ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

6ο Εξάμηνο, Ακαδημαϊκή περίοδος 2017-2018

ΣΤΟΙΧΕΙΑ ΟΜΑΔΑΣ

Ομάδα: oslabe01

Μέλη Ομάδας: ΞΕΝΙΑΣ ΔΗΜΗΤΡΙΟΣ Α.Μ.: 03115084
ΦΙΛΙΠΠΟΥ ΜΙΧΑΗΛ Α.Μ.: 03115756

ΠΕΡΙΛΗΨΗ

Σκοπός της 4ης Εργαστηριακής Άσκησης είναι η υλοποίηση ενός χρονοδρομολογητή κυκλικής επαναφοράς (round-robin). Ο χρονοδρομολογητής πρέπει να εκτελείται ως γονική διεργασία, στο χώρο του χρήστη, κατανέμοντας τον υπολογιστικό χρόνο σε διεργασίες παιδιά. Ο χρονοδρομολογητής θα ελέγχει τα παιδιά με τα σήματα SIGCONT και SIGSTOP για την ενεργοποιήση και την διακοπή κάθε διεργασίας αντίστοιχα. Κάθες διεργασία εκτελείται για χρονικό διάστημα το πολύ ίσο με κβάντο χρόνου tq. Αν η διεργασία τερματιστεί πριν το τέλος του κβάντο χρόνου, ο χρονοδρομολογητής την αφαιρεί από την ουρά των έτοιμων διεργασιών και ενεργοποιεί την επόμενη. Αν το κβάντο χρόνου εκπνεύσει χωρίς η διεργασία να έχει ολοκληρώσει την εκτέλεσή της, τότε αυτή διακόπτεται, τοποθετείται στο τέλος της ουράς έτοιμων διεργασιών και ενεργοποιείται η επόμενη.

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

• Εκπνοή κβάντου χρόνου: Όταν το κβάντο χρόνου εκπνεύσει, ο χρονοδρομολογητής σταματά την τρέχουσα διεργασία. Η περίπτωση αυτή αντιστοιχεί σε χειρισμό του σήματος SIGALRM.

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

1. ΑΣΚΗΣΕΙΣ

1.1 Υλοποίηση χρονοδρομολογητή κυκλικής επαναφοράς στο χώρο χρήστη

Καλώντας το πρόγραμμα scheduler.c ως γονική διεργασία υλοποίησης του ζητούμενου χρονοδρομολογητή, του περνάμε ως όρισμα τα προγράμματα που θέλουμε να χρονοδρομολογηθούν, τα οποία ουσιαστικά ο χρονοδρομολογητής τα κάνει παιδιά του. Επομένως έχουμε μια γονική την διεργασία(τον χρονοδρομολογητή), ο οποίος χρονοδρομολογεί τα παιδία του (τα ορίσματά του). Η χρονοδρομολόγηση εκτελείται με βάση το παραπάνω πρωτόκολλο. Σε κάθε διεργασία αντιστοιχίζεται ένας σειριακός αριθμός id. Ο χρονοδρομολογητής εμφανίζει κατάλληλα μηνύματα κατά την ενεργοποίηση, διακοπή και τερματισμό των διεργασιών. Όταν όλες οι διεργασίες έχουν ολοκληρωθεί, αυτός τερματίζεται. Το πρόγραμμα λοιπόν που υλοποιεί τον χρονοδρομολογητή είναι το εξής:

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

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

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

/* Compile-time parameters. */
#define SCHED_TQ_SEC 2                /* time quantum */
#define TASK_NAME_SZ 60               /* maximum size for a task's name */

typedef struct Node{
        pid_t pid_of_process;
        char * name;
        int id;
        struct Node * next;
} Node_t;

Node_t * head, * tail;
pid_t current_pid;
int elementsinQueue,ids;

void insert(pid_t pid,char * name){
        Node_t * newnode= malloc(sizeof(Node_t));
        newnode->pid_of_process= pid;
        Node_t * tmp = head;
        while (tmp->next!= NULL) tmp=tmp->next;
        tmp->next=newnode;
        newnode->name=strdup(name); //dimiourgei enan pointer se duplicate char
        newnode->id=++ids;
        tail=newnode;
        elementsinQueue++;
}

void remove1 (pid_t pid){
        Node_t * tmp=head;
        while (tmp->next->pid_of_process != pid ) tmp=tmp->next;
        Node_t  * todelete=tmp->next;
        tmp->next= tmp->next->next;
        free(todelete);
        elementsinQueue--;
        if (elementsinQueue==0){
                printf("All the processes terminated\n");
                exit(1);
        }
}
```

```c
/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum){
        if ( kill(current_pid,SIGSTOP)<0) perror("stop error");
}
/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        int p,status;
        for (;;) {
                p=waitpid(-1,&status,WUNTRACED | WNOHANG);// i waitpid perimenei me -1 opoiodipote paidi na termatistei.wstoso me to wuntraced,enime
rwnei to status akomi kai an auto to paidi egine stopped apo kapoio sima. to whoang allazei katastasi an kanena paidi den egine exit

                if (p==0) // kanena paidi dn  allazei katastasi kai epistrefetai amesws to 0
                        break;
                explain_wait_status(p,status);
                if (WIFEXITED(status) || WIFSIGNALED(status) ) {
                        remove1(current_pid);
                        if (kill(head->pid_of_process,SIGCONT) < 0) perror("continues error 1");
                        current_pid= head->pid_of_process;
                        head=head->next;
                        tail=tail->next;
/* 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; // dimiourgei mia maska,wste oso ginetai execute tou handler auti na ginetai block me to sima pou piastike kai na min m
porei na piasei allo sima pou brisketai sto sigset  mexri na teleiwsei o handler tin leitourgia
        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:t sigpipe");
                exit(1);
        }
}
```

```c
int main(int argc, char *argv[])
{
        int nproc;
        /*
         * For each of argv[1] to argv[argc - 1],
         * create a new child process, add it to the process list.
         */
        ids=0;
        nproc = argc-1; /* number of proccesses goes here */
        head=malloc(sizeof(Node_t));
        head->next= NULL;
        int i;
        for ( i=1; i<= nproc; i++){
                pid_t pid= fork();
                if (pid>0) insert(pid,argv[i]);
                else if (pid==0){
                        char *newargv[]={argv[i],NULL,NULL,NULL};
                        raise(SIGSTOP);
                        execve(argv[i],newargv,NULL);
                }
        }
        if (nproc == 0) {
                fprintf(stderr, "Scheduler: No tasks. Exiting...\n");
                exit(1);
        }

        head=head->next;
        free(tail->next);
        tail->next=head;
        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(nproc);

        /* Install SIGALRM and SIGCHLD handlers. */
        install_signal_handlers();
        if ( kill(head->pid_of_process,SIGCONT)<0){
                perror("continue error");
        }
        current_pid=head->pid_of_process;
        head=head->next;
        tail= tail ->next;
        alarm(SCHED_TQ_SEC);

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

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

Η υλοποίηση αυτής της κυκλικής ουράς εκτέλεσης των προγραμμάτων-παιδιών, γίνεται μέσω μιας συνδεδεμένης λίστας, της οποίας η ουρά έχει ενωθεί με το κεφάλι! Έτσι όταν αυτή η ουρά αδειάσει τελειώς, δηλαδή όλα τα παιδιά έχουν τερματιστεί(είτε φυσιολογικά είτε μέσω κάποιου SIGKILL), τότε ο χρονοδρομολογητής σταματάει την εκτέλεση του προγράμματός του και κλείνει. Μία ενδεικτική έξοδος του χρονοδρομολογητή μας με είσοδο 2 προγράμματα (“prog” ως arguments) που ουσιαστικά είναι τα παιδιά του (τα προγράμματα που θα χρονοδρομολογηθούν), είναι το παρακάτω. Σημειώνεται ότι ένα πρόγραμμα prog εμφανίζει 10 μηνύματα(στο ακόλουθο παράδειγμα) και μετά σταματάει.

```
oslabe01@os-node1:~/dimitris/sched$ ./scheduler prog prog
My PID = 3330: Child PID = 3331 has been stopped by a signal, signo = 19
My PID = 3330: Child PID = 3332 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 10, delay = 126
prog[3331]: This is message 0
prog[3331]: This is message 1
prog[3331]: This is message 2
prog[3331]: This is message 3
prog[3331]: This is message 4
prog[3331]: This is message 5
My PID = 3330: Child PID = 3331 has been stopped by a signal, signo = 19
prog: Starting, NMSG = 10, delay = 108
prog[3332]: This is message 0
prog[3332]: This is message 1
prog[3332]: This is message 2
prog[3332]: This is message 3
prog[3332]: This is message 4
prog[3332]: This is message 5
prog[3332]: This is message 6
My PID = 3330: Child PID = 3332 has been stopped by a signal, signo = 19
prog[3331]: This is message 6
prog[3331]: This is message 7
prog[3331]: This is message 8
prog[3331]: This is message 9
My PID = 3330: Child PID = 3331 terminated normally, exit status = 0
prog[3332]: This is message 7
prog[3332]: This is message 8
prog[3332]: This is message 9
My PID = 3330: Child PID = 3332 terminated normally, exit status = 0
All processes terminated
```

ΕΡΩΤΗΣΕΙΣ

1. Στην συνάρτηση install_signal_handlers() δημιουργούμε μια μάσκα στην δομή sa που είναι μια δομή sigaction. Σε αυτή βάζουμε ένα σύνολο απο σήματα sigset(το οποίο το έχουμε αρχικοποιήσει ώστε να είναι άδειο), στο οποίο σύνολο έχουμε προσθέσει τα σήματα SIGALRM και SIGCHLD. Αν λοιπόν πρώτα έρθει ένα σήμα SIGCHLD, η sigaction θα μας στείλει στον sigchld_handler και αυτός θα αρχίσει να εκτελεί τις απαραίτητες λειτουργίες. Παράλληλα όμως με την κλήση του sigchld_handler, η μάσκα αντιλαμβάνεται ότι έχει έρθει ένα σήμα που περιέχει στο σύνολό της και επομένως απαγορεύει την πρόσληψη οποιουδήποτε άλλου σήματος εντός του συνόλου της. Έτσι αν ο sigchld_handler εκτελείται και έρθει SIGALRM, η μάσκα δεν επιτρέπει στην sigaction να ενεργοποιήσει τον sigalrm_handler, καθώς βρίσκεται σε λειτουργία άλλος handler ενός σήματος του συνόλου της. Έτσι περιμένει μέχρι να τελειώσει ο sigchld_handler και στην συνέχεια απελευθερώνει την δέσμευση της και επιτρέπει στον sigalrm_handler να κληθεί, δεσμεύοντας πάλι το σήμα SIGALRM του sigset της. Επομένως εμείς στην υλοποίησή μας χρησιμοποιούμε ένα σύνολο και μια μάσκα που κάνει αποκλεισμό των υπόλοιπων σημάτων μέχρι να τελειώσει ο εν ενεργεία sighandler. Εν αντιθέσει, ένας χρονοδρομολογητής σε χώρο πυρήνα θα χρησιμοποιούσε κάποια hardware interrupts. Έτσι ο χρονοδρομολογητής θα δέχεται ένα σήμα SIGCHLD απο το χώρο χρήστη, θα μεταφέρεται σε χώρο πυρήνα, θα κάνει τις απαραίτητες αλλαγές και ενώ λοιπόν είμαστε σε χώρο πυρήνα, αν έρθει άλλο σήμα SIGALRM, τότε το υλικό δεν θα του επιτρέψει να σταλθεί σε αυτόν το σήμα, αφού το υλικό μας είναι “πιασμένο” από ένα άλλο σήμα.

2. Κάθε φορά που ο χρονοδρομολογητής λαμβλανει σήμα SIGCHLD, περιμένουμε να αναφέρεται στην διεργασία που εκείνη την ώρα βρίσκεται υπό εκτέλεση στο συγκεκριμένο κβάντο χρόνου του χρονοδρομολογητή. Αυτό συμβαίνει γιατί η μοναδική διεργασία που υφίστατει αλλαγές, όπως δηλαδή να πάρει SIGSTOP λόγω alarm είτε να τελειώσει έχοντας ολοκληρώσει τις λειτουργίες της, είναι φυσικά η διεργασία που τρέχει αυτή την στιγμή. Ωστόσο αν λόγω κάποιου εξωτερικού παράγοντα(π.χ. αποστολή SIGKILL) τερματιστεί αναπάντεχα μια οποιαδήποτε διεργασία-παιδί, τότε η waitpid(που λόγω -1, ενημερώνεται για κάθε διεργασία-παιδί) ενημερώνει το status και η WIFSIGNALED δίνει true και τότε αφαιρείται από την λίστα(αφού τερματίστηκε λόγω σήματος) η επιστραφούσα διεργασία.

3. Η όλη διαχείριση και ο καθορισμός των λειτουργιών του χρονοδρομολογητή γίνεται όπως φαίνεται στον sigchld_handler. Είναι ουσιαστικά ο handler που ενεργοποιείται όταν έρθει κάποιο σήμα τερματισμού ή διακοπής σήματος κάποιου παιδιού-διεργασίας. Αντίστοιχα ο sigalrm_handler ενεργοποιείται σε σήμα τερματισμού του κβάντου χρόνου που έχει οριστεί με την alarm(“κβαντο χρονου”). Έστω όμως ότι θα θέλαμε να χρησιμοποιήσουμε μόνο το σήμα SIGALRM για να σταματά την τρέχουσα διεργασία και να ξεκινά την επόμενη. Το πρώτο πράγμα που αντιλαμβανόμαστε είναι ότι ακόμα και αν διεργασία τελείωνε για κάποιο λόγο πριν το κβάντο χρόνου που της δίνεται, ο sigalrm_handler θα ενεργοποιούσε την επόμενη στην ουρά διεργασία μετά το πέρασμα του προκαθορισμένου κβάντου χρόνου. Έτσι καταλαβαίνουμε ότι θα υπήρχε ένας “νεκρός” χρόνος στον χρονοδρομολογητή που καμία διεργασία δεν θα εκτελούνταν. Ο δεύτερος και κυριότερος λόγος, είναι το θέμα συγχρονισμού. Αν ένας handler κάνει μια διεργασία SIGSTOP και μια άλλη SIGCONT, δεν υπάρχει ασφαλής και συχρονισμένη μετάδοση των σημάτων. Έτσι είναι απαραίτητο ένας handler να κάνει το SIGSTOP και ένας άλλος, που θα καλείται με το προηγούμενο SIGSTOP και θα κάνει SIGCONT την επόμενη διεργασία. Έτσι έχουμε μια συχρονισμένη ντετερμινιστική λειτουργία.

1.2. Έλεγχος λειτουργίας χρονοδρομολογητή μέσω φλοιού

Ζητείται η επέκταση του χρονοδρομολογητή του προηγούμενου ερωτήματος, ώστε να υποστηρίζεται ο έλεγχος της λειτουργίας του μέσω προγράμματος-φλοιού, μέσω κάποιων ειδικών εντολών. Το πρόγραμμα shell.c και το request.h καθορίζουν τον τρόπο που λειτουργεί το shell και το πως αλληλεπιδρά με τον χρονοδρομολογητή. Σκοπός είναι η δημιουργία ενός χρονοδρομολογητή που θα παίρνει τα request από το πρόγραμμα του φλοιού και θα εκτελεί τις συγκεκριμένες λειτουργίες ώς ο πατέρας όλων των παιδιών. Σημειώνεται ότι και το πρόγραμμα του φλοιού δίνεται ως διεργασία-παιδί του χρονοδρομολογητή, ο οποίο θα υλοποιεί τις απαιτήσεις του φλοιού. Το πρόγραμμα-πατέρας του χρονοδρομολογητή είναι το ακόλουθο:

```c
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>
#include <sys/wait.h>
#include <sys/types.h>

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

/* 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 */

typedef struct Node{
        pid_t pid_of_process;
        char * name;
        int id;
        struct Node * next;
} Node_t;

Node_t * head, * current;
pid_t current_pid,shells_pid;
int elementsinQueue;
int ida;
void insert(pid_t pid,char * name){
        Node_t * newnode= malloc(sizeof(Node_t));
        newnode->pid_of_process= pid;
        newnode->id=++ida;
        Node_t * tmp = head;
        while (tmp->next!= NULL) tmp=tmp->next;
        tmp->next=newnode;
        newnode->name=strdup(name); //dimiourgei enan pointer se duplicate char
        elementsinQueue++;
}

void remove1 (pid_t pid){
        Node_t * tmp=head;
        while (tmp->next->pid_of_process != pid ) tmp=tmp->next;
        Node_t  * todelete=tmp->next;
        tmp->next=todelete->next;
        free(todelete);
        elementsinQueue--;
        if (elementsinQueue==0){
                printf("All the processes terminated\n");
                exit(1);
        }
}
```

```c
/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
        Node_t *tmp;
        int i;
        for ( tmp=head->next, i=0; i< elementsinQueue; tmp=tmp->next,i++){
                if ( tmp->pid_of_process == current_pid){
                        printf("Name of process: %s (CURRENT)   ID: %d PID: %ld\n", tmp->name,tmp->id,(long)tmp->pid_of_process);
                }
                else {
                        printf("Name of process: %s \t ID: %d PID: %ld\n", tmp->name,tmp->id,(long)tmp->pid_of_process);
                }
                puts("\n");
        }
}


/* Send SIGKILL to a task determined by the value of its
 * scheduler-specific id.
 */
static int
sched_kill_task_by_id(int id)
{
        Node_t *tmp=head;
        while (tmp->next != NULL && tmp->next->id !=id ) tmp=tmp->next;
        if (tmp->next == NULL) return 1;
        kill(tmp->next->pid_of_process,SIGKILL);
        remove1(tmp->next->pid_of_process);
        printf("Killed process with id: %d\n", id);
        return 0;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
        pid_t p=fork();
        if (p>0) insert(p,executable);
        if (p==0){
                char *newargv[]={executable,NULL,NULL,NULL};
                raise(SIGSTOP);
                execve(executable,newargv,NULL);
        }

}

/* 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  (kill(current_pid,SIGSTOP)<0) perror("sigalrm error");

}
```

```c
/*
 * SIGCHLD handler
 */
static void
sigchld_handler(int signum)
{
        int p,status;
        for (;;) {
                p=waitpid(-1,&status,WUNTRACED | WNOHANG);// i waitpid perimenei me -1 opoiodipote paidi na termatistei.wstoso me to wuntraced,enimerwn
oio sima. to whoang allazei katastasi an kanena paidi den egine exit

                if (p==0) // kanena paidi dn  allazei katastasi kai epistrefetai amesws to 0
                        break;
                explain_wait_status(p,status);
                if (WIFEXITED(status)|| WIFSIGNALED(status) ) {
                        if ( p==current_pid){
                                remove1(current_pid);
                                if (current->next!=NULL) current=current->next;
                                else current=head->next;
                                current_pid=current->pid_of_process;
                                if(kill(current->pid_of_process,SIGCONT) < 0) perror("continues error 1");
                                alarm(SCHED_TQ_SEC);
                        }
                }
                if (WIFSTOPPED(status)){
                        if (current->next !=NULL) current=current->next;
                        else current=head->next;
                        current_pid=current->pid_of_process;
                        if (kill(current->pid_of_process,SIGCONT) < 0) perror("continues error 2");
                        alarm(SCHED_TQ_SEC);
                }

        }

}

/* Disable delivery of SIGALRM and SIGCHLD. */
static void
signals_disable(void)
{
        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
                perror("signals_disable: sigprocmask");
                exit(1);
        }
}
```

```c
/* Enable delivery of SIGALRM and SIGCHLD.  */
static void
signals_enable(void)
{
        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
                perror("signals_enable: sigprocmask");
                exit(1);
        }
}


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

        sa.sa_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);
        }
}
```

```c
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 pid_t
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];
        return p;
}
```

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

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

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

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

```c
int main(int argc, char *argv[])
{
        int nproc;
        ida=0;
        /* Two file descriptors for communication with the shell */
        static int request_fd, return_fd;
        /* Create the shell. */
        shells_pid=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 */
        head=malloc(sizeof(Node_t));
        head->next= NULL;
        insert(shells_pid,SHELL_EXECUTABLE_NAME);
        int i;
        for ( i=1; i<= nproc; i++){
                pid_t pid= fork();
                if (pid>0) insert(pid,argv[i]);
                else if (pid==0){
                        char *newargv[]={argv[i],NULL,NULL,NULL};
                        raise(SIGSTOP);
                        execve(argv[i],newargv,NULL);
                }
        }

        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(nproc+1);

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

        if (kill(head->next->pid_of_process,SIGCONT)<0) perror("continues error");
        current_pid=head->next->pid_of_process;
       // if (head->next->next!= NULL) current=head->next->next;
       // else current=head->next;
        current=head->next;
        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;
}
```

Ο φλοιός δέχεται 4 εντολές. Η εντολή “p” κάνει print όλες τι τρέχουσες διεργασίες- παιδιά και δείχνει επίσης ποιά διεργασία τρέχει. Ένα παράδειγμα εξόδου της “p” είναι το ακόλουθο:

```
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Name of process: shell (CURRENT)   ID: 1 PID: 3584


Name of process: prog    ID: 2 PID: 3585


Name of process: prog    ID: 3 PID: 3586
```

Η εντολή “k <id>” σκοτώνει μια διεργασία με το συγκεκριμένο id. Ενδεικτικό παράδειγμα:

Before:

```
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Name of process: shell (CURRENT)   ID: 1 PID: 3593


Name of process: prog    ID: 2 PID: 3594


Name of process: prog    ID: 3 PID: 3595
```

type “k 2”
After:

```
Killed process with id: 2
Shell> Shell: issuing request...
Shell: receiving request return value...
Name of process: shell (CURRENT)   ID: 1 PID: 3593


Name of process: prog    ID: 3 PID: 3595
```

Η εντολή “e <programm>” δημιουργεί μια νέα διεργασία που τρέχει το πρόγραμμα <programm> και το βάζει στον χρονοδρομολογητή. Ενδεικτικό παράδειγμα :

Before:

```
Shell> p
Shell: issuing request...
Shell: receiving request return value...
Name of process: shell (CURRENT)   ID: 1 PID: 3637


Name of process: prog    ID: 2 PID: 3638

e prog
Shell: issuing request...
Shell: receiving request return value...
```

After:

```
Shell: issuing request...
Shell: receiving request return value...
Name of process: shell (CURRENT)   ID: 1 PID: 3637


Name of process: prog    ID: 2 PID: 3638


Name of process: prog    ID: 3 PID: 3639
```

Τέλος η
εντολή “q” σταματάει την λειτουργία του προγράμματος shell. Ενδεικτικό παράδειγμα είναι το εξής :

```
Shell: Exiting. Goodbye.
My PID = 3640: Child PID = 3641 terminated normally, exit status = 0
```

ΕΡΩΤΗΣΕΙΣ

1. Όταν ο φλοιός υφίστατει χρονοδρομολόγηση, η εντολή που εμφανίζεται πάντα ως τρέχουσα διεργασία στη λίστα διεργασιών είναι ο φλοιός. Αυτό ισχύει διότι η εντολή του φλοιού για τύπωμα της λίστας διεργασιών γίνεται προφανώς όταν ο φλοιός τρέχει ως πρόγραμμα. Έτσι κάθε φορά στο τύπωμα η διεργασία που τρέχει στον χρονοδρομολογητή είναι ο ίδιος ο φλοιός. Ένας τρόπος να μην συνέβαινε αυτό θα ήταν ο φλοιός να τρέχει ως background και όχι ως διεργασία-παιδί, ούτως ώστε μόλις δεχόταν την εντολή να εμφάνιζε απευθείας τη λίστα και ποια διεργασία τρέχει αυτή τη στιγμή.

2. Η shell_request_loop() τρέχει καθ’όλη τη διάρκεια εκτέλεσης του χρονοδρομολογητή και ουσιαστικά μας δίνει τη δυνατότητα να πληκτρολογήσουμε οποιαδήποτε στιγμή στο shell μια εντολή(ακόμη και αν μια άλλη διεργασία τρέχει εκείνη τη στιγμή) και αυτή η εντολή να αποθηκευτεί στον buffer για να δωθεί ως εντολή στο πρόγραμμα shell. Η συναρτήσεις signal_disable() και signal_enable() χρησιμοποιούνται στην συγκεκριμένη συνάρτηση, ώστε μόλις δωθεί μια εντολή και περαστεί στον buffer ,να γίνει αποκλεισμός των σημάτων SIGCHLD και SIGALRM ώστε να περαστεί η εντολή που κάναμε request στο πρόγραμμα shell. Μετά βέβαια ξανακάνουμε enable για να συνεχιστεί η ροή του προγράμματος. Είναι πολύ σημαντική η χρήση αυτών των συναρτήσεων, διότι όταν δινουμε μια εντολή στον shell, του λέμε να κάνει κάποια μεταβολή στην ουρά του χρονοδρομολογητή με βάση κάποια εντολή εισόδου. Όμως αν παράλληλα

δεχτεί ο χρονοδρομολογητής μια εντολή για να μετατρέψει μια διεργασία της ουράς, τότε θα έχουμε μια παράλληλη επεξεργασία της λίστας, το οποίο μπορεί να κάνει το πρόγραμμά μας να σκάσει. Έτσι θέλουμε τα κομμάτια επεξεργασίας της λίστας να γίνονται ατομικά.

1.3 Υλοποίηση προτεραιοτήτων στο χρονοδρομολογητή

Στην άσκηση αυτή επεκτείνουμε τον χρονοδρομολογητή του προηγούμενου ερωτήματος, ώστε να υποστηρίζονται δυο κλάσεις προτεραιότητας: LOW και HIGH. Ο αλγόριθμος χρονοδρομολόγησης αλλάζει ως εξής: αν υπάρχουν διεργασίες προτεραιότητας HIGH εκτελούνται μόνο αυτές χρησιμοποιώντας κυκλική επαναφορά. Σε αντίθετη περίπτωση, χρονοδρομολογούνται οι LOW διεργασίες χρησιμοποιώντας κυκλική επαναφορά. Όλες οι διεργασίες δημιουργούνται με LOW προτεραιότητα. Η αλλαγή της προτεραιότητας μιας διεργασίας πραγματοποιείται με τη εντολή : h <id> ή l <id> για την δυναμική αλλαγή της προτεραιότητας της διεργασίας id. Η επέκταση του προηγούμενου χρονοδρομολογητή για τις ανάγκες του συγκεκριμένου ερωτήματος είναι η εξής :

```c
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <assert.h>
#include "stdbool.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 */
typedef enum {LOW, HIGH} priority;
typedef struct Node{
        pid_t pid_of_process;
        char * name;
        int id;
        struct Node * next;
        priority pr;
} Node_t;

Node_t * head, * current;
pid_t current_pid,shells_pid;
int elementsinQueue;
int ida;
void insert(pid_t pid,char * name){
        Node_t * newnode= malloc(sizeof(Node_t));
        newnode->pid_of_process= pid;
        newnode->id=++ida;
        newnode->pr =LOW;
        Node_t * tmp = head;
        while (tmp->next!= NULL) tmp=tmp->next;
        tmp->next=newnode;
        newnode->name=strdup(name); //dimiourgei enan pointer se duplicate char
        elementsinQueue++;
}

void remove1 (pid_t pid){
        Node_t * tmp=head;
        while (tmp->next->pid_of_process != pid ) tmp=tmp->next;
        Node_t  * todelete=tmp->next;
        tmp->next=todelete->next;
        free(todelete);
        elementsinQueue--;
        if (elementsinQueue==0){
                printf("All processes terminated\n");
                exit(1);
        }
}
```

```
void find_next_node(void){
        Node_t *tmp=current;
        while (tmp->next != NULL && tmp->next->pr != HIGH ) tmp=tmp->next;
        if (tmp->next== NULL){
                tmp=head;
                while (tmp->next != NULL && tmp->next->pr != HIGH ){
                        if (tmp==current){
                                break;
                        }
                        tmp=tmp->next;
                }
                if (tmp->next == NULL ) {
                        current=head->next;
                        current_pid=current->pid_of_process;
                        return;
                }
        }
        current=tmp->next;
        current_pid=current->pid_of_process;
}


const char* getPriority( priority pr){
        switch (pr){
                case LOW: return "LOW";
                case HIGH: return "HIGH";
        }
}

/* Print a list of all tasks currently being scheduled.  */
static void
sched_print_tasks(void)
{
        Node_t *tmp;
        int i;
        for ( tmp=head->next, i=0; i< elementsinQueue; tmp=tmp->next,i++){
                printf("Name: %s \t ID: %d PID: %ld Priority: %s\n", tmp->name,tmp->id,(long)tmp->pid_of_process, getPriority(tmp->pr));
                if ( tmp->pid_of_process== current_pid) printf (" CURRENT");
                puts ( "\n");
        }
}
```

```
static int
sched_kill_task_by_id(int id)
{       if (id==1) {
                printf("Press q to terminate shell\n");
                return 1;
        }
        Node_t *tmp=head;
        while (tmp->next != NULL && tmp->next->id !=id ) tmp=tmp->next;
        if (tmp->next == NULL) return 1;
        kill(tmp->next->pid_of_process,SIGKILL);
        remove1(tmp->next->pid_of_process);
        printf("Killed process with id: %d\n", id);
        return 0;
}


/* Create a new task.  */
static void
sched_create_task(char *executable)
{
        pid_t p=fork();
        if (p>0) insert(p,executable);
        if (p==0){
                char *newargv[]={executable,NULL,NULL,NULL};
                raise(SIGSTOP);
                execve(executable,newargv,NULL);
        }

}

static void sched_set_low(int id){
        Node_t *tmp= head;
        while (tmp->next != NULL && tmp->next->id!= id ) tmp=tmp->next;
        if (tmp->next!=NULL) tmp->next->pr = LOW;
}

static void sched_set_high(int id){
        Node_t *tmp=head;
        while (tmp->next!= NULL && tmp->next->id != id) tmp=tmp->next;
        if (tmp != NULL ) tmp->next->pr= HIGH;
}
```

```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:
                        sched_set_high(rq->task_arg);
                        return 0;

                case REQ_LOW_TASK:
                        sched_set_low(rq->task_arg);
                        return 0;

                default:
                        return -ENOSYS;
        }
}

/*
 * SIGALRM handler
 */
static void
sigalrm_handler(int signum){
        if  (kill(current_pid,SIGSTOP)<0) perror("sigalrm error");

}
```

```c
static void
sigchld_handler(int signum)
{
       int p,status;
        for (;;) {
                p=waitpid(-1,&status,WUNTRACED | WNOHANG);// i waitpid perimenei me -1 opoiodipote paidi na termatistei.wstoso me to wuntrac
oio sima. to whoang allazei katastasi an kanena paidi den egine exit

                if (p==0) // kanena paidi dn  allazei katastasi kai epistrefetai ameswss to 0
                        break;
                explain_wait_status(p,status);
                if (WIFEXITED(status)|| WIFSIGNALED(status) ) {
                        if ( p==current_pid){
                                remove1(current_pid);
                                find_next_node();
                                if(kill(current->pid_of_process,SIGCONT) < 0) perror("continues error 1");
                                alarm(SCHED_TQ_SEC);
                        }
                }
                if (WIFSTOPPED(status)){
                        find_next_node();
                        if (kill(current->pid_of_process,SIGCONT) < 0) perror("continues error 2");
                        alarm(SCHED_TQ_SEC);
                }

        }

}

/* Disable delivery of SIGALRM and SIGCHLD. */
static void
signals_disable(void)
{
        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) {
                perror("signals_disable: sigprocmask");
                exit(1);
        }
}

/* Enable delivery of SIGALRM and SIGCHLD.  */
static void
signals_enable(void)
{
        sigset_t sigset;

        sigemptyset(&sigset);
        sigaddset(&sigset, SIGALRM);
        sigaddset(&sigset, SIGCHLD);
        if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) {
                perror("signals_enable: sigprocmask");
                exit(1);
        }
}
```

```c
/* 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);
}
```

```c
static pid_t
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];
        return 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;
                }
        }
}
```

```c
int main(int argc, char *argv[])
{
        int nproc;
        ida=0;
        /* Two file descriptors for communication with the shell */
        static int request_fd, return_fd;
        /* Create the shell. */
        shells_pid=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 */
        head=malloc(sizeof(Node_t));
        head->next= NULL;
        insert(shells_pid,SHELL_EXECUTABLE_NAME);
        int i;
        for ( i=1; i<= nproc; i++){
                pid_t pid= fork();
                if (pid>0) insert(pid,argv[i]);
                else if (pid==0){
                        char *newargv[]={argv[i],NULL,NULL,NULL};
                        raise(SIGSTOP);
                        execve(argv[i],newargv,NULL);
                }
        }

        /* Wait for all children to raise SIGSTOP before exec()ing. */
        wait_for_ready_children(nproc+1);

        /* Install SIGALRM and SIGCHLD handlers. */
        install_signal_handlers();
        current=head;
        find_next_node();
        if (kill(current->pid_of_process,SIGCONT)<0) perror("continues error");

        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;
}
```

Μια τυπική έξοδος εκτέλεσης του προγράμματος με δύο διεργασίες prog ως όρισμα και χωρίς αρχικές μεταβολές των προτεραιοτήτων είναι η εξής (χρησιμοποιώντας την εντολή ‘p’):

```
Shell: receiving request return value...
Name: shell      ID: 1 PID: 4408 Priority: LOW
 CURRENT

Name: prog       ID: 2 PID: 4409 Priority: LOW


Name: prog       ID: 3 PID: 4410 Priority: LOW
```

Στην συνέχεια, ορίζοντας ως HIGH priority τις διεργασίες με id 1 και id 3 μέσω των εντολών h 1 και h 3 αντίστοιχα, παίρνουμε το εξής αποτέλεσμα :

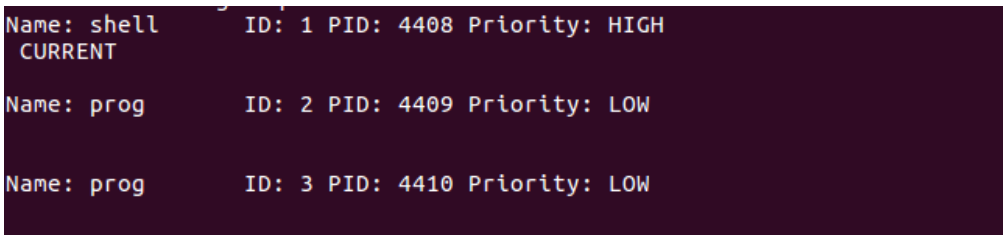
```
Shell: receiving request return value...
Name: shell      ID: 1 PID: 4408 Priority: HIGH
 CURRENT

Name: prog       ID: 2 PID: 4409 Priority: LOW


Name: prog       ID: 3 PID: 4410 Priority: HIGH
```

όπου οι διεργασίες με high priority εκτελούνται αποκλειστικά.

Τέλος, πληκτρολογώντας την εντολή l 3 παίρνουμε την εξής κατάσταση:



στην οποία η υπό εκτέλεση διεργασία είναι μόνο η διεργασία του shell.

ΕΡΩΤΗΣΕΙΣ

1. Ουσιαστικά ένα πολύ μεγάλο ζήτημα λιμοκτονίας είναι να έχουμε HIGH προτεραιότητα σε διεργασίες που για να ολοκληρωθούν χρειάζονται πολλά κβάντα χρόνου , ενώ παράλληλα έχουμε σε priority LOW το shell. Στην περίπτωση αυτή, έχουμε αποκλείσει την δυνατότητα της δυναμικής επέμβασης μας στο χειρισμό των διεργασιών και το μόνο που έχουμε είναι να περιμένουμε τις διεργασίες υψηλής προτεραιότητας να ολοκληρωθούν, ώστε να δώσουμε την σκυτάλη στις χαμηλής προτεραιότητας διεργασίες και προφανώς και στον shell. Επίσης πρόβλημα δημιουργείται όταν για παράδειγμα έχουμε κάποιες διεργασίες χαμηλής προτεραιότητας και παράλληλα δημιουργούμε νέες διεργασίες μέσω του shell τις οποίες τις βάζουμε συνέχεια σε high priority. Τότε δημιουργείται ένας μεγάλος χρόνος κύκλος εκτέλεσης διεργασιών υψηλής προτεραιότητας, με αποτέλεσμα οι διεργασίες χαμηλής προτεραιότητας να λιμοκτονούν μέχρι να λάβουν την σκυτάλη. Είναι επομένως εμφανές ότι όταν χρησιμοποιούμε χρονοδρομολόγηση με σειρές προτεραιότητας, πρέπει να χρησιμοποιούμε το εργαλείο αυτό συνειδητά και με έλεγχο ώστε να μην οδηγηθούμε σε δυσάρεστες και άδικες καταστάσεις.

ΤΕΛΟΣ ΑΝΑΦΟΡΑΣ