ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ - ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Λειτουργικά Συστήματα

Ομάδα 118:

Ρέππας Ευστράτιος (03120002) – Τζουρμανά Ελευθερία (03119927)

Άσκηση 3: Συγχρονισμός

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Σε αυτήν την άσκηση μας ζητείται, στηριζόμενοι στο αρχείο simplesync.c, να συγχρονίσουμε δύο νήματα τα οποία εκτελούν ταυτόχρονα τις εξής λειτουργίες: το πρώτο νήμα αυξάνει N φορές την αρχικοποιημένη τιμή val, ενώ το δεύτερο την μειώνει κατά 1. Στόχος είναι να καταφέρουμε να τα συγχρονίσουμε με τέτοιο τρόπο ώστε, μετά την εκτέλεση του προγράμματος, να μας επιστρέφει τελικό αποτέλεσμα 0.

Προτού κάνουμε οποιαδήποτε επεξεργασία στον κώδικα μας, τον τρέξαμε (είτε το simplesync-atomic, είτε το simplesync-mutex) για να δούμε το αποτέλεσμα που θα μας δώσει. Αυτό που προσέξαμε είναι ότι δεν επέστρεφε σωστά την τελική τιμή, δηλαδή, όπως ήταν κι αναμενόμενο, τα νήματα δεν είχαν συγχρονιστεί.

Χρησιμοποιήσαμε το Makefile για να μεταγλωττίσουμε το πρόγραμμα μας, κι αυτό που παρατηρήσαμε είναι ότι με την εντολή make, παράγονται δύο εκτελέσιμα αρχεία, ένα με όνομα mutex και ένα άλλο με όνομα atomic, δηλαδή το simplesync-atomic και το simplesync-mutex. Αυτό κατέστη δυνατό με τη χρήση διαφορετικών flags στον μεταγλωττιστή, συγκεκριμένα, χρησιμοποιούνται το DSYNC_MUTEX και το DSYNC_ATOMIC τα οποία καθορίζουν ποια θα είναι η τιμή του USE ATOMIC OPS εντός του simplesync.c που λειτουργεί σαν flag για το αν θα χρησιμοποιηθούν mutexes ή atomic operations. Συγκεκριμένα, αν USE ATOMIC OPS λάβει την τιμή 1 γίνεται χρήση ατομικών μεταβλητών αλλιώς χρήση mutex. Τέλος, εντός του κώδικά

μας υπάρχουν τα αντίστοιχα τμήματα που ενεργοποιούνται ανάλογα με ποια σημαία χρησιμοποιήσουμε.

```
oslabl18@orion:~/ask3$ ls
kgarten     mandel.c      pthread-test       simplesync-atomic.o
kgarten.c   mandel-lib.c  pthread-test.c     simplesync.c
kgarten.o   mandel-lib.h  pthread-test.o     simplesync-mutex
Makefile    mandel-lib.o  rand-fork.c        simplesync-mutex.o
mandel      mandel.o      simplesync-atomic
oslabl18@orion:~/ask3$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 2246451.
oslabl18@orion:~/ask3$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 2602336.
oslabl18@orion:~/ask3$ ./simplesync
-bash: ./simplesync: No such file or directory
oslabl18@orion:~/ask3$ make simplesync
gcc -Wall -O2 -pthread    simplesync.c    -o simplesync
simplesync.c:30:3: error: #error You must #define exactly one of SYNC_ATOMIC or
SYNC_MUTEX.
 # error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
   ^
<builtin>: recipe for target 'simplesync' failed
make: *** [simplesync] Error 1
```

Επεκτείνουμε λοιπόν τον παραπάνω κώδικα, έτσι ώστε να καταφέρουμε να συγχρονίσουμε τα 2 νήματα και κατ' επέκταση η τιμή της μεταβλητής val να είναι 0. Χρησιμοποιούμε GCC atomic operations και POSIX mutexes. Αφού κάναμε τις απαραίτητες αλλαγές στον κώδικα μας, η έξοδος εκτέλεσης φαίνεται παρακάτω. Η τιμή val παραμένει 0 άρα το αποτέλεσμα του συγχρονισμού μας είναι σωστό.

```
oslabl18@orion:~/ask3$ ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
oslabl18@orion:~/ask3$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
oslabl18@orion:~/ask3$
```

Ο κώδικας που χρησιμοποιήσαμε είναι ο εξής:

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
```

```c
/*
 * 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 //πόσες φορές θα αφαιρέσει το ένα νήμα και θα
αυξηθεί το άλλο

/* 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 //αν δοθεί η σημαία SYNC_ATOMIC να
χρησιμοποιηθεί μια ατομική μεταβλητή

#else
# define USE_ATOMIC_OPS 0 //αλλιώς 0

#endif

pthread_mutex_t lock;

void *increase_fn(void *arg) //διαδικασία αύξησης κατά 1 την φορά
{
        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) {
//διασφαλίζεται ο αμοιβαίος αποκλεισμός
                        /* ... */
                        __sync_add_and_fetch(ip,1);
//συνάρτηση για atomic operation
//προσθέτει αυτόματα μια τιμή, το ip
//1ο όρισμα είναι ο δείκτης μιας μεταβλητής που θέλουμε να αυξήσουμε
//κατά 1(2ο όρισμα)
                        /* You can modify the following line */
                      ++(*ip);
```

```
                        /* ... */
                } else {
                        /* ... */
                        while (pthread_mutex_lock(&lock));
//ενεργοποίηση κλειδώματος
                        /* You cannot modify the following line */
                        ++(*ip);
                        /* ... */
                        pthread_mutex_unlock(&lock);
//αποδέσμευση κλειδώματος
                }
        }

        fprintf(stderr, "Done increasing variable.\n");

        return NULL;
}

void *decrease_fn(void *arg) //αντίστοιχη διαδικασία αλλά για μείωση

{
        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) {
                        /* ... */
                        __sync_sub_and_fetch(ip,1);
                        /* You can modify the following line */
                     --(*ip);
                        /* ... */
                } else {
                        /* ... */
                        while(pthread_mutex_lock(&lock));
                        /* You cannot modify the following line */
                        --(*ip);
                        /* ... */
                        pthread_mutex_unlock(&lock);
                }
        }

        fprintf(stderr, "Done decreasing variable.\n");

        return NULL;
}
```

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

{
        if (!USE_ATOMIC_OPS) //αν δεν χρησιμοποιούνται ατομικές
μεταβλητές
                if (pthread_mutex_init(&lock, NULL)){
//αρχικοποιούμε έναν mutex επειδή το 2ο όρισμα είναι NULL, mutex έχει
default χαρακτηριστικά
                        perror("mutex init");
                        return 1;
                }
        int val, ret, ok;
        //val η τιμή που θέλουμε να μας επιστρέψει ως τελικό αποτέλεσμα
        //ret η τιμή που επιστρέφει κάθε νήμα
        //ok ελέγχος για το αν επιστράφηκε η σωστή τιμή
        pthread_t t1, t2;

        /*
         * Initial value
         */
        val = 0;

        /*
         * Create threads
         */
        ret = pthread_create(&t1, NULL, increase_fn, &val);
//δημιουργία 1ου νήματος
        if (ret) {
                perror_pthread(ret, "pthread_create");
                exit(1);
        }
        ret = pthread_create(&t2, NULL, decrease_fn, &val);
//δημιουργία 2ου νήματος
        if (ret) {
                perror_pthread(ret, "pthread_create");
                exit(1);
        }

        /*
         * Wait for threads to terminate
         */
        ret = pthread_join(t1, NULL);
//σταματάει την εκτέλεση του νήματος μέχρι το t1 (1ο νήμα, δηλαδή το
νήμα-στόχος και 1ο όρισμα) να τερματίσει

        if (ret)
                perror_pthread(ret, "pthread_join");
        ret = pthread_join(t2, NULL);
```

```
        if (ret)
                perror_pthread(ret, "pthread_join");

        pthread_mutex_destroy(&lock); //καταστρέφουμε τα mutex
        /*
         * Is everything OK?
          */
        ok = (val == 0); //ελέγχουμε αν η τιμή που μας επιστράφηκε
είναι η αναμενόμενη

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

        return ok;

}
```

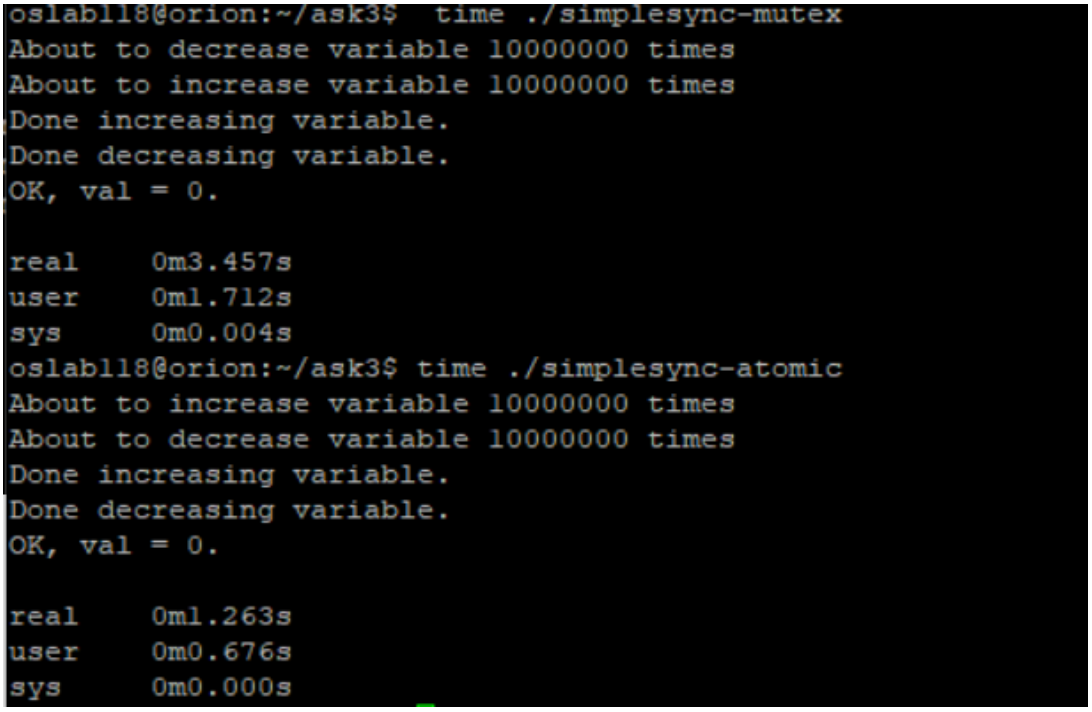
Ερωτήσεις:

1. *Χρησιμοποιήστε την εντολή time 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;*

   - Οι χρόνοι πριν τον συγχρονισμό:

```
oslab118@orion:~/ask3$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 1319611.

real    0m0.298s
user    0m0.144s
sys     0m0.000s
oslab118@orion:~/ask3$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -1145996.

real    0m0.326s
user    0m0.160s
sys     0m0.000s
```

- Οι χρόνοι μετά τον συγχρονισμό:



Παρατηρούμε ότι ο χρόνος εκτέλεσης για τον συγχρονισμό είναι μεγαλύτερος σε σχέση με το χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό. Αυτό συμβαίνει διότι στην δεύτερη περίπτωση, το κάθε νήμα εκτελεί παράλληλα με το άλλο τις πράξεις του, δηλαδή οι εντολές increase και decrease εκτελούνται ταυτόχρονα, αφού δεν είναι συγχρονισμένα.

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

Παρατηρούμε από τους παραπάνω χρόνους, μετά τον συγχρονισμό, ότι η χρήση ατομικών λειτουργιών επιστρέφει πιο γρήγορα το αποτέλεσμα του προγράμματος από ότι με τη χρήση των POSIX mutexes. Αυτό οφείλεται στον μηχανισμό κάθε λειτουργίας. Συγκεκριμένα, τα mutexes κάνουν τον συγχρονισμό δυνατό με τέτοιο τρόπο ώστε να εμποδίζει την ταυτόχρονη πρόσβαση των κοινών δεδομένων από πολλά νήματα, για αυτό χρησιμοποιούμε τη δέσμευση/αποδέσμευση κλειδώματος η οποία έχει επιπλέον χρονικό κόστος από το λειτουργικό σύστημα. Στην περίπτωση των ατομικών λειτουργιών, εκτελείται μία πράξη ακεραίου (αύξηση /μείωση ακεραίου), χωρίς να απαιτείται η χρήση μηχανισμών του κλειδώματος, καθώς περισσότεροι επεξεργαστές διαθέτουν ειδικές αρχιτεκτονικές υπολογισμού ατομικών ακεραίων. Επιπλέον, οι ατομικές λειτουργίες είναι αδιαίρετες, και εκτελούν τις λειτουργίες που τους έχουν ανατεθεί ανεξάρτητα από τις υπόλοιπες. Επιτρέπουν, με άλλα λόγια, ατομικές προσβάσεις και αλλαγές κοινών μεταβλητών, δηλαδή, είναι σαν να εφαρμόζουν σειριακή πρόσβαση στις κοινές μεταβλητές αυτές. Όλα αυτά συμβαίνουν διότι τα atomic operations υλοποιούνται κατευθείαν στο υλικό, ενώ, τα mutexes υλοποιούνται με εντολές λογισμικού, συμπεριλαμβανομένης της συμβολής του λειτουργικού συστήματος (για να επιτελούν τις παραπάνω λειτουργίες), το οποίο

καθυστερεί περισσότερο την εκτέλεση του προγράμματος. Επομένως, συμφέρει η χρήση των atomic operations σε περιπτώσεις σαν αυτές του συγκεκριμένου προβλήματος, ενώ τα mutexes για πιο πολύπλοκες λειτουργίες καθώς είναι πιο ευρείας χρήσεως.

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.*

Η εντολή που χρησιμοποιήσαμε για να παράγουμε τον ενδιάμεσο κώδικα Assembly καθώς και οι εντολές στις οποίες μεταφράζεται η χρήση ατομικών λειτουργιών του gcc φαίνονται παρακάτω:

gcc -Wall -O2 -pthread -S -g -DSYNC_ATOMIC -o simplesync-atomic.s simplesync.c

```
.loc 1 53 0
        lock addl       $1, (%rbx) //αύξηση


 .loc 1 83 0
        lock subl       $1, (%rbx) //μείωση
```

Παρατηρούμε λοιπόν πως οι atomic operations υλοποιούνται πολύ αποδοτικά, μόλις με μία εντολή assembly, για τους λόγους που εξηγήσαμε παραπάνω.

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.*

Όμοια, η εντολή που χρησιμοποιήσαμε για να παράγουμε τον ενδιάμεσο κώδικα Assembly καθώς και οι εντολές στις οποίες μεταφράζεται η χρήση ατομικών λειτουργιών του gcc φαίνονται παρακάτω

gcc -Wall -O2 -pthread -S -g -DSYNC_MUTEX -o simplesync-mutex.s simplesync.c

```
.L2:
        .loc 1 62 0 discriminator 1
//κλήση pthread_mutex_lock
        movl    $lock, %edi
        call    pthread_mutex_lock
.LVL4:
        testl   %eax, %eax
        jne     .L2
        .loc 1 64 0
        movl    0(%rbp), %eax
        .loc 1 66 0
```

```
        movl    $lock, %edi
        .loc 1 64 0
        addl    $1, %eax
        movl    %eax, 0(%rbp)
        .loc 1 66 0
        call    pthread_mutex_unlock


.LVL14:
        testl   %eax, %eax
        jne     .L9
        .loc 1 91 0

//κλήση pthread_mutex_unlock
        movl    0(%rbp), %eax
        .loc 1 93 0
        movl    $lock, %edi
        .loc 1 91 0
        subl    $1, %eax
        movl    %eax, 0(%rbp)
        .loc 1 93 0
        call    pthread_mutex_unlock
```

Η κλήση pthread_mutex_lock: καλεί την συνάρτηση από την υλοποίηση της βιβλιοθήκης καθώς και περεταίρω ατομικές εντολές assembly, τις movl, subl και μία jump μετά από testl. Είναι προφανής λοιπόν η μεγαλύτερη πολυπλοκότητα των mutexes σε σχέση με τα atomic operations.

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Το πρόγραμμα που μας δίνεται υπολογίζει και εξάγει στο τερματικό εικόνες του συνόλου του Mandelbrot το οποίο ορίζεται ως το σύνολο των σημείων c του μιγαδικού επιπέδου, για τα οποία η ακολουθία $z_{n+1} = z_n^2 + c$ είναι φραγμένη.

Ζητείται η επέκταση του προγράμματος mandel.c έτσι ώστε ο υπολογισμός να κατανέμεται σε NTHREADS νήματα POSIX, ενδεικτική τιμή 3. Η κατανομή του υπολογιστικού φόρτου γίνεται ανά σειρά: Για n νήματα, το i-ιοστό (με i = 0, 1, 2, . . . , n−1) αναλαμβάνει τις σειρές i, i + n, i + 2 × n, i + 3 × n, . . .

Υλοποιούμε 2 εκδοχές του προγράμματος όπου ο απαραίτητος συγχρονισμός των νημάτων θα γίνεται: 1. με σημαφόρους που παρέχονται από το πρότυπο POSIX και 2. με μεταβλητές συνθήκης (condition variables) που παρέχονται από το πρότυπο POSIX.

1. Ο κώδικας που χρησιμοποιήσαμε για την υλοποίηση με σημαφόρους είναι καθώς και η έξοδος του προγράμματος είναι οι εξής:

```c
#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 <signal.h>
#include <errno.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

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

struct thread_info_struct {
    pthread_t tid; /* POSIX thread id, as returned by the library */
    int thrid; /* Application-defined thread id */
};

int safe_atoi(char *s, int *val) {
    long l;
    char *endp;

    l = strtol(s, &endp, 10);
    if (s != endp && *endp == '\0') {
        *val = l;
        return 0;
    } else
        return -1;
```

```
}

void *safe_malloc(size_t size) {
    void *p;

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

    return p;
}

/**************************
 * 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;

/*
 * This function computes a line of output
 * as an array of x_char color values.
 */
```

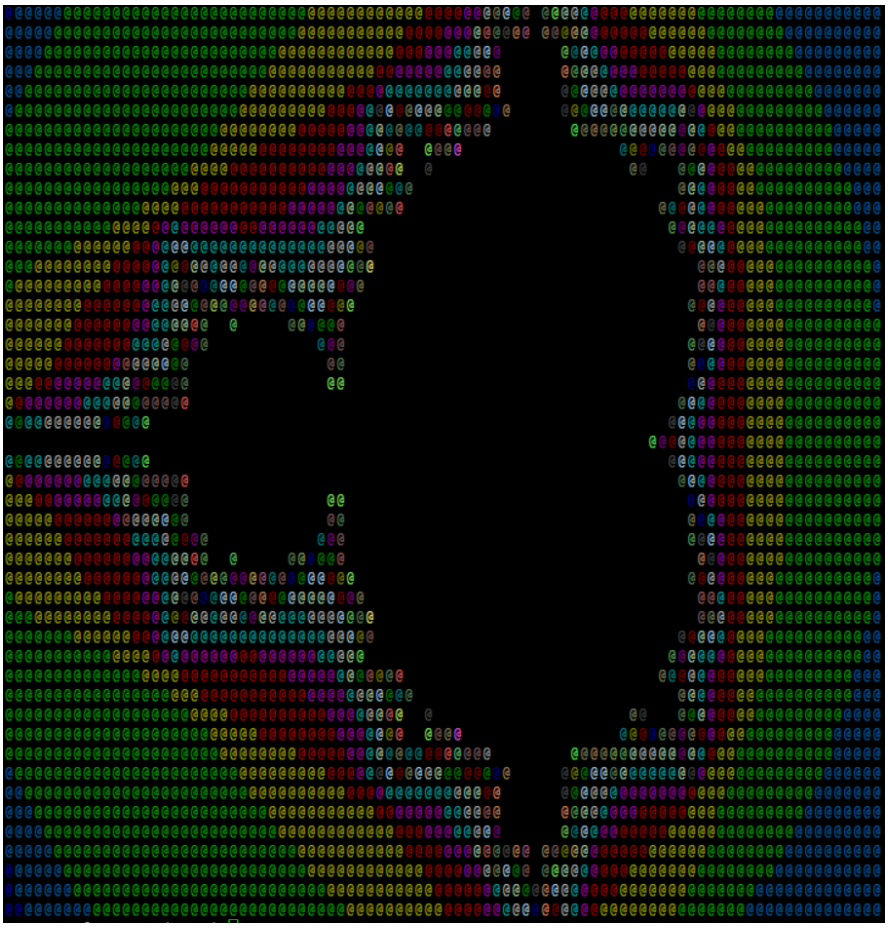
```
void compute_mandel_line(int line, int color_val[]) {
    /*
     * x and y traverse the complex plane.
     */
    double x, y;

    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);
    }
}

/* The next function creates the Mandelbrot area by using the previous
two functions.
 * This is the part that we will accelerate using threads. The specific
process that can be accelerated via parallelization
 * is the computation of the color of each line. However, the lines must
be printed in a specific order and therefore cannot
 *  be  accelerated  (critical  section).  That  is  where  we  will  use
semaphores.
 */

// We initialize the global variables for the threads (the variables that
are being accessed by every thread)
sem_t *sem;
int nrthreads;

void compute_and_output_mandel_line(void *in) {
    /*
     * A temporary array, used to hold color values for the line being
drawn
     */
    struct thread_info_struct *thr = in;
    int line, color_val[x_chars];

    /*
     * We begin an iteration for this thread. The line variable signifies
the  line  that  the  thread  computes  in  each  iteration.  It  increases  by
nrthreads  after  each  iteration,  since  the  in  between  lines  will  be
computed by the other threads. It continues until the line limit has been
met or surpassed.
     */

    for (line = thr->thrid; line < y_chars; line += nrthreads) {
        compute_mandel_line(line, color_val);

//The computations can happen simultaneously by the threads.

/*
         * Creating the lock using semaphores.
         * Initially, the semaphore value of the first thread will be 1
and the rest 0.
         * Function sem_wait() reduces the semaphore value by 1, so it
becomes 0.
         * The thread runs the critical section and then, when the thread
ends its operation,
```

```
         * the value of the semaphore is incremented by 1, so now
         * the next thread can access the critical code. The thread that
just finished running this section, in
          * the next iteration it will be locked out of the critical
section, since its semaphore is 0 and
         * therefore cannot be reduced.
          * It must wait for another thread to unlock its semaphore in
same manner for it to continue running and access
         * the critical section.
         */

        if (sem_wait(&sem[thr->thrid]) < 0) {
//lock (reduce the semaphore of this thread to 0)
            perror("sem_wait");
            exit(1);
        }

        output_mandel_line(1, color_val);
//The critical code: the output of the line

        if(sem_post(&sem[(thr->thrid+1) % nrthreads])<0) {
//unlock (increase the semaphore of the next thread to 1)
            perror("sem_post");
            exit(1);
        }
    }
}

void sigint_handler(int signum){
        reset_xterm_color(1);
// Reset the terminal color to its original state
        exit(1);
} //signal handler is a function that gets called in response to the
occurrence of a specific signal

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

    int i, ret;

    printf("mandel executing\n");

    xstep = (xmax - xmin) / x_chars;
    ystep = (ymax - ymin) / y_chars;

    if(argv[1]==NULL) argv[1]="3"; //check

    if((argc>2)||(safe_atoi(argv[1], &nrthreads)<0)||(nrthreads<=0)) {
```

```
        fprintf(stderr, "Usage: %s thread_count (>0,
default:3)\n",argv[0]);
        exit(1);
    }
 // we use a signal handler for the SIGINT signal (which is generated
when the user presses Ctrl+C)
    struct sigaction sa;
    sigset_t sigset;
    sa.sa_handler=sigint_handler;
    sa.sa_flags=SA_RESTART;// Set the flags for the signal action
(SA_RESTART enables restarting interrupted system calls)
    sigemptyset(&sigset);
    sa.sa_mask= sigset; //specifies a signal mask that indicates which
signals should be blocked (i.e., temporarily ignored) while the signal
handler is executing
    if(sigaction(SIGINT, &sa, NULL)<0) {
            perror("sigaction");
            exit(1);
        }
   sem = safe_malloc(nrthreads * sizeof(*sem));

    if(sem_init(&sem[0], 0, 1)<0) { //Set the semaphore of the first
thread to 1 (unlocked)
        perror("sem_init");
        exit(1);
    }

    for(i=1; i<nrthreads; i++) { //Set the semaphores of the other
threads to 0 (locked)
        if(sem_init(&sem[i], 0, 0)<0) {
            perror("sem_init");
            exit(1);
        }
    }

    struct thread_info_struct *thr;
    thr = safe_malloc(nrthreads * sizeof(*thr));

     for(i=0; i<nrthreads; i++) {
        thr[i].thrid=i;
        ret=pthread_create(&thr[i].tid, NULL,
compute_and_output_mandel_line, &thr[i]);
        if(ret) {
            perror_pthread(ret, "pthread_create");
            exit(1);
        }
    }
```

```
    // We wait for all the threads to terminate

    for(i=0; i<nrthreads; i++) {
        ret=pthread_join(thr[i].tid, NULL);
        if(ret)
            perror_pthread(ret, "pthread_join");
    }


    // We destroy all the semaphores

    for(i=0; i<nrthreads; i++) {
        if(sem_destroy(&sem[i])<0) {
            perror("sem_destroy");
            exit(1);
        }
    }
    free(sem);

    reset_xterm_color(1);

    return 0;
}
```



2. Ο κώδικας για την υλοποίηση με condition variables καθώς και η έξοδος εκτέλεσης είναι οι εξής:

```
#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 <signal.h>
#include <errno.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

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

struct thread_info_struct {
    pthread_t tid; /* POSIX thread id, as returned by the library */
    int thrid; /* Application-defined thread id */
};

int safe_atoi(char *s, int *val) {
    long l;
    char *endp;

    l = strtol(s, &endp, 10);
    if (s != endp && *endp == '\0') {
        *val = l;
        return 0;
    } else
        return -1;
}

void *safe_malloc(size_t size) {
    void *p;

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

    return p;
```

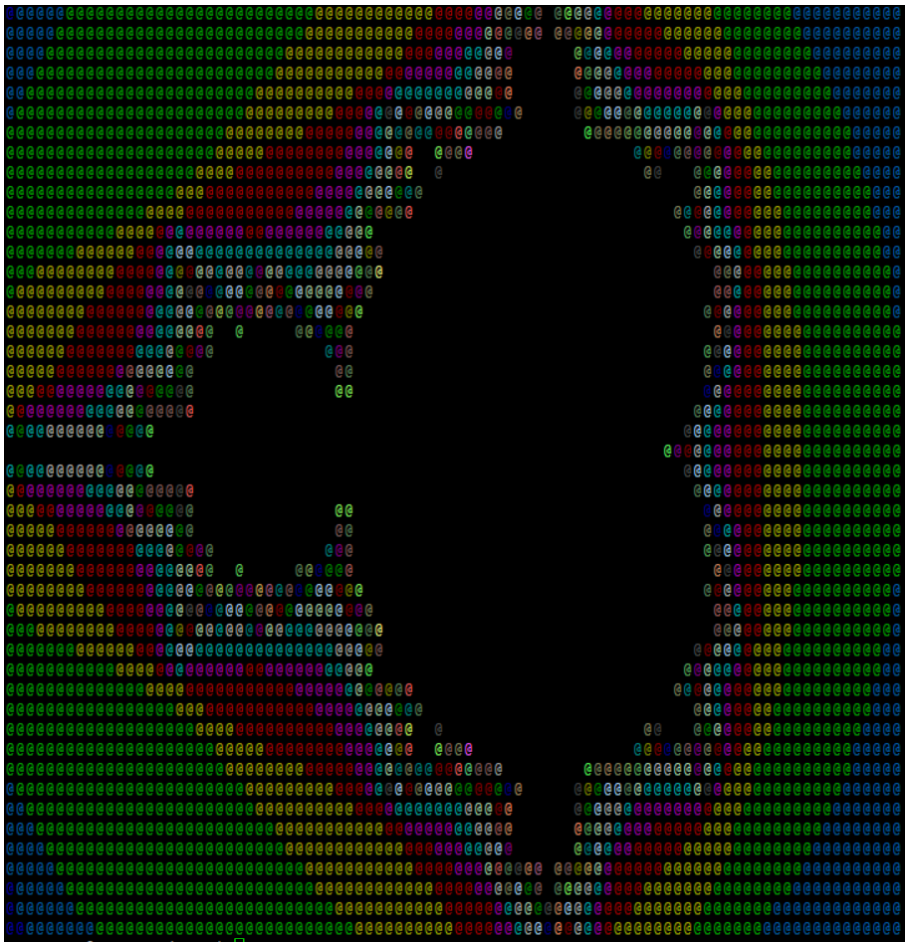
```
}

/**************************
 * 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;

/*
 * 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);
    }
}

/* The next function creates the Mandelbrot area by using the previous
two functions.
 * This is the part that we will accelerate using threads. The specific
process that can be accelerated via parallelization
 * is the computation of the color of each line. However, the lines
must be printed in a specific order and therefore cannot
 * be accelerated (critical section). That is where we will use
condition variables.
 */
```

```
// We initialize the global variables for the threads (the variables
that are being accessed by every thread)
int nrthreads;
pthread_mutex_t lock;
pthread_cond_t cond;
int j=0;


void compute_and_output_mandel_line(void *in) {
    /*
     * A temporary array, used to hold color values for the line being
drawn
     */
    struct thread_info_struct *thr = in;
    int line, color_val[x_chars];

    /*
     * We begin an iteration for this thread. The line variable signifies
the line that the thread computes in each iteration. It increases by
nrthreads after each iteration, since the in between lines will be
computed by the other threads. It continues until the line limit has been
met or surpassed.
     */

    for (line = thr->thrid; line < y_chars; line += nrthreads) {

        compute_mandel_line(line, color_val); //The computations can
happen simultaneously by the threads.

        /*
         * Creating the lock using conditional variables.
         * We signify the critical section using mutexes. These are locked
in each iteration (using the pthread_cond_wait() function) until a
condition is met. When a thread ends its critical code, it broadcasts the
condition for the all the threads to enter the critical section. The
synchronization is achieved via a global variable j. When j=line, the
thread can run. Each time the critical code runs, j increments by one,
so the next line will be plotted
         */

        pthread_mutex_lock(&lock); // Lock the mutex to enter the
critical section

        while(line!=j)
            pthread_cond_wait(&cond, &lock);

            // Wait until the condition is signaled
            j++;
```

```
            output_mandel_line(1, color_val); //The critical code: the
output of the line

            pthread_cond_broadcast(&cond);// Signal all threads to
enter the critical section. Only the thread of the corresponding line
will execute due to the while() loop, the rest will wait to be awaken
again
            pthread_mutex_unlock(&lock); // Unlock the mutex to exit
the critical section

    }
return NULL;
}



int main(int argc, char **argv) {
    int i, ret;
    printf("mandel_cond executing\n");
    xstep = (xmax - xmin) / x_chars;
    ystep = (ymax - ymin) / y_chars;

    if(argv[1]==NULL) argv[1]="3"; //check

    if((argc>2)||(safe_atoi(argv[1], &nrthreads)<0)||(nrthreads<=0)) {
        fprintf(stderr, "Usage: %s thread_count (>0,
default:3)\n",argv[0]);
        exit(1);
    }

    pthread_cond_init(&cond,NULL);
    pthread_mutex_init(&lock,NULL);

    struct thread_info_struct *thr;
    thr = safe_malloc(nrthreads * sizeof(*thr));

    for(i=0; i<nrthreads; i++) {
        thr[i].thrid=i;
        ret=pthread_create(&thr[i].tid, NULL,
compute_and_output_mandel_line, &thr[i]);
        if(ret) {
            perror_pthread(ret, "pthread_create");
            exit(1);
        }
    }

    // We wait for all the threads to terminate

    for(i=0; i<nrthreads; i++) {
```

```
        ret=pthread_join(thr[i].tid, NULL);
        if(ret)
            perror_pthread(ret, "pthread_join");
    }
    // We destroy all the locks and conditions

    if(pthread_mutex_destroy(&lock)<0) {
        perror ("mutex_destroy");
        exit (1);
    }


        if(pthread_cond_destroy(&cond)<0) {
            perror("cond_destroy");
            exit(1);
        }

   reset_xterm_color(1);
   return 0;
}
```



Ερωτήσεις:

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

Χρειάζονται τόσοι σημαφόροι όσα είναι και τα νήματα, καθώς θα πρέπει να σταλθεί σημαφόρος σε κάθε νήμα ξεχωριστά προκειμένου να τεθεί σε αναμονή ή να συνεχίσει την λειτουργία του. Ειδικότερα, ένας σημαφόρος είναι μία ακέραιη μεταβλητή η οποία προσπελαύνεται μόνο μέσω δύο πρότυπων ατομικών λειτουργιών. Οι τροποποιήσεις της ακέραιας τιμής του σημαφόρου πρέπει να εκτελούνται ατομικά. Δηλαδή, όταν μία διεργασία τροποποιεί την τιμή του σημαφόρου, δεν μπορεί κάποια άλλη διεργασία να τροποποιήσει την τιμή του ίδιου σημαφόρου ταυτόχρονα. Ακόμα, οι σημαφόροι δεν μπορούν να πάρουν αρνητικές τιμές, επομένως αν ένας σημαφόρος έχει τιμή 0 και καλείται εντολή μείωσης του, το thread θα περιμένει μέχρι να αυξηθεί (από το προηγούμενο thread στην άσκησή μας, γι' αυτό και χρειαζόμαστε ένα για κάθε thread), ώστε να είναι σε θέση να το μειώσει. Τους αρχικοποιούμε όλους στην τιμή 0 που αντιστοιχεί σε locked εκτός από εκείνον που αντιστοιχεί στο thread '0' που τον αρχικοποιούμε στην τιμή 1, δηλαδή είναι unlocked. Με αυτόν τον τρόπο, ο πρώτος σημαφόρος επιτρέπει στο πρώτο νήμα να εκτελέσει τον κρίσιμο κώδικα, ενώ οι υπόλοιποι σημαφόροι οι οποίοι έχουν αρχική τιμή 0 «κλειδωμένοι», απελευθερώνονται από το προηγούμενο νήμα που ολοκλήρωσε την εκτέλεσή του. Αν δεν έχει δοθεί κάποιος αριθμός νημάτων ως παράμετρος εκτέλεσης του προγράμματος, ο αριθμός νημάτων ορίζεται by default ως 3.

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.*

Εκτελούμε την εντολή όπως ζητήθηκε *(αποτυπώνεται το τελευταίο τμήμα της εντολής)*:

```
processor       : 7
vendor_id       : AuthenticAMD
cpu family      : 6
model           : 6
model name      : QEMU Virtual CPU version 2.5+
stepping        : 3
microcode       : 0x1000065
cpu MHz         : 2400.028
cache size      : 512 KB
physical id     : 7
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 7
initial apicid  : 7
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 syscall nx lm nopl cpuid tsc_known_freq pni cx
16 x2apic hypervisor lahf_lm 3dnowprefetch vmmcall
bugs            : fxsave_leak sysret_ss_attrs null_seg spectre_v1 spectre_v2 spec_store_bypass
bogomips        : 4800.05
TLB size        : 1024 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
```

Βλέπουμε δηλαδή ότι έχουμε 8 επεξεργαστές, με ένα πυρήνα ο καθένας, οπότε έχουμε 8 επεξεργαστικές μονάδες. Όσον αφορά το χρόνο εκτέλεσης:

```
oslab118@os-node2:~/ask3$ time ./mandel 1
mandel executing

real    0m1.528s
user    0m1.490s
sys     0m0.025s
```

```
oslab118@os-node2:~/ask3$ time ./mandel 2
mandel executing

real    0m0.738s
user    0m1.077s
sys     0m0.049s
```

```
oslab118@os-node2:~/ask3$ time ./mandel 8
mandel executing

real    0m0.225s
user    0m1.136s
sys     0m0.092s
```

```
oslab118@os-node2:~/ask3$ time ./mandel 9
mandel executing

real    0m0.293s
user    0m1.476s
sys     0m0.109s
oslab118@os-node2:~/ask3$
```

```
oslab118@os-node2:~/ask3$ time ./mandel 10000
mandel executing

real    0m1.222s
user    0m1.126s
sys     0m1.490s
```

Παρατηρούμε πως η χρήση πολλαπλών threads επιταχύνει την έξοδο του προγράμματός μας. Γενικά, η επιτάχυνση αυτή λειτουργεί με φθίνον ρυθμό (για 2 threads ο χρόνος εκτέλεσης περίπου υποδιπλασιάζεται, αλλά η διαφορά 2 με 3 threads δεν είναι αντίστοιχα μεγάλη). Μάλιστα, παρατηρούμε ελαχιστοποίηση του χρόνου εκτέλεσης για 8 threads, όσα δηλαδή και οι πυρήνες της μηχανής που χρησιμοποιούμε. Για περισσότερα threads ο χρόνος αυξάνεται και για πολύ περισσότερα threads ο χρόνος φαίνεται να έχει φτάσει σε κατάσταση κορεσμού, δηλαδή είναι παρόμοιος για μεγάλο αριθμό threads, και μάλιστα έχει αυξηθεί πολύ, λόγω του κόστους δημιουργίας, συγχρονισμού και δρομολόγησης αυτού του πλήθους νημάτων. Αυτό συμβαίνει γιατί το μηχάνημα μπορεί να τρέχει ταυτόχρονα 8 threads (ένα στον κάθε πυρήνα του), οπότε για παραπάνω threads το λειτουργικό σύστημα δρομολογεί πλέον ποια threads θα

έχουν πλέον πρόσβαση στους επεξεργαστές, περιορίζοντας έτσι τα οφέλη της παράλληλης επεξεργασίας.

3. *Πόσες μεταβλητές συνθήκης χρησιμοποιήσατε στη δεύτερη εκδοχή του προγράμματος σας? Αν χρησιμοποιηθεί μια μεταβλητή πως λειτουργεί ο συγχρονισμός και ποιο πρόβλημα επίδοσης υπάρχει?*

Χρησιμοποιήσαμε μία μεταβλητή συνθήκης με το όνομα "cond". Η μεταβλητή αυτή χρησιμοποιείται για να επιτευχθεί ο συγχρονισμός μεταξύ των νημάτων ο οποίος γίνεται με τη χρήση ενός κλειδώματος (mutex) και μιας μεταβλητής συνθήκης (condition variable). Στην περίπτωση αυτή, η μεταβλητή συνθήκης χρησιμοποιείται για να ειδοποιήσει τα νήματα όταν είναι έτοιμα να εκτελέσουν τον κρίσιμο κώδικα.

Το πρόβλημα επίδοσης που υπάρχει στην παρούσα υλοποίηση είναι ότι όλα τα νήματα περιμένουν στην συνθήκη πριν εκτελέσουν τον κρίσιμο κώδικα. Αυτό έχει ως αποτέλεσμα να μην αξιοποιείται πλήρως η δυνατότητα παράλληλης εκτέλεσης. Με άλλα λόγια, όλα τα νήματα περιμένουν στην συνθήκη cond μέχρι να είναι η σειρά τους να εκτελέσουν τον κρίσιμο κώδικα, δηλαδή μέχρι η μεταβλητή line να γίνει ίση με τη μεταβλητή j. Ακόμα, σε κάθε επανάληψη, όλα τα threads ξυπνάνε ώστε όλα εκτός από ένα να περιμένουν πάλι τη συνθήκη, το οποίο δεν είναι αποδοτικό προγραμματιστικά. Σε αυτό θα βοηθούσε η χρήση πολλών conditions, καθώς κάθε φορά θα ξυπνούσε αποκλειστικά το mutex που θέλουμε.

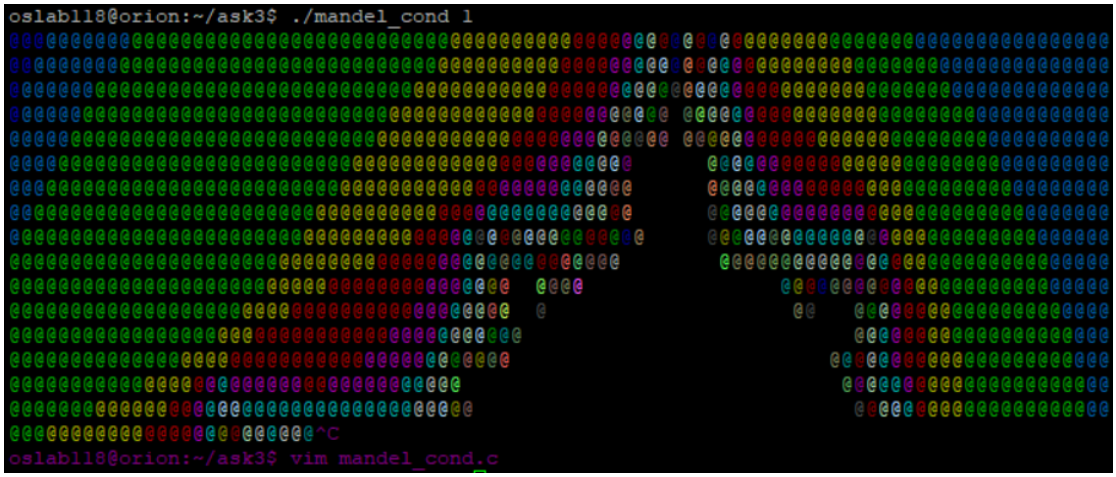
4. *Το παράλληλο πρόγραμμα που φτιάξατε, εμφανίζει επιτάχυνση; Αν όχι, γιατί; Τι πρόβλημα υπάρχει στο σχήμα συγχρονισμού που έχετε υλοποιήσει; Υπόδειξη: Πόσο μεγάλο είναι το κρίσιμο τμήμα; Χρειάζεται να περιέχει και τη φάση υπολογισμού και τη φάση εξόδου κάθε γραμμής που παράγεται;*

Το παράλληλο πρόγραμμα παρουσιάζει επιτάχυνση συγκριτικά με το σειριακό (παίρνει περίπου τον μισό χρόνο). Αυτό συμβαίνει διότι μέσω των νημάτων, το πρόγραμμα τρέχει με τέτοιο τρόπο ώστε να υπολογίζονται N (όσα νήματα έχουμε) γραμμές παράλληλα (ταυτόχρονα). Αντίθετα, το σειριακό πρόγραμμα ο υπολογισμός της γραμμής γίνεται σειριακά, δηλαδή μία τη φορά. Το κρίσιμο σημείο δεν είναι ο υπολογισμός κάθε γραμμής, ο υπολογισμός θα μπορούσε να γίνεται παράλληλα χωρίς να εμφανίσει κάποια επιπλοκή. Το κρίσιμο κομμάτι είναι η εκτύπωση κάθε στοιχείου κάθε γραμμής στη σωστή σειρά. Ωστόσο, δεν χρειάζεται να περιέχει και τις δύο φάσεις (υπολογισμού και εξόδου) καθώς αυτό εξασφαλίζει ότι τα νήματα εκτυπώνουν τα αποτελέσματα στη σωστή σειρά, ανεξάρτητα από τη σειρά με την οποία ολοκληρώνουν τον υπολογισμό τους.

5. *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;*

Αν πατηθεί το Ctrl-C, μία εντολή που διακόπτει την διαδικασία εκτέλεσης ενός προγράμματος, την ώρα που τρέχει το mandel, θα διακοπεί αφήνοντας μία ημιτελή εικόνα του mandelbrot, προφανώς, και στην γραμμή εντολής θα παραμείνει το χρώμα

με το οποίο το πρόγραμμα θα εκτύπωνε την γραμμή την οποία ήθελε να εκτυπώσει την ώρα που δόθηκε η εντολή για διακοπή, όπως φαίνεται και στην εικόνα παρακάτω. Αν το πρόγραμμα εκτελεστεί χωρίς διακοπή (δεν πατήσουμε Ctrl-C), τότε το χρώμα των γραμμάτων επανέρχεται στο default με την συνάρτηση reset_xterm_color() που καλείται πριν το main-thread κάνει return.



Επομένως, για να αποφύγουμε το πρόβλημα αυτό, επεκτείνουμε τον κώδικα μας έτσι ώστε να επαναφέρεται αυτόματα η κανονική κατάσταση της γραμμής εντολών. Χρησιμοποιούμε ένα signal handler όπως φαίνεται παρακάτω.

```
void sigint_handler(int signum){
        reset_xterm_color(1);
// Reset the terminal color to its original state
        exit(1);
} //signal handler is a function that gets called in response to the
occurrence of a specific signal



// we use a signal handler for the SIGINT signal (which is generated
when the user presses Ctrl+C)
    struct sigaction sa;
    sigset_t sigset;
    sa.sa_handler=sigint_handler;
    sa.sa_flags=SA_RESTART;// Set the flags for the signal action
(SA_RESTART enables restarting interrupted system calls)
    sigemptyset(&sigset);
    sa.sa_mask= sigset; //specifies a signal mask that indicates which
signals should be blocked (i.e., temporarily ignored) while the signal
handler is executing
    if(sigaction(SIGINT, &sa, NULL)<0) {
            perror("sigaction");
            exit(1);
        }
```

Άρα, πλέον πατώντας Ctrl + C, το τερματικό επαναφέρεται στην προηγούμενη κατάστασή του.