Λειτουργικά Συστήματα
6ο Εξάμηνο
Άσκηση 4: Χρονοδρομολόγηση
Αναφορά

Χαρδούβελης Γεώργιος-Ορέστης
el15100
6ο Εξάμηνο

Κυριάκου Αθηνά
el17405
6ο Εξάμηνο

Άσκηση 1:

Ο πηγαίος κώδικας του αρχείου scheduler.c όπως τροποποιήθηκε φαίνεται παρακάτω:

```
#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 process_control_block{    //Process Control Block Struct
(node)
     int id;
     pid_t pid;
     char proc_name[TASK_NAME_SZ];
     struct process_control_block *next;  //queue of processes
};

typedef struct process_control_block PCB;
```

```
PCB *head=NULL;
PCB *tail=NULL; //the queue is defined by this 2 pointers, pointing
to its first and last node
//head=nullptr;
//tail=nullptr;
//head=(PCB*)malloc(sizeof(PCB));
//tail=head;

void add_proc(int id,pid_t pid, char proc_name[TASK_NAME_SZ]){
//function to add the new proc, enqueue

      PCB *temp=(PCB*)malloc(sizeof(PCB));
      temp->id=id;
      temp->pid=pid;
      strcpy(temp->proc_name,proc_name);
      temp->next=NULL;

      if(head==NULL) head=temp;
      else tail->next=temp;

      tail=temp;
}

/*
 * SIGALRM handler
 */
static void     //sends the SIGSTOP signal when tq has passed
sigalrm_handler(int signum)
{
      //assert(0 && "Please fill me!");
      printf("ALARM!Time %dsec has passed!\n",SCHED_TQ_SEC);
      printf("Sending SIGSTOP to procedure with PID=%ld\n",head-
>pid);
      if(kill(head->pid,SIGSTOP)<0){
                perror("kill");
                exit(1);
      }
      //initializing the alarm again
      //printf("Initializing the clock!\n");
      //alarm(SCHED_TQ_SEC);          //set the alarm

}
```

```
/*
 * SIGCHLD handler
 */
static void     //sends the SICONT, when a child changes status
sigchld_handler(int signum)
{
     //assert(0 && "Please fill me!");
     for(;;){ //infinite loop

                int *status;
                pid_t p=waitpid(-1,&status,WUNTRACED | WNOHANG);
                if(p<0){
                     perror("waitpid");
                     exit(1);
                }
                else if(p==0)   //WUNTRACED flag is specified, no
child changes state
                break;

                explain_wait_status(p,status); //if a child changed
state

                if(WIFEXITED(status) || WIFSIGNALED(status)){
//child died
                     //printf("Procedure with PID=%ld, died!\
n",head->pid);
                     head=head->next;
                     free(tail->next);
                     tail->next=head;
                }
                if(WIFSTOPPED(status)){ //child stopped
                     //printf("Procedure with PID=%ld, stopped!\
n",head->pid);
                     head=head->next;
                     tail=tail->next;
                }

                printf("Procedure with PID=%ld, about to continue!\
n",head->pid);

                //initializing the alarm again
                printf("Initializing the clock!\n");
```

```
                alarm(SCHED_TQ_SEC);            //set the alarm

                kill(head->pid,SIGCONT);        //starting the next
procedure
        }
}

/* 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;  //signals that should be blocked during
the execution of the handler

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


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


        /*
        * Ignore SIGPIPE, so that write()s to pipes
```

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

int main(int argc, char *argv[])
{
	int nproc,i;
	pid_t p;		//temp variable to store the proccedure
	/*
	* 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(); //creation of new procedure
		if (p<0){ //if there is an error in fork()
			perror("fork");
			exit(1);
		}
		if(p==0){ //code of the child-procedure

			raise(SIGSTOP);
			char *newargv[]={argv[i+1],NULL,NULL,NULL};
			char *newenviron[]={NULL};
			printf("I am Child_%d,PID=%ld\n",i,
(long)getpid());
			printf("Imma 'bout to replace my code with the
executable %s...\n",argv[i+1]);
			sleep(1);
			execve(argv[i+1],newargv,newenviron);
			perror("execve");
			exit(1);
		}
		//scheduler-father code, process in the queue, p is
the PID of the child process
		add_proc(i,p,argv[i+1]);
```

```
	}
	tail->next=head; //circular list (RR)
	/* 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){  //if no procedures
		fprintf(stderr, "Scheduler: No tasks. Exiting...\
n");
		exit(1);
	}
	else{   //waking up the first-created child
		printf("Continuing the first child with PID=%ld!\
n",head->pid);
		printf("Initializing the clock for the first child!\
n");

		alarm(SCHED_TQ_SEC);            //set the alarm
		kill(head->pid,SIGCONT);

	}

	/* loop forever  until we exit from inside a signal handler. */
	while (pause())
		;

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

Ένα ενδεικτικό output του παραπάνω κώδικα φαίνεται εδώ:

```
oslabd14@os-node2:~/lab4$ ./scheduler prog prog
My PID = 27593: Child PID = 27594 has been stopped by a signal, signo = 19
My PID = 27593: Child PID = 27595 has been stopped by a signal, signo = 19
Continuing the first child with PID=27594!
Initializing the clock for the first child!
I am Child_0,PID=27594
Imma 'bout to replace my code with the executable prog...
prog: Starting, NMSG = 20, delay = 44
prog[27594]: This is message 0
prog[27594]: This is message 1
prog[27594]: This is message 2
prog[27594]: This is message 3
prog[27594]: This is message 4
prog[27594]: This is message 5
ALARM!Time 2sec has passed!
Sending SIGSTOP to procedure with PID=27594
My PID = 27593: Child PID = 27594 has been stopped by a signal, signo = 19
Procedure with PID=27595, about to continue!
Initializing the clock!
I am Child_1,PID=27595
Imma 'bout to replace my code with the executable prog...
prog: Starting, NMSG = 20, delay = 90
prog[27595]: This is message 0
prog[27595]: This is message 1
prog[27595]: This is message 2
ALARM!Time 2sec has passed!
Sending SIGSTOP to procedure with PID=27595
My PID = 27593: Child PID = 27595 has been stopped by a signal, signo = 19
Procedure with PID=27594, about to continue!
Initializing the clock!
prog[27594]: This is message 6
prog[27594]: This is message 7
prog[27594]: This is message 8
prog[27594]: This is message 9
prog[27594]: This is message 10
prog[27594]: This is message 11
prog[27594]: This is message 12
prog[27594]: This is message 13
prog[27594]: This is message 14
prog[27594]: This is message 15
prog[27594]: This is message 16
prog[27594]: This is message 17
ALARM!Time 2sec has passed!
```

```
Sending SIGSTOP to procedure with PID=27594
My PID = 27593: Child PID = 27594 has been stopped by a signal, signo = 19
Procedure with PID=27595, about to continue!
Initializing the clock!
prog[27595]: This is message 3
prog[27595]: This is message 4
prog[27595]: This is message 5
prog[27595]: This is message 6
prog[27595]: This is message 7
prog[27595]: This is message 8
ALARM!Time 2sec has passed!
Sending SIGSTOP to procedure with PID=27595
My PID = 27593: Child PID = 27595 has been stopped by a signal, signo = 19
Procedure with PID=27594, about to continue!
Initializing the clock!
prog[27594]: This is message 18
prog[27594]: This is message 19
My PID = 27593: Child PID = 27594 terminated normally, exit status = 0
Procedure with PID=27595, about to continue!
Initializing the clock!
prog[27595]: This is message 9
prog[27595]: This is message 10
prog[27595]: This is message 11
prog[27595]: This is message 12
prog[27595]: This is message 13
prog[27595]: This is message 14
ALARM!Time 2sec has passed!
Sending SIGSTOP to procedure with PID=27595
My PID = 27593: Child PID = 27595 has been stopped by a signal, signo = 19
Procedure with PID=27595, about to continue!
Initializing the clock!
prog[27595]: This is message 15
prog[27595]: This is message 16
prog[27595]: This is message 17
prog[27595]: This is message 18
prog[27595]: This is message 19
My PID = 27593: Child PID = 27595 terminated normally, exit status = 0
Procedure with PID=0, about to continue!
Initializing the clock!
waitpid: No child processes
oslabd14@os-node2:~/lab4$
```

Σημειώνεται ότι το πλήθος των μηνυμάτων που έχει ρυθμιστεί να εκτυπώνονται από το εκτελέσιμο prog είναι 20.

Άσκηση 2:

Ο πηγαίος κώδικας του αρχείου scheduler-shell.c όπως τροποποιήθηκε φαίνεται παρακάτω:

```
#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 4                /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's
name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */


int nproc=0; //number of procedures for scheduling

struct process_control_block{    //Process Control Block Struct
(node)
     int id;
     pid_t pid;
     char proc_name[TASK_NAME_SZ];
     struct process_control_block *next;  //queue of processes
};

typedef struct process_control_block PCB;

PCB *head=NULL;
PCB *tail=NULL; //the queue is defined by this 2 pointers, pointing
to its first and last node
```

```
void add_proc(int id,pid_t pid, char proc_name[TASK_NAME_SZ]){
//function to add the new proc, enqueue

      PCB *temp=(PCB*)malloc(sizeof(PCB));
      temp->id=id;
      temp->pid=pid;
      strcpy(temp->proc_name,proc_name);
      temp->next=NULL;
      if(head==NULL) head=temp;
      else tail->next=temp;

      tail=temp;
      tail->next=head;
}

//_____________SHELL FUNCTIONS_________________

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

      printf("\nPrinting a list of tasks currently scheduled!\n");
      //assert(0 && "Please fill me!");

      PCB *temp=head;
printf("Current running task pid=%ld with id=%d.\n",head->pid, head-
>id);       //scheduler


      //printf("Task %s: id=%d and pid=%ld.\n",temp->proc_name,temp-
>id,temp->pid);

      do{
                  temp=temp->next;
                  printf("Task %s: id=%d and pid=%ld.\n",temp-
>proc_name,temp->id,temp->pid);

      }while(temp!=head);
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
```

```
 */
static int
sched_kill_task_by_id(int id)
{
      //assert(0 && "Please fill me!");

      PCB *temp_h=head;
      PCB *temp_t;

      int found=0;
      do{

                  temp_t=temp_h;
                  temp_h=temp_h->next;
                  if(temp_h->id==id){
                        found=1;
                        if(strcmp(temp_h-
>proc_name,SHELL_EXECUTABLE_NAME)==0){ //if it is a shell
                                    printf("Press q to kill the shell!\
n");
                                    break;
                        }

                        //killing
                        printf("Killing process id=%d...\n",id);
                        kill(temp_h->pid,SIGKILL);

                        //taking the process out of the queue
                        if(temp_h==head)
                                    head=head->next;
                        else if(temp_h==tail)
                                    tail=temp_t;

                        printf("Taking process %s with id=%d and pid=
%ld out of the queue.\n",temp_h->proc_name,temp_h->id,temp_h->pid);
                        temp_t->next=temp_h->next;
                        temp_h->next=NULL;
                        free(temp_h);
                        break;
                  }

      }
      while(temp_h!=head);
```

```
	if(found==0) printf("No task with the specified id was found!\
n");

	//return -ENOSYS;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
	//assert(0 && "Please fill me!");

	nproc++;
	pid_t p=fork();
	if (p<0){ //if there is an error in fork()
			perror("fork");
			exit(1);
	}
	if(p==0){
			raise(SIGSTOP);
			char *newargv[]={executable,NULL,NULL,NULL};
			char *newenviron[]={NULL};
			printf("I am Shell_proc_%d,PID=%ld\n",nproc,
(long)getpid());
			printf("Imma 'bout to replace my code with the
executable %s...\n",executable);
			sleep(1);
			execve(executable,newargv,newenviron);
			perror("execve");
			exit(1);
	}
	add_proc(nproc,p,executable);   //adding the new proc in the
scheduling queue

}

/* 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;
        }
}
//____________________HANDLERS_____________________

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        //assert(0 && "Please fill me!");
        printf("ALARM!Time %dsec has passed!\n",SCHED_TQ_SEC);
        printf("Sending SIGSTOP to procedure with PID=%ld\n",head-
>pid);
        if(kill(head->pid,SIGSTOP)<0){
                perror("kill");
                exit(1);
        }
}


/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        //assert(0 && "Please fill me!");
        for(;;){ //infinite loop
```

```
                int *status;
                pid_t p=waitpid(-1,&status,WUNTRACED | WNOHANG);
                if(p<0){
                     perror("waitpid");
                     exit(1);
                }
                else if(p==0)   //WUNTRACED flag is specified, no
child changes state
                break;

                explain_wait_status(p,status); //if a child changed
state

                if(WIFEXITED(status)){   //child terminated normally
                     //printf("Procedure with PID=%ld, died!\
n",head->pid);
                     head=head->next;
                     free(tail->next);

                     tail->next=head;

                }

                if(WIFSIGNALED(status)) //child killed by a signal
(running process or not??)
                     break;

                if(WIFSTOPPED(status)){ //child stopped
                     //printf("Procedure with PID=%ld, stopped!\
n",head->pid);
                     head=head->next;
                     tail=tail->next;
                }

                printf("Procedure with PID=%ld, about to continue!\
n",head->pid);

                //initializing the alarm again
                printf("Initializing the clock!\n");
                alarm(SCHED_TQ_SEC);            //set the alarm

                kill(head->pid,SIGCONT);        //starting the next
procedure
```

```
	}
}


/* 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 */
```

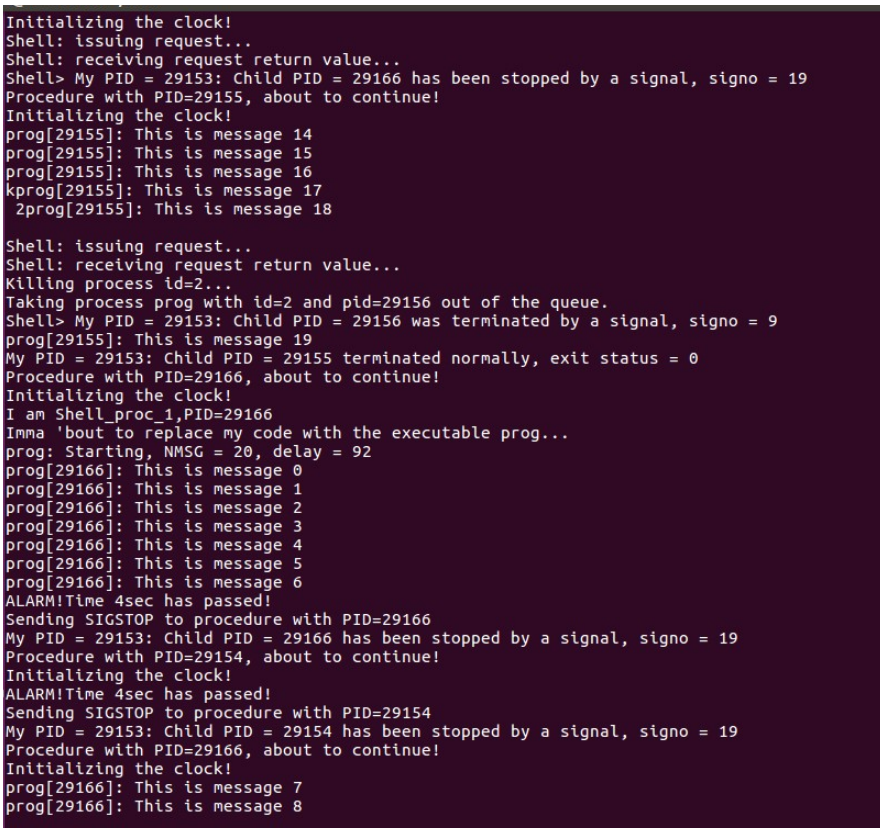
```
	//add the shell in the procedure queue once created
	//nproc++;
	add_proc(nproc,p,SHELL_EXECUTABLE_NAME);

	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;
		}
	}
}
```

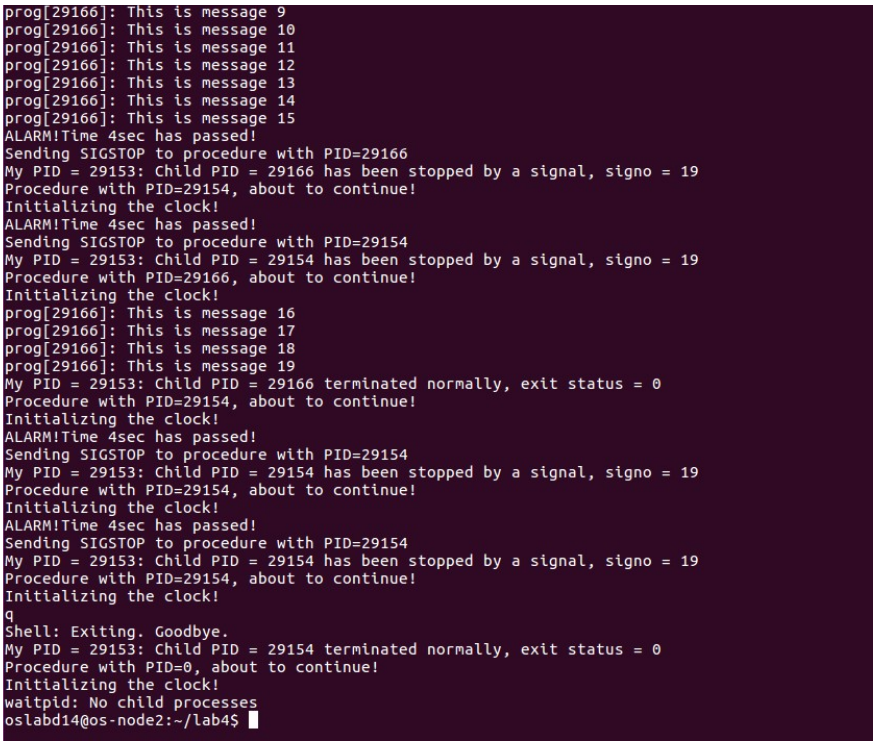
```c
int main(int argc, char *argv[]){

      int i,nproc,nproc2;
      pid_t p;

      /* 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);
      /* TODO: 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 */
      nproc2=argc-1;

      for(i=1; i<=nproc2; i++){ //the first added process in the
queue was the shell

                  p=fork(); //creation of new procedure
                  if (p<0){ //if there is an error in fork()
                        perror("fork");
                        exit(1);

                  }
                  if(p==0){ //code of the child-procedure

                        raise(SIGSTOP);
                        char *newargv[]={argv[i],NULL,NULL,NULL};
                        char *newenviron[]={NULL};
                        printf("I am Child_%d,PID=%ld\n",i,
(long)getpid());
                        printf("Imma 'bout to replace my code with the
executable %s...\n",argv[i]);
                        sleep(1);
                        execve(argv[i],newargv,newenviron);
                        perror("execve");
                        exit(1);
```

```
                }
                //scheduler-father code, process in the queue, p is
the PID of the child process
                add_proc(i,p,argv[i]);

     }
     nproc++; //shell was added
     tail->next=head; //make circular queue

     /* 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);
     }
     else{   //waking up the first-created child
                printf("Continuing the first child with PID=%ld!\
n",head->pid);
                printf("Initializing the clock for the first child!\
n");

                alarm(SCHED_TQ_SEC);            //set the alarm
                kill(head->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("Shell gone!\n");

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

Ένα ενδεικτικό output του παραπάνω κώδικα φαίνεται εδώ:

```
oslabd14@os-node2:~/lab4$ ./scheduler-shell prog prog
My PID = 29153: Child PID = 29156 has been stopped by a signal, signo = 19
My PID = 29153: Child PID = 29155 has been stopped by a signal, signo = 19
My PID = 29153: Child PID = 29154 has been stopped by a signal, signo = 19
Continuing the first child with PID=29154!
Initializing the clock for the first child!

This is the Shell. Welcome.

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

Printing a list of tasks currently scheduled!
Current running task pid=29154 with id=0.
Task prog: id=1 and pid=29155.
Task prog: id=2 and pid=29156.
Task shell: id=0 and pid=29154.
Shell> ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=29154
My PID = 29153: Child PID = 29154 has been stopped by a signal, signo = 19
Procedure with PID=29155, about to continue!
Initializing the clock!
I am Child_1,PID=29155
Imma 'bout to replace my code with the executable prog...
prog: Starting, NMSG = 20, delay = 99
prog[29155]: This is message 0
prog[29155]: This is message 1
prog[29155]: This is message 2
prog[29155]: This is message 3
prog[29155]: This is message 4
prog[29155]: This is message 5
prog[29155]: This is message 6
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=29155
My PID = 29153: Child PID = 29155 has been stopped by a signal, signo = 19
Procedure with PID=29156, about to continue!
Initializing the clock!
I am Child_2,PID=29156
Imma 'bout to replace my code with the executable prog...
prog: Starting, NMSG = 20, delay = 81
prog[29156]: This is message 0
prog[29156]: This is message 1
prog[29156]: This is message 2
prog[29156]: This is message 3
prog[29156]: This is message 4
prog[29156]: This is message 5
prog[29156]: This is message 6
prog[29156]: This is message 7
prog[29156]: This is message 8
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=29156
My PID = 29153: Child PID = 29156 has been stopped by a signal, signo = 19
Procedure with PID=29154, about to continue!
Initializing the clock!
e progALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=29154
My PID = 29153: Child PID = 29154 has been stopped by a signal, signo = 19
Procedure with PID=29155, about to continue!
Initializing the clock!

prog[29155]: This is message 7
prog[29155]: This is message 8
prog[29155]: This is message 9
prog[29155]: This is message 10
prog[29155]: This is message 11
prog[29155]: This is message 12
prog[29155]: This is message 13
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=29155
My PID = 29153: Child PID = 29155 has been stopped by a signal, signo = 19
Procedure with PID=29156, about to continue!
Initializing the clock!
prog[29156]: This is message 9
prog[29156]: This is message 10
prog[29156]: This is message 11
prog[29156]: This is message 12
prog[29156]: This is message 13
prog[29156]: This is message 14
prog[29156]: This is message 15
prog[29156]: This is message 16
prog[29156]: This is message 17
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=29156
My PID = 29153: Child PID = 29156 has been stopped by a signal, signo = 19
Procedure with PID=29154, about to continue!
```





Για τις ασκήσεις 1 και 2 χρησιμοποιήθηκε το δοσμένο Makefile.

Άσκηση 3:

Ο πηγαίος κώδικας του αρχείου part3l.c όπως τροποποιήθηκε φαίνεται παρακάτω:

```
#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 4                /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's name */
#define SHELL_EXECUTABLE_NAME "shell" /* executable for shell */


int nproc=0; //number of procedures for scheduling

struct process_control_block{     //Process Control Block Struct (node)
        int id;
        int prio; //priority: 1->HIGH, 0->LOW
        pid_t pid;
        char proc_name[TASK_NAME_SZ];
        struct process_control_block *next;  //queue of processes
};

typedef struct process_control_block PCB;

PCB *head=NULL;
PCB *tail=NULL;
PCB *high_1=NULL;
PCB *high_2=NULL;

void add_proc(int id,pid_t pid, char proc_name[TASK_NAME_SZ]){ //function to
add the new proc, enqueue
```

```
        PCB *temp=(PCB*)malloc(sizeof(PCB));
        temp->id=id;
        temp->prio=0;
        temp->pid=pid;
        strcpy(temp->proc_name,proc_name);
        temp->next=NULL;

        if(head==NULL){
                head=temp;
                tail=temp;
                tail->next=head;
        }
        else{
                if (high_2==NULL){      //if no HIGHs
                        tail->next=temp;
                        tail=temp;
                        tail->next=head;
                }
                else{                   //there are HIGHs
                        temp->next=high_2->next;
                        high_2->next=temp;
                }
        }
}

//_____________SHELL FUNCTIONS_________________

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

        printf("\nPrinting a list of tasks currently scheduled!\n");
        //assert(0 && "Please fill me!");

        PCB *temp=head;
        printf("Current running task pid=%ld with id=%d.\n",head->pid, head-
>id);       //scheduler

        //printf("Task %s: id=%d and pid=%ld.\n",temp->proc_name,temp-
>id,temp->pid);

        do{
```

```
                temp=temp->next;
                printf("Task %s with priority %d: id=%d and pid=%ld.\n",temp-
>proc_name,temp->prio,temp->id,temp->pid);

        }while(temp!=head);
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id)
{
        //assert(0 && "Please fill me!");

        PCB *temp_h=head;
        PCB *temp_t;

        int found=0;

        do{

                temp_t=temp_h;
                temp_h=temp_h->next;
                if(temp_h->id==id){
                        found=1;
                        if(strcmp(temp_h->proc_name,SHELL_EXECUTABLE_NAME)==0)
{ //if it is a shell
                                printf("Press q to kill the shell!\n");
                                break;
                        }

                        //killing
                        printf("Killing process id=%d...\n",id);
                        kill(temp_h->pid,SIGKILL);


                        //taking the process out of the queue
                        if(temp_h==high_1){
                                if(temp_h==head) //head and high_1
                                        head=head->next;
                                else if(temp_h==tail)
                                        tail=temp_t;
```

```
                                high_1=high_1->next;
                        }
                        else if(temp_h==high_2){
                                if(temp_h==head){
                                        if(high_1==high_2){ //den ehw alla
                                                head=head->next;
                                                high_1=NULL;
                                                high_2=NULL;
                                        }
                                        else{
                                                high_2=temp_t;
                                                head=high_1;
                                        }
                                }
                                else
                                        high_2=temp_t;

                        }
                        else{ //an ehw mono LOW
                                if(temp_h==head)
                                        head=head->next;
                                else if(temp_h==tail)
                                        tail=temp_t;
                        }

                        printf("Taking process %s with id=%d and pid=%ld out
of the queue.\n",temp_h->proc_name,temp_h->id,temp_h->pid);
                        temp_t->next=temp_h->next;
                        temp_h->next=NULL;
                        free(temp_h);
                        break;
                }


        }
        while(temp_h!=head);

        if(found==0) printf("No task with the specified id was found!\n");

        //return -ENOSYS;
}
```

```c
/* Create a new task.  */
static void
sched_create_task(char *executable)
{
        //assert(0 && "Please fill me!");
        nproc++;
        pid_t p=fork();
        if (p<0){ //if there is an error in fork()
                perror("fork");
                exit(1);
        }
        if(p==0){
                raise(SIGSTOP);
                char *newargv[]={executable,NULL,NULL,NULL};
                char *newenviron[]={NULL};
                printf("I am Shell_proc_%d,PID=%ld\n",nproc,(long)getpid());
                printf("Imma 'bout to replace my code with the executable
%s...\n",executable);
                sleep(1);
                execve(executable,newargv,newenviron);
                perror("execve");
                exit(1);
        }
        add_proc(nproc,p,executable);   //adding the new proc in the
scheduling queue

}

//change the priority of a procedure-> HIGH

static void
sched_high_task(int id){
        PCB *temp_h=head;
        PCB *temp_t;

        int found=0;

        do{
                temp_t=temp_h;
                temp_h=temp_h->next;

                if(temp_h->id==id){
```

```
                        found=1;

                        printf("Proc with prio %d: id=%d, pid=%ld is getting
high!\n",temp_h->prio,temp_h->id,temp_h->pid);

                        temp_h->prio=1;

                        if(high_2==NULL){
                                high_1=temp_h;
                                high_2=temp_h;
                                //head=high_1;
                                //while(tail->next!=head)
                                        //tail=tail->next;
                        }
                        else{
                                temp_t->next=temp_h->next;
                                temp_h->next=high_2->next;
                                high_2->next=temp_h;
                                high_2=temp_h;

                        }
                }

        }while(temp_h!=head);

        if(found==0) printf("No task with the specified id was found!\n");

}

//change the priority of a procedure-> LOW
static void
sched_low_task(int id){
        PCB *temp_h=head;
        PCB *temp_t;

        int found=0;

        do{
                temp_t=temp_h;
                temp_h=temp_h->next;

                if(temp_h->id==id){
```

```
                        found=1;

                        printf("Proc with prio %d: id=%d, pid=%ld is getting
low low low low!\n",temp_h->prio,temp_h->id,temp_h->pid);

                        temp_h->prio=0;

                        if(high_2==NULL) //no HIGHs
                                break;

                        else{
                                if(temp_h==high_2){
                                        if(high_2==high_1){
                                                high_1=NULL;
                                                high_2=NULL;
                                        }
                                        else
                                                high_2=temp_t;
                                }
                                else if(temp_h==high_1){
                                        //high_1!=high_2
                                        temp_t->next=temp_h->next;
                                        high_1=temp_h->next;

                                        temp_h->next=high_2->next;
                                        high_2->next=temp_h;
                                }

                                else{
                                        temp_t->next=temp_h->next;
                                        temp_h->next=high_2->next;
                                        high_2->next=temp_h;

                                }
                }
        }

       }while(temp_h!=head);

       if(found==0) printf("No task with the specified id was found!\n");


}
```

```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:     //h
                        sched_high_task(rq->task_arg);
                        return 0;

                case REQ_LOW_TASK:     //l
                        sched_low_task(rq->task_arg);
                        return 0;

                default:
                        return -ENOSYS;
        }
}

//____________________HANDLERS_____________________

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        //assert(0 && "Please fill me!");
        printf("ALARM!Time %dsec has passed!\n",SCHED_TQ_SEC);
        printf("Sending SIGSTOP to procedure with PID=%ld\n",head->pid);
        if(kill(head->pid,SIGSTOP)<0){
```

```
                perror("kill");
                exit(1);
        }
}


/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        //assert(0 && "Please fill me!");
        for(;;){ //infinite loop
                int *status;
                pid_t p=waitpid(-1,&status,WUNTRACED | WNOHANG);
                if(p<0){
                        perror("waitpid");
                        exit(1);
                }
                else if(p==0)   //WUNTRACED flag is specified, no child
changes state
                break;

                explain_wait_status(p,status); //if a child changed state

                if(WIFEXITED(status)){   //child terminated normally
                        //printf("Procedure with PID=%ld, died!\n",head->pid);

                        if(head==high_2){ //The last high terminated (ehw ki
alla / den ehw allo)

                                if(high_1!=high_2){ //ehw ki allo
                                        head=high_1;
                                        tail->next=high_2->next;
                                        free(high_2);
                                        high_2=tail;
                                        while(tail->next!=head)
                                                tail=tail->next;
                                }
                                else if (high_1==high_2){ //den ehw allo
                                        head=head->next;
                                        free(tail->next);
```

```
                                        tail->next=head;
                                        high_1=NULL;
                                        high_2=NULL;
                                }
                        }

                        else if(head==high_1){
                                //head!=high_2
                                head=head->next;
                                free(tail->next);

                                tail->next=head;
                                high_1=head;
                        }

                        else{

                                head=head->next;
                                free(tail->next);
                                tail->next=head;

                        }

                }

                if(WIFSIGNALED(status)) //child killed by a signal (running
process or not??)
                        break;

                if(WIFSTOPPED(status)){ //child stopped
                        //printf("Procedure with PID=%ld, stopped!\n",head-
>pid);
                        if(head==high_2){ //last HIGH stopped, restart with
the HIGHs
                                head=high_1;
                                while(tail->next!=head)
                                        tail=tail->next;
                        }
                        else if(head->prio==0 && high_1!=NULL && high_2!=NULL)
{
                                printf("\nHEYYYYYYYY!\n");
                                head=high_1;
                                while(tail->next!=head)
```

```
                                        tail=tail->next;
                        }

                        else{
                                head=head->next;
                                tail=tail->next;
                        }
                }

                printf("Procedure with PID=%ld, about to continue!\n",head-
>pid);

                //initializing the alarm again
                printf("Initializing the clock!\n");
                alarm(SCHED_TQ_SEC);            //set the alarm

                kill(head->pid,SIGCONT);        //starting the next procedure
        }
}


/* 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 */

        //add the shell in the procedure queue once created
        //nproc++;
        add_proc(nproc,p,SHELL_EXECUTABLE_NAME); //shell also initialized to
LOW

        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 i,nproc,nproc2;
        pid_t p;

        /* 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);
        /* TODO: 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 */
        nproc2=argc-1;

        for(i=1; i<=nproc2; i++){ //the first added process in the queue was
the shell

                p=fork(); //creation of new procedure

                if (p<0){ //if there is an error in fork()
                        perror("fork");
```

```
                        exit(1);
                }
                if(p==0){ //code of the child-procedure

                        raise(SIGSTOP);
                        char *newargv[]={argv[i],NULL,NULL,NULL};
                        char *newenviron[]={NULL};
                        printf("I am Child_%d,PID=%ld\n",i,(long)getpid());
                        printf("Imma 'bout to replace my code with the
executable %s...\n",argv[i]);
                        sleep(1);
                        execve(argv[i],newargv,newenviron);
                        perror("execve");
                        exit(1);
                }
                //scheduler-father code, process in the queue, p is the PID of
the child process
                add_proc(i,p,argv[i]);

        }
        nproc++; //shell was added
        tail->next=head; //make circular queue

        /* 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);
        }
        else{   //waking up the first-created child
                printf("Continuing the first child with PID=%ld!\n",head-
>pid);
                printf("Initializing the clock for the first child!\n");

                alarm(SCHED_TQ_SEC);            //set the alarm
                kill(head->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("Shell gone!\n");

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

Για αυτή την άσκηση χρειάζεται να τροποποιήσουμε και το Makefile, προσθέτοντας τις εντολές που αφορούν το part3. Ο πηγαίος κώδικας του φαίνεται παρακάτω.

```
#
# Makefile
#
# Operating Systems, Exercise 4
#

CC = gcc
#CFLAGS = -Wall -g
CFLAGS = -Wall -O2 -g

all: scheduler scheduler-shell shell prog execve-example strace-test
sigchld-example part3

scheduler: scheduler.o proc-common.o
        $(CC) -o scheduler scheduler.o proc-common.o

scheduler-shell: scheduler-shell.o proc-common.o
        $(CC) -o scheduler-shell scheduler-shell.o proc-common.o

part3: part3.o proc-common.o
        $(CC) -o part3 part3.o proc-common.o

shell: shell.o proc-common.o
        $(CC) -o shell shell.o proc-common.o
```

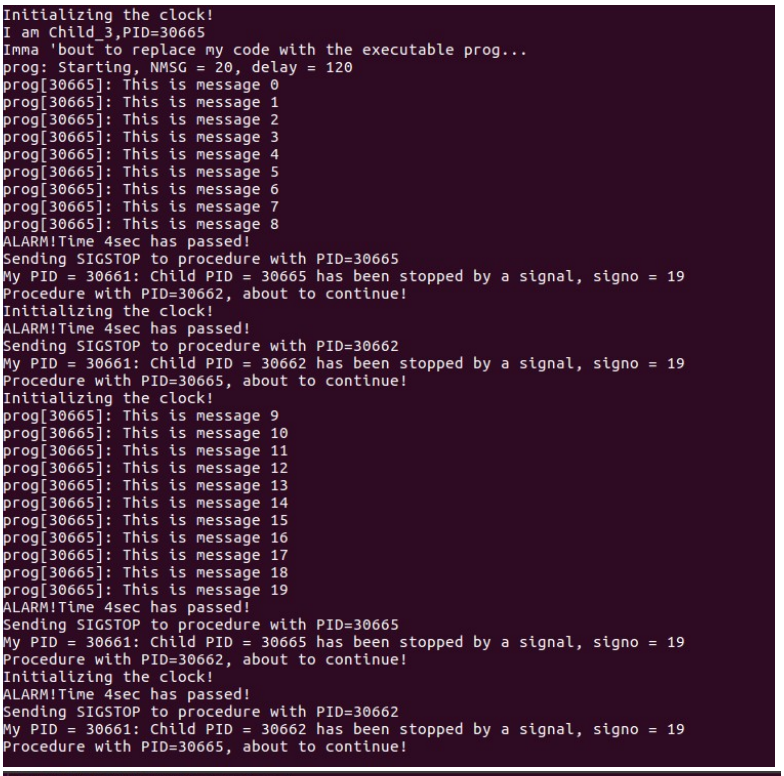
```
prog: prog.o proc-common.o
	$(CC) -o prog prog.o proc-common.o

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) -g -o scheduler-shell.o -c scheduler-shell.c

part3.o: part3.c proc-common.h request.h
	$(CC) $(CFLAGS) -g -o part3.o -c part3.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
```

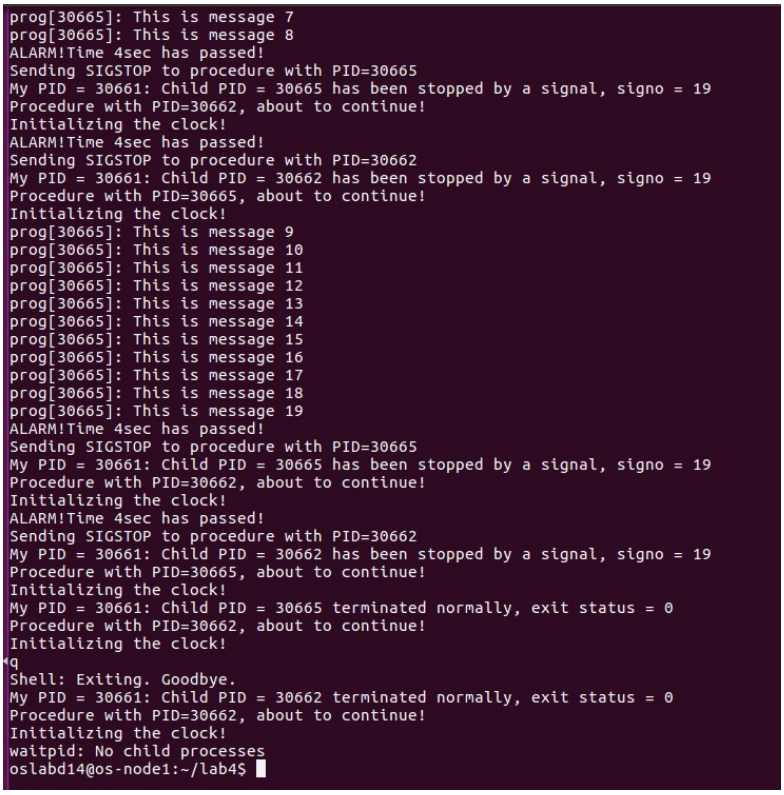
Ένα ενδεικτικό output του παραπάνω κώδικα φαίνεται εδώ:

```
oslabd14@os-node1:~/lab4$ ./part3 prog prog prog
My PID = 30661: Child PID = 30662 has been stopped by a signal, signo = 19
My PID = 30661: Child PID = 30663 has been stopped by a signal, signo = 19
My PID = 30661: Child PID = 30664 has been stopped by a signal, signo = 19
My PID = 30661: Child PID = 30665 has been stopped by a signal, signo = 19
Continuing the first child with PID=30662!
Initializing the clock for the first child!

This is the Shell. Welcome.

Shell> h 0
Shell: issuing request...
Shell: receiving request return value...
Proc with prio 0: id=0, pid=30662 is getting high!
Shell> h 1
Shell: issuing request...
Shell: receiving request return value...
Proc with prio 0: id=1, pid=30663 is getting high!
Shell> h 2
Shell: issuing request...
Shell: receiving request return value...
Proc with prio 0: id=2, pid=30664 is getting high!
Shell> ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30662
My PID = 30661: Child PID = 30662 has been stopped by a signal, signo = 19
Procedure with PID=30663, about to continue!
Initializing the clock!
I am Child_1,PID=30663
Imma 'bout to replace my code with the executable prog...
prog: Starting, NMSG = 20, delay = 92
prog[30663]: This is message 0
prog[30663]: This is message 1
prog[30663]: This is message 2
prog[30663]: This is message 3
prog[30663]: This is message 4
prog[30663]: This is message 5
prog[30663]: This is message 6
prog[30663]: This is message 7
prog[30663]: This is message 8
prog[30663]: This is message 9
```

```
prog[30663]: This is message 10
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30663
My PID = 30661: Child PID = 30663 has been stopped by a signal, signo = 19
Procedure with PID=30664, about to continue!
Initializing the clock!
I am Child_2,PID=30664
Imma 'bout to replace my code with the executable prog...
l prog: Starting, NMSG = 20, delay = 73
prog[30664]: This is message 0
prog[30664]: This is message 1
prog[30664]: This is message 2
prog[30664]: This is message 3
prog[30664]: This is message 4
prog[30664]: This is message 5
prog[30664]: This is message 6
prog[30664]: This is message 7
prog[30664]: This is message 8
prog[30664]: This is message 9
prog[30664]: This is message 10
prog[30664]: This is message 11
prog[30664]: This is message 12
prog[30664]: This is message 13
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30664
My PID = 30661: Child PID = 30664 has been stopped by a signal, signo = 19
Procedure with PID=30662, about to continue!
Initializing the clock!
 1
Shell: issuing request...
Shell: receiving request return value...
Proc with prio 1: id=1, pid=30663 is getting low low low low!
Shell> ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30662
My PID = 30661: Child PID = 30662 has been stopped by a signal, signo = 19
Procedure with PID=30664, about to continue!
Initializing the clock!
prog[30664]: This is message 14
prog[30664]: This is message 15
prog[30664]: This is message 16
prog[30664]: This is message 17
prog[30664]: This is message 18
prog[30664]: This is message 19
```

```
My PID = 30661: Child PID = 30664 terminated normally, exit status = 0
Procedure with PID=30662, about to continue!
Initializing the clock!
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30662
My PID = 30661: Child PID = 30662 has been stopped by a signal, signo = 19
Procedure with PID=30662, about to continue!
Initializing the clock!
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30662
My PID = 30661: Child PID = 30662 has been stopped by a signal, signo = 19
Procedure with PID=30662, about to continue!
Initializing the clock!
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30662
My PID = 30661: Child PID = 30662 has been stopped by a signal, signo = 19
Procedure with PID=30662, about to continue!
Initializing the clock!
ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30662
My PID = 30661: Child PID = 30662 has been stopped by a signal, signo = 19
Procedure with PID=30662, about to continue!
Initializing the clock!
l 0
Shell: issuing request...
Shell: receiving request return value...
Proc with prio 1: id=0, pid=30662 is getting low low low low!
Shell> ALARM!Time 4sec has passed!
Sending SIGSTOP to procedure with PID=30662
My PID = 30661: Child PID = 30662 has been stopped by a signal, signo = 19
Procedure with PID=30663, about to continue!
Initializing the clock!
prog[30663]: This is message 11
prog[30663]: This is message 12
prog[30663]: This is message 13
prog[30663]: This is message 14
prog[30663]: This is message 15
prog[30663]: This is message 16
prog[30663]: This is message 17
prog[30663]: This is message 18
prog[30663]: This is message 19
My PID = 30661: Child PID = 30663 terminated normally, exit status = 0
Procedure with PID=30665, about to continue!
```





Επισημαίνεται πως ο αριθμός των τυπωμένων μηνυμάτων από την prog έχει οριστή στα 20, ενώ ο χρόνος για το alarm στα 4sec.

3 Ερωτήσεις Αναφοράς

3.1 Άσκηση 1.1

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.

Όταν χειριζόμαστε ένα σήμα, τα υπόλοιπα βρίσκονται μπλοκάρονται.
Στην δικιά μας υλοποίηση, η συνάρτηση install_signal_handlers περιέχει το εξής κομμάτι κώδικα:

```
	sigemptyset(&sigset);
	sigaddset(&sigset, SIGCHLD);
	sigaddset(&sigset, SIGALRM);

	sa.sa_mask = sigset;    //signals that should be blocked during the
execution of the handler
```

Έτσι εξασφαλίζουμε πως όταν τρέχει ο handler μπλοκάρονται και τα δύο παραπάνω σήματα (μπλοκάρεται το sigset που τα περιέχει) εφόσον ο κάθε handler εκτελείται ατομικά ώστε να ολοκληρώσει την υλοποίησή του πριν κληθεί ο ίδιος ή ο άλλος για τη διαχείριση του επόμενου σήματος.

Ο χρονοδρομολογητής που υλοποιήσαμε βρίσκεται σε χώρο χρήστη. Αντίθετα, οι πραγματικοί δρομολογητές βρίσκονται σε χώρο πυρήνα, οπότε μέχρι να επιστραφεί ο χειρισμός στον χώρο χρήστη, τα σήματα αναστέλλονται εξ'ορισμού.

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

Το σήμα SIGCHLD λαμβάνεται όταν γίνει κάποια αλλαγή σε ένα οποιοδήποτε σήμα.
Συνηθέστερα αυτό γίνεται στη διεργασία η οποία μόλις έτρεχε, είτε επειδή τελειώσε είτε επειδή τελείωσε το κβάντο χρόνου στο οποίο μπορούσε να τρέξει και σταμάτησε λόγω σήματος SIGSTOP που του στείλαμε.
Αν λόγο εξωτερικού παράγοντα τερματιστεί μια άλλη διεργασία παιδί αναπάντεχα, στον clhd handler θα ανιχνευθεί η αλλαγή και, θα συνεχίσει η διεργασία η οποία είναι η επόμενη από αυτή που έτρεχε ήδη (δηλαδή η επόμενη από αυτή που έδειχνε ο head στην λίστα μας).

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 κύριο μειονέκτημα που θα είχαμε είναι ότι αν μια διεργασία τελείωνε πριν εκπνεύσει το κβάντο χρόνου που έχει οριστεί, δεν θα ξεκινούσε αμέσως η επόμενη αλλά θα περιμέναμε να τελειώσει και θα είχαμε άχρηστο χρόνο κατά τον οποίο δεν θα έτρεχε καμία εργασία. Παράλληλα, έχοντας 2 σήματα έχουμε και καλύτερο έλεγχο εφόσον η παραλαβή του SIGCHLD σήματος επιβεβαιώνει πως το σήμα SIGSTOP στάλθηκε και λήφθηκε.

3.2 Άσκηση 1.2

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;

Στην εντολή 'p' ως τρέχουσα διεργασία τυπώνεται πάντα ο φλοιός. Αυτό είναι εύλογο δεδομένο ότι ο ίδιος ο φλοιός εκτελεί την εντολή 'p' οπότε μόνο όταν εκτελείται αυτός μπορούμε να την καλέσουμε.

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

Οι κλήσεις signals_disable() και signals_enable() γύρω από την εντολή `ret = process_request(&rq);` είναι αναγκαίες διότι δεν είναι επιθυμητό να στέλνονται σήματα που μπορεί να μεταβάλλουν την κατάσταση της ουράς ενώ εκτελούνται τα αιτήματα του φλοιού, αφού και ο φλοιός τροποποιεί την ουρά των διεργασιών εισάγοντας νέα διεργασία ή διαγράφοντας μία υπάρχουσα.
Για παράδειγμα, αν την ώρα που στέλναμε ένα σήμα ο φλοιός διέγραφε μια διεργασία μπορεί ως αποτέλεσμα να είχαμε σφάλματα όπως η διαγραφή λάθος εργασίας ή η λάθος ανάθεση των pointers στην λίστα.
Έτσι οι παραπάνω εντολές "λειτουργούν ως lock", διασφαλίζοντας την αποφυγή όμοιων σφαλμάτων

3.3 Άσκηση 1.3

1. Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας.

Λιμοκτονία ονομάζεται το φαινόμενο κατά το οποίο μία διεργασία δεν αποκτά πρόσβαση στους απαραίτητους πόρους (ΚΜΕ, συσκευές Ε/Ε) προκειμένου να προχωρήσει και να ολοκληρώσει το έργο της με αποτέλεσμα να μην τερματίζεται. Αυτό οφείλεται κυρίως στον αλγόριθμο χρονοδρομολόγησης που χρησιμοποιείται.

Στη συγκεκριμένη άσκηση, λιμοκτονία δημιουργείται για παράδειγμα αν ο φλοιός γίνει διεργασία υψηλής προτεραιότητας, με την εντολή h 0 (shell_id=0) έκκλησης στο φλοιό. Αυτό έχει ως

αποτέλεσμα, αν δεν τερματίσει ο φλοιός (με q) ή αν δεν ξαναγίνει διεργασία χαμηλής προτεραιότητας (με l 0), οι διεργασίες χαμηλής προτεραιότητας να μην εκτελεστούν ποτέ και να λιμοκτονήσουν. Αυτό, διότι ο αλγόριθμος χρονοδρομολόγησής μας δε λαμβάνει υπόψει τη γήρανση στον καθορισμό της προτεραιότητας μιας διεργασίας.