Λειτουργικά Συστήματα Υπολογιστών

Αναφορά στην 3η Εργαστηριακή Άσκηση

Αλέξανδρος Σκούρας, 03120105

Ιωάννης Τσαντήλας, 03120883

Εξάμηνο: Εαρινό 2022-23

3.1 Συγχρονισμός σε υπάρχοντα κώδικα

- *Χρησιμοποιήστε το παρεχόμενο Makefile για να μεταγλωττίσετε και να τρέξετε το πρόγραμμα. Τι παρατηρείτε; Γιατί;*

Παρατηρούμε πως παράγονται δύο εκτελέσιμα αρχεία, με τα αντίστοιχα object files τους, αφού στο Makefile έχουμε:

```
## Simple sync (two versions)
simplesync-mutex: simplesync-mutex.o
	$(CC) $(CFLAGS) -o simplesync-mutex simplesync-mutex.o $(LIBS)

simplesync-atomic: simplesync-atomic.o
	$(CC) $(CFLAGS) -o simplesync-atomic simplesync-atomic.o $(LIBS)

simplesync-mutex.o: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c

simplesync-atomic.o: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c
```

Ενώ στον κώδικα έχουμε:

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

Επομένως, στο εκτελέσιμο **simplesync-atomic**, τα if-statements των συναρτήσεων increase/decrease θα έχουν τιμή **1** (και άρα θα εκτελεστεί το περιεχόμενο τους, δηλαδή υλοποίηση με atomic operations) ενώ στο εκτελέσιμο **simplesync-mutex**, τα if-statements θα έχουν τιμή **0**, και άρα θα εκτελεστούν τα περιεχόμενα των else (δηλαδή εκτέλεση με POSIX mutexes).

Περνώντας στο τρέξιμο των δύο εκτελέσιμων, και τα δύο επιστρέφουν μία φαινομενικά τυχαία τιμή στο value. Αυτό συμβαίνει γιατί και στις δύο περιπτώσεις δεν υπάρχει συγχρονισμός, καθώς η αύξηση και η μείωση της τιμής του value δεν γίνεται με την σειρά που επιθυμούμε, παρά είναι αποτέλεσμα race condition, μεταξύ των εντολών

++(*ip) και --(*ip). Όταν τα αντίστοιχα threads φτάνουν στις εντολές, μπορεί ένα να δεσμεύσει την CPU πολλαπλές φορές προτού το άλλο καταφέρει να εκτελέσει τη δική του εντολή ή οι εντολές assembly να εκτελεστούν με λανθασμένη σειρά.

- *Επεκτείνετε τον κώδικα του simplesync.c ώστε να υπάρχει συγχρονισμός.*

Αλλάζουμε λίγο τον κώδικα μας:

```
pthread_mutex_t 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) {//Use atomic operations to sync
                        /* ... */
                        /* You can modify the following line */

                        __sync_fetch_and_add(&ip, 1);



                        /* ... */
                } else {//Use mutex locks to sync
                        /* ... */
                        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) {//Use atomic operations to sync
                        __sync_fetch_and_sub(&ip, 1);
                        /* ... */
                        /* You can modify the following line */

                        /* ... */
                } else {//Use mutex locks to sync
                        pthread_mutex_lock(&mutex);
                        /* ... */
                        /* You cannot modify the following line */
                        --(*ip);
                        /* ... */
                        pthread_mutex_unlock(&mutex);
                }
        }
        fprintf(stderr, "Done decreasing variable.\n");

        return NULL;
}
```

```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;
}
```

Ερωτήσεις

Ερωτήσεις 1 και 2

Για το «**κακό**» αρχείο, χωρίς συγχρονισμό:

```
oslab16@orion:~/ask3$ time ./simplesyncbad
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -5207161.

real    0m0.293s
user    0m0.152s
sys     0m0.000s
```

Ενώ, για τα **διορθωμένα** και συγκεκριμένα για το **atomic** εκτελέσιμο έχουμε έξοδο:

```
oslab16@orion:~/ask3$ time ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m1.122s
user    0m0.608s
sys     0m0.000s
```

Ενώ για το **mutex**:

```
oslab16@orion:~/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    0m3.265s
user    0m1.596s
sys     0m0.016s
```

Παρατηρούμε πως το «**κακό**» αρχείο χρειάζεται **λιγότερο χρόνο** εκτέλεσης από ότι τα διορθωμένα. Αυτό συμβαίνει διότι στην πρώτη περίπτωση κάθε νήμα έχει ταυτόχρονη πρόσβαση στην κοινή μεταβλητή και εκτελεί ταυτόχρονα τις εντολές που του έχουν οριστεί(παράλληλος υπολογισμός). Στην δεύτερη περίπτωση, η μνήμη δεσμεύεται πρώτα για το ένα νήμα και αφού αυτό τελειώσει, δεσμεύεται από το άλλο, αυξάνοντας έτσι τον χρόνο εκτέλεσης.

Παρατηρούμε επίσης πως το **atomic** εκτελέσιμο είναι **πιο γρήγορο** από το mutex. Αυτό συμβαίνει διότι οι atomic εντολές είναι χαμηλού επιπέδου, σε αντίθεση με τα POSIX mutexes, τα οποία καταλαμβάνουν χρόνο και πόρους από το λειτουργικό σύστημα κάθε φορά που καλούμε τις συναρτήσεις **lock** και **unlock**, οι οποίες με τη σειρά τους για να εκτελεσθούν απαιτούν **system-calls**.

Ερωτήσεις 3 και 4

Για το εκτελέσιμο atomic, η εντολή **sync_fetch_and_add** βρίσκεται στην γραμμή 52, ενώ το for-loop στην γραμμή 47. Εκτελώντας **gcc -DSYNC-ATOMIC -S -g simplesync.c** έχουμε:

```
        .loc 1 47 0
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        .loc 1 52 0
        lock addq       $1, -16(%rbp)
        .loc 1 47 0
        addl    $1, -4(%rbp)
.L2:
        .loc 1 47 0 is_stmt 0 discriminator 1
        cmpl    $9999999, -4(%rbp)
        jle     .L3
```

Αντίστοιχα, για το mutex εκτελέσιμο, όπου οι lock, ++(*ip) και unlock είναι στις γραμμές 59, 61 και 62 αντίστοιχα, έχουμε:

```
        .loc 1 47 0
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        .loc 1 59 0
        movl    $mutex, %edi
        call    pthread_mutex_lock
        .loc 1 61 0
        movq    -16(%rbp), %rax
        movl    (%rax), %edx
        addl    $1, %edx
        movl    %edx, (%rax)
        .loc 1 62 0
        movl    $mutex, %edi
        call    pthread_mutex_unlock
        .loc 1 47 0
        addl    $1, -4(%rbp)
```

3.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

(Δεν έγιναν αλλαγές στις συναρτήσεις compute_mandel_line και output_mandel_line)

3.2.1 Με semaphores

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

```c
//Set global variables for semaphores and threads number so all threads have access to them
int nr_threads;
sem_t *sema;

void sigint_handler(int signal){//Signal Handler:Reset color if changed(We used Ctr-C while program was executed)
        reset_xterm_color(1);
        exit(1);
}
```

```c
void *compute_and_output_mandel_line(void *current_thread)
{
        /*
         * A temporary array, used to hold color values for the line being drawn
         */
        int color_val[x_chars];

        for(int i=(int)current_thread; i<y_chars; i+=nr_threads){//For the lines each thread is responsible

                compute_mandel_line(i, color_val);//We dont need this part in the critical section(Parallel computation)

                sem_wait(&sema[(int)current_thread]);//If it is current thread's turn to print: semaphore will be unlocked so thread can enter
                                                     //critical section otherwise wait

                output_mandel_line(1, color_val);//Output the line

                sem_post(&sema[ ((int)current_thread+1) % nr_threads]);//Unlock semaphore of next responsible thread to print(We unlock circularly)
        }
        return NULL;
}
```

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

        nr_threads = atoi(argv[1]);//Get number of threads from input and turn string to int

        sema = malloc(nr_threads * sizeof(sem_t));//Allocate memory for semaphores(We use one semaphore for each thread)

        pthread_t thread[nr_threads];//Create threads table

        signal(SIGINT, sigint_handler);//Check for Ctr-C

        if((argc!=2) || (nr_threads<=0)){//Check arguments
                fprintf(stderr, "Only one argument needed:number of threads (must be positive)");
                exit(1);
        }

        //Initialize semaphores(Only semaphore of thread[0] is unlocked at the beginning)
        sem_init(&sema[0], 0, 1);
        for(int i=1; i<nr_threads; i++){
                sem_init(&sema[i], 0, 0);}

        //Create threads
        for(int i=0; i<nr_threads; i++){
                pthread_create(&thread[i], NULL, compute_and_output_mandel_line, (void*)i);}

        //Wait threads to terminate
        for(int i=0; i<nr_threads; i++){
                pthread_join(thread[i], NULL);}

        //Destroy semaphores
        for(int i=0; i<nr_threads; i++){
               sem_destroy(&sema[i]);}

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

3.2.2 Με condition variables(Παραθέτουμε δύο εκδοχές)

3.2.2.1: Πολλά condition variables

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

```
//Set global variables(Visible by all threads)
int nr_threads;
pthread_mutex_t mut;
pthread_cond_t *cond;
int current=0;

void sigint_handler(int signal){//Signal handler for Ctr-C(Reset color)
        reset_xterm_color(1);
        exit(1);
}
```

```c
void *compute_and_output_mandel_line(void *current_thread){

	for (int i = (int)current_thread; i < y_chars; i+=nr_threads) {//For the lines each thread is responsible

		int color_val[x_chars];

		compute_mandel_line(i, color_val);//We dont need this part in the critical section(Parallel computation)

		pthread_mutex_lock(&mut);//Enter critical section

		while (current != i % nr_threads) {//Check if it is current thread's turn to output.If not let it wait
						//and give access to lock to other threads
			pthread_cond_wait(&cond[i%nr_threads], &mut);
		}

		output_mandel_line(1, color_val);//Output line

		current=(current+1)%nr_threads;//Next responsible thread circularly

		pthread_cond_signal(&cond[current]);//Awake next responsible thread(We unlock circularly)

		pthread_mutex_unlock(&mut);//Exit critical section
	}

	return NULL;
}
```

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

	nr_threads = atoi(argv[1]);//Get number of threads from input and turn string to int

	pthread_t thread[nr_threads];//Create threads table
	cond=malloc(nr_threads*sizeof(pthread_cond_t));//Allocate memory for condition variables

	//Initialize mutex and condition variables
	pthread_mutex_init(&mut, NULL);
	for(int i=0; i<nr_threads; i++){
	pthread_cond_init(&cond[i], NULL);}

	signal(SIGINT, sigint_handler);//Check for Ctrl-C

	if((argc!=2) || (nr_threads<=0)){//Check arguments
		fprintf(stderr, "Only one argument needed:number of threads(must be positive)");
		exit(1);
	}

	//Create threads
	for(int i=0; i<nr_threads; i++)
		pthread_create(&thread[i], NULL, compute_and_output_mandel_line, (void*)i);

	//Wait for threads to terminate
	for(int i=0; i<nr_threads; i++)
		pthread_join(thread[i], NULL);

	//Destroy mutex and condition variables
	for(int i=0; i<nr_threads; i++){
	pthread_cond_destroy(&cond[i]);
	}
	pthread_mutex_destroy(&mut);


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

3.2.2.2: Ένα condition variable

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.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;
```

```c
//Set global variables(Visible by all threads)
int nr_threads;
pthread_mutex_t mut;
pthread_cond_t cond;
int current=0;

void sigint_handler(int signal){//Signal handler for Ctrl-C
        reset_xterm_color(1);
        exit(1);
}
```

```c
void *compute_and_output_mandel_line(void *current_thread){

        for (int i = (int)current_thread; i < y_chars; i+=nr_threads) {//For the lines each thread is responsible

                int color_val[x_chars];

                compute_mandel_line(i, color_val);//We dont need this part in the critical section(Parallel computation)

                pthread_mutex_lock(&mut);//Enter critical section

                while (current != i % nr_threads) {//Check if it is current thread's turn to output.If not let it wait
                                                   //and give access to lock to other threads
                        pthread_cond_wait(&cond, &mut);
                }

                output_mandel_line(1, color_val);//Output line

                current=(current+1)%nr_threads;//Next thread circularly

                pthread_cond_broadcast(&cond);//Awake all waiting threads

                pthread_mutex_unlock(&mut);//Exit critical section
        }

        return NULL;
}
```

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

        nr_threads = atoi(argv[1]);//Get number of threads from input and turn string to int

        pthread_t thread[nr_threads];//Create threads table

        //Initialize mutex and condition variable
        pthread_mutex_init(&mut, NULL);
        pthread_cond_init(&cond, NULL);

        signal(SIGINT, sigint_handler);//Check for Ctrl-C

        if((argc!=2) || (nr_threads<=0)){//Check arguments
                fprintf(stderr, "Only one argument needed:number of threads(must be positive)");
                exit(1);
        }

        //Create threads
        for(int i=0; i<nr_threads; i++)
                pthread_create(&thread[i], NULL, compute_and_output_mandel_line, (void*)i);

        //Wait for threads to terminate
        for(int i=0; i<nr_threads; i++)
                pthread_join(thread[i], NULL);

        //Destroy mutex and condition variable
        pthread_cond_destroy(&cond);
        pthread_mutex_destroy(&mut);


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

Ερωτήσεις

Ερώτηση 1

Μέσω της εντολής:

sem_t *sema= malloc(nr_threads * sizeof(sem_t));

Ορίζουμε τόσους σημαφόρους όσο και το πλήθος των threads που απαιτεί ο χρήστης. Ξεκλειδώνουμε τους σημαφόρους κυκλικά αφού τους έχουμε αρχικοποιήσει κατάλληλα(Σημαφόρος του thread[0] unlocked και οι υπολοιποι locked), δίνοντας κάθε φορά πρόσβαση στο κατάλληλο thread για να εκτυπώσει την γραμμή του Mandelbrot που του αναλογεί.

Ερώτηση 2

Το σειριακό πρόγραμμα μας δίνει:

```
real    0m4.618s
user    0m1.432s
sys     0m0.056s
```

Ενώ το παράλληλο, για 2 threads:

```
real    0m2.807s
user    0m1.464s
sys     0m0.032s
```

Παρατηρούμε πως ο συνολικός χρόνος εκτέλεσης μειώνεται(σχεδόν υποδιπλασιάζεται), όπως θα περιμέναμε λόγω παράλληλου υπολογισμού των γραμμών(Θέτοντας την compute_mandel_line εκτός του κρίσιμου τμήματος όπως θα εξηγήσουμε και στο ερώτημα 4). Δηλαδή ο υπολογιστικός φόρτος κατανέμεται σε 2 threads αντί για ένα με αποτέλεσμα την γρηγορότερη απόκριση του συστήματος.

Ερώτηση 3

Όπως δείξαμε και παραπάνω υλοποιούμε δύο εκδοχές του προγράμματος, μια με 1 μεταβλητή συνθήκης και μία με πολλαπλές(όσες και ο αριθμός των threads). Στην πρώτη περίπτωση κάνουμε χρήση της εντολής pthread_cond_broadcast με την οποία «ξυπνάμε» όλα τα νήματα τα οποία βρίσκονται σε κατάσταση αναμονής(pthread_cond_wait). Αντίθετα στην δεύτερη εκδοχή χρησιμοποιούμε την pthread_cond_signal για να ενεργοποιήσουμε το αμέσως επόμενο νήμα(που περιμένει) που είναι υπεύθυνο για να τυπώσει την γραμμή του. Όπως φαίνεται στην περίπτωση με τις πολλαπλές μεταβλητές συνθήκης έχουμε έναν πιο ελεγχόμενο τρόπο για να «κοιμίζουμε» και να «αφυπνίζουμε» τα κατάλληλα νήματα ενώ στο άλλο πρόγραμμα θα μπορούσε ένα νήμα να βγει και να ξαναμπεί σε κατάσταση αναμονής χωρίς λόγο.

Ερώτηση 4

Το παράλληλο πρόγραμμα που φτιάξαμε παρουσιάζει επιτάχυνση. Αυτό συμβαίνει διότι το κρίσιμο τμήμα περιλαμβάνει μόνο τη φάση εκτύπωσης των γραμμών και όχι τον υπολογισμό αυτών.

Ο υπολογισμός, σε αντίθεση με την εκτύπωση, δεν χρειάζεται να είναι μέρος του κρίσιμου τμήματος, αφού μπορεί να υλοποιηθεί από πολλαπλά threads ταυτόχρονα χωρίς να υπάρξει πρόβλημα, ενώ την εκτύπωση την αναλαμβάνει ένα thread ανά φορά ώστε να εμφανιστεί η κάθε γραμμή στην έξοδο με την σωστή σειρά.

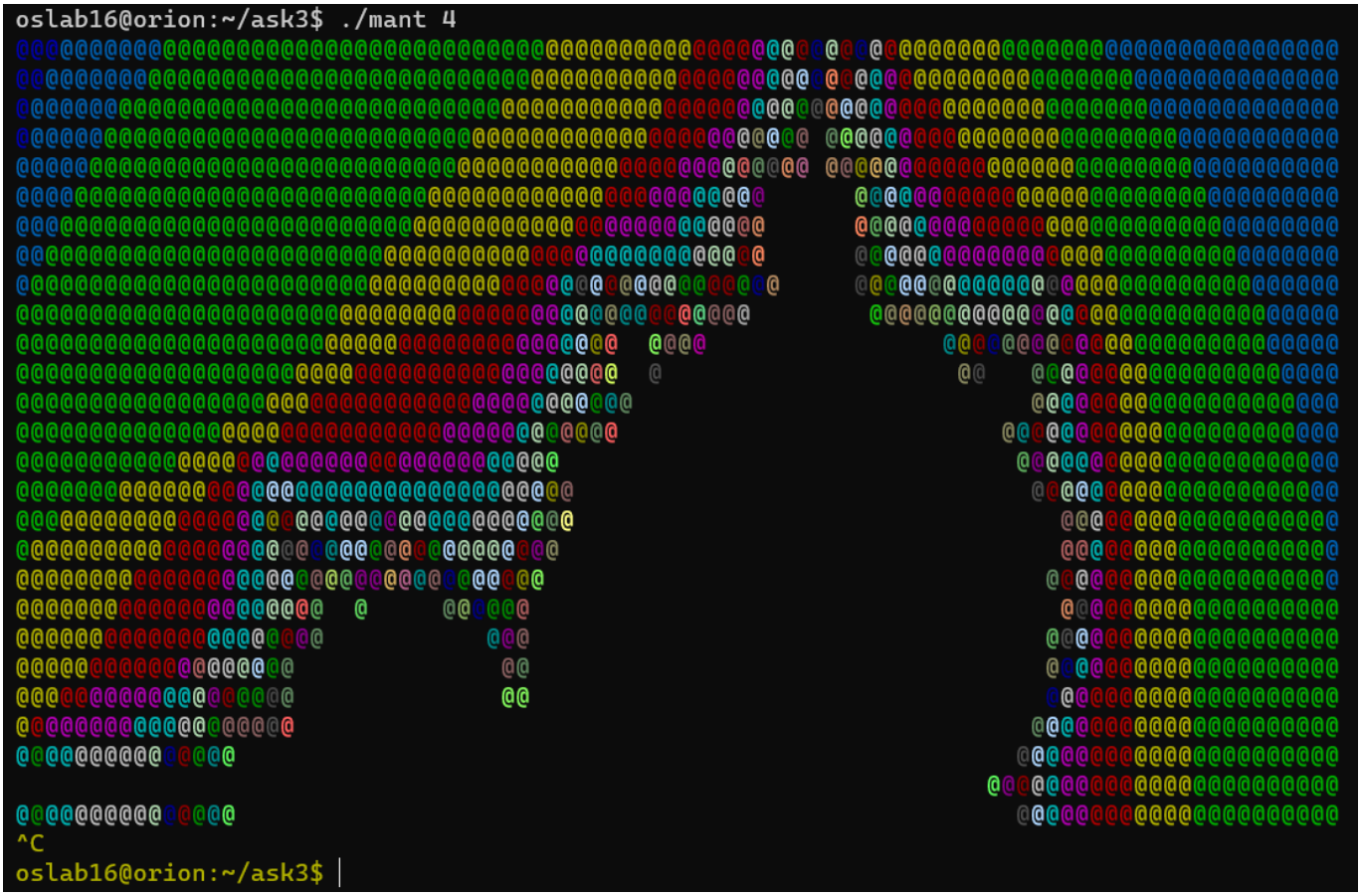
Έτσι, για 5 και 9 threads έχουμε χρόνο εκτέλεσης αντίστοιχα:

```
real    0m0.885s
user    0m0.760s
sys     0m0.008s
```

```
real    0m0.680s
user    0m0.756s
sys     0m0.016s
```

Ερώτηση 5

Εάν πατήσουμε Ctrl-C ενώ το πρόγραμμα εκτελείται, αλλάζει το default χρώμα της γραμμής εντολών και υιοθετεί ένα από τα χρώματα που χρησιμοποιεί το Mandelbrot:



Για να εξασφαλίσουμε ότι ακόμη κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του προσθέτουμε αυτή την συνάρτηση:

```c
void sigint_handler(int signal){
        reset_xterm_color(1);
        exit(1);
}
```

Και αυτή τη γραμμή κώδικα στην main:

```c
signal(SIGINT, sigint_handler);
```

Πλέον, όταν εκτελούμε Ctrl-C, το τερματικό είναι βαμμένο στα default χρώματα του: