san orisma
dimiourgei mia diergasia
             if(p < 0){
                   perror("error: fork");
                   exit(1);
             }
             else if(p == 0){ //an eisai i diergasia paidi
                   raise(SIGSTOP); //child process raises SIGSTOP immediately
                                          //prota dimiourgountai oles oi diergasies
                                           //Otan dothei SIGCONT i kathe mia tha
sinexisei apo edo
                   execve(executable, newargv, newenviron);
                   perror("execve");
                   exit(1);
             }
             else{
                   push(&q,next_id(&q),p,executable); //Scheduler pushes child's
pid to queue
                   increase_next_id(&q);
             }
       }
      nproc = argc - 1; /* number of processes created*/
                                //argc-1 giati i proti parametros einai to onoma 
tou arxeiou
      /* Wait for all children to raise SIGSTOP before exec()ing. */
      wait_for_ready_children(nproc); //perimenei na termatisoun ola ta paidia
      printf("All initialized correctly. Please proceed.\n");
      print_queue(&q);

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

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

      alarm(SCHED_TQ_SEC); //set an alarm for SCHED_TQ_SEC
      printf("Scheduler: Activating (id:%d, pid:%d, name:%s).\n",front(&q)-
>id,front(&q)->pid,front(&q)->name);
      kill((front(&q)->pid), SIGCONT); //send SIGCONT to first process
      //energopoioume ti proti diergasia gia na arxisei tin ektelesi tis

      // loop forever  until we exit from inside a signal handler.
      while (pause())
            ;
      // Unreachable
      fprintf(stderr, "Internal error: Reached unreachable point\n");
      return 1;
}
```

Εντολή Εξόδου Εκτέλεσης `./scheduler prog prog prog`

Έξοδος Εκτέλεσης

```
My PID = 7757: Child PID = 7758 has been stopped by a signal, signo = 19
My PID = 7757: Child PID = 7759 has been stopped by a signal, signo = 19
My PID = 7757: Child PID = 7760 has been stopped by a signal, signo = 19
All initialized correctly. Please proceed.
Processes queue: 1 (pid:7758, name:prog) --> 2 (pid:7759, name:prog) --> 3 
(pid:7760, name:prog) --> NULL
Scheduler: Activating (id:1, pid:7758, name:prog).
prog: Starting, NMSG = 10, delay = 67
prog[7758]: This is message 0
prog[7758]: This is message 1
prog[7758]: This is message 2
prog[7758]: This is message 3
prog[7758]: This is message 4
Scheduler: Stopping (id:1, pid:7758, name:prog).
My PID = 7757: Child PID = 7758 has been stopped by a signal, signo = 19
Scheduler: Activating (id:2, pid:7759, name:prog).
prog: Starting, NMSG = 10, delay = 49
prog[7759]: This is message 0
prog[7759]: This is message 1
prog[7759]: This is message 2
prog[7759]: This is message 3
prog[7759]: This is message 4
prog[7759]: This is message 5
```

```
prog[7759]: This is message 6
Scheduler: Stopping (id:2, pid:7759, name:prog).
My PID = 7757: Child PID = 7759 has been stopped by a signal, signo = 19
Scheduler: Activating (id:3, pid:7760, name:prog).
prog: Starting, NMSG = 10, delay = 31
prog[7760]: This is message 0
prog[7760]: This is message 1
prog[7760]: This is message 2
prog[7760]: This is message 3
prog[7760]: This is message 4
prog[7760]: This is message 5
prog[7760]: This is message 6
prog[7760]: This is message 7
prog[7760]: This is message 8
prog[7760]: This is message 9
Scheduler: Stopping (id:3, pid:7760, name:prog).
My PID = 7757: Child PID = 7760 has been stopped by a signal, signo = 19
Scheduler: Activating (id:1, pid:7758, name:prog).
prog[7758]: This is message 5
^C
```

Ερωτήσεις

1) Στη περίπτωση που έρθει ένα σήμα SIGALRM ενώ εκτελείται η συνάρτηση χειρισμού του σήματος SIGCHLD ή το αντίστροφο το σήμα που θα έρθει θα χαθεί δηλαδή δε θα το πιάσει ποτέ η διεργασία. Αυτό συμβαίνει καθώς στη συνάρτηση που εγκαθιστά του αντίστοιχους χειριστές σημάτων install_signal_handlers ορίζεται μάσκα σημάτων που περιλαμβάνει τα σήματα SIGCHLD και SIGALRM τα οποία θα μπλοκαριστούν κατά την εκτέλεση των handlers.
Ένας πραγματικός χρονοδρομολογητής εύλογα φροντίζει να αποκλειστεί ένα τέτοιο ενδεχόμενο χρησιμοποιώντας διακοπές χρονιστή και όχι σήματα ή πιθανόν εξασφαλίζοντας ότι τα σχετικά σήματα μπαίνουν σε μια ουρά και λαμβάνονται αμέσως μόλις τελειώσει η συνάρτηση χειρισμού του προηγούμενου σήματος ώστε να μη δημιουργούνται απροσδόκητες συμπεριφορές από τη μη λήψη σημάτων.

2) Κάθε φορά που ο χρονοδρομολογητής λαμβάνει ένα σήμα SIGCHLD περιμένουμε αυτό να αναφέρεται στη διεργασία-παιδί που στο προηγούμενο κβάντο χρόνου είχε χρονοδρομολογηθεί καθώς το πρόγραμμα που χρησιμοποιείται για εκτέλεση δεν τερματίζει ποτέ τη λειτουργία του. Αν έρθει ένα σήμα τερματισμού άλλης διεργασίας τότε ο χρονοδρομολογητής λαμβάνει σήμα SIGCHLD (αφού η συνάρτηση χειρισμού των σημάτων έχει υλοποιηθεί να πιάνει τέτοιου είδους σήματα για οποιοδήποτε παιδί) από τη διεργασία που τερματίστηκε ενημερώνοντας με το κατάλληλο μήνυμα ενώ η διεργασία που έχουμε βάλει να χρονοδρομολογογηθεί εκτελείται κανονικά.
Παρατίθεται και η έξοδος από παράδειγμα σχετικής εκτέλεσης.

```
Scheduler: Activating (id:2, pid:7940, name:prog).
prog[7940]: This is message 17
```

```
prog[7940]: This is message 18
My PID = 7938: Child PID = 7939 was terminated by a signal, signo = 9
Scheduler: (id:1, pid:7939, name:prog) has been killed.
prog[7940]: This is message 19
prog[7940]: This is message 20
Scheduler: Stopping (id:2, pid:7940, name:prog).
My PID = 7938: Child PID = 7940 has been stopped by a signal, signo = 19
```

3) Ο χειρισμός δύο σημάτων είναι απαραίτητος για την υλοποίηση του χρονοδρομολογητή προκειμένου να μπορεί να υλοποιηθεί η περίπτωση μια διεργασία να ολοκληρώσει την εκτέλεση της σε χρόνο μικρότερο από το κβάντο χρόνου. Σε αυτή τη περίπτωση θα πρέπει ο χρονοδρομολογητής να ενημερωθεί από το κατάλληλο σήμα για να ξεκινήσει την εκτέλεση της επόμενης διεργασίας στην ουρά διεργασιών. Αν χρησιμοποιούνταν μόνο το SIGALARM σήμα ο χρονοδρομολογητής δε θα άρχιζε την εκτέλεση της επόμενης διεργασίας μέχρι τη λήξη του κβάντου χρόνου καταναλώνοντας άσκοπα υπολογιστικούς κύκλους της ΚΜΕ.
Ακόμη σε περίπτωση χρήσης μόνο ενός σήματος SIGALRM δε διασφαλίζεται ότι η διεργασία που πρέπει να σταματήσει να εκτελείται επειδή εξέπνευσε το κβάντο χρόνου θα λάβει το σήμα SIGSTOP, κάτι για το οποίο μας ενημερώνει η παραλαβή του σήματος SIGCHLD. Οπότε ένα πιθανό ανεπιθύμητο σενάριο θα ήταν να μην έχει ενημερωθεί ο χρονοδρομολογητής για το μη τερματισμό της παλιάς διεργασίας στέλνοντας σήμα συνέχισης για μα νέα.
Τέλος ένα ακόμα ανεπιθύμητο σενάριο θα ήταν να μην ενημερωθεί ο χρονοδρομολογητής για τον απροσδόκητο τερματισμό κάποιας διεργασίας λόγω εξωτερικού παράγοντα (σήματος) ενώ εκτελείται κάποια άλλη διεργασία και να μην ενημερώσει κατάλληλα τις δομές του (πχ ουρά διεργασιών).

Άσκηση 1.2 - Έλεγχος Λειτουργίας Χρονοδρομολογητή μέσω Φλοιού

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 */
#define EXEC_CHAR_LIM 10

struct Node {
      struct Node *next;
      int id;
      int pid;
      char name [EXEC_CHAR_LIM]; //onoma ektelesimou
};
struct Queue {
      struct Node *front;
      struct Node *last;
      unsigned int size;
      unsigned int next_id;
};

void init(struct Queue *q) {
      q->front = NULL;
      q->last = NULL;
      q->size = 0;
      q->next_id=1;
}
 struct Node * front(struct Queue *q) {
      return q->front;
}
int next_id(struct Queue *q){
      return q->next_id;
}
void increase_next_id(struct Queue *q){
      q->next_id++;
}
void pop(struct Queue *q) {
      q->size--;
      struct Node *tmp = q->front;
      q->front = q->front->next;
      free(tmp);
}
void push(struct Queue *q, int id, int pid, char exe []) {
      q->size++;
      if (q->front == NULL) {  //If the queue was empty
            q->front = (struct Node *) malloc(sizeof(struct Node));
            q->front->id = id;
            q->front->pid= pid;
            strcpy(q->front->name,exe);
            q->front->next = NULL;
            q->last = q->front;
      } else {
            q->last->next = (struct Node *) malloc(sizeof(struct Node));
            q->last->next->id= id;
            q->last->next->next = NULL;
            q->last->next->pid= pid;
            strcpy(q->last->next->name,exe);
            q->last = q->last->next;
      }
}
void print_queue(struct Queue *q){
      struct Node *tmp = q->front;
      printf("Processes queue: ");
      while (tmp != NULL) {
            printf("%d (pid:%d, name:%s) --> ", tmp->id,tmp->pid,tmp->name);
            tmp = tmp->next;
      }
      printf("NULL\n");
}
//Pairnei gia orisma to pid tis diergasias pou tha afairethei
void remove_from_queue(struct Queue *q, int pid)
{
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
diagrafis
      if(curr != NULL)
      {
            if(curr->pid == pid){//node to be removed is front
                  q->front = q->front->next;
                  free(curr);
                  q->size--;
                  return;
            }
      }
      struct Node *prev = q->front;
      curr = curr->next;
      while(curr != NULL)
      {
            if(curr->pid == pid)
                  break;
            else{
                  curr = curr->next;
                  prev = prev->next;
            }
      }
      if(curr == NULL) return;
      else{ //Brethike o komvos diagrafis
            prev->next = curr->next;
            curr->next = NULL;
            free(curr);
            q->size--;
            //next 3 lines are to restore q->last
            struct Node *tmp = q->front;
            while(tmp->next != NULL) tmp = tmp->next;
            q->last = tmp;
            return;
      }
}
int empty(struct Queue *q) {
      return q->size == 0;
}
//Euresi tou pid tis diergasias apo to id
int find_pid(struct Queue *q, int id)
{
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
      if(curr != NULL){
            if(curr->id == id){//node is front
                  return curr->pid;
            }
      }
      curr = curr->next;
      while(curr != NULL){
            if(curr->id == id) break;
            else curr = curr->next;
      }
      if(curr == NULL) return -1;
      else//Brethike o komvos
            return curr->pid;
}

struct Node * find_node_from_pid(struct Queue *q, int search_pid)
{
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
      while(curr != NULL){
            if(curr->pid == search_pid) break;
            else curr = curr->next;
      }
      if(curr == NULL) return NULL;
      else//Brethike o komvos
            return curr;
}

struct Queue q;

/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
      struct Node *tmp = (&q)->front;
      printf("\nScheduler: Printing tasks.\nCurrent task (id:%d, pid:%d,
name:%s)\n",
                  tmp->id,tmp->pid,tmp->name);
      //Sigoura tha typothei ena task, to shell
      tmp = tmp->next;
      if(tmp!=NULL)
      {
            printf("Rest of the tasks:\n");
            while (tmp != NULL)
            {
                  printf("(id:%d, pid:%d, name:%s)\n",tmp->id,tmp->pid,tmp->name);
                  tmp = tmp->next;
            }
      }
            printf("End of tasks.\n\n");
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id) //Prepei na brethei to pid
{
      int pid;
      pid=find_pid(&q,id);
      struct Node * curr=find_node_from_pid(&q,pid);
      if (curr!=NULL)
            printf("Scheduler: Killing (id:%d, pid:%d, name:%s).\n",
                  curr->id,curr->pid,curr->name);
      else {
            printf("Scheduler: Process not found.\n");
            return 0;
      }
      return kill(pid, SIGKILL);
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
      char *newargv[] = { executable, NULL, NULL, NULL };
      char *newenviron[] = { NULL };
      pid_t p = fork();
      if(p < 0){
            perror("error: fork");
            exit(1);
      }
      else if(p == 0){
            raise(SIGSTOP); //child process raises SIGSTOP immediately
            execve(executable, newargv, newenviron);
            perror("execve");
            exit(1);
      }
      else{
            printf("Scheduler: Creating (id:%d, pid:%d, name:%s).\n",
                  next_id(&q),p,executable);
            push(&q,next_id(&q),p,executable); //Scheduler pushes child's pid to
queue
            increase_next_id(&q);
      }

}

/* 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)
{
      // printf("alarm handler used\n");
      pid_t to_stop = (front(&q)->pid);
      printf("Scheduler: Stopping (id:%d, pid:%d, name:%s).\n",
                  front(&q)->id,front(&q)->pid,front(&q)->name);
      kill(to_stop, SIGSTOP);
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        // printf("sigchld handler used\n");
        pid_t p;
        int status;

        if (signum != SIGCHLD) {
                fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n",
                        signum);
                exit(1);
        }

        /*
         * Something has happened to one of the children.
         * We use waitpid() with the WUNTRACED flag, instead of wait(), because
         * SIGCHLD may have been received for a stopped, not dead child.
         *
         * A single SIGCHLD may be received if many processes die at the same time.
         * We use waitpid() with the WNOHANG flag in a loop, to make sure all
         * children are taken care of before leaving the handler.
         */

        for (;;) {
                p = waitpid(-1, &status, WUNTRACED | WNOHANG);
                if (p < 0) {
                        perror("waitpid");
                        exit(1);
                }
                if (p == 0)
                        break;

                explain_wait_status(p, status);
                // if (WIFEXITED(status) || WIFSIGNALED(status))
                if (WIFEXITED(status)) {
                        /* A child has died normally */
                        printf("Scheduler: (id:%d, pid:%d, name:%s) terminated
normally.\n",
                        find_node_from_pid(&q,p)->id,find_node_from_pid(&q,p)-
>pid,find_node_from_pid(&q,p)->name);
                        pop(&q); // remove child from queue
                }
                if(WIFSIGNALED(status)){
                        //a child has died by signal
                        printf("Scheduler: (id:%d, pid:%d, name:%s) has been killed.\n",
                                        find_node_from_pid(&q,p)-
>id,find_node_from_pid(&q,p)->pid,find_node_from_pid(&q,p)->name);
                        if(p != (front(&q)->pid)){
```

```
                        remove_from_queue(&q, p);
                        if(empty(&q)){
                              fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                              exit(1);
                        }
                        return;
                  }
            }
            if (WIFSTOPPED(status)) {
                  /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
                  if(p==front(&q)->pid){
                        push(&q,front(&q)->id,front(&q)->pid,front(&q)->name);
                        pop(&q); // remove process from queue
                  //move stopped process at the end of the queue
                  }
            }
            if(empty(&q)){
                  fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                  exit(1);
            }
            alarm(SCHED_TQ_SEC); //set alarm
            printf("Scheduler: Activating (id:%d, pid:%d, name:%s).\n",
                        front(&q)->id,front(&q)->pid,front(&q)->name);
            kill((front(&q)->pid), SIGCONT); //send SIGCONT to next process in
queue
      }
}

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

      push(&q,next_id(&q),p,"shell"); //Scheduler adds shell to queue
      increase_next_id(&q);
}

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[])
{
      int nproc;
      /* Two file descriptors for communication with the shell */
      static int request_fd, return_fd;
      init(&q);
      /* Create the shell. */
      sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
      /* TODO: add the shell to the scheduler's tasks */
      //Done inside sched_create_shell

      /*
       * For each of argv[1] to argv[argc - 1],
       * create a new child process, add it to the process list.
       */
       int i; //arithmos programmaton pou tha eisaxthoun sto scheduler
       for(i = 1; i < argc; i++)
       {
            char *executable = argv[i];
            char *newargv[] = { argv[i], NULL, NULL, NULL };
            char *newenviron[] = { NULL };
            pid_t p = fork();
            if(p < 0){
                  perror("error: fork");
                  exit(1);
            }
            else if(p == 0){
                  raise(SIGSTOP); //child process raises SIGSTOP immediately
                  execve(executable, newargv, newenviron);
                  perror("execve");
                  exit(1);
            }
            else{
                  //printf("I' m going to push process (id:%d, pid:%d, name:%s)
into the queue.\n",i+1,p,executable);
                  push(&q,next_id(&q),p,executable); //Scheduler pushes child to
queue
                  increase_next_id(&q);
            }
       }

       nproc = argc; /* number of processes goes here */

      /* 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);
      }

      print_queue(&q);
      printf("All initialized correctly. Please proceed.\n");

      alarm(SCHED_TQ_SEC); //set an alarm for SCHED_TQ_SEC
      printf("Scheduler: Activating (id:%d, pid:%d, name:%s).\n",
                  front(&q)->id,front(&q)->pid,front(&q)->name);
      kill((front(&q)->pid), SIGCONT); //send SIGCONT to first process
      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;
}
```

Εντολή Εξόδου Εκτέλεσης `./scheduler-shell prog prog`

Σημείωση: Για το παράδειγμα εκτέλεσης στο πρόγραμμα prog που χρονοδρομολογείται οι εντολές printf έχουν αφαιρεθεί ώστε να είναι πιο ευανάγνωστη η έξοδος εκτέλεσης. Στον επισυναπτόμενο κώδικα το prog.c είναι το αρχικό που δίνεται.

Έξοδος Εκτέλεσης

```
My PID = 26450: Child PID = 26451 has been stopped by a signal, signo = 19
My PID = 26450: Child PID = 26452 has been stopped by a signal, signo = 19
My PID = 26450: Child PID = 26453 has been stopped by a signal, signo = 19
Processes queue: 1 (pid:26451, name:shell) --> 2 (pid:26452, name:prog) --> 3
(pid:26453, name:prog) --> NULL
All initialized correctly. Please proceed.
Scheduler: Activating (id:1, pid:26451, name:shell).

This is the Shell. Welcome.

Shell> p
Shell: issuing request...
Shell: receiving request return value...

Scheduler: Printing tasks.
Current task (id:1, pid:26451, name:shell)
Rest of the tasks:
(id:2, pid:26452, name:prog)
(id:3, pid:26453, name:prog)
End of tasks.

Shell> Scheduler: Stopping (id:1, pid:26451, name:shell).
My PID = 26450: Child PID = 26451 has been stopped by a signal, signo = 19
Scheduler: Activating (id:2, pid:26452, name:prog).
Scheduler: Stopping (id:2, pid:26452, name:prog).
My PID = 26450: Child PID = 26452 has been stopped by a signal, signo = 19
Scheduler: Activating (id:3, pid:26453, name:prog).
```

```
e prog
Scheduler: Stopping (id:3, pid:26453, name:prog).
My PID = 26450: Child PID = 26453 has been stopped by a signal, signo = 19
Scheduler: Activating (id:1, pid:26451, name:shell).
Shell: issuing request...
Shell: receiving request return value...
Scheduler: Creating (id:4, pid:26454, name:prog).
Shell> My PID = 26450: Child PID = 26454 has been stopped by a signal, signo = 19
Scheduler: Activating (id:1, pid:26451, name:shell).
p
Shell: issuing request...
Shell: receiving request return value...

Scheduler: Printing tasks.
Current task (id:1, pid:26451, name:shell)
Rest of the tasks:
(id:2, pid:26452, name:prog)
(id:3, pid:26453, name:prog)
(id:4, pid:26454, name:prog)
End of tasks.

Shell> Scheduler: Stopping (id:1, pid:26451, name:shell).
My PID = 26450: Child PID = 26451 has been stopped by a signal, signo = 19
Scheduler: Activating (id:2, pid:26452, name:prog).
Scheduler: Stopping (id:2, pid:26452, name:prog).
My PID = 26450: Child PID = 26452 has been stopped by a signal, signo = 19
Scheduler: Activating (id:3, pid:26453, name:prog).
k 2
Scheduler: Stopping (id:3, pid:26453, name:prog).
My PID = 26450: Child PID = 26453 has been stopped by a signal, signo = 19
Scheduler: Activating (id:4, pid:26454, name:prog).
Scheduler: Stopping (id:4, pid:26454, name:prog).
My PID = 26450: Child PID = 26454 has been stopped by a signal, signo = 19
Scheduler: Activating (id:1, pid:26451, name:shell).
Shell: issuing request...
Shell: receiving request return value...
Scheduler: Killing (id:2, pid:26452, name:prog).
Shell> My PID = 26450: Child PID = 26452 was terminated by a signal, signo = 9
Scheduler: (id:2, pid:26452, name:prog) has been killed.
e prog
Shell: issuing request...
Shell: receiving request return value...
Scheduler: Creating (id:5, pid:26459, name:prog).
Shell> My PID = 26450: Child PID = 26459 has been stopped by a signal, signo = 19
Scheduler: Activating (id:1, pid:26451, name:shell).
Scheduler: Stopping (id:1, pid:26451, name:shell).
My PID = 26450: Child PID = 26451 has been stopped by a signal, signo = 19
Scheduler: Activating (id:3, pid:26453, name:prog).
p
```

```
Scheduler: Stopping (id:3, pid:26453, name:prog).
My PID = 26450: Child PID = 26453 has been stopped by a signal, signo = 19
Scheduler: Activating (id:4, pid:26454, name:prog).
Scheduler: Stopping (id:4, pid:26454, name:prog).
My PID = 26450: Child PID = 26454 has been stopped by a signal, signo = 19
Scheduler: Activating (id:5, pid:26459, name:prog).
Scheduler: Stopping (id:5, pid:26459, name:prog).
My PID = 26450: Child PID = 26459 has been stopped by a signal, signo = 19
Scheduler: Activating (id:1, pid:26451, name:shell).
Shell: issuing request...
Shell: receiving request return value...

Scheduler: Printing tasks.
Current task (id:1, pid:26451, name:shell)
Rest of the tasks:
(id:3, pid:26453, name:prog)
(id:4, pid:26454, name:prog)
(id:5, pid:26459, name:prog)
End of tasks.

Shell> Scheduler: Stopping (id:1, pid:26451, name:shell).
My PID = 26450: Child PID = 26451 has been stopped by a signal, signo = 19
Scheduler: Activating (id:3, pid:26453, name:prog).
My PID = 26450: Child PID = 26453 terminated normally, exit status = 0
Scheduler: (id:3, pid:26453, name:prog) terminated normally.
Scheduler: Activating (id:4, pid:26454, name:prog).
^C
```

Ερωτήσεις

1) Όταν και ο φλοιός υφίσταται χρονοδρομολόγηση τρέχουσα διεργασία στη λίστα διεργασιών με την εντολή ’p’ εμφανίζεται η διεργασία του φλοιού. Αυτό συμβαίνει καθώς για να εκτελεστεί το αίτημα της εντολής p θα πρέπει να περιμένουμε έρθει η σειρά του φλοιού να χρονοδρομολογηθεί. Επομένως, αυτό δεν μπορεί να αποφευχθεί αφού σε κάθε περίπτωση τυπώματος της ουράς διεργασιών, τρέχουσα είναι η διεργασία που παίρνει την εντολή για το εν λόγω τύπωμα, στην περίπτωσή μας ο φλοιός.

2) Οι κλήσεις signals_disable(), _enable() πρέπει να συμπεριληφθούν στη συνάρτηση υλοποίησης αιτήσεων του φλοιού ώστε να είμαστε σίγουροι ότι οι αλλαγές στη διαμοιραζόμενη ουρά εκτέλεσης των διεργασιών θα γίνουν χωρίς να διακοπούν από κάποιο σήμα. Πιθανή διακοπή της επεξεργασίας της ουράς από τη λήψη ενός σήματος SIGALARM ή SIGCHLD μπορεί να δημιουργήσει πρόβλημα στην ουρά και την εκτέλεση του χρονοδρομολογητή. Ουσιαστικά λειτουργούν ως κλείδωμα για να εκτελεστεί ο κώδικας process_request σαν κρίσιμο τμήμα.

Άσκηση 1.3 - Υλοποίηση Προτεραιοτήτων στο Χρονοδρομολογητή

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 */
#define EXEC_CHAR_LIM 10

struct Node {
      struct Node *next;
      int id;
      int pid;
      char priority;
      int is_shell;
      char name [EXEC_CHAR_LIM]; //onoma ektelesimou
};
struct Queue {
      struct Node *front;
      struct Node *last;
      unsigned int size;
      unsigned int next_id;
      unsigned int has_high;
};

void init(struct Queue *q) {
      q->front = NULL;
      q->last = NULL;
      q->size = 0;
      q->next_id=1;
      q->has_high=0;
}
int next_id(struct Queue *q){
      return q->next_id;
}
void increase_next_id(struct Queue *q){
      q->next_id++;
}
 struct Node * front(struct Queue *q) {
      return q->front;
}
void pop(struct Queue *q) {
      q->size--;
      struct Node *tmp = q->front;
      if(tmp->priority == 'h' && tmp->is_shell == 0) q->has_high--;
      q->front = q->front->next;
      free(tmp);
}
void push(struct Queue *q, int id, int pid, char exe [],char priority, int
is_shell) {
      q->size++;
      if (q->front == NULL) {  //If the queue was empty
            q->front = (struct Node *) malloc(sizeof(struct Node));
            q->front->id = id;
            q->front->pid= pid;
            q->front->priority = priority;
            q->front->is_shell = is_shell;
            strcpy(q->front->name,exe);
            q->front->next = NULL;
            q->last = q->front;
      } else {
            q->last->next = (struct Node *) malloc(sizeof(struct Node));
            q->last->next->id= id;
            q->last->next->next = NULL;
            q->last->next->pid= pid;
            q->last->next->priority = priority;
            q->last->next->is_shell = is_shell;
            strcpy(q->last->next->name,exe);
            q->last = q->last->next;
      }
      if(priority == 'h' && is_shell == 0) q->has_high++;
}
void print_queue(struct Queue *q){
      struct Node *tmp = q->front;
      printf("Processes queue: ");
      while (tmp != NULL) {
            printf("%d (pid: %d, name: %s, priority: %c) --> ", tmp->id,tmp-
>pid,tmp->name,tmp->priority);
            tmp = tmp->next;
      }
      printf("NULL\n");
}
//Pairnei gia orisma to pid tis diergasias pou tha afairethei
```

```
void remove_from_queue(struct Queue *q, int pid)
{
      // printf("target data = %d\n", data);
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
diagrafis
      if(curr != NULL)
      {
            if(curr->pid == pid){//node to be removed is front
                  if(q->front->priority == 'h') q->has_high--;
                  q->front = q->front->next;
                  free(curr);
                  q->size--;
                  return;
            }
      }
      struct Node *prev = q->front;
      curr = curr->next;
      while(curr != NULL)
      {
            if(curr->pid == pid)
                  break;
            else{
                  curr = curr->next;
                  prev = prev->next;
            }
      }
      if(curr == NULL) return;
      else{ //Brethike o komvos diagrafis
            if(curr->priority == 'h') q->has_high--;
            prev->next = curr->next;
            curr->next = NULL;
            free(curr);
            q->size--;
            //next 3 lines are to restore q->last
            struct Node *tmp = q->front;
            while(tmp->next != NULL) tmp = tmp->next;
            q->last = tmp;
            return;
      }
}
int empty(struct Queue *q) {
      return q->size == 0;
}
//Euresi tou pid tis diergasias apo to id
int find_pid(struct Queue *q, int id)
{
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
      if(curr != NULL){
            if(curr->id == id){//node is front
```

```
                  return curr->pid;
            }
      }
      curr = curr->next;
      while(curr != NULL){
            if(curr->id == id) break;
            else curr = curr->next;
      }
      if(curr == NULL) return -1;
      else//Brethike o komvos
            return curr->pid;
}

struct Node * find_node_from_pid(struct Queue *q, int search_pid)
{
      struct Node *curr = q->front; //Arxizo apo to front kai psaxno to kombo
      while(curr != NULL){
            if(curr->pid == search_pid) break;
            else curr = curr->next;
      }
      if(curr == NULL) return NULL;
      else//Brethike o komvos
            return curr;
}

struct Queue q;

/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
      struct Node *tmp = (&q)->front;
      printf("\nScheduler: Printing tasks.\n");
      printf("High priority tasks are:\n");
      printf("Current task (id: %d, pid: %d, name: %s, priority: %c)\n",tmp-
>id,tmp->pid,tmp->name,tmp->priority);
      //Sigoura tha typothei ena task, to shell
      tmp = tmp->next;
      if(tmp!=NULL)
      {
            printf("Rest of high priority tasks:\n");
            while (tmp != NULL) {
                  if(tmp->priority == 'h')
                        printf("(id: %d, pid: %d, name: %s, priority: %c)\n",
                                    tmp->id,tmp->pid,tmp->name,tmp->priority);
                  tmp = tmp->next;
            }
      }
      tmp = (&q)->front;
      printf("Low priority tasks:\n");
            while (tmp != NULL) {
                  if(tmp->priority == 'l')
                        printf("(id: %d, pid: %d, name: %s, priority: %c)\n",
                              tmp->id,tmp->pid,tmp->name,tmp->priority);
                  tmp = tmp->next;
            }
            printf("End of Tasks.\n\n");
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id) //Prepei na brethei to pid
{
      int pid;
      pid=find_pid(&q,id);
      struct Node * curr=find_node_from_pid(&q,pid);
      if (curr!=NULL)
            printf("Scheduler: Killing (id:%d, pid:%d, name:%s, priority:%c).\n",
                  curr->id,curr->pid,curr->name,curr->priority);
      else {
            printf("Scheduler: Process not found.\n");
            return 0;
      }
      return kill(pid, SIGKILL);
}

/* Create a new task.  */
static void
sched_create_task(char *executable)
{
      char *newargv[] = { executable, NULL, NULL, NULL };
      char *newenviron[] = { NULL };
      pid_t p = fork();
      if(p < 0){
            perror("error: fork");
            exit(1);
      }
      else if(p == 0){
            raise(SIGSTOP); //child process raises SIGSTOP immediately
            execve(executable, newargv, newenviron);
            perror("execve");
            exit(1);
      }
      else{
            printf("Scheduler: Creating (id:%d, pid:%d, name:%s, priority:l).\n",
                  next_id(&q),p,executable);
            push(&q,next_id(&q),p,executable,'l',0); //Scheduler pushes child's pid 
to queue
            increase_next_id(&q);
      }
}


/* Promote a task to high priority class */
static void
sched_high_task(int search_id)
{
      struct Node *tmp = (&q)->front;
      while(tmp != NULL){
            if(tmp->id== search_id) break;
            tmp = tmp->next;
      }
      if(tmp == NULL) return;
      if(tmp->priority == 'l') {
            (&q)->has_high++;
            printf("Scheduler: Promoting (id:%d, pid:%d, name:%s, priority:%c) to 
high priority.\n",
                        tmp->id,tmp->pid,tmp->name,tmp->priority);
            tmp->priority = 'h';
      }
}

/* Demote a task to low priority class */
static void
sched_low_task(int search_id)
{
      struct Node *tmp = (&q)->front;
      while(tmp != NULL){
            if(tmp->id == search_id) break;
            tmp = tmp->next;
      }
      if(tmp == NULL) return;
      if(tmp->priority == 'h') {
            (&q)->has_high--;
            printf("Scheduler: Demoting (id:%d, pid:%d, name:%s, priority:%c) to 
low priority.\n",
                        tmp->id,tmp->pid,tmp->name,tmp->priority);
            tmp->priority = 'l';
      }
}
/* 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_high_task(rq->task_arg);
                  return 0;
            case REQ_LOW_TASK:
                  sched_low_task(rq->task_arg);
                  return 0;

            default:
                  return -ENOSYS;
      }
}


/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
      // printf("alarm handler used\n");
      pid_t to_stop = (front(&q)->pid);
      printf("Scheduler: Stopping (id:%d, pid:%d, name:%s, priority:%c).\n",
                  front(&q)->id,front(&q)->pid,front(&q)->name,front(&q)-
>priority);
      kill(to_stop, SIGSTOP);
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
      // printf("sigchld handler used\n");
      pid_t p;
      int status;

      if (signum != SIGCHLD) {
            fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n",
                  signum);
            exit(1);
      }

      /*
       * Something has happened to one of the children.
       * We use waitpid() with the WUNTRACED flag, instead of wait(), because
       * SIGCHLD may have been received for a stopped, not dead child.
       *
       * A single SIGCHLD may be received if many processes die at the same time.
       * We use waitpid() with the WNOHANG flag in a loop, to make sure all
       * children are taken care of before leaving the handler.
       */

      for (;;) {
            p = waitpid(-1, &status, WUNTRACED | WNOHANG);
            if (p < 0) {
                  perror("waitpid");
                  exit(1);
            }
            if (p == 0)
                  break;

            explain_wait_status(p, status);

            if (WIFEXITED(status)) {
                  printf("Scheduler: (id:%d, pid:%d, name:%s, priority:%c)
terminated normally.\n",
                              find_node_from_pid(&q,p)-
>id,find_node_from_pid(&q,p)->pid,
                              find_node_from_pid(&q,p)-
>name,find_node_from_pid(&q,p)->priority);
                  pop(&q); // remove child from queue
            }
            if(WIFSIGNALED(status)){
                  //a child has died by signal
                  printf("Scheduler: (id:%d, pid:%d, name:%s, priority:%c) has been
killed.\n",
                              find_node_from_pid(&q,p)-
>id,find_node_from_pid(&q,p)->pid,
                              find_node_from_pid(&q,p)-
>name,find_node_from_pid(&q,p)->priority);
                  if(p != (front(&q)->pid)){
                        remove_from_queue(&q, p);
                        if(empty(&q)){
                              fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                              exit(1);
                        }
                        return;
                  }
            }
            if (WIFSTOPPED(status)) {
                  /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
                  if(p==front(&q)->pid){
                        push(&q,front(&q)->id,front(&q)->pid,front(&q)->name,
                        front(&q)->priority,front(&q)->is_shell);
                        pop(&q); // remove process from queue
                  //move stopped process at the end of the queue
                  }
            }
            printf("Number of high processes except Shell is %d.\n", (&q)-
>has_high);
            if((&q)->has_high){
                  while((&q)->front->priority == 'l'){ //rotate queue
                        push(&q,front(&q)->id,front(&q)->pid,front(&q)->name,
                                    (&q)->front->priority, (&q)->front->is_shell);
                        pop(&q);
                  }
            }
            if(empty(&q)){
                  fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                  exit(1);
            }
            alarm(SCHED_TQ_SEC); //set alarm
            printf("Scheduler: Activating (id:%d, pid:%d, name:%s,
priority:%c).\n",
                        front(&q)->id,front(&q)->pid,front(&q)->name,front(&q)-
>priority);
            kill((front(&q)->pid), SIGCONT); //send SIGCONT to next process in
queue
      }
}

/* 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) {
```

```c
		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 */
      close(pfds_rq[1]);
      close(pfds_ret[0]);
      *request_fd = pfds_rq[0];
      *return_fd = pfds_ret[1];

      push(&q,next_id(&q),p,"shell", 'h',1); //Scheduler adds shell to queue
      increase_next_id(&q);
}

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[])
{
      int nproc;
      /* Two file descriptors for communication with the shell */
      static int request_fd, return_fd;
      init(&q);
      printf("Number of high processes except Shell is %d.\n", (&q)->has_high);
      /* Create the shell. */
      sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
      /* TODO: add the shell to the scheduler's tasks */
      //Done inside sched_create_shell

      /*
       * For each of argv[1] to argv[argc - 1],
       * create a new child process, add it to the process list.
       */
       int i;
       for(i = 1; i < argc; i++)
       {
            char *executable = argv[i];
            char *newargv[] = { argv[i], NULL, NULL, NULL };
            char *newenviron[] = { NULL };
            pid_t p = fork();
            if(p < 0){
                  perror("error: fork");
                  exit(1);
            }
            else if(p == 0){
                  raise(SIGSTOP); //child process raises SIGSTOP immediately
                  execve(executable, newargv, newenviron);
                  perror("execve");
                  exit(1);
            }
            else{
                  push(&q,next_id(&q),p,executable, 'l', 0); //Scheduler pushes
child's pid to queue
                  increase_next_id(&q);
             }
       }

       nproc = argc; /* number of processes goes here */

      /* 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);
      }
      print_queue(&q);
      printf("All initialized correctly. Please proceed\n");

      alarm(SCHED_TQ_SEC); //set an alarm for SCHED_TQ_SEC
      printf("Scheduler: Activating (id:%d, pid:%d, name:%s, priority:%c).\n",
                  front(&q)->id,front(&q)->pid,front(&q)->name,front(&q)-
>priority);
      kill((front(&q)->pid), SIGCONT); //send SIGCONT to first process

      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;
}
```

Εντολή Εξόδου Εκτέλεσης `./scheduler-shell prog prog`

Σημείωση: Για το παράδειγμα εκτέλεσης στο πρόγραμμα prog που χρονοδρομολογείται οι εντολές printf έχουν αφαιρεθεί ώστε να είναι πιο ευανάγνωστη η έξοδος εκτέλεσης. Στον επισυναπτόμενο κώδικα το prog.c είναι το αρχικό που δίνεται.

Έξοδος Εκτέλεσης

```
Number of high processes except Shell is 0.
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
My PID = 27375: Child PID = 27377 has been stopped by a signal, signo = 19
My PID = 27375: Child PID = 27378 has been stopped by a signal, signo = 19
Processes queue: 1 (pid: 27376, name: shell, priority: h) --> 2 (pid: 27377, name:
prog, priority: l) --> 3 (pid: 27378, name: prog, priority: l) --> NULL
All initialized correctly. Please proceed
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).

This is the Shell. Welcome.

Shell> Scheduler: Stopping (id:1, pid:27376, name:shell, priority:h).
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:2, pid:27377, name:prog, priority:l).
Scheduler: Stopping (id:2, pid:27377, name:prog, priority:l).
My PID = 27375: Child PID = 27377 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:3, pid:27378, name:prog, priority:l).
```

```
e prog
Scheduler: Stopping (id:3, pid:27378, name:prog, priority:l).
My PID = 27375: Child PID = 27378 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
Shell: issuing request...
Shell: receiving request return value...
Scheduler: Creating (id:4, pid:27379, name:prog, priority:l).
Shell> My PID = 27375: Child PID = 27379 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
p
Shell: issuing request...
Shell: receiving request return value...

Scheduler: Printing tasks.
High priority tasks are:
Current task (id: 1, pid: 27376, name: shell, priority: h)
Rest of high priority tasks:
Low priority tasks:
(id: 2, pid: 27377, name: prog, priority: l)
(id: 3, pid: 27378, name: prog, priority: l)
(id: 4, pid: 27379, name: prog, priority: l)
End of Tasks.

Shell> Scheduler: Stopping (id:1, pid:27376, name:shell, priority:h).
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:2, pid:27377, name:prog, priority:l).
Scheduler: Stopping (id:2, pid:27377, name:prog, priority:l).
My PID = 27375: Child PID = 27377 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:3, pid:27378, name:prog, priority:l).
Scheduler: Stopping (id:3, pid:27378, name:prog, priority:l).
My PID = 27375: Child PID = 27378 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:4, pid:27379, name:prog, priority:l).
h 3
Scheduler: Stopping (id:4, pid:27379, name:prog, priority:l).
My PID = 27375: Child PID = 27379 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
Shell: issuing request...
Shell: receiving request return value...
Scheduler: Promoting (id:3, pid:27378, name:prog, priority:l) to high priority.
Shell> p
Shell: issuing request...
Shell: receiving request return value...
```

```
Scheduler: Printing tasks.
High priority tasks are:
Current task (id: 1, pid: 27376, name: shell, priority: h)
Rest of high priority tasks:
(id: 3, pid: 27378, name: prog, priority: h)
Low priority tasks:
(id: 2, pid: 27377, name: prog, priority: l)
(id: 4, pid: 27379, name: prog, priority: l)
End of Tasks.

Shell> Scheduler: Stopping (id:1, pid:27376, name:shell, priority:h).
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
Number of high processes except Shell is 1.
Scheduler: Activating (id:3, pid:27378, name:prog, priority:h).
k 2
Scheduler: Stopping (id:3, pid:27378, name:prog, priority:h).
My PID = 27375: Child PID = 27378 has been stopped by a signal, signo = 19
Number of high processes except Shell is 1.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
Shell: issuing request...
Shell: receiving request return value...
Scheduler: Killing (id:2, pid:27377, name:prog, priority:l).
Shell> My PID = 27375: Child PID = 27377 was terminated by a signal, signo = 9
Scheduler: (id:2, pid:27377, name:prog, priority:l) has been killed.
p
Shell: issuing request...
Shell: receiving request return value...

Scheduler: Printing tasks.
High priority tasks are:
Current task (id: 1, pid: 27376, name: shell, priority: h)
Rest of high priority tasks:
(id: 3, pid: 27378, name: prog, priority: h)
Low priority tasks:
(id: 4, pid: 27379, name: prog, priority: l)
End of Tasks.

Shell> Scheduler: Stopping (id:1, pid:27376, name:shell, priority:h).
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
Number of high processes except Shell is 1.
Scheduler: Activating (id:3, pid:27378, name:prog, priority:h).
Scheduler: Stopping (id:3, pid:27378, name:prog, priority:h).
My PID = 27375: Child PID = 27378 has been stopped by a signal, signo = 19
Number of high processes except Shell is 1.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
Scheduler: Stopping (id:1, pid:27376, name:shell, priority:h).
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
Number of high processes except Shell is 1.
Scheduler: Activating (id:3, pid:27378, name:prog, priority:h).
```

```
k 3
Scheduler: Stopping (id:3, pid:27378, name:prog, priority:h).
My PID = 27375: Child PID = 27378 has been stopped by a signal, signo = 19
Number of high processes except Shell is 1.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
Shell: issuing request...
Shell: receiving request return value...
Scheduler: Killing (id:3, pid:27378, name:prog, priority:h).
Shell> My PID = 27375: Child PID = 27378 was terminated by a signal, signo = 9
Scheduler: (id:3, pid:27378, name:prog, priority:h) has been killed.
p
Shell: issuing request...
Shell: receiving request return value...

Scheduler: Printing tasks.
High priority tasks are:
Current task (id: 1, pid: 27376, name: shell, priority: h)
Rest of high priority tasks:
Low priority tasks:
(id: 4, pid: 27379, name: prog, priority: l)
End of Tasks.

Shell> Scheduler: Stopping (id:1, pid:27376, name:shell, priority:h).
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:4, pid:27379, name:prog, priority:l).
Scheduler: Stopping (id:4, pid:27379, name:prog, priority:l).
My PID = 27375: Child PID = 27379 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
Scheduler: Stopping (id:1, pid:27376, name:shell, priority:h).
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:4, pid:27379, name:prog, priority:l).
Scheduler: Stopping (id:4, pid:27379, name:prog, priority:l).
My PID = 27375: Child PID = 27379 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
Scheduler: Stopping (id:1, pid:27376, name:shell, priority:h).
My PID = 27375: Child PID = 27376 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:4, pid:27379, name:prog, priority:l).
q
Scheduler: Stopping (id:4, pid:27379, name:prog, priority:l).
My PID = 27375: Child PID = 27379 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:1, pid:27376, name:shell, priority:h).
Shell: Exiting. Goodbye.
My PID = 27375: Child PID = 27376 terminated normally, exit status = 0
```

```
Scheduler: (id:1, pid:27376, name:shell, priority:h) terminated normally.
Number of high processes except Shell is 0.
Scheduler: Activating (id:4, pid:27379, name:prog, priority:l).
scheduler: read from shell: Success
Scheduler: giving up on shell request processing.
Scheduler: Stopping (id:4, pid:27379, name:prog, priority:l).
My PID = 27375: Child PID = 27379 has been stopped by a signal, signo = 19
Number of high processes except Shell is 0.
Scheduler: Activating (id:4, pid:27379, name:prog, priority:l).
Scheduler: Stopping (id:4, pid:27379, name:prog, priority:l).
^C
```

Ερωτήσεις

1) Ένα σενάριο δημιουργίας λιμοκτονίας είναι να εισαχθούν δύο διεργασίες στο χρονοδρομολογητή με χαμηλή προτεραιότητα και η μια από αυτές να αναβαθμιστεί σε υψηλή προτεραιότητα και να εκτελείται συνεχώς. Σε αυτή τη περίπτωση η διεργασία με χαμηλή προτεραιότητα θα περιμένει επ’ αόριστον και δε θα λάβει ποτέ τη ΚΜΕ.