ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά 4ης Εργαστηρικής Άσκησης

Ιάσων - Λάζαρος Παπαγεωργίου | Α.Μ: 03114034
Γρηγόρης Θανάσουλας | Α.Μ: 03114131

Άσκηση 1

```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 */
#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 Changed 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;
}
```

Ερωτήσεις

1. **Τι συμβαίνει αν το σήμα SIGALRM έρθει ενώ εκτελείται η συνάρτηση χειρισμού τουσήματος SIGCHLD ή το αντίστροφο; Πώς αντιμετωπίζει ένας πραγματικός χρονοδρομολογητής χώρου πυρήνα ανάλογα ενδεχόμενα και πώς η δική σας υλοποίηση;**

   Τα σήματα SIGALRM και SIGCHLD προστίθενται στο sigset και μπλοκάρονται κατά τη διάρκεια εκτέλεσης των handlers, συνεπώς θα αγνοηθούν.

   Ένας πραγματικός χρονοδρομολογητής πυρήνα δε θα αγνοούσε απλώς το σήμα, αλλά θα τα κρατούσε σε κάποια δομή (στοίβα) προκειμένου να το χειριστεί αφότου έχει τελειώσει με τον χειρισμό του τρέχοντος σήματος. Πρέπει να σημειώσουμε ότι στην πραγματικότητα, ένας χρονδορομολογητής λειτουργεί με hardware interrupts.

2. **Κάθε φορά που ο χρονοδρομολογητής λαμβάνει σήμα SIGCHLD, σε ποια διεργασία- παιδί περιμένετε να αναφέρεται αυτό;**

   Κάθε φορά που ο scheduler λαμβάνει σήμα SIGCHLD περιμένουμε να αναφέρεται στο παιδί που βρίσκεται στην κορυφή της ουράς διεργασιών. Έαν για κάποιο λόγο, παραδείγματος χάριν ένα SIGKILL τερματιστεί αναπάντεχα μία διεργασία παιδί, ο χρονομοδρολογητής λαμβάνει σήμα SIGCHLD και εκτελείται το παρακάτω κομμάτι κώδικα από τον sigchld_handler:

   ```
   else {
   ```

```
      /* Handle the case that a different than the head process
      * has changed status
      */
        printf("A process other than the head has changed
status.\n");
       process *pr = erase_proc_by_pid(p_list, pid);

       free_process(pr);
}
```

Διαγράφουμε λοιπόν μέσω αναζήτησης του pid της την διεργασία που τερματίστηκε απρόοπτα, ελευθερώνουμε τον χώρο που καταλάμβανε και η λειτουργία του χρονοδρομολογητή συνεχίζεται κανονικά, με την διεργασία στο head της ουράς να εκτελείται.

3. **Τι συμβαίνει αν λόγω εξωτερικού παράγοντα (π.χ. Αποστολή SIGKILL) τερματιστεί αναπάντεχα μια οποιαδήποτε διεργασία- παιδί;**

   Θα υπήρχε περίπτωση η τρέχουσα διεργασία να μην έχει λάβει το σήμα SIGSTOP και να συνεχίζει κανονικά τη λειτουργία της όταν θα πάμε να προχωρήσουμε στην εκτέλεση της επόμενης στην ουρά, κάτι που θα έχει ως αποτέλεσμα λανθασμένη και απρόβλεπτη συμπεριφορά.

Άσκηση 2

```
#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>

#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. Cleaning 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. Cleaning 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 [name]: %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.  */
```

```c
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, 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 = 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;
}
```

Ερωτήσεις

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

   Παρόλο που και ο φλοιός υφίσταται χρονοδρομολόγηση, εμφανίζεται πάντα ως τρέχουσα διεργασία στη λίστα διεργασιών που τυπώνεται με την εντολή 'p' του φλοιού. Στην υλοποίηση μας, δε θα μπορούσε να συμβαίνει αλλιώς, διότι 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.

Η υλοποίηση αιτήσεων του φλοιού μεταβάλλει δομές όπως την ουρά εκτέλεσης των διεργασιών και συνεπώς όσο αυτό συμβαίνει πρέπει να αποτρέψουμε την παράλληλη μεταβολή τους από κάποια άλλη συνάρτηση-handler η οποία κλήθηκε λόγω signals. Αν δεν απενεργοποιούσαμε τα σήματα, ουσιαστικά θα επιτρέπαμε την παρεμβολή κάποιου άλλου κομματιού κώδικα την ώρα εκτέλεσης των αιτήσεων του φλοιού με κίνδυνο να προκύψει ένα **race condition**, για παράδειγμα όσον αφορά τις λίστες της ουράς εκτέλεσης. Αυτό θα οδηγούσε ενδεχομένως σε **undefined behavior.**

Άσκηση 3

```
#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>

#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);
}
```

```c
/* 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;
}

/* 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. Cleaning 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. Cleaning 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 [name]: %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);
    }
}

/* 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;
}
```

Ερώτηση

1. **Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας.**

   Λιμοκτονία θα μπορούσε να δημιουργηθεί στην περίπτωση που τουλάχιστον μία από τις διεργασίες οι οποίες στην ουρά υψηλής προτεραιότητας δεν τερματιζόταν ποτέ. Ως αποτέλεσμα οι διεργασίες στην ουρά χαμηλής προτεραιότητας θα περίμεναν μάταια για τη σειρά τους και δε θα εκτελούνταν ποτέ, αφού η ουρά υψηλής προτεραιότητας είναι μη κενή.