Λειτουργικά Συστήματα
3η Εργαστηριακή Αναφορά

Συγχρονισμός

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2022-2023

Ομάδα: oslab35

Μέλη

Καμπουγέρης Χαράλαμπος | A.M: el20098

Κουστένης Χρίστος | A.M: el20227

1.Συγχρονισμός σε υπάρχοντα κώδικα

Δίνεται το πρόγραμμα `simplesync.c`, το οποίο λειτουργεί ως εξής: Αφού αρχικοποιήσει μια μεταβλητή `val = 0`, δημιουργεί δύο νήματα τα οποία εκτελούνται ταυτόχρονα: το πρώτο νήμα αυξάνει N φορές την τιμή της μεταβλητής `val` κατά 1, το δεύτερο τη μειώνει N φορές κατά 1.

- Με την εντολή make τρέχουμε το παρεχόμενο Makefile. Παρατηρούμε ότι από την μεταγλώττιση ενός simplesync.c αρχείου παράγονται δύο εκτελέσιμα : το simplesync-atomic και το simplesync-mutex. Εξετάζοντας τον κώδικα simplesync.c παρατηρούμε ότι πριν την main() και στην αρχή του αρχείου simplesync.c χρησιμοποιούνται οι μακροεντολές που φαίνονται στον Κώδικα 1. Αυτές εξασφαλίζουν αφενός ότι ακριβώς ένα από τα macro names SYNC_MUTEX και SYNC_ATOMIC θα είναι predefined στη μεταγλώττιση και αφετέρου αν αυτό είναι το SYNC_ATOMIC να ορίζεται ως 1 το macro_name USE_ATOMIC_OPS αλλιώς να ορίζεται ως 0. Τρέχοντας οποιοδήποτε από τα εκτελέσιμα `simplesync-atomic simplesync-mutex` παρατηρούμε το εξής αποτέλεσμα:

```
chris@LAPTOP-TK5Q3T95:/mnt/c/Users/koust/ECE/6th Semester/OS/os-lab-exer3/sync$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
```

Έξοδος 1

το οποίο μας δείχνει ότι δημιουργείται συναγωνισμός των νημάτων αύξησης και μείωσης της μεταβλητής val καθώς και ότι δεν υπάρχει συγχρονισμός μεταξύ τους.

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

Κώδικας 1

Στην main(), η ροή εκτέλεσης εξασφαλίζει ότι ανάλογα με την τιμή USE_ATOMIC_OPS θα τρέξουν αποκλειστικά τα κομμάτια κώδικα που αφορούν atomic operations είτε εκείνα που χρησιμοποιούν κλείδωμα mutex.

- Το flag -Dmacroname ορίζει ποια μακροεντολή θα χρησιμοποιηθεί στο πρόγραμμα κάτι το οποίο προσδιορίζεται από τον preprocessor. Στην περίπτωση μας χρησιμοποιείται το flag -DSYNC_MUTEX για να παραχθεί το εκτελέσιμο που χρησιμοποιεί mutex locks και το flag -DSYNC_ATOMIC για να παραχθεί το εκτελέσιμο που χρησιμοποιεί atomic operations. Έτσι όταν γίνει η μεταγλώττιση που έπεται του preprocessor τα .o αρχεία που θα παραχθούν θα είναι διαφορετικά και άρα και τα εκτελέσιμά τους διαφορετικά.

```
simplesync-mutex: simplesync-mutex.o
    $(CC) $(CFLAGS) -o simplesync-mutex simplesync-mutex.o $(LIBS)

simplesync-atomic: simplesync-atomic.o
    $(CC) $(CFLAGS) -o simplesync-atomic simplesync-atomic.o $(LIBS)

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
```

Κώδικας 2 (Makefile)

- Ακολουθεί ο κώδικας που επιτελεί σωστά την αύξηση και μείωση της μεταβλητής val είτε με atomic operations είτε με mutex_locks ώστε το τελικό αποτέλεσμα να είναι 0.

```
/*
 * 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 mutex = PTHREAD_MUTEX_INITIALIZER; // pthread_mutex_t mutex;
                                                   // pthread_mutex_init ( &mutex,
NULL);

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 */
            //__atomic_fetch_add(ip, 1,3);
            __sync_fetch_and_add(ip, 1);
            /* ... */
        }
        else
        {
            /* ... */
            int err_ret;
            err_ret = pthread_mutex_lock(&mutex);
            if (err_ret != 0)
            {
                perror_pthread(err_ret, "lock problem on increasing");
            }
            /* You cannot modify the following line */
            ++(*ip);
            /* ... */
            err_ret = pthread_mutex_unlock(&mutex);
            if (err_ret != 0)
            {
                perror_pthread(err_ret, "unlock problem on increasing");
            }
        }
    }
    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_sub(ip, 1);
            /* ... */
        }
        else
        {
            /* ... */
            int err_ret;
            err_ret = pthread_mutex_lock(&mutex);
            if (err_ret != 0)
            {
                perror_pthread(err_ret, "lock problem on decreasing");
            }
            /* You cannot modify the following line */
            --(*ip);
            /* ... */
            err_ret = pthread_mutex_unlock(&mutex);
            if (err_ret != 0)
            {
                perror_pthread(err_ret, "unlock problem on decreasing");
            }
        }
    }
    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_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);
    if (!USE_ATOMIC_OPS)
    {
        ret = pthread_mutex_destroy(&mutex);
        if (ret != 0)
        {
            perror_pthread(ret, "Mutex object not destroyed..\n");
        }
    }
    return ok;
}
```

Κώδικας 3(simplesync.c)

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;

   Παρατηρούμε ότι ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό είναι μεγαλύτερος σε σχέση με το χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό. Αυτό συμβαίνει, διότι στο αρχικό πρόγραμμα (χωρίς συγχρονισμό), δεν υπάρχει ο περιορισμός κάθε νήμα να εκτελεί ατομικά τον κώδικα στο κρίσιμο τμήμα που έχουμε ορίσει, και επομένως η εκτέλεση του κρίσιμου τμήματος από τα δύο νήματα είναι **παράλληλη**. Όταν συγχρονίζουμε τα δύο νήματα (και με τους δύο μηχανισμούς συγχρονισμού που χρησιμοποιήσαμε παραπάνω), τότε στο κρίσιμο τμήμα βρίσκεται το πολύ ένα νήμα και επομένως η εκτέλεσή του από τα δύο νήματα είναι **σειριακή**.

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

   Είναι φανερό πως όταν χρησιμοποιούμε ατομικές λειτουργίες του GCC για συγχρονισμό το πρόγραμμα εκτελείται ταχύτερα σε σχέση με όταν χρησιμοποιούμε το εργαλείο συγχρονισμού POSIX mutexes. Με τη χρήση ατομικών λειτουργιών πετυχαίνουμε έναν low-level συγχρονισμό αφού εκτελούμε πράξεις άμεσα στο υλικό ενώ οι μεταβλητές που χρησιμοποιούμε προστατεύονται από memory barriers όσο εκτελούνται πράξεις στο κρίσιμο τμήμα. Δυστυχώς, οι πράξεις που μας παρέχονται για χρήση είναι πολύ συγκεκριμένες και σε πιο πολύπλοκα προγράμματα παρέχουν περιορισμένες δυνατότητες. Τότε, επωφελούμαστε από τη χρήση των mutexes που υλοποιούν εσωτερικά atomic operations για να κάνουν lock και unlock τα κρίσιμα τμήματα του κώδικα. Τα mutexes υστερούν όμως σε ταχύτητα σε σχέση με τα atomic operations γιατί ο συγχρονισμός τους είναι γενικά πιο χρονοβόρος και απαιτεί context switch δηλαδή μετάβαση από user σε kernel state και αντιστρόφως. Αυτό συμβαίνει γιατί τα mutexes αναστέλλουν τη λειτουργία του thread που δε χρησιμοποιείται και απελευθερώνουν πόρους της ΚΜΕ. Αντιθέτως, τα atomic operations διατηρούν σε μία «ενεργό» αναμονή τα thread που περιμένουν την εκτέλεση τους δεσμεύοντας πόρους του συστήματος μέχρι να γίνει άρση του memory barrier. Αυτό καθιστά τα atomic operations ιδανικά για σενάρια συγχρονισμού όπου ο συναγωνισμός (race) είναι σχετικά χαμηλός αλλά ακατάλληλα για καταστάσεις αυξημένου ανταγωνισμού όπου η χρήση των mutexes έχει πλεονεκτήματα.

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`.

Πραγματοποιούμε αρχικά τις κατάλληλες προσθήκες στο Makefile όπως φαίνεται στον Κώδικα 4 ώστε να παραχθούν τα αντίστοιχα αρχεία Assembly για την υλοποίηση του προγράμματος με atomic operations.

```
37  simplesync-atomic.s: simplesync.c
38      $(CC) $(CFLAGS) -DSYNC_ATOMIC -S -g -o simplesync-atomic.s simplesync.c
39
```

Κώδικας 4

Ακολουθούν τα κομμάτια κώδικα assembly που αντιστοιχούν:

- Στο νήμα αύξησης

```
.LBE15:
    .loc 1 54 3 is_stmt 1 view .LVU16
    .loc 1 59 4 view .LVU17
    lock addl   $1, (%rbx)
    .loc 1 77 5 view .LVU18
    .loc 1 51 22 view .LVU19
```

Κώδικας 5

- και στο νήμα μείωσης

```
.LBE21:
    .loc 1 94 3 is_stmt 1 view .LVU46
    .loc 1 98 4 view .LVU47
    lock subl   $1, (%rbx)
    .loc 1 116 5 view .LVU48
    .loc 1 91 22 view .LVU49
```

Κώδικας 6

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.

Κάνουμε αρχικά την παρακάτω προσθήκη στο Makefile όμοια με πριν.

```
34  simplesync-mutex.s: simplesync.c
35      $(CC) $(CFLAGS) -DSYNC_MUTEX -S -g -o simplesync-mutex.s simplesync.c
36
```

Κώδικας 7

Ακολουθούν τα κομμάτια κώδικα assembly που αντιστοιχούν:

- Στο νήμα αύξησης

```
.LBB26:
    .loc 1 65 4 view .LVU36
    .loc 1 66 4 view .LVU37
    .loc 1 66 14 is_stmt 0 view .LVU38
    movq    %r13, %rdi
    call    pthread_mutex_lock@PLT
```

```
.LBB25:
    .loc 1 69 5 discriminator 1 view .LVU22
    .loc 1 72 4 discriminator 1 view .LVU23
    .loc 1 72 7 is_stmt 0 discriminator 1 view .LVU24
    movl    (%r12), %eax
    .loc 1 74 14 discriminator 1 view .LVU25
    movq    %r13, %rdi
    .loc 1 72 4 discriminator 1 view .LVU26
    addl    $1, %eax
    movl    %eax, (%r12)
    .loc 1 74 4 is_stmt 1 discriminator 1 view .LVU27
    .loc 1 74 14 is_stmt 0 discriminator 1 view .LVU28
    call    pthread_mutex_unlock@PLT
```

Κώδικας 8

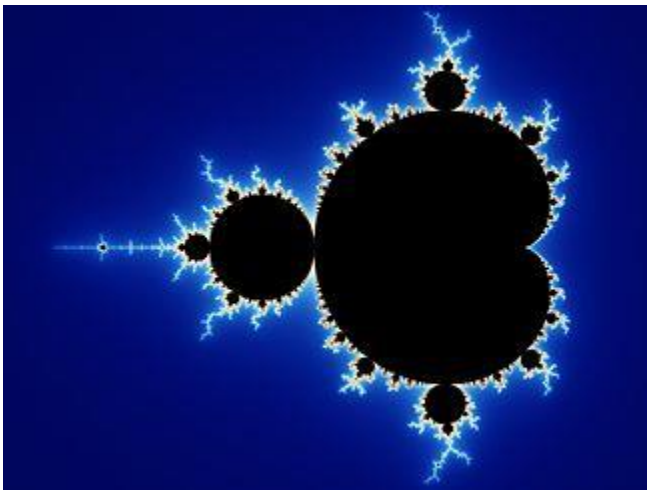
- και στο νήμα μείωσης

```
.LBB41:
    .loc 1 104 4 view .LVU98
    .loc 1 105 4 view .LVU99
    .loc 1 105 14 is_stmt 0 view .LVU100
    movq    %r13, %rdi
    call    pthread_mutex_lock@PLT
```

```
.LBB40:
    .loc 1 108 5 discriminator 1 view .LVU84
    .loc 1 111 4 discriminator 1 view .LVU85
    .loc 1 111 7 is_stmt 0 discriminator 1 view .LVU86
    movl    (%r12), %eax
    .loc 1 113 14 discriminator 1 view .LVU87
    movq    %r13, %rdi
    .loc 1 111 4 discriminator 1 view .LVU88
    subl    $1, %eax
    movl    %eax, (%r12)
    .loc 1 113 4 is_stmt 1 discriminator 1 view .LVU89
    .loc 1 113 14 is_stmt 0 discriminator 1 view .LVU90
    call    pthread_mutex_unlock@PLT
```

Κώδικας 9

2.Παράλληλος υπολογισμός του συνόλου Mandelbrot



Αρχικά, παρατείθεται ο πηγαίος κώδικας του προγράμματος και για τις δύο τεχνικές συγχρονισμού ποο ζητούνται:

1. Με σημαφόρους

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

#include <stdio.h>
#include <unistd.h>
```

```c
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>
#include <signal.h>
#include <errno.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)

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

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);
    }
```

```
60.
61.     return p;
62.}
63.
64.void sig_handler(int signum)
65.{
66.     printf("   Color will be restored after signal : %d\n", signum);
67.     reset_xterm_color(1);
68.     printf("Color restored\n");
69.     printf("Exiting...\n");
70.     exit(EXIT_SUCCESS);
71.}
72.
73./*
74. * Output at the terminal is is x_chars wide by y_chars long
75. */
76.int y_chars = 50;
77.int x_chars = 90;
78.
79./*
80. * The part of the complex plane to be drawn:
81. * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
82. */
83.double xmin = -1.8, xmax = 1.0;
84.double ymin = -1.0, ymax = 1.0;
85.
86./*
87. * Every character in the final output is
88. * xstep x ystep units wide on the complex plane.
89. */
90.double xstep;
91.double ystep;
92.
93./*
94. * This function computes a line of output
95. * as an array of x_char color values.
96. */
97.void compute_mandel_line(int line, int color_val[])
98.{
99.     /*
100.          * x and y traverse the complex plane.
101.          */
102.         double x, y;
103.
104.         int n;
105.         int val;
106.
107.         /* Find out the y value corresponding to this line */
108.         y = ymax - ystep * line;
```

```
109.
110.         /* and iterate for all points on this line */
111.         for (x = xmin, n = 0; n < x_chars; x += xstep, n++)
112.         {
113.
114.             /* Compute the point's color value */
115.             val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
116.             if (val > 255)
117.                 val = 255;
118.
119.             /* And store it in the color_val[] array */
120.             val = xterm_color(val);
121.             color_val[n] = val;
122.         }
123.     }
124.
125.     /*
126.      * This function outputs an array of x_char color values
127.      * to a 256-color xterm.
128.      */
129.     void output_mandel_line(int fd, int color_val[])
130.     {
131.         int i;
132.
133.         char point = '@';
134.         char newline = '\n';
135.
136.         for (i = 0; i < x_chars; i++)
137.         {
138.             /* Set the current color, then output the point */
139.             set_xterm_color(fd, color_val[i]);
140.             if (write(fd, &point, 1) != 1)
141.             {
142.                 perror("compute_and_output_mandel_line: write point");
143.                 exit(1);
144.             }
145.         }
146.
147.         /* Now that the line is done, output a newline character */
148.         if (write(fd, &newline, 1) != 1)
149.         {
150.             perror("compute_and_output_mandel_line: write newline");
151.             exit(1);
152.         }
153.     }
154.
155.     // The Semaphore
156.     sem_t *S;
157.     int threadcnt;
```

```
158.
159.      void *compute_and_output_mandel_line(void *arg)
160.      {
161.          /*
162.           * A temporary array, used to hold color values for the line being drawn
163.           */
164.          int color_val[x_chars], index;
165.          int thread_index = (__intptr_t)arg;
166.          for (index = thread_index; index < y_chars; index += threadcnt)
167.          {
168.              compute_mandel_line(index, color_val);
169.              if (sem_wait(&S[thread_index]) < 0)
170.              {
171.                  perror("sem_wait");
172.              }
173.              output_mandel_line(1, color_val);
174.              if (sem_post(&S[(thread_index + 1) % threadcnt]) < 0)
175.              {
176.                  perror("sem_post");
177.              }
178.          }
179.          return NULL;
180.      }
181.
182.      int main(int argc, char *argv[])
183.      {
184.
185.          if (safe_atoi(argv[1], &threadcnt) < 0 || threadcnt <= 0)
186.          {
187.              fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
188.              exit(1);
189.          }
190.
191.          struct sigaction act;
192.          act.sa_handler = sig_handler;
193.          sigemptyset(&act.sa_mask);
194.          act.sa_flags = 0;
195.          if (sigaction(SIGINT, &act, NULL) == -1)
196.          {
197.              perror("sigaction");
198.              exit(EXIT_FAILURE);
199.          }
200.
201.          pthread_t *threads;
202.          S = (sem_t *)safe_malloc(threadcnt * sizeof(*S));
203.          threads = (pthread_t *)safe_malloc(threadcnt * sizeof(*threads));
204.
205.          int ret;
206.          xstep = (xmax - xmin) / x_chars;
```

```
207.         ystep = (ymax - ymin) / y_chars;
208.
209.         // Initialiaze the semaphores
210.         sem_init(&S[0], 0, 1);
211.         for (int j = 1; j < threadcnt; ++j)
212.         {
213.             sem_init(&S[j], 0, 0);
214.         }
215.         /*
216.          * draw the Mandelbrot Set, one line at a time.
217.          * Output is sent to file descriptor '1', i.e., standard output.
218.          */
219.         /* Spawn threadcnt new threads */
220.         for (int i = 0; i < threadcnt; ++i)
221.         {
222.             ret = pthread_create(&threads[i], NULL, compute_and_output_mandel_line,
    (void *)(__intptr_t)i);
223.             if (ret)
224.             {
225.                 perror_pthread(ret, "pthread_create");
226.                 exit(1);
227.             }
228.         }
229.         /*
230.          * Wait for all threads to terminate
231.          */
232.         for (int i = 0; i < threadcnt; i++)
233.         {
234.             ret = pthread_join(threads[i], NULL);
235.             if (ret)
236.             {
237.                 perror_pthread(ret, "pthread_join");
238.                 exit(1);
239.             }
240.         }
241.         for (int i = 0; i < threadcnt; i++)
242.         {
243.             if (sem_destroy(&S[i]) < 0)
244.             {
245.                 perror("sem_destroy");
246.                 exit(1);
247.             }
248.         }
249.
250.         free(S);
251.         free(threads);
252.         reset_xterm_color(1);
253.         return 0;
254.     }
```

Κώδικας 10

2. Με μεταβλητές συνθήκης(conditional variables)

```c
/*
 * 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 <signal.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)

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

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;
}

void sig_handler(int signum)
{
    printf("   Color will be restored after signal : %d\n", signum);
    reset_xterm_color(1);
    printf("Color restored\n");
    printf("Exiting...\n");
    exit(EXIT_SUCCESS);
}
/**************************
 * 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;
```

```c
/*
 * 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);
    }
}

// The Condition variable and the mutex
// pthread_cond_t *MyTurn;
pthread_cond_t MyTurn;
pthread_mutex_t linewriter = PTHREAD_MUTEX_INITIALIZER;
int threadcnt;
int g_index = -1;
int err_ret;

void waiting_for_myturn(int thread_index, int index_line, int color_val[])
{

    err_ret = pthread_mutex_lock(&linewriter);
    if (err_ret != 0)
    {
        perror_pthread(err_ret, "lock problem");
    }
    while (index_line != g_index + 1)
    {
        // pthread_cond_wait(&MyTurn[thread_index], &linewriter);
        pthread_cond_wait(&MyTurn, &linewriter);
    }
    output_mandel_line(1, color_val);
    g_index = index_line;
    pthread_cond_broadcast(&MyTurn);
    // pthread_cond_signal(&MyTurn[(thread_index + 1) % threadcnt]);
    err_ret = pthread_mutex_unlock(&linewriter);
    if (err_ret != 0)
    {
        perror_pthread(err_ret, "unlock problem");
    }
    return;
}

void *start_fn(void *arg)
{
    /*
```

```
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars];
    int thread_index = (__intptr_t)arg;

    for (int index_line = thread_index; index_line < y_chars; index_line += threadcnt)
    {
        compute_mandel_line(index_line, color_val);
        waiting_for_myturn(thread_index, index_line, color_val);
    }
    return NULL;
}

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

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

    struct sigaction act;
    act.sa_handler = sig_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGINT, &act, NULL) == -1)
    {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    pthread_t *threads;
    // MyTurn = (pthread_cond_t *)safe_malloc(threadcnt * sizeof(*MyTurn));
    threads = (pthread_t *)safe_malloc(threadcnt * sizeof(*threads));

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

    // Initialiaze the mutex and the conditional variable

    /*
    for (int i = 0; i < threadcnt; ++i)
    {
        pthread_cond_init(&MyTurn[i], NULL);
    }
    */
    if (pthread_cond_init(&MyTurn, NULL) != 0)
```

```
{
    perror("problem in initializing a conditional variable");
}

/*
 * draw the Mandelbrot Set, one line at a time.
 * Output is sent to file descriptor '1', i.e., standard output.
 */
/* Spawn threadcnt new threads */
for (int i = 0; i < threadcnt; ++i)
{

    ret = pthread_create(&threads[i], NULL, start_fn, (void *)(__intptr_t)i);
    if (ret)
    {
        perror_pthread(ret, "pthread_create");
        exit(1);
    }
}
/*
 * Wait for all threads to terminate
 */
for (int i = 0; i < threadcnt; i++)
{
    ret = pthread_join(threads[i], NULL);
    if (ret)
    {
        perror_pthread(ret, "pthread_join");
        exit(1);
    }
}

pthread_mutex_destroy(&linewriter);
/*
for (int i = 0; i < threadcnt; ++i)
{
   if(pthread_cond_destroy(&MyTurn[i]) !=0)
   {
     perror("problem in destroying a conditional variable");
   }
}
*/
pthread_cond_destroy(&MyTurn);
if (pthread_cond_destroy(&MyTurn) != 0)
{
    perror("problem in destroying a conditional variable");
}
free(threads);
// free(MyTurn);
```

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

Κώδικας 11

Ερωτήσεις

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

Για την υλοποίηση του `mandel_sem.c` χρησιμοποιήσαμε NTHREADS σημαφόρους, δηλαδή έναν σημαφόρο για κάθε thread που δημιουργήθηκε με την `phthread_create(3)`. Έτσι, ο i-οστος σημαφόρος αντιστοιχεί στο i-οστό thread. Ο πρώτος σημαφόρος αρχικοποιείται στην τιμή 1 ώστε να ξεκινήσει την εκτέλεση το πρώτο thread. Στη συνέχεια, κάθε thread με σημαφόρο θετικό μειώνει κατά 1 τον σημαφόρο του με την `sem_wait(1)` και εισέρχεται στο κρίσιμο τμήμα. Κάθε άλλο thread με μηδενικό σεμαφόρο μπλοκάρει σε αυτόν την εκτέλεση του μέχρι να γίνει θετική η τιμή του σεμαφόρου από το αμέσως προηγούμενο thread το οποίο καλεί την `sem_post(1)`.

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.

Διαπιστώνουμε ότι το μηχάνημα μας διαθέτει έξι πυρήνες επομένως επιχειρούμε με 6 threads την παραπάνω σύγκριση.

```
chris@LAPTOP-TK5Q3T95:/mnt/c/Users/koust/ECE/6th Semester/OS/os-lab-exer3/sync$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 25
model           : 80
model name      : AMD Ryzen 5 5600H with Radeon Graphics
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 3293.717
cache size      : 512 KB
physical id     : 0
siblings        : 12
core id         : 0
cpu cores       : 6
```

Για την παράλληλη εκτέλεση του `mandel` εκτελέσιμου πρόεκυψαν τα ακόλουθα αποτελέσματα:

```
time ./mandel
```

```
real    0m0.311s
user    0m0.308s
sys     0m0.000s
```

Έξοδος 2

Για την σειριακή εκτέλεση του `mandel_sem` εκτελέσιμου με 6 νήματα προέκυψαν τα ακόλουθα αποτελέσματα:

```
time ./mandel_sem 6
```

```
real    0m0.070s
user    0m0.322s
sys     0m0.010s
```

Έξοδος 3

Όπως ήταν αναμενόμενο η σειριακή εκτέλεση ήταν πιο αργή.

3. Πόσες μεταβλητές συνθήκης χρησιμοποιήσατε στη δεύτερη εκδοχή του προγράμματος σας; Αν χρησιμοποιηθεί μια μεταβλητή πώς λειτουργεί ο συγχρονισμός και ποιο πρόβλημα επίδοσης υπάρχει;

Στην έκδοση `mandel_cond.c` του προγράμματος ακολουθήθηκαν δύο προσεγγίσεις. Η πρώτη ήταν αντίστοιχη αυτής των σημαφόρων και χρησιμοποιήθηκαν NTHREADS conditional variables δηλαδή μία για κάθε νήμα. Έτσι, στην υλοποίηση χρησιμοποιήσαμε την `pthread_cond_signal(1)` για να ειδοποιούμε κάθε φορά το άμεσα ενδιαφερόμενο νήμα thread[i+1] από τον κόμβο thread[i] να ελέγξει την συνθήκη του while μέσα στο οποίο αναμένει τη σειρά του. Στην δεύτερη προσέγγιση επιχειρήσαμε να χρησιμοποιήσουμε ένα μοναδικό condition variable το οποίο περέμενε καθήλωνε σε ανάμονη όλους τους κόμβους εκτός από τον μοναδικό που εκτελούντας εκείνη τη στιγμή. Στη συνέχεια, ο εκτελούμενος κόμβος καλούσε τη `pthread_cond_broadcast(1)` και ειδοποιούσε όλους τους κόμβους να ελέγξουν αν πρέπει να βγουν από το while loop. Στην πραγματικότητα, η υλοποίηση αυτή παρά την οικονομία χώρου που εξασφάλιζε αφού χρησιμοποιεί λιγότερες μεταβλητές συνθήκης υποβάλει σε ένα περιττό έλεγχο όλα τα threads ενώ γνωρίζουμε εκ των προτέρων ότι μόνο ένα thread θα περάσει τον έλεγχο.

Σημείωση: Στον παραπάνω κώδικα της `mandel_cond` είναι υλοποιημένη η περίπτωση της μίας μεταβλητής συνθήκης. Η άλλη υλοποίηση γίνεται κατανοητή από τα comments.

4. 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;

Στο πρόγραμμα που κατασκευάσαμε είχε τοποθετηθεί εξαρχής η compute_mandel_line() συνάρτηση έξω από το κρίσιμο τμήμα το οποίο περιλάμβανε μόνο την output_mandel_line(). Έτσι, εξασφαλίστηκε ένας παραλληλισμός στον υπολογισμό NTHREAD γραμμών πριν αυτές τυπωθούν σειριακά. Είναι σημαντικό να τονίσουμε ότι με βάση το ζητούμενο μόνο η output_mandel_line έπρεπε να εκτελείται σειριακά για να εξασφαλίζεται ορθότητα στο πρόγραμμα.

5. 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;

Αν πατήσουμε Ctrl-C τότε στέλνουμε ένα Interruption Signal(SIGINT) στο πρόγραμμα και η εκτέλεση τερματίζει πριν την εκτέλεση όλου του περιεχόμενου κώδικα. Έτσι, δεν προλαβαίνει να εκτελεστεί η εντολή `reset_xterm_color(1)` που επαναφέρει στο φυσιολογικό το χρώμα του τερματικού μετα τις διάφορες αλλαγές που υπέστη στην προσπάθεια εκτύπωσης του Mandelbrot Set. Αυτό έχει ως αποτέλεσμα την εκτύπωση της επόμενης γραμμής του τερματικού σε κάποιο ίσως παράξενο χρώμα(βλ. Έξοδος 2)

Έξοδος 4

Για να αποφευχθεί η παραπάνω μάλλον ανεπιθύμητη συμπεριφορά χρησιμοποιούμε το struct `sigaction` της βιβλιοθήκης `<singal.h>` της C. Συγκεκριμένα ορίζουμε κατάλληλα το struct αρχικοποιώντας κατάλληλα τις μεθόδους του όπως φαίνεται στον κώδικα .. και στη συνέχεια ορίζουμε τη συνάρτηση `sig_handler` η οποία θα διαχειριστεί τη λήψη SIGINT όπως εμείς ορίσουμε.

```
struct sigaction act;
act.sa_handler = sig_handler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
if (sigaction(SIGINT, &act, NULL) == -1)
{
    perror("sigaction");
    exit(EXIT_FAILURE);
}
```

Κώδικας 12

```c
void sig_handler(int signum)
{
    printf("   Color will be restored after signal : %d\n", signum);
    reset_xterm_color(1);
    printf("Color restored\n");
    printf("Exiting...\n");
    exit(EXIT_SUCCESS);
}
```

Κώδικας 13

Στη μέθοδο sa_handler ορίζουμε την συνάρτηση που θα χρησιμοποιήσουμε για τον χειρισμό του σήματος και στην sa_mask αρχικοποιούμε το κενό σύνολο σημάτων με την sigemptyset(1). Η sa_mask περιέχει τα σήματα τα οποία μπορεί να μπλοκάρει το αντικείμενο sigaction που έχουμε δημιουργήσει και στα πλαίσια αυτής της άσκησης δεν έχει κάποια αξία. Τέλος, προσθέτουμε τη δομή του struct sigaction όπως εμφανίζεται στα man-pages.

The *sigaction* structure is defined as something like:

```c
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```