ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ
ΥΠΟΛΟΓΙΣΤΩΝ

Λειτουργικά Συστήματα Υπολογιστών
3ο Εργαστήριο

Ομάδα 36
Αναστάσιος Λαγός - el13531
Κωνσταντίνος Βασιλάκης - el16504

Ασκηση 1

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

Καταρχήν παρατηρούμε ότι κατά την μεταγλώττιση του simplesync.c δημιουργούνται 2 εκτελέσιμα, το simplesync-atomic και το simplesync-mutex. Επίσης όταν τρέχουμε οποιαδήποτε απο τις δυο εκδοχές του παίρνουμε ως αποτέλεσμα != 0. Το οποίο είναι λάθος διότι αρχικοποιούμε από το 0 και κάνουμε increment όσες φορές κάνουμε decrement. Αυτό συμβαίνει διότι η εντολή ++*ip; σε assembly μεταφράζεται σε πάνω απο 1 εντολή. Έτσι ο scheduler μπορεί να αποφασίσει να κάνει context switch χωρίς να ολοκληρωθεί πλήρως η εντολή ++*ip, και θα ξεμπλοκάρει άλλα threads που περιμένουν να εκτελέσουν την −−*ip;. Όμως η τιμή της *ip δεν έχει ανανεωθεί ακόμη από την πρώτη εντολή ++*ip πράγμα που οδηγεί σε διαφορετικό αποτέλεσμα.

Μελετήστε πως παράγονται δύο διαφορετικά εκτελέσιμα simplesync-atomic, simplesync-mutex από το ίδιο αρχείο πηγαίου κώδικα simplesync.c

Το Makefile περιέχει το Preprocessor Option: -Dmacro[=defn]. Έτσι με το -DSYNC_ATOMIC κάνουμε define το macro SYNC_ATOMIC, ενώ με το -DSYNC_MUTEX το macro SYNC_MUTEX. Αυτό στον κώδικα μας βάζει σε διαφορετικό branch του if και έτσι κάνουμε χρήση είτε mutexes είτε atomic ops.

Ερωτήσεις

1. Έχουμε τους παρακάτω χρόνους

simplesync $\sim$ 0.05s - NO SYNC
simplesync-atomic $\sim$ 0.53s - SYNC
simplesync-mutex $\sim$ 1.31s - SYNC

Ο χρόνος χωρίς συγχρονισμό είναι σημαντικά μικρότερος απ' ότι με συγχρονισμό.Στην περίπτωση των mutexes τα threads, εκτός απο τις εντολές στο critical sections πρέπει να εκτελέσουν και οποιαδήποτε εντολή έχει σχέση με το mutexing (π.χ. lock/unlock) και επιπλέον μόνο ένα thread μπορεί να εκτελέσει τις εντολές του critical section κάθε φορά. Στην περίπτωση των atomic ops η καθυστέρηση δημιουργείται διότι αυτές οι εντολές παρεμβαίνουν σε θέματα όπως compiler optimization,pipelining.

2. Γρηγορότερη είναι η μέθοδος των atomic ops διότι, αν και όπως αναφέρθηκε παραπάνω προσπερνάει διάφορα optimizations στην εκτέλεση, δεν περιέχει κλειδώματα οπότε όλο το πρόγραμμα εκτελείται συνολικά γρηγορότερα.

3. Από τον κώδικα assembly βρίσκουμε τις παρακάτω εντολές

lock addl $1, (%rbx) για την __sync_add_and_fetch
lock subl $1, (%rbx) για την __sync_sub_and_fetch

Οι εντολές αυτές υποστηρίζονται από τον επεξεργαστή και τρέχουν την πρόσθεση και την αφαίρεση αντίστοιχα ατομικά σαν μία εντολή.

4. Από τον κώδικα assembly βρίσκουμε τις παρακάτω εντολές

call pthread_mutex_lock@PLT
call pthread_mutex_unlock@PLT

Οι εντολές αυτές καλούν τις συναρτήσεις από την βιβλιοθήκη pthread και κάνουν lock και unlock ένα mutex αντίστοιχα.

Κώδικας Άσκησης

simplesync.c

```c
#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 lock;
```

```c
#if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
#error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
#endif

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

void *increase_fn(void *arg)
{
    int i;
    volatile int *ip = arg;

    fprintf(stderr, "About to increase variable %d times\n", N);
    for (i = 0; i < N; i++) {
        if (USE_ATOMIC_OPS) {
            /* ... */
            __sync_add_and_fetch(ip, 1);
            /* You can modify the following line */
            //++(*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) {
            /* ... */
            __sync_sub_and_fetch(ip, 1);
            /* You can modify the following line */
            //--(*ip);
            /* ... */
        } else {
            /* ... */
            pthread_mutex_lock(&lock);
            /* You cannot modify the following line */
            --(*ip);
            pthread_mutex_unlock(&lock);
```

```
76                 /* ... */
77             }
78         }
79         fprintf(stderr, "Done decreasing variable.\n");
80 
81         return NULL;
82 }
83 
84 int main(int argc, char *argv[])
85 {
86         int val, ret, ok;
87         pthread_t t1, t2;
88 
89         if (pthread_mutex_init(&lock, NULL) != 0) {
90             printf("mutex init error\n");
91             exit(1);
92         }
93 
94         /*
95          * Initial value
96          */
97         val = 0;
98 
99         /*
100          * Create threads
101          */
102         ret = pthread_create(&t1, NULL, increase_fn, &val);
103         if (ret) {
104             perror_pthread(ret, "pthread_create");
105             exit(1);
106         }
107         ret = pthread_create(&t2, NULL, decrease_fn, &val);
108         if (ret) {
109             perror_pthread(ret, "pthread_create");
110             exit(1);
111         }
112 
113         /*
114          * Wait for threads to terminate
115          */
116         ret = pthread_join(t1, NULL);
117         if (ret)
118             perror_pthread(ret, "pthread_join");
119         ret = pthread_join(t2, NULL);
120         if (ret)
121             perror_pthread(ret, "pthread_join");
122 
123         /*
124          * Is everything OK?
125          */
126         ok = (val == 0);
127 
128         printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
129 
130         return ok;
131 }
```

Makefile

```
C = gcc

CFLAGS = -Wall -O2 -pthread
LIBS =

all: simplesync-mutex simplesync-atomic

# Simple sync (two version)
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
```

Ασκηση 2

Ερωτήσεις

1. Χρειαζόμαστε σημαφόρους ίσους με τον αριθμό των threads. Σε κάθε thread αντιστοιχεί ένας σημαφόρος. Τα threads κάνουνε sem_wait στον σημαφόρο του προηγούμενο και sem_post στον δικό τους όταν τελειώσουν με το printing της γραμμής που έχουν υπολογίσει. Έτσι μπορούν να αρχίσουν μόνο όταν έχει τελειώσει ο προηγούμενος και έτσι έχουμε συγχρονισμό. Το κρίσιμο μέρος του κώδικα είναι μόνο το μέρος που τυπώνουμε στην έξοδο τις γραμμές, το οποίο πρέπει να γίνει με την σειρά. Το κομμάτι του υπολογισμού μπορεί να γίνει παράλληλα αφού οι γραμμές δεν έχουν κάποια εξάρτηση μεταξύ τους.

2. Σε ένα vm με 2 πυρήνες έχουμε για

N = 1, T $\sim$ 0,32 seconds
N = 2, T $\sim$ 0,22 seconds

Στον orion οι αντίστοιχες τιμές είναι
N = 1, T ∼ 1,20 seconds
N = 2, T ∼ 0,53 seconds
N = 3, T ∼ 0,35 seconds
N = 4, T ∼ 0,27 seconds

3. Το παράλληλο πρόγραμμα εμφανίζει επιτάχυνση μόνο αν θεωρήσουμε ότι το κρίσιμο τμήμα είναι το κομμάτι που κάνουμε print τους χαρακτήρες στην οθόνη. Αν συμπεριλάβουμε και το κομμάτι του υπολογισμού τότε ο συνολικός χρόνος δεν μειώνεται.

4. Πατώντας CTRL + C στο linux στέλνεται στο πρόγραμμα ένα σήμα SIGINT. Μπορούμε να κάνουμε override τον signal handler για αυτό το σήμα και να κάνουμε reset το χρώμα του terminal πριν κάνουμε exit.

Κώδικας Άσκησης

mandel.c

```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 <pthread.h>
#include <errno.h>
#include <semaphore.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

/*
 * 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)

/**************************
 * 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;

struct thread_info {
  pthread_t tid;
  int threadNumber;
  int threadCount;  /*No. threads */
};

sem_t* sems;

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 usage(char *argv0)
{
  fprintf(stderr, "Usage: %s thread_count \n\n"
```

```
    "Exactly one argument required:\n"
    "    thread_count: The number of threads to create.\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;
}

/*
 * 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[])
{
```

```
135   int i;
136
137   char point ='@';
138   char newline='\n';
139
140   for (i = 0; i < x_chars; i++) {
141     /* Set the current color, then output the point */
142     set_xterm_color(fd, color_val[i]);
143     if (write(fd, &point, 1) != 1) {
144       perror("compute_and_output_mandel_line: write point");
145       exit(1);
146     }
147   }
148
149   /* Now that the line is done, output a newline character */
150   if (write(fd, &newline, 1) != 1) {
151     perror("compute_and_output_mandel_line: write newline");
152     exit(1);
153   }
154 }
155
156 void compute_and_output_mandel_line(int fd, int line, struct
       thread_info* threadInfo)
157 {
158   /*
159    * A temporary array, used to hold color values for the line
          being drawn
160    */
161   int color_val[x_chars];
162
163
164   //No synchronization needed for the calculation
165   compute_mandel_line(line, color_val);
166
167   //Synchronization is added when writing to the output
168   int previousSemPosition = (threadInfo->threadNumber + threadInfo
         ->threadCount - 1) % threadInfo->threadCount;
169
170   //Wait if previous thread has not finished
171   sem_wait(&sems[previousSemPosition]);
172   output_mandel_line(fd, color_val);
173   //When finished signal to the next thread
174   sem_post(&sems[threadInfo->threadNumber]);
175 }
176
177
178 void *thread_fn(void *arg) {
179   /* We know arg points to an instance of thread_info_struct */
180   struct thread_info *threadInfo = arg;
181
182   int line;
183   for (line = threadInfo->threadNumber; line < y_chars; line +=
         threadInfo->threadCount) { /*i, i + N, i + 2N, ... */
184     compute_and_output_mandel_line(1, line, threadInfo);
185   }
186
```

```
187   return NULL;
188 }
189
190 int main(int argc, char *argv[])
191 {
192   if (argc != 2)
193     usage(argv[0]);
194
195
196     int threadCount;
197   /*Parsing arguments */
198   if (safe_atoi(argv[1], &threadCount) < 0 || threadCount <= 0) {
199     fprintf(stderr, "‘%s’ is not valid for ‘thread_count’\n", argv
          [1]);
200     exit(1);
201   }
202
203   xstep = (xmax - xmin) / x_chars;
204   ystep = (ymax - ymin) / y_chars;
205
206     struct thread_info *threadInfo;
207     threadInfo = safe_malloc(threadCount * sizeof(*threadInfo));
208     sems = safe_malloc(threadCount * sizeof(sem_t));
209
210   int i;
211   /*Initializing semaphores
212   Semaphore No. = Thread No.
213   all init to 0 except last one.*/
214   for (i = 0; i < threadCount; i++) {
215     sem_init(&sems[i],0,((i==threadCount-1)?1:0));
216   }
217   /*Initializing threadInfo*/
218   for (i = 0; i < threadCount; i++) {
219     threadInfo[i].threadNumber = i;
220     threadInfo[i].threadCount = threadCount;
221
222     /* Spawn new thread */
223     int ret = pthread_create(&threadInfo[i].tid, NULL, thread_fn, &
          threadInfo[i]);
224     if (ret) {
225       perror_pthread(ret, "pthread_create");
226       exit(1);
227     }
228   }
229
230   /*
231    * Wait for all threads to terminate
232    */
233   for (i = 0; i < threadCount; i++) {
234     int ret = pthread_join(threadInfo[i].tid, NULL);
235     if (ret) {
236       perror_pthread(ret, "pthread_join");
237       exit(1);
238     }
239   }
240
```

```
241   reset_xterm_color(1);
242   return 0;
243 }
```

Makefile

```
1  CC = gcc
2
3  # CAUTION: Always use '-pthread' when compiling POSIX threads-based
4  # applications, instead of linking with "-lpthread" directly.
5  CFLAGS = -Wall -O2 -pthread
6  LIBS =
7
8  all: mandel
9
10 # Mandel
11 mandel: mandel-lib.o mandel.o
12   $(CC) $(CFLAGS) -o mandel mandel-lib.o mandel.o $(LIBS)
13
14 mandel-lib.o: mandel-lib.h mandel-lib.c
15   $(CC) $(CFLAGS) -c -o mandel-lib.o mandel-lib.c $(LIBS)
16
17 mandel.o: mandel.c
18   $(CC) $(CFLAGS) -c -o mandel.o mandel.c $(LIBS)
19
20 clean:
21   rm -f *.s *.o mandel
```