



ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ 3^Η

Εργαστηριακή Ομάδα: oslabb25

| Ιωάννης Μιχαήλ Καζελίδης- 03117885
| Μάριος Κερασιώτης- 03117890

1 ΑΣΚΗΣΕΙΣ

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Αρχικά μέσω του *makefile* μεταγλωττίζουμε και εκτελούμε το *simplesync.c*. Παρατηρούμε πως παράχθηκαν δύο εκτελέσιμα αρχεία, τα *simplesync-atomic* και *simplesync-mutex*. Το ποιο από τα 2 αρχεία θα παραχθεί ορίζεται στον κώδικας μας από τις εντολές:

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

Ανάλογα με την τιμή του *USE_ATOMIC_OPS* επιλέγεται και η κατάλληλη if που θα εκτελέσει το πρόγραμμά μας. Τώρα, το ποιο εκτελέσιμο θα παραχθεί κρίνεται με βάση τις παρακάτω εντολές που βρίσκονται στο *makefile*:

```
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
```

Έτσι παράγονται τα 2 διαφορετικά εκτελέσιμα.

Εκτελώντας τα έχουμε έξοδο:

```
> ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -29736.
> ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = -1412532.
```

Στη συνέχεια επεκτείνουμε τον κώδικα στο αρχείο *simplesync.c* ως έχει παρακάτω:

```
/*
 * 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

/* 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, ret;
    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 */
            // ++(*ip);
            __sync_add_and_fetch(ip, 1);
            /* ... */
        }
        else
        {
            /* ... */

            // lock mutex
            ret = pthread_mutex_lock(&mutex);
            if (ret)
            {
                perror_pthread(ret, "pthread_mutex_lock");
            }

            /* You cannot modify the following line */
            ++(*ip);

            /* ... */

```

```

        //unlock mutex
        ret = pthread_mutex_unlock(&mutex);
        if (ret)
        {
            perror_pthread(ret, "pthread_mutex_unlock");
        }
    }
}
fprintf(stderr, "Done increasing variable.\n");

return NULL;
}

void *decrease_fn(void *arg)
{
    int i, ret;
    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);
            /* ... */
        }
        else
        {
            /* ... */
            // lock mutex
            ret = pthread_mutex_lock(&mutex);
            if (ret)
            {
                perror_pthread(ret, "pthread_mutex_lock");
            }
            /* You cannot modify the following line */
            --(*ip);

            /* ... */
            //unlock mutex
            ret = pthread_mutex_unlock(&mutex);
            if (ret)
            {
                perror_pthread(ret, "pthread_mutex_unlock");
            }
        }
    }
    fprintf(stderr, "Done decreasing variable.\n");

    return NULL;
}

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_thread(ret, "pthread_create");
    exit(1);
}
ret = pthread_create(&t2, NULL, decrease_fn, &val);
if (ret)
{
    perror_thread(ret, "pthread_create");
    exit(1);
}

/*
 * Wait for threads to terminate
 */
ret = pthread_join(t1, NULL);
if (ret)
    perror_thread(ret, "pthread_join");
ret = pthread_join(t2, NULL);
if (ret)
    perror_thread(ret, "pthread_join");

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

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

return ok;
}

```

* simplesync.c

Μεταγλωττίζοντας το πρόγραμμα και εκτελώντας τα δύο παραγόμενα εκτελέσιμα έχουμε έξοδο:

```

> ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
> ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

```

1.1.1 Ερώτηση 1^η

Χρησιμοποιώντας την εντολή `time` στα 2 εκτελέσιμα του αρχικού προγράμματος έχουμε έξοδο:

```
> time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 2147339.
```

```
CPU      193%
user      0.078
system    0.000
total     0.040
max memory      5 KB
```

```
> time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 137219.
```

```
CPU      190%
user      0.097
system    0.000
total     0.051
max memory      5 KB_
```

Ενώ χρησιμοποιώντας την εντολή `time` στα 2 εκτελέσιμα του «επεκταμένου» προγράμματος έχουμε έξοδο:

```
> time ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

```
CPU      199%
user      0.661
system    0.000
total     0.331
max memory      5 KB
```

```
> time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

```
CPU      197%
user      2.140
system    0.976
total     1.578
max memory      5 KB
```

Με βάση τους παραπάνω χρόνους, παρατηρούμε ότι με το να κάνουμε συγχρονισμό η εκτέλεση του προγράμματος γίνεται πιο αργή. Αυτό είναι λογικό, καθώς είτε χρησιμοποιούμε POSIX mutexes είτε gcc atomic operations, το κάθε νήμα αναμένει το άλλο να ολοκληρώσει μια διεργασία και αυτό έχει ως συνέπεια ο χρόνος εκτέλεσης να αυξάνεται σε σχέση με την περίπτωση που δεν έχουμε συγχρονισμό.

1.1.2 Ερώτηση 2^η

Η μέθοδος με χρήση ατομικών λειτουργιών είναι αισθητά γρηγορότερη και μπορεί να βγει το συμπέρασμα χωρίς κιόλας να γίνει χρήση της εντολής `time(1)`. Με την χρήση της εντολής γίνεται αντιληπτό και επιβεβαιώνεται πως η μέθοδος με χρήση ατομικών λειτουργιών του GCC είναι γρηγορότερη σε σχέση με αυτή των POSIX mutexes. Αυτό οφείλεται στην assembly που παράγεται και

αντιστοιχεί στο κάθε εκτελέσιμο. Στην περίπτωση των atomic operations, η εντολή αύξησης (ή μείωσης αντίστοιχα) μεταφράζεται σε μια μόνο εντολή assembly, ενώ η υλοποίηση με mutexes αποτελεί μια πιο high-level προσέγγιση, ο κώδικας του συγχρονισμού μεταφράζεται σε μια σειρά εντολών κώδικα assembly κάνοντας έτσι την εκτέλεση του προγράμματος πιο αργή. Έτσι, λοιπόν, αιτιολογείται η διαφορά ανάμεσα στην ταχύτητα των 2 μεθόδων.

1.1.3 Ερώτηση 3^η

Όπως είπαμε και παραπάνω, οι ατομικές εντολές μεταφράζονται σε μια εντολή assembly. Έτσι έχουμε:

Η εντολή `__sync_add_and_fetch(ip, 1)` μεταφράζεται στην `lock addl $1, (%rbx)`.

Η εντολή `__sync_sub_and_fetch(ip, 1)` μεταφράζεται στην `lock subl $1, (%rbx)`.

Η έξοδος της εντολής make για τον τρόπο μεταγλώττισης του *simplesync.c*

```
> 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 -S -g -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c
gcc -Wall -O2 -pthread -S -g -o simplesync-atomic simplesync-atomic.o
gcc: warning: simplesync-atomic.o: linker input file unused because linking not done
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
```

1.1.4 Ερώτηση 4^η

Όπως και στο προηγούμενο ερώτημα, ακολουθούμε την ίδια διαδικασία και παρατηρούμε πως στην περίπτωση χρήσης των POSIX mutexes οι εντολές assembly που παράγονται είναι οι ακόλουθες:

• Για το κλείδωμα:

```
movq %r13, %rdi
call pthread_mutex_lock@PLT
```

• Για το ξεκλείδωμα:

```
movq %r13, %rdi
call pthread_mutex_unlock@PLT
```

Πλέον γίνεται έμπρακτα φανερό πως στην περίπτωση των POSIX mutexes θέλουμε παραπάνω εντολές άρα και παραπάνω κύκλους μηχανής σε σχέση με την περίπτωση των ατομικών εντολών.

Η έξοδος της εντολής make για τον τρόπο μεταγλώττισης του *simplesync.c* τώρα γίνεται:

```
> 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 -S -g -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c
gcc -Wall -O2 -pthread -S -g -o simplesync-mutex simplesync-mutex.o
gcc: warning: simplesync-mutex.o: linker input file unused because linking not done
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
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Για αυτό το ερώτημα χρησιμοποιούμε τον παρακάτω πηγαίο κώδικα:

```
/*
 * 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 <errno.h>

#include <semaphore.h>
#include <pthread.h>
#include <signal.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

int NTHREADS = 1;

// .....
#define perror_thread(ret, msg) \
do \
{ \
    errno = ret; \
    perror(msg); \
} while (0)

typedef struct
{
    pthread_t tid; /* POSIX thread id, as returned by the library */
    int line;
    sem_t sem;
} thread_info_struct;

thread_info_struct *threads;

// .....

/*****
 * 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;

```



```

/*
 * 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 usage(char *argv0)

```

```

{
    fprintf(stderr, "Usage: %s NTHREADS\n\n"
                  "Exactly one argument required:\n"
                  "    NTHREADS: The number of threads.\n",
            argv0);
    exit(1);
}

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 *safe_malloc(size_t size)
{
    void *p;

    if ((p = malloc(size)) == NULL)
    {
        fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
            size);
        exit(1);
    }

    return p;
}
// .....

/*
 * This function provides user the safety for resetting all character
 * attributes before leaving, to ensure the prompt is
 * not drawn in a fancy color.
 */

void unexpectedSignal(int sign)
{
    signal(sign, SIG_IGN);
    reset_xterm_color(1);
    exit(1);
}

void *compute_and_output_mandel_line(void *argument)
{
    int line = *(int *)argument;
    int i;

    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];

    for (i = line; i < y_chars; i += NTHREADS)
    {
        //compute mandel line
    }
}

```

```

        compute_mandel_line(i, color_val);

        //each thread should wait and so it uses sem_wait func
        //with argument the semaphore referring to this particular thread
        sem_wait(&threads[i % NTHREADS].sem);

        //output mandel line, STDOUT_FILENO for standard output
        output_mandel_line(STDOUT_FILENO, color_val);

        // send signal to the next thread
        sem_post(&threads[((i % NTHREADS) + 1) % NTHREADS].sem);
    }

    return NULL;
}

int main(int argc, char *argv[])
{
    // .....
    if (argc != 2)
        usage(argv[0]);
    if (safe_atoi(argv[1], &NTHREADS) < 0 || NTHREADS <= 0)
    {
        fprintf(stderr, "'%s' is not valid for `NTHREADS'\n", argv[1]);
        exit(1);
    }

    signal(SIGINT, unexpectedSignal);

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

    int line;
    // .....

    /* Create threads */
    threads = safe_malloc(NTHREADS * sizeof(*threads));

    // set the first semaphore as not asleep
    if ((sem_init(&threads[0].sem, 0, 1)) == -1)
    {
        perror("initialization of semaphores");
        exit(1);
    }

    // set the rest of the semaphores to wait
    for (line = 1; line < NTHREADS; ++line)
    {
        if ((sem_init(&threads[line].sem, 0, 0)) == -1)
        {
            perror("initialization of semaphores");
            exit(1);
        }
    }

    /*
     * Create NTHREADS
     */
    for (line = 0; line < NTHREADS; ++line)
    {
        threads[line].line = line;
        if ((
            pthread_create(&threads[line].tid, NULL, compute_and_output_mandel_line, &thr
eads[line].line)) != 0)

```

```

    {
        perror("creation of threads");
        exit(1);
    }
}

/*
 * Wait for all threads to terminate
 */
for (line = 0; line < NTHREADS; ++line)
{
    int ret = pthread_join(threads[line].tid, NULL);
    if (ret)
    {
        perror_pthread(ret, "pthread_join");
        exit(1);
    }
}

// destroy all semaphores
for (line = 0; line < NTHREADS; ++line)
{
    sem_destroy(&threads[line].sem);
}

// free allocated space
free(threads);

reset_xterm_color(1);
return 0;
}

```

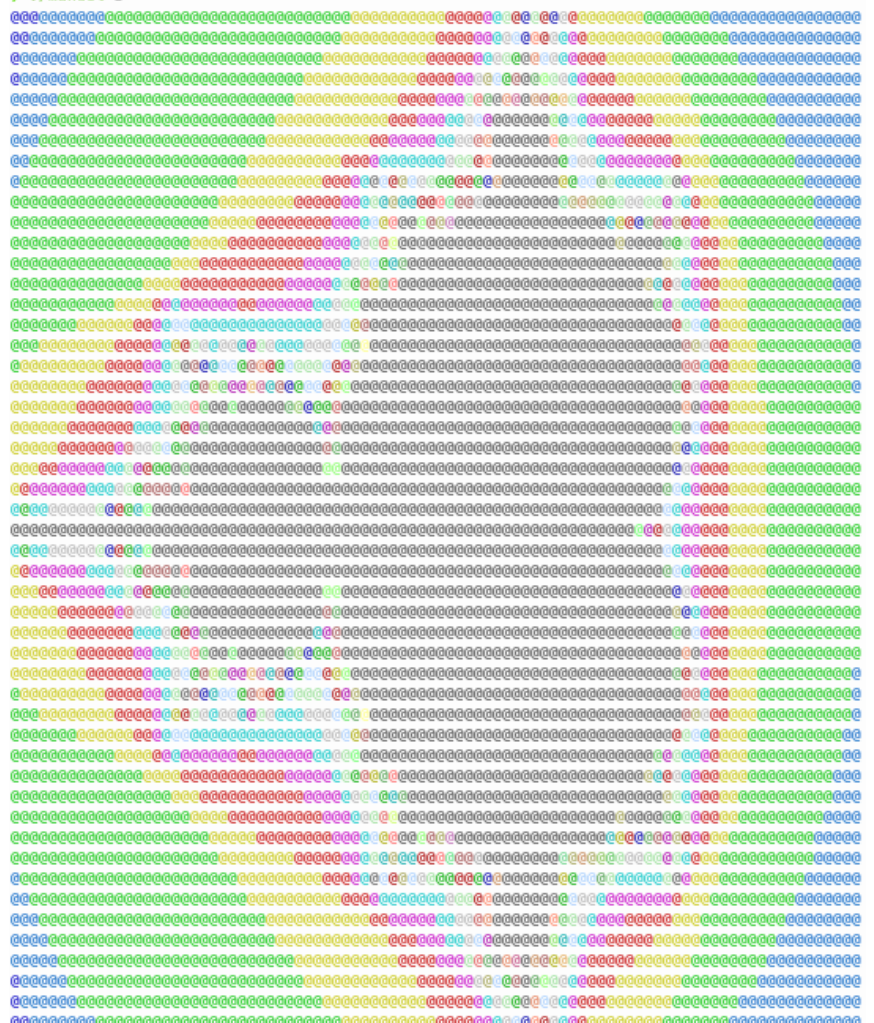
mandel1.c

Εκτελώντας το πρόγραμμα με όρισμα στη γραμμή εντολών την τιμή $NTHREADS = 3$ (προτεινόμενη τιμή) έχουμε έξοδο (σε άσπρο background του terminal):

```

> ./mandel 3

```





1.2.1 Ερώτηση 1^η

Για την υλοποίηση χρησιμοποιούνται N σημαφόροι, δηλαδή ένας σημαφόρος για το κάθε thread. Αρχικά, θέτουμε όλους τους σημαφόρους εκτός του 1^{ου} στο 0. Τον 1^ο τον αρχικοποιούμε σε 1 για να ξεκινήσει αμέσως ο υπολογισμός της πρώτης γραμμής. Όλα τα νήματα αρχίζουν να υπολογίζουν το καθένα την γραμμή που του αντιστοιχεί. Το πρώτο νήμα τυπώνει την 1^η γραμμή και στέλνει σήμα sem_post στο επόμενο για να αυξήσει τον σημαφόρο του, να ξυπνήσει και να τυπώσει την δική του γραμμή, την στιγμή που το πρώτο νήμα υπολογίζει πλέον την επόμενη γραμμή που ενδεχομένως του αντιστοιχεί κάνοντας sem_wait και περιμένοντας sem_post από κάποιο άλλο νήμα για να συνεχίσει να τυπώνει τις γραμμές του. Στην συνέχεια, το 2^ο σήμα ξυπνάει το επόμενο με την ίδια διαδικασία και έτσι τυπώνονται όλες οι γραμμές του σχήματος, ενώ οι υπολογισμοί γίνονται ταυτόχρονα οπότε έτσι γλυτώνουμε χρόνο σε αντίθεση με το σειριακό πρόγραμμα. Αξιοσημείωτο είναι το γεγονός πως αν ο αριθμός των νημάτων είναι μικρότερος από τον αριθμό των γραμμών, τότε η αφύπνιση των νημάτων λειτουργεί σαν κυκλικός δακτύλιος, με το κάθε νήμα να παίρνει την άδεια προς την εκτέλεση από το προηγούμενό του.

1.2.2 Ερώτηση 2^η

Ο υπολογιστής στον οποίο τρέχουμε το πρόγραμμα διαθέτει 4 υπολογιστικούς πυρήνες. Εκτελούμε δύο φορές το πρόγραμμα, μια φορά το αρχικό που ήταν σειριακό και μια φορά το παράλληλο πρόγραμμα με 2 νήματα υπολογισμού και τα χρονομετρούμε:

	
CPU 99%	CPU 197%
user 0.512	user 0.541
system 0.028	system 0.024
total 0.540	total 0.286
max memory 5 KB	max memory 5 KB

Αριστερά φαίνονται τα αποτελέσματα του σειριακού και δεξιά τα αποτελέσματα του παράλληλου προγράμματος. Παρατηρούμε πως ο χρόνος εκτέλεσης του σειριακού προγράμματος είναι μεγαλύτερος (σχεδόν διπλάσιος) του χρόνου του παράλληλου με δυο νήματα.

1.2.3 Ερώτηση 3^η

Με βάση τα παραπάνω αποτελέσματα παρατηρούμε ότι όντως υπάρχει επιτάχυνση στο παράλληλο πρόγραμμα που φτιάξαμε, κάτι το οποίο ήταν αναμενόμενο. Αυτό συμβαίνει διότι οι υπολογισμοί (εργασίες) που γίνονται από το σειριακό πρόγραμμα σε έναν πόρο, τώρα διαμοιράζονται σε 2 πόρους στο παράλληλο πρόγραμμα κατάλληλα και γίνονται ταυτόχρονα. Όμως, για να έχουμε σωστή εκτύπωση αποτελεσμάτων πρέπει να γίνει σωστά ο συγχρονισμός, δηλαδή το κάθε νήμα να εκτυπώνει το αποτέλεσμα των υπολογισμών του και τα άλλα νήματα να το περιμένουν να τελειώσει ώστε να εκτυπώσουν και αυτά με την σειρά τους τα αποτελέσματά τους. Συνεπώς, το κρίσιμο τμήμα στον κώδικα κάθε νήματος είναι η εκτύπωση της γραμμής που έχει υπολογίσει. Οπότε μόνο σε αυτό το κομμάτι κώδικα θα έχουμε μπλοκάρισμα με χρήση σημαφόρων. Βέβαια, η εκτύπωση μιας γραμμής είναι κάτι που γίνεται αρκετά γρήγορα οπότε το μπλοκάρισμα των άλλων σημάτων θα είναι πολύ σύντομο, ενώ οι υπολογισμοί που απαιτεί κάθε γραμμή, που είναι ουσιαστικά το χρονοβόρο κομμάτι της διαδικασίας, γίνονται παράλληλα από όλα τα νήματα, και δεν πρέπει να περιέχονται στο κρίσιμο τμήμα. Αυτό ακριβώς είναι και που οδηγεί στο να εκτελείται το πρόγραμμα πολύ πιο γρήγορα. Ουσιαστικά αν βάζαμε και τους υπολογισμούς μέσα στο κρίσιμο τμήμα τότε δεν θα παρατηρούσαμε καμία ιδιαίτερη διαφορά με το αντίστοιχο σειριακό, εφόσον κάθε νήμα θα υπολόγιζε το αποτέλεσμα του μόνο αφότου

τυπωνόταν η προηγούμενη γραμμή από το προηγούμενο νήμα. Έτσι το πρόγραμμα θα εκφυλιζόταν στο αντίστοιχο σειριακό.

1.2.4 Ερώτηση 4^η

Αν πατήσουμε Ctrl-C, τότε διακόπτεται αμέσως η λειτουργία του προγράμματός μας και εμφανίζεται στο τερματικό μόνο το κομμάτι του συνόλου Mandelbrot που έχει προλάβει να τυπωθεί από το πρόγραμμα. Στην συνέχεια το τερματικό περνάει σε κατάσταση αναμονής και περιμένει την επόμενη εντολή από τον χρήστη. Τότε, αν πάμε να πληκτρολογήσουμε έναν χαρακτήρα θα παρατηρήσουμε πως έχει το χρώμα που είχε ο τελευταίος χαρακτήρας του συνόλου Mandelbrot που πρόλαβε να τυπωθεί πριν τερματιστεί. Για αυτό τον λόγο δημιουργούμε έναν δικό μας signal handler, και στον handler αυτού του σήματος ορίζουμε να καλείται αρχικά η συνάρτηση `reset_xterm_color(1)` και στην συνέχεια να τερματίζει το πρόγραμμα. Με αυτόν τον τρόπο θα επανέρχεται το τερματικό στην προηγούμενη κατάστασή του. Ακολουθεί ο κώδικας που χρησιμοποιήθηκε:

```
signal(SIGINT, unexpectedSignal);
```

Όπου `unexpectedSignal` είναι μια void συνάρτηση η οποία φαίνεται παρακάτω:

```
void unexpectedSignal(int sign)
{
    signal(sign, SIG_IGN);
    reset_xterm_color(1);
    exit(1);
}
```