ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ
2Η ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ 6ΟΥ ΕΞΑΜΗΝΟΥ

ΠΑΠΑΝΔΡΙΚΟΠΟΥΛΟΣ ΓΡΗΓΟΡΗΣ ΠΑΝΑΓΙΩΤΗΣ 03121136
ΝΤΑΒΕΑΣ ΣΤΑΣΙΝΟΣ 03121076

ΑΣΚΗΣΗ 1

Αρχικά αντιγράψαμε όλα τα αρχεία από το path που βρίσκονταν στο oslab025 προκειμένου να μπορούμε να τα επεξεργαστούμε

```
oslab025@os-node1:/home/oslab/code/sync$ cp /home/oslab/code/sync/* /home/oslab/oslab025
```

Έπειτα, με την χρήση της εντολής Makefile μεταγλωττίσαμε και συνδέσαμε τα κατάλληλα αρχεία κώδικα

```
oslab025@os-node1:~$ make
gcc -Wall -O2 -pthread -c -o pthread-test.o pthread-test.c
gcc -Wall -O2 -pthread -o pthread-test pthread-test.o
gcc -Wall -O2 -pthread -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c
gcc -Wall -O2 -pthread -o simplesync-mutex simplesync-mutex.o
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c
gcc -Wall -O2 -pthread -o simplesync-atomic simplesync-atomic.o
gcc -Wall -O2 -pthread -c -o kgarten.o kgarten.c
gcc -Wall -O2 -pthread -o kgarten kgarten.o
gcc -Wall -O2 -pthread -c -o mandel-lib.o mandel-lib.c
gcc -Wall -O2 -pthread -c -o mandel.o mandel.c
gcc -Wall -O2 -pthread -o mandel mandel-lib.o mandel.o
```

Στην συνέχεια, τρέξαμε το πρόγραμμα

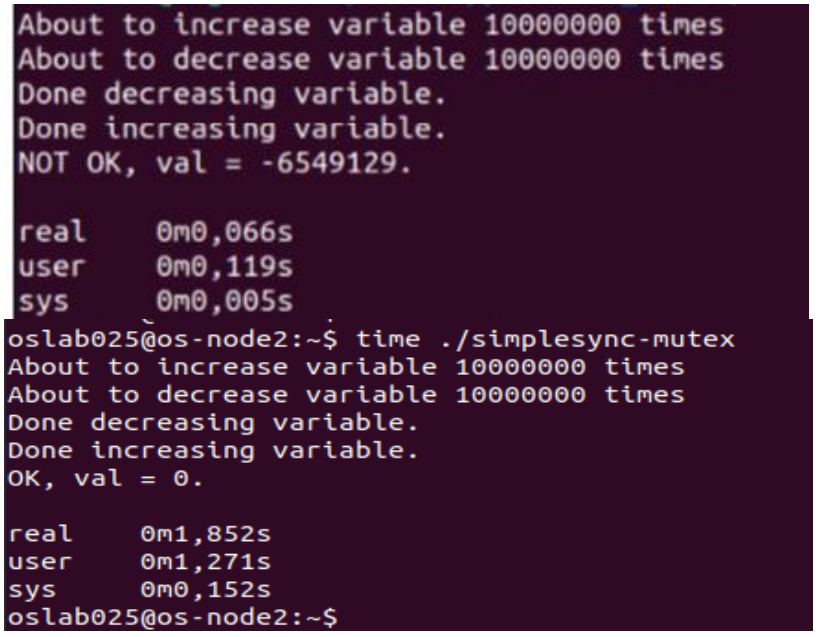
```
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 7166394.
```

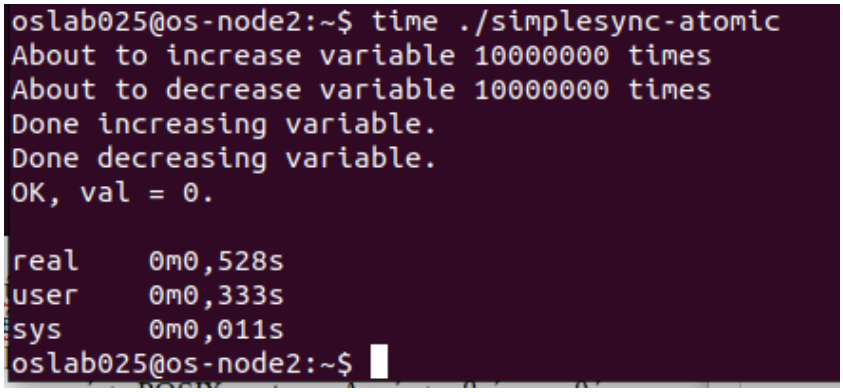
Α)Κατά την χρήση της εντολής make για την εκτέλεση του Makefile συνδέσαμε και μεταγλωττίσαμε τα αρχεία κώδικα που περιελάβανε. Έπειτα εκτελέσαμε το executable simplesync-mutex.Αυτό το οποίο περιμέναμε ήταν το val να ισούται με μηδέν όμως είναι εντελώς διαφορετικό. Αυτό συμβαίνει καθώς γίνεται παράλληλη εκτέλεση των δύο νημάτων που εκτελούν τις N αφαιρέσεις του 1 και N προσθέσεις του 1 αντίστοιχα με αποτέλεσμα να δημιουργείται κατάσταση συναγωνισμού (race condition): κατάσταση κατά την οποία το αποτέλεσμα ενός υπολογισμού εξαρτάται από την σειρά που πραγματοποιούνται οι προσπελάσεις .Ειδικότερα αυτό οφείλεται στο γεγονός ότι η εντολή της πρόσθεσης και της αφαίρεσης στην assembly αποτελείται από περισσότερες από μία εντολές .Συνεπώς, κατά την αλλαγή νήματος υπάρχει περίπτωση το κρίσιμο

τμήμα να μην έχει ολοκληρωθεί σωστά και να έχουμε τα αντίθετα από τα επιδιωκόμενα αποτελέσματα. Για αυτό θα πρέπει να κλειδώσουμε αυτά τα τμήματα προκειμένου να εκτελούνται με σειριακό τρόπο.

Β)Κατά την εκτέλεση του Makefile παρατηρούμε ότι παράγονται δύο διαφορετικά εκτελέσιμα simplesync-atomic, simplesync-mutex, από το ίδιο αρχείο πηγαίου κώδικα simplesync.c. Αυτό συμβαίνει χάρη στις εντολές -D SYNC ATOMIC και -D SYNC MUTEX όπου ορίζονται κατά τη μεταγλώττιση και καθορίζουν ποιος μηχανισμός θα χρησιμοποιηθεί για τη διαχείριση των κρίσιμων τμημάτων κώδικα. Στον κώδικα του προγράμματος simplesync ανάλογα με την τιμή που έχει η macro USE_ATOMIC_OPS ακολουθούμε μια διαφορετική προσέγγιση για την αντιμετώπιση του προβλήματος παράλληλης εκτέλεσης είτε με mutexes είτε μέσω atomic operations.

ΕΡΩΤΗΣΕΙΣ 1.1/1.2



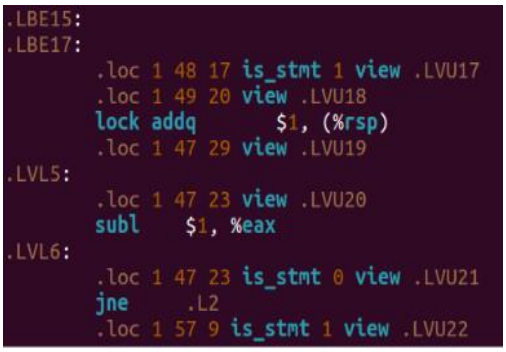


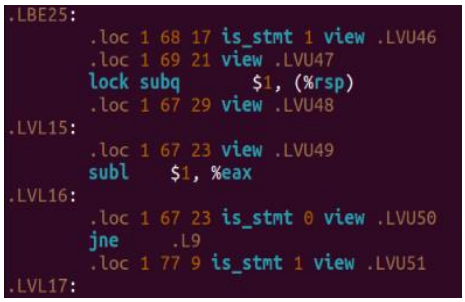
Από την χρήση της εντολής time παίρνουμε τις ακόλουθες πληροφορίες:
real: Ο συνολικός πραγματικός χρόνος που πέρασε από την έναρξη μέχρι την ολοκλήρωση της εκτέλεσης του προγράμματος. Περιλαμβάνει όλο τον χρόνο, συμπεριλαμβανομένου του χρόνου που πέρασε περιμένοντας άλλες διεργασίες του συστήματος, εισόδους/εξόδους (I/O), και οτιδήποτε άλλο μπορεί να συμβεί κατά τη διάρκεια της εκτέλεσης του προγράμματος.

User CPU: Ο χρόνος CPU που χρησιμοποιήθηκε από την διεργασία για την εκτέλεση του χρήστη (user space). Ο χρόνος που ο επεξεργαστής ξόδεψε εκτελώντας τις εντολές του προγράμματος. Αυτός ο χρόνος δεν περιλαμβάνει το χρόνο που το σύστημα ξόδεψε σε λειτουργίες πυρήνα (kernel space).
Sys CPU: Ο χρόνος CPU που χρησιμοποιήθηκε από το λειτουργικό σύστημα (kernel space) για λογαριασμό της διεργασίας σας. Ο χρόνος που ο επεξεργαστής ξόδεψε εκτελώντας εντολές συστήματος για το πρόγραμμά . Περιλαμβάνει λειτουργίες όπως η διαχείριση αρχείων, η διαχείριση μνήμης και άλλες λειτουργίες πυρήνα.
Χρησιμοποιώντας την εντολή time(1) συμπεραίνουμε ότι οι χρόνοι εκτέλεσεις των executable simplesync-atomic και simplesync-mutex είναι σημαντικά πιο αργοί από τον αντίστοιχο χρόνο εκτέλεσης του αρχικού κώδικα. Η διαφορά στον χρόνο εκτέλεσης εντοπίζεται κυρίως στην CPU. Οι ατομικές εντολές και τα mutexes είναι πιο αργές καθώς τα νήματα εκτελούν busy waiting. Ακόμη , το executable που κάνει χρήση ατομικών λειτουργιών του GCC είναι γρηγορότερο από αυτό που συγχρονίζεται με χρήση POSIX mutexes.Αυτό συμβαίνει καθώς τα atomic operations αποτελούν απλές ενέργειες που εκτελούνται κατευθείαν στο επίπεδο αρχιτεκτονικής του επεξεργαστή. Το κλείδωμα του κρίσιμου τμήματος γίνεται από τον μεταγλωττιστή, χωρίς την ανάγκη να γίνει ανταλλαγή μηνυμάτων με τον πυρήνα του λειτουργικού συστήματος με αποτέλεσμα την μείωση του χρόνου κλειδώματος. Στον αντίποδα, ο αμοιβαίος αποκλεισμός απαιτεί περισσότερο χρόνο διότι όταν το νήμα θέλει να αποκτήσει πρόσβαση σε ένα κρίσιμο σημείο πρέπει να γίνει κλήση του συστήματος για να αποκτήσει πρόσβαση στο κλείδωμα και αυτό μπορεί να προκαλέσει σαν αποτέλεσμα είτε spinlock αν αναμένουν πολλά νήματα είτε ακόμη μπλοκάρισμα του νήματος αν στο κρίσιμο σημείο βρίσκεται άλλο νήμα.

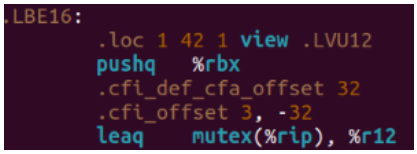
Ερώτηση 1.3/1.4)

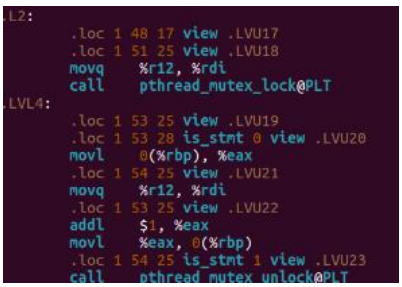
Τώρα θα εντοπίσουμε εντός των αρχείων assembly που δημιουργήσαμε βρίσκουμε τις εντολές που αντιστοιχούν στα atomic operations και στα mutexes και τις παραθέτουμε παρακάτω:

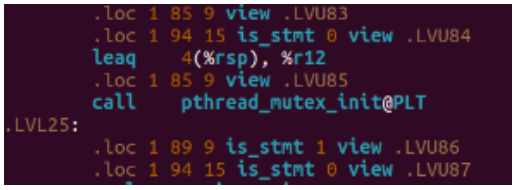


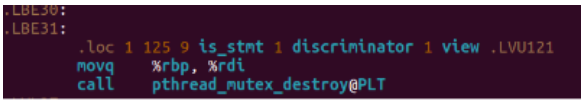


Τα παραπάνω scpeenshots αφορούν τις εντολές για τα atomic operations









Και αντιστοίχα τα παραπάνω screenshots αφορούν τις εντολές για τα mutexes

ΠΑΡΑΚΑΤΩ ΠΑΡΑΘΕΤΟΥΜΕ ΤΟΝ ΚΩΔΙΚΑ ΤΗΣ ΑΣΚΗΣΗΣ 1
ΚΑΙ ΣΧΟΛΙΑΖΟΥΜΕ ΤΗΝ ΛΥΣΗ ΜΑΣ

```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 <stdatomic.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; // Initialize the mutex

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 */
                        // Αύξηση της count ατομικά κατά 1
                        __sync_fetch_and_add(&ip, 1);
```

```c
        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 */
                        // Αύξηση της count ατομικά κατά 1
                        __sync_fetch_and_add(&ip, 1);

                        /* ... */
                } else {
                        /* ... */
                        /* You cannot modify the following line */
                        pthread_mutex_lock(&mutex); // Lock the mutex before accessing the shared variable
                         ++(*ip);
                        pthread_mutex_unlock(&mutex); // Unlock the mutex after modifying the shared

                        /* ... */
                }
        }
        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 {
                        /* ... */
                        /* You cannot modify the following line */
                        pthread_mutex_lock(&mutex); // Lock the mutex before accessing the shared variable
                         --(*ip);
                        pthread_mutex_unlock(&mutex); // Unlock the mutex after modifying the shared

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

        return NULL;
}


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

```c
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;
}
```

Για την υλοποίηση της άσκησης αρχικά συμπεριλάβαμε τις κατάλληλες βιβλιοθήκες που απαιτούνται. Εισαγάγαμε την βιβλιοθήκη #include <pthread.h> καθώς αυτή η βιβλιοθήκη περιέχει τις δηλώσεις για τη βιβλιοθήκη POSIX thread (pthreads). Είναι απαραίτητη για τη δημιουργία και τον χειρισμό νήματος (thread), καθώς και για τη χρήση mutex και άλλων συγχρονιστικών μηχανισμών. Επιπλέον χρησιμοποιήσαμε την βιβλιοθήκη #include <stdatomic.h> για τις ατομικές λειτουργίες που θα χρησιμοποιήσουμε για την υλοποίηση της άσκησης. Στον δοθέν πρόγραμμα, προσθέσαμε την εντολή pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER προκειμένου να αρχικοποιήσουμε το mutex που θα χρησιμοποιήσουμε για να ορίσουμε τις κρίσιμες περιοχές του κώδικά μας στους οποίους θα αποτρέπεται η ταυτόχρονη πρόσβαση από πολλαπλά νήματα. Ακόμη αξίζει να σημειωθεί ότι το mutex αρχικοποιείται με την παραπάνω μακροεντολή προκειμένου να διασφαλιστεί ότι το mutex είναι έτοιμο να εκτελεστεί. Οι συναρτήσεις increase() και decrease() αποτελούν τις συναρτήσεις που εκτελεί κάθε νήμα (thread) .Παίρνουν ως όρισμα το arg που κατόπιν ανάθεσης μέσα στην συνάρτηση αποτελεί ένα δείκτη σε μία μεταβλητή τύπου int (int ***ip).** Οι συναρτήσεις αυτές χρησιμοποιούνται προκειμένου να ελαττώσουν ή αυξήσουν την μεταβλητή val που ανατίθεται στην main . Προκειμένου να προστατεύσουμε την shared variable ip όταν αυξάνουμε ή μειώνουμε την τιμή της κλειδώνουμε τις εντολές ++(*ip) και --(*ip) μέσω των εντολών pthread_mutex_lock(&mutex) (εντολή που παίρνει ώς όρισμα την διεύθυνση του mutex και το κλειδώνει αποτρέποντας έτσι την ταυτόχρονη πρόσβαση των κλειδωμάτων) και της εντολής pthread_mutex_unlock(&mutex)(εντολή που παίρνει ως όρισμα την διεύθυνση του mutex και το ξεκλειδώνει , επιτρέποντας έτσι σε άλλα νήματα να αποκτήσουν πρόσβαση.) .Η δεύτερη υλοποίηση των συναρτήσεων είναι μέσω χρήσης ατομικών πράξεων. Συγκεκριμένα καλούμε τις συναρτήσεις __sync_fetch_and_add(&ip,1) και __sync_fetch_and_sub(&ip,1) στην increase και decrease αντίστοιχα προκειμένου να εκτελεστούν ατομικά οι αυξήσεις και οι μειώσεις του ip κατά 1 χωρίς να υπάρξει πρόβλημα συγχρονισμού. Παίρνουν ως ορίσματα την διεύθυνση της μεταβλητής που θα προσθέσουν ή αφαιρέσουν αντίστοιχα με την τιμή του δευτέρου ορίσματος τους.
Στην main δημιουργούμε δύο μεταβλητές t1,t2 τύπου pthread_t καθώς σε αυτές τις μεταβλητές θα αποθηκευτεί ο αναγνωριστικός αριθμός των νημάτων που θα δημιουργήσουμε. Αμέσως μετά δημιουργούμε τα νήματα μέσω της συνάρτησης pthread_create που παίρνει ως όρισμα αρχικά την μεταβλητή τύπου pthread_t που είναι οι μεταβλητές που θα περιέχουν τα id των threads, ως δεύτερο όρισμα βάζουμε την τιμή NULL (καθώς παίρνει by default τιμές για το συγκεκριμένο όρισμα) ,ως τρίτο την συνάρτηση που θέλουμε να εκτελεί το νέο νήμα increase και decrease αντίστοιχα και ως τέταρτο όρισμα όπως και στο δεύτερο την τιμή NULL.Τέλος καλούμε την pthread_join μία φορά για το κάθε νήμα (παίρνει ως ορίσματα την διεύθυνση αναφοράς του νήματος και την τιμή NULL καθώς δεν ενδιαφερόμαστε για τις τιμές επιστροφής των νημάτων) καθώς θέλουμε να εξασφαλιστεί ότι το κύριο νήμα δεν θα τερματιστεί πριν ολοκληρωθούν τα νήματα που δημιουργήθηκαν, κάτι που θα μπορούσε να προκαλέσει απώλεια δεδομένων ή άλλες ασυνέπειες στην εκτέλεση του προγράμματος. Ακόμη αξίζει να σημειωθεί πως κάνουμε τους απαραίτητους ελέγχους σε περίπτωση που προκύψει σφάλμα κατά την εκτέλεση των pthread_create και των pthread_join .Κατά την εκτέλεση του προγράμματος το val θα εξακολουθήσει να έχει την τιμή 0.

(το αρχείο που εκτελεί το πρόγραμμα είναι το Makefile1 μέσω της εντολής make -f Makefile1)

2Η ΑΣΚΗΣΗ ΠΑΡΑΛΛΗΛΟΣ ΥΠΟΛΟΓΙΣΜΟΣ ΤΟΥ ΣΥΝΟΛΟΥ MANDELBROT

Παρακάτω, παραθέτουμε την υλοποίηση της άσκησης με χρήση συμφόρων

```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 <bits/pthreadtypes.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;

sem_t *semaphore; // Πίνακας σημαφόρων
int NTHREADS; // Αριθμός νημάτων

//ΣΕ ΠΕΡΙΠΤΩΣΗ ΠΟΥ ΠΑΤΗΘΕΙ CTRL^C ΠΑΡΑΚΑΤΩ ΕΙΝΑΙ Ο
//Ο HANDLER
void sigint_handler(int signum) {
    reset_xterm_color(1);
    exit(1);
}
```

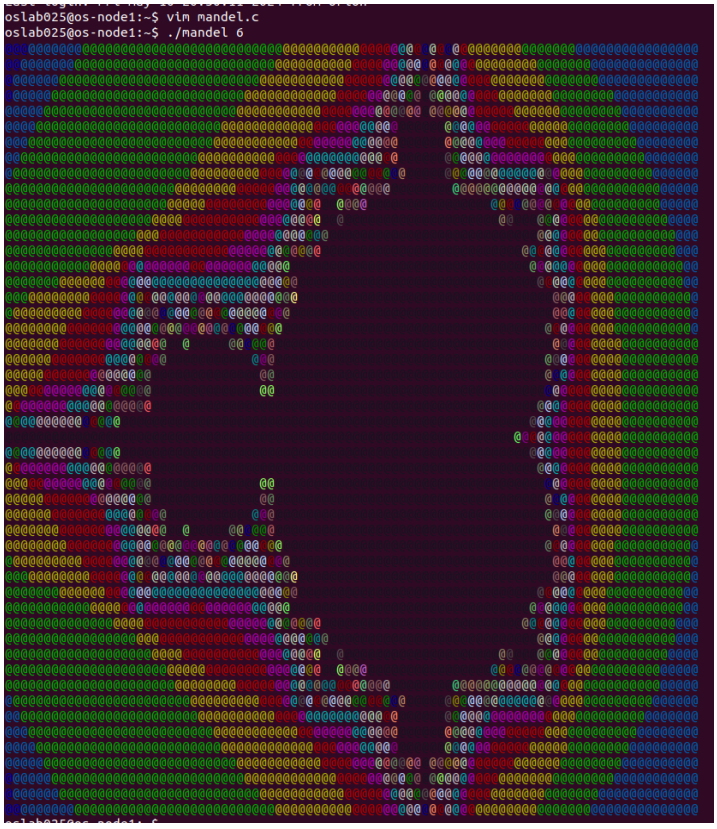
```c
void usage(char *argv0)
{
        fprintf(stderr, "Usage: %s thread_count array_size\n\n"
                "Exactly one argument required:\n"
                "    thread_count: The number of threads to create.\n",
                argv0);
        exit(1);
}

/*
 * 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]);
```

```
      */

    int i;
    if (argc != 2)
                usage(argv[0]);

    NTHREADS = atoi(argv[1]); // Πάρε τον αριθμό των νημάτων από το command line

   //ΣΕ ΠΕΡΙΠΤΩΣΗ ΠΟΥ ΚΛΗΘΕΙ ΔΙΑΚΟΠΗ ΤΟΥ ΠΡΟΓΡΑΜΜΑΤΟΣ ΑΠΟ ΤΟ ΠΛΗΚΤΡΟΛΟΓΙΟ ΜΕΣΩ ΤΗΣ ΕΝΤΟΛΗ CTRL^C
   //ΜΕΣΩ ΤΗΣ SIGNAL ΣΤΕΛΝΟΥΜΕ ΤΟ PC ΣΤΟΝ HANDLER
    if (signal(SIGINT, sigint_handler) == SIG_ERR) {
        perror("signal");
        exit(1);
    }



    // Δέσμευση μνήμης για τον πίνακα των σημαφόρων
    semaphore = (sem_t *)malloc(NTHREADS * sizeof(sem_t));
    if (sem_init(&semaphore[0], 0, 1) < 0) { // Αρχικοποίηση του πρώτου σημαφόρου
        perror("Error in initializing first semaphore");
       return 1;
    }

    // Αρχικοποίηση των υπολοίπων σημαφόρων
    for (int i = 1; i < NTHREADS; i++) {
        if (sem_init(&semaphore[i], 0, 0) < 0) {
            perror("Error in initialization of semaphores");
            return 1;
        }
    }

    pthread_t array_threads[NTHREADS];
    for (i = 0; i < NTHREADS; i++) {
        int *a = malloc(sizeof(int));
        *a = i;
        if (pthread_create(&array_threads[i], NULL, &thread_function, a) != 0) { // Δημιουργία νέου νήματος
            perror("Failed thread creation");
        }
    }

    for (i = 0; i < NTHREADS; i++) {
        if (pthread_join(array_threads[i], NULL) != 0) { // Αναμονή τερματισμού των νημάτων το πρώτο όρισμα παίρνει τι id του νήματος και το δέυτερο την τιμή NULL
            perror("Failed thread termination");
        }
    }

    for (int i = 0; i < NTHREADS; i++) {
        if (sem_destroy(&semaphore[i]) < 0) { // Καταστροφή των σημαφόρων
            perror("Semaphores Failed to be destroyed");//αν η sem_destroy επιστρέψει αρνητική τιμή
            return 1;
        }
    }
```

Σχολιασμός του παραπάνω κώδικα.
Αρχικά, κάνουμε include την κατάλληλη βιβλιοθήκη για την χρήση συμφόρων .Έπειτα δηλώνουμε ένα δείκτη σε συμφόρους sem *semaphore , καθώς στην συνέχεια θα δεσμεύσουμε δυναμικά μνήμη για να δημιουργήσουμε όσους συμφόρους χρειαστούμε (πρόκειται για ένα δείκτη σε πίνακα συμφόρων) .Έπειτα δηλώνουμε την μεταβλητή NTHREADS η οποία εκφράζει πόσα νήματα θα τρέξουν στο πρόγραμμα. Στην συνέχεια δημιουργήσαμε την συνάρτηση thread_function που αποτελεί την συνάρτηση που εκτελείται από κάθε νήμα . Ειδικότερα, η thread_function παίρνει σαν όρισμα ένα δείκτη σε ακέραιο και τον αποθηκεύει στην μεταβλητή start_line. Δημιουργεί ένα πίνακα color_val με μέγεθος x_chars όπου x_chars είναι το πλήθος των χαρακτήρων κάθε γραμμής. Έπειτα εκτελεί ένα for loop το οποίο το ξεκινάμε όπως υποδεικνύει και η άσκηση από τη γραμμή start_line και αυξάνουμε τον δείκτη i κατά NTHREADS προκειμένου η διεργασία i (με i = 0, 1, 2, . . . , n − 1) να αναλαμβάνει τις σειρέςi, i + n, i + 2 × n, i + 3 × n, . . όπου n = NTHREADS. Στην συνέχεια καλούμε τη συνάρτηση compute_mandel_line(i, color_val) για να υπολογίσουμε τις τιμές των χρωμάτων για τη γραμμή i και να τις αποθηκεύσουμε στον πίνακα color_val.Έπειτα, καλούμε τη συνάρτηση sem_wait για να περιμένει να λάβει ένα σήμα από το προηγούμενο νήμα. Εμφανίζουμε τη γραμμή με τη χρήση της συνάρτησης output_mandel_line(1, color_val).Και τέλος, καλούμε τη συνάρτηση sem_post για να στείλει ένα σήμα στο επόμενο νήμα, προκειμένου να συνεχίσει την εκτέλεσή του.
Τέλος ,μέσω της εντολής free αποδεσμεύουμε την μνήμη που καταλαμβάνεται από το όρισμα arg .Στην main προσθέσαμε τα παρακάτω. Αρχικά μέσω της
Της διαδικασίας usage σε περίπτωση που ο χρήστης δεν δώσει όρισμα εκτυπώνει κατάλληλο μήνυμα .Έπειτα, μέσω της συνάρτησης atoi η οποία διαβάζει το όρισμα το

οποίο θέσαμε κατά την εκτέλεση του προγράμματος μετατρέπει τους χαρακτήρες από αλφαριθμητική ακολουθία str και την μετατρέπει σε ακέραιο αριθμό που αποθηκεύεται στην global μεταβλητή NTHREADS (προϋπόθεση να είναι ψηφία σε άλλη περίπτωση επιστρέφει 0).Έπειτα μέσω της malloc δεσμεύουμε δυναμικά μνήμη για τον πίνακα συμφόρων θέσεων NTHREADS.Έπειτα, αρχικοποιούμε τον πρώτο σημαφόρο μέσω της sem_init βάζοντας της ως όρισμα αρχικά την αναφορά του πρώτου σημαφόρου (by reference) μετά αρχικοποιούμε το pshared στο 0(αν είναι μηδέν πρόκειται για σημαφόρους που χειρίζονται threads αν είναι 1 πρόκειται για σημαφόρους που διαχειρίζονται processes) και ως τελευταίο όρισμα παίρνει για τον αρχικό σημαφόρο την τιμή του σημαφόρου αρχικοποιημένη στο 1 (ο πρώτος σημαφόρος είναι ξεκλείδωτος).Έπειτα , αρχικοποιεί όλους τους υπόλοιπους σημαφόρους μέσω ενός for loop με μόνη διαφορά πως όλοι είναι αρχικοποιημένοι στο 0 στο τελευταίο τους όρισμα δηλαδή κλειδωμένοι. Έπειτα μέσω της pthread_create δημιουργούμε NTHREADS νήματα. Η οποία παίρνει ως ορίσματα την αναφορά του thread array δηλαδή την διεύθυνσή του, μετά την τιμή NULL μετά την thread_function δηλαδή την συνάρτηση που θα κληθεί κατά την κλήση του συστήματος και τέλος το a που είναι ο δείκτης που πρόκειται να περαστεί ως όρισμα στην thread_function και έχουμε δεσμεύσει δυναμικά μνήμη μέσω της malloc για κάθε a και του αναθέτουμε την τιμή I δηλαδή το start_line κάθε thread.Ακόμη για κάθε σημαφόρο καλούμε την thread_join όπου αυτό που κάνει είναι να σταματά την εκτέλεση του κύριου νήματος όσο εκτελείται το thread που έχει ως όρισμα. Τέλος για κάθε σημαφόρο όταν τελειώσει την χρήση του αποδεσμεύουμε τον χώρο που έχει δεσμεύσει μέσω της sem_destroy.

Το output, ενδεικτικά για 6 threads είναι:



(το αρχείο που εκτελεί το πρόγραμμα είναι το Makefile1 μέσω της εντολής make -f Makefile3)

2Η ΑΣΚΗΣΗ ΠΑΡΑΛΛΗΛΟΣ ΥΠΟΛΟΓΙΣΜΟΣ ΤΟΥ ΣΥΝΟΛΟΥ MANDELBROT

Παρακάτω, παραθέτω την υλοποίηση της άσκησης με χρήση conditional variables

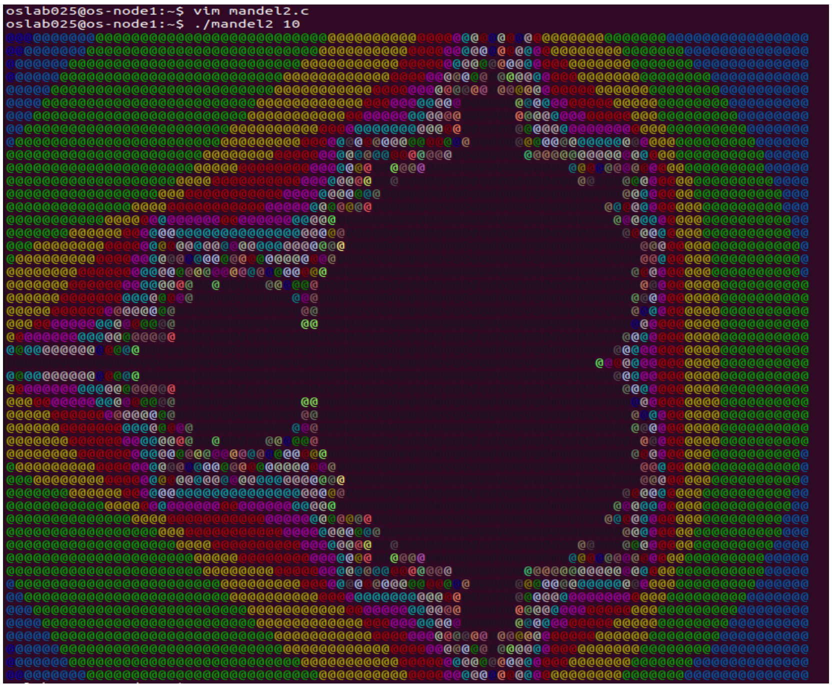
```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>
#include <bits/pthreadtypes.h>
#include <semaphore.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

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

int y_chars = 50;
int x_chars = 90;

double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

double xstep;
double ystep;

sem_t *semaphore; // Πίνακας σημαφόρων
int NTHREADS; // Αριθμός νημάτων

pthread_mutex_t mutex1,mutex2;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;//δημιουργούμε την condition variable που θα χρησιμοποιήσουμε

int current_thread = 0;
/*
 * Η συνάρτηση αυτή υπολογίζει μια γραμμή της εικόνας Mandelbrot ως έναν πίνακα
 *     χρωμάτων που αντιστοιχεί σε κάθε σημείο της γραμμής.
 */
 void compute_mandel_line(int line, int color_val[])
{
    double x, y;
    int n;
    int val;

    y = ymax - ystep * line;

    for (x = xmin, n = 0; n < x_chars; x += xstep, n++) {
        val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
        if (val > 255)
            val = 255;
        val = xterm_color(val);
        color_val[n] = val;
    }
}
```

```
        color_val[n] = val;
    }
}
/*
 * Η συνάρτηση αυτή εκτυπώνει μια γραμμή χρωμάτων στο τερματικό.
 */
void output_mandel_line(int fd, int color_val[])
{
    int i;

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

    for (i = 0; i < x_chars; i++) {
        set_xterm_color(fd, color_val[i]);
        if (write(fd, &point, 1) != 1) {
            perror("compute_and_output_mandel_line: write point");
            exit(1);
        }
    }

    if (write(fd, &newline, 1) != 1) {
        perror("compute_and_output_mandel_line: write newline");
        exit(1);
    }
}

// Κλείδωμα νημάτων
pthread_mutex_t pth;

// Η συνάρτηση που εκτελείται από κάθε νήμα
void *thread_function(void *arg) {
    int start_line = *(int *)arg; // Πάρε τον δείκτη από το όρισμα
    int color_val[x_chars];
    for (int i = start_line; i < y_chars; i += NTHREADS) {
        compute_mandel_line(i, color_val);
        pthread_mutex_lock(&mutex1);
        while (current_thread != start_line%NTHREADS){
            pthread_cond_wait(&cond, &mutex1);
        }
        pthread_mutex_lock(&mutex2);
        output_mandel_line(1, color_val);
        current_thread = (current_thread + 1) % NTHREADS;
        for(int i = 0 ; i<NTHREADS-1;i++){
        pthread_cond_signal(&cond);
        }
        pthread_mutex_unlock(&mutex1);
        pthread_mutex_unlock(&mutex2);

    }
    free(arg);
    return NULL;


}
int main(int argc, char *argv[]) {
    xstep = (xmax - xmin) / x_chars;
    ystep = (ymax - ymin) / y_chars;
```

```c
}
int main(int argc, char *argv[]) {
    xstep = (xmax - xmin) / x_chars;
    ystep = (ymax - ymin) / y_chars;

    int i;
    NTHREADS = atoi(argv[1]); // Πάρε τον αριθμό των νημάτων από το command line



    pthread_t array_threads[NTHREADS];
    for (i = 0; i < NTHREADS; i++) {
        int *a = malloc(sizeof(int));
        *a = i;
        if (pthread_create(&array_threads[i], NULL, &thread_function, a) != 0) { // Δημιουργία νέου νήματος
            perror("Failed thread creation");
        }
    }

    for (i = 0; i < NTHREADS; i++) {
        if (pthread_join(array_threads[i], NULL) != 0) { // Αναμονή τερματισμού των νημάτων το πρώτο όρισμα παίρνει τι id του νήματος και το δέυτερο την τιμή NULL
            perror("Failed thread termination");
        }
    }
    reset_xterm_color(1);
    return 0;
}
```

Τα conditional variables χρησιμοποιούνται κυρίως κατά τον συγχρονισμό νημάτων σε πολυνηματικά προγράμματα. Στο παραπάνω πρόγραμμα τα χρησιμοποιούμε μέσω της thread_function.Αρχικά έχουμε ορίσει ως global μεταβλητές ένα conditional variable με την ονομασία cond και έχουμε ορίσει και δύο mutex με τις μεταβλητές mutex1 και mutex2 καθώς και το current_thread αρχικοποιημένο στην τιμή 0. Η μεταβλητή current_thread χρησιμοποιείται για να ελέγξει ποιο νήμα πρέπει να εκτελείται επόμενο και αρχικοποιείται στο 0 για να εκτελεστεί πρώτα το πρώτο νήμα. Η συνάρτηση thread_function έχει ίδια λειτουργία με αυτή της προηγούμενης άσκησης διαφορετική όμως υλοποίηση. Αρχικά κλειδώνει το πρώτο mutex (mutex1) για να περιορίσει την πρόσβαση στις κοινόχρηστες μεταβλητές. Έπειτα, περιμένει όσο η μεταβλητή current_thread δεν είναι ίση με το υπόλοιπο της διαίρεσης της start_line με τον αριθμό των νημάτων (NTHREADS). Αυτό είναι ώστε το νήμα να περιμένει τη σειρά του να εκτελεστεί. Έπειτα, κλειδώνει το δεύτερο mutex (mutex2) για την πρόσβαση στην συνάρτηση output_mandel_line.Στην συνέχεια καλεί τη συνάρτηση output_mandel_line για να εκτυπώσει τη γραμμή που υπολογίστηκε.
Αυξάνει τη μεταβλητή current_thread κατά έναν και την κάνει κυκλική σε περίπτωση που ξεπεράσει τον αριθμό των νημάτων (NTHREADS).Τέλος εκπέμπει σήμα μέσω της συνάρτησης (pthread_cond_signal) σε όλα τα υπόλοιπα νήματα.
Ξεκλειδώνει τα mutexes

Ενδεικτικό output για 10 threads:



(το Makefile5 χρησιμοποιείται για το execution του παραπάνω προγράμματος)

ΕΡΩΤΗΣΕΙΣ

1)Χρειάζονται τόσοι σημαφόροι όσο και το πλήθος των νημάτων προκειμένου να δημιουργούμε ένα κλείδωμα για κάθε thread που τυπώνει τις γραμμές του.
ΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠΠ
2)Χρησιμοποιήσαμε την εντολή cat /proc/cpuinfo για να δούμε πόσους υπολογιστικούς πυρήνες διαθέτει ο υπολογιστής μας

δ

```
patrinos@patrinos-VivoBook-ASUSLaptop-X513UA-M513UA:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 104
model name      : AMD Ryzen 7 5700U with Radeon Graphics
stepping        : 1
microcode       : 0x8608103
cpu MHz         : 400.000
cache size      : 512 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 16
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
```

και διαπιστώσαμε ότι έχει 8
Έπειτα εκτελέσαμε για ένα νήμα (σειριακός υπολογισμός)

```
real    0m1,718s
user    0m1,268s
sys     0m0,025s
```

Και για δύο νήματα(παράλληλος υπολογισμός)

```
real    0m1,254s
user    0m0,970s
sys     0m0,020s
```

Παρατηρούμε ότι ο χρόνος εκτέλεσης για 2 νήματα είναι σημαντικά μικρότερος από τον σειριακό

3)Στο παραπάνω πρόγραμμα χρησιμοποιήθηκε μία μόνο μεταβλητή συνθήκης (cond) για τον συγχρονισμό των νημάτων. Το πρόβλημα επίδοσης που μπορεί να προκύψει είναι η υπερβολική χρήση της μεταβλητής συνθήκης. Κάθε φορά που ένα νήμα ολοκληρώνει την επεξεργασία μιας γραμμής, ειδοποιεί όλα τα υπόλοιπα νήματα, ανεξαρτήτως εάν πραγματικά χρειάζεται να τρέξουν. Αυτό μπορεί να οδηγήσει σε πολλαπλές εκκλήσεις pthread_cond_wait, οι οποίες μπορεί να είναι περιττές και να προκαλέσουν ανεπιθύμητη καθυστέρηση και χρονοβόρα εκτέλεση. Αυτό το πρόβλημα ονομάζεται "προβλήματα εκταμίευσης" ή "θραυστό σφάλμα" (thundering herd problem). Μια λύση για αυτό το πρόβλημα είναι η χρήση μιας ή περισσότερων κλειδωμένων μεταβλητών, η οποία θα επιτρέψει μόνο στο ένα νήμα να ειδοποιήσει κάθε φορά το επόμενο
4)

Το παράλληλο πρόγραμμα που φτιάξαμε φαίνεται να είναι εμφανώς ταχύτερο από το σειριακό.Το κρίσιμο τμήμα δεν είναι μεγάλο καθώς το ouput γίνεται σε διαφορετικό τμήμα κώδικα από ότι το κρίσιμο τμήμα. Στο κρίσιμο τμήμα γίνεται μόνο ο υπολογισμός των σημείων των γραμμών. Όταν τελείωσει ο υπολογισμός και τα νήματα σταματήσσουν να περιμένου έπειτα τυπώνουμε τις γραμμές

5)Αν πατήσουμε κατά την εκτέλεση του προγράμματος την ctrl ^ c αυτό το οποίο συμβαίνει είναι να αλλάζει το φόντο του cmd .Για να το επαναφέρουμε στην αρχική κατάσταση το επιτυγχάνουμε μέσω των signals. Μέσω του handler περιέχει την εντολή reset_xterm_color επαναφέρουμε το χρώμα του φόντου στην αρχική του κατάσταση

```
//ΣΕ ΠΕΡΙΠΤΩΣΗ ΠΟΥ ΠΑΤΗΘΕΙ CTRL^C ΠΑΡΑΚΑΤΩ ΕΙΝΑΙ Ο
//Ο HANDLER ΠΟΥ ΑΝΤΙΚΑΘΙΣΤΑ ΤΟ ΣΗΜΑ ΚΑΙ ΕΠΑΝΑΦΕΡΕΙ ΤΟ ΚΩΔΙΚΑ ΣΤΗΝ
//ΠΟΥ ΒΡΙΣΚΟΤΑΝ ΠΡΙΝ ΤΗΝ ΔΙΑΚΟΠΗ
void sigint_handler(int signum) {
    reset_xterm_color(1);
    exit(1);
}
```

και στην main καλούμε την signal

```
//ΣΕ ΠΕΡΙΠΤΩΣΗ ΠΟΥ ΚΛΗΘΕΙ ΔΙΑΚΟΠΗ ΤΟΥ ΠΡΟΓΡΑΜΜΑΤΟΣ ΑΠΟ ΤΟ ΠΛΗΚΤΡΟΛΟΓΙΟ ΜΕΣΩ ΤΗΣ ΕΝΤΟΛΗ CTRL^C
//ΜΕΣΩ ΤΗΣ SIGNAL ΣΤΕΛΝΟΥΜΕ ΤΟ PC ΣΤΟΝ HANDLER
 if (signal(SIGINT, sigint_handler) == SIG_ERR) {
     perror("signal");
     exit(1);
 }
```

MAKEFILE1 SIMPLESYNC.C
MAKEFILE3 MANDEL.C
MAKEFILE5 MANDEL2.C