



ΣΗΜΜΥ ΕΜΠ

Λειτουργικά Συστήματα / 3η εργαστηριακή αναφορά

Νικόλαος Πηγαδάς / el18445

Γιώργος Αγγέλης / el18030

6ο εξάμηνο/ Ακαδημαϊκό έτος 2020-2021

Ομάδα Oslaba77

## OS αναφορά 3 - Συγχρονισμός

### 1.1 Συγχρονισμός σε υπάρχοντα κώδικα

#### Πηγαίος κώδικας

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

/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
 * This macro helps with error reporting in this case.
 */
#define perror_thread(ret, msg) \
    do { errno = ret; perror(msg); } while (0)

#define N 10000000

/* Dots indicate lines where you are free to insert code at will */
/* ... */
pthread_mutex_t mutex;
/* ... */
#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
```

```

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

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

    return NULL;
}

```

```

int main(int argc, char *argv[])
{
    int val, ret, ok;
    pthread_t t1, t2;
    pthread_mutex_init(&mutex, NULL);
    /*
     * 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");

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

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

```

---

## Λίγα λόγια για την Άσκηση

Στο πρώτο ερώτημα αυτής της εργαστηριακής Άσκησης κληθήκαμε να χρησιμοποιήσουμε μηχανισμούς συγχρονισμού προκειμένου να επιλύσουμε ζητήματα συγχρονισμού μεταξύ δύο νημάτων. Συγκεκριμένα, στα δύο for loops εντός των **increase()** και **decrease()** δουλεύουμε πρώτα με **ατομικές λειτουργίες** (atomic operations) και ύστερα με **κλειδώματα POSIX** (POSIX mutexes). Στόχος μας είναι αφού τρέξει το πρόγραμμα να παραχθούν τα δύο εκτελέσιμα αρχεία simplesync-mutex και simplesync-atomic τα οποία θα εμφανίζουν στην οθόνη μας σχετικό μήνυμα που θα επιβεβαιώνει τον συγχρονισμό των δύο νημάτων κάθε φορά. Δηλαδή, αφού το ένα νήμα αυξάνει την μεταβλητή val κατά 1 και το δεύτερο νήμα μειώνει την μεταβλητή val κατά 1 στο τέλος αναμένουμε το πρόγραμμα να μας εμφανίσει το μήνυμα 'OK, val = 0'.

**Σημείωση:** Στον κώδικα που παραθέσαμε παραπάνω με έντονο χρώμα φαίνονται οι γραμμές στις οποίες πραγματοποιήσαμε αλλαγές.

### Ερώτημα 1

Θα χρησιμοποιήσουμε την εντολή time για να μετρήσουμε τον χρόνο εκτέλεσης των εκτελέσιμων.

	Εκτελέσιμο	Χρόνος	
Χωρίς συγχρονισμό	simplesync ( -atomic και -mutex είναι ίδια)	real	0m0.039s
		user	0m0.072s
		sys	0m0.000s
Με συγχρονισμό	simplesync-atomic	real	0m0.421s
		user	0m0.832s
		sys	0m0.004s
	simplesync-mutex	real	0m27.560s
		user	0m26.832s
		sys	0m28.272s

Παρατηρούμε ότι χωρίς συγχρονισμό το πρόγραμμα τρέχει σε πολύ σύντομο χρόνο (0.039sec). Με συγχρονισμό όμως το πρόγραμμα τρέχει σε μεγαλύτερο χρόνο. Συγκεκριμένα, με atomic operations το πρόγραμμα τρέχει σε 0.421 sec, που είναι περίπου 10 φορές περισσότερο από ότι χωρίς συγχρονισμό. Με POSIX mutexes ωστόσο, το πρόγραμμα τρέχει σε πολύ πιο μεγάλο χρόνο, περίπου 707 φορές μεγαλύτερο.

Στην πραγματικότητα αναμέναμε τα προγράμματα με συγχρονισμό να τρέχουν σε μεγαλύτερο χρόνο, αφού υπάρχουν τμήματα κώδικα όπου σε μία χρονική στιγμή τρέχει μόνο ένα νήμα, ωστόσο δεν θα μπορούσαμε να προβλέψουμε το μέγεθος της χρονικής διαφοράς μεταξύ του αρχείου -mutex και του αρχείου -atomic.

## Ερώτημα 2

Όπως είδαμε στο Έρώτημα 1 η χρήση atomic operations είναι πολύ πιο γρήγορη από τη χρήση POSIX mutexes. Αυτό συμβαίνει γιατί τα **mutexes** τρέχουν στον **Kernel** του συστήματος (αφορούν δηλαδή το software) και επιτρέπουν κάθε φορά σε ένα από τα νήματα να εκτελέσει τον κώδικα που βρίσκεται στο κρίσιμο τμήμα, κάτι που μπορεί να αποδειχθεί εξαιρετικά χρονοβόρο. Αντίθετα, τα **atomic operations** λαμβάνουν χώρα στη **CPU** του συστήματος (αφορούν δηλαδή το hardware) και δεν διακόπτονται από κανένα νήμα και καμία διεργασία, εξού και η μεγάλη χρονική διαφορά με τα POSIX mutexes. Η χρονική διαφορά μεταξύ των δύο μεθόδων συγχρονισμού έχει να κάνει και με το ότι οι atomic operations μεταφράζονται σε λιγότερες εντολές σε γλώσσα assembly, σε σχέση με τα POSIX mutexes.

## Ερώτημα 3

Πληκτρολογούμε στο terminal την εντολή:

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

και δημιουργείται το αρχείο simplesync.s που περιέχει τον ενδιάμεσο κώδικα Assembly. Παραθέτουμε το τμήμα κώδικα που αφορά τα atomic operations:

```
.L3:
    .loc 1 41 0
    movq    -16(%rbp), %rax
    lock addl    $1, (%rax)
    .loc 1 37 0
    addl     $1, -4(%rbp)
```

## Ερώτημα 4

Πληκτρολογούμε στο terminal την εντολή:

```
gcc -Wall -DSYNC_MUTEX -S -g simplesync.c
```

και δημιουργείται το αρχείο simplesync.s που περιέχει τον ενδιάμεσο κώδικα Assembly. Παραθέτουμε το τμήμα κώδικα που αφορά τα POSIX mutexes:

```
.L3:
    .loc 1 47 0
    movl     $mutex, %edi
    call     pthread_mutex_lock
    .loc 1 48 0
    movq     -16(%rbp), %rax
    movl     (%rax), %eax
    leal     1(%rax), %edx
    movq     -16(%rbp), %rax
    movl     %edx, (%rax)
    .loc 1 49 0
    movl     $mutex, %edi
    call     pthread_mutex_unlock
    .loc 1 37 0
    addl     $1, -4(%rbp)
```

## Παρατηρήσεις για τα ερωτήματα 3,4

Παρατηρούμε ότι το τμήμα των mutexes σε Assembly είναι κατά πολύ μεγαλύτερο από το αντίστοιχο τμήμα των atomic operations, κάτι που δικαιολογεί το ότι τα mutexes καθυστερούν περισσότερο να επιτύχουν συγχρονισμό σε σχέση με τα atomic operations. Αξίζει πάντως να σημειώσουμε ότι δεν μπορούμε να αντικαταστήσουμε τα mutexes με atomic operations κατά βούληση σε κάθε περίπτωση, τόσο γιατί η χρήση των atomic operations δεσμεύει τμήμα της CPU, όσο και γιατί απαιτούν την εις βάθος κατανόηση τους πριν εισαχθούν σωστά στον κώδικα.

## 1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

### Πηγαίος κώδικας

---

```
#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 "mandel-lib.h"
#include <signal.h>

#define MANDEL_MAX_ITERATION 100000

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

/*
 * Output at the terminal 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);
    }
}

```

```

void intHandler (int signum){
    reset_xterm_color(1);
    exit(1);
}

int NTHREADS;
sem_t *semaphore;
void help_sem(void){
    semaphore = (sem_t*) malloc(NTHREADS * sizeof(sem_t));
}

void compute_and_output_mandel_line(int fd, int line)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];
    compute_mandel_line(line, color_val);
    sem_wait(&semaphore[line%NTHREADS]);
    output_mandel_line(fd, color_val);
    sem_post(&semaphore[(line+1)%NTHREADS]);
    // reset_xterm_color(1);
}

void* thready(void* arg){
    int in=(int*)arg;
    int k;
    for (k=in ; k < 50 ; k+=NTHREADS) {
        compute_and_output_mandel_line(1,k);
    }

    return NULL;
}

int main(int argc, char **argv)
{
    int i, j, ret;

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

    /* draw the Mandelbrot Set, one line at a time. Output is sent to file descriptor '1', i.e.,
    standard output.*/

    if (argc<2){
        fprintf(stderr, "wrong input\n");
        return 1;
    }

    sscanf(argv[1], "%d", &NTHREADS);
    pthread_t th[NTHREADS];
    help_sem();
    sem_init(&semaphore[0], 0, 1);
    for (i=1 ; i < NTHREADS; i++)
    {

```



```

        sem_init(&semaphore[i], 0, 0);
    }

    signal(SIGINT, intHandler);

    for (i=0 ; i < NTHREADS; i++) {
        int *x = malloc(sizeof(int));
        *x=i;
        ret = pthread_create(&th[i], NULL, &thready, x);
        if (ret) {
            perror("pthread_create");
            exit(1);
        }
    }

    for (i = 0; i < NTHREADS; i++) {
        ret = pthread_join(th[i], NULL);
        if (ret){
            perror("pthread_join");
            exit(1);
        }
    }
    for (j=0 ; j < NTHREADS; j++) {
        sem_destroy(&semaphore[j]);
    }
    reset_xterm_color(1);
    return 0;
}

```

## Σχόλια

Εδώ γίνεται έλεγχος ότι ο χρήστης έδωσε αριθμό νημάτων και τον διαβάζουμε, αλλιώς έχουμε error:

```

if (argc<2){
    fprintf(stderr, "wrong input\n");
    return 1;
}

sscanf(argv[1], "%d", &NTHREADS);

```

Ο αριθμός των thread ορίζεται σαν global μεταβλητή, γιατί χρησιμοποιείται από συναρτήσεις και αποφύγαμε να τον περνάμε συνεχώς σαν όρισμα. Ορίζουμε πίνακα των threads με μέγεθος ίσο με το δοσμένο πλήθος τους. Από την άλλη, το μέγεθος του πίνακα σημαφόρων είναι όσα τα NTHREADS που μας δίνει ο χρήστης, για αυτό και χρησιμοποιήθηκε βοηθητική συνάρτηση που ορίζει τον πίνακα αυτό, κατόπιν του διαβάσματος του πλήθους των νημάτων. Και αυτός ο πίνακας δηλώνεται global επειδή χρησιμοποιείται σε συναρτήσεις και ορίστηκε με malloc:

```

int NTHREADS;
sem_t *semaphore;
void help_sem(void){
    semaphore = (sem_t*) malloc(NTHREADS * sizeof(sem_t));
}

int main(int argc, char **argv)
{
    ...
    sscanf(argv[1], "%d", &NTHREADS);
    pthread_t th[NTHREADS];
    help_sem();
    ...
}

```

Δημιουργούμε πίνακα σημαφόρων και τον αρχικοποιούμε. Ο πρώτος λαμβάνει την τιμή 1, εφόσον πρώτα θα τυπωθεί η πρώτη γραμμή από το πρώτο νήμα, ενώ οι υπόλοιποι την τιμή 0, για να κάνουν wait:

```

pthread_t th[NTHREADS];
sem_init(&semaphore[0], 0, 1);
for (i=1 ; i < NTHREADS; i++)
{
    sem_init(&semaphore[i], 0, 0);
}

```

Μέσω του pthread\_join γίνεται αναμονή για τερματισμό επόμενων νημάτων, ώστε να τερματίσει ομαλά η λειτουργία τους:

```

for (i = 0; i < NTHREADS; i++) {
    ret = pthread_join(th[i], NULL);
    if (ret){
        perror("pthread_join");
        exit(1);
    }
}

```

Μέσω της sem\_destroy καταστρέφουμε του σημαφόρους, καθώς δεν τους χρειαζόμαστε πια:

```

for (j=0 ; j < NTHREADS; j++) {
    sem_destroy(&semaphore[j]);
}

```

Τέλος, αξίζει να σημειωθεί ότι μέσα στο παρακάτω for υπάρχει η ανάγκη να περνάμε το i σαν \*arg στην pthread\_create, όμως λόγω παραλληλίας η τιμή του γίνεται overwritten από την i++ (π.χ. αν την περνούσαμε ως &i). Για να την προστατέψουμε, έχουμε τον βοηθητικό pointer \*x, αποθηκεύουμε ξεχωριστά τις τιμές που λαμβάνει μέσω malloc και περνάμε το x ως argument:

```

for (i=0 ; i < NTHREADS; i++) {
    int *x = malloc(sizeof(int));
    *x=i;
    ret = pthread_create(&th[i], NULL, &thready, x);
    if (ret) {
        perror("pthread_create");
        exit(1);
    }
}

```

Ο αναλυτικός σχολιασμός της χρήσης των σημαφόρων παρουσιάζεται παρακάτω στην απάντηση 1, ενώ η επαναφορά του χρώματος του τερματικού σε άσπρο, στην απάντηση 4.

## Απαντήσεις στις ερωτήσεις

1.

Στο πρόγραμμά μας κάναμε χρήση σημαφόρων (υλοποίηση με πίνακα), ίσων στο πλήθος με τα νήματα που ζητά ο χρήστης να δημιουργηθούν. Έχουμε αντιστοιχία 1-1 νημάτων και σημαφόρων. Με άλλα λόγια, έχουμε wait του κατάλληλου νήματος για την κάθε γραμμή απεικόνισης που του αναλογεί, μέσω του αντίστοιχου σημαφόρου. Συγκεκριμένα, για η νήματα, το i-οστό (με  $i = 0, 1, 2, \dots, n-1$ ) αναλαμβάνει τις σειρές  $i, i + n, i + 2n, i + 3n, \dots$  και τις τυπώνει όταν του το επιτρέψει ο σημαφόρος που του αντιστοιχεί. Οι σειρές πρέπει να τυπώνονται σειριακά: 0, 1, 2...

### Πρόσθετα σχόλια για την υλοποίηση των σημαφόρων

Αρχικά κάναμε ένα loop που δημιουργούσε κάθε νήμα, για το οποίο καλούμε την βοηθητική συνάρτηση thready:

```

for (i=0 ; i < NTHREADS; i++) {
    int *x = malloc(sizeof(int));
    *x=i;
    ret = pthread_create(&th[i], NULL, &thready, x);
    if (ret) {
        perror("pthread_create");
        exit(1);
    }
}

```

Μέσω της `thready`, για κάθε νήμα καλούμε την `compute_and_output_mandel_line` για κάθε γραμμή που του αντιστοιχεί, θέτοντας βήμα `NTHREADS` στο loop:

```
void* thready(void* arg){
    int in=(int*)arg;
    int k;
    for (k=in ; k < 50 ; k+=NTHREADS) {
        compute_and_output_mandel_line(1,k);
    }

    return NULL;
}
```

Εκμεταλλευόμενοι την παραλληλία των νημάτων, κάνουμε τους υπολογισμούς με τυχαία σειρά και θέτουμε μόνο το τύπων να γίνει με την επιθυμητή σειρά, για να κερδίσουμε χρόνο.

```
void compute_and_output_mandel_line(int fd, int line)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];
    compute_mandel_line(line, color_val);
    sem_wait(&semaphore[line%NTHREADS]);
    output_mandel_line(fd, color_val);
    sem_post(&semaphore[(line+1)%NTHREADS]);
    // reset_xterm_color(1);
}
```

Σε αυτό το σημείο, πρέπει να γνωρίζουμε ποιο νήμα εκτελείται, ώστε να κάνει `wait` μέσω του σημαφόρου του, αλλά εμείς έχουμε την πληροφορία σε ποια γραμμή βρισκόμαστε. Για να κάνει `wait` ο σημαφόρος του κατάλληλου νήματος, χρησιμοποιούμε την έκφραση:

`line%NTHREADS`

Από την άλλη, θέλουμε το νήμα που θα βγαίνει από το `wait` του σημαφόρου, να δίνει το δικαίωμα στο επόμενο νήμα να τυπώσει, αφού αυτό περιέχει την επόμενη γραμμή. Για την ενέργεια αυτή, χρησιμοποιούμε την έκφραση

`(line+1)%NTHREADS`

**Χρόνος ολοκλήρωσης σειριακού προγράμματος:**

[illegible]

**Χρόνος ολοκλήρωσης παράλληλου προγράμματος με 2 νήματα:**

[illegible]

Παρατηρούμε ότι χάρη στον παράλληλο υπολογισμό μέσω δύο νημάτων, το αποτέλεσμα μειώνεται στο μισό σε σχέση με την σειριακή εκτέλεση.

## Πρόσθετες παρατηρήσεις για τον αριθμό νημάτων

Για 3 νήματα, οι υπολογισμοί γίνονται ακόμα πιο γρήγορα:

```
[oslaba77@os-node1:~/ask3$ time ./mandel 3
real    0m0.356s
user    0m0.980s
sys     0m0.032s
```

Παρατηρούμε ότι η ταχύτητα των υπολογισμών αυξάνεται μέχρι την τιμή 50, γεγονός λογικό αν αναλογιστούμε ότι οι υπολογισμοί που κάνουμε αφορούν τις 50 γραμμές απεικόνισης του mandelbrot set. Συγκεκριμένα:

```
[oslaba77@os-node1:~/ask3$ time ./mandel 50
real    0m0.148s
user    0m0.992s
sys     0m0.020s
```

Από την άλλη, για περισσότερα νήματα, ο χρόνος εκτέλεσης είναι περίπου ίσος με αυτόν για 50 νήματα. Παρουσιάζεται πολύ μικρή καθυστέρηση, αναμενόμενο καθώς φορτώνουμε το πρόγραμμα με παραπάνω αρχικοποιήσεις, μνήμη και νήματα, όμως η παραλληλία των 50 νημάτων επιτυγχάνεται:

```
[oslaba77@os-node1:~/ask3$ time ./mandel 100
real    0m0.151s
user    0m0.992s
sys     0m0.032s
```

```
[oslaba77@os-node1:~/ask3$ time ./mandel 51
real    0m0.151s
user    0m0.996s
sys     0m0.016s
```

Και προφανώς για 1 νήμα η εκτέλεση είναι ίση χρονικά με την σειριακή:

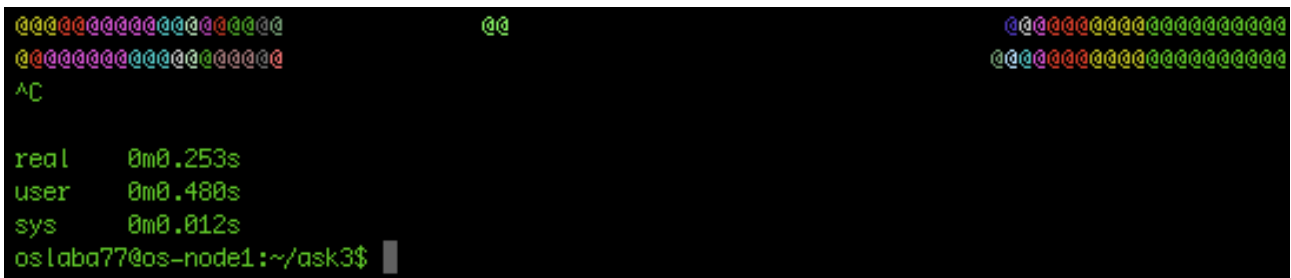
```
[oslaba77@os-node1:~/ask3$ time ./mandel 1
real    0m1.033s
user    0m0.980s
sys     0m0.028s
```

3.

Η εκτέλεση του προγράμματος επιταχύνεται όταν αναθέτουμε την δουλειά σε περισσότερα νήματα, μέχρι το πλήθος των 50 νημάτων, όπως δείχθηκε και παραπάνω. Πλήθος μεγαλύτερο των 50 δεν έχει νόημα, καθώς οι απαιτούμενοι υπολογισμοί είναι 50, ένας για κάθε γραμμή απεικόνισης του mandelbrot set. Έχουμε επιτάχυνση, επειδή οι υπολογισμοί καταμερίζονται ανάμεσα στα νήματα και εκτελούνται παράλληλα. Άρα, όταν έρθει η στιγμή να τυπωθούν είναι ήδη υπολογισμένα. Σε περίπτωση που μέσα στον σημαφόρο (κρίσιμο τμήμα), ανάμεσα στο wait και το post, βρισκόταν και ο υπολογισμός μαζί με την εκτύπωση, τότε η εκτέλεση θα ήταν όσο αργή όσο η σειρική. Αυτό θα συνέβαινε γιατί η σειρά των υπολογισμών θα ακολουθούσε την σειρά εκτύπωσης και δεν θα πραγματοποιούνταν παράλληλα, όπως ακριβώς γίνεται και στο σειριακό πρόγραμμα.

4.

Το τερματικό, με το πάτημα του Ctrl-C καθώς το πρόγραμμα εκτελείται, από άσπρο χρώμα αποκτά το χρώμα που είχε υπολογιστεί ότι θα είχε το mandelbrot set, την στιγμή που το πατήσαμε.



Ο λόγος που όταν τερματίζει το πρόγραμμα από μόνο του, δεν αλλάζει το χρώμα του τερματικού, είναι επειδή το πρόγραμμα παρατείνεται με την εντολή `reset_xterm_color(1);` στο τέλος, η οποία επαναφέρει το χρώμα στο άσπρο. Αλλιώς, θα άλλαζε και πάλι το χρώμα του τερματικού. Για να επαναφέρουμε το χρώμα του τερματικού σε άσπρο, χρησιμοποιούμε την εντολή `reset`. Για επιτελεστεί αυτόματα αυτή η λειτουργία υπάρχουν δύο τρόποι.

Ο πρώτος τρόπος είναι μετά την εκτύπωση κάθε γραμμής, να επαναφέρουμε το χρώμα της επόμενης στο προκαθορισμένο, δηλαδή σε άσπρο. Μάλιστα, η λύση αυτή μετατρέπει και το ^C σε άσπρο, ενώ δεν χρειάζεται `reset` των χρωμάτων στο τέλος του προγράμματος (στην main):

```
void compute_and_output_mandel_line(int fd, int line)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];
    compute_mandel_line(line, color_val);
    sem_wait(&semaphore[line%NTHREADS]);
    output_mandel_line(fd, color_val);
    sem_post(&semaphore[(line+1)%NTHREADS]);
    reset_xterm_color(1);
}
```



```

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
^C
os lab77@os-node1:~/ask3$ █

```

Μια πιο αποδοτική λύση χρονικά, υλοποιείται με την χρήση σημάτων. Συγκεκριμένα, μέσω ενός signal handler, εντοπίζουμε το system call Ctrl-C και το χειριζόμαστε μέσω μιας συνάρτησης όπως επιθυμούμε. Εδώ, κάνουμε reset των χρωμάτων στο προκαθορισμένο άσπρο. Η μεγαλύτερη απόδοση έγκειται στο γεγονός ότι η πρώτη λύση πρέπει να τρέξει για κάθε γραμμή, ενώ ο signal handler καλείται, υπάρχει στο παρασκήνιο και τρέχει μόνο αν κληθεί, εξοικονομώντας μας χρόνο. Εδώ αλλάζει το χρώμα του ^C, ενώ χρειάζεται η εντολή reset χρωμάτων στο τέλος του προγράμματος, για την περίπτωση που αυτό δεν τερματίσει πρόωρα:

```

void intHandler (int signum){
    reset_xterm_color(1);
    exit(1);
}

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

    signal(SIGINT, intHandler);

    ...

    reset_xterm_color(1);
    return 0;
}

```

```

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
^C os lab77@os-node1:~/ask3$ █

```

ΤΕΛΟΣ