Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

4η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ

Ημερομηνία Παράδοσης : 10/02/2018

Ονοματεπώνυμο	**Αριθμός Μητρώου**
Αλεξάκης Κωνσταντίνος	03114086
Βασιλάκης Εμμανουήλ	03114167

Εξάμηνο	**Ακαδημαϊκό Έτος**
7ο	2017-2018

1 Υλοποίηση χρονοδρομολογητή κυκλικής επαναφοράς στο χώρο χρήστη

1.1 Πηγαίος κώδικας

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

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

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

#include <string.h>

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

struct Process_List
{
        int id;
        pid_t pid;
        char name[TASK_NAME_SZ];
        struct Process_List *next;
};

struct Process_List *head, *current;

void child (char *name)
{
        if (raise (SIGSTOP)) {
                perror("raise");
                exit (1);
        }
        /* Wait for the creation of the Process_List */

        char *newenviron[]      ={ NULL };
        char *newargv[]         ={ name, NULL/*, NULL, NULL*/ };
        execve (name, newargv, newenviron);

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

```
void activate_current ()
{
	printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, CONTINUES.\n", current->id, (long)
	                                                                    current->pid, current->name);

	/* Start counting tq */
	if (alarm (SCHED_TQ_SEC) < 0) {
		perror("alarm");
		exit(1);
	}
	if (kill (current->pid, SIGCONT)) {
		perror("kill");
		exit (1);
	}
}

/*
 * Removes from the list the now dead child.
 * Running shows if we change current.
 */
void child_died (pid_t p)
{
	int running = 0;
	if (current->pid == p) running = 1;

	struct Process_List *temp = head, *prev = NULL;

	while (temp != NULL) {
		if (temp->pid == p) {
			printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, HAS DIED.\n",
			                                        temp->id, (long) temp->pid, temp->name);

			if (prev != NULL) {		// child that died is not the head
				prev->next = temp->next;
				free (temp);
			} else {			// child that died is the head
				if (head->next == NULL) {  // last child died
					free (head);
					printf("Scheduler: No tasks remaining. Exiting...\n");
					exit (1);
				}
				temp = head->next;
				free (head);
				head = temp;
			}

			if (running) {
				if (prev == NULL || prev->next == NULL) current = head;
				else current = prev->next;
				activate_current();
			}

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

```
        fprintf(stderr, "Internal error: Child not found\n");
        exit(1);
}


/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        if (signum != SIGALRM) {
                fprintf(stderr, "Internal error: Called for signum %d, not SIGALRM\n", signum);
                exit(1);
        }

        /* Stop current running child */
        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, STOPS.\n", current->id, (long)
                                                                        current->pid, current->name);
        if (kill (current->pid, SIGSTOP)) {
                perror("kill");
                exit (1);
        }
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        if (signum != SIGCHLD) {
                fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n", signum);
                exit(1);
        }

        int status;
        pid_t p;

        /*
         * 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;
```

```
                if (WIFEXITED(status) || WIFSIGNALED(status)) {
                        /* A child has died, remove from list, if current died activate next in line */
                        child_died (p);
                }
                if (WIFSTOPPED(status) && (current->pid == p)) {
                        /* The active child has stopped due to SIGSTOP/SIGTSTP, etc... */
                        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, HAS STOPPED.\n",
                                                current->id, (long) current->pid, current->name);
                        if (current->next == NULL) current = head;
                        else current = current->next;
                        activate_current();
                }
        }
}

/* 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);
        }
}

int main(int argc, char *argv[])
{
        int nproc = argc - 1;           /* number of proccesses goes here */
```

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

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

int i;
for (i = 1; i <= nproc; i++) {
        if (i == 1) {
                head = (struct Process_List *) malloc (sizeof (struct Process_List));
                if (head == NULL) {
                        printf ("malloc");
                        exit (1);
                }
                current = head;
        }
        else {
                current->next = (struct Process_List *) malloc (sizeof (struct Process_List));
                if (current->next == NULL) {
                        printf ("malloc");
                        exit (1);
                }
                current = current->next;
        }

        current->next = NULL;
        current->id = i;
        strcpy (current->name, argv[i]);
        p = fork ();
        if (p == -1) {
                perror("fork");
                exit (1);
        }
        else if (p == 0) {
                /* child code */
                child (argv[i]);
                /* Should not reach this point */
                assert (0);
        }
        /* parent code */
        current->pid = p;
}

/* head points to the 1st node */

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

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

```
        printf("\n\n\n\nSTART SCHEDULING\n\n\n\n");
        /* Process_List ready with head -> start | tail -> end */

        current = head;

        /* Waking MSG */
        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, CONTINUES.\n", head->id, (long)
                                                                        head->pid, head->name);

        /* Start counting tq */
        if (alarm (SCHED_TQ_SEC) < 0) {
                perror("alarm");
                exit(1);
        }

        /* Start 1st process */
        if (kill (head->pid, SIGCONT)) {
                perror("kill");
                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;
}
```

1.2 Έξοδος εκτέλεσης του προγράμματος

Αν τώρα εκτελέσουμε την εντολή *./scheduler prog prog*, θα λάβουμε την παρακάτω έξοδο:

```
oslabc20@os-node1:~/EX4/1$ ./scheduler prog prog
My PID = 23688: Child PID = 23689 has been stopped by a signal, signo = 19
My PID = 23688: Child PID = 23690 has been stopped by a signal, signo = 19



START SCHEDULING



Scheduler: Process with ID = 1, Pid = 23689, Name = prog, CONTINUES.
prog: Starting, NMSG = 200, delay = 46
prog[23689]: This is message 0
prog[23689]: This is message 1
prog[23689]: This is message 2
prog[23689]: This is message 3
prog[23689]: This is message 4
prog[23689]: This is message 5
prog[23689]: This is message 6
prog[23689]: This is message 7
prog[23689]: This is message 8
prog[23689]: This is message 9
prog[23689]: This is message 10
prog[23689]: This is message 11
prog[23689]: This is message 12
prog[23689]: This is message 13
prog[23689]: This is message 14
Scheduler: Process with ID = 1, Pid = 23689, Name = prog, STOPS.
Scheduler: Process with ID = 1, Pid = 23689, Name = prog, HAS STOPPED.
Scheduler: Process with ID = 2, Pid = 23690, Name = prog, CONTINUES.
prog: Starting, NMSG = 200, delay = 94
prog[23690]: This is message 0
prog[23690]: This is message 1
prog[23690]: This is message 2
prog[23690]: This is message 3
prog[23690]: This is message 4
prog[23690]: This is message 5
prog[23690]: This is message 6
prog[23690]: This is message 7
Scheduler: Process with ID = 2, Pid = 23690, Name = prog, STOPS.
Scheduler: Process with ID = 2, Pid = 23690, Name = prog, HAS STOPPED.
Scheduler: Process with ID = 1, Pid = 23689, Name = prog, CONTINUES.
prog[23689]: This is message 15
prog[23689]: This is message 16
prog[23689]: This is message 17
prog[23689]: This is message 18
prog[23689]: This is message 19
prog[23689]: This is message 20
prog[23689]: This is message 21
```

1.3 Ερωτήσεις

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

   Αν εκτελείται μία από τις συναρτήσεις χειρισμού και έρθει σήμα SIGALRM ή SIGCHLD, το σήμα αυτό δεν παραδίδεται, αλλά μένει σε κατάσταση PENDING μέχρι να ολοκληρώσει την λειτουργία της η αντίστοιχη συνάρτηση χειρισμού, λόγω του ότι τα έχουμε εισάγει στην μάσκα του *sigaction*. Όμοια θα είναι και η συμπεριφορά ενός χρονοδρομολογητή χώρου πυρήνα σε μία τέτοια περίπτωση, ώστε να μην δημιουργηθούν ασυνέπειες.

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

   Περιμένουμε ότι το λαμβανόμενο σήμα SIGCHLD θα αναφέρεται στην διεργασία-παιδί που τρέχει αυτήν την στιγμή και υποθέτουμε ότι τελείωσε την λειτουργία της και τερματίστηκε, ή διακόπηκε λόγω της λήξης του χρόνου που της είχε αναθέσει ο χρονοδρομολογητής. Σε περίπτωση τερματισμού άλλου παιδιού, λόγω πχ κάποιου εξωτερικού σήματος SIGKILL που στάλθηκε, μέσα από την συνάρτηση *child_died*, αναζητούμε ποιο παιδί τερματίστηκε και το αφαιρούμε από τις προς εκτέλεση διεργασίες.

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

   Σε περίπτωση που δεν γίνει χειρισμός του σήματος SIGCHLD, παρά μόνο του σήματος SIGALRM, και προχωρήσουμε στην αποστολή

σήματος SIGCONT στο επόμενο παιδί της λίστας, υπάρχει περίπτωση το 1ο παιδί να μην έχει σταματήσει φυσιολογικά την λειτουργία του και έτσι να εκτελούνται παράλληλα και τα δύο παιδιά, οδηγώντας σε σφάλμα του χρονοδρομολογητή, ο οποίος υπάρχει για να αποτρέψει τέτοιες καταστάσεις. Την εξασφάλιση ότι το παιδί σταμάτησε να τρέχει, μας την παρέχει το σήμα SIGCHLD που στέλνει το παιδί στον πατέρα, επομένως για να υλοποιήσουμε έναν συνεπή χρονοδρομολογητή, είναι αναγκαίο να χειριζόμαστε το σήμα αυτό.

2 Έλεγχος λειτουργίας χρονοδρομολογητή μέσω φλοιού

2.1 Πηγαίος κώδικας

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

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

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

#include <string.h>

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

```
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */

struct Process_List
{
	int id;
	pid_t pid;
	char name[TASK_NAME_SZ];
	struct Process_List *next;
};

struct Process_List *head, *current;
int unique_id;


void child (char *name)
{
	if (raise (SIGSTOP)) {
		perror("raise");
		exit (1);
	}
	/* Wait for the creation of the Process_List */

	char *newenviron[]	={ NULL };
	char *newargv[]		={ name, NULL/*, NULL, NULL*/ };
	execve (name, newargv, newenviron);

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

/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
	printf ("Print a list of all tasks currently being scheduled:\n");
	struct Process_List *temp = head;
	while (temp != NULL) {
		printf("\tName = %s, ID = %d, PID = %ld\n", temp->name, temp->id, (long) 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)
{
	printf ("Kill task with ID = %d\n", id);
	struct Process_List *temp = head;
	while (temp != NULL) {
		if (temp->id == id) break;
		temp = temp->next;
	}
	if (temp == NULL) printf("Could not find process with ID = %d\n", id);
	else {
```

```
                if (kill (temp->pid, SIGKILL)) {
                        perror("kill_id");
                        exit (1);
                }
        }
        return -ENOSYS;
}


/* Create a new task.  */
/* Place on head. */
static void
sched_create_task(char *executable)
{
        struct Process_List *temp;
        temp = (struct Process_List *) malloc (sizeof (struct Process_List));
        if (temp == NULL) {
                printf ("malloc");
                exit (1);
        }

        temp->next = head;
        head = temp;

        temp->id = unique_id++;
        strcpy (temp->name, executable);

        pid_t p = fork ();
        if (p == -1) {
                perror("fork");
                exit (1);
        }
        else if (p == 0) {
                /* child code */
                child (executable);
                /* Should not reach this point */
                assert (0);
        }
        /* parent code */
        wait_for_ready_children(1);
        temp->pid = p;
}

/* 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;
    }
}

void activate_current ()
{
    printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, CONTINUES.\n", current->id, (long)
current->pid, current->name);

    /* Start counting tq */
    if (alarm (SCHED_TQ_SEC) < 0) {
        perror("alarm");
        exit(1);
    }
    if (kill (current->pid, SIGCONT)) {
        perror("kill");
        exit (1);
    }
}

/*
 * Removes from the list the now dead child.
 * Running shows if we change current.
 */
void child_died (pid_t p)
{
    int running = 0;
    if (current->pid == p) running = 1;

    struct Process_List *temp = head, *prev = NULL;

    while (temp != NULL) {
        if (temp->pid == p) {
            printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, HAS DIED.\n",
                                                temp->id, (long) temp->pid, temp->name);

            if (prev != NULL) {         // child that died is not the head
                prev->next = temp->next;
                free (temp);
            } else {                    // child that died is the head
                if (head->next == NULL) {  // last child died
                    free (head);
                    printf("Scheduler: No tasks remaining. Exiting...\n");
                    exit (1);
                }
                temp = head->next;
                free (head);
                head = temp;
            }

            if (running) {
                if (prev == NULL || prev->next == NULL) current = head;
                else current = prev->next;
                activate_current ();
```

```
                        }

                        return;
                }
                prev = temp;
                temp = temp->next;
        }

        fprintf(stderr, "Internal error: Child not found\n");
        exit(1);
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        if (signum != SIGALRM) {
                fprintf(stderr, "Internal error: Called for signum %d, not SIGALRM\n", signum);
                exit(1);
        }

        /* Stop current running child */
        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, STOPS.\n", current->id, (long)
                                                                        current->pid, current->name);
        if (kill (current->pid, SIGSTOP)) {
                perror("kill");
                exit (1);
        }
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        if (signum != SIGCHLD) {
                fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n", signum);
                exit(1);
        }

        int status;
        pid_t p;

        /*
         * 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;

                if (WIFEXITED(status) || WIFSIGNALED(status)) {
                        /* A child has died, remove from list, if current died activate next in line */
                        child_died (p);
                }
                if (WIFSTOPPED(status) && (current->pid == p)) {
                        /* The active child has stopped due to SIGSTOP/SIGTSTP, etc... */
                        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, HAS STOPPED.\n",
                                                current->id, (long) current->pid, current->name);
                        if (current->next == NULL) current = head;
                        else current = current->next;
                        activate_current();
                }
        }
}

/* 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]);
	/* ADDED */
	head->pid = p;

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

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

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

		signals_disable();
		ret = process_request(&rq);
		signals_enable();
```

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


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

        /* Create the shell. */
        head = (struct Process_List *) malloc (sizeof (struct Process_List));
        if (head == NULL) {
                printf ("malloc");
                exit (1);
        }
        head->id = 0;
        strcpy (head->name, SHELL_EXECUTABLE_NAME);
        head->next = NULL;
        sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);

        current = head;

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

        for (unique_id = 1; unique_id <= argc-1; unique_id++) {
                current->next = (struct Process_List *) malloc (sizeof (struct Process_List));
                if (current->next == NULL) {
                        printf ("malloc");
                        exit (1);
                }
                current = current->next;

                current->next = NULL;
                current->id = unique_id;
                strcpy (current->name, argv[unique_id]);
                p = fork ();
                if (p == -1) {
                        perror("fork");
                        exit (1);
                }
                else if (p == 0) {
                        /* child code */
                        child (argv[unique_id]);
                        /* Should not reach this point */
                        assert (0);
                }
                /* parent code */
                current->pid = p;
```

```
        }

        /* head points to the 1st node */

        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(argc);      //argc-1 arguments + shell

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

        printf("\n\n\n\nSTART SCHEDULING\n\n\n\n");
        /* Process_List ready with head -> start | tail -> end */

        current = head;
        /* Waking MSG */
        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, CONTINUES.\n", head->id, (long)
                                                                                    head->pid, head->name);

        /* Start counting tq */
        if (alarm (SCHED_TQ_SEC) < 0) {
                perror("alarm");
                exit(1);
        }

        /* Start 1st process */
        if (kill (head->pid, SIGCONT)) {
                perror("kill");
                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;
}
```

2.2 Έξοδος εκτέλεσης του προγράμματος

Αν τώρα εκτελέσουμε την εντολή *./scheduler-shell prog prog*, θα λάβουμε την παρακάτω έξοδο:

```
oslabc20@os-node1:~/EX4/2$ ./scheduler-shell prog prog
My PID = 23777: Child PID = 23778 has been stopped by a signal, signo = 19
My PID = 23777: Child PID = 23779 has been stopped by a signal, signo = 19
My PID = 23777: Child PID = 23780 has been stopped by a signal, signo = 19



START SCHEDULING



Scheduler: Process with ID = 0, Pid = 23778, Name = shell, CONTINUES.

This is the Shell. Welcome.

Shell> e prog
Shell: issuing request...
Shell: receiving request return value...
My PID = 23777: Child PID = 23781 has been stopped by a signal, signo = 19
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Print a list of all tasks currently being scheduled:
        Name = prog, ID = 3, PID = 23781
        Name = shell, ID = 0, PID = 23778
        Name = prog, ID = 1, PID = 23779
        Name = prog, ID = 2, PID = 23780
Shell> Scheduler: Process with ID = 0, Pid = 23778, Name = shell, STOPS.
Scheduler: Process with ID = 0, Pid = 23778, Name = shell, HAS STOPPED.
Scheduler: Process with ID = 1, Pid = 23779, Name = prog, CONTINUES.
prog: Starting, NMSG = 200, delay = 80
prog[23779]: This is message 0
prog[23779]: This is message 1
prog[23779]: This is message 2
prog[23779]: This is message 3
prog[23779]: This is message 4
prog[23779]: This is message 5
prog[23779]: This is message 6
prog[23779]: This is message 7
prog[23779]: This is message 8
Scheduler: Process with ID = 1, Pid = 23779, Name = prog, STOPS.
Scheduler: Process with ID = 1, Pid = 23779, Name = prog, HAS STOPPED.
Scheduler: Process with ID = 2, Pid = 23780, Name = prog, CONTINUES.
prog: Starting, NMSG = 200, delay = 62
prog[23780]: This is message 0
prog[23780]: This is message 1
prog[23780]: This is message 2
prog[23780]: This is message 3
prog[23780]: This is message 4
```

2.3 Ερωτήσεις

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

   Ως τρέχουσα διεργασία εμφανίζεται πάντοτε η διεργασία που αντιπροσωπεύει τον φλοιό. Αυτό δεν θα μπορούσε να μην συμβαίνει, αφού με βάση την εκφώνηση, ο φλοιός είναι και αυτός μία διεργασία-παιδί που δρομολογείται μαζί με τις υπόλοιπες διεργασίες-παιδιά. Άλλωστε, αν δεν ήταν στις τρέχουσες διεργασίες ο φλοιός, δεν θα μπορούσε να εκτελεστεί η εντολή **'p'**, αφού αυτή είναι ειδική εντολή της διεργασίας-παιδί φλοιός.

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

   Σε περίπτωση που δεν αποτρέπαμε την παράδοση σημάτων κατά την υλοποίηση των αιτήσεων του φλοιού, μπορεί να δημιουργηθούν ασυνέπειες σε κάποιο σημείο του προγράμματος, πχ κάποιος δείκτης της λίστας διεργασιών να κατέληγε να δείχνει σε NULL, αφού μεταβάλλουμε και περιεργαζόμαστε αυτήν την λίστα.

3 Υλοποίηση προτεραιοτήτων στο χρονοδρομολογητή

3.1 Πηγαίος κώδικας

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

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

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

```
#include <string.h>

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

struct Process_List
{
        int id;
        pid_t pid;
        char name[TASK_NAME_SZ];
        int priority;                       /* 0 -> LOW, 1 -> HIGH */
        struct Process_List *next;
};

struct Process_List *head, *current, *last_high;
int unique_id;

void child (char *name)
{
        if (raise (SIGSTOP)) {
                perror("raise");
                exit (1);
        }
        /* Wait for the creation of the Process_List */

        char *newenviron[]          ={ NULL };
        char *newargv[]             ={ name, NULL/*, NULL, NULL*/ };
        execve (name, newargv, newenviron);

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

void activate_current ()
{
        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, CONTINUES.\n", current->id, (long)
                                                                        current->pid, current->name);

        /* Start counting tq */
        if (alarm (SCHED_TQ_SEC) < 0) {
                perror("alarm");
                exit(1);
        }
        if (kill (current->pid, SIGCONT)) {
                perror("kill");
                exit (1);
        }
}

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

```
	printf ("Print a list of all tasks currently being scheduled, highest priority first:\n");

	struct Process_List *temp = head;
	printf("HIGH PRIORITY:\n");
	while (temp != NULL && temp->priority == 1) {
		printf("\tName = %s, ID = %d, PID = %ld\n", temp->name, temp->id, (long) temp->pid);
		temp = temp->next;
	}
	printf("LOW PRIORITY:\n");
	while (temp != NULL) {
		printf("\tName = %s, ID = %d, PID = %ld\n", temp->name, temp->id, (long) 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)
{
	printf ("Kill task with ID = %d\n", id);

	struct Process_List *temp = head;
	while (temp != NULL) {
		if (temp->id == id) break;
		temp = temp->next;
	}

	if (temp == NULL) printf("Could not find task with ID = %d\n", id);
	else {
		if (kill (temp->pid, SIGKILL)) {
			perror("kill_id");
			exit (1);
		}
	}
	return -ENOSYS;
}



/* Create a new task, with low priority. */
static void
sched_create_task(char *executable)
{

	struct Process_List *temp;

	temp = (struct Process_List *) malloc (sizeof (struct Process_List));
	if (temp == NULL) {
		printf ("malloc");
		exit (1);
	}


	temp->id = unique_id++;
```

```
        strcpy (temp->name, executable);
        temp->priority = 0;
        pid_t p = fork ();
        if (p == -1) {
                perror("fork");
                exit (1);
        }
        else if (p == 0) {
                /* child code */
                child (executable);
                /* Should not reach this point */
                assert (0);
        }
        /* parent code */
        wait_for_ready_children(1);
        temp->pid = p;

        if (last_high == NULL) {
                temp->next = head;
                head = temp;
        } else {
                temp->next = last_high->next;
                last_high->next = temp;
        }
}

/* Go to the start */
static void
sched_set_high_by_id(int id)
{
        struct Process_List *temp, *prev;

        /* Search only low priority tasks */

        if (last_high == NULL) {
                prev = NULL;
                temp = head;
        } else {
                prev = last_high;
                temp = last_high->next;
        }

        while (temp != NULL) {
                if (temp->id == id) {
                        temp->priority = 1;

                        if (prev != NULL) {
                                prev->next = temp->next;
                                temp->next = head;
                                head = temp;
                        }
                        // Se periptosh pou prev == NULL, den allazo thesi tou temp, einai hdh sthn arxh

                        if (last_high == NULL) last_high = head;

                        if (current->priority == 0) {
                                if (kill (current->pid, SIGSTOP)) {
```

```
                                        perror ("kill");
                                        exit (1);
                                }
                        }
                        return;
                }
                prev = temp;
                temp = temp->next;
        }
        // did not find id
        printf("Could not change priority on ID = %d\n", id);
        return;
}

static void
sched_set_low_by_id(int id)
{

        /* Search only high priority tasks */

        struct Process_List *temp = head, *prev = NULL;

        while (temp != NULL && temp->priority == 1) {
                if (temp->id == id) {
                        temp->priority = 0;

                        //if (last_high == temp) last_high = prev;

                        // An temp == last_high, to last_high ginetai NULL an eimai sthn arxh, afou prev ==
                        // NULL kai den metakino to temp
                        if (temp->next == NULL || temp->next->priority == 0) last_high = prev;
                        else {     // diaforetika metakino to temp
                                temp->next = last_high->next;
                                last_high->next = temp;
                        }

                        if (head->priority == 1 && current->priority == 0) {
                                if (kill (current->pid, SIGSTOP)) {
                                        perror ("kill");
                                        exit (1);
                                }
                        }

                        return;
                }
                prev = temp;
                temp = temp->next;
        }
        // did not find id
        printf("Could not change priority on ID = %d\n", id);
        return;
}

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

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

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

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

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

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

		default:
			return -ENOSYS;
	}
}


/*
 * Removes from the list the
 * now dead child.
 */
void child_died (pid_t p)
{
	int running = 0;
	if (current->pid == p) running = 1;

	struct Process_List *temp = head, *prev = NULL;

	while (temp != NULL) {
		if (temp->pid == p) {
			printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, HAS DIED.\n",
							temp->id, (long) temp->pid, temp->name);

			// change last_high
			// if (temp == last_high) last_high = prev;
			if (temp->priority == 1 && (temp->next == NULL || temp->next->priority == 0))
				last_high = prev;

			if (prev != NULL) {		// child that died is not the head
				prev->next = temp->next;
				free (temp);
			} else {			// child that died is the head
				if (head->next == NULL) {  // last child died
					free (head);
					printf("Scheduler: No tasks remaining. Exiting...\n");
					exit (1);
				}
				temp = head->next;
```

```
                                free (head);
                                head = temp;
                        }

                        if (running) {
                                if (prev == NULL || prev->next == NULL) current = head;
                                else if (prev->next->priority == 0 && head->priority == 1) current = head;
                                else current = prev->next;
                                activate_current ();
                        }

                        return;
                }
                prev = temp;
                temp = temp->next;
        }

        fprintf(stderr, "Internal error: Child not found\n");
        exit(1);
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        if (signum != SIGALRM) {
                fprintf(stderr, "Internal error: Called for signum %d, not SIGALRM\n", signum);
                exit(1);
        }

        /* Stop current running child */
        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, STOPS.\n", current->id, (long)
                                                                        current->pid, current->name);
        if (kill (current->pid, SIGSTOP)) {
                perror("kill");
                exit (1);
        }
}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        if (signum != SIGCHLD) {
                fprintf(stderr, "Internal error: Called for signum %d, not SIGCHLD\n", signum);
                exit(1);
        }

        int status;
        pid_t p;

        /*
         * 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;

                if (WIFEXITED(status) || WIFSIGNALED(status)) {
                        /* A child has died, remove from list, if current died activate next in line */
                        child_died (p);
                }
                if (WIFSTOPPED(status) && (current->pid == p)) {
                        /* The active child has stopped due to SIGSTOP/SIGTSTP, etc... */
                        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, HAS STOPPED.\n",
                                                current->id, (long) current->pid, current->name);

                        if (current->next == NULL) current = head;
                        else if (current->next->priority == 0 && head->priority == 1) current = head;
                        else current = current->next;
                        activate_current();
                }
        }
}


/* 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]);
	/* ADDED */
	head->pid = p;

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

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

	/*
	 * Keep receiving requests from the shell.
```

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

		signals_disable();
		ret = process_request(&rq);
		signals_enable();

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


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

	/* Create the shell. */
	head = (struct Process_List *) malloc (sizeof (struct Process_List));
	if (head == NULL) {
		printf ("malloc");
		exit (1);
	}
	head->id = 0;
	strcpy (head->name, SHELL_EXECUTABLE_NAME);
	head->priority = 0;
	head->next = NULL;
	sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);

	current = head;

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

	for (unique_id = 1; unique_id <= argc-1; unique_id++) {
		current->next = (struct Process_List *) malloc (sizeof (struct Process_List));
		if (current->next == NULL) {
			printf ("malloc");
			exit (1);
		}
		current = current->next;

		current->next = NULL;
		current->id = unique_id;
		strcpy (current->name, argv[unique_id]);
```

```
                current->priority = 0;
                p = fork ();
                if (p == -1) {
                        perror("fork");
                        exit (1);
                }
                else if (p == 0) {
                        /* child code */
                        child (argv[unique_id]);
                        /* Should not reach this point */
                        assert (0);
                }
                /* parent code */
                current->pid = p;
        }

        /* head points to the 1st node */

        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(argc);      //argc-1 arguments + shell

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

        printf("\n\n\n\nSTART SCHEDULING\n\n\n\n");
        /* Process_List ready with head -> start | tail -> end */

        current = head;
        last_high = NULL;
        /* Waking MSG */
        printf("Scheduler: Process with ID = %d, Pid = %ld, Name = %s, CONTINUES.\n", head->id, (long)
                                                                        head->pid, head->name);

        /* Start counting tq */
        if (alarm (SCHED_TQ_SEC) < 0) {
                perror("alarm");
                exit(1);
        }

        /* Start 1st process */
        if (kill (head->pid, SIGCONT)) {
                perror("kill");
                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;
}
```

3.2 Έξοδος εκτέλεσης του προγράμματος

Αν τώρα εκτελέσουμε την εντολή *./scheduler-shell prog*, θα λάβουμε την παρακάτω έξοδο:

```
oslabc20@os-node1:~/EX4/3$ ./scheduler-shell prog
My PID = 23885: Child PID = 23886 has been stopped by a signal, signo = 19
My PID = 23885: Child PID = 23887 has been stopped by a signal, signo = 19



START SCHEDULING



Scheduler: Process with ID = 0, Pid = 23886, Name = shell, CONTINUES.

This is the Shell. Welcome.

Shell> h 0
Shell: issuing request...
Shell: receiving request return value...
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Print a list of all tasks currently being scheduled, highest priority first:
HIGH PRIORITY:
        Name = shell, ID = 0, PID = 23886
LOW PRIORITY:
        Name = prog, ID = 1, PID = 23887
Shell> Scheduler: Process with ID = 0, Pid = 23886, Name = shell, STOPS.
Scheduler: Process with ID = 0, Pid = 23886, Name = shell, HAS STOPPED.
Scheduler: Process with ID = 0, Pid = 23886, Name = shell, CONTINUES.
Scheduler: Process with ID = 0, Pid = 23886, Name = shell, STOPS.
Scheduler: Process with ID = 0, Pid = 23886, Name = shell, HAS STOPPED.
Scheduler: Process with ID = 0, Pid = 23886, Name = shell, CONTINUES.
l 0
Shell: issuing request...
Shell: receiving request return value...
Shell> Scheduler: Process with ID = 0, Pid = 23886, Name = shell, STOPS.
Scheduler: Process with ID = 0, Pid = 23886, Name = shell, HAS STOPPED.
Scheduler: Process with ID = 1, Pid = 23887, Name = prog, CONTINUES.
prog: Starting, NMSG = 200, delay = 107
prog[23887]: This is message 0
prog[23887]: This is message 1
prog[23887]: This is message 2
prog[23887]: This is message 3
prog[23887]: This is message 4
prog[23887]: This is message 5
```

3.3 Ερωτήσεις

1. *Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας.*

   Σε περίπτωση που μία διεργασία έχει προτεραιότητα HIGH, αλλά για κάποιο λόγο, δεν καταφέρνει να ολοκληρώσει την λειτουργία της, ή απλά αργεί πάρα πολύ, οι διεργασίες με χαμηλή προτεραιότητα θα πρέπει να περιμένουν την ολοκλήρωσή της, ανεξάρτητα από το εάν αυτές έχουν πολύ μικρότερο χρόνο διεκπεραίωσης.