Λειτουργικά Συστήματα Υπολογιστών

Κακούρης Δημήτριος (03119019)
Μαρδίκης Κωνσταντίνος (03119867)

3η Εργαστηριακή Άσκηση

1.1 Συγχρονισμός στον υπάρχοντα κώδικα

Ο πηγαίος κώδικας που χρησιμοποιήθηκε για την υλοποίηση είναι:

```c
/*
* 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 */
/* ... */
```

```
#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_fetch_and_add(&ip, 1); /*Critical section*/
         /* ... */
      }
       else {
         /* ... */
           pthread_mutex_lock(&lock);
         /* You cannot modify the following line */
         ++(*ip); /*Critical section*/
           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_fetch_and_add(&ip, -1); /*Critical section*/
         /* ... */
```

```
      } else {
         /* ... */
           pthread_mutex_lock(&lock);
         /* You cannot modify the following line */
         --(*ip); /*Critical section*/
           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;

   /*
    * Initial value
    */
   val = 0;

   /*
    * Create threads
    */
   if (pthread_mutex_init(&lock, NULL) != 0) {
       printf("\n mutex init has failed\n");
       return 1;
   }
   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(&lock);

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

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

    return ok;
}
```

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

```
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ make simplesync
gcc -Wall -O2 -pthread    simplesync.c   -o simplesync
simplesync.c:31:3: error: #error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
 # error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
   ^~~~~
make: *** [<builtin>: simplesync] Error 1
```

Με εκτέλεση του `make` (συγκεκριμένα αναγράφεται στη παραπάνω εικόνα `make simplesync`, δεν έχει διαφορά στα άλλα recipes ούτως ή άλλως δεν εμφανίζει κάποιο μήνυμα), παρατηρούμε πως πρέπει για να παραχθούν τα executables να δοθεί το κατάλληλο macro:

```
90 ▶  simplesync-mutex.o: simplesync.c
91        $(CC) $(CFLAGS) -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c
92
93 ▶  simplesync-atomic.o: simplesync.c
94        $(CC) $(CFLAGS) -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c
95
```

Παρατηρούμε τα δύο compiler directives, `-DSYNC_MUTEX` και `-DSYNC_ATOMIC`, πρόκειται για preprocessor macros. Συγκεκριμένα λαμβάνονται υπόψη από τον compiler κατα τη διάρκεια της μεταγλώττισης και εκτελεί τα κατάλληλα κομμάτια πηγαίου κώδικα ανάλογα το macro. Μπορούμε να δούμε το χειρισμό στο πηγαίο κώδικα στη παρακάτω εικόνα:

```
30  #if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
31  # error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
32  #endif
33
34  #if defined(SYNC_ATOMIC)
35  # define USE_ATOMIC_OPS 1
36  #else
37  # define USE_ATOMIC_OPS 0
38  #endif
```

Έχοντας υλοποιήσει συγχρονισμό τόσο με mutexes (simplesync-mutex) όσο και με atomic operations (`simplesync-atomic`) εκτελούμε και τα δύο και περιμένουμε τελικό `val=0` :

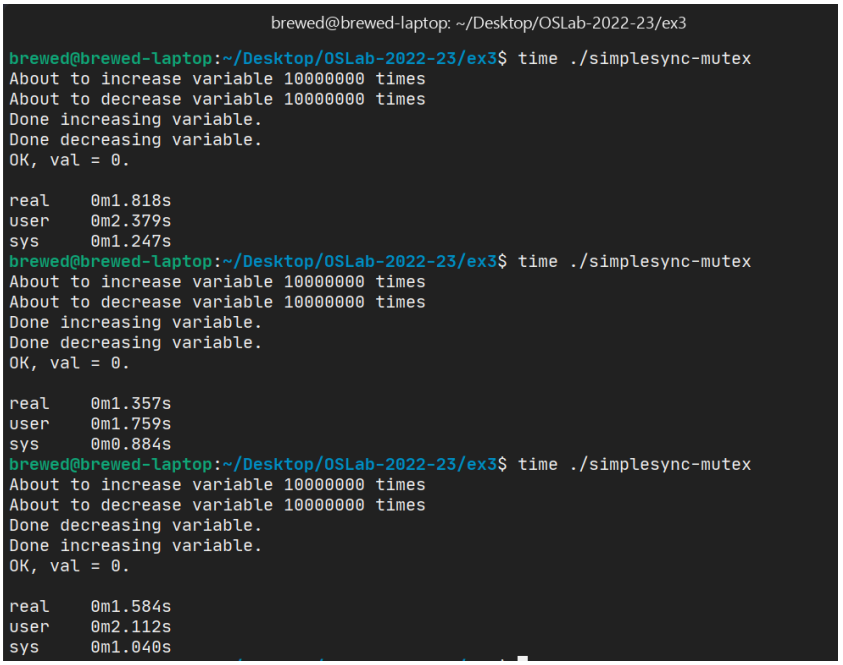
```
                    brewed@brewed-laptop: ~/Desktop/OSLab-2022-23/ex3
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

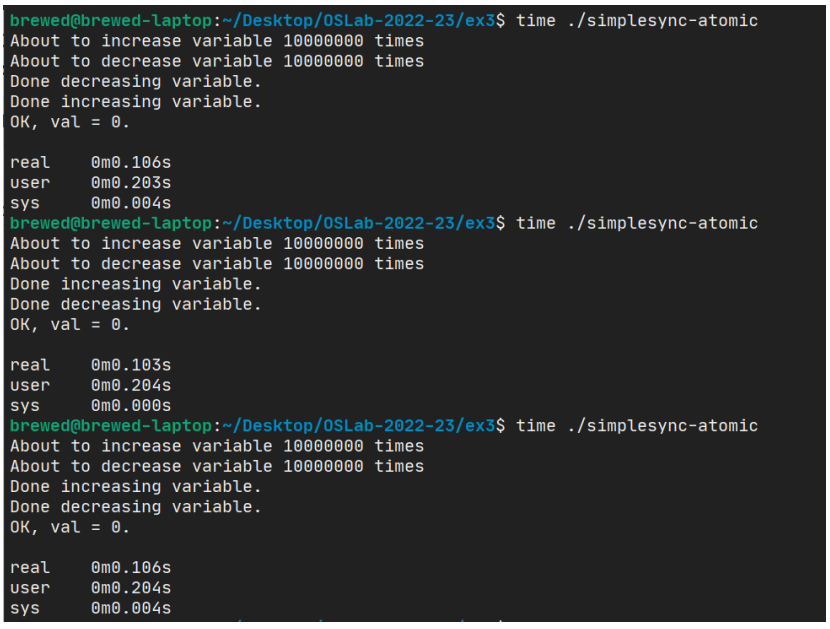
Ερωτήσεις:

1. Με τη χρήση του time χρονομετρούμε το μη συγχρονισμένο κώδικα του simplesync:

```
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 4385403.

real    0m0.045s
user    0m0.087s
sys     0m0.000s
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 5066274.

real    0m0.044s
user    0m0.081s
sys     0m0.004s
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -4109222.

real    0m0.041s
user    0m0.074s
sys     0m0.004s
```

Παρατηρούμε πως τα αποτελέσματα του `val` είναι κάθε φορά τυχαία και δεν εμφανίζουν κάποια συνέπεια μεταξύ τους. Αν τώρα δούμε τους χρόνους εκτέλεσης των δύο συγχρονισμένων υλοποιήσεων:





Το `val` είναι πάντα ίσο με 0, επομένως έχει επιτευχθεί συγχρονισμός των δυο threads. Οι χρόνοι είναι μεγαλύτεροι από τη μη συγχρονισμένη εκδοχή και αυτό διότι τόσο στο `simplesync-mutex` όσο και στο `simplesync-atomic`, βρίσκεται μόνο ένα thread κάθε φορά μέσα στο critical section του κώδικα, δηλαδή κάνει access το shared resource `*ip` μόνο ένα thread κάθε φορά. Ουσιαστικά μέσω του συγχρονισμού επιτυγχάνουμε να εκτελούνται σειριακά το increment και decrement.

Στη περίπτωση της μη συγχρονισμένης εκδοχής χονδρικά μιλώντας και οι δύο συναρτήσεις εκτελούνται παράλληλα με αποτέλεσμα όμως να μην γίνονται όλες οι επαναλήψεις τους λόγω race condition και να έχουμε αυτά τα αποτελέσματα του `val`.

2. Με το `time [executable]` χρονομετρούμε την εκτέλεση των δυο υλοποιήσεων και όπως φαίνεται στις δύο παραπάνω εικόνες έχουμε το χρόνο εκτέλεσης. Γενικά με κάθε τρέξιμο ο χρόνος εκτέλεσης δεν είναι ακριβώς ο ίδιος (μικροδιαφορές που μπορεί να οφείλονται σε context switching στο πυρήνα εκτέλεσης) αλλά με ένα μέσο όρο τριών εκτελέσεων:

$$t_{mutex} = \frac{1.818+1.357+1.584}{3} = 1.586\ sec$$

$$t_{atomic} = \frac{0.103+0.106+0.103}{3} = 0.104\ sec$$

Παρατηρούμε πως ο χρόνος εκτέλεσης με την υλοποίηση mutex παίρνει σαφώς περισσότερο χρόνο από την υλοποίηση με atomic operations. Τα POSIX mutexes σε επίπεδο assembly χρησιμοποιούν atomic operations μαζί με τον υπόλοιπο κώδικα τους και επομένως είναι αναμενόμενο ο χρόνος εκτέλεσης να είναι μεγαλύτερος.

3. Τροποποιούμε το makefile προκειμένου να συμπεριλάβουμε τα flags -S και -g , συγκεκριμένα προσθέτουμε τις παρακάτω δύο γραμμες:

```
simplesync-mutex.s: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_MUTEX -S -g -c -o simplesync-mutex.s simplesync.c

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

Ύστερα κάνουμε `make simplesync-mutex.s` και `make simplesync-atomic.s`:

```
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ make simplesync-mutex.s
gcc -Wall -O2 -pthread -DSYNC_MUTEX -S -g -c -o simplesync-mutex.s simplesync.c
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ make simplesync-atomic.s
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -S -g -c -o simplesync-atomic.s simplesync.c
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$
```

Κάνοντας `nano simplesync-atomic.s` , ψάχνουμε εντολές assembly οι οποίες αρχίζουν με lock καθώς αυτές υλοποιούν atomic operations, κανουν lock πριν εκτελέσουν την πραξη, στη προκειμένη:

- Atomic operation of addition:

```
.LBE19:
        .loc 1 52 0
        lock addq       $1, (%rsp)
```

- Atomic operation of subtraction:

```
.LBE27:
        .loc 1 78 0
        lock subq       $1, (%rsp)
```

4. Με `cat simplesync-mutex.s` αναζητούμε τα critical sections τόσο για increment οσο και για decrement:

Τα παρακάτω είναι κώδικας assembly χωρίς static linking εξού και το @PLT.

- Critical section of addition of `*ip`, between `pthread_mutex_lock` and `pthread_mutex_unlock`:

```
.LBE19:
        .loc 1 56 0
        movq    %rbp, %rdi
        call    pthread_mutex_lock@PLT
.LVL4:
        .loc 1 58 0
        movl    (%r12), %eax
        .loc 1 59 0
        movq    %rbp, %rdi
        .loc 1 58 0
        addl    $1, %eax
        movl    %eax, (%r12)
        .loc 1 59 0
        call    pthread_mutex_unlock@PLT
```

- Critical section of subtraction of `*ip`, between `pthread_mutex_lock` and `pthread_mutex_unlock`:

```
.LBE27:
        .loc 1 82 0
        movq    %rbp, %rdi
        call    pthread_mutex_lock@PLT
.LVL15:
        .loc 1 84 0
        movl    (%r12), %eax
        .loc 1 85 0
        movq    %rbp, %rdi
        .loc 1 84 0
        subl    $1, %eax
        movl    %eax, (%r12)
        .loc 1 85 0
        call    pthread_mutex_unlock@PLT
```

Για να δούμε κώδικα assembly της pthread_mutex_lock κάνουμε static linking διαφορετικά θα δούμε οχι τη δική του assembly αλλά ένα ενδιάμεσο κώδικα (stub).

```
brewed@brewed-laptop: ~/.../OSLab-2022-23/ex3
brewed@brewed-laptop:~/.../OSLab-2022-23/ex3$ gcc -g -DSYNC_MUTEX -static -o simplesync-mutex simplesync.c -pthr
ead
brewed@brewed-laptop:~/.../OSLab-2022-23/ex3$ gdb -q simplesync-mutex
Reading symbols from simplesync-mutex...
(gdb) disassemble pthread_mutex_lock
Dump of assembler code for function pthread_mutex_lock:
   0x00000000004149c0 <+0>:     endbr64
   0x00000000004149c4 <+4>:     mov    0x10(%rdi),%eax
   0x00000000004149c7 <+7>:     mov    %eax,%edx
   0x00000000004149c9 <+9>:     and    $0x17f,%edx
   0x00000000004149cf <+15>:    nop
   0x00000000004149d0 <+16>:    and    $0x7c,%eax
   0x00000000004149d3 <+19>:    jne    0x414a80 <pthread_mutex_lock+192>
   0x00000000004149d9 <+25>:    push   %rbx
   0x00000000004149da <+26>:    sub    $0x10,%rsp
   0x00000000004149de <+30>:    test   %edx,%edx
   0x00000000004149e0 <+32>:    jne    0x414a88 <pthread_mutex_lock+200>
   0x00000000004149e6 <+38>:    mov    0xb14d3(%rip),%r9d        # 0x4c5ec0 <__pthread_force_elision>
   0x00000000004149ed <+45>:    test   %r9d,%r9d
   0x00000000004149f0 <+48>:    jne    0x414a50 <pthread_mutex_lock+144>
   0x00000000004149f2 <+50>:    mov    0x10(%rdi),%esi
   0x00000000004149f5 <+53>:    and    $0x80,%esi
   0x00000000004149fb <+59>:    jne    0x414a06 <pthread_mutex_lock+70>
   0x00000000004149fd <+61>:    cmpb   $0x0,0xaf674(%rip)        # 0x4c4078 <__libc_single_threaded>
   0x0000000000414a04 <+68>:    jne    0x414a40 <pthread_mutex_lock+128>
   0x0000000000414a06 <+70>:    xor    %eax,%eax
   0x0000000000414a08 <+72>:    mov    $0x1,%edx
   0x0000000000414a0d <+77>:    lock cmpxchg %edx,(%rdi)
   0x0000000000414a11 <+81>:    jne    0x414ac8 <pthread_mutex_lock+264>
   0x0000000000414a17 <+87>:    mov    0x8(%rdi),%ecx
   0x0000000000414a1a <+90>:    test   %ecx,%ecx
--Type <RET> for more, q to quit, c to continue without paging--
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Σε αυτήν την άσκηση υλοποιήθηκαν δύο διαφορετικοί τρόποι για την επίτευξη του συγχρονισμού μεταξύ των νημάτων.

Ο πηγαίος κώδικας με **σημαφόρους**:

```c
/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <stdlib.h>
#include <semaphore.h>
#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000
#define perror_pthread(ret, msg) \
      do { errno = ret; perror(msg); } while (0)

/* A struct for passing arguments to threads */
typedef struct {
    int fd;
    sem_t *sem;
    int thr_id;
    int number_of_threads;
} thread_args;


/***************************
 * 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);
      }
}

/* This function is the entry point for each thread. It computes and
outputs lines of the Mandelbrot set */
void *compute_and_output_mandel_line_threaded(void *args){
    /* The color values for the line currently being computed */
    int color_val[x_chars];
    int fd = ((thread_args *)args)->fd;
    int thr_id = ((thread_args *)args)->thr_id;
    /* The array of semaphores used for synchronizing the threads */
    sem_t *sem = ((thread_args *)args)->sem;
    int number_of_threads = ((thread_args *)args)->number_of_threads;


    /*
     * Each thread computes and outputs a subset of the lines.
     * If there are N threads, thread 0 handles lines 0, N, 2N, 3N,
etc.,
     * thread 1 handles lines 1, N+1, 2N+1, 3N+1, etc., and so forth.
     */
    for(int i=thr_id; i<y_chars; i+=number_of_threads){
        compute_mandel_line(i, color_val);

        /*
```

```
         * Wait for the semaphore corresponding to this thread to become
available.*/
        sem_wait(&sem[(thr_id)%number_of_threads]);
        output_mandel_line(fd, color_val);
        /*
         * Signal the next thread in the order that it can now output
its line
         * (Round Robin fashion).
         * If this is the last thread, we wrap around to the first one.
         */
        sem_post(&sem[(thr_id+1)%number_of_threads]);
    }
    return 0;
}

int main(int argc, char *argv[]) {
    int ret;
    int NTHREADS;
    /* The number of threads is read from the program's arguments */
    NTHREADS = atoi(argv[1]);
    pthread_t threads[NTHREADS];
    sem_t sem[NTHREADS];


    /* The semaphores are initialized, the first semaphore's value
     * is equal to 1 so it can start */
    for (int i = 0; i <NTHREADS; i++) {
        if (i == 0) {
            sem_init(&sem[i], 0, 1);
        } else {
            sem_init(&sem[i], 0, 0);
        }
    }

    if (argc != 2) {
        printf("Usage: %s <NTHREADS> \n", argv[0]);
        exit(1);
    }


    thread_args targs[NTHREADS];
    for(int i=0; i<NTHREADS; i++){
       targs[i].fd = 1;
       targs[i].thr_id = i;
       targs[i].sem = sem;
       targs[i].number_of_threads = NTHREADS;
```

```
    }


    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.
     */

    for(int i=0; i<NTHREADS; i++){
        ret = pthread_create(&threads[i], NULL,
compute_and_output_mandel_line_threaded,&targs[i]);
        if (ret) {
            perror_pthread(ret, "pthread_create");
            exit(1);
        }
    }


    /* The main thread waits for all the worker threads to finish */
    for (int i = 0; i < NTHREADS; i++) {
        ret = pthread_join(threads[i], NULL);
        if (ret) perror_pthread(ret, "pthread_join");
    }


    /* The semaphores are destroyed */
    for(int i=0; i<y_chars; i++) {
        ret = sem_destroy(&sem[i]);
        if (ret) perror("sem_destroy");
    }

    /* Reset the terminals color */
      reset_xterm_color(1);
      return 0;
}
```

Ο πηγαίος κώδικας με **μεταβλητές συνθήκης**:

```c
/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm.
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <stdlib.h>
#include <semaphore.h>
#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000
#define perror_pthread(ret, msg) \
      do { errno = ret; perror(msg); } while (0)

/* A struct for passing arguments to threads */
typedef struct {
    int fd;
    int thr_id;
    int number_of_threads;
    int *running;
    pthread_mutex_t* mutex;
    pthread_cond_t* cond;
} thread_args;


/***************************
 * 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);
      }
}

/* This function computes(this in parallel) and outputs the mandelbrot
lines
* The subset of lines is determined by the thread ID and the total
number of threads.
* This function uses a mutex and a condition variable to ensure that the
threads output their lines
* in the correct order. */
void *compute_and_output_mandel_line_threaded(void *args){
    int color_val[x_chars];
    int fd = ((thread_args *)args)->fd;
    int thr_id = ((thread_args *)args)->thr_id;
    int *running= ((thread_args *)args)->running;
    pthread_mutex_t *mutex= ((thread_args *)args)->mutex;
    pthread_cond_t *cond= ((thread_args *)args)->cond;
    int number_of_threads = ((thread_args *)args)->number_of_threads;

    /*
     * Each thread computes and outputs a subset of the lines.
     * If there are N threads, thread 0 handles lines 0, N, 2N, 3N,
etc.,
```

```
     * thread 1 handles lines 1, N+1, 2N+1, 3N+1, etc., and so on.
     */
    for(int i=thr_id; i<y_chars; i+=number_of_threads){

        compute_mandel_line(i, color_val);

        pthread_mutex_lock(mutex);
        /* Wait for the condition variable if another thread is
currently writing its line */
        while ((*running) !=thr_id){
            pthread_cond_wait(cond, mutex); /* critical section */
        }
        output_mandel_line(fd, color_val);
        /* Update the running variable to match the id of the next
thread (Round-Robin fashion) */
        (*running)=((*running)+1)%number_of_threads;

        /* Broadcast to all waiting threads that the condition has
changed */
        pthread_cond_broadcast(cond);

        pthread_mutex_unlock(mutex);

    }
    return 0;
}

int main(int argc, char *argv[]) {
    int ret;
    int NTHREADS;
    NTHREADS = atoi(argv[1]);
    pthread_t threads[NTHREADS];
    pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
    pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
    int running=0;


    if (argc != 2) {
        printf("Usage: %s <NTHREADS> \n", argv[0]);
        exit(1);
    }

    thread_args targs[NTHREADS];

    for(int i=0; i<NTHREADS; i++){
       targs[i].fd = 1;
```

```
        targs[i].thr_id = i;
        targs[i].running = &running;
        targs[i].mutex = &mutex;
        targs[i].cond = &cond;
        targs[i].number_of_threads = NTHREADS;

    }


    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.
     */

    for(int i=0; i<NTHREADS; i++){
        ret = pthread_create(&threads[i], NULL,
compute_and_output_mandel_line_threaded,&targs[i]);
        if (ret) {
            perror_pthread(ret, "pthread_create");
            exit(1);
        }
    }



    /* The main thread waits for all the worker threads to finish */
    for (int i = 0; i < NTHREADS; i++) {
        ret = pthread_join(threads[i], NULL);
        if (ret) perror_pthread(ret, "pthread_join");
    }

    /* Clean up the condition variable and the mutex */
    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&mutex);

    /* Reset the terminal to original state */
      reset_xterm_color(1);
      return 0;
}
```

Ερωτήσεις:

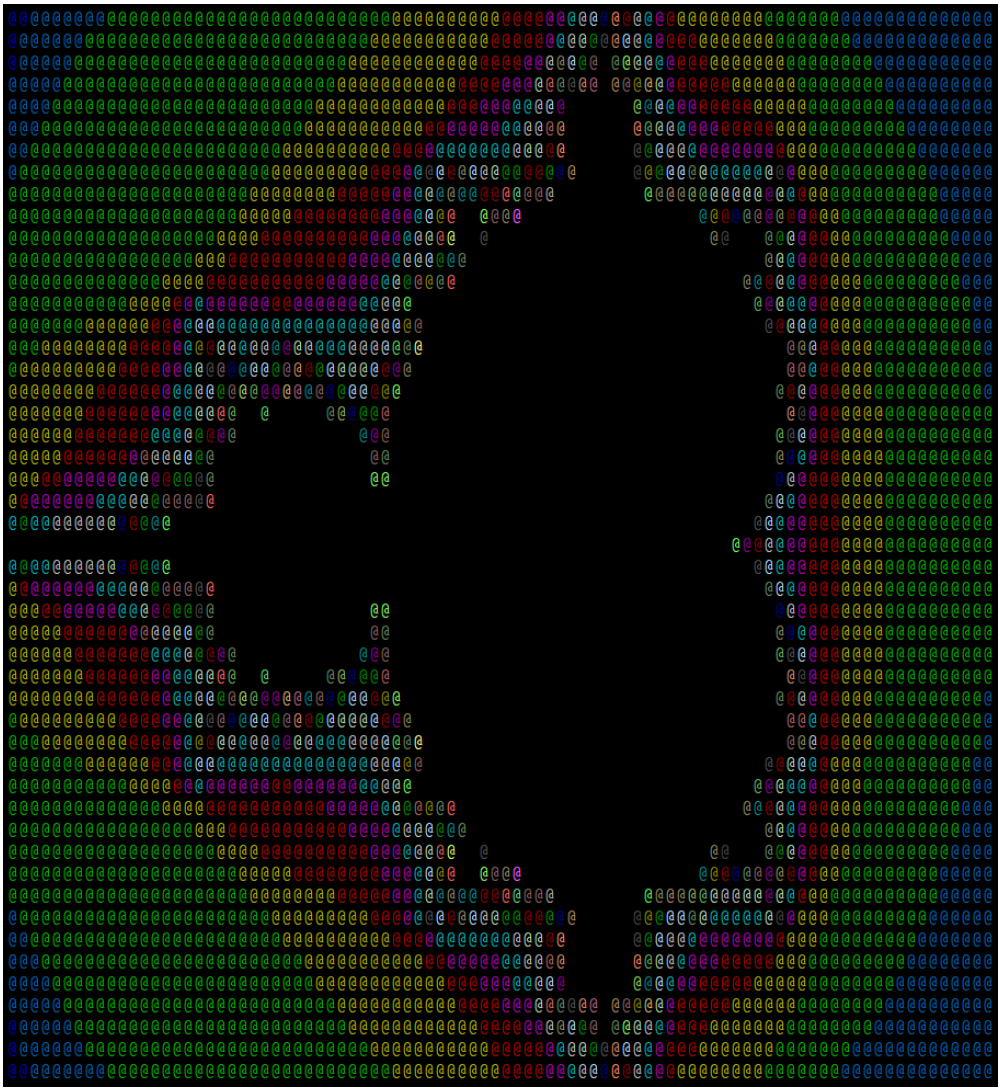
1. Κάθε νήμα αναλαμβάνει να υπολογίσει συγκεκριμένες γραμμές, με την κατανομή να γίνεται ανά γραμμή. Με αυτήν την λογική αποφασίσαμε να χρησιμοποιήσουμε τόσους σημαφόρους όσα και τα νήματα εκτέλεσης.

   Αρχικά, όλοι οι σημαφόροι έχουν την τιμή 0, εκτός από τον σημαφόρο του πρώτου νήματος που έχει την τιμή 1.

```c
/* The semaphores are initialized, the first semaphore's value
 * is equal to 1 so it can start */
for (int i = 0; i <NTHREADS; i++) {
    if (i == 0) {
        sem_init(&sem[i], 0, 1);
    } else {
        sem_init(&sem[i], 0, 0);
    }
}
```

   Έτσι, το πρώτο νήμα μπαίνει πρώτο στο κρίσιμο τμήμα και εκτυπώνει τη γραμμή με αριθμό 0, και στη συνέχεια ξεκλειδώνει τον σημαιοφόρο του επόμενου νήματος που θα εκτυπώσει τη γραμμή 1. Αυτή η διαδικασία επαναλαμβάνεται με κάθε νήμα να περιμένει το προηγούμενο νήμα να το ξεκλειδώσει, να εκτυπώνει τη γραμμή που του αντιστοιχεί, και να ξεκλειδώνει το επόμενο νήμα. Με αυτόν τον τρόπο επιτυγχάνεται η σωστή σειρά εμφάνισης των γραμμών του Mandelbrot set στο τερματικό, ενώ ο υπολογισμός παραμένει παράλληλος. Η παραπάνω λογική ουσιαστικά εκτελείται στην συνάρτηση :
   `compute_and_output_mandel_line_threaded`

   Το αποτέλεσμα είναι το παρακάτω με την εντολή `./mandel-sem 10` (καλύτερο αποτέλεσμα έχουμε με 8-10 νήματα, οι μετρήσεις έγιναν με τετραπύρηνο επεξεργαστή που υποστήριζε hyperthreading άρα 8 νήματα άρα αναμενόμενο peak επιτάχυνσης στα 8 νήματα):



2. Με `cat /proc/cpuinfo`:

```
brewed@brewed-laptop:~/Desktop/OSLab-2022-23/ex3$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 142
model name      : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
stepping        : 11
microcode       : 0xf0
cpu MHz         : 1799.972
cache size      : 8192 KB
physical id     : 0
siblings        : 8
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
```

Οι μετρήσεις έγιναν με τετραπύρηνο επεξεργαστή που υποστήριζε hyperthreading άρα 8 νήματα.

Με `time ./mandel-sem 1`

Χρόνος εκτέλεσης: **0.828 sec**

Με `time ./mandel-sem 2`

Χρόνος εκτέλεσης: **0.480 sec**

Ο χρόνος αυτός είναι πολύ κοντά στο μισό του χρόνου εκτέλεσης με την υλοποίηση με ένα νήμα. Επαληθεύεται το ότι αναμένουμε υποδιπλασιασμό του χρόνου σε σχέση με τη σειριακή υλοποίηση.

3. Στην υλοποίηση με condition variables χρησιμοποιούμε μόνο ένα condition variable, στο οποίο κάνουν wait τα νήματα που περιμένουν έχοντας τελειώσει τον υπολογισμό της γραμμής προκειμένου να τυπώσουν. Για τη σωστή σειρά χρησιμοποιούμε το int running που δίνει round-robin fashion την άδεια στο επόμενο κατα σειρά νήμα να τυπώσει.

4. Το παράλληλο πρόγραμμα που φτιάξαμε εμφανίζει επιτάχυνση και αυτό διότι επιλέγουμε στο κρίσιμο τμήμα να αφήσουμε μόνο το κομμάτι της εκτύπωσης γραμμών. Είναι το μόνο που είναι απαραίτητο να εκτελεστεί σειριακά ώστε να έχουμε σωστή σειρά στις εκτυπωμένες γραμμές. Ο υπολογισμός μιας γραμμής του mandelbrot set είναι ανεξάρτητος από τις άλλες γραμμές του set και μπορεί να γίνει παράλληλα, ανήκει στη κατηγορία των **“embarrassingly parallel”** tasks.

   Το κρίσιμο τμήμα για κάθε υλοποίηση:

   Για `mandel-sem`:

```
/*
 * Wait for the semaphore corresponding to this thread to become available.*/
sem_wait(&sem[(thr_id)%number_of_threads]);

output_mandel_line(fd, color_val); /* critical section */
/*
 * Signal the next thread in the order that it can now output its line
 * (Round Robin fashion).
 * If this is the last thread, we wrap around to the first one.
 */
sem_post(&sem[(thr_id+1)%number_of_threads]);
```

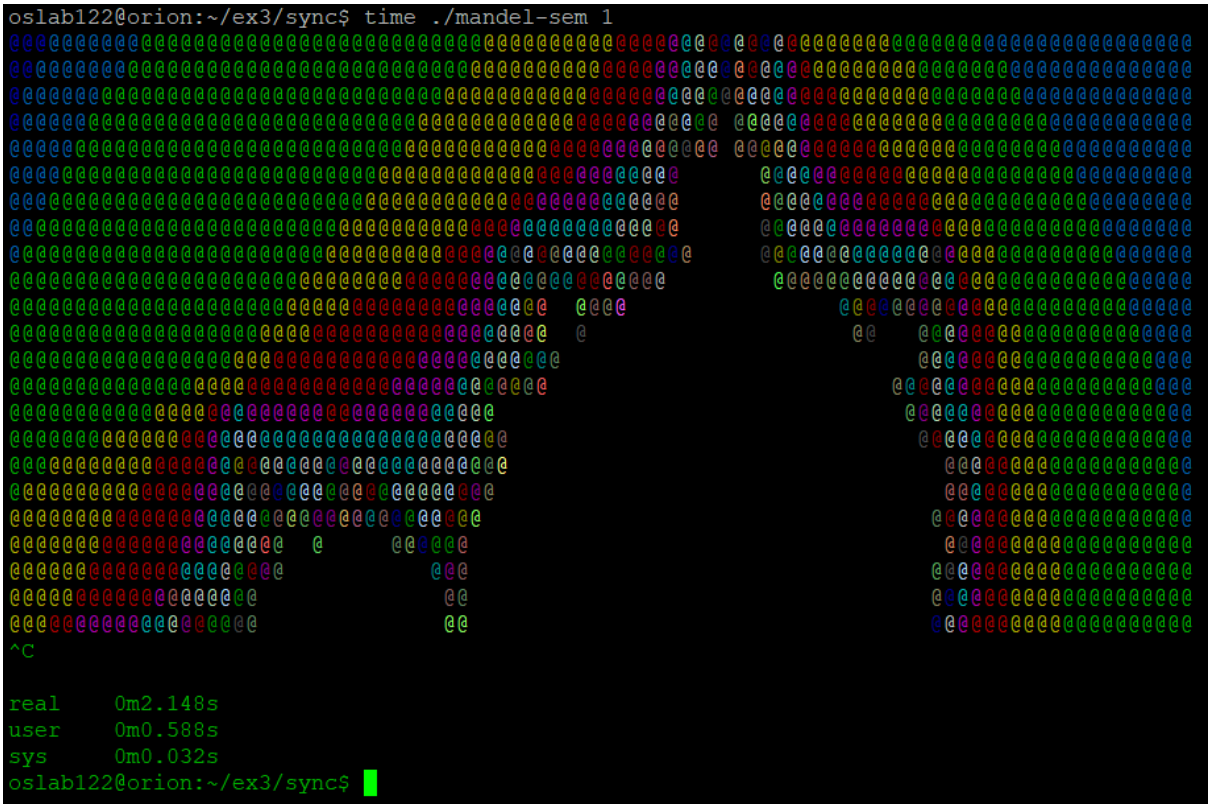
   Εντοπίζουμε το critical section ανάμεσα στο `sem_wait()` και `sem_post()`.

   Για `mandel-var`:

```
pthread_mutex_lock(mutex);
/* Wait for the condition variable if another thread is currently writing its line */
while ((*running) !=thr_id){
    pthread_cond_wait(cond, mutex);
}
output_mandel_line(fd, color_val); /* critical section */
/* Update the running variable to match the id of the next thread (Round-Robin fashion) */
(*running)=((*running)+1)%number_of_threads;

/* Broadcast to all waiting threads that the condition has changed */
pthread_cond_broadcast(cond);

pthread_mutex_unlock(mutex);
```

   Εντοπίζουμε το critical section αμέσως μετά απο το `while()`.

5. Όταν εκτελείται το πρόγραμμα, το χρώμα των γραμμάτων στο τερματικό διατηρείται ίδιο με το χρώμα του τελευταίου χαρακτήρα που εκτυπώθηκε. Αυτό επιτυγχάνεται χρησιμοποιώντας τις συναρτήσεις `set_xterm_color()` και `output_mandel_line()`. Εάν το πρόγραμμα ολοκληρωθεί κανονικά χωρίς να διακοπεί (δηλαδή δεν πατηθεί το Ctrl-C), το χρώμα των γραμμάτων επανέρχεται στο προεπιλεγμένο χρώμα του τερματικού χρησιμοποιώντας τη συνάρτηση `reset_xterm_color()`. Επομένως, πατώντας το Ctrl-C :



Για να αποφύγουμε αυτήν τη μη επιθυμητή κατάσταση, μπορούμε να ελέγξουμε αν έχει πατηθεί το Ctrl+C χρησιμοποιώντας το σήμα SIGINT και τη συνάρτηση signal(). Η signal() επιτρέπει την χρήση ενός χειριστή σήματος (signal handler) για ένα συγκεκριμένο σήμα. Με τη χρήση της signal(), μπορούμε να ορίσουμε μια συνάρτηση που θα εκτελείται όταν ληφθεί το σήμα SIGINT.

Για παράδειγμα:

```c
int reset_color_flag = 0;

void sigint_handler(int signum) {
    reset_color_flag = 1;
}
```

```c
int main(int argc, char *argv[]) {

    signal(SIGINT, sigint_handler);

    if (reset_color_flag) {
        reset_xterm_color(1);
    }

    return 0;
}
```