Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα , Ροή Υ

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2019-2020

Άσκηση 3: Συγχρονισμός

Στοιχεία σπουδαστών:

Όνομα: Μαντζούτας Ανδρέας Α.Μ. : 03117108

Όνομα: Τσιτσής Αντώνιος Α.Μ. : 03117045

Ομάδα εργαστηρίου: Oslabc23

Εξάμηνο: $6^{ο}$

Ημερομηνία: 02/06/2020

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Σε αυτήν την άσκηση, κληθήκαμε να συγχρονίσουμε 2 νήματα, τα οποία τρέχουν ταυτόχρονα στον κώδικα που μας δόθηκε. Πιο συγκεκριμένα, το ένα νήμα αυξάνει την τιμή μιας μεταβλητής κατά 1, ενώ το άλλο τη μειώνει κατά 1, Ν φορές το καθένα.
Συνεπώς, θα περιμέναμε η τιμή της μεταβλητής να παραμείνει σταθερή, ωστόσο αυτό δεν συνέβαινε κατά την εκτέλεση στο αρχικό πρόγραμμα, όπως φαίνεται παρακάτω:

```
oslabc23@os-node1:~/andr/3/oxi_etoima$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -3808975.
```

```
oslabc23@os-node1:~/andr/3/oxi_etoima$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 9999821.
```

Επίσης, να σημειώσουμε ότι το τελικό αποτέλεσμα της τιμής val είναι τυχαίο, δηλαδή αλλάζει από εκτέλεση σε εκτέλεση.
Στην άσκηση, ο συγχρονισμός γίνεται με δύο τρόπους, με χρήση ατομικών λειτουργιών του GCC και με χρήση mutex. Αφού μεταγλωττίσαμε το πρόγραμμα με χρήση Makefile, παρατηρούμε ότι παράγονται 2 εκτελέσιμα αρχεία από τον ίδιο πηγαίο κώδικα. Παρατηρούμε, λοιπόν, ότι στον δοσμένο κώδικα υπάρχει το παρακάτω τμήμα κώδικα:

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

Αντίστοιχα, στο δοσμένο makefile, υπάρχουν οι εξής εντολές:

```
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
```

Επομένως, καταλαβαίνουμε ότι ανάλογα με την τιμή του USE_ATOMIC_OPS, γίνεται διαφορετικός συγχρονισμός και παράγεται το αντίστοιχο εκτελέσιμο.

Συγχρονισμός με ατομικές λειτουργίες του GCC

Για την υλοποίηση του συγχρονισμού με ατομικές λειτουργίες του GCC, χρησιμοποιήσαμε τις εντολές __sync_add_and_fetch(ip,1) και __sync_sub_and_fetch(ip,1). Οι εντολές αυτές προκαλούν την αύξηση και μείωση του ip κατά 1 αντίστοιχα, ατομικά. Πιο αναλυτικά, εξασφαλίζουν ότι η τιμή της μεταβλητής αποθηκεύεται στη θέση μνήμης που αντιστοιχεί, μετά την εφαρμογή της επιθυμητής λειτουργίας και έπειτα, θα είναι δυνατή κάθε αλλαγή στο περιεχόμενο αυτής. Επομένως, το κλείδωμα υλοποιείται σε επίπεδο υλικού.

Εκτελώντας το πρόγραμμα, παίρνουμε την αναμενόμενη έξοδο, και επομένως συμπεραίνουμε ότι ο συγχρονισμός μας λειτουργεί σωστά.

```
oslabc23@os-node1:~/andr/3/etoima$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

Συγχρονισμός με χρήση POSIX mutexes

Σε αυτήν την περίπτωση, ο συγχρονισμός επιτυγχάνεται με POSIX mutexes. Πιο συγκεκριμένα, η υλοποίηση αυτή βασίζεται στην χρήση κλειδωμάτων. Έτσι, κάθε φορά μπορεί να βρίσκεται μόνο ένα νήμα στο κρίσιμο τμήμα. Όταν μπει το πρώτο νήμα, «κλειδώνει» το κρίσιμο τμήμα, με αποτέλεσμα το επόμενο να περιμένει να τελειώσει το πρώτο, ώστε να «ξεκλειδώσει»(αφού τελειώσει τη λειτουργία του) και ύστερα να μπει.
Και αυτή τη φορά, επιβεβαιώνουμε τη σωστή λειτουργία του συγχρονισμού από την εκτέλεση του προγράμματος.

```
oslabc23@os-node1:~/andr/3/etoima$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Παρακάτω παραθέτουμε τον πηγαίο κώδικα που συγγράψαμε για την υλοποίηση της άσκησης.

```
/*
 * simplesync.c
 *
 * A simple synchronization exercise.
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Operating Systems course, ECE, NTUA
 *
 */

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

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

#define N 10000000

pthread_mutex_t locker;
/* ... */
#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) {
			__sync_add_and_fetch(ip,1);
		} else {
			pthread_mutex_lock(&locker);
			++(*ip);
			pthread_mutex_unlock(&locker);
		}
	}
	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) {
			/* ... */
			__sync_sub_and_fetch(ip,1);
			/* ... */
		} else {
			/* ... */
			pthread_mutex_lock(&locker);
			--(*ip);
			pthread_mutex_unlock(&locker);
		}
	}
	fprintf(stderr, "Done decreasing variable.\n");

	return NULL;
}


int main(int argc, char *argv[])
{
	int val, ret, ok;
	pthread_t t1, t2;

	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);
	}
	ret = pthread_join(t1, NULL);
	if (ret)
		perror_pthread(ret, "pthread_join");
	ret = pthread_join(t2, NULL);
	if (ret)
		perror_pthread(ret, "pthread_join");

	ok = (val == 0);
	printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
	return ok;
}
```

Ερωτήσεις:

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

Κάνουμε χρήση της εντολής time, ώστε να δούμε το χρόνο εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμός. Ο χρόνος εκτέλεσης του αρχικού προγράμματος, ο οποίος είναι ίδιος και στα δύο εκτελέσιμα που παράγονται, είναι:

```
oslabc23@os-node1:~/andr/3/oxi_etoima$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -5488099.

real    0m0.038s
user    0m0.072s
sys     0m0.000s
```

Έπειτα, ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό, με χρήση ατομικών λειτουργιών του GCC και με χρήση POSIX mutexes, αντίστοιχα, είναι:

```
oslabc23@os-node1:~/andr/3/etoima$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m0.413s
user    0m0.816s
sys     0m0.000s
```

```
oslabc23@os-node1:~/andr/3/etoima$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m3.894s
user    0m3.960s
sys     0m3.040s
```

Παρατηρούμε ότι με οποιονδήποτε τρόπο κι αν εκτελέσουμε συγχρονισμό ο χρόνος εκτέλεσης του προγράμματος είναι σημαντικά μεγαλύτερος. Αυτό ήταν αναμενόμενο, άλλωστε, αφού πριν εκτελέσει μια λειτουργία ένα νήμα, πρέπει πρώτα να περιμένει να έχει τελειώσει τη λειτουργία του το προηγούμενο, γεγονός που δικαιολογεί την αύξηση χρόνου.

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

Από τους χρόνους που παραθέσαμε παραπάνω, παρατηρούμε ότι η χρήση ατομικών λειτουργιών είναι πιο γρήγορη μέθοδος συγχρονισμού από τη χρήση POSIX mutexes. Ο λόγος που συμβαίνει αυτό, φαίνεται στην assembly που «αντιστοιχεί» στον κώδικα. Στον συγχρονισμό με ατομικές λειτουργίες, η αύξηση της μεταβλητής(και η μείωση), αντιστοιχεί σε μία μόνο εντολή assembly, ενώ στον συγχρονισμό με POSIX mutexes, αντιστοιχεί σε περισσότερες. Συνεπώς, για την εκτέλεση του συγχρονισμού στη δεύτερη περίπτωση, απαιτούνται περισσότεροι κύκλοι μηχανής, και άρα περισσότερος χρόνος, γεγονός που δικαιολογεί τον υψηλότερο χρόνο εκτέλεσης στη χρήση POSIX mutexes.

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.

Για την παραγωγή του ενδιαμέσου κώδικα assembly, στην περίπτωση του συγχρονισμού με ατομικές λειτουργίες, χρησιμοποιούμε την εξής εντολή μεταγλώττισης:

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

Έτσι, παράγεται ένα αρχείο με όνομα simplesync.s, το οποίο περίεχει τον κώδικα assembly που μας ενδιαφέρει. Οι εντολές, λοιπόν, που εξασφαλίζουν τον συγχρονισμό είναι:

```
lock addl       $1, (%rax)
```

```
lock subl       $1, (%rax)
```

Αυτές οι εντολές αντιστοιχούν στις εντολές **__sync_add_and_fetch(ip,1)** και **__sync_sub_and_fetch(ip,1)** της C, αντίστοιχα. Παρατηρούμε ότι πράγματι χρείαζεται μόνο μία εντολή assembly για κάθε εντολή συγχρονισμού, γεγονός που επαληθεύει την απάντηση στην προηγούμενη ερώτηση.

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.

Με αντίστοιχο τρόπο, προκύπτουν οι παρακάτω εντολές για το κλείδωμα και το ξεκλείδωμα του κρίσιμου τμήματος:

Pthread_mutex_lock():

```
movl    $locker, %edi
call    pthread_mutex_lock
```

Pthread_mutex_unlock():

```
movl    $locker, %edi
call    pthread_mutex_unlock
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Για την άσκηση αυτή μας δόθηκε κώδικας που σχεδιάζει την ακολουθία του συνόλου Mandelbrot. Εμείς κληθήκαμε να προσαρμόσουμε τον κώδικα κατάλληλα, ώστε να εκτελεί την ίδια λειτουργία με καταμερισμό της εργασίας σε Ν νήματα, όπου το Ν καθορίζεται από την είσοδο. Αναγκαίο ήταν επίσης να τα συγχρονίσουμε, ώστε να λάβουμε το επιθυμητό αποτέλεσμα.

Για τη δημιουργία των νημάτων, δημιουργήσαμε ένα πίνακα από νήματα, με μέγεθος όσο η είσοδος. Κάθε θέση του πίνακα περιείχε μία δομή με πληροφορίες του εκάστοτε νήματος. Επίσης, για να εξασφαλίσουμε την συγχρονισμό, χρησιμοποιήσαμε ένα πίνακα με Ν σημαφόρους. Αρχικοποιήσαμε κάθε τιμή στο 0, με εξαίρεση τον πρώτο, που τον αρχικοποιήσαμε στο 1, ώστε να ξεκινήσει αμέσως ο υπολογισμός. Κάθε νήμα που τυπώνει τη γραμμή που του ανατίθεται, αυξάνει τον σημαφόρο του επόμενου νήματος, με σκοπό να «μπει» και να τυπώσει την δική του γραμμή. Όταν φτάσουμε και στο τελευταίο νήμα, τότε αυτό τυπώνει τη γραμμή του και αυξάνει το σημαφόρο του πρώτου νήματος(θέση 0 του πίνακα). Αυτό εξασφαλίζεται με τη χρήση του %. Αυτή η διαδικασία επαναλαμβάνεται έως ότου τυπωθεί και η τελευταία γραμμή του συνόλου Mandelbrot.

Να σημειώσουμε ότι για να αποφύγουμε το segmentation fault σε περίπτωση που δε δοθεί είσοδος, ορίσαμε ως default είσοδο το 1. Αντί για αυτό, θα μπορούσαμε φυσικά να εκτυπώνουμε μήνυμα σφάλματος και να τερματίζουμε τη λειτουργία. Επίσης, δε συμπεριλάβαμε κάποιο έλεγχο για την περίπτωση που τα ορίσματα είναι περισσότερα από τα απαιτούμενα. Επομένως, σε αυτήν την περίπτωση εκτελείται κανονικά, με αριθμό νημάτων όσο το δεύτερο όρισμα(argv[1]).

Ο τροποποιημένος κώδικας mandel.c είναι ο ακόλουθος:

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

#define MANDEL_MAX_ITERATION 100000

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

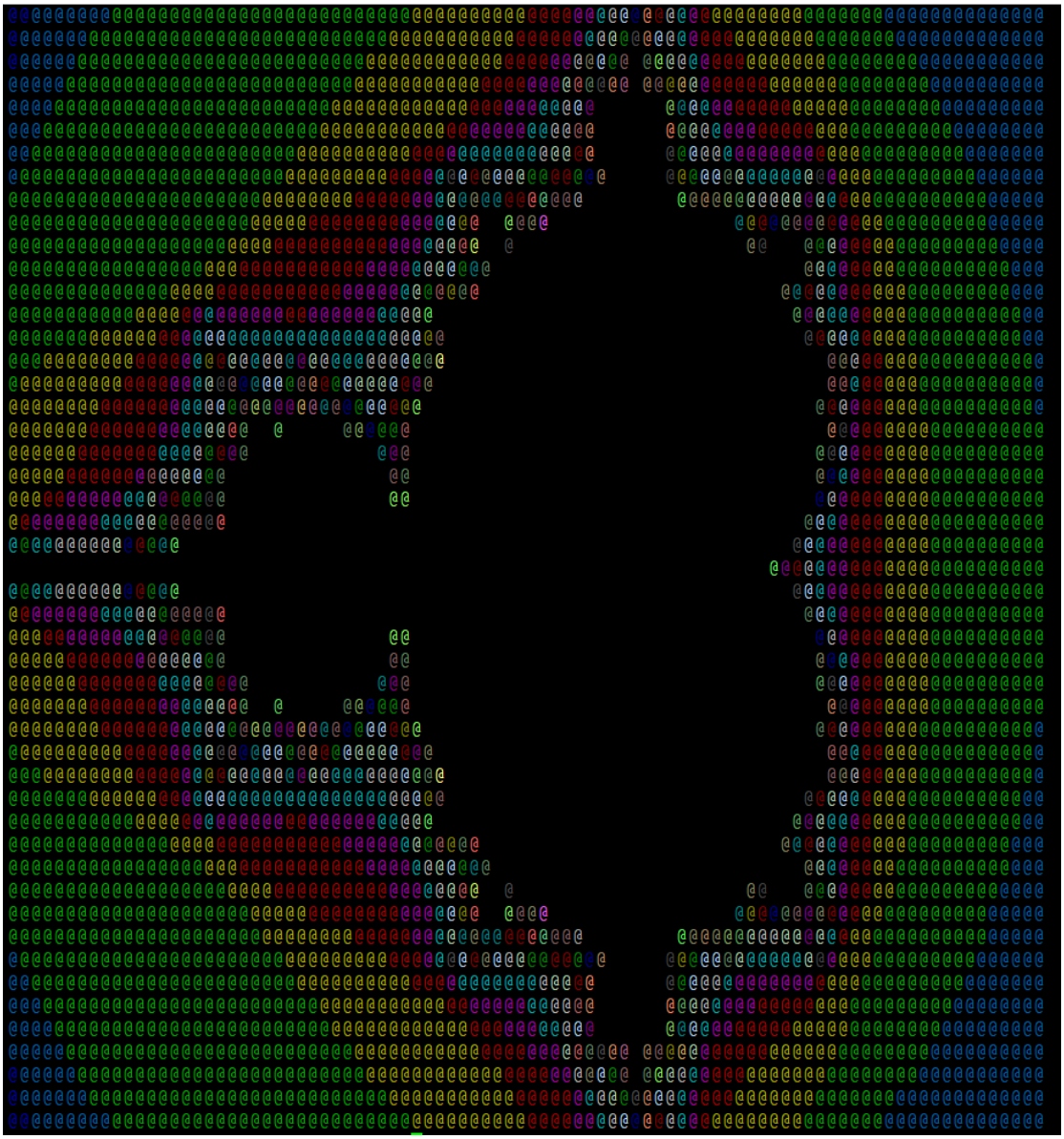
```
int y_chars = 50;
int x_chars = 90;

sem_t *semaphore;
/*
 * 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;

struct thread_info_struct {
	pthread_t tid;  /* POSIX thread id, as returned by the library */
	int tnumber;
	int numofthrd;
};
typedef struct thread_info_struct * thrptr;


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++) {

		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;
	}
}
```

```
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 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);
	output_mandel_line(fd, color_val);

}


void *thread_start_fn(void *arg)
{
	int line;
	int color_val[x_chars];
	/* We know arg points to an instance of thread_info_struct */
	thrptr thr = arg;

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

	for (line = thr->tnumber ; line < y_chars; line += thr->numofthrd ) {
```

```
        compute_mandel_line(line, color_val);
        sem_wait(&semaphore[line%thr->numofthrd]);
        output_mandel_line(1, color_val);
        sem_post(&semaphore[(thr->tnumber +1)%(thr->numofthrd)]);
    }

    return NULL;
}


int main(int argc, char *argv[])
{
    int line,  i;
    thrptr threads;
    int ret, NTHREADS;
    xstep = (xmax - xmin) / x_chars;
    ystep = (ymax - ymin) / y_chars;

    if (argc == 1) {
        NTHREADS = 1;
    }
    else{
        NTHREADS = atoi(argv[1]);
    }
    threads = malloc(NTHREADS * sizeof(*threads));

    semaphore = malloc(NTHREADS * sizeof(*semaphore));

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


    for(line=0; line < NTHREADS ; line++){
        threads[line].tnumber = line;
        threads[line].numofthrd = NTHREADS;

        ret = pthread_create(&threads[line].tid, NULL, thread_start_fn, &threads[line]);
        if (ret) {
            perror_pthread(ret, "pthread_create");
            exit(1);
        }

    }


    for (i = 0; i < NTHREADS; i++) {
        ret = pthread_join(threads[i].tid, NULL);
```

```
            if (ret) {
                perror_pthread(ret, "pthread_join");
                exit(1);
            }
        }

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

Παραθέτουμε και ένα παράδειγμα εκτέλεσης με είσοδο NTHREADS = 10(, το οποίο δυστυχώς δεν φαίνεται ολόκληρο στην φωτογραφία που παραθέτουμε).



Ερωτήσεις:

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

Για το σχήμα συγχρονισμού χρειάζονται συνολικά NTHREADS σημαφόροι, δηλαδή όσοι και τα νήματα. Όπως αναφέραμε ήδη, σε κάθε νήμα αντιστοιχεί ένας σημαφόρος. Έτσι, κάθε νήμα που εκτυπώνει τη γραμμή του, στέλνει σήμα(sem_post) και αυξάνει τον σημαφόρο του επόμενου, με σκοπό να εκτελέσει και αυτό τη λειτουργία του, ενώ κάνει sem_wait(αναστολή της λειτουργίας του), με αποτέλεσμα να περιμένει σήμα από τον προηγούμενο για να συνεχίσει τη λειτουργία του κοκ. Σε περίπτωση που φτάσουμε στον τελευταίο σημαφόρο, αυτός στέλνει μήνυμα στον πρώτο, με αποτέλεσμα να δημιουργείται μία κυκλική κίνηση η οποία και συνεχίζεται έως ότου ολοκληρωθεί το ζητούμενο.

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.

Επειδή δεν είχαμε διαθέσιμο μηχάνημα με δύο υπολογιστικούς πυρήνες, παραθέτουμε την έξοδο που λάβαμε από ένα τετραπήρυνο.

Σειριακός υπολογισμός:

```
real    0m1.021s
user    0m0.980s
sys     0m0.016s
```

Παράλληλος υπολογισμός:

```
real    0m0.520s
user    0m0.988s
sys     0m0.016s
```

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

Παρατηρούμε από τους χρόνους που παρατέθηκαν, ότι το παράλληλο πρόγραμμα που συγγράψαμε εκτελείται πιο γρήγορα από το σειριακό. Θεωρούμε, ότι και σε σύστημα με 2 πυρήνες θα παίρναμε ένα ανάλογο αποτέλεσμα. Ο λόγος που γίνεται αυτό είναι ότι στην περίπτωση του σειριακού προγράμματος, όλη η «δουλεία» ανατίθεται σε έναν μόνο πόρο, ενώ στην περίπτωση του παράλληλου, διαμοιράζεται σε 2 πόρους. Επιπλέον, το κρίσιμο τμήμα στον κώδικα αποτελείται αποκλειστικά από την εκτύπωση της γραμμής, ενώ οι υπολογισμοί γίνονται ξεχωριστά. Αυτό έχει ως αποτέλεσμα το κρίσιμο τμήμα να απαιτεί μικρό χρονικό διάστημα για την εκτέλεση του, και έτσι τα υπόλοιπα νήματα δεν μένουν αδρανή για μεγάλο χρονικό διάστημα. Αντίθετα, στην περίπτωση του σειριακού, για να εκτυπωθεί η επόμενη γραμμή, πρέπει πρώτα να τελειώσει και ο υπολογισμός και η εκτύπωση.

Τέλος, αναφέρουμε ότι αν είχαμε συμπεριλάβει στο κρίσιμο τμήμα και τους υπολογισμούς, τότε το αποτέλεσμα που θα λαμβάναμε από την εκτέλεση της time(1) στην περίπτωση του παράλληλου προγράμματος, θα ήταν πιο κοντά σε αυτά του σειριακού προγράμματος.

Παρατήρηση: Επίσης, παρατηρήσαμε ότι όσο αυξάναμε τον αριθμό των νημάτων που δίναμε ως είσοδο, τόσο πιο γρήγορα εκτελούταν το παράλληλο πρόγραμμα.

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

Παρατηρούμε ότι αν πατηθεί Ctrl-C για τερματισμό του προγράμματος, πριν τερματιστεί η λειτουργία του, το χρώμα των γραμμάτων του τερματικού παραμένουν στο χρώμα του τελικού χαρακτήρα που τυπώθηκε για το Mandelbrot. Για να διορθωθεί αυτό, πρέπει να συμπεριλάβουμε στον κώδικα μας μία ρουτίνα sighandler, η οποία θα διαχειρίζεται το σήμα που στέλνεται. Πιο αναλυτικά, το πάτημα του Ctrl-C στέλνει σήμα SIGINT. Επομένως, καλούμαστε να το χειριστούμε κατάλληλα, ώστε πριν τερματιστεί το πρόγραμμα, να καλείται η εντολή **reset_xterm_color(1)** με αποτέλεσμα να επανέλθει το τερματικό στην αρχική του κατάσταση.