ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ
http://www.cslab.ece.ntua.gr

Πέτρου Νικόλας	oslabb13	03117714
Πεγειώτη Νάταλυ		03117707

Λειτουργικά Συστήματα

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2019-2020

Άσκηση 3
Συγχρονισμός

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

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;

Χρησιμοποιώντας την εντολή time(1) παίρνουμε τα εξής αποτελέσματα για το χρόνο εκτέλεσης των εκτελέσιμων χωρίς συγχρονισμό (α), και συγχρονισμενα (β) simplesync-mutex και (γ) simplesync-atomic:

```
oslabb13@os-node1:~/lab3$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 3224163.

real    0m0.039s
user    0m0.072s
sys     0m0.000s
```

(α)

```
oslabb13@os-node1:~/lab3$ time ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m3.731s
user    0m3.856s
sys     0m2.824s
```

(β)

```
oslabb13@os-node1:~/lab3$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m0.103s
user    0m0.200s
sys     0m0.000s
```

(γ)

Παρατηρούμε ότι το εκτελέσιμο πριν τον συγχρονισμό είναι και το πιο γρήγορο, γεγονός που είναι λογικό. Δεν χρειάζεται να ελέγχει αν το νήμα θα εκτελέσει κάποιο κρίσιμο σημείο στον κώδικα έτσι ώστε να υπάρχει καθυστέρηση επειδή τα άλλα νήματα περιμένουν να εκτελέσουν και αυτά το κρίσιμο σημείο του κώδικα. Οι εντολές χρονοδρομολογούνται τυχαία στον επεξεργαστή και εκτελούνται με τυχαία σειρά. Αυτό όμως προκαλεί λανθασμένο αποτέλεσμα στο τέλος της εκτέλεσης του προγράμματος.

2. Ποια μέθοδος συγχρονισμού είναι γρηγορότερη, η χρήση ατομικών λειτουργιών ή η χρήση POSIX mutexes; Γιατί;
Όπως φαίνεται από τους χρόνους που υπολογίσαμε στο πρώτο ερώτημα, είναι πιο γρήγορη η χρήση ατομικών λειτουργιών. Με mutexes χρησιμοποιούμε locks και unlocks για να κλειδώσουμε το κρίσιμο σημείο του κώδικα, ώστε να ανασταλεί η λειτουργία του νήματος προσωρινά, μέχρι να ξεκλειδώσει ξανά το κρίσιμο σημείο. Αντίθετα, χρησιμοποιώντας ατομικές λειτουργίες το νήμα δεν περιμένει για να μπει στο κρίσιμο σημείο του κώδικα.

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 .
Ο κώδικα assembly για τη χρήση των ατομικών λειτουργιών παράγεται με την εντολή
gcc -S -g -DSYNC_ATOMIC simplesync.c
Τα κομμάτια του κώδικα στα οποία μεταφράζονται οι εντολές του simplesync.c
__sync_add_and_fetch(ip,1); και ***__sync_sub_and_fetch(ip,1);*** είναι τα εξής:

```
.L3:
        .loc 1 49 0
        lock addq       $1, -16(%rbp)
        .loc 1 45 0
        addl    $1, -4(%rbp)
```

__sync_add_and_fetch(ip,1);

```
.L7:
        .loc 1 76 0
        lock subq       $1, -16(%rbp)
        .loc 1 72 0
        addl    $1, -4(%rbp)
```

__sync_sub_and_fetch(ip,1);

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.
Ο κώδικα assembly για τη χρήση POSIX mutexes παράγεται με την εντολή
gcc -S -g -DSYNC_MUTEX simplesync.c
Τα κομμάτια του κώδικα στα οποία μεταφράζονται οι εντολές του simplesync.c ***pthread_mutex_lock();*** και ***pthread_mutex_unlock();*** είναι τα εξής:

```
.L3:
        .loc 1 53 0
        movl    $lock, %edi
        call    pthread_mutex_lock
        .loc 1 56 0
        movq    -16(%rbp), %rax
        movl    (%rax), %edx
        addl    $1, %edx
        movl    %edx, (%rax)
        .loc 1 58 0
        movl    $lock, %edi
        call    pthread_mutex_unlock
        .loc 1 45 0
        addl    $1, -4(%rbp)
```

```
.L7:
        .loc 1 80 0
        movl    $lock, %edi
        call    pthread_mutex_lock
        .loc 1 83 0
        movq    -16(%rbp), %rax
        movl    (%rax), %edx
        subl    $1, %edx
        movl    %edx, (%rax)
        .loc 1 85 0
        movl    $lock, %edi
        call    pthread_mutex_unlock
        .loc 1 72 0
        addl    $1, -4(%rbp)
```

Παρατηρούμε ότι οι εντολές σε assembly που χρειάζονται με τη χρήση mutexes είναι πολύ περισσότερες από αυτές που χρειάζονται για ατομικές λειτουργίες. Σε αυτό οφείλεται και η διαφορά στην ταχύτητα εκτέλεσης των δύο λειτουργιών.

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Ερωτήσεις

1. Πόσοι σημαφόροι χρειάζονται για το σχήμα συγχρονισμού που υλοποιείτε;
Χρειαζόμαστε ένα σημαφόρο για κάθε νήμα που χρησιμοποιείται. Άρα για N νήματα, χρειάζονται N σημαφόροι.

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.
Παρατίθενται οι χρόνοι που απαιτούνται για την εκτέλεση των δύο προγραμμάτων με δύο νήματα.

Σειριακό Πρόγραμμα mandel:

```
real    0m1.022s
user    0m0.980s
sys     0m0.016s
```

Παράλληλο Πρόγραμμα mandel_threads:

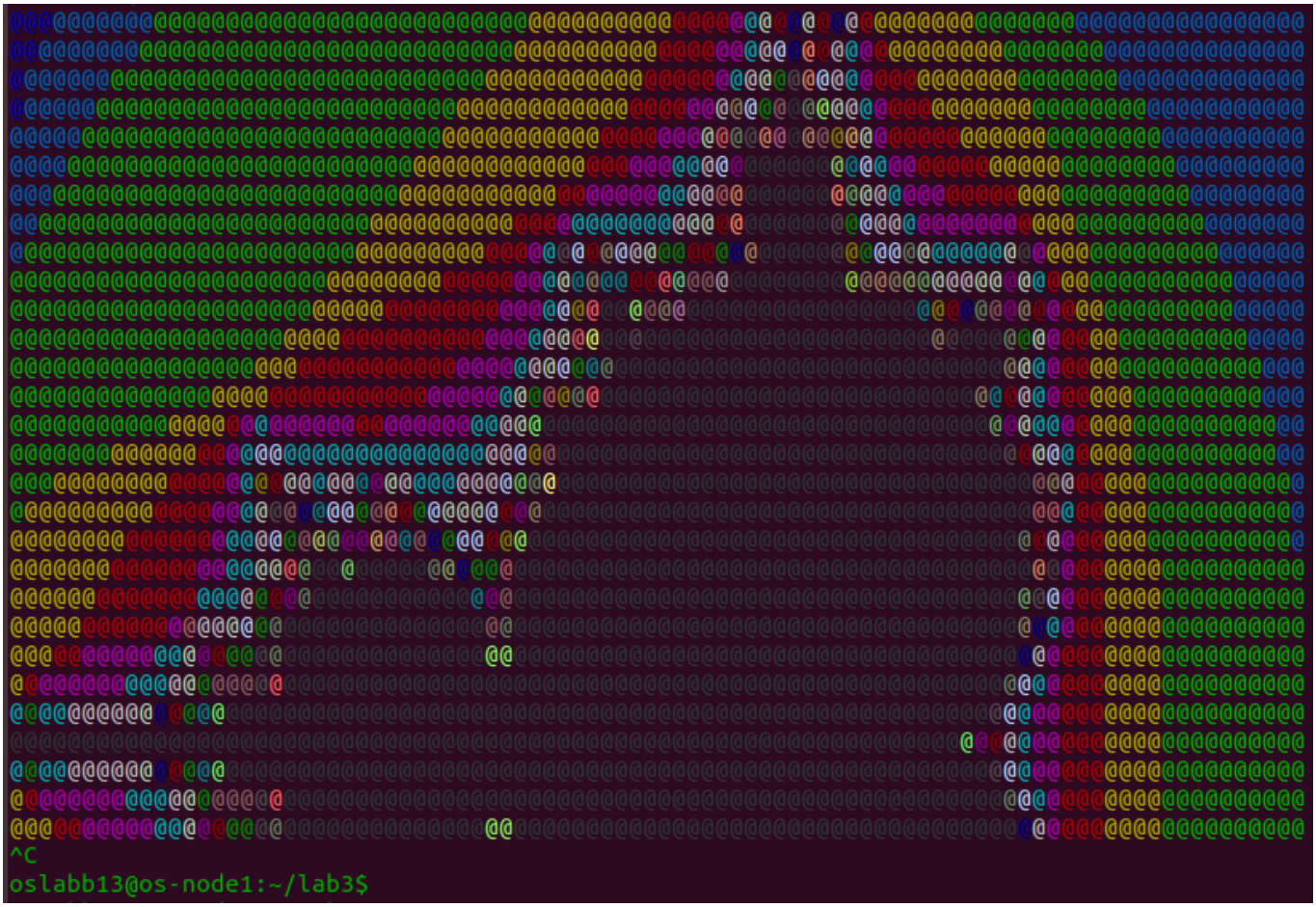
```
real    0m0.520s
user    0m0.972s
sys     0m0.028s
```

Εκτελώντας την εντολή cat /proc/cpuinfo βλέπουμε ότι το σύστημα έχει 8 πυρήνες. Αυτό το καταλαβαίνουμε και από το ότι το user time στο παράλληλο πρόγραμμα είναι μεγαλύτερο από το real, κάτι που συνήθως δείχνει ότι το πρόγραμμα χρησιμοποιεί πολλαπλούς πυρήνες.

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;
Το παράλληλο πρόγραμμα που φτιάξαμε, παρουσιάζει όντως επιτάχυνση, αφού έχουμε πολλούς πυρήνες και έτσι μπορούν να γίνουν ταυτόχρονα οι λειτουργίες των νημάτων έξω από το κρίσιμο τμήμα. Κάνοντας δοκιμές στο πρόγραμμα, καταλαβαίνουμε ότι το κρίσιμο τμήμα δεν χρειάζεται να συμπεριλαμβάνει και το κομμάτι του υπολογισμού, αφού ο κάθε υπολογισμός είναι ανεξάρτητος.
Το μόνο πράγμα που χρειάζεται να συγχρονίσουμε είναι το output, το οποίο πρέπει να εμφανίζεται με συγκεκριμένη σειρά. Οι υπολογισμοί μπορούν να γίνονται παράλληλα, οπότε βγάζοντας τους έξω από το κρίσιμο τμήμα, έχουμε μεγάλη βελτίωση της ταχύτητας.

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;

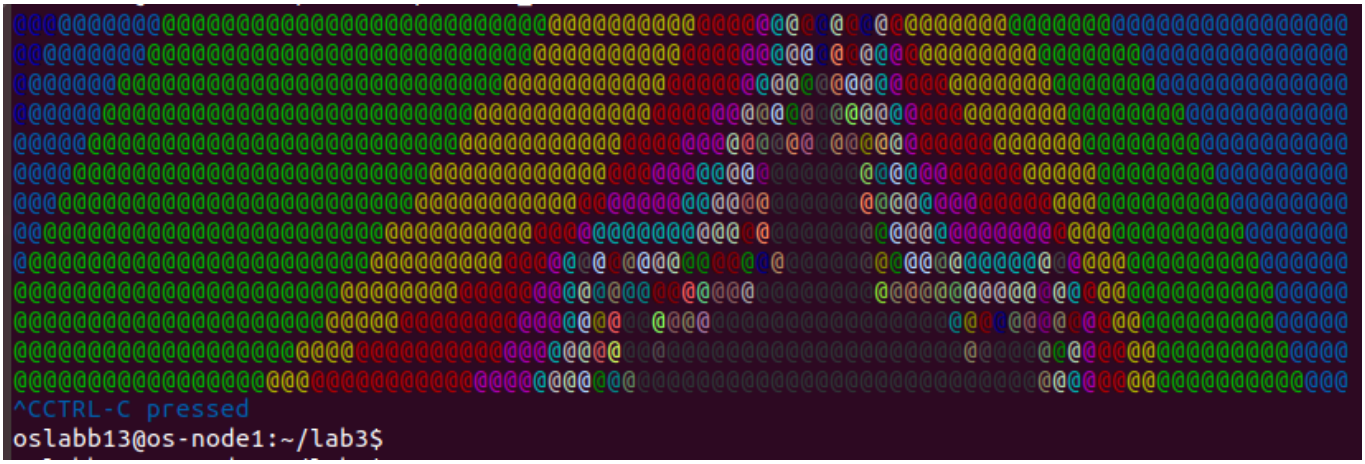
Πατώντας Ctrl-C, το πρόγραμμα τερματίζει στο σημείο εκτέλεσης στο οποίο βρίσκεται εκείνη την στιγμή. Αυτό έχει ως αποτέλεσμα, τα γράμματα στο τερματικό να παίρνουν το χρώμα της τελευταίας εκτύπωσης, αφού το πρόγραμμα δεν φτάνει μέχρι το τέλος για να εκτελέσει την εντολή "reset_xterm_color(1)".



Για να εξασφαλιστεί ότι το τερματικό θα επαναφέρεται στην προηγούμενη του κατάσταση όταν σταλεί το σήμα τερματισμού (πατώντας Ctrl-C), μπορούμε να δημιουργήσουμε έναν signal handler ο οποίος όταν λάβει SIGINT καλεί την "reset_xterm_color(1)".

```
void intHandler(int sig)
{
        printf("CTRL-C pressed\n");
        reset_xterm_color(1);
        exit(0);
}
```

Πλέον, σε περίπτωση που ενεργοποιηθεί το σήμα τερματισμού Ctrl+C, το πρόγραμμα τερματίζει στο σημείο εκτέλεσης στο οποίο βρίσκεται τη δεδομένη στιγμή αλλά το τερματικό επαναφέρεται στην προηγούμενη κατάστασή του.



Παράρτημα Κώδικα

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

```
#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

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

```
	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_sub_and_fetch(&ip,1);
			//--(*ip);
			/* ... */
		} else {
			pthread_mutex_lock(&lock);
			/* ... */
			/* You cannot modify the following line */
			--(*ip);
			/* ... */
			pthread_mutex_unlock(&lock);
		}
	}
	fprintf(stderr, "Done decreasing variable.\n");
	return NULL;
}


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

	if (!USE_ATOMIC_OPS){
		if (pthread_mutex_init(&lock,NULL) !=0)
		{
			printf("\n mutex init failed\n");
		}
	}
	/*
	 * 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;
}
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <fcntl.h>
#include <semaphore.h>
#include <signal.h>
#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000
#include <errno.h>

#define perror_pthread(ret, msg) \
        do { errno = ret; perror(msg);} while (0)
/***************************
 * Compile-time parameters *
 ***************************/
sem_t *semaphore;
/*
 * 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;


struct thread_info_struct{
        pthread_t tid;
        int thrid;
        int thrcnt;
};

/*
 * 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 *compute_and_output_mandel_line(void *arg)
```

```
{
	int line;
	struct thread_info_struct *thr = arg;
	int start = thr->thrid;
	int step = thr->thrcnt;

	int color_val[x_chars];
	for (line=start; line<y_chars; line=line+step)
	{
		compute_mandel_line(line, color_val);
		sem_wait(&semaphore[start]);
		output_mandel_line(1, color_val);
		sem_post(&semaphore[(start+1) % step]);
	}
	return NULL;
}


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

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


void intHandler(int sig)
{
	printf("CTRL-C pressed\n");
	reset_xterm_color(1);
	exit(0);
}


int main(int argc, char *argv[])
{
	int thrcnt;
	struct thread_info_struct *thr;
	signal(SIGINT, intHandler);

	xstep = (xmax-xmin) / x_chars;
	ystep = (ymax-ymin) / y_chars;
	if (argc != 2)
	{
		fprintf(stderr, "Exactly one argument required\n""thread_count:The number of threads to
create.\n");
```

```
		exit(1);
	}
	if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt<=0)
	{
		fprintf(stderr, "%s is not valid for 'thread_count'\n", argv[1]);
		exit(1);
	}

	int i, ret;
	thr = malloc(thrcnt * sizeof(*thr));
	semaphore = malloc(thrcnt * sizeof(sem_t));
	sem_init(&semaphore[0],0,1);
	for (i=1; i<thrcnt; i++)
	{
		sem_init(&semaphore[i],0,0);
	}
	for (i=0; i<thrcnt; i++)
	{
		thr[i].thrid = i;
		thr[i].thrcnt = thrcnt;
		ret = pthread_create(&thr[i].tid, NULL, compute_and_output_mandel_line, &thr[i]);
		if (ret){
			perror_pthread(ret, "pthread_create");
			exit(1);
		}
	}

	/*
	 * draw the Mandelbrot Set, one line at a time.
	 * Output is sent to file descriptor '1', i.e., standard output.
	 */
	for (i = 0; i < thrcnt; i++) {
		ret = pthread_join(thr[i].tid, NULL);
		if (ret){
			perror_pthread(ret, "pthread_join");
			exit(1);
		}
	}


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