ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Λειτουργικά Συστήματα

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2017-2018

Άσκηση 3 : Συγχρονισμός

ΣΤΟΙΧΕΙΑ ΟΜΑΔΑΣ

Ομάδα : oslabe01

Μέλη Ομάδας : Ξενίας Δημήτριος Α.Μ.: 03115084

Φιλίππου Μιχαήλ Α.Μ.: 03115756

ΕΙΣΑΓΩΓΗ

Σκοπός της 3ης Εργαστηριακής Άσκησης είναι η εξοικείωση μας με τον συγχρονισμό πολυνηματικών εφαρμογών βασισμένων στο πρότυπο POSIX threads. Ο συγχρονισμός των νημάτων που τρέχουν πάνω σε μια διεργασία γίνεται απαραίτητος αρκετές φορές, είτε όταν πολλά νήματα πρέπει να χρησιμοποιήσουν μεταβλητές κοινές της διεργασίας, είτε όταν θέλουμε να "συνεργαστούνε" αυτά τα νήματα ώστε να παράγουν ένα αποτέλεσμα σε ίσως γρηγορότερο χρόνο από ότι θα χρειαζόταν ένα νήμα μόνο του. Οι μηχανισμοί λοιπόν με τους οποίους επιτυγχάνεται ο απαιτούμενος συγχρονισμός ποικίλουν. Στην συγκεκριμένη εργαστηριακή άσκηση θα επικεντρωθούμε σε τρείς από από αυτούς οι οποίοι είναι (α) Τα κλειδώματα(mutexes), οι σημαφόροι(semaphores) και οι μεταβλητές συνθήκης (condition variables) που ορίζονται από το POSIX και (β) οι ατομικές λειτουργίες (atomic operations), όπως αυτές ορίζονται από το υλικό και εξάγονται στον προγραμματιστή μέσω ειδκών εντολών (builtins) του μεταγλωττιστή GCC.

1.1. Συγχρονισμός σε υπάρχοντα κώδικα

Αντιγράφοντας τον έτοιμο κώδικα simplesync.c, παρατηρούμε ότι είναι ένα πρόγραμμα που δημιουργεί δυο νήματα, που το ενα αυξάνει κατά N φορές μια μεταβλητη val=0 και το άλλο μειώνει την μεταβλητή αυτή κατά N. Δίχως συγχρονισμό, στο τέλος του προγράμματος η μεταβλητή val δεν θα έχει την τιμή 0 όπως θα έπρεπε. Αυτό οφείλετε στο γεγονός ότι τρέχοντας ταυτόχρονα τα δύο νήματα, ο compiler μετατρέποντας τις εντολές αφαίρεσης και πρόσθεσης σε 3-4 εντολές assembly, τις συγχέει και επομένως δεν γίνονται με σωστή σειρά οι πράξεις στον επεξεργαστή. Αυτό λοιπόν θα επιτευχθεί κάνοντας κάθε κρίσιμο σημείο (πρόσθεση, αφαίρεση) ατομικά. Δηλαδή θα περιμένει το ένα νήμα να γίνει ατομικά η πράξη του άλλου νήματος. Επίσης άλλο ένα πρόβλημα που προκύπτει είναι ότι ο compiler αλλάζει την σειρά των πράξεων στο assembly κώδικα, και αυτό έχει κακή επίδραση στον συγχρονισμό πολλών νημάτων, καθώς πέρα από την ατομική εκτέλεση των κρίσιμων τμημάτων, θέλουμε και η πράξη της πρόσθεση ή αφαίρεσης μιας μεταβλητής να γίνει σε συνεχόμενους κύκλους επεξεργαστή, ώστε να παραμένει η ατομικότητα του τμήματος. Αυτό το επιτυγχάνουμε θέτοντας την μεταβλητή αυτή ως volatile. Πώς όμως θα καταφέρουμε να δηλώσουμε στον compiler ότι ένα τμήμα κώδικα είναι κρίσιμο τμήμα για μια διεργασία? Στην άσκηση αυτή το κάνουμε με την χρήση ενός mutex(κλειδώματος) και με την χρήση GCC atomic operation.

Χρησιμοποιώντας το Makefile για την μεταγλώττιση του προγράμματός μας, παρατηρούμε ότι από ένα αρχείο .c παράγονται δυο εκτελέσιμα αρχεία simplesync-atomic και simplesync-mutex.

Αυτό συμβαίνει διότι στην διαδικασία δημιουργίας των αντικείμενων αρχείων, χρησιμοποιούμε τις εντολές -DSYNC_ATOMIC (για το simplesync-atomic) και -DSYNC_MUTEX (για το simplesync-mutex). Και με την εντολή :

```
#if defined(SYNC_ATOMIC)
# define USE_ATOMIC_OPS 1
#else
# define USE_ATOMIC_OPS 0
#endif
```

όταν βάλουμε -DSYNC_ATOMIC, το use_atomic_ops γίνεται 1 και μπαίνει στο bracket της if που επιτελείται μια atomic operation, αντιθέτως χρησιμοποιούμε mutex όπως φαίνεται και στο τμήμα κώδικα που επισυνάπτεται, το οποίο φυσικά είναι και υλοποιημένο με τις αντίστοιχες μεθόδους συγχρονισμού.

```c
/*
 * simplesync.c
 *
 * A simple synchronization exercise.
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Operating Systems course, ECE, NTUA
 *
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
 * This macro helps with error reporting in this case.
 */
#define perror_pthread(ret, msg) \
        do { errno = ret; perror(msg); } while (0)

#define N 10000000

/* Dots indicate lines where you are free to insert code at will */
/* ... */
#if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
# error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
#endif

#if defined(SYNC_ATOMIC)
# define USE_ATOMIC_OPS 1
#else
# define USE_ATOMIC_OPS 0
#endif
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void *increase_fn(void *arg)
{
        int i;
        volatile int *ip = arg;

        fprintf(stderr, "About to increase variable %d times\n", N);
        for (i = 0; i < N; i++) {
                if (USE_ATOMIC_OPS) {
                        /* ... */
                        /* You can modify the following line */
                        __sync_fetch_and_add(ip,1);
                        /* ... */
                } else {
                        pthread_mutex_lock(&mutex);
                        /* ... */
                        /* You cannot modify the following line */
                        ++(*ip);
                        /* ... */
                        pthread_mutex_unlock(&mutex);
                }
        }
        fprintf(stderr, "Done increasing variable.\n");

        return NULL;
}

void *decrease_fn(void *arg)
{
        int i;
        volatile int *ip = arg;

        fprintf(stderr, "About to decrease variable %d times\n", N);
        for (i = 0; i < N; i++) {
                if (USE_ATOMIC_OPS) {
                        /* ... */
                        /* You can modify the following line */
                        __sync_fetch_and_add(ip,-1);
                        /* ... */
                } else {
                        pthread_mutex_lock(&mutex);
                        /* ... */
                        /* You cannot modify the following line */
                        --(*ip);
                        /* ... */
                        pthread_mutex_unlock(&mutex);
                }
        }
        fprintf(stderr, "Done decreasing variable.\n");

        return NULL;
}


int main(int argc, char *argv[])
{
        int val, ret, ok;
        pthread_t t1, t2;

        /*
         * Initial value
         */
        val = 0;

        /*
         * Create threads
         */
        ret = pthread_create(&t1, NULL, increase_fn, &val);
        if (ret) {
                perror_pthread(ret, "pthread_create");
                exit(1);
        }
        ret = pthread_create(&t2, NULL, decrease_fn, &val);
        if (ret) {
                perror_pthread(ret, "pthread_create");
                exit(1);
        }

        /*
         * Wait for threads to terminate
         */
        ret = pthread_join(t1, NULL);
        if (ret)
                perror_pthread(ret, "pthread_join");
        ret = pthread_join(t2, NULL);
        if (ret)
                perror_pthread(ret, "pthread_join");

        /*
         * Is everything OK?
         */
        ok = (val == 0);

        printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);

        return ok;
}
```

Το αποτέλεσμα του εκτελέσιμου προγράμματος για κάθε μια από τις δυο μεθόδους είναι το ακόλουθο:

```
oslabe01@os-node1:~/ask3/sync$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

```
oslabe01@os-node1:~/ask3/sync$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Το οποίο βέβαια είναι και το ζητούμενο αποτέλεσμα!!

ΕΡΩΤΗΣΕΙΣ:

1. Χρησιμοποιώντας την εντολή time πριν την εκτέλεση των δυο εκτελέσιμων, λαμβάνουμε έναν real χρόνο 3.707 sec για την simplesync-mutex και 0.413 sec για την simplesync-atomic, ενώ χωρίς τον συγχρονισμό, ο χρόνος εκτέλεσης είναι 0.038 sec. Παρατηρούμε ότι χωρίς συγχρονισμό το πρόγραμμα “τρέχει” πολύ πιο γρήγορα, καθώς ο compiler τρέχει στην βέλτιστη απόδοσή του, περιπλέκοντας τις εντολές του assembly κώδικα, που προφανώς είναι λογικό λάθος της άσκησης και βγάζει λάθος αποτέλεσμα.

2. Από τους προηγούμενους χρόνους παρατηρούμε ότι η μέθοδος των ατομικών λειτουργιών είναι γρηγορότερη από την μέθοδο των κλειδωμάτων. Αυτό οφείλετε στο γεγονός ότι οι ατομικές λειτουργίες είναι εντολές υλικού και απευθύνονται στο hardware, επομένως μια εντολή ατομική επιτελεί μια μόνο πράξη, αυτή της πρόσθεσης ή αφαίρεσης και φυσικά σε λίγους κύκλους επεξεργαστή. Αντιθέτως, ένα mutex υλοποιείται από atomic operations, ένα για την δημιουργία του και ένα για την λήξη του κλειδώματος, επομένως έχει τουλάχιστον διπλάσιο χρόνο εκτέλεσης. Επίσης το mutex δημιουργεί ένα τμήμα που ουσιαστικά μέσα σε αυτό μπορείς να κάνεις περισσότερα πράγματα απο μια πράξη, ενώ τα atomic operations εκτελούνται μια φορά και για μία μοναδική εντολή ξεχωριστά.

3. Μετατρέποντας λίγο το Makefile και βάζοντας την εντολή παραγωγής assembly κώδικα από το simplesync.c σε λειτουργία atomic operations, παράγουμε μια λίστα με τον assembly κώδικα και τον κώδικα μαζί (πηγή : internet). Με την εντολή:

```
gcc  -DSYNC_ATOMIC -g -S -c simplesync.c
```

παράγεται ο assembler κώδικας σε ένα αρχείο simplesync.s και παρατηρώντας το αρχείο αυτό βλέπουμε ότι οι ατομικές λειτουργίες μετατρέπονται στις εξής εντολές assembly:

```
.loc 1 75 0
movq    -16(%rbp), %rax
lock subl       $1, (%rax)
```

4. Κάνοντας την ίδια διαδικασία για τα mutexes αυτή τη φορά με εντολή :

```
simplesync.s: simplesync.c
	$(CC) $(CF:AGS) -DSYNC_MUTEX -g -S -c simplesync.c
```

παράγεται η εντολή assembly για το κλείδωμα του mutex :

```
.loc 1 52 0
movl    $mutex, %edi
call    pthread_mutex_lock
```

και για το ξεκλείδωμά του :

```
.loc 1 57 0
movl    $mutex, %edi
call    pthread_mutex_unlock
```

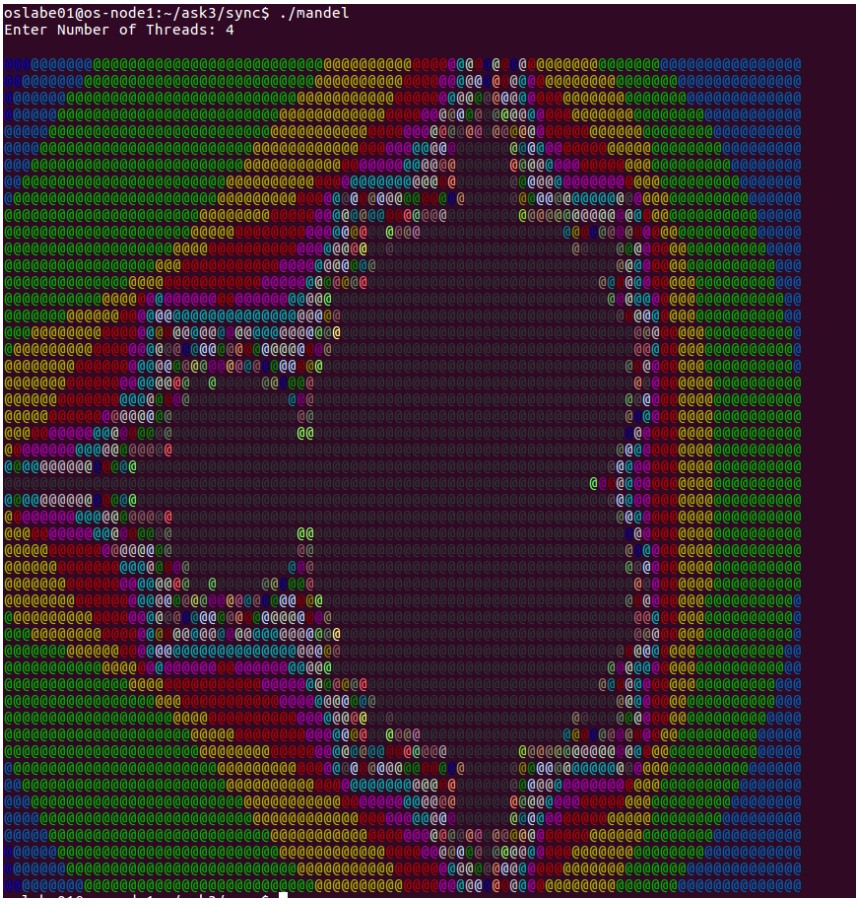
1.2. Παράλληλος υπολογισμός του συνόλου Mandelbrot

Σκοπός της συγκεκριμένης άσκησης είναι η δημιουργία N νημάτων, τα οποία θα δίνονται από τον χρήστη κατά την εκτέλεση του προγράμματος, τα οποία θα συνεργάζονται ώστε να φτιάξουν το σύνολο Mandelbrot, το οποίου υπολογίζεται από κάποιες έτοιμες συναρτήσεις. Η ανάθεση στα νήματα θα ακολουθεί το πρωτόκολλο ότι το νήμα i θα σχεδιάζει τις γραμμές i,i+N,i+2*N...Ο συγχρονισμός αυτός σε αυτή την περίπτωση πρέπει να επιτυγχάνεται με σημαφόρους από το πρότυπο POSIX. Ένας σημαφόρος όταν έχει "χωρητικότητα" ένα, τότε επιτρέπει σε ένα thread να μπει και να εκτελεστεί, ενώ όλα τα υπόλοιπα περιμένουν μέχρι ο σημαφόρος να " αδείασει" , " να αποκτήσει χώρο", "να γίνει =1", ώστε αν ενα άλλο thread ζητήσει να μπει, να μπορέσει να μπει. Ένας σημαφόρος δέχεται κάποιο thread με την εντολή sem_post(), ενώ αν κάποιο μπει, γίνεται sem_wait(). Η ιδέα της άσκησης, ώστε να έχουμε τον πιο γρήγορο χρόνο και με τον επιθυμητό συγχρονισμό, είναι να χρησιμοποιήσουμε N σημαφόρους, και κάθε φορά που ένα thread τελειώνει το κρίσιμο τμήμα υπολογισμού της γραμμής του, να αυξάνει το σημαφόρο του επόμενου thread, το οποίο θα μπαίνει στο κρίσιμο τμήμα του και θα εκτυπώνει και αυτό την γραμμή του, μέχρι να φτάσουμε στο τέλος.Ο κώδικας που υλοποιεί όλα τα παραπάνω είναι ο εξής :

```c
/*program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include <signal.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

/***************************
 * Compile-time parameters *
 ***************************/

#define perror_pthread(ret, msg) \
        do { errno = ret; perror(msg); } while (0)

void ResetAndExit(int sign)             /*if Ctrl-C stops the excecution reset the color of the terminal*/
{
        signal(sign, SIG_IGN);          //ignor signal
        reset_xterm_color(1);           //reset color
        exit(-1);                       //exit
}

/*
 * Output at the terminal is is x_chars wide by y_chars long
 */
int y_chars = 50;
int x_chars = 90;
int n;                                                          //Global metavlhth arithmou threads

typedef struct
{
        pthread_t tid;                                          //id gia kathe thread pou kanw create, oste na ginoun argotera join
        int l;                                                  //to antistoixo line
        sem_t mutex;                                            //to antistoixo semaphore metaksi line kai line+1
}mystruct;
mystruct *saved;                                                //malloc gia n sth main


/*
 * The part of the complex plane to be drawn:
 * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
 */
double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

/*
 * Every character in the final output is
 * xstep x ystep units wide on the complex plane.
 */
double xstep;

double ystep;

/*
 * This function computes a line of output
 * as an array of x_char color values.
 */
void compute_mandel_line(int line, int color_val[])
{
        /*
         * x and y traverse the complex plane.
         */
        double x, y;

        int n;
        int val;

        /* Find out the y value corresponding to this line */
        y = ymax - ystep * line;

        /* and iterate for all points on this line */
        for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {

                /* Compute the point's color value */
                val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
                if (val > 255)
                        val = 255;

                /* And store it in the color_val[] array */
                val = xterm_color(val);
                color_val[n] = val;
        }
}

/*
 * This function outputs an array of x_char color values
 * to a 256-color xterm.
 */
void output_mandel_line(int fd, int color_val[])
{
        int i;

        char point ='@';
        char newline='\n';

        for (i = 0; i < x_chars; i++) {
                /* Set the current color, then output the point */
                set_xterm_color(fd, color_val[i]);
                if (write(fd, &point, 1) != 1) {
                        perror("compute_and_output_mandel_line: write point");
                        exit(1);
                }
        }

        /* Now that the line is done, output a newline character */
        if (write(fd, &newline, 1) != 1) {
                perror("compute_and_output_mandel_line: write newline");
                exit(1);
        }
}
```

```
void *compute_and_output_mandel_line(void *arg){
        int k;                                          //Trexousa grammh
        int line=*(int*)arg;                            //Proth grammh tou Thread(0..n-1)
        for (k = line; k < y_chars; k += n){            //kai gia tis epomenes me vima n
                int color_val[x_chars];
                compute_mandel_line(k, color_val);      //Ypologismos k grammhs *parallhla
                //Sygxronismos:
                sem_wait(&saved[(k % n)].mutex);        //perimenoun to semaphore tou Thread tous.
                output_mandel_line(1, color_val);       //Print *sigxronismena.
                sem_post(&saved[((k % n)+1)%n].mutex);  //Auksanoun to semaphore tou epomenou Thread.
        }
        return 0;
}


int main(void){
        signal(SIGINT, ResetAndExit);                                   //An erthei Ctrl-C signal phgaine sthn ResetAndExit

        int line, ret;
        xstep = (xmax - xmin) / x_chars;
        ystep = (ymax - ymin) / y_chars;
        /*
         * draw the Mandelbrot Set, one line at a time.
         * Output is sent to file descriptor '1', i.e., standard output.
         */

        printf("Enter Number of Threads: ");                            //Diavazw ton arithmo twn Threads
        scanf("%d", &n);
        if ((n < 1) || (n > y_chars-1)){
                printf("invalid input \n");
                return -1;
        }
        printf("\n");
        saved = (mystruct*)malloc(n*sizeof(mystruct));                  //malloc
        sem_init(&saved[0].mutex, 0, 1);                                //To semaphore 0 ksekinaei prwto me value 1
        if (n > 1){                                                     //an uparxoun polla Threads
                for (line = 1; line < n; line++){                       //tha exo alla tosa semaphores
                        sem_init(&saved[line].mutex, 0, 0);             //me value 0 (waiting...)
                }
        }

        for (line = 0; line < n; line++) {                              //Crate n threads kai kathe ena pernaei kai to
                saved[line].l=line;                                     //antistoixo line gia na ksekinisei apo ekei
                ret = pthread_create(&saved[line].tid, NULL, compute_and_output_mandel_line, &saved[line].l);
                if (ret){
                        perror_pthread(ret, "pthread_create");          //error check
                        exit(1);
                }
        }

        for (line = 0; line < n; line++) {                              //Join ta threads sto telos
                ret = pthread_join(saved[line].tid, NULL);
                if (ret)
                        perror_pthread(ret, "pthread_join");            //error check
        }
        for (line = 0; line < n; line++) {                              //Kai destroy ta semaphores
                sem_destroy(&saved[line].mutex);
        }

        reset_xterm_color(1);
        return 0;
-- INSERT --
```

Η έξοδος του προγράμματος για όσα νήματα ζητηθούν, που να μην ξεπερνάνε φυσικά τις γραμμές του mandlebrot είναι το εξής:



ΕΡΩΤΗΣΕΙΣ:

1. Όπως προαναφέραμε, οι σημαφόροι που χρειάζονται είναι N, όσα είναι και τα threads που ζητάει ο χρήστης.

2. Κατά τον παράλληλο υπολογισμό των γραμμών μέσω 2 νημάτων απαιτείται real χρόνος 3.286 sec , ενώ στον σειριακό υπολογισμό του προγράμματος απαιτείται 1.022 sec

3. Έχοντας απο πρίν υλοποιήσει κατάλληλα τα threads, το παράλληλο πρόγραμμα εμφανίζει επιτάχυνση. Αυτό συμβαίνει διότι τον υπολογισμό των χρωμάτων της κάθε γραμμής δεν τον κάνουμε μέσα στο κρίσιμο τμήμα. Έτσι όλα τα threads όταν περιμένουν για να "ανοίξει" ο σημαφόρος τους, έχουν υπολογίσει ήδη τα χρώματα (διαδικασία που χρειάζεται χρόνο) και μόλις πάρουν άδεια να εκτελεστούν κάνουν απευθείας το τύπωμα. Αντιθέτως, αν ο υπολογισμός βρισκόταν μέσα στο κύριο τμήμα, τότε θα χάναμε κύκλους cpu για κάθε γραμμη ώστε να υπολογίζουμε το χρώμα τους και μετά να το εκτελούσαμε.

4. Αν πατήσουμε στο τερματικό Ctrl-C ενώ το πρόγραμμα εκτελείται, παρατηρούμε ότι το χρώμα των εντολών του τερματικού αλλάζουν. Αυτό συμβαίνει γιατί η συνάρτηση που υπολογίζει το χρώμα της γραμμής σταματάει βίαια και επομένως το χρώμα μετά του τερματικού παραμένει ίδιο με αυτό που έδειχνε το χρώμα του τελευταίου χαρακτήρα. Για να διορθώσουμε αυτό το γεγονός, προσθέτουμε στο πρόγραμμα την εντολή signal(SIGINT, ResetAndExit); , η οποία όταν γίνει βίαιος τερματισμός, κάνει reset τα χρώματα του τερματικού.