Λειτουργικά Συστήματα – 4η Εργαστηριακή Άσκηση

Ομάδα εργαστηρίου: **oslabb30**

Φοιτητές: **Γεώργιος Λαγός 03117034, Παναγιώτης Ζευγολατάκος 03117804**

Στην αναφορά χρησιμοποιείται παντού Α’ πληθυντικός γιατί τόσο η αναφορά όσο και η ίδια η εργασία έγιναν με πλήρη επικοινωνία μεταξύ των φοιτητών.

1.1 Υλοποίηση χρονοδρομολογητή κυκλικής επαναφοράς στο χώρο χρήστη

Για την αντιμετώπιση της ζητούμενης άσκησης καταλήξαμε να χρησιμοποιήσουμε λίστες δυναμικά, όπου η κάθε φορά τρέχουσα διεργασία θα είναι το κεφάλι της λίστας. Συνεπώς, μόλις μία διεργασία, είτε τελειώσει είτε λήξει το κβάντο χρόνου της θα αφαιρείται από το κεφάλι, το οποίο θα αντικαθίσταται με την αμέσως επόμενη διεργασία.

Παραθέτουμε τον κώδικα που χρησιμοποιήσαμε:

Συγκεκριμένα κατασκευάσαμε το αρχείο list.c στο οποίο ορίσαμε όλες τις συναρτήσεις που θα χρειαστούμε για την δυναμική χρησιμοποιήση λιστών:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#include "list.h"

//creates a new process
struct process* Create_process (pid_t pid) {

        struct process *temp;
        temp = malloc(sizeof(*temp));
        if (temp == NULL) {
                perror("Error creating proccess");
                exit(1);
        }

        temp->pid = pid;
        temp->next = NULL;

        return temp;

}

struct list* Create_list () {

        struct list* temp;
        temp = malloc(sizeof(*temp));
        if (temp == NULL) {
                perror("Error creating list");
                exit(1);
        }

        temp->root = NULL;
        temp->counter = 0;

        return temp;

}

//add a process at the end of the list
void Add_process (struct list *List,struct process *proc) {

        struct process *temp = List->root;
        struct process *temp2 = temp;
        if(temp == NULL){
                List->root = proc;
                List->counter = List->counter + 1;
        }
        else{
                while (temp != NULL) {
                        //printf("trww kark atermono\n");
                        temp2 = temp; //temp2 will find the last process
                        temp = temp->next; //temp will always turn out NULL
```

```
                }

                proc->next = NULL;
                temp2->next = proc;
                List->counter = List->counter + 1;
        }
}

//remove a process from the beginning of the list
struct process* Cycle_process (struct list *List) {

        struct process *temp = List->root;
        if (temp == NULL) return NULL;
        if (List->counter == 1){
                List->root = NULL;
                List->counter--;
                return temp;
        }
        struct process *temp2 = temp->next;
        List->root = temp2;
        List->counter = List->counter - 1;
        return temp;
}

void Destroy_process (struct list *List,pid_t pid) {

        struct process *temp = List->root;
        if (temp == NULL) return;

        struct process *temp2 = temp;
        while (temp != NULL && temp->pid != pid) {
                temp2 = temp;
                temp = temp->next;
        }
        if (temp != NULL) {

        temp2->next=temp->next;
        free(temp);
        }
```

Στη συνέχεια παραθέτουμε το κύριο πρόγραμμα μας (scheduler.c):

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


/*
 * SIGALRM handler
 */
struct list* list;  // global list pointer

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

        fprintf(stderr,"alarm: about to sigstop: %ld\n", (long)list->root->pid);

        kill((list->root)->pid, SIGSTOP); //SIG-ALARM HANDLER ONLY SIGSTOPS BECAUSE SIGSTOPPI
}

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

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

```
        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 CHILD DIED NORMALLY OR SIGKILLED

                        if (WIFSIGNALED(status)) {
                                Destroy_process (list, p); //SIGKILLED
                        }
                        else { //DIED NORMALLY
                                fprintf(stderr,"%ld child has died\n",(long)list->root->pid);
                                fprintf(stderr,"Parent: Received SIGCHLD, child is dead. Exiting.\n");
                                struct process* temp;
                                temp = Cycle_process(list);
                                free(temp);
                                if(!(list->counter)){
                                        exit(3);
                                }
                                if (alarm(SCHED_TQ_SEC) < 0) {
                                        perror("ALARM\n");
                                        exit(1);
                                }
                                kill((list->root)->pid, SIGCONT);
                        }
                }
                else if (WIFSTOPPED(status)) { //SIGSTOPPED
                        fprintf(stderr,"%ld child has stopped\n",(long)list->root->pid);
                        struct process* temp = Cycle_process(list);
                        Add_process(list,temp);
                        if (alarm(SCHED_TQ_SEC) < 0) {
                                perror("alarm");
                                exit(1);
                        }
                        fprintf(stderr,"stopped:about to sigcont pid: %ld\n",(long)list->root->pid);
                        kill((list->root)->pid, SIGCONT);
                        printf("Parent: Child has been stopped. Moving right along...\n");
                }

        }
}
```

```c
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("Sigchld");
                exit(1);
        }

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

int main(int argc, char *argv[])
{
        int nproc;
        pid_t pid;
        struct process* proc;
        list = Create_list();
        /*
         * For each of argv[1] to argv[argc - 1],
         * create a new child process, add it to the process list.
         */
        nproc = argc-1; /* number of proccesses goes here */
        if (nproc == 0) {
                fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                exit(1);
        }
        int i;
        for (i = 0; i < nproc; i++){ //CREATE ALL THE PROCESSES
                pid = fork();
                if(pid > 0){
                        fprintf(stderr,"about to create_process with pid %ld\n",(long)pid);
                        proc = Create_process(pid);
                        fprintf(stderr,"proc created with pid %ld \n",(long)proc->pid);
                        Add_process(list,proc);
                }
                else if (pid == 0) {
                        raise(SIGSTOP); //PAUSE THEM
                        char *newargv[] = {argv[i+1], NULL, NULL, NULL };
                        char *newenviron[] = { NULL };
                        execve(argv[i+1],newargv,newenviron); //EXECUTE WHAT THEY MUST
                        exit(1);
                }
        }

        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(nproc);
        /* Install SIGALRM and SIGCHLD handlers. */
        install_signal_handlers();
        if (alarm(SCHED_TQ_SEC) < 0) {
                perror("alarm");
                exit(1);
        }
        kill((list->root)->pid, SIGCONT);
        /* loop forever  until we exit from inside a signal handler. */
        while (pause());
        /* Unreachable */
        fprintf(stderr, "Internal error: Reached unreachable point\n");
        return 1;
```

Ενδεικτικά η εκτέλεση του προγράμματος μας δίνει το ακόλουθο αποτέλεσμα:

```
prog: Starting, NMSG = 200, delay = 38
prog[18662]: This is message 0
prog[18662]: This is message 1
prog[18662]: This is message 2
prog[18662]: This is message 3
prog[18662]: This is message 4
prog[18662]: This is message 5
prog[18662]: This is message 6
prog[18662]: This is message 7
prog[18662]: This is message 8
prog[18662]: This is message 9
prog[18662]: This is message 10
prog[18662]: This is message 11
prog[18662]: This is message 12
prog[18662]: This is message 13
prog[18662]: This is message 14
prog[18662]: This is message 15
prog[18662]: This is message 16
prog[18662]: This is message 17
alarm: about to sigstop: 18662
My PID = 18661: Child PID = 18662 has been stopped by a signal, signo = 19
18662 child has stopped
stopped:about to sigcont pid: 18663
Parent: Child has been stopped. Moving right along...
prog: Starting, NMSG = 200, delay = 149
prog[18663]: This is message 0
prog[18663]: This is message 1
prog[18663]: This is message 2
prog[18663]: This is message 3
prog[18663]: This is message 4
alarm: about to sigstop: 18663
My PID = 18661: Child PID = 18663 has been stopped by a signal, signo = 19
18663 child has stopped
stopped:about to sigcont pid: 18664
Parent: Child has been stopped. Moving right along...
prog: Starting, NMSG = 200, delay = 131
prog[18664]: This is message 0
prog[18664]: This is message 1
prog[18664]: This is message 2
prog[18664]: This is message 3
prog[18664]: This is message 4
prog[18664]: This is message 5
alarm: about to sigstop: 18664
My PID = 18661: Child PID = 18664 has been stopped by a signal, signo = 19
18664 child has stopped
stopped:about to sigcont pid: 18665
Parent: Child has been stopped. Moving right along...
prog: Starting, NMSG = 200, delay = 113
prog[18665]: This is message 0
prog[18665]: This is message 1
prog[18665]: This is message 2
prog[18665]: This is message 3
prog[18665]: This is message 4
prog[18665]: This is message 5
^C
oslabb30@os-node1:~/giorgos/ask4$
```

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*

   Στη δοσμένη συνάρτηση install_signal_handlers() έχουμε ορίσει μέσω μάσκας να μπλοκάρεται το σήμα SIGALRM όταν εκτελείται το τμήμα κώδικα του SIGCHLD handler. Αντίστοιχα, έχει οριστεί και στην αντίθετη περίπτωση, όταν δηλαδή εκτελείται ο SIGALRM handler και ληφθεί σήμα SIGCHLD. Επομένως, αν έχουμε λάβει οποιοδήποτε από τα προαναφερθέντα σήματα ελέγχου, η άφιξη του άλλου απλά αγνοείται.

   Όσον αφορά έναν πραγματικό χρονοδρομολογητή, αυτός λειτουργεί με hardware interrupts (διακοπές) και όχι με σήματα. Έτσι, υπάρχει καλύτερη και αμεσότερη απόκριση, αφού με το που γίνει μια διακοπή άμεσα θα εκτελεστεί η ρουτίνα εξυπηρέτησής της. Στη περίπτωση μας, τα σήματα ενδέχεται να έχουν καθυστερήσεις αφού ακόμα και αυτά χρονοδρομολογούνται.

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;*

   Το σήμα SIGCHLD το λαμβάνει ο χρονοδρομολογητής μας, όταν αλλάξει κάποιο παιδί την κατάστασή του. Αυτό κανονικά συμβαίνει όταν ένα παιδί δεχθεί σήμα SIGSTOP ή τερματιστεί κανονικά. Ωστόσο, σήμα KILL μπορεί να λάβει οποιαδήποτε διεργασία χωρίς να το έχουμε στείλει εμείς το σήμα. Σε τέτοια περίπτωση μεριμνήσαμε να διαγράφεται από την λίστα μας η διεργασία αυτή ώστε να παραμένει ορθή η αναπαράσταση με λίστα. Έτσι, όποια διεργασία και να πάθει το οτιδήποτε, ο χρονοδρομολογητής μας θα λειτουργεί σωστά. Παράθεση του κώδικα που αντιμετωπίζει το KILL:

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.*

   Ο λόγος που χρησιμοποιούμε δύο διαφορετικά σήματα είναι ότι μπορεί να υπάρχουν καθυστερήσεις μεταξύ της αποστολής και λήψης των σημάτων. Πιο συγκεκριμένα, αν χρησιμοποιούσαμε μόνο handler για το SIGALRM θα ήταν πιθανό ένα SIGSTOP να σταλεί σε μια διεργασία και αμέσως μετά ένα SIGCONT, ωστόσο είναι πιθανό η δεύτερη διεργασία να λάβει πρώτη το SIGCONT πριν καν σταματήσει η πρώτη, γεγονός που θα έκανε το χρονοδρομολογητή μας ελαττωματικό. Όμως τώρα με τους δύο handlers, όταν έρθει σήμα SIGARLM στέλνουμε SIGSTOP στη διεργασία που εκτελείται και αναμένουμε να μας έρθει σήμα SIGCHLD από τη διεργασία και αφου μας έρθει ελέγχουμε τι της συνέβη και μετά από αυτή τη διαδικασία στέλνουμε SIGCONT στην επόμενη που έχει σειρά να ενεργοποιηθεί. Έτσι, έχουμε εξασφαλίσει την ορθή λειτουργία του χρονοδρομολογητή μας.

1.2 Έλεγχος λειτουργίας χρονοδρομολογητή μέσω φλοιού

Ακολουθήσαμε πανομοιότυπη υλοποίηση με το προηγούμενο ερώτημα (προσθέσαμε μερικές ακόμα συναρτήσεις στην list.c και τροποποιήσαμε το τι επιστρέφουν μερικές).

Παραθέτουμε μόνο το scheduler-shell.c μιας και οι αλλαγές στο list2.c δεν είναι σημαντικές:

```c
#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 "list2.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 list* list;
int current_id;
char flag; // if 0 remove head
/* Print a list of all tasks currently being scheduled.  */
static void sched_print_tasks(void)
{
    struct process* temp;
    temp = list->root;
    if(temp == NULL){
        return;
    }
    printf("\n");
    printf("Currently running");
    do{
    printf("[%s]:with pid %ld and serial_no %d\n\n",temp->name,(long)temp->pid,temp->id);
    }while((temp = temp->next) != NULL);

}
static int sched_kill_task_by_id(int id)
{
    if (list->root->id == id){
        kill(list->root->pid,SIGKILL);
    }
    else{
        pid_t pid = Destroy_process(list,id);
        if (pid<0) {
                perror("CAN'T DELETE SOMETHING THAT DOESN'T EXIST\n");
                return 0;
        }
        list->counter = list->counter - 1;
        // signals_disable();
        flag = 1; //DONT remove head now in SIGCHILD
        kill(pid,SIGKILL);
        // signals_enable();
    }
    return 0;
}
```

```c
static void sched_create_task(char *executable)
{
    pid_t pid;
    pid = fork();
    if(pid < 0){
        perror("FORK ERROR");
        exit(1);
    }
    else if(pid == 0){
        char *newargv[] = {executable, NULL, NULL, NULL };
        char *newenviron[] = { NULL };
        raise(SIGSTOP);
        execve(executable,newargv,newenviron);
        fprintf(stderr,"Shouldn't be here");
        exit(1);
    }
    else {
        struct process* temp = Create_process(pid,current_id++);
        temp->name = strdup(executable);
        Add_process(list,temp);
    }
}
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)
{
    if (signum != SIGALRM){
        fprintf(stderr,"Sigalrm handler called for a different signal");
        exit(1);
    }
    kill((list->root)->pid,SIGSTOP); //Idio me proigoumeni askisi
}
```

```c
static void sigchld_handler(int signum)
{
    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 (!flag) { //THANATOS CHILD
                struct process* temp;
                temp = Remove_process(list);
                //if(temp->name != NULL)
                free(temp->name);
                free(temp);
                if(!(list->counter)){
                    exit(3);
                }
                if (alarm(SCHED_TQ_SEC) < 0) {
                    perror("ALARM");
                    exit(1);
                }
                kill((list->root)->pid, SIGCONT);
            }
            else{
            flag = 0;
            }
        }
        if (WIFSTOPPED(status)) {
            Add_process(list,Remove_process(list));
            if (alarm(SCHED_TQ_SEC) < 0) {
                perror("alarm");
```

```
                exit(1);
            }
            kill((list->root)->pid, SIGCONT);
        }
    }
}

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

    sigemptyset(&sigset);
    sigaddset(&sigset, SIGALRM);
    sigaddset(&sigset, SIGCHLD);
    if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
        perror("signals_disable: sigprocmask");
        exit(1);
    }
}

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

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


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

    sa.sa_handler = sigchld_handler;
    sa.sa_flags = SA_RESTART;
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGCHLD);
    sigaddset(&sigset, SIGALRM);
    sa.sa_mask = sigset;
```

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

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

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

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

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

    raise(SIGSTOP);
    execve(executable, newargv, newenviron);
    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];
    struct process* temp = Create_process(p,current_id);
    temp->name = strdup("Shell");
    Add_process(list,temp);
    current_id++;
}

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;
        }
    }
}
```

```c
int main(int argc, char *argv[])
{
    int nproc;
    /* Two file descriptors for communication with the shell */
    static int request_fd, return_fd;
    current_id = 0;
    pid_t pid;
    struct process* proc;
    list = Create_list();
    /* Create the shell. */
    sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
    /* DONE: add the shell to the scheduler's tasks */
    /*
     * For each of argv[1] to argv[argc - 1],
     * create a new child process, add it to the process list.
     */
    nproc = argc-1; /* number of proccesses goes here */
    if (nproc == 0) {
        fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
        exit(1);
    }
    int i;
    for(i = 0; i < nproc; i++){
        pid = fork();
        if(pid < 0){
            perror("FORK ERROR");
            exit(1);
        }
        if(pid > 0){
            proc = Create_process(pid,current_id);
            proc->name = strdup(argv[i+1]);
            current_id++;
            Add_process(list,proc);
        }
        else if(pid == 0){
            char *newargv[] = {argv[i+1], NULL, NULL, NULL };
            char *newenviron[] = { NULL };
            raise(SIGSTOP);
            execve(argv[i+1],newargv,newenviron);
            exit(1);
        }
    }
    /* Wait for all children to raise SIGSTOP before exec()ing. */
    wait_for_ready_children(nproc);

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

    if (alarm(SCHED_TQ_SEC) < 0) {
        perror("alarm");
        exit(1);
    }
    kill((list->root)->pid, SIGCONT);
    shell_request_loop(request_fd, return_fd);
    /* Now that the shell is gone, just loop forever
    while (pause())
        ;

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

Ενδεικτικά η εκτέλεση του προγράμματος μας δίνει το ακόλουθο αποτέλεσμα:

```
prog[7380]: This is message 99
My PID = 7376: Child PID = 7380 has been stopped by a signal, signo = 19
prog[7381]: This is message 124
prog[7381]: This is message 125
^C
oslabb30@orion:~/giorgos/ask4$ vim scheduler-shell.c
oslabb30@orion:~/giorgos/ask4$ ./scheduler-shell prog prog prog prog
My PID = 7432: Child PID = 7433 has been stopped by a signal, signo = 19
My PID = 7432: Child PID = 7434 has been stopped by a signal, signo = 19
My PID = 7432: Child PID = 7435 has been stopped by a signal, signo = 19
My PID = 7432: Child PID = 7436 has been stopped by a signal, signo = 19

This is the Shell. Welcome.

Shell> My PID = 7432: Child PID = 7437 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 200, delay = 117
prog[7434]: This is message 0
prog[7434]: This is message 1
prog[7434]: This is message 2
prog[7434]: This is message 3
prog[7434]: This is message 4
prog[7434]: This is message 5
My PID = 7432: Child PID = 7434 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 200, delay = 99
prog[7435]: This is message 0
prog[7435]: This is message 1
pprog[7435]: This is message 2
prog[7435]: This is message 3

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

Currently running[prog]:with pid 7435 and serial_no 2

[prog]:with pid 7436 and serial_no 3

[prog]:with pid 7437 and serial_no 4

[Shell]:with pid 7433 and serial_no 0

[prog]:with pid 7434 and serial_no 1

Shell> prog[7435]: This is message 4
prog[7435]: This is message 5
My PID = 7432: Child PID = 7435 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 200, delay = 146
prog[7436]: This is message 0
prog[7436]: This is message 1
prog[7436]: This is message 2
prog[7436]: This is message 3
prog[7436]: This is message 4
My PID = 7432: Child PID = 7436 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 200, delay = 127
prog[7437]: This is message 0
```

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;*

   Πάντα όταν εκτελούμε την εντολή 'p' ως τρέχουσα διεργασία εμφανίζεται να είναι ο φλοιός (id 0). Αυτό είναι αναμενόμενο, καθώς η εκτύπωση των διεργασιών γίνεται μόνο όταν τρέχουσα διεργασία είναι ο φλοιός. Δηλαδή θα ήταν αδύνατο να έχουμε άλλη διεργασία ως τρέχουσα διεργασία στη λίστα διεργασιών, καθώς η εντολή 'p' δίνεται μόνο όταν κεφάλι της λίστας μας είναι ο 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*

   Οι συναρτήσεις signals_disable() και signals_enable() χρησιμοποιούνται για την απενεργοποίηση και ενεργοποίηση λήψης σημάτων αντίστοιχα. Είναι απαραίτητο να χρησιμοποιηθούν αυτές οι συναρτήσεις ώστε να είμαστε σίγουροι ότι όσο εξυπηρετούνται οι αιτήσεις που κάνουμε στο φλοιό δε θα γίνει χειρισμός άλλου σήματος.
   Έτσι εξασφαλίζουμε ότι δε θα πειραχτούν οι δομές που χρησιμοποιούνται εκείνη τη στιγμή. Αν αφήναμε να γίνονται κανονικά οι χειρισμοί σημάτων, θα ήταν πιθανό να τροποποιηθεί η λίστα διεργασιών μας.

1.3 Υλοποίηση προτεραιοτήτων στο χρονοδρομολογητή

Αποφασίσαμε να αντιμετωπίσουμε το ζητούμενο πρόβλημα χρησιμοποιώντας δύο λίστες. Μία λίστα όπου εντάσσονται όλα αρχικά (αυτή είναι η LOW priority list) και άλλη μία όπου εντάσσονται διεργασίες μετά από την εντολή h του φλοιού (αυτή είναι η HIGH priority list). Προφανώς, αν η δεύτερη λίστα είναι μη κενή, τότε εκτελούνται ΜΟΝΟ διεργασίες από εκεί.

Δεν επισυνάπτουμε φωτογραφίες της λειτουργίας του προγράμματος καθώς έχει πολλές λειτουργίες και είναι άσκοπο. Αναμένουμε να δείξουμε την πλήρη λειτουργία του προγράμματος μας όταν θα εξεταστούμε.

```c
#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 "list3.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 list *list, *list2;
int current_id;
char flag; // if 0 then remove head

static void sched_print_tasks(void)
{
        struct process* temp;
        temp = list2->root;
        printf("\nCurrently running->");
        while((temp != NULL)){
                //fprintf(stderr,"now printing list2 %d\n",temp->id);
                printf("[%s](list2):with pid %ld and serial_no %d\n",temp->name,(long)temp->pid,temp->id);
                temp = temp->next;
        }
        temp = list->root;
        if(temp == NULL){
                return;
        }
        do{
                printf("[%s](low):with pid %ld and serial_no %d\n",temp->name,(long)temp->pid,temp->id);
        }while((temp = temp->next) != NULL);

}
static int sched_kill_task_by_id(int id)
{
        if (list->root != NULL  && list->root->id == id) {
                pid_t pid = Destroy_process(list,id);
                kill(pid,SIGKILL);
                list->counter = list->counter - 1;
                flag=1;
                return 0;

        }
        else if (list2->root !=NULL && list2->root->id == id) {
                kill(list2->root->pid,SIGKILL);
                return 0;
```

```
                }
else    {
                pid_t pid = Destroy_process(list,id);
                list->counter = list->counter - 1;
                if(pid == -1){
                        list->counter++;
                        pid = Destroy_process(list2,id);
                        list2->counter = list2->counter - 1;
                        if(pid == -1){
                                fprintf(stderr,"kill:Requested id doesn't exist\n");
                                list2->counter++;
                                return 0;
                        }
                }
                flag = 1;
                kill(pid,SIGKILL);
                return 0;
        }
}

static void sched_create_task(char *executable)
{
        pid_t pid;
        pid = fork();
        if(pid < 0){
                fprintf(stderr,"FORK");
                exit(1);
        }
        else if(pid == 0){
                char *newargv[] = {executable, NULL, NULL, NULL };
                char *newenviron[] = { NULL };
                raise(SIGSTOP);
                execve(executable,newargv,newenviron);
                exit(1);
        }
        else{
                struct process* temp = Create_process(pid,current_id++);
                temp->name = strdup(executable);
                Add_process(list,temp);
        }
}

static int sched_high_task_by_id (int id) {
        char ACTIVE = 0;
        if(list->root->id == id){
                ACTIVE = 1;
        }
        struct process* removed = Remove_process_by_id(list,id);
        list->counter = list->counter - 1;
        if(removed == NULL){
                list->counter++;
                fprintf(stderr,"set_list2:Requested id doesn't exist in low\n");
                return 0;
        }
```

```
        else {
                if(list2->root == NULL){
                        Add_process(list2,removed);
                        struct process* temp;
                        temp = list2->root;
                        temp = list2->root;
                        if(!ACTIVE){
                                kill(list->root->pid,SIGSTOP);
                        }

                        return 0;
                }
                else {
                        Add_process(list2,removed);
                        return 0;
                }
        }
}
static int sched_low_task_by_id(int id){
        char ACTIVE = 0;
        if(list2->root->id == id) {
        ACTIVE = 1;
        }
        struct process* removed = Remove_process_by_id(list2,id);
        list2->counter = list2->counter - 1;
        if(removed == NULL){
                fprintf(stderr,"set_low:Requested id doesn't exist in list2\n");
                return 0;
        }
        else{
                Add_process(list,removed);
                if(ACTIVE) kill(removed->pid,SIGSTOP);
                return 0;
        }
}

/* Process requests by the shell.  */
static int process_request(struct request_struct *rq)
{
        switch (rq->request_no) {
                case REQ_PRINT_TASKS:
                        sched_print_tasks();
                        return 0;

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

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

                case REQ_HIGH_TASK:
                        return sched_high_task_by_id(rq->task_arg);

                case REQ_LOW_TASK:
```

```
                        return  sched_low_task_by_id(rq->task_arg);

                default:
                        return -ENOSYS;
        }
}

/*
 * SIGALRM handler
 */
static void sigalrm_handler(int signum)
{
        if (signum != SIGALRM){
                fprintf(stderr,"Sigalrm handler called for a different signal");
                exit(1);
        }
        if(list2->root != NULL){
                kill((list2->root)->pid,SIGSTOP);
        }
        else{
                kill((list->root)->pid,SIGSTOP);
        }
}

/*
 * SIGCHLD handler
 */
static void sigchld_handler(int signum)
{
        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("WAIT_PID");
                        exit(1);
                }
                if (p == 0)
                        break;
```

```
explain_wait_status(p, status);

if (WIFEXITED(status) || WIFSIGNALED(status)) { //AN KATI PETHANE FUSIOLOGIKA H APO SIGKILL
        if(list2->root == NULL) { //AN EIMASTE STIN LOW LISTA
                if(!flag){ //SIGKILLED
                        struct process* temp;
                        temp = Remove_process(list);
                        if(temp->name != NULL)free(temp->name);
                        free(temp);
                        if(!(list->counter)){
                                exit(3);
                        }
                        if (alarm(SCHED_TQ_SEC) < 0) {
                                perror("ALARM");
                                exit(1);
                        }
                        kill((list->root)->pid, SIGCONT);
                }
                else {
                flag = 0;
                }
        }
        else { //AN EIMASTE STIN HIGH LISTA
                if (!flag) {//SIGKILLED
                        struct process* temp;
                        temp = Remove_process(list2);
                        if(temp->name != NULL)
                        free(temp->name);
                        free(temp);
                        if(!(list2->counter)){
                                if(!(list->counter)){
                                        exit(3); //END OF PROGRAM
                                }
                                else {
                                        if (alarm(SCHED_TQ_SEC) < 0) {
                                                perror("ALARM");
                                                exit(1);
                                        }
                                        kill((list->root)->pid, SIGCONT);
                                        return;
                                }
                        }
                        if (alarm(SCHED_TQ_SEC) < 0) {
                                perror("ALARM");
                                exit(1);
                        }
                        kill((list2->root)->pid, SIGCONT);
                }
                else {
                flag = 0;
                }
        }
}
```

```
                if (WIFSTOPPED(status)) { //SIGSTOPPED
                        /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
                        if (list2->root == NULL){ //LOW PRIORITY LIST
                                Add_process(list,Remove_process(list));
                                /* Setup the alarm again */
                                if (alarm(SCHED_TQ_SEC) < 0) {
                                        perror("alarm");
                                        exit(1);
                                }
                                kill((list->root)->pid, SIGCONT);
                        }
                        else { //HIGH PRIORITY LIST
                                Add_process(list2,Remove_process(list2));
                                if (alarm(SCHED_TQ_SEC) < 0) {
                                        perror("alarm");
                                        exit(1);
                                }
                                kill((list2->root)->pid, SIGCONT);
                        }
                }
        }
}

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

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
                perror("signals_disable: sigprocmask");
                exit(1);
        }
}

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

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

```c
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);
        }
        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.
 */
```

```c
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("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];
        struct process* temp = Create_process(p,current_id);
        temp->name = strdup("Shell");
        Add_process(list,temp);
        current_id++;
}

static void shell_request_loop(int request_fd, int return_fd)
{
        int ret;
        struct request_struct rq;
        for (;;) {
                if (read(request_fd, &rq, sizeof(rq)) != sizeof(rq)) {
                        perror("SHELL ERROR");
                        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;
                }
        }
```

```c
int main(int argc, char *argv[])
{
        int nproc;
        static int request_fd, return_fd;
        current_id = 0;
        pid_t pid;
        struct process* proc;
        list = Create_list(); // LOW PRIORITY
        list2 = Create_list(); //HIGH PRIORITY
        sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
        nproc = argc-1; /* number of proccesses goes here */
        if (nproc == 0) {
                fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                exit(1);
        }
        int i;
        for(i = 0; i < nproc; i++){
                pid = fork();
                if(pid < 0){
                        fprintf(stderr,"kako fork\n");
                        exit(1);
                }
                if(pid > 0){
                        proc = Create_process(pid,current_id);
                        proc->name = strdup(argv[i+1]);
                        current_id++;
                        Add_process(list,proc);
                }
                else if(pid == 0){
                        char *newargv[] = {argv[i+1], NULL, NULL, NULL };
                        char *newenviron[] = { NULL };
                        raise(SIGSTOP);
                        execve(argv[i+1],newargv,newenviron);
                        fprintf(stderr,"Xasame\n");
                        exit(1);
                }
        }
        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(nproc);

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

        if (alarm(SCHED_TQ_SEC) < 0) {
                perror("alarm");
                exit(1);
        }
        kill((list->root)->pid, SIGCONT);
        shell_request_loop(request_fd, return_fd);
        while (pause())
                ;

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

Ερωτήσεις

1. *Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας*

Ένα σενάριο λιμοκτονίας για το χρονοδρομολογητή μας είναι το ακόλουθο:
Αν έχουμε μια διεργασία η οποία έχει low priority και δεν αλλάξει η προτεραιότητά της ποτέ και πάντα έχουμε διεργασίες με high priority (ή αν τελειώσουν προσθέτουμε νέες).
Σε τέτοια περίπτωση, σύμφωνα με τον χρονοδρομολογητή και την υλοποίηση μας η διεργασία αυτή δε θα εκτελεστεί ποτέ καθώς πάντα εκτελούνται (όσο υπάρχουν) διεργασίες με high priority.