Σχολή Ηλεκτρολόγων Μηχανικών

&

Μηχανικών Υπολογιστών

Εργαστήριο Λειτουργικών Συστημάτων

4η Εργαστηριακή Άσκηση

Καραμπάση Αικατερίνη, Α.Μ. : 03112517

Μανδηλαράς Νικηφόρος, Α.Μ: 03112012

Ομάδα : D12

Έβδομο Εξάμηνο

Παραδοτέα: 31/1 /2016

1.1 Υλοποίηση χρονοδρομολογητή κυκλικής επαναφοράς στο χώρο χρήστη

Για την υλοποίηση του χρονοδρομολογητή χρησιμοποιήσαμε δομή κυκλικής λίστας, που διευκόλυνε σημαντικά την εναλλαγές μεταξύ των διαδικασιών.

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 */



typedef struct pcb {
       int Id;
       pid_t PID;
       char name[TASK_NAME_SZ];
       struct pcb * next;
} ProcessControlBlock;

/* The Process queue is a list with the last element showing to the first
one */
/* One pointer to show to the current process and one to the previous one
*/

static ProcessControlBlock *CurrentProcess, *LastNode;

/* SIGALRM handler: Gets called whenever an alarm goes off.
 * The time quantum of the currently executing process has expired,
 * so send it a SIGSTOP. The SIGCHLD handler will take care of
 * activating the next in line.
 */

static void sigalrm_handler(int signum)
{
       kill(CurrentProcess->PID, SIGSTOP);
}

/* SIGCHLD handler: Gets called whenever a process is stopped,
 * terminated due to a signal, or exits gracefully.
 *
 * If the currently executing task has been stopped,
 * it means its time quantum has expired and a new one has
 * to be activated.
 */

static void SearchAndDequeue(pid_t pid)
{
       ProcessControlBlock *a=CurrentProcess, *b=CurrentProcess->next;
       /* Temporary pointers */

       while (b!=CurrentProcess)
       {
              if (b->PID==pid)
```

```
            {
                  printf("Process %d with PID %d is dead due to external
reason ", b->Id, pid);
                  a->next=b->next;
                  free(b);
                  printf("and is now removed from the process list.\n");
                  return;
            }
            a=a->next;
            b=b->next;
      }
}

static void sigchld_handler(int signum)
{
      int status;
      pid_t pid;
      for (;;)
      {
            if ((pid=waitpid(-1, &status, WNOHANG | WUNTRACED))<0) //
gyrizei to pid tis diadikasias sthn opoia anaferetai
            {
                  perror("waitpid");
                  exit(1);
            }
            if (pid==0)
                  break;
                  explain_wait_status(pid, status);
            if (WIFSTOPPED(status)) /* A child has received SIGSTOP */
            {//termatise logo kvantou
                  LastNode=CurrentProcess;
                  CurrentProcess=CurrentProcess->next;
                  alarm(SCHED_TQ_SEC);
                  kill(CurrentProcess->PID, SIGCONT);
            }
            if (WIFEXITED(status) || WIFSIGNALED(status)) /* A child is
dead */
            {//h termatise kanonika h me shma
                  if (CurrentProcess==LastNode) /* All children are now
dead */
                  {
                        free(CurrentProcess);
                        printf("No processes left to schedule.
Exiting... \n");
                        exit(1);
                  }
                  if (pid==CurrentProcess->PID)
                  {//termatise kanonika

                        /* The child that was running died before time
quantum expired */
                        /* Remove child fromn the list */
                        LastNode->next=CurrentProcess->next;
                        free(CurrentProcess);
                        /* Go to the next process in the list */
                        CurrentProcess=LastNode->next;
                        /* Set the alarm */
                        alarm(SCHED_TQ_SEC);
                        /* Send SIGCONT to the next process to continue
*/
                        kill(CurrentProcess->PID, SIGCONT);
                  }
                  else /* A child in the list has died from an external
signal */
                  {
                        /* Dequeue the child without reseting the alarm
*/
                        SearchAndDequeue(pid);
                  }
            }
      }
```

```
}


static void Enqueue(void)
{
	if (LastNode==NULL)
	{
		LastNode=CurrentProcess;
		LastNode->next=LastNode;
		return;
	}
	CurrentProcess->next=LastNode->next;
	LastNode->next=CurrentProcess;
	LastNode=CurrentProcess;
}

/* 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);
	}
}

void Child(char Executable[])
{
	raise(SIGSTOP);
	char *newargv[]={Executable, NULL, NULL, NULL};
	char *newenviron[]={NULL};
	execve(Executable, newargv, newenviron);
	perror("execve");
	exit(1);
}

int main(int argc, char *argv[])
{
	int nproc;
	/*
	 * For each of argv[1] to argv[argc - 1],
	 * create a new child process, add it to the process list.
	 */
```

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

        int i;
       pid_t p;
       for (i=1; i<=nproc; i++)
       {
              if ((p=fork())<0)
              {
                     perror("fork");
                     exit(1);
              }
              else if (p==0)
              {
                       Child(argv[i]);
                    /* char *newargv[]={argv[i], NULL, NULL, NULL};
                       char *newenviron[]={NULL};
                       execve(argv[i], newargv, newenviron);
                       perror("execve");
                      exit(1);
                           */
              }
                 // pateras ftiaxnoume thn domh kathe diergasias
              if ((CurrentProcess=(struct pcb *) malloc(sizeof(struct
pcb)))==NULL)
              {
                     printf("No Memory Available! Exiting...\n");
                     exit(1);
              }

              CurrentProcess->Id=i;
              CurrentProcess->PID=p;
              strncpy(CurrentProcess->name, argv[i], TASK_NAME_SZ);
              Enqueue(); // kai mpainei sth lista
       }



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

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

       /* First process on the list is LastNode->next */
       CurrentProcess=LastNode->next; //epeidh o teleytaios deixnei ston
prwto

       /* Set the alarm */
       alarm(SCHED_TQ_SEC);
       /* Start scheduling from the first Process on the list */
       kill(CurrentProcess->PID, SIGCONT);

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


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

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

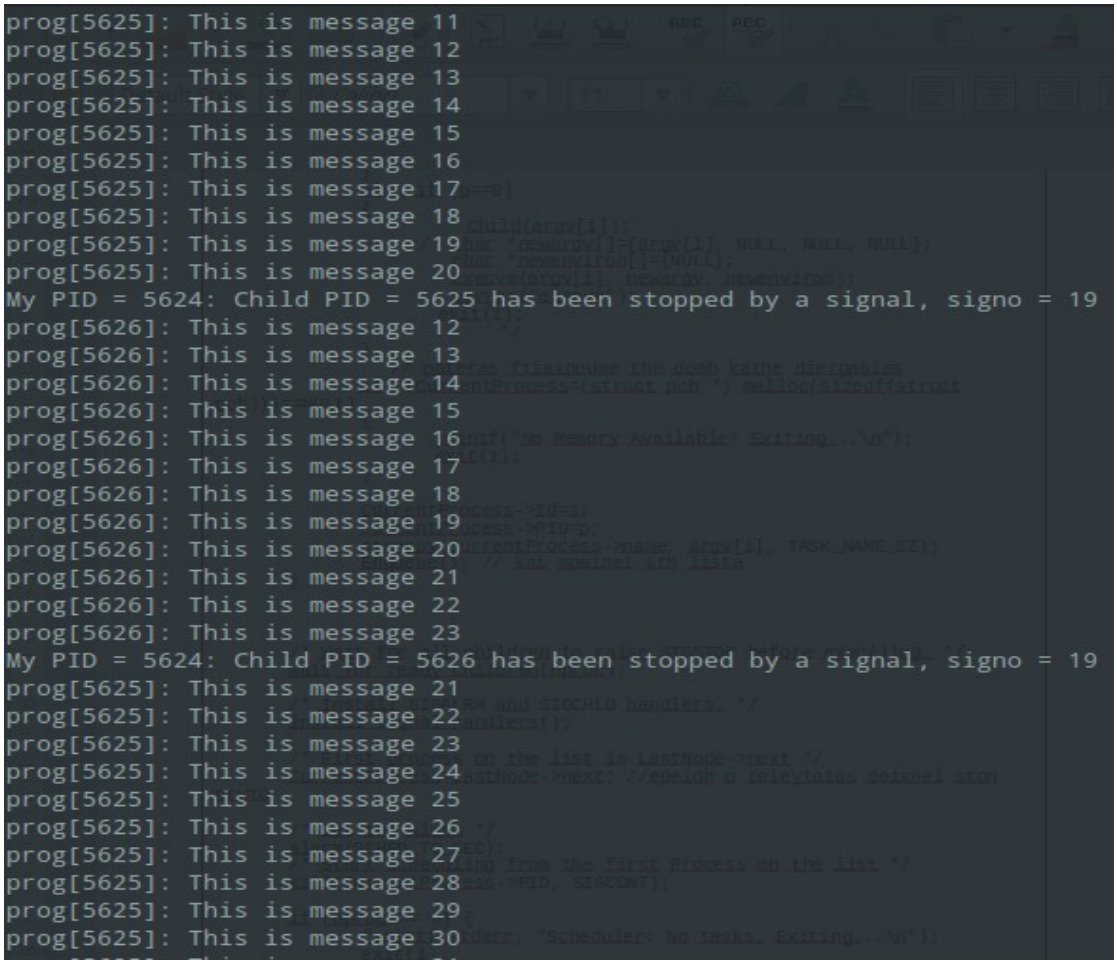
Ερωτήσεις:

1.Όταν έρθει ένα σήμα κατά την εκτέλεση της συνάρτησης χειρισμού του ίδιου σήματος, το σήμα αυτό μπλοκάρεται (παραδίδεται δηλαδή στη διεργασία όταν τελειώσει το χειρισμό του σήματος). Για να μπλοκάρουμε το SIGALRM κατά την εκτέλεση του SIGCHLD handler και το αντίστροφο, έχουμε ορίσει κατά την εγκατάσταση των handlers ότι τα δύο αυτά σήματα θα μπλοκάρονται κατά την εκτέλεση των handlers.

2.Κάθε φορά που ο χρονοδρομολογητής λαμβάνει SIGCHLD το πιθανότερο είναι να προέρχεται από τη διεργασία που τρέχει αυτή τη στιγμη (είτε εξέπνευση το κβάντο χρόνου της και της στείλαμε SIGSTOP και μας ήρθε πίσω SIGCHLD, είτε απλώς τελείωσε την εργασία της και τερματίστηκε κανονικά). Σε περίπτωση που από εξωτερικό παράγοντα πεθάνει κάποιο παιδί, έχουμε φροντίσει κατά τον χειρισμό τους σήματος SIGCHLD από τον χρονοδρομολογητή να αφαιρείται η διεργασία αυτή που πέθανε από την ουρά εκτέλεσης χωρίς να πειράζεται ο χρονιστής. Σε αντίθετη περίπτωση, ο χρονοδρομολογητής θα αφαιρούσε από τη λίστα την τρέχουσα διεργασία νομίζοντας πως αυτή τερματίστηκε επειδή τελείωσε την εργασία της) και όταν ερχόταν η σειρά της διεργασίας που πραγματικά πέθανε ο χρονοδρομολογητής θα έστελνε SIGCONT σε μια ανύπαρκτη διεργασία, δε θα λάμβανε ποτέ SIGCHLD και θα κολλούσε.

3. Κατ' αρχάς εάν δε χρησιμοποιούσαμε καθόλου το σήμα SIGCHLD δε θα μπορούσαμε να χειριστούμε το θάνατο μιας διεργασίας-παιδιού. Επίσης ο χρονοδρομολογητής εκτελείται στο χώρο χρήστη (όπως και οι διεργασίες-παιδιά) άρα κι αυτά χρονοδρομολογούνται κι από το λειτουργικό σύστημα στο οποίο εκτελούνται. Αν έστελνε ο χρονοδρομολογητής σήμα SIGSTOP σε μια διεργασία-παιδί δεν είμαστε σίγουρη ποτέ ότι θα τα παραλάβει για να αναστείλει τη λειτουργία της. Για παράδειγμα, θα μπορούσε ο χρονοδρομολογητής μετά την αποστολή SIGSTOP να στείλει αμέσως SIGCONT στην επόμενη στην ουρά και να αρχίσει να εκτελείται η επόμενη και η προηγούμενη να μην έχει λάβει ακόμα το SIGSTOP. Το αποτέλεσμα είναι ότι θα εκτελούνται 2 διεργασίες παιδιά παράλληλα.

Στη συνέχεια παραθέτουμε ένα στιγμιότυπο από την εκτέλεση του προγράμματος



1.2 Έλεγχος λειτουργίας χρονοδρομολογητή μέσω φλοιού

Προσθέτουμε στην υλοποίηση μας και την διαδικασία φλοιού προκειμένου να μπορούμε να εκτελέσουμε εντολές καθόσο ο δρομολογητής είναι ενεργός.

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 */

void child(char *executable)
{
       raise(SIGSTOP);
```

```
	char *newargv[]={executable, NULL, NULL, NULL};
	char *newenviron[]={NULL};
	execve(executable, newargv, newenviron);
	perror("execve");
	exit(1);
}


/* The Process queue is a list with the last element showing to the first
one */
/* One pointer to show to the current process and one to the previous one
*/
typedef struct pcb
{
	int Id;
	pid_t PID;
	char name[TASK_NAME_SZ];
	struct pcb * next;
} ProcessControlBlock;

static ProcessControlBlock *CurrentProcess, *LastNode;
int nproc;
int flag=0;

static void Enqueue(void){
	if (!LastNode)
	{
		LastNode=CurrentProcess;
		LastNode->next=LastNode;
		return;
	}
	CurrentProcess->next=LastNode->next;
	LastNode->next=CurrentProcess;
	LastNode=CurrentProcess;
}

static void SearchAndDequeue(pid_t pid)
{
	ProcessControlBlock *a=CurrentProcess, *b=CurrentProcess->next; /*
Temporary pointers */
	while (b!=CurrentProcess)
	{
		if (b->PID==pid)
		{
			printf("Process %d with PID %d is dead due to external
reason ", b->Id, pid);
			a->next=b->next;
			free(b);
			printf("and is now removed from the process list.\n");
			return;
		}
		a=a->next;
		b=b->next;
	}

}


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

	ProcessControlBlock *temp=CurrentProcess->next;

	printf("|Task name|\t|Id |\t|PID |\n");
	printf("-------------------------------\n");
	printf("|%9s|\t|%3d|\t|%5d| (Current Process)\n", CurrentProcess-
>name, CurrentProcess->Id, CurrentProcess->PID);
	while (temp!=CurrentProcess)
	{
		printf("|%9s|\t|%3d|\t|%5d|\n", temp->name, temp->Id, temp-
>PID);
```

```
                temp=temp->next;
        }
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int sched_kill_task_by_id(int id){
        ProcessControlBlock *temporary=CurrentProcess;
        while (temporary->Id!=id)
                temporary=temporary->next;
        kill(temporary->PID, SIGTERM);
        return(id);
}


/* Create a new task.  */
static void sched_create_task(char *executable){
        /* Create the process */
        pid_t p=fork();
        if (p<0)
        {
                perror("fork");
                exit(1);
        }
        flag=1;
        /* Child's code */
        if (p==0)
                child(executable);

        /* Allocate memory */
        ProcessControlBlock * New = (struct pcb *) malloc(sizeof(struct
pcb));
        if (New==NULL)
        {
                printf("No Memory Available! Exiting...\n");
                exit(1);
        }
        New->PID=p;
        New->Id=++nproc;
        strncpy(New->name, executable, TASK_NAME_SZ);
        /* Add to the list after the current process (it will be the next
process to run) */
        New->next=CurrentProcess->next;
        CurrentProcess->next=New;
}

/* 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(CurrentProcess->PID, SIGSTOP);
```

```
}

/*
 * SIGCHLD handler
 */
static void sigchld_handler(int signum){
	int status;
	pid_t pid;
	for (;;)
	{
		if ((pid=waitpid(-1, &status, WNOHANG | WUNTRACED))<0)
		{
			perror("waitpid");
			exit(1);
		}
		if (pid==0)
			break;
		printf("\n");
		explain_wait_status(pid, status);
		if (WIFSTOPPED(status)) /* A child has received SIGSTOP */
		{
			if (flag)
			{
				flag=0;
				return;
			}
			LastNode=CurrentProcess;
			CurrentProcess=CurrentProcess->next;
			alarm(SCHED_TQ_SEC);
			kill(CurrentProcess->PID, SIGCONT);
		}
		if (WIFEXITED(status) || WIFSIGNALED(status)) /* A child is
dead */
		{
			if (CurrentProcess==LastNode) /* All children are now
dead */
			{
				free(CurrentProcess);
				printf("No processes left to schedule.
Exiting... \n");
				exit(1);
			}
			if (pid==CurrentProcess->PID)
			{
				/* The child that was running died before time
quantum
				 * expired */
				/* Remove child fromn the list */
				LastNode->next=CurrentProcess->next;
				free(CurrentProcess);
				/* Go to the next process in the list */
				CurrentProcess=LastNode->next;
				/* Set the alarm */
				alarm(SCHED_TQ_SEC);
				/* Send SIGCONT to the next process to continue
*/
				kill(CurrentProcess->PID, SIGCONT);
			}
			else /* A child in the list has died from an external
signal */
			{
				/* Dequeue the child without reseting the alarm
*/
				SearchAndDequeue(pid);
			}
		}
	}
}

/* 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 BlockSignals;
	/* Signals which will be blocked during handler
	 * execution */
	struct sigaction SignalAction;
	sigemptyset(&BlockSignals);
	sigaddset(&BlockSignals, SIGCHLD); /* Add SIGCHLD and SIGALRM to the
set of signals */
	sigaddset(&BlockSignals, SIGALRM); /* which will be blocked */
	SignalAction.sa_flags=SA_RESTART;
	SignalAction.sa_mask=BlockSignals;
	SignalAction.sa_handler=sigchld_handler;
	if (sigaction(SIGCHLD, &SignalAction, NULL)<0)
	{
		perror("sigaction: SIGCHLD");
		exit(1);
	}
	SignalAction.sa_handler=sigalrm_handler;
	if (sigaction(SIGALRM, &SignalAction, NULL)<0)
	{
		perror("sigaction: SIGCHLD");
		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 pid_t 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]);
	}
	/* Parent */
	close(pfds_rq[1]);
	close(pfds_ret[0]);
	*request_fd = pfds_rq[0];
	*return_fd = pfds_ret[1];

	return (p);
}

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

        /* Create the shell. */
        p = sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd,
&return_fd);
        /* add the shell to the scheduler's tasks */
        CurrentProcess=(struct pcb *) malloc(sizeof(struct pcb));
        if (CurrentProcess==NULL)
        {
                printf("No Memory Available! Exiting...\n");
                exit(1);
        }
        CurrentProcess->Id=0;
        CurrentProcess->PID=p;
        strncpy(CurrentProcess->name, SHELL_EXECUTABLE_NAME, TASK_NAME_SZ);
        Enqueue();

        /*
         * 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-1; i++)
        {
                if ((p=fork())<0)
                {
                        perror("fork");
                        exit(1);
                }
                else if (p==0)
                {
                        child(argv[i]);
                }
                CurrentProcess=(struct pcb *) malloc(sizeof(struct pcb));
                if (CurrentProcess==NULL)
                {
                        printf("No Memory Available! Exiting...\n");
                        exit(1);
                }
                CurrentProcess->Id=i;
                CurrentProcess->PID=p;
                strncpy(CurrentProcess->name, argv[i], TASK_NAME_SZ);
                Enqueue();
        }
        nproc=i-1; /* number of proccesses 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();

        CurrentProcess=LastNode->next;
        alarm(SCHED_TQ_SEC);
        kill(CurrentProcess->PID, SIGCONT);

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

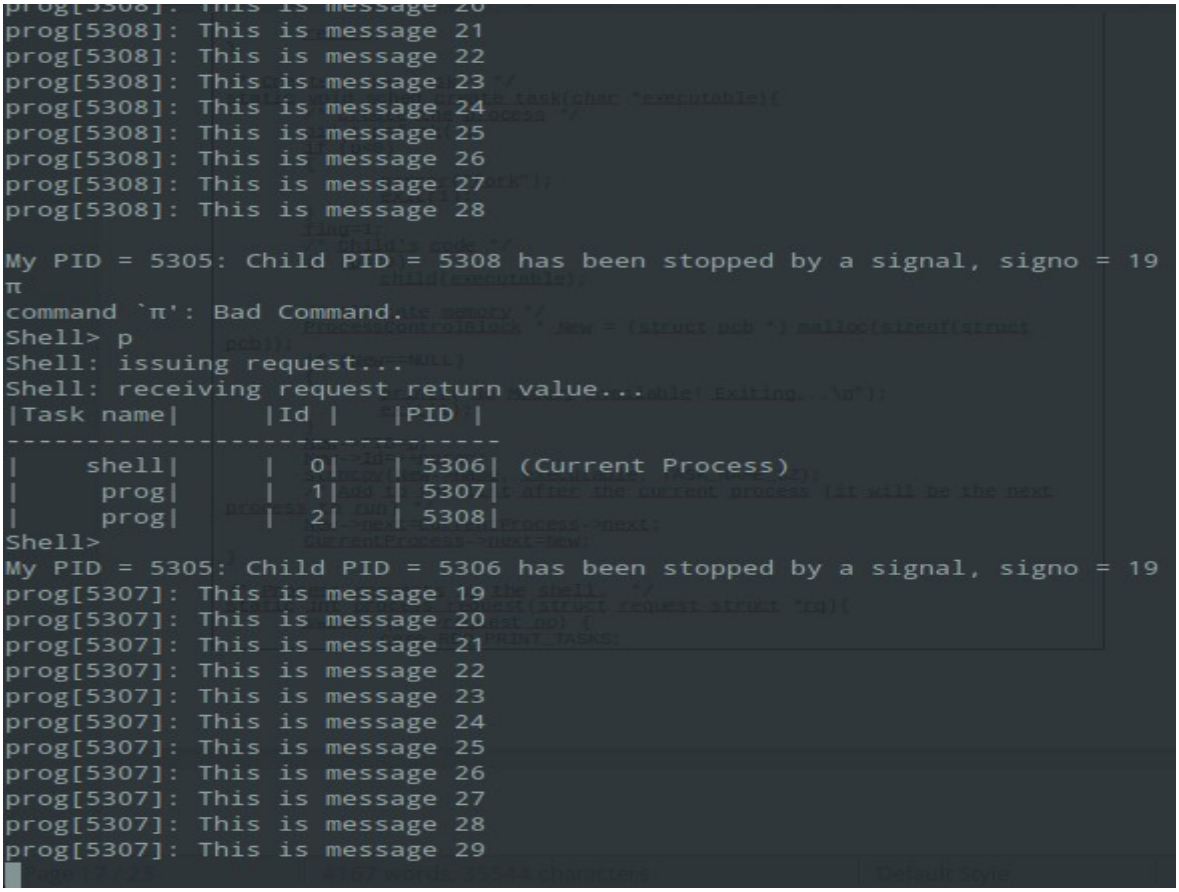
```
        shell_request_loop(request_fd, return_fd);

        /* 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;
}
```

Ερωτήσεις:

1. Για να υλοποιηθεί η εντολή "p" του χρήστη πρέπει να την παραλάβει ο φλοιός όσο εκτελείται. Δηλαδή αφού έχει δοθεί SIGCONT από τον χρονοδρομολογητή και τρέχει το κβάντο χρόνου του. Συνεπώς όταν ο scheduler απενεργοποιήσει τα σήματα και μπει στη διαδικασία υλοποίησης του request του φλοιού (του p εν προκειμένω) το πλέον πιθανό είναι να μην έχει προλάβει να εκπνεύσει το κβάντο χρόνου του φλοιού και άρα να εμφανίζεται αυτός ως η τρέχουσα διεργασία. Ωστόσο σε μια ακραία περίπτωση που ο φλοιός δεχτεί το "p" λίγο πριν εκπνεύσει το κβάντο χρόνου του μπορεί πριν ο scheduler προλάβει να κάνει disable τα σήματα, να υπάρξει SIGALRM, λόγω της εκπνοής του κβάντου χρόνου του φλοιού, να στείλει SIGSTOP ο scheduler στο φλοιό και στη συνέεια να στείλει SIGCONT στην επόμενη διεργασία που βρίσκεται στην ουρά. Σε αυτή την οριακή περίπτωση θα εμφανιστεί ως τρέχουσα διεργασία η επόμενη του shell.

2.Κατά τη διάρκεια υλοποίησης αιτήσεων του φλοιού, ο scheduler πολύ συχνά καλίεται να αλλάξει την ουρά των διεργασιών για να προσθέσει να αφαιρέσει είτε να αλλάξει τη θέση μιας διεργασίας. Κατά τη διάρκεια που γίνονται αυτές οι αλλαγές, πρέπει να μην εκτελούνται οι συναρτήσεις χειρισμού των σημάτων. Διαφορετικά μπορεί την ώρα που ο scheduler αλλάξει τη λίστα στη μνήμη να έρθει κάποιο σήμα, είτε SIGALRM είτε SIGCHLD και να δοθεί κβάντο χρόνου σε μια διεργασία που εκείνη την ώρα διαγράφεται ή γενικώς οι δείκτες της λίστα να μη δείχνουν τη σωστή διεργασία ή ακόμα και να δείχνουν σε NULL (ή σε μια πολύ κακή περίπτωση να έιναι ξεκρέμαστοι δείκτες).

Στη συνέχεια παραθέτουμε ένα στιγμιότυπο από την εκτέλεση του προγράμματος



1.3 Υλοποίηση προτεραιοτήτων στο χρονοδρομολογητή

Το πρόγραμμά μας εκτελεί την ίδια διεργασία με πριν με την διαφορά ότι σε αυτή την υλοποίηση έχουμε προσθέσει δύο επιπλέον εντολές για τις προτεραιότητες. Η υλοποίηση βασίζεται σε δύο κυκλικές ουρές, μία που περιλαμβάνει όλες τις διεργασίες και μία μόνο εκείνες με προτεραιότητα high.

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 */

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

```
	execve(executable, newargv, newenviron);
	perror("execve");
	exit(1);
}


/* The Process queue is a list with the last element showing to the first
one */
/* One pointer to show to the current process and one to the previous one
*/
typedef struct pcb
{
	int Id;
	pid_t PID;
	char name[TASK_NAME_SZ];
	char pr;
	struct pcb * next;
} ProcessControlBlock;

static ProcessControlBlock *CurrentProcess, *LastNode;
static ProcessControlBlock *CurrentProcess2, *LastNode2;
static ProcessControlBlock *c1, *c2;
int nproc;
int flag=0;
static int fl=0;

static void Enqueue(void){
	if (!LastNode)
	{
		LastNode=CurrentProcess;
		LastNode->next=LastNode;
		return;
	}
	CurrentProcess->next=LastNode->next;
	LastNode->next=CurrentProcess;
	LastNode=CurrentProcess;
}

static void Enqueue_2(void){
	if (!LastNode2){
		LastNode2=CurrentProcess2;
		LastNode2->next=LastNode2;
		return;
	}
	CurrentProcess2->next=LastNode2->next;
	LastNode2->next=CurrentProcess2;
	LastNode2=CurrentProcess2;
}

static void SearchAndDequeue(pid_t pid, ProcessControlBlock *a)
{
	ProcessControlBlock *b=a->next; /* Temporary pointers */
	while (b!=CurrentProcess)
	{
		if (b->PID==pid)
		{
			printf("Process %d with PID %d is dead due to external
reason ", b->Id, pid);
			a->next=b->next;
			free(b);
			printf("and is now removed from the process list.\n");
			return;
		}
		a=a->next;
		b=b->next;
	}

}

static void SearchAndDequeue_2(pid_t pid, int id){
	ProcessControlBlock *a=CurrentProcess, *b=CurrentProcess->next; /*
```

```
Temporary pointers */
	if((id==0) && (b!=CurrentProcess)){
		LastNode->next=CurrentProcess->next;
		free(CurrentProcess);
		/* Go to the next process in the list */
		CurrentProcess=LastNode->next;
		/* Set the alarm */
		alarm(SCHED_TQ_SEC);
		/* Send SIGCONT to the next process to continue */
		kill(b->PID, SIGCONT);
		printf("ok3\n");
	}
	else{
		while (b!=CurrentProcess){
			if (b->PID==pid){
				printf("Process %d with PID %d is dead due to
external reason ", b->Id, pid);
				a->next=b->next;
				free(b);
				printf("and is now removed from the process
list.\n");
				return;
			}
			a=a->next;
			b=b->next;
		}
		if(b==CurrentProcess){
			printf("Process %d with PID %d is dead due to external
reason ", b->Id, pid);
			a = NULL;
			b = NULL;
			free(a);
			free(b);
			printf("and is now removed from the process list.The
list is now empty.\n");
			CurrentProcess = c1;
			LastNode = c2;
		}
	}
	return;

}


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

	ProcessControlBlock *temp=CurrentProcess->next;

	printf("|Task name|\t|Id |\t|PID |\n");
	printf("------------------------------\n");
	printf("|%9s|\t|%3d|\t|%5d| (Current Process)\n", CurrentProcess-
>name, CurrentProcess->Id, CurrentProcess->PID);
	while (temp!=CurrentProcess)
	{
		printf("|%9s|\t|%3d|\t|%5d|\n", temp->name, temp->Id, temp-
>PID);
		temp=temp->next;
	}
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int sched_kill_task_by_id(int id){
	ProcessControlBlock *temporary=CurrentProcess;
	while (temporary->Id!=id)
		temporary=temporary->next;
	kill(temporary->PID, SIGTERM);
	return(id);
}
```

```
//we will select the high-prioritized tasks and execute only them
//the rest are going to be deleted
static void only_high(int task_arg){

	ProcessControlBlock *temp=CurrentProcess->next;
	ProcessControlBlock *cur=CurrentProcess;
	ProcessControlBlock *c3;
	//ProcessControlBlock *Cur2;

	if((cur->Id==0) && (CurrentProcess2!=NULL)){
		   c3=c1;
		//printf("ok1\n");
		while(c3->Id!=task_arg){
			c3 = c3->next;
		}

		CurrentProcess2=(struct pcb *) malloc(sizeof(struct pcb));
		if (CurrentProcess2==NULL){
			printf("No Memory Available! Exiting...\n");
			exit(1);
		}

		CurrentProcess2->Id=c3->Id;
		printf("%d\n", CurrentProcess2->Id);
		CurrentProcess2->PID=c3->PID;
		CurrentProcess2->pr = 'H';
		strncpy(CurrentProcess2->name, c3->name, TASK_NAME_SZ);

		Enqueue_2();

	}
	else{
		while(cur->Id!=task_arg){
			cur = temp;
			temp = cur->next;
		}

		CurrentProcess2=(struct pcb *) malloc(sizeof(struct pcb));
		if (CurrentProcess2==NULL){
			printf("No Memory Available! Exiting...\n");
			exit(1);
		}
		CurrentProcess2->Id=cur->Id;
		CurrentProcess2->PID=cur->PID;
		CurrentProcess2->pr = 'H';
		strncpy(CurrentProcess2->name, cur->name, TASK_NAME_SZ);

		Enqueue_2();
	}

	/*if(CurrentProcess->pr=='H' || CurrentProcess->pr=='h'){
		while(temp!=cur){
			if(temp->pr=='L' || temp->pr=='l'){
				CurrentProcess = temp;
				temp = temp->next;
				free(CurrentProcess);
				CurrentProcess = temp;
			}
		}
	}*/
	return;
}

static void prioritize_low(int task_arg){

	ProcessControlBlock *temp=CurrentProcess->next;
	ProcessControlBlock *cur=CurrentProcess;

	while(cur->Id!=task_arg){
		cur = temp;
```

```
                temp = cur->next;
        }
        cur->pr = 'L';
        printf("%c\n", cur->pr);
        //temp = cur->next;
        SearchAndDequeue_2(cur->PID, cur->Id);
        //printf("%d\n", temp->Id);


        return;
}

/* Create a new task.  */
static void sched_create_task(char *executable){
        /* Create the process */
        pid_t p=fork();
        if (p<0)
        {
                perror("fork");
                exit(1);
        }
        flag=1;
        /* Child's code */
        if (p==0)
                child(executable);

        /* Allocate memory */
        ProcessControlBlock * New = (struct pcb *) malloc(sizeof(struct
pcb));
        if (New==NULL)
        {
                printf("No Memory Available! Exiting...\n");
                exit(1);
        }
        New->PID=p;
        New->Id=++nproc;
        strncpy(New->name, executable, TASK_NAME_SZ);
        /* Add to the list after the current process (it will be the next
process to run) */
        New->next=CurrentProcess->next;
        CurrentProcess->next=New;
}

/* 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_LOW_TASK:
                        prioritize_low(rq->task_arg);
                        return 0;

                case REQ_HIGH_TASK:
                        only_high(rq->task_arg);
                        if(fl==0){
                                fl =1;
                                c1 = CurrentProcess;
                                CurrentProcess = CurrentProcess2;
                                c2 = LastNode;
                                LastNode = LastNode2;
                                printf("got here\n");
                                kill(c1->PID, SIGSTOP);
```

```
                        }
                        return 0;

                default:
                        return -ENOSYS;
        }
}

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

/*
 * SIGCHLD handler
 */
static void sigchld_handler(int signum){
        int status;
        pid_t pid;
        for (;;)
        {
                if ((pid=waitpid(-1, &status, WNOHANG | WUNTRACED))<0)
                {
                        perror("waitpid");
                        exit(1);
                }
                if (pid==0)
                        break;
                printf("\n");
                explain_wait_status(pid, status);
                if (WIFSTOPPED(status)) /* A child has received SIGSTOP */
                {
                        if (flag)
                        {
                                flag=0;
                                return;
                        }
                        LastNode=CurrentProcess;
                        printf("%c\n", CurrentProcess->pr);
                        CurrentProcess=CurrentProcess->next;
                        alarm(SCHED_TQ_SEC);
                        kill(CurrentProcess->PID, SIGCONT);
                }
                if (WIFEXITED(status) || WIFSIGNALED(status)) /* A child is
dead */
                {
                        if((CurrentProcess==LastNode) && (CurrentProcess-
>pr=='H')){
                                free(CurrentProcess);
                                CurrentProcess = c1;
                                LastNode = c2;
                        }
                        else if (CurrentProcess==LastNode) /* All children are
now dead */
                        {
                                free(CurrentProcess);
                                printf("No processes left to schedule.
Exiting... \n");
                                exit(1);
                        }

                        if (pid==CurrentProcess->PID)
                        {
                                /* The child that was running died before time
quantum
                                 * expired */
                                /* Remove child fromn the list */
                                LastNode->next=CurrentProcess->next;
                                free(CurrentProcess);
```

```
                        /* Go to the next process in the list */
                        CurrentProcess=LastNode->next;
                        if(CurrentProcess->pr=='H'){
                                SearchAndDequeue(pid, c1);
                                //printf("got here!!!!\n");
                        }
                        /* Set the alarm */
                        alarm(SCHED_TQ_SEC);
                        /* Send SIGCONT to the next process to continue
*/
                        kill(CurrentProcess->PID, SIGCONT);
                }
                else /* A child in the list has died from an external
signal */
                {
                        /* Dequeue the child without reseting the alarm
*/
                        printf("got here1\n");
                        SearchAndDequeue(pid, CurrentProcess);
                        printf("got here2\n");
                        if(CurrentProcess->pr=='H'){
                                SearchAndDequeue(pid, c1);
                                printf("got here!!!!\n");
                        }
                }
        }
    }
}

/* 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 BlockSignals;
    /* Signals which will be blocked during handler
     * execution */
    struct sigaction SignalAction;
    sigemptyset(&BlockSignals);
    sigaddset(&BlockSignals, SIGCHLD); /* Add SIGCHLD and SIGALRM to the
set of signals */
    sigaddset(&BlockSignals, SIGALRM); /* which will be blocked */
    SignalAction.sa_flags=SA_RESTART;
```

```
	SignalAction.sa_mask=BlockSignals;
	SignalAction.sa_handler=sigchld_handler;
	if (sigaction(SIGCHLD, &SignalAction, NULL)<0)
	{
		perror("sigaction: SIGCHLD");
		exit(1);
	}
	SignalAction.sa_handler=sigalrm_handler;
	if (sigaction(SIGALRM, &SignalAction, NULL)<0)
	{
		perror("sigaction: SIGCHLD");
		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 pid_t 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]);
```

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

	return (p);
}

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

	/* Create the shell. */
	p = sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd,
&return_fd);
	/* add the shell to the scheduler's tasks */
	CurrentProcess=(struct pcb *) malloc(sizeof(struct pcb));
	if (CurrentProcess==NULL)
	{
		printf("No Memory Available! Exiting...\n");
		exit(1);
	}
	CurrentProcess->Id=0;
	CurrentProcess->PID=p;
	CurrentProcess->pr = 'L';
	strncpy(CurrentProcess->name, SHELL_EXECUTABLE_NAME, TASK_NAME_SZ);
	Enqueue();

	/*
	 * 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-1; i++)
	{
		if ((p=fork())<0)
		{
			perror("fork");
			exit(1);
```

```
                }
                else if (p==0)
                {
                        child(argv[i]);
                }
                CurrentProcess=(struct pcb *) malloc(sizeof(struct pcb));
                if (CurrentProcess==NULL)
                {
                        printf("No Memory Available! Exiting...\n");
                        exit(1);
                }
                CurrentProcess->Id=i;
                CurrentProcess->PID=p;
                CurrentProcess->pr = 'L';
                strncpy(CurrentProcess->name, argv[i], TASK_NAME_SZ);
                Enqueue();
        }
        nproc=i-1; /* number of proccesses 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();

        CurrentProcess=LastNode->next;
        alarm(SCHED_TQ_SEC);
        kill(CurrentProcess->PID, SIGCONT);

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

        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;
}
```

Ερωτήσεις

1. Λιμοκτονία έχουμε όταν μία διαδικασία καταναλώνει ολοκληρωτικά την επερξεργαστική ισχύ χωρίς να μοιράζεται χρόνο εκτέλεσης με άλλες. Αυτό μπορεί να συμβεί εαν μέσω του φλοιού δώσουμε υψηλή προτεραιότητα σε κάποια διαδικασία. Επειδή τώρα ο φλοιός δεν έχει υψηλή προτεραιότητα δε μπορούμε να παρέμβουμε μα κάποιο τρόπο και η διαδικασία που κάναμε high θα εκτελείτε συνέχεια χωρίς διακοπή μέχρι να ολοκληρωθεί.

Στη συνέχεια παραθέτουμε ένα στιγμιότυπο από την εκτέλεση του προγράμματος όπου η διαδικασία του φλοιού και η πρώτη διαδικασία έχουν μπει σε υψηλή προτεραιότητα.