ΕΘΝΙΚΌ ΜΕΤΣΌΒΙΟ ΠΟΛΥΤΕΧΝΕΊΟ

ΣΧΟΛΉ ΗΛΕΚΤΡΟΛΌΓΩΝ ΜΗΧΑΝΙΚΏΝ ΚΑΙ ΜΗΧΑΝΙΚΏΝ ΥΠΟΛΟΓΙΣΤΏΝ

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΉΜΑΤΑ ΥΠΟΛΟΓΙΣΤΩΝ

(2019-2020)

4η ΕΡΓΑΣΤΗΡΙΑΚΉ ΑΝΑΦΟΡΆ

Ομάδα: oslaba36

Ονοματεπώνυμο: Χρήστος Τσούφης – 03117176

Νίκος Χαράλαμπος Χαιρόπουλος – 03119507

Εκτέλεση Εργαστηρίου: 18/05/2020

Σημείωση: Η άσκηση αυτή επιλύθηκε ατομικά από τον καθένα μας σε επίπεδο υλοποίησης για την καλύτερη κατανόηση των εννοιών. Παρακάτω παρουσιάζεται η κοινή αναφορά.

Για λόγους πληρότητας παρακάτω παρουσιάζεται η υλοποίηση του Χρήστου. Η υλοποίηση του Νίκου είναι στο zip Nikos_report (αφού η απάντηση στα ερωτήματα δεν εξαρτάται από τον τρόπο υλοποίησης).

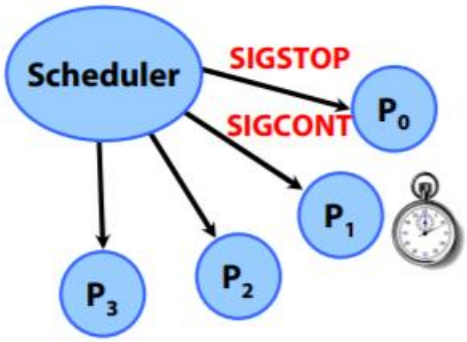
Άσκηση 4: Χρονοδρομολόγηση

1.1 Υλοποίηση χρονοδρομολογητή κυκλικής επαναφοράς στο χώρο χρήστη

Ζητείται η υλοποίηση ενός χρονοδρομολογητή κυκλικής επαναφοράς (round-robin). Ο χρονοδρομολογητής εκτελείται ως γονική διεργασία, στο χώρο χρήστη, κατανέμοντας τον υπολογιστικό χρόνο σε διεργασίες-παιδιά. Για τον έλεγχο των διεργασιών ο χρονοδρομολογητής θα χρησιμοποιεί τα σήματα SIGSTOP και SIGCONT για τη διακοπή και την ενεργοποίηση κάθε διεργασίας, αντίστοιχα.

Κάθε διεργασία εκτελείται για χρονικό διάστημα το πολύ ίσο με το κβάντο χρόνου tq. Αν η διεργασία τερματιστεί πριν από το τέλος του κβάντου χρόνου, ο χρονοδρομολογητής την αφαιρεί από την ουρά των έτοιμων διεργασιών και ενεργοποιεί την επόμενη. Αν το κβάντο χρόνου εκπνεύσει χωρίς η διεργασία να έχει ολοκληρώσει την εκτέλεσή της, τότε αυτή διακόπτεται, τοποθετείται στο τέλος της ουράς έτοιμων διεργασιών και ενεργοποιείται η επόμενη.

Η λειτουργία του χρονοδρομολογητή ζητείται να είναι ασύγχρονη, βασισμένη σε σήματα. Ένας χρονοδρομολογητής χώρου πυρήνα ενεργοποιείται από διακοπές χρονιστή. Αντίστοιχα, ο υπό εξέταση χρονοδρομολογητής θα χρησιμοποιεί τα σήματα SIGALRM και SIGCHLD για να ενεργοποιείται στις εξής δύο περιπτώσεις:



- Εκπνοή κβάντου χρόνου: Όταν το κβάντο χρόνου εκπνεύσει, ο χρονοδρομολογητής σταματά την τρέχουσα διεργασία. Η περίπτωση αυτή αντιστοιχεί σε χειρισμό του σήματος SIGALRM.
- Παύση/τερματισμός διεργασίας: Όταν η τρέχουσα διεργασία πεθάνει ή σταματήσει, επειδή εξέπνευσε το κβάντο χρόνου της, ο χρονοδρομολογητής διαλέγει την επόμενη από την ουρά, θέτει τον χρονιστή ώστε να παραδοθεί σήμα SIGALRM μετά από tq δευτερόλεπτα, και την ενεργοποιεί. Η περίπτωση αυτή αντιστοιχεί σε χειρισμό του σήματος SIGCHLD.

Το τελικό παραδοτέο είναι το πρόγραμμα του χρονοδρομολογητή, το οποίο θα εκτελείται με ορίσματα τα εκτελέσιμα προς χρονοδρομολόγηση: $ ./scheduler prog prog prog prog

Για κάθε όρισμα κατασκευάζεται μία διεργασία που εκτελεί το αντίστοιχο πρόγραμμα. Σε κάθε διεργασία αντιστοιχίζεται σειριακός αριθμός id. Μετά τη δημιουργία των διεργασιών ξεκινά η διαδικασία χρονοδρομολόγησης. Ο χρονοδρομολογητής εμφανίζει κατάλληλα μηνύματα κατά την ενεργοποίηση, διακοπή και τερματισμό των διεργασιών. Όταν όλες οι διεργασίες έχουν ολοκληρώσει την εκτέλεσή τους, τερματίζεται.

Στον κατάλογο /home/oslab/code/sched σας δίνονται: το πρόγραμμα prog.c, το οποίο θα εκτελείται στις υπό χρονοδρομολόγηση διεργασίες, το execve-example.c για τη χρήση της κλήσης συστήματος execve() και ένας σκελετός του χρονοδρομολογητή, scheduler.c.

Όπου, το execve-example.c :

```c
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    char executable[] = "prog";
    char *newargv[] = { executable, NULL, NULL, NULL };
    char *newenviron[] = { NULL };

    printf("I am %s, PID = %ld\n",
        argv[0], (long)getpid());
    printf("About to replace myself with the executable %s...\n",
        executable);
    sleep(2);

    execve(executable, newargv, newenviron);

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

Το proc-common.c :

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

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

#include "proc-common.h"

void
wait_forever(void)
{
    do {
        sleep(100);
    } while (1);
}
```

```c
/*
 * This function performs some not-so-useful computation.
 * Its amount is determined by the value of count.
 */
void compute(int count)
{
    long i;
    volatile long junk;

    junk = 0;
    for (i = 0; i < count * 1000000; i++) {
        junk++;
    }
}

/*
 * Changes the process name, as appears in ps or pstree,
 * using a Linux-specific system call.
 */
void
change_pname(const char *new_name)
{
    int ret;
    ret = prctl(PR_SET_NAME, new_name);
    if (ret == -1){
        perror("prctl set_name");
        exit(1);
    }
}

/*
 * This function receives an integer status value,
 * as returned by wait()/waitpid() and explains what
 * has happened to the child process.
 *
 * The child process may have:
 *    * been terminated because it received an unhandled signal (WIFSIGNALED)
 *    * terminated gracefully using exit() (WIFEXITED)
 *    * stopped because it did not handle one of SIGTSTP, SIGSTOP, SIGTTIN, SIGTTOU
 *      (WIFSTOPPED)
 *
 * For every case, a relevant diagnostic is output to standard error.
 */
void
explain_wait_status(pid_t pid, int status)
{
    if (WIFEXITED(status))
```

```
        fprintf(stderr, "My PID = %ld: Child PID = %ld terminated normally, exit sta
tus = %d\n",
            (long)getpid(), (long)pid, WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        fprintf(stderr, "My PID = %ld: Child PID = %ld was terminated by a signal, s
igno = %d\n",
            (long)getpid(), (long)pid, WTERMSIG(status));
    else if (WIFSTOPPED(status))
        fprintf(stderr, "My PID = %ld: Child PID = %ld has been stopped by a signal,
 signo = %d\n",
            (long)getpid(), (long)pid, WSTOPSIG(status));
    else {
        fprintf(stderr, "%s: Internal error: Unhandled case, PID = %ld, status = %d\
n",
            __func__, (long)pid, status);
        exit(1);
    }
    fflush(stderr);
}


/*
 * Make sure all the children have raised SIGSTOP,
 * by using waitpid() with the WUNTRACED flag.
 *
 * This will NOT work if children use pause() to wait for SIGCONT.
 */
void
wait_for_ready_children(int cnt)
{
    int i;
    pid_t p;
    int status;

    for (i = 0; i < cnt; i++) {
        /* Wait for any child, also get status for stopped children */
        p = waitpid(-1, &status, WUNTRACED);
        explain_wait_status(p, status);
        if (!WIFSTOPPED(status)) {
            fprintf(stderr, "Parent: Child with PID %ld has died unexpectedly!\n",
                (long)p);
            exit(1);
        }
    }
}

/*
 * Print the process tree rooted at process with PID p.
 */
```

```c
void
show_pstree(pid_t p)
{
    int ret;
    char cmd[1024];

    snprintf(cmd, sizeof(cmd), "echo; echo; pstree -G -c -p %ld; echo; echo",
        (long)p);
    cmd[sizeof(cmd)-1] = '\0';
    ret = system(cmd);
    if (ret < 0) {
        perror("system");
        exit(104);
    }
}


/*
 * Create a shared memory area, usable by all descendants of the calling process.
 */
void *create_shared_memory_area(unsigned int numbytes)
{
    int pages;
    void *addr;

    if (numbytes == 0) {
        fprintf(stderr, "%s: internal error: called for numbytes == 0\n", __func__);
        exit(1);
    }

    /* Determine the number of pages needed, round up the requested number of pages
*/
    pages = (numbytes - 1) / sysconf(_SC_PAGE_SIZE) + 1;

    /* Create a shared, anonymous mapping for this number of pages */
    addr = mmap(NULL, pages * sysconf(_SC_PAGE_SIZE),
        PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    if (addr == MAP_FAILED) {
        perror("create_shared_memory_area: mmap failed");
        exit(1);
    }

    return addr;
}
```

To proc-common.h :

```
#ifndef PROC_COMMON_H
#define PROC_COMMON_H

/*********************************************************************
 * Helper Functions
 */

/* does useless computation */
void compute(int count);

/* Does nothing and never returns. */
void wait_forever(void);

/*
 * Print a nice diagnostic based on {pid, status}
 * as returned by wait() or waitpid().
 */
void explain_wait_status(pid_t pid, int status);

/*
 * This function makes sure a number of children processes
 * have raised SIGSTOP, by using waitpid() with the WUNTRACED flag.
 *
 * This will NOT work if children use pause() to wait for SIGCONT.
 */
void wait_for_ready_children(int cnt);

/* Change the name of the process. */
void change_pname(const char *new_name);

/* Print the process tree rooted at process with PID p. */
void show_pstree(pid_t p);

/*
 * Create a shared memory area, usable by all descendants of the calling process.
 */
void *create_shared_memory_area(unsigned int numbytes);

#endif /* PROC_COMMON_H */
```

To prog.c :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

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

```c
#define NMSG 200
#define DELAY 130

int main(int argc, char *argv[])
{
    int i, delay, pid;


    /*
     * Print a number of messages,
     * use a random delay, so that processes terminate
     * in random order.
     */

    pid = getpid();
    srand(pid);
    delay = 30 + ((double)rand() / RAND_MAX) * DELAY;
    printf("%s: Starting, NMSG = %d, delay = %d\n",
        argv[0], NMSG, delay);

    for (i = 0; i < NMSG; i++) {
        printf("%s[%d]: This is message %d\n", argv[0], pid, i);
        compute(delay);
    }

    return 0;
}
```

To 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 "helper.h"

/* Compile-time parameters. */
#define SCHED_TQ_SEC 2                /* time quantum */
```

```c
#define TASK_NAME_SZ 60               /* maximum size for a task's name */

process_list* p_list;

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
    printf("\n*** SCHEDULER: Going to stop process [id]: %d\n",
            p_list->head->id);
    kill(p_list->head->pid, SIGSTOP);
}

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

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

        // Check if head process changed status
        if (pid < 0) {
            perror("waitpid < 0");
            exit(1);
        } else if (pid == 0) {
            break;
        } else if (pid > 0) {
            if (pid == p_list->head->pid) {
                process *p;

                // Process has stopped
                if (WIFSTOPPED(status)) {
                    printf ("*** SCHEDULER: STOPPED: Process [name]: %s  [id]: %d\n"
,
                            p_list->head->name, p_list->head->id);
                    p = get_next(p_list);

                // Process has exited
                } else if (WIFEXITED(status)) {
                    printf ("*** SCHEDULER: EXITED: Process [name]: %s  [id]: %d\n",
                            p_list->head->name, p_list->head->id);
```

```
                    p = pop(p_list);
                    free_process(p);
                    if (empty(p_list)) {
                        printf ("\n***SCHEDULER: No more processes to schedule");
                        exit(0);
                    }
                    p = p_list->head;
                }
                else if (WIFSIGNALED(status)) {
                    printf ("*** SCHEDULER: Child killed by signal: Process [name]:
%s  [id]: %d\n",
                            p_list->head->name, p_list->head->id);

                    p = pop(p_list);
                    free_process(p);
                    if (empty(p_list)) {
                        printf ("*** SCHEDULER: No more processes to schedule");
                        exit(0);
                    }
                    p = p_list->head;
                } else {
                    printf ("*** SCHEDULER: Something strange happened with: Process
 [name]: %s  [id]: %d\n",
                            p_list->head->name, p_list->head->id);
                            exit(1);
                }

                printf ("*** SCHEDULER: Next process to continue: Process [name]: %s
  [id]: %d\n\n",
                        p->name, p->id);

                // It's the turn of next process to continue
                kill (p->pid, SIGCONT);
                alarm (SCHED_TQ_SEC);
            } else {
                /* Handle the case that a different than the head process
                 * has changed status
                 */

                process *pr = erase_proc_by_pid(p_list, pid);
                if (pr != NULL ) {
                    printf ("*** SCHEDULER: A process other than the head has Change
d state unexpectedely: Process [name]: %s  [id]: %d\n",
                    pr->name, pr->id);
                    free_process(pr);
                }
            }
```

```
        }
    }
}

/* 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_flags = SA_RESTART;

    // Specify signals to be blocked while the handling funvtion runs
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGCHLD);
    sigaddset(&sigset, SIGALRM);
    sa.sa_mask = sigset;

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

    // TODO In exercise the sa handler was reassigned, does it work?
    sa.sa_handler = sigalrm_handler;
    if (sigaction(SIGALRM, &sa, NULL) < 0) {
        perror("sigaction: sigalrm");
        exit(1);
    }
}

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

    nproc = argc - 1; /* number of proccesses goes here */
    if (nproc == 0) {
        fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
        exit(1);
```

```
    }

    p_list = initialize_empty_list();

    int i;
    for (i = 1; i < argc; i++) {
        pid_t pid;
        pid = fork();
        if (pid < 0) {
            perror("fork");
            exit(1);
        }
        if (pid == 0) {
            printf("test");
            raise(SIGSTOP);
            char filepath[TASK_NAME_SZ];
            sprintf(filepath, "./%s", argv[i]);
            // TODO
            char* args[] = {filepath, NULL};
            if (execvp(filepath, args)) {
                perror("execvp");
                exit(1);
            }
        }
        process *p = process_create(pid, argv[i]);
        push(p_list, p);
        printf("Process name: %s id: %d is created.\n",
            argv[i], p->id);
    }

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

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

    printf("Scheduler dispatching the first process...\n");
    kill(p_list->head->pid, SIGCONT);
    alarm(SCHED_TQ_SEC);

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

To helper.c :

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#include "helper.h"

// Creates a process struct, containing the name and pid of process
process *process_create(pid_t pid, const char *name) {
    static int id = 0;
    process* new_process = (process*)malloc(sizeof(process));
    if (new_process == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }

    // Strdup allocates memory and copies string
    new_process->name = strdup(name);
    new_process->pid = pid;
    new_process->id = id++;
    new_process->next = NULL;
    return new_process;
}

// Return 1 if list empty, else 0
int empty(process_list* list) {
    if (list->head == NULL) {
        return 1;
    } else {
        return 0;
    }
}

// Removes and returns the head of the list
process* pop(process_list* list) {
    if(empty(list)) {
        printf("Empty list\n");
        exit(1);
    }
    process* temp;
    if(list->head == list->tail) {
        temp = list->head;
        list->head =  NULL;
        list->tail = NULL;
        list->size = 0;
        return temp;
```

```
    } else {
        temp = list->head;
        list->head = list->head->next;
        list->tail->next = list->head;
        list->size--;
        return temp;
    }
}

// Puts a process at the tail of the list
int  push(process_list* l, process* new_p) {
    if (empty(l)) {
        l->head = l->tail = new_p;
        l->head->next = l->head;
        l->size = 1;
        return 1;
    }

    new_p->next = l->head;
    l->tail->next = new_p;
    l->tail = new_p;
    l->size++;
    return 1;
}

// Erases process with specified id
process* erase_proc_by_id(process_list * l, int id) {
    if (empty(l))
        return NULL;

    if (l->head->id == id)
        return pop(l);

    process* temp;
    temp = l->head;
    temp = temp->next;

    while (temp->next->id != id && temp->next != l->tail)
        temp = temp->next;

    if (temp->next->id == id) {
        process* ret = temp->next;
        l->size--;
        temp->next = ret->next;
        if(ret == l->tail)
            l->tail = temp;
        return ret;
    }
```

```
    return NULL;
}

// Erases process with specified pid
process* erase_proc_by_pid(process_list * l, int id) {
    if (empty(l))
        return NULL;

    if (l->head->pid == id)
        return pop(l);

    process * temp;
    temp = l->head;

    while (temp->next->pid != id && temp->next != l->tail)
        temp = temp->next;

    if (temp->next->pid == id) {
        process * ret = temp->next;
        l->size--;
        temp->next = temp->next->next;
        if(ret == l->tail)
            l->tail = temp;
        return ret;
    }

    return NULL;
}

// Returns pointer to process with specified pid
process* get_proc_by_pid(process_list * l, int id) {
    if (empty(l))
        return NULL;

    if (l->head->pid == id)
        return l->head;

    process * temp;
    temp = l->head;

    while (temp->next->pid != id && temp->next != l->tail)
        temp = temp->next;

    if (temp->next->pid == id) {
        return temp->next;
    }
```

```
    return NULL;
}

// Returns pointer to process with specified id
process* get_proc_by_id(process_list * l, int id) {
    if (empty(l))
        return NULL;

    if (l->head->id == id)
        return l->head;

    process * temp;
    temp = l->head;

    while (temp->next->id != id && temp->next != l->tail)
        temp = temp->next;

    if (temp->next->id == id) {
        return temp->next;
    }

    return NULL;
}

// Returns pointer to next process to be processed
process* get_next(process_list * l) {
    if(empty(l)) {
        printf("Empty list\n");
        exit(1);
    }

    l->tail = l->head;
    l->head = l->head->next;
    return l->head;
}

// version to be compatible with multiple lists
process* my_get_next(process_list * l) {
    if(empty(l)) {
        printf("Empty list\n");
        return NULL;
    }

    l->tail = l->head;
    l->head = l->head->next;
    return l->head;
}
```

```
// Empties whole list
void clear(process_list * l) {
    process * i;
    while(!empty(l)) {
        i = pop(l);
        free(i->name);
        free(i);
    }
}

// Initializes an empty list
process_list* initialize_empty_list(void) {
    // Initialize an empty list
    process_list* p_list;
    p_list = (process_list *) malloc (sizeof(process_list));
    p_list->head = NULL;
    p_list->tail = NULL;
    p_list->size = 0;
    return p_list;
}

// Frees allocated process struct memory
void free_process(process* p) {
    free(p->name);
    free(p);
}

void print_list(process_list* l, process* current_p) {
    process* tmp = l->head;
    printf("\n***********************************\n");
    if (empty(l)) {
        printf("Empty list\n");
    } else {
        while (tmp != l->tail) {
            if (tmp->id == current_p->id) {
                printf("CURRENT:");
            }
            printf("--> pid: %ld, id: %d,  name: %s\n", (long)tmp->pid, tmp-
>id, tmp->name);
            tmp = tmp->next;
        }
        if (tmp != NULL) {
            if (tmp->id == current_p->id) {
                printf("CURRENT:");
            }
            printf("--> pid: %ld, id: %d,  name: %s\n", (long)tmp->pid, tmp-
>id, tmp->name);
        }
```

```
    }
    printf("************************************\n");
}

// Returns head element of high and low lists
process* get_head_of_lists(process_list* l, process_list* h) {
    if (!empty(h)) {
        return h->head;
    } else {
        return l->head;
    }
}

// Returns if empty high and low lists
int empty_lists(process_list* l, process_list* h) {
    return (empty(l) && empty(h));
}

// Returns next element from lists low and high
process* get_next_lists(process_list* l, process_list* h) {
    if (!empty(h)) {
        process* p = my_get_next(h);
        return p;
    } else {
        return my_get_next(l);
    }
}

// Pops next element from lists low and high
process* pop_list(process_list* l, process_list* h) {
    if (!empty(h)) {
        return pop(h);
    } else {
        return pop(l);
    }
}

process* get_proc_by_pid_list(process_list* l, process_list* h, int id) {
    process* res =  get_proc_by_pid(h, id);
    if (res == NULL) {
        res = get_proc_by_pid(l, id);
    }
    return res;
}

process* get_proc_by_id_list(process_list* l, process_list* h, int id) {
    process* res =  get_proc_by_id(h, id);
    if (res == NULL) {
```

```
        res = get_proc_by_id(l, id);
    }
    return res;
}

process* erase_proc_by_id_list(process_list* l, process_list* h, int id) {
    process* res =  erase_proc_by_id(h, id);
    if (res == NULL) {
        res = erase_proc_by_id(l, id);
    }
    return res;
}

process* erase_proc_by_pid_list(process_list* l, process_list* h, int id) {
    process* res =  erase_proc_by_pid(h, id);
    if (res == NULL) {
        res = erase_proc_by_pid(l, id);
    }
    return res;
}

// Moves process specified by id from list a into b
int move_from_to(process_list* a, process_list* b, int id) {
    process* res = erase_proc_by_id(a, id);
    if (res == NULL) {
        return 0; // FAIL
    }
    int status = push(b, res);
    return status;
}
/*
 * Color setting
 */
void red () {
      printf("\033[1;31m");
}

void yellow () {
      printf("\033[1;33m");
}

void green () {
      printf("\033[0;32m");
}

void reset () {
      printf("\033[0m");
}
```

To helper.h :

```
#ifndef helper_h
#define helper_h

#include <unistd.h>
#include <stddef.h>
typedef struct process_type {
    pid_t pid;
    int id;
    char* name;
    struct process_type* next;
} process;

typedef struct process_list_type {
    process* head;
    process* tail;
    size_t size;
} process_list;

process* process_create(pid_t pid, const char* name);

int empty(process_list* l);

process* pop(process_list* l);

int push(process_list* l, process* n);

process* get_proc_by_pid(process_list* l, int id);

process* get_proc_by_id(process_list* l, int id);

process* erase_proc_by_id(process_list* l, int id);

process* erase_proc_by_pid(process_list* l, int id);

process* get_next(process_list* l);

void clear(process_list* l);

process_list* initialize_empty_list(void);

void free_process(process* p);

void print_list(process_list* l, process* current_p);

int empty_lists(process_list* l, process_list* h);

process* get_head_of_lists(process_list* l, process_list* h);
```

```
process* get_next_lists(process_list* l, process_list* h);

process* pop_list(process_list* l, process_list* h);

process* get_proc_by_pid_list(process_list* l, process_list * h, int id);

process* get_proc_by_id_list(process_list * l, process_list* h, int id);

process* erase_proc_by_id_list(process_list * l, process_list* h, int id);

process* erase_proc_by_pid_list(process_list* l, process_list* h, int id);

int move_from_to(process_list* a, process_list* b, int id);

void red();
void yellow();
void green();
void reset();
#endif
```

To request.h :

```
#ifndef REQUEST_H_
#define REQUEST_H_

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

/* request ids */
enum request_enum {
    REQ_PRINT_TASKS,  /* print tasks */
    REQ_KILL_TASK,    /* kill ->task_arg task */
    REQ_EXEC_TASK,    /* execute ->exec_task_arg with priority ->prio_arg */
    REQ_HIGH_TASK,    /* set ->task_arg to be of high priority */
    REQ_LOW_TASK,     /* set ->task_arg to be of low priority */
};

#define EXEC_TASK_NAME_SZ 60

/* Structure describing system call. */
struct request_struct {
    /* System call number */
    enum request_enum request_no;

    /*
     * System call arguments.
     * Can you think of a better organization for these fields?
```

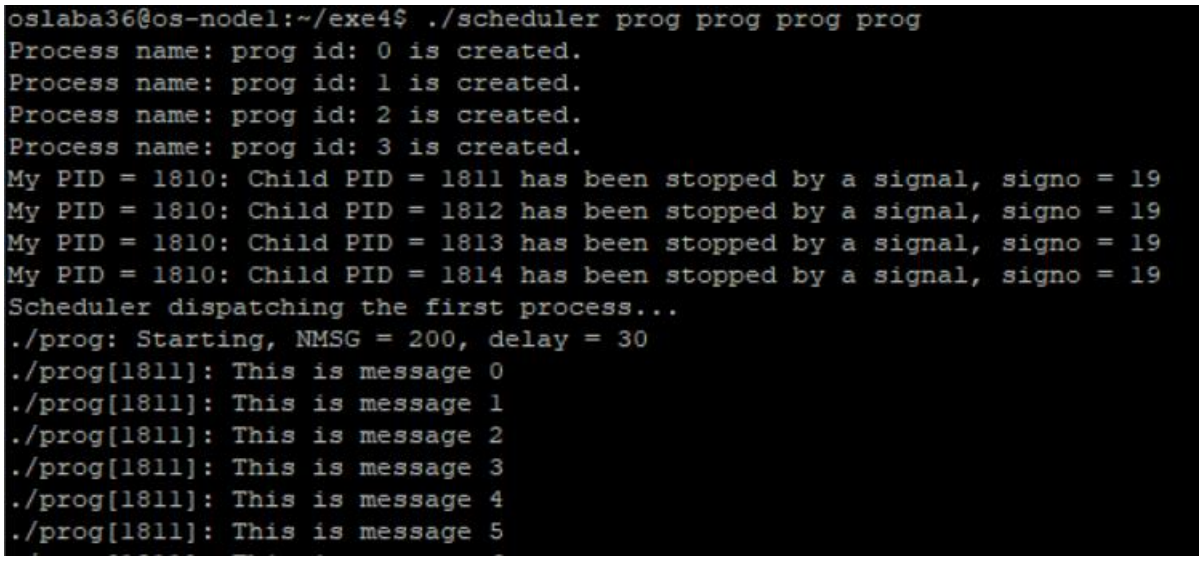
```
     * Contact the lab assistants before any changes to the structure.
     */
    int task_arg;
    char exec_task_arg[EXEC_TASK_NAME_SZ];
};

#endif /* REQUEST_H_ */
```

To Makefile :

```
#
# Makefile
#
# Operating Systems, Exercise 4
#
CC = gcc
#CFLAGS = -Wall -g
CFLAGS = -Wall -O2 -g -ggdb -ggdb3
all: scheduler scheduler-shell shell prog execve-example strace-test sigchld-example scheduler-priority
scheduler: scheduler.o proc-common.o helper.o
	$(CC) -o scheduler scheduler.o proc-common.o helper.o
scheduler-shell: scheduler-shell.o proc-common.o helper.o
	$(CC) -o scheduler-shell scheduler-shell.o proc-common.o helper.o
scheduler-priority: scheduler-priority.o proc-common.o helper.o
	$(CC) -o scheduler-priority scheduler-priority.o proc-common.o helper.o
scheduler-priority.o: scheduler-priority.c
	$(CC) -o scheduler-priority.o -c scheduler-priority.c
shell: shell.o proc-common.o helper.o
	$(CC) -o shell shell.o proc-common.o helper.o
helper.o: helper.c helper.h
	$(CC) -o helper.o -c helper.c
prog: prog.o proc-common.o
	$(CC) -o prog prog.o proc-common.o
execve-example: execve-example.o
	$(CC) -o execve-example execve-example.o
strace-test: strace-test.o
	$(CC) -o strace-test strace-test.o
```

```
sigchld-example: sigchld-example.o proc-common.o
	$(CC) -o sigchld-example sigchld-example.o proc-common.o
proc-common.o: proc-common.c proc-common.h
	$(CC) $(CFLAGS) -o proc-common.o -c proc-common.c
shell.o: shell.c proc-common.h request.h
	$(CC) $(CFLAGS) -o shell.o -c shell.c
scheduler.o: scheduler.c proc-common.h request.h
	$(CC) $(CFLAGS) -o scheduler.o -c scheduler.c
scheduler-shell.o: scheduler-shell.c proc-common.h request.h
	$(CC) $(CFLAGS) -o scheduler-shell.o -c scheduler-shell.c
prog.o: prog.c
	$(CC) $(CFLAGS) -o prog.o -c prog.c
execve-example.o: execve-example.c
	$(CC) $(CFLAGS) -o execve-example.o -c execve-example.c
strace-test.o: strace-test.c
	$(CC) $(CFLAGS) -o strace-test.o -c strace-test.c
sigchld-example.o: sigchld-example.c
	$(CC) $(CFLAGS) -o sigchld-example.o -c sigchld-example.c
clean:
	rm -f scheduler scheduler-shell shell prog execve-example strace-test
sigchld-example *.o
```

Το output (τυχαίο στιγμιότυπο):



```
*** SCHEDULER: Going to stop process [id]: 3
*** SCHEDULER: STOPPED: Process [name]: prog  [id]: 3
*** SCHEDULER: Next process to continue: Process [name]: prog  [id]: 0

./prog[1781]: This is message 11
./prog[1781]: This is message 12
./prog[1781]: This is message 13
./prog[1781]: This is message 14
./prog[1781]: This is message 15

*** SCHEDULER: Going to stop process [id]: 0
*** SCHEDULER: STOPPED: Process [name]: prog  [id]: 0
*** SCHEDULER: Next process to continue: Process [name]: prog  [id]: 1

./prog[1782]: This is message 31
./prog[1782]: This is message 32
./prog[1782]: This is message 33
./prog[1782]: This is message 34
./prog[1782]: This is message 35
./prog[1782]: This is message 36
./prog[1782]: This is message 37
./prog[1782]: This is message 38
./prog[1782]: This is message 39
./prog[1782]: This is message 40
./prog[1782]: This is message 41
./prog[1782]: This is message 42
./prog[1782]: This is message 43
./prog[1782]: This is message 44
./prog[1782]: This is message 45

*** SCHEDULER: Going to stop process [id]: 1
*** SCHEDULER: STOPPED: Process [name]: prog  [id]: 1
*** SCHEDULER: Next process to continue: Process [name]: prog  [id]: 2

./prog[1783]: This is message 9
./prog[1783]: This is message 10
./prog[1783]: This is message 11
./prog[1783]: This is message 12

*** SCHEDULER: Going to stop process [id]: 2
*** SCHEDULER: STOPPED: Process [name]: prog  [id]: 2
*** SCHEDULER: Next process to continue: Process [name]: prog  [id]: 3

./prog[1784]: This is message 10
./prog[1784]: This is message 11
./prog[1784]: This is message 12
./prog[1784]: This is message 13
```

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.*

   Τα σήματα SIGALRM και SIGCHLD αθροίζονται με το sigset και αγνοούνται διότι μπλοκάρονται κατά τη διάρκεια εκτέλεσης των handlers.
   Υπό κανονικές συνθήκες, ένας πραγματικός χρονοδρομολογητής πυρήνα λειτουργεί με hardware interrupts και θα ακολουθούσε την εξής διαδικασία πέραν της αγνόησης του σήματος: θα το αποθήκευε σε κάποια δομή (στοίβα) ώστε να το επεξεργαστεί όταν ολοκληρώσει με τον χειρισμό του τρέχοντος σήματος.

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;*

   - Όταν ο scheduler λαμβάνει σήμα SIGCHLD είναι αναμενόμενο να αναφέρεται στο παιδί που βρίσκεται στην κορυφή της ουράς διεργασιών.
     Στην περίπτωση που κάτι αναπάντεχο συμβεί, π.χ. ένα SIGKILL τερματίσει μια διεργασία-παιδί, ο χρονοδρομολογητής λαμβάνει σήμα SIGCHLD και εκτελείται το κομμάτι κώδικα από τον sigchld_handler.

```
else {
                /* Handle the case that a different than the head process
                 * has changed status
                 */

                process *pr = erase_proc_by_pid(p_list, pid);
                if (pr != NULL ) {
                    printf ("*** SCHEDULER: A process other than the head h
as Changed state unexpectedely: Process [name]: %s  [id]: %d\n",
                    pr->name, pr->id);
                    free_process(pr);
                }
```

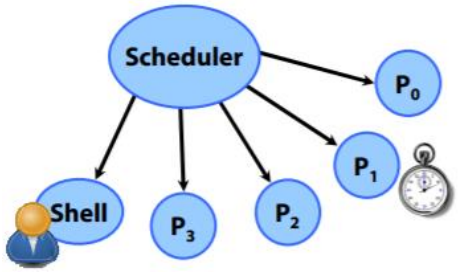
     Η διαδικασία που ακολουθείται είναι η εξής: διαγραφή μέσω αναζήτησης του pid της την διεργασία που τερματίστηκε απρόοπτα, απελευθέρωση του χώρου που καταλάμβανε ενώ η λειτουργία του χρονοδρομολογητή συνεχίζεται κανονικά, με την διεργασία στο head της ουράς να εκτελείται.
   - Εάν λόγω εξωτερικού παράγοντα τερματιστεί αναπάντεχα μια οποιαδήποτε διεργασία-παιδί, υπάρχει περίπτωση η τρέχουσα διεργασία να μην έχει λάβει το σήμα SIGSTOP και να συνεχίσει κανονικά τη λειτουργία της όταν θα προχωρήσει το πρόγραμμα στην εκτέλεση της επόμενης στην ουρά, γεγονός που θα οδηγούσε σε απρόβλεπτη και εσφαλμένη συμπεριφορά.

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.*

   Το SIGCHLD στέλνεται όταν ένα παιδί αλλάξει κατάσταση και με αυτό τον τρόπο εξασφαλίζεται ότι έχει ληφθεί από το παιδί το σήμα SIGSTOP, το οποίο προκαλεί άμεση αναστολή της εκτέλεσης της τρέχουσας διεργασίας. Σε αντίθετη περίπτωση υπάρχει δυνατότητα να τρέχουν ταυτόχρονα δύο διεργασίες και έτσι δεν επαρκεί η χρήση του SIGALRM που ευθύνεται απλά για τη λήξη προκαθορισμένου χρονικού διαστήματος.

1.2 Έλεγχος λειτουργίας χρονοδρομολογητή μέσω φλοιού

Ζητείται η επέκταση του χρονοδρομολογητή του προηγούμενου ερωτήματος, ώστε να υποστηρίζεται ο έλεγχος της λειτουργίας του μέσω προγράμματος-φλοιού. Ο χρήστης του συστήματος έχει τη δυνατότητα να ζητά δυναμική δημιουργία και τερματισμό διεργασιών, αλληλεπιδρώντας με το πρόγραμμα του φλοιού. Ο φλοιός δέχεται εντολές από το χρήστη, κατασκευάζει αιτήσεις κατάλληλης μορφής τις οποίες αποστέλλει προς τον χρονοδρομολογητή, λαμβάνει απαντήσεις που ενημερώνουν για την έκβαση της εκτέλεσής τους (επιτυχία / αποτυχία) και ενημερώνει για το αποτέλεσμά τους το χρήστη.



Ο φλοιός διαθέτει τέσσερις εντολές:

- Εντολή 'p': Ο χρονοδρομολογητής εκτυπώνει στην έξοδο λίστα με τις υπό εκτέλεση διεργασίες, στον οποίο φαίνεται ο σειριακός αριθμός id της διεργασίας, το PID και το όνομα της. Επιπλέον, επισημαίνεται η τρέχουσα διεργασία.
- Εντολή 'k': Δέχεται όρισμα το id μιας διεργασίας (προσοχή: όχι το PID) και ζητά από το χρονοδρομολογητή τον τερματισμό της.
- Εντολή 'e': Δέχεται όρισμα το όνομα ενός εκτελέσιμου στον τρέχοντα κατάλογο, π.χ. prog2 και ζητά τη δημιουργία μιας νέας διεργασίας από τον χρονοδρομολογητή, στην οποία θα τρέχει αυτό το εκτελέσιμο.
- Εντολή 'q': Ο φλοιός τερματίζει τη λειτουργία του.

Τελικό παραδοτέο είναι νέα έκδοση του χρονοδρομολογητή, η οποία θα εξυπηρετεί τις αιτήσεις του φλοιού. Ο φλοιός ζητείται να χρονοδρομολογείται μαζί με τις υπόλοιπες διεργασίες, όντας στην ουρά εκτέλεσης και λαμβάνοντας κβάντα χρόνου σύμφωνα με τον αλγόριθμο RR. Ο χρονοδρομολογητής θα τερματίζεται όταν όλες οι διεργασίες που χειρίζεται τερματίσουν, συμπεριλαμβανομένου του φλοιού.

Σας δίνεται το πρόγραμμα του φλοιού, shell.c και το αρχείο request.h που ορίζει τη μορφή των αιτήσεων από το φλοιό προς τον χρονοδρομολογητή με βάση τη δομή struct request_struct. Για την υποστήριξη του φλοιού είναι απαραίτητος ένας μηχανισμός επικοινωνίας από το φλοιό προς τον χρονοδρομολογητή και αντίστροφα. Ο μηχανισμός αυτός δίνεται στο αρχείο scheduler-shell.c, που αποτελεί σκελετό της ζητούμενης υλοποίησης.

To shell.c :

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

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

#define SHELL_CMDLINE_SZ 100

void issue_request(int wfd, int rfd, struct request_struct *rq)
{
    int ret;

    /* Issue the request */
    fprintf(stderr, "Shell: issuing request...\n");
    if (write(wfd, rq, sizeof(*rq)) != sizeof(*rq)) {
        perror("Shell: write request struct");
        exit(1);
    }

    /* Block until a reply has been received */
    fprintf(stderr, "Shell: receiving request return value...\n");

    if (read(rfd, &ret, sizeof(ret)) != sizeof(ret)) {
        perror("Shell: read request return value");
        exit(1);
    }

    if(ret < 0){
        fprintf(stderr, "Shell: request return value ret = %d\n", ret);
        fprintf(stderr, "       %s\n", strerror(-ret));
    }
}

/*
 * Read a command line from the stream pointed to by fp
 * [a standard C library stream, *not* a file descriptor],
```

```
 * strip any trailing newlines.
 */
void get_cmdline(FILE *fp, char *buf, int bufsz)
{
    if (fgets(buf, bufsz, fp) == NULL) {
        fprintf(stderr, "Shell: could not read command line, exiting.\n");
        exit(1);
    }
    if (buf[strlen(buf) - 1] == '\n')
        buf[strlen(buf) - 1] = '\0';
}

/* print help */
void help(void)
{
    printf(" ?          : print help\n"
           " q          : quit\n"
           " p          : print tasks\n"
           " k <id>     : kill task identified by id\n"
           " e <program>: execute program\n"
           " h <id>     : set task identified by id to high priority\n"
           " l <id>     : set task identified by id to low priority\n");
}

/*
 * Parse a command line, construct and
 * issue the relevant request to the scheduler.
 *
 * Parsing is very simple, a better way would be to
 * break up the command line in tokens.
 */
void process_cmdline(char *cmdline, int wfd, int rfd)
{
    struct request_struct rq;

    if (strlen(cmdline) == 0 || strcmp(cmdline, "?") == 0){
        help();
        return;
    }

    /* Quit */
    if (strcmp(cmdline, "q") == 0 || strcmp(cmdline, "Q") == 0) {
        fprintf(stderr, "Shell: Exiting. Goodbye.\n");
        exit(0);
    }

    /* Print Tasks */
    if (strcmp(cmdline, "p") == 0 || strcmp(cmdline, "P") == 0) {
```

```
        rq.request_no = REQ_PRINT_TASKS;
        issue_request(wfd, rfd, &rq);
        return;
    }

    /* Kill Task */
    if ((cmdline[0] == 'k' || cmdline[0] == 'K') &&
        cmdline[1] == ' ') {
        rq.request_no = REQ_KILL_TASK;
        rq.task_arg = atoi(&cmdline[2]);
        issue_request(wfd, rfd, &rq);
        return;
    }

    /* Exec Task */
    if ((cmdline[0] == 'e' || cmdline[0] == 'E') && cmdline[1] == ' ') {
        rq.request_no = REQ_EXEC_TASK;
        strncpy(rq.exec_task_arg, &cmdline[2], EXEC_TASK_NAME_SZ);
        rq.exec_task_arg[EXEC_TASK_NAME_SZ - 1] = '\0';
        issue_request(wfd, rfd, &rq);
        return;
    }

    /* High-prioritize task */
    if ((cmdline[0] == 'h' || cmdline[0] == 'H') && cmdline[1] == ' ') {
        rq.request_no = REQ_HIGH_TASK;
        rq.task_arg = atoi(&cmdline[2]);
        issue_request(wfd, rfd, &rq);
        return;
    }

    /* Low-prioritize task */
    if ((cmdline[0] == 'l' || cmdline[0] == 'L') && cmdline[1] == ' ') {
        rq.request_no = REQ_LOW_TASK;
        rq.task_arg = atoi(&cmdline[2]);
        issue_request(wfd, rfd, &rq);
        return;
    }

    /* Parse error, malformed command, whatever... */
    printf("command `%s': Bad Command.\n", cmdline);
}

int main(int argc, char *argv[])
{
    int rfd, wfd;
    char cmdline[SHELL_CMDLINE_SZ];
```

```
    /*
     * Communication with the scheduler happens over two UNIX pipes.
     *
     * The scheduler first creates the pipes, then execve()s the shell
     * program. It passes two file descriptors as command-line arguments:
     *
     * argument 1: wfd: the file descriptor to write request structures into.
     * argument 2: rfd: the file descriptor to read request return values from.
     */

    if (argc != 3) {
        fprintf(stderr, "Shell: must be called with exactly two arguments.\n");
        exit(1);
    }

    wfd = atoi(argv[1]);
    rfd = atoi(argv[2]);
    if (!wfd || !rfd) {
        fprintf(stderr, "Shell: descriptors must be non-zero: wfd = %d, rfd = %d\n",
            wfd, rfd);
        exit(1);
    }

    /*
     * Forever: show the prompt, read a command line, then process it.
     */
    printf("\nThis is the Shell. Welcome.\n\n");

    for (;;) {
        printf("Shell> ");
        fflush(stdout);
        get_cmdline(stdin, cmdline, SHELL_CMDLINE_SZ);
        process_cmdline(cmdline, wfd, rfd);
    }

    /* Unreachable */
    fprintf(stderr, "Shell: reached unreachable point.\n");
    return 1;
}
```

To scheduler-shell.c :

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

```c
#include <assert.h>

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

#include "proc-common.h"
#include "request.h"
#include "helper.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 */

process* current_p;
process_list* l;
/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
    printf("\n***THE LIST***");
    print_list(l, current_p);
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id)
{
    printf("\n\nATTEMPTING TO KILL THE PROCESS: %d\n", id);
    process* p = get_proc_by_id(l, id);

    if (p == NULL) {
        printf("Process not exists ins scheduler list\n");
        return 1;
    }

    printf("Process found is scheduler's list, executing SIGKILL\n");
    kill(p->pid, SIGKILL);
    return 0;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
```

```
    green();
    printf("\n\nATTEMPING TO CREATE THE PROCESS FOR: %s\n", executable);
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }
    if (pid == 0) {
        raise(SIGSTOP);
        char filepath[TASK_NAME_SZ];
        sprintf(filepath, "./%s", executable);
        // TOD
        char* args[] = {filepath, NULL};
        if (execvp(filepath, args)) {
            perror("execvp");
            exit(1);
        }
    }
    waitpid(pid, NULL, WUNTRACED);
    process *p = process_create(pid, executable);

    // Push process in low list
    push(l, p);
    printf("SCHEDULER: Process [name]: %s [id]: %d was succesfully created. Added in
 LOW.\n",
        executable, p->id);
    reset();
}


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

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

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

        default:
            return -ENOSYS;
    }
```

```
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{

    red();
    printf("\n*** SCHEDULER: Going to stop process [id]: %d\n",
            current_p->id);
    reset();
    kill(current_p->pid, SIGSTOP);

}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
    printf("Signum: %d, pid: %ld\n", signum, (long)getpid());
    bool pass_to_next = false;
    int status;
    pid_t pid;
    for (;;) {
        pid = waitpid(-1, &status, WUNTRACED | WNOHANG);
        if (pid == 0) {
            break;
        }
        if (pid < 0) {
            perror("waitpid");
            exit(1);
        }
        if (pid != 0) {

            // Check if head process changed status
            process *p;
            red();

            // Process has stopped
            if (WIFSTOPPED(status)) {
                if (pid == (current_p->pid)) {
                    red();
                    printf ("*** SCHEDULER: STOPPED: Current Process [name]: %s  [id
]: %d\n",
```

```
                        current_p->name, current_p->id);
                    reset();
                    p = get_next(l);
                    pass_to_next = true;
                } else {
                    process* affected = get_proc_by_pid(l, pid);
                    if (affected != NULL) {
                        red();
                        printf ("*** SCHEDULER: STOPPED: NOT current Process [name]:
 %s  [id]: %d\n",
                            affected->name, affected->id);
                        reset();
                    } else {
                        perror("\nTHIS SHOULD !NOT HAPPEN!\n");
                    }
                }

            // Process has exited
            } else if (WIFEXITED(status)) {
                if (pid == (current_p->pid)) {
                    printf ("*** SCHEDULER: EXITED: Current Process [name]: %s  [id]
: %d\n",
                    current_p->name, current_p->id);
                    erase_proc_by_id(l, current_p->id);
                    free_process(current_p);

                    if (empty(l)) {
                        printf ("*** SCHEDULER: No more processes to schedule. Clean
ing and exiting...\n");
                        exit(0);
                    }
                    p = l->head;
                    pass_to_next = true;

                } else {
                    process* affected = get_proc_by_pid(l, pid);
                    if (affected != NULL) {
                        printf ("*** SCHEDULER: EXITED: NOT Current Process [name]:
%s  [id]: %d\n",
                                affected->name, affected->id);

                        affected = erase_proc_by_pid(l, pid);
                        free_process(affected);
                    } else {
                        perror("\n\nTHIS SHOULD NOT HAPPEN!\n\n\n");
                        exit(11);
                    }
                }
```

```
            } else if (WIFSIGNALED(status)) {
                if (pid == (current_p->pid)) {
                    printf ("*** SCHEDULER: GOT KILLED: Current Process [name]: %s
[id]: %d\n",
                    current_p->name, current_p->id);
                    p = pop(l);
                    free_process(p);

                    if (empty(l)) {
                        printf ("*** SCHEDULER: No more processes to schedule. Clean
ing and exiting...\n");
                        exit(0);
                    }

                    p = l->head;
                    pass_to_next = true;
                } else {
                    process* affected = get_proc_by_pid(l, pid);
                    if (affected != NULL) {
                        printf ("*** SCHEDULER: GOT KILLED: NOT Current Process [nam
e]: %s  [id]: %d\n",
                                affected->name, affected->id);

                        affected = erase_proc_by_pid(l, pid);
                        free_process(affected);
                    } else {
                        printf("\n\nTHIS SHOULD NOT HAPPEN WHEN SIGNALED\n\n\n");
                        exit(11);
                    }
                }
                reset();
            }
            else {
                red();
                printf("Something really strange happened!\n");
                reset();
                exit(100);
            }
            reset();

            if (pass_to_next) {
                printf ("*** SCHEDULER: Next process to continue: [name]: %s  [id]:
%d\n\n",
                    p->name, p->id);
                current_p = p;
                kill(p->pid, SIGCONT);
                alarm(SCHED_TQ_SEC);
            }
```

```
        }
    }
}

/* 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_flags = SA_RESTART;

    // Specify signals to be blocked while the handling funvtion runs
    sigemptyset(&sigset);
```

```
    sigaddset(&sigset, SIGCHLD);
    sigaddset(&sigset, SIGALRM);
    sa.sa_mask = sigset;

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

    // TODO In exercise the sa handler was reassigned, does it work?
    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);
}
```

```c
/* 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 */
    process *proc = process_create(p, executable);
    push(1, proc);
    green();
    printf("Created process: SHELL: %s with pid: %ld\n",
            executable, (long)p);
    reset();

    waitpid(p, NULL, WUNTRACED);

    //wait_for_ready_children(1);

    close(pfds_rq[1]);
    close(pfds_ret[0]);

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

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

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

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

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

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

    /* Create the shell. */
    /* TODO: add the shell to the scheduler's tasks */
    sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);

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

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

    int i;
```

```
    for (i = 1; i < argc; i++) {
        pid_t pid;
        pid = fork();
        if (pid < 0) {
            perror("fork");
            exit(1);
        }
        if (pid == 0) {
            raise(SIGSTOP);
            char filepath[TASK_NAME_SZ];
            sprintf(filepath, "./%s", argv[i]);
            // TODO
            char* args[] = {filepath, NULL};
            if (execvp(filepath, args)) {
                perror("execvp");
                exit(1);
            }
        }
        process *p = process_create(pid, argv[i]);
        push(l, p);
        green();
        printf("Process name: %s id: %d is created.\n",
            argv[i], p->id);
        reset();
    }
    /* Wait for all children to raise SIGSTOP before exec()ing. */
    wait_for_ready_children(nproc);

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

    if (nproc == 0) {
        fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
        exit(1);
    }
    printf("Scheduler dispatching the first process...\n");
    process* head = l->head;
    current_p = head;
    kill(head->pid, SIGCONT);
    alarm(SCHED_TQ_SEC);

    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;
}
```

Το Makefile παραμένει ίδιο.

Το output :

```
oslaba36@os-node1:~/exe4$ vi scheduler-shell.c
oslaba36@os-node1:~/exe4$ ./scheduler prog prog
Process name: prog id: 0 is created.
Process name: prog id: 1 is created.
My PID = 2203: Child PID = 2204 has been stopped by a signal, signo = 19
My PID = 2203: Child PID = 2205 has been stopped by a signal, signo = 19
Scheduler dispatching the first process...
./prog: Starting, NMSG = 200, delay = 75
./prog[2204]: This is message 0
./prog[2204]: This is message 1
./prog[2204]: This is message 2
./prog[2204]: This is message 3
./prog[2204]: This is message 4
./prog[2204]: This is message 5
```

```
./prog[2204]: This is message 9
./prog[2204]: This is message 10
./prog[2204]: This is message 11
./prog[2204]: This is message 12
./prog[2204]: This is message 13
./prog[2204]: This is message 14
./prog[2204]: This is message 15
./prog[2204]: This is message 16
./prog[2204]: This is message 17

*** SCHEDULER: Going to stop process [id]: 0
*** SCHEDULER: STOPPED: Process [name]: prog  [id]: 0
*** SCHEDULER: Next process to continue: Process [name]: prog  [id]: 1

./prog[2205]: This is message 12
./prog[2205]: This is message 13
./prog[2205]: This is message 14
./prog[2205]: This is message 15
./prog[2205]: This is message 16
./prog[2205]: This is message 17
./prog[2205]: This is message 18
./prog[2205]: This is message 19
./prog[2205]: This is message 20
./prog[2205]: This is message 21
./prog[2205]: This is message 22
./prog[2205]: This is message 23

*** SCHEDULER: Going to stop process [id]: 1
*** SCHEDULER: STOPPED: Process [name]: prog  [id]: 1
*** SCHEDULER: Next process to continue: Process [name]: prog  [id]: 0

./prog[2204]: This is message 18
./prog[2204]: This is message 19
./prog[2204]: This is message 20
./prog[2204]: This is message 21
./prog[2204]: This is message 22
./prog[2204]: This is message 23
./prog[2204]: This is message 24
./prog[2204]: This is message 25
./prog[2204]: This is message 26
```

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;*

   Όταν και ο φλοιός υφίσταται χρονοδρομολόγηση, εμφανίζεται ως τρέχουσα διεργασία στη λίστα διεργασιών που τυπώνεται με την εντολή 'p' του φλοιού.
   Στην υλοποίηση που έγινε, δεν είναι δυνατό να συμβεί διαφορετικά επειδή η εμφάνιση της λίστας διεργασιών είναι ένα request που εκτελεί ο φλοιός και άρα τη στιγμή εκτύπωσης της λίστας διεργασιών, η τρέχουσα διεργασία 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.*

   Όπως αναφέρεται, η συνάρτηση υλοποίησης αιτήσεων του φλοιού μεταβάλλει δομές όπως η ουρά εκτέλεσης των διεργασιών. Επομένως, όσο αυτό συμβαίνει, πρέπει να αποτρέπεται η παράλληλη μεταβολή τους από κάποια άλλη συνάρτηση-handler η οποία μπορεί να κλήθηκε λόγω signals. Εάν αυτά τα σήματα δεν απενεργοποιηθούν, θα επιτρεπόταν η παρεμβολή κάποιου άλλου μέρους του προγράμματος την ώρα της εκτέλεσης των αιτήσεων του φλοιού με κίνδυνο να προκύψει ένα rare condition, όπως π.χ. στις λίστες της ουράς εκτέλεσης και θα οδηγούσε πιθανώς σε undefined behavior.

1.3 Υλοποίηση προτεραιοτήτων στον χρονοδρομολογητή

Ζητείται η επέκταση του χρονοδρομολογητή του προηγούμενου ερωτήματος, ώστε να υποστηρίζονται δύο κλάσεις προτεραιότητας: HIGH και LOW. Ο αλγόριθμος χρονοδρομολόγησης αλλάζει ως εξής: αν υπάρχουν διεργασίες προτεραιότητας HIGH εκτελούνται μόνο αυτές χρησιμοποιώντας κυκλική επαναφορά (round-robin). Σε αντίθετη περίπτωση, χρονοδρομολογούνται οι LOW διεργασίες χρησιμοποιώντας κυκλική επαναφορά. Όλες οι διεργασίες δημιουργούνται με LOW προτεραιότητα. Η αλλαγή της προτεραιότητας μιας διεργασίες θα πραγματοποιείται με τις παρακάτω εντολές φλοιού:

- 'h' ('l') Δέχεται όρισμα το id μιας διεργασίας και θέτει την προτεραιότητα της σε HIGH (LOW)

To scheduler-priority.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 <stdbool.h>
```

```c
#include "proc-common.h"
#include "request.h"
#include "helper.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 */

process* current_p;
process_list* l_list;
process_list* h_list;
/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
    printf("\n***LOW LIST***");
    print_list(l_list, current_p);
    printf("\n***HIGH LIST***");
    print_list(h_list, current_p);
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id)
{
    printf("\n\nATTEMPTING TO KILL THE PROCESS: %d\n", id);
    process* p = get_proc_by_id_list(l_list, h_list, id);

    if (p == NULL) {
        printf("Process not exists ins scheduler list\n");
        printf("END OF MESSAGE\n\n");
        return 1;
    }

    printf("Process found is scheduler's list, executing SIGKILL\n");
    kill(p->pid, SIGKILL);
    printf("END OF MESSAGE\n\n");
    return 0;
}


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

```
    printf("\n\nATTEMPING TO CREATE THE PROCESS FOR: %s\n", executable);
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }
    if (pid == 0) {
        raise(SIGSTOP);
        char filepath[TASK_NAME_SZ];
        sprintf(filepath, "./%s", executable);
        // TOD
        char* args[] = {filepath, NULL};
        if (execvp(filepath, args)) {
            perror("execvp");
            exit(1);
        }
    }
    waitpid(pid, NULL, WUNTRACED);
    process *p = process_create(pid, executable);

    // Push process in low list
    push(l_list, p);
    printf("SCHEDULER: Process [name]: %s [id]: %d was succesfully created. Added in
 LOW.\n",
        executable, p->id);
    reset();
}

int sched_move_to_high(int id) {
    int status =  move_from_to(l_list, h_list, id);
    if (status) {
        printf("\n\nSUCESSFULLY MOVED [pid] : %d TO HIGH", id);
    } else {
        printf("\n\nFAILED MOVING [pid] : %d TO HIGH", id);
    }
    return status;

}

int sched_move_to_low(int id) {
    int status =  move_from_to(h_list, l_list, id);
    if (status) {
        printf("\nSUCESSFULLY MOVED [pid] : %d TO LOW", id);
    } else {
        printf("\nFAILED MOVING [pid] : %d TO LOW", id);
    }
    return status;
}
```

```c
/* 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_move_to_high(rq->task_arg);

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


        default:
            return -ENOSYS;
    }
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{

    red();
    printf("\n*** SCHEDULER: Going to stop process [id]: %d\n",
            current_p->id);
    reset();
    kill(current_p->pid, SIGSTOP);

}

/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
```

```
{
    bool pass_to_next = false;
    int status;
    pid_t pid;
    for (;;) {
        pid = waitpid(-1, &status, WUNTRACED | WNOHANG);
        if (pid == 0) {
            break;
        }
        if (pid < 0) {
            perror("waitpid");
            exit(1);
        }
        if (pid > 0) {

            // Check if head process changed status
            process *p;
            red();

            // Process has stopped
            if (WIFSTOPPED(status)) {
                if (pid == (current_p->pid)) {
                    red();
                    printf ("*** SCHEDULER: STOPPED: Current Process [name]: %s  [id
]: %d\n",
                        current_p->name, current_p->id);
                    reset();
                    p = get_next_lists(l_list, h_list);
                    pass_to_next = true;
                } else {
                    process* affected = get_proc_by_pid_list(l_list, h_list, pid);
                    if (affected != NULL) {
                        red();
                        printf ("*** SCHEDULER: STOPPED: NOT current Process [name]:
 %s  [id]: %d\n",
                            affected->name, affected->id);
                        reset();
                    } else {
                        perror("\nTHIS SHOULD !NOT HAPPEN!\n");
                    }
                }

            // Process has exited
            } else if (WIFEXITED(status)) {
                if (pid == (current_p->pid)) {
                    printf ("*** SCHEDULER: EXITED: Current Process [name]: %s  [id]
: %d\n",
                    current_p->name, current_p->id);
```

```
                    erase_proc_by_id_list(l_list, h_list, current_p->id);
                    free_process(current_p);

                    if (empty_lists(l_list, h_list)) {
                        printf ("*** SCHEDULER: No more processes to schedule. Clean
ing and exiting...\n");
                        clear(l_list);
                        clear(h_list);
                        exit(0);
                    }
                    p = get_head_of_lists(l_list, h_list);
                    pass_to_next = true;

                } else {
                    process* affected = get_proc_by_pid_list(l_list, h_list, pid);
                    if (affected != NULL) {
                        printf ("*** SCHEDULER: EXITED: NOT Current Process [name]:
%s  [id]: %d\n",
                                affected->name, affected->id);

                        affected = erase_proc_by_pid_list(l_list, h_list, pid);
                        free_process(affected);
                    } else {
                        perror("\n\nTHIS SHOULD NOT HAPPEN!\n\n\n");
                        exit(11);
                    }
                }
            } else if (WIFSIGNALED(status)) {
                if (pid == (current_p->pid)) {
                    printf ("*** SCHEDULER: GOT KILLED: Current Process [name]: %s
[id]: %d\n",
                    current_p->name, current_p->id);
                    p = pop_list(l_list, h_list);
                    free_process(p);

                    if (empty_lists(l_list, h_list)) {
                        printf ("*** SCHEDULER: No more processes to schedule. Clean
ing and exiting...\n");
                        clear(l_list);
                        clear(h_list);
                        exit(0);
                    }

                    p = get_head_of_lists(l_list, h_list);
                    pass_to_next = true;
                } else {
                    process* affected = get_proc_by_pid_list(l_list, h_list, pid);
                    if (affected != NULL) {
```

```
                        printf ("*** SCHEDULER: GOT KILLED: NOT Current Process [nam
e]: %s  [id]: %d\n",
                                affected->name, affected->id);

                        affected = erase_proc_by_pid_list(l_list, h_list, pid);
                        free_process(affected);
                    } else {
                        printf("\n\nTHIS SHOULD NOT HAPPEN WHEN SIGNALED\n\n\n");
                        exit(11);
                    }
                }
                reset();
            }
            else {
                red();
                printf("Something really strange happened!\n");
                reset();
                exit(100);
            }
            reset();

            if (pass_to_next) {
                printf ("*** SCHEDULER: Next process to continue: [name]: %s  [id]:
%d\n\n",
                    p->name, p->id);
                current_p = p;
                kill(p->pid, SIGCONT);
                alarm(SCHED_TQ_SEC);
            }
        }
    }
}

/* 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);
    }
}
```

```c
/* 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_flags = SA_RESTART;

    // Specify signals to be blocked while the handling funvtion runs
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGCHLD);
    sigaddset(&sigset, SIGALRM);
    sa.sa_mask = sigset;

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

    // TODO In exercise the sa handler was reassigned, does it work?
    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 */
    process *proc = process_create(p, executable);
    push(l_list, proc);
    green();
    printf("Created process: SHELL: %s with pid: %ld\n",
            executable, (long)p);
    reset();

    waitpid(p, NULL, WUNTRACED);

    //wait_for_ready_children(1);

    close(pfds_rq[1]);
    close(pfds_ret[0]);

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

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

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

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

```
        signals_enable();

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

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

    /* Create the shell. */
    /* TODO: add the shell to the scheduler's tasks */
    sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);

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

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

    int i;
    for (i = 1; i < argc; i++) {
        pid_t pid;
        pid = fork();
        if (pid < 0) {
            perror("fork");
            exit(1);
        }
        if (pid == 0) {
            raise(SIGSTOP);
            char filepath[TASK_NAME_SZ];
            sprintf(filepath, "./%s", argv[i]);
            // TODO
            char* args[] = {filepath, NULL};
            if (execvp(filepath, args)) {
                perror("execvp");
                exit(1);
            }
        }
```

```
        process *p = process_create(pid, argv[i]);
        push(l_list, p);
        green();
        printf("Process name: %s id: %d is created.\n",
            argv[i], p->id);
        reset();
    }


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

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

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


    printf("Scheduler dispatching the first process...\n");
    process* head = get_head_of_lists(l_list, h_list);
    current_p = head;
    kill(head->pid, SIGCONT);
    alarm(SCHED_TQ_SEC);

    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;
}
```

Το Makefile παραμένει το ίδιο.

Ερωτήσεις:

1. *Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας*

   Ένα σενάριο δημιουργίας λιμοκτονίας αποτελεί η περίπτωση που τουλάχιστον μια από τις διεργασίες οι οποίες είναι στην ουρά υψηλής προτεραιότητας και δεν τερματίζονταν ποτέ. Σε αυτό το σενάριο, οι διεργασίες στην ουρά χαμηλής προτεραιότητας θα βρίσκονταν σε αναμονή και δεν θα μπορούσαν να εκτελεστούν, αφού η ουρά υψηλής προτεραιότητας θα ήταν μη κενή.

Τέλος παρατίθενται τα δοσμένα αρχεία.

Το sigchld-example.c :

```c
/*
 * sigchld-test.c
 *
 * A program to demonstrate use of SIGCHLD
 * by a parent process, so it may be notified of
 * state changes in children processes.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <signal.h>
#include <sys/wait.h>
#include "proc-common.h"

#define SLEEP_SEC 1
#define ALARM_SEC 2

void child(void)
{
    pid_t pid = getpid();

    for (;;) {
        printf("I am child %ld, sleeping for %d sec...\n",
            (long)pid, SLEEP_SEC);
        sleep(SLEEP_SEC);
    }
}

/*
 * A handler for SIGALRM in the parent
 */
void sigalrm_handler(int signum)
```

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

    printf("ALARM! %d seconds have passed.\n", ALARM_SEC);

    /* Setup the alarm again */
    if (alarm(ALARM_SEC) < 0) {
        perror("alarm");
        exit(1);
    }
}

/*
 * A handler for SIGCHLD in the parent
 */
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)) {
            /* A child has died */
            printf("Parent: Received SIGCHLD, child is dead. Exiting.\n");
            exit(0);
        }
        if (WIFSTOPPED(status)) {
            /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
            printf("Parent: Child has been stopped. Moving right along...\n");
        }
    }
}

int main(void)
{
    pid_t p;

    /* Install SIGCHLD handler */
    if (signal(SIGCHLD, sigchld_handler) < 0) {
        perror("signal");
        exit(1);
    }

    /* Install SIGALRM handler */
    if (signal(SIGALRM, sigalrm_handler) < 0) {
        perror("signal");
        exit(1);
    }

    /* Arrange for an alarm after 1 sec */
    if (alarm(ALARM_SEC) < 0) {
        perror("alarm");
        exit(1);
    }

    printf("Parent: Creating child...\n");
    p = fork();
    if (p < 0) {
        /* fork failed */
        perror("fork");
        exit(1);
    }
    if (p == 0) {
        /* In child process */
        child();
        /*
```

```
         * Should never reach this point,
         * child() does not return
         */
        assert(0);
    }

    /*
     * In parent process.
     */

    /*
     * Do nothing until the child terminates.
     * The handler will exit().
     */
    printf("Parent: Created child with PID = %ld, waiting for it to terminate...\n",
        (long)p);
    while (pause())
        ;

    return 0;
}
```

Και το strace-test.c :

```
/*
 * strace-test.c
 *
 * Make sure everything works OK when strace'ing
 * a program using fork() and signals.
 *
 * Usage:
 *
 * strace -f ./strace-test 2>/dev/null
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <signal.h>
#include <sys/wait.h>

#define STOP_SIGNAL SIGTSTP

int main(int argc, char *argv[])
{
```

```
    int status;
    pid_t p, waitp;

    p = fork();
    if (p < 0) {
        perror("fork");
        exit(1);
    }

    if (p == 0) {
        raise(STOP_SIGNAL);
        /* Should never reach this point */
        printf("Child: FAIL.\n");
        exit(1);
    }

    /*
     * Father sleeps for a while and kills the child.
     * The child should be alive when the father wakes.
     */
    printf("Parent: sleeping for a while...\n");
    sleep(2);

    if (kill(p, SIGKILL) < 0) {
        perror("kill");
        exit(1);
    }

    waitp = wait(&status);
    assert(p == waitp);
    if (WIFEXITED(status)) {
        printf("Parent: FAIL. Child exited, status = %d\n",
            WEXITSTATUS(status));
        exit(1);
    }

    if (!WIFSIGNALED(status)) {
        printf("Parent: FAIL. Child not signaled?\n");
        exit(1);
    }

    printf("Parent: SUCCESS. Child killed, sig = %d\n",
        WTERMSIG(status));
    return 0;
}
```