Λειτουργικά Συστήματα – 3η Εργαστηριακή Άσκηση

Ομάδα εργαστηρίου: **oslabb30**

Φοιτητές: **Γεώργιος Λαγός 03117034, Παναγιώτης Ζευγολατάκος 03117804**

Στην αναφορά χρησιμοποιείται παντού Α' πληθυντικός γιατί τόσο η αναφορά όσο και η ίδια η εργασία έγιναν με πλήρη επικοινωνία μεταξύ των φοιτητών.

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Στην άσκηση αυτή μας δίνεται μία πολυνηματική διεργασία, της οποίας τα νήματα δρουν σε μία κοινή μεταβλητή, με όνομα val και αρχική τιμή 0. Το ένα νήμα δρα επαναληπτικά στη μεταβλητή άθροισης με δέκα εκατομμύρια μοναδιαίες αυξήσεις ενώ το άλλο δρα επαναληπτικά στη μεταβλητή άθροισης με δέκα εκατομμύρια μοναδιαίες μειώσεις. Επομένως, αν και η λογική μας λέει πως το τελικό αποτέλεσμα είναι μηδέν, διαψευδόμαστε (βλ. Παρακάτω). Έτσι κατανοούμε τη σπουδαιότητα της παρουσίας **συγχρονισμού** σε πολυνηματικές διεργασίες με κοινή μοιραζόμενη μνήμη.

Ερωτήσεις:

1. *Χρησιμοποιήστε την εντολή time(1) για να μετρήσετε το χρόνο εκτέλεσης των εκτελέσιμων. Πώς συγκρίνεται ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό, σε σχέση με το χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό; Γιατί;*

   Με απουσία μηχανισμού συγχρονισμού λαμβάνουμε το ακόλουθο:

```
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 10089439.

real    0m1.233s
user    0m2.452s
sys     0m0.000s
```

   Όπως βλέπουμε το αποτέλεσμα δεν ισούται με μονάδα όπως περιμέναμε.

   Στην συνέχεια χρησιμοποιώντας μηχανισμό συγχρονισμού (GCC atomic operations) λάβαμε:

```
oslabb30@orion:~/giorgos/ask3$ time ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m1.144s
user    0m2.160s
sys     0m0.008s
```

Στην συνέχεια χρησιμοποιώντας μηχανισμό συγχρονισμού (POSIX MUTEXES) λάβαμε:

```
oslabb30@orion:~/giorgos/ask3$ time ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m3.012s
user    0m3.096s
sys     0m2.280s
```

Παρατηρούμε ότι ο χρόνος εκτέλεσης αυξάνεται όπως και είναι λογικό εφ' όσον υλοποιούμε σχήμα συγχρονισμού στην ενημέρωση των τιμών της κοινής μεταβλητής (αύξηση των εντολών που θα εκτελέσει η μηχανή).

2. *Ποια μέθοδος συγχρονισμού είναι γρηγορότερη, η χρήση ατομικών λειτουργιών ή η χρήση POSIX mutexes; Γιατί;*

Η μέθοδος με χρήση ατομικών λειτουργιών είναι αισθητά γρηγορότερη και μπορεί να βγει το συμπέρασμα χωρίς κιόλας να γίνει χρήση της εντολής time(1). Πιο αναλυτικά η χρήση gcc atomic operations, ως μέθοδος συγχρονισμού, αποτελεί μία επέμβαση σε επίπεδο υλικού, δηλαδή το lock για είσοδο του νήματος στο κρίσιμο τμήμα δίνεται μέσω μίας low – level τύπου εντολής. Αντίθετα, τα POSIX mutexes είναι μία υλοποίηση του σχήματος συγχρονισμού σε υψηλό επίπεδο, η οποία θα πρέπει να μετατραπεί σε low – level επίπεδο και η οποία χρησιμοποιεί τόσο για το κλείδωμα όσο και για το ξεκλείδωμα του κρίσιμου τμήματος από μία, κάθε φορά, κλήση συστήματος, προκαλώντας περαιτέρω χρονική επιβάρυνση. Τέλος, οι gcc

atomic operations, σαν low-level υλοποίηση εμφανίζουν θέματα φορητότητας και άρα δεν είναι πάντα μονόδρομος.

3. *Σε ποιες εντολές του επεξεργαστή μεταφράζεται η χρήση ατομικών λειτουργιών του GCC στην αρχιτεκτονική για την οποία μεταγλωττίζετε; Χρησιμοποιήστε την παράμετρο -S του GCC για να παράγετε τον ενδιάμεσο κώδικα Assembly, μαζί με την παράμετρο -g για να συμπεριλάβετε πληροφορίες γραμμών πηγαίου κώδικα (π.χ., ".loc 1 63 0"), οι οποίες μπορεί να σας διευκολύνουν. Δείτε την έξοδο της εντολής make για τον τρόπο μεταγλώττισης του simplesync.c.*

4. *Σε ποιες εντολές μεταφράζεται η χρήση POSIX mutexes στην αρχιτεκτονική για την οποία μεταγλωττίζετε; Παραθέστε παράδειγμα μεταγλώττισης λειτουργίας pthread_ mutex_lock() σε Assembly, όπως στο προηγούμενο ερώτημα.*

Για το τρίτο ερώτημα χρησιμοποιήσαμε την εντολή:

gcc –g –S –DSYNC_ATOMIC –pthread simplesync.c –o assembly

ενώ για το τέταρτο ερώτημα την εντολή

gcc –g –S –DSYNC_MUTEX –pthread simplesync.c –o assembly_mutex

Μελετώντας την υλοποίηση των δύο μηχανισμών συγχρονισμού διαπιστώνουμε τη στενή σύνδεση του πρώτου μηχανισμού με το υλικό, καθώς η μία εντολή που δίνουμε στη γλώσσα στη γλώσσα C μετασχηματίζεται σε μία εντολή σε συμβολική γλώσσα (και επομένως σε γλώσσα μηχανής). Ωστόσο, με το μηχανισμό POSIX mutexes, μετασχηματίζεται σε δέκα εντολές. Έτσι, αν και οι δύο μηχανισμοί πραγματοποιούν επιτυχή συγχρονισμό της πολυνηματικής μας διεργασίας, επιβεβαιώνεται η αναμενόμενη ταχύτερη εκτέλεση της διεργασίας χρησιμοποιώντας gcc atomic operations.

Παράθεση ενδιάμεσου κώδικα σε γλώσσα ASSEMBLY:

3)

```
increase_fn:
.LFB2:
    .file 1 "simplesync.c"
    .loc 1 41 0
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movq    %rdi, -24(%rbp)
    .loc 1 43 0
    movq    -24(%rbp), %rax
    movq    %rax, -16(%rbp)
    .loc 1 45 0
    movq    stderr(%rip), %rax
    movl    $10000000, %edx
    movl    $.LC0, %esi
    movq    %rax, %rdi
    movl    $0, %eax
    call    fprintf
    .loc 1 46 0
    movl    $0, -4(%rbp)
    jmp .L2
.L3:
    .loc 1 51 0
    movq    -16(%rbp), %rax
    lock addl   $1, (%rax)
    .loc 1 46 0
    addl    $1, -4(%rbp)
.L2:
    .loc 1 46 0 is_stmt 0 discrimina
    cmpl    $9999999, -4(%rbp)
    jle .L3
    .loc 1 62 0 is_stmt 1
    movq    stderr(%rip), %rax
    movq    %rax, %rcx
    movl    $26, %edx
    movl    $1, %esi
    movl    $.LC1, %edi
    call    fwrite
    .loc 1 64 0
    movl    $0, %eax
    .loc 1 65 0
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

```
decrease_fn:
.LFB3:
    .loc 1 68 0
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movq    %rdi, -24(%rbp)
    .loc 1 70 0
    movq    -24(%rbp), %rax
    movq    %rax, -16(%rbp)
    .loc 1 72 0
    movq    stderr(%rip), %rax
    movl    $10000000, %edx
    movl    $.LC2, %esi
    movq    %rax, %rdi
    movl    $0, %eax
    call    fprintf
    .loc 1 73 0
    movl    $0, -4(%rbp)
    jmp .L6
.L7:
    .loc 1 77 0
    movq    -16(%rbp), %rax
    lock subl   $1, (%rax)
    .loc 1 73 0
    addl    $1, -4(%rbp)
.L6:
    .loc 1 73 0 is_stmt 0 discriminator 1
    cmpl    $9999999, -4(%rbp)
    jle .L7
    .loc 1 88 0 is_stmt 1
    movq    stderr(%rip), %rax
    movq    %rax, %rcx
    movl    $26, %edx
    movl    $1, %esi
    movl    $.LC3, %edi
    call    fwrite
    .loc 1 90 0
    movl    $0, %eax
    .loc 1 91 0
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

4)

```
increase_fn:
.LFB2:
    .file 1 "simplesync.c"
    .loc 1 41 0
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movq    %rdi, -24(%rbp)
    .loc 1 43 0
    movq    -24(%rbp), %rax
    movq    %rax, -16(%rbp)
    .loc 1 45 0
    movq    stderr(%rip), %rax
    movl    $10000000, %edx
    movl    $.LC0, %esi
    movq    %rax, %rdi
    movl    $0, %eax
    call    fprintf
    .loc 1 46 0
    movl    $0, -4(%rbp)
    jmp .L2
.L3:
    .loc 1 54 0
    movl    $mutex, %edi
    call    pthread_mutex_lock
    .loc 1 57 0
    movq    -16(%rbp), %rax
    movl    (%rax), %eax
    leal    1(%rax), %edx
    movq    -16(%rbp), %rax
    movl    %edx, (%rax)
    .loc 1 59 0
    movl    $mutex, %edi
    call    pthread_mutex_unlock
    .loc 1 46 0
    addl    $1, -4(%rbp)
.L2:
    .loc 1 46 0 is_stmt 0 discrimina
    cmpl    $9999999, -4(%rbp)
    jle .L3
    .loc 1 62 0 is_stmt 1
    movq    stderr(%rip), %rax
    movq    %rax, %rcx
    movl    $26, %edx
    movl    $1, %esi
    movl    $.LC1, %edi
    call    fwrite
    .loc 1 64 0
    movl    $0, %eax
```

```
decrease_fn:
.LFB3:
    .loc 1 68 0
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $32, %rsp
    movq    %rdi, -24(%rbp)
    .loc 1 70 0
    movq    -24(%rbp), %rax
    movq    %rax, -16(%rbp)
    .loc 1 72 0
    movq    stderr(%rip), %rax
    movl    $10000000, %edx
    movl    $.LC2, %esi
    movq    %rax, %rdi
    movl    $0, %eax
    call    fprintf
    .loc 1 73 0
    movl    $0, -4(%rbp)
    jmp .L6
.L7:
    .loc 1 80 0
    movl    $mutex, %edi
    call    pthread_mutex_lock
    .loc 1 83 0
    movq    -16(%rbp), %rax
    movl    (%rax), %eax
    leal    -1(%rax), %edx
    movq    -16(%rbp), %rax
    movl    %edx, (%rax)
    .loc 1 85 0
    movl    $mutex, %edi
    call    pthread_mutex_unlock
    .loc 1 73 0
    addl    $1, -4(%rbp)
.L6:
    .loc 1 73 0 is_stmt 0 discriminator 1
    cmpl    $9999999, -4(%rbp)
    jle .L7
    .loc 1 88 0 is_stmt 1
    movq    stderr(%rip), %rax
    movq    %rax, %rcx
    movl    $26, %edx
    movl    $1, %esi
    movl    $.LC3, %edi
    call    fwrite
    .loc 1 90 0
    movl    $0, %eax
    .loc 1 91 0
    leave
```

Παράθεση του κώδικα που χρησιμοποιήθηκε για το πρώτο μέρος της εργασίας μας:

```
/* 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>

/*
 *   * 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_sub(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;
}
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

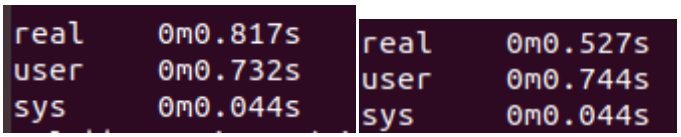
Ερωτήσεις:

1. Πόσοι σημαφόροι χρειάζονται για το σχήμα συγχρονισμού που υλοποιείτε;

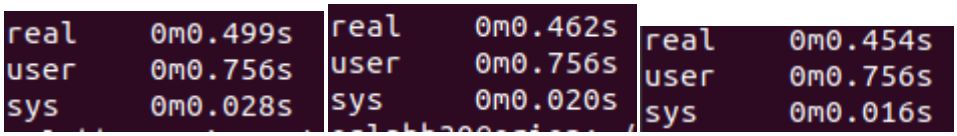
Για το σχήμα συγχρονισμού που υλοποιούμε χρειάζονται τόσοι δυαδικοί σημαφόροι όσα και τα νήματα της πολυνηματικής διεργασίας (δηλαδή N). Κάθε δυαδικός σημαφόρος έχει ταυτόσημο αποτέλεσμα συγχρονισμού με τα κλειδώματα αμοιβαίου αποκλεισμού (POSIX mutexes). Συγκεκριμένα, N-1 νήματα βρίσκονται σε λήθαργο, έχοντας μηδενική τιμή σημαφόρου ενώ ένα νήμα είναι ξύπνιο, έχοντας μοναδιαία τιμή σημαφόρου. Αυτό το (μοναδικό κάθε χρονική στιγμή) νήμα είναι προνομιούχο, καθώς ο μη μηδενικός του σημαφόρος το καθιστά δυνατό να εισέλθει στην κρίσιμη περιοχή.

2. Πόσος χρόνος απαιτείται για την ολοκλήρωση του σειριακού και του παράλληλου προγράμματος με δύο νήματα υπολογισμού; Χρησιμοποιήστε την εντολή time(1) για να χρονομετρήσετε την εκτέλεση ενός προγράμματος, π.χ., time sleep 2. Για να έχει νόημα η μέτρηση, δοκιμάστε σε ένα μηχάνημα που διαθέτει επεξεργαστή δύο πυρήνων. Χρησιμοποιήστε την εντολή cat /proc/cpuinfo για να δείτε πόσους υπολογιστικούς πυρήνες διαθέτει κάποιο μηχάνημα.

Παραθέτουμε τόσο το πρόγραμμα σειριακής υλοποίησης της εφαρμογής όσο και το πρόγραμμα παράλληλης υλοποίησης της και λαμβάνουμε τα εξής αποτελέσματα:



Αυξάνοντας τον αριθμό των νημάτων σε 3,4 και 5 αντίστοιχα λαμβάνουμε:



Παρατηρούμε το αναμενόμενο:

Η χρήση ολοένα και περισσότερων νημάτων αυξάνει την ταχύτητα εκτέλεσης.

3. *Το παράλληλο πρόγραμμα που φτιάξατε, εμφανίζει επιτάχυνση; Αν όχι, γιατί; Τι πρόβλημα υπάρχει στο σχήμα συγχρονισμού που έχετε υλοποιήσει; Υπόδειξη: Πόσο μεγάλο είναι το κρίσιμο τμήμα; Χρειάζεται να περιέχει και τη φάση υπολογισμού και τη φάση εξόδου κάθε γραμμής που παράγεται;*

Το παράλληλο πρόγραμμα που δομήσαμε εμφανίζει επιτάχυνση (βλ. Εικόνες παραπάνω). Το κρίσιμο τμήμα δε περιέχει τη φάση υπολογισμού καθώς ο υπολογισμός των χρωμάτων με βάση τη μιγαδική εξίσωση για κάθε γραμμή υπολογίζεται ανεξάρτητα (δεν χρησιμοποιείται μοιραζόμενος πόρος). Το κρίσιμο σημείο μας είναι η εκτύπωση των ήδη υπολογισμένων δεδομένων στην οθόνη με την σωστή σειρά.

4. *Τι συμβαίνει στο τερματικό αν πατήσετε Ctrl-C ενώ το πρόγραμμα εκτελείται; σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμάτων; Πώς θα μπορούσατε να επεκτείνετε το mandel.c σας ώστε να εξασφαλίσετε ότι ακόμη κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του;*

Εάν πατήσουμε Ctrl + C ενώ το πρόγραμμα εκτελείται εμφανίζεται το σήμα SIGINT και ο (signal handler) διακόπτει αυτόματα την εκτέλεση της διεργασίας, χωρίς επαναφορά

του τερματικού. Αυτό έχει ως αποτέλεσμα το τερματικό να αφεθεί με λανθασμένο χρώμα, αυτό του τρέχοντος χαρακτήρα που τυπώνεται από τη διεργασία, μόλις εμφανίστηκε το σήμα SIGINT. Αλλάζοντας τον χειριστή του σήματος πετυχαίνουμε το ζητούμενο: τερματίζεται η διεργασία αφότου πρώτα έχει γίνει κλήση της συνάρτησης reset_xterm_colour(1).

Παράθεση του κώδικα (τροποποιημένου) mandel.c:

```c
/*
 *  * mandel.c
 *   *
 *    * A 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 <errno.h>
#include <semaphore.h>
#include <signal.h>
#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

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


/*
 *  * Output at the terminal is is x_chars wide by y_chars long
 *  */
int y_chars = 50;
int x_chars = 90;

/*
 *  * 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;

int N;

struct thread_info_struct {
        pthread_t tid; //Posix thread id
        int *lines;    // Pointer to an array of integers who hold which LINES will be printed by this struct
        int number;    // Struct id in int coding: 0,1,2, etc
        sem_t my_sem; // each struct has its own semaphore, total N will be created
```

```
};

struct thread_info_struct *thr;

void *safe_malloc(size_t size)  // In case our attemtp to create the thread array fails to find the needed space in
{
        void *p;

        if ((p = malloc(size)) == NULL) {
                fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",size);
                exit(1);
        }

        return p;
}


/*
 *  * 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 */
                //      printf("Color value about to be inserted about is: %d\n", val);
                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);
                }
                reset_xterm_color(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);
        }
}

int next(int x){ return ((x+1) % N); }

void compute_and_output_mandel_line(int fd, int line)
{
        /*
         * A temporary array, used to hold color values for the line being drawn
         */
        int color_val[x_chars];
        int cur_thread = line % N; // analoga me to poia grammi tupwnetai 3eroume poio thread prepei na tin analavei

        compute_mandel_line(line, color_val);

        sem_wait(&(thr[cur_thread].my_sem));    // edw stop kanoume ta threads na ipologizoun parallila to ti prepei na tupwsoun all

        output_mandel_line(fd, color_val);   // kleidwame to semaphore tou current threa kai kanoume unclock to semaphore tou epomen

        sem_post(&(thr[next(cur_thread)].my_sem));
}


void *all_thread_start(void *arg)
{
        int i;
        struct thread_info_struct *cur = arg;

        for (i = 0; i < cur->number; i++){ // pame gia kathe thread na etoimasoume to ti prepei na tupwsei
                if(cur->lines[i] == -1)
                        continue;
                compute_and_output_mandel_line(1, cur->lines[i]);
        }
        return NULL;
}
```

```c
void find_lines_for_every_thread(int N, int i, int lines[], int number){
	int counter = 0;
	int j;
	for (j = 0; j < y_chars; j++){
		if ( j % N == i){		// bazoume ston pinaka pou tha exei kathe thread mesa tou poia lines ofeilei na ftia3ei kai na tupwsei
			lines[counter] = j;
			counter++;
		}
	}
	for (j = counter; j < number; j++){
		lines[j] = -1;
	}
}

void help(int sign) {
signal(sign,SIG_IGN);
reset_xterm_color(1);
printf("Control C detected\n");
exit(1);
}

int main(int argc, char **argv)
{

	if (argc != 2) {
		fprintf(stderr, "Incorrect number of arguments. Please specify exactly one argument, the number of threads to compute with\n");
		return 0;
	}
	N = atoi(argv[1]);
	if (N <= 0) {
		fprintf(stderr, "Invalid argument. Please specify an integer greater than 0\n");
		return 0;
	}

	int nr_of_lines = (y_chars / N) + 1;
	int ret;
	signal(SIGINT,help);
	thr = safe_malloc(N * sizeof(*thr));

	int i;
	xstep = (xmax - xmin) / x_chars;
	ystep = (ymax - ymin) / y_chars;
	for (i = 0; i < N; i++) {
		int *temp;
		temp = safe_malloc(nr_of_lines * sizeof(*temp));
		find_lines_for_every_thread(N,i,temp,nr_of_lines);
		thr[i].lines = temp;
		thr[i].number = nr_of_lines;
		if (i == 0) sem_init(&thr[i].my_sem, 0, 1);
		else sem_init(&thr[i].my_sem, 0, 0); // oloi oi shmaforoi ektos tou 1ou (midenikou) exoun timh 0, mono autos exei timh 1
		ret = pthread_create(&thr[i].tid, NULL,all_thread_start , &thr[i]);
		if (ret) {
			perror("pthread_create");
```

```
                }
        }

        for (i = 0; i < N; i++){
                ret = pthread_join(thr[i].tid, NULL);
                if(ret){
                        perror("pthread_join");
                        exit(1);
                }
        }

        /*
         * draw the Mandelbrot Set, one line at a time.
         * Output is sent to file descriptor '1', i.e., standard output.
         */
        int k;
        reset_xterm_color(1);
        for (k=0; k<N;k++) sem_destroy(&thr[k].my_sem);
        return 0;
}
```