ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ
ΑΣΚΗΣΗ 4

f15
Κουτσογιαννακόπουλος Ιωάννης 03109061
Στρατήγης Χαρίλαος 03112136

Άσκηση 1

Source code:

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>
#include <sys/wait.h>
#include <sys/types.h>
#include "proc-common.h"
#include "request.h"
/* Compile-time parameters. */
#define SCHED_TQ_SEC 2 /* time quantum */
#define TASK_NAME_SZ 60 /* maximum size for a task's name */
struct child{
pid_t mypid;
int myid ;
struct child *next;
};
typedef struct child child;
child *head;
child *tail;
int id;
void children(const char *name){
pid_t mypid = getpid();
printf("I am child %ld",(long)mypid);
char executable[TASK_NAME_SZ];
strcpy(executable,name);
char *newargv[] = { executable, NULL, NULL, NULL };
char *newenviron[] = { NULL };
printf("About to replace myself with the executable %s....\n",executable);
sleep(2);
execve(executable, newargv, newenviron);
printf("Execve failed,exiting...\n");
perror("execve");
exit(1);
}
void enqueue(pid_t pid, int flag){
//Queue is empty
if(head == NULL){
head = malloc(sizeof(child));
head->mypid = pid;
```

```
if(flag == 1){
head->myid = id;
id++;
}
head->next = NULL;
tail = head;
}
else {
child *new = malloc(sizeof(child));
new->mypid = pid;
if (flag == 1){
new->myid = id;
id++;
}
new->next = NULL;
tail->next = new;
tail = new;
}
}
pid_t dequeue(){
child *temp;
pid_t head_pid;
head_pid = head->mypid;
temp = head;
head = head->next;
free(temp);
return head_pid;
}
int remove_dead_child(pid_t dead_child){
/*
 *3 possible scenarios :
 *1)The child died and queue is empty => All tasks have been accomplished.
 *2)The child, first in the queue, died => Remove it from queue and renew timer on the next
one.
 *3)A child between head and tail died => Simply remove it from queue.
 */
child *temp;
child *pretail;
//First scenario :
if(head == tail){
free(head);
return 1;
}
//Second scenario :
if(head->mypid == dead_child){
temp = head;
```

```
head = head->next;
free(temp);
return 2;
}
temp = head;
pretail = head;
//Third scenario :
while (temp != NULL){

if (temp->mypid == dead_child){
 if (temp->next == NULL){
 tail = pretail;
tail->next = NULL;
 free(temp);
 return 0;
 }
 else{
 pretail->next = temp->next;
 free(temp);
 return 0;
 }
 }
 else{
 pretail = temp;
 temp = temp->next;
 }
 }
return 0;
}
/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
//assert(0 && "Please fill me!");//
if (signum != SIGALRM ) {
fprintf(stderr, " Internal error: Called for signum %d,not SIGALRM\n",signum);
exit(1);
}
printf("ALARM! %d seconds have passed.\n", SCHED_TQ_SEC);
pid_t remove_from_queue;
remove_from_queue = dequeue();
enqueue(remove_from_queue,0);
kill(remove_from_queue,SIGSTOP);
kill(head->mypid,SIGCONT);
```

```
if (alarm(SCHED_TQ_SEC)<0){
perror("alarm");
exit(1);
}
}
/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
//assert(0 && "Please fill me!");//
pid_t p;
int status;
if (signum!= SIGCHLD){
fprintf(stderr,"Internal error: Called for signum %d, not SIGCHLD\n",signum);
exit(1);
}
for(;;) {
p = waitpid(-1, &status, WUNTRACED | WNOHANG);
if (p<0) {
perror("waitpid");
exit(1);
}
if (p==0) break;
explain_wait_status(p,status);
if(WIFEXITED(status) || WIFSIGNALED(status)){
printf("Parent: Received SIGCHLD,child is dead. Removing..\n");
 int result = remove_dead_child(p);
if(result == 1){
printf("All tasks have been accomplished,exiting ....\n");
exit(1);
}
else if(result == 2){
kill(head->mypid,SIGCONT);
printf("Refreshing alarm for next child..\n");
 if(alarm(SCHED_TQ_SEC)<0){
 perror("alarm");
 exit(1);
 }
}
if(WIFSTOPPED(status)){
printf("Parent: Child has been stopped. Moving right along..\n");
}
}
}
```

```
}
/* Install two signal handlers.
 * One for SIGCHLD, one for SIGALRM.
 * Make sure both signals are masked when one of them is running.
 */
static void
install_signal_handlers(void)
{
sigset_t sigset;
struct sigaction sa;
sa.sa_handler = sigchld_handler;
sa.sa_flags = SA_RESTART;
sigemptyset(&sigset);
sigaddset(&sigset, SIGCHLD);
sigaddset(&sigset, SIGALRM);
sa.sa_mask = sigset;
if (sigaction(SIGCHLD, &sa, NULL) < 0) {
perror("sigaction: sigchld");
exit(1);
}
sa.sa_handler = sigalrm_handler;
if (sigaction(SIGALRM, &sa, NULL) < 0) {
perror("sigaction: sigalrm");
exit(1);
}
/*
 * Ignore SIGPIPE, so that write()s to pipes
 * with no reader do not result in us being killed,
 * and write() returns EPIPE instead.
 */
if (signal(SIGPIPE, SIG_IGN) < 0) {
perror("signal: sigpipe");
exit(1);
}
}
int main(int argc, char *argv[])
{
int i,nproc;
pid_t p;
tail = NULL;
head = NULL;
id = 0;
/*
 * 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 */
/* Wait for all children to raise SIGSTOP before exec()ing. */
//wait_for_ready_children(nproc);//
/* Install SIGALRM and SIGCHLD handlers. */
 if (nproc == 0) {
 fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
 exit(1);
 }
for(i=0; i<nproc; i++){
p = fork();
if(p<0){
perror("fork");
exit(1);
}
else if (p == 0){
/*
 *Child
 */
printf("Child created with pid = %ld \n" ,(long)getpid());
printf("Will now stop until all children will be created...\n");
raise(SIGSTOP);
children(argv[i+1]);
}
else {
/*
 *Father
 */
printf("Father process enqueue \n");
enqueue(p,1);
}
}
wait_for_ready_children(nproc);
install_signal_handlers();
printf("All children have been created...\n");
printf("Scheduler starting...\n");
kill(head->mypid,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;
}
```

Ερώτηση 1.1: 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.
Απάντηση:
Το πρόγραμμά μας στην περίπτωση αυτή θα αγνοήσει το σήμα SIGALRM μέχρι ότου εκτελεστεί η συνάρτηση χειρισμού του σήματος SIGCHLD και το αντίστροφο καθώς έχει δημιουργηθεί μια μάσκα για αυτά τα 2 σήματα.
Ένας πραγματικός χρονοδρομολογητής διαθέτει μια δομή για κάθε διαδικασία, η οποία περιέχει μια λίστα από σήματα και μια λιστα από handlers για τα αντίστοιχα σήματα. Έτσι, όταν καταφθάνουν σήματα, ο χρονοδομολογητής τα τοποθετεί σε συγκεκριμένη σειρά βάσει της λίστας αυτής και χρησιμοποιεί τους αντίστοιχους handlers για να τα χειριστεί.

Ερώτηση 1.2: 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;
Απάντηση:
Η υλοποίησή μας βασίζεται σε μία ουρά. Στην κεφαλή της ουράς βρίσκεται η διεργασία-παιδί που τρέχει κάθε φορά. Αν η κατάσταση του παιδιού αλλάξει ο πατέρας χρονοδρομολογητής δέχεται ένα σήμα SIGCHLD. Συνεπώς το σήμα SIGCHLD αναφέρεται κάθε φορά στην διαδικασία παιδί που βρίσκεται στην κεφαλή της ουράς.
Όσον αφορά το σήμα SIGKILL έχουμε τις εξής περιπτώσεις :

- Η διαδικασία-παιδί που βρίσκεται στην κεφαλή της ουράς δέχεται SIGKILL. Σε αυτήν την περίπτωση η τρέχουσα διαδικασία παιδί πεθαίνει και η επόμενη που θα τρέξει είναι η αμέσως επόμενη στην ουρά.
- Μία ενδιάμεση διαδικασία στην ουρά δέχεται SIGKILL. Εδώ η συγκεκριμένη διαδικασία πεθαίνει και την θέση της παίρνει η αμέσως επόμενη στην ουρά , ενώ η διεργασία-παιδί στην κεφαλή συνεχίζει να εκτελείται.
- Η ουρά μας περιέχει μία μόνο διεργασία η οποία δέχεται SIGKILL. Εδώ η διεργασία θα πεθάνει και το πρόγραμμα θα τερματίσει.

Ερώτηση 1.3: 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.
Απάντηση:
Εάν δεν χειριζόμαστε το σήμα SIGCHLD δεν θα μπορούσαμε να διαχειριστούμε τον 'θάνατο' μίας διεργασίας παιδιού. Η διεργασία χρονοδρομολογητής που έχουμε υλοποιήσει βρίσκεται στο επίπεδο χρήστη. Συνεπώς χρονοδρομολογείται και αυτή. Εάν λοιπόν στείλουμε SIGSTOP σε μία διεργασία μέσω SIGALRM υπάρχει το ενδεχόμενο ο χρονοδρομολογητής να στείλει SIGCONT στην επόμενη διεργασία για να εκτελεστεί, ενώ η προηγούμενη

διεργασία παιδί να μην έχει λάβει ακόμη SIGSTOP. Έτσι μπορεί να έχουμε 2 διεργασίες να εκτελούνται παράλληλα,πράγμα που είναι ανεπιθύμητο.

Άσκηση 2

Source code:

```
#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 "queue.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 */

/*the process queue*/
queue * q;

int id = 0;
/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void){
        node * temp ;
        temp = q->head;
        while(temp != NULL){
                printf("Tasks:\nid: %d | pid: %d,| name:
%s\n",temp->p->myid,temp->p->pid,temp->p->name);
                temp = temp->pre;

        }
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id){
        node *temp;
        temp = q->head;
        while(temp !=NULL){
                if (temp->p->myid == id){
                        kill(temp->p->pid,SIGTERM); //why not SIGTERM??
                        break;
```

```
                }
                temp = temp->pre;
        }
        return -ENOSYS;
}


/* Create a new task.  */
static void
sched_create_task(char *executable){
        struct process *proc;
        pid_t p = fork();
                if (p < 0) {
                        perror("fork");
                        exit(1);
                }

                if (p == 0) {
                        char *newargv[] = { executable, NULL, NULL, NULL };
        char *newenviron[] = { NULL };
                        raise(SIGSTOP);
                        execve(executable,newargv,newenviron);
                        exit(1);
                }
                proc = malloc(sizeof(struct process));
                proc->name = malloc(10*sizeof(char));
                proc->pid = p;
                id++;
                proc->myid = id;
                strcpy(proc->name,executable);
                enqueue(proc,q);
}

/* 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;
```

```
        }
}

/* 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);
        }
}


/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum){
        struct process * proc;
        proc = get_top(q);
        kill(proc->pid,SIGSTOP);

         if (alarm(SCHED_TQ_SEC) < 0) {
            perror("alarm");
            exit(1);
       }
}

/*
 * SIGCHLD handler
```

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

        struct process *p;
        pid_t pid;
        int status;
        for (;;) {
        pid = waitpid(-1, &status, WUNTRACED | WNOHANG);
        if (pid < 0) {
                perror("waitpid");
                exit(1);
        }
        if (pid == 0)
                break;

        explain_wait_status(pid, status);

        if (WIFEXITED(status) || WIFSIGNALED(status)) {
                /* A child has died */
               printf("Parent: Received SIGCHLD, child  %s is dead.\n", name_by_pid(pid,q));
                                p = get_top(q);
                                dequeue(q);
                                p = get_top(q);

                                if (p == NULL ) {
                                        printf("No more process\n");
                                        exit(1);
                                        }

                                printf("starting process %s\n",p->name);
                                kill(p->pid,SIGCONT);
                                        if (alarm(SCHED_TQ_SEC) < 0) { // reset timer
                                        perror("alarm");
                                        exit(1);
                                }
                        }
                    if (WIFSTOPPED(status)) {
                                /* A child has stopped due to SIGSTOP/SIGTSTP, etc... */
                                printf("Parent: Child has been stopped. Moving right along...\n");
                                p = dequeue(q);

                                if (p == NULL ) {
                                        printf("No more process\n");
                                        exit(1);
```

```
                        }
                        enqueue(p,q);
                        p = get_top(q);
                        if (p == NULL) break;
                        printf("starting process %s\n",p->name);
                        kill(p->pid,SIGCONT);
                }
        }
}

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

        sa.sa_handler = sigchld_handler;
        sa.sa_flags = SA_RESTART;
        sigemptyset(&sigset);
        sigaddset(&sigset, SIGCHLD);
        sigaddset(&sigset, SIGALRM);
        sa.sa_mask = sigset;
        if (sigaction(SIGCHLD, &sa, NULL) < 0) {
                perror("sigaction: sigchld");
                exit(1);
        }

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

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


static void
do_shell(char *executable, int wfd, int rfd)
```

```
{
	char arg1[10], arg2[10];
	char *newargv[] = { executable, NULL, NULL, NULL };
	char *newenviron[] = { NULL };

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

	raise(SIGSTOP);
	execve(executable, newargv, newenviron);

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

/* Create a new shell task.
 *
 * The shell gets special treatment:
 * two pipes are created for communication and passed
 * as command-line arguments to the executable.
 */
static void
sched_create_shell(char *executable, int *request_fd, int *return_fd)
{
	pid_t p;
	int pfds_rq[2], pfds_ret[2];
	struct process *proc;
	if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
		perror("pipe");
		exit(1);
	}
	p = fork();
	if (p < 0) {
		perror("scheduler: fork");
		exit(1);
	}

	if (p == 0) {
		/* Child */
		close(pfds_rq[0]);
		close(pfds_ret[1]);
		do_shell(executable, pfds_rq[1], pfds_ret[0]);
		assert(0);
	}
	/* Parent */
	close(pfds_rq[1]);
	close(pfds_ret[0]);
	*request_fd = pfds_rq[0];
```

```
        *return_fd = pfds_ret[1];

        //Store in node list
        proc = malloc(sizeof(struct process));
        proc->pid = p;
        id++;
        proc->myid = id;//id for the shell?
        proc->name = executable;
        enqueue(proc,q);
}

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;
        int i;
        pid_t p;

        q = init_queue();
        /* Create the shell. */
        sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
        /* add the shell to the scheduler's tasks */
```

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

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

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

struct process * proc;
for (i = 0; i<nproc;i++) {
        p = fork();
        if (p < 0) {
                perror("fork");
                exit(1);
        }

        if (p == 0) {
                char *newargv[] = { argv[i+1], NULL, NULL, NULL };
        char *newenviron[] = { NULL };
                raise(SIGSTOP);
                execve(argv[i+1],newargv,newenviron);
                exit(1);
        }
        proc = malloc(sizeof(struct process));
        proc->name = malloc(20*sizeof(char));
        proc->pid = p;
        id++;
        proc->myid = id;
        strcpy(proc->name, argv[i+1]);
        enqueue(proc,q);

}

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

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

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

proc = get_top(q);
kill(proc->pid,SIGCONT);
```

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

Ερώτηση 2.1: Όταν και ο φλοιός υφίσταται χρονοδρομολόγηση, ποια εμφανίζεται πάντοτε ως τρέχουσα διεργασία στη λίστα διεργασιών (εντολή 'p'); Θα μπορούσε να μη συμβαίνει αυτό; Γιατί;

Απάντηση:

Στην υλοποίησή μας τυπώνεται πρώτα πάντα ο φλοιός. Σε περίπτωση που δώσουμε την εντολή λίγο πριν εκπνεύσει το κβάντο χρόνου, μπορεί ο χρονοδρομολογητής να έχει ήδη προλάβει να δείξει στην επόμενη διεργασία,οπότε και να εμφανιστεί αυτή.

Ερώτηση 2.2: 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.

Απάντηση:

Ο λόγος ύπαρξης των κλήσεων αυτών είναι για να διασφαλίζεται η δομή της ουράς των διεργασιών καθώς κάποιες αιτήσεις φλοιού όπως η kill ή η create μπορούν να την επηρεάσουν.

Άσκηση 3

Source code:

```
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>
#include <sys/wait.h>
#include <sys/types.h>
#include "proc-common.h"
#include "request.h"
/* Compile-time parameters. */
#define SCHED_TQ_SEC 2 /* time quantum */
#define TASK_NAME_SZ 60 /* maximum size for a task's name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */
/*Struct Process ID*/
struct PT{
int myid;
int mypriority;
pid_t mypid;
char name[TASK_NAME_SZ];
struct PT *next;
};
typedef struct PT ProcessTable ;
/*id for every process*/
int id;
/*Head&Tail of queue*/
ProcessTable *head;
ProcessTable *tail;
void printQ(){
ProcessTable *temp;
temp = head;
printf("-----------Queue-----------------------\n");
while(temp != NULL){
printf("name = %s, pid = %d, prio = %d \n",temp->name,temp->mypid,temp->mypriority);
temp = temp -> next;
}
printf("---------------------------------------\n");
}
void children(const char *name){
pid_t mypid = getpid();
printf("I am child %ld",(long)mypid);
char executable[TASK_NAME_SZ];
```

```
strcpy(executable,name);
char *newargv[] = { executable, NULL, NULL, NULL};
char *newenviron[] = { NULL };
printf("About to replace myself with the executable %s... \n", executable);
sleep(2);
execve(executable,newargv,newenviron);
printf("Execve failed...\n");
perror("execve");
exit(1);
}
void enqueue(ProcessTable node ,int flag){
if(head == NULL){
head = (ProcessTable *) malloc(sizeof(ProcessTable));
head->mypid = node.mypid;
head->mypriority = node.mypriority;
if(flag ==1){
head->myid = id;
id++;
}
printf("priority on enqueue; %d\n", node.mypriority);
strcpy(head->name,node.name);
head->next = NULL;
tail = head;
}
else{
ProcessTable *new = (ProcessTable*) malloc(sizeof(ProcessTable));
new->mypid = node.mypid;
if(flag == 1){
new->myid = id;
id++;
}
else {
new->myid = node.myid;
}
new->mypriority = node.mypriority;
strcpy(new->name,node.name);
new->next = NULL;
tail->next = new;
tail = new;
}
}
ProcessTable dequeue(){
ProcessTable *temp,dequeued;
dequeued.mypid = head->mypid;
dequeued.mypriority = head->mypriority;
dequeued.myid = head->myid;
```

```
strcpy(dequeued.name,head->name);
temp = head;
head = head->next;
free(temp);
return dequeued;
}
int IsItHigh(){
ProcessTable *temp;
temp = head;
while(temp != NULL){
if((temp->mypriority) == 1)
return 1;
else
temp = temp->next;
}
return 0;
}
void FindNext(){
ProcessTable remove_from_queue;
ProcessTable *temp;
int check;
int count = 0;
int i,flag;
flag = 0;
check = IsItHigh();
if(check == 0){
remove_from_queue = dequeue();
kill(remove_from_queue.mypid,SIGSTOP);
enqueue(remove_from_queue,0);
kill(head->mypid,SIGCONT);
}
else{
remove_from_queue = dequeue();
kill(remove_from_queue.mypid,SIGSTOP);
enqueue(remove_from_queue,0);
temp = head;
while(temp != NULL){
if((temp->mypriority) == 1){
flag = 1;
break;
}
else{
temp = temp ->next;
count++;
}
}
```

```
if(flag == 1) {
for(i=0; i<count; i++){
remove_from_queue = dequeue();
enqueue(remove_from_queue,0);
}
}
kill(head->mypid,SIGCONT);
}
}
int remove_dead_child(pid_t dead_process){
ProcessTable *temp;
ProcessTable *pretail;
//First scenario:
if(head == tail){
free(head);
return 1;
}
//Second scenario:
if(head->mypid == dead_process){
temp = head;
head = head->next;
free(temp);
return 2;
}
temp = head;
pretail = head;
//Third scenario:
while(temp!= NULL){
if(temp->mypid == dead_process){
if(temp->next == NULL){
tail = pretail;
tail->next = NULL;
free(temp);
return 0;
}
else{
pretail->next = temp->next;
free(temp);
return 0;
}
}
else{
pretail = temp;
temp = temp->next;
}
}
```

```
return 0;
}
/*: Print a list of all tasks currently being scheduled. */
static void
sched_print_tasks(void)
{
int check;
ProcessTable *temp;
temp = head;
check = IsItHigh();
if(check == 0){
while(temp != NULL){
printf("name : %s | id = %d | pid = %d | prio = %d \n", temp->name, temp->myid,
temp->mypid, temp->mypriority);
temp = temp -> next;
}
}
else{
while(temp != NULL){
if((temp->mypriority) == 1){
printf("name : %s | id = %d | pid = %d | prio = %d", temp->name, temp-
>myid, temp->mypid, temp->mypriority);
}
temp = temp->next;
}
}
}
/*: Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int my_id)
{
//assert(0 && "Please fill me!");
ProcessTable *temp;
temp = head;
while(temp != NULL){
if((temp->myid) == my_id){
kill((temp->mypid),SIGTERM);
break;
}
temp = temp -> next;
}
return -ENOSYS;
}
/* Create a new task. */
```

```
static void
sched_create_task(char *executable)
{
//assert(0 && "Please fill me!");//
pid_t p = fork();
if (p<0){
perror("fork");
exit(1);
}
if (p==0){
raise(SIGSTOP);
children(executable);
}
ProcessTable process;
process.mypid = p;
strcpy(process.name,executable);
enqueue(process,1);
}
static void
sched_priority(int my_id,int my_priority){
ProcessTable *temp;
temp = head;
while(temp != NULL){
if((temp->myid) == my_id){
temp->mypriority = my_priority;
break;
}
else
temp = temp->next;
}
}
/* Process requests by the shell. */
static int
process_request(struct request_struct *rq)
{
switch (rq->request_no) {
case REQ_PRINT_TASKS:
sched_print_tasks();
return 0;
case REQ_KILL_TASK:
return sched_kill_task_by_id(rq->task_arg);
case REQ_EXEC_TASK:
sched_create_task(rq->exec_task_arg);
return 0;
case REQ_HIGH_TASK:
sched_priority(rq->task_arg,1);
```

```
return 0;
case REQ_LOW_TASK:
sched_priority(rq->task_arg,0);
return 0;
default:
return -ENOSYS;
}
}
/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
//assert(0 && "Please fill me!");//
if(signum!= SIGALRM){
fprintf(stderr,"Internal error: Called for signum %d,not SIGALRM",signum);
exit(1);
}
printf("ALARM! %d seconds have passes.\n",SCHED_TQ_SEC);
FindNext();
if(alarm(SCHED_TQ_SEC)<0){
perror("alarm");
exit(1);
}
}
/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
pid_t p;
int status;
if (signum!= SIGCHLD){
fprintf(stderr,"Internal error: Called for signum %d, not SIGCHLD\n",signum);
exit(1);
 }
for(;;){
 p = waitpid(-1, &status, WUNTRACED | WNOHANG);
if (p<0) {
perror("waitpid");
exit(1);
}
if (p==0) break;
explain_wait_status(p,status);
```

```
if (WIFEXITED(status) || WIFSIGNALED(status)){
printf("Parent: Received SIGCHLD,child is dead.Removing...\n");
int result = remove_dead_child(p);
if(result == 1){
printf("All tasks have been accomplished,exiting...\n");
exit(1);
}
else if (result == 2){
FindNext();
printf("Refreshing alarm for next child...\n");
if (alarm(SCHED_TQ_SEC)<0){
perror("alarm");
exit(1);
}
}
if (WIFSTOPPED(status)){
printf("Parent: Child has been stopped. Moving right along..\n");
}
}
}
}
/* Disable delivery of SIGALRM and SIGCHLD. */
static void
signals_disable(void)
{
sigset_t sigset;
sigemptyset(&sigset);
sigaddset(&sigset, SIGALRM);
sigaddset(&sigset, SIGCHLD);
if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
perror("signals_disable: sigprocmask");
exit(1);
}
}
/* Enable delivery of SIGALRM and SIGCHLD. */
static void
signals_enable(void)
{
sigset_t sigset;
sigemptyset(&sigset);
sigaddset(&sigset, SIGALRM);
sigaddset(&sigset, SIGCHLD);
if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
perror("signals_enable: sigprocmask");
exit(1);
}
```

```
}
/* Install two signal handlers.
 * One for SIGCHLD, one for SIGALRM.
 * Make sure both signals are masked when one of them is running.
 */
static void
install_signal_handlers(void)
{
sigset_t sigset;
struct sigaction sa;
sa.sa_handler = sigchld_handler;
sa.sa_flags = SA_RESTART;
sigemptyset(&sigset);
sigaddset(&sigset, SIGCHLD);
sigaddset(&sigset, SIGALRM);
sa.sa_mask = sigset;
if (sigaction(SIGCHLD, &sa, NULL) < 0) {
perror("sigaction: sigchld");
exit(1);
}
sa.sa_handler = sigalrm_handler;
if (sigaction(SIGALRM, &sa, NULL) < 0) {
perror("sigaction: sigalrm");
exit(1);
}
/*
 * Ignore SIGPIPE, so that write()s to pipes
 * with no reader do not result in us being killed,
 * and write() returns EPIPE instead.
 */
if (signal(SIGPIPE, SIG_IGN) < 0) {
perror("signal: sigpipe");
exit(1);
}
}
static void
do_shell(char *executable, int wfd, int rfd)
{
char arg1[10], arg2[10];
char *newargv[] = { executable, NULL, NULL, NULL };
char *newenviron[] = { NULL };
sprintf(arg1, "%05d", wfd);
sprintf(arg2, "%05d", rfd);
newargv[1] = arg1;
newargv[2] = arg2;
raise(SIGSTOP);
```

```
execve(executable, newargv, newenviron);
/* execve() only returns on error */
perror("scheduler: child: execve");
exit(1);
}
/* Create a new shell task.
 *
 * The shell gets special treatment:
 * two pipes are created for communication and passed
 * as command-line arguments to the executable.
 */
static void
sched_create_shell(char *executable, int *request_fd, int *return_fd)
{
pid_t p;
int pfds_rq[2], pfds_ret[2];
if (pipe(pfds_rq) < 0 || pipe(pfds_ret) < 0) {
perror("pipe");
exit(1);
}
p = fork();
if (p < 0) {
perror("scheduler: fork");
exit(1);
}
if (p == 0) {
/* Child */
close(pfds_rq[0]);
close(pfds_ret[1]);
do_shell(executable, pfds_rq[1], pfds_ret[0]);
assert(0);
}
/* Parent p = p_shell store it in queue */
close(pfds_rq[1]);
close(pfds_ret[0]);
*request_fd = pfds_rq[0];
*return_fd = pfds_ret[1];
//Store in queue
ProcessTable process;
process.mypid = p;
process.mypriority = 0;
strcpy(process.name, SHELL_EXECUTABLE_NAME);
enqueue(process,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 i,nproc;
id = 0;
pid_t p;
tail = NULL;
head = NULL;
/* Two file descriptors for communication with the shell */
static int request_fd, return_fd;
/* Create the shell. */
sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
/* add the shell to the scheduler's tasks->mallon egine */
/*
 * 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 */
for(i=0; i<nproc; i++){
p = fork();
if(p < 0){
perror("fork");
exit(1);
}
else if(p == 0){
```

```
//Child
raise(SIGSTOP);
children(argv[i+1]);
}
else {
printf("Father process enqueue..\n");
ProcessTable process;
process.mypid = p;
process.mypriority = 0;
strcpy(process.name,argv[i+1]);
enqueue(process,1);
}
}
/* Wait for all children to raise SIGSTOP before exec()ing. */
wait_for_ready_children(nproc);
/* Install SIGALRM and SIGCHLD handlers. */
install_signal_handlers();
if (nproc == 0) {
fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
exit(1);
}
printf("All children are ready,start schedule....\n");
kill(head->mypid,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 3.1: Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας
Απάντηση:
Θεωρούμε σενάριο λιμοκτονίας όταν μια διεργασία (συνήθως βαριά και χρονοβόρα) καταλαμβάνει το σύστημα και δεν αφήνει τις υπόλοιπες να τρέξουν, οδηγόντας σε κατάσταση κατα την οποια αυτές “λιμοκτονούν”.

Σενάριο το οποίο οδηγεί σε τέτοια κατάσταση είναι τα ακόλουθα :

- Θέτουμε το “shell” με προτεραιότητα “ high - h”. Σε αυτό το σενάριο εκτελείτε συνεχώς η διεργασία - shell και οι υπόλοιπες διεργασίες λιμοκτονούν. Αν δε τεθεί κάποια άλλη διεργασία με προτεραιότητα “high -h” ή αν δε κατεβεί το “shell” σε προτεραιότητα “low - l”, η μονη διεργασία που θα τρέχει θα ειναι το “shell”, καθώς μετά το πέρας κάθε κβάντου χρόνου το alarm θα ξανατίθεται και θα επανεκινείτε η ίδια διεργασία.
- Θέτουμε αντίστοιχα κάποια συγκεκριμένη διεργασία/ίες σε “high - h” προτεραιότητα με το shell να παραμένει σε “low - l” προτεραιότητα. Σε αυτή την περίπτωση είμαστε αναγκασμένοι να περιμένουμε μέχρι όλες οι διεργασίες σε “high - h” προτεραιότητα να εκπεύσουν και ο αριθμός των διεργασιών σε “high” προτεραιότητα να είναι 0.