Λειτουργικά Συστήματα Υπολογιστών
4η Εργαστηριακή Άσκηση

Συνεργάτες:

- **Καλλάς Κωσταντίνος Α.Μ:03112057**
- **Τζίνης Ευθύμιος Α.Μ:03112007**

Οι κώδικες παρατίθενται στο τέλος της αναφοράς για καλύτερη παρουσίαση.

Σημαντική Σημείωση για την Υλοποίηση:

Αποφασίσαμε λόγω της απλότητας της άσκησης και λόγω των λιγοστών διεργασιών να φτιάξουμε τον χρονοδρομολογητή όχι με τον ενδεδειγμένο τρόπο λιστών αλλά με δυναμική δέσμευση πινάκων. Αυτό το κάναμε κυρίως διότι ήταν αρκετά εύκολο με βάση του ότι κάθε φορά κρατάμε έναν πίνακα που περιέχει όλα τα pid's των διεργασιών έχουμε απόλυτο έλεγχο όλων των ενεργών διεργασιών και μη. Διαθέτουμε άλλον έναν πίνακα που περιέχει είτε ένα ταυτοτικό ID που δίνουμε εμείς σε μία διεργασία για παράδειγμα my_id[i]==i, είτε my_id[i]==-1 όταν αυτή η διεργασία έχει πεθάνει για οποιονδήποτε λόγο. Προφανώς μπορούμε κάθε φορά να κάνουμε realloc() τον πίνακα και με τα pid's όσο και με τα ID's για να εισάγουμε μία καιρνούρια διεργασία. Όταν μια διεργασία πεθαίνει δεν κάνουμε τίποτε άλλο από να αλλάξουμε το my_id[] της σε -1 προκειμένου να μην της ξαναδώσουμε τον έλεγχο. Η υλοποίησή μας με πίνακες θα μπορούσε να βελτιωθεί σημαντικά με την δημιουργία ενός αλγορίθμου σάρωσης και διαγραφής των πλεοναζόντων στοιχείων του πίνακα. Προφανώς η υλοποίησή μας δεν είναι η ενδεδειγμένη αλλά προσπαθήσαμε να το υλοποιήσουμε λίγο διαφορετικά την άσκηση για να δοκιμάσουμε διαφορετικές προγραμματιστικές τεχνικές σε ένα πρόβλημα που ξέρουμε την ενδεδειγμένη δομή που πρέπει να επιλεγεί. Με αυτόν τον τρόπο παρατηρούμε το μεγάλο εύρος επιλογών που μας δίνεται στην υλοποίηση του χρονοδρομολογητή μέσω διαφορετικών δομών που για λίγες διεργασίες μπορεί να έχει και καλύτερη επίδοση από έναν scheduler υλοποιημένο με λίστες ειδικά όταν θέλουμε να βρούμε μία συγκεκριμένη διεργασία που θέλουμε να σκοτώσουμε.

Συγκεκριμένα στην 3η άσκηση χρησιμοποιήσαμε πάλι την υλοποίηση με πίνακες όπως στις προηγούμενες για χάριν ευκολίας. Ουσιαστικά δημιουργούμε και ένα 3ο πίνακα high_ids ο οποίος είναι ταυτοτικός για τις διεργασίες που βρίσκονται σε high priority και -1 για όλες τις υπόλοιπες. Έτσι ουσιαστικά η υλοποίηση παραμένει η ίδια απλώς αναζητούμε πρώτα τις διεργασίες στον high_ids πίνακα και μετά στον low.

Ερωτήσεις:

3.1.1. 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;

Στην υλοποίησή μας εξαιτίας της συνάρτησης install_signal_handlers() έχουμε εφαρμόσει μία μάσκα για τα δύο σήματα που χρησιμοποιούμε SIGCHLD και SIGALRM με αποτέλεσμα όταν έρχεται το ένα σήμα ενώ ο έλεγχος βρίσκεται στην ρουτίνα εξυπηρέτησης του άλλου σήματος τότε αυτό που κατέφτασε τελευταίο δεν εξυπηρετείται προσωρινά. Στον κανονικό χρονοδρομολογητή χώρου πυρήνα όταν ένα δεύτερο σήμα έρθει την ώρα που εξυπηρετούμε το πρώτο τότε αυτό δεν θα εξυπηρετηθεί μέχρι ο έλεγχος να επιστρέψει από kernel mode σε user mode.

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

Ο χρονοδρομολογητής λαμβάνει σήμα **sigchld** στην περίπτωση που ένα παιδί τερματίσει απο φυσιολογικά αίτια είτε στην περίπτωση που λάβει **sigstop** από την χρονοδρομολογητή επειδή τέλειωσε το κβάντο χρόνου του. Και στις δύο περιπτώσεις το σήμα αναφέρεται πάντα στη διεργασία που τρέχει την παρούσα στιγμή. Αυτό βέβαια αλλάζει αν λαμβάνουμε υπόψη την περίπτωση λήψης σημάτων από εξωτερικό παράγοντα όπως από το bash είτε από τη διεργασία shell που χρονοδρομολογούμε στα υπόλοιπα ερωτήματα. Για να επιτρέψουμε την απρόσκοπτη λειτουργία του χρονοδρομολογητή μας και στην περίπτωση λήψης σημάτων από εξωτερικούς παράγοντες χειριζόμαστε τέτοια σήματα μέσα στην **sigchld** όπως φαίνεται παρακάτω.

```
if(WIFSIGNALED(status) && WTERMSIG(status) == SIGKILL)
{
for(i=0;i<nproc;i++)
{
        if(child_pids[i] == pid)
        {
                if(i!=0) printf("Pethane o prog: %d sthn xestra\n", my_id[i]);
                else printf("Pethane to Shel MAgka mou :(\n");
                my_id[i] = -1;
                if(--active_children == 0)
                {
                        printf("Ola phgan popa\n");
                        exit(0);
                }
                if(counter == i)
                {
                        counter = (++counter) % nproc;
                        while(my_id[counter] ==  -1)
                        {
                                counter = (++counter)%nproc;
                        }
                        alarm(0);
                        alarm(SCHED_TQ_SEC);
                        kill(child_pids[counter],SIGCONT);
                        if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
                        else printf("The number: %d came back from the toilet queue\n", counter);
                }
                break;
        }

}
}
```

3.1.3. 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;

Αυτό δεν θα ήταν εφικτό διότι ο χρονονοδρομολογητής πρέπει να έχει πλήρη γνώση για την κατάσταση των υπολοίπων διεργασιών. Αν δεν υπήρχε το σήμα SIGCHLD τότε ο χρονονοδρομολογητής δεν θα ήταν ποτέ πραγματικά σίγουρος αν μία διεργασία έχει σταματήσει την λειτουργία της. Με αυτόν τον τρόπο θα μπορούσε να δημιουργηθεί μία κατάσταση που παραπάνω από μία διεργασίες θα προσπαθούσαν να τρέξουν την ίδια στιγμή.

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

Προφανώς και όταν δίνουμε την εντολή 'p' στον scheduler τότε αυτός μας δείχνει τις διεργασίες που είναι ενεργές εκείνη την στιγμή και μέσα σε αυτές είναι πάντα ο scheduler αφού μόνο μέσα από την διεργασία του χρονοδρομολογητή μπορούμε να δώσουμε μία τέτοια εντολή. Πρακτικά λοιπόν δεν μπορεί να πραγματοποιηθεί ποτέ αυτό το σενάριο διότι ο χειρισμός των εντολών γίνεται μέσα στο scheduler σε ένα critical section που περιλαμβάνει signals_enable() και signals_disable().

3.2.2. 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.

Είναι αναγκαίο να συμπεριλάβουμε τις δύο αυτές κλήσεις έτσι ώστε να είμαστε σίγουροι ότι κατά τη διάρκεια εκτέλεσης της διεργασίας shell η οποία επηρεάζει τη λειτουργία του χρονοδρομολογητή δεν θα λάβουμε κάποιο σήμα που θα την διάκοψει. Αυτές οι δύο κλήσεις λειτουργούν σαν κάποιου είδους κλειδώματος που αποτρέπει πάνω απο μία διεργασία να τρέχει στο “critical section” του προγράμματος μας.

3.3.1. Περιγράψτε ένα σενάριο δημιουργίας λιμοκτονίας.

Ένα σενάριο πιθανής λιμοκτονίας θα ήταν στην περίπτωση που ανεβάσουμε την προτεραιότητα σε μία διεργασία που δεν σταματάει απο φυσικά αίτια (πχ εκτελεί ένα infinite loop) και να κατεβάσουμε την προτεραιότητα του shell. Έτσι η διεργασία που είναι high priority δεν θα σταματήσει να εκτελείται ποτέ και δεν θα ξαναεκτελεστούν ποτέ όλες οι υπόλοιποες διεργασίες. Αυτό θα μπορούσε να επιλυθεί αν κάθε κάποια ώρα κατεβάζουμε όλες τις διεργασίες από high σε low. Αυτό ουσιαστικά είναι ένας μηχανισμός γήρανσης.

4.1. Περιγράψτε τον μηχανισμό επικοινωνίας μεταξύ του φλοιού και του χρονοδρομολογητή στην Άσκηση 1.2.

Ο χρονοδρομολογητής (πατέρας) επικοινωνεί με τον φλοιό (παιδί) μέσω ενός pipe. Συγκεκριμένα ο πατέρας στέλνει μέσω του pipe στο φλοιό τη συμβολοσειρά που πληκτρολογήθηκε απο το χρήστη, ο φλοιός την parse – άρει και επιστρέφει στον πατέρα κωδικοποιημένα ποιά εντολή να εκτελέσει. Ο πατέρας ύστερα εκτελέι την εντολή.

4.2. Προδιαγράψτε και σκιαγραφήστε υλοποίηση μηχανισμού γήρανσης για την απο- φυγή λιμοκτονίας στην Άσκηση 1.3.

Όπως περιγράφηκε παραπάνω στην ερώτηση 3.3.1 θα μπορούσαν όλες οι διεργασίες που βρίσκονται σε high priority να αλλάζουν σε Low μετά από κάποιο καθορισμένο χρονικό κβάντο.

4.3. Έστω χρονοδρομολογητής που χρησιμοποιεί πολιτική χρονοδρομολόγησης αντί- στοιχη με αυτή της Άσκησης 1.3, και υποστηρίζει τρεις κλάσεις προτεραιότητας: HIGH, MEDIUM και LOW. Έστω, επίσης, ότι ο χρονοδρομολογητής αναλαμβάνει να αναστέλλει την εκτέλεση διεργασιών, όταν αυτές πρέπει να περιμένουν σε κά- ποιο σημαφόρο. Έστω τρεις διεργασίες Η, Μ, L με προτεραιότητες HIGH, MEDIUM, LOW, αντίστοιχα.

Ουσιαστικά ο σημαφόρος δεν θα ελευθερωθεί ποτέ επειδή η Η θα κολλήσει στο s.wait(). Αυτό θα μπορούσε να λυθεί με ένα μηχανισμό γήρανσης που κατεβάζει ένα επίπεδο προτεραιότητας τις διεργασίες μετά απο προκαθορισμένο χρόνο.Έτσι μετά απο τον χρόνο γήρανσης η Η θα πέσει στο medium priority και η Μ θα πέσει στο low. Μετά από άλλο ένα χρόνο γήρανσης η Η θα πέσει στο low και έτσι θα μπορέσει να εκτελεστεί η L που μετά απο κάποια ώρα θα απελευθερώσει τον σημαφόρο και θα μπορέσει η Η να ξεκολλήσει από το s.wait().

Κώδικες:

Άσκηση 1

```
#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 */

pid_t *child_pids;
int counter=0;
int nproc;
int *my_id;
int active_children;
/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
        if(active_children>1){
        printf("The number: %d stopped to take a piss\n", counter);
        kill(child_pids[counter++],SIGSTOP);
        }
}


/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        pid_t pid;
        int status, i;
        i = 0;
        while((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) != -1)
        {
                if(pid == 0)
                {
                        return;
                }
                if (WIFSTOPPED(status)){
                        counter=counter % nproc;
                        alarm(SCHED_TQ_SEC);    //for 2 sec
                        while(my_id[counter] ==  -1)
```

```
                {
                        counter = (++counter)%nproc;
                }
                printf("The number: %d came back from the toilet queue\n", counter);
                kill(child_pids[counter],SIGCONT);
        }
        else if(WIFEXITED(status) )
        {
        for(i=0;i<nproc;i++)
        {
                if(child_pids[i] == pid)
                {
                        printf("Pethane o prog[%d] sthn xestra\n", pid);
                        my_id[i] = -1;
                        if(--active_children == 0)
                        {
                                printf("Ola phgan popa\n");
                                exit(0);
                        }




                                counter = (++counter) % nproc;
                                while(my_id[counter] ==  -1)
                                {
                                        counter = (++counter)%nproc;
                                }
                                alarm(0);
                                alarm(SCHED_TQ_SEC);
                                kill(child_pids[counter],SIGCONT);
                                printf("The number: %d came back from the toilet queue\n",
counter);

                        break;
                }

        }
        }
        }
}

/* 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;
	/*
	 * For each of argv[1] to argv[argc - 1],
	 * create a new child process, add it to the process list.
	 */
	char *newargv[] = { NULL, "hello", "world", NULL };
	char *newenviron[] = { NULL };
	char path[30];
	nproc = argc - 1; /* number of proccesses goes here */
	my_id=(int *)malloc((nproc)*sizeof(int));;
	child_pids=(pid_t *)malloc((nproc)*sizeof(pid_t));
	pid_t pid;
	active_children=nproc;

	for(i=1;i<argc;i++)
	{



		/* Fork node */
		pid = fork();
		if (pid < 0) {
			perror("ola mantara");
			exit(1);
		}
		if (pid == 0) {
			/* Child */
			strcat(path, "./");
			strcpy(path, argv[i]);
			newargv[0] = path;
			//give id
			execve(argv[i], newargv, newenviron);
			perror("execve");
			exit(1);
		}
		my_id[i-1]=i-1;
		child_pids[i-1]=pid;
		kill(pid, SIGSTOP);
	}
	/* 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);
	}
	kill(child_pids[counter],SIGCONT);

	alarm(SCHED_TQ_SEC);    //for 2 sec
	/* loop forever  until we exit from inside a signal handler. */
	while (active_children > 0)
	{
		//le pipa
	}
	exit(0);

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

Άσκηση 2

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

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

#include "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 */

pid_t *child_pids;
int counter=0;
int nproc;
int *my_id;
int active_children;



/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
	int i;
	printf("List of Processes Active Now\nShell is Active Now\n");
```

```
	for(i=1; i<nproc; i++){
		if(my_id[i]!=-1) printf("Process with ID:%d and PID:%d Active
Now\n",my_id[i],child_pids[i]);
	}
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id)
{
	kill(child_pids[id],SIGKILL);
	my_id[id]=-1;
	if(id!=0) printf("Process %d Die Devil!\n",id);
	else printf("You Killed Shell you Motherfucker!!\n");
	active_children--;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
		char *newargv[] = { NULL, "hello", "world", NULL };
		char *newenviron[] = { NULL };
		char path[30];

		pid_t pid;
		pid = fork();
		if (pid < 0) {
			perror("ola mantara");
			exit(1);
		}
		if (pid == 0) {
			/* Child */
			strcat(path, "./");
			strcpy(path, executable);
			newargv[0] = path;
			//give id
			execve(executable, newargv, newenviron);
			perror("execve");
			exit(1);
		}
		my_id = (int *) realloc(my_id, (nproc + 1) * sizeof(int));
		child_pids=(pid_t *)realloc(child_pids, (nproc+1)*sizeof(pid_t));
		my_id[nproc]=nproc;
		child_pids[nproc]=pid;
		nproc++;
		if(active_children == 1) alarm(SCHED_TQ_SEC);
		active_children++;
		kill(pid, SIGSTOP);


}

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

```
                sched_print_tasks();
                return 0;

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

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

        default:
                return -ENOSYS;
    }
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
    if(active_children>1){
    if(counter==0){printf("\n");}
    else printf("The number: %d stopped to take a piss\n", counter);
    kill(child_pids[counter++],SIGSTOP);
    }
}


/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
    pid_t pid;
    int status, i;
    i = 0;
    while((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) != -1)
    {
        if(pid == 0)
        {
            return;
        }
        if(WIFEXITED(status))
        {
        for(i=0;i<nproc;i++)
        {
            if(child_pids[i] == pid)
            {
                if(i!=0) printf("Pethane o prog: %d sthn xestra\n", my_id[i]);
                else printf("Pethane to Shel MAgka mou :(\n");
                my_id[i] = -1;
                if(--active_children == 0)
                {
                    printf("Ola phgan popa\n");
                    exit(0);
                }

                counter = (++counter) % nproc;
                while(my_id[counter] ==  -1)
                {
```

```
                        counter = (++counter)%nproc;
                    }
                    alarm(0);
                    alarm(SCHED_TQ_SEC);
                    kill(child_pids[counter],SIGCONT);
                    if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
                    else printf("The number: %d came back from the toilet queue\n", counter);
                    break;
                }

            }
            }
            if(WIFSTOPPED(status))
            {

                counter=counter % nproc;
                alarm(SCHED_TQ_SEC);    //for 2 sec
                while(my_id[counter] ==  -1)
                {
                    counter = (++counter)%nproc;
                }
                if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
                else printf("The number: %d came back from the toilet queue\n", counter);
                kill(child_pids[counter],SIGCONT);
            }


            if(WIFSIGNALED(status) && WTERMSIG(status) == SIGKILL)
            {
            for(i=0;i<nproc;i++)
            {
                if(child_pids[i] == pid)
                {
                    if(i!=0) printf("Pethane o prog: %d sthn xestra\n", my_id[i]);
                    else printf("Pethane to Shel MAgka mou :(\n");
                    my_id[i] = -1;
                    if(--active_children == 0)
                    {
                        printf("Ola phgan popa\n");
                        exit(0);
                    }
                    if(counter == i)
                    {
                        counter = (++counter) % nproc;
                        while(my_id[counter] ==  -1)
                        {
                            counter = (++counter)%nproc;
                        }
                        alarm(0);
                        alarm(SCHED_TQ_SEC);
                        kill(child_pids[counter],SIGCONT);
                        if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
                        else printf("The number: %d came back from the toilet queue\n",
counter);
                    }
                    break;
                }

            }
            }
        }
}
```

```
/* 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 */
```

```
        close(pfds_rq[1]);
        close(pfds_ret[0]);
        *request_fd = pfds_rq[0];
        *return_fd = pfds_ret[1];
        child_pids[0]=p;
}

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;
        /*
         * For each of argv[1] to argv[argc - 1],
         * create a new child process, add it to the process list.
         */
        char *newargv[] = { NULL, "hello", "world", NULL };
        char *newenviron[] = { NULL };
        char path[30];

        /* Two file descriptors for communication with the shell */
        static int request_fd, return_fd;

        nproc = argc; /* number of proccesses goes here */
        my_id=(int *)malloc((nproc)*sizeof(int));;
        child_pids=(pid_t *)malloc((nproc)*sizeof(pid_t));
        pid_t pid;
        active_children=nproc;


        /* 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.
         */

        /* number of proccesses goes here */

        for(i=1;i<argc;i++)
        {



                /* Fork node */
                pid = fork();
                if (pid < 0) {
                        perror("ola mantara");
                        exit(1);
                }
                if (pid == 0) {
                        /* Child */
                        strcat(path, "./");
                        strcpy(path, argv[i]);
                        newargv[0] = path;
                        //give id
                        execve(argv[i], newargv, newenviron);
                        perror("execve");
                        exit(1);
                }

                my_id[i]=i;
                child_pids[i]=pid;
                kill(pid, SIGSTOP);
        }


        /* 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);
        }
        kill(child_pids[0],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;
}
```

Άσκηση 3

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

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

#include "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 */

pid_t *child_pids;
int *high_id;
int counter=0;
int nproc;
int *my_id;
int active_children;
int high_children;
int low_children;


static void
sched_high_task(int id)
{
	if(high_id[id] != -1){ printf("It's already high IDIOT\n");  return;}
	else if(my_id[id] == -1){printf("It's dead, you can't give it priority\n"); return;}

	high_id[id] = id;
	if(id != 0)
	{
		if(++high_children == 1) sched_high_task(0);
		printf("Process: %d is now high\n", my_id[id]);
	}
	else
	{
		if(high_children==0) printf("\nThe shell is on high priority, you are on GOD
MODE!!!\nBe careful because this situation is very unstable\n");
		++high_children;
	}
	low_children--;


}


static void
sched_low_task(int id)
{
	//TODO If a task goes low check if it was the one running on high so that you make the
```

```
next one run
	//TODO If we make the shell low while having other processes running on high nothing happens because we can never find the next one
	char c;
	if(id == 0 && high_children > 1)
	{
		printf("\n--------------------\nAre you sure you want to lower the shell??\nIf you do that you will not be able to do anything until all the high processes die\nAre you sure? (y/n)\n");
		c = getchar();
		if(c != 'y') return;
		high_id[0] = -1;
		--high_children;
		++low_children;
		//TODO Here do it for the special case that shell doesnt run anymore

		counter = (counter + 1) % nproc;
		while(high_id[counter] == -1)
		{
			counter = (counter + 1) % nproc;
		}

		kill(child_pids[counter],SIGCONT);
		if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
		else printf("The number: %d came back from the toilet queue\n", counter);
		return;
	}
	if(my_id[id] == -1){ printf("It's dead, you can't give it priority\n");  return;}
	else if(high_id[id] == -1){printf("It's already low SUPER IDIOT\n"); return;}

	//TODO Here do it for the general purpose if something goes low and it was running make the next one run on high if there is
	high_id[id] = -1;
	if(--high_children == 1) sched_low_task(0);
	low_children++;
	if(id != 0) printf("Process: %d is now low\n", my_id[id]);


	if(high_children > 0 && counter == id)
	{
		counter = (counter + 1) % nproc;
		while(high_id[counter] == -1)
		{
			counter = (counter + 1) % nproc;
		}

		kill(child_pids[counter],SIGCONT);
		if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
		else printf("The number: %d came back from the toilet queue\n", counter);
	}
	else if(high_children == 0 && counter == id)
	{
		counter = (counter + 1) % nproc;
		while(my_id[counter] == -1)
		{
			counter = (counter + 1) % nproc;
		}

		kill(child_pids[counter],SIGCONT);
		if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
		else printf("The number: %d came back from the toilet queue\n", counter);

	}
```

```
}
/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
	int i;
	printf("List of Processes Active Now\nShell is Active Now\n");
	for(i=1; i<nproc; i++){
		if(high_id[i]!=-1) printf("--HIGH-- Process with ID:%d and PID:%d Active
Now\n",my_id[i],child_pids[i]);
	}
	for(i=1; i<nproc; i++){
		if(my_id[i]!=-1 && high_id[i] == -1) printf("--LOW-- Process with ID:%d and PID:%d
Active Now\n",my_id[i],child_pids[i]);
	}
}

/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id)
{
	kill(child_pids[id],SIGKILL);
	if(high_id[id] != -1)
	{

		high_id[id] = -1;
		if(--high_children == 1)
		{
			//Move shell down
			sched_low_task(0);
		}
	}
	else if(my_id[id] != -1)
	{
		low_children--;
	}
	else{printf("You cant kill what is already dead!!!\n"); return;}
	my_id[id]=-1;
	if(id!=0) printf("Process %d Die Devil!\n",id);
	else printf("You Killed Shell you Motherfucker!!\n");
	active_children--;

	if(high_children > 0 && counter == id)
	{
		counter = (counter + 1) % nproc;
		while(high_id[counter] == -1)
		{
			counter = (counter + 1) % nproc;
		}

		kill(child_pids[counter],SIGCONT);
		if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
		else printf("The number: %d came back from the toilet queue\n", counter);
	}
	else if(high_children == 0 && counter == id)
	{
		counter = (counter + 1) % nproc;
```

```
		while(my_id[counter] == -1)
		{
			counter = (counter + 1) % nproc;
		}

		kill(child_pids[counter],SIGCONT);
		if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
		else printf("The number: %d came back from the toilet queue\n", counter);

	}
}

static void
print_dan()
{
	if(access("dan.txt", F_OK) != -1) system("cat dan.txt");
	else printf("Sorry but Dan deleted the file :'(\n");


}
/* Create a new task.  */
static void
sched_create_task(char *executable)
{
		char *newargv[] = { NULL, "hello", "world", NULL };
		char *newenviron[] = { NULL };
		char path[30];

		pid_t pid;
		pid = fork();
		if (pid < 0) {
			perror("ola mantara");
			exit(1);
		}
		if (pid == 0) {
			/* Child */
			strcat(path, "./");
			strcpy(path, executable);
			newargv[0] = path;
			//give id
			execve(executable, newargv, newenviron);
			perror("execve");
			exit(1);
		}
		my_id = (int *) realloc(my_id, (nproc + 1) * sizeof(int));
		high_id = (int *) realloc(high_id, (nproc + 1) * sizeof(int));
		child_pids=(pid_t *)realloc(child_pids, (nproc+1)*sizeof(pid_t));
		my_id[nproc]=nproc;
		high_id[nproc] = -1;
		child_pids[nproc]=pid;
		nproc++;
		//If the only process running is the shell
		if(active_children == 1) alarm(SCHED_TQ_SEC);
		active_children++;
		low_children++;
		kill(pid, SIGSTOP);


}
/* 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_high_task(rq->task_arg);
			return 0;
		case REQ_LOW_TASK:
			sched_low_task(rq->task_arg);
			return 0;
		case REQ_PRINT_DAN:
			print_dan();
			return 0;
		default:
			return -ENOSYS;
	}
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum)
{
	if(active_children>1){
		if(high_children == 1/* && high_id[0] != -1*/ ){alarm(SCHED_TQ_SEC);return;}
		if(counter==0){printf("\n");}
		else printf("The number: %d stopped to take a piss\n", counter);
		kill(child_pids[counter++],SIGSTOP);
	}
}


/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
	pid_t pid;
	int status, i;
	i = 0;
	while((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) != -1)
	{
		if(pid == 0)
		{
			return;
		}
		if(WIFEXITED(status) )
		{
		for(i=0;i<nproc;i++)
		{
			if(child_pids[i] == pid)
			{
```

```
                if(i!=0) printf("Pethane o prog: %d sthn xestra\n", my_id[i]);
                else printf("Pethane to Shel MAgka mou :(\n");
                if(high_id[i] != -1)
                {
                        if(--high_children == 1) sched_low_task(0);
                }
                else --low_children;
                high_id[i] = -1;
                my_id[i] = -1;
                if(--active_children == 0)
                {
                        printf("Ola phgan popa\n");
                        exit(0);
                }

                counter = (++counter) % nproc;
                if(high_children > 1)
                {
                        while(high_id[counter] ==  -1)
                        {
                                counter = (++counter)%nproc;
                        }
                }
                else
                {
                        while(my_id[counter] ==  -1)
                        {
                                counter = (++counter)%nproc;
                        }
                }
                alarm(0);
                alarm(SCHED_TQ_SEC);
                kill(child_pids[counter],SIGCONT);
                if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
                else printf("The number: %d came back from the toilet queue\n", counter);
                break;
        }

}
}
if(WIFSTOPPED(status))
{


        counter=counter % nproc;
        alarm(SCHED_TQ_SEC);    //for 2 sec
        if(high_children > 0)
        {
                while(high_id[counter] ==  -1)
                {
                        counter = (++counter)%nproc;
                }
        }
        else
        {
                while(my_id[counter] ==  -1)
                {
                        counter = (++counter)%nproc;
                }
        }
        if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
        else printf("The number: %d came back from the toilet queue\n", counter);
```

```
                kill(child_pids[counter],SIGCONT);
        }


        if(WIFSIGNALED(status) && WTERMSIG(status) == SIGKILL)
        {
        for(i=0;i<nproc;i++)
        {
                if(child_pids[i] == pid)
                {
                        if(i!=0) printf("Pethane o prog: %d sthn xestra\n", my_id[i]);
                        else printf("Pethane to Shel MAgka mou :(\n");
                        if(high_id[i] != -1)
                        {
                                if(--high_children == 1) sched_low_task(0);
                        }
                        else --low_children;
                        high_id[i] = -1;
                        my_id[i] = -1;
                        if(--active_children == 0)
                        {
                                printf("Ola phgan popa\n");
                                exit(0);
                        }

                        counter = (++counter) % nproc;
                        if(high_children > 1)
                        {
                                while(high_id[counter] ==  -1)
                                {
                                        counter = (++counter)%nproc;
                                }
                        }
                        else
                        {
                                while(my_id[counter] ==  -1)
                                {
                                        counter = (++counter)%nproc;
                                }
                        }
                        kill(child_pids[counter],SIGCONT);
                        if(counter==0) {printf("Shell> "); fflush(stdout);}//eimaste sto shell
                        else printf("The number: %d came back from the toilet queue\n", counter);
                        break;
                }

        }
        }
    }
}



/* 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 */
	close(pfds_rq[1]);
	close(pfds_ret[0]);
	*request_fd = pfds_rq[0];
	*return_fd = pfds_ret[1];
	child_pids[0]=p;
}

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;
	/*
	 * For each of argv[1] to argv[argc - 1],
	 * create a new child process, add it to the process list.
	 */
	char *newargv[] = { NULL, "hello", "world", NULL };
	char *newenviron[] = { NULL };
	char path[30];

	/* Two file descriptors for communication with the shell */
	static int request_fd, return_fd;

	nproc = argc; /* number of proccesses goes here */
	my_id=(int *)malloc((nproc)*sizeof(int));;
	high_id=(int *)malloc((nproc)*sizeof(int));;
	child_pids=(pid_t *)malloc((nproc)*sizeof(pid_t));
	pid_t pid;
	active_children=nproc;
	low_children = nproc;
	high_children = 0;


	/* Create the shell. */
	sched_create_shell(SHELL_EXECUTABLE_NAME, &request_fd, &return_fd);
	/* TODO: add the shell to the scheduler's tasks */
	high_id[0] = -1;
	/*
	 * For each of argv[1] to argv[argc - 1],
	 * create a new child process, add it to the process list.
	 */

	/* number of proccesses goes here */

	for(i=1;i<argc;i++)
	{
		/* Fork node */
		pid = fork();
```

```
		if (pid < 0) {
			perror("ola mantara");
			exit(1);
		}
		if (pid == 0) {
			/* Child */
			strcat(path, "./");
			strcpy(path, argv[i]);
			newargv[0] = path;
			//give id
			execve(argv[i], newargv, newenviron);
			perror("execve");
			exit(1);
		}

		my_id[i]=i;
		high_id[i] = -1;
		child_pids[i]=pid;
		kill(pid, SIGSTOP);
	}


	/* 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);
	}
	kill(child_pids[0],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;
}
```

Στην άσκηση 3 έχουμε βάλει μια επιπλέον εντολή στο scheduler 'd' που εμφανίζει το πρόσωπο με ascii art, του Αλέξανδρου (ενός βοηθού του εργαστηρίου)!